Limitations

Gaps and Omissions

Lack of security testing: Even though system is tested for HTTPs and SSL certification. Testing for cross-site scripting, buffer overflows and input validation are essential. More comprehensive testing techniques for security should be implemented such as, fuzz testing.

Lack of testing techniques: Equivalence partitioning could have been used for OrderValidator since we need to account for several invalid labels (as well as a valid label) in order to ensure that the validation is working correctly, and the orders are sorted into the correct category according to the individual order details. However, due to the limited time constraints we were not able to take this into place. As well as that Combinatorial testing could have been done too since, this approach is useful to generate inputs of different kinds in different combinations. This also helps to detect any combinations of inputs that require unusual handling.

Lack of documentation: The tests could be well-documented (given more time-resource). This would allow for an easier testing process and its future alterations, potentially by other testers

Lack of integration test class for the OrderValidator class with the REST server data: The main reason for this omission is that the methods that integrate the data from the REST server with the OrderValidator class have already been tested as part of the unit test for the OrderValidator class. Therefore, this omission again has minimal impact on coverage; however, more tests would again result in more thorough testing and help give us more confidence that the system works as expected.

Potentially insufficient data – OrderValidatorTest synthetic data has been generated manually: Some automation for synthetic data generation could be used. It would require some time to be developed but would then provide a more diverse set of possible inputs and thus a higher degree of confidence in the results of testing.

The primary enhancement in this case would be merely putting these tests into practice, as they would guarantee greater coverage and provide our testing procedure greater depth. The cost-effectiveness of including these tests in our test suite together with all the required synthetic data is the sole negative aspect of this.

Target Coverage/Performance Levels

Functional testing

Tests that are run must pass 100% of the time. This is evident from the fact that the program needs to meet the requirements. Therefore, total test satisfaction guarantees that the functional testing performance target is met because all tests were carefully examined in light of the criteria.

For our code coverage the following target and performance levels are required:

Class coverage aims for the test suite to cover all classes in the system, targeting 100% to ensure each component is tested and functions as expected. This goal is realistic due to the manageable number of classes.

Statement coverage involves the test suite executing most executable statements, aiming ideally for 100%. However, due to resource and time constraints, a practical target is at least 90%, covering most scenarios without exhaustive testing.

Specs coverage targets the test suite to address all requirements in the specifications, aiming for 100%. This ensures the system meets all specified requirements and is achievable without excessive testing, given the reasonable number of requirements.

Test coverage seeks to cover a significant portion of the system's code, including classes and statements. While 100% is ideal, practical considerations suggest a target of at least 90%, balancing thoroughness with resource efficiency and avoiding redundant tests.

How well does the testing meet the target levels

Functional Testing:

We have achieved 100% on all of the tests that has been written so we meet this target level.

Class Coverage:

Our test suite covers 6 of the 6 possible classes so we get 100% on this one too meeting the target level. Except the Node class, which doesn’t do anything.

Statement Coverage:

I was unable to determine a precise figure because IntelliJ lacks built-in capabilities for this measure and there is no practical way to measure it in any other way.

Test Coverage:

We have achieved 95% on test coverage with 95% class coverage and 96% line coverage .Certainly this is greater than our goal of at least 90%.

Specification Coverage:

We have passed all the requirements that stated in the Requirements.pdf. This gives us a specs coverage value of 100%, which meets the target level.

Improvements to achieve target levels

An automated test for the command line input validation could be implemented as a means of reaching the desired levels. We could develop methods inside the test class that have the exact same functionality as the command line input validation methods, and we could utilize artificially generated data to test for different test scenarios. This could be accomplished by using mock functions and mock data. Since our main class only acts as the system's controller, it would be impractical to construct an instance of it. Instead, we implement mock functions to handle validation

inside of our main class. This would undoubtedly enhance our metrics pertaining to test coverage, specifications coverage, statement coverage, and class coverage.

Creating more testing data would be another method to help us meet and perhaps surpass our goals. A mock REST service or an abundance of artificially generated data covering several test scenarios for every method in every class could be used to accomplish this. This would almost probably enable us to surpass the target levels and increase our metrics for test coverage, specifications coverage, statement coverage, and class coverage. But this could result in high test coverage, which can then cause over-testing—developers (me) spending too much time creating and executing tests that yield little to no useful information. This slows down the development process and is a waste of time and resources.