***ILP Testing Process Evaluation***

# **Gaps and omissions**

One of the main gaps/ omissions from the test suite is the ***lack of unit tests for the file writing classes***. These have only been tested as part of the system test since the methods/ classes are only used after all the order deliveries for a day have been made. In order to generate these tests, some synthetic data and simulator would be necessary. This omission does not have a huge impact on coverage, since we still have some piece of code that ensures these classes are functional. However, it would be beneficial to have these additional tests, since it would give us more confidence that the system works as expected.

Another gap/ omission is that of ***lack of integration test class for the LngLat 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 LngLat class have already been tested as part of the unit test for the LngLat 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.

Yet another major gap/ omission is the ***lack of unit tests for the drone class, and hence the path finding algorithm***. The algorithm was only tested at system level due to the large amount of synthetic data that needed to be generated (a potentially separate server with mock data would have been needed, which would have been highly costly both timewise and resource-wise). In addition, some of the testing to ensure the algorithm works as expected was performed manually in parallel with development. Nonetheless, despite some degree of thoroughness in the testing, the additional automated unit test would have also helped increase confidence levels in the functionality of the system.

One other gap/ omission is the ***lack of automated testing for the command line input validation***. The main reason for this omission is that the testing for command line input validation was done manually, and to a thorough enough level to be able to confidently say that the validation works as expected. However, the addition of an automated test class would add more depth to our test suite, and also improve coverage levels.

The last gap/ omission being considered is the ***lack of testing for contents of the JSON/ GeoJSON files***. One reason for this omission is that we assume the data from the REST server has already been pre-validated in a way (no erroneous/ missing information other than the already inputted details on the server). Another reason is the large size of the files, and hence the additional time and resources needed to implement such an exhaustive test. However, having such a test would significantly improve confidence when it comes to the correctness of the file writing.

The main improvement here would be simply implementing these tests since they would add more depth to our testing process and ensure more coverage. The only drawback here is the cost-efficiency of adding these tests (and all the necessary synthetic data) to our test suite.

# **Target levels**

In this section, we will consider the evaluation criteria previously discussed, which are ***class coverage, statement coverage, specs coverage*** and ***test coverage***. For each criterion, we will give a target level that the test suite should ideally meet, together with a justification of why that estimate is adequate.

***Class coverage*** refers to the test suite accounting for a proportion of the classes in the system. An adequate target level for this evaluation criterion would be **100%**. This is because we would want each individual component of our system to be tested as part of our testing process, in order to ensure that all components function accordingly. In doing so, we can be confident that our system works as expected, since all components work as expected. Additionally, since our system does not have a large number of classes, a target of 100% is perfectly attainable without exhaustive testing.

***Statement coverage*** refers to the test suite executing a proportion of the executable statements in the program. Ideally, we would want this value to be 100%, or as close as possible to 100%. However, this might result in exhaustive testing, or a very costly amount of time and resources being spent on achieving this value, which is not ideal. Therefore, in order to ensure a good enough coverage that accounts for most possible cases and does not omit significant/ major faults/ bugs/ outcomes, an adequate target level for this evaluation criterion would be **at least 90%**.

***Specs coverage*** refers to the test suite accounting for a proportion of the requirements in the requirements specification. An adequate target level for this evaluation criterion would be **100%**. This is because we would want to ensure that our system fulfils all requirements outlined in the requirement specification. By having our testing process guarantee that all requirements have been accounted and tested for, we can be confident that the system works according to the specification, and therefore as expected. Additionally, the number of requirements for our system is also not extremely large, and so a target of 100% is certainly attainable through thorough, but not exhaustive, testing.

***Test coverage*** refers to the test suite covering a good proportion of a proportion of the code for the system (classes and executable statements). Ideally, we would want this value to be 100%, or as close as possible to 100%. However, this might result in exhaustive testing, or a very costly amount of time and resources being spent on achieving this value, which is not ideal. Moreover, some of the tests could end up being repetitive in order to ensure enough coverage for both classes and statements; this is far from ideal, since it is very costly in terms of time and resources and does not add any depth to the test suite. Therefore, in order to ensure a good enough coverage that accounts for most possible cases and components, and avoids any major omissions, an adequate target level for this evaluation criterion would be **at least 90%**.

# **How well does the testing meet the target levels**

In this section, we will consider the evaluation criteria previously discussed, which are ***class coverage, statement coverage, specs coverage*** and ***test coverage***. For each criterion, we will give the actual value achieved by our test suite, and how that compares to the target levels previously set.

For ***class coverage***, our test suite covers 14 of the 15 possible classes, giving a class coverage value of 93.33%. This is, of course, lower than our target value of 100%.

For ***specs coverage***, our test suite covers 20 of the 23 requirements outlined in the requirements specification document. This gives us a specs coverage value of 86.96%, which falls below our target level of 100%.

For ***test coverage***, using the in-built tools that IntelliJ gives us, we obtained a value of around 84% for our test suite. This, of course, falls below our target level of minimum 90%.

For ***statement coverage***, since IntelliJ does not have any in-built tools for this measure, and there is no realistic means of measuring it otherwise, I was unable to obtain a specific value. However, a rough estimate for the actual value would be about the same as our test coverage value, which yet again falls below our target level of minimum 90%.

However, these differences can be justified by the manual testing performed to ensure the one omitted class works as expected. Nonetheless, since the manual testing was extensive and covered all possible scenarios, we can say that these differences can be labelled as small but justifiable discrepancies between our target and actual values for the evaluation criteria.

# **Improvements to achieve/ exceed target levels**

One way to achieve the target levels would be ***to implement some automated test for the command line input validation***. This could be done through the use of ***mock functions*** and ***mock data***: we could create methods inside the test class with the exact same functionality as the command line input validation methods and use synthetically generated data to test for multiple test cases. The reason for creating mock functions is that the validation is done inside our *App* (main) class, which means that it would be impractical to create an instance of this class, as it simply serves as a controller class for the system. This would certainly improve our measures for ***class coverage***, ***specs coverage*** and ***statement coverage***, and also in theory for ***test coverage*** (I say in theory because, although the in-built IntelliJ tools might not recognize that the new test adds more coverage, it in fact does by covering the methods inside the *App* class).

Another way that could help us achieve and even exceed our target levels would be to ***generate more testing data***. This could be done through having a mock REST server, or just through plenty of synthetically generated data that covers multiple test cases for each method in each class. This would certainly improve our measures for ***class coverage***, ***specs coverage***, ***statement coverage***, and ***test coverage***, and almost certainly allow us to exceed the target levels. However, this could result in exhaustive testing, which would be very inefficient both timewise and resource-wise.