Outline of the Software Being Tested

The software being tested is the 2022 Informatics Large Practical coursework, which sees students complete a Java project on a simulated drone delivery system. In this system, it must pull relevant data from REST servers, validate them, and calculate an optimal flightpath for a drone to collect and deliver as many orders as possible to Appleton Tower by avoiding designated no-fly zones. Finally, once the drone runs out of battery, the system must then output three separate files detailing the drone's flightpath and information regarding all the valid and invalid orders for that day.

GitHub Repo: https://github.com/DHJKim321/Software_Testing_Portfolio

Learning Outcomes

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

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

I have achieved this sub-learning outcome by considering functional and non-functional requirements. For functional requirements, I split them into different modules: Order validation, navigational, and file writing. Regarding the non-functional requirements, I have used the specification in the ILP document on the execution time, as well as elaborated on others such as the drone needing to fly above buildings and the system needing to be secure. For the full document see 'Requirements Document' in folder 'LO1'.

1.2 Level of requirements, system, integration, unit.

I have achieved this sub-learning outcome by looking at the functional requirements laid out in the previous sub-learning outcome and grouping them into different classes such that each class performs a specific task. For more detail see 'Levels of Requirements' document in folder 'LO1'.

1.3 Identifying test approach for chosen attributes.

I have achieved this sub-learning outcome by considering the different types of test approaches: black-box and white-box testing. In the document 'Testing Approach and Evaluation' in folder 'LO1', I delve into more detail about the specific kinds of tests that I used.

1.4 Assess the appropriateness of your chosen testing approach.

I have achieved this sub-learning outcome by talking about the limitations that I faced while developing the tests for this system. Due to almost finishing ILP before starting this coursework, it meant that I was not able to follow testing procedures as much as I wanted to. To see the further limitations, as well as the conclusion on how my testing approach is still valid, see 'Testing Approach and Evaluation' in folder 'LO1'.

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

2.1 Construction of the test plan.

2.2 Evaluation of the quality of the test plan.

2.3 Instrumentation of the code.

2.4 Evaluation of the instrumentation.

I have decided to write about this entire learning outcome at once, as they all can be found in the 'Test Planning Document' in folder 'LO2'. First, I researched different professional test plans, as well as the template one provided, to see what additional aspects of the test planning phase industry leaders talked about. The trend seemed to be that there were mentions of the scope of the system, as well as what was outside it, talking about the expectation of what my system might do and what it might not. Furthermore, I have also added an 'Assumptions/Managing expectations' section which describes an ideal testing condition (Multiple developers, following testing lifecycles) and why some of these were not met.

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.

I have achieved sub-learning outcomes 3.1, 3.2, and 3.4 by considering the different testing techniques and coverage options that I mentioned in learning outcome 2. To decide on each method for the requirements that I am testing, I looked at the classes they required and their hierarchy.

R1 uses category partition testing for the pathfinding algorithm.

R2 uses category partition and performance testing to vary the OS and test its performance.

R3 uses structural, pairwise, and category partition testing for the order validation.

To find their coverages, I used Intellij's coverage calculator, as that was the most straightforward tool to use and came integrated with Intellij. This is a type of line coverage and was useful in showing me what configuration of certain classes I was missing. See 'Testing Document' in folder 'LO3' for more details.

3.2 Evaluation criteria for the adequacy of the testing.

See 'Testing Document' in folder 'LO3'.

3.3 Results of testing.

I have achieved this sub-learning outcome by running the tests that I have written and inserting screenshots of all the passed tests, as well as describing which ones failed, into the 'Testing Log' document in folder 'LO3'.

3.4 Evaluation of the results.

This sub-learning outcome was extremely similar to that of 4.3. Thus, I have merged this into a document I will talk about in the next learning outcome. See 'System Evaluation Activities' in folder 'LO4'.

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.

I have merged the four sub-learning outcomes into one, as they are all present in the same document 'System Evaluation Activities' in folder 'LO4'. In this document, I talk about the gaps and omissions of my testing, such as not having enough variety in the testing data that I either generated or used from the provided REST servers. This meant that not only could I have missed certain configurations of attributes, but also my performance could not be fully representative of a real-life scenario, and only give ideal readings.

With regards to the evaluation activities themselves, as I mentioned previously, I used Intellij's coverage calculator to find out the percentage of lines covered by the test suites. The result was that three out of four classes exceeded 80%, with the rest being methods such as *toString* and *hashcode*. One exception to this was the class *Polygon*, which saw a 67% coverage, due to not testing its constructors which take up some of the lines in that class. Regardless, the coverage that I got still ensures that most relevant methods and its respective edge cases of each class were tested and verified.

I outlined some ways in which my actual calculated levels might match the target levels. A suggestion was to use a faster algorithm for visibility graph generation, which will see an improvement of the upper bound from $O(n^3)$ to $O(n^2 \log n)$. Another way to meet the target levels would be to include a more varied dataset to run my test suites against, but due to time and resource constraints, I was unable to do so.

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.

For this sub-learning outcome, I have laid out the different code review techniques, such as coding conventions, how consistent my coding style is, and if there are enough comments. Through this analysis, I have found that there could be more comments explaining further what some methods do, but the use of Javadoc also helps alleviate some of the pressure. Moreover, my coding style seems to be consistent across the board due to using Intellij's code formatter, which ensures that all code that I have written will be structured in the same way. A more detailed analysis is available in the 'Code Review Activities' document.

5.2 Construct an appropriate CI pipeline for the software.
5.3 Automate some aspects of the testing.
5.4 Demonstrate the CI pipeline functions as expected.

Since these three learning outcomes all deal with designing a continuous integration pipeline, I have decided to merge them. To come up with its design I researched some different websites on Google and settled on the following: My pipeline will have four phases, as is the convention, composed of 'Source', 'Build', 'Test', and 'Deploy'. The document 'CI Pipeline Design' in folder 'LO5' contains the descriptions of what each phase would do and what kinds of software it would use to fulfil those criteria. Furthermore, I have outlined the different issues that might crop up when using these software artefacts, and what should be done to avoid or fix them.

To automate some aspects of the testing, the tests that I have written for the three requirements laid out in the 'Test Planning Document' will be uploaded to the pipeline such that whenever a new change is committed into a branch, these tests will run automatically. Thus, developers will be able to check that the change that they have implemented has not broken any previous code.

Finally, I have outlined the step-by-step process of what would happen when a change is committed, and what kinds of errors would be caught.