Software Testing 2022 Portfolio

Outline of the Software Being Tested

The software used for testing in this course is a drone-based pizza delivery application, originating from another course called Informatics Large Practical. It retrieves pizza orders from a remote server, checks the validity of the orders, creates a drone flight path using a greedy pathfinding algorithm that avoids no-fly zones, and generates flight results and completed orders in three separate files. The project and all complementary files can be found [HERE](https://github.com/BakerMoney/Informatics-Large-Practical).

Learning Outcomes

1. Analyse requirements to determine appropriate testing strategies [25%]
   1. *Range of requirements, functional requirements, measurable quality attributes, qualitative requirements…*

The full list of requirements can be found in the **Requirements.docx** file. It contains various types of functional requirements, which cover different measurable attributes. A group of requirements that are each different from each other could consist of requirements 1 (correctness), 11(liveness), 12 (performance), 9 (efficiency and fairness) and 8 (privacy and safety).

* 1. Level of requirements, system, integration, unit.

Requirement 2 can be tested at the unit level as it can be a single isolated method that takes coordinates and returns either True or False. Following this, an integration level test can be performed on requirement 6, where we test the program’s capability to both retrieve coordinates from a remote server and check if the drone is within the polygon that is made up of the retrieved coordinates. Requirement 8 would be considered system level because we would need to run the whole program to compute the flight path and then check the generated files if the drone entered a no-fly zone.

* 1. Identifying test approach for chosen attributes

The attributes named for this project can be tested in various ways. These approaches could be iterative testing, where every iteration of the system is thoroughly tested, proactive testing, where the tests are made as the system is being built, or reactive testing, where the tests are made after the whole system is finished. DevOps continuous testing can also be considered, but makes little sense to use for the ILP project due to the project’s limited scalability.

* 1. Assess the appropriateness of your chosen testing approach

I believe that the best testing approach for this project is proactive testing since the project is of significant size. In my opinion, this testing approach helps save the most time, as writing and performing tests after having built the system (reactive testing) for a project of greater size will take up more time digging through code than just performing the testing and clearing out the bugs at the early steps of development. In a corporate environment, this would also help save money and human resources.

1. Design and implement comprehensive test plans with instrumented code [default 15%]
   1. Construction of the test plan

The project plan is in the planning document named *PlanningDocument.docx* and details all the necessary topics like the requirements considered, priority and pre-requisites, scaffolding and instrumentation, and process and risk.

* 1. Evaluation of the quality of the test plan

The plan itself is considered to be optimistic, and in the “process and risk” section in the *PlanningDocument.docx* document it is explained what risks the selected requirements impose and how those risks can be dealt with in more depth. Some of the risks included are time shortage and synthetic data not being able to cover all cases.

* 1. Instrumentation of the code

For a comprehensive understanding of the code's instrumentation, refer to the section titled "Scaffolding and Instrumentation" in the *PlanningDocument.docx* document. The section covers scaffolding and creating simulators, related to the specified requirements.

* 1. Evaluation of the instrumentation

A more in-depth evaluation can be found in the "Evaluation of Instrumentation" section of the *PlanningDocument.docx* document. This section discusses limitations such as the accuracy of synthetic data and how reliable it is, the extent to which the test suite can validate the requirements, as well as other risks and costs.

1. Apply a wide variety of testing techniques and compute test coverage and yield according to a variety of criteria [default 15%]
   1. Range of techniques

Testing was performed on all levels and the types of tests include systematic functional testing, where order placement testing was done by testing the different types of inputs separately, boundary value analysis, where an edge case of the credit card number is tested, equivalence partitioning, where the entire system was tested in separate parts to reduce the amount of tests, and also performance testing of a measurable attribute, where it was measured if some minimum amount of orders get delivered each day. The evidence (tests) can be found in the ‘test’ folder, which is in the ‘src’ folder.

* 1. Evaluation criteria for the adequacy of the testing

The testing that was performed is mainly optimistic because some of the testing had to be manual due to the size of the testable material. For example, the contents of the drone files produced after running the system needed manual checks because it would simply take too long to run tests on the contents. The test coverage criterion justifies the use of boundary value analysis and equivalence partitioning because with these techniques the system can be tested more expansively, also the class coverage criterion justifies systematic functional testing, as it allows testing individual classes separately.

* 1. Results of testing

One of the issues that surfaced when I started testing the code was the validation of the credit card number being wrong. The original code disregarded the edge case of having all zeroes for the number, so after implementing a simple if statement the issue was fixed and functionality was improved. Another issue that I found was not all orders being retrieved. After a slight modification to a for loop the issue was resolved and all 47 orders for a day were retrievable. There was also the issue of a failing efficiency test, but after adjusting how the drone flies back from the restaurant I was able to make the algorithm more efficient and make the tests pass.

* 1. Evaluation of the results

The class coverage criterion checks that a significant portion of the classes in a system have been tested, verifying the design and functionality of all components, reducing the risk of bugs, and increasing confidence in the system's expected behaviour. The test coverage criterion guarantees that a significant part of the code is covered by tests, evaluating various scenarios and comparing their outcomes to the expected behaviour. This helps identify and fix faults in the code, and also increases confidence in the system by considering most possible outcomes.

1. Evaluate the limitations of a given testing process, using statistical methods where appropriate, and summarise outcomes. [default 20%]
   1. Identifying gaps and omissions in the testing process

One of the most notable gaps/omissions in my testing process are lack of tests for some of the classes. This is the case because of the limited hours we have for this course, but ideally all of the classes should have tests to improve coverage and increase confidence levels. Another one is lack of tests for the contents of the generated files. This is because the input data is provided by a credible source (the school of informatics), so we can assume that whatever gets generated by the system won’t need tests. Additionally, due to the massive size of the contents, performing the tests would take a long time, but ideally, they should be there to once again improve confidence in the system.

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

In terms of class coverage, a target of 100% is fully achievable for a project of this size and would not use too many resources or lead to exhaustive testing. As for test coverage, the ideal amount would most likely be between 85% and 90%, because trying to achieve more than that would likely lead to diminishing returns and a suboptimal usage of test development resources and time.

* 1. Discussing how the testing carried out compares with the target levels

A class coverage of 91% and test coverage of around 88% was achieved for this project when measured with provided IntelliJ tools, however, it should be mentioned that these values are somewhat inflated because of the system-level tests that were run on the program within a certain class. When we compare the results to the ideal coverage numbers, class coverage comes slightly below the value of 100%, but test coverage passes the bare minimum requirement of 85%, which would be considered quite good in a production setting.

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

To be able to actually achieve the target levels, in this specific case it would help if more tests were created for each individual method. This would greatly improve test and class coverage, alongside even spec coverage. Additionally, if more effort was put into testing edge cases and the command line input, test coverage would improve yet again without the concern of exhaustive testing or suboptimal resource use.

1. Conduct reviews, inspections, and design and implement automated testing processes. [default 25%]
   1. Identify and apply review criteria to selected parts of the code and identify issues in the code.

A checklist review approach was selected to review the code. For example, in the class *Drone.java* it was identified that there is sufficient commenting of methods, however the comment for *getValidOrdersByDistance* did not explain the method well enough and was adjusted. The method *computeTicks* had an incorrect access modifier, which was later set to private. All other access modifiers were correct, imports are handled correctly and code formatting follows java coding conventions. It should be noted that a pair-programming review approach also exists, but was not suitable for this project as we were only allowed to work on it individually.

* 1. Construct an appropriate CI pipeline for the software

To build an appropriate CI pipeline you could set up a version control system through GitHub, find an appropriate CI tool like Jenkins (it’s popular and works well with many systems) and link it with the VCS, create a job with your code repository, build triggers and build steps, define the build steps( commands and scripts that Jenkins should run such as "mvn clean install" or "npm install"), set up the build trigger so that it builds when changes are pushed, and finally tell Jenkins to what target environment it should deploy the artifacts.

* 1. Automate some aspects of the testing

Ideally, we would incorporate all of our current tests into the CI pipeline, but our primary focus should be on the tests that confirm safety requirements. Furthermore, it's essential to prioritize a mix of unit, integration, and system tests, including those that evaluate performance, functionality, and fairness, as part of the pipeline. This will result in more comprehensive testing.

* 1. Demonstrate the CI pipeline functions as expected.

The proposed CI pipeline would be able to identify failed tests, where the failure could be addressed by simply looking at what kind of testing was performed by those failed tests, coding convention violations, which could be fixed using *Checkstyle*, and syntax errors, which should be easily fixable by analysing the lines that failed execution. It could also identify dependency conflicts in the case that the code was changed significantly between versions. This issue will be detected by Maven and could be fixed my looking at the POM file and adding, removing or modifying dependencies.