Outline of the Software Being Tested

In this portfolio, I will be testing the Informatics Large Practical coursework from 2024 which involved building a pizza drone delivery system in Java for the University of Edinburgh. The system receives and validates orders submitted by clients, determines whether an order is valid, and generates a flight path from a restaurant to Appleton Tower while adhering to navigational and geometrical constraints such as no-fly zones. Throughout my portfolio I have assigned each requirement and test case a unique Test ID to improve traceability between the documents.

Link to GitHub repository: https://github.com/s2322980/Software-Testing-Portfolio/tree/main

Learning Outcomes

To facilitate marking I've included in brackets in which of the documents the evidence of each sub-learning objective can be found.

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

1.1 Range of requirements, functional requirements, measurable quality attributes, qualitative requirements, … (Range of requirements)

I achieved this sub-learning outcome by considering the testable functional requirements, measurable attributes, and non-functional requirements. Within the functional requirements I considered the navigational constraints of drone movement as well as the requirements for order validation. The measurable attributes entailed correct error code and HTTP status handling as well as considering execution time of the system. Other non-functional qualitative requirements such as data security and privacy, drone availability and other non-testable requirements were outlined.

1.2 Level of requirements, system, integration, unit. (Level of requirements)

In the **Level of Requirements** document I grouped every requirement laid out in the **Range of Requirements** document by level of testing: unit, integration, and system. In addition to those requirements, I considered the different classes and methods within my code which would need to be tested.

1.3 Identifying test approach for chosen attributes. (Range of requirements)

For every requirement I outlined originally in **Range of Requirements** I evaluated the test implementation most suited. This included both black-box and white-box testing.

1.4 Assess the appropriateness of your chosen testing approach. (Test Approach)

In my **Test Approach** document, I evaluated the appropriateness of chosen level of testing for the requirements and went into detail about my reasoning for my chosen approach. As I had already finished my ILP coursework before starting testing this influenced my approach and prevented me from following typical testing procedures.

2. Design and implement comprehensive test plans with instrumented code [default 20%] (Test Planning)

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

For this learning requirement I outlined my construction and evaluation of my test plan and instrumentation within the **Test Planning** document where I evaluated and prioritised the most crucial requirements of the system in order to allocate the limited resources I have to testing. I outline 3 fundamental requirements of the system and evaluate their risk and lifecycles. I outline the scaffolding and instrumentation I applied to my code.

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 (Testing Document)
3.2 Evaluation criteria for the adequacy of the testing (Testing Document)

For sub-learning outcomes 3.1 and 3.2 in the **Testing Document** I outline the testing techniques I have chosen to apply to each of the main requirements I considered in LO2 and evaluated the adequacy of the techniques I chose.

3.3 Results of testing (Test Cases)
3.4 Evaluation of the results (Test Cases)

In the **Test Cases** document, I go into detail about every individual test that is needed to ensure an optimal test suit coverage of each requirement. I consider valid, invalid and boundary edge cases for each where appropriate. I then evaluated the result of implementing these test cases and discuss the issues that arose and how I fixed them.

4. Evaluate the limitations of a given testing process, using statistical methods where appropriate, and summarise outcomes. [default 20%] (Limitations)

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.

This learning outcome's evidence is all within the **Limitations** document in the respective LO4 folder. The primary limitation in my testing process is the continued need for the manual visual inspection of generated flight paths via the GeoJson website in order to ascertain whether the no-fly zone and central area constraints were satisfied.

I used IntelliJ's in-built coverage calculator to find the test coverage of my testing suites and evaluated those percentages to determine if they were reasonable. I used the execution time on the tests to estimate performance levels and then compared those as well as the previous test coverage levels to the target levels I outlined. Functional requirements achieved full coverage and structural coverage for

core classes was high. Performance targets were met for most components, although first-run flight path generation exceeded targets due to algorithmic complexity and framework overhead.

Lastly, I considered what further test approaches and resources would be necessary in order to meet the target levels for every requirement.

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. (Code Review)

For this sub-learning outcome, I focused my code review on the order validation method in my controller as it was the most error prone part of my system from the complexity of some of the requirements and from the unreadable and repetitive way I had initially designed it. I applied a checklist-based inspection approach, which is well suited to validation logic where each rule was designed to be independently due to the nature of the ILP auto-marker. The review criteria focused on readability, redundancy, correctness, boundary handling, and null safety. During this review, I identified several issues including duplicated validation logic, and inconsistent error handling, and incorrect assumptions about credit card expiry dates and order pricing. Many of these issues had already been exposed through testing in earlier learning outcomes, primarily in LO3. The way in which I refactored the method to improve clarity and correctness as well as a comparison to the prior version of the method is in the **Code Review** document. This review process helped improve maintainability and reduced the likelihood of subtle validation bugs.

5.2 Construct an appropriate CI pipeline for the software (CI Pipeline)

I designed an appropriate CI pipeline for this project in my **CI Pipeline** document. The proposed pipeline consists of the stages Source, Build, Test, and Feedback, and would be triggered automatically on every push or pull request to the GitHub repository. I left out the Deploy stage out of my pipeline as it is only required in a CI/CD pipeline and deployment is out of scope for the ILP coursework. However, the design of the pipeline allows for it to be extended into a full CD/CI pipeline if required. In the build stage, the system would be compiled using Maven and packaged into a Docker image, allowing configuration or dependency errors to be detected early and automated unit tests would be run to ensure the pipeline follows the principle of ‘fail fast’. This also ensures tests are run in a bottom-up manner. The test stage would then execute all automated integration tests to validate functional correctness and REST behaviour.

5.3 Automate some aspects of the testing (CI Pipeline)

The majority of the testing I developed for this project is suitable for automation and would be embedded directly into the CI pipeline. This includes unit tests for drone movement and geometric calculations, integration tests for order validation, as well as REST-level tests for endpoint behaviour and response structure. These tests can be executed automatically without manual intervention and provide fast feedback on correctness after each code change. System-level tests involving visual inspection of

GeoJSON flight paths were intentionally excluded from automation, as they require a developer to manually inspect flight paths through GeoJSON to assess spatial legality.

5.4 Demonstrate the CI pipeline functions as expected. (CI Pipeline)

The proposed CI pipeline would be effective at identifying a range of issues. Incorrect validation logic, such as missing delivery fees, which would be detected through failing integration tests. Compilation errors, dependency misconfigurations, or malformed Docker images would be caught in the build stage before tests are executed. Reductions in test coverage or unexpected performance regressions would be visible in the feedback reports generated after test execution. By failing fast and providing immediate feedback, the pipeline ensures that errors and defects are identified early and prevents faulty code from being merged into the main branch.