Learning Objective 2

Design and Implement Comprehensive Test Plans with Instrumented Code

The Test Plan

This is the first test plan for the PizzaDronz project and Software Testing submission. It will incorporate the requirements highlighted in the LO1 document (link or something) to outline the testing strategy that will be implemented into PizzaDronz. It will be created in line with a Test-Driven Development process. This document will outline the process and techniques involved, how the aforementioned requirements will be incorporated into the strategy, any necessary scaffolding or code instrumentation.
Time constraints will limit the number of requirements that can be covered and the techniques that can be used, and so they will be prioritised and have their methodologies summarised in this document.

Requirements

I will be testing 6 requirements and their corresponding unit requirements where appropriate in my test suite. At a high level, these are:

- **R1 -** The program will take 2 arguments at execution. These will be the date to retrieve orders (in ISO 8601 format, yyyy-MM-dd) and the URL of the REST server (REST API).
- **R2 -** The program will read and then use data from the REST API.
- **R3 -** The system will check all orders from the supplied date for validation.
- **R4 -** The system will have functions in place for the navigation of the drone.
- **R5 -** For each day the drone operates, the system will generate three JSON files as an output.
- **R6 -** The system must execute and produce the output files in under 60 seconds.

These make up a mixture of Functional and Non-Functional requirements, incorporating multiple levels of requirements to thoroughly test the PizzaDronz system. It is important to prioritise the testing of these requirements as they correspond to different levels of functionality within the overall program.

Priority

- **R4** and **R3** – These make up the core functions of the system; without these, nothing could be done with the data retrieved from the server. They therefore should be implemented and tested first, as they make up the concrete foundations of PizzaDronz. They should be tested at the unit level first, and then together at respective their system levels. To save time, it would be sensible to develop and test these requirements simultaneously, as they are of equal importance to the system.
- **R2** – Without interacting with the REST API, the PizzaDronz system would have no data to work with and would therefore be rendered useless. As something needs to be done with the data, it is ranked as second as those tasks must be completed first. However, it's then imperative to achieve functioning integration with between PizzaDronz and the REST API.

- **R1** – This function ties the previous three requirements together and will allow for testing most of the system together. It would not make sense to implement and test this requirement before the others, as there would be nothing for this component of the system to work with. However, it is a core function of the system and is the method in which PizzaDronz is executed by the user, so it must therefore be functionally complete before deployment. This is also a robustness requirement, which should take a higher priority as it's fundamental for the proper operation of the system.
- **R5** – The output of the system is one of the most important factors of the PizzaDronz system, however it physically could not be implemented before the other requirements are met, as they are what make up the functionality necessary for producing the output files. Therefore, this requirement will be assessed later in development, as one of the last functions to be implemented.
- **R6** – Non-Functional requirements are given less priority as they cannot be tested on an incomplete system, especially those corresponding to the performance of the system. This requirement is not concerned with the unit requirements that make up PizzaDronz, and therefore should not be a main concern through the earlier stages of development.

Test Plan – A&T, V&V

The plan below is document in order of requirement priority, as discussed above. For each requirement, I will detail what is to be tested, and what must be implemented and verified for each requirement to pass and be deemed functioning and fully implemented.

R4: The system will have functions in place for the navigation of the drone.

- This is a vital requirement that makes up one of the two core functions of PizzaDronz. It defines how the drone will navigate between delivery points and restaurants, and therefore how flightpaths will be calculated and mapped later on.
- This requirement will first be tested at the unit level, verifying that each individual function is working as expected. It will then be tested as a whole system.
- To meet this requirement, all unit tests must pass, as well as work together to create valid flightpaths.
- The system must be capable of successfully calculating the following:
    - The distance in degrees between two LngLat points
    - Whether or not two LngLat points are 'close', defined as 0.00015 degrees or less apart.
    - If a LngLat point is within a NamedRegion, defined as a name and a set of LngLat coordinates created a closed polygon.
    - What the next position of a drone will be after a single move, given a starting position and an angle that is a multiple of 22.5 (giving 16 possible directions of movement).
        - Or if the drone is hovering, given an angle of 999.
- These tests can be carried out by manually generating values (primarily coordinates) within the test suite and applying them to the implemented functions.

R3: The system will check all orders from the supplied date for validation.

- This is a vital requirement that makes up the second of the two core functions of PizzaDronz. It checks every order retrieved from the REST API and assesses them for validity through several functions.
- This requirement will first be tested at the unit level, verifying that each individual function is working as expected. It will then be tested as a whole system.
- To meet this requirement, all unit tests must pass, as well as work together to verify orders and identify errors.
- The system must be capable of successfully verifying the following:
    - Credit Card details
        - Card Numbers:
            - 16 digits long
            - Numerical only
        - Expiry Dates:
            - MM/yy format (numerical)
            - Valid date, in the future (current month/year acceptable)
        - CVV:
            - 3 digits long
            - Numerical only
    - Cost of order
        - Must be equivalent to total cost of pizzas in order, plus delivery fee of 100 pence
    - Size of order
        - No greater than 4 pizzas in an order
        - Impossible to order less than 1 pizza, so no check necessary
    - Valid pizzas in order
        - Each pizza ordered must exist on at least one restaurant menu
    - All pizzas ordered come from the same restaurant
        - Two different pizzas in the same order cannot come from more than one restaurant, they must all be available from the same menu
    - If the restaurant is open on the day the order was placed
        - Doesn't matter if restaurant is closed 'today', as long as it was open on the date somebody placed the order
    - All values must be non-null!
- These tests can be carried out by manually generating random restaurants and card details at the unit level, then by generating entire orders for system level testing.

Thorough testing and implementations of **R3** and **R4** can ensure that errors are caught and handled early on in development, so as not to cause delays and more major problems at later dates.

R2: The program will read and then use data from the REST API.

- This requirement is fundamental for proper use of the PizzaDronz system. The REST server stores all of the real data to be used for orders, restaurants and no-fly zones
- This requirement demonstrates the integration between the PizzaDronz system, and the REST API. It can be tested at the unit level for specific data, but must also be

tested at the integration level to ensure proper communication between the systems.

- To pass the tests, all unit tests must return true (or valid data), but the program must also be able to successfully retrieve all of the data at once and perform the necessary functions from R3 and R4 on it
- The system must be capable of the following for this requirement to be met:
    - Recognise that the server is 'alive'
        - Retrieves a value of True from a specific URL path
    - Obtain Restaurant data for the real restaurants
        - This includes name, LngLat coordinates, a menu of pizzas available at this restaurant and their prices in pence, and a list of opening days
    - Obtain No-Fly Zone data
        - This includes name and LngLat coordinates to form a closed polygon
    - Obtain Central Area data
        - This includes name and LngLat coordinates to form a closed polygon
    - Order data for the given date
        - This will be a large number of orders, which comprise of:
            - Order number
            - Order Date
            - Order Status
            - Order Validation Code
            - Credit Card Information
            - Pizzas in order
            - Total cost of order
        - This data will then be utilised by OrderValidator.java
- These tests can be done at the unit level for individual data points, and all at once.
- Tests can also be carried out for further integration testing with OrderValidator.java

R1: The program will take 2 arguments at execution. These will be the date to retrieve orders (in ISO 8601 format, yyyy-MM-dd) and the URL of the REST server (REST API).

- This requirement defines the interface with which a user will interact with PizzaDronz. It is therefore important to thoroughly test this for correct functionality and robustness.
- To meet this requirement, the system should accept two and only two arguments, in the following order:
    - Date to retrieve orders (ISO 8601 format; yyyy-MM-dd)
    - URL of the REST server
- The system should detect errors and alert the user to them, should there be something preventing the system from running as intended.
- Testing should comprise of wrong inputs, different order to inputs and invalid inputs of numerous combinations.
    - Mutation testing would be of benefit here, time permitting

R5: For each day the drone operates, the system will generate three JSON files as an output.

- These files are the measurable output for the PizzaDronz project, and therefore it's of great importance that their contents are all present and correct.

- This is more difficult to test to a finer detail, however the existence and fundamental contents of the files can easily be tested for and verified.
- However, it is not time efficient to dedicate significant resources to ensuring that the files are correct word-for-word. Inspection will have to suffice.
- drone-yyyy-MM-dd.geojson must also be readable by an online geojson interpreter, for this requirement to be met

R6: The system must execute and produce the output files in under 60 seconds.

- This is a non-functional performance requirement. Whilst it is important that it's met for the proper and efficient running of PizzaDronz, it is less of a priority.
- It can easily be tested by logging the execution runtime.
- This can be repeated multiple times, and an average taken, to ensure that the program regularly runs at an expected rate

Before testing can be completed, the tests must be written. This is the first step within each requirement's process, to make sure that tests cases are drawn up for as many scenarios as possible. This will increase the test coverage, and decrease the likelihood of errors once moved on from a requirement. Tests will be carried out in the above order. Where appropriate, Unit tests will be prioritised within a requirement's testing. Where necessary, changes to code will be implemented for any errors identified through testing, before re-running the tests once more. This will be repeated for each level of requirement. Integration level tests will then be carried out, and finally System tests once all the lower-level tests have passed and verified the functionality of each requirement's individual components.

Code Instrumentation and Scaffolding

Code instrumentation and scaffolding can be very beneficial to both the development and testing processes. Code instrumentation can be utilised to help diagnose problems, and often is found in the form of writing to an output to define what has happened. Code scaffolding is more heavily used in the testing process, and here will be used to produce random sample data for orders or restaurants, for example. Here I will cover what will be included for each requirement.

R4: Scaffolding will be used in the form of generating specific mock-up coordinates for each test case involving the drone's navigation functions.
R3: Scaffolding will be used in a similar way to **R4**, to randomly generate (valid) orders, restaurants, pizzas and card information for each test case. This will be used in both unit level tests, and all together in system tests.
R2: The REST server/API will be used to verify the interaction between PizzaDronz and the server, and to obtain real data from somewhere.
R1: Instrumentation will be used to verify the outputs the system should produce from invalid arguments, primarily using assertions or System.error.println, for example
R5: No scaffolding or instrumentation
R6: No scaffolding or instrumentation