Learning Objective 1

Analyse Requirements to Determine Appropriate Testing Strategies

1.1 Range of Requirements

Functional Requirements

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. This will include:
- → If the server is alive.
- → Restaurant data – name, location, menu, opening days.
- → No-Fly zones and their coordinates.
- → Coordinates of the University's Central Area.
- → Order data for the given date.

R3 ◊ The system will check all orders from the supplied date for validation. Validation will cover:
- → Credit Card Information, including:
  - ▪ Valid number.
  - ▪ Valid expiry date (in the future).
  - ▪ Valid CVV.
- → Cost of order in Pence, including delivery fee (100 pence).
- → Size of order (between one and four pizzas).
- → If pizzas in order are valid (exist on at least one menu).
- → If all pizzas in order come from the same restaurant.
- → If the restaurant being ordered from is open on the date of the order.

• Validation will compute and assign both an OrderStatus and OrderValidationCode to each order collected from the REST API. OrderStatus codes are as follows:
- → UNDEFINED – for a new, unvalidated order retrieved from the REST API.
- → INVALID – for any order that has an error and cannot be validated.
- → VALID_BUT_NOT_DELIVERED – for any order that is validated by the system but is yet to be delivered by the drone.
- → DELIVERED – for any successfully validated and then delivered order.

◊ OrderValidationCodes correspond to each step of verification and are as follows:
- → UNDEFINED, NO_ERROR, CARD_NUMBER_INVALID, EXPIRY_DATE_INVALID, CVV_INVAVLID, TOTAL_INCORRECT, PIZZA_NOT_DEFINED, MAX_PIZZA_COUNT_EXCEEDED, PIZZA_FROM_MULTIPLE_RESTAURANTS, RESTAURANT_CLOSED.

◊ The system will give a reason for invalidating an order, should there be an error.

R4 ◊ The system will have functions in place for the navigation of the drone. These will include:
- → The system can calculate the distance (in degrees) between two pairs of Longitude and Latitude coordinates (referred to as LngLat in future).
- → The system can determine if a pair of LngLat locations are 'close' to one another.
  - ▪ 'Close to' refers to a system constant of 0.00015 degrees.

- → The system can verify if a LngLat point is within a 'NamedRegion' (a region defined by its name a set of LngLat points creating a closed polygon) and subsequently the Central Area too.
- → The system can calculate what the next LngLat position of the drone will be, given a starting position and an angle.
    - ▪ The angle will be a multiple of 22.5, giving 16 possible directions of movement. These will be:
        - ▪ N, NNE, NE, ENE, E, ESE, SE, SSE, S, SSW, SW, WSW, W, WNW, NW, NNW
    - ▪ This position will 0.00015 degrees away from the starting position in the specified direction.
- → If the angle given is 999, the drone will 'hover' in place and not move.

- For each valid order, the system will use these functions to map a valid flightpath for each order, starting from a given delivery location (primarily Appleton Tower) each day.
    - → Each day's deliveries will start from the same location, the coordinates of the delivery destination.
    - → Each delivery will navigate from here to (or 'close to') the correct restaurant.
    - → The drone will then navigate back to (or 'close to') the delivery location.
        - ▪ This will take the reversed path to the restaurant.
    - → When both collecting and delivering pizza(s), the drone will hover in place for one turn at/close to the location, not changing position.
    - → Since there are only a small number of restaurants, the system will cache delivery routes and easily re-use them to make the system more efficient, rather than calculating a route for every delivery.
    - → Flightpaths shall not contain points within 'No-Fly Zones', nor on their borders.
    - → Once the drone has entered the Central Area on its delivery route, it shall not leave again until the order is delivered.

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

- → deliveries-yyyy-MM-dd.json
    - ▪ This will include OrderValidationCode, OrderStatus, Order Number and total cost in pence for *every* order on the supplied date.
- → drone-yyyy-MM-dd.geojson
    - ▪ This is a geojson file of the LngLat coordinates of every single move the drone makes on the supplied date. This file must be readable by an online geojson interpreter/visualiser.
- → Flightpath-yyyy-MM-dd.json
    - ▪ This will include the flightpath of every *valid* order throughout the day, and will label the Order Number, From Longitude and From Latitude, the angle of the movement and the To Longitude and To Latitude.

Non-Functional Requirements

Performance:

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

◊ The system must be able to handle a high volume of orders without crashing or encountering errors.

Security:

◊ The system must protect the personal information of its users, in particular credit card information, and prevent it from being accessed by malicious individuals.

Robustness:

◊ The system must check the supplied arguments for correctness, and gracefully terminate in the event of this.

◊ The system should inform the user of this error and where it occurred.

◊ If data cannot be retrieved from the REST API, because it is not there or cannot be accessed, the system should gracefully terminate and inform the user of this error.

Availability:

◊ The system must be available to users 24/7, and not become backed up and unavailable during busy periods.

Accuracy:

◊ The system must deliver correct orders to the right individuals, i.e should not be delivering the wrong pizzas to its users.

User-Friendliness:

◊ The system must be user-friendly and intuitive to use.

Scalability:

◊ The system must be constructed in such a way that allows for new features to be added or for the system to easily grow.

1.2 Range of Requirements

The Functional Requirements section entails a comprehensive variety of levels of requirements, including all of System, Integration and Unit requirements.
The Non-Functional requirements section also covers a broad selection of requirements, including those relating to the performance and robustness of the system. These requirements can be measured through testing. They are also classed as system requirements, as they outline how a completed system must function and cannot be verified on their own.
In section 1.1, all requirements marked with a hollow diamond (◊) represent System level requirements. These are requirements that define what is needed for the program to function properly. The best example here is the execution of the system and the arguments it must take to run. Requirements marked with a solid point (•) represent Integration level requirements, which define how components of the system interact with one another. They address the compatibility of the system to ensure fluid functionality between the different modules of the program. Bridging the gap between the REST API and the PizzaDronz system is a prime example, as it must integrate the connection between these two components to allow the system to function. The requirements marked with an arrow (→) represent Unit level requirements. These are specific requirements that refer to the individual functions

throughout that implement one feature in the system, such as defining the distance between two points.
System level requirements can be represented with several unit-level requirements, such as the navigation of the drone and all the functions required to build up this component of PizzaDronz.
I have chosen a selection of requirements (R1 – R6) that encompass several levels of requirements, including System, Integration, Unit, Performance and Robustness. Robustness is included in R1 as a sub-category of the system requirement.

1.3 Identifying Test Approach for Chosen Attributes

For each level of requirement, their corresponding testing strategy will be employed to affirm their functionality within the PizzaDronz system; Unit tests for Unit requirements, System tests for System requirements and Integration tests for Integration requirements. To further the scope of testing coverage, measurable aspects such as performance will also be tested for, for example the time taken to execute for a day will be tested to be less than 60 seconds. Time permitting, mutation testing could also be implemented to infer the effectiveness of the test suites.

1.4 Assess the Appropriateness of you chosen testing approach

The chosen testing strategies have been carefully selected to match the requirements being tested for, to maximise coverage and ensure complete functionality of the system. The tests will assess both functional and non-functional requirements of PizzaDronz, and if they are successful, will guarantee that these requirements are met within the application. The combination of test types will ensure that features not only work on their own, but also when brought together for larger components of the system. The tests will emphasise unit requirements, as they are easily testable, and if they are all functionally correct then it should follow that at the integration and system levels, they will work together as expected. Some extra tests have also been selected to further improve the system's coverage and functional completeness, such as performance tests for the runtime.
There are however some flaws in the testing strategy. For example, security is not a measurable attribute of the PizzaDronz system and therefore cannot be tested for with this strategy. Another difficulty is testing the content of the output files; these files contain up to thousands of entries, and testing them for correctness would not be an efficient use of time. Some behaviour in the navigation handling is also somewhat tedious to test, as the best method is to simply replicate the equations and test that the outputs are equal, which should be trivial anyway. Due to the nature of the REST API and how the orders are given, availability and handling high volumes of orders is also difficult to test for without effectively re-generating the server data manually. Flightpath efficiency would be a good aspect to investigate, although due to the hundreds of possible pathfinding algorithms and outputs, this is also very difficult and not worth dedicating time to given the scope of testing and given timeline.