# Introduction

Based on functional and non-functional requirements, as well as testing approaches identified and described in the document `LO1-requirements.pdf`, this testing plan is constructed to serve as a starting point in the Test-Driven Design and as the project evolves, requirements and specification may change and evolve to adapt for the project's needs.
In contrast to Waterfall, Agile software development cycle does not require all requirements and objectives laid out before design, testing and implementation, but is rather a continuous cyclic activity. This will allows us to implement and test order validation and flight path calculation before moving onto the client-facing frontend and mobile apps, which will have a significantly different range of requirements and most certainly will force the initial specification to change. Coupled with Test-Driven Development, Agile methodology shall be a favourable option at the moment.

PizzaDronz is a user-centric service and shall therefore prioritize validating functional requirements from `LO1-requirements.pdf`. Nevertheless, sufficient attention must be paid to the quality of the codebase itself verifying the specification, anticipating edge cases and responding to erroneous behaviour.

Broadly, the testing strategy shall be as follows:

1. Design and write Unit tests for the basic functionality to form a foundation of the application
2. Once Unit tests are finished and complete successfully, design and write Integration tests that will ensure the components interact well with each other and
3. When both Unit and Integration tests and the entire application are functional, move onto System level tests to ensure adequate behaviour of the system from the users' perspective.
   Should tests at any level fail, the cause should be identified and respective changes, where needed, must be introduced into the specification culminating in the use of the Agile methodology.

# Detailed plan:

The requirements can be grouped into 3 categories by their complexity within the project and are colour coded respectively.

1. System level - non-functional requirements that specify high-level behaviour of an application and can be tested after the application is completed.
2. Integration level - functionality that is comprised of multiple units interacting together.
3. Unit level - individual piece of functionality that can be implemented and tested separately.
   The tests will have to be done bottom-up, unit tests, then integration and then overall system tests.

## Functional requirements:

1. Testing taking the user input:
    1. Make sure the application only receives exactly 2 arguments in the following order:
        1. Check date not of the format "YYYY-MM-DD" is erroneous, and valid one is OK.
        2. Check malformed Rest Api base URL address is erroneous, and valid one is OK. Regex or built-in library to check for protocol, domain and top-level domain.
           Test inputs consisting of 0, 1 or more than 2 arguments produce and error.
           2 + 3 = 5

2. Testing fetching data from API:
    1. Check cases where the base URL is malformed or unreachable are covered and produce an error message.
    2. Check if the server is not active, an according error message is produced.
    3. Make sure the invalid JSON data received from the server triggers an error in all cases (orders, zones, restaurants).

    2 + 3 = 5
3. Testing order validation:
    1. Check orders without ID are invalidated.
    2. Check orders with a day different to the one supplied by the user are invalidated with UNDEFINED (no code available).
    3. Check orders with the number of items outside the range are invalidated (1-4) with MAX_PIZZA_COUNT_EXCEEDED. 0 or 5 items.
    4. Check pizzas from different restaurants result in order invalidation with PIZZA_FROM_MULTIPLE_RESTAURANTS, as do non-existent pizzas with PIZZA_NOT_DEFINED.
    5. Check if the restaurant is closed on the day of the order, order is invalidated with the RESTAURANT_CLOSED status.
    6. Check that for each order, whose price total does not add up to the price of all pizzas + a fixed £1 delivery fee are invalidated with TOTAL_INCORRECT.
    7. Check that invalid payment details are invalidated with respective error codes:
        1. Orders with card number numbers that are not 16-digit numeric values get CARD_NUMBER_INVALID status.
        2. Orders with card CVV that is not a 3-digit numeric value get CVV_INVALID status.
        3. Orders with card expiry date before the day of the processing are invalidated with EXPIRY_DATE_INVALID.

        Have to verify that the card expiring the same month of the same year as the order is still valid, anything before is invalid.
    8. Check valid orders have a NO_ERROR status.

    6 + 3 + 1 = 10
4. Check orders with the following path do not have an associated flight path:
    1. DELIVERED.
    2. INVALID.

    1 + 1 = 2
5. Check the valid orders are delivered in the order they were received.

    = 1
6. Compute flight path
    1. Check all adjacent moves in any path are "close" to each other.
    2. Check that in any path, coordinates of the Appleton Tower appear exactly twice.
    3. Check that in any path, drone hovers exactly twice.
    4. Check that in any path, no move coordinate is inside a NoFlyZone

    = 4 x N of tests that have flight path calculation
7. Export files
    1. Check if the dedicated folder does not exists, it is created.

    2. Check if the file exists, it is overwritten.
    3. Check at the end of the program, there are 3 files for the date.
       = 3 x N of tests with file creation
8. Test the application flow:
    1. Given valid input arguments, performs as expected and produces the 3 files based on the orders received and processed for a given day.

## Non-functional Requirements:

1. Performance - running time shall not exceed 60 seconds for any day, therefore we shall time the execution and assert.
   = 1
2. Robustness - this will be assessed post-testing and described in the next document `LO3.pdf` where we will devise criteria, such as relative code coverage, component coverage and others.

The total theoretical number of high-level tests = 5 + 5 + 10 + 2 + 1 + 4 + 3 = 30 tests.
The plan provides a list of at least 30 high-level tests covering all components at different levels and provides a reasonable coverage with a variety of approaches and tools used to verify that the application conforms to the specification and fulfils user requirements.
The total number of tests is likely to increase to accommodate for different techniques, such as combinatorial testing, further described in the `LO3-Testing.pdf` document.

## Issues:

Boilerplate status codes in the Order interface provide a range of order validation codes:

- UNDEFINED
- NO_ERROR
- CARD_NUMBER_INVALID
- EXPIRY_DATE_INVALID
- CVV_INVALID
- TOTAL_INCORRECT
- PIZZA_NOT_DEFINED
- MAX_PIZZA_COUNT_EXCEEDED
- PIZZA_FROM_MULTIPLE_RESTAURANTS
- RESTAURANT_CLOSED
  Due to inability to modify the interface, certain cases, where the order is empty, for example, can only be marked with the UNDEFINED status, however each case will be noted with a TODO annotation that will be revised in future iterations and more error codes should be made available.

# Instrumentation

To aid the testing process, certain helper methods and assertions and scaffolding in the form of stubs (mock-ups of parts of the system) will be developed and diagnostic output will utilised to explore the behaviour of subsystems.
To increase the confidence that all requirements are fulfilled and no rule is violated, to reduce time checking manually in every test, additional instrumentation shall be introduced that will allow us to dynamically check for certain attributes and reuse instrumented code.

# Instrumentation to be created:

1. Reusable method that will check attributes for a flight path provided as an argument:
    1. Checks all adjacent moves are "close".
    2. Checks no path coordinates are in noFlyZones (additional argument - list of noFlyZones).
2. Timing functions to measure running time.
3. Assertions to check validity of methods as expected, as well as check presence of certain attributes in objects.
   Scaffolding:
4. Mock-up REST server for internal purposes in the form of:
   Hardcoded NoFlyZones and Restaurants with Pizza Menus for testing.
    - Returns a valid order
    - Returns a valid restaurant
    - Returns a valid list of noFlyZones

# Evaluation

Scaffolding for mock-up REST server and example polygons will probably be the most time-consuming and resource intense tasks and should therefore be designed to be reusable and dynamic in advance.
The possible downsides of such tools might be overreliance on them in the sense that hard-coded examples might not be representative of the actual user input, however, they will address immediate edge cases.
Described instrumentation will provide reusable tools and simplify as well as strengthen the testing process.