LO2: Test-Planning Document for PizzaDronz Project

Introduction

This test plan is meant to represent an initial testing plan for PizzaDronz project based on the requirements (functional and non-functional) specified in the *LO1-Requirements.pdf* file. It may therefore change later on as the project evolves, assuming we are using Test-Driven Design.

The Agile software development lifecycle model is the most suitable for this project. Its main advantage is that there is no need to wait for a certain set of requirements to be met before attempting to test other parts, as specified in advance. Thus the whole development process is very dynamic and is efficient for the test-driven design.

Since we only have a limited amount of resource, it is infeasible to address all requirements in the testing process. Therefore, we will generally discuss when testing for satisfaction of each functional, as well as performance, efficiency and robustness non-functional requirements will take place, and in which capacity. We will go in some more detail for a selection of requirements, particularly what scaffolding and instrumentation are needed. We will also discuss how all testing that is to be carried out for this project fits into the Agile lifecycle.

Priority and Pre-requisites

1. *Functional Requirements*
   - All functional system-level requirements, as outlined in *LO1-Requirements.pdf*, shall be prioritised because they specify the essential functionality of the application. Therefore, a high level of resource should be allocated to meet them. Obviously, this also concerns the lower-level (integration and unit) requirements that form certain system-level requirements as their criteria.
   - Since early detection of issues will reduce the amount of time necessary to be dedicated to A&T, it is vital to include V&V activities in the process of development at the earliest possible stage. This is going to be the priority of our chosen test-driven design, so the software developed will need to satisfy the pre-engineered tests to be considered as correct. This principle mostly applies to unit tests (as they can be fully designed in advance), but it also concerns the integration- and system-level tests which can be designed/altered later on, after the unit tests can guarantee satisfaction of unit-level requirements. This approach fits well into the Agile lifecycle as it is not necessary to have a complete test set in advance of development.

2. *Non-Functional Requirements (Performance, Efficiency and Robustness)*
   - Non-functional requirements, such as performance and efficiency, shall be given a lower priority because they are not concerned with the rules of drone movement and other compulsory application features. Therefore, a lower level of resource should be allocated to meet them.
   - Robustness is a high-priority requirement as it guarantees the application runs smoothly in case of any unexpected problems. It can be satisfied early by simulating errors.
   - Since for other non-functional requirements, V&V activities will only take place after the system is complete, it is infeasible to detect unsatisfactory results early.

General Procedure

1. Unit testing in the process of development.
2. Integration testing when a certain functionality is completed, and unit testing for its components has been carried out and successful (e.g. central area and no-fly zones requirements).
3. System testing when the whole application has been completed.

If at any point the application does not pass certain level tests, corrections should be made, and the tests should be run again. Such procedure should be repeated as long as necessary until the tests pass and thus a requirement is satisfied (because we use test-driven design in the agile lifecycle).

Detailed Test Plan for Chosen Requirements

The 4 requirements to be tested are marked as **R1, R2, R3** and **R4** in the *LO1-Requirements.pdf* document.

- **R1:** This is an important requirement as it defines a user-application "interface". So:
  - To meet this requirement, only the following combination of inputs should be considered as a correct set of arguments for the application:
    There are precisely 3 arguments passed into the application, and they are:
    - a valid date in the format YYYY-MM-DD,
    - a valid base URL address, and
    - any word.
  - Different combinations of correct and incorrect inputs should be used to test if the application functionality meets this requirement.

- **R2:** This is an extremely important and quite a complex requirement as it comprises many unit-level requirements that define a valid order based on separate criteria. So:
  - Two different A&T approaches should be used to ensure this requirement is met:
    - At the unit level, **each criterion** should be thoroughly tested. Testing should be based on inputs of different patterns.
    - At the system level, order validation should be tested by mocking real orders (comprising all components). This will ensure **all criteria** are taken into account when validating orders.
  - Validation of order components (with as many different types of inputs as possible). General test specification: ***tests must pass if a criterion is satisfied and fail otherwise***.
    - Correct general fields:
      - ✓ Customer name not less than 2 characters
      - ✓ Order number is an 8-character string representing a hexadecimal number;
      - ✓ Order date is a valid date.
    - Valid credit card number:
      - ✓ Card number is of length 16;
      - ✓ Card number is in a correct IIN range for VISA (starts with 4) or MasterCard (starts with 51 – 55 or 2221 – 2720);
      - ✓ Card number has a valid checksum of its digits computed by Luhn's algorithm.

- Valid card expiry date:
  - ✓ Expiry date is a valid date in format MM/YY;
  - ✓ Expiry date is after the order date. If they are in the same month, card expiry date is considered the last day of that month and compared accordingly.
- Valid CVV:
  - ✓ CVV is a 3-character string containing only digits.
- Valid pizza count:
  - ✓ There are between 1 and 4 pizzas inclusively in the order.
- All pizzas in the order are defined:
  - ✓ All pizzas' names match the names of the pizzas on the restaurants' menus.
- All pizzas are from the same restaurant:
  - ✓ All pizzas in the order are on the menu of the same restaurant.
- Valid total cost of the order:
  - ✓ $total\ cost = sum\ of\ pizza\ prices + delivery\ fee$

  Sum of the pizza prices is obtained from the restaurant's menu.

➢ The last three unit-level criteria above (which presume information about pizzas, i.e. names and prices, can be obtained from the server) will also require some mocking of the databases of restaurants' menus.

- **R3:** This requirement is important because it is one of the main features of the application to record outcomes for all orders on a given date. So:
  - ➢ It should be ensured that orders are correctly associated with the following outcomes:
    - *DELIVERED,*
    - *VALID_BUT_NOT_DELIVERED,*
    - *INVALID_CARD_NUMBER,*
    - *INVALID_EXPIRY_DATE,*
    - *INVALID_CVV,*
    - *INVALID_TOTAL,*
    - *INVALID_PIZZA_NOT_DEFINED,*
    - *INVALID_PIZZA_COUNT,*
    - *INVALID_PIZZA_COMBINATION_MULTIPLE_SUPPLIERS,* or
    - *INVALID* (if general fields are incorrect).
  - ➢ For the requirement to be fully met, *DELIVERED* outcome should be taken care of in the testing process. This would only be possible with a full system mocking as it would require flightpath planning and deciding which orders to deliver and which not to. Therefore, for the purpose of this test plan, with limited resource, we will consider associating orders with all outcomes except for *DELIVERED*.
  - ➢ Clearly, order mocking will also be involved to verify meeting this requirement.

- **R4:** This non-functional robustness requirement is also important to be met as it guarantees the application does not unexpectedly crash under certain circumstances. So:
  - ➢ Proper exception handling must be ensured (catching them is the scope of testing).
  - ➢ If the server has no data for central area and/or no-fly zones, the application should run, if there is no data for restaurants and/or orders, the application should terminate.
  - ➢ REST server is to be used for simulating different scenarios mentioned above.
  - ➢ Thanks to the simulation, errors can be detected by early testing.

Scaffolding and Instrumentation

For our project, a certain amount of scaffolding and code instrumentation is going to be necessary. Scaffolding will mainly be in the form of mocking the system or its part. Code instrumentation includes writing to an output stream and using assertions in the development process. It will be extensively used during the development process and test engineering. However we will dedicate more attention here to the required scaffolding:

- **R1:** No scaffolding required.
- **R2:**
  - For unit-level tests, no scaffolding required (only if we decide to mock the entire order for rigorous testing – actually later decided to do for LO3). An exception is the three tests that involve accessing restaurants' menus. They would require scaffolding to be scheduled and implemented:
    - A mock restaurant database with their menus that include pizza names and prices.
  - For system-level tests, more scaffolding would be necessary:
    - A mock database of orders, in addition to the restaurants as mentioned above. Thus a significant amount of synthetic data is to be generated.
    - Quite a lot of effort should be allocated to mocking to include as many different scenarios as possible, for the tests to be adequate.
- **R3:**
  - The same scaffolding as for R2 could be used for R3. Ideally, a different set of mock orders should be used, however it would have to be scheduled as a separate task.
- **R4:**
  - The REST server to simulate communication with the server and handling occurrences of missing data. No additional scaffolding.

Process and Risk

One of the longest and resource-taking tasks is creating mock databases for restaurants and orders, therefore it should be scheduled as early as possible in the agile lifecycle. Another reason for that is the fact that R3 cannot be tested before R2 is satisfied, and they potentially use the same mock systems. Following the same reasoning, R2 should be tested immediately after the mock databases have been created (first unit-level requirements, when all are satisfied – system level). In parallel with R2, R1 and R4 should be tested. After R2 is met, R3 should be tested.

As already mentioned before, the general process is to come back to fixing problems discovered by the testing process, fix and test again. This should be done in parallel with the testing for satisfaction of requirements that were planned to be started/continued at a certain stage of the lifecycle.

There is a significant risk that is associated with all requirements involving synthetic data usage. It is that this data might be unrepresentative, and it might not consider some cases that will later result in a failure of the application. This risk is quite likely as it depends on how well the test cases are designed. However, in our project it is totally feasible to mitigate this risk by a careful and thorough selection of test cases that would ensure the application meets the requirements (particularly, R2 and R3).