LO2 - Software Testing

B196275

January 23, 2024

1 Introduction

This document outlines a high-level plan that is based on the requirements in `requirements.pdf`. I will be following a test-driven development (TDD) which entails the development of unit tests before implementation, and incrementally updating the initial plan as I develop the project.

The chosen lifecycle for this project is Agile, specifically Extreme Programming (XP), as we can dynamically test requirements as the project is developed rather than leaving all testing to the end. Testing code after it is developed could contain some unintentional bias, thus creating tests before hand allows for us to avoid this.

2 Priorities & Prerequisites

Due to the limited time, it is impossible to test all the requirements, thus we must prioritise what requirements are most important to test. The functional requirements that will be tested are FR3, FR4 and FR5. With FR4 and FR5 both being high priority, and FR3 being of medium priority. The non-functional requirement that will be tested is NFR1, whilst not being as high a priority as the functional requirements that are going to be tested, it is still important to verify as the the client is very keen on the application running within this timeframe.

Each test **pass/fail criteria** outlined will be monitored throughout the testing process to ensure that tests are passing, and if they do fail, the relevant details will be recorded. These details will be used to help make the relevant changes to fix the code, this will be repeated until the tests pass.

3 Detailed Test Plan

3.1 FR3 (Medium Priority)

Order validation is of medium priority, and it is important to verify to ensure that orders are being correctly validated. Orders have a status and validation code associated with it, which is updated depending on the result of order validation.

The partition principle outlined in CH03 of Y&P suggests that testing this requirement can be decomposed into tests for each order validation code, in addition the redundancy principle also applies here to ensure we cover as many scenarios as possible. This should be feasible as each order can have at most one error associated with it.

To verify this requirement, we need to ensure that orders are assigned the correct status and validation code based on the above criteria. The inputs, outputs and specification for order validation are as follows:

- Inputs:
  1. **order**: order to be validated.
  2. **restaurants**: list of available restaurants.
- Outputs:

1. **order**: same order with its status and validation code updated depending on validation.

- Specification:
  Orders are assigned a status of either **INVALID** or **VALID_BUT_NOT_DELIVERED**, and a validation code, the following list contains each possible one, with a description of why they may be invoked:

  1. **UNDEFINED**: Order has not been validated yet.
  2. **CARD_NUMBER_INVALID**: Length of card number is not 16 or card number string is not numeric.
  3. **EXPIRY_DATE_INVALID**: Expiry date is not the same month and year as the order or before the order was placed. Also, if the expiry date is not in "mm/yy" format.
  4. **CVV_INVALID**: If the card CVV is not numeric or not of length 3.
  5. **TOTAL_INCORRECT**: If the total cost of an order does not add up to the sum of the price of pizzas + 100 (delivery charge).
  6. **PIZZA_NOT_DEFINED**: If a pizza in an order does not exist in any of the defined restaurants.
  7. **MAX_PIZZA_COUNT_EXCEEDED**: If an order contains more than 4 pizzas.
  8. **PIZZA_FROM_MULTIPLE_RESTAURANTS**: If all pizzas in an order are not from the same restaurant.
  9. **RESTAURANT_CLOSED**: If an order for a restaurant is placed for a day when it is closed.
  10. **NO_ERROR**: The order has no error associated with it.

Pass/Fail Criteria: The actual validation code and status of an order must match up with the expected validation code and expected status to pass.

3.2 FR4 (High Priority)

The rest server allows for the application to retrieve necessary data for processing and delivering orders. There are 5 key endpoints, one for the restaurants, one for orders, one for no-fly zones, one for the central area and one for server health (returns true if online and false otherwise).

The verification of this requirement is of high priority, and extremely important, thus I will employ two A&T approaches:

1. Each method for each endpoint should be unit tested by mocking server responses and exceptions to ensure that the method handles any unexpected behaviour correctly.

2. Each method for each endpoint should be tested to verify that it properly integrates with the REST server.

The redundancy principle outlined in CH03 of Y&P can be applied to ensure there are enough tests to cover every possible scenario, so we can ensure that our REST client is robust. The inputs, outputs and specification for the rest client are as follows:

- Inputs:

  1. **base url**: the url of the rest server
  2. **date**: the date of orders (only required when retrieving orders)

- Outputs:
  This depends on what endpoint the data is being retrieved from

  1. **restaurants**: list of restaurants, empty if none are on the server or there is an error during retrieval.

2. **orders**: list of orders for a given date, empty if none are on the server or there is an error during retrieval.
3. **no-fly zones**: list of no-fly zones, empty if none are on the server or there is an error during retrieval.
4. **central area**: the central area, empty if none are on the server or there is an error during retrieval.
5. **server health**: true or false depending on if the server is up or down, empty if none are on the server or there is an error during retrieval.

- Specification:
  Data is de-serialised once retrieved at each endpoint when retrieved from the REST server, and in the event of an error or an exception, it should be caught and dealt with.

Pass/Fail Criteria: Lists returned from each endpoint should match the expected type to pass.

3.3 FR5 (High Priority)

The validity of paths generated by the system is important, particularly for safety reasons as the no-fly zones are highly populated areas, and to ensure that drones are not taking unnecessarily long paths. The verification of this requirement will be split into the testing of methods in 2 classes: **LngLatHandler** and **PathFindingAlgorithm**.

LngLatHandler contains 4 important functions that assist with geometric calculations. Unit testing covering different scenarios will be carried out to ensure each method functions correctly. These are their inputs, outputs and specification:

distanceTo: A function that calculates the Euclidean distance between a pair of coordinates.

- Inputs:
  1. **start position**: the start point
  2. **end position**: the end point
- Outputs
  1. **distance**: distance between the start and end points in degrees
- Specification:
  Distance should be correctly calculated using the "Euclidean Distance" formula.

Pass/Fail Criteria: The distance calculated by the function should match the expected distance to pass.

isCloseTo: A function that calculates if 2 points are "close to" each other.

- Inputs:
  1. **start position**: the start point
  2. **end position**: the end point
- Outputs
  1. **isClose**: true if the distance between the 2 points is <0.00015 degrees, otherwise false.
- Specification:
  True should be returned only if the 2 points are less than 0.00015 degrees apart.

Pass/Fail Criteria: Actual boolean output given 2 coordinates should match the expected boolean output to pass.

isInRegion: A function that determines if a point is within a polygon using the raycasting algorithm.

- Inputs:
    1. **position**: the point we want to check
    2. **polygon**: list of points(vertices) that make up the polygon
- Outputs
    1. **isInside**: true if the point is inside the polygon, false otherwise.
- Specification:
  True should be returned only if the point is inside the polygon, points on vertices or edges of the polygon are considered as inside it.

Pass/Fail Criteria: Actual boolean output given a point and a polygon should match the expected boolean output to pass.

nextPosition: A function that calculates the next position from a point.

- Inputs:
    1. **position**: the point we need to calculate the next position from
    2. **angle**: direction/angle the next position needs to be
- Outputs
    1. **isInside**: true if the point is inside the polygon, false otherwise.
- Specification:
  next point should be correctly calculated using trigonometry, angles not divisible by 22.5 should not be accepted, and the original position should be returned.

Pass/Fail Criteria: The calculated next position given two coordinates should correspond with the expected next position to pass.

PathFindingAlgorithm contains 1 testable function that generates flightpaths to and from Appleton Tower and relies heavily on methods defined in LngLatHandler. Thus the unit testing to imperative to finish before we test this. These are the inputs, outputs and specification for the path generation method:

- Inputs:
    1. **destination**: the point we want to go to.
    2. **no-fly zones**: list of polygons that the path cannot enter.
    3. **central area**: central area polygon
    4. **order**: the order that the path is being calculated for
- Outputs
    1. **moves**: list of moves from the start to the destination and back
- Specification:
  list of moves generated should never enter any no-fly zones, and abide by the central area rule defined in the sub-criteria for FR5 in `LO1.pdf` Section 2.

Pass/Fail Criteria: Every move should never enter a no-fly zone and abide by the central area rule to pass.

3.4 NFR1 (High Priority)

I will take 2 approaches to verify this requirement. A test will be created that executes the program 100 times, each execution being for a random date (to avoid any bias). The time will be recorded for each execution and will be checked to verify that it is less than 60 seconds. We can draw up the **pass/fail criteria** for this test from this.

Pass/Fail Criteria: All executions must be under 60 seconds to pass, otherwise all the test will fail.

To visualise the results of testing, I plan to plot a graph with 10 executions (systematically sampled, every 10 executions), along with the average execution time. This in combination with the test created will allow us to confidently verify this requirement.

4 Process and Risk

4.1 Evaluation of Plan & Gantt Chart

- **FR3**:
  - **Process**: This is a medium-priority requirement and thus building tests, and implementing this can be done after FR4 and FR5 as they are high-priority requirements.
  - **Risk**: There could be issues with creating adequate synthetic data which could cause delays.
  - **Effort**: **Order Validation: 8 Days:** 3 days towards developing tests, 3 days for implementation, 2 days towards testing and making additional changes, and a 3 day buffer for delays.
- **FR4**:
  - **Process**: This is a high-priority requirement and thus building tests, and implementing this should be done before FR3.
  - **Risk**: There are several risks associated with this requirement. Firstly if the REST server is down we cannot perform integration tests. In addition, I have minimal experience with Mockito, thus creating the unit tests may take more time than usual.
  - **Effort**: **REST Client: 12 days**: 6 days towards developing tests, 2 days for implementation, 4 days towards testing and making additional changes, and a 1 day buffer for delays.
- **FR5**:
  - **Process**: This is a high-priority requirement and thus building tests, and implementing this should be done before FR3.
  - **Risk**: However, for FR4, integration tests for the path generation in PathFindingAlgorithm cannot go ahead unless and until LngLatHandler has been thoroughly unit-tested. If LngLatHandler is poorly tested and algorithmic faults relating to LngLatHandler begin to appear in PathFindingAlgorithm during integration testing, it could cause us to go back and redo the unit testing, introducing extra costs.
  - **Effort**:
    * **LngLatHandler: 6 Days**: 2 days towards developing tests, 2 days for implementation, 2 days to testing and making additional changes, and a 1 day buffer for delays.
    * **PathFindingAlgorithm: 8 Days (Due to the complexities associated with path-finding algorithms I have allocated more time to it)**: 2 days towards developing tests, 3 days for implementation, 3 days to testing and making additional changes, and a 1 day buffer for delays.
- **NFR1**:
  - **Process**: This test will be carried out at the end of development as it requires the whole system to be complete, as it is a "System-Level" test.

  - **Risk**: Suppose we have certain dates that run under 60 seconds for some arbitrary reason, we want to ensure it is the case for any date, and thus this form of testing eliminates this risk.
  - **Effort**: **Statistical Tests: 6 Days**: 3 days towards developing tests, 3 days to testing and making additional changes.

General Comments: Unit-level functional testing in general must be initiated as soon as possible to avoid detecting bugs too late, as due to the tight deadlines for this project, significant delays could result in it being delivered late.

4.2 Contingency Plan

Personnel Risks: This is a solo project, thus if I were to fall ill during development or require extra time to up-skill to use a new technology, it could delay project completion.

- **Solution**: The effort for each requirement accounts for continuous education and extra training that may be required.

Schedule Risks: If my unit tests were to be inadequate for a certain requirement, it could lead to unexpected delays due to algorithmic faults being detected late down the line.

- **Solution**: Account for scheduling risks in the test plan with buffer days, allowing for deadlines to be pushed forward.

4.3 Gantt Chart

I have accounted for the risks discussed in the previous 2 sections, and have created a Gantt Chart outlining how I am going to schedule testing and development. Please refer to "`ganttchart.pdf`".

5 Scaffolding and Instrumentation

For this project, scaffolding will mainly be in the form of mocking responses and data. Code instrumentation includes the use of asserts, which is used throughout my tests.

- **FR3**: Substantial scaffolding is required. Mock pizzas and restaurants must be created, and for every test case a mock order must be specially created for that test, for example, if we are testing that CVV's longer than 3 aren't accepted, then we must create an order with credit card information that contains a CVV greater than 3. A comparison-based test oracle will be used as we can assert the order status and validation code values with the expected ones.
- **FR4**:
  - **Unit-Level**: Mockito will be used to mock responses from "ObjectMapper", and the data contained within the responses will be mocked as well. This includes the mocking of restaurants, no-fly zones, the central area, and orders. A comparison-based test oracle will be used here, to check the data returned is of the right size.
  - **Integration-Level**: Minimal scaffolding required. A little bit of code is required to generate random dates for testing order retrieval. A comparison-based test oracle will be used here, to check the data returned is of the right size.
- **FR5**
  - **Unit-Level**: No Scaffolding Required. A comparison-based test oracle will be used here, to check the values returned from the calculations are what we expected.
  - **Integration-Level**: Function required to return true or false depending on if a path is valid based on the defined criteria. A partial test oracle will be used here, to check if the function used for verifying if a path is valid returns true.

- **NFR1**: Code needs to be written to generate random dates, and the for timing each execution. No mocking is required for testing this requirement. A comparison-based test oracle will be used here, to check that each execution time is under 60 seconds.

General: I will employ `IntelliJ's` inbuilt coverage checker to check the code coverage (branch, line and method). I will use `JUnit` to write my tests, and `Mockito` for mocking REST server responses.

6 Evaluation of Scaffolding and Instrumentation

The scaffolding and instrumentation are generally very simple and thus are highly unlikely to contain any errors. The code for the function that checks if a path is valid could contain small errors if not developed properly. Thus I will have to dedicate a large amount of effort towards this.

Mocking responses is useful as we can ensure that each function for each endpoint handles exceptions and unexpected data correctly in F4. However, mocking data for testing, particularly for FR3 and FR4, we run the risk of not covering all possible cases due to not generating data representative of what may appear in the real world. Though this is unlikely in this software, it is important to highlight this as a potential risk. Given a longer time resource, I could have allocated more effort to generating realistic data.

The use of `JUnit Asserts` to verify results is what we expected, and will be used throughout my tests to determine if my tests pass or fail (**test oracle**). Using `JUnit` is good as it provides a standard set of assertion methods, making it easy to write maintainable tests, further detailed feedback is provided when a test fails.