Software Testing 2024/2025 Portfolio - Test Plan

Test plan overview

The Informatics Large Practical has a detailed specification to be followed and, as the code is built around this, it is intuitive that a Test-Driven Development approach would be the most effective in the design of the PizzaDronz system. This approach requires tests to be written before the implementation of the features and ensures the features and functionality of the application will align with the requirements of the ILP specification.

Given the vast amount of requirements from the ILP specification and the limited testing/development time available, my testing approach will make sure to prioritise the core functionalities of the application:

- Argument validation functionality.
- Drone Flightpath functionality,
- Order validation functionality,

These very general functionalities make up system-level requirements, however each of them rely on unit-level component. Therefore, my test approach will mostly focus on performing good and extensive unit tests that contribute to checking the system functionalities. Non-functional aspects and lower-priority requirements will be addressed where feasible to ensure comprehensive coverage.

Detailed testing focus

Argument validation functionality

The unit-level requirements making up the system-level functionality will be tested using partition-based testing to ensure proper input validation is carried out. Below is a breakdown of the tests and their respective partitions:

- Invalid number of arguments
  - Given inputs with two arguments containing a valid date and URL, the system should successfully parse the arguments and not throw any exceptions and return true.
  - Given inputs with only one argument, the system should throw an exception with the message "`Incorrect number of arguments passed`".
  - Given inputs with more than two arguments, the system should throw an exception with the message "`Incorrect number of arguments passed`".

  - Given inputs with no arguments, the system should not throw an exception and instead a warning with the message "`No arguments provided. Will run with default arguments`" and set the arguments to today's date and a default constant URL respectively. It is important to note that this is not explicitly something from the ILP specification, and is instead a feature that came from testing the system (it was quicker and more efficient to test with these default arguments).

- Invalid date
  - Given inputs that contain an invalid date, the system should throw an exception with the message "`Date not in Localdate format YYYY-MM-DD`".

- Invalid URL
  - Given inputs that contain an incorrectly formatted URL, the system should throw an exception with the message "`Invalid URL provided`".
  - Given inputs that contain an correctly formatted but unavailable URL, the system should throw an exception with the message "`Invalid URL entered - cannot be connected to`" .
  - Given inputs that contain an correctly formatted, available URL that is not a REST server containing the relevant data, the system should throw an exception with the message "`Incorrect URL for rest end point`" .

Drone flightpath requirements

The unit-level requirements making up the system-level functionality will be tested using partition-based testing to ensure accurate flightpath data is created. The tests cover typical cases and edge cases; it is important to note that this section of testing will not look at the flight paths themselves, but instead the movement system on which the flight path planning system will be developed be built. Below is a breakdown of the tests and their respective partitions:

- **Correct Distance Calculation**
  - Ensures that the calculated distance between two points matches the Euclidean distance, using points with known coordinates.

- **Close to a Point**
  - Validates the "closeness" of one point to another within a small threshold.
    * Given points very close to each other should return true.
    * Given points that are far apart should return false.
    * Given points that are almost close but just outside the threshold should also return false.

- **Within a Region**

  - Verifies if a point lies within a defined polygonal region.
    - A given point inside a given simple rectangular region (central) will return true.
    - A given point outside a given simple rectangular region will return false.
    - A given point inside a given multi-sided polygonal region (pentagon) will return true.
    - A given point outside a given multi-sided polygonal region will return false.

- **Next Point Calculation**
  - Ensures that the next position calculated based on the compass direction matches precomputed positions for each compass direction (0^o to 337.5^o in 22.5^o increments) with a given point.
  - It is important to note that the system will generate movements with a direction of 999^o in the final flight path. This is for the end points when the drone will hover. As these 999^o 'movements' are added after the flightpaths are generated, they will not be included in this aspect of testing.
- **Far Point Calculation**
  - Validates multi-step movement for the drone (used in the A* path finding heuristics) by comparing results to precomputed positions for each compass direction (0^o to 337.5^o in 22.5^o increments) with a given point.

Order validation requirements

The unit-level requirements making up the system-level functionality will be tested using partition-based testing to ensure accurate order validation is carried out. The tests cover typical cases and edge cases. Below is a breakdown of the tests and their respective partitions:

- **Valid Order**
  - Ensures that valid orders with all the correct information are processed successfully. These orders will return with an `OrderValidationCode` of `NO_ERROR`.
- **Invalid Card Details**
  - Validates that invalid card details are caught and the correct error/validation code is given to the order.
    - Given credit card numbers with invalid lengths (less than or more than 16) are rejected. These orders will return with an `OrderValidationCode` of `CARD_NUMBER_INVALID`.
    - Given credit card numbers containing invalid characters (non-numeric) are rejected. These orders will return with an `OrderValidationCode` of

CARD_NUMBER_INVALID.

  - Given expired credit cards (relative to the order date) are rejected. These orders will return with an OrderValidationCode of EXPIRY_DATE_INVALID.
  - Given credit cards close to validity but just expired (relative to the order date) are rejected. These orders will return with an OrderValidationCode of EXPIRY_DATE_INVALID.
  - Given credit cards close to expiring but still valid (relative to the order date) are processed correctly. These orders will return with an OrderValidationCode of NO_ERROR.
  - Given expiry dates in the wrong format are rejected. These orders will return with an OrderValidationCode of EXPIRY_DATE_INVALID.
  - Given expiry dates containing invalid (non-numeric) characters are rejected. These orders will return with an OrderValidationCode of EXPIRY_DATE_INVALID.
  - Given CVV numbers with invalid lengths (less than or more than 3) are rejected. These orders will return with an OrderValidationCode of CARD_NUMBER_INVALID.
  - Given credit card numbers containing invalid characters (non-numeric) are rejected. These orders will return with an OrderValidationCode of CARD_NUMBER_INVALID.

- **Invalid Pizza details**
  - Validates that invalid card details are caught and the correct error/validation code is given to the order.
    - Given orders in which all its pizzas exist in a restaurant menu are processed correctly. These orders will return with an OrderValidationCode of NO_ERROR.
    - Given orders in which any of its pizzas do not exist in a restaurant menu are rejected. These orders will return with a OrderValidationCode of PIZZA_NOT_DEFINED.
    - Given orders in which all of its pizzas exist the one restaurant menu are processed correctly. These orders will return with a OrderValidationCode of NO_ERROR.
    - Given orders in which any of its pizzas exist from more than one restaurant menu are rejected. These orders will return with a OrderValidationCode of PIZZA_FROM_MULTIPLE_RESTAURANTS.
    - Given orders containing null or empty pizzas are rejected. These orders will return with a OrderValidationCode of PIZZA_NOT_DEFINED. It is important to note that this error is new to the 2024 ILP specification and so the intended

error code (`EMPTY_ORDER`) is not currently available due to my software being created under the 2023 ILP specification.

* Given orders containing a number of pizzas that is greater than the defined `MAX_PIZZA_NUMBER` of 4 are rejected. These orders will return with a `OrderValidationCode` of `MAX_PIZZA_COUNT_EXCEEDED`.
* Given orders in which the total price of the pizzas is added correctly and matches the order total are processed correctly. These orders will return with a `OrderValidationCode` of `NO_ERROR`.
* Given orders in which the total price of the pizzas is incorrect are rejected. These orders will return with a `OrderValidationCode` of `TOTAL_INCORRECT`.
* Given orders in which the restaurant the pizzas will be coming from is open on the order date are processed correctly. These orders will return with a `OrderValidationCode` of `NO_ERROR`.
* Given orders in which the restaurant the pizzas will be coming from is closed on the order date are rejected. These orders will return with a `OrderValidationCode` of `RESTAURANT_CLOSED`.

System requirements

In addition to the partition-based unit-level tests, full system tests will be conducted and repeated to confirm the overall functionality of the PizzaDronz application. The tests will run with valid date and URL inputs and return the JSON and GeoJSON files specified in the ILP course requirements. There will also be inputs with dates that have 0 orders: this should run through completely without errors and instead simply provide a clear warning that the output will be empty due to lack of orders on that date. A timer can be used to ensure each system run-through takes less than 60 seconds as per the requirements.

These full system-level tests may help catch issues and meet requirements not looked at by unit tests as they combine the functionality and interaction of the different parts of the application together.

Unaddressed requirements and potential vulnerabilities

It is important to note that not all requirements are covered by the test plan. While my test plan does indeed cover the most fundamental areas of the system functionality, there are smaller areas which could have more explicit and extensive testing besides the system-level tests which confirm there are no errors in the code:

- The output handling section be tested to ensure that the JSON and GeoJSON files are correctly generated. This testing should cover edge cases like empty data or malformed fields as to make sure the system produces valid output files only.
- The actual flightpath creation section does have measures to ensure the most optimal path is found, however, with more time and resources, tests should be written to

validate the generated flight paths, checking for accuracy and proper handling of obstacles.

Assessment on test plan adequacy

This test plan provides a concrete foundation for making sure the most fundamental functionalities of the system are in place and work well. The unit-level and system-level tests that are in place are well structured and cover a wide breadth of requirements. However, the plan does lack some integration-level testing (especially looking at the A* flight pathing algorithm, or the output handling functionality) and adding tests for these areas would certainly reduce the risk of undetected errors and make sure there are smooth and fully functional interaction between all components of the system. Overall, despite the room for improvement in specific areas, my test plan is very suitable for the requirements gathered from the ILP specification.

Instrumentation

- Describe instrumentation of the code
- Assess clarity and effectiveness of instrumentation
- look further into limitations/improvements of instrumentation

The instrumentation includes print statements and assertions for each test (whether that be unit-level or system-level) to provide constant feedback on test progress and results. Exceptions are caught and validated to ensure the correct error messages are produced. This approach allows for easy identification of unexpected errors, incorrect thrown exceptions, or even just issues with the functionality of the system.

The instrumentation is effective and efficient for debugging this system as the consistent formatting of concise print statements to show exact progress helps with pinpointing exactly where an issue lies. This reliance on print statements in the console output may become less viable for testing on a much larger scale. In future, a separate

However, the reliance on console outputs may become unwieldy for larger-scale testing. Generating a test log report could help with clarity if this project (and therefore the necessary testing suite) became much larger and more complex. It would be able to organise the test feedback with proper timestamps, error messages and even severity levels.