Testing Techniques (B188603)

Overview

This document outlines the different approaches that were taken to testing the project and also the overall test coverage once the tests were completed, all tests that were run were successful mostly due to the project having already been completed in the previous year and so there was not much refinement of the project code itself needed during the process.

1. Functional Testing

Objective:
Ensure the system functions as intended by validating the output of core functionalities against the requirements.

Test Cases (some examples)

- **Order Validation**:
    - Validate orders based on restaurant availability, payment information, and order constraints (e.g., maximum pizzas).
    - Expected: Orders with invalid payment details are flagged appropriately.
- **Flight Path Calculation**:
    - Generate a flight path that avoids no-fly zones and stays within the central area.
    - Expected: The drone does not enter no-fly zones or leave the central area.
- **File Output Validation**:
    - Verify file names, structures, and content.
    - Expected: Files adhere to naming conventions (YYYY-MM-DD) and contain correctly formatted data.

Approach:
Mock data is used to simulate a variety of order and restaurant scenarios. Tests are run using Junit with asserts to confirm outputs match expectations.

2. Combinatorial Testing

Objective:
Explore the interactions between different components and inputs by testing combinations of parameters.

Test Combinations

- **Order and Restaurant Availability**:
    - Combine varying restaurant opening hours with orders placed on different days.
    - Expected: Orders are valid only if placed on days when the restaurant is open.
- **Flight Path and No-Fly Zones**:

- Test different placements of no-fly zones relative to the central area.
- Expected: The flight path avoids all no-fly zones, regardless of configuration.

- **Payment Information**:
  - Test combinations of valid and invalid card numbers, expiry dates, and CVVs.
  - Expected: Orders with invalid card details are flagged.

Approach:
Use parameterized testing in JUnit to automate tests for different combinations of inputs.

3. Structural Testing

Objective:
Ensure all code paths are executed to verify internal logic.

Techniques

- **Code Coverage**:
  - Use a code coverage tool like the built in IntelliJ (run with coverage) feature to measure coverage of classes, methods, and branches.
  - Expected: Achieve at least 80% coverage for critical components like OrderValidator, FlightAStar, and FileHandler.
- **Boundary Value Analysis**:
  - Test boundary conditions, such as orders placed on the edge of the central area or with exactly 2000 drone moves.
  - Expected: The system handles boundary conditions without errors.
- **Control Flow Testing**:
  - Validate conditional logic in methods like getValidOrders and getPathForDay.
  - Expected: All branches of conditional statements are executed.

Approach:
Automated unit tests are run with tools to track coverage and identify untested paths.

Inspection Testing (White-Box)

To ensure that functionality of key aspects such as result files like the geoJson were created correctly I used the website geoJson.io where the file could be easily visualized and this ensured that anything from flightpath that I might have missed during tests could be visually verified to be working correctly as the specification dictates such as not flying into the central area twice during a single path.

4. Test Coverage

Key Metrics

1. **Line Coverage**:

- Measures the percentage of executed lines in the code.
- Target: ≥ 80% for critical components.

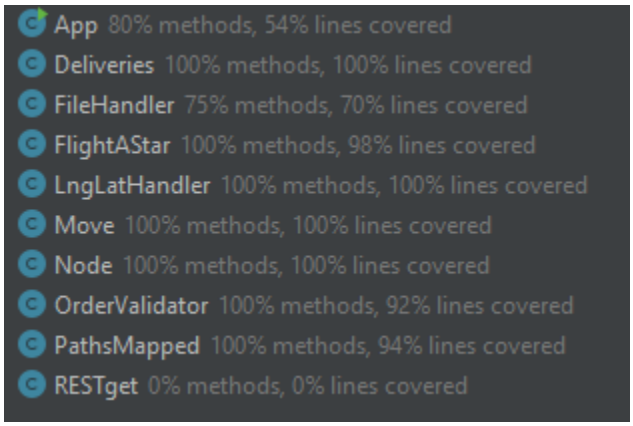
2. **Branch Coverage**:
   - Measures the percentage of executed branches in conditional statements.
   - Target: ≥ 70%.
3. **Path Coverage**:
   - Ensures all possible execution paths are tested.
   - Target: At least one test case for each unique path.

Tools

- **Built in IntelliJ Code Coverage:** to easily see the percentage of lines and methods covered by the tests.
- **Manual Inspection**: For visualizing execution paths in complex algorithms.

5. Evaluation of Results

Test coverage results



These are the results of test coverage. Due to the rest server being taken down for the project I was unable to properly test the RESTget class and also couldn't test the PizzaDronz package itself so the overall coverage was 68% but the coverage is much higher in real terms. This method and line coverage is overall good and does leave some room for improvement but with the time constraints higher coverage couldn't be achieved.