Software Testing 2023/4
Evidence Document

Contents

1 LO.1

1.1 Requirements

1.1.1 Non-functional Requirements

Measurable The program shall handle all fetched orders in sixty seconds or less.

Qualitative The software must incorporate robust exception handling mechanisms, preventing crashes and ensure operational stability, effectively managing both anticipated and unforeseen errors while maintaining user-friendly feedback and operational continuity.

Data Collection The software shall not collect nor store any user data. Any data fetched from the REST server shall not be aggregated and stored elsewhere.

1.1.2 Functional Requirements

1. The program shall accept (validate) two arguments:
   (a) the date for which to handle orders
   (b) the REST server's base URL from which to retrieve data
2. The program shall retrieve its dynamic data from the REST server:
   (a) Orders to be handled
   (b) Participating locations (restaurants)
   (c) No-fly zones and central campus area
3. The program shall validate that the received orders are valid:
   (a) The order must contain a 8-character hexadecimal number.
   (b) The order must contain at least one but no more than four pizzas.
   (c) The order must contain an open restaurant.
   (d) The order must contain a basket from a single restaurant.
   (e) The order must contain a valid payment card:
      i. The payment card must contain a 16-digit card number
      ii. The payment card must contain a 3-digit CVV number
      iii. The payment card must contain a valid expiry date and must not be expired
   (f) The order must contain a valid total which includes the basket price and delivery fee.
4. For valid orders, the program shall calculate the most optimal flight path for each order; starting at Appleton Tower (3.186874, 55.944494) and terminating at the order's restaurant's location:
   (a) The drone must abide by the 16-wind compass rose.
   (b) The drone must make moves of 0.00015° or less.
   (c) The drone must hover for one move once its destination has been reached.
   (d) The drone must not leave the central campus area more than once per leg.
   (e) The drone must not enter no-fly zones.
5. For valid orders, the program shall update their status to **delivered**.
6. The program shall output three files per given day:
   (a) *deliveries-YYYY-MM-DD.json* containing all the orders for the day, each entry must include for the following properties:
      - orderNo

- orderStatus
- orderValidationCode
- costInPencce

(b) *flighpath-YYYY-MM-DD.json* containing all the drone moves for the day, each entry must include for the following properties:

- orderNo
- fromLongitude
- fromLatitude
- angle
- toLongitude
- toLatitude

(c) *drone-YYYY-MM-DD.geojson* containing a visualisation of the drone's trips in GeoJSON format.

2 LO.2

2.1 Testing Plan

1. The program is expected to execute its main method in sixty seconds or less. [system]
2. The program is expected to handle exception within the main-loop (2.1.1): [system]
   (a) The program is expected to accept two arguments: [integration]
      i. The date as a valid string under YYYY-MM-DD format [unit]
      ii. The REST server's base URL as a valid string under the standard Java URL format (includes IP-based URLs) [unit]
   (b) The program is expected to check for REST server heartbeat. [unit]
   (c) The program is expected to check for at least one received restaurant. [unit]
   (d) The program is expected to check for runtime errors during data fetching. [unit]
   (e) The program is expected to check for runtime errors during output-file writing. [unit]
3. The program shall retrieve its dynamic data from the REST server; each endpoint is expected to: [integration]
   - A valid data type; [unit]
   - A returned error handling; [unit]
4. The program shall validate that the received orders are valid: [system]
   (a) The order number is expected to be present as a valid 8-character hexadecimal number string. [unit]
   (b) The order is expected to contain at least one but no more than four pizzas. [unit]
   (c) The order is expected to reference an open restaurant. [unit]
   (d) The order is expected to contain pizzas from the same restaurant. [unit]
   (e) The order is expected to contain a valid payment card: [integration]
      i. The payment card is expected to have a 16-digit card number as a valid string. [unit]
      ii. The payment card is expected to have a 3-digit CVV as a valid string. [unit]
      iii. The payment card is expected to have a valid expire date:
         - Under the MM/YY format; [unit]
         - Date is after order date; [unit]

(f) The order is expected to contain a valid total, calculated as: [unit]

$$\sum_{Pizza}^{Basket} Pizza.costInPence + deliveryFee$$

5. The program shall calculate the most optimal flight path for each order: [system]
   (a) The algorithm is expected to follow these functional directives: [integration]
      i. The drone is expected to hover for one move once its destination has been reached; the angle is expected to be set to 999.0 [unit]
      ii. The drone is expected to leave the central campus area only once per leg. [unit]
      iii. The drone is expected to stay out of no-fly zones. [unit]
   (b) The algorithm is expected to follow these mathematical directives: [integration]
      i. Moves angle is in increments of 22.5° [unit]
      ii. Moves are in increments of 0.00015° [unit]
      iii. Distance d between geographic points is calculated as: [unit]

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

   (c) The algorithm is expected to find a path in *500ms* or less. [unit]
6. For valid orders which have had their flight path calculated, these are expected to have their status changed to **delivered**. [unit]
7. The program is expected to output three files per given day: [system]
   (a) A summary file is expected to contain all the orders handled for the given day: [integration]
      i. The file is expected to hold the name *deliveries-YYYY-MM-DD.json*. [unit]
      ii. The file is expected to contain the following properties for each entry: [unit]
         - orderNo
         - orderStatus
         - orderValidationCode
         - costInPencce
   (b) A summary file is expected to contain all the drone moves for the given day: [integration]
      i. The file is expected to hold the name *flighpath-YYYY-MM-DD.json*. [unit]
      ii. The file is expected to contain the following properties for each entry: [unit]
         - orderNo
         - fromLongitude
         - fromLatitude
         - angle
         - toLongitude
         - toLatitude
   (c) A summary file is expected to all the drone flights paths for the given day: [integration]
      i. The file is expected to hold the name *drone-YYYY-MM-DD.geojson*. [unit]
      ii. The file is expected to under *GeoJSON* format; scaffolding expected, tested manually via *GeoJSON* web interface. [unit]
   (d) These are expected to be created under **/resultfiles**. [unit]

2.1.1 Main-loop Execution Flow

This section adds to the testing plan by providing a comprehensive overview of the program's main-loop execution flow (Figure 1).

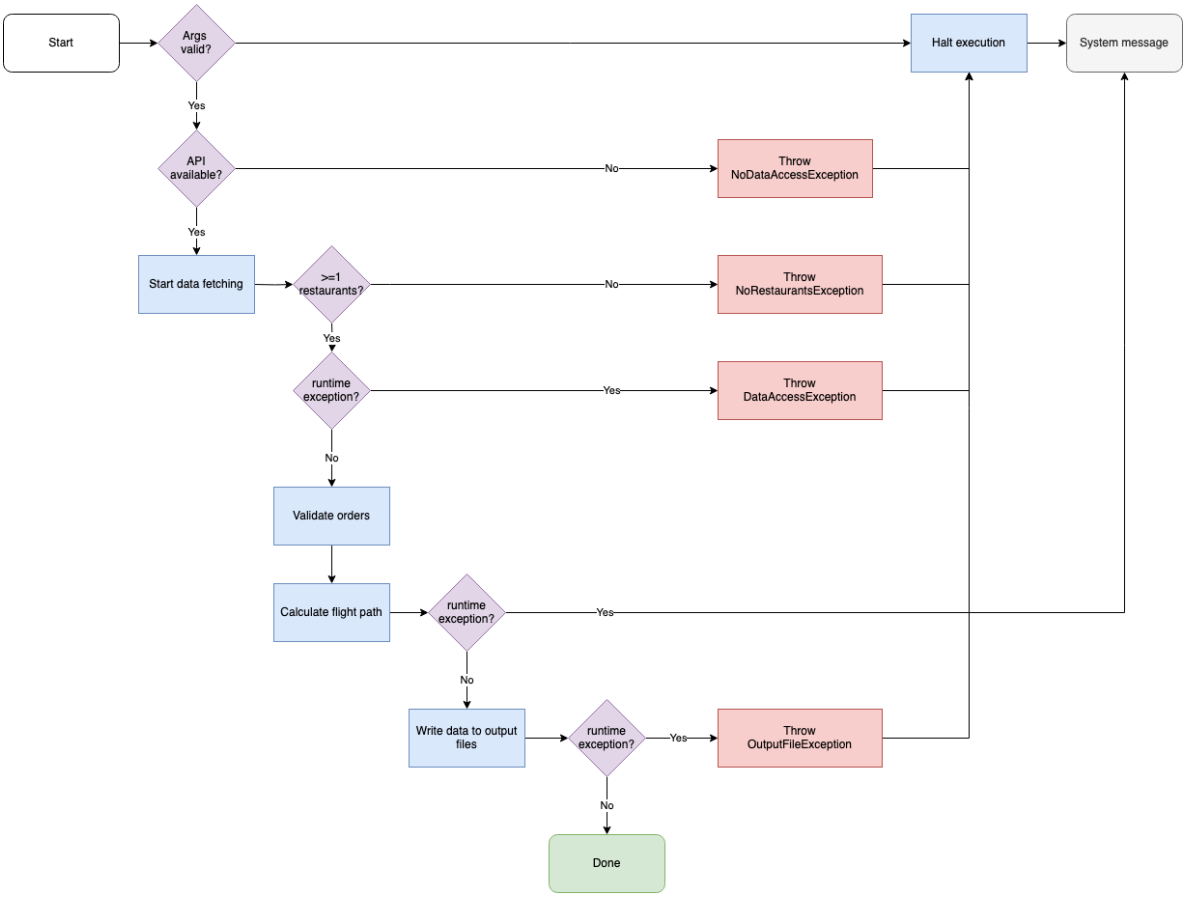


Figure 1: program's main-loop execution flow

2.2 Scaffolding

Scaffolding was utilised under various forms:

2.2.1 Interface Mocking

Here scaffolding was used to mock interfaces of the three primary components of the program: (1) REST client, (2) path finder, and (3) output-file writer. These mocks were designed to simulate the behaviour of the interfaces under different behaviour such as normal, no data, with exception, etc (Figure 2).

2.2.2 Test Data

Here scaffolding was used to mock dynamic data and provide ready-states to various unit tests. This data is used to simulate different scenarios and edge cases that the code may encounter during execution (Figure 3).

2.2.3 Test Utilities

Here scaffolding was used to provide local utilities to testing suites in order to simplify or aid in the attribute assertion (Figure 4).

```
4 usages  new *
private <E extends Exception> void testApp_Execute_Exception(
        Class<E> exception,
        IAPIClient apiClient,
        ISystemFileWriter fileWriter)
{
    try
    {
        App.execute(Logger.getGlobal(), date: "2023-12-01",
                apiClient,
                fileWriter,
                new MockPathFinder( exception: false,  noResult: false));

        fail("expected 'Exception' to be thrown");
    } catch (Exception e)
    {
        if (!exception.isInstance(e))
            fail(String.format("expected '%s' to be thrown, got '%s'", exception.getSimpleName(), e.getClass().getSimpleName()));
    }
}

5 usages  new *
static class MockAPIClient implements IAPIClient
{...}

2 usages  new *
static class MockSystemFileWriter implements ISystemFileWriter
{...}

2 usages  new *
static class MockPathFinder implements IPathFinder
{...}
```

Figure 2: example of interface-mock scaffolding used during exception-testing the primary execution method, as per the control flow defined in 2.1.1.

```
4 usages  Maximilien Zaleski
private Order buildStage1Order()
{
    final Order order = new Order();
    order.setOrderNo("foobar");
    order.setOrderDate(LocalDate.parse( text: "2023-12-01"));
    order.setOrderStatus(OrderStatus.UNDEFINED);
    order.setOrderValidationCode(OrderValidationCode.UNDEFINED);
    return order;
}

8 usages  Maximilien Zaleski
private Order buildStage2Order()
{
    final Order order = buildStage1Order();

    final CreditCardInformation cardInformation = new CreditCardInformation();
    cardInformation.setCvv("123");
    cardInformation.setCreditCardNumber("1234567890123456");
    cardInformation.setCreditCardExpiry("12/24");

    order.setCreditCardInformation(cardInformation);
    return order;
}

5 usages  Maximilien Zaleski
private Restaurant buildRestaurant(Pizza[] items)
{
    return new Restaurant(
            name: "Restaurant",
            location: null,
            new DayOfWeek[]{DayOfWeek.FRIDAY},
            items
    );
}
```

Figure 3: example of test data scaffolding used during testing of *OrderValidator.java*.

```
3 usages  Maximilien Zaleski
private boolean verify(String path)
{
    final File f = new File( pathname: ISystemFileWriter.LOCATION + path);
    assert f.exists() && !f.isDirectory();

    return f.delete();
}

2 usages  Maximilien Zaleski
private <T> T read(String path, Class<T> T)
{
    try
    {
        final File f = new File( pathname: ISystemFileWriter.LOCATION + path);
        final T data = new ObjectMapper().readValue(f, T);
        f.delete();

        return data;
    } catch (Exception e)
    {
        throw new RuntimeException(String.format("failed to read data from '%s': %s", path, e.getMessage()), e);
    }
}
```

Figure 4: example of test-utility scaffolding used during testing of *OutputFileWriter.java*; the utilities provide an easy way of verifying the existence of a file and type-casting the contents thereof.

3 LO.3

3.1 Range of Testing Techniques

3.1.1 White-box Testing

Generally, I will employ white-box testing which relies on having knowledge of the internal program structure. This method will aid us scrutinise the inner workings and validate internal code structures and operations.

Functional & Manual Combinatorial Testing Functional and manual combinatorial testing were used in conjunction to provide a framework for validating input against the above functional requirements. While functional focuses on behaviour, combinatorial testing explores a range of possible inputs and conditions; drawbacks and limitations are covered in 4.1.2. They encompass a formal assessment of the program's functional performance.

```java
public void testCard_Number()
{
    final CreditCardInformation cardInformation = new CreditCardInformation();
    cardInformation.setCvv("123");

    final String[] cases = new String[]{
            "123456789012345",    // 15
            "a1234567890123456",  // 15 + char
            "@1234567890123456",  // 15 + special char
            "123456789012345678", // 17
            "abcde",
    };
    for (String card : cases)
    {
        cardInformation.setCreditCardNumber(card);

        final Order order = buildStage1Order();
        order.setCreditCardInformation(cardInformation);

        validator.validateOrder(order, new Restaurant[]{null});

        assertEquals(OrderValidationCode.CARD_NUMBER_INVALID, order.getOrderValidationCode());
        assertEquals(OrderStatus.INVALID, order.getOrderStatus());
    }
}
```

Figure 5: Example of functional and combinatorial testing employed to asses the validity of the order's card number

Performance Testing Limited performance testing was used to assess certain performance attributes about the software such its overall runtime in a production setting as well as its path finding algorithm. This ensures that baseline performance targets are met.

```java
public void testFindPath_performance()
{
    final long start = System.nanoTime();

    // Location of 'World of Pizza', which is one of the further points from the base.
    final LngLat startPos = new LngLat( lng: -3.1869,  lat: 55.9445);
    final LngLat endPos = new LngLat( lng: -3.179798972064253,  lat: 55.939884084483);

    IPathFinder.Result result = pathFinder.findRoute(startPos, endPos);
    assert result.getOk();

    // Ensure that the algorithm completes within 500 ms.
    assert System.nanoTime() - start < 500000000L;
}
```

Figure 6: Example of performance testing on the path finding calculation

3.1.2 Black-box Testing

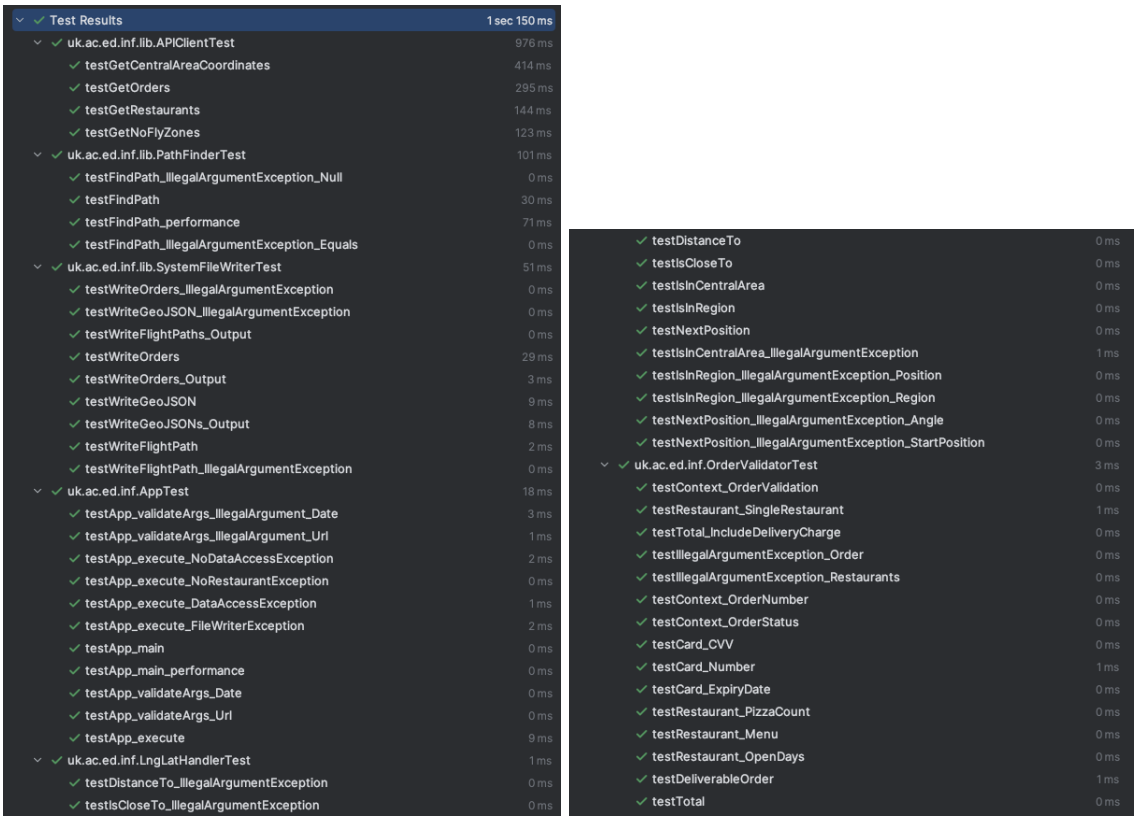
I will employ black-box testing to test how my program integrates with the external REST server.

3.1.3 Impact on Robustness

There techniques were chosen to cover as many internal domains as possible from low to high level. This ensures that the codebase and its algorithms are working as expected, and verifies that the majority of the program is able to cope with potential stressors, atypical inputs, and abnormal scenarios. As such the range of testing used form a comprehensive pictures of the program's reliability and robustness; weaknesses/failure points can be swiftly identified and effectively rectified.

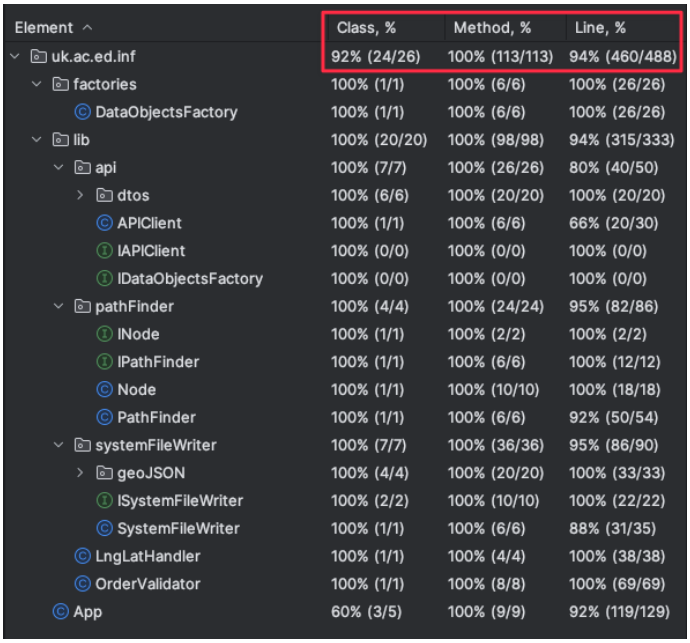
3.2 Results of Testing

All **54 tests** have passed (Figure 7) with an overall class coverage of **92%** (24/26), method coverage of **100%** (113/113), and line coverage of **94%** (460/488) (Figure 8).



(a) (b)

Figure 7: Testing suites passing



Element	Class, %	Method, %	Line, %
uk.ac.ed.inf	92% (24/26)	100% (113/113)	94% (460/488)
factories	100% (1/1)	100% (6/6)	100% (26/26)
DataObjectsFactory	100% (1/1)	100% (6/6)	100% (26/26)
lib	100% (20/20)	100% (98/98)	94% (315/333)
api	100% (7/7)	100% (26/26)	80% (40/50)
dtos	100% (6/6)	100% (20/20)	100% (20/20)
APIClient	100% (1/1)	100% (6/6)	66% (20/30)
IAPIClient	100% (0/0)	100% (0/0)	100% (0/0)
IDataObjectsFactory	100% (0/0)	100% (0/0)	100% (0/0)
pathFinder	100% (4/4)	100% (24/24)	95% (82/86)
INode	100% (1/1)	100% (2/2)	100% (2/2)
IPathFinder	100% (1/1)	100% (6/6)	100% (12/12)
Node	100% (1/1)	100% (10/10)	100% (18/18)
PathFinder	100% (1/1)	100% (6/6)	92% (50/54)
systemFileWriter	100% (7/7)	100% (36/36)	95% (86/90)
geoJSON	100% (4/4)	100% (20/20)	100% (33/33)
ISystemFileWriter	100% (2/2)	100% (10/10)	100% (22/22)
SystemFileWriter	100% (1/1)	100% (6/6)	88% (31/35)
LngLatHandler	100% (1/1)	100% (4/4)	100% (38/38)
OrderValidator	100% (1/1)	100% (8/8)	100% (69/69)
App	60% (3/5)	100% (9/9)	92% (119/129)

Figure 8: Code coverage report

4 LO.4

4.1 Omissions & Gaps in Testing Coverage

4.1.1 Flight Path Calculator

The flight path calculation is a non-deterministic operation and fine-grained testing was hard to perform. Rather than focusing on the underlying algorithm, I tested the output and verified certain required attributes, such as the hover moves; while this provides a certain degree of confidence in the output, it leaves the door open for possible mistakes within the algorithm itself – I tested for performance using one of the furthest locations made available in the testing dataset ('World of Pizza'), allowing a runtime of *500ms* per trip as the baseline. While in theory this should catch edge cases, there is no certainty in that regard.

4.1.2 Manual Combinatorial Testing

As discussed in Learning Outcome 3 (LO.3), The process of integrating input combinations was conducted manually, with selections based on my assessment of the most probable cases. This methodology, while practical, is inherently imperfect due to its reliance on subjective judgement. Consequently, it is likely that some potential input combinations were overlooked. This oversight might result in unforeseen bugs or behaviour following the initial release of the software.

4.2 Achieving Target Levels

4.2.1 Input Combinations

The input combination strategy should be refined by implementing a more systematic and automated approach. This approach reduces the reliance on manual and subjective selection of test cases and helps uncover edges cases which might not be immediately apparent.

5 LO.5

5.0.1 CI Pipeline Construction

The continuous integration pipeline is constructed through GitHub Actions; below is the configuration:

- **Triggers on** code changes to branches *master* (main) and *qa* (quality assurance, merges into *master*)
- **Actions taken** are (1) testing the project, (2) building the project, (3) caching dependencies for future runs (enables devs to quickly asses whether their additions broke the branch), (3) publish the built artifact for later download.

```yaml
name: CI

on:
  push:
    branches: [master, qa]
    paths:
      - 'src/**'
      - 'artifacts/**'
      - '.github/workflows/main.yaml'

jobs:
  build-and-test:
    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@v2
      - name: Set-up JDK 14
        uses: actions/setup-java@v1
        with:
          java-version: 18

      - name: Cache the Maven packages to speed up build
        uses: actions/cache@v1
        with:
          path: ~/.m2
          key: ${{ runner.os }}-m2-${{ hashFiles('**/pom.xml') }}
          restore-keys: ${{ runner.os }}-m2

      - name: Build and test project with Maven
        run: mvn -B package --file pom.xml

  publish-job:
    runs-on: ubuntu-latest
    needs: [build-and-test]
    steps:
      - uses: actions/checkout@v2
      - uses: actions/setup-java@v1
        with:
          java-version: 18
      - run: mvn -B package --file pom.xml -DskipTests
      - run: mkdir staging && cp target/PizzaDronz-*.jar staging
      - uses: actions/upload-artifact@v1
        with:
          name: PizzaDronz
          path: staging
```

Figure 9: Pipeline configuration

The pipeline also includes two environments: (1) **Prod** which is used on the *master* branch, (2) **QA** which is used on the *qa* branch (Figure 10) – a distinction is required in order to accommodate the *App.main* test, which executes the program in a production environment, rather than using mocked components.

Figure 10: Pipeline environments

5.0.2 CI Pipeline Demonstration

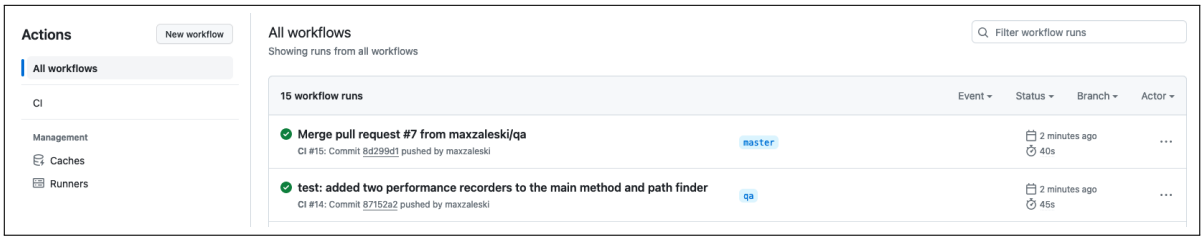


Figure 11: Showcase of the continuous integration workflow described in 5.0.1.

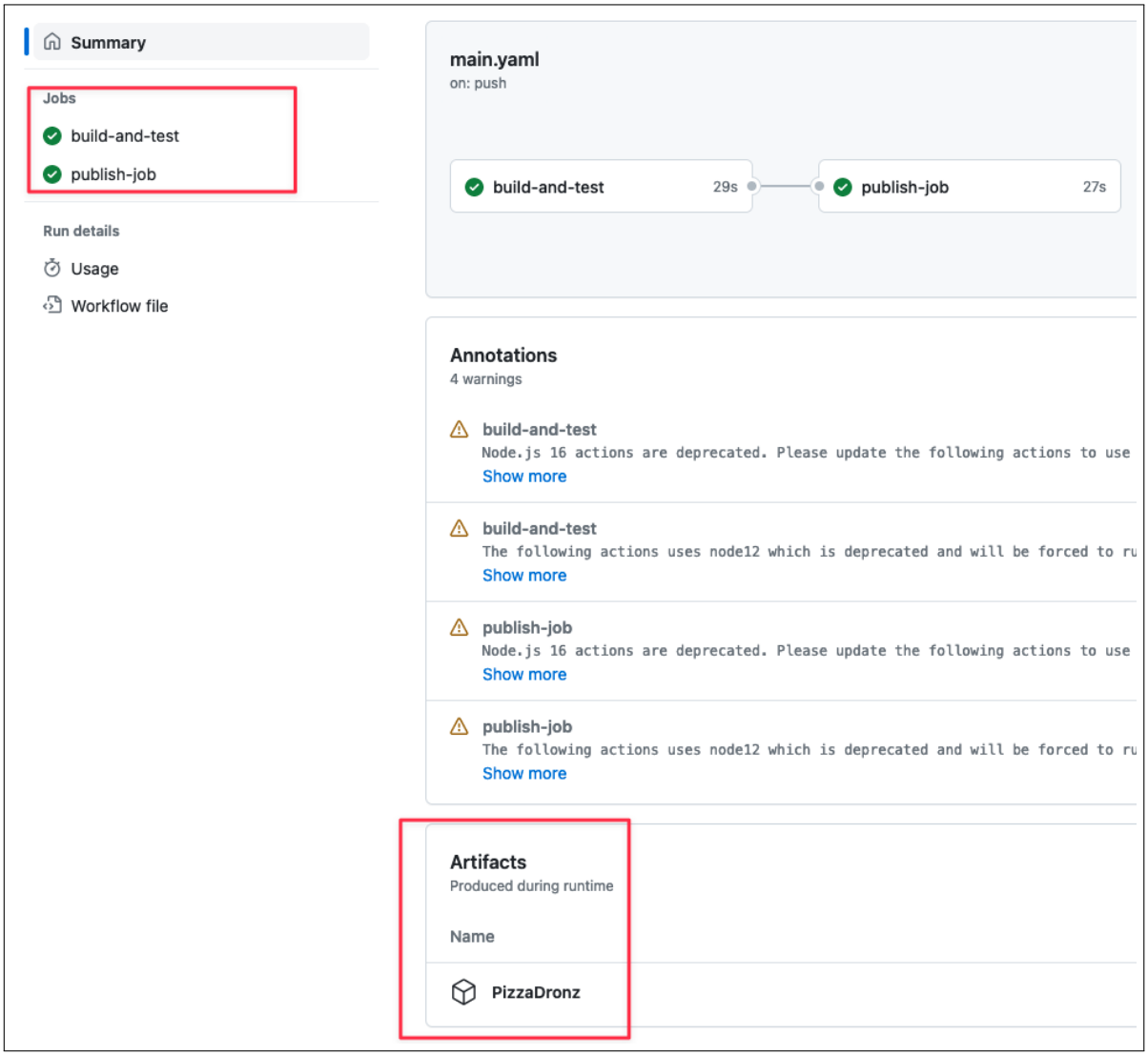


Figure 12: Showcases the successful execution of the pipeline on the *master* branch where the project was built, tested, and the resulting artifact was uploaded and ready to be downloaded.