FUNCTIONAL SOFTWARE TEST PLAN

for

Encost Smart Graph Project

Version 1.0

Prepared by: Student 2
SoftFlux Engineer

SoftFlux

May 5, 2024

Contents

Revision History

Name	Date	Reason for Changes	Version

1 Introduction/Purpose

1.1 Purpose

The purpose of this software test plan is to outline the testing strategies, objectives, and frameworks to ensure the ESGP system meets all functional and performance requirements. It aims to guide the testing team through different forms of testing from unit tests to system integration, ensuring thorough coverage and high-quality results.

1.2 Document Conventions

The following conventions are used in this document:

ESGP: Encost Smart Graph Project
ESHD: Encost Smart Homes Dataset
SRS: Software Requirements Specification
SDS: Software Design Specification
TC: Test Case

1.3 Intended Audience and Reading Suggestions

This document is intended for developers, testers, and project managers. Below are the potential uses for each reader type:

- Tester: The tester who wants a guide to help develop detailed tests cases, procedures, and expected outcomes in verifying the robustness of the software.
- Project Managers: The project manager who wants a better idea in coordinating the project and allocating resources by referring to the testing strategy and timelines.
- Developer: The developer who wants a better understanding how their code will be tested and highlight areas that may require additional attention or correction based on test outcomes.

1.4 Project Scope

This document presents the functional test plan for the Encost Smart Graph Project (ESGP). It aims to ensure that all functional requirements specified in the Software Requirement Specification are fulfilled, in alignment with Student 1's Software Design Specification. The focus is primarily on the high-priority requirements. Testing is conducted using black-box techniques to evaluate the system from an external perspective without considering internal system architecture.

Additionally, a targeted white-box testing approach is employed to assess the Summary Statistics feature of the ESGP. This aspect is evaluated through pseudocode that captures the intended functionality. To further enhance the integrity of our testing procedures, the white-box test cases are subjected to mutation testing. This testing method helps ensure the robustness and effectiveness of our test suites by introducing small changes to the code and evaluating the ability of existing tests to detect these modifications.

2 Testing Tools and Automation

This section provides recommendations for automated testing tools that can enhance the efficiency and coverage of our testing processes.

2.1 Recommended Tools and Their Applications

2.1.1 Unit Testing

- **JUnit:** A Java-based framework ideal for automated unit testing within the ESGP. JUnit allows for rigorous validation of each module within the project, confirming their operational integrity in isolation.
- **TestNG:** Offers advanced features over JUnit, this tool supports a broader scope of testing types, including parallel execution and interdependent tests, making it suitable for more complex testing scenarios in the ESGP system.

3 Black-box Testing

3.1 Categorising Users

3.1.1 Description

Objective: Ensure that the system correctly categorises users based on the type they specify during interaction—either as a Community User or an Encost Employee.

Functionality: The system will accept a single input string to determine the user type. Valid inputs are 'Encost Employee' or 'Community', which are handled with case insensitivity. These inputs correspond to predefined UserType enumeration values, where 'Encost Employee' is stored as '1' and 'Community' as '2'. Any input that does not match these specified values is expected to throw an exception.

3.1.2 Functional Requirements Tested

SRS 4.1 REQ-2 The system should store the user-type that the user has selected (community or encost-verified);

3.1.3 Test Type

- **Level of test:** Black-box testing, unit test
- **Test technique:** Equivalence partitioning for expected inputs

3.1.4 Test Cases

Test	Expected Output	Test Reason
'Community'	2	Valid input, proper casing
'COMMUNITY'	2	Valid input, upper case
'community'	2	Valid input, lower case
'Encost Employee'	1	Valid input, proper casing
'ENCOST EMPLOYEE'	1	Valid input, upper case
'12*34%5'	Throws Exception	Invalid input, numbers & symbols
'Some Word'	Throws Exception	Invalid input, unrelated words
'HELLO'	Throws Exception	Invalid input, random uppercase string
''	Throws Exception	Invalid input, empty string

3.1.5 Execution Conditions

- **Preconditions:** User is at the point of input for selecting user type
- **Postconditions:** User is either categorised correctly, or prompted that the input is invalid.

3.2 ESGP Account Login

3.2.1 ESGP Account Login Prompt

Description

Objective: Validate the login mechanism for Encost-verified users, ensuring proper parsing and authentication of username and password entries.

Functionality: The system requires a username and password entered with valid delimiters. Entries not conforming to this format should be rejected, and users should be prompted to re-enter valid credentials.

Functional Requirements Tested

SRS 4.2 REQ-1 The ESGP Account Login prompt should first prompt the user to enter their username. It should then prompt the user to enter their password;

SRS 4.2 REQ-2 Once the username and password have been entered, the system should check that the inputs are valid;

Test Type

- **Level of test:** Black-box testing, unit test
- **Test technique:** Expected inputs, edge cases, boundary cases.

Test Cases

Refer to Table 3.1 for the Test Cases for ESGP account login.

Table 3.1: Test Cases for ESGP Account Login

Test	Expected Output	Test Reason
'username password'	Proceeds to authentication	Valid delimiter (space)
'username\npassword'	Invalid	Invalid format (linefeed)*
'usernamepassword'	Invalid	No delimiter; incorrect format
'username,password'	Proceeds to authentication	Valid delimiter (comma)
'username—password'	Proceeds to authentication	Valid delimiter (pipe)
''	Invalid – Input required	Empty input; no data provided

**NOTE: According to the Software Design Specifications (SDS), both the username and password should be entered on a single line. Therefore, a line feed between the username and password will be considered invalid.*

Execution Conditions

- **Preconditions:** User is unauthenticated and at the login prompt

- **Postconditions:** Based on the input validation, either proceeds to user-specific features or remains at the login prompt for correct inputs.

3.2.2 ESGP Account Authentication

Description

Objective: Ensure that only users with credentials that match the predefined Encost userbase can log in, enhancing system security and user verification.

Functionality: The authentication method accepts two string parameters for the username and password. It hashes the input password and checks the resulting username and password hash against a stored table of valid credentials for ten Encost users. If the credentials are not found in the table, the system throws an exception, ensuring that only verified users can access the additional ESGP features.

Functional Requirements Tested

SRS 4.2 REQ-3 If the username and/or password are invalid, the system should inform the user that they have entered an invalid username and/or password and prompt them to enter their credentials again;

SRS 4.2 REQ-4 If the username and password are valid, the system should update the user type to be "encost-verified" and should provide the user with the ESGP Feature Options;

Functional Requirements Unable to be Fully Tested

SRS 4.2 REQ-5 Ten username and password pairs will be provided. The passwords should be encrypted before being stored in the application.

Note: The encryption functionality cannot be fully verified through black box testing, as this method requires knowledge of the system's internal mechanisms rather than just its inputs and outputs. Additionally, while test cases can confirm the presence of at least ten passwords, robust testing of the authentication system's security, using only these ten pairs, is impractical. This requirement is more suitably tested using white box testing techniques, which allow for examination of the underlying code and encryption algorithms.

Test Type

- **Level of test:** Black-box testing
- **Test technique:** State Transition Testing

Table 3.2: Test Cases for ESGP Account Login

States	State Description	Expected Actions
S1	Login Prompt displayed	User enters username and password
S2	Encost Features Options Access Given	User accesses all Encost feature

Table 3.3: Test Cases for ESGP Account Login

State	Input Scenario	Expected Output	Notes
S1	Correctly formatted input, valid credentials	Transition to S2	User is authenticated and gains access.
S1	Correctly formatted input, invalid credentials	Exception thrown, remain at S1	User receives error, prompted to try again.
S1	Incorrectly formatted input	Exception thrown, remain at S1	Error due to format, not credential mismatch.
S1	Empty username or password	Exception thrown, stay at S1	Verifies handling of empty fields which should not be valid.
S2	-	Access to Encost Features	Ensures proper system state after login.

Test Cases

Refer to Table 3.3 for the Test Cases for ESGP account authentication.

Execution Conditions

- **Preconditions:** User is unauthenticated and at the login prompt
- **Postconditions:** Based on the input validation, either proceeds to user-specific features or remains at the login prompt for correct inputs.

3.3 ESGP Feature Options

3.3.1 Description

Objective: Ensure that each user type can appropriately access and interact with the ESGP features they are authorised to use, according to their access rights.

Functionality: The ESGP system employs an internal method to determine user types and display corresponding feature options. Encost Users can choose from three features: Graph Visualisation (1), Upload Custom Dataset (2), and View Summary Statistics (3). Community Users have access to only Graph Visualisation (1). The system ensures that feature selection is facilitated through a number-based input system, allowing for seamless transitions to the chosen feature's interface, with safeguards to prevent unauthorised access and handle invalid inputs effectively.

3.3.2 Functional Requirements Tested

SRS 4.3 REQ-1 The ESGP Feature Options prompt should provide the user with a selection of features that they can pick from. For a Community User there is only one feature available: visualising a graph representation of the data. For a verified Encost User there are three features: (a) loading a custom dataset; (b) visualising a graph representation of the data; or (c) viewing summary statistics;

SRS 4.3 REQ-2 Once the user has selected a feature, the system should provide them with the prompt/information for that feature.

3.3.3 Test Type

- **Level of test:** Black box testing, integration test
- **Test technique:** Expected inputs, edge cases, boundary cases.
  - Expected Inputs: Evaluating the system's response to normal, valid inputs from different user types.
  - Error Guessing: Anticipating potential user errors or system mishandlings based on user interactions with the feature menu.

3.3.4 Test Cases

Refer to Table 3.4 for the Test Cases for ESGP Feature Options

3.3.5 Execution Conditions

- **Preconditions:** Users must be logged in with their respective user type identified correctly by the system.
- **Postconditions:** The system should be ready to execute the next action based on the feature selected without any lingering state from the previous interactions.

3.4 Loading the Encost Smart Homes Dataset

3.4.1 Description

Objective: Ensure the program is able to successfully locate the path to the ESHD which is stored inside the system and process the dataset according to its state (corrupted or complete).

Functionality: According to the SDS, internal classes work together to locate and process the ESHD and given invalid dataset file paths, corrupted dataset files, or datasets with partially missing data, the processing method reacts accordingly.

Table 3.4: Test Cases for ESGP Account Login

User Type	Input	Expected Output
Community User	"1"	Display graph visualisation prompt
Encost User	"2"	Transition to dataset loading prompt
Encost User	"1"	Display graph visualisation prompt
Encost User	"3"	Display summary statistics information
Encost User and Community User	"A"	Error message or no action
Encost User and Community User	"4"	Error message or no action
Encost User and Community User	" "	Error message or no action
Community User	"2"	Error message or feature not displayed

3.4.2 Functional Requirements Tested

SRS 4.4 REQ-1 The Encost Smart Homes Dataset file should be located inside the system;

SRS 4.4 REQ-2 The system should know the default location of the Encost Smart Homes Dataset;

SRS 4.4 REQ-3 The system should be able to read the Encost Smart Homes Dataset line by line and extract the relevant device information (an example from the Encost Smart Homes Dataset is included below).

3.4.3 Test Type

- **Level of test:** Black box testing, integration test
- **Test technique:** Expected inputs, edge cases, boundary cases.

3.4.4 Test Cases

Refer to Table 3.5 for the Test Cases for Loading the Encost Smart Homes Dataset.

Table 3.5: Test Cases for Locating and Processing ESHD

Description	Input	Expected Output
ESHD Dataset Located	Correct dataset path in default location	Dataset loaded (read and extracted) successfully
Invalid File Path	Incorrect dataset path in default location	Error message; unable to load dataset
Corrupted Dataset File	Path to a corrupted dataset file in default ESHD location	Error message; unable to process file
Partially Missing Data	Dataset with missing fields in default ESHD location	Error or default handling of missing data

3.4.5 Execution Conditions

- **Preconditions:** User must be authenticated and authorised to access the dataset functionalities.
- **Postconditions:** Dataset remains intact; system state should revert to pre-operation unless changes are required by operations.

3.5 Categorising Smart Home Devices

3.5.1 Description

Objective: Ensure that the Smart Home Devices from the Encost Smart Homes Dataset (ESHD) are correctly categorised and stored according to criteria defined in the SRS and SDS.

Functionality: The program's graph builder uses an internal method to read and process the ESHD. This method is expected to categorise devices accurately based on the dataset's information. The correctness of categorisation can be validated through the Summary Statistics feature, which displays the number of devices per category if the dataset is processed correctly. This test plan recognises that the accuracy and functionality of the Summary Statistics feature are critical to confirming correct categorisation, thereby establishing a dependency that must be acknowledged and managed within the testing strategy.

3.5.2 Functional Requirements Tested

SRS 4.6 REQ-1 The system should determine the device category for each device, based on the information provided on each line of the Encost Smart Homes Dataset (or custom dataset). Device categories are shown in the table below;

SRS 4.6 REQ-2 The system should create an Object for each device. This object should hold all of the information for that device.

3.5.3 Test Type

- **Level of test:** Black box testing, Integration Test
- **Test technique:** Expected inputs, edge cases.

Note: When integration testing the device categorisation functionality using the Summary Statistics feature, it is crucial to recognise the dependency that any issues within the Summary Statistics could affect the validity of the categorisation process. This acknowledgment is essential for managing potential risks in the testing strategy.

3.5.4 Test Cases

Refer to Table 3.6 for the Test Cases for ESGP Feature Options.

3.5.5 Execution Conditions

- **Preconditions:** User must be authenticated and authorised to access the summary statistics functionality.
- **Postconditions:** Dataset remains intact; summary statistics are displayed.

3.6 Building a Graph Data Type & Graph Visualisation

3.6.1 Description

Objective: To ensure that the graph data structure accurately stores Encost Smart Device objects, enabling the execution of ESGP Features.

Functionality: As outlined in the System Design Specification (SDS), the internal Graph Builder class constructs the graph data structure that is essential for visualising connections between Encost Smart Devices. This class processes input device objects and establishes necessary links to form a network of Encost Smart Devices.

3.6.2 Functional Requirements Tested

Building a Graph Data Type

SRS 4.7 REQ-1 Each Encost Smart Device Object should be stored in the graph data structure. The objects should be the nodes in the graph. The connection between objects should be the edges;

SRS 4.7 REQ-2 All unique datapoints should be included in the graph;

SRS 4.7 REQ-3 All households should be represented in the graph.

Table 3.6: Test Cases for ESGP Account Login

Dataset	Inputted Categorisation	Outputted Categorisation
ESHD	Encost Router: 23 Encost Controller: 21 Encost Smart Lighting: 32 Encost Smart Appliances: 54 Encost Smart Whiteware: 10	Encost Router: 23 Encost Controller: 21 Encost Smart Lighting: 32 Encost Smart Appliances: 54 Encost Smart Whiteware: 10
Custom Dataset with No Devices	Encost Router: 0 Encost Controller: 0 Encost Smart Lighting: 0 Encost Smart Appliances: 0 Encost Smart Whiteware: 0	Encost Router: 0 Encost Controller: 0 Encost Smart Lighting: 0 Encost Smart Appliances: 0 Encost Smart Whiteware: 0
Custom Dataset with 1 Category of Devices	Encost Router: 12 Encost Controller: 0 Encost Smart Lighting: 0 Encost Smart Appliances: 0 Encost Smart Whiteware: 0	Encost Router: 12 Encost Controller: 0 Encost Smart Lighting: 0 Encost Smart Appliances: 0 Encost Smart Whiteware: 0
Custom Dataset with 2 Categories of Devices	Encost Router: 12 Encost Controller: 0 Encost Smart Lighting: 21 Encost Smart Appliances: 0 Encost Smart Whiteware: 0	Encost Router: 12 Encost Controller: 0 Encost Smart Lighting: 21 Encost Smart Appliances: 0 Encost Smart Whiteware: 0
Custom Dataset with Multiple Categories of Devices	Encost Router: 12 Encost Controller: 36 Encost Smart Lighting: 156 Encost Smart Appliances: 60 Encost Smart Whiteware: 22	Encost Router: 12 Encost Controller: 36 Encost Smart Lighting: 156 Encost Smart Appliances: 60 Encost Smart Whiteware: 22

Note: These numbers are just for example and are not representative of the actual ESHD or custom datasets.

Graph Visualisation

SRS 4.8 REQ-1 The graph visualisation must be implemented using the GraphStream library;

SRS 4.8 REQ-2 The graph visualisation must show all nodes in the graph data structure;

SRS 4.8 REQ-3 The graph visualisation must show all connections between nodes (i.e.edges) in the graph data structure;

SRS 4.8 REQ-4 The graph visualisation must distinguish between different Device

Categories. For example, Encost Smart Lighting nodes should be visually different to Encost Smart Appliances nodes, and so on;

SRS 4.8 REQ-5 The graph visualisation must, in some way, illustrate each device's ability to send and receive commands from other devices. For example, it should be clear that an Encost Smart Hub 2.0 can both send and receive commands, while an Encost Smart Jug can receive commands but cannot send them.

3.6.3 Test Type

- **Level of test:** Black box testing, Integration Test
- **Test technique:** Visual Testing

  This method includes:
  - **Correct Node Display**: Verification that all elements (nodes) are displayed as per the design, with correct labels and categorisations.
  - **Accurate Edges and Connections**: Ensuring that all relationships (edges) between elements (nodes) are visually represented according to the specifications.
  - **Household Representation**: Checking that logical groupings and spatial relationships are maintained in the visual output.
  - **Comparative Validation**: Conducting side-by-side comparisons with control examples to validate the accuracy and consistency of visual outputs.

3.6.4 Test Cases

- **Test Case 1**
  - Purpose: Verify correct display of household and device nodes in the ESHD with correct connections.
  - Method: Load the ESHD and check the display.
  - Expected Output: Each device and household in the ESHD should appear as distinct nodes with clear, correct connections. The display should feature GraphStream-specific visual elements to confirm the library's use.
- **Test Case 2**
  - Purpose: Verify correct display of individual device nodes, categorised appropriately, without connections.
  - Method: Load a dataset with categorised devices having no connections and check the display.

  - Expected Output: Each device is displayed as a node with category identification.

- **Test Case 3**
  - Purpose: Ensure devices and their corresponding households are displayed as individual nodes without connections.
  - Method: Use a dataset with devices and households that have no interconnections and verify the display.
  - Expected Output: Each device and its household are accurately displayed as separate nodes.

- **Test Case 4**
  - Purpose: Confirm correct display of devices and households with directional connections.
  - Method: Load a dataset with connected devices and households and check the display.
  - Expected Output: Nodes for devices and households are displayed with clear, directional connections.

- **Test Case 5**
  - Purpose: Validate the graph's accuracy against a manually constructed version.
  - Method: Manually create a graph using a subset of the dataset and compare to the automated version.
  - Expected Output: The automated graph should mirror the manually constructed graph in structure and content.

3.6.5 Execution Conditions

- **Preconditions:** Users must be authenticated and authorised to access and load custom datasets.
- **Postconditions:** The graph visualiser should faithfully represent the loaded dataset with no discrepancies.

4 White-box testing

4.1 Calculating Device Distribution Pseudocode

The system is designed to allow verified Encost Users to access and visualize the distribution of devices, organized by category and type. This capability relies on data stored within a graph data structure.

The calculation and display of device distribution are managed through three interconnected functions as outlined in the Software Design Specification (SDS):

- `countByCategory()`: Intended to calculate the number of devices across all categories. However, as the function does not accept any parameters, it implies an aggregation over all categories, making the return of a single integer counterintuitive. It would logically return a summary of counts across all categories.
- `countByType()`: Supposed to calculate the number of devices of a specific type within a category. This function would benefit from accepting a category as a parameter, facilitating a return of counts per type within that category, which is more practical given that multiple types can exist within a single category.
- `outputDistribution()`: Displays the distribution statistics, which ideally requires detailed data from the aforementioned functions to effectively organize and present information.

Given these considerations, it is recommended that `countByCategory()` return a structured collection, such as a map, with keys as categories and values as counts, rather than a single integer. Similarly, `countByType()` should also return a structured collection detailing counts per type within a specified category. These adjustments will significantly enhance the practicality and functionality of the `outputDistribution()` function.

4.1.1 Functional Requirements

SRS 4.2 REQ-1 The system should use the information stored in the graph data structure to calculate the number of devices that exist in each device category.

SRS 4.2 REQ-2 For each device category, the system should also calculate the number of devices that exist for each device type;

SRS 4.2 REQ-3 The system should output these figures to the console in a clear and concise manner.

4.1.2 countByCategory() Pseudocode

Refer to Figure 1 for the Pseudocode of the countByCategory function.

```
Map<String, Integer> countByCategory()
    INITIALISE Map of category counts corresponding to each category
    GET graph of devices
    FOR EACH device in the graph
        SWITCH Category of device
            CASE Router:
                Get Router Category count from Map
                Increase Router count by 1
            END CASE
            CASE Controller:
                Get Controller Category count from Map
                Increase Controller count by 1
            END CASE
            CASE Lighting:
                Get Lighting Category count from Map
                Increase Lighting count by 1
            END CASE
            CASE Appliances:
                Get Appliances Category count from Map
                Increase Appliances count by 1
            END CASE
            CASE Whiteware:
                Get Whiteware Category count from Map
                Increase Whiteware count by 1
            END CASE
        END SWITCH
    END FOR
    RETURN Map of category counts
```

Figure 1: Pseudocode for Counting Categories

4.1.3 countByType() Pseudocode

Refer to Figure 2 for the Pseudocode of the countByType function.

4.1.4 countByCategory() Pseudocode

Refer to Figure 3 for the Pseudocode of the outputDistribution function.

```
Map<String, Integer> countByType(String category)
  INITIALISE Map of device type counts
  GET graph of devices
  FOR EACH device in the graph
    SWITCH Category of device

      CASE Router:
        SWITCH Type of device
          CASE "Router":
              Get "Router" Type count from Map
              Increase "Router" Type count by 1
          END CASE
          CASE "Extender":
              Get "Extender" Type count from Map
              Increase "Extender" Type count by 1
          END CASE
        END SWITCH
      END CASE

      CASE Controller:
        SWITCH Type of device
          CASE "Hub/Controller":
            Get "Hub/Controller" Type count from Map
            Increase "Hub/Controller" Type count by 1
          END CASE
        END SWITCH
        END CASE

      CASE Lighting:
        SWITCH Type of device
          CASE "Light Bulb":
              Get "Light Bulb" Type count from Map
              Increase "Light Bulb" count by 1
          END CASE
          CASE "Strip Lighting":
              Get "Strip Lighting" Type count from Map
              Increase "Strip Lighting" Type count by 1
          END CASE
          CASE "Other Lighting":
              Get "Other Lighting" Type count from Map
              Increase "Other Lighting" Type count by 1
          END CASE
        END SWITCH
      END CASE

      ... (additional cases for other device categories & their device types)

    END SWITCH
  END FOR
  RETURN Map of device type counts within the specified category
```

Figure 2: Pseudocode for Counting Device Types Within a Category

```
void outputDistribution()
  INITIALISE Map of category counts corresponding to each category
  INITIALISE Map of type counts corresponding to each device type
  CALL countByCategory Function AND set to Map of Category Counts

  FOR EACH Category
    OUTPUT the Category Count with current category
    CALL countByType w/ current category AND set to Map of Device Type Counts
    OUTPUT the device type count(s) for current category
  END FOR
```

Figure 3: Pseudocode for Counting Categories

4.2 countByCategory() Testing

4.2.1 countByCategory() Statements

Referring to the Pseudocode for Counting Categories (Figure 1), the following outlines the number of statements present in the countByCategory() function.

1. Initialise Map of category counts
2. Get graph of devices.
3. For each device, check category and update count:
   - The retrieval of the specific category count inside each case
   - The increment of the category count inside each case
4. Return the Map of category counts

Total Number of Statements: $4 + (3 \times \text{number of categories})$

4.2.2 countByCategory() Branches

Referring to the Pseudocode for Counting Categories (Figure 1), the following outlines the number of branches present in the countByCategory() function.

- Switch-Case on Device Category - Each case represents a branch:
  1. Router
  2. Controller
  3. Lighting
  4. Appliances
  5. Whiteware

Total Number of Branches: $1 \times$ number of categories (since the switch statement branches exclusively to one corresponding case for each category).

4.2.3 countByCategory() Program Paths

Referring to the Pseudocode for Counting Categories (Figure 1), the following outlines the number of program paths present in the countByCategory() function.

1. Graph Empty: Loop does not iterate
2. Graph with Single Device: Single iteration with one case executed
3. Graph with Multiple Devices, Single Category: Multiple iterations, single case executed repeatedly
4. Graph with Multiple Devices, Multiple Categories: Multiple iterations, multiple cases exeucted

4.2.4 countByCategory() Test Cases

Test Case 1: Empty Graph

- **Input:** An empty graph of devices.
- **Expected Output:** A Map with zero counts for each category.
- **Purpose:** Tests initialisation and handling of empty collections.
- Branch Coverage: $0/5 = 0\%$

Test Case 2: Graph with One Device of Each Category

- **Input:** A graph containing one device per category (Router, Controller, Lighting, Appliances, Whiteware).
- **Expected Output:** A Map where each category has a count of one.
- **Purpose:** Ensures each switch case can handle an increment operation correctly.
- Branch Coverage: $5/5 = 100\%$

Test Case 3: Graph with Multiple Devices in One Category

- **Input:** A graph containing multiple devices of a single category, e.g., 5 Routers.
- **Expected Output:** The Map shows the count for Routers as 5, and all other counts are zero.
- **Purpose:** Tests the correct incrementation of a single category multiple times, verifying loop and increment logic.

- Branch Coverage: $1/5 = 20\%$

Test Case 4: Graph with Multiple Devices in Multiple Categories

- **Input:** A graph with varying numbers of devices in all categories (e.g., 3 Routers, 2 Controllers, 1 Lighting, 4 Appliances, 1 Whiteware).
- **Expected Output:** A Map correctly reflecting the input distribution.
- **Purpose:** Verifies that the method can handle complex inputs and correctly increment multiple categories.
- Branch Coverage: $5/5 = 100\%$

4.3 countByType() Testing

4.3.1 countByType () Statements

Referring to the Pseudocode for Counting Device Types (Figure 2), the following outlines the number of statements present in the countByType() function.

1. **Initialisation of Map:** Initializes a map to store the counts of each device type.
2. **Retrieve Graph of Devices:** Fetches the graph containing device data.
3. **Loop Through Devices:** Iterates over each device in the graph to process based on category and type.
4. **Category and Type Processing:**
   - Utilises a nested `SWITCH` structure to check device category.
   - For each category, another `SWITCH` handles the type-specific counting:
     - For each type, two operations are performed: (1)
       - Retrieve the current count from the map. (2)
       - Increment the count for the type by one. (3)
5. **Return the Map:** Returns the map containing the counts of device types for the specified category after all devices are processed.

Total Number of Statements: With 5 categories and an average of 1-3 types per category, the total statement count is calculated as follows: $1(\text{init}) + 1(\text{get graph}) + 1(\text{loop}) + 5(\text{number of categories}) + 3N(\text{per type operations}) + 1(\text{return})$, where N is the total number of types across all categories.

4.3.2 countByType() Branches

Referring to the Pseudocode for Counting Device Types Within a Category (Figure 2), the number of branches present in the countByType() function is calculated as follows:

- **Outer SWITCH (Category Level)**: This switch statement handles the categorisation of devices. Each category represents a distinct branch.
  - Categories include: Router, Controller, Lighting, and two additional unspecified categories.
  - Total branches in the outer SWITCH: 5 (one for each category).
- **Inner SWITCH (Device Type Level)**: Each category has an inner switch statement that categorises devices by their specific types.
  - Router Category: Handles 2 device types (Router, Extender).
  - Controller Category: Handles 1 device type (Hub/Controller).
  - Lighting Category: Handles 3 device types (Light Bulb, Strip Lighting, Other Lighting).
  - Appliances Category: Handles 3 device types (Kettle, Toaster, Coffee Maker).
  - Whiteware Category: Handles 3 device types (Washing Machine/Dryer, Refrigerator/Freezer, Dishwasher).
  - Total branches in all inner SWITCH statements: $2 + 1 + 3 + 3 + 3 = 12$ branches.
- **Total Number of Branches**: Combining both levels of SWITCH statements, the total branch count is calculated as:

  5 (categories) + 12 (device types) = 17 (total branches)

4.3.3 countByType() Program Paths

Referring to the Pseudocode for Counting Device Types Within a Category (Figure 2), the following outlines the number of program paths present in the countByType() function:

1. **Empty Graph:**
   - **Scenario:** No devices present in the graph.
   - **Path:** The function returns an empty map without entering any loop.
2. **No Devices Match the Category:**
   - **Scenario:** Devices present but none match the specified category.
   - **Path:** The function iterates over the graph but skips all devices, resulting in an empty map.

3. **Single Device Matching the Category:**
    - **Scenario:** One device that matches the specified category and type.
    - **Path:** The function finds the device, increments the count for its type, and returns the map.
4. **Multiple Devices, Single Type, Matching Category:**
    - **Scenario:** Several devices of the same type within the specified category.
    - **Path:** The function processes each device, repeatedly incrementing the count for this type.
5. **Multiple Devices, Multiple Types, Matching Category:**
    - **Scenario:** Various device types, all within the specified category.
    - **Path:** Each device type is processed, with counts incremented accordingly for each type within the category.

4.3.4 countByType() Test Cases

Given the function's design, where only one category can be processed per function call, achieving 100% branch coverage cannot be accomplished with a single test case. Instead, multiple test cases are necessary, each designed to address a specific category and the device types within that category.

Test Case 1: Empty Graph

- **Input:** An empty graph.
- **Expected Output:** An empty map.
- **Purpose:** Ensures the function handles empty input correctly.
- **Branches Covered:** None.
- **Branch Coverage:** $0/17 = 0\%$.

Test Case 2: No Matching Category

- **Input:** Graph with devices, none matching the specified category.
- **Expected Output:** An empty map.
- **Purpose:** Tests the function's response to non-matching category inputs.
- **Branches Covered:** 1 (Category does not match).
- **Branch Coverage:** $1/17 \approx 5.88\%$.

Test Case 3: Device Types in "Router" Category

- **Input:** A graph with devices only from the "Router" category, including various "Router" and "Extender" types.
- **Expected Output:** A map with counts reflecting the number of each device type within the "Router" category.
- **Purpose:** Verifies accurate counting for each type within the "Router" category.
- **Branches Covered:** There are 2 types under "Router", this would cover 3 branches (1 for the category, 2 for the types).
- **Branch Coverage:** $3/17 \approx 17.65\%$.

Test Case 4: Device Types in "Controller" Category

- **Input:** A graph with devices only from the "Controller" category, including various "Hub/Controller" types.
- **Expected Output:** A map with counts reflecting the number of each device type within the "Controller" category.
- **Purpose:** Verifies accurate counting for each type within the "Controller" category.
- **Branches Covered:** There is 1 type under "Controller", this would cover 2 branches (1 for the category, 1 for the type).
- **Branch Coverage:** $2/17 \approx 11.76\%$.

Test Case 5: Device Types in "Lighting" Category

- **Input:** A graph with devices only from the "Lighting" category, including various "Light Bulb", "Strip Lighting" and "Other Lighting" types.
- **Expected Output:** A map with counts reflecting the number of each device type within the "Lighting" category.
- **Purpose:** Verifies accurate counting for each type within the "Lighting" category.
- **Branches Covered:** There are 3 types under "Lighting", this would cover 4 branches (1 for the category, 3 for the types).
- **Branch Coverage:** $4/17 \approx 23.53\%$.

Test Case 6: Device Types in "Appliances" Category

- **Input:** A graph with devices only from the "Appliances" category, including various "Kettle", "Toaster" and "Coffee Maker" types.
- **Expected Output:** A map with counts reflecting the number of each device type within the "Appliances" category.

- **Purpose:** Verifies accurate counting for each type within the "Appliances" category.
- **Branches Covered:** There are 3 types under "Appliances", this would cover 4 branches (1 for the category, 3 for the types).
- **Branch Coverage:** $4/17 \approx 23.53\%$.

Test Case 7: Device Types in "Whiteware" Category

- **Input:** A graph with devices only from the "Whiteware" category, including various "Washing Machine/Dryer", "Refrigerator/Freezer" and "Dishwasher" types.
- **Expected Output:** A map with counts reflecting the number of each device type within the "Whiteware" category.
- **Purpose:** Verifies accurate counting for each type within the "Whiteware" category.
- **Branches Covered:** There are 3 types under "Whiteware", this would cover 4 branches (1 for the category, 3 for the types).
- **Branch Coverage:** $4/17 \approx 23.53\%$.

4.3.5 outputDistribution() Testing

The `outputDistribution()` method contains no conditional branches, as it lacks direct **if** and **switch** statements. Instead, the method's complexity arises from its dependencies on other functions, specifically `countByCategory()` and `countByType()`. Detailed test cases for these functions can be found in Sections 4.2.4 and 4.3.4, respectively.

5 Mutation Testing

To test the fault tolerance of the function, the following outlines five mutants based on the `countByCategory()` pseudocode.

5.1 Mutant 1: Missing Category Handling

Removes the case for "Appliances" to test how the code handles missing categories. This mutation helps verify that the system can gracefully handle unexpected missing categories without crashing. It tests the system's default behavior and error logging capabilities in scenarios where a category is unexpectedly absent (Figure 4).

```
Map<String, Integer> countByCategory()
  INITIALISE Map of category counts (with counts starting at 0)
  GET graph of devices
  FOR EACH device in the graph
    SWITCH Category of device
      CASE Router:
        Get Router Category count from Map
        Increase Router count by 1
      END CASE
      CASE Controller:
        Get Controller Category count from Map
        Increase Controller count by 1
      END CASE
      CASE Lighting:
        Get Lighting Category count from Map
        Increase Lighting count by 1
      END CASE
      CASE Whiteware:
        Get Whiteware Category count from Map
        Increase Whiteware count by 1
      END CASE
    END SWITCH
  END FOR
  RETURN Map of category counts
```

Figure 4: Mutant (1) Pseudocode for Counting Categories

5.2 Mutant 2: Incorrect Increment

Change the increment operation for the "Lighting" category to decrement. By changing the increment to a decrement, this mutation tests the system's resilience and the accuracy of the error-checking mechanisms when data manipulation goes wrong. This can simulate a common programming error, helping ensure the system detects and handles such discrepancies (Figure 5).

```
Map<String, Integer> countByCategory()
  INITIALISE Map of category counts (with counts starting at 0)
  GET graph of devices
  FOR EACH device in the graph
    SWITCH Category of device
      CASE Router:
        Get Router Category count from Map
        Increase Router count by 1
      END CASE
      CASE Controller:
        Get Controller Category count from Map
        Increase Controller count by 1
      END CASE
      CASE Lighting:
        Get Lighting Category count from Map
        Decrease Lighting count by 1
      END CASE
      CASE Appliances:
        Get Appliances Category count from Map
        Increase Appliances count by 1
      END CASE
      CASE Whiteware:
        Get Whiteware Category count from Map
        Increase Whiteware count by 1
      END CASE
    END SWITCH
  END FOR
  RETURN Map of category counts
```

Figure 5: Mutant (2) Pseudocode for Counting Categories

5.3 Mutant 3: Category Misclassification

Swap the operations for "Router" and "Controller" Categories. Swapping operations between categories can validate if the system is robust against misclassification errors and ensures that the system can either correct or report these errors effectively (Figure 6).

```
Map<String, Integer> countByCategory()
INITIALISE Map of category counts (with counts starting at 0)
GET graph of devices
FOR EACH device in the graph
  SWITCH Category of device
    CASE Router:
      Get Controller Category count from Map
      Increase Controller count by 1
    END CASE
    CASE Controller:
      Get Router Category count from Map
      Increase Router count by 1
    END CASE
    CASE Lighting:
      Get Lighting Category count from Map
      Increas Lighting count by 1
    END CASE
    CASE Appliances:
      Get Appliances Category count from Map
      Increase Appliances count by 1
    END CASE
    CASE Whiteware:
      Get Whiteware Category count from Map
      Increase Whiteware count by 1
    END CASE
  END SWITCH
END FOR
RETURN Map of category counts
```

Figure 6: Mutant (3) Pseudocode for Counting Categories

5.4 Mutant 4: Off-by-One Error

Increase the Router count by 2 instead of 1 (Figure 7). Introducing an off-by-one error in the count is tests for boundary conditions and can help ensure that the system can handle/report small discrepancies in data handling, which are common sources of bugs.

```
Map<String, Integer> countByCategory()
  INITIALISE Map of category counts (with counts starting at 0)
  ET graph of devices
  FOR EACH device in the graph
    SWITCH Category of device
      CASE Router:
        Get Router Category count from Map
        Increase Router count by 2
      END CASE
      CASE Controller:
        Get Controller Category count from Map
        Increase Controller count by 1
      END CASE
      CASE Lighting:
        Get Lighting Category count from Map
        Increas Lighting count by 1
      END CASE
      CASE Appliances:
        Get Appliances Category count from Map
        Increase Appliances count by 1
      END CASE
      CASE Whiteware:
        Get Whiteware Category count from Map
        Increase Whiteware count by 1
      END CASE
    END SWITCH
  END FOR
  RETURN Map of category counts
```

Figure 7: Mutant (4) Pseudocode for Counting Categories

5.5 Mutant 5: Wrong Counts Initialisation

Initialise the counts incorrectly, starting each category at 1 instead of 0. Starting with an incorrect initial count tests the system's assumptions about the initial state. It is essential for verifying that the system can either correct or alert on starting conditions that might lead to inaccurate calculations or reporting (Figure 8).

```
Map<String, Integer> countByCategory()
  INITIALISE Map of category counts (with counts starting at 1)
  GET graph of devices
  FOR EACH device in the graph
    SWITCH Category of device
      CASE Router:
        Get Router Category count from Map
        Increase Router count by 1
      END CASE
      CASE Controller:
        Get Controller Category count from Map
        Increase Controller count by 1
      END CASE
      CASE Lighting:
        Get Lighting Category count from Map
        Increas Lighting count by 1
      END CASE
      CASE Appliances:
        Get Appliances Category count from Map
        Increase Appliances count by 1
      END CASE
      CASE Whiteware:
        Get Whiteware Category count from Map
        Increase Whiteware count by 2
      END CASE
    END SWITCH
  END FOR
  RETURN Map of category counts
```

Figure 8: Mutant (5) Pseudocode for Counting Categories

5.6 Mutation Test Sets and Scores

5.6.1 Test Set 1

Mutation Score $5/5 = 100\%$

- This test set indicates that the software effectively handles significant intentional faults introduced by all five mutants. It suggests that the function is robust and capable of managing a range of errors, such as missing categories, incorrect increments, misclassification, off-by-one errors, and incorrect initial counts.

Table 5.1: Test Set (1) of Inputs/Outputs for Categories and Counts Across Mutants

Device Category	**Actual Count**	**Mutant 1**	**Mutant 2**	**Mutant 3**	**Mutant 4**	**Mutant 5**
Router	2	2	2	4	4	3
Controller	4	4	4	2	4	5
Lighting	4	4	-4	4	4	5
Appliances	3	0	3	3	3	4
Whiteware	2	2	2	2	2	3

5.6.2 Test Set 2

Mutation Score $3/5 = 60\%$

- This score reflects moderate resilience to faults. The function can handle some errors well but is more significantly impacted by others. This result implies that while certain aspects of the function are secure against common errors, other aspects (likely those related to specific operations or data manipulations) need enhancement to improve robustness and error detection.

Table 5.2: Test Set (2) of Inputs/Outputs for Categories and Counts Across Mutants

Device Category	**Actual Count**	**Mutant 1**	**Mutant 2**	**Mutant 3**	**Mutant 4**	**Mutant 5**
Router	2	2	2	2	4	3
Controller	2	2	2	2	2	3
Lighting	1	1	-1	1	1	2
Appliances	0	0	0	0	0	1
Whiteware	1	1	1	1	1	1

5.6.3 Test Set 3

Mutation Score $2/5 = 40\%$

- A lower score in this test set suggests that the function does not effectively cover or mitigate against mutations as well as other scenarios. This may indicate that certain code paths or conditions are not well-guarded or well-tested against specific input or logic errors, particularly in situations where zero counts are expected but the mutations introduce discrepancies.

Table 5.3: Test Set (3) of Inputs/Outputs for Categories and Counts Across Mutants

Device Category	**Actual Count**	**Mutant 1**	**Mutant 2**	**Mutant 3**	**Mutant 4**	**Mutant 5**
Router	0	0	0	0	0	1
Controller	0	0	0	0	0	1
Lighting	0	0	0	0	0	1
Appliances	5	0	5	5	5	6
Whiteware	1	1	1	1	1	2