FUNCTIONAL SOFTWARE TEST PLAN

for

Encost Smart Graph Project

Version 1.0

Prepared by: Ben Lee
SoftFlux Engineer

SoftFlux

June 11, 2023

Contents

Revision History

Name	Date	Reason for Changes	Version

1 Introduction/Purpose

1.1 Purpose

This functional software test plan's purpose is to act as an guide for developers to follow when they are implementing software as this document follows requirements and design philosophies stated in the SRS and SDS documentation. This document will include Black-box testing, White-box testing and Mutation testing. Black-box tests will cover all high priority functional requirements. White-box testing will cover and provide pseudocode for the Device Distribution summary statistic requirement, it will utilise Branch coverage testing to develop tests that achieve 100% branch coverage. Mutation testing will be used to develop four mutants based on the White-box test pseudocode. This document is written in hopes there is enough information that is clear and concise to avoid any ambiguities, if followed correctly the tests will be able to verify that all functional requirements are met.

1.2 Document Conventions

- ESGP: Encost Smart Graph Project
- CLI: Command Line Interface
- SRS: Software Requirement Specifications
- SDS: Software Design Specifications

1.3 Intended Audience and Reading Suggestions

- Developer: A developer who may require guidance for writing code that will pass the tests outlined in this document.
- Tester: A tester may require guidance for writing tests that are based off the outlined information in this document.

1.4 Project Scope

Encost uses the ESGP project to better understand how its devices are being used and linked in New Zealand households. The Encost Smart Homes Dataset, a dataset including details about 100 New Zealand homes and the Encost devices being used, will be able to be visualised and summarised through the ESGP project. The 100 New Zealand households that agreed to have their data collected will also receive the project so they may view a visualisation of the Encost Smart households Dataset.

2 Specialized Requirements Specification

2.1 Classes and Functions

Categorising Users Class
This Class will be added and be used for the user type selection prompt and the Encost login prompt. One function that will be added is the checkUserType function, the input will be the user selection, being either "1" or "2", it will return "invalid" if the user didn't select the valid inputs, and return "Community" for option "1" and "Encost-Unverified" for option "2". The user type must be saved for the input was selected, so getter and setter functions must be used for this. Option "1" will prompt the user with the Community ESGP Feature options. If the User selects option "2", they will be prompted with the Encost login. For the user to become verified and view the Encost feature options they must have a valid username and password. For this two functions must be created, being checkUsername and checkPassword. The purpose for this is that the system can compare the inputs to the saved username and password pairs. checkPassword, must also encrypt the input password to match the encrypted passwords that are saved in the system.

ESGP Feature Options class
This Class already exists as stated in the SDS, however this class must have an additional function named checkOption, this takes an input from the users keypress. The input must be corresponding to one of the options given from the prompted, it will then return whether the input was valid, or it will return a string that corresponds to the feature option selected.

Dataset class
This Class already exists as stated in the SDS, however this class must have an additional function named checkFormat, this takes the number of words in line data from the data set. It will check that this number is valid as the number given will determine if the line is formatted properly. The output will back true or false depending on if it valid or not. One use of this will be when an Encost member wants to use a custom dataset, checkFormat will indicate if this dataset can be used or not.

Device class
This Class already exists as stated in the SDS, however this class must have an additional function named isDeviceValid, this takes a device's name as input and references it against all Encost devices. If the device name is not found to match any of the Encost devices, the return value will be false. This is used to ensure that all devices will be calculated and displayed correctly.

GraphStream class
This Class already exists as stated in the SDS, however this class must have an additional functions. The additional functions will be used to check if any input into the graph is valid. These functions include, isDeviceValid, doesConnectionExist, isConnectionValid and isHouseholdValid. These are all necessary checks if the information on the graph is to be displayed correctly.

3 Black-box Testing

3.1 Categorising Users

Description
This test is being used to test that the user input is a valid key input. In this case any input outside of "1" or "2" should return invalid.

Testing Methodology
Automated, using Selenium.

Equivalence Partitioning		
Invalid	Valid	Invalid
<1	*1-2*	*>2*

Interpretation

- Test Level: Unit test.
- Case 1: Input character being less than 1, including all alphabetic characters, will be invalid as the only options are 1 (Community Member) and 2 (Encost Member).
- Case 2: Input character between 1 and 2, will be valid as it is either 1 (Community Member) and 2 (Encost Member).
- Case 3: Input character being greater than 2, including all alphabetic characters, will be invalid as the only options are 1 (Community Member) and 2 (Encost Member).

Justification
Equivalence Partitioning is being used to limit input possibilities, which is large number due to the input being key presses to the CLI, grouping all input outside of "1" or "2" will decrease the number of tests needed. For this case it shows that the user can only input "1" or "2" and anything other key press should result as an invalid input. This demonstrates the limitations of the inputs for Categorizing Users.

3.2 ESGP Account Login

Description
This test is being used to check that the username and password being passed by the user are valid. In this case any input that doesn't match the stored username password pairs in the system.

Testing Methodology
Automated, using Selenium.

Decision Table				
Conditions	**Case 1**	**Case 2**	**Case 3**	**Case 4**
Username	*Invalid*	*Invalid*	*Valid*	*Valid*
Password	*Invalid*	*Valid*	*Invalid*	*Valid*
Output	*Fail*	*Fail*	*Fail*	*Success*

Interpretation

- Test Level: Unit test.
- Case 1: Username and Password are invalid, an invalid Username will prompt the user to re-enter a Username, on the third failed attempt return the user to the welcome prompt.
- Case 2: Username is invalid and Password is Valid, an invalid Username will prompt the user to re-enter a Username, on the third failed attempt return the user to the welcome prompt.
- Case 3: Username is valid and Password is invalid, a valid Username will prompt the user to enter a Password. An invalid Password will prompt the user to re-enter a Username, on the third failed attempt return the user to the welcome prompt.
- Case 4: Username and Password are valid, this will prompt the user with the ESGP Feature Options and update user type to "verified".

Justification
A Decision table is being used to show a combination of inputs will give predetermined outputs. In this case a combination of invalid and valid usernames and passwords. This ensures tests are created with all possibilities in mind.

3.3 ESGP Feature Options

Description
This test is being used to test that the user input is a valid key input. In this case any input from a community member outside of "1" should return invalid. For an Encost member any input outside of "1", "2" or "3" should return invalid.

Testing Methodology
Automated, using Selenium.

Equivalence Partitioning					
Community Member			**Encost Member**		
Invalid	Valid	Invalid	Invalid	Valid	Invalid
<1	*1*	*>1*	*<1*	*1-3*	*>3*

Interpretation

- Test Level: Unit test.
- Case 1: Input character being less than 1, including all alphabetic characters, will be invalid as the only option is 1 (visualise the data set as a graph) for Community Members.
- Case 2: Input character being 1, will be valid as 1 (visualise the data set as a graph) and will open the graph UI.
- Case 3: Input character being greater than 1, including all alphabetic characters, will be invalid as the only option is 1 (visualise the data set as a graph) for Community Members.
- Case 4: Input character being less than 1, including all alphabetic characters, will be invalid as the only options are 1 (visualise the data set as a graph), 2 (load custom dataset) and 3 (summary statistics) for Encost Members.
- Case 5: Input character being between 1 and 3, will be valid as it 1 (visualise the data set as a graph), 2 (load custom dataset) and 3 (summary statistics) for Encost Members. 1 will open the Graph UI, 2 prompt the User to enter custom data set file path, 3 will display summary statistics to the CLI.
- Case 6: Input character being greater than 3, including all alphabetic characters, will be invalid as the only options are 1 (visualise the data set as a graph), 2 (load custom dataset) and 3 (summary statistics) for Encost Members.

Justification
Equivalence Partitioning is being used to limit all input possibilities, which is large number due to the input being key presses, to a smaller range by grouping tests. For this case it would mean Community Members can only input ”1” and anything other key press should result as an invalid input. This is similar for Encost Members, but they can use ”1”, ”2” or ”3”. This demonstrates the limitations of the inputs for the ESGP Feature Options.

3.4 Loading the Encost Smart Homes Dataset

Description
This test is being used to test that the dataset format is valid. When a line is being read from the dataset it should be split and if it isn’t split in 7 or 8 values it should be invalid.

Testing Methodology
Automated, using Selenium.

Boundary Value Analysis		
Invalid	Valid	Invalid
<7	*7-8*	*>8*

Interpretation

- Test Level: Integration test. Using the Encost Smart Home DataSet and Custom Dataset components.
- Case 1: Items in the split lineData array being less than 7, will be invalid as there must be 7 or 8 items in the array to conform to the format.
- Case 2: Items in the split lineData array between 7 and 8, will be valid as there is 7 or 8 items in the array to conform to the format, as Device Router Connection can be null to make 7 readable items.
- Case 3: Items in the split lineData array being greater than 8, will be invalid as there must be 7 or 8 items in the array to conform to the format.

Justification
Boundary Value Analysis is used to show, that when building a graph every requirement must be met or the insertion should be invalid. This also helps to group all invalid inputs into Causes, which should result in the overarching Effect. Equivalence Partitioning is a better choice for this over the other techniques, another option could have been a decision table, however this ultimately wouldn't be practical as the different users types are displayed different options.

3.5 Categorising Smart Home Devices

Description
This test is being used to check an input device name is valid. When a device is being passed the name should exist in the Encost device list, anything else should return invalid.

Testing Methodology
Automated, using Selenium.

Error Guessing		
Test Case	**Input (Device Type)**	**Output**
Case 1	*" "*	*Invalid*
Case 2	*"television"*	*Invalid*
Case 3	*"router"*	*valid*
Case 4	*"Router"*	*valid*

Interpretation

- Test Level: Unit test.
- Case 1: An empty string is used for the input for CaterogiseDevice function, this would result in an invalid output as there is no empty string as a device type and will not belong to a device category.
- Case 2: A device that is not one of Encosts device types is used for the input for CaterogiseDevice function, this would result in an invalid output as this doesn't belong to a Device category.
- Case 3: An incorrectly capitalised string is used for the input for CaterogiseDevice function, this would result in a valid output as all device types will be decapitalised and the type belongs to a Device category.
- Case 4: An device type that is an Encost device type is used for the input for CaterogiseDevice function, this would result in a valid output as this belongs to a Device category.

Justification
Error Guessing is used to show that only Encost devices will be accepted inputs when categorising device types. This is helpful, because it will indicate whether an input should be valid or invalid, which will help to diagnose any potential errors.

3.6 Building a Graph Data Type

Description
This test is being used to show how you can group possible errors into cause groups, that all return the same effect.

Testing Methodology
Automated, using Selenium.

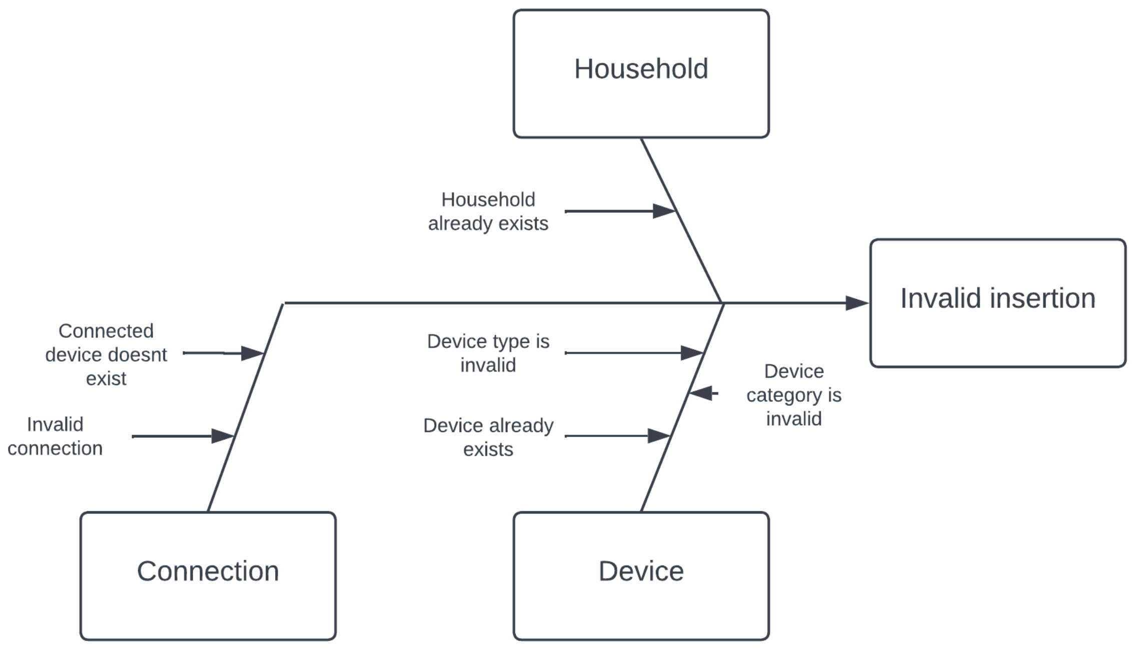

Figure 3.1: Cause-Effect Flowchart

Interpretation

- Test Level: Integration test. Using the Encost Smart Home DataSet and GraphStream Library components.
- Household: This Cause is used to group any errors that could result in an invalid insertion into the Graph, this includes a household that may already exist in the graph, as there should be no duplicate households.
- Connection: This Cause is used to group any errors that could result in an invalid insertion into the Graph, this includes invalid connections or a connection not existing. This can be caused by a device having a connection to another but that device doesn't exist in the graph yet.
- Device: This Cause is used to group any errors that could result in an invalid insertion into the Graph. This includes device type or category being invalid or a device already being inserted, this will ensure duplicates and invalid devices aren't inserted.
- Invalid insertion: This is the Effect that these Causes result in.

Justification
A Cause-Effect Flow-chart is used to show, that when building a graph every requirement must be met or the insertion should be invalid. This also helps to group all invalid inputs into Causes, which should result in the overarching Effect, this can be used to protect the graph by ensuring no invalid insertions are made.

3.7 Graph Visualisation

Description
This test is being used to check that all inputs are true. When inserting a device into the graph, all the checks conditions should be variables stored in the Device object, see the SDS document for clarification.

Testing Methodology
Automated, using Selenium.

Decision Table						
Conditions	**Case 1**	**Case 2**	**Case 3**	**Case 4**	**Case 5**	***Case 6***
GraphStream Library	*F*	*T*	*T*	*T*	*T*	***T***
Show All Nodes	*T*	*F*	*T*	*T*	*T*	*T*
Show All Edges	*T*	*T*	*F*	*T*	*T*	*T*
Distinguish Categories	*T*	*T*	*T*	*F*	*T*	*T*
Illustrate send/receive	*T*	*T*	*T*	*T*	*F*	*T*
Output	*Fail*	*Fail*	*Fail*	*Fail*	*Fail*	*Success*

Interpretation

- Test Level: Integration test. Using the GraphStream Library, Graph Interface and Device components.
- Case 1-5: In each of these cases there is a feature that is not true, so the outcome is deemed as a fail.
- Case 6: In the case every feature works the outcome is deemed as a success.

Justification
A Decision table is being used to show a comprehensive set of test cases can cover a wide range of combinations for inputs. This is necessary to meet the requirements for Graph Visualisation.

3.8 Calculating Device Distribution

Description
This test is being used to check a large variety of possible inputs. A list of devices are inputs and the output will be the counts of each device type and category.

Testing Methodology
Automated, using Selenium.

Equivalence Partitioning	
Input	**Output**
Empty input list	*type counters for all devices will be 0*
Input list with one device	*a single type counter will return as 1*
Input list with multiple devices with one unique category and one type	*a single type counter will return greater than 0*
Input list with multiple devices with one category and multiple device types	*multiple type counters will return greater than 0*
Input list with multiple devices with multiple categories and device types	*multiple type counters will return greater than 0*
Input list with multiple devices that only contains devices that don't match any Encost device type	*type counters for all devices will be 0*
Input list with multiple devices that contains devices that have empty strings for either id, category and/or type	*multiple type counters will return greater than 0*
Input list with devices that have duplicate device ids	*multiple type counters will return greater than 0*

Interpretation

- Test Level: Unit test.
- Inputs: These inputs represent groups of the vast number of possible cases that are needed to test this calculation.
- Outputs: These outputs represent the result of the input groupings, this means an actual input will fall within one of these groupings and the corresponding output will always be true.

Justification
Equivalence Partitioning is being used to limit a wide range of test case possibilities to a smaller range by grouping tests. There are 4 Categories with ten different types, with no limit of how many devices can be in a dataset, so we use Equivalence Partitioning to reduce these possible tests to 8 cases. If implemented correctly, outputs should return in a similar fashion to each test case.

4 White-box testing

4.1 Calculating Device Distribution Pseudocode

Algorithm 1 CalculateDeviceDistribution(List<Device> deviceList)

```
Set up LinkedHashMap for counters
Set the key for counters to be each category and set the value to new LinkedHashMaps for device types
put all device types into their respective category LinkedHashMap value
Set up an array list for inserted device id's
for Each device in the devise list do
    Get the device's id, category and type
    Get the device's type
    if device id OR category OR type are empty then
        Continue to the next device
    end if
    if Inserted device list contains device id then
        Continue to the next device
    end if
    Add device's id to the inserted device list
    Update the corresponding category's LinkedHashMap using the device type key.
end for
Return the counters
```

Justification
The reason for this design is it allows for sufficient white-box testing, as the many mutations can be made to the source code for numerous variations. Using LinkedHashMaps with fixed Keys will work well as Encost only record devices of certain types in their dataset. Using LinkedHashMaps also allow for counters to be ordered and easily incremented while being associated with a certain category and type.

4.2 Branch Coverage Testing

Algorithm 2 TestCalculateDeviceDistribution()

```
//Inputs:
List<Device> deviceListExpected = Arrays.asList(
new Device("device 1", "Category1", "Type1"),
new Device("device 2", "Category1", "Type2"),
new Device("device 3", "Category2", "Type2"),
new Device("device 4", "Category2", "Type3"),
new Device("device 5", "Category3", "Type1"),
new Device("device 6", "Category3", "Type1") );
List< Device >deviceListBranch = Arrays.asList(
new Device("device 1", "Category1", "Type1"),
new Device("device 2", "Category1", "Type2"),
new Device("device 3", "Category2", "Type2"),
new Device("device 4", "Category2", "Type3"),
new Device("device 5", "Category3", "Type1"),
new Device("device 6", "Category3", "Type1"),
new Device("device 5", "Category3", "Type2"),
new Device("", "", "") );

//Get Counters for expected and Branch device lists
Map<String,Map<String,Integer>>countersExpected=new LinkedHashMap<>();
Map<String,Map<String,Integer>> countersBranch=new LinkedHashMap<>();
countersExpected = CalculateDeviceDistribution(deviceListExpected);
countersBranch = CalculateDeviceDistribution(deviceListBranch);
//Compare Counters, should return equal if all branches were used
assertEquals(countersExpected, countersBranch);
```

Interpretation

- Test Level: Integration test. Using the Encost Smart Home Dataset and Frontend components.
- Inputs: The deviceListExpected input is how counters will look with no errors in the list, and deviceListBranch will show how counters will look with errors, the errors in the list will result in 100% branch coverage. If the function works correctly the counters will match.

Justification
The input device list deviceListBranch, will obtain 100% branch coverage, as it has a duplicate device, which will enter the branch "if Inserted device list contains device id". This list also has a device with empty strings which will enter the only other branch being "if device id OR category OR type are empty". The other devices in the list will cover all other branches and will return the correct device counters. The algorithm also has no breaks which means even when an invalid device is entered, the algorithm won't break and still return the correct value.

5 Mutation Testing

5.1 Mutant #1

input
List<Device>deviceList = Arrays.asList(
new Device("1", "Category1", "Type1"), new Device("2", "Category1", "Type2"),
new Device("3", "Category2", "Type2"), new Device("4", "Category2", "Type3"),
new Device("5", "Category3", "Type1"), new Device("6", "Category3", "Type1"),
new Device("7", "Category3", "Type2"), new Device("2", "Category1", "Type2"));

Mutant Testing	
Original	**Expected Output**

// Iterate through the device list and update the counters
for (Device device : deviceList) {
String id = device.getID();
String category = device.getCategory();
String type = device.getType();
//if id OR category OR type is empty, continue to the next device
if(id.isEmpty() || category.isEmpty() || type.isEmpty()){
continue;
}
//if the device already exists, continue to the next device
if(insertedDevices.contains(id)){
continue;
}
insertedDevices.add(id);
// Update the type counter
counters.get(category).compute(type, (k, v) ->v == 0 ? 1 : v + 1);
}
	Category: Category1
	- Type1: 1
	- Type2: 1
	- Type3: 0
	Total: 2
	
	Category: Category2
	- Type1: 0
	- Type2: 1
	- Type3: 1
	Total: 2
	
	Category: Category3
	- Type1: 2
	- Type2: 1
	- Type3: 0
	Total: 3

Mutation 1	**Actual Output**

// Iterate through the device list and update the counters
for (Device device : deviceList) {
String id = device.getID();
String category = device.getCategory();
String type = device.getType();
//if id OR category OR type is empty, continue to the next device
if(id.isEmpty() || category.isEmpty() || type.isEmpty()){
continue;
}
//if the device already exists, continue to the next device
if(insertedDevices.contains(id)){
continue;
}
 	
// Update the type counter
counters.get(category).compute(type, (k, v) ->v == 0 ? 1 : v + 1);
}
	Category: Category1
	- Type1: 1
	- Type2: 2
	- Type3: 0
	Total: 3
	
	Category: Category2
	- Type1: 0
	- Type2: 1
	- Type3: 1
	Total: 2
	
	Category: Category3
	- Type1: 2
	- Type2: 1
	- Type3: 0
	Total: 3

Interpretation

- Mutant 1: Mutation 1 removes the line that adds the current device to the inserted list.

Justification
Mutation 1 won’t save any devices to the inserted device list. This means if the input has any duplicates they will still be inserted, which will be caught if we are comparing this to the expected output for the original algorithm. Mutation 1 will allow us to check if any duplicates are affecting the counters.

5.2 Mutant #2

input
List<Device>deviceList = Arrays.asList(
new Device(”1”, ”Category1”, ”Type1”), new Device(”2”, ”Category1”, ”Type2”),
new Device(”3”, ”Category2”, ”Type2”), new Device(”4”, ”Category2”, ”Type3”),
new Device(”5”, ”Category3”, ”Type1”), new Device(”6”, ”Category3”, ”Type1”),
new Device(”7”, ”Category3”, ”Type2”), new Device(”2”, ”Category1”, ”Type2”));

Mutant Testing	
Original	**Expected Output**
// Iterate through the device list and update the counters	Category: Category1
for (Device device : deviceList) {	- Type1: 1
String id = device.getID();	- Type2: 1
String category = device.getCategory();	- Type3: 0
String type = device.getType();	Total: 2
//if id OR category OR type is empty, continue to the next device	
if(id.isEmpty() || category.isEmpty() || type.isEmpty()){	Category: Category2
continue;	- Type1: 0
}	- Type2: 1
//if the device already exists, continue to the next device	- Type3: 1
if(insertedDevices.contains(id)){	Total: 2
continue;	
}	Category: Category3
insertedDevices.add(id);	- Type1: 2
// Update the type counter	- Type2: 1
counters.get(category).compute(type, (k, v) ->v == 0 ? 1 : v + 1);	- Type3: 0
}	Total: 3
Mutation 2	**Actual Output**
// Iterate through the device list and update the counters	Category: Category1
for (Device device : deviceList) {	- Type1: 1
String id = device.getID();	- Type2: 1
String category = device.getCategory();	- Type3: 0
String type = device.getType();	Total: 2
//if id OR category OR type is empty, continue to the next device	
if(id.isEmpty() || category.isEmpty() || type.isEmpty()){	Category: Category2
continue;	- Type1: 0
}	- Type2: 1
//if the device already exists, continue to the next device	- Type3: 1
if(insertedDevices.contains(id)){	Total: 2
continue;	
}	Category: Category3
insertedDevices.add(id);	- Type1: 1
// Update the type counter	- Type2: 1
counters.get(category).compute(type, (k, v) ->v != 0 ? 1 : v + 1);	- Type3: 0
}	Total: 2

Interpretation

- Mutant 2: Mutation 2 changes the compute function, as any counter greater than 1 will result in 1.

Justification
Mutation 2 will set any device type value that's not 0 equal to 1. This means the type counters will never be over 1, however if the device list input doesn't have devices of the same type the mutation won't be caught as the outputs would be no different. In most cases the device list will have multiple devices of the same type, so this will be caught for most cases. For most device lists it allows us to see if the counter value is being calculated and returned correctly.

5.3 Mutant #3

input
List<Device>deviceList = Arrays.asList(
new Device(”1”, ”Category1”, ”Type1”), new Device(”2”, ”Category1”, ”Type2”),
new Device(”3”, ”Category2”, ”Type2”), new Device(”4”, ”Category2”, ”Type3”),
new Device(”5”, ”Category3”, ”Type1”), new Device(”6”, ”Category3”, ”Type1”),
new Device(”7”, ”Category3”, ”Type2”), new Device(”2”, ”Category1”, ”Type2”));

Mutant Testing	
Original	**Expected Output**

// Iterate through the device list and update the counters
for (Device device : deviceList) {
String id = device.getID();
String category = device.getCategory();
String type = device.getType();
//if id OR category OR type is empty, continue to the next device
if(id.isEmpty() || category.isEmpty() || type.isEmpty()){
continue;
}
//if the device already exists, continue to the next device
if(insertedDevices.contains(id)){
continue;
}
insertedDevices.add(id);
// Update the type counter
counters.get(category).compute(type, (k, v) ->v == 0 ? 1 : v + 1);
}
	Category: Category1
	- Type1: 1
	- Type2: 1
	- Type3: 0
	Total: 2
	
	Category: Category2
	- Type1: 0
	- Type2: 1
	- Type3: 1
	Total: 2
	
	Category: Category3
	- Type1: 2
	- Type2: 1
	- Type3: 0
	Total: 3

Mutation 3	**Actual Output**

// Iterate through the device list and update the counters
for (Device device : deviceList) {
String id = device.getID();
String category = device.getCategory();
String type = device.getType();
//if id OR category OR type is empty, continue to the next device
if(id.isEmpty() || category.isEmpty() || type.isEmpty()){
continue;
}
//if the device already exists, continue to the next device
if(insertedDevices.contains(id)){
continue;
}
insertedDevices.add(id);
// Update the type counter
counters.get(category).compute(type, (k, v) ->v == 0 ? 1 : v + 2);
}
	Category: Category1
	- Type1: 1
	- Type2: 1
	- Type3: 0
	Total: 2
	
	Category: Category2
	- Type1: 0
	- Type2: 1
	- Type3: 1
	Total: 2
	
	Category: Category3
	- Type1: 3
	- Type2: 1
	- Type3: 0
	Total: 4

Interpretation

- Mutant 3: Mutation 3 changes the addition for the counters in the compute function by adding 2 every time a new type counter is added that's greater than 1.

Justification
Mutation 3 will increase the device type value by 2 every time the value is over 1. This shares a similar problem to mutation 2 as it will only be caught if their are multiple devices of the same type, if there isn't, this addition of 2 will never happen. However, in most cases the device list will have multiple devices of the same type, so for most device lists it allows us to see if the counter value is being calculated correctly.

5.4 Mutant #4

input
List<Device>deviceList = Arrays.asList(
new Device("1", "Category1", "Type1"), new Device("2", "Category1", "Type2"),
new Device("3", "Category2", "Type2"), new Device("4", "Category2", "Type3"),
new Device("5", "Category3", "Type1"), new Device("6", "Category3", "Type1"),
new Device("7", "Category3", "Type2"), new Device("2", "Category1", "Type2"));

Mutant Testing	
Original	**Expected Output**

// Iterate through the device list and update the counters
for (Device device : deviceList) {
String id = device.getID();
String category = device.getCategory();
String type = device.getType();
//if id OR category OR type is empty, continue to the next device
if(id.isEmpty() || category.isEmpty() || type.isEmpty()){
continue;
}
//if the device already exists, continue to the next device
if(insertedDevices.contains(id)){
continue;
}
insertedDevices.add(id);
// Update the type counter
counters.get(category).compute(type, (k, v) ->v == 0 ? 1 : v + 1);
}
	Category: Category1
	- Type1: 1
	- Type2: 1
	- Type3: 0
	Total: 2
	
	Category: Category2
	- Type1: 0
	- Type2: 1
	- Type3: 1
	Total: 2
	
	Category: Category3
	- Type1: 2
	- Type2: 1
	- Type3: 0
	Total: 3

Mutation 4	**Actual Output**

// Iterate through the device list and update the counters
for (Device device : deviceList) {
String id = device.getType();
String category = device.getCategory();
String type = device.getType();
//if id OR category OR type is empty, continue to the next device
if(id.isEmpty() || category.isEmpty() || type.isEmpty()){
continue;
}
//if the device already exists, continue to the next device
if(insertedDevices.contains(id)){
continue;
}
insertedDevices.add(id);
// Update the type counter
counters.get(category).compute(type, (k, v) ->v == 0 ? 1 : v + 1);
}
	Category: Category1
	- Type1: 1
	- Type2: 1
	- Type3: 0
	Total: 2
	
	Category: Category2
	- Type1: 0
	- Type2: 0
	- Type3: 1
	Total: 1
	
	Category: Category3
	- Type1: 0
	- Type2: 0
	- Type3: 0
	Total: 0

Interpretation

- Mutant 4: Mutation 4 changes the stored id being the devices id to it storing the device type.

Justification

Mutation 4 causes the stored device id to be extracted as the device type. This will cause any device of the same type to be skipped over as the id check will be checking inserted types not inserted ids. This makes sure the counters are being added up correctly with no missing devices.

5.5 Mutation Score

Mutant Testing Input-Output Pairs	
Set 1 Input	**Set 2 Input**
List<Device>deviceList = Arrays.asList(new Device("1", "Category1", "Type1"), new Device("3", "Category2", "Type2"), new Device("4", "Category2", "Type3"), new Device("5", "Category3", "Type1"), new Device("7", "Category3", "Type2"), new Device("8", "Category2", "Type1"), new Device("6", "Category3", "Type1"), new Device("2", "Category1", "Type3"));	List<Device>deviceList = Arrays.asList(new Device("1", "Category1", "Type1"), new Device("2", "Category2", "Type2"), new Device("4", "Category2", "Type3"), new Device("5", "Category3", "Type1"), new Device("6", "Category1", "Type1"), new Device("3", "Category2", "Type3"), new Device("5", "Category3", "Type1"), new Device("2", "Category2", "Type2"));
Set 1 Output	**Set 2 Output**
Category: Category1 - Type1: 1 - Type2: 0 - Type3: 1 Total: 2 Category: Category2 - Type1: 1 - Type2: 1 - Type3: 1 Total: 3 Category: Category3 - Type1: 2 - Type2: 1 - Type3: 0 Total: 3	Category: Category1 - Type1: 2 - Type2: 0 - Type3: 0 Total: 2 Category: Category2 - Type1: 0 - Type2: 1 - Type3: 2 Total: 3 Category: Category3 - Type1: 1 - Type2: 0 - Type3: 0 Total: 1

Mutant Testing	
Original	**Mutation 1**

// Iterate through the device list and update the counters
for (Device device : deviceList) {
String id = device.getID();
String category = device.getCategory();
String type = device.getType();
//if id OR category OR type is empty, continue to the next device
if(id.isEmpty() —— category.isEmpty() —— type.isEmpty()){
continue;
}
//if the device already exists, continue to the next device
if(insertedDevices.contains(id)){
continue;
}
insertedDevices.add(id);
 	
// Update the type counter
counters.get(category).compute(type, (k, v) ->v == 0 ? 1 : v + 1);
}
	// Iterate through the device list and update the counters
	for (Device device : deviceList) {
	String id = device.getID();
	String category = device.getCategory();
	String type = device.getType();
	//if id OR category OR type is empty, continue to the next device
	if(id.isEmpty() —— category.isEmpty() —— type.isEmpty()){
	continue;
	}
	//if the device already exists, continue to the next device
	if(insertedDevices.contains(id)){
	continue;
	}
	
	// Update the type counter
	counters.get(category).compute(type, (k, v) ->v == 0 ? 1 : v + 1);
	}

Mutation 2	**Mutation 3**

// Iterate through the device list and update the counters
for (Device device : deviceList) {
String id = device.getID();
String category = device.getCategory();
String type = device.getType();
//if id OR category OR type is empty, continue to the next device
if(id.isEmpty() —— category.isEmpty() —— type.isEmpty()){
continue;
}
//if the device already exists, continue to the next device
if(insertedDevices.contains(id)){
continue;
}
insertedDevices.add(id);
 	
// Update the type counter
counters.get(category).compute(type, (k, v) ->v != 0 ? 1 : v + 1);
}
	// Iterate through the device list and update the counters
	for (Device device : deviceList) {
	String id = device.getID();
	String category = device.getCategory();
	String type = device.getType();
	//if id OR category OR type is empty, continue to the next device
	if(id.isEmpty() —— category.isEmpty() —— type.isEmpty()){
	continue;
	}
	
	//if the device already exists, continue to the next device
	
	if(insertedDevices.contains(id)){
	continue;
	}
	insertedDevices.add(id);
	
	// Update the type counter
	counters.get(category).compute(type, (k, v) ->v == 0 ? 1 : v + 2);
	}

Mutant Score	
(Caught Mutants / Total Mutants) x 100 = Mutant Score	
Set 1	(2 / 3) x 100 = 67%
Set 2	(3 / 3) x 100 = 100%

Interpretation

- Set 1: Set 1's input was used to show that in some cases the Mutations may not be caught, this is acceptable as all mutations are caught in Set 2.
- Set 2: Set 2's input was used to show that all mutants can be caught, even when invalid devices are in the input.

Justification
Set 1 doesn't catch all mutants as the input for this set doesn't have any duplicates so Mutation 1 won't be caught. Whereas Set 2 does have duplicates, which will result in a mutant score of 100%. These inputs are used to demonstrate that Mutants won't always be caught, it depends on the inputs, but when they aren't caught, the data is still collected correctly. The mutant tests ensures the test suite is effective by introducing defects into the code and determine if the test suite can detect them. This can help to remove weakness and improve quality of the source code.

6 Conclusion

In conclusion, this functional software test plan has provided an overview of the functional tests that will be used to test the software. This document has been written to meet the requirements and design specifications from the SRS and SDS documents. If the software is designed according to this test plan, all functional components will fulfill the users needs with no bugs or errors.