FUNCTIONAL SOFTWARE TEST PLAN

for

Encost Smart Graph Project

Version 1.0

Prepared by: Student 4
SoftFlux Engineer

SoftFlux

May 6, 2023

Contents

Revision History

Name	Date	Reason for Changes	Version

1 Introduction/Purpose

1.1 Purpose

The purpose of this document is to specify and describe a Test Plan Specification for the Encost Smart Graph Project (ESGP), which will be based on component and system details outlined in a chosen Software Design Specification (SDS) document. This document will consist of black-box testing outlines for all High Priority requirements found in the original Software Requirements Specification (SRS), white-box testing for a chosen individual software component, and mutation testing for that software component. Clarifications or design modifications (due to ambiguities or errors) will be addressed.

All tests described in this document can be implemented with jUnit.

The third SDS document provided is being followed for this document.

1.2 Document Conventions

This document uses the following conventions:

ESGP: Encost Smart Graph Project

SRS: Software Requirements Specification

SDS: Software Requirements Specification

CLI: Command Line Interface

REQ: Requirement

SHA-256: Secure Hashing Algorithm 256

1.3 Intended Audience and Reading Suggestions

This document may serve as a tool and guide for any individual developer, tester or project manager, with the following use-cases described:

- Developer: A developer may use this document, as well as the corresponding testing suites, to implement the ESGP software, with well-defined expectations for how individual components will work and what requirements they have to meet.
- Tester: A tester can use this document to guide further refining of the testing suite based on changing requirements, ensuring that this document's textual description of tests is consistent with the programmatic description.
- Project manager: To aide in tracking progress of the ESGP software's development, and project manager may employ this document and the test suite to verify what requirements have been met. This can serve to identify what parts of the ESGP software system need work.

1.4 Project Scope

Encost is a new and emerging Smart Home development company. They manufacture a series of Smart Home and IoT solutions, including Wifi Routers, Smart Hubs and Controllers, Smart Light Bulbs, Smart Appliances, and Smart Whiteware. Encost is interested in investigating how their smart devices are being used and connected within households across New Zealand. They have worked, in partnership with energy companies and their users, to gather information about the smart devices that have been in use in 100 New Zealand homes between April 2020 and April 2022 (called the Encost Smart Homes Dataset).

The Encost Smart Graph Project (ESGP) is a software system that enables the visualisation of Encost's devices using a graph data structure. When provided with the Encost Smart Homes Dataset, ESGP enables users to view all of the devices in the dataset, along with their connection to one another. It also provides verified users with summary statistics on device distribution, location, and connectivity.

2 Specialized Requirements Specification

2.1 SRS 4.1.3 - Storing the user type

The SDS has incorrectly described the implementation for storing the user's type. The *ConsoleApp* class has a *userIsVerified* boolean field. The client has allowed for field refactoring to take place. This does not allow for representing properly whether a user is a community user. The clarified requirements are as follows:

- A *UserType* enum shall be used instead within the *ConsoleApp* class. This enum shall have these values: *Community*, *EncostVerified*, *EncostUnverified*.
- An *ApplicationState* static class will be defined to store the current UserType value, which must also be static. The field itself shall be private, and public getter and setter methods shall be created to retrieve and modify the *UserType*. The *ConsoleApp* class will then use this class to retrieve and store the user's type.

2.2 SRS 5.2.1.c/d - Password storage and verification

The SDS did not specify the encryption method nor storage method for the ten pre-defined passwords, nor how passwords input by the user are to be checked against the stored passwords. The client indicated this could be specified in this document. The clarified requirements are as follows:

- The pre-defined usernames and passwords will be stored as strings within code, in a private hash table in the *UserVerifier* class, with passwords hashed with any valid SHA-256 hashing tool prior to being defined.
- User input passwords will be checked against the pre-defined passwords using the Java standard library's *MessageDigest* class, with a string representation of the SHA-256 output compared validated against the existing hash table. This can be performed in the *UserVerifier*'s *verifyPassword* method.

2.3 SRS 4.2 REQ-2 - Validating the username and password

The *UserVerifier* class defined in the SDS separates the username and password validation into two methods: *verifyUsername* and *verifyPassword*. This is an odd separation of two closely related concerns, and it makes the testing of logging in more difficult, as the *ConsoleApp*'s *verifyUser* method makes a separate call to each, mixing processing

of standard input with this validation logic. These modifications have been allowed by the client. The clarified requirements are as follows:

- *UserVerifier*'s *verifyUsername* and *verifyPassword* methods shall be replaced with a single *verifyCredentials* method, accepting a *username* string and *password* string. This method will check first that the given username matches one in the credentials hash table, returning *false* if there is no match. The method will then check if the given password matches the password for the matched username, returning *true* if this is the case.

2.4 SRS 4.4 - Loading the Encost Smart Homes Dataset

The SDS defines a *FileParser* class with a *parseFile* method, which accepts a file path string as a parameter. Unit tests generally do not interact with the file system, so this parameter needs modification to improve testability. The client agreed, and has allowed for modification. The clarified requirements are as follows:

- The *filePath* string parameter for the *FileParser*'s *parseFile* method shall be replaced with a *fileStream* parameter of type *BufferedReader*. This will enable for an actual file stream to be passed to this method when the application is running, but also a mock stream during testing, which will read from in-memory data (such as an array of strings).

2.5 SRS 3.1 - Console user interaction

The SDS did not provide explicit examples or mock-ups of a user's interaction with the application via a console. After discussion with the client, an allowance was made to provide and adapt these mock-ups from an SDS submitted that was not ultimately accepted by the client.

2.5.1 The Welcome UI

This is what the user sees first upon opening the application. This page may go either to the Feature Options section straight away for the community user, or to the Account Login section to verify a potential Encost user.

```
Welcome to Encost Smart Graph Project (ESGP)!
-----------------------------
Select your profile type:

1. Community member
2. Encost user
<user input>
```

2.5.2 Account Login UI

The user will enter a username and password, which will be matched (or not) against the pre-defined credential pairs. The password entry should make use of Java's built-in character masking feature to ensure privacy.

Login success

```
Account Login
-----------------------------
Enter your username:
<username>
Enter your password:
••••••••••••

Successfully logged in!
```

Login failure

```
Account Login
-----------------------------
Enter your username:
<username>
Enter your password:
••••••••••••

Those credentials don't match our records!
Would you like to try again?

1. Try again
2. Go back
```

2.5.3 Feature options UI

The FeatureOptionsPage provides the user with different options in relation to the default or custom dataset. Community users will only have access to graph visualisation of the default dataset, while verified Encost users can access the visualisation, as well as custom dataset loading and summary statistics.

Community user or Encost unverified

```
Feature Options
-----------------------------
A dataset has been loaded! What do you want to do?

1. Visualise graph of dataset
2. Exit
```

Encost user

If a custom dataset has already been loaded, then the Feature Options section should display a *restore dataset* prompt. This prevents the user from having to exit the application only to open it and use the default dataset.

```
Feature Options
-----------------------------
A dataset has been loaded! What do you want to do?

1. Load custom dataset
2. Visualise graph of dataset
3. View summary statistics
4. Exit
```

Encost user after loading custom dataset

```
Feature Options
-----------------------------
A dataset has been loaded! What do you want to do?

1. Restore default Encost Smart Homes Dataset
2. Visualise graph of dataset
3. View summary statistics
4. Exit
```

2.5.4 LoadCustomDatasetPage

The user can input a path to a custom dataset file here.

```
Load custom dataset
-----------------------------
Enter the full path of your custom dataset (.txt) file:
<user input>
```

Dataset could not be found

```
Load custom dataset
----------------------------
Enter the full path of your custom dataset (.txt) file:
<user input>
Oh dear! Your dataset could not be found! Please check your filepath.

1. Try again
2. Go back
<user input>
```

Dataset failed to load

```
Load custom dataset
----------------------------
Enter the full path of your custom dataset (.txt) file:
<user input>
Whoops! Your file's formatting may have been wrong, or we don't have permission to read it!

1. Try again
2. Go back
<user input>
```

2.5.5 SummaryStatisticsPage

The summary statistics page requires no further interaction from the user, other than to go back. The example statistics here are a sample only.

```
Summary statistics
----------------------------
Device breakdown by category:
Wifi Routers    Hubs/Controllers    Smart Lighting   Smart Appliances    Smart Whiteware
7               12                  68               54                  27

Device breakdown by type:
Encost Router 360 - 6
Encost Smart Hub - 8
Light Bulb - 39
Strip Lighting - 33
Kettle - 101
Toaster - 16
...
```

```
...
Washing Machine/Dryer - 12

Household breakdown by region:
Auckland - 77
Bay of Plenty - 24
...
Canterbury - 8

Devices per region:
Auckland - 277
Bay of Plenty - 130
...
Canterbury - 44

...
...

Press any key to go back:
<user input>
```

2.6 Modifications to the ConsoleApp class

The *ConsoleApp* class currently consists of a public, static main method, and a series of private member methods. This makes testing certain output difficult, as the accepting of standard input and producing of standard output is mixed together within methods. The client agreed with this assessment, and has allowed for additional public methods to be made which can be used by the existing private methods. Ideally, these methods would **not** be made public, since they are not to be invoked outside the *ConsoleApp* class, but this has been allowed in this case so that the existing design can be worked with. The clarified requirements are as follows:

- Add getOptions(userType: UserType) : String[]
  - This method will return a string array of all the options available based on the given *UserType*. It shall be used by the existing *selectFeature* method, passing in the previously determined *UserType* variable.
- Add getLoginAttemptPrompt(userVerified: boolean) : String
  - This method will return a different prompt depending on the value of the *userVerified* boolean value. If the user is verified, it will show the login success prompt (see Section 2.5.2), otherwise showing the login failure response.

2.7 Modifications to the DataSummary class

The *calculateDeviceDistribution* method on the *DataSummary* class returns a formatted string that is difficult to test, especially given that it combines several data points for different objects into one. It would be preferable to make the output more digestible in the form of a class with fields that can then be converted into the appropriate console output. Statistic refactoring has been approved by the client. A class diagram is depicted in Figure 2.1.

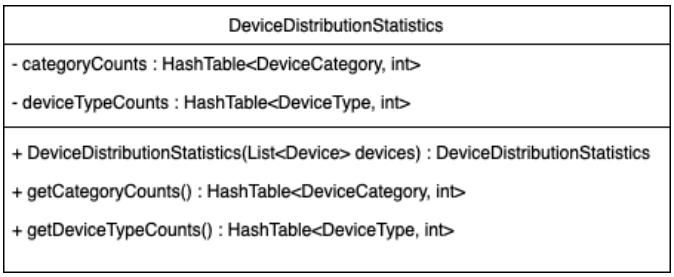

Figure 2.1: A class diagram of the *DeviceDistributionStatistics* class

This class will be used by the *DataSummary* class's *calculateDeviceDistribution* method. The hash tables will be pre-initialised to have a value for every category and device type, with all counts set to 0. When it is constructed, it will process the list of devices, increasing the existing count for the category and device type it sees for each device. Once this is complete, the two hash tables for category and device type counts can be retrieved, the data of which can then be formatted as a string and returned by the *DataSummary* class.

Additionally, a new public method on the *DataSummary* class is required to make SRS 4.11 REQ-2 testable, as shall be seen later in the document. It shall be defined as follows:

- calculateMinAndMaxDevicesForRouters(deviceGraph: DeviceGraph) : Pair<int, int>

2.8 SRS 4.6 REQ-1 - Categorising Smart Home Devices

The SDS did not specify or mention how Encost Smart Devices would be categorised into the five categories defined by the SRS. A suitable choice would be to create a class dedicated to the function of categorising these devices. The clients indicated this was acceptable. A class diagram is depicted in Figure 2.2.

The static class has a private hash map that maps from a device type to its parent category. The appropriate *DeviceCategory* enum for a given is returned for an input

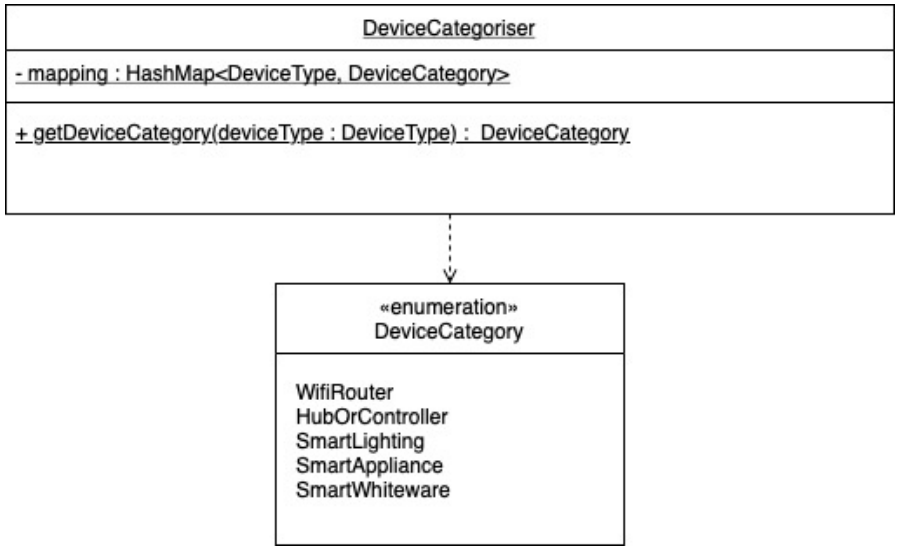

Figure 2.2: A class diagram of the *DeviceCategoriser* class and its associated *DeviceCategory* enum

DeviceType enum.

In addition, the existing *Device* class will need a new *category* field of type *DeviceCategory*, with a corresponding getter method, and the constructor should use the DeviceCategoriser class to assign this field.

3 Black-box Testing

Described are the test outlines, expectations, test cases and justifications for each High Priority functional requirement as defined in the SRS, and according to the design of the SDS.

All of the following black-box tests described will be implemented in JUnit, a Java unit testing framework.

3.1 Categorizing Users (SRS 4.1)

3.1.1 Validating ApplicationState

Description: Tests that the *ApplicationState* class is correctly setting the *UserType*.
Level: Black-box, Unit
Test technique: Value verification
Requirements covered: REQ-2
Class being tested: ApplicationState
Methods being tested:

- getUserType() : UserType
- setUserType(userType: UserType) : void

Table 3.1: Test cases

Input	**Expected output**
setUserType called with the *Community UserType*	*getUserType* is equal to the input value.

3.2 ESGP Account Login (SRS 4.2)

These tests aim to validate the username and password validation logic in the *UserVerifier* class. The valid input pairs seen in Table 3.2 are examples only. The actual credentials to be used with ESGP login have not been provided at the time of writing. Once this data is available, the inputs seen here, and in the corresponding JUnit test suite, will need to be updated.

Table 3.2: Valid inputs for ESGP login

Username	**Password before hashing**
purplecat87	bicycle
rocketman99	telescope
dancingqueen22	ballerina
beachlover123	sandcastle
techwizard42	computer
sportsfanatic55	basketball
bookworm88	library
musicjunkie77	headphones
foodiegalore44	pizza
wanderlust66	adventure

3.2.1 Username and password validation

Description: Tests the validation of an input username and password pair against the existing credentials stored in the application. If a matching username is not found, the method returns *false*. If a matching password for a matched username is not found, the method returns *false*. Otherwise, it returns *true*.
Level: Black-box, Unit
Test technique: Boundary values
Requirements covered: REQ-2, REQ-5
Class being tested: UserVerifier
Methods being tested:

- verifyCredentials(username: String, password: String) : boolean

Table 3.3: Test cases

Username	**Password**	**Expected output**
purplecat87	bicycle	true
purplecat87	potato	false
purplecat87	(null string)	IllegalArgumentException thrown
purplecat87	(empty string)	false
bookworm88	library	true
(empty string)	(empty string)	false
(null string)	(null string)	IllegalArgumentException thrown
(null string)	bicycle	IllegalArgumentException thrown

3.2.2 Login attempt response validation

Description: Tests the prompt returned in response to the login attempt result.
Level: Black-box, Unit
Test technique: Equivalence partitions { true, false }
Requirements covered: REQ-3
Class being tested: ConsoleApp
Methods being tested:

- getLoginAttemptPrompt(userVerified: boolean) : String

Table 3.4: Test cases

userVerified	**Expected output**
true	Login success prompt returned
false	Login failure prompt returned

3.3 ESGP Feature Options (SRS 4.3)

3.3.1 Correct options being returned for user type

Description: Tests the feature options shown to the user based on the *UserType*. If *UserType* is *Community*, then only one option should be shown. If *UserType* is *EncostVerified*, then three options should be shown. If *UserType* is *EncostUnverified*, then an exception should be thrown, as we do not expect to present any options to an *EncostUnverified* user.
Level: Black-box, Unit
Test technique: Equivalence classes { Community, EncostVerified, EncostUnverified }
Requirements covered: REQ-1
Class being tested: ConsoleApp
Methods being tested:

- getOptions(userType: UserType) : String[]

Table 3.5: Test cases

UserType	**Expected output**
Community	One feature option - visualise graph
EncostVerified	Three feature options - load custom dataset, visualise graph, and view summary statistics.
EncostUnverified	IllegalArgumentException thrown

3.4 Loading the Encost Smart Homes Dataset (SRS 4.4)

3.4.1 File parser gives back correct number of objects

Description: Tests whether, after reading all Device data, the expected number of devices matches the actual number of devices. A test case for both one and two lines of data are used to ensure the *FileParser* class can handle new lines correctly.
Level: Black-box, Unit
Test technique: Count verification, Boundary values
Requirements covered: REQ-3
Class being tested: FileParser
Methods being tested:

- parseFile(fileReader: BufferedReader) : Devices[]

Table 3.6: Test cases

Input	**Expected output**
One line of valid Encost Dataset CSV data	One Device object created
Two lines of valid Encost Dataset CSV data	Two Device objects created

3.4.2 File parser gives back the correct objects

Description: Tests whether, after reading all Device data, unique objects are being extracted, rather than the same object assigned to every array element. A *Set* structure can be used to check for uniqueness, based on the Device ID.
Level: Black-box, Unit
Test technique: Count verification
Requirements covered: REQ-3
Class being tested: FileParser
Methods being tested:

- parseFile(fileReader: BufferedReader) : Devices[]

Input: Three lines of valid Encost Dataset CSV data, each with a unique Device ID
Expected output: A *Set* structure loaded with the devices has the same count as the original array.

3.4.3 FileParser handles unexpected input

Description: Tests whether the *FileParser* class can handle arguments and read values that may be unexpected. These should result in controlled exceptions being thrown.
Level: Black-box, Unit
Test technique: Edge cases, Boundary values
Requirements covered: REQ-3

Class being tested: FileParser
Methods being tested:

- parseFile(fileReader: BufferedReader) : Devices[]

Table 3.7: Test cases

Input	**Expected output**
Null *BufferedReader* argument	IllegalArgumentException thrown
One line of malformed Encost Dataset CSV data (incorrect number of comma delimiters)	Returns *null*.
No data in *BufferedReader*	Returns *null*.

3.5 Categorising Smart Home Devices (SRS 4.6)

3.5.1 Device object fields are set correctly (Figure 3.8

Description: Tests whether each of the values passed to the Device constructor correspond to the appropriate fields.
Level: Black-box, Unit
Test technique: Equivalence partitions { Valid, Invalid }
Requirements covered: REQ-1, REQ-2
Class being tested: Device
Methods being tested:

- Device(deviceId: String, ...) : void

Valid inputs: Any non-null arguments (except for the router connection) to the *Device* constructor
Invalid inputs: Any null arguments to the *Device* constructor

3.5.2 Correct category is returned for device type

Description: Tests whether the correct category is being returned from the *DeviceCategoriser* class. The returned *DeviceCategory* is checked against an expected hash table of values.
Level: Black-box, Unit
Test technique: Value verification
Requirements covered: REQ-1
Class being tested: DeviceCategoriser
Methods being tested:

- getDeviceCategory(deviceType: DeviceType) : DeviceCategory

Table 3.8: Test cases

Input type	Description	Expected output
Valid	Attempt to create a Device object with non-null, non-empty, arbitrary arguments.	The Device objects field getters return values that exactly match all of the input arguments.
Valid	Attempt to create a Device object with non-null, non-empty, arbitrary arguments, but do not set the *routerConnection* argument.	The Device objects field getters return values that exactly match all of the input arguments.
Invalid	Attempt to create a Device object with all null arguments where possible.	IllegalArgumentException thrown

Test case: For each unique *DeviceType* value, check that the result equals the expected *DeviceCategory* value.
Expected output: All assertions pass - all *DeviceCategory* values match the expected values.

3.6 Building a Graph Data Type (SRS 4.7)

3.6.1 DeviceGraph assigns internal device array correctly

Description: Tests whether the *DeviceGraph* handles both valid and invalid device array arguments.
Level: Black-box, Unit
Test technique: Equivalence partitions { Valid, Invalid }
Requirements covered: REQ-1, REQ-2, REQ-3
Class being tested: DeviceGraph
Methods being tested:

- DeviceGraph(devices: Device[]) : DeviceGraph

Valid inputs: Any non-null *Device* array
Invalid inputs: Any null argument passed to the *DeviceGraph* constructor

3.6.2 DeviceGraph neighbours calculated correctly

Description: Tests whether the neighbours returned by *getNeighbours* is valid. A *DeviceGraph* is constructed using an array of Devices with some neighbour relationships defined.
Level: Black-box, Unit

Table 3.9: Test cases

Input type	**Description**	**Expected output**
Valid	Attempt to create a DeviceGraph with a non-null *Device* array argument.	*getDevices* returns a reference to the same *Device* array.
Invalid	Attempt to create a Device object with a null *Device* array argument.	IllegalArgumentException thrown

Test technique: Value verification
Requirements covered: REQ-1
Class being tested: DeviceGraph
Methods being tested:

- createEdges() : void (implicitly)
- getNeighbours(deviceId: String) : Device[]

Table 3.10: Test cases

Input	**Expected output**
An array with five *Devices*, including two routers, and three non-routers, with the *routerConnection* argument of two non-router devices set to the first router, and the other non-router set to the second router. A *getNeighbours* call takes each device ID for non-router devices	*getNeighbours* returns a valid adjacency matrix row for each of the non-router devices.
An array with five *Devices*, including two routers, and three non-routers, with the *routerConnection* argument of two non-router devices set to the first router, and the other non-router set to the second router. A *getNeighbours* call takes each device ID for router devices	*getNeighbours* returns all neighbours for each of the router devices.
An array with five *Devices*, with no neighbour relationships. A *getNeighbours* call takes each device ID for all devices.	Each *Device* array returned by *getNeighbours* is empty.

3.6.3 DeviceGraph getNeighbours handles invalid device ID

Description: Tests whether the call to *getNeighbours* will handle invalid or unexpected input correctly. The method should not accept *null* values or device IDs that do not

belong to any device.
Level: Black-box, Unit
Test technique: Edge cases, Boundary values
Requirements covered: REQ-1
Class being tested: DeviceGraph
Methods being tested:

- createEdges() : void (implicitly)
- getNeighbours(deviceId: String) : Device[]

Table 3.11: Test cases

Input	**Expected output**
An array with no *Devices*, and a call to *getNeighbours* with a null argument.	IllegalArgumentException thrown.
An array with no *Devices*, and a call to *getNeighbours* with an arbitrary device ID argument.	NoSuchElementException thrown.

3.7 Graph Visualisation (SRS 4.8)

Though this is a High Priority functional requirement, user interface and visualisation testing is beyond the scope of this Test Plan Specification.

3.8 Calculating Device Distribution (SRS 4.9)

3.8.1 DataDistributionStatistics class calculates statistics correctly

Description: Tests whether the calls to *getCategoryCounts* and *getDeviceTypeCounts* return the expected counts for categories and device types.
Level: Black-box, Unit
Test technique: Count validation
Requirements covered: REQ-1, REQ-2
Class being tested: DataDistributionStatistics
Methods being tested:

- DeviceDistributionStatistics(List<Device> devices) : DeviceDistributionStatistics
- getCategoryCounts() : HashTable<DeviceCategory, int>
- getDeviceTypeCounts() : HashTable<DeviceType, int>

Table 3.12: Test cases

Input	**Method used**	**Expected output**
A list of *devices*, with two *Devices* created for each *DeviceCategory* (use *DeviceTypes* that map to the desired category).	getCategoryCounts	The number of devices per *DeviceCategory* matches what was expected.
A list of *devices*, with two *Devices* created for each *DeviceType*.	getDeviceTypeCounts	The number of devices per *DeviceType* matches what was expected.
A list of *devices*, with two *Devices* created for each *DeviceCategory*.	getCategoryCounts	The number of categories (the number of hash table keys) matches what was expected.

3.8.2 DataDistributionStatistics handles invalid input correctly

Description: Tests whether the *DataDistributionStatistics* handles an invalid list of *Devices* argument
Level: Black-box, Unit
Test technique: Edge cases
Requirements covered: REQ-1, REQ-2
Class being tested: DataDistributionStatistics
Methods being tested:

- DeviceDistributionStatistics(List<Device> devices) : DeviceDistributionStatistics

Test case: Pass a *null* devices argument.
Expected output: IllegalArgumentException thrown.

4 White-box testing

4.1 Calculating Device Connectivity - REQ-2 Pseudocode

"The system should use the information stored in the graph data structure to calculate the minimum number and maximum number of devices that an Encost Wifi Router is connected to."

```
IF device graph is null (B1)
    THROW illegal argument error
ENDIF

SET devices to device graph devices

IF devices is empty (B2)
    RETURN 0, 0
ENDIF

INITIALISE router ID hash table

FOR each device in devices
    IF device type is router (B3)
        IF device ID not in hash table (B4)
            ADD device ID to hash table and set count to 0
        ENDIF

        CONTINUE to next device
    ENDIF

    IF device router ID not in hash table (B5)
        ADD router ID to hash table and set count to 1
    ELSE (B6)
        INCREMENT count for router ID in hash table by 1
    ENDIF
ENDFOR

IF hash table is empty (B7)
    RETURN 0, 0
```

```
ENDIF

SET max devices to first value in hash table
SET min devices to first value in hash table

FOR each value in hash table
    IF value is less than min devices (B8)
        SET min devices to value
    ELSE IF value is greater than max devices (B9)
        SET max devices to value
    ENDIF
ENDFOR

RETURN min devices, max devices
```

4.2 Branch Coverage Testing

Description: Tests the branch coverage of code, and expected output, for the calculating of the minimum and maximum number of devices an Encost router is connected to. The method being tested accepts a *DeviceGraph* object.
Level: White-box, Integration
Test technique: Branch coverage, edge cases, Boundary values
Requirements covered: SRS 4.11 REQ-2
Class being tested: DataSummary
Methods being tested:

- calculateMinAndMaxDevicesForRouters(deviceGraph: DeviceGraph) : Pair<int, int>

Notation:

- { } means an empty list of devices.
- { R1, D1 } means a list with one router and one other device connected.
- { D1 }, { D2 } means a list of devices where none of them are connected to a router.

Total coverage (Figure 4.2): [B1, B2, B3, B4, B5, B6, B7, B8, B9]; 9/9; 100% branch coverage

Table 4.1: Test cases for statistic pseudocode

Test case ID	**Input**	**Expected output**
1	*DeviceGraph* with devices defined as { R1, D1, D2, D3}, { R2, D4, D5 }, { R3, D6, D7}	Pair { 2, 3 }
2	*DeviceGraph* with devices defined as { R1, D2, D3, D4}, { R2, D5, D6 }, { R3 }	Pair { 0, 3 }
3	*DeviceGraph* with devices defined as { R1, D1, D2 }, { D3, R2, D4, D5}, { R3, D6, D7 }	Pair { 2, 3 }
4	*null DeviceGraph*	IllegalArgumentException thrown
5	*DeviceGraph* with devices defined as { }	Pair { 0, 0 }
6	*DeviceGraph* with devices defined as { D1 }, { D2 }, { D3 }	Pair { 0, 0 }

Table 4.2: Test case branch coverage for pseudocode

Test case ID	**Coverage**
1	B3, B4, B6, B8
2	B3, B4, B6, B8
3	B3, B4, B5, B6, B9
4	B1
5	B2
6	B7

5 Mutation Testing

5.1 Mutant #1

Invert device type check:

```
IF device type is not router (B3)
```

Justification: Accidentally checking the inverse of an intended condition is a common mistake.

5.2 Mutant #2

Initialise minimum devices variable to zero instead of the first hash table value seen (between B7 and B8):

```
SET min devices to 0
```

Justification: A programmer may initially intuit that the minimum should begin at zero, but this would prevent any of the actual minimums being tested as less than zero. The minimum needs to have an initial value greater than zero.

5.3 Mutant #3

Change the greater than check to a less than check

```
ELSE IF value is less than max devices (B9)
```

Justification: Accidentally using the wrong operator is a common mistake, especially when a programmer tries to speed up development by copying and pasting lines.

5.4 Mutant #4

Return the min and max in the wrong order (last line)

```
RETURN max devices, min devices
```

Justification: It is easy for a programmer to accidentally switch the return order of a simple tuple or pair.

5.5 Mutation Score

Description: Tests the expected output for calculating the minimum and maximum number of devices an Encost router is connected to. The method being tested accepts a *DeviceGraph* object.
Level: White-box, Integration
Test technique: Mutation, Value verification
Requirements covered: SRS 4.11 REQ-2
Class being tested: DataSummary
Methods being tested:

- calculateMinAndMaxDevicesForRouters(deviceGraph: DeviceGraph) : Pair<int, int>

Table 5.1: Test set 1

Input	**Expected output**
DeviceGraph with devices defined as { R1, D1, D2, D3}, { R2, D4, D5 }, { R3, D6, D7}	Pair { 2, 3 }

Mutations caught: 4/4
Mutation score: 100%

Description: Tests the expected output for calculating the minimum and maximum number of devices an Encost router is connected to. The method being tested accepts a *DeviceGraph* object.
Level: White-box, Integration
Test technique: Mutation, Boundary values, Value verification
Requirements covered: SRS 4.11 REQ-2
Class being tested: DataSummary
Methods being tested:

- calculateMinAndMaxDevicesForRouters(deviceGraph: DeviceGraph) : Pair<int, int>

Table 5.2: Test set 2

Input	**Expected output**
DeviceGraph with devices defined as { R1, D2, D3, D4}, { R2, D5, D6 }, { R3 }	Pair { 0, 3 }

Mutations caught: 3/4
Mutation score: 75%