Marking Criteria

This document contains my response to the specific marking criteria of the assignment. Justification of my design choices will be discussed.

1. Use of Containers

Primary Container

The primary container I decided to use to represent the Electricity Network was the ArrayList.

However, I had initially thought of utilising a HashMap. This being due to the simple code required for searching for a key/value pair and it's fast performance. As part of my input validation, I implemented several methods that required identify and comparing existing data in the network. This is what lead me to believe that the use of a HashMap would be quite beneficial.

The reason why I finally decided on using an ArrayList to contain the network data, was due to how I implemented a recursive algorithm to build a String variable to represent the Tree structure of the network. This algorithm required the use of the classic for-loop to iterate through the network data list based on a varying index value. Please see the following methods within the `/controller/CityNodeController` class for examples of the above mentioned recursive algorithm:

- `buildNetworkStr()`
- `recursiveBuildStr()`

Essentially, I valued use of the classic for-loop, for it's ability to change the starting point of the loop, over the speed and convenience of the HashMap.

Secondary Containers

Data Pools

I decided to use ArrayLists to contain the pool of data that would be used when generating the network data without an input file. This decision was based on the fact that it would prove to be easier to add data to the pool if I was required to do so. A simple call to the lists add() would be all that is required. Also, given that the data pools do not require any search-like function, I believed that other container forms, such as HashMap, were unnecessary. Please see `DataPool.java` in the Model package for examples.

Total Power Consumption

In regards to the function of displaying the total power consumption of the network, I opted to display each category and it's value. Given that the number of categories was a fixed number of 8, I decided contain this data within a simple double Array. Each array element corresponds to a specific and defined power category. This seemed like a simple solution, that only required updating each element value whenever a new leaf node was added to the network tree.

Please see `CityNode.java` in the `/model/CityNetworkCompositePatt/` directory for detailed depiction of the array setup.

The array utilisation can be seen in the `updateTotalPower()` method in the `/controller/CityNodeController` class.

2. Package/class/interface/method responsibilities

I have utilised the MVC design pattern and have therefore, split the application classes into three main packages:

- Model
- View
- Controller

I have also isolated classes that represent the implementation of particular design patterns into sub-packages as well.

The starting point of the application, `MainApp`, remains outside of the MVC packages.

Model

All classes that maintain the application data are found within the Model package. As such the following sub-packages exists within it too:

- `CategoryDecoratorPatt` - Decorator pattern design used for containing power consumption data.
- `CityNetworkCompositePatt` - Composite pattern design used for creating nodes of the network tree.

View

This package only contains the single class used to output the final data to the terminal. Other instances of outputting to the terminal are in relation to data validation, and remain associated with the Controller package.

Controller

All other classes, except for MainApp, are found within the Controller package.

3. Error handling

I created two separate custom Exceptions to handle the following user input validations:

- `ArgsException.java` - Command line argument input
- `InvalidFormatException.java` - Input file

Exceptions that are caught output a custom error message relevant to the particular user error.

4. Strategy Pattern

I decided to implement the Strategy Pattern design to manage the command line arguments that the user inputs. This seemed appropriate as the user could enter the application settings in three different forms - two, three and four arguments. To read all three of these cases would require the same two parameters, this being the args array and an ArgsValidation object. However, the validation process will differ depending on the number of arguments entered by the user.

To demonstrate the Strategy Pattern design, I created a ReadArgs interface to be implemented by three separate classes. These classes will then perform separate tasks specific to their requirements. For example, in the event a user enters two command line arguments:

-
  ```
  ReadArgs readArgs = new TwoArgs(); //ReadArgs object is created using the TwoArgs class implementation.
  ```

Now this form of the ReadArgs class can call on the validation process specific to the event where the user enters two arguments. A same process is executed in the event the user enters three or four command line arguments, except with calls to there respective implementation classes (ThreeArgs, FourArgs).

As a result of this implementation, the MainApp class only needs to only know about one of the three implementation classes when it is executed. Thus, promoting low coupling between the MainApp class and the three ReadArgs implementation classes.

5. Composite Pattern

Given that the composite pattern is used to create a tree like structure of objects, it made sense to use this pattern to build the city electricity network using "nodes" in the following manner:

- `Node` - Base component to be implemented by the below mentioned classes.
- `CityNode` - Composite class that builds the tree structure.
- `SubCityNode` - Child component class containing node name and parent node name.
- `PowerUsage` - Leaf component classes containing node name, parent node name, and power consumption values.

With this setup, the base of the tree is created when the CityNode is instantiated. Building the tree is done by adding objects of the SubCityNode and PowerUsage. Now with the implementation of the two concrete classes, SubCityNode and PowerUsage, the CityNode class is able to build the tree to any size/depth required.

6. UML

The UML class design for this application clearly illustrates the separation of classes into the corresponding MVC packages. Additional design patterns used are also contained within their own sub-package. The diagram has been updated to reflect the current version of the application, as at submission date.