COMP.SE.110 Software Design Group Assignment

Weather and Electrical Grid Data Monitoring Tool

Group Rotta Software

Arttu Leppäaho	282591	arttu.leppaaho@tuni.fi
Johannes Simulainen	284213	johannes.simulainen@tuni.fi
Paul Ahokas	283302	paul.ahokas@tuni.fi
Petrus Jussila	283047	petrus.jussila@tuni.fi

Table of Contents

1. Project Planning

In the start of the project, we held a meeting to figure out what the project would entail and what we needed to do. We also came up with a GUI layout that we thought would have all the necessary controls for the final program.

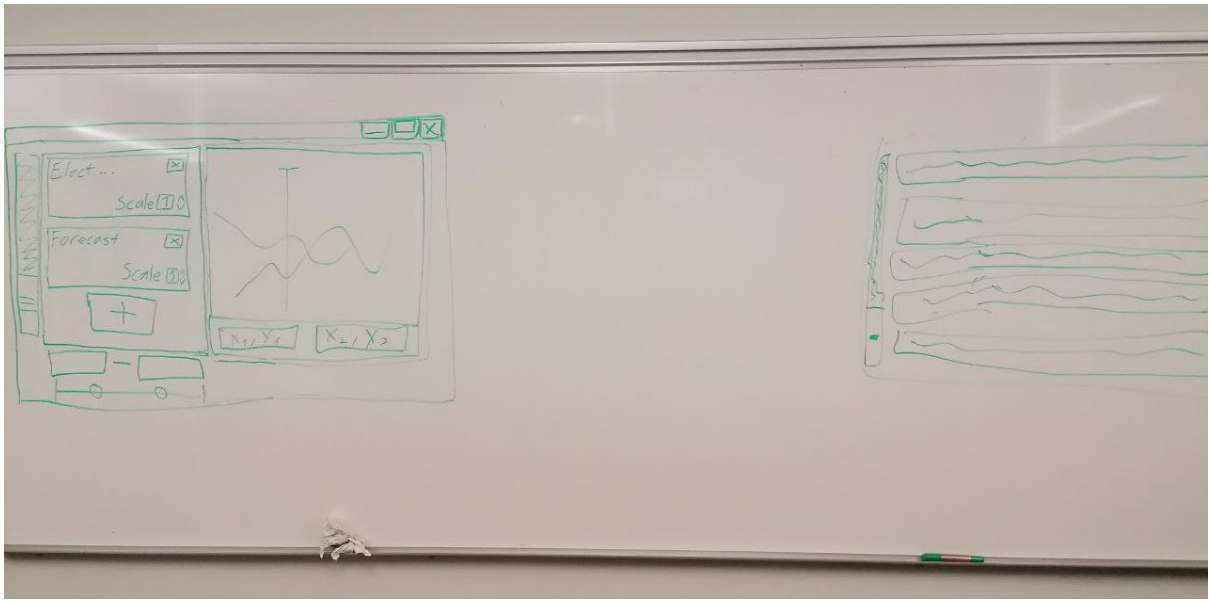

Image 1: Whiteboard sketches for the main view of the application and a data source selection view

For the prototype, we thought we should start with the core functionality of the program, which is to fetch data from the APIs and display it in a graph. After that it should be straightforward to expand that core into more features.

We had already decided to use C++ and Qt to implement the program, as both were tools everyone in the group was familiar with, but during the meeting we became unsure if fetching data from the APIs would prove to be too much work using C++. The raw XML weather data looked intimidating and we were not able to understand how it was meant to be used at first.

We decided that if we could figure out how to fetch and parse the data during the first week of work, we would continue the project in C++, and otherwise we would switch to Python. Python would have been a good choice for the project as well, as it was another programming language that we all knew, and we could have used libraries to fetch FMI weather data easily and still use Qt for the user interface through PyQt.

However, when we looked more into the weather data API, we realized it was not as complicated as we had initially thought, and we were able to fetch the data programmatically within the week. After that, we had a rough outline of the program to start implementing.

After the prototype, we began to plan other basic features of the program in more detail. This included fetching all the required types of data, the ability to add and remove data sources from the application GUI without hardcoding and making the graph view more readable with labels. Detailed planning of some more specialized features was left for the final submission, which is something we could have done earlier, as they did cause some issues with the program's existing structure.

2. High Level Description

The application is programmed in C++ and uses Qt as the only framework/library. As described in the previous chapter, we chose these tools because everyone in the group was already familiar with them, but as we were starting the project, we also realized that Qt has a lot of useful functionalities that remove the need for several more specialized libraries.

We initially planned to at least find a separate library for parsing XML data, but Qt turned out to have that feature already. Graphs and network requests were also easy to make with Qt. Overall, Qt seemed like a good, straightforward framework to use for the project. We chose XML as the format of data to fetch, as both FMI and Fingrid were able to send XML data, and that allowed us to share code for importing both types of data.

Image 2 contains a class diagram of the program's structure.

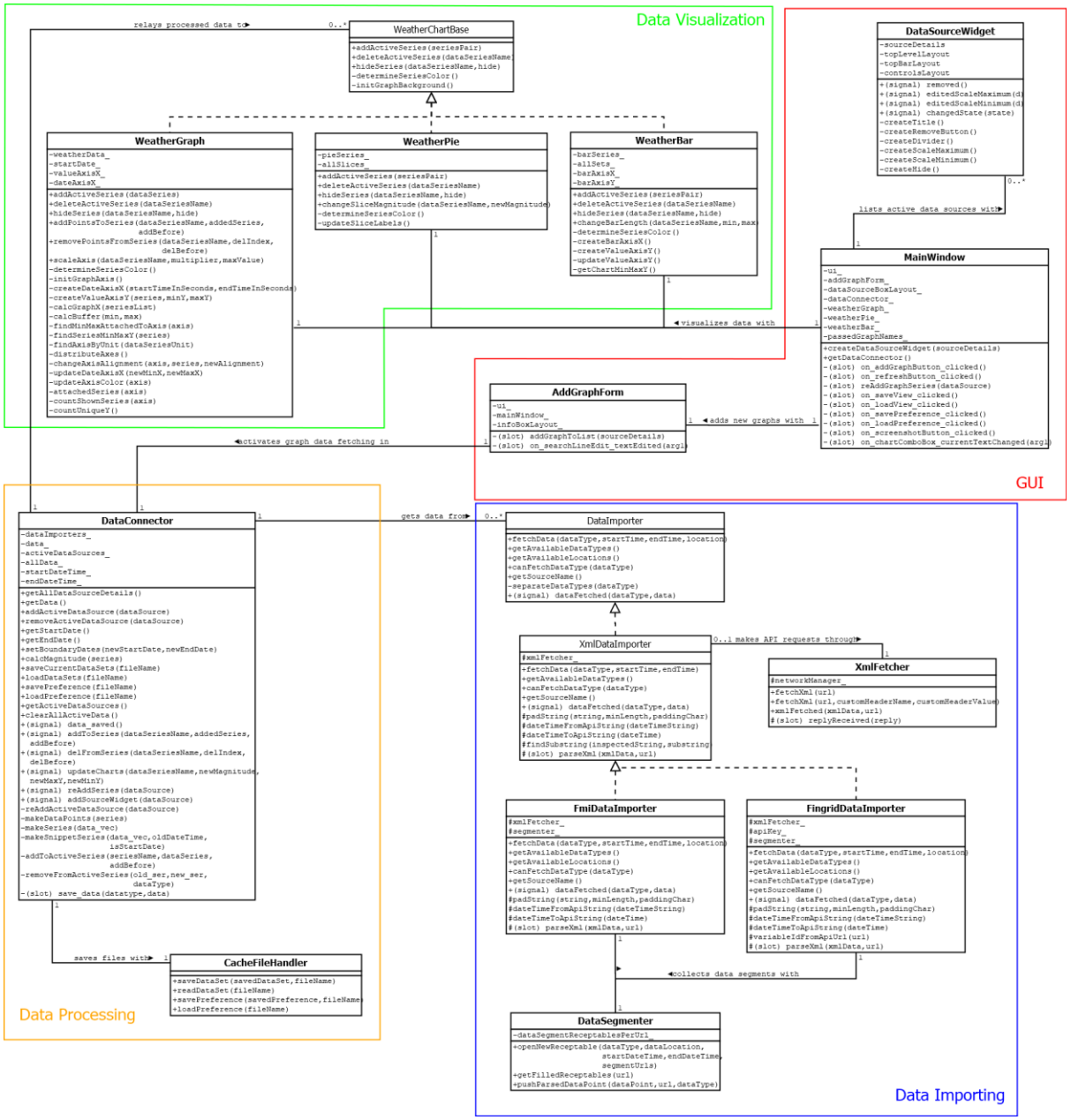

Image 2: A class diagram of the application's classes. The colored polygons represent the four parts of the program described below.

The application is split into four parts: the GUI, data importing, data processing and data visualization. The GUI part handles the windows the user interacts with, the data importing part handles fetching data from the REST APIs the application uses, the data processing part stores fetched data and converts it into a displayable format and the data visualization part handles displaying the data in various graphs and charts. This seemed like a logical division: each part has its own job and they don't overlap much. It also allowed for easy division of the project's workload between all four group members.

3. Individual Program Components

As stated above, the application has been roughly divided into four parts. The GUI part contains the MainWindow, AddGraphForm and DataSourceWidget classes, which handle the main window, data source selection and active data source boxes respectively. Visualization is handled by the three classes inheriting from WeatherChartBase, which display the given series as graphs, pie charts and bars in the main window. FmiDataImporter and FingridDataImporter fetch data from the APIs using XmlFetcher and parse it. Between visualization and data importing is a class called DataConnector, which further processes the fetched data so that it can be used in the visualization part of the program.

Below are short descriptions of each class in the program and how they interact with each other.

3.1. MainWindow

This is the bread and butter when it comes to the GUI. MainWindow is an XML file which organizes the classes' and modules' outcomes to display properly and neatly for the user. In it, the user can see the graphs displayed by WeatherGraph and open the AddGraphForm window. The user can close any graph by pressing the "X" symbol above the said graph's info box.

Beneath the layout widget, there are buttons for storing preferences and data as well as loading them respectively. Whenever pressed, they initialize a QFileDialog where there wanted file location is given by user.

Under the ChartView widget are pair of buttons, two TextBrowser boxes, two TimeEdit boxes and one ComboBox. The TextBrowser and TimeEdit boxes are used to display the chart's beginning date and time as well as the end date and time (x-axis). The user can change the value of the date and time in the TimeEdit boxes and apply changes with conveniently named button. User can also take a screenshot of the displayed chart and store it. Next to the screenshot button, is a combobox which handles the chart type. The user can change it to make the data be displayed as line, pie or bar chart.

3.2. WeatherChartBase

WeatherChartBase is an abstract class inherited from QChart which describes the common functionality of the different types of charts the program uses for data visualization. The common functionality, which is implemented by the charts themselves, includes the addition and removal of series, hiding series, and determining displayed series' color. The only method implemented in WeatherChartBase is the initialization of the charts background and legend. The class also includes different constants used by the charts, such as colors and boundaries.

The DataSeries structure is also defined in WeatherChartBase. DataSeries includes a data series and its characteristics, like the unit of measurement, minimum and maximum values, and the axis the series is attached to.

3.3. WeatherGraph

WeatherGraph handles displaying the fetched data points as a spline graph in the main window by inheriting WeatherChartBase. On top of the functionality defined in WeatherChartBase, WeatherGraph can handle the addition and removal of points from the different data series as the time interval changes. The time interval changes automatically depending on the length of the interval defined by the user. The graph's series can also be scaled by the user. WeatherGraph is displayed as a part of MainWindow.

The class has multiple features to keep the graph area tidy while the user interacts with the displayed data. One of these features is axis distribution, which prevents the axes from "stacking" on one side. The axes get automatically scaled as the displayed data changes. If multiple series are of the same type of data, WeatherGraph uses the same value axis for all of them. WeatherGraph also adds a buffer to the axis values to prevent the series' line from getting cut out.

3.4. WeatherPie

WeatherPie displays the magnitudes of the data series in relation to other active series. On top of the WeatherChartBase functionality WeatherPie changes slices magnitudes as the active data changes and displays the data's percentages after their names. WeatherPie should be mainly used to compare different energy production sources.

3.5. WeatherBar

WeatherBar displays the data series' minimum and maximum values as a bar chart. The chart has a normal value axis as the y-axis and a custom x-axis which displays the different categories defined in a constant in the header file. The value axis gets scaled automatically as the active data changes. WeatherBar should mainly be used to compare different minimum and maximum temperatures.

3.6. AddGraphForm

Whenever the user wants to display a new graph in MainWindow, clicking the Add Graph-button does exactly that. It creates a new window called AddGraphForm where the user can select the appropriate data they want to see. Whenever data is being fetched, the window will create a new widget for each type of data. The user can filter through the contents by typing into the search bar which filters as contents change. After the Add-button is pressed next to a data source description, that data source is added to a list of active data sources and its graph is displayed in WeatherGraph.

3.7. DataSourceWidget

This class is used when a new data source widget is being created in MainWindow. Previously widgets' sizes and functions were fully in MainWindow but now they are in their own class. DataSourceWidget has the all the parameters for creating a widget and handles connect-functions for each button and spinbox. MainWindow just simply calls them.

3.8. DataConnector

DataConnector is used as a connector between the data provider and the user of the data. It relays details of data sources and requests data from DataImporters and converts it so it can be easily used by other classes.

Making the request to fetch data happens when the user chooses the weather/electricity type. A signal is emitted and DataConnector's connected slot save_data is called in response. Because of the delay in fetching the data, a signal data_saved is emitted after saving the data so it can be safely acquired using a connected slot.

Whenever the user chooses to change the time interval, DataConnector gets notified of this and uses already fetched data if possible, otherwise it requests more data. It merges old data with new data and emits signals to update the charts according to the new changes.

Additionally, DataConnector uses the CacheFileHandler class for saving and loading data sets and presets to xml files whenever the user wishes to do so.

3.9. DataImporter

DataImporter is an abstract base class that all data importers in the program inherit from. It matches the Facade design pattern in that it hides all the steps necessary to make and parse an API request and lets you import data by simply using the methods fetchData and dataFetched.

fetchData is a method used by inheriting classes to fetch data from a source and dataFetched is a Qt signal used to retrieve the data once it has been fetched. The data importers in the program all use REST APIs as the source of data, so the data must be fetched across the internet and won't arrive immediately: this is why the data cannot simply be returned in fetchData.

fetchData allows selecting the type of data to fetch and the time period from which to fetch it from. Not every DataImporter is required to be able to fetch every type of data handled by the program, so the methods getAvailableDataTypes and canFetchDataType can be used to ask a DataImporter about the data types it's capable of importing before calling fetchData. All data returned is always in an std::vector containing generic data points with a value and a date and time from when the value was taken. The dataFetched signal passes the type of the returned data separately, as it can't be inferred from the raw data points.

Having this base class allows the rest of the program to handle data importers through DataImporter pointers without knowing their actual type, which allows for easy extension of the program with more importer types. We originally planned for DataImporter to be an interface, but Qt doesn't let interfaces have signals, so we instead made it an abstract class.

3.10. XmlDataImporter

XmlDataImporter is another abstract class inheriting from DataImporter, which implements a few utility methods for parsing XML data and stores an XmlFetcher for fetching the data.

3.11. XmlFetcher

XmlFetcher's job is to fetch XML data from the REST APIs used in the project. The class isn't tied to a specific format of XML data, as it simply relays the fetched data to an XmlDataImporter that parses it for the actual data.

Fetching XML data using this class happens by calling its member function fetchXml, which takes a raw URL as its parameter and tries to obtain XML data from it using a request made by a QNetworkAccessManager. A custom HTTP header can also be given as a parameter to fetchXml: this is used in FingridDataImporter to pass an API key. As the data cannot be obtained immediately across the internet, it will arrive after a delay. This is accounted for with the signal xmlFetched, which is emitted when the API has replied to the fetch request and can be connected to a slot to retrieve the XML data.

3.12. FmiDataImporter

FmiDataImporter is a DataImporter that makes requests to the Finnish Meteorological Institute's weather data API and parses the reply returned by it for data. The parameters of fetchData are turned into a URL of the right format and sent to the API. The XML format reply of the API is then parsed for usable data when it's received.

3.13. FingridDataImporter

FingridDataImporter is a DataImporter that fetches data from the transmission system operator Fingrid's electricity data API. An API key is required to make requests to the API: FingridDataImporter reads this key from an external text file contained in the program's root folder. This way the key can be changed easily without recompiling the entire program.

Like with FmiDataImporter, the parameters of fetchData are turned into a URL for the API request, but this time the API key also needs to be supplied in an HTTP header: FingridDataImporter gives XmlFetcher this header's details to include it in the request. The XML data returned by the API is then parsed similarly to FmiDataImporter's behavior.

3.14. DataSegmenter

DataSegmenter is a utility class used by FmiDataImporter and FingridDataImporter to collect fetched data segments and combine them into larger, singular data series. This is needed because both APIs used by the program have a maximum limit for how long the time frame in a request can be: to fetch larger time frames, multiple requests have to be sent. DataSegmenter lets FmiDataImporter and FingridDataImporter group these requests together so that they can be correctly combined when they've all been received for a single fetchData call.

DataImporter doesn't store a DataSegmenter by itself to allow for more flexibility in how DataImporters operate.

3.15. CacheFileHandler

CacheFileHandler is a class used for saving and loading data sets and preferences to/from an xml file.

A preference contains information of the weather/electricity data type so data of this type can be fetched immediately after the file has been read. Loading a preference is a fast and easy way to load a combination of data types.

A dataset also contains information of the weather/electricity data type, but additionally it stores the time interval and datapoints that are within the interval. This way there is no need to fetch data from the API. This is useful for comparing predicted data to correct data.

4. Design Decisions

We chose Qt as the only external library due to our familiarity with it and its good amount of features that proved useful in implementing the application.

The application is structured mostly like the push MVC pattern, where the visualization part of the program is the View, the GUI is the Controller and the data processing and data importing parts are the Model. The Model directly pushes new data to the View, which doesn't do much on its own. This division isn't perfectly seamless though, as there is some overlap in the GUI, which could count both as a part of the Controller and the View. We chose this structure to increase the code's clarity and modularity, and to separate the program's functionalities into easily manageable parts.

We also thought about SOLID principles while designing classes, particularly the single-responsibility principle and Liskov substitution principle. For example, DataImporter's only responsibility is to import data and it doesn't process or save it in any way: that's done in DataConnector. WeatherGraph, WeatherPie and WeatherBar only display data and depend on other classes for importing and saving the data. In general, we tried to give each class only one responsibility.

The Liskov substitution principle can be seen in how DataConnector handles DataImporter pointers: it doesn't need to know anything about the DataImporters' actual types, and it can use them entirely through the shared interface defined by DataImporter no matter how they work internally. This fulfills the requirement for easily adding new REST APIs for the program to import data from: creating new DataImporter types doesn't require any modifications in the rest of the program besides adding the new DataImporter to DataConnector's list of available importers.

5. Self-Evaluation

Our initial design for the program worked decently, and we were able to implement the required features without major changes. However, we did have to add several smaller classes around the core structure of the program.

These smaller classes were DataSourceWidget, WeatherChartBase, WeatherPie, WeatherBar, CacheFileHandler, DataSegmenter and XmlDataImporter. The functionality of most of these classes could have been contained in the planned core classes, but we decided to separate them into their own classes to follow the single-responsibility principle and maintain a clearer program structure. The sudden addition of some of these classes was also because we didn't plan out the program in enough detail initially. For example, we should have always known we would need classes for displaying pie charts and saving files, but we didn't spend enough time planning the application to note that in our initial plans.

Some features also got omitted in favor of other features and due to time constraints. The initial design had a way of showing the exact values of the graph, but this was deemed too time-consuming. On top of that this feature wasn't mentioned in the assignment specification, so we chose to use the time elsewhere for more important features.

When we developed the prototype, we hadn't yet fully thought out the internal behavior of some classes, but after the prototype submission we were able to come up with more detailed designs for all of them that worked. More specifically, we hadn't fully planned how the classes needed to interact and share information between each other.

We also had to change the interface IDataImporter into the abstract class DataImporter due to Qt signal issues. DataImporter is still almost a pure interface, but it needs to inherit from QObject in order to have a signal, which forces it not to be an interface.