COMP.SE.110 Software Design

Design document – group ilmanenfemma

Initial high level description

Our application is built with C++ and Qt/QML. The frontend is done with QML, and the backend is done with C++. The choice of these technologies was made due to the familiarity of Qt from previous course work. None of the members of our group had experience with QML, but it was chosen anyway as it was expected to be somewhat similar the Qt framework, therefore making it at least somewhat familiar.

Our initial design can be seen in figure 1.

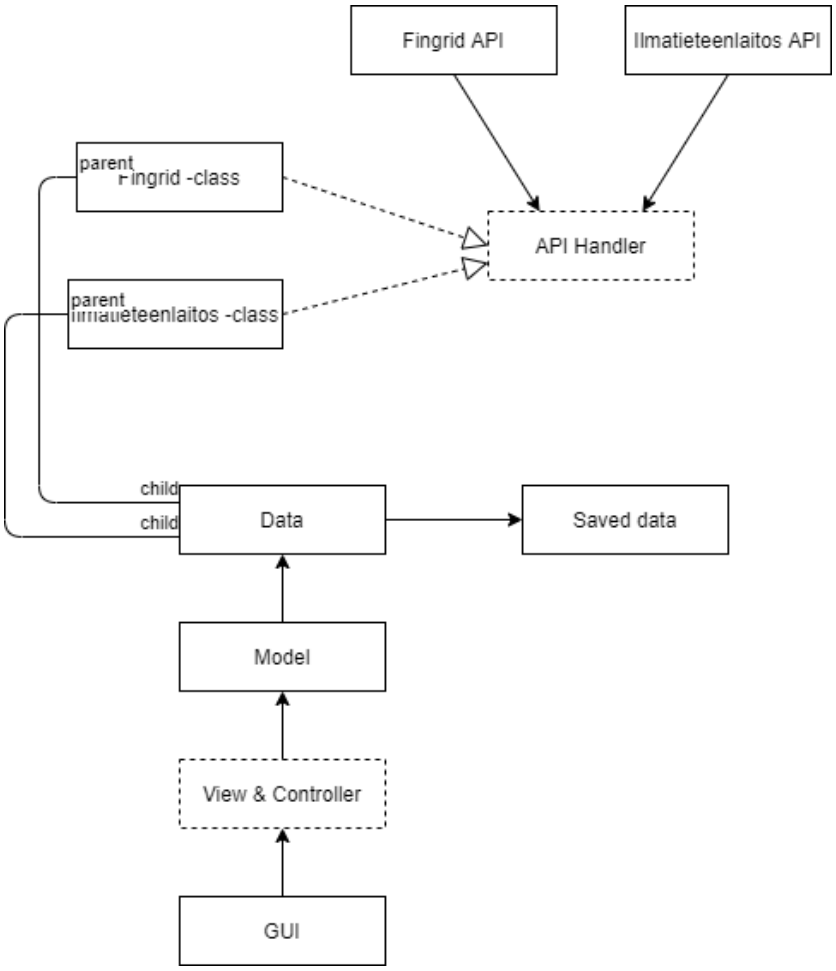

Figure 1. Initial High level design

In this design, we expected to use a base class for interacting with the required APIs. This design seemed good, as the modules interacting with the APIs were expected to be using similar functionalities. Also, this design would make it easy to add new sources of data into the application.

Next in the flow of data in this design would be the data module. This was intended to manage the data in the application. Initial ideas were that the data would be parsed here and stored for later use.

We also had some initial ideas on data could be stored, hoping we could use one interface to deal with data directly from the APIs or data stored somewhere on the computer.

Next in the flow is the model, which was intended to manage the data in the previously mentioned data module. The relationship between the model, view, and controller seemed odd at this point, and their actual use wasn't very clear. We expected to use some form of the MVC pattern, but the pattern (and its actual implementations in Qt) was not so clear at the time. Nevertheless, the data flow would have been to the GUI and commands from GUI to the controller (or something similar).

Boundaries and interfaces

In this section, we'll cover the internal data flow of our application, and describe interfaces and objects in a more concrete sense, as they were implemented.

The most important module in the application is going to be the actual graphical user interface. This is defined the file 'main.qml'. It handles all the inputs of the user: there are checkboxes for different options plotting options of the graphs, and a combobox for choosing locations for weather data.

This main interface implemented with QML is in direct communication with C++ backend, in particular the class Chart This main purpose of this class is to interact with the graphical interface: graphing the plots, dealing with user inputs, and communicate with rest of the backend components.

The Chart class has access to an XML parser module. This module has only one responsibility of parsing raw data from the APIs, and to convert into a data structure we can use with the backend. There are two functions for this, one for each API.

Lastly, there are two classes for handling direct access to the APIs. These are called Fingridhandler and FMIhandler. They derive from the same base class "BaseAPIhandler". These classes are responsible for getting the data out of the APIs and sending it to whichever component needs the data.

In figure 2 you can see the implemented classes and the relations between them.

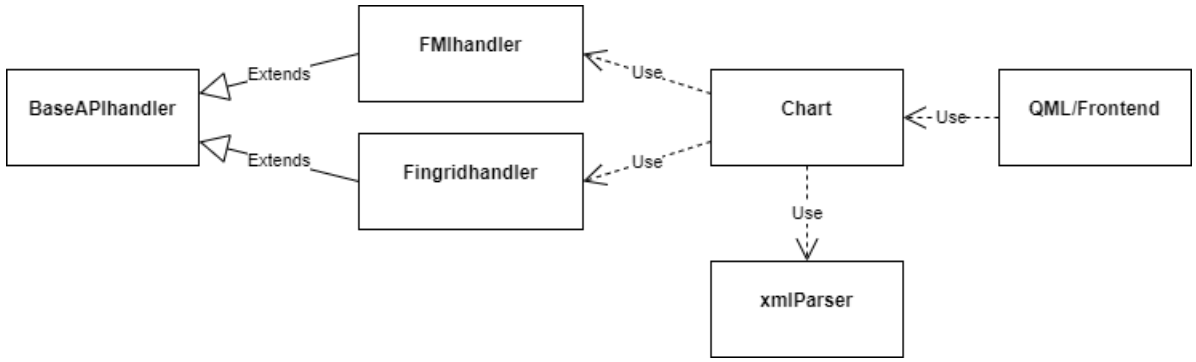

Figure 2: Relations between implemented classes.

Internal information flow

Let's take a closer look at the information flow, particularly how we get to a desired result in one data visualization.

Consider the following use case: the user wants to see the electricity consumption from the period defined in the date parameters. As the corresponding checkbox is ticked, it sends a call to the Chart class and the class method getFingridData, which is specified to be invoked by QML components. The parameters given to this method are the API identifier, start, and end times as strings.

At this stage, getFingridData takes the given parameters and modifies them in a way that Fingridhandler can use the arguments to make an API call. The function then calls a method of Fingridhandler called getFromFingrid. This method is responsible for getting the data from the Fingrid API. As the getFromFingrid method invokes a network reqest defined in the baseAPIhandler, the reply of the request is sent back to the Chart class. Using the signals & slots mechanism in Qt, the there is a signal coming from the baseAPIhandler class, which is then inherited by Fingridhandler. As the request is finished, the reply is sent back a slot in the Chart class. The signal contains two arguments: one is the raw data, and a string ID to identify, which API data is sent.

Here, the Chart class has a slot (corresponding to the sendData-signal in the Fingridhandler), called receiveFingridData, and it takes the same arguments as in the sendData signal from the Fingridhandler: raw data from the API and an ID. As this data is parsed and stored in the data structure storing all the data gathered from the call, it gets stored in a QMap, where the key is the ID as a string, and the value is a QVector, containing all the currently stored data points fetched from this ID.

Once the data is parsed, and stored, the Chart class sends a signal back to the interface (main.qml). The signal is called fingridSeriesReady, and it has an argument of an ID. The purpose of this signal is to emit information that data requested by the call initiated by the user is ready to be drawn on the screen. As main.qml receives the signal, a graph is created dynamically on the QML side. This graph is then filled with the data fetched from the API with the Chart class method setLineSeries, and finally the requested data drawn on the screen.

The process for each different option is almost identical, the difference being that API ID is different, and API requests and parsers use different functions (but the end results is still same). Figure 3 illustrates the flow of information in a checkbox call.

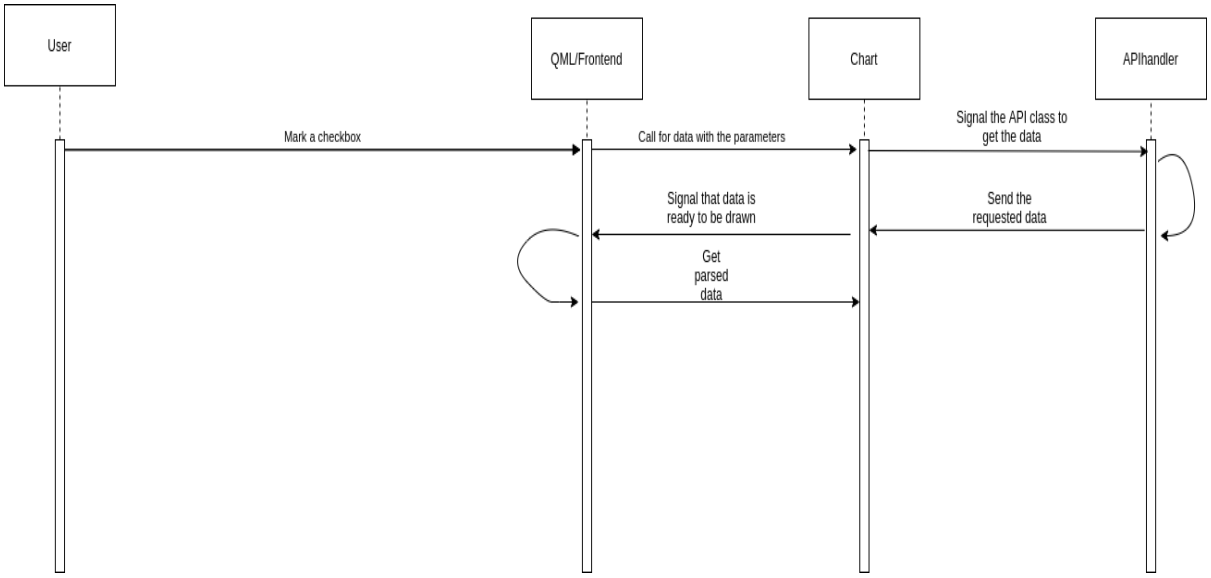

Figure 3. Interaction diagram

Describing components in more detail

- BaseAPIhandler

BaseAPIhandler contains basic methods both of the used APIs can use. These include making requests to the API and then sending the received data forward to the next class in the data flow.

- FMIhandler

FMIhandler extends the BaseAPIhandlers functionalities with methods required by the FMI API. First, this class can manage the IDs required to make different type of data requests to the FMI API. Then it has methods that edit the base API call to be suitable for the FMI API.

- Fingridhandler

Fingrindhandler functions just as FMIhandler does and extends the BaseAPIhandlers functionalities to work with the Fingrid API. It can enumerate between different types of data requests and can edit the base API call to work with the Fingrid API.

- Chart

Chart is the C++ class that is connected to the QML/Frontend of the application. Chart has the methods to initiate data requests using the handler classes described above.

- XmlParser

XmlParser extracts the data from the .xml files and edits it to the form it is used in the QML charts.

- QML

The frontend of the application. Contains the code behind the UI and is connected to the backend through the Chart class.

Design decisions

In this section we describe the design decisions we made during the process of development.

Like we described in the initial high-level description, we decided to build the application with C++ and Qt/QML. Some other components were considered at the start: a database to store data, and a 3rd party library to implement the XML parsing. The decision to only implement XML parsing was made, as the FMI API only sends XML data, and the Fingrid API had the option of getting XML data as well. This meant that we could implement one similar function for parsing. However, both ideas were ditched early in the development process. The idea of a database seemed too complex for the start, as we thought we should be able to store the data using the data structures STL and Qt provide, and we got decently functioning XML parser quite early in the development process: therefore, the idea of using a database and a 3rd party XML parser was ditched.

As it was somewhat obvious that classes interacting with the APIs should have somewhat similar characteristics, it was decided to use base class for them. With inheritance, this design achieves SOLID principles nicely and code reuse is achieved. With this design, new APIs could be added with ease.

In the original design, we expected to have a class between the Chart class and the API handler classes. The purpose of this class was preparing the requests and manage the incoming data (sending it to a parser, storing it somewhere for more permanent use etc). We did have an actual class implemented here to act as a link between the view and the data. The supposed benefit with this class was that it'd reduce the responsibilities of the Chart class. The trade-off was that the information flow would be harder to understand with one component between the Chart class and the API handlers. A decision was made to scrap this class, and in the final design data only flows between the Chart class and the API handlers. This does add some responsibility outside of the Chart class, but the made the program much easier to reason about. As this responsibility is quite limited, and the class is already connected to the UI making the requests, this might not be even considered additional responsibility.

First approach for saving the data long term was to open a file explorer and let the user choose the location of the save. However, the user would have to remember the location when opening the graph again. This moves some of the responsibility (and memory load) to the actual user and thus it was decided that the save location would be static. It is also worth noting here that no actual datapoints are saved. Instead, it is just the chart being grabbed from the view and saved as .png file.

Self-evaluation

After the mid-term submission, we almost had to ditch all the plans regarding the front end of the application. It turned out that while we were successful plotting a graph, plotting multiple graphs seemed to be impossible (or then we overlooked some features QML/Qt offers) just using the back end. The solution which we found to this problem was to dynamically generate graphs from the QML side of the logic. While this solution works for plotting, it does have massive drawbacks, and generally seems to violate good design principles. For example, all the checkboxes share the same logic, yet each similar function call must be defined for each checkbox individually. Additionally, there must be a better way to isolate responsibilities of these components. Ideally, all the data for these checkboxes (which component are manipulating etc.) should be declared in one place only, and only then used by

other components. Currently, we're spreading data in parts of the front-end as well as the back end, which isn't a good way to do it.

Some changes were made to the back end as well. The data flowing from the APIs to the GUI was too complex, as it was sent through signals & slots chained as follows: APIhandler -> XMLparser -> back to GUI. This design was poor as it made it much harder to see how the data was handled. This design was simplified in the final application, and there is now a direct signal from the API to the GUI to receive the data: parsing is done afterwards in a separate function. This decision made much more sense and improved maintainability by a lot.

As we didn't meet all the functional requirements in the application, it should be analyzed why. Requirements we didn't meet:

- Large scale data collection (collecting data from 180 days and such)

Our current design signals the UI to create a new graph every time an API call is made. This is problematic as large-scale data collection requires the API handler to do it in multiple batches, creating multiple calls for one graph (if not done in real time over a period of 180 days). The fix to this problem would be to use a lock (or similar solution relying in concurrent execution) to block the signal generating graphs dynamically, and only allow it at the end, and adding a small delay between each API call to not to throttle the API. Attempts were made to fix this problem, without success.

- Saving data sets and producing visualizations
- Saving preferences

These two are somewhat connected to each other and this is also linked to the problem described in first paragraph. Our front-end is not designed in a way which would make it easy to implement functionality as the current implementation is tightly coupled with the problems we were facing at the beginning of implementing multiple graphs in one view.

To summarize, technical problems mostly arise from poor understanding of the framework, thus leading to a poor design overall. Better understanding of QML, and interaction between QML and C++ would have helped a lot. Also, compared to plain Qt within C++, QML doesn't feel familiar at all, or then we have terribly overlooked something. At least some of us felt that this combination of QML/Qt and C++ was a poor choice, as we had committed this combination as a group, we had to go through with it.