Software design

1.Submission

Group_1

Julia Harttunen	H291740
Verna Leisti	H274423
Krista Mätäsniemi	H292064
Eero Tarri	H283568

The structure

The application uses MVC implementation with adapters to connect APIs to the model as a structure. View for displaying and model for communicating with source of data.

Most important components

View is the most important visible component. It provides a graphical user interface to use the application. Views main functions are visually presenting the data and passing commands to controller as in Qt signals are sent from the actual components which are in View.

Controller is responsible for user actions. It separates the visual representation from the controls. Control mostly takes signals from View and calls prompts Model to do something. It can also manipulate the input data into correct form if necessary.

Model is responsible for the business logic behind the application. It gets parameters from controller to change data. Model can communicate indirectly through adaptors to access data from websites. Model forms graphical components from given data and gives them to View to visualize.

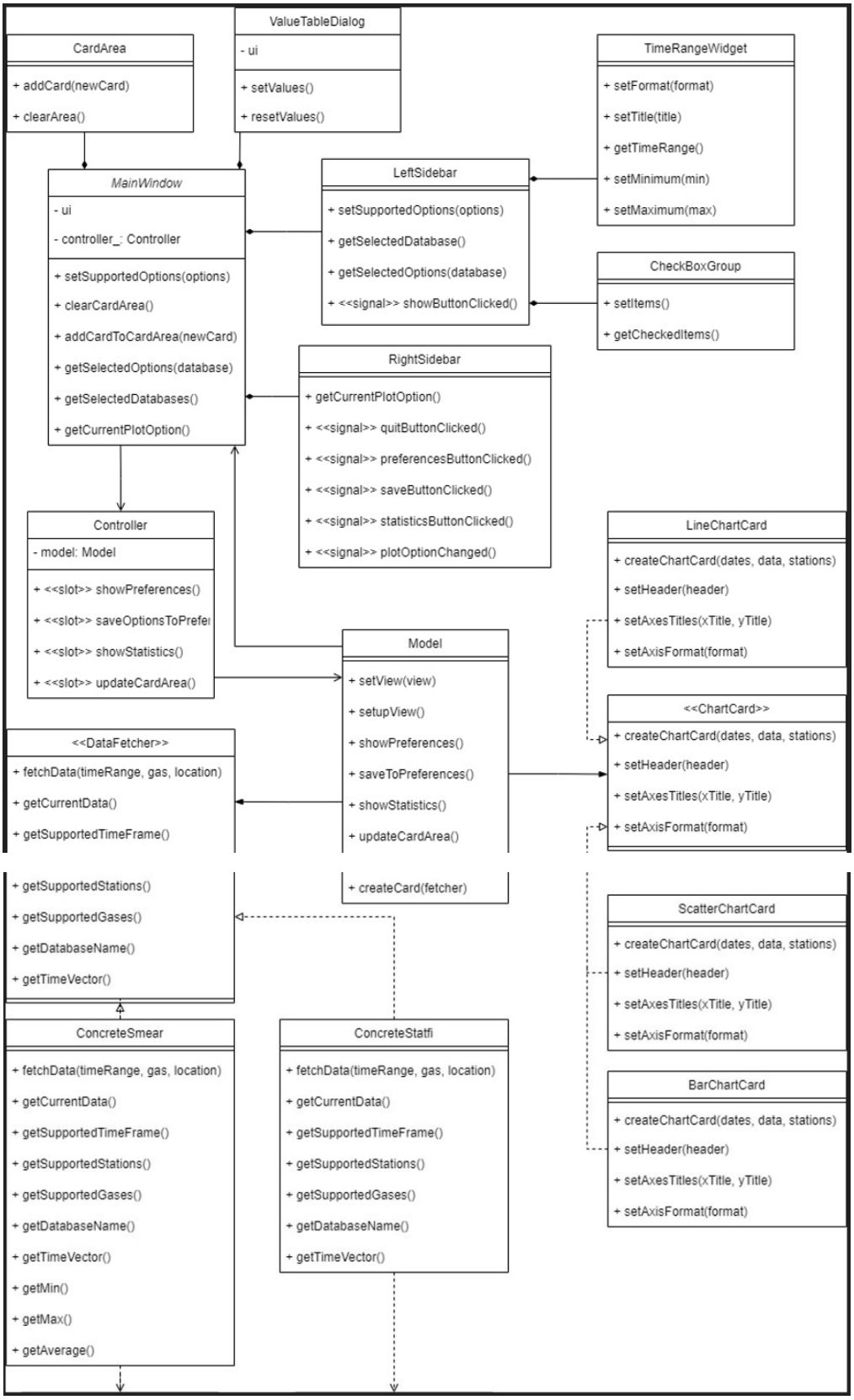

Figure 1: Class diagram

Most important interfaces

STATFI API is an interface for the model to get historical data from STATFI. Its purpose is to simplify the interface for model so that model can access the data with only very few lines of code. Statfi interface first and foremost gets the data from the network service. It will then take the data and manipulate it into a vector that is considerably easier to handle in the model. The interface takes parameters from model for posting and notifies when data is ready to use.

SMEAR API is an interface for the model to get recent data from SMEAR. Like STATFI API, SMEAR API also simplifies the interface so the model can access it easily. It takes parameters defining which information is being fetched from a network service and it returns the data to the model in a vector.

The classes

MainWindow

MainWindow-class is implementation from the view. The class contains UI components and receives user's inputs. MainWindow calls Controller according to user inputs and Controller recalls Model. MainWindow presents data fetched from Model to the user. This means that MainWindow doesn't form a graph, but only dispalys it to the user. MainWindow prevents user from making input errors by locking unnecessary options. For example, user can't choose options which the database doesn't give.

We decided to implement UI so that it notices possible changes in data. This means that UI changes depending on what information is available in databases. If a new gas or measure station enters the database, it will appear in the UI. This will help in future development and reduces the making of major changes to the code.

LeftSidebar

LeftSidebar-class is inherited from QWidget. It is a container for all elements in view's first column. It contains all output fields and checkboxes for user's input. The LeftSidebar signals showButtonClicked()-signal to the view when user presses the showButton. This class books up selected options and sends them when asked. The class also prevents the emergence of user errors by checking that the user has given all necessary inputs.

CheckBoxGroup

CheckBoxgroup-class is inherited from QGroupBox. This class creates groups of checkboxes from the items given by the setItems()-function. The class keeps track of selected items and sends selected items when the getCheckedItems()-function is called.

TimeRangeWidget

TimeRangeWidget-class is inherited from QWidget. It is container for user input fields about time range selection. The class prevents the emergence of user errors by limiting allowed inputs. This class also books up selected time range and sends it when asked.

RightSidebar

RightSidebar-class is inherited from QWidget. It's a container for all elements in view's last column. There are four pushButtons and one group of radioButtons. It sends signal when one of these buttons is pressed or new radioButton is selected and also receive it. The buttons close the program, save the selected options, show the most recent saved options and show SMEAR's statistics. When this class knows which of plot options has been selected, it sends signal to the view so that knows ask selection. This class also create groupBox of radioButtons.

CardArea

CardArea-class is inherited from QWidget. CardArea is the middle part of the view. It is container for ChartCards. This class keeps track of shown ChartCards and deletes them when area is updated.

ValueTableDialog

This is a new window, which shows min, max and average values to user.

Controller

Controller offers an interface to the view. It uses the model to implement functionalities that process data according to user inputs.

Model

Model takes care of data sources and logic. It processes data and offers it in suitable format to MainWindow. This model is passive, and only does something if Controller calls for services.

Model gets selected user inputs from view by calling get-functions. According to inputs, Model updates CardArea. Updating includes fetching data from IDataFetcer and creating new ChartCards.

Model can save preferences for user that can be pulled again later.

ChartCard (interface)

ChartCard is an interface for all different implementations. It is inherited from QChartView. This interface allows easy expansion of the program for new ChartCard implementations.

Each implementation must have four public functions. Most important function is createChartCard(). Model calls it when new ChartCard must be created. Three other functions are for setting correct headers for chart and axes.

LineChartCard

LineChartCard-class is inherited from ChartCard. This class contains only the same functions than ChartCard.

BarChartCard

BarChartCard-class is inherited from ChartCard. This class contains the same functions than ChartCard, but also it has one more function, createChart().

ScatterChartCard

ScatterChartCard-class is inherited from ChartCard. This class contains only the same functions than ChartCard.

IDataFetcher (interface)

Interface for model to access data from networks. Concrete implementations of interfaces are implemented from this interface class. Its purpose is to provide easy access for MODEL to access network data while hiding the actual implementation.

FetchData() is an interface method that concrete classes must implement because model will be using this when calling for the data. It will start the process of fetching the data from network. *FetchData()* takes trivial input as parameters while concrete classes manipulate it into non-trivial data for queries. FetchData() only gets the data from website and saves it into a private member.

Acquiring the data requires a second function getCurrentData(). It is just a getter function.

Other interface methods are *getSupportedTimeframe()*, *getSupportedGases()*, and *getSupportedStations()*. MODEL calls these to know, what are the available options for gases and stations, and what is the timeframe where there is data available. If there are no options for a certain category in some APIs, an empty vector is returned.

Originally the interface was designed to separate model from fetchers but later as we ran into problems with asynchronicity the implementation had to be altered a bit. Only fetchData() was supposed to be needed but a single function would end its process before data would be gotten from the website. This however introduces some dependencies to interfaces as they would now have to call a function in model that uses the getCurrentData() call. Model still doesn't have to know about interface implementations as all this could be done in IDataFetcher.

GetDatabaseName() was added for model to call it to know what database data it is processing.

ConcreteStatfi

A concrete interface implementation from *IDataFetcher*. This interface object is designed to get data from STATFI (https://pxnet2.stat.fi/PXWeb/api/v1/en/ymp/taulukot/Kokodata.px).

Most important method from outside perspective is *fetchData()*. It is a public method derived from *IDataFetcher*. Model can call it with parameters it wants, if they are supported by the network, and it will start a process that gets the data from network to model. This method sort of collects the function calls into one for a simpler interface.

Post() method is responsible for sending the query to network so there would be something to collect when it is finished. *ReadyRead()* is a slot that is called when the data fetched from network is ready to be read. It is responsible for manipulating the data into easier to handle data such as a vector with doubles.

GenerateQuery() is a private help method to simplify the implementation. It creates a query from gas type and time range parameters that can be sent to the network. *ArrayToVector()* is private help method too and its responsibility is to convert networks data into data that can be easily handled by model.

ToEncodedQuery() simply takes user readable input such as "CO2 in tonnes" and changes it to corresponding query string such as "khk_yht".

ConcreteStatfi utilizes Qt class QNetworkAccessManager to communicate with websites and QJsonObject to handle the received data. QNetworkAccessManager was chosen as it was the simplest and easiest to learn how to use network handler available. It made working on the project easier than using curl or http packages. As STATFI sends data in JSON form, using QJsonObject, QJsonDocument and QJsonArray made the most sense. Again, Qt implementation was used as it simplifies the code and makes it more understandable and also it was very easy to pick up on.

ConcreteSmear

A concrete interface implementation from *IDataFetcher*. This interface object is designed to get data from SMEAR (https://smear-backend.rahtiapp.fi/q/openapi-ui/).

The most important component here is the same as in Concrete Statfi: *fetchData().* Modell calls it with parameters that tells which gases user wants to see from whichs stations and what is the timeframe. *FetchData()* is inherited from the IDataFetcher interface.

Other public methods inherited from the interface are *getSupportedTimeframe()*, *getSupportedGases()*, and *getSupportedStations()*. The purpose and functionality of these functions is already explained under the title “interface”.

GenerateUrl() method takes in the starting time, ending time, gas and station, from which the data is fetched. It then generates an URL where the data is retrieved. ProcessReply() processes the data. API returns the data in json format and processReply() collects the necessary parts of the data and returns them as a vector with doubles.

GetMin(), *getMax() and getAverage()* returns minimum, maximum and average values for specific gas and all the stations that are chosen. The gas which it returns the values for is defined by private attribute *currentGas_*.

The *fetchData()* -function is called once for every gas, and qt being asynchronous caused problems. ConcreteSmear has different vectors for each gas for that reason. ConcreteSmear also knows in which order the stations are listed, so it can return the values in right order even though they are not always processed in that order.

Work timeline

We have almost been able to stick our original design, but some changes have been made. We decided to change design pattern to MVC and use adapter pattern where we originally planned to use builder. Interface class works as sort of a template for the actual implementations. While Model might have multiple adapters, they all work the same way and can be called with the same structure. Because the project is still in progress, we might have to make some changes to implement the remaining features.

At the beginning of designing this project, we chose to use Model-View-Delegate (picture 2).

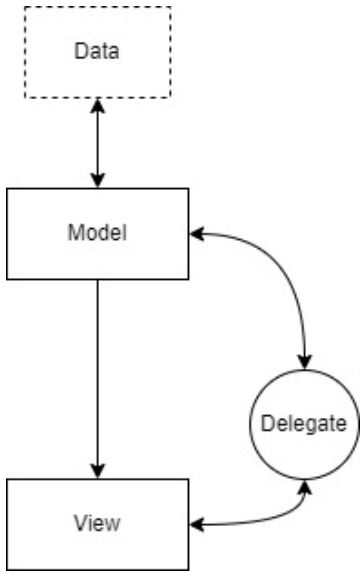

Figure 2: Mode-View-Delegate

This pattern is QT’s approach, and we code with QT so we thought that it would be obvious choice and easy pattern. We decided to change the “delegate”-model to the MVC (picture 3) because we didn’t understand the purpose of delegate.

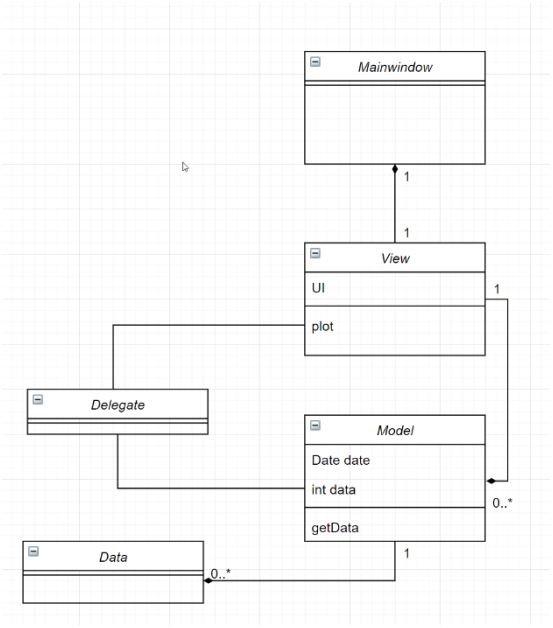

Figure 3: Model-View-Delegate class diagram

MVC is the most well-known design pattern, and it separates the UI and the software's other parts so it's easy for an outsider to understand the code. The separated parts make it much easier to test the code independently. Also, a well-designed interfaces and separated parts will facilitate further development of the software.

To get data from APIs to model we planned to use builder pattern (picture 4) at first. After reading more about the difference between creational and structural patterns we noticed that we need to use a structural one.

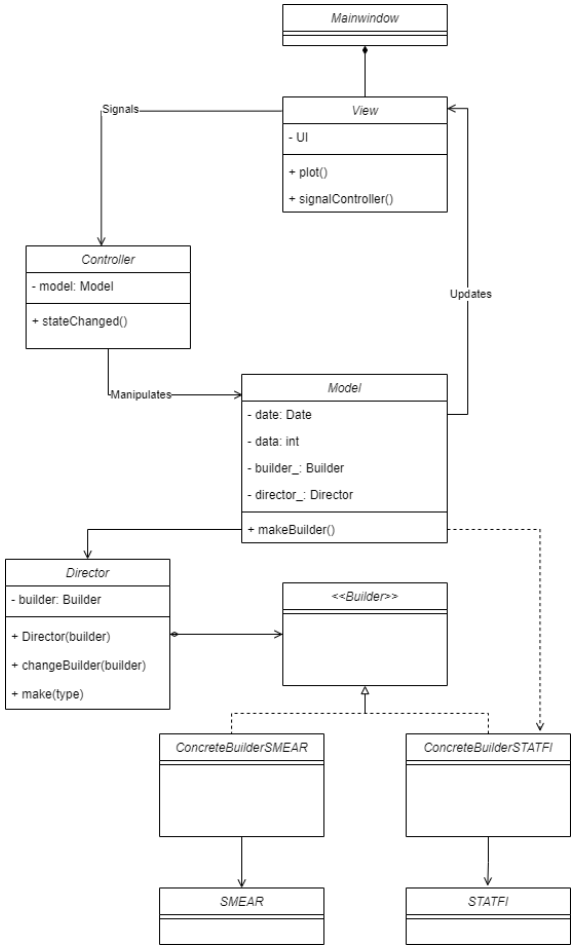

Figure 4: Class diagram with builder

We decided to use adapter (picture 5), both APIs has their own adapter which are inherited from the same interface. It's a simple implementation and makes it easier to develop and test the program in parts.

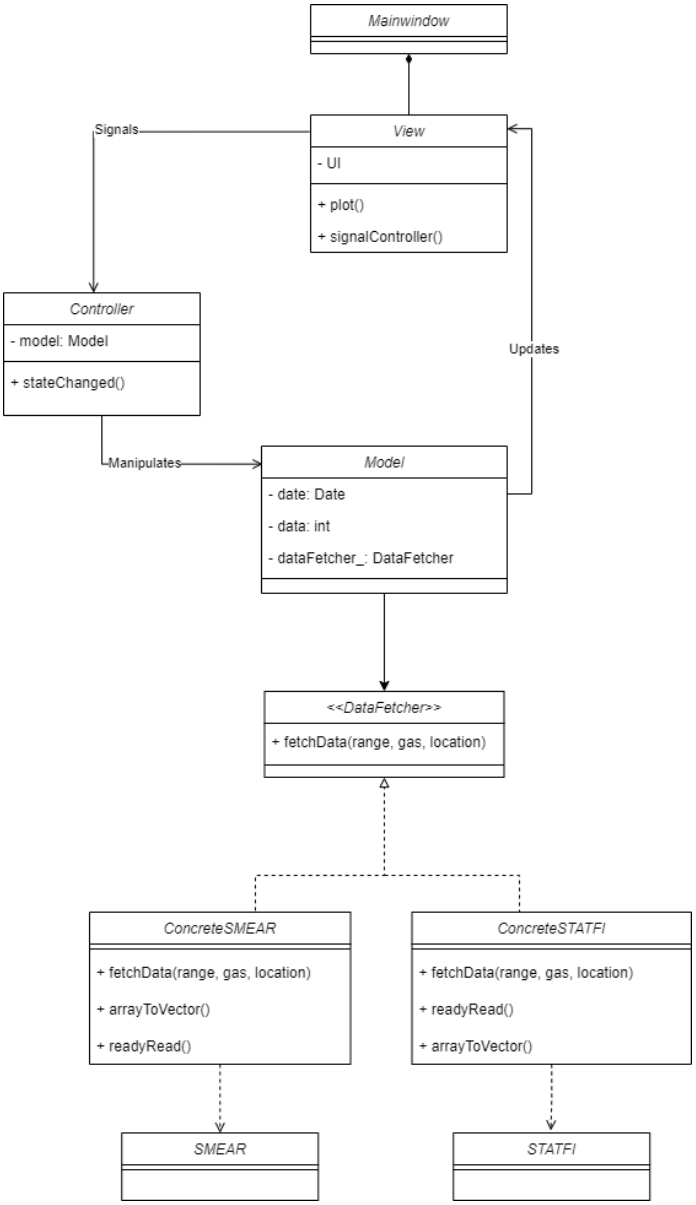

Figure 5: Class diagram with adapter

We also considered bridge, but it felt unnecessary complicated, so we ended up using adapter.

This design is simple and a known pattern but because of lack of previous experience with asynchronous operations this was not going to work. The solution to this problem does not complicate implementation that much but concretes do need to be aware of the model to be able to communicate with it when asynchronous operations are finished.

The UI also needed more classes (picture 6). User input would have looked messy and crowded if implemented only in MainWindow. Dialogs were introduced to simplify the interface and make it clearer for user and easier to maintain.

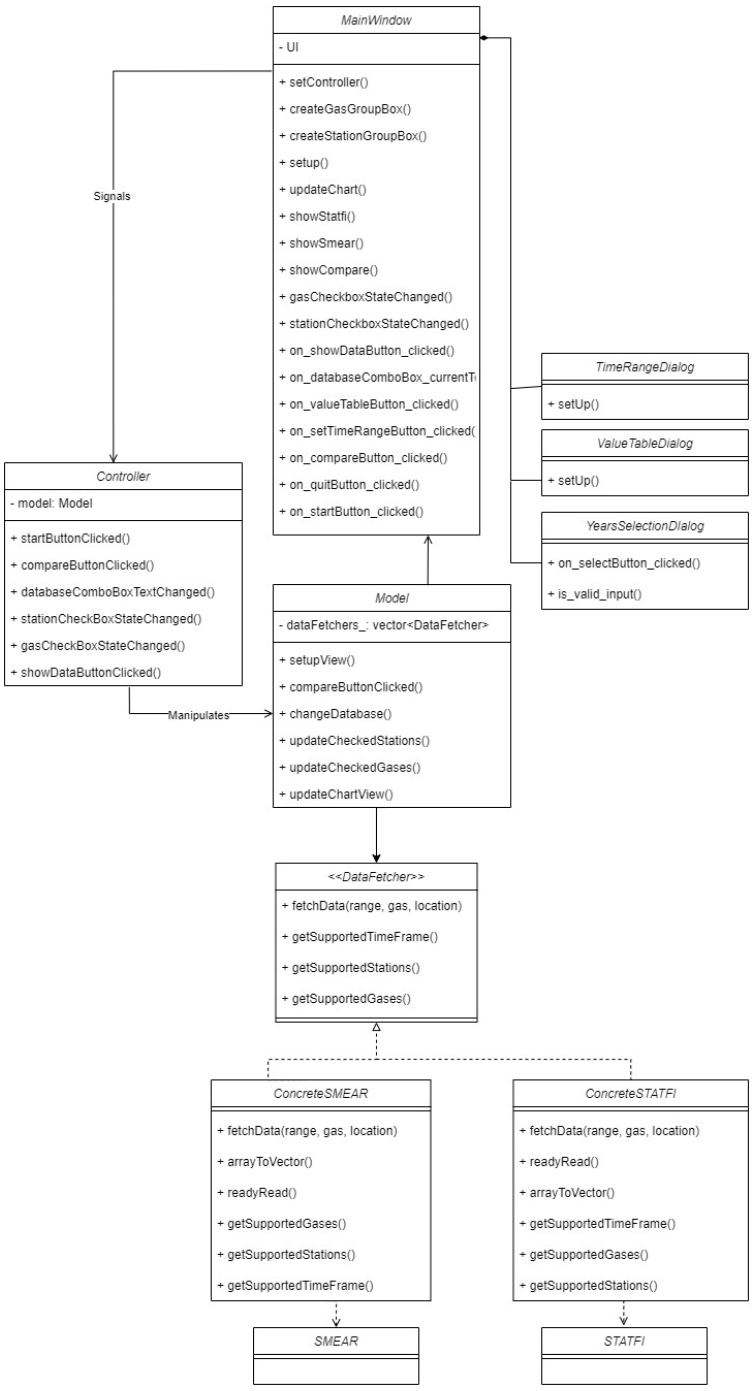

Figure 6: Class diagram with functions and some classes added

Eventually we ended up removing almost all dialogs (picture 7). We kept only ValueTableDialog to simplify the user interface. However, to simplify the code in MainWindow, we decided split MainWindow to the three main parts and implement own classes to these parts. New class diagram is in figure 7.

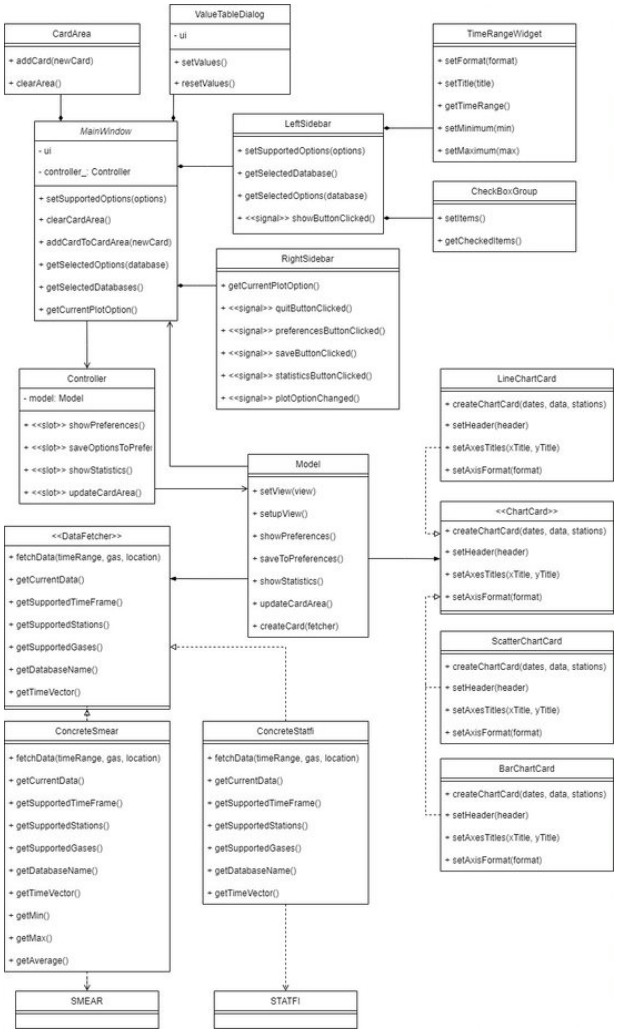

Figure 7: Final version's class diagram

We created new widgets and implemented them in classes so that they can also be used for other purposes in the future. One of the new widgets is TimeRangeWidget. We deleted TimeRangeDialog and YearSelectionDialog and decided to implement time periods as widget. So SMEAR's and STATFI's time range input fields can both use the same class implementation. By separating code into smaller classes, we try to follow the *open/closed principle* (OCP). Smaller components are easier to modify by adding new functions without breaking code somewhere else.

We added ChartCard-interface which allows easy expansion of the program for new ChartCard implementations. In this case, the program can easily implement more additional plotting options. This design decision follows *Liskov substitution principle* (LSP). In this case, the program doesn't have to know exactly which subtypes are used to draw to graph.

Self-evaluation

The original idea of our software has sticked. During our first meetings we discussed about an interface that would separate the MVC from the objects that would act as the messengers of data. We were able to implement something along that design. Some different patterns were discussed such as bridge, but our variant of adapter was the simplest. We were mostly able to expand our ideas from the original without changing existing code.

We are confident in our implementation because the design has supported the implementation. From the beginning, we have had weekly meetings with the group. In meetings we have planned together. We have noticed everyone's implementations and new proposals. As all members of the group have been aware of the joint designing from the beginning. We had the most discussion about the transfer of data between classes. Compared to the first plan, the biggest change has been in the number of classes. Some dependencies had to be created to implement all the functionalities. This was mainly due to synchronization issues, and they helped to get past them. There would have been too many functions individual classes, so we decided to add more classes for better software designing. We are most satisfied with the final solution.