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.

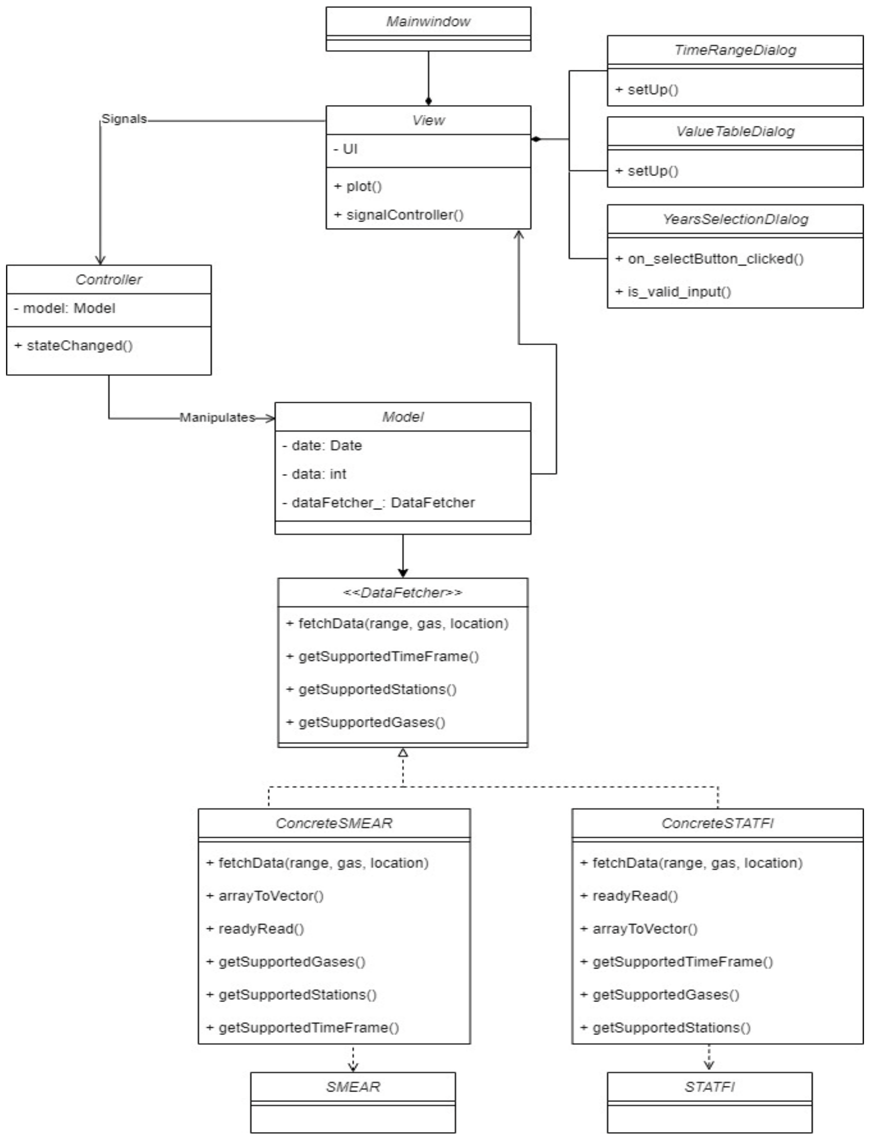

Figure 1: Class diagram

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.

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.
(pitäisikö toiminnan tulla takaisin controllerin kautta?) ei?

TimeRangeDialog

This is a new window, where user can set a time range in days. This class prevents user from making input errors by locking unavailable days.

ValueTableDialog

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

YearsSelectionDialog

This is new window, where user can set a time range in years.

Interface (IDataFetcher)

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 without model knowing the actual implementation behind concrete classes. *FetchData()* takes trivial input as parameters while concrete classes manipulate it into non-trivial data for queries.

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.

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.

ConreteSmear

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.

Self-evaluation

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).

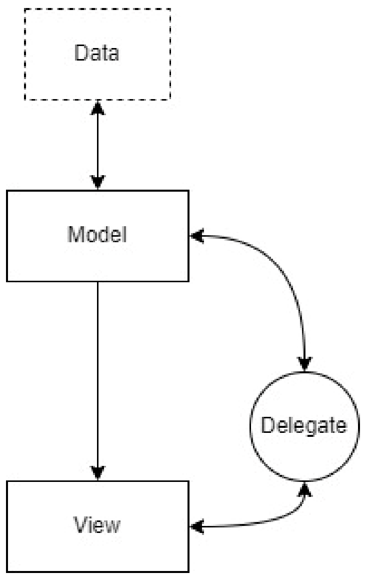

Figure 2: Mode-View-Delegate

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.

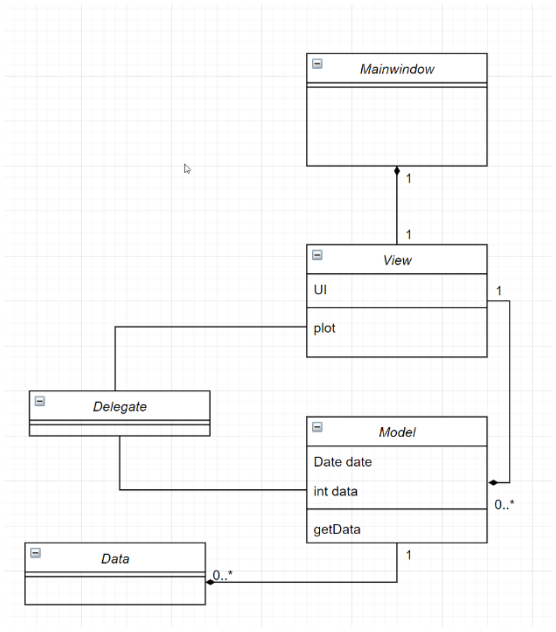

Figure 3: Model-View-Delegate class diagram

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.

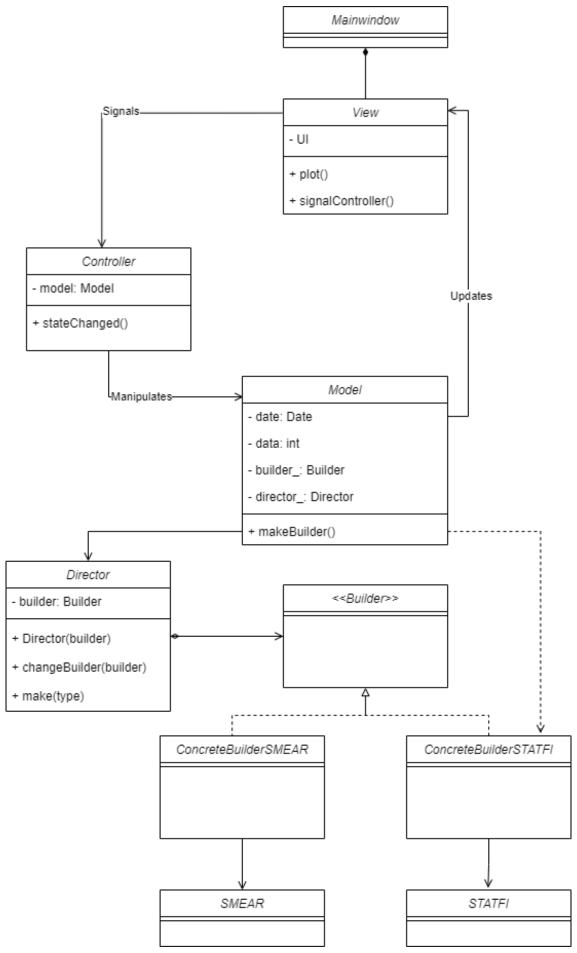

Figure 4: Class diagram with builder

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.

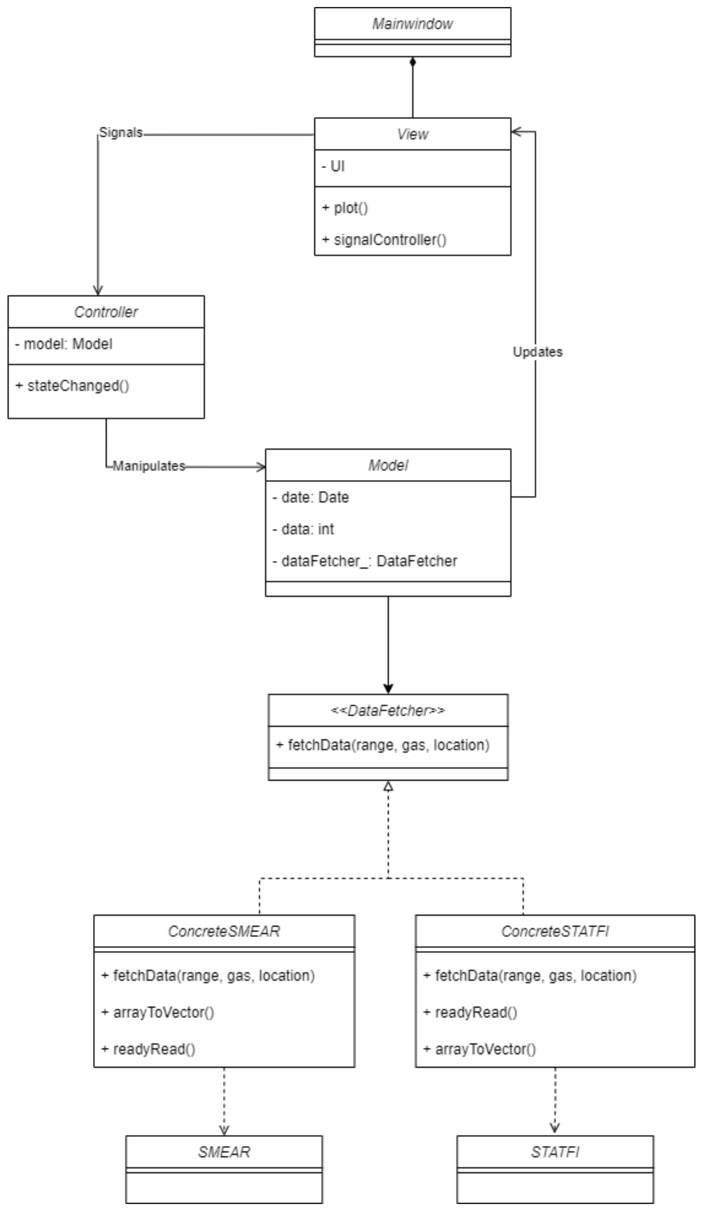

Figure 5: Class diagram with adapter

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

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.

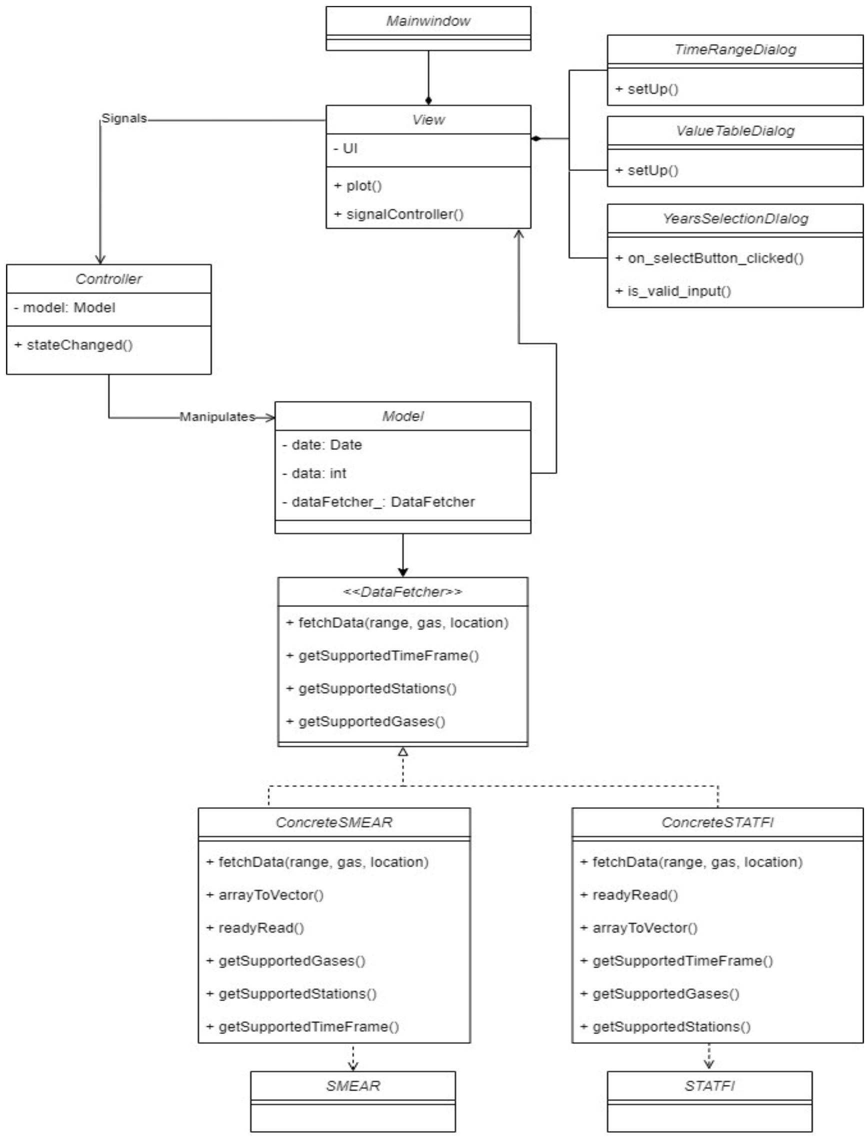

Figure 6: Class diagram

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 except the designing to continue to support the implementation.

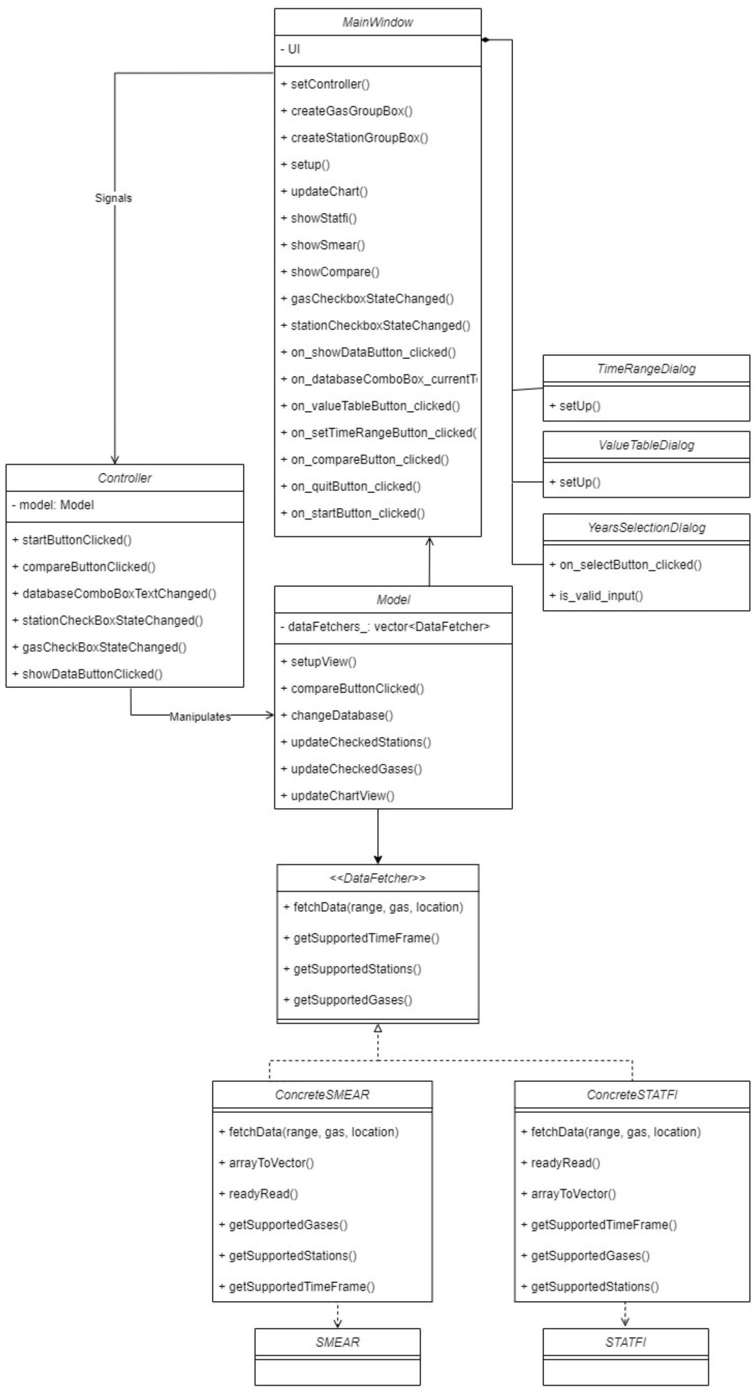
MainWindow
- UI
+ setController()
+ createGasGroupBox()
+ createStationGroupBox()
+ setup()
+ updateChart()
+ showStatfi()
+ showSmear()
+ showCompare()
+ gasCheckboxStateChanged()
+ stationCheckboxStateChanged()
+ on_showDataButton_clicked()
+ on_databaseComboBox_currentT
+ on_valueTableButton_clicked()
+ on_setTimeRangeButton_clicked(
+ on_compareButton_clicked()
+ on_quitButton_clicked()
+ on_startButton_clicked()
Signals
TimeRangeDialog
+ setUp()
ValueTableDialog
+ setUp()
YearsSelectionDialog
+ on_selectButton_clicked()
+ is_valid_input()
Controller
- model: Model
+ startButtonClicked()
+ compareButtonClicked()
+ databaseComboBoxTextChanged()
+ stationCheckBoxStateChanged()
+ gasCheckBoxStateChanged()
+ showDataButtonClicked()
Manipulates
Model
- dataFetchers_: vector<DataFetcher>
+ setupView()
+ compareButtonClicked()
+ changeDatabase()
+ updateCheckedStations()
+ updateCheckedGases()
+ updateChartView()
<<DataFetcher>>
+ fetchData(range, gas, location)
+ getSupportedTimeFrame()
+ getSupportedStations()
+ getSupportedGases()
ConcreteSMEAR
+ fetchData(range, gas, location)
+ arrayToVector()
+ readyRead()
+ getSupportedGases()
+ getSupportedStations()
+ getSupportedTimeFrame()
ConcreteSTATFI
+ fetchData(range, gas, location)
+ readyRead()
+ arrayToVector()
+ getSupportedTimeFrame()
+ getSupportedGases()
+ getSupportedStations()
SMEAR
STATFI

Figure 7: Class diagram with functions