Olivia Saukonoja K438991

Miina Rautakorpi H291552

Malla Nyrhinen 50287776

Sini Kivioja H293970

Design document

1. High level description

The software monitors real-time data on greenhouse gases and compares current data with historical data on greenhouse gases. Real-time data comes from SMEAR API and historical data comes from Statfi API.

We are using C++ language, QtCreator and Qt's libraries for the project, because everybody in our group has some experience of it. To run this program, you can use TUNI's virtual desktop and QT Create. For fetching data, you need to open greenhouse_gas.pro in QtCreator from greenhouse_gas folder and run main.cpp.

For plotting and visualizing the data we are using QCustomPlot library. QCustomPlot is made for plotting and data visualization. We chose this library because it's suitable for 2D plots, graphs, and charts. It was downloaded from qcustomplot.com and the needed header and source files were added to the code. QCustomPlot was used in PlotWindow-class.

Chart 1 will explain the high-level description and show in what order the classes are created. We selected this structure, because it's easy to keep the UI and the data handling separate from each other. Error handling is also clearly in its own class. This way no class has too big responsibility of the program, and we can keep it running smoothly. SMEAR and Statfi's APIs are their own classes because SMEAR data is fetched with get-request and Statfi data with post-request.

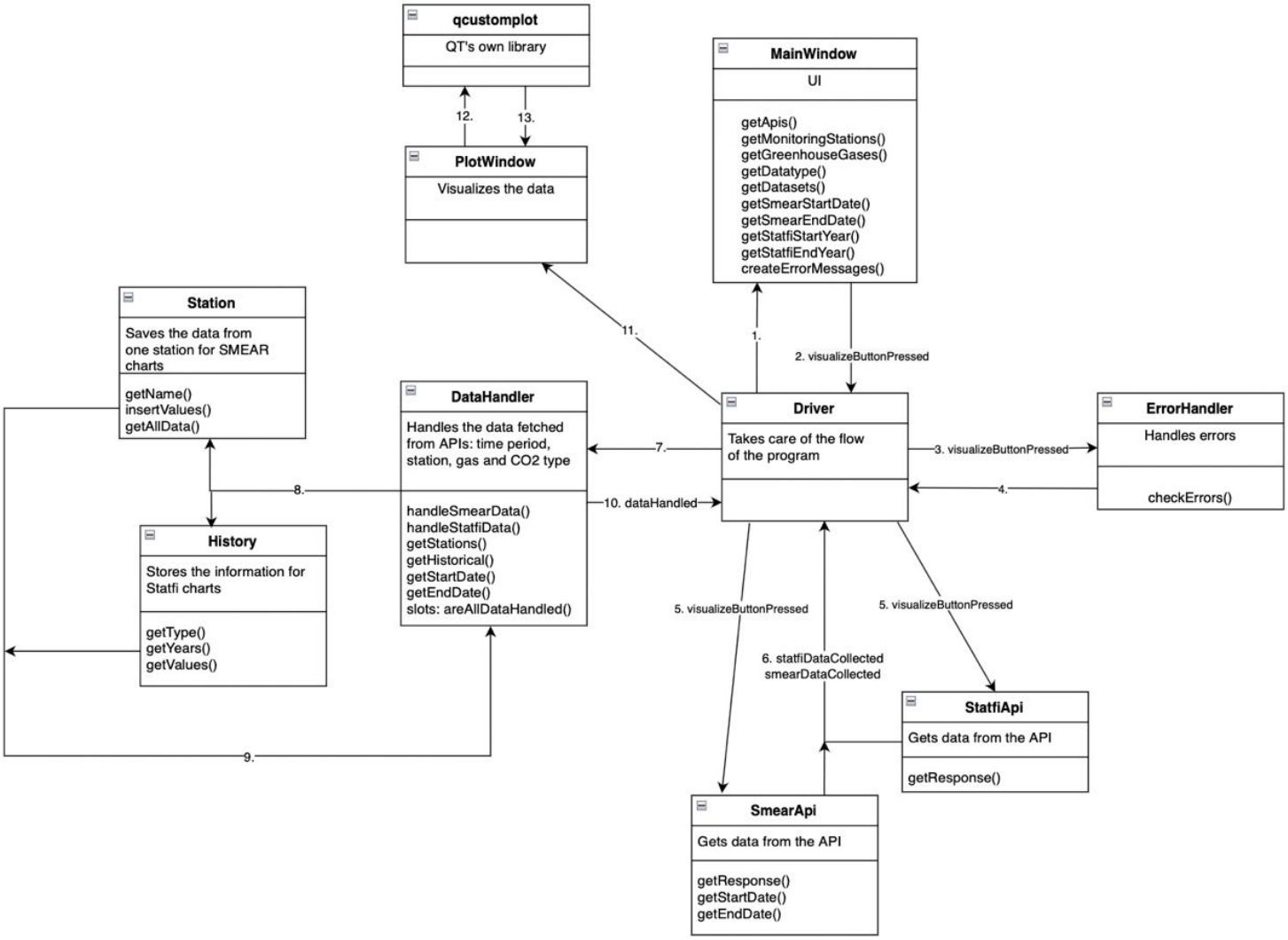

Chart 1: Class diagram

We used multiple QTs own classes in our project. In dataHandler, we used QObject, QJsonDocument, QJsonObject, QJsonArray, QDateTime. These json-classes were necessary because a use of json-formatting. QDateTime made it easier to fetch the current date and time. In MainWindow we used QMainWindow to enable the UI, and QCheckBox, QRadioButton and QPushButton to enable a smooth use of different buttons.

In PlotWindow we decided to use QWidget to implement widgets into our code. To enable API fetching, we used QNetworkReply in smearApi and statfiApi. QNetworkAccess was the best solution we found for making the get- and post-requests.

The QT-classes we used were chosen because of their ability to make the project work better. By using these classes, we were for example able to use APIs more effectively.

2. Boundaries and interfaces

Driver starts the MainWindow, that starts the user interface application. After MainWindow's Visualize-button has been pressed, it starts the errorHandler-class. If there are errors, MainWindow shows error messages. If there are no errors, driver creates smearApi- and statfiApi-classes. Api classes fetch user's choices straight from MainWindow-class, and based on user's choices, smearApi uses get-request to access SMEAR's API and statfiApi uses post-request to access Statfi's API. Then driver creates dataHandler-class that fetches the data from smearApi and statfiApi and modifies the data. DataHandler creates station- and history-classes that handle specific data for SMEAR's and Statfi's charts. After that, driver creates PlotWindow-class which fetches the modified data from dataHandler, and then creates the visualizations.

3. Components and responsibilities

The responsibilities of the components are shown in chapter "High level description" and chart 1.

Driver takes care of the flow of the program, i. e. it creates classes in the right order. MainWindow starts the UI and asks the user what kind of diagram they want to see. ErrorHandler checks if there are errors in user's selections and creates the error message related to the error. For example, if the user selects dates where there is no data available (dates are in the future or too far in the history), MainWindow shows an error message to the user.

SmearApi and statfiApi fetch the user's selections from MainWindow and based on them, it fetches data from the correct API's. DataHandler fetches the API's data from smearApi- and statfiApi-classes, and modifies it to the format, that is applicable to Qt's libraries. History-class stores one Statfi's chart's data. Station-class stores SMEAR's one station's data.

PlotWindow fetches the data from dataHandler and makes visualizations based on it. PlotWindow uses QCustomPlotlibrary when making charts to the UI. PlotWindow also starts the new UI window that shows the visualizations.

4. Design solutions

4.1. UI Design Solutions

We decided to put the SMEAR and the Statfi decision at the top of the front page, because it effects on what to ask next from the user.

If the user selects SMEAR, they will be asked about the time (in calendar form), monitoring station (Hyytiälä, Kumpula, Värriö), greenhouse gas (CO_2, SO_2, NO_x) and different ways to present the data (raw, average, minimum, maximum). Different monitoring stations will be represented in different graphs, and different greenhouse gases will be represented in different lines inside those graphs.

If the user selects Statfi, they will be asked about the time in years and CO_2 datasets (in tonnes, intensity, indexed, intensity indexed). Then the data will be shown in a graph. Different datasets will be represented in different lines. We chose to present the times in years that have actual data in the API, and this way the user is not able to make mistakes so easily.

If the user wants to compare the SMEAR and the Statfi data, they can select everything related to the data. Then the SMEAR data and Statfi data will be shown in separate graphs (meaning 2-4 different graphs, depending on how many monitoring stations have been selected).

In case user selects something that leads to some data being unavailable, it is shown as “grey” to indicate that the user cannot select it. This also leads to fewer mistakes and confusions when using the program.

4.2. Software design pattern

The design solution we implemented was Model-View-Controller framework. This was our choice because it fits well with an application with UI and APIs. This solution separates the application into three main logical components, that can be seen in chart 2.

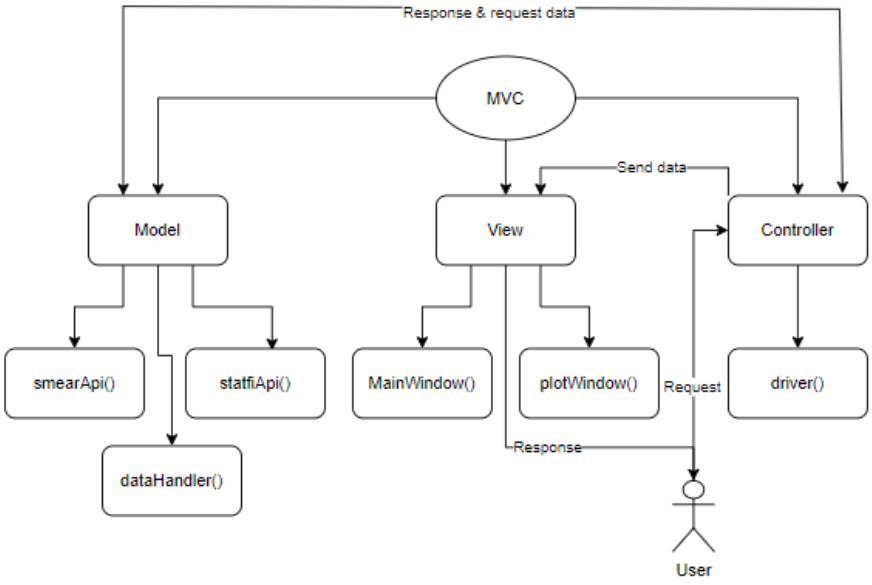

Chart 2: Design framework

View is a component that contains the UI, in this case MainWindow and PlotWindow. They control the UI and presents the information that is fetched from model component. Controller in our program is the driver class (and classes related to that). The model is smearApi, statfiApi and dataHandler, as they contain the actual information fetched from the API.

There are few benefits with using Model-View-Controller in our program. First being that the MVC methodology allows to modify the application if needed. Adding new classes is relatively simple and the design solution would not change too much. Other benefits include that it supports test-driven development (TTD). This means the testing process is easier and better. This also meant it was easier to work together in a group, because we could separate the model, view and controller to different persons and the testing process went more smoothly. We also did a lot of testing in early parts of development – this way we could make sure we didn't do any big mistakes in the beginning that would make the process more difficult in the end.

5. Self-evaluation

We have been able to stick to the original design quite well. The UI in the program is really like the prototype we made before any coding. The difference is that there are now two calendars for SMEAR, and spin boxes for Statfi. Because of different types of data, we get

from the sources, we changed the calendar view a bit. This also meant that the user cannot make so many mistakes as we had given the years the correct Statfi years.

Design corresponds to quality so that the classes are clear, and the mistakes are easier to find. Responsibilities of the classes are clear and well defined. It was easy to start implementing the program based on prototype.

With the class structure we were also almost able to stick to the original plan. The changes we made were making the driver class, history class and station-class. This is because we realize the program will be better and more efficient when the classes are broken down to smaller classes. Driver helps with the program running smoothly.

Even though we did some minor changes to the design, we think the planning was a success. We believe it is good to keep the software creation process a bit open and "agile-like". This way the finished program will be better, because the process has been shifted to the left and we have been able to make the program better along the way. By doing testing and implementing better solutions while doing the project, we think the final project will be better.

From now on, it would be easy to implement new APIs to the program, and it would be easy to update the program. For this the programmer should just add a new API-class and modify the MainWindow to ask the user the new choices. PlotWindow should be modified according to the new data so that it would draw the charts. DataHandler would need a new function that would handle the new source of data.

The group worked well together, and we had meetings mostly 1-2 times a week, where we checked the program's status, coded together, and shared the new responsibilities for each person. The team worked seamlessly, and we helped each other's when needed. We also had a rough timeline on when certain classes needed to be ready, and we sticked to the plan.