SOFTWARE DESIGN PROJECT

COMP.SE.110 Software Design (spring 2022)

Aatu Laurikainen, aatu.laurikainen@tuni.fi, 283348

Aleksi Sirviö, aleksi.sirvio@tuni.fi, K444293

Mohammad Hassan, mohammad.hassan@tuni.fi, 50457535

Sakari Klemola, sakari.klemola@tuni.fi, 273086

Course assignment

22.4.2022

TABLE OF CONTENTS

1 INTRODUCTION

The goal of the project is to create and deploy a software program that visualizes data from stations that monitor greenhouse gases. Two types of data will be presented in the application: one is real time data provided by SMEAR[1] and another one is historical data provided by STATFI[2].

The application will allow the user to monitor real-time data, check historical values, and compare the current situation to history. It will break down averages based on data from STATFI. The user will be able to select the time period and one specific or more than one monitoring station to visualize the data. This software project is for a university course named Software Design and the project is developed by a group of students. The purpose of this document is to record the development of said software.

2 TECHNOLOGIES

The exercise group working on this project was formed based on the preference of working with C++. Thus, C++ is the chosen programming language for the implementation of the application. Qt[3] is the group's graphical user interface library of choice, as the group members have experience working with Qt in previous courses. Using a robust graphical user interface library like Qt speeds up the project and allows the team to focus on the business logic of the application. Various libraries inside Qt are used for specific tasks. These include QChart, QLineSeries, QDateTimeAxis, and QValueAxis for drawing graphs, QDate, QDateTime, and QValidator for interpreting user inputs as dates, and QtNetwork for doing network calls. QJsonObject and QJsonArray are used for parsing JSON data. Finally, QFile is used for saving user loadouts into a local file.

3 STRUCTURE

The application will be structured according to the model-view-controller (MVC) design pattern. The MVC pattern has a number of benefits, such as helping make the code base

clearer, more structured, and easier to iterate on and expand in the future. The division of responsibilities inside the program naturally translates to a division of responsibilities among the developers. Traditionally Qt uses an MVC variant called the model/view architecture, which utilizes a delegate doing similar work as the controller, but in a smaller capacity. The plan is to create a controller with more responsibility than the traditional Qt delegate. In this document, the terms controller and delegate are used interchangeably. The three main components and their responsibilities will be discussed next. Image 1 describes the high-level structure of the application with a very simple component diagram.

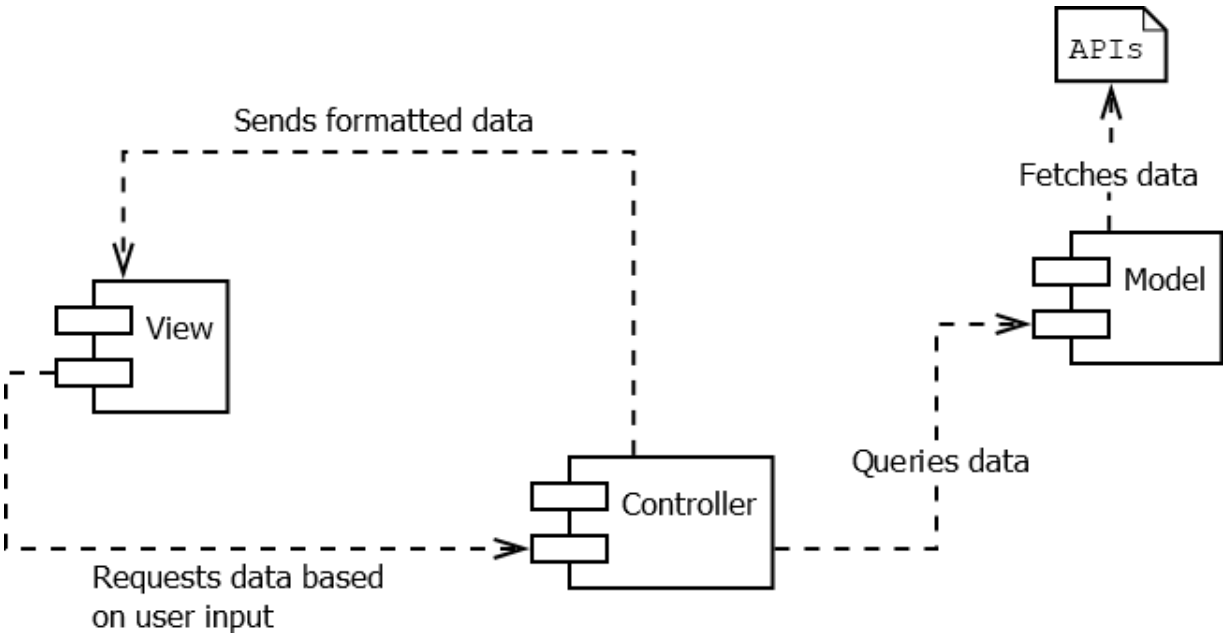

Image 1: Application structure as a simple component diagram

In image 2 below is a class diagram of the entire application. This diagram shows all the dependencies between classes as well as all members of each class aside from constructors, setters and getters, and custom operators such as comparison and assignment operators. The diagram is also included as a png file in the same directory as this document.

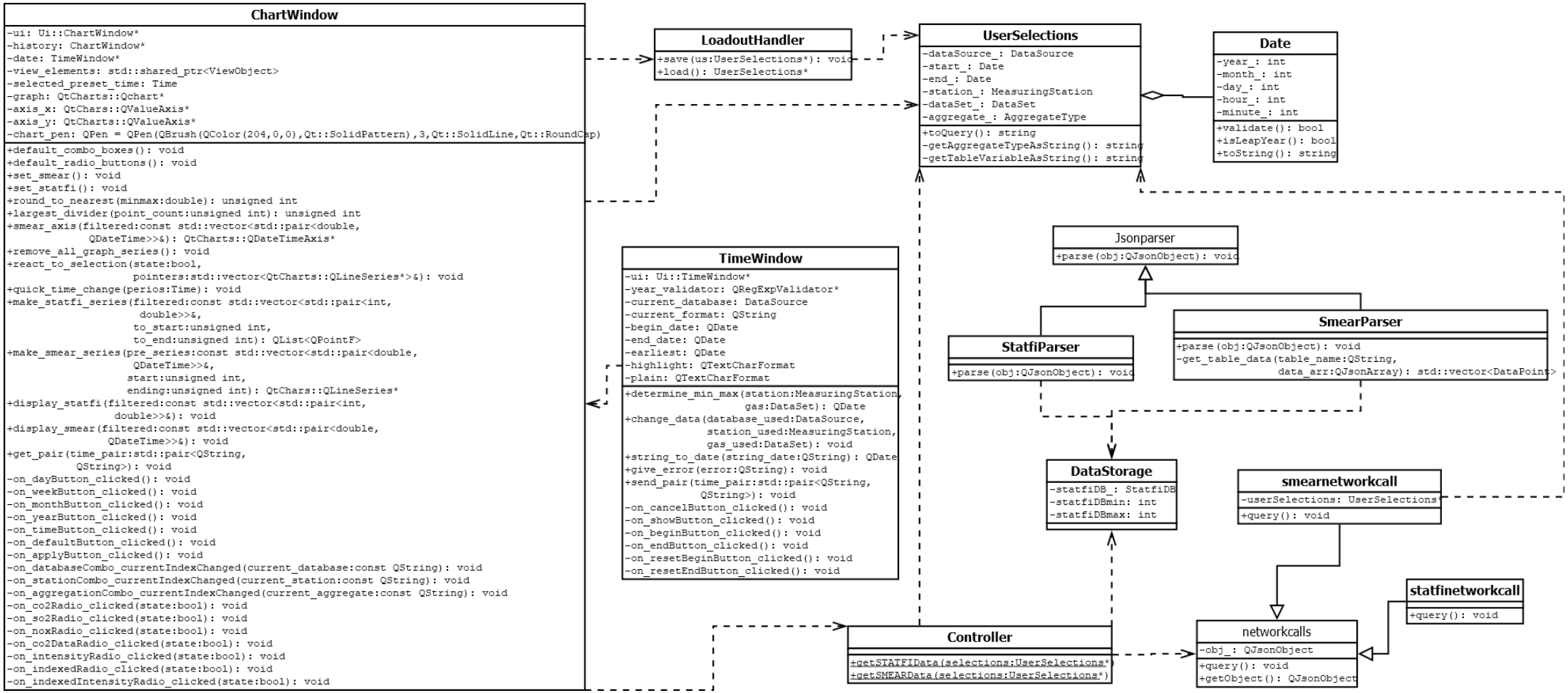

Image 2: Class diagram of the whole application

3.1 Model

The model is a representation of the data used in the application. The responsibilities of the model include:

1. Fetching data via the APIs of the given databases
2. Parsing the JSON data and arranging it into suitable container(s)
3. Storing the data and offering a simple interface for the controller to access it

To follow the single responsibility principle (SRP), the model must be divided into multiple classes. For example, one set of classes will be entirely responsible for accessing the databases and fetching the data, while another class will maintain the containers for the data. Since fetching data from SMEAR and STATFI is different, a separate fetcher classes are implemented for both. These classes implement a common data fetcher base class and follow the strategy design pattern.

A JSON parser class is responsible for parsing the fetched data into data structures that fit our needs. Likewise, different parsing strategies are needed for SMEAR and STATFI data. Both the data fetchers and the data parsers follow the open-closed principle (OCP), as more strategies could easily be added for both by adding new subclasses without touching the existing ones. However, the actual difficulty of adding support for another data source depends heavily on the data source. For example, The Liskov substitution principle (LSP) has also been carefully followed wherever inheritance is used.

Finally, there is a data storage class, the only purpose of which is to store and offer access to the parsed data. The data storage follows the singleton pattern, as only one data storage needs to exist during the runtime of the program. A key difference between STATFI and SMEAR data handling is that since the STATFI data is much simpler and there is much less of it, all of it is fetched and parsed at the launch of the application. The data is then stored in the data storage, where it is always readily available via the singleton. On the other hand, SMEAR data is only fetched and parsed when the user has made specific selections in the GUI and wants to update a graph.

3.2 View

The view is responsible for the graphical user interface (GUI) of the application. The view maintains the UI components and waits for user input. The way the application is being designed, there can be 1 to n number of views, all of which represent the model differently. This makes comparing different representations of the data convenient for the user. Separate windows for showing separate views can be opened by using the New Window button accessible via the dropdown menu behind the Window button in the top bar of the application. The GUI is mostly implemented inside a chart window class, but there is also a separate time window class that is used when the user inputs start and end dates for the data they seek.

In addition to that, there is a user selection class that keeps track of user actions in the application. The data inside the user selection class is formatted as such that it can conveniently be sent to the controller and used to either filter STATFI data from the model or to form a SMEAR query based on what kind of selections the user has made. There is also a utility class called Date, that is used to represent and easily compare different points in time. We want back and forth on whether we should keep the Data class, as the qDate class in Qt does many of the same things. We ended up keeping it for more utility, even though some overlap exists between the use of the two.

There is also a separate class for handling user loadouts. The user can save and load their favorite loadout using the dropdown menu accessible via the Window button in the top bar. A local JSON file is used for storing the saved loadout. The graph can also be saved as a local png file using the same dropdown menu as mentioned before.

Looking at the diagram in image 2, the UI classes may look like God classes due to the huge number of methods required to handle inputs and generate graphs. A lot of the GUI information in QT is private or protected, so it isn't feasible to divide these methods into other classes. Also, if you look at the actual number of dependencies between the UI classes and other classes, the number is quite low. In general, we are quite happy

with the low number of dependencies, which indicates that the software is quite carefully designed.

3.3 Controller

The controller is the component that connects the view to the model. It contains a big part of the business logic of the application. The responsibilities of the controller include:

1. Reacting to data requests from the view
2. Accessing or querying data from the model
3. Operating on the data based on the user input
4. Passing the processed data back to the view

The controller is implemented as a class with a set of static methods that are called by the view as needed. The controller itself doesn't store any data and no instances of the controller need to be created. Separate methods exist for processing STATFI and SMEAR data.

4 BOUNDARIES AND INTERFACES

Image 3 shows data flow of components throughout the application. A component diagram approach has been used to design this data flow. This diagram may be unnecessary after the inclusion of the detailed class diagram, it was kept as it shows component interaction at a higher, more easily understandable level.

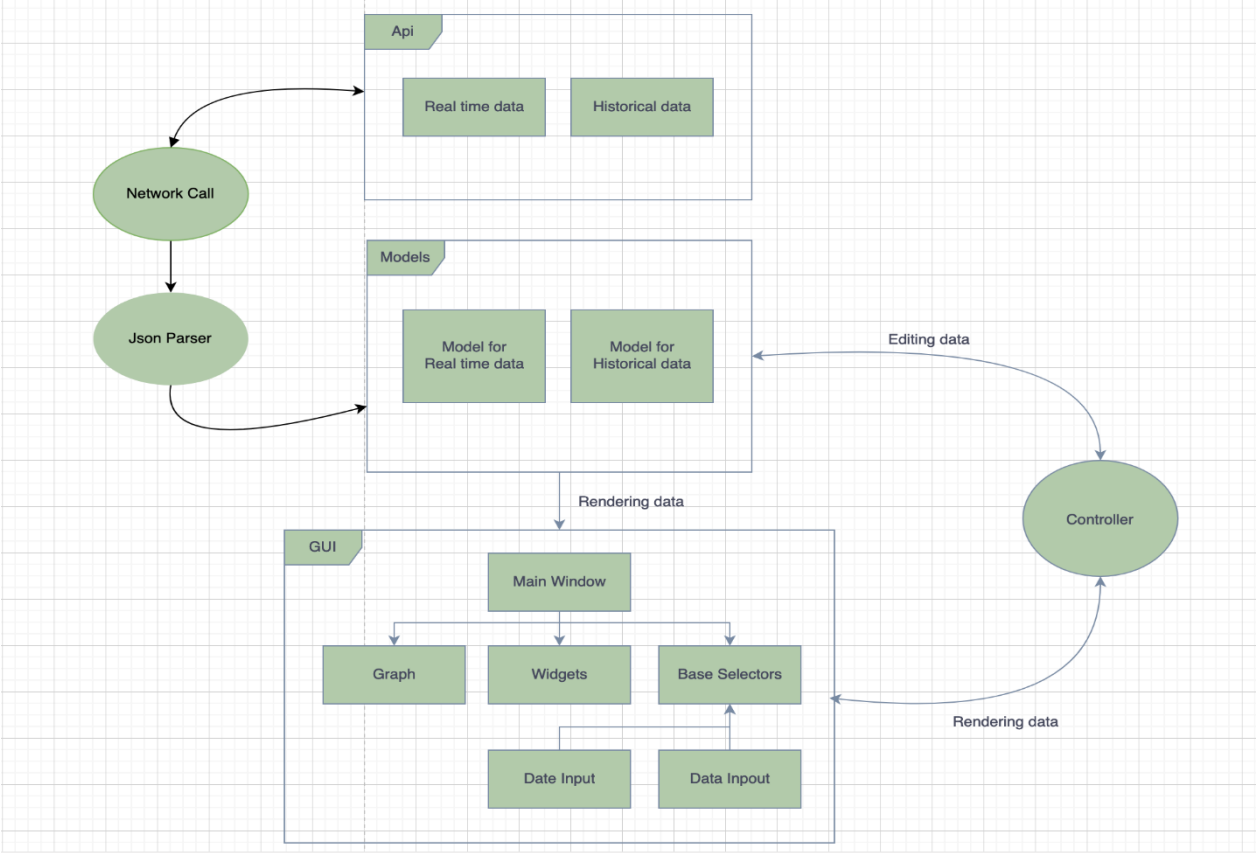

Image 3: Application components interaction in component diagram

Since two kinds of data have been provided from API one is real time data and historical data. Therefore, only two models are needed to synchronize the data in the application. A network call brings the data and through a JSON parser data satisfy the models. In the GUI main window there are graphs to visualize desired data, widgets to select certain options for data and base selectors where date and data inputs are accepted for desired data. A controller has been established to filter data according to user selection from models. After filtering the data, controller send the data to be rendered in GUI.

5 GRAPHICAL USER INTERFACE

On start-up the program opens the main view. Normally there is no data at start-up, but here it is demonstrated how it looks. Most of the available space is taken by the graph where the data is shown. On the x-axis is the given timeframe and on the y-axis is the emission amount. On said graph the data will be shown as points which are connected to each other by lines. The user can compare different gasses by opening a new

window from the navigation bar. We decided to make the graph take a lot of space because doing so will ensure that a lot of data can be shown at once without it becoming hard to discern and that small changes are easier to notice.

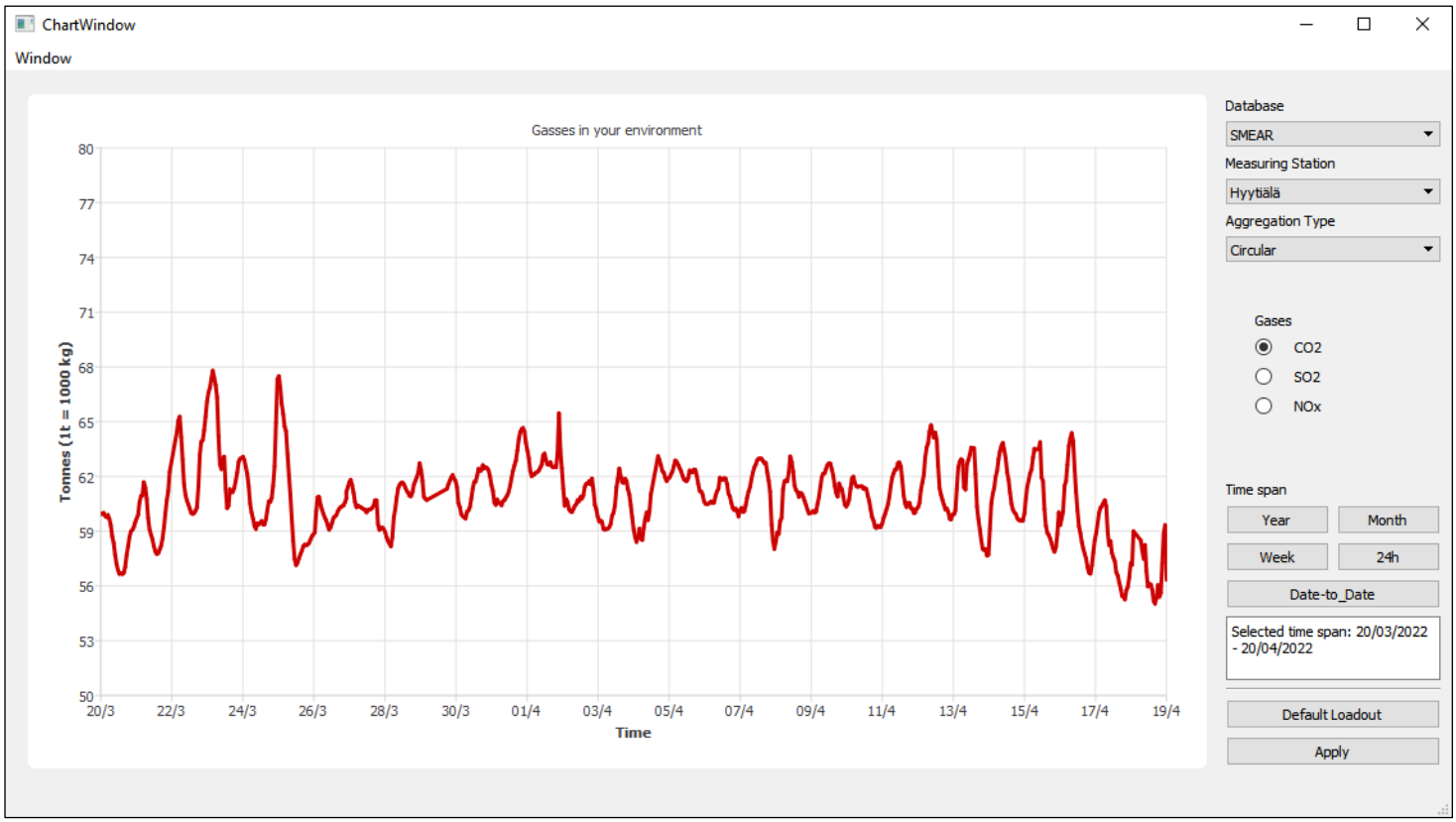

Image 3: GUI When SMEAR is the chosen database

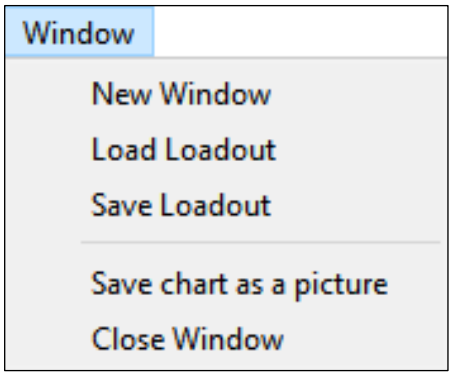

Image 4: Navigation bar menu opened

On the right side of the main view are different buttons and boxes with which the user chooses the data they want to see in the graph. At the top there is a drop-down menu from which the user chooses the database from where the data is stored. The next drop-down menu is for choosing the distinct measuring station from where the emission measurements have been taken. Third drop-down menu is for the aggregation type the user wants to select. Under these there are radio buttons for different gases. Only one gas can be selected at the time. However different windows can have different gases selected.

Vital part of showing the data in the graph form is for the user to click the Apply button when they are done with their query. This activates the program to fetch the wanted data and put it into a visual form. This design decision is done to prevent the user from accidentally fetching any unnecessary data. If the view would update every time a check-box is changed, or radio button pressed the program would probably freeze for a few seconds due to the lack of multi-threaded programming.

As the database is changed from SMEAR to STATFI in the drop-down menu above, the possible selections will also change as seen in image 5. The drop-down menus for different measuring stations and aggregation types will also be absent when STATFI is selected as the database. This is because STATFI doesn't give out data from their individual measuring stations nor does it have any aggregation to their data. As a final difference the Date-to-Date button is changed into Year-to-Year button. This will also affect the window that pops up as the buttons is pressed as the user input for this will be different.

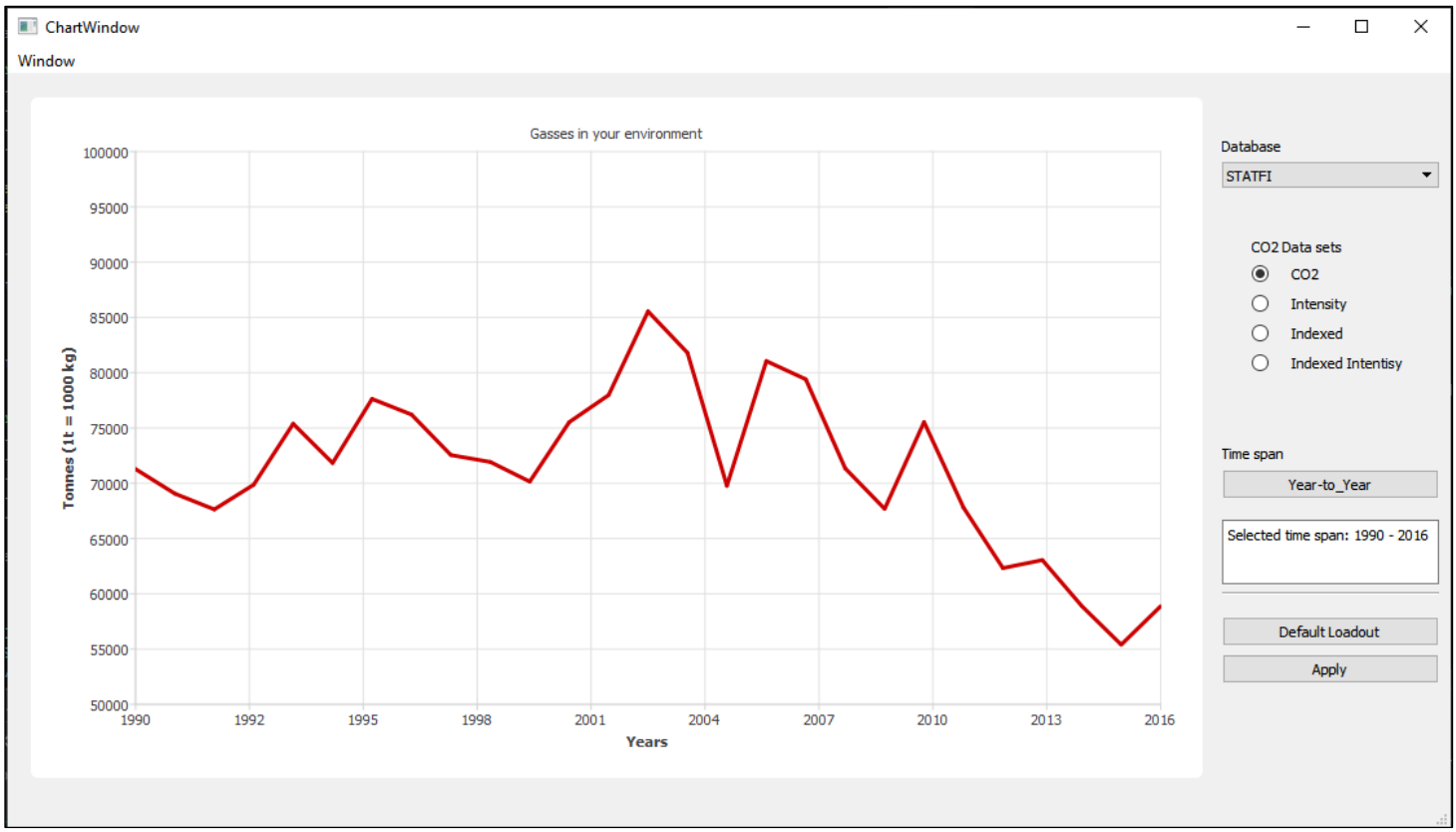

Image 5: GUI When STATFI is the chosen database

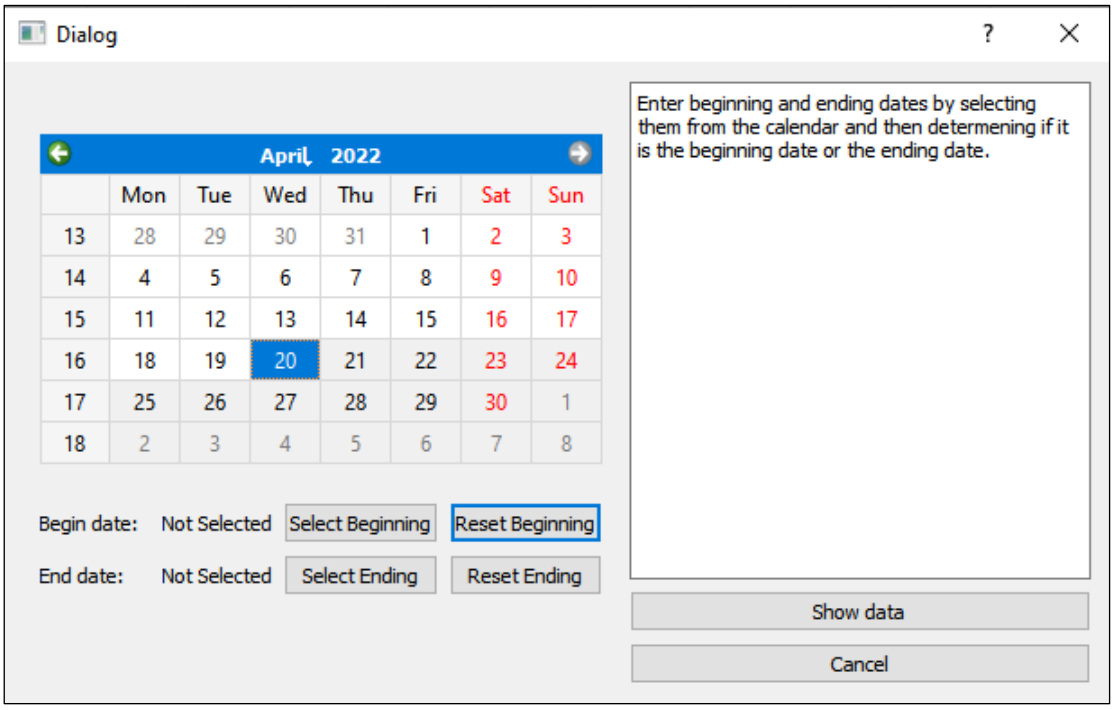

Image 6: The Date-to-Date window

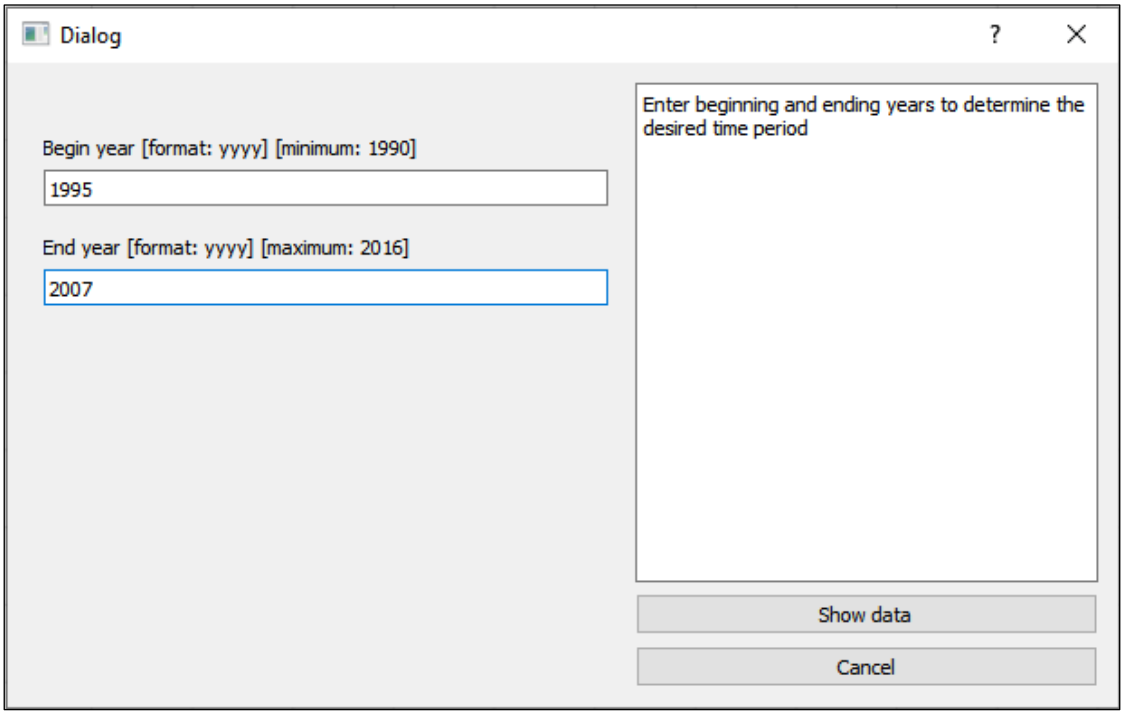

Image 7: The Year-to-Year window

Next up is the date-to-date button which, when pressed, opens a small modal window to which the user can select from a visual calendar the start- and end-dates as long as they are in certain limits provided by the databases. An example of this window can be seen in image 6 and the Year-to-Year window can be seen in image 7. After the user has selected the time span, they desire they can see their selection in the main window text view widget.

On the top of the main view there is the main navigation bar. There we put other functionalities such as choosing a specific loadout or changing default settings. On the navigation bar, seen in image 4, there is also an option to open a new window for data comparison. This action opens a carbon copy of the main view. This copy works the same as the previous window and can display all the same data the first window could. These views can show different data and actual comparison between views is done manually by the user.

The actions in the navigation bar are there to save space for the actual data and graphs. We chose to do this because we deem it important that only settings that can immediately affect the view are easy to access and are visible at all times compared to settings which the user may or may not always need.

6 EVALUATION OF DECISIONS AND PROCESSES

For the mid-submission, we designed the pipeline of STATFI data from the online database all the way to the graph quite carefully. All of this was worked out in a design session right after the prototype submission. All the way until the mid-term submission, we were able to follow the design very well. At the time of the mid-submission, we felt that the STATFI data pipeline was efficient and easy to follow and update. One change that was made to the initial design, was that for practical reasons we decided to implement the actual updating of the graph in the view instead of the controller. This way we eliminated one dependency, as the controller didn't need to be aware of the view.

For the mid-submission we focused so much on STATFI, that we somewhat neglected the integration of other databases. Since the beginning, there were ideas of implementing STATFI and SMEAR data related functions in separate classes that would inherit a common base class, perhaps following the strategy design pattern, but none of that had been implemented in practice. Some refactoring had to be done after the mid-submission to achieve the program structure we wanted. Thankfully all of this went quite smoothly.

With SMEAR data, we ended up only supporting three measuring stations (Värriö, Hyytiälä, and Kumpula) and three greenhouse gases (CO2, SO2, and NOx). This was due to the table/variable names for each gas in each station being completely arbitrary. Adding support for more options wouldn't have been difficult, but it would have been meaningless busywork. Figuring out what gases are available for which time period for each station was also incredibly tedious. We decided to stick with these basic options as implementing more of them wouldn't have proven us to be more skilled or knowledge. It simply would've proven that we have too much free time on our hands.

With the experience accumulated during the implementation of the STATFI data pipeline, fetching the SMEAR data, parsing it, and updating the graph with it was not difficult either. The difficulties with SMEAR data, as mentioned before, had more to do with making decisions about which stations and gases to support. Additionally, we wanted to create a calendar widget that the user can conveniently choose start and end dates from. The biggest problem with the calendar was guiding the user by indicating the availability of data on specific days.

The main quality requirement we were concerned with was performance. With STATFI data we decided to fetch all of it at program launch, as then it would be least noticeable. As there's much more of SMEAR data, we couldn't do the same with SMEAR while keeping a solid performance. SMEAR data is also acquired in such amounts at a time that strong performance and stability are maintained. In other words, we utilized eager acquisition with STATFI data and lazy acquisition with SMEAR data.

The processes we used throughout the project worked quite well. We used a Trello board to organize tasks and keep track of responsibilities and deadlines. We had weekly design meetings that were fruitful but could drag on for up to 3 hours. Perhaps agendas and schedules should have been stricter and more well defined. In hindsight, coding conventions and Git usage are some things we should have discussed more in the beginning. If you look at our code, you will find inconsistent naming in variables, functions, and classes.

Everyone should also have merged with main much more often. All of us worked in our own branches for far too long and ended up running into conflicts and a lot of confusion, especially towards the end of the project. Deciding on Git usage guidelines and enforcing them would've been very beneficial. Making these mistakes has taught us a lot and we are now much better equipped for the future.

REFERENCES

1. SMEAR database and API. https://smear-backend.rahtiapp.fi/swagger-ui/ (Viewed 16.2.2022)
2. STATFI database and API. https://pxnet2.stat.fi/PXWeb/pxweb/en/ymp/ymp__taulukot/Kokodata.px/ (Viewed 16.2.2022)
3. Qt. https://www.qt.io/ (Viewed 16.2.2022)