Design document

This document describes the software design project work of the Panssarinyrkki group. The purpose of the program is to implement an application for environment infrastructure monitoring and visualizing. Initially weather data from Finnish Meteorological Institute and electricity data from Fingrid are used as the main data providers.

The program is developed using a C++ framework Qt that provides all necessary features and libraries for building the entire application. Mainly the program consists of two parts: the C++ backend and QML frontend. This design document primarily focuses on the C++ backend since there is almost direct connection from the view module to the QML frontend. Also, all data processing and user interaction is handled inside the C++ part of the application.

High-level description

The high-level description contains module level description of the implementation. It is mainly intended for understanding the rough operating principles and relationships between the different components. There are some utility classes and interfaces that are not listed here but they are not important for understanding the basic structure of the program. More detailed description about the interfaces and components can be found later in this document.

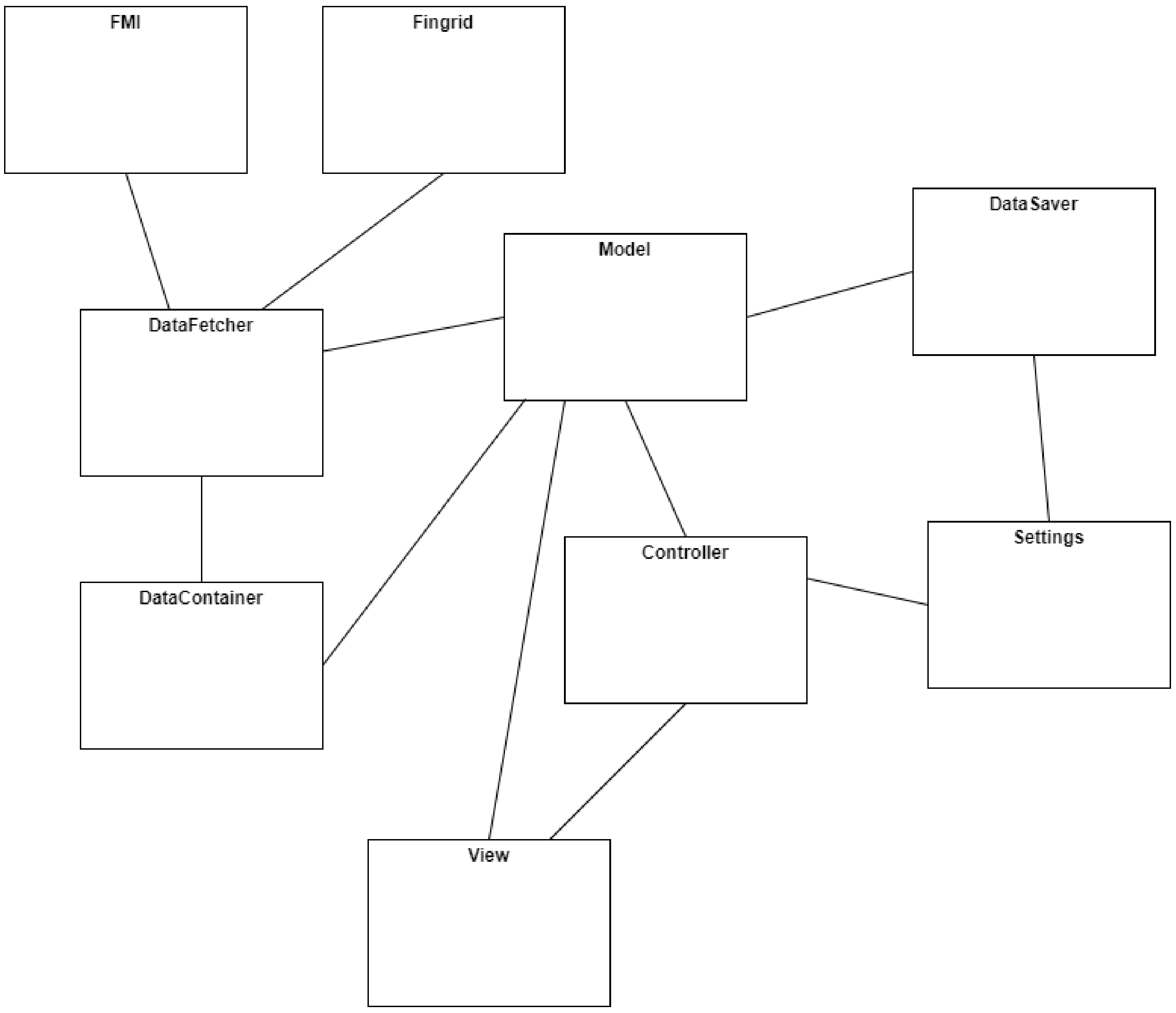

Figure 1 High level diagram of the program.

The Figure 1 shows the high-level architecture of the program. In practice, there are the main Model, View and Controller modules in the core of the application. Also, different internet APIs that produce the data handled in the application have their own modules. In addition, there are few utility modules that help other classes to implement the required functionality. Utility classes in this case are the DataSaver, Settings, and DataFetcher that for example handle saving the application settings and data between different sessions.

Internal interfaces

According to the general MVC architecture the most important interfaces are between the Model, View and Controller modules/classes. Every one of them has an interface that they implement, and other classes depend on that. The Controller reacts to the user actions and controls the Model, which in turn will notify the View for changed content.

Another important internal interface is between the internet data source classes and the data fetcher. All data sources implement an interface which the data fetcher will use to interact with the actual implementation. This way there are not any direct dependencies to specific data sources (in the ideal case).

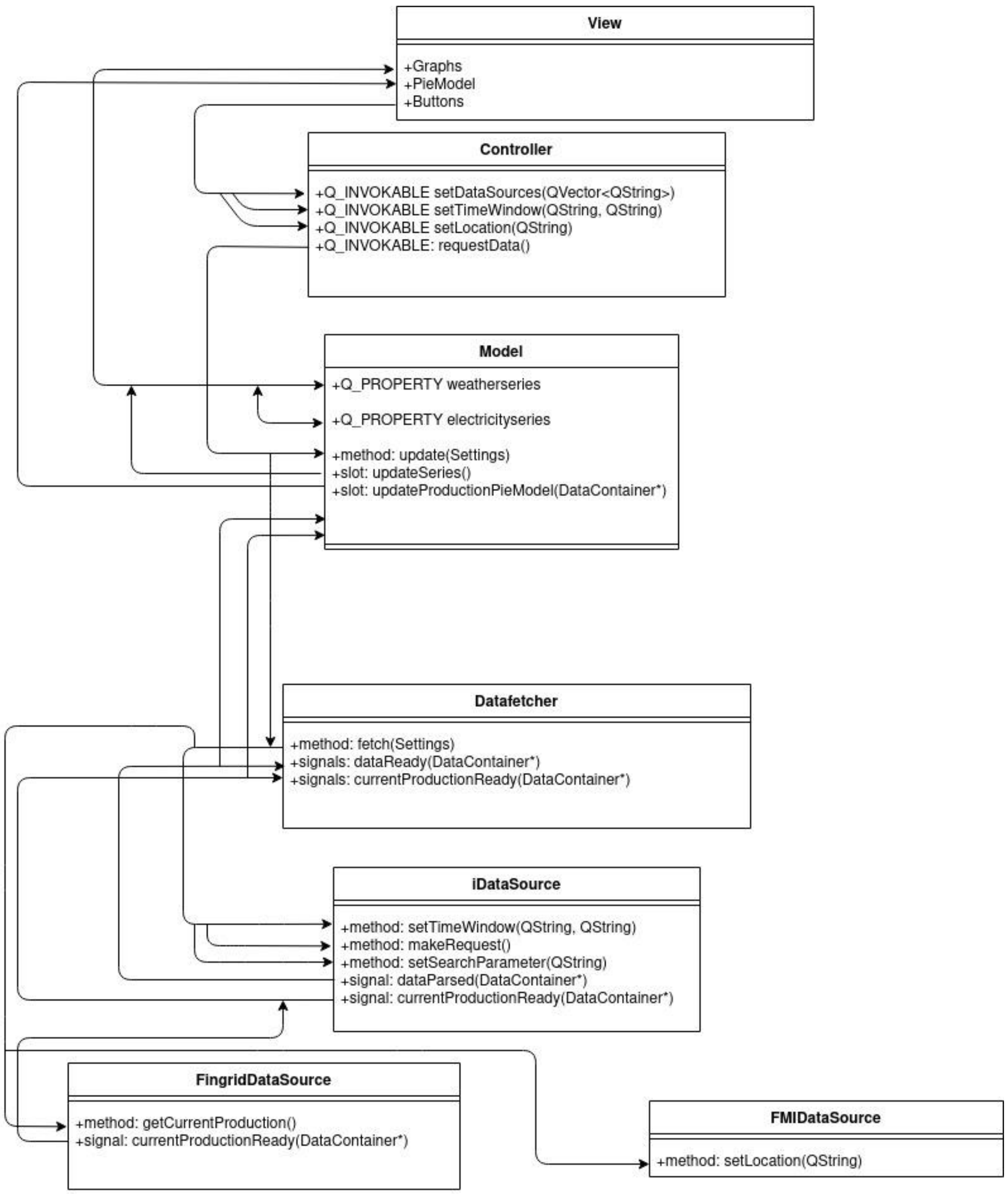
View
+Graphs
+PieModel
+Buttons
Controller
+Q_INVOKABLE setDataSources(QVector<QString>)
+Q_INVOKABLE setTimeWindow(QString, QString)
+Q_INVOKABLE setLocation(QString)
+Q_INVOKABLE: requestData()
Model
+Q_PROPERTY weatherseries
+Q_PROPERTY electricityseries
+method: update(Settings)
+slot: updateSeries()
+slot: updateProductionPieModel(DataContainer*)
Datafetcher
+method: fetch(Settings)
+signals: dataReady(DataContainer*)
+signals: currentProductionReady(DataContainer*)
iDataSource
+method: setTimeWindow(QString, QString)
+method: makeRequest()
+method: setSearchParameter(QString)
+signal: dataParsed(DataContainer*)
+signal: currentProductionReady(DataContainer*)
FingridDataSource
+method: getCurrentProduction()
+signal: currentProductionReady(DataContainer*)
FMIDataSource
+method: setLocation(QString)

Figure 2 General interfaces related to the dataflow.

Figure 2 describes the general data flow between different classes of the program. It does not show all of the interfaces and the helper classes that do not directly contribute to the data flow. The different interfaces in the view component that are visible to the user are connected to different interfaces in the model and controller component which in turn react to the user's input and either call utility classes' methods or redirect the call to other components in the program. The controller class does not directly contribute to the data flow in the program structure, but rather orchestrates the program flow.

The most important connection is the connection between the model and dataFetcher class, as it connects the different network APIs to the inner data structures of the program. The dataFetcher will make the correct calls to correct API based on currently configured settings that are implemented as a class. Some of the methods are shared between the APIs by iDataSource class that acts as an interface class for all network APIs.

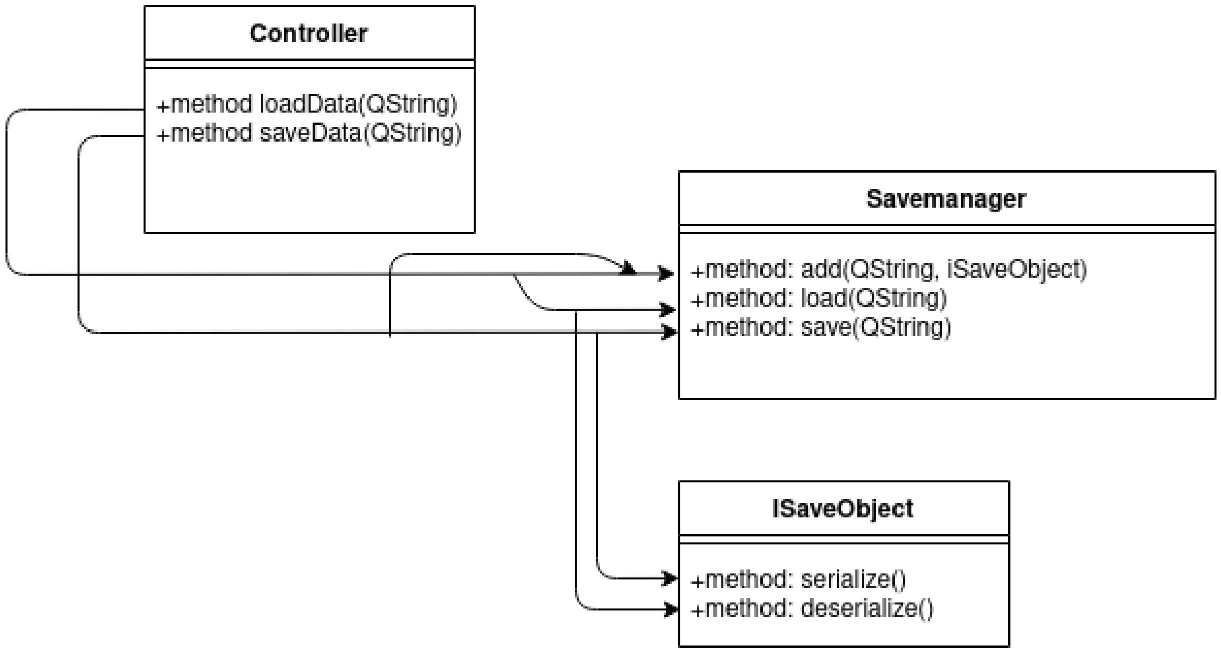

Figure 3 Interfaces related to saving and loading.

Figure 3 describes the interfaces related to saving and loading the programs settings and data. The saving is initiated in the Controller, and it adds all ISaveObects to SaveManager using the add method. Calling the save method in SaveManager causes it to serialize all components that were previously added. Serialized components are then saved to the specified file in JSON format. Loading the components is performed similarly in reversed order. The SaveManager can save and load all classes that are derived from the ISaveObject.

In general, most of the components operate and communicate with each other in the interface level and are not dependent on the actual implementation. This gives flexibility and extendibility to the overall design. It also makes sure that good programming conventions (e.g., SOLID) are followed throughout the application implementation.

Component descriptions

All different components and their functionalities are listed here. Also, the most important functions from each component are listed below the general description.

- View: Consists of the QML interface. User directly interacts with this component. User interacts with buttons and adjusts settings displayed in View. View informs to user about error if one is occurred. Interactions with view is delegated to controller which handles errors and fetching data. Data from the model is directly mapped to the View. User can choose time window which kind of information he wants to see about electricity or weather. User can enable automatic updates when graphs are updated in every 10 seconds. User can choose if he/she wants to see raw data or daily average, minimum or maximum. User can save and load data with file name.
    - Since the view is implemented in QML there really are not any important functions that are called by other components. Instead, the mapping and handling are defined in other components (Model and Controller) with Qt specific constructs.
- Model: The main high-level component for data storage and fetching. The behavior is controlled by controller. Changes in Model are showed in View which reacts to changes in Model. Model keeps track of all weather data displayed and updates data according to the settings user has selected.
    - updateSeries(DataContainer*): Updates line series based on the received data.
    - updatePieModel(DataContainer*): Updates the pie chart based on the received data.
    - update(Settings): Initiates the model to update itself.
- Controller: Receives the user interaction and controls the application based on that. The actual implementation of these features is handled elsewhere. Controller passes the selected settings to module when new data request is executed. Controller also checks that settings are valid, and calls save and load functions from SaveManager.
    - requestData(): Initiates a new data request.
    - setLocation(QString): Sets location for data request.
    - setDataSources(QVector<QString>): Sets the type of data to be fetched.
    - setTimeWindow(QString, QString): Sets start and end time of data request.
    - setAutomaticUpdate(bool): Sets automatic update status.
    - loadData(QString): Loads the program settings and data from a file.
    - saveData(QString, QString): Save the program settings and data to a file.
    - setDataProcessing(QString): Sets type of data calculations e.g., avg, min and max.
- SaveManager: Handles saving different entities of data. Settings and data are saved between different sessions using this component. Save file function takes file name and used settings as a parameter and writes a file as json according to these parameters. Load function takes file name as parameter and loads file according to name given. If loading file fails i.e., there is no file with given name, load function returns false.
    - add(QString, ISaveObject*): Adds new object to save manager.
    - load(QString): Loads file by given name.

    - save(QString): Saves file by given name.

- Settings: Contains the currently selected settings of the program. This component is mainly only a container for data. Settings contains all selections which are required to make requests for APIs. Controller sets the data here and DataFetcher uses them for requests.
    - getLocation(): Returns the stored location.
    - setLocation(QString): Sets the location.
    - getDataSources(): Returns list of stored data sources.
    - setDataSources(QVector<QString>): Sets all sources in input vector.
    - setTimeWindow(QString, QString): Sets time window by input limits.
    - getStartTime(): Return time window start time.
    - getEndTime(): Return time window end time.
    - setDataProcessing(QString): Sets data processing method by input string.
    - getDataProcessing(): Returns stored data processing method.
    - getParams(): returns all parameters except data sources.
- DataContainer: This component contains the business data of the program. All weather and electricity data are stored in objects of DataContainer. DataContainer holds time series of data and DataFetcher informs Model when Data is ready and passes DataContainer to Model which updates graphs and fields to view.
    - setType(QString): Sets type of data e.g., Hydro power, wind power etc.
    - setUnit(QString): Sets unit of data contained.
    - setCategory(QString) Sets category of data e.g., electricity or weather.
    - addElement(QPointF): Adds element where x value is date as UNIX time and y is value of data.
    - getType(): Returns type of data.
    - getUnit(): Returns unit of data.
    - getCategory(): Returns category of data.
    - getElement(int): Return element of given index.
    - getData(): Returns whole data contained.
    - getLimits(): Returns maximum and minimum value of data contained.
    - size(): Returns size of data.
- DataFetcher: The data fetcher handles selection of different APIs. It uses Settings to choose correct API where data is fetched and informs FMI and Fingrid about details of request. DataFetcher informs Model when data is ready and passes data to Model. DataFetcher also calculates average, minimum and maximum values from data from one day. User can choose from view which kind of data he wants to see. User can also choose to view raw data when used data is not manipulated at any way.

    - fetch(Settings&): Fetch data according to settings.
    - setDataProcessingMethod(QString): Sets method of processing fetched data e.g., average, minimum etc.

- FMI: Parsing of the Finnish Meteorological Institute. Data regarding weather information is fetched and parsed here. Makes requests to FMI API according to the details from DataFetcher. Parses data to DataContainer, informs DataFetcher when data is ready and passes it to DataFetcher.
    - setTimeWindow(QString, QString): Sets time window for fetch.
    - makeRequest(): Sends request to FMI API.
    - setSearchParameter(QString): Sets parameter for request.
    - setLocation(QString): set location of weather data.
- Fingrid: Parsing of the Fingrid API. Data regarding electricity production and consumption is fetched and parsed here. Makes requests to Fingrid API according to the details from DataFetcher. Parses data to DataContainer, informs DataFetcher when data is ready and passes it to DataFetcher.
    - setTimeWindow(QString, QString): Sets time window for fetch.
    - makeRequest(): Sends request to Fingrid API.
    - setSearchParameter(QString): Sets parameter for request.
    - getCurrentProduction(): Fetches data for current production of electricity.

Design choices

Main drivers for these design choices come from common design patters that aim to make software maintainable and easy to develop. For example, the separation of Model, View and Controller follows the common and well tested MVC design pattern. Also, aspects of the SOLID principles are considered in this design document and in the implementation. All components in this software are designed to operate at high abstraction level and to be general as possible. This reduces the dependencies between components and keep the modules as reusable as possible. Also adding new data sources for example will remain fairly straight forward.

For the technology stack we decided on the C++ framework Qt simply because every one of us already was familiar with it. The Qt also has some useful utility classes that we needed in our implementation e.g., XML parsers and such. This helped a lot during the implementation since every single algorithm does not need to be coded from the scratch. We also used the QML design language which helped to keep the UI separate from the implementation.

We made quite a few conscious implementation choices because the specification was quite loose (apparently this was also the intention). For example, our implementation combines single energy-related time series, single weather-related time series and single saved time series, all in a single view. In these series observations and forecasts can be freely combined based on the selected time if they are available from the used APIs.

Self-evaluation

This design document has helped to better understand and keep track the overall structure of the program. With that information it is easier to implement the features and define dependencies in the code itself. During the implementation it has been very useful to use this document as a reference when writing the code itself. Our original design was largely appropriate, with the only significant change being the combination of weather and energy production into one view after receiving clarification on the specification of the program.

There are few changes that we have made to the overall design of the program that we already did for the mid-term submission. The first obvious change was to the data collecting component that was removed due to the changed requirements. With the new requirements specification, the data collecting feature is replaced with large-scale data handling requirement. The large-scale data handling does not need any additional components, but instead additional attention to the efficiency needs to be paid. The second change to the structure of the program was few additional utility components. For example, data fetcher component was added to the design to help correctly formatting the API endpoints.

Now that the program is finished, we can see that our initial plans, especially after the mid-term submission were sufficient for the most parts. Generally, we were able to just follow the design and implement all the required functionalities around that. There was not really a need for any significant changes to the architecture and we feel that the end result is quite close to the original plan we had in the beginning.

All in all, we feel that we have succeeded in the learning objectives of the course and implemented a software that matches the specification. We have also familiarized ourselves with some common design patterns and implemented them in an actual large scale software program that does something useful. We also managed to implement one bonus requirement on top of the main requirements.