Unnamed

Design Document

COMP.SE.110-2020-2021-1 Software Design

Joonas Pelttari
Kalle-Henrik Raitanen
Jani Uolamo
Sipi Ylä-Nojonen
21.3.2021

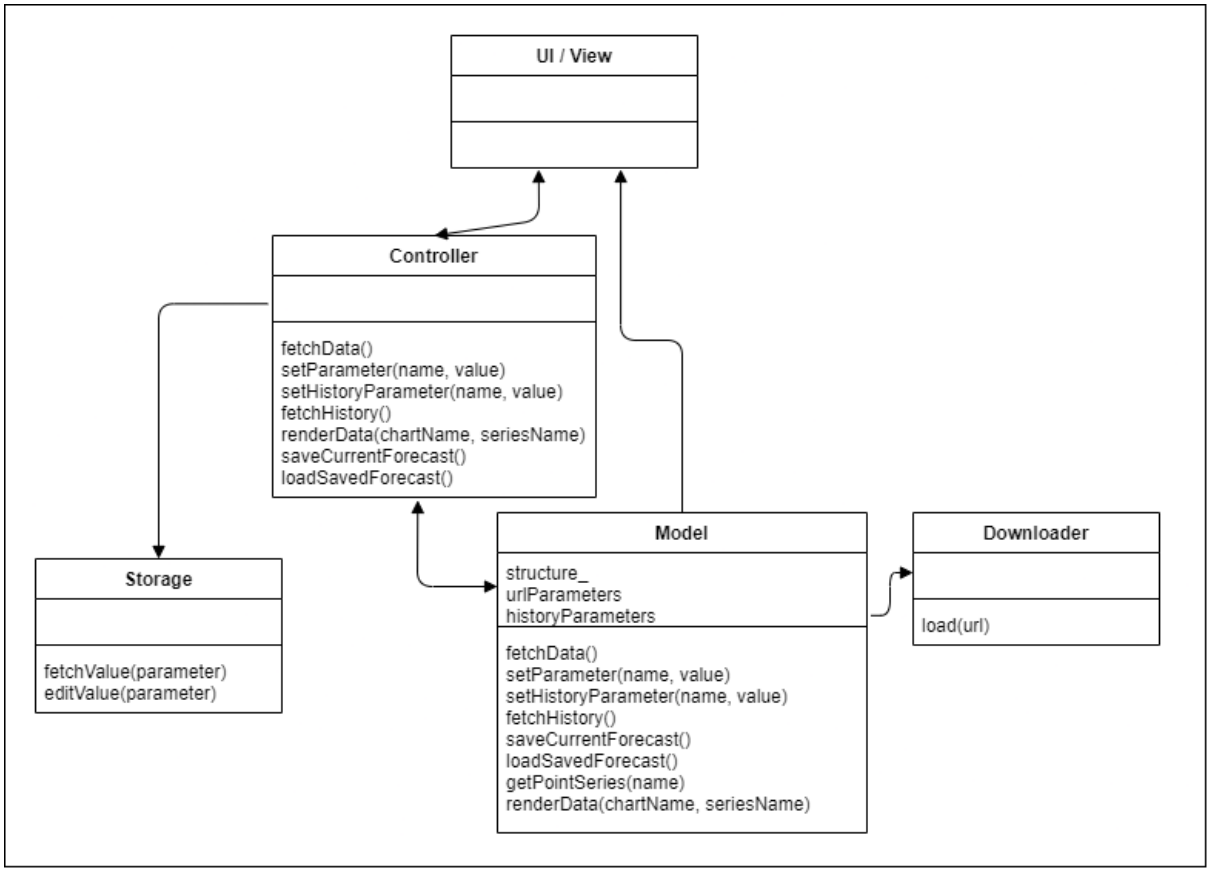
High level description

The applications GUI is implemented with QtQuick using QML. Other classes for application include Model, Controller and DownLoader which interfaces are further discussed in “Boundaries and Interfaces” section.

QtCharts is included as additional Qt-library to easily present data visualizations in a user-friendly way.

UI contains three views that separate displayed data into categories Weather/Power consumption/ Power production. Each view contains several graphs for separate predictions from each category (temperature, wind speed, visibility / total production, water production, wind production etc.) These can be individually shown or hidden based on user selection. In addition to this under the graphs are shown current values for weather and current total energy production + consumption.

Class diagram



Boundaries and Interfaces

Controller

Controller class is responsible for processing and delegating user requests from the user interface whereas Model stores downloaded data and updates user view according to orders from controller.

- fetchData(), calls for Model’s interface to initiate data download,
- setParameter(name, value), calls for Model to set single query url parameter (eg. City) by name, **Added since the prototype plan**

- setHistoryParameter(name, value), Call for model to set starting or ending time for long-scale history fetch, **Added since the prototype plan**
- fetchHistory(), Call for Model to fetch long-time history based on saved starting and ending point, **Added since the prototype plan**
- renderData(chartName, seriesName), requests Model for rendition of changed data in view.
- saveCurrentForecast(), calls for Model to save currently displayed forecast data for later use, **Added since the prototype plan**
- loadSavedForecast(), loads saved forecast data based on currently selected timeline if found, **Added since the prototype plan**

Model

Model class implements storaging of downloaded data as well as interface for controller to send user requests that are then displayed into the GUI through view. Model itself is mainly responsible for updating the shown data based on eg. City, but all categories of data are fetched every time. The selection between actual shown graphs is done in view in QML side.

- urlParameters<name, value>, map that contains parameters for constructing query based on user preference, (City (weather specific), starting time, ending time, scale (energy specific)), **Added since the prototype plan**
- historyParameters<name, value>, parameters for constructing long-history fetch for energy by starting and ending time, **Added since the prototype plan**
- structure_<time, map<name, value>>, Contains fetched points for each time point separated into a map by parameter name

- fetchData(), constructs url from parameters and calls for DownLoader to download data
- setParameter(name, value), Sets single query url parameters (eg City) by user setting, **Added since the prototype plan**
- setHistoryParameter(name, value), Set starting or ending time for long-scale history fetch, **Added since the prototype plan**
- fetchHistory(), initialize data collection for large-scale history fetch, **Added since the prototype plan**
- saveCurrentForecast(), saves currently displayed forecast data for later use, **Added since the prototype plan**
- loadSavedForecast(), loads saved forecast data based on currently selected timeline if found, **Added since the prototype plan**
- getPointSeries(name), returns data points respective to series defined by name eg. temperature, power production
- renderData(chartName, seriesName), pushes request for View to update data in UI.

DownLoader

DownLoader class is used to handle connections and fetching data from internet sources to be stored by Model.

- load(url), downloads data from url source

View (Changed from separate class to integrated in QML)

View is part of applications QML implementation and consists of slots that connect straight to signals caused by user from UI elements (buttons and checkboxes etc.). This is then used to

define what graphs in each category is shown to user and how views are scaled. **This was changed from separate C++ class to integrated, since it seemed separate class would become more complicated and QML has been designed to well support managing changes in UI.**

- Integrated to QML structure
- ChartView.onSeriesAdded(), for each AbstractSeries shown adjust UI graph axis to show correct scale
- CheckBox.onCheckedChanged(), for each checkbox control change visibility of single AbstractSeries linked to said checkbox.
- Slider.onValueChanged(), change graph scaling between day/week scale in the view based on UI slider position

Storage

Storage class is used for storing user preferences regarding automatic datacollection etc. It provides interface for Controller to save single user preference parameters from UI as well fetch these values later-on. However, UI-side implementation only supports saving all parameters at once, so all settings will be overwritten each time preferences are saved.

- fetchValue(parameter), get currently saved value for setting
- editValue(parameter, newValue), modify saved setting

Self-evaluation

Some things in original design have already been changed, for example the View class was originally only implemented as part of QML implementation but was planned to be changed to separate C++ class, however this idea was abandoned, and view implementation remains as it was: as part of QML side.

In general, the current plan is not too different from the original and mostly aims at expanding the original and defining plans more clearly. On the other hand, later implementation may need changes in some aspect such as how different things are shown to user in GUI and how user can interact with controls (for example how different graphs actually lay out on top of each other in practise / is the current categorization sensible). Changes to this might be required for the better user experience in fetching and displaying the data user wants.

On the other hand, many things have been implemented as they were planned in prototype phase and for example structure of separation into classes planned classes and responsibilities of these classes mostly remain unchanged (View class being exception).