**PROJECT NORDICWATT**

**DESIGN DOCUMENT**

COMP.SE.110 Software Desing – Group Assignment

Antti Hakkarainen  
Markus Hissa  
Heikki Hohtari  
Janne Taskinen

Table of Contents

[2 Introduction 3](#_Toc149241202)

[3 Requirements 4](#_Toc149241203)

[3.1 Functional requirements 4](#_Toc149241204)

[3.2 Non-Functional requirements 5](#_Toc149241205)

[3.3 Chart Types 5](#_Toc149241206)

[3.3.1 Line diagram 5](#_Toc149241207)

[3.3.2 Area Chart 6](#_Toc149241208)

[3.3.3 Scatter Dot Chart 6](#_Toc149241209)

[3.3.4 correlation heatmap (ei ole ainakaan protossa vielä toteutettuna) 6](#_Toc149241210)

[3.3.5 Other charts (extra) 7](#_Toc149241211)

[3.4 Available datatypes per chart 7](#_Toc149241212)

[3.5 Example UI 8](#_Toc149241213)

[4 API’s 9](#_Toc149241214)

[4.1 Fingrid 9](#_Toc149241215)

[4.2 Finnish Meteorogical Institute 9](#_Toc149241216)

[4.3 porssisahko.net (extra) 9](#_Toc149241217)

[5 Planned Architecture and design 9](#_Toc149241218)

[5.1 Activity diagrams of high-level use cases 9](#_Toc149241219)

[5.1.1 Program start-up 9](#_Toc149241220)

[5.1.2 Validation of user inputs 11](#_Toc149241221)

[5.1.3 Diagram creation process 12](#_Toc149241222)

[5.2 Diagram of components within the system 13](#_Toc149241223)

[5.3 Design patterns and decisions 13](#_Toc149241224)

[5.4 User interface design choices 14](#_Toc149241225)

[6 Realized architecture and design 15](#_Toc149241226)

[6.1 Application overview 15](#_Toc149241227)

[6.2 Class diagram 19](#_Toc149241228)

[6.3 Changes compared to original design 20](#_Toc149241229)

[6.3.1 Decision to not include database 20](#_Toc149241230)

[6.3.2 Decision to not implement API call prioritization and data pre-caching to support Pörssisähkö API properly 21](#_Toc149241231)

[6.3.3 Introduction of session controller and extending tab manager 21](#_Toc149241232)

[6.4 Design patterns and solutions used 21](#_Toc149241233)

[7 Attachments 22](#_Toc149241234)

[8 References 22](#_Toc149241235)

# Introduction

Our application serves as a comprehensive weather and energy tracking platform for desktop environments. The software aggregates information from a minimum of two distinct APIs, displaying the data in a user-friendly format suitable for analysis.

Specifically, users have the capability to review historical as well as forecasted weather data for regions in Finland. The weather metrics provided include temperature, wind speed, and rainfall. Additionally, the application monitors real-time and forecasted energy production and consumption rates within Finland.

As a future enhancement, if time permits, we also plan to integrate Finnish electricity stock market prices into the application.

The concept for this application originated from ChatGPT and was chosen from a variety of options as the most compelling and feasible for production. Alternative possibilities were either less intriguing or necessitated the use of paid APIs or non-public data.

We intend to develop this desktop application as a Java Maven project, utilizing the JavaFX library for its graphical interface. Additional libraries will be specified as they are identified and incorporated into the project.

# Requirements

This chapter consists of functional and non-functional requirements for the application. The requirements are prioritized using criticality metric. Criticality value 1 means highest priority, and criticality value 5 means lowest priority. The requirements have been formed before the actual implementation work has been started. It is possible (and likely) that not all the requirements will be fulfilled in the final version of the application. It is also possible that during the implementation phase some priorities are changed and new requirements are introduced. Documentation regarding the changes to requirements in later sections of this document.

Since the main user-facing functionality of the application is displaying different charts to the user based on users’ inputs, more detailed planning regarding the charts has been introduced in this section.

## Functional requirements

Users workflow of the program follows structure:

1. User types chooses the document style (for example line diagram)
2. User chooses values x and y parameter datatypes (for instance temperature and energy consumption)
3. User selects desired date range
4. User presses Create Diagram to create new diagram

|  |  |  |
| --- | --- | --- |
| **ID** | **Description** | **Criticality** |
| **F01** | Save required API keys (Fingrids) | **1** |
| **F02** | Fetch specified data from Fingrid | **1** |
| **F03** | Program fetches specified data from Finnish Meteorological institute | **1** |
| **F04** | Choose between line, area and scatter dot charts | **1** |
| **F07** | Generate chart based on search criteria | **1** |
| **F15** | Save search terms of their query for later use | **1** |
| **F16** | Apply saved search terms (check F15) | **1** |
| **F19** | Adjust the parameters of the visualization | **1** |
| **F21** | Save and load chart generation settings | **1** |
| **F05** | Change the range of the data | **2** |
| **F06** | Select predefined datatypes for x and y axis | **2** |
| **F20** | Create multiple diagrams into same view (max 4) | **2** |
| **F13** | Close a tab | **3** |
| **F17** | Apply time as either specific timerange or “Last n days/weeks/months” | **3** |
| **F08** | See detailed information of the datatype selected (unit, description and source) | **4** |
| **F12** | Generate either chart in new tab or the active tab | **4** |
| **F18** | Fetch specified data from porssisahko.net | **4** |
| **F22** | Drag & drop diagrams between tabs | **4** |
| **F23** | Change diagram's position inside a tab with drag & drop | **4** |
| **F26** | Switch between 1 or 1..4 chart view as an option | **4** |
| **F26** | Display placeholder charts for non-generated charts in 4 chart mode | **4** |
| **F27** | Reload previously open tabs and generated charts | **4** |
| **F08** | Hover over chart to see individual datapoints and their values | **5** |
| **F09** | View the datasets average Y axis value | **5** |
| **F10** | Choose to view average horizontal line on the chart on Y axis | **5** |
| **F11** | Choose to view two horizontal lines on upper and lower quartile on Y axis | **5** |
| **F14** | Export data visualized as image or pdf | **5** |
| **F24** | Pie chart as a chart option | **5** |
| **F25** | Correlation heatmap as chart option | **5** |
|  |  |  |
|  |  |  |

## Non-Functional requirements

|  |  |  |
| --- | --- | --- |
| **ID** | **Description** | **Criticality** |
| **N3** | API fetch should always result in success (provided service is not down). User should not be able to put search terms that cause invalid or erroneous API call. | **1** |
| **N7** | The design must be such that further data sources or additional data from existing sources could be easily added. | **1** |
| **N1** | Program should keep the data once fetched so we don’t have to fetch it again | **2** |
| **N2** | Data and user defined settings should be stored over shutdown | **2** |
| **N4** | Program must implement error handling class | **2** |
| **N5** | Fetching of data should not take over 30 seconds | **2** |
| **N6** | User can see animation during data fetching and parsing (to see that something is happening) | **4** |

## Chart Types

### Line diagram

In a standard line diagram, the X and Y axes represent chosen data types, such as temperature and energy consumption. Optionally, and depending on development time, a second Y-axis parameter can be added to represent a third variable within the same line diagram.

It's important to note the role of the date range in these diagrams:

1. If the date range is set on the X-axis, it serves to indicate the minimum and maximum values for the X-axis variable.
2. However, if a variable other than the date range is placed on the X-axis, the chosen date range will instead define the time interval for which data is collected and displayed.

The date range's function can either limit the X-axis values or serve as a filter for the dataset, and these are two distinct uses.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Data unit | **Time** | **Consumption** | **Price** | **Temperature** | **Wind** |
| **Time** | I | C | C | C | C |
| **Consumption** | C | I | P | P | P |
| **Price** | C | P | I | P | P |
| **Temperature** | C | P | P | I | P |
| **Wind** | C | P | P | P | I |

Table 1. Example of data combinations on line diagram.

|  |  |  |
| --- | --- | --- |
| Sign | **Compatibility** | **Notes** |
| C | Compatible | No problems to display on basic XY-chart |
| P | Partially compatible | Might need different y-axes due to the difference in scale. |
| I | Incompatible | No sense to combine in same chart. |

Table 2. Legend for table 1.

### Area Chart

Area chart is basically the same as line chart, but it will color the area below the line.

### Scatter Dot Chart

On scatter dot chart we can visualize individual datapoints on x y axis. Time will not be available on neither axis, but the data is searched by specific date range. The dots on if there multiple dots on top of each other we can have two options to visualize that there are in fact multiple dots at that location:

* Make the dot bigger, if there are multiple dots or
* put a little variance to throw off the dot from its actual point

### correlation heatmap (ei ole ainakaan protossa vielä toteutettuna)

TODO: Tää kappale vaatii aika paljojn tarkennusta

https://duckduckgo.com/?q=correlation+heatmap&t=vivaldi&iar=images&iax=images&ia=images

The cells contain the correlation coefficient between the two variables represented by the corresponding row and column. This coefficient ranges from -1 to 1.

* 1 indicates a perfect positive correlation
* -1 indicates a perfect negative correlation.
* 0 indicates no correlation.

We could use Pearson correlation coefficient for example: <https://en.wikipedia.org/wiki/Pearson_correlation_coefficient>

Example scenarios:

* Cell at the intersection of "Electricity Price" and "Temperature" contains the correlation coefficient indicating how electricity price relates to temperature.
* Cell at the intersection of "Wind Amount" and "Electricity Consumption" contains the correlation coefficient indicating how the amount of wind correlates with electricity consumption.

Colors:

For example Blue to Red Spectrum: Blue can represent negative correlation, white (or a neutral color like gray) can represent no correlation, and red can represent positive correlation.

### Other charts (extra)

While the initial Minimum Viable Product (MVP) will focus on core functionalities, there are several other chart types that could enhance the application's utility and user experience in future releases.

* radar chart
* Pie chart (how much does wather variable affect price in percentages).
* Price guesstimation
* Caching

## Available datatypes per chart

For line chart and area chart options data options available are as follows:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | A black and white image of a cross  Description automatically generated |  | | **Time** | **Electricity Consumption** | **Electricity Production** | **Hydro Power** | **Nuclear Power** | **Wind Power** | **Temperature** | **Wind** | **Rain** | **Humidity** | **Air Pressure** | **Electricity Stock Price** |
| **Time** |  | yes | yes | yes | yes | yes | yes | yes | yes | yes | yes | yes |
| **Electricity Consumption** |  |  | yes | yes | yes | yes | yes | yes | yes | yes | yes | yes |
| **Electricity Production** |  | yes |  | yes | yes | yes | yes | yes | yes | yes | yes | yes |
| **Hydro Power** |  | yes | yes |  | yes | yes | yes | yes | yes | yes | yes | yes |
| **WNuclear Power** |  | yes | yes | yes |  | yes | yes | yes | yes | yes | yes | yes |
| **Wind Power** |  | yes | yes | yes | yes |  | yes | yes | yes | yes | yes | yes |
| **Temperature** |  | yes | yes | yes | yes | yes |  | yes | yes | yes | yes | yes |
| **Wind** |  | yes | yes | yes | yes | yes | yes |  | yes | yes | yes | yes |
| **Rain** |  | yes | yes | yes | yes | yes | yes | yes |  | yes | yes | yes |
| **Humidity** |  | yes | yes | yes | yes | yes | yes | yes | yes |  | yes | yes |
| **Air Pressure** |  | yes | yes | yes | yes | yes | yes | yes | yes | yes |  | yes |
| **Electricity Stock Price** |  | yes | yes | yes | yes | yes | yes | yes | yes | yes | yes |  |

Chart 1. Line and area chart options.

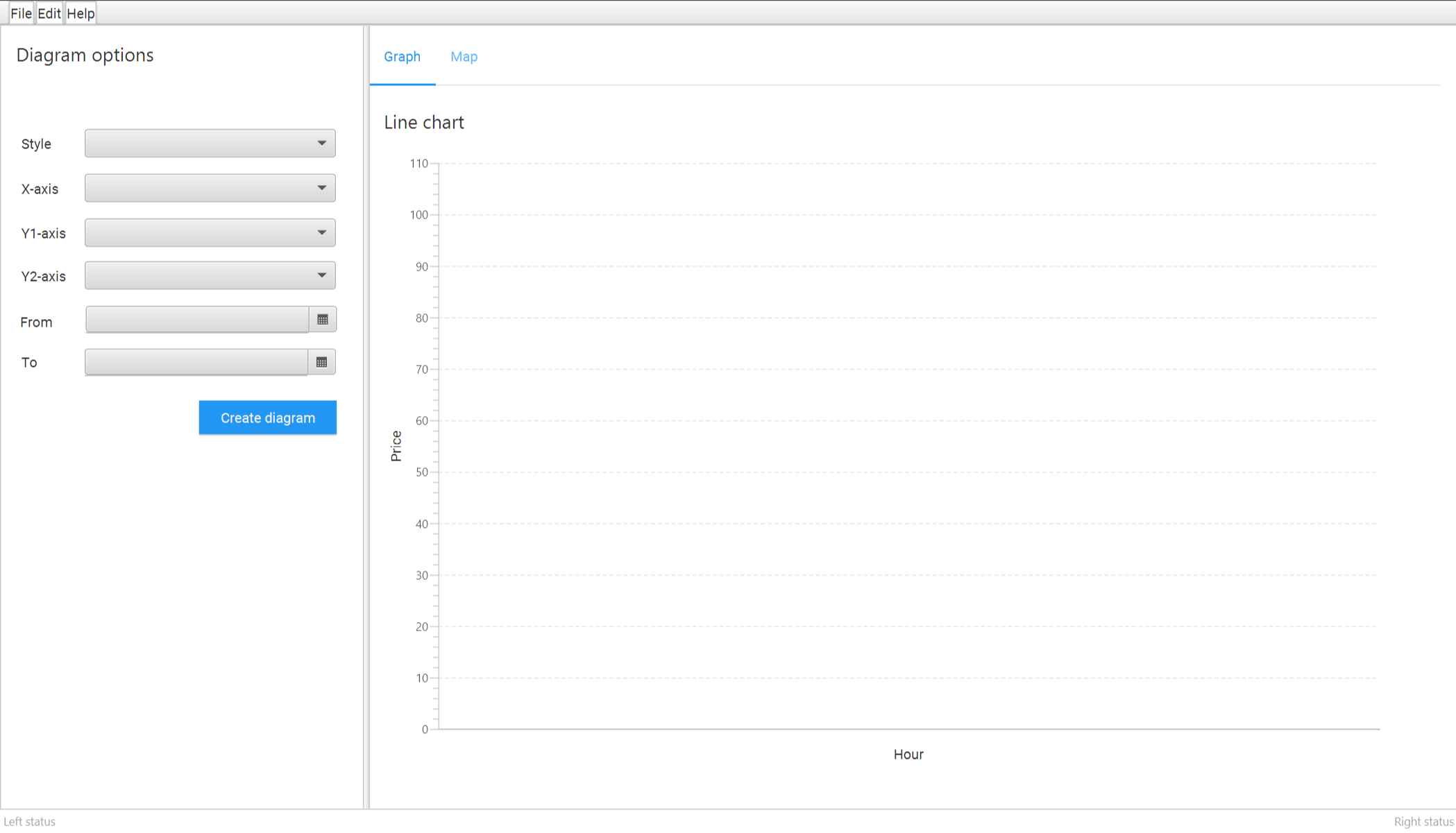
For the scatter dot chart, we also have the same options, but time cannot be selected. Therefore, in scatter chart date range always is only for data population selection.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | A black and white image of a cross  Description automatically generated |  | | **Electricity Consumption** | **Electricity Production** | **Hydro Power** | **Nuclear Power** | **Wind Power** | **Temperature** | **Wind** | **Rain** | **Humidity** | **Air Pressure** | **Electricity Stock Price** |
| **Electricity Consumption** |  | yes | yes | yes | yes | yes | yes | yes | yes | yes | yes |
| **Electricity Production** | yes |  | yes | yes | yes | yes | yes | yes | yes | yes | yes |
| **Hydro Power** | yes | yes |  | yes | yes | yes | yes | yes | yes | yes | yes |
| **Nuclear Power** | yes | yes | yes |  | yes | yes | yes | yes | yes | yes | yes |
| **Wind Power** | yes | yes | yes | yes |  | yes | yes | yes | yes | yes | yes |
| **Temperature** | yes | yes | yes | yes | yes |  | yes | yes | yes | yes | yes |
| **Wind** | yes | yes | yes | yes | yes | yes |  | yes | yes | yes | yes |
| **Rain** | yes | yes | yes | yes | yes | yes | yes |  | yes | yes | yes |
| **Humidity** | yes | yes | yes | yes | yes | yes | yes | yes |  | yes | yes |
| **Air Pressure** | yes | yes | yes | yes | yes | yes | yes | yes | yes |  | yes |
| **Electricity Stock Price** | yes | yes | yes | yes | yes | yes | yes | yes | yes | yes |  |

Chart 2. Scatter dot chart options.

## Example UI

Made with Scene Builder

   
Picture 1. First iteration of user interface prototype

# API’s

In the solution we are using two API’s: Fingrid for energy data, and Finnish Meteorogical Institute (FMI) for weather data. Also if time permits we can gather electricity stock market price data from porssisahko.net.

Fingrid and FMI API’s were selected due to recommendation of ChatGPT. Pörssisähkö API was found manually without using ChatGPT’s help.

## Fingrid

Fingrid provides data related to the electricity production, consumption and multiple other variables that are directly related to electricity. Fingrids publicly listed but is owned by Finnish government and Finnish pension funds. Whole Fingrids dataset is vast, but we found about we are going to use five of them as described in Chart 1 and 2. Even those have some variation. For instance depending on data type it can be searched with hourly intervals, three minute intervals or even as forecast for coming hours (up to four days depending on the variable).

**Fingrids API documentation:** <https://data.fingrid.fi/open-data-api/>

**Fingrids available datasets:** <https://data.fingrid.fi/en/dataset/>

**Collection of multiple Fingrid datasets:** <https://fingrid-public.s3-eu-west-1.amazonaws.com/files/Avoin+data+tietoaineistoluettelo+9.6.2021.xlsx>

## Finnish Meteorogical Institute

Finnish meteorological institute provides weather data on multiple Finnish places. Datasets available are massive, but on our project we selected 5 datatypes to be included in the project (as seen on charts 1 and 2).

Finnish Meterogical Institute API documentation: <https://en.ilmatieteenlaitos.fi/open-data>

## porssisahko.net (extra)

Pörssishäkö.net is a small API service to get hourly electricity stock price rates. Data source it is not clearly stated, but it is strongly implied that at least forecast data comes from Nord Pool (which is an paid API service). This service only produces the stock price (and stock price forecast for the next day). It is a very simple API. On the downside each hourly data has to be looked by separate API calls. Searching for a days data therefore takes 24 API calls in total which makes it somewhat slow service for big datasets.

# Planned Architecture and design

## Activity diagrams of high-level use cases

### Program start-up

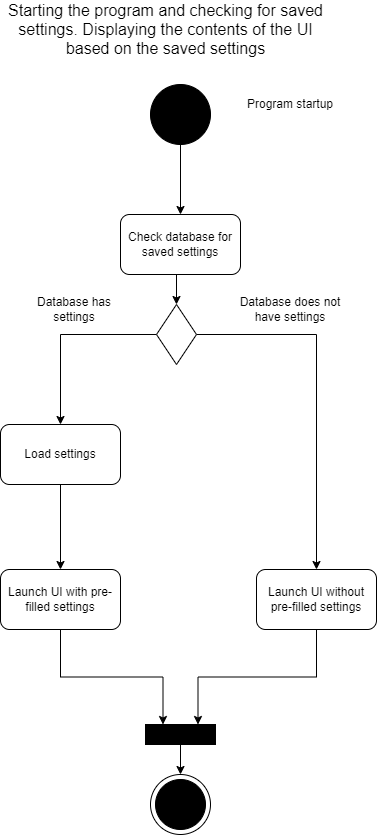


Diagram 1. Activity diagram presenting the process of the programs start up.

Before the programs user-interface is presented to user, the program will check, if the database has certain settings that have been stored. Reason why the check is done before the launch of the user interface is because the settings can have an affect how the user interface is presented. For example, if user has saved chart diagram creation related settings to database, these settings will need to be presented to user.

Note that the diagram does not consider all the possible setting related checks that will be executed. It is demonstrating the high-level idea. Also, as of writing this document it has not been decided whether a “real Database” (such as SQLite) shall be used or whether the settings are stored in a file. Regardless of the decision, the idea presented in the diagram applies.

If the Pörssisähkö API (section 3.3) is implemented, initiation of parallel process of fetching price-related information can be launched after the launch of the user-interface in case where the database lacks price-related information. The reason for this is that Pörssisähkö API is quite slow and in case user wishes to create price-related diagrams, it is good to cache price related information in advance for better user experience. The paraller process mentioned above is not described in the diagram.

### Validation of user inputs

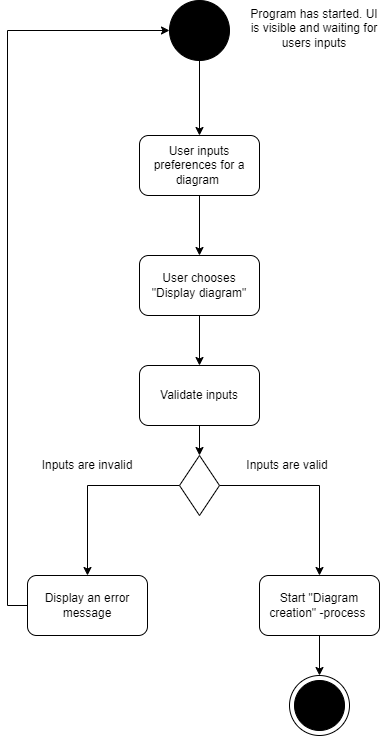


Diagram 2. Activity diagram presenting the process of user inputs validation.

Users’ inputs (such as variables regarding a diagram to be created, start date and end date) should be validated before a diagram will be created. Reason to validate user inputs is to prevent execution of expensive operations, such as API calls in case where user has for example forgotten to something mandatory from some of the fields in the user-interface. While the diagram does not clearly present it, the idea is that user-input related checks are done in user-interface layer (controller) before a service from a model is called. Assuming user’s inputs are valid, meaning that all the mandatory fields have appropriate values, the “Diagram creation” process can be started.

### Diagram creation process

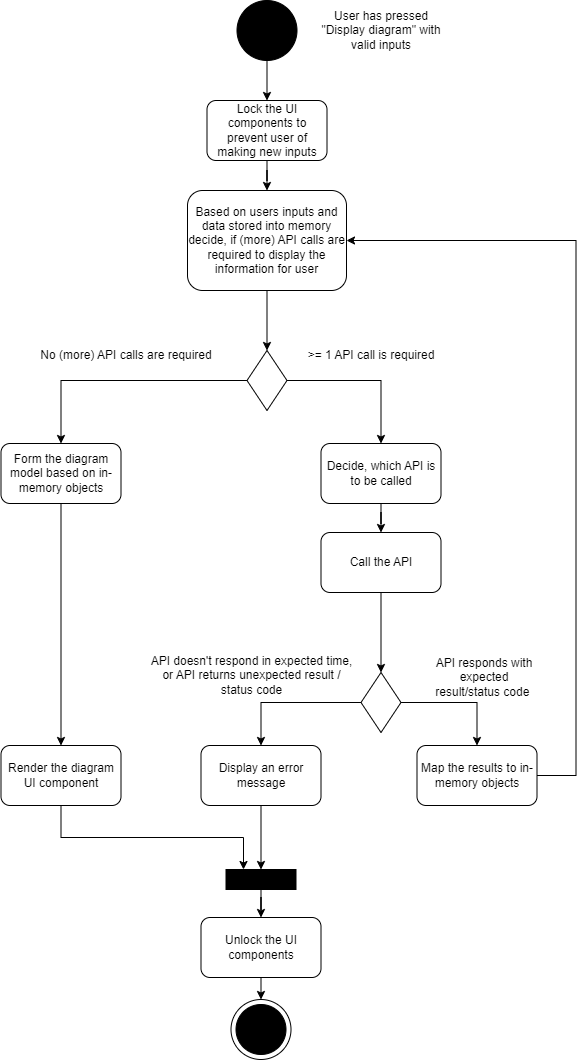


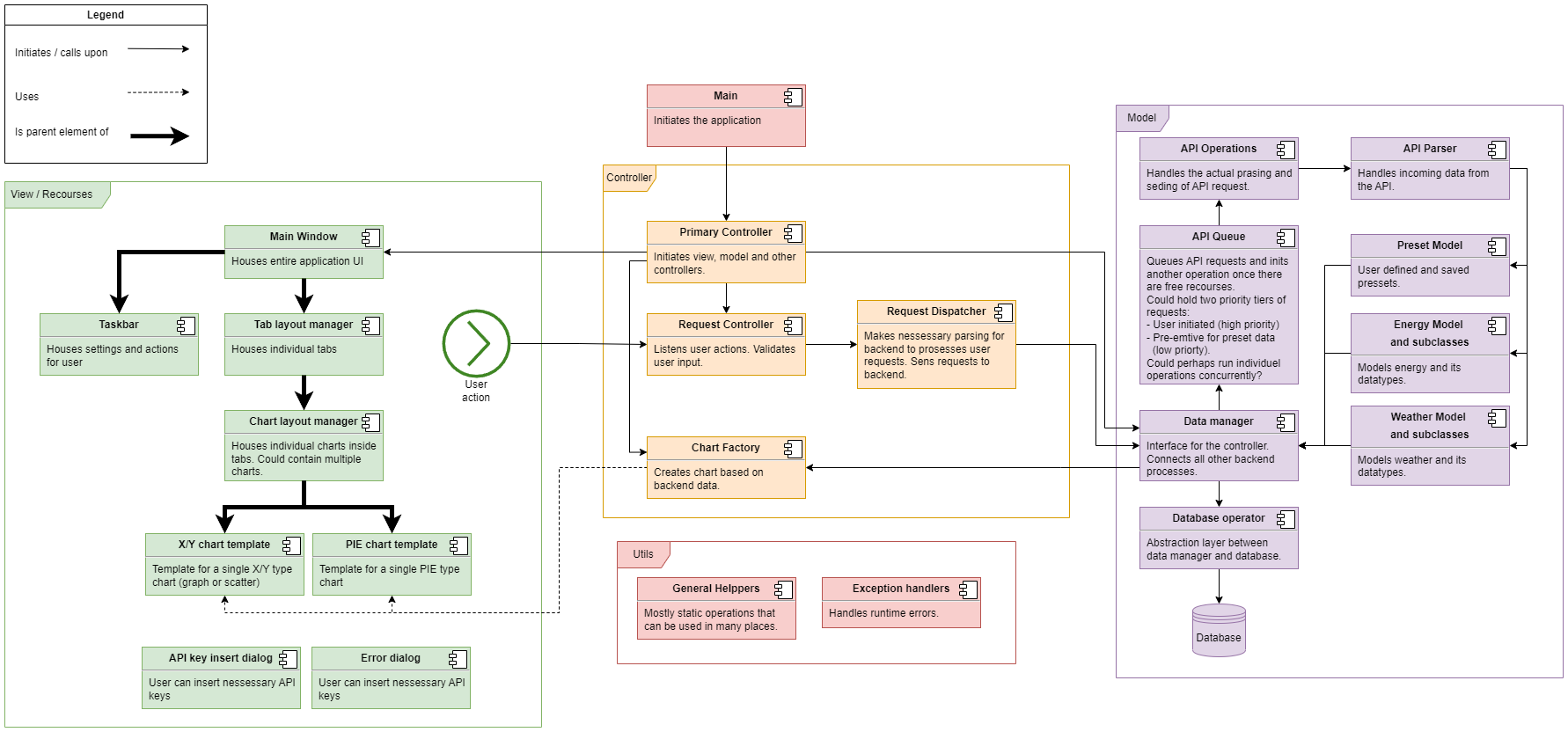
Diagram 3. Activity diagram presenting the process of diagram creation.

Once the users’ inputs have been validated, the diagram creation process can begin. The diagram creation process happens in the background, during which the user-interface should be locked. Locking the user-interface can mean for example a loading spinner to be shown in the user until the diagram creation process is ready. The diagram creation process in a nutshell means collecting the necessary data to serve users request, mapping the data to models of the program, and providing this information to user interface layer (controller), so that the diagram can be visually built and rendered.

Depending on the users request, getting the data requires 0..n API calls to be executed. In some situations, the program can already have the required data in memory. For example, if user has recently created a line chart with variables x and y for date range z to v, creating second diagram will not require an API call if the date range of new diagram is a subset of the date range z to v. In this case the program should have all the necessary information in memory.

Since API call can result an error caused by 3rd party (e.g. the API is down), these errors need to be properly dealt with. The program should not crash but instead display an error message stating “Technical error happened. Please try again later” as an example.

## Diagram of components within the system



## Design patterns and decisions

The program will be implemented utilizing MVC (Model View Controller) design approach. Our approach can be seen as “stack-like” where View is the highest element of the stack and model is the lowest element of the stack. In our approach the view never communicates directly with the model, meaning that the controller is responsible for acting as an interface between the model and the view.

User inputs are given from the user interface, which we consider as “view”. For any given input from the view, the controller validates the input and decides, whether the model is required. For example, when user presses “Generate diagram” button, the controller checks whether all the required fields have a proper input. If something mandatory is missing (e.g. end date has not been set), the controller informs the view to display proper error message and therefore it does not need to interact with the model.

The controller does validations such as mandatory field is not missing, or end date has been selected to be after start date. However, the controller does not know (and is not supposed to know) if an API is down, or whether date x has any information that an API can provide or not. This means that data and API related check is done in model and input related check is done in controller.

From technical point of view the view is static .fxml -file in the “resources” folder of the project. The controller will be implemented by creating classes which dynamically change the UI components defined in the .fxml -file depending on users interactions and information received from the model.

The model can be seen as the brain of the program. The model is responsible for acting as an interface toward the controller, executing API calls, maintaining the data models, mapping the API results to internal objects, and interacting with database. Heart of the model is data management service, which offers and interface towards the controller. The data management service is responsible for maintaining the data (internal objects regarding e.g. Weather, Energy) and knowing where new information can be asked if the controller requests information that is not stored in-memory during the time of the request. New information is mainly received from API’s, but the data management service is not the one executing the API calls. Instead of making the API calls, the data management service can call API service. The API service is responsible of knowing the API-specific requirements (such as required HTTP methods, parameters) and executing the API calls based on those requirements. The API service is never called from controller.

MVVM, an alternative to MVC, was not used, because the added complexity was deemed to outweigh the benefits of MVVM – easier testability, more extensive use of data binds to update UI automatically and separation of concerns. MVC is more suitable to simple applications with short development and shelf life. If application development lifecycle was longer, or the interaction and feature palette more complex, MVVM would be more suitable.

On the other hand, many team members had experience with MVVM, but not with MVC, so as an another benefit it was deemed better to learn a new architectural pattern, and not immediately get stuck with the same old people had used before.

## User interface design choices

We will start with a disclaimer. UI will have many moving parts, and the “best” decisions are not instantly obvious. Achieving a good balance with usability and features would require extensive prototyping and user testing, while this course work mainly concentrates on software design and programming. We will make here some assumptions about which solutions might work in our UI. Because of the complex subject, the actual results might not be the same as described here. We will most likely have to do multiple changes to UI and to they way how it works, while making the application. Some design decisions might be too hard or difficult to implement in practice, or the “cost-benefit” of the feature might not make the cut.

For the best user experience, the UI should allow the user to continuously interact with the UI, while for example chart generation tasks run seamlessly on background in a different thread. In practice this will cause problems with concurrency – for example if the user decides to delete a chart, or a tab, while another chart is being generated. Not locking the UI while a chart is being generated should thus be probably saved as a “stretch goal” for the application.

Many computers support displaying multiple charts at a same time, so the interface should be able to do it. We discussed having a 1 vs 4 chart display toggle button, and multiple tabs for charts, drag & drop between tabs etc. In practice these manifest multiple problems, of which the biggest could be data loss:

* What happens when the 1 -> 4 chart display button is pressed?
* Which chart is displayed, and which discarded when 4-> 1 chart display button is pressed?
* Should UI give pop-up warnings?
* Should user be able to select which chart is displayed when switching display modes?
* Should UI provide undo/redo in case user made mistakes while interacting with UI?
* Should the 1 vs 4 chart button be enabled only for new tabs?
* What happens in four chart mode, when yet another chart is generated? Does it go to a new tab, in a 4 chart or one chart mode?
* Should the UI have placeholders for charts in a 4 chart mode to imply that the UI supports displaying more than one chart at a time?
* What happens when the program is closed, and reopened, to the generated charts and tabs? It adds 0+
* 2even more complexity to regenerate the tabs & charts, like with a web browser.
* Should there be some kind of tab limit, and what the UI does if such a limit is reached?
* Tabs and charts should be able to be closed. Right click menu, a X button, both?
* Does every chart have a X button always visible, or do you need to select a chart for the close button to be displayed?
* How the drag & drop feature (if implemented) should work, and how the dragging & dropping action should be presented to the user?

In practice we must make initial design decisions. If something is revealed to be too difficult to implement, or requires too much time to implement, or is too confusing to use in practice, we must change the design decisions on the fly. It does not make sense to stick with dysfunctional decisions to the end, unless time constraints and other features dictate that we must do so.

# Realized architecture and design

## Application overview

As described in the introduction, this application takes electricity- and weather data and combines them in charts configured by the user. Data can be displayed related to time or to another data type which are chosen as the charts’ axes in the UI (user interface) and the type of chart to be used. The charts are displayed in tabs, the user can add another tab or replace the chart displayed in the currently selected tab. The user may select the timeframe in which the data will be displayed in relation to current time or select a start- and end date. The user can also save the current settings to be loaded and used at another time and specify which town’s weather data will be used.

The electricity related data is fetched from FINGRID and the weather data from the Finnish Meteorological Institute (FMI). User-defined settings are saved in a .conf file in the root folder of the program. The application uses MVC (Model-view-controller) architecture with each component being described more thoroughly in the following paragraphs.

Model is responsible for processing all the data of the software. Be it FINGRID’s electricity data, FMI’s weather data or user-defined settings, it’s all handled in the model. Model is divided into the following directories: api, data, datamodel, service and session.

View is the visual side of the program. It consists of an .fxml file in the resource folder. It defines all the components used in the UI.

Finally, Controller acts as a link between View and Model. It calls for data from model and makes a visual representation for the UI as defined by user inputs. Controller is divided into the main directory and factory subdirectory.

Along with these main components the program has types and utils directories which are used throughout the program. The directory types contains different enumerated variables which will be specified in chapter 6.1.4 and utils is used to house things like custom exception and environment variables (also in 6.1.4).

In the following chapter the described components of the software are described more in depth where the containing classes will be presented for each component.

### Model overview

Model is the backend of the program, and it is divided into directories which are called api, data, datamodel, service, session.

Directory api has api data handling related classes. AbstractAPIRequestBuilder which is an abstract parent class for API request creation. Api directory has two subdirectories which are called fmi and fingrid. These both contain a class that extends AbstractAPIRequestBuilder. These classes are used to create the API requests for their respective APIs and are called FingridAPIRequestBuilder and FmiAPIRequestBuilder. Also in both of those directories are classes for parsing the data given by the APIs. These are called FingridApiParser and FmiApiParser. Both parser classes extend an interface that is called APIParserInterface. In api there is also a listener class for API data APIDataListener, APIQueue which acts as a queue for incoming API requests and APIOperator which handles the requests.

The directory called data contains three classes: ChartRequest, DataRequest an DataResponse. ChartRequest contains the parameters required to create a chart, DataRequest has the parameters where an API request is going to be built from and DataResponse contains the DataRequest and its corresponding data.

Next up is a directory called datamodel. In short datamodel contains objects for storing different kind of data the program uses. We have AbstractDataModel which acts as a parent class for API data related classes, these are WeatherModel and EnergyModel. There is a class called SettingsData which stores user preset data, Bundle which acts as a parent class for the remaining two classes ResponseBundle and RequestBundle. They are used to bundle up requests and responses into a single object.

Directory service contains the following classes: DataManager, DataManagerListener, PresetParsing and DataStorage. DataManager acts as a link between Controller and Model. It receives from Controller what data needs to be fetched and forwards it to the correct classes and then returns the data to Controller. It also checks whether the requesting of data was successful or not. DataManagerListener acts as a listener for DataManager, DataStorage is used for storing data models and means of accessing said data and PresetParsing is used for saving and loading user presets.

Finally, the session directory contains the classes SessionChangeData and TabInfo. SessionChangeData is used to keep the UI updated what to show to the user and TabInfo has the name and id of a tab.

### Controller

Controller is the bridge between Model and View. It tells Model what to do based on events happening in View and makes a representation from the data it got from Model to be shown in View. Besides its main directory it has one subdirectory called factory.

The main Controller directory houses the following classes: PrimaryController, RequestController, RequestDispatcher, SessionController and SessionControllerListener. PrimaryController loads views and controllers and serves as the main logic of the program. It listens to DataManager in Model and SessionController. RequestController handles the user inputs from View, when the user does something it “informs” the correct classes in the program. RequestDispatcher sends the data from user inputs to Model and checks if the inputs are valid. SessionController stores instance of a tabmanager, and instances of chartmanagers inside the tabs. Gets updates to the session's state made by user, emits changes to session state to primarycontroller. SessionControllerListener is the listener class for SessionController.

The factory directory has three classes. One for handling chart creation called ChartFactory and two chart implementation related classes. One of these is ChartImpl, it is used as an abstract parent class for the more specific implementation class XYChartImpl. XYChartImpl generates the X and Y axis-based charts.

### View

View is the component visible to the user. It consists of an .fxml file that places the UI elements. It is in the resources directory as mainworkspace.fxml.

### Other

Along the MVC structure there are directories called types and utils. Classes in these folders are used across the program. The type classes are:

* APIType – types of APIs used.
* AxisType – types of axes available.
* ChartType – types of charts available.
* DataType – types of data available to be used in charts.
* MeasurementUnit – units of data types.
* RelativeTimePeriod – for relative time options in the UI.
* Scene – supported .fxml files.
* SessionChangeType – to inform listeners about changes in the UI.
* Status – simple success or failure (or pending).

The utils directory contains the following:

* CustomAlerts – for alerts shown to the user in the program.
* DataNotFoundException – thrown when data is not found.
* DateTimeConverter – converts time related objects.
* EnvironmentVariables – API key for FINGRID.
* Logger

There are three more additional classes within the program. These are App, which acts as a main class for launching the application, Constants which houses a constant for API timeouts and the maximum number of charts there can be in a tab. The last class is module-info which is the module descriptor.

Outside libraries used:

* JavaFX controls
* JavaFX graphics
* HTTP plugin
* Jackson core
* Jackson databind
* Jackson annotations
* JUnit 5

### Examples of communication between models

Kuva, joka sisältää kohteen teksti, diagrammi, Suunnitelma, Samansuuntainen

Kuvaus luotu automaattisesti

Diagram 1. Sequence for creating a Chart

Kuva, joka sisältää kohteen teksti, kuvakaappaus, diagrammi, Samansuuntainen

Kuvaus luotu automaattisesti

Diagram 2. Sequence for fetching data from backend

* Korkealla tasolla kuvaus applikaatiosta
* Komponenttikohtainen kuvaus

## Class diagram and internal interfaces

A class diagram can be found on as design documents attachment (attachment 2 classdiagram.pdf). The provided class diagram offers a structured visual representation of the application's architecture, detailing the primary classes, their methods, relationships, and interactions. It captures the core components ranging from data management and request handling to user interactions and backend processing, effectively laying out the blueprint for the application's internal workings and interfaces. Class diagram lists all public and protected interfaces of the application. Following however is description of the most important ones.

Interface between the controller and model concerning data requests is probably the most important interface and can be found on the DataManager class. Datamanager receives a RequestBundle type object. Once that is handled Datamanager will as a response create ResponseBundle to match the RequestBundle. This will trigger event on the PrimaryController to notify that the new data for chart generation is ready.

From the end users point of view the interface is the FXML files content generated in JavaFX is the interface for the application. One layer down the PrimaryController and RequestController handle the user inputs therefore being the interface for the FXML files. RequestControllers interface is used only when user actually requests data to be displayed. RequestControllers interface hides for instance input validation and chart request process from the user.

As said, once backend has processed data an event will be triggered so PrimaryController knows it can fetchData against sent request. This is implemented through DataManagerListeners interface which both PrimaryController and SessionController have implemented. This interface handles events for successful and failed data requests. On successful request the ChartRequest (what user asked for) and ResponseBundle (what data was given) are passed to the chartFactory by RequestController. ChartFactory hides the implementation of the chart creation process therefore serving as internal interface for chart creation.

On the backend there are also important internal interfaces. Firstly ApiQueue splits the Request received from the Datamanager. This is so that it can use APIOperators interface to process individual external API requests. This interface is quite convenient, since behind the scenes there is process of building, executing, and parsing of any individual (external) API request.

The application stores data n memory at runtime. Since DataManager has one interface for getting data it hides the fact that data can either be retrieved for external recourse or it can be retrieved from in memory data (if it is available). The controller will not actually know if successful data request was called via internet or from memory. Hiding this implementation behind DataManagers interface simplifies the process for the controller.

One interface that ties together many of the applications functions in many places is the enumeration class DataType. This class is responsible for defining what data types are available, what are they like, what do they describe and what API has their recourses. This interface is used in many different parts of the application. It centralizes the data types characteristics and therefore serves as an interface many parts of the application use.

One more interface that does not belong to any one particular part of the application are the utilities, error handlers and constants of the application. These can also be called almost from anywhere of the application and they add good level of abstraction for semi complex operations.

## Changes compared to original design

### Decision to not include database

In the component diagram (subsection 5.2), database operator and database were introduced as an idea of storing data. The idea was to cache data which has been received from API requests, to prevent executing unnecessary API calls. For example, if users input would have resulted in getting electricity production data from Fingrid between dates x and y, this information could have been stored into the database. If user created another input, resulting to requirement of getting data that was already stored, this information could have been gotten from database, instead of API.

Due to the nature of the project and introducing new requirements for the running environment of the application, idea of the database was scrapped. However, the problem of unnecessary API calls remained. The problem was decided to be solved by introducing data storage class instead of database operator and database.

The data storage stores data models (Energy models and Weather models) as in-memory objects. When data manager is called for data request purposes, the data manager checks the data storage, if callers request can be fulfilled with the data contained in the data storage. If callers request can be fulfilled, data manager does not call API Queue to provide the data from API. If callers request can not be fulfilled, data manager calls API Queue to get the required from data. After receiving the data, data manager calls data storage to store the data, so similar request can be fulfilled with in-memory data next time. The implemented solution allows caching data which has already been received from the API for the applications runtime. Once the

### Decision to not implement API call prioritization and data pre-caching to support Pörssisähkö API properly

The API Queue was initially designed to prioritize certain requests. For example, user inputs, which would have resulted in an API call, would have been prioritized higher than API calls which would have been done in the background unrelated to current user inputs. Idea of background API was found due to the nature of Pörssisähkö API, which according to tests could take minutes to return data. Data from Pörssisähkö API could have been pre-cached into data storage after initialization of the application. If background API requests were ongoing and another API call was required due to users input in the application, the API call initiated by user input could have been prioritized over background API requests.

A decision was made that implementing the Pörssisähkö API was not required due to two working API’s implemented in the system already. Also, while prioritizing the API calls sounded very nice idea, it would have been consuming too much time and wouldn’t have necessary created as much value, compared to value created by spending the time implementing other planned functionality.

### Introduction of session controller and extending tab manager

In the component diagram (section 5.2) there was no clear vision of which component is responsible for the state of the program. While data manager was considered as brains of the program, it did not seem to be obvious place to store information such as what charts are displayed in the user interface.

This led to the decision of introducing a session controller and tab manager. Session controller manages the programs state. It forwards users requests to data manager as a results of user inputs. Before each request towards data manager, session controller utilizes tab manager to find the correct tab where a chart will eventually be added into. Tab manager manages tabs in manners such as allowing creation of new tabs, allowing removal of existing tabs and maintaining the order of tabs created. Tab manager and data manager can be seen as independent components in the model, which are utilized by session controller as a result of user input coming from request dispatcher.

## Design patterns and solutions used

Adhering to well-established programming patterns provides a structured foundation that enhances code quality, maintainability, and readability. Each pattern we chose serves a distinct purpose, fulfilling specific requirements within our architecture.

1. Factory Pattern: Utilized for chart creation, the Factory pattern allows for the instantiation of objects without exposing the logic to the client. This makes it easier to add new chart types or modify existing ones without affecting the rest of the system. It encapsulates the complexities involved, providing a clean interface for chart creation. We only need to provide the data and parameters, and factory handles the rest.
2. Builder Pattern: Employed for constructing API requests, the Builder pattern enables the creation of complex request objects step-by-step. This improves code readability and provides a more maintainable approach for managing various request parameters. We chose to use this due to different API’s requiring completely different requests, with a large number of parameters.
3. Singleton Pattern: Many managers and controllers within the application are designed as Singletons, ensuring that only one instance of these components exists throughout the lifecycle of the application. This is crucial for coordinating processes and maintaining a single point of control. Singleton constructors had to be carefully engineered with double checked locking to avoid problems with multiple threads potentially initializing the same instance.
4. Threading: In order to enhance UI responsiveness, API calls are executed in a separate thread from the main UI thread. API calls and chart generation may take many seconds, depending on the size of the dataset. This allows the UI to remain responsive while lengthy operations are performed in the background. Threading must be employed carefully to mitigate the risks of race conditions and deadlocks.
5. Command Pattern: We utilize the pattern to streamline data requests and responses. DataRequest objects serve as encapsulated commands, which are bundled and processed asynchronously. The Command pattern allows us to decouple data-fetching operations from the UI, enabling better traceability and the ability to undo actions. This results in a more robust and maintainable architecture.

## Tests

Unit tests were added for certain parts of the application. Certain parts of the application, such as API request builders, API result parsers, and data models were developed practicing test driven development (TDD), where tests were written before the actual code. It was decided that including tests was not mandatory and therefore only certain parts of the application are tested with unit tests. Tests were also added for data manager and data storage. It is worth noting that especially tests regarding data manager test larger scope of the application rather than just the data manager. Data manager tests that the flow starting from situation where data request has been sent and ending where data request has been fulfilled with help of an API.

# Attachments

Attachment 1. Discussion with ChatGPT to form the idea.

Attachment 2. Class diagram. (classdiagram.pdf)

# References