COMP.SE.110 Software Design -project

Design document

Joni Niemelä H296319

Victor Zhou Jiang H290591

Tittamari Salonen K428205

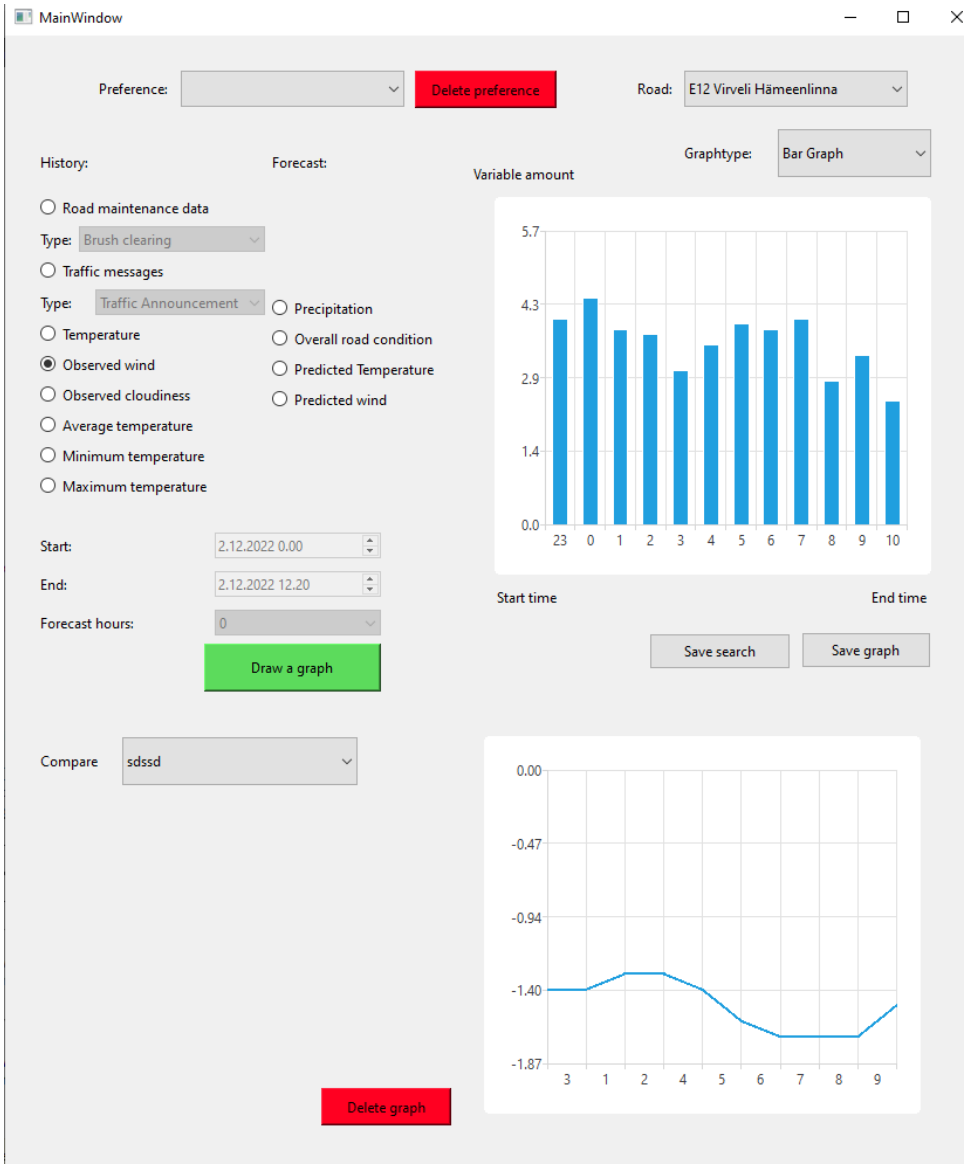
Altti Mäkelä H290890

Introduction

In this project the task is to design and implement a software for monitoring how weather affects road maintenance and condition. In the application the user can monitor road condition forecasts and weather separately. Road data will be accessed via Digitraffic API and weather data from FMI API. The group has agreed to use C++ as programming language and QT Creator as editor. The UI design prototype was made with Figma.

Graphical user interface

The GUI uses a search function to create graphs and compare data with the option to compare a currently searched one or a previously saved one simultaneously. The graphs are drawn separately vertically with time on the x-axis. In some cases there might be too many values which causes that the x-axis becomes too crowded and it won't display any real values. Search settings are inputted with drop down menus with the option to save them as "preferences". Drawn graphs can also be saved and imported at a later date. Saved preferences can be deleted from the drop-down menu where they are shown. Graphs can be deleted by viewing them from the compare-menu. When a saved graph is shown, it can be also deleted. The entire GUI is situated in a single window.



Picture 1. The finished UI

About the UI

We decided that the whole UI should be in a one window or view (Picture 1). The user can do a search with any data type, road or timeline possible for that data type. With the “draw a graph”-button the graph will be shown to the user. If the user wants to compare the graph to another, they can either save it and pick it from the “compare”-dropdown and do a new search or pick something previously saved from the “compare”-dropdown to see that graphs at the same time.

When the user has picked a data type that does not allow the user to pick a timeline, the possibility of picking the timeline is disabled. This is supposed to reduce user errors and make using the software easier.

If the user wants to save a search as a preference, for example a specific road and data type, they can use the “save search”-button. This will save the picked search options to the preferences-dropdown. From there the user can pick a saved preference (or search) and draw a graph with that information.

If the user wants to delete a saved search or graph they can do so by picking it from the dropdown and pushing either “Delete preference” or “Delete graph” -button depending on which one it is that they want to delete.

Right now there are five locations that you can choose from to draw a graph or get the wanted data but more can be added quite easily.

Note: precipitation data is only available 2,4,6,12 and as such the default 0 from the dropdown doesn’t draw a graph

Functionality and how it is implemented

Our team decided on the passive MVC model which can be seen from the class diagram. MainWindow uses the controller to communicate with the APIs through model which uses a networker class. Depending on the type of the API, returned file will be parsed using the corresponding parser. For example road conditions json from digitraffic will be parsed using jsonRoadConditionParser. The parsers will store the value/values and have a function that returns the stored values. After parsing the model gives stored datasets to the controller which passes them onto createGraph class which will do the necessary calculations and form a wanted graph. Then the graphs and calculations will be sent to mainWindow through controller which updates the user interface accordingly. Most of our functionality will be implemented using QT libraries as it has so many components that are perfect for our use cases. For example, mainWindow will be fully composed of QT assets.

Components and responsibilities

MainWindow will act as the view in the MVC model. It will create the user interface and show the graphs that the user request and also send signals to controller based on the user inputs.

Controller class is the one that will be the controller from the MVC model. It is everything that is happening in the background. Something that the user won’t see. It will receives signals from MainWindow and uses the model or other components (ex. Writing to files) as needed.

Model is basically a schema to hold the graph. It will also make API calls based on inputs to get the wanted data in a required form.

Networker is the class that will handle network requests to get data from the APIs. After that it will return the date in an unparsed format.

XmlParser is the class that handles parsing the data that is in xml form. In this case the data from Ilmatieteenlaitos. After it has parsed the data it will store it for later use.

IjsonParser is an interface for classes that parse json.

jsonRoadConditionParser parses road condition data that comes the digitraffic and stores it.

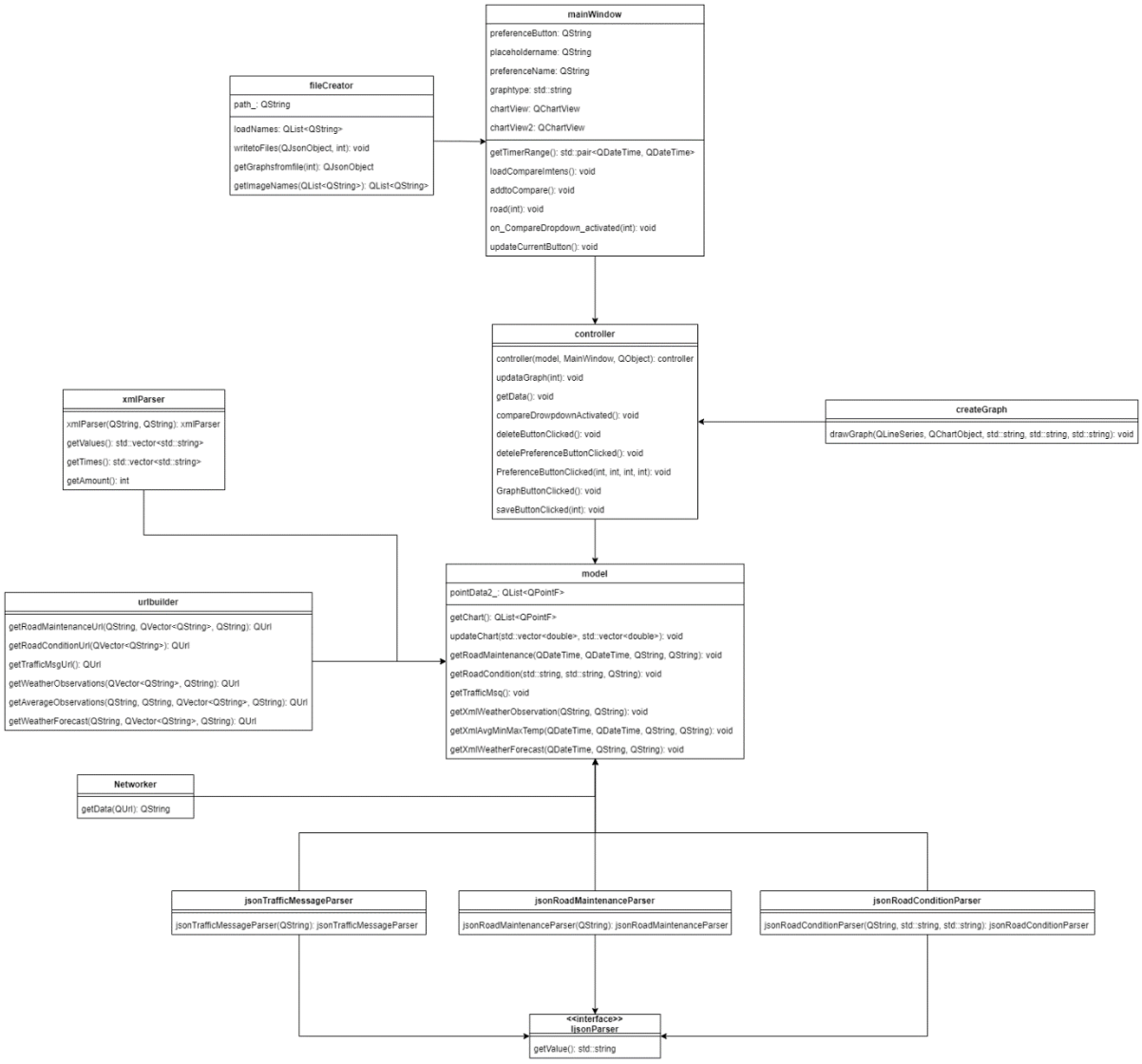
jsonRoadMaintenanceParser does thes same as jsonRoadConditionParser but for road maintenancedata.

jsonTrafficMessageParser does the same as jsonRoadConditionParser and jsonRoadMaintenanceParser but for traffic message data.

fileCreator class is used to save objects to files as json and to read and parse the jsonobjects from said files. Objects are graphs as text or images or preferences as text.

CreateGraph class creates the graphs that the user request and attaches it to an object that is given by a parameter.

UrlBuilder generates URL for network request from base URL with given parameters.



Picture 2.2 Class diagram

Decision making process

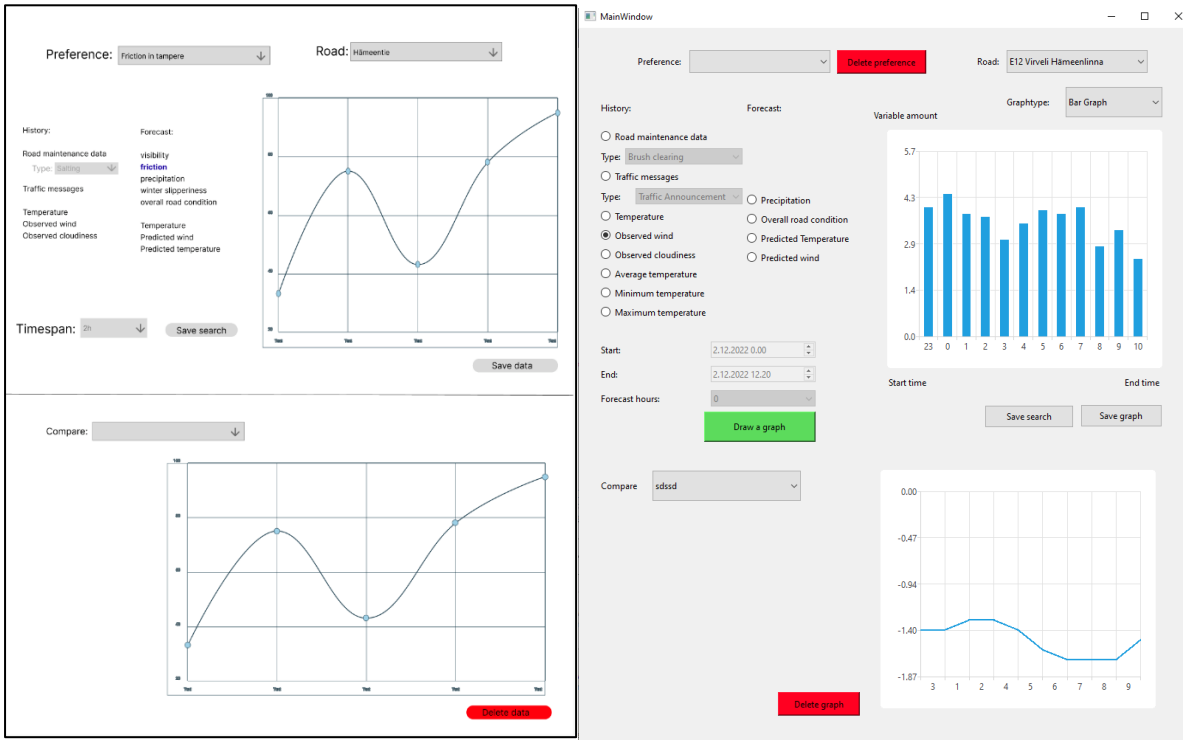
We decided on the passive MVC model because it fitted our needs the most and wasn't overly complicated for a project of this size. The passive version was chosen because we didn't feel the need for the active model as we thought that passive is more easier to use for the user.

The decision that led to current GUI layout and looks were that introducing multiple tabs wouldn't have been as user friendly as having everything in one single tab. In the start of the project we did think about having two sections in the window where both of the sections have their own input buttons etc and were able to draw their own graph on their own graph widget. This would have given user the ability to compare the drawn graph but we decided that it would be too much of copy pasting code and the wanted result can be achieved by input buttons in one section and using that you can draw on both of the sections.

Self-evaluation

Regarding the UI we were able to use our Figma prototype fairly well. The prototype is simpler because we did not know all the needed functionalities yet. For example, in our Figma prototype we have only one dropdown for picking the timeline but in the finished software we have tree for the different data types. Also in our Figma we did not have the "Draw a graph"-button as we thought that the graphs would be drawn in real time but ended up deciding that a button to draw the graph would be easier for us to implement and clearer for the user to use.

Still most of the functionalities ended up in the same places as we intended in our prototype. The graphs and most of the dropdowns are in the same places. Also the visual appearance is mostly the same type as we intended in the prototype.



picture 3. Figma prototype (left) comparison to finalized product (right)

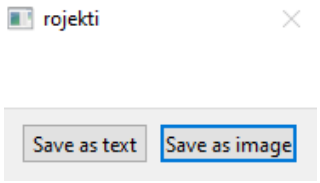
The backend of the code is also implemented quite well and it followed our original project diagram relatively well. Some changes here and there were made such as getting rid of unnecessary classes and adding new classes that made the code more easier to use and understand. For example we got rid of storer classes and added a class explicitly to create urls for API request. The urlBuilder class was a last minute decision but it was a needed add-on.

During the implementation process the code was factored. For example the model used to be passed onto the mainwindow but we decided to split the responsibilities of each component more clearly to follow the mvc model. As a result the model passes values onto the controller instead. Other refactoring decisions include combining json networker and xml networker together and creating an interface for json parsers. These would have been clear solutions from the start if we were more accustomed to the APIs and networking and data parsing in general.

Implemented bonus requirements

The user can choose between several plotting options by using the dropdown situated above the graph. This will change the current graph to the selected option and future comparisons to the same option.

The user can also save the visualizations as an image. A dialog window will pop up when the user tries to save the graph, which asks the user if they want to save the graph as an image or as a text. Saved graphs can also be visualized the same as text.



picture 4. Save options for data