# Design document Software Design COMP.SE.110-2022-2023-1

Emil Kaskikallio 050106789, Ilmari Marttila 265040, Joonas Paajanen 050115767, Tomi Lotila 274802

## Contents

# 1. Introduction

This is the design document for the task to design and implement a desktop application for traffic data visualization. In this document we will introduce the graphical user interface (GUI) and a high-level description for classes and interfaces to represent how individual components handle our data flow. We will be using Digitraffic API to fetch data to our programs use. The Programs outline is that it is done with C++ as a QT application which already gives us a good starting point for the communication between the user interface and the program itself. The prototype of the graphical user interface was implemented with Figma.

# 2. Software outline

Our goal is to make a piece of software that helps users to monitor weather and road conditions in Finland. Users can study how different weather conditions affect road maintenance requirements. The software fetches data such as weather forecasts, temperature, visibility, road slippage, road traffic, and traffic cameras. The road data is fetched from Digitraffic and weather data from Ilmatieteenlaitos. The software allows users to select locations and study data in graphical format. Users can also save data for later studies.

# 3. Graphical user interface prototype

A graphical user interface prototype is implemented with Figma prototyping software. For the home screen, in figure 1, we wanted to show a collection of everything such as road conditions currently and forecast, weather, and current ongoing maintenance work. If user wants to study more about a specific area he/she can press the ‘more’-button. Selecting or reselecting location is on the top bar.

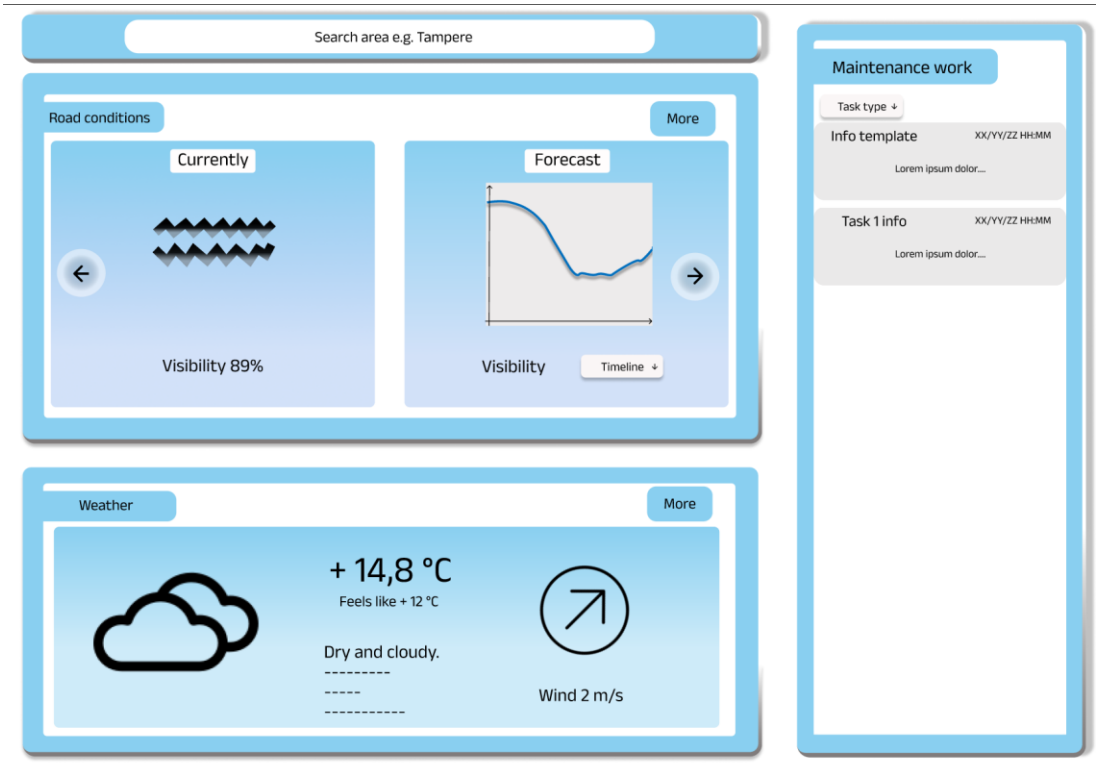



Figure 1. GUI prototype home screen.

After the user has pressed the more-button the screen will update to show bigger graphs and allows the user to change different variables such as the timeline. The ongoing maintenance work window and the location search bar will always show on different screens because users may want to go back to those. A home button will appear on the top bar to allow the user to return to the home screen.


Figure 2. The road condition screen is on the left and the weather info screen is on the right.

# 4. Component diagram and boundaries

We need to separate UI, logic, and data from each other. To accomplish this, we choose model-view-controller (MVC) as our software architecture. MVC architecture is very structural and doesn't allow circumventing the structure. The architecture allows us to work on different parts at the same time.

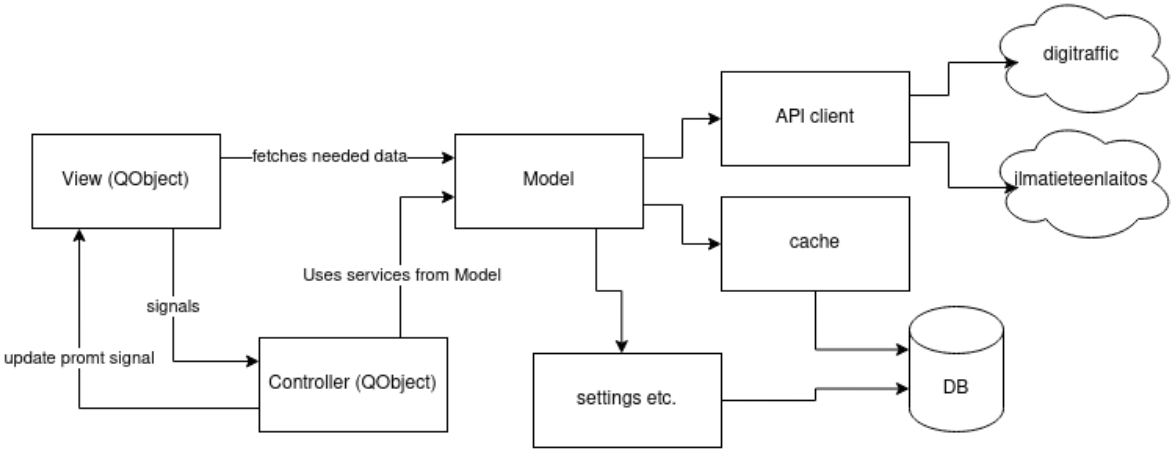



Figure 3. The software structure after design phase.

## 4.1 View

The view contains graphical interface elements. The view takes update command from controller and updates the graphical interface with the latest data. The view has access to data from model. The view will be implemented with QWidgets from Qt library. The model interacts with the controller with Qt signals and with the model trough pointer.

## 4.2 Controller

User interaction towards user interface triggers controller to call services from the model. After or during that controller – model interaction the controller tells the view to update its content from the model. The controller inherits from the QObject so that it can interact with the view with signals. The controller also interacts with model trough pointer.

## 4.3 Model

Model executes service calls from controller. The model might inform the controller if new data differs from the old data. The model should not have anything to do with the Qt. This makes it possible to change GUI modules later if one want to. Ultimately GUI could be replaced with CLI.

# 5. Class structure

Model is mainly used trough one main Model class that is composed of separate model submodules.

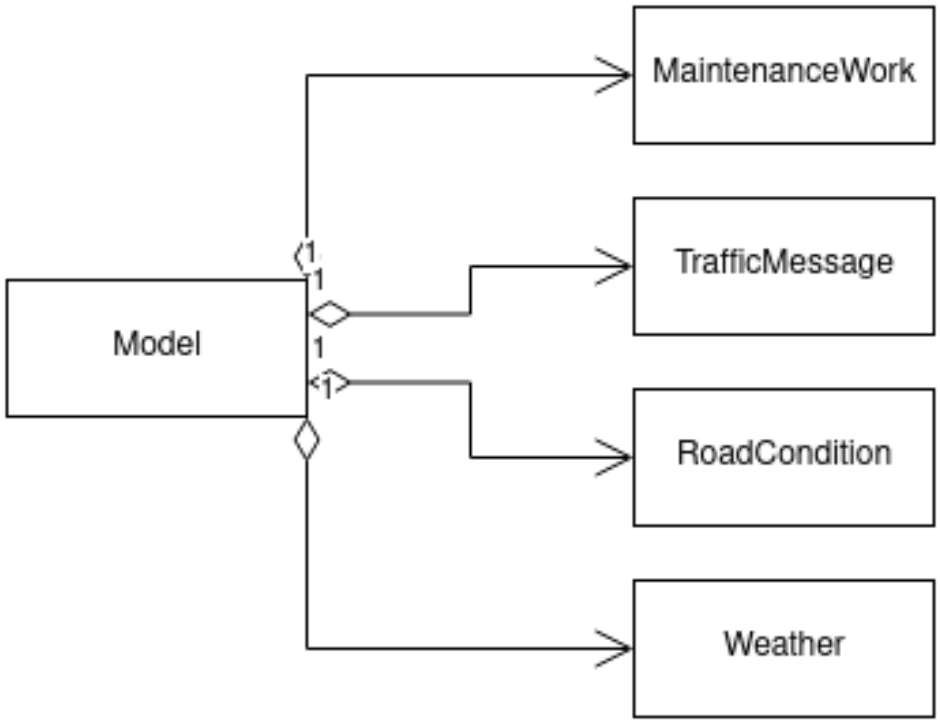



Figure 4. Class structure.

Submodules are separated from each other's because they will have different internal logic or interfaces. Most of these submodules will have methods that take a location and a time slot as parameters.

# 6. Self-Evaluation

Original design has been implemented well in terms of Graphical user interface for the mid-term submission. The data retrieval side of the project could have been implemented a little bit further for this submission. But the hopes are there that that part can be added to the existing UI without too big troubles and therefore make our projects current state Good in terms of the time we have left to implement rest of the application. For the visual part our design might even be a bit overkill

for this task as there were not really any demands on the visual appearance. The GUI however is a big part of the applications over all functionality and user experience, so we tried to stick to our prototype design as far as we could. Few corners have been cut for example the blue boxes don't have the radial corners as the prototype did because that part was highly only cosmetical issue. What we can't see from the applications current instance is the preparation work done behind the scenes to implement the data retrieval for the next deadline. We have prepared for example parsers for our different datasets to feed the data to our model. So those should be in a good position to start with the data flow.

# 7. Design decisions for mid-term

Changes made to design are listed in this section.

## 7.1 Model Design Decisions

We have made changes how model uses utility classes to get data from APIs. API client class is just for fetching data from internet. After Model gets data (API response) from API client it uses one of the parser functions from parser modules.

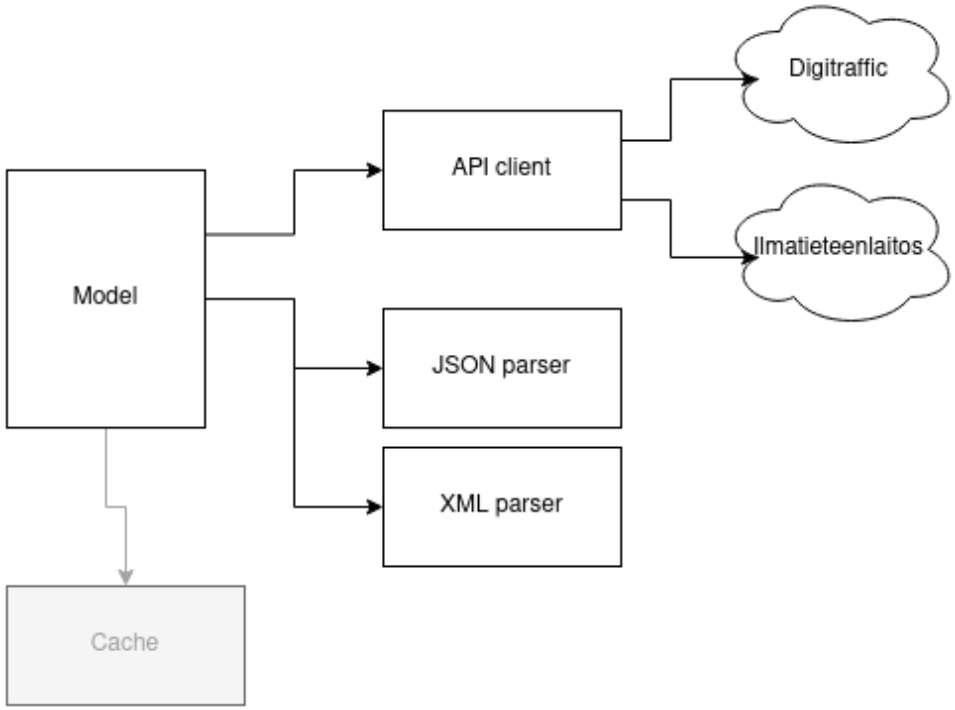



Figure 5. Model and utilities.

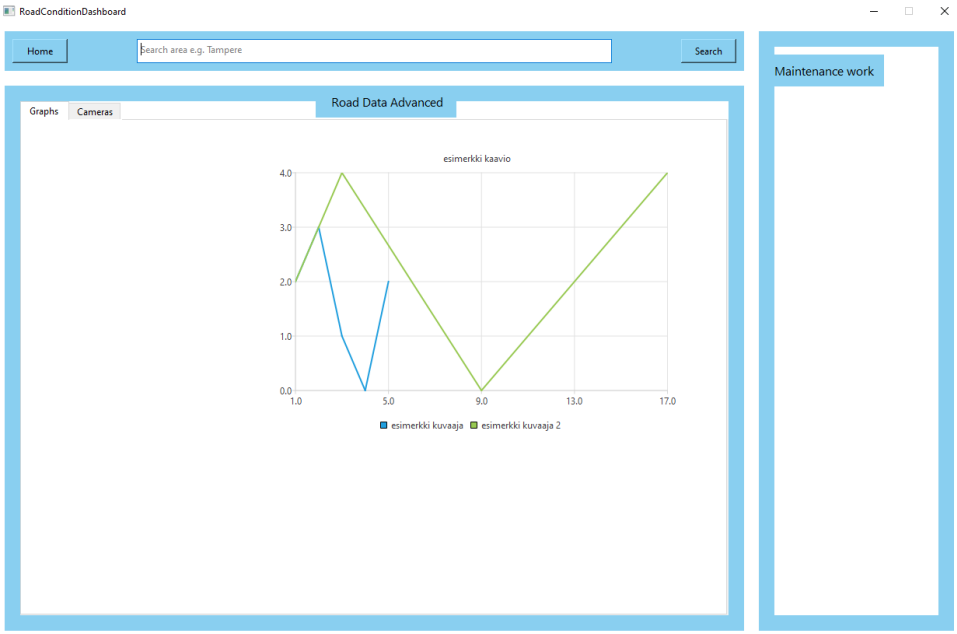
We are maybe leaving that cache feature out of this project. It would take too much time, and we want to do other things well.

## 7.2 GUI Design decisions

For the graphical user interface (GUI) we used the Figma prototype file to replicate our prototypes design with the QT design editor tool. In the screenshot below we can see the h*ome* screen which looks very familiar with the prototype we introduced earlier. For the mid-term submission we were able to implement the r*oad preview cards* for the current and predicted values. You can scroll

through a few cards with the arrow buttons and go to the *advanced road window* with the *more*-button. On the w*eather preview frame*, we have implemented a simple demo for data retrieval from the API. It is showing the windspeed and a timestamp for the measurement. The *road preview* element has currently only placeholder values, but the data retrieval should be straight forward to implement for the final submission. The data retrieval will also need the *Search bar* to be implemented as well as the *maintenance work* field. The layout for different screens is done with QT Stacked widget.


For the advanced Road window we will need to implement the controls for our graph to choose what data we want to see and what is the timeline we want to use. For Bonus we might try to implement the *camera* feature as a bonus.




### 7.2.1 Class for Road data preview cards ('Cardswidget')

One major design decision for the GUI was to build a separate QWidget class for the element inside the Road container in *home* screen. This is a QWidget class which has its own UI interface designed with the Qt design editor. Main principle of this class is that the class creates a graphical QWidget instance that can be created dynamically and then placed on the GUI.

Every card has the *Now* and *Predicted* sections with placeholder values. Ones creating a new instance of *CardsWidget* class the placeholder values for this instance get replaced with values based on the parameters given to the constructor. *Cardswidget* class also creates a new instance of the Chart class described below to show the predicted data for certain timeframe.

### 7.2.2 Class for Forming Qt charts ('Chart')

To plot the data, we decided to use Qt's own library to create charts. Qt charts have all the options we wanted to use for showing the data, like multiple plots for one chart, and labelled x- and y-axis.

Creating a Qt chart requires a lot of methods to be called and the plot data format has to be changed from vector to QLineSeries. So to keep the MainWindow class clean we decided to create a new class for chart creation. Our software has multiple charts, so we have less code duplication when using a chart class.

# 8. Libraries

We have mainly decided which libraries we are using for this assignment. We predicted that cache is done by using SQLite library, but at the time it seems that we do not have time to do that.

## 8.1 GUI

GUI parts of this software will be made with Qt6 libraries.

## 8.2 Web Requests

Web requests are made with libcurl. Libcurl is used through its C API, which is maybe not the most convenient way to write software. C style programming is kind of a different in comparison to modern C++ development. There is a risk that C style manners are spreading outside its wrapper classes.

## 8.3 Response Parsing

Incoming web responses have to be parsed so that their data is usable by model classes. This parsing is done in parser modules. At the time parser modules consist of two namespaces which include parsing functions for every datatype we are parsing.

Finnish meteorology institute data is in xml format. These incoming xml documents are parsed with well-known xml parsing library called TinyXml-2. API on this library is not too easy to use and its kind an error prone to typos. Anyway, it is still usable.

Data from digitraffic is in json format. Json format is more modern than xml in scope of data APIs. Unfortunately, C++ does not have any built-in way to handle json. For handling and parsing these json responses we are using json header only library maintained by Niels Lohmann. One of the group members have already used this library a lot, so the decision was easy. Because library is header only, it should be included as few times as possible and only to implementation files to keep compilation units small. At the time it is included only once, to the digitraffic parser implementation module.

# 9. Sources

## 9.1 Weather and Road Data

- https://www.digitraffic.fi/en/road-traffic/
- https://tie.digitraffic.fi/swagger/
- https://tie.digitraffic.fi/swagger/#/Maintenance
- https://tie.digitraffic.fi/swagger/#/Data%20v3/roadConditions
- https://tie.digitraffic.fi/swagger/#/Traffic%20message
- https://en.ilmatieteenlaitos.fi/open-data-manual

## 9.2 Libraries

- https://curl.se/libcurl/
- https://leethomason.github.io/tinyxml2/
- https://github.com/nlohmann/json