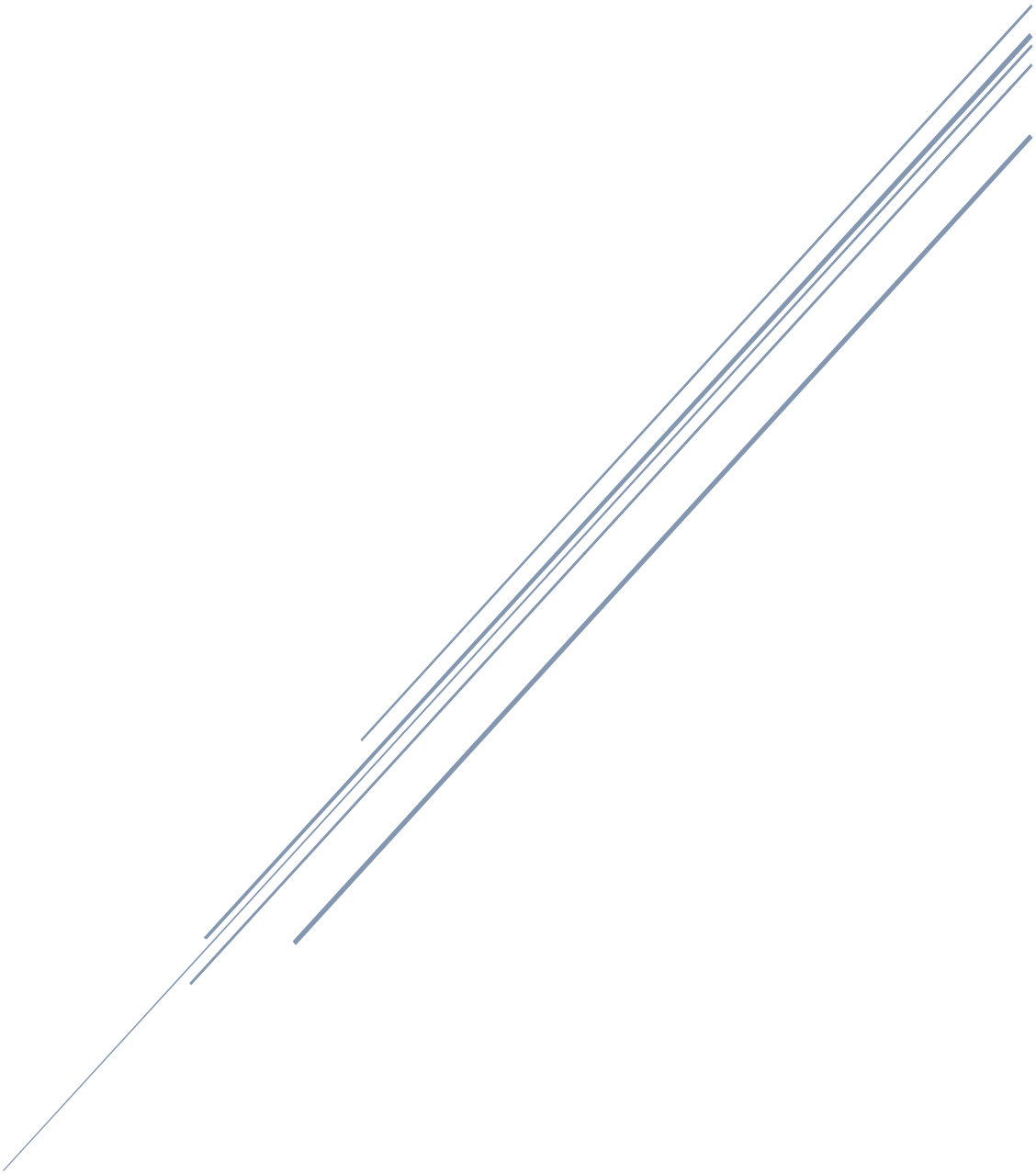
TRAFFIC-WEATHER APP DOCUMENTATION

Katariina Kariniemi, Hiski Hämäläinen, Arttu Raatikainen



Tampere University

COMP.SE.110 Software Design (2023)

CONTENTS

1. Introduction

Documentation for software project. The following chapters contain breakdown of the project; used design methods, and components of the program.

2. Application idea and prototype

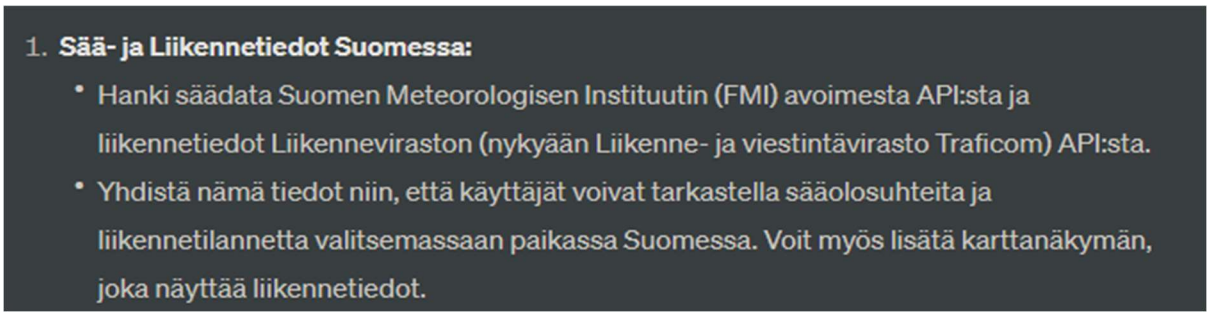
Our app shows weather and traffic data, alongside with a picture gotten from a traffic cam, of a specific place (where there are sensors of course). We also calculate whether the driving conditions are good or bad or decent and show that with an emoji.

Our program is made with Java. We're using two APIs:

https://www.digitraffic.fi/tieliikenne/#liikenteen-automaattiset-mittaustiedot-lam and

https://www.ilmatieteenlaitos.fi/kysymyksia-avoimesta-datasta

We used ChatGPT to generate our idea (picture 1) and refined it further ourselves.

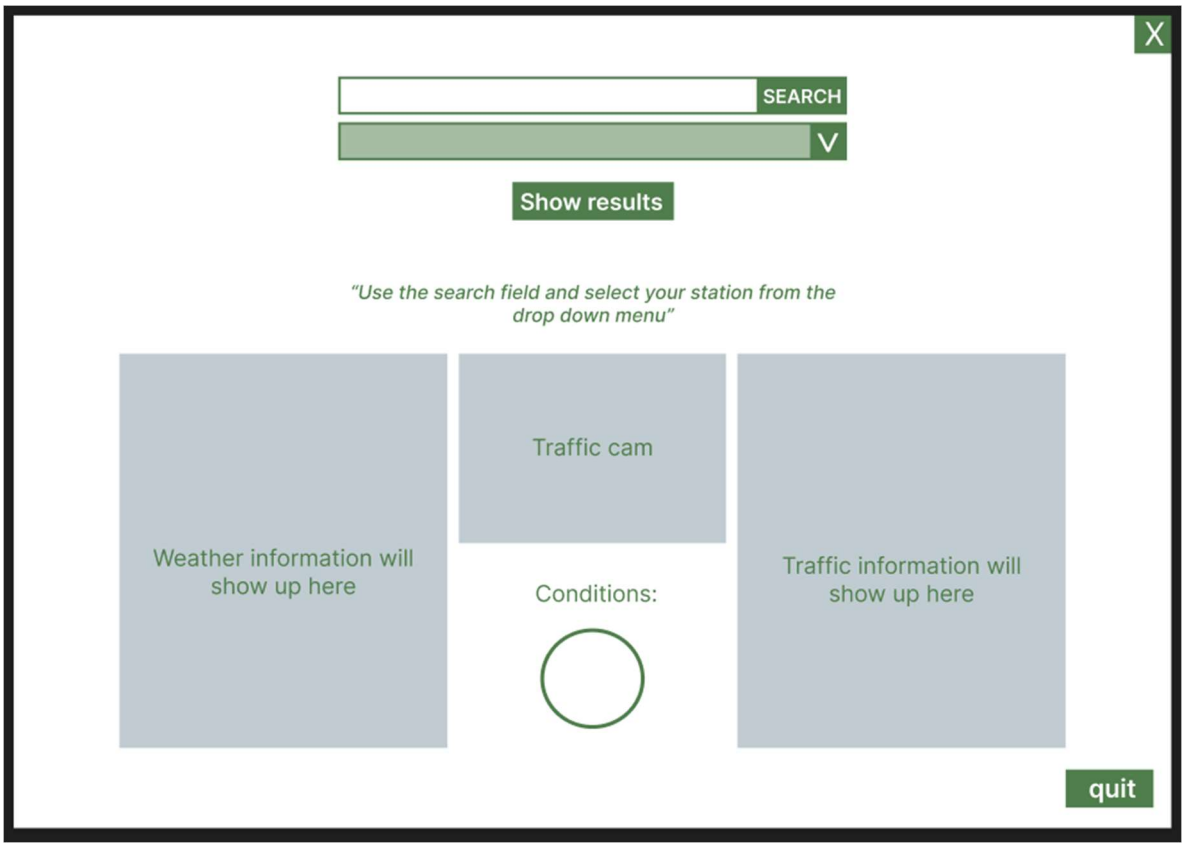


Picture 1: ChatGPT prompt.

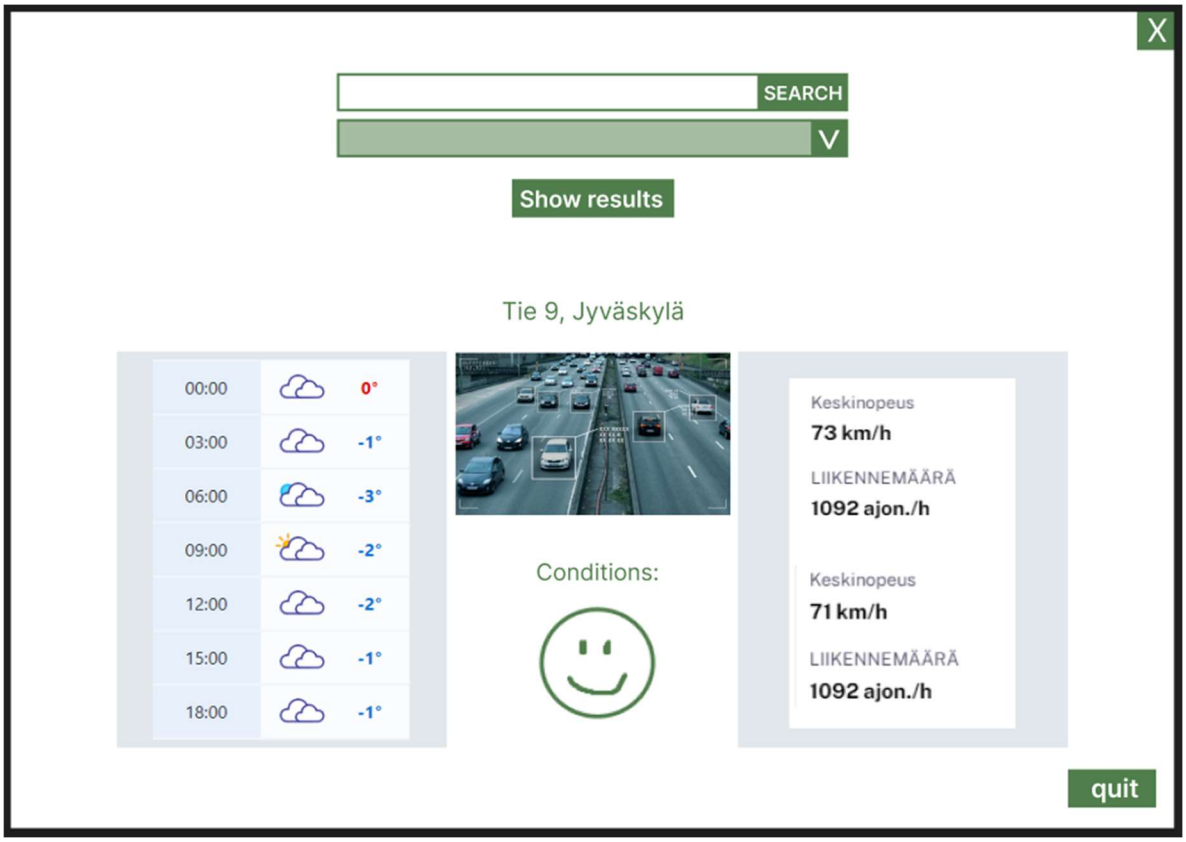
2.1 Prototype

Below are presented examples of the initial idea of how the program should look like. A very rough idea of the UI is illustrated in picture 1. In picture 2, we used SceneBuilder (which we plan to use for the design of the UI) for an idea of how the actual application window should look like.

These will provide a mainframe around which we will start building the program. However, these are not final designs and may and will get, more or less, refined and redesigned.



Picture 2: Sketch of the UI starting screen



Picture 3: Sketch of the UI after station selection.

3. High-Level Description

Application aims to provide users real-time weather and traffic data as it was mentioned in chapter 2. It shows user weather conditions for the next 18 hours with 3-hour intervals, weather camera from highway and traffic data such as average speed and car amounts per hour. Users can change camera or view easily by searching stations by city and clicking on station.

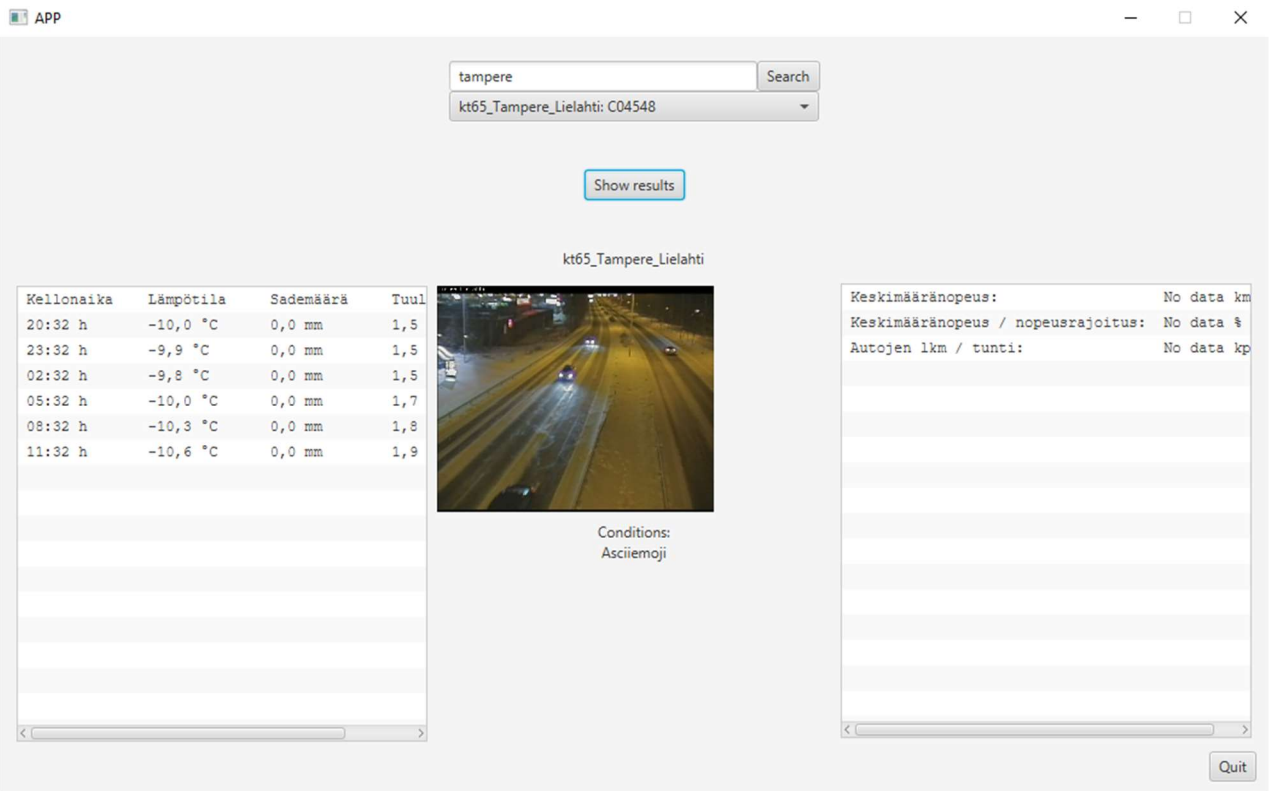
Application is developed using Java and integrates two APIs which are:

- https://www.ilmatieteenlaitos.fi/ (weather)
- https://www.digitraffic.fi/tieliikenne/#liikenteen-automaattiset-mittaustiedot-lam (traffic data)

Sketch of the user interface is presented in picture 3.

Final UI is presented in picture 4.

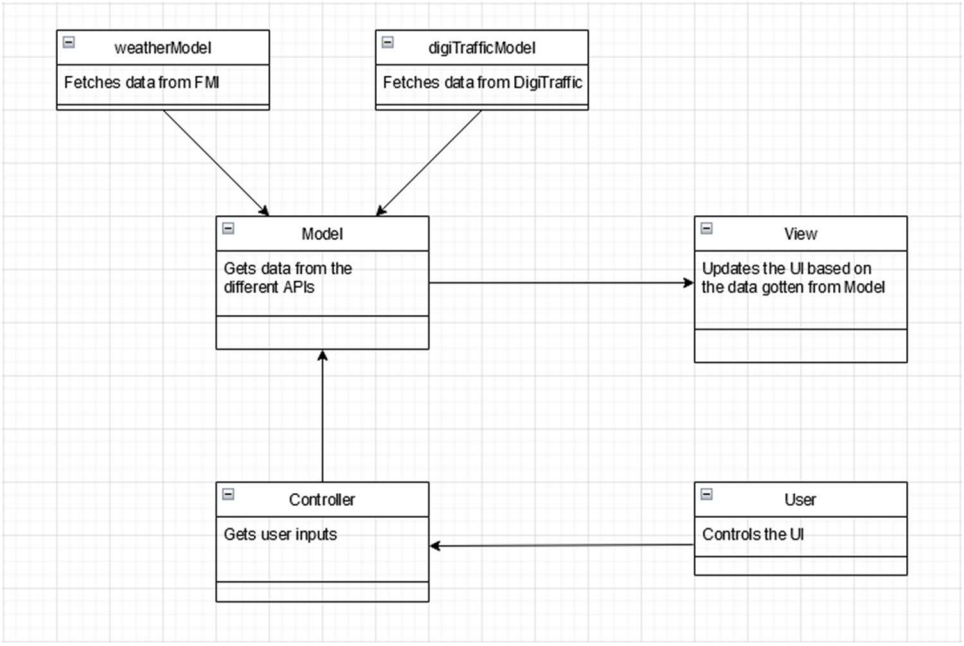
One of the main goals was to use design patterns and we chose to use MVC pattern. More on that in chapter 6.



Picture 4: Picture of UI

4. Boundaries and interfaces

Application works through a UI, with which the user can interact. Controller gets the inputs of the user and sends the inputs to the model. The model fetches data from our two API extractors and sends it to the view, which updates the UI.



Picture 5: Class diagram

5. Components and responsibilities

WeatherApi.java

Gets weather data from https://www.ilmatieteenlaitos.fi/ . Data from APIs are in. wfs format and it transforms the data to a string format of "hours:temperature:rain:windspeed".

There are implementations for weather observations and forecast.

DigitrafficApiExtractor.java

Gets traffic data from https://www.digitraffic.fi/tieliikenne/#liikenteen-automaattiset-mittaustiedot-lam . There actually are two different base API urls to call: one for the weather cams and second for the traffic data. Extractor handles data in JSON format. Data from the API is encoded in GZIP-format, which is decoded and turned into a JSON object. We used a library made for handling GZIP and the GSON library for this. Libraries made for fetching data with HTTP request were also used in order to send proper HTTP requests to the API. This data can be then used by the Model to update our app. The data contains the number of cars, their average speed compared to the speed limit and the average speed of the cards during the last hour. A picture of the road is also fetched.

There are some places where we can't get a picture with the headers provided in the API documentation.

Image snippets from digitraffic.fi which traffic sensors we are using.

5054 5055	OHITUKSET_60MIN_KIINTEA_SUUNTA1 OHITUKSET_60MIN_KIINTEA_SUUNTA2	kpl/h	Ilmoitetun 60 minuutin aikavälin automäärä.

5158 5161	KESKINOPEUS_5MIN_LIUKUVA_SUUNTA1_VVAPAAS1 KESKINOPEUS_5MIN_LIUKUVA_SUUNTA2_VVAPAAS2	% vapaasta nopeudesta	Keskinopeuden prosenttiosuus määritellystä tien vapaasta nopeudesta viimeisen viiden minuutin ajalta. Arvovastaavuudet: 0 – 10 Seisoo 10 – 25 Pysähtelee 25 – 75 Hidasta 75 – 90 Jonoutunut 90 – 100 Sujuvaa 100 – Yli vapaan nopeuden
5056 5057	KESKINOPEUS_60MIN_KIINTEA_SUUNTA1 KESKINOPEUS_60MIN_KIINTEA_SUUNTA2	km/h	The average speed for a given 60 minute period.

Model

Sends request for weatherApi and digiTrafficApiExtractor with provided parameters from controller and updates model class private objects. The Model takes responsibility for formatting data to a suitable form for the View.

The model component also calculates a suitable emoji for the driving conditions. The algorithm for the calculations is provided by ChatGPT and it considers parameters weather, rain, windspeed and traffic. Emoji has three states based on the calculations: “”, “😐” and “☹”

ViewBuilder

Constructs the UI window.

FXMLController

A class for implementing functionalities to elements in fxml-file.

Controller

Communicates between view and model elements. It gets data from Model and then updated viewBuilder accordingly.

App

Launces and runs the app.

6. MVC Implementation

In planning phase, we discussed about different design patterns and tried to think about the best solutions. We weren’t quite sure which pattern would be the best, so we ended up with MVC. We did learn quite a lot from other patterns such as MVP or MVVM when reading and trying to understand implementation though.

Advantages of using MVC are quite simple. Developers can work simultaneously on model, controller and views and components become more manageable and less dependent on each other.

Disadvantages could be a too complex implementation and misunderstood architecture which will lead to troubles.

Diagram of the implementation is shown in picture 5. The diagram shows each component and a little of their responsibilities.

Model

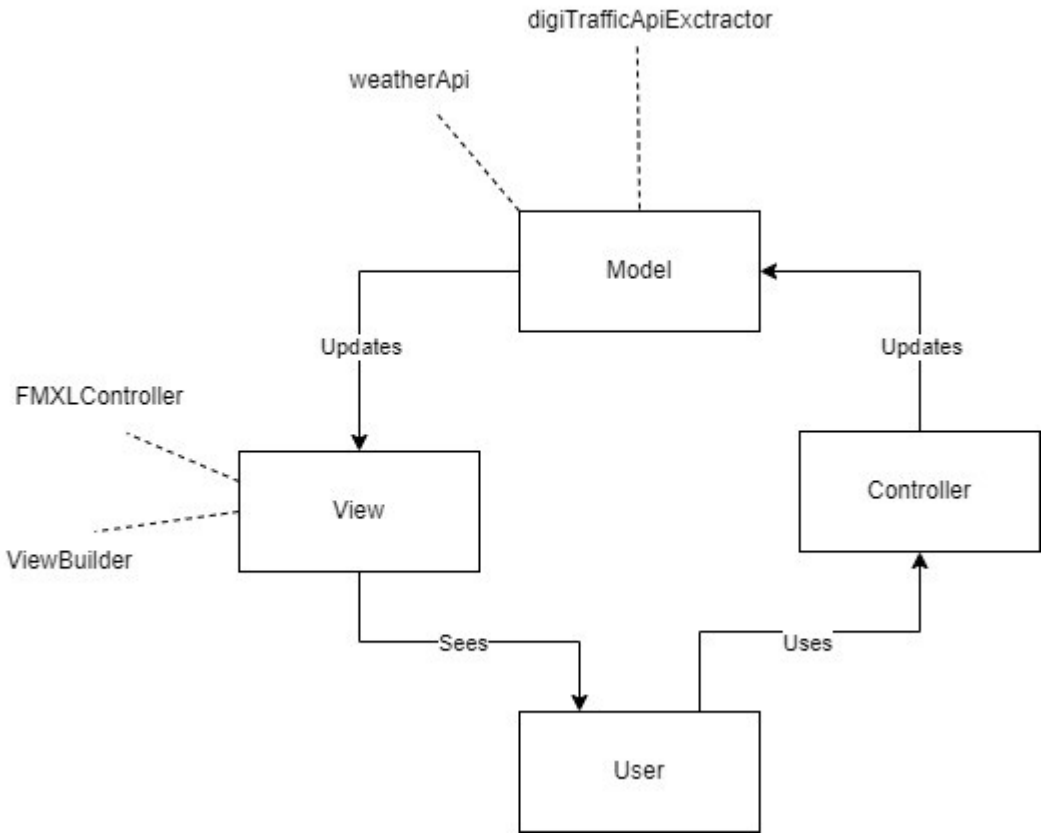
The model component takes care of the logic and data handling. In our implementation we have one Model class which uses weatherApi and digiTrafficApiExtractor for APIs.

View

The view component represents the UI. It has been divided into two components FMXLController and ViewBuilder.

Controller

The controller component works between the Model and View components. It takes care of the user input, such as buttons and text labels. Controller uses the input to activate Model component. For example, when user chooses city and station, it triggers Model to fetch data from APIs and update data components. After triggering Model, it updates changes to View.



Picture 6: MVC diagram

7. Self-evaluation

Usage of interfaces was quite challenging as the data in weatherApi was in wfs format and the developer working on it didn't get it to fetch JSON format. Therefore, a decision was made to not to change it to a JSON format and use it as String. In our opinion it didn't make later development any harder, but it could have made the code more complex to read/follow. In the best scenario, there would be an interface for APIs.

Fetching data from DigiTraffic was also quite challenging, since it required a lot of fiddling with HTTP calls and protocols, which wasn't entirely clear or well explained in the API documentation. The data was also encoded, which wasn't explained anywhere, which slowed the implementation. It didn't, however, make later development harder.

Java as the programming language was a good choice and implementation of the MVC was quite clear (at least at the end). We are not sure if it could have been easier to implement with other languages, but with our skillsets, Java was the best option.

8. The use of AI

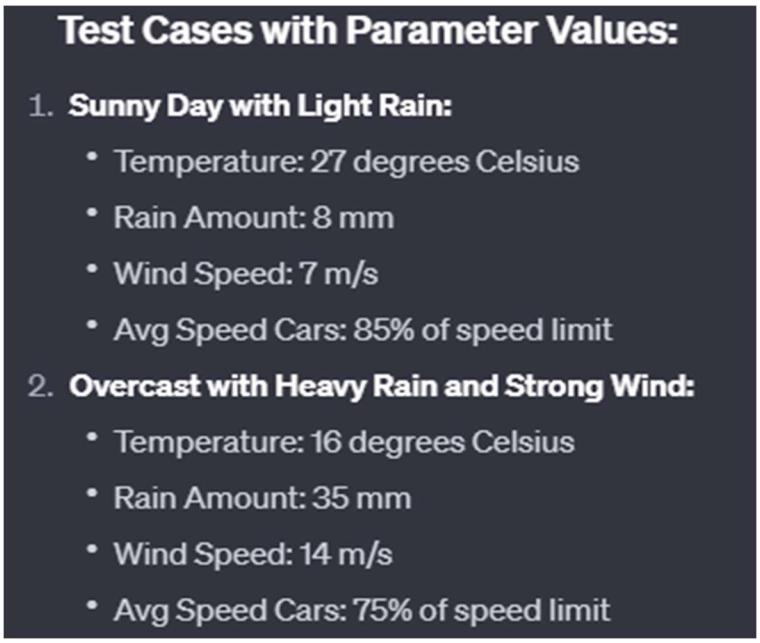
In the planning phase of the application, we used ChatGPT for app ideas. We gave ChatGPT the group_assigment_specification pdf document and it gave us multiple great and doable ideas for application. We ended up with the one presented in previous chapter 2 (picture 1).

The usage of ChatGPT in the planning phase was a success. It can brainstorm ideas much faster than us and present them in an easily readable way.

ChatGPT was also used to help create an algorithm for calculating driving conditions. Message to ChatGPT was:

"I have java project where i try to calculate whether driving conditions are good, medium, or bad somewhere. My parameters are weather temperature, rain amount, windpeedMS, trafficCars (how many cars passing a sensor in a hour), avgSpeedCars (% amount of speedlimit) and avgSpeedKmh. Can you give me an algorithm for how to calculate driving conditions "

The algorithm provided was good and gave more ideas to improve it. After a few rounds of chatting, we ended up with a solution that was good enough for our case. The test cases for algorithm were done by telling it to create 5 different real-life conditions for parameters. Two examples of the test cases are shown in picture 5.



Picture 6: Test cases in ChatGPT

Another tool used was github Copilot. Copilot was used as a help in MVC implementation. Usage of Copilot was very easy. We only needed to use the chat feature just like in ChatGPT.

Copilot was a great help for us. MVC became more understandable when examples were provided from our project. Two key responses are shown in pictures 7 and 8. However our implementation differs from Copilot answers. For example, Copilot puts the Model parameter for Controller constructor. We did not see that needed in our way of implementing.

In this example, the `getWeather` and `getTrafficCam` methods return the current weather and traffic cam data, respectively.

Ratikka

Can you show me how now controller would work with model class

GitHub Copilot

Used 1 reference >

Sure, the controller would interact with the `Model` class to update the model based on user input and to retrieve data from the model to update the view. Here's an example of how you could implement this:

```
public class Controller {
    private Model model;
    private FXMLController view;

    public Controller(Model model, FXMLController view) {
        this.model = model;
        this.view = view;
    }

    public void updateWeather(String url) throws Exception
        model.updateWeather(url);
        String weather = model.getWeather();
        view.updateWeatherLabel(weather);
    }

    public void updateTrafficCam(String url) throws Except
        model.updateTrafficCam(url);
        String trafficCam = model.getTrafficCam();
        view.updateTrafficCamLabel(trafficCam);
    }
}
```

In this example, the `updateWeather` and `updateTrafficCam` methods in the `Controller` class call the corresponding methods in the `Model` class, get the updated data from the model, and then call methods in the `FXMLController` class to update the view.

Picture 7: Controller

ure, based on the MVC pattern and the previous `Controller` class, you can modify your `FXMLController` class like this:

```
public class FXMLController implements Initializable {
    @FXML
    private Label stationLabel;
    @FXML
    private Label emojiLabel;

    private Controller controller;

    public FXMLController() {
        Model model = new Model();
        controller = new Controller(model, this);
    }

    @Override
    public void initialize(URL location, ResourceBundle re
        // You can put any initialization code here
    }

    @FXML
    private void quitProgram() {
        Platform.exit();
    }

    @FXML
    private void searchStation() {
        try {
            String weatherUrl = // get the weather API URL
            String trafficCamUrl = // get the traffic cam
            controller.updateWeather(weatherUrl);
            controller.updateTrafficCam(trafficCamUrl);
        } catch (Exception e) {
            // Handle exception...
        }
    }

    public void updateWeatherLabel(String weather) {
        stationLabel.setText(weather);
    }

    public void updateTrafficCamLabel(String trafficCam) {
        emojiLabel.setText(trafficCam);
    }
}
```

Picture 8: View