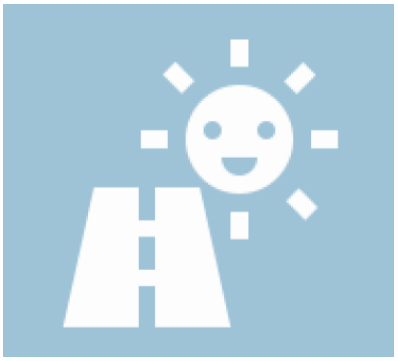
COMP.SE.110 - Software Design - Group3

Amanda Dieuaide
Aleksi Iso-Seppälä
Jukka Pelli
Lauri Puoskari

Project RoadCast

December 2nd, 2022



Road and Weather Condition Forecasts Monitoring Application

Design Document (v.3)

Fintraffic

ILMATIETEEN LAITOS
METEOROLOGISKA INSTITUTET
FINNISH METEOROLOGICAL INSTITUTE

Document version control

Version release history

VERSION	MODIFICATIONS	DATE
1.0	Added: High-level description Boundaries and interfaces Libraries and features Design solutions	30.09.2022
2.0	Updated: - Code Architecture - UML Diagrams Added: Self-evaluation Work Division	28.10.2022
3.0	Updated: - Chosen technologies and libraries - Architecture and Detailed Design - User Manual - Self-evaluation Added: How to run the project	02.12.2022

Table of contents

Introduction & Project Requirements

The following document will go over the specifications of project RoadCast and the design and architecture choices of Group3.

Goal: Develop a desktop application that allows users to access road and weather forecasts according to variables such as location, time, and user preferences.

Functional Specifications:

- The User can choose the Traffic data they want to see (set of options) + Visualizations with Graphs/Plots:
    - Adjust Timeline: hours, days, weeks, months
    - Adjust Coordinates: latitude/longitude GPS
    - Show how weather affects road conditions forecast -> Show for 2, 4, 6 or 12 hours + Select Visibility/Friction/Precipitation/Winter slipperiness/Overall road condition
      -> Detailed Messages

API: https://tie.digitraffic.fi/swagger/#/Data%20v3/roadConditions ;

    - Show required road maintenance -> Show Different Tasks types + Average amount of tasks/day (for given Timeline + Coordinates)
      -> Graph

API: https://tie.digitraffic.fi/swagger/#/Maintenance ;

    - Show traffic messages -> Show Amount of messages (for given Timeline + Coordinates) *-> Detailed Messages*

API: https://tie.digitraffic.fi/swagger/#/Traffic%20message ;

- The User can choose the Weather data they want to see (set of options) + Visualizations with Graphs/Plots:
    - Show and combine:
        - Temperature: Average Daily and Min/Max (in Location + Month)

API: https://opendata.fmi.fi/meta?observableProperty=observation&param=t2m&language=eng

        - Observed wind
        - Observed cloudiness
        - Predicted wind
        - Predicted temperature

API: https://en.ilmatieteenlaitos.fi/open-data-manual ;

- The User can combine Both Data:
    - Weather + Road Maintenance (select Location and Time intervals)
    - Weather + Road Condition (select Location and Time intervals)
    - Select a location from a determined set (at least 5) OR Free coordinates
    - Select a Timeline (the timeline cannot be too short)

- The User can Save Data
    - Save Dataset on Traffic Messages and Weather on a given Day -> User can compare saved and current data/2 days
    - Save Preferences -> ex: 1 Type of Maintenance at one location

The development team will focus on implementing the mandatory functionalities requested by our client, first on the UI and then on the Back-End and data flow.

High-level description & Boundaries and Interfaces

Chosen technologies & libraries

As the team members are more familiar with the Java programming language, the application will be developed using Java rather than C++. Java 17 is the latest long-term support release as of now and delivers improvements to performance and security.

UI

JavaFX: For implementing the GUI.

JavaFX is a framework that developers use to construct rich client applications that work reliably across multiple platforms. It is mostly used for designing, building, testing, debugging, and deploying these applications. We will use it for project RoadCast as it is the most used collection for graphics and media implementation.

Back-End

Java 17 - Libraries and packages: GSON, JUnit4, org W3C Dom.

In order to handle JSON datasets from the APIs and to create JSON documents when saving data from our application, we will use the open-source Java library GSON to convert the JSON strings into Java objects and vice versa (serialization and deserialization).

In addition to JavaFX Document builders, the org W3C Dom package was used in the handling and parsing of XML data from the API. These give us Java representations of XML documents by providing interfaces for DOM.

Finally, if we were to test our application, check for any defects/bugs, and overall make our application more reliable for the client, we can use JUnit4. JUnit is an open-source testing framework for Java that is used to write and execute automated tests.

APIs - Digitraffic, Finnish Meteorological Institute

For getting the data on Finnish roads and weather conditions. Digitraffic offers real-time data on road, railway, and marine traffic, however, this project will focus on road traffic. For the weather forecast, the FMI offers services on weather, sea, air, and climate.

Management

GitLab: For version control

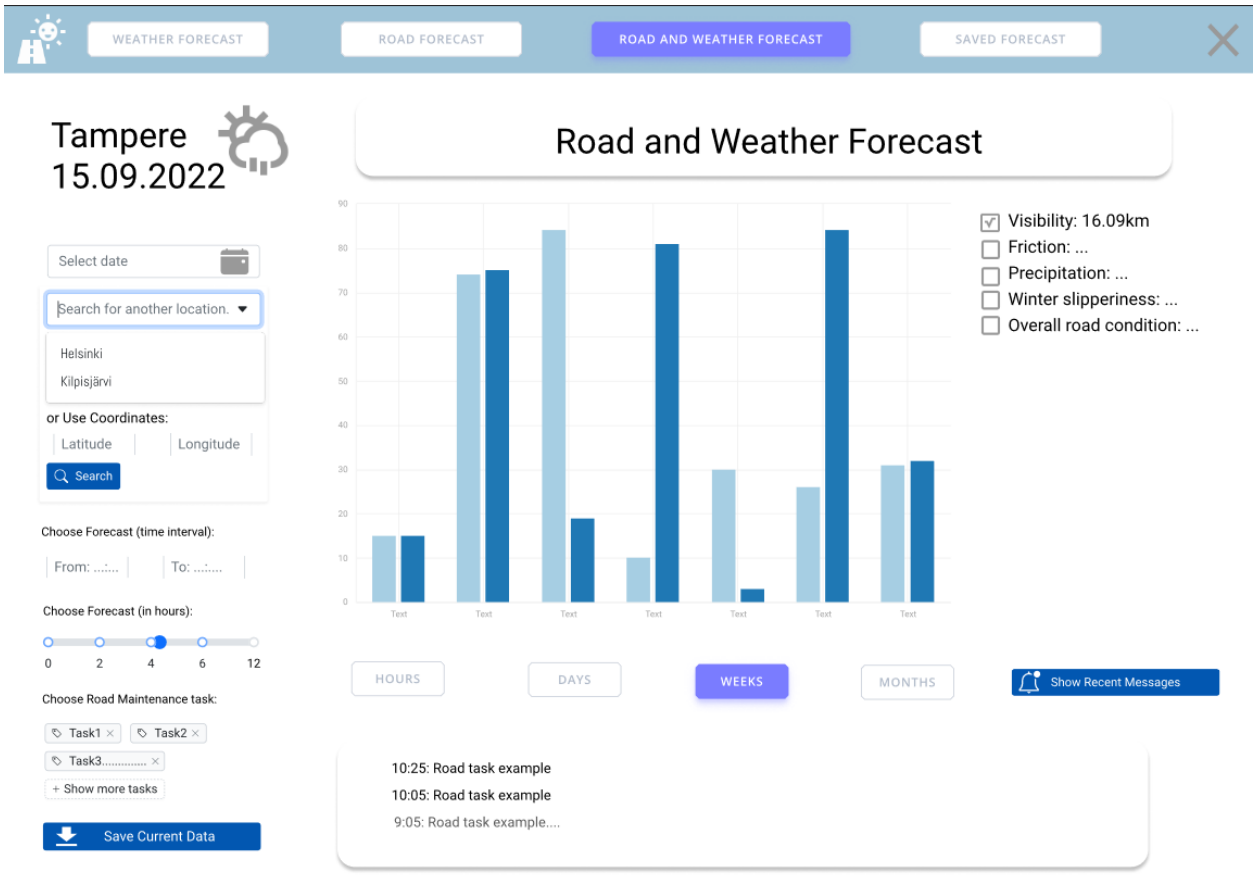
GitLab is the open repository given to us by our client for this project. It is a comprehensive DevOps platform that can help our development team to carry out project RoadMap (planning, managing source code, and monitoring). We can easily share our code and work on different branches to avoid conflicts when merging them later on.

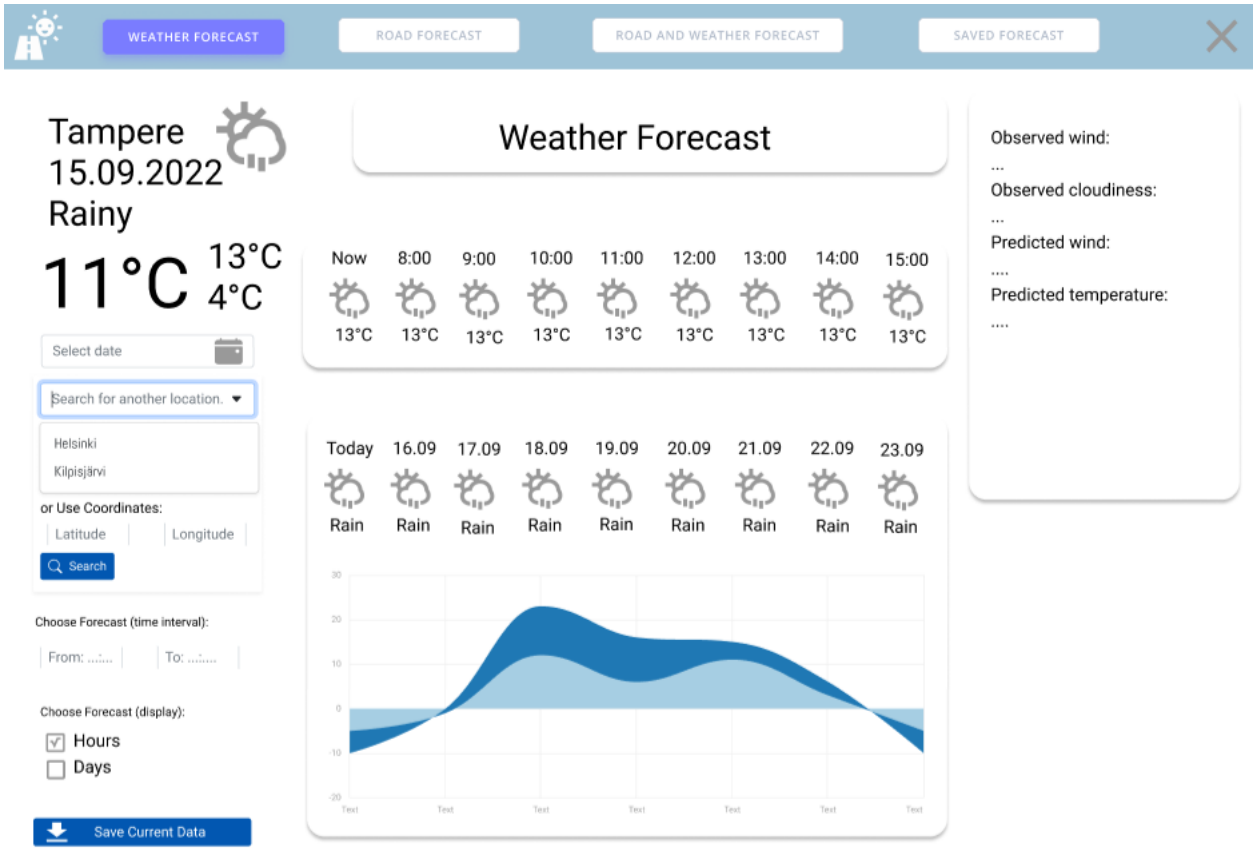
Java NetBeans: Maven

For project management. It is the tool the team members have the most experience using. Maven is a tool that is used for application builds, dependencies, and deployments based on POM (project object model). It will help us with criteria such as reusability and maintainability.

Figma Mockups

We first started a prototype sketch using Figma, a collaborative design tool used for creating designs for mobile and desktop interfaces.

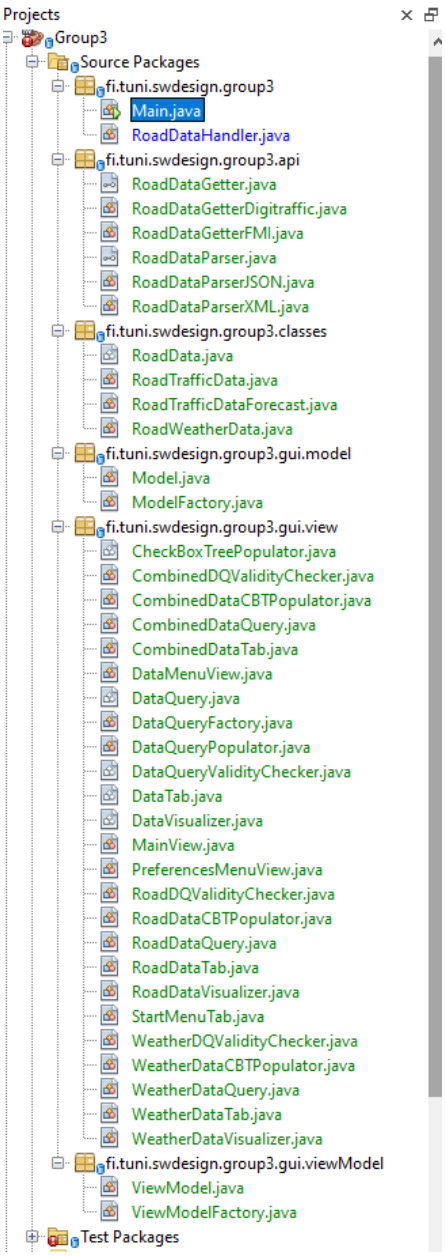




This first sketch can be found in our git repository as well as in our Figma team project:
https://course-gitlab.tuni.fi/comp-se-110-software-design_2022-2023/group-3.git
https://www.figma.com/file/sVrAbAr26B0PWMXTxgbjHk/Project---Prototype-sketch-(AD)?node-id=1%3A2

After learning more about how to design an UI in JavaNetBeans with JavaFX, we decided to switch to a simpler UI design and directly continue the prototype by coding it. The rest of this report will go further in detail about our design choices.

Architecture and Detailed Design



Current architecture of the code

MVVM pattern implementation

What is MVVM and Why did we choose this pattern?

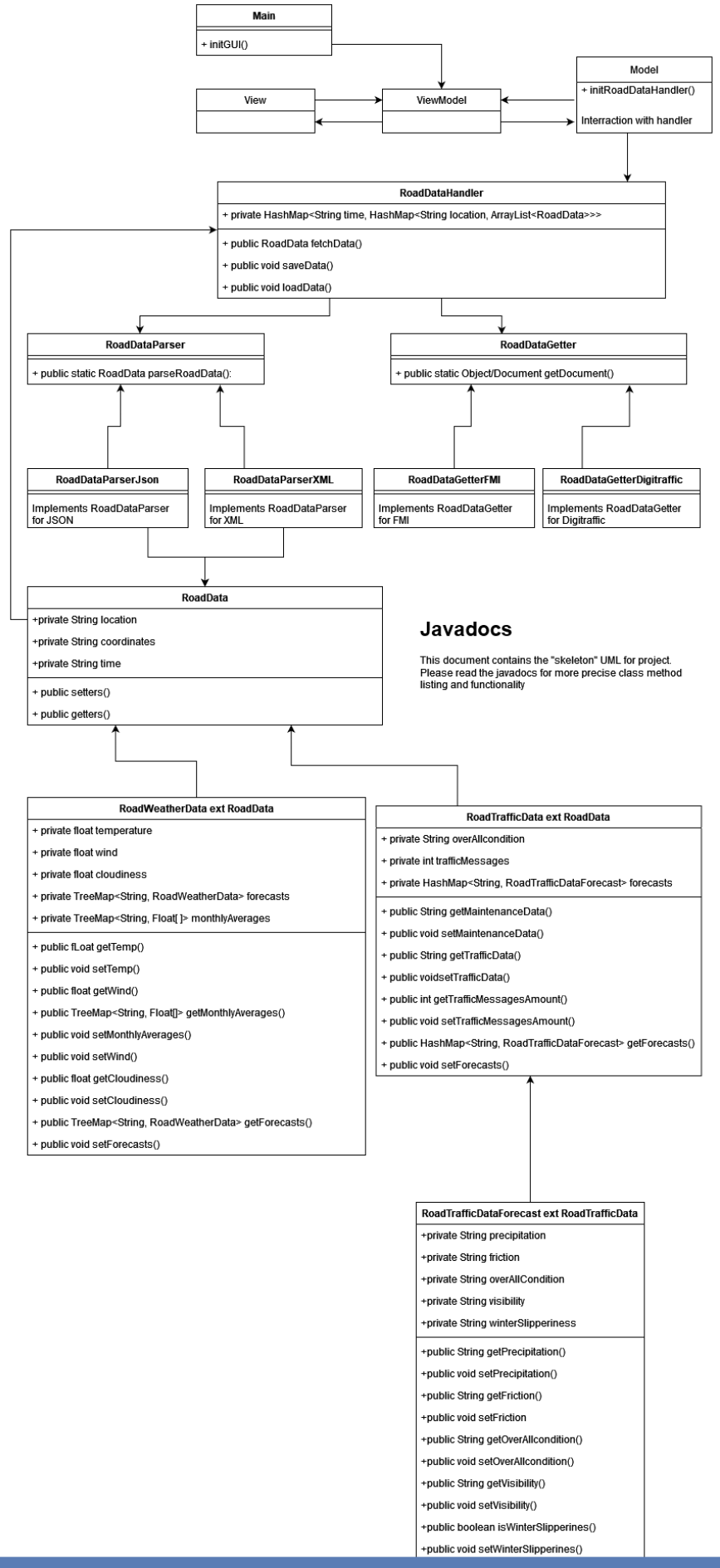
MVVM is a variant of MVC (Model-View-Controller).

The project will be implemented using the MVVM (Model-View-ViewModel) design pattern. The team members chose the pattern for its separation of UI and the rest of the software. This way team members can focus and work on GUI, ViewModel, and Model separately without depending on the progress of one another. Using MVVM will also make unit testing easier.

Factory Model

What is the Factory model and Why did we choose it?

In order to follow good practices and SOLID principles in design making, we can use design patterns to handle Object Creations. We chose to use the Factory model as it was the most suitable for our project. This pattern enables us to avoid tight coupling between Creator and Concrete products (thus following the SRP and OCP SOLID principles when creating and extending objects). The factory model is used mostly in the GUI in creating the view, viewmodel and model.



UML Diagrams UI/Back-End

MVVM implementation UML

The mother class is RoadDataHandler which acts as a facade.

API calls: RoadDataGetter fetches the data and its subclasses handle the different APIs.

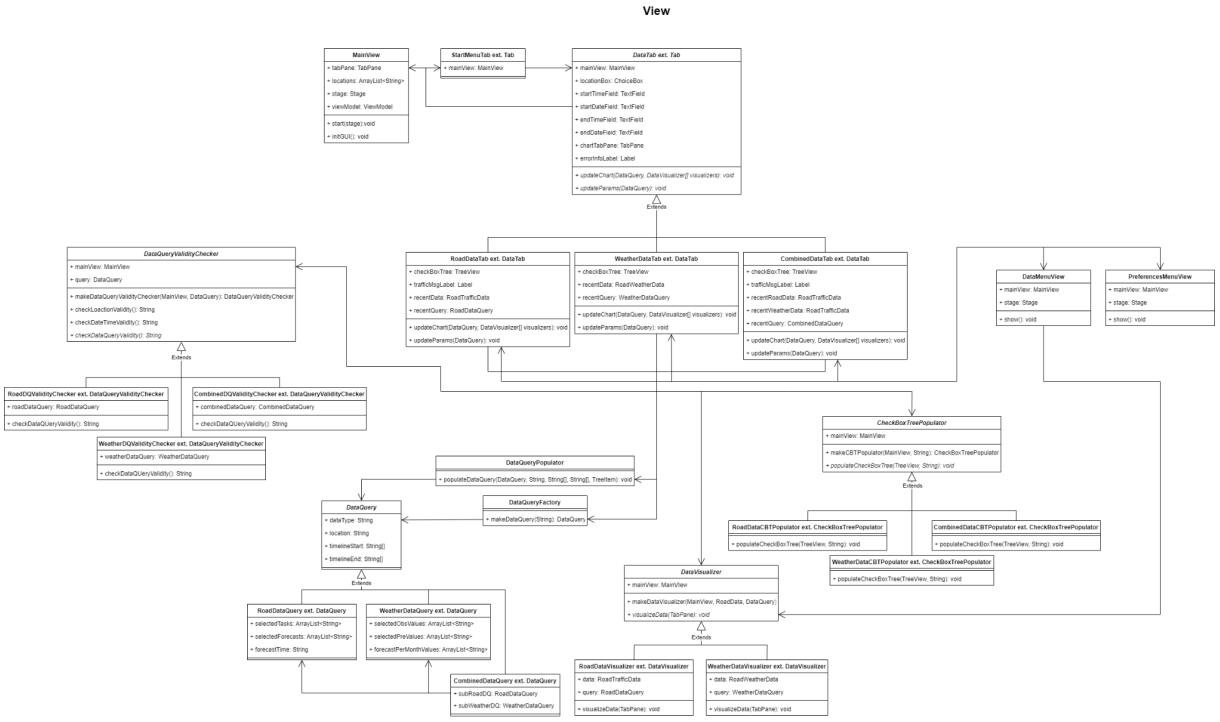
RoadDataParser handles the JSON and XML parsing for the data from the APIs and makes them as RoadData classes.

Concrete classes:

RoadWeatherData and RoadTrafficData extending from the abstract RoadData class (data fetching).

All dependencies are shown through

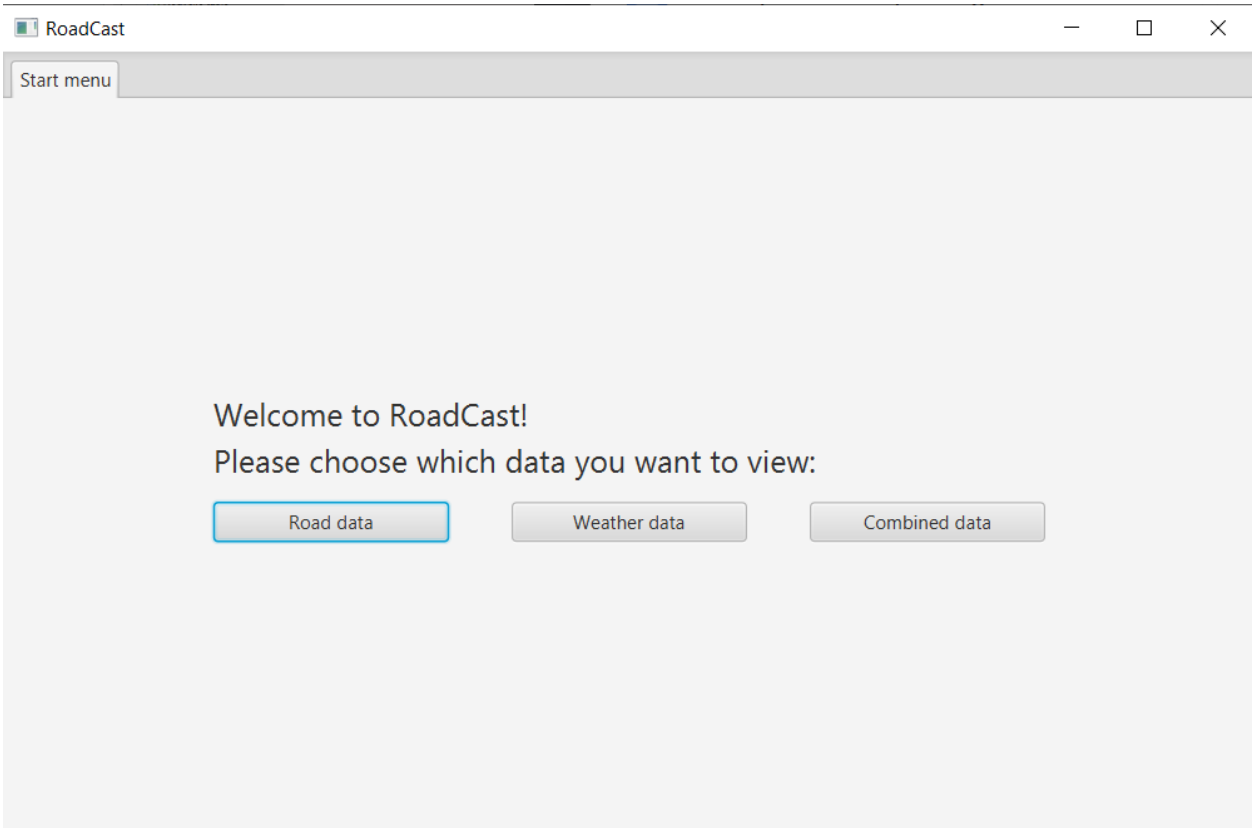
->



The UML diagram of the View component consists of the different classes and their associations. The MainView-class acts as a parent to all the other View- and Tab-classes and they are connected through the parent. All the other View- and Tab-classes' responsibilities are the events of their respective views. On top of that StartMenuTab initializes the Road-, Weather- and CombinedDataTab through DataTab, which they all inherit. The Road-, Weather- and CombinedDataTab-classes also initialize the Data- and PreferencesMenuViews. All the communication with other parts of the software happens through the MainView-class. Initialized Tabs will be stored in the TabPane of the MainView-class. CheckBoxTreePopulators are used by the different DataTabs to populate their respective CheckBoxTrees. DataVisualizers are used for visualizing the fetched data in the different DataTabs. DataQueries are used for storing the parameters for fetching the data. DataQueries have a Factory-class, Populator-class and DataQueryValidityCheckers, that are used for checking if the parameters stored in the DataQuery are valid.

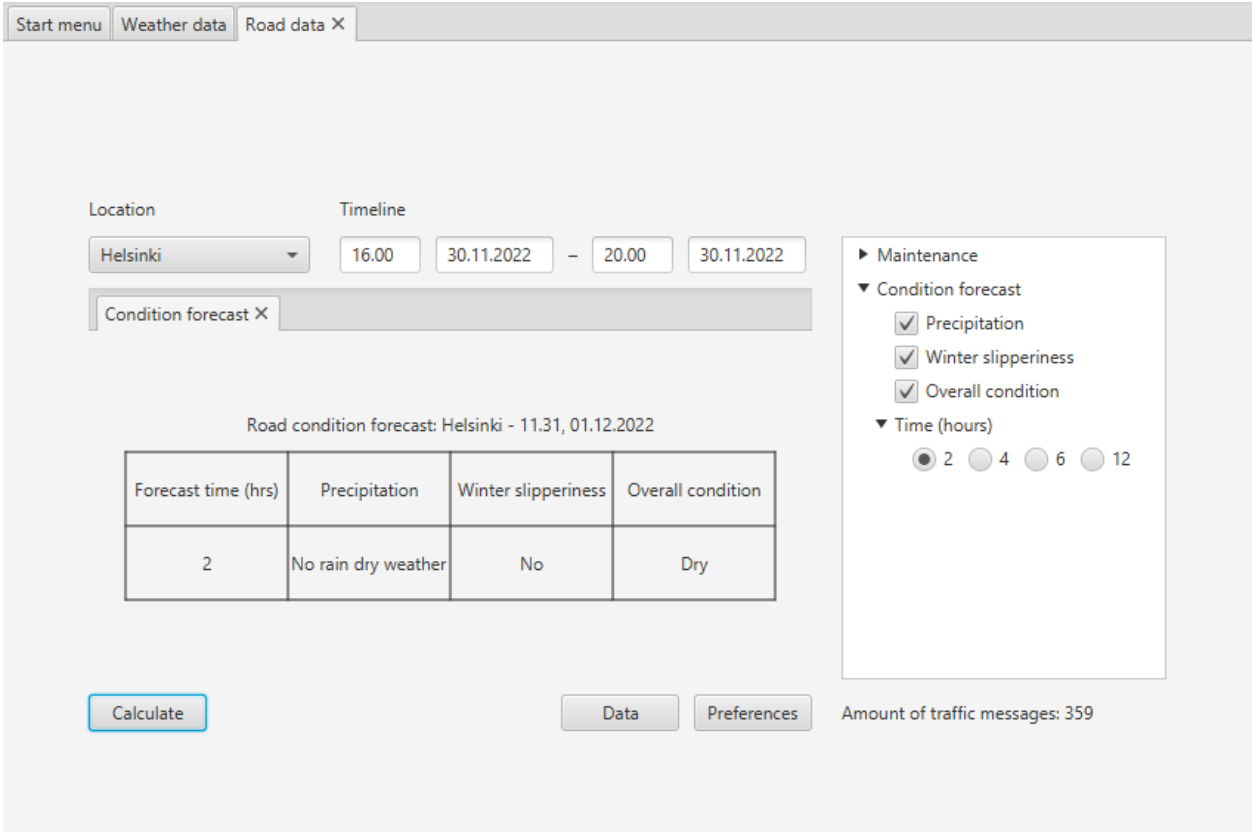
User Manual

Main menu



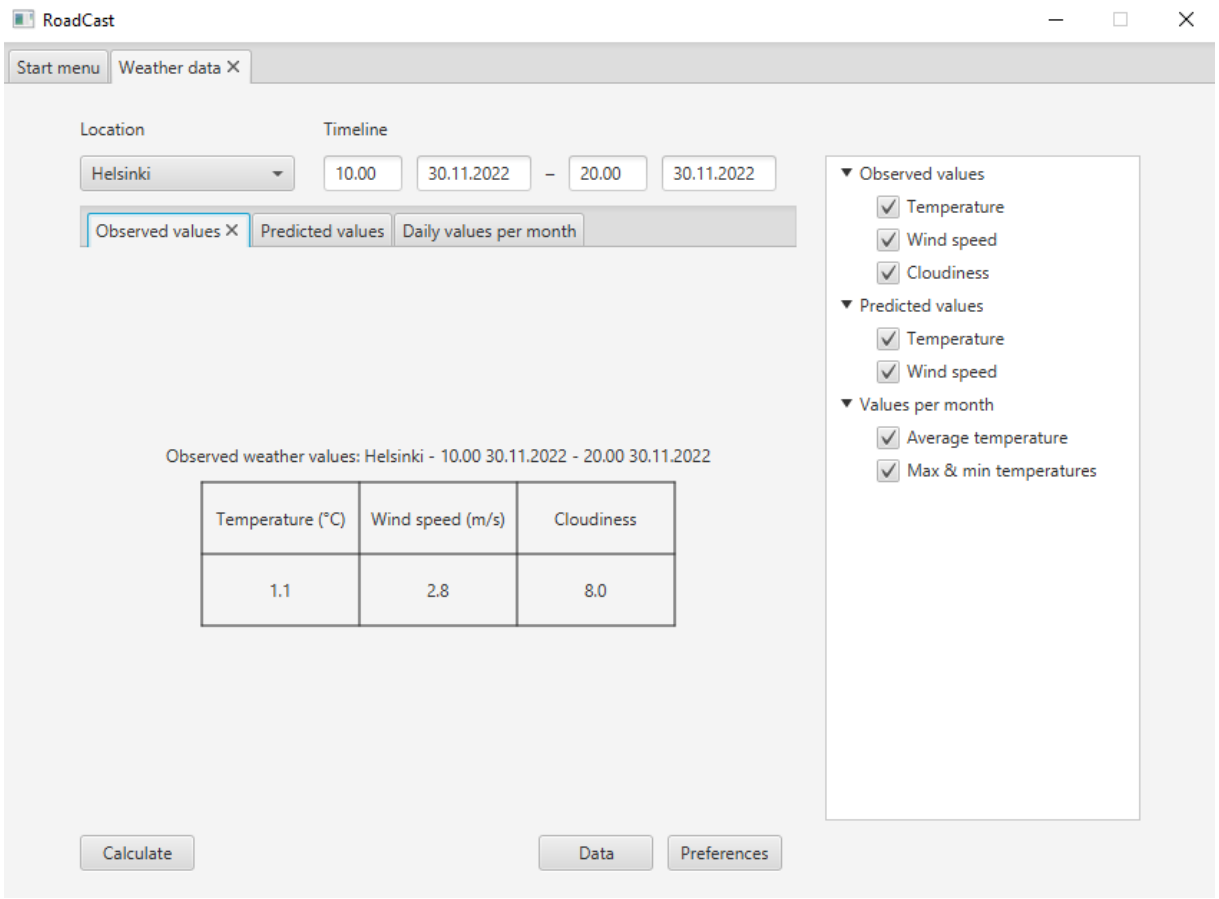
When first starting the software you will be greeted with the start menu. In the start menu, you can choose between three options: road, weather, and combined data. Choosing road data will create a new tab with tools and data visualization for data on Finnish roads. Choosing weather data will create a new tab with tools and data visualization for data on Finnish weather. Choosing combined data will create a new tab with tools and data visualization for both road and weather data simultaneously. The application can be closed by pressing the x on the top right corner.

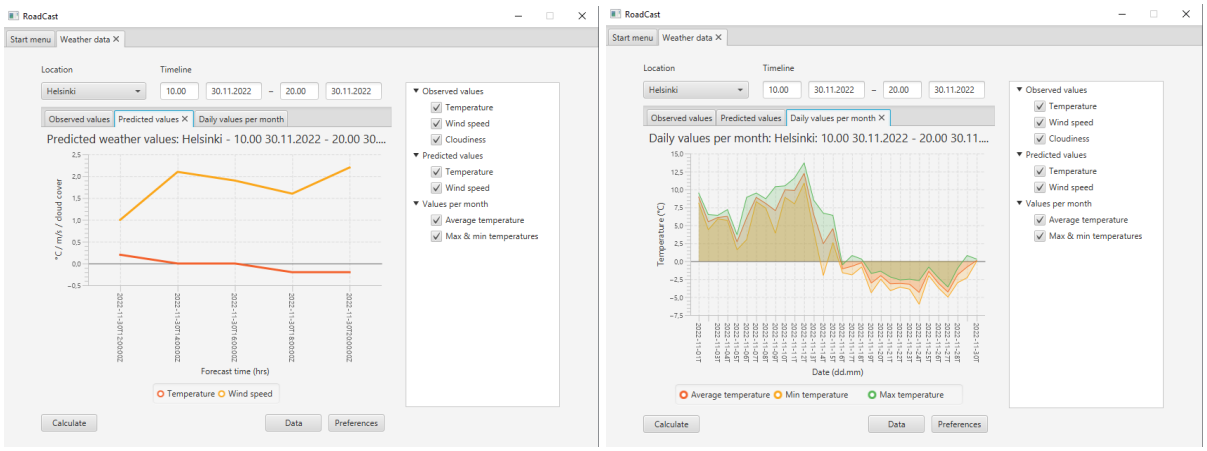
Road data options



In the road data tab you have to give parameters for the data visualization. First, in the top left corner, you have to choose the location for the data. There are currently five different cities in Finland between which you can choose. Next to the location dropdown menu, you have the timeline fields in which you have to give the starting time and date and ending time and date. The dates and times will have to be valid and typed in hh.mm and dd.mm.yyyy format. In the road data tab, the dates and times aren't checked and therefore adding them isn't necessary. On the right side of the window, you have the check box view where you can choose which data types you want to visualize. After choosing the parameters you can start the visualization by clicking the calculate button on the bottom left corner. If some of the parameters are invalid there will be a red text on the right side of the button explaining what went wrong. In the road data view on the bottom right corner, you also have the traffic messages label. The number on the label indicates how many traffic messages were found with the given parameters.

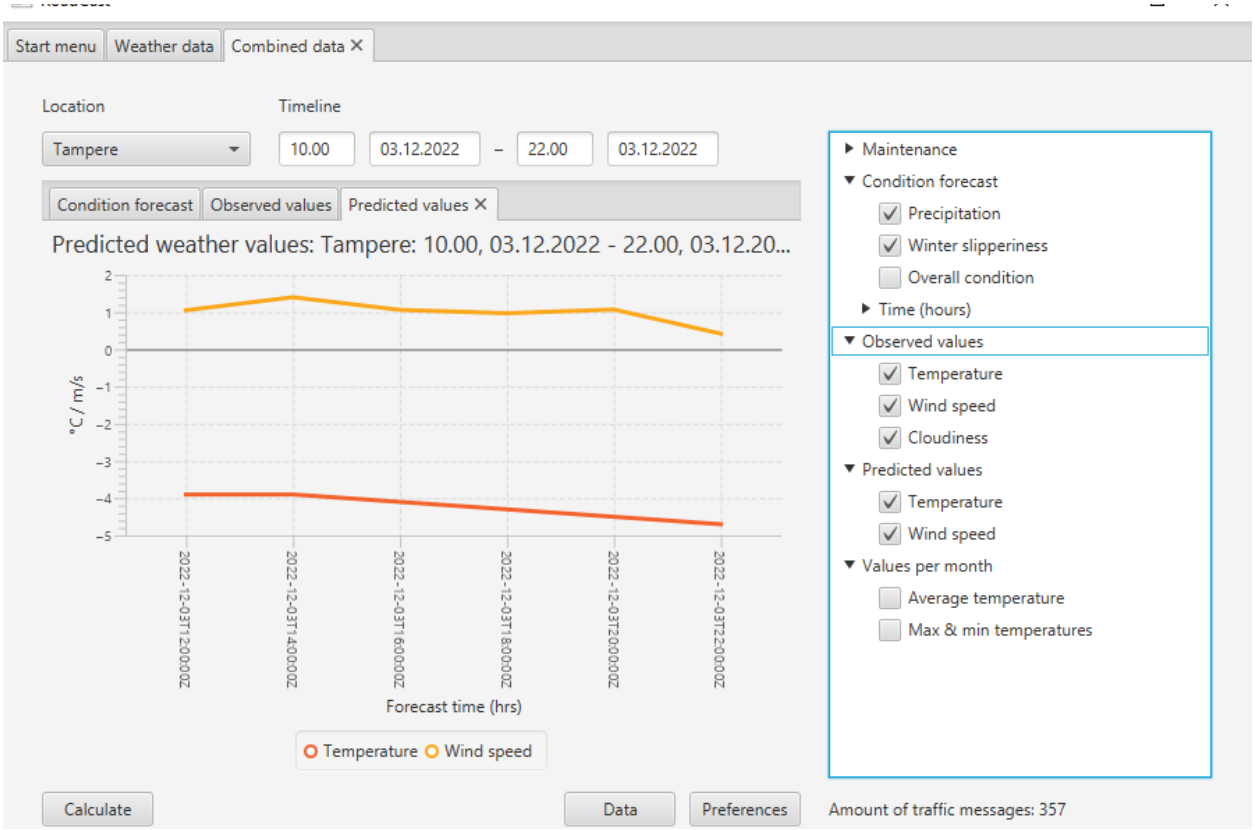
Weather data options





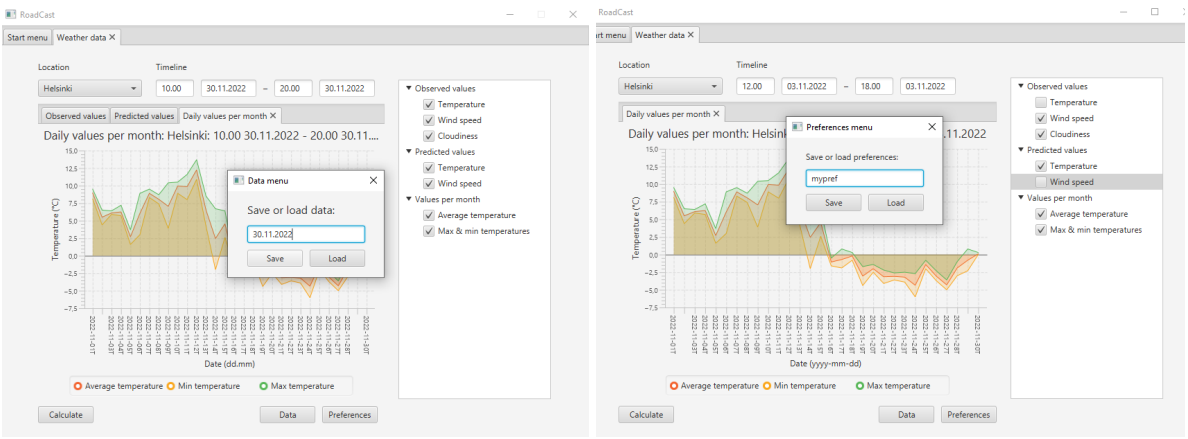
The weather data view is structured the same way as the road data view. The only difference is that there will be no traffic messages label and the checkbox view's content is related to weather data instead of road data.

Combined data options



The combined data is a combination of both the road data view and weather data view. The checkbox view now consists of both the road data view's parameters and the weather data view's parameters.

Other



Pressing the data or preferences buttons will create a new popup window for the data or preferences menus. In these menus, you can either choose to save or load data or preferences. Saving on the data menu will save the current data visualization into the data save file and loading will load the data of your choice to the currently selected tab. Saving on the preferences menu will save the current data visualization parameters to the preferences save file and loading will load the preferences of your choice to the currently selected tab. You cannot interact with the main window until you have closed the data or preferences menu by pressing the x on the top right corner.

How to run the project

Follow the guidelines from the repository's README:

README.md

Project RoadCast

COMP.SE.110 - Software Design - Group3
Group assignment

Description

A program for parsing weather and maintenance data from Fintraffic and Finnish Meteorological Institute public APIs and visualizing them with in a desktop application
More info for the assignment @ https://moodle.tuni.fi/pluginfile.php/2680373/mod_resource/content/0/group_assignment_specification.pdf.

Built with

- Java
- JavaFX

Frameworks used

- GSON
- JDOM2
- JUnit5

Getting started

To get your local copy up and running follow these steps

Prerequisites

Make sure you have Java JDK installed

1. Install Java SE and Maven in Windows by downloading them from

```
https://www.java.com/en/
https://maven.apache.org/download.cgi
```

2. Install Java SE in linux(Ubuntu/debian based) by

```
sudo apt-update
sudo apt install default-jdk
sudo apt install default-jre
sudo apt-get install maven
```

Installation

1. Open commandline and navigate to wanted folder and clone this repository

```
cd /path/to/wantedfolder
```

```
git clone git@course-gitlab.tuni.fi:comp-se-110-software-design_2022-2023/group-3.git nameyouwanttouse
```

2. if you have the project supplied in a zip file place it to a wanted folder and unzip it with

```
tar -xf archive.zip
```

Or use the GUI to unzip the project

3. Run the tests to check that everything is ok and compile the package

```
mvn compile
```

4. Start the program in the commandline by using

```
java -jar Group3/target/Group3-1.0.one-jar.jar
```

Or open the project in your chosen IDE compatible with Java and Maven and press run project

Self-evaluation

We have been able to stick to the original design pretty well, the only changes we made were small modifications to the View-component's structure. We've been able to implement all the features we wanted based on our original plan, with the exception of the traffic message -system, which proved to be more challenging than we thought. We think that our design corresponds to quality pretty well. We have been trying to use the principles and methods of good design taught in this course as much as we can. Javadoc comments and descriptions for the classes and methods were added to the project to add clarification.

What was implemented?

Basic functionality for the database (saving and getting data):

The fetched and parsed data gets converted to RoadData, from the object the GUI visualizes it and also has the possibility to save the data to a dataset. After saving to a dataset the object can be loaded for the purposes of visualizing the fetched data again. Users can also save preferences for the datavisualization so the user doesn't need to pick the preferences every time he / she gets data for visualization.

FMI data fetching and parsing:

- **RoadDataGetterFMI**

One main function *getDOMDocument()* opens a server connection and takes in parameters the query type (for example, observed or predicted), location coordinates, observation and/or prediction timestep, observation and/or prediction start and end times, and the actual parameters we want to fetch (for example "temperature" for the predicted temperature or "t2m", "TA_PT1H_AVG" for the observed temperature). It checks if the connection is actually made, and builds a document from which we can get nodes' elements.

A simple query check is made beforehand with *checkIfCorrectQuery()* in addition to the query validity checker classes that make a more thorough verification when the User makes the query through the UI.

-

- **RoadDataParserXML**

One main function *getDOMParsedDocument()* takes in parameter the w3cDocument returned by RoadDataGetterFMI's *getDOMDocument()* function as well as location name, coordinates and query start time values in order to create and return an actual RoadWeatherData object with the data from the start time queried by the User as well as the forecasted values following the start time (2, 4, 6, 12 hours after). The returned object will contain the queried weather data from the starting time until the end time or by default the next 12 hours. Each RoadWeatherData object's values are set with setWeatherData(), and the values we get (for example, the parameter name "temperature" and its value "2.3") are obtained by going through the different nodes' elements of the w3cDocument.

Another function, *getMonthlyTemperatureData()*, takes care of gathering the Average, Minimum, and Maximum for each day of a specific month at a specific location queried by the User. Contrary to the hourly observed values, as this function loads a lot more values by going through 28-31 days, the data is only fetched if a User queries it. The function returns a TreeMap as the data needs to be organized and the keys sorted by date (day 1 to day 28-31) to make the visualization easier.

Digitraffic data fetching and parsing:

- **RoadDataGetterDigitraffic**

The class is meant to get data from Digitraffic and get it into an usable JsonObject that can then be used with RoadDataParser. The class has multiple functions that each fetch different types of data. Some functions require coordinates and location names as parameters. There are functions for getting jsonObjects for active maintenance tasks, names of all maintenance tasks, Road conditions and traffic messages. All functions can throw errors if no data is found or if the coordinates are not valid. In those cases the functions will return null.

- **RoadDataParserJSON**

The class is meant to parse the data obtained from RoadDataGetterDigitraffic and turn it into data usable by other classes. The most used function is getRoadData() which creates a RoadTrafficData object with all possible information inside it. It calls multiple different functions from the class to achieve this and so requires multiple different parameters, including coordinates and JsonObjects obtained from RoadDataGetterDigitraffic. The

function was created with ease of use in mind. One function call gets you a complete dataset of RoadTrafficData. Another function worth mentioning is readMaintenanceTaskNames, which gathers all possible maintenance task names.

Visualization of fetched and parsed data:

Visualization of fetched data is done by DataVisualizers. After fetching the data, the data and the DataQuery will be stored in a new DataVisualizer, and visualizeData() will be called. The DataVisualizers then will create charts and tables according to the data and the DataQuery, and update the DataTab's chartTabPane, in which the data will be visualized in different tabs.

Local data saving and user preferences saving:

- **Saving data and preferences**

Saving of data and preferences is done with two functions. SaveDataBase() saves the actual dataset of a view possible including RoadTrafficData, RoadWeatherData, or both. It adds them to a Json file within an array. The saved dataset has an id which can be used when the user wants to load saved data. Two different datasets cannot be with the same name.

When saving a dataset you also need to save the preference that was used when viewing the dataset. Preferences change what you see when viewing the data and can be changed on the right side of the application. This happens automatically when the saveDataBase() function calls the savePreferences() function that saves the preference. The saved preferences are saved in a different Json file that also has an array with all the preferences inside. Preferences will also include an id so that the user can later load the wanted preference.

Preferences can also be saved separately. In that case the exact same thing happens, but only saving the preference and not the dataset. The user can have multiple preferences.

- **Loading data and preferences**

When loading data, the loadDataBase() function is called. The function looks through the datasets and sees if any matches the id the user has given. If they do, the data is fetched from the SavedData Json file and turned into a correct RoadData object. The dataset is then loaded for the user.

When loading preferences, the user also writes the id of the preference, and the function loadPreferences() will try to find a match to the id from the file SavedPreferences. Then the preference is converted into a DataQuery object with all the information. After that they are loaded into the view.

What was changed?

----- Old (Midterm submission) ------

The structure of the View-component was changed so that StartMenuView and the DataViews were replaced with corresponding Tab-classes. Also, DataTab-class, which the Road-, Weather- and CombinedDataTab-classes inherit, was added. DataTab-class also inherits the Tab-class of JavaFX. In the original structure the instances of different DataViews weren't stored anywhere, and the contents of the Views were also accessed through the tab they were in. By changing them into subclasses of Tab-class, different instances can be stored in the TabPane of the MainView, and it makes more sense overall. Also adding the class DataTab which the different DataTabs inherit, reduces redundant code. The new CheckBoxTreePopulator was also added to handle the population of the DataTabs' CheckBoxTrees. This way redundant code was reduced and the design follows the Single Responsibility Principle better.

Added a class for RoadTrafficDataForecast which extends the RoadTrafficData from Digitraffic. Forecasts have more information about the road than the observed situation, for example, slipperiness, friction, and precipitation, so it was logical to add a class for forecasts.

------ New -------

Split a big method in RoadDataGetterFMI to the Parser class so that the Getter is only fetching data while RoadDataParserXML actually takes care of creating and setting the RoadWeatherData objects.

Also split the RoadDataParserJSON functions so RoadDataGetterDigitraffic handles getting the data from Digitraffic and the parser class goes through the data and sets them to RoadTrafficData objects.

Updated the internal interfaces between the classes and modified the structure to follow MVVM pattern better.

Added the possibility to save and load datasets and preferences.

Future possible changes

In the future the program could use a document-oriented database such as MongoDB. After that, the separation of database handling now and the RoadDataHandler would be beneficial, so it will have separate classes for the database and the facade class controlling the fetching, parsing and providing the data for GUI visualization. By this, we would follow the SOLID principles “S” as in single responsibility better.

Also for the next steps, we would be adding a third data source and visualization for data from that source.

Refactoring the code to be faster and cleaner, and adding more unit tests for the code would be nice! 😀

What's next?

Following the design choices explained in this document, if we were to continue working on the project, we should take into account the elements mentioned in Self-evaluation - Future possible changes:

- Add tests for the functions and classes.
- Refactor some code to follow Functional Programming and Clean code principles.

An application will always continuously need to be maintained and updated in real-life production.

Work Division

Design Document v.	1.0	2.0	3.0
Amanda Dieuaide	Figma mockup	FMI API fetch/parse (daily, hourly)	FMI API fetch/parse (monthly + refactoring)
Aleksi Iso-Seppälä	Getting and parsing roadData from Digitraffic	Getting and parsing roadData from Digitraffic	Saving and loading of datasets and user preferences
Jukka Pelli	Overall structure and UML	RoadTrafficData, RoadWeatherData and RoadDataHandler i.e. database/facade class	RoadDataHandler, Model, viewModel i.e. the interfaces between internal classes. Javadocs & project structure.
Lauri Puoskari	View-component	View-component	View-component