

COMP.SE.110 - Software Design -

Group3

Amanda Dieuaide
Aleksi Iso-Seppälä
Jukka Pelli
Lauri Puoskari

Project RoadCast

October 28th, 2022

Road and Weather Condition
Forecasts Monitoring Application



Design Document (v.2)



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

Table of contents

Road and Weather Condition Forecasts Monitoring Application	1
Design Document (v.2)	1
Document version control	2
Table of contents	3
Introduction & Project Requirements	4
High-level description & Boundaries and Interfaces	6
Chosen technologies & libraries	6
UI	6
Back-End	6
Management	7
Figma Mockups	8
Architecture and Detailed Design	10
MVVM pattern implementation	10
Factory Model	10
UML Diagrams UI/Back-End	11
User Manual	13
Main menu	13
Road data options	14
Weather data options	15
Combined data options	16
Other	17
Self-evaluation	18
What was implemented?	18
What was changed?	19
Future possible changes	19
What's next?	20
Work Division	20

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¶m=t2m&language=eng>

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

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

5 - Design Document | Version 2.0 | Group3 - Project RoadCast | 28.10.2022

- 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: GSON, JUnit5, JDOM2

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).

JDOM2 will be used to handle and parse XML data from the APIs. It will give us Java representations of XML documents.

Finally, in order to test our application, check for any defects/bugs, and overall make our application more reliable for the client, we will use JUnit5. 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

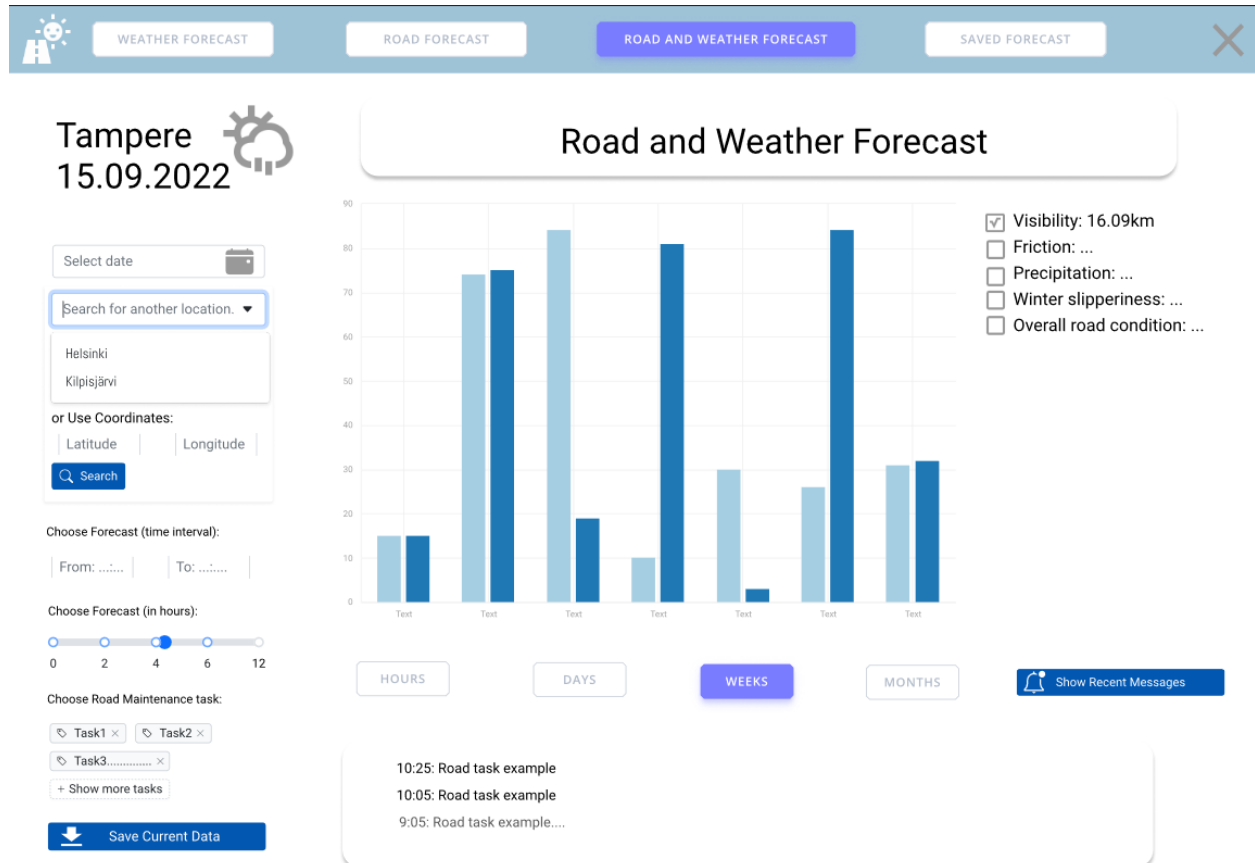
GitLab is the open repository given to us by our client for this project. It is a comprehensive DevOps platform that can help our development team to carry out project RoadMap (planning, managing source code, and monitoring). We can easily share our code and work on different branches to avoid conflicts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

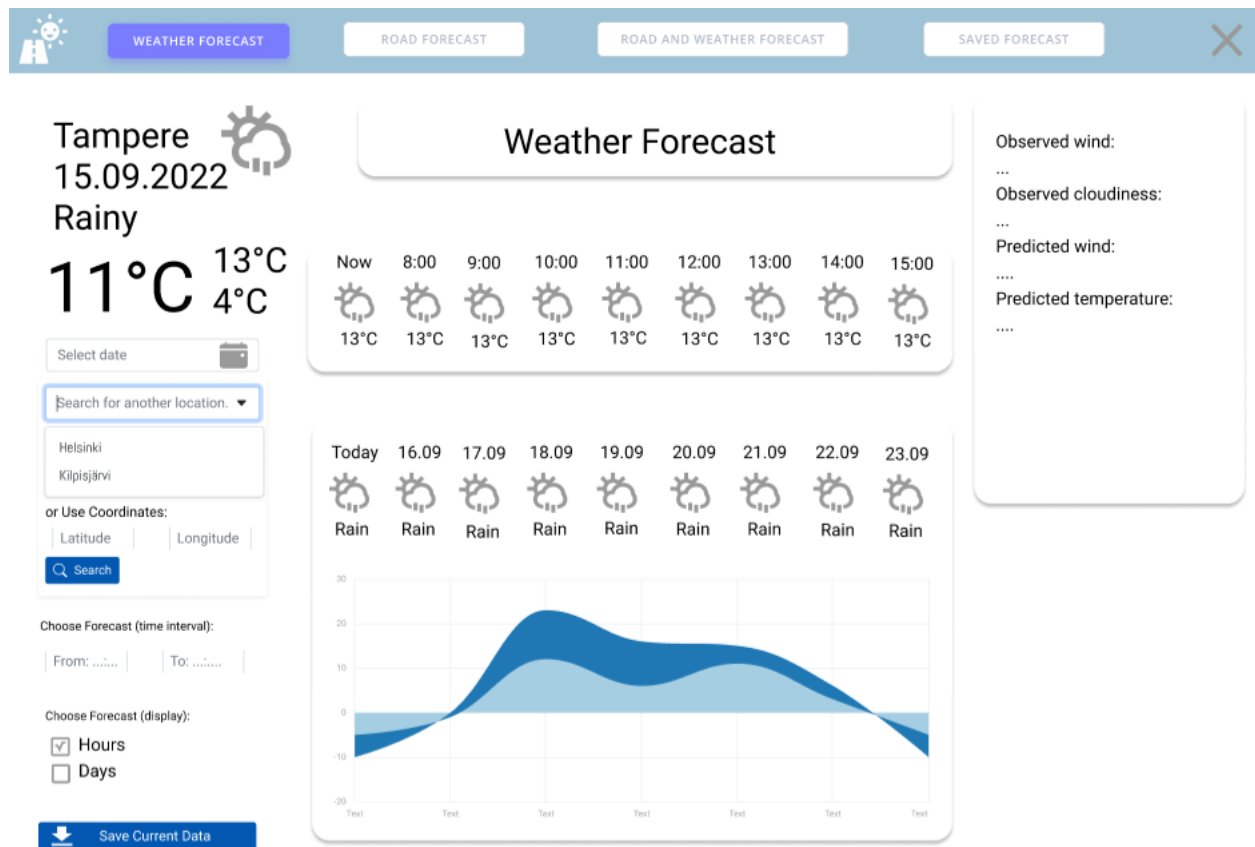
Java NetBeans: Maven

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:

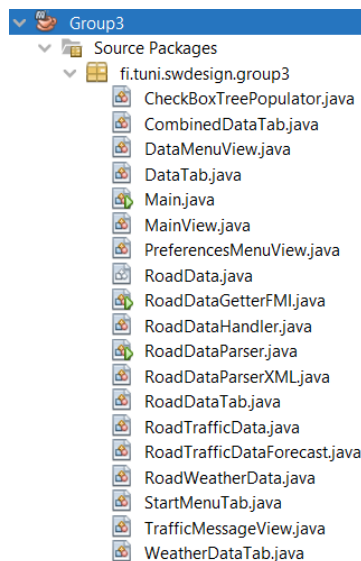
https://course-gitlab.tuni.fi/comp-se-110-software-design_2022-2023/group-3.git

[https://www.figma.com/file/sVrAbAr26B0PWMXTxgbjHk/Project--Prototype-sketch-\(AD\)?node-id=1%3A2](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

MVVM pattern implementation



Current architecture of the code (will be updated in the future).

What is MVVM and Why did we choose this pattern?

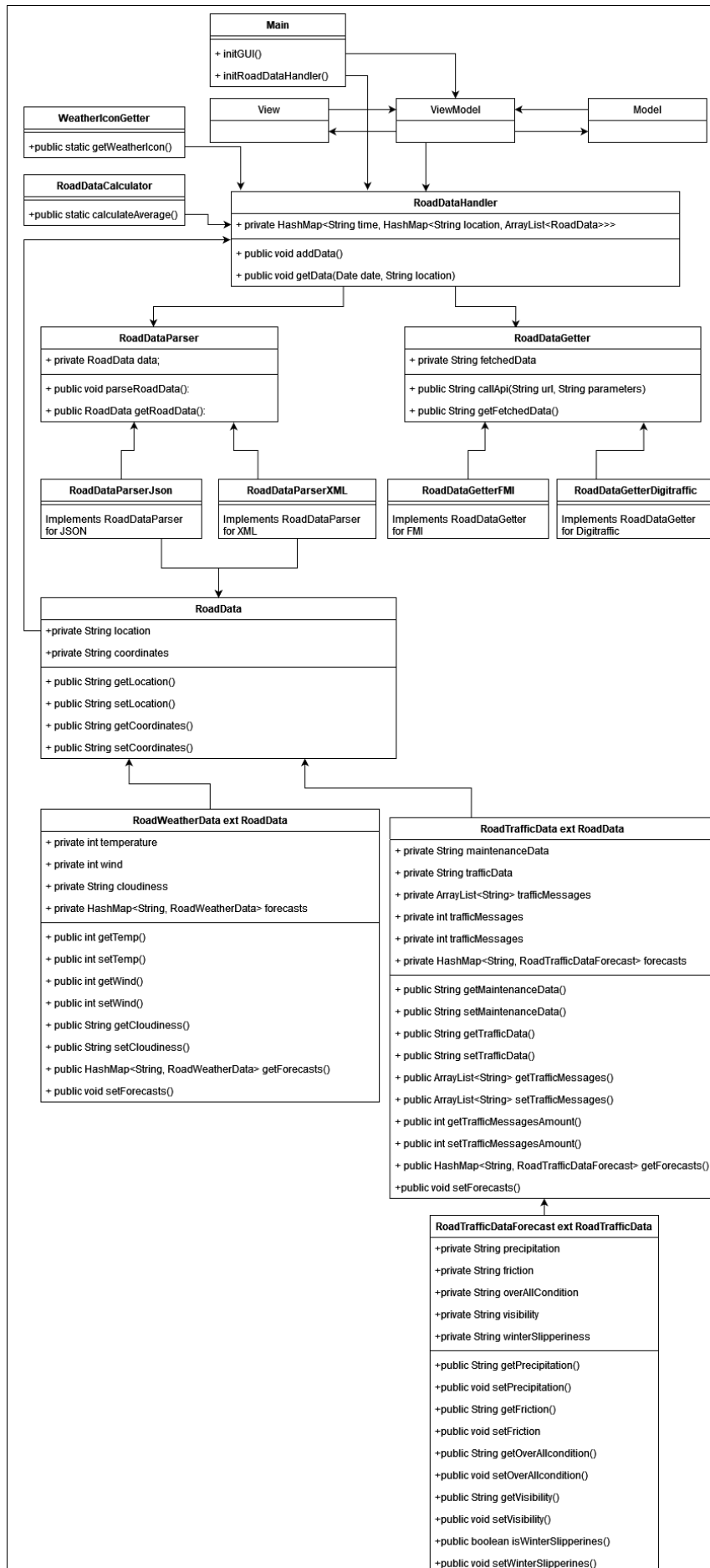
MVVM is a variant of MVC (Model-View-Controller).

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. For the moment, we chose to use the Factory model as we are not yet sure about the exact types and dependencies our code will have. 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).



UML Diagrams UI/Back-End

MVVM

implementation UML

The mother class is `RoadDataHandler` (data keeping and flow related).

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.

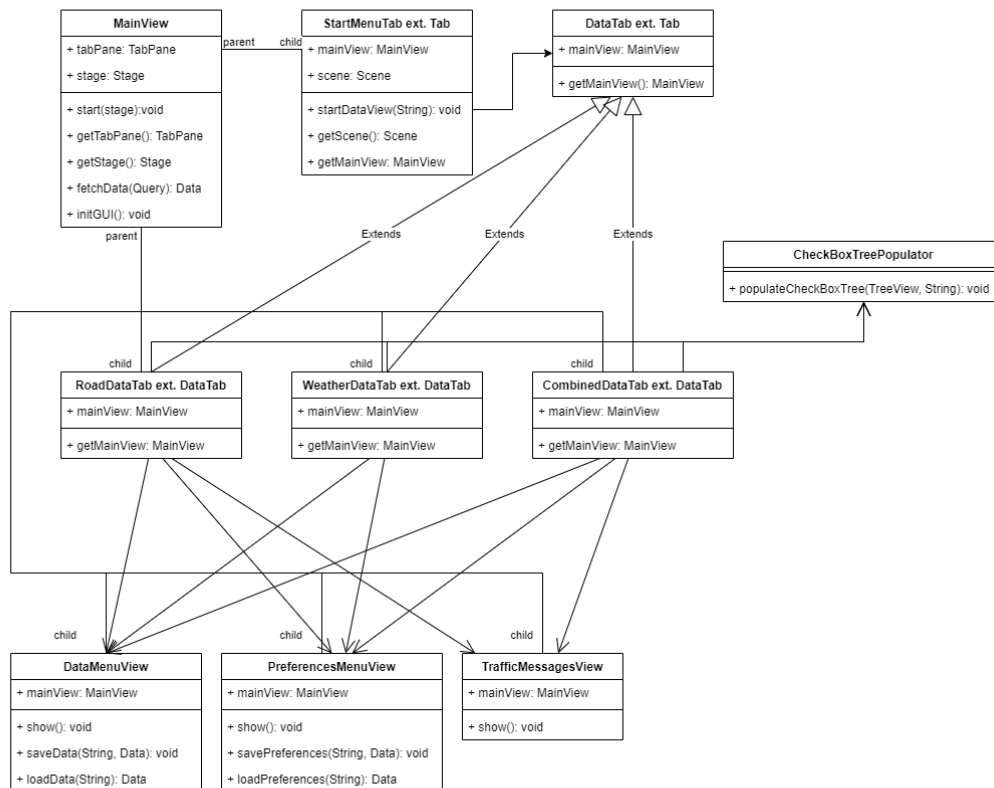
Concrete classes:

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

All dependencies are shown through

->

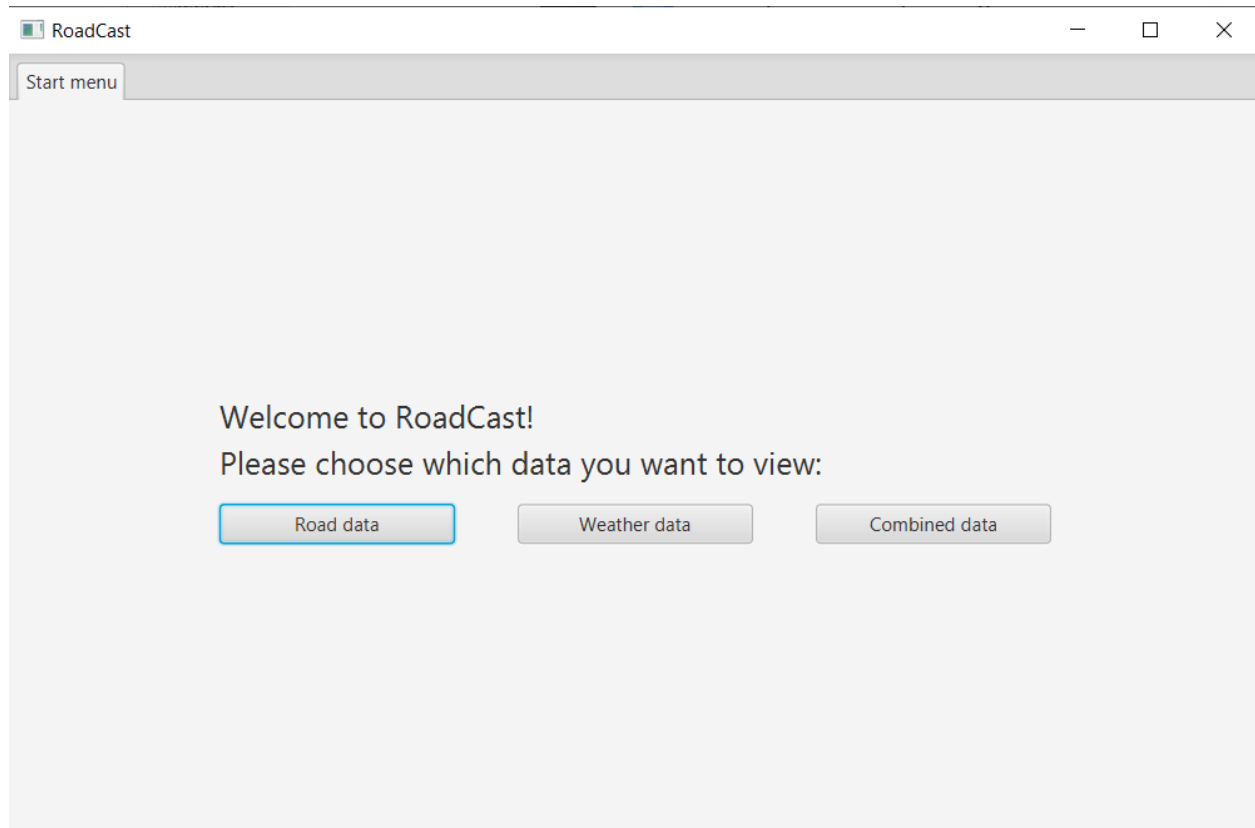
View



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**. **RoadDataView** and **CombinedDataView** also initialize the **TrafficMessagesView**. 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. **CheckBoxPopulator** is used by the different **DataTabs** to populate their respective **CheckBoxTrees**.

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

The screenshot shows the 'RoadCast' application window with the 'Road data' tab selected. The interface is divided into several sections:

- Location:** A dropdown menu currently set to 'Tampere'.
- Timeline:** Fields for starting time (13.00), starting date (20.09.2022), ending time (14.00), and ending date (20.09.2022).
- Grid:** A large empty grid with a y-axis from 0 to 100 and an x-axis from 0 to 110.
- Right Panel:**
 - Maintenance:** Five checkboxes for 'Maintenance task 1' through 'Maintenance task 5'. Tasks 1 and 2 are checked.
 - Condition forecast:** Five checkboxes for 'Visibility', 'Friction', 'Precipitation', 'Winter slipperiness', and 'Overall'. All are currently unchecked.
 - Time (hours):** A slider set to 4, with a range from 2 to 12.
- Bottom:**
 - A 'Calculate' button is disabled, with a red error message 'Invalid parameters!' next to it.
 - Buttons for 'Data' and 'Preferences'.
 - A 'Traffic messages (5)' button.

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. 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 button. The number on the button indicates how many traffic messages were found with the given parameters. Clicking the button will open the traffic messages view, which we will come to later.

Weather data options

The screenshot shows the 'RoadCast' application window with the 'Weather data X' tab selected. The interface includes a 'Location' dropdown set to 'Helsinki' and a 'Timeline' section with input fields for 'hh.mm', 'dd.mm.yyyy', and another 'hh.mm', 'dd.mm.yyyy' pair separated by a minus sign. A large grid area is present for data visualization. On the right, a sidebar contains checkboxes for 'Temperature' (Average daily/month, Max & min values/month, Observed values, Predicted values), 'Wind' (Observed values, Predicted values), and 'Cloudiness' (Observed values). At the bottom, there are buttons for 'Calculate', 'Data', and 'Preferences'. A red error message 'Invalid parameters!' is displayed next to the 'Calculate' button.

Location: Helsinki

Timeline: hh.mm dd.mm.yyyy - hh.mm dd.mm.yyyy

Calculate Invalid parameters! Data Preferences

▼ Temperature

- ☒ Average daily/month
- ☐ Max & min values/month
- ☐ Observed values
- ☐ Predicted values

▼ Wind

- ☐ Observed values
- ☐ Predicted values

▼ Cloudiness

- ☐ Observed values

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 button and the checkbox view's content is related to weather data instead of road data.

Combined data options

RoadCast

Start menu Road data Weather data Combined data X

Location

Timeline

hh.mm dd.mm.yyyy - hh.mm dd.mm.yyyy

100
90
80
70
60
50
40
30
20
10
0

0 10 20 30 40 50 60 70 80 90 100 110

Calculate Invalid parameters! Data Preferences Traffic messages (5)

▼ Maintenance

- ☐ Maintenance task 1
- ☐ Maintenance task 2
- ☐ Maintenance task 3
- ☐ Maintenance task 4
- ☐ Maintenance task 5

► Condition forecast

▼ Temperature

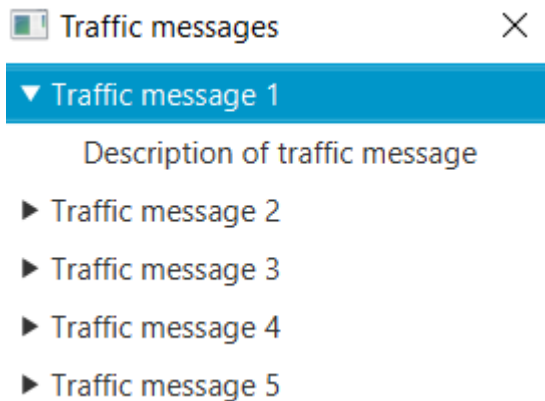
- ☐ Average daily/month
- ☐ Max & min values/month
- ☐ Observed values
- ☐ Predicted values

► Wind

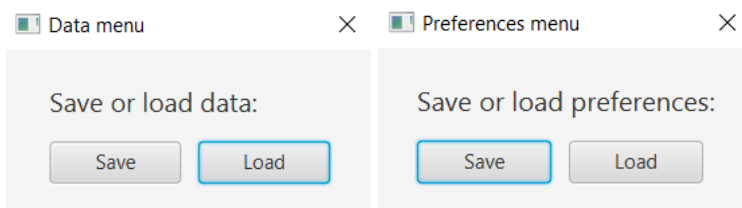
► Cloudiness

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 traffic messages button will create a new popup window that will show all the current traffic messages. You can click on the messages to read the description related to the message. With the traffic messages window open you can still interact with the main window and you can create multiple traffic messages windows. You can close the traffic messages window by pressing the x on the top right corner.



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.

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.

What was implemented?

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

The fetched and parsed data gets converted to RoadData, which are then stored to a HashMap. From the HashMap the GUI has the possibility to get saved data for the purposes of visualizing the fetched data. Methods are *addData()* and *getRoadData()*.

Basic fetching and parsing from the FMI API: RoadDataGetterFMI class

One main function *getDOMParsedDocument()* opens a server connection and takes in parameters the query type, 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. The information queried is then printed by the *readWeatherData()* function. For now, this basic implementation works with the three example queries from the API guide and prints out the first observed and/or predicted values from the API.

Digitraffic data fetching and parsing:

Implemented private functions for getting and parsing the traffic messages. *readTrafficMessages()* returns the number of messages based on the situation type. Implemented a smoother way to parse road conditions. They now can be combined into a single RoadTrafficData object which can then be used by other classes. Most importantly implemented the function *getRoadData()*, which gets all road data for given coordinates

(meaning maintenance tasks, road conditions, and traffic messages), and combines them to a single RoadTrafficData object. This makes using the roadData smooth and easy by having only a single function call for getting all the needed data for the database and for the visual presentation.

What was changed?

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.

Future possible changes

Cleaner separation of code following the SOLID principles: splitting getting and parsing from the APIs to different classes; completing the FMI API fetching and parsing by using a RoadWeatherData class to get the data; Refactoring the classes to have static methods so there will be no need to make objects for them.

Database structure change:

HashMap<Time, HashMap<Location, Data> -> HashMap<Location, HashMap<Time, Data>.
Because there are less locations used than different times. So it would be logical to use locations as the first separation.

Separate the Database and the RoadDataHandler, 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.

What's next?

Following the design choices explained in this document, we will continue the implementation of the functionalities (mainly on the visualization of the data gathered by the handlers from the APIs):

- Implement visualization of fetched and parsed data.
- Implement local data saving and user preferences saving.
- Add tests for the functions and classes.
- Add comments to the code, descriptions for the classes and make a Javadoc for the project.
- Optional (according to the time left): show not only the number of traffic messages but also the actual messages pinpointed to the actual locations.

The team will check applicability based on the implemented functions and work on refactoring the Software to improve and optimize the application.

This design document will be updated for the final meeting with our client (2.12.2022).

Work Division

Design Document versions	1.0	2.0
Amanda Dieuaide	Figma mockup	FMI API fetch/parse
Aleksi Iso-Seppälä	Getting and parsing roadData from Digitraffic	Getting and parsing roadData from Digitraffic
Jukka Pelli	Overall structure and UML	RoadTrafficData, RoadWeatherData and RoadDataHandler i.e. database/facade class
Lauri Puoskari	View-component	View-component