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 is made with Figma.

GUI

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. 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.

Functionality and how it is implemented

Our team decided on the MVC model which can be seen from the class diagram. MainWindow uses the controller to communicate with the APIs through model which uses two networker class. We decided to use two different networkers as the digitraffic API and FMI API are quite different. 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 parseing the model gives stored datasets to the createGraph class which will do the necessary calculations and forms a wanted graph. Then the graphs and calculations will be sent to mainWindow through model 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.

Controller class is the one that will be the controller from the MVC model. It will convey the data demands that the view, in this case mainWindow, has to the model but this is only a one way interaction. The controller won't convey the data that the model gives to the view.

Model in this case is everything that is happening in the background. Something that the user won't see. It will make API calls to get the wanted data in a required form and give it straight to the view without controller acting as a middleman. Each class that isn't controller or mainView is part of the model.

jsonNetworker is the class that will handle getting datas from the digitraffic API. After that it will return it without parsing the data.

jsonNetworker is the class that will handle getting datas from the FMI API. After that it will return it without parsing the data.

XmlParser is the class that handles parsing the data that is in xml form. It implements the xmlParser interface. In this case the data from Ilmatieteenlaitos. After it has parsed the data. It will create a new xmlDataStorer object and return that. **Still in works and might change.**

XmlDataStorer stores the parsed xml data and it's used by xmlParser. **Still in works and might change.**

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.

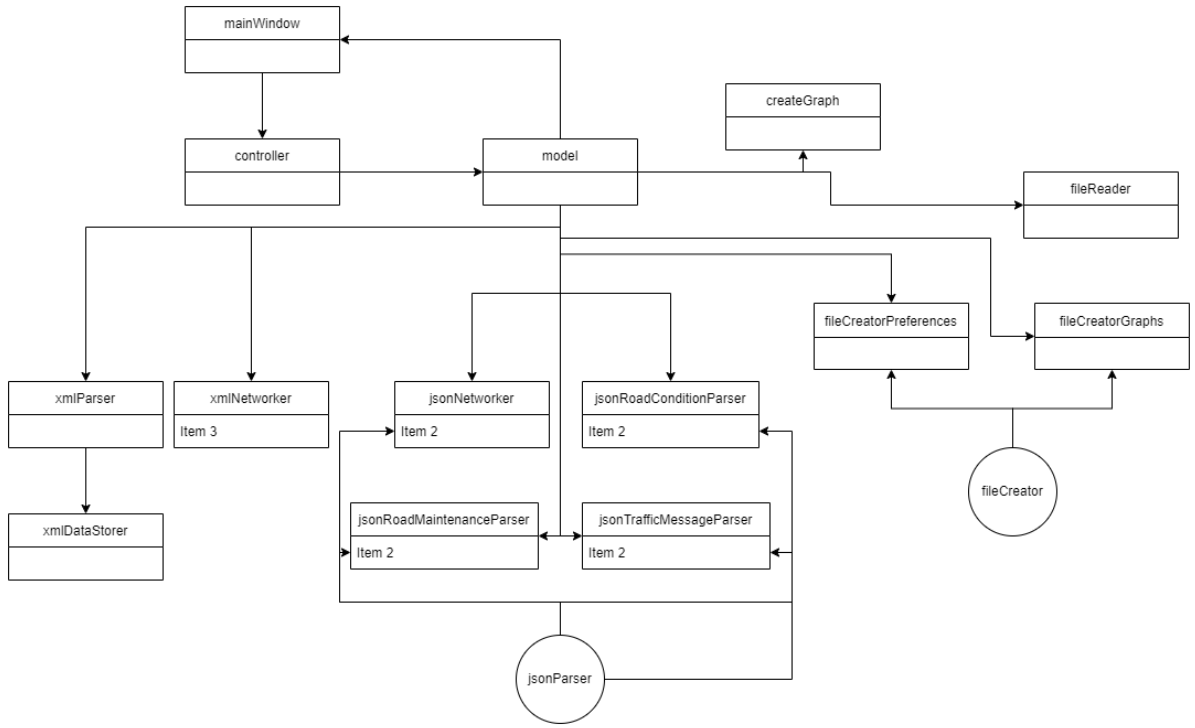
FileReader class is used to read saved files, which have the user preferences and saved graphs and gives the data to the view.

FileCreatorPreferences creates a file that stores the user preferences when it comes to the user interface settings in JSON format. Created file will be stored locally. This will implement the **fileCreator** interface

FileCreatorGraphs does the same as fileCreatorPreferences but it stores graphs that the user has created. These are also stored in JSON format. This will also implement the fileCreator interface.

CreateGraph class creates the graphs that the user request and are returned to the view.

jsonParser and **fileCreator** are interfaces that different classes implement.



Picture 1 Class diagram

Decision making process

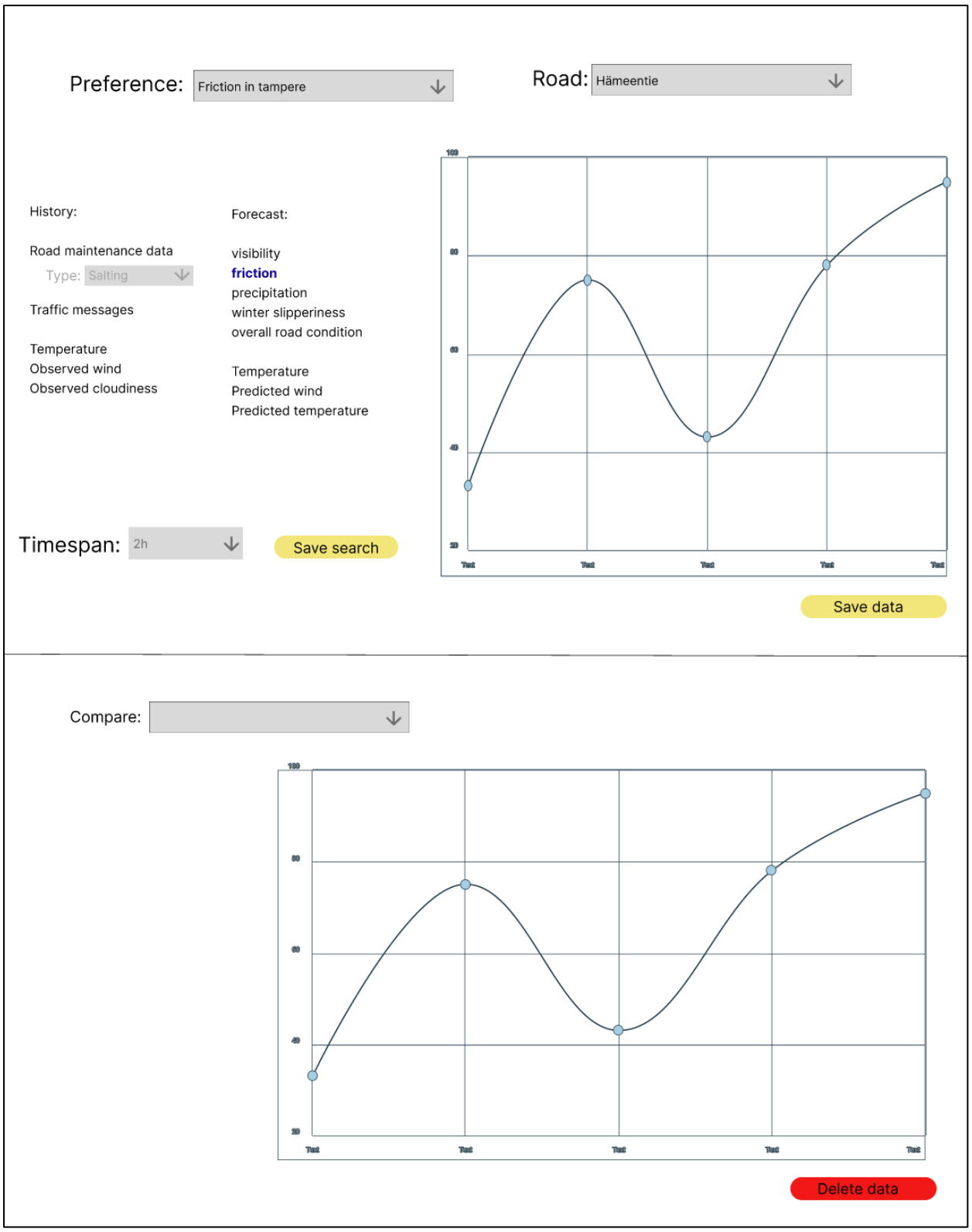
Before first submission

The mainlines for the prototype came from trying to decipher the given instructions in group meetings. We decided to limit the amount of windows and transitions the program has, to create a more concise and simple-to-use product. Hence why almost all of the visual components are also visible on the main window instead of being hidden behind other components. Decisions relating to visualisation of graphs, saving of data and other classes were made with mostly ease of programming in mind.

Mid-term decisions

After first submission we realised that the APIs and the data formats of FMI and digitraffic are so different that it was clear that we needed two separate networker classes, one for each API. We also realised that the files that are returned from APIs are so different that it was more rational to a specific parser for each file type.

We also decided to remove the third window from the application (picture 2) because we deemed it as unnecessary from the user perspective and useless. We can achieve the requirements with two windows so the third was unnecessary.



Picture 2 Figma prototype with the changes made in mid-term submission