Roadwatch Documentation

H292001 Aapo Kärki

H293115 Matias Aitolahti

H292259 Onni Vitikainen

50121133 Ronja Lipsonen

Program structure diagrams

Below is a diagram of the general structure of our program. Figure 1 contains a big-picture view of the classes and their relations to each other.

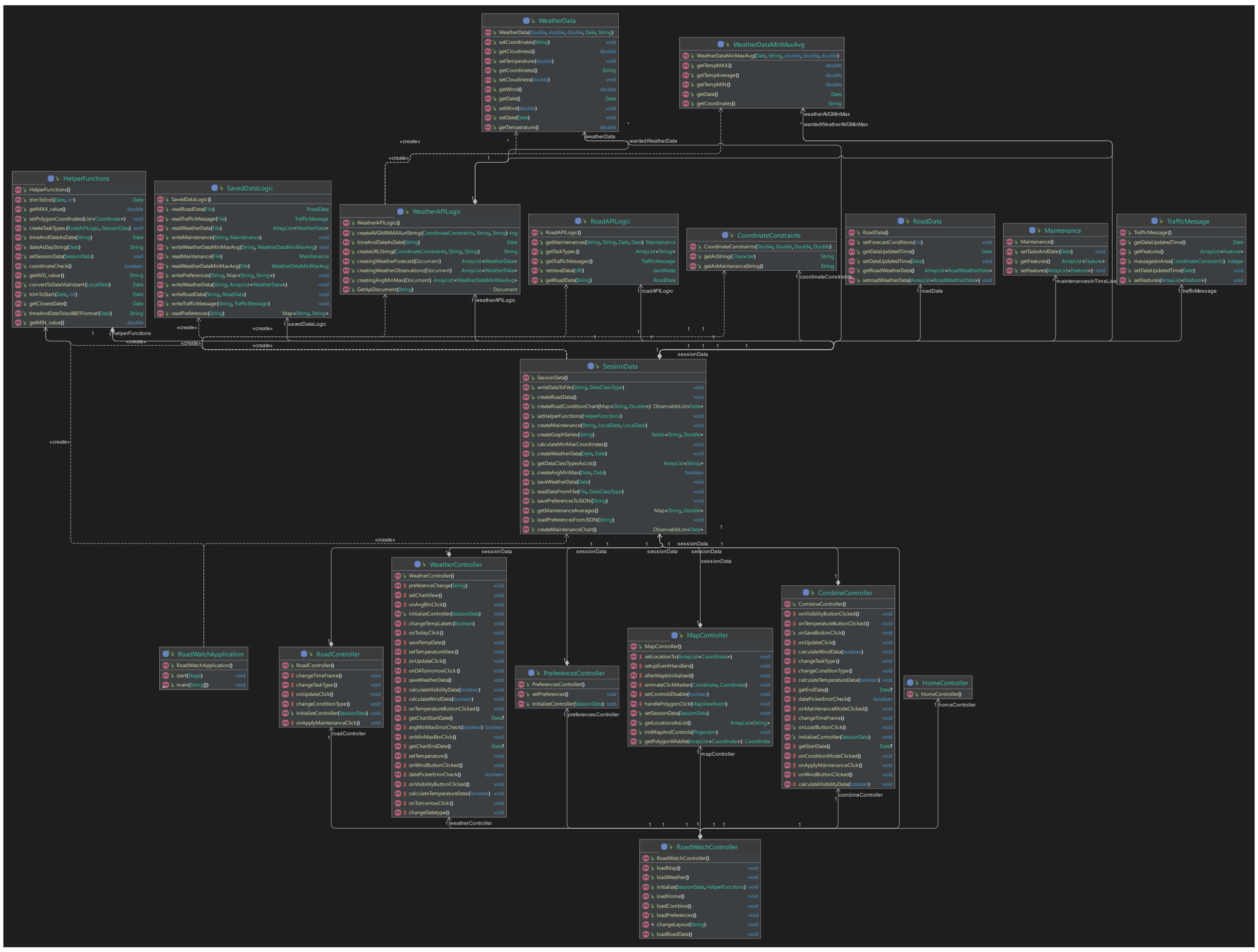

Figure 1: Program structure diagram

General description of the application

The structure of the class definitions can be roughly divided into three sections:

1. API input
2. Data/file handling
3. The UI-components

Starting from the API side, there are APILogic classes for each separate API. Their purpose is to do the actual API calls and instantiate new objects of the data classes (here WeatherData for FMI and RoadData/Maintenance/TrafficMessage for the Digitraffic) from the data received from the API's.

These data classes are used for objects to store and organize the data from the API's. The SessionData-class stores the relevant data objects based on the current location. SessionData is also responsible for many of the data handling tasks related to those objects, however some of the smaller, more menial functions are divided into a separate HelperFunctions class. This helps keep clutter away from SessionData.

Each WeatherData and RoadData are unique by their coordinates. Calling for a new report on different coordinates will result in a new set of data objects. Data classes of previous coordinates are overwritten when creating a new request to the respective API, apart from TrafficMessage. Since the Digitraffic API provides a convenient (and relatively fast) way to fetch all traffic messages for the entirety of Finland, there is no need to create multiple requests.

Saved data and file handling is the responsibility of the SavedDataLogic –class. It saves data from the SessionData class into a file that can be used later in another session. Accessing and importing the saved data is also done in SavedDataLogic. The data format is JSNO, and the data is presumed to be formatted in a manner similar to the API call responses.

The UI components are roughly divided into controller classes and FXML files. The main component class of the program is RoadWatchApplication, and it calls the main UI component, roadwatch.fxml. Each FXML file has its own controller. The FXML files are named after the main views in the program, (Home, Weather, Quickview, Combine, Preferences, Roaddata) and the Map (which is used in many views).

The Map is implemented from the open-source map library https://github.com/sothawo/mapjfx to make navigating the roads of Finland much friendlier to the user. The main two ways of using the map are by clicking around to form a bounding box for the data search, or by selecting one of the predefined locations.

Boundaries and interfaces

Starting from the UI side, the first boundary to look at is between the UI components and the SessionData – class. All controller classes have references to SessionData, which they use to retrieve necessary attributes for displaying.

Next, the boundary between SessionData and SavedDataLogic. As SavedDataLogic is used to save and load information from files, it provides SessionData with data objects for further processing and delivery to the UI. Conversely, SessionData is also used to provide data objects for writing to JSON or XML.

The boundaries between SessionData and the APILogic classes (RoadAPILogic and WeatherAPILogic) are tailored to the fetched data received from the APIs. These APILogic classes work as a middleman between connecting to the API's and constructing the different data objects for SessionData. Only the APILogic classes are in touch with the APIs.

The data classes (WeatherData, RoadData, Maintenance and TrafficMessage) form the structures for storing the relevant information needed in the application. These data points are pulled from the APIs in the APILogic classes and then used to construct data objects. These data objects are held in SessionData for easy use and access in the UI controllers.

The main controller for UI classes is the RoadWatchController. Its FXML file (roadwatch) contains the main window, where all the components are laid out. The button bar is also connected to the FXML file, so RoadWatchController contains all the logic for navigating through different views. The view in the main window (roadwatch.fxml) is changed by calling the FXML file of the wanted view. The FXML file is then connected to its controller (if it hasn't been connected already) and casted as pane. The pane can then be shown in the main window in the infoPane. The UI components are styled via the stylesheet CSS file. The stylesheet is connected to the main FXML file, roadwatch.fxml.

The Map is an external, open-source map library https://github.com/sothawo/mapjfx for Java. It allows the user to click on the map and get the needed coordinates for the API calls.

Components and their responsibilities

In order to avoid writing the same information twice (i.e., writing proper comments in the code, and then explaining the same things in this document), we chose the sensible approach of creating a Javadoc. We found this to clearly demonstrate the components, their responsibilities, and their internal structure. The Javadoc can be found from our git repository. The Javadoc contains all components and their responsibilities in the application. Open the **javadoc/allclasses-index.html** file to see documentation.

Design decisions and reasonings

We chose to do our project with Java, specifically JDK 18. We wanted to use Java since for most of our group it was already familiar from the past and we thought it would be a great fit for this kind of project. The use of Maven also enabled us to easily add external libraries, which proved to be essential in our work.

The main ones to mention were the map library (mapjfx) we used for the map views and Jackson, “a suite of data processing tools for Java” which we mainly used for JSON-related parsing. We also used the Apache httpcomponents library for HTTP calls to Digitraffic.

The broadest design pattern we’ve used in this project is MVC (Model-View-Controller), which can also be seen from the diagram view (Figure 1). FXML being the view, controllers being the controllers and the SessionData being the centre of all models like RoadData, TrafficMessage, Maintenance, WeatherData etc. RoadAPILogic and WeatherAPILogic can be viewed as component factories. They are only used as middlemen in creating the data components by requesting data from the APIs. The project is not 100% MVC, since the approaching deadline forced our hand to create quick solutions outside of MVC principles. However, these are kept to the minimum.

We did it like this because it divides the project into specific areas, allowing the four of us to work together with everyone focusing on their own sector. We have not used SOLID principles as a guideline. However, the principles are present in our project in some ways. For example, the interface segregation principle has been followed by totally removing unnecessary interfaces.

As far as interface classes (or rather, lack thereof) are concerned, in the end we found it redundant to try “fit a round peg in a square hole” and make all the data and API classes follow the same methods. It was quickly apparent that the data types were different enough to warrant just making the APILogic and data classes quite tailored to each API.

When deciding how we wanted to split the functionalities of the program into classes, we concluded that it was better to have more classes, instead of making each class hundreds or even thousands of rows long. Considering the size of the project, we feel that we we’re able to strike a good balance between having each class be distinct and clear, while still keeping the overall number of classes relatively low.

Self-evaluation

The most significant changes we’ve made from the midterm submission are the removal of the APILogic and EnvironmentData interfaces as well as the split from one single RoadData class to three data classes for the Digitraffic data. We found these to be necessary changes to both make the program more streamlined and more clearly defined. As mentioned before, these were made mostly because it became quite obvious these changes were clearly necessary.

We also quickly discovered that the single RoadData class we had initially would eventually prove to be unnecessarily large, and thus we decided to split the data classes to match the three different types of API calls for Digitraffic. This enabled us to not have to implement (quite frankly redundant) checks to see if the data type in the class was actually what was needed for each function. Instead, we were able to make each data class more condensed and only include the things we’d need from each call type.

To stick with the theme of splitting large classes into multiple smaller ones for the sake of clarity and flexibility, we also decided to create a new HelperFunctions class on the last stretch of the project cycle. This was done to enable us to remove a bunch of the clutter from SessionData, which at this point had become quite difficult to navigate around and develop. By moving most of the trivial, smaller functions to a separate class, SessionData became much more readable and contained only the core functionality we wanted.

While keeping all of these changes in mind, we actually ended up sticking to our initial plan quite well. While the details and even large components were completely changed or split up, the core principles we wanted to have in our program from the start stayed. We were able to keep the three-section structure that we had even in very early stages and make it (at least mostly) conform to the MVC pattern. Most of the API/backend classes also stayed relatively close to our initial decisions (having the APILogic classes be the only ones to directly interface with the APIs and have them construct data objects to store in SessionData).