COMP.SE.110 Documentation

H292001 Aapo Kärki

H293115 Matias Aitolahti

H292259 Onni Vitikainen

50121133 Ronja Lipsonen

Program structure diagram

Below is a diagram of the general structure of our program. The main takeaway here are the classes. The methods are still very much subject to change as we move further with development.

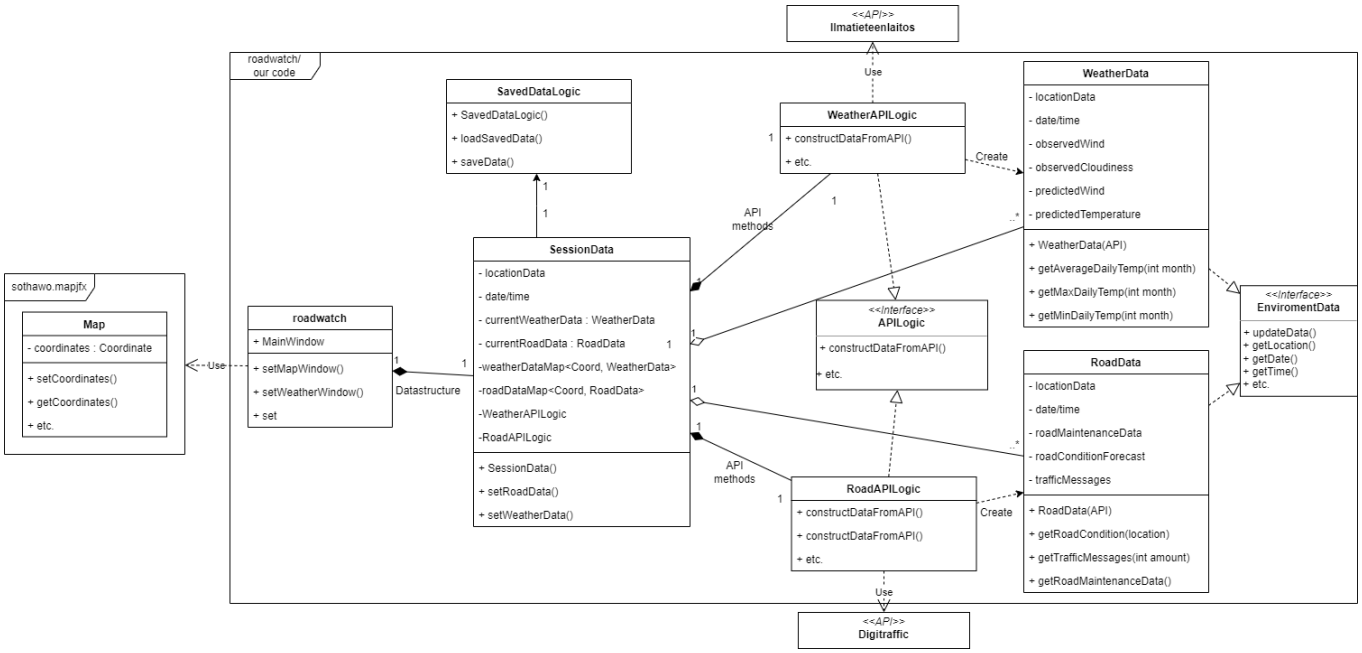

Figure 1: Class diagram of the program structure.

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

These are then divided further into the classes show in Figure 1.

Starting from the API side, there are APILogic classes for each separate API. These implement the APILogic Interface, thusly enabling the addition of further data sources later. Their purpose is to do the actual API calls and instantiate new object of the data classes (here WeatherData and RoadData) from the data received from the API's.

These data classes implement the EnvironmentData Interface. They are used for storing and organizing the data from the API's. The SessionData-class holds the relevant WeatherData and RoadData based on the current location. 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 classes. Data classes of previous coordinates are saved for later use in the session (up to a limit).

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 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. Clicking the road of which you want to see the condition of, is much more user friendly than typing coordinates or cities. This will be implemented along with the option to type coordinates manually.

Boundaries and interfaces

Starting from the UI side, the first boundary to look at is between the roadwatch (UI) class and the SessionData – class. Roadwatch is used primarily for the displaying of data, and it retrieves the information for graphs and tables from SessionData.

Next, the boundary between SessionData and SavedDataLogic. As SavedDataLogic is used to save and load information from files, it provides SessionData with data packets for further processing and delivery to roadwatch.

The boundary between SessionData and RoadAPILogic, and WeatherAPILogic is the same as these classes implement the APILogic Interface. These APILogic classes work as a middleman between connecting to the API's and constructing weather- and road data for SessionData. Only the APILogic classes are in touch with the APIs.

WeatherData and RoadData classes hold the relevant information needed in the application. These data points are pulled from the APIs in the APILogic classes and then added to WeatherData and RoadData classes. These WeatherData and RoadData classes are held in the SessionData for easy use and access in the roadwatch (UI).

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.

Design decisions and reasonings

The starting point for our design process was, of course, the specification we had been given. While the project description document has multiple different practical requirements, it still leaves a lot of questions unanswered. Thus, our initial process started with figuring out the actual features we'd want to have implemented, and how those features corresponded to the requirements set in the specification.

We set off by separating the different requirements into sections and collecting the features for each section in separate parts. This helped us visualize and organize the different parts of the feature set. Based on the division of the parts, we moved on to designing the different screens/views we'd want to have in the application (this was done in Figma).

This UI-first approach meant that we were able to lessen the amount of conceptual planning needed, since the classes we'd need became quickly apparent in the process of the UI design. In addition, by ensuring that the UI prototype meets the necessary requirements, also gave us a much clearer to-do-list of things we'd need to implement.

The definition of classes was mostly based on the needs of the UI elements, as well as the requirements of the API's. Based on past experience of working with similar projects, we chose to separate the logic for fetching API data into the data classes. This helps keep SessionData less cluttered than it needs to be.

For the visual part of the UI we started off by thinking about what colours come to mind when thinking about weather and roads. We chose dark grey and light blue for our main colours. The dark theme of our website is easy on the eyes. We then generated a name for the application using a name generator website and chose the best fitting one. We aim to make the website accessible by using high contrast colours and by keeping the font size above 12pt.