High level description:

Our application's root class is WeatherApp which is responsible for startup process, saving and loading information for concurrent use of the application and linking necessary components together.

We chose MVC for this project because it fits the best with the plan we had. For generating the application UI, we are using JavaFx library. JavaFx follows mostly MVC-architecture. Generated FXML-files are the views, each view has its own controller and model class. Controller handles the interaction between the interface and the model. Model classes contains the information to linked each view.

We have four main MVC sets, each set represents single tab of the application. The tabs are weather, road and combined, settings. In addition to these sets, we also have root set, which function is to offer UI root component, for attaching all the other views. This makes class separation possible.

At startup all the views are anchored to the root view and rendered.

User can save option preferences for easier use of the application. Each view have their own preferences.

Application also contains dataset functionality. By saving dataset, can the user save all the application settings to the single dataset file.

We are using two classes to fetch the data from the APIs used. FmiApi fetches, parses and saves the data from the FMI api and DigiTrafficApi does the same to DigiTraffic api.

Classes and their dependencies are seen in this class diagram (Figure 1).

Components and class responsibilities:

WeatherController:

Controls the UI of the weather forecast tab. Reads and checks user inputs and calls WeatherModel's functions. Also makes the function calls to work on action (pressing a button).

- Method setGraphContent: Reads user inputs from locationTextfield and sctDayDatePicker. Also checks which radiobutton is pressed and calls the drawGraph method with these three parameters.
- Method initializeParentController: Initializes an acces point for using the rootController's functions.
- Method getOptions: Returns weather view's selected settings.
- Method getHashOptions: Return weather view's selected settings.
- Method loadPreference: Opens the preference naming window and loads the selected preference settings.
- Method setPreference(String): Loads the selected preference settings, takes preference name as input.
- Method setPreference(Map): Load the selected preference settings, takes the preference Map as input.
- Method savePreference: Saves the current settings.

RoadConditionController:

Controls the UI of the road maintenances and tasks tab. Reads and checks user inputs and calls RoadConditionModel's functions.

- Method getCoordinates: Reads and checks user given coordinates. Saves them to an arraylist and returns it.

- Method setGraphContent: Reads user's selected radio button and draws wanted graphs by calling RoadConditionModel's functions.
- Method showTrafficMessages: Prints information about traffic messages when the checkbox is checked by calling RoadConditionModel's function.
- Method setStreetIDOptions: Gets street ID data from the API by calling DigiTrafficApi's function and adds the street IDs to the dropdown list.
- Method initialize: adds different info options to the dropdown list.
- Method initializeParentController: Initializes an acces point for using the rootController's functions.
- Method getOptions: Returns selected RoadCondition's settings.
- Method loadPreference: Opens the preference naming window and loads the selected preference settings.
- Method setPreference(String): Loads the selected preference settings, takes preference name as input.
- Method setPreference(Map): Load the selected preference settings, takes the preference Map as input.
- Method savePreference: Saves the current settings.

CombinedController:

Controls the UI of the combined tab. Reads and checks user inputs and calls RoadConditionModel's and WeatherModel's functions.

- Method getCoordinates: Reads and checks user given coordinates. Saves them to an arraylist and returns it. Also saves the coordinates if the user selected a city from the dropdown.
- Method getLocation: Gets a city from the given coordinates.
- Method setGraphContents: Reads selected radio button and date. Then draws the graphs by calling CombinedModel's functions.
- Method setStreetIDOptions: Gets street ID data from the API by calling DigiTrafficApi's function and adds the street IDs to the dropdown list.
- Method initializeParentController: Initializes an acces point for using the rootController's functions.
- Method getOptions: Returns selected RoadCondition's settings.
- Method loadPreference: Opens the preference naming window and loads the selected preference settings.
- Method setPreference(String): Loads the selected preference settings, takes preference name as input.
- Method setPreference(Map): Load the selected preference settings, takes the preference Map as input.
- Method savePreference: Saves the current settings.
- Method initialize: Sets the city dropdown list.

SettingsController:

Controls the UI of the settings tab.

- Method saveDataset: Saves all the of the application settings.
- Method loadDataset: Load and updates settings contained in the dataset.
- Method initializeParentController: Initializes an acces point for using the rootController's functions.

RootController:

Initializes the application UI.

- Method initialize: Initializes all the application tabs with fxml files.
- Method initializeRoot: Initializes an access point to the main Application class.
- Method selectPreference: Opens the preference seletion window.
- Method savePreference: Opens the preference saving window.
- Method getPreference: Gets the the saves preference with
- Method saveDataset: Saves the whole application state to a dataset file.
- Method loadDataset: Reads the dataset file and loads and updates the application settings.

App:

Root class of the application. Holds the data of the application.

- Method start: Starts the UI with root.fxml, so it gives the base to the application.
- Method stop: Saves the preferences, called automatically at shutdown.
- Method savePreferences: Saves the preferences to the json-file.
- Method loadPreferences: Reads the preference-file and updates the application preferences.
- Method addNewPreference: Add new preference to the preference map.
- Method getPreferenceNames: Get all views preferences.
- Method getPreference: Get preference map.

FmiApi:

Gets and parses the wanted data from FMI API.

- Method getMonthlyTemperature: calls the api call function with given parameters and returns an arraylist of WeatherAverageTemperaturePoints
- Method getWeatherData: calls the api call functions with given parameters for place and time
- Method formatDateStringToLocalDateTime: parses localDateTime
- Method parseForecastData: gets windspeed, temperature and cloudiness from the fetched data
- Method parseObservationDataTemperatures: goes thorugh the xml to return temperature data
- Method parseObservationData: goes through the xml data to return the wanted data
- Method loadXMLFromString: gets xml file from string
- Method harmonieForecastSimple: the actual api call
- Method weatherObservationHourlySimple: the actual api call

DigiTrafficApi:

Gets and parses data from the digitraffi api into wanted roadData model.

- Method getRoadMaintenanceTasks: calls the api call function and parses the returned string. Returns a hashmap to the caller with the task name as key and amount as value
- Method getTrafficMessageAmount: call the api call function and goes through the json to return the amount as an integer
- Method trafficMessages: api call to get the traffic messages
- Method roadMaintenanceTasks: api call to get road maintenance tasks
- Method getRoadConditionsForecast: calls the api call function and gets all the wanted info from the json. Returns a hashmap to the caller with the street id as key and arraylist of RoadConditionForecastPoints as value.
- Method roadCondtionsForecast: api call to get road conditions in certain coordinate area

PreferenceType:

Sets the different types of preferences as enum.

WeatherModel:

Draws weather graphs with user's preferences and fetches the data from weather API.

- Method fetchWeatherData calls the fmiApi's getWeatherData function to get the data based on the selected date and location.
- Method fetchMonthlyData calls the fmiApi's getMonthlyTemperature function to get the data
- Method drawGraph gets the selected radiobutton, location and date as parameters from the controller. Based on the radiobutton selection, the correct info for the chart is chosen

RoadConditionModel:

Draws road condition graphs with user's preferences and fetches the data from road condition API.

- Method fetchRoadConditionData calls the DigiTrafficApis getRoadConditionForecast function with users chosen coordinates
- Method fetchRoadMaintenanceData calls the DigiTrafficApis getRoadMaintenanceTasks function with given start time and coordinates
- Method drawChart returns a LineChart that is drawn based on the coordinates the user has provided. The right street is chosen by the id and the chosen info type also comes from the attributes.
- Method drawBars returns a BarChart that is made based on the coordinates and startdate the user has provided. The task names are on y-axis and amounts on x-axis
- Method getTrafficMsgInfo calls the DigiTrafficApi's getTrafficMessageAmount function and returns a string that contains the message to be printed

CombinedModel:

Draws road condition and weather graphs with user's preferences and fetches the data from road condition and weather API.

- Method drawWeatherGraph calls the WeatherModel's drawGraph function with "temperature" as the prechosen "radiobutton" value
- Method drawMaintenanceGraph calls the RoadConditonModel's drawBars function with the given attributes
- Method drawRoadConditionGraph calls the RoadConditonModel's drawChart function with "Overall road condition" as the prechosen info to be shown.

RoadConditionForecastPoint:

Class for road condition data objects

WeatherDataPoint:

Class for weather data objects

WeatherAverageTemperaturePoint:

Class for average weather data obejcts

JsonConverter:

Class to make JSON from string or to read JSON

- Method getFromJSON: Deserialize json string to object.
- Method toJSON: Serialize Java object into json string.

Utils:

Class for formatting function

- Method formatDate. Get formatted vesion of the date.

Weather.fxml

Implements the UI for the Weather tab.

Root.fxml

Implements the UI base for the app.

RoadCondition.fxml

Implements the UI for the roadCondition tab.

Combined.fxml

Implements the UI for the Combined tab.

Settings.fxml

Implements the UI for the Settings tab.

Design decisions:

To be honest, we didn't use SOLID principles enough in this project. We know that they of course should've been used but here our main goal was to get the necessary functionalities done first and then the schedule didn't really allow anything else.

On top of the MVC we didn't use any particular design technologies or patterns. We actually didn't have any common decisions about the patterns or how to do certain parts on a detailed level. Everyone did how they thought was best. We divided the work thinking about past experience in coding. Matti had experience of APIs so he did that, for example.

Self-evaluation:

Before mid-term:

After the first TA meeting we had to change our plan to include which MVC variant we were going to use. We decided to use the basic MVC. In the updated class diagram (figure 2) you can see the added model classes that are made for all the controllers. This is the biggest change we have made. Also the methods of different classes have been updated. After starting the coding, we realized that the planned api manager classes and the ApiHandler class were unnecessary.

Otherwise, we feel that our design has supported the implementation quite well. We have found some variables to be unnecessary and figured that some important variables were missing from the design and class diagram. So far the quality of the app hasn't been perfected but it does what it's supposed to. For the remaining features we probably need to make some changes to the methods and variables we have planned.

Final:

Now that the project is finished, we can state that we have been able to implement all the wanted things. The process of modifying the class diagram can be seen below. Some supporting classes had to be added for the final submission (figure 3) but the difference between the midterm (figure 2) and the final version isn't that big. Mostly we have added variables that come from the FXML file. So we could say that we were able to stick to the midterm design plan quite well. Compared to the first diagram (figure 1) it can be seen that we were off with how many classes and functions we would end up needing.

One idea we had to drop was having a table of weather data to be shown in the weather tab. The JavaFX TableView object didn't end working the way we wanted it to, so we just deleted it. It would have been an extra element and maybe could have been done if there was more time. It doesn't affect the requirements as there are graphs. One challenge we faced had to do with how to get the road info data presented to the user in the best way so that it wouldn't be confusing. The api returned road condition data with a street id which we realize isn't very practical. Trying to convert the street ID to the street name, we ran in to some issues and couldn't get it done. There is a lot of things the user has to choose on the UI but with the error pop-ups we made, it should guide the user if they're having problems.

We also had some trouble figuring how we wanted to combine the weather and road info data. We decided to go with always showing the temperature with overall road condition or maintenance tasks. We feel that the temperature graph can kind of give a reason for the road data. For example if it's very could outside, the tasks could include snow ploughing etc. Some issues also appeared with the apis since they didn't always have data to return so it was a bit difficult at first to make it so that whatever prints on the screen is informative enough for the user.

Overall we feel that our plan and implementation of it works well for the wanted product.

Reasonings:

We decided to use a class WeatherApp to save and hold all the data. This way it's easy to use different UI controllers using the same data. The APIs have models for fetching the data from the DigiTrafficApi and FmiApi are managed through API managers.

For tracking project status and assigning tasks to developers we are using Trello and Kanban tables. We have decided to use JavaFx as an UI-library in the project and for common tools we have chosen SceneBuilder and NetBeans IDE.

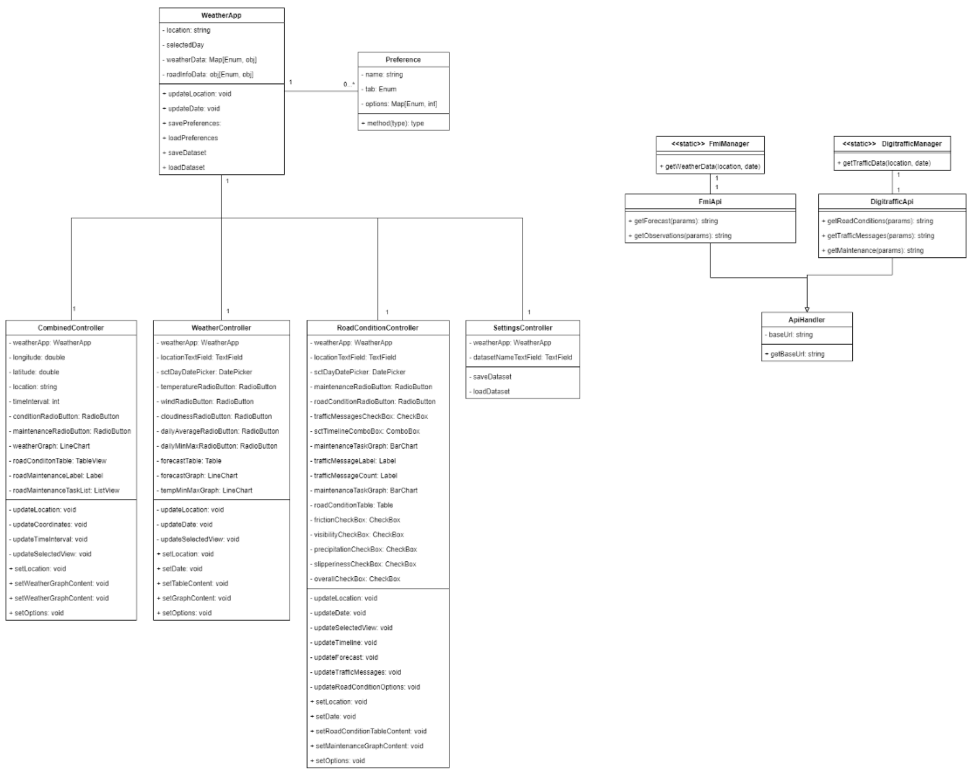

Figure 1: the first original class diagram

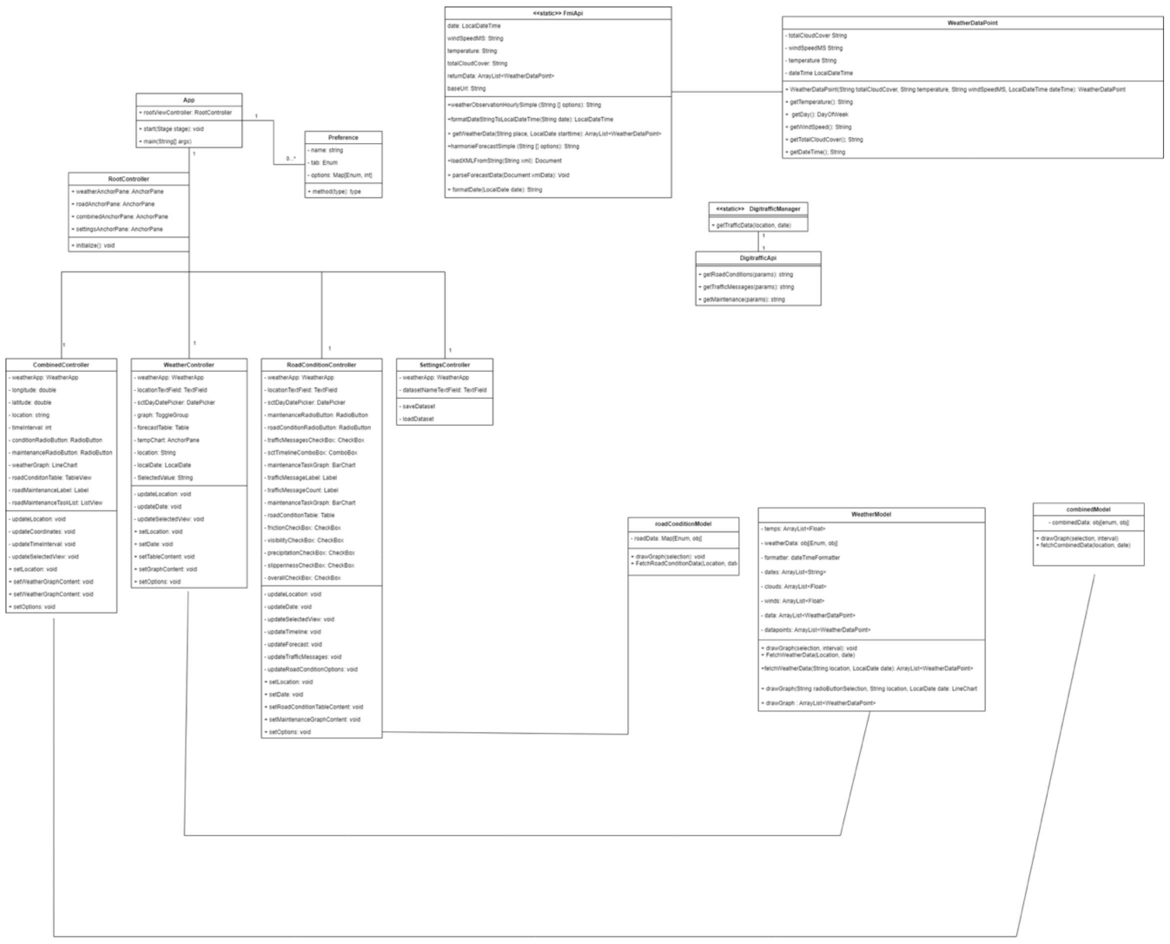

Figure 2: midterm class diagram

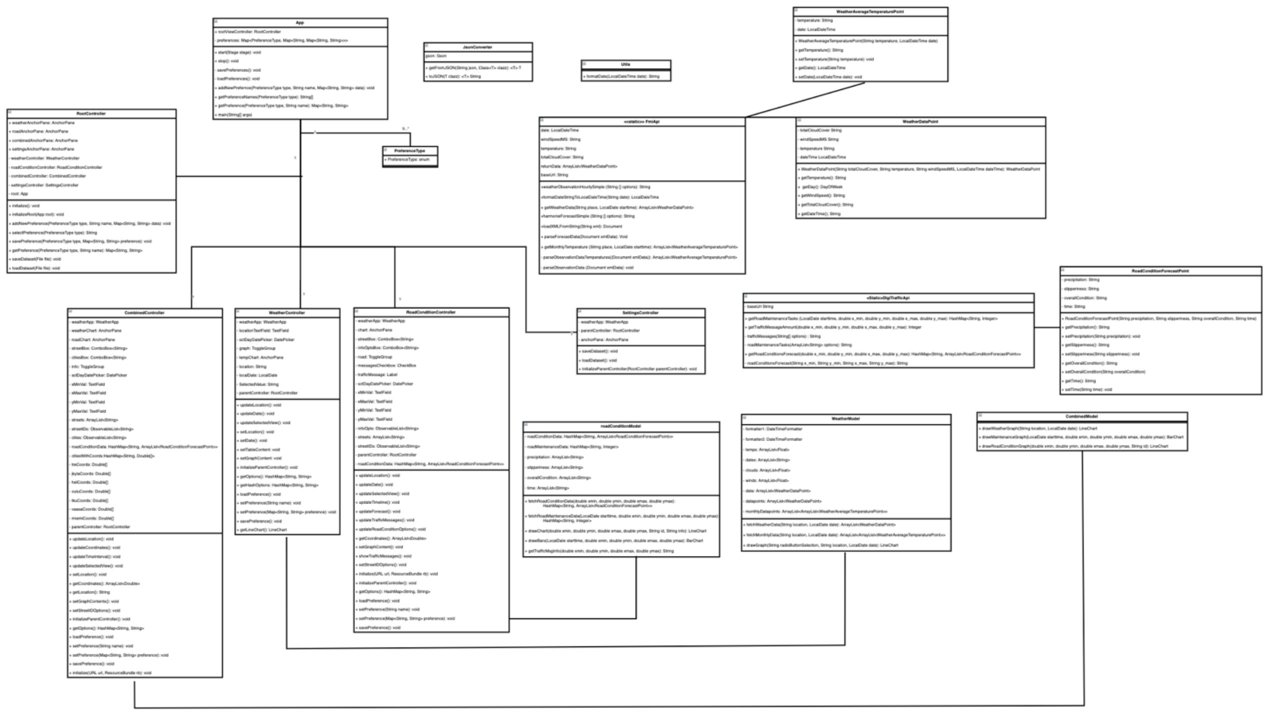

Figure 3: final class diagram

How the work was divided:

Class	Who made it
App	Akseli
CombinedController	Pinja/Akseli/Melina
CombinedModel	Melina
DigiTrafficApi	Matti
FmiApi	Matti
JsonConverter	Akseli
PreferenceType	Akseli
RoadConditionController	Pinja/Akseli/Melina
RoadConditionForecastPoint	Matti
RoadConditionModel	Melina
RootController	Akseli
SettingsController	Akseli
Utils	Matti
WeatherAverageTemperaturePoint	Matti
WeatherController	Pinja /Akseli/Melina
WeatherDataPoint	Matti
WeatherModel	Melina

For the controller classes the work was divided so that Akseli made the functions to do with preferences, Melina did some of the error checking and Pinja did calling the functions and adding things to the UI.