Design Document: Design Decisions

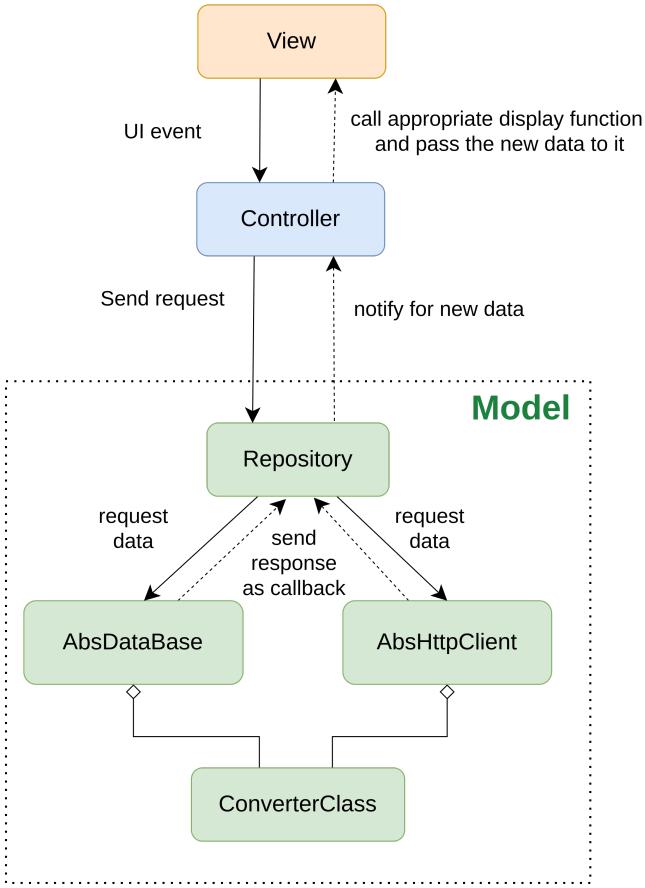
Application structure

The application is divided virtually into 4 main parts

- Messages section
- Forecast section
- Maintenance section
- Weather section

Each one of these sections has its own DataBase and API communication logic as well as a separate UI class which can be integrated into the general frame of the application and can be **combined** with other sections in the same window.

General class diagram for each section



The core of each section is built upon the **push MVC** Pattern as follows (Fig1):

- **View** class: which contains the UI that represents the data from the Model
- **Controller** Class : which reacts to events from the UI and forwards the appropriate request to the Model. It also receives data from the model and calls the appropriate functions of the view to display the data on the UI.
- **Model**: which is divided into 3 other classes to **separate the responsibilities:**
  - **DataBase**: handles the retrieving and storing local copies of data saved by the user
  - **HttpClient**: handles communication with a certain API to fetch data requested by the user
  - **Repository**: an intermediate class that acts as a **Facade** which knows how to delegate the tasks to either the DataBase or the HttpClient` and can support many variations of HttpClient, which makes the project easily scalable and decouples the UI classes from any internal data class. In addition, this class allows adding new sources like Statistics finland easily by extending the Repository class and/or overloading some of its functions.

General Design decisions

1. **DataBase and HttpClient (Data Source Classes):** Since many classes will be having similar functionalities when it comes to fetching and storing data from the local data base or from the API. We decided to make two general abstract classes **DataBase and HttpClient** which implement all the common functionalities needed, then in each section (Messages, Maintenance, Forecast and Weather) creat classes that **inherit** the **DataBase and HttpClient** and add the logic for Converting the actual data fetched from a certain source into c++ objects which can be used for displaying on graphs and diagrams in the UI.
2. **Converter Classes:** we also chose to encapsulate the data conversion logic (from Json or Xml to a specific c++ class type) in separate classes called **conversion classes**, then these classes can be invoked from the Specific implementation of the **Data Source Classes**. This decision was taken because the database stores files in the same format as fetched from the API, so both DataBase and HttpClient classes can use the same conversion logic which we can reuse by calling the converter class. In addition this **further divides the responsibilities of the classes.**
3. **How data is stored in the database:** We store the data in JSON or XML files, depending on the source (DigiTraffic or FMI) to specific directories that have the same

name as the app sections. For instance, Messages data is stored in its own directory which contains json files of data selected by the user.

4. **Parameter classes:** for each section we have what is called a **ParamClass (eg: MaintenanceParams)** these classes store the parameters given by the user on each UI. For example the **MaintenanceParams** class hold the coordinates, task type, start time, and end time chosen by the user. These Param classes bring many benefits:
    - They make it easy to propagate the parameters chosen by the user through different classes, avoiding to have a large number of function parameters,
    - Most importantly, they allow us to preserve the **Open/Close principle**, by avoiding to change functions signatures each time a new input field is added to the UI.
    - Instances of these classes can be serialized to JSON objects and stored to JSON files, which means, if we want to store the user inputs to be remembered for the next app usage, we just need to store them in JSON format and parse them back again when needed, *satisfying requirement 5 described in the project specifications document*.
5. **Combining data sources**: Our decision for satisfying requirements 1.5 and 3 from the specifications document requirements was as follows. Since both requirements ask us to combine/show in the same window traffic data with messages or weather data, we thought that the least invasive way is to have a side bar when showing traffic data. The side bar can be used to view messages inflammations in the normal case. And when the user wants to combine and compare weather data with the traffic data, he can choose to hide the messages and show the weather option. This option can have two cases:
    a. When maintenance data is showing: the weather section shows Hourly weather observations.
    b. When forecast data is showing: the weather sidebar shows Weather Forecast for the next 12 hours, which can be compared with our forecast graph that presents the same data but from the Digitraffic API instead of FMI.

Used design patterns

Pattern	Example Classes	Reason
push MVC	MaintenanceView, MaintenanceController, MaintenanceModel=(Maintenance-Repository/Client/DataBase/Data)	Decouples data fetching and Business logic from UI classes
Facade	Any repository class	hide away the different data sources from the controller classes. This allows easy addition of new data sources, and combining sources data.
Singleton	AppDataBase	avoid creating multiple instances for a shared resource.

SOLID principles can also be noticed from the general architecture of this project. For instance single responsibility principle can be clearly seen the the Model sub-classes. The decision about parameter classes and The HttpClient and AbsDataBase classes also emphasize on the OPEN/CLOSED principle.

Class diagrams and internal implementation

Please refer to the **Design_Document_class_diagrams.pdf** document for an overview of the class diagrams and interfaces, in addition to details about the responsibility of each class.

Documentation of “back-end classes”

Tasks were split between the members of the group such that some focused on the writing logic for the UI classes (View and Controller) while others focused on writing (Model) code that fetches the data from the different APIs and convert it into C++ objects or storing the objects to the local database. In order to facilitate cooperation, usage of so called ‘back-end classes’ or the classes that make the Model in MVC, are all documented in a single class called **backendexample.cpp** which lies in the root of the project. This way not only group members can take code snippets and refer to this documentation to know how to use the appropriate classes, but also the peer reviewers can understand our implementation easily if they take a look at that file.

Self evaluation

Comparing the initial prototype to the final application, we would say that the prototype helped us a lot in determining how each section was going to look like and how the data will be presented. Eventually, this allowed us to determine clearly which data elements were of interest to us from the different data sources as well as which kind of widgets we needed to represent that data. So, the final implementation ended up looking very similar to what we have anticipated during the prototype phase. However, saying that, we also had some parts that needed some adjustments to fit in the actual implementation as well as some parts that needed a complete redesign which will be discussed in details as follows:

- **Parts that needed some adjustments:** the weather data source provided hourly, daily and monthly data only for observations and not for forecast. So, we had to add a forth choice in the time step drop down menu for forecast, as it did not have intervals like the weather observation, but only provided a maximum of 56 hours of data. In addition to this, some of the graphs shown in min/max/avg calculation provided redundant data so we had to omit those as well.
- **Parts that needed complete redesign:** initially we had decided to use an SQL database for locally storing the data coming FMI and DigiTraffic. So, the design document contained parts that assumed data is going to be stored to an SQL data base. However, while implementing the back-end of the app, we came to the conclusion that storing data in the same format that was provided by the data source allows us to use common conversion logic and further reduce the complexity of the project. So, some of the classes structure had to be changed, and some new classes had to be introduced.

In total, we would say that around 65% of decisions from the prototype phase remained the same. 15% needed to be readjusted, and 20% required complete redesign. Moreover, the prototype phase helped us to define clearly the different types of tasks that needed to be done during the development of the project and allowed us to easily split those tasks between us.