Software Design Project

Kharel Shurakshya, Meghla Tamara, Mehraj Ali,
Salminen Aleksi

Software Design Document

Version History

Version	Date	Description
1	20.02.2022	Initial version of document
2	20.03.2022	Mid-term design document

Table of Contents

1 INTRODUCTION

The document describes the design and system specification of the desktop application built during the spring 2022 implementation of Software design course in Tampere University. The document provides high level description of the design describing major components used for creating software and their interfaces.

1.1 PURPOSE AND SCOPE OF PROJECT

The aim of the project is to design and develop software for monitoring real-time data on greenhouse gases and comparing current data with historical data on greenhouse gases. During this project, the group shall create a standalone desktop application for visualizing and monitoring greenhouse gases utilizing the database and API provided by SMEAR and Statistics Finland STATFI. The scope of project includes monitoring real-time data from SMEAR, checking historical values provided by statistics Finland, comparing historical data to current data, and visualizing the trends of greenhouse gas from history to present.

1.2 SYSTEM OVERVIEW

The desktop application will be built using Java and its libraries. The software will be built using Model-View-Presenter (MVP) architecture pattern, an architecture pattern which divides application into three layers model, view, and presenter. Java Swing, which is a graphical user interface (GUI) widget toolkit for Java will be used for the graphical user interface needed for the visualization of the data.

The software utilizes the data from the two different sources SMEAR and STATFI and is visualized and monitored in the application. The model component stores the data from the sources and communicates with the sources. The view provides visualization of the data as per the requirements and built prototype. Finally, the presenter fetches the data from the model and applies the UI logic to decide what to display. Since, the application focuses on UI components and the requirements consists of particularly four main use case, MVP pattern makes convenient to build logic with each use case. The loose coupling between view and model helps to manipulate the data without the change in model.

1.3 DEFINITIONS, ACRONYMS AND ABBREVIATIONS

GUI	Graphical User Interface
MVP	Model-View-Presenter
STATFI	Statistics Finland
API	Application Programming Interface
Class Diagram	Describes the structure of the system and interfaces between classes
UI	User Interface

2 DESIGN CONSIDERATION

The section explains some of the restricted development methods as per the course requirement along with explaining system features and architectural strategies explaining the reasons for choosing the preferred language and design patterns as mentioned in section 1.2 as well.

2.1 DEVELOPMENT CONSTRAINT

The software is supposed to be developed using any of the four-programming language Java, C++, C#, and Python as per the course requirements. In addition, the software should be standalone, or desktop application. Java being extremely portable which allows the application to run on any computer regardless of hardware features or operating system and the familiarity of the team members with the language made it the choice to use Java.

Moreover, web application being not permitted, UI compatible with Java needed to be chosen. Java Swing is a lightweight GUI toolkit which has a variety of widgets for building optimized window-based applications. Swing is built on top of the AWT API and entirely written in java. It is platform independent unlike AWT. Moreover, the javax.swing package provides classes for java swing API such as JButton, JTextField, JTextArea, JRadioButton, JCheckbox, JMenu etc. The compatibility with Java language and already defined components like button, checkbox makes the language easier to integrate and develop user interface as well.

2.2 SYSTEM FEATURES

1. Users can select four options to view data from the software. Each option can be selected from the nav bar. The options are:
    - Monitoring real-time data from SMEAR (Tab name: Real-Time Data)
    - Checking historical values from STATFI (Tab name: Historical Values)
    - Comparing the current situation to history (Tab name: Comparison)
    - Breaking down averages (Tab name: averages)
2. In Real-Time data window, user can select which monitoring station's (one or more) data they are interested in.
3. In Real-Time data window, user can select which greenhouse gas / data variable they are interested in (minimum options: CO2, SO2, NOx)
4. In Real-Time data window, user can set a time-period from which data is shown
5. In Real-time data window, in addition to viewing raw data, user can ask for a minimum, maximum and average values of the selected data
    a) For a given time-period for a given station
    b) For several stations for a given time
    c) For several stations for a defined time-period
6. In Real-Time data window, users should not be able to select time range where data is not available, null values should be handled.
7. In Historical values window, the user can select a time range for viewing data from STATFI
8. In Historical values window, user selects one or more available datasets: CO2 (in tonnes), CO2 intensity, CO2 indexed, or CO2 intensity indexed (one or more).
9. In Historical values window, user is shown a visualization of the data based on selection.
10. In Comparison window, user is given a visualization of the CO2, SO2 and NOx values from SMEAR for a time-period specified by the user, for the station(s) selected by the user

11. In Comparison window, the user can select data available from STATFI on historical averages
12. In Comparison window, the user can specify the time range to view data
13. In Comparison window, the historical values are shown alongside real data in a comparable way.
14. In Averages window, the user selects a time range for viewing data from STATFI
15. In Averages window, user selects data from CO2 (in tonnes), CO2 intensity, CO2 indexed, or CO2 intensity indexed.
16. In Averages window, user is shown a visualization of the data according to the data selected.
17. In Averages window, the user can choose real time data from SMEAR
18. In Averages window, breakdown of the historical average to the selected year based on SMEAR data is visualized.
19. The user can save preferences (e.g. certain stations, certain time period in history, which greenhouse gas(es) from SMEAR, which statistics from STAT FI) and apply them when using the software later.

2.3 ARCHITECTURAL STRATEGIES

PCoModel-View-Presenter (MVP), the main goal is the separation of concerns between the user interface (UI), the Model deals with application data, View is for displaying data and reacting to user input and the Presenter handles business or presentation logic. In MVP, View does not actively update itself, instead choosing to allow the presenter to handle that task. Presenter also allows for Model and View to communicate with each other.

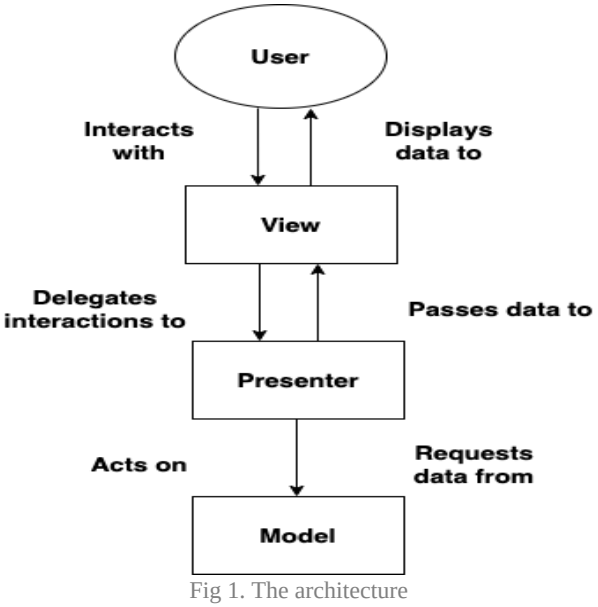

Fig 1. The architecture

As we can see in figure 1, the Presenter listens for any user interaction through the View. The Presenter then fetches the relevant info from a particular service or APIs. After that the Presenter updates both the View as well as the Model. Thus, the model and the view layers are completely isolated from each other.

3 SYSTEM DESIGN

The section includes the detail description of the application. The class diagram describing high level components and interfaces is included along with details of use environment, programming standards and so on.

3.1 CLASS DIAGRAM

3.1.1 General

The application is divided into four main components with respect to the user interface designed according to the use case requirement. The components are Realtime, History, Comparison and Averages. Each component has model, view, and presenter class of its own. In addition, each model includes communicator object for the communication with external API. Separating the UI components into four parts enables the model to not change frequently and handles the application logic in convenient way.

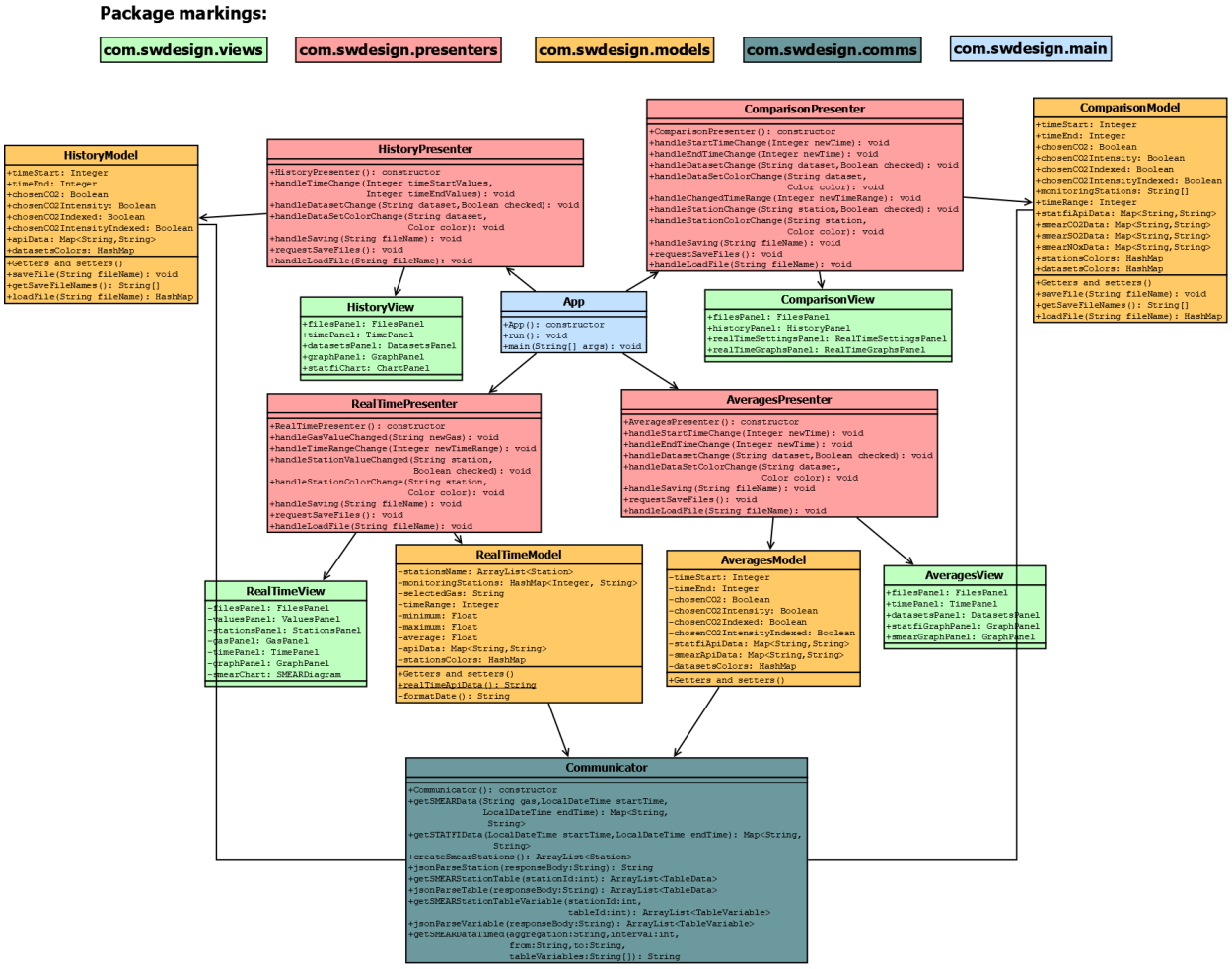

Fig 2. Class Diagram of application

Figure 2 highlights the main classes and their methods. The “App”- class is the main class of the application. It contains four different presenters. These four presenters hold references to their own views and models. Each model contains a Communicator object which handles everything related to connecting to the external APIs. The View classes use JFreeChart objects to display the data from the APIs.

3.1.2 View

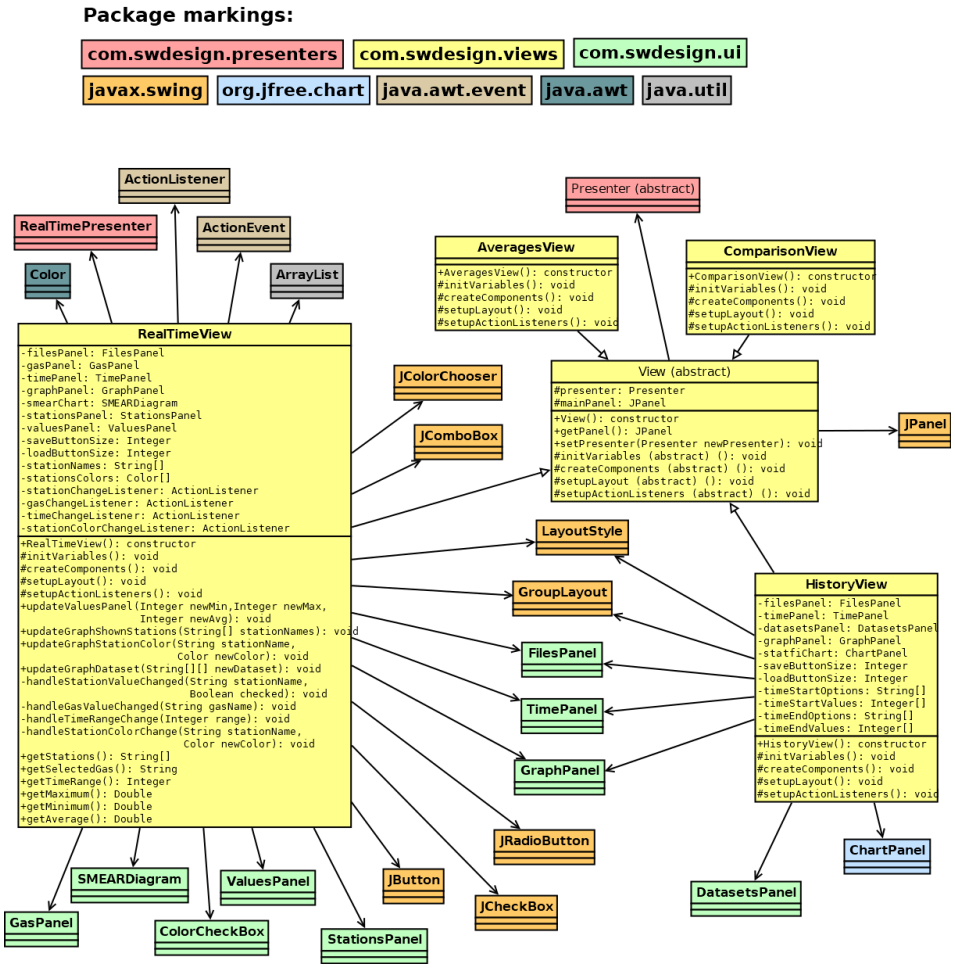

Fig 3. Class Diagram of the View-part of the application

Figure 3 shows the current classes and dependencies of the View-part of the application. The package each class belongs to has been color-coded for ease of reading. The abstract class View defines the basic functions and attributes of each View-instance. The implementation of View-part of the application is divided into four different classes (RealTimeView, HistoryView, ComparisonView, AveragesView) which all inherit the abstract View-class.

So far, the RealTimeView-class includes most of the required functionality, while the HistoryView-class only contains the UI components, but no functionality. AveragesView and ComparisonView do not yet contain any UI components (other than the main panel) or functionality.

For the graphs and charts, the application uses JFreeChart (https://www.jfree.org/jfreechart/)

For other UI components, the application uses Java Swing components, and self-made components defined in the com.swdesign.ui-package. ActionListener- and ActionEvent-classes from java.awt.event-package are used to handle events and create action listeners for UI components.

3.1.3 PRESENTER

Presenter implements the interaction between the view and the model. As the model and the view do not interact directly, the presenter acts as a bridge retrieving data from model and formatting it in a way that view can understand it.

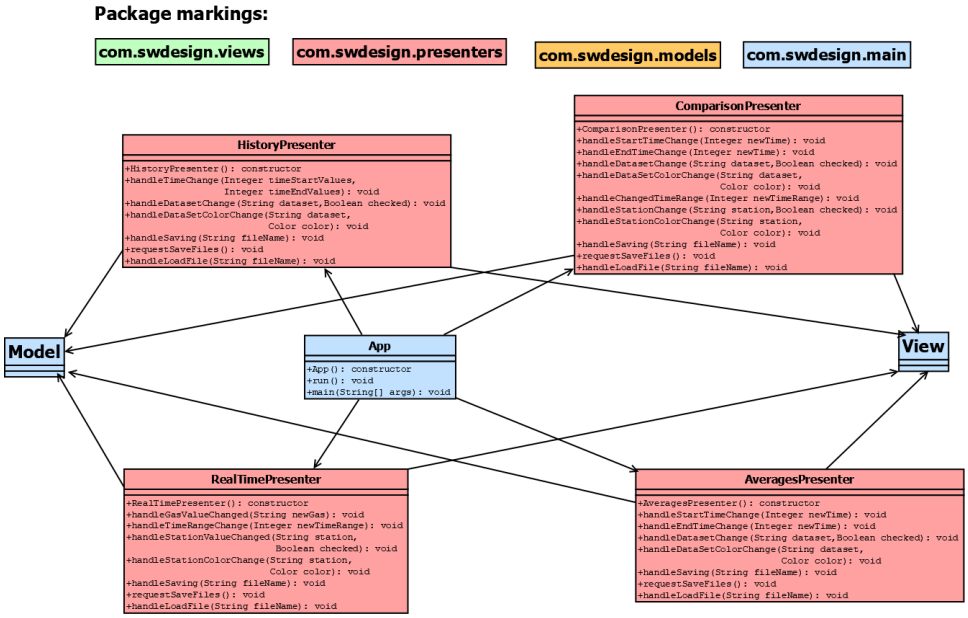

Fig 4. Class Diagram of Model package

3.1.4 MODEL

The model stores the data from the API and provides interface for the presenter to get the data via the communicator. The RealTimeModel has the data needed for the Realtime user interface. It gets the station data from communicator for listing the monitoring stations name in Realtime user interface; the checked monitoring stations, the gases name and time range are passed from the user interface to the model to get the time series data. The RealTimeModel has getters and setters for all the class variables and class method for getting real time SMEAR Data with the necessary parameters like the time series variable for calling the getSMEARDataTimed function. The variable name of gas and database name are different for each type of station and gas name, so the final class Values stores the name with key, value pairs where keys are the name from the user interface and value is variable name for the SMEAR Api to get the timeseries data.

The class diagram explaining the Realtime component along with the three other components are provided.

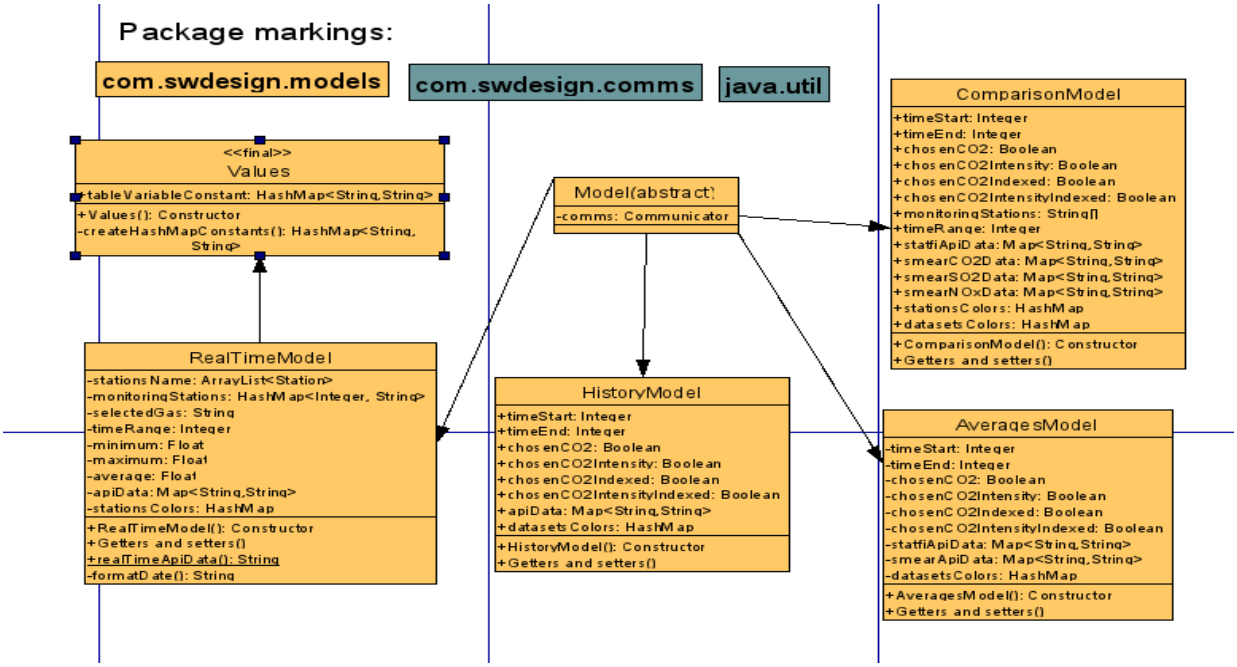

Fig 5. Class Diagram of Model package

Figure 5 shows the class diagram with the inner components and interfaces for the model package. RealTimeModel class has the dependency on another final class Values. The common java utilities library has been used for formatting time, date, collections. The HistoryModel class is implemented with class variables and getters and setters methods. The remaining two classes don't have any functionality and will be implemented in future.

3.1.5 COMMUNICATOR

The communicator holds all the API endpoints that are requested through the application for SMEAR and STATFI. The main function to fetch Station data from SMEAR API is createSMEARStations. The function fetches the station data from SMEAR station API and parses the values in JSON and stores the values in the Station Object created by the Station class. As each station has multiple tables, each station also has an ArrayList of TableData objects, for which a separate class TableData is created. And each TableData has multiple Table variable data, for which TableVariable class is created and stored as ArrayList values in TableData. The 'jsonParseStation' function parses the Station values in JSON format and extracts the values to save as Station objects. This function is called from the createSMEARStations function. Similarly, the 'jsonParseStation' function calls the getSMEARStationTable to get the Station table values from the table API and jsonParseTable is responsible for parsing the values and save the data as TableData objects. The 'getSMEARStationTableVariable' function is called from the jsonParseTable function to fetch the table variables for each table. The jsonParseVariable function parses the values fetched from the function 'getSMEARStationTableVariable' and saves the values as TableVariable objects. The model can access all the generated values and objects by simply calling the createSMEARStations function and it will return an ArrayList of SMEAR stations after running all the functions. The structure can be seen from the diagram below:

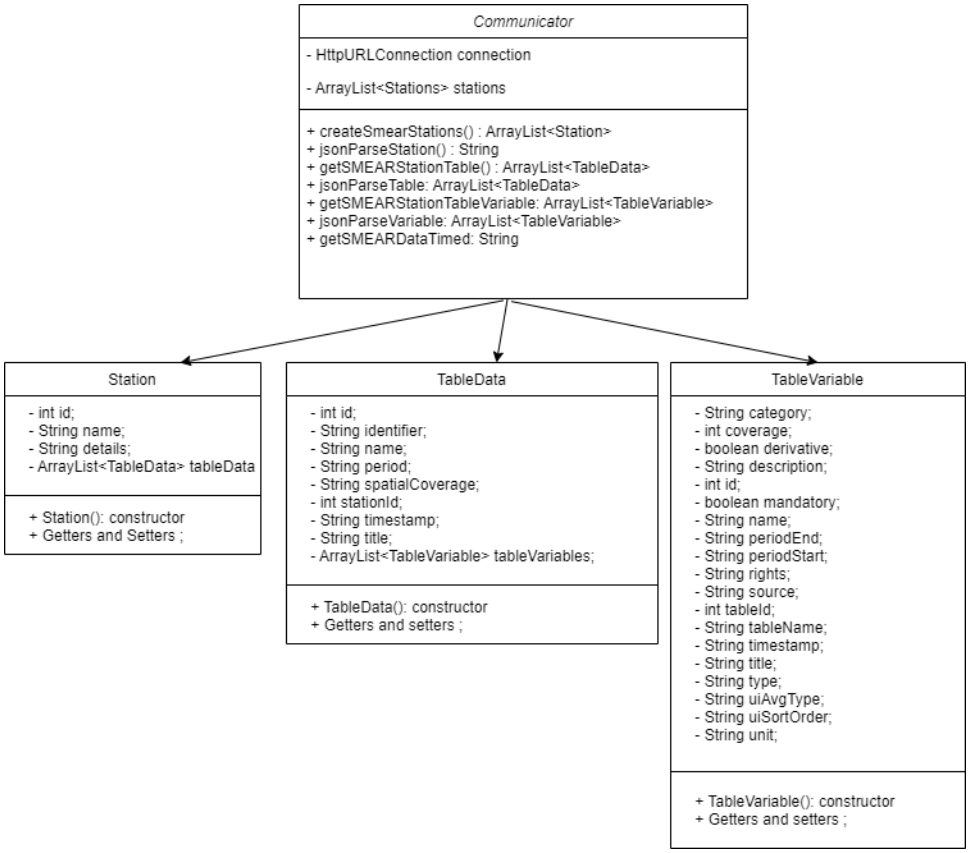

Fig 6. Class Diagram of the Communicator-part of the application

3.2 PROGRAMMING STANDARDS

The software is developed using Java version 17. Java being the object-oriented programming language, the conventions of OOP concept and java coding standards shall be followed.

3.3 NAMING CONVENTIONS

The standard naming conventions of Java shall be followed. The packages name in lowercase, class, interface name in CamelCase, methods, variables in mixedcase and constants in uppercase.

3.4 SOFTWARE DEVELOPMENT TOOLS

- **Java** is used for development
- **Java Swing** is used for GUI
- **Figma** is used for creating UI prototype
- **NetBeans** is used as an editor for writing java code
- **Gitlab** is used for codebase and version control management
- **Gitlab Issues and Boards** is used for tracking tasks and features completion

3.5 OUTSTANDING ISSUES

The prototype of the application is built. Some of the components like communicator for SMEAR API is implemented but however, STATFI API is yet to be implemented. The API calls require several calls to manage the data. Caching of data and further structuring of data is remaining for future implementation. The model defined currently can yet change due to user interface components.

3.6 SELF EVALUATION

/1 View: The original design offered a good start for implementation. The prototype helped a lot in getting started, since it already contained most of the UI components, although in a quite rough package. The original design covered the high-level components of the view, that is, the division of the View-part of the application into four different View-instances. It also covered all the UI components in a quite rough manner. In the updated version, the basic UI components were divided into their own classes. For these components, a separate package was created, the ui-package, while the views reside in the views-package. In the updated version, the UI components are reusable and more flexible, while the old version contained only hard-coded components. The new version is also a lot more detailed and contains event listeners and update functions, which the original version did not. I believe the current design will hold well for the rest of the implementation process, since by now most of the core functionality of the view and the application is quite familiar. The new design is also quite flexible, and a lot of the UI components are already done, which will speed up the process in the future.

/2 Presenter: The Presenter acts as middleman containing UI business logic for the View. Presenters retrieve data from the model and return it formatted to the View. We have 4 presenters for 4 different tabs. RealTimePresenter gets gas data, time ranges data, station data and color data from View and retrieves real time api data with getRealTimeApiData from the Model. The system fetches monitoring station data and updates it to the View with updateGraphShownStations function.The system retrieves the color data from the Model with getStationColors and passes it to the View by updateGraphStationColor.

/3 Model: The model stores the SMEAR Data as well structures of data in accordance with the user interface currently for the midterm submission. The model is divided into four different models. The RealTimeData model has data for the real-time presentation of data from SMEAR API via the communicator. The instance of communicator gets the stations data to list in the user interface. The function for getting data via communicator function is provided with parameters received from the presenter. The changes for storing the station data were made to get the list of station. The variables needed to call the timeseries from SMEAR API had several names for the gases so that was stored with static key-value pairs in Values class for mapping user interface variables with Api call variables which was not present in the original design.

/4 Communicator: The communicator holds all the API endpoints that are requested through the application for SMEAR and STATFI. In our midterm submission we focused on completing the SMEAR APIs for the application. The communicator is designed in such a way that it has clear static functions to generate SMEAR station data objects and pass the API data fetched for the stations to the Model as Station objects. There is a static function in the Communicator to fetch timed SMEAR data as well. We had to make small changes to our Communicator from previous version by adding Station, TableData and TableVariable classes to support Station data fetched from the SMEAR API.