TA	COMP.SE.110. SOFTWARE DESIGN PROJECT	29/10/2023
Kristian Skogberg	Software Project	GROUP NAME: DESIGN DRIVEN DEVS Long Nguyen Minh Hoang

CONTENTS

1. Introduction

Weather applications have become an integral part of our daily lives, helping individuals plan their schedules, decide the activities, and be cautious about weather conditions. This project seeks to create a friendly, with some functionality and solutions by integrating Application Programming Interfaces (APIs) into a graphical user interface (GUI) implementation to deliver necessary, accurate, and up-to-date weather data to users.

2. Scope and Objectives

The project is aimed at these purposes:

- Utilizing object-oriented design to create a Weather system.
- Integration of a GUI as a component of the program
- Applying the Model-View-Controller (MVC) software design pattern for development, classification of the project
- Incorporating external libraries, such as OpenWeather, into the implementation
- Providing and ensuring the high-quality GUI adapts to Desktop screens by applying heuristic, principles of design guidelines based on human technology interaction topics.

3. Overall Architecture:

The application can follow the MVC architectural pattern, which is suitable for JavaFX applications. This pattern helps maintain the separation of concerns and makes your application more maintainable.

4. GUI Implementation

4.1. Packages:

The project's packages will be divided into four parts, which will support us in minimizing the unnecessary teamworking Git workflow conflicts, and helpful for us to divide the tasks in the future.

The packages will be separated based on each window, which can be demonstrated in Figure 1.

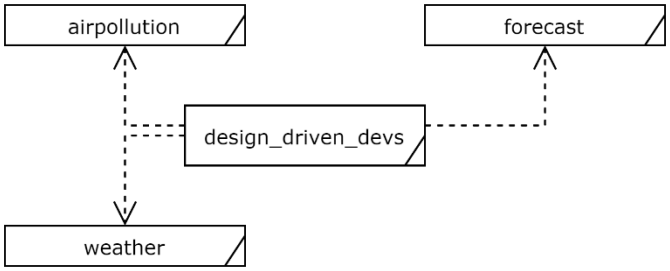

Figure 1. Packages demonstration of the Project

The relationship between the classes can be demonstrated right now by Figure 2

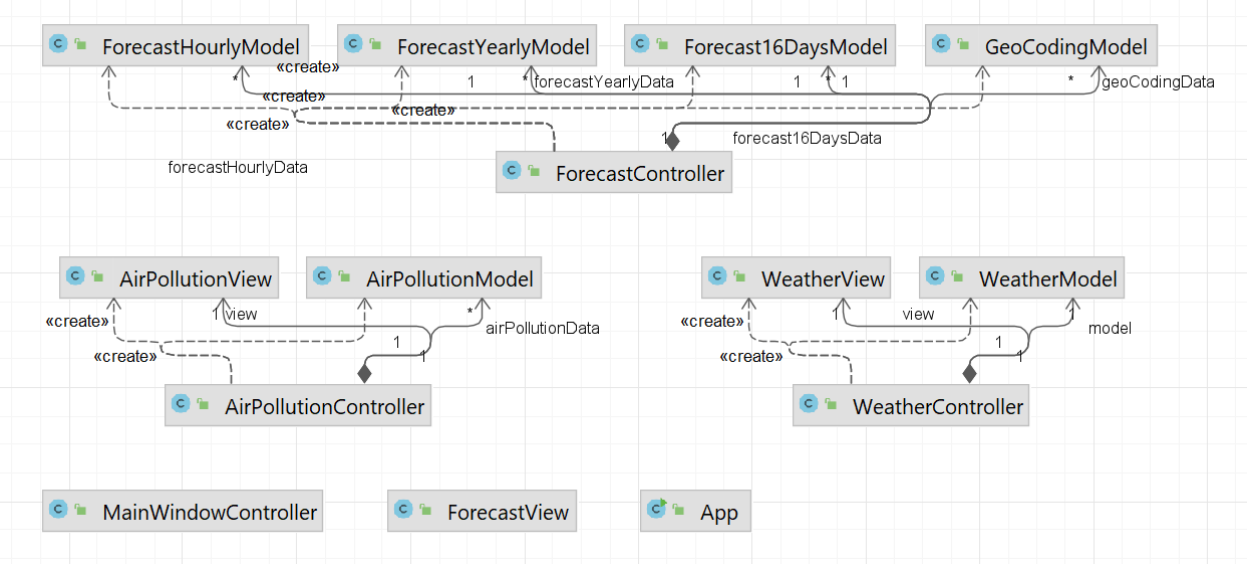

Figure 2: Current classes' relationship

4.2. Selection Screen

Since this is the second stage of the project, I would like to implement the screen with the necessary components and rough designs. Therefore, the GUI layout can be presented in Figure 3, and the UML diagram for the necessary classes can be illustrated in Figure 4

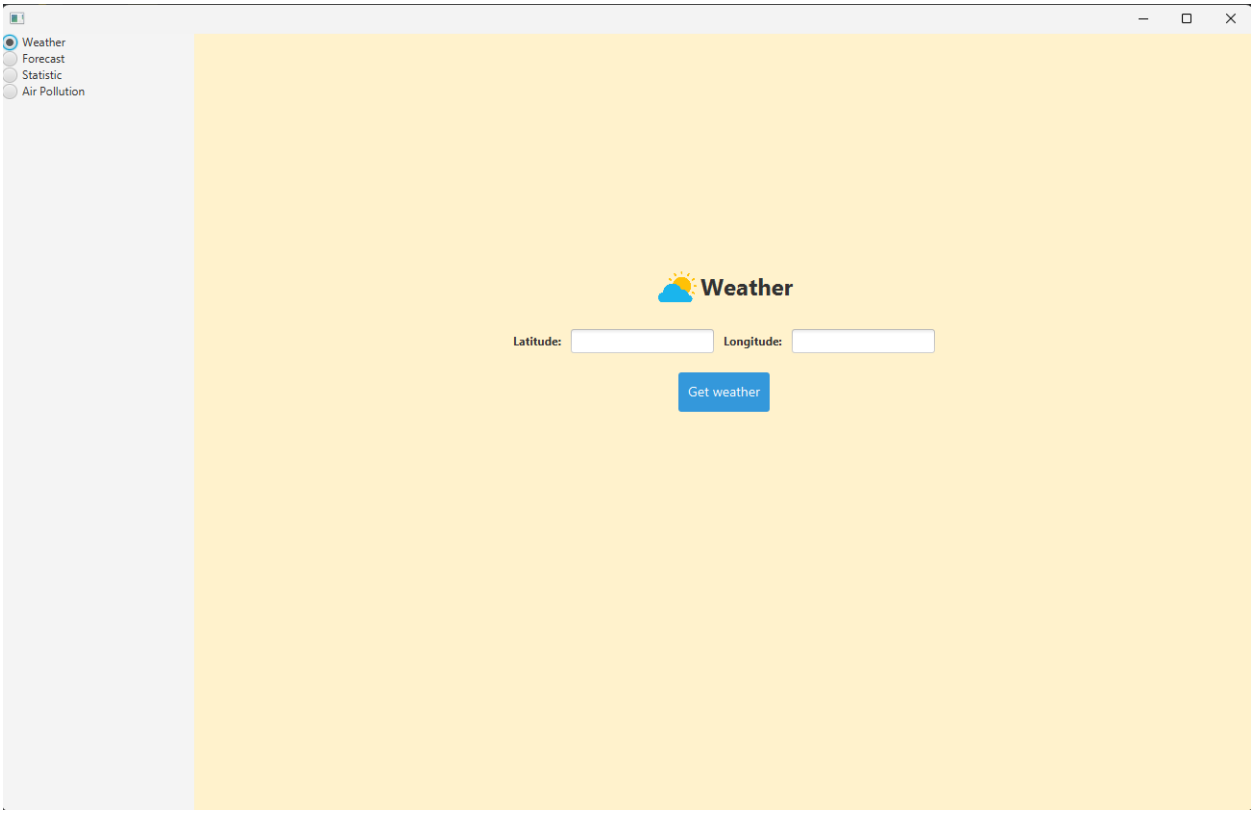

Figure 3: GUI layout for the first screen

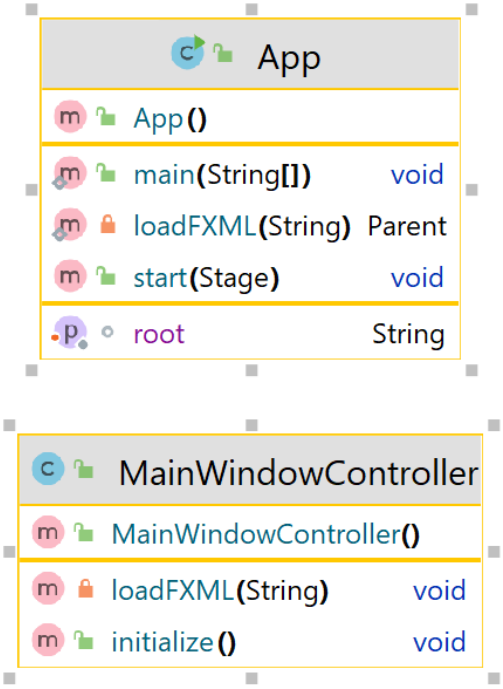

Figure 4: UML diagram for the selection screen

4.3. First Screen:

We would choose to apply OpenWeather API for this screen to demonstrate the up-to-date weather data.

Components:

- View: JavaFX UI with latitude and longitude input fields, a "Fetch" button, and a display area for weather data.
- Controller: Handles user input, communicates with the API, and updates the view.
- Model: Represents weather data received from the API.

To demonstrate our idea, we would use a UML diagram, as in Figure 5. Moreover, discussing further development, we would like to apply one more API for generating the map for extracting the latitude and longitude. And integrating with OpenStreetMap could be a good choice for us. It will be implemented in the next phase.

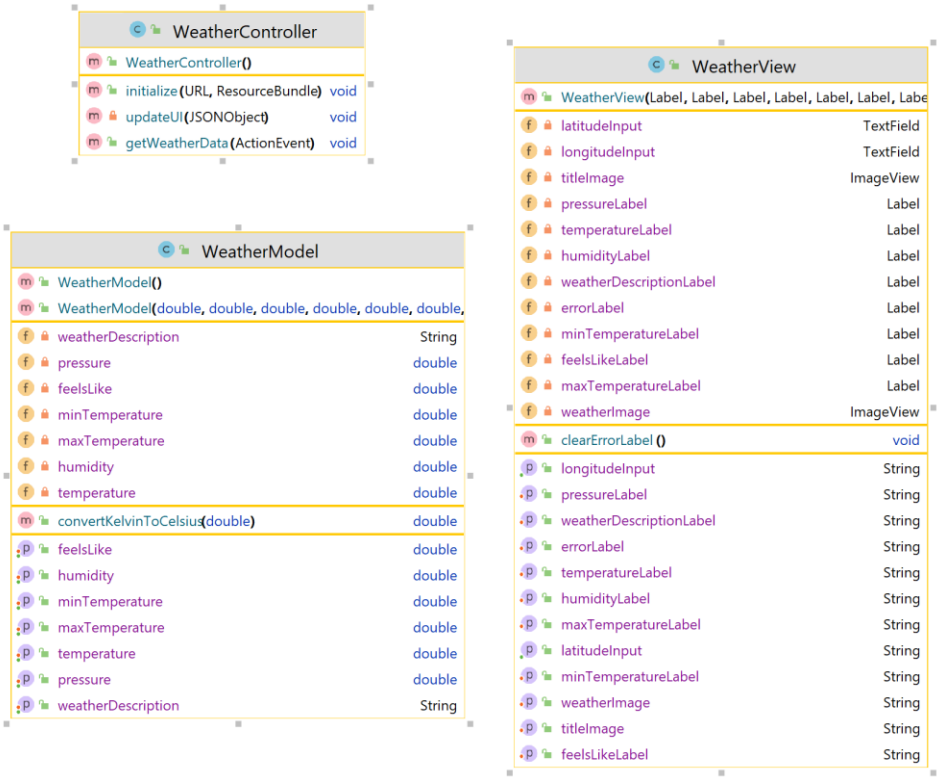

Figure 5. UML presentation for the First Screen necessary components

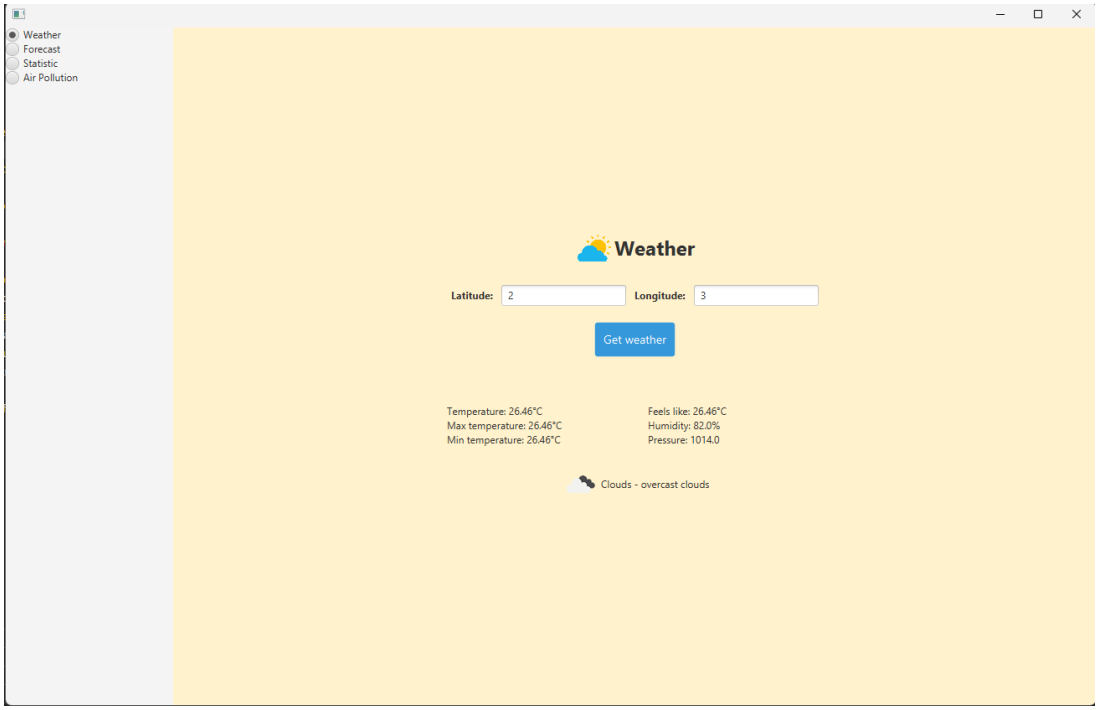

Figure 6: GUI layout for the first screen when fetching the data

4.4. Second Screen:

API Links:

- https://openweathermap.org/api/hourly-forecast
- https://openweathermap.org/forecast16
- https://openweathermap.org/api/geocoding-api
- https://openweathermap.org/api/statistics-api

Components:

- View: JavaFX UI with options to select either 4 days, 16 days, or 365 days forecast, a list view to display forecast data.
- Controller: Handles user input, communicates with the API, and updates the view.
- Model: Represents forecast data received from the API.

The beginning approach of the second screen is represented in Figure 7

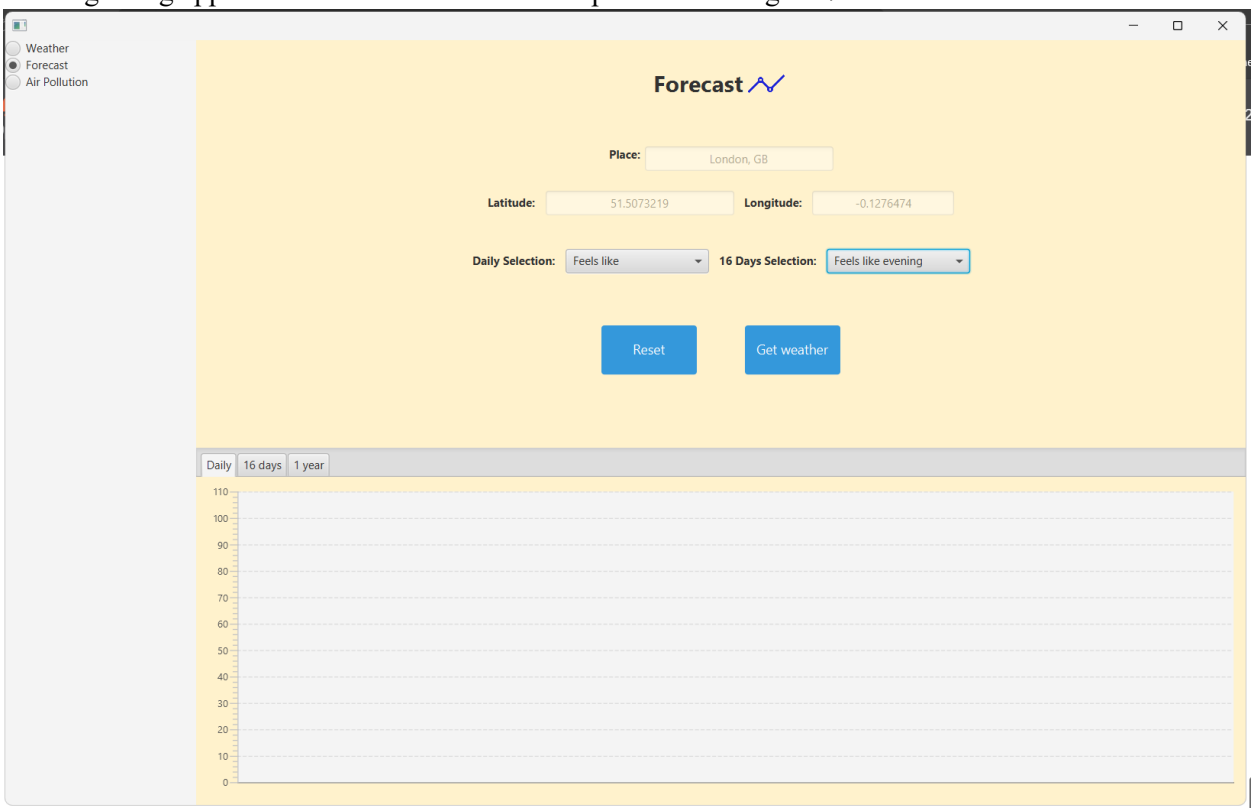

Figure 7. Sketching of the second screen

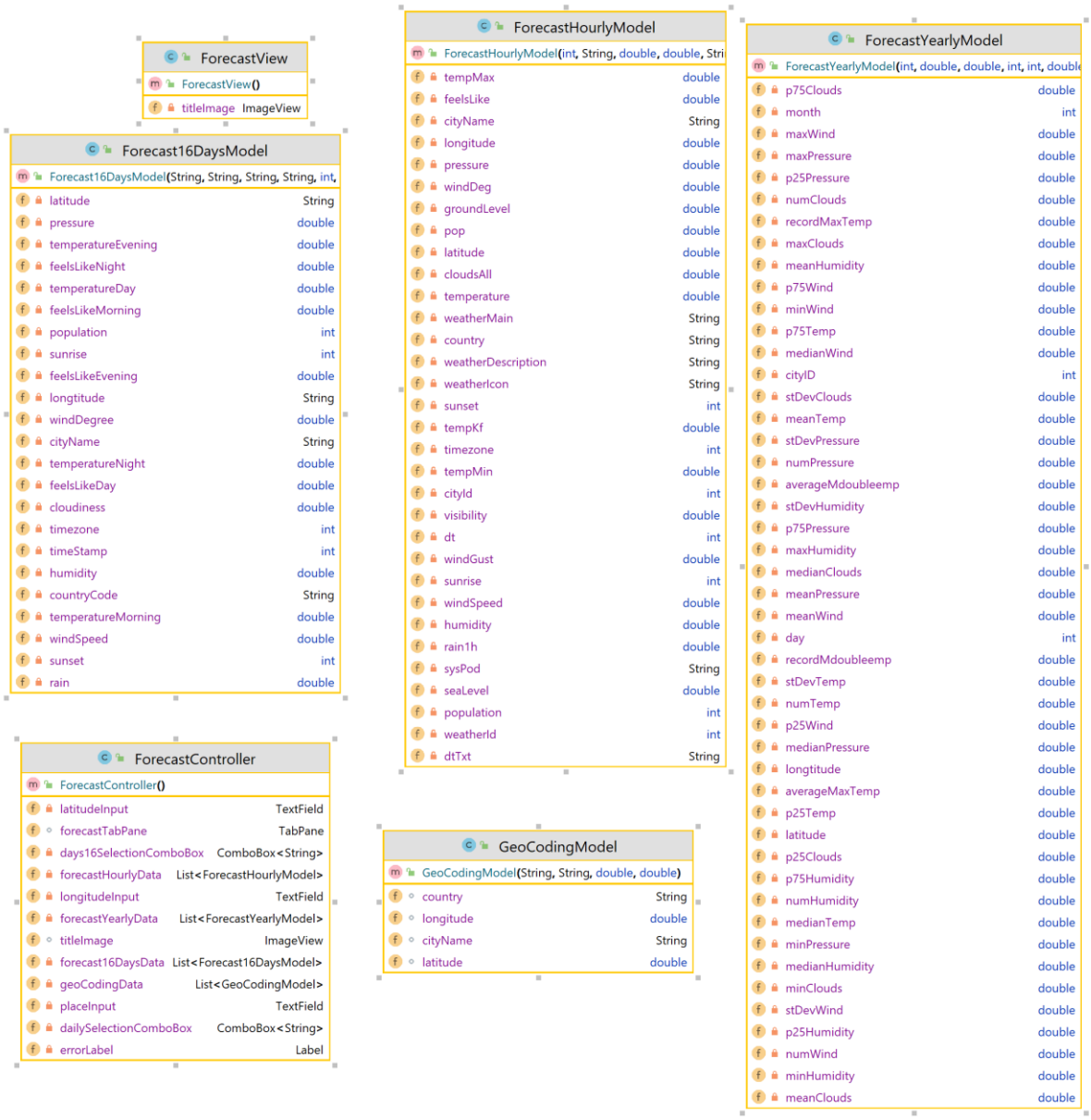

Figure 8: UML representation (implementation phase, excluding getters and setters methods) related to the second screen

4.5. Third Screen:

API Link: https://openweathermap.org/api/air-pollution

Components:

- View: JavaFX UI with latitude and longitude input fields and a display area for air pollution forecast data.
- Controller: Handles user input, communicates with the API, and updates the view.
- Model: Represents air pollution data received from the API.

We would use the sketch and the UML diagram to demonstrate our idea precisely

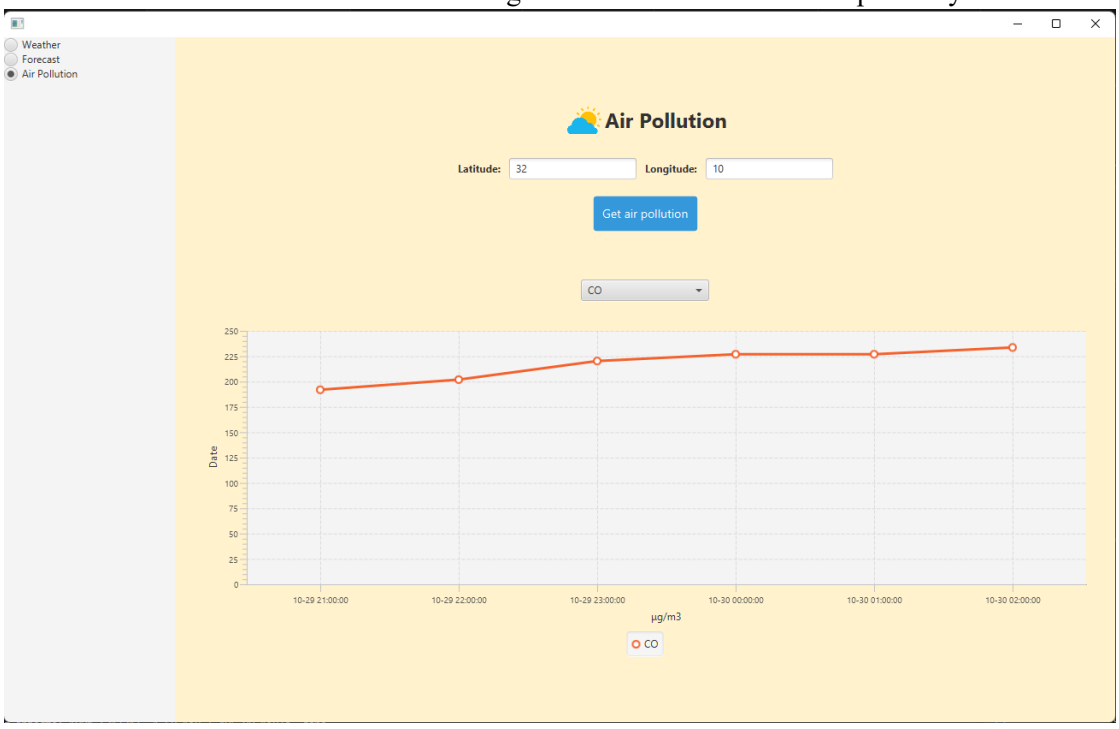

Figure 9. Sketching of the third screen

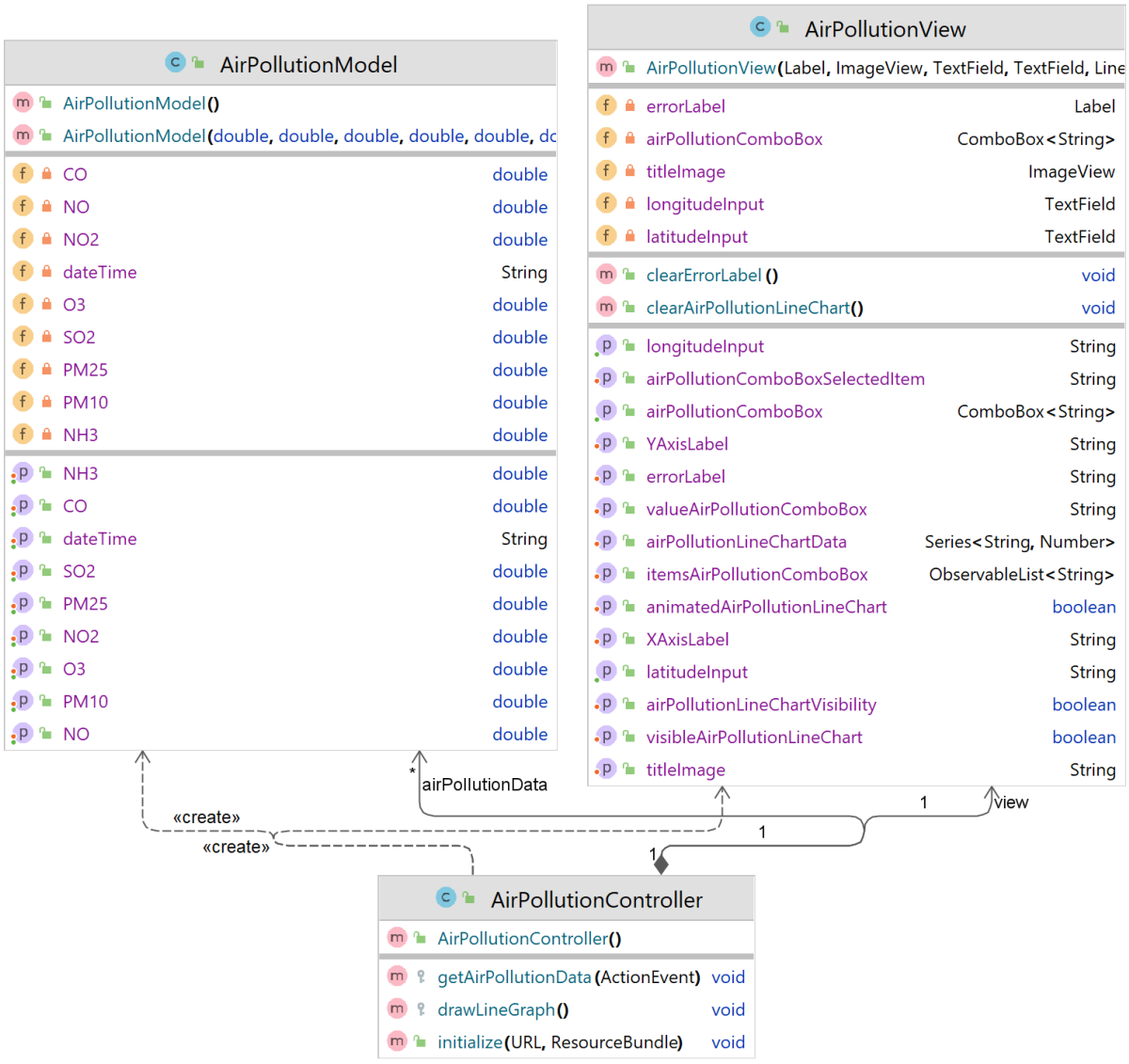

Figure 10: UML representation (implementation phase) related to the third screen

5. API Integration and Usage

Discussing API Integration, we would create separate classes that implement the data provider. These classes will interact with the 5 OpenWeather APIs and return the data required on each screen. Furthermore, we would like to apply the interaction between these five APIs, which can be supported for the missing deficiency of each APIs and demonstrate the convenience we would like to bring for the users.

Discussing the usage of the API, we would demonstrate the main idea for each screen as follows:

- For the First Screen, you will use the "Current Weather Data" API to fetch current weather data based on latitude and longitude. In the future, we will apply the location tracking for converting into longitude and latitude and in the reversed direction.

- For the Second Screen, you will use the "Hourly Forecast 4 days", "Daily Forecast 16 days", and the “Yearly Statistic Forecast” APIs depending on the user's choice. Therefore, based on the forecast, we could use it to demonstrate the graphs as a function of a range of times.
- For the Third Screen, you will use the "Air Pollution" API to fetch air pollution data based on latitude and longitude. In the last phase, we would like to apply location tracking besides tracking according to the coordinates.

6. Design Patterns:

We would consider using the following design patterns:

- Singleton Pattern: Implement singletons for data provider classes to ensure only one instance exists.
- Factory Pattern: Create a factory to generate instances of data provider classes based on user input (e.g., forecast type).
- Observer Pattern: Use this pattern for communication between controllers and views to update the UI when data is fetched.