COMP.SE.110 Software Design (Spring 2022)

DESIGN DOCUMENT

Group SWD
Duy Vu, Hung Anh Pham, Safwane Benbba, Sang Nguyen

TABLE OF CONTENTS

INTRODUCTION

This document provides an overview of the application's architecture, components, and the different relationships between them. The application visualizes the changes in different greenhouse gas metrics in the time ranges and locations chosen by the user. Data is fetched from SMEAR (Station for Measuring Ecosystem–Atmosphere Relations) and STATFI (Statistics Finland). Users can use the application to easily choose what to visualise, analyse and compare; user input preferences can also be saved in the local filesystem to be loaded after the application restarts.

Chapter 2 discusses the design pattern the application follows, and how it ties to the specific implementation. Chapter 3 discusses the components of this application: an interface overview is given alongside the general data flow. Each component's details, and other external libraries would also be discussed in this section. Chapter 4 contains references that this document uses.

ARCHITECTURE

Qt provides a set of classes that use a model-view architecture to handle the flow of data and its presentation to users. Since PyQt has been chosen as the graphical user interface library for this project, it is designed with a model-view architecture in mind which is similar to the MVC design pattern but combines the view and the controller. [1]

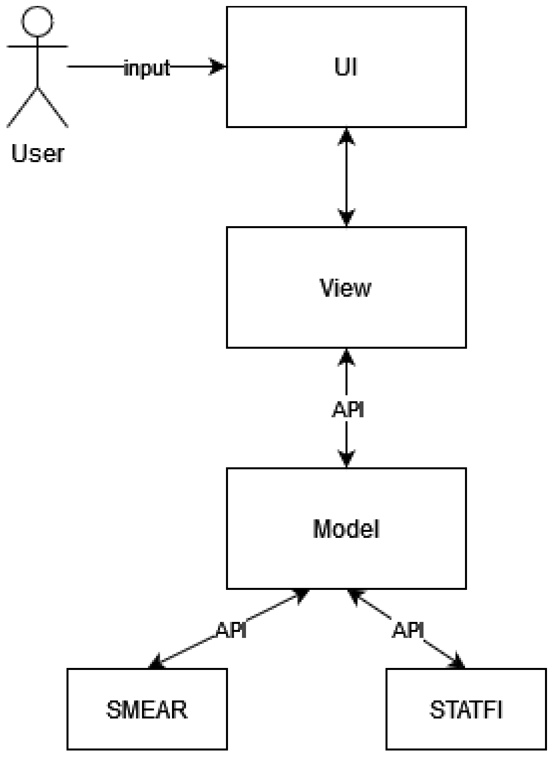


Figure 1. High level architecture

The model-view architecture still segregates the way the data is stored within the application and how it is displayed to the users but provides greater flexibility when handling user inputs.

The UI, which is integrated into the view, handles user input and the presentation of the data. The view communicates with the model through various interfaces to send and receive data, which in turn gets its own original data from the APIs provided by SMEAR and STATFI.

COMPONENTS

This section discusses the components in the project. An overview of all components, interfaces, the relationships between them and the data flow is given below. Details of each component and its respective interface are given in later sections.

Components overview

Figure 2 below presents an overview of the current components of the software, and the relationships between their interfaces. The details regarding eachinterface are discussed in later sections.

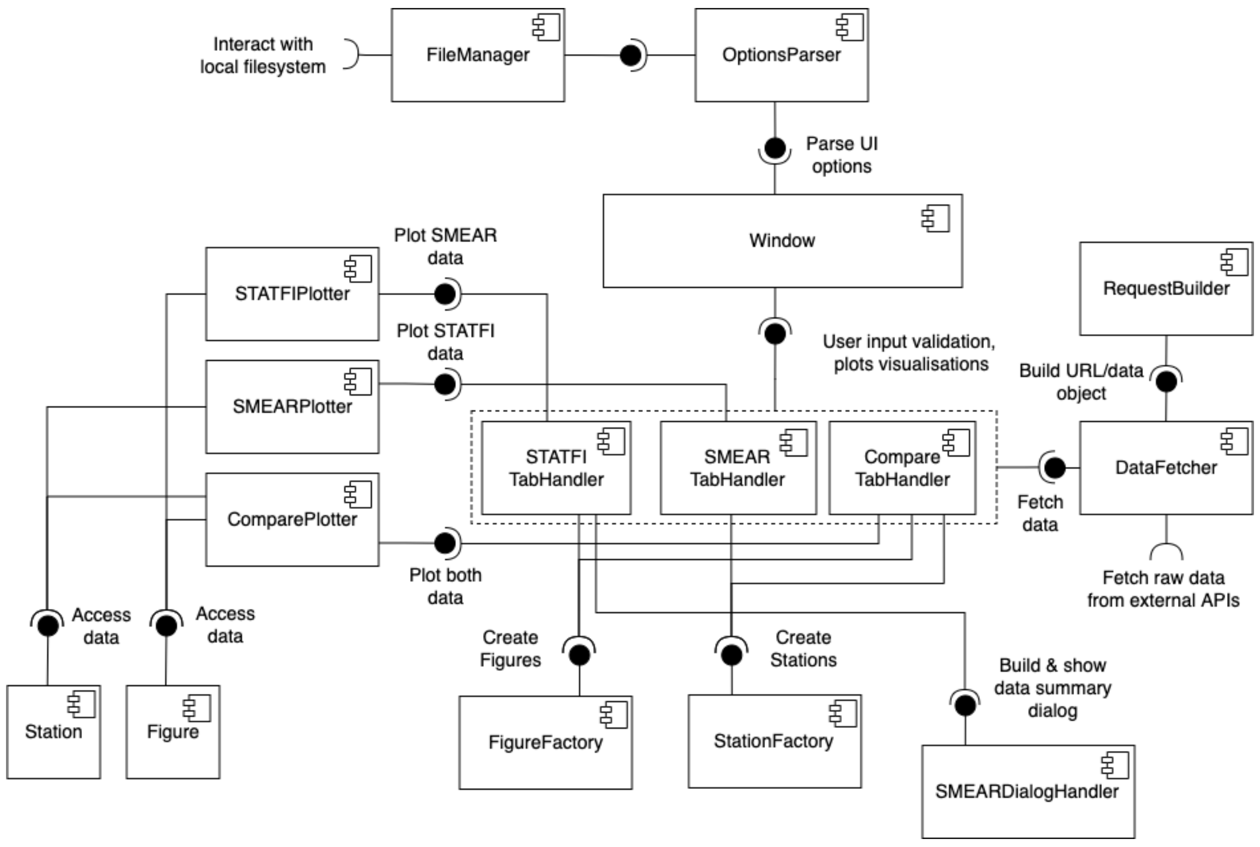


Figure 2. Interface diagram of the product

As discussed in Chapter 2, the MVC pattern was chosen for our application. However, since PyQt merges the View and the Controller components into one, let us refer to our architecture as the Model-View pattern. From the figure above, the User interface (Window) is the View, while the rest are parts that make up the Model. The User interface delivers visuals and handles user input, which is sent to the Model components to be processed according to business logic. After this, the resulting visualizations are drawn on the User interface, and shown to the user.

Figure 4 describes the data flow between components. Starting with the input from the user, different components of the system work and communicate with each other to visualize information on the user interface, as well as interacting with the local filesystem. Notice the blocks in the diagram which are colored light blue – these stand for the different types of blocks for each use case tab (SMEAR, STATFI, and Compare). To simplify the diagram, we decided not to draw each and every one of them due to their shared similarities.

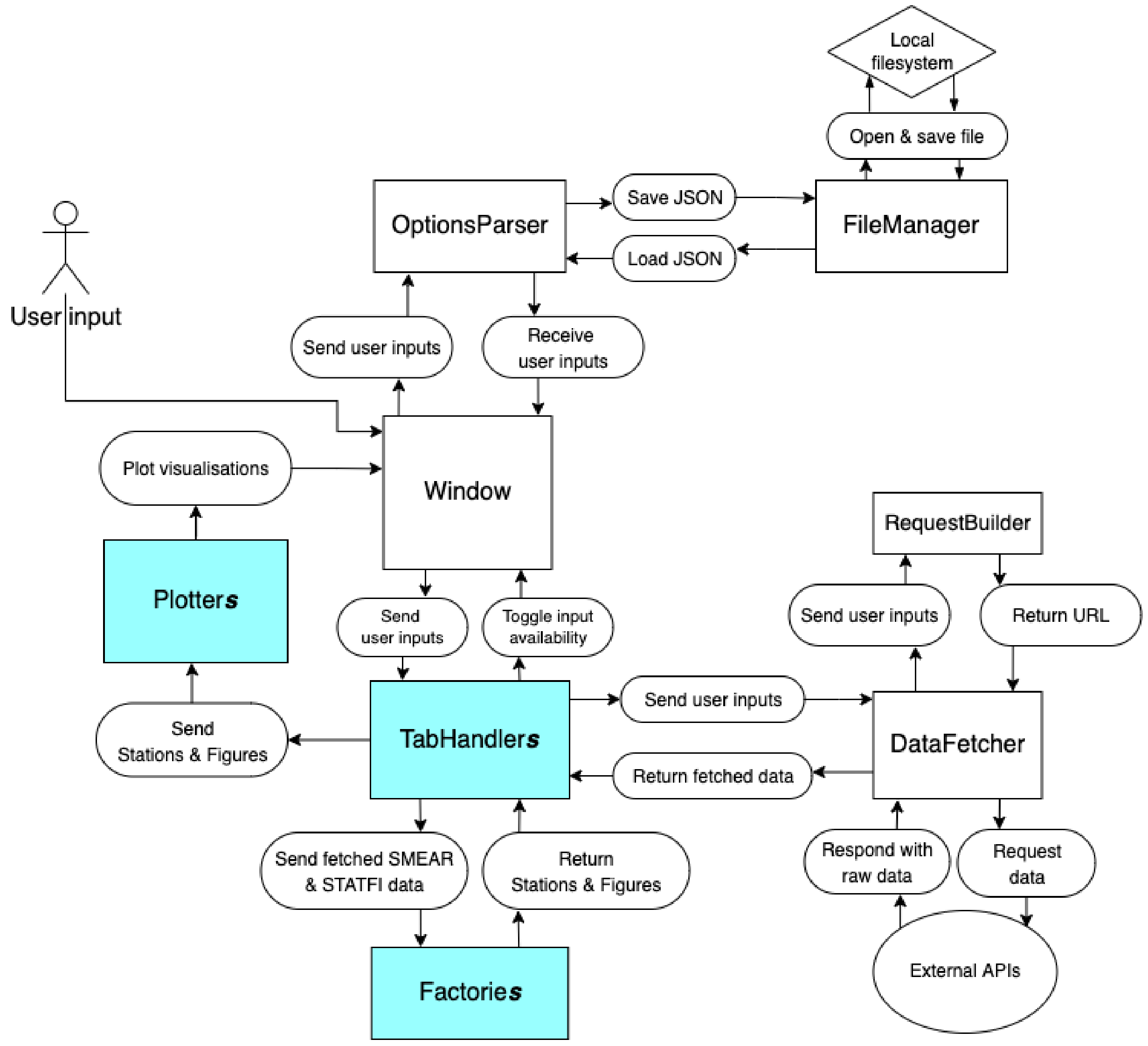
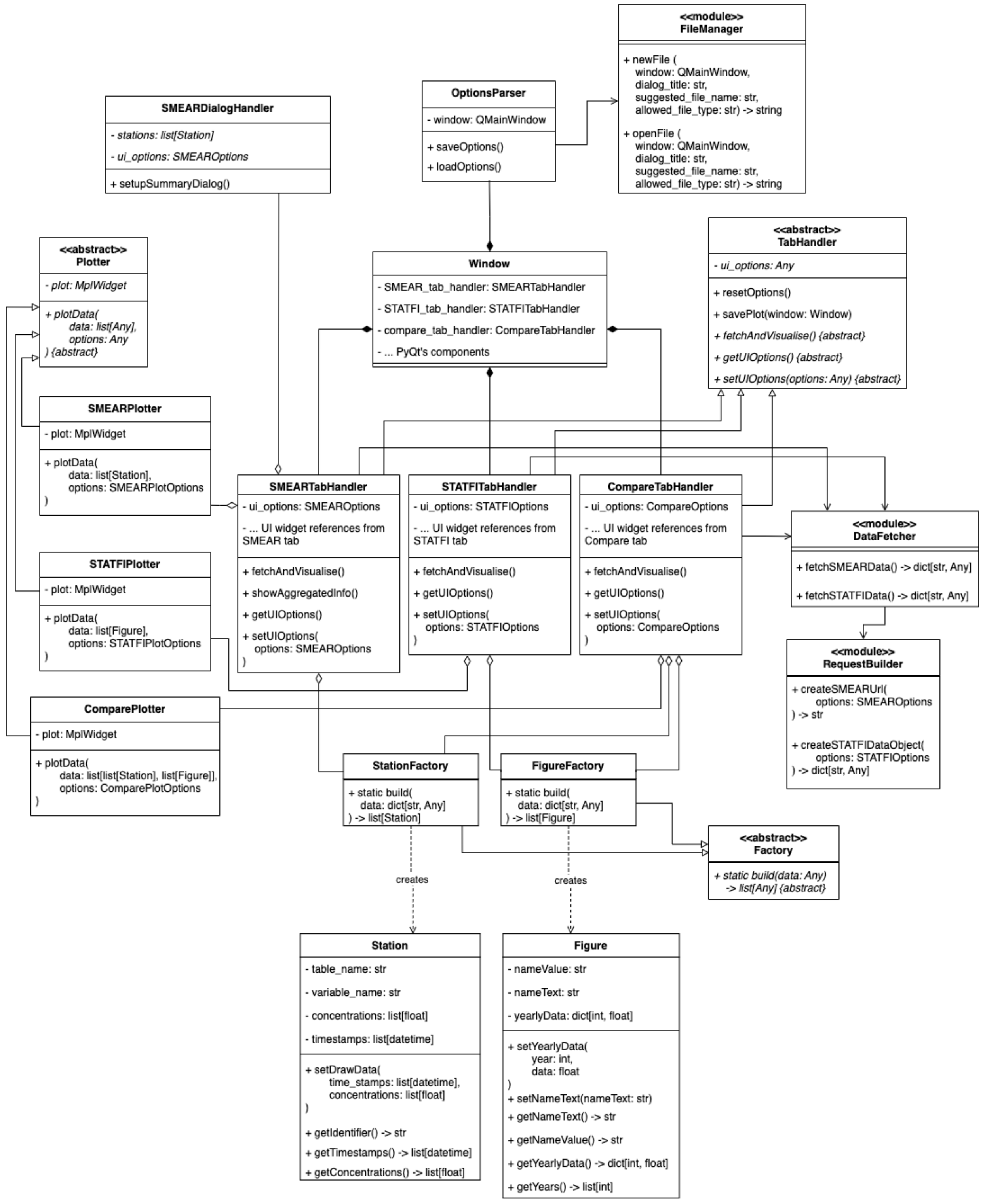


Figure 3. Data flow diagram

Figure 6 shows a comprehensive diagram of the classes and modules used in the project along with the relationships between them. A more detailed description of each class and module is included in the next chapters.



<<module>>
FileManager
+ newFile (window: QMainWindow, dialog_title: str, suggested_file_name: str, allowed_file_type: str) -> string
+ openFile (window: QMainWindow, dialog_title: str, suggested_file_name: str, allowed_file_type: str) -> string
OptionsParser
- window: QMainWindow
+ saveOptions()
+ loadOptions()
SMEARDialogHandler
- stations: list[Station]
- ui_options: SMEAROptions
+ setupSummaryDialog()
<<abstract>>
Plotter
- plot: MplWidget
+ plotData(data: list[Any], options: Any) {abstract}
Window
- SMEAR_tab_handler: SMEARTabHandler
- STATFI_tab_handler: STATFITabHandler
- compare_tab_handler: CompareTabHandler
- ... PyQt's components
<<abstract>>
TabHandler
- ui_options: Any
+ resetOptions()
+ savePlot(window: Window)
+ fetchAndVisualise() {abstract}
+ getUIOptions() {abstract}
+ setUIOptions(options: Any) {abstract}
SMEARPlotter
- plot: MplWidget
+ plotData(data: list[Station], options: SMEARPlotOptions)
SMEARTabHandler
- ui_options: SMEAROptions
- ... UI widget references from SMEAR tab
+ fetchAndVisualise()
+ showAggregatedInfo()
+ getUIOptions()
+ setUIOptions(options: SMEAROptions)
STATFITabHandler
- ui_options: STATFIOptions
- ... UI widget references from STATFI tab
+ fetchAndVisualise()
+ getUIOptions()
+ setUIOptions(options: STATFIOptions)
CompareTabHandler
- ui_options: CompareOptions
- ... UI widget references from Compare tab
+ fetchAndVisualise()
+ getUIOptions()
+ setUIOptions(options: CompareOptions)
<<module>>
DataFetcher
+ fetchSMEARData() -> dict[str, Any]
+ fetchSTATFIData() -> dict[str, Any]
STATFIPlotter
- plot: MplWidget
+ plotData(data: list[Figure], options: STATFIPlotOptions)
<<module>>
RequestBuilder
+ createSMEARUrl(options: SMEAROptions) -> str
+ createSTATFIDataObject(options: STATFIOptions) -> dict[str, Any]
ComparePlotter
- plot: MplWidget
+ plotData(data: list[list[Station], list[Figure]], options: ComparePlotOptions)
StationFactory
+ static build(data: dict[str, Any]) -> list[Station]
FigureFactory
+ static build(data: dict[str, Any]) -> list[Figure]
<<abstract>>
Factory
+ static build(data: Any) -> list[Any] {abstract}
creates
creates
Station
- table_name: str
- variable_name: str
- concentrations: list[float]
- timestamps: list[datetime]
+ setDrawData(time_stamps: list[datetime], concentrations: list[float])
+ getIdentifier() -> str
+ getTimestamps() -> list[datetime]
+ getConcentrations() -> list[float]
Figure
- nameValue: str
- nameText: str
- yearlyData: dict[int, float]
+ setYearlyData(year: int, data: float)
+ setNameText(nameText: str)
+ getNameText() -> str
+ getNameValue() -> str
+ getYearlyData() -> dict[int, float]
+ getYears() -> list[int]

Figure 4. Class diagram

TabHandlers

SMEARTabHandler, STATFITabHandler, and CompareTabHandler are classes responsible for managing the flow of data from user inputs into visualization charts in the application. They inherit from the abstract class TabHandler, which defines methods responsible for most of the tab's functionality: user input management and visualization.

Each TabHandler stores references to its respective tab's UI components, from which user inputs can be collected and sent to be processed in other components. User input validation is also done in the TabHandlers, so that the application will not fetch invalid data. TabHandlers, therefore, are the central point for validating user input, process them, and plot the resulting visualizations (using Plotters, which are initialized when a TabHandler is created).

DataFetcher

As the name implies, this class handles data fetching for the application. It requests raw data based on users' options and returns the API's response in JSON format (dictionary in Python).

This is a static class, as it only contains static methods: fetchSMEARData() and fetchSTATFIData(). Each of these methods takes in user's input, requests the necessary information from the APIs, parses the JSON response and returns it.

RequestBuilder

RequestBuilder is not a class, but a Python module (collection of methods) that takes necessary user inputs to build and return an appropriate URL or data object containing necessary parameters to fetch data.

SMEAR data can be fetched with a GET request, given an URL with all necessary query parameters. RequestBuilder would therefore build such a URL using createSMEARUrl(), using the user input options passed to it as parameters.

STATFI data, on the other hand, can be fetched with a POST request using a JSON data object. RequestBuilder then builds a Python dictionary using user input options using createSTATFIDataObject() and returns it.

Station

This class holds necessary data from SMEAR for each station. The station contains identifiable information, such as its table name and variable name in the SMEAR API, as well as the fetched greenhouse gas data (timestamps and concentrations)

While it is very intuitive that each Station stores information of different greenhouse gases in an i.e., dictionary, our group decided that we would only store information of only one greenhouse gas in a Station object at a time. This is because when choosing greenhouse gas information to be visualized from SMEAR, only one gas is allowed to be chosen. Each time SMEAR data is fetched, we create new Stations. This means that under no circumstances could there be a Station storing data of different gas types at once.

Figure

Figure contains data to be drawn from STATFI data. Each object corresponds to a STATFI “Tiedot”. Like Station, identifiable information of each Figure is stored, such as its name and ID from the STATFI API, and a dictionary containing its data spanning multiple years.

Factories

StationFactory and FigureFactory, both inherited from abstract class Factory, are responsible for creating Station and Figure objects from fetched SMEAR and STATFI data, respectively.

Both Factory classes have a method, build(), that takes care of creating a list of objects, either Station or Figure from data passed as a parameter.

Plotters

SMEARPlotter, STATFIPlotter and Compare Plotter handle all drawings of the SMEAR, STATFI, and Compare tab, respectively. They take Station and Figure objects created by the Factory classes, extract necessary information from them and plot graphs/charts on the application UI.

Its method plotData() takes care of this. Each class stores a reference to its respective tab’s plot canvas. Data from Figures/Stations will then be drawn on the canvases directly using Matplotlib.

OptionsParser

The responsibility of this class is saving current user’s UI options to the local storage, so that they can load it back and give the UI the same options after restarting the application.

OptionsParser would collect the user input options from all 3 tabs and save it to a JSON file using method saveOptions(). This JSON file can be loaded back, using loadOptions(), after which the UI options are applied back to the user interface (Window).

FileManager

The FileManager module communicates directly with the local filesystem to save and open JSON files. OptionsParser uses FileManager to save parsed user options or load it when needed.

Using PyQt’s interface, it lets users browse directories, save files with appropriate naming without any complicated configurations from developers. FileManager does not need to know about how the program works at all, its job is simply to connect to the folders/files on the local machine.

Window

The component contains everything that deals with user interface of the application. Window is inherited from Ui_main_window, which contains code generated from the .ui files which are Qt Designer files. As it only concerns the UI, Window does not offer any public interface.

SMEARDialogHandler

As its name suggests, this class is only used in the SMEAR tab. It's responsible of creating and handling the data summary dialog, where SMEAR data is aggregated (min, max, average). This class receives SMEAR stations' data and selected user options and process them to populate the Qt Dialog with appropriate data. Its public interface only contains function setupSummaryDialog(), which populates and renders the dialog on screen.

External libraries

The chosen libraries are absolute minimum for this project so far as we need simple, familiar libraries for building a desktop application and GUI, making HTTP requests, and visualizing data. Furthermore, we prioritize quick rollout of application.

- PyQt6 is used to create the GUI for application. [2]
- Requests is used to make HTTP requests to get SMEAR and STATFI data. Not only is it simple, human-friendly tool for developer to make HTTP requests, but it is also the most downloaded Python library.
- Matplotlib is used to plot data on the GUI. It is a simple object-oriented API for embedding plots into applications using general-purpose GUI toolkits like PyQt. [3]
- Numpy is used for mathematical operations necessary to build visualisations. [4]

DESIGN DECISIONS

In this chapter, we aim to provide some insights into why the application architecture is designed the way presented above.

Division of responsibilities between components is the main driving force of the application architecture. We try to follow the Single Responsibility Principle (SRP) so that each class is clearly responsible for one purpose, and when a class's functionalities expands beyond its scope, a new class is designed. The data flow diagram illustrates this well, as data flows through each component in a clear manner, each component is named so that its functionality is obvious just from reading it: the DataFetcher only fetches the data (even the URL to fetch is build by another module), the Factories only builds objects, the Plotters only plot the data in the charts, etc. There is an exception, however, with the TabHandlers, whose responsibilities is to handle the general communication in each tab and control the tab's UI components. Their responsibilities lead to their implementations being more complex than other classes, but we consider this unavoidable, as we could not find a way to separate these classes further without introducing some complication overhead. An idea would be to separate the user input validation part to another class, but this poses a great challenge as data validation goes hand in hand with UI components manipulation, which means the separation cannot be done properly.

Some simple inheritance hierarchy is adopted to simplify the architecture when multiple use cases are concerned. The Liskov Substitution Principe (LSP) is followed very carefully here. By making sure the parameters and return types are consistent throughout the application's codebase, we can safely take advantage of the inheritance pattern to simplify our application and make the development experience much easier. As we have three completely separate tabs (corresponding to different use cases), but some common behavior is shared among them, having a parent class not only provides a general interface of what functionalities are needed to implement but also, if the three individual classes share a same method, it could be moved to the parent class. For example, all three tabs have the Reset functionality, which resets the user options to the currently plotted state. The method responsible for this, therefore, could be moved to the parent class, as its implementation is the same across all TabHandlers.

The Factory pattern is utilized thanks to the dynamic nature of the data APIs. To make the application easier to understand and more scalable, our group treats the numbers/data fetched from the APIs as objects, and this is reflected in the visualizations as well. Each line representing a SMEAR station is considered an object, each bar representing a STATFI Tieto is considered an object. The project's requirements to properly handle errors and perform data aggregations mean that we cannot plot the numbers fetched from APIs directly. We need a dynamic way to manage data. Using the Factory, we can easily filter data and create objects from them. These objects are much easier to be reused, evaluated, passed as arguments, etc.; effectively make the application a lot more logical and less error prone. The Factory pattern here also helps separating the components' responsibilities further.

The Mediator pattern is used to manage communication & responsibilities between the different components. TabHandlers are the Mediators in our application. Not only does it mediate the communication between Plotters, Fetchers, Factories, etc. but also between different UI components in the tabs. The communication between UI components are a lot more complex then we expected, due to the requirements that users do not have to fetch data before realizing that it's not available. This means certain UI components must be disabled, greyed out, reduced, re-enabled, etc. when user inputs change in other components. Instead of letting all these components talk to each other directly, creating a nightmarish web of signals and slots, we enforce communication through TabHandler in separate and clear methods. This makes the code a lot clearer than it would otherwise.

A final point regarding the programming language. Python is chosen since it's a lot less verbose compared to Java, and we don't have to deal with pointers in C++. However, this also means that we cannot enforce type safety, making the development process problematic in the long run, and make it harder to follow SOLID principles. To combat this, we use Python's type hints and the mypy external package, which helps a lot by giving type warnings and hints when necessary. Mypy also helps us to follow the Liskov Substitution Principe, as it warns when parameters/return types are inconsistent between parent and child classes. We also follow the recommended Python coding convention, having both code linters and formatters to make sure we comply with high coding standards.

SELF-EVALUATION

The architecture of the application has greatly changed during the development of the project. This is what we expected from the beginning, as we do not attempt to have a deep understanding of the requirements and design the perfect architecture from the get-go. As we go along with the iterative development process, we start from a simple architecture that enables things to work, and slowly evolve into an architecture that allows flexibility and scalability. In other words, the "original plan" was always intended to be changed, or even trashed completely as the application being developed.

The original design of the application can be seen in the data flow diagram below. There are some differences compared to the latest architecture. Firstly, the division of responsibilities is not clear, as not much thought has gone into the original design. The data flows from the main application Window to the DataFetcher, then being parsed by the DataParser (creating Stations and Figures), then forwarded back to the Window to be plotted by the Plotter. The TabHandler does not exist in this architecture. In a sense, this is a simpler architecture, but certain features cannot be implemented easily, for instance checking data availability before fetching, SMEAR data aggregation, etc.

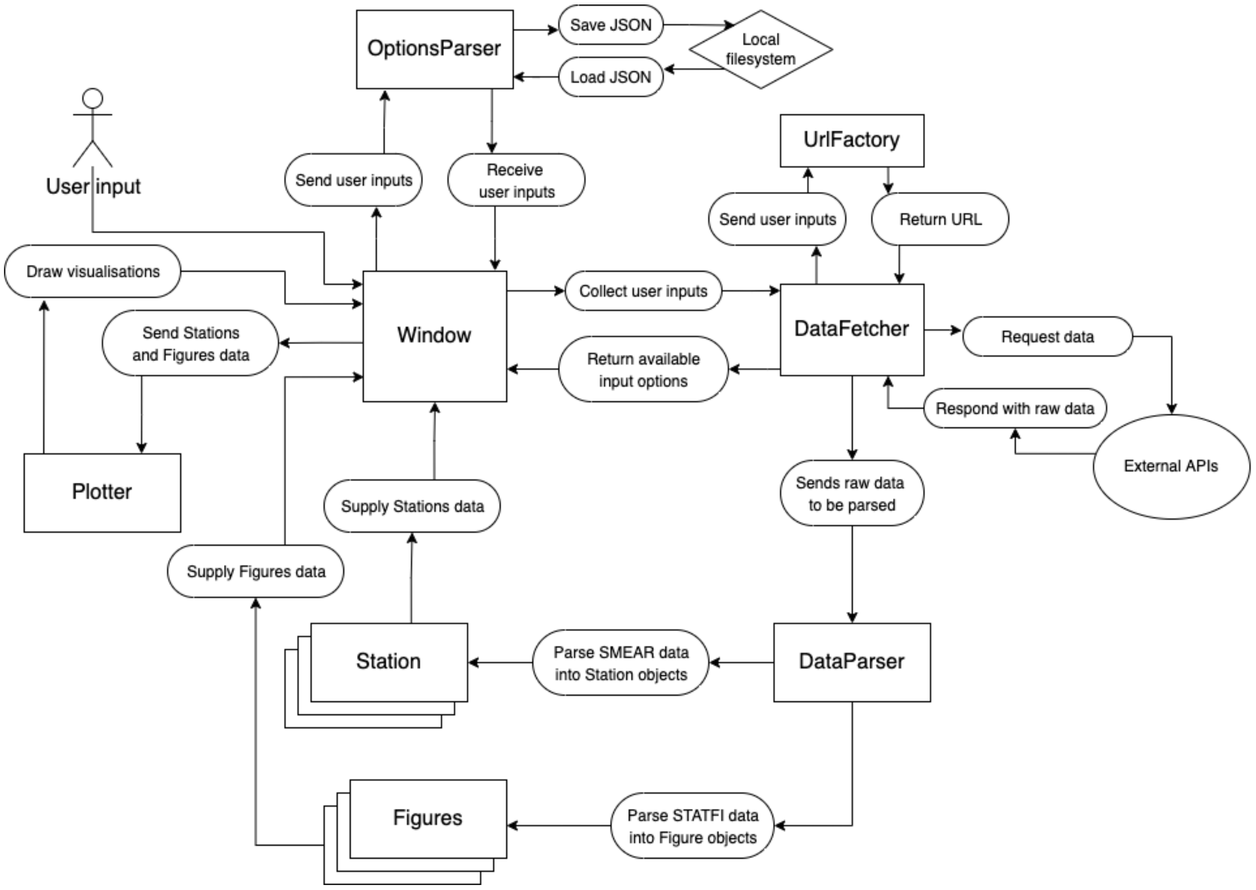


Figure 5. Original architecture

The final architecture introduces the TabHandlers as the main controllers of the application. This means that the Window class, which is only supposed to be the UI component class, no longer has to perform complicated data processing/UI manipulation tasks. The Factory pattern is also introduced to help clarify the architecture and data flow. Class inheritance is introduced to make developing functionalities for different tabs more straightforward. In retrospective, we do not think the architectural changes we made are super critical in making the app work, but their impacts on clarity, app flexibility, and the general development experience are certainly significant. We might have been able to implement all requirements using the original architecture, but the results will be nowhere near as good as the solution we present in the end.

REFERENCES

[1] “Model/View Programming,” The Qt Company, [Online]. Available: https://doc.qt.io/qtforpython/overviews/model-view-programming.html. [Accessed 19 February 2022].

[2] “Reference Guide -- PyQt Documentation v6.2.1,” The Qt Company, [Online]. Available: https://www.riverbankcomputing.com/static/Docs/PyQt6/. [Accessed 19 February 2022].

[3] “Matplotlib -- Visualization with Python” [Online]. Available: https://matplotlib.org/. [Accessed 19 February 2022]

[4] “NumPy” [Online]. Available: https://numpy.org/. [Accessed 19 February 2022]