Final Design Document

Software Design
COMP.SE.110
Group - CSGR

Version history

Version	Date	Author	Description
0.1	10.02.2022	Entire team	First draft
0.2	15.03.2022	Entire Team	Mid-term Document
0.3	20.04.2022	Entire Team	Final Document

Table Of Contents

Introduction

Purpose of the Document and Project.

The purpose of the document is to explain how the design has happened during the project planning process and in further implementation. The goal of the project is to build a stand-alone application for monitoring real-time data on greenhouse gases and comparing current data with historical data on greenhouse gases.

Product and Environment

The product is standalone application software which can enable greenhouse gases monitoring. The team has decided to develop the product using the Java programming language. The project also uses JavaFX library for the most part of implementation. The team has decided to use IntelliJ as the IDE and Gitlab for version control management.

Team Members

The project will be done by four team members Cecylia Borek, Gaurab Mahat, Roger Wanamo and Sai Polineni.

Related organization

The project is assisted by Ville Heikkilä. Ville is one of the Teaching Assistants of the course. Responsible for guiding the project management towards the correct direction and providing assistance and answers when needed.

High Level Description

The application consists of several main parts. Query is responsible for creating all UI components specific for the data source (dates for Smear, years for Statfi, etc). It creates a ResultView, with result for the specific query. It also stores all the values selected by the users and enables creating HTTP request from them. The QueryClient uses those requests to call APIs and fetch data. The data obtained in Json form is converted into a common Result class, which is then fed to GraphDataManager. The GraphDataManager stores all fetched data in observable containers. This way the ResultView changes immediately to reflect the changes in the GraphDataManager.

A very high-level overview of the classes, their dependencies and connections is presented in Figure 1.

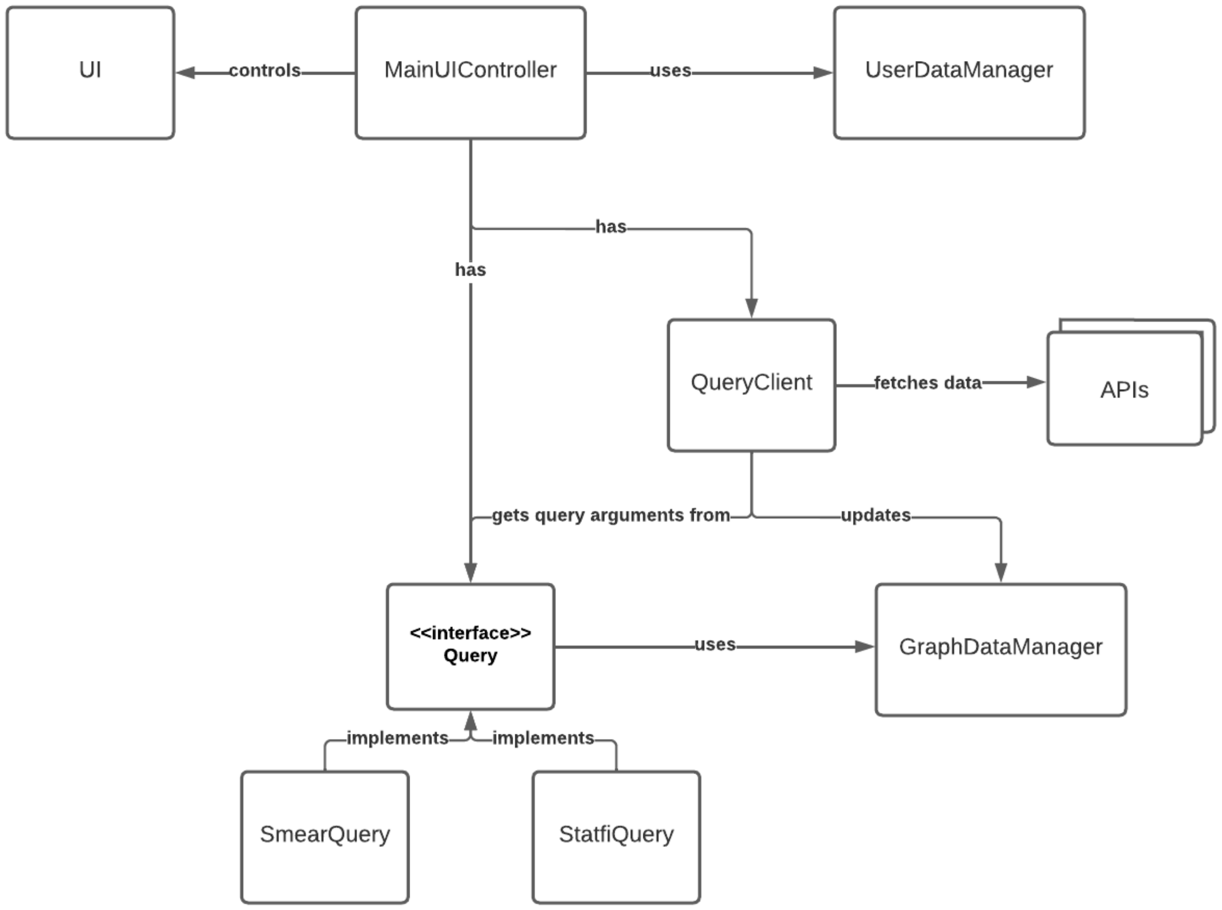

Fig 1. High level system architecture.

Components, Responsibilities, Boundaries and Interfaces

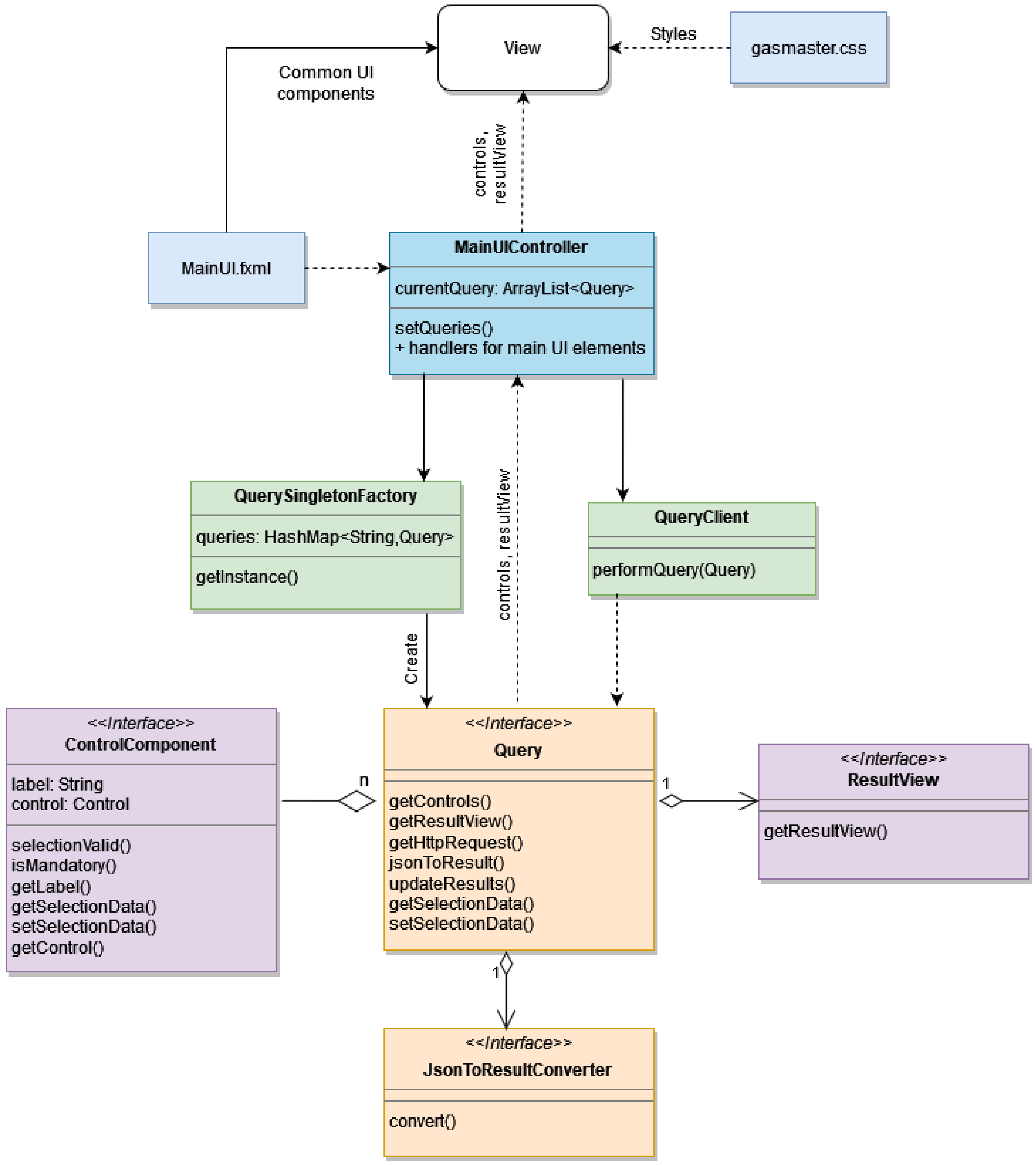

Figure 2. High level overview of flow between query interfaces and view

Composing the view

MainUI.fxml defines the main interface and common elements, including the menu, dropdown to select query type, and buttons to show, load and save queries. The fxml also allocates space for query components and displaying the results. All controls specific to a query, as well as the view displaying the results, are provided by the Query object itself, through the MainUIController.

Styles for everything in the view are defined in gasmaster.css.

Building the UI this way allows us to redesign and style everything without touching the queries. As long as the fxml allocates space for controls and resultsview, we can redefine the look and feel of everything on the screen and use the new look with all queries.

Query interface

The Query interface defines the central component for performing queries. A class implementing the Query interface must define:

- Controls needed to perform the query
- How to convert control input to HttpRequest
- How to convert HttpResponse to ResultList object
- What kind of view to use to display the results

Converting Json response to results is outsourced to another class, that must implement the JsonToResultConverter interface.

Queries are created using a singleton factory. The factory will return an existing Query object, or create a new Query object if that type of Query object hasn't been created yet. Classes implementing the Query interface should have a protected constructor, to make sure it isn't called by anything but the factory in the same package. This way queries are initialized only once (initialization might include API-calls retrieving data for control components) and existing queries, query results and search settings are remembered when switching between query types.

When the Query object is created, it creates a list of all ControlComponents it needs, as well as an ResultView, which are returned to the MainUIController, that formats them and adds them to the screen. To perform a query, the Query-object is sent to the QueryClient, which calls the required functions in order.

ControlComponent interface

ControlComponents consist of a label and a Control, with functionality to get and set the selection data in the control. Currently ControlComponents are defined for DatePicker, YearPicker and both single- and multichoice dropdowns. ControlComponents for other types of input controls can easily be added.

The Query object groups controls into a ControlPanel object, that offers functions to getSelectionData for the entire control panel for saving the query, as well as setting selection data when loading.

Json to Result Converter

To store the fetched results in a common form, generic classes SGResult and ResultList were created. The SGResult stores a hash map of dates and corresponding values for a specific station and a specific gas. The ResultList stores all SGResults that were obtained during ome data fetch. It has functions that allow to extract from it the SGResult for a specific gas and station. In the case of Statfi where there are no stations, the station variable is set to a specific common value like "Statfi".

To extract the needed data from the JSONs, GSON library was used. It allows to convert a JSON to a Java object. Because the JSONs are different for each API (structure, naming, etc), a separate converter for each of the APIs was needed.

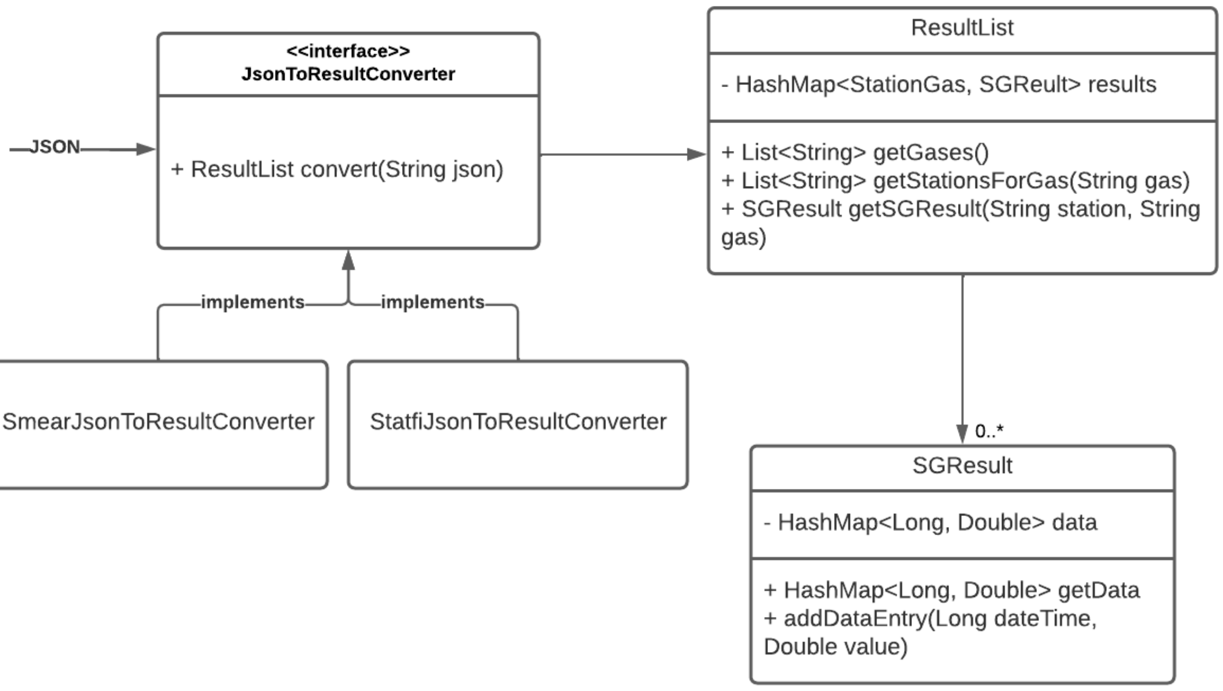
JSON
<<interface>>
JsonToResultConverter
+ ResultList convert(String json)
implements
implements
SmearJsonToResultConverter
StatfiJsonToResultConverter
ResultList
- HashMap<StationGas, SGReult> results
+ List<String> getGases()
+ List<String> getStationsForGas(String gas)
+ SGResult getSGResult(String station, String gas)
0..*
SGResult
- HashMap<Long, Double> data
+ HashMap<Long, Double> getData
+ addDataEntry(Long dateTime, Double value)

Fig. 3. JsonToResultConverter diagram

ChartView Interface

ChartView gets a VBox containing a chart that represents the grouped data. The acquired VBox can contain a chart with a different chart type. It can be used to represent the data in a line chart or bar chart and allows the flexibility to use other types of charts if required.

Graph Data Manager

The graph data manager is responsible for storing all fetched data. After each fetch, it compares the new results with the already stored ones, and updates accordingly. All data is stored in observable containers (observable map and observable list from JavaFX library). This allows the UI to subscribe to them and automatically update graphs whenever the data is changed.

Smear Aggregates and Statfi Aggregates

Both Aggregates classes present inside the Utilities package will generate the minimum, maximum and average values for the series data generated for the smear selection, Statfi selection and displays it at the terminal level.

Design Decisions

The goal of the design is to make adding more queries as easy as possible. Everything that is unique to one specific query is isolated in the Query object, while common objects can operate with these without knowing any specifics of their implementation. To add a new query, one only need to define a new class that implements the Query interface, and possibly one that implements JsonToResultsConverter. The new query can use existing ControlComponents and ResultViews. If new types of controls are needed or a new type of ResultView is desired, these can easily be added. The fxml and main controller don't need any modification at all to handle any number of different queries.

The project also implements the model-view-controller design pattern. The fxml and the components are the view, the MainUIController handles interaction with the user, and the graph data manager acts as the model storing all the data to be displayed. The internal representation of the data is separated from how it is displayed to the user. This way it is relatively easy to change the UI without changing how the data is stored internally (which we did throughout the development of the project).

Self-Evaluation

The final product fulfils the requirements stated at the beginning of the project – a standalone application for monitoring greenhouse gas emissions. We managed to implement most of the required features – monitoring real-time data from SMAER, checking historical values from StatFi, comparing current data with historical data, and loading and saving preferences.

There were also some changes to the initial plans and designs. During the implementation of the product, we decided to change the style of the UI. The first version of the UI turned out to be inefficient for our use case. The new version of the UI also allows for easily adding new sources which wasn’t that easy in the initial design.

The final class structure and interdependency looks different to the one we have designed in the planning phase of the project. This is mainly because some ideas and solutions were only thought of during the implementation phase. As we don’t have that much experience in software design yet, it was hard to come up with all the needed classes, interfaces and dependencies at the very beginning of the project.