The correlation between economy and weather

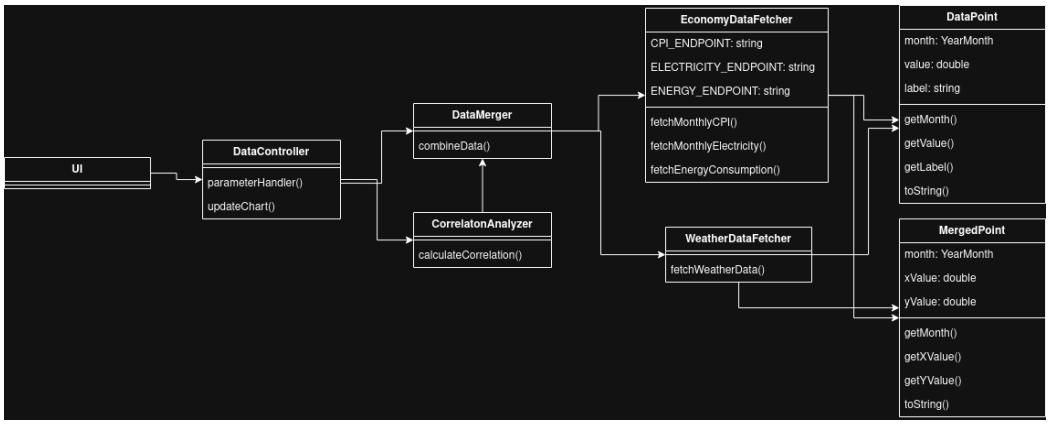
1. High-level Description

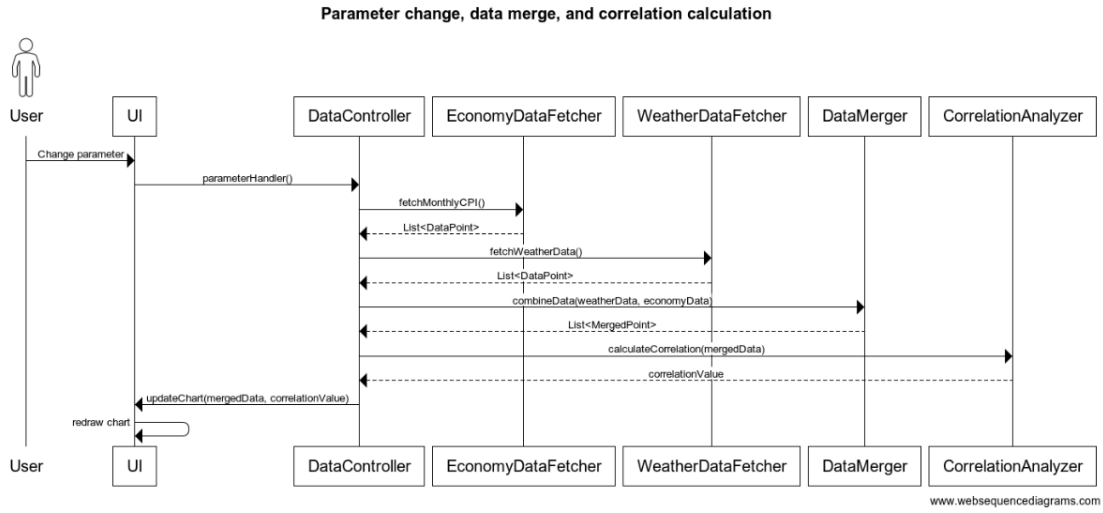
The purpose of the application is to study and visualize correlations between weather conditions and economic variables. The application consists of the following main components:

- EconomyDataFetcher – fetches economic data (CPI, electricity, and energy consumption) from Statistics Finland PXWeb API.
- WeatherDataFetcher – fetches monthly weather data (temperature, rainfall) from the Finnish Meteorological Institute WFS API.
- DataMerger – merges datasets from different sources into a unified timeline.
- CorrelationAnalyzer – calculates correlation coefficients for merged datasets.
- DataController – acts as the controller part of the MVC architectural pattern (handles user input)
- DataModel – acts as the model part of the MVC architectural pattern (represents the data, state, and business logic of the application)
- DataView – acts as the view part of the MVC architectural pattern (presents the model's data to the user)

All data flows through DataPoint and MergedPoint objects, providing a unified way to handle both economic and weather data.

Below is a class diagram showcasing the connections between the classes and a sequence diagram better showcasing the workflow of the application. The logic of the application, meaning everything after DataController, will be added under a separate DataModel class to implement the MVC structure.





We implemented EconomyDataFetcher and WeatherDataFetcher as their own separate classes, since we weren't familiar with Java API calls and the calls differed quite a lot. Now that we know their functionality, it would've been possible to make the software more modular by making a parent class DataFetcher with functions that fetch data by URL and parameters, and a second function that fetches data by HTTP POST commands.

The parent class would have an abstract function for data handling, which would then have to be implemented in the derived classes accordingly. This would allow for any API fetch mentioned above to be quite easily added into our software with simple modifications to fetch parameters and query, as well as handling the fetched data. Since our limited time for the assignment, we will probably have to stick with the current implementation.

2. Components and Responsibilities

2.1 EconomyDataFetcher

Role: Fetches economic data from Statistics Finland PXWeb API.

Responsibilities:

- Builds JSON queries for PXWeb API.
- Executes HTTP POST requests.
- Parses JSON-stat2 responses into DataPoint objects.
- Provides monthly data points labeled by variable (e.g., "cpi").

Helper methods:

- buildMonthCodes() – Generates month codes such as 2024M01.
- buildQuery() – Constructs a valid PX-Web query payload.

- postJson() – Sends a POST request and returns the raw response.
- parseJsonStatMonthly() – Parses the JSON-stat2 response and produces a list of DataPoint objects.

Main functionality:

- fetchMonthlyCPI() – Retrieves monthly CPI values for a full year by:
    - generating month codes
    - building the query
    - sending the request
    - parsing the result into DataPoint objects

Design Rationale:

- HttpClient and Gson were chosen for performance and modern Java support.
- Multiple endpoints (CPI, energy, electricity) can be added easily.
- Immutable objects ensure reliable and thread-safe data handling.

2.2 WeatherDataFetcher

Role: Fetches weather data from FMI WFS API.

Responsibilities:

- Constructs WFS URL based on location, parameter, and time range.
- Parses XML using DOM parser.
- Produces a list of DataPoint objects, including month, value, and label (e.g., "rrmon", "tmon").
- Handles missing values safely (NaN).

Structure:

- Static URLs and namespaces used for parsing the FMI WFS response (opengisurl, xmlurl).
- A DateTimeFormatter to convert timestamps into YearMonth.
- A single public static method (fetchWeatherData) that encapsulates the complete fetch–parse–convert workflow.

Design Rationale:

- DOM parser fits well for hierarchical XML with namespaces.
- YearMonth format ensures month-level data is correctly ordered.
- Including labels allows combining multiple variables in one dataset.

2.3 DataPoint

Role: Generic data point with a timestamp, value, and variable label.

Responsibilities:

- Serves as a common data structure for fetchers and analytics components.
- Immutable design (final fields) ensures safe and consistent data handling.

Fields:

Field	Type	Description
month	YearMonth	Observation month
value	double	Numeric value (NaN allowed for missing data)
label	String	Variable name

Functions:

- getMonth() / getValue() / getLabel() – Simple accessors for the stored data
- toString() – Returns a readable representation

We decided to implement DataPoint class because we didn't have a proper way to store the fetched data.

2.4 MergedPoint

Role: Combined data point for two variables, used for correlation analysis.

Responsibilities:

- Stores month, X value, and Y value (e.g., weather vs. economic variable).
- Immutable design.
- Used as input for CorrelationAnalyzer.

Functions:

- getMonth() / getXValue() / getYValue() – Simple accessors for the stored data
- toString() – Returns a readable representation

We decided to implement MergedPoints class so we could handle the merged data more easily.

2.5 DataMerger

Role: Merges DataPoint lists from different sources into a list of MergedPoint objects.

Responsibilities:

- Iterates through both DataPoint lists.
- Includes only months where both X and Y values exist.
- Sorts merged list chronologically.

2.6 CorrelationAnalyzer

Role: Calculates correlation coefficients from a MergedPoint list.

Responsibilities:

- Computes correlation between two variables.
- Produces a value in [-1, 1] for visualization in the UI.

2.7 DataController

Role: Communicate user inputs for model

Responsibilities:

- Handles user input.

We decided to implement DataController because we wanted to have an MVC architecture.

2.8 DataModel

Role: All the logic of the software will be under model, meaning fetchers and datapoints etc.

Responsibilities:

- Represents the data, state, and business logic of the application.

We decided to implement DataModel because we wanted to have an MVC architecture.

2.9 DataView

Role: Updates the user interface, and shows all the variable components. Draws the graphs and provides a clear user interface.

Responsibilities:

- Presents the model's data to the user.

We decided to implement DataView because we wanted to have an MVC architecture.

3. **Interfaces and Data Flow**

- DataController handles user input and tells the DataModel what it needs to do
- DataModel calls fetchers with the attributes that DataController has given to it. DataModel also passes the data to DataView.
- Fetchers return List<DataPoint> to the DataMerger.
- DataMerger produces List<MergedPoint>.
- CorrelationAnalyzer calculates correlation from the MergedPoint list.
- DataView receives the final results and visualizes them in graphs.

4. **Design Decisions**

Modularity: Each component has a clear responsibility.

Unified data structure: DataPoint and MergedPoint simplify merging and analysis.

Model structure: MVC model was decided to be used for a clearer structure, better maintainability and to make working on different parts of the project at the same time easier.

Extensibility: New variables can be added without changing existing components.

Testability: Fetchers and analyzers can be tested independently with mock data.

5. **Self-Evaluation**

Design support for implementation

The overall design has remained clear, and our data fetchers provide uniform, consistent data to downstream components, which has supported implementation effectively. The planned structure has mostly aligned with how the system was built, although some parts required adjustments during development.

Correspondence to original design

We have largely been able to follow our initial structural design. However, the early prototype did not reflect an MVC architecture, which limited how accurately the prototype represented the final structure. This gap also became visible in our diagrams, which still lack all components of an MVC model. We identified these missing architectural elements only shortly before the midterm deadline.

Implementation based on the original plan

Most features implemented so far follow the intended logic and data flow. Nevertheless, we introduced additional classes for data handling (e.g., DataPoint and MergedPoint) to support merging and processing, which were not explicitly defined in the prototype but became necessary once we began implementing actual functionality. We also renamed the UI layer to DataView to better align with MVC terminology.

Design quality

The current design includes immutable data objects and well-isolated methods, which enhances maintainability, reduces side effects, and supports robustness. These choices should make future growth and modification of the system easier.

Anticipated changes for the remaining implementation

For the final version, we need to formalize a complete MVC architecture that integrates the components already implemented. Some additional structural improvements are also expected.