Final Design Document

Ryhymä

High level description

The application is a JavaFX-based data visualization tool that allows the user to view weather and economic data for a chosen location or region over time. The user can select which type of data to display via the GUI. The application uses a modular architecture based on Model, Controller, and View, following an MVC-like pattern.

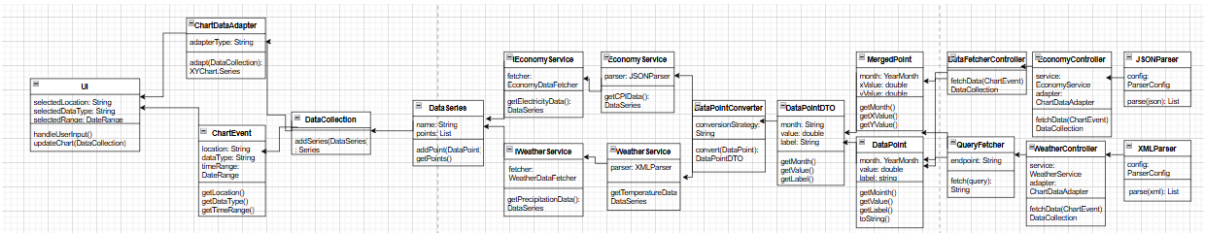
The structure of the application is organized into several packages. The controller package contains logic responsible for managing data retrieval. DataFetcherController acts as a shared abstraction for all controllers that fetch data. WeatherController and EconomyController implement this abstraction by retrieving datasets and returning them as DataCollection, which organizes multiple DataSeries. A helper component called QueryFetcher supports the controllers by handling raw API and service queries.

The data/model layers define how the application represents information internally. A DataSeries stores a named sequence of data points (xValues as YearMonth and yValues as Double) suitable for charting. DataCollection aggregates multiple DataSeries and provides chart metadata (title, x-axis label, y-axis label). The main pipeline now directly uses list of DataPoints, DataSeries and DataCollection for transferring data.

User interactions are handled through event classes like ChartEvent, which describes user-driven changes to what the chart should display. The service layer contains logic for data retrieval. WeatherService fetches and parses weather data from FMI WFS XML, while EconomyService retrieves JSON-based economic data. Utility classes (XmlParser, JsonParser and ChartDataAdapter) convert raw responses into domain objects and optionally adapt them for JavaFX charts. The View is implemented with JavaFX and dynamically displays charts based on DataCollection objects. Main initializes and launches the application.

Diagrammatically, the architecture can be expressed via a class or component diagram showing controllers requesting data from services, services using parsers to load external data, DataCollection/DataSeries holding structured results, and the View

consuming these objects to update charts. Below is a class diagram:

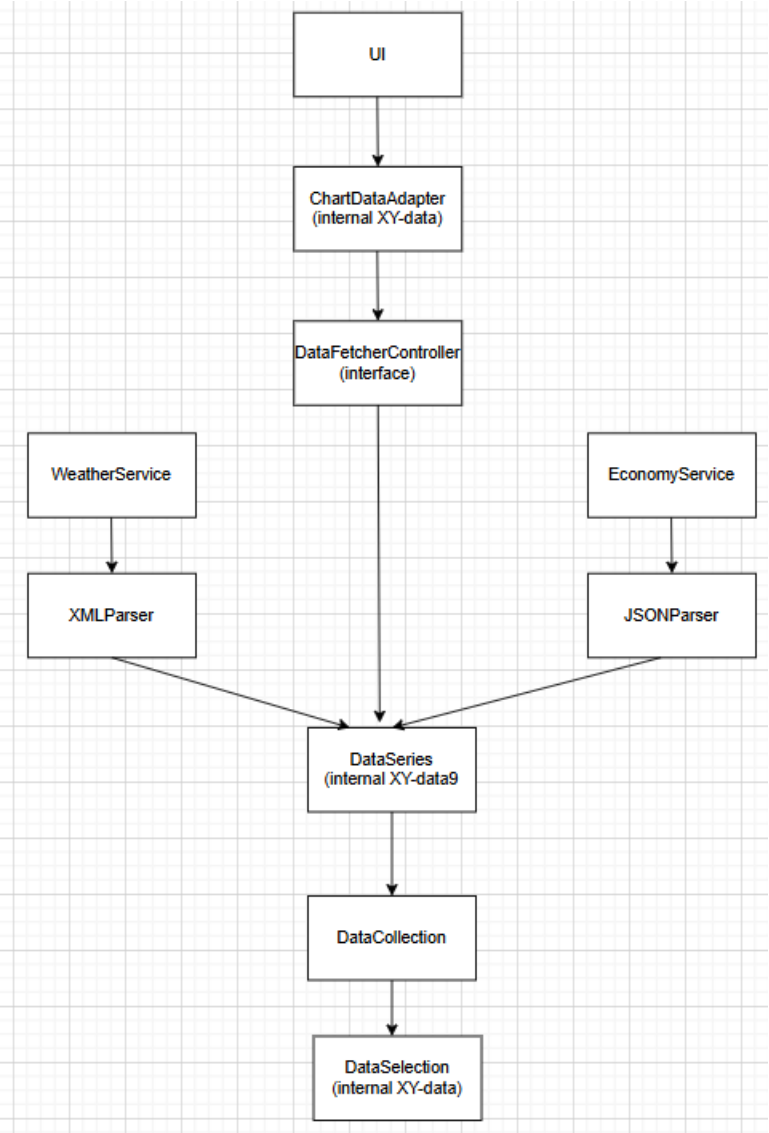


Boundaries and internal interfaces

Information flows through the system in a well-defined sequence:

1. When the user interacts with the GUI (selecting a city, data type, or time range), a ChartEvent is triggered.
2. The View sends this event to the appropriate controller (WeatherController or EconomyController) through the DataFetcherController interface, ensuring decoupling from concrete implementations.
3. The controller delegates data retrieval to the corresponding service (WeatherService or EconomyService).
4. Services parse raw API responses (XmlParser or JsonParser) and produce domain objects (DataPoint).
5. These domain objects are organized into DataSeries, which are then aggregated into a DataCollection.
6. The controller returns the DataCollection to the View, which converts the series to JavaFX chart series (XYChart.Series) internally before updating the GUI.

This sequence ensures separation of concerns: the View only depends on abstract data representations (DataCollection), not on service or parser implementations. Below is a diagram of the given topic:



Component responsibilities

The model layer defines how data is represented within the application. The DataPoint class represents a single measurement, while the DataSeries class stores an ordered sequence of data points, using YearMonth values for the x-axis and Double values for the y-axis. The DataCollection class serves as a container for multiple DataSeries objects and also stores metadata needed for chart visualization, such as the chart title and axis labels.

The service layer is responsible for retrieving and parsing external data. The WeatherService fetches weather data in XML format from the FMI API, and the EconomyService retrieves economic data in JSON format from remote datasets.

Parsers such as XmlParser and JsonParser convert the raw responses into domain objects that can be further processed.

Controllers act as intermediaries between user input and the service layer. Both the WeatherController and EconomyController implement the DataFetcherController interface. This ensures that the View can request data in a uniform way, regardless of the source or type of data. The QueryFetcher class supports controllers by handling generic HTTP requests.

The View is implemented using JavaFX and manages all user-facing functionality. It allows the user to select the data source, variable, location, and time period. When new data arrives, the View converts the DataSeries objects contained in a DataCollection into XYChart.Series objects and updates the chart display accordingly.

Utility components support the entire data processing pipeline. The ChartDataAdapter builds chart-ready objects that can be directly consumed by the JavaFX View.

Design decisions

The application uses JavaFX for its graphical interface and charting features, as it provides built-in controls and visualization capabilities suited for dynamic data. Maven is used for project configuration and dependency management. Weather data is fetched from the FMI WFS API, which delivers XML responses, while the economic data uses a JSON-based remote dataset.

Architecturally, the system largely follows an MVC-style separation. A ViewController was planned but ultimately not implemented due to time constraints, as development efforts had to be redirected toward resolving other system issues. As a result, the View does not follow MVC principles. The Controllers mediate between user actions and the service layer, while the View was supposed to be limited to presentation.

Several software design principles guided the implementation. The Single Responsibility Principle ensures that each class handles only one logical concern. The system is structured to be open to extension but closed to modification, allowing new data types or chart variants to be added without rewriting existing components. Dependency Inversion is reflected in the use of interfaces such as IWeatherService, IEconomyService, and DataFetcherController, allowing higher-level components to rely on abstractions rather than concrete classes. We chose to convert the fetched data into DataPoints first, and only then into DataSeries and DataCollection, because DataPoints would have been easier to work with had we implemented the correlation-coefficient calculator.

These choices were made to improve modularity and extensibility. JavaFX was selected due to its integration with charting libraries.

Self-evaluation

The initial design supported the basic implementation of data fetching. However, the earlier structure did not follow the MVC paradigm or SOLID principles effectively. As a result, refactoring toward these principles had to be carried out during implementation rather than beforehand, which created additional challenges and slowed down progress. Despite this, the revised architecture ultimately improved maintainability and clarity.

The introduction of DataSeries and DataCollection simplifies chart generation and makes the architecture cleaner and more maintainable. Earlier challenges with type compatibility have been resolved with these abstractions.

The initial design included a correlation analyzer but due to the tight time schedule we decided to put all of our focus on implementing the graph. The UI also does not fully match the prototype, as we allocated the remaining time to implementing the core logic of the application rather than refining the visual design.

The use of AI

During the implementation, we encountered some issues with certain parts of the code, particularly in parsing and adapting the data for chart visualization. To support debugging and explore potential solutions, we consulted AI-based tools to get guidance on best practices and code examples. All final implementations and design decisions were reviewed and adapted manually by the team.