Tampereen yliopisto

FINAL DESIGN DOCUMENTATION

COMP.SE.110

Javengers group assignment

Jussila, Henri
Kari, Julius
Karjalainen, Esa
Lappalainen, Lotta

TABLE OF CONTENTS

1. HIGH LEVEL DESCRIPTION

1.1. Description

The application is a general-purpose electricity analytics application. It combines Finnish electricity production data (solar, water, wind) and weather data (temperature, wind, cloudiness), displaying correlations between different weather metrics and specific types of electricity production methods across different timeframes.

1.2. Structure

The application is made using a model-view-controller design pattern.

- Model
    - FMIFetcher: Fetches weather data from the FMI API (Finnish Meteoroligal Institute).
    - FMIParser: Parses the data from FMI API.
    - FingridFetcher: Fetches electricity production data from the Fingrid API.
    - FingridParser: Parses the data from Fingrid API.
    - Enumerated classes for datatypes: EnergyType, TimeRange and WeatherType to define and easily specify information for making external API calls.
- View
    - MainApp: Initializes and starts the applications.
    - Viewer: Provides the UI for interacting with the application.
    - DualAxisLineChart: Custom chart that supports dual Y-axes for displaying multiple datasets simultaneously.
    - Utility: Provides utility methods for generating labels and converting data formats.

- Controller
    - LogicController: Core processing unit for managing applications features, such as saved views and data retrieval.
    - ChartController: Manages the updates to the DualAxisLineChart, ensuring that the displayed data reflects user selections.

1.3. Libraries and third-party components

- FMI API: Provides the application with weather data, which includes the temperature, wind, and cloudiness. Provided data is in XML format. Manual can be found here. Used by FMIFetcher.
- Fingrid API: Provides the application with energy production data with different methods, which include water, solar, and wind. Provided data is in JSON format. Manual can be found here. Used by FingridFetcher.
- Gson: Provides easy to use methods for converting JSON into java objects. Chosen because of its ease of use. Used by FingridFetcher.
- JavaFX: Provides the tools for creating the UI and managing user inputs. Used by MainApp.
- OkHttp: Provides an HTTP client. Chosen because of its efficiency and ease of use. Used by FMIFetcher.
- SLF4J: Provides methods for logging, used by FMIFetcher and FMIParser.
- JUnit 5: Provides a framework for unit testing. Used for its readability. Used by all the test classes.

1.4. Class Diagram

Figure 1 shows the relationships between classes and the APIs

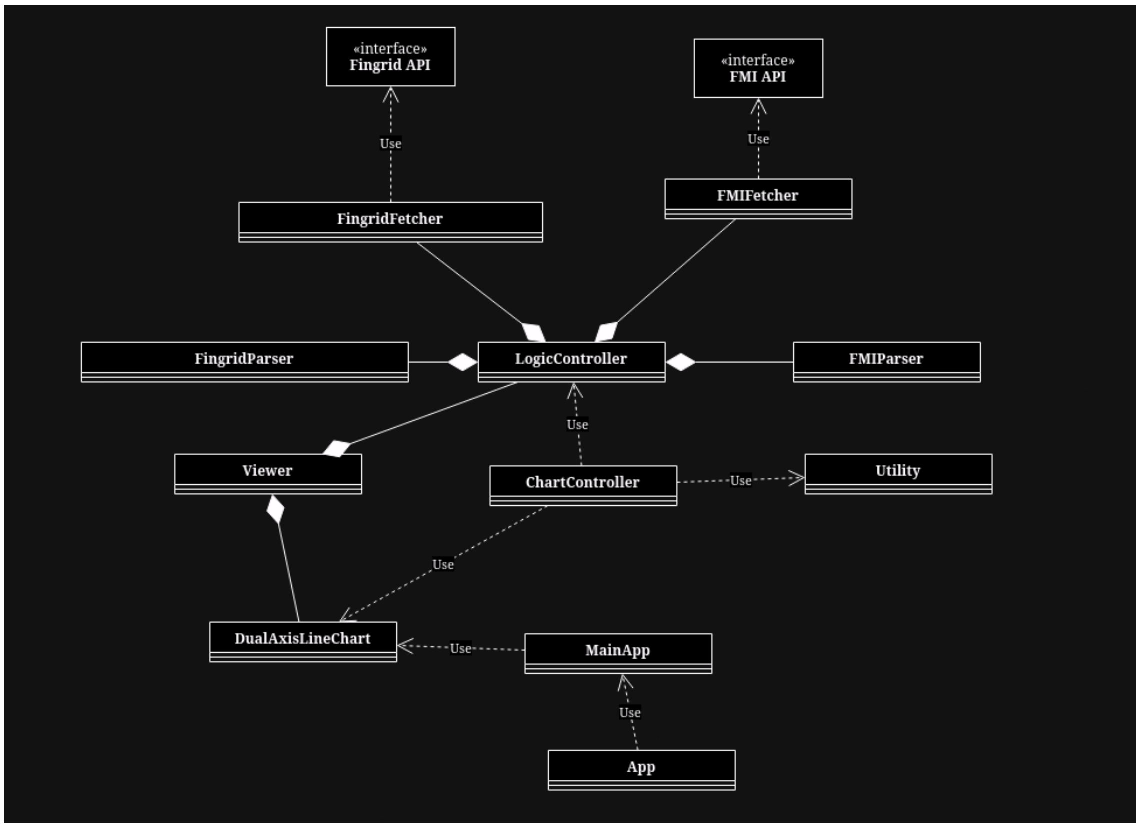


Figure 1. *A simple class diagram of the application*

2. BOUNDARIES AND INTERFACES (INTERNAL)

2.1. General Flow of Information

The application relies on a model-view-controller (MVC) structure, where each component has distinct responsibilities. The **Model** supplies data to the **Controller** and, ultimately, to the **View**. The **Controller** acts as the central processor and coordinator, managing the flow of data between components, while the **View** presents this data to the user. Each component interacts through well-defined interfaces, with specific classes supplying or demanding data and functions based on their responsibilities. Figure 1 displays a UML sequence diagram for the general operation of displaying weather and energy data in the UI.

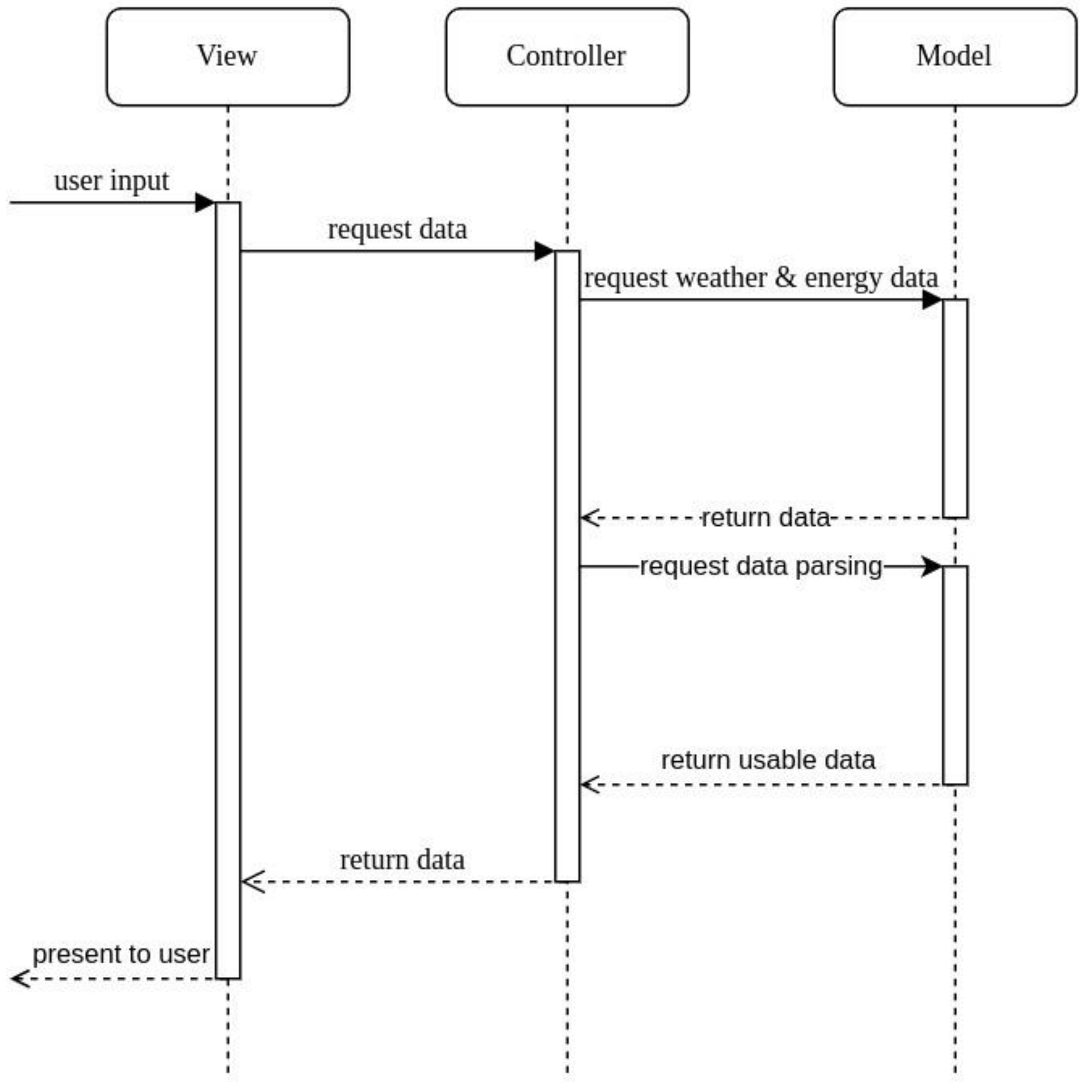


Figure 2. *Sequence diagram of the processes for displaying data to the user.*

2.2. Component Interfaces

1. **FMIFetcher and FingridFetcher (Model to Controller Interface)**

Suppliers: FMIFetcher and FingridFetcher (within Model).
Demanded By: LogicController (within Controller).
Description: FMIFetcher fetches and processes weather data from the FMI API, while FingridFetcher retrieves electricity production data from the Fingrid API. Both fetchers return their data in Java object format, ensuring compatibility with the application's internal data structures.

Data Flow: The fetchers pass data to the LogicController, which manages the data flow to and from the View.

2. **FMIParser and FingridParser (Model to Controller Interface)**

Suppliers: FMIParser and FingridParser (within Model).
Demanded by: LogicController (within Controller).
Description: FMIParser provides weather data parsing functionality, while FingridParser does the same to the electricity data. Parsed data is returned as Java objects or primitive types, to abstract any XML or JSON data from the users of this interface.
Data Flow: The parsers pass the parsed data to LogicController, which can subject the data to more processing or supply it to the View.

3. **LogicController and ChartController (Controller Interface)**

Suppliers: LogicController (main business logic and processing).
Demanded By: ChartController (within Controller).
Description: LogicController manages data retrieval and business logic for saved views, acting as the primary coordinator. It provides ChartController with processed data and handles the user's selected configurations, thus streamlining data visualization updates.
Data Flow: ChartController calls methods from LogicController to fetch or apply settings, such as weather and energy type selections, and time range adjustments, which are then rendered in the View.

4. **LogicController and MainApp/Viewer (Controller to View Interface)**

Suppliers: LogicController (data retrieval and settings management).
Demanded By: MainApp and Viewer (within View).
Description: LogicController processes user requests from the View and passes data to MainApp and Viewer for display. It also facilitates updating saved view configurations and time range selections.
Data Flow: Data flows from LogicController to the View components, enabling real-time updates based on user interactions and saved configurations.

5. **Viewer and DualAxisLineChart (View Interface)**

Suppliers: Viewer (UI controls and layout).
Demanded By: DualAxisLineChart (within View).
Description: Viewer sets up the UI and creates instances of visual elements, including DualAxisLineChart, which displays the data with dual Y-axes.
Data Flow: Viewer supplies layout and control configurations, while DualAxisLineChart demands this data to update visualizations based on user selections.

6. **Utility (Helper Interface)**

Supplier: Utility (helper functions).
Demanded By: Various components, including LogicController, Viewer, and ChartController.
Description: Utility provides helper functions, such as label generation and data conversions, which support data visualization and formatting.
Data Flow: Utility methods are called by components as needed to streamline data processing and enhance visual clarity.

2.3. Responsibilities of Data Supply and Demand

Model (FMIFetcher, FingridFetcher and respective Parsers): Supplies data to LogicController based on API responses. It is the primary data source, transforming raw API data into internal Java objects.
Controller (LogicController and ChartController): Demands data from the Model and supplies processed information to the View. It acts as the primary intermediary for managing and processing application data, ensuring consistency across user interactions.
View (MainApp, Viewer, and DualAxisLineChart): Demands data from the Controller to render it for the user. Viewer and DualAxisLineChart rely on data provided by LogicController and ChartController to update the interface and data visualizations accordingly.
Utility: Supplies helper functions across components, ensuring efficient data handling, formatting, and conversion for a consistent user experience.

These boundaries and interfaces facilitate clear communication between components, allowing for smooth data flow and maintaining separation of concerns across the application's structure.

2.4. Changes between mid-project and final state

No major changes to the interfaces were made after the mid-project checkup. The only required change was separating fetching monthly data from the FMI API into its own method in **FMIFetcher**. This was done due to restrictions in the API not allowing timespans of over 168 hours to be fetched in a single request regardless of the amount of datapoints returned. Subsequently, the monthly weather data required a parsing method of its own in **FMIParser** due to technical limitations and for maintaining code readability.

3. COMPONENTS AND RESPONSIBILITIES:

3.1. General purpose and responsibilities of components

Model

The model component of the application is primarily responsible for data management and retrieval. It includes the FMIFetcher, which communicates with the Finnish Meteorological Institute's (FMI) API to fetch weather data, the FingridFetcher, which communicates with Fingrid's API to fetch energy production data, as well as the FMIParser and FingridParser, which parse the data acquired from the respective APIs. Additionally, the EnergyType, TimeRange, and WeatherType enums define and manage energy sources, time intervals, and weather parameters for fetching the correct data.

Controller

This component handles fetching data for weather and energy production and coordinates the presentation of data in the user interface. The LogicController serves as the intermediary between the model and view components. It processes data, manages business logic, and maintains the application state, including saved views. The ChartController manages the display of data on a dual-axis line chart and makes updates to the Viewer based on user interactions.

View

The view component comprises the user interface and visualization elements of the application. It includes the MainApp class for application initialization, the Viewer for layout and controls, the DualAxisLineChart for supporting dual Y-axes in data visualization, and the Utility class, which provides helper methods for data processing and formatting. The view ensures the proper display of data and allows users to interact with the application through controls and charts.

3.2. Detailed structure and functions of components

Model: EnergyType (Data Type Enum)

- **Author:** Julius
- **Purpose:** Represents different types of energy sources and their corresponding dataset IDs in Fingrid's API.
- **Functions:**
  - getValue(): Returns the dataset ID associated with the energy type.
- **Responsibilities:**
  - Provide the correct dataset ID for API queries.

Model: TimeRange (Data Type Enum)

- **Author:** Esa
- **Purpose:** Represents different time spans for data queries.
- **Functions:**
  - getValue(): Returns the number of days the time range spans.
  - getTimestep(): Returns the interval between data points (in minutes) based on the time range.
- **Responsibilities:**
  - Define and manage time intervals for data fetching.
  - Return appropriate timestep values for API queries.

Model: WeatherType (Data Type Enum)

- **Author:** Esa
- **Purpose:** Represents different weather parameters and their corresponding identifiers in FMI's API.
- **Functions:**
  - getValue(): Returns the parameter identifier for API queries.
  - toString(): Returns a human-readable name for the weather parameter.
- **Responsibilities:**
  - Map weather parameters to their API names.
  - Provide descriptive labels for UI or logs.

Model: FMIFetcher (Data Fetching Component)

- **Author:** Esa
- **Purpose:** Handles communication with the Finnish Meteorological Institute's (FMI) API to fetch weather data.
- **Functions:**
  - FMIFetcher(): Initializes an OkHttpClient for making HTTP requests.
  - fetchWeatherData(weatherType, timeRange): Constructs the API request URL and calls executeRequest().
  - fetchMonthlyWeatherData(weatherType, timeRange): Generates and executes multiple requests for data spanning over a month due to API constraints.
  - executeRequest(requestString): Executes the HTTP request and returns the API response as a string, logging success and handling errors.
- **Responsibilities:**
  - Build and execute HTTP requests to fetch weather data.
  - Handle API limitations for large timespans by splitting requests into smaller intervals.
  - Log successes and errors during the data-fetching process.
  - Return weather data as an XML string.

Model: FMIParser (Data Processing Component)

- **Author:** Esa
- **Purpose:** Parses and processes weather data fetched from the FMI API to make it usable by the application.
- **Functions:**
    - parseWeatherData(data, weatherType): Parses weather data from an XML string and converts it to an array of doubles.
    - parseMonthlyWeatherData(data, weatherType): Parses multiple weekly XML data strings and aggregates the results into a single array of doubles.
    - convertOktasToPercentages(data): Converts cloudiness values from oktas to percentages.
- **Responsibilities:**
    - Parse XML-formatted weather data into numerical arrays.
    - Handle different weather data formats and adjust for specific units (e.g., cloudiness in oktas).
    - Return parsed weather data in a format usable by the application.

Model: FingridFetcher (Data Fetching Component)

- **Author:** Julius
- **Purpose:** Handles communication with Fingrid's API to fetch energy production data based on user-specified parameters.
- **Functions:**
    - fetchEnergyData(timeRange, energyType): Constructs the API request URL, executes the HTTP call, and returns the response body as a JSON string or null if an error occurs.
- **Responsibilities:**
    - Build and execute HTTP requests for energy production data.
    - Return energy production data as a JSON-formatted string.

Model: FingridParser (Data Processing Component)

- **Author:** Julius
- **Purpose:** Parses and processes energy production data from Fingrid's API, transforming it into a usable format for the application.
- **Functions:**
    - parseEnergyData(data, energyType, timeRange): Parses the provided JSON data and returns it as an array of doubles, adjusted for the specified energy type and time range.
    - parseJSON(JSON): Extracts the "data" array from a JSON string and returns it as a JsonArray.
- **Responsibilities:**
    - Parse raw JSON data from the Fingrid API.
    - Convert and format energy data into a form usable by the application, including handling different observation intervals and energy types.

Controller: LogicController (Business Logic Component)

- **Author:** All team members
- **Purpose:** Core processing unit for managing application features, including saved views, data retrieval, and data processing.
- **Functions:**
  - saveView(name, time, weatherType, energyType): Saves user-defined views, consisting of weather type, energy type, and time range.
  - getSavedViewNames(): Returns a set of names for all saved views for easy access.
  - getViewSettings(name): Retrieves the settings (time, weather type, energy type) associated with a specific saved view.
  - fetchWeatherData(weatherType, timeRange): Fetches weather data based on type and time range, parsing monthly or general data accordingly.
  - fetchEnergyData(energyType, timeRange): Fetches energy data based on type and time range, returning parsed results for analysis.
- **Responsibilities:**
  - Coordinate data fetching from the model components and provide processed data to the view.
  - Maintain and manage application state through saved views.
  - Process user interactions and handle data updates efficiently.

Controller: ChartController (Data Visualization Component)

- **Author:** Henri & Lotta
- **Functions**:
  - updateChart(): Refreshes the dual-axis line chart with new data based on user selections from the weather and energy combo boxes. It dynamically adjusts the chart according to the selected time range (Current, Week, Month) and populates the chart with appropriate labels and data points.
  - setYaxisRanges(): Sets the range for a given Y-axis based on the provided data. The min and max values will be calculated dynamically from the data, with padding for better visualization.
  - updateSavedViewsComboBox(): Populates the ComboBox with the names of all available saved views, enabling users to select previously defined configurations.
  - applySavedView(): Applies the settings of a selected saved view to the current user interface components, updating the weather type, energy type, and time range based on the saved configuration.
  - addSaveButtonFunctionality(): Implements functionality for the Save button, allowing users to save their current selections as a new view. It prompts the user to enter a name for the view and updates the saved views ComboBox accordingly.
  - getAverageEnergy(): Returns the formatted string representing the average energy value.

    - getAverageWeather(): Returns the formatted string representing the average weather value.
    - getPeakEnergy(): Returns the formatted string representing the peak energy value.
    - getPeakWeather(): Returns the formatted string representing the peak weather value.
    - createLoadingScreen(): Creates and returns a loading screen (VBox) with a loading message and progress bar, which is displayed during data fetching.
    - getTimeLabelsForRange(): Determines the appropriate time labels based on the selected time range (Current, Week, Month).
    - setUIControlsState(): Enables or disables the user interface controls (ComboBoxes, ToggleGroup, etc.) based on the provided flag, to manage the state during data loading.
- **Responsibilities**:
    - Facilitate the visualization of weather and energy data in a user-friendly format.
    - Ensure that the chart accurately represents the latest available data.
    - Maintain synchronization between user selections and the displayed data.

View: MainApp (Application Initialization Component)

- **Author:** Henri & Lotta
- **Functions**:
    - start(Stage primaryStage): Initializes the primary stage and scene, incorporating the Viewer component to display the main layout. Configures the window size and title before rendering the interface.
    - main(String[] args): Serves as the entry point of the application, launching the JavaFX runtime environment and initiating the start method.
- **Responsibilities**:
    - Create and display the main user interface.
    - Ensure proper initialization and coordination between the model, controller, and view components.
    - Establish the foundational structure for rendering and interacting with the application interface.

View: Viewer (User Interface Component)

- **Author:** Henri & Lotta
- **Purpose**: Provides the main user interface for interacting with the application.
- **Functions**:
    - Viewer(): Initializes UI components, sets up the layout, and configures the dual-axis line chart for displaying weather and energy data.

    - getMainLayout(): Returns the main layout of the Viewer as a VBox, including padding around the chart for improved aesthetics and usability.
    - createTopControlsBox(): Configures the top controls of the user interface, including labels, combo boxes for selecting weather and energy metrics, a saved views ComboBox, and buttons for saving views. Organizes all interactive components in a visually coherent manner.
    - createSaveButton(): Creates a Save button along with its layout and assigns functionality to save the current view settings, ensuring that user-defined views can be stored and retrieved later.
    - createTimeSelectionBox(): Constructs a set of RadioButtons for selecting the time range (Current, Week, Month), allowing users to specify the temporal context of the data displayed in the chart.
    - createAveragesBox(): Generates a visual display of average and peak values for energy consumption and weather metrics in a square-shaped layout.
    - createSquareLabel(): Styles and formats individual labels for averages and peaks, displaying title and value in a visually appealing two-line format.
    - updateAveragesBox(): Updates the averages box with the latest data based on user selections.
    - updateChart(): Refreshes the dual-axis line chart with the latest data based on the user's selections from the weather combo box, energy combo box, and time range toggle.
    - getWeatherUnit(): Determines the appropriate unit (e.g., °C, m/s, %) for the selected weather metric.
    - showErrorAlert(): Displays an error alert with a specified title and message to inform the user of any issues.
- **Responsibilities**:
    - Render the user interface and handle user inputs.
    - Update the display based on interactions with the controls.
    - Facilitate data visualization through the dual-axis chart.

View: DualAxisLineChart (Custom Chart Component)

- **Author:** Henri & Lotta
- **Purpose**: Custom chart that supports dual Y-axes for displaying multiple datasets simultaneously.
- **Functions**:
    - DualAxisLineChart(): Initializes the chart with specified X and primary Y axes.
    - setAlternativeYAxis(): Associates a secondary Y-axis with a specific data series.
    - layoutChildren(): Adjusts the positioning of the second Y-axis relative to the chart area.
- **Responsibilities**:
    - Provide a visual representation of both weather metrics and energy consumption data.

- Dynamically update based on user selections in the interface.

View: Utility (Helper Functions Component)

- **Author:** Henri & Lotta
- **Purpose**: Provides utility methods for generating labels, converting data formats, and performing basic statistical calculations.
- **Functions**:
  - getLast15HoursLabels(): Generates labels for the last 16 hours in a 24-hour format.
  - getLast15DaysLabels(): Creates labels for the last 15 days, formatted as "MMM d".
  - convertIntToDouble(): Converts an integer array to a double array for chart data processing.
  - calculateAverage(): Calculates the average value from a dataset.
  - calculatePeak(): Finds the peak (maximum) value in a dataset.
- **Responsibilities**:
  - Support other components by providing commonly used functionalities.
  - Streamline data handling and formatting for better usability.

4. DESIGN DECISIONS

4.1. Technologies

Java: The primary programming language for this project, as required by the assignment.

Maven: A build automation tool used to manage the project's dependencies, build lifecycle, and unit tests. We chose Maven because it simplifies the integration of external libraries and facilitates a consistent build process.

JavaFX: Used for the development of the graphical user interface (GUI). We chose JavaFX because it allows the creation of desktop applications with user interfaces and is well-suited for this project's needs for interactive charts and controls.

4.2. External Libraries

JUnit 5: The chosen testing framework for unit testing. We chose JUnit 5 because it provides a powerful and flexible framework for testing Java applications, with features such as lifecycle annotations, assertions, and test discovery.

Log4j: A logging library used for logging messages in various formats. Used because it helps track and debug the application during development and in production environments.

OkHttp: A popular HTTP client for Java. OkHttp is used in this project to make network requests, enabling communication with APIs or other services.

Gson: Another library used for converting Java objects into their JSON representation and vice versa, used in scenarios where we need to interact with external JSON-based data sources.

4.3. Design Patterns

In our project we decided to use the MVC design pattern where the program consists of Model-, View- and Controller components. The decision was made based on the nature of our program; we felt that MVC serves the best regarding what we are trying to accomplish. Below we explain in more detail what the responsibilities of each component are to justify our decision.

4.3.1. Model

The Model-component is responsible for representing the data, the business logic and how data is manipulated. It handles the core functionality and data operations without concerning itself with how the data is displayed or interacted with by the user. In short, our Model is responsible for retrieving the weather and energy data from their respective APIs and altering it to the wanted form.

4.3.2. View

The View-component is responsible for the UI of the program. It creates all the elements, shows the data to the user and sends user requests to the Controller. We use View to let users interact with the program, but it does not handle any data processing.

4.3.3. Controller

The Controller-component is intermediary between the Model and the View. It handles user inputs received from the View, processes them, and sends commands to the Model to update data or retrieve new data, and then updates the View. Our controller receives user inputs of selected time range, type of weather and energy metrics and then sends requests to the Model asking for the corresponding data. After it receives the data, the Controller updates the View accordingly.

We believe that we have accomplished using MVC effectively in our project and that it was the right decision to go with. We ended up using the active MVC model which was our initial plan.

Singleton: Used to ensure that certain objects (such as the main application) are only instantiated once and shared throughout the application.

4.4. Other Design Decisions

We used separate classes for both API fetchers. If they had been more similar or if there were more fetchers to use, we could have used an abstract class to create each fetcher from. In scope of this project we felt like this was not necessary and would have just caused more work.

From SOLID principles we followed Single Responsibility Principle (SRP) in most of our classes since they only have one responsibility. Some classes could have been divided into even smaller classes but because of time constraints we did not do this. We follow the Open Closes Principle (OCP) since we don’t modify the extended classes to add new features. But to be fair we did not create any interfaces or abstract classes from which we would create subclasses. We saw no need for this in this project. Since we did not use any subclasses, we did not violate Liskov Substitution Principle (LSP). Same goes for Interface Segre Principle (ISP) and Dependency Inversion Principle (DIP).

5. SELF-EVALUATION

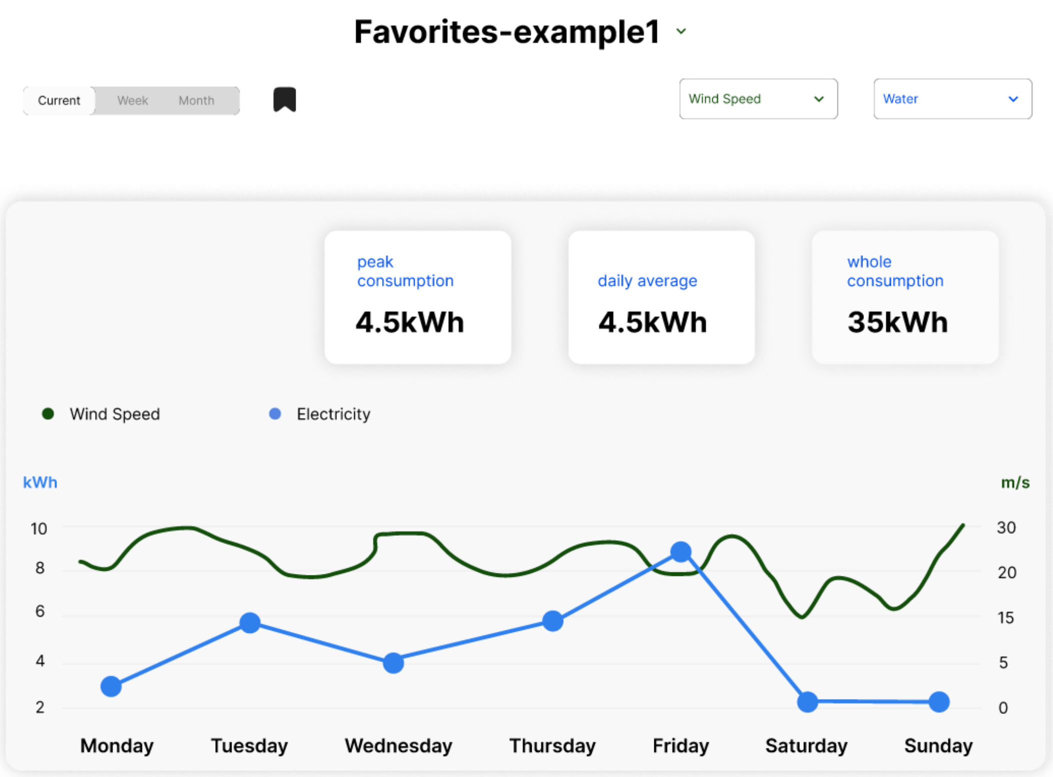
5.1. The UI

Our original UI design and our current implementation can be seen on page XX. The first picture is the Figma design and the second our final product. We have been able to implement all the features that were in the design. The look of the UI is not as clean but since this was not the focus of the project, we did not try to spend too much time on it. We also felt that Java FX has its limitations when it comes to “stylish” UI design. Overall, we have been able to stick with the original UI design well, only deviating from it in some minor details. For the UI we haven’t had to make any notable changes.

5.2. The Design

Our original plan was to follow the MVC pattern, the active model to be specific. We feel like we have been able to accomplish this in our project. Our Viewer is responsible for creating the chart and all the elements of the UI. All the user interactions that modify the Viewer (UI) are sent to the Controller. The Controller then communicates with the Model that is responsible for fetching and handling the data. Finally, the Controller updates the View with the data it received from the Model. We believe that MVC and active model was the most effective and simplest way to implement this project. We did not have to make any changes to our original plan and implementing the project with this design proved to be straightforward, which to us proves that it was the right design pattern to go with.

Original



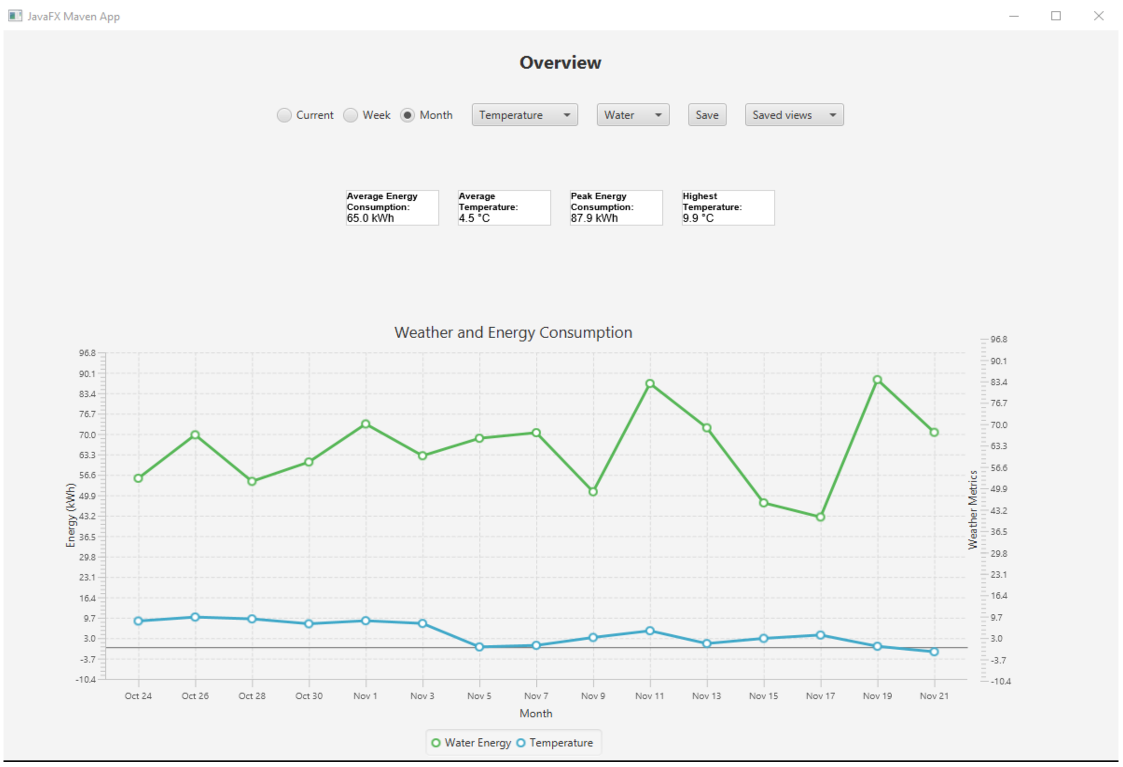
Favorites-example1

peak consumption
4.5kWh

daily average
4.5kWh

whole consumption
35kWh

Final Product



JavaFX Maven App

Overview

( ) Current ( ) Week (●) Month

Temperature | Water | Save | Saved views

Average Energy Consumption:
65.0 kWh

Average Temperature:
4.5 °C

Peak Energy Consumption:
87.9 kWh

Highest Temperature:
9.9 °C

6. THE USE OF AI

The use of AI varied depending on the student. High-level decisions were always made by the students but with certain low-level decisions we used AI to assist. We only used Chat-GPT and Copilot.

6.1. Tools Used and Their Application

During the implementation phases of this project, we utilized ChatGPT and GitHub Copilot as prompt-based AI tools to enhance our development process. These tools were primarily used for generating code, troubleshooting and refactoring.

6.1.1. ChatGPT

We used ChatGPT to generate different ideas for the project. We provided ChatGPT with the whole Group Assignment file and asked it to generate 10 different ideas. Out of those 10 ideas we chose one to investigate further. We checked if there were APIs that would align with the idea’s requirements. Based on the data accessible through these APIs, we made slight adjustments to the idea.

ChatGPT helped us with creating detailed explanations of specific Java concepts and libraries, which speeded up our learning process in unfamiliar areas (e.g., JavaFX). For new features, we asked ChatGPT for advice on how to structure the code. Prompts like, "How can I create a dual-axis chart in JavaFX?" were used. Based on its suggestions, we were able to implement features quickly, learning best practices along the way. When encountering issues with specific functionality, such as unexpected behavior in chart rendering or incorrect calculation logic, we asked ChatGPT for help in debugging. We provided a description of the problem and asked for debugging strategies. For instance, we asked, "Why isn't my JavaFX chart rendering correctly?" and received step-by-step troubleshooting tips. We also used ChatGPT for advice on refactoring code to improve readability and maintainability. Prompts like, "Can you help me refactor this large function into smaller, more manageable functions?" helped us optimize our code.

ChatGPT was also used to review grammar in some sections of the documentation with prompts like “Can you write this better?”.

6.1.2. GitHub Copilot

GitHub Copilot was used to generate code suggestions while writing our application. Copilot's autocomplete functionality helped us with repetitive tasks and ensured that our code followed the best practices. Copilot assisted with fixing smaller bugs by providing suggestions in real-time, such as correcting syntax errors. Its real-time suggestions saved us time with debugging. While refactoring, Copilot helped by suggesting improvements in code structure, such as recommending the use of helper methods.

6.2. Effectiveness of AI Tools in Design and Implementation

6.2.1. Advantages

AI tools helped us speed up feature development, bug fixing, and code refactoring by providing instant solutions and suggestions. This saved time during the coding process. ChatGPT was a valuable educational resource, providing us with explanations of new concepts, libraries or frameworks. Both ChatGPT and Copilot assisted with catching common mistakes and improving code quality.

6.2.2. Disadvantages

Relying too much on AI tools can result in a lack of deeper understanding of certain parts of the code. Learning should not be delegated, so to speak. In some cases, AI-generated suggestions needed to be carefully reviewed and adjusted to align with our specific requirements. While AI tools were very helpful, there were occasions when they provided suggestions that didn't fully align with our project, requiring us to refine or reject some of the recommendations.