Tampereen yliopisto

MID-TERM 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 Meteorological 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.

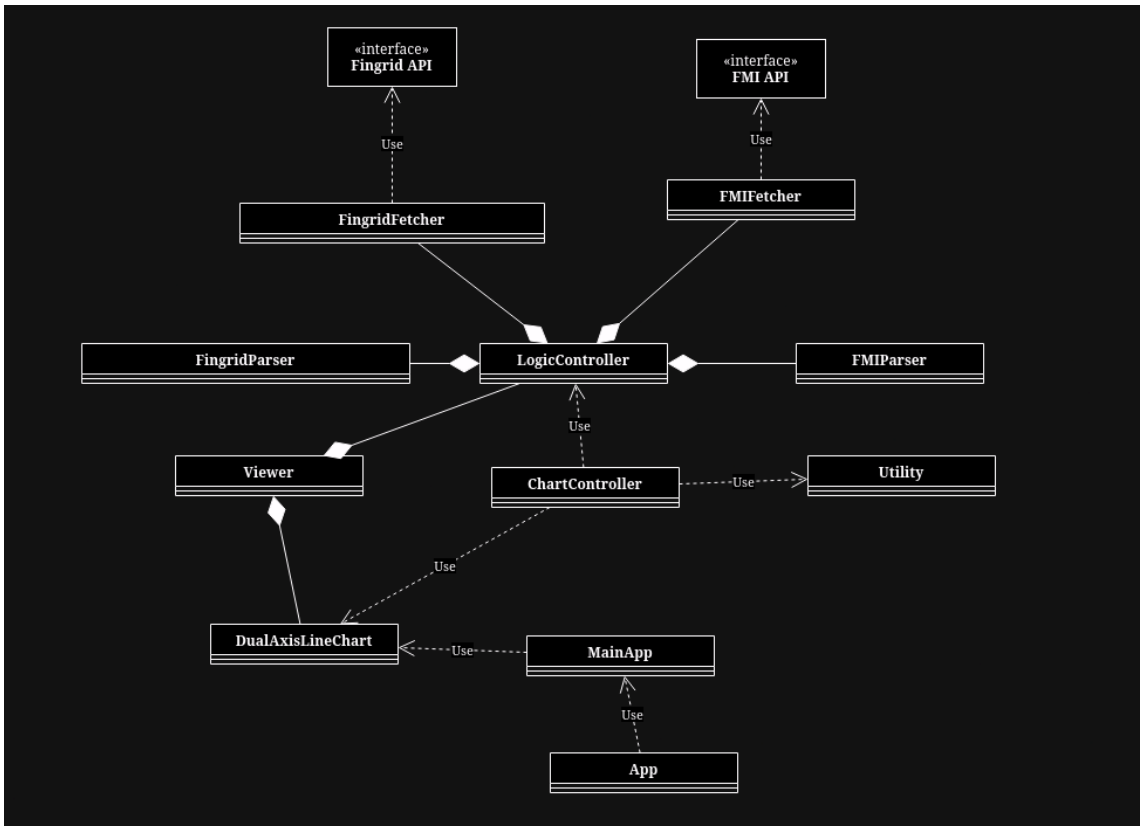
- Controller
    - LogicController: Core processing unit for managing application features, such as saved views and data retrieval.
    - ChartController: Manages the updates to the DualAxisLineChart, ensuring that the displayed data reflects user selections.

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. 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. 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.

1.4 Class Diagram

This class diagram shows the relationships between classes and the APIs



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.

2.2 Component Interfaces

1. **FMIFetcher and FingridFetcher (Model to Controller Interface)**
    a. **Suppliers**: FMIFetcher and FingridFetcher (within Model).
    b. **Demanded By**: LogicController (within Controller).
    c. **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.
    d. **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)**
    a. **Suppliers:** FMIParser and FingridParser (within Model).
    b. **Demanded by:** LogicController (within Controller).
    c. **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.
    d. **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)**
    a. **Suppliers**: LogicController (main business logic and processing).
    b. **Demanded By**: ChartController (within Controller).

   c. **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.
   d. **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)**
   a. **Suppliers**: LogicController (data retrieval and settings management).
   b. **Demanded By**: MainApp and Viewer (within View).
   c. **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.
   d. **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)**
   a. **Suppliers**: Viewer (UI controls and layout).
   b. **Demanded By**: DualAxisLineChart (within View).
   c. **Description**: Viewer sets up the UI and creates instances of visual elements, including DualAxisLineChart, which displays the data with dual Y-axes.
   d. **Data Flow**: Viewer supplies layout and control configurations, while DualAxisLineChart demands this data to update visualizations based on user selections.
6. **Utility (Helper Interface)**
   a. **Supplier**: Utility (helper functions).
   b. **Demanded By**: Various components, including LogicController, Viewer, and ChartController.
   c. **Description**: Utility provides helper functions, such as label generation and data conversions, which support data visualization and formatting.
   d. **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.

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. In this case, it includes the **FMIFetcher**, which communicates with the Finnish Meteorological Institute's API to fetch weather data, **FingridFetcher** class, which communicates with Fingrid's API to fetch energy production data, as well as **FMIParser** and **FingridParser**, which parse the data acquired from the respective API's.

Controller

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 the 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.

3.1 Detailed structure and functions of components

Model: FMIFetcher (Data Fetching Component)

- **Purpose**: Handles communication with the Finnish Meteorological Institute's API to fetch relevant weather data.
- **Functions**:

    - **FMIFetcher()**: Initializes an OkHttpClient for making HTTP requests.
    - **fetchWeatherData()**: Constructs the API request URL and calls executeRequest() with it.
    - **executeRequest()**: Executes the API call and returns the response. It logs success and error messages.
- **Responsibilities**:
    - Build and execute HTTP requests for weather data.
    - Handle and log errors during the data-fetching process.
    - Return weather data as an XML string.

Model: FMIParser (Data Processing Component)

- **Purpose:** Parses and processes weather data to make it usable by the application.
- **Functions**:
    - **parseWeatherData():** parses and processes the data.
- **Responsibilities:**
    - Parse the data from the FMI API.
    - Return weather data in a format usable by the application.

Model: FingridFetcher (Data Fetching Component)

- **Purpose:** Handles communication with Fingrid's API to fetch relevant energy production data.
- **Functions**:
    - **fetchEnergyData():** Constructs the API request URL, executes the call, and returns the response body.
- **Responsibilities**:
    - Build and execute HTTP requests for energy production data.
    - Return energy production data as a JSON string.

Model: FingridParser (Data Processing Component)

- **Purpose:** Parses and processes energy production data to make it usable by the application.
- **Functions**:

    - **parseEnergyData():** parses and processes the data.
- **Responsibilities:**
    - Parse the data from Fingrid's API.
    - Return energy production data in a format usable by the application.

Controller: LogicController (Business Logic Component)

- **Purpose**: Core processing unit for managing application features, such as saved views and data retrieval.
- **Functions**:
    - **saveView()**: Saves user-defined views, consisting of weather type, energy type, and time range.
    - **getSavedViewNames()**: Returns a list of names for all saved views for easy access.
    - **getViewSettings()**: Retrieves the settings associated with a specific saved view.
    - **Incoming functions** for retrieving energy and weather data from the model and returning it for use in the view.
- **Responsibilities**:
    - Coordinate data fetching from the model and provide it to the view.
    - Maintain application state through saved views.
    - Process user interactions and update data accordingly.

Controller: ChartController (Data Visualization Component)

- **Purpose**: Manages the updates to the dual-axis line chart, ensuring that the displayed data reflects user selections.
- **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. For now uses 0 as lower bound but we will determine this once we start using real data.
- **getWeatherDataForType():** Helper method to get weather data based on the selected type and time range.
- **getEnergyDataForType():** Helper method to get energy data based on the selected type and time range.
- **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.

- **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)

- **Purpose**: Responsible for setting up the JavaFX application and launching the user interface.
- **Function**:

- **start(Stage primaryStage)**: The main entry point that initializes the primary stage and scene, integrating the various components of the application.

- **Responsibilities**:
  - Create and display the main user interface.
  - Ensure proper initialization and coordination between the model, controller, and view components.

View: Viewer (User Interface Component)

- **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. This constructor establishes the initial state of the user interface.
  - **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. This method 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. This method ensures 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.
  - **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. This function calls the

updateChart() method from the ChartController class to ensure the displayed data is current.

- **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)

- **Purpose**: Custom chart that supports dual Y-axes for displaying multiple datasets simultaneously.
- **Functions**:
  - **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)

- **Purpose**: Provides utility methods for generating labels and converting data formats.
- **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.
- **Responsibilities**:
  - Support other components by providing commonly used functionalities.
  - Streamline data handling and formatting for better usability.

4. SELF-EVALUATION

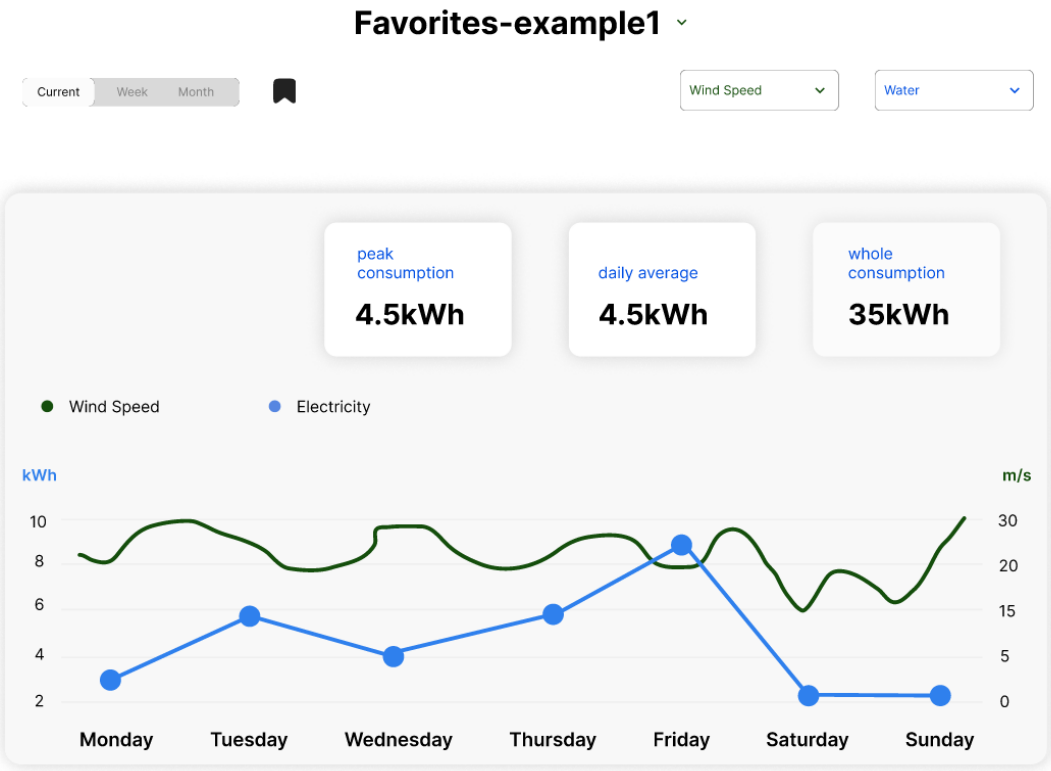
4.1 The UI

Our original UI design and our current implementation can be seen on page 9. The first picture is the Figma design and the second our current implementation. We have been able to implement almost all the main features that were in the design. Features that are missing are the boxes that display consumption values numerically. We will implement those in the next phase once we start using data from the API's instead of the mock data. If we have time, we will also focus more on the overall look of the UI making it look "prettier". Overall, we have been able to stick with the original UI design pretty well, only deviating from it in some minor details. For the UI we haven't had to make any notable changes, and we believe we don't have to moving forward.

4.2 The Design

Our original plan was to follow the MVC pattern, the active model to be specific. We have been mostly able to follow this design pattern in our implementation. 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 done in the Controller. So far, we are only using mock data, so we haven't used the Model at all. We will focus on this in the next phase. We are still following the MVC model, but we have moved more towards the passive MVC model where the Controller interacts with the View too. In the future when we start using the actual data from API's we might move more back towards the active MVC model where View interacts with Controller and then Controller interacts with the Model that pushes the updated info back to the View. We believe that this is the simplest way to implement this app but using passive or semi-passive model are also options if we run into problems trying to use the active mode which is the "strictest" out of the three.

Original



Implementation