Migration & Modularization Report: we_mechload_viewer

Introduction

This document provides a comprehensive analysis of the architectural evolution of the we_mechload_viewer application. It details the migration from the original, monolithic script (we_mechload_viewer.py) to the current, highly-refined modular architecture in modular_version_v2/.

The latest changes represent a maturation of the initial refactoring effort, primarily by **extracting complex business logic** out of the UI and controller layers into a dedicated **analysis layer**. This move completes the transition to a robust, scalable, and maintainable application structure.

Part 1: Architectural Overview - Before and After

To understand the scale of the improvement, it's essential to compare the old and new structures.

The Monolithic Past (we_mechload_viewer.py)

The original application was a classic example of a monolithic design. All functionality was contained within a single, massive WE_load_plotter class spanning nearly 2,000 lines.

- **UI, Data, and Logic Combined:** This single class was responsible for creating the UI, handling user events, loading and processing data, and generating plots.
- **Tightly Coupled:** A method triggered by a button click would directly read from other UI elements, perform mathematical calculations, and update a plot widget all within the same function.
- **The "God Object":** The WE_load_plotter instance held all the application's state, making it difficult to track data flow and leading to unpredictable side effects when making changes.

The Evolved Modular Present (modular_version_v2/)

The new architecture embraces the **Model-View-Controller (MVC)** pattern, but with a clearer distinction for **business logic**. It separates the application into distinct, cooperative layers.

Think of it like a professional restaurant kitchen:

- **Model:** The pantry and the chefs (data_manager.py, analysis/). Manages ingredients and performs the core cooking.

- **View:** The dining room and the menu (main_window.py, ui/). Displays information to the user.
- **Controller:** The head waiter (controllers/). Takes orders from the customer (View) and coordinates the kitchen (Model).

The latest updates introduced a dedicated "chef" (data_processing.py), which was a crucial step. Previously, the "waiter" was doing some of the cooking.

New Directory Structure

```
modular_version_v2/
├── main.py              # Application entry point
└── app/
    ├── main_window.py      # The main application window (View)
    ├── data_manager.py     # Central data store (Model)
    ├── config_manager.py   # Manages application settings (Model)
    │
    ├── ui/              # Self-contained UI components (View)
    │   ├── tab_*.py
    │   └── directory_tree_dock.py
    │
    ├── plotting/
    │   └── plotter.py      # Specialized component for rendering plots (View)
    │
    ├── analysis/         # **KEY ADDITION**: Business Logic (Model)
    │   ├── data_processing.py
    │   └── ansys_exporter.py
    │
    └── controllers/       # Connects View and Model (Controller)
        ├── action_handler.py
        └── plot_controller.py
```

Part 2: The Core Improvement - Extraction of Business Logic

The most significant change in this iteration is the creation of the app/analysis/data_processing.py module.

The Problem It Solves

In the previous modular version, the UI tabs or the controllers still contained some data processing logic. For example, the function to calculate a "rolling FFT" might have lived inside

a method in a controller, which was triggered by a UI event.

This was not ideal because:

1. **It made the controllers "too smart."** A controller's job is to route requests, not perform complex calculations.
2. **The logic was not reusable.** If another part of the app needed to calculate a rolling FFT, it couldn't easily call that controller method.
3. **It was impossible to test the logic in isolation.** You couldn't test the FFT calculation without creating the entire UI and simulating button clicks.

The New Approach: The analysis Layer

The data_processing.py module now contains **pure functions** for business logic.

- **What is a "pure function"?** It's a function that, for the same input, will always return the same output, and it has no side effects (it doesn't modify any external state). These are the building blocks of reliable software.

Example: Calculating a Rolling Envelope

- **Before (Monolith we_mechload_viewer.py):** The update_plots_tab3 method would:
  1. Read the window size directly from self.spin_box_roll_env.value().
  2. Access the main DataFrame via self.df.
  3. Call rolling_min_max_envelope(self.df[column], window_size).
  4. Directly update the plot.
- **After (New Modular Architecture):**
  1. A spin box in app/ui/tab_time_domain_represent.py is changed by the user. It **emits a signal**.
  2. The PlotController in app/controllers/plot_controller.py **receives this signal**.
  3. The controller gets the current DataFrame from the DataManager.
  4. It calls the pure function in the analysis layer:
     results = data_processing.calculate_rolling_envelope(dataframe=df, column=col, window_size=new_value)
  5. The controller takes the results and tells the Plotter in app/plotting/plotter.py to create a new figure with this data.

This new workflow is vastly superior. The core calculation in data_processing.py knows nothing about the UI. You can now **write a separate test script** that imports data_processing and verifies that calculate_rolling_envelope works correctly without ever launching the application window.

Part 3: A Refined Model-View-Controller (MVC) Implementation

With the latest changes, the roles of each component are now crystal clear.

Model (data_manager.py, analysis/, config_manager.py)

- **Responsibilities:**
  - To manage the application's state (the "single source of truth").
  - To contain all the business rules and data manipulation logic.
  - To notify controllers when the state changes.
- **Key Files:**
  - data_manager.py: Holds the main pandas DataFrame. Its only job is to load, store, and provide access to the raw data.
  - data_processing.py: Performs all calculations and transformations on data (FFT, filtering, statistics, etc.).
  - config_manager.py: Handles saving and loading user preferences and application settings.

View (main_window.py, app/ui/, app/plotting/)

- **Responsibilities:**
  - To display the data and state provided by the Model.
  - To be as "dumb" as possible. The View should not contain application logic.
  - To send user actions (button clicks, text changes) to the Controller via signals.
- **Key Files:**
  - main_window.py: The main container that assembles all the UI elements.
  - app/ui/*.py: Each file defines a self-contained piece of the UI, like a single tab or a dock widget. They are responsible for layout and emitting signals.
  - plotter.py: A specialized view component that only knows how to take data and render it as a Plotly chart.

Controller (app/controllers/)

- **Responsibilities:**
  - To act as the intermediary between the Model and the View.
  - To listen for signals from the View.
  - To call methods on the Model to update the state or perform calculations.
  - To take the results from the Model and update the View.
- **Key Files:**
  - action_handler.py: Handles general application actions like opening files, saving settings, etc.
  - plot_controller.py: A specialized controller that manages the complex interactions related to updating plots based on user input from multiple UI sources.

Conclusion and Benefits

This migration from a monolithic script to an evolved, multi-layered modular architecture is the difference between a personal script and a professional software application.

The benefits of this new structure are immense:

1. **Testability:** The separation of business logic into data_processing.py is the single greatest improvement. Core functionality can now be automatically tested, ensuring accuracy and preventing regressions.
2. **Maintainability:** Bugs are now isolated. A calculation error will be in the analysis folder. A layout issue will be in a ui file. This drastically reduces debugging time.
3. **Scalability:** Adding a new feature is a clear, repeatable process:
    - Add the calculation logic to data_processing.py.
    - Create a new UI file in app/ui/.
    - Wire the signals and slots in the appropriate controller.
4. **Readability & Collaboration:** The codebase is now self-documenting. A developer can look at the file structure and immediately understand the application's architecture. Multiple developers can work on the View, Model, and Controllers simultaneously with minimal conflict.

This refined architecture provides a robust and flexible foundation for the future of the we_mechload_viewer application.