Conceptual design for an energy consumption meter simulator (Workshop 4)

Students:

- Andres Jeronimo Ramirez
- David Santiago Ibañez
- Johan Danilo Trujillo

Course: Object-Oriented Programming — Semester 2025-II

Methodology and Deliverables:

1. Layered Design Review

This section presents the updated architectural analysis of the domotic energy-monitoring simulator developed for Workshop 4. The project implements a layered architecture, a widely adopted software design pattern that separates the system into independent layers, each responsible for a distinct set of functionalities. This structure improves modularity, maintainability, scalability, and clarity of the overall implementation.

1.1 Overview of the Layered Architecture

The system is organized into three conceptual layers:

1. **Presentation Layer**
   Manages user interaction and visual output. It includes all graphical components developed with PyQt5, dashboards, and any view-oriented modules.

2. **Business Layer**
   Contains the core logic of the domotic simulation. It models devices, sensors,

measurements, and the energy-consumption behaviors of the system.

3. **Data & Utilities Layer**
   Handles file-based persistence, JSON serialization, and global constants. It operates as the lowest-level layer, not dependent on any upper components.

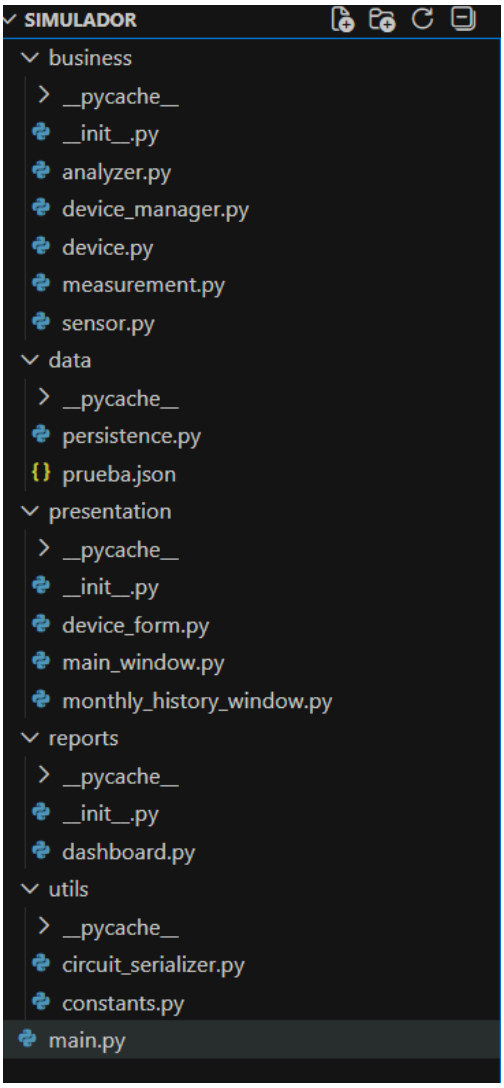
The dependencies intentionally follow a **top-down direction**:

Presentation → Business → Data/Utils

No layer communicates upward, which ensures low coupling and high maintainability.

1.2 Layer-to-Class Mapping

The following classification reflects the actual implemented modules and their responsibilities.



1.2.1 Presentation Layer

Folders: `presentation/`, `reports/`
Role: User interface, visualization, and interaction.

Included modules:

- **presentation/MainWindow.py**
  Primary GUI controller. Displays device lists, dashboard cards (current, daily, monthly consumption), and a real-time consumption chart. Manages the simulation cycle using a QTimer.

- **presentation/DeviceForm.py**
  Form window used to add or edit devices. Collects device attributes and sends updates back to the main GUI through callbacks.

- **presentation/MonthlyHistoryWindow.py**
  Displays the monthly accumulated consumption stored in each device.

- **reports/dashboard.py**
  A visual dashboard/reporting module. Although located in a separate folder, it functions as a presentation component, generating graphical output and providing summary visualizations. It does not contain business logic or persistence logic.

Responsibilities:

- Rendering the user interface.
- Presenting consumption data visually and interactively.
- Collecting user input and forwarding it to the Business Layer.
- Triggering persistence operations (saving and loading circuits).
- Never containing device logic, simulation rules, or serialization mechanisms.

1.2.2 Business Layer

Folder: `business/`
Role: Core domain logic and simulation behavior.

Included modules:

- **business/DeviceManager.py**
  Central controller of the simulation. Stores the list of devices, executes the "tick" simulation cycle, and exposes add/find/remove operations.

- **business/Device.py**
  Represents a domotic device. Manages its measurement history, expected consumption, and monthly accumulated energy.

- **business/Sensor.py**
  Simulates electrical behavior by generating fluctuating voltage and current readings every virtual minute. Produces `Measurement` objects.

- **business/Measurement.py**
  Small data class containing timestamp, voltage, current, power, and

accumulated energy.

Responsibilities:

- Executing the energy-monitoring simulation.
- Generating, storing, and updating device measurements.
- Calculating daily and monthly consumption.
- Remaining independent from UI and persistence concerns.

1.2.3 Data & Utilities Layer

Folders: `data/`, `utils/`
Role: Persistence, serialization, and reusable global constants.

Included modules:

- **data/persistence.py**
  High-level interface for saving and loading circuit files. Delegates serialization details to the utils layer. Ensures proper handling of file paths and I/O operations.
- **utils/circuit_serializer.py**
  Handles JSON serialization/deserialization of the entire system: devices, sensors, histories, and monthly records. Rebuilds a valid DeviceManager instance during loading.
- **utils/constants.py**
  Contains global parameters such as `COP_PER_KWH`, ensuring consistent calculations across all layers.

Responsibilities:

- Converting simulation data into a storable format.
- Reloading circuit configurations on demand.
- Maintaining reusable constants.
- Not depending on the GUI or simulation logic.

1.3 Layer Interactions

Presentation → Business

The user interface calls Business Layer methods to:

- Add or modify devices
- Remove devices
- Retrieve device states and metrics
- Advance the simulation (via QTimer triggers)

Business → Presentation

The Business Layer exposes data structures and metrics.
The UI simply *reads* the data; no business logic is executed in the Presentation Layer.

Presentation → Data & Utils

The GUI invokes persistence functions:

- "Save Circuit" → calls `save_circuit()`
- "Load Circuit" → calls `load_circuit()`

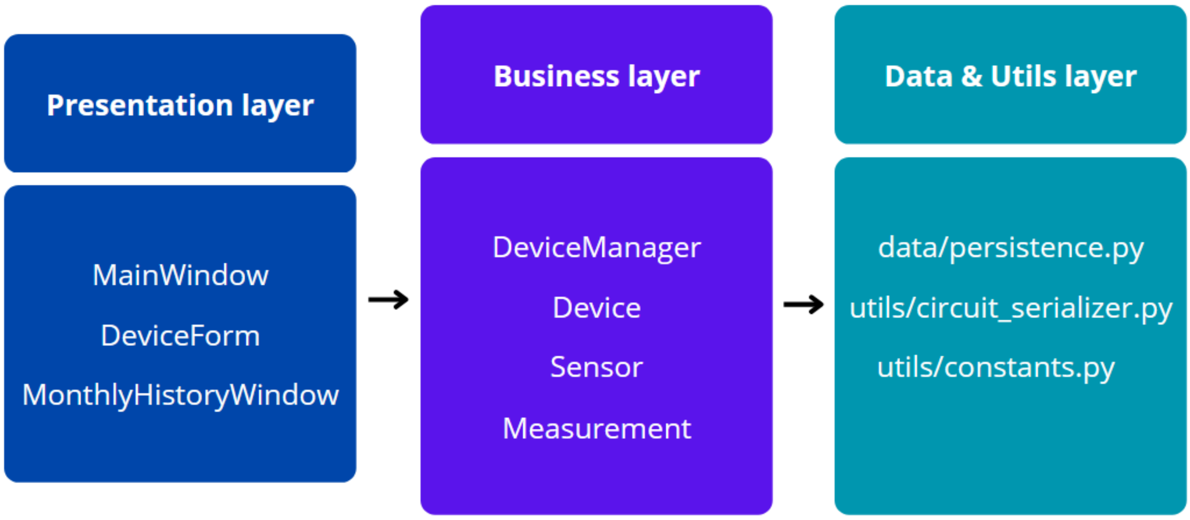
Business → Data & Utils

During save/load operations, device data is passed to serialization utilities.

No upward dependencies

Neither the Business Layer nor the Data Layer interacts with the GUI.
This safeguards the architectural integrity.

1.4 Layered Architecture Diagram



1.5 Summary and Justification

The final architecture demonstrates a clear and correct layered approach consistent with object-oriented principles and the Workshop 4 requirements. The structure ensures:

- High cohesion within each layer
- Low coupling between layers
- Testability, since business logic is GUI-independent
- Scalability, as new devices, visualization tools, or persistence mechanisms can be added without architectural changes
- Maintainability, thanks to well-defined responsibilities across modules

This design fully satisfies the “Layered Design Review” requirement, providing a solid foundation for the GUI prototype, persistence mechanisms, and documentation included in the remaining workshop deliverables.

2. Python GUI Prototype:

2. Python GUI Prototype

This section describes the graphical user interface (GUI) implemented for the domotic energy-monitoring simulator. The prototype was developed using PyQt5, following the project's layered architecture. Although the primary objective of the workshop is to build a functional prototype—not a polished product—the implemented interface provides a clear, intuitive, and interactive environment that successfully demonstrates the functional capabilities of the simulator.

2.1 Overview of the GUI Prototype

The GUI allows users to:

- Add new domotic devices with configurable parameters
- Edit or remove existing devices
- Visualize real-time consumption data
- Display aggregated daily and monthly consumption
- Generate and update energy charts dynamically
- Save and load entire circuit configurations
- Inspect the monthly consumption history
- Interact with a responsive dashboard-style interface

The interface is event-driven and updates once per virtual minute (1 second in real time). This enables continuous monitoring and simulation of energy usage.

2.2 Technologies and Frameworks Used

The GUI was developed using:

- **PyQt5** – for widgets, layouts, dialogs, event handling, and timers

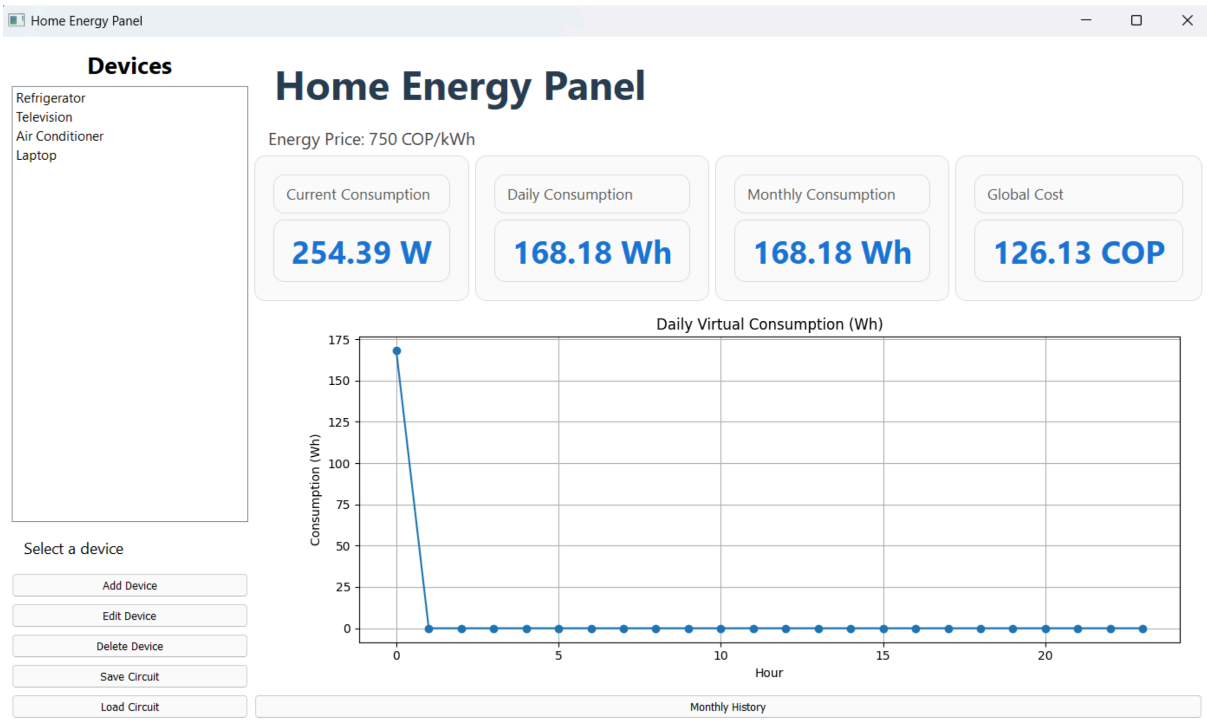
- **matplotlib** – for graphical chart generation
- **Qt event loop** – for real-time simulation updates
- **QTimer** – to simulate electrical readings every second

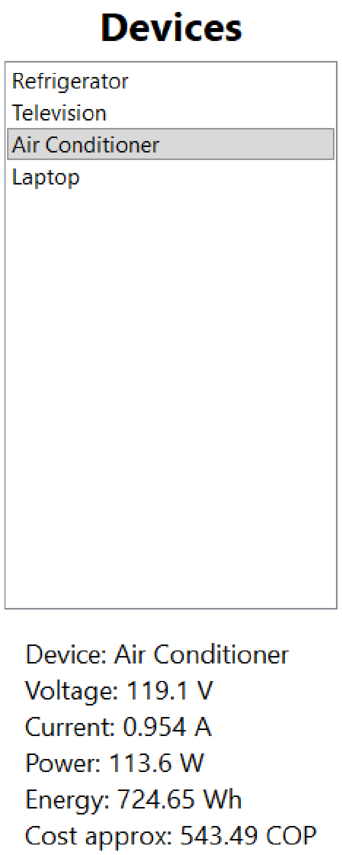
The GUI layer remains fully independent from the Business and Persistence layers, interacting only through clearly defined interfaces.

2.3 Main GUI Components

The GUI is structured around several windows and components:

2.3.1 MainWindow (Primary Interface)





This is the central controller of the user interface. Its responsibilities include:

- Displaying a list of all registered devices
- Rendering three high-level consumption cards:
    - **Current Consumption (W)**
    - **Daily Consumption (Wh)**
    - **Monthly Consumption (Wh)**
    - **Global Cost (COP)**
- Displaying a dynamic, real-time energy consumption chart
- Providing buttons or actions for:
    - Add Device

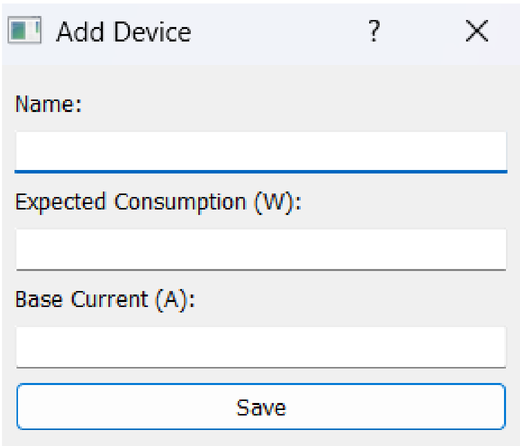
- Edit Device
- Delete Device
- Save Circuit
- Load Circuit
- Open Monthly History

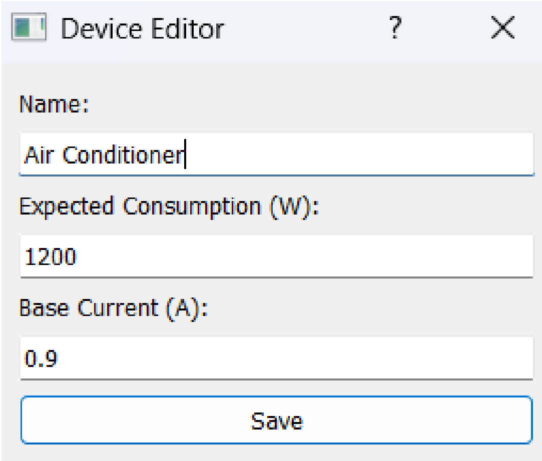
The window includes a live simulation loop powered by `QTimer`. Every tick triggers:

1. Sensor updates for each device
2. Measurement generation
3. Dashboard and chart refresh

This results in a continuously evolving display, simulating real electrical fluctuations.

2.3.2 DeviceForm (Add/Edit Device Dialog)





This dialog window allows the user to create or modify a device. It gathers:

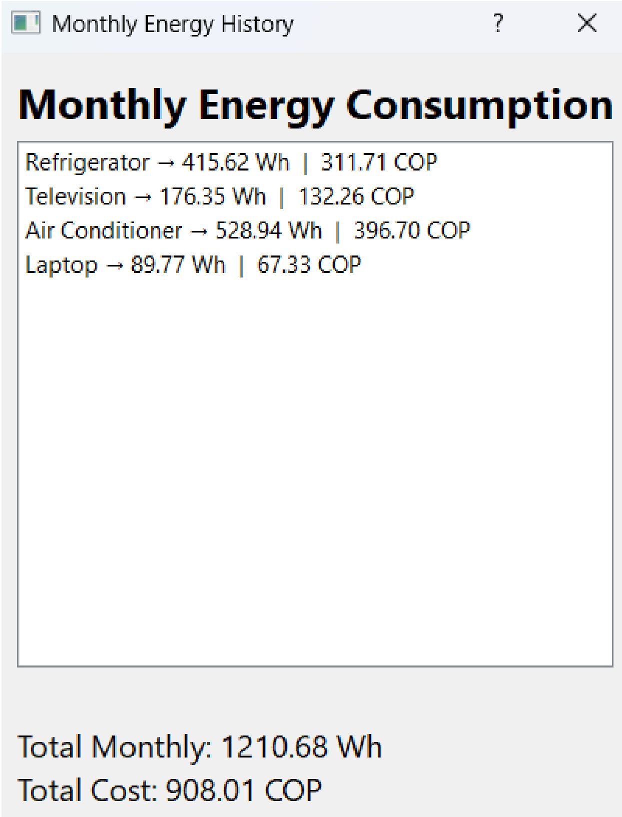
- Device name
- Expected consumption (Wh)
- Base current for the associated sensor

When the user saves the form:

- The Business Layer updates the device list
- The GUI refreshes automatically through a callback
- No business logic is embedded inside the form

This ensures strict compliance with the layered architecture.

2.3.3 MonthlyHistoryWindow



This window provides access to the monthly consumption history stored in `Device.monthly_history`. It allows users to:

- Review day-by-day totals
- Inspect accumulated energy usage per device
- Confirm long-term consumption patterns

This functionality helps validate that the simulator correctly computes and retains long-term usage data.

2.3.4 Visual Dashboard (`reports/dashboard.py`)

Although located in a separate folder, this module functions as a **visual reporting component**. It is part of the Presentation Layer and may:

- Print visual summaries
- Display additional charts or dashboards

- Provide an alternative, non-GUI (but still visual) output

It does not modify the business logic.
It serves as an auxiliary visualization tool.

2.4 Usability and Interaction

The GUI prototype emphasizes functional clarity over aesthetics. The layout is intentionally minimalistic to:

- Highlight system behavior
- Facilitate user interaction during testing
- Avoid visual complexity that could obscure simulation results

Key usability characteristics:

- **Real-time updates:** users immediately see the effects of sensor variations
- **Simple navigation:** all actions are visible and directly accessible
- **Immediate feedback:** changes to the circuit reflect instantly in charts and dashboard cards
- **Persistent state:** Save/Load ensures users can continue a session later

2.5 Integration With the Simulation Engine

The GUI does not perform simulation logic. Instead:

- Every second (virtual minute), the GUI calls `DeviceManager.tick()`
- The Business Layer generates new sensor measurements
- The GUI retrieves updated values through accessor methods

- Charts and dashboard values are refreshed accordingly

This design preserves a clean separation of concerns while allowing continuous visual monitoring of the simulation.

2.6 Prototype Limitations

As per the workshop requirements, the GUI prototype:

- Does **not** implement advanced styling
- Does **not** contain complex animations
- Uses simple default widgets and layouts
- Prioritizes correctness and clarity over appearance

These limitations are acceptable since the prototype focuses on demonstrating functionality, architecture, and interaction workflows.

2.7 Conclusion

The Python GUI prototype successfully meets the workshop requirements by:

- Providing a functional and intuitive interface
- Maintaining strict compliance with the layered architecture
- Supporting real-time simulation of domotic devices
- Allowing users to add, edit, remove, save, load, and inspect devices
- Offering dynamic dashboards and charts for data visualization

The GUI reflects a clear separation between presentation, logic, and persistence, aligned with modern software engineering practices.

3. File-based Persistence:

3. File-Based Persistence

This section describes the persistence mechanisms implemented in the domotic energy-monitoring simulator. The goal of the Workshop 4 requirement is to demonstrate the ability to **save and load circuit configurations** using file-based storage, ensuring that the system's state can be preserved across program executions.

The project implements this feature using the **Data & Utilities Layer**, specifically through the modules `data/persistence.py` and `utils/circuit_serializer.py`, following the layered architecture described earlier.

3.1 Overview of Persistence Approach

Persistence is achieved using a **JSON file** that stores all relevant information about the domotic circuit, including:

- Device names
- Expected consumption
- Sensor configuration
- Base voltage and current
- Measurement history
- Monthly consumption history
- Accumulated energy

The use of JSON ensures that:

- The format is human-readable
- The file can be easily inspected or modified for debugging
- It is portable and platform-independent
- It integrates naturally with Python's built-in JSON module

The system provides two main operations:

1. **Save Circuit** → Serializes the current state into a `.json` file
2. **Load Circuit** → Reconstructs the entire simulator from that JSON file

Both operations are accessible directly from the GUI.

3.2 Modules Involved in Persistence

The persistence implementation is spread across two primary modules.

3.2.1 `data/persistence.py` (Persistence Adapter Layer)

This module acts as a **high-level persistence interface**. It provides simple functions that the Presentation Layer can call:

- `save_to_file(manager)`
- `load_from_file()`

It **does not implement serialization itself**.
Instead, it delegates the conversion to and from JSON to the lower-level module:

- `utils/circuit_serializer.py`

This keeps the responsibilities clearly separated and ensures compliance with layered design principles.

3.2.2 `utils/circuit_serializer.py` (JSON Serialization Engine)

This module contains the core logic for converting Python objects into JSON and reconstructing them later.

It includes two essential functions:

`save_circuit(manager)`

This function:

1. Iterates over all devices
2. Extracts:
    - Device attributes
    - Sensor configuration
    - Measurement history (if included)
    - Monthly history
    - Accumulated energy
3. Converts each object into a serializable dictionary
4. Writes the complete structure to a JSON file

This ensures that the entire simulation state can be restored.

`load_circuit()`

This function:

1. Reads the JSON file
2. Recreates:
    - `Sensor` objects

    - `Device` objects
    - Histories and accumulated energy
3. Assembles all devices into a new `DeviceManager` instance
4. Returns the fully reconstructed system

Because the Business Layer classes (`Device`, `Sensor`, etc.) are pure Python objects without external dependencies, the reconstruction process is reliable and predictable.

3.3 JSON Structure

The JSON file typically contains the following structure:

```
{
  "devices": [
    {
      "name": "Refrigerator",
      "expected_consumption": 150,
      "sensor": {
        "base_voltage": 120,
        "base_current": 0.7,
        "accumulated_energy": 23.54
      },
      "history": [
        { "timestamp": "...", "voltage": 119.8, "current": 0.72,
"power": 86.3, "energy": 5.3 },
        ...
      ],
      "monthly_history": [120, 140, 150, ...]
    }
  ]
}
```

This structure is easy to read and mirrors the internal structure of the Business Layer.

3.4 Integration With the GUI

The Presentation Layer (MainWindow) integrates persistence through simple button actions:

- **Save Circuit** button → calls `save_to_file(manager)`
- **Load Circuit** button → calls `load_from_file()` and replaces the current `DeviceManager`

Because the GUI interacts only with the adapter functions in `data/persistence.py`, the Presentation Layer remains completely isolated from serialization details.

3.5 Persistence Flow

The following process illustrates how persistence operates internally:

Saving

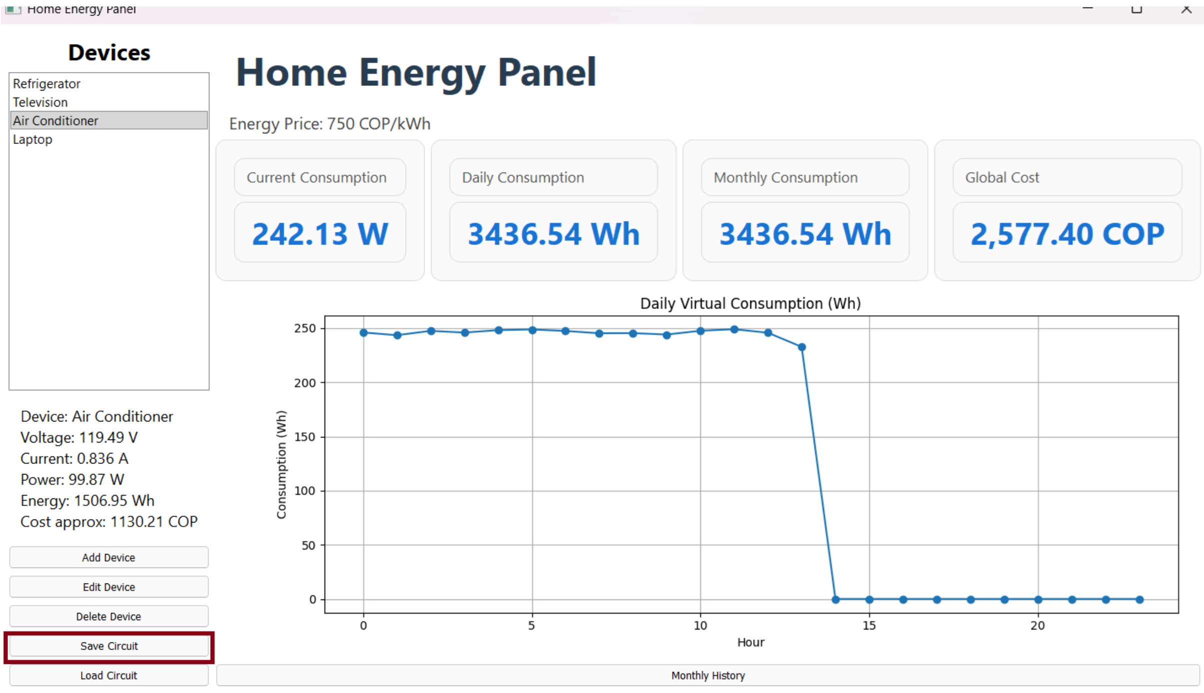
1. User clicks "Save Circuit"
2. MainWindow calls `save_to_file(self.manager)`
3. `persistence.py` delegates to `save_circuit()`
4. `circuit_serializer.py` converts all objects into JSON
5. File is saved inside the project directory

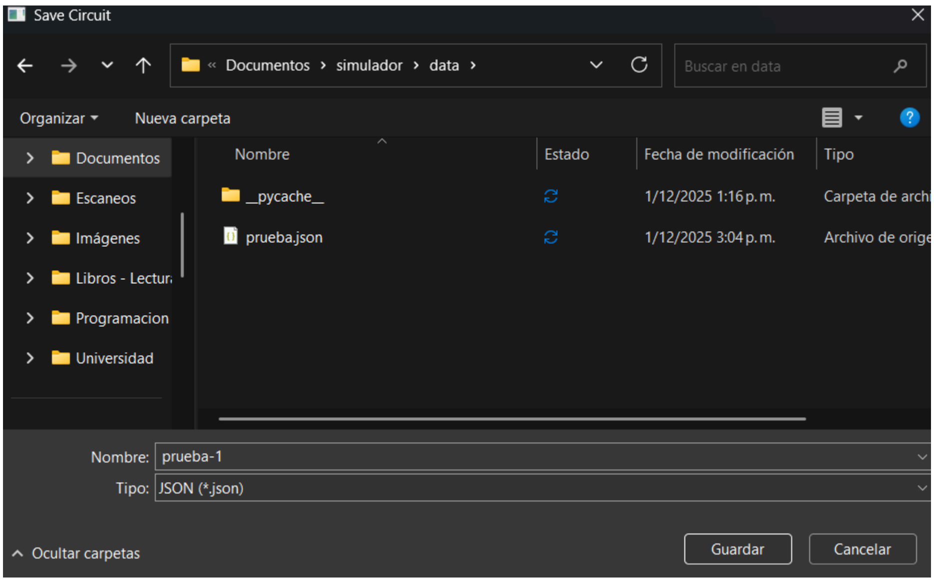
Loading

1. User clicks "Load Circuit"
2. MainWindow calls `load_from_file()`
3. `persistence.py` triggers `load_circuit()`

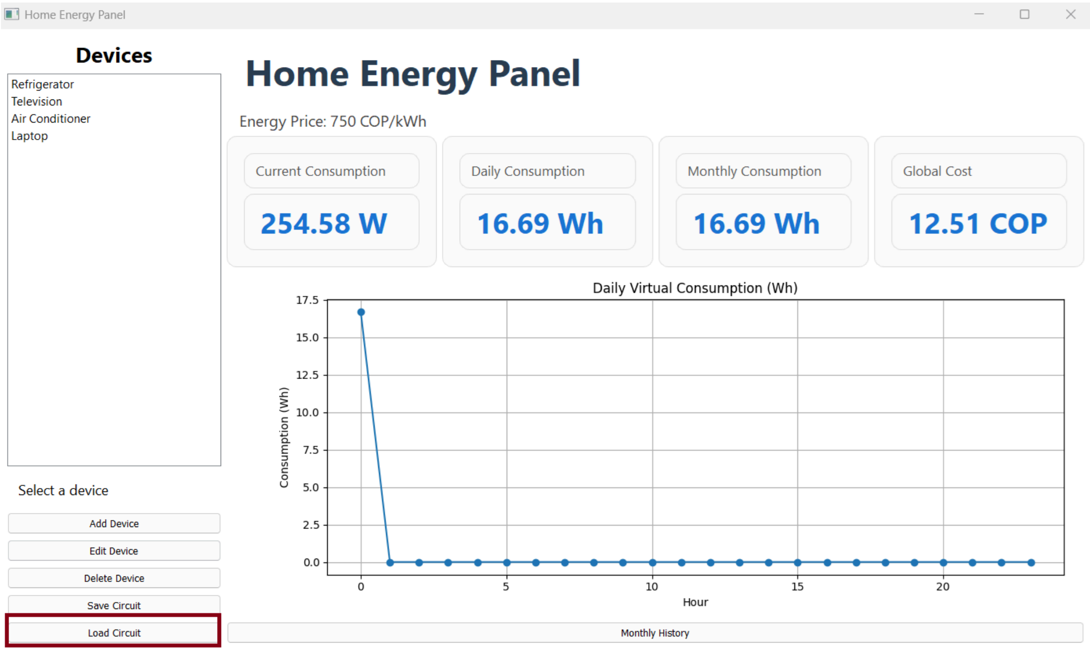
4. JSON is read and parsed
5. New `DeviceManager` is created
6. GUI refreshes automatically

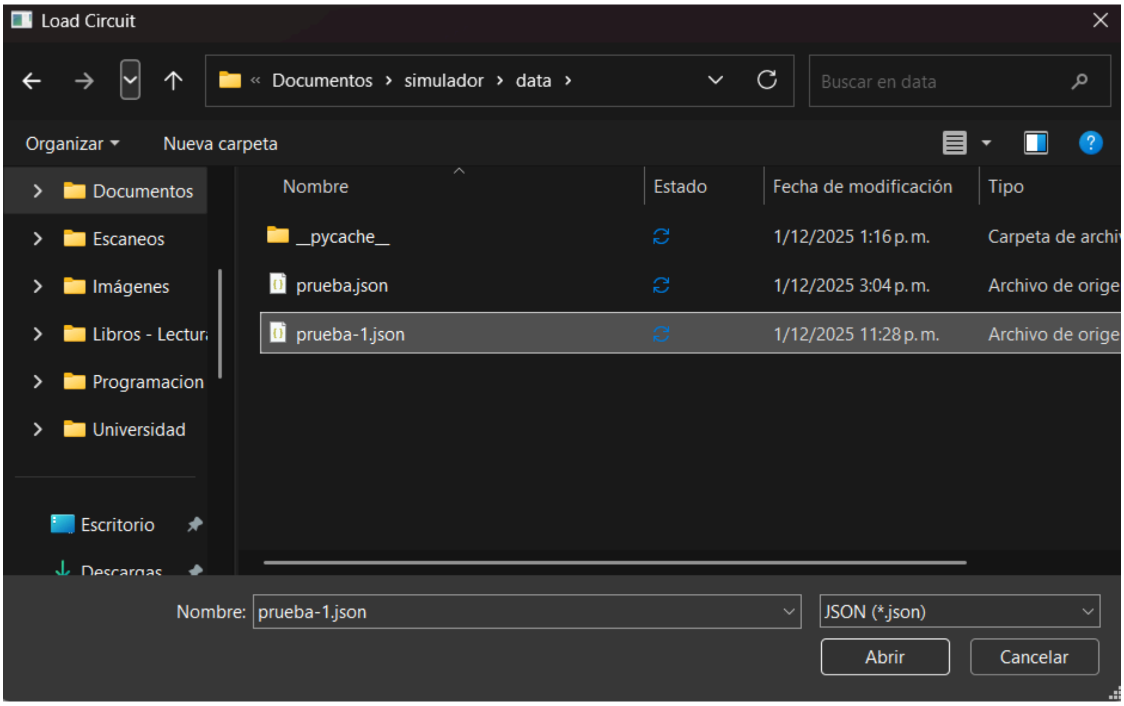
3.6 Demonstration of Persistence

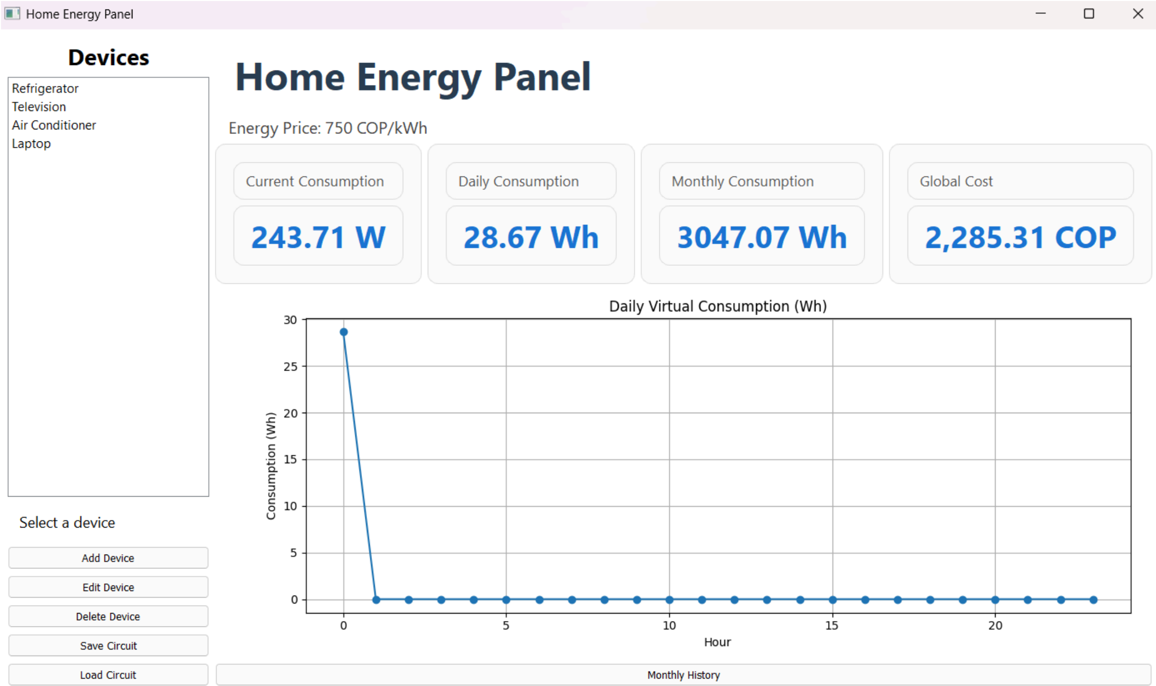




(Closing and opening the application).







The system fully satisfies the workshop requirement of persistence by enabling:

- Saving a circuit
- Exiting the program
- Reloading the same circuit state upon restart

All device configurations, histories, and accumulated energy values are restored accurately.

3.7 Summary

The persistence implementation meets all Workshop 4 expectations:

- JSON-based file storage
- Complete domotic circuit serialization
- Accurate reconstruction of simulation state
- Clear separation between GUI, business logic, and data handling

- High portability and readability

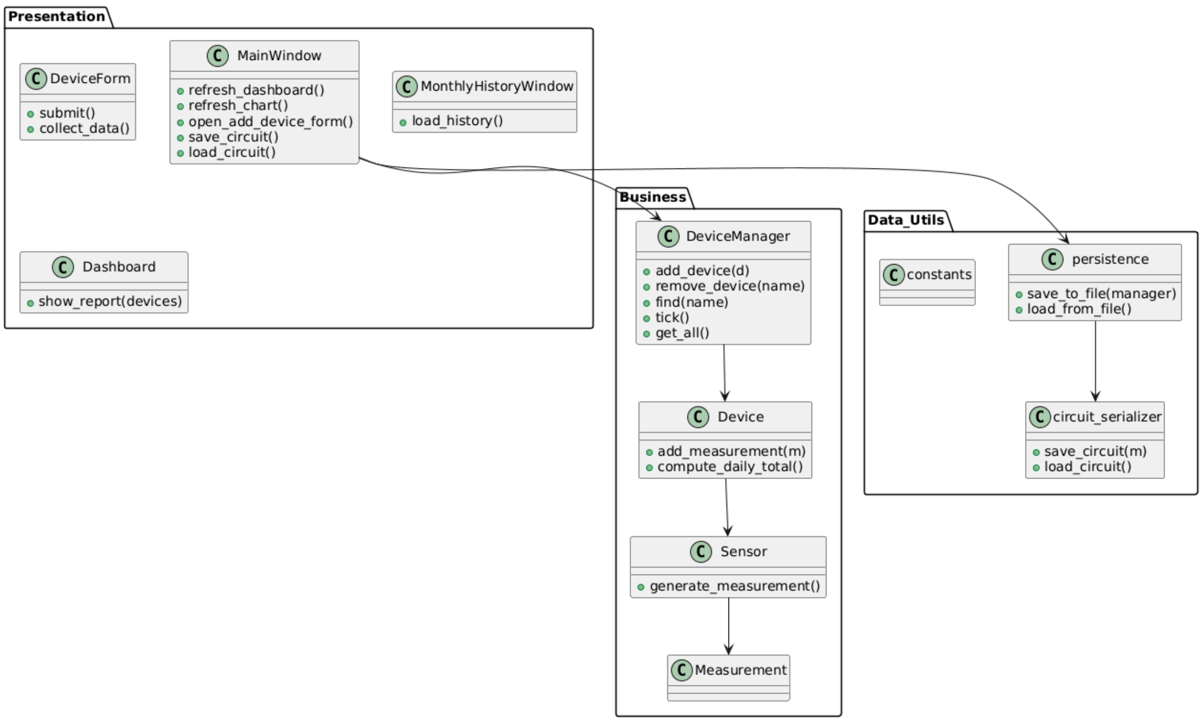
The system ensures reliable and maintainable long-term storage of circuit configurations.

4. UML Diagrams, System Communication, and User Manual

This section presents the formal documentation required for Workshop 4, including UML diagrams and a complete user manual. The diagrams illustrate the interactions between layers and components, while the manual details how to run and use the domotic simulator.

4.1 UML Class Diagram

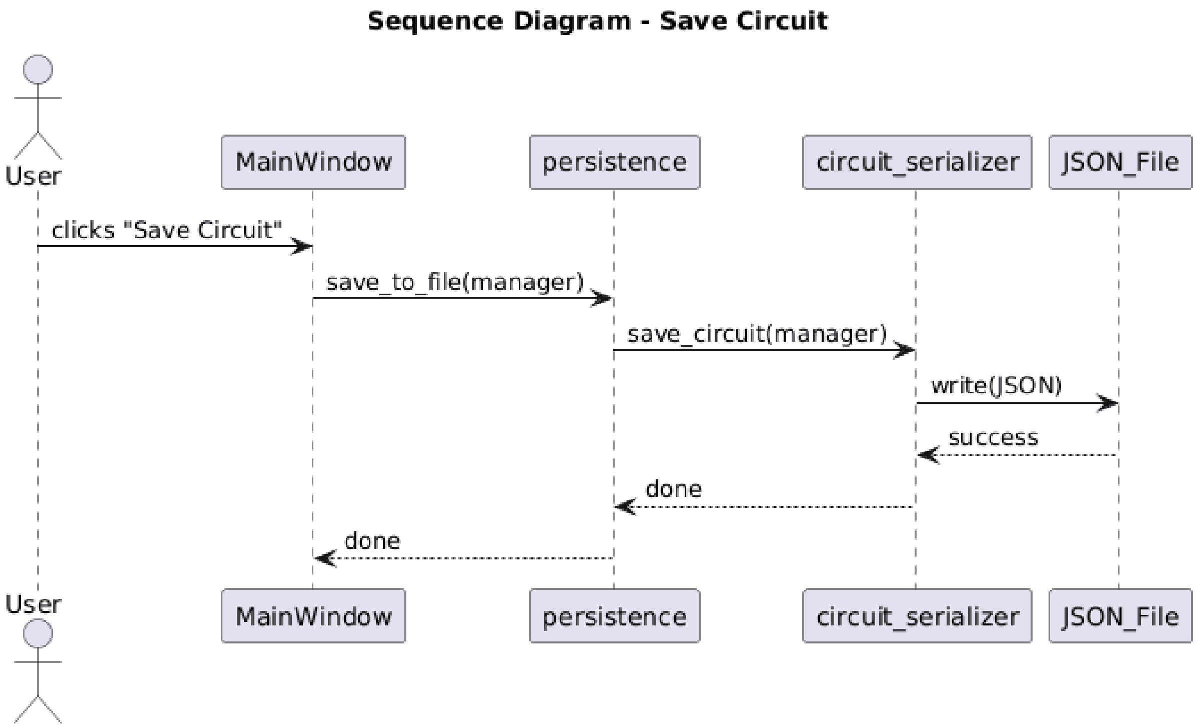
The following textual UML class diagram represents all core system components. Classes are grouped according to their corresponding layer in the architecture.



4.2 Sequence Diagrams

The following diagrams illustrate key interactions between the layers.

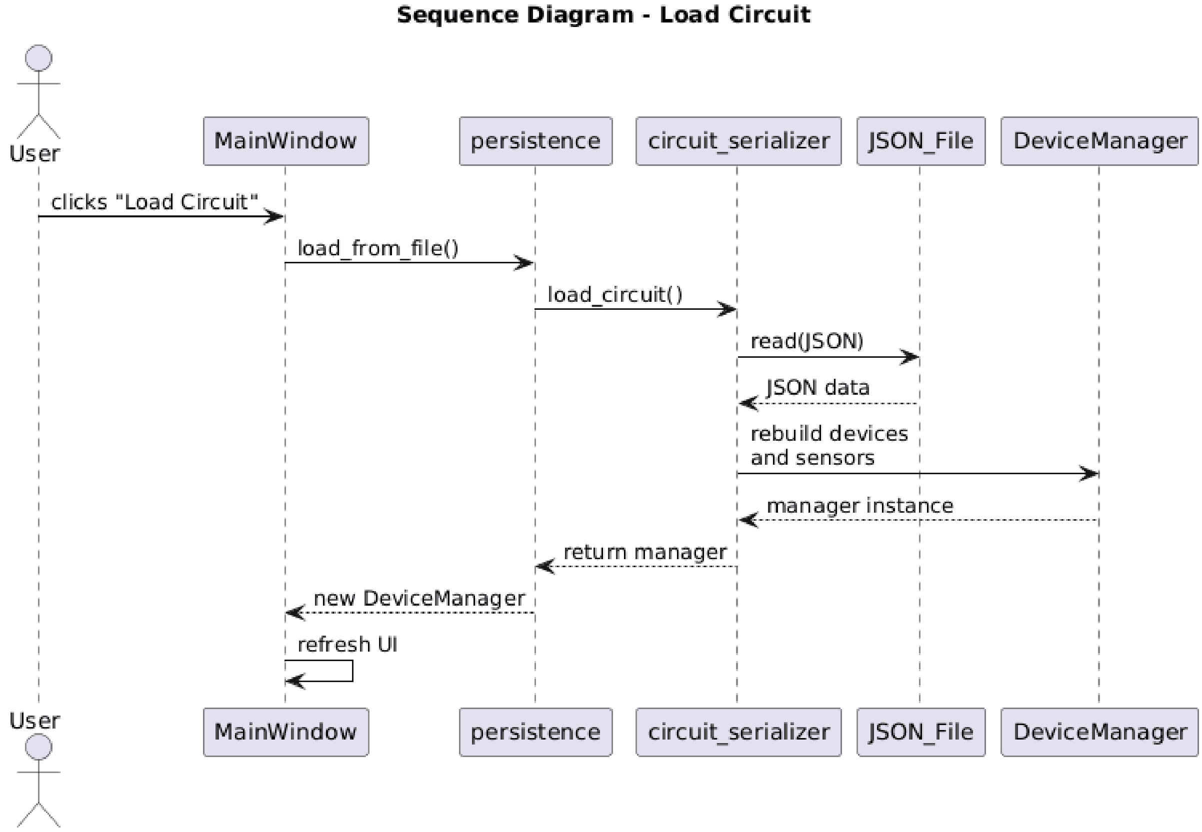
4.2.1 Sequence Diagram: Saving a Circuit



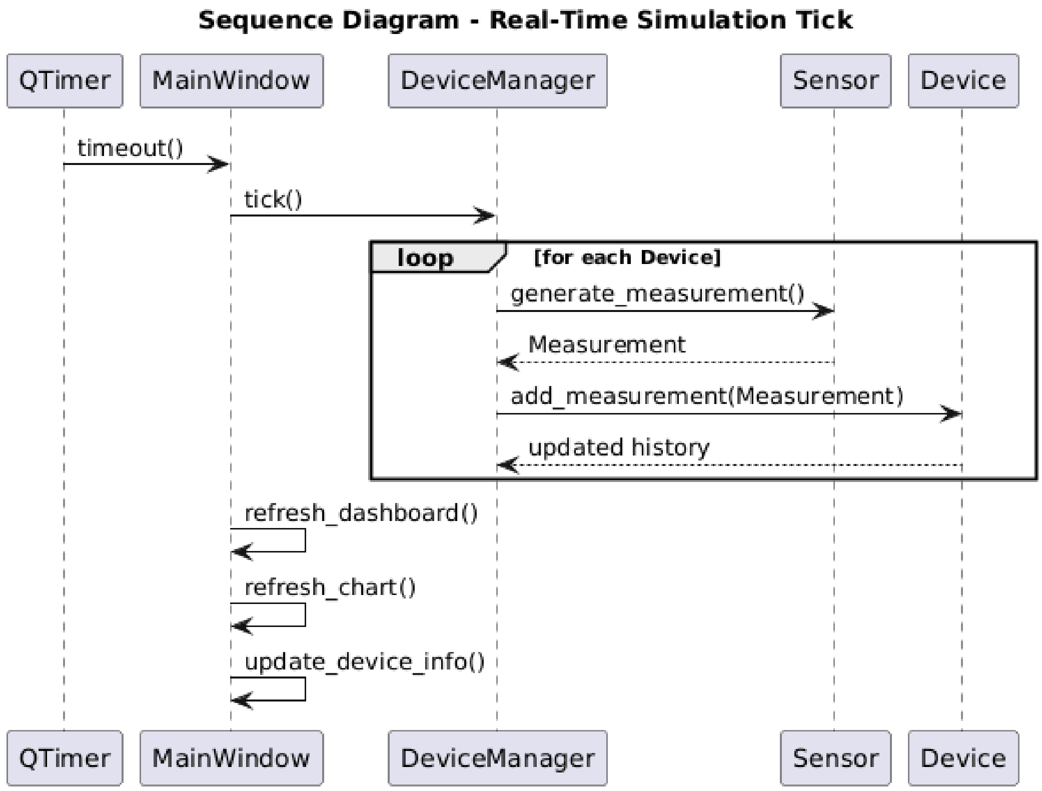
Explanation:

- The GUI triggers persistence.
- The persistence adapter delegates the work to the serializer.
- The serializer creates a full JSON representation of the system state.

4.2.2 Sequence Diagram: Loading Circuit



4.2.3 Sequence Diagram: Real-Time Simulation Tick



Explanation:

- Every second, a virtual minute passes.
- The Business Layer updates all device measurements.
- The GUI refreshes the dashboard and chart.

4.3 User Manual

This user manual explains how to install, run, and operate the domotic simulator.

4.3.1 Requirements

- Python 3.8+
- PyQt5
- matplotlib
- JSON (built-in)

Install dependencies:

```
pip install pyqt5 matplotlib
```

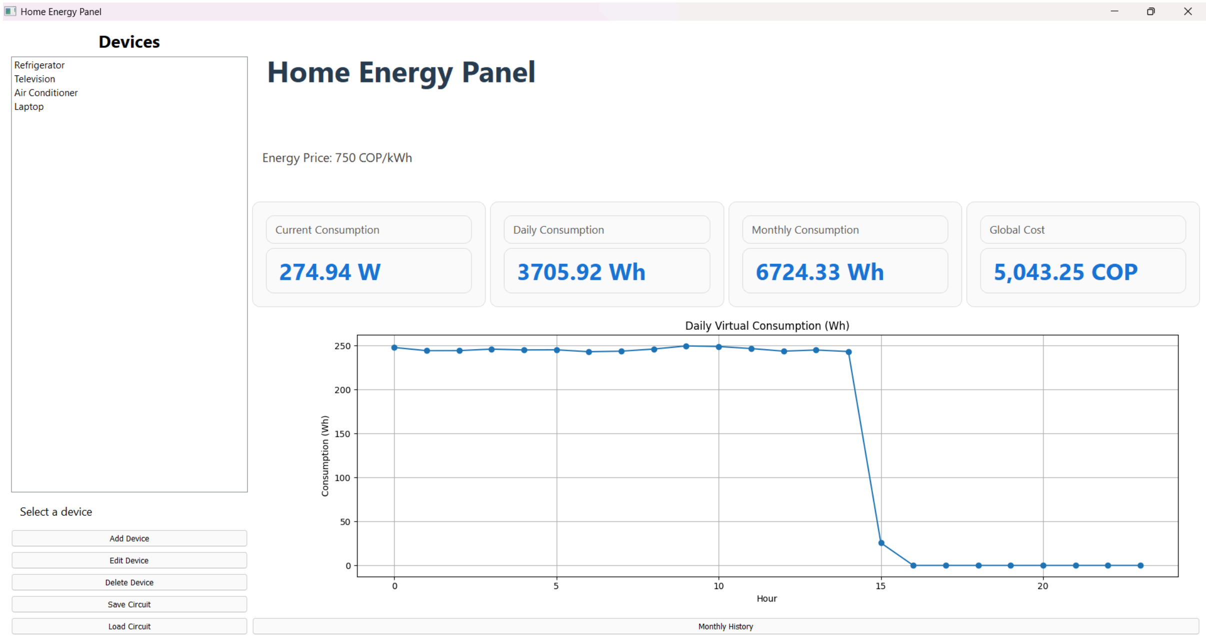
4.3.2 Running the Application

Run:

```
python main.py
```

The Main Window will open automatically.

4.3.3 Main Window Overview



The interface contains:

- **Device List** (left panel)
- **Dashboard Cards** (current, daily, monthly consumption)
- **Real-Time Chart** (center)
- **Buttons**:
  - Add Device
  - Edit Device
  - Delete Device
  - Save Circuit
  - Load Circuit
  - View Monthly History

4.3.4 Adding a Device

1. Click **Add Device**
2. Enter:
    - Device name
    - Expected consumption
    - Base sensor current
3. Click **Save**
4. The device appears immediately in the list
5. Simulation begins automatically (via QTimer)

4.3.5 Editing a Device

1. Select the device from the list
2. Click **Edit Device**
3. Update the fields
4. Click **Save**

4.3.6 Deleting a Device

1. Select the device
2. Click **Delete Device**
3. Confirm removal

4.3.7 Viewing Monthly History

1. Click **Monthly History**
2. A window appears showing stored daily totals for each device

4.3.8 Saving the Circuit

1. Click **Save Circuit**
2. Choose a location
3. A JSON file is generated containing:
    - All devices
    - Sensors
    - History
    - Monthly totals

4.3.9 Loading a Circuit

1. Click **Load Circuit**
2. Select a previously saved JSON
3. The system reconstructs:
    - Devices
    - Sensor states
    - Monthly history
4. The dashboard updates automatically

4.3.10 Simulation Behavior

- Every 1 second = 1 simulated minute
- Sensors generate new measurements
- Consumption data updates in real time

- Daily data clears after 1440 ticks (24 hours)
- Monthly totals accumulate automatically

4.4 Summary

This documentation fulfills Workshop 4 requirements by providing:

- Updated UML class diagram
- Two detailed sequence diagrams
- A complete user manual
- Clear mapping between GUI interactions and system behavior

The diagrams accurately reflect the implemented layered architecture and demonstrate how each layer communicates with the others.