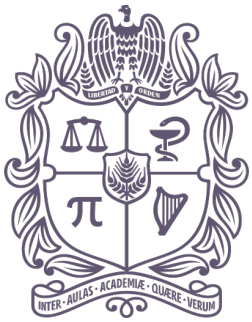

Universidad Nacional de Colombia

Facultad de Ingeniería

Energy Consumption Meter Simulator

Final Report

Course:
Object-Oriented Programming

Professor:
Eng. Carlos Andrés Sierra

Students:
Andrés Jerónimo Ramírez
David Santiago Ibáñez
Johan Danilo Trujillo

December 15, 2025

Abstract

This document presents the final development of the *Energy Consumption Meter Simulator*, a desktop-based software system designed as an academic project for the Object-Oriented Programming course. The application simulates household electrical energy consumption and allows users to manage devices, visualize real-time and historical data, estimate costs, and persist system state through file storage. The project was developed iteratively across four workshops, progressively incorporating user-centered requirements, CRC-based design, SOLID principles, layered architecture, graphical user interfaces, and persistence mechanisms. This report consolidates the complete analysis, design, and implementation of the system into a single, coherent final deliverable.

1 Introduction

Electrical energy monitoring has become an essential component of modern households and smart environments. However, in many real-world scenarios, users only receive aggregated consumption information through monthly billing, which limits their ability to identify inefficient devices or abnormal usage patterns.

From an academic perspective, energy monitoring systems provide an excellent problem domain for applying Object-Oriented Programming (OOP) concepts. Such systems naturally involve interacting entities, evolving states, and multiple abstraction levels, making them suitable for practicing object modeling, responsibility allocation, and architectural design.

This project aims to simulate a domestic energy consumption monitoring system through a desktop application. The focus is not on real hardware integration, but on the correct application of object-oriented analysis, design, and implementation principles. The simulator allows users to register virtual devices, observe real-time energy consumption, analyze historical usage, estimate costs, and manage persistent simulation states.

The development process was structured into four workshops. Each workshop addressed a specific stage of the software lifecycle, including requirements analysis, CRC modeling, SOLID principles, layered architecture, graphical user interface design, and persistence. This final report integrates all workshop results and incorporates the feedback received throughout the semester.

2 Problem Statement

In residential environments, users often lack access to detailed information about how electrical energy is distributed among individual devices. Energy bills usually provide total consumption values without explaining which appliances contribute most significantly to that total. This makes it difficult for users to optimize consumption, detect inefficient devices, or understand the cost impact of daily habits.

Additionally, students learning Object-Oriented Programming frequently struggle to connect theoretical concepts such as encapsulation, responsibility assignment, and design principles with concrete software systems.

The problem addressed by this project is therefore twofold: to simulate an energy consumption monitoring system that presents information in a clear and meaningful way for users, and to serve as an educational artifact that demonstrates correct object-oriented design and implementation practices.

3 Objectives

3.1 General Objective

To design and implement a desktop-based energy consumption meter simulator applying Object-Oriented Programming principles.

3.2 Specific Objectives

- Define functional and non-functional requirements from a user-centered perspective.
- Model the system using CRC cards to identify classes, responsibilities, and collaborations.
- Apply SOLID principles to improve modularity, maintainability, and extensibility.
- Implement a layered architecture separating presentation, business logic, and utilities.
- Develop an interactive graphical user interface for desktop environments.
- Implement file-based persistence to save and restore the simulation state.

4 Requirements Specification

4.1 Functional Requirements

Following the feedback from Workshop 1, functional requirements were explicitly written from the user's perspective, describing what the user can do or observe in the system.

- The user can view real-time energy consumption for each registered device.
- The user can visualize daily and monthly historical consumption.
- The user can add new devices to the system.
- The user can edit the name, expected consumption, and electrical parameters of a device.
- The user can remove devices from the system.
- The user can view aggregated consumption metrics at a global level.
- The user can estimate energy costs based on accumulated consumption.

- The user can save the current simulation state to a file.
- The user can load a previously saved simulation.

4.2 Non-Functional Requirements

Table 1: Non-Functional Requirements

Category	**Description**
Usability	The interface must be intuitive and readable for non-technical users.
Performance	The system must handle multiple simulated devices without instability.
Reliability	The application must remain stable during continuous execution.
Portability	The system must run on major desktop operating systems using Python.
Maintainability	The code must follow modular and object-oriented design principles.
Educational Value	The system must clearly illustrate energy consumption concepts.

5 User Stories

User stories were defined based on the functional scope described in previous workshops and refined using feedback provided throughout the course. Each story is directly supported by the implemented system and can be validated through observable behavior in the graphical interface.

Table 2: User Stories Overview

ID	**User Story**
US-01	**As a user**, I want to view a global energy dashboard so that I can understand overall consumption and estimated cost in real time.
US-02	**As a user**, I want to add new devices so that I can represent the appliances in my household.
US-03	**As a user**, I want to edit existing devices so that I can update their expected consumption and electrical parameters.

ID	User Story
US-04	**As a user**, I want to delete devices so that obsolete or unused appliances do not affect the simulation.
US-05	**As a user**, I want to view detailed information for a selected device so that I can analyze its individual consumption.
US-06	**As a user**, I want to visualize daily energy consumption in charts so that I can identify usage patterns over time.
US-07	**As a user**, I want to review monthly energy consumption and cost so that I can evaluate long-term energy usage.
US-08	**As a user**, I want to save the current simulation state so that I can continue my analysis later.
US-09	**As a user**, I want to load a previously saved simulation so that I can restore devices and historical data.
US-10	**As a user**, I want the system to update automatically over time so that I can observe energy consumption dynamically.

5.1 Acceptance Criteria

Table 3: Acceptance Criteria per User Story

User Story	Acceptance Criteria
US-01	Given the application is running, when the main window is opened, then global consumption metrics are displayed and updated automatically.
US-02	Given valid input, when the user creates a device, then it appears in the device list and affects consumption metrics.
US-03	When a device is edited, then updated values are reflected immediately in the system.
US-04	When a device is deleted, then it is removed from the system and no longer contributes to consumption.
US-05	When a device is selected, then voltage, current, power, energy, and cost information is displayed.
US-06	When the daily chart is displayed, then energy consumption is shown aggregated by hour and updates dynamically.

User Story	Acceptance Criteria
US-07	When the monthly history window is opened, then per-device and total consumption and costs are displayed.
US-08	When the user saves the simulation, then all devices and accumulated data are stored in a file.
US-09	When a saved file is loaded, then the system state and GUI are restored correctly.
US-10	Given the simulation is active, when time advances, then new measurements are generated automatically.

6 CRC Cards

CRC (Class–Responsibility–Collaborator) cards were used as the main object-oriented analysis technique to identify core classes, assign responsibilities, and define interactions before implementation. The following CRC cards correspond directly to the final implementation of the system and reflect the definitive architecture of the application.

6.1 Class: Device

Table 4: CRC Card – Device

Responsibilities	Collaborators
Represent an electrical device	DeviceManager
Store device name and expected consumption	Sensor
Maintain measurement history	Measurement
Provide consumption data to the GUI	MainWindow

6.2 Class: Sensor

Table 5: CRC Card – Sensor

Responsibilities	**Collaborators**
Simulate voltage and current values	Measurement
Calculate power and accumulated energy	Device
Maintain accumulated monthly energy	DeviceManager

6.3 Class: Measurement

Table 6: CRC Card – Measurement

Responsibilities	**Collaborators**
Store voltage, current, and power values	Sensor
Represent a single measurement in time	Device
Provide data for charts and reports	EnergyChart

6.4 Class: DeviceManager

Table 7: CRC Card – DeviceManager

Responsibilities	**Collaborators**
Register, remove, and search devices	Device
Control simulation time progression	Sensor
Trigger periodic measurements (tick)	Measurement
Provide aggregated data to the GUI	MainWindow

6.5 Class: MainWindow

Table 8: CRC Card – MainWindow

Responsibilities	**Collaborators**
Coordinate user interaction	DeviceManager
Display dashboard metrics	DashboardCard
Render charts and device details	EnergyChart, Device
Handle save/load actions	Serializer

6.6 Class: DeviceForm

Table 9: CRC Card – DeviceForm

Responsibilities	**Collaborators**
Collect user input for device creation/editing	Device
Validate device parameters	Sensor
Notify GUI to refresh device list	MainWindow

6.7 Class: DashboardCard

Table 10: CRC Card – DashboardCard

Responsibilities	**Collaborators**
Display a single aggregated metric	MainWindow
Update visual values dynamically	DeviceManager

6.8 Class: EnergyChart

Table 11: CRC Card – EnergyChart

Responsibilities	**Collaborators**
Render historical consumption charts	Measurement
Update visual data dynamically	DeviceManager

6.9 Class: MonthlyHistoryWindow

Table 12: CRC Card – MonthlyHistoryWindow

Responsibilities	**Collaborators**
Display monthly energy summary	DeviceManager
Calculate estimated costs	Sensor

6.10 Utility Components

Table 13: CRC Card – Utility Components

Component	**Responsibility**
Circuit Serializer	Save and load system state to/from JSON files
Constants	Provide global configuration values (COP per kWh)

7 System Architecture

The final system was designed following a layered architecture approach. This decision was made before implementation in order to clearly separate responsibilities, reduce coupling, and facilitate the correct application of object-oriented principles and SOLID guidelines.

Rather than generating the architecture from existing code, the structure was first conceptualized and later validated during implementation, addressing the feedback received in Workshop 2.

7.1 UML Class Diagram

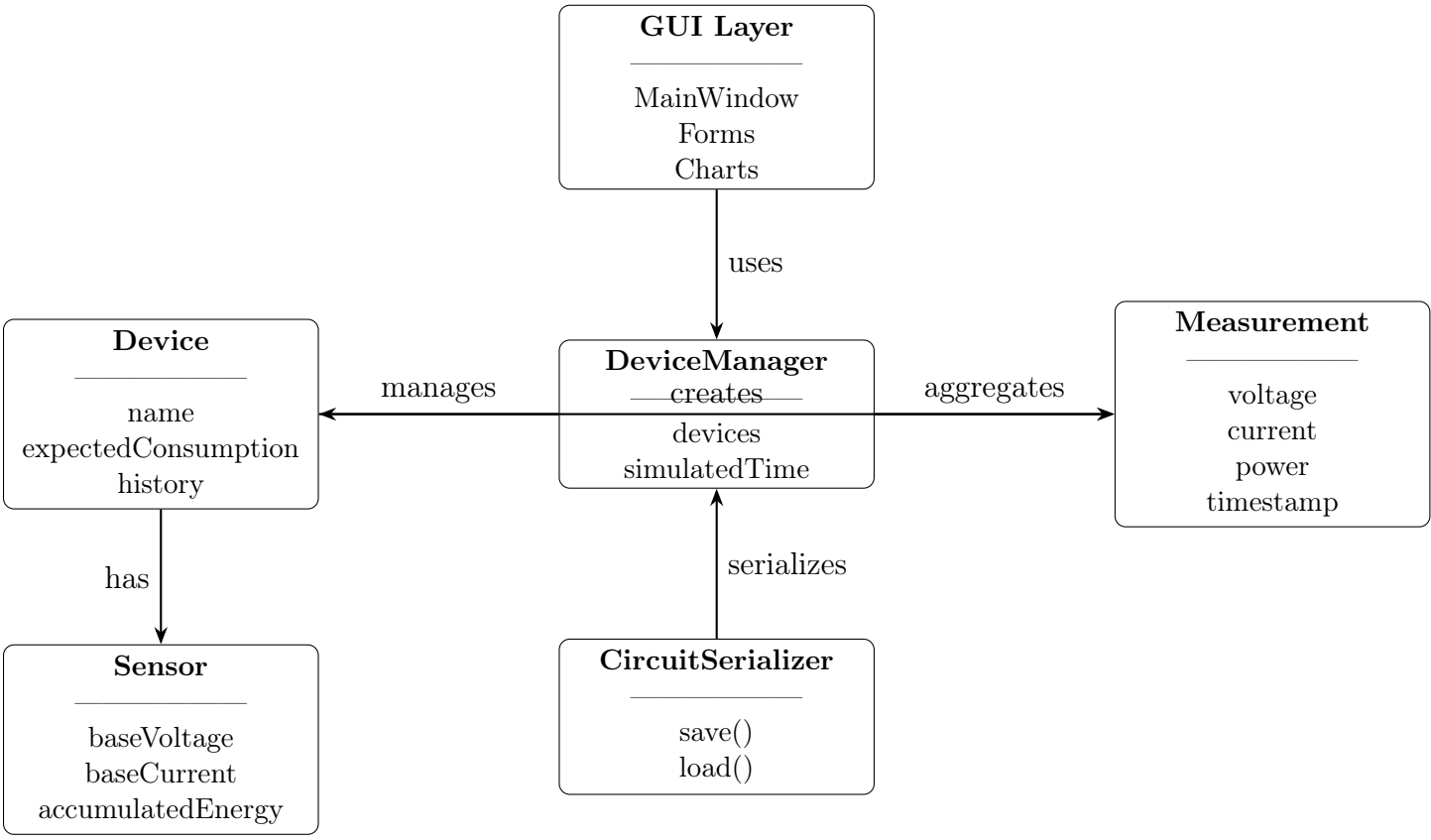

Figure 1: UML class diagram emphasizing responsibility separation and dependencies (design-level)

The layout of the UML diagram reflects the layered design of the system, clearly separating presentation, business logic, and persistence concerns. The diagram was intentionally kept at a design level to avoid implementation bias and to demonstrate responsibility-driven modeling prior to coding.

Table 14: Primary responsibilities of core UML classes

Class	**Main Responsibility**
MainWindow	Orchestrate user interaction, trigger actions, and display real-time and historical energy data through the GUI.
DeviceForm	Collect and validate user input for creating or editing electrical devices.
DashboardCard	Present aggregated consumption metrics in a visual and reusable UI component.
DeviceManager	Coordinate the simulation lifecycle, manage device registration, control time progression, and aggregate measurements.
Device	Represent an electrical appliance, store expected consumption values, and maintain measurement history.
Sensor	Simulate electrical behavior by generating voltage and current readings and accumulating consumed energy.
Measurement	Encapsulate a single electrical measurement, including voltage, current, power, and timestamp.
CircuitSerializer	Persist and restore the complete system state using file-based serialization.

7.2 Layered Architecture Overview

The application is divided into three main layers:

- **Presentation Layer:** Responsible for user interaction and visualization.
- **Business Layer:** Contains the core domain logic and simulation behavior.
- **Utilities and Persistence Layer:** Provides shared services such as serialization and configuration constants.

Dependencies are strictly unidirectional and follow a top-down approach:

$$\text{Presentation} \rightarrow \text{Business} \rightarrow \text{Utilities}$$

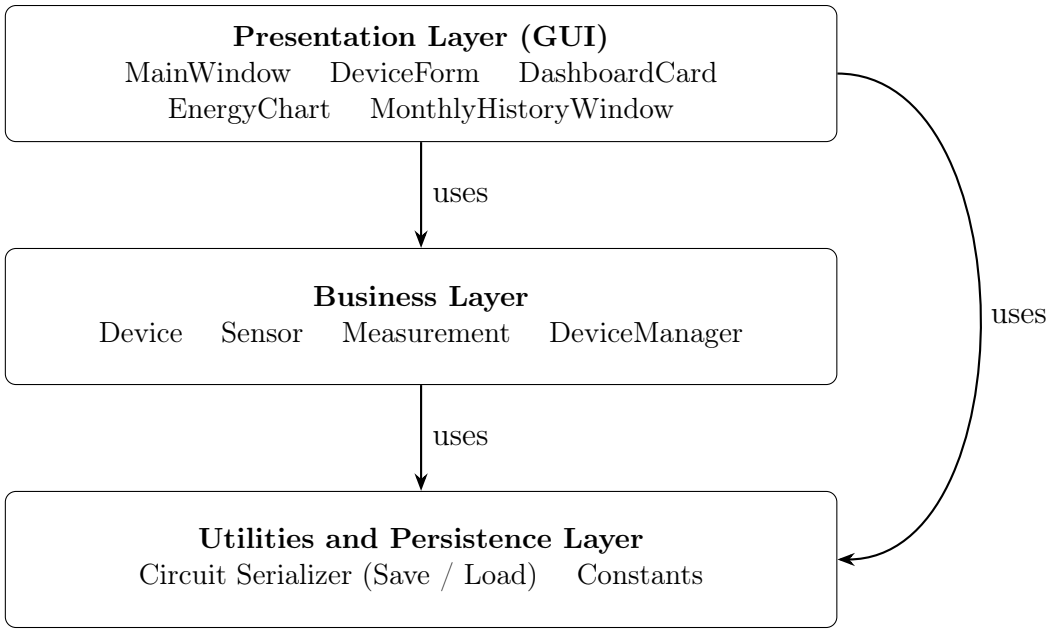

Figure 2: High-level layered architecture showing explicit persistence responsibilities

7.3 Presentation Layer

This layer is implemented using the PyQt5 framework and contains all graphical components and user interaction logic. Its primary responsibility is to present information and forward user actions to the business layer without containing domain logic.

Main classes in this layer include:

- `MainWindow`
- `DeviceForm`
- `DashboardCard`
- `EnergyChart`
- `MonthlyHistoryWindow`

7.4 Business Layer

The business layer contains the core simulation logic and domain entities. It is completely independent of the graphical interface, which allows the simulation to be tested or extended without modifying the GUI.

Key classes include:

- `Device`
- `Sensor`
- `Measurement`
- `DeviceManager`

7.5 Utilities and Persistence Layer

This layer provides cross-cutting services that support the rest of the system. It includes JSON serialization utilities and shared constants, such as the energy cost per kilowatt-hour.

Typical elements include:

- Circuit serializer (save/load functionality)
- Global constants (e.g., COP per kWh)

8 Application of SOLID Principles

One of the main objectives of Workshop 3 was to demonstrate the concrete application of SOLID principles within the context of the project. Instead of limiting the discussion to theoretical definitions, this section explicitly links each principle to specific design decisions and code fragments.

Table 15: SOLID principles applied in the system design

Class	**SOLID Principle**	**Justification**
Device	SRP	The class focuses exclusively on representing a device and its consumption history, without UI or persistence responsibilities.
Sensor	SRP	Responsible only for simulating electrical behavior and accumulating energy.
Measurement	SRP	Encapsulates a single measurement record without additional behavior.
DeviceManager	SRP	Centralizes device coordination and simulation time management, avoiding logic dispersion.
MainWindow	DIP	Depends on abstractions of business logic rather than concrete persistence or calculation implementations.
CircuitSerializer	DIP	Isolated persistence logic prevents business classes from depending on storage mechanisms.
DashboardCard	OCP	New visual metrics can be added without modifying existing dashboard components.

8.1 Single Responsibility Principle (SRP)

Each class in the system was designed to have a single, well-defined responsibility.

For example, the `Device` class is responsible only for representing an electrical device and storing its consumption history. It does not handle user interaction or persistence.

Code Evidence:

- `Device` stores name, expected consumption, sensor, and history.

```
class Device:
    def __init__(self, name, expected_consumption, sensor):
        self.name = name
        self.expected_consumption = expected_consumption
        self.sensor = sensor

        self.history = []           # last 24h (1440 min)
```

```
        self.monthly_history = []  # one entry per simulated day

    def add_measurement(self, m):
        self.history.append(m)
        if len(self.history) > 1440:
            self.history.pop(0)

    def close_day(self):
        daily_wh = sum(m.power * (1 / 60) for m in self.history)
        self.monthly_history.append(daily_wh)
        self.history.clear()
```

Listing 1: Device storing state and measurement history

- `DeviceManager` manages collections of devices and simulation time.

```
class DeviceManager:
    def __init__(self, devices=None):
        self.devices = devices if devices else []
        self.minute_counter = 0
        self.simulated_minutes = 0
        self.day_length = 1440  # 24 hours

    def get_all(self):
        return self.devices

    def add_device(self, device):
        self.devices.append(device)

    def remove_device(self, name):
        self.devices = [d for d in self.devices if d.name !=
            name]

    def find(self, name):
        for d in self.devices:
            if d.name == name:
                return d
        return None

    # 1 tick = 1 simulated minute
```

```
    def tick(self):
        self.minute_counter += 1
        self.simulated_minutes += 1

        for d in self.devices:
            d.add_measurement(d.sensor.generate_measurement())

        # Close simulated day
        if self.minute_counter >= self.day_length:
            for d in self.devices:
                d.close_day()
            self.minute_counter = 0
```

Listing 2: DeviceManager managing devices and simulation time

- `DeviceForm` handles only user input for device creation and editing.

```
from PyQt5.QtWidgets import (
    QDialog, QVBoxLayout, QLabel, QLineEdit, QPushButton,
       QHBoxLayout
)
from business.device import Device
from business.sensor import Sensor

class DeviceForm(QDialog):
    def __init__(self, parent, callback_refresh, device=None):
        super().__init__(parent)

        self.callback_refresh = callback_refresh
        self.device_to_edit = device
        self.created_device = None
        self.edited = False

        self.setWindowTitle("Device Editor" if device else "Add
           Device")
        self.resize(300, 200)

        layout = QVBoxLayout()

        # --- NAME ---
```

```
22          layout.addWidget(QLabel("Name:"))
23          self.input_name = QLineEdit()
24          layout.addWidget(self.input_name)
25
26          # --- EXPECTED ---
27          layout.addWidget(QLabel("Expected␣Consumption␣(W):"))
28          self.input_expected = QLineEdit()
29          layout.addWidget(self.input_expected)
30
31          # --- BASE CURRENT ---
32          layout.addWidget(QLabel("Base␣Current␣(A):"))
33          self.input_current = QLineEdit()
34          layout.addWidget(self.input_current)
35
36          # PRELOAD IF EDITING
37          if device:
38              self.input_name.setText(device.name)
39              self.input_expected.setText(str(device.
                    expected_consumption))
40              self.input_current.setText(str(device.sensor.
                    base_current))
41
42          # Buttons
43          btn_save = QPushButton("Save")
44          btn_save.clicked.connect(self.save)
45
46          layout.addWidget(btn_save)
47
48          self.setLayout(layout)
49
50      def save(self):
51          name = self.input_name.text()
52          expected = float(self.input_expected.text())
53          base_current = float(self.input_current.text())
54
55          # EDIT MODE
56          if self.device_to_edit:
57              self.device_to_edit.name = name
```

```
58              self.device_to_edit.expected_consumption = expected
59              self.device_to_edit.sensor.base_current =
                    base_current
60
61              self.callback_refresh()
62              self.edited = True
63              self.close()
64              return
65
66          # ADD MODE
67          sensor = Sensor(base_voltage=120, base_current=
                base_current)
68          self.created_device = Device(name, expected, sensor)
69          self.callback_refresh()
70          self.close()
```

Listing 3: DeviceForm handling user input only

This separation prevents classes from becoming overly complex and simplifies future modifications.

8.2 Open/Closed Principle (OCP)

The system is designed to be open for extension but closed for modification. This is particularly evident in the way visualization components and device-related logic are structured.

For instance, new dashboard cards or charts can be added without modifying existing classes such as `Device` or `DeviceManager`. The GUI simply queries aggregated data and renders it accordingly.

Example: Additional consumption metrics can be displayed by introducing new `DashboardCard` instances without altering the simulation logic.

8.3 Liskov Substitution Principle (LSP)

Although the current implementation does not heavily rely on inheritance, the design respects LSP by avoiding incorrect subclassing. Domain entities are composed rather than extended when behavior differs.

This design choice reduces the risk of violating substitutability and maintains predictable behavior across the system.

8.4 Interface Segregation Principle (ISP)

The system avoids forcing classes to depend on methods they do not use. Rather than defining large, generic interfaces, responsibilities are distributed across focused classes.

For example, GUI components interact with the business layer only through the methods they require, such as retrieving device lists or aggregated consumption values.

8.5 Dependency Inversion Principle (DIP)

High-level modules, such as the GUI, do not depend on low-level implementation details. Instead, they depend on abstractions provided by the business layer.

For instance, `MainWindow` interacts with the system through the `DeviceManager`, without direct knowledge of how measurements are generated or how energy is accumulated internally.

8.6 Design vs Implementation Consistency

Table 16: Consistency between UML design and final implementation

Design Element	**UML Design**	**Final Implementation**
Layer separation	Clear separation between GUI, business, and persistence	Implemented using PyQt5, business modules, and utility packages
Device–Sensor relation	Composition (Device has a Sensor)	Implemented as a contained object within Device
Measurement creation	Device-generated measurements	Measurements instantiated during simulation ticks
Persistence responsibility	External serializer component	CircuitSerializer handling save/load operations
Dependency direction	GUI depends on business, not vice versa	Enforced through import structure and module separation

This comparison confirms that the final codebase remains faithful to the original object-oriented design decisions, ensuring architectural coherence and facilitating future system evolution.

9 Graphical User Interface Design

The graphical user interface was developed as a desktop application using PyQt5. The design emphasizes usability, clarity, and real-time feedback, ensuring that users can easily interpret consumption data and manage devices.

9.1 Main Window Layout

The `MainWindow` class acts as the central coordinator between user interaction and the business layer. The interface is divided into a sidebar and a main dashboard area.

The sidebar allows users to:

- Select a device
- Add, edit, or delete devices
- Save or load the simulation state

The main dashboard displays:

- Current power consumption
- Daily and monthly energy usage
- Estimated global cost

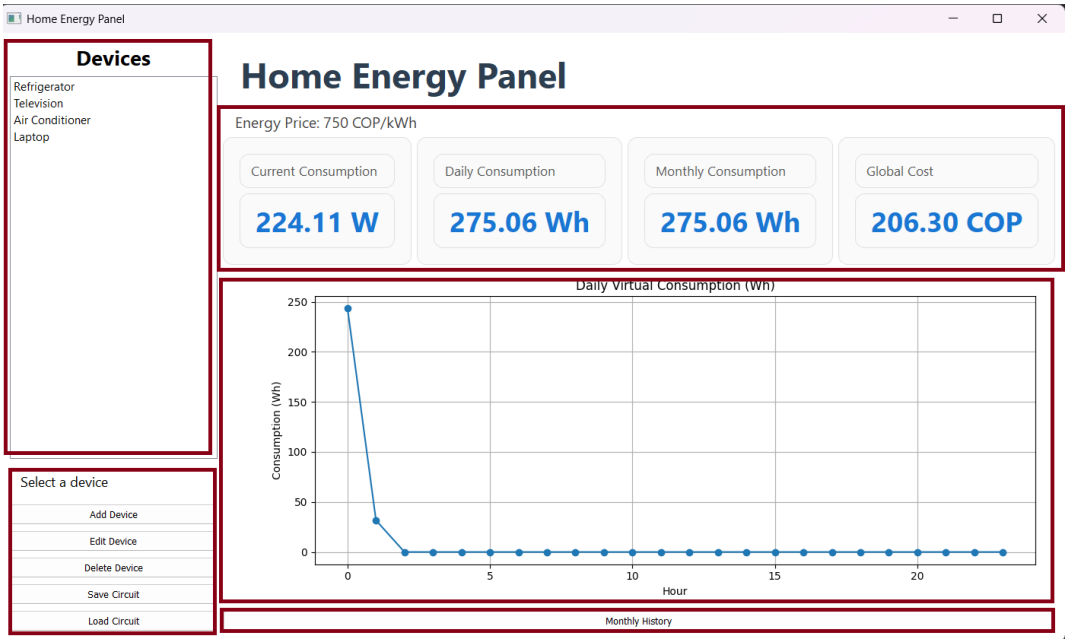

Figure 3: Highlighted architectural component within the system

9.2 Device Interaction Flow

When a user selects a device from the list, detailed information such as voltage, current, power, accumulated energy, and estimated cost is displayed dynamically.

Device creation and editing are handled through a modal dialog (`DeviceForm`), which isolates input logic and prevents inconsistent system states.

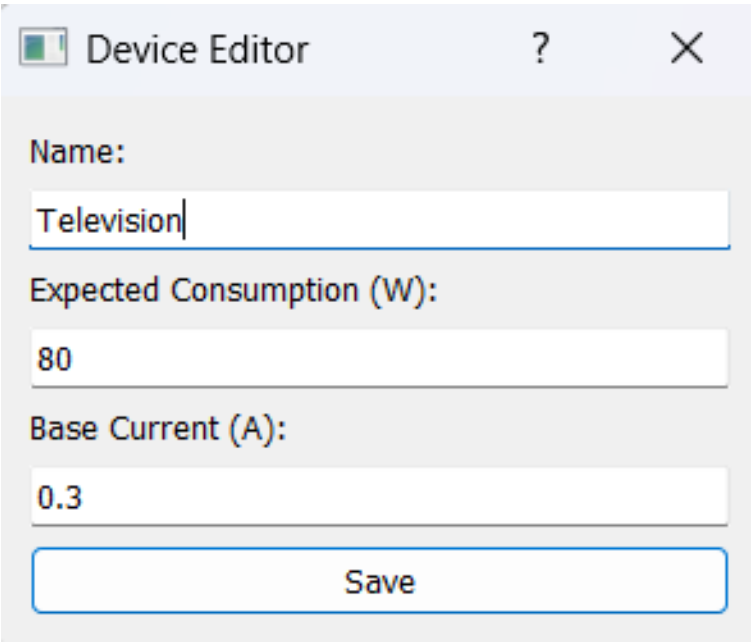

Figure 4: Device creation and editing dialog used to configure electrical devices

Figure 4 illustrates the device creation and editing dialog, which supports both adding new devices and modifying existing ones without directly exposing internal business logic to the user.

9.3 Charts and Visualization

Historical energy consumption is visualized using line charts implemented with Matplotlib. The `EnergyChart` component renders daily virtual consumption aggregated by hour and updates automatically as the simulation advances.

This visualization allows users to identify consumption trends and peak usage periods.

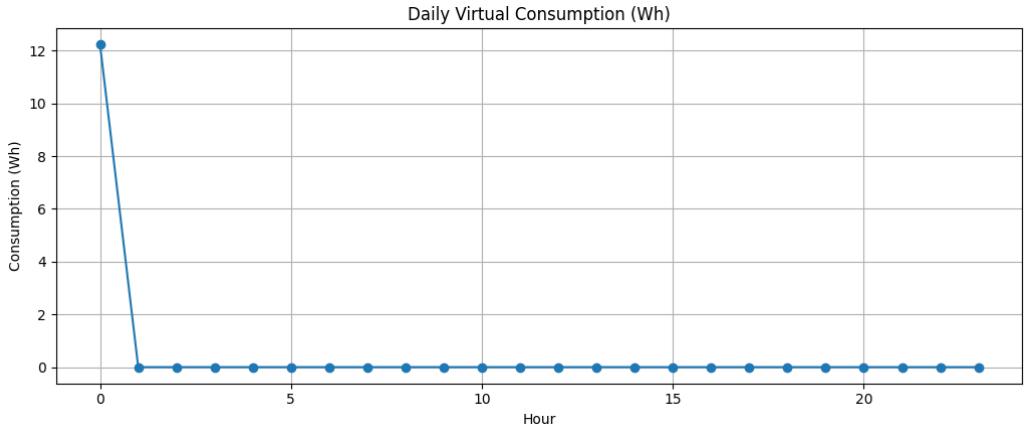

Figure 5: Daily energy consumption chart embedded in the main dashboard

As shown in Figure 5, the chart is updated in real time as the simulation progresses, reflecting the dynamic interaction between the business logic and the presentation layer.

10 Simulation Logic and Time Management

The system simulates electrical energy consumption using a virtual time model. Instead of relying on real-world clocks, the application advances time in discrete steps to allow accelerated observation of consumption behavior.

In the implemented model, one real-time second corresponds to one simulated minute. This design choice allows users to visualize daily and monthly energy consumption within a reasonable execution time.

10.1 Simulation Cycle

The simulation cycle is coordinated by the `DeviceManager` class, which acts as the central controller of time progression. At each simulation tick, the following steps occur:

1. The simulated time counter is incremented.
2. Each device requests a new measurement from its associated sensor.
3. A new `Measurement` object is generated and stored in the device history.
4. Accumulated energy values are updated.
5. Aggregated metrics are recalculated for visualization.

This deterministic cycle ensures consistency between the business logic and the graphical interface.

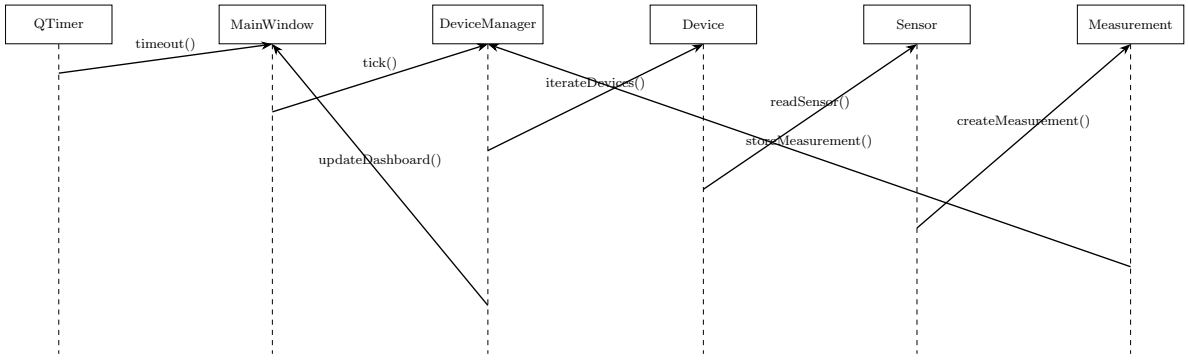

Figure 6: Sequence diagram illustrating the simulation tick and measurement flow

Figure 6 illustrates how the simulation tick propagates from the timer through the presentation and business layers, ultimately resulting in the creation and aggregation of measurement objects. This sequence confirms the clear separation of responsibilities and the unidirectional flow of control.

11 Persistence Mechanism

To allow users to preserve and restore simulation states, the system implements file-based persistence using JSON serialization. This mechanism captures all relevant domain data, including devices, sensors, accumulated energy, and historical measurements.

11.1 Save and Load Process

Persistence is handled through dedicated utility functions that serialize the internal state of the `DeviceManager`. When the user selects the save option, the current simulation state is exported to a JSON file. Loading performs the inverse operation, reconstructing domain objects and refreshing the graphical interface.

This design avoids tight coupling between persistence logic and business classes, maintaining a clean separation of concerns.

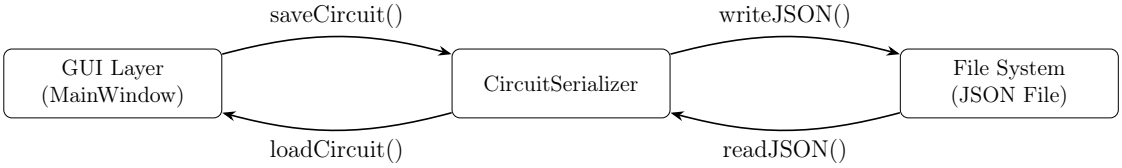

Figure 7: Interaction between GUI, serializer, and file system during save and load operations

Figure 7 illustrates how persistence operations follow a unidirectional flow, where the graphical interface never interacts directly with the file system. This design enforces the Dependency Inversion Principle and improves maintainability and testability.

11.2 Benefits of the Persistence Design

The chosen persistence strategy provides the following advantages:

- Simplicity and transparency of stored data.
- Platform independence.
- Easy debugging and inspection of saved files.
- Adequacy for the academic scope of the project.

12 Monthly Analysis and Cost Estimation

The application includes a dedicated window for reviewing monthly energy consumption (`MonthlyHistoryWindow`). This view aggregates the accumulated energy of each device and estimates associated costs using a predefined cost per kilowatt-hour.

The analysis window presents:

- Energy consumption per device.
- Estimated cost per device.
- Total monthly consumption.
- Total estimated cost.

This feature allows users to identify the most energy-intensive devices and better understand the financial impact of their consumption patterns.

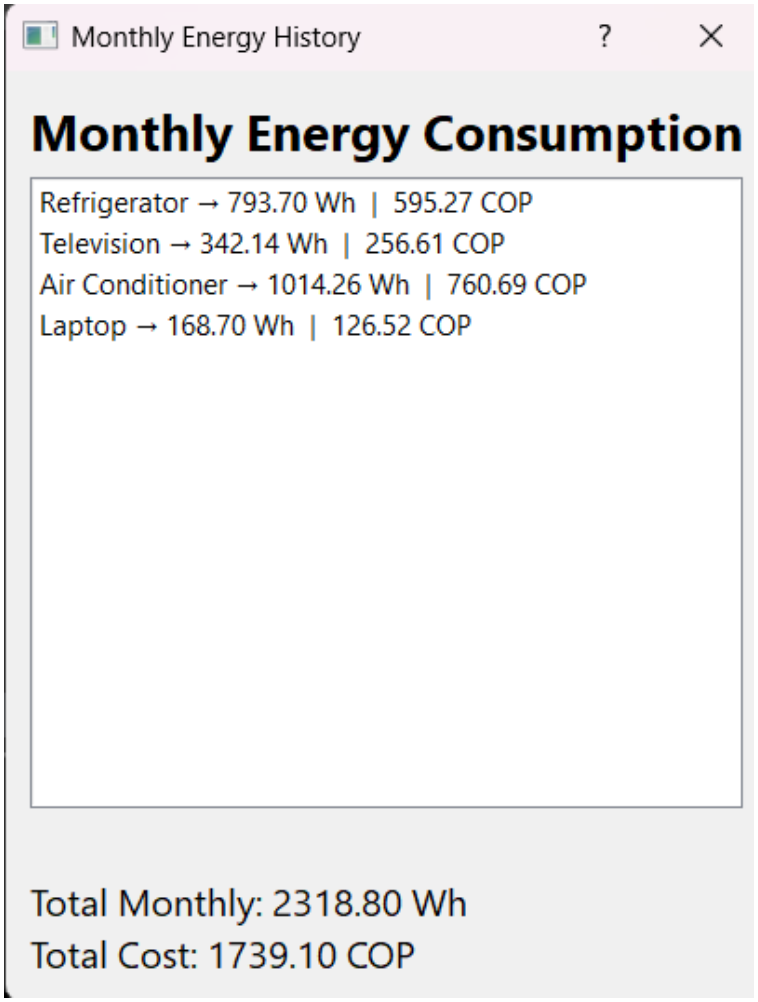

Figure 8: Monthly history window displaying per-device energy consumption and total cost

As shown in Figure 8, the monthly history window summarizes energy usage per device and computes the total estimated cost, offering a clear overview of long-term consumption behavior.

13 Validation and Usability Considerations

While formal usability testing was not conducted as part of this academic project, the graphical interface was iteratively refined based on internal testing and feedback. Emphasis was placed on readability, logical grouping of information, and minimal interaction steps.

Future versions of the system could include structured usability tests to quantitatively evaluate user experience and interface efficiency.

14 Limitations

Despite fulfilling its academic objectives, the system presents certain limitations. The simulator relies entirely on synthetic data and does not integrate with real electrical hardware or smart meters. Consumption behavior follows deterministic rules and does not incorporate stochastic variability or user behavior modeling.

Additionally, the persistence mechanism is file-based and not intended for large-scale or concurrent usage scenarios.

15 Future Work

Several improvements and extensions could be considered for future iterations of the project:

- Integration with real IoT energy sensors.
- Support for multiple users and profiles.
- Advanced anomaly detection techniques.
- Enhanced visualization options and filtering.

16 Final Conclusions

The Energy Consumption Meter Simulator represents a comprehensive application of Object-Oriented Programming concepts within a realistic and educational problem domain. Throughout the development process, theoretical principles were systematically translated into concrete design and implementation decisions.

The final system integrates user-centered requirements, CRC-based modeling, SOLID principles, layered architecture, graphical visualization, and persistence into a cohesive desktop application. Beyond satisfying the requirements of the Object-Oriented Programming course, the project provides a strong foundation for future extensions and serves as a practical reference for object-oriented software design.