Object-Oriented Programming

Semester 2025 II

Domotic Circuit Simulation

Maria Paula Betancourth Hernández

Arley Leonardo Quintana Sepulveda

Juan Esteban Rincón Zambrano

Electronic Engineering

School of Engineering

Universidad Nacional de colombia

IMPLEMENTATION PLAN FOR OOP CONCEPTS

After a review of our classes and functions, we decided to incorporate encapsulation, inheritance and polymorphism:

We use the encapsulation to protect the important data of each class and to give a controlled access to attributes.
The classes Component, User, and ProjectManager will have their state and properties (like voltage, resistance, etc) as private or protected attributes (using _attribute or __attribute).
Access to these values will be managed through getters and setters to ensure the integrity of data.

We think of this encapsulations in our domotic simulator

- Component will encapsulate attributes like type, resistance, voltage, and position, providing methods like set_voltage() or get_properties() to modify them in a safety way.

- User will manage _username and _password privately, allowing secure access through methods like authenticate() instead of exposing credentials directly.

- SettingManager will manage preferences through internal attributes (language, theme, autosave time), modified only by validated functions.

This will help us to have security interactions between classes.

We use the inheritance to reuse and extend common behaviors across different elements of the simulator.
A base abstract class Component will define general attributes and methods for all components (e.g., move(), rotate(), connect(), delete()).
Specific component types like Resistor, Sensor, Relay, and Microcontroller will inherit from Component and extend certain behaviors to represent their individual electrical properties.

We think of this inheritance in our domotic simulator

- The FileHandler class may serve as a base for specialized handlers for different file formats (JsonFileHandler, XmlFileHandler), each overriding methods for reading/writing data.

- UIElement could serve as a base class for interface objects such as buttons, panels, and monitors.

This will help us to extend the simulator when we will add new types of components without having to modify the system.

We use the polymorphism to allow different classes to be treated uniformly through common interfaces.
For instance, all Component subclasses will share the same interface methods (connect(), simulate(), get_data()), but each will perform specific actions according to its type.

The SimulationEngine and UIController will use this feature to interact with all devices without knowing their concrete class.
When executing simulate() on a list of components, each subclass (Resistor, Sensor, Relay) will process its behavior differently but respond to the same method call.

This will help us to support scalability making sure that new devices can be integrated.

The project will follow a modular and object-oriented structure to ensure scalability, maintainability, and logical separation of functionalities. Each folder groups related classes and modules according to their specific roles within the Domotic Circuit Simulator. This organization follows the principles of high cohesion and low coupling, allowing the system to grow and evolve without affecting its core functionality.

The proposed directory structure is as follows:

The proposed structure for the *Domotic Circuit Simulator* is designed to maintain modularity, scalability, and logical organization according to Object-Oriented Programming (OOP) principles. Each directory represents a functional layer of the system, ensuring that different responsibilities are properly separated and that the code remains easy to understand, modify, and extend in future versions.

At the top level, the file **main.py** acts as the entry point of the simulator. This script initializes the main classes (the project manager, the circuit board, and the user interface controller) and establishes the communication flow between them. It serves as the starting point where the system components are created and the simulation process begins.

The **core/** directory contains the fundamental logic of the simulator, representing the physical domain of the system. It includes the abstract class **Component**, which defines common attributes and methods for all components (such as name, position, and connection behavior). Specific component types like **Resistor**, **Sensor**, and **Relay** inherit from this base class and implement their own simulation logic. The **connection.py** module manages the relationships between components, verifying and transmitting data or electrical signals between nodes. This package defines the essential building blocks of the domotic simulation.

The **system/** directory manages the operational logic of the simulator. It contains classes responsible for project management, circuit organization, and real-time simulation. The **CircuitBoard** class handles the placement of components and validation of connections, while the **SimulationEngine** performs electrical calculations such as voltage, current, and power. The **ProjectManager** and **FileHandler** modules handle saving, loading, and exporting project data. This layer ensures that all internal processes function efficiently and consistently.

The **ui/** directory is dedicated to user interaction and graphical representation. The **UIController** captures user actions (such as clicks, drags, or keyboard events) and communicates them to the system. The **UIManager** handles the rendering of graphical elements like buttons, menus, and the workspace. The **Monitor** displays simulation results in tables or graphs, while the **HelpSystem** provides contextual tips and guidance. Together, these classes create an intuitive interface that connects the user with the underlying simulation logic.

The **utils/** directory contains auxiliary modules that support the simulator's operation. The **SettingsManager** handles user preferences such as language, theme, and autosave intervals, while the **Logger** records events, warnings, and system errors. These tools improve the stability, configurability, and maintainability of the application.

Each subdirectory represents a logical layer of the simulator:

- The core package contains all the essential logic related to electronic and domotic components.

- The system package manages the simulation process, file handling, and overall project management.

- The ui package controls the user interface and visual feedback, providing an interactive experience for the user.

- The utils package contains auxiliary tools such as configuration management and error logging.

This modular organization ensures that each class has a well-defined purpose and that the codebase remains easy to understand, extend, and maintain. By structuring the project this way, future developers or students can easily add new components or functionalities without modifying the existing architecture.