# Domotic Circuit Simulator — Layered Architecture

**Object-Oriented Programming – Semester 2025-II**

**Juan Diego Arévalo Bareño**

**November 2025**

**1. Layered Design Review**

For this workshop, the Domotic Circuit Simulator was reorganized using a layered architecture. The design is now divided into three layers: the Presentation Layer, the Business Logic Layer, and the Data Layer.

The Presentation Layer corresponds to the graphical interface built with PyQt5. Its role is to handle user interaction; it displays the workspace, buttons, and component icons. This layer does not execute any actual simulation logic; instead, it forwards user actions to the underlying layers. Keeping the UI isolated from the internal functionality ensures that future changes to the interface will not require modifications to core logic.

The Business Logic Layer contains all domotic components; all behavioral rules, such as how signals are transmitted or how components react, are handled here. This layer is independent from how data is stored or how the interface looks, allowing it to focus exclusively on the correct execution of the simulator's internal logic.

Finally, the Data Layer is responsible for persistence. It provides methods for saving and loading circuit configurations. The Business Logic Layer sends its data to the Data Layer when the user chooses to save a circuit, and it receives reconstructed component objects when the user loads an existing file. By keeping these operations separate, the project can change its storage method later without altering either the interface or the simulation logic.

**2. Python GUI Prototype**

To represent the layered architecture, the UML model was reorganized to show clearly how the responsibilities of the system are distributed across the Presentation, Business Logic, and Data layers. allowing relationships and dependencies to be understood visually.

In the Presentation Layer, the UML diagram includes the classes associated with the graphical interface, such as the main window, the canvas where components are displayed, and any auxiliary controllers required to interpret user actions. These UI classes are shown as depending on the Business Logic Layer, since they request operations such as adding components, starting the simulation, or updating the visual state. The Presentation Layer uses the Business Logic Layer, but the logic classes do not depend on the UI.

The Business Logic Layer appears at the center of the diagram and includes the abstract Component class. The Circuit class aggregates all components and manages the relationships between them, while the Simulator class coordinates the execution of the simulation itself. The UML diagram shows inheritance relationships among the components, associations between the circuit and each component, and the delegation of responsibilities between simulator and circuit.

Finally, the Data Layer includes a storage manager responsible for saving circuit data into a JSON file and reconstructing component objects when loading a circuit. In the UML diagram, this layer depends on the Business Logic Layer because it needs access to the circuit structure and component attributes in order to serialize or deserialize them. However, the Business Layer does not depend on the Data Layer, ensuring the system remains aligned with good architectural practices.

**3. File-based Persistence**

To implement the layered architecture in Python, the project structure was reorganized so that each layer occupies its own dedicated directory and contains only the elements that correspond to its responsibilities. The Presentation Layer contains all the user interface elements developed with PyQt5. These classes do not execute any simulation logic but instead delegate all functional operations to the Business Logic Layer. Their role is limited to interpreting user actions and updating the visual display when necessary.

The Business Logic Layer is isolated in its own Python file, ensuring that the structure remains modular. This layer is completely unaware of the graphical interface and does not interact with any PyQt5 elements, which preserves a strict separation between logic and presentation. The only communication with the Presentation Layer occurs through method calls initiated by the UI.

The Data Layer is implemented as a small module dedicated exclusively to saving and loading information. It contains a storage manager responsible for converting the circuit and its components into a JSON representation and later reconstructing them when the user chooses to load a saved project. This module does not perform any logic beyond serialization and deserialization. It interacts solely with the Business Logic Layer, receiving objects to store and returning reconstructed instances when needed. This design ensures that persistence can be modified or replaced without requiring changes in other interfaces.

**4. Documentation and Submission:**

Each snippet reflects one of the layers: Presentation, Business Logic, and Data, showing how the system begins to form a coherent, modular structure.

The Business Logic Layer is the core of the system. In this layer, the abstract Component class defines the interface and shared behavior for all domotic elements, while specific components such as Switch and Light implement their own logic.

```
#business/component.py
      class Component:
            """Base class for all domotic components in the Business
            Logic Layer."""
            def init(self, name):
                  self.name = name
                  self.connections =    []

            def connect(self, other):
                  self.connections.append(other)

            def simulate(self, signal=None):
                  """Override in subclasses."""
                  pass

#business/switch.py
      from .component import Component

      class Switch(Component):
            """A switch that toggles ON/OFF and sends signals to other
            components."""
            def init(self, name):
                  super().init(name)
                  self.state = False

            def toggle(self):
                  self.state = not self.state

            def simulate(self):
                  for target in self.connections:
                      target.simulate(self.state)
```

```
#business/light.py
     from .component import Component

     class Light(Component):
          """A light that turns ON when receiving a True signal."""
          def init(self, name):
               super().init(name)
               self.is_on = False

          def simulate(self, signal=False):
               self.is_on = signal
               print(f"Light '{self.name}' is {'ON' if self.is_on
               else 'OFF'}")
```

The **Presentation Layer** manages user interaction, relying entirely on the Business Logic Layer to perform real functionality. The following snippet shows a minimal PyQt5-based interaction where pressing a button toggles a switch.

```
#presentation/main_window.py
     from PyQt5.QtWidgets import QPushButton
     from business.switch import Switch

     class MainWindow:
          """Presentation Layer: handles UI events and delegates logic
          to the Business Layer."""

     def __init__(self):
          self.main_switch = Switch("Main Switch")

          self.toggle_button = QPushButton("Toggle Switch")
          self.toggle_button.clicked.connect(self.on_toggle_clicked)

     def on_toggle_clicked(self):
          self.main_switch.toggle()
          self.main_switch.simulate()

          print("UI updated according to the new switch state.")
```

The **Data Layer** manages storage and loading operations. A simple StorageManager is responsible for serializing the circuit structure into JSON and restoring it later. This layer interacts *only* with the Business Logic Layer, ensuring no circular dependencies:

```
#data/storage.py
     import json
     class StorageManager:
          """Data Layer: Handles saving and loading        circuits
          in JSON format."""
     @staticmethod

     def save_circuit(circuit, path):
          data = {
          "components": [
           {"name": c.name, "type": c.__class__.__name__}
           for c in circuit.components
        ]
    }
    with open(path, "w") as file:
        json.dump(data, file)

@staticmethod
def load_circuit(path):
    with open(path, "r") as file:
        data = json.load(file)
    return data
```