|

Domotic Circuit Simulator — Applying SOLID Principles

Object-Oriented Programming – Semester 2025-II

Juan Diego Arévalo Bareño

November 2025

1. SOLID Principles Analysis

The first principle, Single Responsibility Principle (SRP), ensures that each class in the simulator has a single, well-defined purpose. This is reflected in the current design: the Switch class focuses exclusively on managing its state and sending its signal; the Light class is only responsible for updating its state, and the Circuit class manages components and their connections. By separating concerns in this way, each class remains easier to modify and test.

The Open/Closed Principle (OCP) also plays a crucial role in the simulator. The base class component allows the system to be extended with new component types without altering existing code. Developers can create new subclasses that provide their own behavior simply. This makes the system open for extension but closed for modification.

The design additionally supports the Liskov Substitution Principle (LSP), which states that subclasses should be usable anywhere their parent class is expected. In the simulator, the circuit class treats every component as what it is, allowing Switch, Light, or Sensor objects to be substituted interchangeably.

The Interface Segregation Principle (ISP) can prevent classes from becoming unnecessarily large or forcing subclasses to implement irrelevant methods. The design must incorporate smaller, more specific interfaces, such as an Activatable interface for devices that respond to signals. This avoids overloading the inheritance structure and ensures each component only depends on the functionalities it requires.

Finally, the Dependency Inversion Principle (DIP) is visible in the relationship between high-level and low-level components of the system. The Simulator does not depend on concrete component implementations like Switch or Light; instead, it interacts with them. This reduces coupling within the system and ensures that new components can be introduced without changing the simulator core logic.

2. Updated UML and CRC Cards

After applying SOLID principles to the design of the *Domotic Circuit Simulator*, several adjustments were incorporated into both the UML diagrams and the CRC cards. The first major modification is the inclusion of a more explicit abstract base class, which now serves as the central abstraction for all domotic elements. This class defines the core interface and establishes a clear contract for subclasses. By reinforcing this abstraction, the UML represents the Open/Closed and Liskov Substitution principles.

To incorporate the Interface Segregation Principle, the updated UML introduces optional lightweight interfaces. For example, an Activatable interface is shown to represent any component that reacts to incoming signals, while a Detectable interface describes components (such as sensors) that generate readings.

The Switch, Light, and Sensor classes reflect more refined responsibilities: the switch toggles and transmits its state, the light updates its activation state, and the sensor produces a value based on environmental conditions.

3. Python Code Snippets:

- **Single Responsibility Principle (SRP):** The state-handling logic of the switch is isolated in a dedicated class. Instead of mixing simulation behavior with state control, the following snippet focuses exclusively on managing the ON/OFF value.

```
class SwitchStateController:
      def __init__(self):
            self._state = False

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

      def get_state(self):
            return self._state
```

  This separation keeps the switching logic independent from rendering or circuit-wide behavior, ensuring that each class has only one reason to change.

- **Open/Closed Principle (OCP):** This one is illustrated through the abstract Component class, which can be extended to create new domotic elements without modifying its internal implementation.

```
class Component:
      def __init__(self, name):
            self._name = name
            self._connections = []
```

```
    def connect(self, other):
        self._connections.append(other)
    def simulate(self):
        pass
```

- **Liskov Substitution Principle (LSP):** This appears naturally in the simulation loop. Because all domotic components inherit from the abstract Component class, the simulator can execute them interchangeably without checking their concrete types.

```
def run_simulation(components):
    for c in components:
        c.simulate()
```

- **Interface Segregation Principle (ISP):** The ISP is applied by introducing light, focused interfaces such as Activatable, so that subclasses only implement the methods they truly need. This avoids forcing methods onto unrelated components.

```
class Activatable:
    def activate(self, signal):
        raise NotImplementedError

class Light(Component, Activatable):
    def __init__(self, name):
        super().__init__(name)
        self._is_on = False

    def activate(self, signal):
        self._is_on = signal
```

- **Dependency Inversion Principle (DIP):** Finally, The DIP is demonstrated in the structure of the Simulator class. Rather than depending on specific concrete components such as Switch or Light, the simulator relies solely on the abstract Component interface. This allows new component types to be added without changing the simulator code.

```
class Simulator:
    def __init__(self, circuit):
        self._circuit = circuit
```

```
def start(self):
     for component in self._circuit.get_components():
         component.simulate()
```

4. Reflection:

Applying the SOLID principles to the design of the Domotic Circuit Simulator has been a meaningful learning experience. Before this workshop, my focus was mainly on making the simulator functional, but I had not considered in depth how the internal structure of the code could influence long-term maintainability or scalability. Working with SOLID made me realize that good software design is not only about making a program work but about ensuring that it can evolve and improve without becoming harder to understand or modify, This experience will influence how I approach future projects, encouraging me to design with clarity, extensibility, and modularity from the start.

One of the main insights I gained is how valuable it is to separate responsibilities clearly. When I first created the classes for the simulator, some of them handled too many tasks at once. By applying the Single Responsibility Principle, I identified the parts of each class that did not belong there and reorganized them into more focused components. The Open/Closed and Liskov Substitution principles also helped me understand how to extend the simulator without constantly rewriting existing code. Designing with inheritance and polymorphism in mind allowed me to add new domotic elements simply by creating new subclasses.

Implementing the Interface Segregation and Dependency Inversion principles was more challenging. These principles required me to think in terms of abstractions and interfaces. Making the simulator depends on abstractions instead of concrete classes taught me how to reduce coupling and design systems that are easier to maintain and test. Although these ideas were difficult at first, I now see how they contribute to building a cleaner and more flexible architecture.