Domotic Circuit Simulator — Object-Oriented Implementation

Object-Oriented Programming – Semester 2025-II

Juan Diego Arévalo Bareño

November 2025

1. Conceptual Design Updates

After reviewing the conceptual design, some refinements were made to make the Domotic Circuit Simulator ready for implementation in Python:

- The **Circuit class** now includes automatic validation of all component connections before the simulation starts, ensuring logical consistency.
- The **Component class** was adjusted to include unique identifiers and an internal list of connections, allowing for easier management of relationships between devices.
- The **Switch and Light classes** were slightly modified to better reflect inheritance and polymorphism: both now implement their own version of the `simulate()` method according to their specific roles.

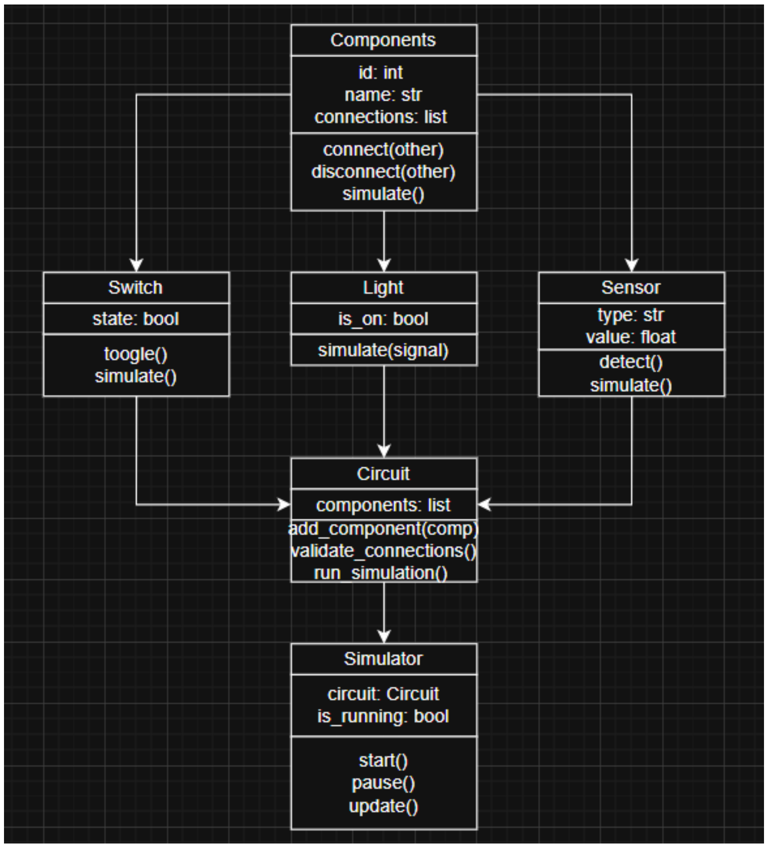
These updates strengthen the original design by adding scalability. The system now better represents real domotic behavior while maintaining clear modularity to the core OOP principles.

2. Technical Design

- **Component (Base Class)**
  Attributes: `id`, `name`, `connections`
  Methods: `connect()`, `disconnect()`, `simulate()`
  Description: Acts as the parent class for all other components, providing shared functionality and structure.
- **Switch (Derived Class)**
  Attributes: `state` (boolean)
  Methods: `toggle()`, `simulate()`
  Description: Inherits from `Component`. It changes its state between ON and OFF and sends signals to connected devices.
- **Light (Derived Class)**
  Attributes: `is_on` (boolean)
  Methods: `simulate(signal)`

Description: Inherits from `Component`. It receives input from switches or sensors and displays visual feedback when activated.

- **Sensor (Derived Class)**
  Attributes: `type`, `value`
  Methods: `detect()`, `simulate()`
  Description: Inherits from `Component`. It detects environmental conditions and triggers connected components.
- **Circuit**
  Attributes: `components` (list of Component objects)
  Methods: `add_component()`, `validate_connections()`, `run_simulation()`
  Description: Manages all components and their interactions within the circuit. It ensures valid connections before running simulations.
- **Simulator**
  Attributes: `circuit`, `is_running`
  Methods: `start()`, `pause()`, `stop()`, `update()`
  Description: Controls the execution of the simulation and updates all components in real time.



The classes Switch, Light, and Sensor all inherit from the base class Component. This allows them to share core attributes while redefining specific behaviors in their own simulate() methods.

3. Implementation Plan for OOP Concepts

Encapsulation: Each class will keep its internal data private and expose only necessary information through public methods. Attributes such as component state, connections, and identifiers will be declared private to prevent direct access from outside.

Example:

```
class Component:
     def_init_(self, name):
          self._name = name
          self.__id = id(self)
          self._connections = []

     def get_name(self):
          return self._name

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

Inheritance: This is for sharing common functionality among all domotic components. A base class Component will define the shared attributes and methods, while specialized classes such as Switch, Light, and Sensor will extend it and include their own behavior.

Example:

```
class Switch(Component):
     def_init_(self, name):
          super()._init_(name)
          self._state = False

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

Polymorphism: This will be applied through the method simulate(). This allows the simulator to iterate through all components generically, calling simulate() without needing to know their specific types.

Example:

```
class Light(Component):
      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'}")
```

Project Directory Structure:

```
DomoticSimulator/
      main.py
      components/
            __init__.py
            component.py
            switch.py
            light.py
            sensor.py
      system/
            __init__.py
            circuit.py
            simulator.py
      utils/
            __init__.py
            helpers.py
```

4. Initial Python Code Snippets

- **Base Class:**

```
class Component:

      def _init_(self, name):
          self._name = name
          self._id = id(self)
          self._connections = []

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

      def disconnect(self, other):
          if other in self._connections:
              self._connections.remove(other)

      def simulate(self):
          """To be overridden by subclasses."""

      pass
```

- **Switch Class:**

```
from component import Component

class Switch(Component):
        def _init_(self, name):
            super()._init_(name)
            self._state = False

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

        def simulate(self):
            print(f"Switch '{self._name}' is {'ON' if self._state
            else 'OFF'}")
            for component in self._connections:
                component.simulate(self._state)
```

- **Light Class:**

```
from component import Component

class Light(Component):
        def _init_(self, name):
```

```
        super()._init_(name)
        self._is_on = False

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

- **Sensor Class:**

```
from component import Component

class Sensor(Component):
    def _init_(self, name, sensor_type):
        super()._init_(name)
        self._type = sensor_type
        self._value = 0.0

    def detect(self, new_value):
        self._value = new_value

    def simulate(self):
        print(f"Sensor '{self._name}' detected value:
    {self._value}")
        for connected in self._connections:
            connected.simulate(self._value > 0)
```

- **Circuit Class:**

```
class Circuit:
    def _init_(self):
        self._components = []

    def add_component(self, component):
        self._components.append(component)

    def validate_connections(self):
        return len(self._components) > 0
```

```
     def run_simulation(self):
           if not self.validate_connections():
           print("Invalid circuit.")
           return

     for component in self._components:
         component.simulate()
```

- **Simulator Class:**

```
class Simulator:
     def _init_(self, circuit):
           self._circuit = circuit
           self._is_running = False

     def start(self):
           self._is_running = True
           self._circuit.run_simulation()

     def pause(self):
           self._is_running = False

     def stop(self):
           self._is_running = False
```

Each Python file corresponds to one of the classes in the model, maintaining the same attributes, methods, and relationships:

- **Base class in the UML:**
  The Component class in the code implements the attributes and the methods exactly as specified in the UML diagram.
- **Inheritance hierarchy:**
  These classes inherit from Component exactly as shown in the UML diagram.
  The snippets can maintain the inheritance chain and redefine simulate() to demonstrate polymorphism, as described in the design rationale.
- **System-level classes:**
  The Circuit class aggregates multiple Component objects, just as shown in the UML

association relationship. The Simulator class depends on Circuit and controls execution flow, consistent with the UML model.