WORKSHOP 3: Applying SOLID Principles to a Domotic Circuit Simulator

Javier Camilo Orduz Acero

Angel Arturo Varela Duque

Carlos Raúl Rojas Vergara

SOLID Principles Analysis:

1. SOLID Principles Analysis
1.1 Single Responsibility Principle (SRP)

Definition:
A class should have only one reason to change, meaning it must focus on a single responsibility.

Relevance:
SRP avoids large “god classes,” reduces complexity, improves readability, and makes debugging easier.

Application in our simulator:

Sensors currently read data and send it to the SecuritySystem. These are two responsibilities.
→ We should separate “sensor measurement” from “communication logic.”

Workspace handles too many tasks (simulation time, components, logs, grid, undo/redo, theme).
→ It should be split into smaller classes (e.g., ComponentManager, SimulationManager).

SecuritySystem checks alarms, sensors, cameras, notifications… a lot.
→ Could be broken into SecurityChecker + NotificationHandler.

1.2 Open/Closed Principle (OCP)

Definition:
Software entities should be open for extension but closed for modification.

Relevance:
Allows adding new features without touching existing code, reducing bugs and increasing maintainability.

Application in our simulator:

New Sensor types (SmokeSensor, MotionSensor, TemperatureSensor) can be added since all inherit from Sensor.

LightsSystem could support new light types (SmartLight, OutdoorLight) without modifying existing code.

SecuritySystem currently uses explicit methods for each device (CheckSecurity(Alarm), CheckSecurity(Camera)…).
→ Better to use an interface ISecurityDevice so new secure devices can be added without modifying SecuritySystem.

1.3 Liskov Substitution Principle (LSP)

Definition:
Subclasses must be completely replaceable by their base class without breaking the system.

Relevance:
Guarantees that inheritance is correct and prevents unexpected behavior from subclasses.

Application in our simulator:

All Sensors subtypes (SmokeSensor, MotionSensor…) should work anywhere a Sensor is expected.
→ They must all implement ReadData() and SendData() correctly.

All future Lights classes must behave consistently with the base Lights class (turnOn/turnOff must not change behavior type).

Future Camera subclasses (NightVisionCamera, MotionCamera) must respect StartRecorder() / StopRecorder() without altering expected behavior.

1.4 Interface Segregation Principle (ISP)

Definition:
Clients should not depend on interfaces they do not use.

Relevance:
Prevents forcing classes to implement methods irrelevant to them and keeps components focused and flexible.

Application in our simulator:

Currently, components like Alarm, Camera, Lights, Sensors all interact with SecuritySystem using different custom methods.
→ Instead of one big interface, we should define multiple small ones:

IActivatable (turnOn/turnOff)

ISensorReader (ReadData)

IRecordable (StartRecorder/StopRecorder)

Workspace should interact with components through small interfaces instead of knowing all details of every device.

1.5 Dependency Inversion Principle (DIP)

Definition:
High-level modules must depend on abstractions, not on concrete classes.

Relevance:
Improves flexibility, reduces coupling, and allows replacing components more easily (e.g., simulation vs real hardware).

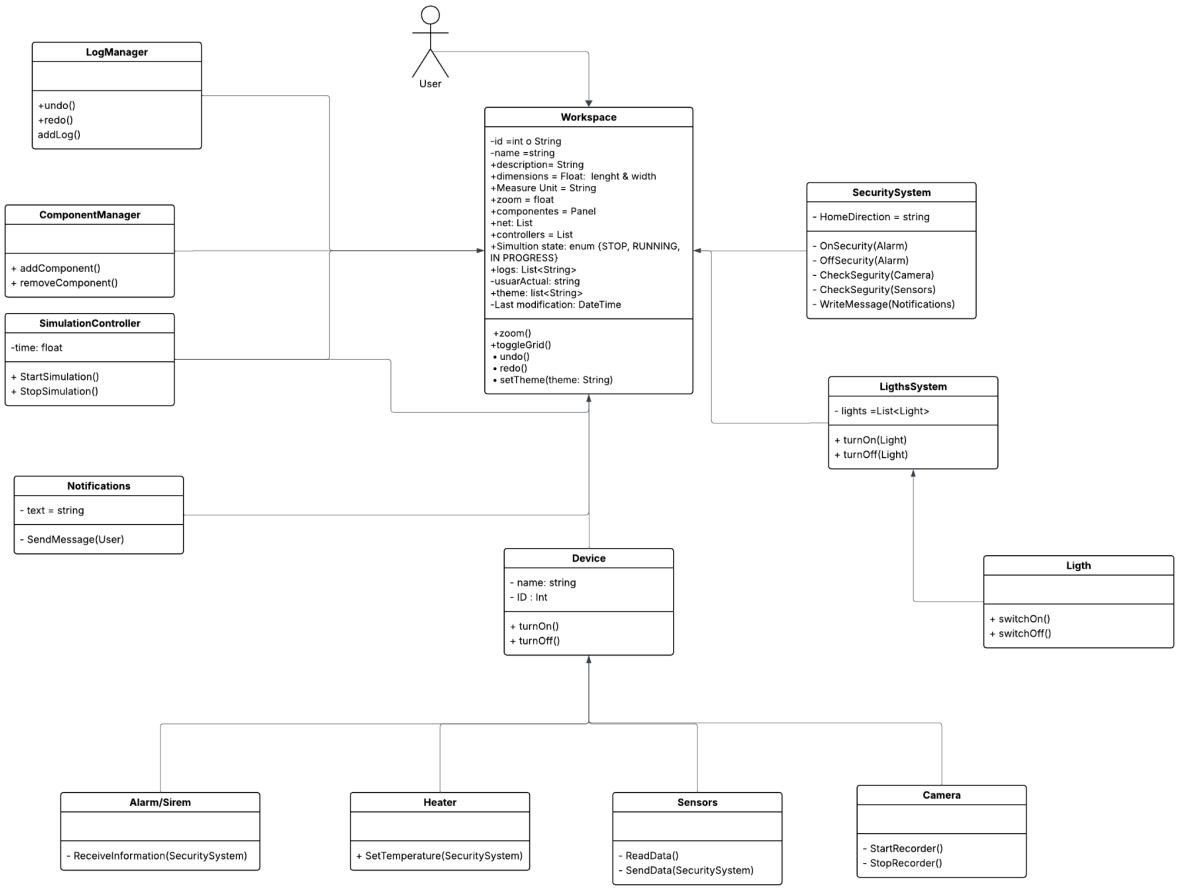
Application in our simulator:

SecuritySystem depends directly on concrete classes (Alarm, Camera, Sensors, Notifications).
→ Should depend on interfaces like ISecurityDevice, INotificationSender.

LightsSystem depends directly on Lights.
→ Should depend on an abstraction like ILight.

Workspace depends on a list of specific components.
→ Should depend on a general IComponent interface for better scalability.

Updated UML and CRC Cards:



CRC CARDS

Class: Workspace	
Responsibilities: Global workspace information (id, name, description, theme, zoom). Coordinate ComponentManager, SimulationController, and LogManager. Manage global actions like theme changes, zoom, and grid toggling.	**Collaborators:** ● ComponentManager ● SimulationController ● LogManager ● User

Class: ComponentManager	
Responsibilities: Manage all system components. Add and remove components. Handle component connections.	**Collaborators:** ● Workspace ● Device

Class: LogManager	
Responsibilities: Control simulation stae. Manage simulation time.	**Collaborators:** ● Workspace ● ComponentManager

Class: SimulationController	
Responsibilities: Control simulation state. Manage simulation time.	**Collaborators:** ● Workspace ● ComponentManager

Class: SimulationController	
Responsibilities: Store and manage system logs. provide undo and redo operations.	**Collaborators:** ● Workspace ● ComponentManager ● SimulationController

Class: Notifications	
Responsibilities: Send message to user. notification text.	**Collaborators:** ● User ● SecuritySystem

Class: SecuritySystem	
Responsibilities: Monitor using alarms, sensors, and cameras. Activate and deactivate. Checks and send notifications.	**Collaborators:** ● Alarm/Siren ● Sensors ● Camera ● Notifications

Class: LightsSystem	
Responsibilities: Control all light devices (on/off) Manage list of lights.	**Collaborators:** ● Light ● Workspace

Class: Device	
Responsibilities: Common attributes for all devices.	**Collaborators:** ● Heater ● Alarm/Siren ● Sensors ● Camera

	• Light • ComponentManager

Class: Alarm/Siren	
Responsibilities: Activate and desactivate the alarm system. Receive data from SecuritySystem.	**Collaborators:** • SecuritySystem • Device

Class: Heater	
Responsibilities: Manage temperature. Temperature changes from Security System when it take grade lower than 10 degrees.	**Collaborators:** • SecuritySystem • Device

Class: Sensors	
Responsibilities: Collect environmental data. Send readings to SecuritySystem.	**Collaborators:** • SecuritySystem • Device

Class: Camera	
Responsibilities: Start and stop video recording and provide security footage.	**Collaborators:** • SecuritySystem • Device

Class: Light	
Responsibilities: Turn light on and off.	**Collaborators:** ● LightsSystem ● Device

Class: User	
Responsibilities: Interact with Workspace.	**Collaborators:** ● Workspace ● Notifications

Python Code Snippets:

capacitors.py

OOP-proyect > sourceCode > capacitors.py > Capacitor

```python
"""Into this module are defined the capacitors of all typeOfCap and their caracteristics."""

from components import Component

# In this class,we define the bassic characteristics of a capacitor.
class Capacitor(Component):
    def __init__(self,name:str, description:str, typeOfCap:str, pieceNumber:str, capacitance:float, voltageRating:float):
        super().__init__(name, description, pieceNumber)
        self.capacitance = capacitance  #in farads
        self.voltageRating = voltageRating  #in volts
        self.energyStored = 0.5 * self.capacitance * (self.voltageRating ** 2)  #in joules
        self.typeOfCap = typeOfCap

# Method that define the characteristics own of each type of capacitor
    def capacitorInfo(self):
        capacitorInfo = self.get_info()
        capacitorInfo.update({
            "Capacitance": self.capacitance,
            "Voltage Rating": self.voltageRating,
            "Energy Stored": self.energyStored,
            "Type of Capacitor": self.typeOfCap
        })
        return capacitorInfo

# Method that checks if the user input values are valid
    def checkUserCapacitorValues(self, capacitance:float, voltageRating:float):
        if capacitance <= 0:
            raise ValueError("Capacitance must be a positive value.")
        if voltageRating <= 0:
            raise ValueError("Voltage Rating must be a positive value.")
        return True

# Method that simulates the breakdown of the capacitor under excessive voltage
    def capacitorBreakdown(self, appliedVoltage:float):
        if appliedVoltage > self.voltageRating:
            breakdown = True  # Capacitor breaks down
            self.capacitance = 0  # Set capacitance to zero to simulate failure
        else:
            breakdown = False
        return breakdown
```

components.py

OOP-proyect > sourceCode > components.py > Component > __init__

```python
"""Super class for designing the components of the model whith the general characteristics."""
# Here is defined the Component class that will be the super class for all the components of the model.
class Component:
    def __init__(self, name:str, description: str, pieceNumber:str ,
                 parameters: dict, dimensions:dict):
        self.name = name
        self.description = description
        self.parameters = parameters
        self.pieceNumber = pieceNumber
        self.dimensions = dimensions

        def get_info(self):
            info = {
                "Name": self.name,
                "Description": self.description,
                "Piece Number": self.pieceNumber,
                "Parameters": self.parameters,
                "Dimensions": self.dimensions
            }
            return info

```

powerSources.py

OOP-proyect > sourceCode > powerSources.py > PowerSource > suppyInfo

```python
"""From the components module, import the Component class.
and using the Component class, create a PowerSource class that inherits from it."""
from components import Component

class PowerSource(Component):
    """Class to model power sources, inheriting from Component."""

    def __init__(self, name: str, description: str, type: str, pieceNumber: str,
                 parameters: dict, dimensions: dict, voltage: float, current: float):
        super().__init__(name, description, type, pieceNumber, parameters, dimensions)
        self.voltage = voltage  #in volts
        self.current = current    #in amperes
        self.power = self.voltage * self.current   #in watts

    def suppyInfo(self):
        power_info = self.get_info()
        power_info.update({
            "Voltage": self.voltage,
            "Current": self.current,
            "Power": self.power,
            })
        return power_info
```

relays.py

OOP-proyect > sourceCode > relays.py > Relay > relayInfo

```python
"""At this module, the relays class is defined, inheriting from the Component class.
The relays are a electromechanical switch that use a coi to create a magnetic
field and open or close the contacts, these switch type are used to control high power"""

from components import Component

class Relay(Component):
    def __init__(self, name: str, description: str, typeOfRelay: str, pieceNumber: str,
                 coilVoltage: float, coilCurrent: float, contactConfiguration: str,
                 maxSwitchingVoltage: float, maxSwitchingCurrent: float):
        super().__init__(name, description, typeOfRelay, pieceNumber)
        self.coilVoltage = coilVoltage  # in volts
        self.coilCurrent = coilCurrent  # in amperes
        self.contactConfiguration = contactConfiguration  # e.g., SPST, DPDT
        self.maxSwitchingVoltage = maxSwitchingVoltage  # in volts
        self.maxSwitchingCurrent = maxSwitchingCurrent  # in amperes

    def relayInfo(self):
        relay_info = self.get_info()
        relay_info.update({
            "Coil Voltage": self.coilVoltage,
            "Coil Current": self.coilCurrent,
            "Contact Configuration": self.contactConfiguration,
            "Max Switching Voltage": self.maxSwitchingVoltage,
            "Max Switching Current": self.maxSwitchingCurrent
        })
        return relay_info
```

switchs.py

OOP-proyect > sourceCode > switchs.py > Switch > checkUserSwitchValues

```python
# In this class,we defin the bassic characteristics of a switch.
class Switch(Component):
    def __init__(self, name: str, description: str, typeOfSw: str, pieceNumber: str, switchtypeOfSw: str,
                 maxVoltage: float, maxCurrent: float, defaultState: str, toggleType: str):
        super().__init__(name, description, typeOfSw, pieceNumber)
        self.switchtypeOfSw = switchtypeOfSw  # e.g., toggle, push button
        self.maxVoltage = maxVoltage  # in volts
        self.maxCurrent = maxCurrent  # in amperes
        self.typeOfSw = typeOfSw  # e.g., mechanical, electronic
        self.defaultState = defaultState
        self.toggleType = toggleType

# Method thad define the characteristics own of each type of switch
    def switchInfo(self):
        switch_info = self.get_info()
        switch_info.update({
            "Switch typeOfSw": self.switchtypeOfSw,
            "Max Voltage": self.maxVoltage,
            "Max Current": self.maxCurrent
        })
        return switch_info

# Method that checks if the user input values are valid
    def checkUserSwitchValues(self, maxVoltage: float, maxCurrent: float):
        if maxVoltage <= 0:
            raise ValueError("Max Voltage must be a positive value.")
        if maxCurrent <= 0:
            raise ValueError("Max Current must be a positive value.")
        return True

# Method that simulates the breakdown of the switch under excessive voltage or current
    def switchBreakdown(self, voltage: float, current: float):
        if voltage > self.maxVoltage or current > self.maxCurrent:
            breakdown = True  # Switch breaks down
            self.defaultState = "broken"  # Set state to broken to simulate failure
        else:
            breakdown = False
        return breakdown
```

resistors.py

OOP-proyect > sourceCode > resistors.py > Resistor

```python
"""Into this module are defined the resistors of al type and their caracteristics."""

from components import Component

# In this class,we defin the bassic characteristics of a resistor.
class Resistor(Component):
    def __init__(self, name: str, description: str, type: str, pieceNumber: str,resistance:float, maxPowerRating:float):
        super().__init__(name, description, type, pieceNumber)
        self.resistance = resistance  #in ohms
        self.maxPowerRating = maxPowerRating  #in watts
# Method thad define the characteristics own of each type of resistor
    def resistorInfo(self):
        resistorInfo = self.get_info()
        resistorInfo.update({
            "Resistance": self.resistance,
            "Max Power Rating": self.maxPowerRating
        })
        return resistorInfo

# Method that checks if the user input values are valid
    def checkUserResistorValues(self, resistance:float, maxPowerRating:float):
        if resistance <= 0:
            raise ValueError("Resistance must be a positive value.")
        if maxPowerRating <= 0:
            raise ValueError("Max Power Rating must be a positive value.")
        return True

# Method that simulates the breakdown of the resistor under excessive voltage or current
    def resistorBreakdown(self, voltage:float, current:float):
        powerDissipated = voltage * current
        if powerDissipated > (self.resistance * current**2):  # P = I^2 * R
            breakdown=True  # Resistor breaks down
            self.resistance=10e100  # Set resistance to a very high value to simulate open circuit
        else:
            breakdown=False
        return breakdown
```

Reflection:

As a group of Electronics engineering students, applying SOLID principles was both challenging and very confusing at the first. At the beginning, most of our project was built in a very straightforward way: one large class that controlled everything and several components function each other at the same time, with many responsibilities in the workspace, etc. When we started applying SOLID, we realized how much re-thinking the structure was required. The most difficult part was understanding how to distribute responsibilities correctly. Separating the Workspace class into several controllers felt confusing at first and so many classes for programming, because it was used to keeping everything in a single place.

Using interfaces was another challenge, since we had to learn when they were necessary and how they help reduce dependencies. At the same time, creating more classes and interfaces made the UML look more complex. However, we understood that this complexity actually represents a cleaner and more scalable architecture.

The biggest thing the SOLID is not just theory because it improves the organization and clarity of a project. After applying the principles, the system became easier to understand, easier to extend to a python program with the components. We now see how professional software design relies on these principles, and I feel more confident in structuring future projects with better practices.