UNIVERSIDAD NACIONAL DE COLOMBIA

FACULTAD DE INGENIERÍA

OBJECT ORIENTED PROGRAMMING

ENGINERING FACULTY

WORKSHOP 3

AUTHORS:

Eddy Johan Tocancipa Muñoz

Valeria Aranda Pacheco

Luis Santiago Vargas Rodríguez

Bogotá DC.2025

1. SOLID Principles — Definitions & Relevance

S — Single Responsibility Principle (SRP): A class should have only one reason to change. In domotics, separate responsibilities — hardware representation (e.g., Bulb), control logic (e.g., SwitchController), persistence/logging — into distinct classes.

O — Open/Closed Principle (OCP): Software entities should be open for extension and closed for modification. Implement base interfaces/abstract classes for devices so new device types (e.g., SmartBulb, Dimmer) can be added without changing existing code.

L — Liskov Substitution Principle (LSP): Subtypes must be substitutable for their base types without altering correctness. Ensure derived device classes respect expected behaviors (e.g., DimmableBulb supports set_brightness(value) but does not break turn_on() semantics).

I — Interface Segregation Principle (ISP): Prefer many client-specific interfaces over one general-purpose interface. For example separate Switchable (turn_on/turn_off) from Dimmable (set_brightness) and Sensor (read_value).

D — Dependency Inversion Principle (DIP): High-level modules should not depend on low-level modules; both should depend on abstractions. Use dependency injection so controller classes depend on interfaces (abstract base classes) rather than concrete device classes.

2. UPDATED UML AND CRC CARDS:

Class: House		
Responsibilities:	**Collaborators:**	**SOLID Application**
It displays the project title and main menu.	Rooms	SPR: Only manages UI navigation and house layout. OCP New rooms or panels can be added without modifying House ISP: Does not force implementation of device logic DIP: depends on abstractions like room objects, not concrete electronics
Load the general house layout (bedroom, living room, kitchen).	Component Panel	
Detect clicks on rooms and zoom into the selected one.	Connection Simulator	
Load the corresponding side panel according to the selected room.		

Class: Kitchen		
Responsibilities:	**Collaborators:**	**SOLID Application**
Graphically represent a kitchen.	Kitchen Electronics	SRP: Focused only on spatial representation. OCP: New furniture or electronics can be added through composition. LSP: Any subtype of Room can replace Kitchen in higher modules.
Serves as a structure for the classes attached to the kitchen behaviour.	Bug	
Contains the furniture of a kitchen.	Temperatures	

Class: Bedroom		
Responsibilities:	**Collaborators:**	**SOLID Application**
Graphically represent a Bedroom.	Bedroom electronics	SRP: Only manages bedroom structure. OCP: Decor changes do not affect other rooms. LSP: Behaves like any other Room subclass.
Serves as a structure for the classes attached to the kitchen behaviour.		
Contains the furniture of a Bedroom..		

Class: Living Room		
Responsibilities:	**Collaborators:**	**SOLID Application**
Graphically represent a Living Room.	Living Room Electronics	SRP: One responsibility—visual/spatial living room. OCP: Can be extended with new furniture. LSP: Interchangeable with other Room types.
Serves as a structure for the classes attached to the living room behaviour.		
Contains the furniture of a Livingroom.		

Class: Kitchen Electronics		
Responsibilities:	**Collaborators:**	**SOLID Application**
Represents sensors	Bulb	SRP: Only handles kitchen device behavior. ISP: Only devices with sensor capabilities implement sensing. OCP: New devices can be added without modifying existing ones.
Store its electrical properties (voltage, polarized)	Heat Sensor	
Respond to events (turn on/off, connection error, alarm).	Movement Sensor	
	Resistance	

Class: Living Room Electronics		
Responsibilities:	**Collaborators:**	**SOLID Application**
Represents Electrodomestics	Resistance	SRP: Only manages living room appliances. OCP: Appliances can be extended. ISP: Only appliances needing audio/display implement those behaviors.
Store its electrical properties (voltage, power)	TV	
Respond to events (turn on/off, connection error, display, audio).	Radio	

Class: Bedroom Electronics		
Responsibilities:	**Collaborators:**	**SOLID Application**
Represent Lights	Resistance	SRP: Controls lighting electronics only. OCP: New light types can be added. ISP: Illumination and intensity are separate behaviors
Store its electrical properties (voltage, Luminosity)	Bulb	
Respond to events (turn on/off, connection error, light).	Desk Lamp	
	Lamp	

Class: Cables		
Responsibilities:	**Collaborators:**	**SOLID Application**
Represent connections between components.	Kitchen, Bedroom, Livingroom Electronics	SRP: Only connection logic. ISP: Separation between devices and wiring. LSP: Any cable subtype behaves like a generic connection.
Verify electrical continuity.	Plug	
It Detects connection errors (short circuits, missing ground, etc.).	Test	
Conducts current	Control Panel	

Class: Test		
Responsibilities:	**Collaborators:**	**SOLID Application**
Analyze user-made connections.	Control Panel, Kitchen, Bedroom, Living Room	SRP: Runs circuit validation only. OCP: New test conditions can extend behavior. DIP: Should depend on interfaces, not concrete devices
Determine if the circuit is correctly assembled.	Kitchen, Bedroom, Livingroom Electronics	
Execute the test and display the results (lights on, sensors activated, errors).	Cable	
Provide visual or sound feedback.	Result Panel	

Class: Result Panel		
Responsibilities:	**Collaborators:**	**SOLID Application**
Display messages about the simulation status.	Connection Simulator	SRP: Output/display only. ISP: Does not depend on electronics directly. OCP: Can add new result types without modifying existing logic.
Indicate whether the components work properly or have malfunctions.	Rooms	
Show specific alerts (Sensor not calibrated correctly, Short circuit in living room).		

Class: Resistance		
Responsibilities	**Collaborators**	**SOLID Application**
Allows to regulate the current	Electronic Elements	SRP: Only resistance logic. LSP: Behaves consistently as an ElectronicComponent. OCP: Can extend with special resistances.
Prevents electronics to burn	Plug	
Gives ohms	Cables	

Class: Bulb		
Responsibilities	**Collaborators**	**SOLID Application**
Generates light	Resistance	SRP: Light emission only. ISP: Illumination is not forced onto other components. LSP: Smart bulbs can extend behavior safely
Gives differents intensities of light	Cables	
Consume Volts		

Class: Lamp		
Responsibilities	**Collaborators**	**SOLID Application**
Can give differents colors of lights	Resistance	OCP: Extends Bulb without altering it. LSP: Maintains Bulb behavior. SRP: Manages only color-based lighting.
Gives differents intensities of lights	Cables	
Consume volts		

Class: Desk Lamp		
Responsibilities	**Collaborators**	**SOLID Application**
Gives differents intensities of lights	Resistance	SRP: Only desk lamp behavior. LSP: Substitutes Lamp safely. OCP: Supports added

Consume Volts	Cables	features without modifying Lamp

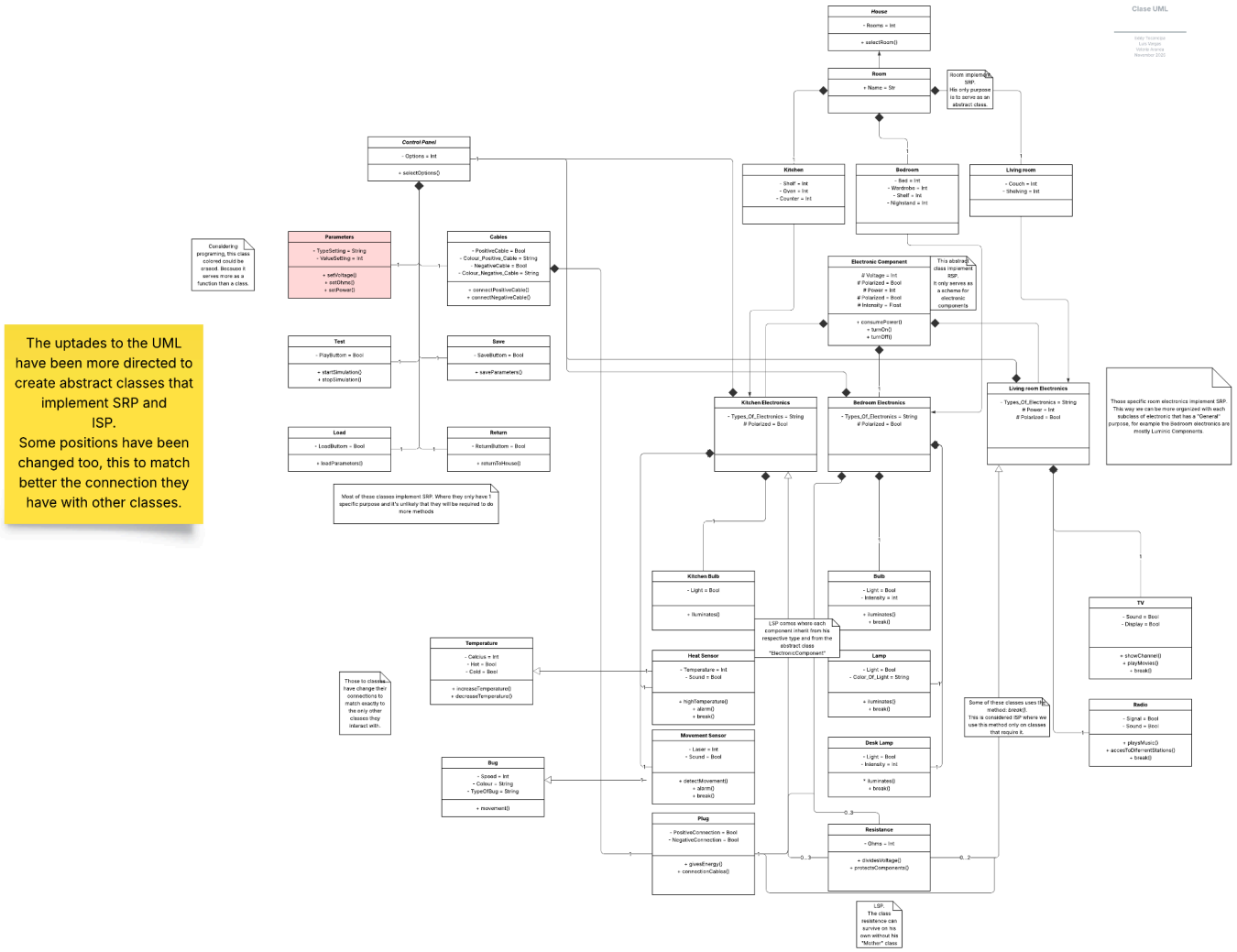
Class: Heat Sensor		
Responsibilities	**Collaborators**	**SOLID Application**
Detects high temperatures	Resistance	ISP: Only HeatSensor implements temperature logic. SRP: Focuses solely on heat detection. LSP: Interchangeable with other sensors.
Can give an alarm when temperatures are high	Cables	
Parameters vary by volts received	Bulb	

Class: Motion Sensor		
Responsibilities	**Collaborators**	**SOLID Application**
Detects Movement	Resistance	ISP: Motion detection isolated from temperature. SRP: Movement detection only. LSP: Substitute for Sensor types.
Can give an alarm when it detects movements.	Cables	
Parameters vary by volts received	Bulb	

Class: TV		
Responsibilities	**Collaborators**	**SOLID Application**
Gives display	Stereo	SRP: Multimedia output only. ISP: Only implements audio/display behavior. OCP: Extendable with new multimedia capabilities.
Gives audio	Resistance	
Consume watts	Cables	

Class: Radio		
Responsibilities	**Collaborators**	**SOLID Application**
Gives audio	Stereo	SRP: Audio-only device. ISP: Does not implement display. OCP: Expandable (e.g., Bluetooth radio).
Plays music	Resistance	
Plays news	Cables	
Plays podcast		
Consume watts		

Class: Plug		
Responsibilities	**Collaborators**	**SOLID Application**
Gives volts	Electronic Components	SRP: Power delivery only. ISP: Separate from cable logic. LSP: Any outlet subtype must preserve its electrical contract.
Give a terminal connection to the cables	Cables	
Gives earth		



The UML has been updated with numerous new notes, positions and classes added.

For better visualization use the link:

[Click me to go see the UML!]

```python
# ============================
# BASIC COMPONENTS AND UTILITIES
# ============================

# Base class with a single responsibility: represent an electronic component
# SRP (Single Responsibility Principle): this class only handles generic electronic component behavior.
class ElectronicComponent:
    def __init__(self, name:str, voltage=None, intensity=None, resistance=None, connected=False):
        self.name = name                    # Component name
        self.voltage = voltage              # Voltage
        self.intensity = intensity          # Current
        self.resistance = resistance        # Resistance
        self.connected = connected          # Connection status

    # Turns the component on
    def turn_on(self):
        self.connected = True
        print(f"{self.name} is ON")

    # Turns the component off
    def turn_off(self):
        self.connected = False
        print(f"{self.name} is OFF")

    # Indicates if it consumes power
    def consumePower(self):
        if self.connected:
            print(f"{self.name} is consuming power")
        else:
            print(f"{self.name} is not consuming power")


# Mixin for components that can illuminate
# ISP (Interface Segregation Principle): illumination is separated into a mixin so only illumination-capable classes use it.
# OCP (Open/Closed): new illumination behaviors can be added through mixins without modifying existing classes.
```

```python
class IlluminationMixin:
    def iluminates(self):
        if self.connected:
            print(f"{self.name} is illuminating")
        else:
            print(f"{self.name} cannot illuminate because it's OFF")


# Mixin for classes that can break
# ISP again: being "breakable" is optional and injected via mixin instead of forcing all components to implement it.
class BreakableMixin:
    def brake(self):
        print(f"{self.name} is broken")


# ====================
# PASSIVE COMPONENTS
# ====================

class Resistance(ElectronicComponent):
    # LSP (Liskov Substitution Principle): this class can replace its parent ElectronicComponent without breaking behavior.
    def __init__(self, resistance):
        super().__init__("Resistance", None, None, resistance, False)
        self.ohms = resistance  # Keeps the original logic

    def dividesVoltage(self):
        print("Voltage has been divided")

    def protectsComponents(self):
        print("Components have been protected")

# ============
# LIGHTING
# ============

# Multiple inheritance combining mixins respects ISP and OCP.
class Bulb(ElectronicComponent, IlluminationMixin, BreakableMixin):
    def __init__(self, intensity=None, light=False):
        super().__init__("Bulb", None, intensity, None, False)
        self.light = light
```

```python
# OCP: Lamp customizes illumination behavior via method overriding,
without altering the parent class.
class Lamp(Bulb):
    def __init__(self, color_of_light:str):
        super().__init__(None, False)
        self.name = "Lamp"
        self.color_of_light = color_of_light

    # Polymorphism: overriding method from mixin to provide specialized
illumination.
    def iluminates(self):
        if self.connected:
            print(f"Lamp is illuminating with {self.color_of_light}
light")
        else:
            print("Lamp cannot illuminate because it's OFF")


class DeskLamp(Lamp, BreakableMixin):
    # LSP and OCP: DeskLamp extends Lamp and modifies behavior without
breaking substitution.
    def __init__(self, intensity=None):
        super().__init__("white")
        self.name = "Desk Lamp"
        self.intensity = intensity

    # Custom illumination behavior
    def iluminates(self):
        print(f"Desk Lamp illuminates with intensity {self.intensity}")


# =======
# ROOM
# =======

# SRP: Room only manages room information, nothing else.
class Room:
    def __init__(self, name:str):
        self.name = name

    def selectRoom(self):
        print(f"You are in the {self.name}")
```

```python
# ==========
# SENSORS
# ==========

class HeatSensor(ElectronicComponent, BreakableMixin):
    # LSP: A sensor behaves like an electronic component.
    # ISP: Break functionality only added where needed.
    def __init__(self, temperature=None):
        super().__init__("HeatSensor")
        self.temperature = temperature
        self.sound = False

    def highTemperature(self):
        if self.temperature:
            print("High temperature detected!")

    def alarm(self):
        self.sound = True
        print("HEAT ALARM ACTIVE")


class MovementSensor(ElectronicComponent, BreakableMixin):
    # Similar SOLID principles as HeatSensor
    def __init__(self, laser=None):
        super().__init__("MovementSensor")
        self.laser = laser
        self.sound = False

    def detectMovement(self):
        print("Movement detected!")

    def alarm(self):
        self.sound = True
        print("MOVEMENT ALARM ACTIVE")


# ====================
# KITCHEN AND ELECTRONICS
# ====================

class KitchenElectronics(ElectronicComponent):
```

```
    #  SRP: represents only electronics in kitchens.
    #  LSP: can be used wherever an ElectronicComponent is expected.
    def __init__(self, type_of_electronics:str):
        super().__init__("Kitchen Electronics")
        self.type_of_electronics = type_of_electronics
        self.polarized = False


class Kitchen(Room):
    #  SRP: handles room structure only.
    def __init__(self, shelf:int, oven:int, counter:int):
        super().__init__("Kitchen")
        self.shelf = shelf
        self.oven = oven
        self.counter = counter


# ======================
# BEDROOM AND ELECTRONICS
# ======================

class BedroomElectronics(ElectronicComponent):
    def __init__(self, type_of_electronics:str):
        super().__init__("Bedroom Electronics")
        self.type_of_electronics = type_of_electronics
        self.polarized = False


class Bedroom(Room):
    def __init__(self, bed:int, wardrobe:int, shelf:int,
nightstand:int):
        super().__init__("Bedroom")
        self.bed = bed
        self.wardrobe = wardrobe
        self.shelf = shelf
        self.nightstand = nightstand


# ======================
# LIVING ROOM AND ELECTRONICS
# ======================

class LivingRoomElectronics(ElectronicComponent):
```

```
    def __init__(self, type_of_electronics:str, power=None):
        super().__init__("Living Room Electronics")
        self.type_of_electronics = type_of_electronics
        self.power = power
        self.polarized = False


class LivingRoom(Room):
    def __init__(self, couch:int, shelving:int):
        super().__init__("Living Room")
        self.couch = couch
        self.shelving = shelving


class TV(LivingRoomElectronics, BreakableMixin):
    #  ISP: TV is breakable but LivingRoomElectronics is not forced to
implement it.
    def __init__(self):
        super().__init__("TV")
        self.display = False
        self.sound = False

    def showChannel(self):
        if self.connected:
            print("Showing channel")
        else:
            print("TV is OFF")

    def playMovies(self):
        print("Playing movie")


class Radio(LivingRoomElectronics, BreakableMixin):
    def __init__(self):
        super().__init__("Radio")
        self.signal = False
        self.sound = False

    def playsMusic(self):
        print("Music is playing")

    def accessToDifferentStations(self):
        print("Changing radio station")
```

```python
# ===========
# WIRES
# ===========

class Wire(ElectronicComponent):
    #  LSP: behaves like any other electronic component.
    def __init__(self, length=None, material=None):
        super().__init__("Wire")
        self.length = length
        self.material = material


class Cables:
    #  SRP: only handles cable behavior.
    def __init__(self, positive_color:str, negative_color:str):
        self.positive = False
        self.negative = False
        self.positive_color = positive_color
        self.negative_color = negative_color

    def connectPositiveCable(self):
        print("Positive cable connected")

    def connectNegativeCable(self):
        print("Negative cable connected")


# ================
# TEMPERATURE
# ================

class Temperature:
    #  SRP: manages only temperature-related logic.
    def __init__(self, celsius:float, hot:bool, cold:bool):
        self.celsius = celsius
        self.hot = hot
        self.cold = cold

    def increaseTemperature(self, celsius):
        if celsius > 20:
            print("It's warm")
```

```
            print("Turning on the air conditioning")

    def decreaseTemperature(self, celsius):
        if celsius < 5:
            print("It's cold")
            print("Turning on the heating")


# =====
# OTHERS
# =====

class Bug:
    # SRP: represents only a bug.
    def __init__(self, speed=None, color:str=None, bugType:str=None):
        self.speed = speed
        self.color = color
        self.type = bugType

    def movement(self):
        print("Bug is moving")


# Gives energy for electric components
class Plug:
    # SRP: only manages plug behavior.
    def __init__(self):
        self.positive_connection = False
        self.negative_connection = False

    def givesEnergy(self):
        print("Energy supplied")

    def connectionCables(self):
        print("Cables connected to plug")


class return_to_house:
    # SRP: handles return action only.
    def __init__(self):
        self.returnButton = False

    def returnToHouse(self):
```

```
        print("Returning to house")
```

4. Reflection on Solid Principles:

Most of the team is not associated with electronic components, only 1 member is in that career. Even so it was interesting seeing how taking concepts like separating one “Mother” class that just do 1 thing, create multiple new subclasses to extend the functionality of the “Mother” class and making them not break the program comes in a practical and intuitive way when we see it with logical sense in how electronic components work

For example, creating a hierarchy:

First we created the class “ElectronicComponent” This will be our “Mother” class and will be used as a template for any other component (Abstract class). Instead of creating inside this “Mother” class all the other components that the program will require, we initialize subclasses that categorize components for the specific room and uses they are designed for, with an important thing to remark that these subclasses achieve the independence that LSP Principle brings.

All of this mentality and line of work doesn’t require a deep understanding of electronic concepts, just coherence and logical thinking.

When it comes to values, we indeed will need to acquire some rules in which electronic components behave. It could become a problem if we do not treat inheritance as it should be, but it’s not known yet.

A remarkable thing to say is that our understanding of how classes work improved in a notorious way. This happened during the process of thinking which classes should be abstracts and make them with the least amount of things that have to do, or how to separate them accordingly so they don’t inherit worthless methods.

We don’t really see trade-offs, guess those will come when we start testing values and understanding the complexity of having a lot of fragmented classes.