Universidad Politécnica Salesiana
Ingeniería en Electrónica y Automatización.

Integrantes Chipugsi Anthony
Oibe Cinthya

Práctica: #3
Tema: Clases en Raspy.
Fecha: 05/11/2025

05-11-25

Práctica #3
Trabajo en Clase Herencia
Fecha: 10/11/2025

Trabajo Robot herencia + Raspy PI
Fecha: 13/11/2025

RECUPERACIÓN
Trabajo API con Telegram (Led ON - Led OFF)
Fecha: 21/11/2025

PRÁCTICA #5

RECUPERACIÓN
Trabajo CLASS ROBOT + API + ARCHIVO + TELEGRAM PRÁCTICA #6
Fecha: 21/11/2025

RECUPERACIÓN
Robot LED + BUTTON + DHT11
Fecha: 21/11/2025

Arquitectura MVC en Raspberry Pi

Práctica 4

Anthony Fabricio Chipugsi Pilicita
achipugsip@est.ups.edu.ec
Cinthya Aracelly Orbe Muñoz
corbem@est.ups.edu.ec

I. Introducción

La implementación de la arquitectura Modelo-Vista-Controlador (MVC) en proyectos con Raspberry Pi permite estructurar el código de manera ordenada y modular, facilitando su comprensión y mantenimiento. El Modelo gestiona la lógica y el estado de los dispositivos (por ejemplo, el encendido y apagado del LED según la señal del botón), la Vista se encarga de mostrar la información o retroalimentación al usuario (como mensajes en la terminal de PuTTY que indican el estado del LED), y el Controlador actúa como intermediario, interpretando las acciones del usuario y coordinando la interacción entre el Modelo y la Vista.[2]

II. Metodología

1. Estructura de las carpetas

```
/mvc_project
    /modelo/dispositivos.py
    /vista/consola.py
    /controlador/main.py
```

2. Código Modelo

```python
import RPi.GPIO as GPIO

GPIO.setmode(GPIO.BCM)

class Led:
    def __init__(self, pin):
        self.pin = pin
        GPIO.setup(self.pin, GPIO.OUT)

    def encender(self):
        GPIO.output(self.pin, GPIO.HIGH)

    def apagar(self):
        GPIO.output(self.pin, GPIO.LOW)


class Boton:
    def __init__(self, pin):
        self.pin = pin
        GPIO.setup(self.pin, GPIO.IN, pull_up_down=GPIO.PUD_DOWN)

    def esta_presionado(self):
        return GPIO.input(self.pin) == GPIO.HIGH
```

Figura 1. Código Modelo agragando la clase Boton.

- Agregamos la clase Boton, configurada como entrada con resistencia pull-down para evitar lecturas flotantes. El método esta_presionado() devuelve True si el botón está presionado.

-Responsabilidad: Representar y manejar los dispositivos físicos (LED y botón). Aquí se definen las clases Led y Boton, con sus métodos (encender, apagar, esta_presionado).

3. Código Vista

```python
def mostrar_estado(mensaje):
    print(f"[ESTADO] {mensaje}")
```

Figura 2. Código Vista.

-La vista no tiene cambios al agregar la clase boton, se encarga de mostar los mensajes en la terminal de PuTTY.

-Responsabilidad: Mostrar información al usuario. La vista es la terminal PuTTY, donde se imprime el estado del sistema, no controla hardware ni toma decisiones, solo recibe mensajes y los presenta.

4. Código Controlador

```python
from modelo.dispositivos import Led, Boton
from vista.consola import mostrar_estado
import time
import RPi.GPIO as GPIO

led = Led(18)       # Pin GPIO para LED
boton = Boton(17)   # Pin GPIO para Botón

try:
    while True:
        if boton.esta_presionado():
            led.encender()
            mostrar_estado("Botón presionado → LED encendido")
        else:
            led.apagar()
            mostrar_estado("Botón liberado → LED apagado")
        time.sleep(0.2)
except KeyboardInterrupt:
    print("Finalizando...")
    GPIO.cleanup()
```

Figura 3. Código Controlador.

-Responsabilidad: Coordinar la interacción entre el modelo y la vista. Aquí se decide la lógica: "si el botón está presionado, enciende el LED y muestra el mensaje".

-En el controlador el LED depende directamente del estado del botón, si el botón está presionado el LED se enciende, caso contrario, se apaga.

-Se añade GPIO.cleanup() al final para liberar los pines correctamente.[1]

II. Resultados

Interacción física:
-El LED se enciende únicamente cuando el botón está presionado. Al soltar el botón, el LED se apaga de inmediato.

Consola (Vista):
-En PuTTY aparecen mensajes como:
[ESTADO] Botón presionado → LED encendido
[ESTADO] Botón liberado → LED apagado

Controlador:
-El programa ejecuta un bucle que evalúa constantemente el estado del botón y actúa sobre el LED.
-Se mantiene un pequeño retardo time.sleep(0.2) para evitar rebotes y sobrecarga.

Modelo:
-Se logra encapsular el hardware de clases Led, Boton; lo que permite reutilizar y extender fácilmente el código si se agregan más dispositivos.

Vista:
-La salida en consola está separada del control del hardware, lo que facilita cambiar la interfaz.

Controlador
-La lógica de decisión queda centralizada y clara: si el botón está precionado, enceinde el LED y muestra el estado.

III. Discusión

La aplicación de la arquitectura Modelo-Vista-Controlador (MVC) en proyectos de electrónica con Raspberry Pi demuestra cómo un patrón de diseño, comúnmente utilizado en el desarrollo de software, puede trasladarse al ámbito del control de hardware. En este caso, la separación de responsabilidades permite que cada componente cumpla un rol específico: el Modelo abstrae el hardware y facilita su manipulación mediante clases y métodos; la Vista se encarga de comunicar al usuario el estado del sistema a través de mensajes en consola; y el Controlador coordina la lógica de interacción entre ambos, garantizando que el LED responda únicamente cuando el botón está presionado.[2]

Este enfoque modular ofrece ventajas significativas. Por un lado, mejora la claridad y mantenibilidad del código, ya que las funciones de control, visualización y hardware no se mezclan en un único archivo. Por otro, facilita la escalabilidad, permitiendo incorporar nuevos dispositivos o cambiar la interfaz de salida sin alterar la lógica principal. Además, la práctica refuerza la importancia de aplicar buenas prácticas de programación incluso en proyectos pequeños, mostrando que la organización del código es tan relevante como el funcionamiento físico del circuito.[2]

V. Conclusiones

- La aplicación del patrón Modelo-Vista-Controlador permitió dividir el proyecto en capas independientes, logrando que el hardware, la lógica de control y la interfaz de usuario se mantengan organizados y fáciles de comprender.
- Se comprobó que el LED responde únicamente cuando el botón está presionado, mostrando en la consola el estado correspondiente. Esto evidencia que la interacción entre Modelo, Vista y Controlador se ejecuta de manera coherente y efectiva.
- La estructura modular facilita la incorporación de nuevos dispositivos o la modificación de la interfaz sin alterar la lógica principal, lo que convierte al proyecto en una base sólida para prácticas más complejas.

VI. Referencias

[1] Raspberry Pi Foundation, "GPIO and RPi.GPIO library," Raspberry Pi Docs, 2025. [Online]. Available: https://www.raspberrypi.com/documentation/computers/os.html#gpio

[2] CódigoFacilito, "MVC: Model View Controller explicado," CódigoFacilito, Online. Available: https://codigofacilito.com/articulos/mvc-model-view-controller-explicado. [Accessed: Nov. 24, 2025].