# Ejemplo de Patrones de Diseño: Factory y Observer

Angel Mauricio Rojas Rodríguez

## Introducción

En esta práctica veremos cómo aplicar **dos patrones de diseño importantes** en un caso de simulación de la vida real:

- **Factory (Fábrica):** se encarga de crear objetos sin que el cliente conozca las clases concretas.

- **Observer (Observador):** permite que un objeto *Sujeto* notifique a varios *Observadores* cuando ocurre un evento.

Aplicaremos estos patrones en una **simulación de un hogar inteligente (Smart Home)** con luces, termostatos y cámaras.

---

## Explicación Teórica

### Patrón Factory

El patrón **Factory** centraliza la creación de objetos, ocultando la lógica de construcción.
Ventajas: - El cliente no depende de clases concretas. - Facilita añadir nuevos tipos de objetos sin modificar el código existente.

En este ejemplo, la `DeviceFactory` creará dispositivos como `Light`, `Thermostat` o `Camera`.

## Patrón Observer

El patrón **Observer** define una relación uno a muchos:
- Un **Sujeto** (Subject) mantiene una lista de observadores.
- Cada vez que el sujeto cambia, **notifica a los observadores**.

En el ejemplo:
- Los **dispositivos inteligentes** (luces, termostato, cámara) son sujetos.
- Los **observadores** (`Logger`, `Dashboard`, `MobileApp`) reaccionan a los eventos.

---

# Código en Python

from abc import ABC, abstractmethod from datetime import datetime import time

———————————

# Observer (Interfaz)

———————————

class Observer(ABC): @abstractmethod def update(self, subject, event_type, data): """Método que será llamado por el subject cuando ocurra un evento.""" pass

---

## Subject (base para dispositivos)

---

class Subject: def **init**(self): self._observers = []

```
def attach(self, observer: Observer):
    """Agregar un observador."""
    if observer not in self._observers:
        self._observers.append(observer)


def detach(self, observer: Observer):
    """Quitar un observador."""
    if observer in self._observers:
        self._observers.remove(observer)


def _notify(self, event_type: str, data: dict):
    """Notifica a todos los observadores sobre un evento."""
    for obs in list(self._observers):
        obs.update(self, event_type, data)
```

---

## Observadores concretos

---

class LoggerObserver(Observer): def update(self, subject, event_type, data): timestamp = datetime.now().isoformat(timespec='seconds') print(f"[{timestamp}] [LOG] {subject.name} -> {event_type} | {data}")

class DashboardObserver(Observer): def update(self, subject, event_type, data): print(f"[DASHBOARD] {subject.name} actualizó: {event_type} ({data})")

class MobileAppObserver(Observer): def update(self, subject, event_type, data): if event_type == "alert": print(f"[MOBILE PUSH] ALERTA desde {subject.name}: {data.get('message')}") else: print(f"[MOBILE] Notificación: {subject.name} -> {event_type}")

---

## Dispositivos (Subjects concretos)

---

class Device(Subject): def **init**(self, device_id: str, name: str): super().__init__() self.device_id = device_id self.name = name

class Light(Device): def **init**(self, device_id, name, is_on=False): super().__init__(device_id, name) self.is_on = is_on

```
def switch(self, on: bool):
    prev = self.is_on
    self.is_on = on
    event = "turned_on" if on else "turned_off"
    if prev != self.is_on:
        self._notify(event, {"state": self.is_on})
```

class Thermostat(Device): def **init**(self, device_id, name, temperature=22.0): super().__init__(device_id, name) self.temperature = temperature

```
def set_temperature(self, new_temp: float):
    prev = self.temperature
    self.temperature = new_temp
    self._notify("temperature_changed", {"from": prev, "to": new_temp})
```

class Camera(Device): def **init**(self, device_id, name, recording=False): super().__init__(device_id, name) self.recording = recording

```
def detect_motion(self):
    self._notify("alert", {"message": "Movimiento detectado", "severity
": "high"})
    self.recording = True
    self._notify("recording_started", {"recording": True})
```

---

## Factory (creador de dispositivos)

---

class DeviceFactory: @staticmethod def create_device(device_type: str, device_id: str, name: str, **kwargs) -> Device: dtype = device_type.lower() if dtype == "light": return Light(device_id, name, is_on=kwargs.get("is_on", False)) elif dtype == "thermostat": return Thermostat(device_id, name, temperature=kwargs.get("temperature", 22.0)) elif dtype == "camera": return Camera(device_id, name, recording=kwargs.get("recording", False)) else: raise ValueError(f"Tipo de dispositivo desconocido: {device_type}")

---

## Simulación de caso real

---

*def simulate_smart_home(): logger = LoggerObserver() dashboard = DashboardObserver() mobile = MobileAppObserver()*

```
light1 = DeviceFactory.create_device("light", "L1", "Sala - Luz Princip
al", is_on=False)
thermo = DeviceFactory.create_device("thermostat", "T1", "Termostato -
Sala", temperature=21.5)
cam = DeviceFactory.create_device("camera", "C1", "Cámara - Entrada", r
ecording=False)


for device in (light1, thermo, cam):
    device.attach(logger)
    device.attach(dashboard)
    device.attach(mobile)


print("\n--- Inicio de simulación Smart Home ---\n")


print("Acción: Encender la luz de la sala")
light1.switch(True)
time.sleep(0.5)


print("\nAcción: Ajustar termostato a 24.0°C")
thermo.set_temperature(24.0)
time.sleep(0.5)


print("\nAcción: Cámara detecta movimiento")
cam.detect_motion()
time.sleep(0.5)


print("\nAcción: Apagar la luz de la sala")
light1.switch(False)
time.sleep(0.5)
```

```
print("\n--- Fin de simulación ---\n")
```

*simulate_smart_home()*