Guía Práctica de Patrones de Diseño y Arquitectura (Java & Python)

Esta guía incluye patrones que puedes aplicar sin problema en proyectos Java y Python, tanto en backend como en sistemas distribuidos. Se centra en patrones útiles para arquitectos, desarrolladores senior y diseñadores de software.

📦 PATRONES CREACIONALES

1. Singleton

- **Propósito:** Asegura una única instancia global.
- **Cuándo usar:** Recursos compartidos (p. ej. Logger, Config).
- **Ventajas:** Control total de instancia.
- **Desventajas:** Difícil testeo, acoplamiento global.
- **Ejemplo real:** Logger de sistema o base de datos compartida.

```
public class Singleton {
    private static Singleton instancia;
    private Singleton() {}

    public static Singleton getInstancia() {
        if (instancia == null) {
            instancia = new Singleton();
        }
        return instancia;
    }
}
```

```
class Singleton:
    _instancia = None

    def __new__(cls):
        if cls._instancia is None:
            cls._instancia = super(Singleton, cls).__new__(cls)
        return cls._instancia
```

2. Factory Method

- **Propósito:** Delegar la creación de objetos a subclases.
- **Cuándo usar:** Cuando no sabes de antemano qué clase instanciar.
- **Ventajas:** Mayor flexibilidad que `new` directo.
- **Desventajas:** Aumenta complejidad.
- **Ejemplo real:** Creación de controladores HTTP en frameworks.

```
interface Animal {
    void hablar();
}
```

```
class Perro implements Animal {
    public void hablar() {
        System.out.println("Guau");
    }
}

class Gato implements Animal {
    public void hablar() {
        System.out.println("Miau");
    }
}

class AnimalFactory {
    public static Animal crearAnimal(String tipo) {
        switch(tipo) {
            case "perro": return new Perro();
            case "gato": return new Gato();
            default: throw new IllegalArgumentException("Tipo desconocido");
        }
    }
}
```

```
class Animal:
    def hablar(self):
        pass

class Perro(Animal):
    def hablar(self):
        print("Guau")

class Gato(Animal):
    def hablar(self):
        print("Miau")

class AnimalFactory:
    @staticmethod
    def crear_animal(tipo):
        if tipo == "perro": return Perro()
        elif tipo == "gato": return Gato()
        else: raise ValueError("Tipo desconocido")
```

3. Abstract Factory

- **Propósito:** Crear familias de objetos relacionados.
- **Cuándo usar:** Para múltiples variantes de componentes.
- **Ventajas:** Aislamiento de familias de objetos.
- **Desventajas:** Complejidad si no hay muchas variantes.
- **Ejemplo real:** UI toolkit para distintos SO.

```
// Productos
interface Button {
    void paint();
```

```
}

class WindowsButton implements Button {
    public void paint() {
        System.out.println("Rendering Windows button");
    }
}

class MacButton implements Button {
    public void paint() {
        System.out.println("Rendering Mac button");
    }
}

// Fábrica Abstracta
interface GUIFactory {
    Button createButton();
}

// Fábricas Concretas
class WindowsFactory implements GUIFactory {
    public Button createButton() {
        return new WindowsButton();
    }
}

class MacFactory implements GUIFactory {
    public Button createButton() {
        return new MacButton();
    }
}

// Cliente
public class Application {
    private Button button;

    public Application(GUIFactory factory) {
        button = factory.createButton();
    }

    public void render() {
        button.paint();
    }

    public static void main(String[] args) {
        GUIFactory factory = new WindowsFactory(); // o new MacFactory()
        Application app = new Application(factory);
        app.render();
    }
}
```

```
# Productos
class Button:
    def paint(self):
        raise NotImplementedError()
```

```
class WindowsButton(Button):
    def paint(self):
        print("Rendering Windows button")

class MacButton(Button):
    def paint(self):
        print("Rendering Mac button")

# Fábrica Abstracta
class GUIFactory:
    def create_button(self):
        raise NotImplementedError()

# Fábricas Concretas
class WindowsFactory(GUIFactory):
    def create_button(self):
        return WindowsButton()

class MacFactory(GUIFactory):
    def create_button(self):
        return MacButton()

# Cliente
class Application:
    def __init__(self, factory):
        self.button = factory.create_button()

    def render(self):
        self.button.paint()

# Uso
factory = WindowsFactory()  # o MacFactory()
app = Application(factory)
app.render()
```

4. Builder

- **Propósito:** Separar construcción de un objeto complejo.
- **Cuándo usar:** Objetos con muchos parámetros opcionales.
- **Ventajas:** Código limpio, inmutable.
- **Desventajas:** Requiere más clases.
- **Ejemplo real:** Configuradores de conexión a BD o APIs.

```
class Usuario {
    private String nombre;
    private int edad;

    public static class Builder {
        private String nombre;
        private int edad;

        public Builder nombre(String nombre) {
            this.nombre = nombre;
            return this;
        }
```

```
        public Builder edad(int edad) {
            this.edad = edad;
            return this;
        }

        public Usuario build() {
            return new Usuario(this);
        }
    }

    private Usuario(Builder builder) {
        this.nombre = builder.nombre;
        this.edad = builder.edad;
    }
}
```

```
class Usuario:
    def __init__(self, nombre=None, edad=None):
        self.nombre = nombre
        self.edad = edad

class UsuarioBuilder:
    def __init__(self):
        self._usuario = Usuario()

    def nombre(self, nombre):
        self._usuario.nombre = nombre
        return self

    def edad(self, edad):
        self._usuario.edad = edad
        return self

    def build(self):
        return self._usuario
```

5. Prototype

- **Propósito:** Clonar objetos existentes.
- **Cuándo usar:** Cuando copiar es más eficiente que crear.
- **Ventajas:** Performance.
- **Desventajas:** Problemas con objetos complejos anidados.
- **Ejemplo real:** Plantillas de configuración o documentos.

```
class Documento implements Cloneable {
    public String texto;

    public Documento clone() {
        try {
            return (Documento) super.clone();
        } catch (CloneNotSupportedException e) {
            return null;
```

```
        }
    }
}
```

```
import copy

class Documento:
    def __init__(self, texto):
        self.texto = texto

    def clone(self):
        return copy.deepcopy(self)
```

🏗️ PATRONES ESTRUCTURALES

6. Adapter

- **Propósito:** Convertir la interfaz de una clase en otra.
- **Cuándo usar:** Integración con sistemas externos.
- **Ventajas:** Reutilización.
- **Desventajas:** Puede ocultar problemas de incompatibilidad.
- **Ejemplo real:** Adaptadores de drivers, APIs externas.

```
interface OldSystem {
    void processRequest();
}

class OldSystemImpl implements OldSystem {
    @Override
    public void processRequest() {
        System.out.println("Processing request...");
    }
}

interface NewSystem {
    void executeRequest();
}

class Adapter implements NewSystem {
    private OldSystem oldSystem;

    public Adapter(OldSystem oldSystem) {
        this.oldSystem = oldSystem;
    }

    @Override
    public void executeRequest() {
        oldSystem.processRequest();
    }
}

public class Main {
    public static void main(String[] args) {
```

```
        OldSystem oldSystem = new OldSystemImpl();
        NewSystem newSystem = new Adapter(oldSystem);
        newSystem.executeRequest();
    }
}
```

```
class OldSystem:
    def process_request(self):
        print("Processing request...")

class NewSystem:
    def execute_request(self):
        pass

class Adapter(NewSystem):
    def __init__(self, old_system):
        self.old_system = old_system

    def execute_request(self):
        self.old_system.process_request()

# Uso
old_system = OldSystem()
adapter = Adapter(old_system)
adapter.execute_request()
```

7. Bridge

- **Propósito:** Separar una abstracción de su implementación.
- **Cuándo usar:** Evitar herencia múltiple.
- **Ventajas:** Escalabilidad.
- **Desventajas:** Mayor número de clases.
- **Ejemplo real:** Sistemas de renderizado de UI o de medios.

```
interface Renderer {
    void renderCircle(float radius);
}

class VectorRenderer implements Renderer {
    @Override
    public void renderCircle(float radius) {
        System.out.println("Drawing circle with radius: " + radius + " using vector renderer.");
    }
}

class RasterRenderer implements Renderer {
    @Override
    public void renderCircle(float radius) {
        System.out.println("Drawing circle with radius: " + radius + " using raster renderer.");
    }
}

abstract class Shape {
```

```
    protected Renderer renderer;

    public Shape(Renderer renderer) {
        this.renderer = renderer;
    }

    abstract void draw();
}

class Circle extends Shape {
    private float radius;

    public Circle(float radius, Renderer renderer) {
        super(renderer);
        this.radius = radius;
    }

    @Override
    void draw() {
        renderer.renderCircle(radius);
    }
}

public class Main {
    public static void main(String[] args) {
        Renderer vectorRenderer = new VectorRenderer();
        Shape circle = new Circle(5, vectorRenderer);
        circle.draw();

        Renderer rasterRenderer = new RasterRenderer();
        circle = new Circle(10, rasterRenderer);
        circle.draw();
    }
}
```

```
class Renderer:
    def render_circle(self, radius):
        pass

class VectorRenderer(Renderer):
    def render_circle(self, radius):
        print(f"Drawing circle with radius: {radius} using vector renderer.")

class RasterRenderer(Renderer):
    def render_circle(self, radius):
        print(f"Drawing circle with radius: {radius} using raster renderer.")

class Shape:
    def __init__(self, renderer):
        self.renderer = renderer

    def draw(self):
        pass

class Circle(Shape):
    def __init__(self, radius, renderer):
```

```
        super().__init__(renderer)
        self.radius = radius

    def draw(self):
        self.renderer.render_circle(self.radius)

# Uso
vector_renderer = VectorRenderer()
circle = Circle(5, vector_renderer)
circle.draw()

raster_renderer = RasterRenderer()
circle = Circle(10, raster_renderer)
circle.draw()
```

8. Composite

- **Propósito:** Tratar objetos individuales y grupos de la misma manera.
- **Cuándo usar:** Árboles jerárquicos.
- **Ventajas:** Simplicidad de uso.
- **Desventajas:** Complejidad al modificar la jerarquía.
- **Ejemplo real:** DOM, menús, estructuras de archivos.

```
interface Component {
    void display();
}

class Leaf implements Component {
    private String name;

    public Leaf(String name) {
        this.name = name;
    }

    @Override
    public void display() {
        System.out.println(name);
    }
}

class Composite implements Component {
    private List<Component> children = new ArrayList<>();

    public void add(Component component) {
        children.add(component);
    }

    @Override
    public void display() {
        for (Component component : children) {
            component.display();
        }
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Composite root = new Composite();
        Leaf leaf1 = new Leaf("Leaf 1");
        Leaf leaf2 = new Leaf("Leaf 2");

        root.add(leaf1);
        root.add(leaf2);
        root.display();
    }
}
```

```
class Component:
    def display(self):
        pass

class Leaf(Component):
    def __init__(self, name):
        self.name = name

    def display(self):
        print(self.name)

class Composite(Component):
    def __init__(self):
        self.children = []

    def add(self, component):
        self.children.append(component)

    def display(self):
        for child in self.children:
            child.display()

# Uso
root = Composite()
leaf1 = Leaf("Leaf 1")
leaf2 = Leaf("Leaf 2")

root.add(leaf1)
root.add(leaf2)
root.display()
```

9. Decorator

- **Propósito:** Añadir funcionalidades a objetos dinámicamente.
- **Cuándo usar:** Cuando necesitas extensibilidad.
- **Ventajas:** No rompe la clase base.
- **Desventajas:** Puede haber demasiadas capas.
- **Ejemplo real:** Streams en Java, middlewares en FastAPI.

```
interface Coffee {
    double cost();
```

```
}

class SimpleCoffee implements Coffee {
    @Override
    public double cost() {
        return 5;
    }
}

abstract class CoffeeDecorator implements Coffee {
    protected Coffee decoratedCoffee;

    public CoffeeDecorator(Coffee coffee) {
        this.decoratedCoffee = coffee;
    }

    public double cost() {
        return decoratedCoffee.cost();
    }
}

class MilkDecorator extends CoffeeDecorator {
    public MilkDecorator(Coffee coffee) {
        super(coffee);
    }

    @Override
    public double cost() {
        return decoratedCoffee.cost() + 1;
    }
}

class SugarDecorator extends CoffeeDecorator {
    public SugarDecorator(Coffee coffee) {
        super(coffee);
    }

    @Override
    public double cost() {
        return decoratedCoffee.cost() + 0.5;
    }
}

public class Main {
    public static void main(String[] args) {
        Coffee coffee = new SimpleCoffee();
        System.out.println("Cost: " + coffee.cost());

        coffee = new MilkDecorator(coffee);
        System.out.println("Cost with milk: " + coffee.cost());

        coffee = new SugarDecorator(coffee);
        System.out.println("Cost with milk and sugar: " + coffee.cost());
    }
}
```

```
class Coffee:
    def cost(self):
        pass

class SimpleCoffee(Coffee):
    def cost(self):
        return 5

class CoffeeDecorator(Coffee):
    def __init__(self, coffee):
        self.decorated_coffee = coffee

    def cost(self):
        return self.decorated_coffee.cost()

class MilkDecorator(CoffeeDecorator):
    def cost(self):
        return self.decorated_coffee.cost() + 1

class SugarDecorator(CoffeeDecorator):
    def cost(self):
        return self.decorated_coffee.cost() + 0.5

# Uso
coffee = SimpleCoffee()
print("Cost:", coffee.cost())

coffee = MilkDecorator(coffee)
print("Cost with milk:", coffee.cost())

coffee = SugarDecorator(coffee)
print("Cost with milk and sugar:", coffee.cost())
```

10. Facade

- **Propósito:** Proporcionar una interfaz simplificada.
- **Cuándo usar:** Simplificar subsistemas complejos.
- **Ventajas:** Código más limpio.
- **Desventajas:** Puede ocultar detalles importantes.
- **Ejemplo real:** Librerías de clientes SDK.

```
class Engine {
    public void start() {
        System.out.println("Engine starting...");
    }
}

class AirConditioner {
    public void turnOn() {
        System.out.println("Air conditioner turned on...");
    }
}

class CarFacade {
```

```
    private Engine engine;
    private AirConditioner airConditioner;

    public CarFacade() {
        engine = new Engine();
        airConditioner = new AirConditioner();
    }

    public void startCar() {
        engine.start();
        airConditioner.turnOn();
        System.out.println("Car started!");
    }
}

public class Main {
    public static void main(String[] args) {
        CarFacade car = new CarFacade();
        car.startCar();
    }
}
```

```
class Engine:
    def start(self):
        print("Engine starting...")

class AirConditioner:
    def turn_on(self):
        print("Air conditioner turned on...")

class CarFacade:
    def __init__(self):
        self.engine = Engine()
        self.air_conditioner = AirConditioner()

    def start_car(self):
        self.engine.start()
        self.air_conditioner.turn_on()
        print("Car started!")

# Uso
car = CarFacade()
car.start_car()
```

11. Proxy

- **Propósito:** Controlar el acceso a un objeto.
- **Cuándo usar:** Seguridad, lazy loading, logs.
- **Ventajas:** Control.
- **Desventajas:** Overhead.
- **Ejemplo real:** ORM, cachés, protección de recursos.

```
interface Image {
    void display();
}

class RealImage implements Image {
    private String filename;

    public RealImage(String filename) {
        this.filename = filename;
        loadFromDisk();
    }

    private void loadFromDisk() {
        System.out.println("Loading image: " + filename);
    }

    @Override
    public void display() {
        System.out.println("Displaying image: " + filename);
    }
}

class ProxyImage implements Image {
    private RealImage realImage;
    private String filename;

    public ProxyImage(String filename) {
        this.filename = filename;
    }

    @Override
    public void display() {
        if (realImage == null) {
            realImage = new RealImage(filename);
        }
        realImage.display();
    }
}

public class Main {
    public static void main(String[] args) {
        Image image1 = new ProxyImage("image1.jpg");
        Image image2 = new ProxyImage("image2.jpg");

        // La imagen se carga solo cuando se necesita
        image1.display();  // La imagen se carga y muestra
        image1.display();  // Ya está cargada, solo muestra
        image2.display();  // La imagen se carga y muestra
    }
}
```

```
from abc import ABC, abstractmethod

class Image(ABC):
    @abstractmethod
```

```
    def display(self):
        pass

class RealImage(Image):
    def __init__(self, filename):
        self.filename = filename
        self.load_from_disk()

    def load_from_disk(self):
        print(f"Loading image: {self.filename}")

    def display(self):
        print(f"Displaying image: {self.filename}")

class ProxyImage(Image):
    def __init__(self, filename):
        self.filename = filename
        self.real_image = None

    def display(self):
        if self.real_image is None:
            self.real_image = RealImage(self.filename)
        self.real_image.display()

# Uso
image1 = ProxyImage("image1.jpg")
image2 = ProxyImage("image2.jpg")

# La imagen se carga solo cuando se necesita
image1.display()  # La imagen se carga y muestra
image1.display()  # Ya está cargada, solo muestra
image2.display()  # La imagen se carga y muestra
```

🔁 PATRONES DE COMPORTAMIENTO

12. Observer

- **Propósito:** Notificar cambios a múltiples objetos.
- **Cuándo usar:** Eventos y subscripciones.
- **Ventajas:** Desacoplamiento.
- **Desventajas:** Debug complicado.
- **Ejemplo real:** Eventos GUI, listeners de servicios.

```
class Subject {
    private List<Observer> observers = new ArrayList<>();
    public void attach(Observer o) { observers.add(o); }
    public void notifyObservers() {
        for (Observer o : observers) o.update();
    }
}
```

```
class Subject:
    def __init__(self):
```

```
        self._observers = []
    def attach(self, observer):
        self._observers.append(observer)
    def notify(self):
        for o in self._observers:
            o.update()
```

13. Strategy

- **Propósito:** Cambiar comportamiento en tiempo de ejecución.
- **Cuándo usar:** Múltiples algoritmos para un problema.
- **Ventajas:** Flexible.
- **Desventajas:** Multiplicación de clases.
- **Ejemplo real:** Métodos de pago, políticas de validación.

```
interface PaymentStrategy {
    void pay(int amount);
}

class CreditCardPayment implements PaymentStrategy {
    public void pay(int amount) {
        System.out.println("Paid " + amount + " using Credit Card");
    }
}
```

```
class PaymentStrategy:
    def pay(self, amount): pass

class CreditCardPayment(PaymentStrategy):
    def pay(self, amount):
        print(f"Paid {amount} using Credit Card")
```

14. Template Method

- **Propósito:** Definir esqueleto de algoritmo y permitir redefinición parcial.
- **Cuándo usar:** Flujo común con variantes.
- **Ventajas:** Reutilización.
- **Desventajas:** Acoplamiento con la clase base.
- **Ejemplo real:** Frameworks de pruebas, pipelines.

```
abstract class Game {
    public void play() {
        initialize(); startPlay(); endPlay();
    }
    abstract void initialize();
    abstract void startPlay();
    abstract void endPlay();
}
```

```
class Game:
    def play(self):
        self.initialize()
        self.start_play()
        self.end_play()
    def initialize(self): pass
    def start_play(self): pass
    def end_play(self): pass
```

15. Command

- **Propósito:** Encapsular una petición como objeto.
- **Cuándo usar:** Undo, cola de tareas.
- **Ventajas:** Historia de acciones.
- **Desventajas:** Boilerplate.
- **Ejemplo real:** Macros, ejecución remota, CLI.

```
interface Command {
    void execute();
}

class LightOnCommand implements Command {
    private Light light;

    public LightOnCommand(Light light) {
        this.light = light;
    }

    @Override
    public void execute() {
        light.turnOn();
    }
}

class LightOffCommand implements Command {
    private Light light;

    public LightOffCommand(Light light) {
        this.light = light;
    }

    @Override
    public void execute() {
        light.turnOff();
    }
}

class Light {
    public void turnOn() {
        System.out.println("Light is ON");
    }

    public void turnOff() {
        System.out.println("Light is OFF");
```

```
    }
}

class RemoteControl {
    private Command command;

    public void setCommand(Command command) {
        this.command = command;
    }

    public void pressButton() {
        command.execute();
    }
}

public class Main {
    public static void main(String[] args) {
        Light light = new Light();
        Command lightOn = new LightOnCommand(light);
        Command lightOff = new LightOffCommand(light);

        RemoteControl remote = new RemoteControl();

        remote.setCommand(lightOn);
        remote.pressButton();  // Enciende la luz

        remote.setCommand(lightOff);
        remote.pressButton();  // Apaga la luz
    }
}
```

```
from abc import ABC, abstractmethod

class Command(ABC):
    @abstractmethod
    def execute(self):
        pass

class LightOnCommand(Command):
    def __init__(self, light):
        self.light = light

    def execute(self):
        self.light.turn_on()

class LightOffCommand(Command):
    def __init__(self, light):
        self.light = light

    def execute(self):
        self.light.turn_off()

class Light:
    def turn_on(self):
        print("Light is ON")
```

```
    def turn_off(self):
        print("Light is OFF")

class RemoteControl:
    def set_command(self, command):
        self.command = command

    def press_button(self):
        self.command.execute()

# Uso
light = Light()
light_on = LightOnCommand(light)
light_off = LightOffCommand(light)

remote = RemoteControl()

remote.set_command(light_on)
remote.press_button()  # Enciende la luz

remote.set_command(light_off)
remote.press_button()  # Apaga la luz
```

16. State

- **Propósito:** Cambiar comportamiento según estado.
- **Cuándo usar:** Objetos que cambian de estado.
- **Ventajas:** Organización.
- **Desventajas:** Multiplicación de clases.
- **Ejemplo real:** Ciclo de vida de una orden.

```
interface State {
    void handle(Context context);
}

class Context {
    private State state;
    public void setState(State state) {
        this.state = state;
    }
    public void request() {
        state.handle(this);
    }
}
```

```
class State:
    def handle(self, context): pass

class Context:
    def __init__(self):
        self.state = None
    def set_state(self, state):
        self.state = state
```

```
    def request(self):
        self.state.handle(self)
```

17. Chain of Responsibility

- **Propósito:** Pasar petición entre una cadena de objetos.
- **Cuándo usar:** Procesamiento secuencial.
- **Ventajas:** Flexibilidad.
- **Desventajas:** Difícil trazabilidad.
- **Ejemplo real:** Filtros HTTP, validadores.

```
abstract class Handler {
    protected Handler next;

    public void setNext(Handler next) {
        this.next = next;
    }

    public abstract void handleRequest(String request);
}

class ConcreteHandlerA extends Handler {
    @Override
    public void handleRequest(String request) {
        if (request.equals("A")) {
            System.out.println("Handler A handling request");
        } else if (next != null) {
            next.handleRequest(request);
        }
    }
}

class ConcreteHandlerB extends Handler {
    @Override
    public void handleRequest(String request) {
        if (request.equals("B")) {
            System.out.println("Handler B handling request");
        } else if (next != null) {
            next.handleRequest(request);
        }
    }
}

public class Main {
    public static void main(String[] args) {
        Handler handlerA = new ConcreteHandlerA();
        Handler handlerB = new ConcreteHandlerB();

        handlerA.setNext(handlerB);

        // La solicitud pasa a lo largo de la cadena
        handlerA.handleRequest("B");
        handlerA.handleRequest("A");
```

```
    }
}
```

```
from abc import ABC, abstractmethod

class Handler(ABC):
    def __init__(self):
        self.next = None

    def set_next(self, handler):
        self.next = handler

    @abstractmethod
    def handle_request(self, request):
        pass

class ConcreteHandlerA(Handler):
    def handle_request(self, request):
        if request == "A":
            print("Handler A handling request")
        elif self.next:
            self.next.handle_request(request)

class ConcreteHandlerB(Handler):
    def handle_request(self, request):
        if request == "B":
            print("Handler B handling request")
        elif self.next:
            self.next.handle_request(request)

# Uso
handlerA = ConcreteHandlerA()
handlerB = ConcreteHandlerB()
handlerA.set_next(handlerB)

# La solicitud pasa a lo largo de la cadena
handlerA.handle_request("B")
handlerA.handle_request("A")
```

🏛️ PATRONES DE ARQUITECTURA

18. Layered Architecture

- **Propósito:** Separar la aplicación en capas.
- **Cuándo usar:** Casi siempre.
- **Ventajas:** Organización.
- **Desventajas:** Overhead entre capas.
- **Ejemplo real:** Presentación, servicio, repositorio.

```
// Capa de Datos
class UserRepository {
    public String getUserById(int id) {
        return "User" + id;
```

```
    }
}

// Capa de Negocio
class UserService {
    private UserRepository repository = new UserRepository();

    public String getUserName(int id) {
        return repository.getUserById(id);
    }
}

// Capa de Presentación
public class UserController {
    private UserService service = new UserService();

    public void showUser(int id) {
        System.out.println("User: " + service.getUserName(id));
    }

    public static void main(String[] args) {
        new UserController().showUser(1);
    }
}
```

```
# Capa de Datos
class UserRepository:
    def get_user_by_id(self, id):
        return f"User{id}"

# Capa de Negocio
class UserService:
    def __init__(self):
        self.repository = UserRepository()

    def get_user_name(self, id):
        return self.repository.get_user_by_id(id)

# Capa de Presentación
class UserController:
    def __init__(self):
        self.service = UserService()

    def show_user(self, id):
        print(f"User: {self.service.get_user_name(id)}")

# Uso
controller = UserController()
controller.show_user(1)
```

19. Hexagonal / Clean / Onion Architecture

- **Propósito:** Independencia del dominio.
- **Cuándo usar:** Alta mantenibilidad.
- **Ventajas:** Testeabilidad, bajo acoplamiento.

- **Desventajas:** Complejidad inicial.
- **Ejemplo real:** Sistemas con lógica de negocio rica.

```
// Core: Application layer
interface ProductRepository {
    void save(Product product);
}

class Product {
    private String name;

    public Product(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}

// Application logic (Use cases)
class ProductService {
    private ProductRepository repository;

    public ProductService(ProductRepository repository) {
        this.repository = repository;
    }

    public void createProduct(String name) {
        Product product = new Product(name);
        repository.save(product);
    }
}

// Infrastructure: Database (implementation of repository)
class DatabaseProductRepository implements ProductRepository {
    @Override
    public void save(Product product) {
        System.out.println("Saving product: " + product.getName());
    }
}

public class Main {
    public static void main(String[] args) {
        ProductRepository repository = new DatabaseProductRepository();
        ProductService productService = new ProductService(repository);
        productService.createProduct("Phone");
    }
}
```

```
# Core: Application layer
from abc import ABC, abstractmethod

class ProductRepository(ABC):
```

```
    @abstractmethod
    def save(self, product):
        pass

class Product:
    def __init__(self, name):
        self.name = name

    def get_name(self):
        return self.name

# Application logic (Use cases)
class ProductService:
    def __init__(self, repository: ProductRepository):
        self.repository = repository

    def create_product(self, name):
        product = Product(name)
        self.repository.save(product)

# Infrastructure: Database (implementation of repository)
class DatabaseProductRepository(ProductRepository):
    def save(self, product):
        print(f"Saving product: {product.get_name()}")

# Uso
repository = DatabaseProductRepository()
product_service = ProductService(repository)
product_service.create_product("Phone")
```

20. Microservicios

- **Propósito:** Dividir la aplicación en servicios pequeños.
- **Cuándo usar:** Alta escalabilidad y despliegue independiente.
- **Ventajas:** Escalabilidad, autonomía.
- **Desventajas:** Complejidad, devops.
- **Ejemplo real:** Amazon, Netflix.

```
// Microservice 1: User Service
class UserService {
    public void createUser(String username) {
        System.out.println("Creating user: " + username);
    }
}

// Microservice 2: Order Service
class OrderService {
    public void createOrder(String user) {
        System.out.println("Creating order for user: " + user);
    }
}

// API Gateway
```

```
public class APIService {
    private UserService userService = new UserService();
    private OrderService orderService = new OrderService();

    public void createUserOrder(String username) {
        userService.createUser(username);
        orderService.createOrder(username);
    }

    public static void main(String[] args) {
        APIService apiService = new APIService();
        apiService.createUserOrder("JohnDoe");
    }
}
```

```
# Microservice 1: User Service
class UserService:
    def create_user(self, username):
        print(f"Creating user: {username}")

# Microservice 2: Order Service
class OrderService:
    def create_order(self, user):
        print(f"Creating order for user: {user}")

# API Gateway
class APIService:
    def __init__(self):
        self.user_service = UserService()
        self.order_service = OrderService()

    def create_user_order(self, username):
        self.user_service.create_user(username)
        self.order_service.create_order(username)

# Uso
api_service = APIService()
api_service.create_user_order("JohnDoe")
```

21. Event-Driven Architecture

- **Propósito:** Reaccionar a eventos, no llamadas directas.
- **Cuándo usar:** Sistemas desacoplados.
- **Ventajas:** Escalabilidad, flexibilidad.
- **Desventajas:** Debugging.
- **Ejemplo real:** Kafka, RabbitMQ.

```
import java.util.ArrayList;
import java.util.List;

interface EventListener {
    void onEvent(String event);
}
```

```
class OrderCreatedEventListener implements EventListener {
    @Override
    public void onEvent(String event) {
        System.out.println("Handling event: " + event);
    }
}

class EventManager {
    private List<EventListener> listeners = new ArrayList<>();

    public void addListener(EventListener listener) {
        listeners.add(listener);
    }

    public void notifyListeners(String event) {
        for (EventListener listener : listeners) {
            listener.onEvent(event);
        }
    }
}

public class Main {
    public static void main(String[] args) {
        EventManager eventManager = new EventManager();
        eventManager.addListener(new OrderCreatedEventListener());

        eventManager.notifyListeners("Order Created");
    }
}
```

```
# Event-driven example
from typing import List

class EventListener:
    def on_event(self, event: str):
        pass

class OrderCreatedEventListener(EventListener):
    def on_event(self, event: str):
        print(f"Handling event: {event}")

class EventManager:
    def __init__(self):
        self.listeners: List[EventListener] = []

    def add_listener(self, listener: EventListener):
        self.listeners.append(listener)

    def notify_listeners(self, event: str):
        for listener in self.listeners:
            listener.on_event(event)

# Uso
event_manager = EventManager()
```

```
event_manager.add_listener(OrderCreatedEventListener())
event_manager.notify_listeners("Order Created")
```

22. CQRS + Event Sourcing

- **Propósito:** Separar comandos y consultas.
- **Cuándo usar:** Dominios complejos.
- **Ventajas:** Rendimiento, trazabilidad.
- **Desventajas:** Complejidad extra.
- **Ejemplo real:** Sistemas bancarios, pedidos online.

```
// Command side (write model)
class OrderCommandService {
    public void placeOrder(String orderId) {
        System.out.println("Placing order: " + orderId);
    }
}

// Query side (read model)
class OrderQueryService {
    public void getOrder(String orderId) {
        System.out.println("Getting order: " + orderId);
    }
}

// Event Sourcing (record events)
class EventStore {
    public void recordEvent(String event) {
        System.out.println("Event recorded: " + event);
    }
}

public class Main {
    public static void main(String[] args) {
        OrderCommandService commandService = new OrderCommandService();
        OrderQueryService queryService = new OrderQueryService();
        EventStore eventStore = new EventStore();

        String orderId = "123";
        commandService.placeOrder(orderId);
        eventStore.recordEvent("Order placed: " + orderId);
        queryService.getOrder(orderId);
    }
}
```

```
# Command side (write model)
class OrderCommandService:
    def place_order(self, order_id: str):
        print(f"Placing order: {order_id}")

# Query side (read model)
class OrderQueryService:
    def get_order(self, order_id: str):
```

```
        print(f"Getting order: {order_id}")

# Event Sourcing (record events)
class EventStore:
    def record_event(self, event: str):
        print(f"Event recorded: {event}")

# Uso
command_service = OrderCommandService()
query_service = OrderQueryService()
event_store = EventStore()

order_id = "123"
command_service.place_order(order_id)
event_store.record_event(f"Order placed: {order_id}")
query_service.get_order(order_id)
```

🔗 PATRONES DE INTEGRACIÓN

23. API Gateway

- **Propósito:** Punto de entrada único para microservicios.
- **Cuándo usar:** Arquitecturas distribuidas.
- **Ventajas:** Centraliza seguridad, logging, routing.
- **Desventajas:** Single point of failure si no está bien gestionado.
- **Ejemplo real:** Zuul, API Gateway en AWS.

```
// Simulación simple de un API Gateway
class UserService {
    public String getUser(int id) {
        return "User" + id;
    }
}

class OrderService {
    public String getOrder(int id) {
        return "Order" + id;
    }
}

public class APIGateway {
    private UserService userService = new UserService();
    private OrderService orderService = new OrderService();

    public void routeRequest(String path, int id) {
        switch (path) {
            case "/user":
                System.out.println(userService.getUser(id));
                break;
            case "/order":
                System.out.println(orderService.getOrder(id));
                break;
            default:
```

```
            System.out.println("404 Not Found");
        }
    }

    public static void main(String[] args) {
        APIGateway gateway = new APIGateway();
        gateway.routeRequest("/user", 1);
        gateway.routeRequest("/order", 10);
    }
}
```

```
# Simulación simple de un API Gateway
class UserService:
    def get_user(self, id):
        return f"User{id}"

class OrderService:
    def get_order(self, id):
        return f"Order{id}"

class APIGateway:
    def __init__(self):
        self.user_service = UserService()
        self.order_service = OrderService()

    def route_request(self, path, id):
        if path == "/user":
            print(self.user_service.get_user(id))
        elif path == "/order":
            print(self.order_service.get_order(id))
        else:
            print("404 Not Found")

# Uso
gateway = APIGateway()
gateway.route_request("/user", 1)
gateway.route_request("/order", 10)
```

24. Service Registry & Discovery

- **Propósito:** Localizar servicios dinámicamente.
- **Cuándo usar:** Servicios que escalan o cambian de ubicación.
- **Ventajas:** Desacoplamiento.
- **Desventajas:** Necesita infraestructura extra.
- **Ejemplo real:** Eureka, Consul.

```
import java.util.HashMap;
import java.util.Map;

class ServiceRegistry {
    private Map<String, String> services = new HashMap<>();

    public void register(String name, String url) {
```

```
        services.put(name, url);
    }

    public String discover(String name) {
        return services.getOrDefault(name, "Not Found");
    }
}

public class Main {
    public static void main(String[] args) {
        ServiceRegistry registry = new ServiceRegistry();
        registry.register("user-service", "http://localhost:8081");
        registry.register("order-service", "http://localhost:8082");

        System.out.println("User Service: " + registry.discover("user-service"));
    }
}
```

```
class ServiceRegistry:
    def __init__(self):
        self.services = {}

    def register(self, name, url):
        self.services[name] = url

    def discover(self, name):
        return self.services.get(name, "Not Found")

# Uso
registry = ServiceRegistry()
registry.register("user-service", "http://localhost:8081")
registry.register("order-service", "http://localhost:8082")

print("User Service:", registry.discover("user-service"))
```

25. Circuit Breaker

- **Propósito:** Prevenir fallos en cascada.
- **Cuándo usar:** Llamadas a servicios externos.
- **Ventajas:** Resiliencia.
- **Desventajas:** Configuración adecuada no trivial.
- **Ejemplo real:** Hystrix, Resilience4j.

```
class CircuitBreaker {
    private boolean isOpen = false;
    private int failureCount = 0;
    private int threshold = 3;

    public void call(Runnable serviceCall) {
        if (isOpen) {
            System.out.println("Circuit is open. Skipping call.");
            return;
        }
```

```
        try {
            serviceCall.run();
            failureCount = 0;
        } catch (Exception e) {
            failureCount++;
            System.out.println("Service failed");
            if (failureCount >= threshold) {
                isOpen = true;
                System.out.println("Circuit opened!");
            }
        }
    }
}
```

```
class CircuitBreaker:
    def __init__(self, threshold=3):
        self.threshold = threshold
        self.failure_count = 0
        self.open = False

    def call(self, func):
        if self.open:
            print("Circuit is open. Skipping call.")
            return

        try:
            func()
            self.failure_count = 0
        except Exception as e:
            self.failure_count += 1
            print("Service failed")
            if self.failure_count >= self.threshold:
                self.open = True
                print("Circuit opened!")
```

26. Retry / Backoff

- **Propósito:** Reintentar operaciones fallidas con espera.
- **Cuándo usar:** Fallos temporales.
- **Ventajas:** Mejora tolerancia a errores.
- **Desventajas:** Puede agravar problemas si no se limita.
- **Ejemplo real:** SDKs de AWS, clientes HTTP robustos.

```
class RetryBackoff {
    public void execute(Runnable task) {
        int attempts = 0;
        int delay = 1000;

        while (attempts < 3) {
            try {
                task.run();
                return;
            } catch (Exception e) {
```

```
            attempts++;
            System.out.println("Retrying in " + delay + "ms...");
            try {
                Thread.sleep(delay);
            } catch (InterruptedException ignored) {}
            delay *= 2;
        }
    }

    System.out.println("Failed after retries.");
  }
}
```

```
import time

class RetryBackoff:
    def execute(self, func):
        delay = 1
        for attempt in range(3):
            try:
                func()
                return
            except Exception as e:
                print(f"Retrying in {delay}s...")
                time.sleep(delay)
                delay *= 2
        print("Failed after retries.")
```

27. Message Broker

- **Propósito:** Comunicación asíncrona entre servicios.
- **Cuándo usar:** Alta concurrencia, desacoplamiento.
- **Ventajas:** Escalabilidad, resiliencia.
- **Desventajas:** Tiempos no deterministas, gestión de mensajes.
- **Ejemplo real:** Kafka, RabbitMQ.

```
class MessageBroker {
    public void publish(String topic, String message) {
        System.out.println("Publishing to " + topic + ": " + message);
    }

    public void subscribe(String topic) {
        System.out.println("Subscribed to " + topic);
    }
}
```

```
class MessageBroker:
    def publish(self, topic, message):
        print(f"Publishing to {topic}: {message}")
```

```
    def subscribe(self, topic):
        print(f"Subscribed to {topic}")
```

💾 PATRONES DE PERSISTENCIA

28. Repository

- **Propósito:** Abstraer la lógica de acceso a datos.
- **Cuándo usar:** Acceso frecuente a almacenamiento.
- **Ventajas:** Testeabilidad, separación de lógica.
- **Desventajas:** Capa extra de abstracción.
- **Ejemplo real:** Spring Data, SQLAlchemy.

```
// Dominio
class User {
    private int id;
    private String name;

    public User(int id, String name) { this.id = id; this.name = name; }
    public int getId() { return id; }
    public String getName() { return name; }
}

// Repositorio
import java.util.HashMap;

interface UserRepository {
    void save(User user);
    User findById(int id);
}

class InMemoryUserRepository implements UserRepository {
    private HashMap<Integer, User> storage = new HashMap<>();

    public void save(User user) {
        storage.put(user.getId(), user);
    }

    public User findById(int id) {
        return storage.get(id);
    }
}
```

```
# Dominio
class User:
    def __init__(self, id, name):
        self.id = id
        self.name = name

# Repositorio
class UserRepository:
    def __init__(self):
        self._storage = {}
```

```
    def save(self, user):
        self._storage[user.id] = user

    def find_by_id(self, user_id):
        return self._storage.get(user_id)
```

29. Unit of Work

- **Propósito:** Agrupar múltiples operaciones como una transacción.
- **Cuándo usar:** Múltiples acciones sobre persistencia.
- **Ventajas:** Consistencia.
- **Desventajas:** Overhead en sistemas simples.
- **Ejemplo real:** EntityManager en JPA, sesión en SQLAlchemy.

```
import java.util.ArrayList;
import java.util.List;

class UnitOfWork {
    private List<User> newUsers = new ArrayList<>();

    public void registerNew(User user) {
        newUsers.add(user);
    }

    public void commit(UserRepository repo) {
        for (User u : newUsers) {
            repo.save(u);
        }
        newUsers.clear();
    }
}
```

```
class UnitOfWork:
    def __init__(self):
        self.new_objects = []

    def register_new(self, obj):
        self.new_objects.append(obj)

    def commit(self, repo):
        for obj in self.new_objects:
            repo.save(obj)
        self.new_objects.clear()
```

30. Data Mapper

- **Propósito:** Convertir entre objetos y tablas BD.
- **Cuándo usar:** Separación total de lógica de dominio y persistencia.
- **Ventajas:** Modelo limpio, sin dependencias.
- **Desventajas:** Mayor complejidad.

- **Ejemplo real:** Doctrine en PHP, Hibernate con DTOs.

```
class UserMapper {
    public static User mapRowToUser(ResultSet rs) throws SQLException {
        return new User(rs.getInt("id"), rs.getString("name"));
    }

    public static PreparedStatement mapUserToInsert(Connection conn, User user) throws SQLException {
        PreparedStatement stmt = conn.prepareStatement("INSERT INTO users (id, name) VALUES (?, ?)");
        stmt.setInt(1, user.getId());
        stmt.setString(2, user.getName());
        return stmt;
    }
}
```

```
class UserMapper:
    @staticmethod
    def map_row_to_user(row):
        return User(row["id"], row["name"])

    @staticmethod
    def map_user_to_dict(user):
        return {"id": user.id, "name": user.name}
```

31. Active Record

- **Propósito:** Combinar lógica de negocio y persistencia en una clase.
- **Cuándo usar:** Proyectos pequeños o CRUD sencillos.
- **Ventajas:** Simplicidad.
- **Desventajas:** Acopla dominio y almacenamiento.
- **Ejemplo real:** Django ORM, Eloquent en Laravel.

```
class ActiveRecordUser {
    private int id;
    private String name;

    public ActiveRecordUser(int id, String name) {
        this.id = id;
        this.name = name;
    }

    public void save(Connection conn) throws SQLException {
        PreparedStatement stmt = conn.prepareStatement("INSERT INTO users (id, name) VALUES (?, ?)");
        stmt.setInt(1, id);
        stmt.setString(2, name);
        stmt.executeUpdate();
    }
}
```

```
import sqlite3

class ActiveRecordUser:
    def __init__(self, id, name):
        self.id = id
        self.name = name

    def save(self):
        conn = sqlite3.connect(":memory:")
        conn.execute("CREATE TABLE IF NOT EXISTS users (id INTEGER, name TEXT)")
        conn.execute("INSERT INTO users (id, name) VALUES (?, ?)", (self.id, self.name))
        conn.commit()
        conn.close()
```

♻️ PATRONES TRANSVERSALES

32. Dependency Injection

- **Propósito:** Invertir el control de creación de dependencias.
- **Cuándo usar:** Código reusable, fácilmente testeable.
- **Ventajas:** Bajo acoplamiento.
- **Desventajas:** Requiere framework o infraestructura.
- **Ejemplo real:** Spring, FastAPI, Guice.

```
// Servicio
class EmailService {
    public void sendEmail(String to) {
        System.out.println("Sending email to " + to);
    }
}

// Cliente con inyección de dependencia
class UserController {
    private EmailService emailService;

    // Inyección por constructor
    public UserController(EmailService emailService) {
        this.emailService = emailService;
    }

    public void registerUser(String userEmail) {
        System.out.println("Registering user...");
        emailService.sendEmail(userEmail);
    }

    public static void main(String[] args) {
        EmailService emailService = new EmailService();
        UserController controller = new UserController(emailService);
        controller.registerUser("test@example.com");
    }
}
```

```
# Servicio
class EmailService:
    def send_email(self, to):
        print(f"Sending email to {to}")

# Cliente
class UserController:
    def __init__(self, email_service):
        self.email_service = email_service

    def register_user(self, email):
        print("Registering user...")
        self.email_service.send_email(email)

# Uso
email_service = EmailService()
controller = UserController(email_service)
controller.register_user("test@example.com")
```

33. Aspect-Oriented Programming (AOP)

- **Propósito:** Separar lógica transversal (logs, seguridad).
- **Cuándo usar:** Reglas comunes aplicadas en muchas partes.
- **Ventajas:** Código limpio, DRY.
- **Desventajas:** Difícil trazabilidad en errores.
- **Ejemplo real:** Logs automáticos, manejo de excepciones global.

```
// Spring AOP: ejemplo conceptual de Logging Aspect
@Aspect
@Component
public class LoggingAspect {
    @Before("execution(* UserService.*(..))")
    public void logBefore(JoinPoint joinPoint) {
        System.out.println("Executing: " + joinPoint.getSignature().getName());
    }
}
```

```
# Decorador en Python para simular AOP
def log(func):
    def wrapper(*args, **kwargs):
        print(f"[LOG] Executing: {func.__name__}")
        return func(*args, **kwargs)
    return wrapper

class UserService:
    @log
    def register(self):
        print("Registering user...")

# Uso
```

```
service = UserService()
service.register()
```

34. Configuration as Code

- **Propósito:** Definir configuración como parte del código.
- **Cuándo usar:** Entornos múltiples, CI/CD.
- **Ventajas:** Reproducibilidad, versionado.
- **Desventajas:** Curva de aprendizaje con herramientas.
- **Ejemplo real:** YAML/K8s, Terraform, Ansible.

```
# application.yml (Spring Boot)
server:
  port: 8080

email:
  sender: noreply@example.com
```

```
# config.py con dotenv
import os
from dotenv import load_dotenv

load_dotenv()

EMAIL_SENDER = os.getenv("EMAIL_SENDER", "default@example.com")
```

```
#.env
EMAIL_SENDER=noreply@example.com
```

35. Health Check

- **Propósito:** Verificar que los servicios están vivos.
- **Cuándo usar:** Sistemas distribuidos o contenedores.
- **Ventajas:** Detección rápida de errores.
- **Desventajas:** Puede consumir recursos.
- **Ejemplo real:** `/health`, `/ready`, `/live` endpoints.

```
// Spring Boot
@RestController
public class HealthController {
    @GetMapping("/health")
    public String health() {
        return "OK";
    }
}
```

```
# FastAPI o Flask
from fastapi import FastAPI

app = FastAPI()

@app.get("/health")
def health():
    return {"status": "OK"}
```

36. Rate Limiting

- **Propósito:** Controlar el número de peticiones permitidas.
- **Cuándo usar:** APIs públicas o sensibles.
- **Ventajas:** Protección ante abusos.
- **Desventajas:** Necesita mecanismo de control distribuido.
- **Ejemplo real:** NGINX, Kong, API Gateway.

```
// Usando Bucket4j
Bucket bucket = Bucket4j.builder()
    .addLimit(Bandwidth.classic(10, Refill.greedy(10, Duration.ofMinutes(1))))
    .build();

public boolean tryRequest() {
    return bucket.tryConsume(1);
}
```

```
# Rate limit simple en Flask con Flask-Limiter
from flask import Flask
from flask_limiter import Limiter
from flask_limiter.util import get_remote_address

app = Flask(__name__)
limiter = Limiter(get_remote_address, app=app, default_limits=["5 per minute"])

@app.route("/api")
@limiter.limit("2/second")
def limited_api():
    return "Request allowed"
```