■ Patrones de Diseño y Antipatrones

Los patrones de diseño son soluciones probadas a problemas comunes de arquitectura o comportamiento. No son reglas fijas, sino plantillas mentales que ayudan a escribir código más mantenible, reutilizable y extensible.

■ 1. Patrones Creacionales

Singleton

Idea: garantizar que exista una única instancia de una clase.

Cuándo usarlo: punto de acceso global (configuración, conexión, logger, etc.).

```
const Config = (function () {
  let instance;
  function createInstance() {
    return { modo: "producción", puerto: 8080 };
  }
  return {
    getInstance: function () {
      if (!instance) instance = createInstance();
      return instance;
    },
  };
})();

const c1 = Config.getInstance();
const c2 = Config.getInstance();
console.log(c1 === c2); // true – misma instancia
```

■■ Cuidado: evita usarlo como variable global disfrazada.

Factory Method

Idea: centraliza la creación de objetos sin acoplar el código.

Cuándo usarlo: hay varias clases parecidas y eliges en tiempo de ejecución.

```
class Vehiculo { conducir() {} }
class Auto extends Vehiculo { conducir() { console.log("Conduciendo un auto ■"); } }
class Moto extends Vehiculo { conducir() { console.log("Conduciendo una moto ■■"); } }

class VehiculoFactory {
  static crear(tipo) {
    if (tipo === "auto") return new Auto();
    if (tipo === "moto") return new Moto();
    throw new Error("Tipo no válido");
  }
}

const v1 = VehiculoFactory.crear("moto");
v1.conducir(); // ■■
```

■ Evita usar `new` repetido en el código.

Builder

Idea: construir objetos complejos paso a paso.

Cuándo usarlo: un objeto tiene muchas propiedades opcionales.

```
class PCBuilder {
  constructor() { this.pc = {}; }
  setCPU(cpu) { this.pc.cpu = cpu; return this; }
  setRAM(ram) { this.pc.ram = ram; return this; }
  setGPU(gpu) { this.pc.gpu = gpu; return this; }
  build() { return this.pc; }
}

const pcGamer = new PCBuilder()
  .setCPU("i9").setRAM("32GB").setGPU("RTX 4080").build();
console.log(pcGamer);
```

Ventaja: el código se lee como una receta.

■■ 2. Patrones Estructurales

Adapter

Adapta una interfaz antigua a una nueva sin modificar el código original.

```
class OldPrinter { print(t) { console.log(`Imprimiendo: ${t}`); } }
class PrinterAdapter {
  constructor(oldPrinter) { this.oldPrinter = oldPrinter; }
  printText(text) { this.oldPrinter.print(text); }
}

const adapter = new PrinterAdapter(new OldPrinter());
adapter.printText("Documento PDF");
```

Útil para integrar librerías viejas con sistemas nuevos.

Decorator

Añade comportamiento extra sin alterar la clase original.

```
function loggear(fn) {
  return function (...args) {
    console.log("Ejecutando:", fn.name);
    return fn(...args);
  };
}
function sumar(a,b){ return a+b; }
const sumarConLog = loggear(sumar);
console.log(sumarConLog(3,2));
```

Útil para logging, validación o permisos.

Proxy

Controla el acceso a un objeto interceptando sus operaciones.

```
const usuario = { nombre: "Rick", rol: "user" };
const proxy = new Proxy(usuario, {
  get(t, p){ console.log(`Accediendo a ${p}`); return t[p]; },
  set(t, p, v){ if(p==="rol"&&v==="admin")throw Error("Prohibido"); t[p]=v; }
});
proxy.nombre; proxy.rol="admin";
```

■■ 3. Patrones Comportamentales

Observer (Publicar/Suscribir)

Un objeto notifica a otros cuando cambia.

```
class Evento {
  constructor(){ this.obs=[]; }
  suscribir(fn){ this.obs.push(fn); }
  notificar(d){ this.obs.forEach(fn=>fn(d)); }
}
const evento=new Evento();
evento.suscribir(m=>console.log("Recibido:",m));
evento.notificar("Hola observador!");
```

Ideal para sistemas reactivos o interfaces dinámicas.

Strategy

Encapsula diferentes algoritmos intercambiables.

```
class PagoTarjeta{ pagar(m){console.log(`Pagando $${m} con tarjeta`);} }
class PagoPayPal{ pagar(m){console.log(`Pagando $${m} con PayPal`);} }
class ContextoPago{
  setEstrategia(e){ this.e=e; }
  ejecutar(m){ this.e.pagar(m); }
}
const pago=new ContextoPago();
pago.setEstrategia(new PagoTarjeta()); pago.ejecutar(100);
pago.setEstrategia(new PagoPayPal()); pago.ejecutar(200);
```

Command

Encapsula una acción como objeto, útil para deshacer o macros.

```
class Encender{ execute(){console.log("Encendiendo ■");} undo(){console.log("Apagando ■");} }
class Control{ ejecutar(c){c.execute();} deshacer(c){c.undo();} }
const c=new Control(); const cmd=new Encender(); c.ejecutar(cmd); c.deshacer(cmd);
```

■ Puntos Avanzados y Antipatrones

■ *Inmutabilidad*

Evita modificar estructuras originales.

```
arr.push(5); // ■
arr = [...arr, 5]; // ■
```

Composición sobre herencia

```
const volador={volar(){console.log("Vuelo!");}};
const nadador={nadar(){console.log("Nado!");}};
const pato={...volador,...nadador};
pato.volar(); pato.nadar();
```

Principio de responsabilidad única (SRP)

Cada módulo debe hacer una sola cosa y hacerla bien.

Programación declarativa

```
// Imperativo
let pares=[]; for(let i of [1,2,3,4]) if(i%2===0) pares.push(i);
// Declarativo
const pares2=[1,2,3,4].filter(x=>x%2===0);
```

■ Antipatrones comunes

Callback Hell: Usa Promesas o async/await para evitar anidamientos profundos.

God Object: Divide responsabilidades en módulos o clases específicas.

Global State: Evita estados globales; usa inyección de dependencias o Singletons controlados.

Premature Optimization: Primero haz que funcione, luego mide y optimiza.

Duplicated Code: Extrae funciones o clases reutilizables.

Spaghetti Code: Usa patrones de diseño para separar capas.

Abusar de la herencia: Prefiere composición, interfaces o estrategias.

Resumen:
Creacionales → cómo crear objetos.
Estructurales → cómo componer clases.
Comportamentales → cómo interactúan.

■ Los antipatrones no son 'malos' por sí mismos, solo cuando se aplican fuera de contexto.