Patrones GoF – Versión ultra simple con ejemplos

> Lenguaje de ejemplo: JavaScript muy sencillo.
> En todos los casos la idea es que se vea **para qué sirve** y **un posible uso real**.

---

1. Factory Method

Idea: Una clase decide **qué objeto concreto crear**, pero el código que lo usa solo conoce la "fábrica".

Uso real simple: Crear distintos tipos de notificaciones (`EmailNotification`, `SMSNotification`) según un tipo recibido de la API.

```
class NotificationFactory {
  static create(type) {
    if (type === "email") return new EmailNotification();
    if (type === "sms") return new SMSNotification();
  }
}
```

---

2. Abstract Factory

Idea: Fábrica de **familias de objetos relacionados**, que encajen entre sí.

Uso real simple: Crear componentes de UI para "tema claro" o "tema oscuro" (botón, modal, input) con el mismo estilo.

```
class LightUIFactory {
  createButton() {
    return new LightButton();
  }
  createModal() {
    return new LightModal();
  }
}
```

---

3. Builder

Idea: Construir un objeto complejo **paso a paso**, con una API fluida.

Uso real simple: Montar un pedido de comida (plato principal, bebida, postre) sin un constructor gigante.

```
class OrderBuilder {
  setMain(main) {
    this.main = main;
    return this;
  }
  setDrink(drink) {
    this.drink = drink;
    return this;
  }
  build() {
    return new Order(this.main, this.drink);
  }
}
```

---

4. Prototype

Idea: Crear nuevos objetos **clonando** un objeto existente.

Uso real simple: Duplicar una plantilla de documento con algunos cambios (título, fecha).

```
const invoiceTemplate = { tax: 21, footer: "Gracias por su compra" };
const invoice = { ...invoiceTemplate, number: 123 };
```

---

5. Singleton

Idea: Tener **una única instancia global** de algo.

Uso real simple: Configuración compartida en toda la app (idioma, tema).

```
class Config {
  constructor() {
    if (Config.instance) return Config.instance;
    this.lang = "es";
    Config.instance = this;
  }
}
```

---

6. Adapter

Idea: Un “enchufe” que **adapt(a) una interfaz a otra**.

Uso real simple: Adaptar la respuesta de una API antigua al formato que espera tu app nueva.

```
class OldUserAdapter {
  constructor(oldUser) {
    this.oldUser = oldUser;
  }
  getName() {
    return this.oldUser.full_name;
  }
}
```

---

7. Bridge

Idea: Separar **abstracción** de **implementación** para combinarlas libremente.

Uso real simple: Enviar notificaciones (abstracción) por email, SMS o push (implementación).

```
class Notification {
  constructor(channel) {
    this.channel = channel;
  }
  send(message) {
    this.channel.send(message);
  }
}
```

---

8. Composite

Idea: Tratar **objetos individuales y grupos** de la misma forma.

Uso real simple: Representar carpetas y archivos en un árbol y recorrerlos igual.

```
class Folder {
  constructor() {
    this.children = [];
  }
  add(item) {
    this.children.push(item);
  }
}
```

---

9. Decorator

Idea: Añadir funcionalidad extra a un objeto **sin modificar su código**.

Uso real simple: Envolver una función de API para añadir logs o cache.

```
function withLog(fn) {
  return (...args) => {
    console.log("Llamando a función");
    return fn(...args);
  };
}
```

---

10. Facade

Idea: Crear una **interfaz simple** a un sistema complejo.

Uso real simple: Una clase que encapsula todas las llamadas a `localStorage`.

```
class StorageFacade {
  save(key, data) {
    localStorage.setItem(key, JSON.stringify(data));
  }
  load(key) {
    return JSON.parse(localStorage.getItem(key) || "null");
  }
}
```

---

11. Flyweight

Idea: Compartir objetos **ligeros y repetidos** para ahorrar memoria.

Uso real simple: Reutilizar iconos o estilos comunes en una app de mapas para miles de marcadores.

```
class IconFactory {
  constructor() {
    this.cache = {};
  }
  get(type) {
    if (!this.cache[type]) this.cache[type] = new Icon(type);
    return this.cache[type];
  }
}
```

---

12. Proxy

Idea: Un objeto que **actúa como sustituto** de otro para añadir control.

Uso real simple: Proxy de una API que cachea respuestas o comprueba permisos antes de llamar.

```
class ApiProxy {
  constructor(api) {
    this.api = api;
    this.cache = {};
  }
  getUser(id) {
    if (!this.cache[id]) this.cache[id] = this.api.getUser(id);
    return this.cache[id];
  }
}
```

---

13. Chain of Responsibility

Idea: Una **cadena de manejadores**, cada uno decide si procesa o pasa al siguiente.

Uso real simple: Sistema de soporte donde primero revisa el bot, luego un agente normal, luego el supervisor.

```
class Handler {
  setNext(next) {
    this.next = next;
  }
  handle(request) {
    if (this.next) return this.next.handle(request);
  }
}
```

---

14. Command

Idea: Encapsular una petición en un **objeto comando**.

Uso real simple: Botones que deshacen o rehacen acciones (deshacer borrar, deshacer mover).

```
class DeleteCommand {
  constructor(file) {
    this.file = file;
  }
  execute() {
    this.file.delete();
  }
  undo() {
    this.file.restore();
  }
}
```

---

15. Iterator

Idea: Recorrer una colección **sin exponer su estructura interna**.

Uso real simple: Recorrer un resultado paginado sin saber cómo se guarda por dentro.

```
class ArrayIterator {
  constructor(items) {
    this.items = items;
    this.index = 0;
  }
  next() {
    return this.items[this.index++];
  }
  hasNext() {
    return this.index < this.items.length;
  }
}
```

---

16. Mediator

Idea: Un objeto central que **coordina la comunicación** entre muchos objetos.

Uso real simple: Sala de chat donde los usuarios no se hablan directamente, sino a través de la sala.

```
class ChatRoom {
  constructor() {
    this.users = [];
  }
  send(from, message) {
    this.users.forEach((u) => {
      if (u !== from) u.receive(message);
    });
  }
}
```

---

17. Memento

Idea: Guardar y restaurar el **estado** de un objeto sin exponer sus detalles.

Uso real simple: Deshacer cambios de un editor de texto guardando “fotografías” del contenido.

```
class Editor {
  save() {
    return { text: this.text };
  }
```

```
  restore(memento) {
    this.text = memento.text;
  }
}
```

---

18. Observer

Idea: Uno emite eventos y **muchos observadores** reaccionan.

Uso real simple: Sistema de notificaciones donde varios componentes escuchan cambios de usuario.

```
class Subject {
  constructor() {
    this.observers = [];
  }
  subscribe(o) {
    this.observers.push(o);
  }
  notify(data) {
    this.observers.forEach((o) => o.update(data));
  }
}
```

---

19. State

Idea: Cambiar el **comportamiento** de un objeto según su estado interno.

Uso real simple: Pedido que se comporta distinto si está “nuevo”, “enviado” o “entregado”.

```
class Order {
  setState(state) {
    this.state = state;
  }
  next() {
    this.state.next(this);
  }
}
```

---

20. Strategy

Idea: Varias **estrategias intercambiables** para hacer lo mismo de formas distintas.

Uso real simple: Calcular gastos de envío según país o tipo de cliente.

```
class ShippingContext {
  setStrategy(strategy) {
    this.strategy = strategy;
  }
  getCost(amount) {
    return this.strategy.cost(amount);
  }
}
```

---

21. Template Method

Idea: Definir el **esqueleto de un algoritmo** y dejar que las subclases rellenen pasos.

Uso real simple: Exportar datos a distintos formatos (CSV, JSON) con pasos comunes: preparar datos, formatear, guardar.

```
class Exporter {
  export(data) {
    const prepared = this.prepare(data);
    const formatted = this.format(prepared);
    this.save(formatted);
  }
}
```

---

22. Visitor

Idea: Separar operaciones de la estructura de objetos, **añadiendo operaciones sin tocar las clases**.

Uso real simple: Recorrer una estructura de elementos de factura (líneas, impuestos, descuentos) para calcular un informe distinto (precio total, IVA, etc.).

```
class PriceVisitor {
  visitProduct(p) {
    this.total += p.price;
  }
  visitDiscount(d) {
    this.total -= d.amount;
  }
}
```
