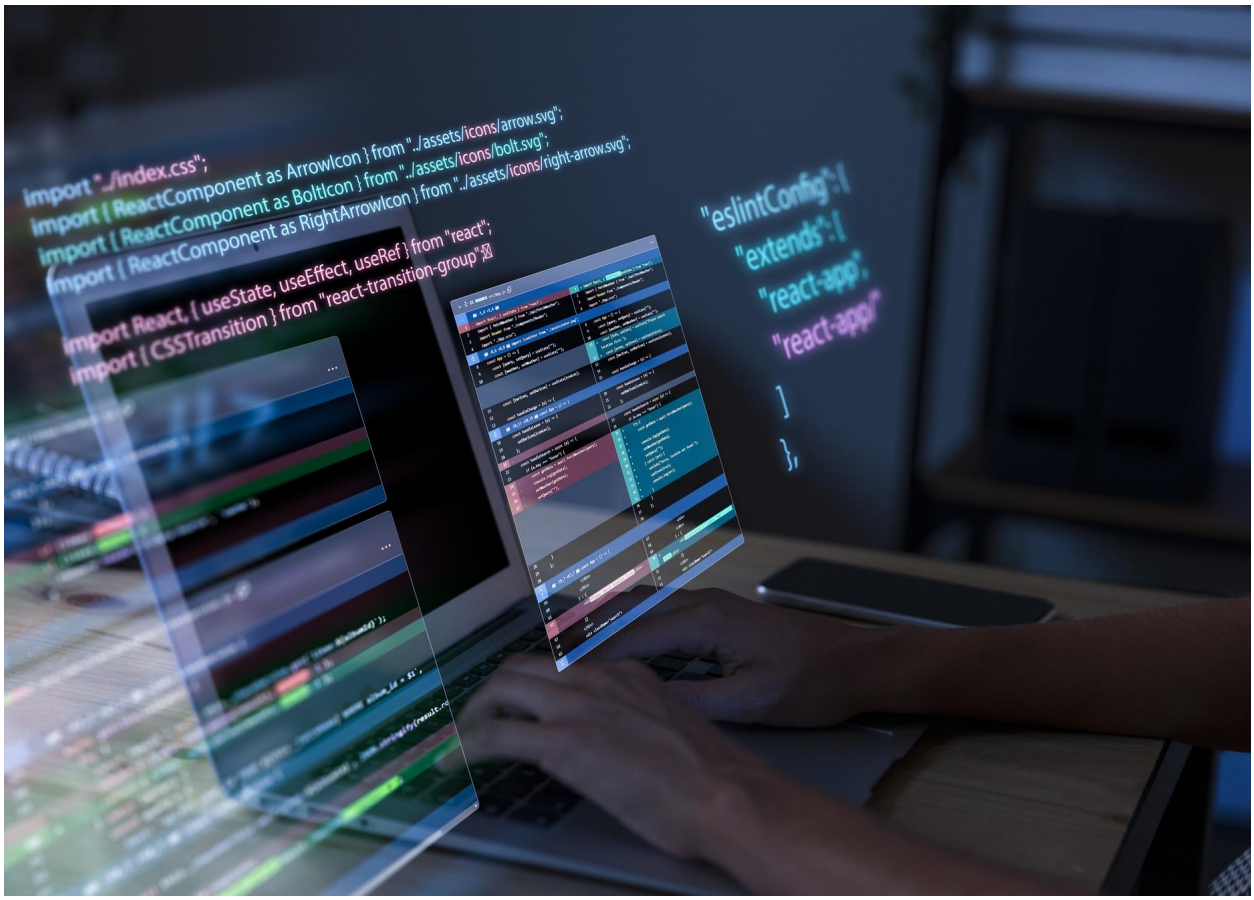
PROMETEO

DAM & DAW / ENTORNOS DE DESARROLLO

UNIDAD 4

Optimización, patrones y buenas prácticas



Autor: Luz María Álvarez Moreno
Fecha: 2/12/2025

Sesión 11: Patrones de diseño (creacionales y estructurales)

¿Qué es un patrón de diseño?

Problema

En programación orientada a objetos, los equipos se encuentran siempre con problemas que se repiten:

- "¿Cómo creo este objeto sin que el código quede rígido?"
- "¿Cómo añado nuevas funcionalidades sin romper el código viejo?"
- "¿Cómo conecto mi sistema con otro que tiene una interfaz distinta?"

En lugar de inventar una solución nueva cada vez, distintas personas fueron recopilando **soluciones típicas** que funcionaban bien.

A esas soluciones generales se les llamó **patrones de diseño**.

Un **patrón de diseño** es una **solución reutilizable, probada y general** a un problema recurrente de diseño de software.

- **No es código copiado y pegado.**
- Es más bien: *"Cuando tengas este tipo de problema, esta es una forma ordenada de resolverlo"*.

¿Qué suele incluir la descripción de un patrón?

Para cada patrón normalmente se explica:

1. **Nombre**: Singleton, Factory, Adapter…
2. **Problema:** que resuelve cuándo aparece.
3. **Idea clave**: cómo lo resuelve.
4. **Participantes:** qué clases/objetos intervienen.
5. **Consecuencias:** ventajas y posibles inconvenientes.
6. A veces un **diagrama de clases** y **ejemplos de código**.

¿Por qué usar patrones?

Los patrones están muy ligados a tres ideas importantes:

1. **Desacoplamiento**
   El código no debería depender de detalles concretos, sino de **interfaces/abstracciones**.
   - Así, si mañana cambias una clase concreta, no tienes que reescribir todo.

2. **Reutilización**
   - Un mismo patrón lo puedes aplicar en un programa pequeño, en una app web, en un videojuego, etc.

3. **Mantenibilidad**
   - Si otro programador ve "esto es un Factory" o "esto es un Decorator", entiende rápido qué hace el diseño, aunque no conozca tu proyecto.

Clasificación general

No vamos a ver todos los patrones del mundo, solo **dos familias**:

1. **Patrones creacionales**: se centran en **cómo se crean los objetos**.
   - Singleton
   - Factory Method / Abstract Factory
   - Builder

2. **Patrones estructurales**: se centran en **cómo se combinan/relacionan objetos y clases**.
   - Adapter
   - Composite
   - Decorator

Existen más tipo, como los **de comportamiento**, pero los veremos más adelante.

Patrones Creacionales

Problema general de los creacionales

Si creamos objetos con new por todas partes (por ejemplo new MySQLConnection()), acabamos con código:

- Difícil de cambiar (¿y si mañana queremos PostgreSQL?).
- Difícil de testear.
- Muy rígido.

Los patrones creacionales intentan **separar la decisión de "cómo se crea" el objeto** del código que lo usa.

Singleton

Problema
Hay recursos que en una aplicación **debe haber solo uno**:

- Un único objeto de configuración global.
- Un único registro de logs.
- Una única conexión compartida (en aplicaciones pequeñas).

No quieres que el programador pueda hacer new Config() por todos lados y crear instancias duplicadas.

Idea

- La propia clase controla que **solo exista una instancia**.
- Se guarda la instancia en un atributo estático.
- Ofrece un método estático, por ejemplo getInstance(), que devuelve siempre la misma.

Ventajas

- Controlas que sólo haya una instancia.
- Es fácil acceder: Config.getInstance().

Inconvenientes

- Abusar de Singleton hace el código similar a variables globales.
- Dificulta las pruebas unitarias (es más difícil “simular” esa instancia).

Cuándo usarlo

- Cuando **de verdad** solo tenga sentido una instancia.
- Cuando se use en varios sitios y quieres centralizarlo.

Factory Method / Abstract Factory

Problema

Imagina que tienes una aplicación que envía notificaciones:

```
EmailNotification n = new EmailNotification();
n.send("Hola");
```

Si el día de mañana quieres usar SMSNotification, cambiarías mucho código:

```
SMSNotification n = new SMSNotification();
```

Cada vez que cambias el tipo concreto, tocas muchas partes del sistema.

Idea

- El código cliente no instancia directamente con new.
- Llama a un **“método factory”** que decide qué clase concreta crear.

Ejemplo conceptual:

```
Notification n = NotificationFactory.create("email");
n.send("Hola");
```

Si mañana cambias el tipo, solo cambias la factory, no el resto del código.

Factory Method

Es cuando una **clase** tiene un método protegido/abstracto que decide qué objeto concreto crear.

Abstract Factory
Es una **factory de familias de objetos**.

Por ejemplo: GUI Windows, GUI Mac, GUI Linux: cada factory crea botones, menús, ventanas, etc., pero todos compatibles.

Ventajas

- Desacopla el **qué** quieres crear del **cómo** lo creas.
- Facilita cambiar implementaciones concretas (Email → SMS, MySQL → PostgreSQL, etc.).

Builder

Problema
Supón que tienes una clase Usuario con muchos parámetros:

Usuario u = new Usuario("Luis", "Gómez", 35, true, false, true, "ES", ...);

Esto es:

- Difícil de leer.
- Difícil de saber qué hace cada true/false.
- Lioso cuando algunos atributos son obligatorios y otros opcionales.

Idea

Separar la **construcción** de un objeto complejo de su **representación final**.

La clase Builder se va configurando por pasos:

```
Usuario u = new UsuarioBuilder("Luis")   // nombre obligatorio
            .apellidos("Gómez")
            .edad(35)
            .esAdmin(true)
```

```
        .build();
```

Ventajas

- Código mucho más legible.
- Evita constructores con 15 parámetros.
- Puedes tener configuraciones distintas sin mil constructores sobrecargados.

Patrones Estructurales

Problema general de los estructurales

Tienen que ver con **cómo organizar y conectar** clases y objetos.

Ejemplos de problemas típicos:

- Tengo una clase A y una B con interfaces diferentes; quiero que colaboren: **Adapter**.
- Quiero tratar objetos individuales y conjuntos de objetos igual: **Composite**.
- Quiero añadir capacidades a un objeto sin crear mil subclases nuevas: **Decorator**.

Adapter

Problema
Tu sistema espera una interfaz, por ejemplo:

```
interface Player {
   void play();
}
```

Pero el dispositivo o API externa que quieres usar tiene otro formato:

```
class OldMediaDevice {
   public void startPlayback() { ... }
}
```

No quieres cambiar todo tu código para adaptarte al dispositivo.

Idea

Crear una clase **adaptador** que implemente la interfaz que tu sistema espera (Player) y por dentro delegue en el dispositivo real:

```
class OldMediaAdapter implements Player {
   private OldMediaDevice device;

   public OldMediaAdapter(OldMediaDevice device) {
      this.device = device;
   }

   @Override
   public void play() {
      device.startPlayback();  // traducción
   }
}
```

Así el resto de tu programa ve un Player, pero por dentro usas OldMediaDevice.

Composite

Idea básica

- Permite tratar **objetos individuales** y **grupos de objetos** de la misma forma.
- Ejemplo: en un sistema de menús, una opción sencilla y un submenú (que tiene muchas opciones) se tratan igual.

Decorator

Problema

Tienes una clase básica:
class BasicCoffee { getCost(); getIngredients(); }

Y quieres versiones:

- Café con leche
- Café con azúcar
- Café con leche y azúcar
- Café descafeinado con leche y azúcar
- …

Si usas herencia pura, acabarías con muchas subclases: CafeConLeche, CafeConAzucar, CaféConLecheYAzucar, etc.

Idea

- Mantienes una interfaz común, por ejemplo Café.
- La implementación básica es BasicoCafe.
- Luego creas **decoradores** que reciben un Café y añaden comportamiento:

```
class MilkDecorator implements Coffee {
   private Coffee base;
   public MilkDecorator(Coffee base) { this.base = base; }

   public double getCost() { return base.getCost() + 0.5; }
   public String getIngredients() { return base.getIngredients() + ",
leche"; }
}
```

Puedes “envolver” un café con varios decoradores:

```
Coffee c = new SugarDecorator(new MilkDecorator(new
BasicCoffee()));
```

Ventajas

- Añades funcionalidades **dinámicamente**, sin explotar la jerarquía de herencia.
- Puedes combinar decoradores de forma flexible.

Ejemplo real (Netflix)

- Funcionalidades opcionales: descargas, perfiles infantiles, estadísticas de visionado, etc.

- Netflix puede envolver el servicio de vídeo con decoradores según el plan del usuario.

6. Resumen

1. **Patrones de diseño: soluciones reutilizables** a problemas comunes.
2. No son recetas cerradas, son **formas de pensar** el diseño.
3. **Creacionales**: se preocupan de **cómo se crean los objetos**.
4. **Estructurales**: se preocupan de **cómo se relacionan y organizan los objetos**.
5. Beneficios: **desacoplamiento, reutilización, mantenibilidad, escalabilidad**.
6. Debemos **entender qué problema resuelve cada patrón**.