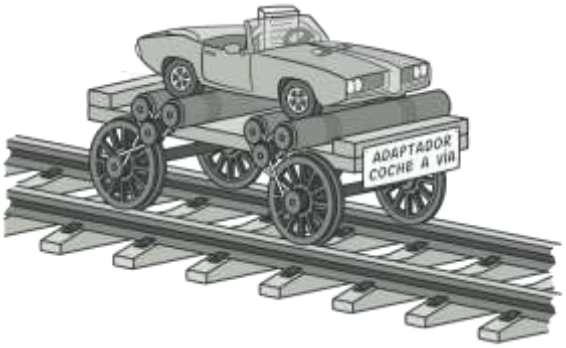
Patrones Estructurales

Los patrones estructurales explican cómo ensamblar objetos y clases en estructuras más grandes, a la vez que se mantiene la flexibilidad y eficiencia de estas estructuras.

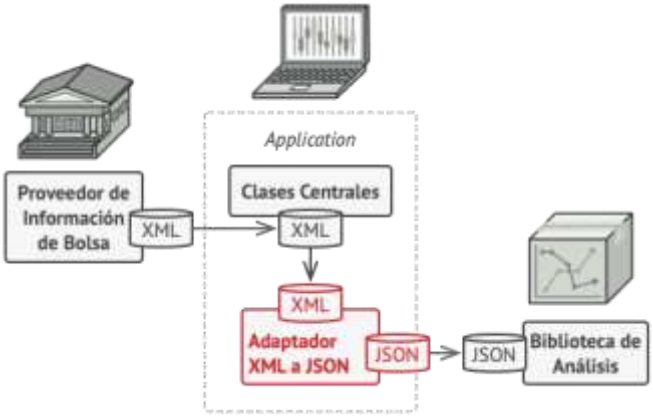
Adapter

También llamado: Adaptador, Envoltorio, Wrapper



Propósito

El **patrón Adapter** se usa cuando dos clases no son compatibles entre sí, pero necesitamos que trabajen juntas.
Ejemplo Aplicación A – Gestiona sus datos en formato XML
Ejemplo Aplicación B – Gestiona sus datos en formato JSON
Mediante el Adaptador puedo lograr comunicar ambas aplicaciones



Básicamente, es como un **enchufe adaptador**: si tu notebook tiene un cargador con clavija americana y en tu casa solo hay enchufes europeos, necesitas un adaptador. El adaptador no cambia ni el enchufe ni la notebook, solo hace que puedan conectarse.

Ejemplo en Java

Imaginemos que tenemos un sistema que espera objetos que implementen una interfaz **MediaPlayer**, pero tenemos una clase existente (**AdvancedMediaPlayer**) que no implementa esa interfaz.

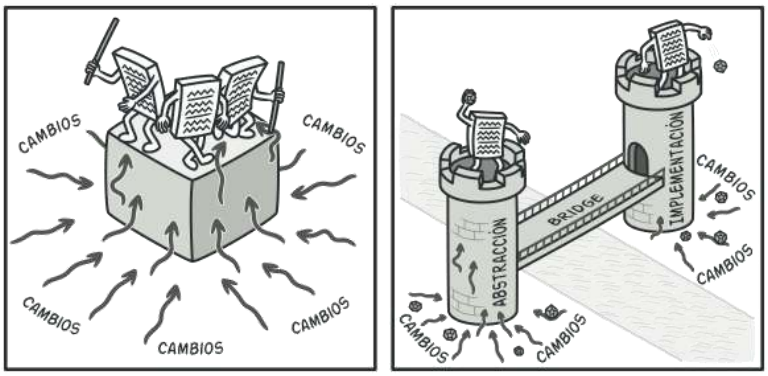
Usamos un **Adapter** para que sea compatible.

```java
// La interfaz que nuestro sistema espera
interface MediaPlayer {
    void play(String audioType, String fileName);
}

// Clase existente con una interfaz distinta
class AdvancedMediaPlayer {
    void playMp4(String fileName) {
        System.out.println("Reproduciendo archivo MP4: " + fileName);
    }

    void playVlc(String fileName) {
        System.out.println("Reproduciendo archivo VLC: " + fileName);
    }
}

// El adaptador que conecta ambas interfaces
class MediaAdapter implements MediaPlayer {
    private AdvancedMediaPlayer advancedPlayer;

    public MediaAdapter() {
        advancedPlayer = new AdvancedMediaPlayer();
    }

    @Override
    public void play(String audioType, String fileName) {
        if(audioType.equalsIgnoreCase("mp4")) {
            advancedPlayer.playMp4(fileName);
        } else if(audioType.equalsIgnoreCase("vlc")) {
            advancedPlayer.playVlc(fileName);
        } else {
            System.out.println("Formato no soportado: " + audioType);
        }
    }
}

// El cliente que utiliza la interfaz MediaPlayer
class AudioPlayer implements MediaPlayer {
    private MediaAdapter mediaAdapter;

    @Override
    public void play(String audioType, String fileName) {
        if(audioType.equalsIgnoreCase("mp3")) {
            System.out.println("Reproduciendo archivo MP3: " + fileName);
        } else {
            mediaAdapter = new MediaAdapter();
            mediaAdapter.play(audioType, fileName);
        }
    }
}

// Clase principal con main para ejecutar
public class Main {
    public static void main(String[] args) {
```

```
        AudioPlayer player = new AudioPlayer();

        player.play("mp3", "cancion1.mp3");
        player.play("mp4", "video1.mp4");
        player.play("vlc", "pelicula1.vlc");
        player.play("avi", "archivo.avi"); // no soportado
    }
}
```

Bridge

También llamado: Puente



Propósito

El patrón **Bridge** (puente) se usa cuando queremos **separar una abstracción de su implementación**, de modo que ambas puedan evolucionar de forma independiente.

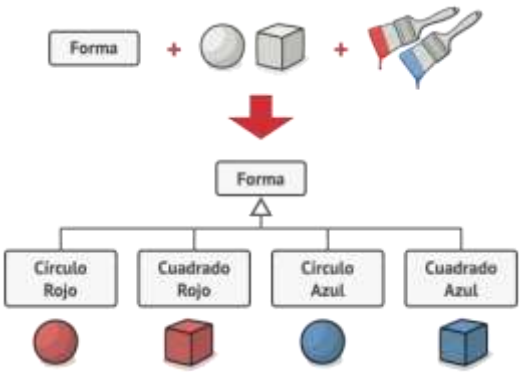
En palabras simples en lugar de tener una sola jerarquía enorme de clases, dividimos el problema en **dos jerarquías separadas** que se conectan mediante un **puente (bridge)**.

Ejemplo:

Digamos que tienes una clase geométrica `Forma` con un par de subclases: `Círculo` y `Cuadrado`. Deseas extender esta jerarquía de clase para que incorpore colores, por lo que planeas crear las subclases de forma `Rojo` y `Azul`. Sin embargo, como ya tienes dos subclases, tienes que crear cuatro combinaciones de clase, como `CírculoAzul` y `CuadradoRojo`.

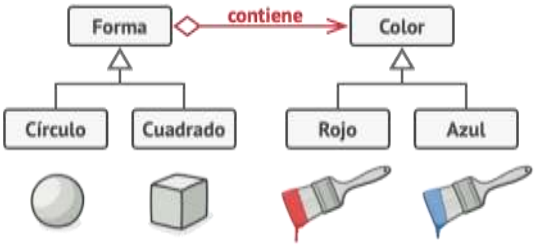
Añadir nuevos tipos de forma y color a la jerarquía hará que ésta crezca exponencialmente. Por ejemplo, para añadir una forma de triángulo deberás introducir dos subclases, una para cada color. Y, después, para añadir un nuevo color habrá que crear tres subclases, una para cada tipo de forma. Cuanto más avancemos, peor será.

Problema:



Solución:

Podemos evitar la explosión de una jerarquía de clase transformándola en varias jerarquías relacionadas.



Con esta solución, podemos extraer el código relacionado con el color y colocarlo dentro de su propia clase, con dos subclases: `Rojo` y `Azul`. La clase `Forma` obtiene entonces un campo de referencia que apunta a uno de los objetos de color. Ahora la forma puede delegar cualquier trabajo relacionado con el color al objeto de color vinculado. Esa referencia actuará como un puente entre las clases `Forma` y `Color`. En adelante, añadir nuevos colores no exigirá cambiar la jerarquía de forma y viceversa.

```
// Implementor
interface Color {
    String aplicarColor();
}

// Concrete Implementors
class Rojo implements Color {
    @Override
    public String aplicarColor() {
        return "Rojo";
    }
}

class Azul implements Color {
    @Override
    public String aplicarColor() {
        return "Azul";
    }
}

// Abstraction
abstract class Forma {
    protected Color color;

    public Forma(Color color) {
```

```
        this.color = color;
    }

    abstract void dibujar();
}

// Refined Abstractions
class Circulo extends Forma {
    public Circulo(Color color) {
        super(color);
    }

    @Override
    void dibujar() {
        System.out.println("Dibujando un círculo de color " + color.aplicarColor());
    }
}

class Cuadrado extends Forma {
    public Cuadrado(Color color) {
        super(color);
    }

    @Override
    void dibujar() {
        System.out.println("Dibujando un cuadrado de color " + color.aplicarColor());
    }
}

// Demo
public class BridgeDemo {
    public static void main(String[] args) {
        Forma circuloRojo = new Circulo(new Rojo());
        Forma cuadradoAzul = new Cuadrado(new Azul());

        circuloRojo.dibujar();
        cuadradoAzul.dibujar();
    }
}
```

“separar una abstracción de su implementación”

1. ¿Qué es la abstracción?

La **abstracción** es **qué hace** algo, sin importar cómo lo hace.

Ejemplo:

- Un **control remoto** sirve para *encender, apagar, cambiar canal, subir volumen*.
- No importa si es de un televisor Samsung, LG o Sony, esas funciones son la abstracción.

2. ¿Qué es la implementación?

La **implementación** es **cómo realmente hace eso cada marca o modelo**.

- El televisor Samsung tiene un código interno propio para cambiar el canal.
- El LG lo hace distinto.
- El Sony tiene otro sistema.

3. ¿Qué significa separar la abstracción de la implementación?

Significa que no queremos mezclar en una sola clase el **qué hace** (la abstracción) con el **cómo lo hace** (la implementación).

En lugar de eso, hacemos dos jerarquías:

1. Una jerarquía para la **abstracción** (ej: "ControlRemoto" → Botones que usa el usuario).
2. Otra jerarquía para la **implementación** (ej: "TelevisorSamsung", "TelevisorLG").

El **Bridge** conecta ambas.

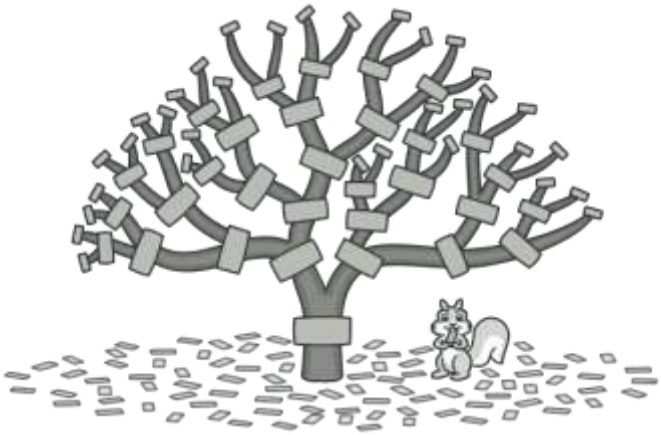
En pocas palabras, separar abstracción de implementación = poder cambiar el **qué** y el **cómo** de manera independiente, sin que uno dependa rígidamente del otro.

Composite

También llamado: Objeto compuesto, Object Tree

Propósito

Composite es un patrón de diseño estructural que te permite componer objetos en estructuras de árbol y trabajar con esas estructuras como si fueran objetos individuales, es decir queremos tratarlos **de la misma manera**.



Imaginá un **sistema de archivos**:

- Un archivo es un elemento individual.
- Una carpeta puede contener varios archivos y también otras carpetas.

Gracias a **Composite**, podemos:

- Usar **la misma interfaz** para archivos y carpetas.
- Ejecutar operaciones de forma **uniforme**, sin importar si es un único objeto o un grupo.

Idea clave: "Tratar objetos individuales y compuestos como si fueran iguales."

La gran ventaja de esta solución es que no tienes que preocuparte por las clases concretas de los objetos que componen el árbol. No tienes que saber si un objeto es un producto simple o complejo. Puedes tratarlos a todos por igual a través de la interfaz común. Cuando invocas un método, los propios objetos pasan la solicitud a lo largo del árbol.

El uso del patrón Composite sólo tiene sentido cuando el modelo central de tu aplicación puede representarse en forma de árbol.

```
// Componente común
interface FileSystemItem {
    void show(String indent);
}

// Clase hoja: Archivo
class File implements FileSystemItem {
    private String name;

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

    @Override
    public void show(String indent) {
        System.out.println(indent + "- Archivo: " + name);
    }
}

// Clase compuesta: Carpeta
import java.util.ArrayList;
import java.util.List;

class Folder implements FileSystemItem {
    private String name;
    private List<FileSystemItem> items = new ArrayList<>();

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

    public void addItem(FileSystemItem item) {
        items.add(item);
    }

    @Override
    public void show(String indent) {
        System.out.println(indent + "+ Carpeta: " + name);
        for (FileSystemItem item : items) {
            item.show(indent + "    ");
```

```
        }
    }
}

// Ejemplo de uso
public class CompositeExample {
    public static void main(String[] args) {
        FileSystemItem archivo1 = new File("Documento.txt");
        FileSystemItem archivo2 = new File("Foto.png");
        FileSystemItem archivo3 = new File("Video.mp4");

        Folder carpeta1 = new Folder("Mis Documentos");
        carpeta1.addItem(archivo1);
        carpeta1.addItem(archivo2);

        Folder carpeta2 = new Folder("Multimedia");
        carpeta2.addItem(archivo3);
        carpeta2.addItem(carpeta1); // una carpeta dentro de otra

        carpeta2.show("");
    }
}
```

Decorator

También llamado: Decorador, Envoltorio, Wrapper

Propósito

Decorator es un patrón de diseño estructural que te permite añadir funcionalidades a objetos colocando estos objetos dentro de objetos encapsuladores especiales que contienen estas funcionalidades, es decir permite **añadir responsabilidades extra a un objeto de forma dinámica**, sin modificar su código original ni crear muchas subclases.

En palabras simples ws como "envolver" un objeto con capas adicionales que le agregan nuevas funcionalidades.

Una analogía muy usada es la del **café en una cafetería**:

- Tienes un café simple (objeto base).
- Puedes agregarle leche, azúcar, crema, etc.
- Cada "decorador" agrega algo más al café original, sin que haya que crear clases separadas como CafeConLeche, CafeConAzucarYCrema, etc.

Estructura

1. **Componente (interface/abstracta)** → Define la funcionalidad básica.
2. **Componente Concreto** → Implementación real del objeto base.
3. **Decorador abstracto** → Tiene una referencia al objeto Componente y delega llamadas.
4. **Decoradores concretos** → Agregan funcionalidades extra al objeto que envuelven.

```java
// 1. Componente
interface Cafe {
    String descripcion();
    double costo();
}

// 2. Componente concreto
class CafeSimple implements Cafe {
    @Override
    public String descripcion() {
        return "Café simple";
    }

    @Override
    public double costo() {
        return 20.0;
    }
}

// 3. Decorador abstracto
abstract class CafeDecorador implements Cafe {
    protected Cafe cafe;

    public CafeDecorador(Cafe cafe) {
        this.cafe = cafe;
    }

    @Override
    public String descripcion() {
        return cafe.descripcion();
    }

    @Override
    public double costo() {
        return cafe.costo();
    }
}

// 4. Decoradores concretos
class ConLeche extends CafeDecorador {
    public ConLeche(Cafe cafe) {
        super(cafe);
    }

    @Override
    public String descripcion() {
        return cafe.descripcion() + " + Leche";
    }

    @Override
    public double costo() {
        return cafe.costo() + 5.0;
    }
}
```

```
class ConAzucar extends CafeDecorador {
    public ConAzucar(Cafe cafe) {
        super(cafe);
    }

    @Override
    public String descripcion() {
        return cafe.descripcion() + " + Azúcar";
    }

    @Override
    public double costo() {
        return cafe.costo() + 2.0;
    }
}
```

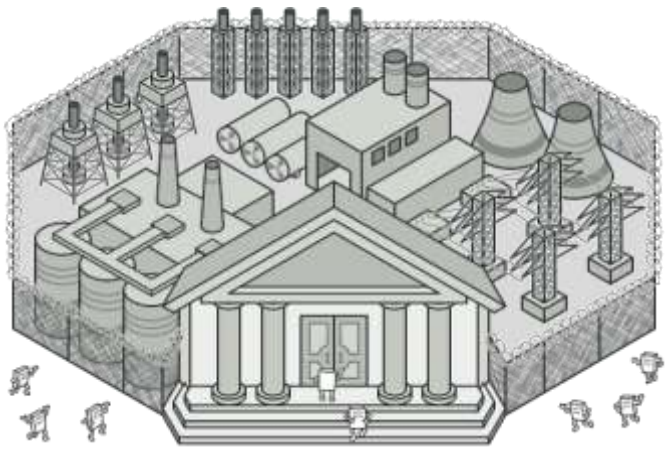
Con Decorator, puedes combinar los objetos como quieras, sin necesidad de hacer una subclase para cada combinación posible.

Facade

También llamado: Fachada

Propósito

Facade es un patrón de diseño estructural que proporciona una interfaz simplificada a una biblioteca, un framework o cualquier otro grupo complejo de clases.



El patrón **Facade** (fachada) se utiliza para **simplificar el acceso a sistemas complejos**. La idea es que, en lugar de que el cliente tenga que interactuar directamente con muchos objetos y clases, se le da una **única interfaz simplificada** que concentra las operaciones más comunes.

Metáfora de la vida real:

Imagina que vas a un **restaurante**.

- Podrías entrar a la cocina y pedirle al cocinero la comida, al ayudante que lave los platos, al mozo que te traiga los cubiertos, etc. (muchas interacciones, complejo).

- Pero en la práctica, **solo hablas con el mozo** (la *fachada*) y él se encarga de coordinar todo lo que pasa “detrás de escena”.

Beneficios

- Reduce la **complejidad** para el cliente.
- Centraliza el acceso a un subsistema complejo.
- Mantiene bajo acoplamiento: si cambia algo interno, el cliente no se ve afectado.

Ejemplo en Java – Patrón Facade

Supongamos que tenemos un **sistema de cine en casa** con varias clases:

- Reproductor de DVD
- Proyector
- Sistema de sonido

El cliente no debería tener que interactuar con cada uno de ellos. Mejor damos una **fachada** que los controle.

```
// Subsistemas
class DVDPlayer {
    public void on() { System.out.println("DVD Player encendido"); }
    public void play(String movie) { System.out.println("Reproduciendo película: " +
movie); }
    public void off() { System.out.println("DVD Player apagado"); }
}

class Projector {
    public void on() { System.out.println("Proyector encendido"); }
    public void off() { System.out.println("Proyector apagado"); }
}

class SoundSystem {
    public void on() { System.out.println("Sonido encendido"); }
    public void setVolume(int level) { System.out.println("Volumen ajustado a: " +
level); }
    public void off() { System.out.println("Sonido apagado"); }
}

// Facade
class HomeTheaterFacade {
    private DVDPlayer dvd;
    private Projector projector;
    private SoundSystem sound;

    public HomeTheaterFacade(DVDPlayer dvd, Projector projector, SoundSystem sound) {
        this.dvd = dvd;
        this.projector = projector;
```

```
        this.sound = sound;
    }

    public void watchMovie(String movie) {
        System.out.println("Preparando para ver una película...");
        dvd.on();
        projector.on();
        sound.on();
        sound.setVolume(10);
        dvd.play(movie);
    }

    public void endMovie() {
        System.out.println("Apagando el sistema de cine en casa...");
        dvd.off();
        projector.off();
        sound.off();
    }
}

// Cliente
public class FacadeExample {
    public static void main(String[] args) {
        DVDPlayer dvd = new DVDPlayer();
        Projector projector = new Projector();
        SoundSystem sound = new SoundSystem();

        HomeTheaterFacade homeTheater = new HomeTheaterFacade(dvd, projector, sound);

        homeTheater.watchMovie("El Señor de los Anillos");
        homeTheater.endMovie();
    }
}
```

Explicación del ejemplo

- El cliente **solo usa la clase HomeTheaterFacade**.
- La fachada se encarga de encender/apagar todos los dispositivos y coordinarlos.
- Si mañana cambiamos DVDPlayer por un NetflixApp, el cliente **no se entera**, solo sigue usando la fachada.

El **Facade** se usa muchísimo en librerías y frameworks.

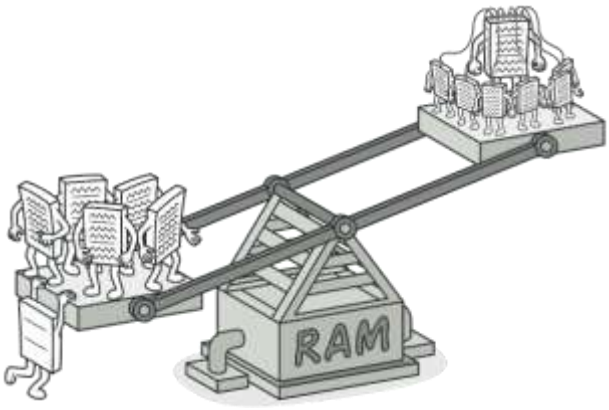
Por ejemplo: en lugar de que interactúes con todos los objetos de Hibernate, Spring, etc., ellos te dan métodos y clases de más alto nivel que simplifican el trabajo.

Flyweight

También llamado: Peso mosca, Peso ligero, Cache

Propósito

Flyweight es un patrón de diseño estructural que te permite mantener más objetos dentro de la cantidad disponible de RAM compartiendo las partes comunes del estado entre varios objetos en lugar de mantener toda la información en cada objeto.



Ejemplo de la vida real:

Imaginá un procesador de texto como Word. Cada vez que escribís la letra **“a”**, el programa **no crea un objeto nuevo para esa letra**, sino que reutiliza el mismo objeto que ya existe en memoria y solo cambia su posición.
De esa manera, si tenés un documento de 1.000 páginas con millones de letras, no te llena la memoria.

En resumen:

- **Estado Intrínseco (compartido):** datos que no cambian (ej: forma de la letra "a").
- **Estado Extrínseco (externo):** datos que cambian (ej: posición donde aparece la letra).

Ejemplo en Java

Vamos a simular un **editor de texto** donde muchas letras se reutilizan en memoria.

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

// Flyweight
interface Letra {
    void dibujar(int x, int y); // Estado extrínseco: posición
}

// Implementación concreta de la letra
class LetraConcreta implements Letra {
    private final char simbolo; // Estado intrínseco: la forma de la letra

    public LetraConcreta(char simbolo) {
        this.simbolo = simbolo;
        System.out.println("Creando objeto para la letra: " + simbolo);
    }

    @Override
```

```
    public void dibujar(int x, int y) {
        System.out.println("Dibujando letra '" + simbolo + "' en posición (" + x +
"," + y + ")");
    }
}

// Flyweight Factory
class FabricaDeLetras {
    private Map<Character, Letra> letras = new HashMap<>();

    public Letra obtenerLetra(char c) {
        Letra letra = letras.get(c);
        if (letra == null) {
            letra = new LetraConcreta(c);
            letras.put(c, letra);
        }
        return letra;
    }
}

// Cliente
public class FlyweightDemo {
    public static void main(String[] args) {
        FabricaDeLetras fabrica = new FabricaDeLetras();

        String texto = "hola hola";

        int x = 0;
        for (char c : texto.toCharArray()) {
            Letra letra = fabrica.obtenerLetra(c);
            letra.dibujar(x, 10); // Estado extrínseco: posición
            x += 10;
        }
    }
}
```

Explicación del ejemplo:

1. LetraConcreta representa una **letra única** (estado intrínseco: el símbolo 'h', 'o', 'l', 'a').
2. FabricaDeLetras guarda las letras ya creadas y **reutiliza** las mismas en lugar de crear nuevas.
3. El **cliente** (editor de texto) solo cambia la **posición** (estado extrínseco).
4. Cuando se ejecuta, verás que las letras "h", "o", "l", "a" se crean una sola vez, y luego se reutilizan.

En pocas palabras el **Flyweight** es ideal cuando tenemos **muchísimos objetos iguales** y queremos **ahorrar memoria** compartiendo los datos repetidos.

Flyweight y la inmutabilidad

Debido a que el mismo objeto flyweight puede utilizarse en distintos contextos, debes asegurarte de que su estado no se pueda modificar. Un objeto flyweight debe inicializar su estado una sola vez a través de

parámetros del constructor. No debe exponer ningún método *set* (modificador) o campo público a otros objetos.

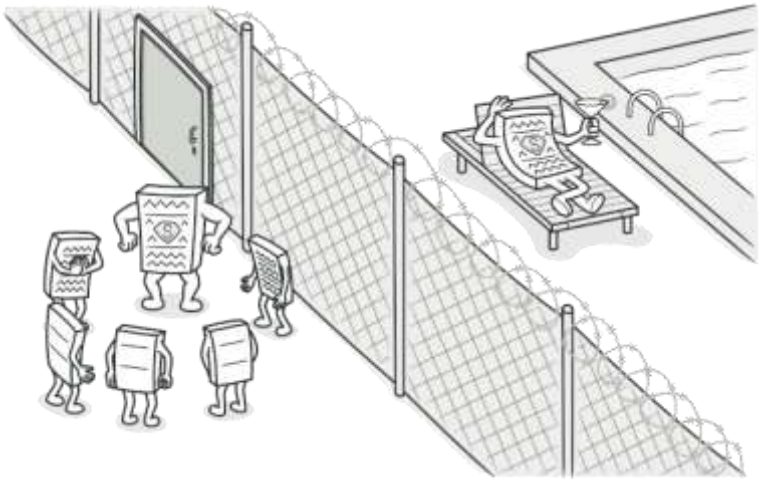
Proxy

Propósito

Proxy es un patrón de diseño estructural que te permite proporcionar un sustituto o marcador de posición para otro objeto. Un proxy controla el acceso al objeto original, permitiéndote hacer algo antes o después de que la solicitud llegue al objeto origina

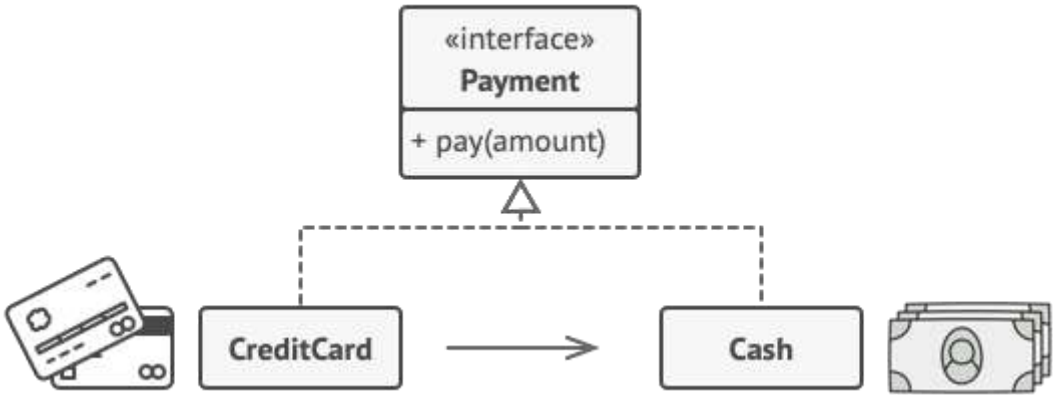
En palabras simples:

Es como un *portero* en un edificio. No entras directamente al departamento, primero hablas con el portero (proxy), y él decide si puedes pasar, si debe avisar al dueño, o si puede responderte sin molestarlo.



I.

Analogía en el mundo real



Las tarjetas de crédito pueden utilizarse para realizar pagos tanto como el efectivo.

Una tarjeta de crédito es un proxy de una cuenta bancaria, que, a su vez, es un proxy de un manojo de billetes. Ambos implementan la misma interfaz, por lo que pueden utilizarse para realizar un pago. El consumidor se siente genial porque no necesita llevar un montón de efectivo encima. El dueño de la tienda también está contento porque los ingresos de la transacción se añaden electrónicamente a la cuenta bancaria de la tienda sin el riesgo de perder el depósito o sufrir un robo de camino al banco.

Problema a resolver
Imagina que tienes un objeto enorme que consume una gran cantidad de recursos del sistema. Lo necesitas de vez en cuando, pero no siempre.

El patrón Proxy sugiere que crees una nueva clase proxy con la misma interfaz que un objeto de servicio original. Después actualizas tu aplicación para que pase el objeto proxy a todos los clientes del objeto original. Al recibir una solicitud de un cliente, el proxy crea un objeto de servicio real y le delega todo el trabajo.

Usos comunes del Proxy

1. **Control de acceso** → proteger el objeto real (ejemplo: permisos, autenticación).
2. **Carga diferida (Lazy Loading)** → crear el objeto real solo cuando es necesario.
3. **Registro o logging** → el proxy intercepta llamadas y registra información.
4. **Comunicación remota** → el proxy representa un objeto que está en otra máquina o servidor.

Ejemplo en Java: Proxy de un servicio de imágenes

Imaginemos que tenemos un servicio que carga imágenes pesadas desde internet.

No queremos cargar todas las imágenes de golpe, sino **solo cuando se usen** → el Proxy nos ayuda a diferir la carga.

```
// 1. Interfaz común
interface Imagen {
    void mostrar();
}

// 2. Objeto real (pesado de crear)
class ImagenReal implements Imagen {
    private String archivo;

    public ImagenReal(String archivo) {
        this.archivo = archivo;
        cargarDesdeDisco(); // simulamos algo pesado
    }

    private void cargarDesdeDisco() {
        System.out.println("Cargando imagen desde disco: " + archivo);
    }

    @Override
    public void mostrar() {
        System.out.println("Mostrando imagen: " + archivo);
    }
}

// 3. Proxy que retrasa la carga
class ImagenProxy implements Imagen {
    private String archivo;
```

```
    private ImagenReal imagenReal;

    public ImagenProxy(String archivo) {
        this.archivo = archivo;
    }

    @Override
    public void mostrar() {
        if (imagenReal == null) {
            imagenReal = new ImagenReal(archivo); // se carga solo la primera vez
        }
        imagenReal.mostrar();
    }
}

// 4. Cliente
public class Main {
    public static void main(String[] args) {
        Imagen img1 = new ImagenProxy("foto1.jpg");
        Imagen img2 = new ImagenProxy("foto2.jpg");

        // todavía no se cargaron
        System.out.println("Las imágenes aún no están cargadas");

        // ahora se carga y muestra al usarla
        img1.mostrar();

        // esta ya se carga solo cuando se usa
        img2.mostrar();
    }
}
```

El **Proxy** nos muestra cómo un objeto puede ser un intermediario inteligente para controlar el acceso a otro.