# Patrones de diseño estructurales

**Juan Pablo Dios Castro Vanegas**

**Código:1077842737**

**Emili García Bermúdez**

**Código:1091202597**

**Sarita Londoño Perdomo**

**Código: 1091884459**

**Profesor:Jose Wilson Capera**

**Universidad del Quindío**

**Facultad de ingeniería**

**Ingeniería en sistemas y computación**

**Sistemas de Información**

**Armenia, Quindío 2024**

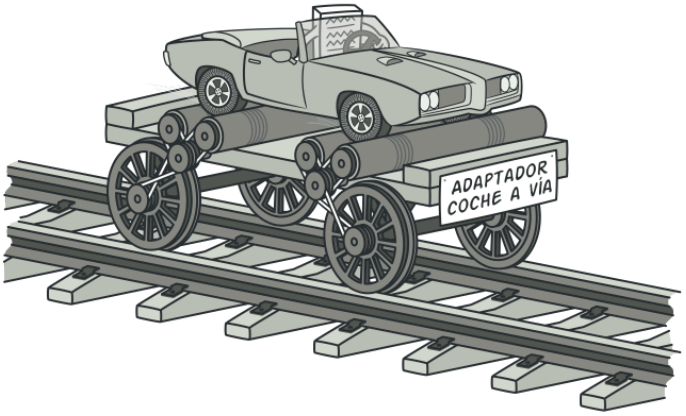
# Patrón Adapter

**Propósito:**

El patrón Adapter se utiliza para permitir que interfaces incompatibles trabajen juntas. En otras palabras, convierte la interfaz de una clase en otra interfaz que el cliente espera. Esto se logra creando una clase intermediaria que conecta la interfaz del cliente con la interfaz de la clase existente.

En resumen, el patrón Adapter se usa para:

Hacer que dos interfaces incompatibles sean compatibles.
Convertir una interfaz de un tipo a otro.
Permitir la reutilización de código antiguo en nuevos sistemas.




**Aplicaciones Típicas:**

1. .Utiliza la clase adaptadora cuando quieras usar una clase existente, pero cuya interfaz no sea compatible con el resto del código: El patrón Adapter te permite crear una clase intermedia que sirva como traductora entre tu código y una clase heredada, una clase de un tercero o cualquier otra clase con una interfaz extraña.

2. Utiliza el patrón cuando quieras reutilizar varias subclases existentes que carezcan de alguna funcionalidad común que no pueda añadirse a la superclase: Puedes extender cada subclase y colocar la funcionalidad que falta, dentro de las nuevas clases hijas. No obstante, deberás duplicar el código en todas estas nuevas clases, lo cual huele muy mal.

**Otras aplicaciones:**

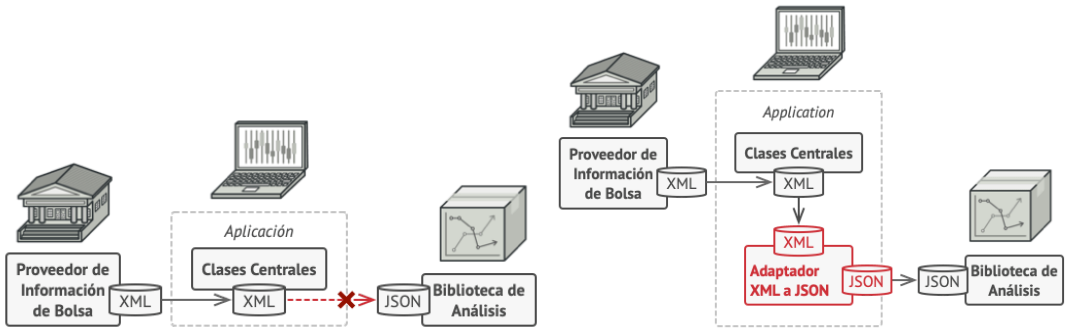
-Adaptación de interfaces de usuario: para que sean accesibles a usuarios con diferentes discapacidades.
-Adaptación de datos: para que sean compatibles con diferentes formatos o sistemas.
-Adaptación de protocolos de red: para que dos sistemas que usan diferentes protocolos puedan comunicarse entre sí.

**Ventajas de usar el patrón Adapter:**

- Aumenta la flexibilidad y la modularidad del código.
- Facilita la integración de diferentes componentes.
- Puede ayudar a evitar la duplicación de código.

**Desventajas de usar el patrón Adapter:**

- Puede agregar una capa de complejidad al código.
- Puede disminuir el rendimiento del sistema.
- Puede ser difícil de mantener a largo plazo.




**Problema:**

En Europa, los enchufes eléctricos tienen dos clavijas redondas, mientras que en América, los enchufes tienen dos clavijas planas y una clavija de tierra en forma de U. Un turista europeo que viaja a América no puede conectar sus dispositivos electrónicos a los enchufes americanos.

**Abstracción:**

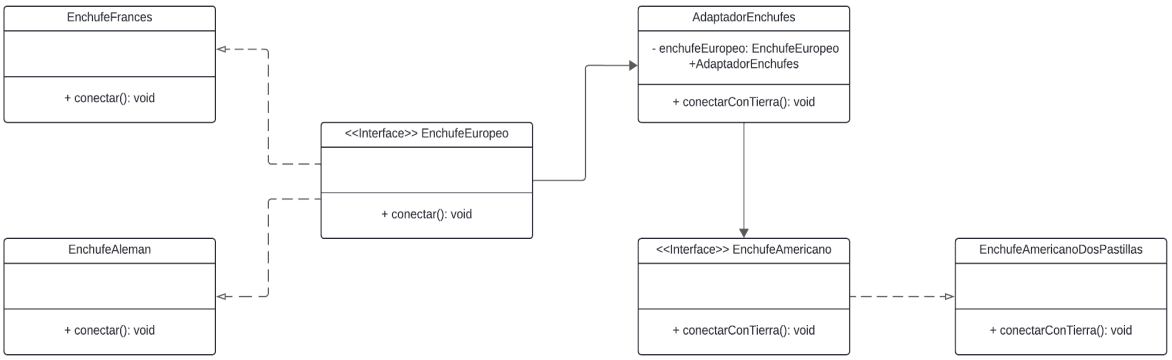
1. **Interfaces:**

**EnchufeEuropeo**: Conectar un enchufe europeo a una toma de corriente.
**EnchufeAmericano:** Conectar un enchufe americano con tierra a una toma de corriente.

2. **Clases:**

**EnchufeAleman, EnchufeFrances:** Implementan EnchufeEuropeo para enchufes específicos.
**EnchufeAmericanoDosPatillas:** Implementa EnchufeAmericano para enchufes americanos de dos patillas.
**AdaptadorEnchufeEuropeoAmericano:** Adapta un enchufe europeo a un enchufe americano con tierra.

**Diagrama de clases:**




**Implementacion del codigo**:

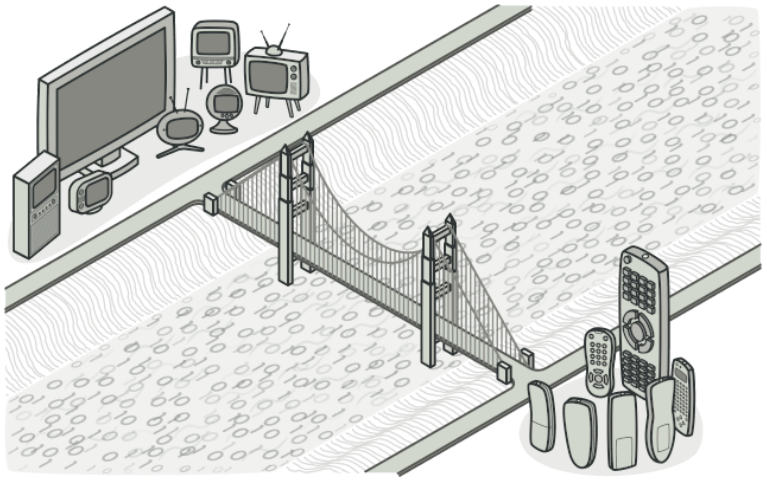
```
package co.edu.uniquindio.poo;

//La interfaz EnchufeEuropeo define un método conectar().
public interface EnchufeEuropeo {
    void conectar();

}
//La clase EnchufeAleman implementa la interfaz EnchufeEuropeo y proporciona una implementación para el método conectar().
public class EnchufeAleman implements EnchufeEuropeo{

    @Override
    public void conectar() {
        System.out.println("Enchufe aleman conectado");

    }
}
//La clase EnchufeFrances implementa la interfaz EnchufeEuropeo y proporciona una implementación para el método conectar().
public class EnchufeFrances implements EnchufeEuropeo {

    @Override
```

```
    public void conectar() {
        System.out.println("Enchufe frances conectado");

    }


}
//La interfaz EnchufeAmericano define un método conectarConTierra().
public interface EnchufeAmericano {
    void conectarConTierra();

}

//La clase EnchufeAmericanoDosPatillas implementa la interfaz EnchufeAmericano y proporciona una implementación para el método conectarConTierra().
public class EnchufeAmericanoDosPastillas implements EnchufeAmericano{

    @Override
    public void conectarConTierra() {
        System.out.println("Enchufe americano de dos pastillas conectado con tierra");

    }

}
public class AdaptadorEnchufes {
    private EnchufeEuropeo enchufeEuropeo;
    //Esta clase tiene un constructor que toma un objeto EnchufeEuropeo como argumento.
    public AdaptadorEnchufes(EnchufeEuropeo enchufeEuropeo) {
        this.enchufeEuropeo = enchufeEuropeo;
    }
     //Este método primero imprime un mensaje que indica que se está adaptando el enchufe europeo. Luego, llama al método conectar() del objeto EnchufeEuropeo. Finalmente, imprime un mensaje que indica que el enchufe europeo se ha conectado al enchufe americano con tierra.
    public void conectarConTierra() {
        System.out.println("Adaptando enchufe europeo...");
        enchufeEuropeo.conectar();
        System.out.println("Enchufe europeo conectado a enchufe americano con tierra.");
     }
    }
//La clase App crea instancias de EnchufeAleman, EnchufeFrances y AdaptadorEnchufes. Luego, llama al método conectarConTierra() del adaptador para cada tipo de enchufe europeo.
public class App {
    public static void main(String[] args) {
        EnchufeAleman enchufeAleman = new EnchufeAleman();
    AdaptadorEnchufes adaptador = new AdaptadorEnchufes (enchufeAleman);
    adaptador.conectarConTierra();

    EnchufeFrances enchufeFrances = new EnchufeFrances();
```

```
    adaptador = new AdaptadorEnchufes(enchufeFrances);
    adaptador.conectarConTierra();
  }
}
```

**Conclusión:** El adaptador AdaptadorEnchufes permite que un objeto EnchufeEuropeo se use con un objeto EnchufeAmericano, El adaptador convierte la forma y el tamaño del enchufe europeo para que coincidan con las del enchufe americano,Este es un ejemplo simple de cómo se puede usar el patrón Adapter en Java

## Patrón Bridge (Puente)



El patrón Bridge separa una abstracción de su implementación, permitiendo que ambas varíen independientemente. Esto significa que la lógica de la aplicación (abstracción) puede ser independiente de la forma en que se implementa (implementación).

**Problema que resuelve:**

El patrón Bridge resuelve el problema de acoplamiento rígido entre la abstracción y la implementación. En muchos casos, es necesario cambiar la implementación de una clase sin afectar a la lógica que la utiliza. Sin embargo, si la abstracción y la implementación están estrechamente acopladas, esto no es posible.

**Solución:**

El patrón Bridge introduce una capa de indirección entre la abstracción y la implementación. La abstracción se define en términos de una interfaz, y la implementación se define como una clase que implementa esa interfaz. Esto permite que la abstracción y la implementación se modifiquen independientemente una de la otra.

**Aplicaciones Típicas**

**1. Aplicaciones multiplataforma:**

El patrón Bridge se puede utilizar para desarrollar aplicaciones multiplataforma con una interfaz de usuario específica para cada plataforma. La abstracción define la interfaz de usuario general, mientras que la implementación define la interfaz de usuario específica para cada plataforma.

**2. Sistemas con diferentes tecnologías de almacenamiento:**

El patrón Bridge se puede utilizar para desarrollar sistemas que pueden funcionar con diferentes tecnologías de almacenamiento. La abstracción define la lógica de acceso a datos, mientras que la implementación define la forma en que se accede a los datos en una tecnología de almacenamiento específica.

**Ejemplo de implementación:**

**Contexto:**

Imagina una aplicación de software para la gestión de una biblioteca. La aplicación necesita manejar dos tipos de entidades:

- **Libros:** Son los recursos físicos que se prestan a los usuarios.
- **Recursos digitales:** Son libros electrónicos, audiolibros y otros materiales accesibles en línea.

La aplicación necesita ofrecer funcionalidades comunes para ambos tipos de entidades, como:

- Búsqueda por título, autor o ISBN.
- Visualización de la información detallada de la entidad.
- Préstamo a usuarios.
- Devolución por parte de los usuarios.

Sin embargo, existen diferencias importantes en la forma en que se gestionan los libros físicos y los recursos digitales:

- **Préstamo de libros físicos:** Se requiere un registro físico del préstamo, como el marcado del libro como prestado en el sistema y la entrega física del libro al usuario.
- **Préstamo de recursos digitales:** El préstamo se realiza de forma online, generalmente mediante la descarga del archivo o la activación de una licencia temporal.

**Problema:**

Si se implementa la funcionalidad de forma monolítica, mezclando la gestión de libros físicos y recursos digitales, el código se vuelve complejo y difícil de mantener. Además, sería difícil agregar nuevos tipos de entidades en el futuro.

**Solución:**

- La clase Producto define las características comunes a todos los productos.
- Las clases ProductoSimple y ProductoCompuesto implementan las características específicas para cada tipo de producto.

- La clase ProductoCompuesto puede contener una lista de otros productos, creando una estructura jerárquica.

**Código:**

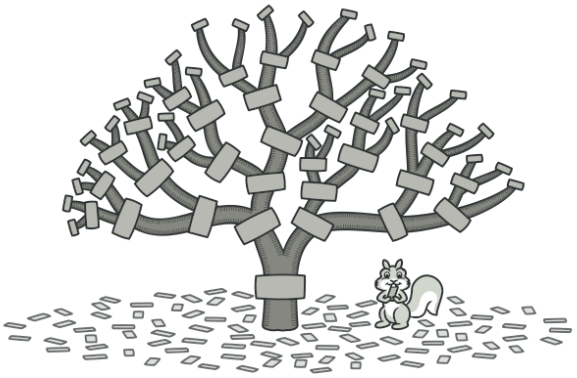
```
public interface Entidad {
    String getNombre();

    void buscar(String criterio);

    void mostrarInformacion();

    void prestar(Usuario usuario);

    void devolver(Usuario usuario);
}

public class Libro implements Entidad {

    private String nombre;
    private String autor;
    private String ISBN;
    private int stock;
    private Usuario prestadoA;

    public Libro(String nombre, String autor, String ISBN) {
        this.nombre = nombre;
        this.autor = autor;
        this.ISBN = ISBN;
        this.stock = 0;
        this.prestadoA = null;
    }

    @Override
    public String getNombre() {
        return nombre;
    }

    @Override
    public void buscar(String criterio) {
        // Implementar la búsqueda específica para libros
    }

    @Override
    public void mostrarInformacion() {
        System.out.println("**Libro:** " + nombre);
        System.out.println("Autor: " + autor);
        System.out.println("ISBN: " + ISBN);
        System.out.println("Stock: " + stock);
```

```
        if (prestadoA != null) {
            System.out.println("Prestado a: " + prestadoA.getNombre());
        }
    }

    @Override
    public void prestar(Usuario usuario) {
        if (stock > 0) {
            stock--;
            prestadoA = usuario;
            System.out.println("Libro prestado a " + usuario.getNombre());
        } else {
            System.out.println("No hay stock disponible del libro " + nombre);
        }
    }

    @Override
    public void devolver(Usuario usuario) {
        if (prestadoA != null && prestadoA.equals(usuario)) {
            stock++;
            prestadoA = null;
            System.out.println("Libro devuelto por " + usuario.getNombre());
        } else {
                System.out.println("El usuario " + usuario.getNombre() + " no tiene el libro " +
nombre + " prestado");
        }
    }
}

public class RecursoDigital implements Entidad {

    private String nombre;
    private String autor;
    private String formato;
    private int licenciasDisponibles;

    public RecursoDigital(String nombre, String autor, String formato) {
        this.nombre = nombre;
        this.autor = autor;
        this.formato = formato;
        this.licenciasDisponibles = 0;
    }

    @Override
    public String getNombre() {
        return nombre;
    }

    @Override
    public void buscar(String criterio) {
```

```
        // Implementar la búsqueda específica para recursos digitales
    }

    @Override
    public void mostrarInformacion() {
        System.out.println("**Recurso digital:** " + nombre);
        System.out.println("Autor: " + autor);
        System.out.println("Formato: " + formato);
        System.out.println("Licencias disponibles: " + licenciasDisponibles);
    }

    @Override
    public void prestar(Usuario usuario) {
        if (licenciasDisponibles > 0) {
            licenciasDisponibles--;
            System.out.println("Recurso digital prestado a " + usuario.getNombre());
        } else {
            System.out.println("No hay licencias disponibles del recurso digital " + nombre);
        }
    }

    @Override
    public void devolver(Usuario usuario) {
        licenciasDisponibles++;
        System.out.println("Recurso digital devuelto por " + usuario.getNombre());
    }
}

public class Usuario {

    private String nombre;

    public Usuario(String nombre) {
        this.nombre = nombre;
    }

    public String getNombre() {
        return nombre;
    }
}

public class Main {

    public static void main(String[] args) {
                Entidad libro = new Libro("El Principito", "Antoine de Saint-Exupéry",
"978-84-204-4701-7");
        Entidad recursoDigital = new RecursoDigital("El Hobbit", "J.R.R. Tolkien", "EPUB");

        libro.mostrarInformacion();
        recursoDigital.mostrarInformacion();
```

```
Usuario usuario1 = new Usuario("Usuario1");
libro.prestar(usuario1);
recurso
```

## Patrón Composite (Compuesto)



**Propósito**
El patrón Composite (Compuesto) permite componer objetos en estructuras de árbol, de modo que se puedan tratar de forma uniforme tanto los objetos individuales como las estructuras compuestas.

**Problema que resuelve:**

El patrón Composite resuelve el problema de manejar jerarquías de objetos de forma eficiente y flexible. En muchos casos, las aplicaciones necesitan trabajar con objetos que pueden tener una estructura jerárquica, como un árbol de archivos o una organización. Sin embargo, el código para manejar estas jerarquías puede ser complejo y difícil de mantener.

**Solución:**

El patrón Composite introduce dos tipos de objetos:

Componentes: Son los objetos que forman parte de la estructura jerárquica.
Contenedores: Son los objetos que agrupan a otros componentes.
Los contenedores pueden contener tanto componentes como otros contenedores, lo que permite crear estructuras de árbol de cualquier profundidad.

El patrón Composite define una interfaz común para los componentes y los contenedores, lo que permite que se traten de forma uniforme. Esto significa que el código que opera con componentes no necesita saber si está trabajando con un componente individual o con una estructura compuesta.

**Aplicaciones Típicas**
1. Árboles de archivos:

El patrón Composite se puede utilizar para representar un árbol de archivos. Los archivos son los componentes y las carpetas son los contenedores.

2. Organizaciones:

El patrón Composite se puede utilizar para representar una organización. Los empleados son los componentes y los departamentos son los contenedores.

**Ejemplo de implementación:**

**Contexto:**

Imagina una aplicación para la gestión de inventario de una tienda. La tienda vende diferentes tipos de productos:

- Productos simples: Son productos individuales, como una camisa o un libro.
- Productos compuestos: Son conjuntos de productos que se venden como un único paquete, como un kit de herramientas o una cesta de regalo.

**La aplicación necesita:**

- Mostrar una lista de todos los productos disponibles en la tienda.
- Calcular el precio total de un pedido, que puede incluir varios productos simples y/o compuestos.
- Gestionar el stock de productos, tanto de los productos simples como de los productos compuestos.

**Problema:**

Si se implementa la gestión de productos de forma monolítica, mezclando los productos simples y los productos compuestos, el código se vuelve complejo y difícil de mantener. Además, sería difícil agregar nuevos tipos de productos en el futuro.

**Solución:**

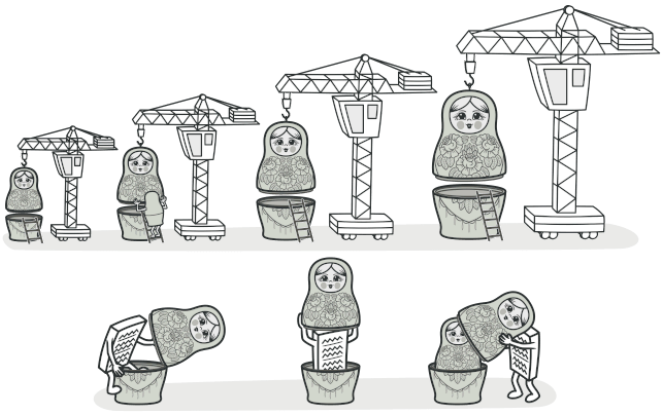
- La interfaz Producto define las características comunes a todos los productos.
- Las clases ProductoSimple y ProductoCompuesto implementan las características específicas para cada tipo de producto.
- La clase ProductoCompuesto puede contener una lista de otros productos, creando una estructura jerárquica.

```
public interface Producto {

    String getNombre();

    double getPrecio();
```

```
    int getStock();

}

public class ProductoSimple implements Producto {

    private String nombre;
    private double precio;
    private int stock;

    public ProductoSimple(String nombre, double precio, int stock) {
        this.nombre = nombre;
        this.precio = precio;
        this.stock = stock;
    }

    @Override
    public String getNombre() {
        return nombre;
    }

    @Override
    public double getPrecio() {
        return precio;
    }

    @Override
    public int getStock() {
        return stock;
    }

}

public class ProductoCompuesto implements Producto {

    private String nombre;
    private List<Producto> productos;

    public ProductoCompuesto(String nombre) {
        this.nombre = nombre;
        this.productos = new ArrayList<>();
    }

    public void agregarProducto(Producto producto) {
        productos.add(producto);
    }

    @Override
    public String getNombre() {
        return nombre;
```

```
    }

    @Override
    public double getPrecio() {
        double precioTotal = 0;
        for (Producto producto : productos) {
            precioTotal += producto.getPrecio();
        }
        return precioTotal;
    }

    @Override
    public int getStock() {
        int stockTotal = 0;
        for (Producto producto : productos) {
            stockTotal += producto.getStock();
        }
        return stockTotal;
    }

}

public class Main {

    public static void main(String[] args) {

        Producto productoSimple = new ProductoSimple("Camisa", 20.0, 10);

                ProductoCompuesto productoCompuesto = new ProductoCompuesto("Kit de herramientas");
        productoCompuesto.agregarProducto(new ProductoSimple("Martillo", 10.0, 5));
        productoCompuesto.agregarProducto(new ProductoSimple("Destornillador", 5.0, 8));

        System.out.println("Precio del producto simple: " + productoSimple.getPrecio());
                        System.out.println("Precio del producto compuesto: " + productoCompuesto.getPrecio());

    }

}
```

**Decorator (Decorador)**



El patrón Decorador añade funcionalidades a un objeto de forma dinámica sin modificar su clase.

**Problema que resuelve:**

El patrón Decorador resuelve el problema de extender la funcionalidad de un objeto sin subclasificarlo. En muchos casos, es necesario agregar funcionalidades adicionales a un objeto existente. Sin embargo, la subclasificación puede llevar a una jerarquía de clases grande y difícil de mantener.

**Solución:**

El patrón Decorador introduce la idea de un objeto decorador. Un decorador es un objeto que se envuelve alrededor de otro objeto y añade funcionalidades adicionales a su comportamiento. El decorador delega la mayoría de las llamadas al objeto que está decorando, pero puede interceptar o modificar el comportamiento de esas llamadas para añadir su propia funcionalidad.

**Aplicaciones Típicas**
1. Personalización de widgets:

El patrón Decorador se puede utilizar para personalizar la apariencia o el comportamiento de los widgets de una interfaz gráfica de usuario (GUI).

2. Registro de llamadas a métodos:

El patrón Decorador se puede utilizar para registrar las llamadas a los métodos de un objeto.

3. Validación de datos:

El patrón Decorador se puede utilizar para validar los datos introducidos por el usuario antes de pasarlos a un objeto.

**Ejemplo de Código**

**Contexto:**

Imagina una aplicación para la gestión de pedidos de una cafetería. La aplicación permite a los clientes crear pedidos con diferentes tipos de café:

- Café solo: Es un café sin ningún aditivo.
- Café con leche: Es un café con leche añadida.
- Café con leche y azúcar: Es un café con leche y azúcar añadidos.

La aplicación necesita:

- Mostrar una lista de todos los tipos de café disponibles.
- Calcular el precio total de un pedido, que puede incluir varios cafés de diferentes tipos.
- Permitir a los clientes personalizar sus pedidos con diferentes opciones, como la cantidad de leche o azúcar.

Problema:

Si se implementa la gestión de cafés de forma monolítica, mezclando los diferentes tipos de café y las opciones de personalización, el código se vuelve complejo y difícil de mantener. Además, sería difícil agregar nuevas opciones de personalización en el futuro.

Solución:

- Se define una clase abstracta Cafe que representa las características comunes a todos los cafés (nombre, precio).
- CafeSolo: Representa un café sin ningún aditivo.
- CafeConLeche: Representa un café con leche añadida.

Además, se definen dos clases decoradoras que heredan de la clase Cafe:

- Leche: Agrega leche a un café.
- Azúcar: Agrega azúcar a un café.

Ejemplo de código:

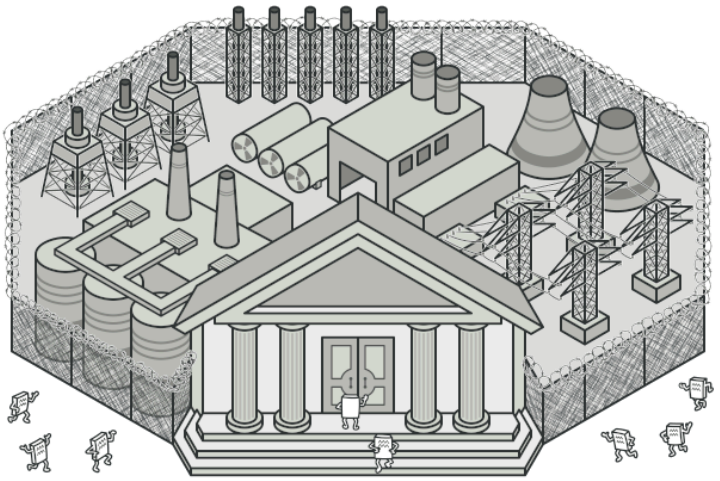
```
public interface Cafe {

    String getNombre();

    double getPrecio();

}

public class CafeSolo implements Cafe {

    private final String NOMBRE = "Café solo";
    private final double PRECIO = 1.5;

    @Override
```

```
    public String getNombre() {
        return NOMBRE;
    }

    @Override
    public double getPrecio() {
        return PRECIO;
    }

}

public class CafeConLeche extends Cafe {

    private final String NOMBRE = "Café con leche";
    private final double PRECIO = 2.0;

    private Cafe cafe;

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

    @Override
    public String getNombre() {
        return cafe.getNombre() + " con leche";
    }

    @Override
    public double getPrecio() {
        return cafe.getPrecio() + PRECIO;
    }

}

public abstract class DecoradorCafe implements Cafe {

    protected Cafe cafe;

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

    @Override
    public abstract String getNombre();

    @Override
    public abstract double getPrecio();

}
```

```
public class Leche extends DecoradorCafe {

    private final String NOMBRE = "Leche";
    private final double PRECIO = 0.5;

    public Leche(Cafe cafe) {
        super(cafe);
    }

    @Override
    public String getNombre() {
        return cafe.getNombre() + " con " + NOMBRE;
    }

    @Override
    public double getPrecio() {
        return cafe.getPrecio() + PRECIO;
    }

}

public class Azucar extends DecoradorCafe {

    private final String NOMBRE = "Azúcar";
    private final double PRECIO = 0.2;

    public Azucar(Cafe cafe) {
        super(cafe);
    }

    @Override
    public String getNombre() {
        return cafe.getNombre() + " con " + NOMBRE;
    }

    @Override
    public double getPrecio() {
        return cafe.getPrecio() + PRECIO;
    }

}

public class Main {

    public static void main(String[] args) {

        Cafe cafeSolo = new CafeSolo();
        Cafe cafeConLeche = new CafeConLeche(cafeSolo);
        Cafe cafeConLecheYAzucar = new Azucar(cafeConLeche);
```

```
        System.out.println("Precio del café solo: " + cafeSolo.getPrecio());
        System.out.println("Precio del café con leche: " + cafeConLeche.getPrecio());
                        System.out.println("Precio del café con leche y azúcar: " +
cafeConLecheYAzucar.getPrecio());

    }

}
```

**Facade (Fachada)**



**Propósito**

El patrón Fachada proporciona una interfaz única y simplificada para un conjunto de subclases.

**Problema que resuelve:**

El patrón Fachada resuelve el problema de la complejidad de usar un conjunto de clases relacionadas. En muchos casos, las aplicaciones necesitan trabajar con un conjunto de clases que están relacionadas entre sí y que ofrecen una funcionalidad compleja. Esto puede dificultar el uso de estas clases, ya que el código necesita conocer los detalles de implementación de cada una de ellas.

**Solución:**

El patrón Fachada introduce una clase fachada que actúa como un intermediario entre el cliente y el conjunto de subclases. La clase fachada proporciona una interfaz única y simplificada que oculta los detalles de implementación de las subclases.

**Aplicaciones Típicas**

1. Sistemas complejos:

El patrón Fachada se puede utilizar para simplificar la interfaz de un sistema complejo, proporcionando una única entrada para un conjunto de operaciones relacionadas.

2. Bibliotecas de terceros:

El patrón Fachada se puede utilizar para integrar una biblioteca de terceros con una aplicación existente,ocultando los detalles de implementación de la biblioteca.

**Ejemplo de Código**

## Contexto:

Imagina una aplicación para la gestión de una biblioteca. La aplicación permite a los usuarios realizar diferentes tareas:

- Buscar libros: Los usuarios pueden buscar libros por título, autor o ISBN.
- Prestar libros: Los usuarios pueden prestar libros de la biblioteca.
- Devolver libros: Los usuarios pueden devolver los libros que han prestado.
- Renovar préstamos: Los usuarios pueden renovar los préstamos de los libros que han prestado.
- Pagar multas: Los usuarios pueden pagar las multas por retraso en la devolución de libros.

**La aplicación necesita:**

- Proporcionar una interfaz sencilla e intuitiva para los usuarios.
- Ocultar la complejidad del sistema a los usuarios.
- Facilitar la integración con otros sistemas.

**Problema:**

Si se implementa la gestión de las tareas de forma monolítica, el código se vuelve complejo y difícil de mantener. Además, sería difícil agregar nuevas tareas en el futuro.

**Solución:**

El patrón Facade puede ser utilizado para resolver este problema. Se define una clase FacadeBiblioteca que actúa como un único punto de acceso a la funcionalidad de la aplicación.

La clase FacadeBiblioteca encapsula la complejidad del sistema y proporciona una interfaz sencilla e intuitiva para los usuarios.

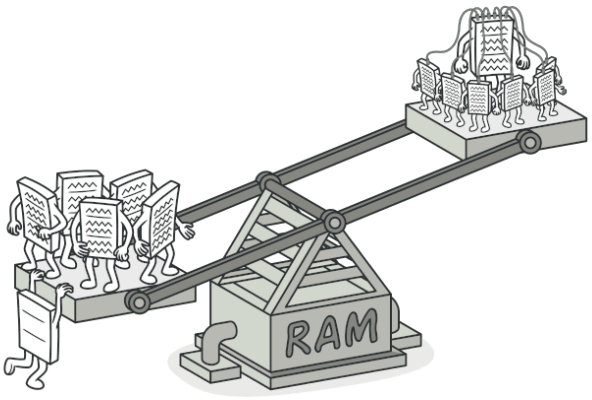
**Ejemplo de código:**

```
public class FacadeBiblioteca {
```

```
private SistemaBusqueda sistemaBusqueda;
private SistemaPrestamo sistemaPrestamo;
private SistemaRenovacion sistemaRenovacion;
private SistemaPagoMultas sistemaPagoMultas;
public FacadeBiblioteca() {
    this.sistemaBusqueda = new SistemaBusqueda();
    this.sistemaPrestamo = new SistemaPrestamo();
    this.sistemaRenovacion = new SistemaRenovacion();
    this.sistemaPagoMultas = new SistemaPagoMultas();
}
public List<Libro> buscarLibros(String criterio) {
    return sistemaBusqueda.buscarLibros(criterio);
}
public void prestarLibro(Usuario usuario, Libro libro) {
    sistemaPrestamo.prestarLibro(usuario, libro);
}
public void devolverLibro(Usuario usuario, Libro libro) {
    sistemaPrestamo.devolverLibro(usuario, libro);
}
public void renovarPrestamo(Usuario usuario, Libro libro) {
    sistemaRenovacion.renovarPrestamo(usuario, libro);
}
public void pagarMulta(Usuario usuario, Libro libro) {
    sistemaPagoMultas.pagarMulta(usuario, libro);
```

```
    }
}
public class Main {
    public static void main(String[] args) {
        FacadeBiblioteca facadeBiblioteca = new FacadeBiblioteca();
        List<Libro> libros = facadeBiblioteca.buscarLibros("El Principito");
        for (Libro libro : libros) {
            System.out.println(libro.getNombre());
        }
        Usuario usuario = new Usuario("Usuario1");
        Libro libro = libros.get(0);
        facadeBiblioteca.prestarLibro(usuario, libro);
        // ...
        facadeBiblioteca.devolverLibro(usuario, libro);
    }
}
```

**Patrón Flyweight (Peso Ligero)**



**Propósito**

El patrón Flyweight reduce el uso de memoria al compartir objetos que son intrínsecamente iguales.

**Problema que resuelve:**

El patrón Flyweight resuelve el problema del consumo excesivo de memoria en aplicaciones que utilizan una gran cantidad de objetos similares. En muchos casos, las aplicaciones necesitan crear una gran cantidad de objetos que son muy similares entre sí, lo que puede consumir una gran cantidad de memoria.

**Solución:**

El patrón Flyweight introduce la idea de un objeto flyweight. Un objeto flyweight es un objeto que se puede compartir entre diferentes clientes. El patrón Flyweight define una fábrica que se encarga de crear y almacenar los objetos flyweight. Cuando un cliente necesita un objeto, la fábrica busca un objeto flyweight existente que sea compatible con las necesidades del cliente. Si no se encuentra un objeto flyweight existente, la fábrica crea uno nuevo.

**Aplicaciones Típicas**

1. Editores de texto:

El patrón Flyweight se puede utilizar para reducir el uso de memoria en un editor de texto al compartir los caracteres que se muestran en la pantalla.

2. Interfaces gráficas de usuario:

El patrón Flyweight se puede utilizar para reducir el uso de memoria en una interfaz gráfica de usuario al compartir los iconos y otros elementos gráficos.

**Ejemplo de Código**

**Contexto:**

Imagina una aplicación para la gestión de un juego de rol online. La aplicación necesita crear y gestionar miles de personajes no jugadores (PNJs) que pueblan el mundo del juego.

Cada PNJ tiene diferentes características:

- Nombre: El nombre del PNJ.
- Raza: La raza del PNJ (humano, elfo, enano, etc.).
- Clase: La clase del PNJ (guerrero, mago, ladrón, etc.).
- Nivel: El nivel del PNJ.
- Apariencia: La apariencia del PNJ (pelo, ojos, ropa, etc.).

**Problema:**

Si se crea un objeto para cada PNJ, el consumo de memoria y la carga de procesamiento

serán demasiado altos, especialmente en un juego con miles de PNJs.

**Solución:**

El patrón Flyweight puede ser utilizado para resolver este problema. Se define una clase PNJFlyweight que almacena las características comunes a todos los PNJs de la misma raza y clase.

Luego, se crean objetos PNJConcreto que heredan de PNJFlyweight y que almacenan las características específicas de cada PNJ, como el nombre y la apariencia.

De esta manera, se reduce el número de objetos que se necesitan crear, lo que reduce el consumo de memoria y la carga de procesamiento.

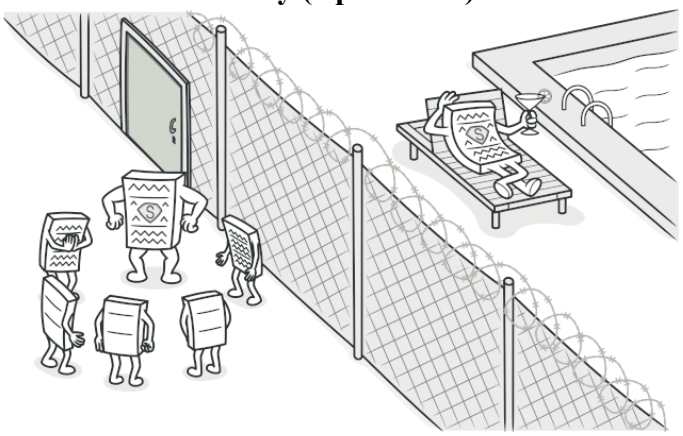
**Ejemplo de código:**

```
public interface PNJ {

    String getNombre();

    Raza getRaza();

    Clase getClase();

    int getNivel();

    void atacar(PNJ objetivo);

    void hablar(Jugador jugador);

}

public enum Raza {

    HUMANO, ELFO, ENANO, ORCO

}

public enum Clase {

    GUERRERO, MAGO, LADRON, SANADOR

}

public abstract class PNJFlyweight implements PNJ {

    private Raza raza;
    private Clase clase;
    private int nivel;
```

```
    public PNJFlyweight(Raza raza, Clase clase, int nivel) {
        this.raza = raza;
        this.clase = clase;
        this.nivel = nivel;
    }

    @Override
    public Raza getRaza() {
        return raza;
    }

    @Override
    public Clase getClase() {
        return clase;
    }

    @Override
    public int getNivel() {
        return nivel;
    }

    public abstract void atacar(PNJ objetivo);

    public abstract void hablar(Jugador jugador);

}

public class PNJConcreto extends PNJFlyweight {

    private String nombre;
    private Apariencia apariencia;

    public PNJConcreto(PNJFlyweight flyweight, String nombre, Apariencia apariencia) {
        super(flyweight.getRaza(), flyweight.getClase(), flyweight.getNivel());
        this.nombre = nombre;
        this.apariencia = apariencia;
    }

    @Override
    public String getNombre() {
        return nombre;
    }

    @Override
    public void atacar(PNJ objetivo) {
        // Implementación específica del ataque para la raza y clase del PNJ
    }

    @Override
    public void hablar(Jugador jugador) {
```

```
        // Implementación específica del diálogo para la raza y clase del PNJ
    }

}

public class FabricaPNJ {

    private Map<Raza, Map<Clase, PNJFlyweight>> flyweights = new HashMap<>();

    public PNJ crearPNJ(Raza raza, Clase clase, String nombre, Apariencia apariencia) {
        PNJFlyweight flyweight = obtenerFlyweight(raza, clase);
        return new PNJConcreto(flyweight, nombre, apariencia);
    }

    private PNJFlyweight obtenerFlyweight(Raza raza, Clase clase) {
        Map<Clase, PNJFlyweight> flyweightsClase = flyweights.get(raza);
        if (flyweightsClase == null) {
            flyweightsClase = new HashMap<>();
            flyweights.put(raza, flyweightsClase);
        }

        PNJFlyweight flyweight = flyweightsClase.get(clase);
        if (flyweight == null) {
            flyweight = new PNJFlyweight(raza, clase, 1);
            flyweightsClase.put(clase, flyweight);
        }

        return flyweight;
    }

}

public class Main {

    public static void main(String[] args) {

        FabricaPNJ fabricaPNJ = new Fabrica
}
```

**Proxy (Apoderado)**



**Propósito**
El patrón Proxy proporciona un intermediario entre un objeto y el cliente que lo usa.

**Problema que resuelve:**

El patrón Proxy resuelve el problema del acoplamiento directo entre el objeto y el cliente. En muchos casos, es necesario que un objeto pueda ser utilizado por diferentes clientes. Sin embargo, si el objeto está directamente acoplado a los clientes, esto puede dificultar la modificación del objeto o el uso del objeto en diferentes contextos.

**Solución:**

El patrón Proxy introduce un objeto proxy que actúa como intermediario entre el objeto real y el cliente. El objeto proxy proporciona una interfaz similar al objeto real, pero puede ocultar la implementación del objeto real y realizar otras tareas adicionales.

**Aplicaciones Típicas**
1. Control de acceso:

El patrón Proxy se puede utilizar para controlar el acceso a un objeto. El objeto proxy puede verificar si el cliente tiene permiso para acceder al objeto real antes de concederle acceso.

2. Caché:

El patrón Proxy se puede utilizar para implementar un caché. El objeto proxy puede almacenar en caché los resultados de las llamadas al objeto real para mejorar el rendimiento.

**Ejemplo de Código**

**Contexto:**

Imagina una aplicación para la gestión de una biblioteca online. La aplicación permite a los usuarios acceder a una gran cantidad de recursos digitales, como libros electrónicos, audiolibros y vídeos.

**La aplicación necesita:**

Controlar el acceso a los recursos digitales.
Limitar el número de usuarios que pueden acceder a un recurso digital al mismo tiempo.
Registrar el uso de los recursos digitales.
Problema:

Si se implementa el control de acceso directamente en los recursos digitales, el código se vuelve complejo y difícil de mantener. Además, sería difícil agregar nuevas funcionalidades, como el registro del uso.

**Solución:**

El patrón Proxy puede ser utilizado para resolver este problema. Se define una clase ProxyRecursoDigital que actúa como un intermediario entre el usuario y el recurso digital.

El ProxyRecursoDigital controla el acceso al recurso digital, limita el número de usuarios que pueden acceder al mismo tiempo y registra el uso del recurso.

**Ejemplo de código:**

```
public interface RecursoDigital {

    String getNombre();

    void acceder(Usuario usuario);

}

public class RecursoDigitalConcreto implements RecursoDigital {

    private String nombre;

    public RecursoDigitalConcreto(String nombre) {
        this.nombre = nombre;
    }

    @Override
    public String getNombre() {
        return nombre;
    }

    @Override
    public void acceder(Usuario usuario) {
        // Implementación del acceso al recurso digital
    }

}
```

```
public class ProxyRecursoDigital implements RecursoDigital {

    private RecursoDigitalConcreto recursoDigital;
    private int numeroUsuariosActuales;

    public ProxyRecursoDigital(RecursoDigitalConcreto recursoDigital) {
        this.recursoDigital = recursoDigital;
        this.numeroUsuariosActuales = 0;
    }

    @Override
    public String getNombre() {
        return recursoDigital.getNombre();
    }

    @Override
    public void acceder(Usuario usuario) {
        if (numeroUsuariosActuales < MAX_USUARIOS_SIMULTANEOS) {
            numeroUsuariosActuales++;
            recursoDigital.acceder(usuario);
            // Registrar el uso del recurso digital
        } else {
            // Mostrar un mensaje de error al usuario
        }
    }

}

public class Main {

    public static void main(String[] args) {

                RecursoDigitalConcreto libroElectronico = new RecursoDigitalConcreto("El Principito");
                                    ProxyRecursoDigital proxyLibroElectronico = new ProxyRecursoDigital(libroElectronico);

        Usuario usuario1 = new Usuario("Usuario1");
        proxyLibroElectronico.acceder(usuario1);

        Usuario usuario2 = new Usuario("Usuario2");
        proxyLibroElectronico.acceder(usuario2);

    }

}
```

**Bibliografía:**

***Web Content Accessibility Guidelines (WCAG) 2.1.*** (2023b, septiembre 21). https://www.w3.org/TR/WCAG21/

**Initiative, W. W. A. (s. f.).** *Home*. Web Accessibility Initiative (WAI). https://www.w3.org/WAI/

***Adapter*. (s. f.).** https://refactoring.guru/es/design-patterns/adapter

***Fundación ONCE*. (s. f.).** Fundación ONCE. https://www.fundacionONCE.es/

***Patrones de diseño / Design patterns*. (n.d.).** **https://refactoring.guru/es/design-patterns**