UNIVERSIDAD
Popular del cesar

Ingeniería de Sistemas

ESPECIALIZACION EN INGENIERIA DE SOFTWARE

MODULO PATRONES DE DISEÑO DE SOFTWARE

EL DOCENTE

JAIRO FRANCISCO SEOANES LEON

jairoseoanes@unicesar.edu.co
(300) 600 06 70

Educación formal

- ✓ **Ingeniero de sistemas**, Universidad Popular del Cesar sede Valledupar, Feb 2002 – Jun 2009.
- ✓ **MsC en Ingeniería de Sistemas y Computación**, Universidad Nacional de Colombia, Bogotá, Feb 2011 – Mar 2015
- ✓ **PhD Ciencia, Tecnologia e innovación, Urbe, Venezuela, Mayo 2024**

Formación complementaria

- ✓ **AWS Academy Graduate -** AWS Academy Cloud Foundations, 2022
  https://www.credly.com/go/p3Uwht36
- ✓ **Associate Cloud Engineer Path -** Google Cloud Academy, 2022
  https://www.cloudskillsboost.google/public_profiles/c7e7936c-3e37-4bad-b822-74d40c49d0db
- ✓ **Fundamentos De Programación Con Énfasis En Cloud Computing –** AWS Academy y Misión Tic 2022
- ✓ **Google Cloud Computing Foundations –** Google Academy, 2022
- ✓ **Aplicación de cloud: retos y oportunidades de mejora para las empresas de software gestionando la computación en la nube –** Fedesoft, 2023
- ✓ **Desarrollo De Aplicaciones Web En Angular, Para El Nivel Frontend** – Universidad EAFIT, 2023
- ✓ **Microsoft Scrum Foundations –** Intelligent Training - MinTic , 2023

Experiencia profesional

- ✓ **Docente Universitario**, Universidad Popular del Cesar sede Valledupar, marzo del 2013.
- ✓ **Técnico de Sistemas Grado 11**, Rama judicial Seccional Cesar, SRPA Valledupar, Junio del 2009

MODULO DE PATRONES DE DISEÑO DE SOFTWARE

Unidad 1

Fundamentos avanzados de diseño orientado a objetos

Unidad 2

Patrones de diseño creacionales

Unidad 3

Patrones de diseño estructurales

Unidad 4

Patrones de diseño de comportamiento

Unidad 5

Antipatrones y refactorizacion

Cierre Curso

MODULO DE PATRONES DE DISEÑO DE SOFTWARE

Unidad 1

Fundamentos avanzados de diseño orientado a objetos

→

Unidad 2

Patrones de Diseño creacionales

→

Unidad 3

Patrones de diseño estructurales

↓

Unidad 4

Patrones de Diseño de Comportamiento

→

Unidad 5

Antipátrones y refactorización

→

Cierre Curso

UNIVERSIDAD Popular del cesar

Unidad 3. Patrones estructurales

3.1 Adapter
3.2 Bridge
3.3 Composite
3.4 Decorator
3.5 Facade
3.6 Flyweight
3.7 Proxy

Adapter

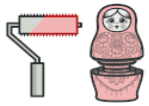
Permite la colaboración entre objetos con interfaces incompatibles.

Bridge

Permite dividir una clase grande o un grupo de clases estrechamente relacionadas, en dos jerarquías separadas (abstracción e implementación) que pueden desarrollarse independientemente la una de la otra.

Composite

Permite componer objetos en estructuras de árbol y trabajar con esas estructuras como si fueran objetos individuales.

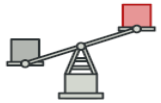


Decorator

Permite añadir funcionalidades a objetos colocando estos objetos dentro de objetos encapsuladores especiales que contienen estas funcionalidades.

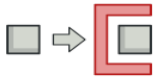
Facade

Proporciona una interfaz simplificada a una biblioteca, un framework o cualquier otro grupo complejo de clases.



Flyweight

Permite mantener más objetos dentro de la cantidad disponible de memoria RAM compartiendo las partes comunes del estado entre varios objetos en lugar de mantener toda la información en cada objeto.

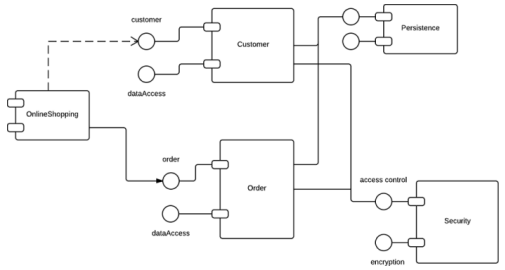


Proxy

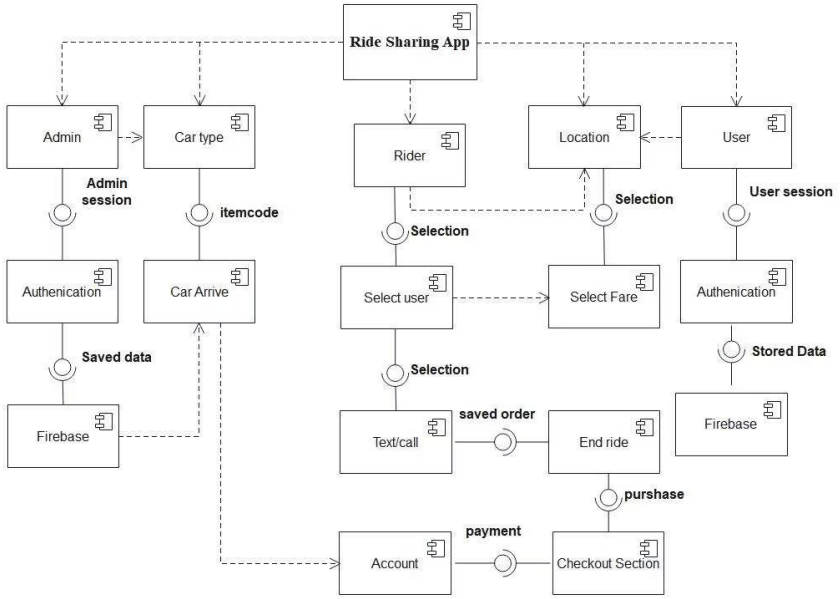
Permite proporcionar un sustituto o marcador de posición para otro objeto. Un proxy controla el acceso al objeto original, permitiéndote hacer algo antes o después de que la solicitud llegue al objeto original.

Patrones Estructurales - Generales

En el desarrollo de software, **la estructura del sistema** determina en gran medida su mantenibilidad, extensibilidad y reutilización. **A medida que los sistemas crecen en complejidad**, se hace evidente la necesidad de organizar las clases y objetos de manera que sus relaciones no generen dependencias rígidas, acoplamientos innecesarios o estructuras difíciles de escalar.



Patrones de diseño estructurales como soluciones reutilizables y probadas para mejorar la organización interna del software.



UNIVERSIDAD Popular del cesar

Patrones Estructurales - Generales

- ✓ **Los patrones estructurales** son un tipo de patrones de diseño que se enfocan en cómo se componen las clases y los objetos de manera eficiente y flexible para formar estructuras más grandes.
- ✓ En vez de combinar implementaciones (estructuras estáticas), los patrones estructurales de objetos describen formas de componer objetos para obtener nueva funcionalidad a partir de las interfaces.
- ✓ Estos patrones promueven la separación de responsabilidades, la delegación, la encapsulación de variaciones estructurales y el uso eficiente de recursos.

Principales objetivos

Reducción del acoplamiento entre clases o capas del sistema.

Aumento de la cohesión interna dentro de los módulos

Facilitación de la extensión y reutilización de componentes.

Mejora de la legibilidad y organización del sistema

Soporte para interoperabilidad entre sistemas heterogéneos o legados.

Transformación de sistemas altamente acoplados hacia arquitecturas modulares y flexibles mediante la aplicación estratégica de estos patrones.

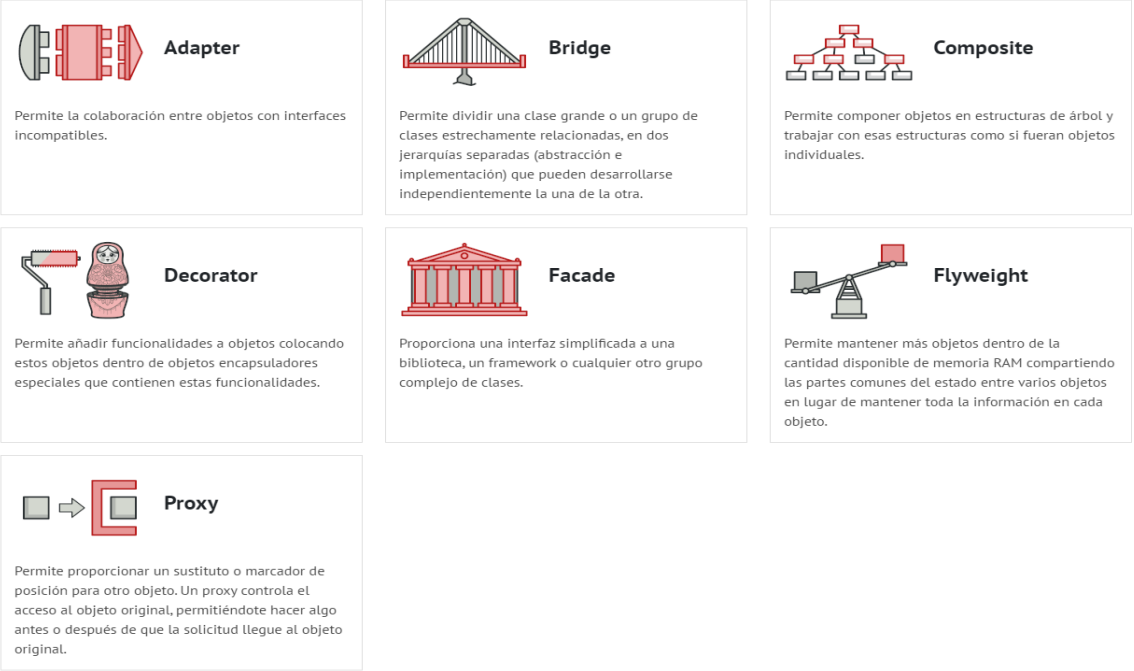
Particularmente relevantes en arquitecturas modernas como **Microservicios, Clean Architecture, Hexagonal Architecture o sistemas distribuidos**

UNIVERSIDAD Popular del cesar

Patrones Estructurales - Generales

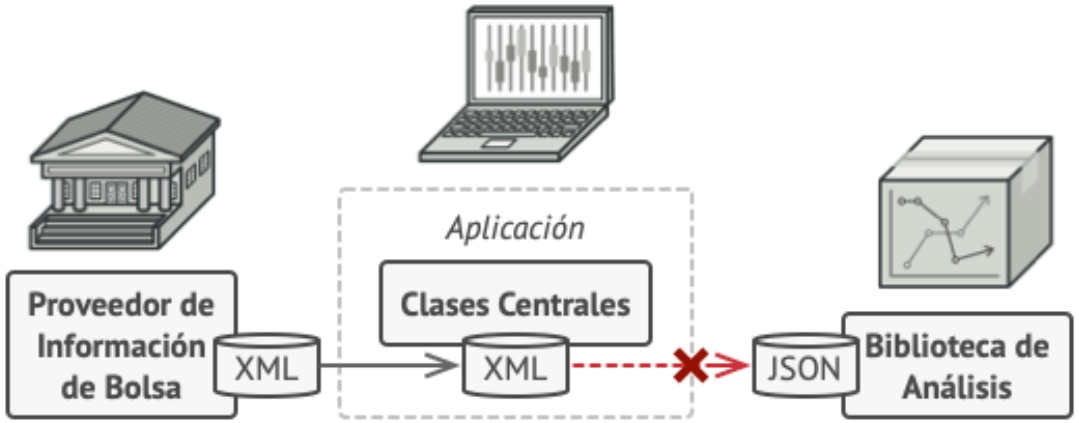
Patrones estructurales más relevantes

- **Adapter**: Permite la compatibilidad entre interfaces distintas.
- **Bridge**: Desacopla una abstracción de su implementación para que ambas puedan variar independientemente.
- **Composite**: Permite tratar objetos individuales y compuestos de manera uniforme.
- **Decorator**: Añade funcionalidad a objetos de manera dinámica y flexible.
- **Facade**: Proporciona una interfaz simplificada a un conjunto complejo de subsistemas.
- **Flyweight**: Minimiza el uso de memoria compartiendo datos comunes entre múltiples objetos.
- **Proxy**: Controla el acceso a otro objeto, añadiendo una capa intermedia.



Patrones Estructurales – Adapter

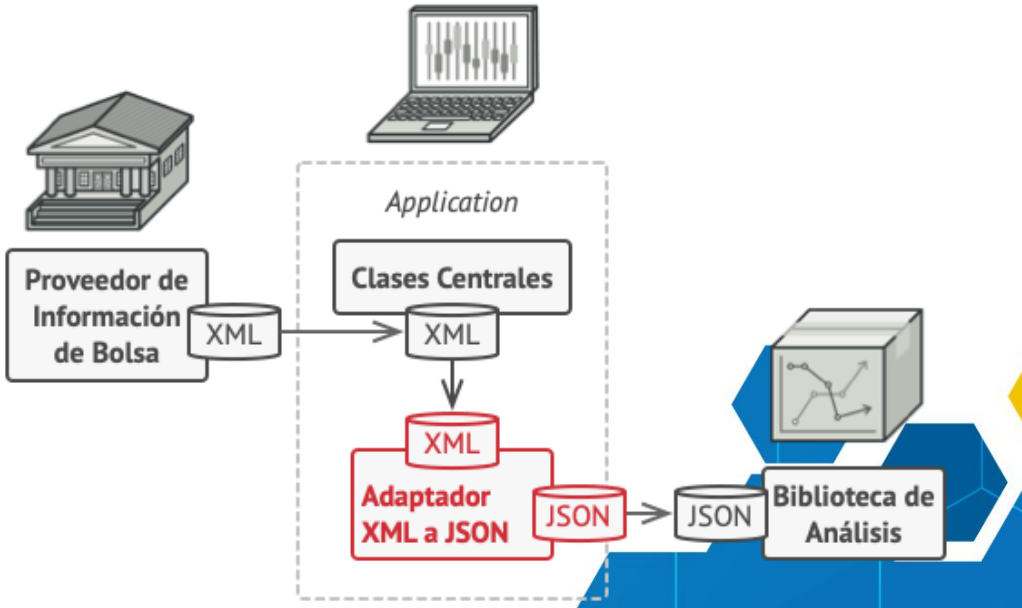
El patrón **Adapter** (Adaptador) permite que interfaces incompatibles trabajen juntas, actuando como un traductor entre dos interfaces que no pueden comunicarse directamente. Su propósito principal es resolver problemas de incompatibilidad sin modificar el código existente.



Problemas que Resuelve

En proyectos reales, es común encontrarse con **clases ya existentes** —quizás de **bibliotecas de terceros**, **módulos legados** o **APIs externas**— que no implementan la interfaz que necesita nuestro sistema.

Refactorizar estas clases no siempre es viable, por razones técnicas o contractuales. Aquí es donde Adapter es valioso.

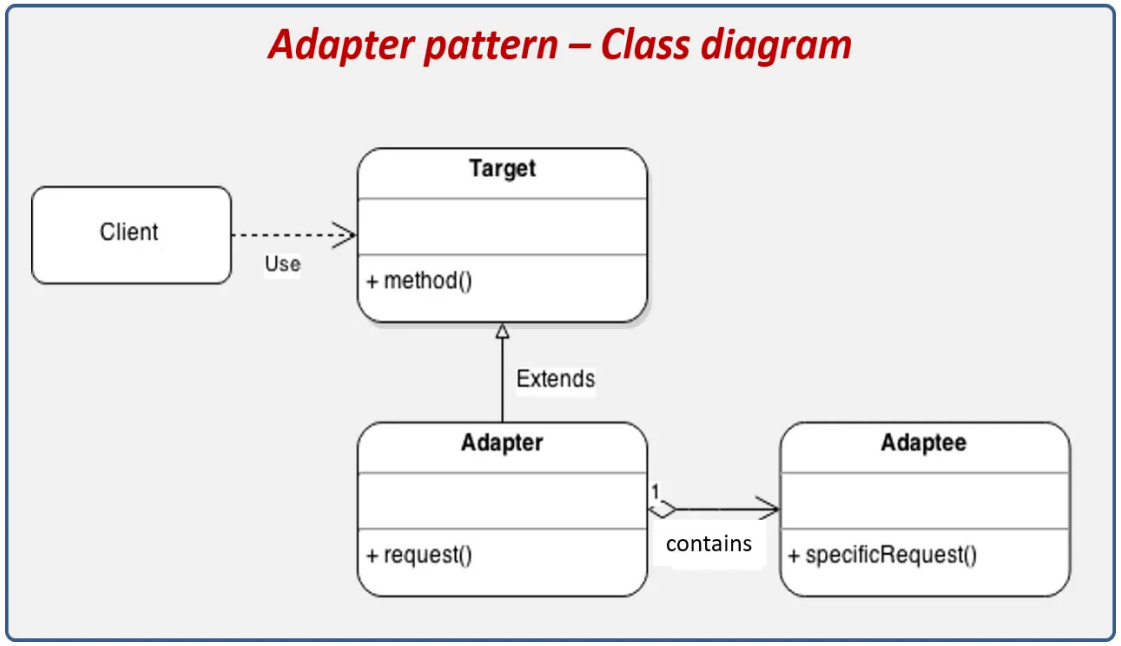


UNIVERSIDAD Popular del cesar

Patrones estructurales – Adapter

Target: Interface que nos permitirá homogenizar la forma de trabajar con las interfaces incompatibles, esta interface es utilizada para crear los Adapter.

Client: Actor que interactúa con el Adapter.



Adaptee: Representa la clase con interface incompatible.

Adapter: Representa la implementación del Target, el cual tiene la responsabilidad de mediar entre el Client y el Adaptee. Oculta la forma de comunicarse con el Adaptee.

UNIVERSIDAD Popular del cesar

Patrones estructurales – Adapter

Cuándo Implementar Adapter

Quieres usar una clase existente, pero su interfaz no es compatible con la que necesitas.

Deseas reutilizar una clase sin modificarla (por ejemplo, por ser parte de una biblioteca externa o por razones de compatibilidad hacia atrás).

Necesitas integrar múltiples sistemas heterogéneos que deben interoperar a través de interfaces unificadas.

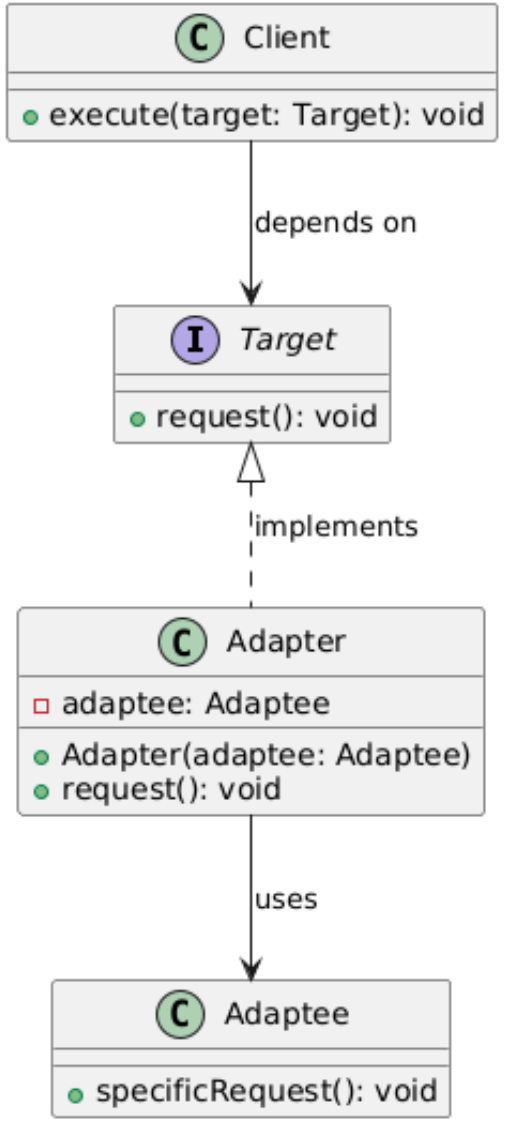
Casos de Uso Comunes y Realistas

Adaptadores JDBC en Java para convertir una interfaz estándar a las implementaciones específicas de Oracle, PostgreSQL, etc.

Integración de servicios SOAP en sistemas RESTful modernos.

En frameworks como Spring, el patrón Adapter se utiliza extensamente en capas de integración con múltiples tecnologías (por ejemplo, HandlerAdapter, HttpMessageConverter).

Adapter



UNIVERSIDAD Popular del cesar

Patrones estructurales – Adapter

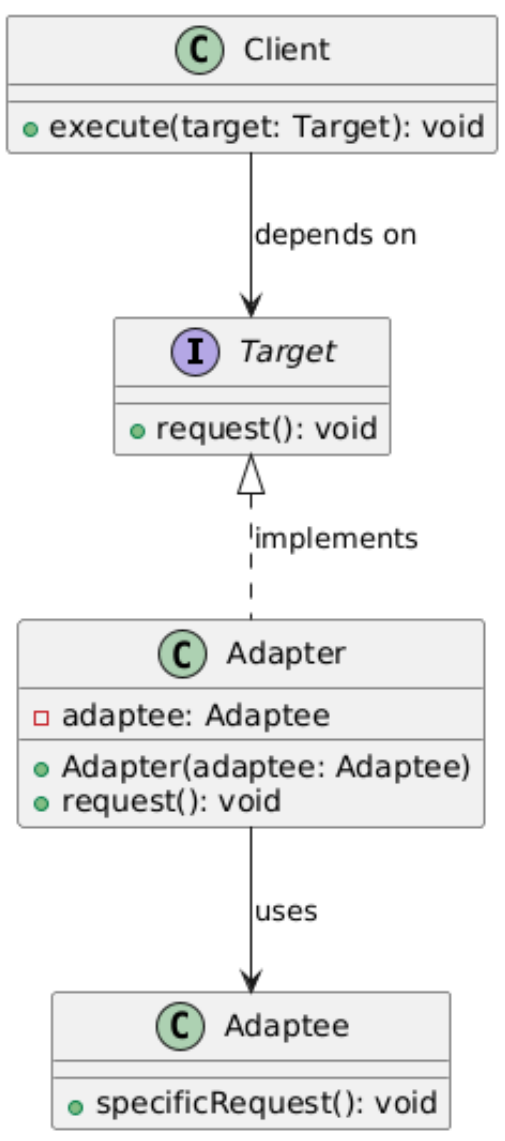
Ventajas

- **Reutilización de código** existente sin modificarlo.
- **Desacopla el cliente** de la implementación específica del adaptado.
- **Facilita la interoperabilidad** entre componentes con interfaces distintas.
- **Cumple con los principios SOLID** (especialmente SRP y OCP).

Desventajas

- Puede **aumentar la complejidad** si hay muchos adaptadores que mantener.
- Puede introducir **sobrecarga innecesaria** si se abusa de su uso en lugar de refactorizar correctamente.
- Oculta detalles de implementación, lo cual a veces puede dificultar la depuración.

Adapter



UNIVERSIDAD Popular del cesar

Patrones estructurales – Adapter

Un sistema moderno espera imprimir documentos usando esta interfaz

```
public interface DocumentPrinter {
    void print(String content);
}
```

Sin embargo, tienes una biblioteca de impresión antigua que no puedes modificar, y su método no es compatible con la interfaz actual

```
public class LegacyPrinter {
    public void sendToPrinter(byte[] data) {
        System.out.println("Imprimiendo datos binarios: " + new String(data));
    }
}
```

Adaptador

```
public class LegacyPrinterAdapter implements DocumentPrinter {
    private LegacyPrinter legacyPrinter;

    public LegacyPrinterAdapter(LegacyPrinter legacyPrinter) {
        this.legacyPrinter = legacyPrinter;
    }

    @Override
    public void print(String content) {
        // Convertir el String en bytes para adaptarse al método antiguo
        byte[] data = content.getBytes();
        legacyPrinter.sendToPrinter(data);
    }
}
```

Aplicaciones en Sistemas Modulares

El **Adapter** es fundamental para la integración de módulos externos, APIs de terceros y sistemas legacy.

Facilita la migración gradual de sistemas y permite la reutilización de componentes existentes sin modificaciones.

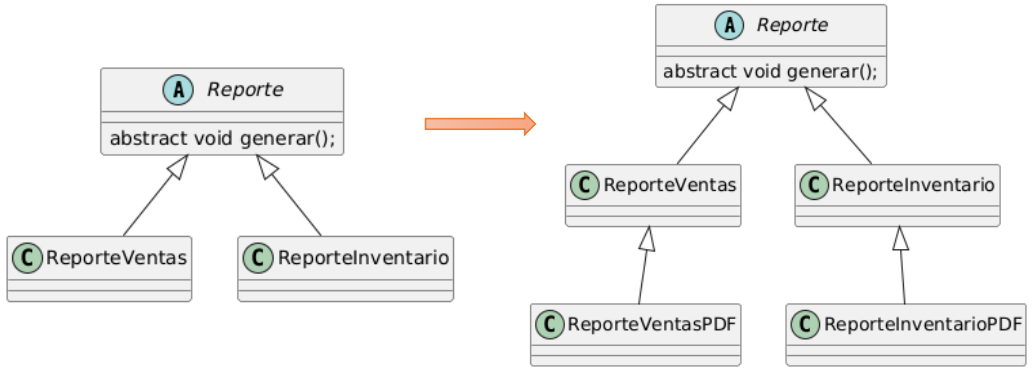
Cliente

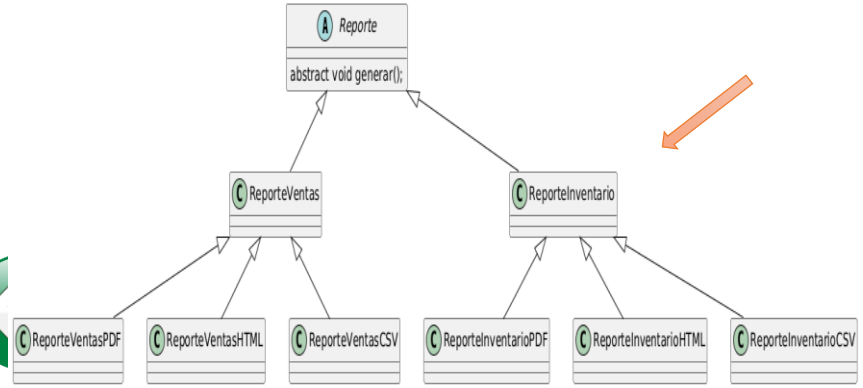
```
public class Main {
    public static void main(String[] args) {
        // Creamos el adaptador con la impresora antigua
        LegacyPrinter legacy = new LegacyPrinter();
        DocumentPrinter printer = new LegacyPrinterAdapter(legacy);

        // Imprimimos usando la interfaz moderna
        printer.print("Hola mundo desde la nueva interfaz");
    }
}
```

UNIVERSIDAD Popular del cesar

Patrones Estructurales – Bridge

El patrón de diseño **Bridge** separa una abstracción de su implementación, permitiendo que ambas varíen independientemente. Este patrón es esencial para evitar la explosión combinatoria de clases y promover la extensibilidad.





Problemas que Resuelve

La herencia liga la implementación a la abstracción de forma permanente:

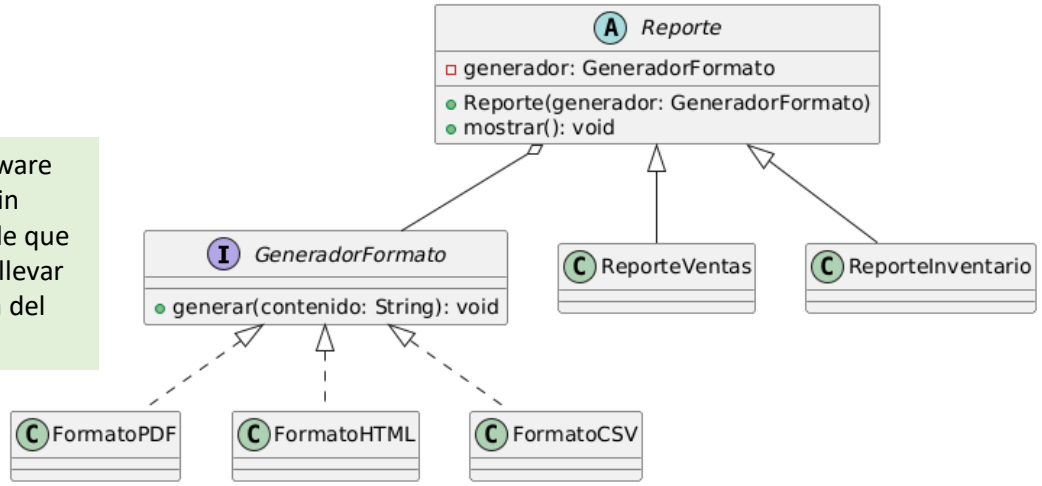
Pérdida de flexibilidad: no puedes reutilizar la funcionalidad en de las implementaciones

Combinación explosiva de clases: Por cada nueva variante en la abstracción o implementación, debes crear una **nueva clase combinada.**

Violación de principios de diseño:

- **OCP**: No puedes extender fácilmente sin modificar clases existentes.
- **SRP**: Cada clase termina teniendo múltiples razones para cambiar.

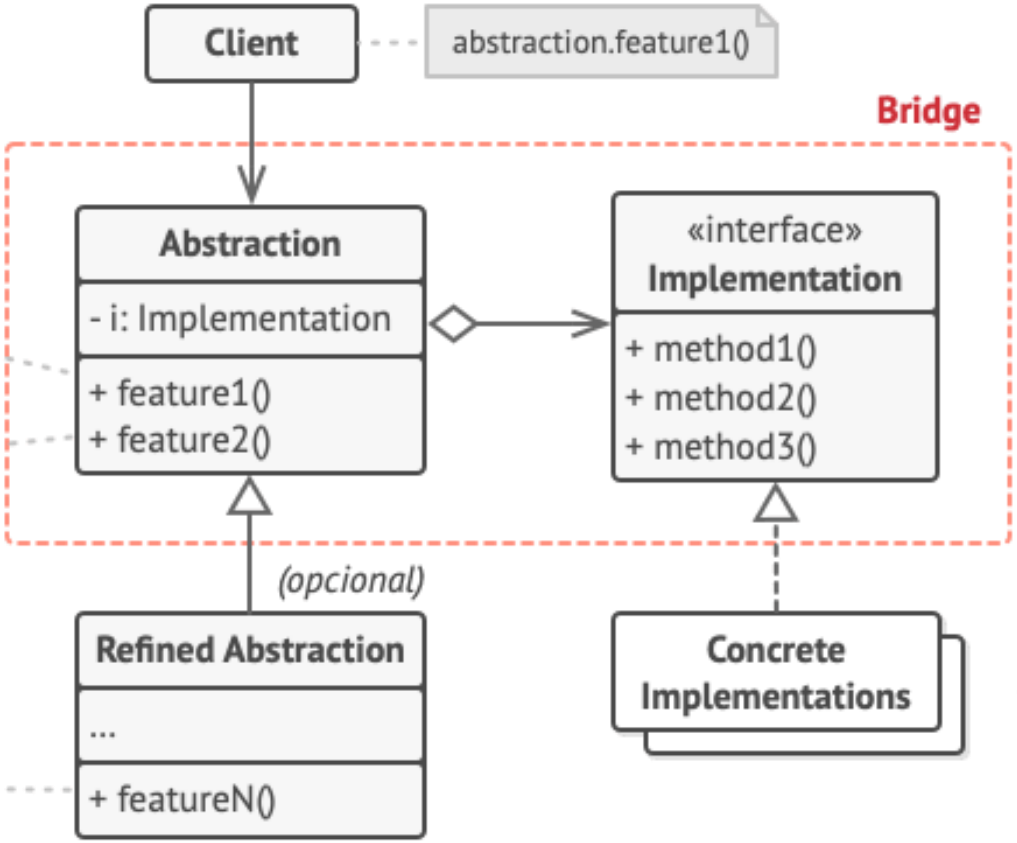
Muy útil cuando dos piezas de software están relacionadas directamente, sin embargo, existe una probabilidad de que una de ellas cambien y esto puede llevar a la necesidad de modificar la pieza del otro lado.



UNIVERSIDAD Popular del cesar

Patrones Estructurales – Bridge

Abstraction: Define la interfaz de alto nivel que utiliza el cliente y mantiene una referencia a implementation



Implementation: Define una estructura de clase común para todas los Implementaciones

RefinedAbstraction: Variación de la Abstraccion que añade funcionalidad adicional o especializada

ConcreteImplementations: Conjunto de clases que heredan de Implementation y que son propensas a cambiar, es por esta razón que se opta por implementar el patrón de diseño Bridge.

UNIVERSIDAD Popular del cesar

Patrones Estructurales – Bridge

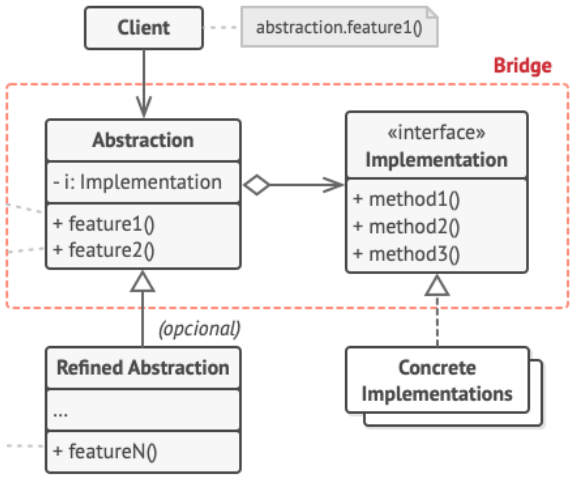
¿Cuándo implementarlo?

Quieres evitar la explosión de subclases al combinar múltiples dimensiones de variabilidad.

Necesitas que la abstracción y la implementación puedan evolucionar de forma independiente.

Requieres cambiar dinámicamente la implementación concreta durante la ejecución.

Deseas ocultar la complejidad de las implementaciones al cliente.

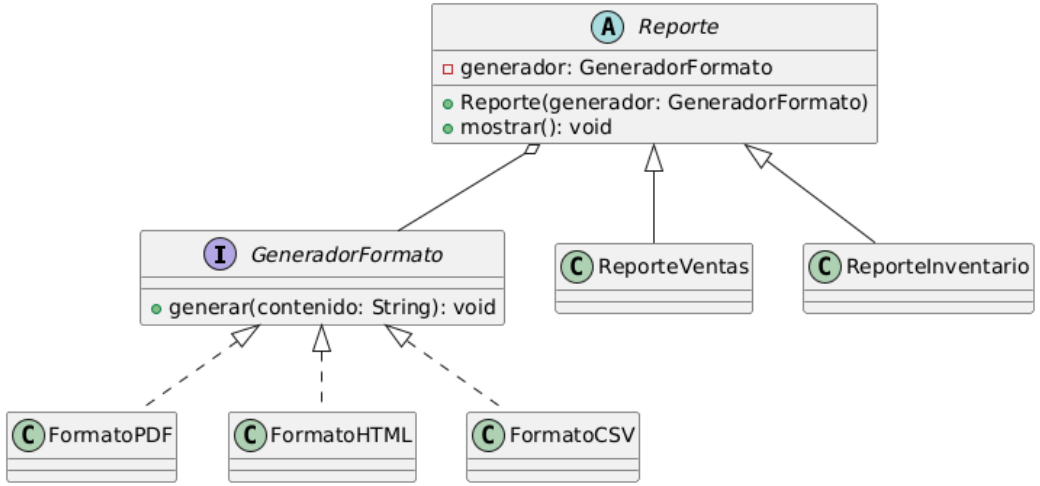


Casos de Uso Comunes y Realistas

Frameworks gráficos multiplataforma (ej. Qt, AWT/Swing en Java). Una API gráfica debe funcionar sobre diferentes sistemas operativos (Windows, Linux, macOS), sin que la abstracción (botones, ventanas, sliders) dependa de la plataforma.

Sistemas de persistencia de datos (ORMs). Separar la lógica de negocio del mecanismo de persistencia, para que el sistema funcione con diferentes motores de base de datos sin cambiar la lógica de negocio.

Servicios de pago en plataformas de e-commerce. Permitir distintos tipos de pagos (factura, suscripción, recarga) con diferentes pasarelas (PayPal, Stripe, MercadoPago).

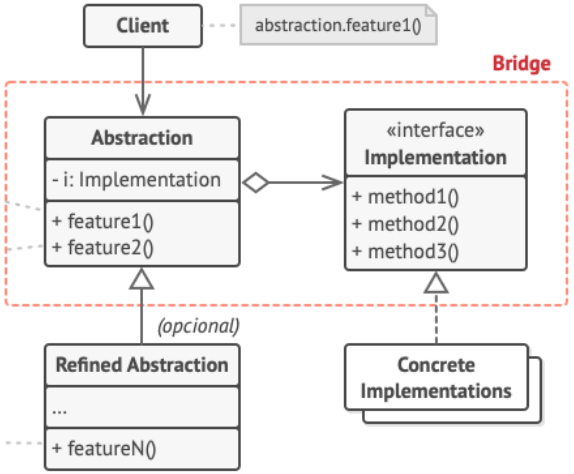


UNIVERSIDAD Popular del cesar

Patrones Estructurales – Bridge

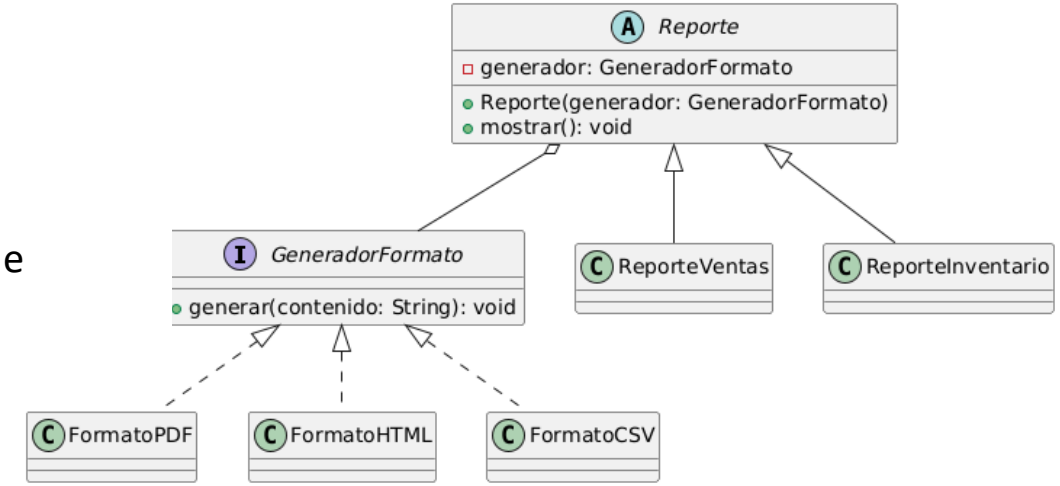
Ventajas

- **Desacoplamiento fuerte**. La abstracción y la implementación no están unidas por herencia directa.
- **Mayor extensibilidad.** Puedes agregar nuevas abstracciones o implementaciones sin afectar las existentes.
- **Cambios independientes.** Se pueden modificar las abstracciones o implementaciones sin reescribir toda la jerarquía.
- **Soporta cambios dinamico.** Puedes intercambiar implementaciones en tiempo de ejecución.



Desventajas

- **Complejidad inicial.** Requiere una estructura más elaborada que una simple herencia.
- **Puede ser innecesario para casos simples.** Si no hay múltiples dimensiones de variación, el patrón puede ser excesivo.

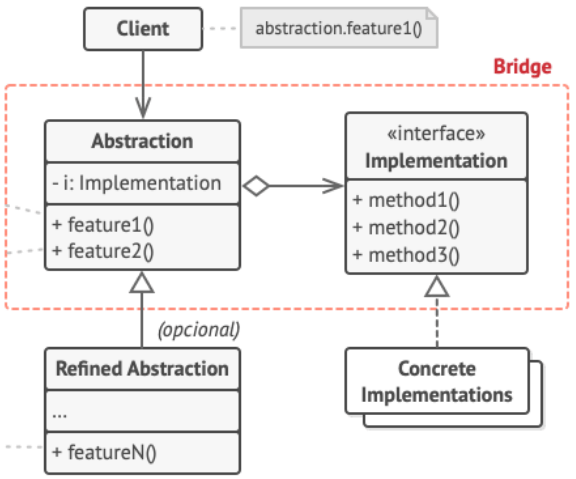


UNIVERSIDAD Popular del cesar

Patrones Estructurales – Bridge

```java
// Implementor
interface GeneradorFormato {
    void generar(String contenido);
}

// ConcreteImplementorA
class FormatoPDF implements GeneradorFormato {
    public void generar(String contenido) {
        System.out.println("[PDF] " + contenido);
    }
}

// ConcreteImplementorB
class FormatoHTML implements GeneradorFormato {
    public void generar(String contenido) {
        System.out.println("<html><body>" + contenido + "</body></html>");
    }
}
```

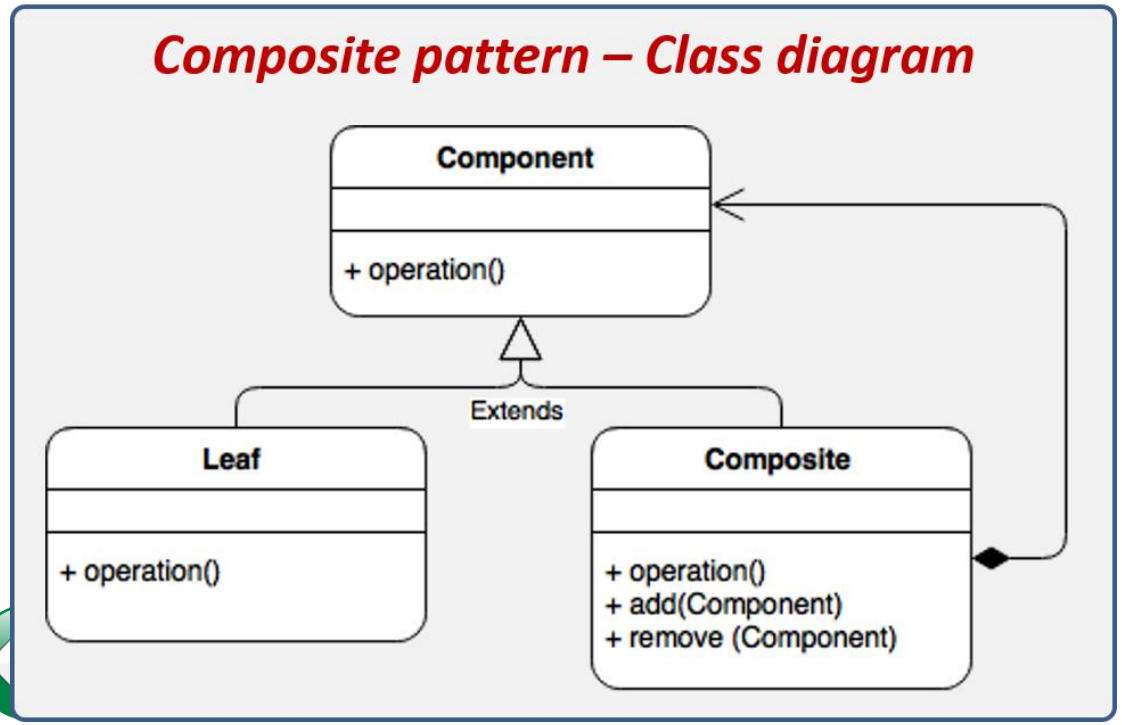
```java
// Abstraction
abstract class Reporte {
    protected GeneradorFormato generador;

    public Reporte(GeneradorFormato generador) {
        this.generador = generador;
    }

    public abstract void mostrar();
}

// RefinedAbstraction
class ReporteVentas extends Reporte {
    public ReporteVentas(GeneradorFormato generador) {
        super(generador);
    }

    public void mostrar() {
        generador.generar("Contenido del Reporte de Ventas");
    }
}
```



UNIVERSIDAD Popular del cesar

Patrones Estructurales – Composite

El **patrón Composite** permite **tratar objetos individuales y composiciones de objetos de manera uniforme**. Es decir, proporciona una estructura para construir árboles de objetos donde tanto los objetos simples (hojas) como los objetos compuestos (nodos internos) implementan la misma interfaz.



Problemas que Resuelve

Estructuras jerárquicas complejas y anidadas. Sin Composite, el cliente debe lidiar con diferentes clases para representar elementos simples y elementos compuestos, lo que lleva a lógica duplicada, acoplamiento innecesario y código más frágil.

Tratamiento uniforme de objetos simples y compuestos. En diseños sin Composite, se necesitan múltiples estructuras de control condicional para diferenciar entre elementos simples y compuestos.

Facilita operaciones recursivas. Cada componente, ya sea hoja o compuesto, conoce su comportamiento, y el cliente no necesita preocuparse por la profundidad de la jerarquía.

Mejora la extensibilidad y mantenibilidad. El patrón permite agregar nuevos tipos de hojas o compuestos sin modificar el código existente.

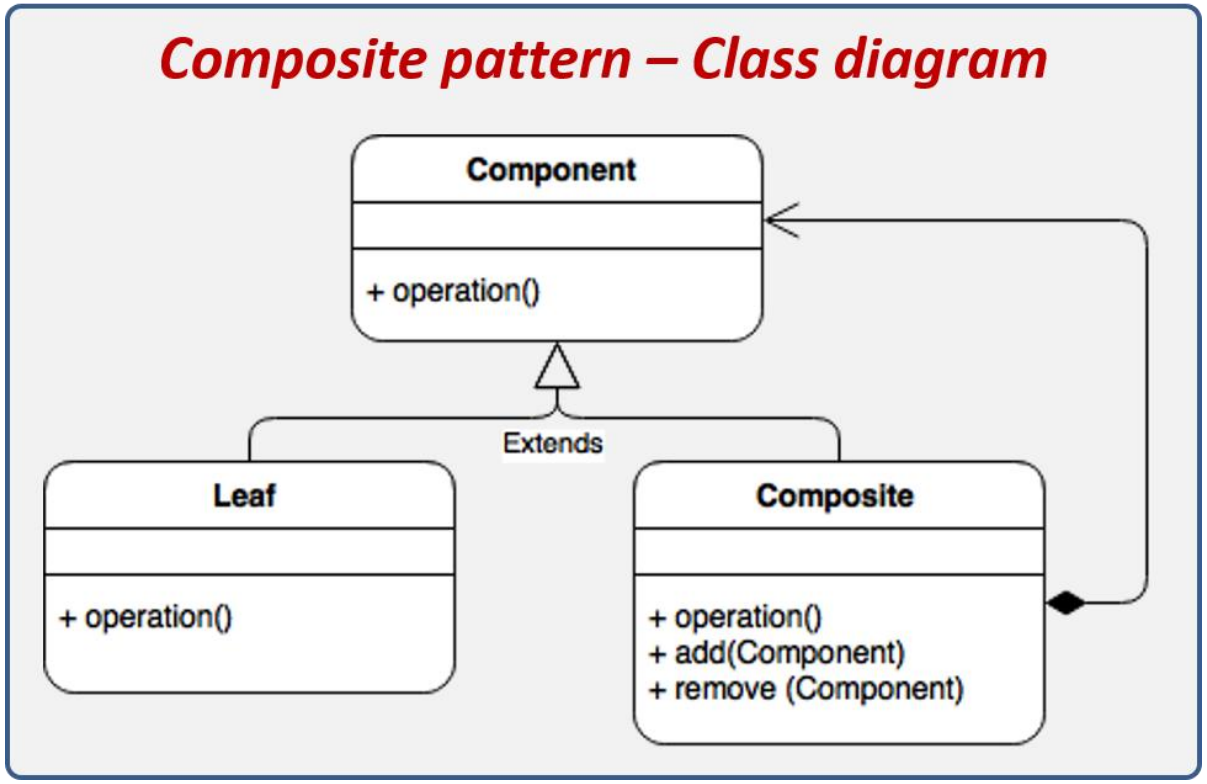
Desacopla el cliente de la estructura interna. El cliente solo ve la interfaz común, sin conocer si se trata de una hoja o de un nodo compuesto.

"Trata los objetos individuales y las colecciones de objetos de la misma manera."

UNIVERSIDAD Popular del cesar

Patrones Estructurales – Composite

Component: Generalmente es una interface o clase abstracta que tiene las operaciones mínimas que serán utilizadas, este componente deberá ser extendido por los otros dos componentes Leaf y Composite.



Composite: Este componente es el que le da vida al patrón de diseño ya que este objeto está conformado por un conjunto de Component y Leaf. En teoría de árboles este componente representaría una rama.

Leaf: El leaf u hoja, representa la parte más simple o pequeña de toda la estructura y ésta hereda de Component. Leaf recibe su nombre de la teoría de árboles, donde se le nombra así a todo nodo que no tiene descendencia, en este caso son clases simple que no están compuestas de otras.

UNIVERSIDAD Popular del cesar

Patrones Estructurales – Composite

¿Cuándo implementarlo?

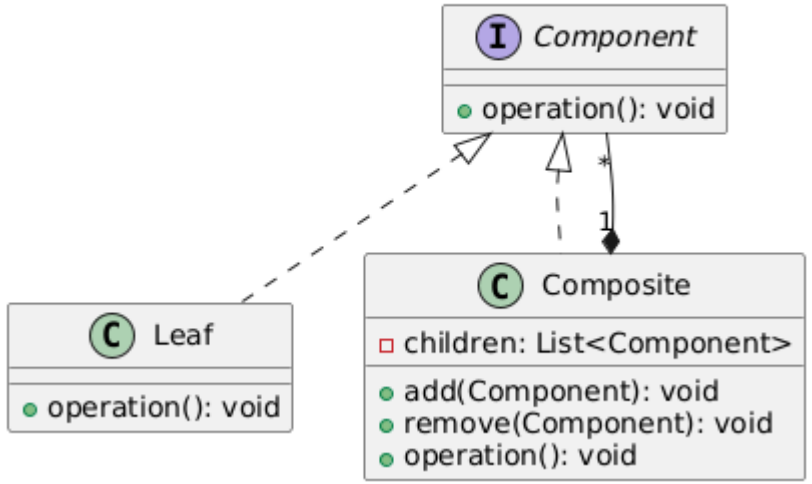
Tienes una estructura jerárquica de objetos (como un árbol).

Quieras tratar objetos individuales y contenedores de manera uniforme.

Necesites recorrer recursivamente una estructura..

Desees aplicar operaciones genéricas sobre todos los elementos, sin importar su tipo concreto.

Patrón Composite

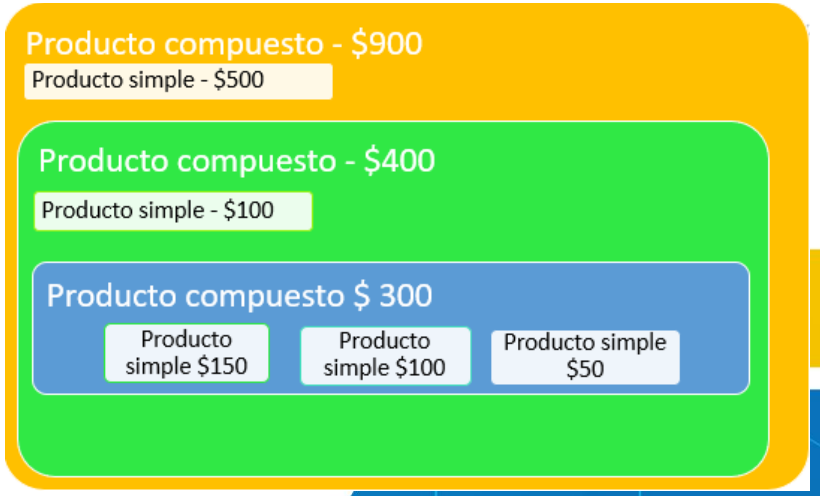


Casos de Uso Comunes y Realistas

Sistemas de Archivos (Archivos y Carpetas). Un sistema operativo representa archivos y carpetas. Las carpetas pueden contener archivos y otras carpetas.

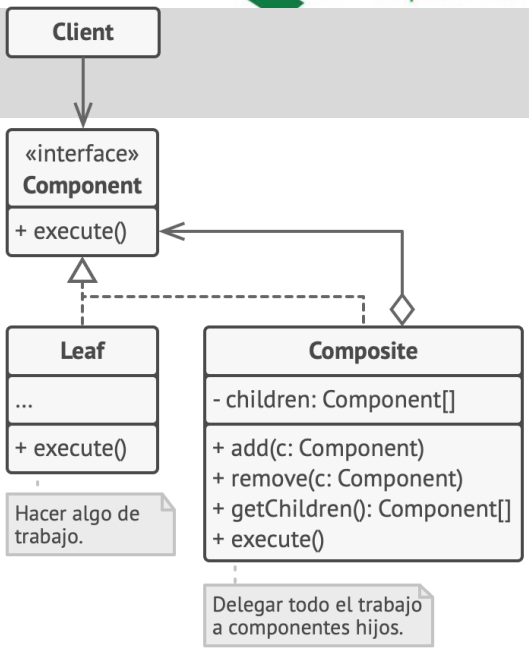
Menús y Submenús en Interfaces Gráficas (GUI). Aplicaciones de escritorio o móviles manejan menús anidados.

Escenas Gráficas en Juegos o UI. Motores de videojuegos o librerías gráficas usan árboles de escenas (Scene Graph).



UNIVERSIDAD Popular del cesar

Patrones Estructurales – Composite

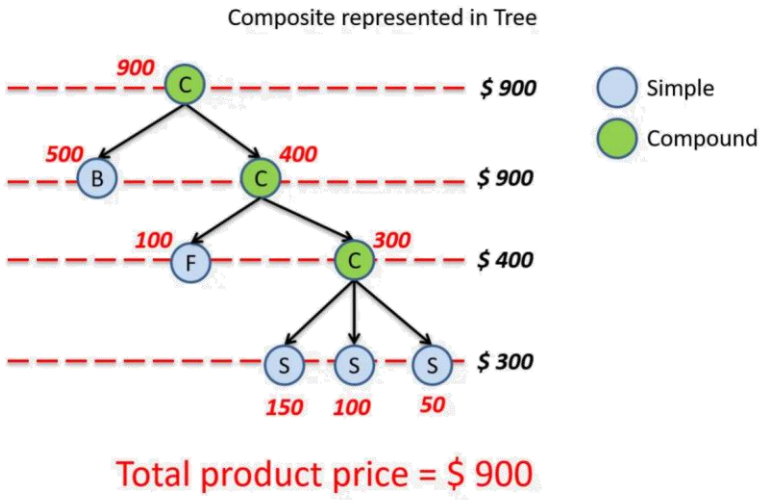


Ventajas

- **Transparencia:** Se puede tratar a los objetos hoja y a los compuestos de la misma manera.
- **Extensibilidad:** Puedes agregar nuevas clases de componentes sin modificar las existentes.
- **Simplifica el cliente:** El código que usa la estructura no necesita diferenciar entre tipos de componentes

Desventajas

- **Diseño menos específico:** Puede ser difícil restringir qué tipos de hijos puede tener un Composite.
- **Sobrecarga en operaciones innecesarias:** En algunos casos, las hojas pueden tener que implementar métodos que no usan (por ejemplo, add()), lo cual viola el principio de segregación de interfaces si no se cuida.
- **Complejidad de depuración:** Las llamadas recursivas y estructuras grandes pueden hacer difícil el debugging.



UNIVERSIDAD Popular del cesar

Patrones Estructurales – Composite

```java
// Componente base
public abstract class FileSystemComponent {
    protected String name;

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

    public abstract void display(int depth);
    public abstract long getSize();

    // Operaciones por defecto para composites
    public void add(FileSystemComponent component) {
        throw new UnsupportedOperationException();
    }

    public void remove(FileSystemComponent component) {
        throw new UnsupportedOperationException();
    }

    public FileSystemComponent getChild(int index) {
        throw new UnsupportedOperationException();
    }
}
```

```java
// Hoja
public class File extends FileSystemComponent {
    private long size;

    public File(String name, long size) {
        super(name);
        this.size = size;
    }

    public void display(int depth) {
        System.out.println("-".repeat(depth) + " " + name + " (" + size + " bytes)");
    }

    public long getSize() {
        return size;
    }
}
```

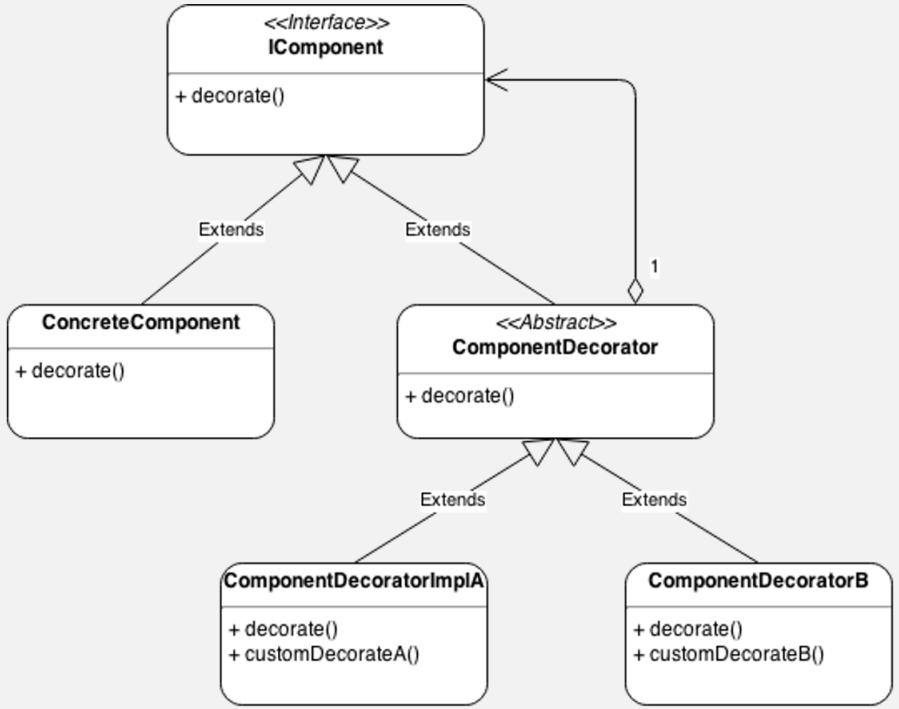
```java
// Compuesto
public class Directory extends FileSystemComponent {
    private List<FileSystemComponent> components = new ArrayList<>();

    public Directory(String name) {
        super(name);
    }

    public void add(FileSystemComponent component) {
        components.add(component);
    }

    public void remove(FileSystemComponent component) {
        components.remove(component);
    }

    public void display(int depth) {
        System.out.println("-".repeat(depth) + "+ " + name + "/");
        for (FileSystemComponent component : components) {
            component.display(depth + 2);
        }
    }

    public long getSize() {
        return components.stream()
                .mapToLong(FileSystemComponent::getSize)
                .sum();
    }
}
```

UNIVERSIDAD Popular del cesar

Patrones Estructurales – Decorator (Wrapper)

El patrón de diseño **Decorator** permite añadir nuevas funcionalidades a objetos dinámicamente sin alterar su estructura, proporcionando una alternativa flexible a la herencia para extender funcionalidad.

Decorator pattarn – Class diagram



Problemas que Resuelve

Explosión de subclases por combinaciones de comportamiento. Cuando una clase debe admitir múltiples combinaciones de comportamientos opcionales, la solución tradicional con herencia obliga a crear subclases para cada combinación posible.

Limitaciones de la herencia. La herencia en Java es estática y vertical. Una vez que extiendes una clase, ya no puedes cambiar dinámicamente su comportamiento sin modificarla o reescribirla.

Violación del principio de responsabilidad única. Una clase con demasiadas responsabilidades se vuelve difícil de mantener, probar y extender. **Decorator** favorece la separación de responsabilidades y promoviendo una arquitectura más limpia.

Falta de extensibilidad sin modificar código existente. Modificar una clase para agregar nuevas funcionalidades viola el principio abierto/cerrado

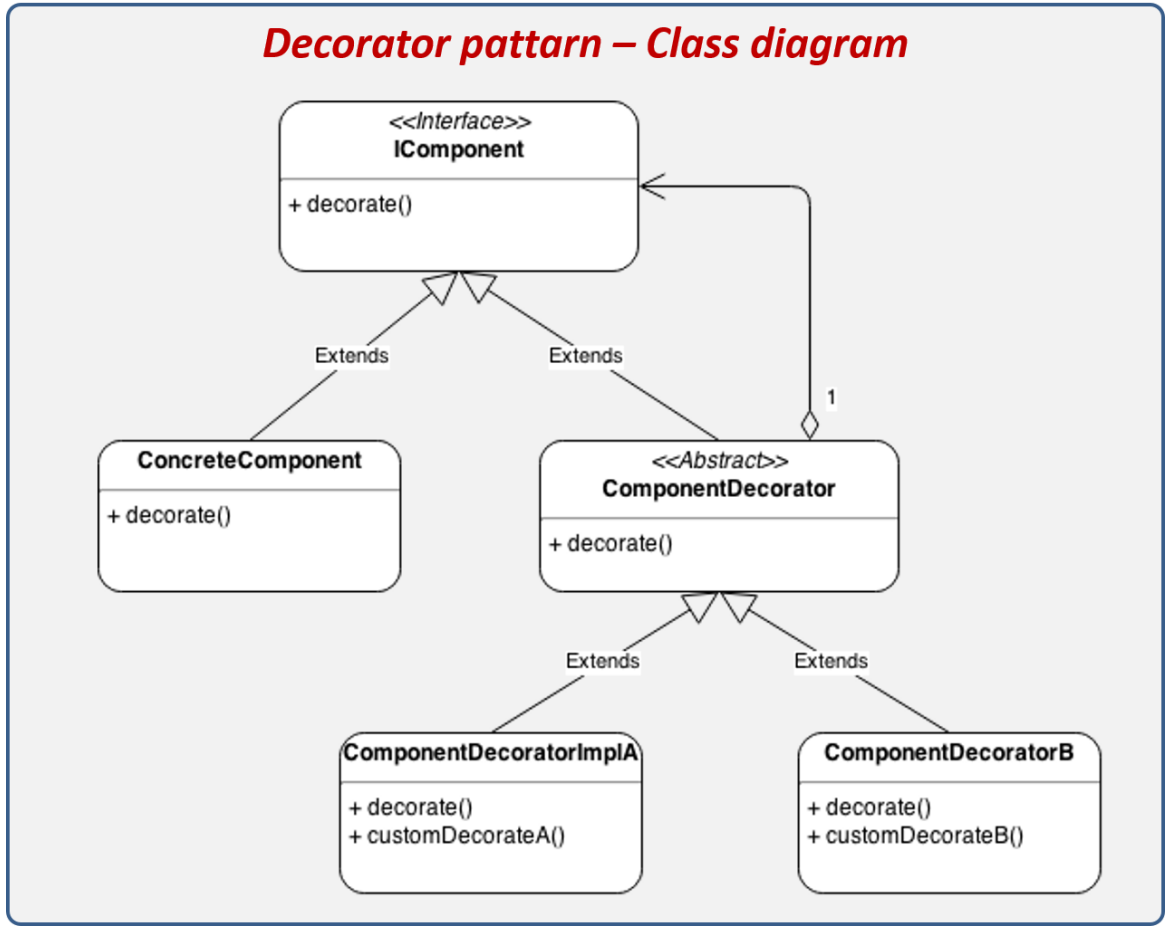
En vez de crear múltiples subclases para cada combinación de comportamientos, el patrón Decorator permite componerlos en tiempo de ejecución usando composición.

UNIVERSIDAD Popular del cesar

Patrones Estructurales – Decorator (Wrapper)

IComponent: Interface que define la estructura minina del componente o componentes que pueden ser decorados.

ConcreteComponent: Implementación de IComponent y define un objeto concreto que puede ser decorado.



ComponentDecorator: Por lo general es una clase abstracta que define la estructura mínima de un Decorador, el cual mínimamente deben de heredar de IComponent y contener alguna subclase de IComponent al cual decorarán.

ComponentDecoratorImpl: Representan todos los decoradores concretos que heredan de ComponentDecorator.

UNIVERSIDAD Popular del cesar

Patrones Estructurales – Decorator (Wrapper)

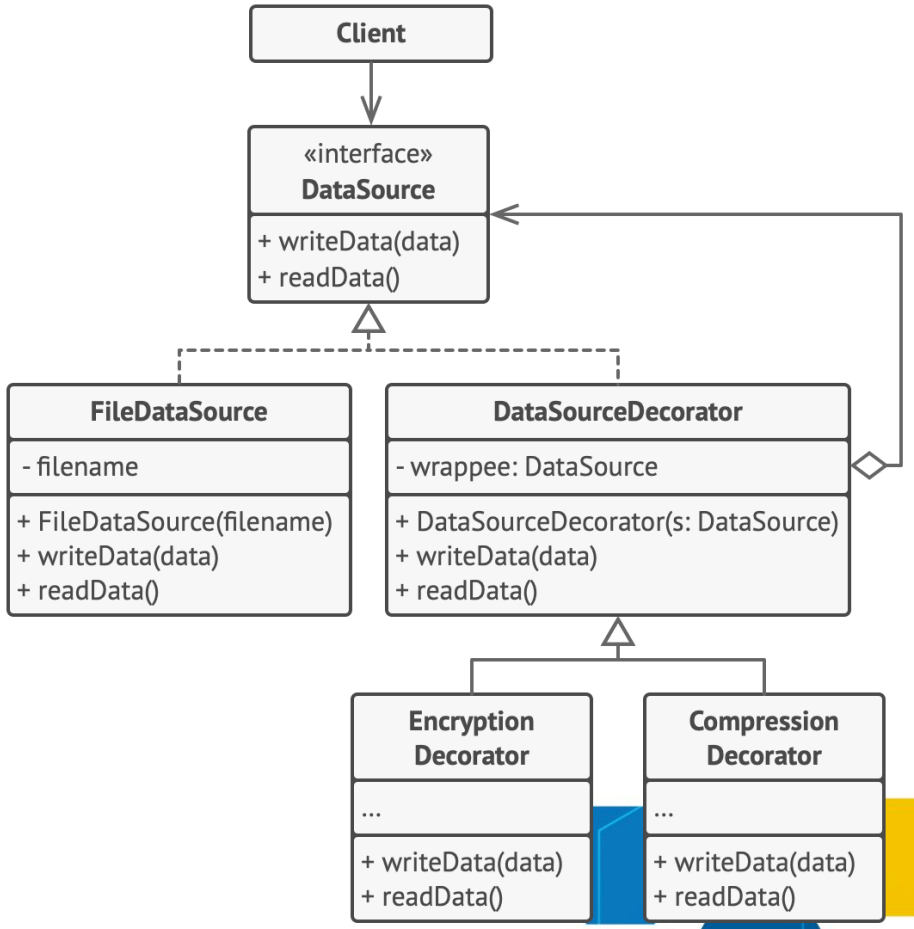
¿Cuándo implementarlo?

Cuando necesitas **agregar responsabilidades** a objetos individuales de forma dinámica y controlada

Cuando no puedes usar herencia (por restricciones del lenguaje o porque ya estás usando otra clase base).

Cuando quieres seguir el principio de abierto/cerrado sin multiplicar subclases.

Cuando deseas aplicar **composición en lugar de herencia**.



Casos de Uso Comunes y Realistas

Flujos de Entrada/Salida en Java (InputStream, BufferedInputStream, etc.). Cuando trabajamos con flujos de datos (lectura o escritura de archivos, sockets, streams de red, etc.) es frecuente querer agregar funcionalidades.

Intercepción de peticiones HTTP. Frameworks web como Spring Boot, Servlet Filters, Express.js, NestJS aplican decoradores para interceptar solicitudes y añadir funcionalidades.

Procesamiento de mensajes, logs o auditoría. En sistemas de logging o auditoría puedes tener: Log simple, Log con timestamps, Log con nivel de severidad, Log persistente

UNIVERSIDAD Popular del cesar

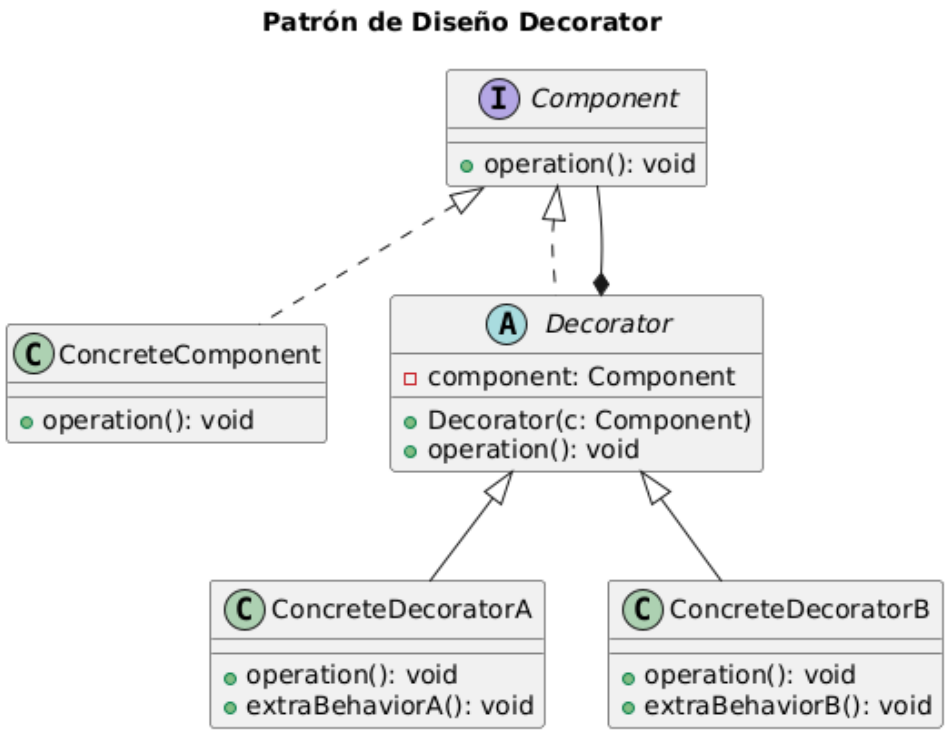
Patrones Estructurales – Decorator

Ventajas

- Respeta el principio abierto/cerrado.
- Composición flexible de comportamientos.
- Permite la adición de funcionalidades en tiempo de ejecución.
- Evita la explosión de subclases.
- Se alinea con el principio de responsabilidad única.

Desventajas

- Puede generar una gran cantidad de clases si hay muchos decoradores.
- La depuración puede ser más compleja, debido a múltiples capas de objetos.
- El orden de los decoradores afecta el comportamiento (por lo tanto, debe controlarse cuidadosamente).
- Aumenta la complejidad estructural si se abusa del patrón.



UNIVERSIDAD Popular del cesar

Patrones Estructurales – Decorator

Componente base

```
public interface Message {
    String getContent();
}
```

Decorador base

```
public abstract class MessageDecorator implements Message {
    protected final Message message;

    public MessageDecorator(Message message) {
        this.message = message;
    }
}
```

Decorador concreto

```
public class EncryptedDecorator extends MessageDecorator {
    public EncryptedDecorator(Message message) {
        super(message);
    }

    @Override
    public String getContent() {
        return "[Encrypted] " + message.getContent();
    }
}
```

Componente concreto

```
public class SimpleMessage implements Message {
    private final String content;

    public SimpleMessage(String content) {
        this.content = content;
    }

    @Override
    public String getContent() {
        return content;
    }
}
```

Decorador concreto

```
public class UppercaseDecorator extends MessageDecorator {
    public UppercaseDecorator(Message message) {
        super(message);
    }

    @Override
    public String getContent() {
        return message.getContent().toUpperCase();
    }
}
```

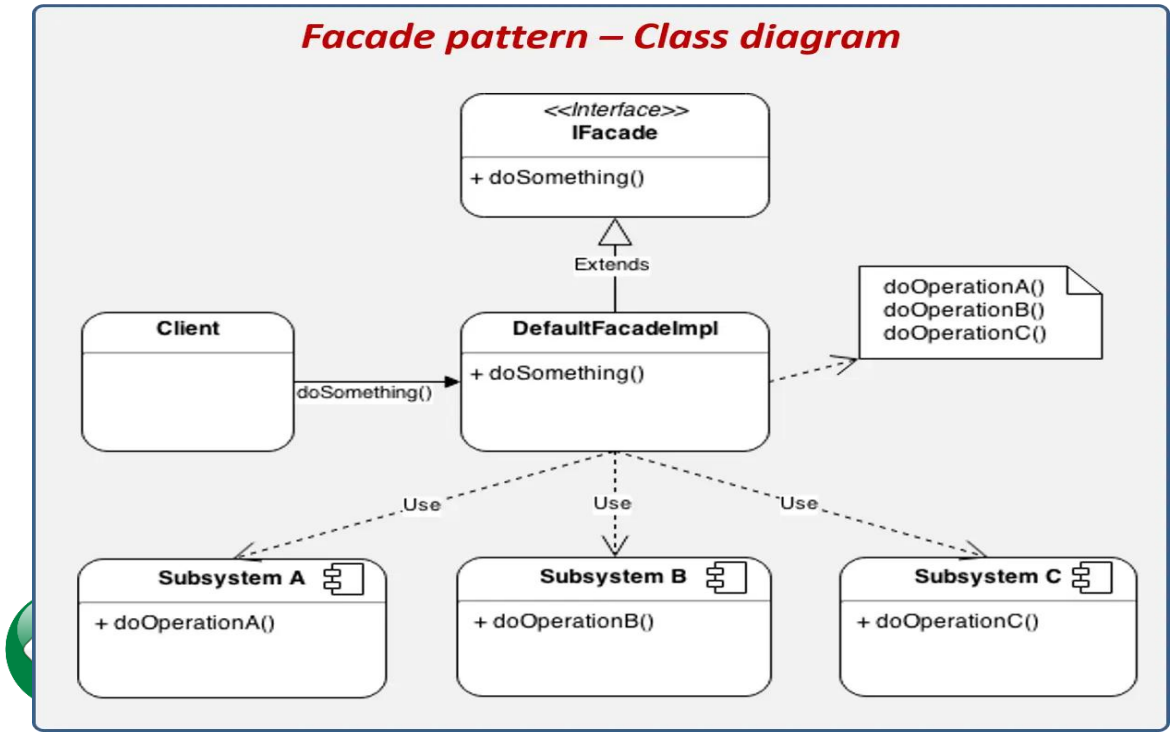
Cliente

```
public class Main {
    public static void main(String[] args) {
        Message simple = new SimpleMessage("hola mundo");

        Message upper = new UppercaseDecorator(simple);
        Message encrypted = new EncryptedDecorator(upper);

        System.out.println(encrypted.getContent());
        // Salida: [Encrypted] HOLA MUNDO
    }
}
```

UNIVERSIDAD Popular del cesar

Patrones Estructurales – Facade

El patrón **Facade** proporciona una interfaz unificada y simplificada para las interfaces de un conjunto de subsistemas, ocultando la complejidad de los subsistemas y exponiendo únicamente operaciones de alto nivel que sean más fáciles de utilizar para el cliente



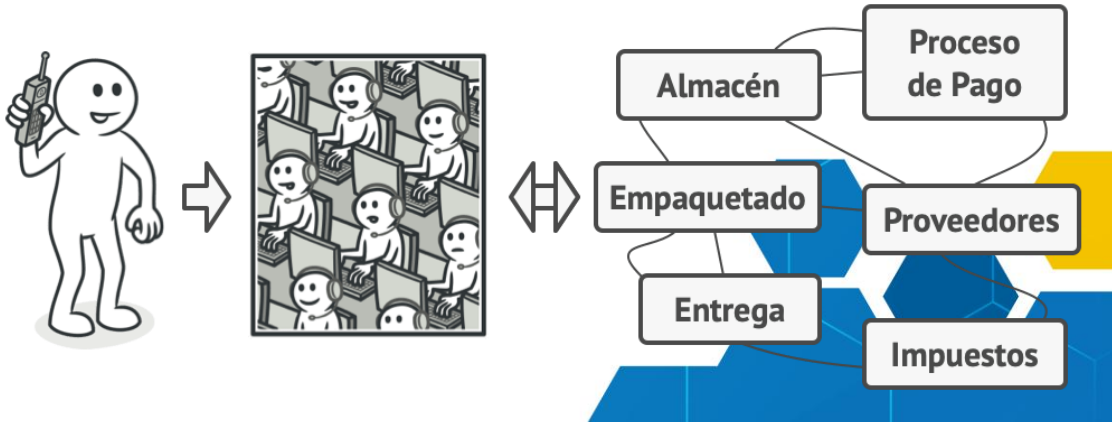
Problemas que Resuelve

Complejidad exponencial en sistemas grandes. El cliente debe conocer demasiadas dependencias, asociadas a **m**uchos subsistemas interconectados.

Acoplamiento fuerte cliente-subsistema. Los clientes dependen de clases internas. Cambios internos afectan múltiples capas.

Falta de control de acceso uniforme. Dificultad para aplicar validaciones, seguridad o logging centralizado.

Múltiples pasos para ejecutar operaciones de negocio. El cliente debe orquestar los pasos manualmente.

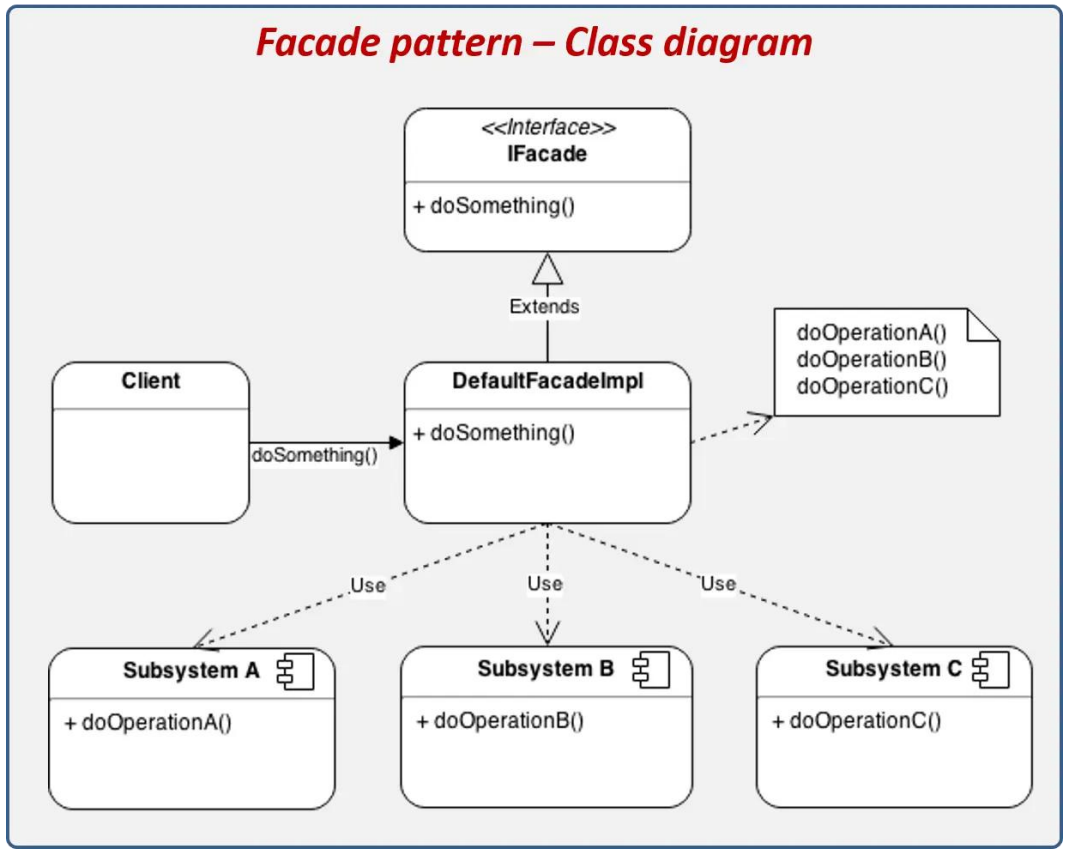


UNIVERSIDAD Popular del cesar

Patrones Estructurales – Facade

IFacade: Proporciona una interface de alto nivel que oculta la complejidad de interactuar con varios sistemas para realizar una operación.

Client : Sistema o evento que interactúa con la fachada.



DefaultFacadeImpl: Representa la implementación de IFacade y se encarga de comunicarse con todos los subsistemas.

Subsystems: Representa módulos o subsistemas que exponen interfaces para comunicarse con ellos.

UNIVERSIDAD Popular del cesar

Patrones Estructurales – Facade

¿Cuándo implementarlo?

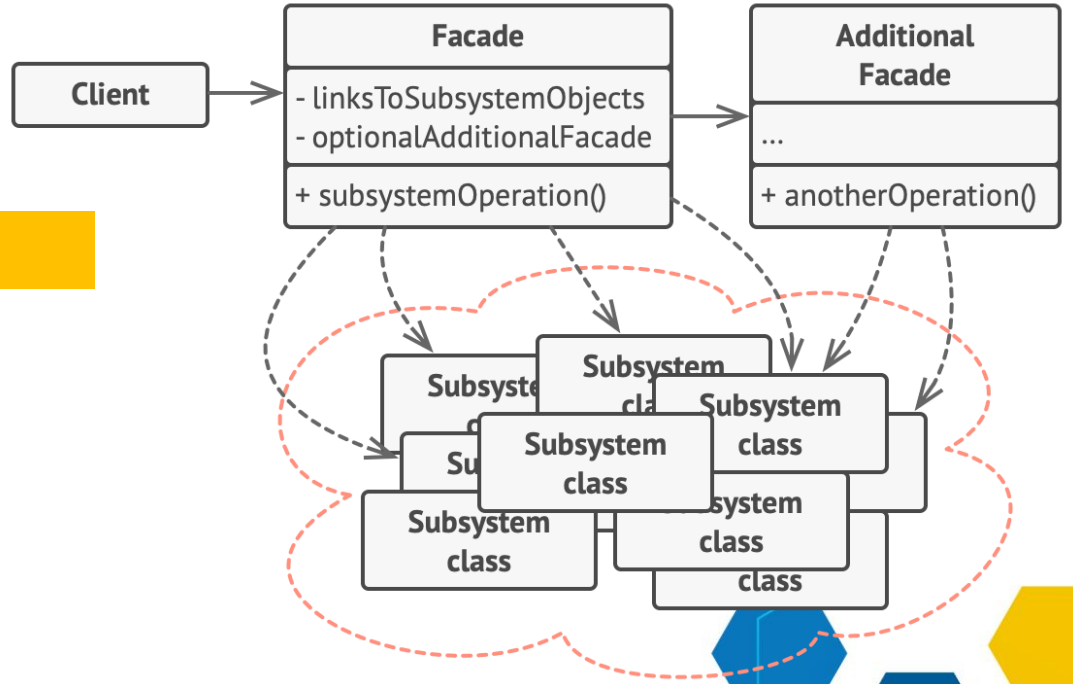
Cuando tienes subsistemas complejos con muchas dependencias.

Cuando deseas reducir el conocimiento que necesita el cliente.

Cuando buscas mejorar la mantenibilidad, estabilidad y encapsulamiento.

Cuando deseas aplicar un único punto de acceso seguro, validado o auditado.

Cuando necesitas desacoplar librerías/frameworks complejos del código cliente.



Casos de Uso Comunes y Realistas

- APIs de servicios complejos (orquestación de microservicios).
- Integración de librerías/frameworks externos.
- Gestión de procesos de negocio multi-etapa.
- Creación de SDKs o clientes API simplificados.
- Interacción con sistemas legacy.

UNIVERSIDAD Popular del cesar

Patrones Estructurales – Facade

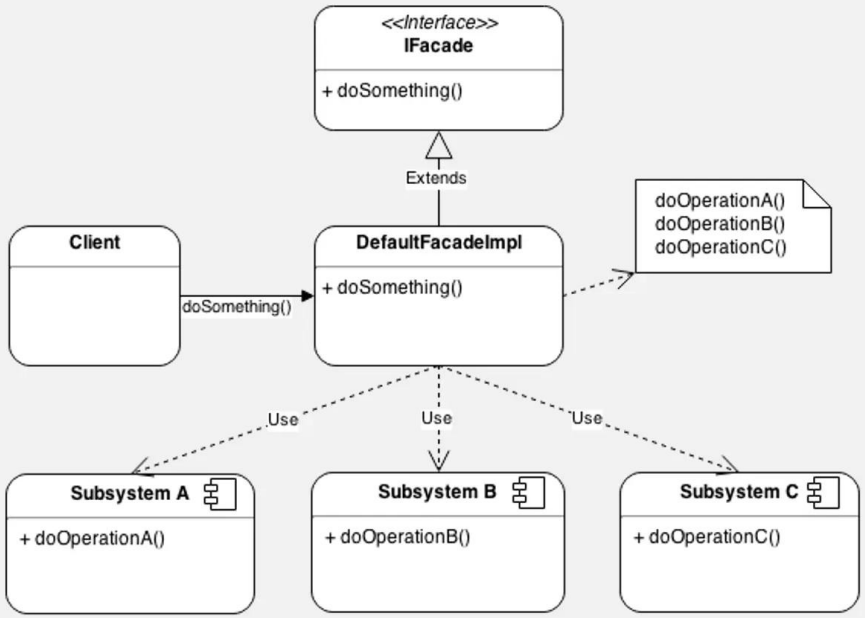
Ventajas

- **Reduce el acoplamiento.** El cliente no necesita conocer los detalles del subsistema.
- **Oculta la complejidad interna.** Expone una API simple y enfocada en casos de negocio.
- **Facilita el mantenimiento.** Cambios internos no afectan al cliente mientras la fachada no cambie.
- **Mejora la legibilidad del código.** Interfaces simples, menos ruido de bajo nivel
- **Facilita pruebas unitarias.** Puedes probar el facade como unidad, sin preocuparte por la orquestación interna.

Desventajas

- **Puede esconder demasiado.** Si la fachada abstrae en exceso, puede ser difícil diagnosticar problemas internos.
- **Sobrecarga de diseño.** Si no hay suficiente complejidad, puede ser innecesario y sobre diseñado.
- **Riesgo de convertirse en Dios-Object.** Mal aplicado, el facade puede centralizar demasiadas responsabilidades.

Facade pattern – Class diagram



UNIVERSIDAD Popular del cesar

Patrones Estructurales – Facade

Sub Sistemas

```
public class InventoryService {
    public void reserveItem(String itemId, int qty) { ... }
}

public class PaymentService {
    public void processPayment(String accountId, double amount) { ... }
}

public class ShippingService {
    public void arrangeShipping(String orderId) { ... }
}
```

Facade

```
public class OrderFacade {
    private InventoryService inventory;
    private PaymentService payment;
    private ShippingService shipping;

    public OrderFacade() {
        this.inventory = new InventoryService();
        this.payment = new PaymentService();
        this.shipping = new ShippingService();
    }

    public void placeOrder(String itemId, int qty, String accountId, double amount,
        inventory.reserveItem(itemId, qty);
        payment.processPayment(accountId, amount);
        shipping.arrangeShipping(orderId);
    }
}
```

Cliente sin facade

```
public class OrderClient {
    public void placeOrder() {
        InventoryService inventory = new InventoryService
        inventory.reserveItem("item-123", 2);

        PaymentService payment = new PaymentService();
        payment.processPayment("acct-456", 99.99);

        ShippingService shipping = new ShippingService();
        shipping.arrangeShipping("order-789");
    }
}
```

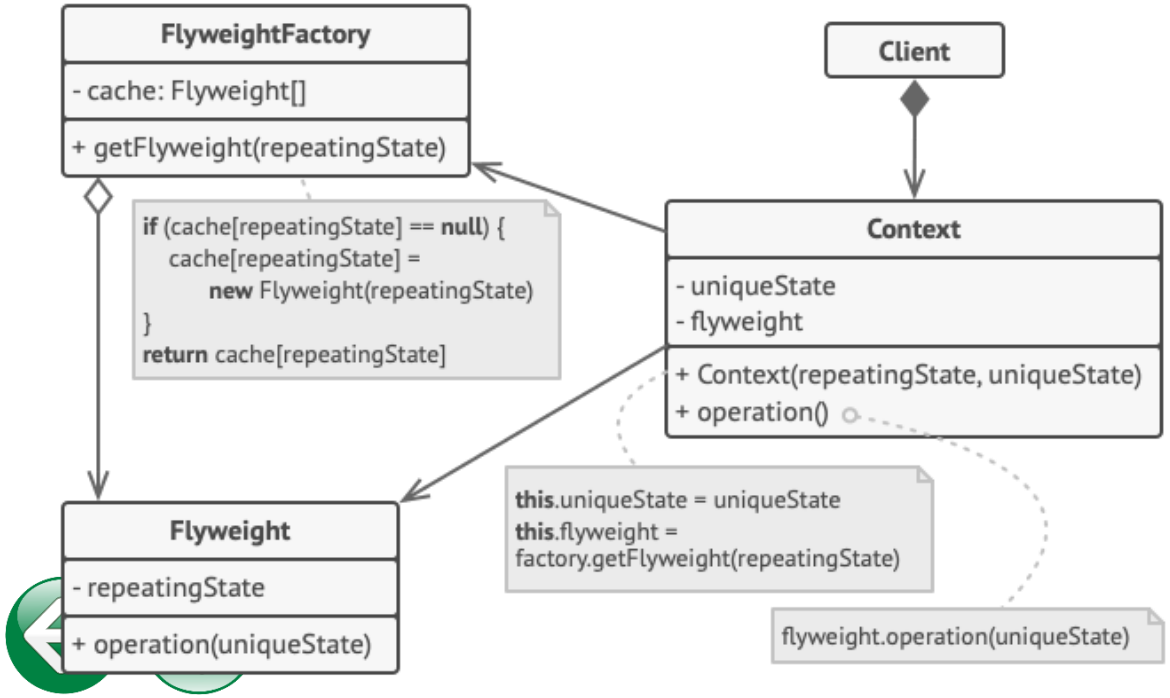
Cliente con facade

```
public class OrderClient {
    public void placeOrder() {
        OrderFacade facade = new OrderFacade();
        facade.placeOrder("item-123", 2, "acct-456", 99.99, "order-789");
    }
}
```

UNIVERSIDAD Popular del cesar

Patrones Estructurales – Flyweight

El patrón **Flyweight** minimiza el uso de memoria compartiendo eficientemente datos comunes entre múltiples objetos similares, separando el estado intrínseco del extrínseco.



Propósitos de FlyWeight

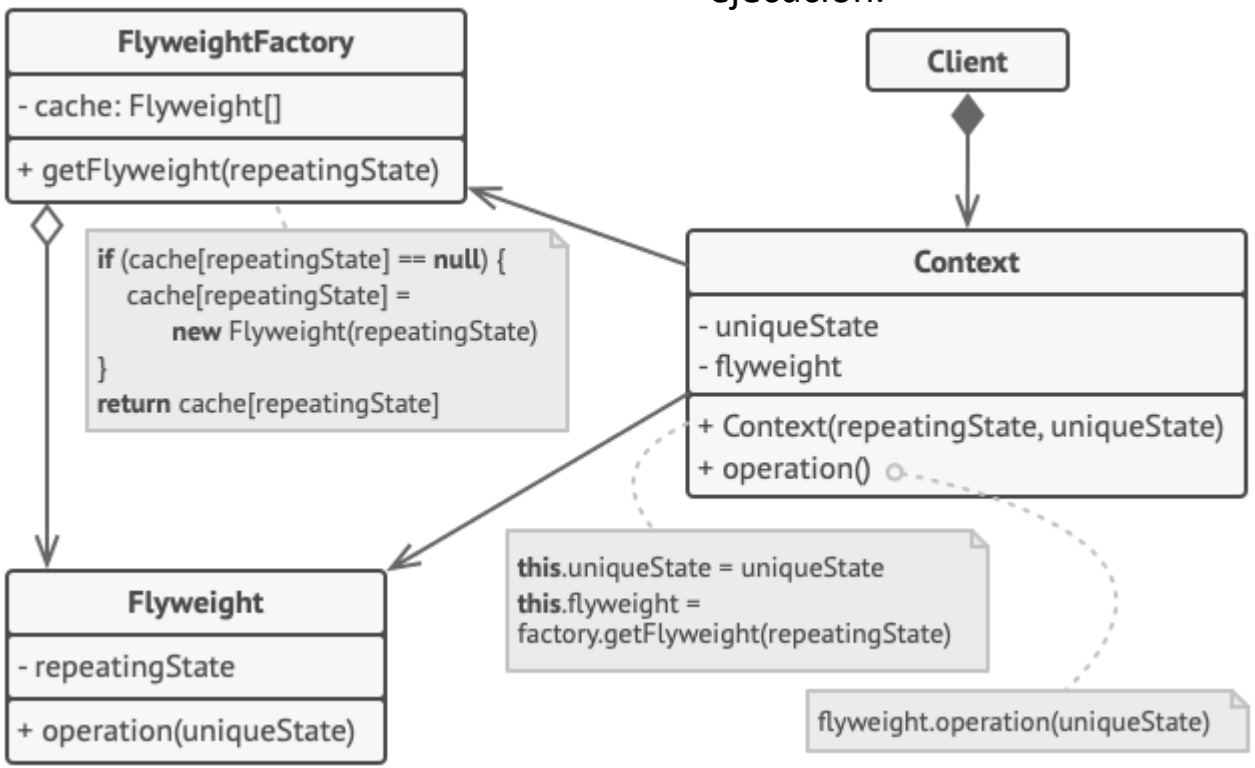
- **Centra su atención en la creación de objetos ligeros**, compartiendo las partes reutilizables con otros objetos.
- **Permite un manejo eficiente de memoria**, reduce la cantidad de memoria requerida por la aplicación.
- Es útil cuando la optimización de recursos es primordial, este patrón **elimina la redundancia de objetos con estados idénticos**.
- Divide los objetos en dos partes: estado **intrínseco** y estado **extrínseco**.
- Utiliza cache para almacenar todos los estados intrínsecos (común)

UNIVERSIDAD Popular del cesar

Patrones Estructurales – Flyweight

FlyweightFactory: Fábrica que utilizaremos para crear los objetos Flyweight u objetos ligeros.

Client: Objeto que dispara la ejecución.



El **Contexto** contiene el estado extrínseco, único en todos los objetos originales.

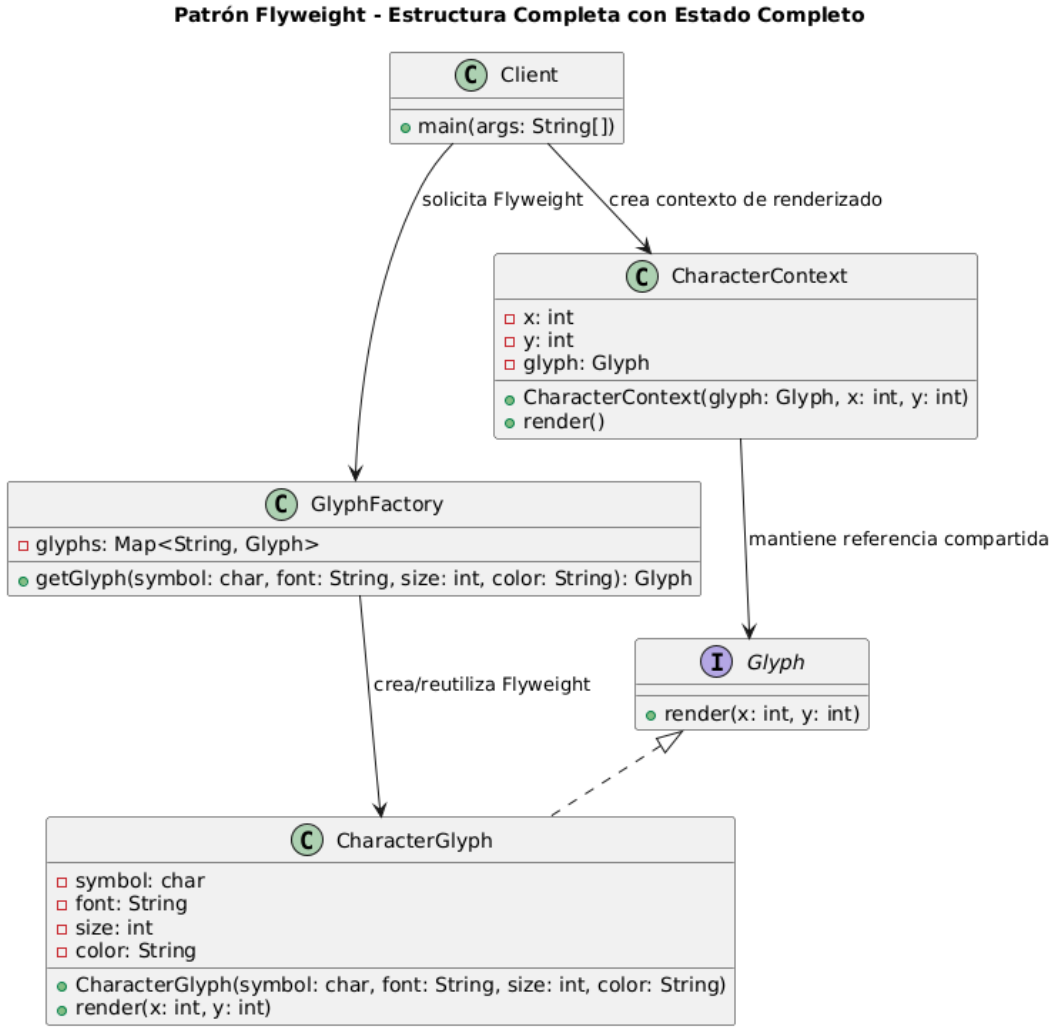
Cuando un contexto se empareja con uno de los objetos flyweight, representa el estado completo del objeto original.

Flyweight: Corresponde a los objetos que deseamos reutilizar con el fin de que nuestros objetos sean más ligeros (intrinseco).

Patrones Estructurales – Flyweight

Casos de Uso Comunes y Realistas

- **Sistemas gráficos**: editores de texto, editores vectoriales.
- **Motores de videojuegos** (gestión de partículas, árboles, terrenos, enemigos repetidos).
- **Sistemas de caching**
- **Sistemas GIS** (mapas con miles de puntos o polígonos repetidos).
- **Tableros electrónicos** y paneles de control.



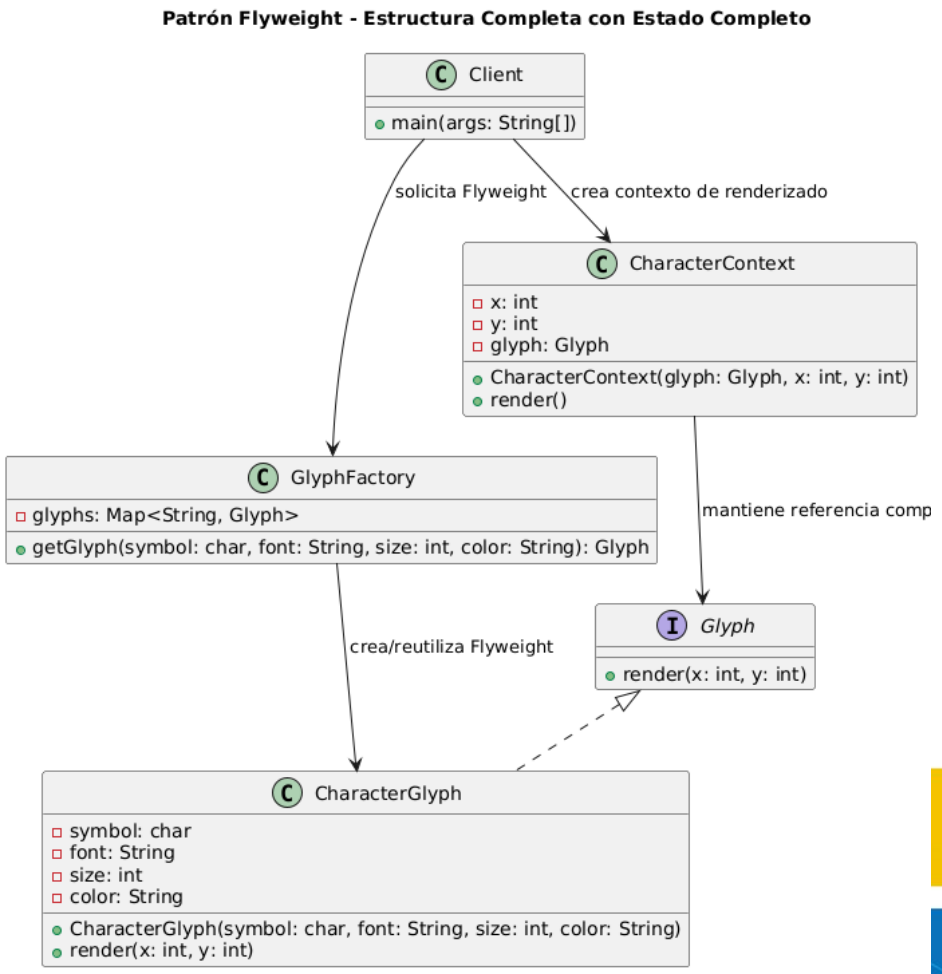
Patrones Estructurales – Flyweight

Ventajas

- **Reduce considerablemente el uso de memoria**.
- **Mejora el rendimiento** cuando existen grandes volúmenes de objetos similares.
- **Facilita la reutilización** y el control centralizado de objetos.

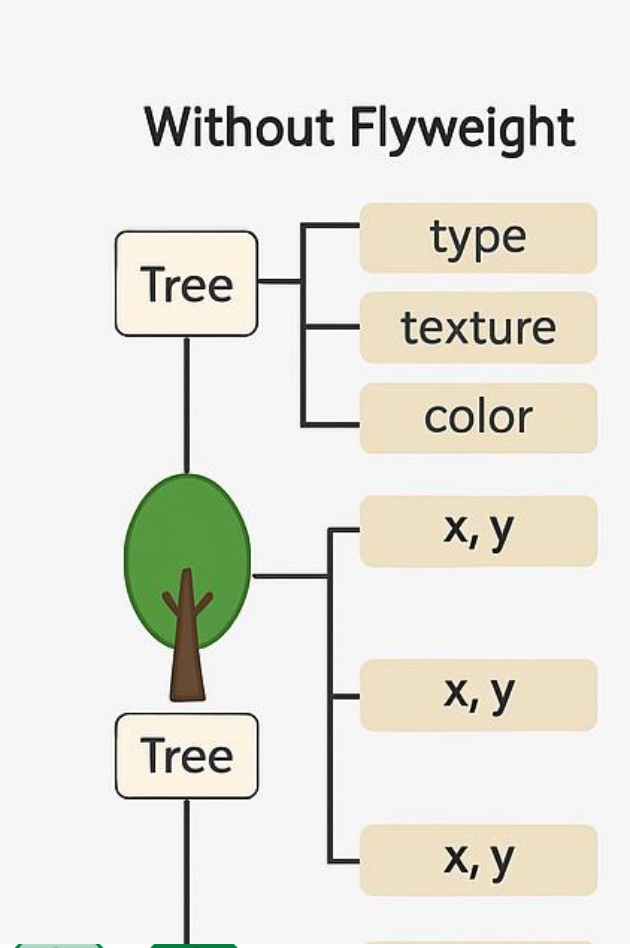
Desventajas

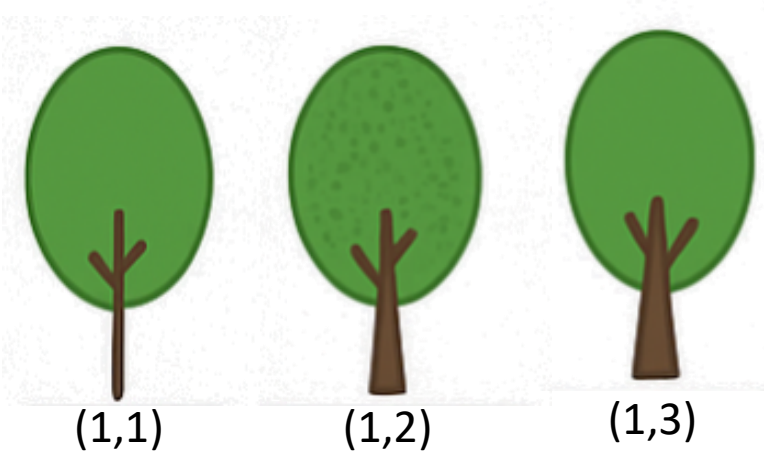
- **Aumenta la complejidad de diseño.** Es difícil separar el estado intrínseco (compartido) y extrínseco (no compartido).
- **Puede dificultar el mantenimiento** si no se documenta bien.
- **No siempre aplicable.** Solo funciona cuando hay suficiente redundancia.

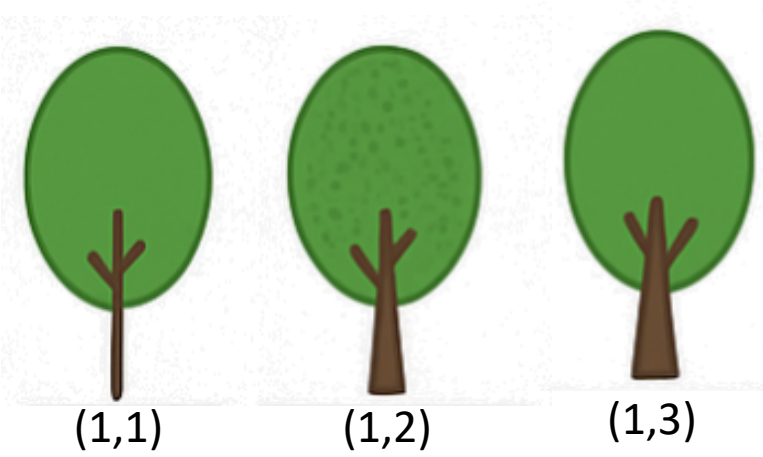


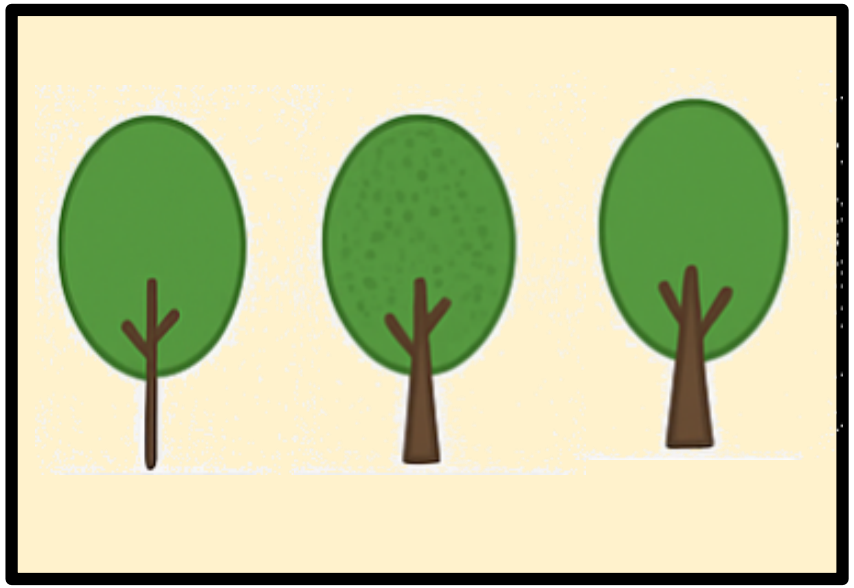
UNIVERSIDAD Popular del cesar

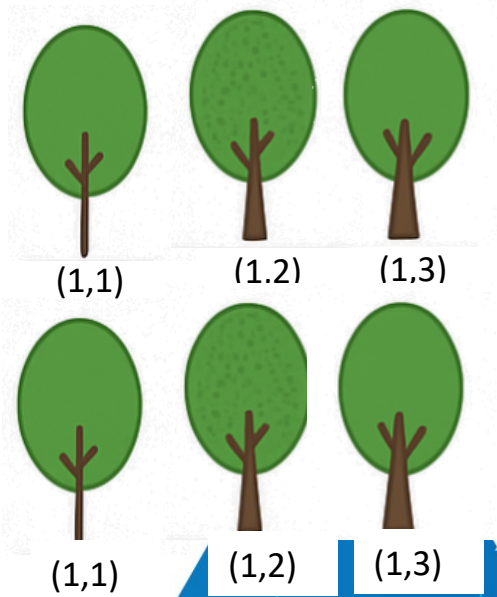
Patrones Estructurales – Flyweight











UNIVERSIDAD Popular del cesar

Patrones Estructurales – Flyweight

```java
// Flyweight
interface TreeType {
    void draw(int x, int y);
}

// ConcreteFlyweight
class ConcreteTreeType implements TreeType {
    private final String species;
    private final String texture;
    private final String color;

    public ConcreteTreeType(String species, String texture, String color) {
        this.species = species;
        this.texture = texture;
        this.color = color;
    }

    @Override
    public void draw(int x, int y) {
        System.out.println("Drawing " + species + " tree at (" + x + "," + y
    }
}
```

```java
// Contexto: El objeto lógico completo
class Tree {
    private final int x;
    private final int y;
    private final TreeType type;

    public Tree(int x, int y, TreeType type) {
        this.x = x;
        this.y = y;
        this.type = type;
    }

    public void draw() {
        type.draw(x, y);
    }
}
```

```java
// Flyweight Factory
class TreeFactory {
    private static final Map<String, TreeType> treeTypes = new HashMap<>();

    public static TreeType getTreeType(String species, String texture, String color) {
        String key = species + "-" + texture + "-" + color;
        if (!treeTypes.containsKey(key)) {
            treeTypes.put(key, new ConcreteTreeType(species, texture, color));
            System.out.println("Created new TreeType: " + key);
        }
        return treeTypes.get(key);
    }
}
```

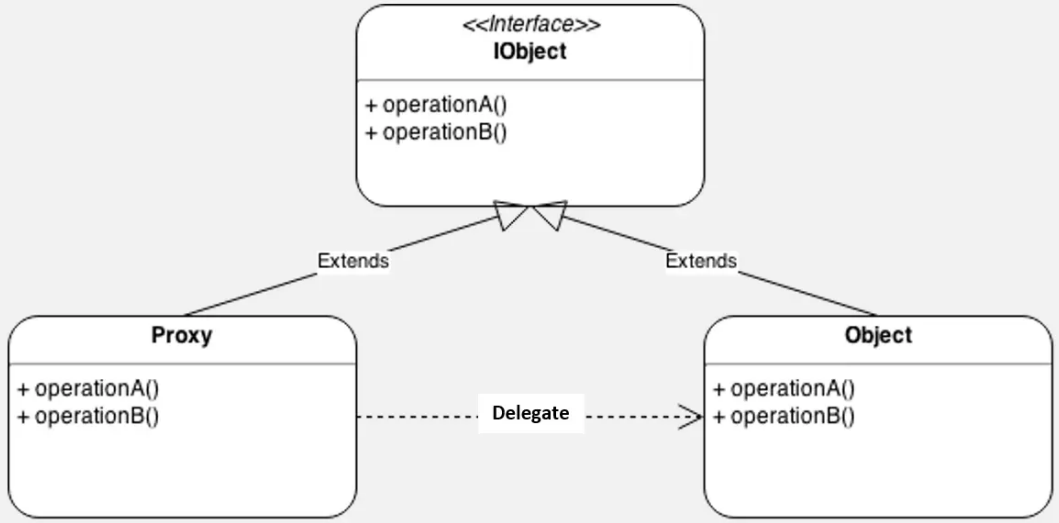
```java
// Cliente
public class FlyweightGameDemo {
    public static void main(String[] args) {
        List<Tree> forest = new ArrayList<>();

        // Crear 3 tipos de árboles, pero ubicados en muchas posiciones
        for (int i = 0; i < 1000; i++) {
            TreeType oakType = TreeFactory.getTreeType("Oak", "Rough", "Green");
            forest.add(new Tree(random(0, 1000), random(0, 1000), oakType));

            TreeType pineType = TreeFactory.getTreeType("Pine", "Smooth", "Dark Green");
            forest.add(new Tree(random(0, 1000), random(0, 1000), pineType));
        }

        // Dibujar el bosque
        forest.forEach(Tree::draw);
    }

    private static int random(int min, int max) {
        return new Random().nextInt(max - min) + min;
    }
}
```

UNIVERSIDAD Popular del cesar

Patrones Estructurales – Proxy

El patrón **Proxy** proporciona un sustituto o marcador de posición para otro objeto, controlando el acceso al objeto original y añadiendo funcionalidades adicionales como lazy loading, caching, o control de acceso.

Proxy pattern – Class diagram



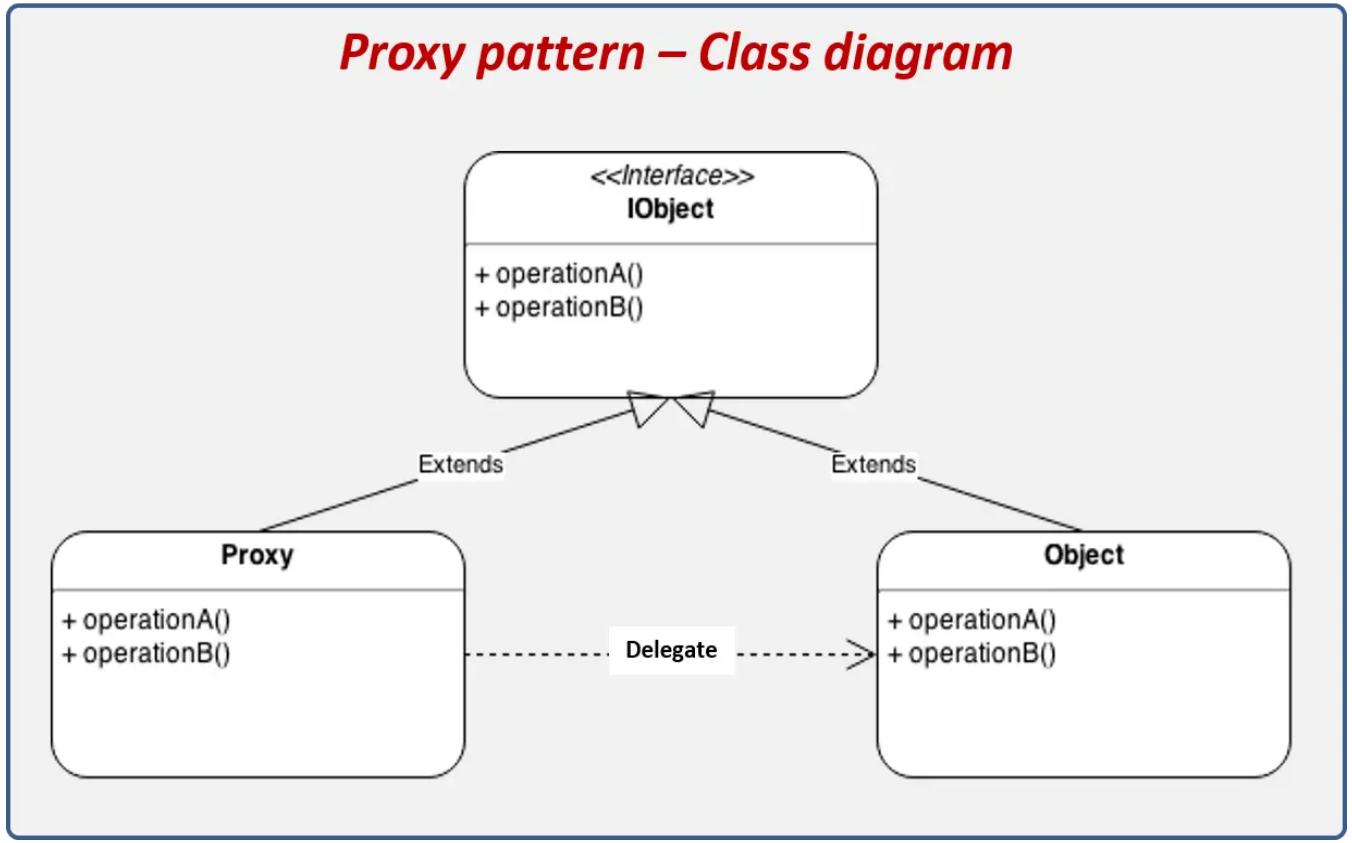
Propósitos de Proxy

- **Controlar la creación, inicialización o destrucción.**
- **Control de acceso:** verificar permisos antes de permitir la operación.
- **Carga diferida (Lazy Initialization):** crear el objeto real solo cuando es necesario.
- **Optimización de recursos:** mantener cargados solo los objetos realmente utilizados.
- **Caching de resultados:** evitar ejecuciones repetidas costosas.
- **Control de concurrencia:** sincronizar acceso en entornos multihilo.
- **Registro o logging de operaciones:** auditar el acceso a los métodos.

UNIVERSIDAD Popular del cesar

Patrones Estructurales – Proxy

Proxy: Clase que implementa IObject y delega la responsabilidad al Object, sin embargo, puede realizar una acción antes y después de llamar al Object.



IObject: Representa la interface común entre el Objet y el Proxy.

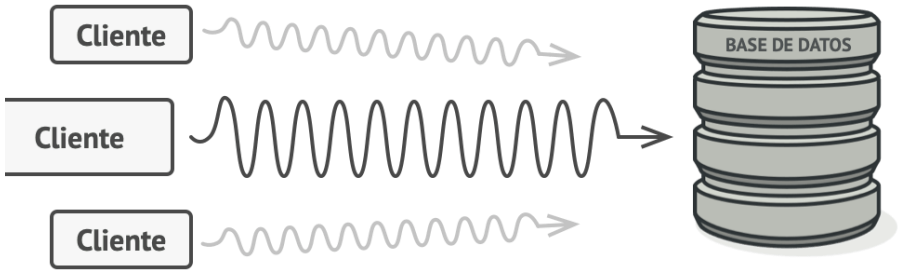
Object: Representa el objeto al que el cliente quiere ejecutar.

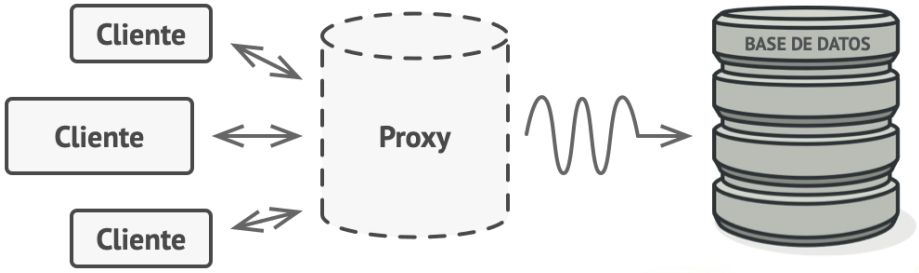
UNIVERSIDAD Popular del cesar

Patrones Estructurales – Proxy

Casos de Uso Comunes y Realistas

- **Carga diferida (Lazy Initialization).** Carga de imágenes, documentos, recursos multimedia
- **Seguridad (Protection Proxy).** Control de acceso en sistemas empresariales
- **Caching (Cache Proxy).** Servicios de consultas de bases de datos o APIs costosas
- **Auditoría y logging (Smart Proxy).** Registro de accesos, auditoría y monitoreo
- **Validación de parámetros.** Validación previa al acceso





UNIVERSIDAD Popular del cesar

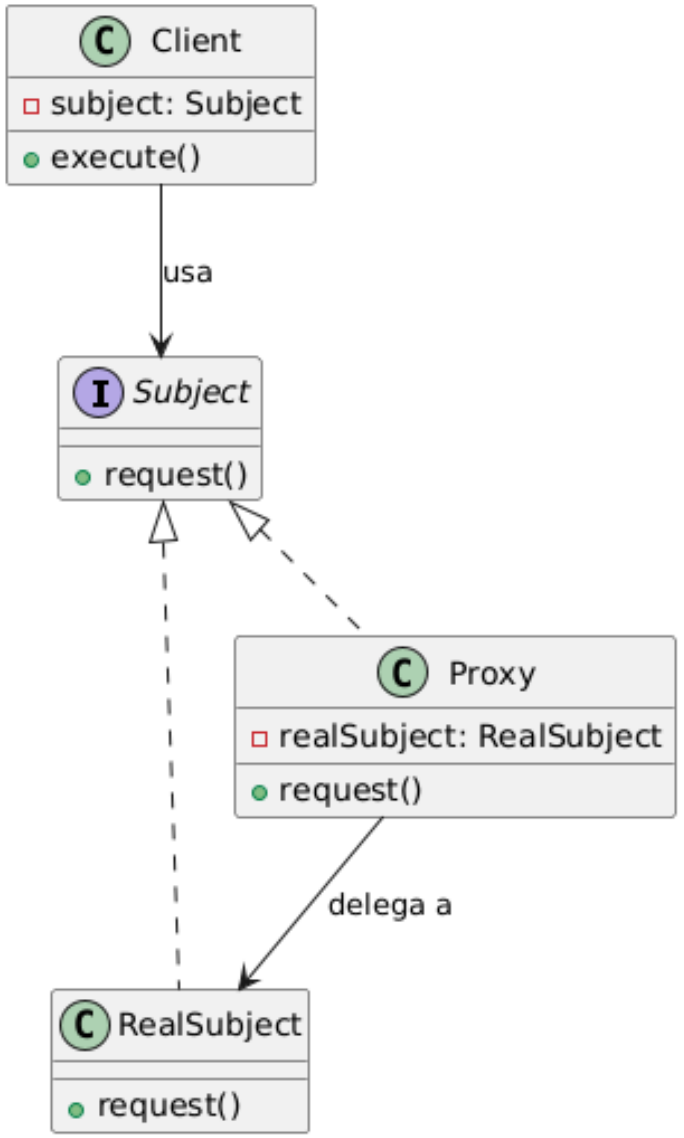
Patrones Estructurales – Proxy

Ventajas

- **Separación de responsabilidades:** el objeto real se enfoca en la lógica de negocio, el proxy controla el acceso.
- **Reducción de costos de recursos:** permite carga bajo demanda.
- **Mayor control de acceso y seguridad**. Facilidad para aplicar logging, caching, sincronización, etc.

Desventajas

- Mayor **complejidad** del diseño.
- **Sobrecarga de abstracción** si no es estrictamente necesario.
- **Puede complicar la depuración**, especialmente cuando hay muchos proxies anidados.
- **Riesgo de olvidar** diferenciar claramente cuándo se accede al proxy y cuándo al objeto real.



UNIVERSIDAD Popular del cesar

Patrones Estructurales – Proxy

```java
// Subject interface
public interface Image {
    void display();
    void loadFromDisk();
}

// Real Subject
public class RealImage implements Image {
    private String filename;
    private boolean loaded = false;

    public RealImage(String filename) {
        this.filename = filename;
    }

    public void loadFromDisk() {
        if (!loaded) {
            System.out.println("Loading image: " + filename);
            // Simulación de carga costosa
            try {
                Thread.sleep(2000);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
            loaded = true;
        }
    }

    public void display() {
        if (!loaded) {
            loadFromDisk();
        }
        System.out.println("Displaying image: " + filename);
    }
}
```

```java
// Proxy con cache y lazy loading
public class ImageProxy implements Image {
    private RealImage realImage;
    private String filename;
    private static Map<String, RealImage> cache = new HashMap<>();

    public ImageProxy(String filename) {
        this.filename = filename;
    }

    public void display() {
        if (realImage == null) {
            // Verificar cache primero
            realImage = cache.get(filename);
            if (realImage == null) {
                realImage = new RealImage(filename);
                cache.put(filename, realImage);
            }
        }
        realImage.display();
    }

    public void loadFromDisk() {
        if (realImage == null) {
            realImage = cache.get(filename);
            if (realImage == null) {
                realImage = new RealImage(filename);
                cache.put(filename, realImage);
            }
        }
        realImage.loadFromDisk();
    }
}
```

```java
// Protection Proxy
public class ProtectedImageProxy implements Image {
    private RealImage realImage;
    private String filename;
    private String userRole;

    public ProtectedImageProxy(String filename, String userRole) {
        this.filename = filename;
        this.userRole = userRole;
    }

    public void display() {
        if (hasAccess()) {
            if (realImage == null) {
                realImage = new RealImage(filename);
            }
            realImage.display();
        } else {
            System.out.println("Access denied for user role: " + userRole);
        }
    }

    public void loadFromDisk() {
        if (hasAccess()) {
            if (realImage == null) {
                realImage = new RealImage(filename);
            }
            realImage.loadFromDisk();
        } else {
            System.out.println("Access denied for user role: " + userRole);
        }
    }

    private boolean hasAccess() {
        return "ADMIN".equals(userRole) || "USER".equals(userRole);
    }
}
```

UNIVERSIDAD Popular del cesar

Patrón de diseño – Recursos

Libros:

- ***"Patrones de diseño. Elementos de software orientado a objetos reutilizables"***
  Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides

- ***"Sumérgete en los patrones de diseño"***
  Alexander Shvets

- ***"Introducción a los patrones de diseño. Un enfoque practico"***
  Oscar Blancharte

- ***"Patrones de diseño en java. Los 23 modelos de diseño: descripción y soluciones ilustradas "***
  Lauren Debrauwer

UNIVERSIDAD
Popular del cesar