



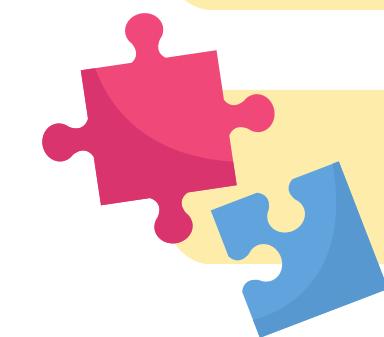
PATRONES DE DISEÑO

EN JAVA

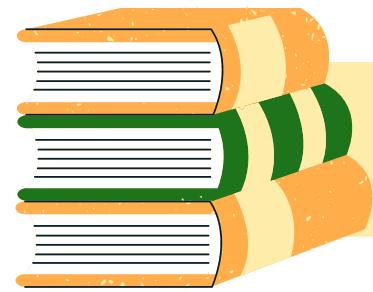
PATRONES ESTRUCTURALES



ADAPTER



DECORATOR



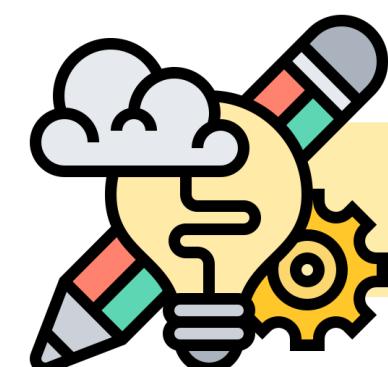
COMPOSITE



FACADE



FLYWEIGHT



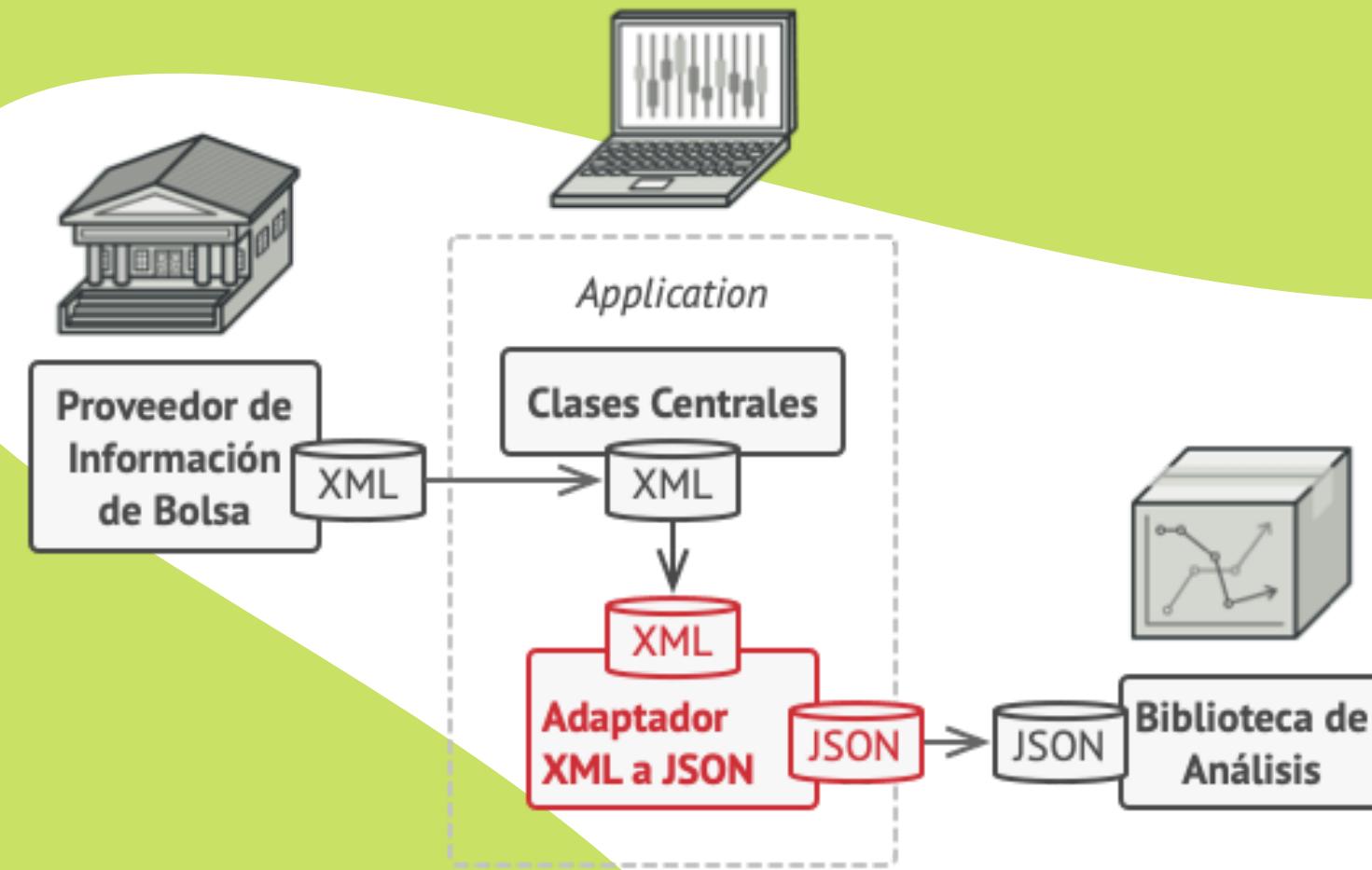
PROXY

ADAPTER

I. PROPOSITO

Adapter es un patrón de diseño estructural que permite la colaboración entre objetos con interfaces incompatibles.

El adaptador se encarga de traducir las solicitudes de uno de los objetos en términos que el otro objeto pueda entender.



ADAPTER - SOLUCION

El patrón de diseño del Adaptador implica que un objeto envuelva a otro con el fin de ocultar la complejidad de la conversión.

Se trata de un objeto que convierte la interfaz de un objeto, de forma que otro objeto pueda comprenderla.

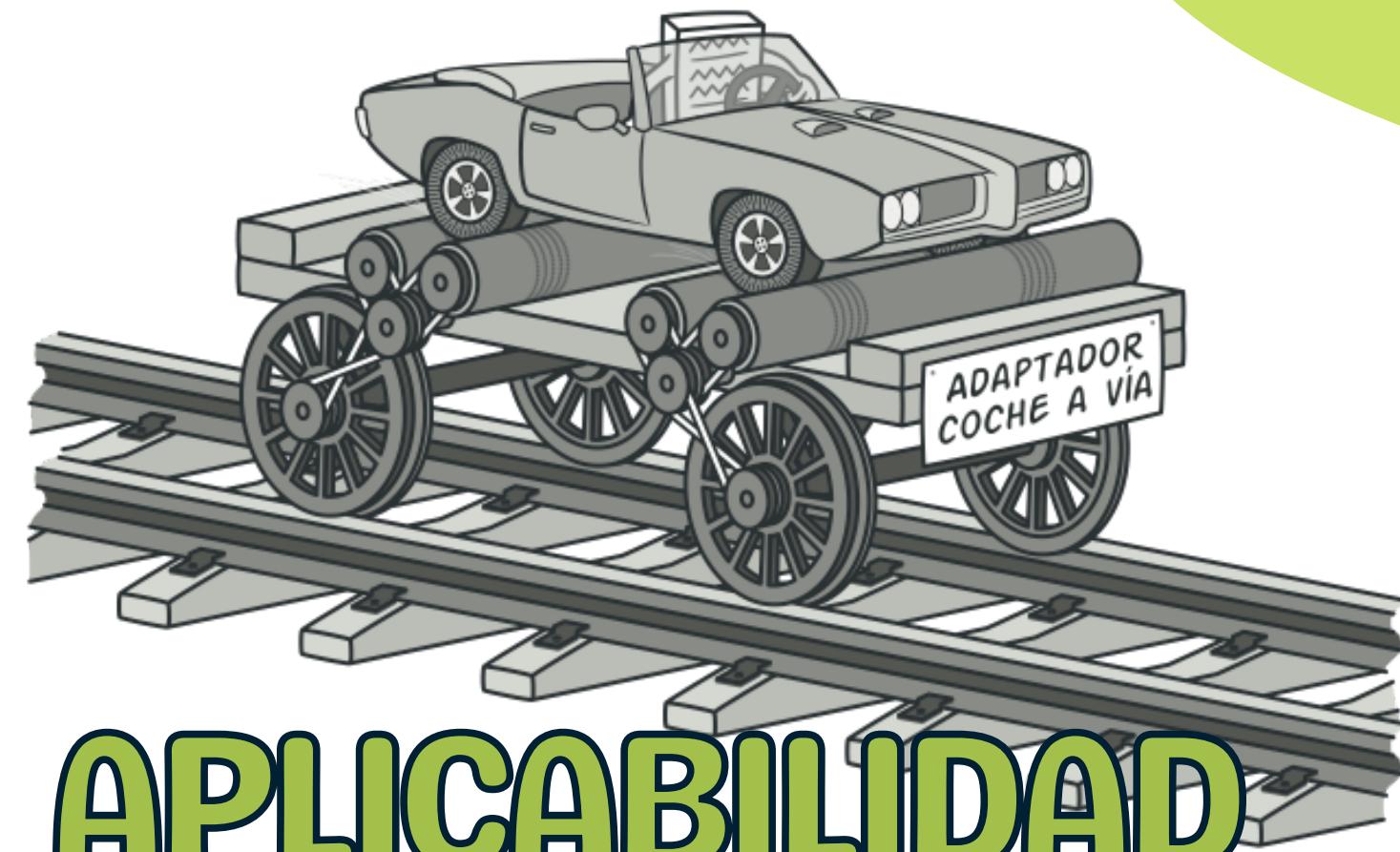
ANALOGIA AL MUNDO REAL



Adapter



Adaptee



APLICABILIDAD

Usa la clase adaptadora cuando quieras usar una clase existente que no sea compatible con otro código.

Usa el patrón cuando quieras reutilizar subclases existentes que no tenga alguna función que no pueda añadirse

COMO IMPLEMENTARLO

1. **Primero que nada debes tener en cuenta tener dos o mas clases incompatibles**
2. **Declara la interfaz y describe el modo en que la primera clase se comunican con la segunda.**
3. **Crea la clase adaptadora y haz que siga la interfaz con el primera clase.**
4. **Añade un campo a la clase adaptadora para almacenar una referencia al objeto . Lo común es inicializar este campo a través del constructor**
5. **Uno por uno, implementa todos los métodos de la interfaz primaria en la clase adaptadora**

6. **Las clases cliente deberán utilizar la clase adaptadora a través de la interfaz con el cliente.**

PROS

- Separar el código de la lógica principal del programa implica mantener una distinción entre como los datos se manejan y se presentan

US.

- Puedes introducir nuevos tipos de adaptadores al programa sin descomponer

CONTRAS

- La complejidad general del código aumenta, ya que debes introducir un grupo de nuevas interfaces y clases.

RELACIONES

Bridge suele diseñarse por anticipado, pero, Adapter se utiliza habitualmente con una aplicación existente

Adapter cambia una interfaz de un objeto, decorator mejora la misma

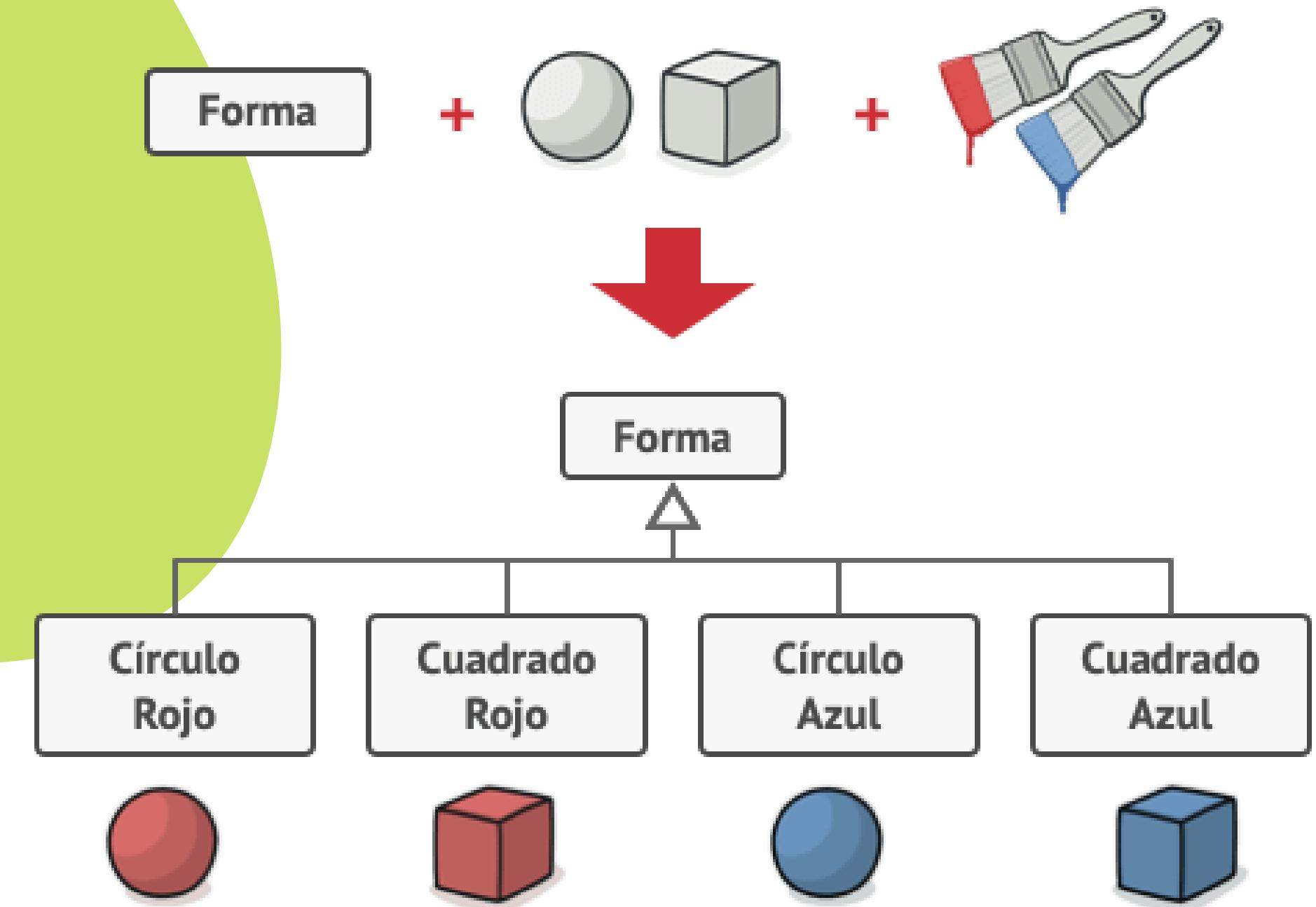
Facade, crea interfaces por objetos existentes, adaptar hace que la misma interfaz sea usable

Bridge, State, Strategy (y, hasta cierto punto, Adapter) tienen estructuras similares ya que, todos se basan delegar trabajo a otros objetos.

BRIDGE

I. PROPOSITO

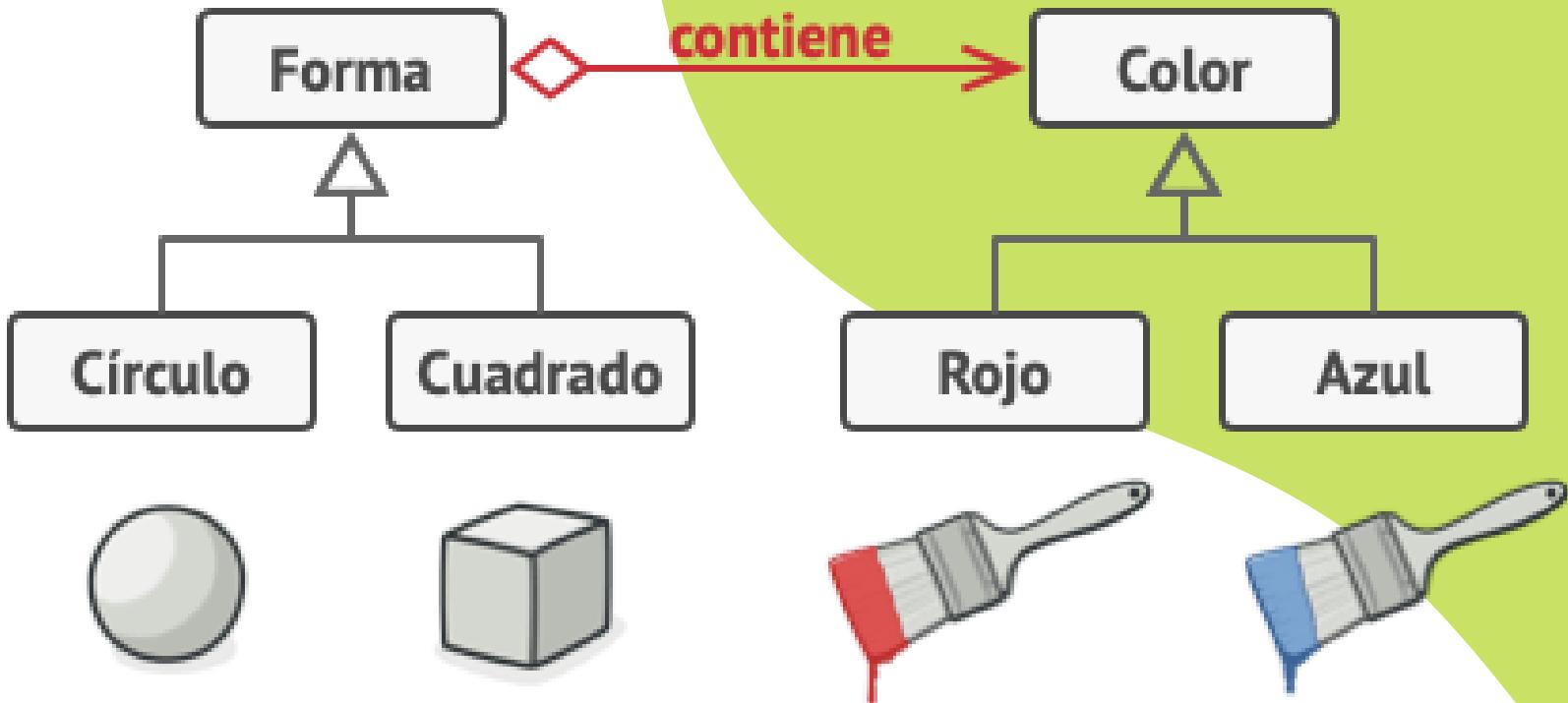
Bridge es un patrón de diseño que te permite dividir una clase grande, o un grupo de clases relacionadas, en dos separadas (abs e imp) que pueden desarrollarse independientemente la una de la otra, pero no pierden la relación.



2. PROBLEMA

SOLUCION

- Pasando de la herencia a la composición del objeto.
- Extrae una de las dimensiones a una jerarquía de clases separada



APLICABILIDAD

Usa el patrón Bridge cuando quieras dividir una clase monolítica que tenga muchas variantes de una funcionalidad

Utiliza el patrón cuando necesites extender una clase en varias dimensiones

COMO IMPLEMENTARLO

I. Identifica las dimensiones principales de tus clases,

3. Crea clases de implementación concretas

5. Si hay múltiples variantes de lógica, crea abstracciones extendiendo la clase base de abstracción para cada variante.

2. Define los métodos necesarios en la clase de abstracción

4. Dentro de la clase de abstracción, agrega un campo de referencia para el tipo de implementación.

En el código del cliente, pasa un objeto de implementación al constructor de la abstracción para asociarlos

PROS

- Puedes crear clases y aplicaciones independientes
- Puedes introducir nuevas abstracciones e implementaciones
- Puedes centrarte en la lógica de alto nivel en la abstracción y en detalles de la plataforma en la implementación.

CONTRAS

- Puede ser que el código se complique si aplicas a una clase muy relacionada.

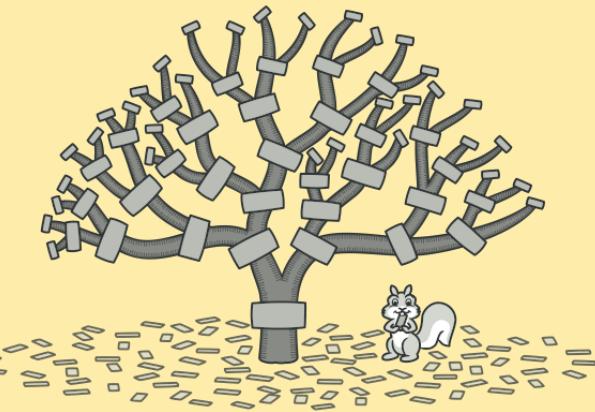
RELACIONES

Bridge suele diseñarse por anticipado, pero, Adapter se utiliza habitualmente con una aplicación existente

Puedes utilizar Abs. Factory junto a Bridge. Este emparejamiento resulta útil cuando algunas abstracciones sólo pueden funcionar con implementaciones específicas.

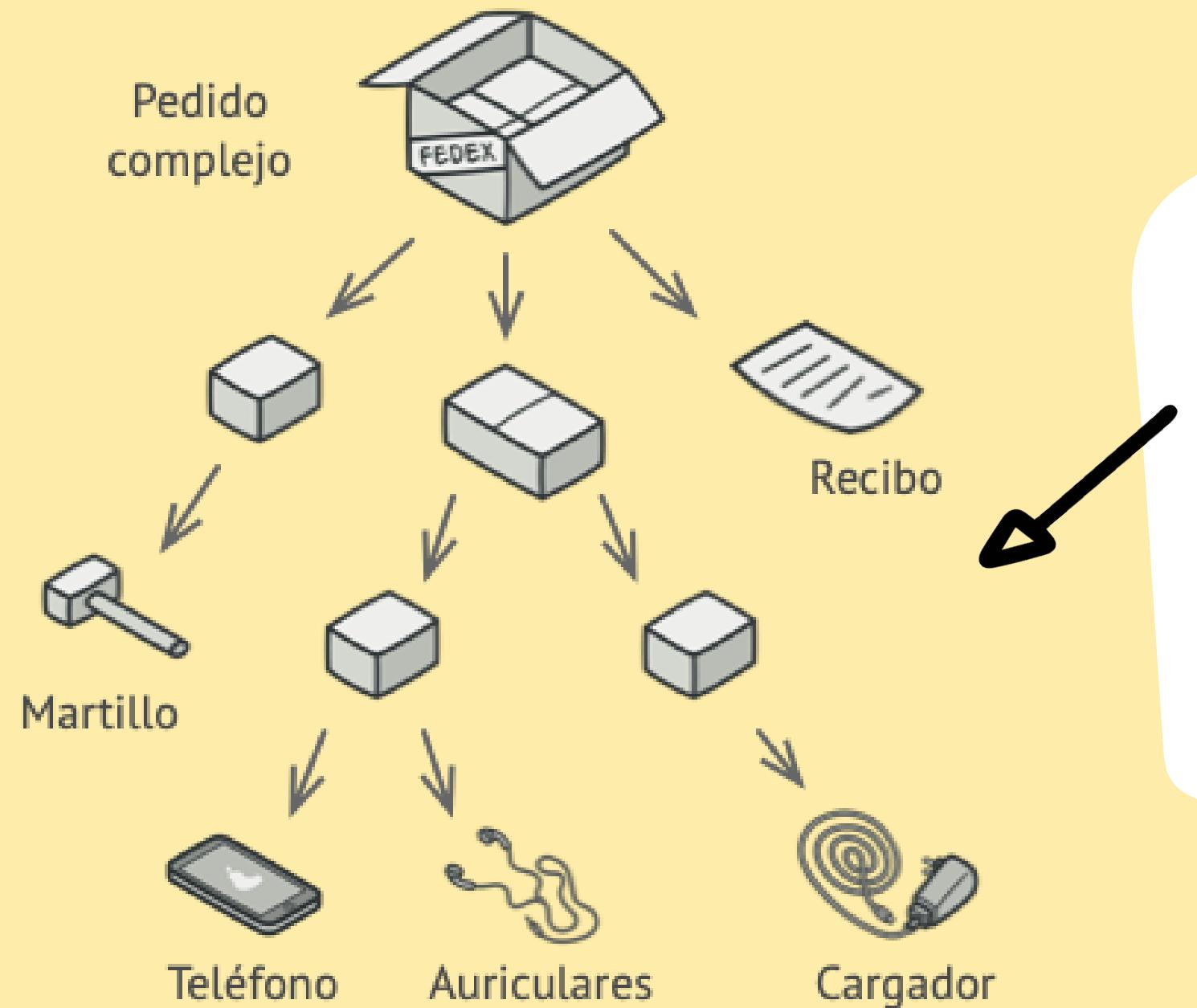
Bridge, State, Strategy (y, hasta cierto punto, Adapter) tienen estructuras similares ya que, todos se basan delegar trabajo a otros objetos.

COMPOSITE



PROPOSITO

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



PROBLEMA

Su uso sólo tiene sentido cuando el modelo central de tu aplicación puede representarse en forma de árbol.



"UN PEDIDO PUEDE INCLUIR VARIOS PRODUCTOS EMPAQUETADOS EN CAJAS, QUE A SU VEZ ESTÁN EMPAQUETADOS EN CAJAS MÁS GRANDES Y ASÍ SUCESIUAMENTE. LA ESTRUCTURA SE ASEMEJA A UN ÁRBOL BOCA ABAJO."

COMPOSITE

SOLUCIÓN

Sugiere trabajar con un método que:

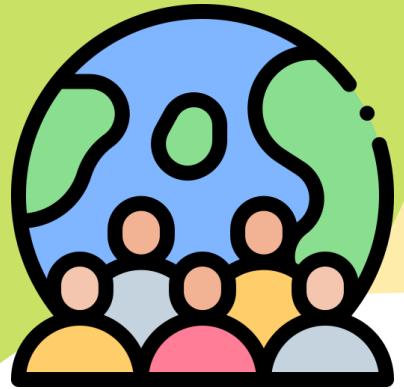
- Para un producto, sencillamente devuelve el precio del producto.

- Para una caja, recorre cada artículo que contiene la caja.

- Si uno de esos artículos fuera una caja más pequeña, repasara su contenido y así sucesivamente



"EL PATRÓN COMPOSITE TE PERMITE EJECUTAR UN COMPORTAMIENTO DE FORMA RECURSIVA SOBRE TODOS LOS COMPONENTES DE UN ÁRBOL DE OBJETOS."



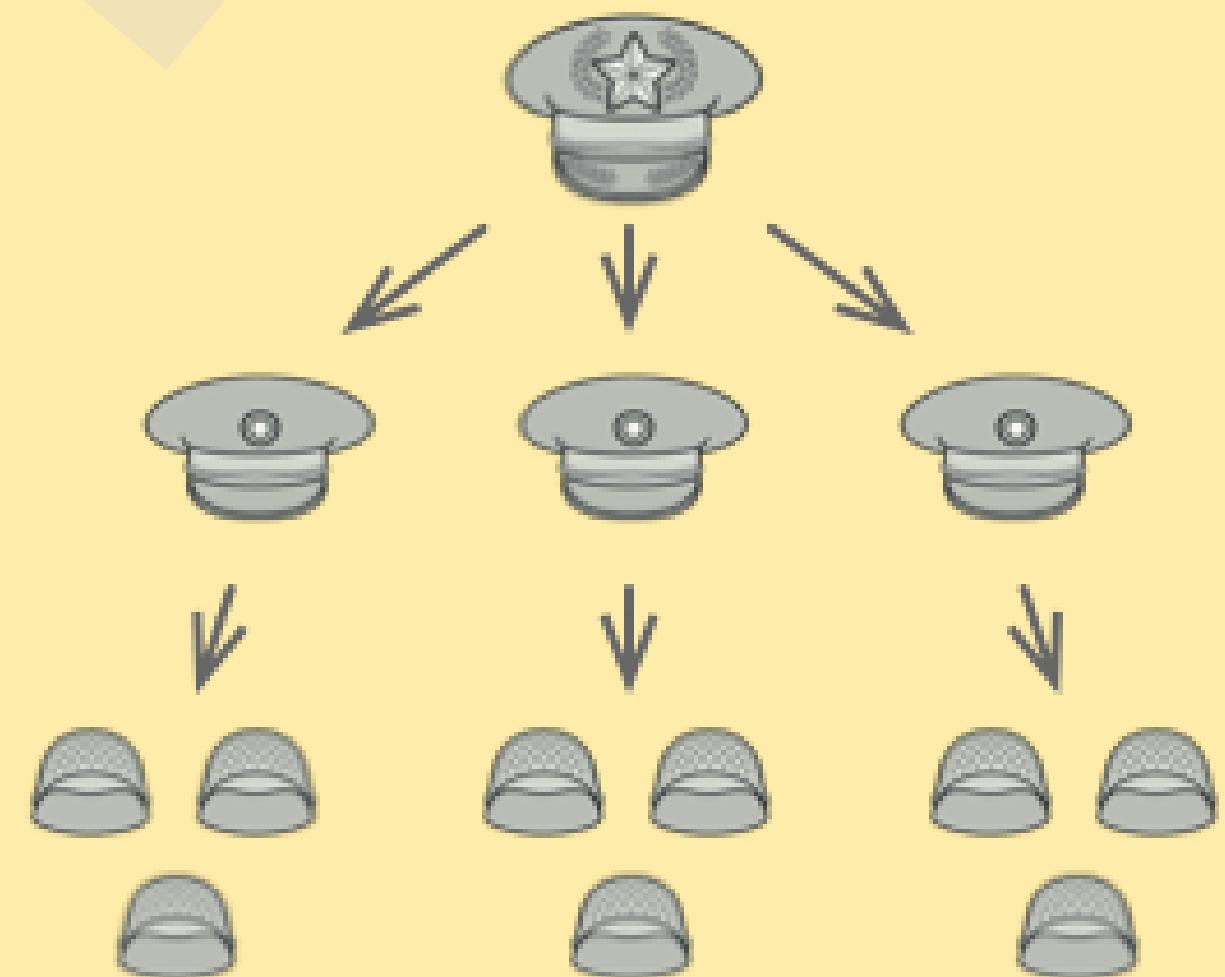
ANALOGÍA DEL MUNDO REAL

Los ejércitos de la mayoría de países se estructuran como jerarquías.

Un ejército está formado por varias divisiones.

1. Una división es un grupo de brigadas
2. una brigada está formada por pelotones,
3. Un pelotón en escuadrones.
4. Un escuadrón en grupos de soldados.

Las órdenes se dan en la parte superior de la jerarquía y se pasan hacia abajo.



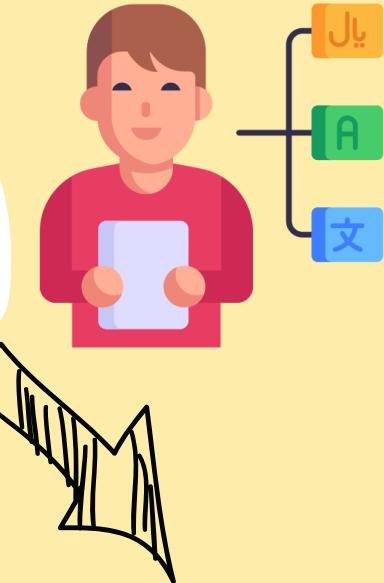
COMPOSITE

APLICABILIDAD

- Cuando tengas que implementar una estructura de objetos con forma de árbol.
- Cuando quieras que el código cliente trate elementos simples y complejos de la misma forma.



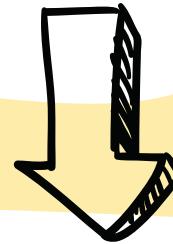
CÓMO IMPLEMENTARLO



1. Asegurarse que el modelo puede representarse con estructura árbol e intenta dividirlo en elementos simples y contenedores.



3. Crea una clase hoja para representar elementos simples.



4. Crea una clase contenedora para representar elementos complejos. Incluye un campo matriz en esta clase para almacenar referencias a subelementos.



5. Define los métodos para añadir y eliminar elementos hijos dentro del contenedor.

COMPOSITE

PROS Y CONTRAS

Puedes trabajar con estructuras de árbol complejas con mayor comodidad

Puedes introducir nuevos tipos de elemento en la aplicación sin descomponer el código

Puede resultar difícil proporcionar una interfaz común para clases cuya funcionalidad difiere demasiado.

RELACIONES CON OTROS PATRONES

- Puedes utilizar Builder al crear árboles Composite complejos porque puedes programar sus pasos de construcción para que funcionen de forma recursiva.
- Chain of Responsibility cuando un componente hoja recibe una solicitud, puede pasarla a lo largo de la cadena de todos los componentes padre hasta la raíz del árbol de objetos.
- Iteradores para recorrer árboles Composite.
- Visitor para ejecutar una operación sobre un árbol Composite entero.
- Puedes implementar nodos de hoja compartidos del árbol Composite como Flyweights para ahorrar memoria RAM.
- Composite y Decorator tienen diagramas de estructura similares ya que ambos se basan en la composición recursiva

DECORATOR

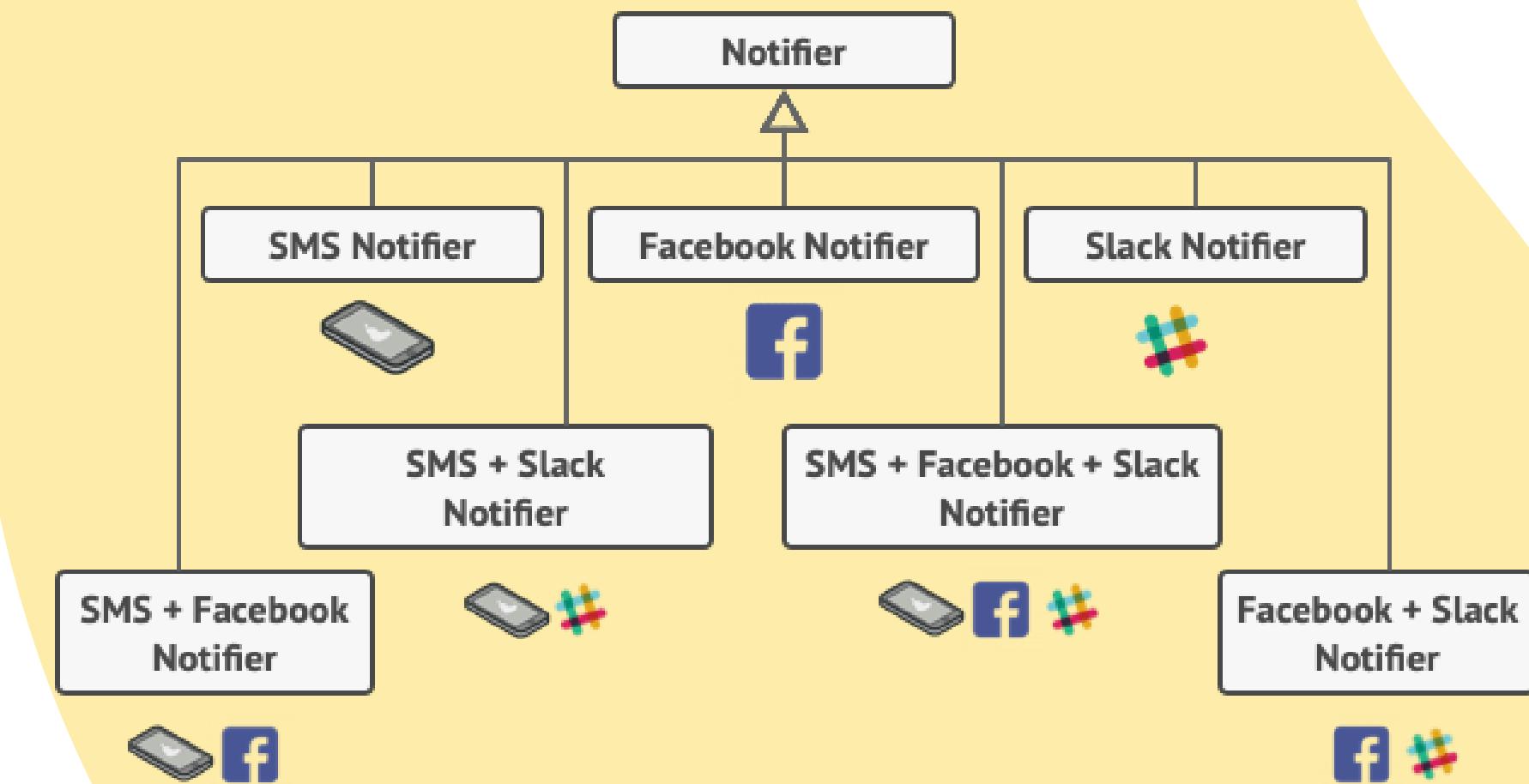
PROPOSITO

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



PROBLEMA

El problema viene cuando necesitamos alterar la funcionalidad de un objeto, lo mas facil suele ser extender una clase y combinar metodos, lo que resulta en un codigo inflado.



DECORATOR

SOLUCIÓN

Una de las formas de superar las limitaciones de la herencia es usando la "Agregación" o la "Composición"

Con esto un objeto puede utilizar el comportamiento de varias clases con referencias a varios objetos, delegándoles todo tipo de tareas.



un objeto tiene una referencia a otro y le delega parte del trabajo, mientras que con la herencia, el propio objeto hereda el comportamiento de su superclase.

```
stack = new Notifier()
if (facebookEnabled)
    stack = new FacebookDecorator(stack)
if (slackEnabled)
    stack = new SlackDecorator(stack)

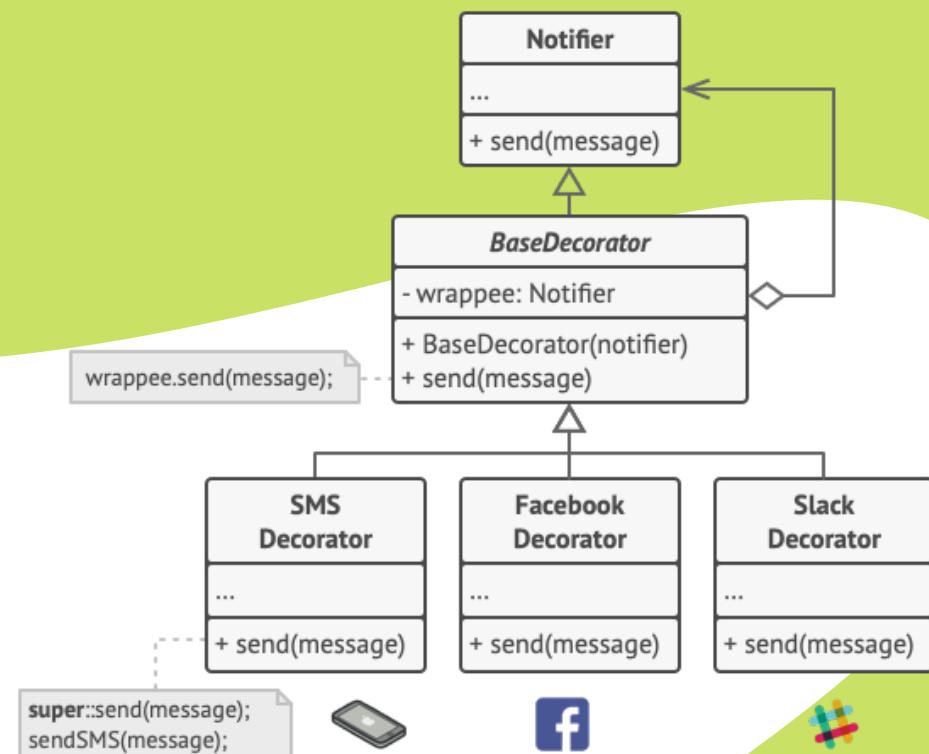
app.setNotifier(stack)
```

```
Application
- notifier: Notifier
+ setNotifier(notifier)
+ doSomething()
```

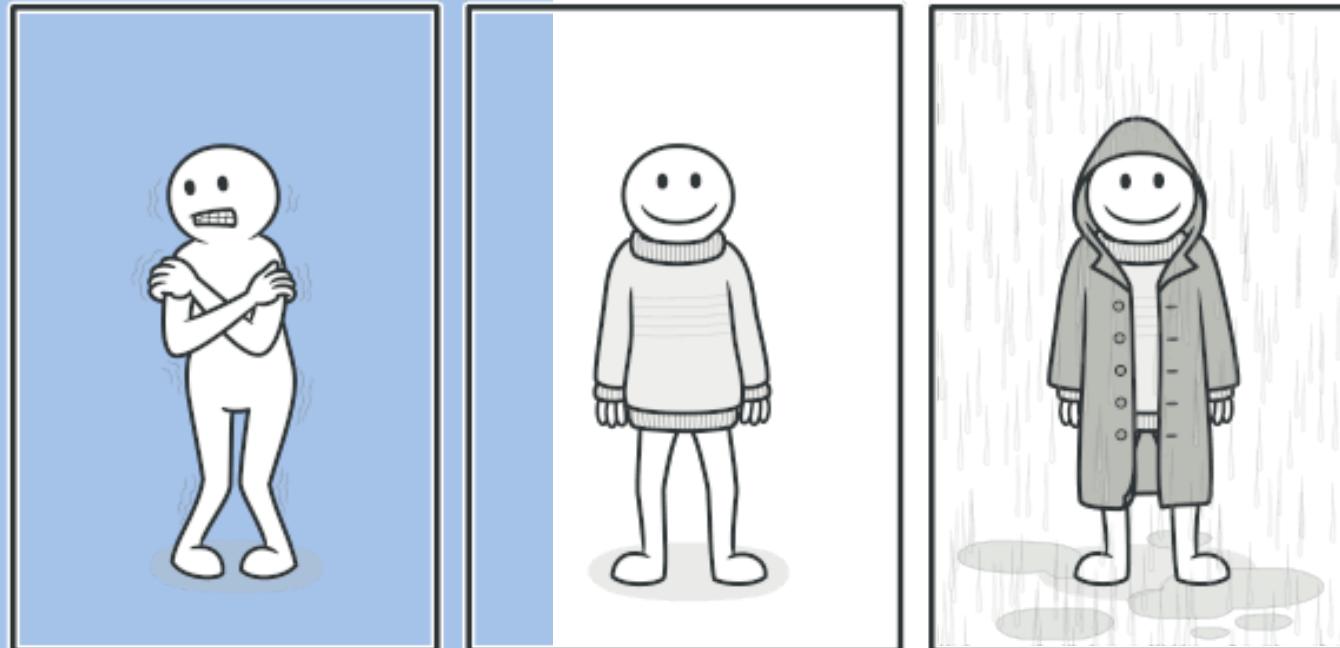
notifier.send("¡Alerta!")
// Email → Facebook → Slack



El wrapper o decorator contiene el mismo grupo de métodos que el objetivo y le delega todas las solicitudes que recibe. No obstante, puede alterar el resultado haciendo algo antes o después de pasar la solicitud al objetivo.



DECORATOR



ANALOGÍA EN EL MUNDO REAL

Vestir ropa es un ejemplo del uso de decoradores. Todas estas prendas “extienden” tu comportamiento básico pero no son parte de ti, y puedes quitarte fácilmente cualquier prenda cuando lo deseas.

APLICABILIDAD

- Usar cuando necesites asignar funcionalidades adicionales a objetos durante el tiempo de ejecución sin descomponer el código.
- Usar cuando resulte extraño o no sea posible extender el comportamiento de un objeto utilizando la herencia.



DECORATOR

CÓMO IMPLEMENTARLO

3. Crea una clase concreta de componente y define en ella el comportamiento base.

6. Crea decoradores concretos extendiéndolos a partir de la decoradora base.

1. Asegurate de que puedes representar con un componente con varias capas.

4. Crea una clase base decoradora. Debe tener un campo para almacenar una referencia a un objeto envuelto.

7. El código cliente debe ser responsable de crear decoradores y componerlos del modo que el cliente necesite.

2. Elegir los métodos más comunes y crear una interfaz con estos.

5. Asegúrate de que todas las clases implementan la interfaz de componente.

DECORATOR

PROS

- Puedes extender el comportamiento de un objeto sin crear una nueva subclase.
- Puedes añadir o eliminar responsabilidades de un objeto durante el tiempo de ejecución.
- Puedes combinar varios comportamientos envolviendo un objeto con varios decoradores.
- Principio de responsabilidad única. Puedes dividir una clase monolítica que implementa muchas variantes posibles de comportamiento, en varias clases más pequeñas.

CONTRAS

- Resulta difícil eliminar un wrapper específico de la pila de wrappers.
- Es difícil implementar un decorador de tal forma que su comportamiento no dependa del orden en la pila de decoradores.
- El código de configuración inicial de las capas pueden tener un aspecto desagradable.

DECORATOR

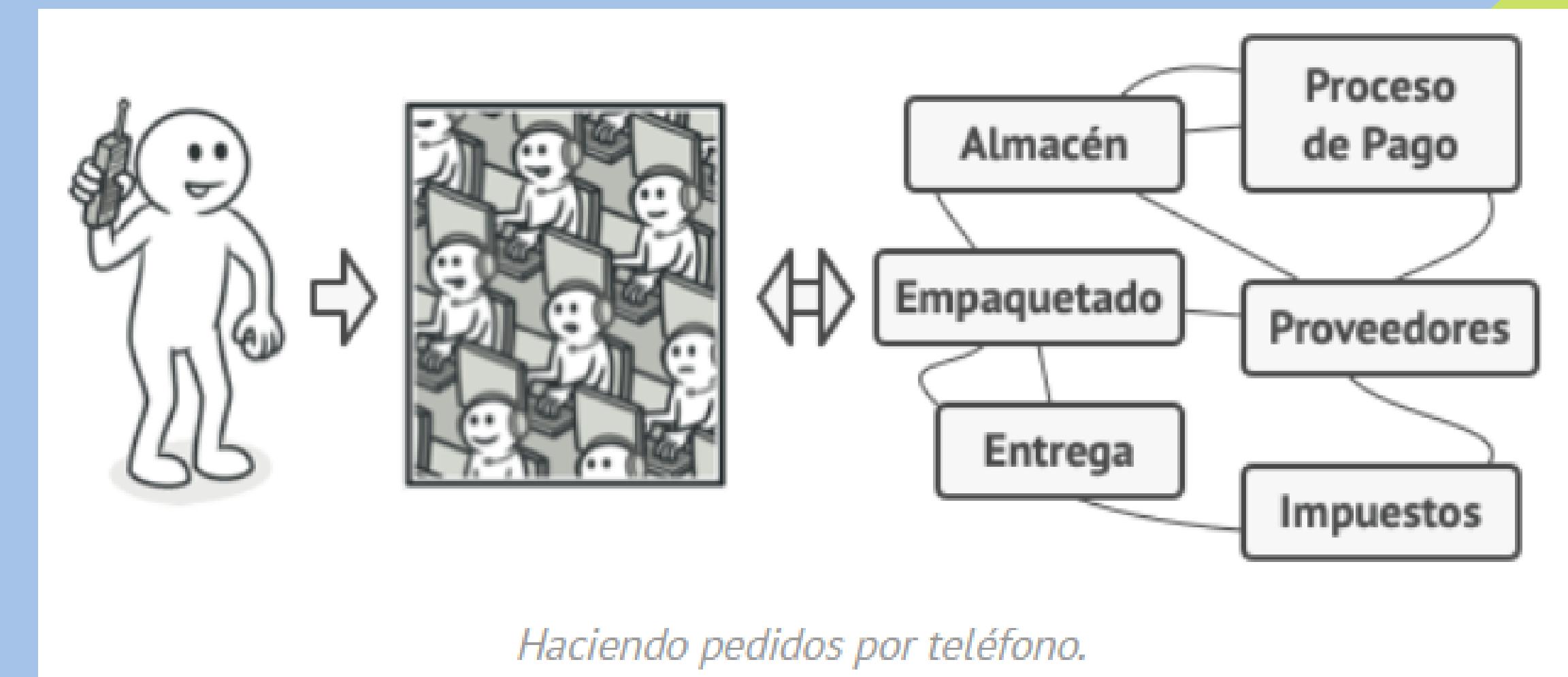
Relaciones con otros patrones

- Adapter cambia la interfaz de un objeto existente mientras que Decorator mejora un objeto sin cambiar su interfaz.
- Proxy le proporciona la misma interfaz y Decorator le proporciona una interfaz mejorada.
- Chain of Responsibility y Decorator. Los manejadores de CoR pueden ejecutar operaciones arbitrarias con independencia entre sí. También pueden dejar de pasar la solicitud en cualquier momento. Por otro lado, varios decoradores pueden extender el comportamiento del objeto manteniendo su consistencia con la interfaz base. Además, los decoradores no pueden romper el flujo de la solicitud.
- Composite y Decorator: Decorator añade responsabilidades adicionales al objeto envuelto, mientras que Composite se limita a “recapitular” los resultados de sus hijos.
- Composite y Decorator pueden beneficiarse de Prototype. Aplicar el patrón te permite clonar estructuras complejas.
- Decorator te permite cambiar la piel de un objeto, mientras que Strategy te permite cambiar sus entrañas.
- Decorator y Proxy, Proxy gestiona el ciclo de vida de su objeto de servicio por su cuenta, mientras que la composición de los Decoradores siempre está controlada por el cliente.

FACADE

FACADE ES UN PATRÓN DE DISEÑO ESTRUCTURAL QUE PROPORCIONA UNA INTERFAZ SIMPLIFICADA A UNA BIBLIOTECA, UN FRAMEWORK O CUALQUIER OTRO GRUPO COMPLEJO DE CLASES.

- PROBLEMA
- SOLUCIÓN
- ANALOGÍA AL MUNDO REAL
- APLICABILIDAD
- CÓMO IMPLEMENTARLO
- PROS Y CONTRAS

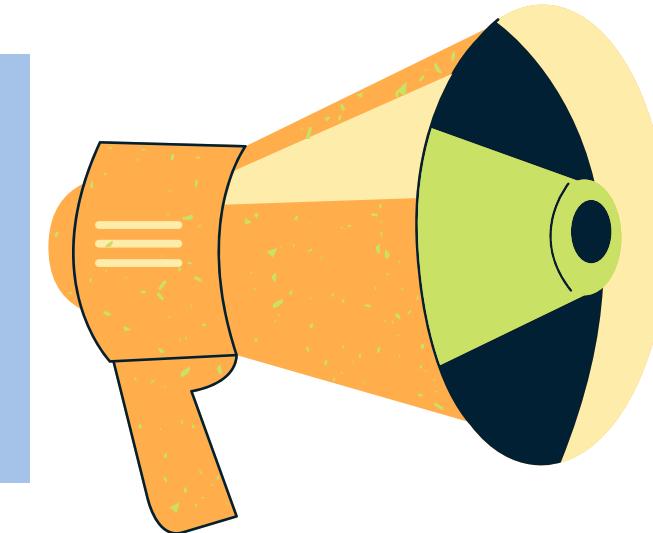


RELACIONES CON OTROS PATRONES

FAÇADE DEFINE UNA NUEVA INTERFAZ PARA OBJETOS EXISTENTES, MIENTRAS QUE ADAPTER INTENTA HACER QUE LA INTERFAZ EXISTENTE SEA UTILIZABLE.

FAÇADE Y MEDIATOR TIENEN TRABAJOS SIMILARES: AMBOS INTENTAN ORGANIZAR LA COLABORACIÓN ENTRE MUCHAS CLASES ESTRECHAMENTE ACOPLADAS.

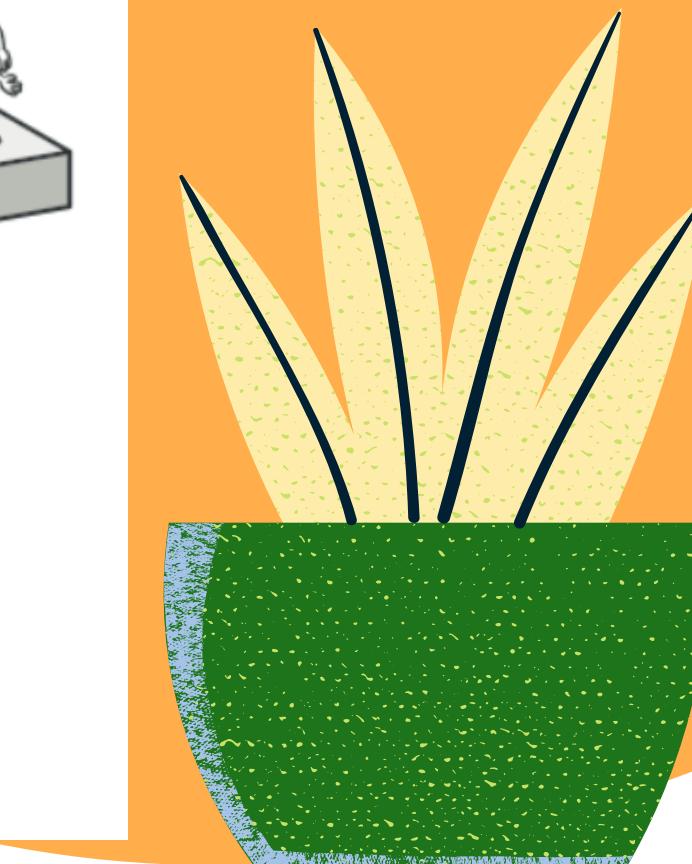
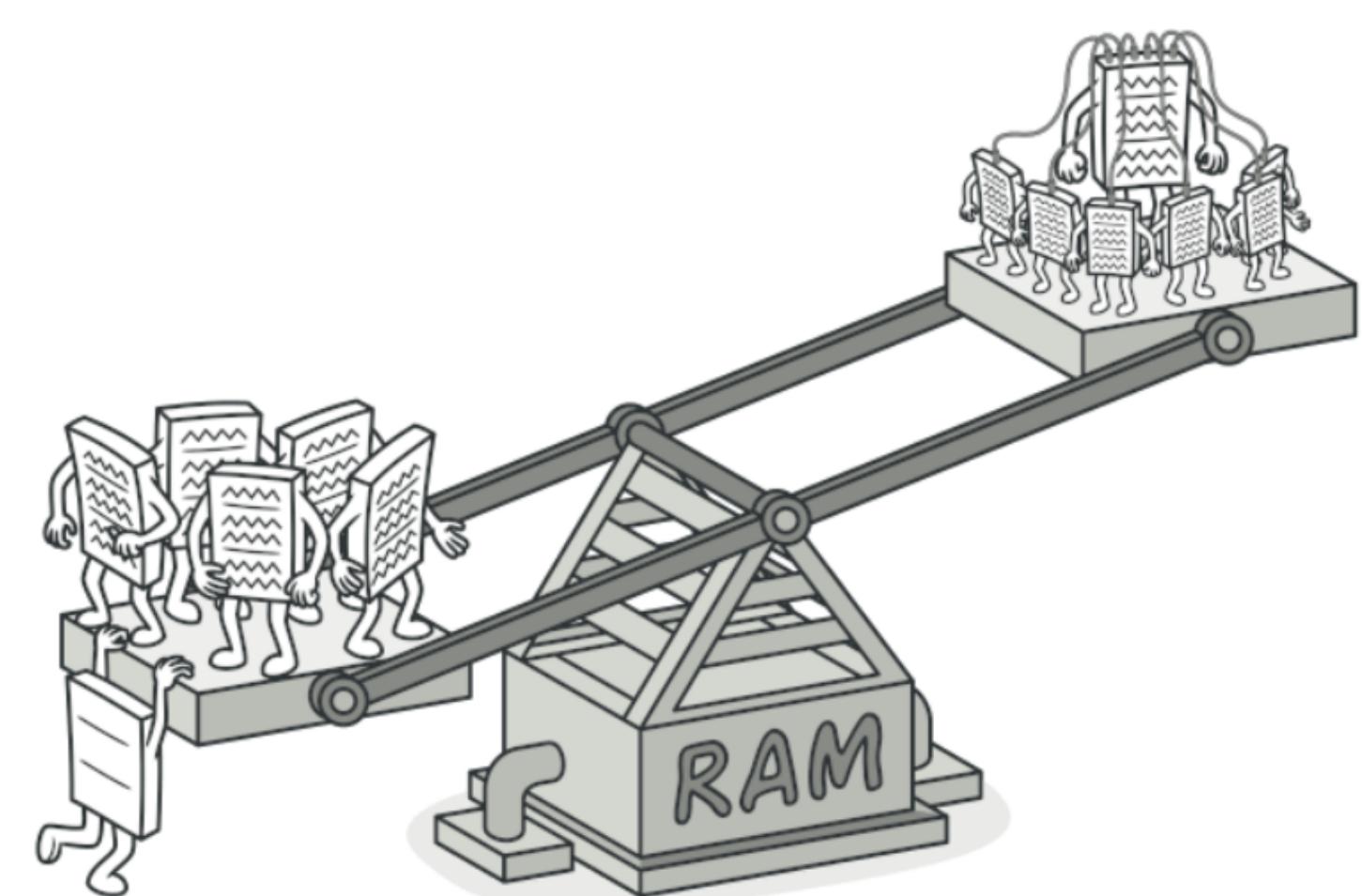
FLYWEIGHT MUESTRA CÓMO CREAR MUCHOS PEQUEÑOS OBJETOS, MIENTRAS QUE FAÇADE MUESTRA CÓMO CREAR UN ÚNICO OBJETO QUE REPRESENTE UN SUBSISTEMA COMPLETO.



FLYWEIGHT

FLYWEIGHT ES UN PATRÓN DE DISEÑO ESTRUCTURAL QUE TE PERMITE MANTENER MÁS OBJETOS DENTRO DE LA CANTIDAD DISPONIBLE DE RAM COMPARTIENDO LAS PARTES COMUNES DEL ESTADO ENTRE VARIOS OBJETOS EN LUGAR DE MANTENER TODA LA INFORMACIÓN EN CADA OBJETO.

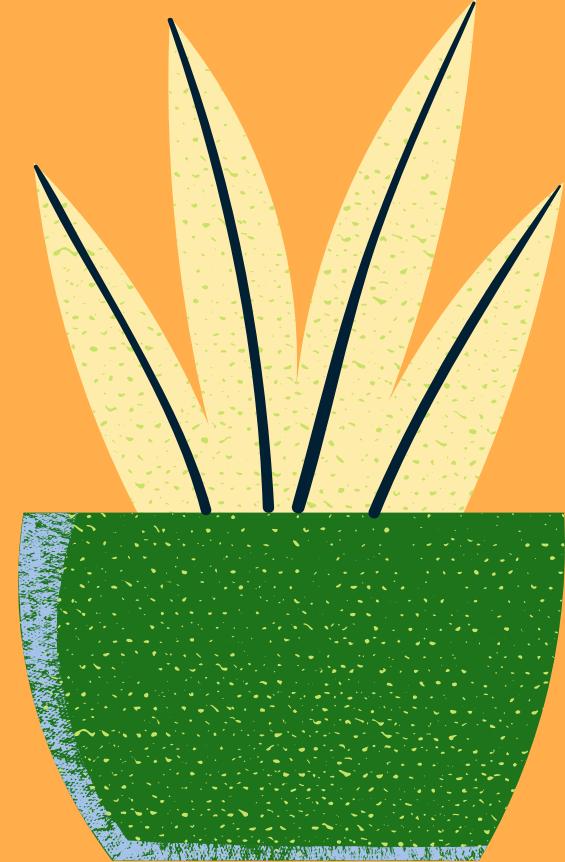
- PROBLEMA
- SOLUCIÓN
- ANALOGÍA AL MUNDO REAL
- APLICABILIDAD
- CÓMO IMPLEMENTARLO
- PROS Y CONTRAS



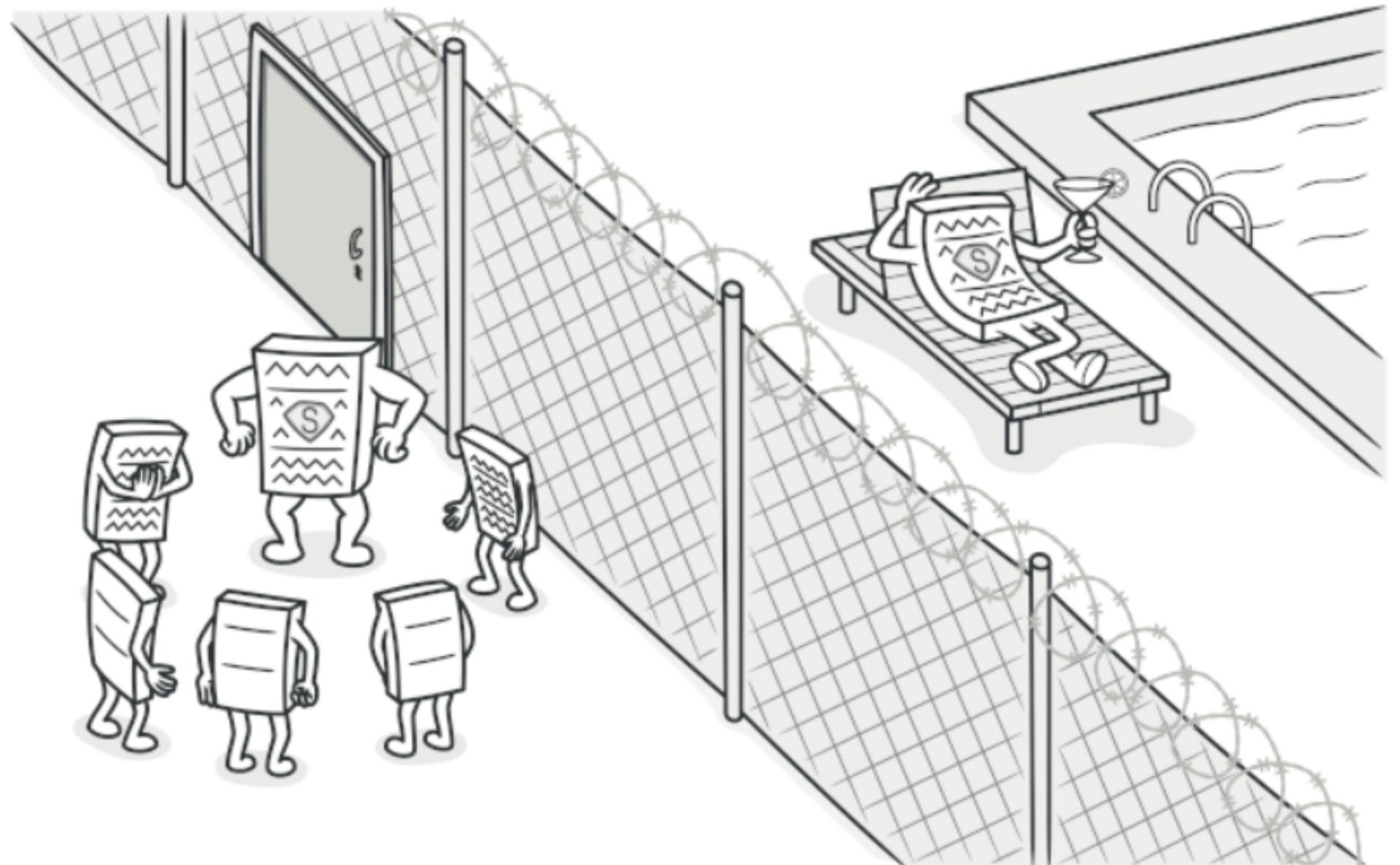
RELACIONES CON OTROS PATRONES

PUEDES IMPLEMENTAR NODOS DE HOJA COMPARTIDOS DEL ÁRBOL COMPOSITE COMO FLYWEIGHTS PARA AHORRAR MEMORIA RAM.

FLYWEIGHT PODRÍA ASEMEJARSE A SINGLETON SI DE ALGÚN MODO PUDIERAS REDUCIR TODOS LOS ESTADOS COMPARTIDOS DE LOS OBJETOS A UN ÚNICO OBJETO FLYWEIGHT.



PROXY



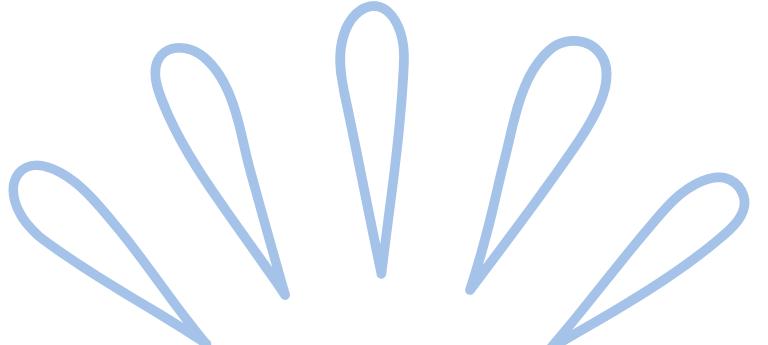
PROXY ES UN PATRÓN DE DISEÑO ESTRUCTURAL QUE TE PERMITE PROPORCIONAR UN SUSTITUTO O MARCADOR DE POSICIÓN PARA OTRO OBJETO. UN PROXY CONTROLA EL ACCESO AL OBJETO ORIGINAL, PERMITIÉNDOTE HACER ALGO ANTES O DESPUÉS DE QUE LA SOLICITUD LLEGUE AL OBJETO ORIGINAL.

- PROBLEMA
- SOLUCIÓN
- ANALOGÍA AL MUNDO REAL
- APLICABILIDAD
- CÓMO IMPLEMENTARLO
- PROS Y CONTRAS

RELACIONES CON OTROS PATRONES



- ADAPTER PROPORCIONA UNA INTERFAZ DIFERENTE AL OBJETO ENVUELTO, PROXY LE PROPORCIONA LA MISMA INTERFAZ Y DECORATOR LE PROPORCIONA UNA INTERFAZ MEJORADA.
- FAÇADE ES SIMILAR A PROXY EN EL SENTIDO DE QUE AMBOS PUEDEN ALMACENAR TEMPORALMENTE UNA ENTIDAD COMPLEJA E INICIALIZARLA POR SU CUENTA.



**¡MUCHAS
GRACIAS!**