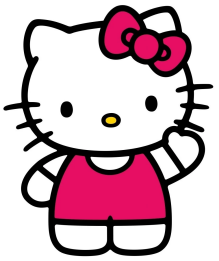
Patrones de Diseño

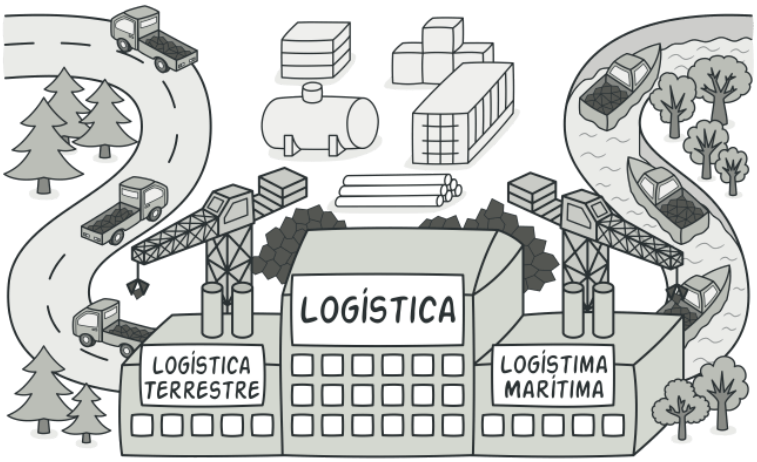


Patrones Creacionales

Proporcionan varios mecanismos de creación de objetos que incrementan la flexibilidad y la reutilización del código existente.

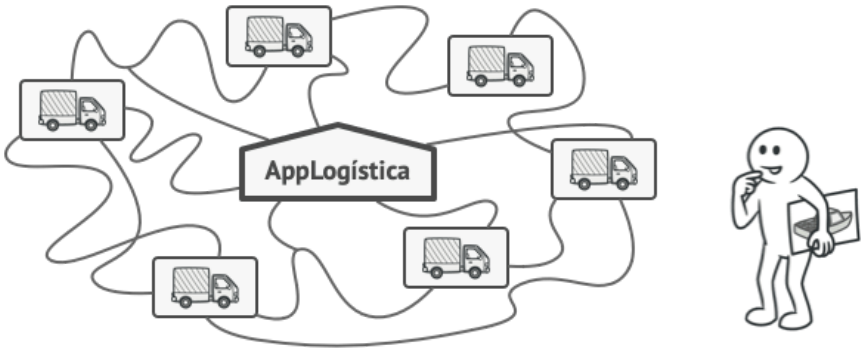
Factory Method

Proporciona una interfaz para crear objetos en una superclase, mientras permite a las subclases alterar el tipo de objetos que se crearán.



Problema en el cual se aplica:

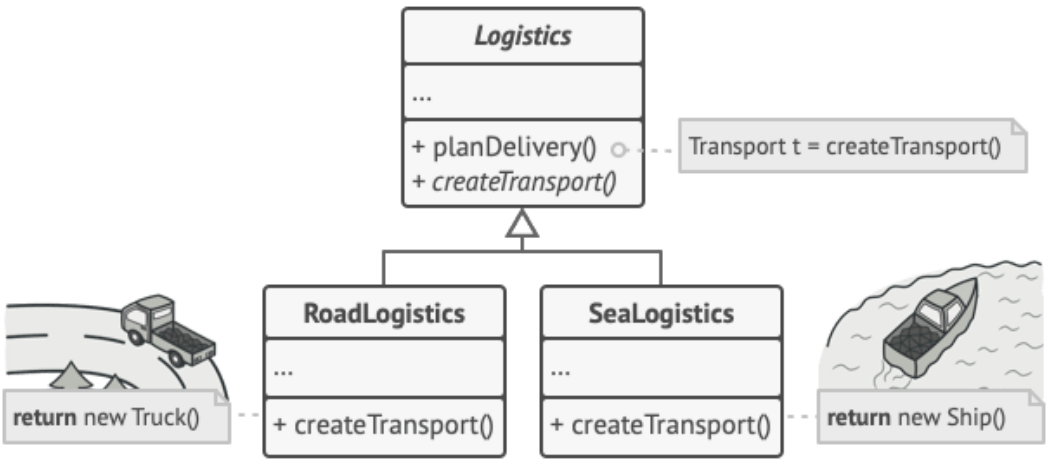
Si quisiera añadir una nueva clase, tendría que cambiar la mayoría del código ya que se encuentra bastante acoplado a las clases existentes.



Añadir una nueva clase al programa no es tan sencillo si el resto del código ya está acoplado a clases existentes.
Voy a tener un código bastante sucio, lleno de ifs que cambian el comportamiento de la aplicación dependiendo de la clase de los objetos del cual estoy trabajando (en el ejemplo serían las distintas logísticas).

Cómo se soluciona:

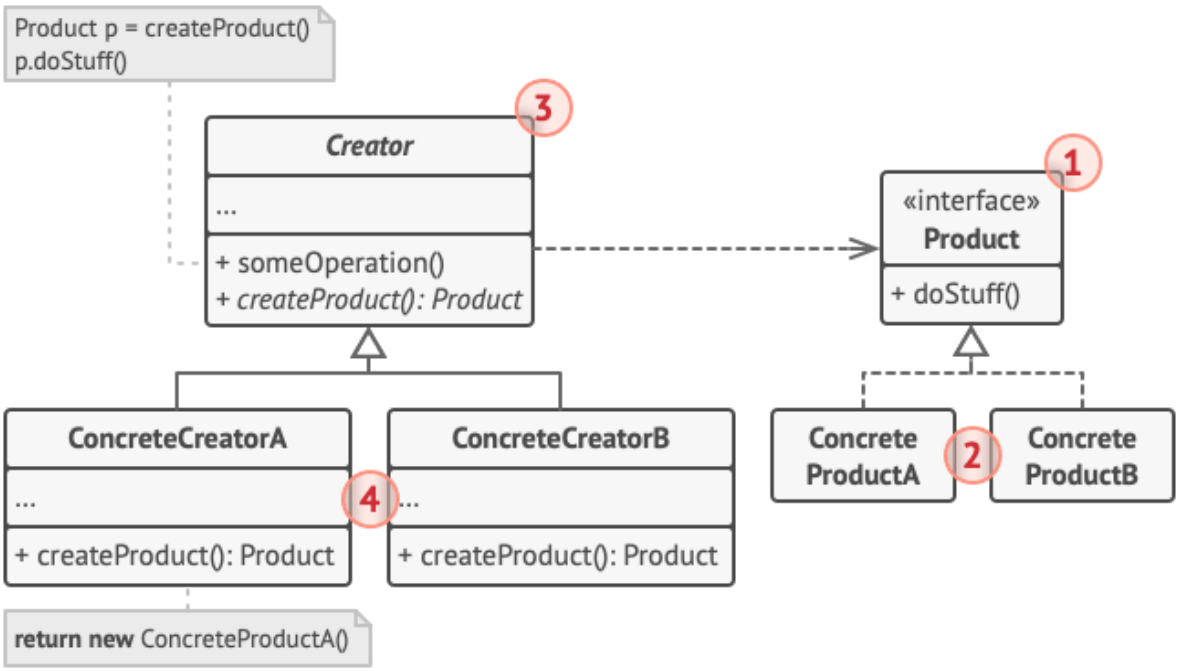
El patrón Factory Method sugiere que, en lugar de llamar al operador new para construir objetos directamente, se invoque a un método fábrica especial.



Las subclases pueden alterar la clase de los objetos devueltos por el método fábrica. Ahora puedo sobrescribir el método fábrica en una subclase y cambiar la clase de los productos creados por el método.

Limitación: las subclases sólo pueden devolver productos de distintos tipos si dichos productos tienen una clase base o interfaz común. Además, el método fábrica en la clase base debe tener su tipo de retorno declarado como dicha interfaz.

Estructura (en UML)



Tengo la interfaz producto común a la clase creadora y sus subclases. La clase creadora tiene el método fábrica que devuelve nuevos objetos de producto. Los creadores concretos sobreescriben el Factory Method, así devuelve un tipo diferente tipo de producto.

El Producto declara la interfaz, que es común a todos los objetos que puede producir la clase creadora y sus subclases.

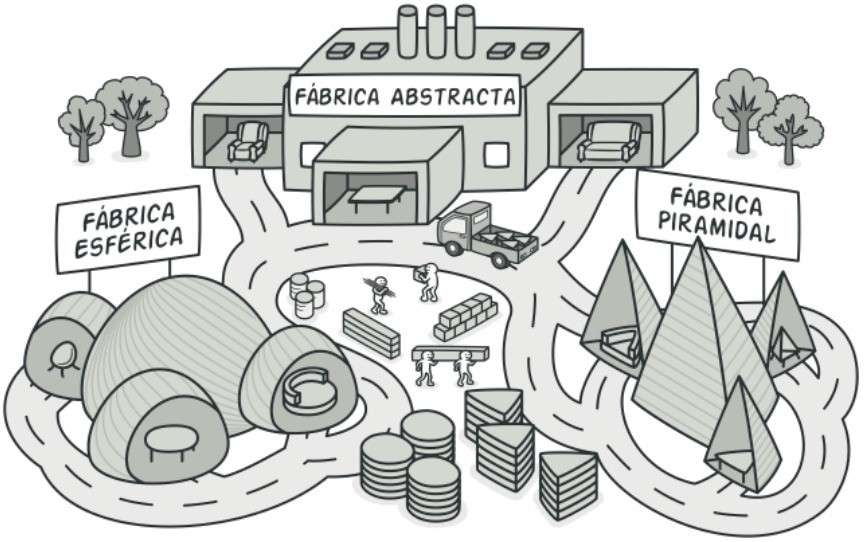
Cómo se implementa:

1. Todos los productos deben implementar la misma interfaz, que deberá tener métodos que tengan sentido en cada producto.
2. Añadir un patrón Factory Method vacío dentro de la clase creadora. El tipo de retorno del método deberá coincidir con la interfaz común de los productos.
1. Crear un grupo de subclases creadoras para cada tipo de producto enumerado en el Factory Method. Sobrescribe el Factory Method en las subclases y se extraen las partes adecuadas del código constructor del método base.

Ejemplo de código en catálogo.

Abstract Factory

Permite producir familias de objetos relacionados sin especificar sus clases concretas.



Problema en el cual se aplica:

En un simulador de tienda de muebles, tengo código compuesto por clases que representa lo siguiente:

1. Una familia de productos relacionados, digamos: `Silla` + `Sofá` + `Mesilla`.

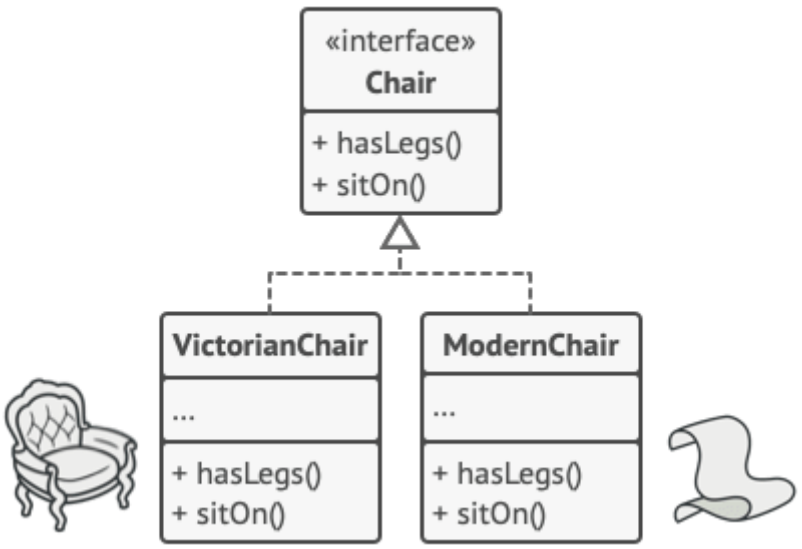
2. Algunas variantes de esta familia. Por ejemplo, los productos `Silla` + `Sofá` + `Mesilla` están disponibles en estas variantes: `Moderna`, `Victoriana`, `ArtDecó`.

Familias de productos y sus variantes.

Necesitamos una forma de crear objetos individuales de mobiliario para que se combinen con otros objetos de la misma familia.

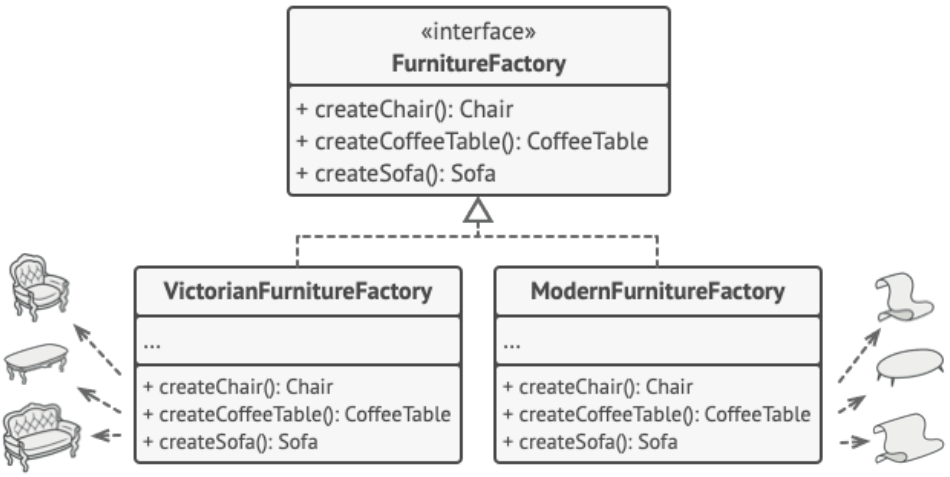
Cómo se soluciona:

Lo primero que sugiere el patrón Abstract Factory es que declaremos de forma explícita interfaces para cada producto diferente de la familia de productos (por ejemplo, silla, sofá o mesilla). Después podemos hacer que todas las variantes de los productos sigan esas interfaces. Por ejemplo, todas las variantes de silla pueden implementar la interfaz Silla, así como todas las variantes de mesilla pueden implementar la interfaz Mesilla, y así sucesivamente.



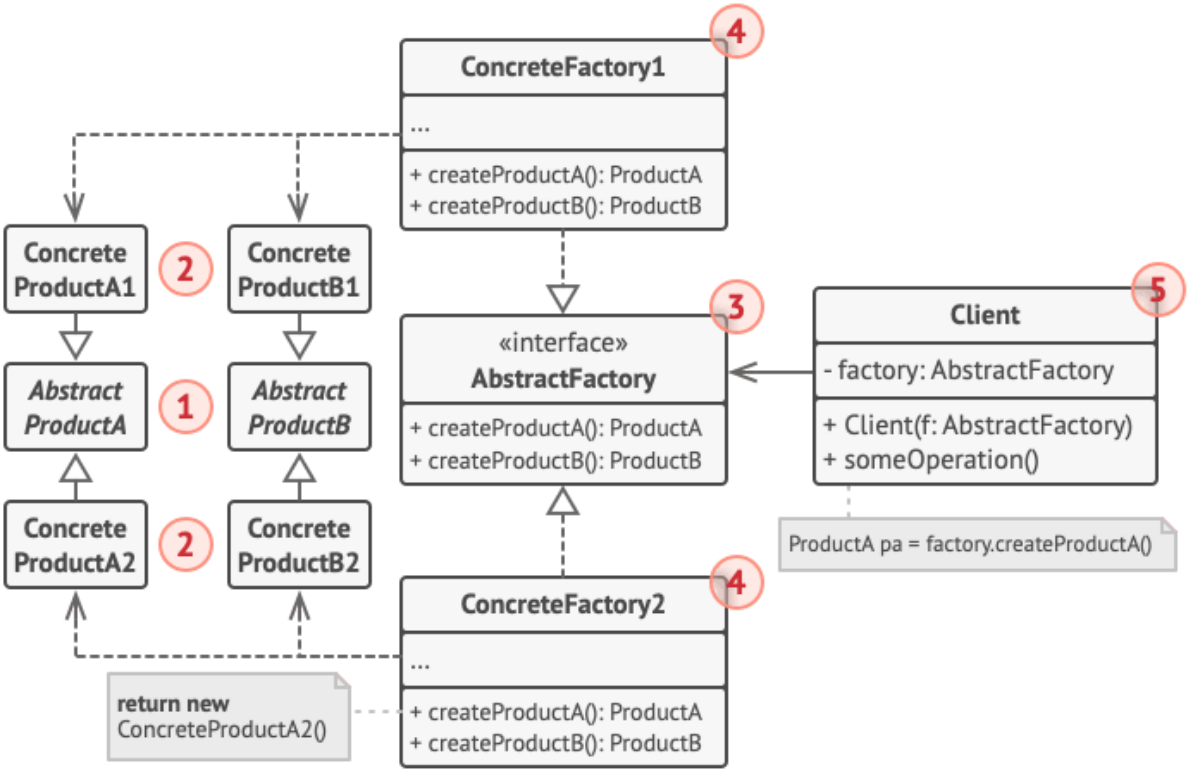
El siguiente paso consiste en declarar la Fábrica abstracta: una interfaz con una lista de métodos de creación para todos los productos que son parte de la familia de productos (por

ejemplo, crearSilla, crearSofá y crearMesilla). Estos métodos deben devolver productos abstractos representados por las interfaces que extrajimos previamente: Silla, Sofá, Mesilla, etc.



Cada fábrica concreta se corresponde con una variante específica del producto.

Estructura (en UML)



Cómo se implementa:

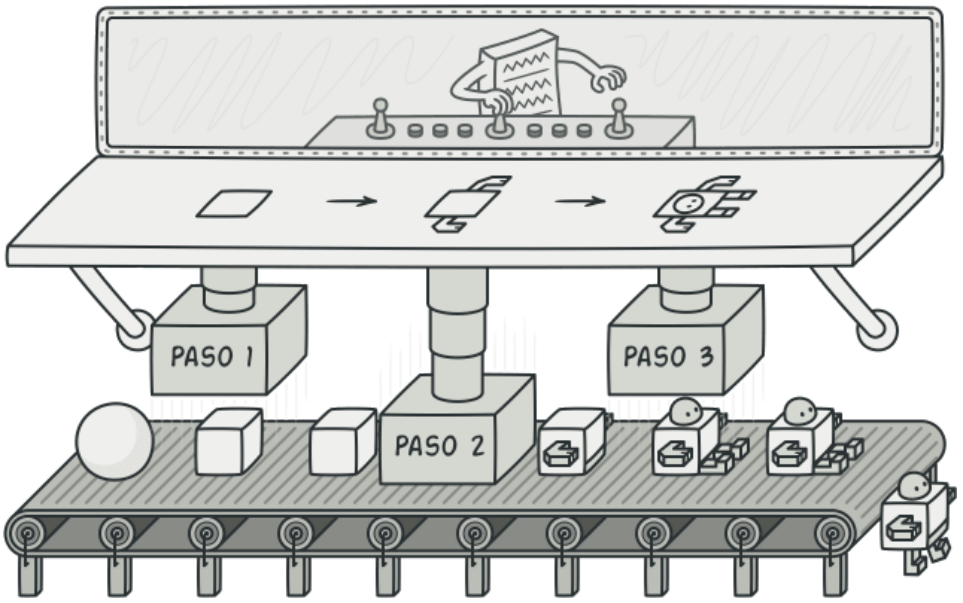
1. Mapear una matriz de distintos tipos de productos frente a variantes de dichos productos.

2. Declarar interfaces abstractas de producto para todos los tipos de productos. Las clases concretas deben implementar esas interfaces.
3. Declarar la interfaz de la fábrica abstracta con un grupo de métodos de creación para todos los productos abstractos.
4. Implementar un grupo de clases concretas de fábrica, una por cada variante de producto.

Ejemplo de código en catálogo

Builder (No en catálogo)

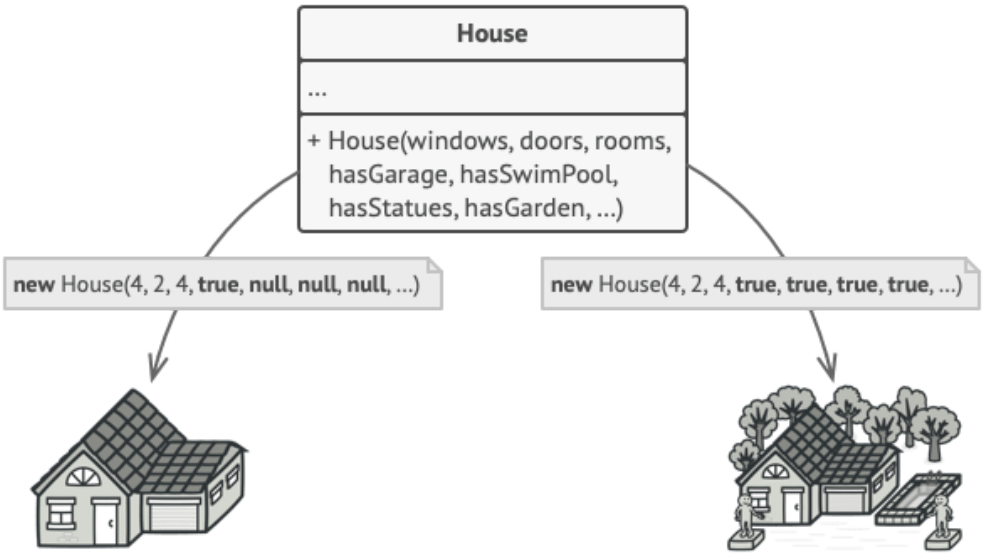
Permite construir objetos complejos paso a paso. El patrón nos permite producir distintos tipos y representaciones de un objeto empleando el mismo código de construcción.



Problema en el cual se aplica:

Sobre un objeto complejo que requiere una inicialización laboriosa, paso a paso, de muchos campos y objetos anidados. Normalmente es un código de inicialización con un constructor con gran cantidad de parámetros o peor, disperso por todo el código cliente.

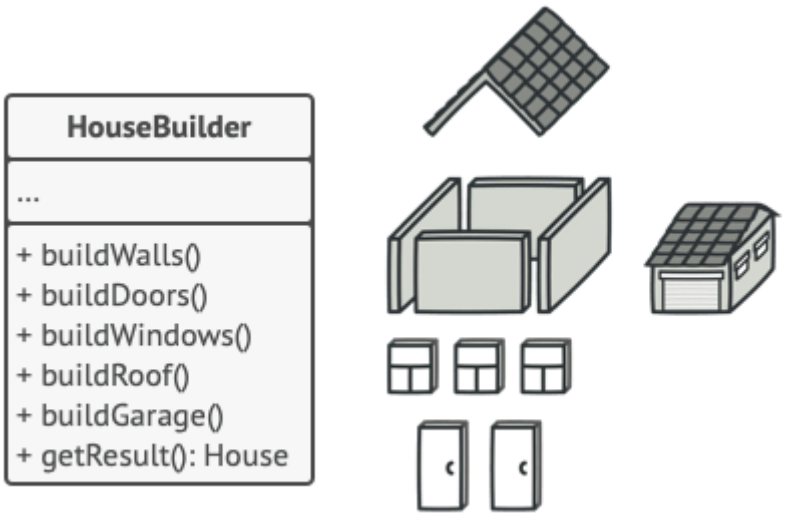
Crear una subclase por cada configuración posible de un objeto puede complicar demasiado el programa.



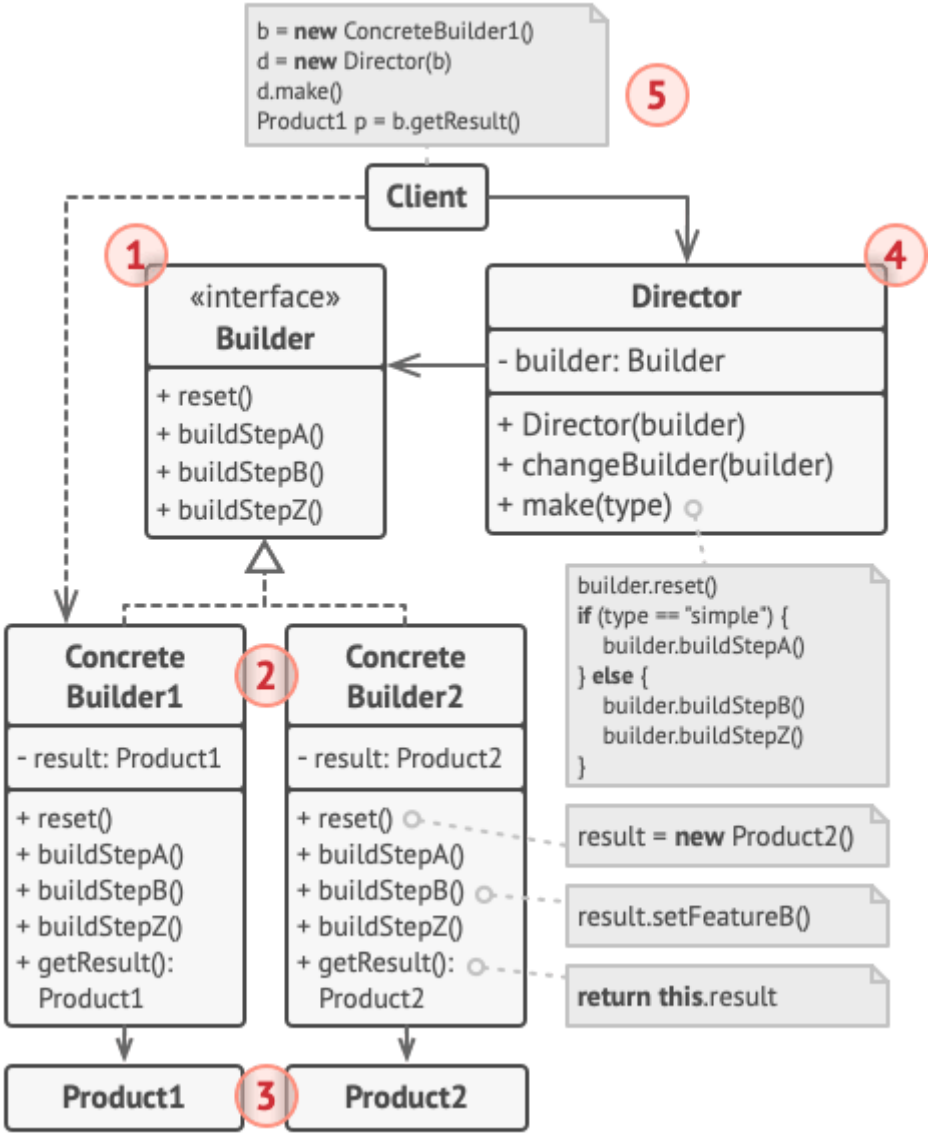
Un constructor con un montón de parámetros tiene su inconveniente: no todos los parámetros son necesarios todo el tiempo.

Cómo se soluciona:

Mediante este patrón se sugiere sacar el código de construcción del objeto de su propia clase y se coloque dentro de objetos independientes llamados constructores.



Estructura (en UML)



Prototype (No en catálogo)

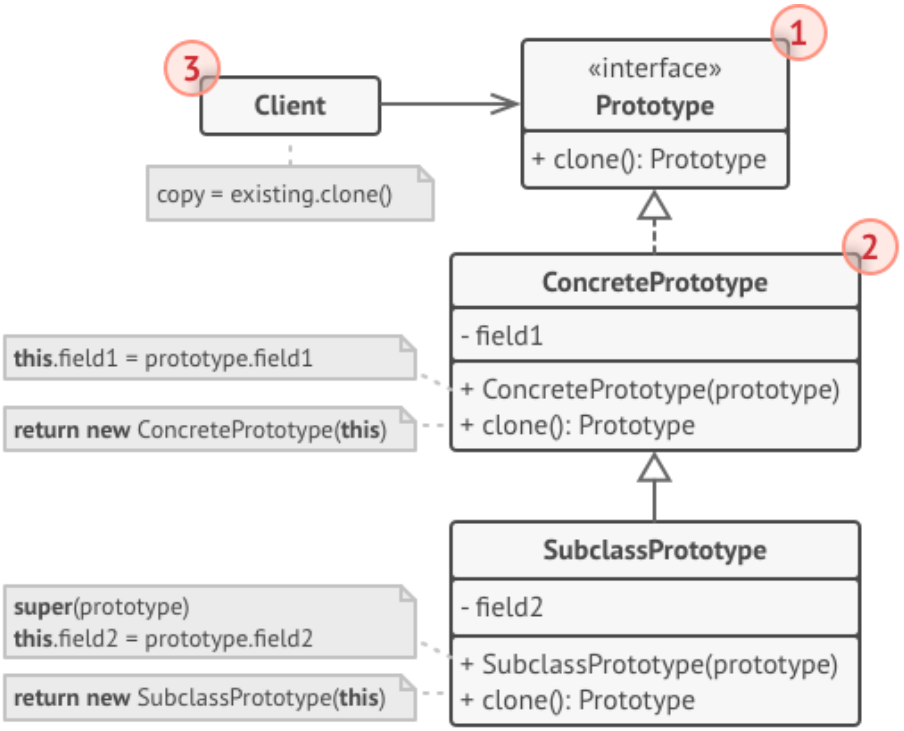
Permite copiar objetos existentes sin que el código dependa de sus clases.

Problema en el que se aplica:
Cuando quiero tener una copia exacta de un objeto y no puedo, ya que puede tener algunos atributos privados que no pueden ser vistos desde fuera del propio objeto. Además, dado que quiero conocer la clase del objeto al que quiero duplicar, el código se vuelve dependiente de esa clase.

Cómo se soluciona:

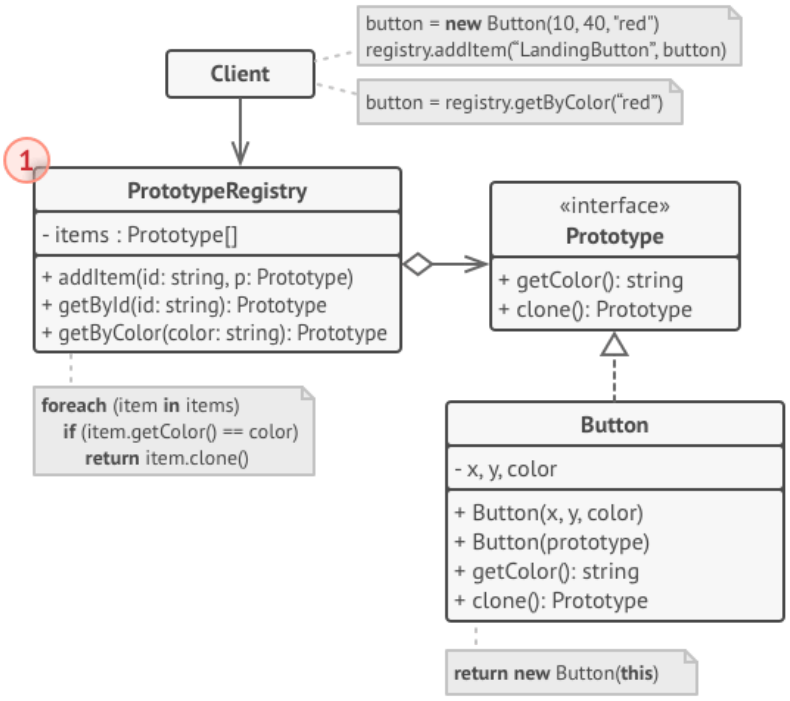
Prototype delega el proceso de clonación a los propios objetos que están siendo clonados. Se declara una interfaz común para todos los objetos que soportan la clonación. Así no acoplamos el código a la clase del objeto. clonar() crea un objeto a partir de la clase actual y lleva todos los valores del viejo al nuevo.

Estructura (en UML)



1. La interfaz Prototipo declara los métodos de clonación. En la mayoría de los casos, se trata de un único método clonar.
2. La clase Prototipo Concreto implementa el método de clonación.
3. El Cliente puede producir una copia de cualquier objeto que siga la interfaz del prototipo.

Cómo se implementa:



Singleton

Permite asegurarnos de que una clase tenga una única instancia, a la vez que proporciona un punto de acceso global a dicha instancia.

Problema al que aplica:

Resuelve dos problemas al mismo tiempo, violando el Principio de Responsabilidad Única:

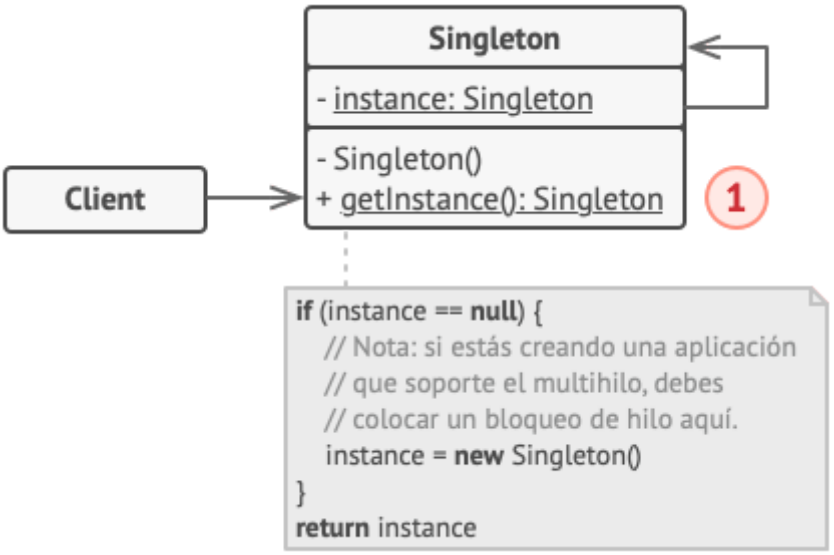
1. Garantizar que una clase tenga una única instancia.
2. Proporcionar un punto de acceso global a dicha instancia.

Cómo se soluciona:

- Hacer privado el constructor por defecto para evitar que otros objetos utilicen el new de Singleton
- Crear método de creación estático que actúe como constructor.

Analogía
El gobierno es un ejemplo excelente del patrón Singleton. Un país sólo puede tener un gobierno oficial. Independientemente de las identidades personales de los individuos que forman el gobierno, el título “Gobierno de X” es un punto de acceso global que identifica al grupo de personas a cargo.

Estructura (en UML)



La clase Singleton declara el método estático obtenerInstancia que devuelve la misma instancia de su propia clase.
El constructor del Singleton debe ocultarse del código cliente. La llamada al método obtenerInstancia debe ser la única manera de obtener el objeto de Singleton.

Cómo se implementa:

Se añade un campo estático privado a la clase para almacenar la instancia Singleton, creamos un método de creación estático público para instanciarla. El constructor de la clase será privado (para crear 1 único objeto).

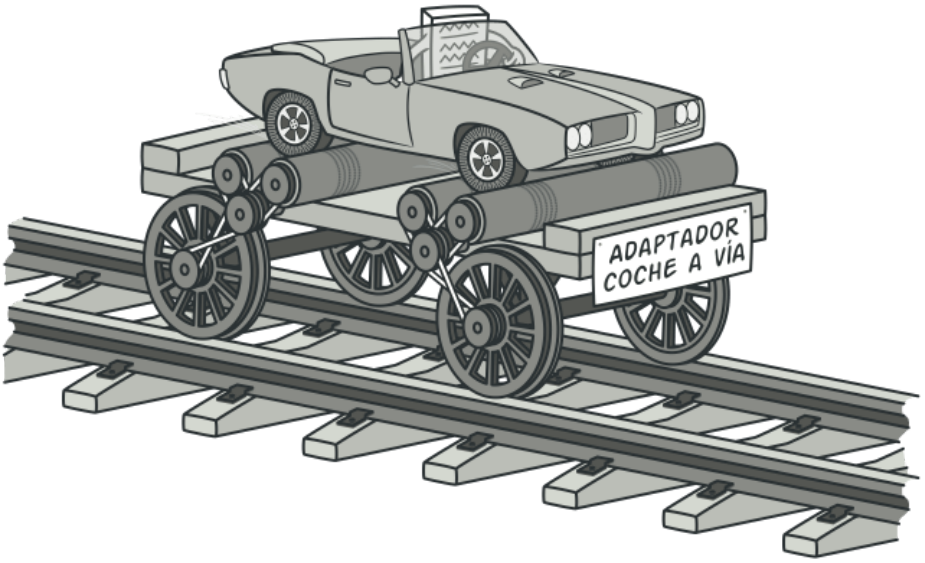
Ejemplo de código en catálogo

Patrones Estructurales

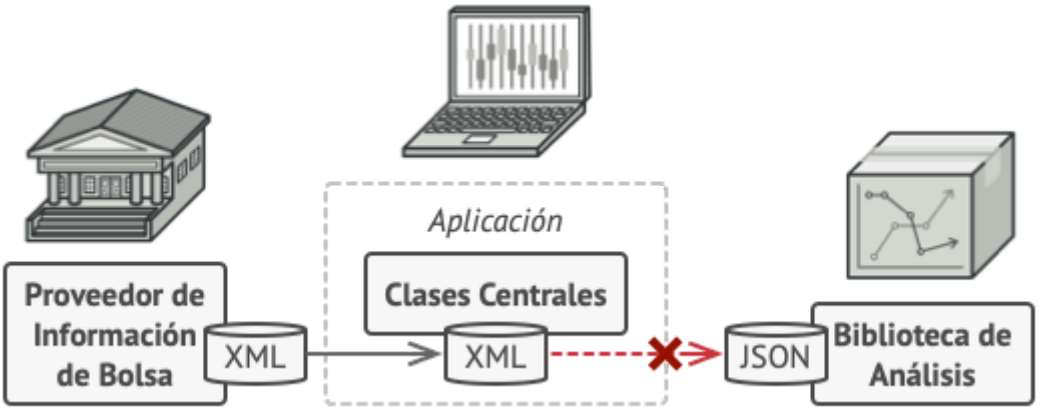
Explican cómo ensamblar objetos y clases en estructuras más grandes, manteniendo flexibilidad y eficiencia de estas estructuras.

Adapter (No en catálogo)

Permite colaboración entre objetos con interfaces incompatibles.



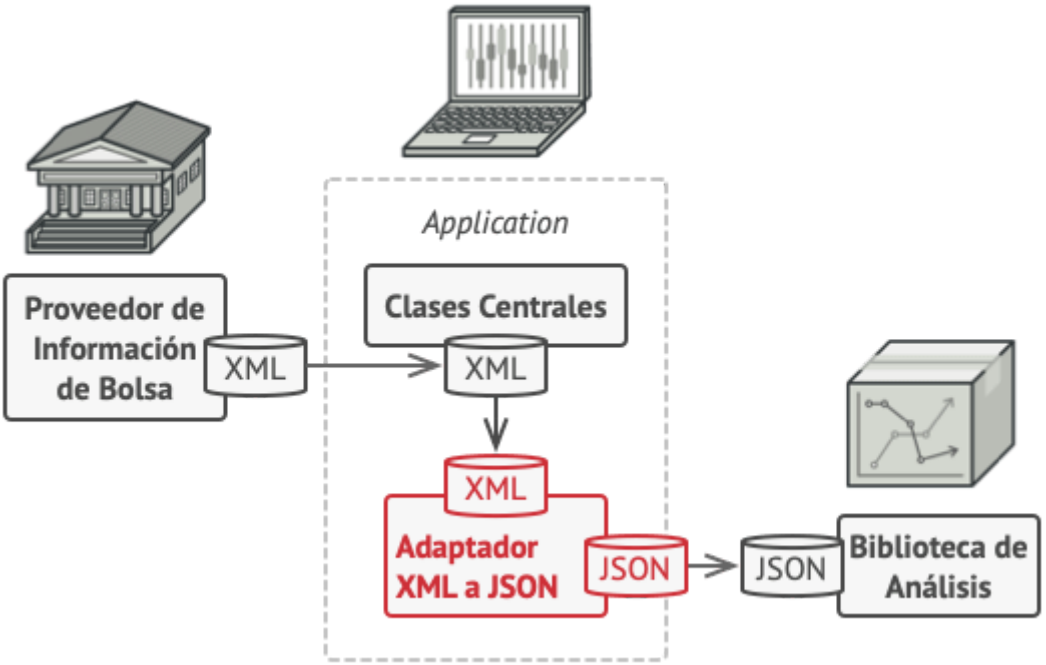
Problema al cual aplica:



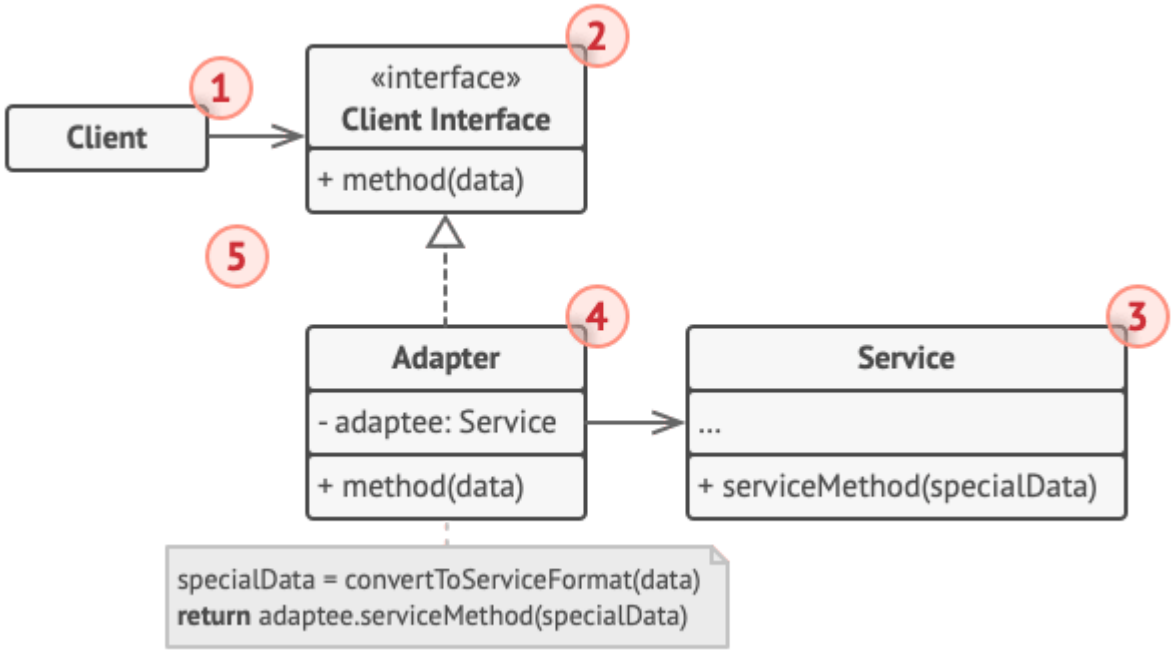
No se puede utilizar la biblioteca de análisis “tal cual” porque ésta espera los datos en un formato que es incompatible con tu aplicación.

Cómo se soluciona:

Se puede crear un adaptador, un objeto especial que convierte la interfaz de un objeto de forma que otro objeto pueda comprenderla.

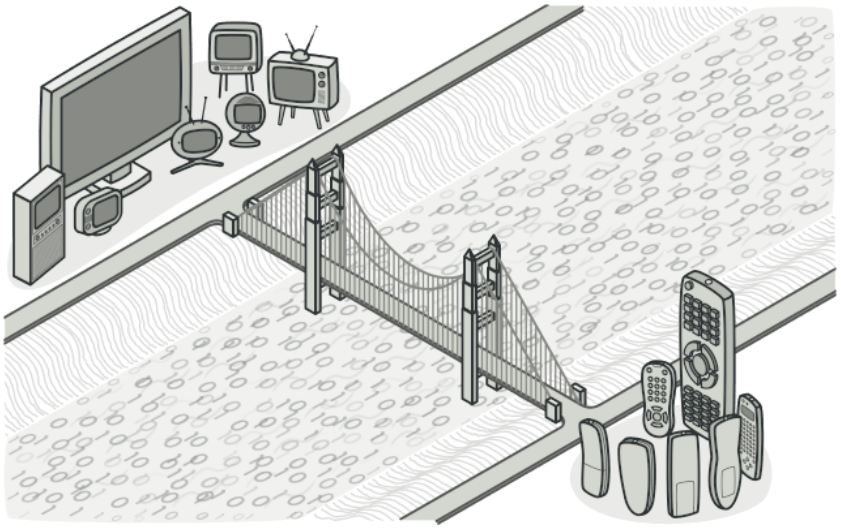


Estructura (en UML)



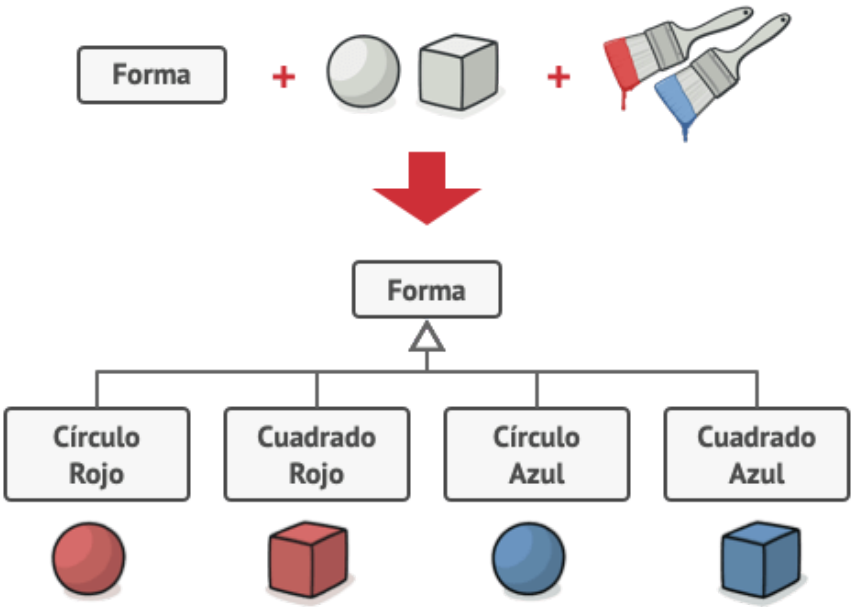
Bridge (No en catálogo)

Permite dividir una clase grande o grupo de clases muy relacionadas en 2 jerarquías separadas (abstracción e implementación) que se desarrollen independientemente entre sí.



Problema al cual aplica:

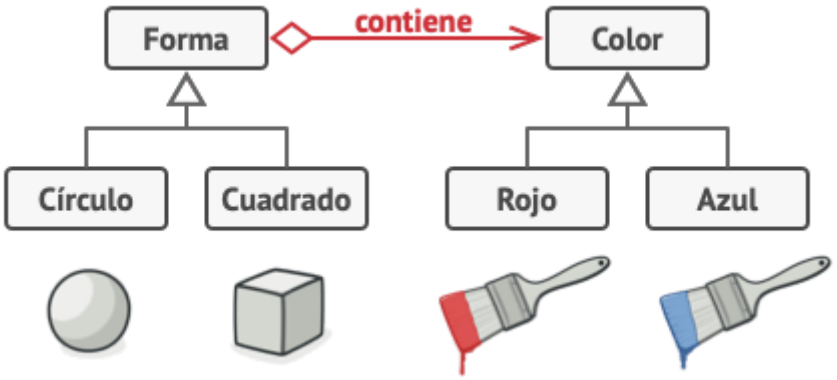
Tengo una clase Forma con dos subclases Círculo y Cuadrado, quiero que incorporen colores pero para eso tengo que crear 4 combinaciones de clases (lo que vimos como explosión de clases).



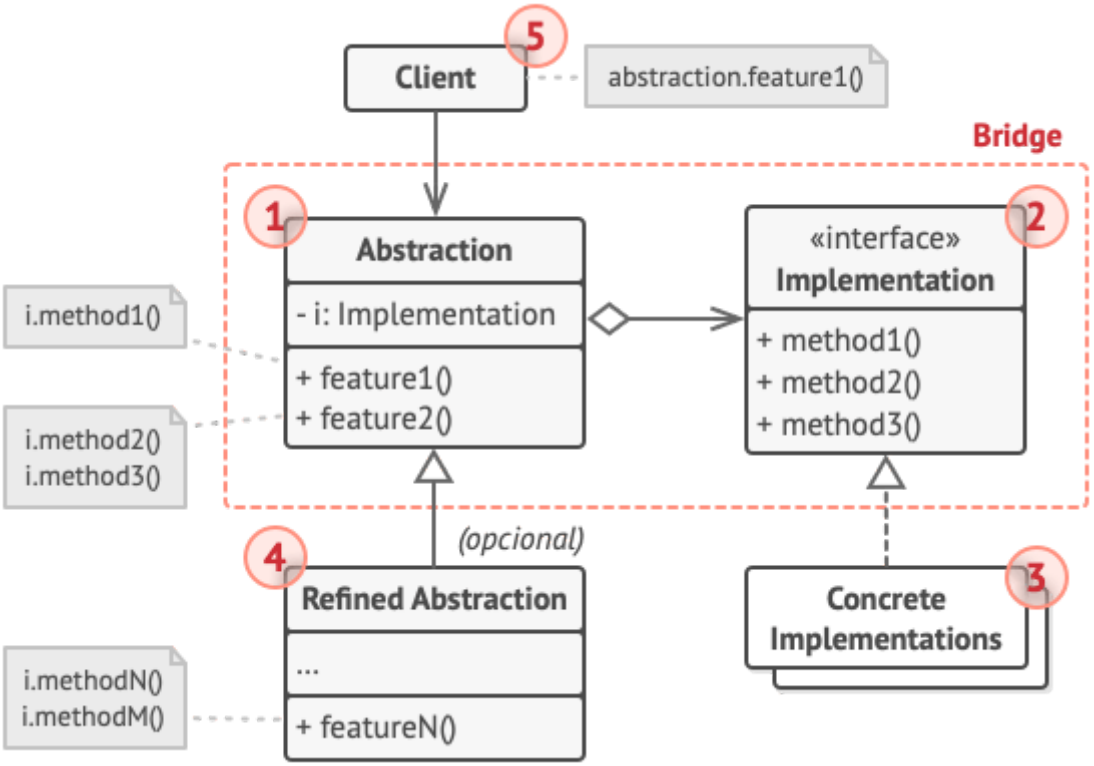
El número de combinaciones de clase crece en progresión geométrica.

Cómo se soluciona:

Lo que queremos lograr es extender las clases en dimensiones independientes (en el ejemplo forma y color).
Para solucionarlo pasamos de la herencia (es) a la composición (tiene).

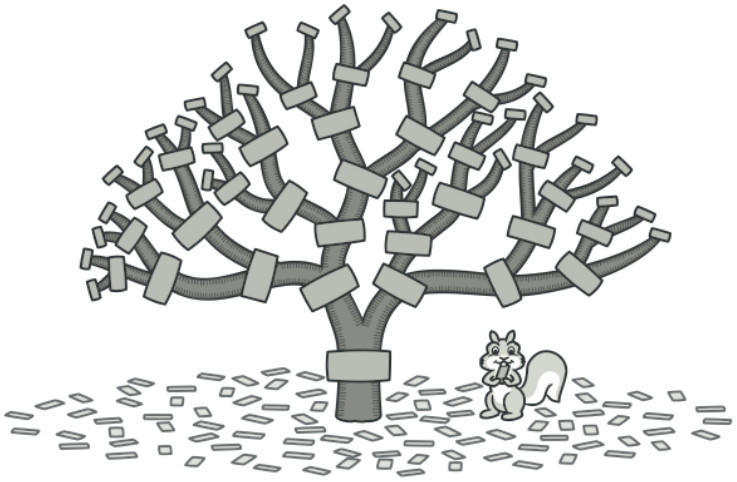


Estructura (en UML)

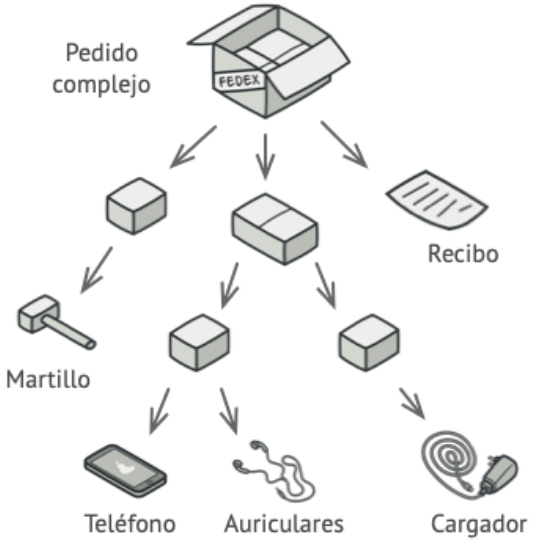


Composite (No en catálogo)

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



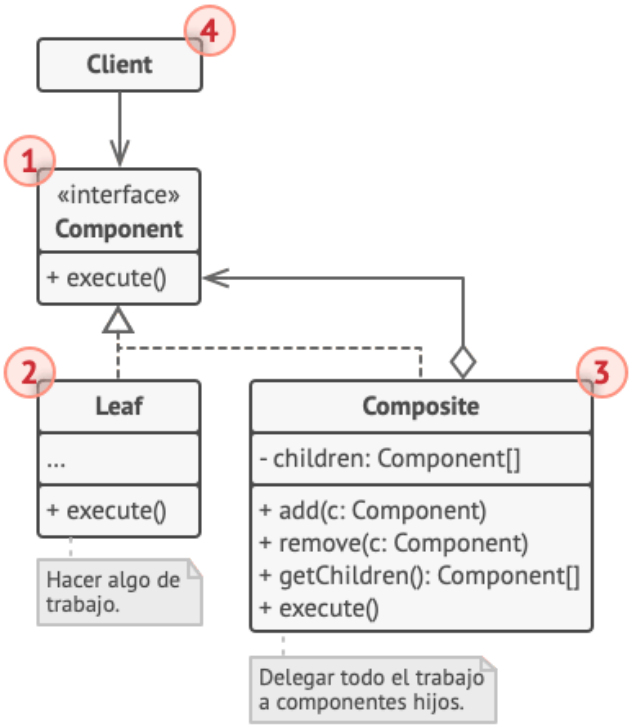
Problema al cual aplica:
Cuando se puede modelar como árbol.



Llegar al nodo final de los árboles implica una solución bastante complicada.

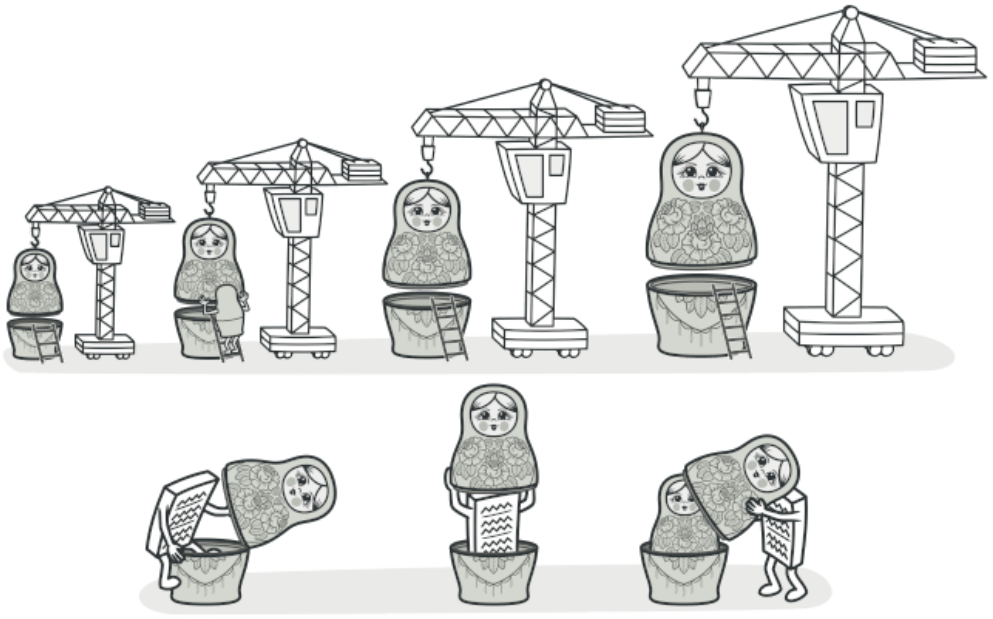
Cómo se soluciona:

Ejecutamos un comportamiento de forma recursiva sobre todos los componentes de un árbol de objetos.



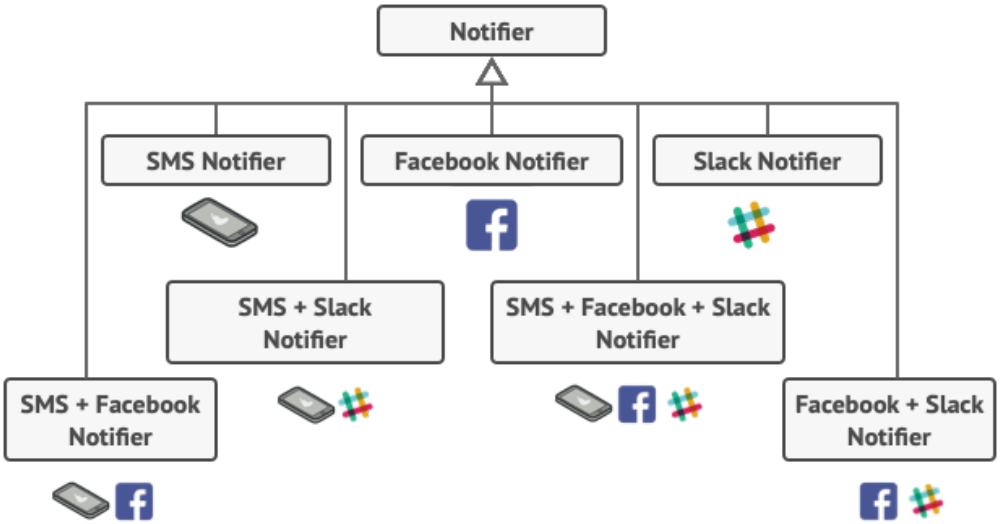
Decorator

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



Problema al cual aplica:

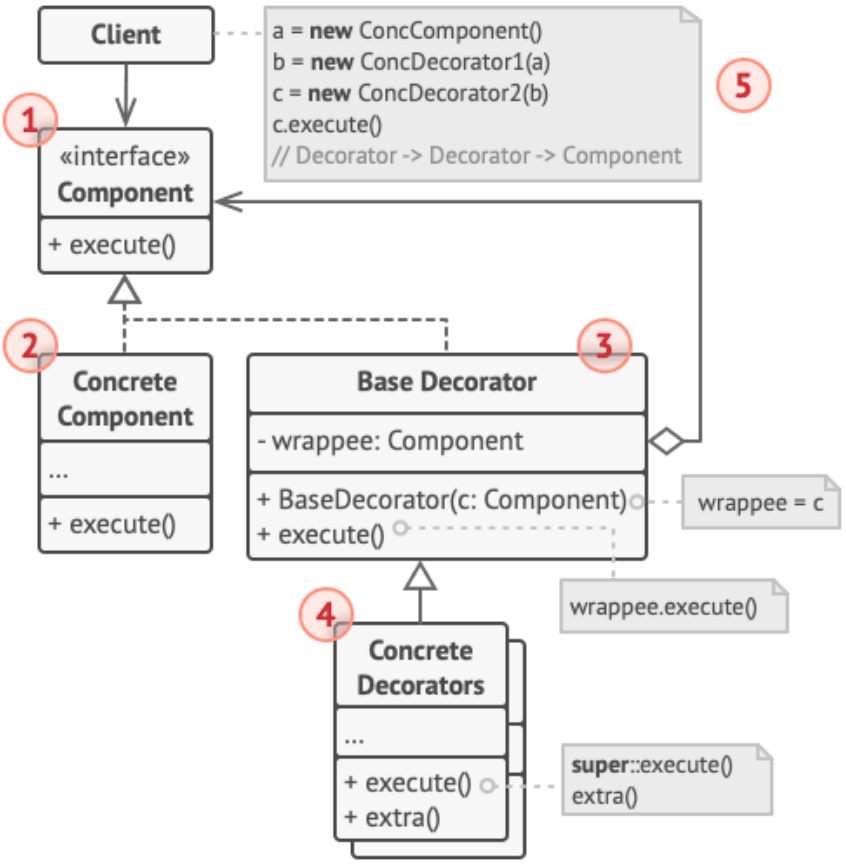
Explosión combinatoria de subclases.



Cómo se soluciona:

Empleamos Agregación (A contiene objetos B; B puede existir sin A) o Composición (A compuesto de B; A gestiona ciclo vital de B; B no existe sin A). Un objeto tiene una referencia a otro y le delega parte del trabajo, mientras que con la herencia, el propio objeto puede realizar ese trabajo, heredando el comportamiento de su superclase.

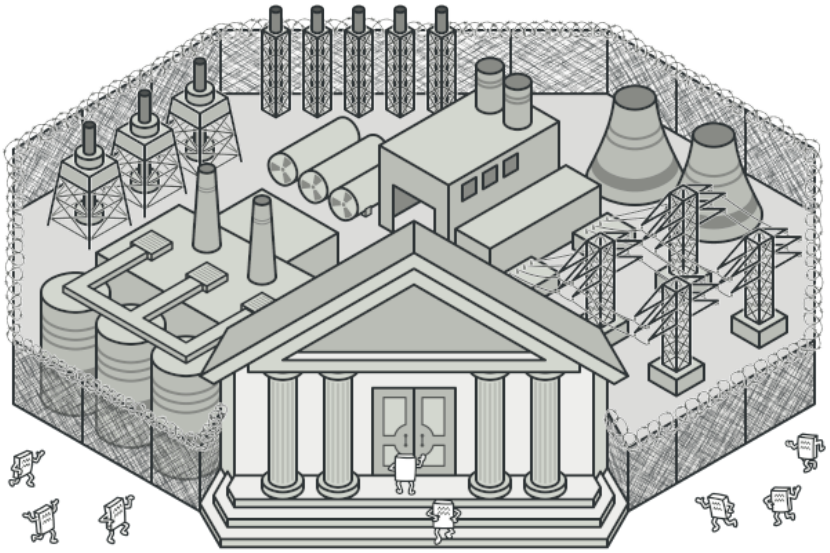
Estructura (en UML)



Ejemplo de código en catálogo.

Facade (muito obrigado)

Proporciona una interfaz simplificada a un grupo complejo de clases.



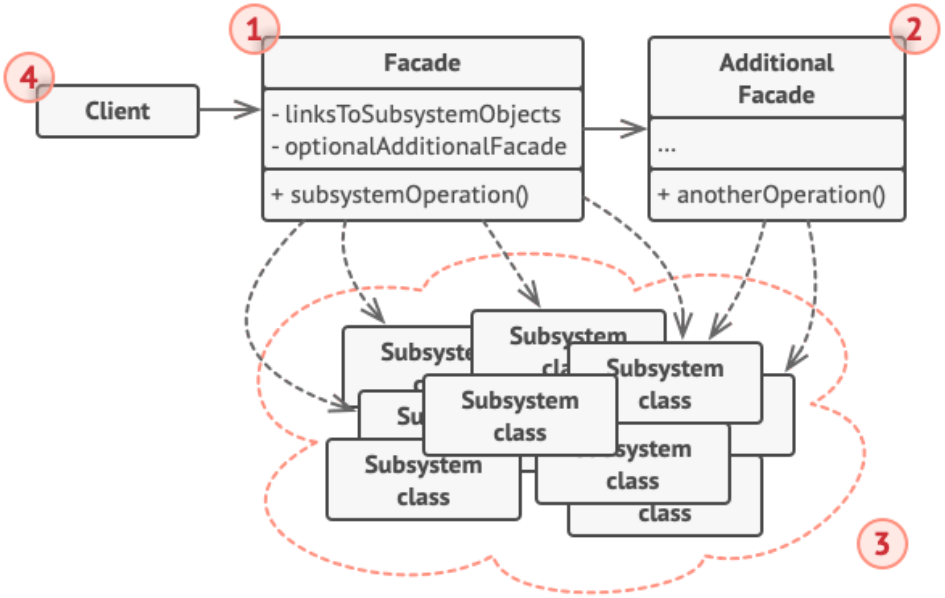
Problema donde aplica:

La lógica de negocio de las clases se ve estrechamente acoplada a los detalles de implementación de otras clases, generando poca comprensión y mantenibilidad.

Cómo se soluciona:

Una fachada es una clase que proporciona una interfaz simple a un subsistema complejo que contiene muchas partes móviles.

Estructura (en UML)



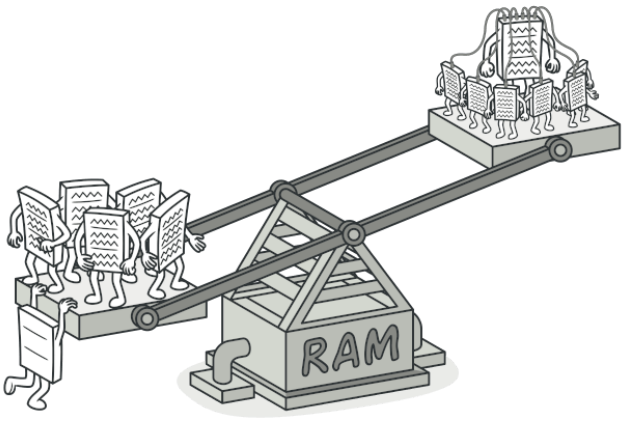
1. El patrón Facade proporciona un práctico acceso a una parte específica de la funcionalidad del subsistema. Sabe a dónde dirigir la petición del cliente y cómo operar todas las partes móviles.

2. El Cliente utiliza la fachada en lugar de invocar directamente los objetos del subsistema.

Ejemplo de código en catálogo.

Flyweight (No en catálogo)

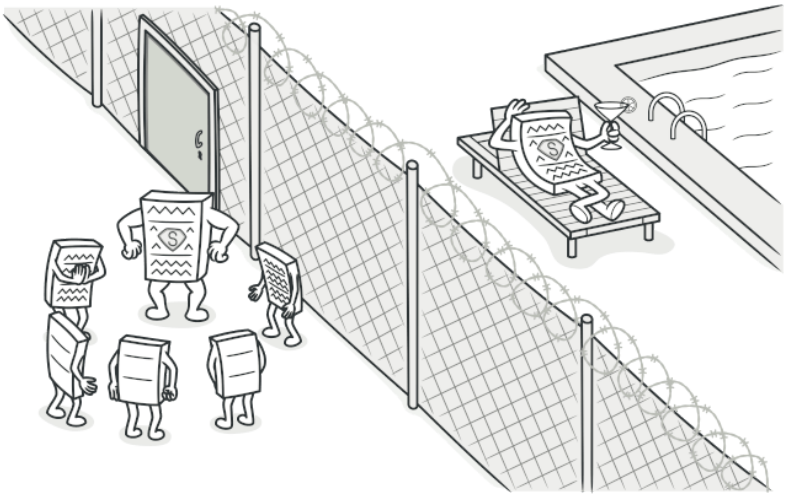
Permite mantener más objetos dentro de la cantidad disponible de RAM compartiendo las partes comunes del estado entre varios objetos en lugar de mantener toda la información en cada objeto.



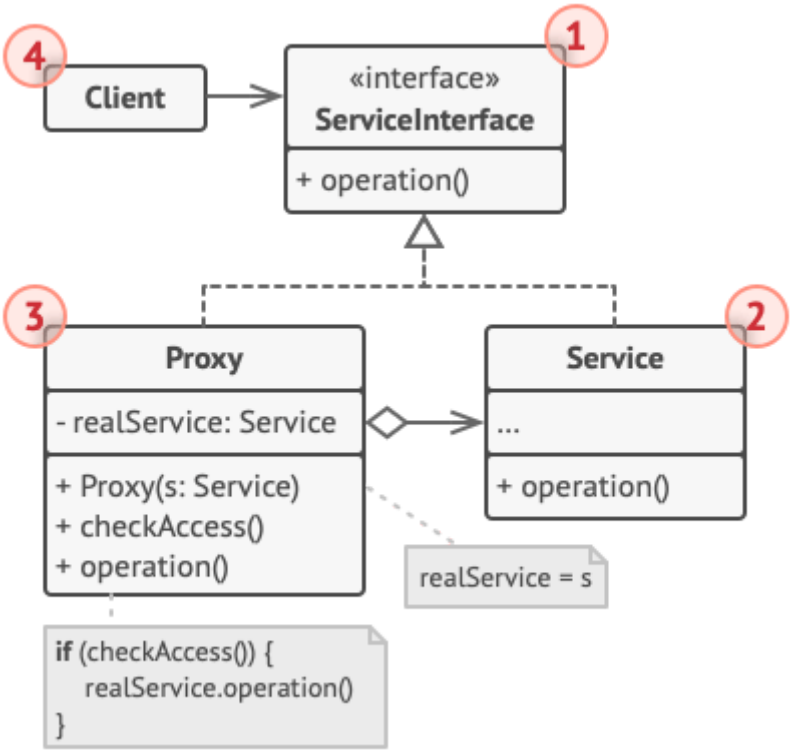
(No me parece sumamente importante, al menos por ahora)

Proxy (No en catálogo)

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



Estructura (en UML)

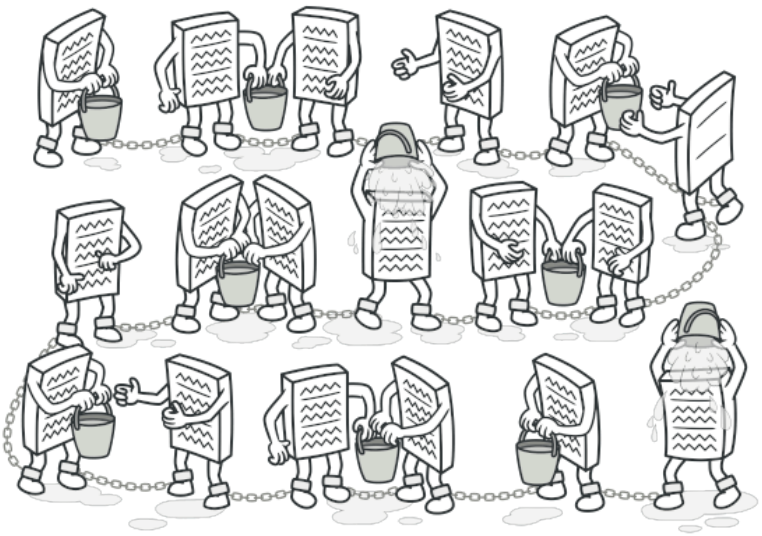


Patrones de Comportamiento

Tratan con algoritmos y la asignación de responsabilidades entre objetos.

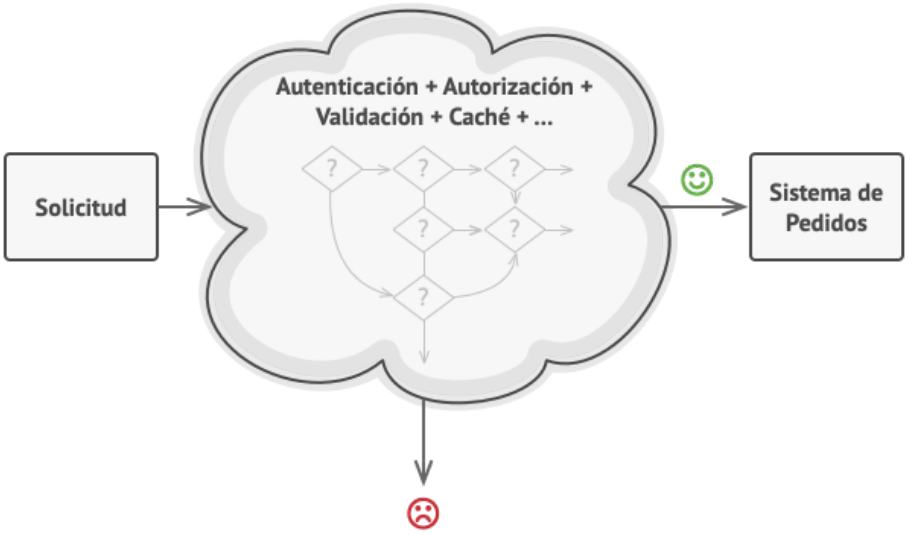
Chain of Responsibility

Permite pasar solicitudes a lo largo de una cadena de manejadores. Al recibir una solicitud, cada manejador decide si la procesa o si la pasa al siguiente manejador de la cadena.

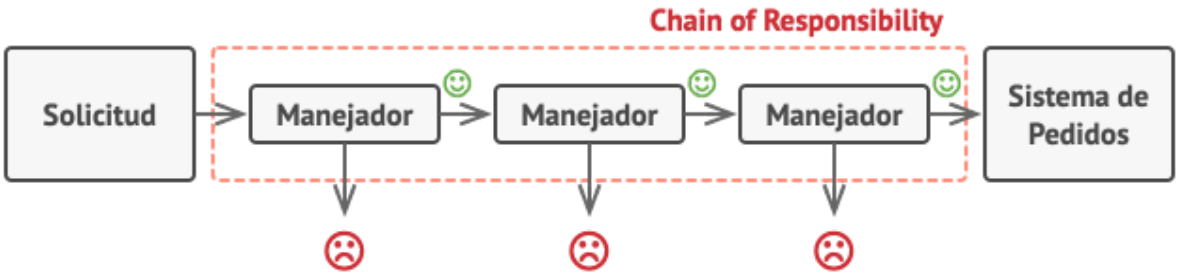


Problema al que aplica:

Cuando necesito hacer comprobaciones secuencialmente.

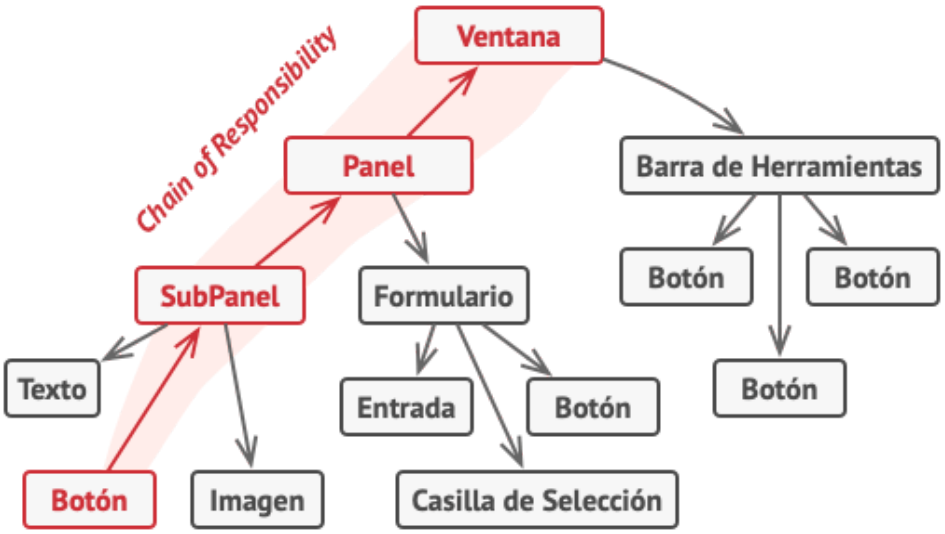


Cómo se soluciona:



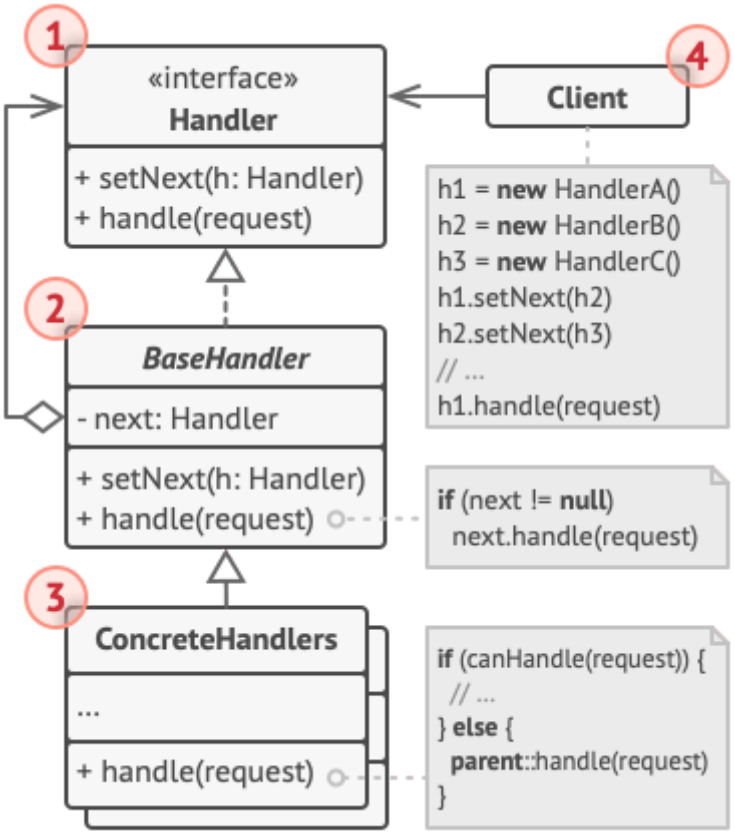
Los manejadores se alinean uno tras otro, formando una cadena.

Otra más estandarizada:



Es fundamental que todas las clases manejadoras implementen la misma interfaz.

Estructura (en UML)

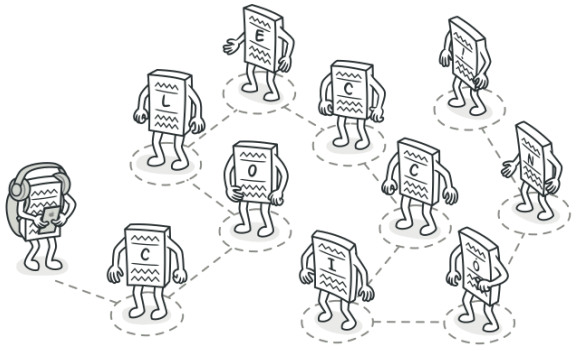


Command (No en catálogo)

Convierte una solicitud en un objeto independiente que contiene toda la información sobre la solicitud. Esta transformación permite parametrizar los métodos con diferentes solicitudes, retrasar o poner en cola la ejecución de una solicitud y soportar operaciones que no se pueden realizar.

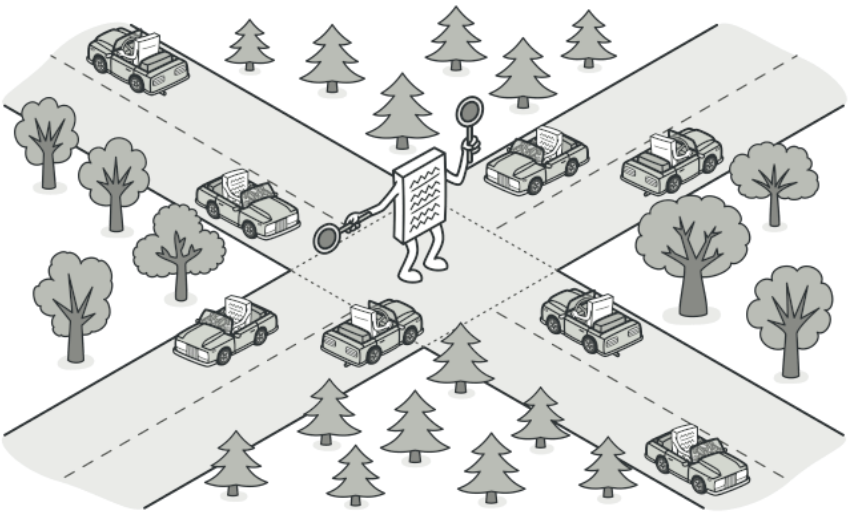
Iterator (No en catálogo)

Permite recorrer elementos de una colección sin exponer su representación subyacente (osea, accedo a recorrer un elemento, sin saber cómo éste es por dentro).

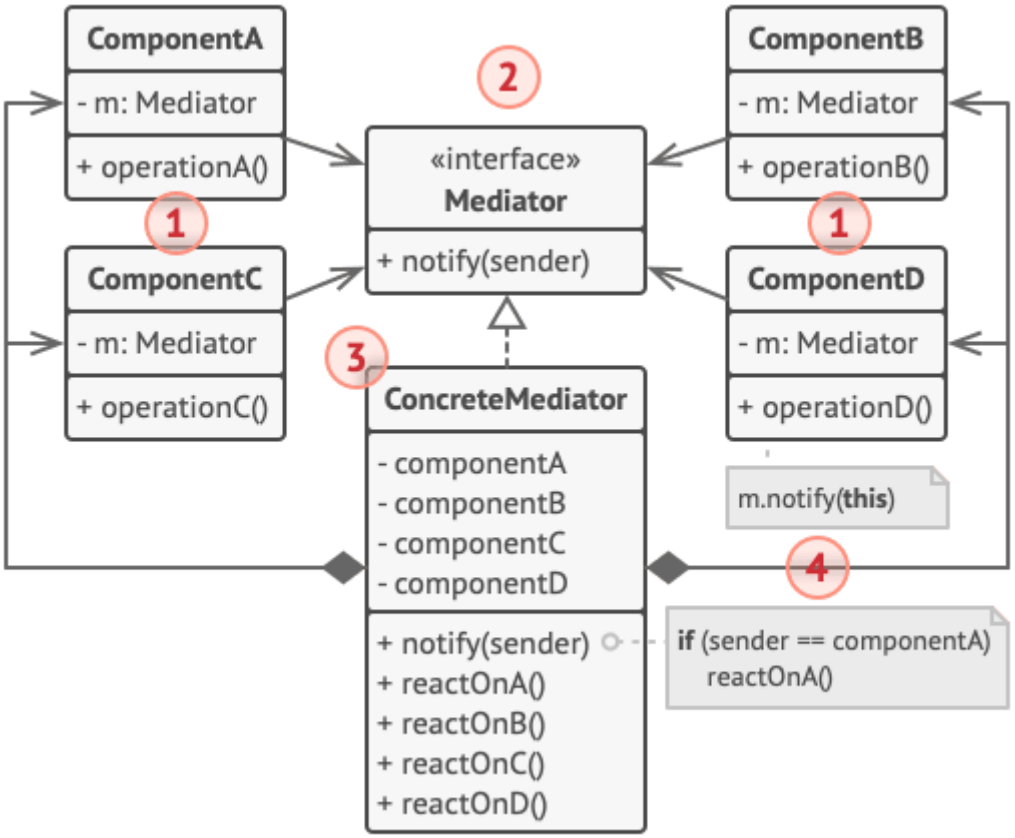


Mediator (No en catálogo)

Permite reducir las dependencias caóticas entre objetos, restringiendo las comunicaciones directas entre objetos, forzándolos a colaborar a través de un objeto mediador.



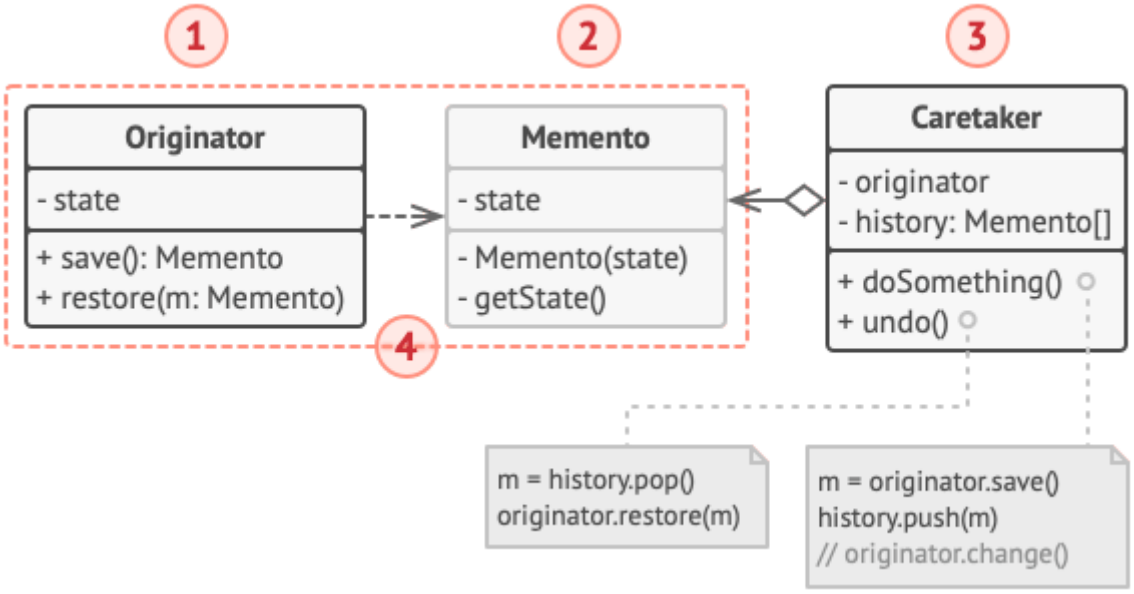
Estructura (en UML)



Memento (No en catálogo) (es una buena peli)

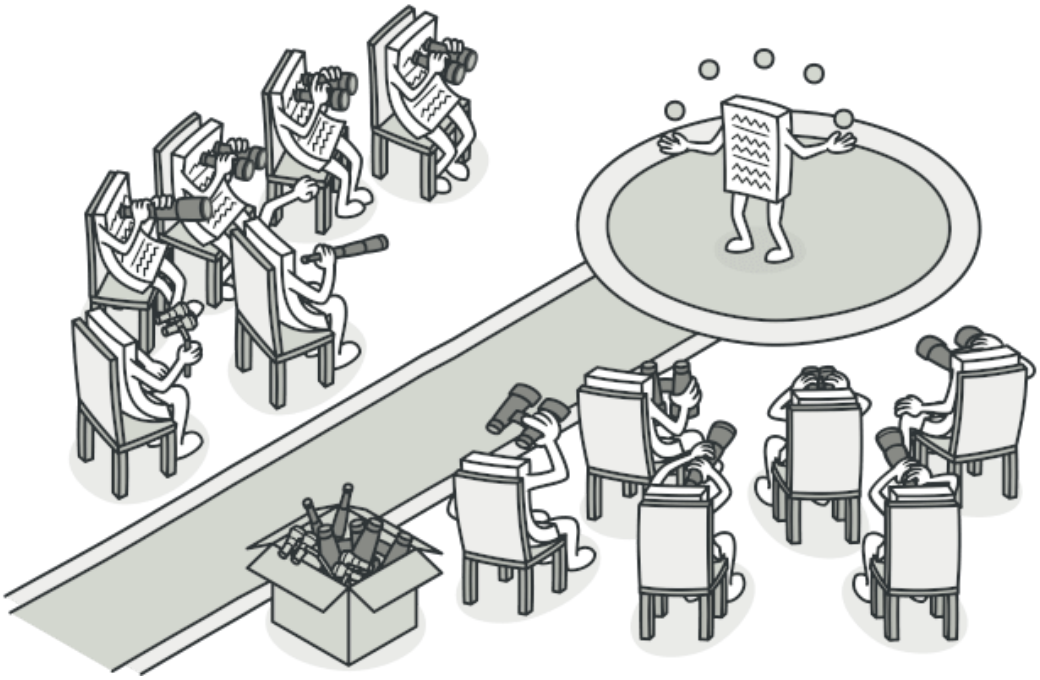
Permite guardar y restaurar el estado previo de un objeto sin revelar los detalles de su implementación.

Estructura (en UML)



Observer

Permite definir un mecanismo de suscripción para notificar a varios objetos sobre cualquier evento que le suceda al objeto que están observando.



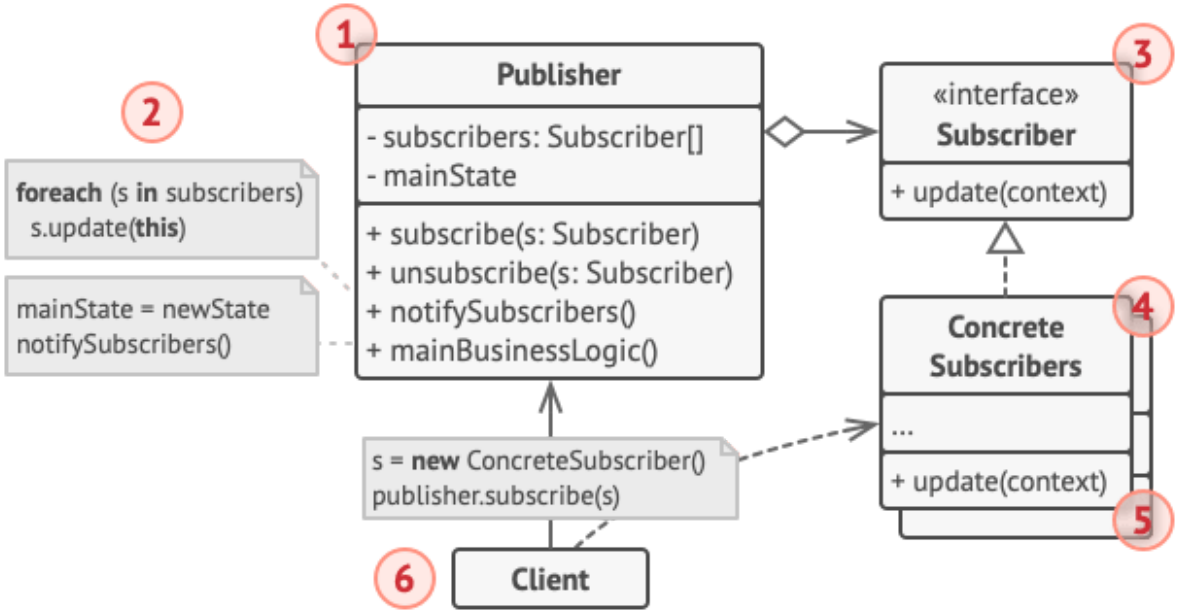
Problema al que aplica:

Cuando tengo que verificar muchas veces en vano sobre un evento particular en un objeto.

Cómo se soluciona:

Tendremos un notificador que avisará ante un evento que suceda sobre un objeto en particular a los objetos que se encuentran suscritos a estas notificaciones.

Estructura (en UML)



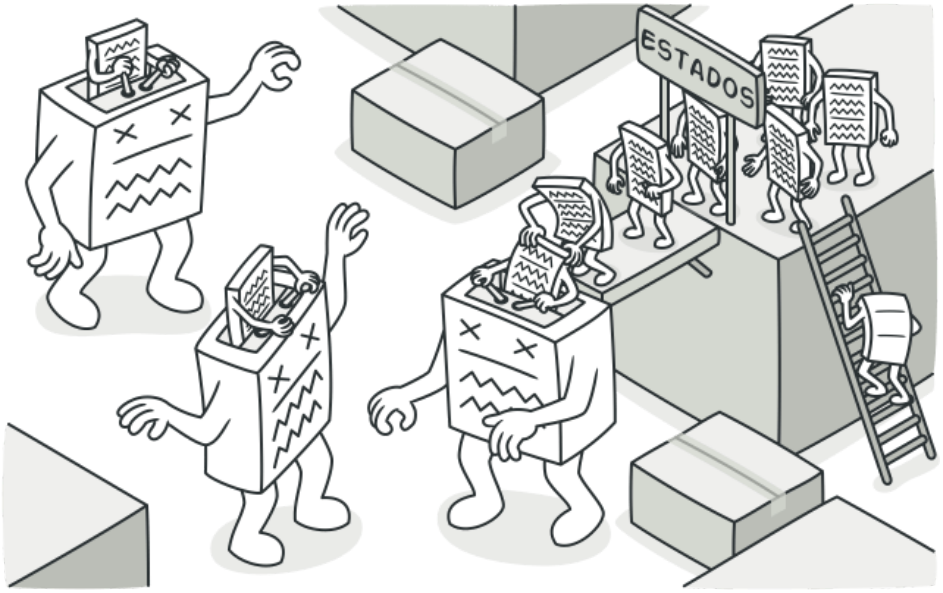
1. El Notificador envía eventos de interés a otros objetos. Esos eventos ocurren cuando el notificador cambia su estado o ejecuta algunos comportamientos. Los

notificadores contienen una infraestructura de suscripción que permite a nuevos y antiguos suscriptores abandonar la lista. Cuando sucede un nuevo evento, el notificador recorre la lista de suscripción e invoca el método de notificación declarado en la interfaz suscriptora en cada objeto suscriptor.

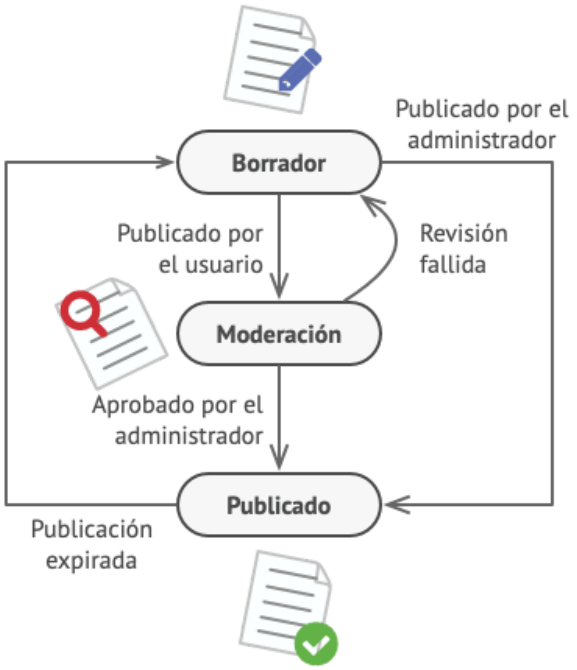
Ejemplo de código en catálogo.

State

Permite a un objeto alterar su comportamiento cuando su estado interno cambia. Es como si el objeto cambiara de clase.



Problema al cual aplica:

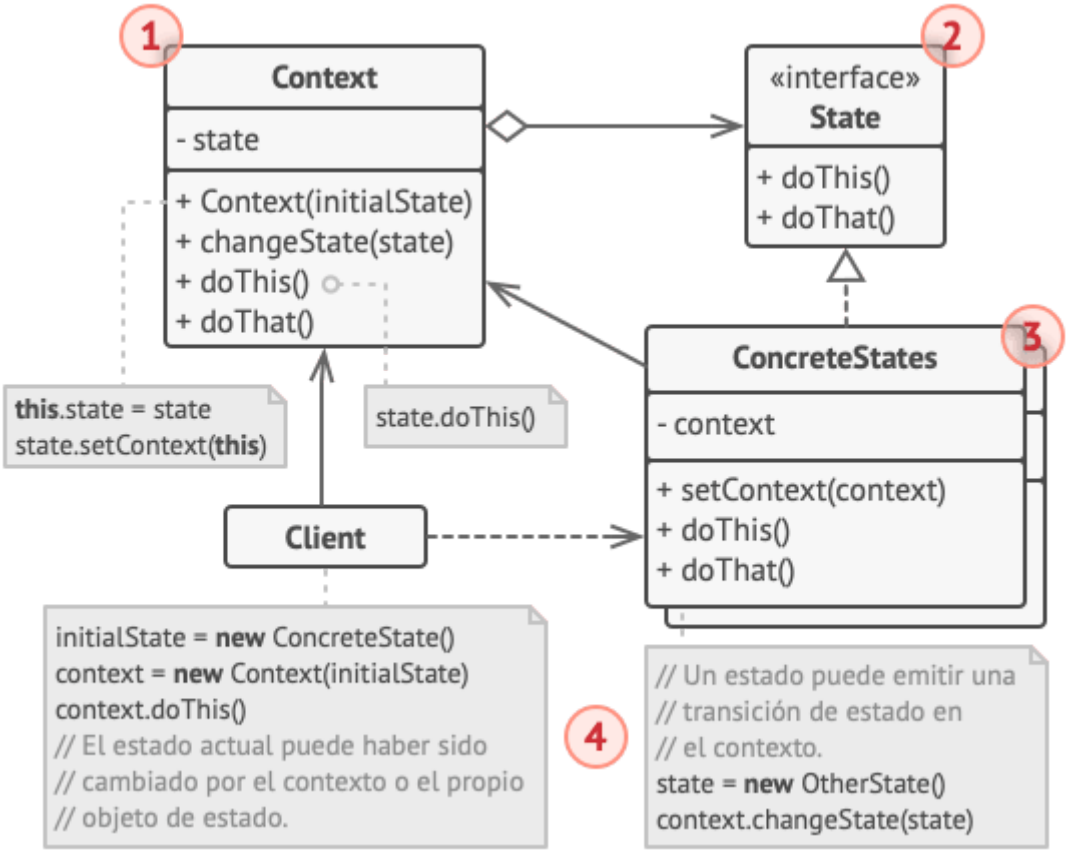


Las máquinas de estado se implementan normalmente con muchos operadores condicionales (if o switch) que seleccionan el comportamiento adecuado dependiendo del estado actual del objeto. Normalmente, este “estado” es tan solo un grupo de valores de los campos del objeto.

Cómo se soluciona:

Creamos nuevas clases para todos los estados posibles de un objeto de los cuales extraemos todos sus comportamientos específicos.

Estructura (en UML)



Ejemplo de código en catálogo.

Strategy

Permite definir una familia de algoritmos, colocar cada uno en una clase separada y hacer sus objetos intercambiables.

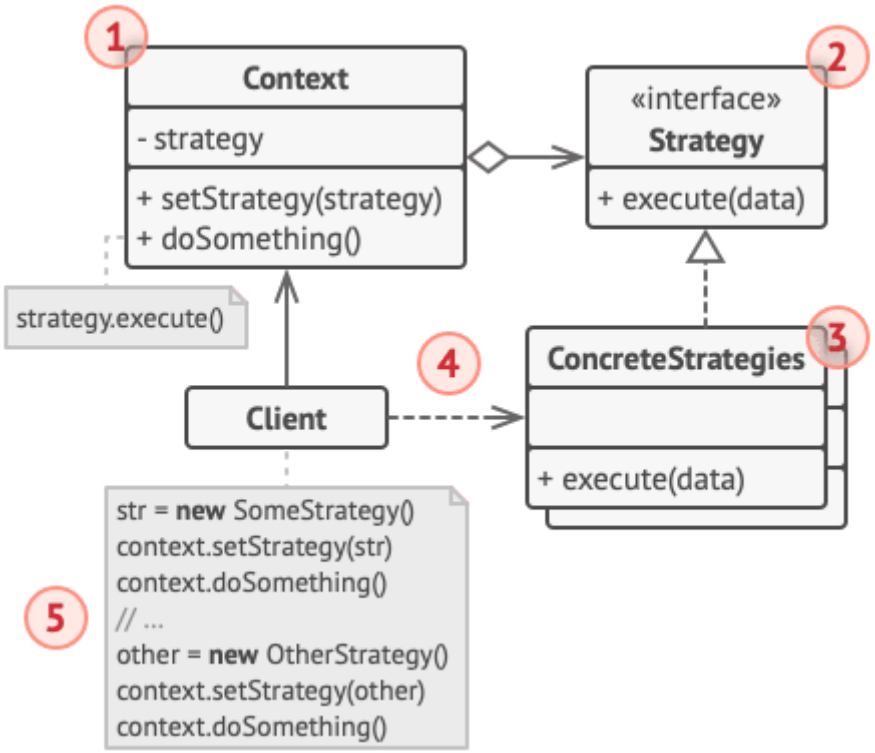
Problema al que aplica:

Cuando tengo algoritmos que están fuertemente acoplados que ante algún cambio se hace difícil de mantener. Cuando tengo el mismo algoritmo para un grupo distinto de clases.

Cómo se soluciona:

Tomo esa clase que hace algo en específico de muchas formas diferentes y extraigo todos esos algoritmos para colocarlos en clases separadas.

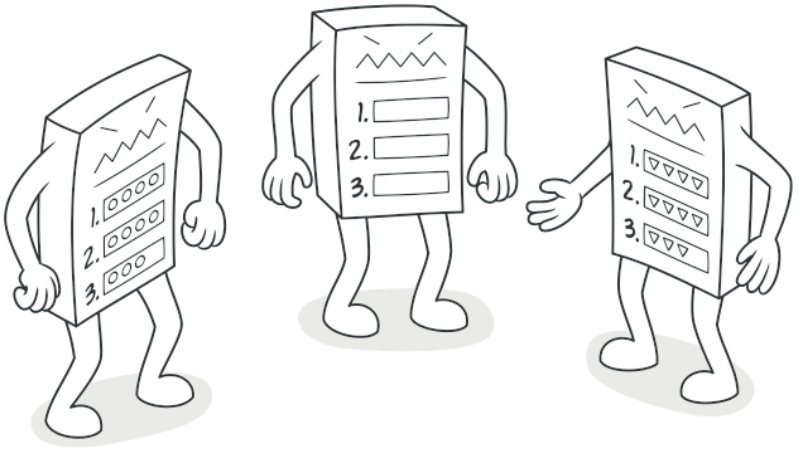
Estructura (en UML)



Ejemplo de código en catálogo.

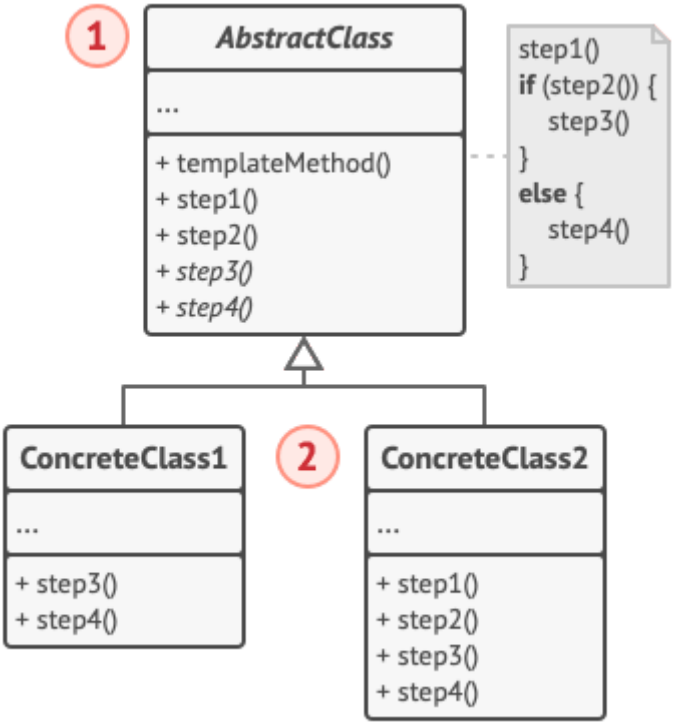
Template Method

Define el esqueleto de un algoritmo en la superclase pero permite que las subclases sobrescriban pasos del algoritmos sin cambiar su estructura.



(es como el meme de los spidermans)

Estructura (en UML)

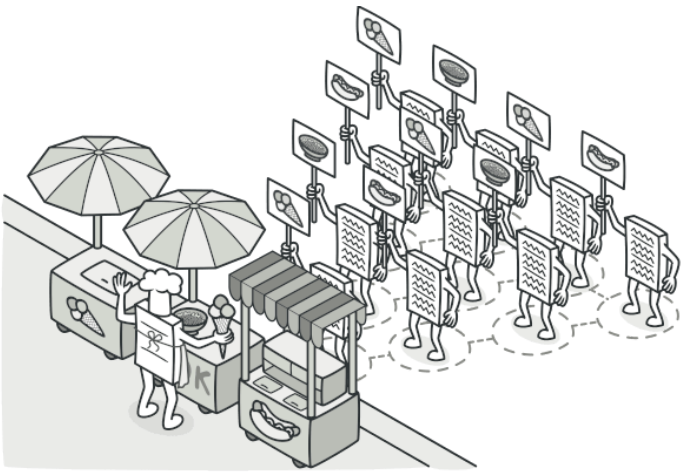


La Clase Abstracta declara métodos que actúan como pasos de un algoritmo, así como el propio método plantilla que invoca estos métodos en un orden específico.
Las Clases Concretas pueden sobrescribir todos los pasos, pero no el propio método plantilla.

Ejemplo de código en catálogo.

Visitor (el último gracias dios)

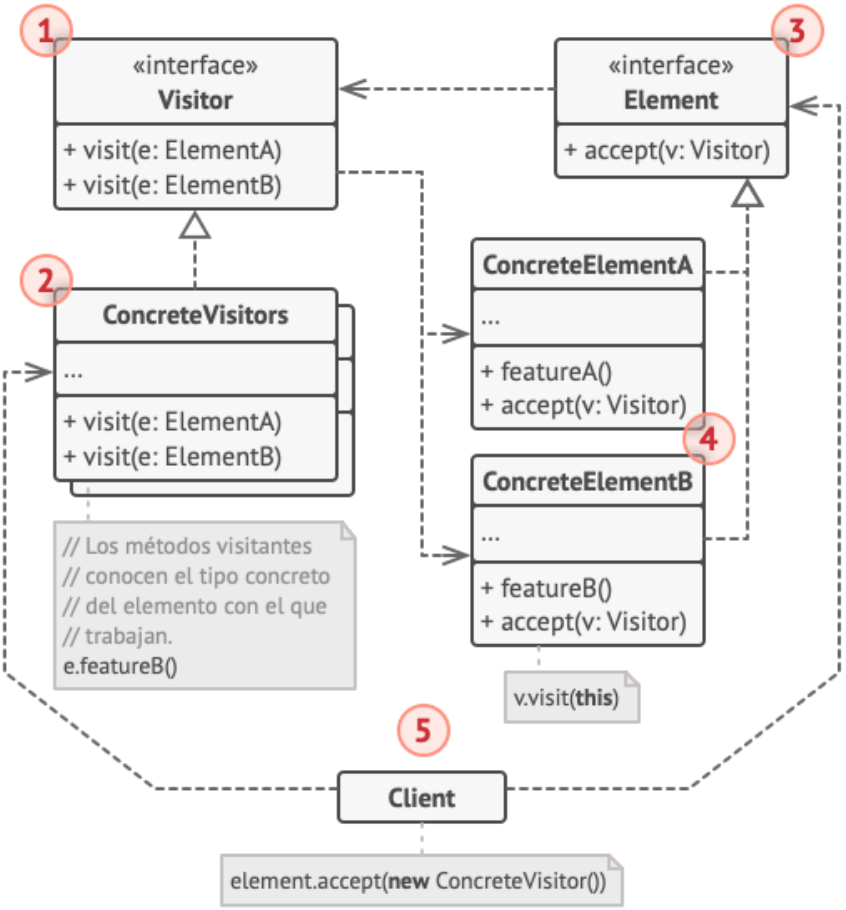
Permite separar algoritmos de los objetos sobre los que operan.



Problema al que aplica:

Cuando necesito realizar una operación sobre todos los elementos de una compleja estructura de objetos.

Estructura (en UML)



Ejemplo de código en catálogo.