Ingeniería de Software II

Patrones de diseño Estructurales

2023

Facultad de Ciencia y Tecnología
Universidad Autónoma de Entre Ríos

Sumérgete en los patrones de diseño

Alexander Shvets, Refactoring.Guru, 2019

support@refactoring.guru

https://refactoring.guru/es/design-patterns

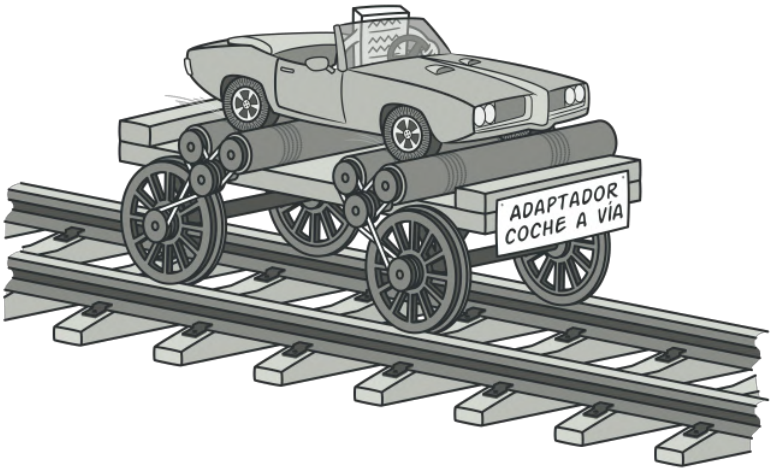
PATRONES ESTRUCTURALES

Adapter

También llamado: Adaptador, Envoltorio, Wrapper

Propósito

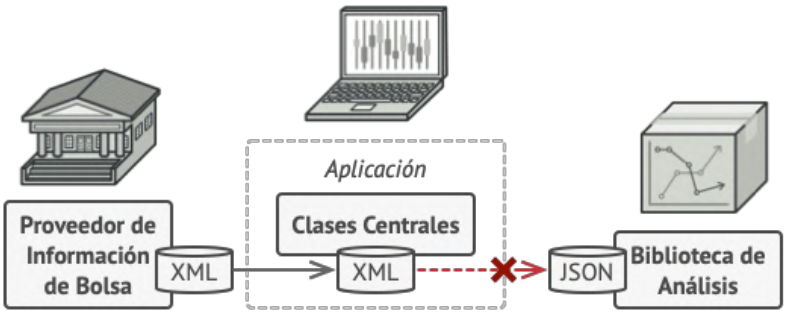
Adapter es un patrón de diseño estructural que permite la colaboración entre objetos con interfaces incompatibles.



Problema

Imagina que estás creando una aplicación de monitoreo del mercado de valores. La aplicación descarga la información de bolsa desde varias fuentes en formato XML para presentarla al usuario con bonitos gráficos y diagramas.

En cierto momento, decides mejorar la aplicación integrando una inteligente biblioteca de análisis de una tercera persona. Pero hay una trampa: la biblioteca de análisis solo funciona con datos en formato JSON.



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

Podrías cambiar la biblioteca para que funcione con XML. Sin embargo, esto podría descomponer parte del código existente que depende de la biblioteca. Y, lo que es peor, podrías no tener siquiera acceso al código fuente de la biblioteca, lo que hace imposible esta solución.

Solución

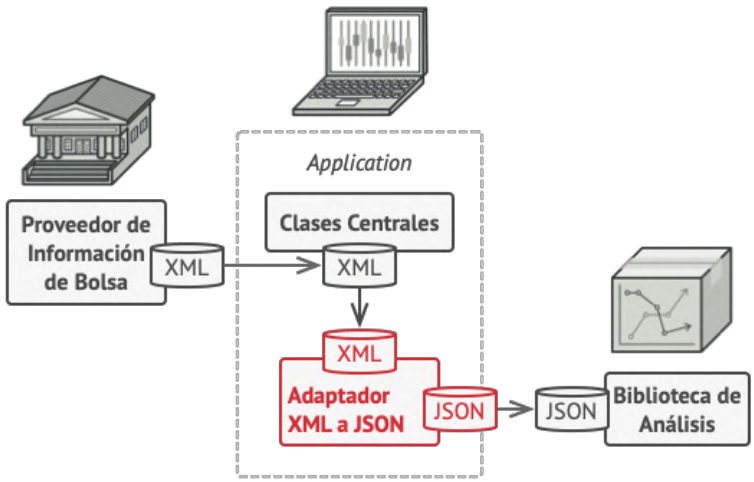
Puedes crear un *adaptador*. Se trata de un objeto especial que convierte la interfaz de un objeto, de forma que otro objeto pueda comprenderla.

Un adaptador envuelve uno de los objetos para esconder la complejidad de la conversión que tiene lugar tras bambalinas. El objeto envuelto ni siquiera es consciente de la existencia del adaptador. Por ejemplo, puedes envolver un objeto que opera con metros y kilómetros con un adaptador que convierte todos los datos al sistema anglosajón, es decir, pies y millas.

Los adaptadores no solo convierten datos a varios formatos, sino que también ayudan a objetos con distintas interfaces a colaborar. Funciona así:

1. El adaptador obtiene una interfaz compatible con uno de los objetos existentes.
2. Utilizando esta interfaz, el objeto existente puede invocar con seguridad los métodos del adaptador.
3. Al recibir una llamada, el adaptador pasa la solicitud al segundo objeto, pero en un formato y orden que ese segundo objeto espera.

En ocasiones se puede incluso crear un adaptador de dos direcciones que pueda convertir las llamadas en ambos sentidos.



Regresemos a nuestra aplicación del mercado de valores. Para resolver el dilema de los formatos incompatibles, puedes crear adaptadores de XML a JSON para cada clase de la biblioteca de análisis con la que trabaje tu código directamente. Después ajustas tu código para que se comunique con la biblioteca únicamente a través de estos adaptadores. Cuando un adaptador recibe una llamada, traduce los datos XML entrantes a una estructura JSON y pasa la llamada a los métodos adecuados de un objeto de análisis envuelto.

Analogía en el mundo real

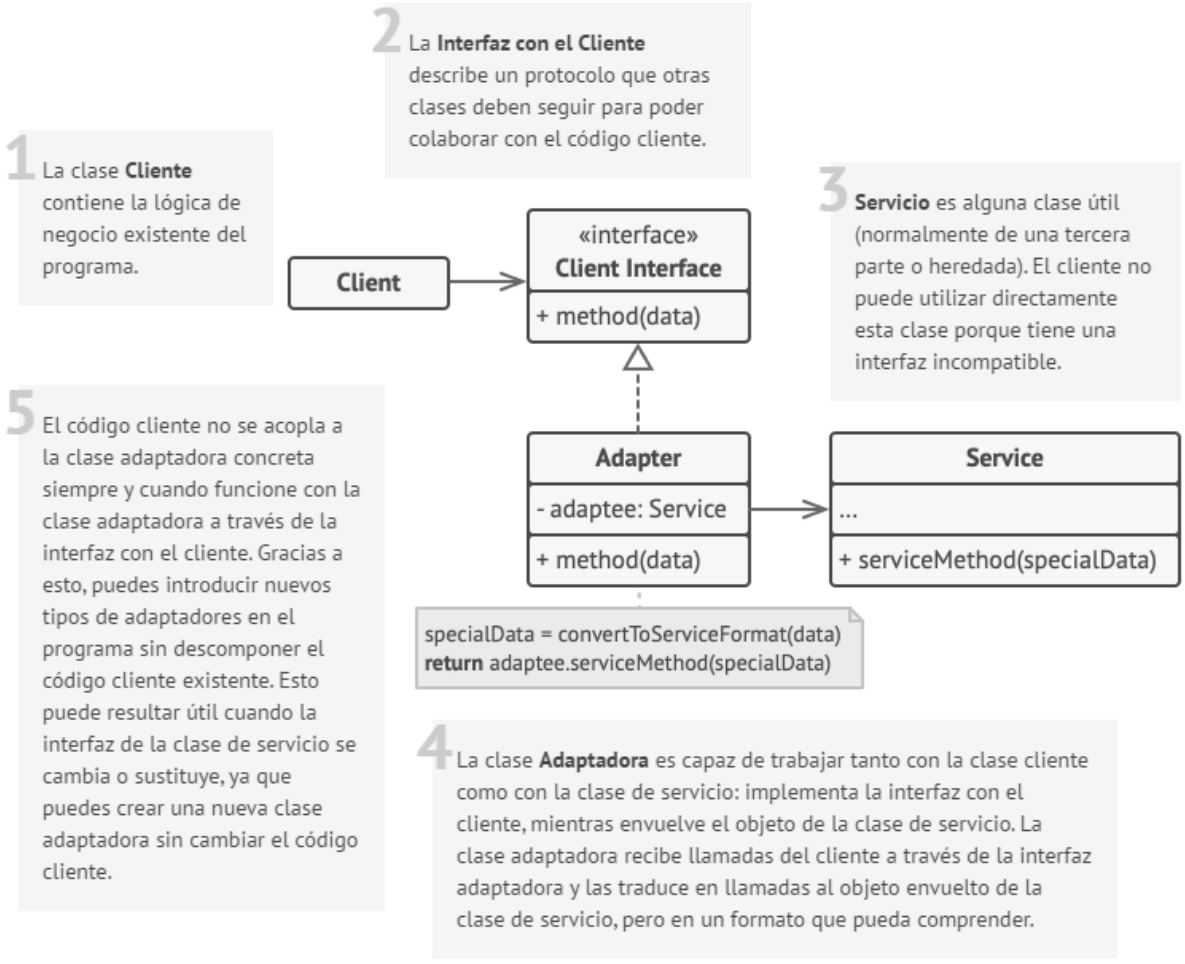
Una maleta antes y después de un viaje al extranjero.

Cuando viajas de Europa a Estados Unidos por primera vez, puede ser que te lleves una sorpresa cuanto intentes cargar tu computadora portátil. Los tipos de enchufe son diferentes en cada país, por lo que un enchufe español no sirve en Estados Unidos. El problema puede solucionarse utilizando un adaptador que incluya el enchufe americano y el europeo.

Estructura

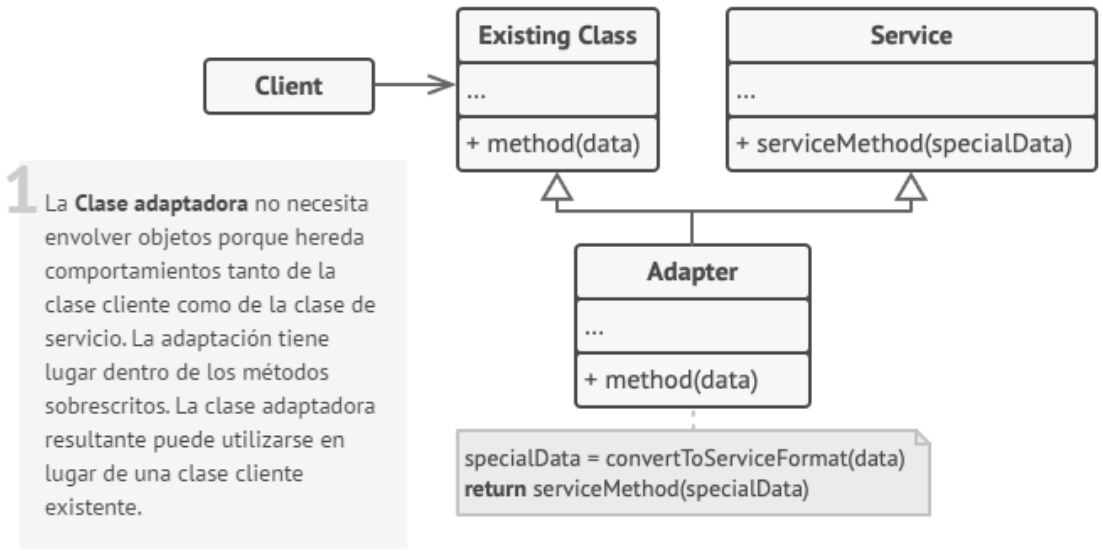
Adaptador de objetos

Esta implementación utiliza el principio de composición de objetos: el adaptador implementa la interfaz de un objeto y envuelve el otro. Puede implementarse en todos los lenguajes de programación populares.



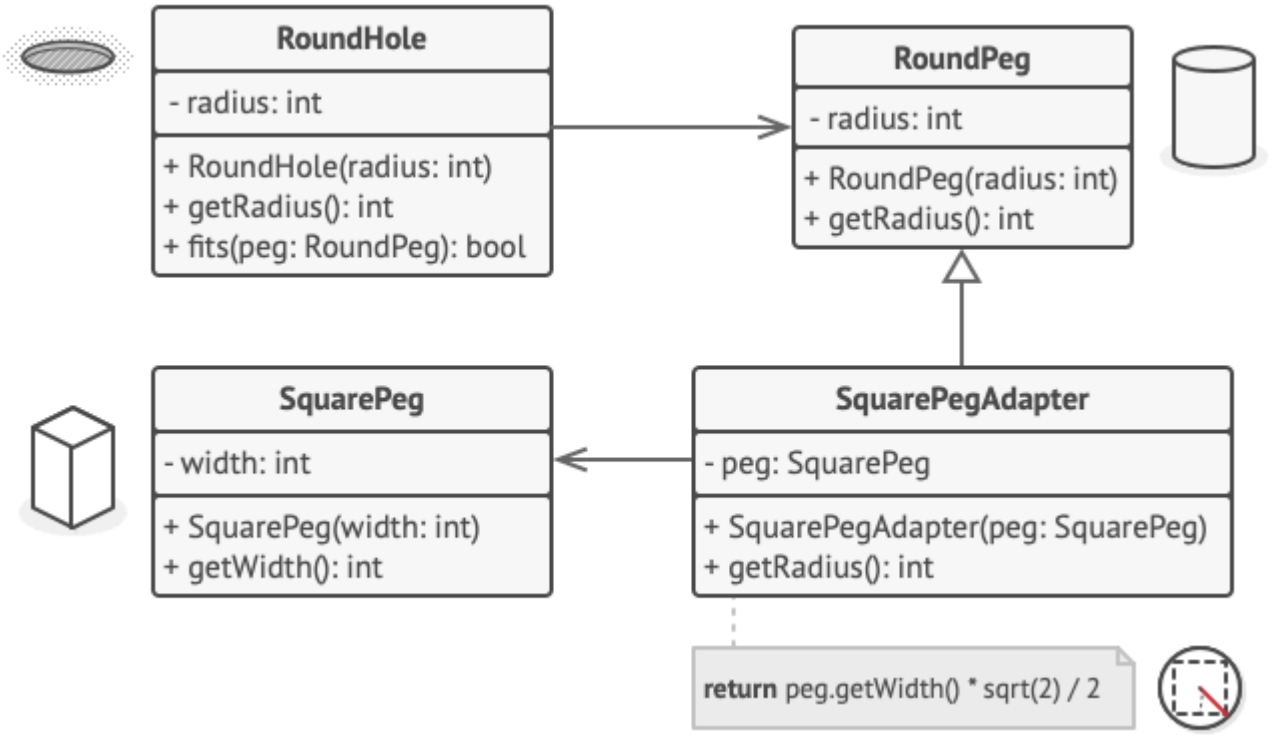
Clase adaptadora

Esta implementación utiliza la herencia, porque la clase adaptadora hereda interfaces de ambos objetos al mismo tiempo. Ten en cuenta que esta opción sólo puede implementarse en lenguajes de programación que soporten la herencia múltiple, como C++.



Pseudocódigo

Este ejemplo del patrón **Adapter** se basa en el clásico conflicto entre piezas cuadradas y agujeros redondos.



Adaptando piezas cuadradas a agujeros redondos.

El patrón Adapter finge ser una pieza redonda con un radio igual a la mitad del diámetro del cuadrado (en otras palabras, el radio del círculo más pequeño en el que quepa la pieza cuadrada).

```
// Digamos que tienes dos clases con interfaces compatibles:
// RoundHole (HoyoRedondo) y RoundPeg (PiezaRedonda).
class RoundHole is
    constructor RoundHole(radius) { ... }

    method getRadius() is
        // Devuelve el radio del agujero.

    method fits(peg: RoundPeg) is
        return this.getRadius() >= peg.getRadius()

class RoundPeg is
    constructor RoundPeg(radius) { ... }

    method getRadius() is
        // Devuelve el radio de la pieza.


// Pero hay una clase incompatible: SquarePeg (PiezaCuadrada).
class SquarePeg is
    constructor SquarePeg(width) { ... }

    method getWidth() is
        // Devuelve la anchura de la pieza cuadrada.


// Una clase adaptadora te permite encajar piezas cuadradas en
// hoyos redondos. Extiende la clase RoundPeg para permitir a
// los objetos adaptadores actuar como piezas redondas.
class SquarePegAdapter extends RoundPeg is
    // En realidad, el adaptador contiene una instancia de la
    // clase SquarePeg.
    private field peg: SquarePeg

    constructor SquarePegAdapter(peg: SquarePeg) is
        this.peg = peg

    method getRadius() is
        // El adaptador simula que es una pieza redonda con un
        // radio que pueda albergar la pieza cuadrada que el
        // adaptador envuelve.
        return peg.getWidth() * Math.sqrt(2) / 2


// En algún punto del código cliente.
hole = new RoundHole(5)
rpeg = new RoundPeg(5)
hole.fits(rpeg) // verdadero

small_sqpeg = new SquarePeg(5)
large_sqpeg = new SquarePeg(10)
hole.fits(small_sqpeg) // esto no compila (tipos incompatibles)

small_sqpeg_adapter = new SquarePegAdapter(small_sqpeg)
large_sqpeg_adapter = new SquarePegAdapter(large_sqpeg)
hole.fits(small_sqpeg_adapter) // verdadero
hole.fits(large_sqpeg_adapter) // falso
```

Aplicabilidad

- **Utiliza la clase adaptadora cuando quieras usar una clase existente, pero cuya interfaz no sea compatible con el resto del código.**
- El patrón Adapter te permite crear una clase intermedia que sirva como traductora entre tu código y una clase heredada, una clase de un tercero o cualquier otra clase con una interfaz extraña.
- **Utiliza el patrón cuando quieras reutilizar varias subclases existentes que carezcan de alguna funcionalidad común que no pueda añadirse a la superclase.**
- Puedes extender cada subclase y colocar la funcionalidad que falta, dentro de las nuevas clases hijas. No obstante, deberás duplicar el código en todas estas nuevas clases, lo cual **huele muy mal**.

  Una solución mucho más elegante sería colocar la funcionalidad que falta dentro de una clase adaptadora. Después puedes envolver objetos a los que les falten funciones, dentro de la clase adaptadora, obteniendo esas funciones necesarias de un modo dinámico. Para que esto funcione, las clases en cuestión deben tener una interfaz común y el campo de la clase adaptadora debe seguir dicha interfaz. Este procedimiento es muy similar al del patrón **Decorator**.

Implementación

1. Asegúrate de que tienes al menos dos clases con interfaces incompatibles:
   - Una útil clase *servicio* que no puedes cambiar (a menudo de un tercero, heredada o con muchas dependencias existentes).
   - Una o varias clases *cliente* que se beneficiarían de contar con una clase de servicio.
2. Declara la interfaz con el cliente y describe el modo en que las clases cliente se comunican con la clase de servicio.
3. Crea la clase adaptadora y haz que siga la interfaz con el cliente. Deja todos los métodos vacíos por ahora.
4. Añade un campo a la clase adaptadora para almacenar una referencia al objeto de servicio. La práctica común es inicializar este campo a través del constructor, pero en ocasiones es adecuado pasarlo al adaptador cuando se invocan sus métodos.
5. Uno por uno, implementa todos los métodos de la interfaz con el cliente en la clase adaptadora. La clase adaptadora deberá delegar la mayor parte del trabajo real al objeto de servicio, gestionando tan solo la interfaz o la conversión de formato de los datos.
6. Las clases cliente deberán utilizar la clase adaptadora a través de la interfaz con el cliente. Esto te permitirá cambiar o extender las clases adaptadoras sin afectar al código cliente.

Pros y contras

✓ *Principio de responsabilidad única*. Puedes separar la interfaz o el código de conversión de datos de la lógica de negocio primaria del programa.

✓ *Principio de abierto/cerrado*. Puedes introducir nuevos tipos de adaptadores al programa sin descomponer el código cliente existente, siempre y cuando trabajen con los adaptadores a través de la interfaz con el cliente.

✗ La complejidad general del código aumenta, ya que debes introducir un grupo de nuevas interfaces y clases. En ocasiones resulta más sencillo cambiar la clase de servicio de modo que coincida con el resto de tu código.

Relaciones con otros patrones

- **Bridge** suele diseñarse por anticipado, lo que te permite desarrollar partes de una aplicación de forma independiente entre sí. Por otro lado, **Adapter** se utiliza habitualmente con una aplicación existente para hacer que unas clases que de otro modo serían incompatibles, trabajen juntas sin problemas.
- **Adapter** cambia la interfaz de un objeto existente mientras que **Decorator** mejora un objeto sin cambiar su interfaz. Además, *Decorator* soporta la composición recursiva, lo cual no es posible al utilizar *Adapter*.

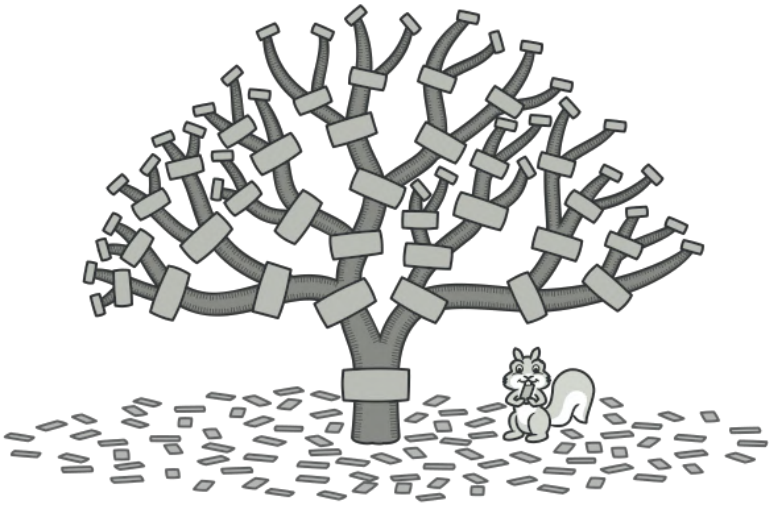
- **Adapter** proporciona una interfaz diferente al objeto envuelto, **Proxy** le proporciona la misma interfaz y **Decorator** le proporciona una interfaz mejorada.
- **Facade** define una nueva interfaz para objetos existentes, mientras que **Adapter** intenta hacer que la interfaz existente sea utilizable. Normalmente *Adapter* sólo envuelve un objeto, mientras que *Facade* trabaja con todo un subsistema de objetos.
- **Bridge**, **State**, **Strategy** (y, hasta cierto punto, **Adapter**) tienen estructuras muy similares. De hecho, todos estos patrones se basan en la composición, que consiste en delegar trabajo a otros objetos. Sin embargo, todos ellos solucionan problemas diferentes. Un patrón no es simplemente una receta para estructurar tu código de una forma específica. También puede comunicar a otros desarrolladores el problema que resuelve.

Composite

También llamado: Objeto compuesto, Object Tree

Propósito

Composite es un patrón de diseño estructural que te permite componer objetos en estructuras de árbol y trabajar con esas estructuras como si fueran objetos individuales.

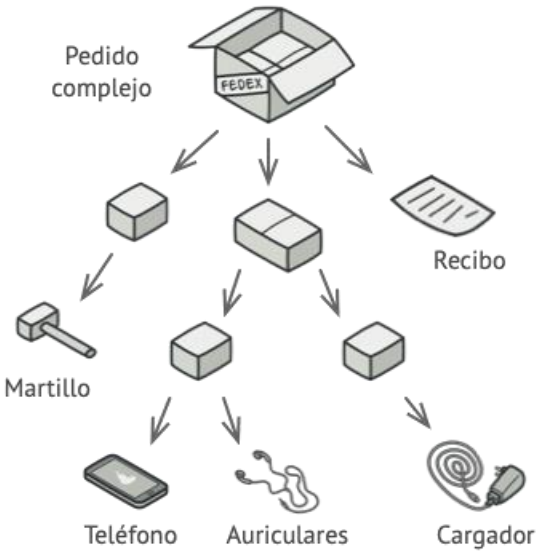


Problema

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

Por ejemplo, imagina que tienes dos tipos de objetos: `Productos` y `Cajas`. Una `Caja` puede contener varios `Productos` así como cierto número de `Cajas` más pequeñas. Estas `Cajas` pequeñas también pueden contener algunos `Productos` o incluso `Cajas` más pequeñas, y así sucesivamente.

Digamos que decides crear un sistema de pedidos que utiliza estas clases. Los pedidos pueden contener productos sencillos sin envolver, así como cajas llenas de productos... y otras cajas. ¿Cómo determinarás el precio total de ese pedido?



Un pedido puede incluir varios productos empaquetados en cajas, que a su vez están empaquetados en cajas más grandes y así sucesivamente. La estructura se asemeja a un árbol boca abajo.

Puedes intentar la solución directa: desenvolver todas las cajas, repasar todos los productos y calcular el total. Esto sería viable en el mundo real; pero en un programa no es tan fácil como ejecutar un bucle. Tienes que conocer de antemano las clases de `Productos` y `Cajas` a iterar, el nivel de anidación de las cajas y otros detalles desagradables. Todo esto provoca que la solución directa sea demasiado complicada, o incluso imposible.

Solución

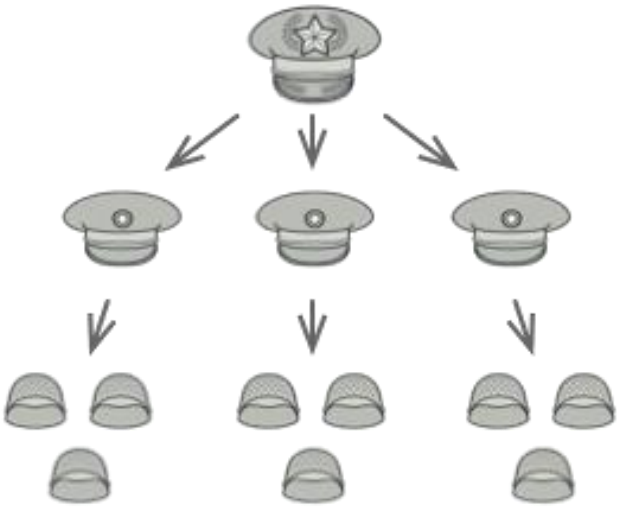
El patrón Composite sugiere que trabajes con `Productos` y `Cajas` a través de una interfaz común que declara un método para calcular el precio total.

¿Cómo funcionaría este método? Para un producto, sencillamente devuelve el precio del producto. Para una caja, recorre cada artículo que contiene la caja, pregunta su precio y devuelve un total por la caja. Si uno de esos artículos fuera una caja más pequeña, esa caja también comenzaría a repasar su contenido y así sucesivamente, hasta que se calcule el precio de todos los componentes internos. Una caja podría incluso añadir costos adicionales al precio final, como costos de empaquetado.

El patrón Composite te permite ejecutar un comportamiento de forma recursiva sobre todos los componentes de un árbol de objetos.

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

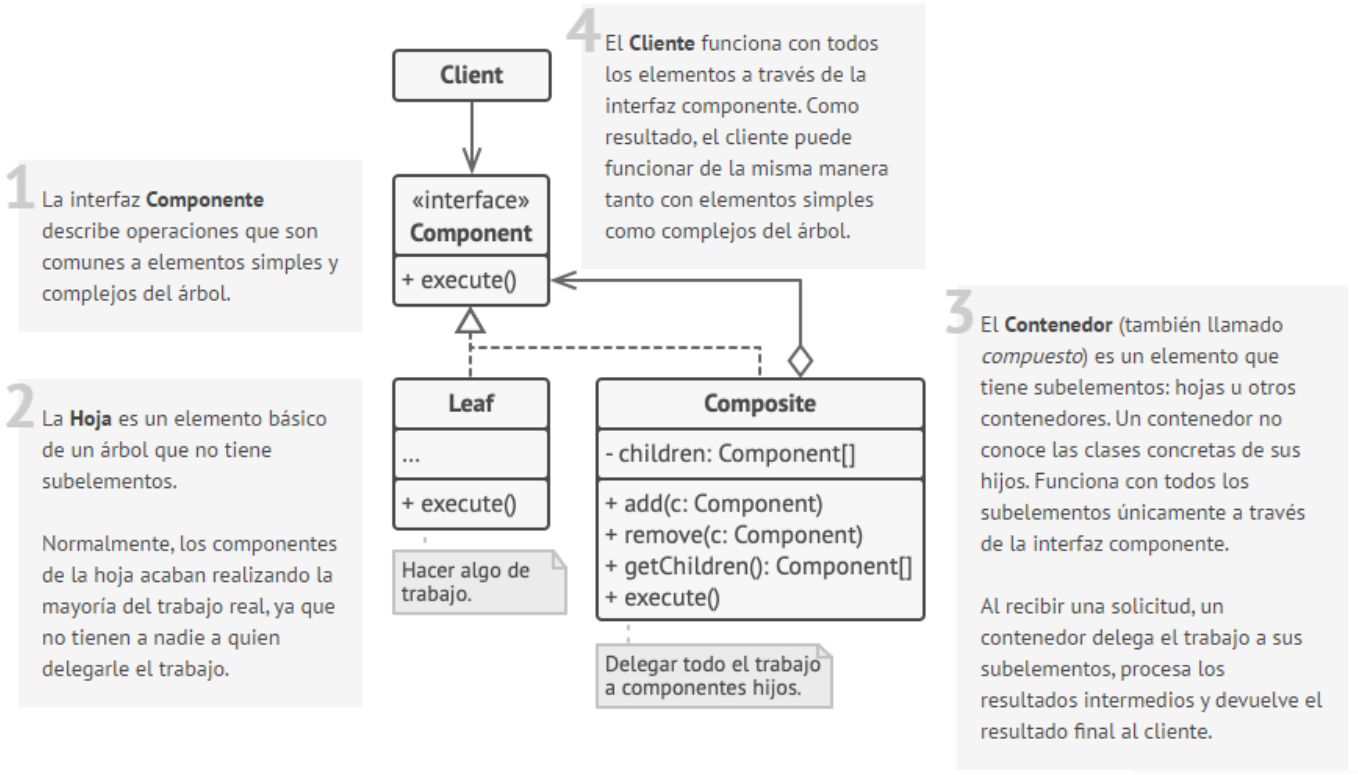
Analogía en el mundo real



Un ejemplo de estructura militar.

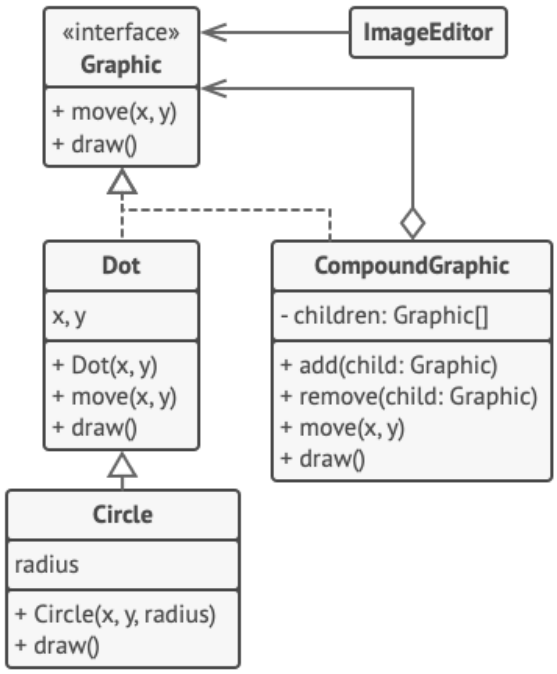
Los ejércitos de la mayoría de países se estructuran como jerarquías. Un ejército está formado por varias divisiones; una división es un grupo de brigadas y una brigada está formada por pelotones, que pueden dividirse en escuadrones. Por último, un escuadrón es un pequeño grupo de soldados reales. Las órdenes se dan en la parte superior de la jerarquía y se pasan hacia abajo por cada nivel hasta que todos los soldados saben lo que hay que hacer.

Estructura



Pseudocódigo

En este ejemplo, el patrón **Composite** te permite implementar el apilamiento (*stacking*) de formas geométricas en un editor gráfico.



Ejemplo del editor de formas geométricas.

La clase `GráficoCompuesto` es un contenedor que puede incluir cualquier cantidad de subformas, incluyendo otras formas compuestas. Una forma compuesta tiene los mismos métodos que una forma simple. Sin embargo, en lugar de hacer algo por su cuenta, una forma compuesta pasa la solicitud de forma recursiva a todos sus hijos y “suma” el resultado.

El código cliente trabaja con todas las formas a través de la interfaz común a todas las clases de forma. De este modo, el cliente no sabe si está trabajando con una forma simple o una compuesta. El cliente puede trabajar con estructuras de objetos muy complejas sin acoplarse a las clases concretas que forman esa estructura.

```
// La interfaz componente declara operaciones comunes para
// objetos simples y complejos de una composición.
interface Graphic is
    method move(x, y)
    method draw()

// La clase hoja representa objetos finales de una composición.
// Un objeto hoja no puede tener ningún subobjeto. Normalmente,
// son los objetos hoja los que hacen el trabajo real, mientras
// que los objetos compuestos se limitan a delegar a sus
// subcomponentes.
class Dot implements Graphic is
    field x, y

    constructor Dot(x, y) { ... }

    method move(x, y) is
        this.x += x, this.y += y

    method draw() is
        // Dibuja un punto en X e Y.

// Todas las clases de componente pueden extender otros
// componentes.
class Circle extends Dot is
    field radius

    constructor Circle(x, y, radius) { ... }

    method draw() is
        // Dibuja un círculo en X y Y con radio R.

// La clase compuesta representa componentes complejos que
// pueden tener hijos. Normalmente los objetos compuestos
// delegan el trabajo real a sus hijos y después "recapitulan"
// el resultado.
class CompoundGraphic implements Graphic is
    field children: array of Graphic

    // Un objeto compuesto puede añadir o eliminar otros
    // componentes (tanto simples como complejos) a o desde su
    // lista hija.
    method add(child: Graphic) is
        // Añade un hijo a la matriz de hijos.

    method remove(child: Graphic) is
        // Elimina un hijo de la matriz de hijos.

    method move(x, y) is
        foreach (child in children) do
            child.move(x, y)

    // Un compuesto ejecuta su lógica primaria de una forma
    // particular. Recorre recursivamente todos sus hijos,
    // recopilando y recapitulando sus resultados. Debido a que
    // los hijos del compuesto pasan esas llamadas a sus propios
    // hijos y así sucesivamente, se recorre todo el árbol de
    // objetos como resultado.
    method draw() is
        // 1. Para cada componente hijo:
        //     - Dibuja el componente.
        //     - Actualiza el rectángulo delimitador.
        // 2. Dibuja un rectángulo de línea punteada utilizando
        // las coordenadas de delimitación.


// El código cliente trabaja con todos los componentes a través
// de su interfaz base. De esta forma el código cliente puede
// soportar componentes de hoja simples así como compuestos
// complejos.
class ImageEditor is
    field all: CompoundGraphic

    method load() is
        all = new CompoundGraphic()
        all.add(new Dot(1, 2))
        all.add(new Circle(5, 3, 10))
        // ...

    // Combina componentes seleccionados para formar un
    // componente compuesto complejo.
    method groupSelected(components: array of Graphic) is
        group = new CompoundGraphic()
        foreach (component in components) do
            group.add(component)
            all.remove(component)
        all.add(group)
        // Se dibujarán todos los componentes.
        all.draw()
```

Aplicabilidad

- ✓ **Utiliza el patrón Composite cuando tengas que implementar una estructura de objetos con forma de árbol.**

  El patrón Composite te proporciona dos tipos de elementos básicos que comparten una interfaz común: hojas simples y contenedores complejos. Un contenedor puede estar compuesto por hojas y por otros contenedores. Esto te permite construir una estructura de objetos recursivos anidados parecida a un árbol.

- ✓ **Utiliza el patrón cuando quieras que el código cliente trate elementos simples y complejos de la misma forma.**

  Todos los elementos definidos por el patrón Composite comparten una interfaz común. Utilizando esta interfaz, el cliente no tiene que preocuparse por la clase concreta de los objetos con los que funciona.

Implementación

1. Asegúrate de que el modelo central de tu aplicación pueda representarse como una estructura de árbol. Intenta dividirlo en elementos simples y contenedores. Recuerda que los contenedores deben ser capaces de contener tanto elementos simples como otros contenedores.
2. Declara la interfaz componente con una lista de métodos que tengan sentido para componentes simples y complejos.
3. Crea una clase hoja para representar elementos simples. Un programa puede tener varias clases hoja diferentes.
4. Crea una clase contenedora para representar elementos complejos. Incluye un campo matriz en esta clase para almacenar referencias a subelementos. La matriz debe poder almacenar hojas y contenedores, así que asegúrate de declararla con el tipo de la interfaz componente.

   Al implementar los métodos de la interfaz componente, recuerda que un contenedor debe delegar la mayor parte del trabajo a los subelementos.

5. Por último, define los métodos para añadir y eliminar elementos hijos dentro del contenedor.

   Ten en cuenta que estas operaciones se pueden declarar en la interfaz componente. Esto violaría el *Principio de segregación de la interfaz* porque los métodos de la clase hoja estarían vacíos. No obstante, el cliente podrá tratar a todos los elementos de la misma manera, incluso al componer el árbol.

Pros y contras

- ✓ Puedes trabajar con estructuras de árbol complejas con mayor comodidad: utiliza el polimorfismo y la recursión en tu favor.
- ✓ *Principio de abierto/cerrado*. Puedes introducir nuevos tipos de elemento en la aplicación sin descomponer el código existente, que ahora funciona con el árbol de objetos.
- ✗ Puede resultar difícil proporcionar una interfaz común para clases cuya funcionalidad difiere demasiado. En algunos casos, tendrás que generalizar en exceso la interfaz componente, provocando que sea más difícil de comprender.

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** se utiliza a menudo junto a **Composite**. En este caso, 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.
- Puedes utilizar **Iteradores** para recorrer árboles **Composite**.
- Puedes utilizar el patrón **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 para organizar un número indefinido de objetos.

  Un *Decorator* es como un *Composite* pero sólo tiene un componente hijo. Hay otra diferencia importante: *Decorator* añade responsabilidades adicionales al objeto envuelto, mientras que *Composite* se limita a "recapitular" los resultados de sus hijos.
  No obstante, los patrones también pueden colaborar: puedes utilizar el *Decorator* para extender el comportamiento de un objeto específico del árbol *Composite*.

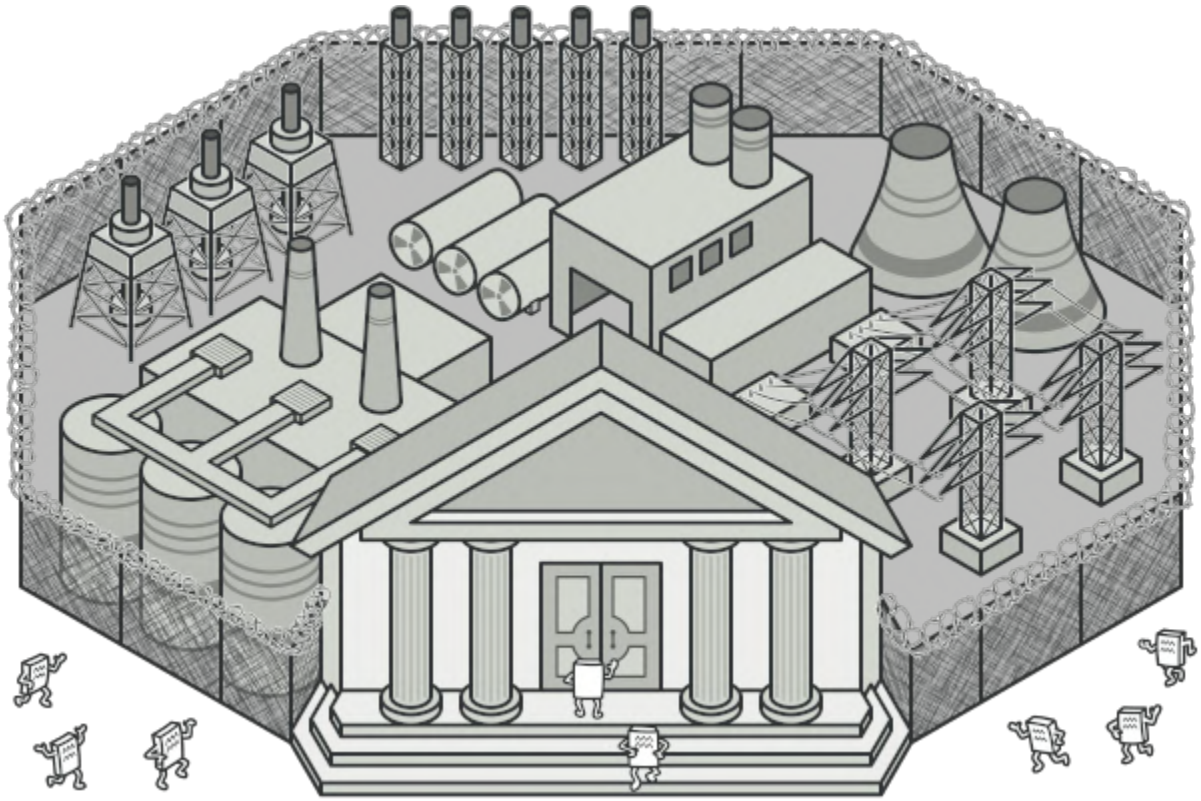
- Los diseños que hacen un uso amplio de **Composite** y **Decorator** a menudo pueden beneficiarse del uso del **Prototype**. Aplicar el patrón te permite clonar estructuras complejas en lugar de reconstruirlas desde cero.

Facade

También llamado: Fachada

Propósito

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



Problema

Imagina que debes lograr que tu código trabaje con un amplio grupo de objetos que pertenecen a una sofisticada biblioteca o *framework*. Normalmente, debes inicializar todos esos objetos, llevar un registro de las dependencias, ejecutar los métodos en el orden correcto y así sucesivamente.

Como resultado, la lógica de negocio de tus clases se vería estrechamente acoplada a los detalles de implementación de las clases de terceros, haciéndola difícil de comprender y mantener.

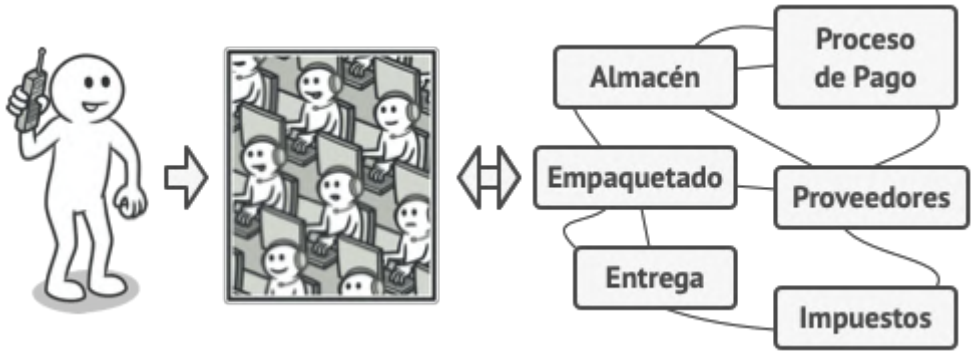
Solución

Una fachada es una clase que proporciona una interfaz simple a un subsistema complejo que contiene muchas partes móviles. Una fachada puede proporcionar una funcionalidad limitada en comparación con trabajar directamente con el subsistema. Sin embargo, tan solo incluye las funciones realmente importantes para los clientes.

Tener una fachada resulta útil cuando tienes que integrar tu aplicación con una biblioteca sofisticada con decenas de funciones, de la cual sólo necesitas una pequeña parte.

Por ejemplo, una aplicación que sube breves vídeos divertidos de gatos a las redes sociales, podría potencialmente utilizar una biblioteca de conversión de vídeo profesional. Sin embargo, lo único que necesita en realidad es una clase con el método simple `codificar(nombreDelArchivo, formato)`. Una vez que crees dicha clase y la conectes con la biblioteca de conversión de vídeo, tendrás tu primera fachada.

Analogía en el mundo real



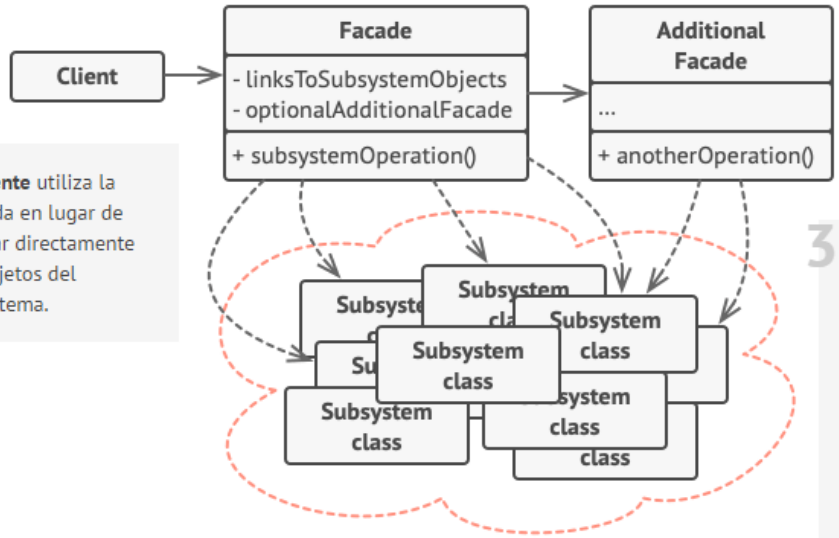
Haciendo pedidos por teléfono.

Cuando llamas a una tienda para hacer un pedido por teléfono, un operador es tu fachada a todos los servicios y departamentos de la tienda. El operador te proporciona una sencilla interfaz de voz al sistema de pedidos, pasarelas de pago y varios servicios de entrega.

Estructura

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 Puede crearse una clase **Fachada Adicional** para evitar contaminar una única fachada con funciones no relacionadas que podrían convertirla en otra estructura compleja. Las fachadas adicionales pueden utilizarse por clientes y por otras fachadas.



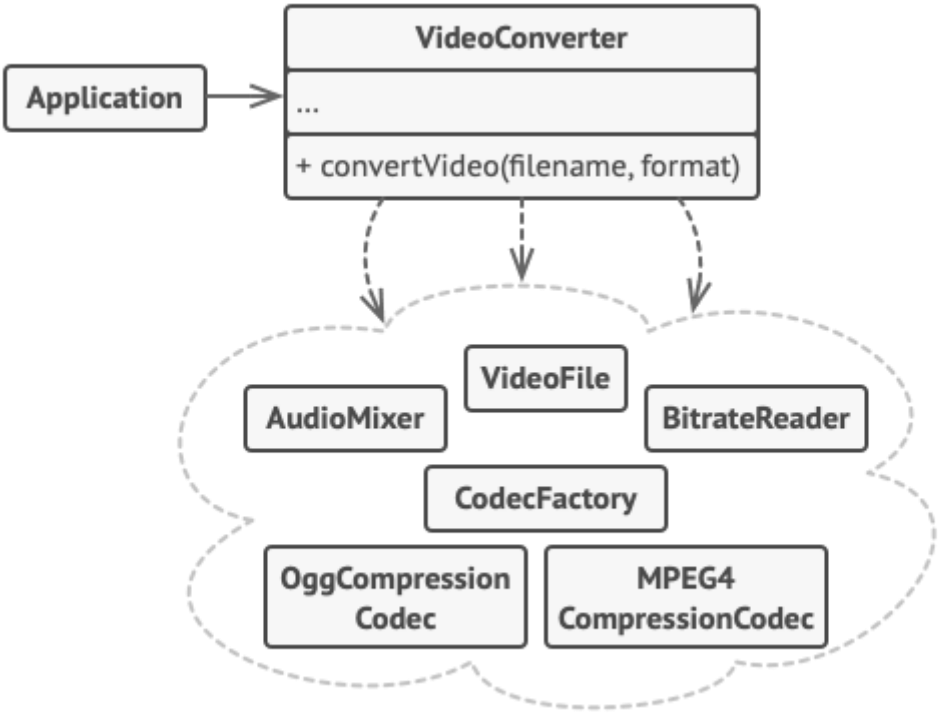
4 El **Cliente** utiliza la fachada en lugar de invocar directamente los objetos del subsistema.

3 El **Subsistema Complejo** consiste en decenas de objetos diversos. Para lograr que todos hagan algo significativo, debes profundizar en los detalles de implementación del subsistema, que pueden incluir inicializar objetos en el orden correcto y suministrarles datos en el formato adecuado.

Las clases del subsistema no conocen la existencia de la fachada. Operan dentro del sistema y trabajan entre sí directamente.

Pseudocódigo

En este ejemplo, el patrón **Facade** simplifica la interacción con un framework complejo de conversión de vídeo.



Un ejemplo de aislamiento de múltiples dependencias dentro de una única clase fachada.

En lugar de hacer que tu código trabaje con decenas de las clases del framework directamente, creas una clase fachada que encapsula esa funcionalidad y la esconde del resto del código. Esta estructura también te ayuda a minimizar el esfuerzo de actualizar a futuras versiones del framework o de sustituirlo por otro. Lo único que tendrías que cambiar en la aplicación es la implementación de los métodos de la fachada.

```
// Estas son algunas de las clases de un framework de conversión
// de video de un tercero. No controlamos ese código, por lo que
// no podemos simplificarlo.

class VideoFile
// ...

class OggCompressionCodec
// ...

class MPEG4CompressionCodec
// ...

class CodecFactory
// ...

class BitrateReader
// ...

class AudioMixer
// ...


// Creamos una clase fachada para esconder la complejidad del
// framework tras una interfaz simple. Es una solución de
// equilibrio entre funcionalidad y simplicidad.
class VideoConverter is
    method convert(filename, format):File is
        file = new VideoFile(filename)
        sourceCodec = (new CodecFactory).extract(file)
        if (format == "mp4")
            destinationCodec = new MPEG4CompressionCodec()
        else
            destinationCodec = new OggCompressionCodec()
        buffer = BitrateReader.read(filename, sourceCodec)
        result = BitrateReader.convert(buffer, destinationCodec)
        result = (new AudioMixer()).fix(result)
        return new File(result)

// Las clases Application no dependen de un millón de clases
// proporcionadas por el complejo framework. Además, si decides
// cambiar los frameworks, sólo tendrás de volver a escribir la
// clase fachada.
class Application is
    method main() is
        convertor = new VideoConverter()
        mp4 = convertor.convert("funny-cats-video.ogg", "mp4")
        mp4.save()
```

Aplicabilidad

✓ **Utiliza el patrón Facade cuando necesites una interfaz limitada pero directa a un subsistema complejo.**

A menudo los subsistemas se vuelven más complejos con el tiempo. Incluso la aplicación de patrones de diseño suele conducir a la creación de un mayor número de clases. Un subsistema puede hacerse más flexible y más fácil de reutilizar en varios contextos, pero la cantidad de código de configuración que exige de un cliente, crece aún más. El patrón Facade intenta solucionar este problema proporcionando un atajo a las funciones más utilizadas del subsistema que mejor encajan con los requisitos del cliente.

✓ **Utiliza el patrón Facade cuando quieras estructurar un subsistema en capas.**

Crea fachadas para definir puntos de entrada a cada nivel de un subsistema. Puedes reducir el acoplamiento entre varios subsistemas exigiéndoles que se comuniquen únicamente mediante fachadas.

Por ejemplo, regresemos a nuestro framework de conversión de vídeo. Puede dividirse en dos capas: la relacionada con el vídeo y la relacionada con el audio. Puedes crear una fachada para cada capa y hacer que las clases de cada una de ellas se comuniquen entre sí a través de esas fachadas. Este procedimiento es bastante similar al patrón **Mediator**.

Implementación

1. Comprueba si es posible proporcionar una interfaz más simple que la que está proporcionando un subsistema existente. Estás bien encaminado si esta interfaz hace que el código cliente sea independiente de muchas de las clases del subsistema.
2. Declara e implementa esta interfaz en una nueva clase fachada. La fachada deberá redireccionar las llamadas desde el código cliente a los objetos adecuados del subsistema. La fachada deberá ser responsable de inicializar el subsistema y gestionar su ciclo de vida, a no ser que el código cliente ya lo haga.
3. Para aprovechar el patrón al máximo, haz que todo el código cliente se comunique con el subsistema únicamente a través de la fachada. Ahora el código cliente está protegido de cualquier cambio en el código del subsistema. Por ejemplo, cuando se actualice un subsistema a una nueva versión, sólo tendrás que modificar el código de la fachada.
4. Si la fachada se vuelve **demasiado grande**, piensa en extraer parte de su comportamiento y colocarlo dentro de una nueva clase fachada refinada.

Pros y contras

✓ Puedes aislar tu código de la complejidad de un subsistema.

✗ Una fachada puede convertirse en **un objeto todopoderoso** acoplado a todas las clases de una aplicación.

Relaciones con otros patrones

- **Facade** define una nueva interfaz para objetos existentes, mientras que **Adapter** intenta hacer que la interfaz existente sea utilizable. Normalmente *Adapter* sólo envuelve un objeto, mientras que *Facade* trabaja con todo un subsistema de objetos.
- **Abstract Factory** puede servir como alternativa a **Facade** cuando tan solo deseas esconder la forma en que se crean los objetos del subsistema a partir del código cliente.
- **Flyweight** muestra cómo crear muchos pequeños objetos, mientras que **Facade** muestra cómo crear un único objeto que represente un subsistema completo.
- **Facade** y **Mediator** tienen trabajos similares: ambos intentan organizar la colaboración entre muchas clases estrechamente acopladas.

    - *Facade* define una interfaz simplificada a un subsistema de objetos, pero no introduce ninguna nueva funcionalidad. El propio subsistema no conoce la fachada. Los objetos del subsistema pueden comunicarse directamente.
    - *Mediator* centraliza la comunicación entre componentes del sistema. Los componentes conocen únicamente el objeto mediador y no se comunican directamente.
- Una clase **fachada** a menudo puede transformarse en una **Singleton**, ya que un único objeto fachada es suficiente en la mayoría de los casos.
- **Facade** es similar a **Proxy** en el sentido de que ambos pueden almacenar temporalmente una entidad compleja e inicializarla por su cuenta. Al contrario que *Facade*, *Proxy* tiene la misma interfaz que su objeto de servicio, lo que hace que sean intercambiables.