



Ayuda a Ucrania a detener a Rusia

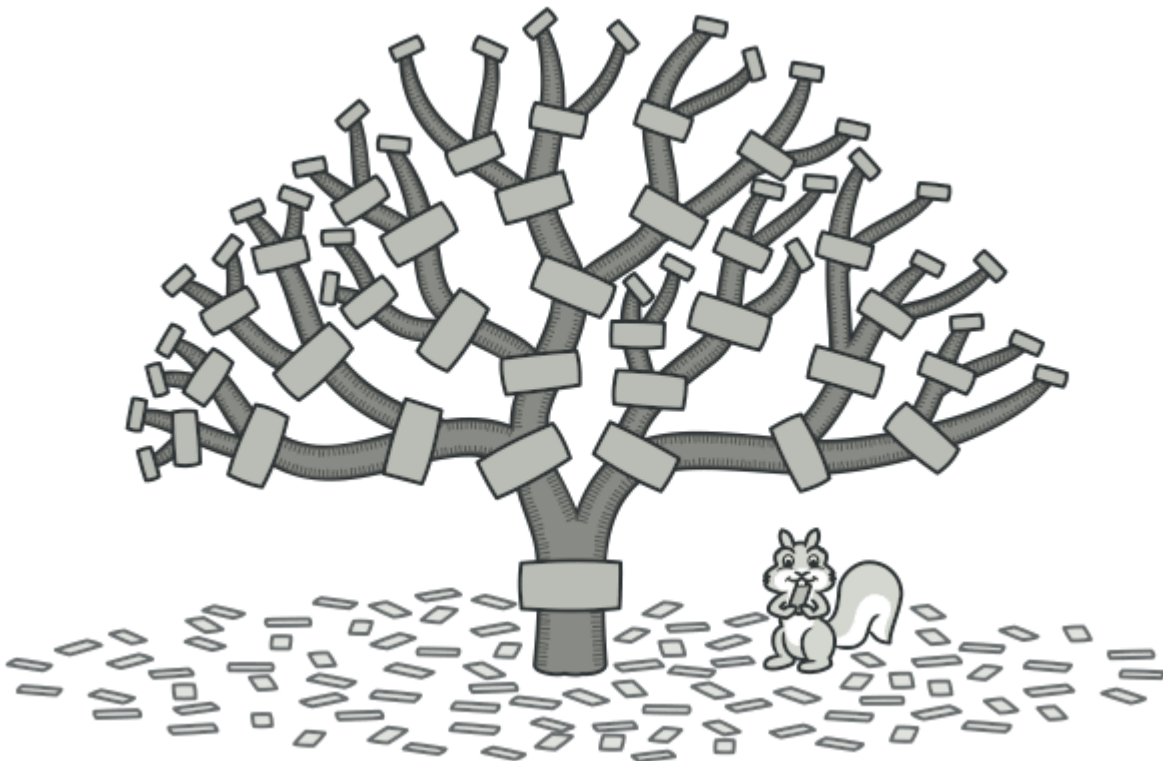
[🏠](#) / [Patrones de diseño](#) / [Patrones estructurales](#)

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.

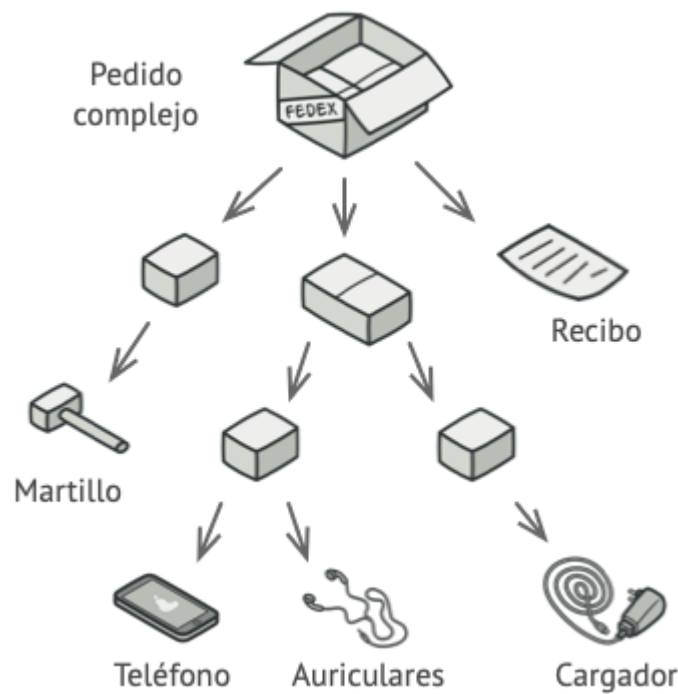


Problema

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.

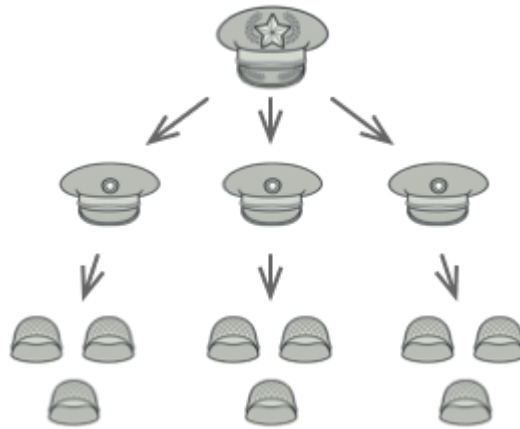
¿Cómo funcionaría este método? Para un producto, sencillamente devuelve el precio del producto. Para una caja, recorre cada artículo que contiene la caja, pregunta su precio y devuelve un total por la caja. Si uno de esos artículos fuera una caja más pequeña, esa caja también comenzaría a repasar su 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.

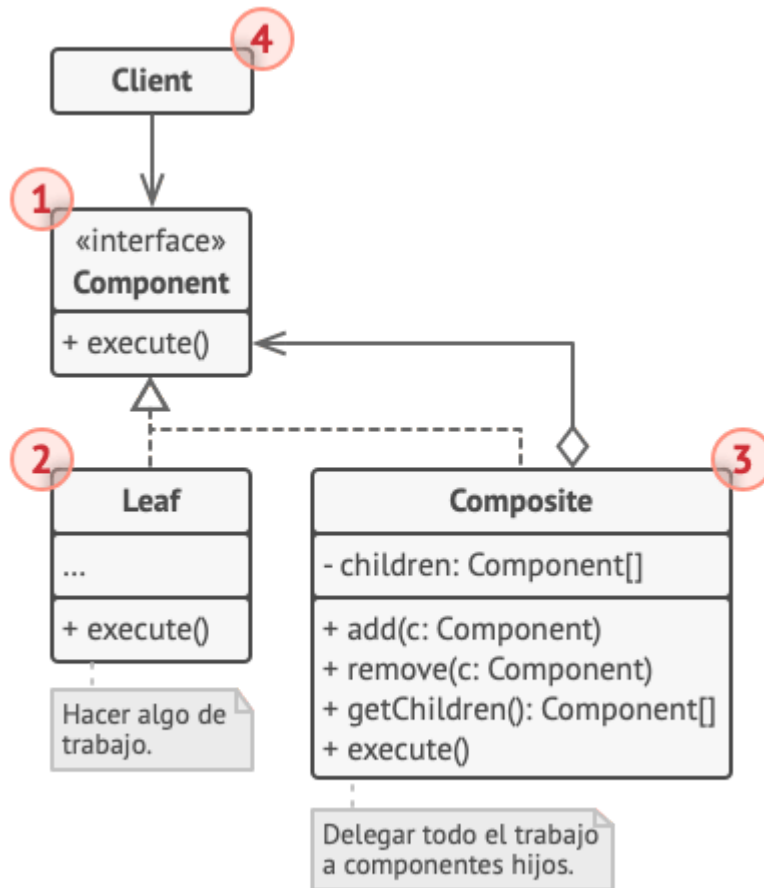
Analogía en el mundo real



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



1. La interfaz **Componente** describe operaciones que son comunes a elementos simples y complejos del árbol.

2. La **Hoja** es un elemento básico de un árbol que no tiene subelementos.

Normalmente, los componentes de la hoja acaban realizando la mayoría del trabajo real, ya que no tienen a nadie a quien delegarle el trabajo.

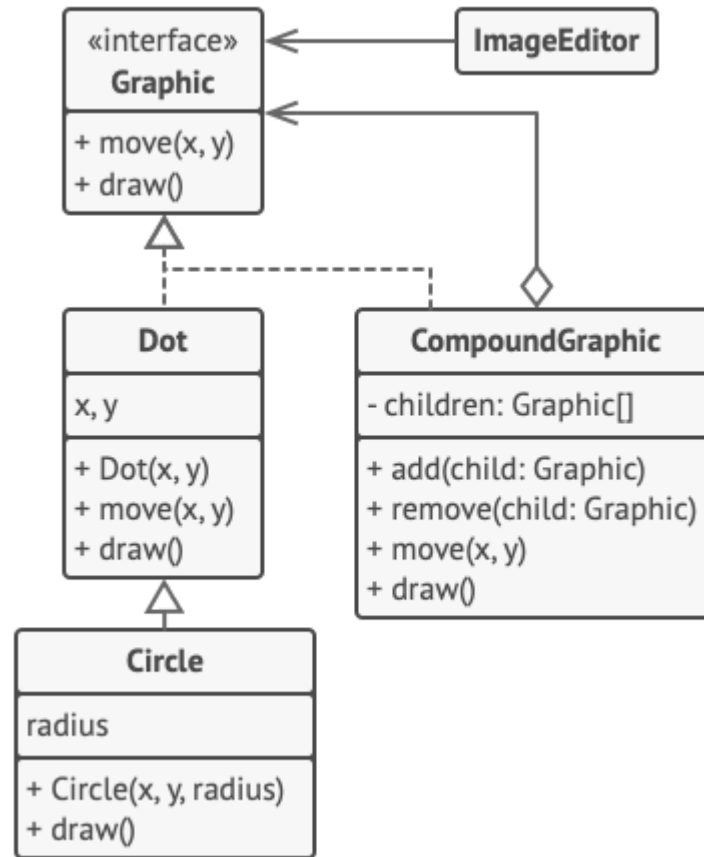
3. El **Contenedor** (también llamado *compuesto*) es un elemento que tiene subelementos: hojas u otros contenedores. Un contenedor no conoce las clases concretas de sus hijos. Funciona con todos los subelementos únicamente a través de la interfaz componente.

Al recibir una solicitud, un contenedor delega el trabajo a sus subelementos, procesa los resultados intermedios y devuelve el resultado final al cliente.

4. El **Cliente** funciona con todos los elementos a través de la interfaz componente. Como resultado, el cliente puede funcionar de la misma manera tanto con elementos simples como complejos del árbol.

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.

📋 Cómo implementarlo

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

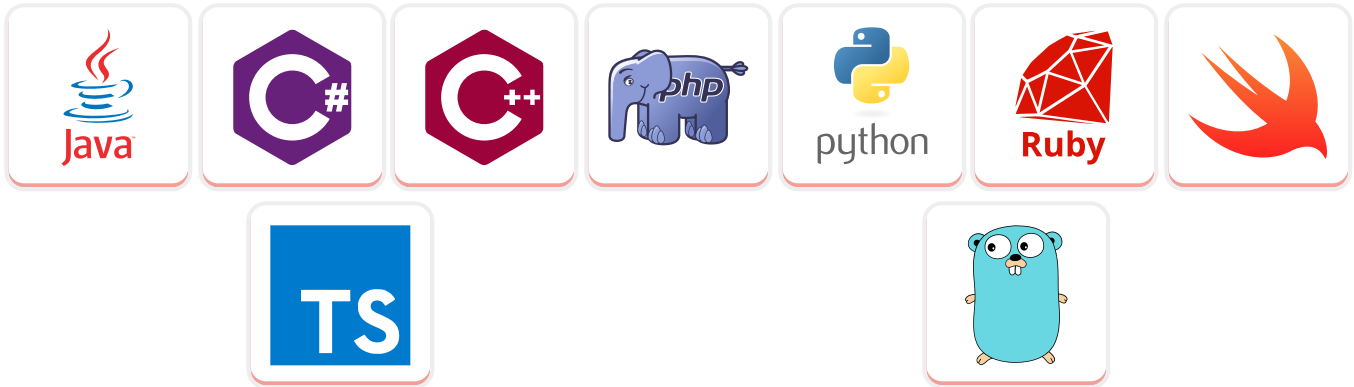
- Puedes utilizar **Builder** al crear árboles **Composite** complejos porque puedes programar sus pasos de construcción para que funcionen de forma recursiva.
- **Chain of Responsibility** se utiliza a menudo junto a **Composite**. En este caso, cuando un componente hoja recibe una solicitud, puede 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.

</> Ejemplos de código



¿Por qué debes llevar este eBook en tus desplazamientos?

- Aprenderás algo útil de camino al trabajo
- No necesita internet: siempre a mano y consultable
- Respetuoso con tu vista con una variedad de modos de lectura
- Una cosa menos que llevar y poder olvidar
- Todos los dispositivos soportados: Formatos PDF/EPUB/MOBI/KFX

 Saber más...

[Inicio](#)



[Refactorización](#)



[Patrones de diseño](#)




[Contenido Premium](#)

[Foro](#)

[Contáctanos](#)

© 2014-2022 Refactoring.Guru. Todos los derechos reservados

 Ilustraciones por Dmitry Zhart

 Khmelnytske shosse 19 / 27, Kamianets-Podilskyi, Ucrania, 32305

 Email: support@refactoring.guru

[Términos y condiciones](#)

[Política de privacidad](#)

[Política de uso de contenido](#)