**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?

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.

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.

**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.

**Pros**

* 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.

## Contras

* 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.