PATRONES DE DISEÑO

**¿Qué es un patrón de diseño?**

Los patrones de diseño son soluciones habituales a problemas que ocurren con frecuencia en el diseño de software. Son como planos prefabricados que se pueden personalizar para resolver un problema de diseño recurrente en tu código.

No se puede elegir un patrón y copiarlo en el programa como si se tratara de funciones o bibliotecas ya preparadas. El patrón no es una porción específica de código, sino un concepto general para resolver un problema particular. Puedes seguir los detalles del patrón e implementar una solución que encaje con las realidades de tu propio programa.

**¿En qué consiste el patrón?**

La mayoría de los patrones se describe con mucha formalidad para que la gente pueda reproducirlos en muchos contextos.

* El propósito del patrón explica brevemente el problema y la solución.
* La motivación explica en más detalle el problema y la solución que brinda el patrón.
* La estructura de las clases muestra cada una de las partes del patrón y el modo en que se relacionan.
* El ejemplo de código en uno de los lenguajes de programación populares facilita la asimilación de la idea que se esconde tras el patrón.

**Clasificación de los patrones**

* Los **patrones creacionales** proporcionan mecanismos de creación de objetos que incrementan la flexibilidad y la reutilización de código existente.
* Los **patrones estructurales** explican cómo ensamblar objetos y clases en estructuras más grandes a la vez que se mantiene la flexibilidad y eficiencia de la estructura.
* Los **patrones de comportamiento** se encargan de una comunicación efectiva y la asignación de responsabilidades entre objetos.

**¿Por qué deberíamos aprender sobre patrones?**

* Los patrones de diseño son un juego de herramientas de soluciones comprobadas a problemas habituales en el diseño de software. Incluso aunque nunca te encuentres con estos problemas, conocer los patrones sigue siendo de utilidad, porque te enseña a resolver todo tipo de problemas utilizando principios del diseño orientado a objetos.
* Los patrones de diseño definen un lenguaje común que puedes utilizar con tus compañeros de equipo para comunicaros de forma más eficiente. Podrías decir: “Oh, utiliza un singleton para eso”, y todos entenderían la idea de tu sugerencia.

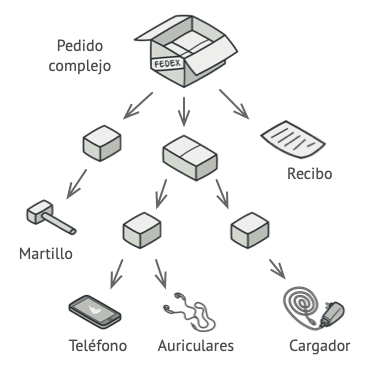
**PATRON DE DISEÑO COMPOSITE**

**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.



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.

**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 sentida para componentes simples y complejos.
3. Crea una clase hoja para representar elementos simples. Un programa puede tener varias clases hojas 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.
5. Al implementar los métodos de la interfaz componente, recuerda que un contenedor debe delegar la mayor parte del trabajo a los subelementos.
6. Por último, define los métodos para añadir y eliminar elementos hijos dentro del contenedor.

**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.

**COMPOSITE**

* Nos permite crear jerarquías estructuradas con componentes sencillos o grupos de componentes, a los que se les puede aplicar las mismas operaciones.
* El componente puede ser un objeto sencillo o una colección de otros componentes
* Las operaciones más comunes son: adicionar, remover, agrupar, buscar
* Se utiliza donde necesitamos tener a los elementos de forma organizada o jerárquica
* Trabajamos con dos tipos de elementos: el componente y el compuesto
* Ambos tipos de elementos tienen una interfaz común
* Permite adicionar nuevos tipos de componentes siempre y cuando implementen a la interfaz

**Variaciones**

* Orden-> Los elementos son organizados
* Cache-> Si un valor se calcula constantemente a partir de los hijos, puede guardarse en el padre para no tener que calcularlo constantemente

**Elementos principales**

* Interfaz componente
  + Interfaz que representa a los comportamientos que serán comunes a todos los objetos de la composición
* Operación
  + Método que lleva a cabo la operación que esperamos en un elemento de la composición
* Componente
  + Representa un objeto sencillo que no se puede descomponer más e implementa las operaciones según su caso
* Compuesto
  + Implementa las operaciones trabajando con una colección de componentes

