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, 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.

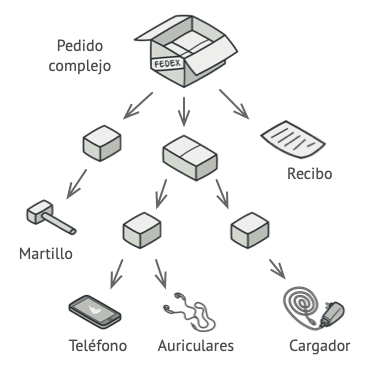
**PATRON DE DISEÑO COMPOSITE**

**Propósito:** El patrón Composite permite construir y trabajar con estructuras jerárquicas en forma de árbol, tratando elementos individuales y grupos de elementos de manera uniforme.

**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**

Útil cuando se necesita representar una estructura jerárquica. El patrón ofrece hojas simples y contenedores complejos con una interfaz común, permitiendo que el cliente trate ambos de manera uniforme.

**Implementación**

1. Divide el modelo en componentes simples y compuestos.
2. Crea una interfaz componente con métodos para ambos tipos.
3. Implementa clases hoja para componentes simples.
4. Crea clases compuestas con referencias a subelementos y delegación de trabajo.
5. Implementa los métodos de la interfaz componente, delegando en subelementos.
6. Define métodos para agregar/eliminar elementos en contenedores.

**Pros**

* Manejo sencillo de estructuras jerárquicas.
* Cumple el principio de abierto/cerrado al permitir nuevas adiciones sin modificar el código existente.

**Contras**

* Puede ser complicado proporcionar una interfaz común para componentes con funcionalidades muy diferentes.

**Elementos Clave**

* Interfaz Componente: Define operaciones comunes.
* Operación: Realiza acciones específicas en un componente.
* Componente: Representa objetos individuales.
* Compuesto: Representa objetos compuestos que contienen componentes.

