Patrón: Composite

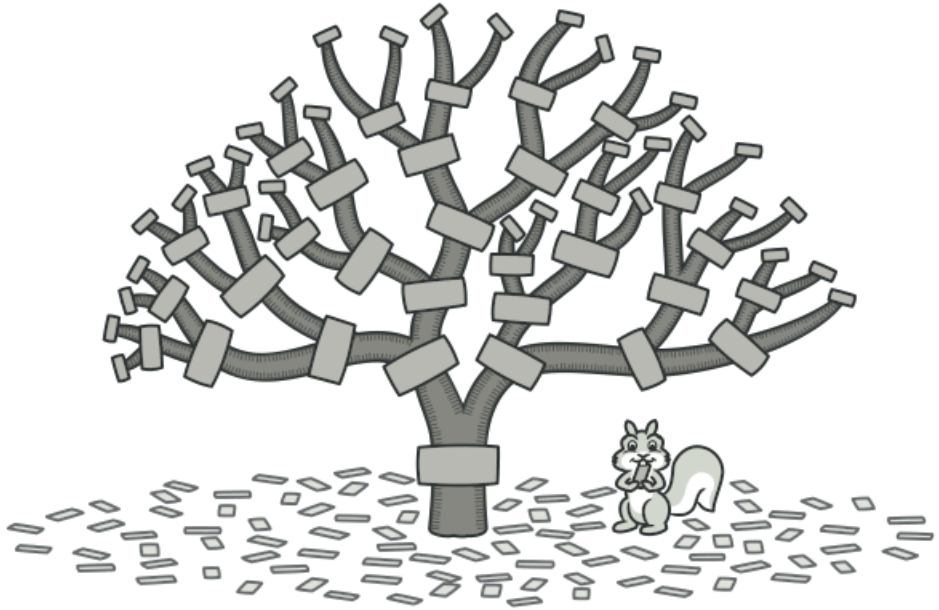
Patrón estructural (POO)

Fuente principal:
https://refactoring.guru/es/design-patterns/composite

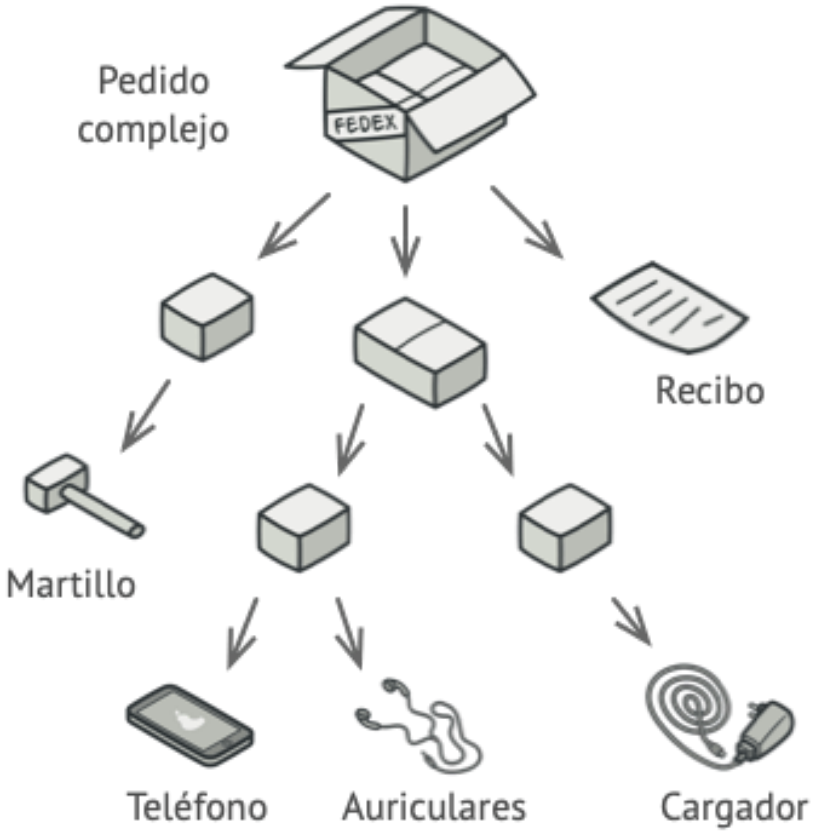
Propósito

Composite es un patrón de diseño estructural que permite **componer objetos en estructuras de árbol** y tratarlas como si fueran **un único objeto**.

- El cliente trabaja contra una **interfaz común**.
- Un **compuesto** delega operaciones a sus hijos de forma **recursiva**.



Imágenes extraídas de Refactoring.Guru



Problema

El patrón tiene sentido cuando tu dominio se modela naturalmente como un **árbol** (jerarquías, contenedores, grupos, etc.).

Ejemplo típico: `Producto` y `Caja`.

- Una caja puede contener **productos** y **otras cajas**.
- Calcular el precio total "a mano" obliga a conocer tipos concretos, niveles de anidación, etc.

Imágenes extraídas de Refactoring.Guru

Solución

Define una **interfaz común** (Componente) para hojas y compuestos.

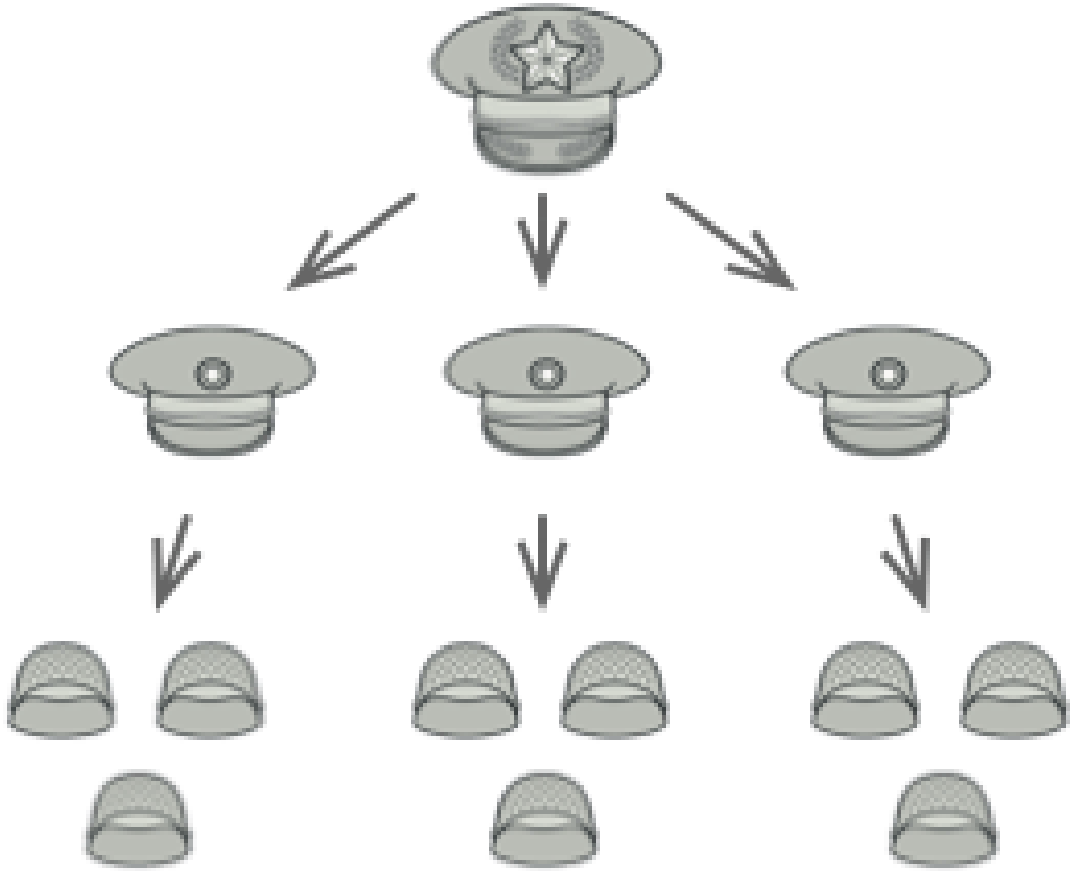
- Una **Hoja** ejecuta la operación (p. ej. `dibujar()`).
- Un **Compuesto** recorre sus hijos, delega la operación y puede “recapitular” el resultado.

Imágenes extraídas de Refactoring.Guru

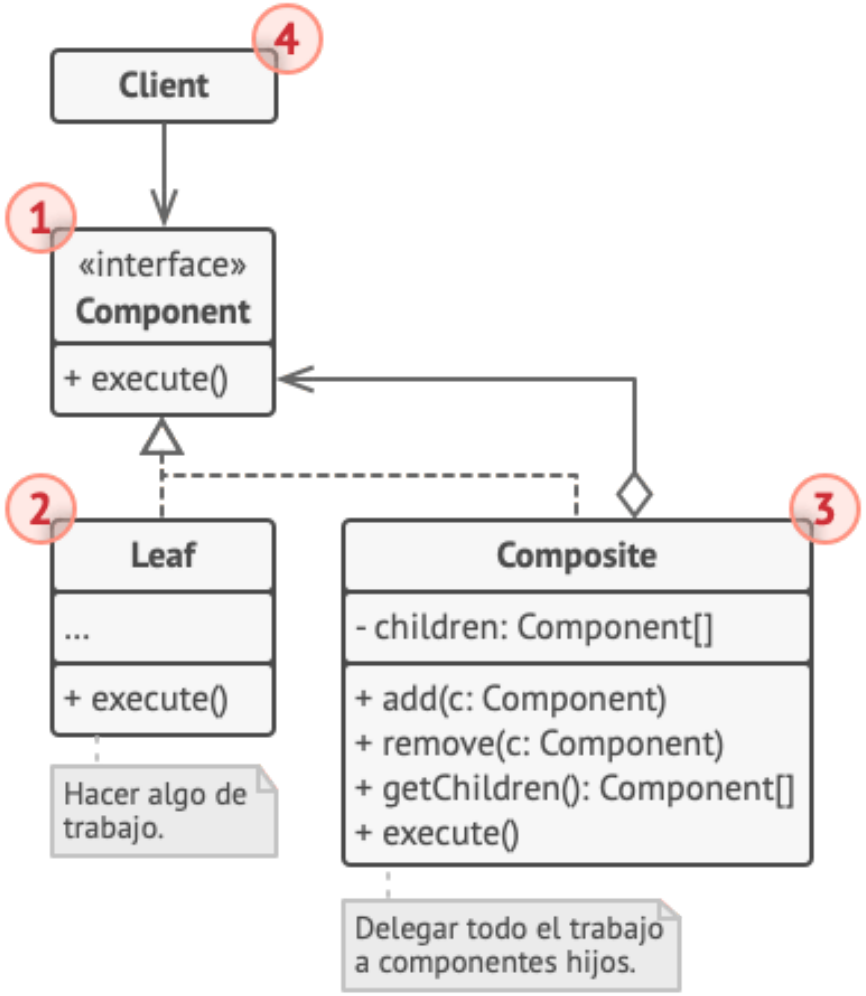
Analogía en el mundo real

Una estructura militar es una jerarquía:

- Ejército → Divisiones → Brigadas → Pelotones → Escuadrones → Soldados
- Una orden se da arriba y se propaga hacia abajo por los niveles.



Imágenes extraídas de Refactoring.Guru

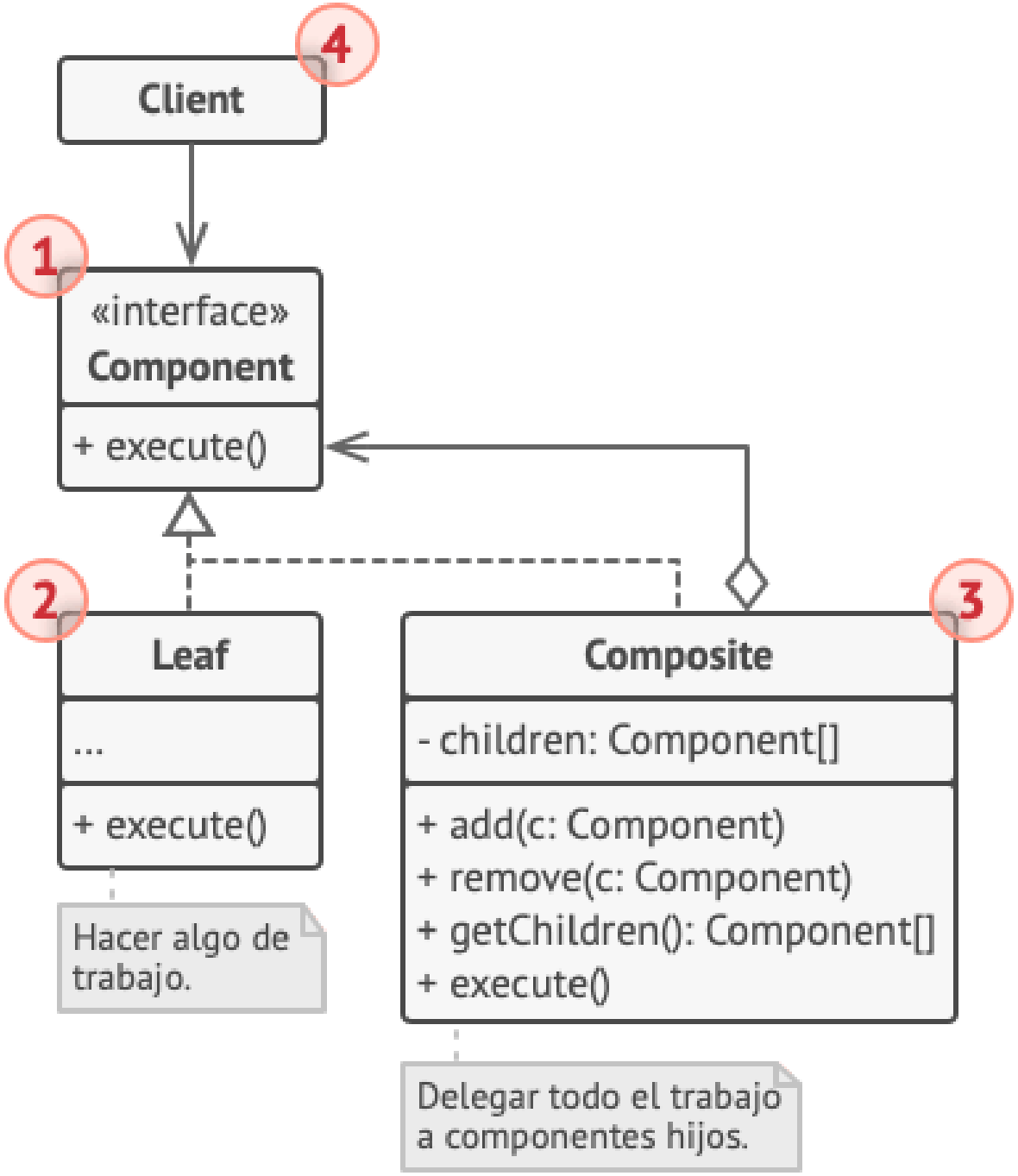


Estructura

Roles típicos:

1. **Componente**: interfaz común para hojas y compuestos.
2. **Hoja**: objeto simple sin hijos.
3. **Compuesto**: contiene hijos (hojas u otros compuestos) y delega.
4. **Cliente**: usa solo la interfaz Componente.

Imágenes extraídas de Refactoring.Guru

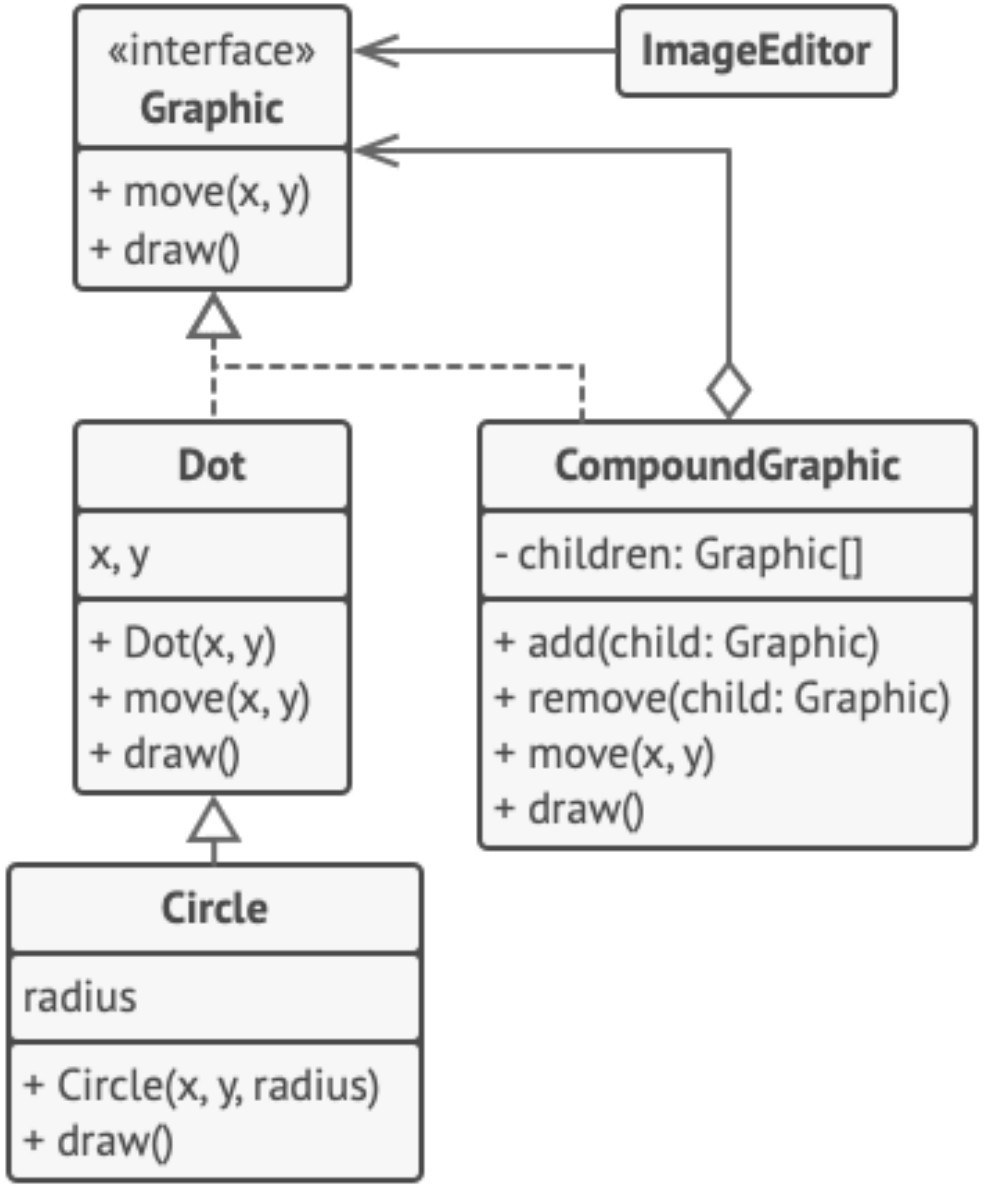


Imágenes extraídas de Refactoring.Guru

Aplicabilidad

Úsalo cuando:

- Necesites representar el dominio como una **estructura en árbol**.
- Quieras que el cliente trate igual elementos **simples** y **compuestos**.
- Quieras **añadir nuevos tipos** (hojas/compuestos) sin tocar el cliente (abierto/cerrado).



Imágenes extraídas de Refactoring.Guru

Pros y contras

Pros

- Facilita trabajar con árboles: **polimorfismo + recursión**.
- **Abierto/cerrado**: añadir nuevos elementos sin romper al cliente.

Contras

- Puede ser difícil definir una interfaz común si las clases difieren mucho.
- A veces fuerza una interfaz demasiado general y menos legible.

Código Java (puntos clave)

En este ejemplo, hojas y compuestos comparten la interfaz `Grafico`.

```
public interface Grafico {
    void mover(int dx, int dy);
    void dibujar();
}
```

Código Java (delegación recursiva)

El compuesto mantiene hijos `Grafico` y delega operaciones.

```
public class CompuestoGrafico implements Grafico {
    private final List<Grafico> hijos = new ArrayList<>();

    public void agregar(Grafico grafico) {
        hijos.add(grafico);
    }

    @Override
    public void mover(int dx, int dy) {
        for (Grafico hijo : hijos) {
            hijo.mover(dx, dy);
        }
    }

    @Override
    public void dibujar() {
        for (Grafico hijo : hijos) {
            hijo.dibujar();
        }
    }
}
```

Código Java (cliente)

El cliente (`EditorDeImagen`) no distingue hoja/compuesto: usa `Grafico` .

```
EditorDeImagen editor = new EditorDeImagen();
editor.cargar();
editor.dibujar();

editor.agruparSeleccion();  // crea un CompuestoGrafico
editor.moverTodo(5, -10);   // el grupo mueve a sus hijos
editor.dibujar();
```