Composite Pattern

Tomás Felipe Melli

June 10, 2025

Índice

1 Intent **2**

2 Motivation **2**

3 Applicability **2**

4 Structure **2**

5 Participants **3**

6 Example **3**

1 Intent

El **Composite** permite componer objetos en estructuras de árbol para representar jerarquías. Permite tratar objetos individuales y composiciones de objetos de manera uniforme.

2 Motivation

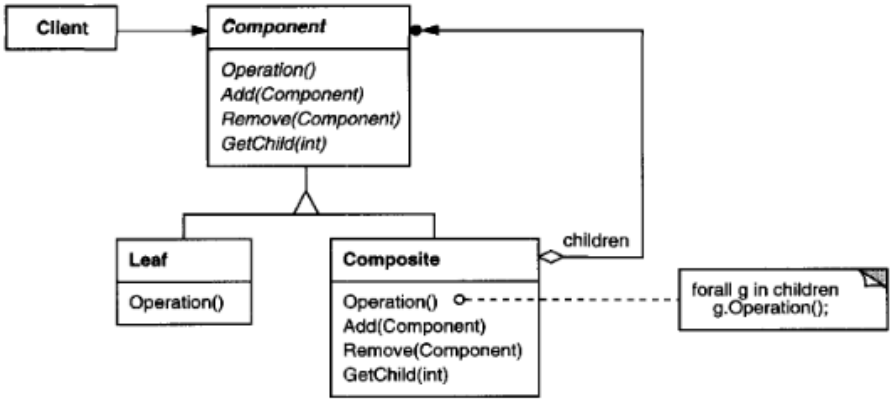
En aplicaciones que manipulan estructuras jerárquicas (como gráficos, archivos, GUI), se necesita poder tratar elementos simples (hojas) y complejos (compuestos) de forma similar. Por ejemplo, un dibujo puede estar compuesto de líneas, rectángulos o incluso de otros dibujos. El patrón Composite permite que el cliente no tenga que distinguir si está trabajando con un objeto simple o compuesto.

3 Applicability

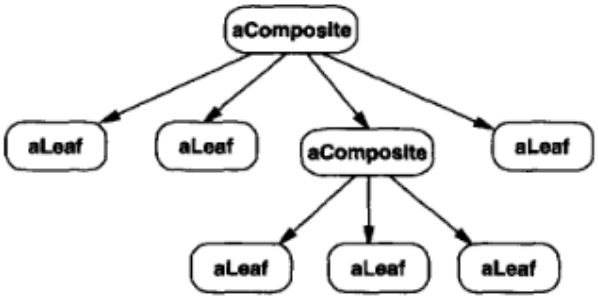
Usar Composite cuando:

- Se quiere representar jerarquías de objetos.
- Se quiere tratar objetos individuales y grupos de manera uniforme.
- Se desea que clientes ignoren diferencias entre composiciones y objetos individuales.

4 Structure



A typical Composite object structure might look like this:



5 Participants

- **Component**: Declara la interfaz para todos los objetos de la composición e implementa el comportamiento por default de la interfaz común a todas las clases.
- **Leaf**: Representa objetos simples (sin hijos). Implementa el comportamiento base.
- **Composite**: Representa objetos compuestos (con hijos). Almacena y administra los hijos. Implementa las operaciones de ‘Component‘.
- **Client**: Usa la interfaz ‘Component‘ para interactuar con objetos simples y compuestos.

6 Example

Component

Queremos construir un editor gráfico que permita manipular distintos elementos visuales como líneas, rectángulos o textos. Algunos de estos elementos pueden contener otros elementos (por ejemplo, un grupo de objetos). El desafío es poder tratar todos estos elementos —ya sean simples o compuestos— de la misma forma en el código.

La solución es usar el patrón **Composite** para que todos los objetos compartan una interfaz común. Así, el cliente puede trabajar con una línea individual o con un grupo de líneas sin preocuparse por la diferencia.

```
class Graphic {
public:
    virtual void Draw() = 0;
    virtual void Add(Graphic* g) {}
    virtual void Remove(Graphic* g) {}
    virtual Graphic* GetChild(int index) { return nullptr; }
    virtual ~Graphic() {}
};
```

Leaf

```
class Line : public Graphic {
public:
    void Draw() override {
        // Dibuja una l nea
    }
};
```

Composite

```
#include <vector>

class Picture : public Graphic {
private:
    std::vector<Graphic*> children;

public:
    void Draw() override {
        for (Graphic* g : children) {
            g->Draw();
        }
    }

    void Add(Graphic* g) override {
        children.push_back(g);
    }

    void Remove(Graphic* g) override {
        // L gica para eliminar g de children
    }

    Graphic* GetChild(int index) override {
        return children.at(index);
    }
};
```

Resultado

El cliente puede trabajar con cualquier objeto que implemente la interfaz ‘Graphic‘, ya sea una ‘Line‘, ‘Rectangle‘ o una composición como ‘Picture‘.

```
Picture* drawing = new Picture();
drawing->Add(new Line());
drawing->Add(new Picture()); // otra subcomposici n

drawing->Draw(); // Dibuja todo, sin importar qu  contiene
```