refactoring.guru

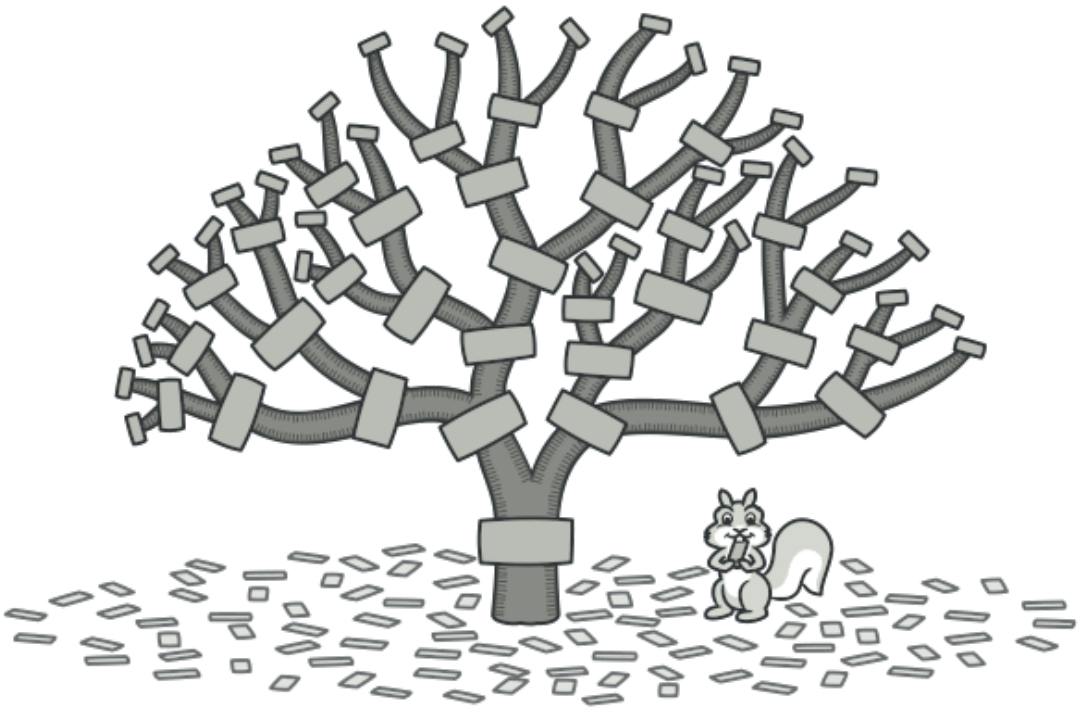
Composite

12–16 minutes

Também conhecido como: Árvore de objetos, Object tree

Propósito

O **Composite** é um padrão de projeto estrutural que permite que você componha objetos em estruturas de árvores e então trabalhe com essas estruturas como se elas fossem objetos individuais.



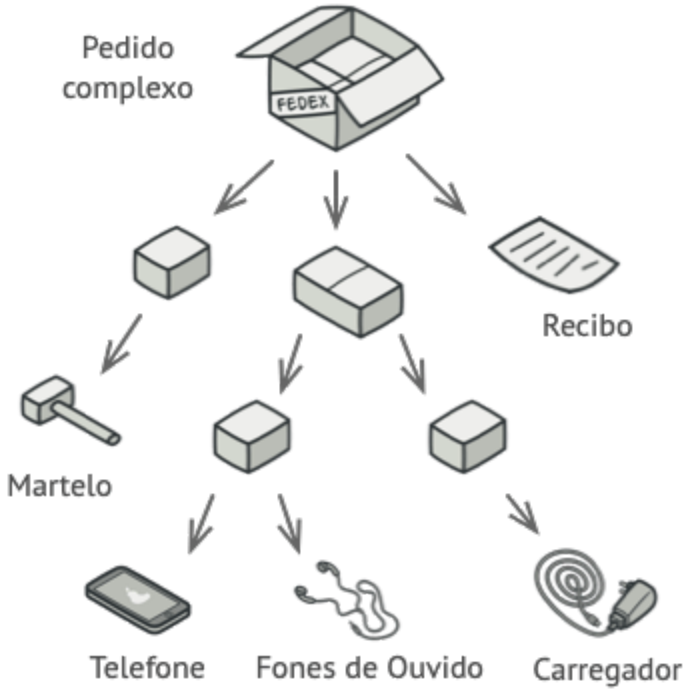
Problema

Usar o padrão Composite faz sentido apenas quando o modelo

central de sua aplicação pode ser representada como uma árvore.

Por exemplo, imagine que você tem dois tipos de objetos: `Produtos` e `Caixas`. Uma `Caixa` pode conter diversos `Produtos` bem como um número de `Caixas` menores. Essas `Caixas` menores também podem ter alguns `Produtos` ou até mesmo `Caixas` menores que elas, e assim em diante.

Digamos que você decida criar um sistema de pedidos que usa essas classes. Os pedidos podem conter produtos simples sem qualquer compartimento, bem como caixas recheadas com produtos... e outras caixas. Como você faria para determinar o preço total desse pedido?



Um pedido pode envolver vários produtos, embalados em caixas, que são embalados em caixas maiores e assim em diante. Toda a estrutura se parece com uma árvore de cabeça para baixo.

Você pode tentar uma solução direta: desempacotar todas as caixas, verificar cada produto e então calcular o total. Isso pode

ser viável no mundo real; mas em um programa, não é tão simples como executar uma iteração. Você tem que conhecer as classes dos `Produtos` e `Caixas` que você está examinando, o nível de aninhamento das caixas e outros detalhes cabeludos de antemão. Tudo isso torna uma solução direta muito confusa ou até impossível.

Solução

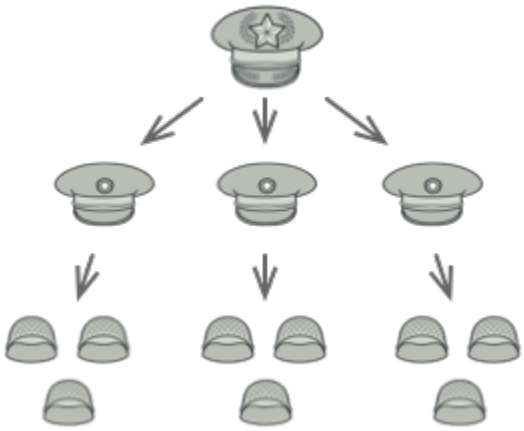
O padrão Composite sugere que você trabalhe com `Produtos` e `Caixas` através de uma interface comum que declara um método para a contagem do preço total.

Como esse método funcionaria? Para um produto, ele simplesmente retornaria o preço dele. Para uma caixa, ele teria que ver cada item que ela contém, perguntar seu preço e então retornar o total para essa caixa. Se um desses itens for uma caixa menor, aquela caixa também deve verificar seu conteúdo e assim em diante, até que o preço de todos os componentes internos sejam calculados. Uma caixa pode até adicionar um custo extra para o preço final, como um preço de embalagem.

O padrão Composite permite que você rode um comportamento recursivamente sobre todos os componentes de uma árvore de objetos.

O maior benefício dessa abordagem é que você não precisa se preocupar sobre as classes concretas dos objetos que compõem essa árvore. Você não precisa saber se um objeto é um produto simples ou uma caixa sofisticada. Você pode tratar todos eles com a mesma interface. Quando você chama um método os próprios objetos passam o pedido pela árvore.

Analogia com o mundo real



Um exemplo de uma estrutura militar.

Exércitos da maioria dos países estão estruturados como hierarquias. Um exército consiste de diversas divisões; uma divisão é um conjunto de brigadas, e uma brigada consiste de pelotões, que podem ser divididos em esquadrões. Finalmente, um esquadrão é um pequeno grupo de soldados reais. Ordens são dadas do topo da hierarquia e são passadas abaixo para cada nível até cada soldado saber o que precisa ser feito.

Estrutura

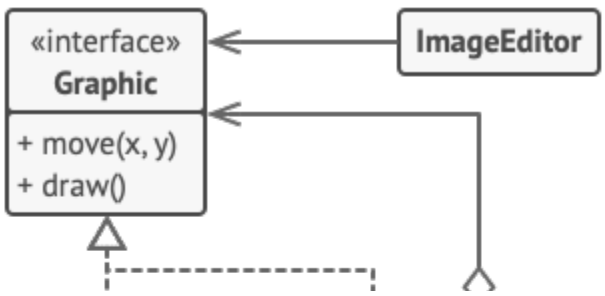
1. A interface **Componente** descreve operações que são comuns tanto para elementos simples como para elementos complexos da árvore.
2. A **Folha** é um elemento básico de uma árvore que não tem sub-elementos.

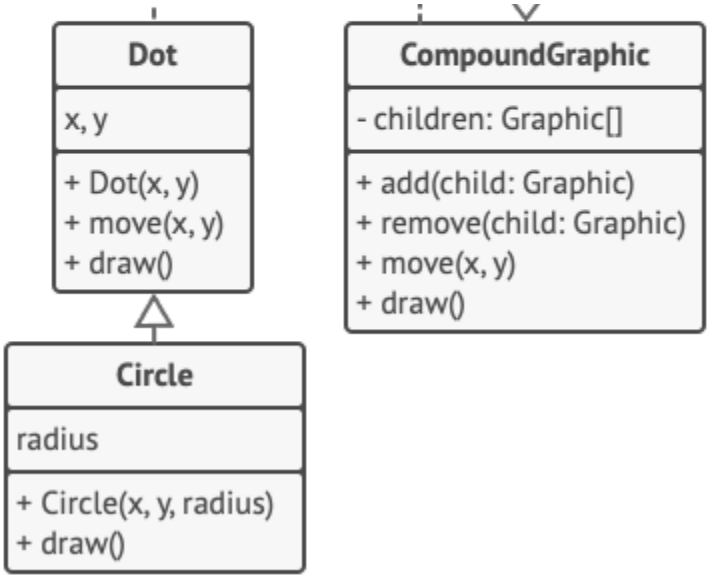
   Geralmente, componentes folha acabam fazendo boa parte do verdadeiro trabalho, uma vez que não tem mais ninguém para delegá-lo.
3. O **Contêiner** (ou *composite*) é o elemento que tem sub-elementos: folhas ou outros contêineres. Um contêiner não sabe a classe concreta de seus filhos. Ele trabalha com todos os sub-elementos apenas através da interface componente.

   Ao receber um pedido, um contêiner delega o trabalho para seus sub-elementos, processa os resultados intermediários, e então retorna o resultado final para o cliente.
4. O **Cliente** trabalha com todos os elementos através da interface componente. Como resultado, o cliente pode trabalhar da mesma forma tanto com elementos simples como elementos complexos da árvore.

Pseudocódigo

Nesse exemplo, o padrão **Composite** deixa que você implemente pilhas de formas geométricas em um editor gráfico.





Exemplo do editor de formas geométricas.

A classe `GráficoComposto` é um contêiner que pode ter qualquer número de sub-formas, incluindo outras formas compostas. Uma forma composta tem os mesmos métodos que uma forma simples. Contudo, ao invés de fazer algo próprio, uma forma composta passa o pedido recursivamente para todas as suas filhas e “soma” o resultado.

O código cliente trabalha com todas as formas através da interface única comum à todas as classes de forma. Portanto, o cliente não sabe se está trabalhando com uma forma simples ou composta. O cliente pode trabalhar com estruturas de objeto muito complexas sem ficar acoplado à classe concreta que formou aquela estrutura.

```
// A interface componente declara operações comuns para ambos
os
// objetos simples e complexos de uma composição.
interface Graphic is
    method move(x, y)
    method draw()
```

```
// A classe folha representa objetos finais de uma composição.
// Um objeto folha não pode ter quaisquer sub-objetos.
// Geralmente, são os objetos folha que fazem o verdadeiro
// trabalho, enquanto que os objetos composite somente delegam
// para seus sub componentes.
class Dot implements Graphic is
    field x, y

    constructor Dot(x, y) { ... }

    method move(x, y) is
        this.x += x, this.y += y

    method draw() is
        // Desenhar um ponto em X e Y.

// Todas as classes componente estendem outros componentes.
class Circle extends Dot is
    field radius

    constructor Circle(x, y, radius) { ... }

    method draw() is
        // Desenhar um círculo em X e Y com raio R.

// A classe composite representa componentes complexos que
podem
// ter filhos. Objetos composite geralmente delegam o verdadeiro
// trabalho para seus filhos e então "somam" o resultado.
```

```
class CompoundGraphic implements Graphic is
    field children: array of Graphic

    // Um objeto composite pode adicionar ou remover outros
    // componentes (tanto simples como complexos) para ou de sua
    // lista de filhos.
    method add(child: Graphic) is
        // Adiciona um filho para o vetor de filhos.

    method remove(child: Graphic) is
        // Remove um filho do vetor de filhos.

    method move(x, y) is
        foreach (child in children) do
            child.move(x, y)

    // Um composite executa sua lógica primária em uma forma
    // particular. Ele percorre recursivamente através de todos
    // seus filhos, coletando e somando seus resultados. Já que
    // os filhos do composite passam essas chamadas para seus
    // próprios filhos e assim em diante, toda a árvore de
    // objetos é percorrida como resultado.
    method draw() is
        // 1. Para cada componente filho:
        //     - Desenhe o componente.
        //     - Atualize o retângulo limitante.
        // 2. Desenhe um retângulo tracejado usando as
        // limitantes.

// O código cliente trabalha com todos os componentes através de
```

```
// suas interfaces base. Dessa forma o código cliente pode
// suportar componentes folha simples e composites complexos.
class ImageEditor is
    field all: CompoundGraphic

    method load() is
        all = new CompoundGraphic()
        all.add(new Dot(1, 2))
        all.add(new Circle(5, 3, 10))



    // Combina componentes selecionados em um componente
    // composite complexo.
    method groupSelected(components: array of Graphic) is
        group = new CompoundGraphic()
        foreach (component in components) do
            group.add(component)
            all.remove(component)
        all.add(group)
        // Todos os componentes serão desenhados.
        all.draw()
```

Aplicabilidade

Utilize o padrão Composite quando você tem que implementar uma estrutura de objetos tipo árvore.

O padrão Composite fornece a você com dois tipos básicos de elementos que compartilham uma interface comum: folhas simples e contêineres complexos. Um contêiner pode ser composto tanto de folhas como por outros contêineres. Isso

permite a você construir uma estrutura de objetos recursiva aninhada que se assemelha a uma árvore.

Utilize o padrão quando você quer que o código cliente trate tanto os objetos simples como os compostos de forma uniforme.

Todos os elementos definidos pelo padrão Composite compartilham uma interface comum. Usando essa interface o cliente não precisa se preocupar com a classe concreta dos objetos com os quais está trabalhando.

Como implementar

1. Certifique-se que o modelo central de sua aplicação possa ser representada como uma estrutura de árvore. Tente quebrá-lo em elementos simples e contêineres. Lembre-se que contêineres devem ser capazes de conter tanto elementos simples como outros contêineres.
2. Declare a interface componente com uma lista de métodos que façam sentido para componentes complexos e simples.
3. Crie uma classe folha que represente elementos simples. Um programa pode ter múltiplas classes folha diferentes.
4. Crie uma classe contêiner para representar elementos complexos. Nessa classe crie um vetor para armazenar referências aos sub-elementos. O vetor deve ser capaz de armazenar tanto folhas como contêineres, então certifique-se que ele foi declarado com um tipo de interface componente.

   Quando implementar os métodos para a interface componente, lembre-se que um contêiner deve ser capaz de delegar a maior parte do trabalho para os sub-elementos.

5. Por fim, defina os métodos para adicionar e remover os elementos filhos no contêiner.

   Tenha em mente que essas operações podem ser declaradas dentro da interface componente. Isso violaria o *princípio de segregação de interface* porque os métodos estarão vazios na classe folha. Contudo, o cliente será capaz de tratar de todos os elementos de forma igual, mesmo ao montar a árvore.

Prós e contras

- Você pode trabalhar com estruturas de árvore complexas mais convenientemente: utilize o polimorfismo e a recursão a seu favor.
- *Princípio aberto/fechado*. Você pode introduzir novos tipos de elemento na aplicação sem quebrar o código existente, o que agora funciona com a árvore de objetos.
- Pode ser difícil providenciar uma interface comum para classes cuja funcionalidade difere muito. Em certos cenários, você precisaria generalizar muito a interface componente, fazendo dela uma interface de difícil compreensão.

Relações com outros padrões

- Você pode usar o Builder quando criar árvores Composite complexas porque você pode programar suas etapas de construção para trabalhar recursivamente.
- O Chain of Responsibility é frequentemente usado em conjunto com o Composite. Neste caso, quando um componente folha recebe um pedido, ele pode passá-lo através de uma corrente de todos os componentes pai até a raiz do objeto árvore.

- Você pode usar Iteradores para percorrer árvores Composite.
- Você pode usar o Visitor para executar uma operação sobre uma árvore Composite inteira.
- Você pode implementar nós folha compartilhados da árvore Composite como Flyweights para salvar RAM.
- O Composite e o Decorator tem diagramas estruturais parecidos já que ambos dependem de composição recursiva para organizar um número indefinido de objetos.

  Um *Decorador* é como um *Composite* mas tem apenas um componente filho. Há outra diferença significativa: o *Decorador* adiciona responsabilidades adicionais ao objeto envolvido, enquanto que o *Composite* apenas "soma" o resultado de seus filhos.

  Contudo, os padrões também podem cooperar: você pode usar o *Decorador* para estender o comportamento de um objeto específico na árvore Composite
- Projetos que fazem um uso pesado de Composite e do Decorator podem se beneficiar com frequência do uso do Prototype. Aplicando o padrão permite que você clone estruturas complexas ao invés de reconstruí-las do zero.