refactoring.guru

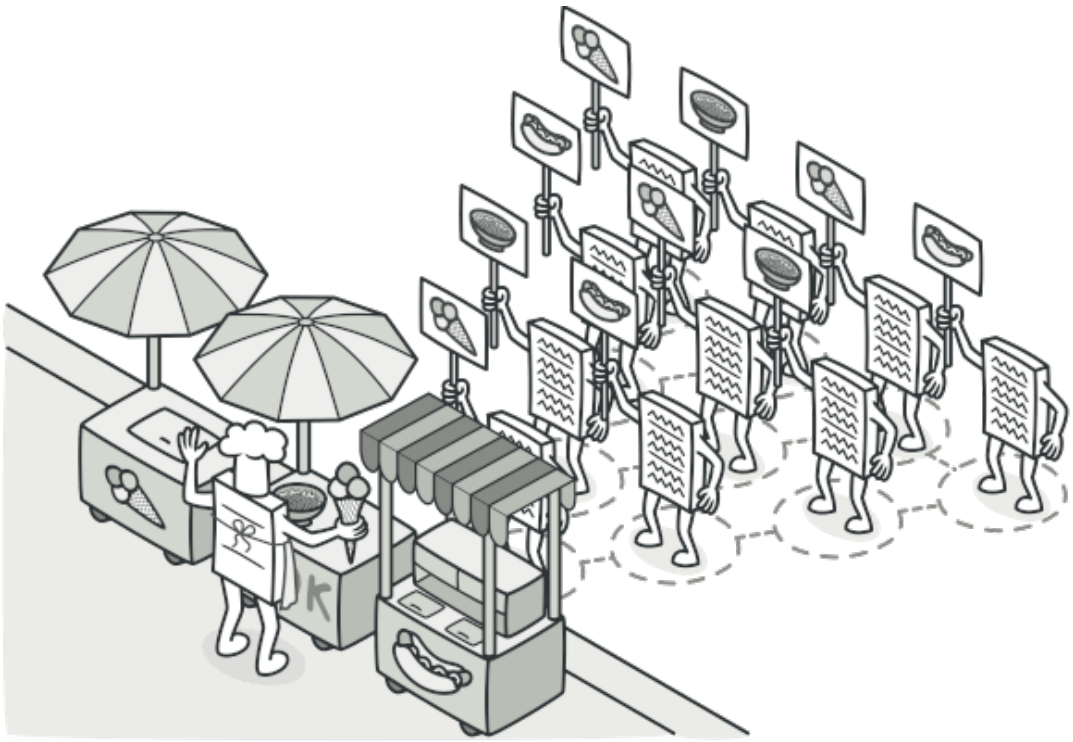
Visitor

15–20 minutes

Também conhecido como: Visitante

Propósito

O **Visitor** é um padrão de projeto comportamental que permite que você separe algoritmos dos objetos nos quais eles operam.



Problema

Imagine que sua equipe desenvolve uma aplicação que funciona com informações geográficas estruturadas em um grafo colossal.

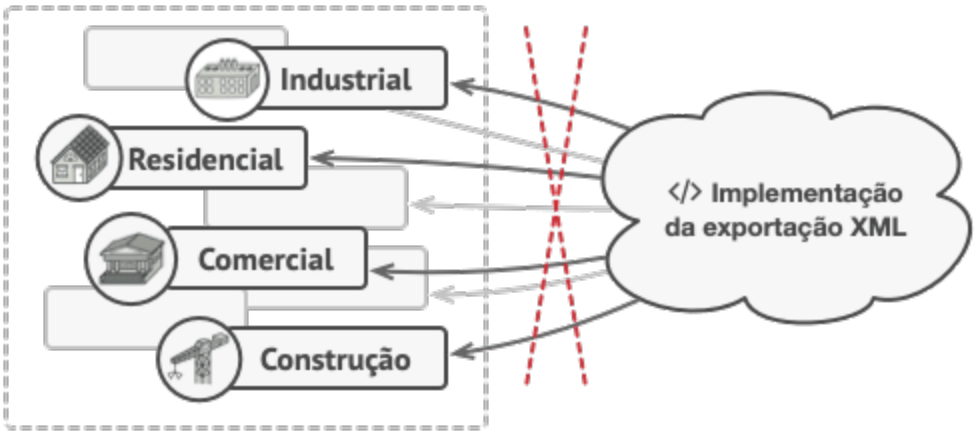
Cada vértice do gráfico pode representar uma entidade complexa como uma cidade, mas também coisas mais granulares como indústrias, lugares turísticos, etc. Os vértices estão conectados entre si se há uma estrada entre os objetos reais que eles representam. Por debaixo dos panos, cada tipo de vértice é representado por sua própria classe, enquanto que cada vértice específico é um objeto.

Exportando o grafo para XML.

Em algum momento você tem uma tarefa de implementar a exportação do grafo para o formato XML. No começo, o trabalho parecia muito simples. Você planejou adicionar um método de exportação para cada classe nó e então uma alavancagem recursiva para ir a cada nó do grafo, executando o método de exportação. A solução foi simples e elegante: graças ao polimorfismo, você não estava acoplando o código que chamava o método de exportação com as classes concretas dos nós.

Infelizmente, o arquiteto do sistema se recusou a permitir que você alterasse as classes nó existentes. Ele disse que o código já estava em produção e ele não queria arriscar quebrá-lo por causa de um possível bug devido às suas mudanças.

Classes existentes da aplicação



O método de exportação XML teve que ser adicionado a todas as classes nodo, o que trouxe o risco de quebrar toda a aplicação se quaisquer bugs passarem junto com a mudança.

Além disso, ele questionou se faria sentido ter um código de exportação XML dentro das classes nó. O trabalho primário dessas classes era trabalhar com dados geográficos. O comportamento de exportação XML ficaria estranho ali.

Houve outra razão para a recusa. Era bem provável que após essa funcionalidade ser implementada, alguém do departamento de marketing pediria que você fornecesse a habilidade para exportar para um formato diferente, ou pediria alguma outra coisa estranha. Isso forçaria você a mudar aquelas frágeis e preciosas classes novamente.

Solução

O padrão Visitor sugere que você coloque o novo comportamento em uma classe separada chamada *visitante*, ao invés de tentar integrá-lo em classes já existentes. O objeto original que teve que fazer o comportamento é agora passado para um dos métodos da visitante como um argumento, desde que o método acesse todos os dados necessários contidos dentro do objeto.

Agora, e se o comportamento puder ser executado sobre objetos de classes diferentes? Por exemplo, em nosso caso com a exportação XML, a verdadeira implementação vai provavelmente ser um pouco diferente nas variadas classes nó. Portanto, a classe visitante deve definir não um, mas um conjunto de métodos, cada um capaz de receber argumentos de diferentes tipos, como este:

```
class ExportVisitor implements Visitor is
    method doForCity(City c) { ... }
    method doForIndustry(Industry f) { ... }
    method doForSightSeeing(SightSeeing ss) { ... }
```

Mas como exatamente nós chamaríamos esses métodos, especialmente quando lidando com o grafo inteiro? Esses métodos têm diferentes assinaturas, então não podemos usar o polimorfismo. Para escolher um método visitante apropriado que seja capaz de processar um dado objeto, precisaríamos checar a classe dele. Isso não parece um pesadelo?

```
foreach (Node node in graph)
    if (node instanceof City)
        exportVisitor.doForCity((City) node)
    if (node instanceof Industry)
        exportVisitor.doForIndustry((Industry) node)

}
```

Você pode perguntar, por que não usamos o sobrecarregamento de método? Isso é quando você dá a todos os métodos o mesmo nome, mesmo se eles suportam diferentes conjuntos de

parâmetros. Infelizmente, mesmo assumindo que nossa linguagem de programação suporta o sobrecarregamento (como Java e C#), isso não nos ajudaria. Já que a classe exata de um objeto nó é desconhecida de antemão, o mecanismo de sobrecarregamento não será capaz de determinar o método correto para executar. Ele irá usar como padrão o método que usa um objeto da classe Nó base.

Contudo, o padrão Visitor resolve esse problema. Ele usa uma técnica chamada Double Dispatch, que ajuda a executar o método apropriado de um objeto sem precisarmos de condicionais pesadas. Ao invés de deixar o cliente escolher uma versão adequada do método para chamar, que tal delegarmos essa escolha para os objetos que estamos passando para a visitante como argumentos? Já que os objetos sabem suas próprias classes, eles serão capazes de escolher um método adequado na visitante de forma simples. Eles “aceitam” uma visitante e dizem a ela qual método visitante deve ser executado.

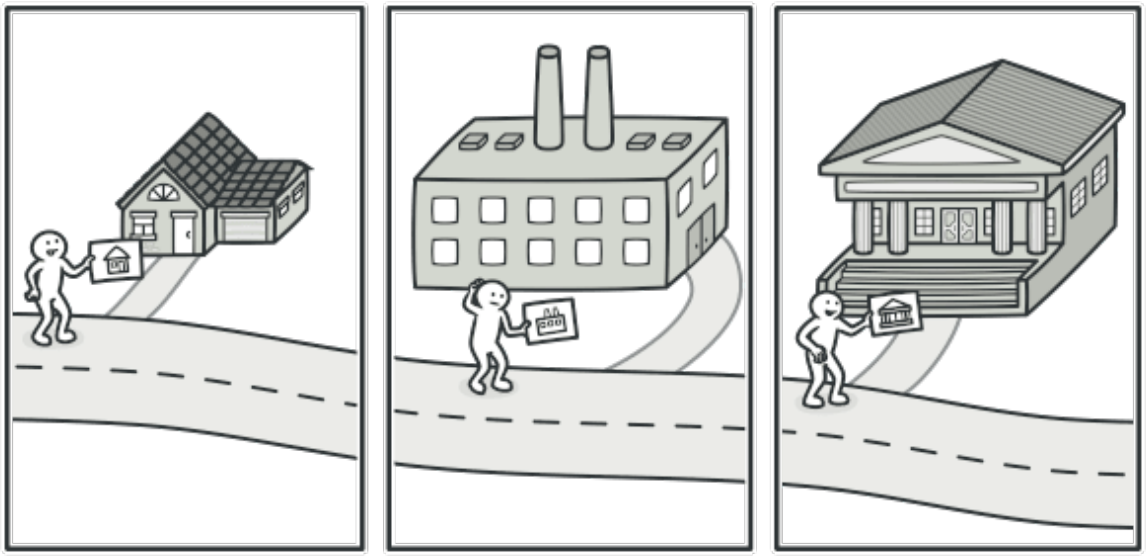
```
// Código cliente
foreach (Node node in graph)
    node.accept(exportVisitor)

// Cidade
class City is
    method accept(Visitor v) is
        v.doForCity(this)


// Indústria
class Industry is
```

```
method accept(Visitor v) is
    v.doForIndustry(this)
```

Eu confesso. Tivemos que mudar as classes nó de qualquer jeito. Mas ao menos a mudança foi trivial e ela permite que nós adicionemos novos comportamentos sem alterar o código novamente.

Agora, se extrairmos uma interface comum para todas as visitantes, todos os nós existentes podem trabalhar com uma visitante que você introduzir na aplicação. Se você se deparar mais tarde adicionando um novo comportamento relacionado aos nós, tudo que você precisa fazer é implementar uma nova classe visitante.

Analogia com o mundo real



Um bom agente de seguros está sempre pronto para oferecer diferentes apólices para vários tipos de organizações.

Imagine um agente de seguros experiente que está ansioso para obter novos clientes. Ele pode visitar cada prédio de uma

vizinhança, tentando vender apólices para todos que encontra. Dependendo do tipo de organização que ocupa o prédio, ele pode oferecer apólices de seguro especializadas:

- Se for um prédio residencial, ele vende seguros médicos.
- Se for um banco, ele vende seguro contra roubo.
- Se for uma cafeteria, ele vende seguro contra incêndios e enchentes.

Estrutura

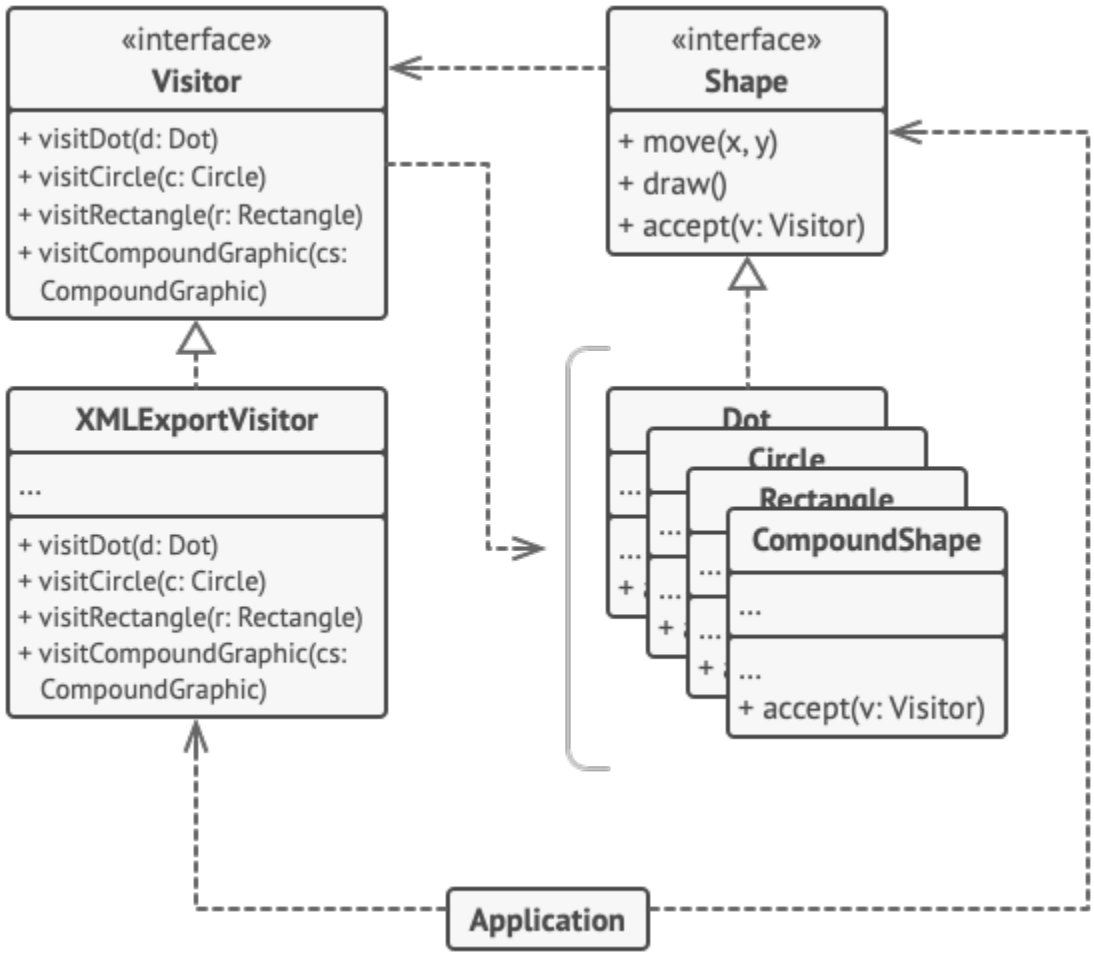
1. A interface **Visitante** declara um conjunto de métodos visitantes que podem receber elementos concretos de uma estrutura de objetos como argumentos. Esses métodos podem ter os mesmos nomes se o programa é escrito em uma linguagem que suporta sobrecarregamento, mas o tipo dos parâmetros devem ser diferentes.
2. Cada **Visitante Concreto** implementa diversas versões do mesmo comportamento, feitos sob medida para diferentes elementos concretos de classes.
3. A interface **Elemento** declara um método para “aceitar” visitantes. Esse método deve ter um parâmetro declarado com o tipo da interface do visitante.
4. Cada **Elemento Concreto** deve implementar o método de aceitação. O propósito desse método é redirecionar a chamada para o método visitante apropriado que corresponde com a atual classe elemento. Esteja atento que mesmo se uma classe elemento base implemente esse método, todas as subclasses deve ainda sobrescrever esse método em suas próprias classes e

chamar o método apropriado no objeto visitante.

5. O **Cliente** geralmente representa uma coleção de outros objetos complexos (por exemplo, uma árvore Composite). Geralmente, os clientes não estão cientes de todas as classes elemento concretas porque eles trabalham com objetos daquela coleção através de uma interface abstrata.

Pseudocódigo

Neste exemplo, o padrão **Visitor** adiciona suporte a exportação XML para a hierarquia de classe de formas geométricas.



Exportando vários tipos de objetos para o formato XML atráves do objeto visitante.

```
// O elemento interface declara um método `accept` que toma a
// interface do visitante base como um argumento.
interface Shape is
    method move(x, y)
    method draw()
    method accept(v: Visitor)

// Cada classe concreta de elemento deve implementar o método
// `accept` de tal maneira que ele chama o método visitante que
// corresponde com a classe do elemento.
class Dot implements Shape is


    // Observe que nós estamos chamando `visitDot`, que coincide
    // com o nome da classe atual. Dessa forma nós permitimos
    // que o visitante saiba a classe do elemento com o qual ele
    // trabalha.
    method accept(v: Visitor) is
        v.visitDot(this)

class Circle implements Shape is

    method accept(v: Visitor) is
        v.visitCircle(this)

class Rectangle implements Shape is

    method accept(v: Visitor) is
        v.visitRectangle(this)
```

```
class CompoundShape implements Shape is

    method accept(v: Visitor) is
        v.visitCompoundShape(this)


// A interface visitante declara um conjunto de métodos
// visitantes que correspondem com as classes elemento. A
// assinatura de um método visitante permite que o visitante
// identifique a classe exata do elemento com o qual ele está
// lidando.
interface Visitor is
    method visitDot(d: Dot)
    method visitCircle(c: Circle)
    method visitRectangle(r: Rectangle)
    method visitCompoundShape(cs: CompoundShape)

// Visitantes concretos implementam várias versões do mesmo
// algoritmo, que pode trabalhar com todas as classes elemento
// concretas.
//
// Você pode usufruir do maior benefício do padrão Visitor
// quando estiver usando ele com uma estrutura de objeto
// complexa, tal como uma árvore composite. Neste caso, pode
ser
// útil armazenar algum estado intermediário do algoritmo
// enquanto executa os métodos visitantes sobre vários objetos
// da estrutura.
class XMLExportVisitor implements Visitor is
    method visitDot(d: Dot) is
```

```
        // Exporta a ID do dot (ponto) e suas coordenadas de
        // centro.

    method visitCircle(c: Circle) is
        // Exporta a ID do circle (círculo), coordenadas do
        // centro, e raio.


    method visitRectangle(r: Rectangle) is
        // Exporta a ID do retângulo, coordenadas do topo à
        // esquerda, largura e altura.

    method visitCompoundShape(cs: CompoundShape) is
        // Exporta a ID da forma bem como a lista de ID dos seus
        // filhos.


// O código cliente pode executar operações visitantes sobre
// quaisquer conjuntos de elementos sem saber suas classes
// concretas. A operação accept (aceitar) direciona a chamada
// para a operação apropriada no objeto visitante.
class Application is
    field allShapes: array of Shapes

    method export() is
        exportVisitor = new XMLExportVisitor()

        foreach (shape in allShapes) do
            shape.accept(exportVisitor)
```

Se você está se perguntando por que precisamos do método `aceitar` neste exemplo, meu artigo Visitor e Double Dispatch responde essa dúvida em detalhes.

Aplicabilidade

Utilize o Visitor quando você precisa fazer uma operação em todos os elementos de uma estrutura de objetos complexa (por exemplo, uma árvore de objetos).

O padrão Visitor permite que você execute uma operação sobre um conjunto de objetos com diferentes classes ao ter o objeto visitante implementando diversas variantes da mesma operação, que correspondem a todas as classes alvo.

Utilize o Visitor para limpar a lógica de negócio de comportamentos auxiliares.

O padrão permite que você torne classes primárias de sua aplicação mais focadas em seu trabalho principal ao extrair todos os comportamentos em um conjunto de classes visitantes.

Utilize o padrão quando um comportamento faz sentido apenas dentro de algumas classes de uma uma hierarquia de classe, mas não em outras.

Você pode extrair esse comportamento para uma classe visitante separada e implementar somente aqueles métodos visitantes que aceitam objetos de classes relevantes, deixando o resto vazio.

Como implementar

1. Declare a interface da visitante com um conjunto de métodos “visitando”, um para cada classe elemento concreta que existe no

programa.

2. Declare a interface elemento. Se você está trabalhando com uma hierarquia de classes elemento existente, adicione o método de “aceitação” para a classe base da hierarquia. Esse método deve aceitar um objeto visitante como um argumento.
3. Implemente os métodos de aceitação em todas as classes elemento concretas. Esses métodos devem simplesmente redirecionar a chamada para um método visitante no objeto visitante que está vindo e que coincide com a classe do elemento atual.
4. As classes elemento devem trabalhar apenas com visitantes através da interface do visitante. Os visitantes, contudo, devem estar cientes de todas as classes elemento concretas referenciadas como tipos de parâmetros dos métodos visitantes.
5. Para cada comportamento que não possa ser implementado dentro da hierarquia do elemento, crie uma nova classe visitante concreta e implemente todos os métodos visitantes.

   Você pode encontrar uma situação onde o visitante irá necessitar acesso para alguns membros privados da classe elemento. Neste caso, você pode ou fazer desses campos ou métodos públicos, violando o encapsulamento do elemento, ou aninhando a classe visitante na classe elemento. Está última só é possível se você tiver sorte e estiver trabalhando com uma linguagem de programação que suporta classes aninhadas.
6. O cliente deve criar objetos visitantes e passá-los para os elementos através dos métodos de “aceitação”.

Prós e contras

- *Princípio aberto/fechado*. Você pode introduzir um novo comportamento que pode funcionar com objetos de diferentes classes sem mudar essas classes.
- *Princípio de responsabilidade única*. Você pode mover múltiplas versões do mesmo comportamento para dentro da mesma classe.
- Um objeto visitante pode acumular algumas informações úteis enquanto trabalha com vários objetos. Isso pode ser interessante quando você quer percorrer algum objeto de estrutura complexa, tais como um objeto árvore, e aplicar o visitante para cada objeto da estrutura.
- Você precisa atualizar todos os visitantes a cada vez que a classe é adicionada ou removida da hierarquia de elementos.
- Visitantes podem não ter seu acesso permitido para campos e métodos privados dos elementos que eles deveriam estar trabalhando.

Relações com outros padrões

- Você pode tratar um Visitor como uma poderosa versão do padrão Command. Seus objetos podem executar operações sobre vários objetos de diferentes classes.
- Você pode usar o Visitor para executar uma operação sobre uma árvore Composite inteira.
- Você pode usar o Visitor junto com o Iterator para percorrer uma estrutura de dados complexas e executar alguma operação sobre seus elementos, mesmo se eles todos tenham classes diferentes.

Conteúdo Adicional

- Confuso com o por que de não podermos simplesmente substituir o padrão Visitor com o sobrecarregamento de método? Leia meu artigo Visitor e Double Dispatch para aprender mais sobre os detalhes sórdidos a respeito.