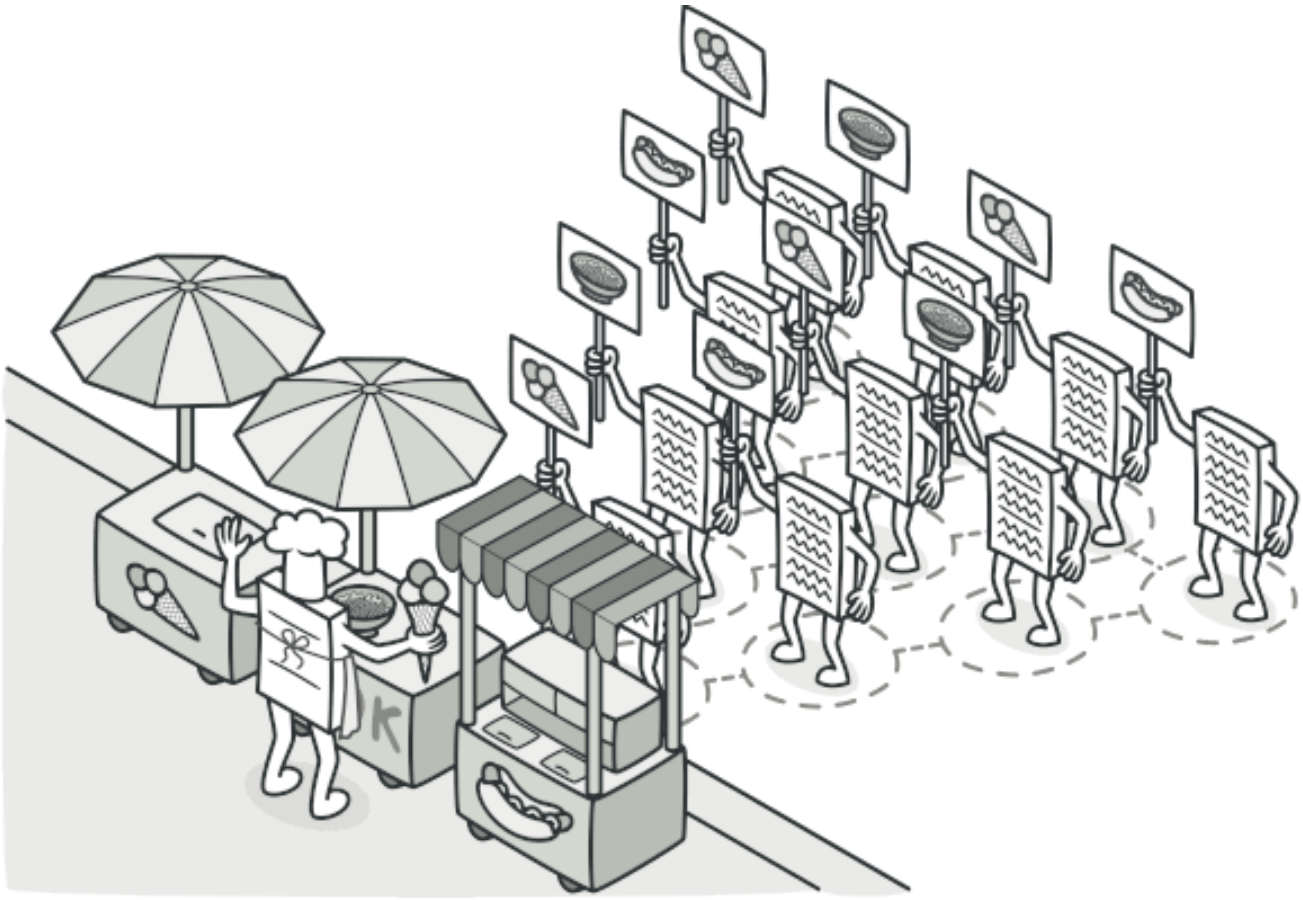
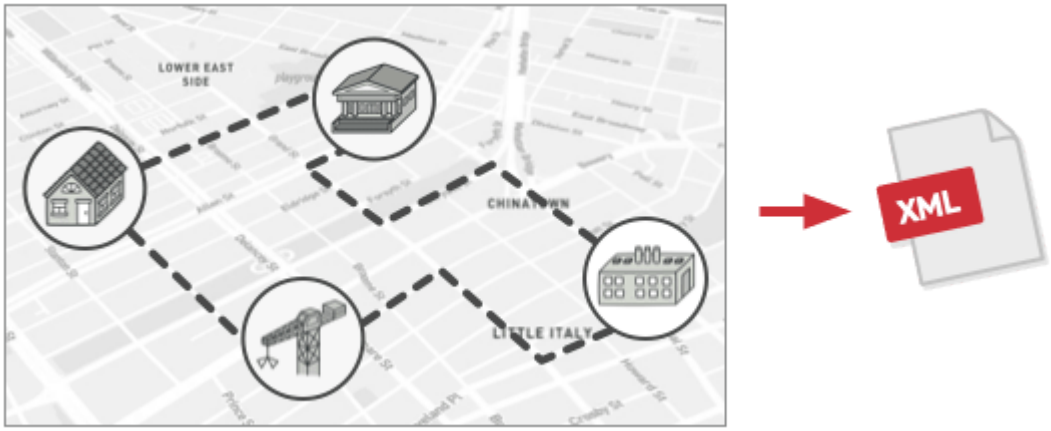
Visitor

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

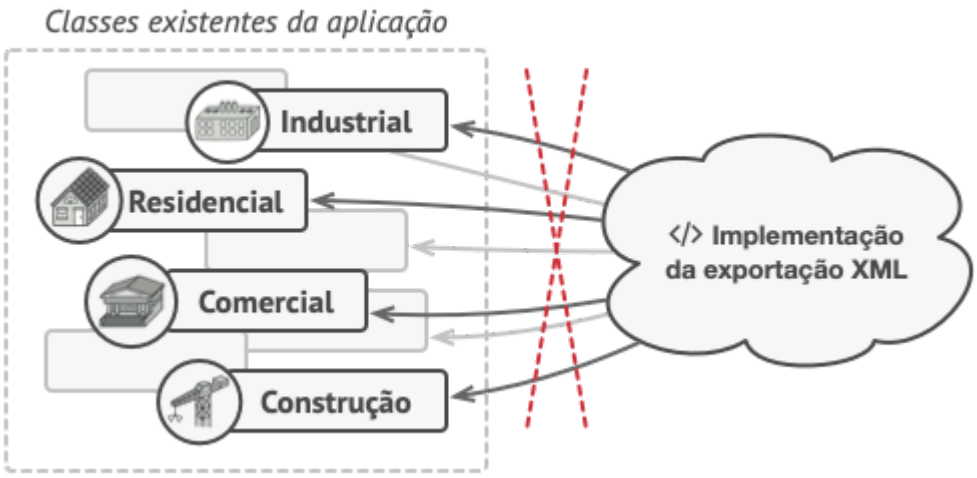


Problema

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.



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.



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 ismethod 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?

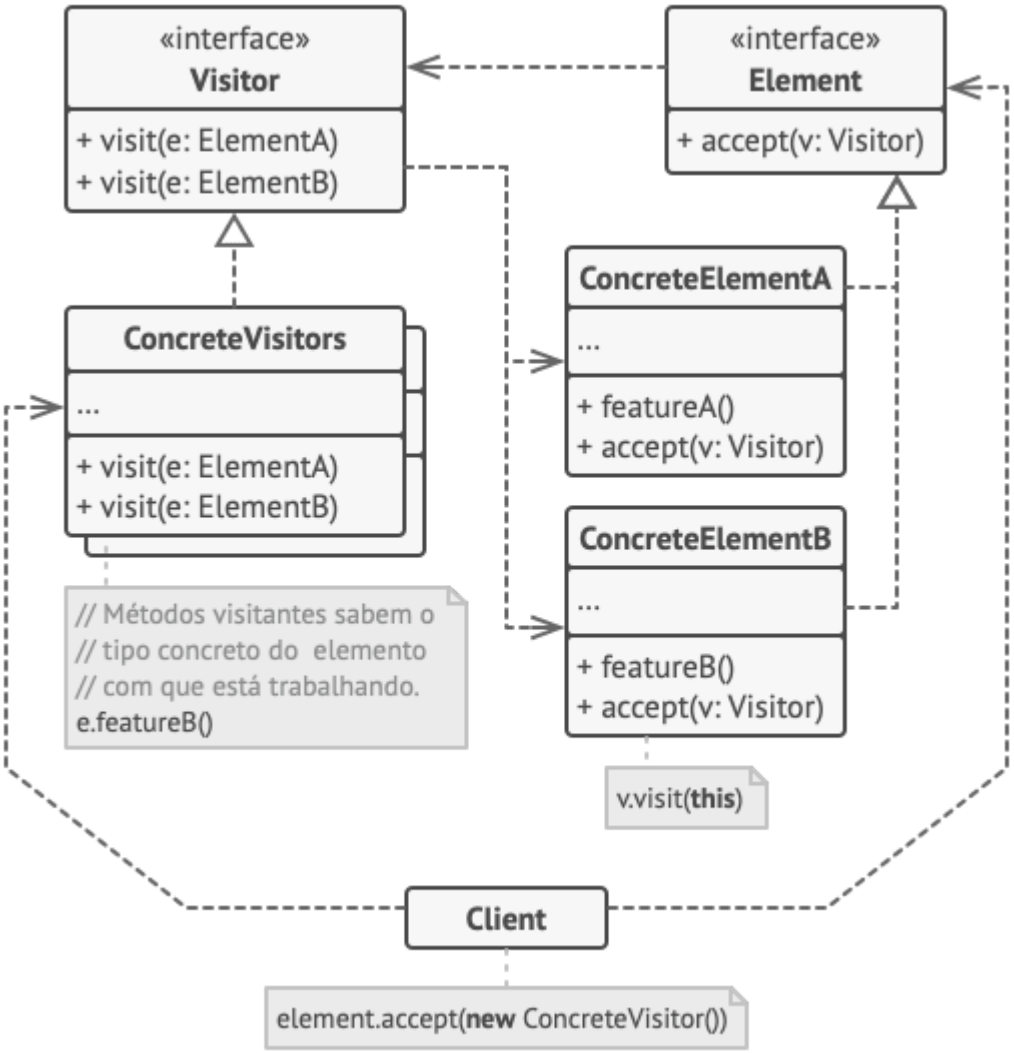
Já que os objetos sabem suas próprias classes, eles serão capazes de escolher um método adequado na visitante de forma simples. Eles “aceitam” uma visitante e dizem a ela qual método visitante deve ser executado.

```
// Código clienteforeach (Node node in graph)node.accept(exportVisitor)// Cidadeclass
 City ismethod accept(Visitor v) isv.doForCity(this)// ...// Indústriaclass Industry
              ismethod accept(Visitor v) isv.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.

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 todos as classes elemento

concretas porque eles trabalham com objetos daquele coleção através de uma interface abstrata.