Visitor

Padrões Comportamentais

O padrão *Visitor* representa uma operação a ser executada nos membros de uma estrutura de objetos. Ele permite definir uma nova operação sem mudar as classes dos membros sobre os quais opera.

Motivação (Por que utilizar?)

O padrão *Visitor* é uma solução para separar algoritmos da estrutura dos objetos. Uma das vantagens desse padrão é a possibilidade de adicionar novas operações a uma estrutura de objetos já existente. Durante o desenvolvimento de *software* é comum encontrar funcionalidades que não são exatamente parte de um objeto ou outro, colocar tais funcionalidades dentro da classe dos próprios objetos poderia poluir o código das classes e abrir brechas para o surgimento de bugs.

Para exemplificar o padrão *Visitor* considere as classes abaixo que fazem parte do sistema de uma rede de supermercados. Temos uma hierarquia de classes:

- Existem vários supermercados;
- Em cada supermercado existem vários departamentos;
- Cada departamento possui uma série de produtos;

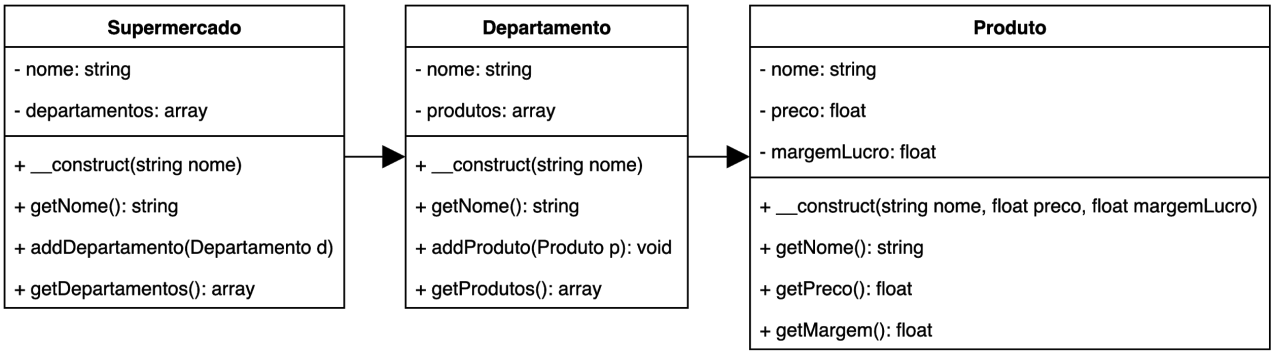


Diagrama com as classes de um sistema de rede de supermercados

Imagine que o dono da rede de supermercados deseja saber quantos reais de lucro cada produto, departamento e supermercado é capaz de gerar. Ele também disse que no futuro deseja adicionar novas funcionalidades a nível de produto, departamento e supermercado.

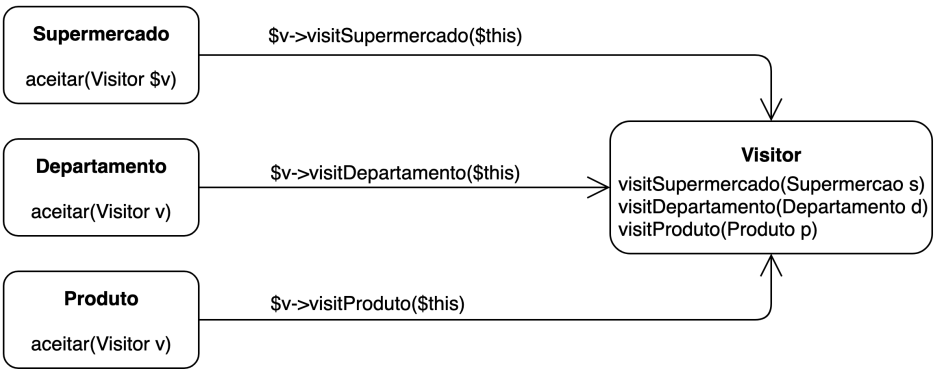
Uma possível abordagem para a adição da funcionalidade de cálculo de lucro seria criar um novo método em cada classe. Para isso as três classes teriam que ser editadas e bugs poderiam surgir. Além disso, ainda existe o fato de que novas funcionalidades serão pedidas no futuro, provocando assim mais mudanças nas 3 classes.

O padrão Visitor pode reduzir os impactos da adição de novas funcionalidades nas classes **Supermercado**, **Departamento** e **Produto**. Duas interfaces são as grandes responsáveis pela redução de tal impacto. São elas: **Visitor** e **Elemento** (É claro que os nomes podem ser mais apropriados ao contexto do código. Porém, para facilitar o entendimento iremos codificar com esses nomes mesmo).

```php
interface Visitor
{
    //Método para a classe Supermercado.
    public function visitSupermercado(Supermercado $supermercado): float;

    //Método para a classe Departamento.
    public function visitDepartamento(Departamento $departamento): float;

    //Método para a classe Produto.
    public function visitProduto(Produto $produto): float;
}
```

```php
interface Elemento
{
    //Método que diz para o Visitor qual classe está utilizando ele.
    public function aceitar(Visitor $visitor): float;
}
```

As classes **Supermercado**, **Departamento** e **Produto** precisarão implementar a interface **Elemento**, isso implica na edição do código nas três classes para a inserção do método **aceitar()** que é imposto pela interface. Tal método estará presente nas três classes mas a lógica do cálculo da lucratividade será implementada apenas no *Visitor*, a responsabilidade do método aceitar é apenas informar ao *Visitor*, que foi recebido por parâmetro, qual de seus métodos ele deve executar. Veja o esquema a seguir.



Comunicação entre interfaces do padrão *Visitor*

No futuro, quando for necessário adicionar novas funcionalidades às classes **Supermercado**, **Departamento** e **Produto** bastará criar outro **Visitor** e o passar por parâmetro para o método aceitar das 3 classes. Cada **Visitor** implementa uma funcionalidade diferente. Isso torna a aplicação escalável, pois não é mais necessário editar as classes **Supermercado**, **Departamento** e **Produto** a cada nova funcionalidade solicitada pelo dono da rede de supermercados.

Vejamos como ficam nossas classes implementando a interface **Elemento:**

```php
class Produto implements Elemento
{
    private string $nome;
    private float $preco;
    private float $margemLucro;

    public function __construct(string $nome, float $preco, float $margemLucro)
    {
        $this->nome = $nome;
        $this->preco = $preco;
        $this->margemLucro = $margemLucro;
    }

    public function getNome(): string
    {
        return $this->nome;
    }

    public function getPreco(): float
    {
        return $this->preco;
    }

    public function getMargemLucro(): float
    {
        return $this->margemLucro;
    }

    //Recebe um Visitor como parâmetro.
    public function aceitar(Visitor $visitor): float
    {
        //Diz para o visitor executar o seu método visitProduto().
        //Passa a própria classe por parâmetro para o visitor.
        return $visitor->visitProduto($this);
    }
}
```

```php
class Departamento implements Elemento
{
    private string $nome;
    private array $produtos;

    public function __construct(string $nome)
    {
        $this->nome = $nome;
    }

    public function addProduto(Produto $produto)
    {
        $this->produtos[] = $produto;
    }

    public function getNome(): string
    {
        return $this->nome;
    }

    public function getProdutos(): array
    {
        return $this->produtos;
    }

    //Recebe um Visitor como parâmetro.
    public function aceitar(Visitor $visitor): float
    {
        //Diz para o visitor executar o seu método visitDepartamento().
        //Passa a própria classe por parâmetro para o visitor.
        return $visitor->visitDepartamento($this);
    }
}
```

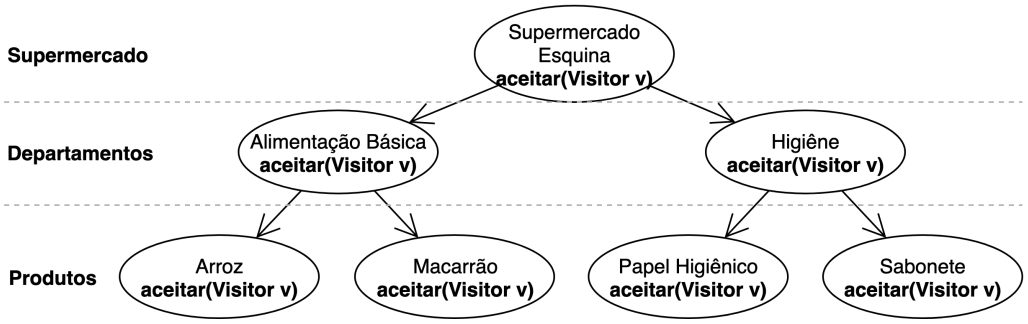
```php
class Supermercado implements Elemento
{
    private string $nome;
    private array $departamentos;

    public function __construct(string $nome)
    {
        $this->nome = $nome;
    }

    public function addDepartamento(Departamento $departamento)
    {
        $this->departamentos[] = $departamento;
    }

    public function getNome(): string
    {
        return $this->nome;
    }

    public function getDepartamentos(): array
    {
        return $this->departamentos;
    }

    //Recebe um Visitor como parâmetro.
    public function aceitar(Visitor $visitor): float
    {
        //Diz para o visitor executar o seu método visitSupermercado().
        //Passa a própria classe por parâmetro para o visitor.
        return $visitor->visitSupermercado($this);
    }
}
```

Já temos as interfaces definidas e as classes **Supermercado**, **Departamento** e **Produto** implementando **Elemento**, agora precisamos criar nosso *Visitor* concreto que será a classe **Lucratividade**.

```
class Lucratividade implements Visitor
{
    //Método que calcula a lucratividade de um supermercado.
    public function visitSupermercado(Supermercado $supermercado): float
    {
        $lucratividade = 0;
        $departamentos = $supermercado->getDepartamentos();
        //Calcula a lucratividade de cada departamento.
        foreach ($departamentos as $departamento) {
            $produtos = $departamento->getProdutos();
            //Calcula a lucratividade de cada produto do departamento.
            foreach ($produtos  as $produto) {
                $lucratividade += $this->calculaLucratividadeProduto($produto);
            }
        }

        return $lucratividade;
    }

    //Método que calcula a lucratividade de um departamento.
    public function visitDepartamento(Departamento $departamento): float
    {
        $lucratividade = 0;
        $produtos = $departamento->getProdutos();
        //Calcula a lucratividade de cada produto do departamento.
        foreach ($produtos  as $produto) {
            $lucratividade += $this->calculaLucratividadeProduto($produto);
        }

        return $lucratividade;
    }

    //Método que calcula a lucratividade de um produto.
    public function visitProduto(Produto $produto): float
    {
        return $this->calculaLucratividadeProduto($produto);
    }

    //Método que calcula a lucratividade de um produto (método base).
    private function calculaLucratividadeProduto(Produto $produto): float
    {
        return ($produto->getPreco() * $produto->getMargemLucro()) / 100;
    }
}
```

A classe **Lucratividade** implementa a interface **Visitor**. Ela é a classe que terá os métodos invocados pelo método **aceitar()** das classes que implementam **Elemento**.

Antes de fazermos o teste veja abaixo a hierarquia de objetos que iremos utilizar.



Exemplo de hierarquia de objetos de um supermercado (todos possuem o método aceitar)

Vamos ao código do teste:

```
//Criação do supermercado
$supermercado = new Supermercado('Supermercado Esquina');

//Criação do departamento 1 e seus produtos
$departamento_1 = new Departamento('Alimentação Básica');
$arroz = new Produto('Arroz 5Kg', 18, 30);
$macarrao = new Produto('Macarrão', 3.20, 15);

//Adição dos produtos ao departamento 1
$departamento_1->addProduto($arroz);
$departamento_1->addProduto($macarrao);

//Criação do departamento 2 e seus produtos
$departamento_2 = new Departamento('Higiene');
$papelHigienico = new Produto('Papel Higiênico', 11, 35);
$sabonete = new Produto('Sabonete', 1.20, 10);

//Adição dos produtos ao departamento 2
$departamento_2->addProduto($papelHigienico);
$departamento_2->addProduto($sabonete);

//Adição dos departamentos ao supermercado
$supermercado->addDepartamento($departamento_1);
$supermercado->addDepartamento($departamento_2);

//==== Cálculo de Lucratividade ====

//Criação do Visitor que calcula a lucratividade
$visitorLucratividade = new Lucratividade();

//Passagem do Visitor para o método Aceitar() do Supermercado, Departamentos e produtos.

$lucratividadeSupermercado = $supermercado->aceitar($visitorLucratividade);
echo 'Lucratividade Supermercado: R$' . $lucratividadeSupermercado . '<br><br>';

$lucratividadeDepartamento1 = $departamento_1->aceitar($visitorLucratividade);
echo 'Lucratividade Departamento 1: R$' . $lucratividadeDepartamento1 . '<br>';

$lucratividadeArroz = $arroz->aceitar($visitorLucratividade);
echo 'Lucratividade Arroz: R$' . $lucratividadeArroz . '<br>';

$lucratividadeMacarrao = $macarrao->aceitar($visitorLucratividade);
echo 'Lucratividade Macarrão: R$' . $lucratividadeMacarrao . '<br><br>';

$lucratividadeDepartamento2 = $departamento_2->aceitar($visitorLucratividade);
echo 'Lucratividade Departamento 2: R$' . $lucratividadeDepartamento2 . '<br>';

$lucratividadePapelHigienico = $papelHigienico->aceitar($visitorLucratividade);
echo 'Lucratividade Papel Higiênico: R$' . $lucratividadePapelHigienico . '<br>';

$lucratividadeSabonete = $sabonete->aceitar($visitorLucratividade);
echo 'Lucratividade Sabonete: R$' . $lucratividadeSabonete . '<br>';
```

Saída:

```
Lucratividade Supermercado: R$9.85

Lucratividade Departamento 1: R$5.88
Lucratividade Arroz: R$5.4
Lucratividade Macarrão: R$0.48

Lucratividade Departamento 2: R$3.97
Lucratividade Papel Higiênico: R$3.85
Lucratividade Sabonete: R$0.12
```

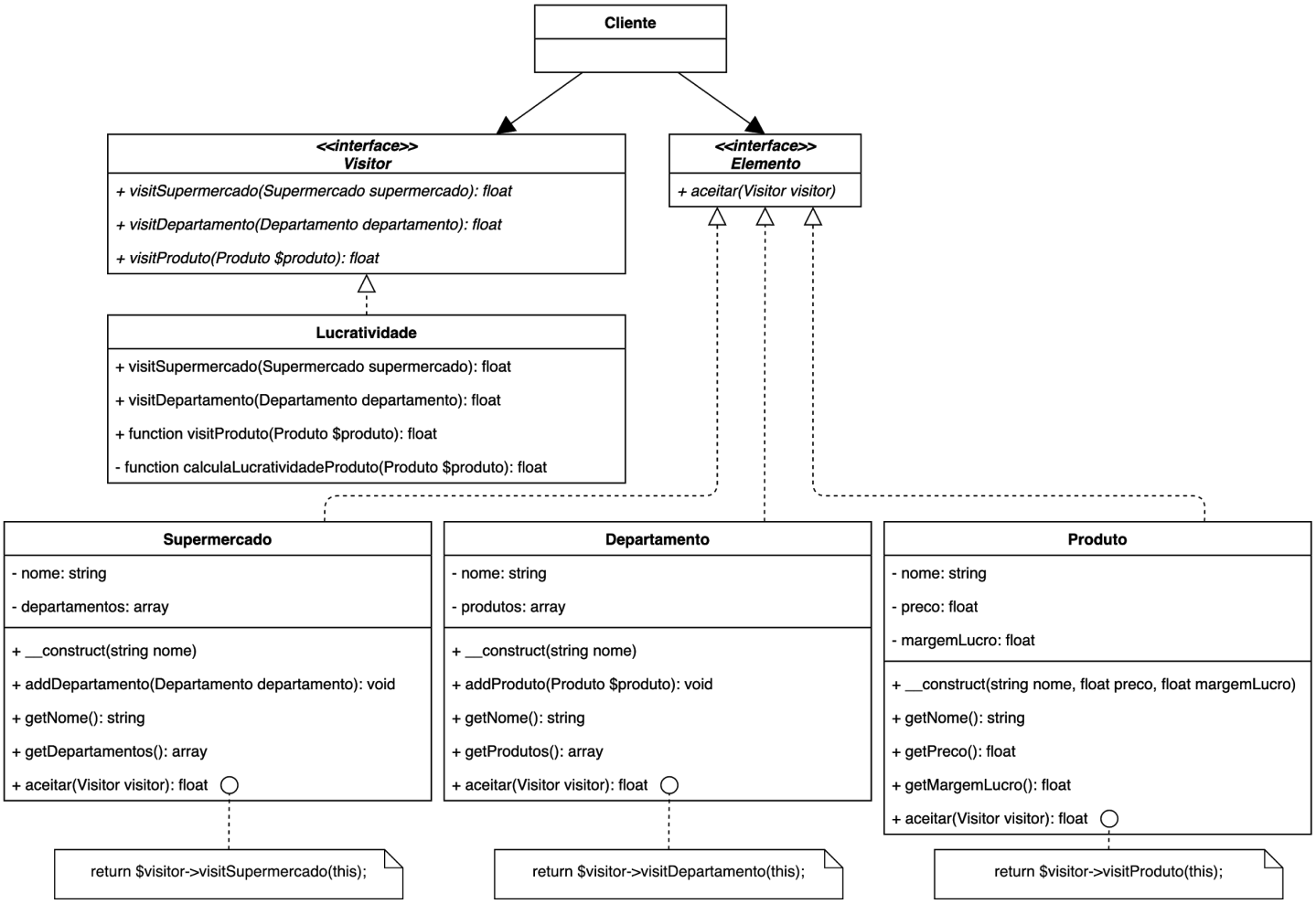


Diagrama de classes completo do exemplo

Aplicabilidade (Quando utilizar?)

- Quando uma estrutura de objetos contém muitas classes com interfaces diferentes e é necessário executar operações nas instâncias dessas classes, que por sua vez dependem de suas classes concretas.

- Quando muitas operações distintas e não relacionadas precisam ser executadas em objetos que compõem uma estrutura de objetos e não é desejável criar responsabilidades adicionais a tais objetos por conta de tais operações. O *Visitor* permite manter juntas as operações relacionadas, definindo-as em uma única classe. Quando a estrutura do objeto for compartilhada por muitas aplicações, use o *Visitor* para por operações apenas nas aplicações que precisam delas.

- Quando as classes que definem a estrutura do objeto raramente mudam, mas geralmente é preciso definir novas operações sobre tal estrutura. Alterar as classes da estrutura do objeto requer redefinir a interface para todos os *Visitors*, o que é muito trabalhoso e pode criar oportunidades para surgimentos de *bugs*. Se as classes de estrutura dos objetos mudam com frequência, provavelmente é melhor definir as operações nessas classes.

Componentes

- **Visitor:** Declara um método *Visit()* para cada classe de VisitanteConcreto na estrutura do objeto. O nome e assinatura do método indicam qual é a classe que envia a solicitação de visita ao Visitante. Isso permite que o Visitante saiba qual é o ElementoConcreto que está sendo visitado. Em seguida, o Visitante pode acessar o elemento diretamente através de sua interface específica.

- **VisitanteConcreto:** Um VisitanteConcreto implementa versões diferentes do mesmo comportamento, feitos sob medida para diferentes classes de ElementosConcretos.

- **Elemento:** Define uma operação aceitar() que aceita um Visitor como parâmetro.

- **ElementoConcreto:** Implementa uma operação aceitar() que aceita um Visitor como parâmetro. O propósito de tal método é redirecionar a chamada para o método apropriado do Visitor recebido como parâmetro que corresponde com a atual classe ElementoConcreto.

- **EstruturaDeObjetos:**
    - É opcional.
    - Pode enumerar seus elementos.
    - Pode fornecer uma interface de alto nível para permitir que o visitante visite seus elementos.
    - Pode ser uma composição (*design pattern Composite*) ou uma coleção tal como uma lista, pilha ou matriz.

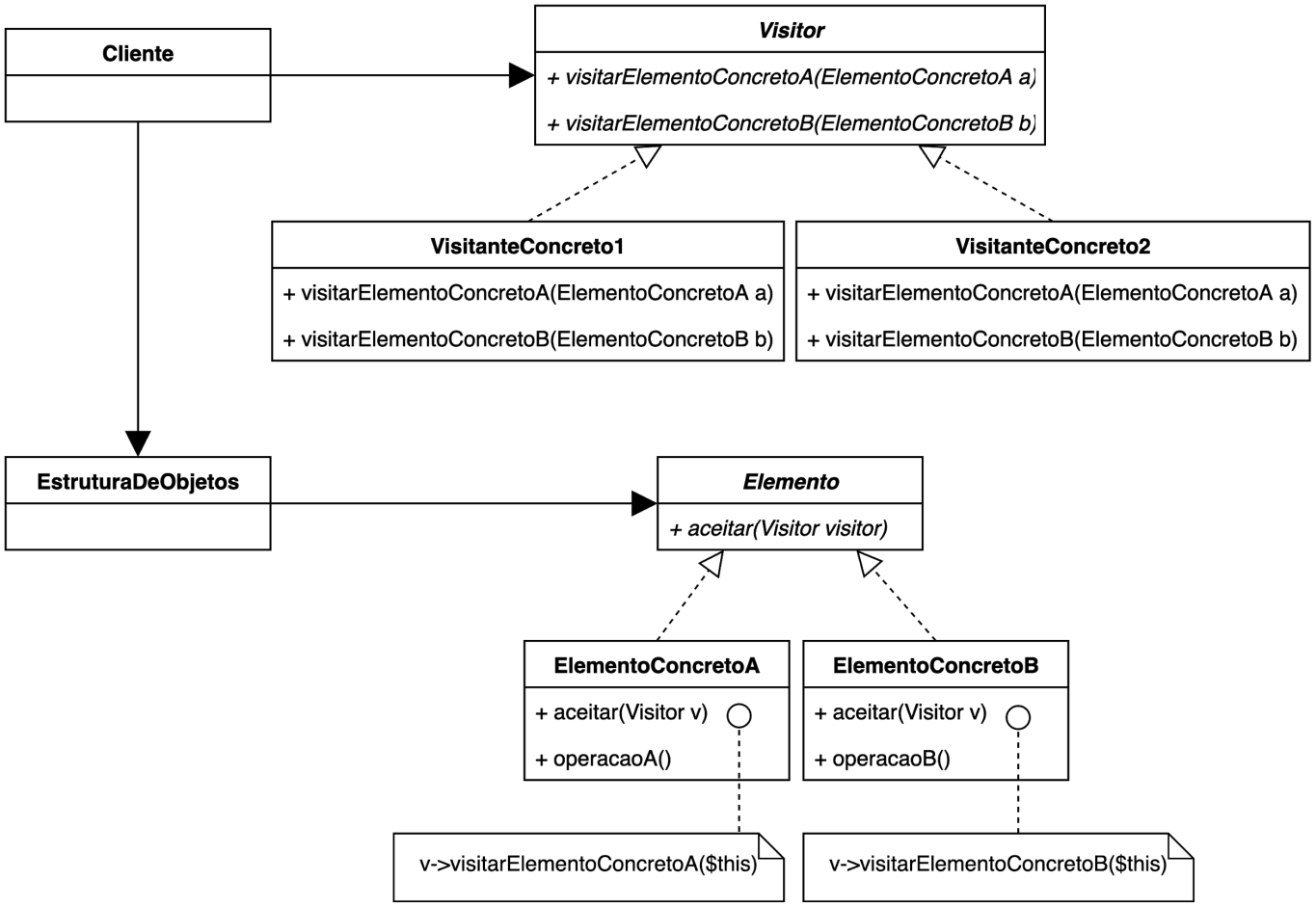


Diagrama de Classes

Consequências

- O padrão *Visitor* facilita a adição de operações que dependem dos componentes de objetos complexos. Pode-se definir uma nova operação sobre uma estrutura de objeto simplesmente criando um novo *Visitor*. Caso muitos *Visitors* sejam criados, as funcionalidades ficarão espalhadas por muitas classes.

- Um *Visitor* reúne operações relacionadas e separa as não relacionadas. O comportamento relacionado não está espalhado pelas classes que definem a estrutura dos objetos, mas sim, centralizados em um *Visitor*. Conjuntos de comportamento não relacionados são particionados em suas próprias subclasses de *Visitor*. Isso simplifica as classes que definem os Elementos e os algoritmos definidos nos *Visitors*. Qualquer estrutura de dados específica de algoritmo pode ser ocultada no *Visitor*.

- Adicionar novas classes ElementoConcreto é difícil. O padrão *Visitor* torna difícil adicionar novas subclasses de Elemento. Cada novo ElementoConcreto gera uma nova operação abstrata na interface *Visitor* e uma implementação correspondente em todas as classes VisitanteConcreto.

  Portanto, o principal fator a ser analisado antes da aplicação do padrão *Visitor:*

  - ➔ Se é mais provável que se altere o algoritmo aplicado sobre uma estrutura de objetos, o padrão *Visitor* é uma boa solução. Neste caso a hierarquia de objetos é estável, mesmo que seja preciso adicionar operações continuamente ou alterar algoritmos, o padrão Visitor ajudará a gerenciar tais alterações.

  - ➔ Se é mais provável que se altere as classes de objetos que compõem a estrutura (ElementoConcreto) o padrão *Visitor* poderá ser difícil de se manter. Neste caso é melhor adicionar as novas funcionalidades na própria classe que compõem a estrutura (ElementoConcreto).

- Estado acumulativo. Os *Visitors* podem acumular estados à medida que visitam cada elemento na estrutura do objeto. Sem um *Visitor*, esse estado seria passado como parâmetros extras para operações que executam o percurso de visitas a cada objeto da estrutura, ou então, tais estados poderiam aparecer como variáveis globais.

- Quebra de encapsulamento. A abordagem do padrão *Visitor* pressupõe que a interface ElementoConcreto seja poderosa o suficiente para permitir que os Visitors façam seu trabalho. Como resultado, o padrão geralmente obriga os ElementosConcreto a fornecer operações públicas que acessam seus estados internos, o que pode comprometer seu encapsulamento.