Strategy

Padrões Comportamentais

O padrão de projeto *Strategy* define uma família de algoritmos, encapsula cada um deles e os torna intercambiáveis. O *Strategy* permite que o algoritmo varie independentemente dos clientes que o utilizam.

Motivação (Por que utilizar?)

O padrão *Strategy* aprimora a comunicação entre objetos, pois passa a existir uma distribuição de responsabilidades. O objetivo é representar os comportamentos de um objeto por meio de uma família de algoritmos que os implementam.

Em projetos de *software* orientados a objetos é possível encontrar objetos semelhantes que variam seu comportamento apenas em alguns pontos específicos. Cada um destes comportamentos pode ser externalizado em uma família de algoritmos.

Tome como exemplo o cálculo do valor de diferentes tipos de frete para um pedido realizado em um e-commerce. Os fretes disponíveis são **Frete Comum** e **Frete Expresso**. Todos os pedidos devem saber calcular seu frete.

Uma solução seria implementar os métodos de cálculo de frete na classe **Pedido**. No momento do cálculo do frete bastaria chamar o método responsável por calcular o frete escolhido.

```php
class Pedido
{
    protected float $valor;

    public function getValor(): float
    {
        return $this->valor;
    }

    public function setValor(float $valor): void
    {
        $this->valor = $valor;
    }

    public function calculaFreteComum(): float
    {
        return $this->valor * 0.05; //O frete comum custa 5% o valor do pedido.
    }

    public function calculaFreteExpresso(): float
    {
        return $this->valor * 0.1;  //O frete comum custa 10% o valor do pedido.
    }
}
```

Vejamos um teste da classe pedido.

```php
//Criação do pedido.
$pedido = new Pedido();

//Atribuição do valor;
$pedido->setValor(100);

//Cálculo do frete comum.
echo 'Frete Comum: R$' . $pedido->calculaFreteComum() . '<br>';
//Cálculo do frete Expresso.
echo 'Frete Expresso: R$' . $pedido->calculaFreteExpresso() . '<br><br>';
```

Saída:

```
Frete Comum: R$5
Frete Expresso: R$10
```

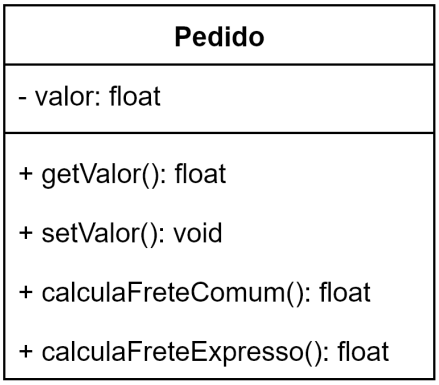


Diagrama de classes do exemplo no cenário 1

Imagine que agora o e-commerce cresceu e foi dividido em setores. Os pedidos de cada setor possuem características diferentes, de modo a ser necessária a criação de uma classe de pedido para cada setor. Inicialmente existirão dois setores, móveis e eletrônicos. Neste caso basta transformar a superclasse **Pedido** em uma superclasse abstrata, que por sua vez implementa os métodos responsáveis por calcular os diferentes tipos de frete. Os pedidos de cada setor, que são subclasses, herdam as características da classe **Pedido**, e graças a herança também passam a saber calcular os diferentes tipos de frete.

```php
abstract class Pedido
{
    protected float $valor;

    public function getValor(): float
    {
        return $this->valor;
    }

    public function setValor(float $valor): void
    {
        $this->valor = $valor;
    }

    public function calculaFreteComum(): float
    {
        return $this->valor * 0.05; //O frete comum custa 5% o valor do pedido.
    }

    public function calculaFreteExpresso(): float
    {
        return $this->valor * 0.1; //O frete comum custa 10% o valor do pedido.
    }
}
```

Agora vamos criar as subclasses de cada setor. Repare que estamos criando uma pequena diferença nos construtores apenas para fins didáticos.

```php
class PedidoEletronicos extends Pedido
{
    private string $nomeSetor;

    public function __construct()
    {
        $this->nomeSetor = 'Eletrônicos';
    }

    public function getNomeSetor(): string
    {
        return $this->nomeSetor;
    }

    public function setNomeSetor(string $nomeSetor): void
    {
        $this->nomeSetor = $nomeSetor;
    }
}
```

```php
class PedidoMoveis extends Pedido
{
    private string $nomeSetor;

    public function __construct()
    {
        $this->nomeSetor = 'Móveis';
    }

    public function getNomeSetor(): string
    {
        return $this->nomeSetor;
    }

    public function setNomeSetor(string $nomeSetor): void
    {
        $this->nomeSetor = $nomeSetor;
    }
}
```

Vamos ao teste:

```php
//Criação de um pedido do setor de móveis. (O teste seria semelhante para Eletrônicos)
$pedido = new PedidoEletronicos();

//Atribuição do valor
$pedido->setValor(100);

//Cálculos dos dois tipos de frete
echo 'Frete Comum: R$' . $pedido->calculaFreteComum() . '<br>';
echo 'Frete Expresso: R$' . $pedido->calculaFreteExpresso() . '<br><br>';
```

Saída:

```
Frete Comum: R$5
Frete Expresso: R$10
```

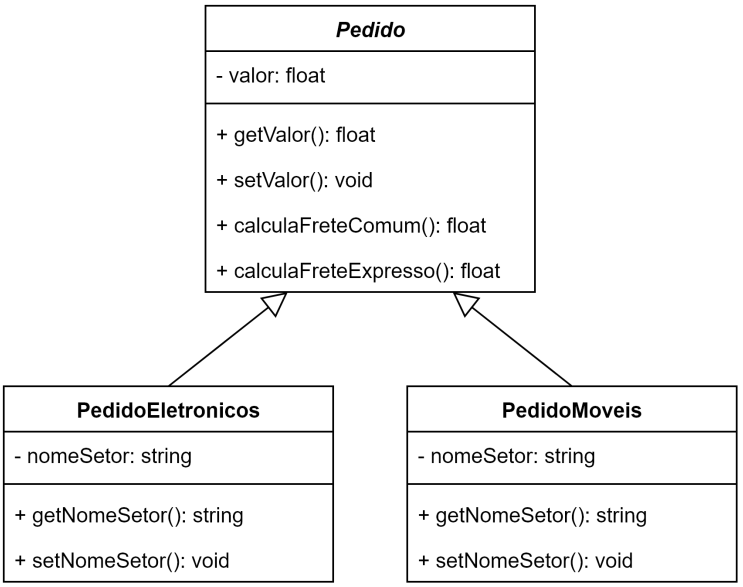


Diagrama de classes do exemplo no cenário 2

Tudo certo até agora, mas considere que o setor de móveis fica em um estado do Brasil onde o frete comum é o único disponível. Temos um problema, pois devido a herança todas as subclasses de **Pedido** podem calcular todos os tipos de frete, porém, a subclasse **PedidoMoveis** deveria aceitar apenas o frete econômico.

É possível contornar isso tornando abstratos os métodos de cálculo de frete da classe abstrata **Pedido**, Assim, todos os subtipos de que herdam de **Pedido** serão obrigados e implementar seus próprios cálculos de frete.

```php
abstract class Pedido
{
    protected float $valor;

    public function getValor(): float
    {
        return $this->valor;
    }

    public function setValor(float $valor): void
    {
        $this->valor = $valor;
    }

    //A implementação deste método passa a ser responsabilidade das subclasses.
    abstract public function calculaFreteComum(): float;

    //A implementação deste método passa a ser responsabilidade das subclasses.
    abstract public function calculaFreteExpresso(): float;
}
```

```php
class PedidoEletronicos extends Pedido
{
    private string $nomeSetor;

    public function __construct()
    {
        $this->nomeSetor = 'Eletrônicos';
    }

    public function getNomeSetor(): string
    {
        return $this->nomeSetor;
    }

    public function setNomeSetor(string $nomeSetor): void
    {
        $this->nomeSetor = $nomeSetor;
    }

    public function calculaFreteComum(): float
    {
        return $this->valor * 0.05; //O frete comum custa 5% o valor do pedido.
    }

    public function calculaFreteExpresso(): float
    {
        return $this->valor * 0.1; //O frete expresso custa 10% o valor do pedido.
    }
}
```

```php
class PedidoMoveis extends Pedido
{
    private string $nomeSetor;

    public function __construct()
    {
        $this->nomeSetor = 'Móveis';
    }

    public function getNomeSetor(): string
    {
        return $this->nomeSetor;
    }

    public function setNomeSetor(string $nomeSetor): void
    {
        $this->nomeSetor = $nomeSetor;
    }

    public function calculaFreteComum(): float
    {
        return $this->valor * 0.05; //O frete comum custa 5% o valor do pedido.
    }

    public function calculaFreteExpresso(): float
    {
        throw new \Exception('Indisponível'); //Frete não aceito, uma exceção é lançada.
    }
}
```

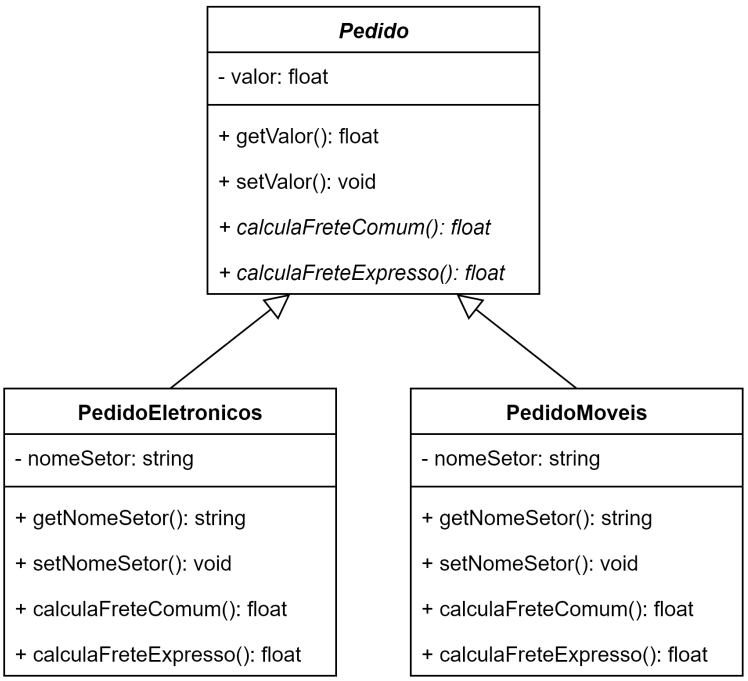


Diagrama de classes do exemplo no cenário 3

Com a essa solução cada sub-pedido controla seus fretes, a subclasse **PedidoMoveis** pode bloquear o frete expresso dentro dela. A desvantagem dessa abordagem é que não existe reaproveitamento de código. Repare que o método **calculaFreteComum()** é exatamente igual nas duas subclasses, se ele mudar todas as subclasses de **Pedido** deverão ser editadas. No momento existem apenas duas subclasses, mas imagine se forem dez. Essa edição de todas as subclasses além de ser muito trabalhosa pode permitir o surgimento de *bugs* no processo.

O padrão *Strategy* encapsula algoritmos que representam um comportamento similar, ou seja, isola o código que toma a decisão de modo que ele possa ser editado ou incrementado de forma totalmente independente.

Podemos utilizar o *Strategy* em nosso problema, vamos começar com uma interface chamada de **Frete** que possui o método **calcula()**. Tal interface define uma família de algoritmos, onde cada membro dessa família é capaz de calcular um determinado tipo de frete.

```
interface Frete
{
    //Recebe o valor do pedido que é utilizado no calculo do frete.
    public function calcula(float $valorPedido): float;
}
```

```
class FreteComum implements Frete
{
    public function calcula(float $valorPedido): float
    {
        return $valorPedido * 0.05; //O frete comum custa 5% o valor do pedido.
    }
}
```

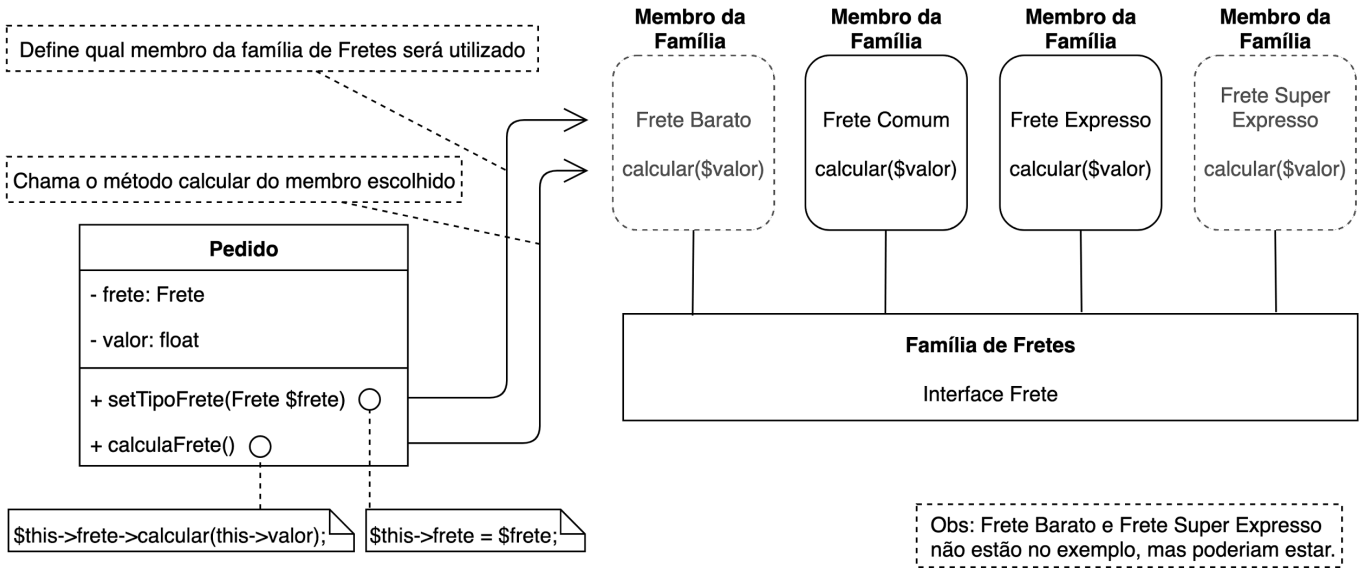
```
class FreteExpresso implements Frete
{
    public function calcula(float $valorPedido): float
    {
        return $valorPedido * 0.1; //O frete expresso custa 10% o valor do pedido.
    }
}
```

Cada uma das classes acima implementam a interface **Frete**, portanto todas elas possuem o método **calcula()** que faz o cálculo conforme sua fórmula interna. Agora o cálculo de frete não fica mais na classe **Pedido** e nem em suas subclasse, este comportamento foi encapsulado em classes separadas.

Com a criação das classes de frete, basta adaptar que a classe abstrata **Pedido** para que ela tenha os seguintes métodos métodos:

- **setTipoFrete(Frete $frete):** Este método recebe como parâmetro um objeto que implementa a interface **Frete** e mantenha a instância deste objeto em uma de suas variáveis internas.

- **calculaFrete():** Tal método é o responsável por invocar o método **calcula()** do objeto que foi recebido por **setTipoFrete(Frete $frete)**.



Esquema da utilização de famílias de algoritmos

```php
abstract class Pedido
{
    protected float $valor;
    protected Frete $tipoFrete; //Referência para o tipo de frete.

    public function getValor(): float
    {
        return $this->valor;
    }

    public function setValor(float $valor): void
    {
        $this->valor = $valor;
    }

    //Define o tipo de frete.
    public function setTipoFrete(Frete $frete)
    {
        $this->tipoFrete = $frete;
    }

    //Calcula o frete de acordo com a classe Frete recebida.
    public function calculaFrete()
    {
        return $this->tipoFrete->calcula($this->valor);
    }
}
```

Uma vez que o cálculo de frete foi encapsulado em classes separadas, ele deve sair das subclasses de **Pedido**.

```
class PedidoEletronicos extends Pedido
{
    private string $nomeSetor;

    public function __construct()
    {
        $this->nomeSetor = 'Eletrônicos';
    }

    public function getNomeSetor(): string
    {
        return $this->nomeSetor;
    }

    public function setNomeSetor(string $nomeSetor): void
    {
        $this->nomeSetor = $nomeSetor;
    }
}
```

```
class PedidoMoveis extends Pedido
{
    private string $nomeSetor;

    public function __construct()
    {
        $this->nomeSetor = 'Móveis';
    }

    public function getNomeSetor(): string
    {
        return $this->nomeSetor;
    }

    public function setNomeSetor(string $nomeSetor): void
    {
        $this->nomeSetor = $nomeSetor;
    }
}
```

Vamos fazer o teste de cálculo de fretes de um pedido do setor de eletrônicos e móveis utilizando o padrão *Strategy*:

```
//Criação do tipos de frete.
$freteComum = new FreteComum();
$freteExpresso = new FreteExpresso();

//Criação de um pedido do setor de eletrônicos.
$pedido = new PedidoEletronicos();

//Atribuição do valor.
$pedido->setValor(100);

//Definição do frete comum com sendo o frete escolhido.
$pedido->setTipoFrete($freteComum);

//Cálculo do frete comum.
echo 'Eletrônicos Frete Comum: R$' . $pedido->calculaFrete() . '<br>';
//No mesmo medido podemos trocar o tipo de frete
$pedido->setTipoFrete($freteExpresso);

//Agora o cálculo do frete expresso.
echo 'Eletrônicos Frete Expresso: R$' . $pedido->calculaFrete() . '<br>';

//Criação de um pedido do setor de eletrônicos.
$pedido = new PedidoMoveis();

//Atribuição do valor.
$pedido->setValor(100);

//Definição do frete comum com sendo o frete escolhido. Expresso não está disponível.
$pedido->setTipoFrete($freteComum);

//Cálculo do frete comum.
echo 'Móveis Frete Comum: R$' . $pedido->calculaFrete() . '<br>';
```

Saída:

```
Eletrônicos Frete Comum: R$5
Eletrônicos Frete Expresso: R$10
Móveis Frete Comum: R$5
```

Com isso o algoritmo fica mais flexível, o tipo de frete passa a ser definido dinamicamente em tempo de execução. Além disso passa a obedecer alguns princípios básicos da OO:

1. **Programe para abstrações:** a classe *Pedido* não depende diretamente de nenhum calculador de frete concreto e sim da interface *Frete*.

2. **Open-closed principle:** o Pedido não terá nenhum impacto um novo tipo de frete seja aceito pelo e-commerce. Bastaria criar uma nova classe de frete que implemente a interface *Frete*.

3. **De prioridade a composição em relação à herança:** ao invés de herdar os cálculos de frete os pedidos obtém a capacidade de calcular os fretes ao serem compostos pelo objeto do tipo Frete que lhe convém.

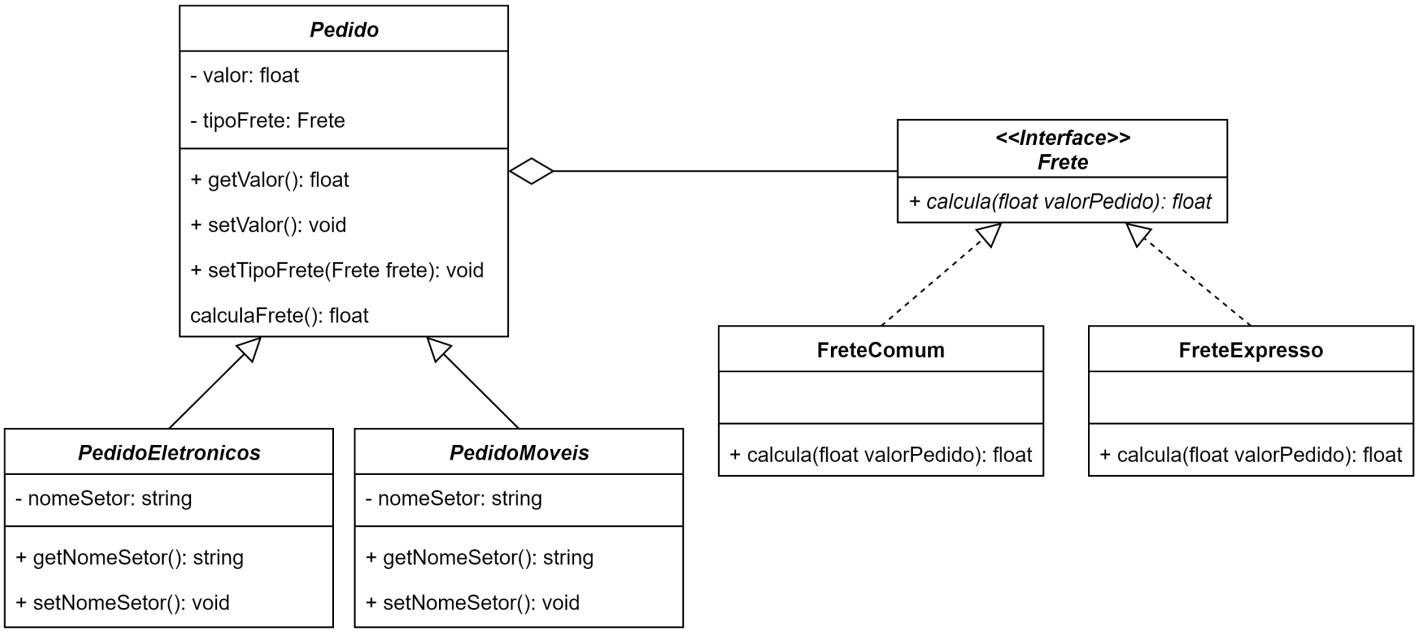


Diagrama de classes do exemplo no cenário 3 (Utilizando o padrão *Strategy*)

Aplicabilidade (Quando utilizar?)

- O padrão é aplicado quando muitas classes fazem a mesma coisa de forma diferente.
- Quando se necessita de variantes de um algoritmo.
- Quando é necessário evitar a exposição de dados ou algoritmos sensíveis os quais clientes não podem ter conhecimento.
- Remoção de operadores condicionais que determinam o comportamento do algoritmo com base em objetos diferentes.

Componentes

- **Contexto:** Classe que é composta por um objeto que implementa a interface *Strategy*. Ele é responsável por orquestrar as classes EstrategiasConcretas. Sempre que uma solicitação é feita à classe contexto ela é delegada o objeto *Strategy* que a compõe.
- **Strategy:** Contrato que as EstragiaConcretas devem respeitar. Tal contrato será exigido pela classe Contexto.
- **EstrategiaConcreta:** Lidam com as solicitações provenientes do contexto. Cada EstrategiaConcreta fornece a sua própria implementação

de uma solicitação. Deste modo, quando o contexto muda de estratégia o seu comportamento também muda.

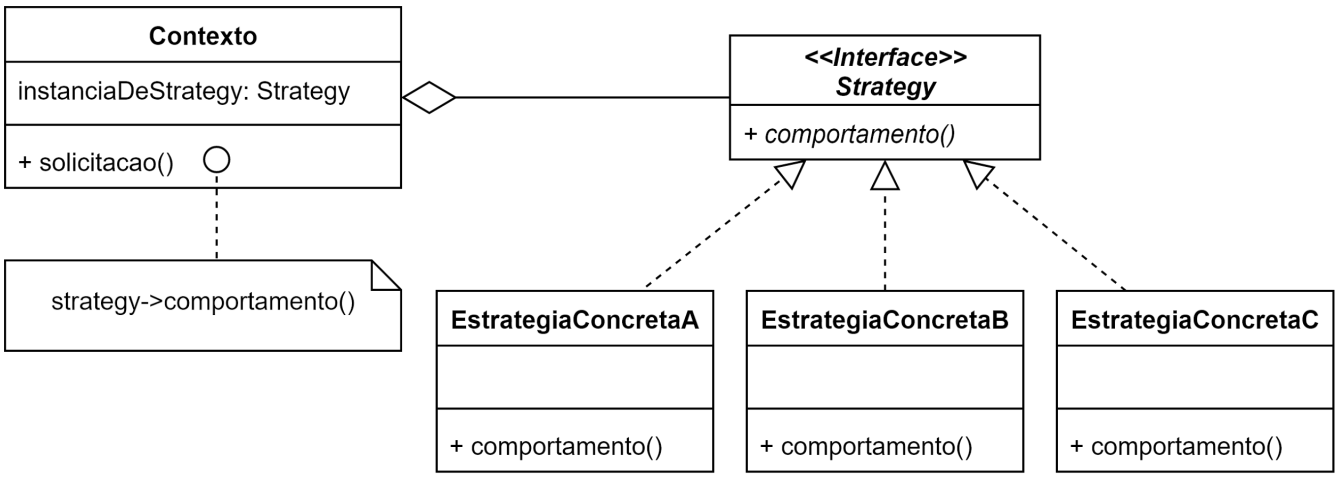


Diagrama de Classes

Consequências

- Família de Algoritmo: Permite a criação de uma hierarquia de classes do tipo *Strategy* em um mesmo contexto.
- O encapsulamento dos algoritmos nas classes Strategy permite alterar o algoritmo independentemente do seu contexto, tornando mais fácil de efetuar possíveis alterações no código.
- É uma estratégia para remover operadores condicionais.
- Flexibilidade na escolha de qual *strategy* (algoritmo) utilizar.
- Clientes devem conhecer as classes Strategy, pois, se o cliente não compreender como essas classes funcionam, não poderá escolher o melhor comportamento.
- Custo entre a comunicação Strategy e Context: as classes que implementam a interface *Strategy* podem não utilizar as informações passadas por ela, ou seja, pode acontecer da classe Contexto criar e iniciar parâmetros que não serão utilizados.
- Aumento do número de classes na aplicação.