Strategy

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 quê 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(CommonShipping)` e `Frete Expresso(ExpressShipping)`. Todos os pedidos devem saber calcular seu frete.

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

```
export class Order {
  protected _value: number

  public get value(): number {
    return this._value
  }

  public set value(value: number) {
    this._value = value
  }

  public calculateCommonShipping(): number {
    return this._value * 0.05
  }
```

```
  public calculateExpressShipping(): number {
    return this._value * 0.1
  }
}
```

Vejamos um teste da classe pedido.

```
const order = new Order()
order.value = 100

console.log('Common shipping: R$', order.calculateCommonShipping())
console.log('Express shipping: R$', order.calculateExpressShipping())
```

Saída:

```
Common shipping: R$ 5
Express shipping: R$ 10
```

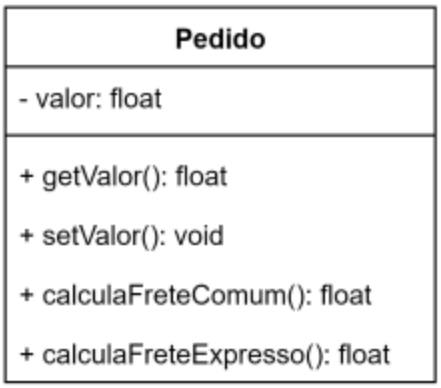


Diagrama de classes do exemplo no cenário 1

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.

```
export abstract class Order {
  protected _value: number

  public get value(): number {
    return this._value
  }

  public set value(value: number) {
    this._value = value
  }

  public calculateCommonShipping(): number {
    return this._value * 0.05
  }

  public calculateExpressShipping(): number {
    return this._value * 0.1
  }
}
```

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

```
export class EletronicOrder extends Order {
  private _sectorName: string

  constructor(){
    super()
    this._sectorName = 'Eletronic'
  }

  get sectorName(): string {
    return this._sectorName
  }

  set sectorName(value: string) {
    this._sectorName = value
  }
}
```

```
export class FurnitureOrder extends Order {
  private _sectorName: string
```

```
  constructor(){
    super()
    this._sectorName = 'Furniture'
  }

  get sectorName(): string {
    return this._sectorName
  }

  set sectorName(value: string) {
    this._sectorName = value
  }
}
```

Vamos ao teste:

```
const order = new EletronicOrder()
order.value = 100

console.log('Common shipping: R$', order.calculateCommonShipping())
console.log('Express shipping: R$', order.calculateExpressShipping())
```

Saída:

```
Common shipping: R$ 5
Express shipping: R$ 10
```

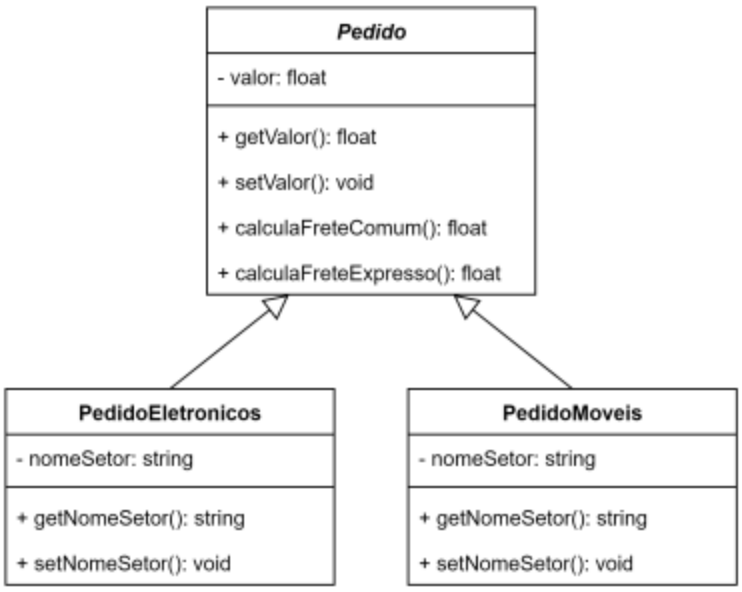


Diagrama de classes do exemplo no cenário 2

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.

```
abstract class Order {
  protected _value: number

  public get value(): number {
    return this._value
  }

  public set value(value: number) {
    this._value = value
  }

  public abstract calculateCommonShipping(): number

  public abstract calculateExpressShipping(): number
}
```

```
class EletronicOrder extends Order {
  private _sectorName: string

  constructor(){
    super()
    this._sectorName = 'Eletronic'
  }

  get sectorName(): string {
    return this._sectorName
  }

  set sectorName(value: string) {
    this._sectorName = value
  }

  public calculateCommonShipping(): number {
    return this._value * 0.05
  }
  public calculateExpressShipping(): number {
    return this._value * 0.1
  }
}
```

```
class FurnitureOrder extends Order {
  private _sectorName: string

  constructor(){
    super()
    this._sectorName = 'Furniture'
  }

  get sectorName(): string {
    return this._sectorName
  }

  set sectorName(value: string) {
    this._sectorName = value
  }

  public calculateCommonShipping(): number {
    return this._value * 0.05
  }
  public calculateExpressShipping(): number {
    throw new Error("Method not implemented.")
  }
}
```

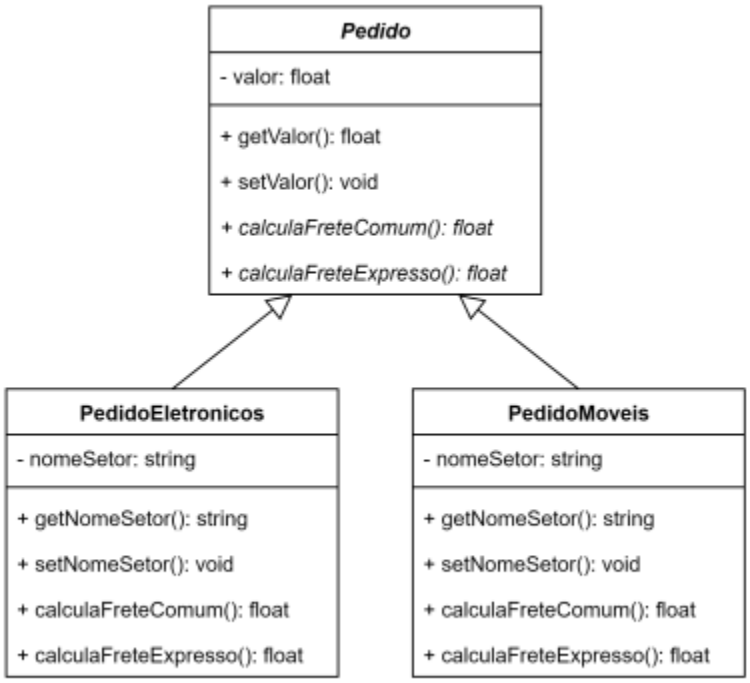


Diagrama de classes do exemplo no cenário 3

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 IShipping {
  calculate(value: number): number
}
```

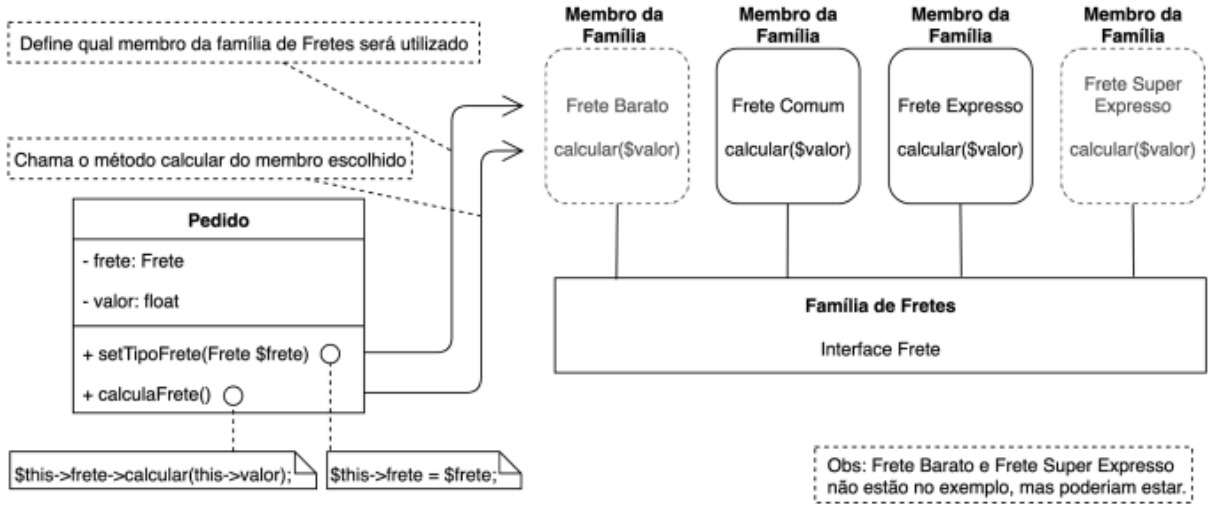
```
class CommonShipping implements IShipping {
  public calculate(value: number): number {
    return value * 0.05
  }
}
```

```
class ExpressShipping implements IShipping {
  public calculate(value: number): number {
    return value * 0.1
  }
}
```

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

```
import { IShipping } from '../interface/shipping.interface'

abstract class Order {
  protected _value: number
  protected _shipping: IShipping;

  public get value(): number {
    return this._value
  }

  public set value(value: number) {
    this._value = value
  }

  public set shipping(shipping: IShipping) {
    this._shipping = shipping
  }

  public calculateShipping(): number {
    return this._shipping.calculate(this.value)
  }
}
```

```
import { Order } from "./order"

export class EletronicOrder extends Order {
  private _sectorName: string

  constructor(){
    super()
```

```
    this._sectorName = 'Eletronic'
  }

  get sectorName(): string {
    return this._sectorName
  }

  set sectorName(value: string) {
    this._sectorName = value
  }
}
```

```
import { Order } from "./order"

export class FurnitureOrder extends Order {
  private _sectorName: string

  constructor(){
    super()
    this._sectorName = 'Furniture'
  }

  get sectorName(): string {
    return this._sectorName
  }

  set sectorName(value: string) {
    this._sectorName = value
  }
}
```

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

```
import { EletronicOrder } from './class/eletronic-order'
import { FurnitureOrder } from './class/furniture-order'

import { CommonShipping } from './class/common-shipping'
import { ExpressShipping } from './class/express-shipping'


const eletronicOrder = new EletronicOrder()
eletronicOrder.value = 100
eletronicOrder.shipping = new CommonShipping()
console.log('Eletronic common shipping: R$', eletronicOrder.calculateShipping())

eletronicOrder.shipping = new ExpressShipping()
console.log('Eletronic express shipping: R$', eletronicOrder.calculateShipping())
```

```
const furnitureOrder = new FurnitureOrder()
furnitureOrder.value = 100
furnitureOrder.shipping = new CommonShipping()
console.log('Furniture common shipping: R$', furnitureOrder.calculateShipping())
```

Saída:

```
Eletronic common shipping: R$ 5
Eletronic express shipping: R$ 10
Furniture common shipping: 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:

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

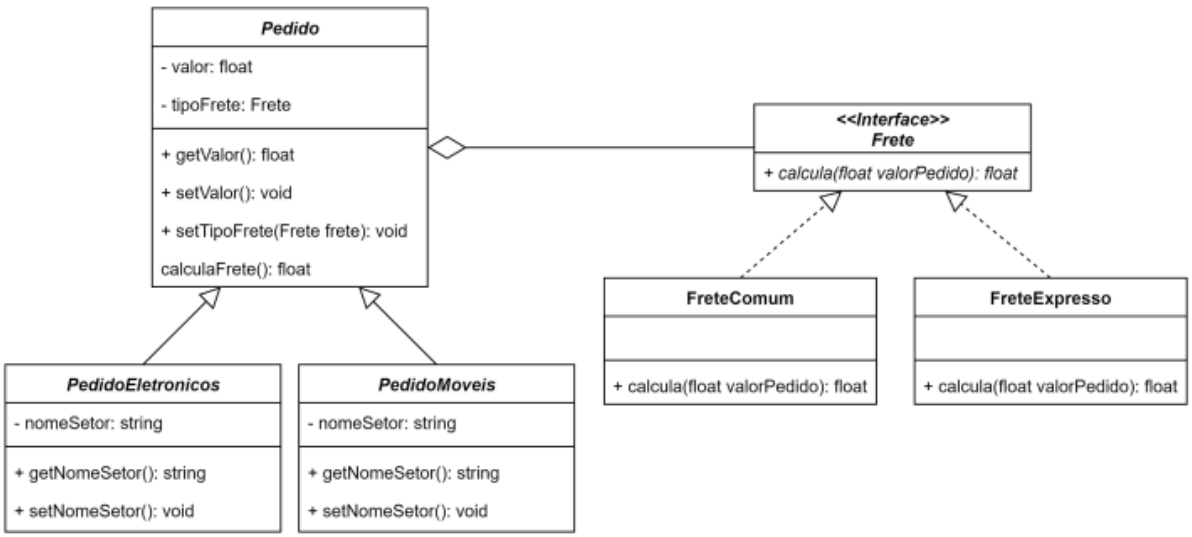


Diagrama de classes do exemplo no cenário 3 (Utilizando o padrão *Strategy*)

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.

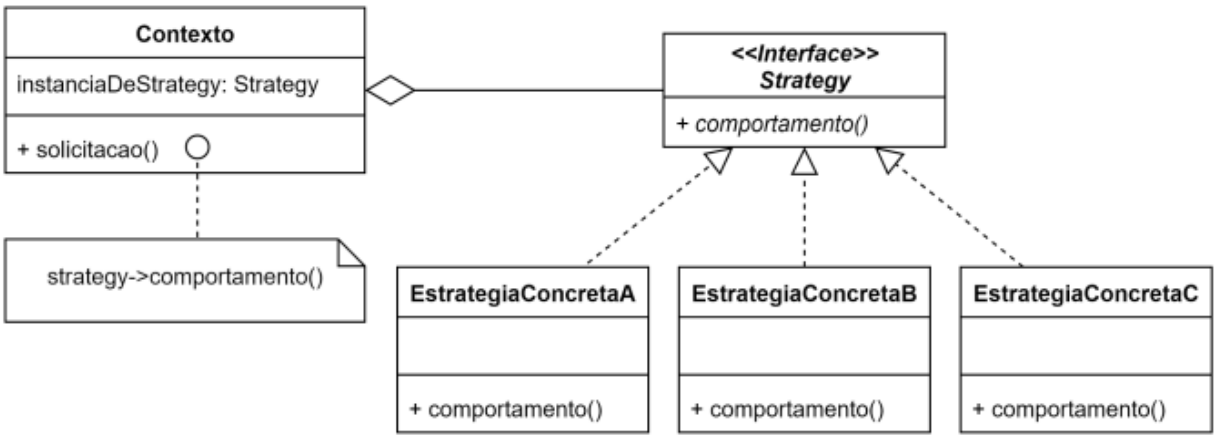


Diagrama de Classes

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.