Chain of Responsibility

Padrões Comportamentais

O Padrão *Chain of Responsibility* evita o acoplamento do remetente de uma solicitação ao seu receptor, dando a mais de um objeto a oportunidade de tratar a solicitação. Ele encadeia os objetos receptores, passando a solicitação ao longo da cadeia até que um objeto a trate.

Motivação (Por que utilizar?)

Durante o desenvolvimento de software é comum o surgimento de situações onde é necessário que apenas um tratamento entre muitos seja aplicado a um determinado fluxo. Isso pode parecer abstrato demais, então, apenas para exemplificar considere o código abaixo:

```
class Exemplo
{
    public int $atributo;

    public function testarAtributo()
    {
        if ($this->atributo < 10) {
            //Implementação
        } elseif ($this->atributo < 200) {
            //Implementação
        } elseif ($this->atributo < 500) {
            //Implementação
        } else {
            //Implementação
        }
    }
}
```

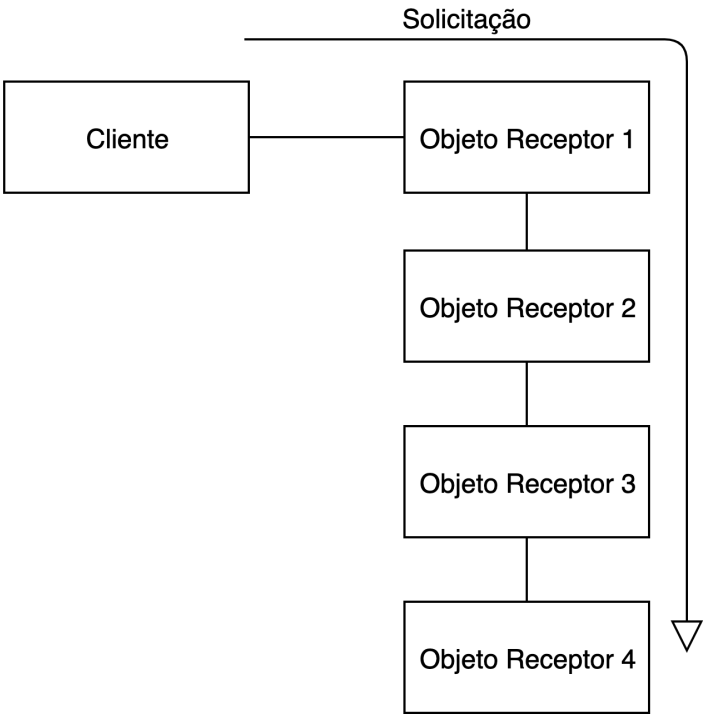
É esperado que apenas uma das condições do método `testarAtributo()` seja satisfeita. Assim que a primeira for satisfeita não é mais necessário verificar as demais condições, ou seja, apenas um tratamento entre muitos é necessário.

Repare os comentários "`//Implementação`" no código acima, tais comentários representam implementações que podem conter de uma, centenas, ou nos piores casos, milhares de linhas de código.

Códigos como o de cima, quando mal implementados, podem se tornar difíceis de ler e conter repetições de código dentro de cada condição.

Além das possíveis más práticas de programação também existe o problema de forte acoplamento. A classe `Exemplo` precisa conhecer todas as manipulações possíveis para o `atributo`. Imagine que a `‘//implementação’` contenha instâncias de diversos objetos de outras classes e essas classes foram editadas por algum motivo. Pense também que pode ser necessário adicionar ou remover uma condição de tratamento do `atributo`. De uma forma ou de outra a classe `Exemplo` terá que ser editada por consequência.

O Padrão *Chain of Responsibility* tem como objetivo evitar o acoplamento entre o objeto solicitante de uma chamada a um objeto solicitado. Isso é possível devido ao fato do padrão *Chain of Responsibility* dar a mais de um objeto a oportunidade de tratar a solicitação por meio de um encadeamento.



Cadeia de objetos que podem responder uma solicitação

Vamos nos apegar a algo mais concreto. Considere o cenário onde uma hamburgueria está criando um plano de fidelização de clientes. O dono percebeu que o faturamento da hamburgueria cai consideravelmente na segunda quinzena dos meses. Visando resolver esse problema, ele deseja que do dia 16 até o último dia do mês os pontos ganhos pelos cliente sejam o dobro do que ganhavam na primeira quinzena.

Foram estabelecidos critérios de pontuação para determinar quantos pontos um pedido dá a um cliente no programa de fidelização:

Critérios	**Pontos ganhos na (Primeira quinzena)**	**Pontos ganhos na (Segunda quinzena)**
Pedido acima de R$69,99	1 Ponto a cada 5 reais	2 Pontos a cada 5 reais
Pedido acima de R$39,99	1 Ponto a cada 7 reais	2 Pontos a cada 7 reais
Pedido acima de R$19,99	1 Ponto a cada 10 reais	2 Pontos a cada 10 reais
Pedidos abaixo de R$20,00	0 pontos	0 Pontos

Apenas para viabilizar o exemplo leve em consideração a classe **Pedido** abaixo.

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

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

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

Sem a aplicação do Padrão *Chain of Responsibility* o problema poderia ser resolvido da seguinte maneira:

```php
class CalculadoraDePontos
{
    public function calcularPontosDoPedido(Pedido $pedido, int $dia): int
    {
        if ($pedido->getValor() >= 70) { //Pedido acima de R$69,99
            $pontos = intdiv($pedido->getValor(),  5);
        } elseif ($pedido->getValor() >= 40) { //Pedido acima de R$39,99
            $pontos = intdiv($pedido->getValor(),  7);
        } elseif ($pedido->getValor() >= 20) { //Pedido acima de R$19,99
            $pontos = intdiv($pedido->getValor(),  10);
        } else { //Pedidos abaixo de R$19,99
            return 0;
        }

        if ($dia >= 16 && $dia <= 31) { //Se for a segunda quinzena.
            return $pontos * 2; //retorna o dobro de pontos.
        }

        return $pontos; //Senão retorna normalmente.
    }
}
```

Nota: *a função* **intdiv** *do PHP Retorna o resultado inteiro da divisão do primeiro parâmetro pelo segundo.*

Testando:

```
//Criação de um pedido.
$pedido = new Pedido();

//Criação da calculadora de pontos.
$calculadoraDePontos = new CalculadoraDePontos();

//Definição do valor do pedido igual 21 reais.
$pedido->setValor(21);

//Cálculo de pontos na primeira quinzena (dia 15).
echo 'Dia 15: ' . $calculadoraDePontos->calcularPontosDoPedido($pedido, 15) . " Pontos<br/>";
//Cálculo de pontos na segunda quinzena (dia 16).
echo 'Dia 16: ' . $calculadoraDePontos->calcularPontosDoPedido($pedido, 16) . " Pontos<br/>";

//Apenas para separar os resultados.
echo "--------------------------<br>";

//Mudança do valor do pedido para 100 reais.
$pedido->setValor(100);

//Cálculo de pontos na primeira quinzena (dia 15).
echo 'Dia 15: ' . $calculadoraDePontos->calcularPontosDoPedido($pedido, 15) . " Pontos<br/>";
//Cálculo de pontos na segunda quinzena (dia 16).
echo 'Dia 16: ' . $calculadoraDePontos->calcularPontosDoPedido($pedido, 16) . " Pontos<br/>";
```

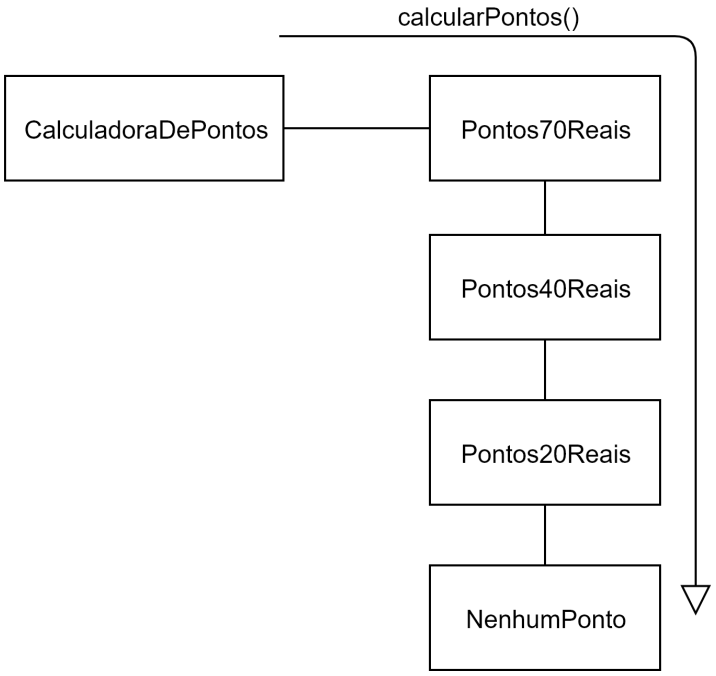
Saída:

```
Dia 15: 2 Pontos
Dia 16: 4 Pontos
-----------------
Dia 15: 20 Pontos
Dia 16: 40 Pontos
```

A princípio pode ser que você pense que não existem grandes problemas na classe **CalculadoraDePontos**. Ela realmente não é uma classe tão grande ou difícil de entender, porém se o dono da hamburgueria decidir tornar seu plano de fidelidade mais complexo, ou então adicionar novos critérios de pontuação, a classe **CalculadoraDePontos** pode crescer de forma descontrolada.

Um dos principais objetivos dos padrões de projeto é facilitar a manutenção de código. Vamos ver como ficaria a solução deste problema utilizando o padrão *Chain of Responsibility*.

Precisamos criar uma cadeia de objetos que calculam pontos, se um objeto não puder calcular ele delega para o próximo objeto calculador de pontos da cadeia.



Cadeia de objetos que podem calcular pontos

A classe **CalculadoraDePontos** não deve conhecer os objetos da cadeia de cálculo de pontos, ela só precisa ter a certeza que eles são capazes de tentar calcular pontos, ou seja, serem capazes de executar o método **calcularPontos()**. A **CalculadoraDePontos** também precisa saber que caso o objeto da cadeia, por qualquer motivo, não consiga calcular os pontos ele possa passar a responsabilidade para outro objeto tentar fazer o cálculo.

Precisamos então de uma interface comum para todos os objetos calculadores de pontos, tal interface irá garantir para a **CalculadoraDePontos** que o objeto recebido por ela sabe calcular pontos e caso o cálculo não seja feito ele passará para o próximo objeto da cadeia de cálculo.

```
interface CalculadorDePontos
{
    //Calcula os pontos gerados pelo pedido.
    public function calcularPontos(Pedido $pedido): int

    //Define o próximo objeto calculador de pontos da cadeia.
    public function setProximo(CalculadorDePontos $proximo): void;
}
```

Vamos agora as classe que calculam os pontos:

```
class Pontos70Reais implements CalculadorDePontos
{
    private CalculadorDePontos $proximoCalculadorDePontos;

    public function calcularPontos(Pedido $pedido): int
    {
        //Se o valor do pedido for maior ou igual a 70 reais.
        if ($pedido->getValor() >= 70) {
            //Retorne o resultado inteiro do valor do pedido dividido por 5.
            return intdiv($pedido->getValor(),  5);
        }

        //Senão chame o método calcularPontos() do próximo calculador de pontos.
        return $this->proximoCalculadorDePontos->calcularPontos($pedido);
    }

    /*Guarda a referência do objeto da cadeia.
      Caso o método calcularPontos() da classe Pontos70Reais não consiga fazer o cálculo.
      o método calcularPontos() de $this->proximaCalculadoraDePontos será chamado.*/
    public function setProximo(CalculadorDePontos $proximo): void
    {
        $this->proximoCalculadorDePontos = $proximo;
    }
}
```

A classe **Pontos70Reais** implementa a interface **CalculadorDePontos** então poderá fazer parte da cadeia, uma vez que é aceita pelo método **setProximo()** da interface **CalculadorDePontos**, e por consequência dos calculadores de pontos concretos.

As classes **Pontos40Reais** e **Pontos20Reais** são semelhantes a classe **Pontos70Reais** diferindo apenas em seu critério de cálculo de pontos.

```
class Pontos40Reais implements CalculadorDePontos
{
    private CalculadorDePontos $proximoCalculadorDePontos;

    public function calcularPontos(Pedido $pedido): int
    {
        if ($pedido->getValor() >= 40) { //Valores maiores que 40 reais.
            return intdiv($pedido->getValor(),  7); //Valor dividido por 7.
        }

        return $this->proximoCalculadorDePontos->calcularPontos($pedido);
    }

    public function setProximo(CalculadorDePontos $proximo): void
    {
        $this->proximoCalculadorDePontos = $proximo;
    }
}
```

```php
class Pontos20Reais implements CalculadorDePontos
{
    private CalculadorDePontos $proximoCalculadorDePontos;

    public function calcularPontos(Pedido $pedido): int
    {
        if ($pedido->getValor() >= 20) { //Valores maiores que 20 reais.
            return intdiv($pedido->getValor(),  10); //Valor dividido por 10.
        }

        return $this->proximoCalculadorDePontos->calcularPontos($pedido);
    }

    public function setProximo(CalculadorDePontos $proximo): void
    {
        $this->proximoCalculadorDePontos = $proximo;
    }
}
```

Para manter o foco no conceito do padrão, as classes calculadoras de pontos do nosso exemplo possuem os métodos `calcularPontos()` muito parecidos. É importante dizer que dependendo do contexto estes métodos podem ser completamente diferentes um do outro, essa é a "mágica" do padrão *Chain of Responsibility*. Poderíamos ter calculadores de pontos que levam em consideração a idade do cliente, o tipo de hambúrguer, forma de pagamento entre outros fatores.

Vamos implementar agora a classe cujo objeto estará no fim da cadeia de calculadores de pontos.

A classe `NenhumPonto` precisa implementar a interface `CalculadorDePontos` para ser aceita na cadeia, porém sua instância sempre será o último objeto da cadeia, então ela não terá um próximo objeto. Repare que o método `setProximo()` existe para manter o contrato da interface mas sua implementação é vazia.

```php
class NenhumPonto implements CalculadorDePontos
{
    public function calcularPontos(Pedido $pedido)
    {
        //Se nenhuma das classes anteriores retornou os pontos então o cliente não pontua.
        return 0;
    }

    public function setProximo(CalculadorDePontos $proximo)
    {
        //Fim da cadeia
    }
}
```

Todos os calculadores de pontos já estão implementados, mas ainda falta o orquestrador da cadeia de processamento. Vejamos como fica a implementação da classe **CalculadoraDePontos** seguindo o padrão *Chain of Responsibility*.

```php
class CalculadoraDePontos
{
    //O método recebe um Pedido e um dia do mês.
    public function calcularPontosDoPedido(Pedido $pedido, int $dia): int
    {
        //Os objetos calculadores de pontos são criados.
        $pontos70 = new Pontos70Reais();
        $pontos40 = new Pontos40Reais();
        $pontos20 = new Pontos20Reais();
        $nenhumPonto = new NenhumPonto();

        //Agora a ordem da cadeia de cálculo de pontos é definida.

        //Se Pontos70Reais não calcular ela passará o cálculo para Pontos40Reais.
        $pontos70->setProximo($pontos40);

        //Se Pontos40Reais não calcular ela passará o cálculo para Pontos20Reais.
        $pontos40->setProximo($pontos20);

        //Se Pontos40Reais não calcular ela passará o cálculo para NenhumPonto.
        //NenhumPonto irá retornar 0 e a cadeia chegará ao fim.
        $pontos20->setProximo($nenhumPonto);

        //Se dia for maior ou igual a 16 e menor ou igual a 31 (Dia da segunda quinzena)
        if ($dia >= 16 && $dia <= 31) {
            //Retorna o cálculo de pontos multiplicado por 2 (dobrado).
            //Repare que o calculo é iniciado em Pontos70Reais que é o início da cadeia.
            return $pontos70->calcularPontos($pedido) * 2;
        }

        //Senão, apenas retorna o cálculo de pontos.
        return $pontos70->calcularPontos($pedido);
    }
}
```

Repetindo o teste:

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

//Criação da calculadora de pontos.
$calculadoraDePontos = new CalculadoraDePontos();

//Definição do valor do pedido igual 21 reais.
$pedido->setValor(21);

//Cálculo de pontos na primeira quinzena (dia 15).
echo 'Dia 15: ' . $calculadoraDePontos->calcularPontosDoPedido($pedido, 15) . " Pontos<br/>";
//Cálculo de pontos na segunda quinzena (dia 16).
echo 'Dia 16: ' . $calculadoraDePontos->calcularPontosDoPedido($pedido, 16) . " Pontos<br/>";

//Apenas para separar os resultados.
echo "---------------------------<br>";

//Mudança do valor do pedido para 100 reais.
$pedido->setValor(100);

//Cálculo de pontos na primeira quinzena (dia 15).
echo 'Dia 15: ' . $calculadoraDePontos->calcularPontosDoPedido($pedido, 15) . " Pontos<br/>";
//Cálculo de pontos na segunda quinzena (dia 16).
echo 'Dia 16: ' . $calculadoraDePontos->calcularPontosDoPedido($pedido, 16) . " Pontos<br/>";
```

Saída:

```
Dia 15: 2 Pontos
Dia 16: 4 Pontos
-----------------
Dia 15: 20 Pontos
Dia 16: 40 Pontos
```

Funcionou, os pontos do valor de 21 reais foi calculado pela classe **Pontos20Reais** que dividiu 21 por 10, a parte inteira do resultado é de 2 pontos, na segunda quinzena foi multiplicado por 2 resultando em 4 pontos. Já os pontos do valor de 100 reais foi calculado pela classe **Pontos70Reais** que dividiu 100 por 5, resultando em 20 e 40 pontos na primeira e segunda quinzena respectivamente.

Agora, caso o dono da hamburgueria decida criar novos critérios de pontuação, sejam eles simples ou complexos, a classe **CalculadoraDePontos** não irá crescer desenfreadamente. Será necessário apenas criar novas classes para cada novo critério e os adicionar na cadeia de processamento. Cada critério ficará encapsulado em sua própria classe facilitando a manutenção e entendimento do código.

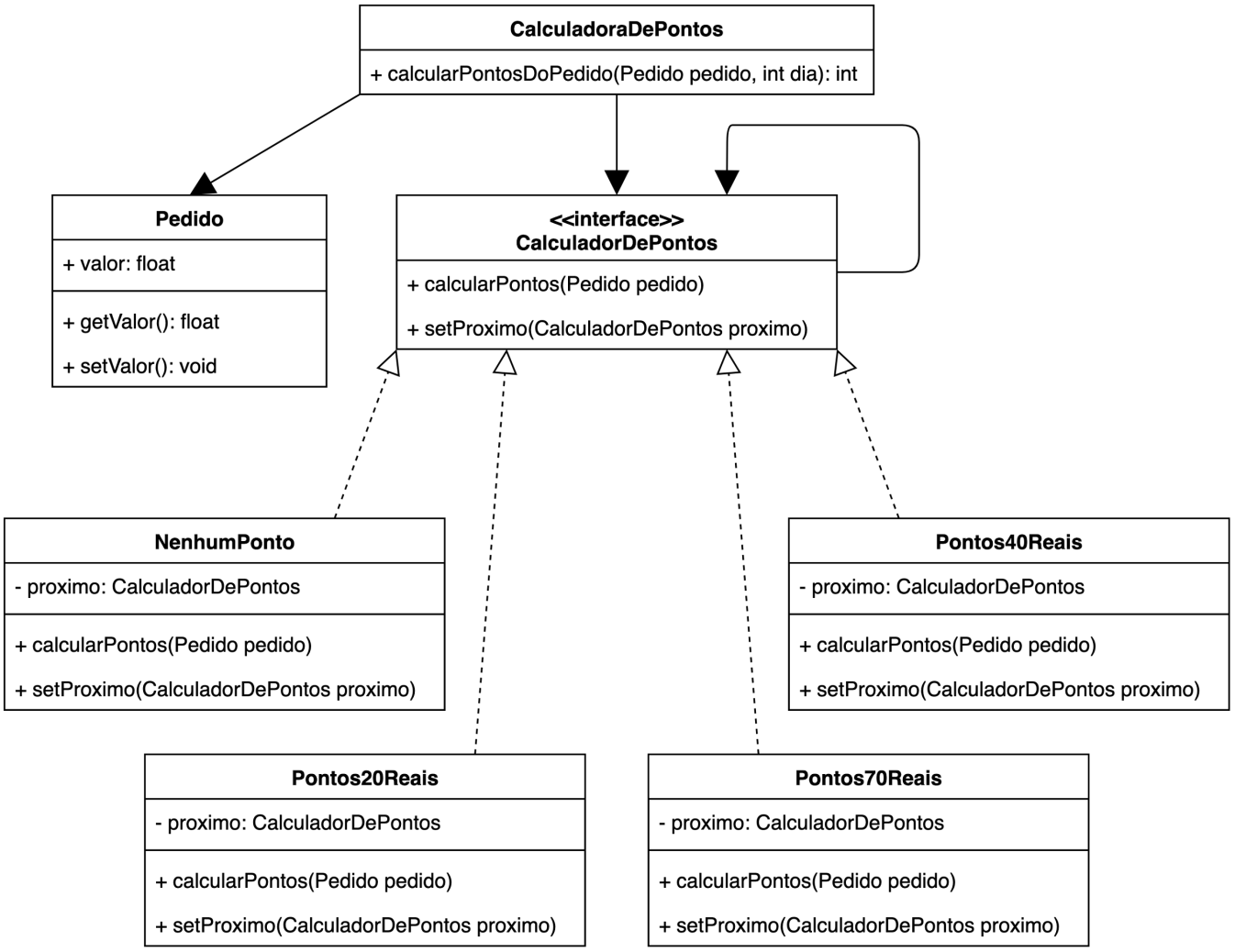


Diagrama de classes completo do exemplo

Aplicabilidade (Quando utilizar?)

- Quando mais de um objeto pode tratar uma solicitação e não se sabe qual objeto fará tal tratamento. O objeto que trata a solicitação deve ser escolhido automaticamente.

- Ao ser necessário fazer uma solicitação para um dentre vários objetos sem especificar explicitamente para qual.

- Quando um conjunto de objetos que pode tratar uma solicitação deve ser especificado dinamicamente.

Componentes

- **Cliente:** Objeto que faz a solicitação.

- **Manipulador:** Define uma interface para tratar solicitações e pode implementar o elo (link) ao seu sucessor.

- **ManipuladorConcreto:** Classes candidatas a atender a solicitação do cliente. Pode acessar seu sucessor, portanto ele tenta atender a solicitação do cliente, caso não consiga, passa tal solicitação ao seu sucessor.

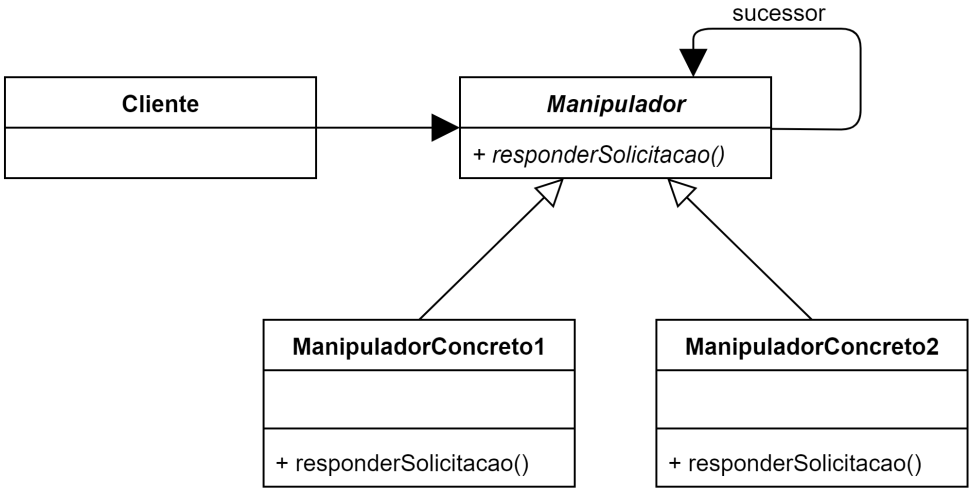


Diagrama de Classes

Consequências

- Redução de acoplamento: O padrão permite que um objeto não precise saber que outro objeto lida com uma solicitação. Precisando saber apenas que uma solicitação será tratada de forma adequada. O receptor e o remetente não têm conhecimento explícito um do outro, e um objeto na cadeia não precisa conhecer toda a estrutura da cadeia. Deste modo a cadeia de responsabilidade simplifica as interconexões de objetos. Onde cada objeto mantém uma única referência, ao seu sucessor, ao invés de manter referência a todos os receptores candidatos.

- Maior flexibilidade na atribuição de responsabilidades aos objetos: A Cadeia de responsabilidade oferece flexibilidade adicional na distribuição de responsabilidades entre objetos. Sendo possível adicionar ou alterar responsabilidades para manipular uma solicitação, adicionando ou alterando a cadeia em tempo de execução.

- A resposta da solicitação não é garantida: Como uma solicitação não possui destinatário explícito, não há garantia de que ela será tratada. A solicitação pode chegar ao final da cadeia e não ser tratada. Uma solicitação também pode não ser tratada quando a cadeia estiver configurada de forma incorreta.