Adapter

Padrões Estruturais

O padrão *Adapter* converte a interface de uma classe para outra interface que o cliente espera encontrar. O Adaptador permite que classes com interfaces incompatíveis trabalhem juntas.

Motivação (Por que utilizar?)

O *Adapter* serve para tornar compatíveis classes que antes não poderiam serem utilizadas em conjunto devido a suas diferenças de interface. Este padrão é utilizado frequentemente em manutenções de códigos legados, muitas vezes novas classes não são compatíveis com as existentes no código, de modo que um cliente não pode as utilizar de forma transparente.

Para tornar essa explicação mais ilustrativa, suponha que trabalhamos na empresa **FreteExpress**, ela possui um serviço de agendamento de fretes por um aplicativo, e faz a cobrança de seus clientes antecipadamente por um *gateway* de pagamentos.

Os serviços da **PagFácil** têm sido utilizados desde a fundação da FreteExpress, porém surgiu um novo fornecedor no mercado, a **TopPagamentos.** Ele cobra uma taxa fixa maior que a PagFácil por cada pagamento, porém cobra juros menores por parcelamentos no cartão de crédito.

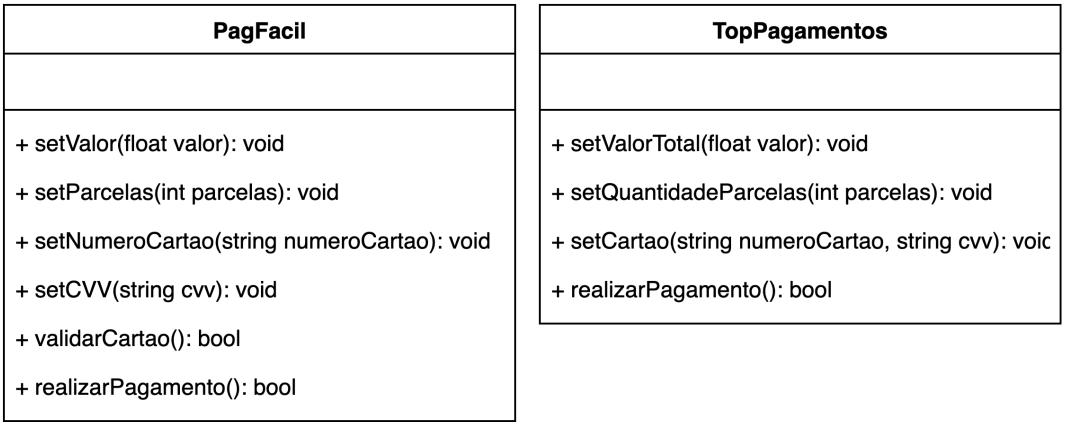
Valores das taxas cobradas pelos *gateways* (Apenas para ilustração)		
	PagFácil	**TopPagamentos**
Taxa fixa por pagamento	R$0,40	R$5,00
Juros ao mês (parcelamento)	5%	1%

A FreteExpress decidiu aderir os serviços da TopPagamentos em conjunto com a PagFácil:

- Para pagamentos à vista deverá ser utilizado o serviço da PagFácil, já que a taxa fixa do pagamento é consideravelmente menor que a da TopPagamentos. Além disso, a taxa de 5% ao mês da PagFácil não será aplicada a pagamentos à vista.

- Já para pagamentos parcelados o serviço da TopPagamentos deverá ser utilizado. Ela cobra uma taxa fixa de R$5,00 mas esse valor se justifica em relação ao baixo juros ao mês (1%) que incide sobre as parcelas.

A documentação dos *gateways* de pagamento diz que as seguintes classes estão disponíveis para seus clientes.



Classes oferecidas pelos *gateways* de pagamentos

Essas classes são fornecidas, então, elas não podem ser modificadas. Nosso código deve utilizá-las da forma que estão.

No momento o projeto que precisamos incrementar se encontra da seguinte forma:

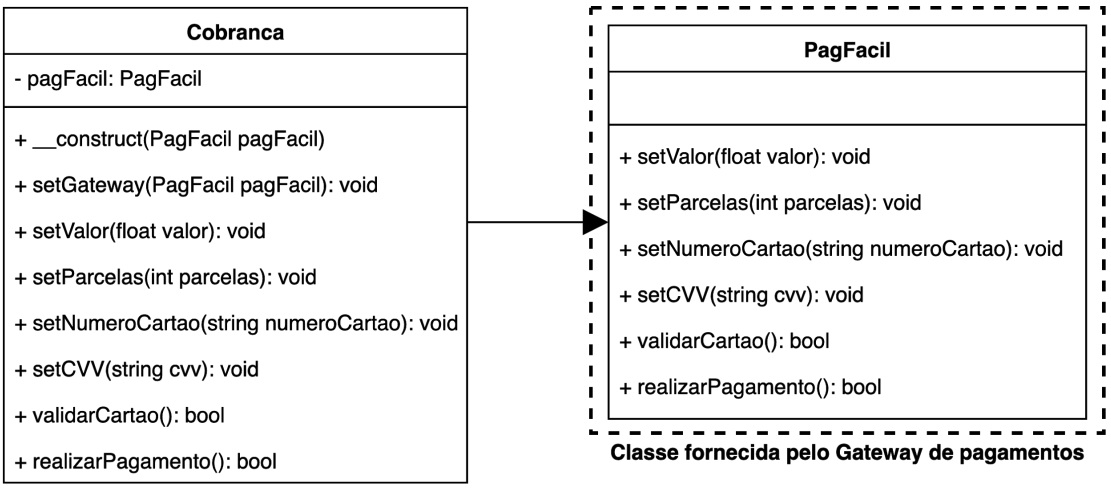


Diagrama de classes do aplicativo da FreteExpress

A classe `Cobranca` é a responsável por solicitar a cobrança dos clientes por meio de um *gateway* de pagamentos. A empresa PagFácil foi o primeiro gateway a ser utilizado, portanto, o projeto da classe `Cobranca` foi baseado na classe `PagFacil`.

No momento a classe `Cobranca` depende diretamente da classe `PagFacil` que é uma classe concreta fornecida por um terceiro. Neste cenário seria mais prudente que a classe `Cobranca` dependesse de uma abstração, como uma interface por exemplo.

Vamos criar uma interface para intermediar a relação entre a classe cobrança e as classes dos *gateways* de pagamentos. Para minimizar os

impactos causados pela refatoração, vamos manter a mesma nomenclatura dos métodos da classe **PagFacil** nos métodos da interface. Vejamos como fica a implementação da interface **Gateway**.

```php
interface Gateway
{
    //Define o valor do pagamento.
    public function setValor(float $valor): void;

    //Define a quantidade de parcelas.
    public function setParcelas(int $parcelas): void;

    //Define o número do cartão.
    public function setNumeroCartao(string $numeroCartao): void;

    //Define o CVV do cartão.
    public function setCVV(string $cvv): void;

    //Verifica se o cartão é válido.
    public function validarCartao(): bool;

    //Realiza o pagamento propriamente dito.
    public function realizarPagamento(): bool;
}
```

Uma vez que a interface for criada, precisamos agora adaptar a classe **Cobranca**. Agora ela deve receber em seu construtor um objeto do tipo **Gateway** ao invés de um do tipo **PagFacil**. Todo o processamento de pagamentos é delegado a esse objeto recebido.

```php
class Cobranca
{
    private Gateway $gateway;

    //Recebe e mantém referência a um objeto do tipo Gateway.
    public function __construct(Gateway $gateway)
    {
        $this->gateway = $gateway;
    }

    //Todos os métodos abaixo delegam responsabilidades para os métodos de $gateway.

    //Agora o parâmetro recebido é do tipo Gateway.
    public function setGateway(Gateway $gateway)
    {
        $this->gateway = $gateway;
    }

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

```
    public function setParcelas(int $parcelas): void
    {
        $this->gateway->setParcelas($parcelas);
    }

    public function setNumeroCartao(string $numeroCartao): void
    {
        $this->gateway->setNumeroCartao($numeroCartao);
    }

    public function setCVV(string $cvv): void
    {
        $this->gateway->setCVV($cvv);
    }

    public function realizarPagamento(): bool
    {
        //Verifica se cartão é válido
        if ($this->gateway->validarCartao()) {
            //retorna se o pagamento foi realizado com sucesso
            return $this->gateway->realizarPagamento();
        }

        return false;
    }
}
```

Já criamos a interface **Gateway** e adaptamos a classe **Cobranca** para aceitar apenas objetos do tipo **Gateway**. Embora já tenha todos os métodos necessários implementados, a classe **PagFacil** agora precisa ser do tipo **Gateway** para ser aceita pela classe **Cobranca**. Não podemos editar a classe **PagFacil**, já que ela é de propriedade do fornecedor, mas podemos criar uma subclasse que herda todas as características de **PagFacil** e ainda implemente a interface **Gateway**, ou seja, faremos isso apenas para cumprir os requisitos de **Cobranca**.

```
class PagFacilAdapter extends PagFacil implements Gateway
{
    //Não é necessário implementar métodos aqui, a classe PagFacil já os implementa (Herança).
    //Repare que os métodos da classe PagFacil são idênticos aos da interface Gateway.
}
```

Antes tínhamos o tipo PagFacil, a classe **PagFacilAdapter** estende este tipo (PagFacil), e passa a implementar a interface **Gateway**. Agora a classe **PagFacilAdapter** já é aceita pela classe **Cobranca**. Vale resaltar que o padrão adapter ainda não foi aplicado em sua totalidade. Na classe **PagFacilAdapter** fizemos apenas uma adaptação de tipo da classe, não foi necessário adaptar os métodos já que eles são os mesmos presentes na interface **Gateway**.

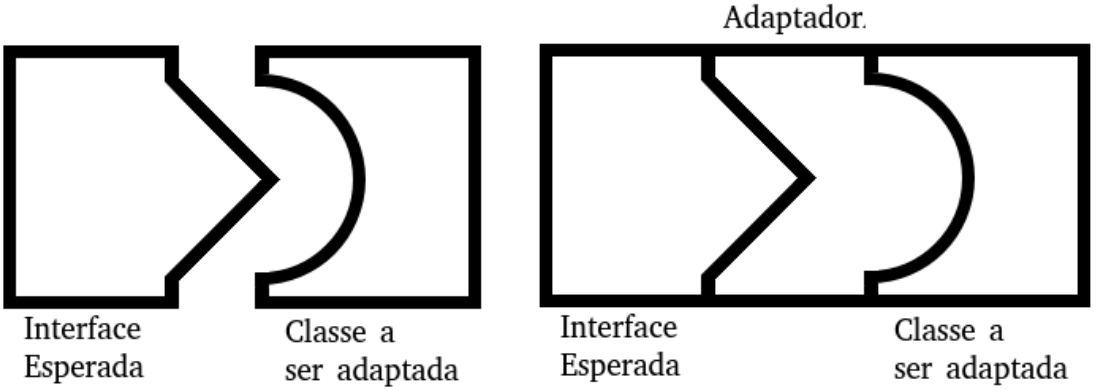
Ainda temos o objetivo de fazer com que um objeto da classe **TopPagamentos** também seja aceito pela classe **Cobranca**, para isso ele precisaria implementar a interface **Gateway**, entretanto os métodos da classe

TopPagamentos não são iguais aos da interface Gateway. Se fizéssemos o mesmo procedimento de PagFacil em TopPagamentos iríamos continuar com o problema, uma vez a subclasse de TopPagamentos teria as mesmas características que ela, ou seja, seus métodos continuariam diferente dos exigidos pela interface Gateway.

Vejamos uma comparação de equivalência entre os métodos da interface Gateway e da classe TopPagamentos:

Gateway	TopPagamentos
setValor($valor);	setValorTotal($valor);
setParcelas($parcelas);	setQuantidadeParcelas($parcelas);
setNumeroCartao($numeroCartao);	Sem Equivalência
setCVV($cvv);	Sem Equivalência
validaCartao();	Sem Equivalência
realizarPagamento();	realizarPagamento();

Repare que a classe TopPagamentos não possui métodos equivalentes aos métodos setNumeroCartao(), setCVV() e ValidaCartao() da interface Gateway. Portanto a classe TopPagamentos também precisa de um adaptador. Mas diferente da classe PagFacil que precisava apenas de uma conversão de interface, a classe TopPagamentos precisa de um trabalho extra de adaptação de seus métodos para atender a interface Gateway.



Esquema ilustrativo de um adapter

Sabemos que a interface alvo é Gateway então o adaptador precisa implementar essa interface. Sabemos também que vamos adaptar a classe TopPagamentos, deste modo devemos manter uma referência a ela.

```php
class TopPagamentosAdapter implements Gateway
{
    private TopPagamentos $topPagamentos;
    private ?string $numeroCartao = null;
    private ?string $cvv = null;

    //Precisamos de uma referência ao objeto que está sendo adaptado.
    public function __construct(TopPagamentos $topPagamentos)
    {
        $this->topPagamentos = $topPagamentos;
    }

    public function setValor(float $valor): void
    {
        //Chama setValorTotal() de TopPagamentos;
        $this->topPagamentos->setValorTotal($valor);
    }

    public function setParcelas(int $parcelas): void
    {
        //Chama setQuantidadeParcelas() de TopPagamentos;
        $this->topPagamentos->setQuantidadeParcelas($parcelas);
    }

    public function setNumeroCartao(string $numeroCartao): void
    {
        //TopPagamento não possui o método setNumeroCartao();
        //Então vamos guardar o número do cartão em uma variável local.
        $this->numeroCartao = $numeroCartao;

        //Sabemos que o número do cartão já foi definido.
        //Se o CVV também já foi definido. Chama o método setCartao().
        if ($this->cvv != null) {
            $this->topPagamentos->setCartao($this->numeroCartao, $this->cvv);
        }
    }

    public function setCVV(string $cvv): void
    {
        //TopPagamento não possui o método setCVV();
        //Então vamos guardar o CVV em uma variável local.
        $this->cvv = $cvv;

        //Sabemos que o CVV já foi definido.
        //Se o número do cartão também já foi definido. Chama o método setCartao().
        if ($this->numeroCartao != null) {
            $this->topPagamentos->setCartao($this->numeroCartao, $this->cvv);
        }
    }

    public function validarCartao(): bool
    {
        return true; //Sempre retorna true. Esse método não existe em TopPagamentos.
    }

    public function realizarPagamento(): bool
    {
        //Chama realizarPagamento() de TopPagamentos;
        return $this->topPagamentos->realizarPagamento();
    }
}
```

Agora já temos um adaptador para a classe TopPagamentos. Vamos ver como utilizar este adaptador.

```php
//==== Classes dos fornecedores ====

//E uma instância da classe de TopPagamentos.
$topPagamentos = new TopPagamentos();

//==== Adapters ====
//Criação do adaptador de PagFacil.
$pagFacilAdapter = new PagFacilAdapter();

//Criação do adaptador de TopPagamentos.
$topPagamentosAdapter = new TopPagamentosAdapter($topPagamentos);

//==== Cobranca ====
echo 'Cobrança utilizando PagFacil como Gateway <br>';

//Criação de uma Cobrança utilidando a classe PagFacil.
//Repare que o adaptador de $pagFacil é passado para o construtor.
$cobranca = new Cobranca($pagFacilAdapter);

$cobranca->setValor(100);
$cobranca->setParcelas(3);
$cobranca->setNumeroCartao(1234123412341234);
$cobranca->setCVV(123);

if ($cobranca->realizarPagamento()) {
    echo 'Pagamento Realizado com sucesso <br>';
} else {
    echo 'O pagamento falhou <br>';
}

//Cobrança utilidando a classe PagFacil.
echo '<br>Cobrança utilizando TopPagamentos como Gateway <br>';

//Troca do Gateway de Cobrança para TopPagamentos
$cobranca->setGateway($topPagamentosAdapter);

$cobranca->setValor(100);
$cobranca->setParcelas(3);
$cobranca->setNumeroCartao(1234123412341234);
$cobranca->setCVV(123);

if ($cobranca->realizarPagamento()) {
    echo "Pagamento Realizado com sucesso";
} else {
    echo "O pagamento falhou";
}
```

Saída:

```
Cobrança utilizando PagFacil como Gateway
Pagamento Realizado com sucesso

Cobrança utilizando TopPagamentos como Gateway
Pagamento Realizado com sucesso
```

Repare que no teste que a utilização das classes `PagFacil` de `TopPagamentos` ficou padronizada, a classe `Cobranca` utiliza as classes `PagFacilAdapter` e `TopPagamentosAdapter` como classes intermediárias que permitem tal padronização. Como resultado disso, a utilização da classe `Cobranca` é transparente para o cliente, ele não precisa se preocupar em utilizar métodos diferentes de acordo com o *Gateway* de pagamento que está sendo utilizado por `Cobranca`.

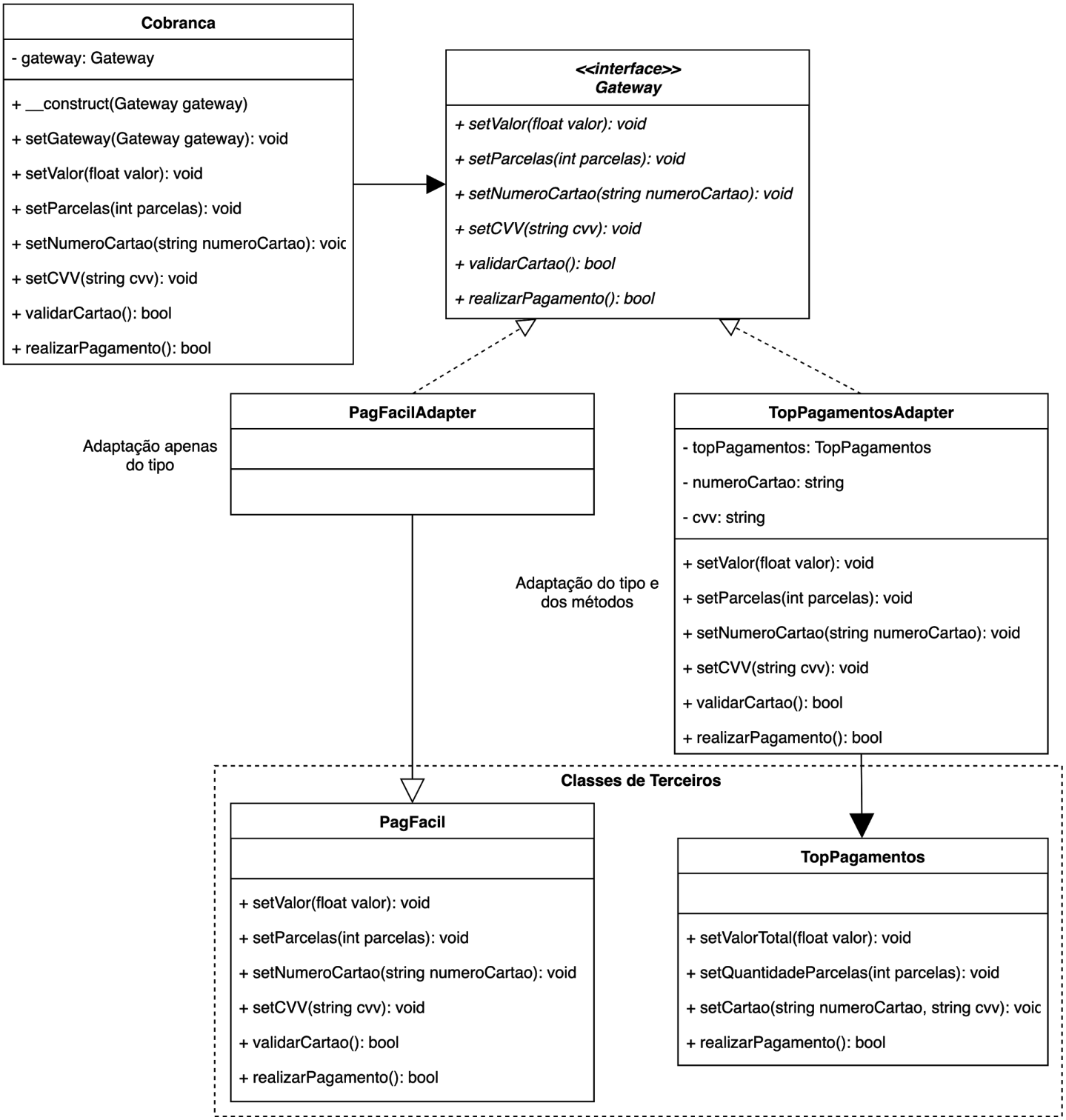


Diagrama de classes do aplicativo da FreteExpress após aplicação do adapter

Aplicabilidade (Quando utilizar?)

- Quando existe a necessidade de utilizar uma classe existente e sua interface é diferente da esperada.

- Quando se deseja criar uma classe reutilizável que coopera com classes não relacionadas a ela ou que não foram previstas, ou seja, classes que não necessariamente têm interfaces compatíveis.

- (Somente para adaptadores de objeto) Quando é necessário usar várias subclasses existentes, mas é impraticável adaptar sua interface sub-classificando cada uma delas. Um adaptador de objeto pode adaptar a interface de sua superclasse.

Componentes

- **Cliente:** É a classe que espera a interface alvo.

- **Alvo:** Interface esperada pelo Cliente. Deve ser implementada pelo Adapter.

- **Adapter:** Converte a interface de Adaptado para a interface Alvo. Delega todas as solicitações para Adaptado.

- **Adaptado:** Classe que possui interface incompatível com o cliente e por isso precisa ser adaptada.

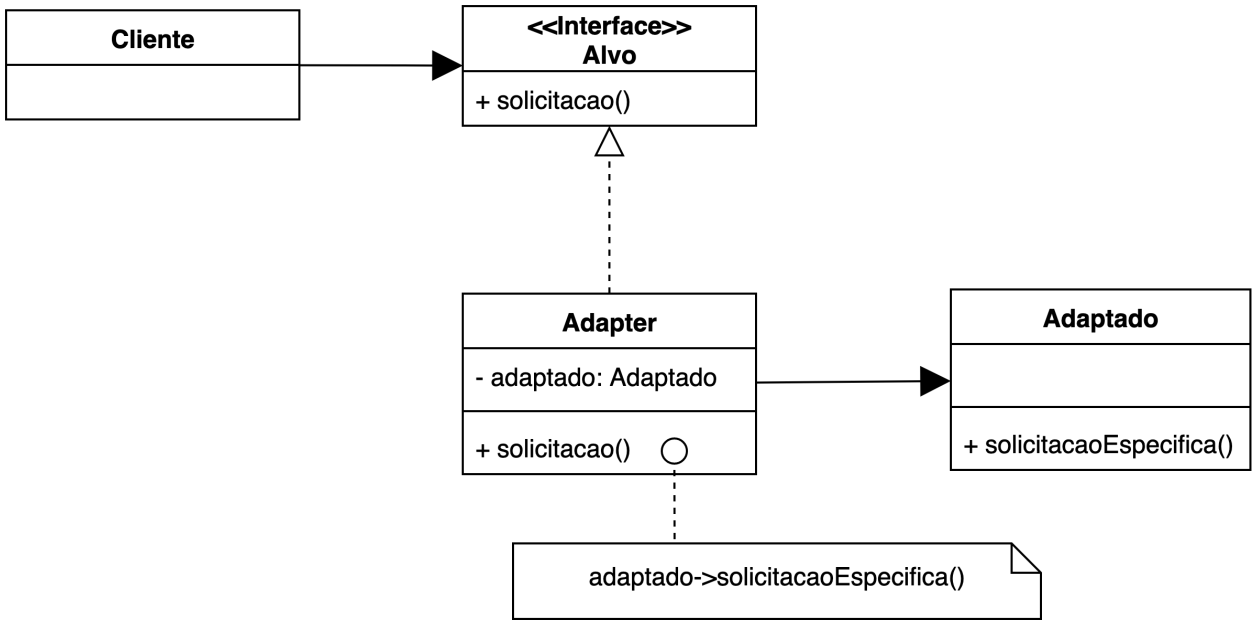


Diagrama de Classes (Adaptador de Objetos)

Consequências

- A quantidade de trabalho que o Adapter faz depende de quão semelhante a interface do Alvo é do Adaptado. Há uma variação no volume de trabalhos que executar para adaptar um classe para a interface alvo, vai desde a simples conversão de interface, por exemplo, alterando os nomes das solicitações, até o suporte a um conjunto de solicitações totalmente diferente.

- Uma classe é mais reutilizável quanto menor for a quantidade de suposições que outras classes devem fazer para usá-la, o *Adapter* elimina tais suposições. Em outras palavras, um adaptador de interface permite incorporar uma classe em sistemas existentes que podem esperar interfaces diferentes para a classe.

- Um problema em potencial com adaptadores é que eles não são transparentes para todos os clientes. Um objeto adaptado não oferece a interface do objeto original, portanto, ele não pode ser usado onde o objeto original é esperado. Adaptadores bidirecionais podem fornecer essa transparência. Especificamente, são úteis quando dois clientes diferentes precisam exibir um objeto de maneira diferente.

  Para criar um adaptador bidirecional que atua simultaneamente como uma interface antiga e uma interface nova, basta que ele implemente ambas as interfaces envolvidas. Em alguns cenários é preciso utilizar herança múltipla.

- Até agora falamos apenas sobre **adaptadores de objeto**, inclusive o diagrama de classe da sessão "Componentes" representa um adaptador desse tipo. Entretanto existe outro tipo, que são os **adaptadores de classes**, cuja implementação requer herança múltipla e este recurso não está disponível no PHP até sua versão atual (7.4).

  Neste caso, ao invés de usar a composição para adaptar o Adaptado, o Adaptador utiliza subclasses das classes adaptada e classe Alvo.

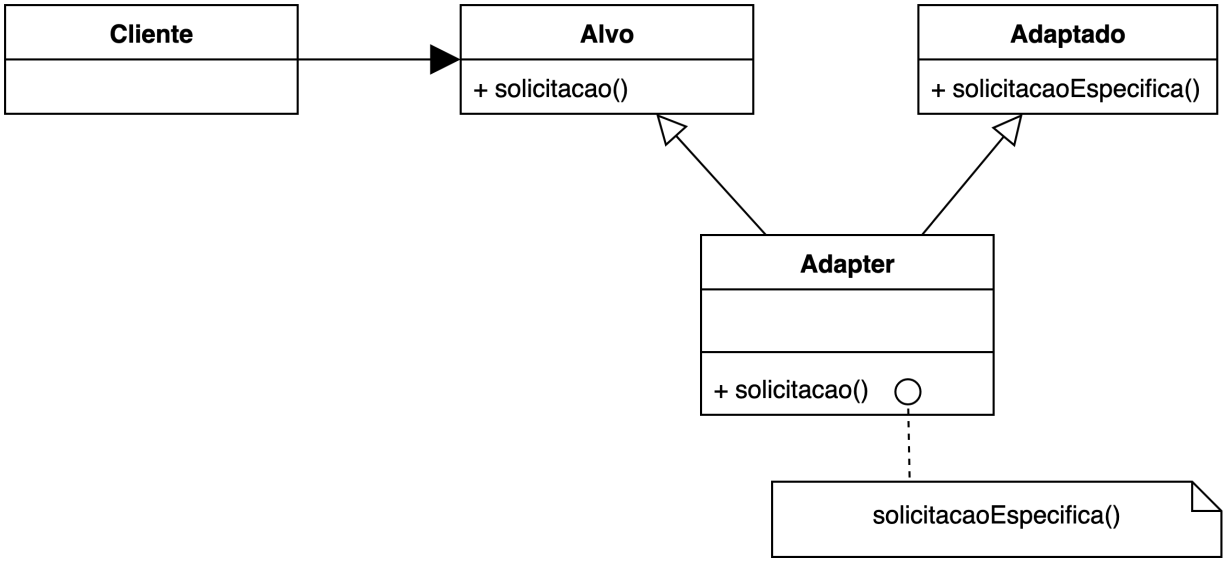


Diagrama de Classes (Adaptador de Classes)

- Adaptadores de objetos e de classes causam diferentes consequências.
    - Um adaptador de Objetos (Por composição):
        - Permite que um único adaptador funcione com muitos adaptados, ou seja, o próprio adaptado e todas as suas subclasses (se houver). O adaptador também pode adicionar funcionalidade a todos os adaptados de uma só vez.
        - Dificulta sobrescrever (*override*) o comportamento de Adaptado. Isso exigirá uma sub-classificação de Adaptado e fará com que o Adaptador se refira à subclasse em vez do próprio Adaptado.
    - Um adaptador de Classes (Por herança):
        - Adapta a classe Adaptado ao Alvo comprometendo-se com uma classe concreta Adapter. Como consequência, um adaptador de classe não funcionará quando queremos adaptar uma classe e todas as suas subclasses.
        - Permite que o Adapter substitua parte do comportamento do Adaptado, pois o Adapter é uma subclasse Adaptado.
        - Introduz apenas um objeto, e não é necessário nenhum direcionamento adicional do ponteiro (manter referência) para a classe Adaptado.