Licensed to Farley de Souza Rufino - farley.t.i@hotmail.com - 114.052.636-79

Template Method

Padrões Comportamentais

O padrão *Template Method* define o esqueleto de um algoritmo dentro de um método, transferindo alguns de seus passos para subclasses. O *Template Method* permite que as subclasses redefinam certos passos de um algoritmo sem alterar a estrutura do mesmo.

Motivação (Por que utilizar?)

O padrão *Template Method* auxilia na definição de um algoritmo que contém algumas de suas partes definidas por métodos abstratos. Subclasses são responsáveis por implementar as partes abstratas deste algoritmo. Tais partes poderão ser implementadas de formas distintas, ou seja, cada subclasse irá implementar conforme sua necessidade. Deste modo a superclasse posterga algumas implementações para que sejam feitas por suas subclasses.

Este padrão ajuda na reutilização de código e no controle de como o código deve ser executado.

Para exemplificar considere o módulo de pagamentos do *software* de uma loja de confecções, este módulo foi desenvolvido a alguns anos atrás e possui as seguintes classes.

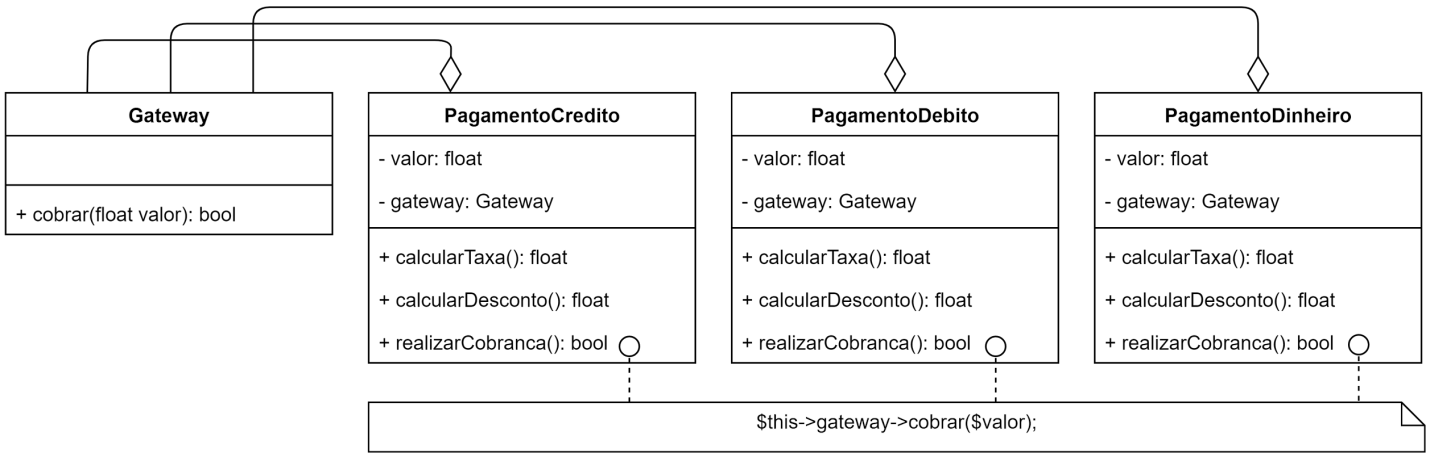

Diagrama de classes do exemplo sem *Template Method*

As regras da loja para pagamentos são:

- Taxa
  - Crédito - 5% sob o valor;
  - Débito - Acrescentar o custo fixo de 4 reais sob o valor.
  - Dinheiro - Sem taxa.
- Desconto:
  - Crédito - 2% somente sob o valores maiores que 300 reais.
  - Débito - 5% sob o valor.

Licensed to Farley de Souza Rufino - farley.t.i@hotmail.com - 114.052.636-79

- Dinheiro - 10% sob o valor.

A taxa é referente à cobrança feita pelo *gateway* de pagamentos utilizado pelo *software* da loja para realizar as cobranças. O método **realizarCobranca()** presente nas três classes é o responsável por delegar a cobrança ao serviço do *gateway*.

Essa é a classe que utilizaremos para **simular** o **Gateway** de pagamentos. O método **cobrar()** retorna **true** ou **false** de forma aleatória.

```php
class Gateway
{
    //Simulação de uma cobrança - retorna 'true' ou 'false' de forma randômica.
    public function cobrar(float $valor): bool
    {
        echo 'R$ ' . $valor . '<br>'; //Apenas para visualizar o valor final.
        $respostas = [true, false];
        return $respostas[rand(0, 1)];
    }
}
```

Vejamos como as classes foram previamente implementadas:

```php
class PagamentoCredito
{
    private float $valor;
    private Gateway $gateway;

    public function __construct(float $valor, Gateway $gateway)
    {
        $this->valor = $valor;
        $this->gateway = $gateway;
    }

    //Calcula a taxa do Gateway.
    public function calcularTaxa(): float
    {
        return $this->valor * 0.05; //Retorna uma taxa de 5% o valor do pagamento.
    }

    //Calcula o desconto.
    public function calcularDesconto(): float
    {
        //Se o valor pago for maior que 300 reais.
        if ($this->valor > 300) {
            return $this->valor * 0.02; //Retorna um desconto de 2% sob o valor do pagamento.
        }
        return 0; //Pagamentos de valores menores que 300 reais não possuem desconto.
    }

    //Realiza a cobrança.
    public function realizaCobranca(): bool
    {
        //Calcula o valor total (Valor do pagamento + taxa - desconto).
        $valorFinal = $this->valor + $this->calcularTaxa() - $this->calcularDesconto();
        return $this->gateway->cobrar($valorFinal); //Delega a cobrança para o Gateway.
    }
}
```

Licensed to Farley de Souza Rufino - farley.t.i@hotmail.com - 114.052.636-79

```php
class PagamentoDebito
{
    private float $valor;
    private Gateway $gateway;

    public function __construct(float $valor, Gateway $gateway)
    {
        $this->valor = $valor;
        $this->gateway = $gateway;
    }

    //Calcula a taxa do Gateway.
    public function calcularTaxa(): float
    {
        return 4; //Retorna uma taxa de 4 reais.
    }

    //Calcula o desconto.
    public function calcularDesconto(): float
    {
        return $this->valor * 0.05; //Retorna o valor do pagamento com desconto de 5%.
    }

    public function realizaCobranca(): bool
    {
        $valorFinal = $this->valor + $this->calcularTaxa() - $this->calcularDesconto();
        return $this->gateway->cobrar($valorFinal);
    }
}
```

```php
class PagamentoDinheiro
{
    private float $valor;
    private Gateway $gateway;

    public function __construct(float $valor, Gateway $gateway)
    {
        $this->valor = $valor;
        $this->gateway = $gateway;
    }

    //Calcula a taxa do Gateway.
    public function calcularTaxa(): float
    {
        return 0; //Pagamento em dinheiro não possui taxa.
    }

    //Calcula o desconto.
    public function calcularDesconto(): float
    {
        return $this->valor * 0.1; //Retorna o valor do pagamento com desconto de 10%.
    }

    public function realizaCobranca(): bool
    {
        $valorFinal = $this->valor + $this->calcularTaxa() - $this->calcularDesconto();
        return $this->gateway->cobrar($valorFinal);
    }
}
```

Licensed to Farley de Souza Rufino - farley.t.i@hotmail.com - 114.052.636-79

Testando o código acima :

```
$valor = 1000; //Definição do valor do pagamento.
$gateway = new Gateway(); //Criação de uma instância de Gateway.

//Pagamento Crédito.
echo 'Crédito: ';
$pagamentoCredito = new PagamentoCredito($valor, $gateway);
$pagamentoCredito->realizaCobranca();

//Pagamento Débito.
echo 'Débito: ';
$pagamentoDebito = new PagamentoDebito($valor, $gateway);
$pagamentoDebito->realizaCobranca();

//Pagamento Dinheiro.
echo 'Dinheiro: ';
$pagamentoDinheiro = new PagamentoDinheiro($valor, $gateway);
$pagamentoDinheiro->realizaCobranca();
```

Saída:

```
Crédito: 1030
Débito: 954
Dinheiro: 900
```

O dono da loja de confecções está modernizando a loja e deseja aceitar novas formas de pagamento no futuro. Nossa tarefa é refatorar o módulo de pagamentos de modo que ele seja apto a aceitar novas formas de pagamento de maneira segura, sem afetar as formas de pagamentos já existentes, e minimizando as chances de surgimento de *Bugs*.

É possível encontrar características comuns nas classes acima:

- Todas elas têm o método **realizarCobranca()** idêntico.
- Todas possuem um método para cálculo de taxas, mas são diferentes entre si.
- Todas possuem um método para cálculo de desconto mas são diferentes entre si.
- Todas possuem uma variável de instância **$valor**.
- Todas mantêm uma referência a um objeto **Gateway**.

O método **realizarCobranca()** é quem dita como a cobrança será feita, é nele onde o algoritmo está implementado. Ele utiliza os métodos **calcularTaxa()** e **calcularDesconto()** que são a única variação entre uma classe e outra.

Licensed to Farley de Souza Rufino - farley.t.i@hotmail.com - 114.052.636-79

Vamos transformar o método **realizarCobranca()** em nosso *Template Method* (método de gabarito/modelo) e migrar tudo que é comum entre as classes para uma superclasse **Pagamento**.

```php
abstract class Pagamento
{
    protected float $valor;
    protected Gateway $gateway;

    public function __construct(float $valor, Gateway $gateway)
    {
        $this->valor = $valor;
        $this->gateway = $gateway;
    }

    //Hook (gancho) - Implementação Mínima. Pode ser sobreescrito pelas subclasses.
    public function calcularTaxa(): float
    {
        return 0;
    }

    //Será implementado pelas subclasses.
    abstract public function calcularDesconto();

    final public function realizaCobranca(): bool
    {
        $valorFinal = $this->valor + $this->calcularTaxa() - $this->calcularDesconto();

        //-- Início da impressão dos valores separados (Apenas para visualização)
        echo $this->valor . ' + ' .
            $this->calcularTaxa() . ' - ' .
            $this->calcularDesconto() . ' = ';
        //-- Fim da impressão dos valores separados

        return $this->gateway->cobrar($valorFinal);
    }
}
```

Existem alguns pontos importantes a serem observados na classe **Pagamento**:

- A variável de instância **$valor** e referência a **Gateway** ficam agora na superclasse **Pagamento**. Todas as suas subclasses já terão esses recursos disponíveis.

- O método **realizaCobranca()** foi declarado como **final**, isso garante que as subclasses de **Pagamento** não poderão sobrescrever este método e a realização da cobrança será feita conforme as regras da superclasse, ele é o *Template Method*.

- O método **calcularDesconto()** foi declarado como abstrato, deste modo é responsabilidade das subclasses o implementar conforme suas regras específicas.

Licensed to Farley de Souza Rufino - farley.t.i@hotmail.com - 114.052.636-79

- O método **calcularTaxa()** é um *hook* (gancho). Trata-se de um método que é implementado na classe abstrata mas recebe apenas uma implementação vazia, ou mínima possível como padrão. No nosso exemplo isso é útil devido ao fato que a classe **PagamentoDinheiro** não tem incidência de cobrança de taxa, assim ela pode utilizar a implementação padrão do método que retorna 0. Já as classes **PagamentoCredito** e **PagamentoDebito** deverão sobrescrever este método conforme suas regras específicas.

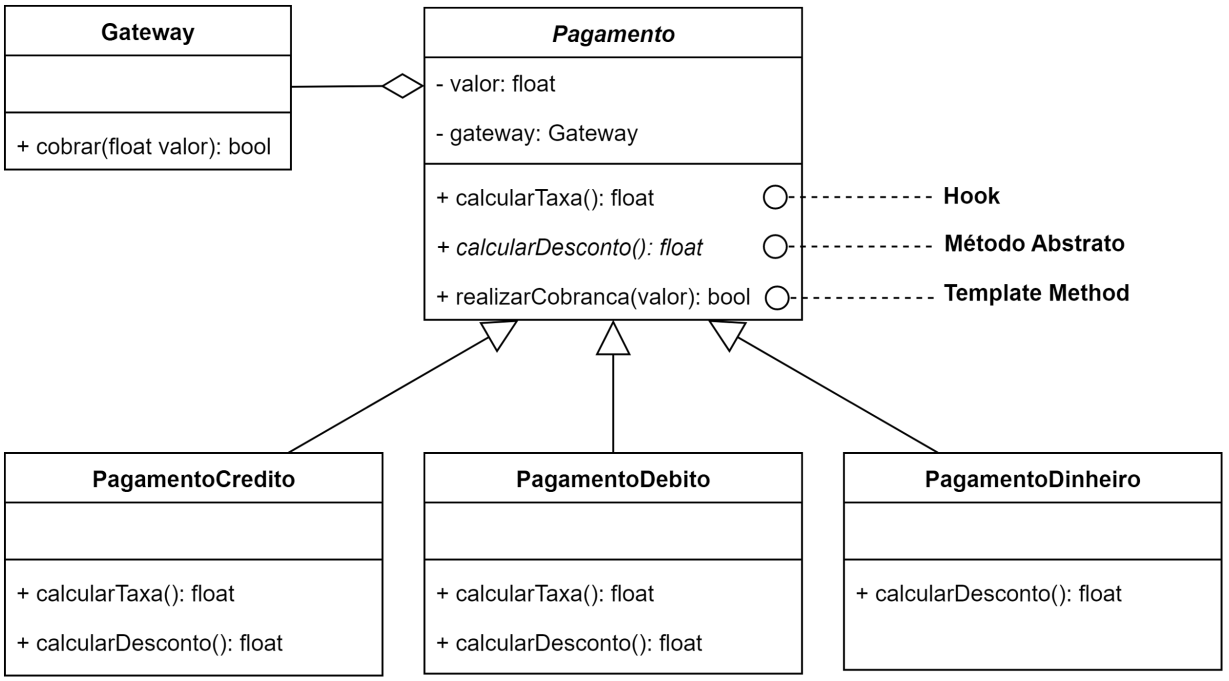

Diagrama de classes do exemplo com *Template Method*

Quem dita as regras a respeito de como uma cobrança será feita é o método **realizaCobranca()** da superclasse **Pagamento**. Cabe às subclasses completar as peças do quebra-cabeças.

```
class PagamentoCredito extends Pagamento
{
    //Calcula a taxa do Gateway.
    public function calcularTaxa(): float
    {
        //Retorna uma taxa de 5% o valor do pagamento.
        return $this->valor * 0.05;
    }

    //Calcula o desconto.
    public function calcularDesconto(): float
    {
        //Se o valor pago for maior que 300 reais.
        if ($this->valor > 300) {
            //Retorna um desconto de 2% sob o valor do pagamento.
            return $this->valor * 0.02;
        }
        //Pagamentos de valores menores que 300 reais não possuem desconto.
        return 0;
    }
}
```

Licensed to Farley de Souza Rufino - farley.t.i@hotmail.com - 114.052.636-79

```php
class PagamentoDebito extends Pagamento
{
    //Calcula a taxa do Gateway.
    public function calcularTaxa(): float
    {
        //Retorna uma taxa de 4 reais.
        return 4;
    }

    //Calcula o desconto.
    public function calcularDesconto(): float
    {
        //Retorna o valor do pagamento com desconto de 5%.
        return $this->valor * 0.05;
    }
}
```

```php
class PagamentoDinheiro extends Pagamento
{
    //Calcula o desconto.
    public function calcularDesconto(): float
    {
        //Retorna o valor do pagamento com desconto de 10%.
        return $this->valor * 0.1;
    }
}
```

Testando o código com *Template Method:*

```php
$valor = 1000; //Definição do valor do pagamento.
$gateway = new Gateway(); //Criação de uma instância de Gateway.

//Pagamento Crédito.
echo 'Crédito: ';
$pagamentoCredito = new PagamentoCredito($valor, $gateway);
$pagamentoCredito->realizaCobranca();

//Pagamento Débito.
echo 'Débito: ';
$pagamentoDebito = new PagamentoDebito($valor, $gateway);
$pagamentoDebito->realizaCobranca();

//Pagamento Dinheiro.
echo 'Dinheiro: ';
$pagamentoDinheiro = new PagamentoDinheiro($valor, $gateway);
$pagamentoDinheiro->realizaCobranca();
```

Saída:

```
Crédito: 1000 + 50 - 20 = 1030
Débito: 1000 + 4 - 50 = 954
Dinheiro: 1000 + 0 - 100 = 900
```

Licensed to Farley de Souza Rufino - farley.t.i@hotmail.com - 114.052.636-79

Na nova implementação não existe mais código repetido, as subclasses complementam a classe pai com os comportamentos que variam. Isso causa uma inversão de dependência que diz *"Abstrações não devem ser baseadas em detalhes. Detalhes devem ser baseados em abstrações"*. É exatamente o que estamos fazendo, pois os detalhes de cada tipo de pagamento dependem da classe abstrata Pagamento e não o contrário.

Se no futuro a loja decidir adotar um novo meio de pagamento, como pagamento via *smartphone* por exemplo, basta criar uma nova classe PagamentoSmatphone, estender a classe Pagamento e implementar suas especificidades. Deste modo não será necessário modificar a classe Pagamento que está seguindo outro princípio de Orientação a objetos, ela está *"aberta para extensão mas fechada para mudanças"*.

Aplicabilidade (Quando utilizar?)

- Para implementar partes invariantes de um algoritmo apenas uma vez, deixando para as subclasses apenas a implementação daquilo que pode variar.

- Controlar extensões de subclasses, sabendo o que as subclasses devem implementar e até onde devem implementar.

- Evitar duplicação de código entre classes comuns.

Componentes

- **ClasseAbstrata:** Superclasse abstrata que contém os métodos concretos e abstratos que serão comuns a todas suas subclasses. Implementa o templateMethod() que define o esqueleto de um algoritmo.

- **ClasseConcreta:** Classes que herdam os métodos concretos de ClasseAbstrata e implementam os métodos abstratos conforme suas especificidades.

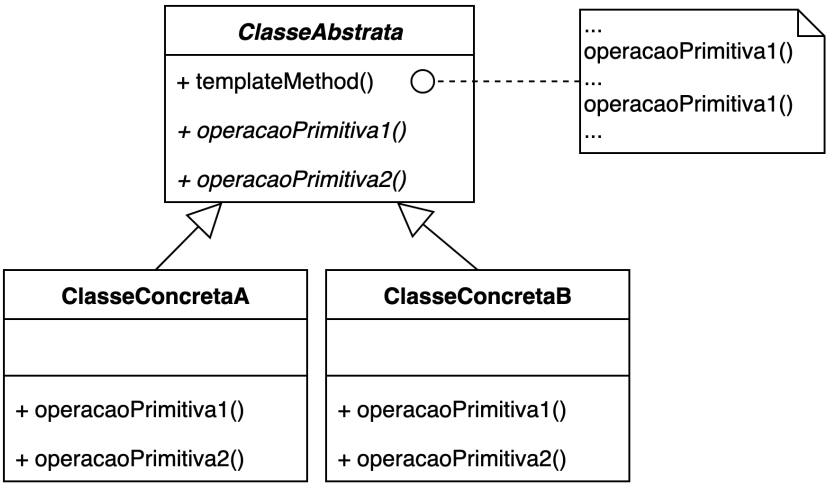

Diagrama de Classes

Licensed to Farley de Souza Rufino - farley.t.i@hotmail.com - 114.052.636-79

Consequências

Os T*emplates Methods*:

- São uma técnica fundamental para a reutilização de código. São particularmente importantes em bibliotecas de classes, pois são os meios para definir o comportamento comum nas classes das bibliotecas.
- Proporcionam a inversão de dependência. Isso se refere a como uma classe pai chama as operações de uma subclasse e não o contrário.
- Permitem controlar a sequência da execução de métodos das subclasses.
- Possibilitam ter pontos que chamam código ainda não implementado.
- Podem chamar os seguintes tipos de operações:
    - Métodos Concretos: implementados na própria classe abstrata onde o *Template Method* se encontra.
    - Métodos Abstratos: implementados nas subclasses.
    - Operações primitivas e funções da linguagem.
    - Outros *Template Methods*.
    - Hooks.