Template Method

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.

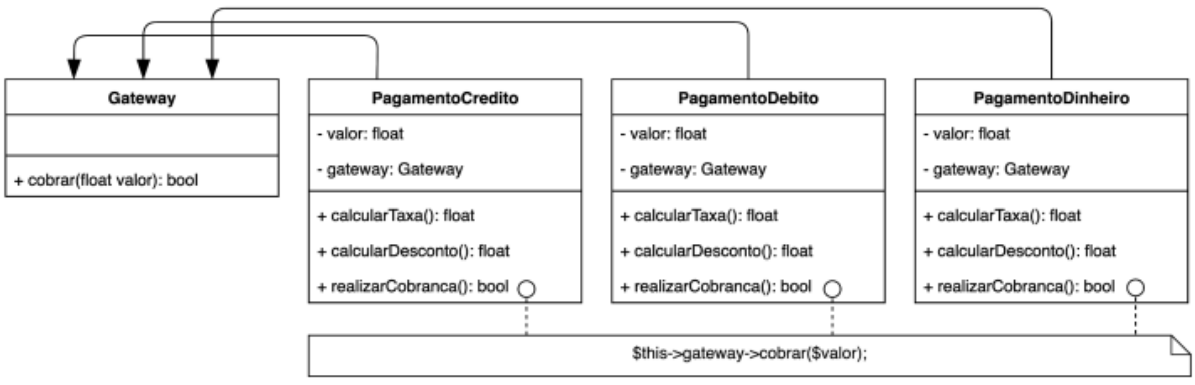

Diagrama de classes do exemplo sem *Template Method*

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
  - 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.

```
export class Gateway {

  collect(value: number): boolean {

    const valueFormated = Intl.NumberFormat('pt-br', {
      currency: 'BRL',
      style: 'currency'
    }).format(value)

    console.log(valueFormated)

    const response = [true, false]
    return response[Math.round(Math.random())]
  }

}
```

Vejamos como as classes foram previamente implementadas:

```
import { Gateway } from "./gateway"

export class CreditPayment {

  constructor(private value: number,  private gateway: Gateway) {
    this.value = value
    this.gateway = gateway
  }

  calculateRate(): number {
    return this.value * 0.05
  }

  calculateDiscount(): number {
    if (this.value > 300) {
      return this.value * 0.02
    }
    return 0
  }

  applyCollect(): boolean {
    const value = this.value + this.calculateRate() - this.calculateDiscount()
    return this.gateway.collect(value)
  }

}
```

```
import { Gateway } from "./gateway"

export class DebitPayment {

  constructor(private value: number,  private gateway: Gateway) {
    this.value = value
    this.gateway = gateway
  }

  calculateRate(): number {
    return 4
  }

  calculateDiscount(): number {
      return this.value * 0.05
  }

  applyCollect(): boolean {
    const value = this.value + this.calculateRate() - this.calculateDiscount()
    return this.gateway.collect(value)
  }

}
```

```
import { Gateway } from "./gateway"

export class MoneyPayment {

  constructor(private value: number,  private gateway: Gateway) {
    this.value = value
    this.gateway = gateway
  }

  calculateRate(): number {
    return 0
  }

  calculateDiscount(): number {
      return this.value * 0.1
  }

  applyCollect(): boolean {
    const value = this.value + this.calculateRate() - this.calculateDiscount()
    return this.gateway.collect(value)
  }

}
```

Testando o código acima :

```
import { CreditPayment } from "./class/credit-payment"
import { DebitPayment } from "./class/debit-payment"
import { Gateway } from "./class/gateway"
import { MoneyPayment } from "./class/money-payment"

const value = 1000
const gateway = new Gateway()

const creditPayment = new CreditPayment(value, gateway)
console.log('Credit: ')
creditPayment.applyCollect()

const debitPayment = new DebitPayment(value, gateway)
console.log('Debit: ')
debitPayment.applyCollect()

const moneyPayment = new MoneyPayment(value, gateway)
console.log('Money: ')
moneyPayment.applyCollect()
```

Saída:

```
Credit:
R$ 1.030,00

Debit:
R$ 954,00

Money:
R$ 900,00
```

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.

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`.

```
export interface IGateway {
  collect(value: number): boolean
```

```
}
```

```
import { IGateway } from "../interface/gateway"
export class Gateway implements IGateway {

  collect(value: number): boolean {

    const valueFormated = Intl.NumberFormat('pt-br', {
      currency: 'BRL',
      style: 'currency'
    }).format(value)

    console.log(valueFormated)

    const response = [true, false]
    return response[Math.round(Math.random())]
  }

}
```

```
import { IGateway } from "../interface/gateway"

export abstract class Payment {

  constructor(protected value: number,  protected gateway: IGateway) {
    this.value = value
    this.gateway = gateway
  }

  calculateRate(): number {
    return 0
  }

  abstract calculateDiscount(): number

  applyCollect(): boolean {
    const value = this.value + this.calculateRate() - this.calculateDiscount()
    return this.gateway.collect(value)
  }

}
```

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

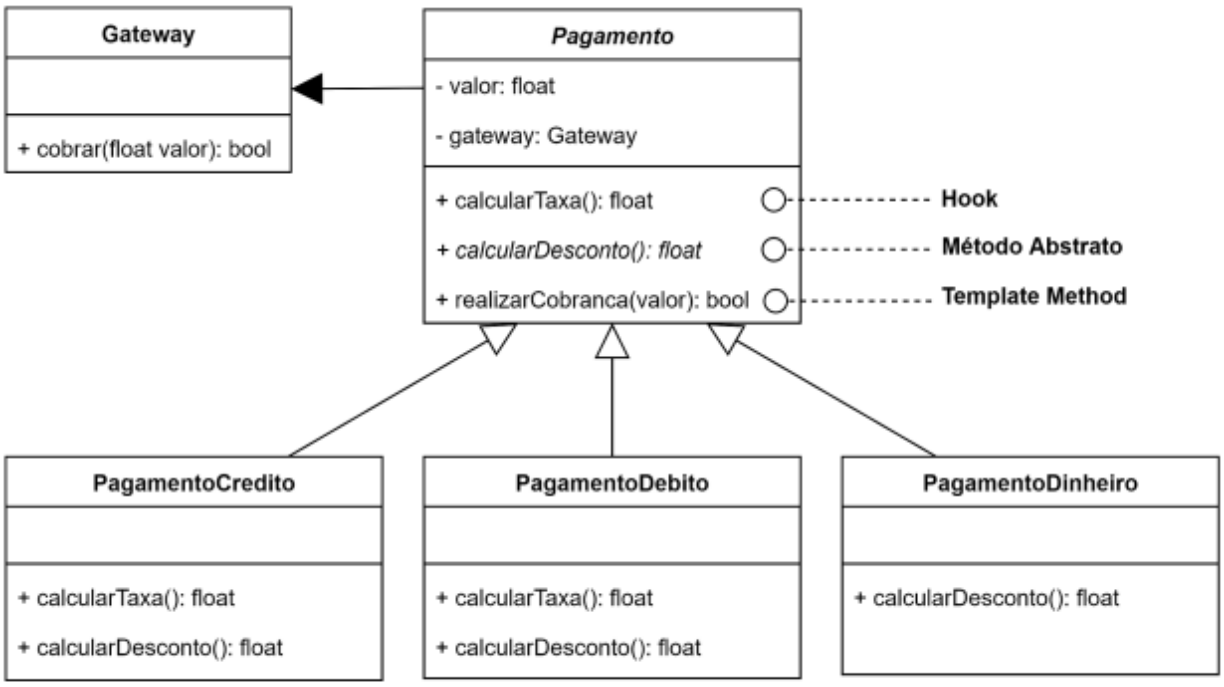

Diagrama de classes do exemplo com *Template Method*

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.

```
import { Payment } from "./payment"
export class CreditPayment extends Payment {

  calculateRate(): number {
```

```
    return this.value * 0.05
  }

  calculateDiscount(): number {
    if (this.value > 300) {
      return this.value * 0.02
    }
    return 0
  }

}
```

```
import { Payment } from "./payment"
export class DebitPayment extends Payment {

  calculateRate(): number {
    return 4
  }

  calculateDiscount(): number {
      return this.value * 0.05
  }

}
```

```
import { Payment } from "./payment";
export class MoneyPayment extends Payment {

  calculateDiscount(): number {
      return this.value * 0.1
  }

}
```

Testando o código com Template Method:

```
import { CreditPayment } from "./class/credit-payment"
import { DebitPayment } from "./class/debit-payment"
import { Gateway } from "./class/gateway"
import { MoneyPayment } from "./class/money-payment"

const value = 1000
const gateway = new Gateway()

const creditPayment = new CreditPayment(value, gateway)
console.log('Credit: ')
```

```
creditPayment.applyCollect()

const debitPayment = new DebitPayment(value, gateway)
console.log('Debit: ')
debitPayment.applyCollect()

const moneyPayment = new MoneyPayment(value, gateway)
console.log('Money: ')
moneyPayment.applyCollect()
```

Saída:

```
Credit:
R$ 1.030,00

Debit:
R$ 954,00

Money:
R$ 900,00
```

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.

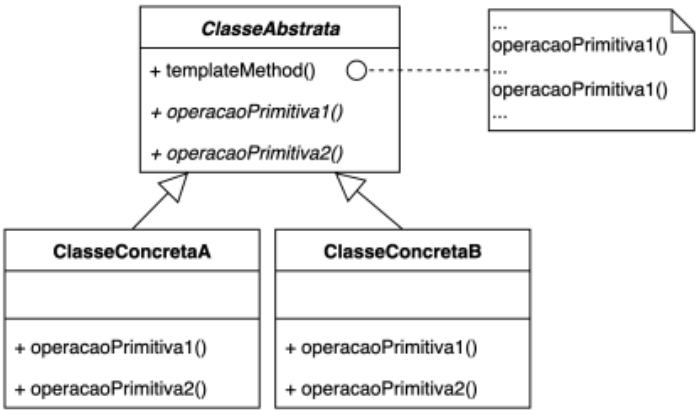

Diagrama de Classes

Consequências

Os Templates 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.