**Roteiro 02**

Os roteiros a serem desenvolvidos a seguir visam trazer a percepção da evolução no processo de desenvolvimento de software. Por isso, iremos criar um projeto com vários pacotes, onde cada pacote representa a evolução da implementação deste projeto.

Neste caso iremos explorar os Padrões de Projeto do GoF (Gang of Four), e neste roteiro iremos trabalhar com o **Template Method**.

Agora que já conhecemos algumas técnicas da OO, alguns princípios e boas práticas, iremos evoluir um novo cenário até chegar a um dos padrões, o Template Method.

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

Ele utiliza conceitos básicos de orientação a objetos, como herança e polimorfismo. Sua principal proposta é estabelecer um "esqueleto" de algoritmos em uma classe abstrata, permitindo que subclasses implementem partes específicas do algoritmo, enquanto mantêm a estrutura geral intacta.

A simplicidade do Template Method gera discussões na comunidade de desenvolvimento. Por ser tão direto e depender apenas de princípios básicos da programação orientada a objetos, alguns questionam se ele realmente merece ser considerado um "padrão de projeto" da mesma relevância que outros padrões mais complexos, como Factory Method ou Observer. Há quem argumente que o Template Method pode ser visto mais como uma boa prática de design do que como um padrão em si.

No entanto, sua inclusão no catálogo dos GOF se justifica pelo impacto e pela recorrência de seu uso para resolver problemas específicos em design de software. Ele fornece uma maneira estruturada e eficaz de organizar código reutilizável, o que ainda é fundamental em muitos contextos.

**Cenário:**

Imagine que temos uma equipe dentro de uma *Software house* e nossa equipe é responsável por manter um sistema de ERP de uma empresa do ramo de varejo. Nossa equipe estará focada especificamente no módulo de pagamento do ERP. Ou seja, iremos trabalhar em cima de um sistema legado. Além disso, no módulo de pagamento temos uma integração com um sistema de terceiro. Trata-se de um Gateway de pagamento, semelhante ao Paypal, PagSeguro e etc. Este Gateway é responsável por fazer a transação bancária e nós precisamos apenas fazer a requisição através deste Gateway. No nosso cenário iremos usar uma classe bem simples para simular o Gateway, apenas para efeitos didáticos e discutir a modelagem.

Como se trata de um sistema legado, já temos um código implementado com as classes abaixo, mas precisamos analisar para evoluir esta modelagem para futuras mudanças

**Diagrama

O conteúdo gerado por IA pode estar incorreto.**

Observe que temos 3 formas de pagamento e uma classe correspondente para cada uma delas (Crédito, Débito, Dinheiro). E todas elas utilizam uma instância de Gateway para realizar a cobrança através do método cobrar(). Este método por sua vez, retorna um boolean para indicar se a operação financeira foi bem-sucedida ou não.

Cada uma das classes de pagamento possui :

* Atributo
  + valor – representa o valor que deve ser cobrado
  + gateway – guarda a instância do Gateway para realizar a cobrança
* Método
  + calcularTaxa() – calcula a taxa cobrada pelo uso do gateway ou qualquer outra taxa dependendo do tipo de pagamento.
    - Crédito – cobrança de 5% do valor da compra
    - Débito – cobrança de R$ 4,00 pela operação
    - Dinheiro – não tem nenhuma taxa envolvida
  + calcularDesconto() – calcula o desconto que loja vai dar por alguns tipos de pagamento.
    - Crédito – Desconto de 2% para valores maiores que R$ 300,00. Caso contrário, não tem desconto
    - Débito – Desconto de 5%
    - Dinheiro – Desconto de 10%
  + RealzarCobrança() – Método que contêm a lógica para calcular o valor final a ser pago e é responsável por delegar a cobrança para o Gateway.

**Pacote : roteiro2.parte1**

1 – Dê sequência ao mesmo projeto no NetBeans chamado **PADROESroteiros**

2 – Como foi explicado no cenário inicial, estamos trabalhando em um sistema legado. Por isso, crie as classes abaixo no pacote **roteiro2.parte1**, e considere que já temos este código implementado como ponto de partida.

package roteiro2.parte1;

public class PagamentoCredito {

private double valor;

private Gateway gateway;

public PagamentoCredito(double valor, Gateway gateway){

this.valor = valor;

this.gateway = gateway;

}

public double calcularTaxa(){

return this.valor\*0.05;

}

public double calcularDesconto(){

if (this.valor > 300)

{

return this.valor\*0.02;

}

return 0;

}

public boolean realizarCobranca(){

double taxa = this.calcularTaxa();

double desconto = this.calcularDesconto();

double valorFinal = this.valor + taxa - desconto;

return this.gateway.cobrar(valorFinal);

}

}

package roteiro2.parte1;

import java.util.Random;

public class Gateway {

public boolean cobrar(double valor){

System.out.println("Valor cobrado : "+ valor);

Random random = new Random(); // Cria uma instância de Random

boolean autorizado = random.nextBoolean(); // Gera um valor boolean aleatório como resposta do Gateway

System.out.println("Cobrança Autorizada : "+ autorizado);

return autorizado;

}

}

package roteiro2.parte1;

public class PagamentoDebito {

private double valor;

private Gateway gateway;

public PagamentoDebito(double valor, Gateway gateway){

this.valor = valor;

this.gateway = gateway;

}

public double calcularTaxa(){

return 4;

}

public double calcularDesconto(){

return this.valor\*0.05;

}

public boolean realizarCobranca(){

double taxa = this.calcularTaxa();

double desconto = this.calcularDesconto();

double valorFinal = this.valor + taxa - desconto;

return this.gateway.cobrar(valorFinal);

}

}

package roteiro2.parte1;

public class PagamentoDinheiro {

private double valor;

private Gateway gateway;

public PagamentoDinheiro(double valor, Gateway gateway){

this.valor = valor;

this.gateway = gateway;

}

public double calcularTaxa(){

return 0;

}

public double calcularDesconto(){

return this.valor\*0.1;

}

public boolean realizarCobranca(){

double taxa = this.calcularTaxa();

double desconto = this.calcularDesconto();

double valorFinal = this.valor + taxa - desconto;

return this.gateway.cobrar(valorFinal);

}

}

package roteiro2.parte1;

public class TestePagamento {

public static void main(String[] args) {

Gateway gateway = new Gateway();

double valor = 1000;

PagamentoCredito pagtoCredito = new PagamentoCredito(valor, gateway);

pagtoCredito.realizarCobranca();

System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

PagamentoDebito pagtoDebito = new PagamentoDebito(valor, gateway);

pagtoDebito.realizarCobranca();

System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

PagamentoDinheiro pagtoDinheiro = new PagamentoDinheiro(valor, gateway);

pagtoDinheiro.realizarCobranca();

}

}

3 – Faça sua análise crítica em relação a essa modelagem e implementação ?

**Pacote : roteiro2.parte2**

Tentaremos agora fazer a refatoração do código adotando o padrão Template Method. Como já foi explicado no início do roteiro, este padrão tem como proposta estabelecer um "esqueleto" de algoritmos em uma classe abstrata, permitindo que subclasses implementem partes específicas do algoritmo, enquanto mantêm a estrutura geral intacta

Se a ideia é refatorar o código adotando o Template Method teremos que utilizar uma estrutura de classe conforme o modelo abaixo

Observe que temos uma superclasse Abstrata que tem :

* Métodos concretos -
  + metodoA – temos este método apenas como exemplo, mas poderíamos ter vários outros método concretos e não tem grande importância o código implementado neles. Isso vai detender o contexto.
  + templateMethod – este método sim, define a proposta do padrão Template Method. Observe que o código implementado neste método possui um “esqueleto” lógico e depende de outros métodos desta classe. Alguns são concretos e os abstratos serão implementados na subclasse. Mas, o esqueleto deste método é mantido. Qualquer implementação feita na subclasse não vai interferir na logica deste método em questão (templateMethod).
* Métodos abstratos -
  + metodoB, metodoC, metodoD – estes métodos serão implementados nas subclasses. Mesmo estes métodos sendo abstratos eles já compõem o templateMethod. Este é o grande mérito deste padrão.

public abstract class Superclasse {

public void metodoA(){

// implementação

}

public abstract void metodoB;

public abstract void metodoC;

public abstract void metodoD;

public void templateMethod(){

metodoA();

metodoB();

metodoC();

medodoD();

}

}

Vamos ver como isso pode nos ajudar nesta refatoração

1 – No mesmo projeto crie o pacote roteiro2.parte2

2 – Copie a classe Gateway da parte1, pois ela não sofrerá nenhum impacto.

3 – Vamos agora tentar criar a nossa superclasse abstrata (**Pagamento**) conforme indica o padrão Template Method.

Procure analisar as seguintes análises :

Qual o método desta classe realiza o papel do template method ?

Por que o método calcular desconto() é abstrato e o método calcularTaxa é concreto ?

package roteiro2.parte2;

public abstract class Pagamento {

protected double valor;

protected Gateway gateway;

public Pagamento(double valor, Gateway gateway){

this.valor = valor;

this.gateway = gateway;

}

public double calcularTaxa(){

return 0;

}

public abstract double calcularDesconto();

public boolean realizarCobranca(){

double taxa = this.calcularTaxa();

double desconto = this.calcularDesconto();

double valorFinal = this.valor + taxa - desconto;

return this.gateway.cobrar(valorFinal);

}

}

4 – Deveremos aproveitar parte do código na parte2 das seguintes classes : **PagamentoCredito**, **PagamentoDebito**, **PagamentoDinheiro**. Elas agora deverão subclasses de Pagamento.

Faça as seguintes refatorações :

* Adicione **: extends Pagamento** em cada uma destas classes
* Faça os ajustes necessários no construtor de cada uma destas classes.
* O método realizarCobranca() deve ser eliminado destas classes já que agora ele pertence a Suberclasse Pagamento.
* O método calcularTaxa() pode ser eliminado da classe PagamentoDinheiro, uma vez que o pagamento em dinheiro não tem Taxa, e este método existe na Superclasse Pagamento.
* Faça a sobrecarga do método abstrato conforme a regra de negócio de cada classe.

5 – Podemos agora refazer os testes utilizando a classe TestePagamento

package roteiro2.parte2;

public class TestePagamento {

public static void main(String[] args) {

Gateway gateway = new Gateway();

double valor = 1000;

PagamentoCredito pagtoCredito = new PagamentoCredito(valor, gateway);

pagtoCredito.realizarCobranca();

System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

PagamentoDebito pagtoDebito = new PagamentoDebito(valor, gateway);

pagtoDebito.realizarCobranca();

System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

PagamentoDinheiro pagtoDinheiro = new PagamentoDinheiro(valor, gateway);

pagtoDinheiro.realizarCobranca();

}

}

6 – Observe que no teste acima cada instância de formas de pagamento utiliza diferentes entidades : PagamentoCredito, PagamentoDebito, PagamentoDinheiro.

Esta modelagem permite que usemos todas as instâncias de um único tipo (Pagamento) ?

Por que essa análise é importante ?

7 – Utilize uma ferramenta de software qualquer para geração do diagrama de classes para esta etapa do projeto (Sugestão : Astah Community). Obs.: Adicione aqui o diagrama para que seja disponibilizado no teams

Obs: Outra possibilidade para geração dos diagramas é o StarUML : <https://staruml.io/> . Para fazer a engenharia reversa do código siga as orientações conforme o vídeo a seguir : <https://www.youtube.com/watch?v=z6DDuZQArro>

**Análise**

Já tivemos a oportunidade de estudar o padrão Strategy no Roteiro 01 e agora estamos estudando o padrão Template Method. Podemos dizer que estes dois padrões somados formam uma combinação perfeita de desing de software.

Essa combinação se alinha bem com o princípio do SOLID, particularmente o princípio de Aberto/Fechado (Open/Closed Principle),

* Permite criar algoritmos com partes fixas e extensíveis (Template Method)
* Ao mesmo tempo que encapsula variações de comportamento (Strategy)

Essa sinergia é útil em cenários onde existe a necessidade de uma estrutura geral consistente, mas com a possibilidade de alterar comportamentos específicos, como em sistemas de processamento de pagamentos, configurações de jogos, ou até frameworks de desenvolvimento.

Isso torna essa combinação não apenas poderosa, mas também amplamente adotada no mundo real.

8 – Agora faça a sua análise.

Consegue identificar semelhanças nesta modelagem com o padrão Strategy ?

Tomando como referência o modelo formal do Strategy, qual classe está fazendo o papel do “Strategy” e qual classe está fazendo o papel do “Client do Strategy” ?

**Pacote : roteiro2.parte3**

Vamos exercitar um pouco a aplicação dos dois padrões associados (Strategy + Template Method)

Utilize como referência o cenário do roteito1.parte5, onde temos o cenário de um sistema de e-commerce com os departamentos de eletrônicos e de móveis

No roteiro 1 provavelmente você implementou uma solução aplicando o padrão Strategy.

A sua solução já contempla o Template Method ? Como ?

Caso não contemple o Template Method, procure desenvolver uma solução de forma que associe os dois padrões.

**OBS**.: Se achar necessário, pode fazer qualquer modificação no cenário e adicionar novas funcionalidades ao projeto. Use a criatividade! O importante é exercitar a técnica e encontrar uma modelagem que faça sentido e associe os dois padrões.