**A grande variedade de impostos e o padrão strategy**

**Muitas regras e código complexo**

Tomando como exemplo uma aplicação cujo objetivo é a criação de orçamentos, temos uma regra de negócio na qual os valores dos orçamentos podem ser submetidos à alguns impostos, como ISS, ICMS e assim por diante. Com isso, temos a simples classe que representa o orçamento, recebendo via construtor o seu valor:

class Orcamento {

private $valor;

function \_\_construct($valor) {

$this->valor = $valor;

}

public function getValor() {

return $this->valor;

}

}

Com isso, podemos criar novos orçamentos, instanciando objetos do respectivo tipo e caso queiramos calcular um imposto sobre seu valor, basta utilizarmos o atributo valor para isso. Assim, podemos estipular que o ICMS valha 10% e precisamos calculá-lo, baseado no valor do orçamento. Para isso, podemos ter a seguinte classe com um simples método para realizar o cálculo:

class CalculadorDeImpostos {

public function realizaCalculo(Orcamento $orcamento) {

$icms = $orcamento->getValor() \* 0.1;

echo $icms; // imprimirá 50

}

}

Podemos ainda querer calcular outro imposto, como o ISS, que é 6% do valor do orçamento. Com isso, adicionamos a nova regra ao código anterior. Mas devemos escolher qual o imposto que será calculado. Portanto, o método realizaCalculo deverá receber uma informação, indicando qual o imposto terá o cálculo realizado:

class CalculadorDeImpostos {

public void realizaCalculo(Orcamento $orcamento, $imposto) {

if( $imposto == "ICMS" ) {

$icms = $orcamento->getValor() \* 0.1;

echo $icms; // imprimirá 50.0

} else if($imposto == "ISS") {

$iss = $orcamento->getValor() \* 0.06;

echo $iss; // imprimirá 30.0

}

}

}

Note que uma das consequências do código que acabamos de criar, é que espalhamos os cálculos e nossas regras de negócio. Dessa maneira, não temos nenhum encapsulamento de nossas regras de negócio e elas se tornam bastante suscetíveis a serem replicadas em outros pontos do código da aplicação. Por que não encapsulamos as regras dos cálculos em uma classe especializada para cada imposto?

**Encapsulando o comportamento**

Em vez de mantermos as regras espalhadas pela nossa aplicação, podemos encapsulá-las em classes cujas responsabilidades sejam realizar os cálculos. Para isso, podemos criar as classes ICMS e ISS cada um com seu respectivo método para calcular o valor do imposto de acordo com o orçamento.

class ICMS {

public function calculaICMS(Orcamento $orcamento) {

return $orcamento->getValor() \* 0.1;

}

}

class ISS {

public function calculaISS(Orcamento $orcamento) {

return $orcamento->getValor() \* 0.06;

}

}

Agora temos as duas classes que separam a responsabilidade dos cálculos de impostos, com isso, podemos utilizá-las na classe CalculadorDeImpostos da seguinte maneira:

class CalculadorDeImpostos {

public function realizaCalculo(Orcamento $orcamento, $imposto) {

if($imposto == "ICMS" ) {

$icms = new ICMS();

$icms = $icms->calculaICMS($orcamento);

echo $icms; // imprimirá 50.0

} else if( $imposto == "ISS" ) {

$iss = new ISS();

$iss = $iss->calculaISS($orcamento);

echo $iss; // imprimirá 30.0

}

}

}

Agora o código está melhor, mas não significa que esteja bom. Um ponto extremamente crítico desse código é o fato de que quando quisermos adicionar mais um tipo diferente de cálculo de imposto em nosso calculador, teremos que alterar essa classe adicionando mais um bloco de if, além de criarmos a classe que encapsulará o cálculo do novo imposto. Parece bastante trabalho.

**Eliminando os condicionais com polimorfismo e o pattern Strategy**

O que queremos em nosso código é não realizar nenhum condicional, ou seja, não termos mais que fazer ifs dentro do CalculadorDeImpostos. Dessa forma, não devemos mais receber a String com o nome do imposto, no qual realizamos os ifs. Mas como escolheremos qual o imposto que deve ser calculado?

Uma primeira possibilidade é criar dois métodos separados na classe CalculadorDeImposto. Um para o ICMS e outro para o ISS, dessa forma teremos:

class CalculadorDeImpostos {

public function realizaCalculoICMS(Orcamento $orcamento) {

$icms = new ICMS();

$icms = $icms->calculaICMS($orcamento);

echo $icms;

}

public function realizaCalculoISS(Orcamento $orcamento) {

$iss = new ISS();

$iss = $iss->calculaISS($orcamento);

echo $iss;

}

}

No entanto, agora só transferimos o problema dos vários ifs para vários métodos. O que não resolve o problema. O próximo passo para conseguirmos melhorar essa solução é termos um único método, genérico, que consegue realizar o cálculo para qualquer imposto, sem fazer nenhum if dentro dele.

class CalculadorDeImpostos {

public function realizaCalculo(Orcamento $orcamento) {

$icms = new ICMS();

$icms = $icms->calculaICMS($orcamento);

// Mas e se quisermos outro imposto?

echo $icms;

}

}

Agora estamos presos ao ICMS. Precisamos que nosso código fique flexível o bastante para utilizarmos diferentes impostos na realização do cálculo. Uma possibilidade para resolvermos esse problema é, ao invés de instanciarmos o imposto que desejamos dentro do método, recebermos uma instância do Imposto que queremos utilizar, como no código seguinte:

class CalculadorDeImpostos {

public function realizaCalculo(Orcamento $orcamento, Imposto $imposto) {

$valor = $imposto->calcula($orcamento);

echo $valor;

}

}

No entanto, não temos o tipo Imposto em nossa aplicação e além disso, nesse tipo precisamos passar uma instância de ISS e ICMS. Para isso, podemos criar uma interface chamada Imposto e fazermos as classes ISS e ICMS a implementar.

interface IImposto {

public function calcula(Orcamento $orcamento);

}

class ICMS implements IImposto {

public function calcula(Orcamento $orcamento) {

return $orcamento->getValor() \* 0.1;

}

}

class ISS implements IImposto {

public function calcula(Orcamento $orcamento) {

return $orcamento->getValor() \* 0.06;

}

}

E agora o nosso CalculadorDeImposto está pronto para ser utilizado e flexível o bastante para receber diferentes tipos (ou "estratégias") de impostos. Um código que demonstra essa flexibilidade é o seguinte:

function autoload($class) {

include $class.".php";

}

spl\_autoload\_register("autoload");

$orcamento = new Orcamento(500);

$icms = new ICMS();

$iss = new ISS();

$calculadorDeImposto = new calculadorDeImpostos();

echo $calculadorDeImposto->calculaImposto($orcamento,$icms);

echo "<br />";

echo $calculadorDeImposto->calculaImposto($orcamento,$iss);

Agora, com um único método em nosso CalculadorDeImposto, podemos realizar o cálculo de diferentes tipos de impostos, apenas recebendo a estratégia do tipo do imposto que desejamos utilizar no cálculo.

Quando utilizamos uma hierarquia, como fizemos com a interface Imposto e as implementações ICMS e ISS, e recebemos o tipo mais genérico como parâmetro, para ganharmos o polimorfismo na regra que será executada, simplificando o código e sua evolução, estamos usando o Design Pattern chamado Strategy.

Repare que a criação de uma nova estratégia de cálculo de imposto não implica em mudanças no código escrito acima! Basta criarmos uma nova classe que implementa a interface Imposto, que nosso CalculadorDeImpostoconseguirá calculá-lo sem precisar de nenhuma alteração!