**Desing paterns – Padrões de projetos**

# Aula 1: A grande variedade de impostos e o padrão Strategy

* **Design Patterns**

Classes e métodos gigantes? Cinco minutos para entender o que aquele método faz ou onde está o código que faz uma alteração simples? Diversas variáveis e diversos ifs e fors no mesmo método? Código complexo e obscuro? Toda vez que uma alteração aparece, você precisa mudar em 20 classes diferentes? Sim, problemas muito comuns do nosso dia-a-dia. Mas por que isso acontece?

Um fato que é conhecido sobre todo software é que ele, mais cedo ou mais tarde, vai mudar: novas funcionalidades aparecerão, outras deverão ser alteradas etc. O problema é que, geralmente, essas mudanças não são feitas de forma muito planejada. Por esse motivo, durante o desenvolvimento de um projeto de software, é bem comum a criação de código onde as responsabilidades se misturam e espalham por várias classes, fazendo com que a manutenção do código fique cada vez mais difícil, já que uma simples mudança pode obrigar o desenvolvedor a alterar diversas classes. Nesse cenário, temos uma situação onde o 'Design' das classes não está bom e é possível melhorá-lo.

Para atingir o objetivo da melhora do 'Design' e diminuir o custo de manutenção, existem algumas técnicas, onde podemos aplicar a orientação a objetos de uma maneira a simplificar o código escrito.

Resolveremos diversos dos problemas que vemos em códigos, como classes e métodos gigantes que fazem muita coisa, classes que precisam conhecer mais 10 outras classes para fazer seu trabalho, métodos complicados com muitos ifs e condições, mudanças que se propagam por várias classes e assim por diante.

Algumas das principais técnicas para atingir um bom 'Design' são tão comuns que foram catalogadas em um conjunto de alternativas para solucionar problemas de Design de código, chamados de Design Patterns, mais conhecido em português como os Padrões de Projetos, os quais, durante esse curso, aprenderemos a utilizar em um projeto. Um padrão de projeto nada mais é do que uma solução elegante para um problema que é muito recorrente em nosso dia-a-dia.

Mais importante do que entender como é a implementação de um padrão de projeto, é entender a motivação do padrão: em quais casos ele faz sentido e deve ser aplicado. Durante os próximos capítulos, observe o cenário do exemplo dado com atenção e o leve para o seu mundo e para o contexto de sua aplicação. Aquele cenário acontece? Se sim, então talvez o padrão de projeto ensinado naquele capítulo seja uma boa saída para controlar a crescente complexidade daquele código.

Durante todos os outros capítulos, faremos muitos exercícios. Todas as explicações serão baseadas em problemas do mundo real e você, ao final, utilizará os conhecimentos adquiridos para resolver os exercícios, que são desafiadores! Não se esqueça de prestar muita atenção ao cenário e ao problema proposto. Entenda a motivação de cada padrão para que você consiga levá-lo para os problemas do seu mundo!

* **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:

public class Orcamento

{

public double Valor { get; private set; }

public Orcamento(double valor)

{

this.Valor = 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:

public class CalculadorDeImpostos

{

public void RealizaCalculo(Orcamento orcamento)

{

double icms = orcamento.Valor \* 0.1;

Console.WriteLine(icms); // imprimirá 50.0

}

}

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:

public class CalculadorDeImpostos

{

public void RealizaCalculo(Orcamento orcamento, String imposto)

{

if( "ICMS".Equals(imposto) )

{

double icms = orcamento.Valor \* 0.1;

Console.WriteLine(icms); // imprimirá 50.0

}

else if( "ISS".Equals(imposto) )

{

double iss = orcamento.Valor \* 0.06;

Console.WriteLine(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**

Ao invés 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 cálcular o valor do imposto de acordo com o orçamento.

public class ICMS

{

public double CalculaICMS(Orcamento orcamento)

{

return orcamento.Valor \* 0.1;

}

}

public class ISS

{

public double CalculaISS(Orcamento orcamento)

{

return orcamento.Valor \* 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:

public class CalculadorDeImpostos

{

public void RealizaCalculo(Orcamento orcamento, String imposto)

{

if( "ICMS".Equals(imposto) )

{

double icms = new ICMS().CalculaICMS(orcamento);

Console.WriteLine(icms); // imprimirá 50.0

}

else if( "ISS".Equals(imposto) )

{

double iss = new ISS().CalculaISS(orcamento);

Console.WriteLine(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 separadas na classe CalculadorDeImposto. Um para o ICMS e outro para o ISS, dessa forma teremos:

public class CalculadorDeImpostos

{

public void RealizaCalculoICMS(Orcamento orcamento)

{

double icms = new ICMS().CalculaICMS(orcamento);

Console.WriteLine(icms);

}

public void RealizaCalculoISS(Orcamento orcamento)

{

double iss = new ISS().CalculaISS(orcamento);

Console.WriteLine(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.

public class CalculadorDeImpostos

{

public void RealizaCalculo(Orcamento orcamento)

{

double icms = new ICMS().CalculaICMS(orcamento);

// Mas e se quisermos outro imposto?

Console.WriteLine(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:

public class CalculadorDeImpostos

{

public void RealizaCalculo(Orcamento orcamento, Imposto imposto)

{

double valor = imposto.Calcula(orcamento);

Console.WriteLine(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.

public interface Imposto

{

double Calcula(Orcamento orcamento);

}

public class ICMS : Imposto

{

public double Calcula(Orcamento orcamento)

{

return orcamento.Valor \* 0.1;

}

}

public class ISS : Imposto

{

public double Calcula(Orcamento orcamento)

{

return orcamento.Valor \* 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:

public class TesteDeImpostos

{

static void Main(String[] args)

{

Imposto iss = new ISS();

Imposto icms = new ICMS();

Orcamento orcamento = new Orcamento(500.0);

CalculadorDeImposto calculador = new CalculadorDeImposto();

// Calculando o ISS

calculador.RealizaCalculo(orcamento, iss);

// Calculando o ICMS

calculador.RealizaCalculo(orcamento, icms);

}

}

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 CalculadorDeImposto conseguirá calculá-lo sem precisar de nenhuma alteração!

O padrão Strategy é muito útil quando temos um conjunto de algoritmos similares, e precisamos alternar entre eles em diferentes pedaços da aplicação. No exemplo do vídeo, temos diferentes maneira de calcular o imposto, e precisamos alternar entre elas.

O Strategy nos oferece uma maneira flexível para escrever diversos algoritmos diferentes, e de passar esses algoritmos para classes clientes que precisam deles. Esses clientes desconhecem qual é o algoritmo "real" que está sendo executado, e apenas manda o algoritmo rodar. Isso faz com que o código da classe cliente fique bastante desacoplado das implementações concretas de algoritmos, possibilitando assim com que esse cliente consiga trabalhar com N diferentes algoritmos sem precisar alterar o seu código.

# Muitos Descontos e o Chain of Responsibility

Nosso orçamento pode receber um desconto de acordo com o tipo da venda que será efetuada. Por exemplo, se o cliente comprou mais de 5 itens, ele recebe 10% de desconto; se ele fez uma compra casada de alguns produtos, recebe 5% de desconto, e assim por diante.

Em uma implementação tipicamente procedural, teríamos algo do tipo:

public class CalculadorDeDescontos

{

public double Calcula(Orcamento orcamento) {

// verifica primeira regra de possível desconto

if(orcamento.Itens.Count > 5)

{

return orcamento.Valor \* 0.1;

}

// verifica segunda regra de possível desconto

else if(orcamento.Valor > 500.0)

{

return orcamento.Valor \* 0.07;

}

// verifica terceira, quarta, quinta regra de possível desconto ...

// um monte de ifs daqui pra baixo

}

}

Ver esse monte de ifs em sequência nos lembra o capítulo anterior, na qual extraímos cada um deles para uma classe específica. Vamos repetir o feito, já que com classes menores, o código se torna mais simples de entender:

public class DescontoPorCincoItens

{

public double Desconta(Orcamento orcamento)

{

if(orcamento.Itens.Count > 5)

{

return orcamento.Valor \* 0.1;

}

else

{

return 0;

}

}

}

public class DescontoPorMaisDeQuinhentosReais

{

public double Desconta(Orcamento orcamento)

{

if(orcamento.Valor > 500)

{

return orcamento.Valor \* 0.07;

}

else

{

return 0;

}

}

}

Veja que tivemos que sempre colocar um return 0;, afinal o método precisa sempre retornar um double. Vamos agora substituir o conjunto de ifs na classe CalculadorDeDesconto. Repare que essa classe procura pelo desconto que deve ser aplicado; caso o anterior não seja válido, tenta o próximo.

public class CalculadorDeDescontos

{

public double Calcula(Orcamento orcamento)

{

// vai chamando os descontos na ordem até que algum deles dê diferente de zero...

double desconto = new DescontoPorCincoItens().Desconta(orcamento);

if(desconto == 0)

desconto = new DescontoPorMaisDeQuinhentosReais().Desconta(orcamento);

if(desconto == 0)

desconto = new ProximoDesconto().Desconta(orcamento);

// ...

return desconto;

}

}

O código já está um pouco melhor. Cada regra de negócio está em sua respectiva classe. O problema agora é como fazer essa sequência de descontos ser aplicada na ordem, pois precisamos colocar mais um if sempre que um novo desconto aparecer.

Precisávamos fazer com que um desconto qualquer, caso não deva ser executado, automaticamente passe para o próximo, até encontrar um que faça sentido. Algo do tipo:

public class DescontoPorMaisDeQuinhentosReais

{

public double Desconta(Orcamento orcamento)

{

if(orcamento.Valor > 500)

{

return orcamento.Valor \* 0.07;

}

else

{

return PROXIMO DESCONTO;

}

}

}

Todos os descontos têm algo em comum. Todos eles calculam o desconto dado um orçamento. Podemos criar uma abstração para representar um desconto genérico. Por exemplo:

public interface IDesconto

{

double Desconta(Orcamento orcamento);

}

public class DescontoPorCincoItens : IDesconto { ... }

public class DescontoPorMaisDeQuinhentosReais : IDesconto { ... }

Para fazer aquele código funcionar agora, basta fazer com que todo desconto receba um próximo desconto! Observe o código abaixo:

public interface IDesconto

{

double Desconta(Orcamento orcamento);

Desconto Proximo { get; set; }

}

public class DescontoPorCincoItens : IDesconto

{

public Desconto Proximo { get; set; }

public double Desconta(Orcamento orcamento)

{

if(orcamento.Itens.Count > 5) {

return orcamento.Valor \* 0.1;

}

else

{

return Proximo.Desconta(orcamento);

}

}

}

public class DescontoPorMaisDeQuinhentosReais : IDesconto

{

public Desconto Proximo { get; set; }

public double Desconta(Orcamento orcamento)

{

if(orcamento.Valor > 500)

{

return orcamento.Valor \* 0.07;

}

else

{

return Proximo.Desconta(orcamento);

}

}

}

Ou seja, se o orçamento atende a regra de um desconto, o mesmo já calcula o desconto. Caso contrário, ele passa para o "próximo" desconto, qualquer que seja esse próximo desconto.

Basta agora plugarmos todas essas classes juntas. Veja que um desconto recebe um "próximo". Para o desconto, pouco importa qual é o próximo desconto. Eles estão totalmente desacoplados um do outro!

public class CalculadorDeDescontos {

public double Calcula(Orcamento orcamento)

{

IDesconto d1 = new DescontoPorCincoItens();

IDesconto d2 = new DescontoPorMaisDeQuinhentosReais();

d1.Proximo = d2;

return d1.Desconta(orcamento);

}

}

public class TestaDescontos

{

static void Main(String[] args)

{

CalculadorDeDescontos calculador = new CalculadorDeDescontos();

Orcamento orcamento = new Orcamento(500.0);

orcamento.AdicionaItem(new Item("CANETA", 250.0));

orcamento.AdicionaItem(new Item("LAPIS", 250.0));

double desconto = calculador.Calcula(orcamento);

Console.WriteLine(desconto);

}

}

Esses descontos formam como se fosse uma "corrente", ou seja, um ligado ao outro. Daí o nome do padrão de projeto: Chain of Responsibility. A ideia do padrão é resolver problemas como esses: de acordo com o cenário, devemos realizar alguma ação. Ao invés de escrevermos código procedural, e deixarmos um único método descobrir o que deve ser feito, quebramos essas responsabilidades em várias diferentes classes, e as unimos como uma corrente.

Nosso problema é só fazer o algoritmo parar agora. Se ele não encontrar nenhum desconto válido, o valor deve ser 0. Vamos criar a classe SemDesconto, que será o fim da corrente.

public class SemDesconto : IDesconto

{

public Desconto Proximo { get; set; }

public Double Desconta(Orcamento orcamento)

{

return 0;

}

}

public class CalculadorDeDescontos

{

public double Calcula(Orcamento orcamento)

{

IDesconto d1 = new DescontoPorCincoItens();

IDesconto d2 = new DescontoPorMaisDeQuinhentosReais();

IDesconto d3 = new SemDesconto();

d1.Proximo = d2;

d2.Proximo = d3;

return d1.Desconta(orcamento);

}

}

A classe SemDesconto não atribui o próximo desconto, pois ela não possui um próximo. Na realidade, ela é o ponto final da nossa cadeia de responsabilidades.

Note também que nossa classe Orcamento cresceu, e agora recebe itens também. A mudança é simples:

public class Orcamento

{

public double Valor { get; private set; }

public IList<Item> Itens { get; private set; }

public Orcamento(double valor)

{

this.Valor = valor;

this.Itens = new List<Item>();

}

public void AdicionaItem(Item item)

{

itens.Add(item);

}

}

public class Item

{

public String Nome { get; private set; }

public double Valor { get; private set; }

public Item(String nome, double valor)

{

this.Nome = nome;

this.Valor = valor;

}

}