Skip to navigation

# Marcos Brizeno

## Desenvolvimento de Software #showmethecode

# Command

novembro 4, 2011

# Mão na massa: Command

## *Problema*

Suponha uma loja que vende produtos e oferece várias formas de pagamento. Ao executar uma compra o sistema registra o valor total e, dada uma forma de pagamento, por exemplo, cartão de crédito, emite o valor total da compra para o cartão de crédito do cliente.

Para este caso então vamos supor as seguintes classes para simplificar o exemplo: Loja e Compra. A classe Loja representa a loja que está efetuando a venda. Vamos deixar a classe Loja bem simples, como mostra o seguinte código:

```
public class Loja {
        protected String nomeDaLoja;

        public Loja(String nome) {
                nomeDaLoja = nome;
        }

        public void executarCompra(double valor) {
                Compra compra = new Compra(nomeDaLoja);
                compra.setValor(valor);
        }
}
```

Para focar apenas no que é necessário para entender o padrão, vamos utilizar a classe Compra, que representa o conjunto de produtos que foram vendidos com o seu valor total:

```
public class Compra {
        private static int CONTADOR_ID;
        protected int idNotaFiscal;
        protected String nomeDaLoja;
        protected double valorTotal;

        public Compra(String nomeDaLoja) {
                this.nomeDaLoja = nomeDaLoja;
                idNotaFiscal = ++CONTADOR_ID;
        }

        public void setValor(double valor) {
                this.valorTotal = valor;
        }

        public String getInfoNota() {
                return new String("Nota fiscal nº: " + idNotaFiscal + "\nLoja: "
                                + nomeDaLoja + "\nValor: " + valorTotal);
        }
}
```

Pronto, agora precisamos alterar a classe Loja para que ela, ao executar uma compra saiba qual a forma de pagamento.

Uma maneira interessante de fazer esta implementação seria adicionando um parâmetro a mais no método executar que nos diga qual forma de pagamento deve ser usada. Poderíamos então utilizar um Enum para identificar a forma de pagamento e daí passar a responsabilidade ao objeto específico. Veja o exemplo que mostra o código do método executar compra utilizando uma enumeração:

```
        public void executarCompra(double valor, FormaDePagamento formaDePagament
                Compra compra = new Compra(nomeDaLoja);
                compra.setValor(valor);
                if(formaDePagamento == FormaDePagamento.CartaoDeCredito){
                        new PagamentoCartaoCredito().processarCompra(compra);
                } else if(formaDePagamento == FormaDePagamento.CartaoDeDebito){
                        new PagamentoCartaoDebito().processarCompra(compra);
                } else if(formaDePagamento == FormaDePagamento.Boleto){
                        new PagamentoBoleto().processarCompra(compra);
                }
        }
```

O problema desta solução é que, caso seja necessário incluir ou remover uma forma de pagamento precisaremos fazer várias alterações, alterando tanto a enumeração quando o método que processa a compra. Veja também a quantidade de ifs aninhados, isso é um sintoma de um design mal feito.

Poderíamos passar o objeto que faz o pagamento como um dos parametros, ao invés de utilizar a enumeração, assim não teríamos mais problemas com os ifs aninhados e as alterações seriam locais. Ok, está é uma boa solução, mas ainda não está boa, pois precisaríamos de um método diferente pra cada tipo de objeto.

A saída obvia então é utilizar uma classe comum a todos as formas de pagamento, e no parâmetro passar um objeto genérico! Essa é justamente a ideia do Padrão Command.

# ***Command***

Intenção:

"Encapsular uma solicitação como objeto, desta forma permitindo parametrizar cliente com diferentes solicitações, enfileirar ou fazer o registro de solicitações e suportar operações que podem ser desfeitas." [1]

Pela intenção vemos que o padrão pode ser aplicado em diversas situações. Para resolver o exemplo acima vamos encapsular as solicitações de pagamento de uma compra em objetos para parametrizar os clientes com as diferentes solicitações.

Perceba no código com os vários ifs outra boa oportunidade para refatorar o código. Dentro de cada um dos if, a ação é a mesma: criar um objeto para processar o pagamento e realizar uma chamada ao método de processamento da compra.

Então vamos primeiro definir a interface comum aos objetos que processam um pagamento. Todos eles possuem um mesmo método, processar pagamento, que toma como parâmetro uma compra e faz o processamento dessa compra de várias formas.

A classe interface seria a seguinte:

```
public interface PagamentoCommand {
        void processarCompra(Compra compra);
}
```

Uma possível implementação seria a de processar um pagamento via boleto. Vamos apenas emitir uma mensagem no terminal para saber que tudo foi executado como esperado:

```
public class PagamentoBoleto implements PagamentoCommand {

        @Override
        public void processarCompra(Compra compra) {
                System.out.println("Boleto criado!\n" + compra.getInfoNota());
        }

}
```

Agora o método de execução da compra na classe Loja seria assim:

```
public void executarCompra(double valor, PagamentoCommand formaDePagament
        Compra compra = new Compra(nomeDaLoja);
        compra.setValor(valor);
        formaDePagamento.processarCompra(compra);
}
```

O código cliente que usaria o padrão Command seria algo do tipo:

```
public static void main(String[] args) {
        Loja lojasAfricanas = new Loja("Afriacanas");
        lojasAfricanas.executarCompra(999.00, new PagamentoCartaoCredito(
        lojasAfricanas.executarCompra(49.00, new PagamentoBoleto());
        lojasAfricanas.executarCompra(99.00, new PagamentoCartaoDebito());

        Loja exorbitante = new Loja("Exorbitante");
        exorbitante.executarCompra(19.00, new PagamentoCartaoCredito());

}
```

Ao executar uma compra nós passamos o comando que deve ser utilizado, neste caso, a forma de pagamento utilizado.

O diagrama UML seria algo do tipo:

# Um pouco de teoria

O padrão tem um conceito muito simples, utiliza-se apenas da herança para agrupar classes e obrigar que todas tenham uma mesma interface em comum. A primeira vista o padrão pode ser confundido com o padrão Template Method, pois ambos utilizam a herança para unificar a interface de várias classes.

A diferença básica é que no padrão Command não existe a ideia de um “algoritmo” que será executado. No padrão Template Method as subclasses definem apenas algumas operações, sem alterar o esqueleto de execução. O padrão Command não oferece uma maneira de execução de suas subclasses, apenas garante que todas executem determinada requisição.

Como visto na intenção do padrão existem várias aplicações do padrão. Um exemplo seria oferecer um maior controle da execução de uma requisição. Por exemplo, caso fosse necessário primeiro armazenar todos as compras, cada uma com formas diferentes de pagamentos, para só então executar todas. O método de processar compra armazenaria os dados e, utilizando um outro método, por exemplo, finalizarCompra, todas as compras seriam processadas.

Ainda seguindo o exemplo, o padrão Command também poderia ser utilizado para desfazer operações, antes que elas sejam executadas de fato. Por exemplo, caso o cliente desista das compras, um método poderia desfazer/cancelar as alterações. Desta mesma forma poderíamos manter o controle sobre as mudanças, criando assim um log das operações.

As vantagens do padrão são: a facilidade de extensão da arquitetura, permitindo adicionar novos commands sem efeitos colaterais; e o bom nível de desacoplamento entre objetos, separando os objetos que possuem os dados dos que manipulam os dados.

Um problema que pode ocorrer ao utilizar o padrão Command é a complexidade dos comandos crescer demais. Por exemplo, se todos os commands precisam realizar várias ações, como: manter a persistência nas alterações, oferecer uma maneira de desfazer alterações, etc. Neste caso pode ser viável a utilização de um agrupamento de comandos, com o padrão Composite.

## *Código fonte completo*

O código completo pode ser baixado no seguinte repositório Git: https://github.com/MarcosX/Padr-es-de-Projeto.

Os arquivos estão como um projeto do eclipse, então basta colocar no seu Workspace e fazer o import.

Se gostou do post compartilhe com seus amigos e colegas, senão, comente o que pode ser melhorado. Encontrou algum erro no código? Comente também. Possui alguma outra opinião ou alguma informação adicional? Comenta ai! 😀

## *Referências:*

[1] GAMMA, Erich et al. Padrões de Projeto: Soluções reutilizáveis de software orientado a objetos.

 Command  Command, Java, Padrões, Projeto  4 Comentários

__PRESENT