Skip to navigation

Marcos Brizeno

Desenvolvimento de Software #showmethecode

Builder

setembro 25, 2011

Mão na massa: Builder

Quase encerrando a lista de Padrões de Criação, vamos exemplificar agora o padrão Builder!

Problema

Antes de falar do problema é preciso dizer que este padrão é bem parecido com o Factory Method e o Abstract Factory, então vamos analisar o mesmo exemplo para ver melhor as diferenças e semelhanças entre eles.

O problema é que precisamos modelar um sistema de venda de carros para uma concessionária. Queremos que o sistema seja flexível, para adição de novos carros, e de fácil manutenção. Vimos que com os padrões Factory Method e o Abstract Factory podemos alcançar este resultado de maneira bem simples.

Vamos então apresentar o padrão Builder e ver como ele funcionária nesta situação.

Builder

A intenção do padrão segundo [1]:

"Separar a construção de um objeto complexo de sua representação de modo que o mesmo processo de construção possa criar diferentes representações."

Separar a construção da representação segue a mesma ideia dos padrões Factory Method e Abstract Factory. No entanto o padrão Builder permite separar os passos de construção de um objeto em pequenos métodos. Então vamos direto ao código para ver o que muda.

No padrão Builder temos também uma interface comum para todos os objetos que constroem outros objetos. Essa interface Builder define todos os passos necessários para construir um objeto. Vamos então ver o nosso objeto produto:

```
1  public class CarroProduct {
2      double preco;
3      String dscMotor;
4      int anoDeFabricacao;
5      String modelo;
6      String montadora;
7  }
```

Nada mais é do que uma estrutura de dados (Lembre: estrutura de dados != objeto) que armazena as informações de um carro. A nossa classe Builder vai possui um método para construir cada um dos dados do nosso Produto:

```
1  public abstract class CarroBuilder {
2
3      protected CarroProduct carro;
4
5      public CarroBuilder() {
6          carro = new CarroProduct();
7      }
8
9      public abstract void buildPreco();
10
11     public abstract void buildDscMotor();
12
13     public abstract void buildAnoDeFabricacao();
14
15     public abstract void buildModelo();
16
17     public abstract void buildMontadora();
18
19     public CarroProduct getCarro() {
20         return carro;
21     }
22 }
```

Nesta classe temos o carro que será construído, os passos para sua construção e um método que devolve o carro construído. Apesar da aparente semelhança com o padrão Template Method, que deixa as subclasses definirem alguns métodos do algoritmo, na classe Builder não existe um "algoritmo" bem definido, o algoritmo será definido em outro lugar. Então vejamos agora as classes Builder concretas:

```java
public class FiatBuilder extends CarroBuilder {

    @Override
    public void buildPreco() {
        // Operação complexa.
        carro.preco = 25000.00;
    }

    @Override
    public void buildDscMotor() {
        // Operação complexa.
        carro.dscMotor = "Fire Flex 1.0";
    }

    @Override
    public void buildAnoDeFabricacao() {
        // Operação complexa.
        carro.anoDeFabricacao = 2011;
    }

    @Override
    public void buildModelo() {
        // Operação complexa.
        carro.modelo = "Palio";
    }

    @Override
    public void buildMontadora() {
        // Operação complexa.
        carro.montadora = "Fiat";
    }
}
```

Na classe FiatBuilder nós “personalizamos” o carro, com as informações da Fiat. Note os comentários “// Operação complexa”. Antes da atribuição do preço, por exemplo, poderíamos realizar todo o cálculo necessário, por exemplo, buscando o valor no banco de dados, calcular impostos, desvalorização, entre outras operações. Essa é a ideia principal do padrão Builder, dividir em pequenos passo a construção do objeto.

Agora vamos ver a classe chamada de Director, ela utiliza a estrutura do Builder para definir o algoritmo de construção do Produto.

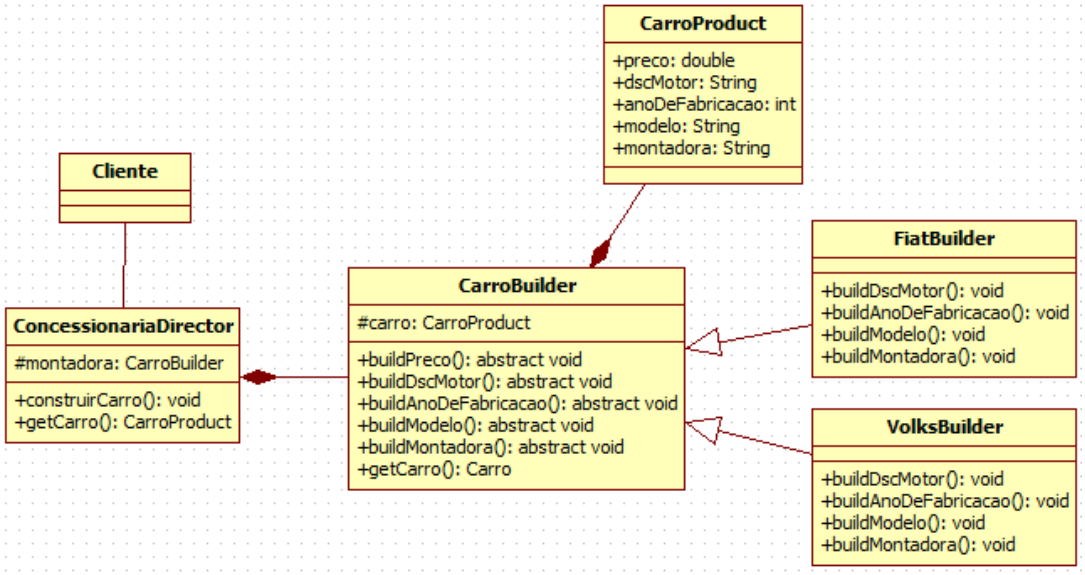
```
public class ConcessionariaDirector {
    protected CarroBuilder montadora;

    public ConcessionariaDirector(CarroBuilder montadora) {
        this.montadora = montadora;
    }

    public void construirCarro() {
        montadora.buildPreco();
        montadora.buildAnoDeFabricacao();
        montadora.buildDscMotor();
        montadora.buildModelo();
        montadora.buildMontadora();
    }

    public CarroProduct getCarro() {
        return montadora.getCarro();
    }
}
```

Dado um Builder, a classe vai executar os métodos de construção, definindo assim o algoritmo de construção do carro, e depois devolve o carro. O código cliente vai lidar apenas com o Director, toda a estrutura e algoritmos utilizados para construir o carro ficarão por debaixo dos panos.

```
public static void main(String[] args) {
    ConcessionariaDirector concessionaria = new ConcessionariaDirector(
            new FiatBuilder());

    concessionaria.construirCarro();
    CarroProduct carro = concessionaria.getCarro();
    System.out.println("Carro: " + carro.modelo + "/" + carro.montadora
            + "\nAno: " + carro.anoDeFabricacao + "\nMotor: "
            + carro.dscMotor + "\nValor: " + carro.preco);

    System.out.println();

    concessionaria = new ConcessionariaDirector(new VolksBuilder());
    concessionaria.construirCarro();
    carro = concessionaria.getCarro();
    System.out.println("Carro: " + carro.modelo + "/" + carro.montadora
            + "\nAno: " + carro.anoDeFabricacao + "\nMotor: "
            + carro.dscMotor + "\nValor: " + carro.preco);
}
```

Veja que foi bastante fácil mudar o produto, apenas precisamos utilizar um novo Builder para construir o produto que quisermos. A duplicação de código no cliente foi intencional, para mostrar que o cliente pode manipular o produto. Caso queira basta colocar o código que exibe as informações do carro em um método da classe Director, escondendo do cliente a estrutura do Produto.

Veja o resultado da utilização do padrão Builder:



Um pouco de teoria:

Já mostramos a principal vantagem do padrão Builder, separar em pequenos passos a construção do objeto complexo. Vamos então comparar, como foi dito no início, o padrão Builder com o Abstract Factory e o Factory Method.

Todos eles servem para criar objetos, além disse escondem a implementação do cliente, tornando o programa mais flexível. No entanto, no padrão Builder, não existe o conceito de vários produtos ou de famílias de produtos, como nos outros dois padrões.

Volte ao código e veja que definimos apenas um produto: Carro. Cada fábrica do Builder vai personalizar o seu Produto nos pequenos passos de construção. Essa diferença fica mais evidente ao comparar os diagramas UML do Facottory Method, Abstract Factory e Bulder. A estrutura do Builder é bem menor que a dos outros dois.

Outra diferença é que o Builder foca na divisão de responsabilidades na construção do Produto. Enquanto nos padrões Abstract Factory e Factroy Method tínhamos apenas o método criarCarro(), que deveriam executar todo o processo de criação e devolver o produto final, no padrão Builder nós definimos quais os passos devem ser executados (na classe Builder) e como eles devem ser executados (na classe Director).

Várias classes Director também podem reutilizar classes Builder. Como o Builder separar bem os passos de construção, o Director tem um controle bem maior sobre a produção do Produto.

Um problema com o padrão é que é preciso sempre chamar o método de construção para depois utilizar o produto em si. No nosso exemplo essa responsabilidade foi dada ao código cliente. No entanto a classe Director poderia realizar todas as chamadas em um único método e depois apenas retornar o produto final ao cliente.

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.

 Builder, Padrões de Projeto Builder, Java, Padrões, Projeto 20 Comentários

__PRESENT