Skip to navigation

# Marcos Brizeno

## Desenvolvimento de Software #showmethecode

setembro 17, 2011

# Mão na massa: Factory Method

Já ouviu falar em método fábrica? Este é o padrão! Factory Method

## *Problema*

Suponha que você deve trabalhar em um projeto computacional com um conjunto de carros, cada um de uma determinada fábrica. Para exemplificar suponha os quatro seguintes modelos/fabricantes:

- Palio – Fiat
- Gol – Volkswagen
- Celta – Chevrolet
- Fiesta – Ford

Será necessário manipular este conjunto de carros em diversas operações, como poderíamos modelar este problema?

Uma primeira solução, mais simples, seria criar uma classe para representar cada carro, no entanto ficaria muito difícil prever as classes ou escrever vários métodos iguais para tratar cada um dos tipos de objetos.

Poderíamos então criar uma classe base para todos os carros e especializá-la em subclasses que representem cada tipo de carro, assim, uma vez definida uma interface comum poderíamos tratar todos os carros da mesma maneira. O problema surge quando vamos criar o objeto, pois, de alguma forma, precisamos identificar qual objetos queremos criar. Ou seja, precisaríamos criar uma enumeração para identificar cada um dos carros e, ao criar um carro, identificaríamos seguindo essa enumeração. Veja o código abaixo:

```
1  public enum ModeloCarro {
2      palio,gol, celta, fiesta
3  }
```

A classe de criação de carros:

```
public abstract class FabricaCarro {
    public Carro criarCarro(ModeloCarro modelo) {
        switch (modelo) {
        case celta:
            return new Celta();
        case fiesta:
            return new Fiesta();
        case gol:
            return new Gol();
        case palio:
            return new Palio();
        default:
            break;
        }
    }
}
```

Esta implementação já corresponde a uma implementação do Factory Method, pois um método fábrica cria Objetos concretos que só serão definidos em tempo de execução. No entanto, esta implementação traz um problema quanto a manutenibilidade do código, pois, como utilizamos um switch para definir qual objeto criar, a cada criação de um novo modelo de carro precisaríamos incrementar este switch e criar novas enumerações. Como resolver este problema?

# ***Factory Method***

O padrão Factory Method possui a seguinte intenção:

"Definir uma interface para criar um objeto, mas deixar as subclasses decidirem que classe instanciar. O Factory Method permite adiar a instanciação para subclasses." [1]

Ou seja, ao invés de criar objetos diretamente em uma classe concreta, nós definimos uma interface de criação de objetos e cada subclasse fica responsável por criar seus objetos. Seria como se, ao invés de ter uma fábrica de carros, nós tivéssemos uma fábrica da Fiat, que cria o carro da Fiat, uma fábrica da Ford, que cria o carro da Ford e etc.

A nossa interface de fábrica seria bem simples:

```
public interface FabricaDeCarro {
    Carro criarCarro();
}
```

E, tão simples quanto, seriam as classes concretas para criar carros:

```
public class FabricaFiat implements FabricaDeCarro {

    @Override
    public Carro criarCarro() {
        return new Palio();
    }

}
```

As outras fábricas seguem a mesma ideia, cada uma define o método de criação de carros e cria o seu próprio carro. Agora que vimos as classes fábricas, vamos analisar os produtos.

Como já discutimos antes, vamos criar uma interface comum para todos os carros, assim poderemos manipulá-los facilmente:

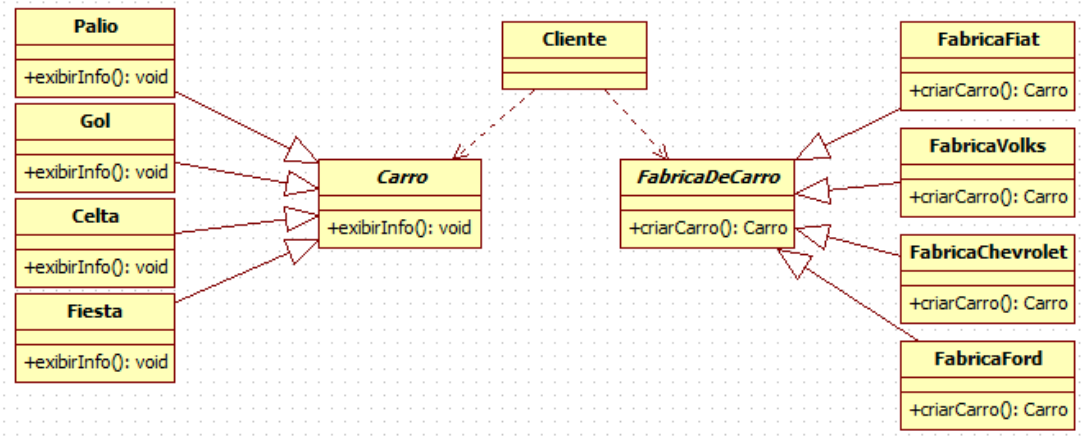
```
1  public interface Carro {
2      void exibirInfo();
3  }
```

Para o nosso exemplo vamos considerar apenas que precisamos exibir informações sobre os carros. Quaisquer outras operações seriam definidas nessa interface também. Caso uma mesma operação precisasse ser definida para todos os carros poderíamos implementar esta classe como uma classe abstrata e implementar os métodos necessários.

Os produtos concretos seriam definidos da seguinte maneira:

```
1  public class Palio implements Carro {
2      @Override public void exibirInfo() {
3          System.out.println("Modelo: Palio\nFabricante: Fiat");
4      }
5  }
```

Ou seja, no final das contas teríamos a seguinte estrutura:




(https://brizeno.files.wordpress.com/2011/09/factory-method.png)

# Um pouco de teoria

Como vimos, a principal vantagem em utilizar o padrão Factory Method é a extrema facilidade que temos para incluir novos produtos. Não é necessário alterar NENHUM código, apenas precisamos criar o produto e a sua fábrica. Todo o código já escrito não será alterado.

No entanto isto tem um custo. Perceba que criamos uma estrutura relativamente grande para resolver o pequeno problema, temos um conjunto grande de pequenas classes, cada uma realizando uma operação simples. Apesar de seguir o princípio da responsabilidade única [2], para cada novo produto precisamos sempre criar duas classes, uma produto e uma fábrica.

Na primeira sugestão de implementação nós definimos o Factory Method em uma classe concreta, isso evita a criação de várias classes pequenas de fábrica, no entanto acaba criando um código gigante para criação de objetos. Durante a implementação é necessário escolher qual tipo de implementação resolve melhor o seu problema.

## *Código fonte completo*

O código completo pode ser baixado no seguinte repositório Git: https://github.com/MarcosX/Padr-es-de-Projeto (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.
[2] WIKIPEDIA. SOLID. Disponível em: http://en.wikipedia.org/wiki/SOLID_(object-oriented_design) (http://en.wikipedia.org/wiki/SOLID_(object-oriented_design)). Acesso em: 15 set. 2011.

 Factory Method, Padrões de Projeto  Factory Method, Java, Padrões, Projeto  36 Comentários

# 36 comentários sobre “Mão na massa: Factory Method”

1.  janeiro 18, 2013 às 10:20 AM
   **Tiago**
   Cara muito obrigado por isso, você facilitou muito o meu aprendizado, Deus te abençoe!

   Responder
2.  junho 3, 2013 às 5:45 PM
   **Diego Souza**
   E se eu quiser implementar na fábrica da Fiat a opção de criar um Uno, Strada, Siena, Bravo e Punto? pela interface de Fabrica de Carro só me dá uma opção.

   Responder
   - junho 4, 2013 às 9:59 AM
     **marcosbrizeno**
     Correto. Esse é um dos pontos fracos do Factory Method, apesar da interface flexível, não dá pra fazer esse tipo de coisa.
     Uma opção seria utilizar o Abstract Factory (https://brizeno.wordpress.com/2011/09/18/mao-na-massa-abstract-factory/) para criar famílias de objetos com as fábricas.

     Responder
     - julho 9, 2016 às 9:17 PM
       **Anônimo**

Mas posso chamar uma lista de carros de cada fabrica, porém tenho que criar uma classe para cada marca (Uno, Strada, Saveiro, etc…)

```
@Override
public List criarCarros() {
// TODO Auto-generated method stub
List carros = new ArrayList();
carros.add(new Palio());
carros.add(new Uno());
return carros;
}
```

julho 9, 2016 às 9:22 PM

**Diovanni**

Mas dá para criar uma lista de carros de cada fabrica, porém tenho que criar uma classe para cada marca (Uno, Strada, Saveiro, etc…)

```
@Override
public List criarCarros() {
// TODO Auto-generated method stub
List carros = new ArrayList();
carros.add(new Palio());
carros.add(new Uno());
return carros;
}
```

dezembro 11, 2016 às 2:02 AM

**Rafael - Rafa**

Foi exatamente o que pensei. Para o caso citado pelo colega Diego Souza seria melhor implementar o padrão Abstract Factory. A abstração seria melhor para o caso de uma fábrica de carros devido aos modelos.

3. março 21, 2014 às 3:12 PM

**Jr**

Olá Marcos, parabéns pela iniciativa.
Uma observação, no último exemplo, antes de mostrar o diagrama de classes, o código não deveria mostrar a realização de um produto das classes carro? Está mostrando novamente a realização de uma fábrica.

Abraço!

Responder

março 21, 2014 às 7:46 PM

**Marcos Brizeno**

O código antes do diagrama mostra como seria uma Factory e dentro dela, no método criarCarro() ele retorna um novo objeto do tipo carro. Ou seja, ao chamar criarCarro na factory um novo carro vai ser retornado.

Responder

maio 14, 2015 às 10:22 AM

**Naka**

Marcos, ótima iniciativa cara, parabéns.

Vi que, assim como eu, alguns tiveram problemas com o exemplo duplicado, por mais que ele defina as duas afirmativas do artigo.
Pra quem tá lendo o artigo e tendo dificuldade, recomendo ler a série como um todo, minha opinião é que a medida que vcs forem entendendo os padrões, verão que realmente "Os produtos concretos seriam definidos da seguinte maneira".

Agora, por mais conceitual que seja aquele diagrama de classes, as classes concretas de fábrica não teriam relação com as classes concretas de produto?

maio 16, 2015 às 6:15 AM

**Marcos Brizeno**

Obrigado, fico feliz que tenha gostado.
Cada fábrica concreta estaria cria um produto, então existe sim uma relação entre fábricas concretas e produtos. Só não coloquei no diagrama pois ficariam muitas linhas.

4. maio 20, 2014 às 8:48 PM

**Paulo**

Isso para mim é um exemplo do Padrão Abstract Factory, não? O Factory Method ficaria só naquela primeira parte…

Responder

maio 21, 2014 às 9:54 AM

**Marcos Brizeno**

Oi Paulo, sim é bem difícil desenhar a linha entre o Factory Method e o Abstract Factory. Mas acho que no caso do Abstract Factory você tem mais de um tipo base de produto.
Se tu olhar o outro post sobre Abstract Factory talvez fique mais fácil de ver devido aos dois tipos de carros que são criados.

Responder

5. julho 15, 2014 às 1:14 AM

**Anônimo**

Olá, no trecho onde você inseriu o código do produto concreto tem um problema, acredito que você colocou o código da fábrica concreta por engano, por favor corrija.

Responder

julho 15, 2014 às 8:39 AM

**Marcos Brizeno**

Não sei se entendi muito bem, mas acho que o código está correto. A ideia é que cada fábrica saiba apenas como criar seu próprio produto sem saber nada sobre os outros, e a classe abstrata serve apenas de interface pra que o Java saiba quais métodos chamar.

Responder

novembro 6, 2015 às 2:18 PM

**neycandidoribeiro**

Grande Marcos,
Acredito que o Anônimo esteja certo. O código que está logo acima do diagrama da estrutura deveria ser a implementação de "Palio", como uma classe concreta implementando "Carro". Ali você repetiu a implementação da Fábrica FIAT.

E, parabéns pelo blog, estou desbravando ele aqui. A Chain of Responsibility está especialmente fantástica!

novembro 6, 2015 às 3:12 PM

**Marcos Brizeno**

Ah, entendi, agora que reparei o problema. Corrigido, valeu pelos avisos!

6. julho 19, 2014 às 1:39 AM
**Rodrigo Santana Porto**
Acho que a intenção ali na parte final era de colocar os produtos Palio, Gol, Celta e Fiesta e você acabou repetindo código da criação das fábricas, dá uma conferida ai.

Responder

julho 20, 2014 às 10:38 AM
**Marcos Brizeno**
Valeu Rodrigo! Dei uma lida de novo no post, a ideia no último código é mostrar como uma fábrica concreta vai criar um produto concreto. Por isso tem o código da FabricaFiat e o método criarCarro() retornando um new Palio(), sempre respeitando a interface comum.

Responder

7. setembro 30, 2014 às 2:59 PM
**Leo**
Olá Marcos, parabéns tbm pelo post. Só fiquei com uma dúvida. A classe FabricaDeCarro não teria que ter pelo menos uma ligação de dependencia com a classe Carro no diagrama, ja que usou ela em um método ? De onde vem aquele Carro ??? Desde já obrigado !!

Responder

setembro 30, 2014 às 6:32 PM
**Marcos Brizeno**
Sim Leo, acho que como a classe Fábrica instancia um Carro seria interessante ter uma ligação no diagrama. Vou tentar arrumar um tempo pra atualizar o post, obrigado!

Responder

8. novembro 12, 2014 às 9:09 AM
**Fernando**
Marcos, primeiro parabéns pela série. Assunto interessante e importante, e posts muito bem escritos. Em segundo, uma dúvida. Pelo o que entendi, no segundo exemplo (cada fábrica gerando seu próprio carro), para instanciar um carro eu preciso saber qual carro estou criando? Já que eu preciso iniciar a criação de uma fábrica concreta. Por exemplo, se for criar um Palio, eu devo instanciar a FabricaFiat e solicitar a criação do seu carro. Correto?

Responder

novembro 12, 2014 às 9:31 AM
**Marcos Brizeno**
Obrigado Fernando, fico feliz que tenha gostado ☺
Sim, sua observação está correta. A classe FabricaFiat está lá só pra deixar o código mais interessante, mas é possível deixar uma única classe de Fabrica para criar todos os produtos.
A classe FabricaFiat fica mais interessante quando você tem vários produtos para serem criados, como no padrão Abstract Factory: https://brizeno.wordpress.com/2011/09/18/mao-na-massa-abstract-factory/

Responder

novembro 13, 2014 às 10:06 AM
**Fernando**
Era o que tinha entendido mesmo. Li hoje o posto da Abstract Factory, e realmente tornou o cenário mais interessante.

9. novembro 25, 2014 às 8:37 PM
**Vinicius Rocha**

Muito legal cara eu sou Vinicius Rocha e faço computação na Univali aqui em floripa.

Estou fazendo OO2 e implementando um jogo de naves em java.

Muito obrigado pelos exemplos estão me ajudando muito no jogo e na implementação de um sistema Zend framework com asterisk o qual eu trabalho.

Parabens.

Responder

novembro 25, 2014 às 8:44 PM

**Marcos Brizeno**

Valeu pelo comentário. Fico feliz que tenha gostado!

Responder

10. março 30, 2015 às 2:52 PM

**Anônimo**

Não deveria ter um método recebendo um parametro para decidir qual objeto?

Responder

março 30, 2015 às 2:59 PM

**Marcos Brizeno**

Essa é uma das ideias de implementação do padrão, a fábrica recebe um parâmetro e decide qual objeto instanciar.
A ideia apresentada aqui é que, ao invés de passar um parâmetro, o cliente chamaria diretamente a fábrica para instanciar o produto, diminuindo as indireções e deixando o código mais claro.

Responder

11. julho 2, 2015 às 8:40 PM

**Anônimo**

ótima explicação!! Obrigado!

Responder

12. setembro 7, 2015 às 8:09 PM

**Santos**

Olá, Marcos. Parabéns pelo exemplo didático.
Tenho uma questão: pereceba que, no primeiro exemplo do factory method, o que usa um enum, o comportamento de criar uma instância fica completamente relacionado ao processo de execução. Ou seja, se em sua app você tiver um controle que define qual enum será passado como parâmetro para a fábrica, ela analisará (switch) quem será instanciado. Porém, como você mesmo afirmou, caso eu precise estender o código com um novo carro (Camaro, por exemplo), eu quebro a recomendação do Open/Close do SOLID. No segundo exemplo, você tem a opção de estender o seu código sem que seja necessário, teoricamente, modificar nada. Por que teoricamente? Porque, caso você precise mudar o objeto a ser criado, você tem que mudar no momento da criação; no momento em que o objeto precisa ser criado.
Então… com base no que eu escrevi, como seria uma implementação que tivesse a independência para a criação da primeira opção e a extensibilidade da segunda opção?
Espero ter sido claro.
Abraço.

Responder

setembro 24, 2015 às 1:16 PM

**Marcos Brizeno**

Oi Santos. Obrigado pelos pontos levantados.
Ter um código que contempla todos os princípios SOLID é bem difícil pois, em algum momento, os objetos precisam falar uns com os outros e nem sempre é possível deixar isso bonitinho e simples. A ideia é dividir essa complexidade pra que as alterações futuras tenham o menor impacto possível.
Os princípios são só princípios e não regras.
Em qualquer situação, é necessário que algum objeto tome a decisão de qual produto criar e ai não dá pra fugir muito.
Uma saída para o problema que você levantou, de ter independência de criação e extensibilidade, é tomar essa decisão uma só vez e em um só lugar.
Se o switch estiver encapsulado e centralizado em uma classe, beleza, podemos viver com isso. O problema vem quando essa decisão é (re)feita várias vezes dentro do mesmo fluxo.

Responder

13. novembro 16, 2015 às 10:25 PM

**Giuliana**
Muito bom! Bem direto e simples de entender. A adição de exemplos para consolidar a teoria ajudou muito!

Responder

14. julho 6, 2017 às 11:02 PM

**Fabiano Góes**
Marcos, primeiro parabéns pelo belo post e obrigado por compartilhar.
Sobre a primeira implementação: qual o problema você vê em usar Reflaction para construir os objetos dinamicamente?
Alguma coisa do tipo:

```
public Carro criaCarro(ModeloCarro modelo){
Carro carro = null;
try {
Class c = Class.forName("factorymethod." + modelo.name());
carro = (Carro)c.newInstance();
} catch (ClassNotFoundException | InstantiationException | IllegalAccessException e) {
throw new UnsupportedOperationException(e.getMessage());
}
return carro;
}
```

Responder

julho 7, 2017 às 7:11 AM

**Marcos Brizeno**
Oi Fabiano, obrigado pelo comentário! Fico feliz que tenha gostado do conteúdo.
Sobre o uso de Reflection eu não vejo problema nenhum, se isso resolve o problema no seu contexto, então tá tudo certo ☺
Mas a minha experiência com Reflection – e metaprogramação em geral – eu começaria a me preocupar caso fosse preciso adicionar lógica ai nesse método (por exemplo caso um tipo especial de fábrica precise ser tratado de maneira diferente do resto). Metaprogramação é muito bom pra reduzir código inútil, mas quando ela começa a ter lógica de negócio, ai fica tenso.

Responder

15. novembro 19, 2017 às 1:06 AM

**Marcus Java**
– Não pode repetir atributos (Ex: existe ExibirInfo e CriarCarro repetidas vezes em várias classes).
E melhor usar Herança para construir esse diagrama…

- Responder
  - dezembro 5, 2017 às 2:16 AM

    **Leinylson Fontinele Pereira**

    Não entendi muito o comentário, ExibirInfo e CriarCarro são métodos. Eles estão sobrecarregando, por isso aparecem várias vezes, mas cada um de uma maneira diferente.

    - Responder

16. março 30, 2019 às 10:26 AM

    **Anônimo**

    muito boa a Explicação!!

    - Responder

__PRESENT