Skip to navigation

Marcos Brizeno

Desenvolvimento de Software #showmethecode

setembro 18, 2011

Mão na massa: Abstract Factory

Falamos no post anterior sobre o famoso Factory Method (http://wp.me/p1Mek8-1c), agora vamos demonstrar o Abstract Factory!

Problema

Vamos utilizar como o problema o mesmo discutido no post sobre o Factory Method (http://wp.me/p1Mek8-1c). Queremos representar um sistema que, dado um conjunto de carros deve manipulá-los. A diferença é que, desta vez, precisamos agrupar os carros em conjuntos. A ideia de conjuntos é agrupar objetos que tem comportamentos parecidos. Para exemplificar veja como os objetos devem ser organizados:

Sedan:

- Siena – Fiat
- Fiesta Sedan – Ford

Popular:

- Palio – Fiat
- Fiesta – Ford

Ou seja, agora precisamos agrupar um conjunto de carros Sedan e outro conjunto de carros Popular para cada uma das fábricas.

Se considerarmos os carros Sedan como um produto, poderíamos criar uma classe produto para cada novo carro e, consequentemente uma classe fábrica para cada novo produto.

O problema é que, desta forma não teríamos a ideia de agrupamento dos produtos. Por exemplo, para criar os carros da Fiat precisaríamos de uma fábrica de carros populares da Fiat e outra fábrica de carros sedan da Fiat. Essa separação dificultaria tratar os carros de uma mesma marca.

Então surge o padrão Abstract Factory, que leva a mesma ideia do Factory Method para um nível mais alto.

Abstract Factory

Vejamos a intenção do Padrão Abstract Factory:

"Fornecer uma interface para criação de famílias de objetos relacionados ou dependentes sem especificar suas classes concretas."[1]

Então, de acordo com a descrição da intenção do padrão, nós poderemos criar famílias de objetos, no nosso exemplo seriam a família de carros Sedan e a família de carros Populares. Sem mais demora vamos ao código.

Inicialmente vamos escrever a classe interface para criação de Fábricas. Cada fabrica vai criar um objeto de cada tipo, ou seja, para o nosso exemplo, precisaremos de dois métodos fábrica, um para carros Sedan e outro para carros Populares:

```
1  public interface FabricaDeCarro {
2      CarroSedan criarCarroSedan();
3      CarroPopular criarCarroPopular();
4  }
```

E agora vamos escrever as classes que vão criar os carros de fato:

```
 1  public class FabricaFiat implements FabricaDeCarro {
 2
 3      @Override
 4      public CarroSedan criarCarroSedan() {
 5          return new Siena();
 6      }
 7
 8      @Override
 9      public CarroPopular criarCarroPopular() {
10          return new Palio();
11      }
12
13  }
```

Pronto! Todas as outras fábricas precisam apenas implementar esta pequena interface. Vamos então para o lado do produto. Como já comentamos os produtos são divididos em dois grupos, Sedan e Popular, em que cada um deles possui um conjunto de atributos e métodos próprios. Para o nosso exemplo vamos considerar que existe um método para exibir informações de um carro Sedan e outro para exibir informações de carros Populares. As interfaces seriam assim:

```
1  public interface CarroPopular {
2      void exibirInfoPopular();
3  }
```

```
1  public interface CarroSedan {
2      void exibirInfoSedan();
3  }
```

Apesar de os métodos basicamente executarem a mesma operação, diferindo apenas no nome, suponha que os carros Populares estão em um banco de dados e os carros Sedan em outros, assim cada método precisaria criar sua própria conexão.

Agora vamos ver os produtos concretos:

```
1  public class Palio implements CarroPopular {
2
3      @Override
4      public void exibirInfoPopular() {
5          System.out.println("Modelo: Palio\nFábrica: Fiat\nCategoria:Popular
6      }
7
8  }
```

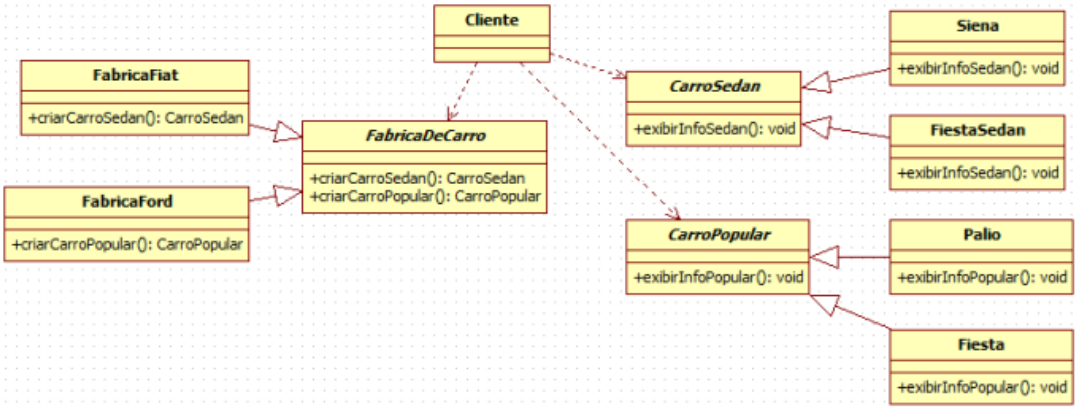
```
1  public class Siena implements CarroSedan {
2
3      @Override
4      public void exibirInfoSedan() {
5          System.out.println("Modelo: Siena\nFábrica: Fiat\nCategoria:Sedan"
6      }
7
8  }
```

Pronto, definido as fábricas e os produtos vamos analisar o código cliente:

```
1   public static void main(String[] args) {
2       FabricaDeCarro fabrica = new FabricaFiat();
3       CarroSedan sedan = fabrica.criarCarroSedan();
4       CarroPopular popular = fabrica.criarCarroPopular();
5       sedan.exibirInfoSedan();
6       System.out.println();
7       popular.exibirInfoPopular();
8       System.out.println();
9
10      fabrica = new FabricaFord();
11      sedan = fabrica.criarCarroSedan();
12      popular = fabrica.criarCarroPopular();
13      sedan.exibirInfoSedan();
14      System.out.println();
15      popular.exibirInfoPopular();
16  }
```

Perceba que criamos uma referência para uma fábrica abstrata e jogamos nela qualquer fábrica, de acordo com o que necessitamos. De maneira semelhante criamos referências para um carro Popular e para um carro Sedan, e de acordo com nossas necessidades fomos utilizando os carros dos fabricantes.

Essa é a estrutura do projeto de fábricas de carros Sedan e Popular:



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

Um pouco de teoria

Como você pode notar, este padrão sofre do mesmo problema do Factory Method, ou seja criamos uma estrutura muito grande de classes e interfaces para resolver o problema de criação de objetos. No entanto, segundo o princípio da Segregação de Interface [2] isto é uma coisa boa, pois o nosso código cliente fica dependendo de interfaces simples e pequenas ao invés de depender de uma interface grande e que nem todos os métodos seria utilizados.

Para exemplificar suponha que criássemos apenas uma interface para carros, nela precisaríamos definir os métodos de exibir informações tanto para Populares quanto para Sedan. O porblema é que, dado um carro qualquer não dá pra saber se ele é um sedan ou um popular. E o que fazer para implementar o método de Populares em carros Sedan (e vice-versa)?

Ou seja, criar várias interfaces simples e pequenas é melhor do que uma interface faz-tudo. Então qual o problema com o padrão?

Pense como seria se tivéssemos que inserir uma nova família de produtos, por exemplo Sedan de Luxo. Do lado dos produtos apenas definiríamos a interface e implementaríamos os produtos. Mas do lado das fábricas será necessário inserir um novo método de criação em TODAS as fábricas. Tudo bem que o método de criação no exemplo é bem simples, então seria apenas tedioso escrever os códigos para todas as classes.

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.

Abstract Factory, Padrões de Projeto Abstract Factory, Java, Padrões, Projeto 22 Comentários

22 comentários sobre “Mão na massa: Abstract Factory”

1. setembro 20, 2011 às 9:42 PM
   Gizelle Pauline
   Muito massa a explicação, Marquinhos! ;D
   Essa sua iniciativa com certeza está ajudando muita gente.

   Responder
2. abril 2, 2013 às 2:40 PM
   Augusto Cesar Nunes
   Excelentes artigos, Marcos! Parabéns! Realmente está ajudando bastante a entender os conceitos de Design Patterns.

   Estou começando a estudar Padrões de Projeto e tenho uma dúvida quanto à utilização dos mesmos em uma situação real: no caso de um sistema onde existam diversa possibilidade de layouts (JPanels, JTextAreas, JLabels, etc.), que podem ser arranjados (com repetição), à critério do usuário, que padrão de projetos deveria ser utilizado para construir estes diversos layouts? Abstract Factory? Composite? Decorator? Buider?

   Agradecendo antecipadamente sua ajuda,

   Augusto Cesaer

   Responder
   abril 3, 2013 às 6:19 AM
   marcosbrizeno
   Com certeza o Composite ajudaria bastante, já que você provavelmente vai organizar elementos dentro de elementos. O Java Swing inclusive já faz isso.
   Quanto aos outros ai depende. Se o usuário tiver uma tela específica onde ele pode customizar a interface, então talvez um builder seja interessante, pois você vai apenas adicionando os elementos e quando o processe tiver concluído, o builder vai te dar toda a informação sobre a interface.
   Factories talvez ajudem, mas depende do projeto. Não sei quão melhor seria a factory ou o construtor.
   É provável também que o framework que você utilize, já faça uso de vários padrões de projeto internamente.
   Espero ter ajudado, Obrigado!

   Responder

3. agosto 10, 2013 às 3:20 PM
Luis Ricardo
Parabéns, muito boa sua explicação. Tirei bastante proveito desse seu artigo.

Responder
4. setembro 16, 2013 às 7:16 PM
Andrey
Parabéns! Só uma correção, Siena está para carro Sedan, assim como Palio está para carro Hatch

Responder
5. outubro 3, 2013 às 12:34 AM
Leinylson Fontinele
Simplesmente excelente!

Responder
6. janeiro 30, 2014 às 2:05 PM
Anônimo
Muito bom ajudou bastante, mais em FabricaFiat→criar CarroSedan() não faltou o método criar CarroPopular() e em FabricaFord → criar CarroPopular() não faltou o método criar CarroSedan().

Responder
7. dezembro 9, 2014 às 8:52 PM
Gustavo Corso Ribeiro
Boa cara, finalmente caiu a ficha deste padrão! 😀

Responder
8. julho 10, 2015 às 11:41 AM
Anônimo
No diagrama das Fabricas esta faltando método da interface, no mais muito bom, parabéns.

Responder
9. julho 24, 2015 às 5:26 PM
W
"…No caso de uma nova família de produtos, por exemplo sedan de Luxo…"
Uma possível alternativa a isso seria criar um método
default na interface FrabricaDeCarro e fazer um "hook" com a(s)
fábrica(s) de que consegue fabricar sedans de luxo (considerando que
nem todas as fábricas fabriquem este tipo de carro). Dependendo do
tamanho do código já implementado, isso salvaria um bom tempo; pois
ás fábricas com a capacidade de fabricarem sedans de luxo poderiam
substituir esse método e as outras não teriam essa obrigatoriedade.

Responder
10. agosto 9, 2015 às 1:55 PM
pedroinacreditavel
Republicou isso em Pensamentos de Programador.

Responder
11. março 15, 2016 às 11:26 AM
André
Muito bom, Marcos, obrigado!

Responder
12. novembro 21, 2016 às 10:25 PM

Leonardo
Bem detalhado, obrigado.

Responder

13. fevereiro 23, 2017 às 10:30 AM

João Guedes
Marcos,

Seus posts são fantásticos e tem me ajudado bastante dos estudos, mas fiquei com uma dúvida sobre a diferença de Factory Method e Abstract Factory, e pesquisando encontrei a seguinte diferenciação:
– Abstract Factory:
AbstractFactory abstractFactory = new AbstractFactory();
IFactory factory = abstractFactory.create(tipo);
IObject object = factory.create(tipo);

– Factory Method
Factory factory = new Factory(tipo);
IObject object = factory.create(tipo);

Segundo esta diferenciação o AbstractFactory é uma fábrica de fábricas, onde os tipos das fábricas não devem ser criados diretamente, mas sim através da AbstractFactory. Já no FactoryMethod a fábrica pode ser diretamente instanciada já no tipo desejado. (O "I" utilizado em IFactory e IObject representam Interfaces mãe, implementadas por classes filhas, note que não utilizei IFactory no FactoryMethod).

O que acha?

Responder

fevereiro 23, 2017 às 11:14 AM

Marcos Brizeno
Oi João, faz sentido sim e é bem simples de entender. Eu escrevi mais sobre isso no meu livro https://www.casadocodigo.com.br/products/livro-refatoracao-ruby e no ebook grátis também https://leanpub.com/primeiros-passos-com-padroes-de-projeto.

Responder

14. junho 13, 2017 às 10:16 PM

Anônimo
Parabéns muito bom me ajudou muito no seminário na faculdade.

Responder

15. agosto 24, 2017 às 2:06 PM

Ygor
E se houvesse a necessidade de se criar mais de um modelo Sedan ou Popular em cada empresa?

Responder

agosto 24, 2017 às 2:22 PM

Marcos Brizeno
Oi Ygor, ai depende do seu contexto. Uma opção seria implementar novas classes que estendem CarroSedan e CarroPopular e deixar a lógica de qual objeto criar nas fábricas. Outra opção é modificar o design e mudar a abstração de CarroSedan/CarroPopular para outra coisa.

Responder

16. setembro 12, 2017 às 9:17 PM

Michel

E se por exemplo existisse uma terceira fábrica (FabricaVolks) que fabricasse somente carro popular. Como ficaria essa questão? Pois quando herdamos de FabricaDeCarro somos obrigados a implementar todos os métodos.

Responder

setembro 13, 2017 às 7:15 PM

Marcos Brizeno

Oi Michel!

Quando temos um esquema de herança onde alguns métodos não fazem sentido é um sinal de falha no design. No caso que você sugeriu o melhor seria rever o design e talvez esse esquema de fábricas não faça tanto sentido.

Responder

17. novembro 27, 2017 às 10:16 AM

Veridiana Melo

Qual o nome da classe do método main ? É cliente ?

Responder

18. março 30, 2019 às 10:31 AM

Anônimo

Adorei a explicação,Obrigado Amigo,Abç

Responder

__PRESENT