Marcos Brizeno

Desenvolvimento de Software #showmethecode

Padrões de Projeto

O livro Refatorando com Padrões de Projeto, um guia em Ruby (http://www.casadocodigo.com.br/products/livro-refatoracao-ruby) traz uma análise bem detalhada sobre como aplicar padrões de projeto através de pequenos passos de refatoração. Se você gosta da abordagem mão na massa, o livro tem o conteúdo certo para você!

Padrões de Projeto – Mão na Massa.

O objetivo desta série é mostrar ao leitor como os padrões podem ser utilizados para resolver problemas práticos, baseando-se na teoria descrita nos livros. Em cada padrão apresentado será sugerido um problema a ser resolvido utilizando o padrão. Em seguida um código será feito para exemplificar o problema e uma maneira de resolvê-lo. Ao final será feita uma análise da teoria por trás do padrão, suas vantagens e desvantagens, entre outros assuntos que venham a ser importantes.

A seguir a lista de padrões já demonstrados:

Criação

Factory Method (http://wp.me/p1Mek8-1c)

Criando objetos *on the fly* com alta flexibilidade!

Abstract Factory (http://wp.me/p1Mek8-1h)

Criando **famílias** de objetos *on the fly* com alta flexibilidade!

Builder (http://wp.me/p1Mek8-2a)

Construindo o produto passo-a-passo!

Prototype (http://wp.me/p1Mek8-51)

Criando objetos por cópia de uma instância!

Singleton (http://wp.me/p1Mek8-1Z)

Centralizando e compartilhando recursos!

Estrutura

Composite (http://wp.me/p1Mek8-M)

Tratando todos os objetos com justiça!

Adapter (http://wp.me/p1Mek8-2z)

Plugando conteúdo ao sistema!

Bridge (http://wp.me/p1Mek8-2K)

Separando implementações de abstrações em prol da flexibilidade!

Decorator (http://wp.me/p1Mek8-h)

Incrementando funcionalidades dinamicamente!

Facade (http://wp.me/p1Mek8-4c)

Simplificando a utilização de subsistemas complexos!

Proxy (http://wp.me/p1Mek8-2o)

Redirecionando o acesso aos objetos!

Flyweight (http://wp.me/p1Mek8-45)

Compartilhando pequenos recursos para economizar espaço!

Comportamento

Strategy (http://wp.me/s1Mek8-strategy)

Separando os dados dos algoritmos para alcançar a reusabilidade!

Iterator (http://wp.me/p1Mek8-15)

Percorrendo um conjunto de dados independente da implementação!

Template Method (http://wp.me/p1Mek8-1C)

Definindo algoritmos extensíveis!

Observer (http://wp.me/p1Mek8-2T)

Compartilhando recursos de maneira inteligente!

Mediator (http://wp.me/p1Mek8-3l)

Simplificando relacionamentos complexos!

Command (http://wp.me/p1Mek8-3y)

Transformando requisições em objetos!

Memento (http://wp.me/p1Mek8-3l)

Externalizando estados sem quebrar o encapsulamento!

Chain of Responsibility (http://wp.me/p1Mek8-3S)

Repassando requisições para evitar decisões!

Interpreter (http://wp.me/p1Mek8-4o)

Definindo uma gramática e um interpretador!

State (http://wp.me/p1Mek8-4y)

Simplificando a troca de estados internos de um objeto!

Visitor (http://wp.me/p1Mek8-4K)

Separando operações de estruturas!

Agradecimentos especiais ao pessoal da disciplina de Padrões de Projeto: Débora, Fabiano, Gizelle, Isac, Guilherme, Thaís e Wairton. Também ao Professor Jerffeson, que proporcionou encontros úteis para discussão do assunto.

17 comentários sobre “Padrões de Projeto”

1. outubro 17, 2013 às 11:37 AM
   Anônimo
   Muito boa a sua iniciativa. Parabéns pelo trabalho!

   Responder
2. fevereiro 13, 2014 às 11:35 PM
   Fernando Cardia
   Excelente! Estou aprendendo com muito mais facilidade! Parabéns!

   Responder
3. fevereiro 26, 2014 às 10:06 AM
   Daniel Cavalcante
   Belo blog. Ótimos artigos. Espero que continue postando!

   Responder
4. junho 19, 2014 às 7:04 PM
   Leo
   realmente muito bom!!

   Responder
5. novembro 17, 2014 às 3:35 PM
   Luis Gustavo Spern Barreto
   Fantástico!
   Nós alunos da faculdade SENAC-RS Pelotas estamos desenvolvendo um trabalho cujo um dos materiais de apoio são esses teus posts sobre padrões. Valeu!

   Responder
   - novembro 17, 2014 às 3:39 PM
     Marcos Brizeno
     Muito bom! Fico feliz que tenham gostado 🙂

     Responder
6. fevereiro 11, 2015 às 12:56 PM
   Ezequiel Godoy | Links interessantes
   […] https://brizeno.wordpress.com/padroes/ […]

   Responder
7. março 9, 2015 às 9:06 PM
   Thiago AS. Lopes
   Gratidão pelo seu tempo dedicado amigo, muito obrigado!

   Responder
8. abril 1, 2015 às 4:25 PM
   Anônimo
   Parabéns Marcos, obrigado por disponibilizar seu trabalho.

   Responder
9. abril 9, 2015 às 7:48 PM
   Iago
   Muito obrigado por disponiblizar um conteúdo com uma didática muito boa!

Responder

10. novembro 12, 2015 às 11:49 AM
    Meu primeiro livro publicado: Refatorando com Padrões de Projeto, um guia em Ruby | Marcos Brizeno
    […] a ideia da série Padrões de Projeto Mão na Massa, o livro aborda alguns dos padrões da famosa Gangue dos Quatro mas com uma dinâmica um pouco […]

    Responder
11. junho 23, 2016 às 9:37 PM
    lccezinha
    Parabéns pelo seu trabalho e seus posts, muito bem escritos e didáticos!

    Responder
12. dezembro 12, 2016 às 2:04 PM
    Paulo Henrique
    Excelente! Estou estudando para uma avaliação e é tudo que eu precisava!

    Responder
13. junho 30, 2017 às 12:54 PM
    Raphael Lacerda
    que post fodda !! congrats

    Responder
14. julho 6, 2017 às 12:45 PM
    Anônimo
    Parabéns, trabalho de extrema qualidade! Gostei bastante e tem ajudado bastante.

    Responder
15. julho 14, 2017 às 10:12 AM
    Anônimo
    Parabéns, as explicações são muito boas.

    Responder
16. dezembro 5, 2017 às 3:36 PM
    Jorge
    Excelente trabalho, otima didatica.

    Responder

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:

```
public class FabricaFiat implements FabricaDeCarro {

    @Override
    public CarroSedan criarCarroSedan() {
        return new Siena();
    }

    @Override
    public CarroPopular criarCarroPopular() {
        return new Palio();
    }

}
```

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:

```
public interface CarroPopular {
    void exibirInfoPopular();
}
```

```
public interface CarroSedan {
    void exibirInfoSedan();
}
```

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:

```
public class Palio implements CarroPopular {

    @Override
    public void exibirInfoPopular() {
        System.out.println("Modelo: Palio\nFábrica: Fiat\nCategoria:Popular");
    }

}
```

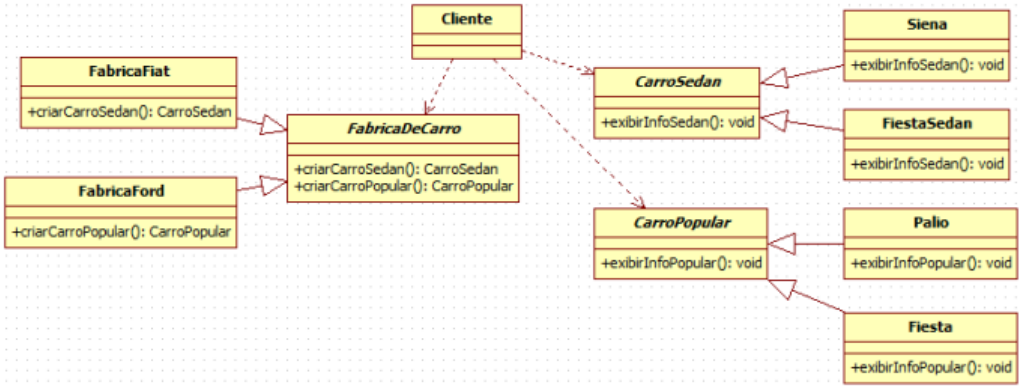
```
public class Siena implements CarroSedan {

    @Override
    public void exibirInfoSedan() {
        System.out.println("Modelo: Siena\nFábrica: Fiat\nCategoria:Sedan");
    }

}
```

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

```
public static void main(String[] args) {
    FabricaDeCarro fabrica = new FabricaFiat();
    CarroSedan sedan = fabrica.criarCarroSedan();
    CarroPopular popular = fabrica.criarCarroPopular();
    sedan.exibirInfoSedan();
    System.out.println();
    popular.exibirInfoPopular();
    System.out.println();

    fabrica = new FabricaFord();
    sedan = fabrica.criarCarroSedan();
    popular = fabrica.criarCarroPopular();
    sedan.exibirInfoSedan();
    System.out.println();
    popular.exibirInfoPopular();
}
```

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 21 Comentários

21 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

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:

```
 1  public abstract class FabricaCarro {
 2      public Carro criarCarro(ModeloCarro modelo) {
 3          switch (modelo) {
 4          case celta:
 5              return new Celta();
 6          case fiesta:
 7              return new Fiesta();
 8          case gol:
 9              return new Gol();
10          case palio:
11              return new Palio();
12          default:
13              break;
14          }
15      }
16  }
```

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:

```
1  public interface FabricaDeCarro {
2      Carro criarCarro();
3  }
```

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

```
1  public class FabricaFiat implements FabricaDeCarro {
2
3      @Override
4      public Carro criarCarro() {
5          return new Palio();
6      }
7
8  }
```

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:

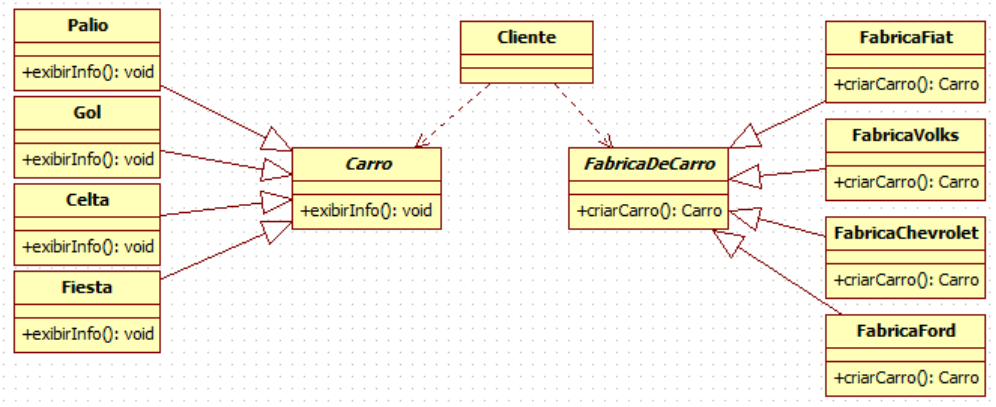
```
1  public interface Carro {
2      void exibirInfo();
3  }
```

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 35 Comentários

35 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

Marcos Brizeno

Desenvolvimento de Software #showmethecode

dezembro 5, 2011

Mão na massa: Prototype

Para encerrar a série de posts sobre os padrões de projeto, o padrão Prototype!

Problema:

O padrão Prototype é mais um dos padrões de criação. Assim seu intuito principal é criar objetos. Este intuito é muito parecido com todos os outros padrões criacionais, tornando todos eles bem semelhantes.

Para explicitar ainda mais esta semelhança vamos analisar o mesmo problema apresentado na discussão sobre o padrão Factory Method (http://wp.me/p1Mek8-1c).

O problema consiste em uma lista de carros que o cliente precisa utilizar, mas que só serão conhecidos em tempo de execução. Vamos analisar então como o problema pode ser selecionado utilizando o padrão Prototype.

Prototype

A intenção do padrão:

“Especificar tipos de objetos a serem criados usando uma instância protótipo e criar novos objetos pela cópia desse protótipo.” [1]

Pela intenção podemos perceber como o padrão vai resolver o problema. Precisamos criar novos objetos a partir de uma instância protótipo, que vai realizar uma cópia de si mesmo e retornar para o novo objeto.

A estrutura do padrão inicia então com definição dos objetos protótipos. Para garantir a flexibilidade do sistema, vamos criar a classe base de todos os protótipos:

```
1  public abstract class CarroPrototype {
2      protected double valorCompra;
3
4      public abstract String exibirInfo();
5
6      public abstract CarroPrototype clonar();
7
8      public double getValorCompra() {
9          return valorCompra;
10     }
11
12     public void setValorCompra(double valorCompra) {
13         this.valorCompra = valorCompra;
14     }
15 }
```

Definimos a partir dela que todos os carros terão um valor de compra, que será manipulado por um conjunto de getters e setters. Também garantimos que todos eles possuem os métodos para exibir informações e para realizar a cópia do objeto.

Para exemplificar uma classe protótipo concreta, vejamos a seguinte classe:

```
public class FiestaPrototype extends CarroPrototype {

    protected FiestaPrototype(FiestaPrototype fiestaPrototype) {
        this.valorCompra = fiestaPrototype.getValorCompra();
    }

    public FiestaPrototype() {
        valorCompra = 0.0;
    }

    @Override
    public String exibirInfo() {
        return "Modelo: Fiesta\nMontadora: Ford\n" + "Valor: R$"
                + getValorCompra();
    }

    @Override
    public CarroPrototype clonar() {
        return new FiestaPrototype(this);
    }

}
```

Note que no início são definidos dois construtores, um protegido e outro público. O construtor protegido recebe como parâmetro um objeto da própria classe protótipo. Este é o chamado construtor por cópia, que recebe um outro objeto da mesma classe e cria um novo objeto com os mesmos valores nos atributos. A necessidade deste contrutor será vista no método de cópia.

O método exibirInfo() exibe as informações referentes ao carro, retornando uma string com as informações. Ao final o método de clonagem retorna um novo objeto da classe protótipo concreta. Para garantir que será retornado um novo objeto, vamos utilizar o construtor por cópia definido anteriormente.

Agora, sempre que for preciso criar um novo objeto FiestaPrototype vamos utilizar um único protótipo. Pense nesta operação como sendo um método fábrica, para evitar que o cliente fique responsável pela criação dos objetos e apenas utilize os objetos.

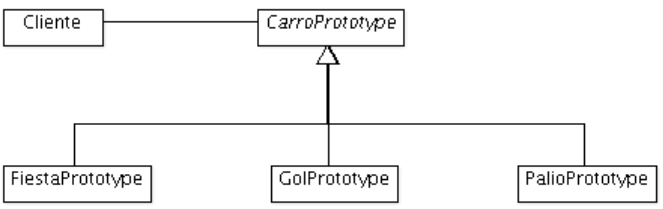
Para verificar esta propriedade, vamos analisar o código cliente a seguir, que faz uso do padrão prototype.

```
public static void main(String[] args) {
    PalioPrototype prototipoPalio = new PalioPrototype();

    CarroPrototype palioNovo = prototipoPalio.clonar();
    palioNovo.setValorCompra(27900.0);
    CarroPrototype palioUsado = prototipoPalio.clonar();
    palioUsado.setValorCompra(21000.0);

    System.out.println(palioNovo.exibirInfo());
    System.out.println(palioUsado.exibirInfo());
}
```

Observe com atenção o cliente. Criamos dois protótipos de carros, cada um deles é criado utilizando o prótipo PalioPrototype instanciado anteriormente. Ou seja, a partir de uma instância de um protótipo é possível criar vários objetos a partir da cópia deste protótipo.

Outro detalhe é que, se a operação de clonagem não fosse feita utilizando o construtor de cópia, quando a chamada ao setValorCompra fosse feita, ela mudaria as duas instâncias, pois elas referenciariam ao mesmo objeto.

O diagrama UML que representa a estrutura do padrão Prototype utilizada nesta solução é o seguinte:



(https://brizeno.files.wordpress.com/2011/12/prototype.png)

Um pouco de teoria

Inicialmente é fácil ver que o padrão Prototype oferece as mesmas vantagens que outros padrões de criação: esconde os produtos do cliente, reduz o acoplamento e oferece maior flexibilidade para alterações nas classes produtos.

A diferença básica deste padrão é a flexibilidade. Por exemplo: o cliente instancia vários protótipos, quando um deles não é mais necessário, basta removê-lo. Se é preciso adicionar novos protótipos, basta incluir a instanciação no cliente. Essa flexibilidade pode ocorrer inclusive em tempo de execução.

O padrão Prototype também poderia fazer uso do Abstract Factory (http://wp.me/p1Mek8-1h). Imagine que uma classe instância todos os protótipos e oferece métodos para copiar estes protótipos, ela seria uma fábrica de famílias de produtos.

Os produtos do Prototype podem ser alterados livremente apenas mudando os atributos, como no exemplo onde criamos um palio novo e um palio usado. No entanto é preciso garantir que o método de cópia esteja implementado corretamente, para evitar que a alteração nos valores mude todas as instâncias.

O padrão Prototype leva grande vantagem quando o processo de criação de seus produtos é muito caro, ou mais caro do que uma clonagem. No lugar de criar um Proxy (http://wp.me/p1Mek8-2o) para cada produto, basta definir, no objeto protótipo, como será essa inicialização, ou parte dela.

Um detalhe que torna o Prototype único em relação aos outros padrões de criação é que ele utiliza objetos para criar os produtos, enquanto os outros utilizam classes. Dependendo da arquitetura ou linguagem/plataforma do problema, é possível tirar vantagem deste comportamento.

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.

Prototype Java, Padrões, Projeto 7 Comentários

7 comentários sobre “Mão na massa: Prototype”

1. junho 5, 2012 às 6:42 PM
   João Gilberto
   Muito bom! Parabéns!

   Responder
2. maio 27, 2014 às 6:27 PM
   brunobolandini
   Seria adequado para criação de um baralho?
   O cliente não pode ter acesso e apenas uma carta de cada tipo deve ser criada no baralho?
   Ou teria que instaciar 52 subclasses ficando assim sem um bom aproveitamento?

   Responder
   maio 28, 2014 às 8:42 AM
   Marcos Brizeno
   Acho que esse seria um bom uso sim. A classe protótipo teria atributos compartilhados pelas cartas e cada “clone” teria seus próprios atributos, como no exemplo do Palio novo e usado.

   Responder
3. fevereiro 20, 2015 às 1:50 PM
   Leandro Martins
   Olá Marcos, muito boa a explicação. Só fiquei com uma dúvida quando você diz que:

“Um detalhe que torna o Prototype único em relação aos outros padrões de criação é que ele utiliza objetos para criar os produtos, enquanto os outros utilizam classes. ”

Eu li seu artigo sobre Abstract Factory, e pelo que entendi, você também cria um produto (objeto carro sedan ou popular) a partir de um objeto já existente, chamando o método abstrato do objeto “fabrica” do tipo FabricaDeCarro.

Sendo assim, o Abstract Factory teria essa mesma característica do Prototype, ou não ?
Vlw!

Responder

fevereiro 20, 2015 às 6:28 PM

Marcos Brizeno

Oi Leandro, que bom que você gostou.
Você está certo, a classificação do Prototype e do Abstract Factory é a mesma, são padrões de Criação. Isso é por que os dois resolvem o mesmo tipo de problema, gerenciar a criação de objetos.
A diferença é em qual contexto cada um é melhor utilizado.

Responder

4. junho 13, 2016 às 8:54 PM

André Celestino

Olá, Marcos.

Acompanho o seu site já há algum tempo. Parabéns pelo conteúdo!
Eu estou fazendo um estudo detalhado sobre cada padrão de projeto e me deparei com o Prototype. Fiquei com uma dúvida: em que situação real este padrão se encaixaria?

Responder

junho 14, 2016 às 12:05 PM

Marcos Brizeno

Oi André, então situações reais eu particularmente não encontrei ainda. Talvez seja um padrão que é mais utilizado por frameworks e bibliotecas, vale a pena investigar.

Se puder compartilhar as informações que você conseguir depois seria muito bom também 🙂

Obrigado!

Responder

Marcos Brizeno

Desenvolvimento de Software #showmethecode

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 (http://wp.me/p1Mek8-1c) e o Abstract Factory (http://wp.me/p1Mek8-1h), 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 (http://wp.me/p1Mek8-1c) e o Abstract Factory (http://wp.me/p1Mek8-1h) 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:

```
public abstract class CarroBuilder {

    protected CarroProduct carro;

    public CarroBuilder() {
        carro = new CarroProduct();
    }

    public abstract void buildPreco();

    public abstract void buildDscMotor();

    public abstract void buildAnoDeFabricacao();

    public abstract void buildModelo();

    public abstract void buildMontadora();

    public CarroProduct getCarro() {
        return carro;
    }
}
```

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 (http://wp.me/p1Mek8-1C), 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:

```
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.

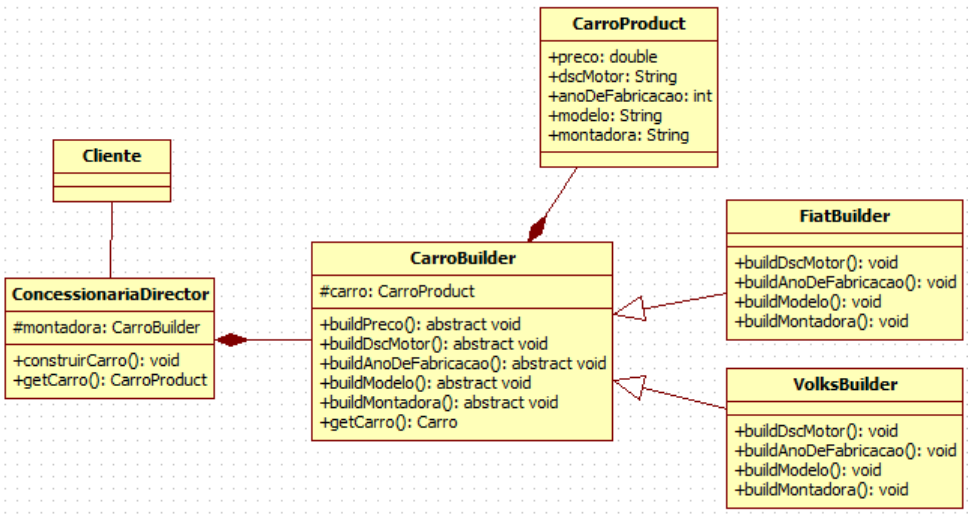
```java
public class ConcessionariaDirector {
    protected CarroBuilder montadora;

    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.

```java
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:



(https://brizeno.files.wordpress.com/2011/09/builder.png)

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 (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

20 comentários sobre “Mão na massa: Builder”

1. maio 15, 2013 às 9:47 PM
   Alan Arantes
   Baixei o Arquivo e não encontrei o Builder!!!!!! 😟

   Responder
   maio 16, 2013 às 7:39 AM
   marcosbrizeno
   Boa! Vou tentar atualizar esse final de semana. Desculpa ai.

   Responder
2. julho 4, 2013 às 10:22 AM
   Felipe
   O Builder ainda não esta la, poderia adicioná-lo Marcos, por favor!

   Responder
   julho 8, 2013 às 6:17 AM
   marcosbrizeno
   Pronto, adicionei ontem ao repositório. Valeu!

   Responder
3. outubro 2, 2013 às 11:48 AM
   Fábio Luciano

Parabens!! Irei fazer uma prova de Padrões de criação e essas suas postagens me exclareceu e muito. E vou usar as outras para estudar para as proximas provas.

Responder

outubro 2, 2013 às 4:41 PM

Marcos Brizeno

Massa, fico feliz em ter ajudado!

Responder

4. março 31, 2014 às 2:23 PM

lgustavolol

A sua explicação é muito boa, porém você usa o nome 'montadora' quando fala de 'modelos de carro'. Fico muito confuso com isso, por exemplo: […] public class FiatBuilder extends CarroBuilder […], nesse caso eu fico pensando: – será que ele está falando de carro ou de montadora? Preciso reler os exemplos muitas vezes para entender, porém ainda fico em dúvida. Nesse caso, eu preciso de um builder para cada montadora ou para cada modelo de carro? Se eu tiver os carros: Uno, Palio e Siena, preciso criar um builder para cada um, ou o builder da montadora Fiat já resolve o problema? Senão, como seria?

Acho que esse é um ponto a melhorar, no mais, está excelente a série!

Responder

março 31, 2014 às 7:03 PM

Marcos Brizeno

Verdade. Mas acho que todo domínio sempre é um pouco confuso, mesmo que em um exemplo 🙂
Valeu pelo feedback, vou atentar pra isso nas próximas postagens.

Responder

5. novembro 19, 2014 às 7:38 PM

Wellington Rodrigues

Marcos,

Analisando um diagrama sobre Builder na wikipedia (http://pt.wikipedia.org/wiki/Builder#mediaviewer/File:Builder_UML_class_diagram.svg) observei que a classe Director não recebe em seu construtor a classe/interface da fábrica de objetos. Porque seu exemplo está diferente?

Responder

novembro 20, 2014 às 8:59 AM

Marcos Brizeno

A implementação do padrão muda um pouco dependendo de qual funcionalidade da linguagem você utiliza e até mesmo de qual linguagem utilizada. Mesmo com uma pequena mudança no contexto de aplicação do padrão ainda é possível utilizá-lo. O bom de você ter olhado exemplos em outros lugares é que você pode comparar os dois e extrair a essência do padrão. Não dá pra criar um diagrama igual para todas as utilizações dos padrões (em alguns casos talvez), sempre o contexto precisa ser levado em consideração.

Responder

fevereiro 9, 2015 às 9:02 PM

Sérgio Alves

Preciso saber se é possível a classe cliente passar parâmetros para a classe diretor, se sim, como fazer isso? Pode ser através do método construirCarro?

6. janeiro 3, 2015 às 3:32 PM

Sérgio Alves

Preciso saber se é possível a classe cliente passar parâmetros para a classe diretor, se sim, como fazer isso? Pode ser através do método construirCarro?

Responder

fevereiro 12, 2015 às 9:05 AM

Marcos Brizeno

Os parâmetros podem ser passados tanto no método construirCarro quanto no próprio construtor do diretor. Só lembre que a ideia é deixar a classe diretor o mais simples possível.

Responder

7. dezembro 11, 2016 às 2:55 PM

Rafael - Rafa

Marcos, observando este padrão e o Abstract Factory, consegui imaginar como seria a integrações de ambos, no caso, o Builder para definir os atributos do objeto e o Abstract Factor para agrupá-los por fatores gerais comuns. Isso está correto, ou quando se usa um não faz sentido usar outro?

Responder

dezembro 12, 2016 às 9:20 AM

Marcos Brizeno

Oi Rafael, não tem problemas em utilizar dois padrões o único problema é que você vai ter muito código pra fazer pouca coisa. O ideal é tentar minimizar o uso de padrões para simplificar o seu design.
No exemplo que você deu acho que seria melhor tentar criar vários builders (caso os parâmetros mudem bastante) ou vários métodos fábrica (caso os parâmetros não mudem muito mas ainda assim precise de variações do objeto).
Eu escrevi um pouco sobre como evoluir com padrões de projeto aqui: https://leanpub.com/primeiros-passos-com-padroes-de-projeto/read.

Responder

8. janeiro 1, 2017 às 9:50 AM
Dênis Schimidt de Oliveira
Marcos, estava observando esse padrão e seria possível dentro desse builder termos classes de serviços e DAOs por exemplo já que ele realiza coisas complexas? Outro ponto é o que chamam de builder hoje no qual o cliente acessa diretamente o builder diretamente com Fluent Interface para criar um objeto. Para mim nesse caso talvez devessem até chamar de outro nome pois é muito diferente do padrão do GOF né. O que você acha?

Responder

janeiro 1, 2017 às 4:43 PM
Marcos Brizeno
Oi Dênis, o ponto principal do Builder é ter maior flexibilidade com a criação de objetos, então a pergunta que você precisa fazer é: Meu DAO (ou qualquer que seja o objeto) precisa dessa flexibilidade?
Muita gente utiliza o builder sem precisar dessa flexibilidade, só pra ter um Fluent Interface e evitar um construtor gigante (como o que você comentou). Mas isso na verdade só esconde o problema de ter um objeto com muitas dependências e nem é uma aplicação do Builder.

Responder

9. março 29, 2017 às 11:55 AM
3 dicas para utilizar Padrões de Projeto | Marcos Brizeno
[…] dificultar sua implementação quanto dificultar o entendimento dos outros. O coitado do Builder que o […]

Responder

10. novembro 17, 2017 às 10:11 PM
Veridiana Melo
Marcos qual é o nome da classe que tem o main ?

Responder

novembro 27, 2017 às 9:12 AM
Marcos Brizeno
Oi Veridiana, o nome da classe tanto faz na verdade, pode ser Application, Main ou qualquer outra coisa 🙂

Responder

Marcos Brizeno

Desenvolvimento de Software #showmethecode

setembro 24, 2011

Mão na massa: Singleton

Neste post apresentaremos o tão famoso padrão Singleton!

Problema

Como já vimos antes no padrões Abstract Factory (http://wp.me/p1Mek8-1h) e Factory Method (http://wp.me/p1Mek8-1c) é possível criar um objeto que fique responsável por criar outros objetos. Desta maneira nós centralizamos a criação destes objetos e podemos ter mais controle sobre eles.

Imagine o exemplo da fábrica de carros do padrão Factory Method (http://wp.me/p1Mek8-1c). A classe fábrica centraliza a criação de objetos carro. Por exemplo, se fosse necessário armazenar quantos carros foram criados, para elaborar um relatório de quais foram os carros mais vendidos, seria bem simples não? Bastaria adicionar um contador para cada tipo de carro e, ao executar o método que cria um carro, incrementar o contador referente a ele.

Vamos então ver como ficaria a nossa classe fábrica, para simplificar, apenas retornamos uma String para dizer que o carro foi criado:

```
1  public class FabricaDeCarro {
2      protected int totalCarrosFiat;
3      protected int totalCarrosFord;
4      protected int totalCarrosVolks;
5
6      public String criarCarroVolks() {
7          return new String("Carro Volks #" + totalCarrosVolks++ + " criado.");
8      }
9
10     public String criarCarroFord() {
11         return new String("Carro Ford #" + totalCarrosFord++ + " criado.");
12     }
13
14     public String criarCarroFiat() {
15         return new String("Carro Fiat #" + totalCarrosFiat++ + " criado.");
16     }
17
18     public String gerarRelatorio() {
19         return new String("Total de carros Fiat vendidos: " + totalCarrosFiat
20                 + "\nTotal de carros Ford vendidos: " + totalCarrosFord
21                 + "\nTotal de carros Volks vendidos: " + totalCarrosVolks);
22     }
23
24 }
```

A cada carro criado nós incrementamos o contador e exibimos a informação que o carro foi criado. No final adicionamos um método que gera um relatório e mostra todas as vendas.

O código cliente seria algo do tipo então:

```
1  public static void main(String[] args) {
2      FabricaDeCarro fabrica = new FabricaDeCarro();
3      System.out.println(fabrica.criarCarroFiat());
4      System.out.println(fabrica.criarCarroFord());
5      System.out.println(fabrica.criarCarroVolks());
6
7      System.out.println(fabrica.gerarRelatorio());
8  }
```

Uma excelente solução não? Agora imagine que, em algum outro lugar do código acontece isso:

```
fabrica = new FabricaDeCarro();
System.out.println(fabrica.gerarRelatorio());
```

Todos os dados até o momento foram APAGADOS! Todas as informações estão ligadas a uma instância, quando alteramos a instância, perdemos todas as informações. Qual o centro do problema então?

Temos que proteger que o objeto seja instanciado. Como fazer isso?

Singleton

A intenção do padrão é esta:

"Garantir que uma classe tenha somente uma instância e fornece um ponto global de acesso para a mesma." [1]

Pronto, achamos a solução! Com o uso do padrão garantimos que só teremos uma instância da classe fábrica.

O padrão é extremamente simples. Para utilizá-lo precisamos apenas de:

Uma referência para um objeto fábrica, dentro da própria fábrica:

```
public class FabricaDeCarro{
    public static FabricaDeCarro instancia;
}
```

Não deixar o construtor com acesso público:

```
public class FabricaDeCarro {

    public static FabricaDeCarro instancia;

    protected FabricaDeCarro() {
    }
}
```

E por fim um método que retorna a referência para a fábrica:

```
 public class FabricaDeCarro {

     public static FabricaDeCarro instancia;

     protected FabricaDeCarro() {
     }

     public static FabricaDeCarro getInstancia() {
         if (instancia == null)
             instancia = new FabricaDeCarro();
         return instancia;
     }
 }
```

Pronto, esta agora é uma classe Singleton. Ao proteger o construtor nós evitamos que esta classe possa ser instanciada em qualquer outro lugar do código que não seja a própria classe.

O código cliente ficaria da seguinte forma:

```
public static void main(String[] args) {
    FabricaDeCarro fabrica = FabricaDeCarro.getInstancia();
    System.out.println(fabrica.criarCarroFiat());
    System.out.println(fabrica.criarCarroFord());
    System.out.println(fabrica.criarCarroVolks());
    System.out.println(fabrica.gerarRelatorio());
}
```

Perceba que, mesmo que em outra parte do código apareça algo do tipo:

```
fabrica = FabricaDeCarro.getInstancia();
System.out.println(fabrica.gerarRelatorio());
```

Não será perdido nenhuma informação, pois não houve uma nova instanciação. O método getInstancia() verifica se a referência é válida e, se preciso, instância ela e depois retorna para o código cliente.

Seria possível, no código cliente, nem mesmo utilizar uma referência para uma fábrica, veja o código a seguir:

```
System.out.println(FabricaDeCarro.getInstancia().gerarRelatorio());
```

Um pouco de teoria

Já mostramos em código que a maior vantagem do Singleton é unificar o acesso das instâncias. Além disso msotramos também que não é preciso nem mesmo criar referências para classes Singleton.

No C++ nós temos as tão temidas variáveis globais. Temidas não pela sua natureza, mas pelo uso que se faz de variáveis globais. As classes Singleton são uma excelente saída quando é necessário "globalizar" certos aspectos do programa.

Em concorrência é muito comum utilizar recursos compartilhados que precisam de um acesso unificado, geralmente utilizando monitores. Encapsular o recurso compartilhado em uma classe Singleton torna muito mais fácil sincronizar o acesso.

Outro fator bem positivo do padrão é que, ao mesmo tempo que nós unificamos o acesso aos recursos, nós compartilhamos todos eles! Lembre que não é necessário criar referências, ou seja, qualquer objeto/classe/método em qualquer lugar do seu programa tem acesso aos recursos.

Isto também pode ser visto como uma grande desvantagem, pois não é possível inibir o acesso a classe Singleton. Qualquer parte do código por chamar o método getInstance(), pois ele é estático, e ter acesso aos dados da classe.

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.

 Padrões de Projeto, Singleton Java, Padrões, Projeto, Singleton 8 Comentários

8 comentários sobre "Mão na massa: Singleton"

1. outubro 28, 2012 às 1:24 PM
   Glaubert
   Muito bom seu site. está me ajudando muito com seus exemplos, muito claro e objetivo. Parabéns!

    Responder
2. outubro 3, 2013 às 10:36 AM
   Leinylson Fontinele
   Muito boa suas explicações. Só duas observações, o singleton não está no pacote e o incremento dos contadores deve vir antes como informado abaixo, ou na declaração inicializar com 1(um):

   public String criarCarroVolks() {
   return new String("Carro Volks #" + ++totalCarrosVolks + " criado.");
   }

   public String criarCarroFord() {
   return new String("Carro Ford #" + ++totalCarrosFord + " criado.");
   }

   public String criarCarroFiat() {
   return new String("Carro Fiat #" + ++totalCarrosFiat + " criado.");
   }

Responder

3. agosto 9, 2015 às 2:47 PM
   Mão na massa: Singleton | Pensamentos de Programador
   […] Mão na massa: Singleton. […]

   Responder
4. novembro 21, 2015 às 8:06 PM
   anderson
   Amigo o codigo do singleton n ta disponivel lá nos padrões do git :s

   Responder
5. dezembro 29, 2015 às 8:32 AM
   Roberto Licciardo
   Parabéns, muito bons seus artigos. Uma observação: sendo o construtor “protected” qualquer classe derivada poderia instanciar os objetos.

   Responder
6. fevereiro 17, 2018 às 6:05 PM
   Antonio Lira
   Muito Bom o Post, parabéns,…
   sou novo e programação e gostaria de tirar uma duvida, tenho um projeto com dez classes preciso implementar o singleton em cada uma delas, ou crio uma classe singleton e crio instancio as dez na classe singleton ???

   Responder
   fevereiro 17, 2018 às 9:11 PM
   Marcos Brizeno
   Oi Antonio, valeu pelo comentário!
   Usar um ou vários singletons depende do que você quer alcançar. Se cada classe precisar de uma instância única, é melhor criar um singleton pra cada, já se a ideia é fornecer um ponto único de acesso a todas elas, daí criar um singleton só pode ser uma ideia melhor.
   Mas na verdade acho que vale repensar o design pois utilizar uma instância global geralmente não é uma boa ideia devido ao alto acoplamento que isso cria. Se você criar 10 delas imagino que vai ficar bem estrutural e você não vai conseguir se beneficiar da Orientação a Objetos pra ter flexibilidade no seu design.

   Responder
7. julho 16, 2018 às 10:02 PM
   Anônimo
   Marcos, boa noite.

   No padrão Singleton, a classe de criação deve ser PRIVATE e não PROTECTED. Senão, é possível instaciar várias com … = new …

   private FabricaDeCarro() {
   }

   Responder