Design Patterns - Factory Method

Fabricio Griebler, Manuela Valente, Marlon Lopes

[1] Especialização em Tecnologias Aplicadas a Sistemas de Informação com Métodos Ágeis - 4ª Edição
Uniritter
Porto Alegre, RS

fabricio@fasagri.com.br, valente.manu@gmail.com, marlonglopes@gmail.com

Resumo. *Este artigo apresenta uma descrição sucinta do padrão de projetos orientado a objetos chamado Factory Method (Método de Fabrica),*

Abstract. *This article presents a brief description of object-oriented project pattern called Factory Method.*

1. Introdução

1.1. O que é um padrão de design (*Design Pattern*)

Cada padrão descreve um problema que ocorre ao longo do tempo mais de uma vez em nosso meio, e então descreve o núcleo da solução para esse problema, de tal forma que você pode usar esta solução um milhão de vezes mais, sem nunca fazê-lo da mesma forma duas vezes. Mesmo que se estivesse falando em padrões de construção de edifícios, também é verdade para padrões de construção de software orientado a objeto. Nossas soluções são expressas em termos de objetos e interfaces em vez de paredes e portas, mas no cerne dos dois tipos de padrões existe uma solução para um problema em um contexto.[GAM95]

Em geral, um padrão tem quatro elementos essenciais:[GAM95]

1. O **nome do padrão** é um identificador que podemos usar para descrever um problema de *design*, suas soluções, e suas consequências em uma ou duas palavras. A nomeação de um padrão imediatamente aumenta o nosso vocabulário de design. Ela nos permite projetar em um nível mais elevado de abstração. Ter um vocabulário de padrões permite-nos falar sobre eles com nossos colegas, em nossa documentação, e até mesmo para nós mesmos. Torna-se mais fácil pensar sobre os projetos e comunicar-se com as pessoas envolvidas. Encontrar bons nomes tem sido uma das partes mais difíceis de desenvolver nosso catálogo.
2. O **problema** descreve quando aplicar o padrão. Ele explica o problema e seu contexto. Poderia descrever os problemas específicos de design tais como como representar algoritmos como objetos. Poderia descrever classe ou estruturas de objeto que são sintomáticas de um projeto inflexível. Às vezes, a problema irá incluir uma lista de condições que devem ser cumpridas antes que faça sentido de aplicar o padrão.
3. A **solução** descreve os elementos que compõem o projeto, suas relações, responsabilidades e colaborações. A solução não descreve um determinado design concreto ou de execução, porque um padrão é como um modelo que pode ser aplicado em muitas situações diferentes. Em vez disso, o padrão fornece uma descrição abstrata de um problema de projeto e como um regime geral de elementos (classes e objetos no nosso caso) resolve ele.
4. As **consequências** são os resultados da aplicação do modelo. Quando descrevemos as decisões de design, eles são críticos para avaliar alternativas de projeto e para a compreensão dos custos e benefícios da aplicação do modelo. As consequências para o software são preocupação com freqüência e tempo de execução. Podem sinalizar linguagem e questões de implementação. Uma vez que a reutilização é frequentemente um fator de *design* orientado a objeto, as conseqüências de um padrão inclui o seu impacto sobre a flexibilidade de um sistema, extensibilidade ou portabilidade. Listando estas conseqüências explicitamente ajuda você a compreender e avaliá-los.

1.2. *Design Patterns* existentes

Os padrões hoje existentes estão divididos em três grupos:[GAM95]

- Padrões de criação (*Creational patterns*)

  - *Abstract factory*, Fornece uma interface para criar famílias de objetos relacionados ou dependentes sem especificar suas classes concretas.

  - *Builder*, Separa a construção de um objeto complexo da sua representação para que o mesmo processo de construção possa criar diferentes representações.

  - *Factory method*, Define uma interface para criar um objeto, mas deixa as subclasses decidirem qual classe instanciar. *Factory Method* permite subclasses façam a instanciação.

  - *Prototype*, Especifica o tipo de objectos a criar usando uma instância de protótipo, e cria novos objetos copiando este protótipo.

  - *Singleton*, Certifica-se de que uma classe tem apenas um exemplo, e fornece um ponto global de acesso à ele.

- Padrões estruturais (*Structural patterns*)

  - *Adapter*, Converte a interface de uma classe para outra interface que os clientes esperam. Este adaptador permite que classes trabalhem em conjunto, de outro modo não seria possivel por causa da incompatibilidade interfaces.

  - *Bridge*, Desacopla uma abstração de sua implementação para que as duas possam variar de forma independente.

  - *Composite*, Tem o objetivo de compor objetos em estruturas de árvore para representar hierarquias parte-todo. *Composite* permite que clientes tratem objetos individuais e composições de objetos uniformemente.

  - *Decorator*, Anexa responsabilidades adicionais a um objeto dinamicamente. *Decorators* fornecem uma alternativa flexível a subclasses para extensão da funcionalidade.

  - *Facade*, Fornece uma interface unificada para um conjunto de interfaces em um subsistema. *Facade* define uma interface de alto nível que torna o subsistema mais fácil de usar.

  - *Flyweight*, Usa compartilhamento para suportar um grande número de objetos de forma eficiente.

  - *Proxy*, Proporciona um espaço para outro objeto para controlar o acesso a ele.

- Padrões comportamentais (*Behavioral patterns*)

  - *Chain of Responsibility*, Evita o acoplamento do remetente de um pedido de seu receptor, dando a mais de um objeto a oportunidade para manipular a solicitação. Faz com que a cadeia os objetos e passe a receber pedido até que um objeto gere.

  - *Command*, Encapsula uma solicitação como um objeto, assim permitindo parametrizar clientes com diferentes pedidos, cria fila ou solicitações de log e suporte às operações reversível.

  - *Interpreter*, Dada uma linguagem, define uma representação para sua gramática juntamente com um intérprete que usa a representação para interpretar sentenças na linguagem.

  - *Iterator*, Fornece uma maneira de acessar os elementos de um objeto agregado seqüencialmente sem expor sua representação subjacente.

  - *Mediator*, Define um objeto que encapsula como um conjunto de objetos *interact.Mediator* promove acoplamento, mantendo os objetos de cada EPI referindo explicitamente outros, e ele permite que você varie sua interação de forma independente.

  - *Memento*, Sem violar o encapsulamento, captura e externaliza o estado de um *object sinternal* para que o objeto pode ser restaurado para este estado mais tarde.

– *Observer*, Define uma dependência um-para-muitos entre objetos de modo que quando um objeto muda de estado, todos seus dependentes sejam notificados e atualizados automaticamente.

– *State*, Permite que um objeto altere seu comportamento quando seu estado interno muda.

– *Strategy*, Define uma família de algoritmos, encapsula cada um, e torna-os intercambiáveis. Esta estratégia permite que o algoritmo pode variar independentemente do cliente que usá-lo.

– *Template Method*, Define o esqueleto de um algoritmo numa operação, adiando alguns passos para as subclasses. Permite que subclasses redefinam certos passos de um algoritmo sem alterar a estrutura do algoritmo.

– *Visitor*, Representa uma operação a ser realizada sobre os elementos de uma estrutura de objetos. Permite que você defina uma nova operação sem alterar as classes dos elementos em que opera.

Estes são os padrões mais comumente utilizados. Neste trabalho será detalhado o Padrão de criação *Factory Method*, também conhecido como *Virtual Constructor* (Construtor virtual).

2. *Factory Method*

O padrão *Factory Method* está relacionado a todos os outros padrões que conduzam algum tipo de construção encapsulada de objetos. Além de *factory* os padrões Iterator, Singleton , Builder, Prototype e Bridge- para citar alguns – estão relacionados ao padrão *Factory Method*.

Define uma interface para a criação de um objeto, mas deixa que as subclasses decidam qual a classe instanciar. Permite a uma classe delegar a instanciação para as subclasses.[GAM95]

2.1. Motivação

Frameworks usam classes abstratas para definir e manter relacionamentos entre objetos. Um *framework* é muitas vezes responsável por criar esses objetos também.

Considere-se um *framework* de aplicações que podem apresentar vários documentos ao usuário. Duas abstrações chave neste *framework* são as classes Application e Documento. Ambas as classes são abstratas, e os clientes têm usalas como subclasses para perceber suas implementações de aplicações específicas. Para criar um aplicativo de desenho, por exemplo, podemos definir as classes e *DrawingApplication* e *DrawingDocument*. A classe Application é responsável pela gestão de documentos e como criá-los assim que o usuário selecione Open ou Novo a partir de um menu, por exemplo.[GAM95]

Como subclasse do documento em particular para instanciar é específico do aplicativo, a classe Application não consegue prever que documento deve criar, a classe Application só sabe quando um novo documento deve ser criado, não o tipo do documento para criar. Isso cria um dilema: O *framework* deve instanciar classes, mas só sabe sobre classes abstratas, que não pode instanciar.[GAM95]

O padrão *Factory Method* oferece uma solução. Ele encapsula o conhecimento de que subclasse Documento deve criar e move este conhecimento para fora do *framework*.[GAM95]

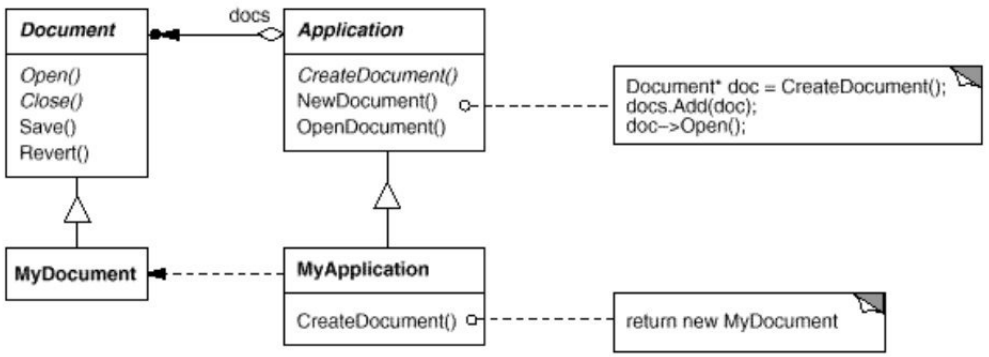


Figura 1: Representação da estrutura para criação de documentos

As subclasses da aplicação redefinem uma operação abstrata *CreateDocument* no aplicativo para retornar a subclasse apropriada documento. Uma vez que uma subclasse da aplicação é instanciada, ele pode então instanciar documentos sem saber sua classe. Chamamos *CreateDocument* um *factory method* porque é responsável por ”produzir”um objeto.[GAM95]

2.2. Aplicação

Se usa *Factory pattern* quando:[GAM95]

- uma classe (o criador) não pode antecipar a classe dos objetos que deve criar. .
- uma classe quer que suas subclasses possam especificar os objetos que criam.
- uma classe quiser localizar o conhecimento a respeito de subclasses às quais são delegadas responsabilidades específicas (em hierarquias de classes paralelas)

2.3. Problema

Uma classe precisa instanciar a derivação de uma outra, mas não sabe qual. O *Factory Method* permite que uma classe derivada tome essa decisão.

2.4. Solução

Uma classe derivada decide qual classe instanciar e o modo como instanciá-la. Algumas razões existem para que não queria utilizar new diretamente. A primeira, e mais óbvia, é não saber que classe de objeto instanciar. Isso é comum numa aplicação bem desenhada onde variáveis são estruturadas com base em interfaces. Assim, vários tipos de objetos diferentes podem ser associados a essa variável. Outra razão é a necessidade de inicializar o objeto instanciado antes de ser atribuído à variável. Importante notar que esta inicialização não depende do ambiente onde o objeto será utilizado, mas apenas da estrutura do próprio objeto.[WOR01] [GAM95]

2.5. Como utilizar

- É possível criar um objeto sem ter conhecimento algum de sua classe concreta?
- Esse conhecimento deve estar em alguma parte do sistema, mas não precisa estar no cliente
- *FactoryMethod* define uma interface comum para criar objetos
- O objeto específico é determinado nas diferentes implementações dessa interface
- O cliente do *FactoryMethod* precisa saber sobre implementações concretas do objeto criador do produto desejado

2.6. Diagrama de classes

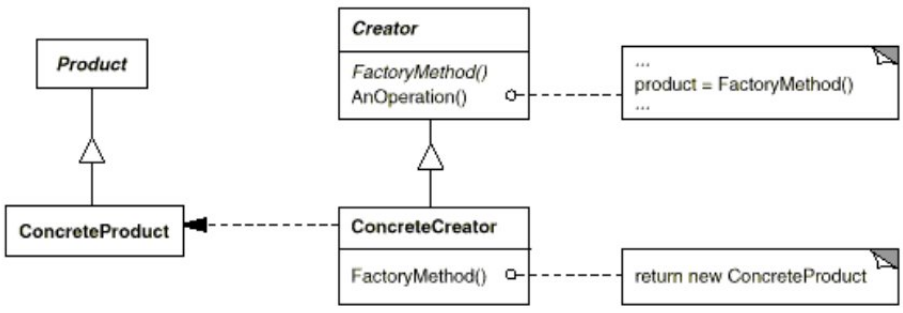


Figura 2: Representação da estrutura de classes utilizando *factory method*

- *Product*: define a interface dos objetos criados pelo *Factory Method*
- *ConcreteProduct*: implementa a interface definida em *Product*
- *Creator*: declara o *Factory Method* que retorna um objeto do tipo *Product*
  - Às vezes, o *Creator* não é apenas uma interface, mas pode envolver uma classe concreta que tenha uma implementação default para o *Factory Method* para retornar um objeto com algum tipo *ConcreteProduct* default
  - Pode chamar o *Factory Method* para criar um produto do tipo *Product*

- *ConcreteCreator*: faz override do *Factory Method* para retornar uma instância de *ConcreteProduct*

O *Creator* depende de suas subclasses para definir o *Factory Method* que retorne uma instância do *ConcreteProduct* apropriado.

Se o cliente tem que criar de qualquer maneira uma subclasse da classe *Creator*, usar subclasses pode ser uma boa solução, mas de outra maneira o cliente tem que lidar com outro ponto da evolução.

Uma desvantagem potencial do *factory method* é que os clientes podem precisar ter uma subclasse *Creator* apenas para criar um objeto *ConcreteProduct* específico.

2.7. Benefícios e consequencias

Factory Method elimina a necessidade de vincular as classes específicas do aplicativo em seu código. O código lida apenas com a interface *Product*, pois pode trabalhar com qualquer classe *ConcreteProduct* definida pelo usuário. Isso diminui o acoplamento, torna as classes mais flexíveis e amarra apenas a estrutura, transferindo a responsabilidade para subclasses.[GAM95]

Factory Methods eliminam a necessidade de colocar classes específicas da aplicação no código.[SAU01]

- O código só lida com a interface *Product*
- O código pode, portanto funcionar com qualquer classe *ConcreteProduct* Provê ganchos (*Hook Methods*) para subclasses.
- Criar objetos dentro de uma classe com um *Factory Method* é sempre mais flexível do que criar objetos diretamente.
- O *Factory Method* provê um gancho para que subclasses forneçam uma versão estendida de um objeto
- Criação de objetos é desacoplada do conhecimento do tipo concreto do objeto
- Conecta hierarquias de classe paralelas
- Facilita a extensibilidade

Aqui estão duas consequências adicionais do padrão *Factory Method*:[GAM95]

1. Fornecer ganchos para subclasses. Criação de objetos dentro de uma classe com *factory methods* é sempre mais flexível do que criar um objeto diretamente. *Factory Method* da para subclasses um gancho para fornecer uma versão estendida de um objeto.

   No exemplo do documento, a classe *Document* poderia definir um *factory method* chamado *CreateFileDialog* que cria uma caixa de dialogo padrão na abertura de documentos existentes.

   Uma subclasse *Document* pode definir uma caixa de dialogo específico do aplicativo, substituindo este *factory method*. Neste caso, o *factory method* não é abstrato, mas fornece implementação default.
2. Conecta hierarquia de classes paralelas, nos exemplos que temos considerado até agora, o *factory method* é chamado apenas por *Creators*. Mas isso não tem que ser o caso, os clientes podem encontrar *factory methods* úteis, especialmente no caso de hierarquias de classes paralelas.

   Hierarquias de classes paralelas resulta de quando uma classe delega algumas de suas responsabilidades para uma classe separada. Considere figuras que podem ser manipuladas de forma interativa, ou seja, elas podem ser esticadas, movidas, ou giradas usando o mouse. A execução de tais interações nem sempre é fácil. Muitas vezes isso exige armazenamento e atualização das informações que registra o estado da manipulação em um determinado tempo.

   Este estado é necessária apenas durante manipulação e, portanto, não precisa ser mantido no objeto. Além disso, diferentes objetos se comportam de maneira diferente quando o usuário as manipula. Para exemplo, esticar uma figura da linha poderia ter o efeito de mover um ponto final, considerando um valor de alongamento o texto pode mudar o seu espaçamento entre as linhas.

   Com estas restrições, é melhor usar um objeto Manipulator separado que implementa a interação e mantém registro de qualquer Estado de manipulação específica necessário. Diferentes figuras usarão diferentes manipuladores para lidar com interações específicas. O hierarquia de classes manipulatoras resultante (pelo menos parcialmente) é ilustrada na figura 3:

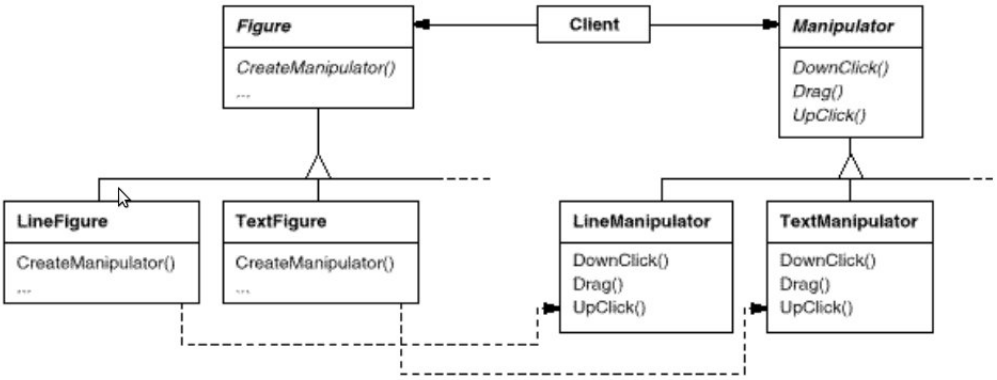


Figura 3: Hierarquia de classes paralelas

A classe *Figure* proporciona um *factory method CreateManipulator* que permite os clientes a criar um *Manipulator* correspondente ao da figura. As subclasses *Figure* sobreescrevem esse método para retornar uma instância da subclasse *Manipulator* correto para eles. Alternativamente, a classe *Figure* pode implementar *CreateManipulator* para retornar uma instancia de um *Manipulator* padrão, e a classe *Figure* pode simplesmente herdar esse padrão.

2.8. Exemplos de uso

Nesta seção será ilustrada a utilização de *Factory Method* em uma aplicação de criação de planta baixa de apartamentos. As classes não são implementadas em detalhes, o objetivo é simplesmente estruturar o código segundo o padrão desenvolvido até o momento.

Classe Produto

```
1  package br.com.uniritter.methodFactory;
2
3  import java.util.HashMap;
5
6  //PRODUCT
7  public abstract class ElementOfApart {
8
9      public static final int NORTE = 1;
10     public static final int LESTE = 2;
11     public static final int OESTE = 3;
12     public static final int SUL = 4;
13
14     Map<Integer, ElementOfApart> neighbors = new HashMap<Integer, ElementOfApart>();
15
16     public void setNeighbor(Integer posicao, ElementOfApart element){
17         this.neighbors.put(posicao, element);
18     }
19
20     public Map<Integer, ElementOfApart> getVisinhos() {
21         return neighbors;
22     }
23 }
```

Classes de produto concretas

```java
package br.com.uniritter.methodFactory;

import java.util.ArrayList;

public class FloorPlan extends ElementOfApart {

    List<ElementOfApart> elements = new ArrayList<ElementOfApart>();

    public void addElement(ElementOfApart element){
        this.elements.add(element);
    }
}
```

```java
package br.com.uniritter.methodFactory;

public class Room extends ElementOfApart{

}
```

```java
package br.com.uniritter.methodFactory;

public class Bedroom extends ElementOfApart {

    private boolean suit;

    private boolean closet;

    public Bedroom(boolean suit, boolean closet){
        this.suit = suit;
        this.closet = closet;
    }

    public boolean isSuit() {
        return suit;
    }

    public void setSuit(boolean suit) {
        this.suit = suit;
    }

    public boolean isCloset() {
        return closet;
    }

    public void setCloset(boolean closet) {
        this.closet = closet;
    }
}
```

```java
package br.com.uniritter.methodFactory;

public class Kitchen extends ElementOfApart {

}
```

```java
package br.com.uniritter.methodFactory;

public class AmericanKitchen extends Kitchen {

}
```

Classe criadora

```java
package br.com.uniritter.methodFactory;

//CREATOR
public abstract class Apartament {

    public FloorPlan createFloorPlan(){
        return new FloorPlan();
    }

    public Room createRoom(){
        return new Room();
    }

    public Kitchen createKitchen(){
        return new Kitchen();
    }

    public Bedroom createBedroom(boolean suite, boolean closet){
        return new Bedroom(suite, closet);
    }

    public FloorPlan createApartament(){
        FloorPlan floorPlan = createFloorPlan();

        Room room1 = createRoom();
        floorPlan.addElement(room1);

        Room room2 = createRoom();
        floorPlan.addElement(room2);

        Kitchen kitchen = createKitchen();
        floorPlan.addElement(kitchen);

        Bedroom bedroom1 = createBedroom(true, true);
        floorPlan.addElement(bedroom1);

        Bedroom bedroom2 = createBedroom(true, false);
        floorPlan.addElement(bedroom2);

        room1.setNeighbor(ElementOfApart.LESTE, bedroom2);
        room1.setNeighbor(ElementOfApart.SUL, room2);
        room2.setNeighbor(ElementOfApart.SUL, kitchen);
        bedroom2.setNeighbor(ElementOfApart.LESTE, bedroom1);
        return floorPlan;
    }
}
```

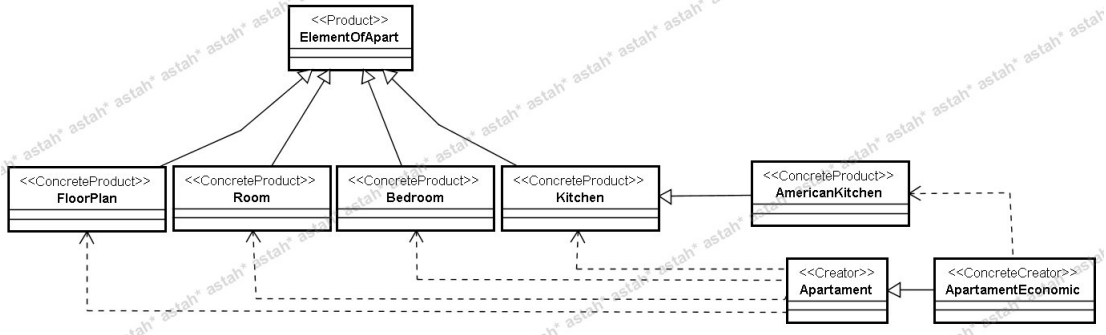
Classe criadora concreta

```java
package br.com.uniritter.methodFactory;

public class ApartamentEconomic extends Apartament {

    public Kitchen createKitchen(){
        return new AmericanKitchen();
    }
}
```

Esse é o diagrama que corresponde aos códigos acima



Referências

Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John M. **Design patterns: elements of reusable object-oriented software**. Addisonwesley Reading, MA, 1995.

Sauve, Jacques Philippe. **Factory method**. 2001. (http://www.dsc.ufcg.edu.br/ jacques/cursos/map/html/-pat/factory.htm (Ultimo acesso Agosto de 2010)).

World, Java. **Factory methods**. 2001. (http://www.javaworld.com/javaworld/javaqa/2001-05/02-qa-0511-factory.html (Ultimo acesso Agosto de 2010)).