refactoring.guru

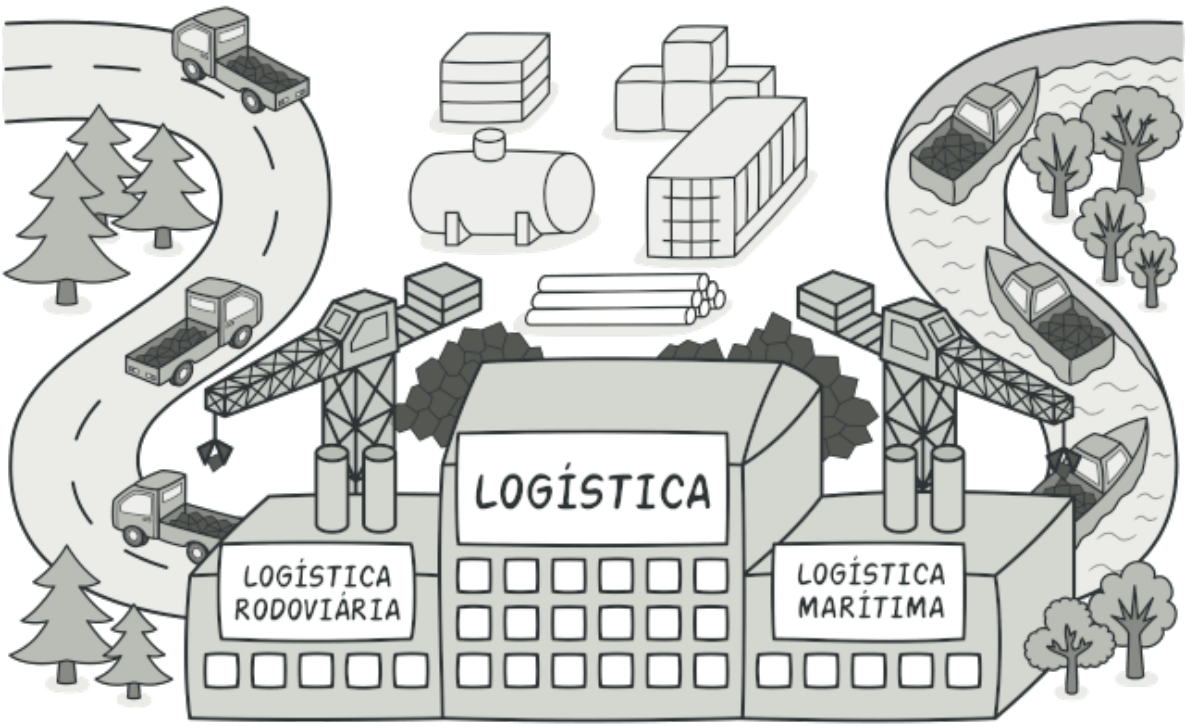
Factory Method

15–21 minutes

Também conhecido como: Método fábrica, Construtor virtual

Propósito

O **Factory Method** é um padrão criacional de projeto que fornece uma interface para criar objetos em uma superclasse, mas permite que as subclasses alterem o tipo de objetos que serão criados.



Problema

Imagine que você está criando uma aplicação de gerenciamento de logística. A primeira versão da sua aplicação pode lidar apenas com o transporte de caminhões, portanto a maior parte do seu código fica dentro da classe `Caminhão`.

Depois de um tempo, sua aplicação se torna bastante popular. Todos os dias você recebe dezenas de solicitações de empresas de transporte marítimo para incorporar a logística marítima na aplicação.

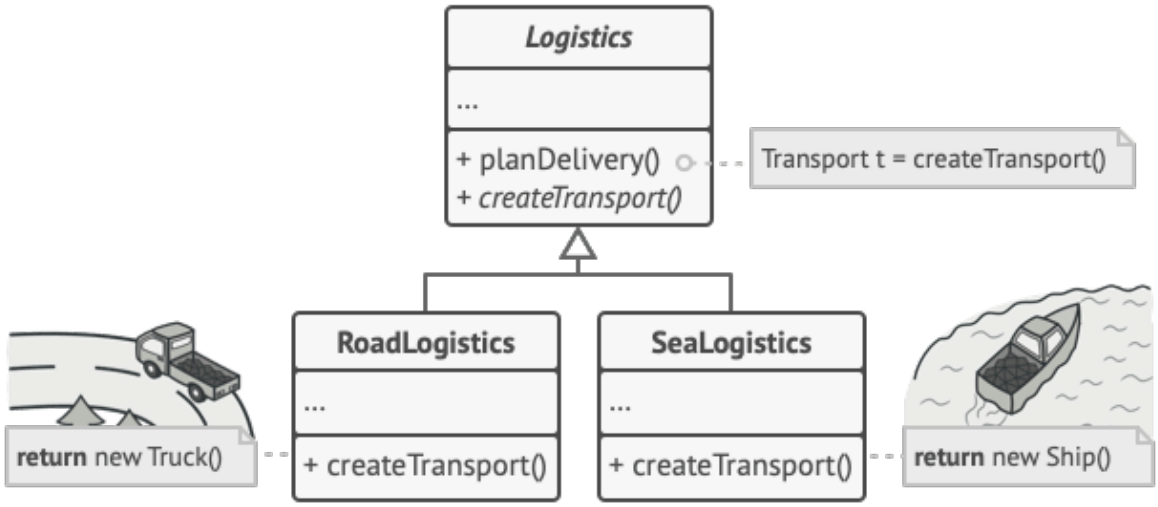
Adicionar uma nova classe ao programa não é tão simples se o restante do código já estiver acoplado às classes existentes.

Boa notícia, certo? Mas e o código? Atualmente, a maior parte do seu código é acoplada à classe `Caminhão`. Adicionar `Navio` à aplicação exigiria alterações em toda a base de código. Além disso, se mais tarde você decidir adicionar outro tipo de transporte à aplicação, provavelmente precisará fazer todas essas alterações novamente.

Como resultado, você terá um código bastante sujo, repleto de condicionais que alteram o comportamento da aplicação, dependendo da classe de objetos de transporte.

Solução

O padrão Factory Method sugere que você substitua chamadas diretas de construção de objetos (usando o operador `new`) por chamadas para um método *fábrica* especial. Não se preocupe: os objetos ainda são criados através do operador `new`, mas esse está sendo chamado de dentro do método fábrica. Objetos retornados por um método fábrica geralmente são chamados de *produtos*.

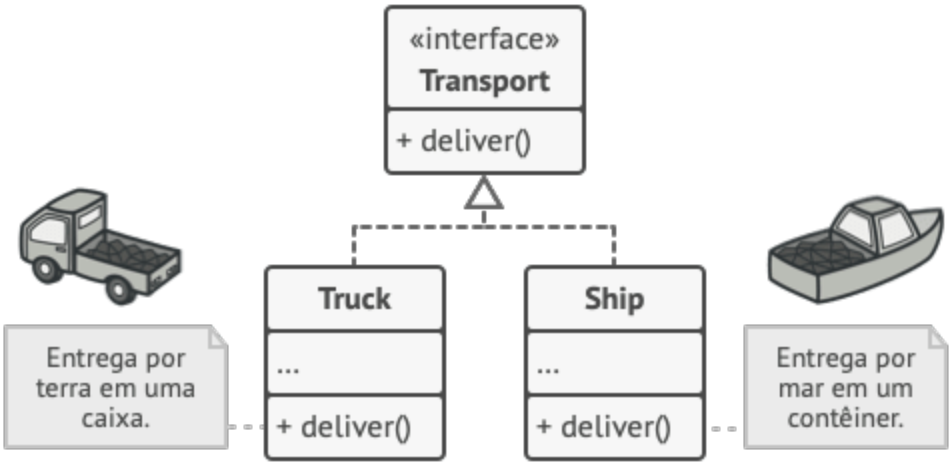


As subclasses podem alterar a classe de objetos retornados pelo método fábrica.

À primeira vista, essa mudança pode parecer sem sentido: apenas mudamos a chamada do construtor de uma parte do programa para outra. No entanto, considere o seguinte: agora você pode sobrescrever o método fábrica em uma subclasse e alterar a classe de produtos que estão sendo criados pelo método.

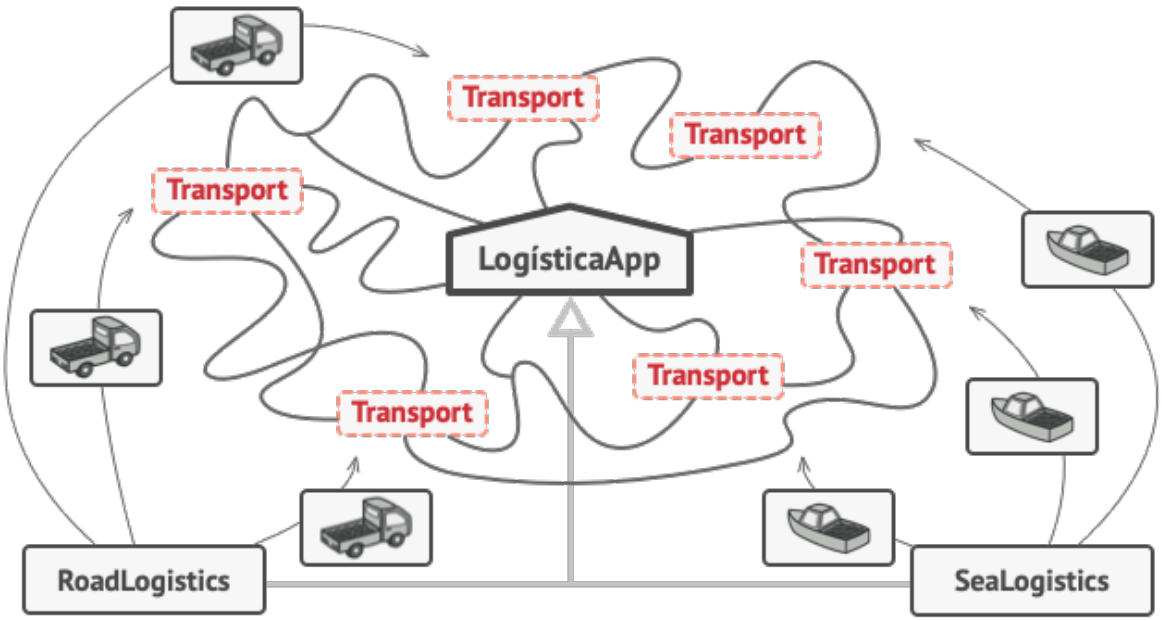
Porém, há uma pequena limitação: as subclasses só podem retornar tipos diferentes de produtos se esses produtos tiverem uma classe ou interface base em comum. Além disso, o método fábrica na classe base deve ter seu tipo de retorno declarado

como essa interface.



Todos os produtos devem seguir a mesma interface.

Por exemplo, ambas as classes `Caminhão` e `Navio` devem implementar a interface `Transporte`, que declara um método chamado `entregar`. Cada classe implementa esse método de maneira diferente: caminhões entregam carga por terra, navios entregam carga por mar. O método fábrica na classe `LogísticaViária` retorna objetos de caminhão, enquanto o método fábrica na classe `LogísticaMarítima` retorna navios.



Desde que todas as classes de produtos implementem uma

interface comum, você pode passar seus objetos para o código cliente sem quebrá-lo.

O código que usa o método fábrica (geralmente chamado de código *cliente*) não vê diferença entre os produtos reais retornados por várias subclasses. O cliente trata todos os produtos como um `Transporte` abstrato. O cliente sabe que todos os objetos de transporte devem ter o método `entregar`, mas como exatamente ele funciona não é importante para o cliente.

Estrutura

1. O **Produto** declara a interface, que é comum a todos os objetos que podem ser produzidos pelo criador e suas subclasses.
2. **Produtos Concretos** são implementações diferentes da interface do produto.
3. A classe **Criador** declara o método fábrica que retorna novos objetos produto. É importante que o tipo de retorno desse método corresponda à interface do produto.

   Você pode declarar o método fábrica como abstrato para forçar todas as subclasses a implementar suas próprias versões do método. Como alternativa, o método fábrica base pode retornar algum tipo de produto padrão.

   Observe que, apesar do nome, a criação de produtos **não** é a principal responsabilidade do criador. Normalmente, a classe criadora já possui alguma lógica de negócio relacionada aos produtos. O método fábrica ajuda a dissociar essa lógica das classes concretas de produtos. Aqui está uma analogia: uma grande empresa de desenvolvimento de software pode ter um

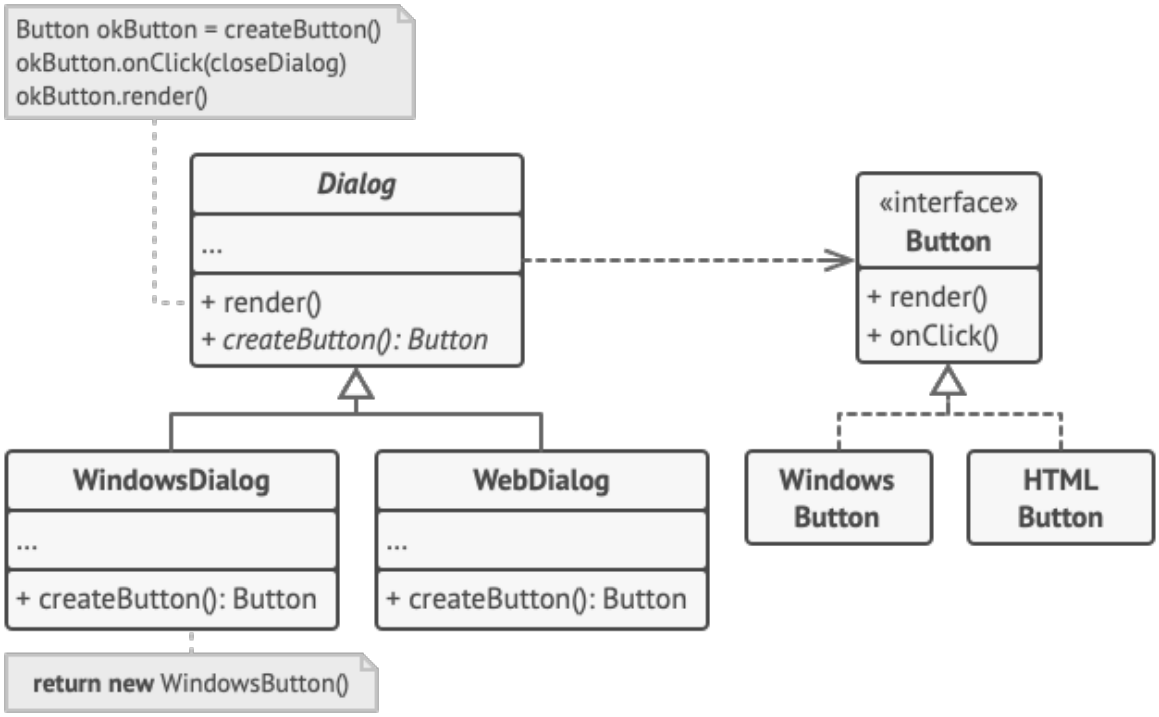
departamento de treinamento para programadores. No entanto, a principal função da empresa como um todo ainda é escrever código, não produzir programadores.

4. **Criadores Concretos** sobrescrevem o método fábrica base para retornar um tipo diferente de produto.

   Observe que o método fábrica não precisa **criar** novas instâncias o tempo todo. Ele também pode retornar objetos existentes de um cache, um conjunto de objetos, ou outra fonte.

Pseudocódigo

Este exemplo ilustra como o **Factory Method** pode ser usado para criar elementos de interface do usuário multiplataforma sem acoplar o código do cliente às classes de UI concretas.



Exemplo de diálogo de plataforma cruzada.

A classe base diálogo usa diferentes elementos da UI do usuário para renderizar sua janela. Em diferentes sistemas operacionais,

esses elementos podem parecer um pouco diferentes, mas ainda devem se comportar de forma consistente. Um botão no Windows ainda é um botão no Linux.

Quando o método fábrica entra em ação, você não precisa reescrever a lógica da caixa de diálogo para cada sistema operacional. Se declararmos um método fábrica que produz botões dentro da classe base da caixa de diálogo, mais tarde podemos criar uma subclasse de caixa de diálogo que retorna botões no estilo Windows do método fábrica. A subclasse herda a maior parte do código da caixa de diálogo da classe base, mas, graças ao método fábrica, pode renderizar botões estilo Windows na tela.

Para que esse padrão funcione, a classe base da caixa de diálogo deve funcionar com botões abstratos: uma classe base ou uma interface que todos os botões concretos seguem. Dessa forma, o código da caixa de diálogo permanece funcional, independentemente do tipo de botão com o qual ela trabalha.

Obviamente, você também pode aplicar essa abordagem a outros elementos da UI. No entanto, com cada novo método fábrica adicionado à caixa de diálogo, você se aproxima do padrão Abstract Factory. Não se preocupe, falaremos sobre esse padrão mais tarde.

```
// A classe criadora declara o método fábrica que deve retornar
// um objeto de uma classe produto. As subclasses da criadora
// geralmente fornecem a implementação desse método.
class Dialog is
    // A criadora também pode fornecer alguma implementação
    // padrão do Factory Method.
```

```
    abstract method createButton():Button

    // Observe que, apesar do seu nome, a principal
    // responsabilidade da criadora não é criar produtos. Ela
    // geralmente contém alguma lógica de negócio central que
    // depende dos objetos produto retornados pelo método
    // fábrica. As subclasses pode mudar indiretamente essa
    // lógica de negócio ao sobrescreverem o método fábrica e
    // retornarem um tipo diferente de produto dele.
    method render() is
        // Chame o método fábrica para criar um objeto produto.
        Button okButton = createButton()
        // Agora use o produto.
        okButton.onClick(closeDialog)
        okButton.render()


// Criadores concretos sobrescrevem o método fábrica para mudar
// o tipo de produto resultante.
class WindowsDialog extends Dialog is
    method createButton():Button is
        return new WindowsButton()

class WebDialog extends Dialog is
    method createButton():Button is
        return new HTMLButton()


// A interface do produto declara as operações que todos os
// produtos concretos devem implementar.
```

```
interface Button is
    method render()
    method onClick(f)

// Produtos concretos fornecem várias implementações da
// interface do produto.
class WindowsButton implements Button is
    method render(a, b) is
        // Renderiza um botão no estilo Windows.
    method onClick(f) is
        // Vincula um evento de clique do SO nativo.

class HTMLButton implements Button is
    method render(a, b) is
        // Retorna uma representação HTML de um botão.
    method onClick(f) is
        // Vincula um evento de clique no navegador web.



class Application is
    field dialog: Dialog

    // A aplicação seleciona um tipo de criador dependendo da
    // configuração atual ou definições de ambiente.
    method initialize() is
        config = readApplicationConfigFile()

        if (config.OS == "Windows") then
            dialog = new WindowsDialog()
        else if (config.OS == "Web") then
```

```
            dialog = new WebDialog()
        else
            throw new Exception("Error! Unknown operating system.")

    // O código cliente trabalha com uma instância de um criador
    // concreto, ainda que com sua interface base. Desde que o
    // cliente continue trabalhando com a criadora através da
    // interface base, você pode passar qualquer subclasse da
    // criadora.
    method main() is
        this.initialize()
        dialog.render()
```

Aplicabilidade

Use o Factory Method quando não souber de antemão os tipos e dependências exatas dos objetos com os quais seu código deve funcionar.

O Factory Method separa o código de construção do produto do código que realmente usa o produto. Portanto, é mais fácil estender o código de construção do produto independentemente do restante do código.

Por exemplo, para adicionar um novo tipo de produto à aplicação, só será necessário criar uma nova subclasse criadora e substituir o método fábrica nela.

Use o Factory Method quando desejar fornecer aos usuários da sua biblioteca ou framework uma maneira de estender seus componentes internos.

Herança é provavelmente a maneira mais fácil de estender o

comportamento padrão de uma biblioteca ou framework. Mas como o framework reconheceria que sua subclasse deve ser usada em vez de um componente padrão?

A solução é reduzir o código que constrói componentes no framework em um único método fábrica e permitir que qualquer pessoa sobrescreva esse método, além de estender o próprio componente.

Vamos ver como isso funcionaria. Imagine que você escreva uma aplicação usando um framework de UI de código aberto. Sua aplicação deve ter botões redondos, mas o framework fornece apenas botões quadrados. Você estende a classe padrão `Botão` com uma gloriosa subclasse `BotãoRedondo`. Mas agora você precisa informar à classe principal `UIFramework` para usar a nova subclasse no lugar do botão padrão.

Para conseguir isso, você cria uma subclasse `UIComBotõesRedondos` a partir de uma classe base do framework e sobrescreve seu método `criarBotão`. Enquanto este método retorna objetos `Botão` na classe base, você faz sua subclasse retornar objetos `BotãoRedondo`. Agora use a classe `UIComBotõesRedondos` no lugar de `UIFramework`. E é isso!

Use o Factory Method quando deseja economizar recursos do sistema reutilizando objetos existentes em vez de recriá-los sempre.

Você irá enfrentar essa necessidade ao lidar com objetos grandes e pesados, como conexões com bancos de dados, sistemas de arquivos e recursos de rede.

Vamos pensar no que deve ser feito para reutilizar um objeto existente:

1. Primeiro, você precisa criar algum armazenamento para manter o controle de todos os objetos criados.
2. Quando alguém solicita um objeto, o programa deve procurar um objeto livre dentro desse conjunto.
3. ...e retorná-lo ao código cliente.
4. Se não houver objetos livres, o programa deve criar um novo (e adicioná-lo ao conjunto de objetos).

   Isso é muito código! E tudo deve ser colocado em um único local para que você não polua o programa com código duplicado.

   Provavelmente, o lugar mais óbvio e conveniente onde esse código deve ficar é no construtor da classe cujos objetos estamos tentando reutilizar. No entanto, um construtor deve sempre retornar **novos objetos** por definição. Não pode retornar instâncias existentes.

   Portanto, você precisa ter um método regular capaz de criar novos objetos e reutilizar os existentes. Isso parece muito com um método fábrica.

Como implementar

1. Faça todos os produtos implementarem a mesma interface. Essa interface deve declarar métodos que fazem sentido em todos os produtos.
2. Adicione um método fábrica vazio dentro da classe criadora. O tipo de retorno do método deve corresponder à interface comum do produto.
3. No código da classe criadora, encontre todas as referências aos construtores de produtos. Um por um, substitua-os por chamadas

ao método fábrica, enquanto extrai o código de criação do produto para o método fábrica.

Pode ser necessário adicionar um parâmetro temporário ao método fábrica para controlar o tipo de produto retornado.

Neste ponto, o código do método fábrica pode parecer bastante feio. Pode ter um grande operador `switch` que escolhe qual classe de produto instanciar. Mas não se preocupe, resolveremos isso em breve.

4. Agora, crie um conjunto de subclasses criadoras para cada tipo de produto listado no método fábrica. Sobrescreva o método fábrica nas subclasses e extraia os pedaços apropriados do código de construção do método base.
5. Se houver muitos tipos de produtos e não fizer sentido criar subclasses para todos eles, você poderá reutilizar o parâmetro de controle da classe base nas subclasses.

   Por exemplo, imagine que você tenha a seguinte hierarquia de classes: a classe base `Correio` com algumas subclasses: `CorreioAéreo` e `CorreioTerrestre`; as classes `Transporte` são `Avião`, `Caminhão` e `Trem`. Enquanto a classe `CorreioAéreo` usa apenas objetos `Avião`, o `CorreioTerrestre` pode funcionar com os objetos `Caminhão` e `Trem`. Você pode criar uma nova subclasse (por exemplo, `CorreioFerroviário`) para lidar com os dois casos, mas há outra opção. O código do cliente pode passar um argumento para o método fábrica da classe `CorreioTerrestre` para controlar qual produto ele deseja receber.
6. Se, após todas as extrações, o método fábrica base ficar vazio, você poderá torná-lo abstrato. Se sobrar algo, você pode tornar

isso em um comportamento padrão do método.

Prós e contras

- Você evita acoplamentos firmes entre o criador e os produtos concretos.
- *Princípio de responsabilidade única*. Você pode mover o código de criação do produto para um único local do programa, facilitando a manutenção do código.
- *Princípio aberto/fechado*. Você pode introduzir novos tipos de produtos no programa sem quebrar o código cliente existente.
- O código pode se tornar mais complicado, pois você precisa introduzir muitas subclasses novas para implementar o padrão. O melhor cenário é quando você está introduzindo o padrão em uma hierarquia existente de classes criadoras.

Relações com outros padrões

- Muitos projetos começam usando o Factory Method (menos complicado e mais customizável através de subclasses) e evoluem para o Abstract Factory, Prototype, ou Builder (mais flexíveis, mas mais complicados).
- Classes Abstract Factory são quase sempre baseadas em um conjunto de métodos fábrica, mas você também pode usar o Prototype para compor métodos dessas classes.
- Você pode usar o Factory Method junto com o Iterator para permitir que uma coleção de subclasses retornem diferentes tipos de iteradores que são compatíveis com as coleções.
- O Prototype não é baseado em heranças, então ele não tem os

inconvenientes dela. Por outro lado, o *Prototype* precisa de uma inicialização complicada do objeto clonado. O Factory Method é baseado em herança mas não precisa de uma etapa de inicialização.

- O Factory Method é uma especialização do Template Method. Ao mesmo tempo, o *Factory Method* pode servir como uma etapa em um *Template Method* grande.