DESCRIÇÃO

Padrões GoF de Projeto do grupo Criação: Template Method, Abstract Factory, Builder, Prototype e Singleton.

PROPÓSITO

Compreender os padrões de projeto ligados à criação de objetos em projetos orientados a objetos e identificar oportunidades para a sua aplicação são habilidades importantes para um projetista de software, pois, sem elas, as soluções geradas podem ser inflexíveis e dificultar a evolução de sistemas de software em prazo e custo aceitáveis.

PREPARAÇÃO

Antes de iniciar o conteúdo, é recomendado instalar em seu computador um programa que lhe permita elaborar modelos sob a forma de diagramas da UML (Linguagem Unificada de Modelagem). Nossa sugestão inicial é o **Free Student License for Astah UML**, usado nos exemplos deste estudo. Para isso, será necessário usar seu e-mail institucional para ativar a licença.

Preencha os dados do formulário no site do software, envie e aguarde a liberação de sua licença em seu e-mail institucional. Ao receber a licença, siga as instruções do e-mail e instale o produto em seu computador. Os **arquivos Astah** com diagramas UML utilizados nesse conteúdo estão disponíveis para download.

Sugestões de links adicionais de ferramentas livres para modelagem de sistemas em UML (UML Tools) podem ser encontradas em buscas na Internet.

Além disso, recomendamos a instalação de um ambiente de programação em Java. O ambiente recomendado para iniciantes em Java é o **Apache NetBeans**, cujo instalador pode ser encontrado no site do ambiente, acessando o menu Download. Porém, antes de instalar o

NetBeans, é necessário ter instalado o **JDK** (Java Development Kit) referente à edição Java SE (Standard Edition), que pode ser encontrado no site da Oracle Technology Network: **Java SE - Downloads | Oracle Technology Network | Oracle**.

Os códigos com exemplos de aplicação dos padrões estão disponíveis ao longo do conteúdo em formato texto, bastando copiar para inserir no ambiente de programação.

OBJETIVOS

MÓDULO 1

Reconhecer o propósito, a estrutura e as situações de aplicação do padrão de projeto Factory Method

MÓDULO 2

Reconhecer o propósito, a estrutura e as situações de aplicação do padrão de projeto Abstract Factory

MÓDULO 3

Reconhecer o propósito, a estrutura e as situações de aplicação do padrão de projeto Builder

MÓDULO 4

Reconhecer o propósito, a estrutura e as situações de aplicação dos padrões de projeto Prototype e Singleton

INTRODUÇÃO

Os padrões GoF, do inglês “Gang of Four”, são padrões de projeto orientado a objetos divididos em três categorias: de criação, estruturais e comportamentais. São assim denominados por terem sido introduzidos pelos quatro autores do livro *Design Patterns: Elements of Reusable Object-Oriented Software* (GAMMA *et al*., 1994).

Os padrões de projeto GoF de criação nos ajudam a construir sistemas independentes da forma com que os objetos são criados e representados. Quando criamos um objeto da classe B em um método da classe A por meio de um simples comando “new B()”, estabelecemos uma relação de dependência entre duas implementações, visto que, em Java, uma classe é uma implementação concreta de um conjunto de operações. Neste exemplo, portanto, a classe A é dependente da classe B.

Em várias situações, entretanto, a criação de uma relação de dependência entre duas ou mais implementações torna o sistema inflexível, dificultando a sua evolução. Os **princípios SOLID** de Inversão de Dependências e Open Closed nos orientam a fazer as implementações dependerem de abstrações, especialmente em casos nos quais um módulo depende de um serviço que possa ter diferentes implementações.

Imagine um módulo que utilize um intermediador de pagamentos fornecido por uma empresa externa. Existe a chance de mudarmos a empresa fornecedora ou de termos de trabalhar com mais de uma empresa? Claro que sim!

Nesse caso, não é uma boa ideia fazer os módulos do nosso sistema dependerem da implementação de uma empresa específica.

Os módulos devem, portanto, depender de abstrações, de forma que seja possível trabalhar com diferentes intermediadores de pagamentos, sem haver necessidade de alterar os módulos

clientes desse serviço.

Entretanto, para utilizarmos as implementações específicas, precisamos instanciar objetos referentes a essas implementações. Como podemos instanciar esses objetos sem estabelecermos dependências indesejáveis? Esse é um dos problemas fundamentais tratados pelos padrões de projeto GoF do grupo Criação.

Neste conteúdo, você aprenderá os cinco padrões desse grupo: Factory Method, Abstract Factory, Builder, Prototype e Singleton. Esses padrões encapsulam o conhecimento sobre as classes concretas que o sistema utiliza e sobre como as instâncias dessas classes são criadas, permitindo que os demais módulos do sistema trabalhem com interfaces abstratas em lugar de implementações específicas.

Projetos construídos com a aplicação correta desses padrões possuem maior flexibilidade em relação aos objetos criados, aos criadores desses objetos e a como e quando esses objetos são criados, facilitando a implementação de variações e evoluções do sistema.

PRINCÍPIOS SOLID

O acrônimo SOLID se refere, na programação orientada a objetos, a cinco princípios ou postulados de design, destinados a facilitar a compreensão, o desenvolvimento e a manutenção de software. São eles:

Single-responsibility (responsabilidade única);

Open-closed (aberto-fechado);

Liskov substitution (substituição de Liskov);

Interface segregation (segregação de interface);

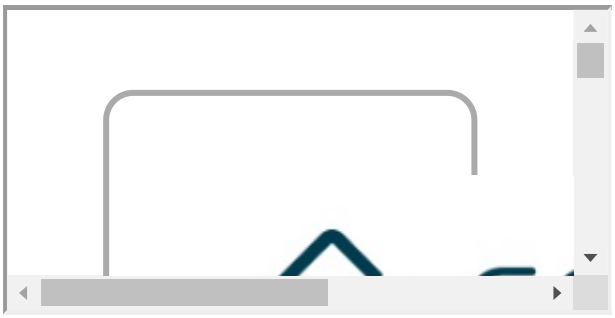
Dependency inversion (inversão de dependência).

MÓDULO 1

⦿ **Reconhecer o propósito, a estrutura e as situações de aplicação do padrão de projeto Factory Method**

PADRÃO DE PROJETO FACTORY METHOD

Apresentamos, no vídeo a seguir, um exemplo de aplicação do padrão Factory Method no desenvolvimento de frameworks.



INTENÇÃO DO PADRÃO FACTORY METHOD

Factory Method é um padrão frequentemente utilizado na implementação de frameworks.

Ele define uma interface para a criação de objetos, deixando para as subclasses a decisão sobre a classe específica a ser instanciada.

PROBLEMA DO PADRÃO FACTORY METHOD

Você conhece o problema que o padrão Factory Method busca resolver?

Suponha que a sua tarefa seja implementar um método que remova os itens inválidos de uma coleção de itens de um pedido. Um item de pedido possui os seguintes atributos:

A quantidade

O preço unitário

O produto solicitado

Imagine que um item válido é aquele que tenha uma quantidade de 1 a 100.

Veja a implementação da classe **ItemPedido** a seguir.

Além dos atributos, das operações de acesso e do construtor, essa classe define a operação valor, que retorna o valor do item resultante da multiplicação da quantidade pelo preço unitário.

```
public class ItemPedido {
private Produto produto;
private int quantidade;
private int precoUnitarioEmCentavos;

public ItemPedido(Produto produto, int quantidade, int valorEmCentavos) {
this.produto = produto;
this.quantidade = quantidade;
this.precoUnitarioEmCentavos = valorEmCentavos;
}
public int valor() {
return quantidade * precoUnitarioEmCentavos;
}
public Produto getProduto() {
return produto;
}
public void setProduto(Produto produto) {
this.produto = produto;
}
public int getQuantidade() {
return quantidade;
}
```

```
public void setQuantidade(int quantidade) {
this.quantidade = quantidade;
}
public int getPrecoUnitarioEmCentavos() {
return precoUnitarioEmCentavos;
}
public void setPrecoUnitarioEmCentavos(int precoUnitarioEmCentavos) {
this.precoUnitarioEmCentavos = precoUnitarioEmCentavos;
}
}
```

Uma primeira implementação da operação de remoção dos itens inválidos está listada a seguir:

```
public void removerItensInvalidos(ArrayList‹ItemPedido› itens) {
ArrayIterator‹ItemPedido› cursor = new ArrayIterator(itens);
while (cursor.hasNext()) {
ItemPedido item = cursor.next();
if (! isValido(item)) {
cursor.remove(item);
}
}
}

public boolean isValido(ItemPedido item) {
return (item.getQuantidade() › 0 && item.getQuantidade() ‹ 100);
}
```

Nessa implementação, imagine que definimos uma classe **ArrayIterator** que implementa um cursor sobre os itens de pedido recebidos como parâmetro com as seguintes operações:

HASNEXT

Que verifica se existe um próximo elemento no ArrayList ou se o cursor já está posicionado no último elemento.

NEXT

Que retorna o próximo elemento do ArrayList. Na primeira chamada, ele retorna o primeiro elemento da coleção.

REMOVE

Que remove um elemento da coleção.

Você consegue identificar o principal problema dessa solução?

Embora funcione, essa solução utiliza a classe **ArrayList**, criando um acoplamento da implementação com uma forma específica de organização dos itens.

Suponha que os pedidos passem a ser organizados em um **HashSet**, por exemplo. O efeito negativo desse acoplamento fica evidente, pois teremos de modificar a implementação, uma vez que a forma de percurso em um HashSet é diferente daquela utilizada em um ArrayList.

Poderíamos desenvolver outra versão específica para um HashSet, definindo uma classe HashSetIterator e implementando as mesmas operações da classe ArrayListIterator, mas com um algoritmo específico para o percurso e a manipulação dos elementos.

O código a seguir apresenta a versão da operação removerItensInvalidos implementada a partir de um HashSet:

```
public void removerItensInvalidos(HashSet‹ ItemPedido› itens) {
HashSetIterator‹ItemPedido› cursor = new HashSetIterator(itens);
while (cursor.hasNext()) {
ItemPedido item = cursor.next();
if (! isValido(item)) {
cursor.remove(item);
}
}
}
```

Imagine, agora, que houvesse diversos outros tipos de coleção. Você faria uma nova implementação para cada tipo específico de coleção?

Indo além, imagine que esse problema que você enfrentou em uma operação específica do sistema (isto é, remover itens de pedido inválidos) ocorra em dezenas de outras situações do mesmo sistema. O resultado será uma enorme replicação de código, que é um dos principais inimigos da evolução sustentável de um sistema.

Note que as duas implementações apresentadas são muito parecidas, diferindo apenas pelo tipo de coleção e do cursor criado. Considerando que todas as coleções implementam um tipo específico Collection, uma alternativa seria definir uma única operação removerItensInvalidos e instanciar o cursor específico para a coleção recebida como parâmetro.

O código a seguir apresenta essa implementação alternativa:

```
public void removerItensInvalidos(Collection‹ItemPedido› itens) throws Exception {
Iterator‹ItemPedido› cursor = null;
if (itens instanceof ArrayList)
cursor = new ArrayIterator((ArrayList) itens);
else if (itens instanceof HashSet)
cursor = new HashSetIterator((HashSet) itens);

if (cursor == null)
throw new Exception("tipo da coleção de itens inválido");

while (cursor.hasNext()) {
ItemPedido item = cursor.next();
if (! isValido(item)) {
cursor.remove(item);
}
}
}
```

Com essa solução, implementamos apenas uma operação **removerItensInvalidos** capaz de operar com um ArrayList ou com um HashSet. Agora, imagine que existissem vários outros tipos de coleção.

Você consegue visualizar a enorme quantidade de comandos condicionais que deveriam ser adicionados?

Portanto, esse código teria de ser modificado a cada novo tipo de implementação de coleção, acumulando uma quantidade significativa de expressões condicionais, o que é uma violação clara do princípio **Open Closed** – um dos princípios SOLID.

Além disso, esse código apresenta estruturas baseadas em downcasting, o que é um indicativo de deficiência na estrutura da solução. O downcasting está presente na conversão da coleção de itens para ArrayList ou para HashSet, dependendo do tipo da coleção recebida como parâmetro.

💬 COMENTÁRIO

Perceba como essa implementação adiciona complexidade em relação à implementação anterior. Você deve ter sempre em mente que alta complexidade também é um dos principais inimigos da evolução sustentável de um sistema.

O problema específico, portanto, consiste em implementar o método **removerItensInvalidos**, e todos os demais métodos nos quais você precise percorrer e interagir com uma coleção de objetos, de modo que ele funcione com qualquer forma de organização dessa coleção, sem que haja necessidade de recorrer a soluções baseadas em clonagem ou em estruturas condicionais complexas presentes nos exemplos apresentados.

📢 ATENÇÃO

O problema mais geral resolvido pelo Factory Method é fazer com que um módulo cliente não precise instanciar diretamente uma dentre várias possíveis implementações de uma abstração, tornando-o, portanto, dependente apenas da abstração e não de suas implementações específicas.

SOLUÇÃO DO PADRÃO FACTORY METHOD

O framework de estrutura de dados da linguagem Java implementa uma solução para esse problema por meio da aplicação do padrão Factory Method.

As estruturas de dados são classes que implementam uma interface genérica chamada Collection. São exemplos de diferentes implementações dessa interface:

ARRAYLIST

Que representa estruturas como arrays.

LINKEDLIST

Que representa estruturas como listas encadeadas.

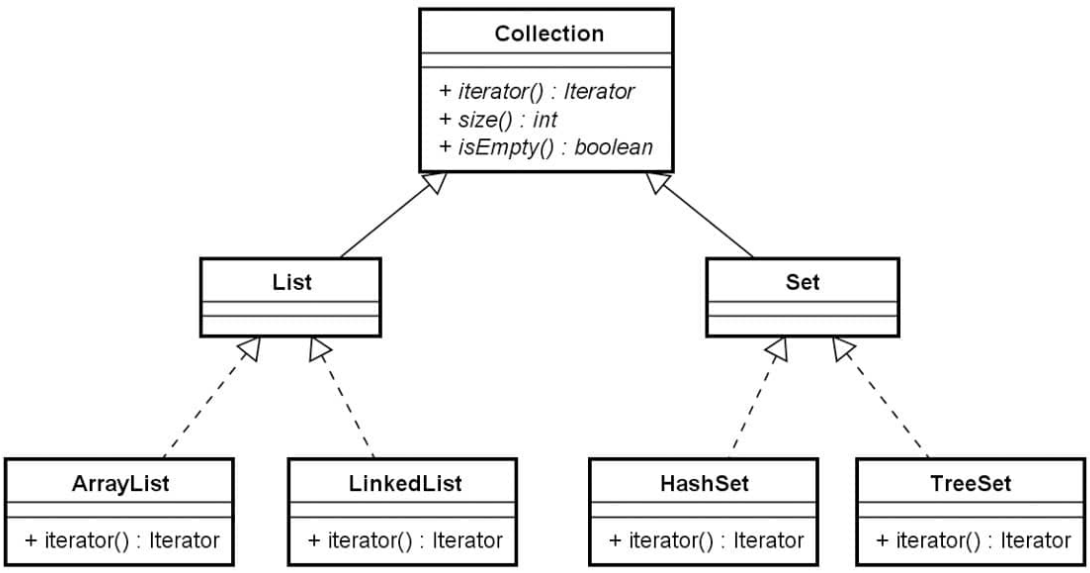
HASHSET

Que representa estruturas como conjuntos chave-valor.

TREESET

Que representa estruturas como conjuntos organizados em árvores de busca.

A organização dessas classes está ilustrada, de forma simplificada, no diagrama de classes a seguir:



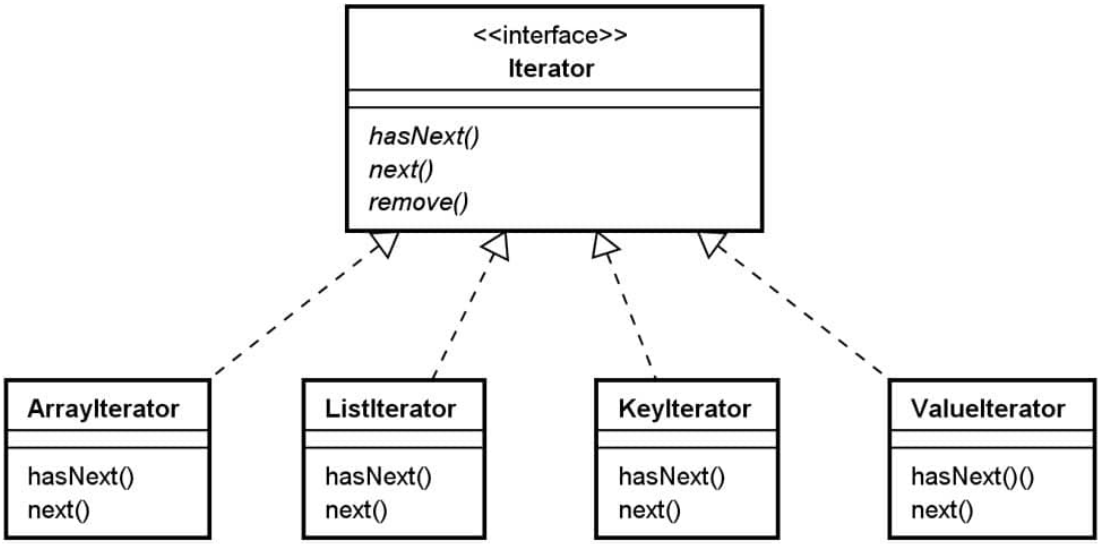
A interface **Collection** define uma operação abstrata chamada **iterator**, que é implementada em cada estrutura de dados específica. Essa operação cria e retorna um objeto que implementa a interface Iterator.

📢 ATENÇÃO

A interface Iterator define um cursor que possibilita a navegação em uma coleção de dados e a exclusão de elementos com as mesmas operações apresentadas nos exemplos anteriores, isto

é, hasNext, next e remove.

Você deve ter percebido que a implementação do percurso depende da forma com que os dados são organizados na coleção. A remoção de um elemento da coleção também é dependente da forma como seus elementos são estruturados. Isso significa que existe uma implementação da interface Iterator para cada classe que implementa a interface Collection, como você pode ver na imagem a seguir:



Dessa forma, a operação iterator de ArrayList instancia um ArrayIterator, a de **LinkedList** instancia um **ListIterator**, e assim por diante, para cada coleção específica. Esse esquema é uma simplificação, para fins didáticos, das classes realmente implementadas na linguagem Java.

Veja, no código a seguir, como você poderia implementar a operação removerItensInvalidos usando esse framework:

```
public void removerItensInvalidos(Collection‹ItemPedido› itens) {
Iterator‹ItemPedido› iterator = itens.iterator();
while (iterator.hasNext()) {
ItemPedido item = iterator.next();
if (! isValido(item)) {
itens.remove(item);
}
}
}
```

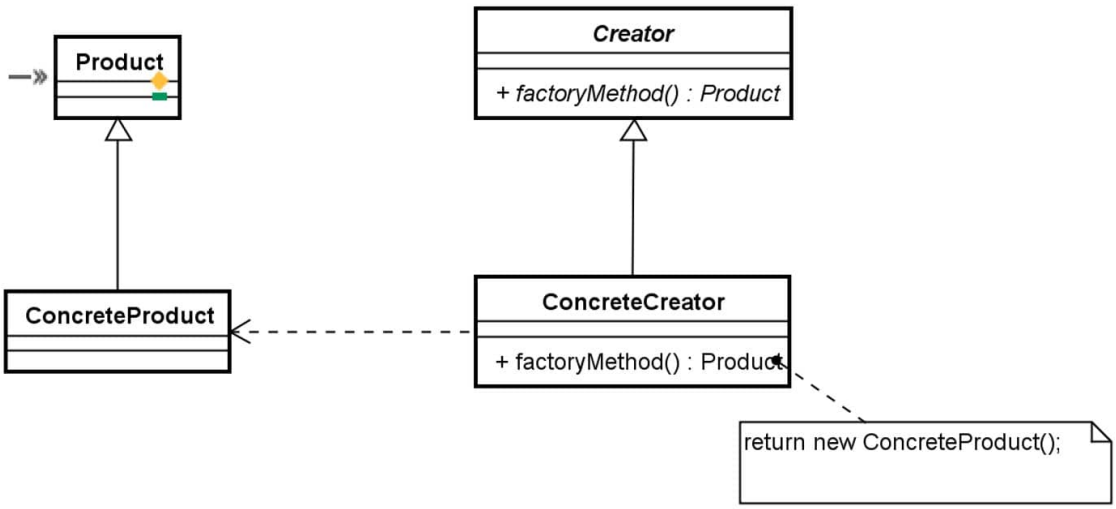
O comando **itens.iterator()** é uma chamada polimórfica a partir da interface Collection, que resulta na criação de uma das implementações específicas da interface Iterator.

Portanto, se a Collection for um ArrayList, por exemplo, esse comando será executado pela classe ArrayList, que criará um ArrayIterator e retornará essa instância (cursor), a qual será referenciada pela variável iterator.

COMENTÁRIO

Note que, nessa solução, a única responsabilidade da operação iterator é criar (fabricar) a instância da interface Iterator apropriada para a estrutura de dados. Esta é a ideia central do padrão Factory Method: definir uma operação “fábrica” na classe abstrata (Collection), deixando para cada subclasse (ArrayList, LinkedList, HashSet, TreeSet) a decisão da implementação específica da interface (Iterator) retornada pela fábrica.

Na imagem a seguir, você pode observar que a estrutura geral da solução proposta pelo padrão Factory Method define quatro participantes:



Do lado esquerdo, estão os produtos a serem fabricados. O participante **Product** corresponde ao tipo genérico do elemento a ser fabricado, enquanto **ConcreteProduct** corresponde a cada especialização do produto a ser fabricado.

No exemplo das estruturas de dados, a interface Iterator desempenha o papel de **Product**, enquanto as classes ArrayIterator, ListIterator, KeyIterator e ValueIterator desempenham o papel de **ConcreteProduct**.

Do lado direito, estão os criadores, isto é, as classes que são responsáveis pela instanciação dos produtos. O participante **Creator** define uma operação (**factoryMethod**) que retorna uma instância da interface genérica **Product**, enquanto **ConcreteCreator** representa cada

implementação concreta de **Creator** responsável pela instanciação do **ConcreteProduct** específico.

Portanto, no exemplo das estruturas de dados, a interface Collection corresponde ao participante **Creator**, e sua operação abstrata iterator é o **factoryMethod**. Já as classes ArrayList, LinkedList, HashSet e TreeSet desempenham o papel de **ConcreteCreator** e são responsáveis pela implementação da operação **factoryMethod**, na qual será feita a instanciação de um ArrayIterator, ListIterator, KeyIterator e ValueIterator (**ConcreteProduct**), respectivamente.

CONSEQUÊNCIAS E PADRÕES RELACIONADOS AO FACTORY METHOD

O padrão **Factory Method** permite que diferentes implementações de um mesmo serviço possam ser utilizadas por um cliente sem que seja necessário replicar códigos similares ou utilizar estruturas condicionais complexas, conforme ilustrado no exemplo anterior.

Além disso, esse padrão possibilita a conexão de duas hierarquias paralelas representadas pelos participantes genéricos **Creator** e **Product**.

O Factory Method é muito útil quando precisamos segregar uma hierarquia de objetos detentores de informações (objetos de domínio) dos diferentes algoritmos de manipulação dessas informações.

Portanto, esse padrão pode ser aplicado em conjunto com o padrão **Strategy**, que tem como objetivo a separação de diferentes algoritmos dos objetos de domínio sobre os quais eles atuam.

Template Method é outro padrão frequentemente utilizado em conjunto com o Factory Method.

O Template Method é uma implementação genérica definida em uma superclasse que contém passos que podem ser especializados nas subclasses.

Um desses passos pode corresponder à criação de um objeto específico, que pode ser realizada pela aplicação do padrão Factory Method.

VERIFICANDO O APRENDIZADO

MÓDULO 2

⦿ **Reconhecer o propósito, a estrutura e as situações de aplicação do padrão de projeto Abstract Factory**

INTENÇÃO DO PADRÃO ABSTRACT FACTORY

Abstract Factory é um padrão que fornece uma interface para a criação de famílias de objetos relacionados ou dependentes, sem criar dependências entre o cliente e as classes concretas instanciadas.

PROBLEMA DO PADRÃO ABSTRACT FACTORY

Imagine que você esteja trabalhando em uma implementação que tenha integrações com sistemas externos de diferentes organizações. Considere que os sistemas de cada organização externa enviem os mesmos tipos de mensagens em diferentes formatos, como, por exemplo, texto contendo campos de tamanho predefinido, XML, CSV, entre outros. Considere, ainda, que cada organização envie suas mensagens sempre no mesmo formato.

O quadro a seguir apresenta um exemplo de integração com três organizações que enviam mensagens codificadas no formato especificado:

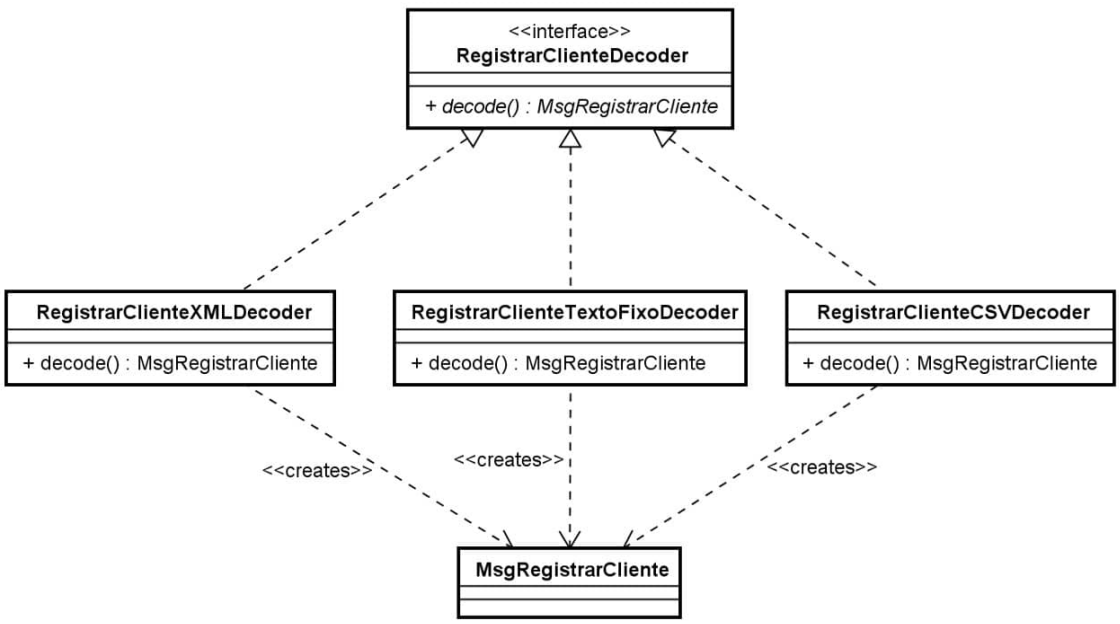
Emissor	Mensagem	Formato
Organização X	Registrar Cliente	XML
Organização Y	Registrar Cliente	CSV
Organização Z	Registrar Cliente	Campos de tamanho fixo
Organização X	Registrar Conta	XML
Organização Y	Registrar Conta	CSV
Organização Z	Registrar Conta	Campos de tamanho fixo

Atenção! Para visualizaçãocompleta da tabela utilize a rolagem horizontal

Quadro elaborado por Alexandre Correa

Podemos definir um conjunto de classes responsáveis pela decodificação de mensagens de um formato específico em objetos independentes desse formato.

A imagem a seguir apresenta a estrutura dessa solução:



A interface **RegistrarClienteDecoder** representa um serviço genérico que traduz uma mensagem de registro de cliente, recebida em um formato qualquer, para um objeto da classe **MsgRegistrarCliente**, que corresponde à representação independente de formato da mensagem recebida.

As implementações dessa interface para cada formato específico são:

XML

classe RegistrarClienteXMLDecoder

TEXTO FIXO

classe RegistrarClienteTextoFixoDecoder

CSV

classe RegistrarClienteCSVDecoder

📢 ATENÇÃO

Para cada mensagem recebida pelo sistema (como, por exemplo, Registrar Cliente, Registrar Conta), deve ser criada uma estrutura de classes similar à apresentada.

O código a seguir corresponde ao esqueleto de implementação da classe **ServicoIntegracao**: um exemplo de módulo que utiliza as classes de decodificação.

As operações dessa classe representam as mensagens recebidas das diferentes organizações.

A operação **registrarCliente**, por exemplo, recebe um texto com o conteúdo da mensagem enviada por uma origem. Essa origem é codificada em um texto (X, Y ou Z), representando as

diferentes organizações. O texto da mensagem deve ser decodificado do formato específico em uma instância da classe MsgRegistrarCliente para seu posterior tratamento.

```
public class ServicoIntegracao {
public void registrarCliente (String textoMsg, String origem) {
RegistrarClienteDecoder msgDecoder = null;

if (“X”.equals(origem)) {
msgDecoder = new RegistrarClienteXMLDecoder();
} else if ("Y".equals(origem)) {
msgDecoder = new RegistrarClienteCSVDecoder();
} else if ("Z".equals(origem)) {
msgDecoder = new RegistrarClienteTextoFixoDecoder();
}
MsgRegistrarCliente msg = msgDecoder.decode(textoMsg);
...
// código para o tratamento da mensagem recebida
}

public void registrarConta (String textoMsg, String origem) {
RegistrarContaDecoder msgDecoder = null;

if (“X”.equals(origem)) {
msgDecoder = new RegistrarContaXMLDecoder();
} else if ("Y".equals(origem)) {
msgDecoder = new RegistrarContaCSVDecoder();
} else if ("Z".equals(origem)) {
msgDecoder = new RegistrarContaTextoFixoDecoder();
}
MsgRegistrarConta msg = msgDecoder.decode(textoMsg);
...
// código para o tratamento da mensagem recebida
}
... // operações para recepção e tratamento das demais mensagens
}
```

Você consegue identificar o problema dessa implementação da classe ServicoIntegracao?

Ela está acoplada com todos os tipos possíveis de decodificadores, concentrando toda a complexidade de resolução sobre o decodificador apropriado para traduzir uma mensagem vinda de determinada origem.

💬 COMENTÁRIO

Você deve ter percebido que, caso novos formatos e origens sejam adicionados, esse código terá de ser modificado, o que configura uma violação do princípio Open Closed, um dos princípios SOLID.

Nesse exemplo, temos várias famílias de decodificadores de acordo com o formato da mensagem, como: decodificadores XML, CSV e Texto Fixo.

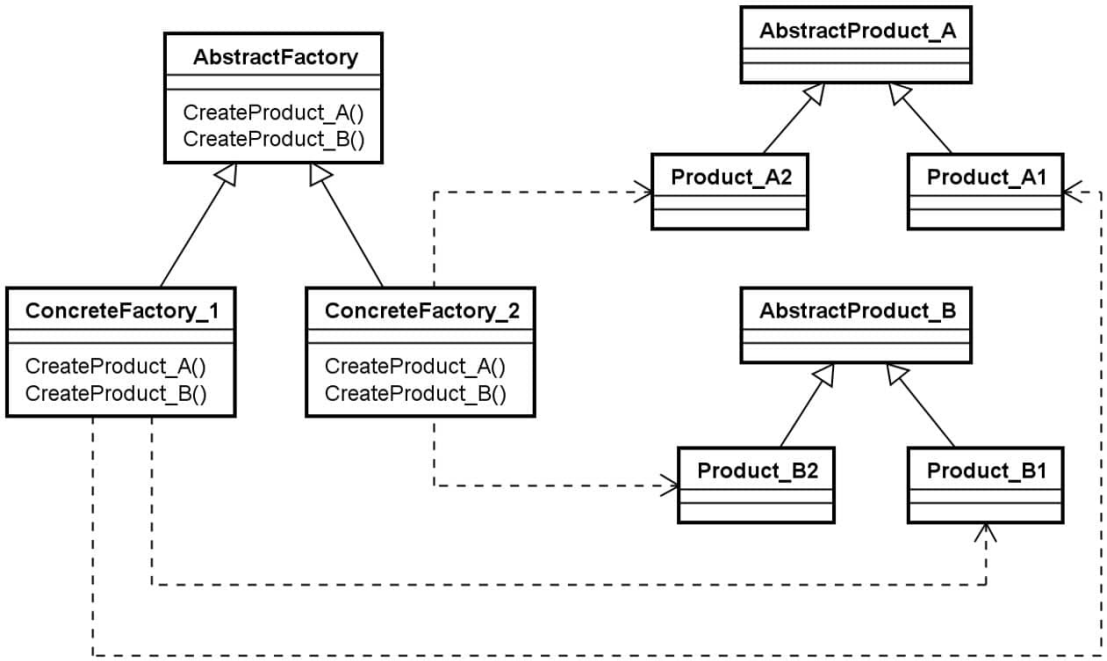
Ao recebermos uma mensagem da origem X, por exemplo, sabemos que precisamos utilizar o conversor XML correspondente a essa mensagem, pois essa origem envia todas as suas mensagens no formato XML.

Portanto, o problema tratado pelo padrão Abstract Factory consiste em isolar o cliente de uma família de produtos relacionados de suas implementações específicas, respondendo à seguinte pergunta:

Como podemos remover todas as instanciações dos decodificadores da classe ServicoIntegracao, criando uma solução genérica que permita que esse serviço trabalhe com novos formatos de decodificação sem que seu código precise ser alterado?

SOLUÇÃO DO PADRÃO ABSTRACT FACTORY

A estrutura da solução proposta pelo padrão Abstract Factory está representada no diagrama de classes a seguir:



Do lado direito, estão os vários produtos criados pelas fábricas. Cada tipo de produto é definido por uma interface genérica (AbstractProduct_A e AbstractProduct_B) e possui diversas implementações que definem os objetos específicos a serem criados pelas fábricas. Product_A1 e Product_A2, por exemplo, são implementações concretas de AbstractProduct_A.

No problema apresentado anteriormente, a interface RegistrarClienteDecoder corresponde ao participante AbstractProduct_A, enquanto as classes RegistrarClienteXMLDecoder, RegistrarClienteTextoFixoDecoder e RegistrarClienteCSVDecoder representam os produtos concretos Product_A1, Product_A2 e Product_A3.

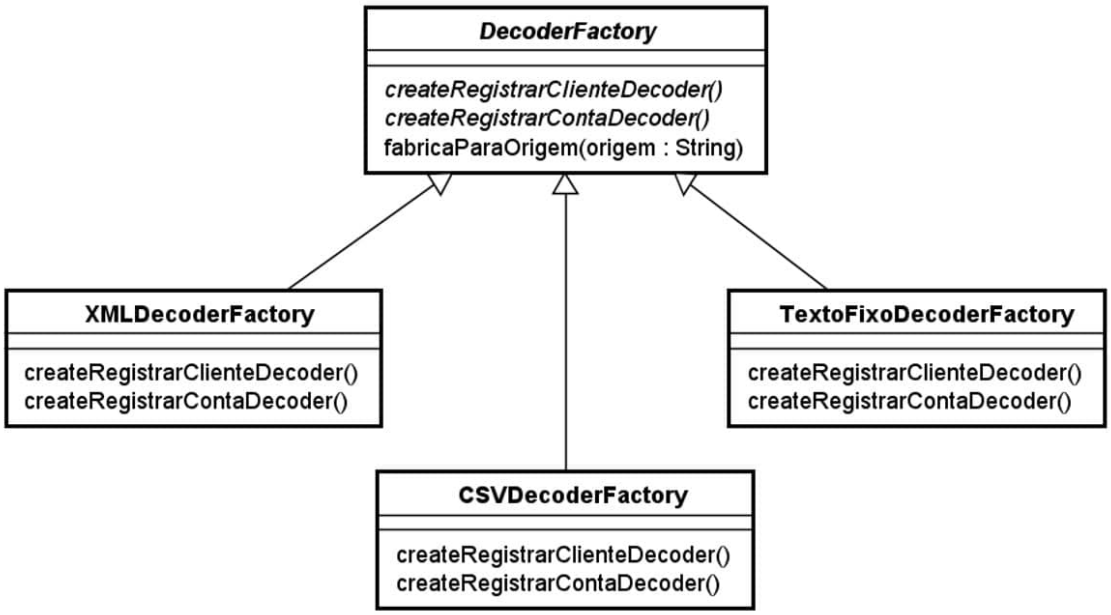
De forma análoga, a interface RegistrarContaDecoder corresponde ao participante AbstractProduct_B, enquanto as classes RegistrarContaXMLDecoder, RegistrarContaTextoFixoDecoder e RegistrarContaCSVDecoder representam os produtos concretos Product_B1, Product_B2 e Product_B3.

Do lado esquerdo, estão as fábricas. Cada fábrica é responsável por criar instâncias específicas de uma família definida por seus produtos abstratos.

Dessa forma, a fábrica ConcreteFactory_1 é responsável por criar instâncias das classes Product_A1 e Product_B1, enquanto a fábrica ConcreteFactory_2 é responsável por criar instâncias das classes Product_A2 e Product_B2. Portanto, Product_A1 e Product_B1 formam uma família de produtos, enquanto Product_A2 e Product_B2 formam outra família de produtos.

Vamos ver como ficaria a nova solução para o serviço de integração com a utilização desse padrão?

Primeiro, precisamos definir as fábricas. Podemos definir as famílias de decodificadores de acordo com o formato das mensagens, conforme ilustrado na imagem a seguir:



A classe **DecoderFactory** representa o participante **AbstractFactory** do padrão, definindo uma interface genérica para a criação dos diversos decodificadores de mensagens. Cada especialização dessa classe corresponde ao participante **ConcreteFactory** do padrão, sendo responsável pela criação dos decodificadores correspondentes a um formato específico de mensagem (XML, CSV ou TextoFixo).

A implementação dessa estrutura é apresentada a seguir. Note que a responsabilidade de cada fábrica é apenas instanciar a classe de um decodificador específico da família (XML, CSV etc.). A fábrica abstrata possui uma operação adicional (fabricaParaOrigem) que retorna a fábrica apropriada para determinada origem. Ela funciona como uma espécie de registro de todas as fábricas e suas respectivas origens.

```
public abstract class DecoderFactory {
public abstract RegistrarClienteDecoder createRegistrarClienteDecoder();
public abstract RegistrarContaDecoder createRegistrarContaDecoder();

public static DecoderFactory fabricaParaOrigem(String origem) {
if (“X”.equals(origem)) {
return new XMLDecoderFactory();
} else if ("Y".equals(origem)) {
return new CSVDecoderFactory();
} else if ("Z".equals(origem)) {
return new TextoFixoDecoderFactory();
```

```
}
}
}

public class XMLDecoderFactory extends DecoderFactory {
public RegistrarClienteDecoder createRegistrarClienteDecoder() {
return new RegistrarClienteXMLDecoder();
}
public RegistrarContaDecoder createRegistrarContaDecoder() {
return new RegistrarContaXMLDecoder();
}
}

public class CSVDecoderFactory extends DecoderFactory {
public RegistrarClienteDecoder createRegistrarClienteDecoder() {
return new RegistrarClienteCSVDecoder();
}
public RegistrarContaDecoder createRegistrarContaDecoder() {
return new RegistrarContaCSVDecoder();
}
}
```

Agora, vamos utilizar as fábricas para modificar a implementação das operações da classe **ServicoIntegracao**.

Veja, no código a seguir, que a operação **registrarCliente** chama a operação **fabricaParaOrigem**, a partir da origem recebida como parâmetro, para obter a instância da fábrica apropriada para as mensagens recebidas dessa origem.

Na sequência, a chamada para a operação **createRegistrarClienteDecoder** da fábrica cria o decodificador específico para a mensagem RegistrarCliente.

```
public class ServicoIntegracao {
public void registrarCliente (String textoMsg, String origem) {
DecoderFactory decoderFactory = DecoderFactory.fabricaParaOrigem(origem);
RegistrarClienteDecoder msgDecoder = decoderFactory.createRegistrarClienteDecoder();
MsgRegistrarCliente msg = msgDecoder.decode(textoMsg);
...
```

```
// código para tratamento da mensagem MsgRegistrarCliente
}

public void registrarConta (String textoMsg, String origem) {
DecoderFactory decoderFactory = DecoderFactory.fabricaParaOrigem(origem);
RegistrarContaDecoder msgDecoder = decoderFactory.createRegistrarContaDecoder();
MsgRegistrarConta msg = msgDecoder.decode(textoMsg);
...
// código para tratamento da mensagem MsgRegistrarConta
}

... // código para demais mensagens
}
```

Note que esse código não precisará ser modificado para novos formatos de mensagem, pois bastará adicionar novos decodificadores e definir uma nova fábrica concreta. Além disso, a estrutura do código ficou muito mais enxuta e menos complexa.

Esse padrão é utilizado, por exemplo, na implementação do framework **AWT** de interface com o usuário da linguagem Java. Os componentes visuais específicos de plataforma, como Windows e **Motif**, por exemplo, formam uma família de produtos (Button, Frame, Panel etc.). A classe Toolkit corresponde à fábrica abstrata que oferece operações de criação de cada componente visual. Cada plataforma é implementada em uma subclasse de Toolkit específica.

AWT

Abstract Window Toolkit é o toolkit gráfico original da linguagem de programação Java.

MOTIF

Interface gráfica padrão para usuários de sistema operacional Unix.

CONSEQUÊNCIAS E PADRÕES RELACIONADOS AO ABSTRACT FACTORY

O padrão Abstract Factory promove o encapsulamento do processo de criação de objetos, isolando os clientes das implementações concretas, permitindo que os clientes sejam implementados pelo uso apenas de abstrações.

Além disso, esse padrão promove a consistência entre produtos relacionados, isto é, produtos da mesma família que devem ser utilizados em conjunto. Entretanto, a introdução de novos produtos não é simples, pois exige mudança em todas as fábricas.

📢 ATENÇÃO

Cada novo produto inserido exige a adição de uma nova operação de criação em cada fábrica da estrutura.

O padrão Abstract Factory está relacionado com outros padrões de criação. Cada operação de criação é tipicamente implementada utilizando o padrão Factory Method.

É possível configurar fábricas mais flexíveis utilizando o padrão **Prototype**. Cada fábrica concreta pode ser definida como um **Singleton**, já que, normalmente, apenas uma instância de uma fábrica específica precisa ser instanciada.

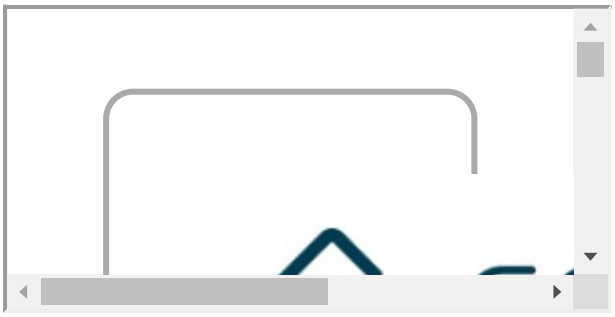
No exemplo apresentado neste módulo, poderíamos ainda eliminar a duplicação de código similar presente nas operações **registrarCliente** e **registrarConta**, generalizando a fábrica para retornar objetos de um tipo genérico **Decoder** (ao invés de decodificadores específicos) e transformando essas operações em objetos por meio da aplicação de outros padrões, como o **Command** e o **Template Method**, por exemplo.

Desafio

Estude os padrões mencionados e tente modificar a estrutura do exemplo, aplicando-os na nova solução.

ABSTRACT FACTORY X INJEÇÃO DE DEPENDÊNCIAS

Assista ao vídeo a seguir. Nele, apresentamos situações nas quais um framework de injeção de dependências pode substituir uma implementação manual do padrão Abstract Factory, assim como situações em que as duas abordagens podem ser combinadas.



VERIFICANDO O APRENDIZADO

MÓDULO 3

⦿ **Reconhecer o propósito, a estrutura e as situações de aplicação do padrão de projeto Builder**

INTENÇÃO DO PADRÃO BUILDER

Builder é um padrão que visa separar a construção de um objeto complexo de sua representação, de forma que o mesmo processo de construção possa construir diferentes representações desse objeto.

PROBLEMA DO PADRÃO BUILDER

Suponha que você esteja fazendo um sistema para uma corretora de valores mobiliários, e que esse sistema permita que o cliente exporte suas notas de negociação em diferentes formatos, tais como: **XML**, **PDF** ou **XLS**.

Imagine que o processo de construção de qualquer representação da nota de negociação seja definido por três passos fundamentais:

Construir o cabeçalho da nota.

Listar as operações da nota.

Gerar o sumário com os totais e taxas de todas as operações do dia.

Uma solução frequente utilizada para tal problema é definir todas as possíveis conversões em uma única classe, como ilustrado no código a seguir:

```
public class ExportadorNota {
public byte[] exportarNota(NotaNegociacao nota, String formato) {
if (“XML”.equals(formato))
return gerarNotaXML(nota);
else if (“PDF”.equals(formato))
return gerarNotaPDF(nota);
else if (“XLS”.equals(formato))
return gerarNotaXLS(nota);
}

private byte[] gerarNotaXML(NotaNegociacao nota) {
// construir cabeçalho em XML
// listar os itens da nota em XML
// gerar sumário em XML
// retornar conteúdo da nota no formato XML
```

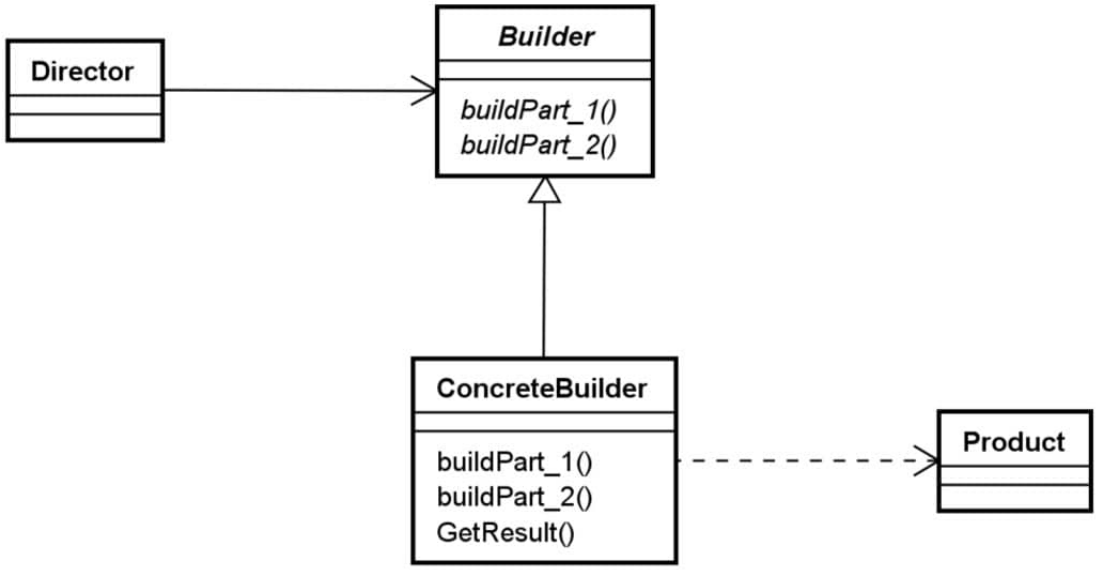
```
}

private byte[] gerarNotaPDF(NotaNegociacao nota) {
// construir cabeçalho em PDF
// listar os itens da nota em PDF
// gerar sumário em PDF
// retornar conteúdo da nota no formato PDF
}

private byte[] gerarNotaXLS(NotaNegociacao nota) {
// construir cabeçalho em XLS
// listar os itens da nota em XLS
// gerar sumário em XLS
// retornar conteúdo da nota no formato XLS
}

}
```

A operação **exportarNota** recebe a nota de negociação a ser exportada e o formato de exportação (XML, PDF ou XLS).

A solução apresentada não é adequada, pois, além de concentrar em um único módulo todas as possíveis representações de exportação da nota de negociação, o algoritmo de construção é repetido em cada formato específico. Além disso, o módulo deve ser modificado a cada nova forma de representação que for necessária para a nota, violando o princípio Open Closed, um dos princípios SOLID.

SOLUÇÃO DO PADRÃO BUILDER

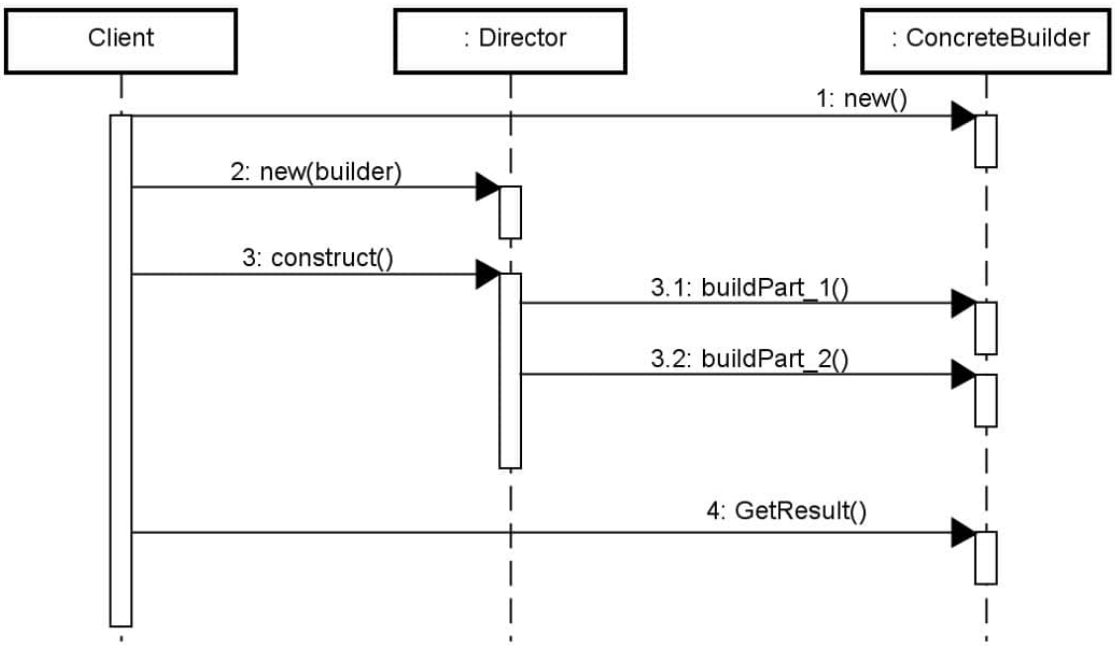
A solução proposta pelo padrão Builder consiste em separar a criação de objetos complexos de quem demanda esses objetos, conforme a estrutura definida no diagrama de classes a seguir:



A interface Builder define as operações que criam as diferentes partes de um produto. Cada forma particular de criação desse produto é definida em uma classe **ConcreteBuilder**, que implementa as operações específicas para a criação das partes definidas na interface Builder.

O participante **Director** corresponde à classe que constrói o produto utilizando a interface Builder. Nessa solução, a classe Director fica isolada do conhecimento sobre as diferentes formas de representação do produto a ser construído.

O diagrama de sequência a seguir ilustra a colaboração entre os participantes do padrão Builder:



Nessa colaboração, o elemento **Client** representa o objeto que solicita a criação de um produto para o Director. Para isso, ele cria o Builder específico para o produto desejado

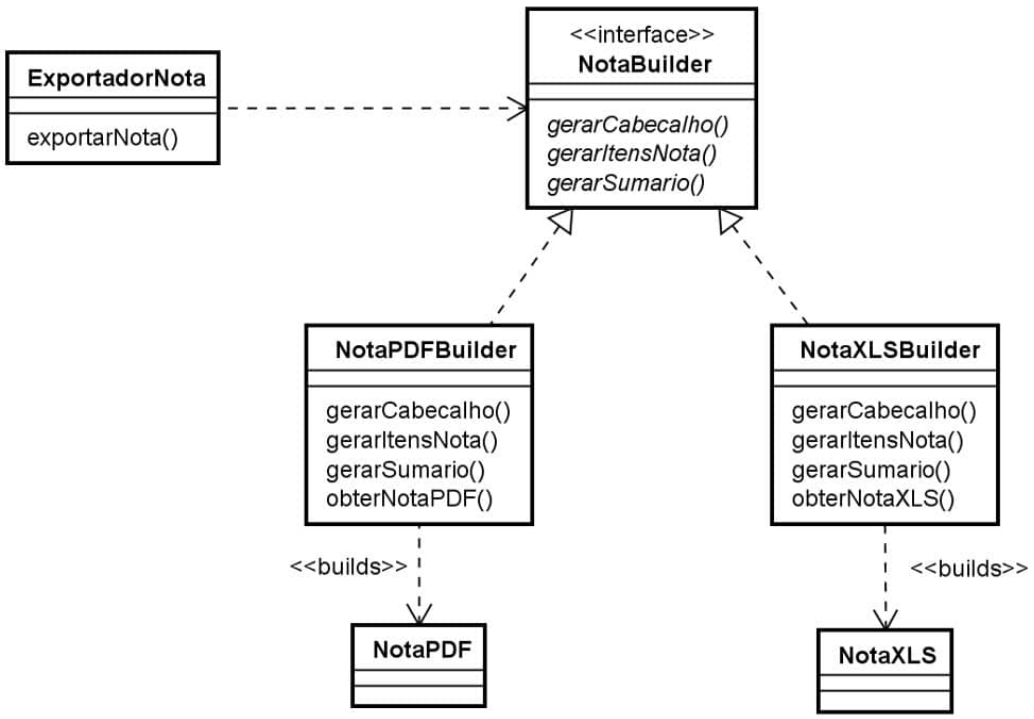
(ConcreteBuilder), injetando-o na instanciação da classe Director.

A partir daí, o objeto Director é responsável por criar as diferentes partes do produto, chamando as operações específicas do ConcreteBuilder (buildPart_1, buildPart_2 etc.).

O Builder concreto adiciona as partes solicitadas pelo Director, pois somente ele conhece os detalhes da representação do produto.

Ao final, o elemento Client pede o produto construído para o ConcreteBuilder por meio da operação GetResult.

O diagrama a seguir apresenta a aplicação do padrão Builder no problema apresentado. A classe **ExportadorNota** corresponde ao participante Director na estrutura definida pelo padrão. **NotaBuilder** representa a interface Builder, enquanto **NotaPDFBuilder** e **NotaXLSBuilder** correspondem ao participante ConcreteBuilder. Cada builder específico constrói uma representação específica do produto (**NotaPDF** e **NotaXLS**).



O código a seguir ilustra a estrutura da implementação da solução, utilizando o padrão Builder.

A classe ExportadorNota recebe um builder em seu construtor. Esse builder é utilizado no método exportarNota para gerar as partes que compõem uma nota exportada tanto em PDF quanto em XLS, isto é, o cabeçalho, os itens negociados e o sumário.

```
public class ExportadorNota {
private NotaBuilder builder;
public ExportadorNotaNegociacao(NotaBuilder builder) {
```

```
this.builder = builder;
}
public void exportarNota(NotaNegociacao nota) {
builder.gerarCabecalho(nota);
builder.gerarItensNota(nota);
builder.gerarSumario(nota);
}
}
```

A classe **ComandoExportarNotaPDF** é um exemplo de cliente do exportador de nota. O método executar instancia um builder concreto (NotaPDFBuilder), cria um diretor (ExportadorNota), passando o builder a ser utilizado, e chama a operação de construção do produto desejado (exportarNota). O último passo é solicitar ao builder concreto o objeto NotaPDF construído.

```
public class ComandoExportarNotaPDF {
public NotaPDF executar(NotaNegociacao nota) {
NotaPDFBuilder builder = new NotaPDFBuilder();
ExportadorNota diretor = new ExportadorNota (builder);
diretor.exportarNota(nota);
return builder.obterNotaPDF();
}
}
```

💬 COMENTÁRIO

Note que, nessa solução, o algoritmo geral de exportação é definido apenas na classe ExportadorNota. Além disso, a estrutura condicional baseada no formato desejado, presente na solução original, não é mais necessária. Dessa forma, novas representações de exportação da nota de negociação podem ser adicionadas ao sistema, bastando adicionar novos builders e produtos correspondentes.

Existem algumas questões importantes na implementação desse padrão.

A primeira é se o objeto Builder deve dar acesso apenas ao produto pronto, isto é, após a realização de todas as etapas de construção, ou se ele pode dar acesso às partes intermediárias já construídas do produto.

No exemplo apresentado, como o objeto que desempenha o papel de Director não precisa acessar as partes em seu algoritmo de construção, cada Builder concreto fornece acesso apenas ao produto construído por meio das operações obterNotaPDF e obterNotaXLS. Entretanto, caso necessário, é admissível que as operações de construção (buildPart_1, buildPart_2 etc.) retornem uma parte intermediária do produto.

Outra questão é se os produtos devem ser estruturados em uma hierarquia. No exemplo, as classes NotaPDF e NotaXLS não foram definidas com uma superclasse comum, pois assumimos que elas seriam utilizadas de forma bem específica. Entretanto, nada impede que elas sejam derivadas de uma superclasse ou implementem uma interface genérica.

CONSEQUÊNCIAS E PADRÕES RELACIONADOS AO BUILDER

O padrão Builder é aplicável na construção de objetos complexos e que possam ter diferentes representações internas. Encapsulando o conhecimento dessas representações em builders concretos que implementam uma interface genérica comum, os clientes ficam isolados da forma como esses objetos são internamente construídos.

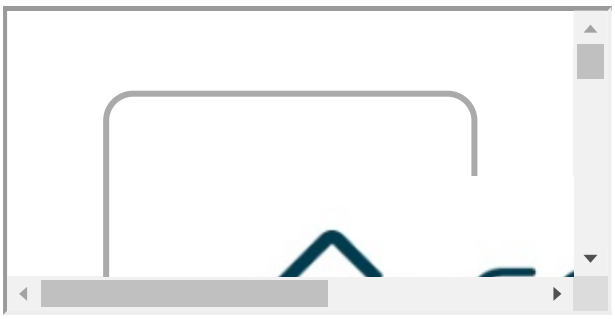
📢 ATENÇÃO

O padrão Abstract Factory, assim como o Builder, pode construir objetos complexos. A diferença principal entre os dois padrões é que o Builder oferece um mecanismo de construção de um objeto complexo em etapas, enquanto o foco do Abstract Factory é definir famílias de produtos. Um produto da família é retornado com apenas uma chamada de operação.

O padrão **Composite** é utilizado para representar objetos compostos por outros em uma hierarquia de especializações de um mesmo elemento comum, como ocorre, por exemplo, em uma estrutura de diretórios e arquivos. O padrão Builder pode ser utilizado para implementar a construção de objetos com uma estrutura de composição complexa resultante da utilização do padrão Composite.

APLICAÇÕES DO PADRÃO DE PROJETO BUILDER

No vídeo a seguir, abordamos algumas situações típicas do uso do Builder, como, por exemplo, para evitar o uso de “construtores telescópicos” ou na criação de estruturas em árvore com elementos de diferentes tipos.



VERIFICANDO O APRENDIZADO

MÓDULO 4

⊙ **Reconhecer o propósito, a estrutura e as situações de aplicação dos padrões de projeto Prototype e Singleton**

INTENÇÃO DO PADRÃO PROTOTYPE

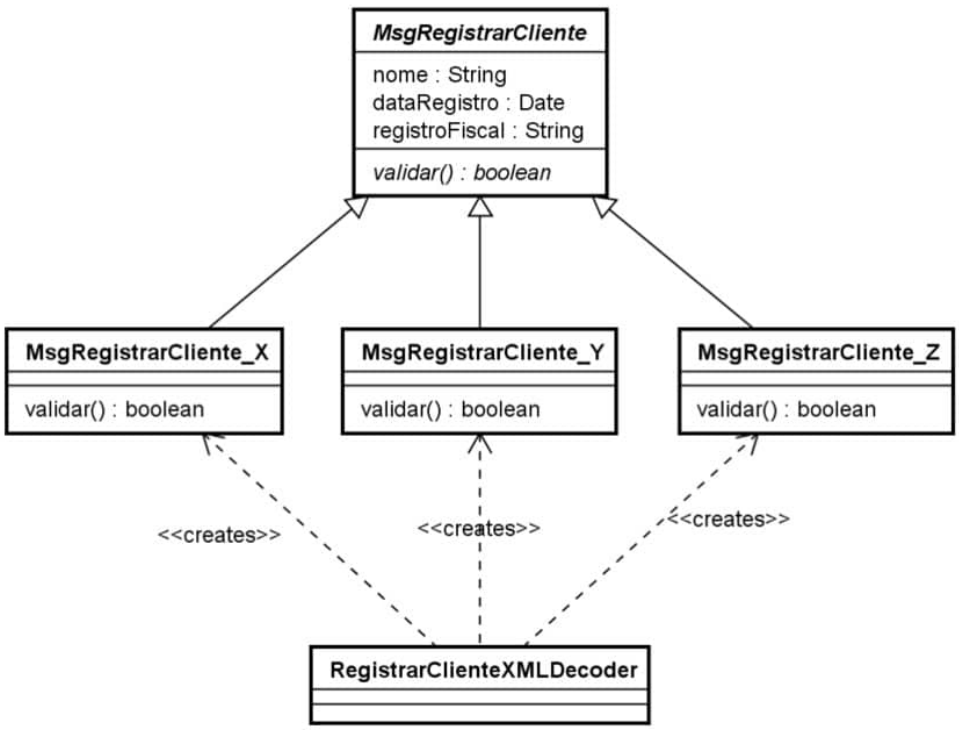
O padrão Prototype permite a instanciação de objetos a partir da geração de uma cópia de um objeto protótipo, fazendo com que o módulo cliente não precise conhecer a classe específica que está sendo instanciada.

PROBLEMA DO PADRÃO PROTOTYPE

Suponha que, no problema apresentado no padrão Abstract Factory, os decodificadores da mensagem Registrar Cliente, ao invés de criarem apenas objetos da classe **MsgRegistrarCliente**, tivessem de criar objetos de classes específicas conforme a origem da mensagem.

Isso significa que a classe **RegistrarClienteXMLDecoder**, por exemplo, ao invés de criar uma instância de MsgRegistrarCliente, teria de criar uma instância de **MsgRegistrarCliente_X**, **MsgRegistrarCliente_Y** ou **MsgRegistrarCliente_Z**, dependendo da organização origem da mensagem, imaginando que a validação de cada registro de cliente variasse conforme a organização.

O diagrama a seguir ilustra essa solução. Definimos uma especialização de MsgRegistrarCliente para cada origem (veja os sufixos X, Y e Z definidos nas subclasses). Cada subclasse implementa um método específico de validação da mensagem.



Você consegue perceber que essa solução adiciona complexidade ao decodificador?

Veja, no código a seguir, como a classe RegistrarClienteXMLDecoder fica mais complexa, uma vez que tem de conhecer cada subclasse de MsgRegistrarCliente:

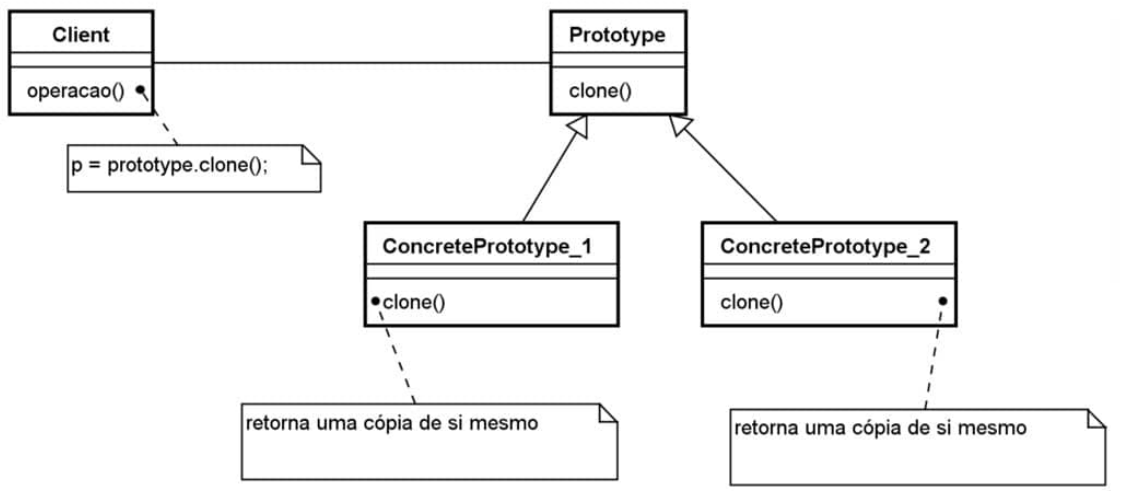
```
public class RegistrarClienteXMLDecoder {
public MsgRegistrarCliente decode(String textoMsg, String origem) {
```

```
MsgRegistrarCliente msg;
if (“X”.equals(origem)) {
msg = new MsgRegistrarCliente_X();
else if (“Y”.equals(origem)) {
msg = new MsgRegistrarCliente_Y();
else if (“Z”.equals(origem)) {
msg = new MsgRegistrarCliente_Z();
}
// … aqui viria o código de decodificação e preenchimento dos atributos
// do objeto MsgRegistrarCliente
return msg;
}
}
```

Inserir o processo de decisão sobre o objeto a ser instanciado na implementação do método de decodificação da mensagem, além de adicionar complexidade, torna a implementação inflexível à adição de novas origens, pois teríamos de modificar o código inserindo novos comandos condicionais, o que é uma clara violação do princípio Open Closed, um dos princípios SOLID.

SOLUÇÃO DO PADRÃO PROTOTYPE

O diagrama a seguir ilustra a estrutura da solução proposta pelo padrão Prototype:



COMENTÁRIO

A ideia central do padrão é fazer com que uma classe cliente que precise criar instâncias de uma subclasse específica ou de diferentes subclasses registre uma instância protótipo dessa(s) subclasse(s) e chame a operação clone do protótipo registrado sempre que precisar de uma nova instância.

A operação clone é definida em cada subclasse e retorna para o módulo cliente uma nova instância com uma réplica de seu estado. Dessa forma, o módulo cliente não sabe qual subclasse específica foi instanciada, e novas subclasses podem ser adicionadas ao esquema, sem que o cliente precise ser modificado.

Você consegue visualizar como ficaria a solução do problema apresentado com a aplicação da estrutura proposta pelo padrão Prototype?

Em Java, todo objeto já oferece uma implementação padrão para a operação clone, conhecida pelo termo **shallow copy**. Essa implementação padrão apenas copia os valores dos atributos de um objeto para sua réplica.

Se um objeto Venda, por exemplo, possuir um atributo que seja uma referência para um objeto Cliente relacionado, a cópia desse objeto Venda compartilhará com o objeto original a referência para o mesmo objeto Cliente, ou seja, em uma shallow copy, os objetos referenciados pelo objeto original não são clonados.

Se você precisar criar cópias dos objetos referenciados, deverá criar uma implementação específica da operação clone, sobrepondo a implementação padrão disponível na classe Object. Esse processo de geração da cópia de toda a árvore de objetos relacionados ao objeto que está sendo clonado é conhecido pelo termo deep copy.

Em nosso exemplo, vamos utilizar a cópia padrão já oferecida pela classe Object. Portanto, não precisaremos modificar as subclasses de MsgRegistrarCliente.

O próximo passo é criar e registrar as instâncias protótipo de cada subclasse de MsgRegistrarCliente, associando-as com a respectiva origem. Faremos isso criando um HashMap e associando o código da origem com a respectiva instância protótipo, conforme o código a seguir. Antes de criar um decodificador para mensagens XML, a fábrica cria as

instâncias protótipo de cada subclasse, passando-as para o construtor da classe RegistrarClienteXMLDecoder.

```
public RegistrarClienteDecoder createRegistrarClienteDecoder() {
HashMap‹String, MsgRegistrarCliente›prototypes;
prototypes.put(“X”, new MsgRegistrarCliente_X());
prototypes.put(“Y”, new MsgRegistrarCliente_Y());
prototypes.put(“Z”, new MsgRegistrarCliente_Z());

return new RegistrarClienteXMLDecoder(prototypes);
}
```

Agora, modificamos a classe RegistrarClienteXMLDecoder, de forma que seu construtor passe a receber essas instâncias das subclasses de MsgRegistrarCliente, isto é, os protótipos de cada subclasse associados às respectivas origens. Além disso, substituímos todo o código condicional existente na versão anterior por uma única chamada à operação clone da instância de MsgRegistrarCliente associada à origem recebida como parâmetro da operação decode.

```
public class RegistrarClienteXMLDecoder {
private HashMap‹String, MsgRegistrarCliente› prototypes;

public RegistrarClienteXMLDecoder(HashMap‹String, MsgRegistrarCliente›prototypes) {
this.prototypes = prototypes;
}

public MsgRegistrarCliente decode(String textoMsg, String origem) {
MsgRegistrarCliente prototype = prototypes.get(origem);

MsgRegistrarCliente msg = (MsgRegistrarCliente) prototype.clone();

// … aqui viria o código de decodificação e preenchimento dos atributos
// do objeto MsgRegistrarCliente
return msg;
}
```

Você percebeu que esse código, agora, pode instanciar novas subclasses de MsgRegistrarCliente, sem que seja necessário modificá-lo?

Nessa solução, a classe MsgRegistrarCliente desempenha o papel de Prototype, e cada subclasse de MsgRegistrarCliente desempenha o papel de ConcretePrototype. A classe RegistrarClienteXMLDecoder corresponde ao participante Client definido na estrutura do padrão.

CONSEQUÊNCIAS E PADRÕES RELACIONADOS AO PROTOTYPE

O padrão Prototype é aplicável em pelo menos três situações específicas:

1. QUANDO EXISTIREM MUITAS FÁBRICAS ESPECÍFICAS PARA A CRIAÇÃO DE DIFERENTES FAMÍLIAS DE PRODUTOS

Esse padrão permite uma solução sem que haja necessidade de implementar uma subclasse para cada família. Basta definir uma única classe fábrica e criar uma instância para cada família configurada com os protótipos que serão clonados.

2. QUANDO AS INSTÂNCIAS DE UMA CLASSE FOREM RESULTADO DE POUCAS COMBINAÇÕES DE ESTADO

Neste caso, é mais simples criar as instâncias típicas a priori e gerar cópias delas, ao invés de instanciá-las manualmente.

3. QUANDO O ESTADO DE UMA CLASSE ENVOLVER MUITOS ATRIBUTOS E RELACIONAMENTOS COM UM PROCESSO DE CRIAÇÃO DE NOVAS INSTÂNCIAS MUITO CUSTOSO OU COMPLEXO

Neste caso, necessitamos criar objetos com estados idênticos ou com poucas diferenças.

Enquanto o padrão Factory Method define uma hierarquia de classes de criação paralela às classes produto que são instanciadas, o padrão Prototype substitui essa hierarquia e a chamada a um método fábrica pelo registro de uma instância protótipo e sua posterior clonagem.

📢 ATENÇÃO

O padrão Prototype permite a criação de fábricas flexíveis que podem ter sua configuração de instâncias definida e modificada em tempo de execução, ao contrário da solução dada pelo padrão Abstract Factory, que é estática.

Em contrapartida, o padrão Prototype demanda que cada subclasse do produto a ser instanciado implemente a operação clone, o que pode ser complexo ou difícil, especialmente nos casos de utilização de classes de terceiros ou compartilhadas com outros sistemas.

Além disso, os efeitos colaterais oriundos de uma cópia baseada em uma estratégia shallow copy e a complexidade de implementar uma estratégia **deep copy**, especialmente quando existir uma árvore complexa de relacionamentos ou relacionamentos circulares, podem trazer dificuldades à implementação desse padrão.

INTENÇÃO DO PADRÃO SINGLETON

O propósito do padrão Singleton é garantir que exista uma (e apenas uma) instância de uma classe, provendo um ponto de acesso global a essa instância.

PROBLEMA DO PADRÃO SINGLETON

Suponha uma situação na qual você queira garantir que apenas uma instância de uma classe possa existir em determinado processo, como, por exemplo, no gerenciamento de recursos como cache de objetos, log, conexões com banco de dados e objetos que representem recursos compartilhados por todo o processo.

Uma possível solução seria definir uma variável global, referenciando o objeto a ser compartilhado. Dessa forma, todos os módulos que precisassem desse objeto fariam o acesso via essa variável global compartilhada. O problema é que nada impediria outros módulos de criar múltiplas instâncias dessa classe.

Outra solução seria definir com o escopo de classe todas as operações da classe cujo objeto único deve ser compartilhado. Em Java, isso significa definir todas as operações da classe com

o modificador static. Essa solução, porém, não é flexível, pois não admite a definição de subclasses e a utilização de **polimorfismo**.

SOLUÇÃO DO PADRÃO SINGLETON

O diagrama a seguir apresenta a estrutura do padrão Singleton.

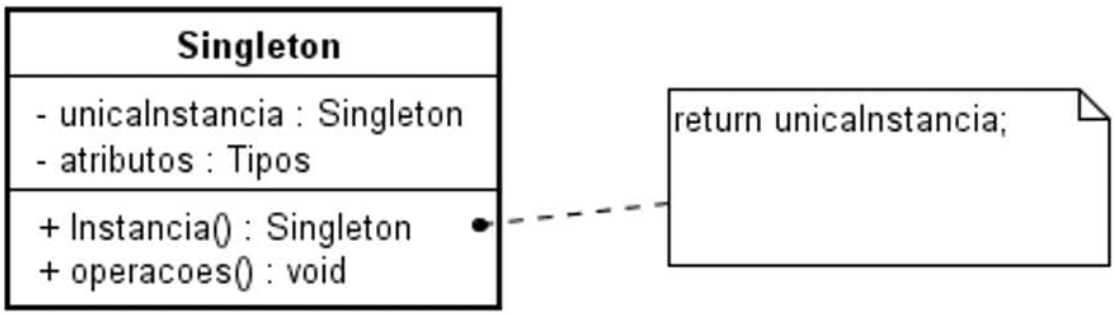
POLIMORFISMO

Princípio do modelo orientado a objetos, pelo qual duas ou mais subclasses de uma mesma superclasse podem conter métodos com a mesma assinatura, mas com implementações diferentes, resultando em comportamentos especializados para cada subclasse.

O nome **Singleton** representa o nome da classe que você deseja que tenha apenas uma instância.

O atributo **unicaInstancia** é uma referência a essa única instância a ser compartilhada pelos demais módulos.

O construtor dessa classe deve ser privativo, garantindo que outros módulos não possam instanciá-la diretamente. Tanto a operação **Instancia** quanto o atributo unicaInstancia são propriedades com escopo de classe (static).



Um possível uso do padrão Singleton consiste na implementação do padrão Abstract Factory. Veja, no código a seguir, a implementação de uma fábrica concreta utilizando o padrão Singleton. A instância compartilhada é referenciada pelo atributo factory, definido com o modificador static. O construtor da classe é definido como private, o que impede que ela seja

diretamente instanciada em outros módulos. A operação getFactory retorna a instância única compartilhada.

```
public class XMLDecoderFactory extends DecoderFactory {
private static DecoderFactory factory = null;
private XMLDecoderFactory() {
}
public static DecoderFactory getFactory() {
if (factory == null)
factory = new XMLDecoderFactory();
return factory;
}
public abstract RegistrarClienteDecoder createRegistrarClienteDecoder() {
return new RegistrarClienteXMLDecoder();
}
public abstract RegistrarContaDecoder createRegistrarContaDecoder() {
return new RegistrarContaXMLDecoder();
}
```

Podemos definir a fábrica abstrata como um registro dos diversos singletons correspondentes às fábricas concretas.

Veja, no exemplo a seguir, que as fábricas concretas são registradas em um **HashMap** codificado pela origem (X, Y ou Z). Cada entrada dessa estrutura de dados associa uma origem ao singleton da respectiva fábrica concreta. A operação **getInstance** acessa essa estrutura para retornar a fábrica concreta correspondente à origem recebida como parâmetro.

```
public abstract class DecoderFactory {
private static HashMap‹String, DecoderFactory› factoryMap;

static {
factoryMap = new HashMap‹›();
factoryMap.put(“X”, XMLDecoderFactory.getInstance());
factoryMap.put(“Y”, CSVDecoderFactory.getInstance());
factoryMap.put(“Z”, TextoLivreDecoderFactory.getInstance());
}

public static DecoderFactory getInstance(String origem) {
```

```
return factoryMap.get(origem);
}

public abstract RegistrarClienteDecoder createRegistrarClienteDecoder();
public abstract RegistrarContaDecoder createRegistrarContaDecoder();
}
```

CONSEQUÊNCIAS E PADRÕES RELACIONADOS AO SINGLETON

O padrão Singleton permite o acesso controlado a uma única instância de uma classe, sendo uma solução superior à utilização de variáveis globais. Permite, inclusive, a criação de subclasses mais específicas sem impacto para os módulos que utilizam a instância Singleton.

O padrão Singleton é frequentemente utilizado em conjunto com o padrão Abstract Factory, conforme ilustrado no exemplo anterior.

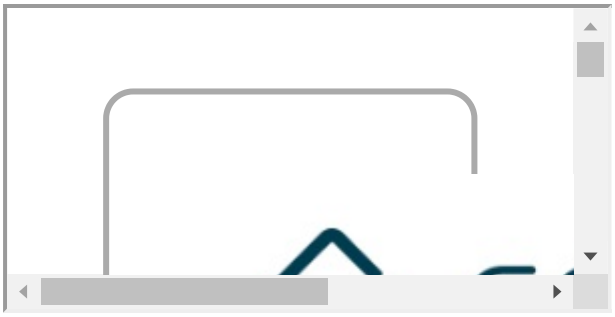
Entretanto, após o surgimento de abordagens fortemente baseadas na construção de testes unitários automatizados e na aplicação de princípios como o da inversão de dependências, o padrão Singleton passou a ser visto como um potencial problema. Ele pode dificultar a implementação de testes unitários, visto que a unidade a ser testada pode estar acoplada a Singletons que dificultam o isolamento da unidade em relação às suas dependências.

Além disso, existem linguagens que permitem quebrar o objetivo original do padrão, pois construções como reflection e serialização permitem a criação independente de objetos de classes Singleton.

Portanto, esse é um padrão que deve ser utilizado apenas em casos muito específicos para não criar acoplamentos desnecessários que tornem a estrutura do software menos flexível e dificultem o processo de testes e depuração dos módulos.

SINGLETON: PATTERN OU ANTI-PATTERN

No vídeo a seguir, apresentamos os aspectos positivos e negativos do Singleton e por que ele é considerado por muitos como um anti-pattern.



VERIFICANDO O APRENDIZADO

CONCLUSÃO

CONSIDERAÇÕES FINAIS

Neste conteúdo, vimos como os padrões de projeto GoF de criação podem ser usados em soluções de projeto de software mais flexíveis e menos acopladas.

O padrão Factory Method é baseado em um modelo em que as subclasses implementam uma interface padrão definida na superclasse para a instanciação dos objetos específicos. Os padrões Abstract Factory, Builder e Prototype delegam a responsabilidade pela criação de objetos para classes específicas com essa finalidade. O padrão Abstract Factory sugere a criação de uma hierarquia de fábricas responsável pela instanciação de uma hierarquia paralela de produtos.

O padrão Builder é aplicável na construção de objetos complexos, compostos por muitas partes e com um processo de construção custoso e complexo, isolando os módulos clientes dessa complexidade.

O padrão Prototype é baseado na geração de cópias de objetos protótipos pré-fabricados e mais voltado para a composição de objetos prontos, ao contrário do Abstract Factory, que é

baseado em uma estrutura estática de hierarquia de classes.

Por fim, o padrão Singleton já foi bastante utilizado, mas, atualmente, é considerado por muitos um antipadrão, isto é, uma solução inadequada e que deve ser evitada, com exceção de situações muito específicas de gerenciamento de recursos que não podem ser utilizados de forma simultânea.

PODCAST

Ouça o podcast e descubra por que é importante conhecer os padrões GoF de criação.

AVALIAÇÃO DO TEMA

REFERÊNCIAS

GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. **Design Patterns:** Elements of Reusable Object-Oriented Software. 1. ed. Boston: Addison-Wesley, 1994.

MARTIN, R. C. **Clean Architecture:** A Craftsman´s Guide to Software Structure and Design. 1. ed. Upper Saddle River, NJ: Prentice Hall, 2017.

METSKER, S. J.; WAKE, W. C. **Design Patterns in Java.** 1.ed. Boston: Addison-Wesley, 2006.

EXPLORE+

Para saber mais sobre a programação orientada a objetos, acesse o site da DevMedia e leia o artigo intitulado *Utilização dos princípios SOLID na aplicação de padrões de projeto*.

O site “Padrões de projeto/Design patterns – Refactoring.Guru” apresenta um conteúdo interativo e bastante completo de todos os padrões GoF com exemplos de código em diversas linguagens de programação.

Além dos padrões GoF tradicionais, outros padrões voltados para o desenvolvimento de aplicações corporativas em Java EE podem ser encontrados no livro *Java EE 8 Design Patterns and Best Practices*, escrito por Rhuan Rocha e João Purificação. A obra aborda padrões para interface com o usuário, lógica do negócio, integração de sistemas, orientação a aspectos, programação reativa e microsserviços.

CONTEUDISTA

Alexandre Luis Correa

🔗 CURRÍCULO LATTES