**Roteiro 09**

Neste roteiro vamos trabalhar com o padrão Abstract Factory. Para entender bem a lógica deste padrão é essencial que tenha estudado o padrão Factory Method (Roteiro 07). Além disso, vamos utilizar o mesmo cenário com algumas modificações para facilitar o entendimento.

**Padrão Factory Method** – É um padrão criacional, que fornece uma interface para criar **famílias de objetos** relacionados ou dependentes sem especificar suas classes concretas.

Relembrando o diagrama formal do Padrão Factory Method

Diagrama

O conteúdo gerado por IA pode estar incorreto.

O principal recurso de OO utilizado no Factory Method é a herança, onde a implementação do método createProduct() determina a criação do objeto produto, fazendo o papel de fábrica de objetos.

Abaixo temos o **diagrama formal do Abstract Factory**

Diagrama

O conteúdo gerado por IA pode estar incorreto.

Observe uma outra representação do Abstract Factory que talvez ajude no entendimento da sua estrutura.

A lógica do Abstract Factory se baseia em um agrupamento de classes (ou Família), que por meio de uma interface (ou classe abstrata), organizam toda a lógica necessária para criação dos seus objetos, fazendo também o papel de fábrica de objetos.

Veja que a relação entre cliente e AbstractFactory, e até mesmo entre as FactoryA e FactoryB, acontece por meio de **composição**. Ou seja, as fabricas serão passadas como parâmetro (isso ficará mais claro com o código). Já no Factory Method essa relação acontece por meio de **herança**.

**Diagrama

O conteúdo gerado por IA pode estar incorreto.**

**FAMÍLIA B**

**FAMÍLIA A**

**Relembrando o cenário onde trabalhamos com o Factory Method :**

Trabalhamos em um módulo de cobranças, onde precisávamos melhorar a rotina de geração dos boletos de cobrança. Inicialmente trabalhamos com regras bem específicas do banco Caixa conforme segue abaixo.

Boletos com vencimento para :

* 10 Dias : Juros de 2% : Desconto de 10%; Multa de 5%
* 30 Dias : Juros de 5% : Desconto de 5%; Multa de 10%
* 60 Dias : Juros de 10% : Desconto de 0%; Multa de 20%

Depois precisamos evoluir o projeto acrescentando o Banco do Brasil que também tinha as suas regras.

Boletos com vencimento para :

* 10 Dias : Juros de 3% : Desconto de 5%; Multa de 2%
* 30 Dias : Juros de 5% : Desconto de 2%; Multa de 5%
* 60 Dias : Juros de 10% : Desconto de 0%; Multa de 15%

**Para este cenário**, vamos simplificar um pouco as regras dos boletos, com o objetivo facilitar o entendimento do Abstract Factory. Neste caso vamos considerar que não existe uma variedade tão grande de objetos boletos a serem gerados (3 boletos possíveis para cada Banco, conforme o vencimento). Vamos continuar com 2 Bancos (Caixa e Banco do Brasil), e cada um dele irá gerar apenas um objeto boleto, desconsiderando as variedades conforme o vencimento.

Então teremos :

* Banco Caixa : Juros de 2% : Desconto de 10%; Multa de 5%
* Banco do Brasil : Juros de 3% : Desconto de 5%; Multa de 2%

**Pacote : roteiro9.parte1**

1 – Dê sequência ao mesmo projeto no NetBeans chamado **PADROESroteiros**

2 – Como foi explicado acima, tanto o cenário quanto a ideia principal do Abstract Factory, precisamos agora redefinir a modelagem que inicialmente foi feita para o Factory Method, para que agora se comporte como um Abstract Factory.

O primeiro passo é definir quem serão as classes necessárias que irão compor a Família de Objetos para o nosso produto final. Entenda que o nosso produto final deve ser o Boleto, e por isso devemos criar a modelagem da família de classes que irá concentrar as regras de negócio para criação deste Boleto.

Diagrama

O conteúdo gerado por IA pode estar incorreto.

Sendo assim, crie o **pacote roteiro9.parte1** e dentro deles as interfaces abaixo : **Juros**, **Desconto** e **Multa**

package roteiro9.parte1;

public interface Juros {

public double getJuros();

}

package roteiro9.parte1;

public interface Desconto {

public double getDesconto();

}

package roteiro9.parte1;

public interface Multa {

public double getMulta();

}

E para cada família implemente a classe correspondentes, conforme as regras de negócio descritas no cenário . Abaixo temos a implementação da família Banco Caixa.

package roteiro9.parte1;

public class CaixaJuros implements Juros{

@Override

public double getJuros() {

return 0.02;

}

}

package roteiro9.parte1;

public class CaixaDesconto implements Desconto{

@Override

public double getDesconto() {

return 0.1;

}

}

package roteiro9.parte1;

public class CaixaMulta implements Multa{

@Override

public double getMulta() {

return 0.05;

}

}

Implemente agora a família Banco do Brasil, conforme as respectivas regras de negócio.

3 – Devemos agora criar as classes que farão o papel da fábrica propriamente dito.

Diagrama

O conteúdo gerado por IA pode estar incorreto.

A interface CalculosFactory fará o papel do AbstractFactory

Perceba que nesta modelagem a fábrica será responsável por toda a inteligência de cálculo, chamando os métodos correspondentes de Juros, Desconto, Multa. Sendo que cada fábrica acionará somente as classes da família que lhe diz respeito.

* CaixaCalculosFactory : acionará somente as classes da família Banco Caixa
* BBCalculosFactory : acionará somente as classes da família Banco do Brasil

Implemente a interface **CalculosFactory** conforme o código abaixo

package roteiro9.parte1;

public interface CalculosFactory {

public Juros criarJuros();

public Desconto criarDesconto();

public Multa criarMulta();

}

Abaixo temos a classe concreta **CaixaCalculosFatory** que implementa CalculosFatory.

Faça o mesmo para **BBCalculosFactory** Atenção ao instanciar a família de classes corretas.

package roteiro9.parte1;

public class CaixaCalculosFactory implements CalculosFactory{

@Override

public Juros criarJuros() {

return new CaixaJuros();

}

@Override

public Desconto criarDesconto() {

return new CaixaDesconto();

}

@Override

public Multa criarMulta() {

return new CaixaMulta();

}

}

Observe que os métodos existentes na classe concreta CaixaCalculosFactory são responsáveis por criar todos os objetos necessários (Juros, Desconto, Multa) para a geração do produto final (Boleto).

4 – Copie as classes Banco e Boleto do roteiro7.parte3. Com essas classes dentro do roteiro9.parte1 precisaremos fazer algumas refatorações.

A primeira refatoração é que as classes Banco e Boleto agora devem ser concretas, pois nessa modelagem temos uma única entidade de Banco e de Boleto. A classe Banco por sua vez, irá gerar o boleto correto (Caixa ou banco do Brasil), utilizando a fábrica.

Na classe Boleto, os atributos juros, desconto e multa devem ser objetos e não simplesmente double.

O construtor do boleto passa a receber uma “factory” como parâmetro.

package roteiro9.parte1;

public ~~abstract~~ class Boleto {

protected double valor;

protected ~~double~~ Juros juros;

protected ~~double~~ Desconto desconto;

protected ~~double~~ Multa multa;

public Boleto( double valor, CalculosFactory factory) {

this.valor = valor;

this.juros = factory.criarJuros();

this.desconto = factory.criarDesconto();

this.multa = factory.criarMulta();

}

public double calcJuros(){

return this.valor \* this.juros.getJuros();

}

public double calcDesconto(){

return this.valor \* this.desconto.getDesconto();

}

public double calcMulta(){

return this.valor \* this.multa.getDesconto();

}

}

Veja que a variável factory é do tipo CalculosFactory. Ou seja, uma interface.

* Se for passado como parâmetro uma instância da fábrica Banco Caixa, o boleto será criado com todos os objetos da família Banco Caixa.
* Se for passado como parâmetro uma instância da fábrica Banco do Brasil, o boleto será criado com todos os objetos da família Banco do Brasil.

Veja que aí está uma diferença marcante entre o Abstract Factory e o Factory Method. Neste caso estamos usando a **composição** passando como parâmetro a fator e **não** **herança**.

Como simplificamos o cenário, não precisamos mais do vencimento como passagem de parâmetro no método gerarBoleto().

Por outro lado, precisamos adicionar no método gerarBoleto() o parâmetro factory do tipo CalculosFactory. Novamente o uso de **composição** e **inversão de dependência** (mesma estratégia utilizada na classe Boleto)

O método criarBoleto() também precisa ser retirado. Lembre-se, se nesta modelagem não usamos herança, não existe mais uma subclasse para implementar este método como foi feito no Factory Method.

package roteiro9.parte1;

public ~~abstract~~ class Banco {

public Boleto gerarBoleto(~~int vencimento~~, double valor, CalculosFactory factory){

Boleto boleto = new Boleto(~~vencimento~~, valor, factory);

System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

System.out.println("Boleto gerado com sucesso. Valor = "+ valor);

System.out.println("Valor Juros = "+ boleto.calcJuros());

System.out.println("Valor Desconto = "+ boleto.calcDesconto());

System.out.println("Valor Multa = "+ boleto.calcMulta());

System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

return boleto;

}

~~public abstract Boleto criarBoleto(int vencimento, double valor);~~

}

4 – Faça os testes na classe **TesteBoleto**

Observe abaixo os efeitos da refatoração que fizemos.

* Banco agora é um objeto genérico. Não existe mais uma entidade individual para o Banco Caixa e o Banco do Brasil.
* Também não existe diferentes entidades de boletos para vencimentos diferentes. Decidimos simplificar um pouco o cenário para melhor entendimento do Abstract Factory. Ainda assim, mantivemos os comentários no código para comparação entre as duas modelagens.
* Quem irá definir o Boleto correto a ser gerado é a fábrica. Seja o boleto da Caixa ou do Banco do Brasil. Para isso, devemos criar a fábrica adequada e passar como parâmetro para que seja gerado o boleto correto.
  + CaixaCalculosFactory passado como parâmetro irá gerar os boletos da Caixa
  + BBCalculosFactory passado como parâmetro irá gerar os boletos do Banco do Brasil.
* Observe que o cliente ficou totalmente desacoplado das classes que envolvem as regras de negócio para geração do boleto. O cliente deve apenas passar como parâmetro a fábrica que ele deseja e os cálculos serão feitos da forma correta.

package roteiro9.parte1;

public class TesteBoleto {

public static void main(String[] args) {

Banco banco = new Banco();

System.out.println(" \n --- BANCO CAIXA --- \n ");

CalculosFactory caixaCalculosFatory = new CaixaCalculosFactory();

try {

banco.gerarBoleto(100, caixaCalculosFatory);

//bancocaixa.gerarBoleto(30, 100);

//bancocaixa.gerarBoleto(60, 100);

//bancocaixa.gerarBoleto(90, 100);

} catch (Exception e) {

System.out.println(e.getMessage());

}

System.out.println(" \n --- BANCO DO BRASIL --- \n ");

CalculosFactory bbCalculosFatory = new BBCalculosFactory();

try {

banco.gerarBoleto(100, bbCalculosFatory);

//bancobrasil.gerarBoleto(30, 100);

//bancobrasil.gerarBoleto(60, 100);

//bancobrasil.gerarBoleto(90, 100);

} catch (Exception e) {

System.out.println(e.getMessage());

}

}

}

OBS.: as famílias de objetos Juros, Desconto e Multa são bem simples e talvez não fosse necessário criá-las desta forma. Utilizamos desta modelagem apenas para fins didáticos. Suponha que essas famílias poderiam ter métodos e cálculos maiores e complexos, e neste caso o Abstract Factory ajudaria bastante nesta abstração, encapsulando todos os objetos de cálculo.

5 – Utilize uma ferramenta de software qualquer para geração do diagrama de classes para esta etapa do projeto (Sugestão : Astah Community). Obs.: Adicione aqui o diagrama para que seja disponibilizado no teams

Obs: Outra possibilidade para geração dos diagramas é o StarUML : <https://staruml.io/> . Para fazer a engenharia reversa do código siga as orientações conforme o vídeo a seguir : <https://www.youtube.com/watch?v=z6DDuZQArro>

7 – Vamos analisar uma possível evolução deste cenário. Não é necessário implementar ! Com esta modelagem do Abstract Factory o que deveríamos fazer se tivéssemos que gerar um boleto para o Banco Santander ?

**Pacote : roteiro9.parte2**

1 – No mesmo projeto crie o pacote **roteiro9.parte2**

2 – Copie todas as classes criadas na parte1 para o novo pacote.

3 – Voltando ao cenário original. Utilizando esta modelagem com o Abtract Factory, como você faria as refatorações necessárias para contemplar o cenário original onde temos regras específicas não só para cada banco, mas por dias de vencimento ? Faça os ajustes que achar necessário, mas procure respeitar o padrão Abstract Factory

Boletos da Caixa com vencimento para :

* 10 Dias : Juros de 2% : Desconto de 10%; Multa de 5%
* 30 Dias : Juros de 5% : Desconto de 5%; Multa de 10%
* 60 Dias : Juros de 10% : Desconto de 0%; Multa de 20%

Boletos do Banco do Brasil com vencimento para :

* 10 Dias : Juros de 3% : Desconto de 5%; Multa de 2%
* 30 Dias : Juros de 5% : Desconto de 2%; Multa de 5%
* 60 Dias : Juros de 10% : Desconto de 0%; Multa de 15%