**Roteiro 07**

Neste roteiro vamos trabalhar com o padrão Factory Method.

**Padrão Factory Method** – É um padrão criacional, que define uma interface para criar um objeto, mas permite que as subclasses possam decidir qual classe instanciar, possibilitando que uma classe seja capaz de prorrogar a instanciação de uma classe para subclasses.

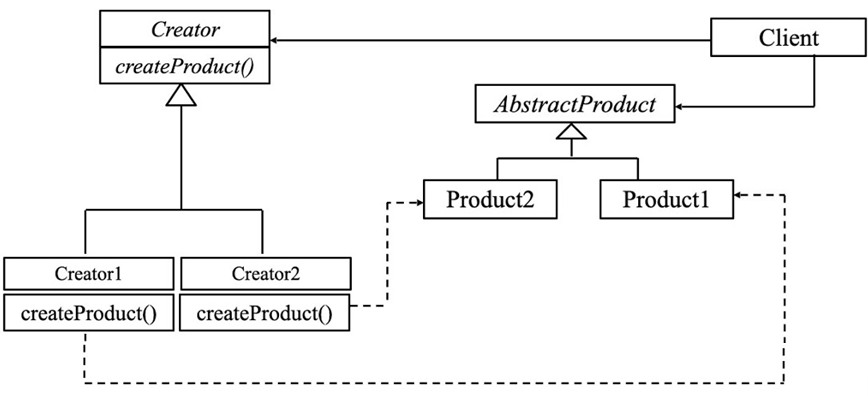
Ou seja, em vez de instanciar diretamente objetos, você usa um método conhecido como “fábrica” que delega a criação destes objetos para as subclasses, permitindo uma flexibilidade maior na escolha do objeto a ser criado.

Abaixo temos o diagrama formal do Padrão Factory Method

Diagrama

O conteúdo gerado por IA pode estar incorreto.

A seguir temos também uma representação do Factory Method de forma um pouco mais detalhada.



Vamos evoluir o cenário a seguir de formar que facilite o entendimento do padrão Factory Method

**Cenário :**

Precisamos trabalhar em apenas um dos módulos de um grande sistema legado. Trata-se do módulo de cobranças, e mais especificamente nas funcionalidades que geram os boletos de cobrança. Inicialmente vamos gerar boletos de um banco específico com regras bem específicas.

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%

**Pacote : roteiro7.parte1**

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

2 – Como foi explicado no cenário inicial, temos uma modelagem inicial que trata de módulo de cobrança dentro de um sistema legado, onde são gerados os coletos de cobrança. Por isso, crie as classes abaixo no pacote **roteiro7.parte1**.

package roteiro7.parte1;

public abstract class Boleto {

protected double valor;

protected double juros;

protected double desconto;

protected double multa;

public Boleto( double valor) {

this.valor = valor;

}

public double calcJuros(){

return this.valor \* this.juros;

}

public double calcDesconto(){

return this.valor \* this.desconto;

}

public double calcMulta(){

return this.valor \* this.multa;

}

}

package roteiro7.parte1;

public class BoletoCaixa10Dias extends Boleto{

public BoletoCaixa10Dias(double valor) {

super(valor);

this.juros = 0.02;

this.desconto = 0.1;

this.multa = 0.05;

}

}

package roteiro7.parte1;

public class BoletoCaixa30Dias extends Boleto{

public BoletoCaixa30Dias(double valor) {

super(valor);

this.juros = 0.05;

this.desconto = 0.05;

this.multa = 0.1;

}

}

package roteiro7.parte1;

public class BoletoCaixa60Dias extends Boleto{

public BoletoCaixa60Dias(double valor) {

super(valor);

this.juros = 0.1;

this.desconto = 0;

this.multa = 0.2;

}

}

3 – Por enquanto o nosso módulo de cobrança trabalha apenas com um banco (Caixa Econômica). Por isso crie a classe **BancoCaixa**. E o método **gerarBoleto** é responsável criar o boleto de acordo com o vencimento.

package roteiro7.parte1;

public class BancoCaixa {

public BancoCaixa() {

}

public Boleto gerarBoleto(int vencimento, double valor){

Boleto boleto = null;

switch (vencimento) {

case 10: boleto = new BoletoCaixa10Dias(valor);

break;

case 30: boleto = new BoletoCaixa30Dias(valor);

break;

case 60: boleto = new BoletoCaixa60Dias(valor);

break;

default:

throw new UnsupportedOperationException("ERRO: Vencimento indisponível");

}

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;

}

}

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

package roteiro7.parte1;

public class TesteBoleto {

public static void main(String[] args) {

BancoCaixa banco = new BancoCaixa();

try {

banco.gerarBoleto(10, 100);

banco.gerarBoleto(30, 100);

banco.gerarBoleto(60, 100);

banco.gerarBoleto(90, 100);

} catch (Exception e) {

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

}

}

}

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>

6 – Consegue identificar que existe um forte acoplamento entre as classes BancoCaixa e os diferentes tipos de Boletos ? Qual impacto caso surgisse a necessidade de incluir novos tipos de boletos ou trabalhar com outros bancos ?

**Pacote : roteiro7.parte2**

1 – No mesmo projeto crie o pacote **roteiro7.parte2**

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

3 – Para tentar reduzir o forte acoplamento entre a classe BancoCaixa e os Boletos, vamos criar a classe **BoletoSimpleFactory** conforme segue abaixo.

Estamos aplicando neste momento a ideia de uma modelagem **Simple Factory**. Esta modelagem é frequentemente utilizada na comunidade, mas não é considerada formalmente como um padrão de projeto. Assim como também não está entre os 23 padrões do GoF.

package roteiro7.parte2;

public class BoletoSimpleFactory {

public Boleto criarBoleto(int vencimento, double valor){

Boleto boleto = null;

switch (vencimento) {

case 10: boleto = new BoletoCaixa10Dias(valor);

break;

case 30: boleto = new BoletoCaixa30Dias(valor);

break;

case 60: boleto = new BoletoCaixa60Dias(valor);

break;

default:

throw new UnsupportedOperationException("ERRO: Vencimento indisponível");

}

return boleto;

}

}

4 – Observe que agora a classe BoletoSimpleFactory passa a ser responsável pela criação dos objetos do tipo Boleto e não mais a classe BancoCaixa. Por isso, precisamos refatorar a classe BancoCaixa.

BancoCaixa agora possui um atributo do tipo BoletoSimpleFactory e delega a criação do Boleto a classe BoletoSimpleFactory.

package roteiro7.parte2;

public class BancoCaixa {

private BoletoSimpleFactory boletoSimpleFactory;

public BancoCaixa(BoletoSimpleFactory boletoSimpleFactory) {

this.boletoSimpleFactory = boletoSimpleFactory;

}

public Boleto gerarBoleto(int vencimento, double valor){

~~Boleto boleto = null;~~

~~switch (vencimento) {~~

~~case 10: boleto = new BoletoCaixa10Dias(valor);~~

~~break;~~

~~case 30: boleto = new BoletoCaixa30Dias(valor);~~

~~break;~~

~~case 60: boleto = new BoletoCaixa60Dias(valor);~~

~~break;~~

~~default:~~

~~throw new UnsupportedOperationException("ERRO: Vencimento indisponível");~~

~~}~~

Boleto boleto = this.boletoSimpleFactory.criarBoleto(vencimento, valor);

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;

}

}

5 – Analise a refatoração que acabou de ser feita.

* Basicamente mudamos de lugar o trecho de código com o switch da classe BancoCaixa para a classe BoletoSimpleFactory. Ou seja, não foi uma grande mudança. Só reposicionamos um trecho de código. Consegue ver alguma vantagem nessa refatoração ?
* No código antes da refatoração, o que aconteceria se tivéssemos que trabalhar com mais de um banco (Caixa Econômica, Banco do Brasil, Itaú e etc) ?

6 – 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>

**Pacote : roteiro7.parte3**

**Evoluindo o cenário :**

Precisamos agora que o nosso módulo de cobrança trabalhe com um outro banco. Devemos incluir o Banco do Brasil na nossa modelagem. Precisamos ter na modelagem 2 classes de banco para cobrança : BancoCaixa e BancoBrasil. Nossas classes neste contexto servem apenas para efeitos didáticos, e por isso são bem simples. Certamente poderíamos implementar outra solução, mas vamos explorar aqui a modelagem para o Factory Method.

Neste novo banco, assim como no anterior temos regras bem específicas.

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%

1 – No mesmo projeto crie o pacote **roteiro7.parte3**

2 – Copie todas as classes da família Boleto (**Boleto**, **BoletoCaixa10Dias**, **BoletoCaixa30Dias**, **BoletoCaixa60Dias**) da parte anterior para o novo pacote.

3 – Crie as novas classes como subclasses de Boleto : **BoletoBrasil10Dias**, **BoletoBrasil30Dias**, **BoletoBrasil60Dias.** Essas classes representam os boletos específicos do Banco do Brasil e são semelhantes aos boletos da Caixa Econômica. Devem apenas considerar as regras do Banco do Brasil informadas acima no cenário.

4 – Crie agora a classe abstrata **Banco**. A partir desta classe criaremos as subclasses concretas **BancoCaixa** e **BancoBrasil**. Observe que nesta classe temos o método também abstrato criarBoleto que será implementado nas classes concretas.

package roteiro7.parte3;

public abstract class Banco {

public Boleto gerarBoleto(int vencimento, double valor){

Boleto boleto = this.criarBoleto(vencimento, valor);

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);

}

5 – Copie a classe **BancoCaixa** da parte anterior e faça as refatorações abaixo. Observe que o método criarBoleto herdado da classe **Banco** está sendo implementado na classe concreta **BancoCaixa**.

Observe que a partir de agora não faz mais sentido utilizar a classe BoletoSimpleFactory.

package roteiro7.parte3;

public class BancoCaixa extends Banco{

~~private BoletoSimpleFactory boletoSimpleFactory;~~

~~public BancoCaixa(BoletoSimpleFactory boletoSimpleFactory) {~~

~~this.boletoSimpleFactory = boletoSimpleFactory;~~

~~}~~

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

~~Boleto boleto = this.boletoSimpleFactory.criarBoleto(vencimento, valor);~~

~~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;~~

~~}~~

@Override

public Boleto criarBoleto(int vencimento, double valor) {

switch (vencimento) {

case 10: return new BoletoCaixa10Dias(valor);

case 30: return new BoletoCaixa30Dias(valor);

case 60: return new BoletoCaixa60Dias(valor);

default:

throw new UnsupportedOperationException("ERRO: Vencimento indisponível");

}

}

}

6 – Crie a classe **BancoBrasil** também como subclasse de Banco e faça as implementações semelhantes ao que foi feito na classe BancoCaixa. **ATENÇÃO** : O Banco do Brasil tem boletos específicos, de forma que cada banco deve instanciar os boletos específicos do seu banco.

7 – 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>

8 – Agora com o diagrama em mãos procure relacionar as classes deste cenário com o diagrama formal do Factory Method, e analise o que seria necessário se fossemos incluir um novo banco neste cenário.