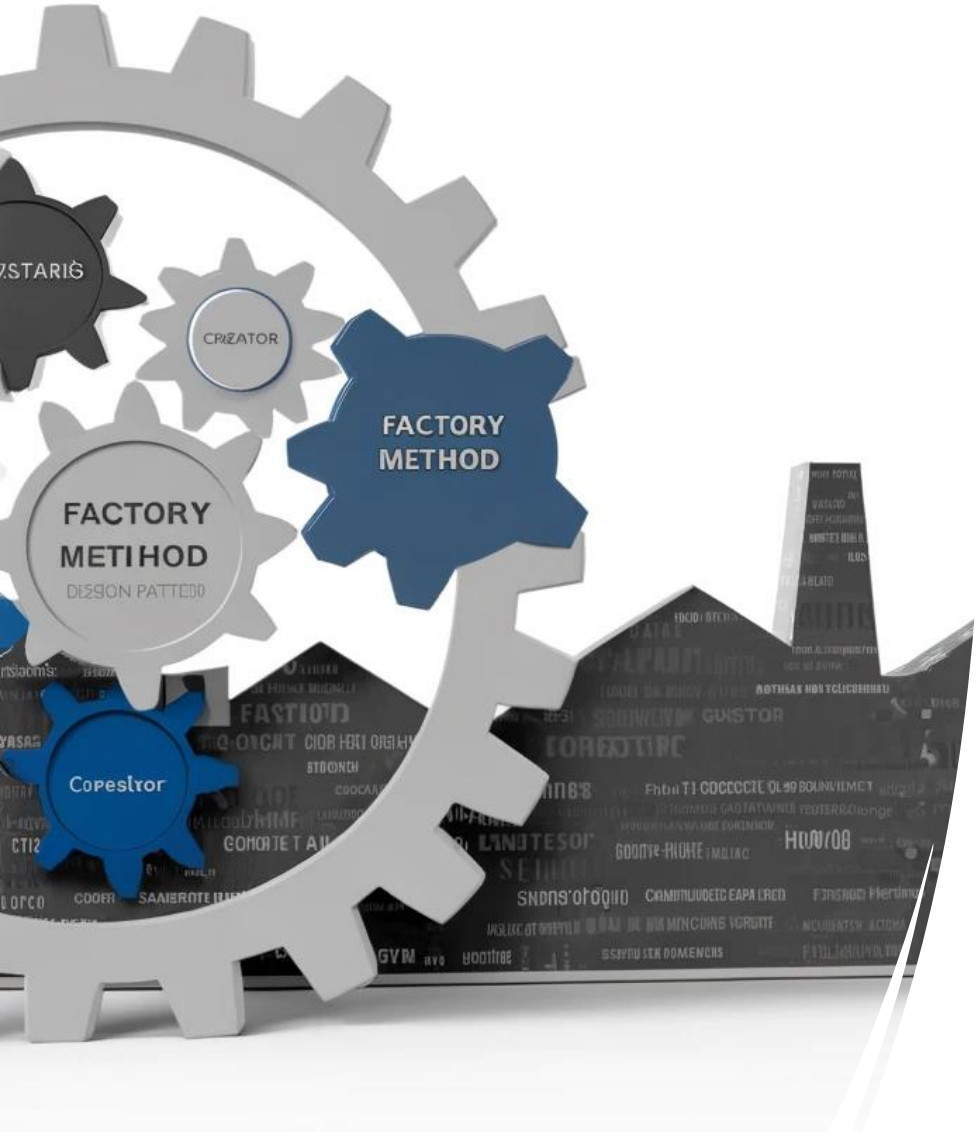


Factory Method

Padrões de Projeto Criacional I

Prof. Me Jefferson Passerini

O padrão ***Factory Method*** define uma interface para criar um objeto, mas permite que a subclasses possam decidir qual classe instânciar, possibilitando que uma classe seja capaz de prorrogar a instanciação de uma classe para subclasses.

Aplicabilidade (Quando utilizar?)

- Quando uma classe não sabe antecipar qual o tipo de objeto deve criar, ou seja, entre várias classes possíveis, não é possível prever qual delas deve ser utilizada.

- Quando se precisa que uma classe delegue para suas subclasses especificação dos objetos que instanciam.

- Quando classes delegam responsabilidade a uma dentre várias subclasses auxiliares, se deseja manter o conhecimento nelas e ainda saber qual subclasse foi utilizada em determinado contexto.

Componentes

- **Produto**: define a interface dos objetos que serão criados pelo método de factoryMethod() dos CriadoresConcretos.

- **ProdutoConcreto**: implementa a interface Produto. Isso permite que classes que usam os Produtos possam esperar a interface Produto ao invés de um ProdutoConcreto.

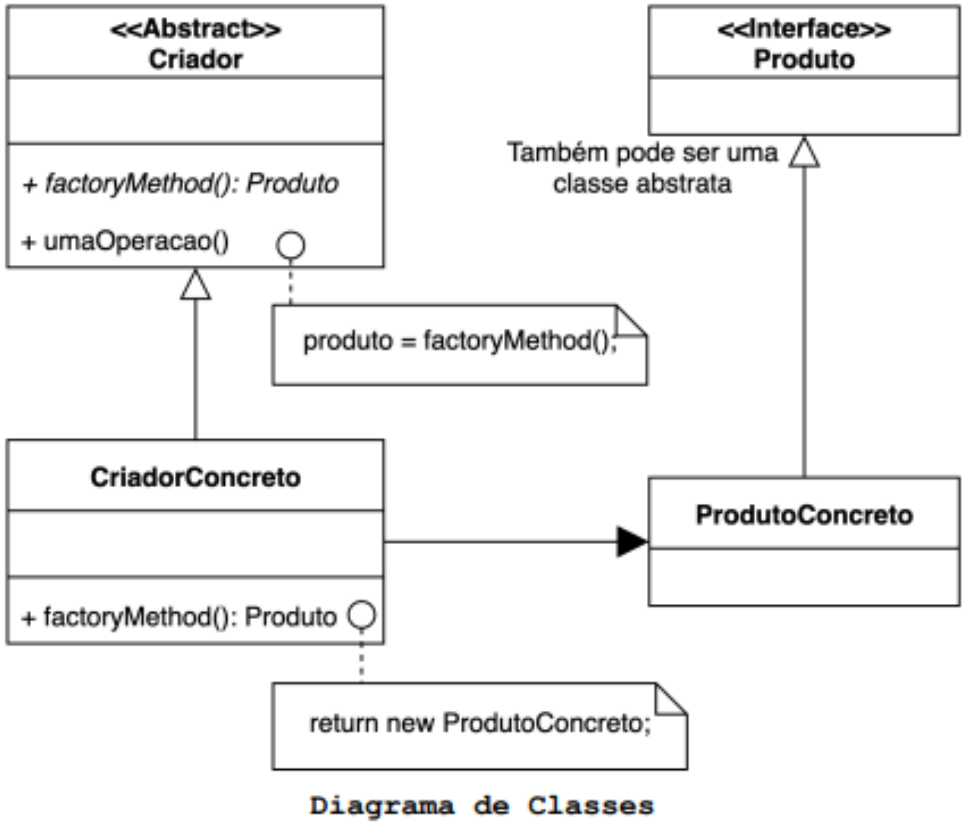


Diagrama de Classes

Componentes

- **Criador**: declara o método fabrica (factory method) o qual retorna um objeto ProdutoConcreto.
  - Também pode definir uma implementação padrão do factory method, para o caso de uma subclasse o omitir.
  - Tal implementação também precisa retornar um ProdutoConcreto.
  - O criador também é uma classe que utiliza o ProdutoConcreto retornado pelo método factory method.

- **CriadorConcreto**: implementa ou sobrescreve o factoryMethod(), para retornar uma instância de um ProdutoConcreto.

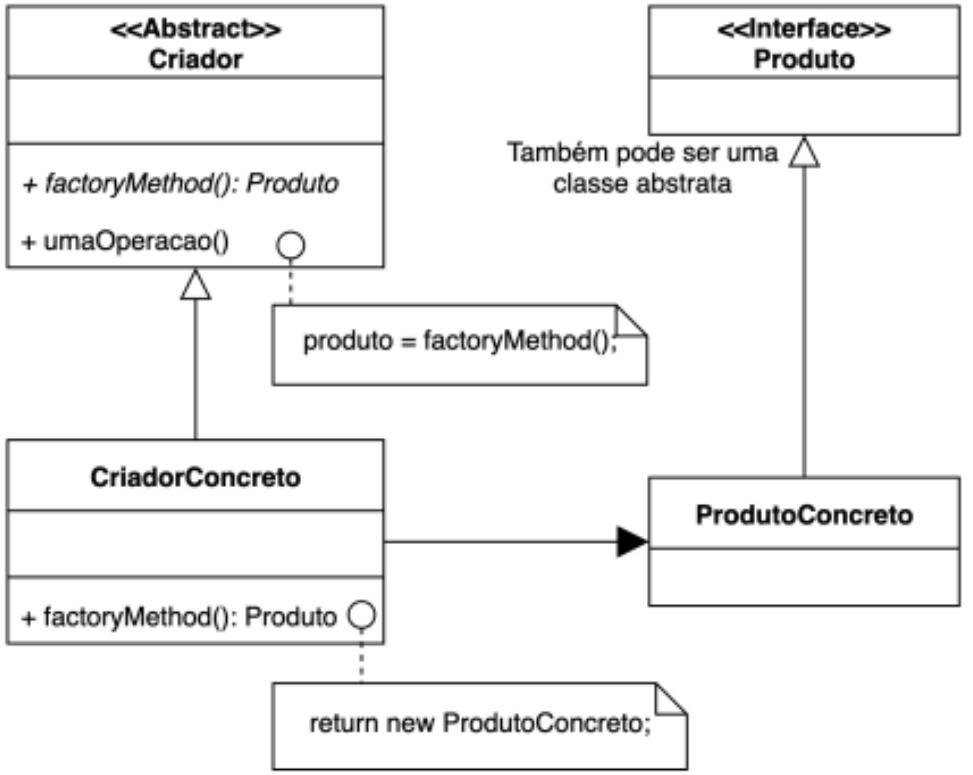


Diagrama de Classes

Motivação (Problema)

- Ao criar sistemas orientados a objetos não há como deixar de instanciar classes concretas, não existe nenhum problema nisso, porém como e onde tais objetos são instanciados pode criar um forte acoplamento entre classes de um sistema.

- Instanciar um objeto pode requerer processos complexos para que ele seja construído corretamente. Também pode causar uma significativa duplicação de código em diferentes classes onde ele é utilizado.

- Para facilitar o entendimento imagine um cenário onde temos um módulo de emissão de boletos em um software de cobranças. Tal módulo emite boletos com 3 intervalos possíveis de vencimentos e diferentes juros, descontos e multas:

Dias p/ vencimento	Juros	Desconto	Multa
10	2%	10%	5%
30	5%	5%	10%
60	10%	0%	20%

Diferentes vencimentos de boletos e seus respectivos cálculos baseados em seu valor.

Motivação (Problema)

- Atualmente o módulo de cobranças emite boletos de apenas um banco, no caso o **BancoCaixa**. O módulo se encontra estruturado conforme o diagrama de classes a seguir:

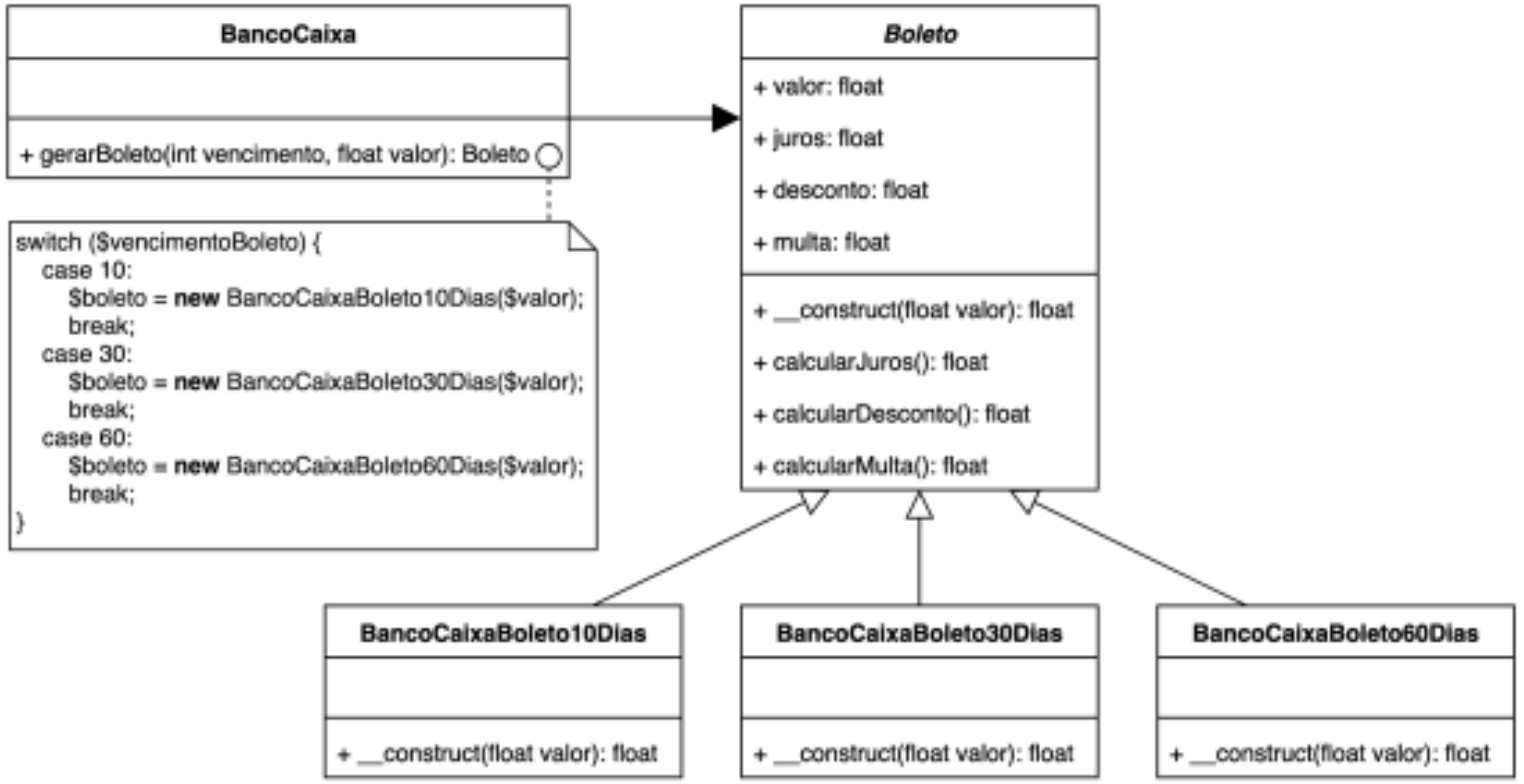


Diagrama de classes da estrutura inicial do módulo de cobranças.

Motivação (Problema)

- A classe **BancoCaixa** é a responsável por instanciar objetos do tipo **Boleto10Dias**, **Boleto30Dias** ou **Boleto60Dias** de acordo com o valor recebido no parâmetro vencimento do método **gerarBoleto**().

- Boleto é uma classe abstrata que contém os métodos responsáveis por calcular o Juros, Desconto e Multa do boleto. As subclasses de boleto apenas definem as porcentagens em seu construtor. Veja o código a seguir.

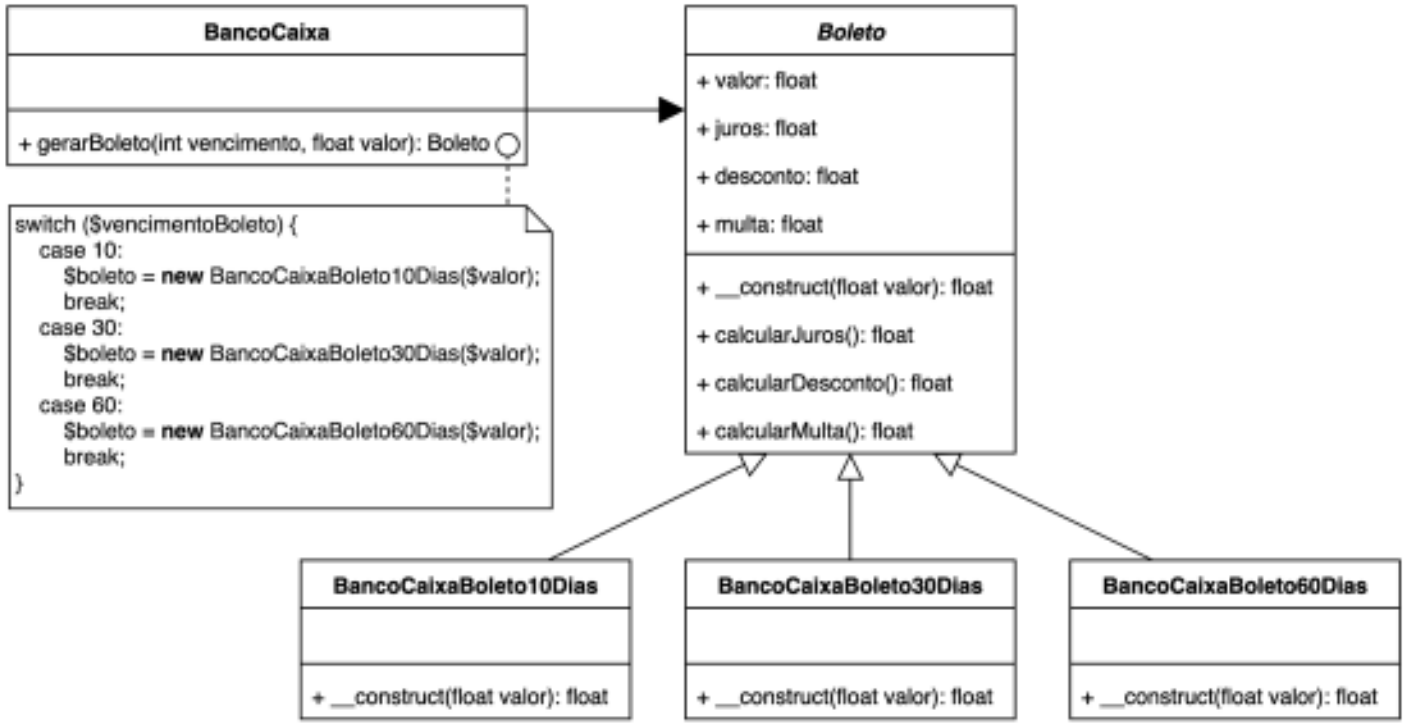


Diagrama de classes da estrutura inicial do módulo de cobranças.

Motivação (Problema)

- O problema desta abordagem é o forte acoplamento entre a classe **BancoCaixa** e as classes de boleto.
- Para instanciar os objetos concretos de boleto a classe **BancoCaixa** precisa conhecer as classes **Boleto10Dias**, **Boleto30Dias** e **Boleto60Dias**.
- Sempre que usamos a palavra reservada **new** estamos criando um objeto concreto, e por consequência estamos criando uma dependência entre classes.
- É ***impossível criar um sistema orientado a objetos sem criar objeto***, portanto **a palavra reservada new é fundamental*, ela precisa ser usada, porém onde usá-la pode fazer toda diferença.***

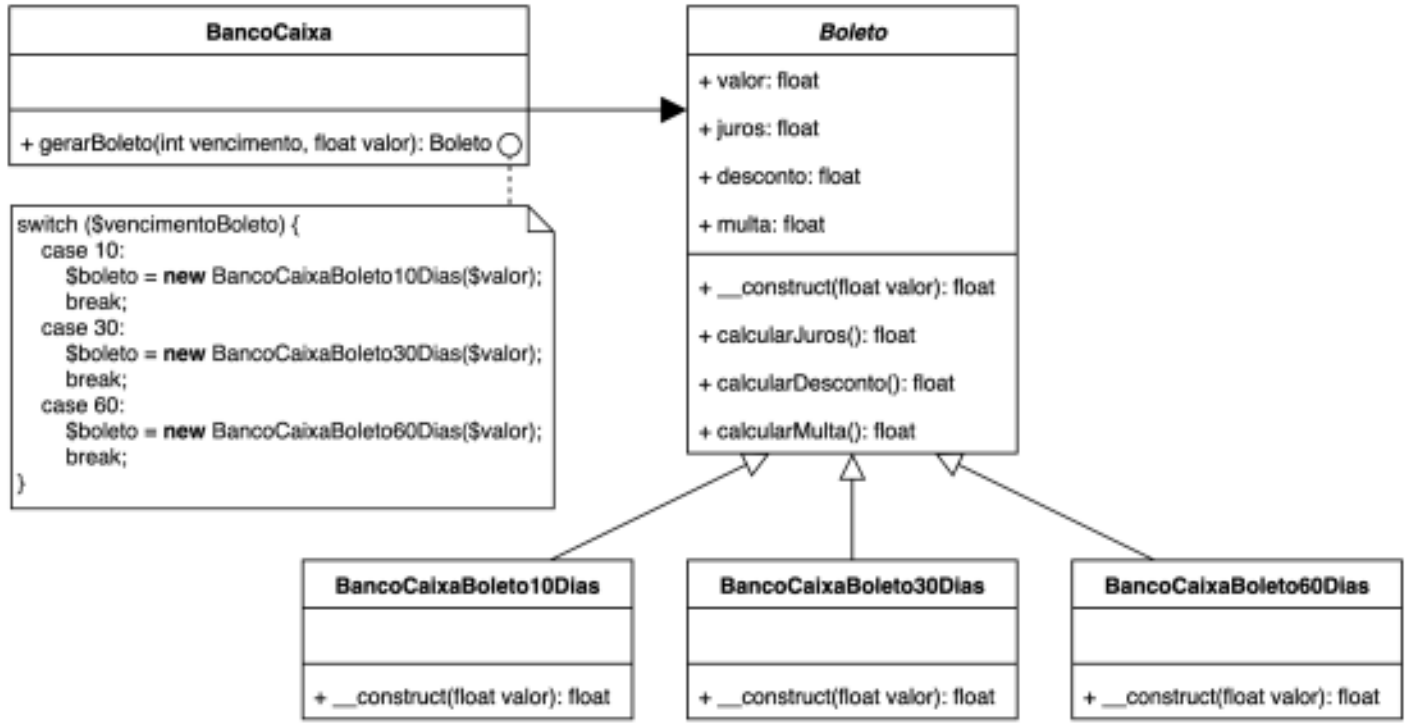
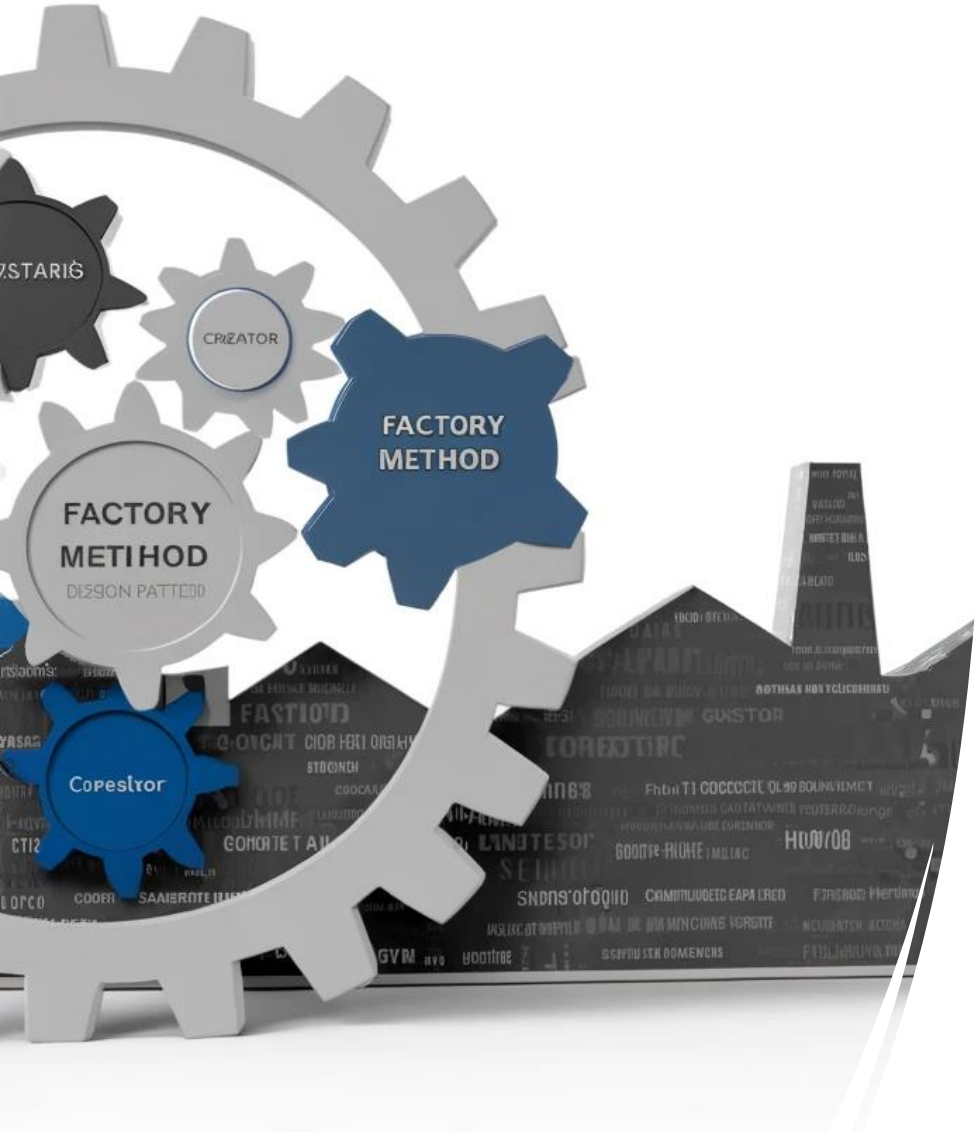


Diagrama de classes da estrutura inicial do módulo de cobranças.



Simple Factory

Padrões de Projeto Criacional I

Prof. Me Jefferson Passerini

Simple Factory

- Vamos apenas mudar o local onde instanciamos os objetos.
- Ao invés de criá-los na classe **BancoCaixa** vamos criá-los na classe **BoletoSimpleFactory**.
- Essa classe irá criar e retornar o boleto adequado, cabendo a classe **BancoCaixa** somente solicitar o objeto para classe **BoletoSimpleFactory**.
- As classes Boleto, **Boleto10Dias**, **Boleto30Dias** ou **Boleto60Dias** não mudam. Vejamos a criação da classe **BoletoSimpleFactory** e as mudanças na classe **BancoCaixa**. (C# e Java)

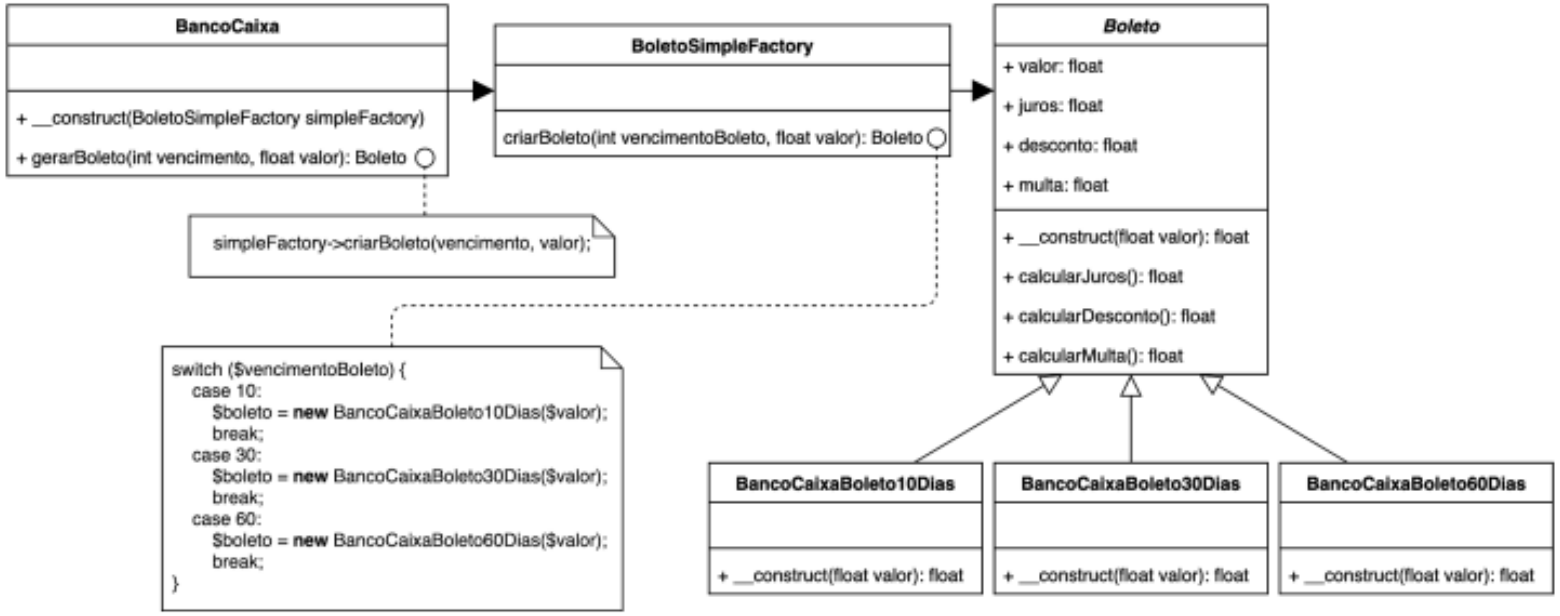
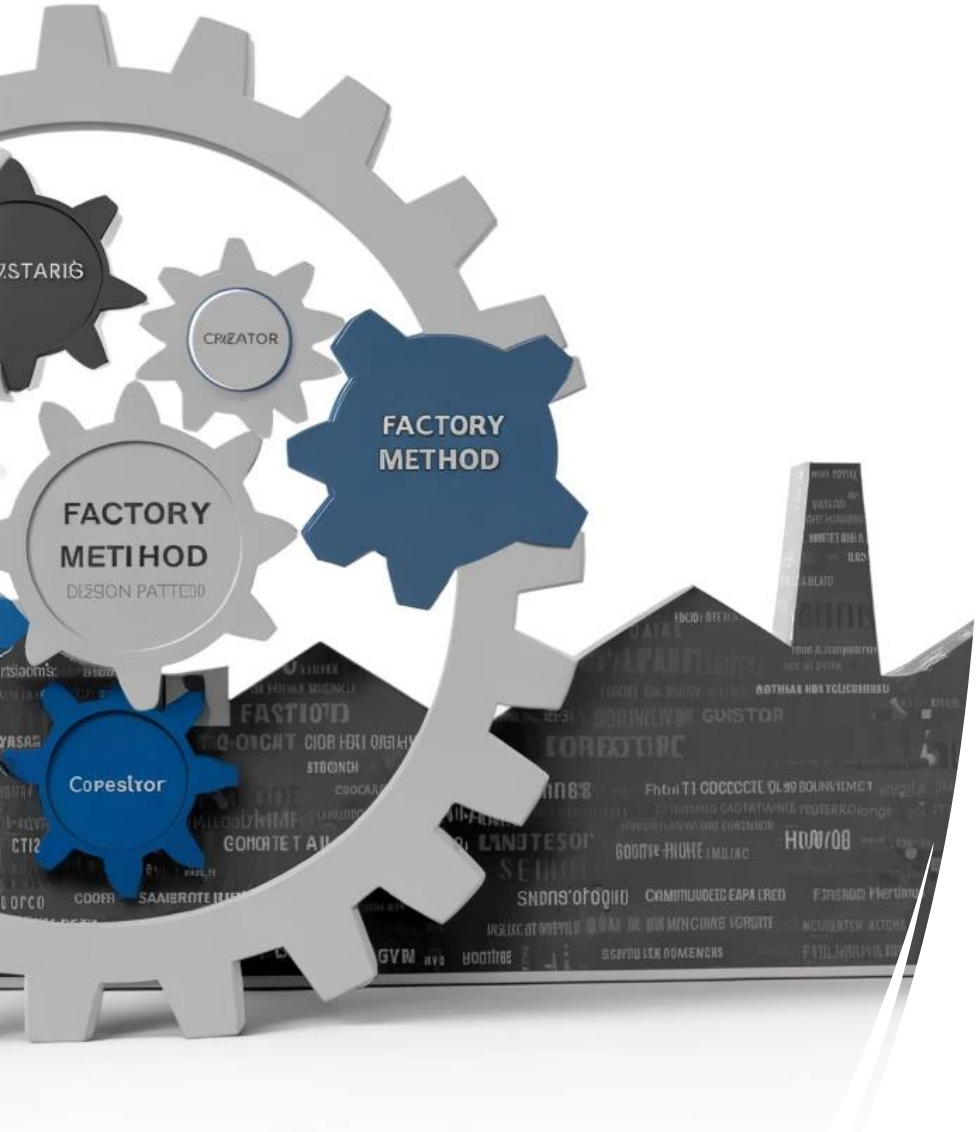


Diagrama de classes da estrutura do módulo de cobranças utilizando *Simple Factory*.



Simple Factory Java

Padrões de Projeto Criacional I

Prof. Me Jefferson Passerini

Simple Factory (Solução 1)

- Vamos apenas mudar o local onde instanciamos os objetos.

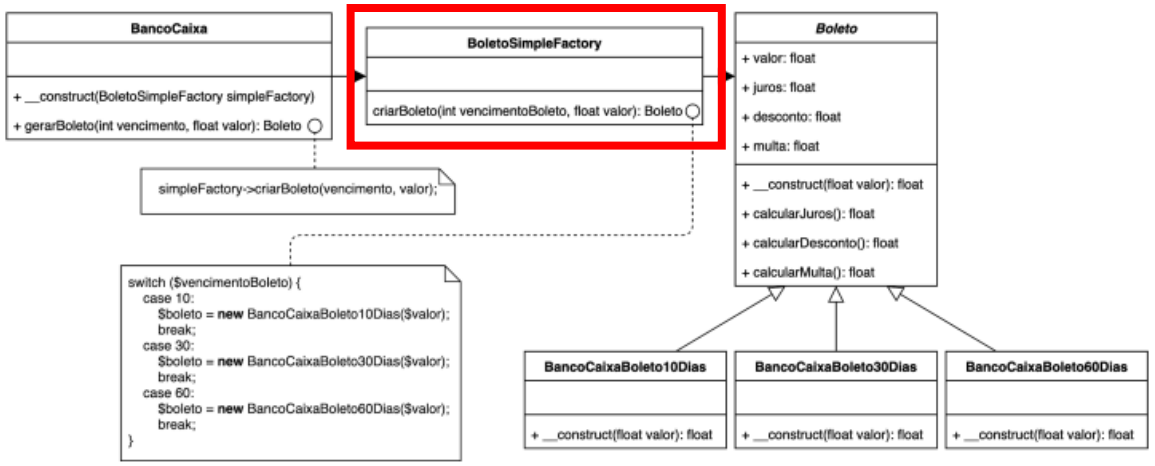
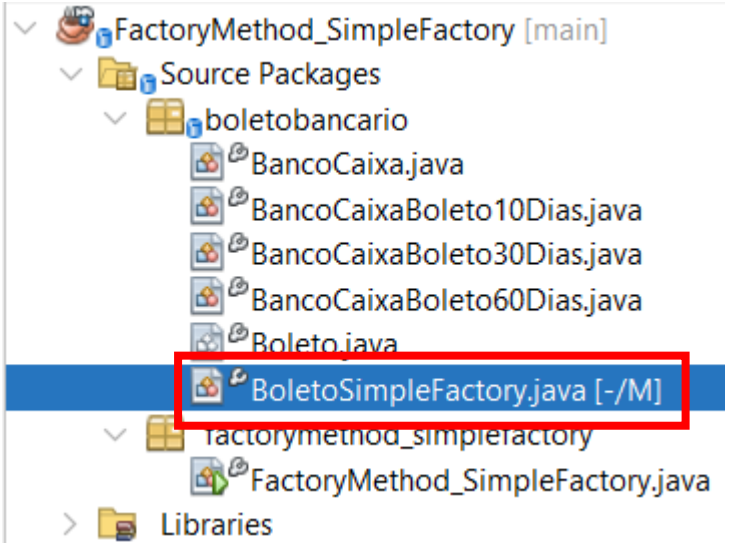


Diagrama de classes da estrutura do módulo de cobranças utilizando *Simple Factory*.




```
1  ...4 lines
5  package boletobancario;
6
7  /**...4 lines */
11 public class BoletoSimpleFactory {
12
13     public Boleto criarBoleto(int vencimento, double valor)
14             throws Exception{
15         Boleto boleto;
           switch (vencimento) {
17             case 10:
18                 boleto = new BancoCaixaBoleto10Dias(valor);
19                 break;
20             case 30:
21                 boleto = new BancoCaixaBoleto30Dias(valor);
22                 break;
23             case 60:
24                 boleto = new BancoCaixaBoleto60Dias(valor);
25                 break;
26             default:
27                 throw new Exception("Vencimento indisponível");
28         }
29         return boleto;
30     }
31 }
```

Simple Factory

- Vamos apenas mudar o local onde instanciamos os objetos.

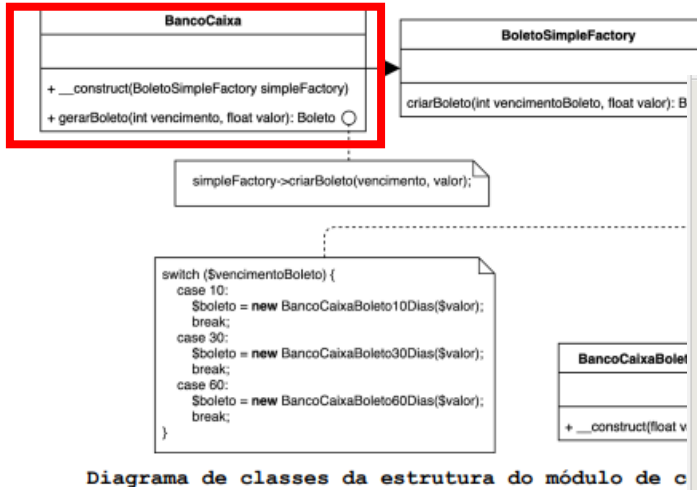
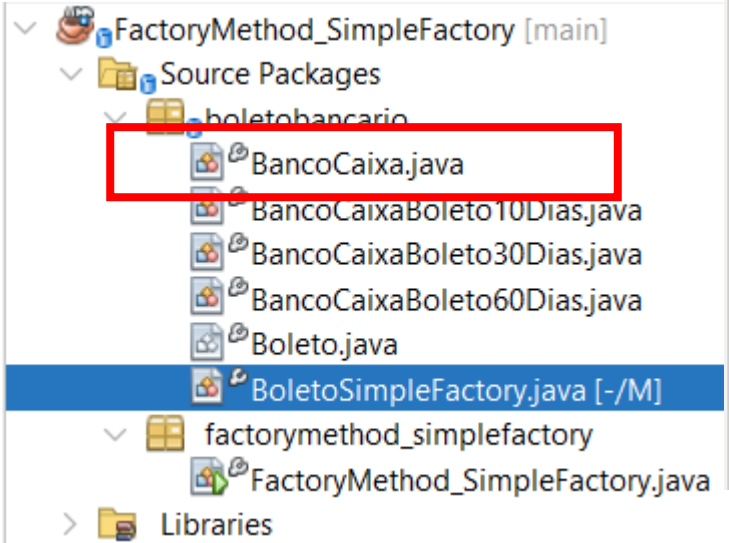


Diagrama de classes da estrutura do módulo de c




```java
package boletobancario;
/**...4 lines */
public class BancoCaixa {

    private BoletoSimpleFactory boletoSimpleFactory;

    public BancoCaixa(BoletoSimpleFactory boletoSimpleFactory) {
        this.boletoSimpleFactory = boletoSimpleFactory;
    }

    public void gerarBoleto(int vencimento, double valor) throws Exception{
        Boleto boleto = this.boletoSimpleFactory.criarBoleto(vencimento, valor);
        System.out.println("Boleto gerado com sucesso no valor de R$"+valor);
        System.out.println("Valor Juros R$"+boleto.calcularJuros());
        System.out.println("Valor Multa R$"+boleto.calcularMulta());
        System.out.println("Valor Desconto R$"+boleto.calcularDesconto());
        System.out.println("-----------------------------------------------");
    }
}
```

Simple Factory

- Vamos apenas mudar o local onde instanciamos os objetos.

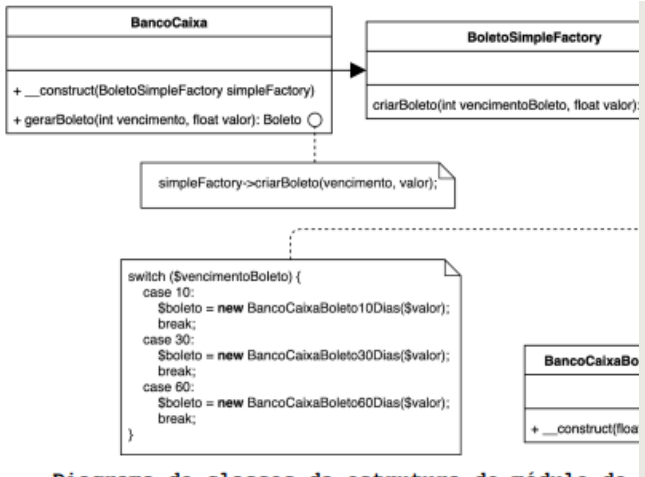
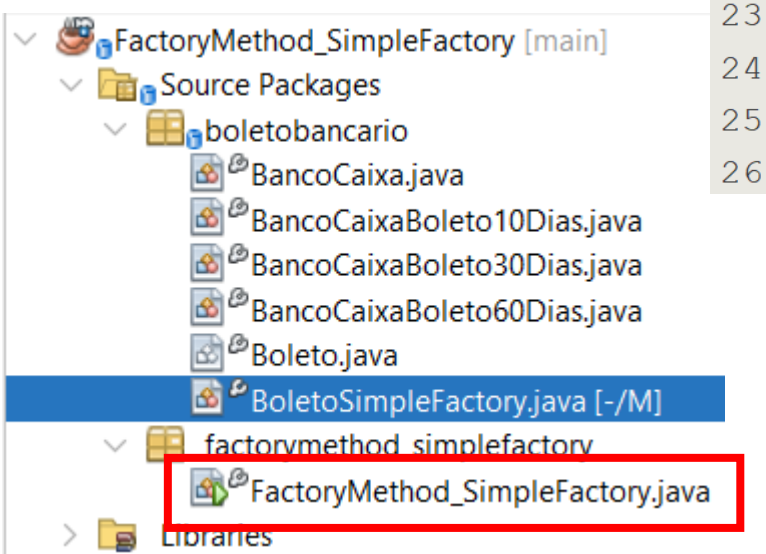
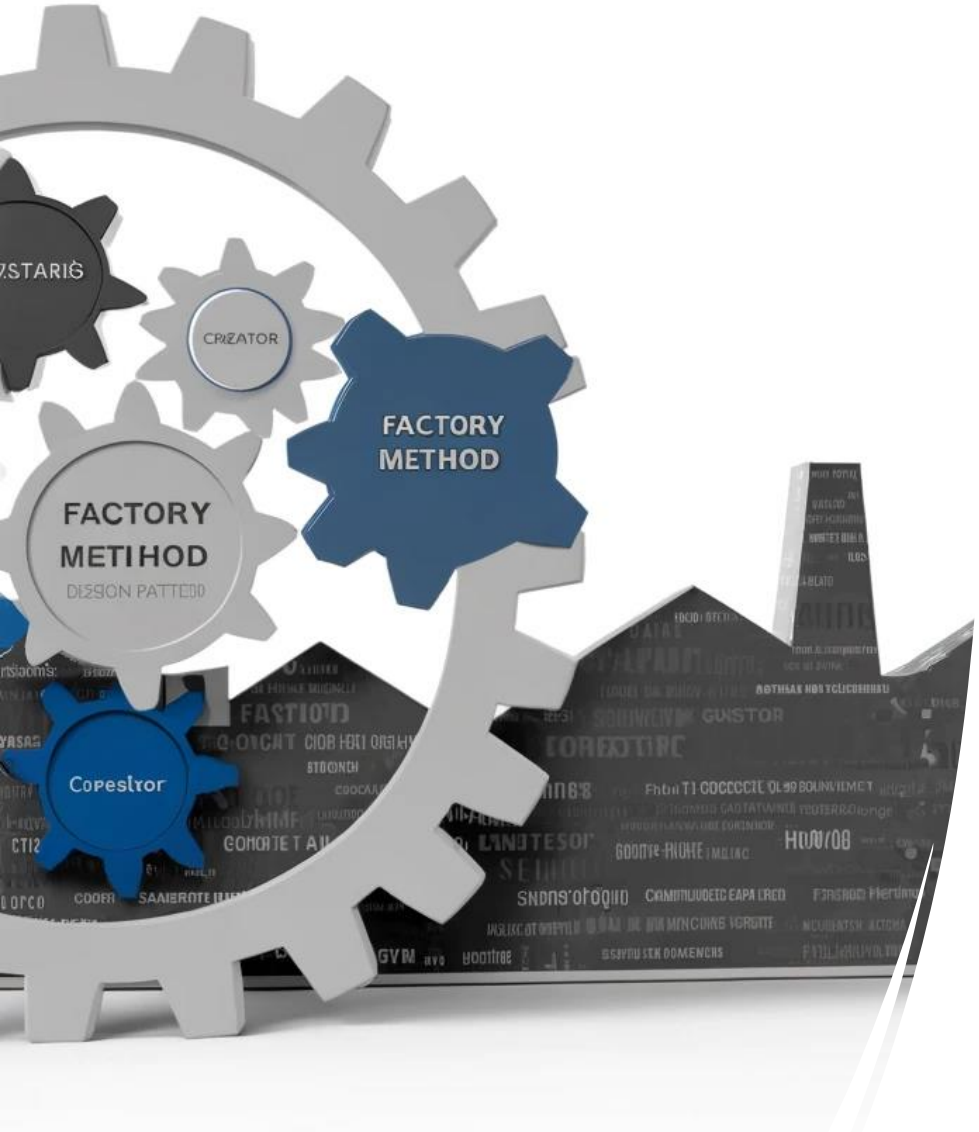


Diagrama de classes da estrutura do módulo de




```
14  public class FactoryMethod_SimpleFactory {
15
16      /**
17       * @param args the command line arguments
18       */
        public static void main(String[] args) throws Exception {
20          BancoCaixa bancoCaixa = new BancoCaixa(new BoletoSimpleFactory());
21          bancoCaixa.gerarBoleto(10, 100);
22          bancoCaixa.gerarBoleto(30, 100);
23          bancoCaixa.gerarBoleto(60, 100);
24      }
25
26  }
```



Simple Factory C#

Padrões de Projeto Criacional I

Prof. Me Jefferson Passerini

Simple Factory

- Vamos apenas mudar o local onde instanciamos os objetos.

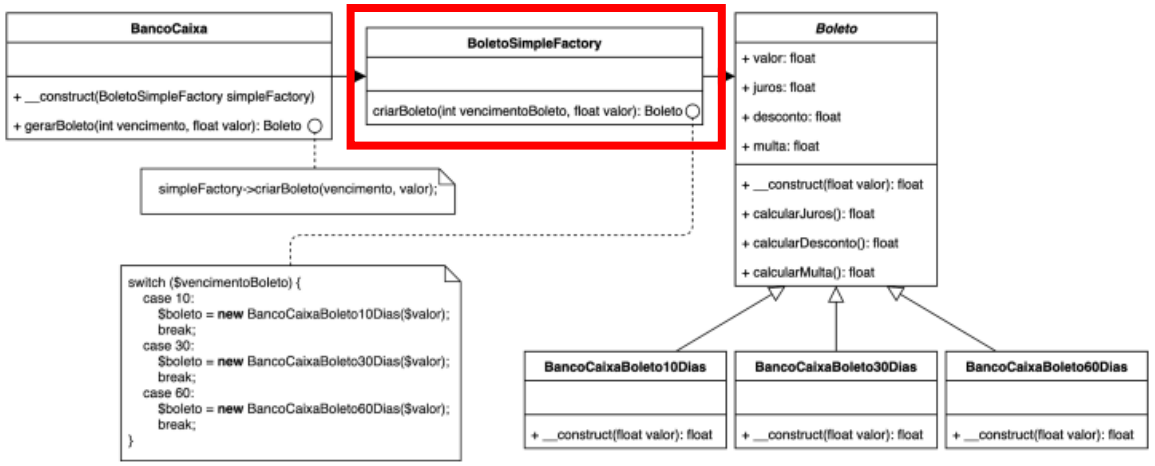
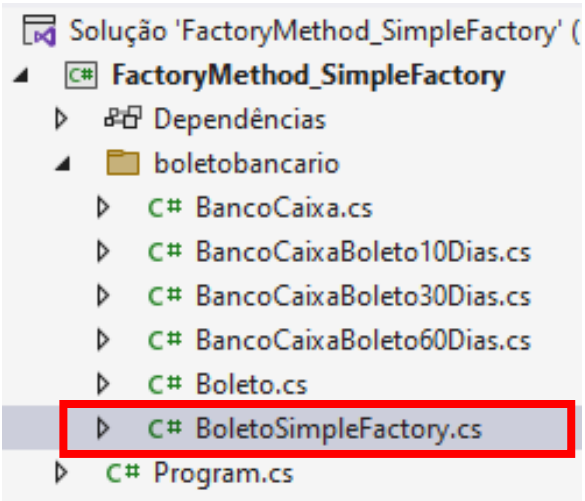


Diagrama de classes da estrutura do módulo de cobranças utilizando *Simple Factory*.




```
1  using FactoryMethod_SimpleFactory.boletobancario;
2  using System;
3  using System.Collections.Generic;
4  using System.Drawing;
5  using System.Linq;
6  using System.Text;
7  using System.Threading.Tasks;
8
9  namespace FactoryMethod_SimpleFactory.boletobancario
10 {
       3 referências
11     public class BoletoSimpleFactory
12     {
           1 referência
13         public Boleto criarBoleto(int vencimento, double valor) {
14             Boleto boleto;
15             switch (vencimento) {
16                 case 10:
17                     boleto = new BancoCaixaBoleto10Dias(valor);
18                     break;
19                 case 30:
20                     boleto = new BancoCaixaBoleto30Dias(valor);
21                     break;
22                 case 60:
23                     boleto = new BancoCaixaBoleto60Dias(valor);
24                     break;
25                 default:
26                     throw new Exception("Vencimento indisponível");
27             }
28             return boleto;
29         }
30     }
31 }
```

Simple Factory

- Vamos apenas mudar o local onde instanciamos os objetos.

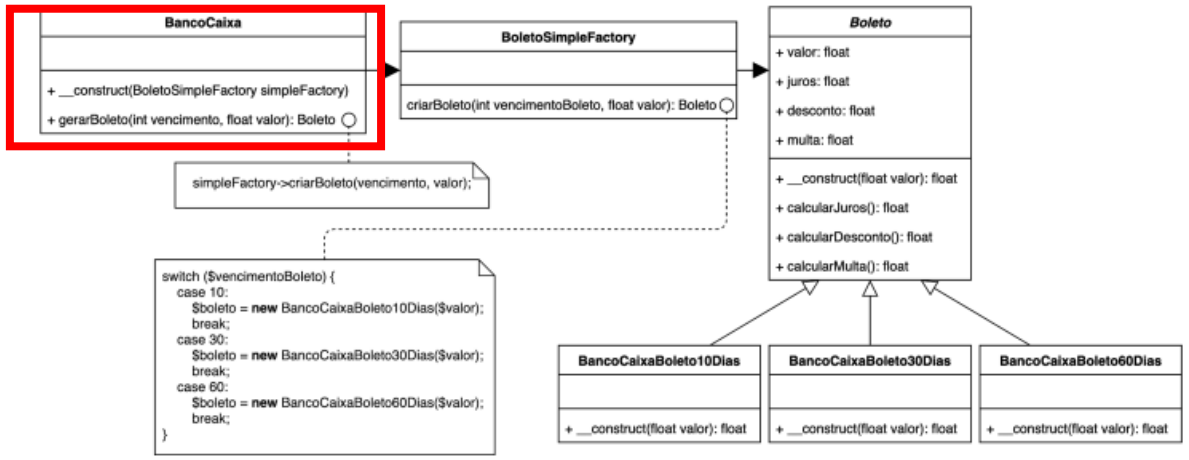
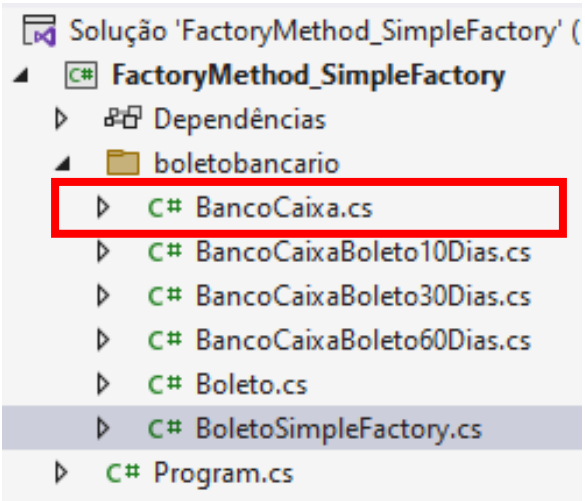


Diagrama de classes da estrutura do módulo de cobranças utilizando *Simple Factory*.




```
1   using System;
2   using System.Collections.Generic;
3   using System.Drawing;
4   using System.Linq;
5   using System.Text;
6   using System.Threading.Tasks;
7
8   namespace FactoryMethod_SimpleFactory.boletobancario
9   {
        3 referências
10      public class BancoCaixa
11      {
12          private BoletoSimpleFactory boletoSimpleFactory;
13
            1 referência
14          public BancoCaixa(BoletoSimpleFactory boletoSimpleFactory)
15          {
16              this.boletoSimpleFactory = boletoSimpleFactory;
17          }
18
            3 referências
19          public void gerarBoleto(int vencimento, double valor)
20          {
21              Boleto boleto = this.boletoSimpleFactory.criarBoleto(vencimento, valor);
22
23              Console.WriteLine("Boleto gerado com sucesso no valor de R$" + valor);
24              Console.WriteLine("Valor Juros R$" + boleto.calcularJuros());
25              Console.WriteLine("Valor Multa R$" + boleto.calcularMulta());
26              Console.WriteLine("Valor Desconto R$" + boleto.calcularDesconto());
27              Console.WriteLine("-----------------------------------------------");
28          }
29
30      }
31  }
```

Simple Factory

- Vamos apenas mudar o local onde instanciamos os objetos.

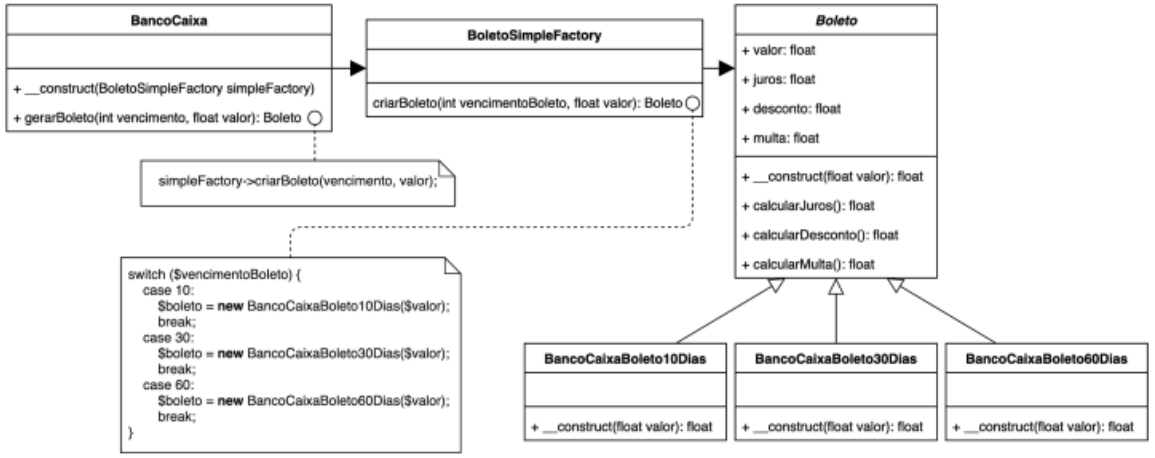
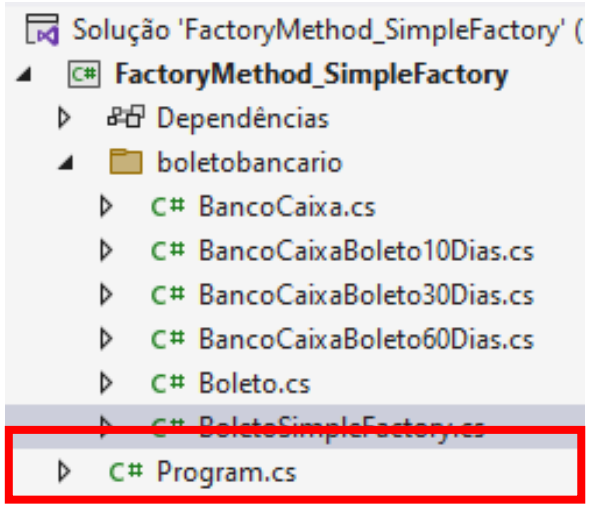
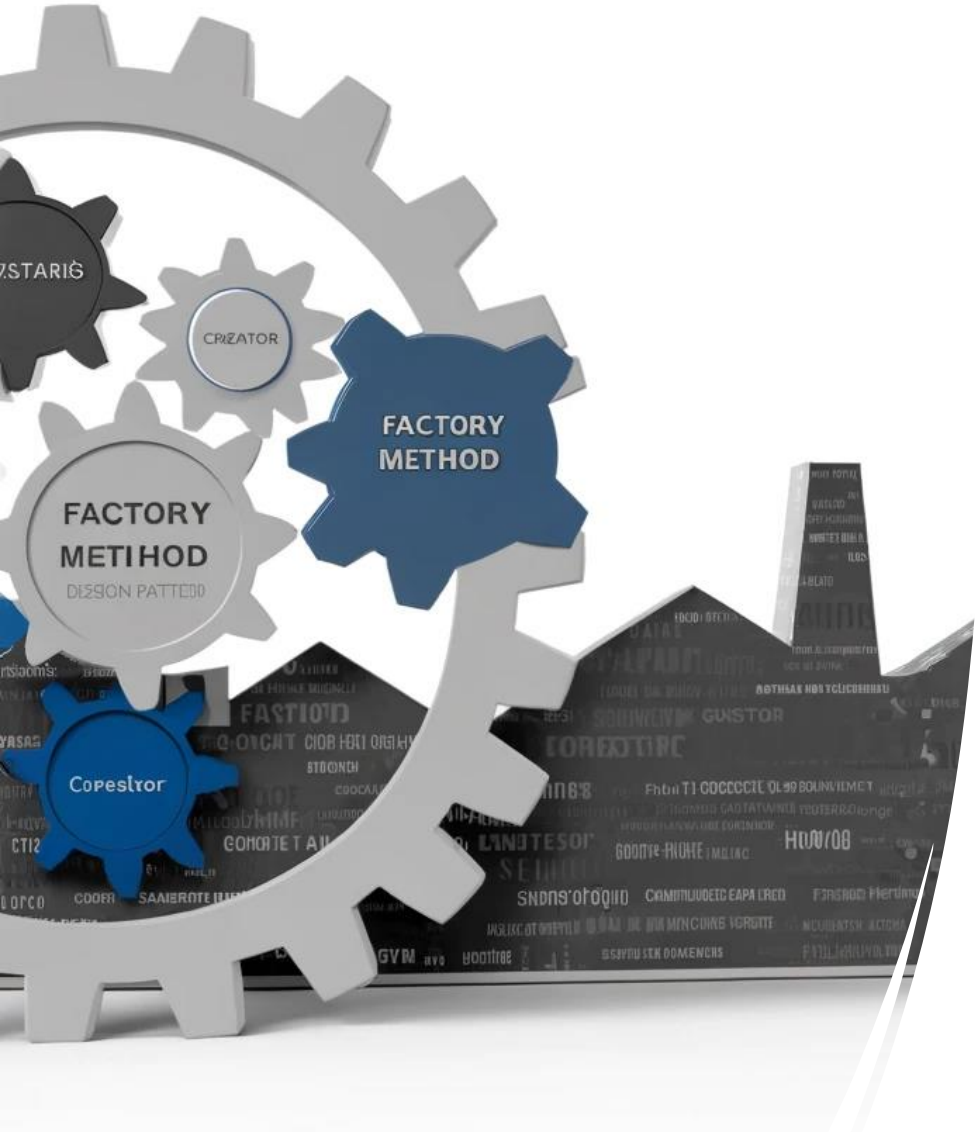


Diagrama de classes da estrutura do módulo de cobranças utilizando *Simple Factory*.




```
1   using FactoryMethod_SimpleFactory.boletobancario;
2
3   BancoCaixa bancoCaixa = new BancoCaixa(new BoletoSimpleFactory());
4   bancoCaixa.gerarBoleto(10, 100);
5   bancoCaixa.gerarBoleto(30, 100);
6   bancoCaixa.gerarBoleto(60, 100);
```

```
Boleto gerado com sucesso no valor de R$100
Valor Juros R$2
Valor Multa R$5
Valor Desconto R$10
-----------------------------------------------
Boleto gerado com sucesso no valor de R$100
Valor Juros R$5
Valor Multa R$10
Valor Desconto R$5
-----------------------------------------------
Boleto gerado com sucesso no valor de R$100
Valor Juros R$10
Valor Multa R$20
Valor Desconto R$0
-----------------------------------------------
```



Simple Factory (continuação)

Padrões de Projeto Criacional I

Prof. Me Jefferson Passerini

Simple Factory

- Como é possível observar nas mudanças no diagrama de classes e no código, a criação dos objetos saiu da classe **BancoCaixa** e foi para o método **criarBoleto()** da classe **BoletoSimpleFactory**.
- Você pode estar se perguntando se isso apenas não mudou o problema de lugar, de fato, no nosso caso sim.
- Porém agora a criação dos objetos estão encapsuladas na classe **BoletoSimpleFactory**.

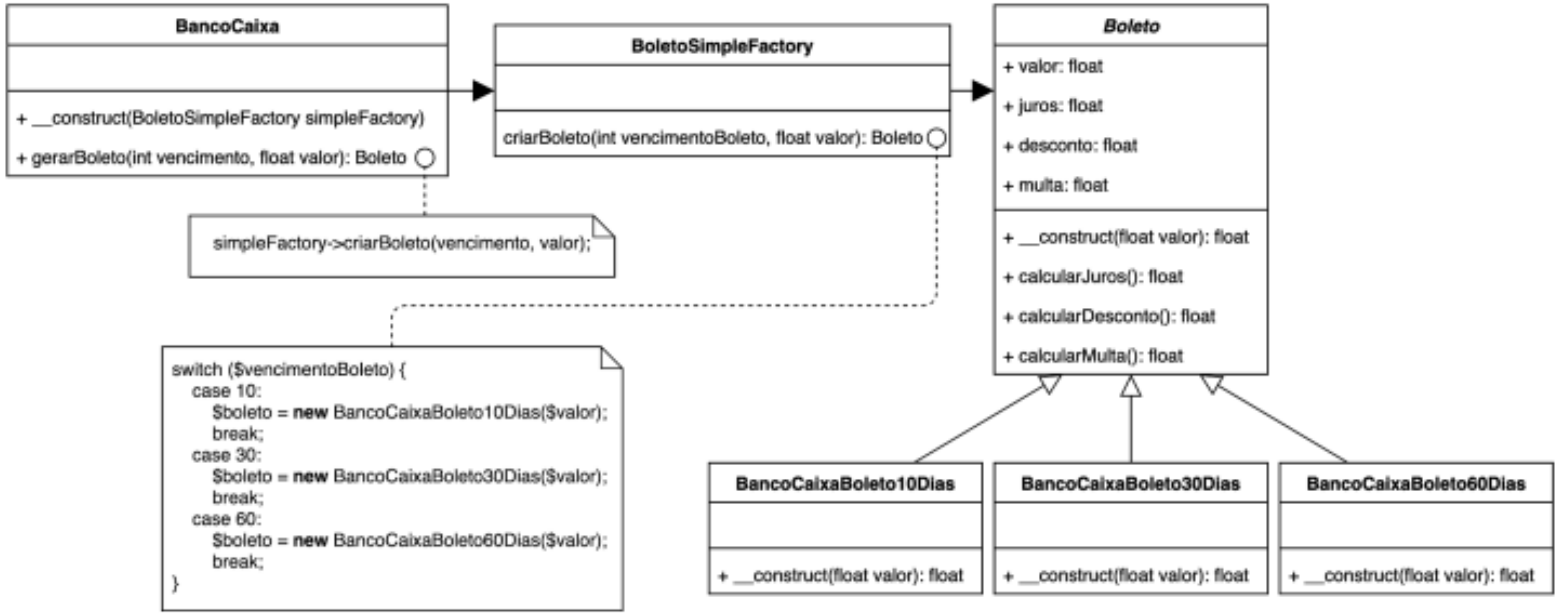


Diagrama de classes da estrutura do módulo de cobranças utilizando *Simple Factory*.

Simple Factory

- Caso os objetos criados pela classe **BoletoSimpleFactory** fossem utilizados em outros locais do software sua criação estaria unificada e controlada pela classe **BoletoSimpleFactory** evitando duplicação de código.
- E ainda caso a criação dos objetos fosse complexa os clientes não precisariam se preocupar com ela, bastaria pedir para a classe BoletoSimpleFactory criar e retornar o objeto pronto.

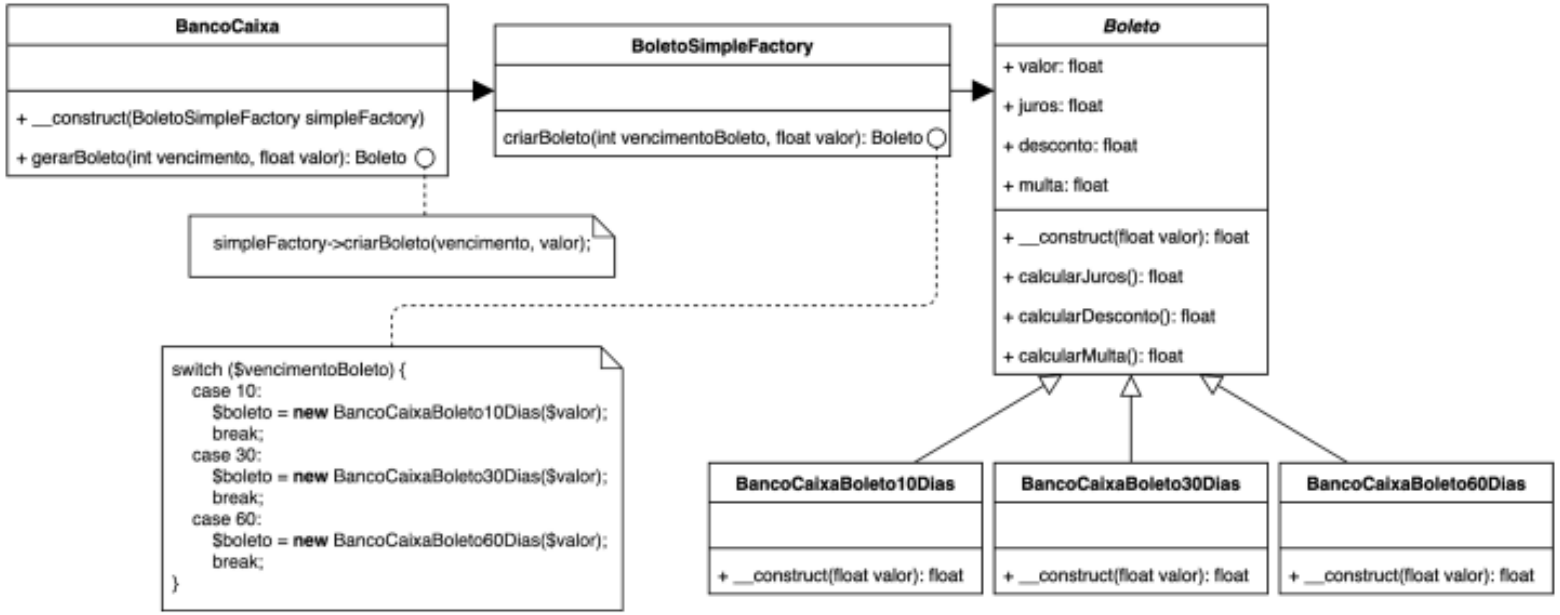


Diagrama de classes da estrutura do módulo de cobranças utilizando *Simple Factory*.

Simple Factory

- Essa estratégia onde se isola a criação de objetos em uma classe separada se chama **Simple Factory**, em português, **Fábrica Simples**.
- **Não se trata de um padrão de projeto, porém esse conceito irá nos ajudar a entender o padrão Factory Method que veremos a partir de agora.**

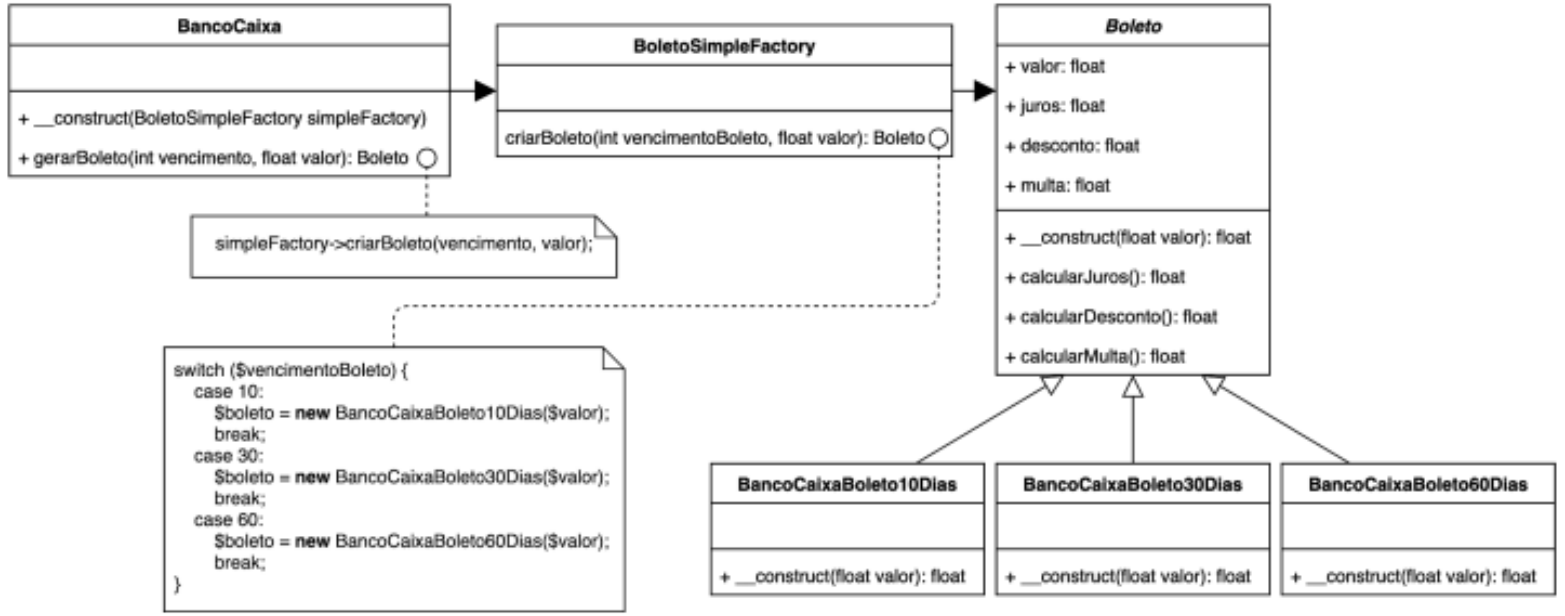
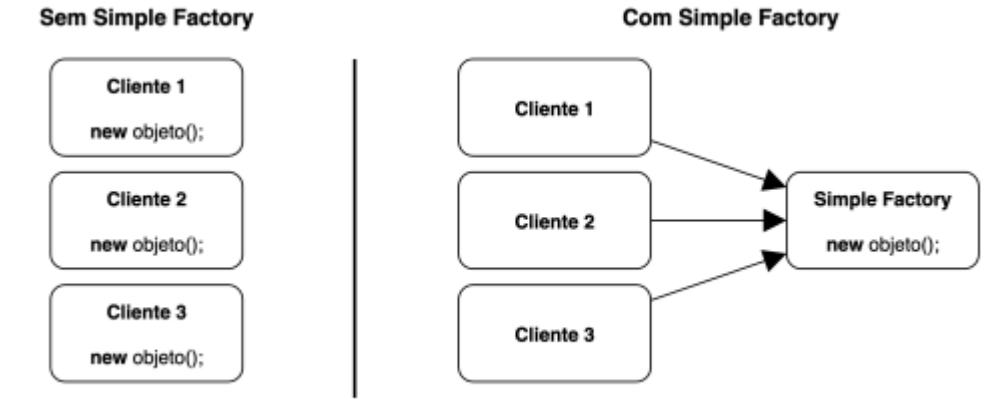
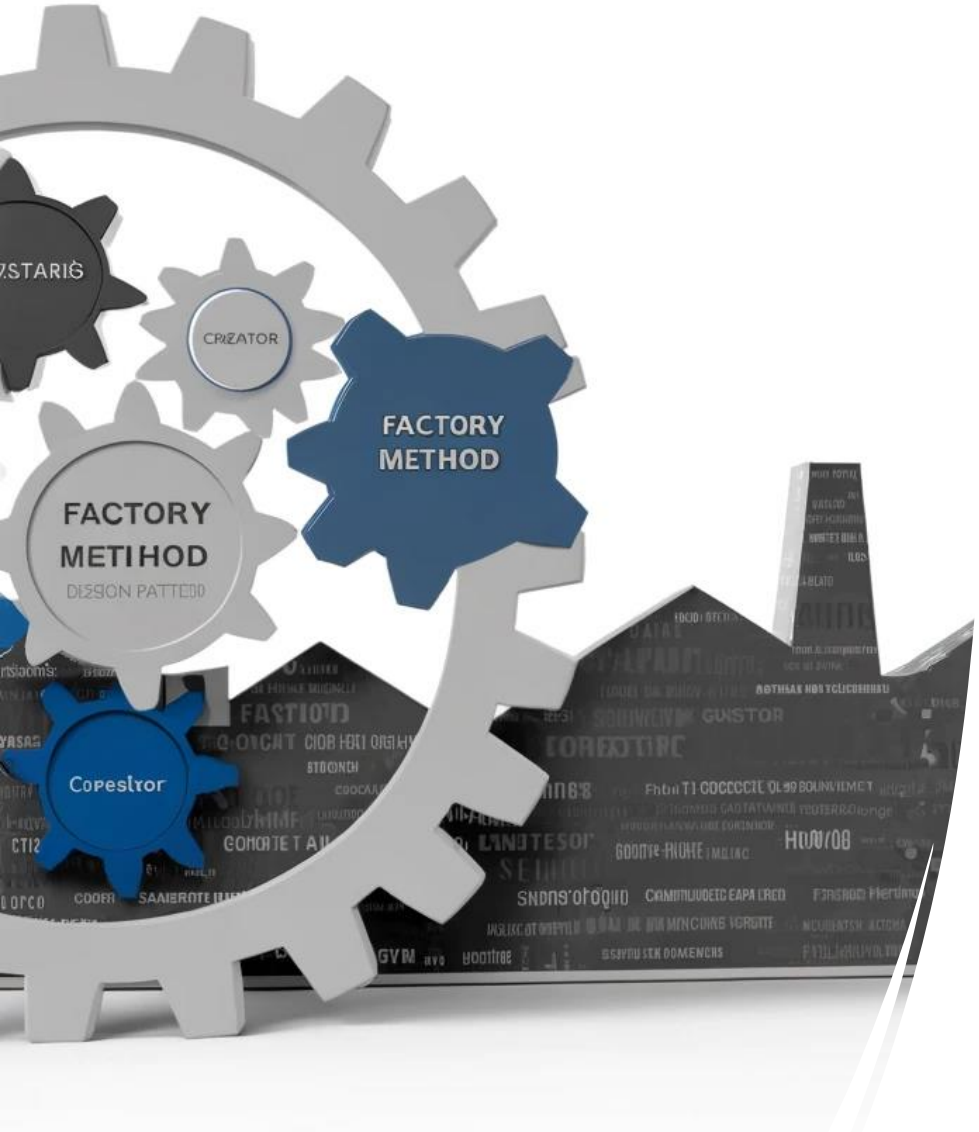


Diagrama de classes da estrutura do módulo de cobranças utilizando *Simple Factory*.



Esquema de reutilização de código utilizando o Simple Factory.



Factory Method Java

Padrões de Projeto Criacional I

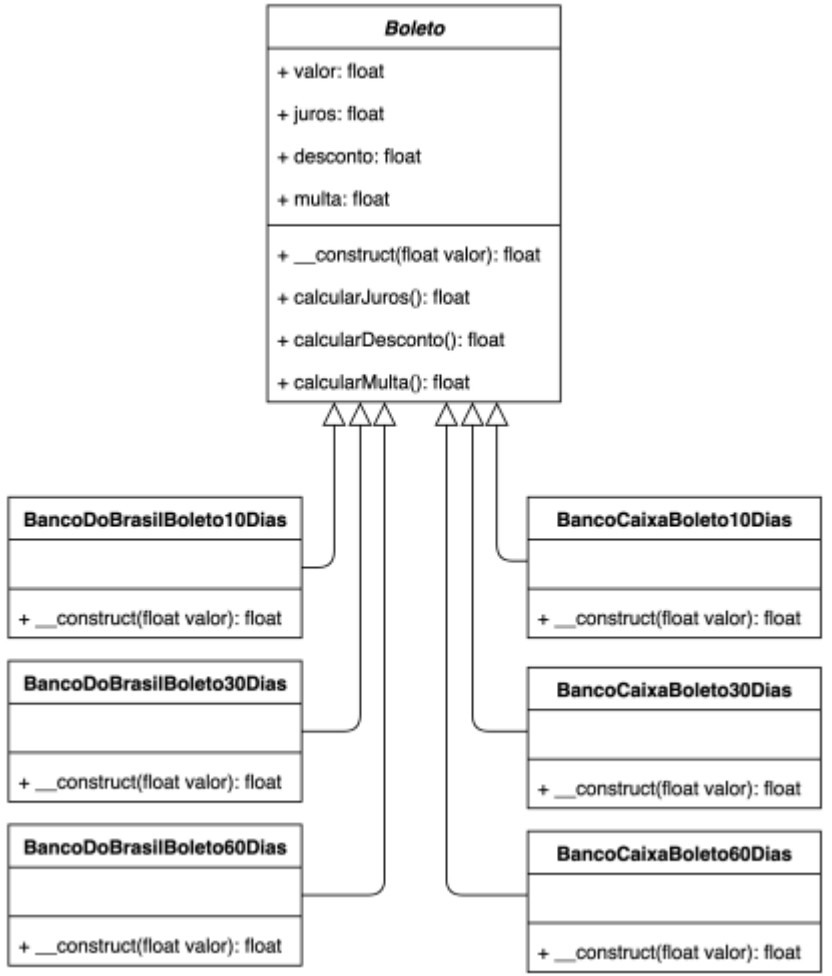
Prof. Me Jefferson Passerini

Factory Method

- Suponha que surgiu a necessidade de emitir boletos de mais um banco.
- Agora além de **BancoCaixa** precisaremos emitir boletos do **BancoDoBrasil**.
- Nós poderíamos criar outra Simple Factory para os boletos do BancoDoBrasil, ***o problema desta abordagem é que não temos controle se todos os bancos são iguais***, o Cliente não teria garantias que poderia emitir boletos do BancoDoBrasil da mesma forma que emite em BancoCaixa.

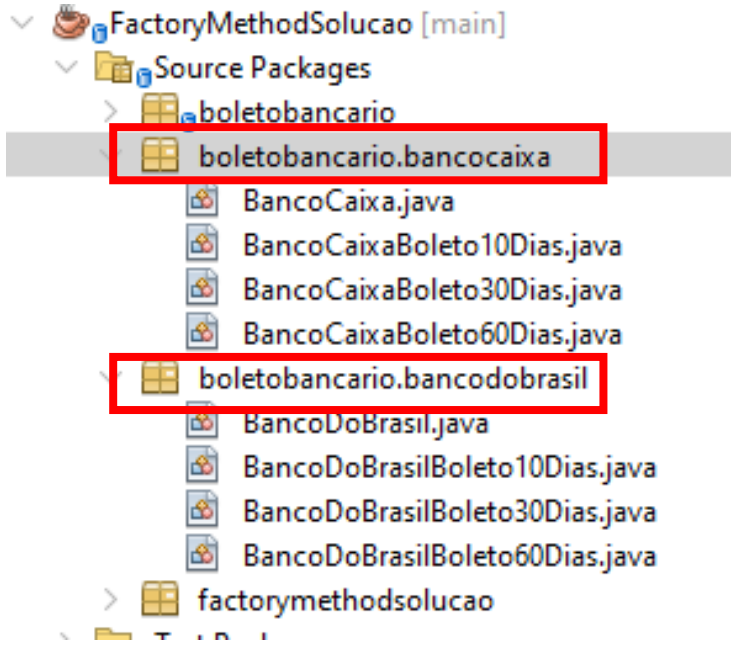
Banco	Dias p/ vencimento	Juros	Desconto	Multa
BancoCaixa	10	2%	10%	5%
BancoCaixa	30	5%	5%	10%
BancoCaixa	60	10%	0%	20%
BancoDoBrasil	10	3%	5%	2%
BancoDoBrasil	30	5%	2%	5%
BancoDoBrasil	60	10%	0%	15%

Diferentes vencimentos de boletos e seus respectivos cálculos baseados em seu valor.

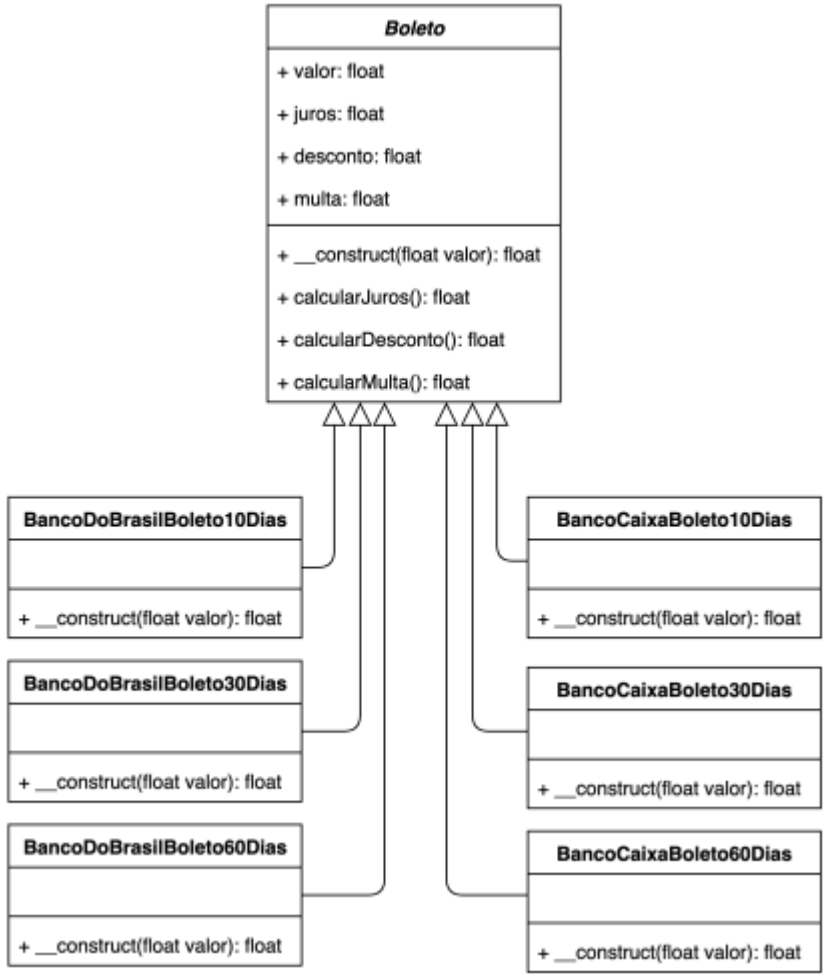


Factory Method

- Vamos implementar as classes de boletos do BancoDoBrasil com as condições da tabela do slide anterior.

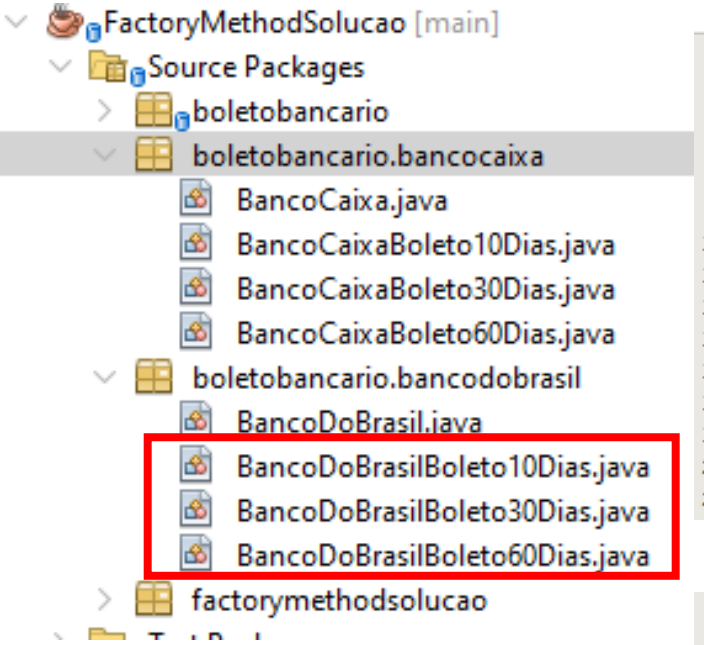


- Separe os boletos de **BancoCaixa** e de **BancoDoBrasil** em pacotes diferentes.
- Esses pacotes estarão abaixo do pacote **boletobancario**.
- Crie os pacotes:
  - **bancocaixa**
  - **bancodobrasil**.
- **Coloque os boletos de BancoCaixa em seu pacote.**
- **E crie os boletos para o BancoDoBrasil em seu pacote.**



Factory Method

- Crie as classes de boletos do BancoDoBrasil como abaixo.




```java
...4 lines
package boletobancario.bancodobrasil;

import boletobancario.Boleto;

/**...4 lines */
public class BancoDoBrasilBoleto10Dias extends Boleto {

    public BancoDoBrasilBoleto10Dias(double valor) {
        this.valor=valor;
        this.juros = 0.03;
        this.desconto=0.05;
        this.multa=0.02;
    }
}
```

```java
...4 lines
package boletobancario.bancodobrasil;

import boletobancario.Boleto;

/**...4 lines */
public class BancoDoBrasilBoleto30Dias extends Boleto {

    public BancoDoBrasilBoleto30Dias(double valor) {
        this.valor=valor;
        this.juros = 0.05;
        this.desconto=0.02;
        this.multa=0.05;
    }

}
```

```java
...4 lines
package boletobancario.bancodobrasil;

import boletobancario.Boleto;

/**...4 lines */
public class BancoDoBrasilBoleto60Dias extends Boleto {

    public BancoDoBrasilBoleto60Dias(double valor) {
        this.valor=valor;
        this.juros = 0.1;
        this.desconto=0;
        this.multa=0.15;
    }
}
```

Factory Method

- Agora que já temos os boletos, podemos criar a classe Banco

```
public abstract class Banco {

    protected abstract Boleto criarBoleto(int vencimento, double valor);

    public Boleto gerarBoleto(int vencimento, double valor){

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

        System.out.println("Boleto gerado com sucesso no valor de R$"+valor);
        System.out.println("Valor Juros R$"+boleto.calcularJuros());
        System.out.println("Valor Multa R$"+boleto.calcularMulta());
        System.out.println("Valor Desconto R$"+boleto.calcularDesconto());
        System.out.println("-----------------------------------------------------------------");

        return boleto;
    }

}
```

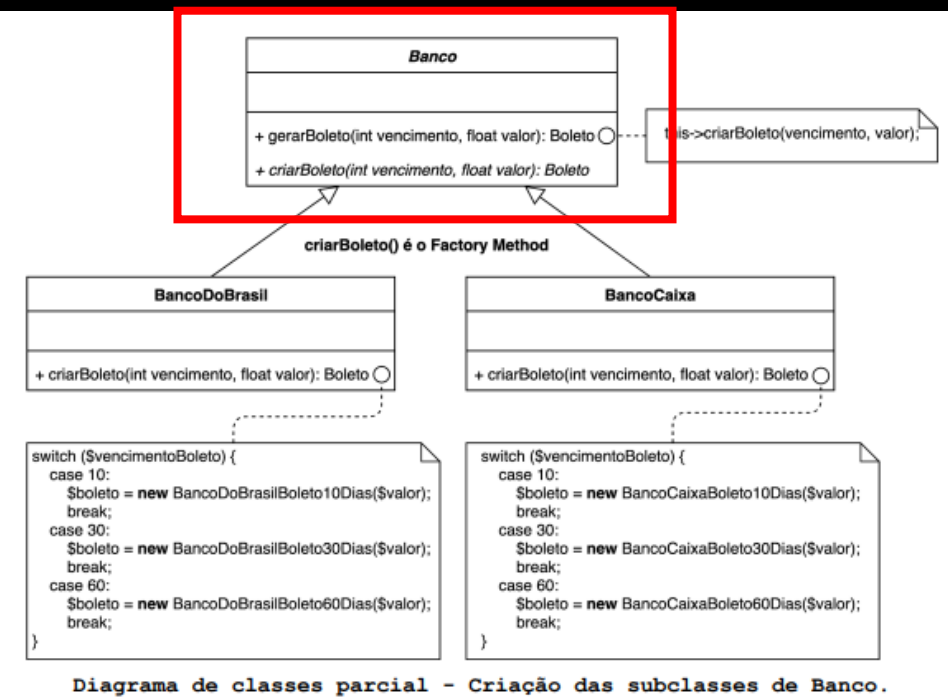
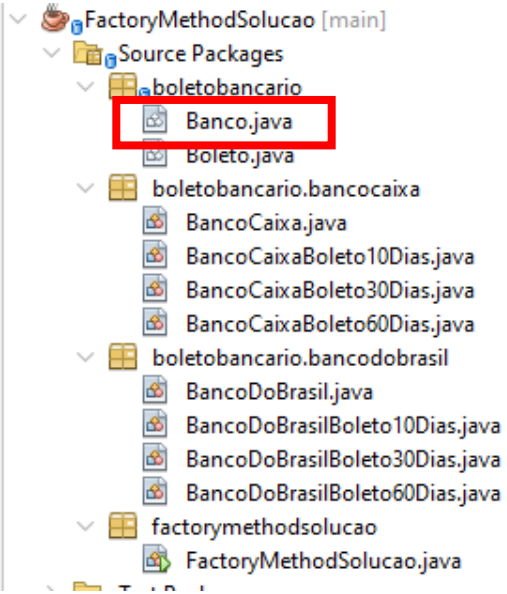


Diagrama de classes parcial - Criação das subclasses de Banco.

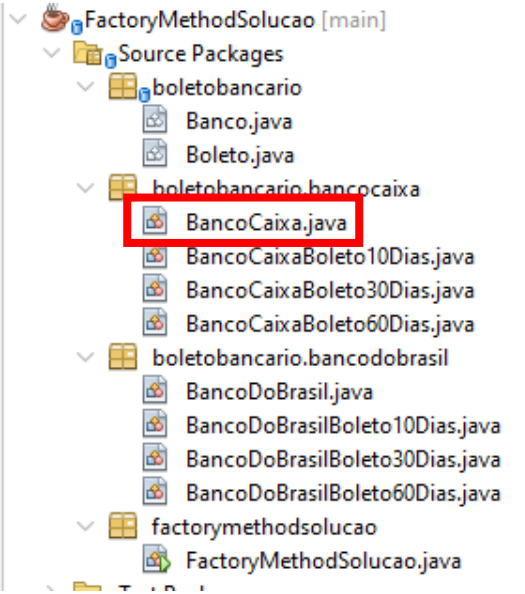


- O método **gerarBoleto()** da classe **Banco** chama o **método abstrato criarBoleto()** da **subclasse sem conhecê-la**.
- **Ou seja, método gerarBoleto() pode chamar o método criarBoleto() da subclasse BancoDoBrasil ou BancoCaixa, quem escolhe é o Cliente no momento em que ele instancia um objeto.**
- Se o Cliente instanciar a subclasse BancoDoBrasil consequentemente um boleto de 10, 30 ou 60 dias do BancoDoBrasil será criado pelo método criarBoleto(). Da mesma forma, se o Cliente instanciar a subclasse BancoCaixa consequentemente um boleto de 10, 30 ou 60 dias do BancoCaixa será criado pelo método criarBoleto().

Factory Method

- Agora vamos implementar a classe **BancoDoBrasil** e refatorar a classe **BancoCaixa**
- **Refatore a classe BancoCaixa**

```java
import boletobancario.Banco;
import boletobancario.Boleto;

/**...4 lines */
public class BancoCaixa extends Banco {

    @Override
    protected Boleto criarBoleto(int vencimento, double valor){
        Boleto boleto = null;
        switch (vencimento) {
            case 10:
                boleto = new BancoCaixaBoleto10Dias(valor);
                break;
            case 30:
                boleto = new BancoCaixaBoleto30Dias(valor);
                break;
            case 60:
                boleto = new BancoCaixaBoleto60Dias(valor);
                break;
            default:
            {
                try {
                    throw new Exception("Vencimento indisponível");
                } catch (Exception ex) {
                    System.out.println(ex.getMessage());
                }
            }
        }
        return boleto;
    }
}
```



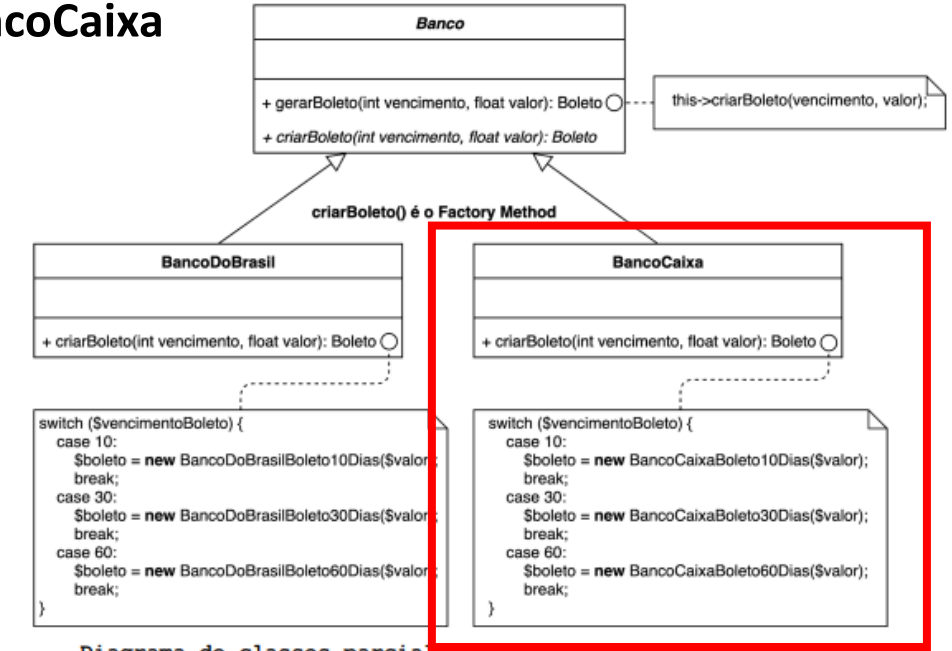


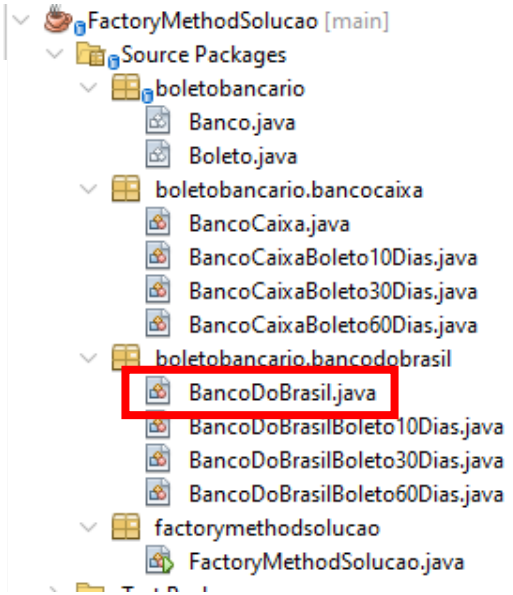
Diagrama de classes parcial – Criação das subclasses de Banco.

- Como é possível observar no diagrama de classes acima o método **criarBoleto()** da a classe **BancoDoBrasil** cria os objetos: BancoDoBrasilBoleto10Dias; BancoDoBrasilBoleto30Dias ou BancoDoBrasilBoleto60Dias.
- Já o método **criarBoleto()** da classe **BancoCaixa** cria os objetos: BancoCaixaBoleto10Dias; BancoCaixaBoleto30Dias ou BancoCaixaBoleto60Dias.

Factory Method

- Agora vamos implementar a classe **BancoDoBrasil** e refatorar a classe **BancoCaixa**
- **Refatore a classe BancoCaixa**

```java
import boletobancario.Banco;
import boletobancario.Boleto;

/**...4 lines */
public class BancoDoBrasil extends Banco{

    @Override
    protected Boleto criarBoleto(int vencimento, double valor) {
        Boleto boleto = null;
        switch (vencimento) {
            case 10:
                boleto = new BancoDoBrasilBoleto10Dias(valor);
                break;
            case 30:
                boleto = new BancoDoBrasilBoleto30Dias(valor);
                break;
            case 60:
                boleto = new BancoDoBrasilBoleto60Dias(valor);
                break;
            default:
            {
                try {
                    throw new Exception("Vencimento indisponível");
                } catch (Exception ex) {
                    System.out.println(ex.getMessage());
                }
            }
        }
        return boleto;
    }
}
```



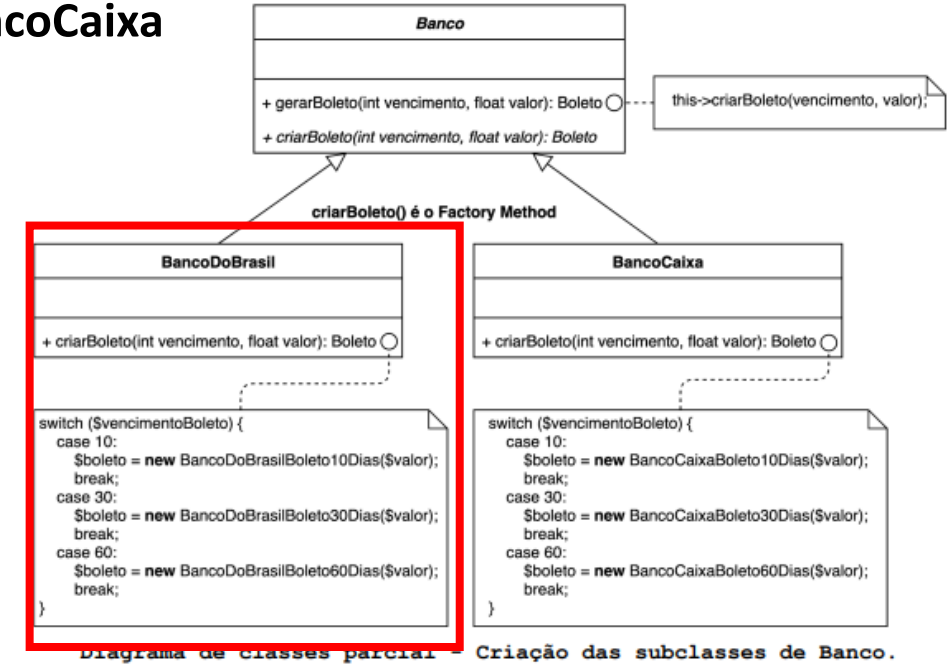


Diagrama de classes parcial – Criação das subclasses de Banco.

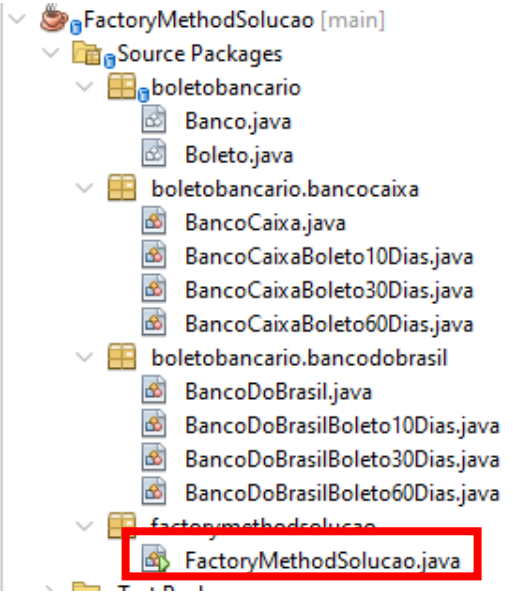
- Como é possível observar no diagrama de classes acima o método **criarBoleto()** da a classe **BancoDoBrasil** cria os objetos: BancoDoBrasilBoleto10Dias; BancoDoBrasilBoleto30Dias ou BancoDoBrasilBoleto60Dias.
- Já o método **criarBoleto()** da classe **BancoCaixa** cria os objetos: BancoCaixaBoleto10Dias; BancoCaixaBoleto30Dias ou BancoCaixaBoleto60Dias.

Factory Method

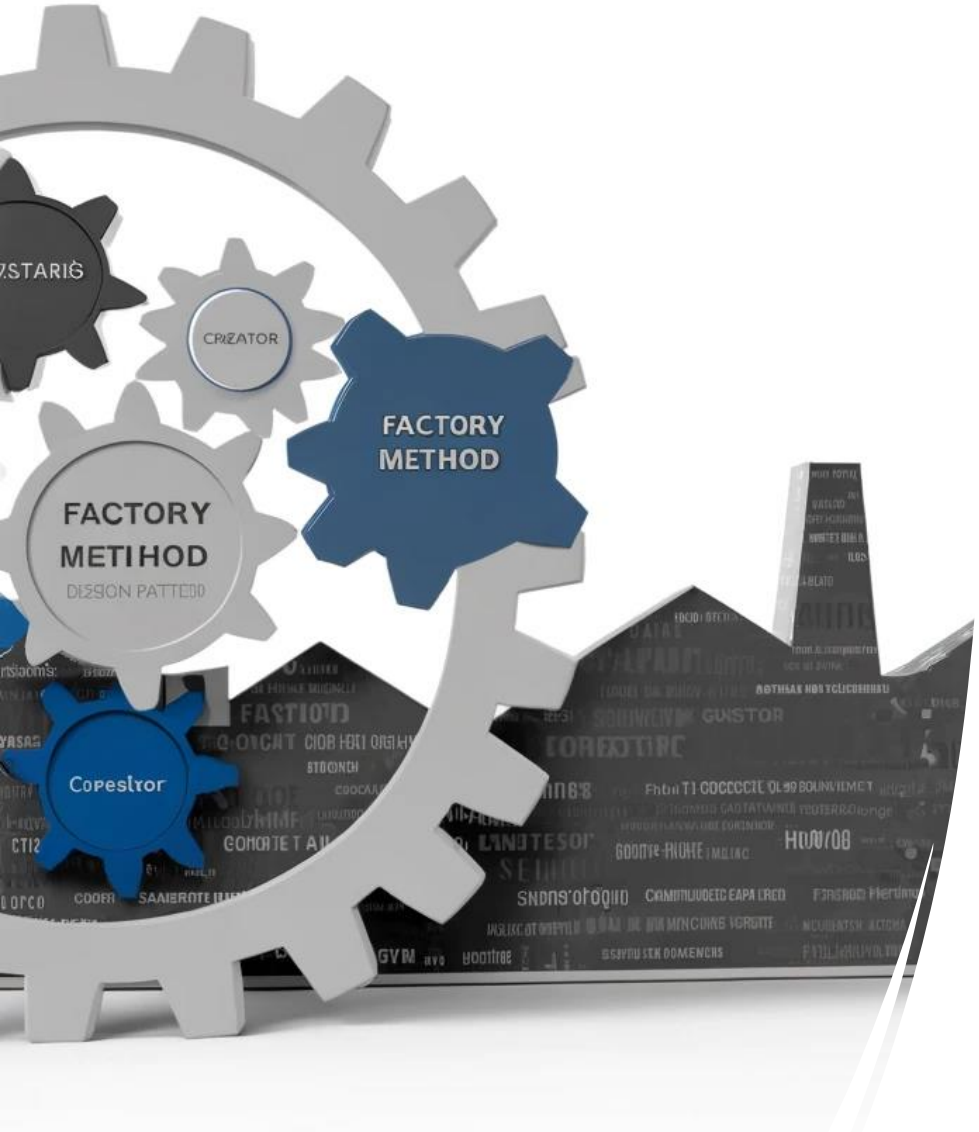
- Atualize a main do seu projeto.

```
...4 lines
package factorymethodsolucao;

import boletobancario.Banco;
import boletobancario.bancocaixa.BancoCaixa;
import boletobancario.bancodobrasil.BancoDoBrasil;

/**...4 lines */
public class FactoryMethodSolucao {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {

        System.out.println("Banco Caixa --------------------------------");
        Banco bancoCaixa = new BancoCaixa();
        bancoCaixa.gerarBoleto(10, 100);
        bancoCaixa.gerarBoleto(30, 100);
        bancoCaixa.gerarBoleto(60, 100);

        System.out.println("Banco do Brasil --------------------------------");
        Banco bancoDoBrasil = new BancoDoBrasil();
        bancoDoBrasil.gerarBoleto(10, 100);
        bancoDoBrasil.gerarBoleto(30, 100);
        bancoDoBrasil.gerarBoleto(60, 100);
    }

}
```

```
run:
Banco Caixa --------------------------------
Boleto gerado com sucesso no valor de R$100.0
Valor Juros R$2.0
Valor Multa R$5.0
Valor Desconto R$10.0
---------------------------------------------------------------
Boleto gerado com sucesso no valor de R$100.0
Valor Juros R$5.0
Valor Multa R$10.0
Valor Desconto R$5.0
---------------------------------------------------------------
Boleto gerado com sucesso no valor de R$100.0
Valor Juros R$10.0
Valor Multa R$20.0
Valor Desconto R$0.0
---------------------------------------------------------------
Banco do Brasil --------------------------------
Boleto gerado com sucesso no valor de R$100.0
Valor Juros R$3.0
Valor Multa R$2.0
Valor Desconto R$5.0
---------------------------------------------------------------
Boleto gerado com sucesso no valor de R$100.0
Valor Juros R$5.0
Valor Multa R$5.0
Valor Desconto R$2.0
---------------------------------------------------------------
Boleto gerado com sucesso no valor de R$100.0
Valor Juros R$10.0
Valor Multa R$15.0
Valor Desconto R$0.0
---------------------------------------------------------------
BUILD SUCCESSFUL (total time: 7 seconds)
```



- Repare em nosso teste (Cliente) que sempre estamos chamando o método **gerarBoleto()** independe se o objeto banco aponta para um objeto da classe BancoCaixa ou BancoDoBrasil.
- ***O que garante para o Cliente que o método gerarBoleto() existe é o fato de que BancoCaixa e BancoDoBrasil são subclasses de Banco.*** O método criarBoleto() que é abstrato na classe Banco sempre irá retornar uma instância de uma subclasse de Boleto. Temos então duas hierarquias paralelas de classes no padrão Factory Method.



Factory Method C#

Padrões de Projeto Criacional I

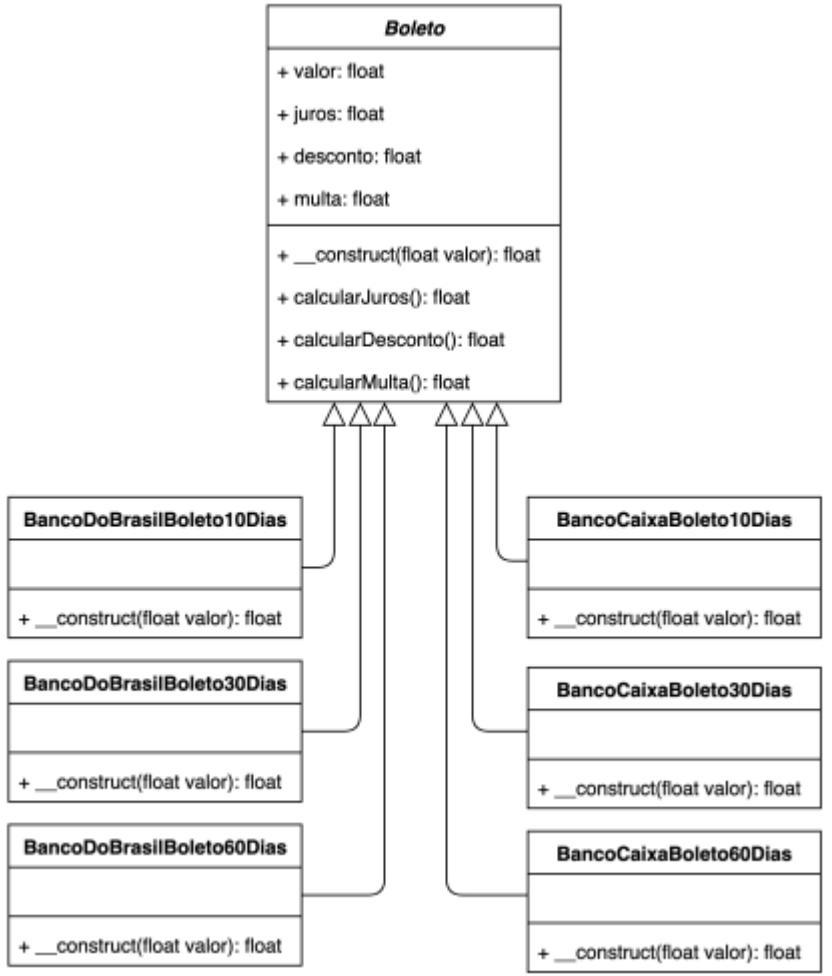
Prof. Me Jefferson Passerini

Factory Method

- Suponha que surgiu a necessidade de emitir boletos de mais um banco.
- Agora além de **BancoCaixa** precisaremos emitir boletos do **BancoDoBrasil**.
- Nós poderíamos criar outra Simple Factory para os boletos do BancoDoBrasil, ***o problema desta abordagem é que não temos controle se todos os bancos são iguais***, o Cliente não teria garantias que poderia emitir boletos do BancoDoBrasil da mesma forma que emite em BancoCaixa.

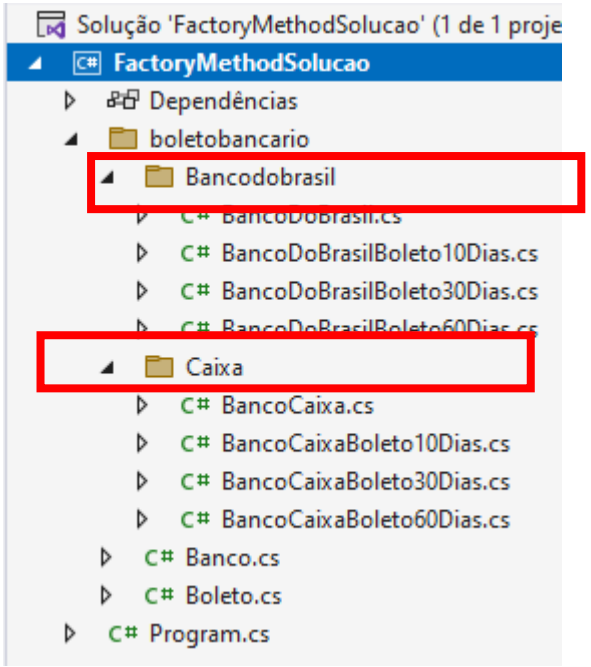
Banco	Dias p/ vencimento	Juros	Desconto	Multa
BancoCaixa	10	2%	10%	5%
BancoCaixa	30	5%	5%	10%
BancoCaixa	60	10%	0%	20%
BancoDoBrasil	10	3%	5%	2%
BancoDoBrasil	30	5%	2%	5%
BancoDoBrasil	60	10%	0%	15%

Diferentes vencimentos de boletos e seus respectivos cálculos baseados em seu valor.

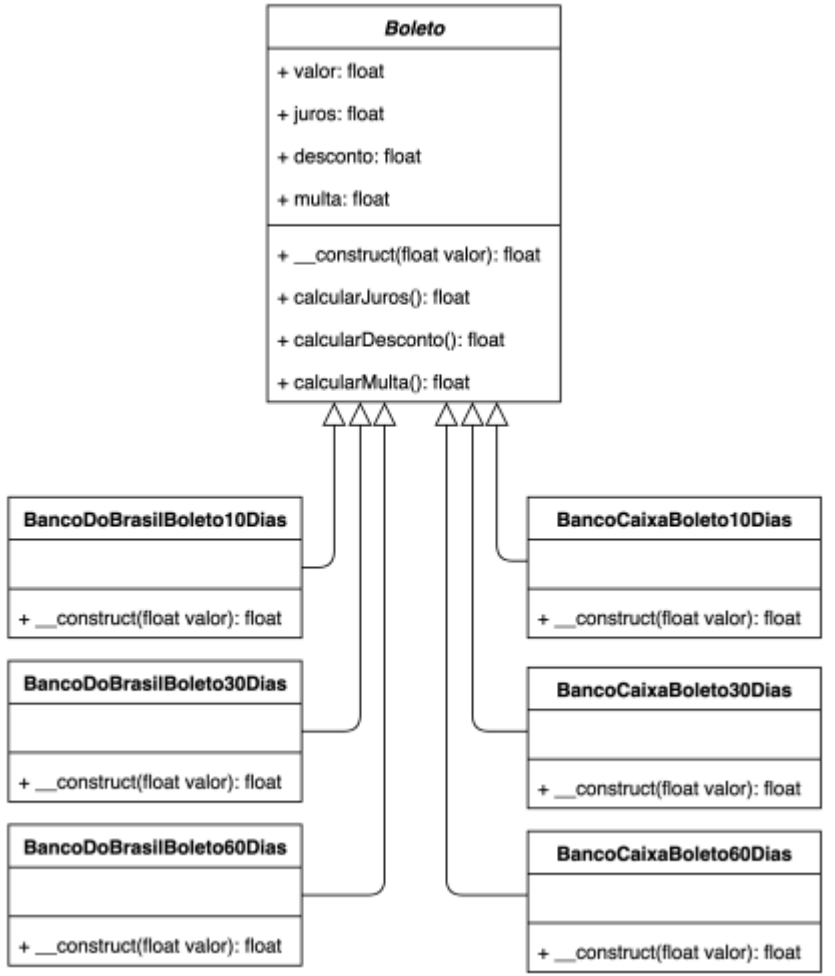


Factory Method

- Vamos implementar as classes de boletos do BancoDoBrasil com as condições da tabela do slide anterior.

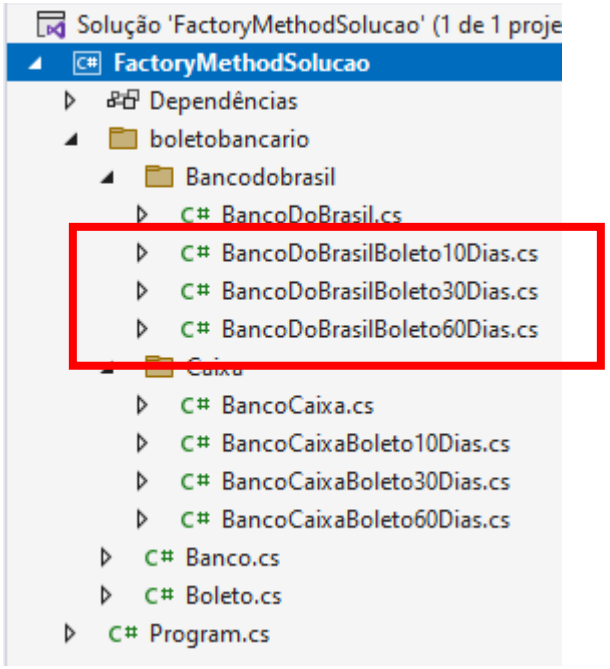


- Separe os boletos de **BancoCaixa** e de **BancoDoBrasil** em pacotes diferentes.
- Esses pacotes estarão abaixo do pacote **boletobancario**.
- Crie os pacotes:
  - **caixa**
  - **bancodobrasil**.
- **Coloque os boletos de BancoCaixa em seu pacote.**
- **E crie os boletos para o BancoDoBrasil em seu pacote.**



Factory Method

- Crie as classes de boletos do BancoDoBrasil como abaixo.




```cs
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using FactoryMethodSolucao.boletobancario;

namespace FactoryMethodSolucao.boletobancario.Bancodobrasil
{
    2 referências
    public class BancoDoBrasilBoleto10Dias : Boleto
    {
        1 referência
        public BancoDoBrasilBoleto10Dias(double valor)
        {
            this.valor = valor;
            juros = 0.03;
            desconto = 0.05;
            multa = 0.02;
        }

    }
}
```

```cs
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using FactoryMethodSolucao.boletobancario;

namespace FactoryMethodSolucao.boletobancario.Bancodobrasil
{
    2 referências
    public class BancoDoBrasilBoleto30Dias : Boleto
    {
        1 referência
        public BancoDoBrasilBoleto30Dias(double valor)
        {
            this.valor = valor;
            juros = 0.05;
            desconto = 0.02;
            multa = 0.05;
        }
    }
}
```

```cs
using System;
using System.Collections.Generic;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using FactoryMethodSolucao.boletobancario;

namespace FactoryMethodSolucao.boletobancario.Bancodobrasil
{
    2 referências
    public class BancoDoBrasilBoleto60Dias : Boleto
    {
        1 referência
        public BancoDoBrasilBoleto60Dias(double valor)
        {
            this.valor = valor;
            juros = 0.1;
            desconto = 0;
            multa = 0.15;
        }
    }
}
```

Factory Method

- Agora que já temos os boletos, podemos criar a classe Banco

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace FactoryMethodSolucao.boletobancario
{
    4 referências
    public abstract class Banco
    {
        3 referências
        protected abstract Boleto criarBoleto(int vencimento, double valor);

        6 referências
        public void gerarBoleto(int vencimento, double valor)
        {
            Boleto boleto = this.criarBoleto(vencimento, valor);

            Console.WriteLine("Boleto gerado com sucesso no valor de R$" + valor);
            Console.WriteLine("Valor Juros R$" + boleto.calcularJuros());
            Console.WriteLine("Valor Multa R$" + boleto.calcularMulta());
            Console.WriteLine("Valor Desconto R$" + boleto.calcularDesconto());
            Console.WriteLine("-------------------------------------------------");
        }
    }
}
```

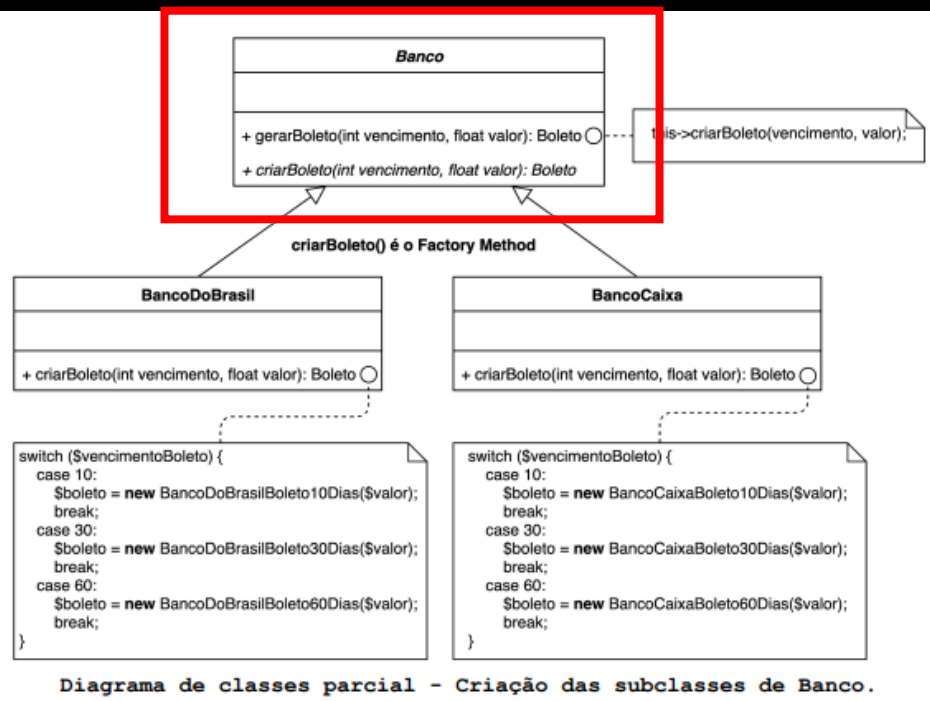
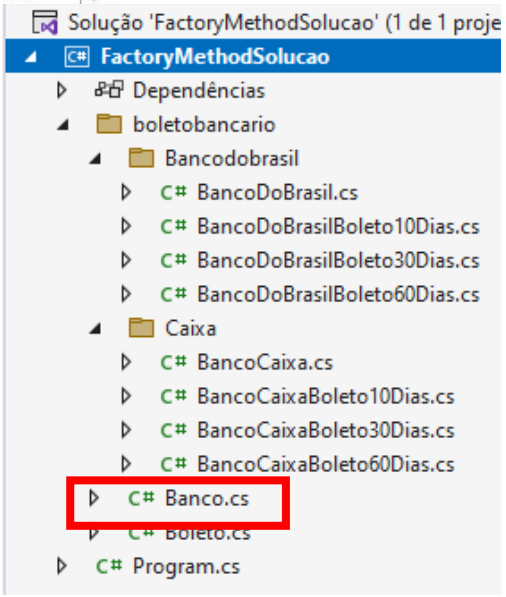


Diagrama de classes parcial - Criação das subclasses de Banco.

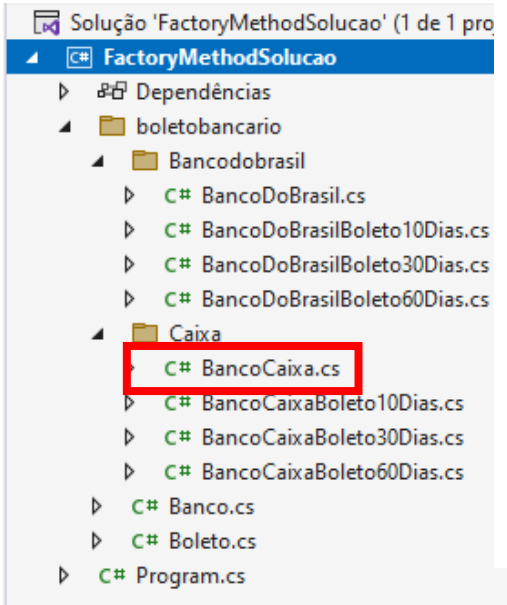


- O método **gerarBoleto()** da classe **Banco** chama o **método abstrato criarBoleto()** da **subclasse sem conhecê-la**.
- **Ou seja, método gerarBoleto() pode chamar o método criarBoleto() da subclasse BancoDoBrasil ou BancoCaixa, quem escolhe é o Cliente no momento em que ele instancia um objeto.**
- Se o Cliente instanciar a subclasse BancoDoBrasil consequentemente um boleto de 10, 30 ou 60 dias do BancoDoBrasil será criado pelo método criarBoleto(). Da mesma forma, se o Cliente instanciar a subclasse BancoCaixa consequentemente um boleto de 10, 30 ou 60 dias do BancoCaixa será criado pelo método criarBoleto().

Factory Method

- Agora vamos implementar a classe **BancoDoBrasil** e refatorar a classe **BancoCaixa**
- **Refatore a classe BancoCaixa**

```
using System;
using System.Collections.Generic;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace FactoryMethodSolucao.boletobancario.Caixa
{
    1 referência
    public class BancoCaixa : Banco
    {
        2 referências
        protected override Boleto criarBoleto(int vencimento, double valor)
        {
            Boleto boleto;
            switch (vencimento)
            {
                case 10:
                    boleto = new BancoCaixaBoleto10Dias(valor);
                    break;
                case 30:
                    boleto = new BancoCaixaBoleto30Dias(valor);
                    break;
                case 60:
                    boleto = new BancoCaixaBoleto60Dias(valor);
                    break;
                default:
                    throw new Exception("Vencimento indisponível");
            }
            return boleto;
        }
    }
}
```



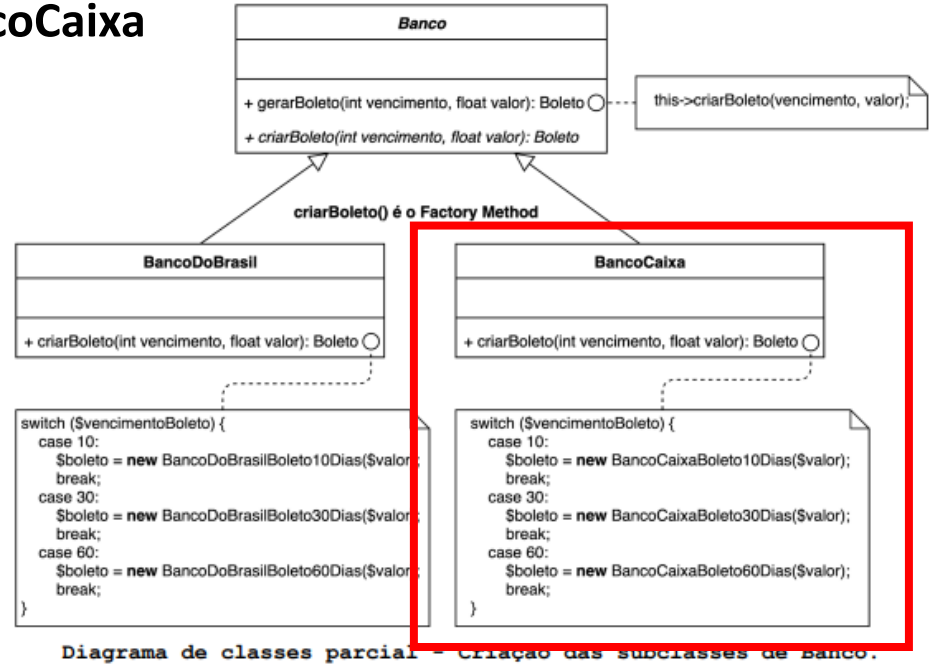


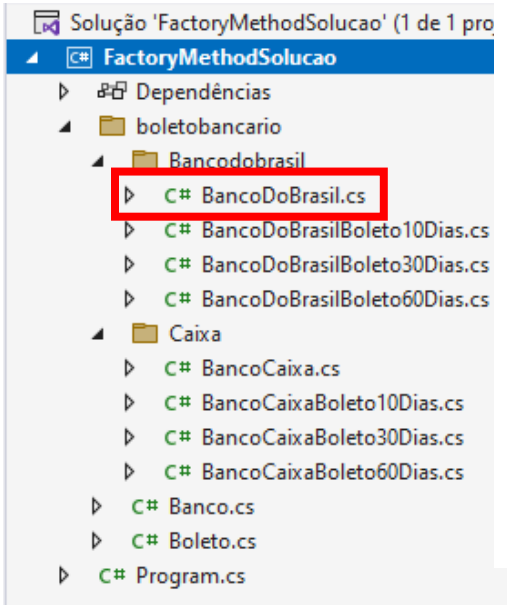
Diagrama de classes parcial – Criação das subclasses de Banco.

- Como é possível observar no diagrama de classes acima o método **criarBoleto()** da a classe **BancoDoBrasil** cria os objetos: BancoDoBrasilBoleto10Dias; BancoDoBrasilBoleto30Dias ou BancoDoBrasilBoleto60Dias.
- Já o método **criarBoleto()** da classe **BancoCaixa** cria os objetos: BancoCaixaBoleto10Dias; BancoCaixaBoleto30Dias ou BancoCaixaBoleto60Dias.

Factory Method

- Agora vamos implementar a classe **BancoDoBrasil** e refatorar a classe **BancoCaixa**
- **Refatore a classe BancoCaixa**

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace FactoryMethodSolucao.boletobancario.Bancodobrasil
{
    1 referência
    public class BancoDoBrasil : Banco
    {
        2 referências
        protected override Boleto criarBoleto(int vencimento, double valor)
        {
            Boleto boleto;
            switch (vencimento)
            {
                case 10:
                    boleto = new BancoDoBrasilBoleto10Dias(valor);
                    break;
                case 30:
                    boleto = new BancoDoBrasilBoleto30Dias(valor);
                    break;
                case 60:
                    boleto = new BancoDoBrasilBoleto60Dias(valor);
                    break;
                default:
                    throw new Exception("Vencimento indisponível");
            }
            return boleto;
        }
    }
}
```



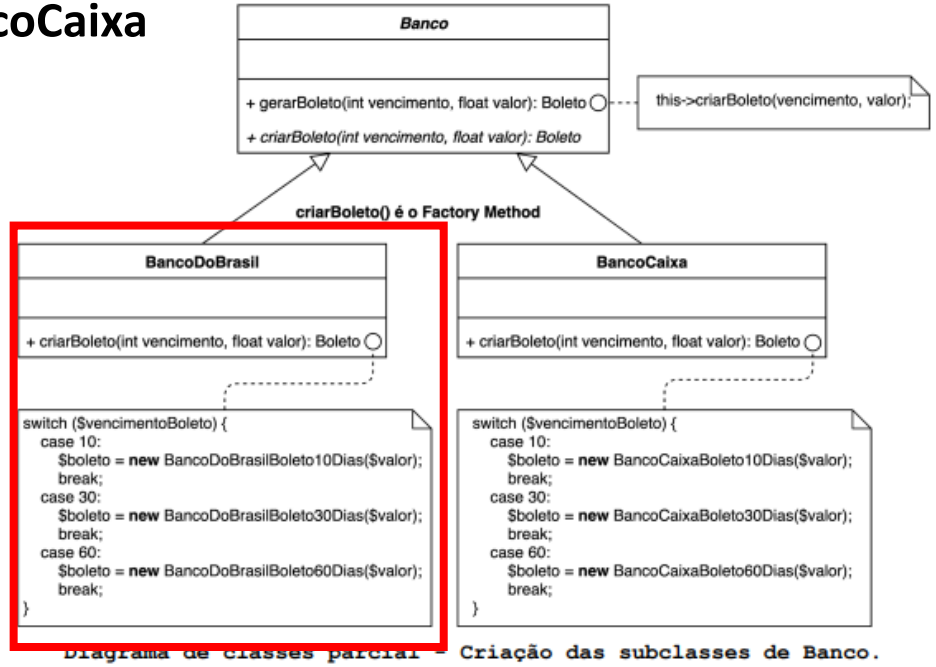


Diagrama de classes parcial – Criação das subclasses de Banco.

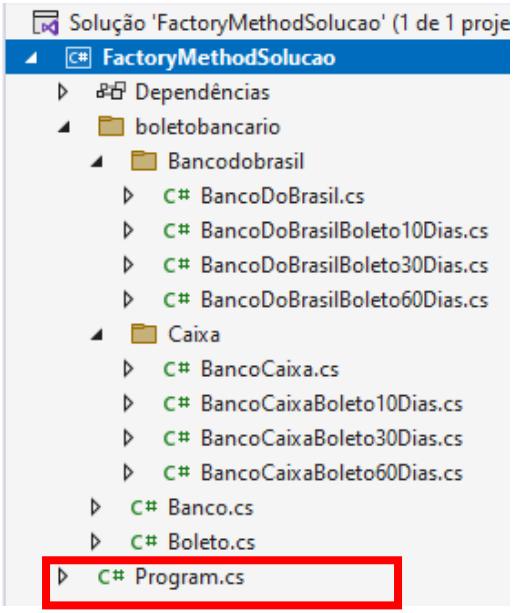
- Como é possível observar no diagrama de classes acima o método **criarBoleto()** da a classe **BancoDoBrasil** cria os objetos: BancoDoBrasilBoleto10Dias; BancoDoBrasilBoleto30Dias ou BancoDoBrasilBoleto60Dias.
- Já o método **criarBoleto()** da classe **BancoCaixa** cria os objetos: BancoCaixaBoleto10Dias; BancoCaixaBoleto30Dias ou BancoCaixaBoleto60Dias.

Factory Method

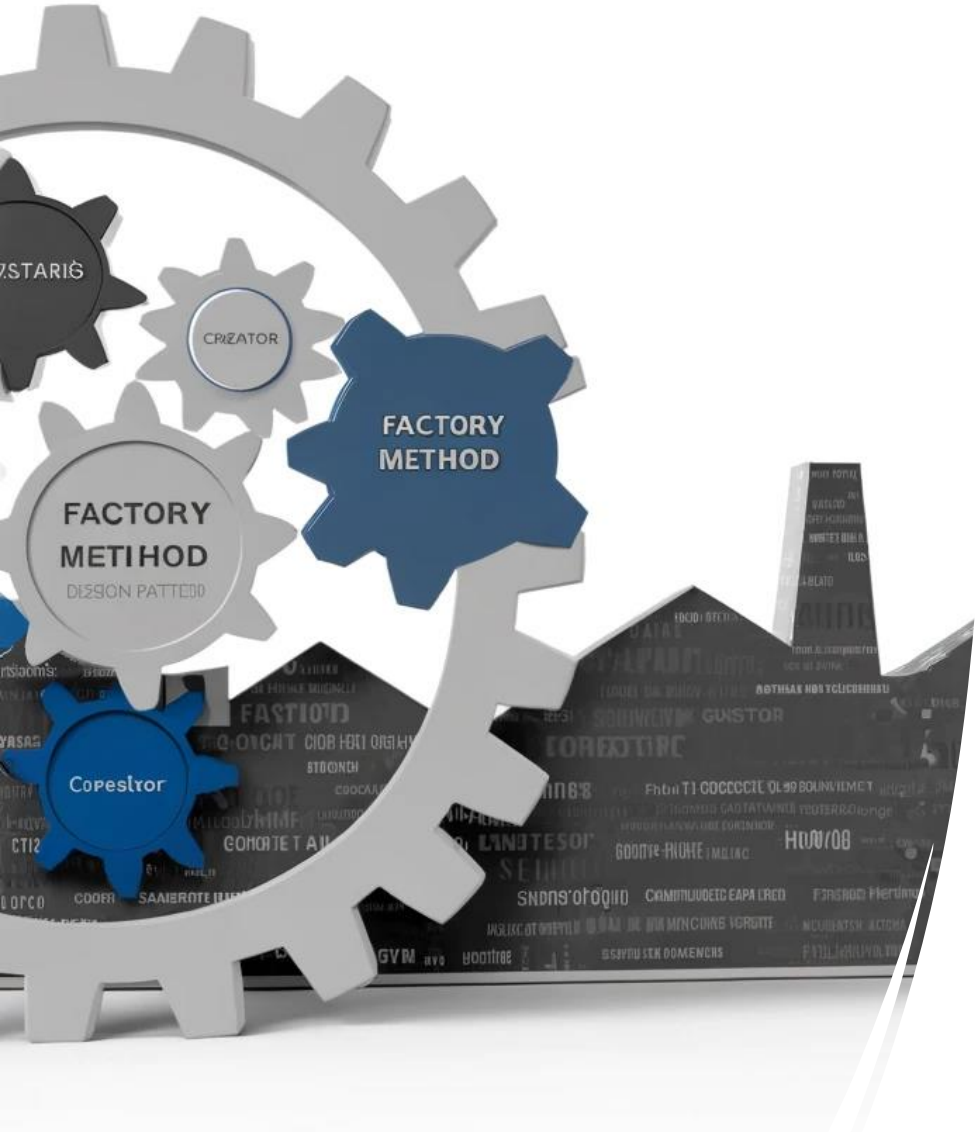
- Atualize a main do seu projeto.

```
1
2   using FactoryMethodSolucao.boletobancario;
3   using FactoryMethodSolucao.boletobancario.Bancodobrasil;
4   using FactoryMethodSolucao.boletobancario.Caixa;
5
6   Console.WriteLine("-------------CAIXA---------------");
7   Banco bancoCaixa = new BancoCaixa();
8   bancoCaixa.gerarBoleto(10, 100);
9   bancoCaixa.gerarBoleto(30, 100);
10  bancoCaixa.gerarBoleto(60, 100);
11
12  Console.WriteLine("-------------BANCO DO BRASIL---------------");
13  Banco bancoBrasil = new BancoDoBrasil();
14  bancoBrasil.gerarBoleto(10, 100);
15  bancoBrasil.gerarBoleto(30, 100);
16  bancoBrasil.gerarBoleto(60, 100);
```

```
-------------CAIXA---------------
Boleto gerado com sucesso no valor de R$100
Valor Juros R$2
Valor Multa R$5
Valor Desconto R$10
-------------------------------------------------
Boleto gerado com sucesso no valor de R$100
Valor Juros R$5
Valor Multa R$10
Valor Desconto R$5
-------------------------------------------------
Boleto gerado com sucesso no valor de R$100
Valor Juros R$10
Valor Multa R$20
Valor Desconto R$0
-------------------------------------------------
-------------BANCO DO BRASIL---------------
Boleto gerado com sucesso no valor de R$100
Valor Juros R$3
Valor Multa R$2
Valor Desconto R$5
-------------------------------------------------
Boleto gerado com sucesso no valor de R$100
Valor Juros R$5
Valor Multa R$5
Valor Desconto R$2
-------------------------------------------------
Boleto gerado com sucesso no valor de R$100
Valor Juros R$10
Valor Multa R$15
Valor Desconto R$0
-------------------------------------------------
```



- Repare em nosso teste (Cliente) que sempre estamos chamando o método **gerarBoleto()** independe se o objeto banco aponta para um objeto da classe BancoCaixa ou BancoDoBrasil.
- ***O que garante para o Cliente que o método gerarBoleto() existe é o fato de que BancoCaixa e BancoDoBrasil são subclasses de Banco.*** O método criarBoleto() que é abstrato na classe Banco sempre irá retornar uma instância de uma subclasse de Boleto. Temos então duas hierarquias paralelas de classes no padrão Factory Method.



Factory Method Consequências

Padrões de Projeto Criacional I

Prof. Me Jefferson Passerini

Factory Method

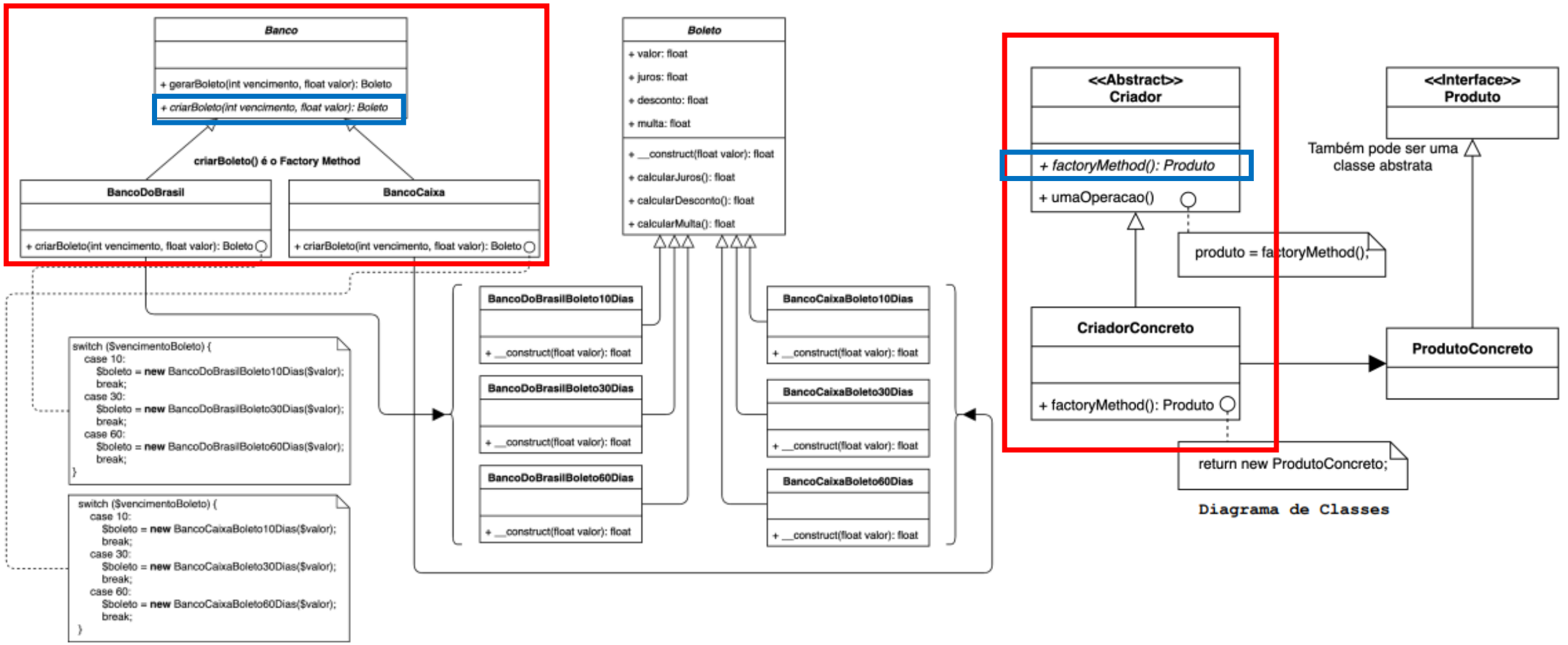


Diagrama de Classes

Diagrama de classes do módulo de cobranças emitindo boletos para dois bancos diferentes por meio do *Factory Method*.

Consequências

- O padrão Factory Method elimina o forte acoplamento entre classes concretas. O código lida apenas com a interface do Produto, portanto, ele pode funcionar com qualquer classe ProdutoConcreto definida no sistema.

- Uma desvantagem potencial do Factory Method é que os clientes podem ter que subclassificar a classe Criador apenas para criar um objeto produtoConcreto específico.
  - Subclassificar é bom quando o cliente precisa subclassificar a classe Creator de qualquer maneira, mas, caso contrário, o cliente agora deve lidar com outro problema.

- Criar objetos dentro de uma classe com um método factoryMethod() é sempre mais flexível do que criar um objeto diretamente. O padrão Factory Method fornece às subclasses um gancho (hook) para fornecer uma versão diferente de um objeto.

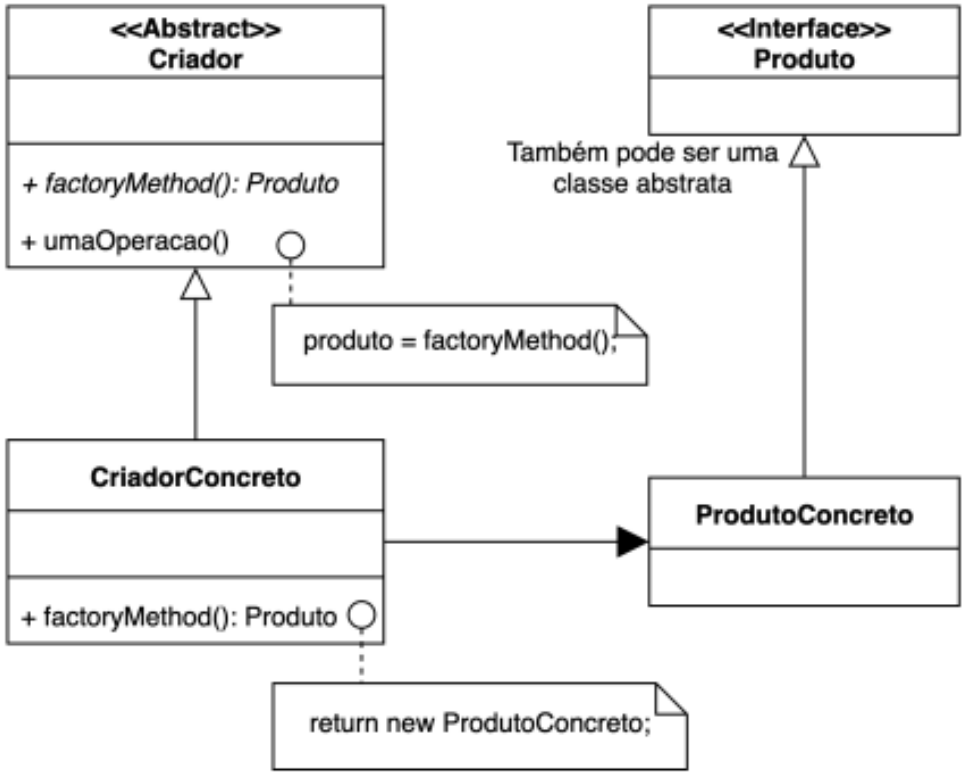


Diagrama de Classes

Consequências

- No exemplo que consideramos até agora, o método fábrica criarBoleto() é chamado apenas pelos criadores concretos. Mas isso não precisa ser sempre assim. Os clientes podem achar os métodos de fábrica úteis, e os utilizar de forma direta, especialmente no caso de hierarquias de classes paralelas.

- Hierarquias de classe paralelas resultam quando uma classe delega algumas de suas responsabilidades a uma outra classe separada.

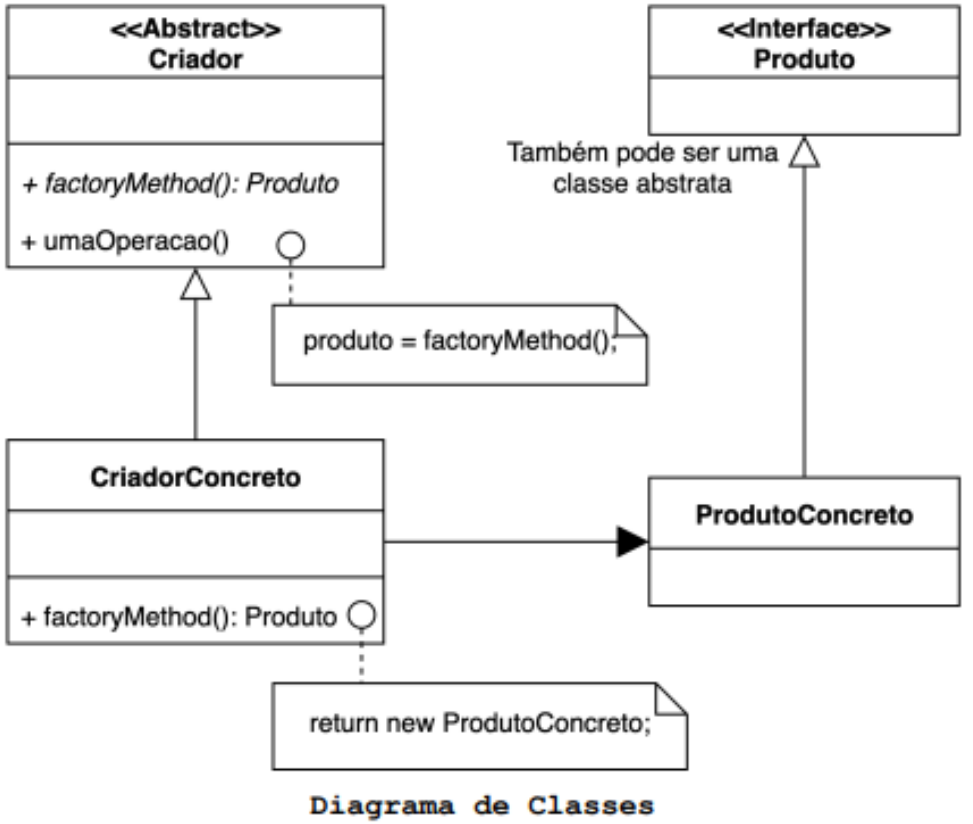


Diagrama de Classes