



Abstract Factory

Padrões de Projeto Criacional I

Prof. Me Jefferson Passerini



**ABSTRACT
FACTORY**

O padrão **Abstract Factory** fornece uma interface para a criar famílias de objetos relacionados ou dependentes sem especificar suas classes concretas.

Aplicabilidade (Quando utilizar?)

- Quando um sistema deve ser independente de como seus produtos são criados, compostos ou representados.
- Quando um sistema deve ser configurado com uma dentre múltiplas famílias de produtos.
- Quando uma família de objetos relacionados foi projetada para ser usada em conjunto, e é necessário impor essa restrição.
- Quando se deseja fornecer uma biblioteca de produtos e se deseja revelar para o cliente apenas suas interfaces, e não suas implementações.

Componentes

- **FabricaConcreta:**

- Implementa a interface declarada por AbstractFactory
- Criam as diferentes famílias de produtos.
- Para criar um produto, o Cliente usa uma dessas fabricas concretas, então ele nunca precisar utilizar fabricas concretas diretamente.

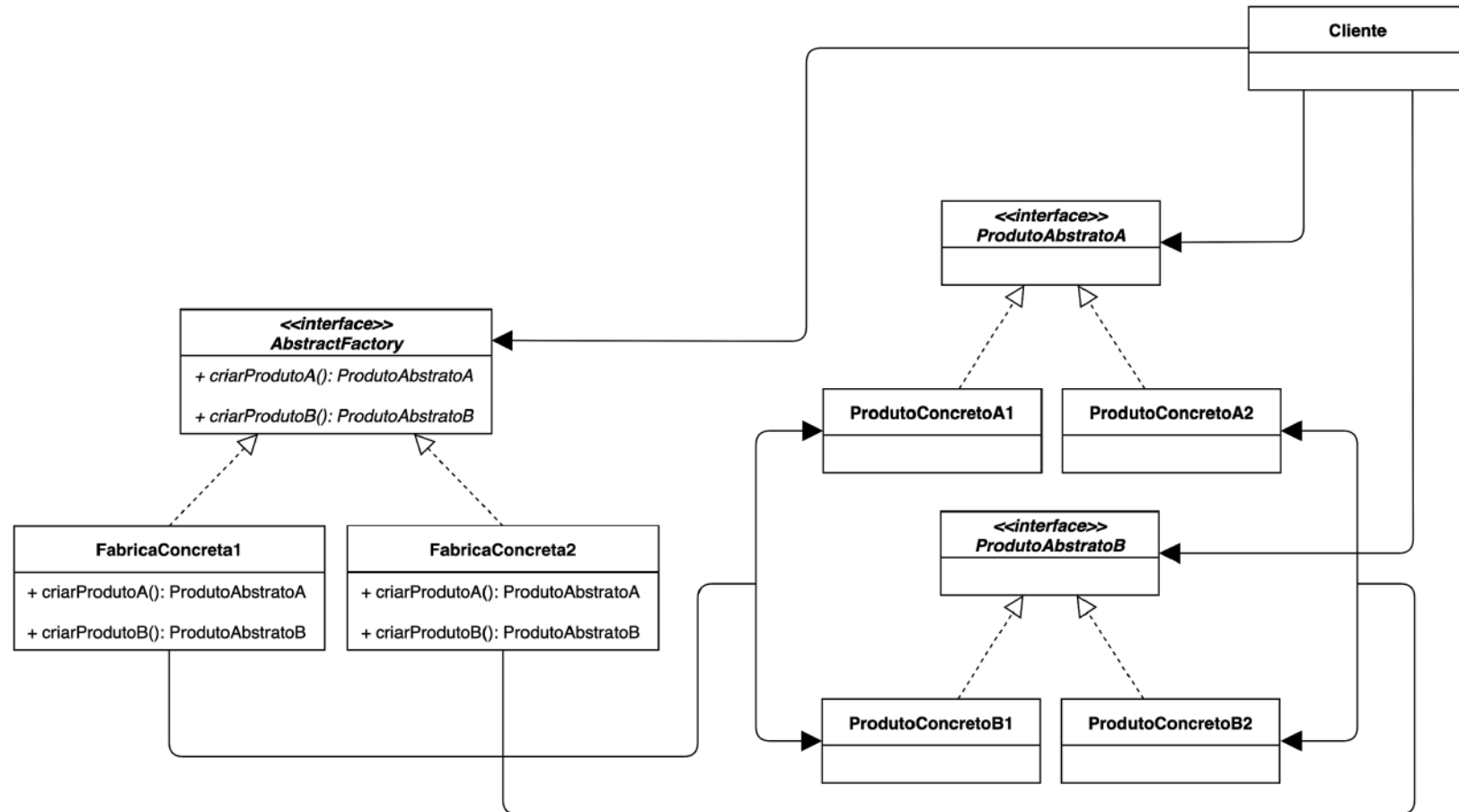


Diagrama de Classes

Componentes

- **ProdutoAbstrato:**

- Define a interface para um determinado tipo de produto.

- **ProdutoConcreto:**

- Implementa a interface ProdutoAbstrato.
- São os integrantes de uma família de produtos.
- É criado por uma FabricaConcreta.

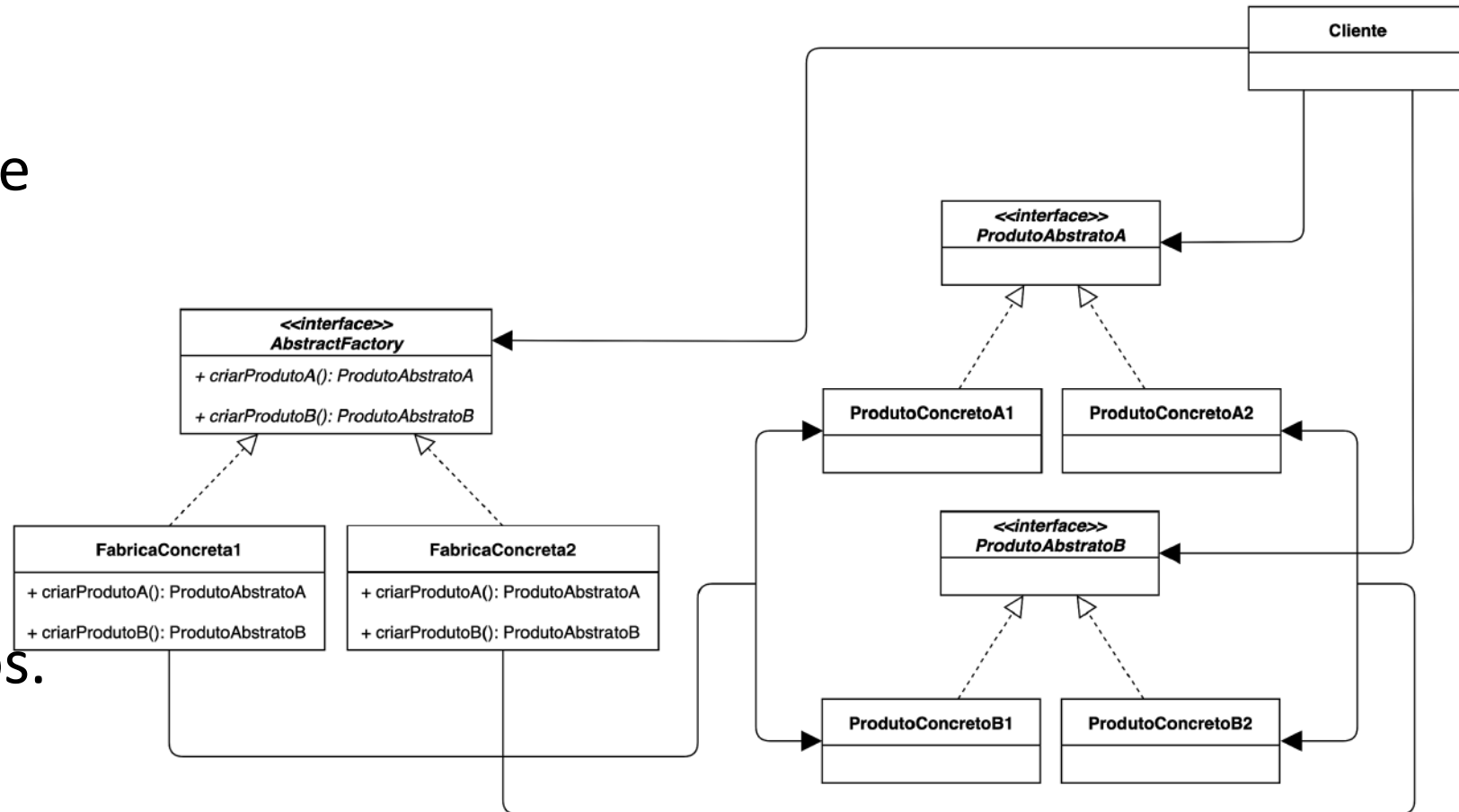


Diagrama de Classes

Componentes

- **Cliente:**

- Usa apenas interfaces declaradas pelas as classes AbstractFactory e ProdutoAbstrato, e é composto em tempo de execução por fábricas e produtos concretos.

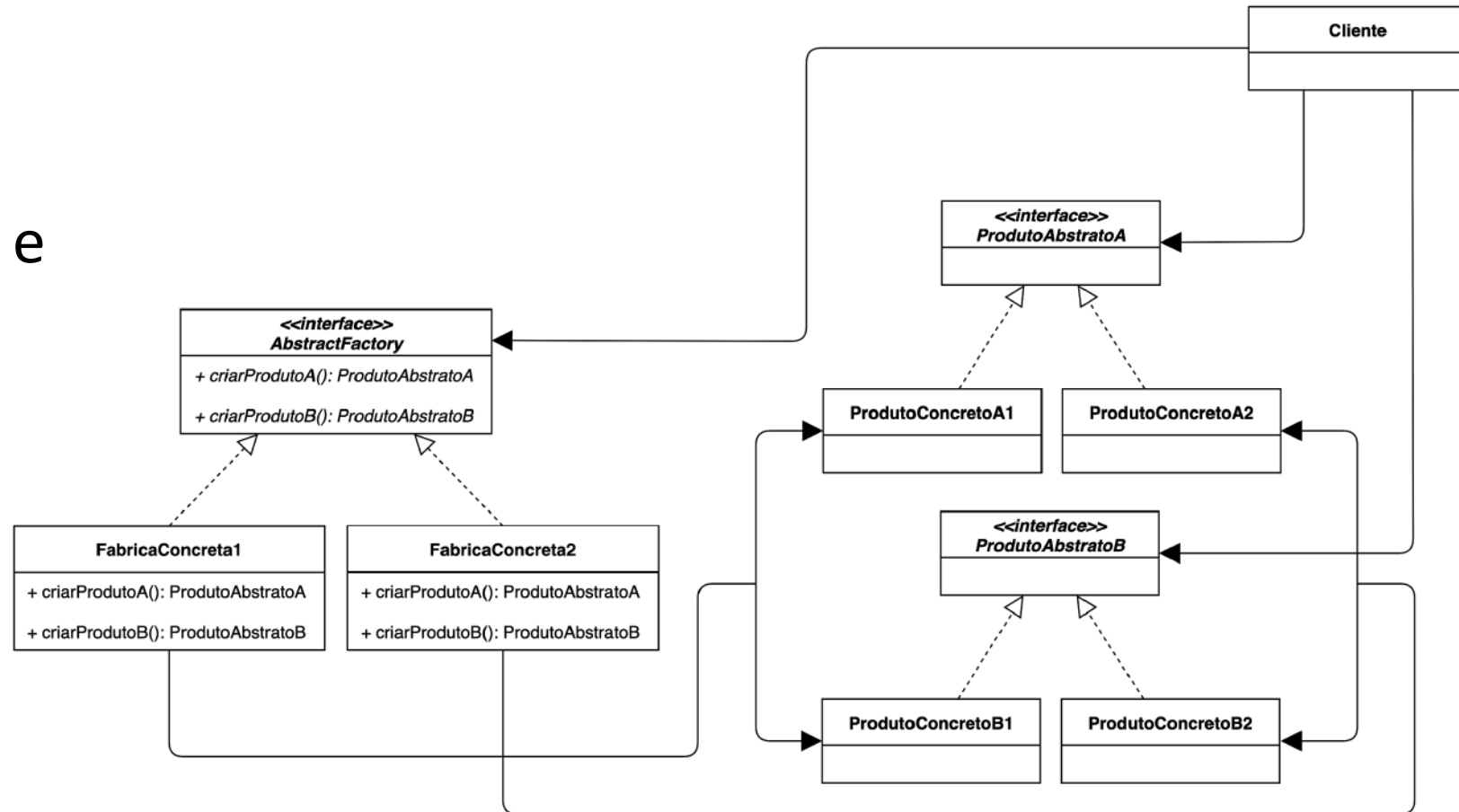
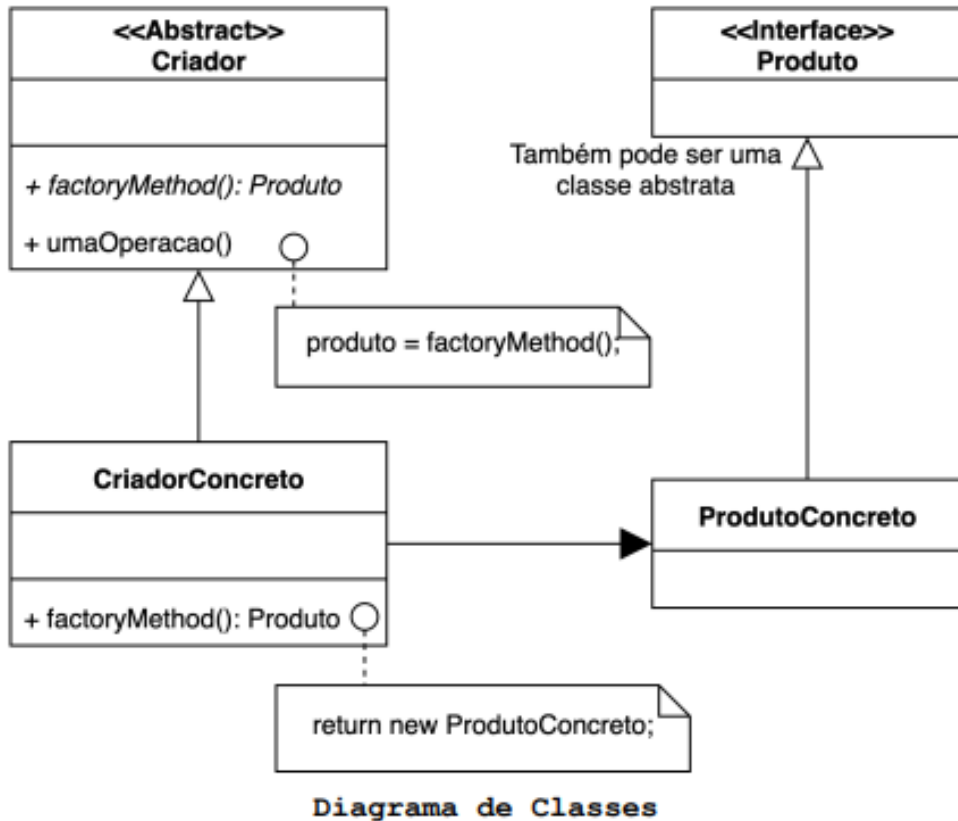


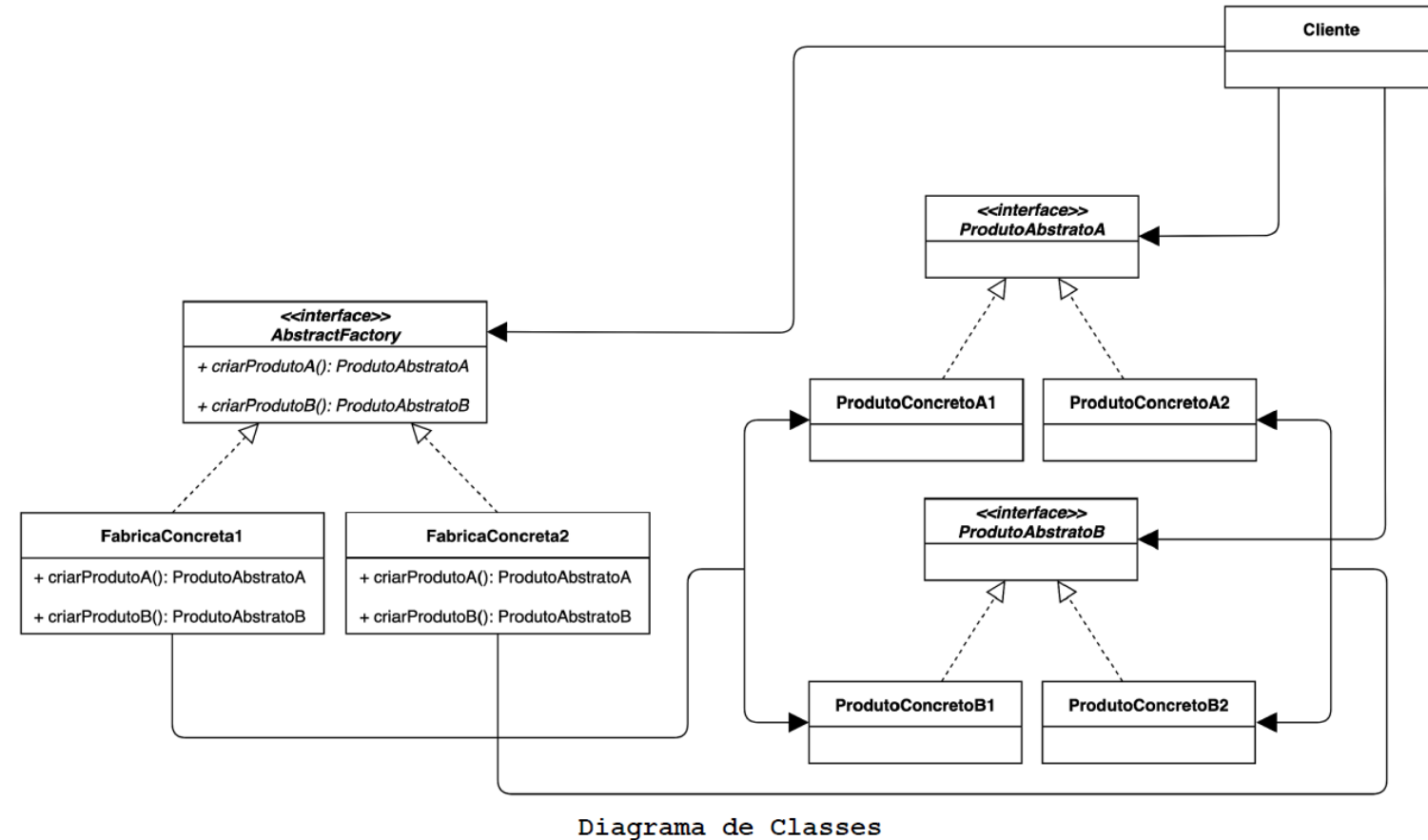
Diagrama de Classes

AbstractFactory x FactoryMethod

Padrões de Projetos Criacional – Abstract Factory



Factorymethod
Herança



AbstractFactory
Composição

Motivação (Por que utilizar?)

- O padrão Abstract Factory utiliza a composição para expandir suas funcionalidades dependendo apenas de supertipos e não de classes concretas, isso isola a criação de objetos de seu uso e cria famílias de objetos relacionados que são necessários para compor o objeto que os utiliza.
- Isto permite que novos tipos derivados sejam introduzidos sem qualquer alteração ao código que utiliza a classe base.
- As informações acima podem ser muito abstratas e difíceis de entender, então vamos por partes utilizando um exemplo para ilustrar.

Motivação (Por que utilizar?)

- Seguindo com o exemplo que utilizamos no padrão Factory Method continue considerando o cenário onde precisamos implementar um módulo de cobranças que gera boletos emitidos por 2 bancos diferentes (Caixa e Banco do Brasil);
- ainda considerando que novos bancos podem ser inseridos ao longo do tempo. Cada banco têm sua própria maneira de implementar os cálculos de juros, desconto e multa, esses cálculos devem ser objetos.

Motivação (Por que utilizar?)


- Para simplificar a implementação e focar no conceito do padrão Abstract Factory neste exemplo vamos remover o conceito de diferentes vencimentos 10, 30 e 60 dias que tínhamos no exemplo do padrão Factory Method.

Motivação (Por que utilizar?)

- Vamos extrair os trechos da definição para que possamos entendê-lo por partes:
 - Trecho1 – “O padrão Abstract Factory utiliza a composição para expandir suas funcionalidades dependendo apenas de supertipos e não de classes concretas”
- Deste trecho podemos concluir que precisaremos de supertipos, portanto um Boleto dependerá de supertipos, não de objetos concretos.

Motivação (Por que utilizar?)

- Vamos extrair os trechos da definição para que possamos entendê-lo por partes:
 - Trecho2 – “Isso isola a criação de objetos de seu uso e cria famílias de objetos relacionados que são necessários para compor um objeto que os utiliza.”
- Sabemos que temos 3 tipos de objetos que se relaciona para compor um boleto, são eles: juros, desconto e multa. Portanto teremos 3 supertipos que irão compor um boleto.



Abstract Factory

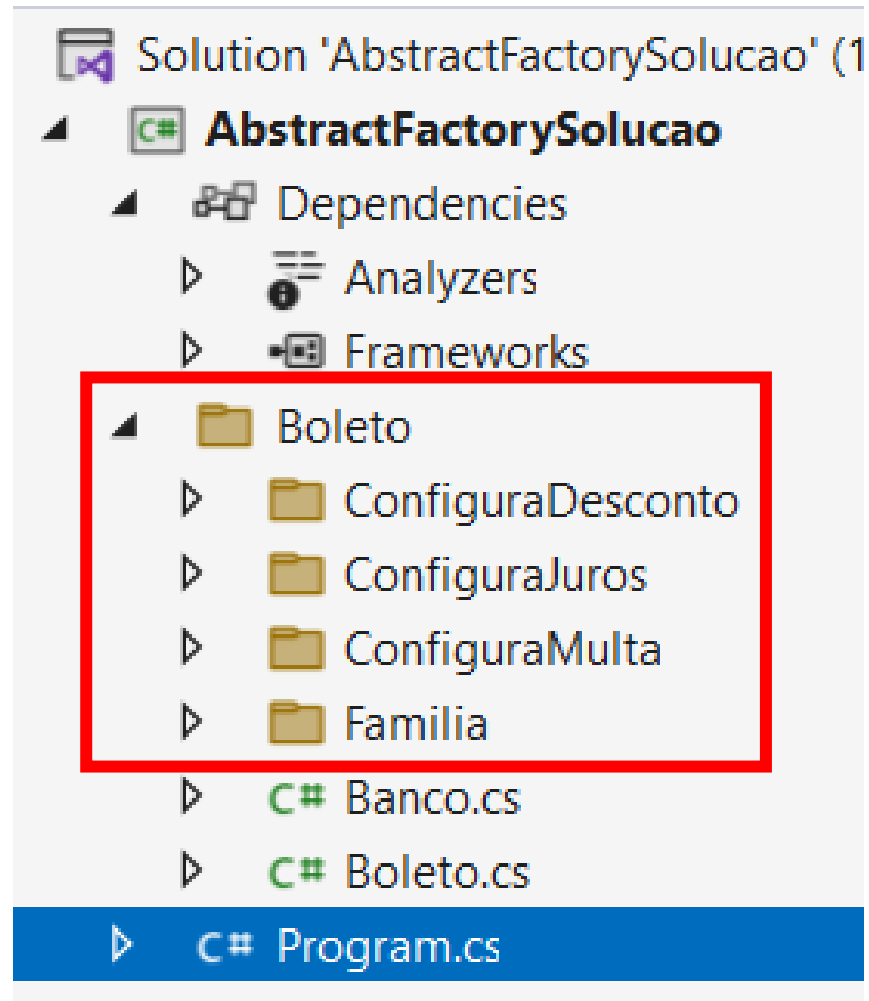
Implementação em C#

Padrões de Projeto Criacional I

Prof. Me Jefferson Passerini



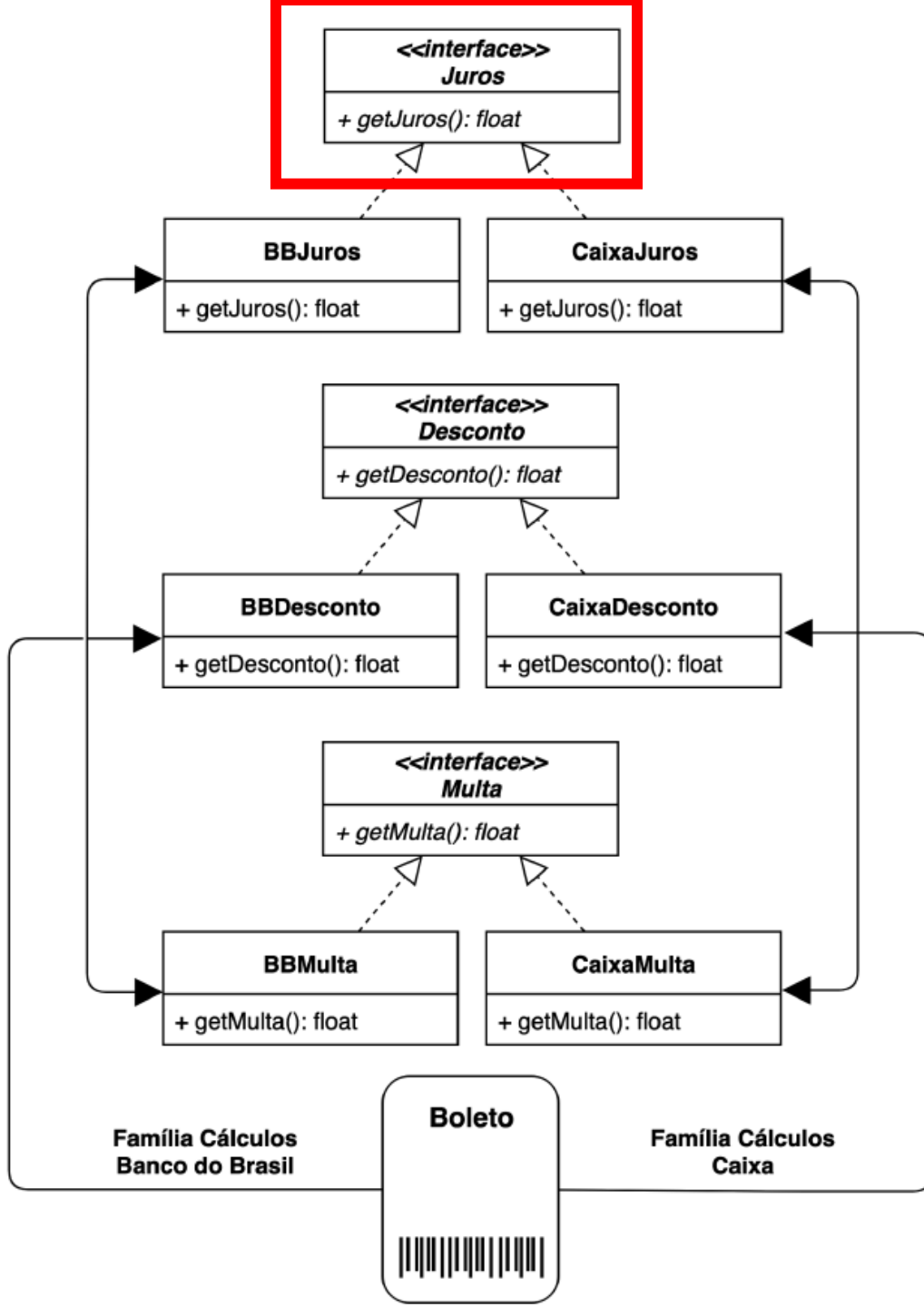
**ABSTRACT
FACTORY**



C#

- Para iniciarmos a implementação crie o projeto **AbstractFactorySolucao**.
- E a seguir crie a seguinte estrutura de pastas.

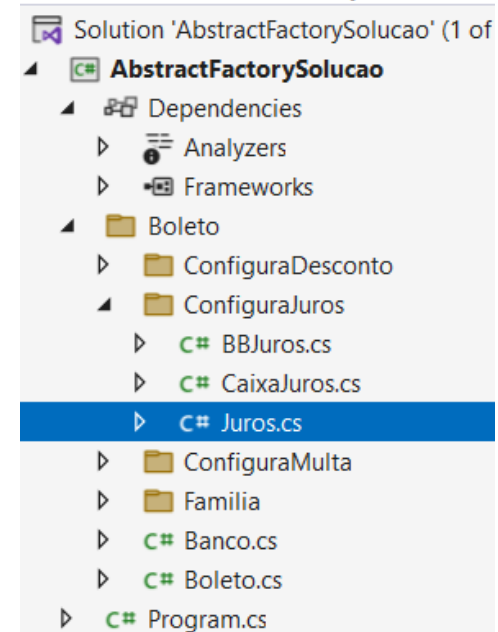
C# - Implementando os 3 Supertipos



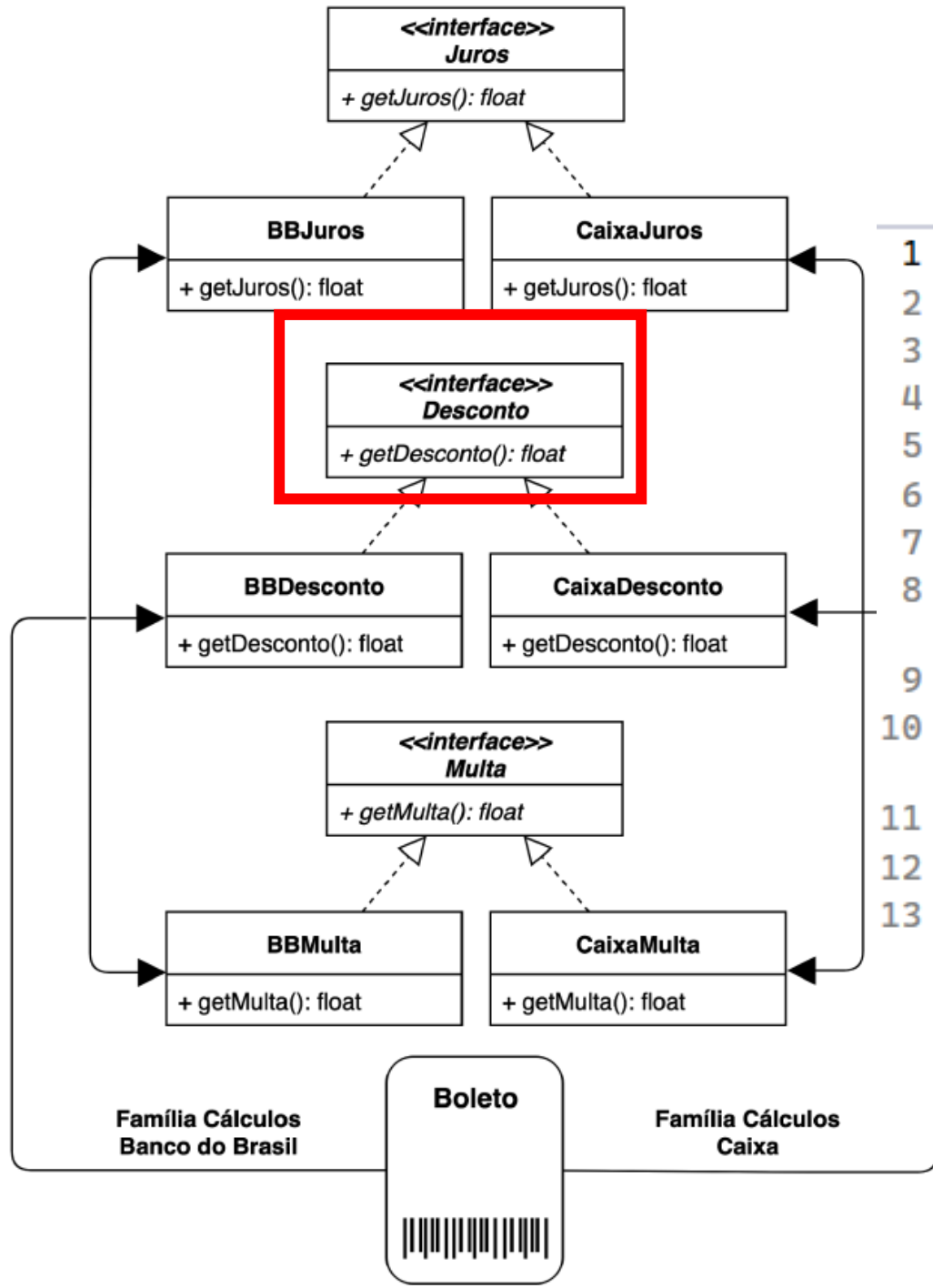
```

1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.Threading.Tasks;
6
7  namespace AbstractFactorySolucao.Boleto.ConfiguraJuros
8  {
9      6 references
10     public interface Juros
11     {
12         3 references
13         public double getJuros();
14     }
15 }

```



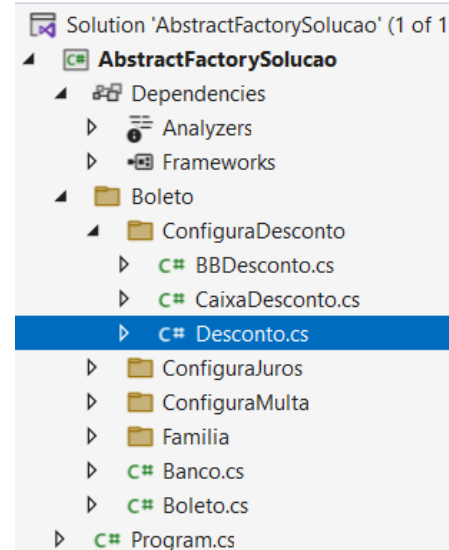
C# - Implementando os 3 Supertipos



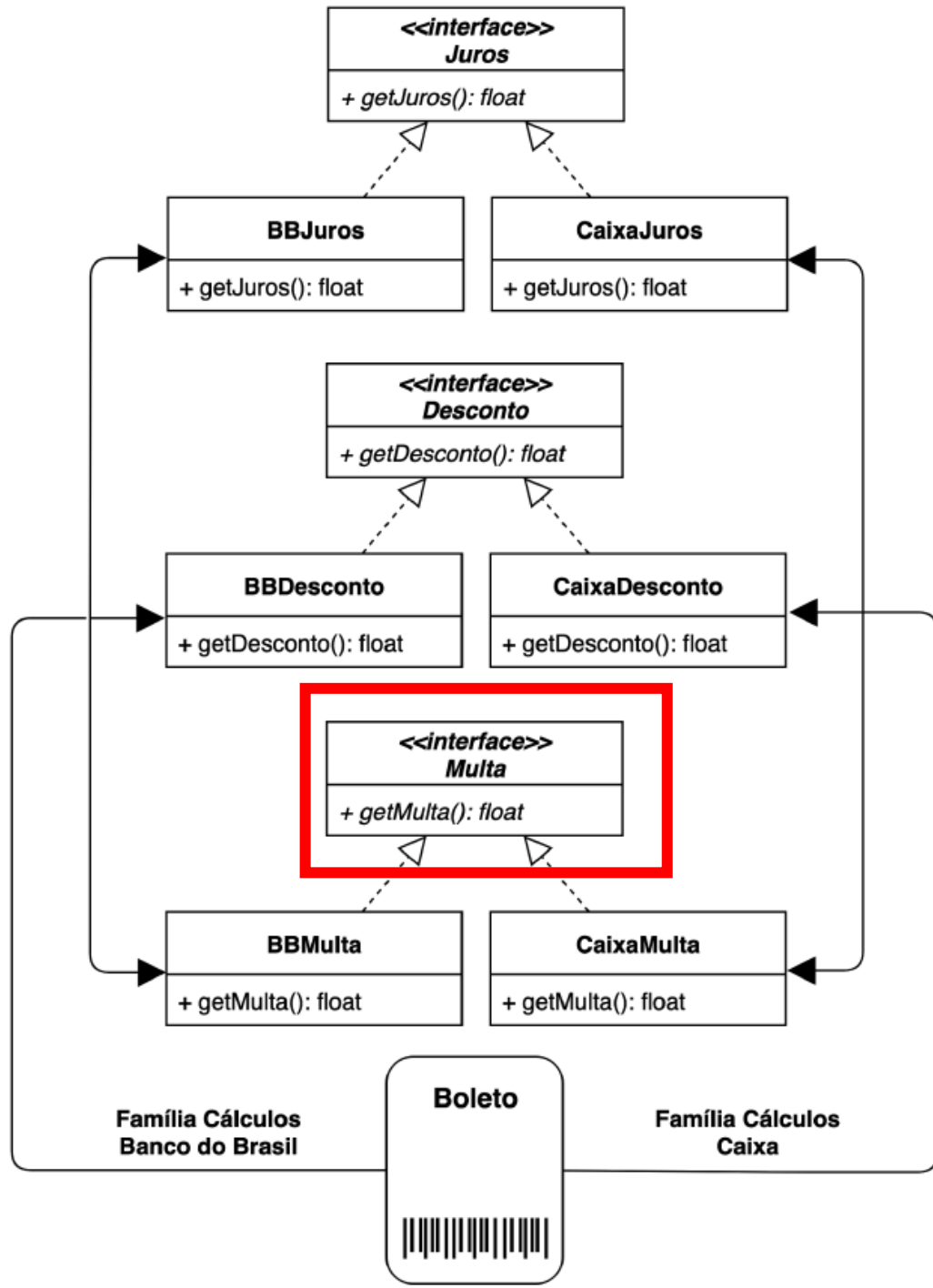
```

1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.Threading.Tasks;
6
7  namespace AbstractFactorySolucao.Boleto.ConfiguraDesconto
8  {
9
10     6 references
11     public interface Desconto
12     {
13         3 references
14         public double getDesconto();
15     }
16 }

```

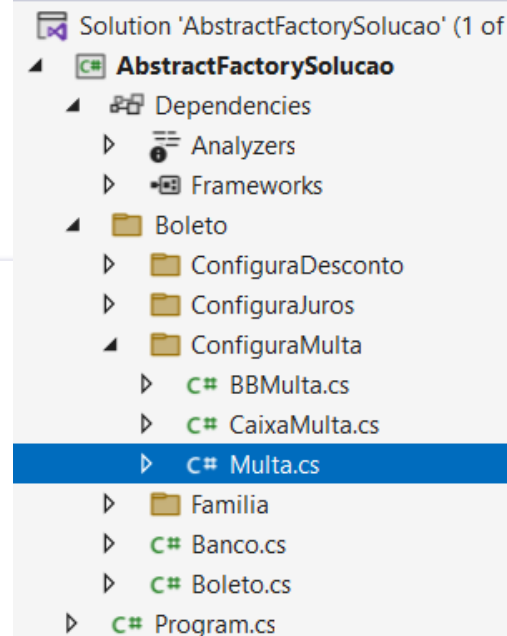


C# - Implementando os 3 Supertipos

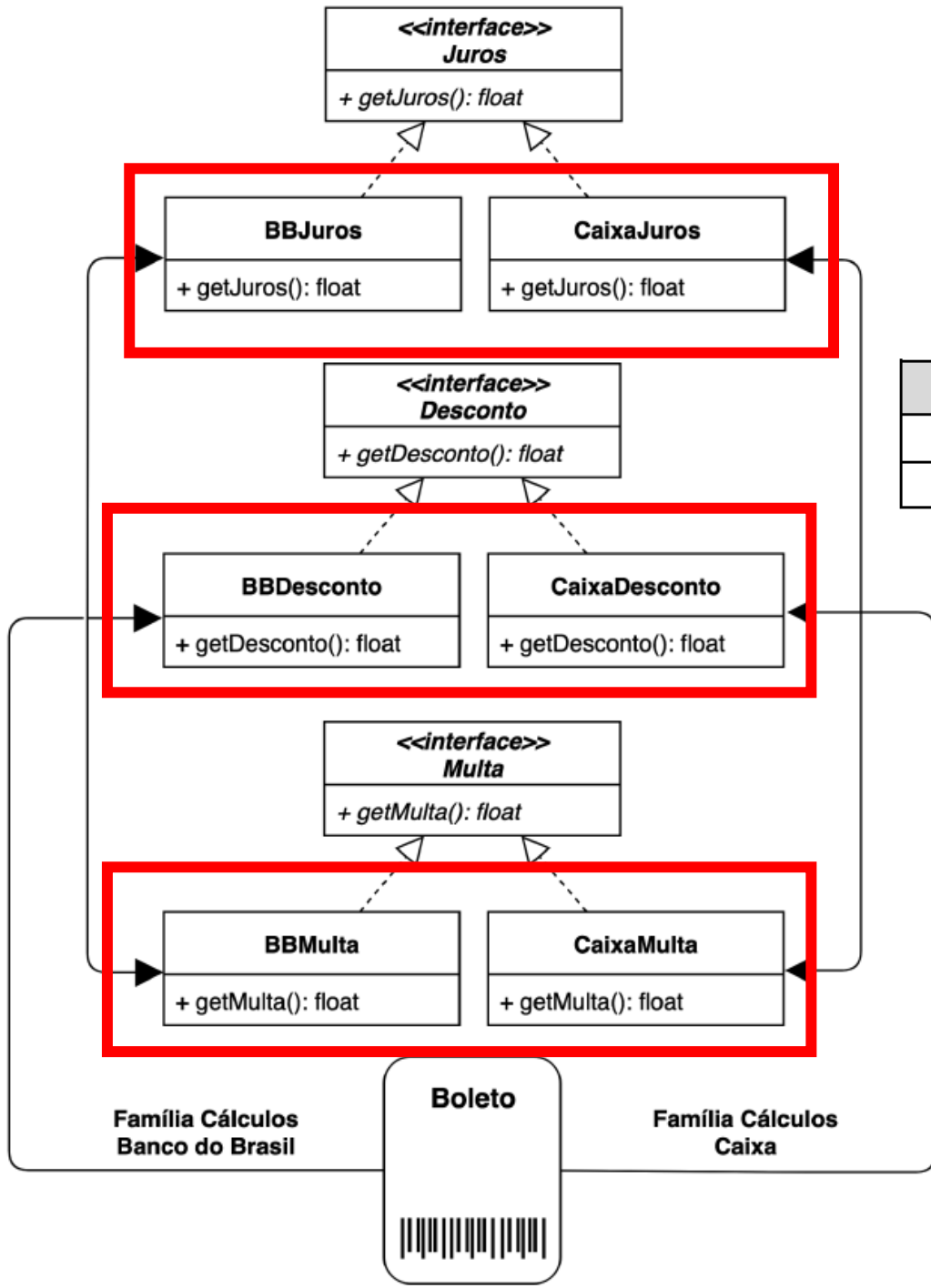


```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
```

```
namespace AbstractFactorySolucao.Boleto.ConfiguraMulta
{
    6 references
    public interface Multa
    {
        3 references
        public double getMulta();
    }
}
```



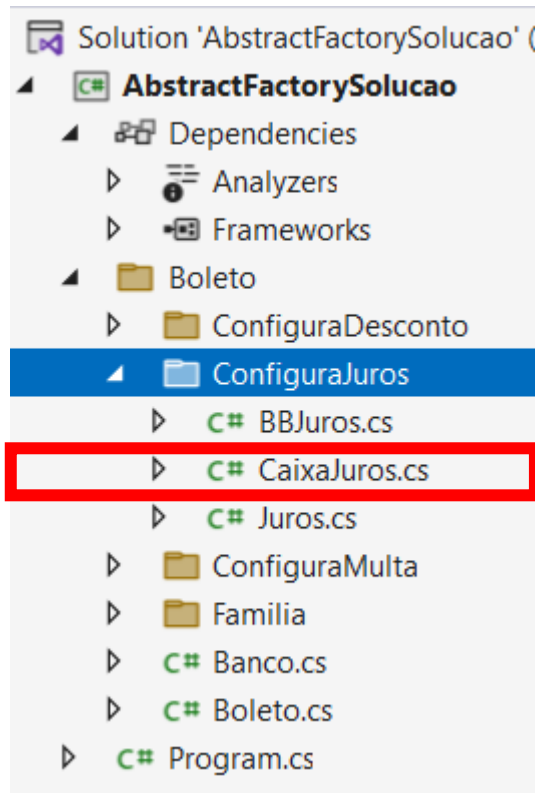
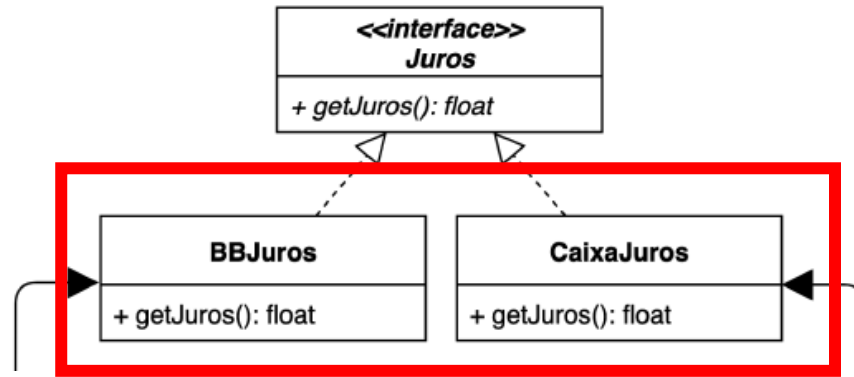
C# - Agora vamos criar os tipos concretos para cada um dos supertipos



Banco	Juros	Desconto	Multa
Caixa	2%	10%	5%
Banco Do Brasil	3%	5%	2%

Cálculos de cada banco incidindo sobre o valor do boleto.

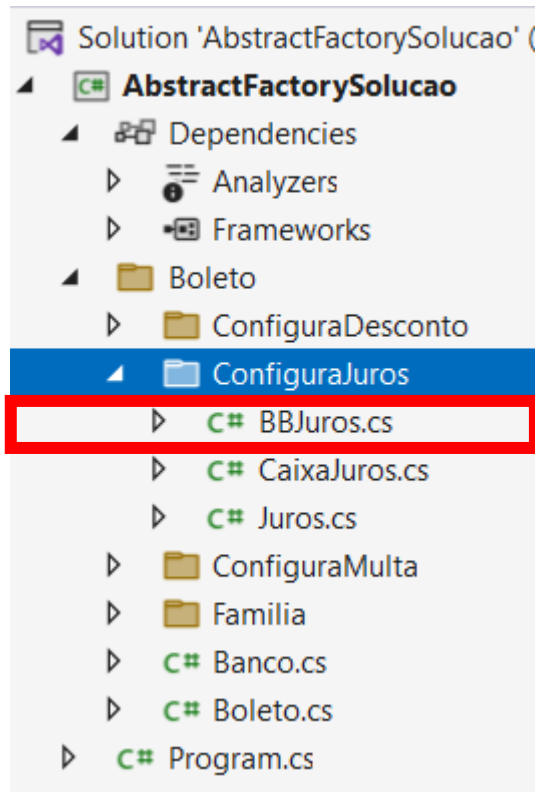
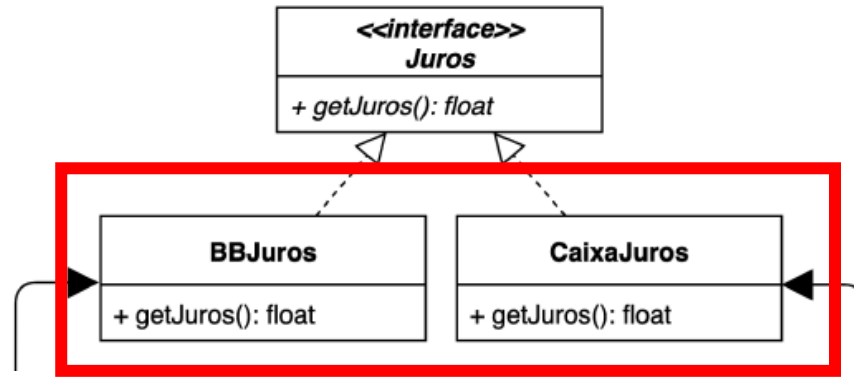
C# - Agora vamos criar os tipos concretos para cada um dos supertipos



```

1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.Threading.Tasks;
6
7  namespace AbstractFactorySolucao.Boleto.ConfiguraJuros
8  {
9      1 reference
      internal class CaixaJuros : Juros
10     {
11         2 references
12         public double getJuros()
13         {
14             return 0.02;
15         }
16     }
  
```

C# - Agora vamos criar os tipos concretos para cada um dos supertipos

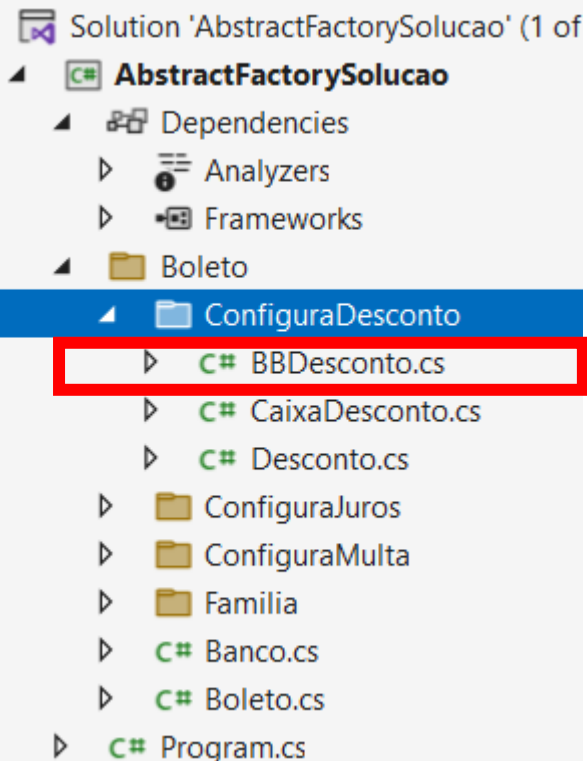
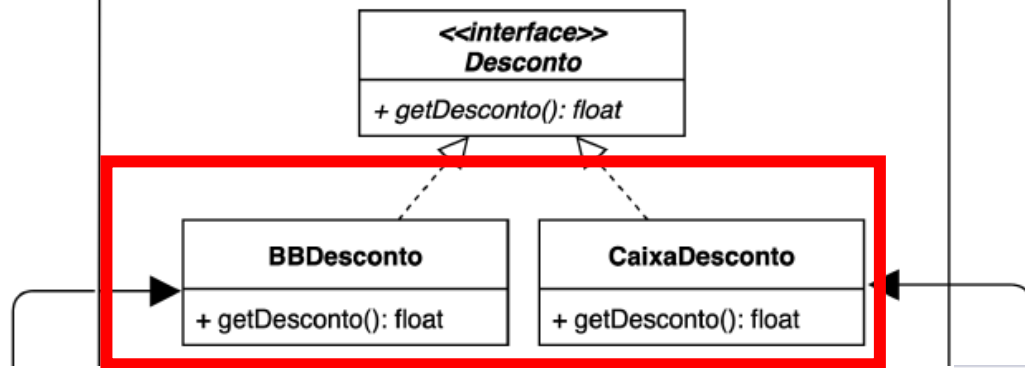


```

1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.Threading.Tasks;
6
7  namespace AbstractFactorySolucao.Boleto.ConfiguraJuros
8  {
9      1 reference
10     internal class BBJuros : Juros
11     {
12         2 references
13         public double getJuros()
14         {
15             return 0.03;
16         }
17     }
18 }

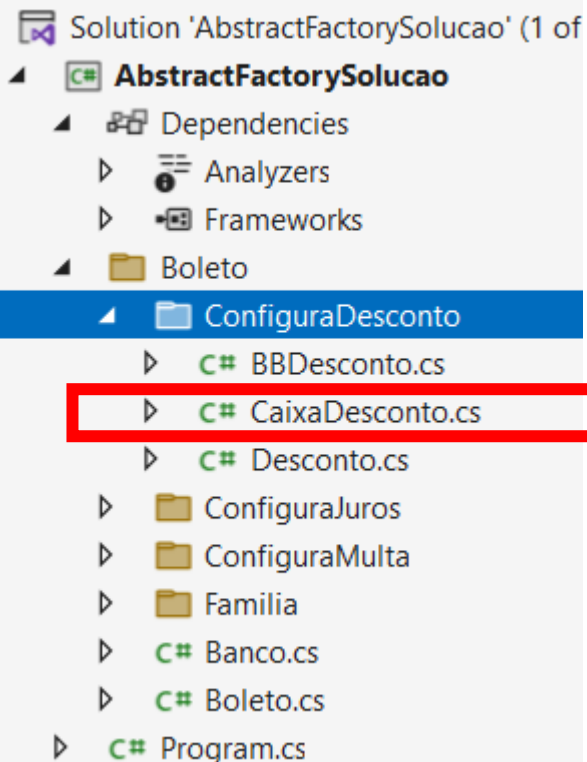
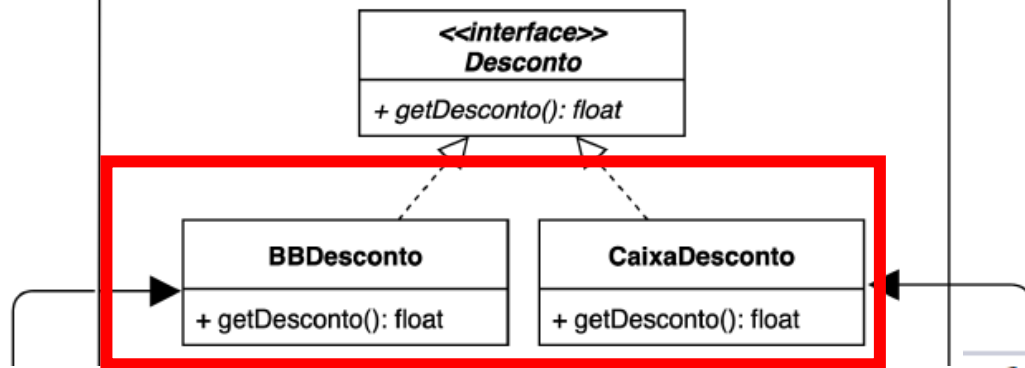
```

C# - Agora vamos criar os tipos concretos para cada um dos supertipos



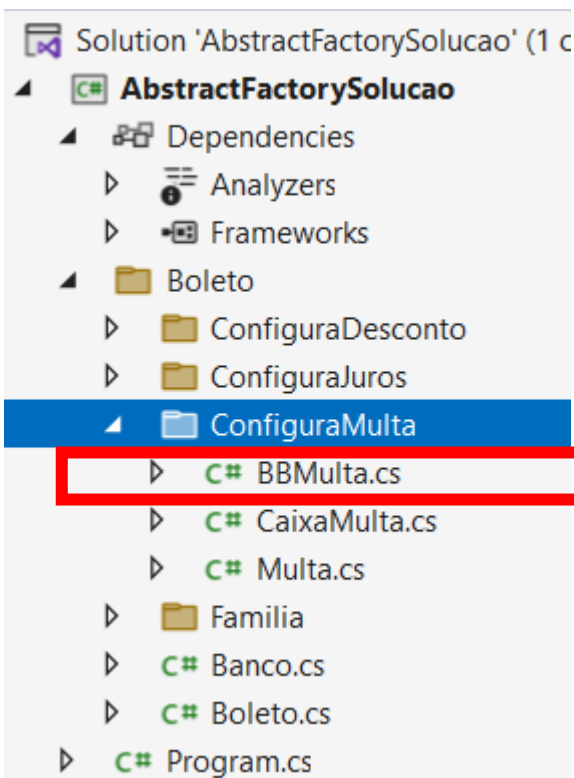
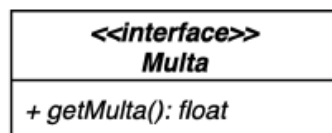
```
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5 using System.Threading.Tasks;
6
7 namespace AbstractFactorySolucao.Boleto.ConfiguraDesconto
8 {
9     1 reference
10     internal class BBDesconto : Desconto
11     {
12         2 references
13         public double getDesconto()
14         {
15             return 0.05;
16         }
17     }
18 }
```

C# - Agora vamos criar os tipos concretos para cada um dos supertipos



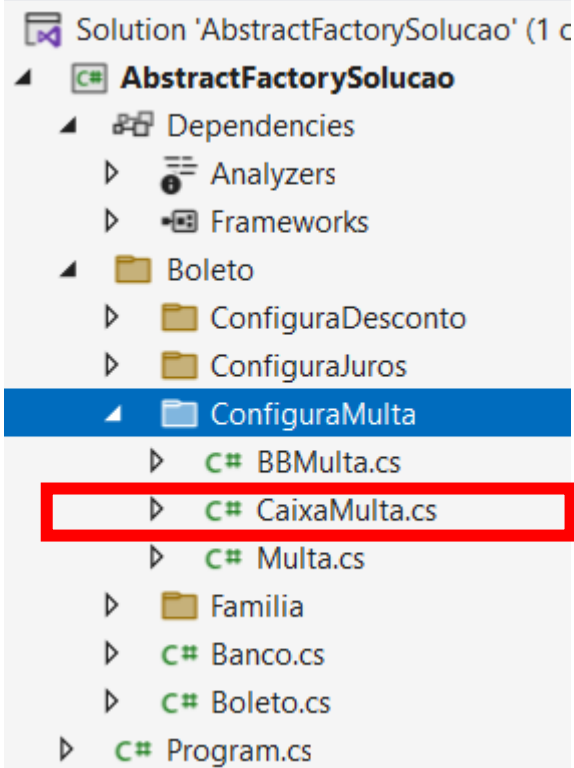
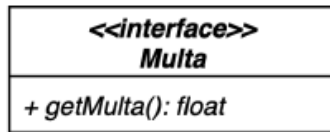
```
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5 using System.Threading.Tasks;
6
7 namespace AbstractFactorySolucao.Boleto.ConfiguraDesconto
8 {
9     1 reference
10     internal class CaixaDesconto : Desconto
11     {
12         2 references
13         public double getDesconto()
14         {
15             return 0.1;
16         }
17     }
18 }
```

C# - Agora vamos criar os tipos concretos para cada um dos supertipos



```
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5 using System.Threading.Tasks;
6
7 namespace AbstractFactorySolucao.Boleto.ConfiguraMulta
8 {
9     1 reference
10     internal class BBMulta : Multa
11     {
12         2 references
13         public double getMulta()
14         {
15             return 0.02;
16         }
17     }
18 }
```

C# - Agora vamos criar os tipos concretos para cada um dos supertipos

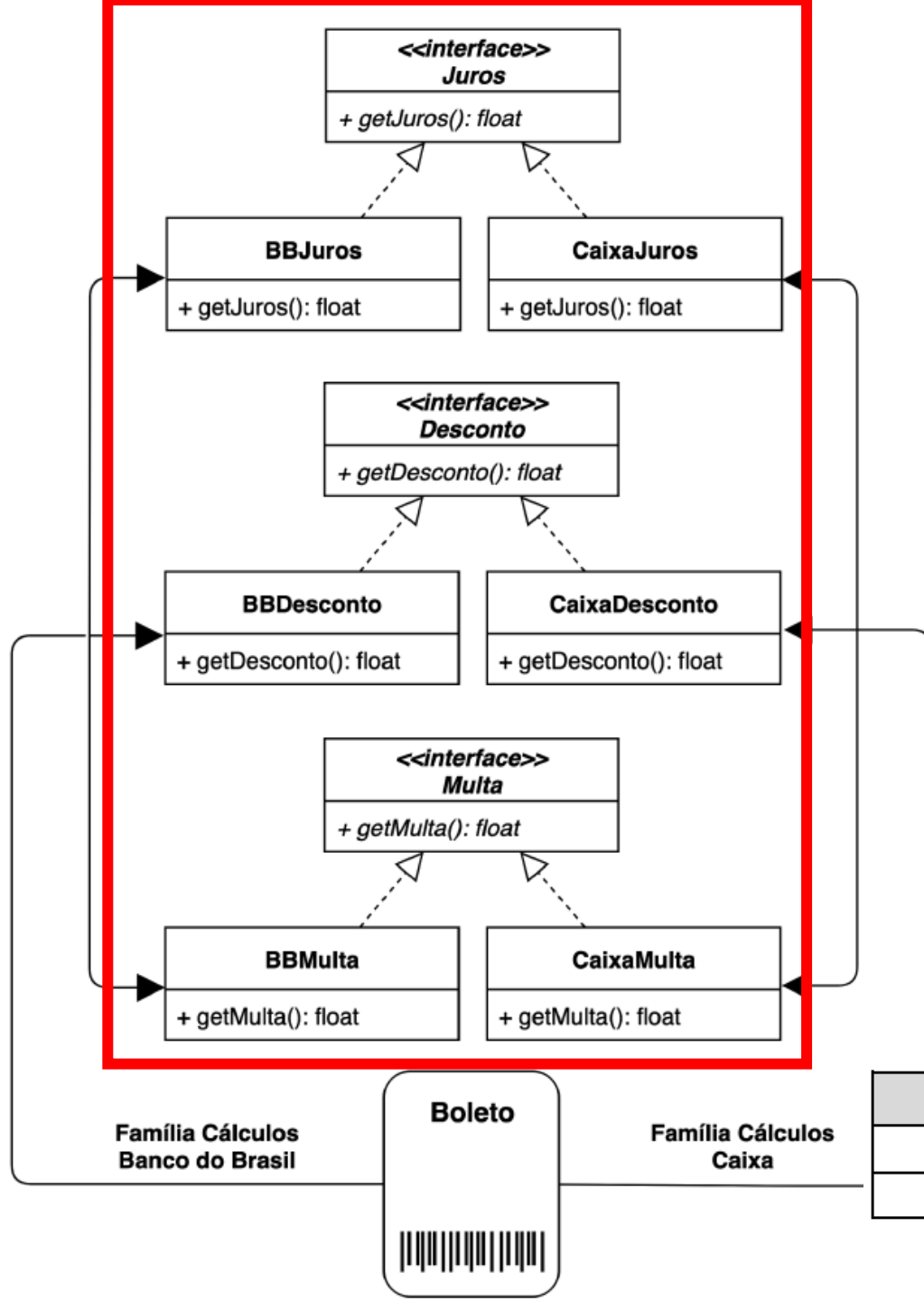


```
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5 using System.Threading.Tasks;
6
7 namespace AbstractFactorySolucao.Boleto.ConfiguraMulta
8 {
9     1 reference
10     public class CaixaMulta : Multa
11     {
12         2 references
13         public double getMulta()
14         {
15             return 0.05;
16         }
17     }
18 }
```


C# - Agora vamos criar os tipos concretos para cada um dos supertipos

Nesta etapa já temos os tipos concretos de cada um dos supertipos criados

E precisamos implementar as Fábricas de famílias para os boletos.



Banco	Juros	Desconto	Multa
Caixa	2%	10%	5%
Banco Do Brasil	3%	5%	2%

Cálculos de cada banco incidindo sobre o valor do boleto.

Sabemos que um boleto precisa de um cálculo de juros, desconto e multa.

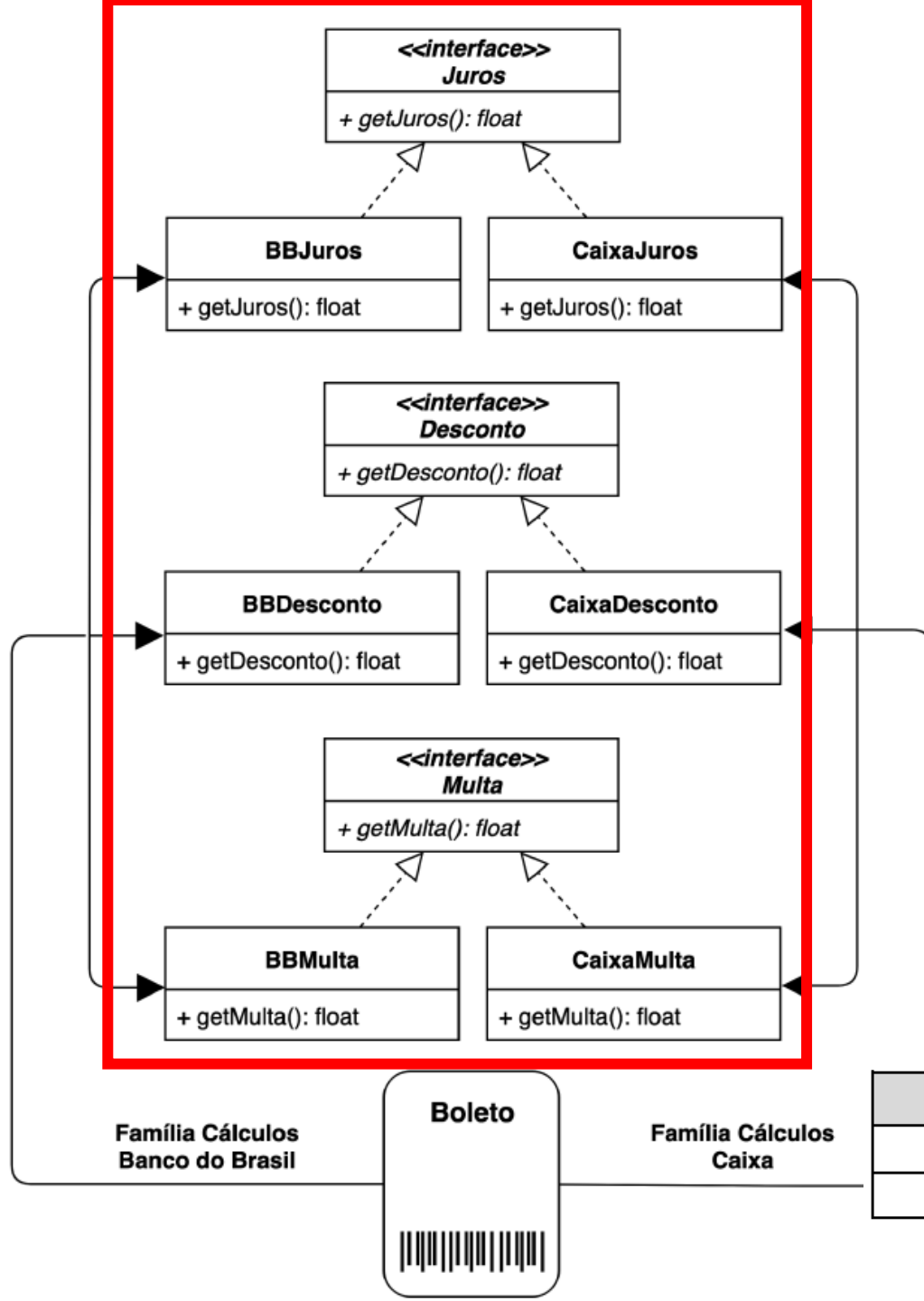
Esses objetos combinados formam uma família de objetos:

Família de objetos = 1 obj. juros + 1 obj. desconto + 1 obj. Multa

Deste modo em nosso exemplo temos:

1 família de cálculos Caixa

1 família de cálculos BB



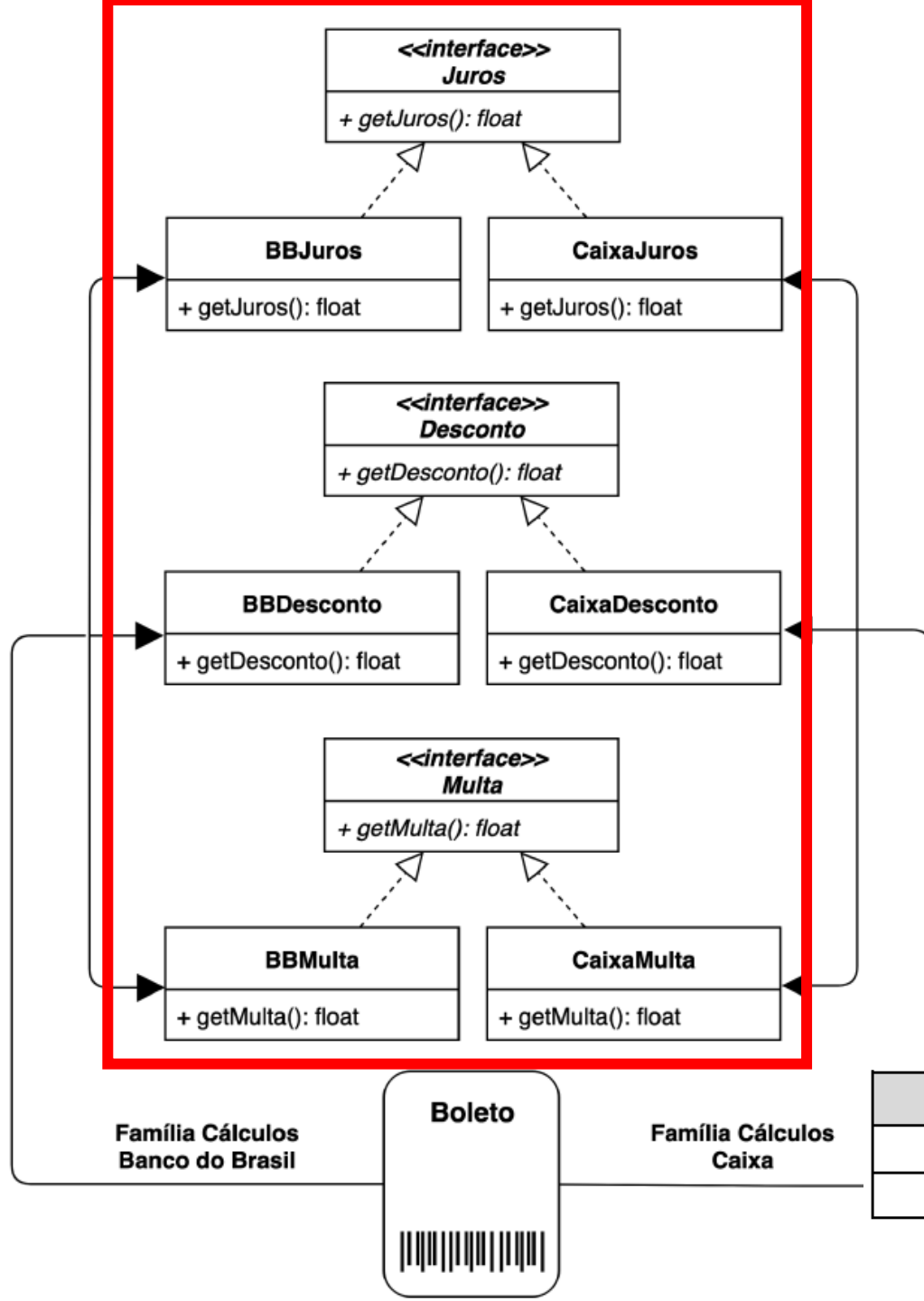
Banco	Juros	Desconto	Multa
Caixa	2%	10%	5%
Banco Do Brasil	3%	5%	2%

Cálculos de cada banco incidindo sobre o valor do boleto.

Um boleto deve ser configurado a partir de uma família de objetos relacionados, isso quer dizer que ele pode ser configurado a família de cálculos caixa ou BB.

Temos que garantir que as famílias sejam criadas de forma correta

Temos boletos de dois bancos, e sabemos que novos bancos podem ser inseridos

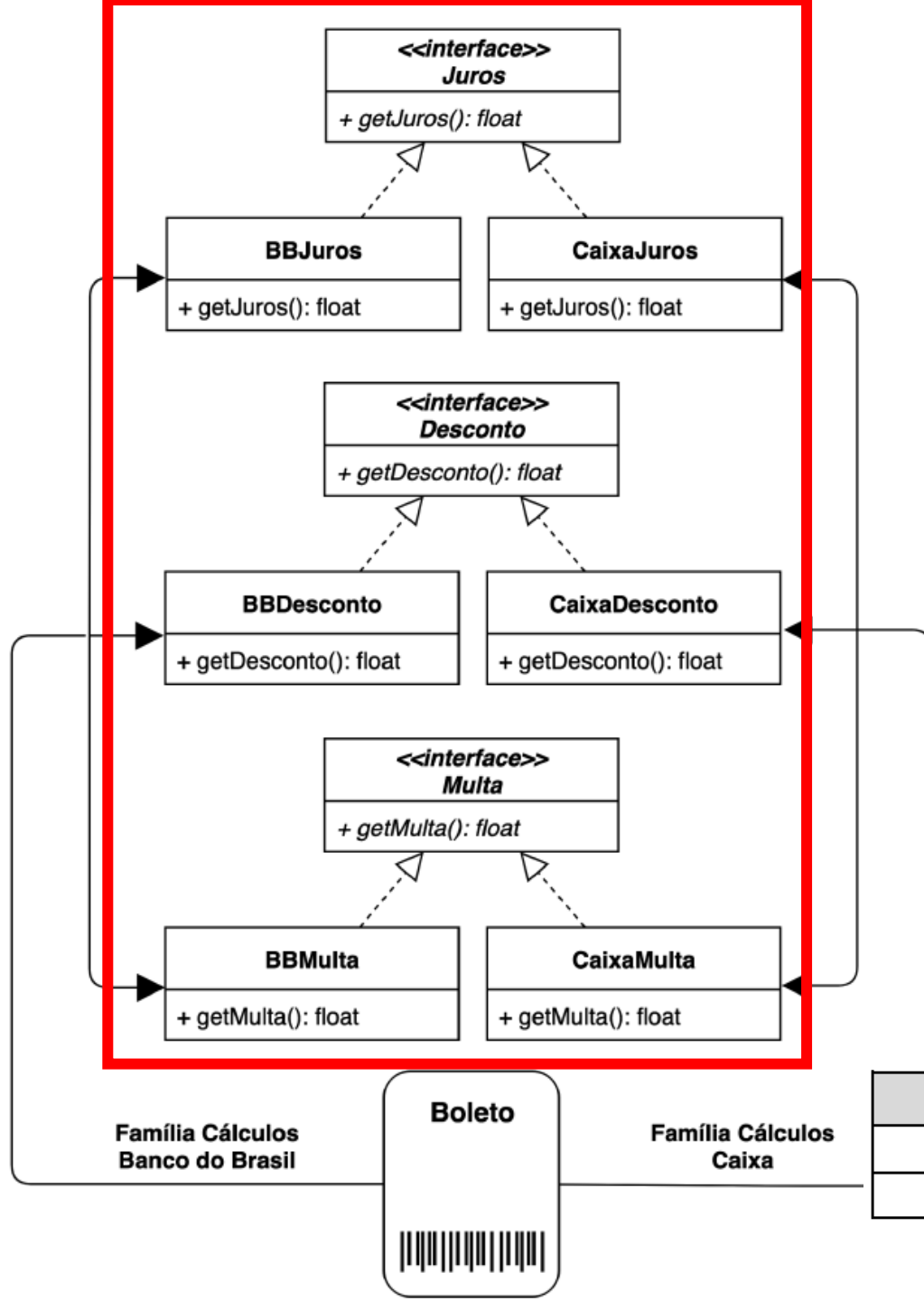


Banco	Juros	Desconto	Multa
Caixa	2%	10%	5%
Banco Do Brasil	3%	5%	2%

Cálculos de cada banco incidindo sobre o valor do boleto.

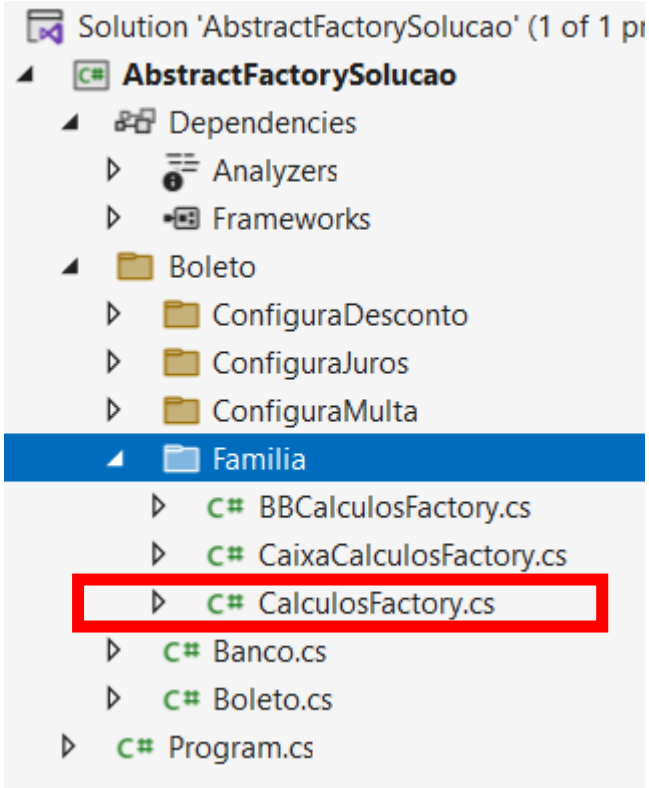
Temos dois bancos, portanto precisamos de duas classes de fábrica.

Para que tais classes tenham um padrão, vamos criar uma interface que dita os métodos que toda classe fábrica deve implementar.



Banco	Juros	Desconto	Multa
Caixa	2%	10%	5%
Banco Do Brasil	3%	5%	2%

Cálculos de cada banco incidindo sobre o valor do boleto.

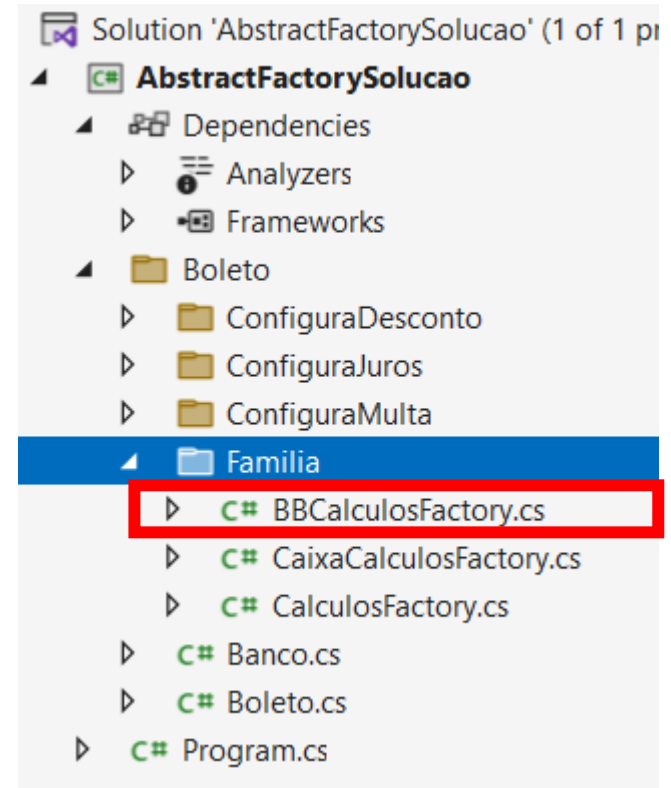


C# - Implementando a interface (supertipo) CalculosFactory

```
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5 using System.Threading.Tasks;
6 using AbstractFactorySolucao.Boleto.ConfiguraJuros;
7 using AbstractFactorySolucao.Boleto.ConfiguraDesconto;
8 using AbstractFactorySolucao.Boleto.ConfiguraMulta;
9
10 namespace AbstractFactorySolucao.Boleto.Familia
11 {
12     4 references
13     public interface CalculosFactory
14     {
15         3 references
16         public Juros criarJuros();
17         3 references
18         public Desconto criarDesconto();
19         3 references
20         public Multa criarMulta();
21     }
22 }
```

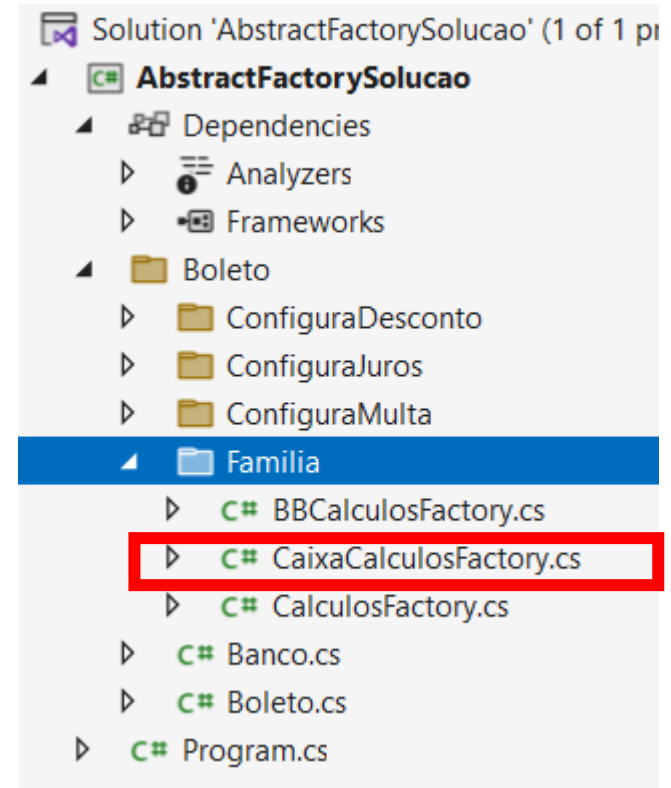
C# - Implementando as classes concretas de CalculosFactory → BBCalculosFactory

```
1 using AbstractFactorySolucao.Boleto.ConfiguraDesconto;  
2 using AbstractFactorySolucao.Boleto.ConfiguraJuros;  
3 using AbstractFactorySolucao.Boleto.ConfiguraMulta;  
4 using System;  
5 using System.Collections.Generic;  
6 using System.Linq;  
7 using System.Text;  
8 using System.Threading.Tasks;  
9  
10 namespace AbstractFactorySolucao.Boleto.Familia  
11 {  
12     2 references  
13     public class BBCalculosFactory : CalculosFactory  
14     {  
15         2 references  
16         public Desconto criarDesconto()  
17         {  
18             return new BBDesconto();  
19         }  
20  
21         2 references  
22         public Juros criarJuros()  
23         {  
24             return new BBJuros();  
25         }  
26  
27         2 references  
28         public Multa criarMulta()  
29         {  
30             return new BBMulta();  
31         }  
32     }  
33 }
```



C# - Implementando as classes concretas de CalculosFactory → CaixaCalculosFactory

```
1 using AbstractFactorySolucao.Boleto.ConfiguraDesconto;  
2 using AbstractFactorySolucao.Boleto.ConfiguraJuros;  
3 using AbstractFactorySolucao.Boleto.ConfiguraMulta;  
4 using System;  
5 using System.Collections.Generic;  
6 using System.Linq;  
7 using System.Text;  
8 using System.Threading.Tasks;  
9  
10 namespace AbstractFactorySolucao.Boleto.Familia  
11 {  
12     2 references  
13     public class CaixaCalculosFactory : CalculosFactory  
14     {  
15         2 references  
16         public Desconto criarDesconto()  
17         {  
18             return new CaixaDesconto();  
19         }  
20  
21         2 references  
22         public Juros criarJuros()  
23         {  
24             return new CaixaJuros();  
25         }  
26  
27         2 references  
28         public Multa criarMulta()  
29         {  
30             return new CaixaMulta();  
31         }  
32     }  
33 }
```



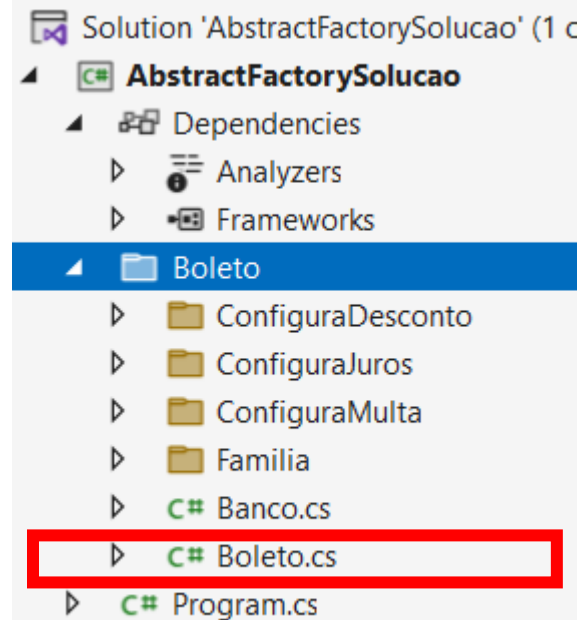
C# - Implementando a classe Boleto

```
1 using AbstractFactorySolucao.Boleto.ConfiguraDesconto;
2 using AbstractFactorySolucao.Boleto.ConfiguraJuros;
3 using AbstractFactorySolucao.Boleto.ConfiguraMulta;
4 using AbstractFactorySolucao.Boleto.Familia;
5 using System;
6 using System.Collections.Generic;
7 using System.Linq;
8 using System.Text;
9 using System.Threading.Tasks;

10
11 namespace AbstractFactorySolucao.Boleto
12 {
13     4 references
14     public class Boleto
15     {
16         protected double valor;
17         protected Juros juros;
18         protected Desconto desconto;
19         protected Multa multa;
20
21         1 reference
22         public Boleto(double valor, CalculosFactory factory)
23         {
24             this.valor = valor;
25             this.juros = factory.criarJuros();
26             this.desconto = factory.criarDesconto();
27             this.multa = factory.criarMulta();
28
29         1 reference
30         public double calcularJuros()
31         {
32             return this.valor * this.juros.getJuros();
33         }
34     }
35 }
36
37
38
39
40
41
42
43
44 }
```

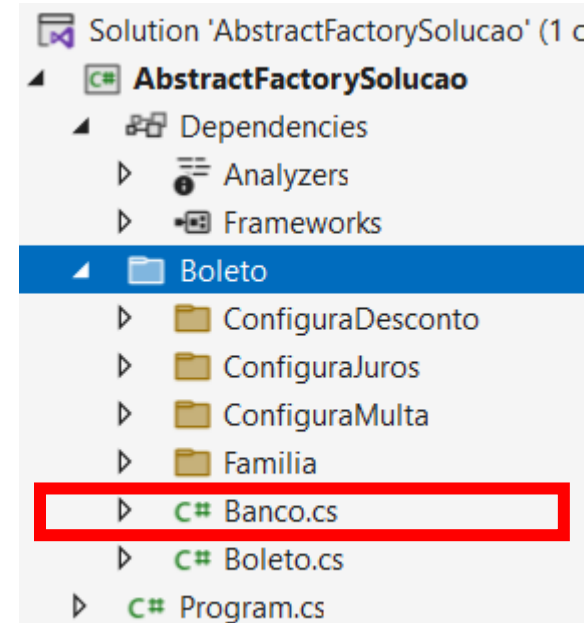
```
1 reference
public double calcularDesconto()
{
    return this.valor * this.desconto.getDesconto();
}

1 reference
public double calcularMulta()
{
    return this.valor * this.multa.getMulta();
}
}
```



C# - Implementando a classe Banco

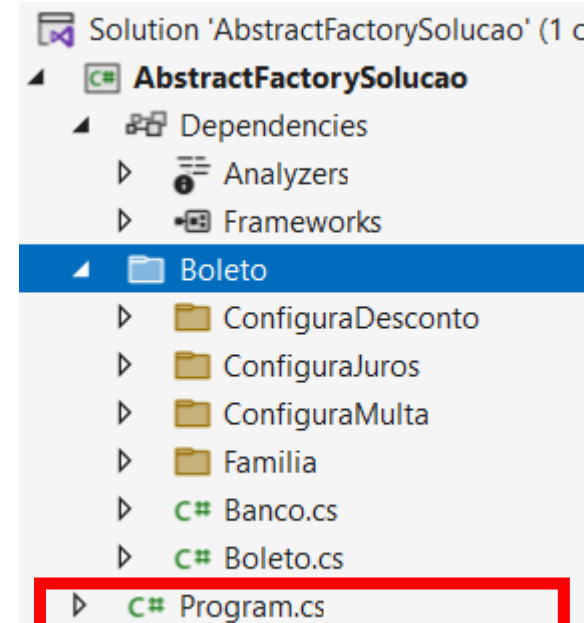
```
1  using AbstractFactorySolucao.Boleto.Familia;
2  using System;
3  using System.Collections.Generic;
4  using System.Linq;
5  using System.Text;
6  using System.Threading.Tasks;
7
8  namespace AbstractFactorySolucao.Boleto
9  {
10     2 references
11     public class Banco
12     {
13         2 references
14         public Boleto gerarBoleto(double valor, CalculosFactory factory)
15         {
16             Boleto boleto = new Boleto(valor, factory);
17
18             Console.WriteLine("Boleto gerado com sucesso no valor de R$" + valor);
19             Console.WriteLine("Valor Juros R$" + boleto.calcularJuros());
20             Console.WriteLine("Valor Multa R$" + boleto.calcularMulta());
21             Console.WriteLine("Valor Desconto R$" + boleto.calcularDesconto());
22             Console.WriteLine("-----");
23
24             return boleto;
25         }
26     }
27 }
```




C# - Implementando a classe Cliente

```
1 using AbstractFactorySolucao.Boleto;  
2 using AbstractFactorySolucao.Boleto.Familia;  
3  
4 Banco banco = new Banco();  
5  
6 CalculosFactory factoryCaixa = new CaixaCalculosFactory();  
7  
8 CalculosFactory factoryBB = new BBCalculosFactory();  
9  
10 Console.WriteLine("cria boleto caixa");  
11 banco.gerarBoleto(100, factoryCaixa);  
12  
13 Console.WriteLine("cria boleto BB");  
14 banco.gerarBoleto(100, factoryBB);
```

```
Microsoft Visual Studio Debug Console  
cria boleto caixa  
Boleto gerado com sucesso no valor de R$100  
Valor Juros R$2  
Valor Multa R$5  
Valor Desconto R$10  
-----  
cria boleto BB  
Boleto gerado com sucesso no valor de R$100  
Valor Juros R$3  
Valor Multa R$2  
Valor Desconto R$5  
-----
```





Abstract Factory

Implementação em Java

Padrões de Projeto Criacional I

Prof. Me Jefferson Passerini



**ABSTRACT
FACTORY**



Abstract Factory

Consequências

Padrões de Projeto Criacional I

Prof. Me Jefferson Passerini



Consequências

- Promove o isolamento de classes concretas.
 - O padrão Abstract Factory ajuda a controlar as classes de objetos que um sistema cria. Como uma fábrica encapsula a responsabilidade e o processo de criação de produtos concretos, ela isola os clientes de tais responsabilidades.
 - Os clientes manipulam instâncias concretas por meio de suas interfaces abstratas.
 - Os nomes das classes ProdutoConcreto ficam isolados na implementação da fábrica concreta e não chegam no Cliente.

Consequências

- Facilita a troca de famílias de produtos.
 - Uma fábrica concreta aparece apenas uma vez em um cliente, ou seja, onde é instanciada, isso facilita sua alteração.
 - Um cliente pode usar diferentes configurações de produtos simplesmente alterando sua fábrica concreta em tempo de execução.

Consequências

- Promove a consistência entre produtos.
 - Quando os objetos são projetados para trabalhar juntos em uma família de produto, é importante que o cliente seja composto por objetos de apenas uma família por vez, ou seja, as famílias não devem se misturar.
 - O padrão Abstract Factory facilita tal controle.

Consequências

- Suportar novos tipos de produtos é difícil.
 - Isso ocorre porque a interface `AbstractFactory` define o conjunto de produtos que podem ser criados.
 - O suporte a novos tipos de produtos requer a extensão da interface ou classe abstrata `AbstractFactory` e de todas as suas subclasses (`FabricaConcreta`) também precisarão ser extendidas

Consequências

- Embora criar novos tipos de produtos seja difícil, criar novos produtos de um tipo já existente é fácil e não causa refatoração no Cliente.