Abstract Factory

Padrões Criacionais

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

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.

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.

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*.

Vamos extrair os trechos do primeiro parágrafo para que possamos entendê-lo por partes:

- Trecho 1: *"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.

- Trecho 2: *"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 relacionam para compor um boleto, são eles: juros, desconto e multa. Portanto teremos 3 supertipos que irão compor um boleto.

Temos abaixo os três supertipos (interfaces) que compõem um boleto.

Juros

```
interface Juros
{
    //Retorna a taxa de juros do boleto,
    public function getJuros(): float;
}
```

Desconto

```
interface Desconto
{
    //Retorna o desconto do boleto,
    public function getDesconto(): float;
}
```

Multa

```
interface Multa
{
    //Retorna a multa do boleto,
    public function getMulta(): float;
}
```

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

Cálculos de cada banco incidindo sobre o valor do boleto.

Como temos dois banco teremos duas classes concretas para cada interface acima:

Juros

```
class CaixaJuros implements Juros
{
    public function getJuros(): float
    {
        return 0.02; //Retorna 2%.
    }
}
```

```
class BBJuros implements Juros
{
    public function getJuros(): float
    {
        return 0.03; //Retorna 3%.
    }
}
```

Desconto

```
class CaixaDesconto implements Desconto
{
    public function getDesconto(): float
    {
        return 0.1; //Retorna 10%.
    }
}
```

```
class BBDesconto implements Desconto
{
    public function getDesconto(): float
    {
        return 0.05; //Retorna 5%.
    }
}
```

Multa

```
class CaixaMulta implements Multa
{
    public function getMulta(): float
    {
        return 0.05; //Retorna 5%.
    }
}
```

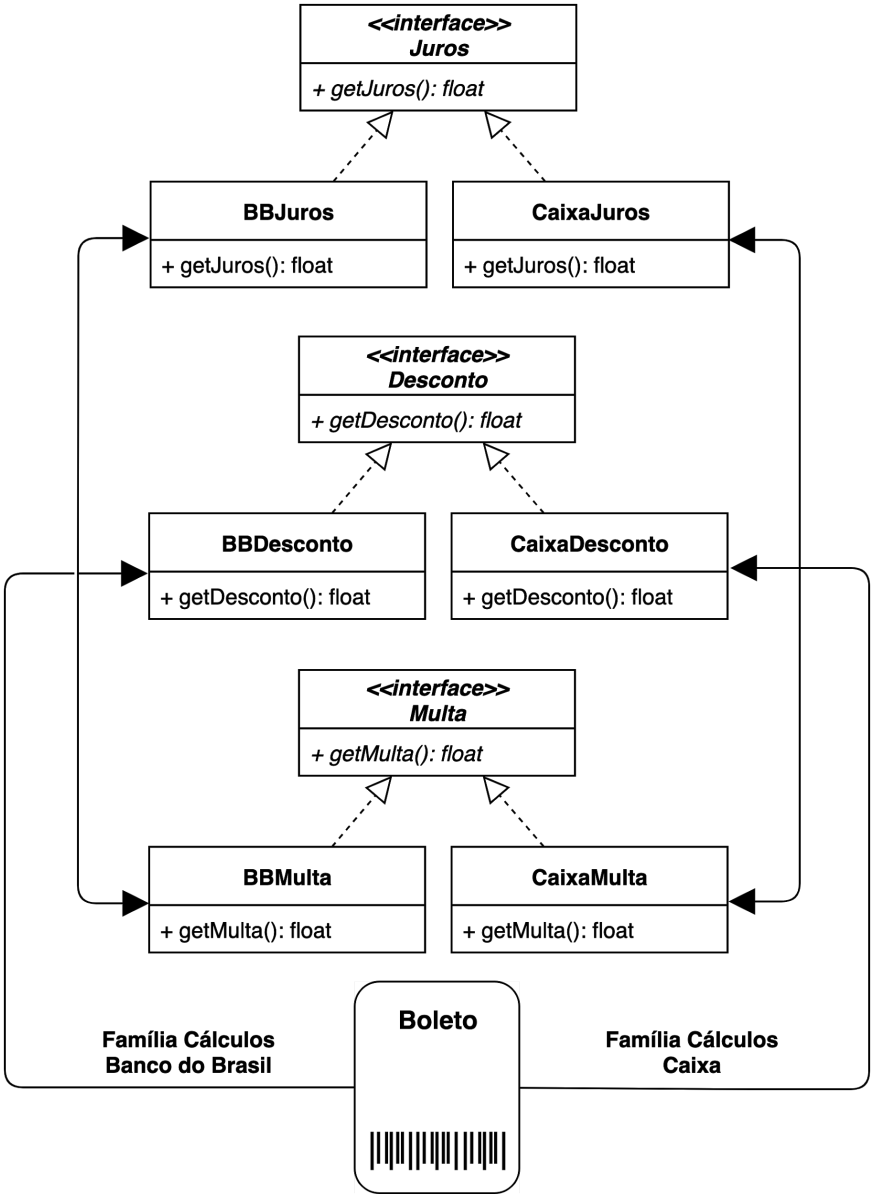
```
class BBMulta implements Multa
{
    public function getMulta(): float
    {
        return 0.02; //Retorna 2%.
    }
}
```

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 objeto Juros + 1 objeto Desconto + 1 objeto Multa

Deste modo, em nosso exemplo existem famílias de objetos:

- Família de cálculos Caixa;
- Família de cálculos Banco do Brasil.



Boleto criando diretamente um objeto de cada tipo conforme a sua necessidade

Um boleto deve ser configurado a partir de uma família de objetos relacionados, isso quer dizer ele pode ser configurado com a **família de cálculos Caixa** ou com a **Família de cálculos Banco do Brasil**.

Precisamos garantir para o `Cliente` que as famílias sejam montadas de forma correta. Temos boletos que serão emitidos por dois bancos distintos, sabemos que novos bancos poderão ser inseridos no módulo de cobranças. Para garantir que todo boleto, independente do banco de origem, tenha um objeto do tipo `Juros`, um do tipo `Desconto` e outro do tipo `Multa` vamos criar uma classe fábrica para cada banco, e será responsabilidade dessas classes criar um objeto de cada tipo que se relacionam corretamente.

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.

```
interface CalculosFactory
{
    //Uma classe fábrica deve ter um método para criar um objeto correto de cada tipo.

    public function criarJuros(): Juros; //Cria um objeto do tipo Juros.

    public function criarDesconto(): Desconto; //Cria um objeto do tipo Desconto.

    public function criarMulta(): Multa; //Cria um objeto do tipo Multa.
}
```

Agora que já sabemos como cada fábrica deve ser, vamos criar uma fábrica de cálculos para cada Banco.

A classe **CaixaCalculosFactory** é responsável por instanciar os objetos de cálculos concretos que irão compor um boleto do Banco Caixa. Podemos dizer que essa classe fábrica cria a Família de Cálculos Caixa.

```
class CaixaCalculosFactory implements CalculosFactory
{
    public function criarJuros(): Juros
    {
        return new CaixaJuros(); //Retorna um objeto do tipo Juros.
    }

    public function criarDesconto(): Desconto
    {
        return new CaixaDesconto(); //Retorna um objeto do tipo Desconto.
    }

    public function criarMulta(): Multa
    {
        return new CaixaMulta(); //Retorna um objeto do tipo Multa.
    }
}
```

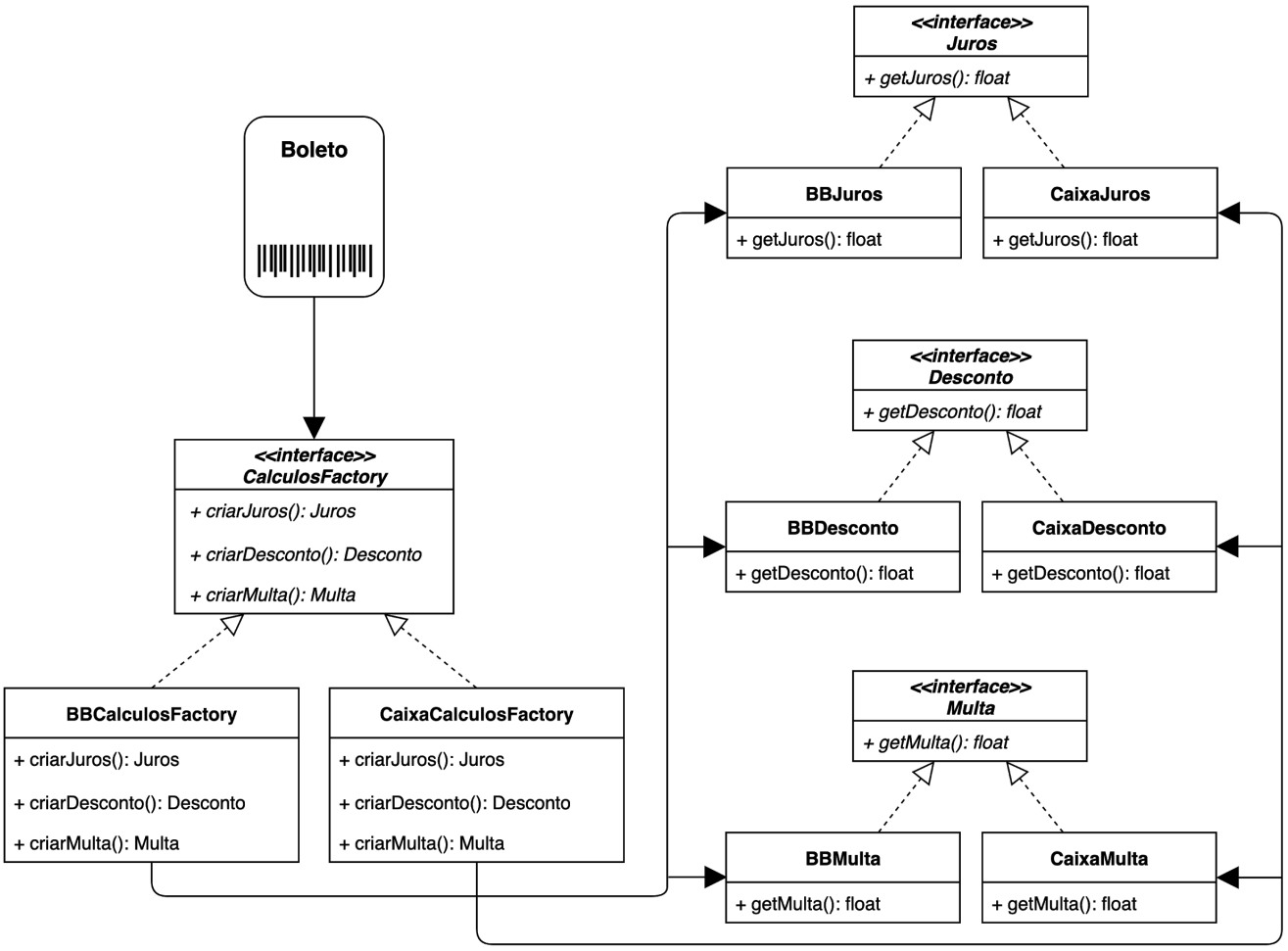
Já a classe **BBCalculosFactory** é responsável por instanciar os objetos de cálculos concretos que irão compor um boleto do Banco Caixa. Podemos dizer que essa é a família de cálculos Caixa.

```
class BBCalculosFactory implements CalculosFactory
{
    public function criarJuros(): Juros
    {
        return new BBJuros(); //Retorna um objeto do tipo Juros.
    }

    public function criarDesconto(): Desconto
    {
        return new BBDesconto(); //Retorna um objeto do tipo Desconto.
    }

    public function criarMulta(): Multa
    {
        return new BBMulta(); //Retorna um objeto do tipo Multa.
    }
}
```

Como pode ser observado no diagrama a seguir existe uma relação entre as fábricas e os diferentes tipos de objetos. Cada fábrica cria uma família de objetos que fazem sentido juntos.



Boleto criando objetos por intermédio de fábricas

Boleto aqui é apenas uma representação do restante das classes do módulo de cobranças. Antes, Boleto precisava criar diretamente as famílias objetos que lhe eram necessárias. Agora a responsabilidade de criação das famílias é das fábricas, basta que o boleto solicite a fábrica apropriada que crie a família de objetos para ele.

Podem existir várias classes fábricas concretas, o Boleto pode utilizar todas elas, portanto, ele não pode esperar por uma fábrica concreta específica e sim um supertipo (Classe Abstrata ou Interface), ou seja, o Boleto deve esperar por uma fábrica abstrata em inglês (***Abstract Factory***), no nosso caso é a interface `CalculosFactory`.

Vamos à classe onde podemos ver os benefícios do padrão *Abstract Factory* de forma bem explicita. Criaremos a classe `Boleto`, repare que nela recebemos em seu construtor um objeto do supertipo `CalculosFactory`, então, poderá ser uma instância de `CaixaCalculosFactory` ou `BBCalculosFactory`.

O objeto recebido no construtor de `Boleto` define qual família de cálculos será criada, em seguida, iremos atribuir as variáveis de

instância **$juros**, **$desconto** e **$multa** os respectivos objetos produzidos pela *factory* recebida no construtor, ou seja, os membros da família criada.

```
class Boleto
{
    protected float $valor;
    protected Juros $juros; //Guarda referência a um objeto do tipo Juros.
    protected Desconto $desconto; //Guarda referência a um objeto do tipo Juros.
    protected Multa $multa; //Guarda referência a um objeto do tipo Juros.

    //O boleto recebe em seu construtor o seu valor e a fábrica que deve utilizar
    //para criar os cálculos.
    public function __construct(float $valor, CalculosFactory $factory)
    {
        $this->valor = $valor;
        //Guarda localmente os objetos criados pela fábrica.
        $this->juros = $factory->criarJuros(); //A fábrica cria um objeto de Juros.
        $this->desconto = $factory->criarDesconto(); //A fábrica cria um objeto de Desconto.
        $this->multa = $factory->criarMulta(); //A fábrica cria um objeto de Multa.
    }

    public function calcularJuros(): float
    {
        //Retorna o valor do juros em reais, usa o método getJuros() do objeto do tipo Juros
        //criado pela fábrica.
        return $this->valor * $this->juros->getJuros();
    }

    public function calcularDesconto(): float
    {
        //Retorna o valor do Desconto em reais, usa o método getDesconto() do objeto
        //do tipo Desconto criado pela fábrica.
        return $this->valor * $this->desconto->getDesconto();
    }

    public function calcularMulta(): float
    {
        //Retorna o valor da Multa em reais, usa o método getMulta() do objeto do tipo Multa
        //criado pela fábrica.
        return $this->valor * $this->multa->getMulta();
    }
}
```

Apenas para manter uma certa equivalência em relação ao exemplo que utilizamos no padrão *Factory Method* vamos criar **Banco** que será a classe responsável por criar o boleto para o **Cliente.** No exemplo do Factory Method a escolha do banco ditava qual boleto seria criado, por este motivo existiam duas classes de Banco **BancoDoBrasil** e **BancoCaixa.**

O que varia entre os os bancos são os cálculos de cada boleto: juros, desconto e multa. Com o *Factory Method* precisamos de apenas uma classes **Banco** para criar os boletos da Caixa e do Banco do Brasil, pois o que determina os cálculos do boleto não são as classes de Banco, mas sim a fábrica que foi escolhida.

```php
class Banco
{
    //O Banco recebe no método gerarBoleto() o valor do boleto a ser criado
    //e a fábrica que deve utilizar para criar os cálculos.
    public function gerarBoleto(float $valor, CalculosFactory $factory): Boleto
    {
        //Passa o valor e a fábrica para o boleto.
        //Boleto utiliza a fábrica recebida para o Juros, Desconto e Multa.
        $boleto = new Boleto($valor, $factory);

        //Apenas imprime os resultados dos cálculos do boleto que foi criado.
        echo 'Boleto gerado com sucesso no valor de R$ ' . $valor . '<br>';
        echo 'Valor Juros: R$' . $boleto->calcularJuros() . '<br>';
        echo 'Valor Desconto: R$' . $boleto->calcularDesconto() . '<br>';
        echo 'Valor Multa: R$' . $boleto->calcularMulta() . '<br>';
        echo '---------------' . '<br><br>';

        return $boleto;
    }
}
```

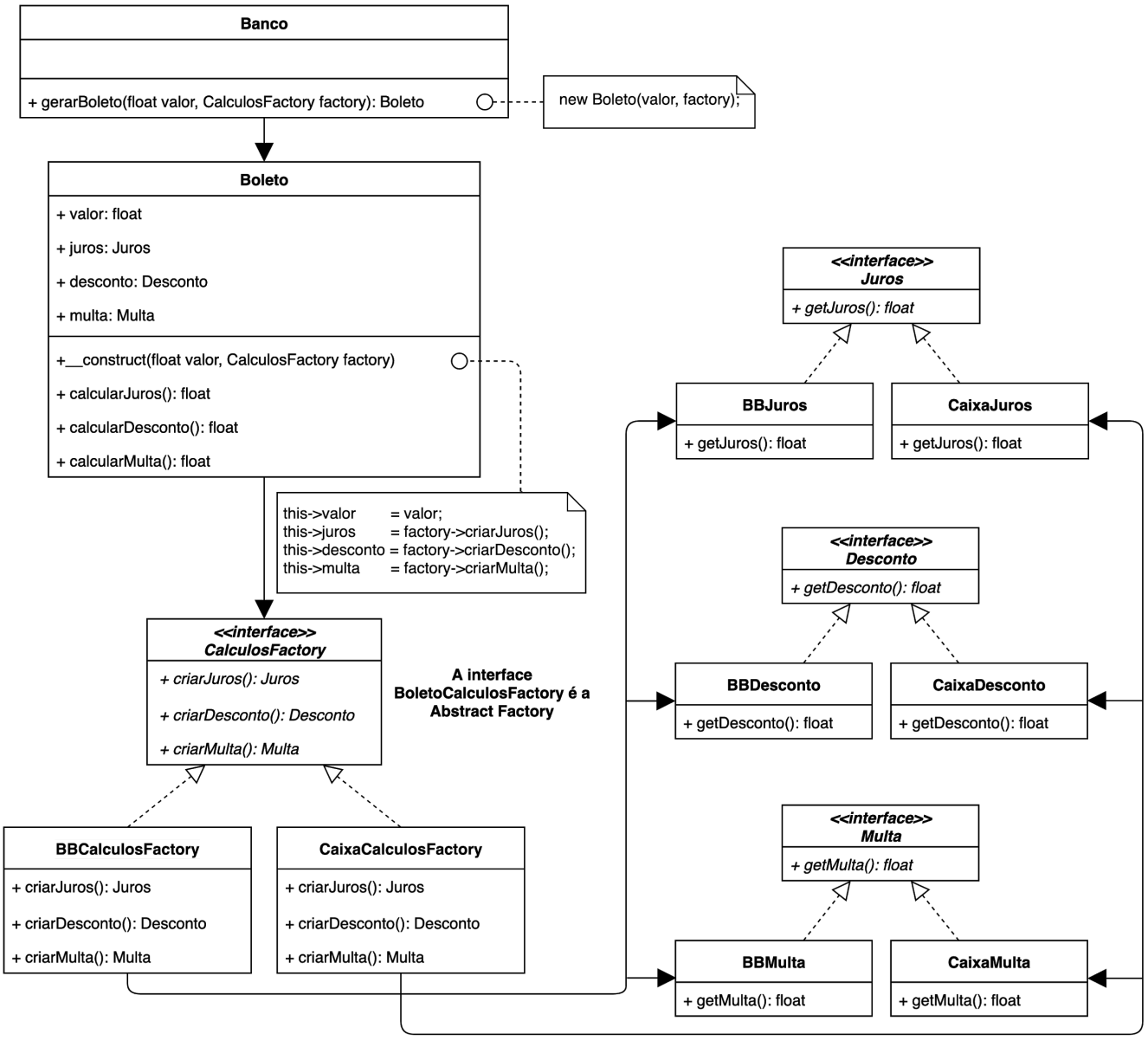


Diagrama de classes completo do módulo de cobranças

Já temos todas as classes que precisávamos e podemos testar nosso módulo de cobranças.

```php
//Criação de um objeto Banco.
$banco = new Banco();

//Criação de uma fábrica calculos para boletos da Caixa.
$factoryCaixa = new CaixaCalculosFactory();

//Criação de uma fábrica calculos para boletos do Banco do Brasil.
$factoryBancoBrasil = new BBCalculosFactory();

//Criação de um boleto da Caixa.
echo "### Boleto da Caixa ###<br>";
$banco->gerarBoleto( 100, $factoryCaixa);

//Criação de um boleto do Banco do Brasil.
echo "### Boleto do Banco do Brasil ###<br>";
$banco->gerarBoleto(100, $factoryBancoBrasil);
```

Saída:

```
### Boleto da Caixa ###
Boleto gerado com sucesso no valor de R$ 100
Valor Juros: R$3
Valor Desconto: R$5
Valor Multa: R$2
---------------

### Boleto do Banco do Brasil ###
Boleto gerado com sucesso no valor de R$ 100
Valor Juros: R$2
Valor Desconto: R$10
Valor Multa: R$5
---------------
```

Veja como criamos apenas um objeto de **Banco** e chamamos exatamente o mesmo método **gerarBoleto()** para criar os boletos da Caixa e do Banco do Brasil. A única coisa diferente foi a fábrica passada por parâmetro para o método **gerarBoleto()**, pudemos fazer isso devido ao fato do método esperar em seu parâmetro por uma fábrica abstrata (**CalculosFactory**) e assim os objetos do tipo **CaixaCalculosFactory** e **BBCalculosFactory** foram aceitos por ele provocando a criação de famílias de objetos diferente e por consequência boletos com diferentes cálculos.

Voltando para o primeiro parágrafo desta seção, vamos extrair seu último trecho que diz:

- *"Isto permite que novos tipos derivados sejam introduzidos sem qualquer alteração ao código que usa a classe base."*
  Se no futuro for necessário gerar boletos para mais bancos não precisaremos mexer em nossa classe base, que seria a classe **Boleto** no nosso exemplo. Bastaria criar as classes de cálculos do novo banco, por exemplo:

  - **BancoItauJuros** - Implementa a interface **Juros**.
  - **BancoItauDesconto** - Implementa a interface **Desconto**.
  - **BancoItauMulta** - Implementa a interface **Multa**.

  E ainda seria necessário criar uma nova fábrica, que cria uma nova família de cálculos, para o novo banco. Deste modo o **Cliente** seria capaz de criar boletos para novos bancos apenas utilizando a nova fábrica. Da mesma forma que temos as fábricas **BBCalculosFactory** e **CaixaCalculosFactory** poderíamos ter uma classe **BancoItauCalculosFactory** que também implementaria a interface **CalculosFactory**.

Com o padrão *Abstract Factory* somos capazes de expandir a classe **Boleto** sem precisar mudar seu código, deste modo, ela está aberta para extensão e fechada para modificação.

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

- **AbstractFactory:** Declara uma interface a qual todas as fábricas concretas devem implementar, o que consiste em um conjunto de métodos para fabricar produtos concretos.

- **FabricaConcreta:**

    - Implementam a interface declarada por AbstractFactory.
    - Criam as diferentes famílias de produtos.
    - Para criar um produto, o Cliente usa uma dessas fábricas concretas, então ele nunca precisa criar produtos concretos diretamente.

- **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.

- **Cliente:** Usa apenas interfaces declaradas pelas classes AbstractFactory e ProdutoAbstrato, e é composta em tempo de execução por fábricas e produtos concretos.

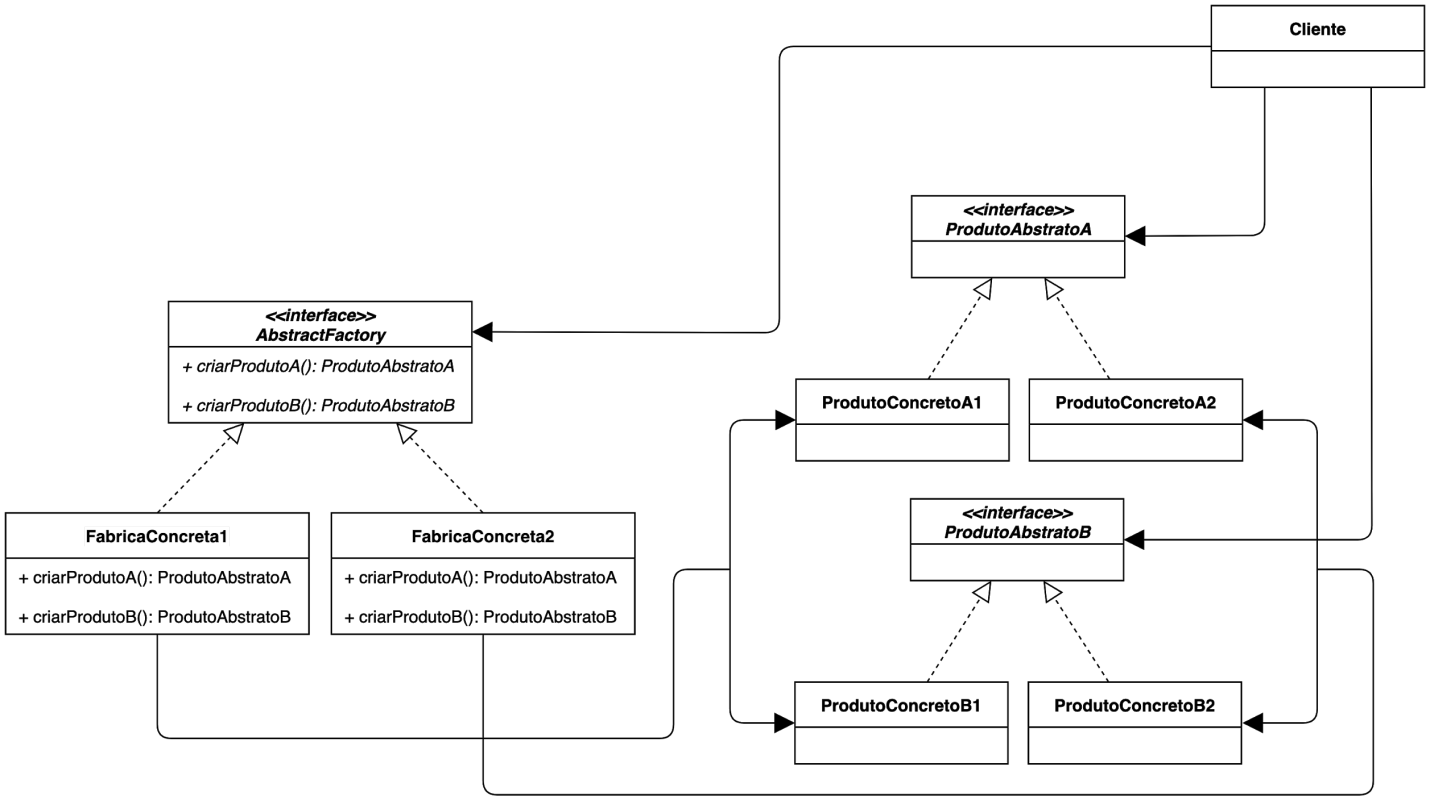


Diagrama de Classes

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.

- 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.

- 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.

- 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.

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