Factory Method

Padrões Criacionais

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

Motivação (Por que utilizar?)

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.

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:

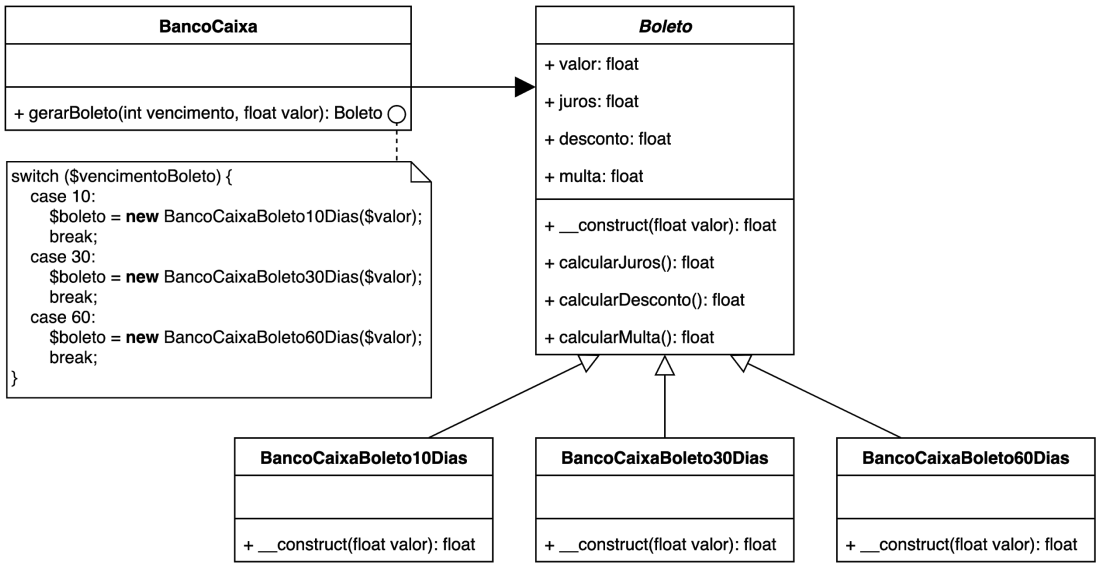


Diagrama de classes da estrutura inicial do módulo de cobranças.

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.

```
abstract class Boleto
{
    protected float $valor;
    protected float $juros;
    protected float $desconto;
    protected float $multa;

    public function __construct(float $valor)
    {
        $this->valor = $valor;
    }

    public function calcularJuros(): float //Calcula valor do Juros do boleto.
    {
        return $this->valor * $this->juros;
    }

    public function calcularDesconto(): float //Calcula valor do Desconto do boleto.
    {
        return $this->valor * $this->desconto;
    }

    public function calcularMulta(): float //Calcula valor da Multa do boleto.
    {
        return $this->valor * $this->multa;
    }
}
```

```
class BancoCaixaBoleto10Dias extends Boleto
{
    public function __construct(float $valor)
    {
        parent::__construct($valor);
        $this->juros = 0.02;
        $this->desconto = 0.1;
        $this->multa = 0.05;
    }
}
```

```
class BancoCaixaBoleto30Dias extends Boleto
{
    public function __construct(float $valor)
    {
        parent::__construct($valor);
        $this->juros = 0.05;
        $this->desconto = 0.05;
        $this->multa = 0.1;
    }
}
```

```php
class BancoCaixaBoleto60Dias extends Boleto
{
    public function __construct(float $valor)
    {
        parent::__construct($valor);
        $this->juros = 0.10;
        $this->desconto = 0;
        $this->multa = 0.2;
    }
}
```

```php
class BancoCaixa
{
    public function gerarBoleto(int $vencimentoBoleto, float $valor): Boleto
    {
        //Criação do boleto conforme o vencimento. A palavra new é utilizada 3 vezes.
        switch ($vencimentoBoleto) {
            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');
        }

        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;
    }
}
```

Uma maneira de utilizar as classes acima seria a seguinte:

```php
//Criação do Banco
$banco = new BancoCaixa();

//Boleto com 10 dias para o vencimento. Valor R$100,00.
$banco->gerarBoleto(10, 100);

//Boleto com 10 dias para o vencimento. Valor R$100,00.
$banco->gerarBoleto(30, 100);

//Boleto com 10 dias para o vencimento. Valor R$100,00.
$banco->gerarBoleto(60, 100);
```

Saída:

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

Boleto gerado com sucesso no valor de R$ 100
Valor Juros: R$5
Valor Desconto: R$5
Valor Multa: R$10
---------------

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

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.

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**.

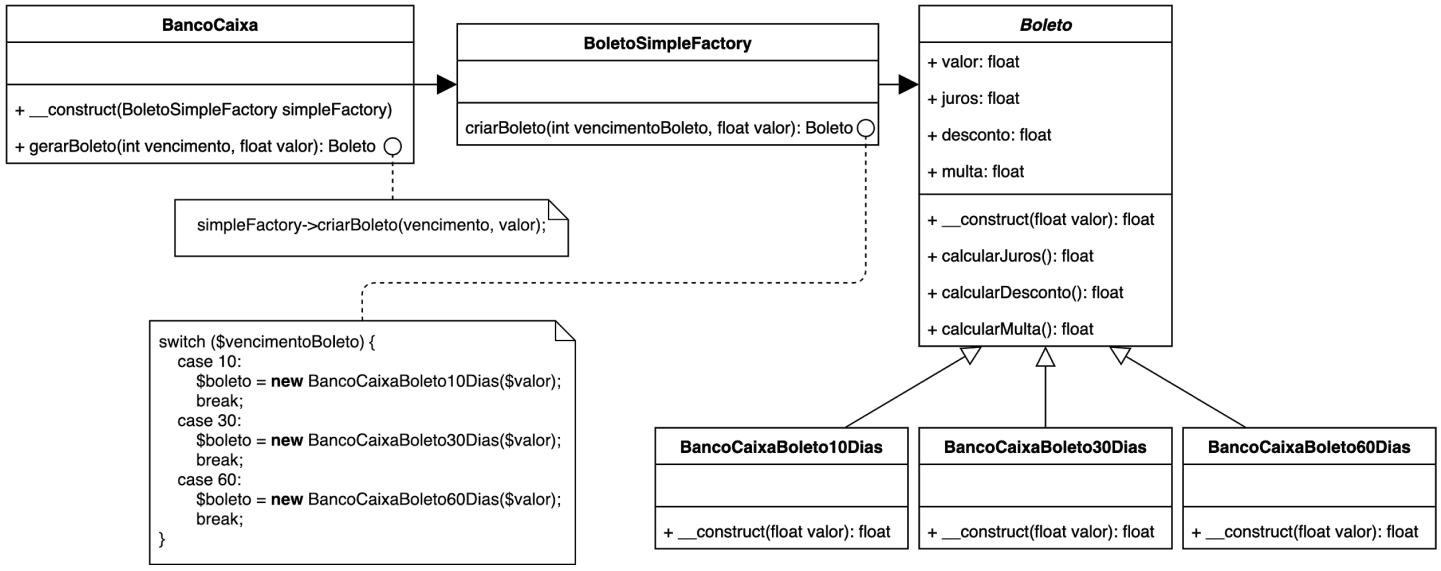


Diagrama de classes da estrutura do módulo de cobranças utilizando *Simple Factory*.

As classes **Boleto**, **Boleto10Dias**, **Boleto30Dias** ou **Boleto60Dias** não mudam. Vejamos a criação da classe **BoletoSimpleFactory** e as mudanças na classe **BancoCaixa**.

```php
class BoletoSimpleFactory
{
    public function criarBoleto(int $vencimentoBoleto, float $valor): Boleto
    {
        //Criação do boleto conforme o vencimento. A palavra new é utilizada 3 vezes.
        switch ($vencimentoBoleto) {
            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;
    }
}
```

```php
class BancoCaixa
{
    private BoletoSimpleFactory $boletoSimpleFactory;

    public function __construct(BoletoSimpleFactory $boletoSimpleFactory)
    {
        //As criações dos boletos saíram daqui. A palavra new não é mais utilizada.
        $this->boletoSimpleFactory = $boletoSimpleFactory;
    }

    public function gerarBoleto(int $vencimentoBoleto, float $valor): Boleto
    {
        $boleto = $this->boletoSimpleFactory->criarBoleto($vencimentoBoleto, $valor);

        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;
    }
}
```

Uma maneira de utilizar o novo módulo de cobranças seria a seguinte:

```
//Criação da Simple Factory.
$boletoSimpleFactory = new BoletoSimpleFactory();

//Criação do Banco passando a Simple Factory por parâmetro.
$banco = new BancoCaixa($boletoSimpleFactory);

//Boleto com 10 dias para o vencimento. Valor R$100,00.
$banco->gerarBoleto(10, 100);

//Boleto com 10 dias para o vencimento. Valor R$100,00.
$banco->gerarBoleto(30, 100);

//Boleto com 10 dias para o vencimento. Valor R$100,00.
$banco->gerarBoleto(60, 100);
```

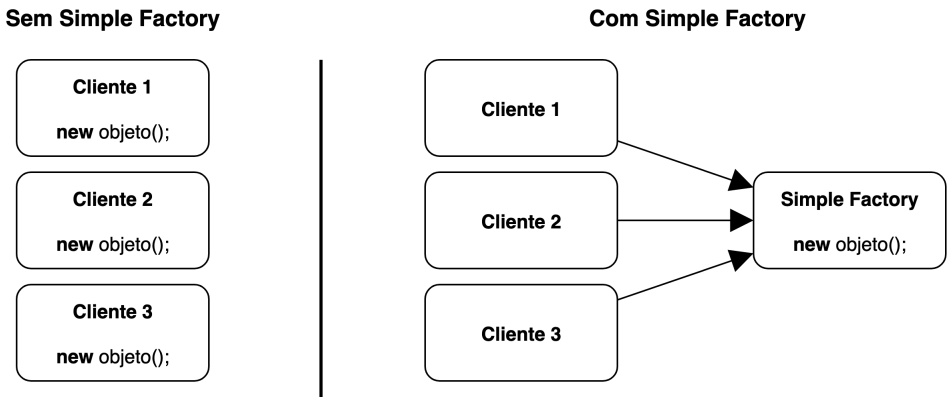
Saída:

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

Boleto gerado com sucesso no valor de R$ 100
Valor Juros: R$5
Valor Desconto: R$5
Valor Multa: R$10
---------------

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

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**.

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.

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.



Esquema de reutilização de código utilizando o Simple Factory.

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**.

Precisamos padronizar os bancos, garantir que eles possuam os mesmos métodos. Tanto o **BancoDoBrasil** quanto o banco **BancoCaixa** geram boletos o possuem o método **gerarBoleto()** a única diferença entre eles são os boletos que devem instanciar. Vamos criar uma classe abstrata **Banco** que possui o método **gerarBoleto()** e deixa a criação dos boletos para as subclasses de **Banco** que serão **BancoCaixa** e **BancoDoBrasil**.

Vejamos o código da classe abstrata **Banco**:

```php
abstract class Banco
{
    public function gerarBoleto(int $vencimentoBoleto, float $valor): Boleto
    {
        //Chama o método criarBoleto() que é abstrato, portanto será implementado apenas
        //Pelas subclasses de Banco.
        $boleto = $this->criarBoleto($vencimentoBoleto, $valor);

        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;
    }

    //Cada subclasse implementa o método criarBoleto criando  seus respectivos objetos.
    abstract protected function criarBoleto(int $vencimentoBoleto, float $valor): Boleto;
}
```

O Método criarBoleto() é um método fábrica, ele é o Factory Method.

Voltando aos boletos, além dos boletos de **BancoCaixa** também teremos agora os boletos do **BancoDoBrasil.**

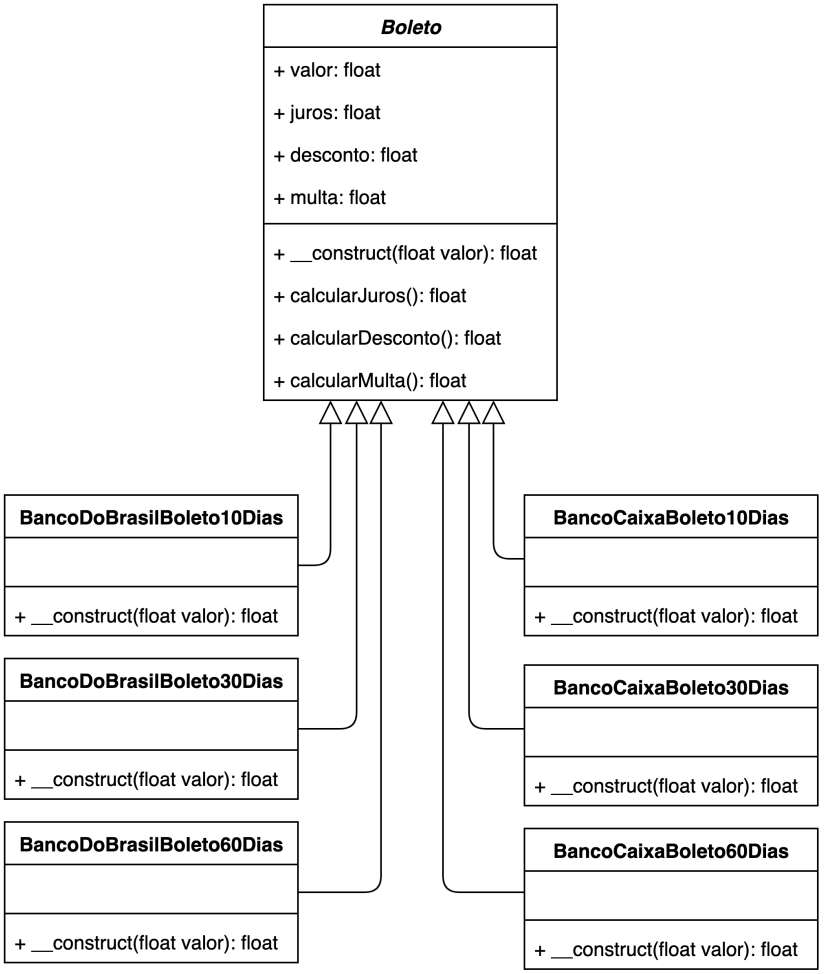


Diagrama de classes parcial - Inserção dos boletos do BancoDoBrasil.

Nossa tabela de porcentagem fica assim:

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.

Vejamos o código dos boletos do **BancoDoBrasil.**

```php
class BancoDoBrasilBoleto10Dias extends Boleto
{
    public function __construct(float $valor)
    {
        parent::__construct($valor);
        $this->juros = 0.03;
        $this->desconto = 0.05;
        $this->multa = 0.02;
    }
}
```

```php
class BancoDoBrasilBoleto30Dias extends Boleto
{
    public function __construct(float $valor)
    {
        parent::__construct($valor);
        $this->juros = 0.05;
        $this->desconto = 0.02;
        $this->multa = 0.5;
    }
}
```

```php
class BancoDoBrasilBoleto60Dias extends Boleto
{
    public function __construct(float $valor)
    {
        parent::__construct($valor);
        $this->juros = 0.1;
        $this->desconto = 0;
        $this->multa = 0.15;
    }
}
```

Agora que já temos os boletos, podemos criar as subclasses de **Banco**.

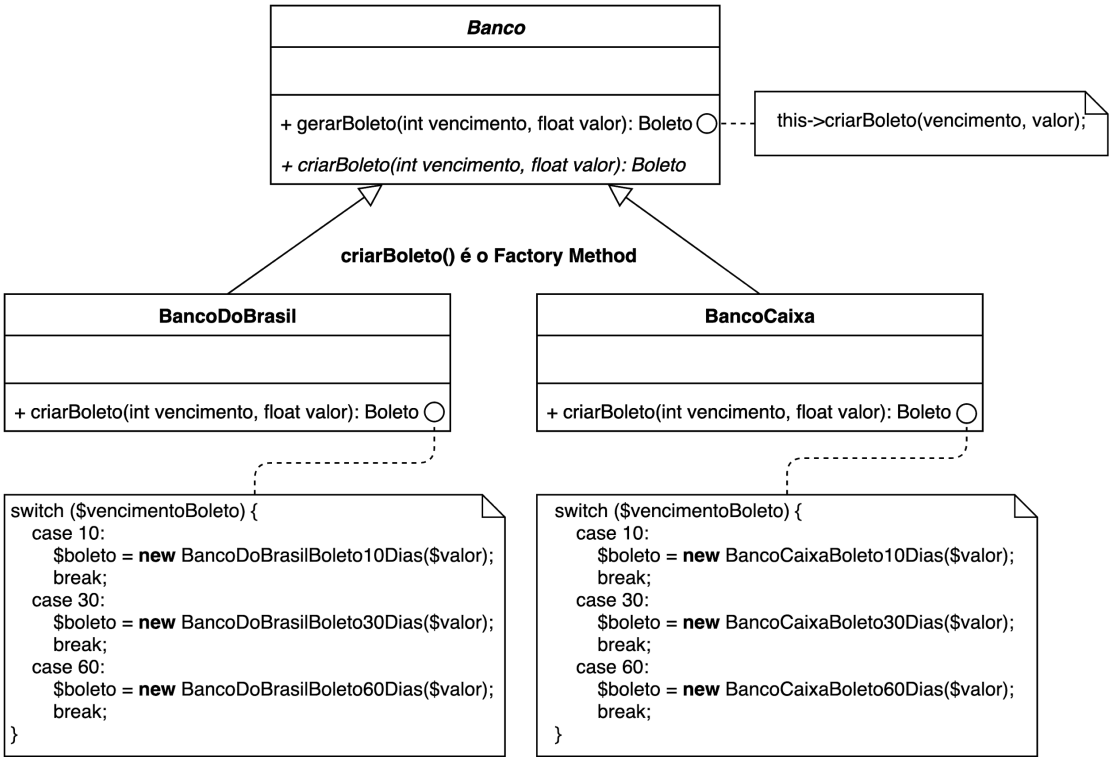


Diagrama de classes parcial - Criação das subclasses de Banco.

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.**

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().**

É importante ressaltar que um *Factory Method* não precisa ser parametrizado. No nosso caso estamos escolhendo qual objeto criar com base no parâmetro **vencimento.** É muito comum que o *Factory Method* não tenha parâmetros e instancie apenas um objeto.

Vamos juntar as partes agora e visualizar o diagrama de classes completo.

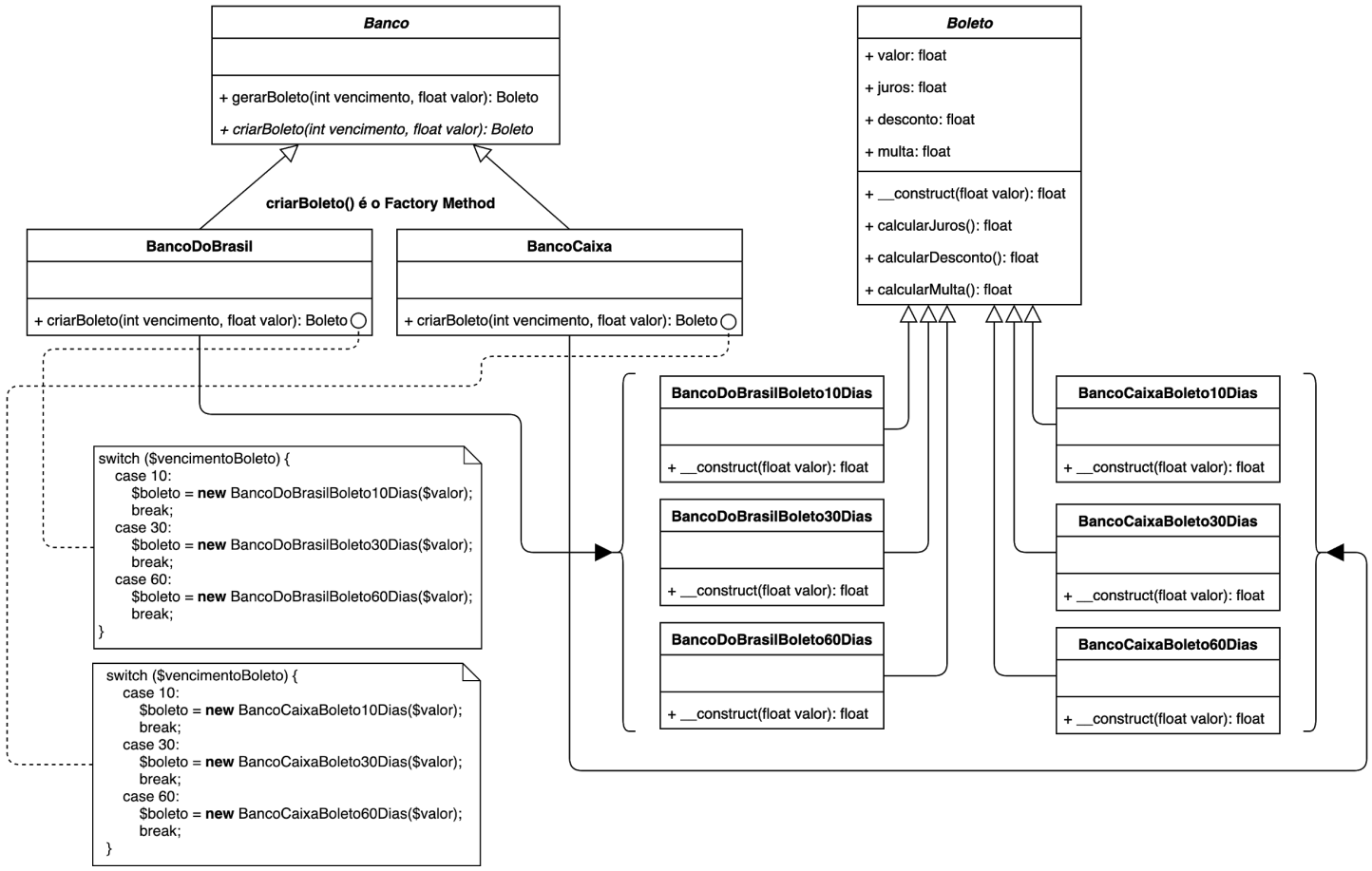


Diagrama de classes do módulo de cobranças emitindo boletos para dois bancos diferentes por meio do *Factory Method*.

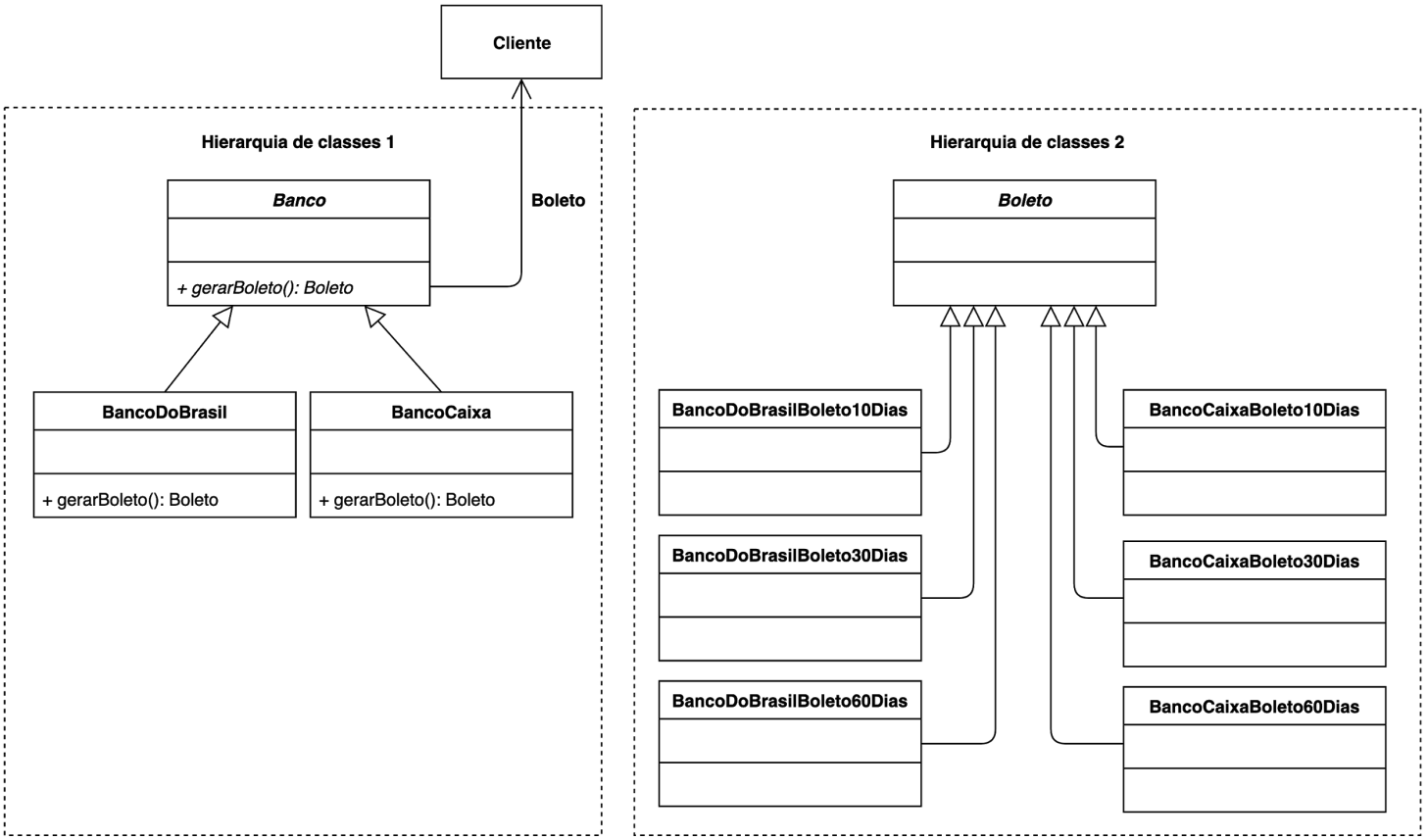
Vejamos um teste onde geramos boletos para o Banco do Brasil e para a Caixa.

```
echo "######### Caixa #########<br>"; //Apenas imprime um separador no navegador.
//Criação de uma instância de BancoCaixa. Boletos gerados serão da Caixa.
$banco = new BancoCaixa();

//Gera um BancoCaixaBoleto10Dias.
$banco->gerarBoleto(10, 100);

//Gera um BancoCaixaBoleto30Dias.
$banco->gerarBoleto(30, 100);

//Gera um BancoCaixaBoleto60Dias.
$banco->gerarBoleto(60, 100);

echo "######### Banco do Brasil #########<br>"; //Apenas imprime um separador no navegador.
//Criação de uma instância de BancoDoBrasil. Boletos gerados serão do Banco do Brasil.
$banco = new BancoDoBrasil();

//Gera um BancoDoBrasilBoleto10Dias.
$banco->gerarBoleto(10, 100);

//Gera um BancoDoBrasilBoleto30Dias.
$banco->gerarBoleto(30, 100);

//Gera um BancoDoBrasilBoleto60Dias.
$banco->gerarBoleto(60, 100);
```

Saída:

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

Boleto gerado com sucesso no valor de R$ 100
Valor Juros: R$5
Valor Desconto: R$5
Valor Multa: R$10
---------------

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

######### Banco do Brasil #########
Boleto gerado com sucesso no valor de R$ 100
Valor Juros: R$3
Valor Desconto: R$5
Valor Multa: R$2
---------------

Boleto gerado com sucesso no valor de R$ 100
Valor Juros: R$5
Valor Desconto: R$2
Valor Multa: R$50
---------------

Boleto gerado com sucesso no valor de R$ 100
Valor Juros: R$10
Valor Desconto: R$0
Valor Multa: R$15
---------------
```

Repare em nosso teste (**Cliente**) que sempre estamos chamando o método **gerarBoleto()** independe se a variável **$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*.



Destaque das hierarquias paralelas de classes em nosso exemplo. Alguns métodos e atributos foram desconsiderados.

A **hierarquia de classes 2** da imagem acima (Boleto) possibilita que a **hierarquia de classes 1** (Banco) garanta para **Cliente** que um boleto de qualquer um de seus bancos será criado a partir de um método padronizado, que é o método fábrica (*Factory Method*), no caso de nosso exemplo é o método **criarBoleto().** Deste modo temos duas hierarquias de classes distintas que trabalham juntas para garantir a consistência de criação de boletos de acordo com o banco escolhido pelo **Cliente**, ou seja, um banco da **hierarquia de classes 1** cria determinados boletos da **hierarquia de classes 2.** Para expandir o módulo de cobranças para emitir boletos de 3 bancos distintos as duas hierarquias teriam que ser expandidas.

Aplicabilidade (Quando utilizar?)

- Quando uma classe não sabe antecipar qual 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.

- **Criador:** Declara o método fábrica (*Factory Method*) o qual retorna um objeto ProdutoConcreto. Também pode definir uma implementação padrão do *factoryMethod*, para o caso de uma subclasse o omitir. Tal implementação também precisa retornar um ProdutoConcreto. O Criador também é a classe que utiliza o ProdutoConcreto retornado pelo método *factoryMethod*.

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

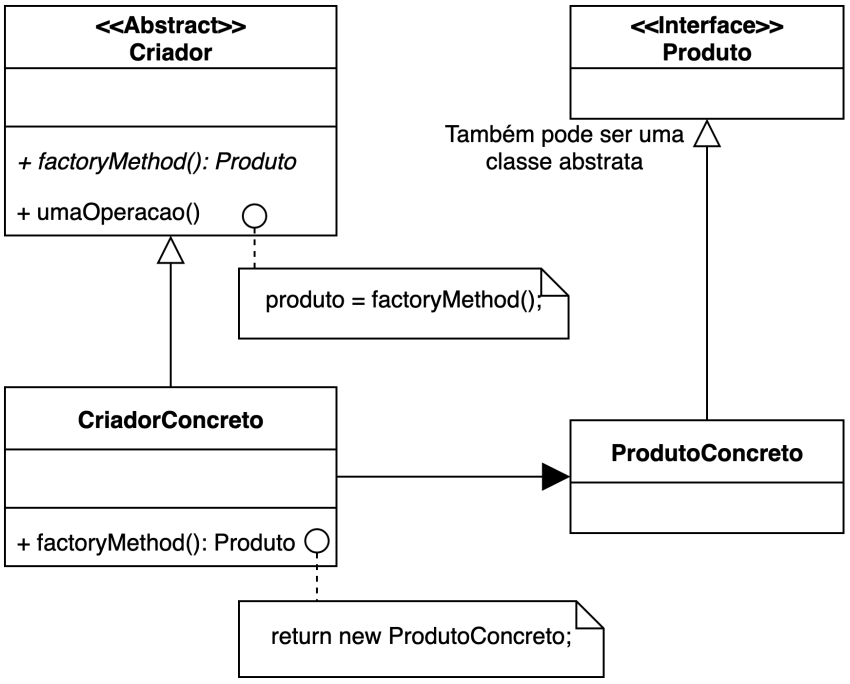


Diagrama de Classes

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.

- 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.