

Padrões de Projeto

Prof. Marum Simão Filho

Agenda

- Padrão Factory Method
- Padrão Abstract Factory
- Padrão Singleton
- Padrão Adapter
- Padrão Template-Method

Padrão Factory Method



Fábricas

Fábrica
Simples

- Simple Factory

Método
Fábrica

- Factory Method

Fábrica
Abstrata

- Abstract Factory

Programar para Interface Relembrando o Strategy

- Como definir as variáveis de instância do comportamento?

```
public class PatoSelvagem extends Pato {  
  
    public PatoSelvagem() {  
        modoDeVoar = new VoarComAsas();  
        modoDeGrasnar = new Quack();  
    }  
}
```

Com classes concretas relacionadas

```
Pato pato;
```

```
if (situacao1) {  
    pato = new PatoSelvagem();  
} else if (situacao2) {  
    pato = new PatoDeBorracha();  
} else if (situacao3) {  
    pato = new PatoCabecaVermelha();  
}
```

Quais os riscos?

- Mudanças.
- Criação de novas classes concretas apesar do polimorfismo.
- **Como encapsular do resto do aplicativo as partes que instanciam classes concretas?**

“Identificando os aspectos que variam”

```
public class Pizzaria {  
    public Pizza pedirPizza() {  
        Pizza pizza = new Pizza();  
  
        pizza.preparar();  
        pizza.assar();  
        pizza.cortar();  
        pizza.embalar();  
  
        return pizza;  
    }  
}
```



Precisamos mais que 1 Pizza

Determinando o sabor da Pizza

```
public Pizza pedirPizza(String sabor) {  
    Pizza pizza;  
  
    if (sabor.equals("muzzarela")) {  
        pizza = new PizzaMuzzarela();  
    } else if (sabor.equals("calabresa")) {  
        pizza = new PizzaCalabresa();  
    } else if (sabor.equals("mista")) {  
        pizza = new PizzaMista();  
    }  
  
    pizza.preparar();  
    pizza.assar();  
    pizza.cortar();  
    pizza.embalar();  
  
    return pizza;  
}
```

Novas classes

- Alteração no código:
 - Adicionar a Pizza Portuguesa e Frango com Catupiry.
 - Retirar a Pizza Mista do cardápio.
- Solução:
 - Retirar a criação de objetos do método pedirPizza().

Retira-se o código de criação da Pizza

```
public Pizza pedirPizza(String sabor) {  
    Pizza pizza;  
    if (sabor.equals("muzzarela")) {  
        pizza = new PizzaMuzzarela();  
    } else if (sabor.equals("calabresa")) {  
        pizza = new PizzaCalabresa();  
    } else if (sabor.equals("mista")) {  
        pizza = new PizzaMista();  
    }  
    pizza.preparar();  
    pizza.assar();  
    pizza.cortar();  
    pizza.embalar();  
  
    return pizza;  
}
```

Criando uma Fábrica

- Objetos que cuidam dos detalhes da criação de objetos.
- Qualquer criação de objeto é pedido à Fábrica.
- O único interesse do cliente é que receba um objeto do tipo que pediu.

O objeto Fábrica Simples

```
public class FabricaSimplesDePizza {  
    public Pizza criarPizza(String sabor) {  
        Pizza pizza = null;  
  
        if (sabor.equals("muzzarela")) {  
            pizza = new PizzaMuzzarela();  
        } else if (sabor.equals("calabresa")) {  
            pizza = new PizzaCalabresa();  
        } else if (sabor.equals("mista")) {  
            pizza = new PizzaMista();  
        }  
        return pizza;  
    }  
}
```

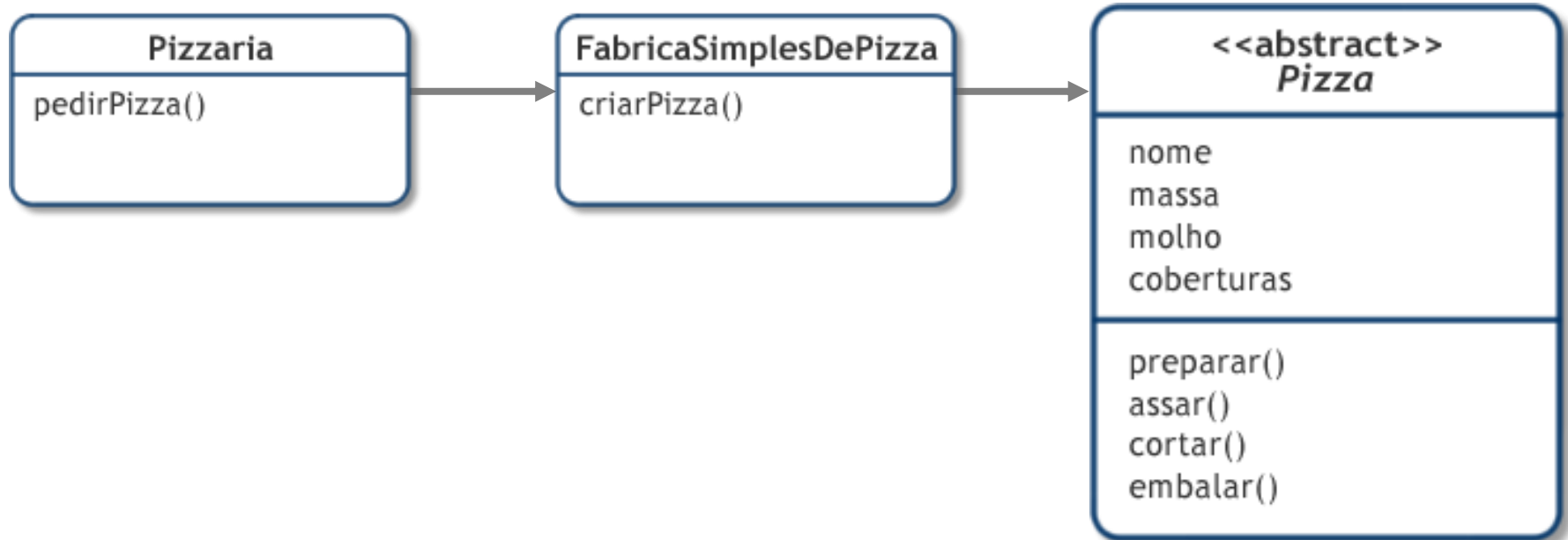
Qual a vantagem afinal?

- Um objeto fábrica pode ter muitos clientes.
- Pode haver outros métodos.
- 1 local apenas para ser modificado.

A Pizzaria refatorada

```
public class Pizzaria {  
    FabricaSimplesDePizza fabrica;  
  
    Pizzaria(FabricaSimplesDePizza fabrica) {  
        this.fabrica = fabrica;  
    }  
  
    public Pizza pedirPizza(String sabor) {  
        Pizza pizza;  
        pizza = fabrica.criarPizza(sabor);  
  
        pizza.preparar();  
        pizza.assar();  
        pizza.cortar();  
        pizza.embalar();  
  
        return pizza;  
    }  
}
```

Onde está o new()????



Pensando...

- A composição nos permite alterar um comportamento de maneira dinâmica.
 - Tempo de execução.
- Como aplicar isso em Pizzaria?

Novos objetos Fábrica

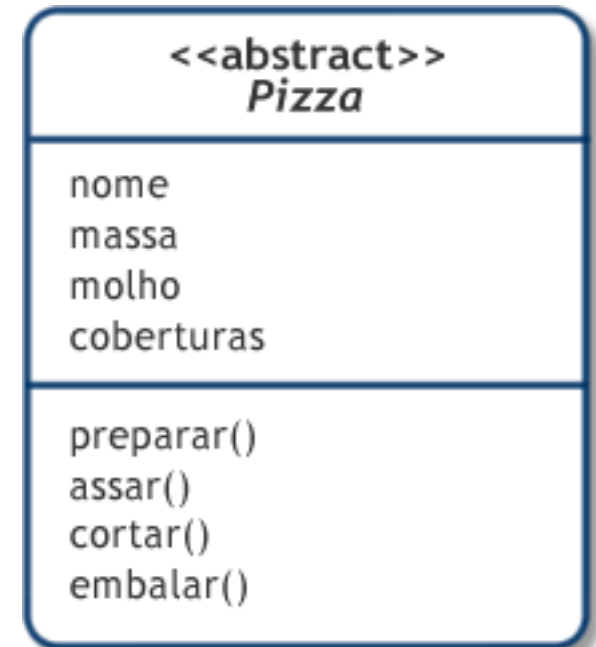
- Criação de novas fábricas para gerar pizzas especializadas.

```
FabricaDePizzaCE ceFabrica = new FabricaDePizzaCE();  
Pizzaria cePizzaria = new Pizzaria(ceFabrica);  
cePizzaria.pedirPizza("mussarela");
```

```
FabricaDePizzaSP spFabrica = new FabricaDePizzaSP();  
Pizzaria spPizzaria = new Pizzaria(spFabrica);  
spPizzaria.pedirPizza("mussarela");
```

Um objeto Pizza

```
public abstract class Pizza {  
    String nome, massa, molho;  
    ArrayList coberturas = new ArrayList();  
  
    void preparar() {  
        System.out.println("Preparando " + nome);  
        System.out.println("Assando a massa...");  
        System.out.println("Adicionando molho...");  
        System.out.println("Adicionando coberturas: ");  
        for (int i = 0; i < coberturas.size(); i++) {  
            System.out.println(" " + coberturas.get(i));  
        }  
    }  
    void assar() {  
        System.out.println("Assando por 25min a 120°");  
    }  
    void cortar() {  
        System.out.println("Fatiando a pizza em pedaços diagonais.");  
    }  
    void embalar() {  
        System.out.println("Embalando a pizza com a caixa da franquia");  
    }  
}
```



Pizzas Especializadas

Pizza de Queijo Cearense

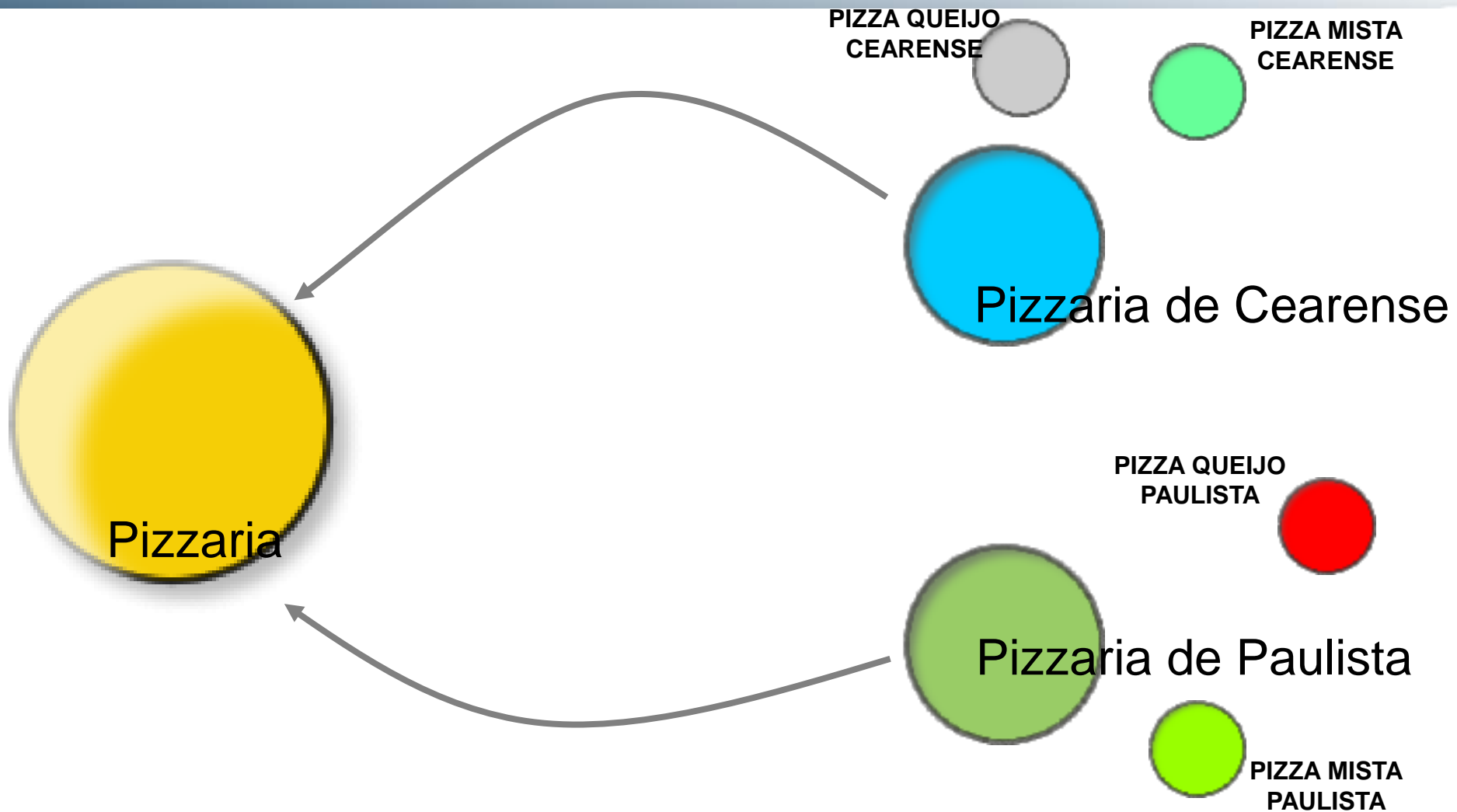
```
public class PizzaQueijoCE extends Pizza {  
  
    public PizzaQueijoCE() {  
        nome = "Pizza de Queijo Cearense";  
        massa = "Grossa";  
        molho = "Molho Apimentado ('quentinha')";  
  
        coberturas.add("Maionese");  
        coberturas.add("Catchup");  
  
    }  
}
```

Pizzas Especializadas

Pizza de Queijo Paulista

```
public class PizzaQueijoSP extends Pizza {  
    public PizzaQueijoSP() {  
        nome = "Pizza de Queijo Paulista";  
        massa = "Fina";  
        molho = "Molho Paulista";  
  
        coberturas.add("Azeite");  
    }  
  
    public void cortar() {  
        System.out.println(  
            "Cortando a pizza em quadrados");  
    }  
}
```

Repensando a Pizzaria



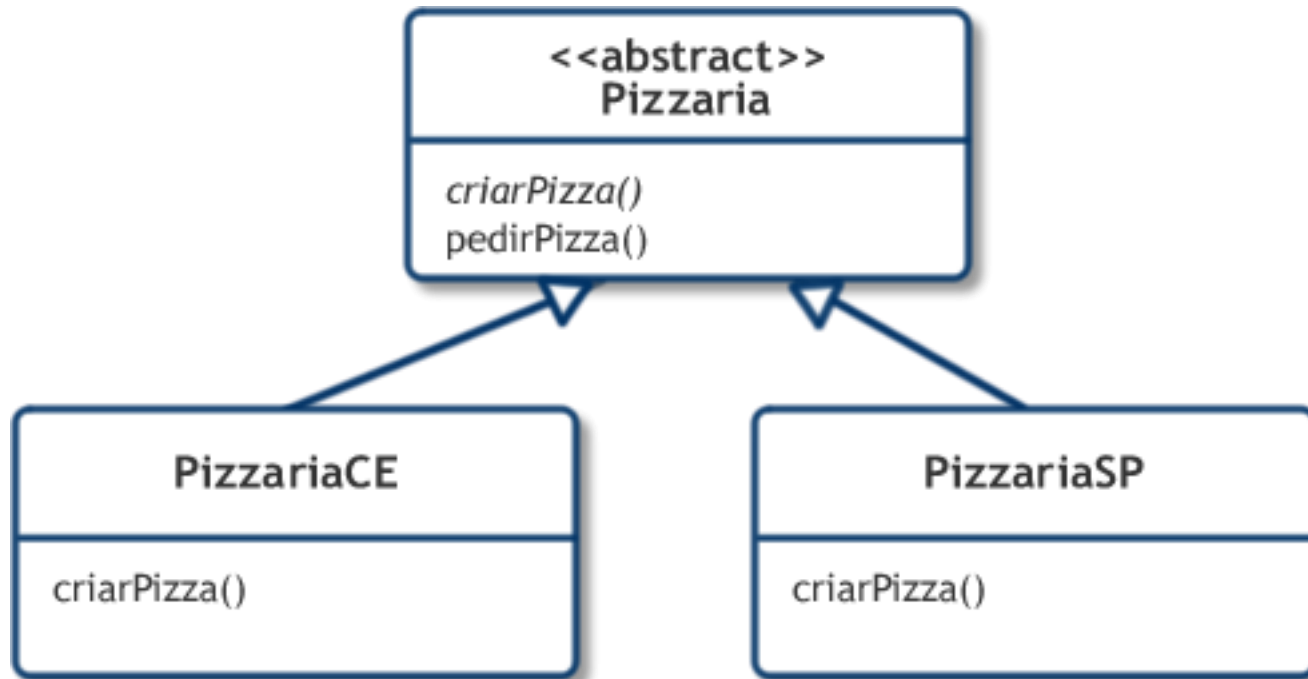
Qual o propósito?

- As pizzarias especializadas são como se fossem franquias.
 - Devem poder servir pizzas de sabores iguais.
 - Mas com o diferencial da região.
- Como são franquias...
 - O processo de pedido de pizza deve ser igual em todas as franquias.

Framework para as Pizzarias

```
public abstract class Pizzaria {  
  
    public final Pizza pedirPizza(String sabor) {  
        Pizza pizza;  
        pizza = criarPizza(sabor);  
  
        pizza.preparar();  
        pizza.assar();  
        pizza.cortar();  
        pizza.embalar();  
  
        return pizza;  
    }  
  
    abstract Pizza criarPizza(String sabor);  
}
```


Deixar as subclasses decidirem através de um método Fábrica



PizzariaCE

```
public class PizzariaCE extends Pizzaria {  
  
    public Pizza criarPizza(String sabor) {  
        Pizza pizza = null;  
  
        if (sabor.equals("queijo")) {  
            pizza = new PizzaQueijoCE();  
        } else if (sabor.equals("calabresa")) {  
            pizza = new PizzaCalabresaCE();  
        } else if (sabor.equals("mista")) {  
            pizza = new PizzaMistaCE();  
        }  
        return pizza;  
    }  
}
```

PizzariaSP

```
public class PizzariaSP extends Pizzaria {  
  
    public Pizza criarPizza(String sabor) {  
        Pizza pizza = null;  
  
        if (sabor.equals("queijo")) {  
            pizza = new PizzaQueijoSP();  
        } else if (sabor.equals("calabresa")) {  
            pizza = new PizzaCalabresaSP();  
        } else if (sabor.equals("mista")) {  
            pizza = new PizzaMistaSP();  
        }  
        return pizza;  
    }  
}
```

Método Fábrica (Factory Method)

- Criação de objetos.
- Encapsula o processamento de criação nas subclasses.

```
abstract Produto factoryMethod(String tipo)
```

Método Fábrica (Factory Method)

1

2

3

4

abstract Produto **factoryMethod**(String **tipo**)

1 Abstrato

- Subclasses responsáveis pelo processo de criação.

2 Produto

- Um método fábrica retorna um produto geralmente usado na superclasse.

3 O método

- Isola o cliente, que não sabe qual tipo concreto está sendo realmente criado.

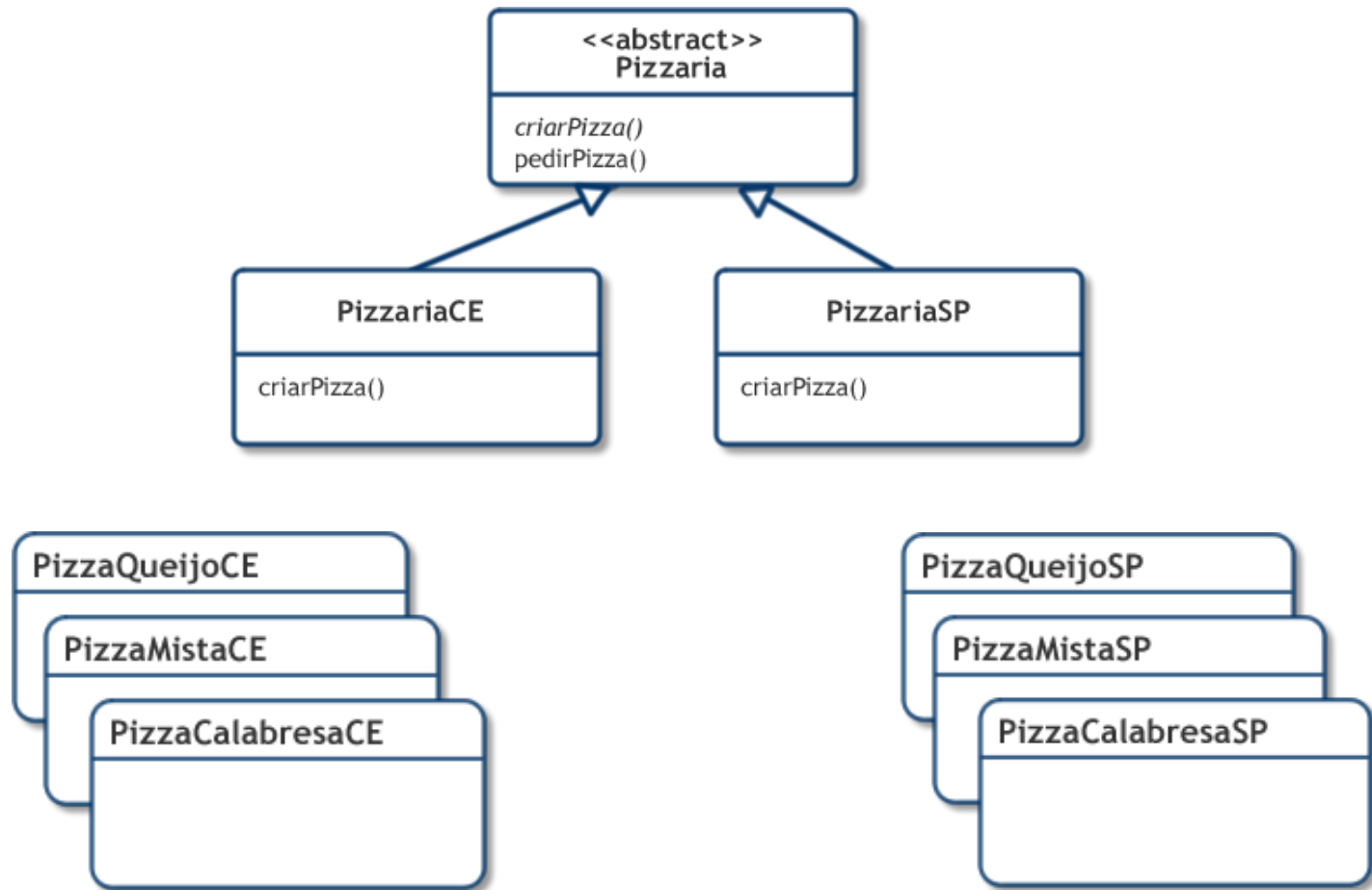
4 Parâmetro

- Pode ser parametrizado (ou não) para selecionar entre várias versões de um mesmo produto.

A pizzaria cearense

```
public class PizzariaCE extends Pizzaria {  
    Pizza criarPizza(String sabor) {  
        Pizza pizza = null;  
  
        if (sabor.equals("muzzarela")) {  
            pizza = new PizzaMuzzarelaCE();  
        } else if (sabor.equals("calabresa")) {  
            pizza = new PizzaCalabresaCE();  
        } else if (sabor.equals("mista")) {  
            pizza = new PizzaMistaCE();  
        }  
        return pizza;  
    }  
}
```

Classes de Criação e Classes Produto



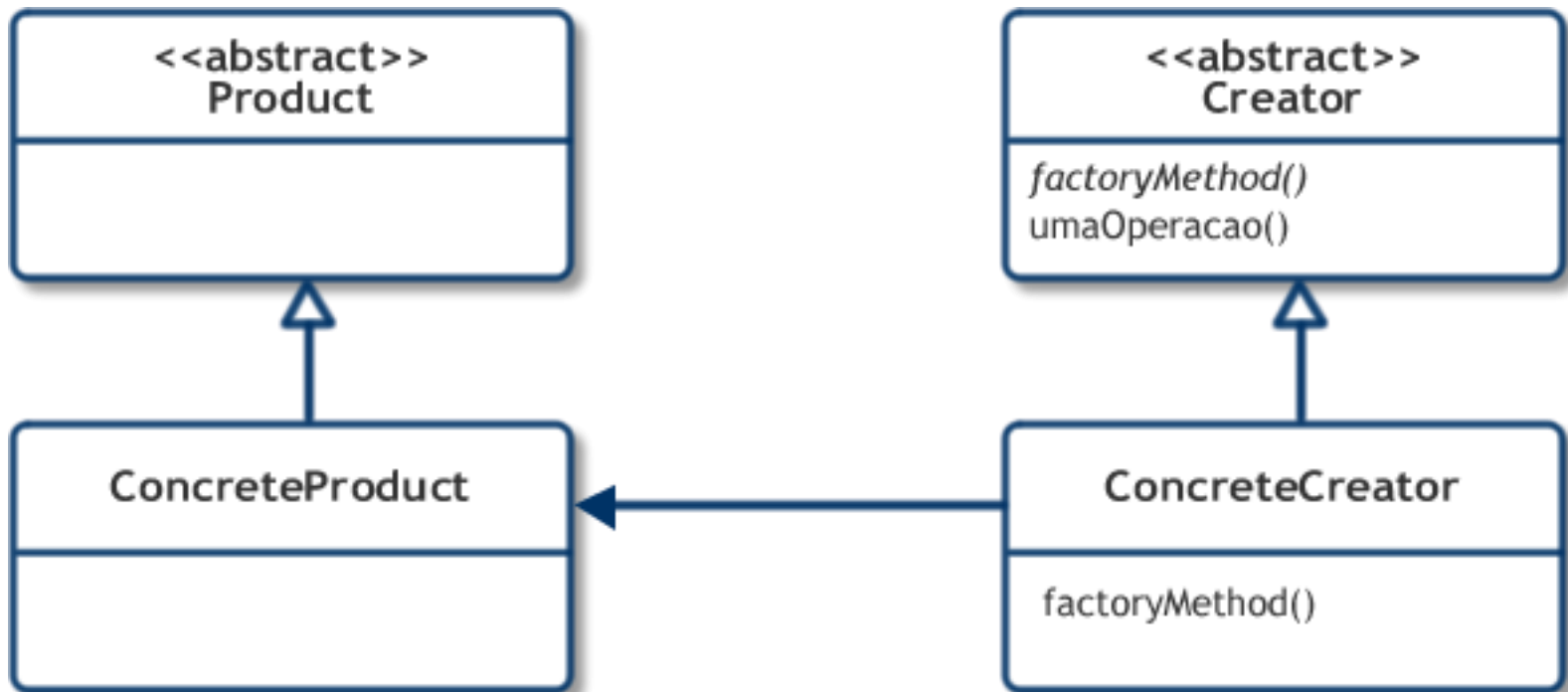
FACTORY METHOD(Método Fábrica)

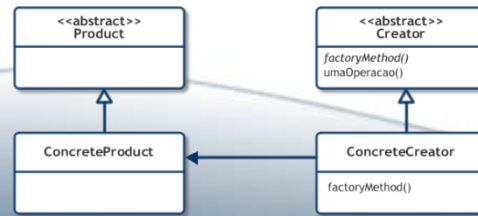
O **Padrão Factory Method** define uma interface para a **criação de objetos** mas deixa as **subclasses decidirem** qual classe instanciar. O Método Fábrica permite uma classe **delegar** a instanciação para as subclasses.

Aplicabilidade

- Uma classe não sabe, a priori, a classe dos objetos que deve criar.
- Uma classe quer que suas subclasses especifiquem os objetos que criam.

Diagrama de Classes





Participantes

■ Product

- Define a interface de objetos que o método fábrica cria.

■ ConcreteProduct

- Implementa a interface de Product.

■ Creator

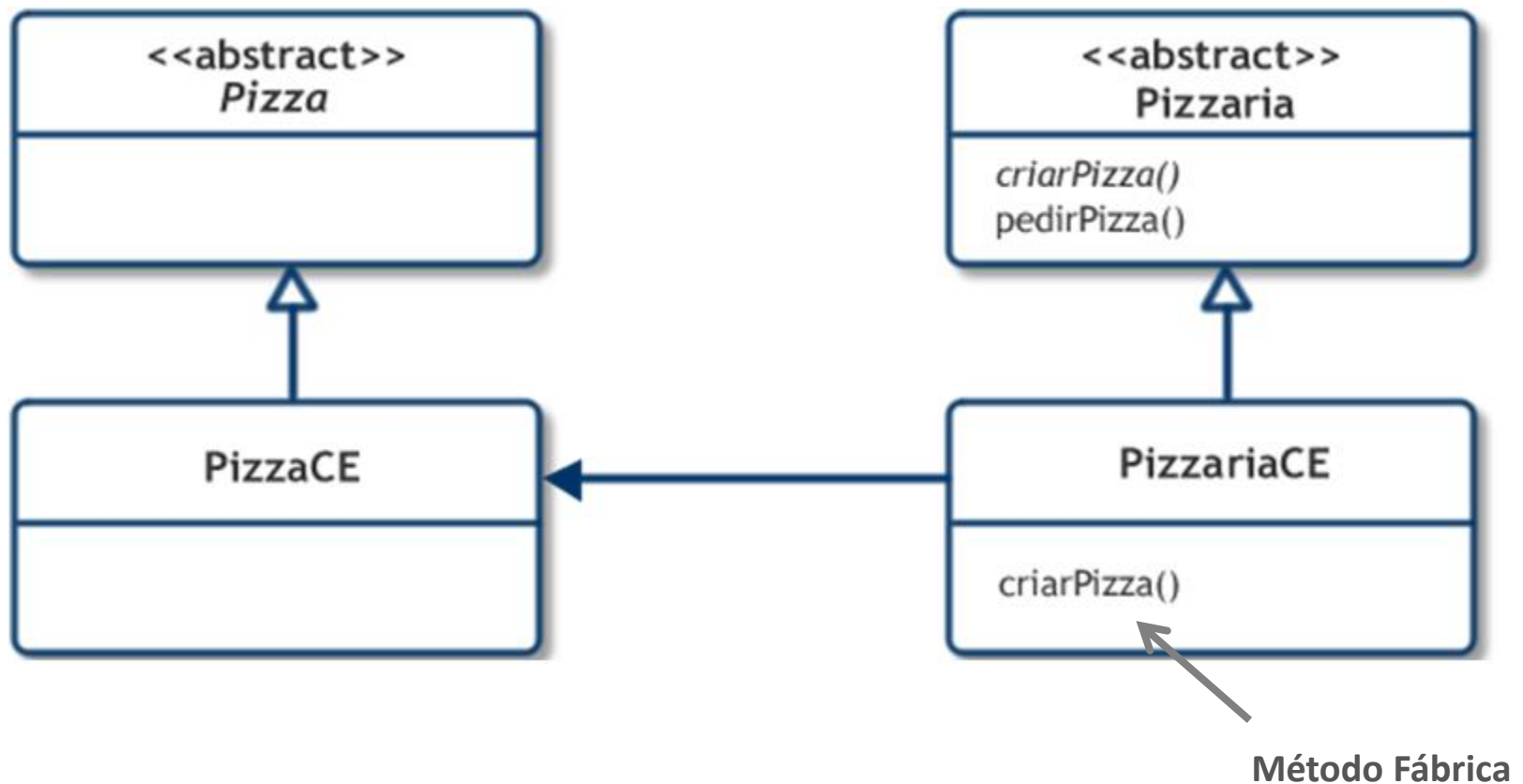
- Declara o método fábrica o qual retorna um objeto do tipo Product.
- Pode chamar o método fábrica para criar um objeto Product.

■ ConcreteCreator

- Redefine o método fábrica para retornar uma instância de um ConcreteProduct.

- Creator depende das suas subclasses para definir o método fábrica de maneira que retorne uma instância do ConcreteProduct apropriado.

Aplicando à Pizzaria...



```

public class PizzariaDependente {
    public Pizza criarPizza(String estado, String sabor) {
        Pizza pizza = null;

        if (estado.equals("CE")) {
            if (sabor.equals("queijo")) {
                pizza = new PizzaQueijoCE();
            } else if (sabor.equals("calabresa")) {
                pizza = new PizzaCalabresaCE();
            } else if (sabor.equals("mista")) {
                pizza = new PizzaMistaCE();
            }
        } else if (estado.equals("SP")) {
            if (sabor.equals("queijo")) {
                pizza = new PizzaQueijoSP();
            } else if (sabor.equals("calabresa")) {
                pizza = new PizzaCalabresaSP();
            } else if (sabor.equals("mista")) {
                pizza = new PizzaMistaSP();
            }
        } else {
            System.out.println("Erro: Sabor de pizza inválido!");
        }

        pizza.preparar();
        pizza.assar();
        pizza.cortar();
        pizza.embalar();

        return pizza;
    }
}

```

```

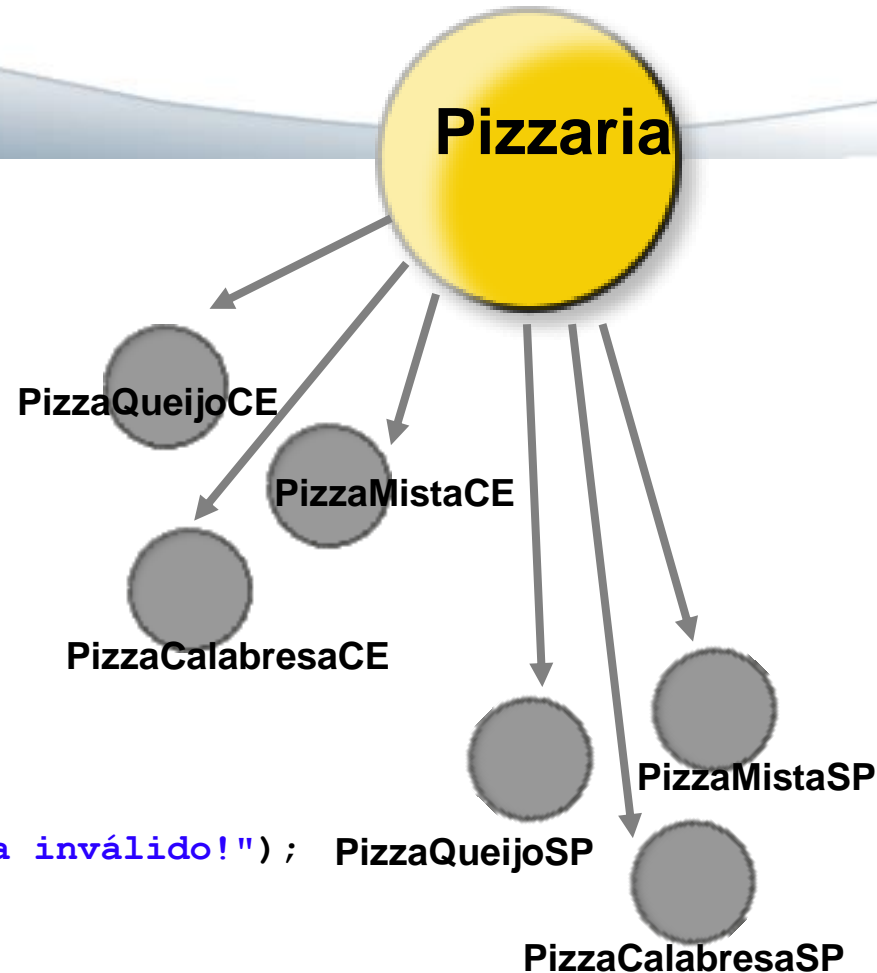
public class PizzariaDependente {
    public Pizza criarPizza(String estado, String sabor) {
        Pizza pizza = null;

        if (estado.equals("CE")) {
            if (sabor.equals("queijo")) {
                pizza = new PizzaQueijoCE();
            } else if (sabor.equals("calabresa")) {
                pizza = new PizzaCalabresaCE();
            } else if (sabor.equals("mista")) {
                pizza = new PizzaMistaCE();
            }
        } else if (estado.equals("SP")) {
            if (sabor.equals("queijo")) {
                pizza = new PizzaQueijoSP();
            } else if (sabor.equals("calabresa")) {
                pizza = new PizzaCalabresaSP();
            } else if (sabor.equals("mista")) {
                pizza = new PizzaMistaSP();
            }
        } else {
            System.out.println("Erro: Sabor de pizza inválido!");
        }

        pizza.preparar();
        pizza.assar();
        pizza.cortar();
        pizza.embalar();

        return pizza;
    }
}

```



Código refatorado!

```
public abstract class Pizzaria {  
    public final Pizza pedirPizza(String sabor) {  
        Pizza pizza;  
        pizza = criarPizza(sabor);  
        pizza.preparar();  
        pizza.assar();  
        pizza.cortar();  
        pizza.embalar();  
        return pizza;  
    }  
    abstract Pizza criarPizza(String sabor);  
}  
public class PizzariaCE extends Pizzaria {  
    Pizza criarPizza(String sabor) {  
        Pizza pizza = null;  
        if (sabor.equals("muzzarela")) {  
            pizza = new PizzaMuzzarelaCE();  
        } else if (sabor.equals("calabresa")) {  
            pizza = new PizzaCalabresaCE();  
        } else if (sabor.equals("mista")) {  
            pizza = new PizzaMistaCE();  
        }  
        return pizza;  
    }  
}
```

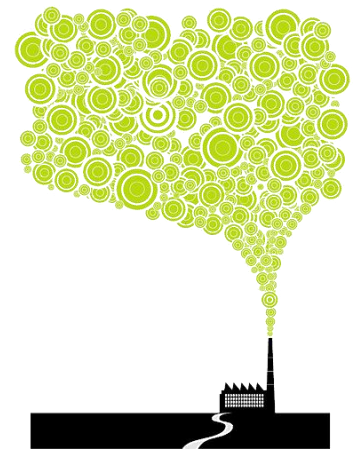

Consequências

- Eliminam a necessidade de anexar classes específicas das aplicações no código.
 - O código da aplicação lida somente com a interface de Product, portanto, ele pode trabalhar com quaisquer subclasses ConcreteProduct definidas pelo usuário.
- Fornece ganchos para subclasses.
 - Criar objetos dentro de uma classe com método fábrica é sempre mais flexível do que criar um objeto diretamente.

Consequências

- Conecta hierarquias de classe paralelas.
 - Hierarquias de classe paralelas ocorrem quando uma classe delega alguma de suas responsabilidades para uma classe separada.

Padrão Abstract Factory



Ainda sobre Pizzas...

- Imaginem que algumas franquias da nossa pizzeria comesçassem a usar ingredientes mais baratos para reduzir os custos...
- O que fazer para evitar isso?
- Vamos fornecer uma **fábrica** que produz os ingredientes para as pizzarias.

Criando famílias de produtos relacionados...

- Todas as pizzarias da franquia utilizam a mesma receita de pizza, à base de massa, molho, queijo, etc.
- O **problema** é que cada franqueado utiliza suas próprias variações de ingredientes locais.
- A **saída** para isso é assegurar um modelo de fábrica que garanta a qualidade.
- Mas como?

Criando famílias de produtos relacionados...

- Vamos definir uma **fábrica abstrata** que indica quais são os ingredientes que devem ser utilizados na criação da pizza.

```
public interface FabricaIngredientesPizza
{
    public Massa criarMassa();
    public Molho criarMolho();
    public Queijo criarQueijo();
    public Verdura [] criarVerduras();
    public Linguica criarLinguica();
    public Mariscos criarMarisco();
}
```

Criando famílias de produtos relacionados...

- E como implementar as diferenças locais?
 - Criando um fábrica para cada região
 - subclasses que implementam a interface fábrica abstrata `FabricaIngredientesPizza`.
 - Criando um conjunto de classes filhas para cada classe abstrata ingrediente.
 - Por fim, ligando as fábricas ao código da nossa pizzeria.

Criando famílias de produtos relacionados...

- E como implementar as diferenças locais?
 - Criando um fábrica para cada região

```
public class FabricaIngredientesPizzaCE implements
    FabricaIngredientesPizza
{
    public Massa criarMassa() { return new MassaGrossa(); }
    public Molho criarMolho() { return new MolhoTomate(); }
    public Queijo criarQueijo() { return new QueijoCoalho(); }
    public Verdura[] criarVerduras() {
        Verdura verduras [] = { new Alho(), new Cebola(),
                                new Pimentao(), new Tomate() };
        return verduras; }
    public Pepperoni criarLinguica(){return new Calabresa();}
    public Mariscos criarMarisco() { return new Caranguejo(); }
}
```

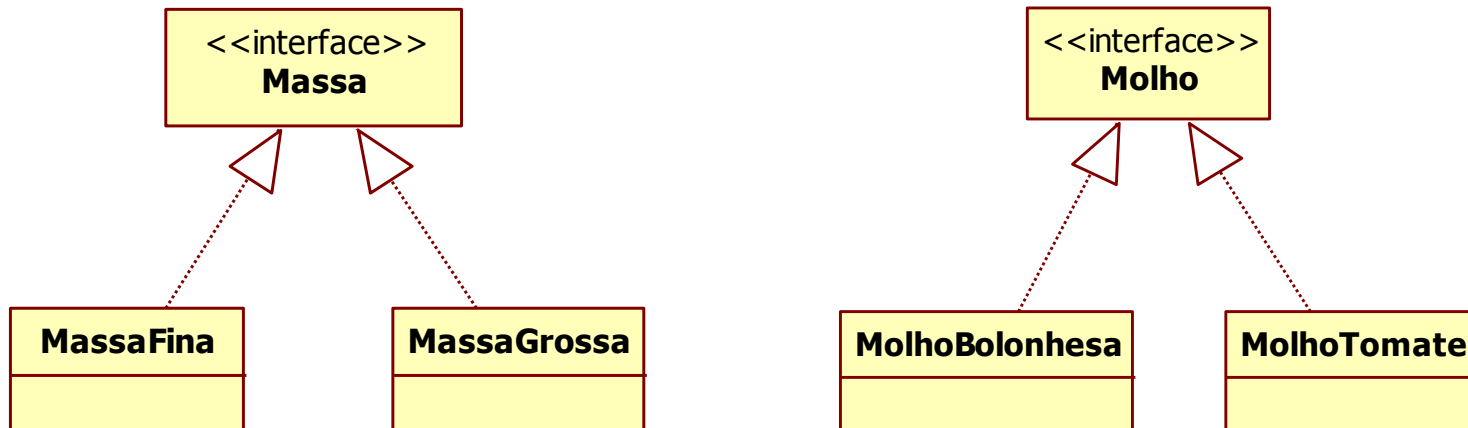

Criando famílias de produtos relacionados...

- E como implementar as diferenças locais?
 - Criando um fábrica para cada região

```
public class FabricaIngredientesPizzaSP implements
    FabricaIngredientesPizza
{
    public Massa criarMassa() { return new MassaFina(); }
    public Molho criarMolho() { return new MolhoBolhonesa(); }
    public Queijo criarQueijo() { return new QueijoMuzzarela(); }
    public Verdura[] criarVerduras() {
        Verdura verduras[] = { new AlhoPoroh(), new CebolaRoxa(),
                                new TomateSeco(), new Majericao() };
        return verduras; }
    public Pepperoni criarLinguica() { return new Pepperoni(); }
    public Mariscos criarMariscos() { return new Camarao(); }
}
```

Criando famílias de produtos relacionados...

- E como implementar as diferenças locais?
 - Criando um conjunto de classes filhas para cada classe abstrata/interface ingrediente.



Ajustando a Pizza para usar somente fábricas de ingredientes

```
public abstract class Pizza {  
    String nome;  
    Massa massa;  
    Molho molho;  
    Queijo queijo;  
    Verdura [] verduras;  
    Calabresa linguica;  
    Mariscos mariscos;  
  
    void assar() {  
        System.out.println("Assando por 25min a 120°");  
    }  
    void cortar() {  
        System.out.println("Fatiando a pizza em pedaços diagonais.");  
    }  
    void embalar() {  
        System.out.println("Embalando a pizza com a caixa da franquia");  
    }  
    abstract void preparar();  
}
```

As Pizzas concretas implementam a Pizza abstrata

```
public class PizzaDeQueijo implements Pizza
{
    nome = "Pizza de Queijo";
    FabricaIngredientesPizza fabricaIngredientes;
    public PizzaDeQueijo (FabricaIngredientesPizza fabricaIngredientes)
    {
        this.fabricaIngredientes = fabricaIngredientes;
    }
    void preparar()
    {
        System.out.println("Preparando " + nome);
        massa = fabricaIngredientes.criarMassa();
        molho = fabricaIngredientes.criarMolho();
        queijo = fabricaIngredientes.criarQueijo();
    }
}
```

As Pizzas concretas implementam a Pizza abstrata

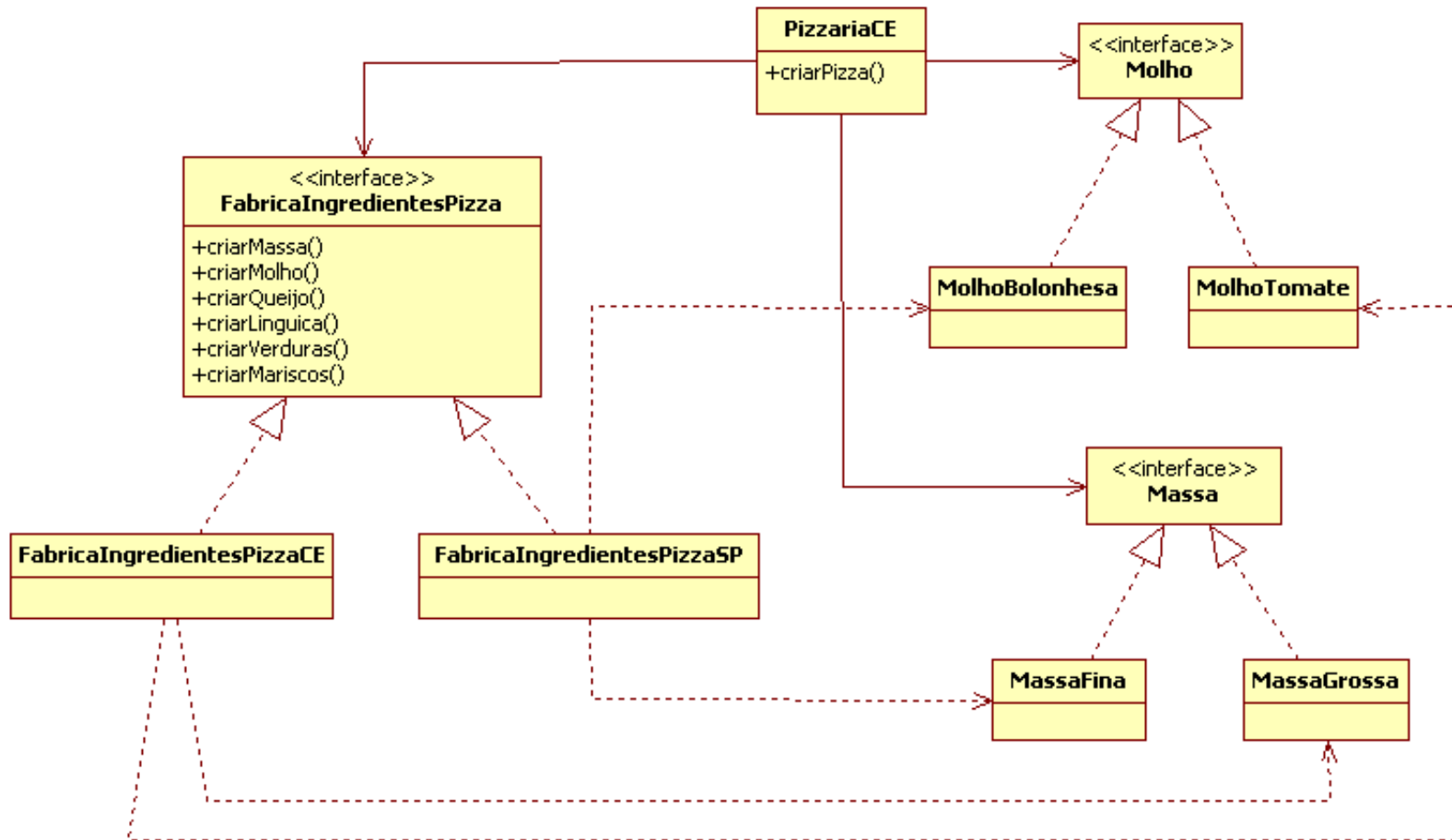
```
public class PizzaDeCalabresa implements Pizza
{
    nome = "Pizza de Calabresa";
    FabricaIngredientesPizza fabricaIngredientes;
    public PizzaDeCalabresa(FabricaIngredientesPizza fabricaIngredientes)
    {
        this.fabricaIngredientes = fabricaIngredientes;
    }
    void preparar()
    {
        System.out.println("Preparando " + nome);
        massa = fabricaIngredientes.criarMassa();
        molho = fabricaIngredientes.criarMolho();
        queijo = fabricaIngredientes.criarQueijo();
        linguica = fabricaIngredientes.criarLinguica();
    }
}
```

As Pizzarias agora teriam a seguinte forma...

- E como implementar as diferenças locais?
 - Por fim, ligando as fábricas ao código da nossa pizzaria.

```
public class PizzariaCE extends Pizzaria {
    protected Pizza criarPizza(String sabor) {
        Pizza pizza = null;
        FabricaIngredientesPizza fabricaIngredientes =
            new FabricaIngredientesPizzaCE();
        if (sabor.equals("queijo")) {
            pizza = new PizzaDeQueijo(fabricaIngredientes);
        } else if (sabor.equals("calabresa")) {
            pizza = new PizzaDeCalabresa(fabricaIngredientes);
        } else if (sabor.equals("caranguejo")) {
            pizza = new PizzaDeCaranguejo(fabricaIngredientes);
        } ...
        return pizza;
    }
}
```

Diagrama...



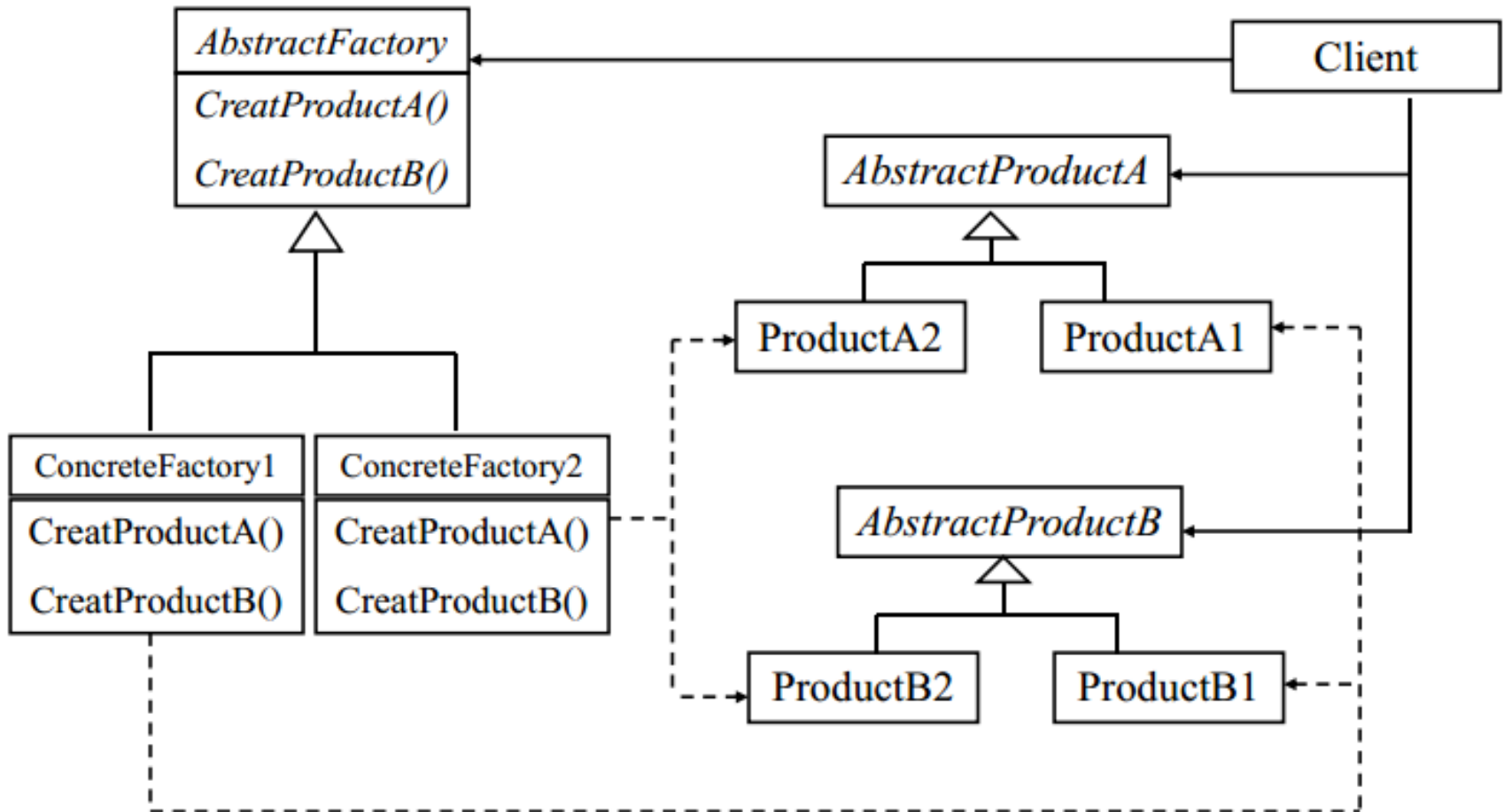
ABSTRACT FACTORY (Fábrica Abstract)

O **Padrão Abstract Factory** provê uma interface para criar uma família de objetos relacionados ou dependentes sem especificar suas classes concretas.

Aplicabilidade

- Quando um sistema deve ser independente de como seus produtos são criados, compostos e representados.
- Quando um sistema deve ser configurado com uma entre várias famílias de produtos.
- Quando uma família de produtos relacionados foi projetada para uso conjunto e você deve implementar essa restrição.
- Quando você quer fornecer uma biblioteca de classes e quer revelar sua interface e não sua implementação.
 - Não permita portanto que objetos sejam diretamente criados com new.

Diagrama de Classes



Participantes

- **AbstractFactory** ([FabricaIngredientesPizza](#))
 - Define uma interface para as operações que criam objetos produtos abstratos.
- **ConcreteFactory** ([FabricaIngredientesPizzaCE](#), [FabricaIngredientesPizzaSP](#))
 - Implementa as operações para criar objetos produtos concretos.
- **AbstractProduct** ([Massa](#), [Molho](#), [Queijo](#))
 - Declara uma interface para um tipo de objeto produto

Participantes

- **Product** (MassaFina, MassaGrossa, MolhoTomate, MolhoBolonhesa)
 - Define um objeto produto a ser criado pela ConcreteFactory correspondente.
 - Implementa a interface de AbstractProduct.
- **Client** (PizzariaCE, PizzariaSP)
 - Usa somente as interfaces definidas por AbstractFactory e AbstractProduct.

Colaborações

- Normalmente uma única instância de uma classe ConcreteFactory é criada em tempo de execução.
- Essa ConcreteFactory cria objetos com uma implementação particular.
- Para criar produtos diferentes, clientes devem usar uma ConcreteFactory diferente.
- AbstractFactory depende de suas subclasses ConcreteFactory para criar objetos de produtos.

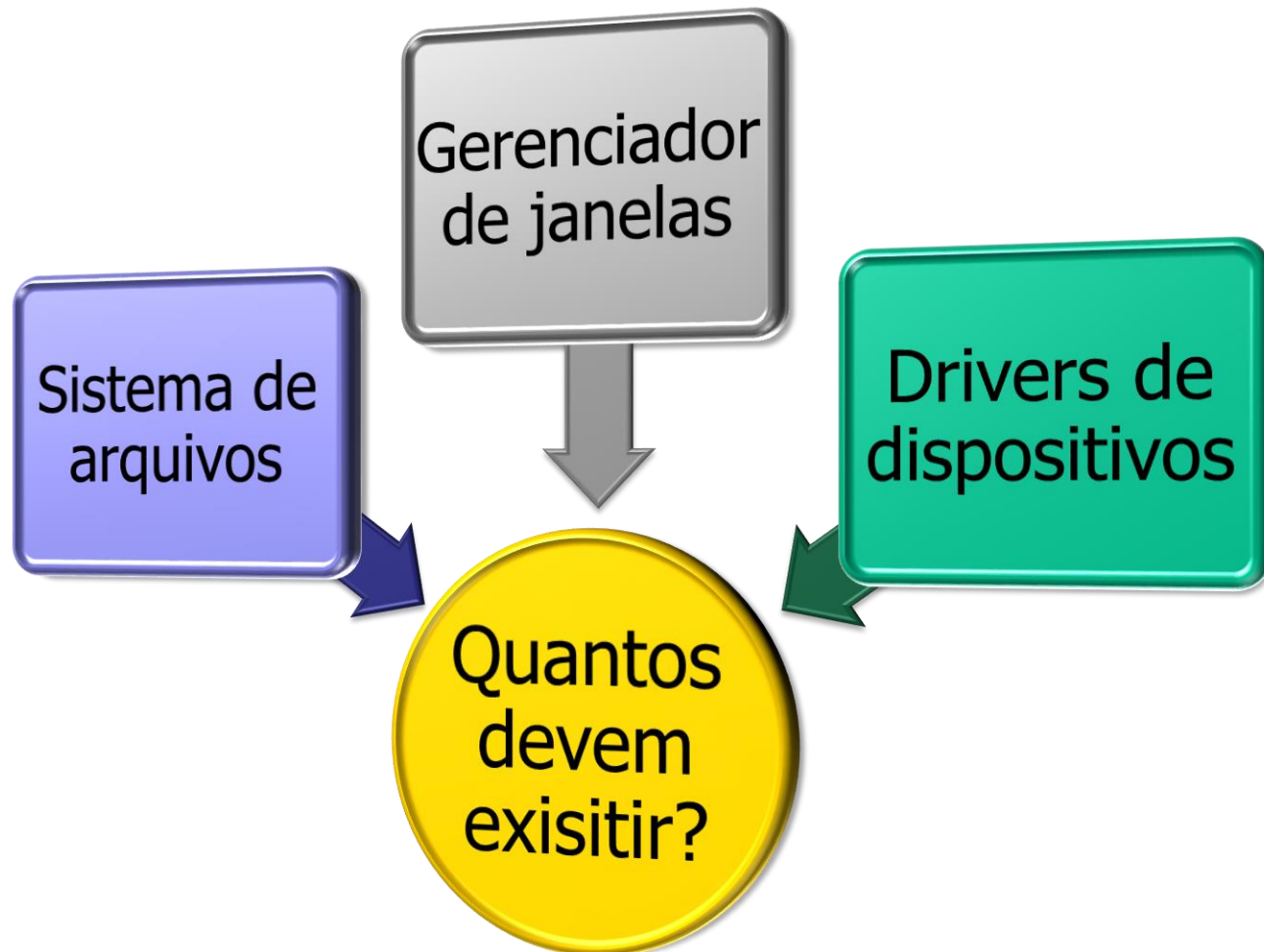
Consequências

- O padrão Abstract Factory isola as classes concretas dos clientes.
- Facilita a troca de famílias de produtos
 - Basta trocar uma linha do código pois a criação da fábrica concreta aparece em um único ponto do programa).
- Promove a consistência de produtos
 - Não há possibilidade de misturar objetos de famílias diferentes.
- Dificulta a criação de novos produtos diferentes
 - Pois temos que modificar a fábrica abstrata e todas as fábricas concretas.

Padrão Singleton



Motivação



- Contextos em que só deva existir um único objeto de uma classe.
- A existência de mais de uma instância de certos objetos pode causar problemas na execução de um programa.
- Importante quando um determinado objeto possui muitos recursos.

Singleton

Motivação



Global

Acesso único

Uma maneira de garantir que haja somente um único objeto de uma determinada classe

Criando um objeto

- Como fazer para criar um único objeto?
 - `new MeuObjeto();`
- É possível criar outro?
 - Sim
- Podemos fazer o que se segue?

Singleton

```
public class MeuObjeto {  
  
    private MeuObjeto() {}  
  
}
```

- O que significa isso?
- Existe algum código que possa usar este construtor?

Singleton

```
public class MeuObjeto {  
  
    public static MeuObjeto getInstance() {}  
  
}
```

■ O que isto significa?

```
MeuObjeto.getInstance();
```

Singleton

- Juntando as 2 coisas

```
public class MeuObjeto {  
  
    private MeuObjeto() {}  
  
    public static MeuObjeto getInstance() {  
        return new MeuObjeto();  
    }  
  
}
```

Singleton

- Uma outra forma de instanciar um objeto

`MeuObjeto.getInstance() ;`

Exercício - 10min

- Altere a classe abaixo de forma que ela só possa criar uma única instância da classe.

```
public class MeuObjeto {  
  
    private MeuObjeto() {}  
  
    public static MeuObjeto getInstance() {  
        return new MeuObjeto();  
    }  
  
}
```


Singleton

Implementação clássica

```
public class Singleton {  
    private static Singleton instanciaUnica;  
  
    private Singleton() {}  
  
    public static Singleton getInstance() {  
        if (instanciaUnica == null) {  
            instanciaUnica = new Singleton();  
        }  
        return instanciaUnica;  
    }  
}
```

Singleton

Implementação clássica

```
public class Singleton {  
    private static Singleton instanciaUnica;  
  
    private Singleton() {}  
  
    public static Singleton getInstance() {  
        if (instanciaUnica == null) {  
            instanciaUnica = new Singleton();  
        }  
        return instanciaUnica;  
    }  
}
```

- Uma variável estática para garantir uma única instância para toda a classe.
- Um construtor privado que somente a própria classe tem acesso.
- O método getInstance instancia o objeto único e o retorna.
- Pode haver outros métodos?
 - Sim.

+1 Padrão SINGLETON

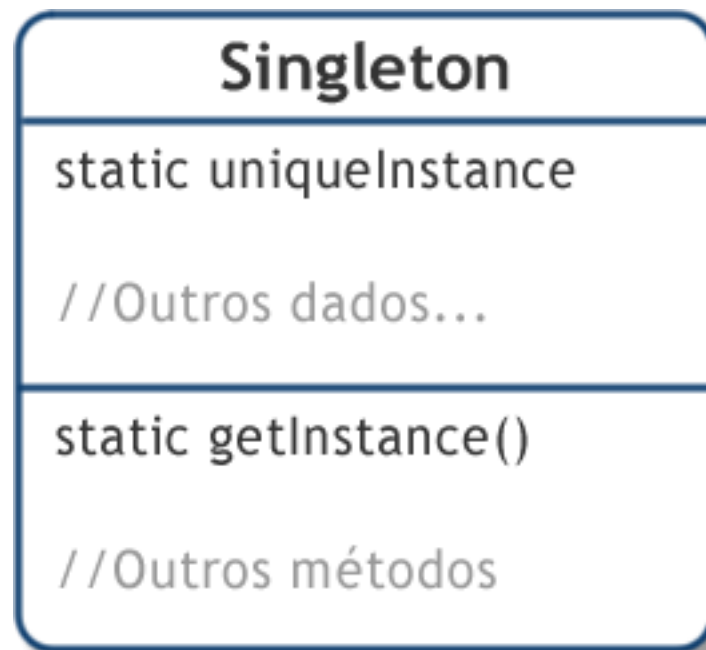
O **Padrão Singleton** garante que uma classe possua apenas uma **única instância** e fornece um **ponto global** de acesso a ela.

- A classe Singleton gerencia sua única instância.
 - Nenhuma outra classe pode criar uma nova instância de uma classe Singleton.
 - É preciso fazer uso da própria classe para criá-la.
- Ponto de acesso global.
 - A própria classe.

Aplicabilidade

- Quando for necessário existir apenas uma **única instância** de uma classe, e essa instância deve dar **acesso** aos clientes através de **um ponto bem conhecido**.
- Quando a **única instância** tiver de ser **extensível através de subclasses**, possibilitando aos clientes usarem uma instância estendida sem alterar o seu código.

Diagrama de classes



■ Singleton

- Define uma operação **getInstance()** que permite aos clientes acessarem sua única instância. **getInstance** é uma **operação de classe**, ou seja **estática**.

Colaborações

- Os clientes acessam uma instância Singleton unicamente pela operação getInstance do Singleton.

Caldeira de chocolate



- Pode-se:
 - Encher a caldeira de chocolate.
 - Somente se ela estiver vazia.
 - Ferver o chocolate existente.
 - Deve estar cheia mas não pode está fervida.
 - Drenar o chocolate.
 - Deve estar cheia e fervida.

Uma caldeira de Chocolate

```
public class CaldeiraDeChocolate {  
    private boolean vazia;  
    private boolean fervida;  
  
    public CaldeiraDeChocolate() {  
        vazia = true;  
        fervida = false;  
    }  
  
    public void encher() {  
        if (estaVazia()) {  
            vazia = false;  
            fervida = false;  
            //encher de chocolate  
        }  
    }  
  
    public void drenar() {  
        if (!estaVazia() && estaFervida()) {  
            //drene o chocolate  
            vazia = true;  
        }  
    }  
}
```

```
    public void ferver() {  
        if (!estaVazia() && !estaFervida()) {  
            //ferver  
            fervida = true;  
        }  
    }  
  
    public boolean estaVazia() {  
        return vazia;  
    }  
  
    public boolean estaFervida() {  
        return fervida;  
    }  
}
```

Como transformar a caldeira em um Singleton???

```
public class CaldeiraDeChocolate {
```

```
    private static CaldeiraDeChocolate unicaInstancia;
```

```
    ...
```

```
    private CaldeiraDeChocolate() {  
        vazia = true;  
        fervida = false;  
    }
```

```
    public static CaldeiraDeChocolate getInstance() {  
        if (unicaInstancia == null) {  
            unicaInstancia = new CaldeiraDeChocolate();  
        }  
        return unicaInstancia;  
    }
```

```
    ...
```

```
}
```

Ainda existe uma forma de termos mais de um objeto a partir do Singleton??

Threads

Chocolate & Threads

■ Thread 1	■ Thread 2	valor
<pre>public static CaldeiraDeChocolate getInstance() { if (unicaInstancia == null) { unicaInstancia = new CaldeiraDeChocolate(); } return unicaInstancia; }</pre>	<pre>public static CaldeiraDeChocolate getInstance() { if (unicaInstancia == null) { unicaInstancia = new CaldeiraDeChocolate(); } return unicaInstancia; }</pre>	<p><i>unicaInstancia</i></p> <p>null</p> <p> </p> <p>null</p> <p> </p> <p>null</p> <p><objeto1></p> <p> </p> <p><objeto2></p> <p> </p> <p><objeto2></p>

Chocolate & Threads

- Como resolver → Sincronização.
- Transformar getInstance em um método sincronizado.

```
public static synchronized CaldeiraDeChocolate getInstance() {  
    if (unicaInstancia == null) {  
        unicaInstancia = new CaldeiraDeChocolate();  
    }  
  
    return unicaInstancia;  
}
```

Formas mais rápidas

```
public class Singleton {  
    private static Singleton instanciaUnica = new Singleton();  
  
    private Singleton() {}  
  
    public static Singleton getInstance() {  
        return instanciaUnica;  
    }  
}
```

Formas mais rápidas Inicializador estático

```
public class Singleton {  
    private static Singleton instanciaUnica;  
  
    static {  
        instanciaUnica = new Singleton();  
    }  
  
    private Singleton() {}  
  
    public static Singleton getInstance() {  
        return instanciaUnica;  
    }  
}
```


Consequências

- Acesso controlado a uma instância única.
 - Permite controle total sobre como e quando os clientes acessam a instância única.
- Pode permitir um número variado de instâncias.
 - É fácil mudar de idéia, permitindo mais de uma instância da classe Singleton. Pode-se usar a mesma abordagem para controlar o número de instâncias que a aplicação utiliza. Basta alterar a operação que permite acesso à instância de Singleton.

Consequências

- Permite um refinamento de operações e da representação.
 - Singleton pode ter subclasses e é fácil configurar uma aplicação com uma instância dessa classe estendida.
- Espaço de nomes reduzidos.
 - Representa uma melhoria em relação ao uso de variáveis globais. Evita a poluição do espaço de nomes com variáveis globais que armazenam instâncias únicas.

Padrão Adapter



Padrão Adapter



Pato.java

```
public interface Pato {  
    public void grasnar();  
    public void voar();  
}
```

Subclasse de Pato

PatoSelvagem.java

```
public class PatoSelvagem implements Pato {  
  
    public void grasnar() {  
        System.out.println("Quack");  
    }  
  
    public void voar() {  
        System.out.println("Estou voando");  
    }  
}
```

```
public interface Peru {  
    public void gorgolejar();  
    public void voar();  
}
```

Subclasse de Peru

PeruSelvagem

```
public class PeruSelvagem implements Peru {  
  
    public void gorgolejar() {  
        System.out.println("Glu! Glu!");  
    }  
  
    public void voar() {  
        System.out.println(  
            "Estou voando uma pequena distância");  
    }  
  
}
```


Problemas

- Em uma determinada situação, você tem um objeto Peru em tempo de execução mas precisa de um Pato.
 - Pontos favoráveis:
 - Você já tem seu sistema orientado à interface e não à implementação.
 - Suas variáveis de referência são do tipo da interface Pato.

PeruAdapter

```
public class PeruAdapter implements Pato {  
  
    Peru peru;  
  
    public PeruAdapter(Peru peru) {  
        this.peru = peru;  
    }  
  
    public void grasnar() {  
        peru.gorgolejar();  
    }  
  
    public void voar() {  
        for (int i = 0; i < 5; i++) {  
            peru.voar();  
        }  
    }  
}
```

Teste para Pato e Peru

```
public class PatoTeste {  
  
    public static void testarPato(Pato p) {  
        p.grasnar();  
        p.voar();  
    }  
  
    public static void main(String[] args) {  
        PatoSelvagem pato = new PatoSelvagem();  
  
        PeruSelvagem peru = new PeruSelvagem();  
        Pato peruAdaptado = new PeruAdapter(peru);  
  
        System.out.println("O peru faz...");  
        peru.gorgolejar();  
        peru.voar();  
  
        System.out.println("\nO pato faz...");  
        testarPato(pato);  
  
        System.out.println("\nO Peru Adaptado faz...");  
        testarPato(peruAdaptado);  
    }  
}
```

Explicando o Adapter

Um cliente implementado para uma interface específica (interface-alvo)



Interface adaptada



Interface alvo

O adaptador implementa a interface-alvo e possui uma instância do adaptado

requisicaoTraduzida()

requisicao()



Explicando

- 1** O Cliente faz uma solicitação ao adaptador chamando um método dele através da interface-alvo.
- 2** O adaptador traduz a solicitação para uma ou mais chamadas de métodos no objeto adaptado usando a interface desse objeto.
- 3** O cliente recebe os resultados da chamada sem jamais perceber que há um adaptador fazendo a tradução.

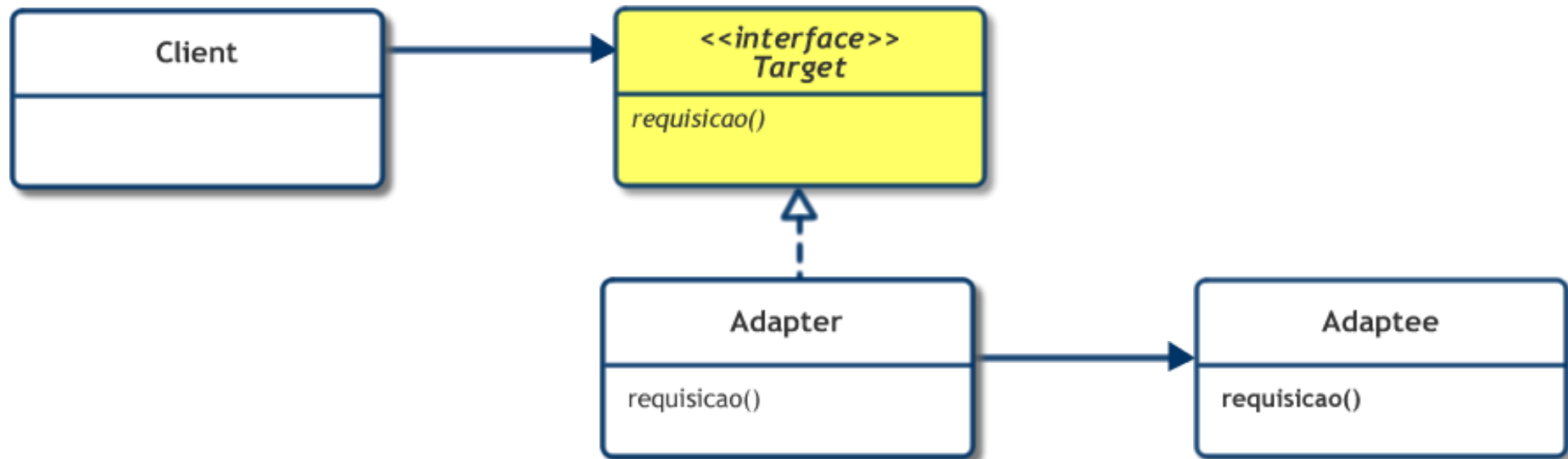
+1 Padrão ADAPTER

O **Padrão Adapter** converte a interface de uma classe em **outra** que o cliente espera. O **Adapter** permite que classes com **interfaces incompatíveis** trabalhem em **conjunto** – o que, de outra forma, seria impossível.

Aplicabilidade

- Necessidade de usar uma classe existente, mas sua interface não corresponde à interface necessária.
- Necessidade de criar uma classe reutilizável com classes não relacionadas ou não previstas, ou seja, classes que não necessariamente possuam interfaces compatíveis.
- Necessidade de usar várias subclasses existentes, porém, for impraticável adaptar estas interfaces criando subclasses para cada uma. Um adaptador pode adaptar a interface da sua classe mãe.

Diagrama de classes



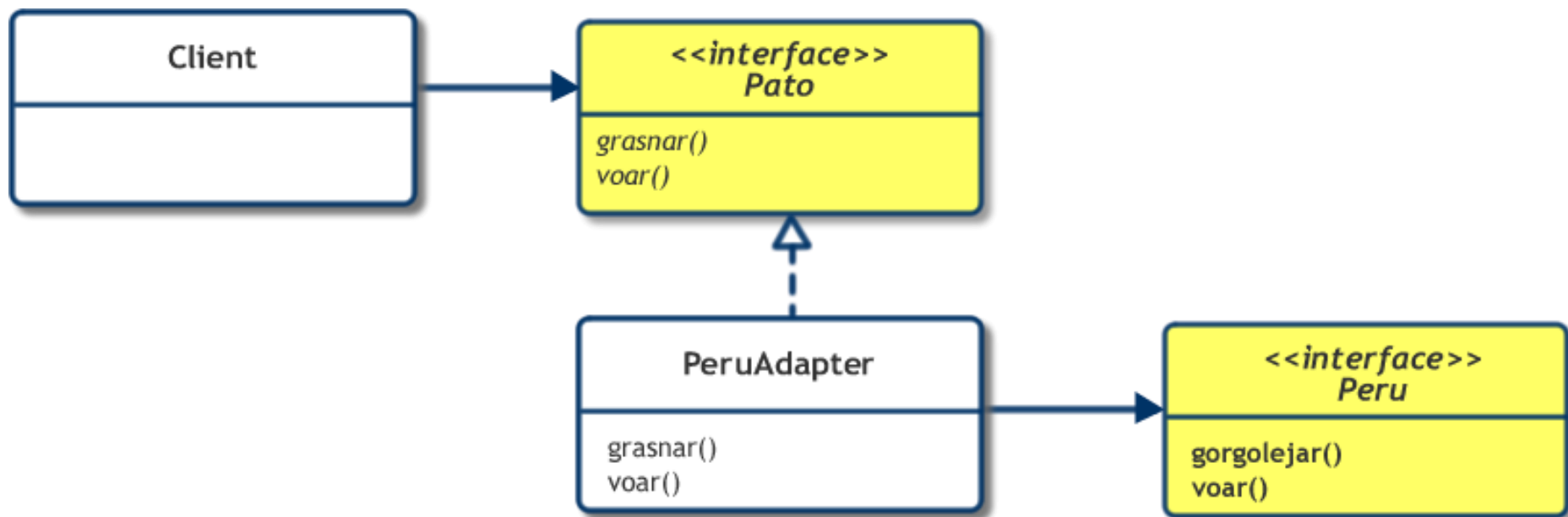
Participantes

- **Target** (Alvo)
 - Define a interface específica que o **Client** usa.
- **Client**
 - Colabora com objetos compatíveis com a interface **Target**.
- **Adaptee** (Objeto a ser adaptado)
 - Define uma **interface** existente que necessita ser adaptada.
- **Adapter**
 - **Adapta** a interface do **Adaptee** à interface do **Target**.

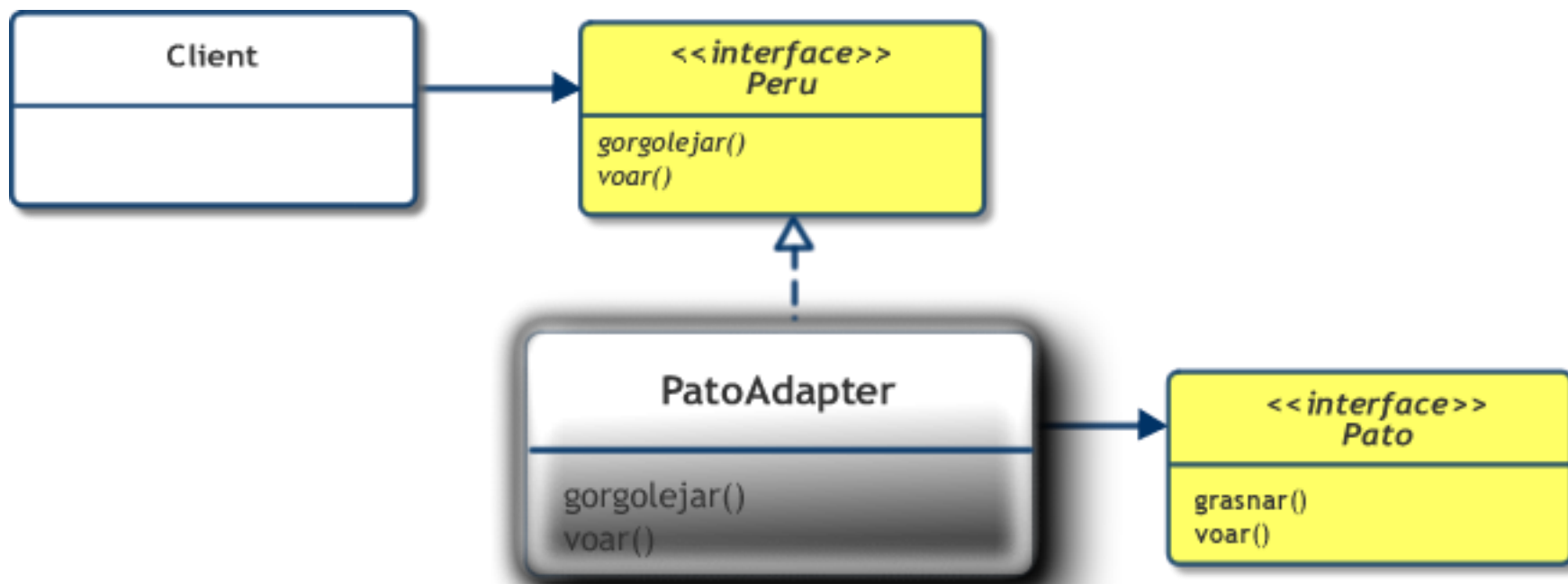
Colaborações

- Os clientes chamam operações em uma instância de Adapter.
- Por sua vez, o Adapter chama operações de Adaptee que executam a solicitação.

No exemplo do Pato



Exercício PatoAdapter



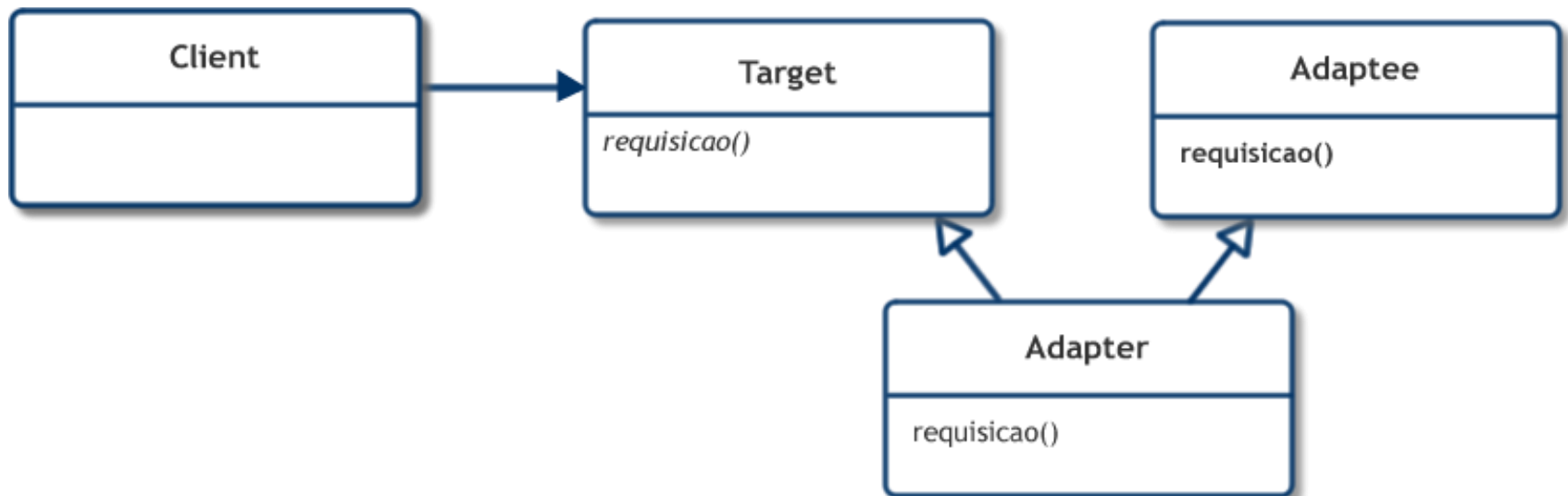
PatoAdapter.java

```
public class PatoAdapter implements Peru {  
  
    Pato pato;  
  
    public PatoAdapter(Pato pato) {  
        this.pato = pato;  
    }  
  
    public void gorgolejar() {  
        pato.grasnar();  
    }  
  
    public void voar() {  
        pato.voar();  
    }  
}
```

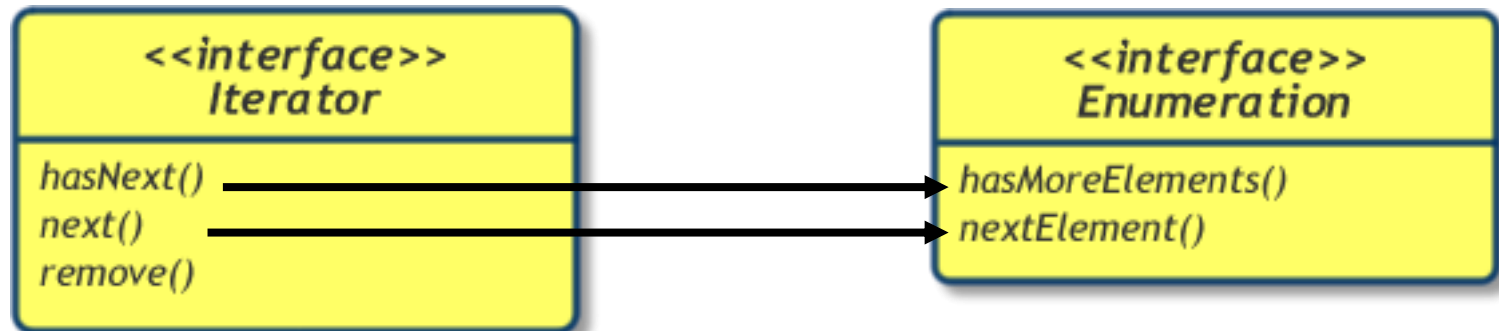
Adaptadores de Objeto

Adaptadores de Classe

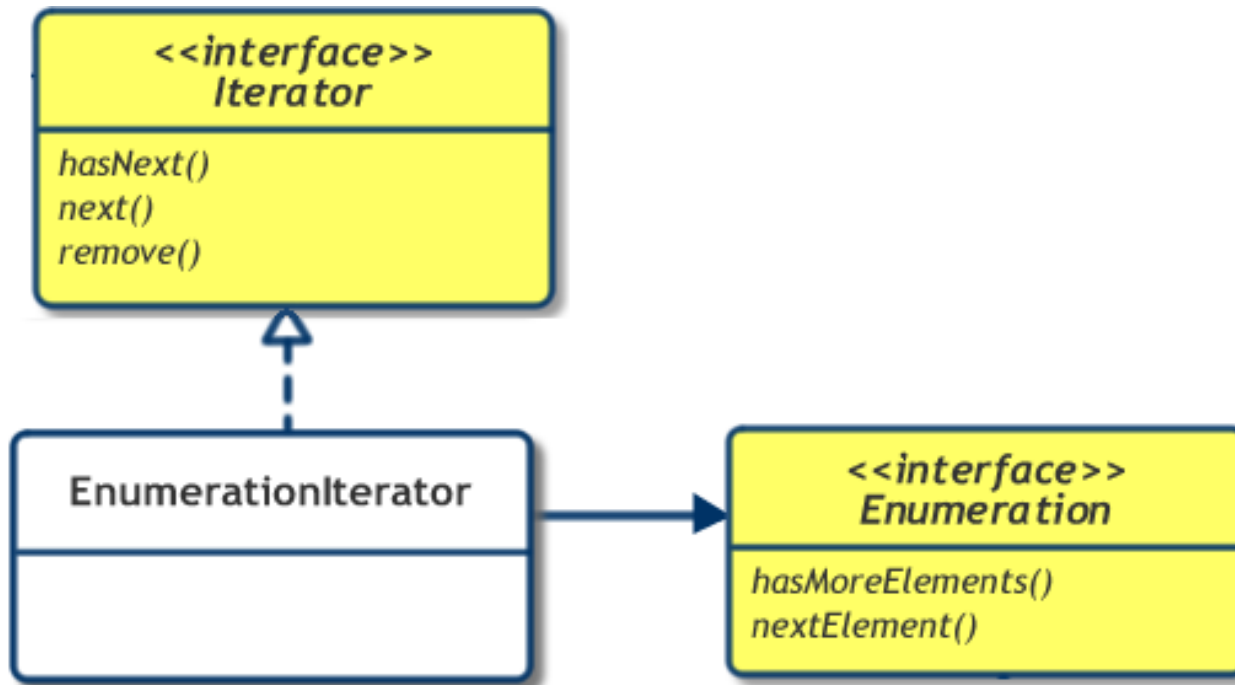
- Adaptador de classe
 - Múltipla herança



Adaptando um Enumeration a um Iterator



Adaptando



Um Exemplo: EnumerationIterator.java

```
public class EnumerationIterator implements Iterator{
    Enumeration enumeration;

    public EnumerationIterator(Enumeration enumeration) {
        this.enumeration = enumeration;
    }
    public boolean hasNext() {
        return enumeration.hasMoreElements();
    }
    public Object next() {
        return enumeration.nextElement();
    }
    public void remove() {
        throw new UnsupportedOperationException();
    }
}
```

Consequências

- Adaptadores *plugáveis*.
 - A adaptação por meio de interfaces permite incorporar a nossa classe a sistemas existentes que podem estar esperando interfaces diferentes para a classe.
- Permite a um único Adapter trabalhar com muitos Adaptees (o Adaptee e suas subclasses).

Consequências

- O Adapter também pode acrescentar funcionalidade a todos os Adaptees de uma só vez.
- Torna mais difícil redefinir um comportamento de Adaptee. Ele exigirá a criação de subclasses de Adaptee e fará com que Adapter referencie a subclasse ao invés do Adaptee em si.

Padrão

Template-Method



Padrão Método-Gabarito

■ Receita de Café

- Ferver um pouco de água.
- Misturar o café na água fervente.
- Servir o café na xícara.
- Acrescentar açúcar e leite.

■ Receita de Chá

- Ferver um pouco de água.
- Colocar o chá em infusão na água fervente.
- Despejar o chá na xícara.
- Acrescentar o limão.

2 receitas bem parecidas

Transformando as bebidas em código

Cafe.java

```
public class Cafe {  
    public void prepararReceita() {  
        ferverAgua();  
        misturarCafeComAgua();  
        servirNaXicara();  
        adicionarAcucarELeite();  
    }  
    public void ferverAgua() {  
        System.out.println("Agua Fervendo");  
    }  
    public void misturarCafeComAgua() {  
        System.out.println("Misturando café com água");  
    }  
    public void servirNaXicara() {  
        System.out.println("Servindo na xicara");  
    }  
    public void adicionarAcucarELeite() {  
        System.out.println("Adicionando acucar e leite");  
    }  
}
```

■ Receita de Café

- Ferver um pouco de água
- Misturar o café na água fervente
- Servir o café na xícara
- Acrescentar açúcar e leite

**Cada 1 dos passos
está implementado
como um método**

Agora o chá

Cha.java

```
public class Cha {  
    public void prepararReceita() {  
        ferverAgua();  
        misturarChaComAgua();  
        servirNaXicara();  
        adicionarLima() ;  
    }  
    public void ferverAgua() {  
        System.out.println("Agua Fervendo");  
    }  
    public void misturarChaComAgua() {  
        System.out.println("Mergulhando o cha");  
    }  
    public void servirNaXicara() {  
        System.out.println("Servindo na xicara");  
    }  
    public void adicionarLima() {  
        System.out.println("Adicionando limão");  
    }  
}
```

Muito parecido com a seqüência de código. O segundo e o quarto método são diferentes mas praticamente iguais.

Duplicação de código

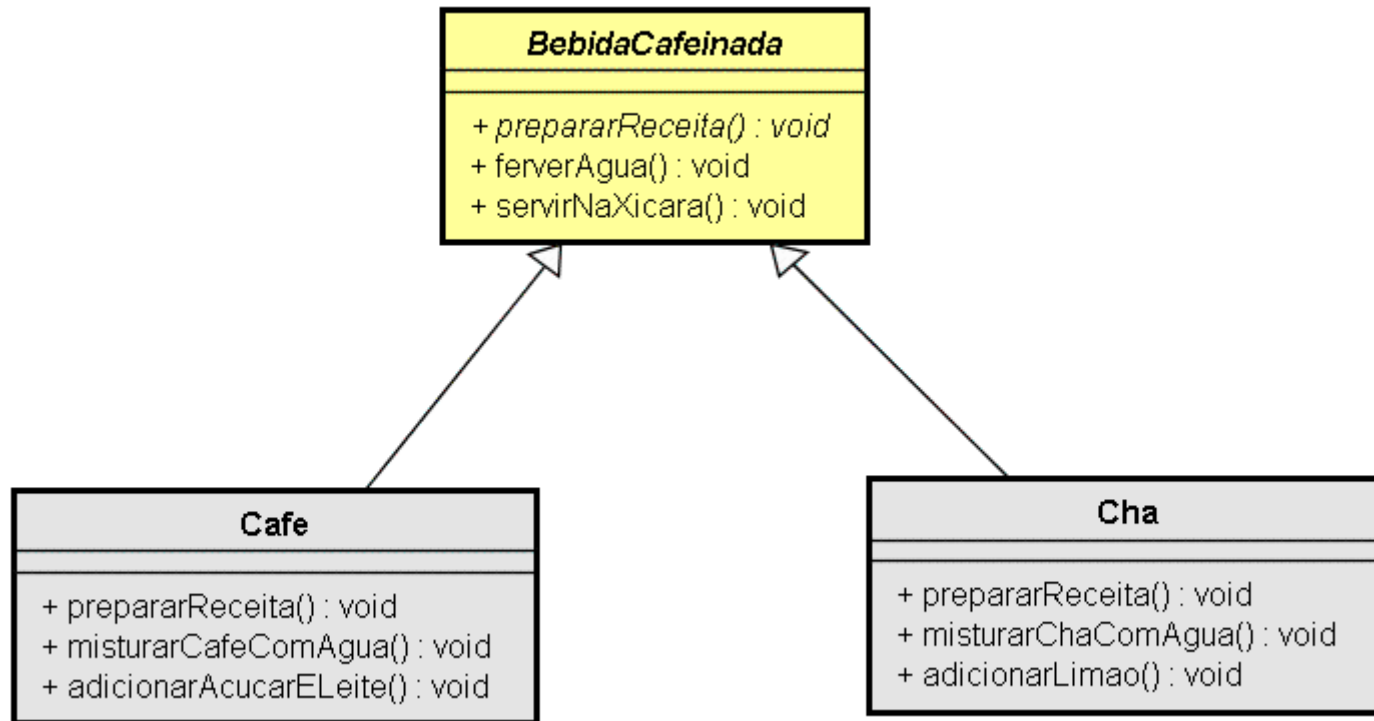
- Devemos depurar o projeto!
- Qual seria a melhor alternativa para evitar essa duplicação?

Compare os códigos

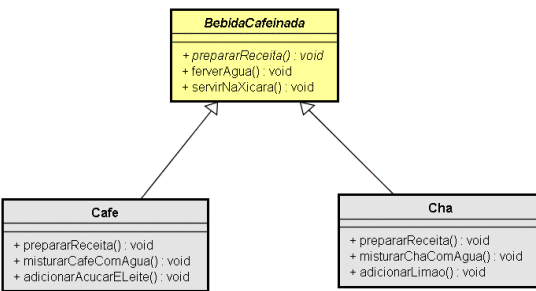
```
public class Cafe {  
  
    public void prepararReceita() {  
        ferverAgua();  
        misturarCafeComAgua();  
        servirNaXicara();  
        adicionarAcucarELeite();  
    }  
  
    public void ferverAgua() {  
        System.out.println("Agua Fervendo");  
    }  
  
    public void misturarCafeComAgua() {  
        System.out.println("Misturando café com  
        água");  
    }  
  
    public void servirNaXicara() {  
        System.out.println("Servindo na xicara");  
    }  
  
    public void adicionarAcucarELeite() {  
        System.out.println("Adicionando acucar e  
        leite");  
    }  
}
```

```
public class Cha {  
  
    public void prepararReceita() {  
        ferverAgua();  
        misturarChaComAgua();  
        servirNaXicara();  
        adicionarLimao();  
    }  
  
    public void ferverAgua() {  
        System.out.println("Agua Fervendo");  
    }  
  
    public void misturarChaComAgua() {  
        System.out.println("Mergulhando o cha");  
    }  
  
    public void servirNaXicara() {  
        System.out.println("Servindo na xicara");  
    }  
  
    public void adicionarLimao() {  
        System.out.println("Adicionando limão");  
    }  
}
```

Sugestão para uma nova estrutura



Características da nova estrutura



- O método **prepararReceita()** vai para a **superclasse** e torna-se **abstrato**, pois ele é diferente nas duas classes.
- Cada **subclasse sobrescreve** o método **prepararReceita()**.
- Os **métodos** que são **iguais** – `ferverAgua` e `servirNaXicara` – são **extraídos** para a **superclasse**.
- Os **métodos específicos** de cada classes são **implementados** na própria **subclasse**.

Indo além

- O que mais há em comum?
- Receita de Café
 - Ferver um pouco de água
 - Misturar o café na água fervente
 - Servir o café na xícara
 - Acrescentar açúcar e leite
- Receita de Chá
 - Ferver um pouco de água
 - Colocar o chá em infusão na água fervente
 - Despejar o chá na xícara
 - Acrescentar o limão

no final das contas é o mesmo algoritmo!!!

■ O mesmo algoritmo?

- 1 Ferver a água
- 2 Misturar com água quente o café ou o chá
- 3 Servir em uma xícara
- 4 Adicionar os condimentos da bebida

Existe uma maneira de extrair para a superclasse o próprio prepararReceita()?

Abstraindo o método prepararReceita()

<pre>public class Cafe { public void prepararReceita() { ferverAgua(); misturarCafeComAgua(); servirNaXicara(); adicionarAcucarELeite(); } }</pre>	<pre>public class Cha { public void prepararReceita() { ferverAgua(); misturarChaComAgua(); servirNaXicara(); adicionarLimao(); } }</pre>
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

misturar o café não é muito diferente de misturar o chá

adicionar açúcar e leite não é diferente de adicionar limão

O novo prepararReceita()

```
public final void prepararReceita() {  
    ferverAgua();  
    misturar();  
    servirNaXicara();  
    adicionarCondimentos();  
}
```

Com o método final, uma **subclasse não pode sobrescrever** o método e **alterar** a receita

A nova BebidaCafeinada.java

```
public abstract class BebidaCafeinada {  
  
    public final void prepararReceita() {  
        ferverAgua();  
        misturar();  
        servirNaXicara();  
        adicionarCondimentos();  
    }  
  
    abstract void misturar();  
    abstract void adicionarCondimentos();  
  
    public void ferverAgua() {  
        System.out.println("Agua Fervendo");  
    }  
  
    public void servirNaXicara() {  
        System.out.println("Servindo na xicara");  
    }  
}
```

Os métodos que são
específicos serão
implementados nas
subclasses

As novas bebidas

```
public class Café
    extends BebidaCafeinada {

    public void misturar() {
        System.out.println(
            "Misturando café");
    }

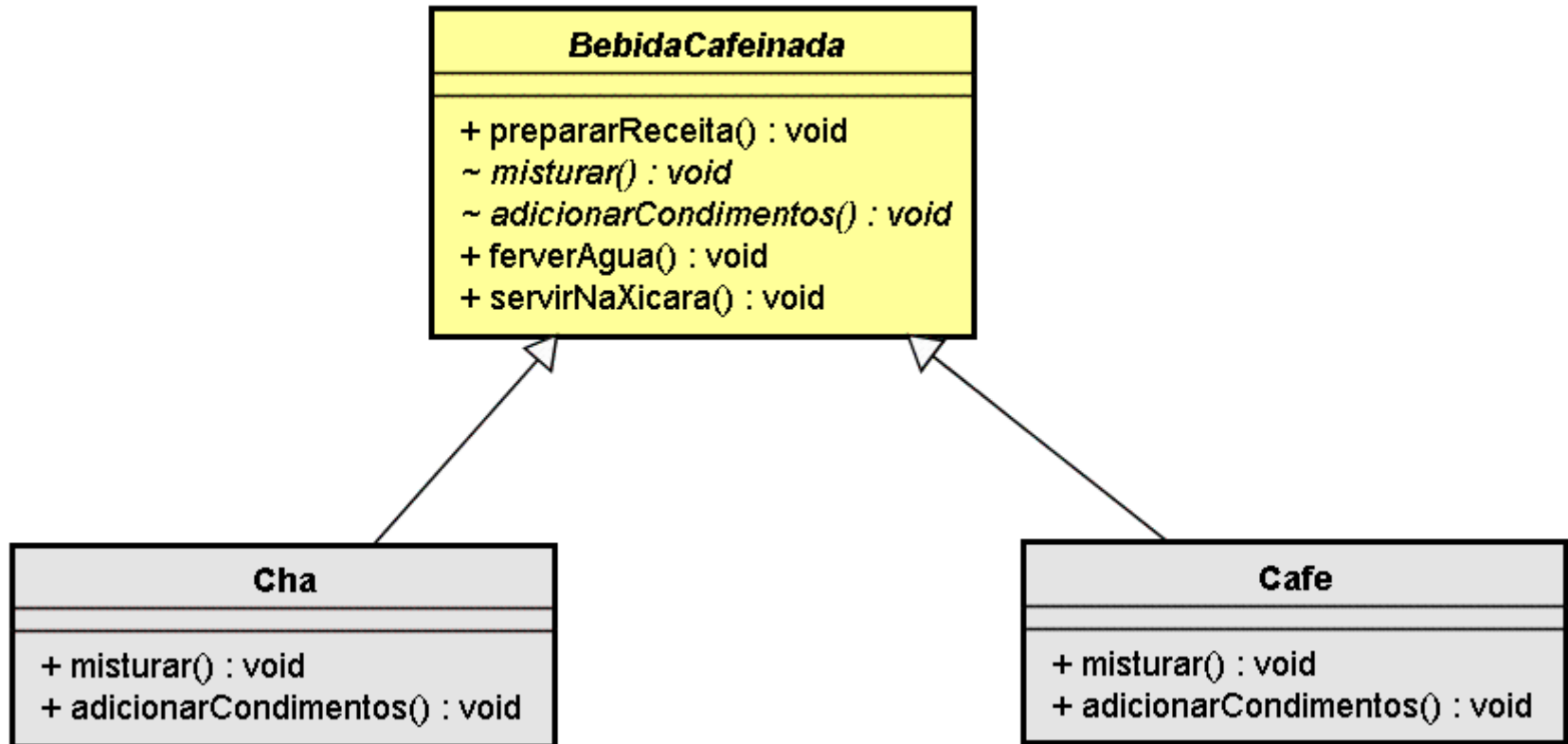
    public void
    adicionarCondimentos() {
        System.out.println(
            "Adicionando acucar");
    }
}
```

```
public class Cha
    extends BebidaCafeinada {

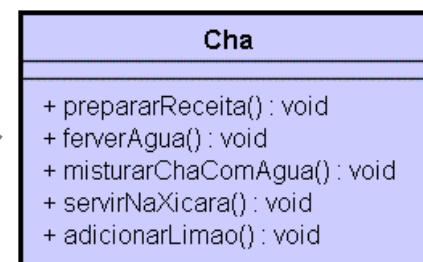
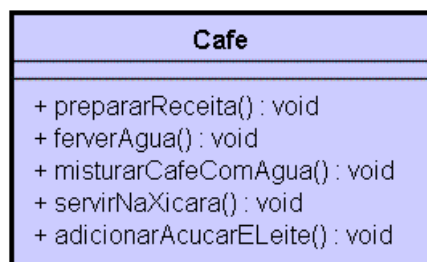
    public void misturar() {
        System.out.println(
            "Mergulhando o cha");
    }

    public void
    adicionarCondimentos() {
        System.out.println(
            "Adicionando limão");
    }
}
```

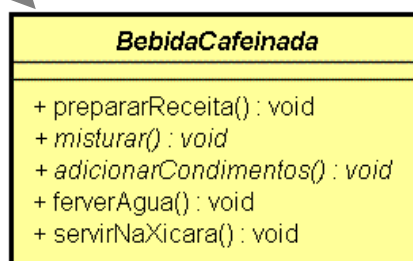
A estrutura final



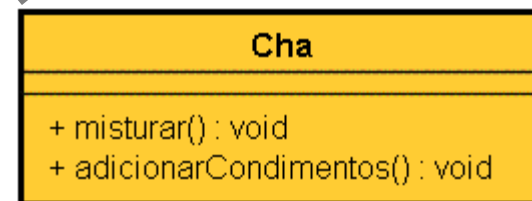
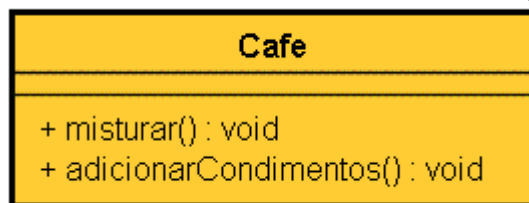
O que fizemos



generalizamos



Conhece e controla
o processo



O Método-Gabarito (Template-Method)

- O **Método-Gabarito** define os passos de um algoritmo e permite que a implementação de um ou mais desses passos seja fornecida pela subclasse.

O antes e depois

■ Antes

- As classes dispersas controlavam o algoritmo.
- Havia duplicação de código.
- Toda alteração exigiria mudanças nas subclasses gerando múltiplas alterações.
- A organização das classes exige muito trabalho para adicionar uma nova bebida.
- As informações sobre o algoritmo e a sua forma de implementação está dispersa em muitas classes.

■ Depois

- A superclasse controla o algoritmo.
- A superclasse maximiza o reuso através das subclasses.
- As mudanças no algoritmo ficam concentradas na superclasse.
- A superclasse fornece uma estrutura geral para criação de novas bebidas. Novas bebidas precisam implementar poucos métodos.
- A superclasse concentra o conhecimento sobre o algoritmo e confia às subclasses o fornecimento completo das implementações.

TEMPLATE-METHOD(Método-Gabarito)

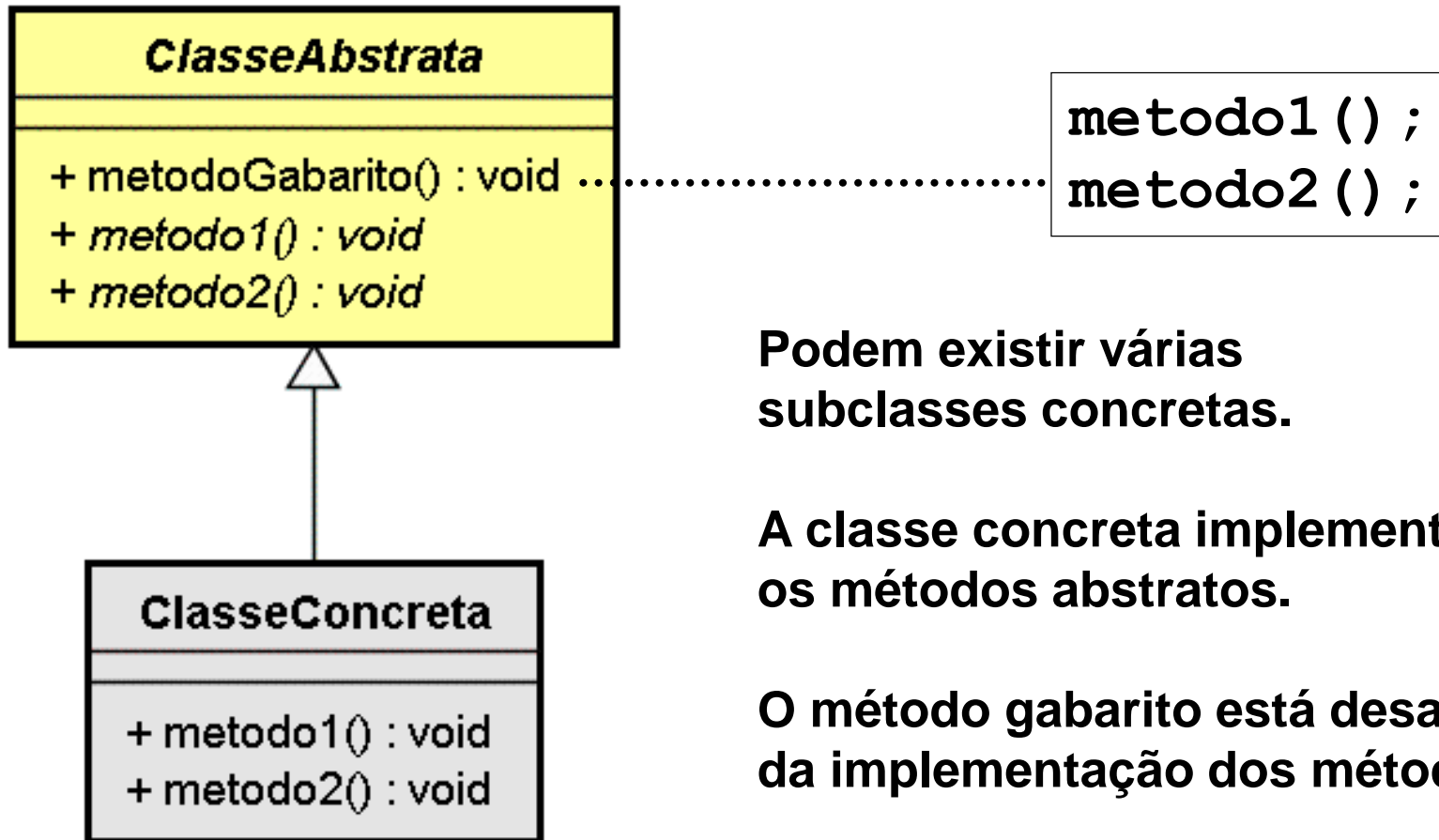
O **Padrão Template-Method** define o **esqueleto** de um algoritmo dentro de um método, **transferindo** alguns de seus **passos** para as **subclasses**. O Método-Gabarito permite que as subclasses **redefinam** certos passos de um algoritmo **sem alterar** a estrutura do mesmo.

- O padrão consiste na criação de um gabarito.
- O que é um gabarito?
 - Um método;
 - O método define um algoritmo como uma seqüência de passos;
 - Um ou mais passos podem ser redefinidos;
 - A estrutura permanece a mesma.

Aplicabilidade

- Para implementar as partes variantes de um algoritmo uma só vez e deixar para as subclasses a implementação do comportamento que pode variar.
- Quando o comportamento comum entre subclasses deve ser fatorado e concentrado numa classe comum para evitar duplicação de código.
- Para controlar extensões de subclasses.

Diagrama de classes



Podem existir várias subclasses concretas.

A classe concreta implementa os métodos abstratos.

O método gabarito está desacoplado da implementação dos métodos.

■ Classe Abstrata

- Define as operações abstratas fundamentais que as subclasses concretas definem para implementar passos de um algoritmo.
- Implementa um método gabarito que define o esqueleto de um algoritmo.

■ Classe Concreta


- Implementa as operações fundamentais para executarem os passos específicos do algoritmo da subclasse.

- A Classe Concreta depende da Classe Abstrata para implementar os passos invariantes do algoritmo.

O “gancho”

- Um método vazio:
 - `public void gancho() {}`

```
public abstract class ClasseAbstrata {  
  
    public void metodoGabarito() {  
        metodo1();  
        metodo2();  
        metodoConcreto();  
        gancho();  
    }  
  
    abstract void metodo1();  
    abstract void metodo2();  
  
    final void metodoConcreto() {  
        //implementacao...  
    }  
  
    void gancho() {}  
}
```



Um gancho (hook) é um método declarado na classe abstrata mas só recebe como implementação um corpo vazio ou quase vazio.

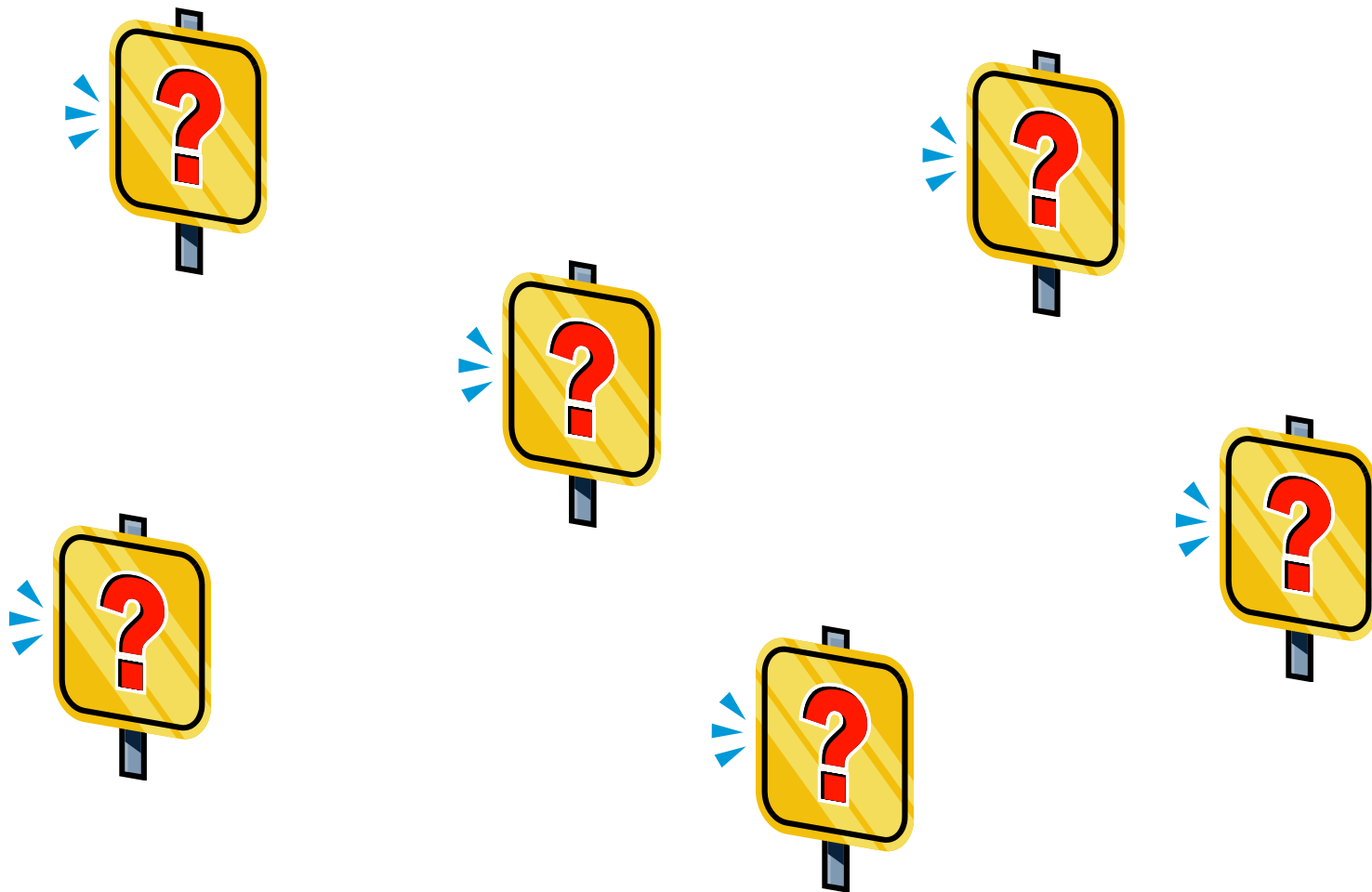
O “gancho”

```
public abstract class BebidaCafeinada {
    public final void prepararReceita() {
        ferverAgua();
        misturar();
        servirNaXicara();
        if ( clienteQuerCondimentos() ) {
            adicionarCondimentos();
        }
    }
    public boolean clienteQuerCondimentos() { return true; }
    // outros métodos ...
}

public class Cafe extends BebidaCafeinada {
    public void misturar() {...}
    public void adicionarCondimentos() { ... }
    public boolean clienteQuerCondimentos() {
        // pergunta ao cliente se ele quer condimentos...
    }
}
```

Consequências

- Servem como meios para fatoração dos comportamentos comuns nas bibliotecas de classes.
- Conduzem a uma estrutura de inversão de controle, ou seja, “não nos chame, nós chamaremos você” (também conhecido como **Princípio de Hollywood**).
 - Isto se refere a como uma classe-mãe chama as operações de uma subclasse, e não o contrário.





Obrigado!!!

Agradecimentos:

Prof. Eduardo Mendes

Prof. Régis Simão

Faculdade 7 de Setembro