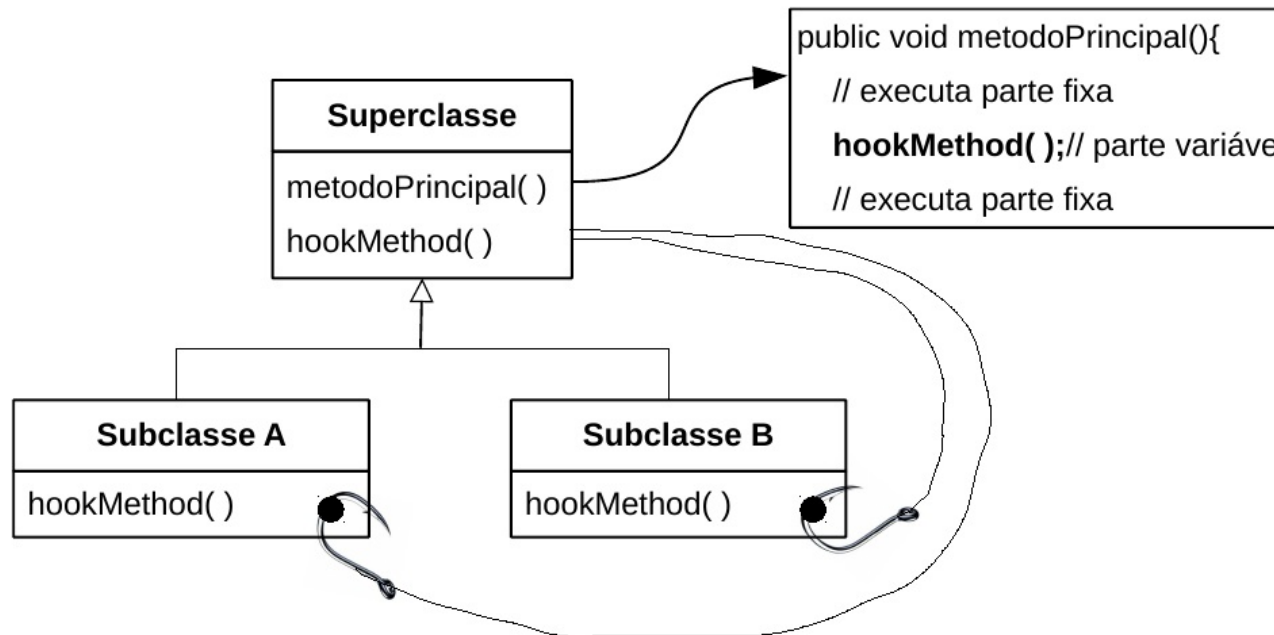


Template Method

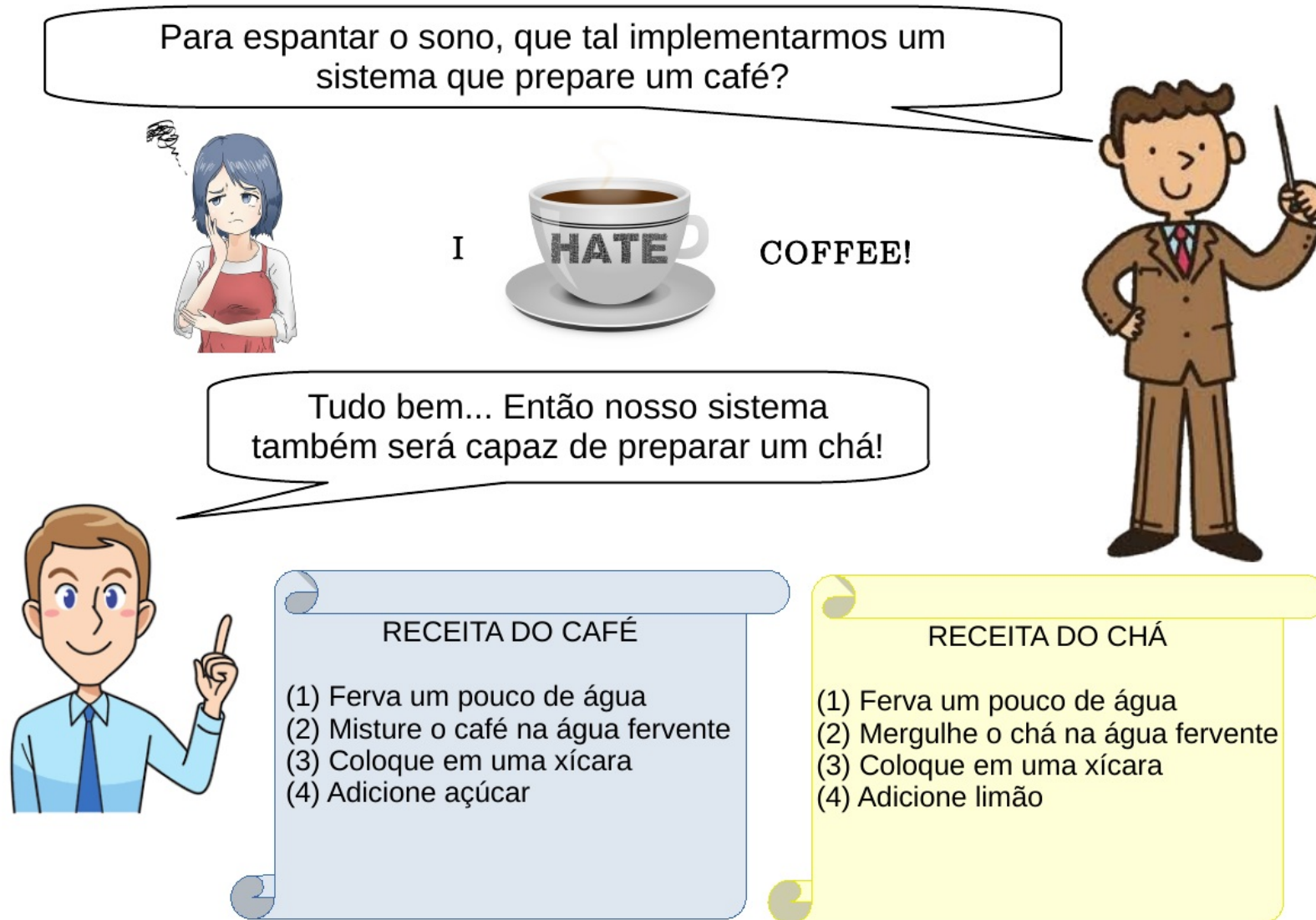
Padrão de Projeto *Template Method*

Descrição do problema

- ▶ O padrão *Template Method* define em um método a estrutura (“esqueleto”) de um algoritmo que possui algumas partes fixas e outras variáveis. As partes variáveis são representadas como *hook methods* que podem ser implementados nas subclasses.



Padrão de Projeto *Template Method*



Padrão de Projeto *Template Method*

Aqui estão as implementações de nossas bebidas.
Alguma coisa chama atenção nesses dois códigos?

Códigos muito semelhantes.
Apenas o 2º e 4º passos são diferentes!



```
public class Cafe {
```

```
    public void prepararReceita(){  
        ferverAgua();  
        misturarCafe();  
        colocarNaXicara();  
        adicionarAcucar();  
    }
```

```
    public void ferverAgua() {  
        System.out.println("Fervendo a água...");  
    }
```

```
    public void misturarCafe() {  
        System.out.println("Coando o café...");  
    }
```

```
    public void colocarNaXicara() {  
        System.out.println("Colocando na xícara...");  
    }
```

```
    public void adicionarAcucar() {  
        System.out.println("Adicionando açúcar...");  
    }
```

```
}
```

```
public class Cha {
```

```
    public void prepararReceita(){  
        ferverAgua();  
        mergulharSache();  
        colocarNaXicara();  
        adicionarLimao();  
    }
```

```
    public void ferverAgua() {  
        System.out.println("Fervendo a água...");  
    }
```

```
    private void mergulharSache() {  
        System.out.println("Mergulhando o sachê...");  
    }
```

```
    public void colocarNaXicara() {  
        System.out.println("Colocando na xícara...");  
    }
```

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

```
}
```

Iguais

Diferentes

Iguais

Diferentes

Padrão de Projeto *Template Method*

Aqui estão as implementações de nossas bebidas.
Alguma coisa chama atenção nesses dois códigos?



Códigos muito semelhantes.
Apenas o 2º e 4º passos são diferentes!

```
public class Cafe {
```

```
    public void prepararReceita(){  
        ferverAgua();  
        misturarCafe();  
        colocarNaXicara();  
        adicionarAcucar();  
    }
```

```
    public void ferverAgua() {  
        System.out.println("Fervendo a água...");  
    }
```

```
    public void misturarCafe() {  
        System.out.println("Coando o café...");  
    }
```

```
    public void colocarNaXicara() {  
        System.out.println("Colocando na xícara...");  
    }
```

```
    public void adicionarAcucar() {  
        System.out.println("Adicionando açúcar...");  
    }
```

```
}
```

```
public class Cha {
```

```
    public void prepararReceita(){  
        ferverAgua();  
        mergulharSache();  
        colocarNaXicara();  
        adicionarLimao();  
    }
```

```
    public void ferverAgua() {  
        System.out.println("Fervendo a água...");  
    }
```

```
    public void mergulharSache() {  
        System.out.println("Mergulhando o sachê...");  
    }
```

```
    public void colocarNaXicara() {  
        System.out.println("Colocando na xícara...");  
    }
```

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

```
}
```

DUPLICAÇÃO DE CÓDIGO!!!

Diferentes

Padrão de Projeto *Template Method*

Como vocês remodelariam as classes para eliminar essa redundância?

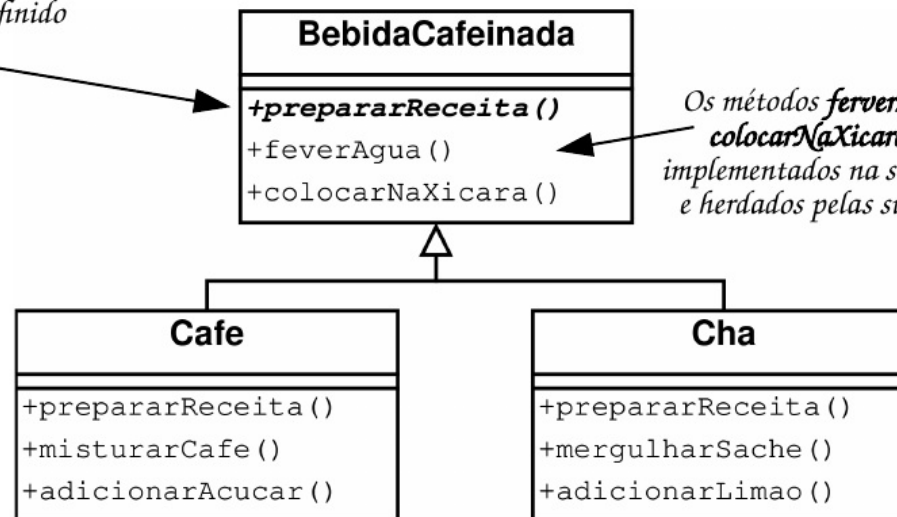
Provavelmente alguns de vocês pensariam na solução a seguir.

O método ***prepararReceita()*** é diferente para cada subclasse e, desse modo, ele é definido como abstrato.

Os métodos ***ferverAgua()*** e ***colocarNaXicara()*** são implementados na superclasse e herdados pelas subclasses.

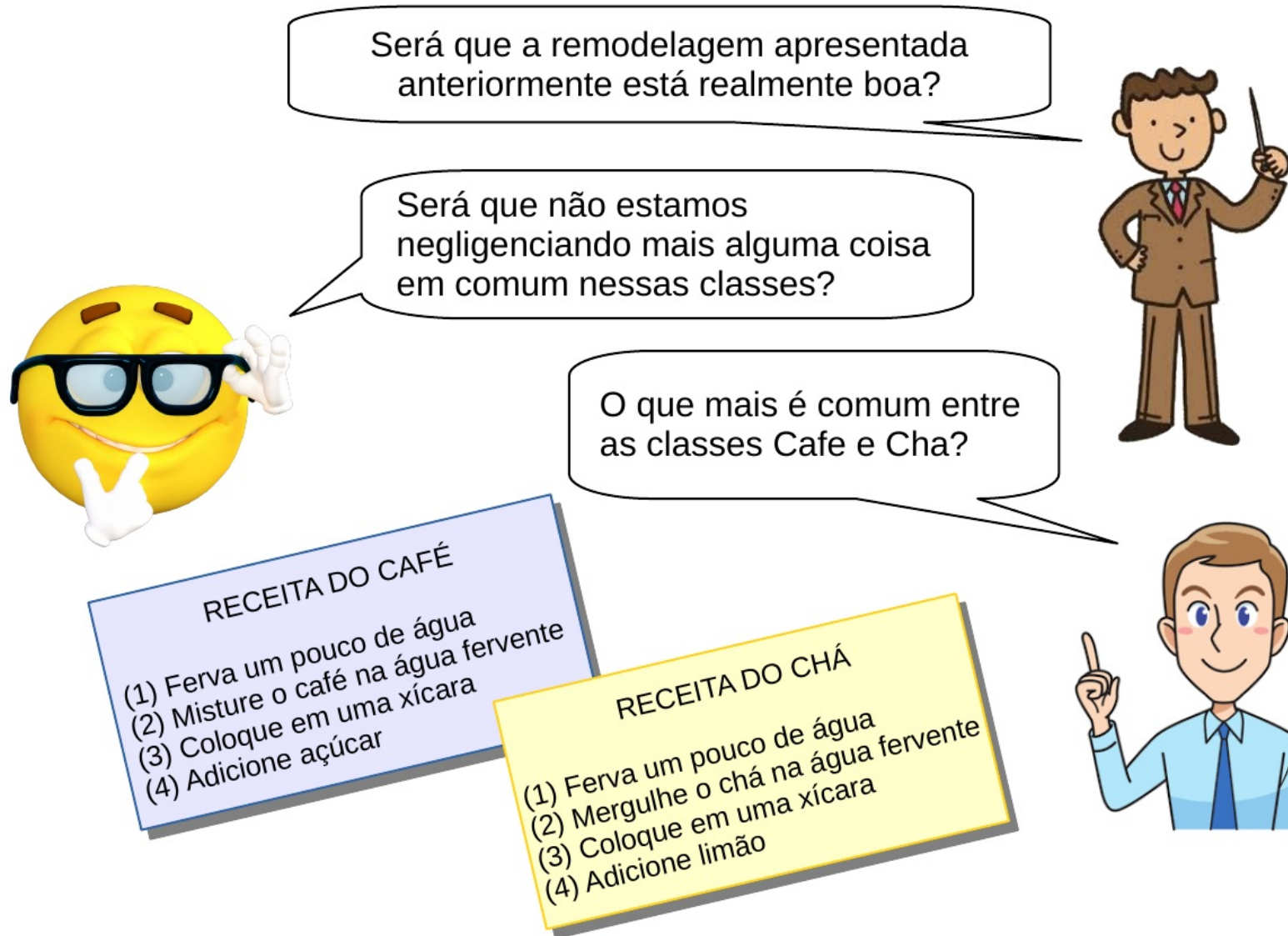
Cada subclasse implementa sua própria receita.

Cada subclasse sobrescreve o método ***prepararReceita()*** para implementar sua própria receita.



Os métodos específicos de ***Cafe*** e ***Cha*** ficam nas subclasses.

Padrão de Projeto *Template Method*



Padrão de Projeto *Template Method*

Note que as duas receitas seguem o mesmo algoritmo!

Esses métodos eram iguais e já estavam implementados na superclasse.

- (1) *Ferva um pouco de água*
- (2) *Use água fervente para extrair a bebida*
- (3) *Coloque a bebida em uma xícara*
- (4) *Adicione os condimentos*

Portanto, acabamos de encontrar uma maneira de abstrair o método `prepararReceita()`.

Qual a importância disso para nossa implementação?

Já esses outros dois métodos não estavam implementados na superclasse. No entanto, eles são iguais, sendo apenas aplicados a diferentes bebidas.



Padrão de Projeto *Template Method*

Agora podemos incluir a implementação do método prepararReceita() na superclasse BebidaCafeinada.

```
public abstract class BebidaCafeinada {
```

← Essa classe é abstrata.

```
    public final void prepararReceita(){
```

Agora o mesmo método **PrepararReceita()** será usado para chá e café. Ele é declarado como **final** para evitar que as subclasses possam alterar a receita.

```
        ferverAgua();
        misturarComAguaFervente();
        colocarNaXicara();
        adicionarCondimentos();
    }
```

```
    public abstract void misturarComAguaFervente();
```

```
    public abstract void adicionarCondimentos();
```

```
    public void ferverAgua() {
        System.out.println("Fervendo a água...");
    }
```

```
    public void colocarNaXicara() {
        System.out.println("Colocando na xícara...");
    }
```

```
}
```

Como a implementação desses métodos é específica para café e chá, eles são declarados como abstratos e, portanto, implementados nas subclasses.

Assim como havíamos feito na remodelagem anterior, esses dois métodos continuam sendo implementados nesta classe.



Padrão de Projeto *Template Method*

Por fim, veja como ficam as classes Cafe e Cha...

```
public class Cafe extends BebidaCafeinada{  
    @Override  
    public void misturarComAguaFervente() {  
        System.out.println("Coando o café...");  
    }  
    @Override  
    public void adicionarCondimentos() {  
        System.out.println("Adicionando açúcar...");  
    }  
}
```

Assim como havíamos feito na remodelagem anterior, Cafe e Cha *herdam* de BebidaCafeinada.

Agora *Cafe* precisa implementar os 2 métodos abstratos da classe *BebidaCafeinada*.

```
public class Cha extends BebidaCafeinada{  
    @Override  
    public void misturarComAguaFervente() {  
        System.out.println("Mergulhando o sachê...");  
    }  
    @Override  
    public void adicionarCondimentos() {  
        System.out.println("Adicionando limão...");  
    }  
}
```

Do mesmo modo, *Cha* precisa implementar os 2 métodos abstratos da classe *BebidaCafeinada*.



Padrão de Projeto *Template Method*

Analisando a estrutura da classe BebidaCafeinada ...

```
public abstract class BebidaCafeinada {  
    public final void prepararReceita(){  
        ferverAgua();  
        misturarComAguaFervente();  
        colocarNaXicara();  
        adicionarCondimentos();  
    }  
  
    public abstract void misturarComAguaFervente();  
    public abstract void adicionarCondimentos();  
  
    public void ferverAgua() {  
        System.out.println("Fervendo a água...");  
    }  
  
    public void colocarNaXicara() {  
        System.out.println("Colocando na xícara...");  
    }  
}
```

Temos aqui nosso **template method**. Ele serve como um template para um algoritmo.

Nesse template, cada etapa do algoritmo é representada por um método.

Alguns métodos são implementados nesta classe.

E outros métodos são implementados nas subclasses.

Os métodos que precisam ser implementados nas subclasses são declarados como **abstratos**.



Padrão de Projeto *Template Method*

Analisando a estrutura da classe BebidaCafeinada ...

```
public abstract class BebidaCafeinada {  
    public final void prepararReceita()  
    {  
        ferverAgua();  
        misturarComAçúcar();  
        colocarNaXícara();  
        adicionarCondimento();  
    }  
    public abstract void ferverAgua();  
    public abstract void misturarComAçúcar();  
    public void colocarNaXícara() {  
        System.out.println("Colocando na xícara...");  
    }  
}
```

O *Template Method* define as etapas de um algoritmo e permite que as subclasses forneçam a implementação para uma ou mais dessas etapas.

Temos aqui nosso **template method**. Ele serve como um template para um algoritmo.



O algoritmo é definido por um método.

Todos os métodos são declarados nesta classe.

E outros métodos são implementados nas subclasses.

Os métodos que precisam ser implementados nas subclasses são declarados como **abstratos**.

Padrão de Projeto *Template Method*

Afinal, como iremos preparar o café e o chá?

```
public class TemplateMetodo1 {  
    public static void main(String[] args) {  
        BebidaCafeinada cafe = new Cafe();  
        BebidaCafeinada cha = new Cha();  
  
        System.out.println("### Preparando o Café ###");  
        cafe.prepararReceita();  
  
        System.out.println("### Preparando o Chá ###");  
        cha.prepararReceita();  
    }  
}
```



```
### Preparando o Café ###  
Fervendo a água...  
Coando o café...  
Colocando na xícara...  
Adicionando açúcar...  
### Preparando o Chá ###  
Fervendo a água...  
Mergulhando o sachê...  
Colocando na xícara...  
Adicionando limão...
```

Padrão de Projeto *Template Method*

Quais as consequências do uso desse padrão de projeto?

SEM Uso do Padrão de Projeto



As classes Cafe e Cha controlam o algoritmo.

Código duplicado nas classes Cafe e Cha.

Alterações no código do algoritmo de preparação das bebidas requerem mudanças em múltiplas classes.

As classes estão organizadas de modo que muito trabalho é necessário para inclusão de novas bebidas cafeinadas.

O conhecimento do algoritmo e sua implementação é distribuída entre múltiplas classes.

COM Uso do Padrão de Projeto



A classe BebidaCafeinada controla o algoritmo e o protege.

A classe BebidaCafeinada possibilita a reutilização de código.

Como o algoritmo de preparação das bebidas está implementado em uma única classe, as mudanças ocorrerão apenas nela.

A utilização do padrão de projeto fornece um framework que facilita a inclusão de novas bebidas cafeinadas.

O conhecimento do algoritmo e sua implementação é concentrada em uma única classe, a qual faz uso de subclasses para completar a implementação.

Estrutura do Padrão de Projeto *Template Method*

