Padrões de Projeto de Software Orientados a Objetos

Tecnologia em Análise e Desenvolvimento de Sistemas

Paulo Mauricio Gonçalves Júnior

Instituto Federal de Educação, Ciência e Tecnologia de Pernambuco

16 de março de 2018

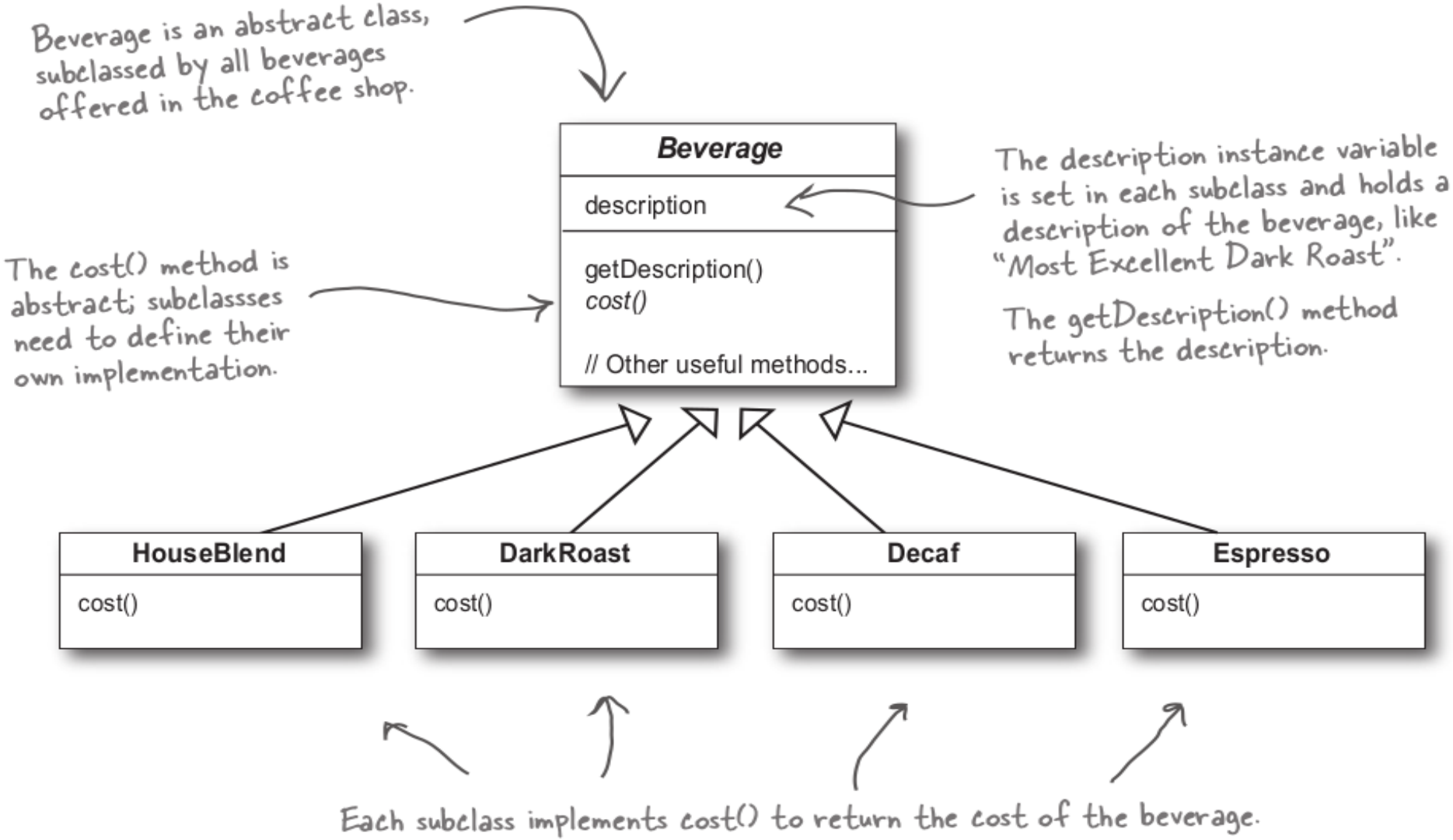
Parte III

Decorator

Introdução I

- Vamos aprender a decorar classes em tempo de execução usando uma forma de composição de objetos.
- Poderemos assim dar novas responsabilidades sem modificações de código às classes.
- Vamos implementar um sistema para uma cafeteria. Inicialmente, eles projetaram as classes desse jeito:

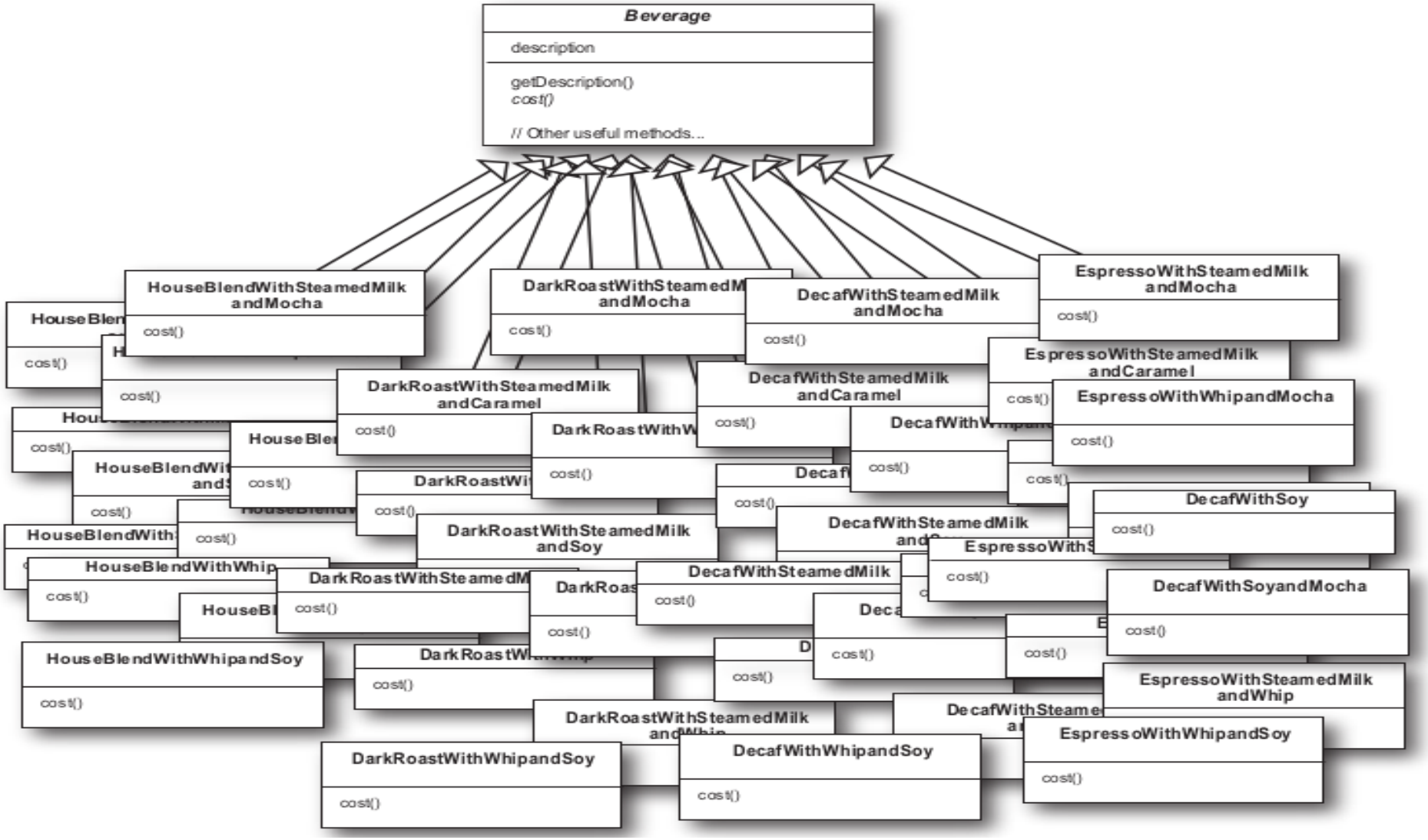
Introdução II



Introdução III

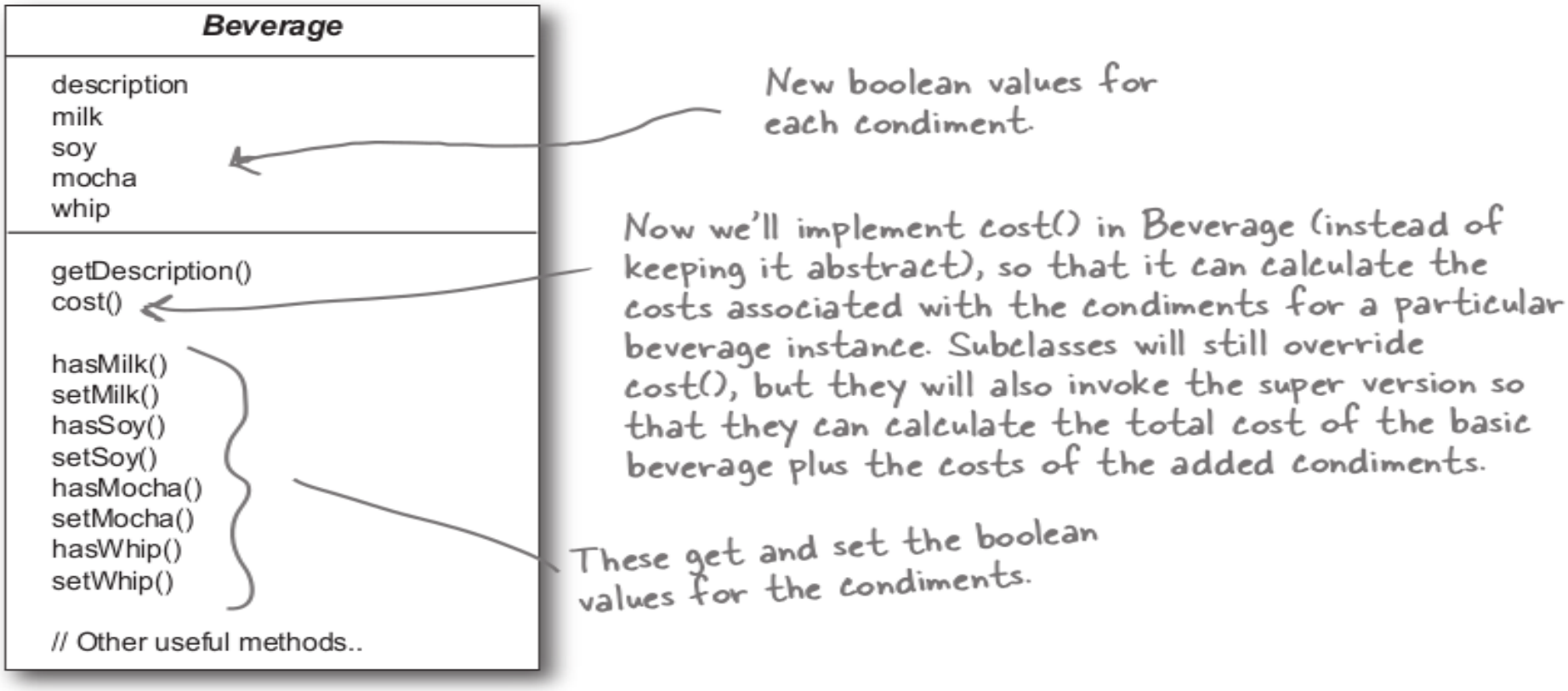
- Podemos também incrementar o café com leite fervido, leite de soja, chocolate, etc., sendo cobrado um valor por cada um deles.

Introdução IV



Introdução V

- Isso gerou uma explosão de classes! Imagine se o preço do leite aumenta, ou se oferecermos caramelo?
- Podemos usar atributos e herança para gerenciar as coberturas.

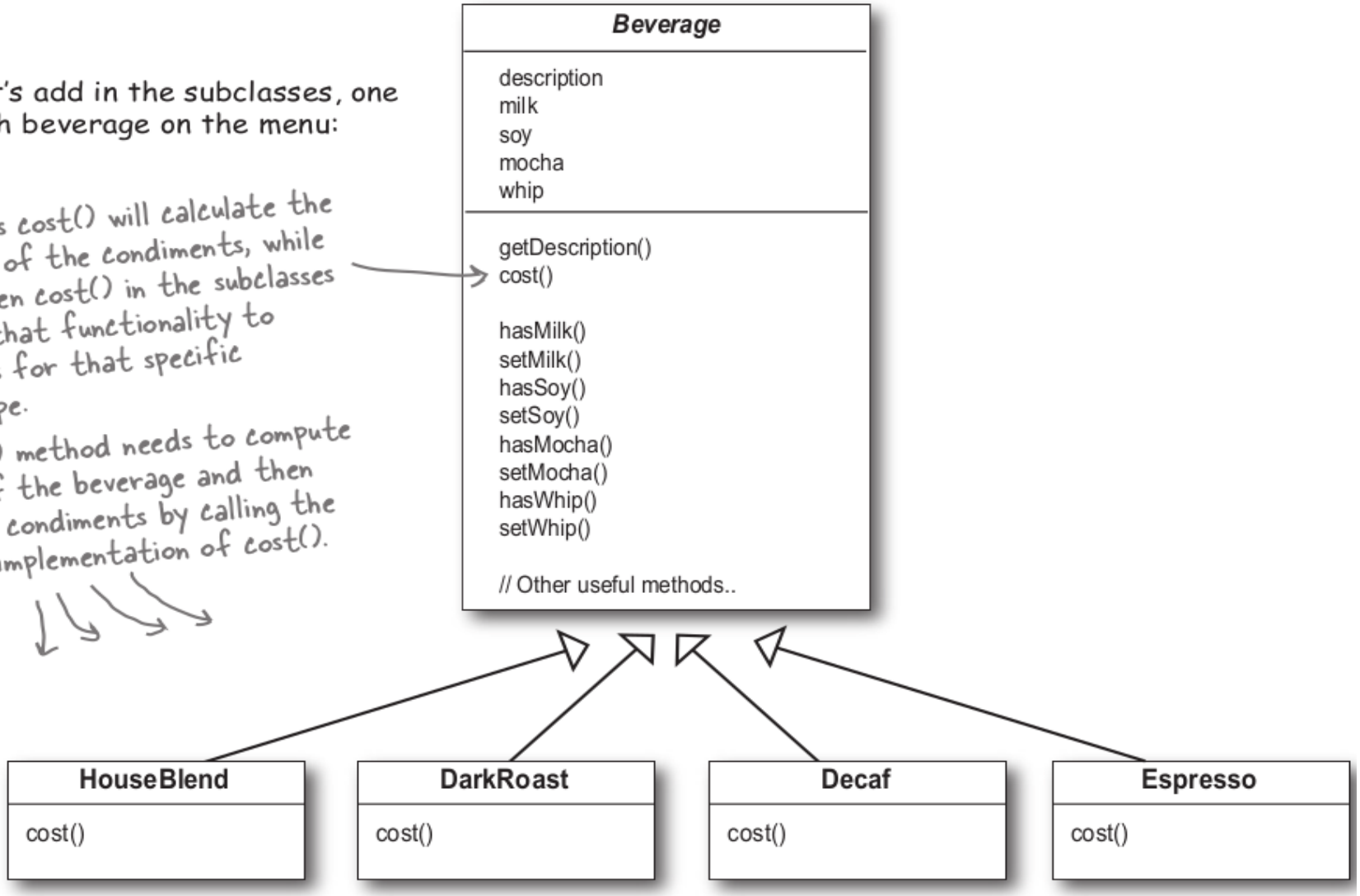


Introdução VI

Now let's add in the subclasses, one for each beverage on the menu:

The superclass cost() will calculate the costs for all of the condiments, while the overridden cost() in the subclasses will extend that functionality to include costs for that specific beverage type.

Each cost() method needs to compute the cost of the beverage and then add in the condiments by calling the superclass implementation of cost().



Introdução VII

- Modificações de preços de coberturas forçam mudanças em código existente.
- Novas coberturas forçam adicionar novos métodos e alterar método `cost`.
- Para algumas novas bebidas algumas coberturas não se aplicam.
- E se o cliente quiser duplo chocolate?

Princípio de Projeto

Classes devem ser abertas para extensão, mas fechadas para modificação.

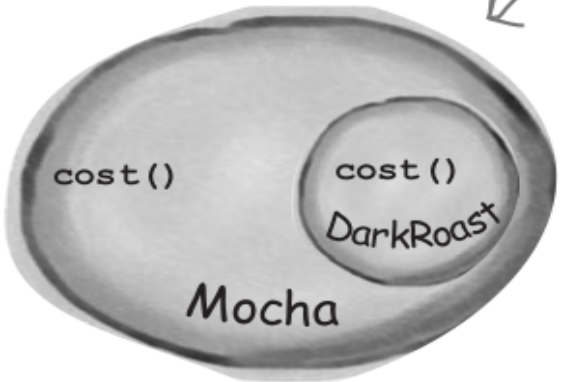
- Solução
  - Pegar um objeto DarkRoast
  - Decorar com um objeto Chocolate
  - Decorar com um objeto Leite Fervido
  - Chamar o método `cost` e delegar para adicionar coberturas.

Introdução VIII

1 **We start with our DarkRoast object.**

Remember that DarkRoast inherits from Beverage and has a cost() method that computes the cost of the drink.

2 **The customer wants Mocha, so we create a Mocha object and wrap it around the DarkRoast.**

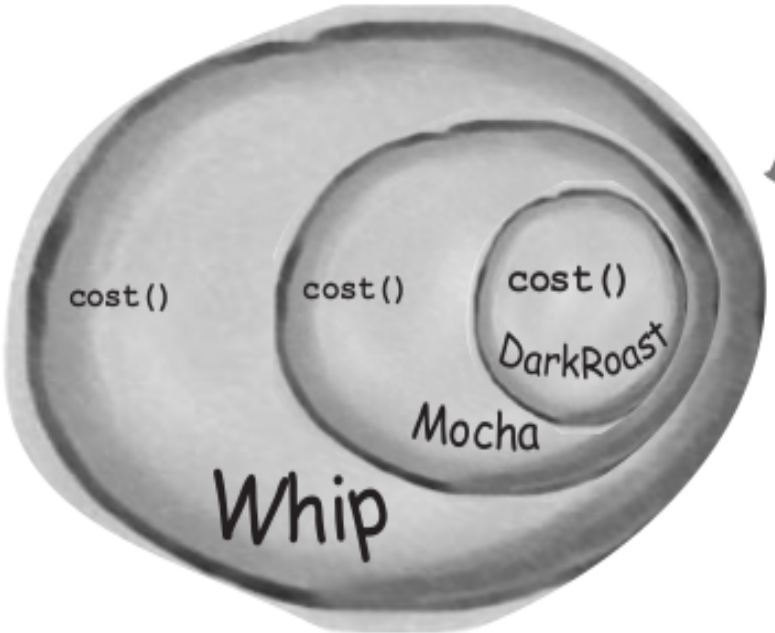


The Mocha object is a decorator. Its type mirrors the object it is decorating, in this case, a Beverage. (By "mirror", we mean it is the same type..)

So, Mocha has a cost() method too, and through polymorphism we can treat any Beverage wrapped in Mocha as a Beverage, too (because Mocha is a subtype of Beverage).

Introdução IX

3 The customer also wants Whip, so we create a Whip decorator and wrap Mocha with it.

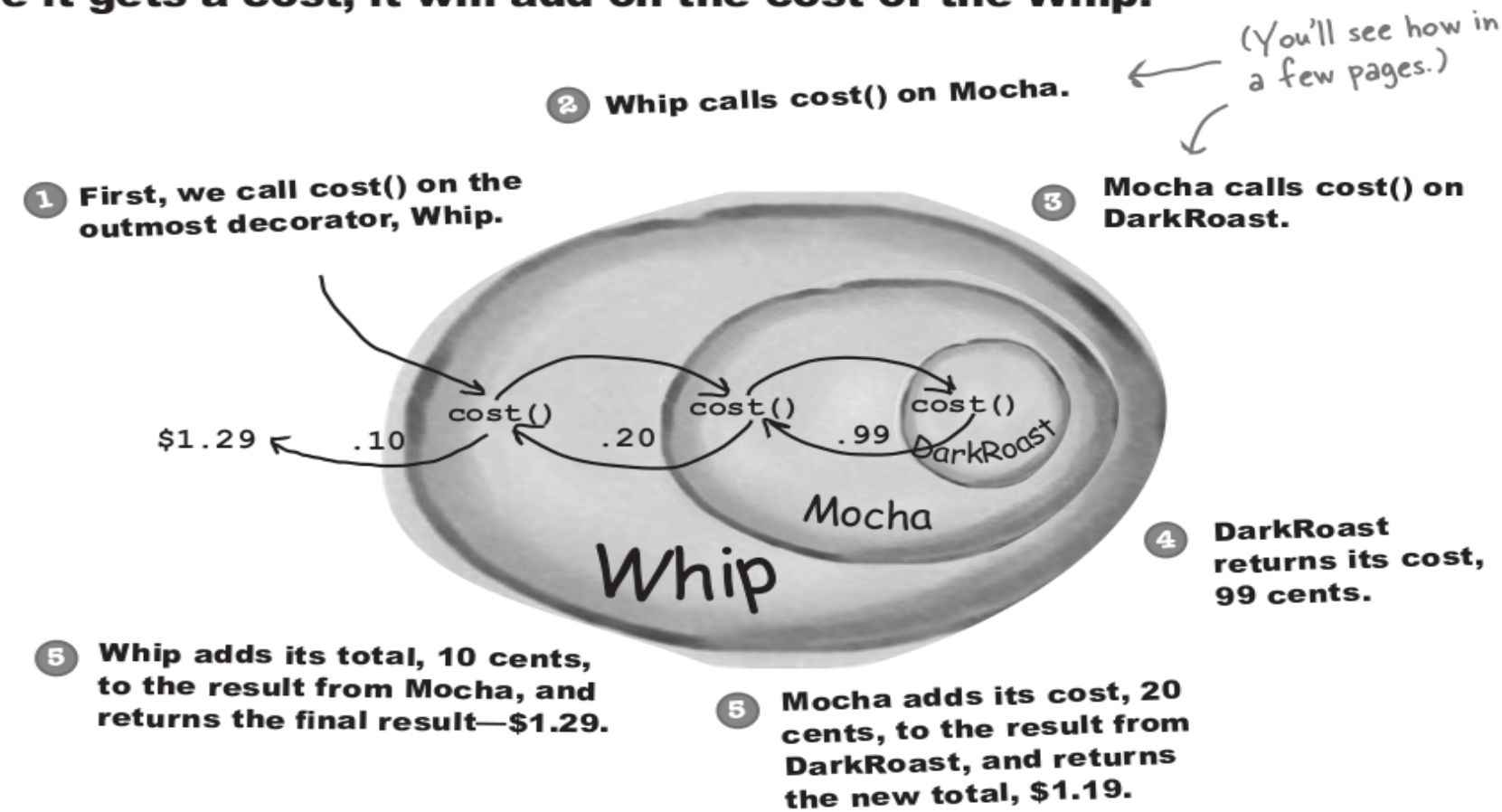


Whip is a decorator, so it also mirrors DarkRoast's type and includes a cost() method.

So, a DarkRoast wrapped in Mocha and Whip is still a Beverage and we can do anything with it we can do with a DarkRoast, including call its cost() method.

Introdução X

4 **Now it's time to compute the cost for the customer. We do this by calling cost() on the outermost decorator, Whip, and Whip is going to delegate computing the cost to the objects it decorates. Once it gets a cost, it will add on the cost of the Whip.**



Introdução XI

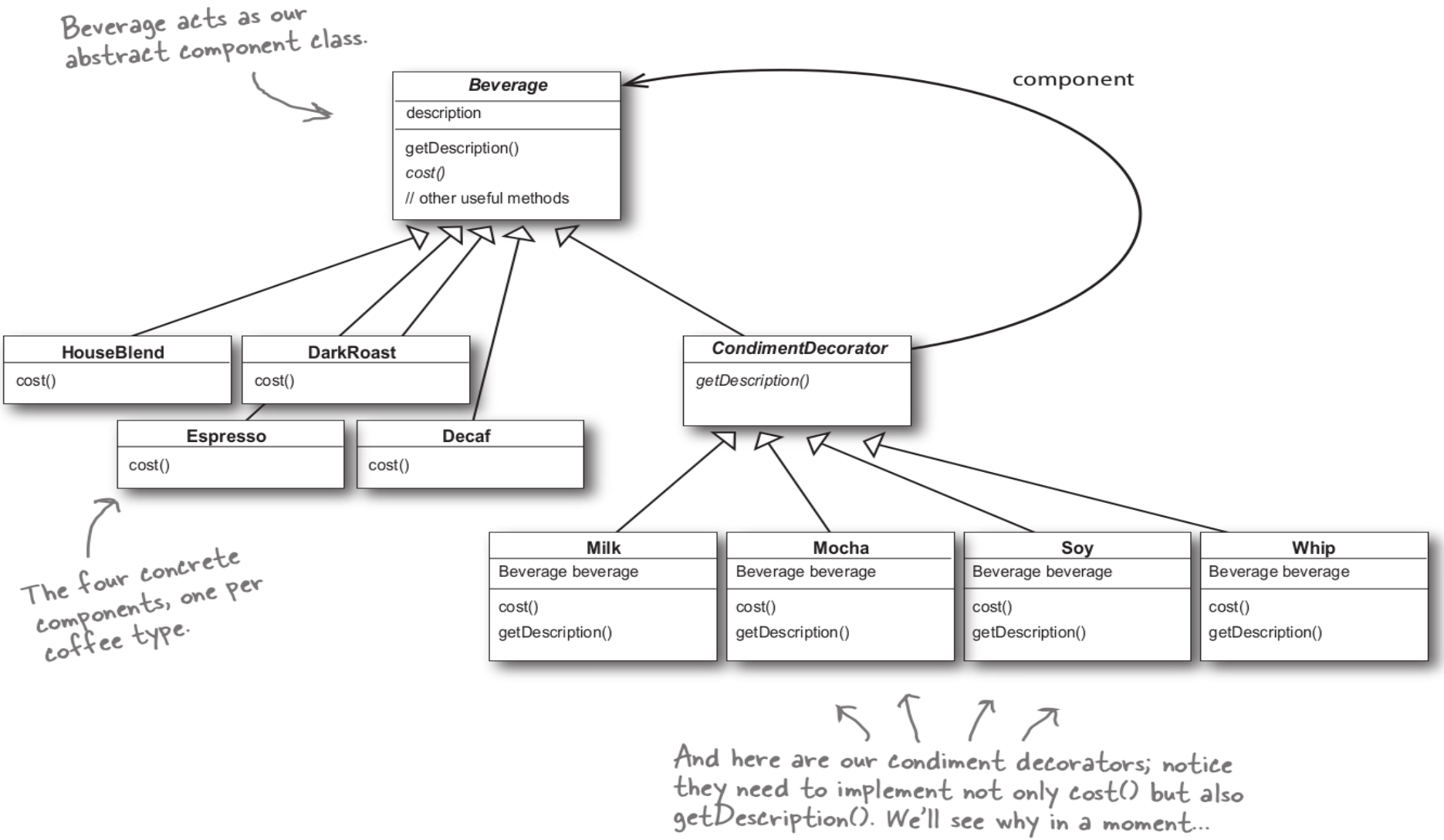
- Decoradores possuem o mesmo tipo dos objetos que você decora.
- Você pode usar um ou mais decoradores para encapsular um objeto.
- Podemos usar um objeto decorado no lugar do objeto encapsulado original.
- O decorador adiciona seu próprio comportamento tanto antes e/ou depois delegando para o objeto que ele decora fazer o resto do trabalho.
- Objetos podem ser em qualquer momento, em qualquer quantidade.

Introdução XII

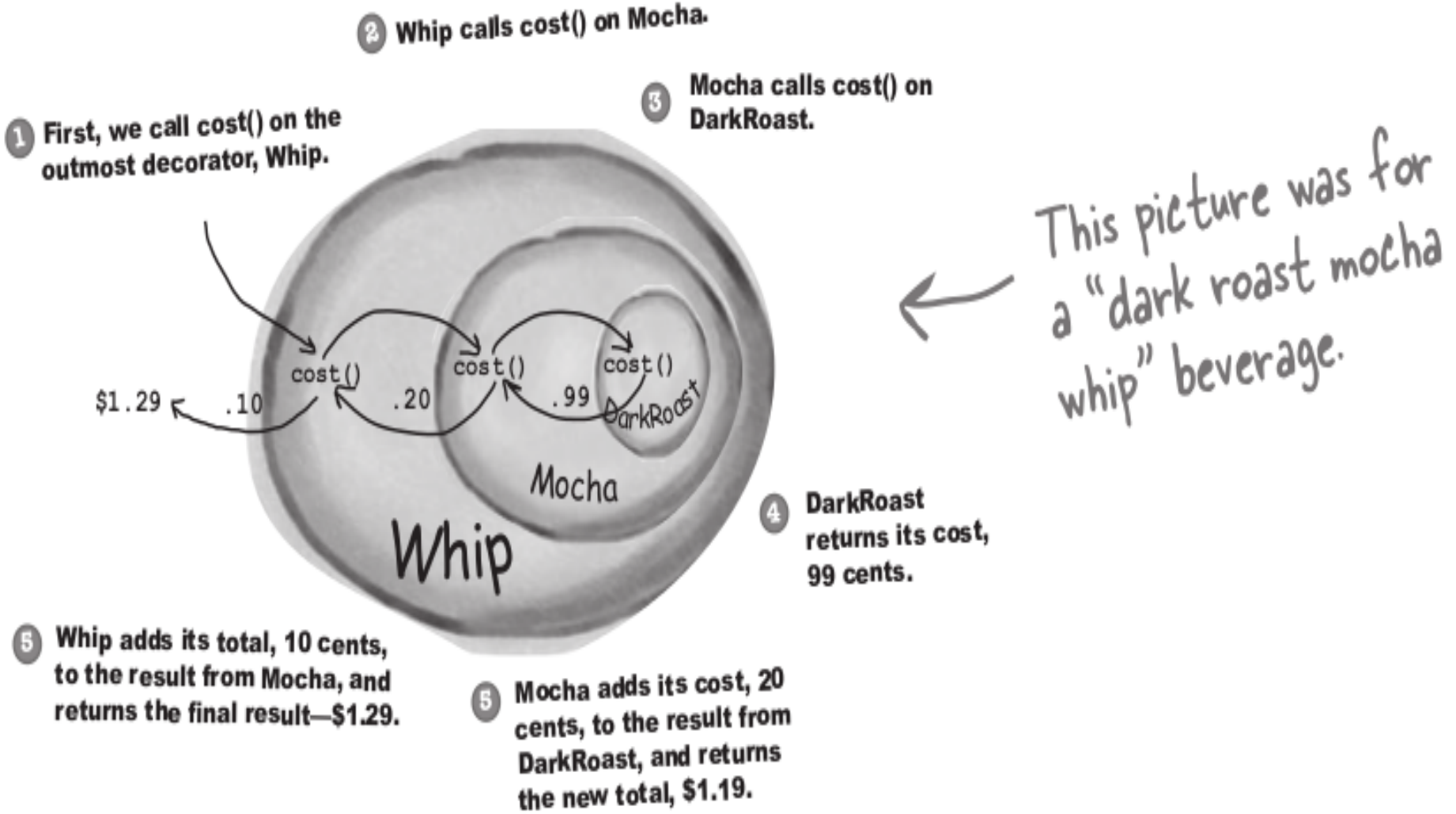
Definição

O padrão Decorator anexa responsabilidades adicionais a um objeto dinamicamente. Decoradores provém uma alternativa flexível à herança para extensão de funcionalidades.

Introdução XIII



Introdução XIV



Introdução I

```
public abstract class Beverage {
  String description = "Unknown Beverage";
  public String getDescription() {
    return description;
  }
  public abstract double cost();
}

public abstract class CondimentDecorator extends Beverage {
  public abstract String getDescription();
}

public class Espresso extends Beverage {
  public Espresso() {
    description = "Espresso";
  }
  public double cost() {
    return 1.99;
  }
}
```

Introdução II

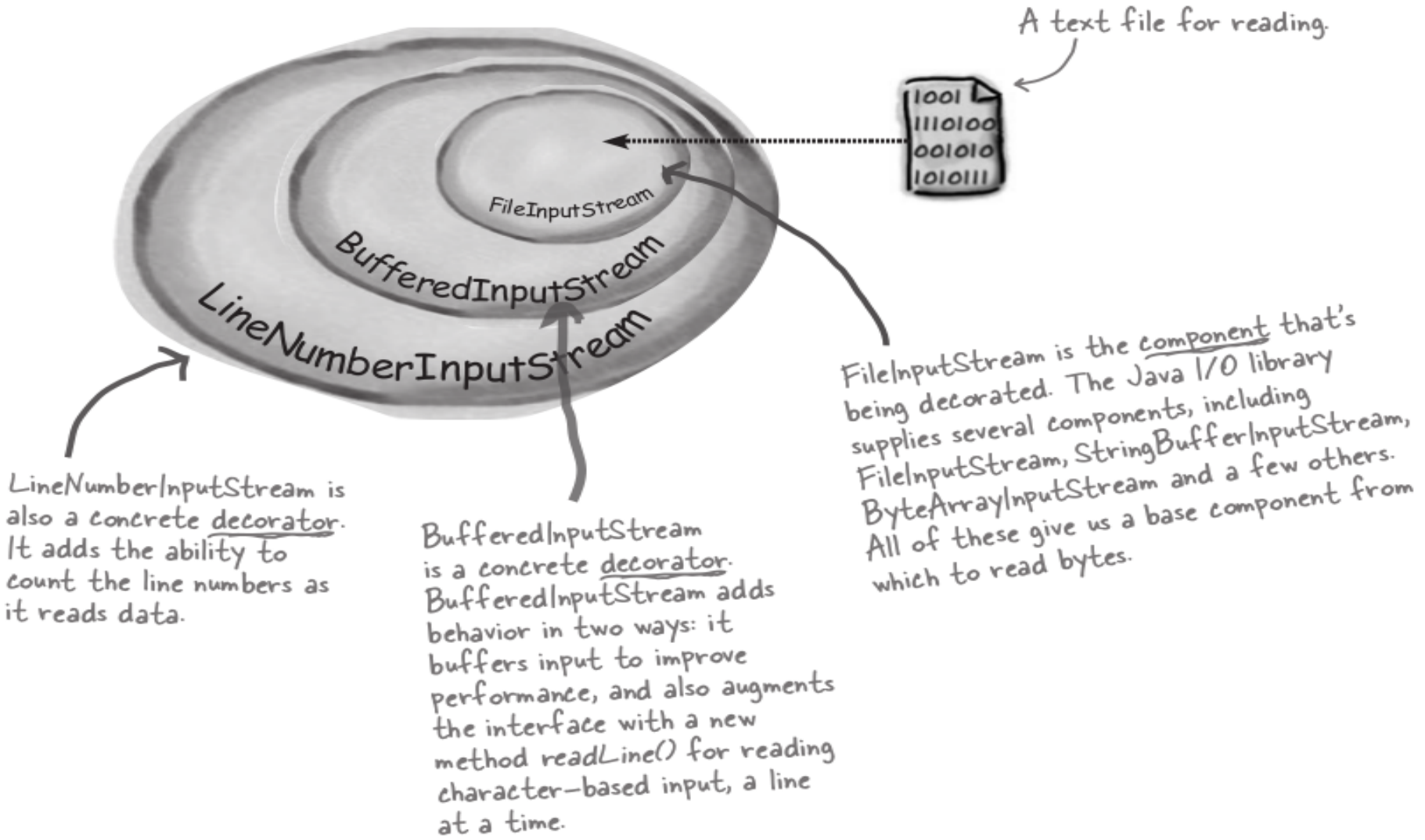
```
public class Mocha extends CondimentDecorator {
  Beverage beverage;
  public Mocha(Beverage beverage) {
    this.beverage = beverage;
  }
  public String getDescription() {
    return beverage.getDescription() + ", Mocha";
  }
  public double cost() {
    return .20 + beverage.cost();
  }
}

public class StarbuzzCoffee {
  public static void main(String args[]) {
    Beverage beverage = new Espresso();
    System.out.println(beverage.getDescription() + " $" + beverage.
        cost());
    Beverage beverage2 = new DarkRoast();
    beverage2 = new Mocha(beverage2);
    beverage2 = new Mocha(beverage2);
```

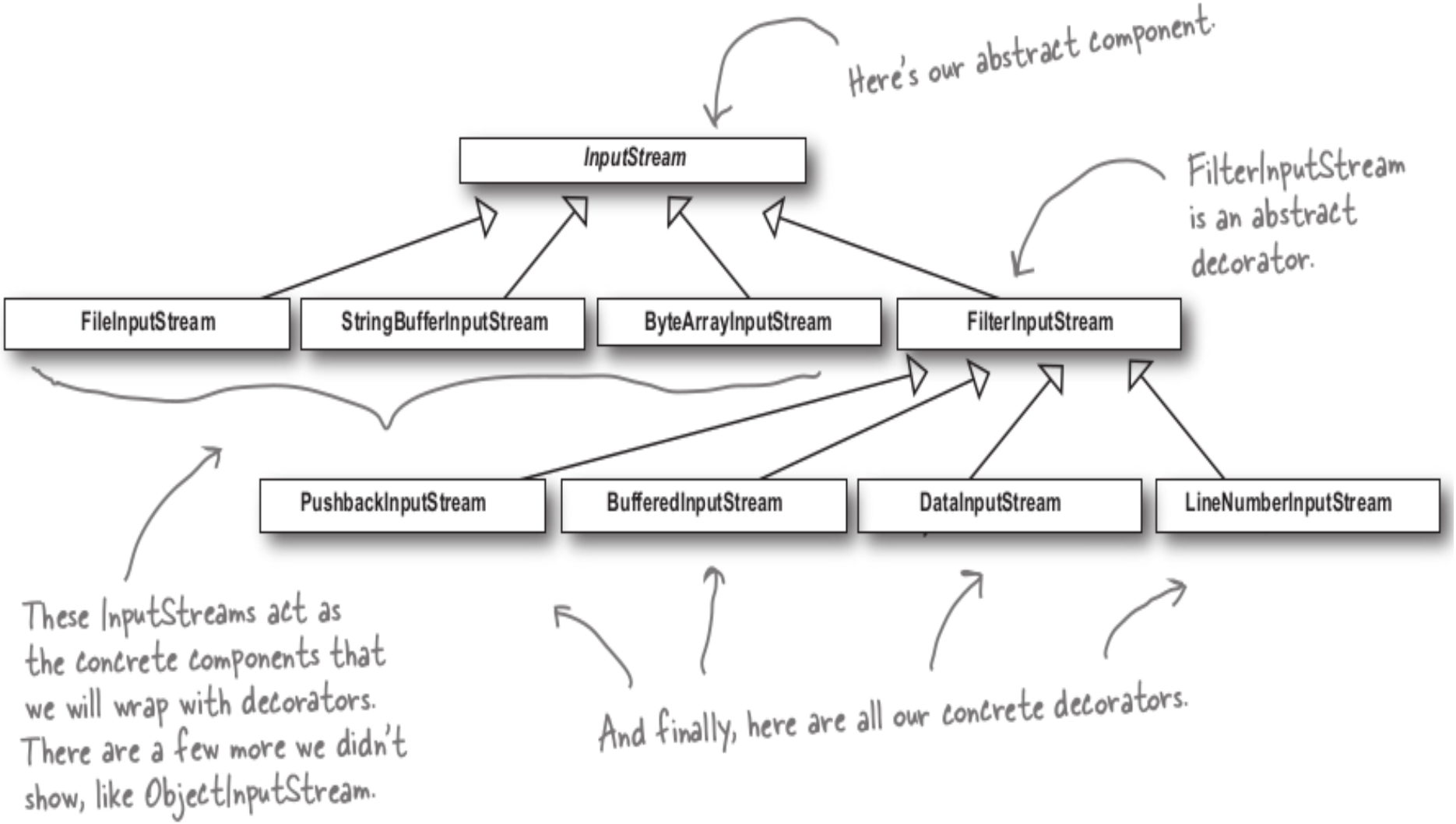
Introdução III

```
        beverage2 = new Whip(beverage2);
        System.out.println(beverage2.getDescription() + " $" +
            beverage2.cost());
        Beverage beverage3 = new HouseBlend();
        beverage3 = new Soy(beverage3);
        beverage3 = new Mocha(beverage3);
        beverage3 = new Whip(beverage3);
        System.out.println(beverage3.getDescription() + " $" +
            beverage3.cost());
    }
}
```

Exemplo: Java I/O I



Exemplo: Java I/O II



Exemplo: Java I/O III

```
public class LowerCaseInputStream extends FilterInputStream {
  public LowerCaseInputStream(InputStream in) {
    super(in);
  }

  public int read() throws IOException {
    int c = super.read();
    return (c == -1 ? c : Character.toLowerCase((char)c));
  }

  public int read(byte[] b, int offset, int len) throws IOException
      {
    int result = super.read(b, offset, len);
    for (int i = offset; i < offset+result; i++) {
      b[i] = (byte)Character.toLowerCase((char)b[i]);
    }
  }
}

public class InputTest {
  public static void main(String[] args) throws IOException {
```

Exemplo: Java I/O IV

```
    int c;
    try {
      InputStream in = new LowerCaseInputStream(new
          BufferedInputStream(new FileInputStream("test.txt")));
      while((c = in.read()) >= 0) {
        System.out.print((char)c);
      }
      in.close();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```