Computação Orientada a Objetos

Padrões de Projeto
Decorator e Adapter

Slides baseados em:
-E. Gamma and R. Helm and R. Johnson and J. Vlissides. Design Patterns - Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.
-E. Freeman and E. Freeman. Padrões de Projetos. Use a cabeça. Alta Books Editora. 2009.
- Slides Prof. Christian Danniel Paz Trillo
- Slides Profa. Patrícia R. Oliveira

Profa. Karina Valdivia Delgado
EACH-USP

PADRÃO DECORATOR

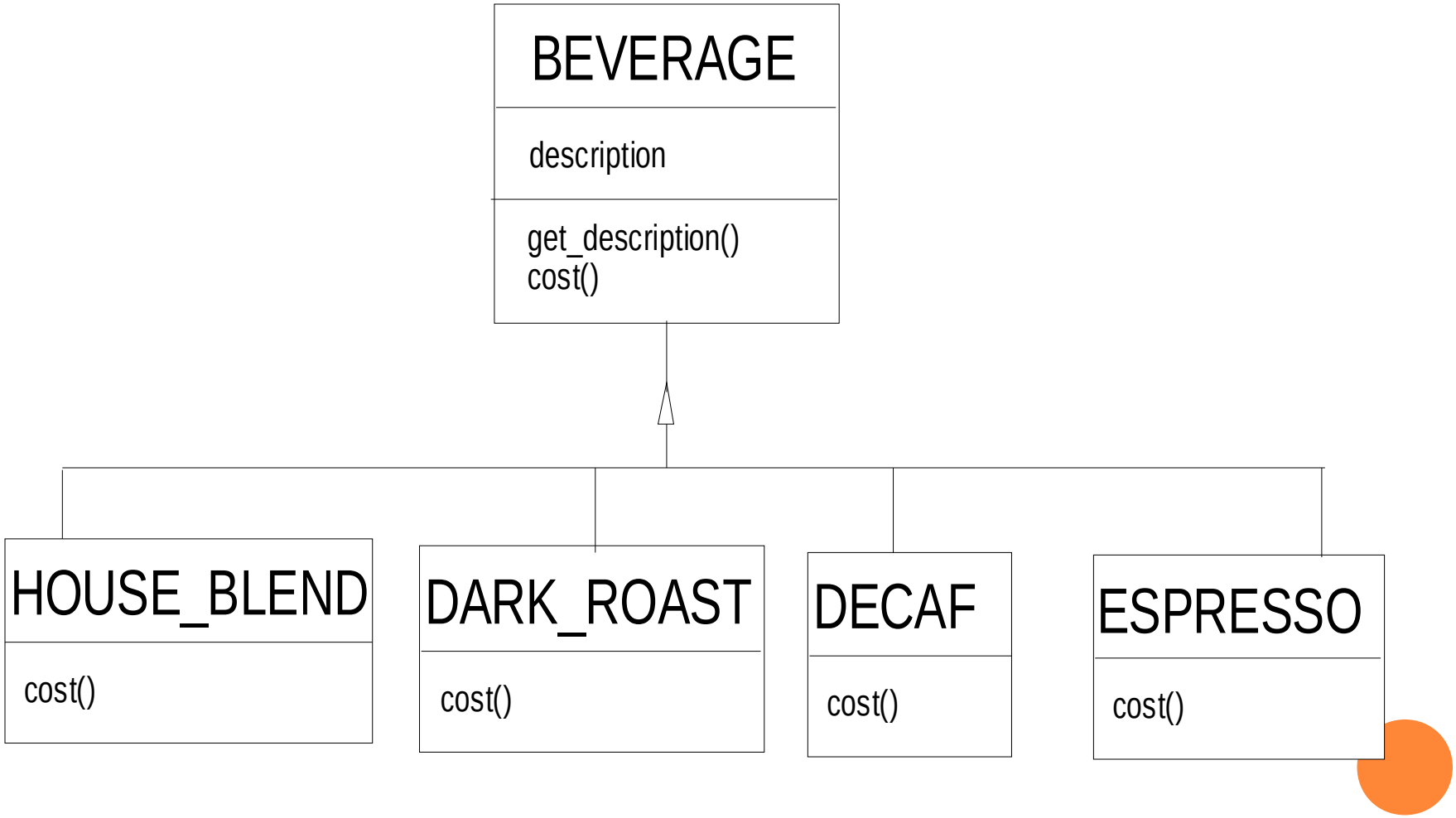
PADRÃO DECORATOR

- Padrão Estrutural.
- Objetivo:
  - Anexar responsabilidades adicionais a um objeto dinamicamente (“Enfeitar”).
  - Decorators oferecem uma alternativa flexível de subclasse para estender a funcionalidade.
- Motivação

Exemplo: Starbuzz Coffe

- Starbuzz Coffe vende diferentes tipos de bebida
- O custo de cada bebida é calculada de maneira diferente
- Starbuzz precissa um sistema para calcular o custo de cada bebida

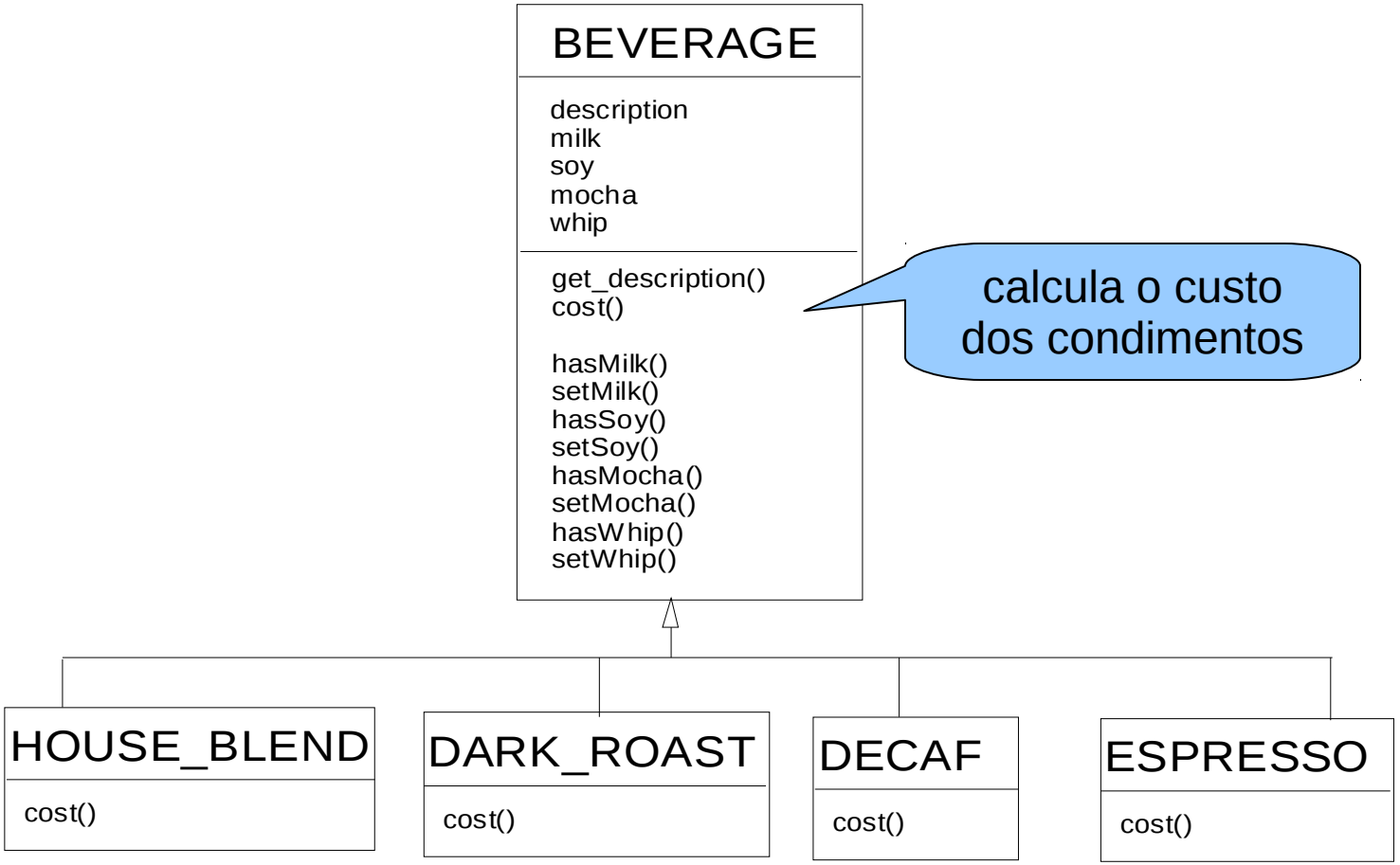
Exemplo: Starbuzz Coffe



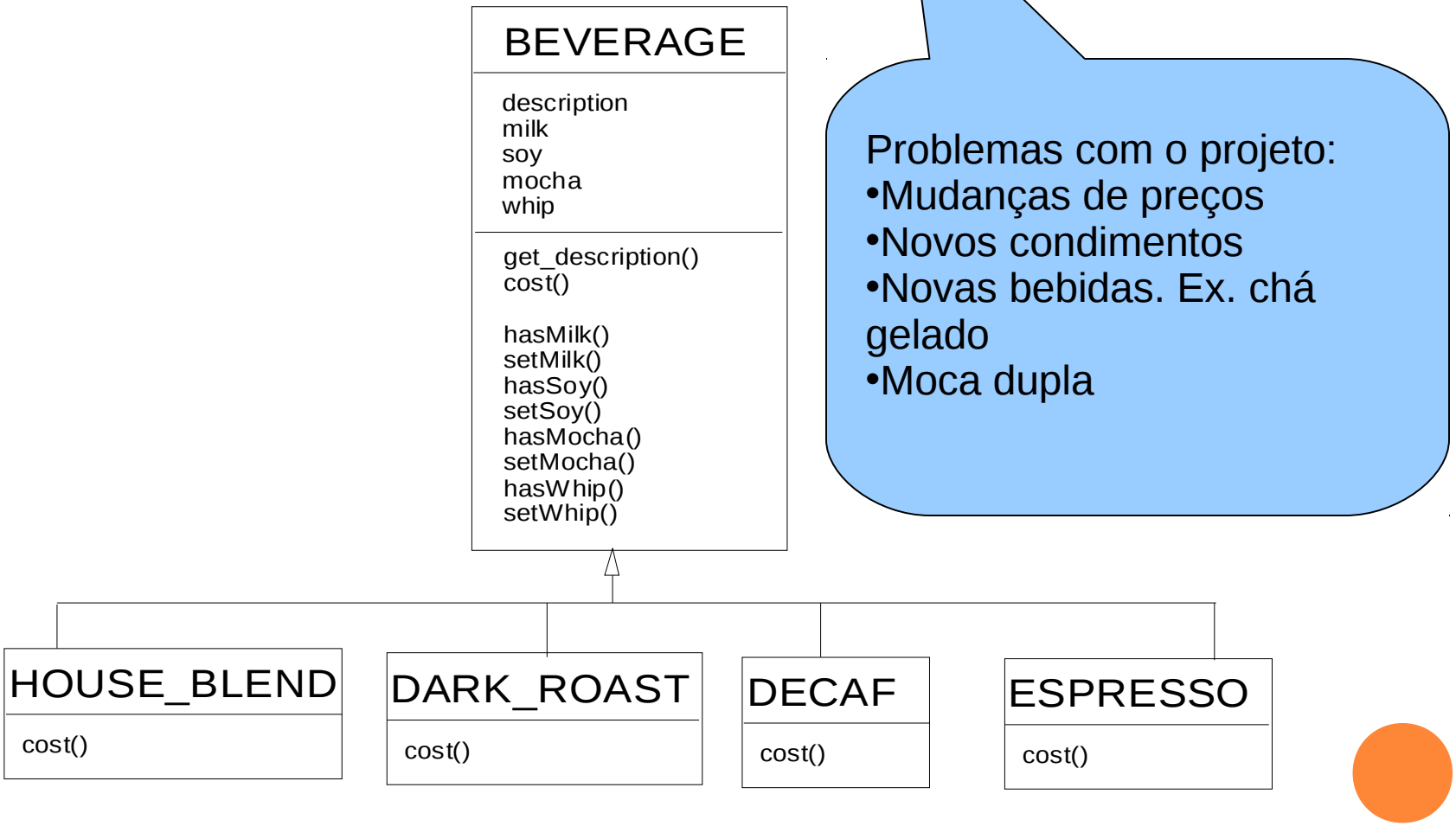
Exemplo: Starbuzz Coffe

- Você pode pedir vários “condimentos”, como leite com espuma, soja e moca, misturados com leite batido.
- Como Starbuzz cobra um valor por cada condimento, eles realmente precisam incluí-los no sistema de pedido.

Outra Opção


BEVERAGE
description
milk
soy
mocha
whip
get_description()
cost()
hasMilk()
setMilk()
hasSoy()
setSoy()
hasMocha()
setMocha()
hasWhip()
setWhip()
calcula o custo dos condimentos
HOUSE_BLEND
cost()
DARK_ROAST
cost()
DECAF
cost()
ESPRESSO
cost()

Outra Opção

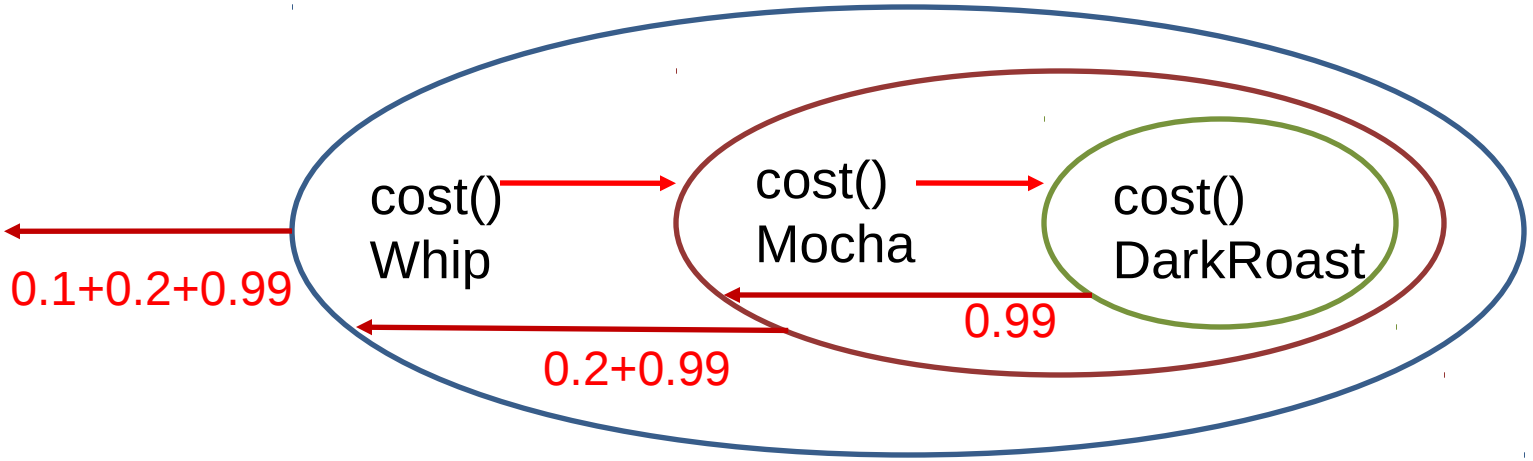


Problemas com o projeto:

- Mudanças de preços
- Novos condimentos
- Novas bebidas. Ex. chá gelado
- Moca dupla

Aplicando o Padrão Decorator ao exemplo Starbuzz

O cliente quer Cafe (Dark Roast) com Moca (Mocha) e Creme(Whip)

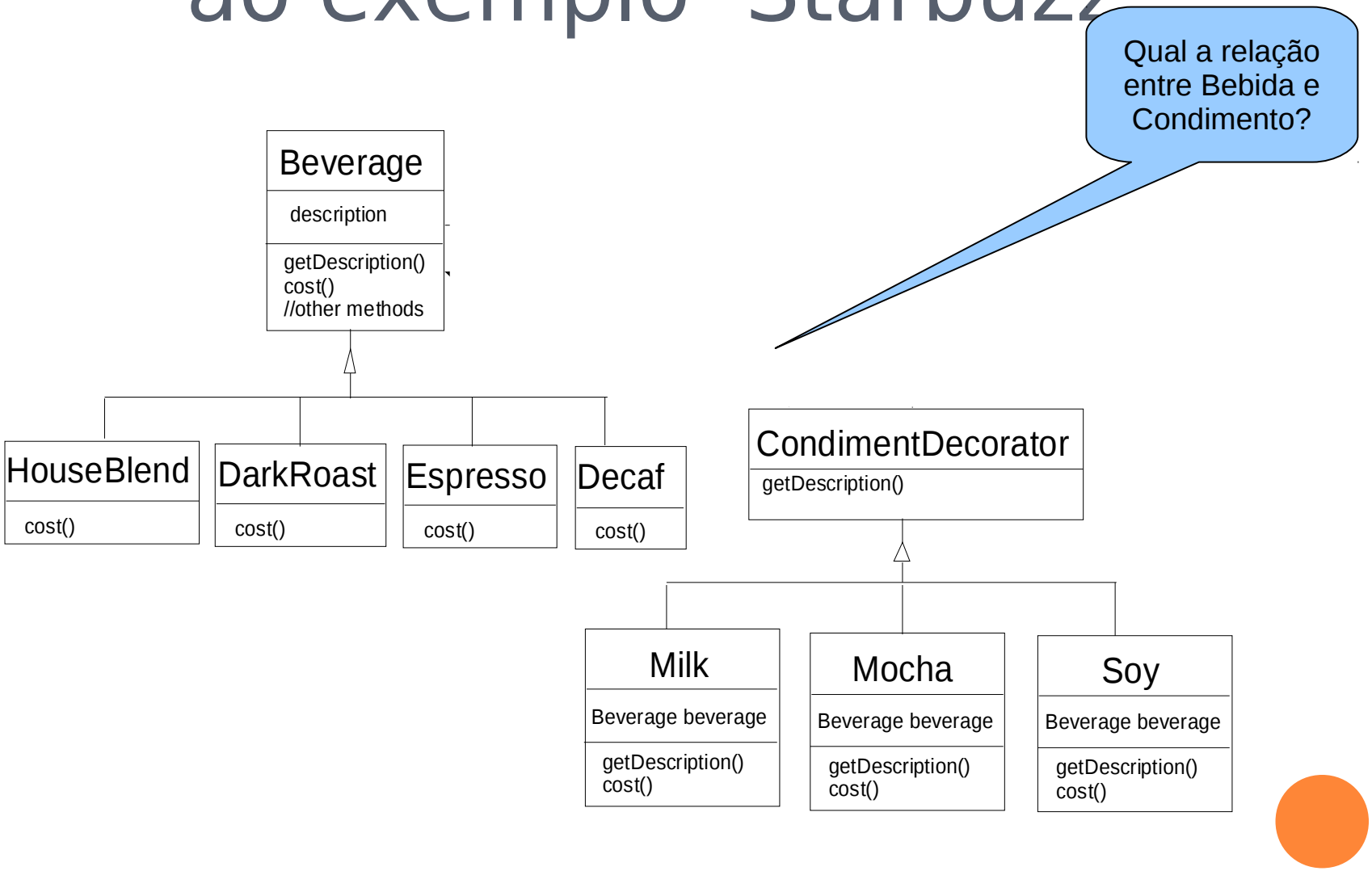


Criamos um Calculando o custo no envoltório o

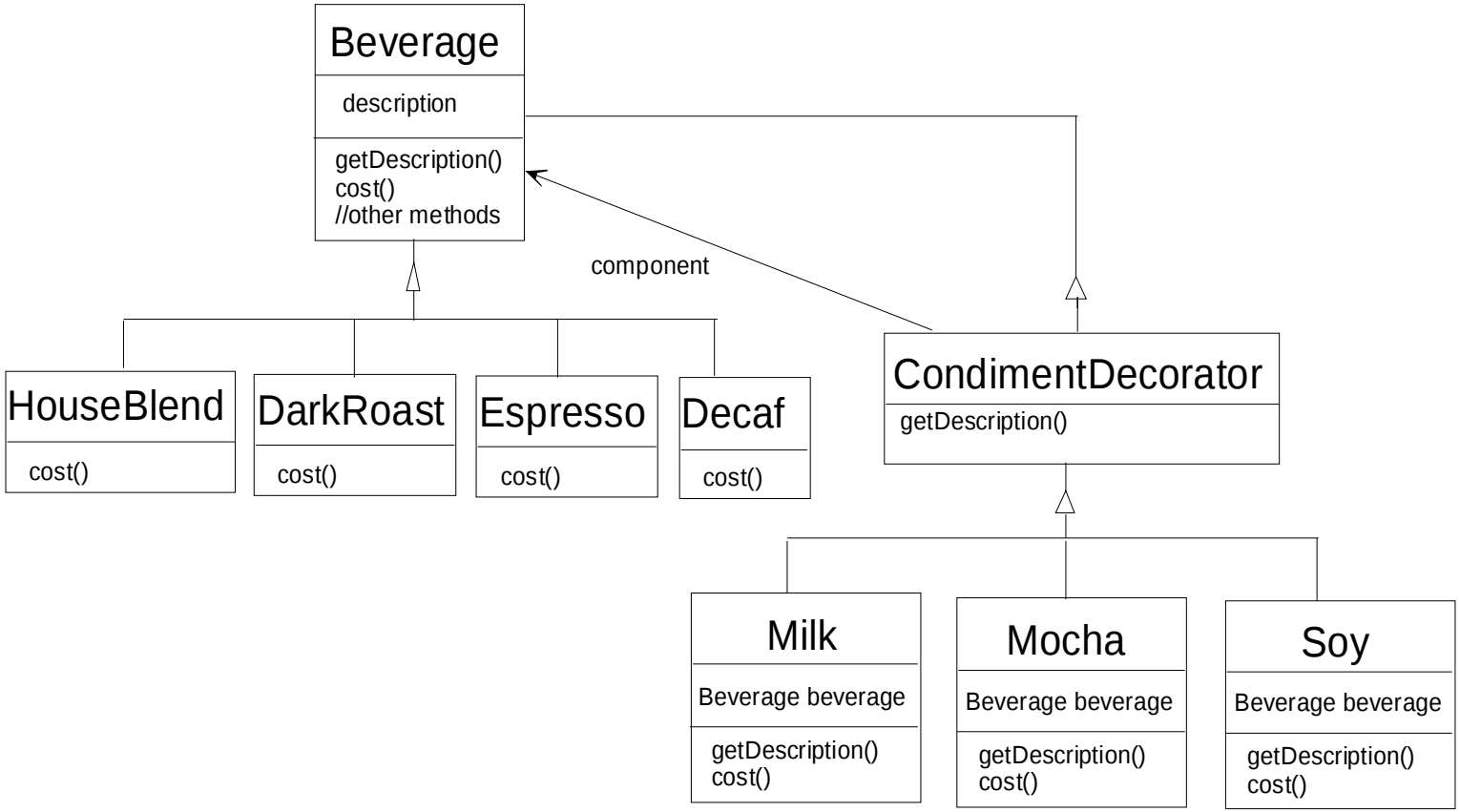
Aplicando o Padrão Decorator ao exemplo Starbuzz

- Os decoradores têm o mesmo supertipo que os objetos que eles decoram
- Pode usar um ou mais decoradores para englobar um objeto
- Uma vez que o decorador tem o mesmo supertipo que o objeto decorado, podemos passar um objeto decorado no lugar do objeto original
- O decorador adiciona seu próprio comportamento
- Quando compomos um decorador com um componente, estamos adicionando um novo comportamento.

Aplicando o Padrão Decorator ao exemplo Starbuzz



Aplicando o Padrão Decorator ao exemplo Starbuzz



Starbuzz

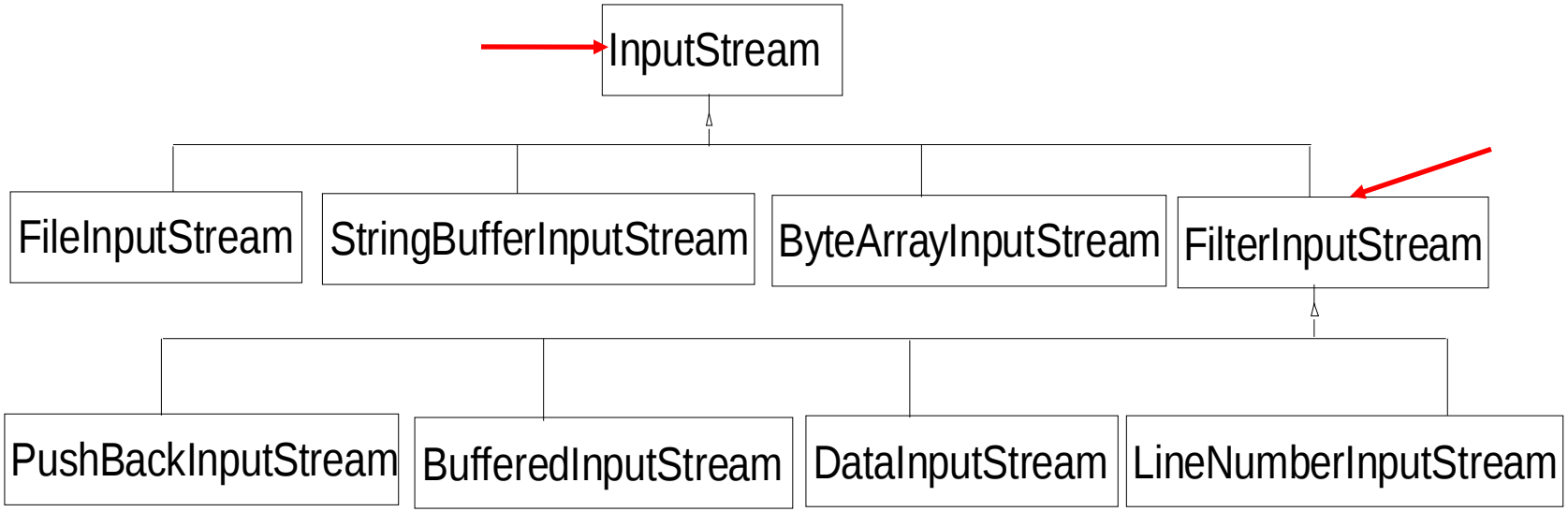
```
public class Mocha extends CondimentDecorator {
   Beverage beverage;
   puclic Mocha(Beverage beverage){
      this.beverage = beverage;
   }
   public String getDescription(){
      return beverage.getDescription()+", Mocha";
   }
   public double cost(){
      return 0.20 + beverage.cost();
   }
```

delegamos a chamada ao objeto que estamos decorando, para que ele possa calcular o custo; depois adicionamos o custo de Mocha.

Starbuzz

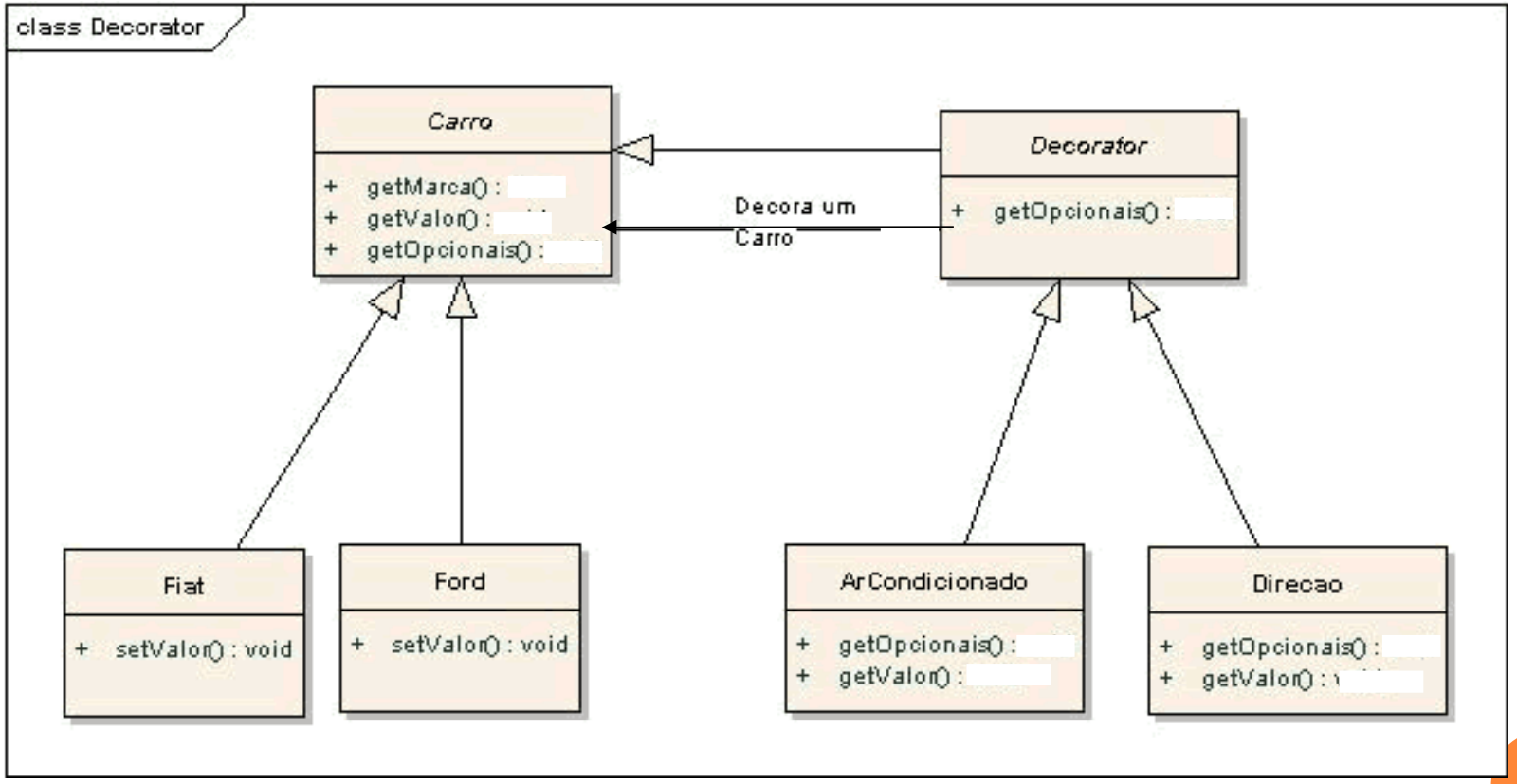
```
public static void main (String args[]){
   Beverage bev3=new DarkRoast();
   bev3 = new Soy(bev3);
   bev3 = new Mocha(bev3);
   bev3 = new Whip(bev3);
   System.out.println(bev3.getDescription()+" Cost:"+
   bev3.cost());
}
```

Java IO API e o Padrão Decorator



PADRÃO DECORATOR

- Exemplo:



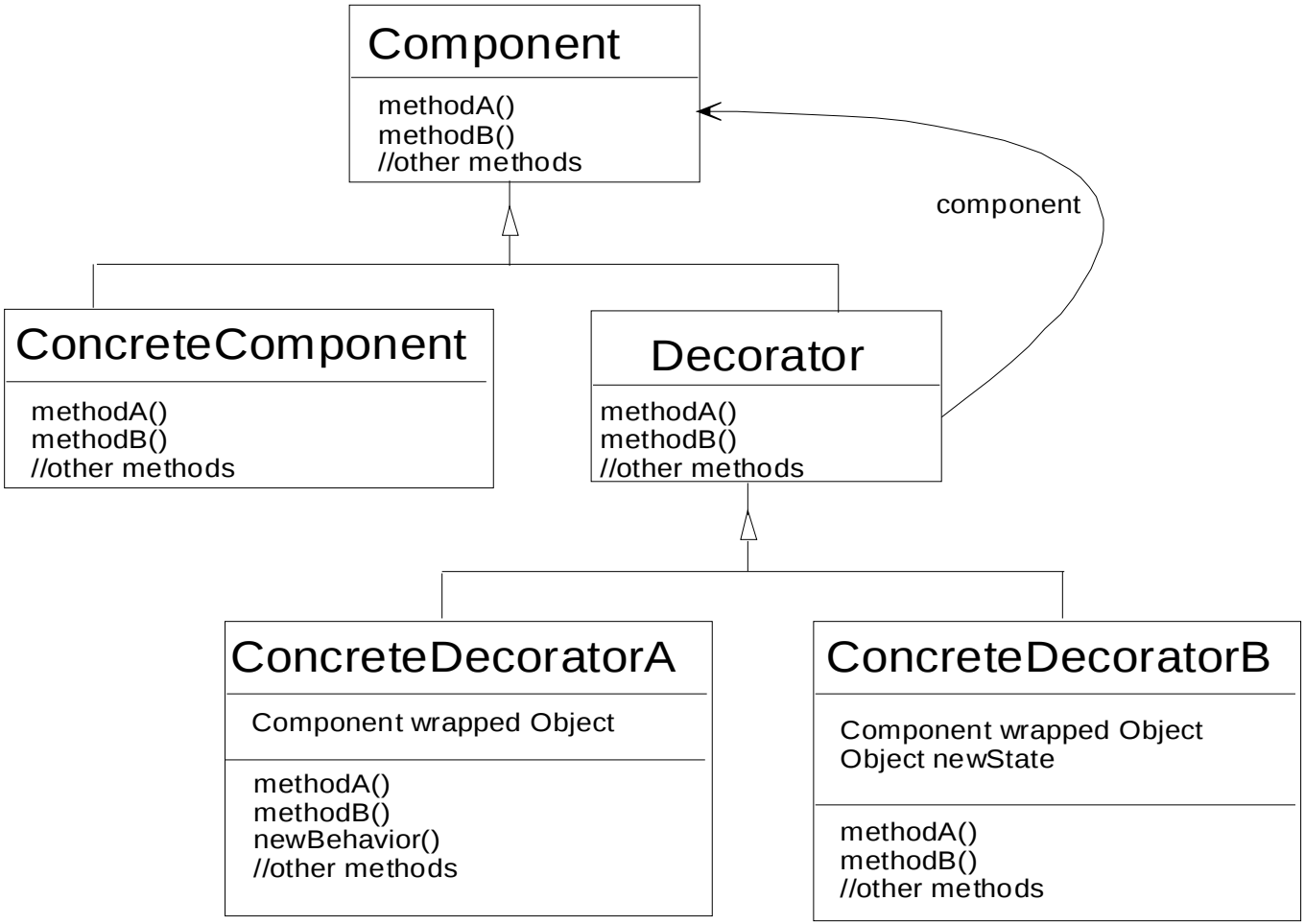
- Fonte: http://www.bdjogos.com/biblioteca_conteudo.php?id=36

PADRÃO DECORATOR

- Aplicabilidade
  - Acrescentar responsabilidades a objetos individuais de forma dinâmica e transparente sem afetar outros objetos.
  - Responsabilidades que são opcionais.

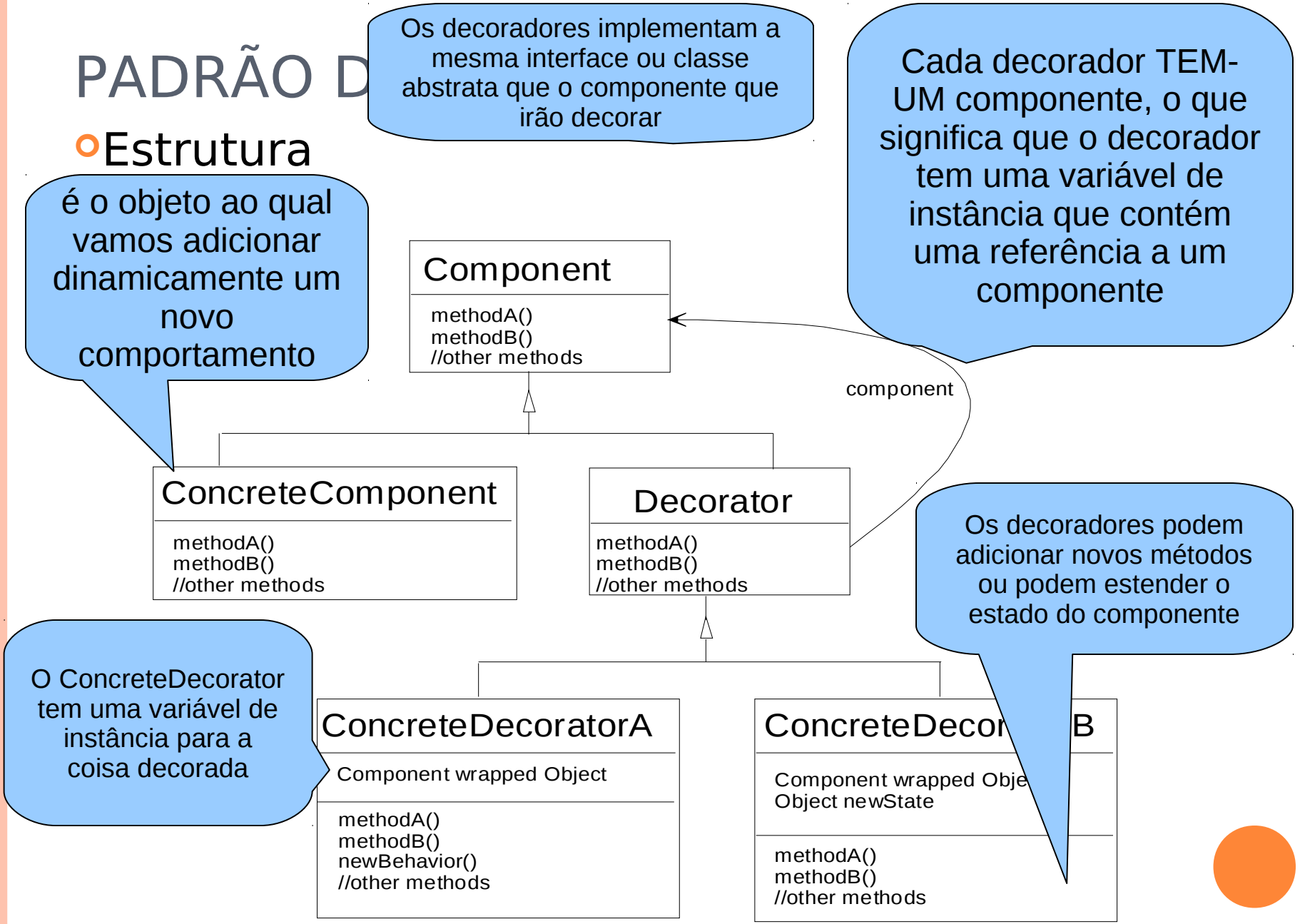
PADRÃO DECORATOR

- Estrutura



PADRÃO D

- Estrutura



PADRÃO DECORATOR

- Participantes
  - Component:
    - A interface do objeto que será “enfeitado”.
  - ConcreteComponent:
    - A implementação “básica” do componente.
  - Decorator:
    - Mantém uma referência para um objeto Component.
    - Sobre-escreve o método (ou os métodos) do componente, chamando a funcionalidade básica e preparando o caminho para os decoradores concretos.
  - ConcreteDecorator:
    - Enfeita o método adicionando algum comportamento ao método

PADRÃO DECORATOR

- Colaborações
  - Decorator repassa solicitações para o seu objeto Component e pode executar operações antes e depois de repassá-las.
- Consequências
  - Maior flexibilidade do que a herança.
  - Evita sobrecarregar classes com métodos que não serão implementados por todas as instâncias.

Princípio Aberto-Fechado e o PADRÃO DECORATOR

- Aberto para extensão e fechado para modificação
  - •Fique à vontade para estender nossas classes com qualquer comportamento novo que desejar.
  - •Estamos fechados para mudanças. Gastamos muito tempo deixando esse código correto e sem erros, por isso não podemos deixá-lo alterar o código existente

Aplicar o princípio Aberto-Fechado em todo lugar é um desperdício desnecessário.

PADRÃO DECORATOR

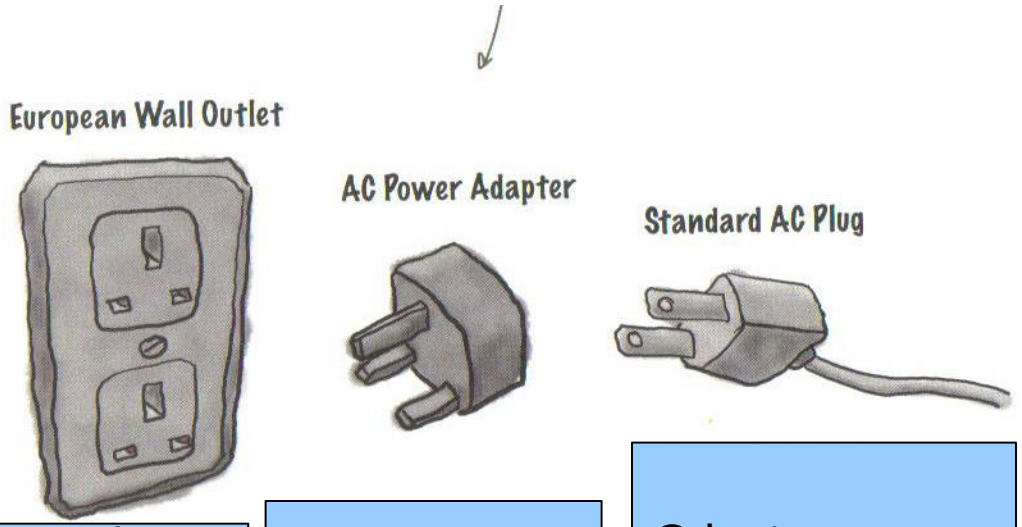
- Comentários adicionais
  - A herança é uma forma de extensão, mas não necessariamente a melhor maneira de obter flexibilidade em nosso projetos
  - Composição e delegação podem ser sempre usadas para adicionar novos comportamentos no tempo de execução.
  - Os decoradores mudam o comportamento de seus componentes adicionando novos recursos antes e/ou depois de chamadas de método para o componente.
  - Os decoradores podem resultar em muitos objetos pequenos em nosso design e o uso exagerado deve ser evitado

PADRÃO ADAPTER

PADRÃO ADAPTER

- Padrão Estrutural
- Objetivo:
  - Converter a interface de uma classe para outra interface que o cliente espera encontrar.
  - Permite que classes com interfaces incompatíveis trabalhem juntas
- Motivação:

Adaptadores no mundo real



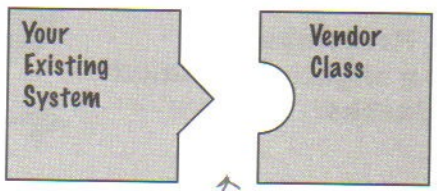
A tomada européia tem uma interface para alimentação elétrica

O adaptador faz a conversão de uma interface para a outra

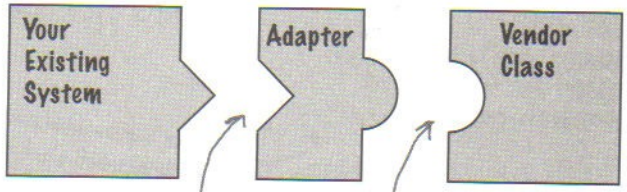
O laptop espera outra interface

Pag 178 – Head First Design Patterns

Adaptadores orientados a objetos

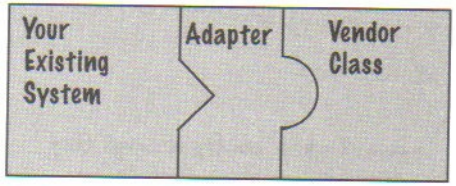


A interface não é compatível com o que você usou no seu código.



O adaptador implementa a interface que as suas classes esperam

O adaptador se comunica com a interface do novo fornecedor



Nenhuma alteração no código

Novo código

Nenhuma alteração no código

Pag 179 Head First Design Patterns

Exemplo: um peru que deseja ser um pato

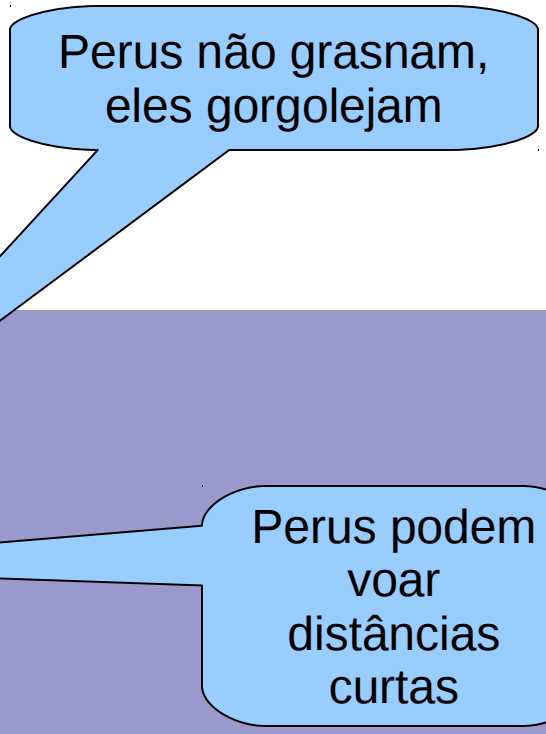
```
public interface Duck {
   public void quack();
   public void fly();
}
```

Uma subclasse de Duck – Mallard Duck

```
public class MallardDuck implements Duck {
    public void quack() {
      System.out.println("Quack");
    }
    public void fly() {
      System.out.println("I'm flying");
    }
}
```

Interface Turkey

```
public interface Turkey {
    public void gobble();
    public void fly();
}
```



Uma subclasse de Turkey - WildTurkey

```
public class WildTurkey implements Turkey {
    public void gobble() {
      System.out.println("Gobble gobble");
    }
    public void fly() {
      System.out.println("I'm flying a short distance");
    }
}
```

Adaptador – peru disfarçado de pato

```
public class TurkeyAdapter implements Duck {
    Turkey turkey;
    public TurkeyAdapter(Turkey turkey) {
      this.turkey = turkey;
    }
    public void quack() {
      turkey.gobble();
    }
    public void fly() {
      for(int i=0; i < 5; i++) {
            turkey.fly();
      }
    }
}
```

Adaptador – peru disfarçado de pato

A interface que seu cliente espera encontrar

O objeto que estamos adaptando

```
public class TurkeyAdapter implements D      {
    Turkey turkey;
    public TurkeyAdapter(Turkey turkey) {
      this.turkey = turkey;
    }
    public void quack() {
      turkey.gobble();
    }
    public void fly() {
      for(int i=0; i < 5; i++) {
            turkey.fly();
      }
    }
}
```

Testando o adaptador

```
public class DuckTestDrive {
    public static void main(String[] args) {
      MallardDuck duck = new MallardDuck();
      WildTurkey turkey = new WildTurkey();
      Duck turkeyAdapter = new TurkeyAdapter(turkey);
      System.out.println("The Turkey says...");
      turkey.gobble();
      turkey.fly();
      System.out.println("\nThe Duck says...");
      testDuck(duck);
      System.out.println("\nThe TurkeyAdapter says...");
      testDuck(turkeyAdapter);
    }
    static void testDuck(Duck duck) {
      duck.quack();
      duck.fly();
    }
}
```

Test run – turkey that behaves like a duck

```
The Turkey says...
Gobble gobble
I'm flying a short distance

The Duck says...
Quack
I'm flying

The TurkeyAdapter says...
Gobble gobble
I'm flying a short distance
I'm flying a short distance
I'm flying a short distance
I'm flying a short distance
I'm flying a short distance
```

PADRÃO ADAPTER

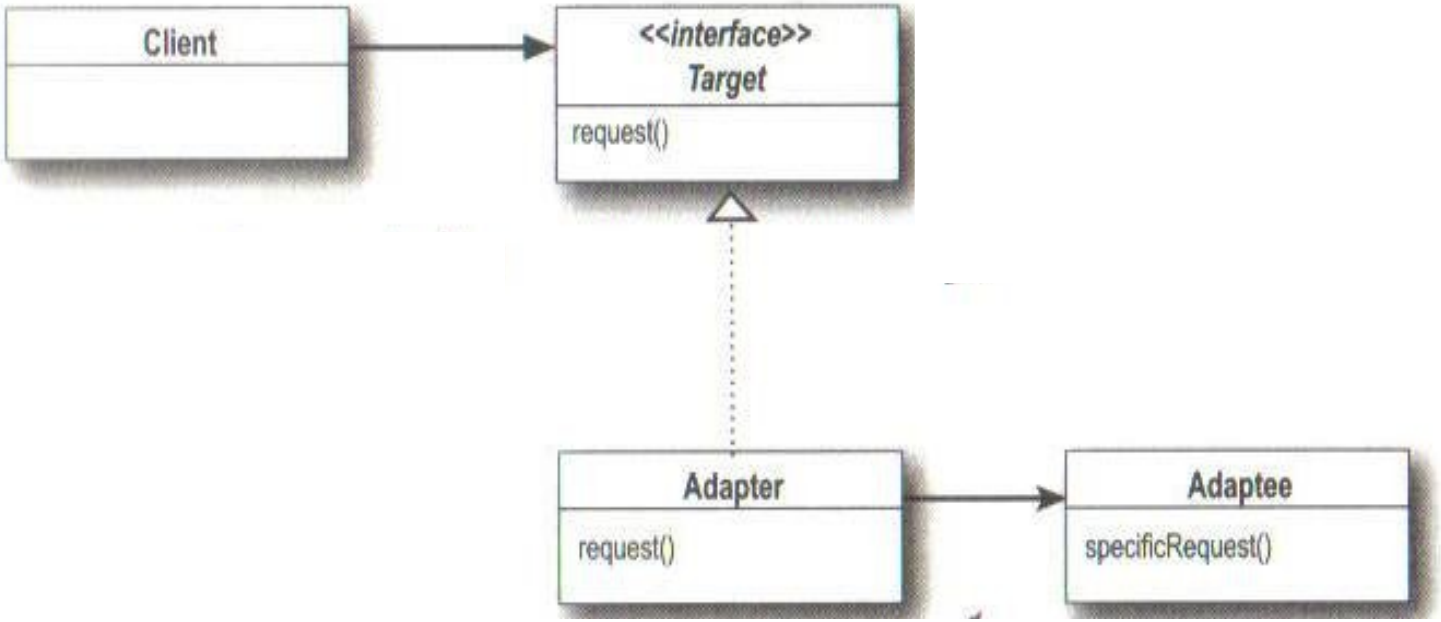
- Exemplo do WebService de Clima:
  - Nossa aplicação “cliente” só requer dois métodos de um WebService:
    - getTemperature(String nomeCidade, Date data)
    - getForecast(String nomeCidade , Date data)
  - O WeatherBug oferece outros métodos:
    - getLocationsList,
    - getLiveWeather,
    - getForecastWeather
  - O Serviço de NOAA dos Estados Unidos oferece:
    - getCityCode,
    - getHistoricalData,
    - ...
  - Nossa aplicação cliente precisa de um adapter para poder utilizar um ou outro WebService indistintamente.

PADRÃO ADAPTER

- Aplicabilidade:
  - Use um adaptador quando:
    - Precisar utilizar uma classe já existente, mas a interface dela não se adéqua ao que sua classe espera.
    - Há mais de uma forma de acessar a um mesmo serviço, mas essas formas não são diretamente compatíveis uma com a outra.

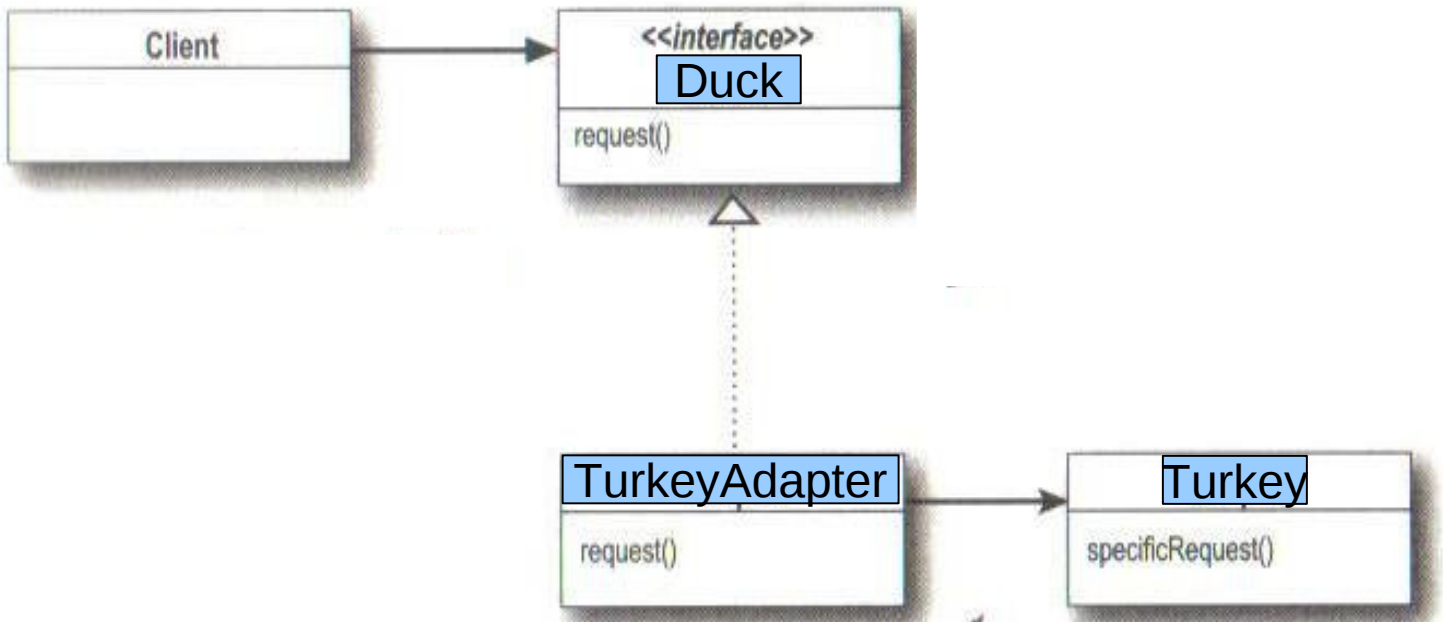
PADRÃO ADAPTER

- Estrutura



PADRÃO ADAPTER

- Estrutura



PADRÃO ADAPTER

- Participantes:
  - Target
    - Define quais operações serão acessadas pelo cliente.
  - Adapter
    - Adapta uma classe determinada ao Target, oferecendo as operações acessadas pelo cliente
  - Adaptee
    - É o módulo pré-existente que oferece o serviço que pretendemos adaptar
  - Client
    - Utiliza apenas a interface Target para acessar os serviços do provedor.
- Cadeia de chamadas: Client -> Target -> ObjectAdapter -> Adaptee.

PADRÃO ADAPTER

- Consequências
  - Permite isolar as adaptações de um módulo já existente em uma única classe.
- Implementação
  - Utilização de Interface para definir as operações a serem utilizadas pelo cliente.
  - Utilização de Composição para utilizar o Adaptee dentro do Adapter.
  - Os resultados das operações do sub-sistema adaptado deverão ser traduzidos a um resultado “compreensível” pelo cliente.

EXERCÍCIO: PADRÃO ADAPTER

- Os primeiros tipos de coleções (ex. Vector e Hashtable) implementavam um método chamado elements(), que retorna uma Enumeração. A interface Enumeration permite que você acesse sequencialmente os elementos da coleção.
- As coleções mais recentes usam a interface Iterator para acessar os elementos da coleção e acrescentam a capacidade de remover itens.
- Desejamos converter uma interface para outra.

EXERCÍCIO: PADRÃO ADAPTER

- Os métodos da interface Enumeration são:
  - hasNextElement()
  - nextElement()
- Os métodos da interface Iterator são:
  - hasNext()
  - next()
  - remove()
- Elabore um Diagrama de Classes que ilustre o seu projeto de software e escreva o código para adaptar a interface Enumeration para a interface Iterator.
- Identifique os participantes do padrão Adapter.

PADRÕES GOF VISTOS

- Padrões de Criação:
  - Singleton
  - Abstract Factory
- Padrões Estruturais:
  - Façade
  - Decorator.
  - Composite.
  - Adapter
- Padrões Comportamentais:
  - Template Method
  - Iterator.
  - Observer.