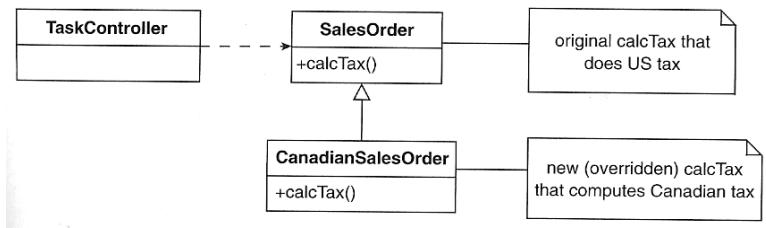
Padrão Strategy (estratégia) – padrão comportamental

- Sistema de comércio electrónico
  - Processar encomendas em diferentes países
    - » Preencher a encomenda
    - » Calcular taxas
    - » Processar a encomenda e emitir um recibo
- "Desenhar com mudança na mente" – não é tentar antecipar a natureza exacta da mudança, mas sim assumir que haverá mudanças, e tentar antecipar onde elas ocorrerão
- Novos requisitos – alterar a forma de cálculo das taxas; calcular taxas para clientes de outros países – adicionar novas regras de cálculo
- Como fazer?

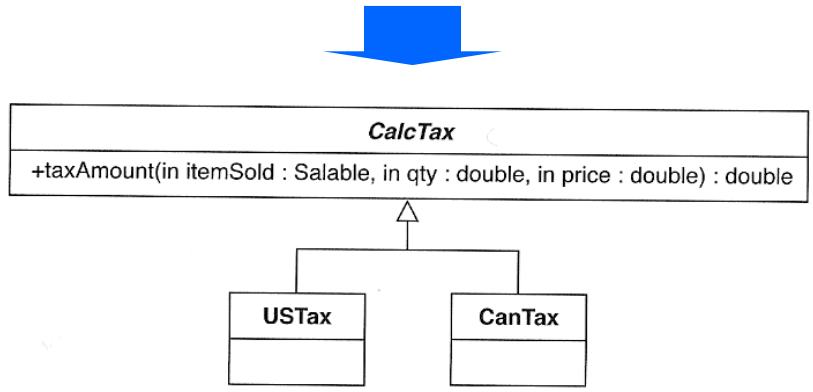
- Copy + paste
- Switches…
- Herança



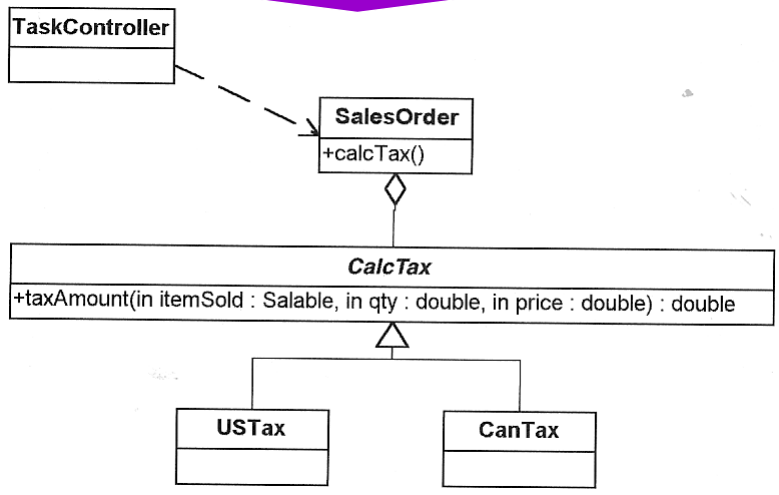
- Segundo GoF
  - Considerar o que deve ser variável no desenho e encapsular o conceito que varia
  - Favorecer agregação-objecto em vez de herança-classe

Fonte: Shalloway e Trott, 2005

- No exemplo – regras de cálculo das taxas variam
  - Para encapsular deverá criar-se uma classe abstracta que define conceptualmente como se calcula uma taxa, derivando-se classes concretas para cada variação



Favorecer agregação em vez de herança



Padrão Strategy

- Definir uma família de algoritmos, encapsular cada um, e torná-los permutáveis; permite que o algoritmo varie independentemente do clientes que o usam (GoF)
- Baseado nos princípios:
  - Objectos têm responsabilidades
  - Implementações diferentes, específicas destas responsabilidades são manifestadas através do polimorfismo
  - Há a necessidade de gerir diferentes implementações do que é, conceptualmente, o mesmo algoritmo
- Boa prática de desenho – separar os comportamentos do domínio do problema uns dos outros
  - alterar a classe responsável por um comportamento sem alterar outras

Padrão Strategy

- Útil quando há um conjunto de algoritmos relacionados e um objecto cliente necessita de escolher dinamicamente um algoritmo desse conjunto para satisfazer a sua necessidade corrente
- Manter a implementação de cada algoritmo numa classe - cada um destes algoritmos encapsulado numa classe separada é designado como *Strategy*
  - Um objecto que usa um Strategy é normalmente referido como um objecto Contexto
  - Todos os Strategy devem ter a mesma interface
    - » Desenhar cada Strategy ou como um implementador de uma interface comum, ou como uma subclasse de uma classe abstracta que declara necessária interface comum
- Separa a implementação do algoritmo do contexto que o usa
  - Alterações no algoritmo, ou novos algoritmos não implicam alterações no contexto nem no cliente

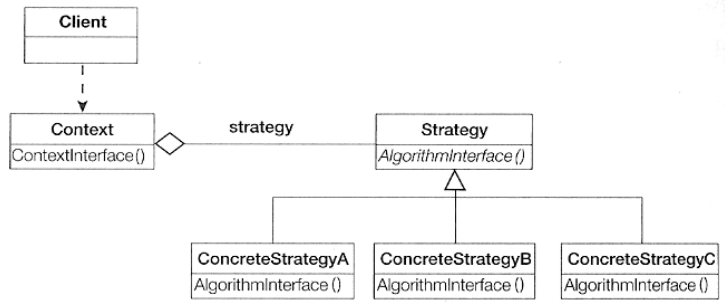
Principais características do Padrão Strategy

- Intenção: Permite usar diferentes regras de negócio ou algoritmos dependendo do contexto onde ocorrem
- Problema: A selecção do algoritmo a ser aplicado depende de um pedido do cliente ou dos dados que estão a ser usados. Se apenas houver uma regra, que não muda, então não é necessário o Strategy
- Solução: Separar a selecção do algoritmo da sua implementação. Permitir que a selecção seja feita com base no contexto
- Participantes e colaboradores:
  - Strategy especifica como são usados os diferentes algoritmos
  - ConcreteStrategies implementam esses algoritmos
  - Context usa um dado ConcreteStrategy com uma referência do tipo Strategy. Startegy e Context interagem para implementar o algoritmo escolhido. O Context reencaminha o pedido do seu cliente para o Strategy
- Consequências:
  - Define uma família de algoritmos
  - Permite eliminar switches e/ou condições
  - Todos os algoritmos têm que ter a mesma interface

Principais características do Padrão Strategy

- Implementação:

  Fazer com que a classe que usa o algoritmo (context) contenha uma classe abstracta (Strategy) que tem um método abstracto que especifica como chamar o algoritmo. Cada classe derivada implementa o algoritmo como pretendido
- Estrutura genérica:



Fonte: Shalloway e Trott, 2005

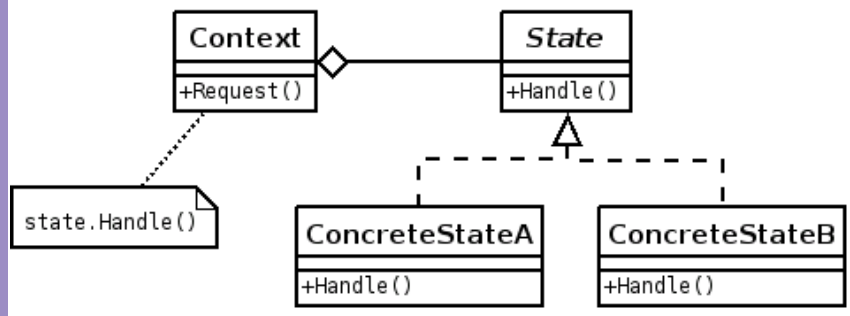
Padrão State (estado) – padrão comportamental

- Estado de um objecto – a condição exacta do objecto num dado momento, dependendo dos valores das suas propriedades ou atributos
  - Alteração nos valores dos seus atributos => mudança de estado
- Exemplo: selecção de uma fonte ou côr num editor HTML – muda propriedades do objecto editor – mudança do seu estado interno
- Padrão State – permite o desenho eficaz da estrutura de uma classe, da qual uma típica instância pode existir em estados diferentes e ter diferentes comportamentos dependendo do estado em que se encontra
  - O ou os comportamentod de um objecto da classe são influenciados pelo seu estado actual
  - No âmbito da terminologia do padrão State, tal classe chama-se classe Contexto
  - Um objecto Contexto pode alterar o seu comportamento quando há uma alteração no seu estado interno e como tal é designado "Stateful"

Padrão State (estado)

- Exemplo de um objecto "Stateful"
  - Editores HTML permitem diferentes vistas de uma página HTML na altura da criação
    - » Vista desenho (desconhecendo HTML)
    - » Vista HTML (personalizar o códifo HTML)
    - » Vista Quick page (preview da página a desenhar)
  - A selecção de uma destas vistas (mudança do estado no objecto Editor) altera o comportamento do objecto Editor, na forma como é mostrada a página Web
  - Padrão State – mover o comportamento específico de um estado para fora da classe Contexto para classes separadas designadas por classes State (estado)

Padrão State – Estrutura genérica:



Exemplo – utilização Padrão State

- Conta num banco com possibilidade de descoberto
- Conta pode existir em cada um dos 3 estados, num dado momento:
  - Sem honorários: desde que o saldo seja superior ao saldo mínimo, não serão cobrados honorários por depósitos ou levantamentos
  - Com honorários – quando o saldo é positivo mas abaixo do saldo mínimo – serão cobrados honorários por qualquer levantamento ou depósito
  - A descoberto – quando o saldo é negativo, mas dentro do limite a descoberto permitido – serão cobrados honorários por qualquer levantamento ou depósito
- Em qualquer um dos estados uma retirada que exceda o valor descoberto definido, não é permitida

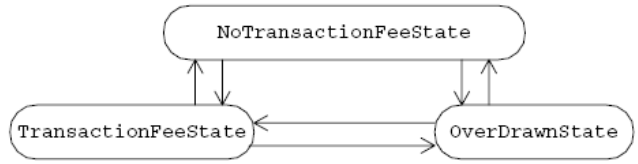
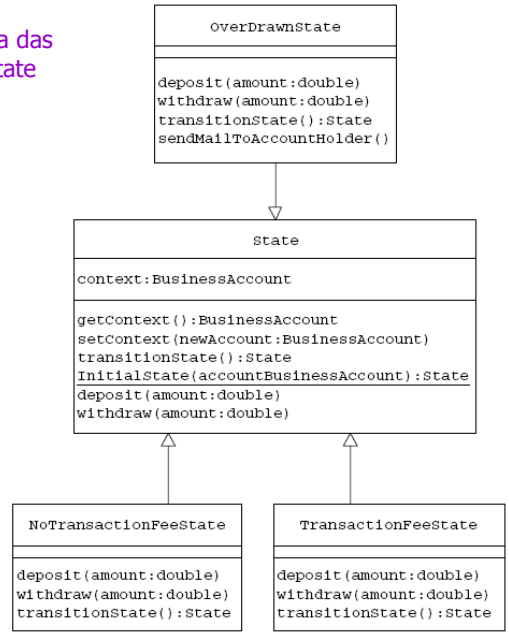


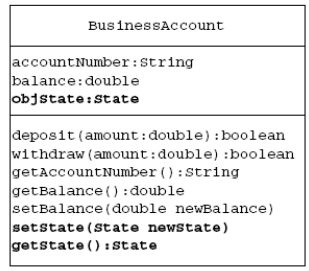
Figure 35.1 State Transitions among Different Account States

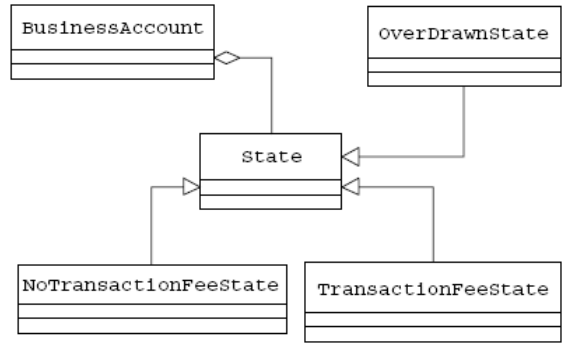
Table 35.1 State Transitions among Different Account States

From	*To*	*What Causes the Transition*
No transaction fee state	Transaction fee state	A withdrawal that can make the balance positive but less than the minimum balance.
	Overdrawn state	A withdrawal that can make the balance negative.
Transaction fee state	No transaction fee state	A deposit that can make the balance greater than the minimum balance.
	Overdrawn state	A withdrawal that can make the balance negative.
Overdrawn state	No transaction fee state	A deposit that can make the balance greater than the minimum balance.
	Transaction fee state	A deposit that can make the balance positive but less than the minimum balance.

Hierarquia das classes State







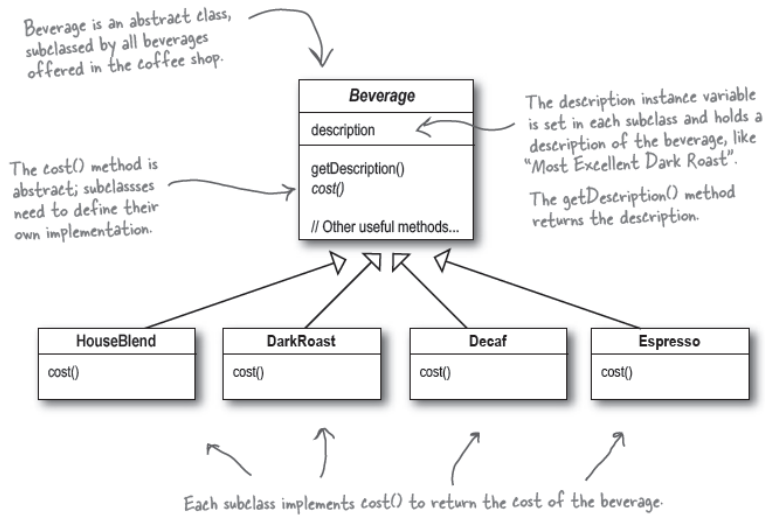
Problema – Coffee Shop

- Empresa vende diferentes tipos de bebidas (cafés)
  - Expresso – .8 euro
  - Descafeinado – .8 euro
  - Suave – .6 euro
  - Cevada – .45 euro
- Cada um destes tipos pode ser condimentado com
  - Baunilha – .4 euro
  - Soja – .35 euro
  - Moka – .3 euro
  - Nata – .5 euro

Que solução?

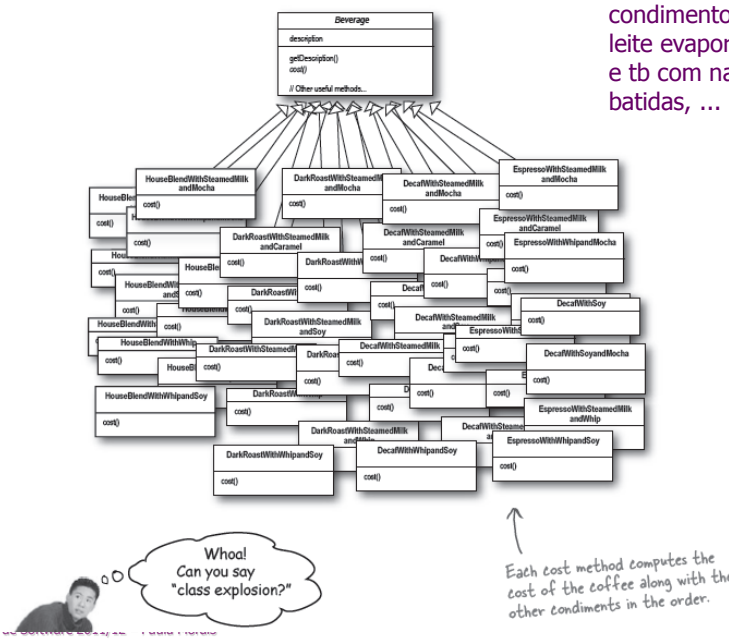
- Pretende calcular o preço dos pedidos de clientes
  - Por exemplo: custo de um expresso com nata
    - Custo de um Suave com Soja e baunilha

Exemplo – Coffee Shop

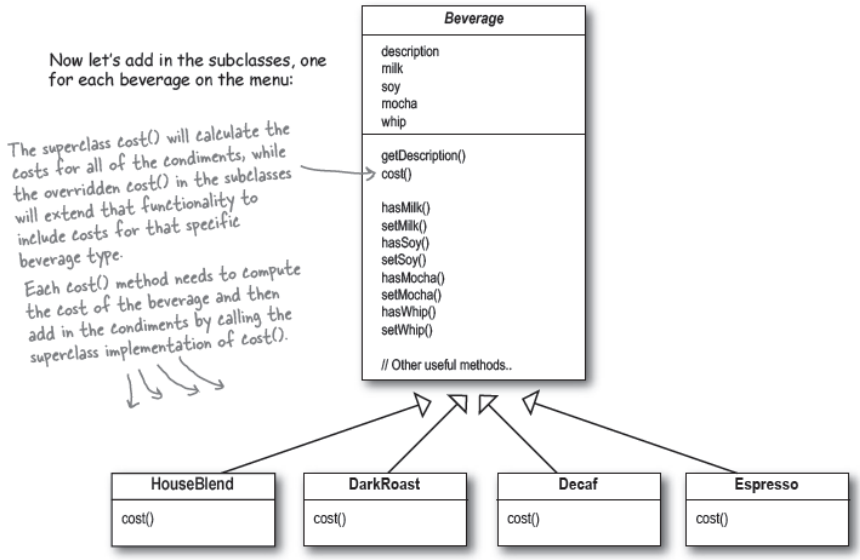


Fonte: Head First design patterns, Elisabeth Freeman, Kathy Sierra, Bert Bates

Cada bebida pode ter condimentos: soja, leite evaporado, moca, e tb com natas batidas, ...



Que solução para evitar a confusão de classes?



Será a melhor solução?

Escreva os métodos cost() para as seguintes classes:

```
public class Beverage {

    public double cost()
    {
    }
}
```

```
Public class DarkRoast extends Beverage
    {
    public DarkRoast() {
    descricao="o melhor Dark Roast";
    }
    public double cost()
    {
    }

}
```

Que requisitos ou outros factores podem mudar e implicar alterações neste desenho?

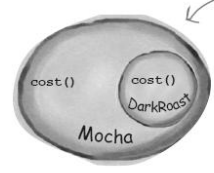
- Representar as bebidas e seus condimentos /preços com herança não é a melhor solução
  - Explosão de classes
  - Desenho rígido
  - Pode-se resultar na adição de funcionalidades na classe base não apropriadas para algumas subclasses

- Melhor: Começar com uma bebida e decorá-la com os condimentos em *runtime*
  - Exemplo: cliente quer um DarkRoast com Moka e Whip
    - » Chamar método cost() e contar com delegação para adicionar os custos dos diferentes condimentos

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.

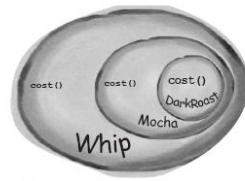
2 **The customer wants Mocha, so we create a Mocha object and wrap it around the DarkRoast.**



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).

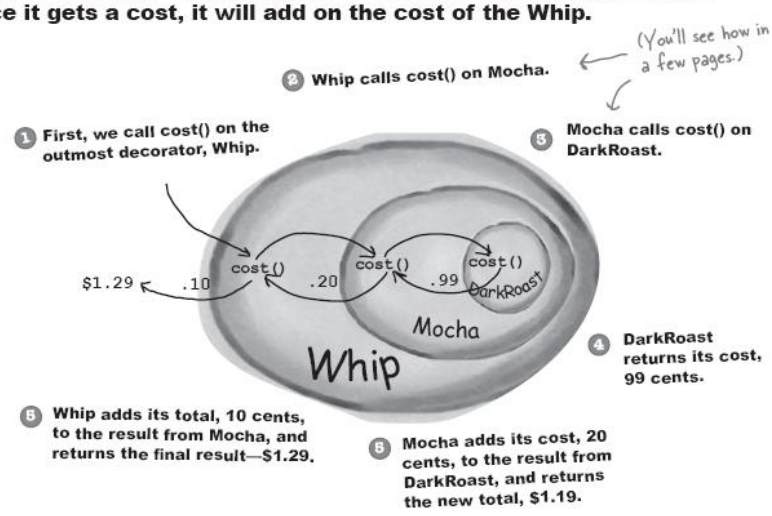
3 **The customer also wants Whip, so we create a Whip decorator and wrap Mocha with it.**



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

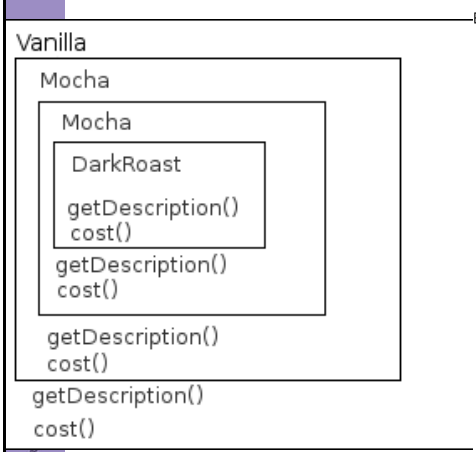
So, a DarkRoast wrapped in Mocha and Whip is still a Beverage and we can do anything with it we can do with a DarkRoast, including call its cost() method.

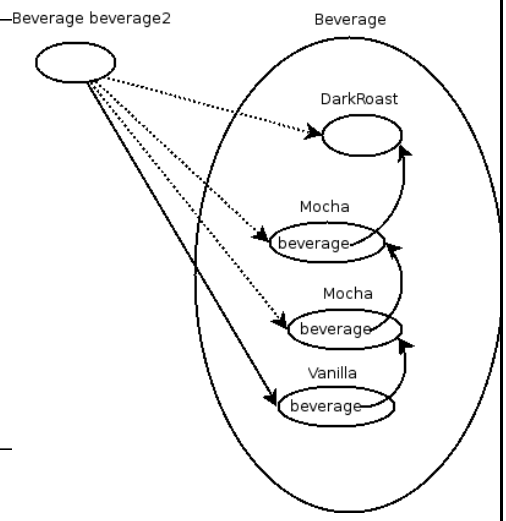
4 Now it's time to compute the cost for the customer. We do this by calling cost() on the outermost decorator, Whip, and Whip is going to delegate computing the cost to the objects it decorates. Once it gets a cost, it will add on the cost of the Whip.



Ou um DarkRoast, duplo moka com baunilha?

Empacotando (wrapping) classes



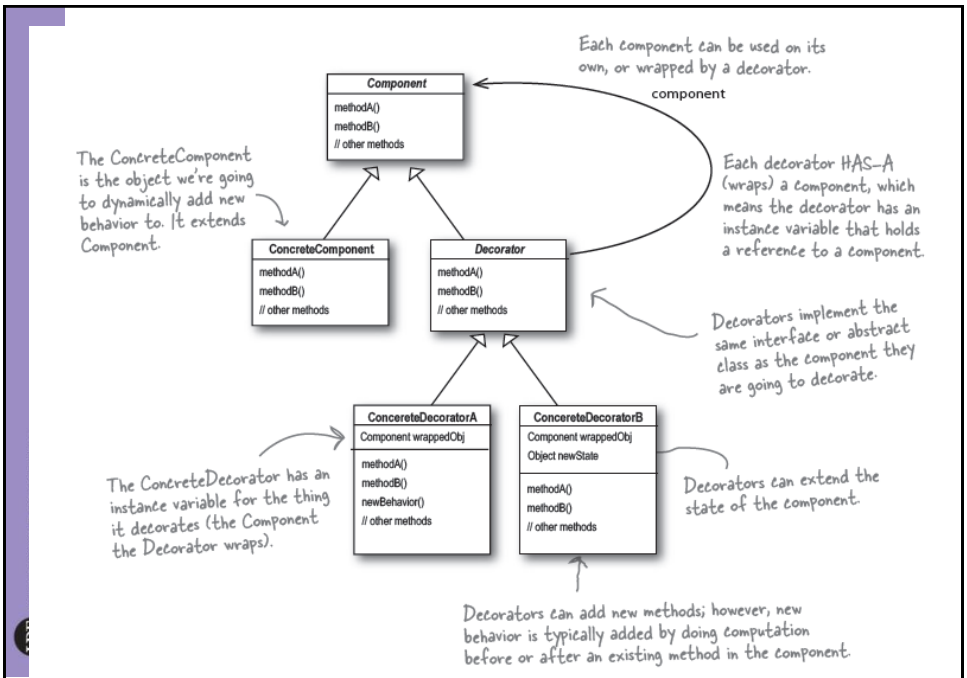


https://zenit.senecac.on.ca/wiki/index.php/Decorator

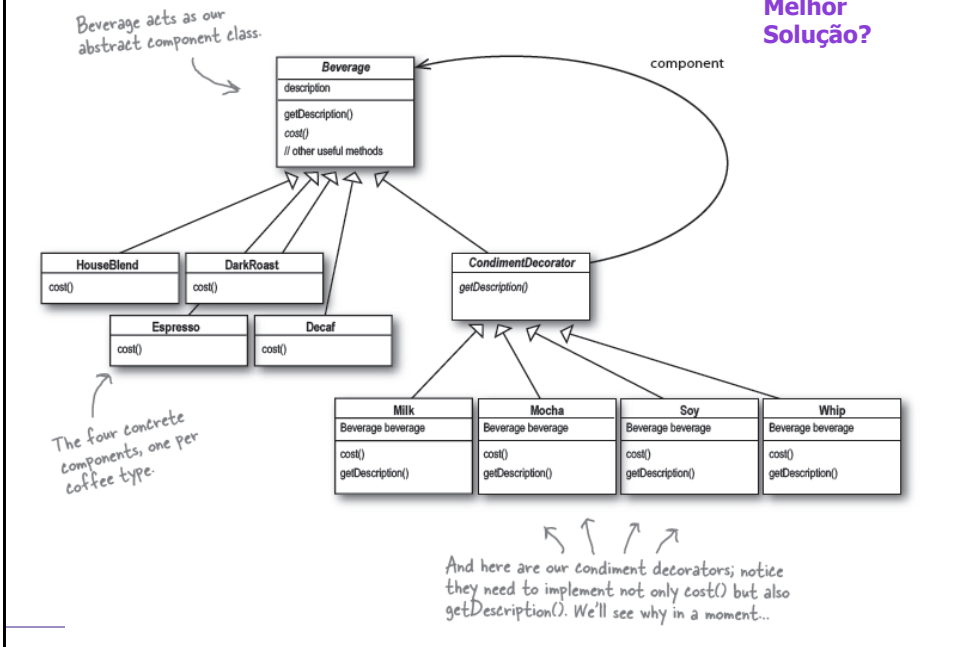
- Decorators têm o mesmo supertipo do objecto que decoram
- Pode-se usar um ou mais decorators para empacotar um objecto
- Sendo que o decorator tem o mesmo supertipo do objecto que ele decora, pode-se passar um objecto decorado em vez do objecto original (wrapped)
- O decorator acrescenta o seu próprio comportamento ou antes e/ou depois de delegar no objecto que ele decora para fazer o resto do trabalho
- Objectos podem ser decorados em qualquer altura, ou seja podem-se decorar os objectos dinamicamente durante a execução, com os decorators que quisermos

Padrão Decorator (estrutural)

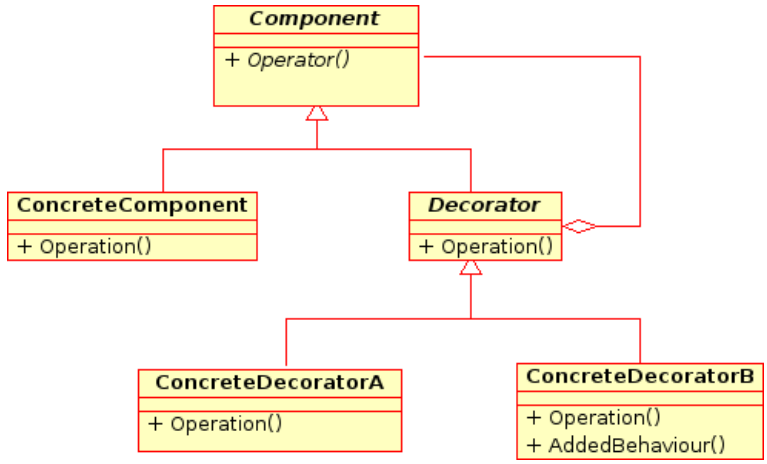
- Acrescenta responsabilidades a um objecto dinamicamente (durante a execução) **através de composição**
- Alternativa à criação de sub-classes que adicionam comportamentos no momento da compilação
- Também é um wrapper
- Empacotar (wrapping) permite adicionar detalhes à componente sem que todas as subclasses herdem estas novas qualidades
- Cada classe decorator empacota uma componente, o que significa que o decorator contém uma variável de instância que guarda uma referência para uma componente



Melhor Solução?



Decorator - Estrutura genérica



- **Usando herança,** o comportamento pode apenas ser determinado estaticamente durante o tempo de compilação – apenas temos o comportamento que a super classe dá ou o que redefinimos (override)
- **Com composição** pode-se misturar decorators como se pretender... *at runtime.*
- **Podem-se implementar novos decorators em qualquer altura para acrescentar comportamentos**. Com herança seria necessário alterar código existente, de cada vez que quiséssemos um novo comportamento

Decorator

- Herança é uma forma de extensão, mas não necessariamente a melhor maneira de conseguir desenhos flexíveis
- No desenho devemos permitir que o comportamento seja estendido sem necessidade de alterar o código existente
- **Composição e delegação** podem, frequentemente, ser usadas para adicionar novos comportamento em runtime
- O padrão Decorator dá uma alternativa à criação de subclasses para estender comportamentos

Exemplo - Pizaria

- Vários tipos de pizas
- Uma encomenda: preparar, cozer, encaixotar
- Que piza "preparar, cozer, encaixotar"
- Pizas do tipo: Queijo, Grega e Pimento, ….

```
Public Piza encomendar(String tipo) {
    Piza piza;
    if (tipo.equals ("queijo"))
            piza = new PizaQueijo();
    else {
    if (tipo.equals ("grega"))
            piza = new PizaGrega();
    else {
    if (tipo.equals ("pimento"))
            piza = new PizaPimento(); }}

    piza.preparar();
    piza.cozer ();
    piza.encaixotar();
    return  piza;
}
```

Mais tipos de pizas?

Solução?

- 1º encapsular a criação do objecto
  - Retirar da classe a parte que varia

```
Public Piza encomendar(String tipo) {
    Piza piza;

    piza.preparar();
    piza.cozer ();
    piza.encaixotar();
    return  piza; }
```

```
if (tipo.equals ("queijo"))
                piza = new
PizaQueijo();
        else {
        if (tipo.equals ("grega"))
        piza = new PizaGrega();
        else {
        if (tipo.equals ("pimento"))
        piza = new PizaPimento();
}}
```

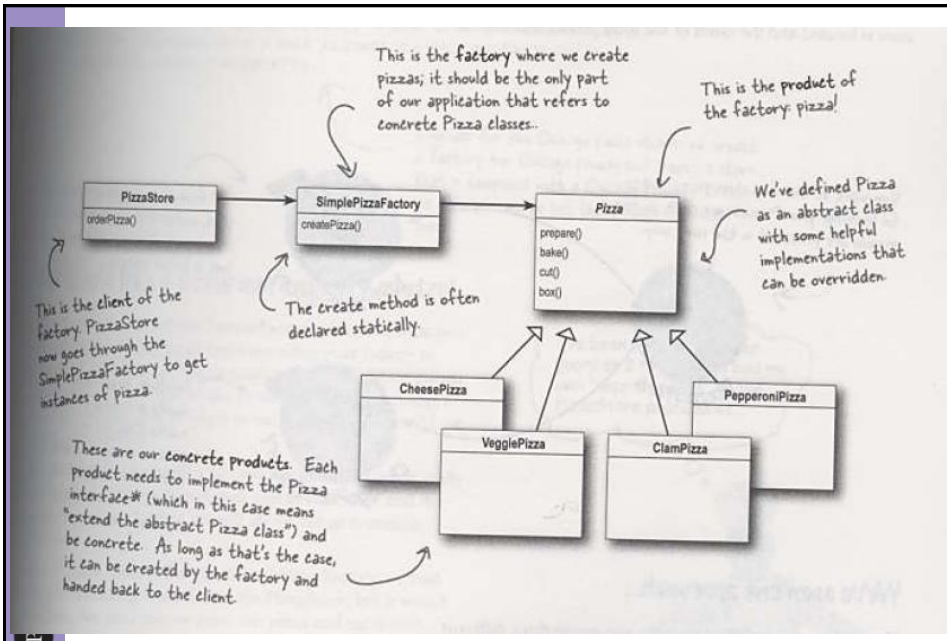
Objecto que se preocupa apenas com a criação de pizas

Padrão Factory

```
public class PizaFactory {
public Piza criarPiza(String tipo) {
Piza piza = null;
if (tipo.equals ("queijo"))
    piza = new PizaQueijo();
    else {
    if (tipo.equals ("grega"))
    piza = new PizaGrega();
    else {
    if (tipo.equals ("pimento"))
    piza = new PizaPimento(); }
Return piza}
}}
```

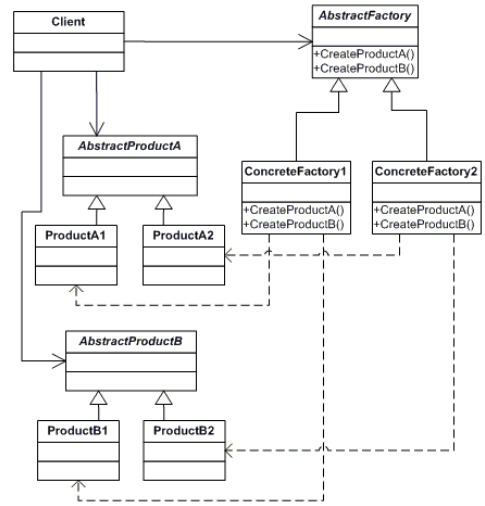
```
public class Pizaria
{
PizaFactory factory;

Public Pizaria (PizaFactory factory)
    {this.factory = factory;
}
Public Piza encomendar(String tipo) {
    Piza piza;
    piza = factory.criarPiza(tipo);
    piza.preparar();
    piza.cozer ();
    piza.encaixotar();
    return  piza; }
}
```

Padrão Factory – padrão de criação

- Criação de objectos dinamicamente
- Define uma interface para a criação de um objecto, mas deixa as subclasses decidir que classe instanciar



Exemplo de utilização conjunta de padrões

- Problema: simulação do comportamento de patos, em que se registam os "quacks" de cada pato
- Adicionar gansos à simulação
  - Qual o padrão que permitirá, facilmente, que se simulem comportamentos de gansos ou patos, reutilizando classes já existentes'

?

Precisamos de um **adapter** para adaptar um ganso a um pato:

```
public class GansoAdapter implements Quackable{

ganso Ganso;

    public GansoAdapter(ganso Ganso){
        this.Ganso = Ganso;
    }
    public void quack() {
        Ganso.grasna();
    }
}
```

Um Adapter implementa a interface desejada

O construtor recebe o ganso que vamos adaptar

Quando quack() é chamado, a chamada é delegada para o método grasna() do ganso

Agora é preciso utilizar gansos na simulação:

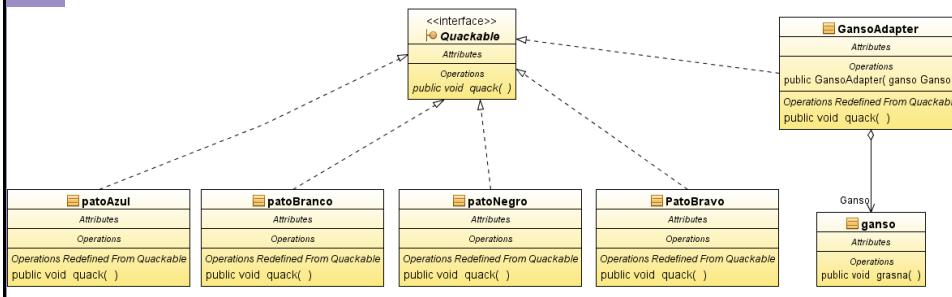
```
void simula() {
    Quackable patobravo = new PatoBravo();
    Quackable patoazul = new patoAzul();
    Quackable patobranco = new patoBranco();
    Quackable patonegro = new patoNegro();
    Quackable gansoPato = new GansoAdapter(new ganso());

    System.out.println("\nPato simulacao com Gansos");
    simula(patobravo);
    simula(patoazul);
    simula(patobranco);
    simula(patonegro);
    simula (gansoPato);
}
```

Fazemos com que um ganso actue como um pato, empacotando-o num GansoAdapter

Depois de empacotar o ganso, podemos tratá-lo tal como qq outro pato - quackable

Diagrama de classes com **adapter**



- E contar o nº de "quacks" de um grupo de patos?
  - Como faze-lo sem alterar as classes pato?
  - Que padrão poderia ajudar?

?

Precisamos de um **decorator** que adiciona comportamento aos patos

Tal como o Adapter, necessitamos de implementar a interface desejada

```
public class PatoCounterDecorator implements Quackable {
   Quackable pato;
   static int numeroQuacks;
   public PatoCounterDecorator (Quackable pato) {
      this.pato = pato;
   }
   public void quack() {
      pato.quack();
      numeroQuacks ++;
   }
   public static int getQuacks () {
      return numeroQuacks;
   }
}
```

Variável de instância referencia o pato que estamos a decorar e variável de classe

Conta o nº geral de quacks

Recebemos a refrência do pato que estamos a decorar no construtor

Quando quack() é chamado, a chamada é delegada para o pato que estamos a decorar

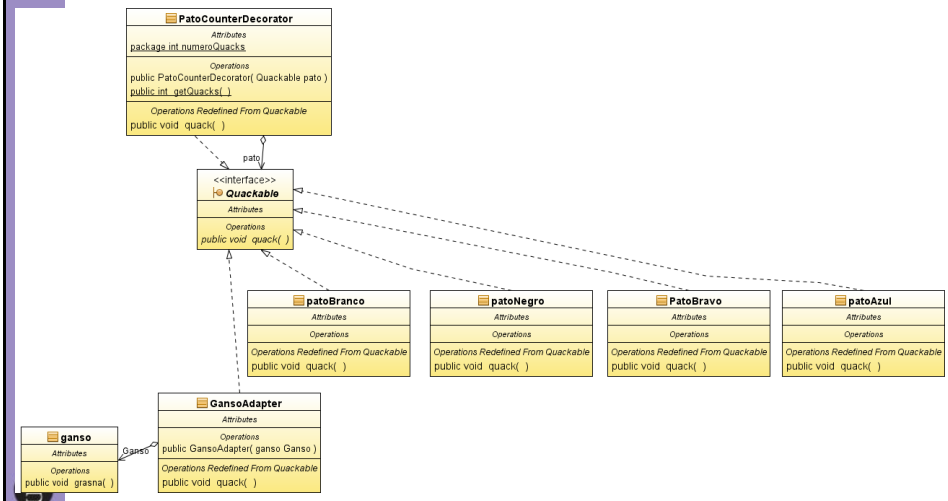
Agora é preciso empacotar cada objecto Quackable instanciado num decorator

```
public static void main(String[] args) {
      Main simulador = new Main();
      simulador.simula();
   }
   void simula() {
      Quackable patobravo = new PatoCounterDecorator (new PatoBravo());
      Quackable patoazul = new PatoCounterDecorator (new patoAzul());
      Quackable patobranco = new PatoCounterDecorator (new patoBranco());
      Quackable patonegro = new PatoCounterDecorator (new patoNegro());

      Quackable gansoPato = new GansoAdapter(new ganso());
      System.out.println("\nPato simulacao com Gansos");
      simula(patobravo);
      simula(patoazul);
      simula(patobranco);
      simula(patonegro);
      simula (gansoPato);
      System.out.println("Os patos fizeram " + PatoCounterDecorator.getQuacks() + "
    quacks");
   }
   void simula(Quackable pato)
   {pato.quack();} }
```

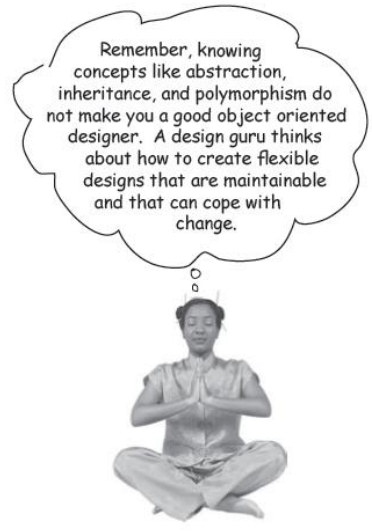
De cada vez que criamos um Quackable, empacotamo-lo com um decorator

Diagrama de classes com **decorator**



Resumindo – porquê padrões

- **Algúem já resolveu os seus problemas**
- *The Timeless Way of Coding Java*



Head First design patterns, Elisabeth Freeman, Kathy Sierra, Bert Bates

UNIVERSIDADE PORTUCALENSE

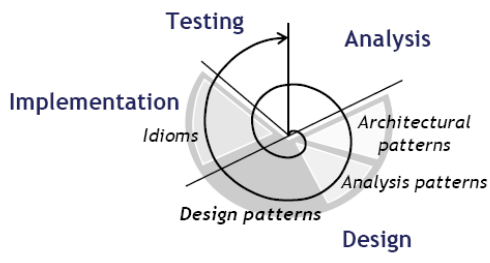
- A questão **não é usar ou não padrões** de desenho no processo de desenho de sw, **mas que padrões usar**, se algum.
- *"The object—oriented model makes it easy to build up programs by accretion. What this often means, in practice, is that it provides a structured way to write spaghetti code"* — ***Paul Graham***
- *"In a room full of top software designers, if any two of them agree, that is a majority"* — ***Bill Curtis***
-

- A aproximação OO ao desenho de software, tenta gerir a complexidade inerente aos problemas do mundo real, abstraindo o conhecimento e encapsulando-o nos objectos
- Identificar os objectos, relacionamentos e interacções apropriados é o objectivo de um desenho OO com sucesso, mas tal não é fácil
- Os sistemas OO são desenhados para reutilização, manutenção e modificação fáceis
- Os padrões podem ser usados como ferramentas no processo de desenho porque oferecem soluções a problemas comuns
- Qualquer padrão usado no desenvolvimento OO reflecte implicitamente conceitos de OO, como objectos, classes, herança, encapsulamento, polimorfismo, etc. Para entender os padrões, devem-se perceber bem estes conceitos

Padrões

- Programar para uma interface, não uma implementação [Gamma95, p.18]
- Favorecer composição de objectos em detrimento de herança de classes [Gamma95, p.20]
  - Preferir relações dinâmicas (runtime) a relações estáticas

Lembrar: Utilização de padrões no ciclo de vida de desenvolvimento de SIBC



The OO software development life–cycle traditionally consists of an analysis, design, implementation, and testing phase, which may be overlapping or re–iterated as dictated by the OO method used, each time refining the design and implementation.

Different categories of patterns are used in different phases of the life–cycle. *Architectural patterns* have large design granularity and are used early in the design phase. *Analysis patterns* target the domain. *Design patterns* have medium granularity and can be used throughout the entire design phase, but are also closely related to the implementation. *Idioms* have the smallest granularity and are connected with a specific language.

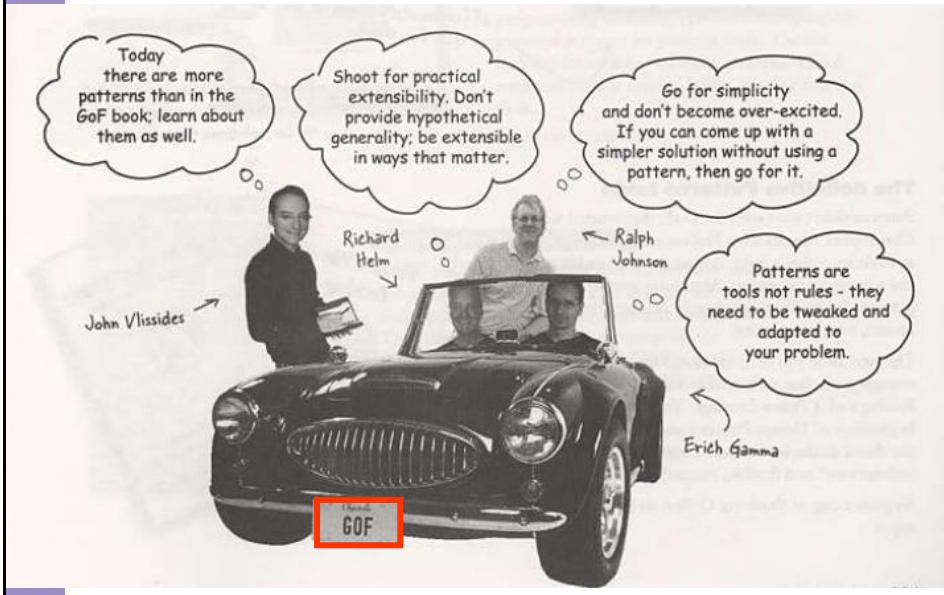
Padrões usados em diferentes fases do ciclo

Gunni Rode — http://www.rode.dk/thesis

Diferentes categorias de padrões

Category	Description	Target
Architectural Patterns De arquitectura	An architectural pattern expresses a fundamental structural organisation schema for software systems. It provides a set of predefined sub–systems, specifies their responsibilities, and include rules and guidelines for organising relationships between them [Buschmann96, p.12].	Entire (sub–) systems, applications, and frameworks
Analysis Patterns De análise	An analysis pattern reflects the conceptual structures of business processes rather than actual software implementations [Fowler97, p.XV].	Domain and Business Object Model
Design Patterns **De desenho**	A design pattern provides a scheme for refining the sub–systems or components of a system, or the relationships between them [Buschmann96, p.13]. It does so by describing communicating objects and classes that are customised to solve a general design problem in a particular context [Gamma95, p.3].	Micro–architectures within sub–systems or components
Idioms	An idiom is as a low–level pattern, specific to a particular programming language that describes how to implement particular aspects of components or the relationships between them using the features of the given language [Buschmann96, p.14]. An implementation of a design pattern that is unique to the language chosen is also considered an idiom in this thesis.	Classes, Objects, and Methods

Gunni Rode: http://www.rode.dk/thesis



Head First design patterns, Elisabeth Freeman, Kathy Sierra, Bert Bates