Computação Orientada a Objetos

Padrões de Projeto
Composite e Observer

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

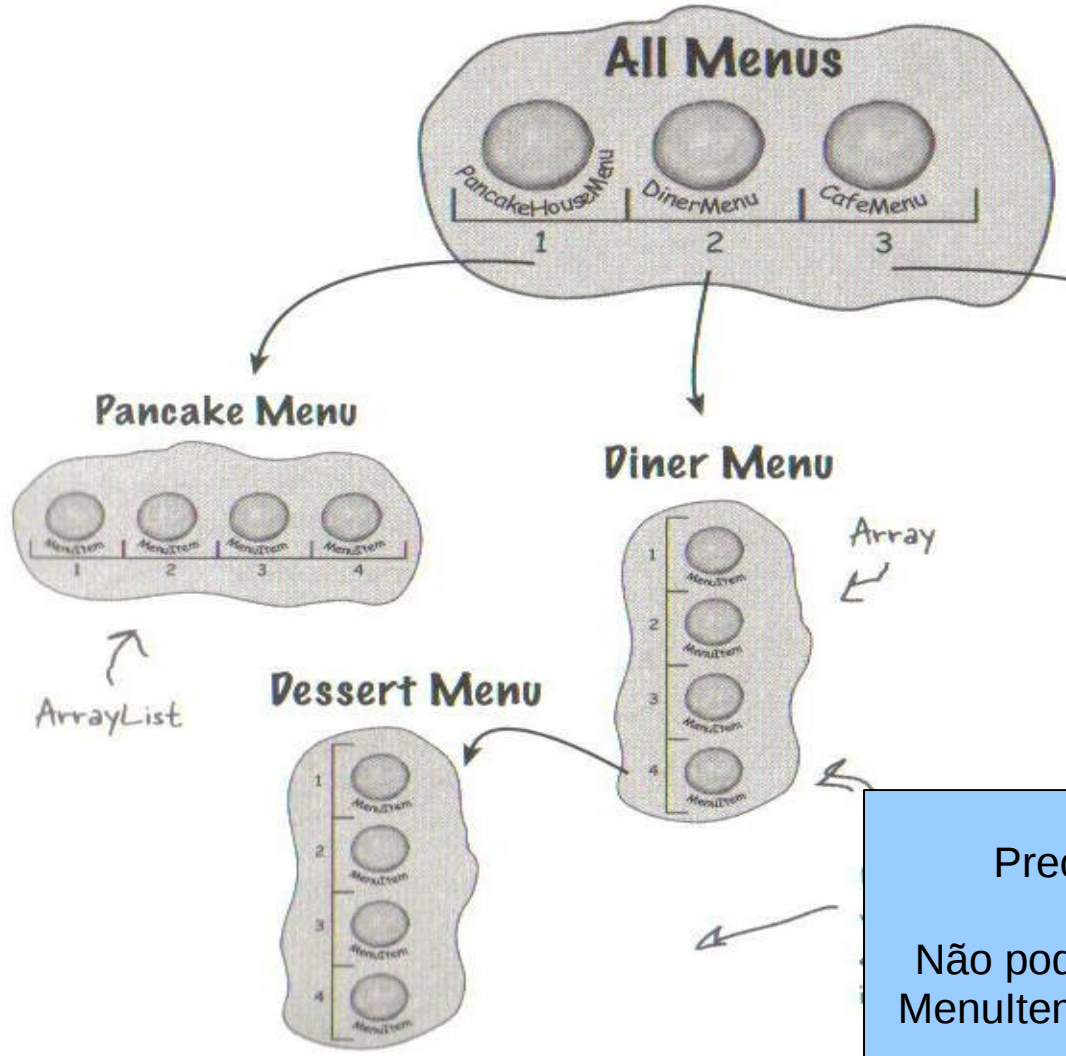
Profa. Karina Valdivia Delgado
EACH-USP

PADRÃO COMPOSITE

- Padrão Estrutural
- Objetivo:
  - Compor objetos em estruturas de árvore para representar hierarquias todo-parte.
  - Composite permite que clientes tratem objetos individuais e composições de objetos de maneira uniforme.
- Motivação:

  (Ex. pag. 266 Head First)

What we want (something like this):



Aqui está nosso ArrayList contendo os menus de todos os restaurantes

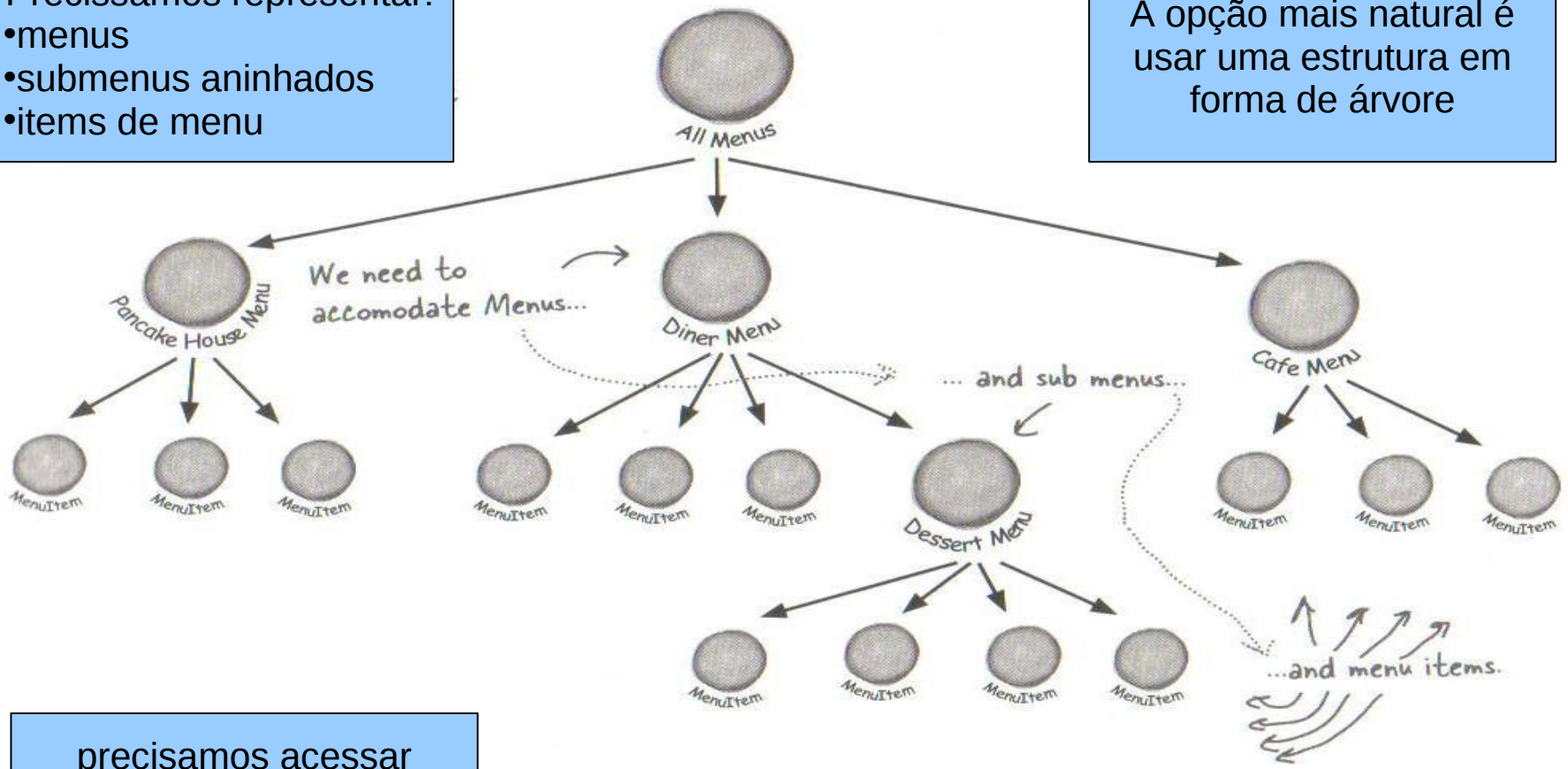
Precisamos incluir um submenu ao menu Diner Menu.
Não podemos atribuir um menu ao vetor MenuItem, porque os tipos são diferentes!

But this won't work!

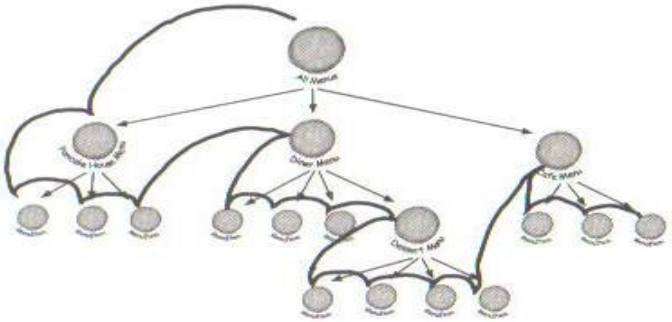
Teremos que mudar alguma coisa!

Precissamos representar:
- menus
- submenus aninhados
- items de menu

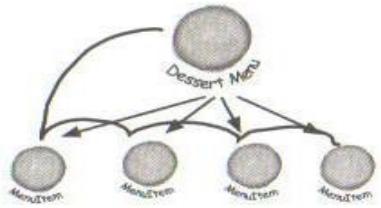
A opção mais natural é usar uma estrutura em forma de árvore



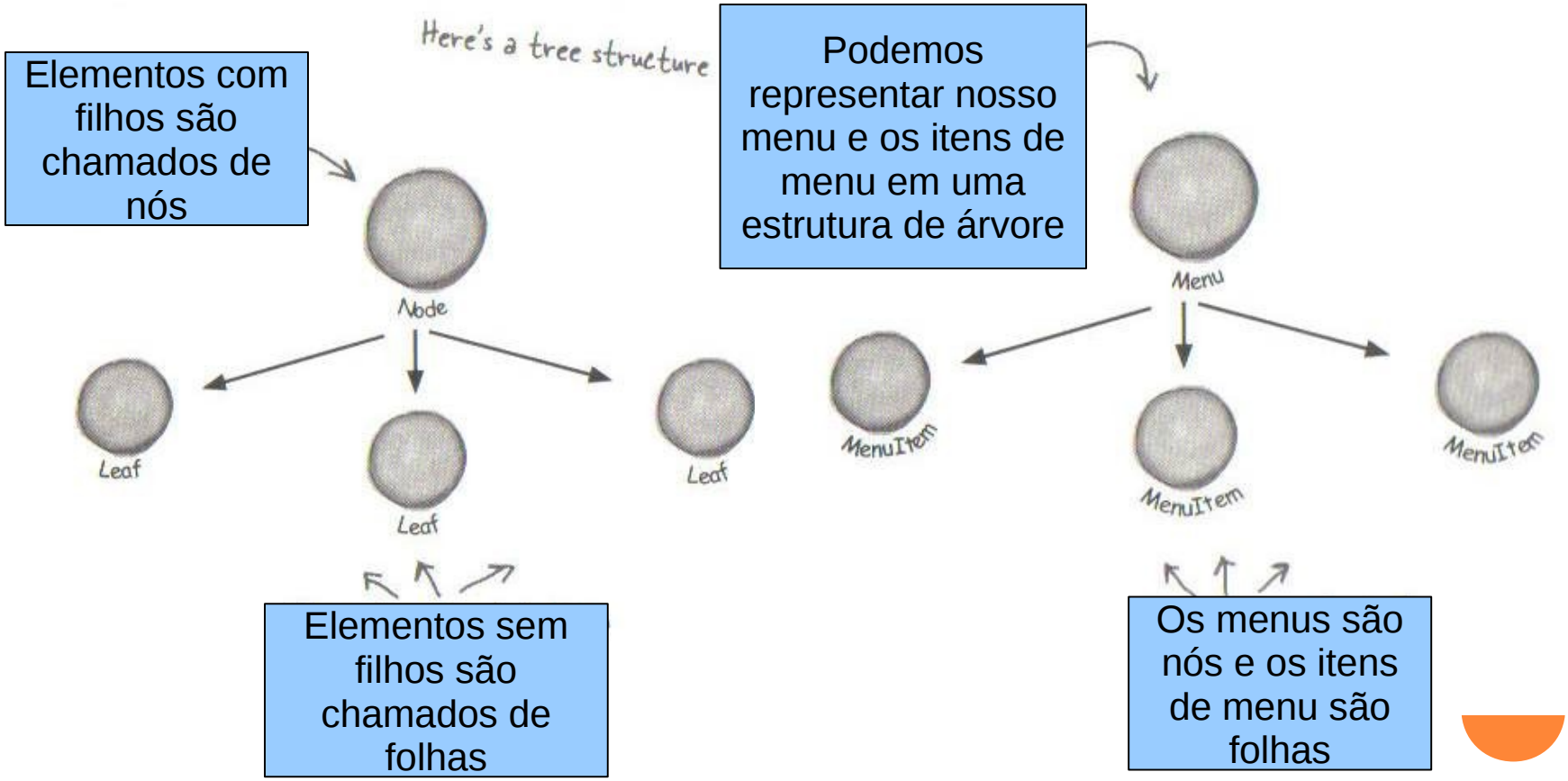
precisamos acessar individualmente todos os elementos da árvore



o processo deve ser mais flexível para que possamos acessar, por exemplo, somente um menu



Using Composite pattern

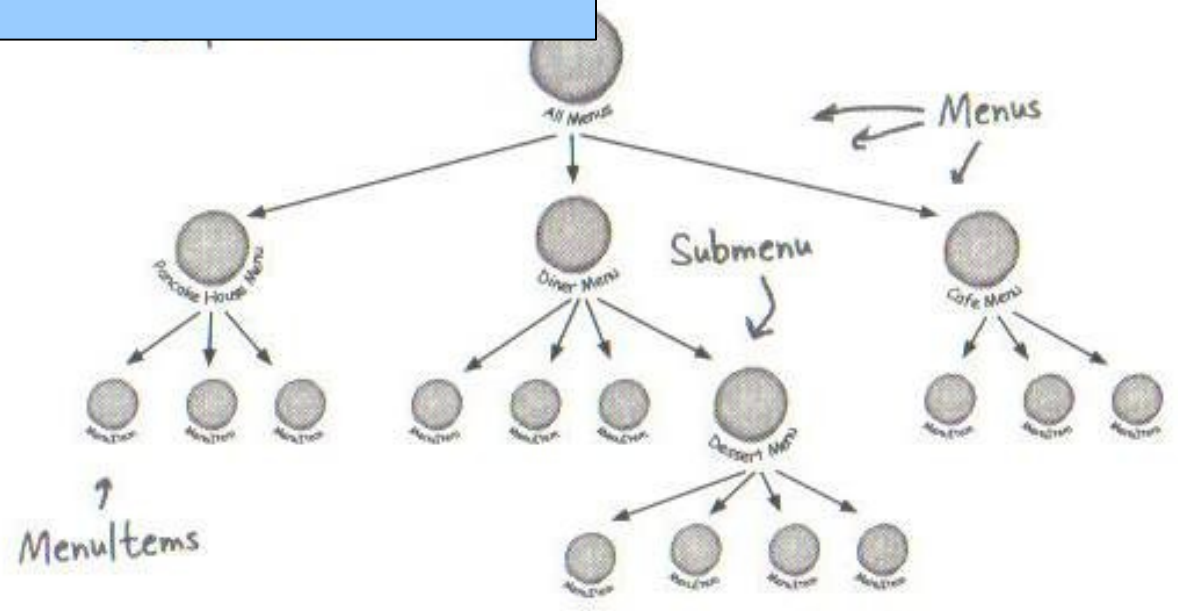


Composite Pattern

- O Padrão Composite permite construir estruturas de objetos na forma de árvores, contendo tanto composições de objetos como objetos individuais
- Usando uma estrutura composta, podemos aplicar as mesmas operações tanto à composição de objetos como a objetos individuais (i.e., podemos ignorar as diferenças entre eles).

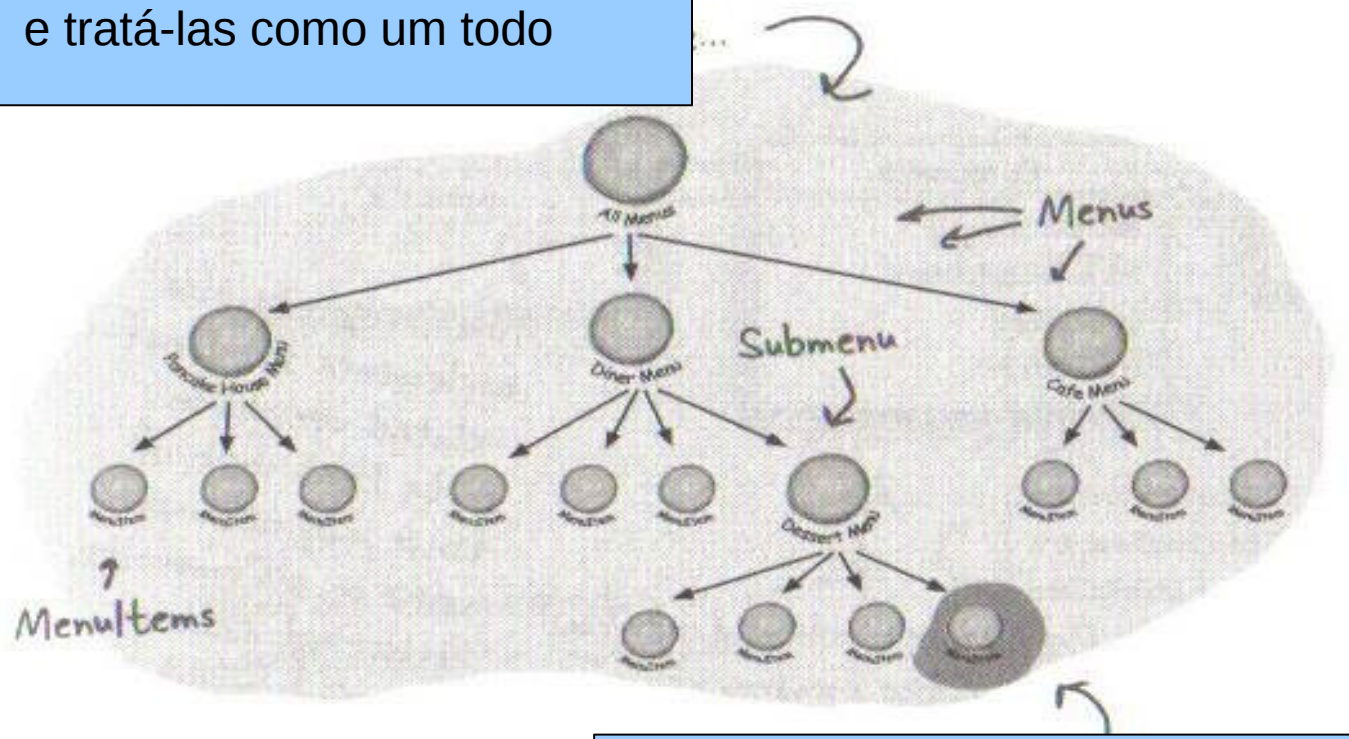
Using Composite pattern

Podemos criar árvores arbitrariamente complexas



Using Composite pattern

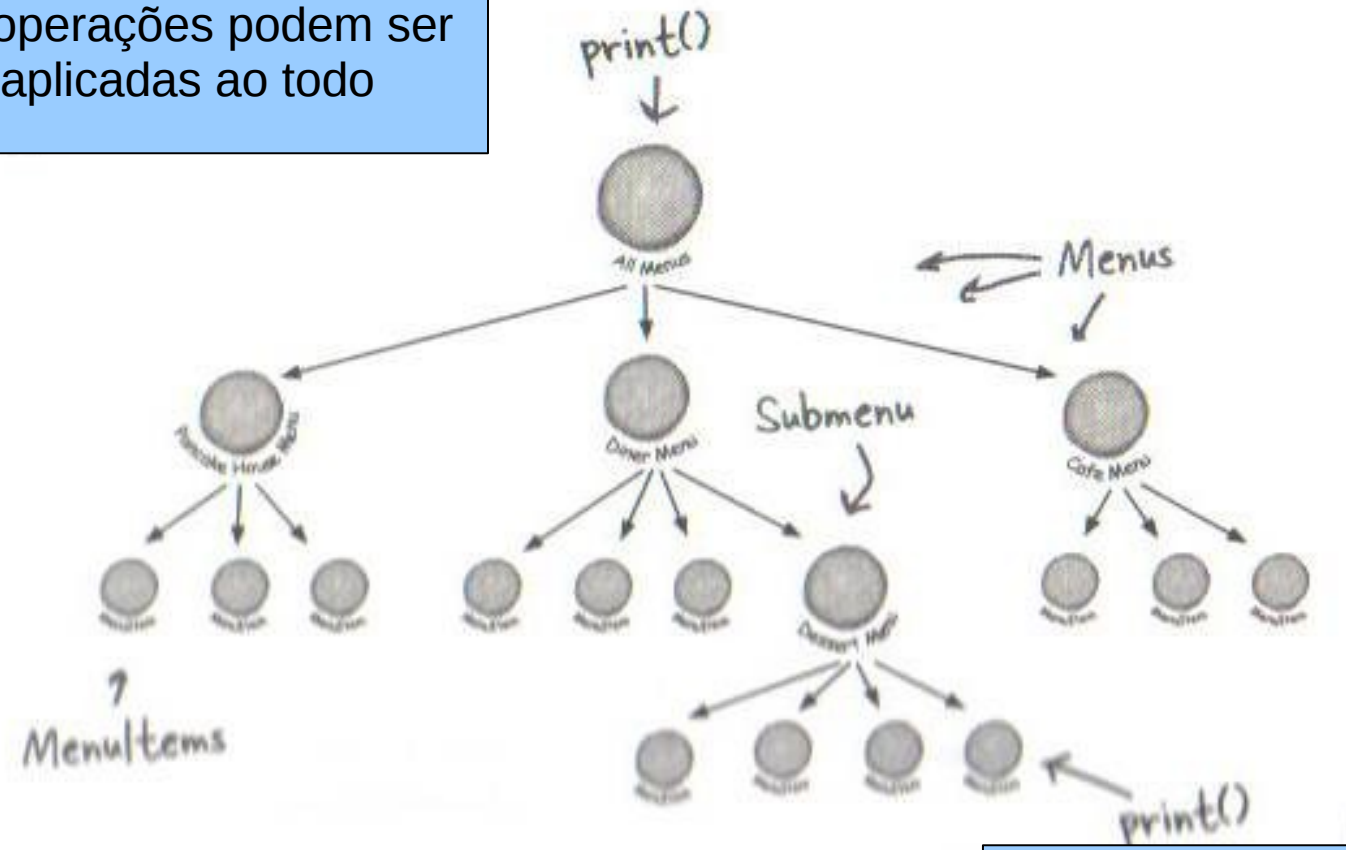
e tratá-las como um todo



ou como partes

Using Composite pattern

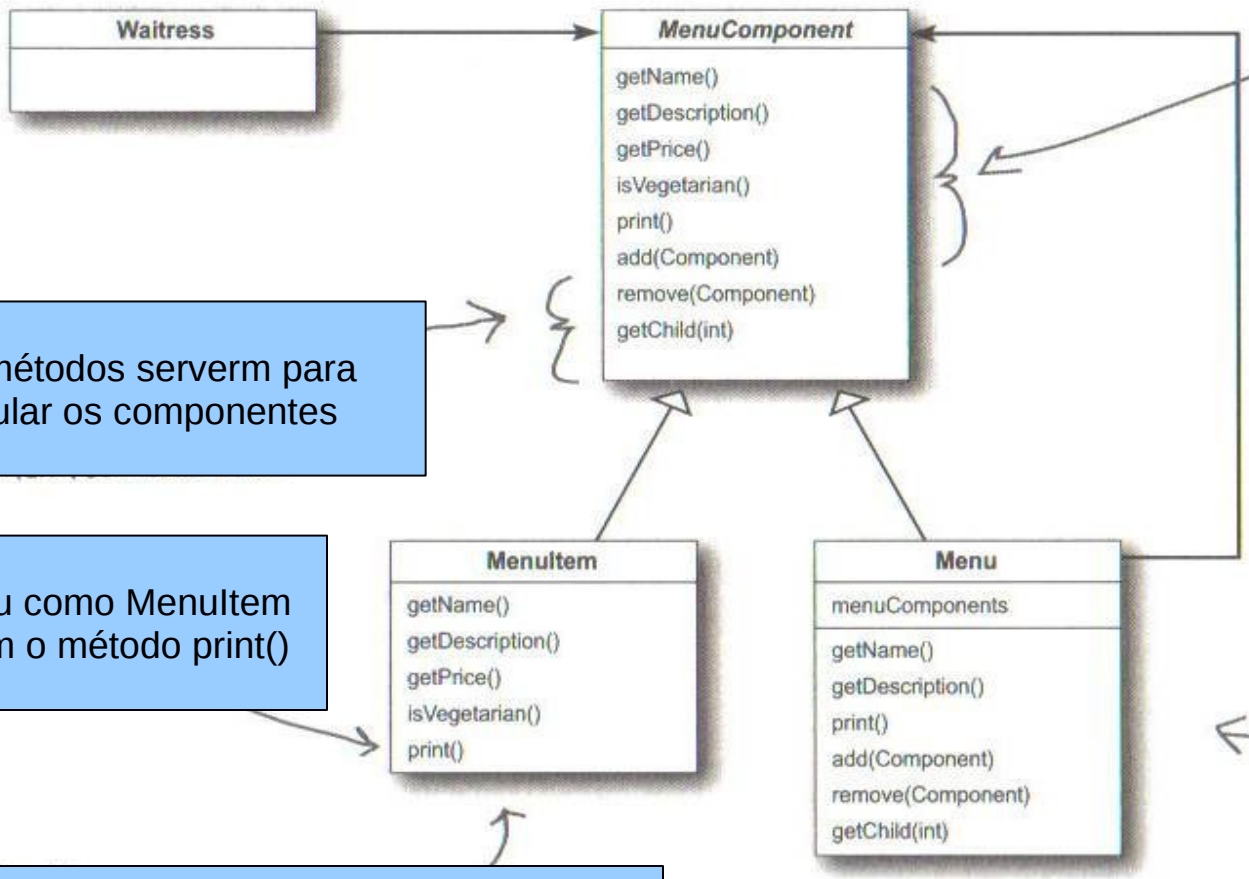
As operações podem ser aplicadas ao todo



Ou às partes

A Garçonete usará a interface para acessar tanto os menus como os itens de menu

Usamos uma classe abstrata para fornecer uma implementação default para estes métodos



Esses métodos serverm para manipular os componentes

Métodos já existentes

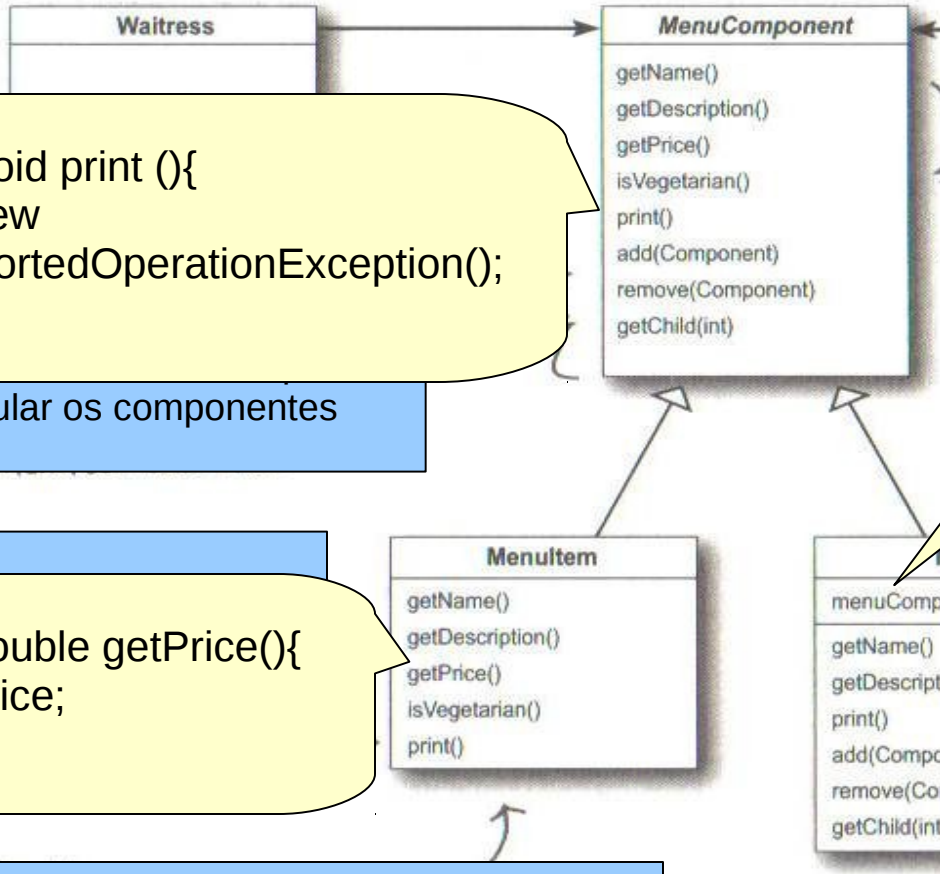
Tanto Menu como MenuItem susbstituem o método print()

Susbstitue os métodos que fazem sentido e usa a implementação padrão para aqueles que não lhe são relevantes, como add(), uma vez que não podemos acrescentar um componente a uma folha

Substitue os métodos que fazem sentido, como aqueles que lhe permitem acrescentar ou remover items de menu (ou outros menus).

A Garçonete usará a interface para acessar tanto os menus como os itens de menu

Usamos uma classe abstrata para fornecer uma implementação default para estes métodos




```
public void print (){
throw new
UnsupportedOperationException();
}
```

manipular os componentes

Poderia ser um ArrayList

métodos já existentes

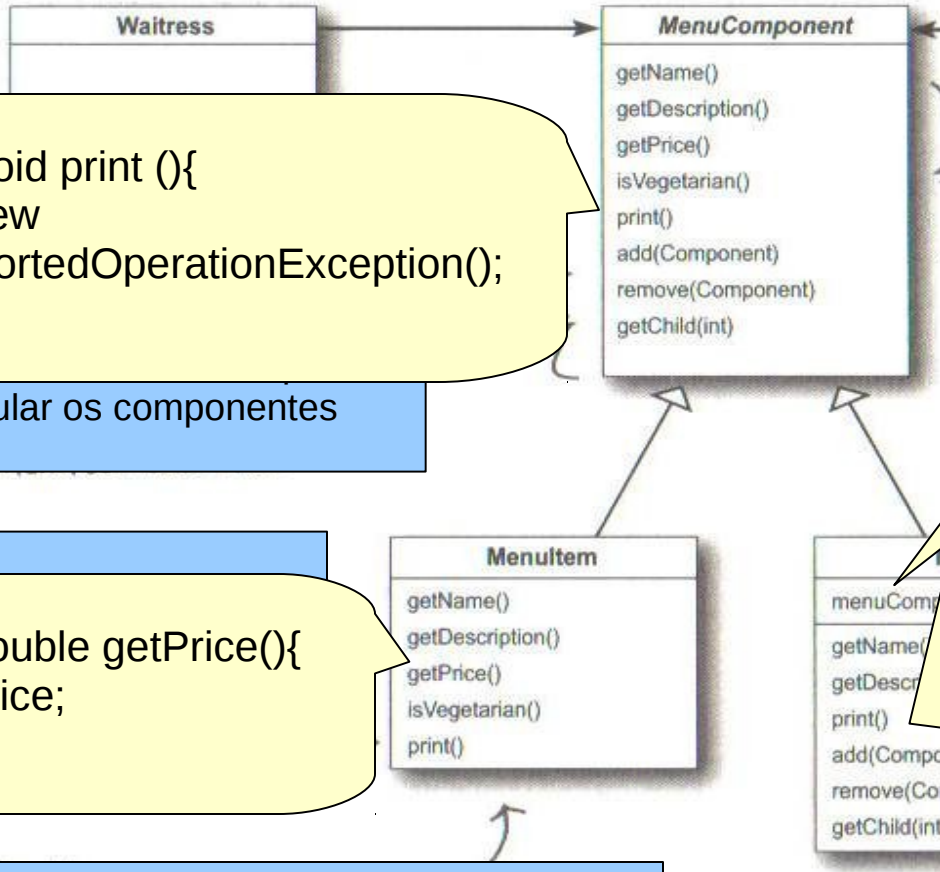
```
public double getPrice(){
return price;
}
```

Susbstitue os métodos que fazem sentido e usa a implementação padrão para aqueles que não lhe são relevantes, como add(), uma vez que não podemos acrescentar um componente a uma folha, nesses casos, uma exceção pode ser gerada.

Substitue os métodos que fazem sentido, como aqueles que lhe permitem acrescentar ou remover items de menu (ou outros menus).

A Garçonete usará a interface para acessar tanto os menus como os itens de menu

Usamos uma classe abstrata para fornecer uma implementação default para estes métodos




```
public void print (){
throw new
UnsupportedOperationException();
}
```

Poderia ser um ArrayList

manipular os componentes

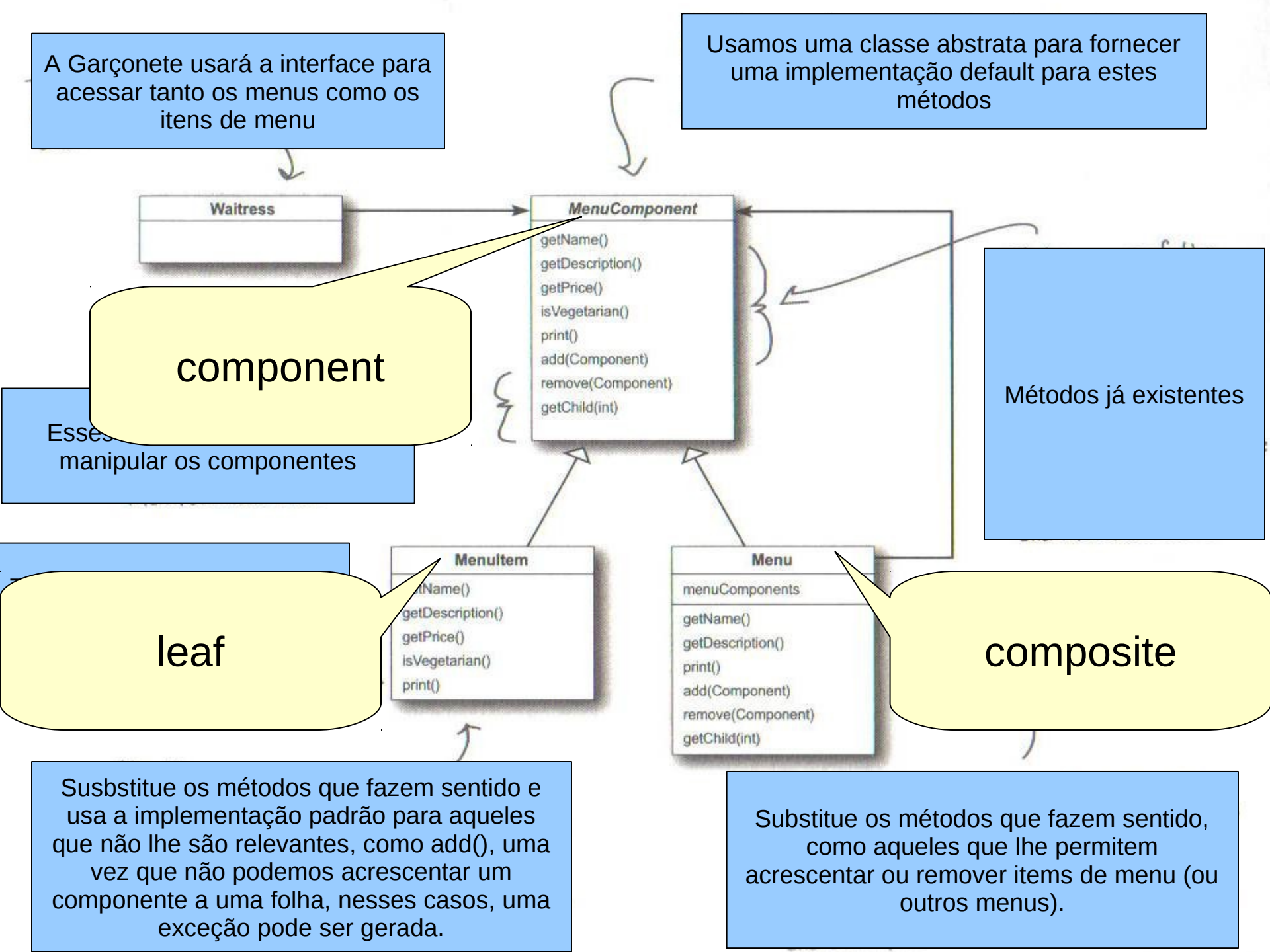
```
public double getPrice(){
return price;
}
```

```
public void print(){
System.out.print( getname());
 ...
Iterator it=
menuComponents.iterator();
while (iterator.hasNext()){
 MenuComponent menuComponent=
(MenuComponent) iterator.next();
 menuComponent.print();
}
}
```

Susbstitue os métodos que fazem sentido e usa a implementação padrão para aqueles que não lhe são relevantes, como add(), uma vez que não podemos acrescentar um componente a uma folha, nesses casos, uma exceção pode ser gerada.

A Garçonete usará a interface para acessar tanto os menus como os itens de menu

Usamos uma classe abstrata para fornecer uma implementação default para estes métodos



component

Esses
manipular os componentes

Métodos já existentes

leaf

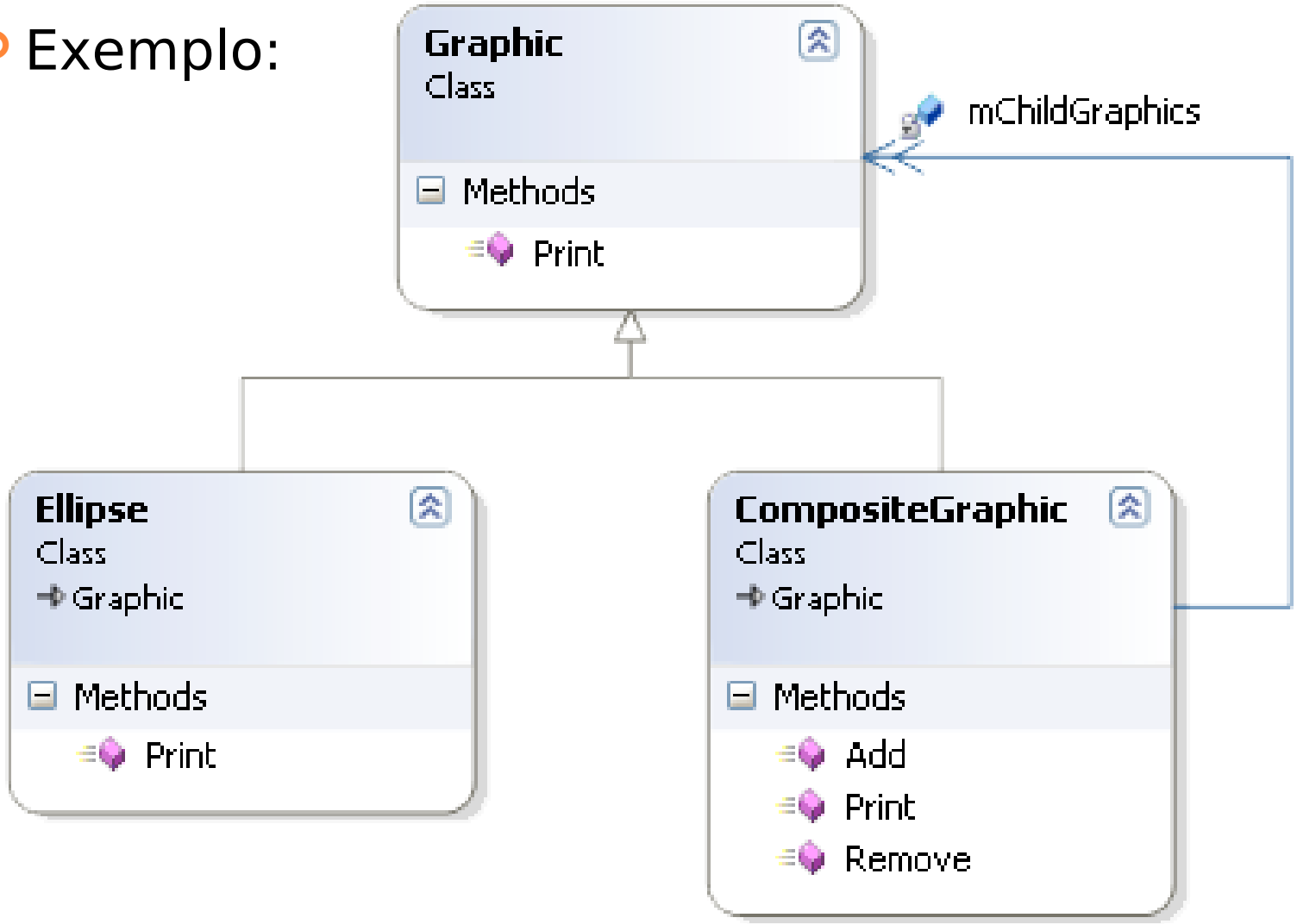
composite

Susbstitue os métodos que fazem sentido e usa a implementação padrão para aqueles que não lhe são relevantes, como add(), uma vez que não podemos acrescentar um componente a uma folha, nesses casos, uma exceção pode ser gerada.

Substitue os métodos que fazem sentido, como aqueles que lhe permitem acrescentar ou remover items de menu (ou outros menus).

PADRÃO COMPOSITE

- Exemplo:

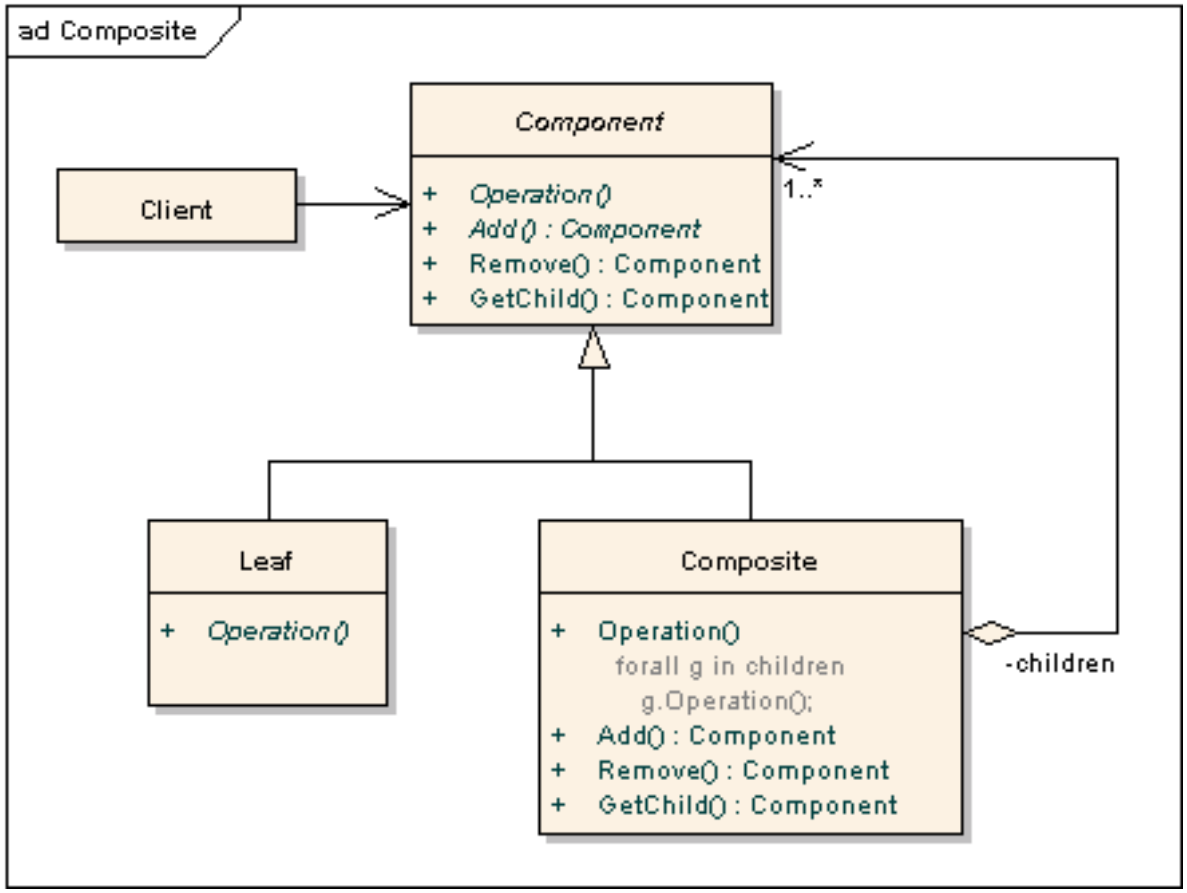


PADRÃO COMPOSITE

- Aplicabilidade
  - Sempre que houver necessidade de tratar um conjunto como um indivíduo.
  - Funciona melhor se relacionamentos entre os objetos for no formato de uma árvore.
  - Caso o relacionamento contenha ciclos, é preciso tomar precauções adicionais para evitar loops infinitos, já que Composite depende de implementações recursivas

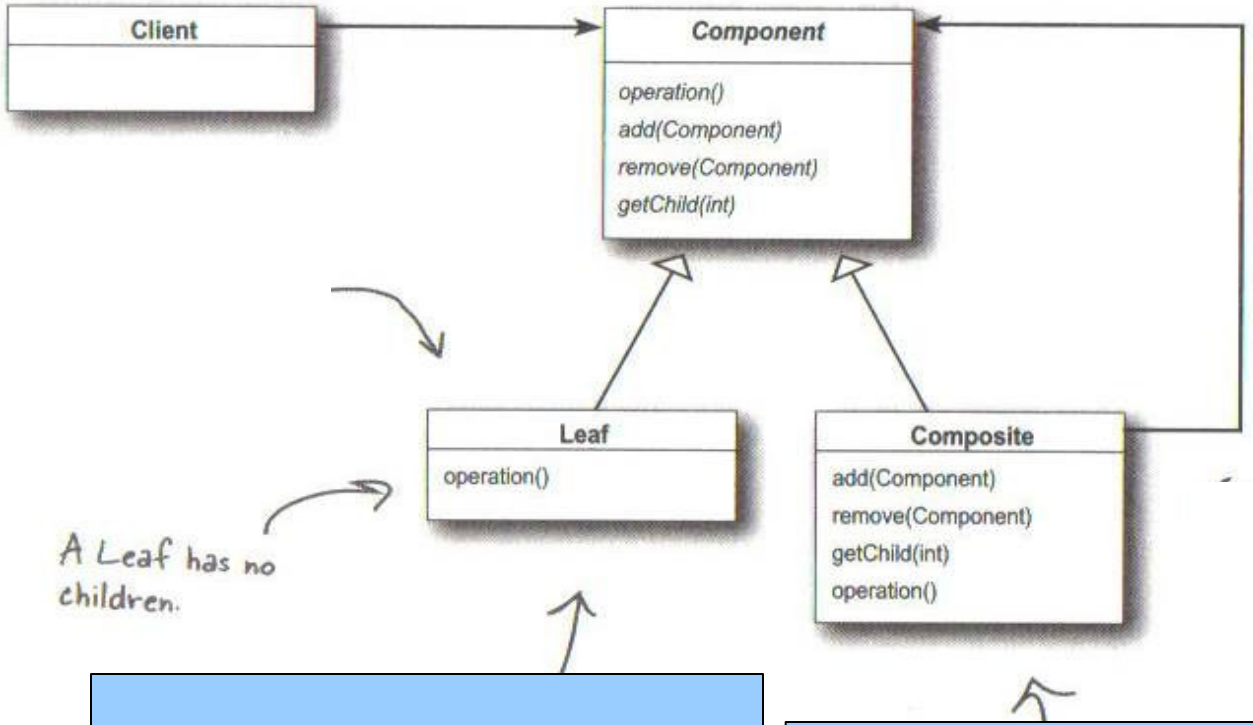
PADRÃO COMPOSITE

- Estrutura:



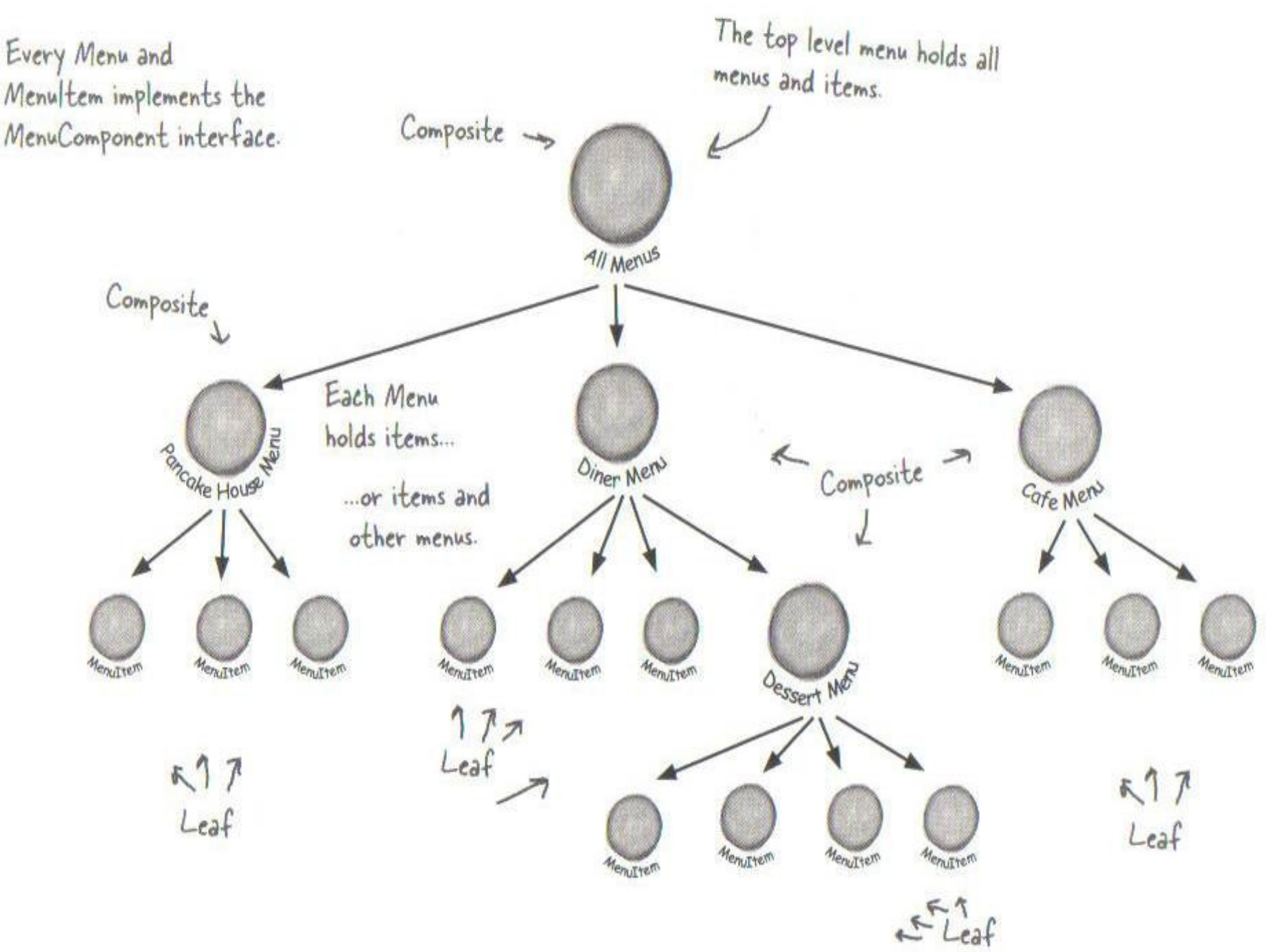
O Cliente usa a interface Component para manipular os objetos da composição

Component define uma interface para todos os objetos da composição, o que inclui tanto os compostos (Composite) como os nós folhas (Leaf).



Leaf define o comportamento dos elementos da composição.

Composite define o comportamento dos componentes que possuem filhos


Every Menu and MenuItem implements the MenuComponent interface.
The top level menu holds all menus and items.
Composite
All Menus
Composite
Pancake House Menu
Each Menu holds items...
...or items and other menus.
Diner Menu
Composite
Cafe Menu
MenuItem
Dessert Menu
Leaf
Leaf
Leaf
Leaf

PADRÃO COMPOSITE

- Participantes:
  - Component
    - Declara a interface para os objetos na composição.
    - Implementa, se necessário, o comportamento padrão.
    - Declara uma interface para acessar e gerenciar os seus componentes filhos.
  - Leaf
    - Representa objetos folha na composição (sem filhos).
    - Define comportamento para esses objetos.
  - Composite
    - Define comportamento para os componentes que têm filhos.
    - Armazena referências a os componentes filho.
  - Client
    - Manipula os objetos na hierarquia utilizando instâncias Component.

PADRÃO COMPOSITE

- Consequências
  - Define hierarquias de classe que consistem de objetos primitivos e objetos compostos.
  - O cliente consegue tratar da mesma forma os componentes folha e os componentes compostos.
  - Facilita a manipulação de novos tipos de componentes (novas folhas ou novos compostos).
- Implementação
  - Manter referências entre objetos pai e filho.
  - A classe Component deve definir todos os métodos permitidos pelos nós folha e os nós compostos, mesmo que alguns só façam sentido em um ou outro.

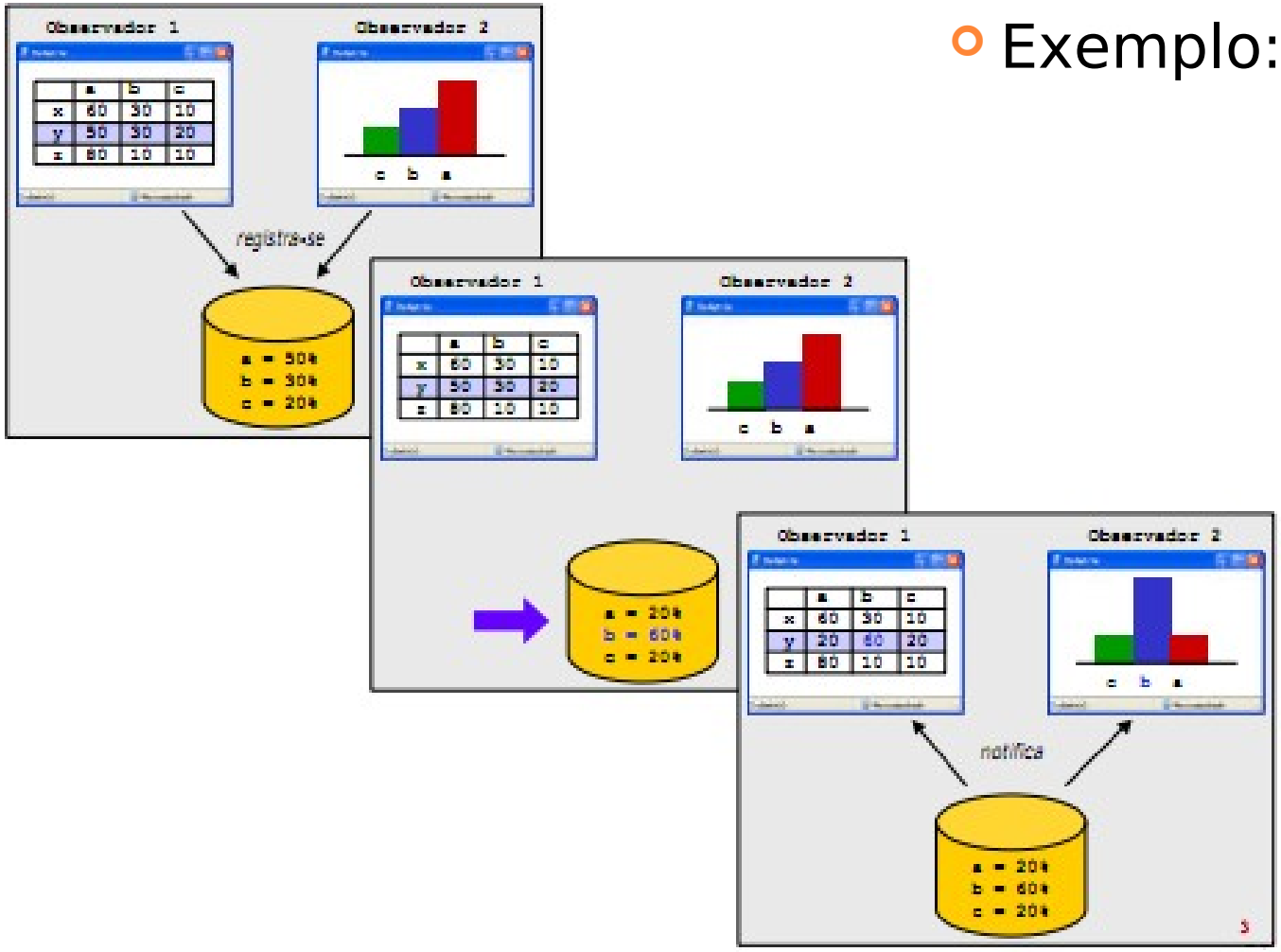
PADRÃO OBSERVER

PADRÃO OBSERVER

- Padrão Comportamental.
- Objetivo:
  - Definir uma dependência um – para – muitos entre objetos, para que quando um objeto mudar de estado, todos os seus dependentes sejam notificados e atualizados automaticamente.
- Motivação:
  - Quando temos um objeto (ou conjunto de objetos) cujo estado ou comportamento depende do estado de outro objeto.
  - O objeto “observador” precisa ser informado das mudanças no objeto “observado”.
  - O objeto “observador” também pode alterar o “observado”.

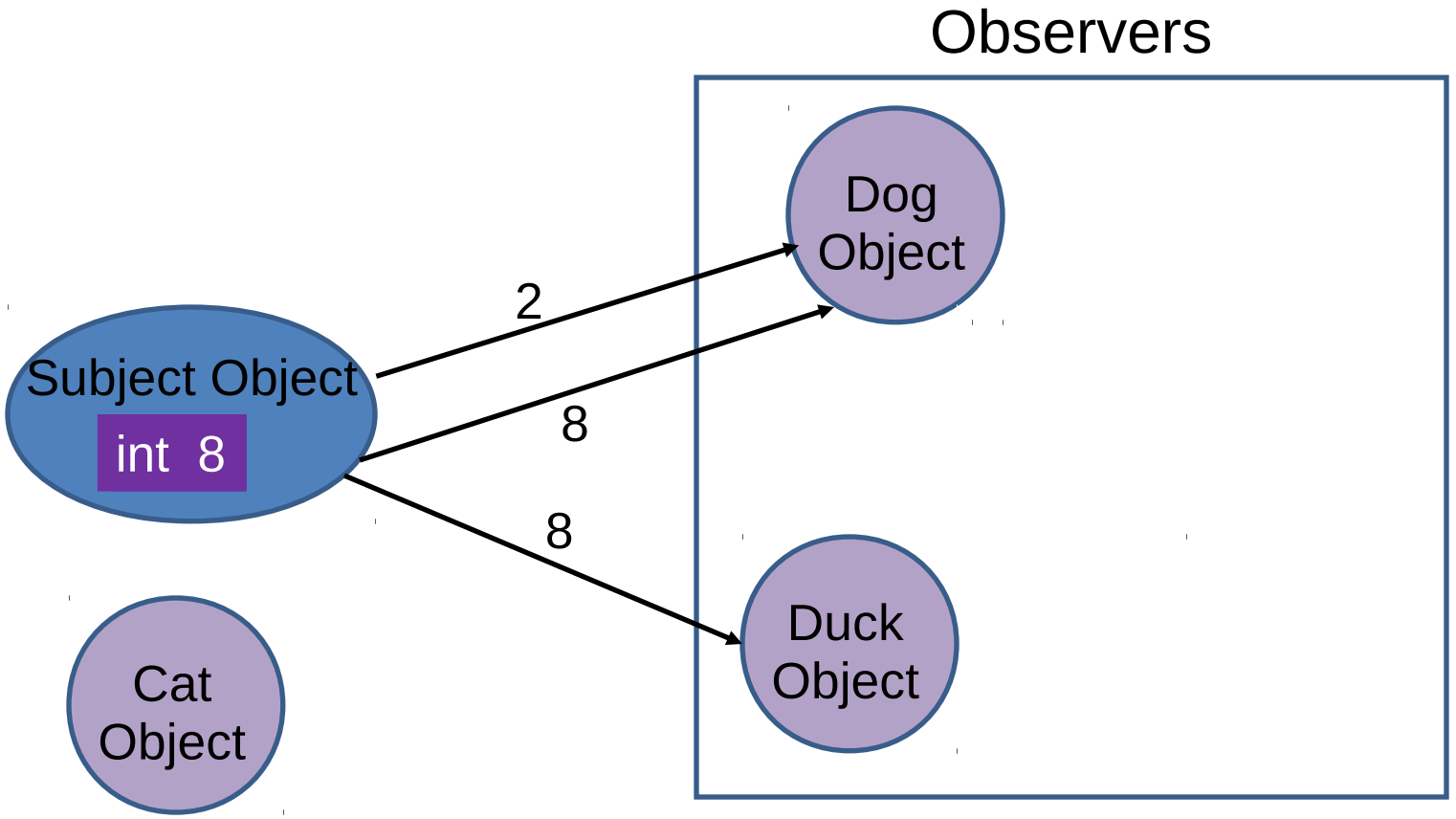
PADRÃO OBSERVER

- Exemplo:

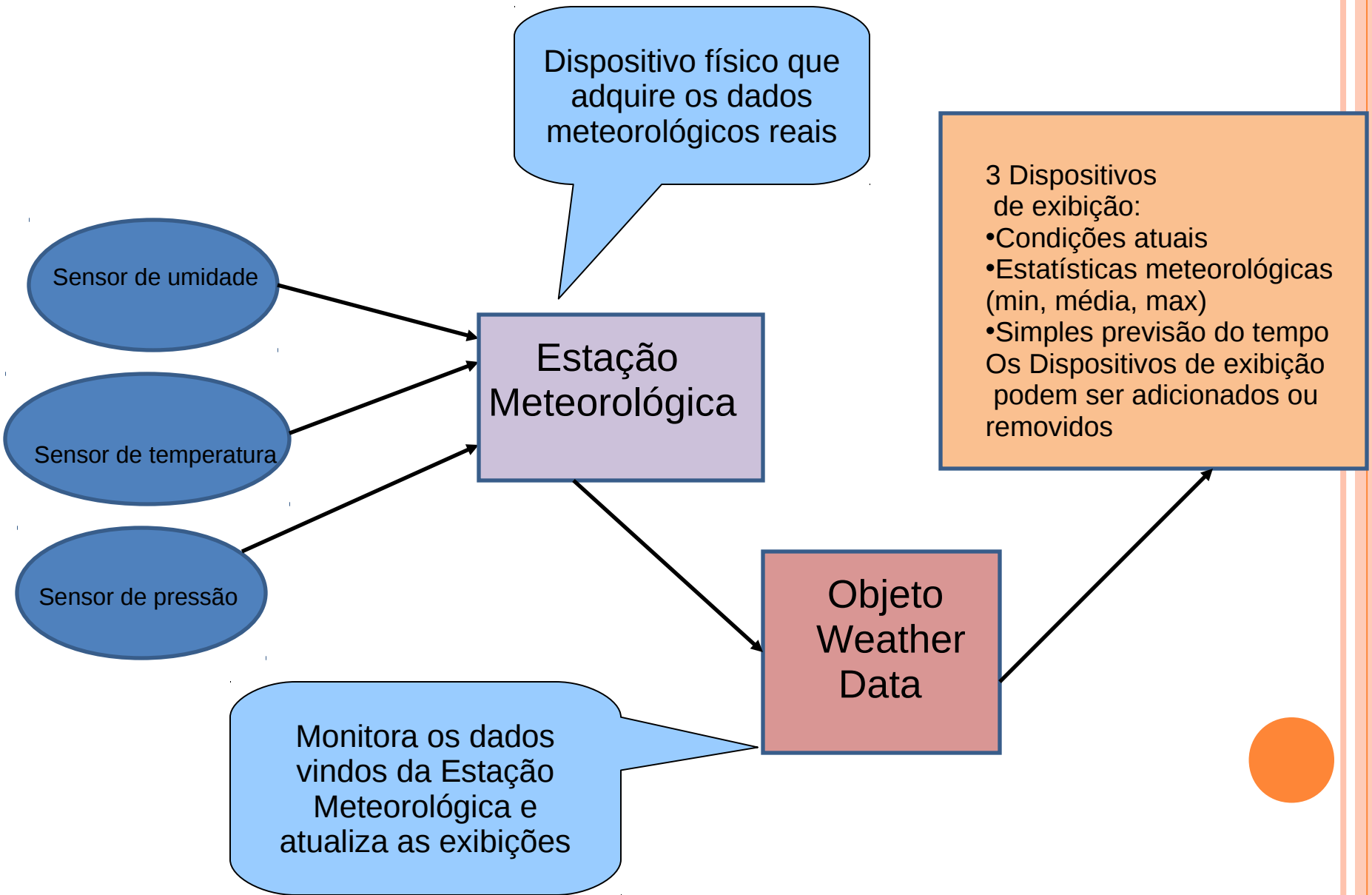


Padrão Observer

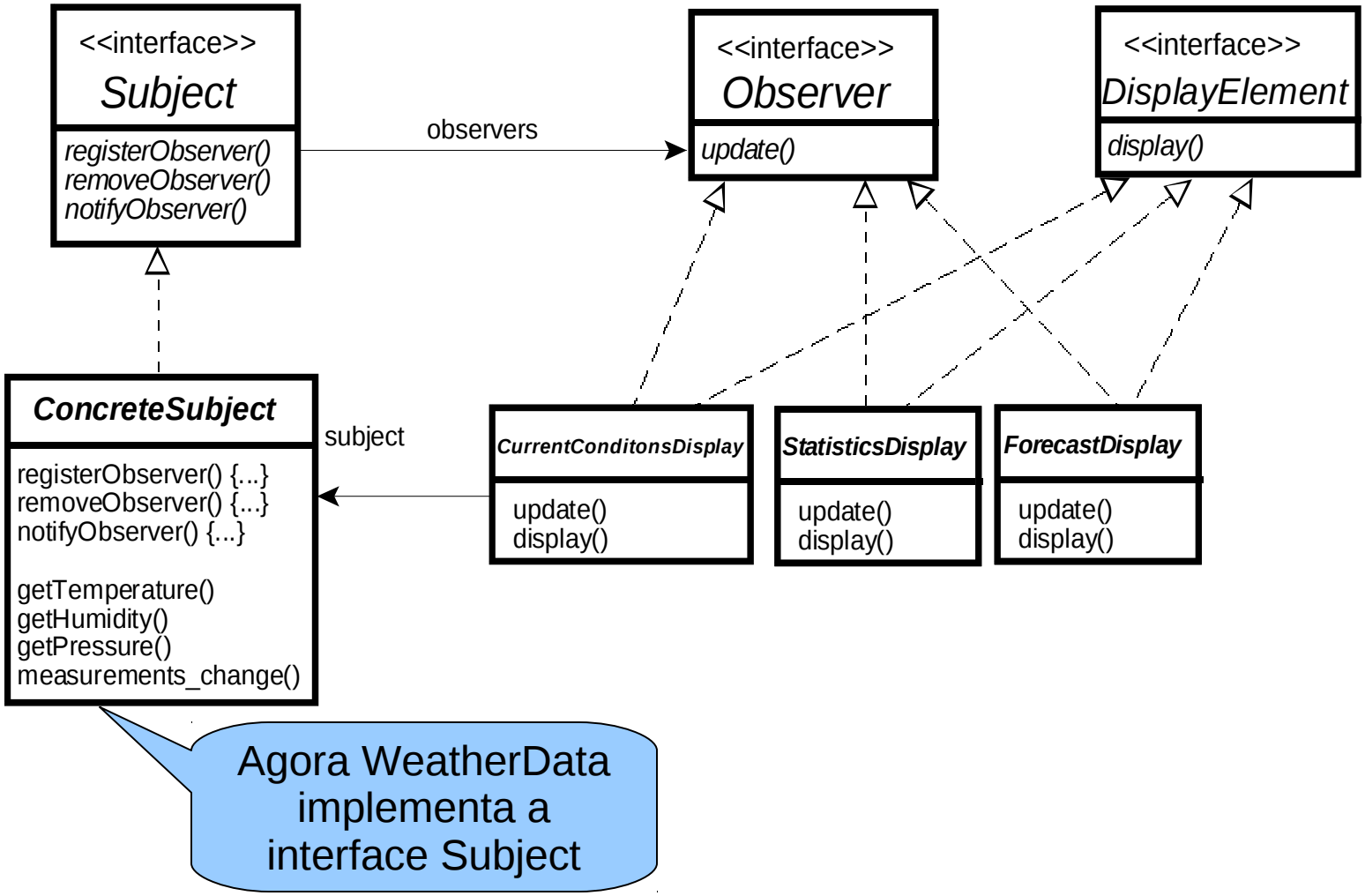
Subject Broadcasts ers



Estação Meteorológica



Estação Meteorológica



Estação Meteorológica

```
public interface Subject{
   public void registerObserver (Observer o);
   public void removeObserver (Observer o);
   public void notifyObservers();
}

public interface Observer{
   public void update(float temp, float humidity,
  float pressure);
}

public interface DisplayElement{
   public void display();
}
```

Estação Meteorológica

```
public class ConcreteSubject implements Subject{
   private ArrayList <Observer> observers;
   private float temperature;
   private float humidity;
   private float pressure;
  public ConcreteSubject(){
          observers =new ArrayList<Observer>();
  }
  public void registerObserver(Observer o){
      observer.add(o);
  }
   public void removeObserver(Observer o){...}
   public void notifyObservers(){
      for (int i=0; i<observers.size(); i++){
          Observer observer=observers.get(i);
          observer.update(temperatura,humidity,pressure);
      }
   }
   public void measurementsChanged(){
      notifyObservers();
   }
  public void setMeasurements(float temperatura, float humidity, float
   pressure){
      this.temperature=temperature;
      this.humidity=humidity;
      this.pressure=pressure;
      measurementsChanged();
  }
}
```

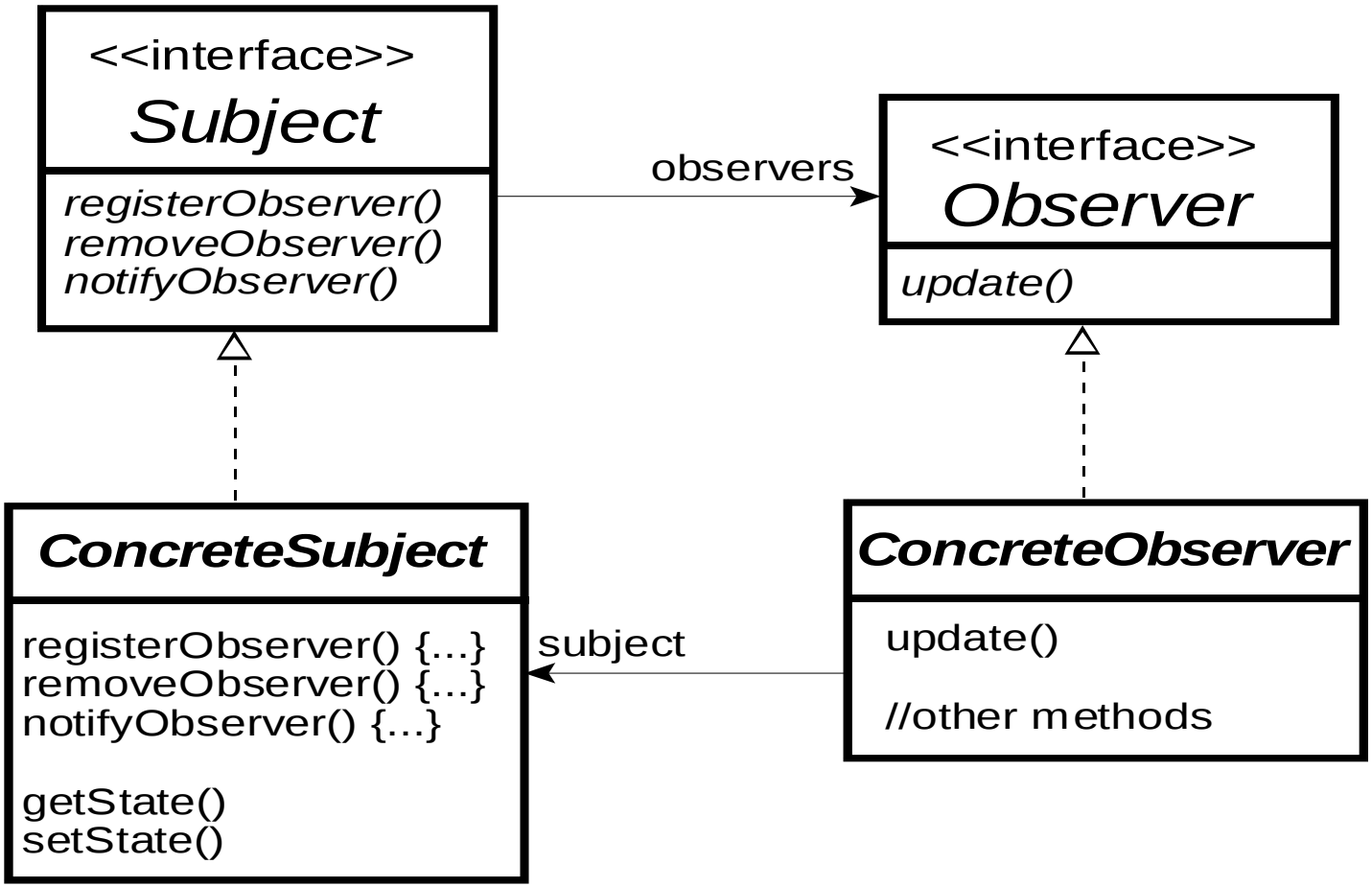
Estação Meteorológica

```
public class CurrentConditionsDisplay implements Observer, Display
   Element{
   private float temperature;
   private float humidity;
   private float pressure;
    private Subject concreteSubject;

   public CurrentConditionsDisplay(Subject concreteSubject){
       this.concreteSubject=concreteSubject;
        concreteSubject.registerObserver(this);
  }
  public void update(float temperatura, float humidity, float pressure)
   {
       this.temperature=temperature;
       this.humidity=humidity;
       this.pressure=pressure;
       display();
   }
  public void display(){
     System.out.println("Current conditions: "+ ...)
  }
```

PADRÃO OBSERVER

- Estrutura



PADRÃO OBSERVER

- Participantes
  - Subject:
    - Pode possuir uma lista dos seus observadores, mas não conhece o tipo ou o objetivo de cada um deles.
    - Possui interface para adicionar e remover “assinantes” ou observadores.
  - ConcreteSubject:
    - Notifica seus observadores quando o seu estado muda.
  - Observer:
    - Define uma interface para ser notificado de mudanças.
  - ConcreteObserver:
    - Mantém uma referência ao objeto observado.
    - Guarda estado do observado para identificar as mudanças.
    - Implementa a resposta às mudanças do observado.

PADRÃO OBSERVER

- Consequências
  - O observado tem uma lista de observadores.
  - O observador precisa conhecer o estado do observado.
  - As notificações podem ser vistas como envios de mensagens:
    - Mensagens podem ter diferentes importâncias para cada observador.
    - Algumas podem ser spam (observadores podem obviar mensagens).
    - O observado poderia decidir a quem enviar cada tipo de mensagem: categorizar os observadores.
- Implementação
  - Evitar envio de mensagens desnecessárias.
  - Notificar mudanças em bloco.

O poder da ligação leve e o PADRÃO OBSERVER

- Projetos levemente ligados permitem:
  - construir sistemas OO flexíveis que podem lidar com mudanças
- O Padrão Observer fornece um design de objeto onde os sujeitos observados e os observadores são levemente ligados porque:
  - A única cosa que o sujeito sabe sobre um observador é que ele implementa uma certa interface
  - Podemos adicionar observadores a qualquer momento
  - Não precisamos modificar o sujeito para adicionar novos observadores

Princípio de projeto:
Busque designs levemente ligados entre objetos que interagem

PADRÕES GOF VISTOS

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