Padrões de Projeto

Prof. Marum Simão Filho

Agenda

- Princípios de Projeto
- Padrão Strategy
- Padrão Observer
- Padrão Composite

Padrão

Strategy

Contexto

The SimsDuck

Grande variedade de espécies

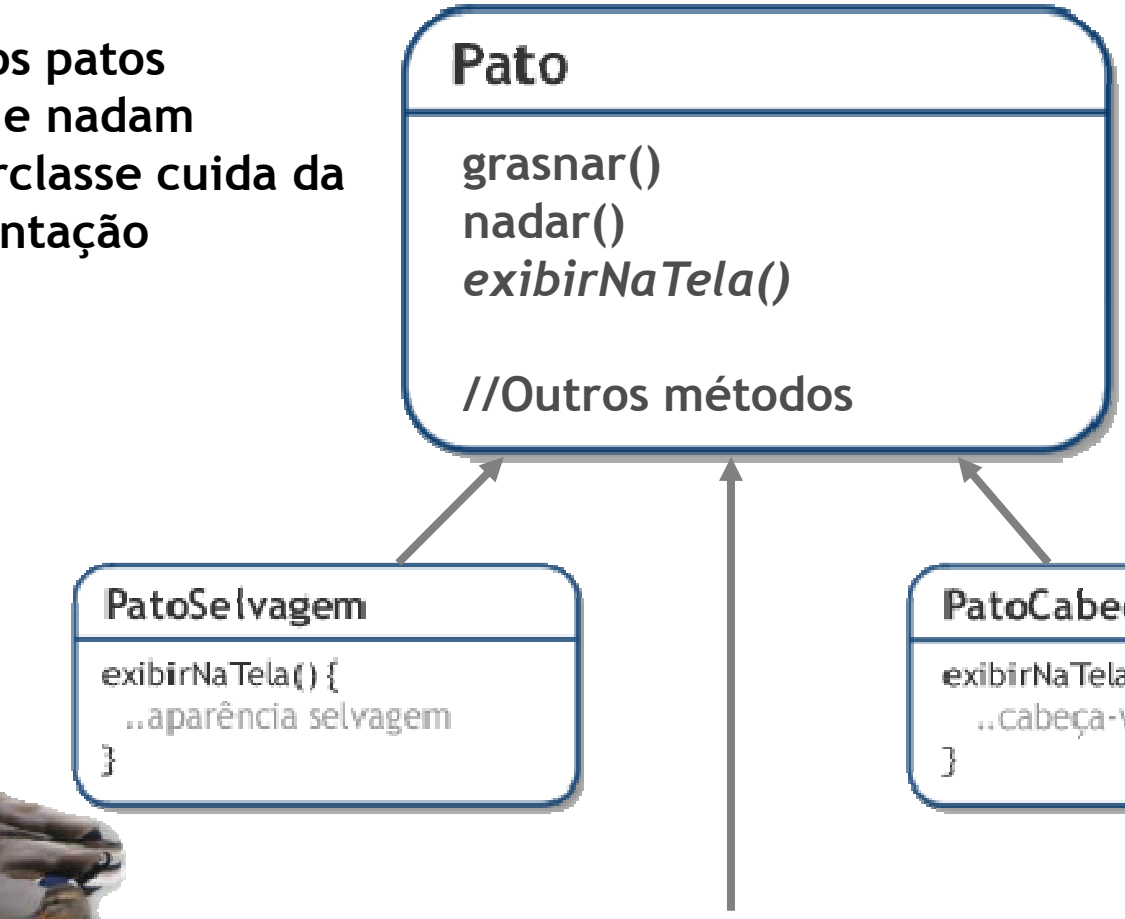
Técnicas OO

Superclasse Pato

Herdada por todos os outros tipos de pato

Começando com Padrões

- Todos os patos grasnam e nadam
- A superclasse cuida da implementação

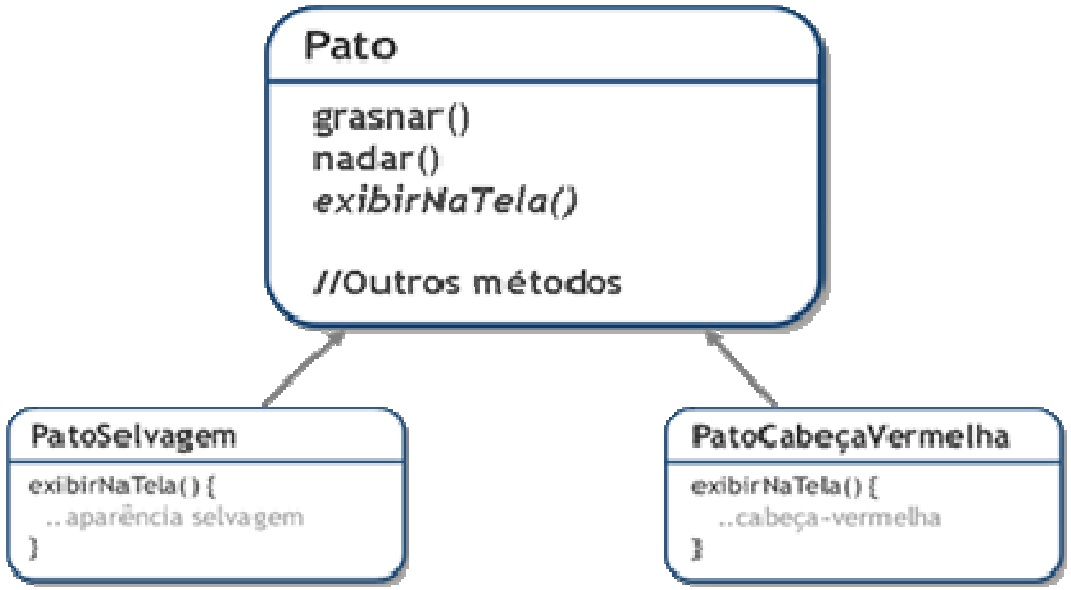


- O método exibirNaTela() é abstrato já que todos os subtipos de pato são diferentes

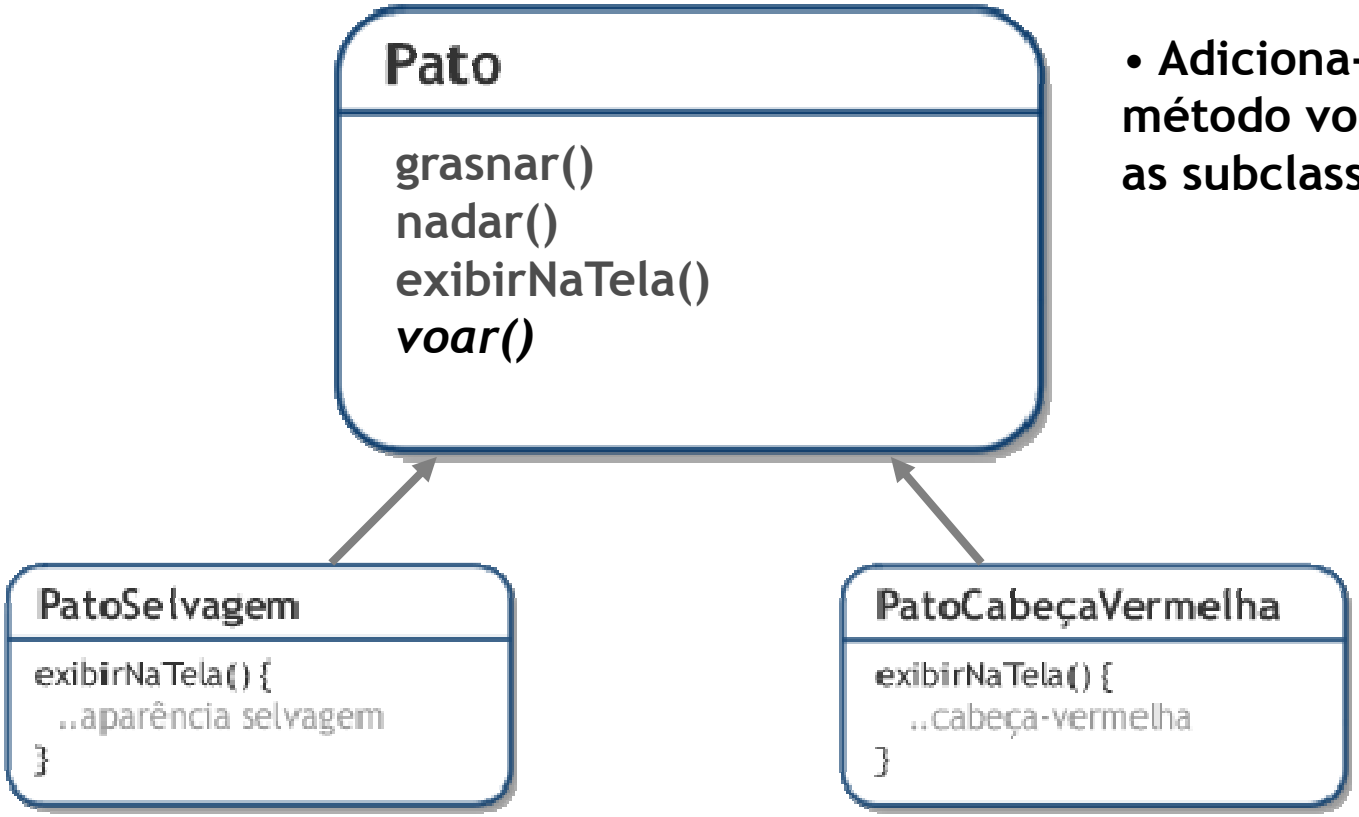
Requisitos

- Alteração nos Requisitos

- ADICIONAR PATOS QUE PRECISAM VOAR

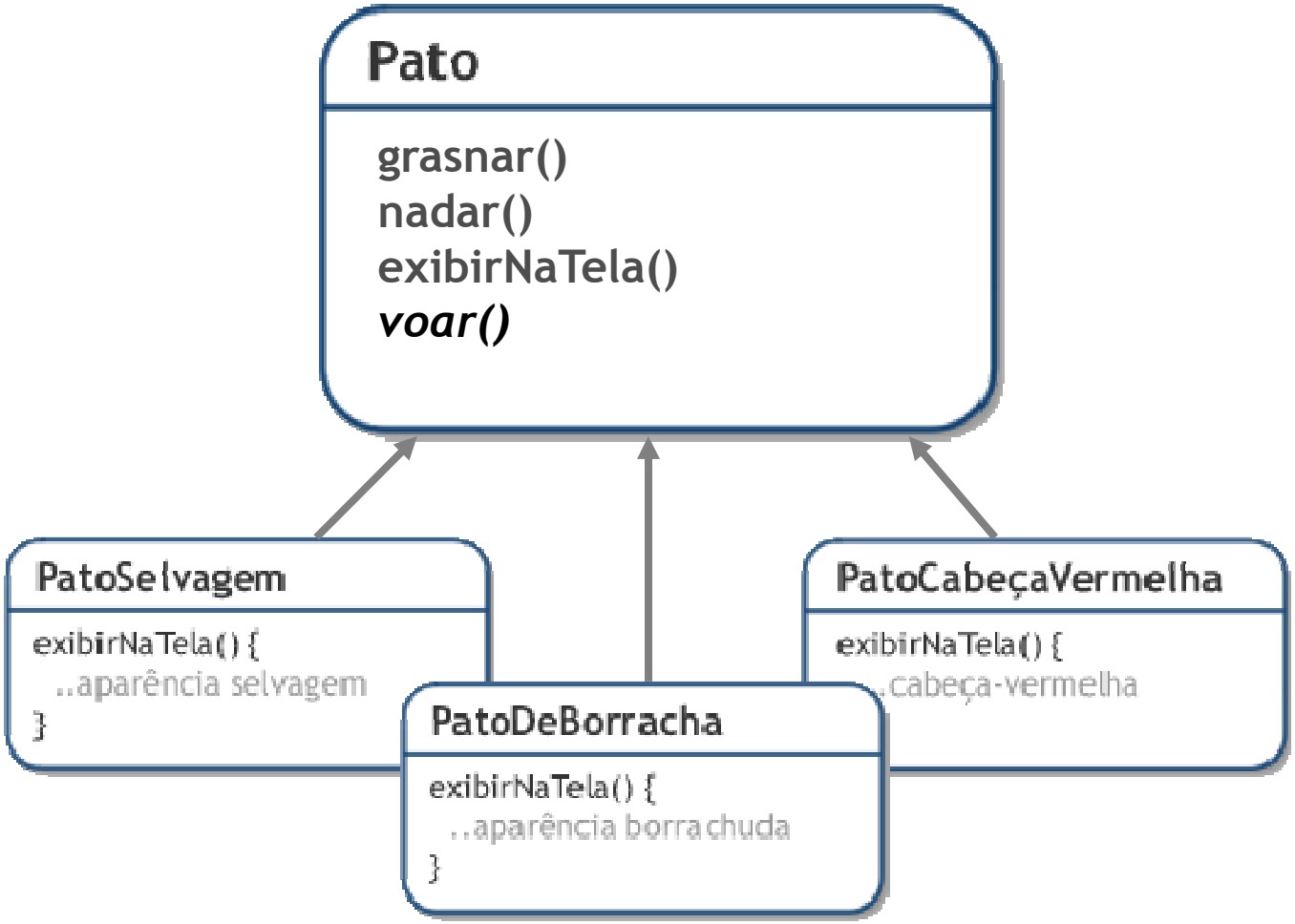


Começando com Padrões



- Adiciona-se o método voar e todas as subclasses herdam

Começando com Padrões



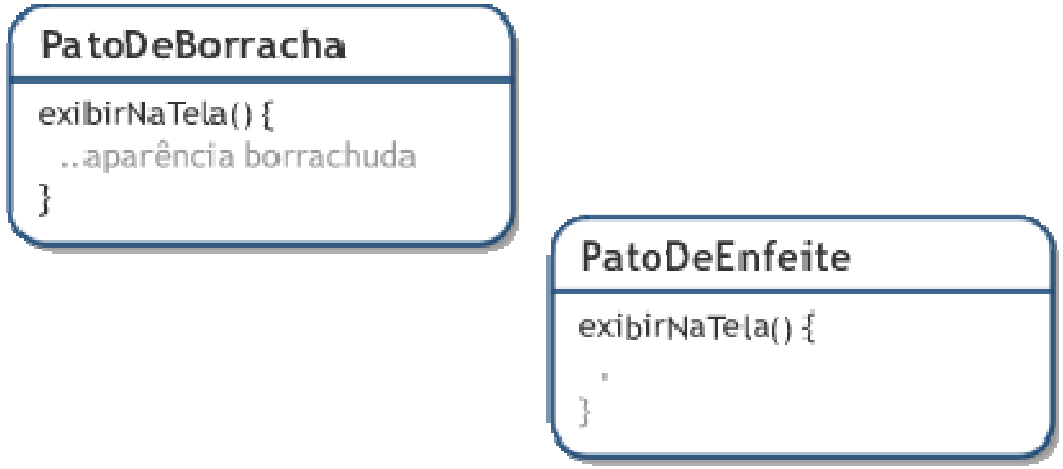
Problemas

- O que pode parecer ser um excelente uso para a herança para fins de reutilização pode não ser tão eficiente quando se trata de manutenção

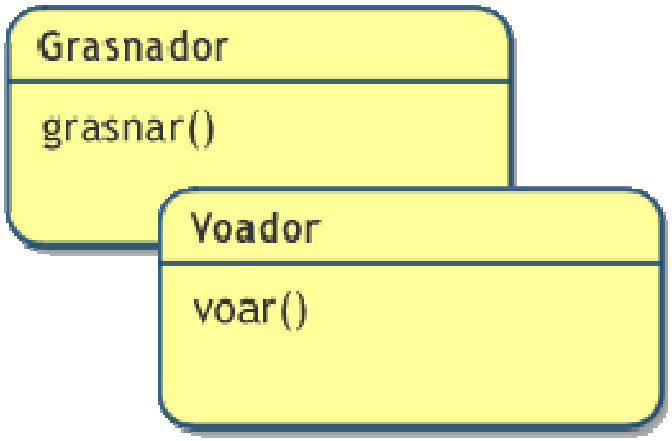
- Problema
  - Nem todas as classes deveriam voar
  - Uma atualização localizada no código causou um efeito colateral não local
    - Patos de borracha voadores

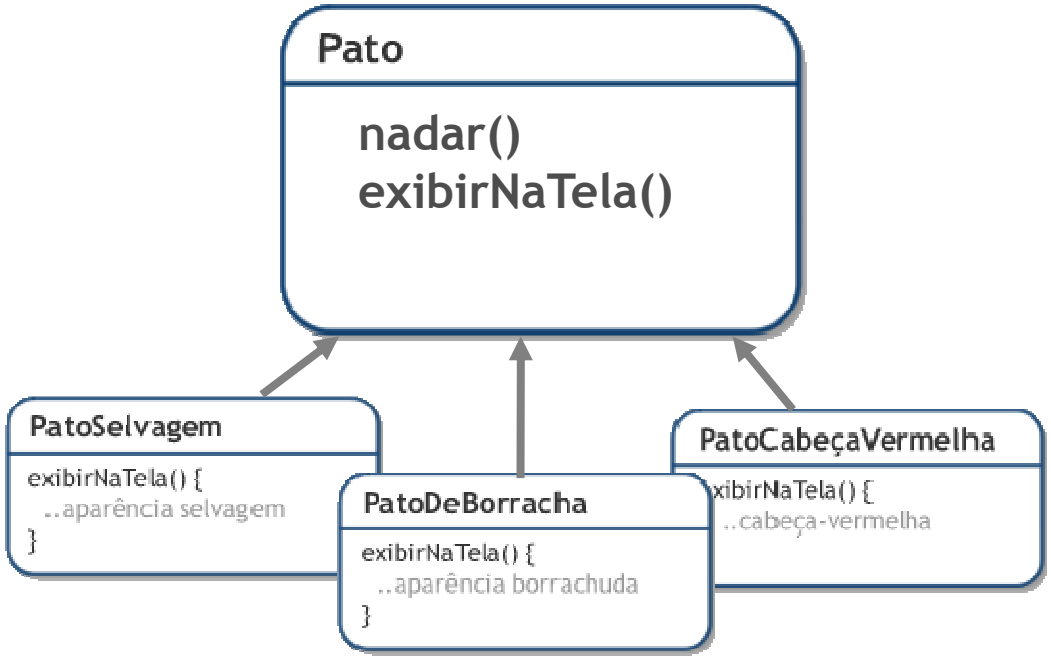
Opções

- Sobrescrever o método voar() do pato de borracha



- Que tal uma interface?????



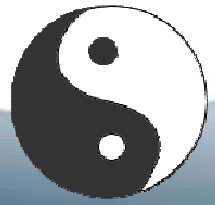


A Constante no desenvolvimento de software

A L T E R A Ç Ã O

HERANÇA

- A HERANÇA não funcionou muito bem
  - Comportamento de patos ficam sempre alterando
  - Nem todas as subclasses necessitam ter o mesmo comportamento
  - Interfaces parecem "OK"
    - Não possuem implementação
      - Não há reutilização de código
    - Sempre que modificar o comportamento
      - Monitoração
      - Alteração



Princípio de Design

- "Identifique os aspectos de seu aplicativo que **variam** e **separe-os** do que permanece igual"
  - Pegue o que variar e "encapsule" para que isso não afete o restante do código
  - Menos consequências indesejadas
  - Mais flexibilidade

Mesma coisa

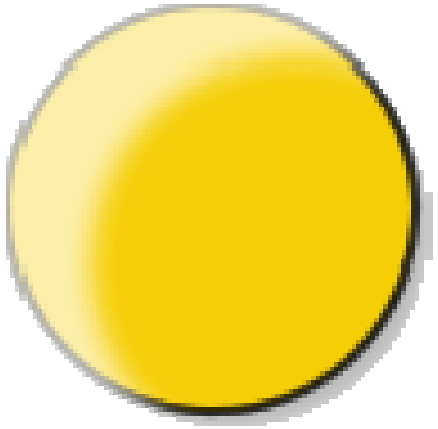
- "Pegue as partes que variam e encapsule-as para depois poder alterar ou estender as partes que variam sem afetar as que não variam"
- Base de quase todos os padrões de projeto
- Hora de voltar aos patos

Aos patos

- O que está variando?
  - voar()
  - grasnar()

  - Criaremos dois conjuntos de classes
  - Totalmente separados
  - 1º → Voar
  - 2º → Grasnar
  - Cada conjunto contém todas as implementações possíveis de comportamento

Retiram-se os métodos da classe e criam-se classes para estes comportamentos



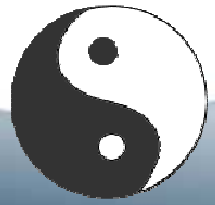
Classe Pato

Comportamentos de vôo

Comportamentos de grasnar

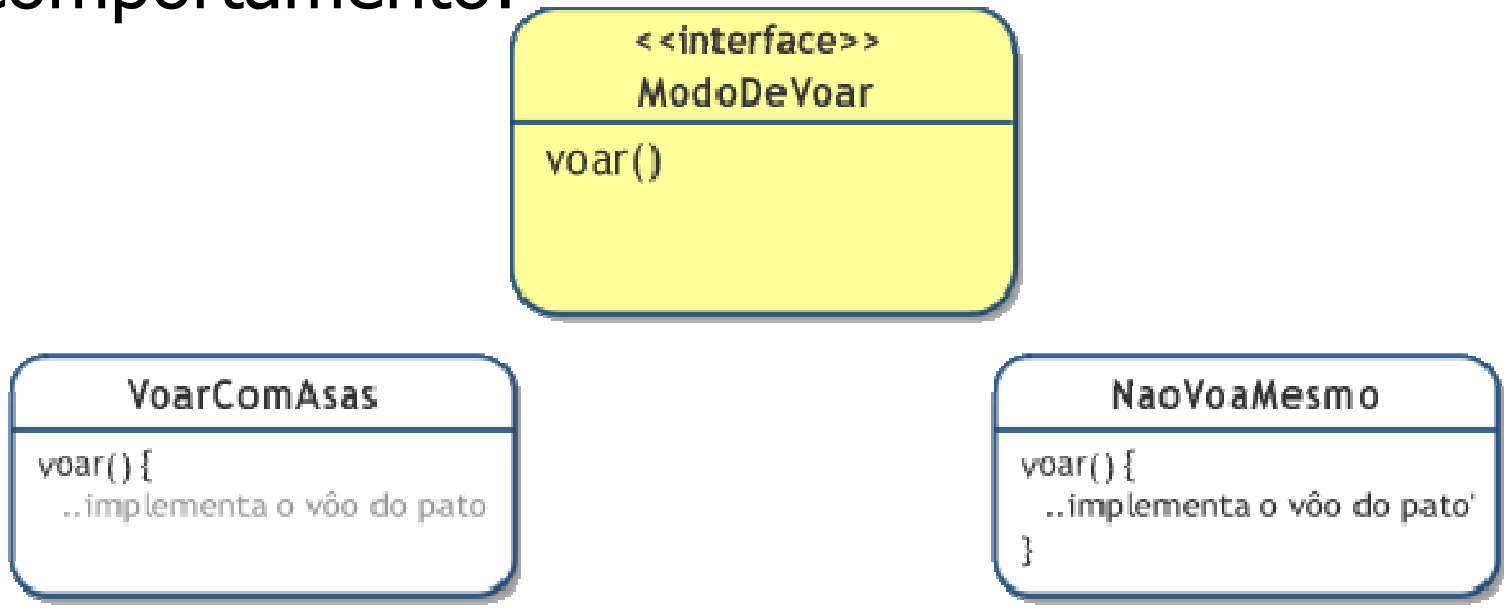
Flexibilidade

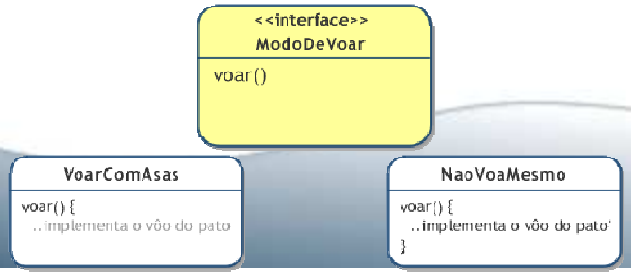
- Como desenvolver comportamento e manter a flexibilidade?
- Podemos alterar o comportamento de forma dinâmica?
- É possível alterar o comportamento em tempo de execução?



Princípio de Design

- "Programe para uma interface e não para uma implementação"
  - Utilizaremos interfaces para representar cada comportamento:





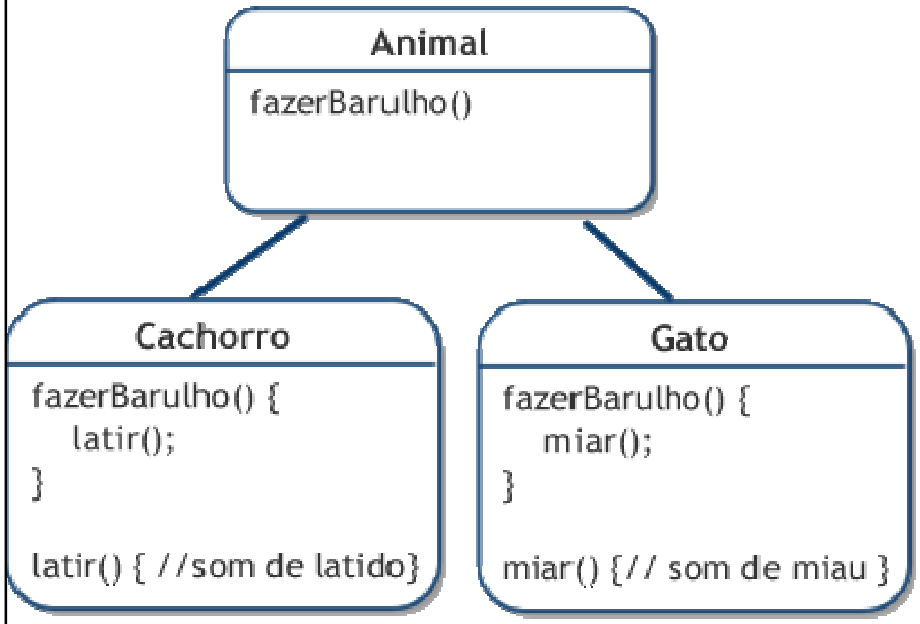
O que é diferente???

- Não é a classe Pato que implementa o comportamento
- Não existe uma implementação concreta destes comportamentos na classe Pato
- Isso nos deixava preso a estas implementações específicas
- As classes Pato irão usar um comportamento externo

Programar para uma *interface*

- Significa
  - Programar para um supertipo
- É possível programar desta maneira sem usar uma interface em Java
- Polimorfismo
  - O **tipo declarado → supertipo**
  - **Exemplo →**

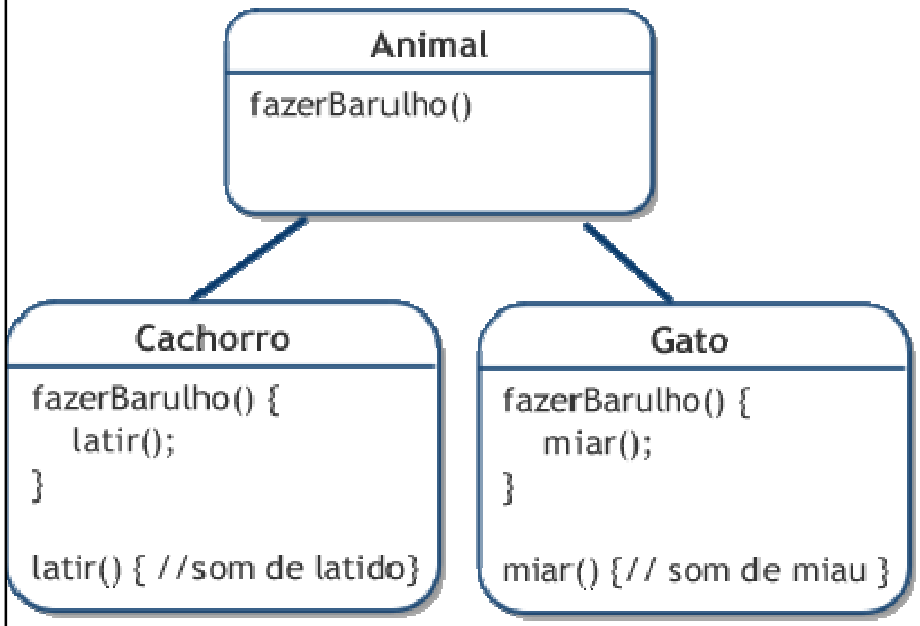
Programando Interface x Implementação



- **Para Implementação:**

```
Cachorro c = new Cachorro();

c.latir();
```

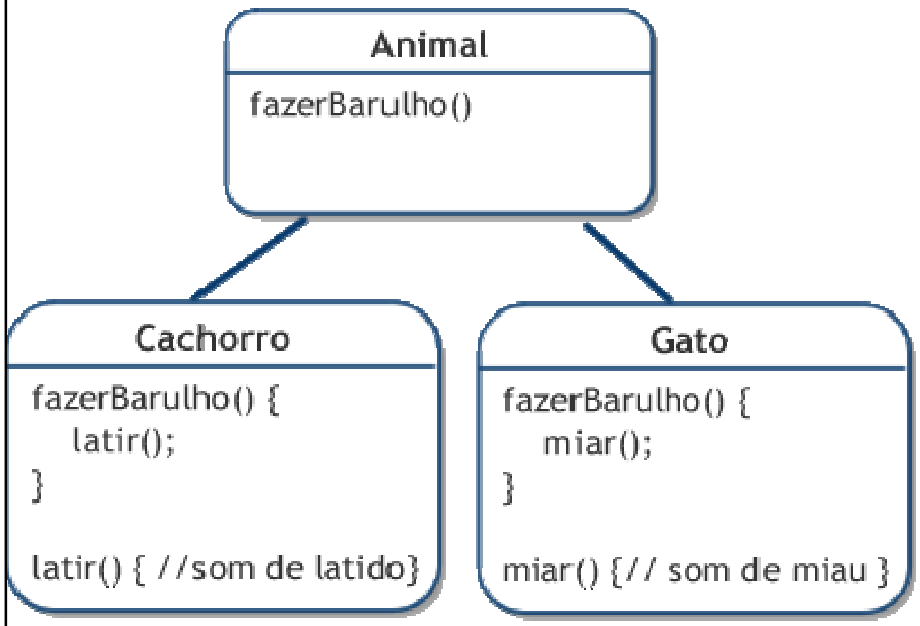
Programando Interface x Implementação



- **Para interface/supertipo:**

```
Animal c = new Cachorro();

c.fazerBarulho();
```

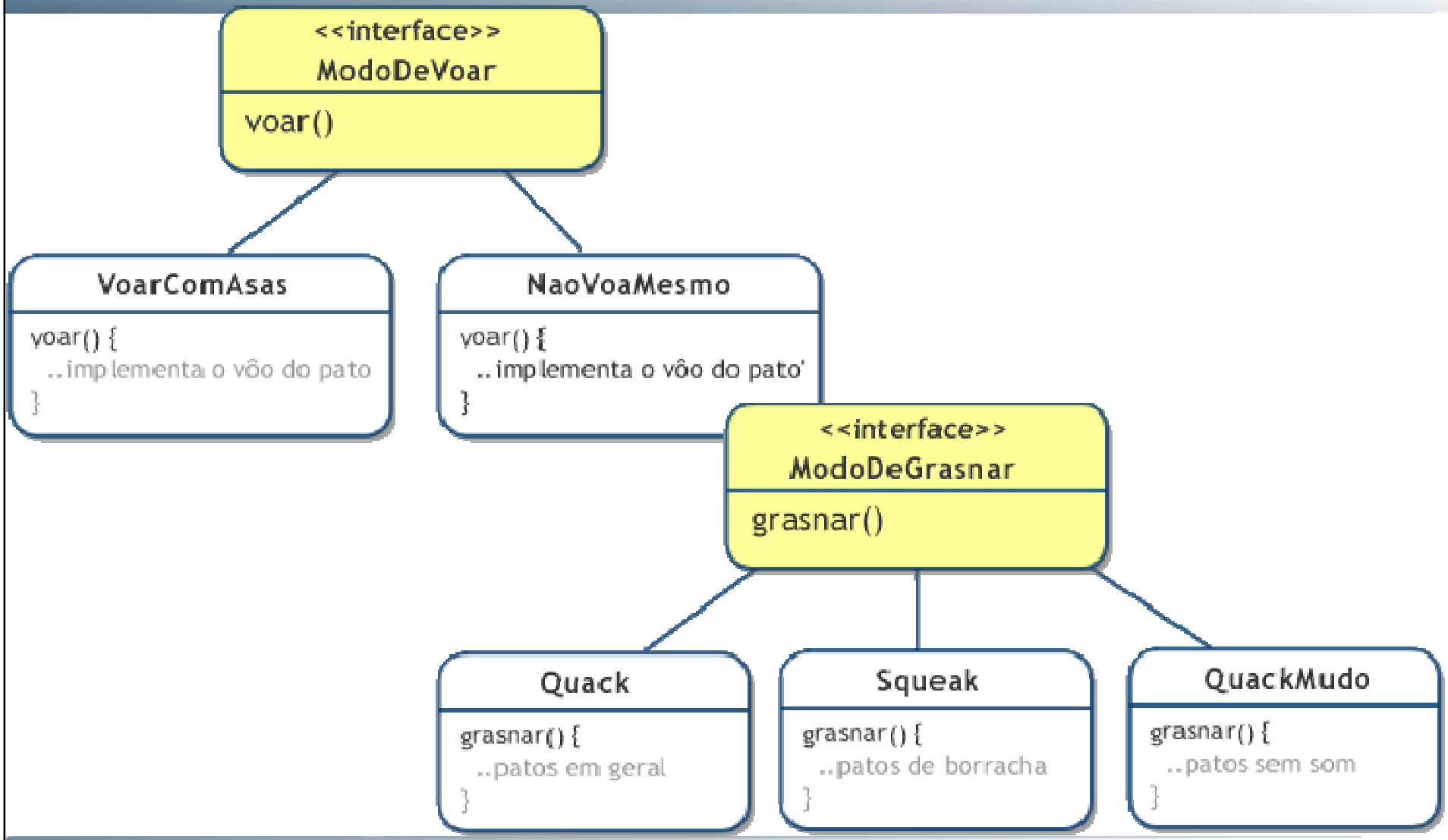
Programando Interface x Implementação



- **Melhorando**

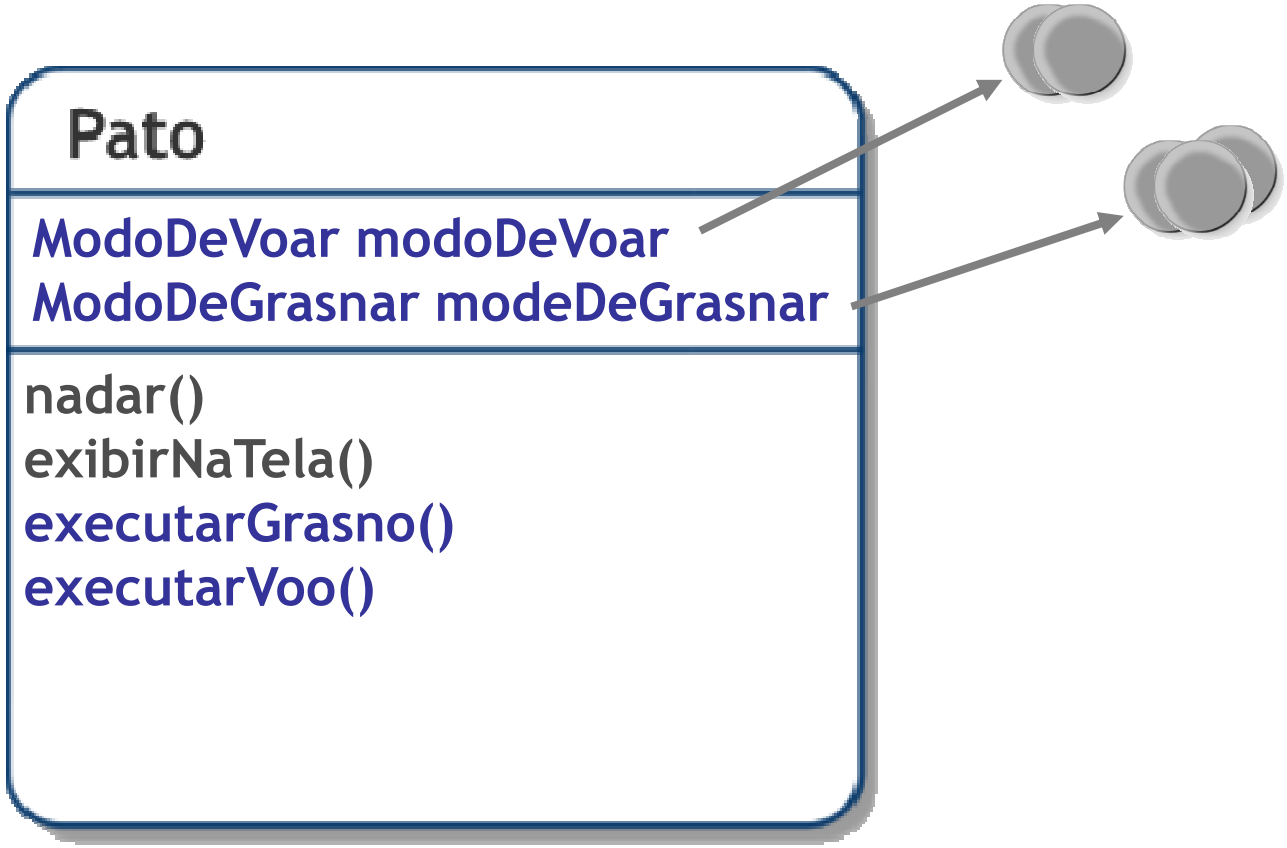
```
Animal c = getAnimal();

c.fazerBarulho();
```

Implementando os comportamentos



Integrando o comportamento

1. Adicionar 2 variáveis de instância ao Pato



Integrando o comportamento

2. Implementamos, por exemplo, o executarGrano()

```
public class Pato {
   ModoDeGrasnar modoDeGrasnar;

   public void executarGrasno() {
      modoDeGrasnar.grasnar();
   }
}
```

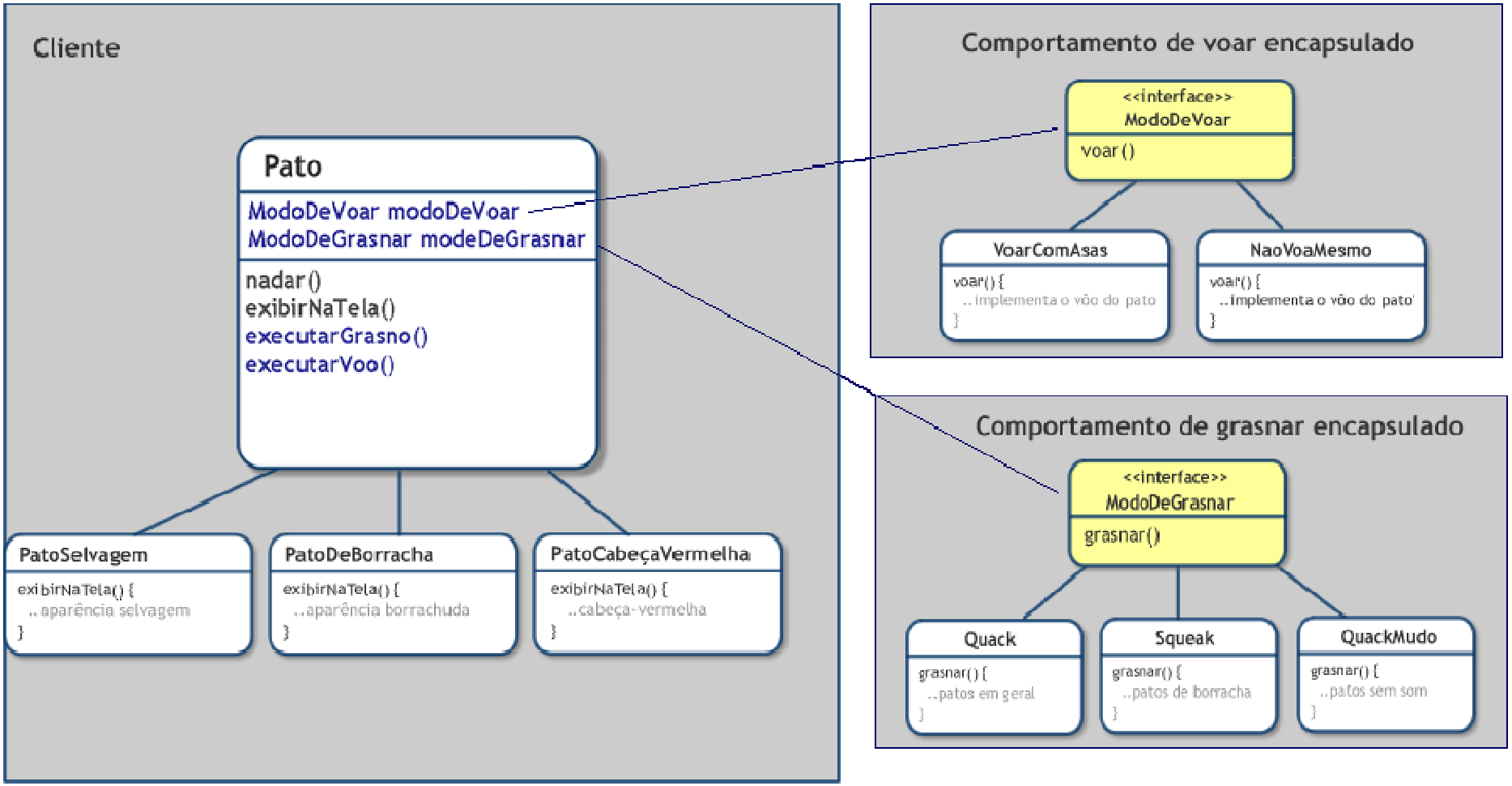
Delegou o comportamento para outra classe

Integrando o comportamento

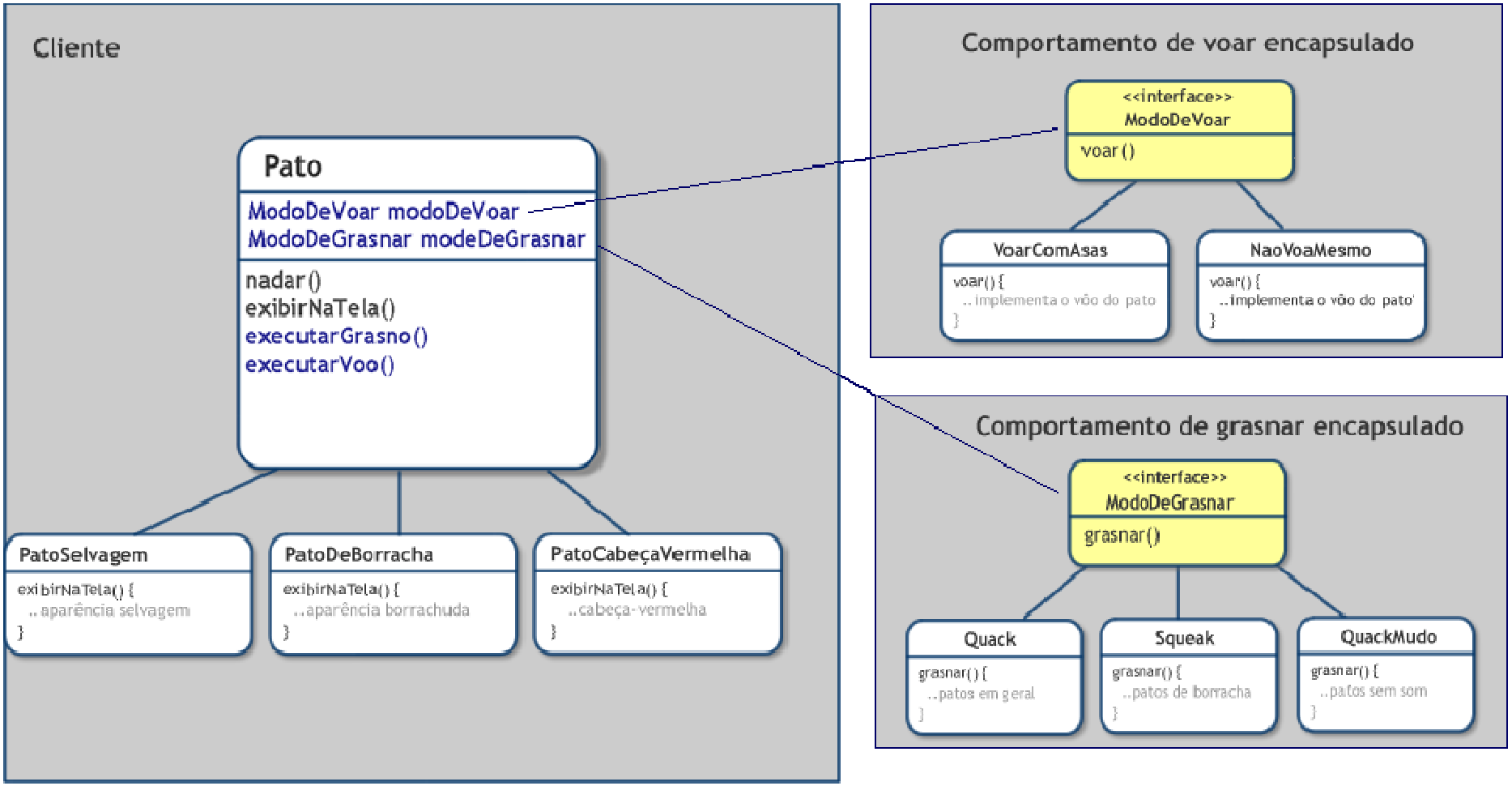
3. Como definir as variáveis de instância do comportamento?

```
public class PatoSelvagem extends Pato {

   public PatoSelvagem() {
     modoDeVoar = new VoarComAsas();
     modoDeGrasnar = new Quack();
   }

   public void exibirNaTela() {
     System.out.println("Eu sou um pato selvagem");
   }
}
```

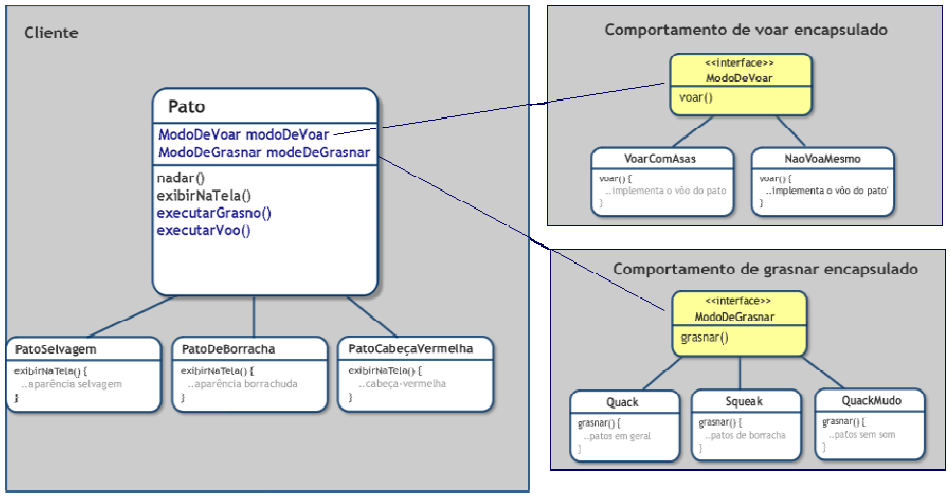
Tudo junto


Cliente
Pato
ModoDeVoar modoDeVoar
ModoDeGrasnar modeDeGrasnar
nadar()
exibirNaTela()
executarGrasno()
executarVoo()
PatoSelvagem
exibirNaTela() {
..aparência selvagem
}
PatoDeBorracha
exibirNaTela() {
..aparência borrachuda
}
PatoCabeçaVermelha
exibirNaTela() {
..cabeça-vermelha
}
Comportamento de voar encapsulado
<<interface>>
ModoDeVoar
voar()
VoarComAsas
voar() {
..implementa o vôo do pato
}
NaoVoaMesmo
voar() {
..implementa o vôo do pato
}
Comportamento de grasnar encapsulado
<<interface>>
ModoDeGrasnar
grasnar()
Quack
grasnar() {
..patos em geral
}
Squeak
grasnar() {
..patos de borracha
}
QuackMudo
grasnar() {
..patos sem som
}

Verifica-se a composição



Princípio de Design



- "Dar prioridade à composição ao invés da herança"
  - Maior flexibilidade
  - Conjunto de algoritmos encapsulados em uma família de classes
  - Alteração do comportamento em tempo de execução

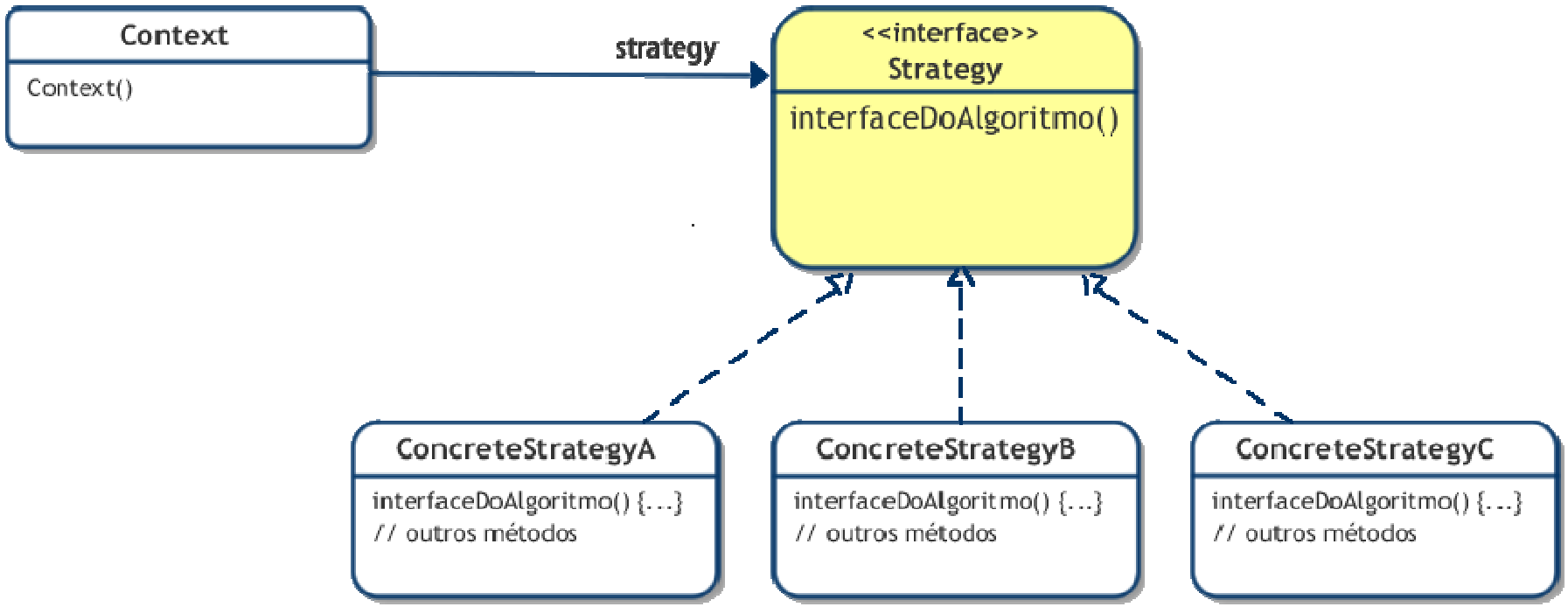
Primeiro Padrão → STRATEGY

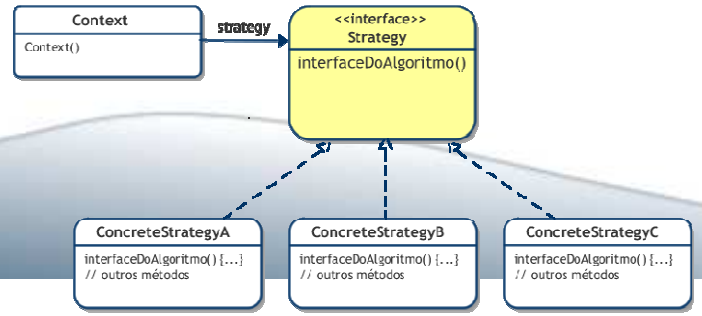
Define uma **família** de **algoritmos**, **encapsula** cada um deles e os torna **intercambiáveis**. O **Strategy** permite que os algoritmos **variem** independentemente dos clientes que o utilizam.

Aplicabilidade

- Muitas classes relacionadas diferem somente no seu comportamento.
  - O *Strategy* fornecem uma maneira de configurar uma classe com um, dentre muitos comportamentos.
- Precisa-se de variantes de um algoritmo.
- Um algoritmo usa dados de que os clientes não deveriam ter conhecimento.
  - O *Strategy* evita a exposição de estruturas de dados complexos específicas de um algoritmo.
- Uma classe define muitos comportamentos e estes aparecem em suas operações como múltiplos comandos condicionais da linguagem.
  - Mova os condicionais para sua classe *Strategy.*

Strategy
Diagrama de Classes





Participantes

- **Context**
  - Possui uma referência para um objeto **Strategy**
  - É configurado com um objeto **ConcreteStrategy**
  - Pode definir uma interface que permite o **Strategy** acessar seus dados
- **Strategy**
  - Define uma interface comum para todos os algoritmos suportados. **Context** usa essa interface para chamar um algoritmo definido por um **ConcreteStrategy**
- **ConcreteStrategy**
  - Implementa o algoritmo usando a interface **Strategy**

Colaborações

- Strategy e Context interagem para implementar o algoritmo escolhido.
- Um Context repassa solicitações dos seus clientes para seu Strategy.

Consequências

- Famílias de algoritmos relacionados.
- Alternativa ao uso de subclasses.
- Possibilidade da escolha de implementações.

Consequências

- Os clientes devem conhecer diferentes Strategies
  - O cliente deve compreender qual a diferença entre os Strategies
  - Usar o padrão Strategy somente quando a variação em comportamento é importante
- Custo de comunicação entre Strategy e Context
- Aumento do número de objetos

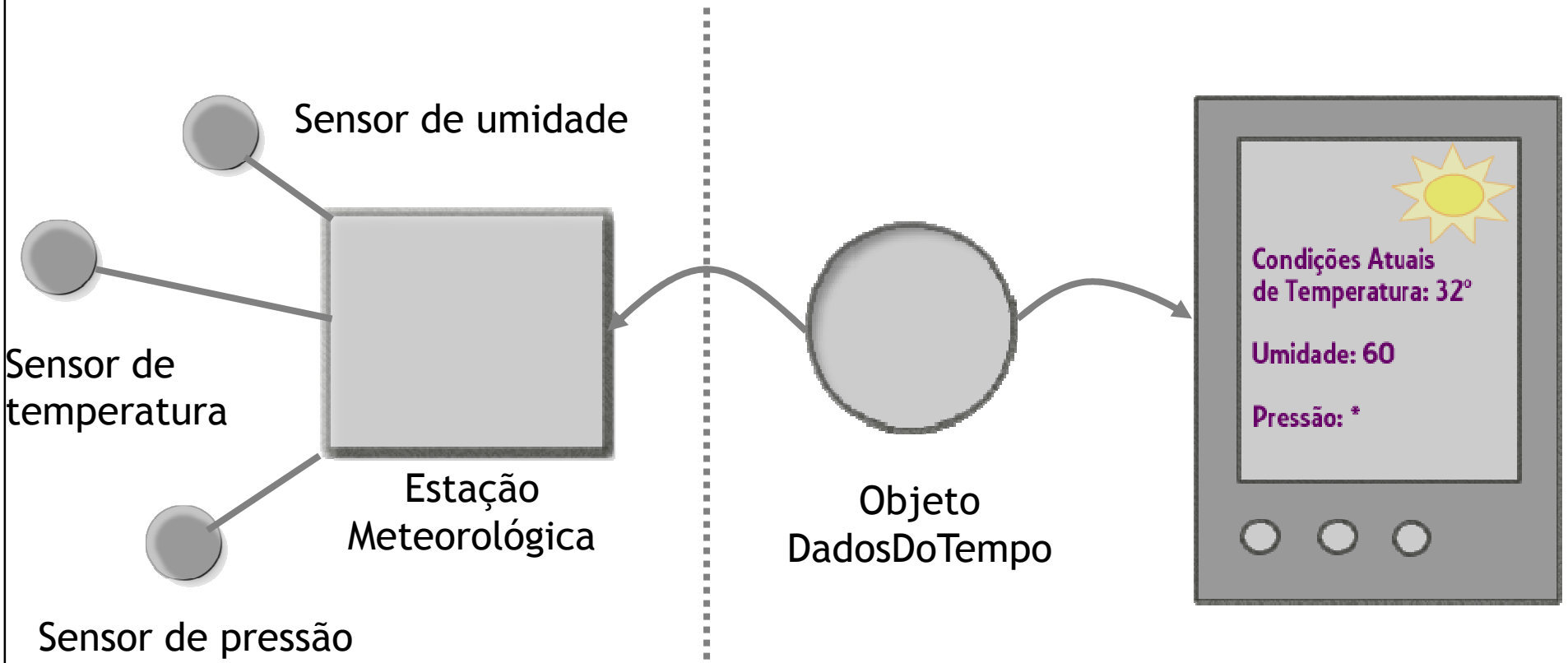
Padrão

Observer

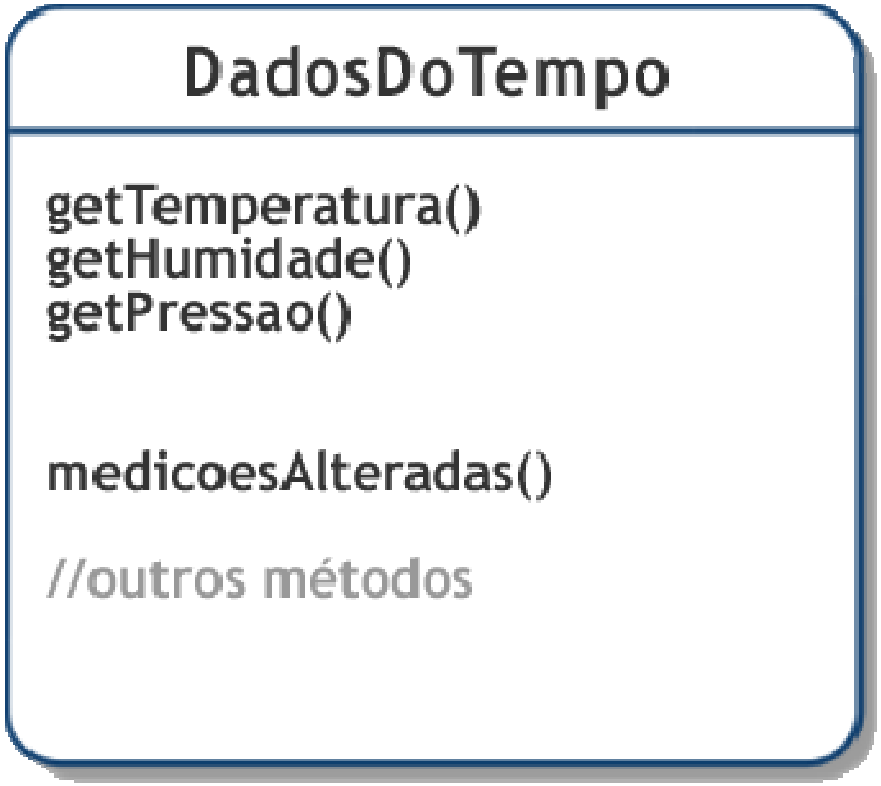
O Padrão Observer

- Nossa turma acaba de ganhar um contrato!
- Aplicação Web para monitoração do tempo.

Estação Meteorológica Visão Geral



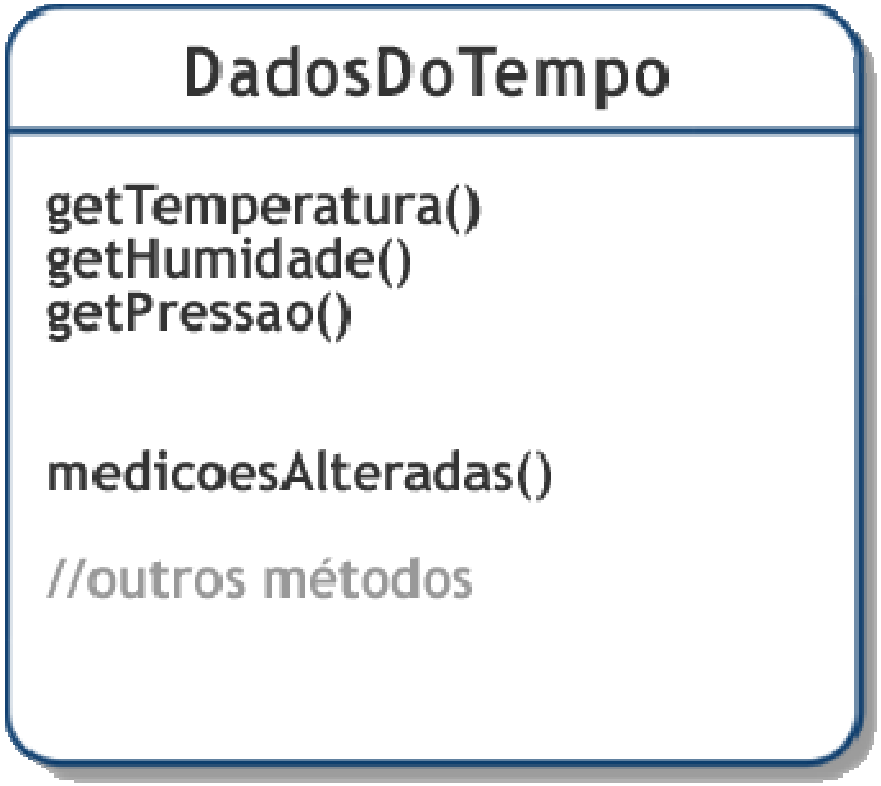
A classe DadosDoTempo



- Retornam as medições mais recentes
  - Este objeto sabe como obter estas informações

```
/**
 * Este método é executado
 * sempre que as medidas
 * meteorológicas forem alteradas
 *
 */
public void medicoesAlteradas() {
   //seu código aqui
}
```

A classe DadosDoTempo



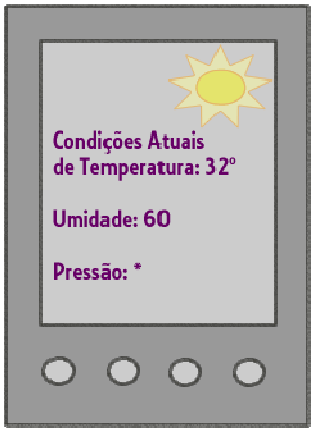
- Tarefa:
  - Implementar medicoesAlteradas() de modo que atualize as três visões: condições climáticas atuais, status meteorológico e previsão

Juntando as informações

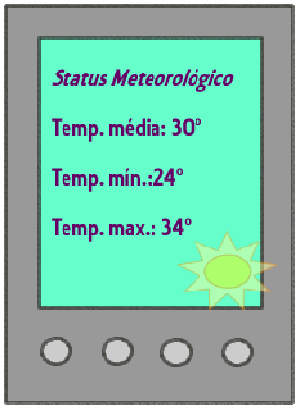
- A classe DadosDoTempo possui métodos para obter 3 medições diferentes:
  - Temperatura
  - Umidade
  - Pressão
- O método **medicoesAlteradas** é chamado sempre que dados de medição estão disponíveis (não sabemos como esse método é chamado, apenas que ele é).

Requisitos

- Precisamos implementar 3 elementos de exibição que usem os dados meteorológicos
  - Condições Atuais
  - Estatísticas
  - Previsão



Tela 1



Tela 2

Uma implementação

```
public void medicoesAlteradas() {
  float temperatura = getTemperatura();
  float umidade = getUmidade();
  float pressao = getPressao();

  telaCondicoes.atualizar(
                  temperatura, umidade, pressao);
  telaEstatisticas.atualizar(
                  temperatura, umidade, pressao);
  telaPrevisao.atualizar(temperatura, umidade, pressao);
}
```

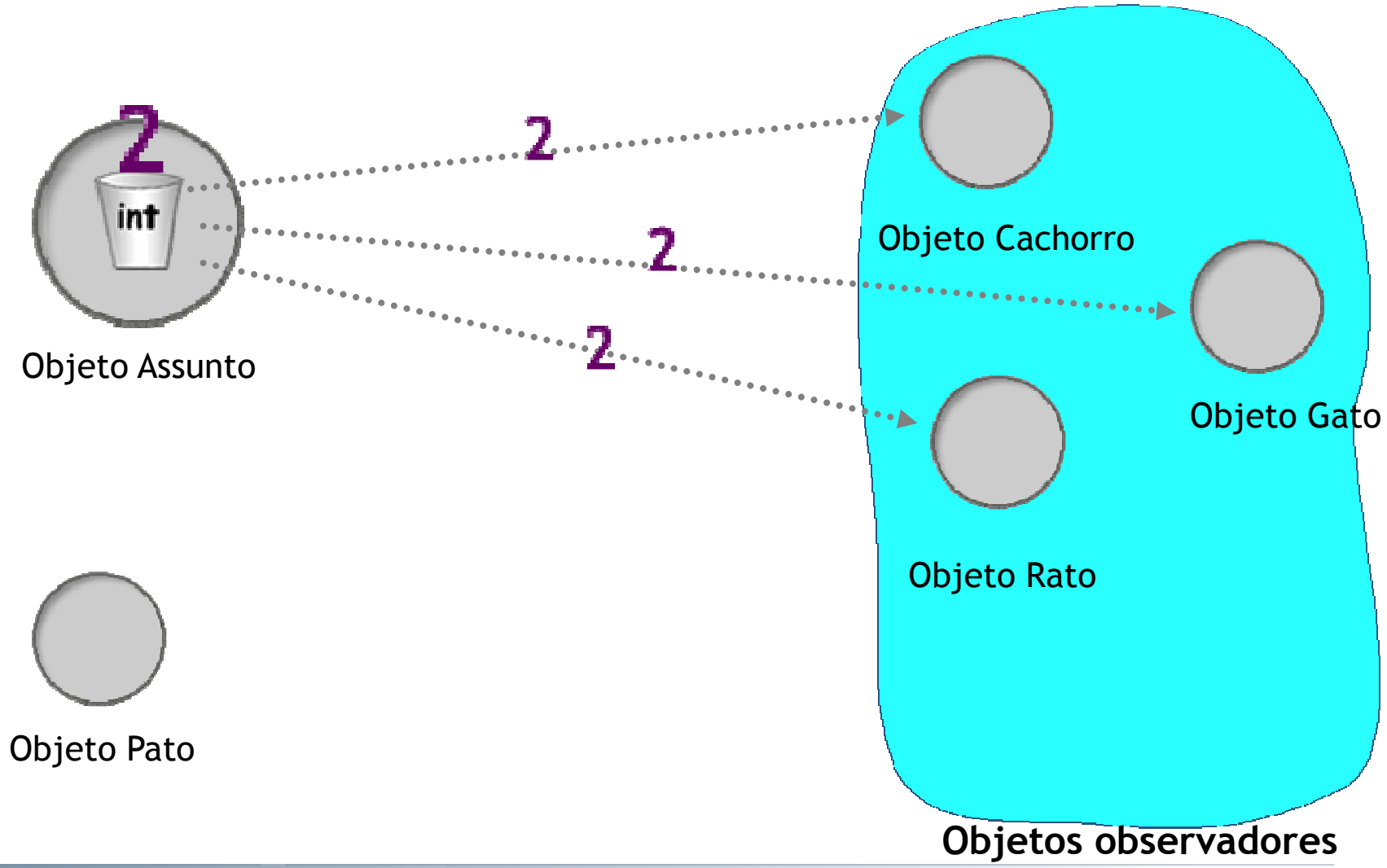
Como funciona o padrão Observer?

- Como uma assinatura de revista ou jornal
  - Uma editora publica uma revista.
  - Você assina a revista e sempre recebe novas edições.
  - Enquanto assinante, continua recebendo revistas.
  - Você cancela a assinatura quando não quer mais receber as revistas.
  - Enquanto a editora permanecer em um negócio, outras pessoas, empresas e companhias podem assinar e cancelar o recebimento da revista.

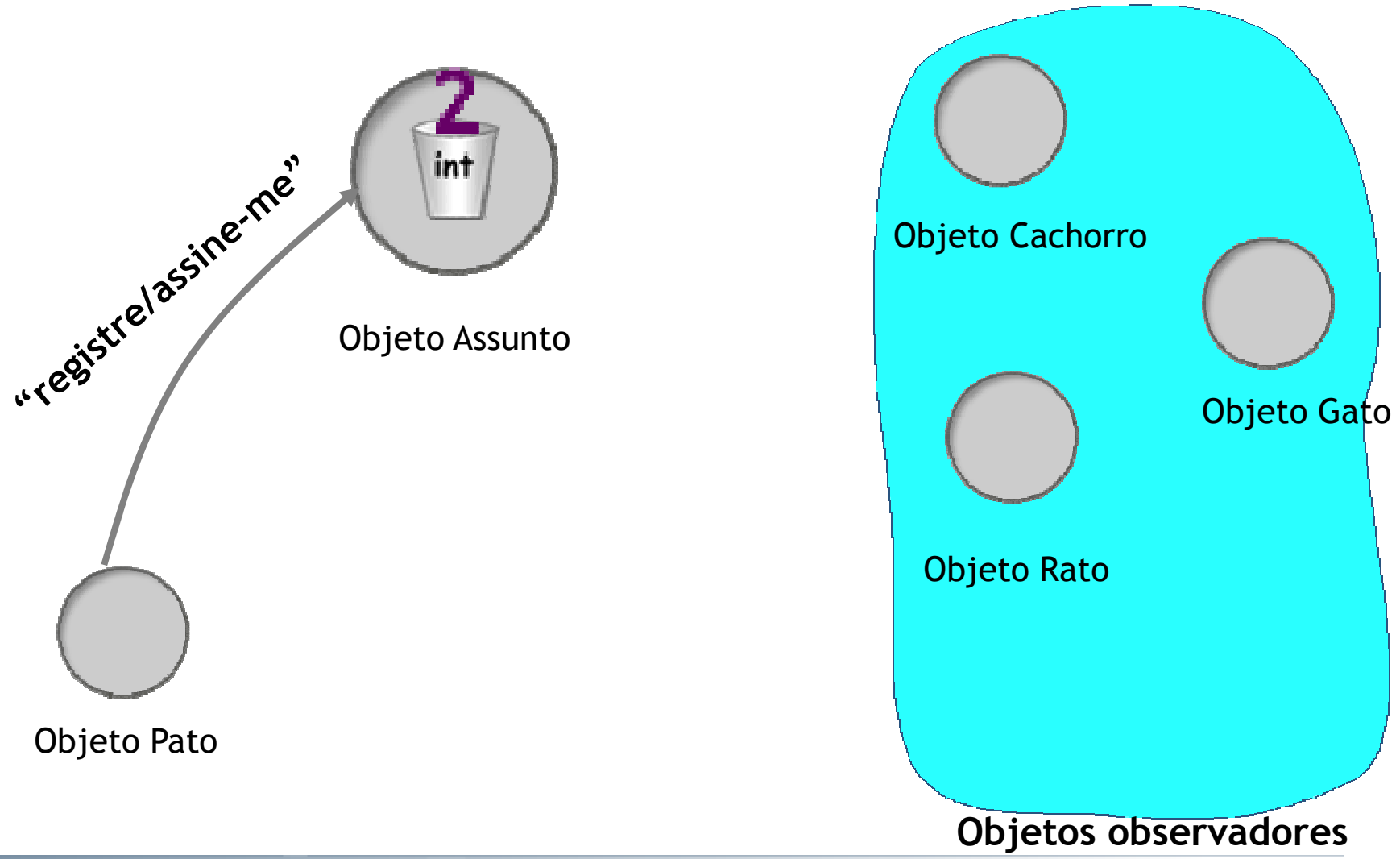
Da revista para o padrão

- Editora → Assunto (Subject)
- Assinantes → Observadores (Observers)

Padrão Observer

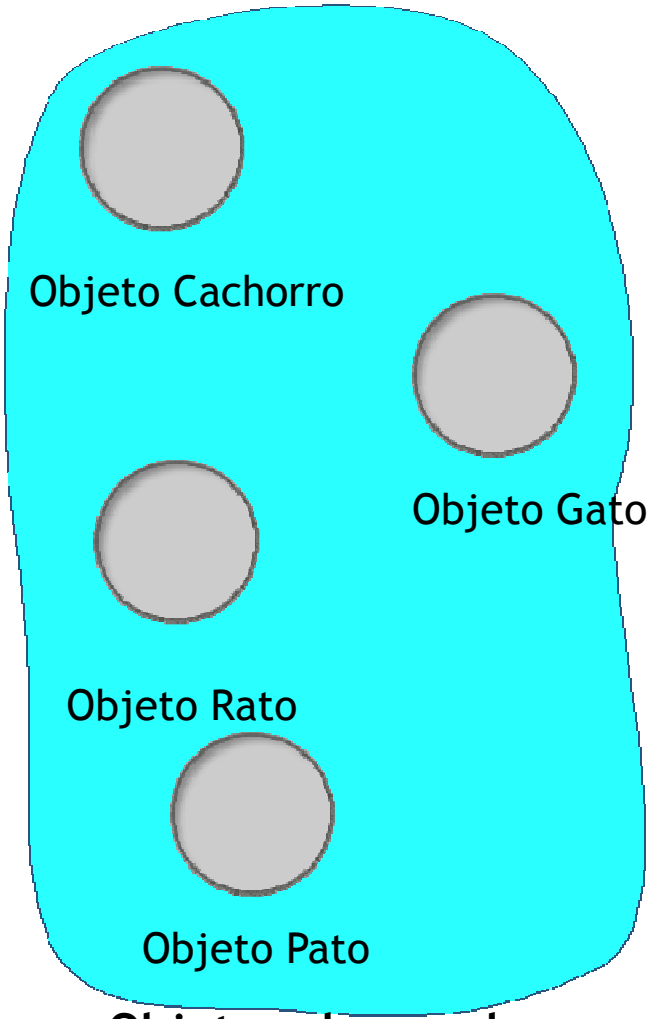


O Pato informa ao Objeto Assunto que quer ser um observador



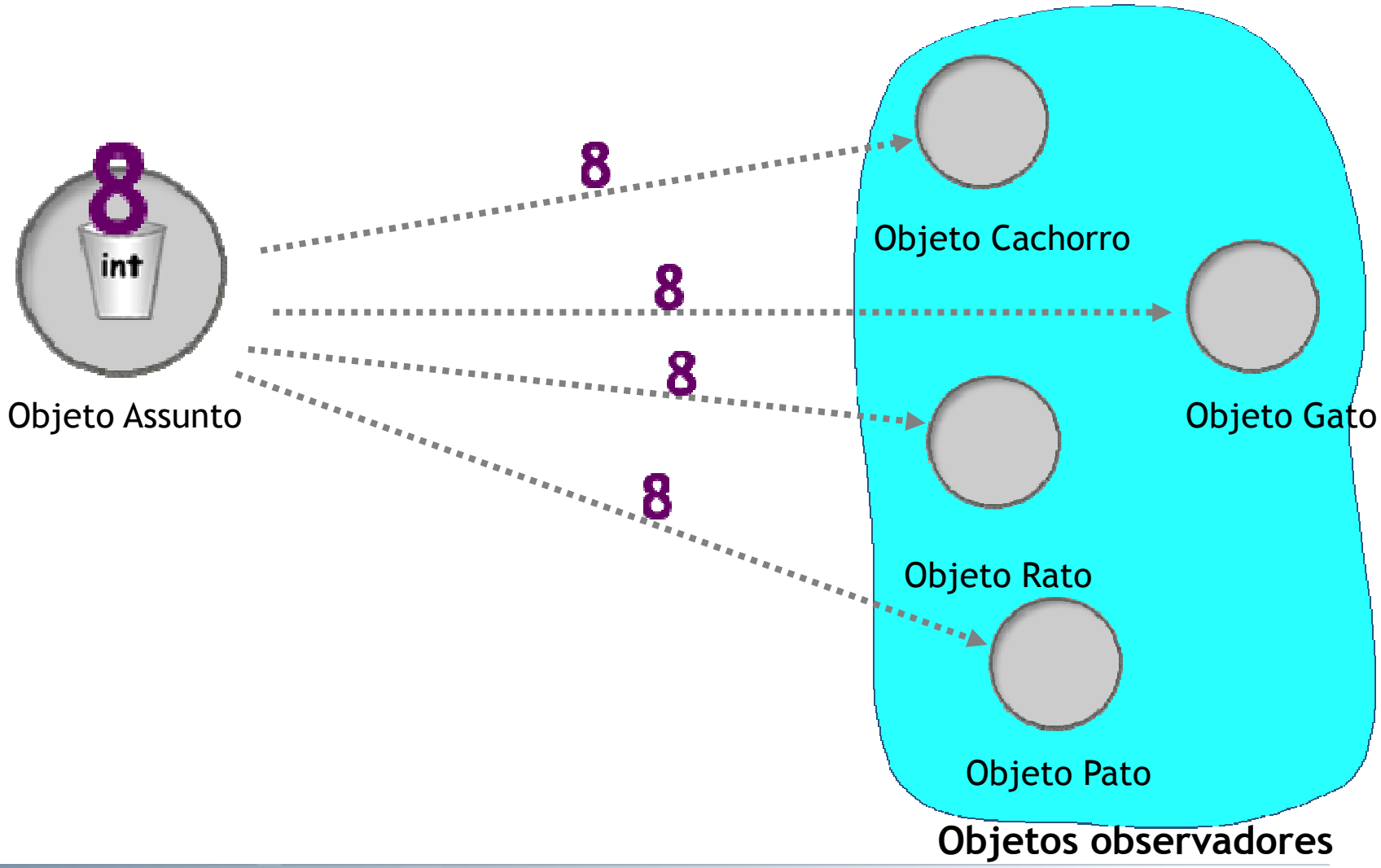
Agora o Pato é um observador

Objeto Assunto

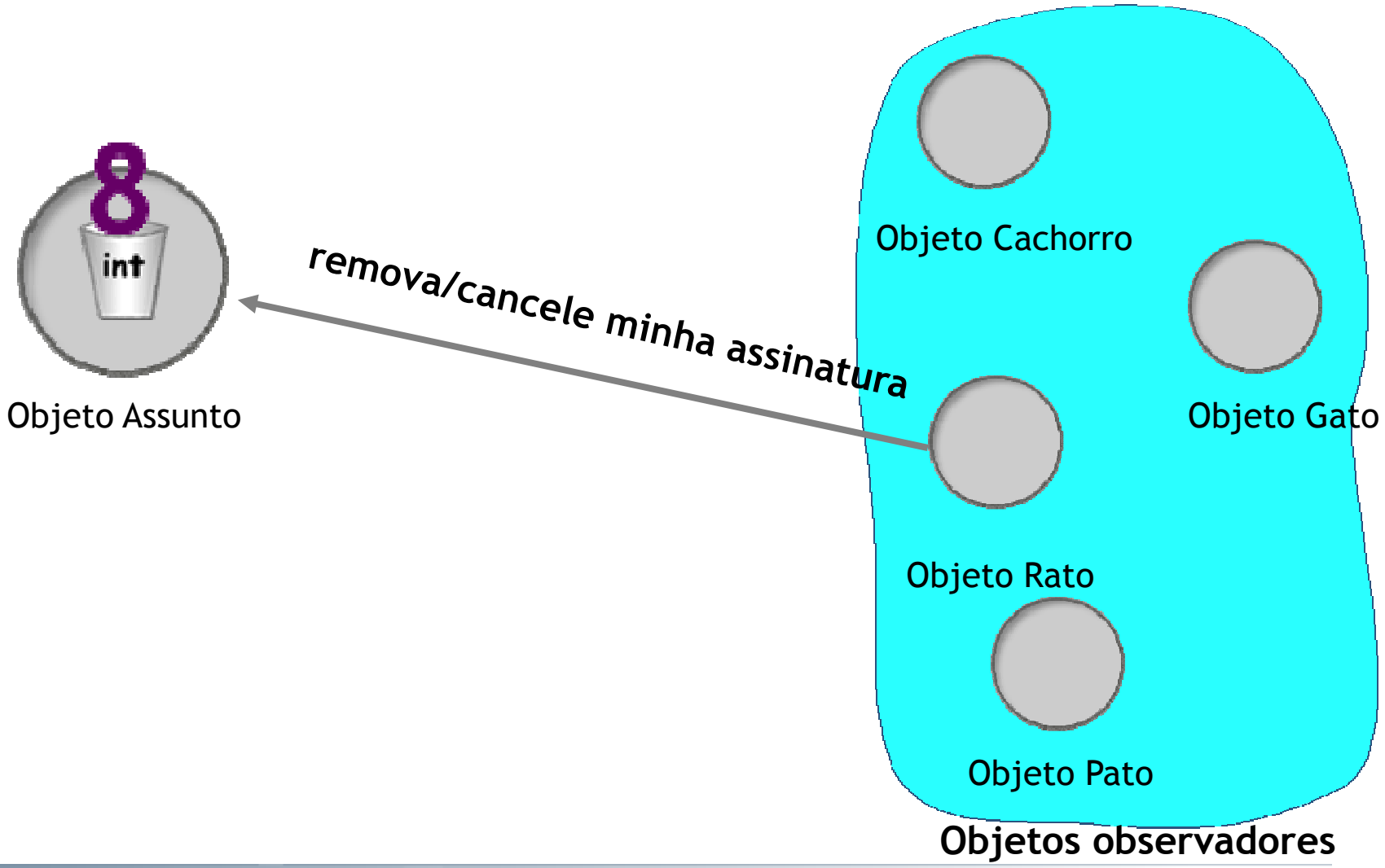


Objetos observadores

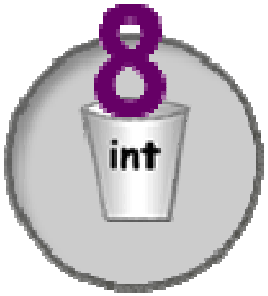
O Assunto recebe um novo valor



O objeto Rato não quer mais ser um observador



O Rato está fora



Objeto Assunto

Objeto Rato

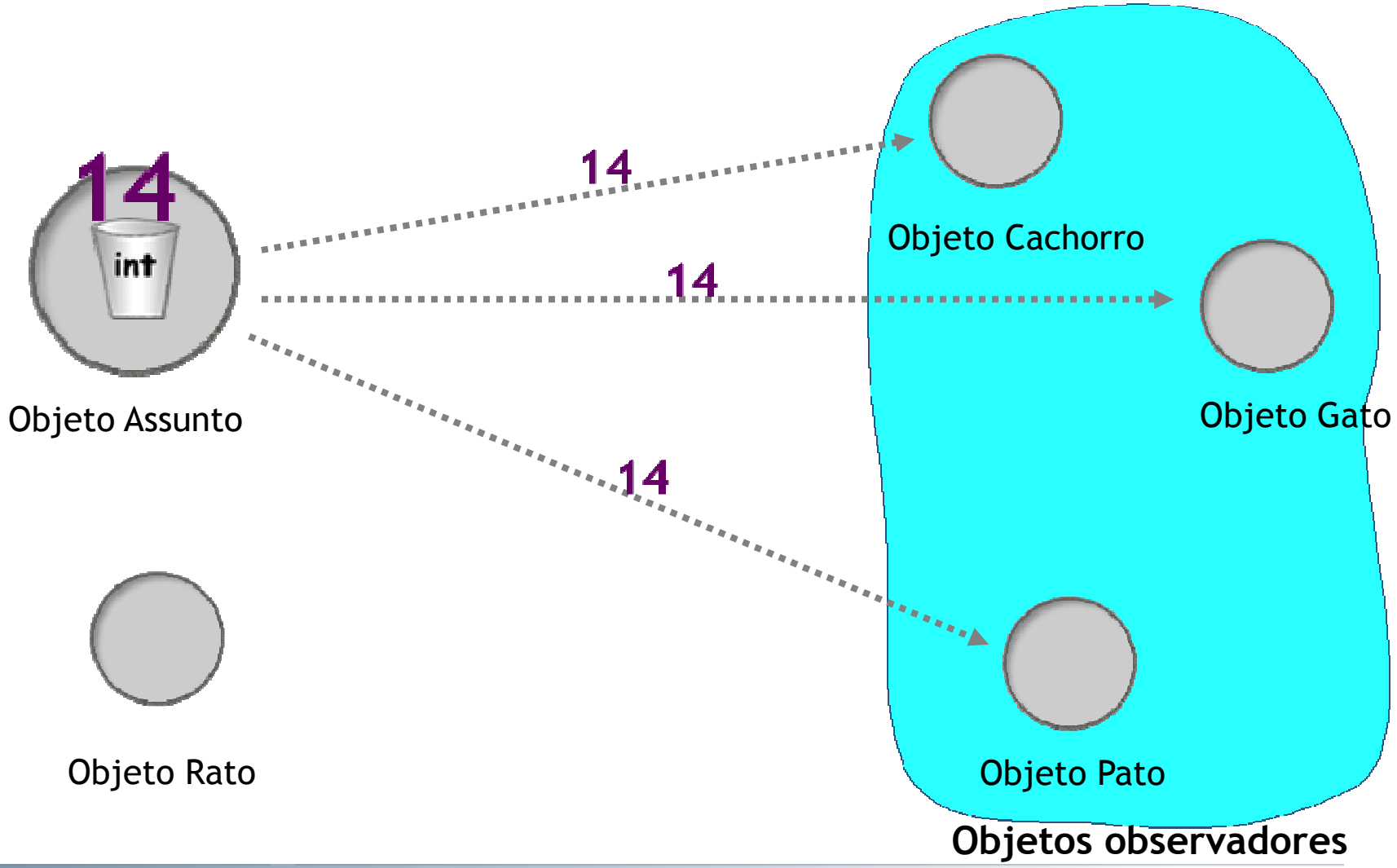
Objeto Cachorro

Objeto Gato

Objeto Pato

Objetos observadores

O Assunto recebe um novo valor



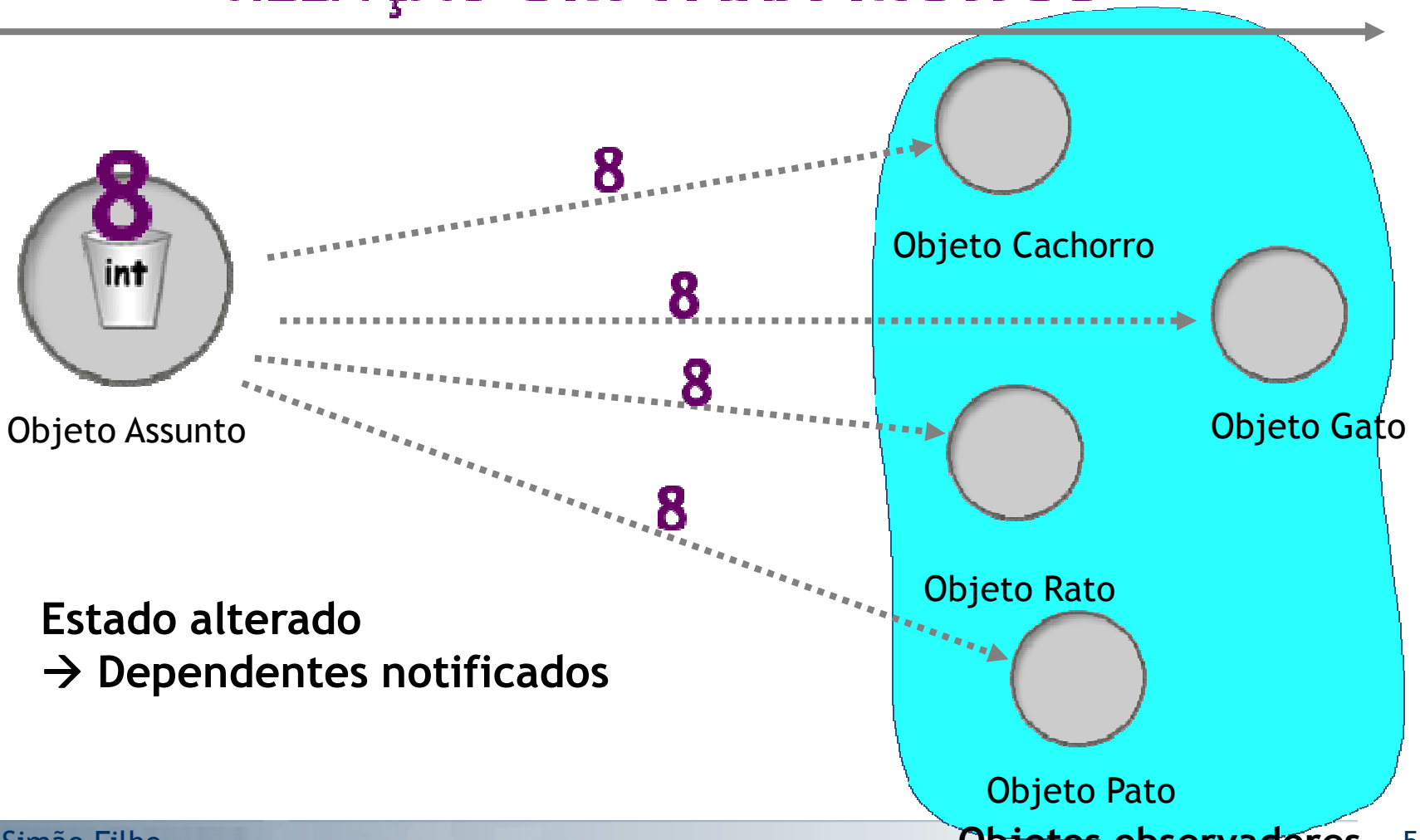
Segundo Padrão OBSERVER

O **Padrão Observer** define a dependência um-para-muitos entre objetos para que, quando um objeto **mude** de estado, todos os seus **dependentes** sejam **avisados** e **atualizados** automaticamente.

Aplicabilidade

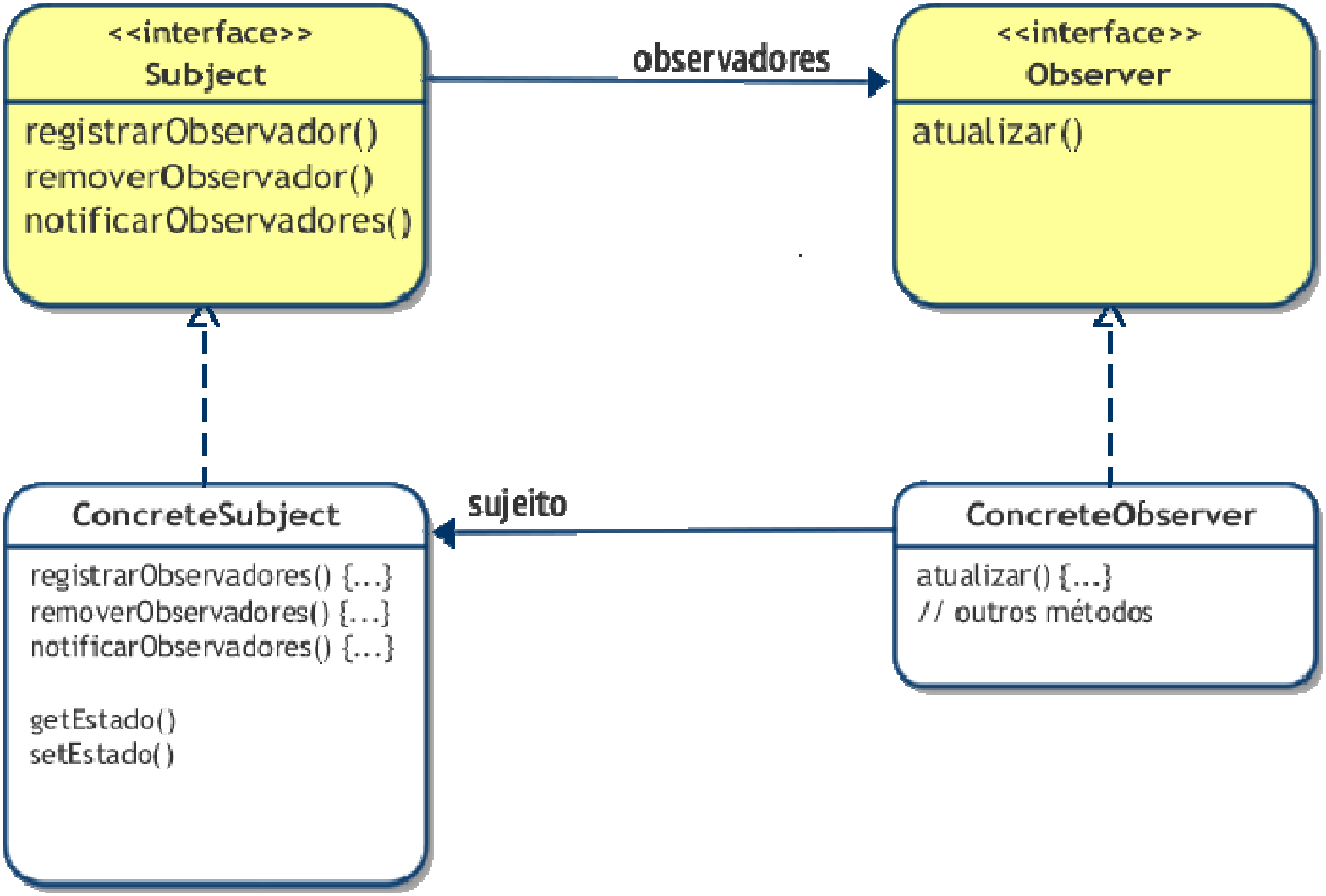
- Uma abstração tem dois aspectos, um dependente do outro.
- Uma mudança em um objeto exige mudanças em outros, e não se sabe quantos objetos necessitam ser mudados.
- Um objeto deve ser capaz de notificar outros objetos sem fazer hipóteses sobre quem são estes objetos.
  - Fraco acoplamento.

RELAÇAO UM-PARA-MUITOS



Estado alterado
→ Dependentes notificados

Observer
O diagrama de classes



Participantes

- **Subject**
  - Conhece os seus observadores. Um número qualquer de objetos Observer pode observar um Subject.
  - Fornece uma interface para acrescentar e remover objetos para associar e desassociar objetos Observer.
- **ConcreteSubject**
  - Armazena estados de interesse para objetos ConcreteObserver.
  - Envia uma notificação para os seus observadores quando seu estado muda.

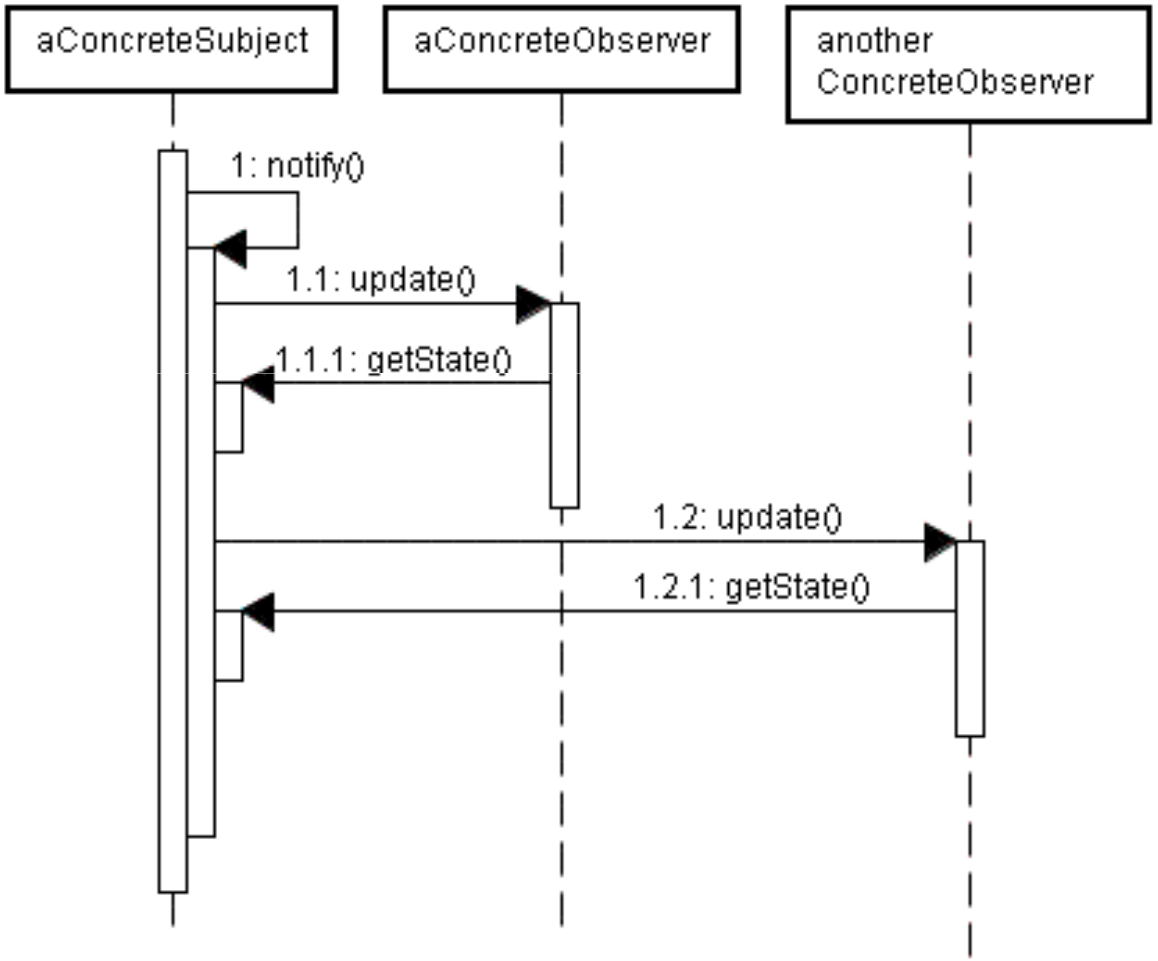
Participantes

- **Observer**
  - Define uma interface de atualização para objetos que deveriam ser notificados sobre mudanças em um Subject.
- **ConcreteObserver**
  - Mantém uma referência para um objeto ConcreteSubject.
  - Armazena estados que deveriam permanecer consistentes com os do Subject.
  - Implementa a interface de atualização de Observer, para manter seu estado consistente com o do Subject.

Colaborações

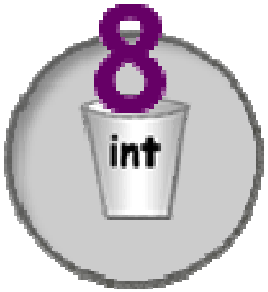
- O ConcreteSubject notifica seus observadores sempre que ocorrer uma mudança que poderia tornar inconsistente o estado deles com o seu próprio.
- Após ter sido informado de uma mudança no ConcreteSubject, um objeto ConcreteObserver pode consultá-lo para obter informações.
- O ConcreteObserver usa esta informação para reconciliar o seu estado com aquele ConcreteSubject.

Colaborações

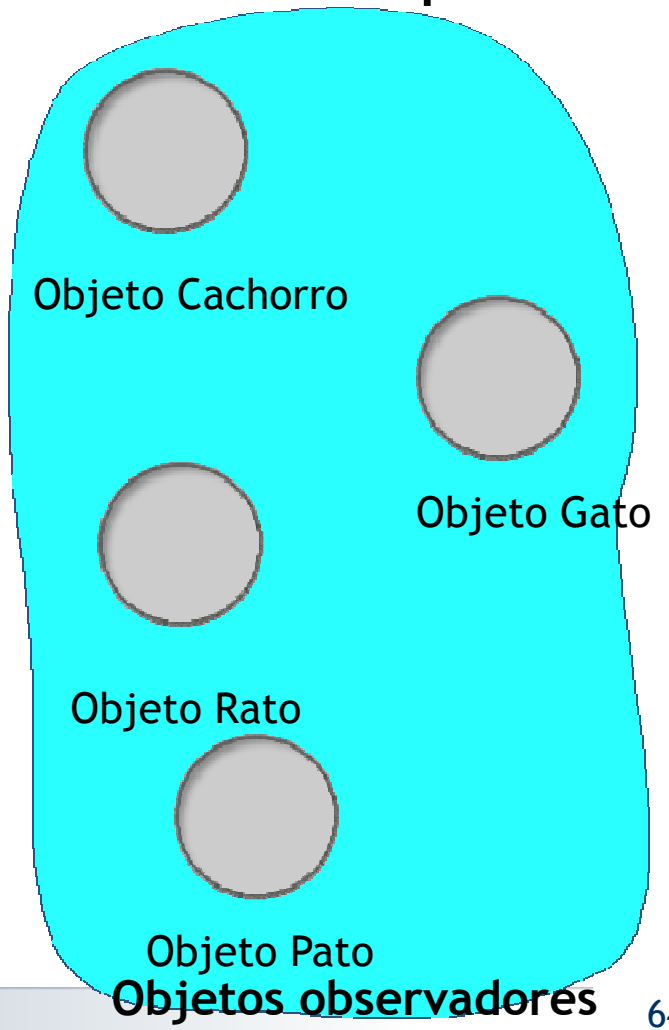


A dependência entre os participantes

- O Assunto é o objeto que contém o estado e que o controla



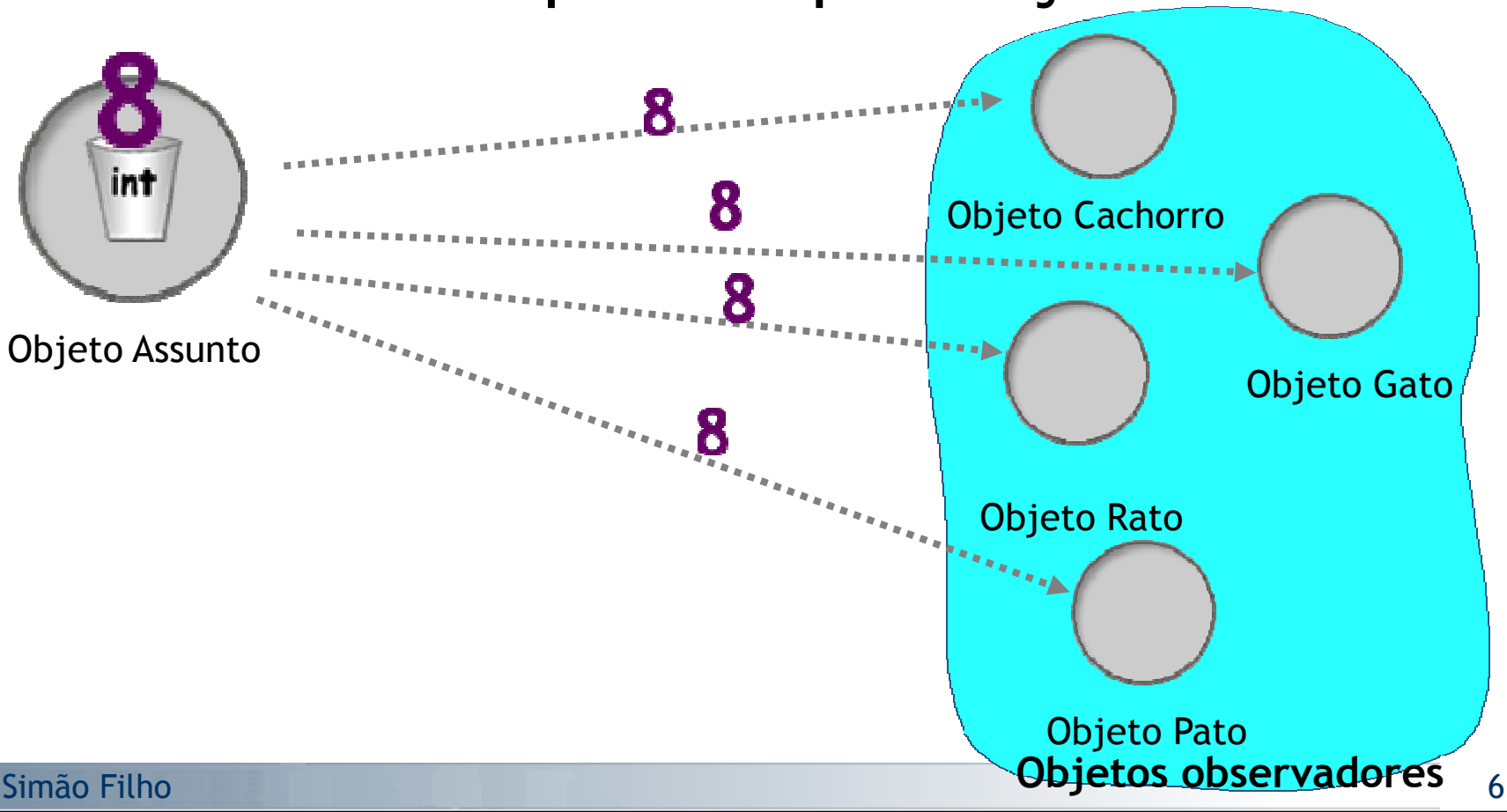
Objeto Assunto



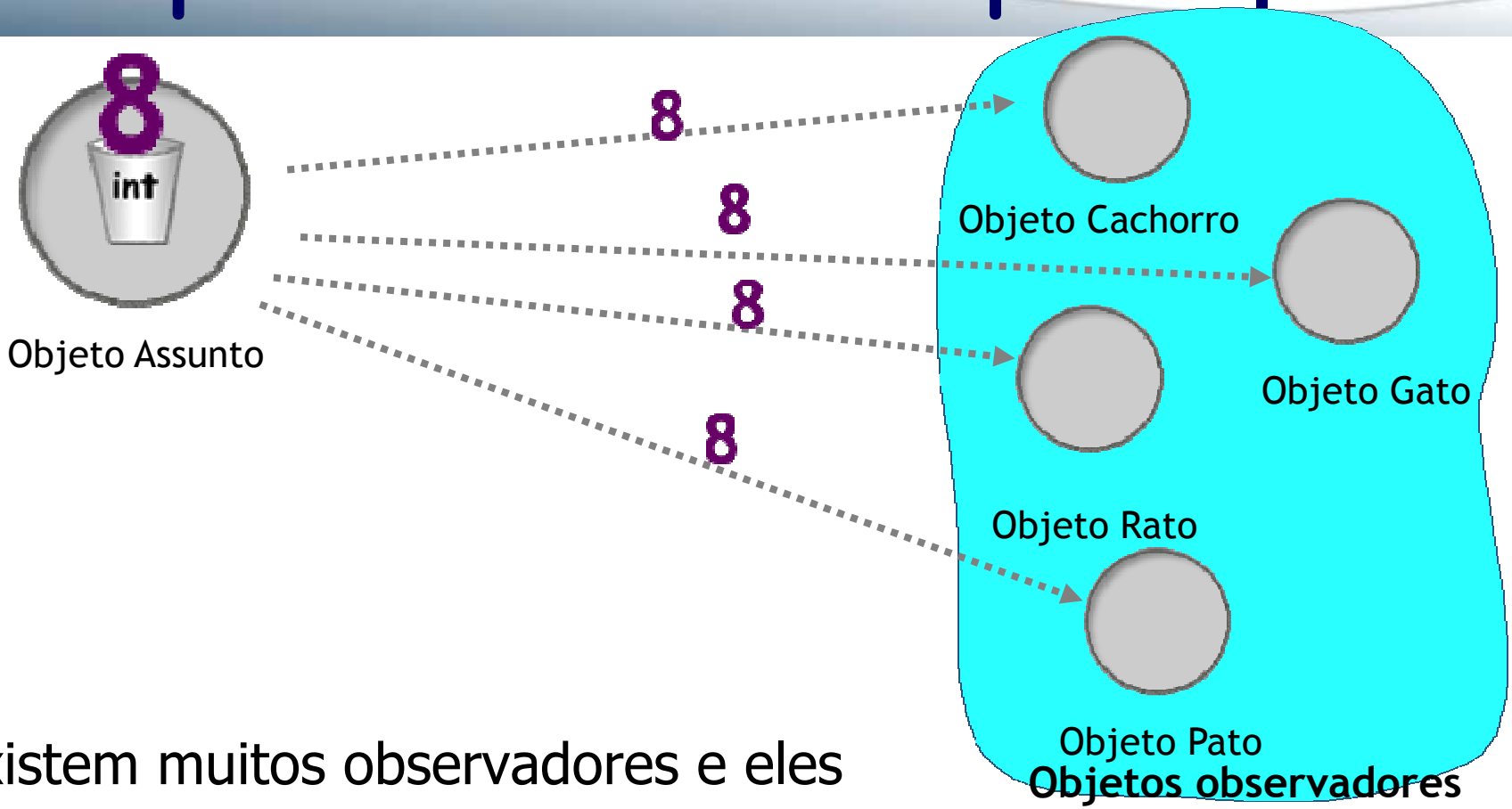
- Existe apenas UM assunto com estado

A dependência entre os participantes

- Os observadores usam este estado quando necessário ainda que não pertença a eles



A dependência entre os participantes



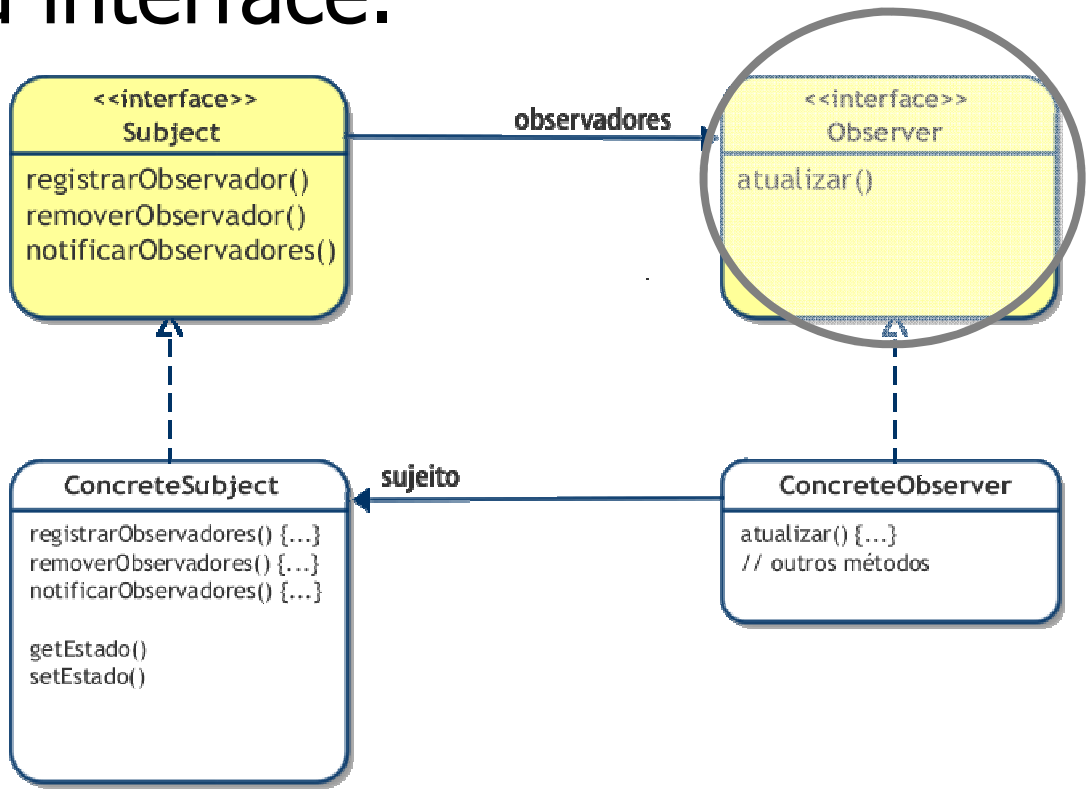
- Existem muitos observadores e eles confiam no Assunto para contar-lhes quando ocorrerem alterações no estado

Fraco Acoplamento

- Objetos que interagem entre si.
- Possuem pouco conhecimento sobre cada um.
- Padrão Observer
  - Fornece um design fracamente acoplado
  - Por quê?

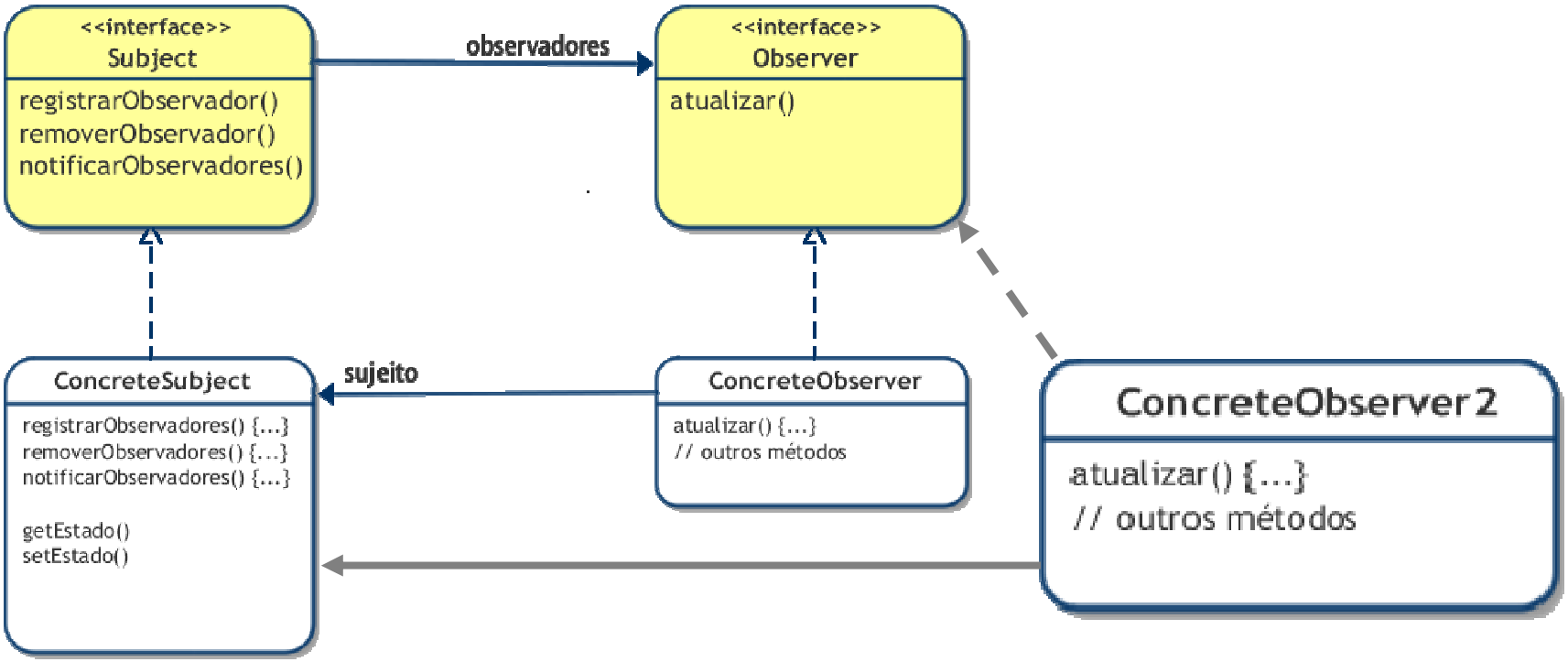
Fraco Acoplamento no Observer

- A única coisa que o Assunto (Subject) sabe sobre um Observador é que ele implementa um determinada interface.



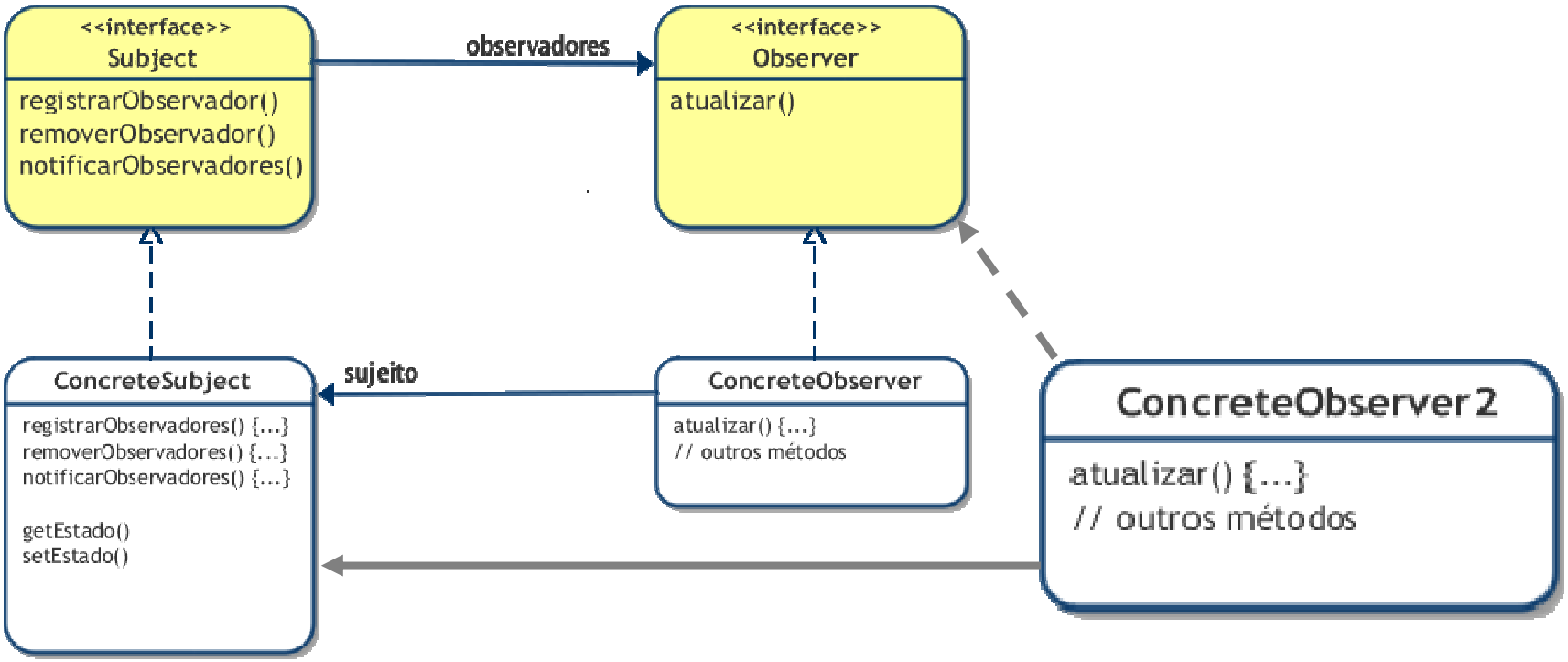
Fraco Acoplamento no Observer

- É possível adicionar um novo observador a qualquer momento.



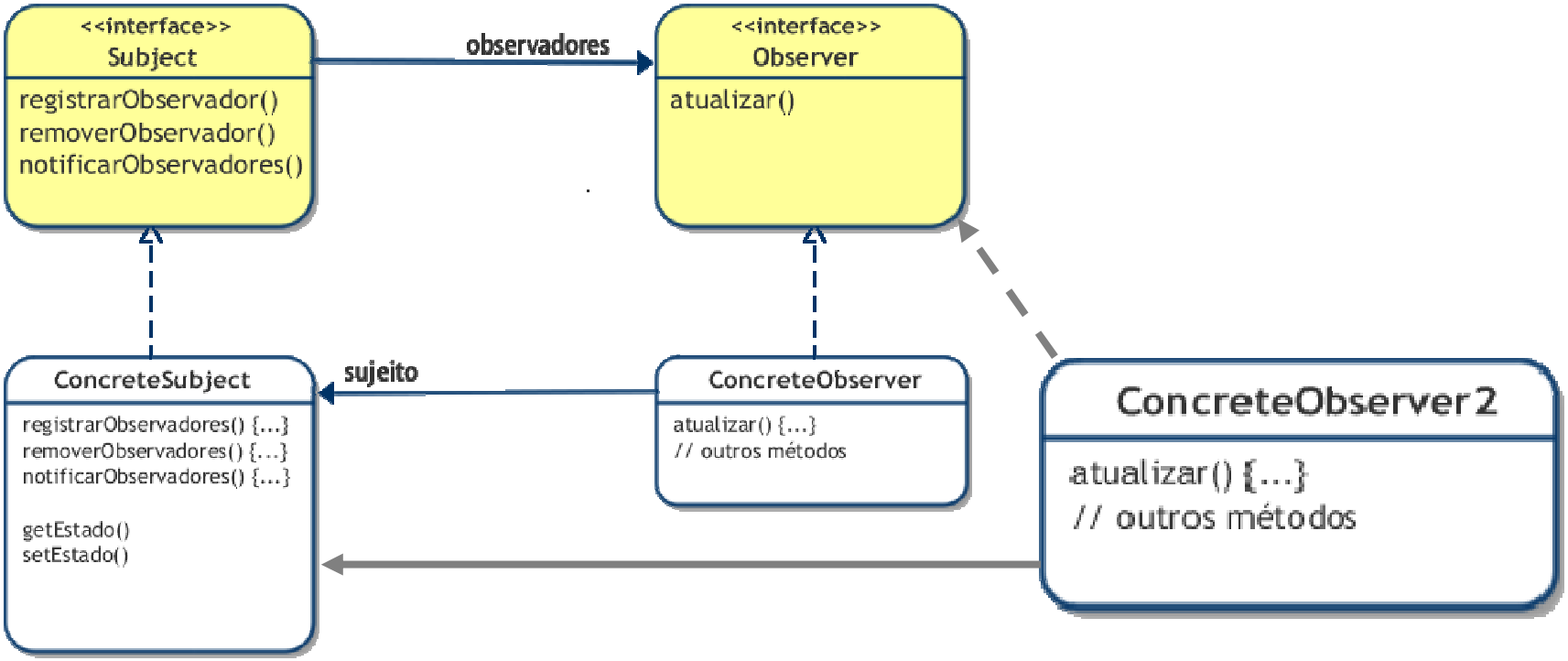
Fraco Acoplamento no Observer

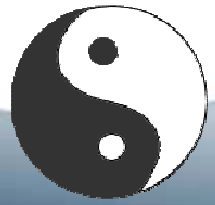
- Não precisamos modificar o Subject (Assunto) para adicionar novos tipos de observadores.



Fraco Acoplamento no Observer

- Podemos reusar os objetos.
- Alterações feitas em cada um deles não afetarão o outro.

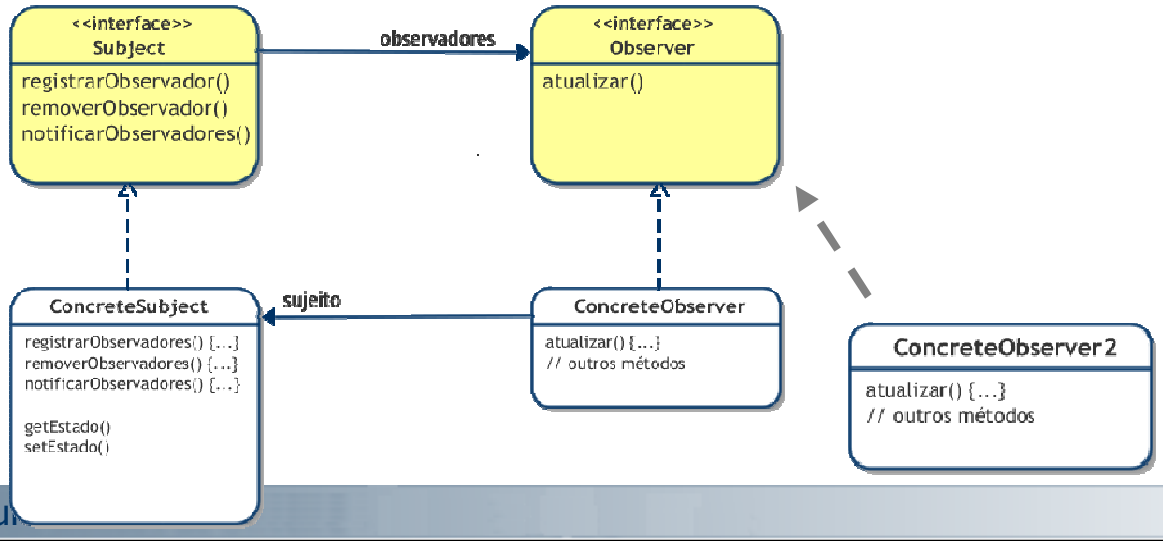




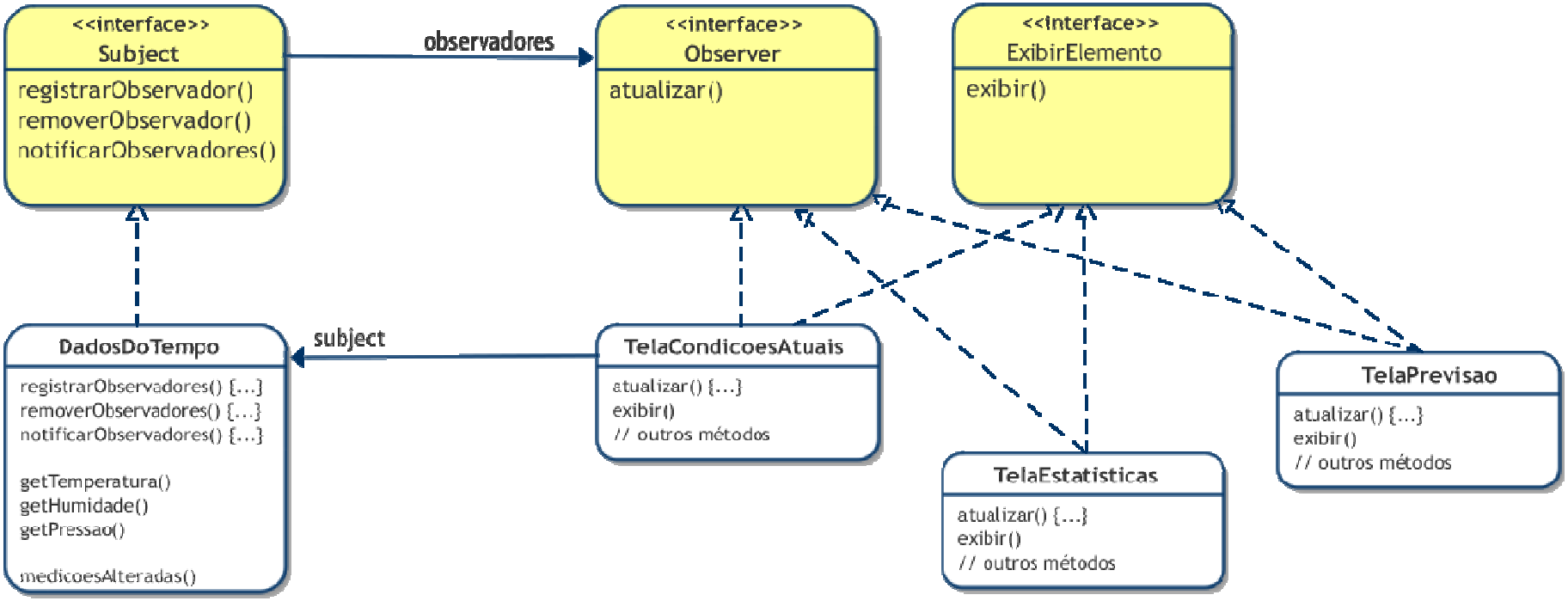
Princípio de Design

- Procure desenvolver *designs* fracamente acoplados entre objetos.
  - Permite construir sistemas OO flexíveis.
  - Com possibilidade de sucesso nas mudanças.
  - Porque eles minimizam a interdependência entre objetos.

Como aplicar?

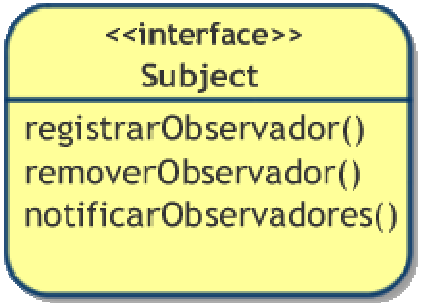


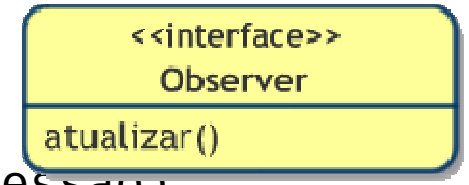
Projetando a Estação Meteorológica

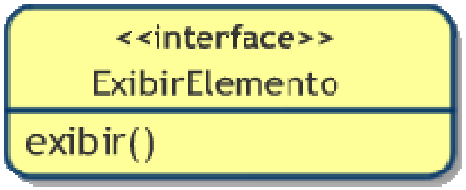


Implementação

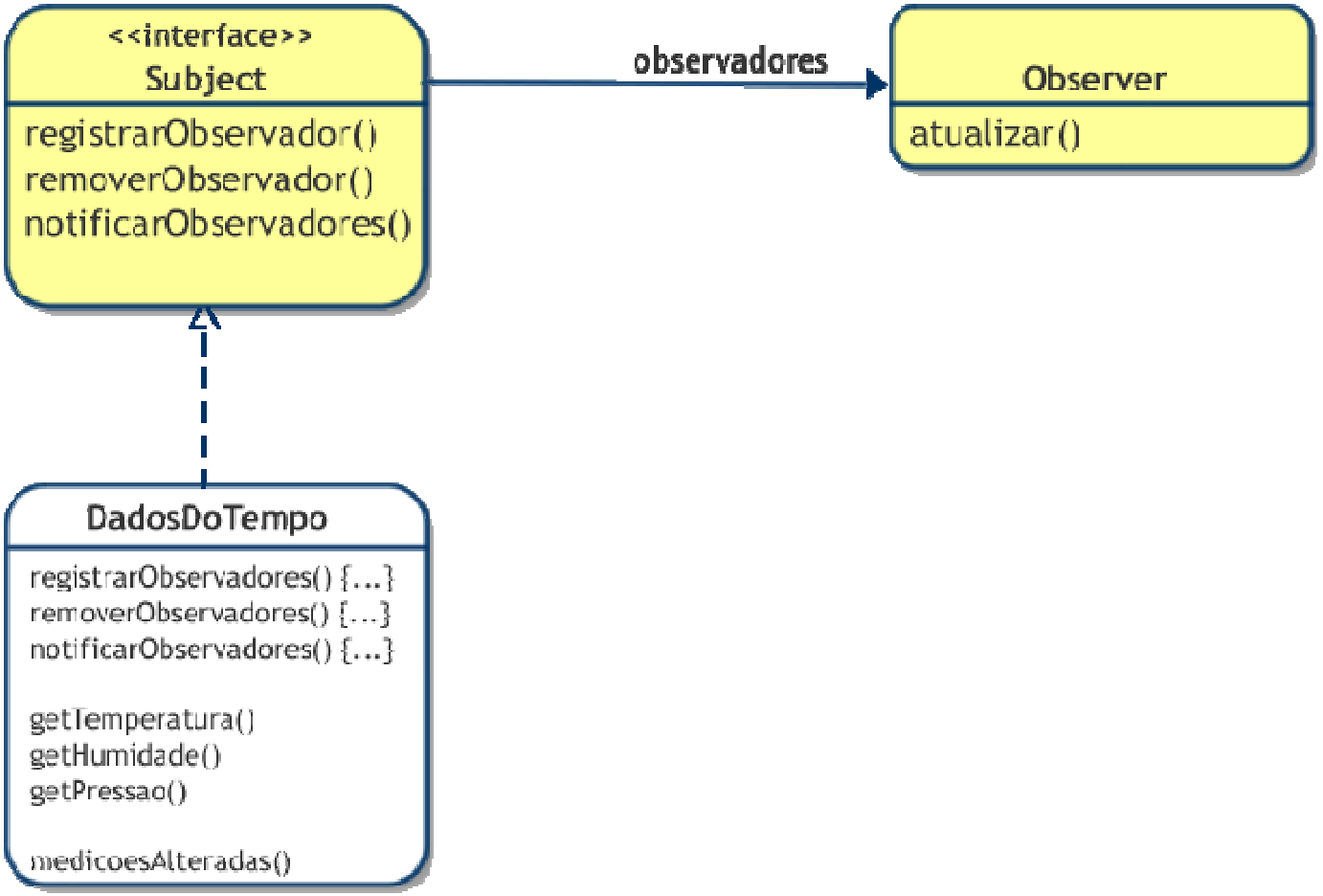
```
public interface Subject {
  void registrarObservadores(Observer o);
  void removerObservadores(Observer o);
  void notificarObservadores();
}

public interface Observer {
  public void atualizar(
      float temperatura, float umidade, float pressao);
}

public interface ExibirElemento {
  public void exibir();
}
```







Objetos DadosDoTempo



DadosDoTempo.java

```
import java.util.ArrayList;

public class DadosDoTempo implements Subject {
  /**
   * Esta lista guarda os observadores
   */
  public ArrayList<Observer> observadores;
  /**
   * Temperatura em graus celsius
   */
  public float temperatura;
  /**
   * Umidade em percentuais
   */
  public float umidade;
  /**
   * Pressao em joules
   */
  public float pressao;

  /**
   * Construtor Inicializa a lista de observadores
   */
  public DadosDoTempo() {
    observadores = new ArrayList<Observer>();
  }
```

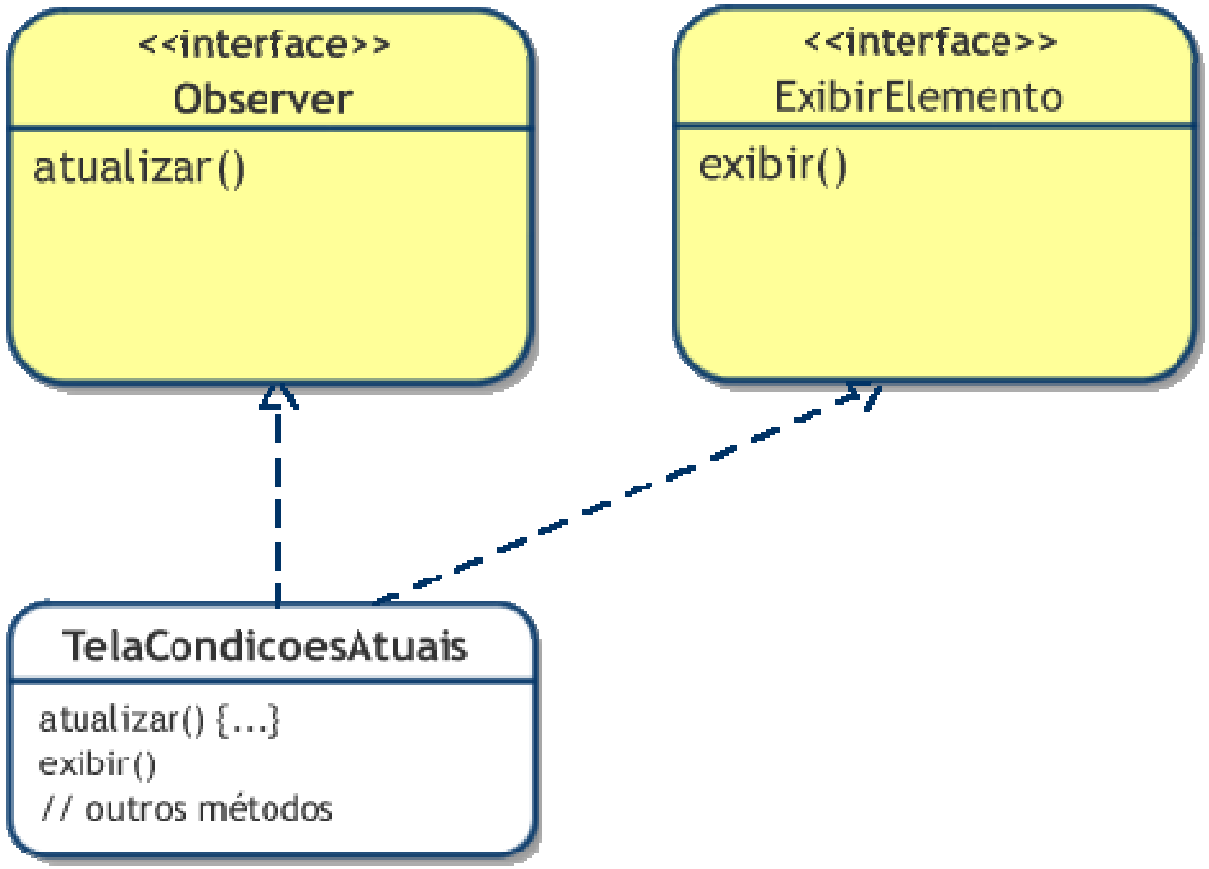
DadosDoTempo.java

```
/**
    * Registra o observador através da inclusão dele na lista
    */
  public void registrarObservadores(Observer o) {
    observadores.add(o);
  }

  /**
   * Quando um observador não desejar mais recever notificações, usamos este
   * método para removê-lo da lista de observadores
   */
  public void removerObservadores(Observer o) {
    int i = observadores.indexOf(o);
    if (i >= 0) {
      observadores.remove(i);
    }
  }

  /**
   * Notifica os observadores que uma mudança ocorreu
   */
  public void notificarObservadores() {
    for (int i = 0; i < observadores.size(); i++) {
      Observer observador = (Observer) observadores.get(i);
      observador.atualizar(temperatura, umidade, pressao);
    }
  }
```

DadosDoTempo.java

```
  /**
   * Método usado quando ocorrer alterações nas medidas do tempo
   *
   */
  public void medicoesAlteradas() {
    notificarObservadores();
  }
  /**
   * Método que será usado pela Estacao Meteorológica para alterar
   * o estado da temperatura, da umidade e da pressao
   * @param temperatura Valor em graus Celsius
   * @param umidade Valor em percentuais
   * @param pressao Valor em joules
   */
  public void setMedicoes(float temperatura, float umidade, float pressao) {
    this.temperatura = temperatura;
    this.umidade = umidade;
    this.pressao = pressao;

    medicoesAlteradas();
  }
}
```

TelaCondicoesAtuais



TelaCondicoesAtuais.java

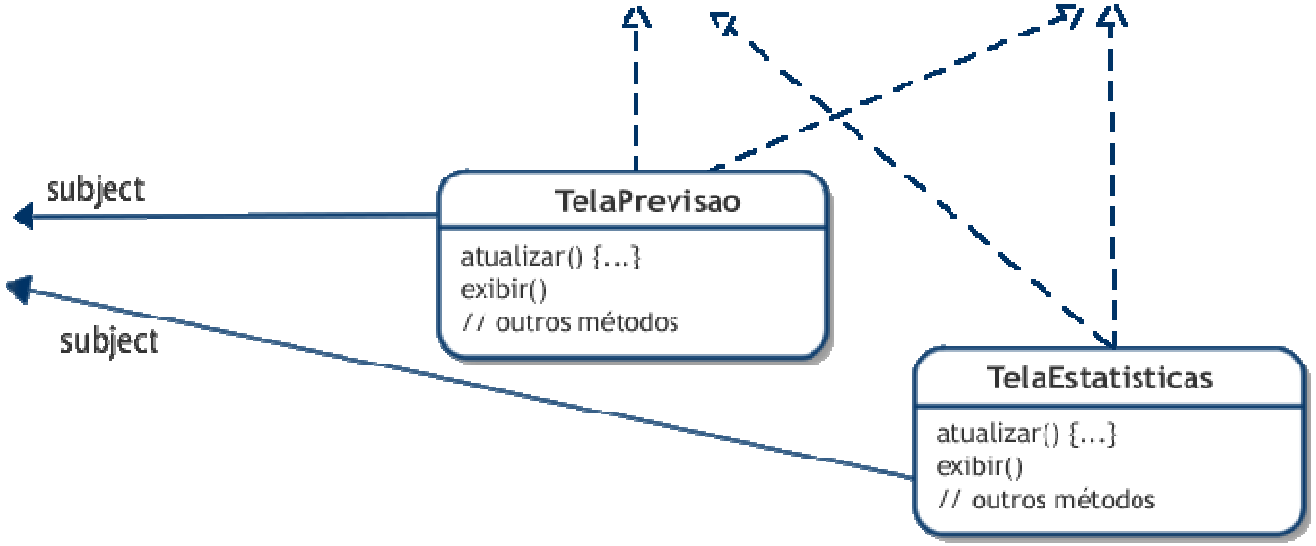
```
/**
 * Esta classe implementa o Padrão Observer, mais especificamente o participante
 * Observer.
 * @author eduardo
 *
 */
public class TelaCondicoesAtuais implements Observer, ExibirElemento {
  /**
   * Temperatura em graus celsius
   */
  private float temperatura;
  /**
   * umidade em percentuais
   */
  private float umidade;
  /**
   * Este é objeto Subject ao qual este Observer estará dependente
   */
  private Subject dadosDoTempos;
  /**
   * Construtor que inicializa o Subject e registra este Observador junto
   * ao Subject
   * @param dadosDoTempos Subject ao qual este observador está ligado
   */
  public TelaCondicoesAtuais(Subject dadosDoTempos) {
    this.dadosDoTempos = dadosDoTempos;
    this.dadosDoTempos.registrarObservadores(this);
  }
```

TelaCondicoesAtuais.java

```
 /**
   * Atualiza os valores de temperatura, umidade e pressao.
   * Este método é chamado no Subject (DadosDoTempo.java)
   * Após a atualização de valores o método exibir() é chamado
   */
  public void atualizar(float temperatura, float umidade, float pressao) {
    this.temperatura = temperatura;
    this.umidade = umidade;
    this.pressao = pressao;
    exibir();
  }
  /**
   * <p>Este método é chamado sempre que o Observador receber novos valores
   * do Subject.</p>
   */
  public void exibir() {
    System.out.println("\n======== TELA CONDIÇÕES ATUAIS =========");

    System.out.println("Condições atuais: " + temperatura + "ºC e  " + umidade
        + "% de umidade");
  }
}
```

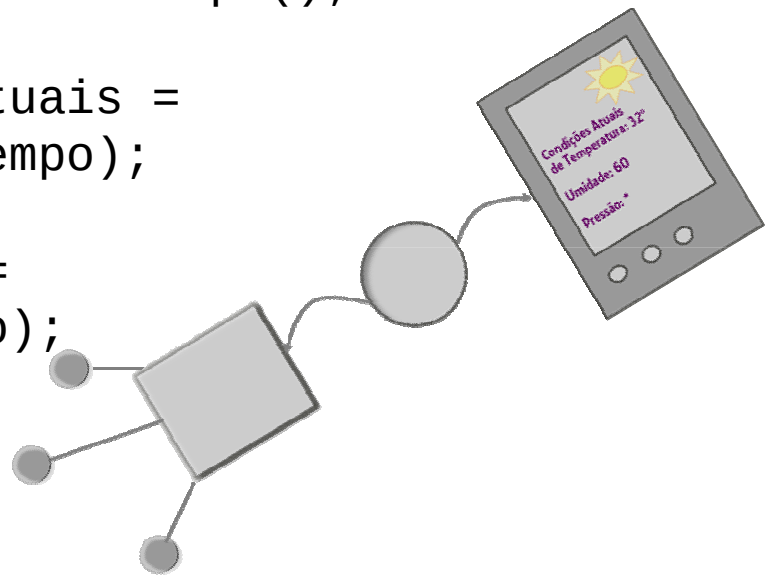
Como seria o código para as outras Telas?

- TelaPrevisao e TelaEstatisticas



Testando com uma mini estação

```
public class EstacaoMeteorologica {
  public static void main(String[] args) {
    DadosDoTempo dadosDoTempo = new DadosDoTempo();

    TelaCondicoesAtuais telaCondicoesAtuais =
      new TelaCondicoesAtuais(dadosDoTempo);

    TelaEstatisticas telaEstatisticas =
      new TelaEstatisticas(dadosDoTempo);

    TelaPrevisao telaPrevisao =
      new TelaPrevisao(dadosDoTempo);

    dadosDoTempo.setMedicoes(30, 65, 30.4F);
    dadosDoTempo.setMedicoes(34, 70, 29.2F);
    dadosDoTempo.setMedicoes(28, 98, 29.2F);
  }
}
```



Consequências

- Permite variar subjects e observadores de forma independente.
  - Desde que implementam a mesma interface.
- Permite acrescentar observadores sem modificar o Subject ou outros observadores.
- Acoplamento abstrato entre Subject e Observer.
  - Tudo o que o Subject sabe é que ele tem uma lista de observadores. O Subject não conhece a classe concreta de nenhum observador.

Consequências

- Apresenta suporte para comunicações *broadcast*.
  - A notificação do Subject é enviada automaticamente a todos os seus observadores registrados. É de responsabilidade do Observador tratar ou ignorar uma notificação recebida.
- Problemas com eficiência.
  - Uma vez que podem existir muitos observadores para um mesmo Subject, uma simples mudança no estado do Subject pode provocar uma cascata de atualizações nos observadores.

QUESTÃO

- O que é delegação?
- Como podemos usar a composição para implementar delegação?

Padrão

Composite

Motivação

Dois Restaurantes diferentes

Unificação de empresas

Cada uma possui um cardápio diferente

Uma especializada em Cafés da Manhã
- A Panquecaria

Outra em Almoços
- O Restaurante

2 Coleções de Objetos

Menu Restaurante

Sanduíche Vegetariano 2,99
Alface e Pão Integral

Sopa do dia 2,80
Tigela de sopa com torradas

Cachorro Quente 1,50
Salsicha com molho e queijo

Legumes Cozidos 2,50
Mistura de legumes

MENU PANQUECARIA

Desjejum de Panqueca 2,99
Panquecas com ovos mexidos

Desjejum Tradicional 2,99
Panquecas com ovos fritos e salsicha

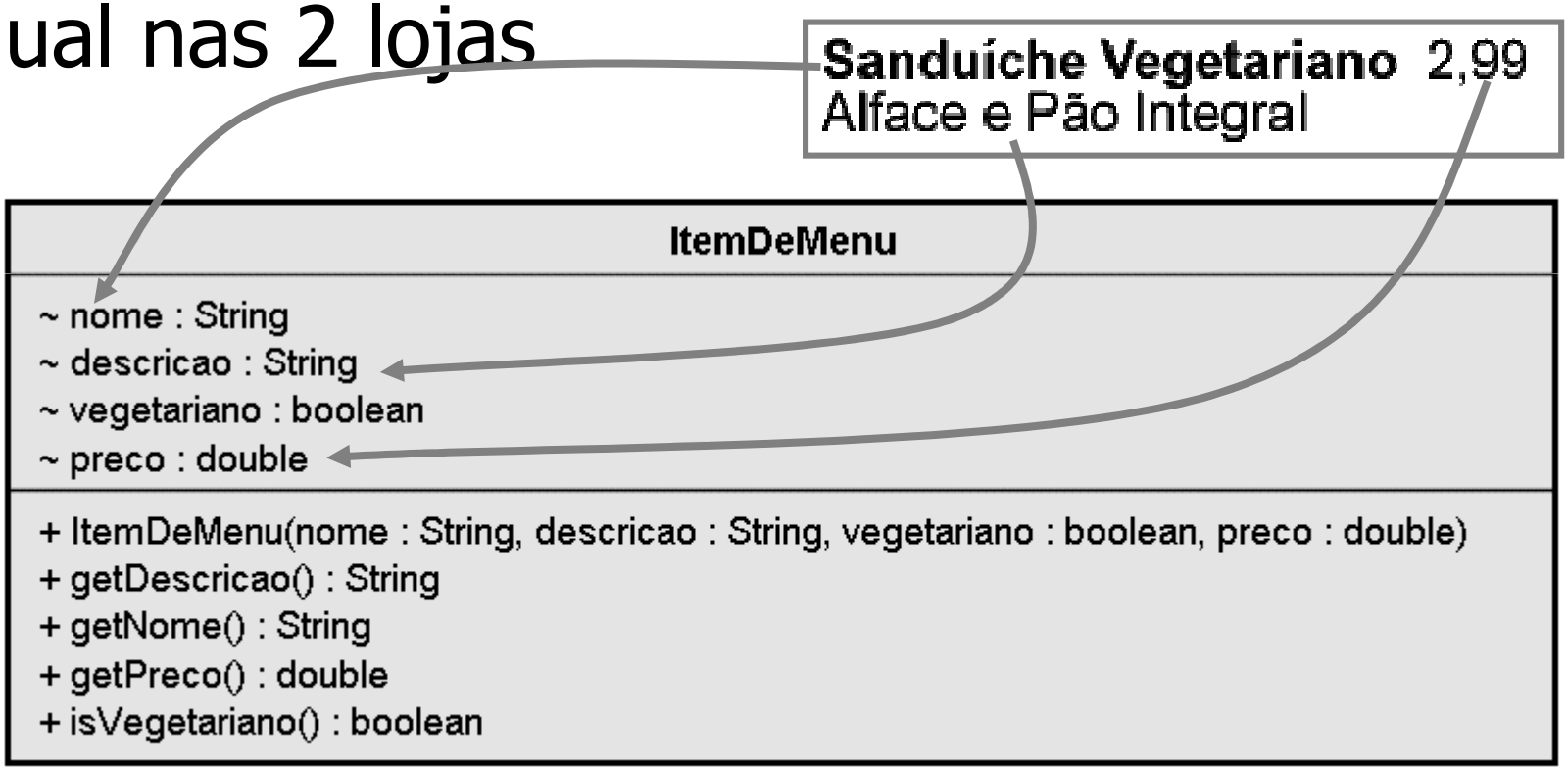
Panquecas Doces 3,49
Panquecas com geléia de amora

Waffles 3,59
Waffles com geléia de mocotó

ItemDeMenu

O Item de Menu

- A classe que representa cada item de menu é igual nas 2 lojas



Os Menus Semelhantes

- Os menus são semelhantes

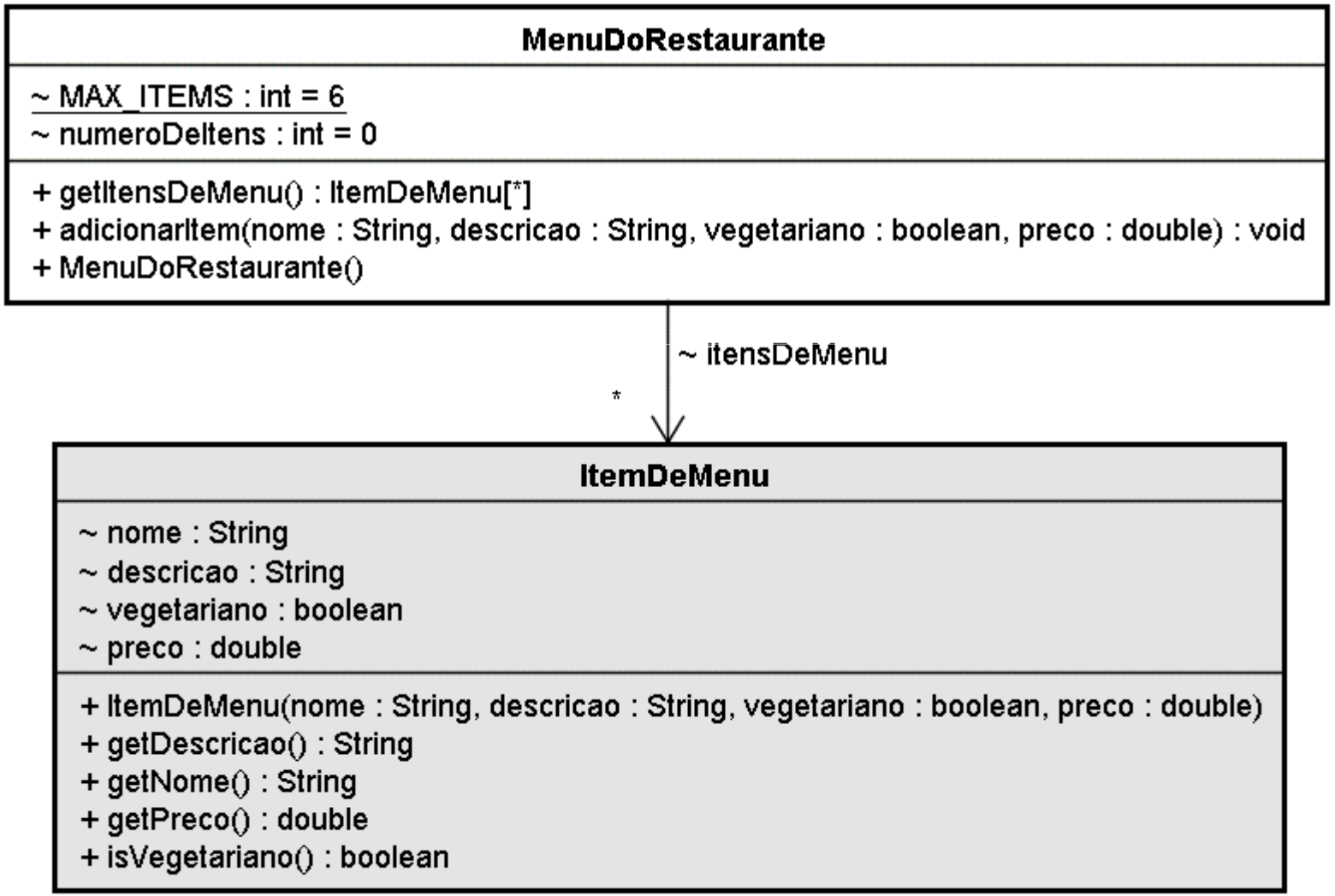
ArrayList<ItemDeMenu>

ItemDeMenu[]

- Diferença
  - O armazenamento dos objetos ItemDeMenu

O Menu do Restaurante

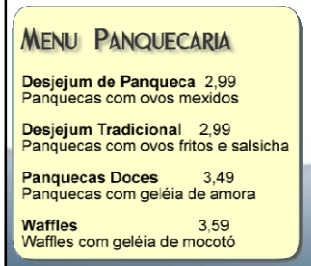
ItemDeMenu[]



MenuDoRestaurante.java

```
public class MenuDoRestaurante {

  static final int MAX_ITEMS = 6;
  int numeroDeItens = 0;
  ItemDeMenu[] itensDeMenu;

  public ItemDeMenu[] getItensDeMenu() {
    return itensDeMenu;
  }

  public void adicionarItem(
      String nome, String descricao, boolean vegetariano, double preco) {
    ItemDeMenu itemDeMenu =
      new ItemDeMenu(nome, descricao, vegetariano, preco);
    if (numeroDeItens >= MAX_ITEMS) {
      System.out.println("Menu está cheio");
    } else {
      itensDeMenu[numeroDeItens] = itemDeMenu;
      numeroDeItens++;
    }

  }

  public MenuDoRestaurante() {
    itensDeMenu = new ItemDeMenu[MAX_ITEMS];

    adicionarItem("Canja", "Canja", false, 3.99);
    adicionarItem("Waffles", "Waffles", true, 3.59);
  }

}
```

ItemDeMenu[]



ArrayList<ItemDeMenu>

O Menu da Panquecaria

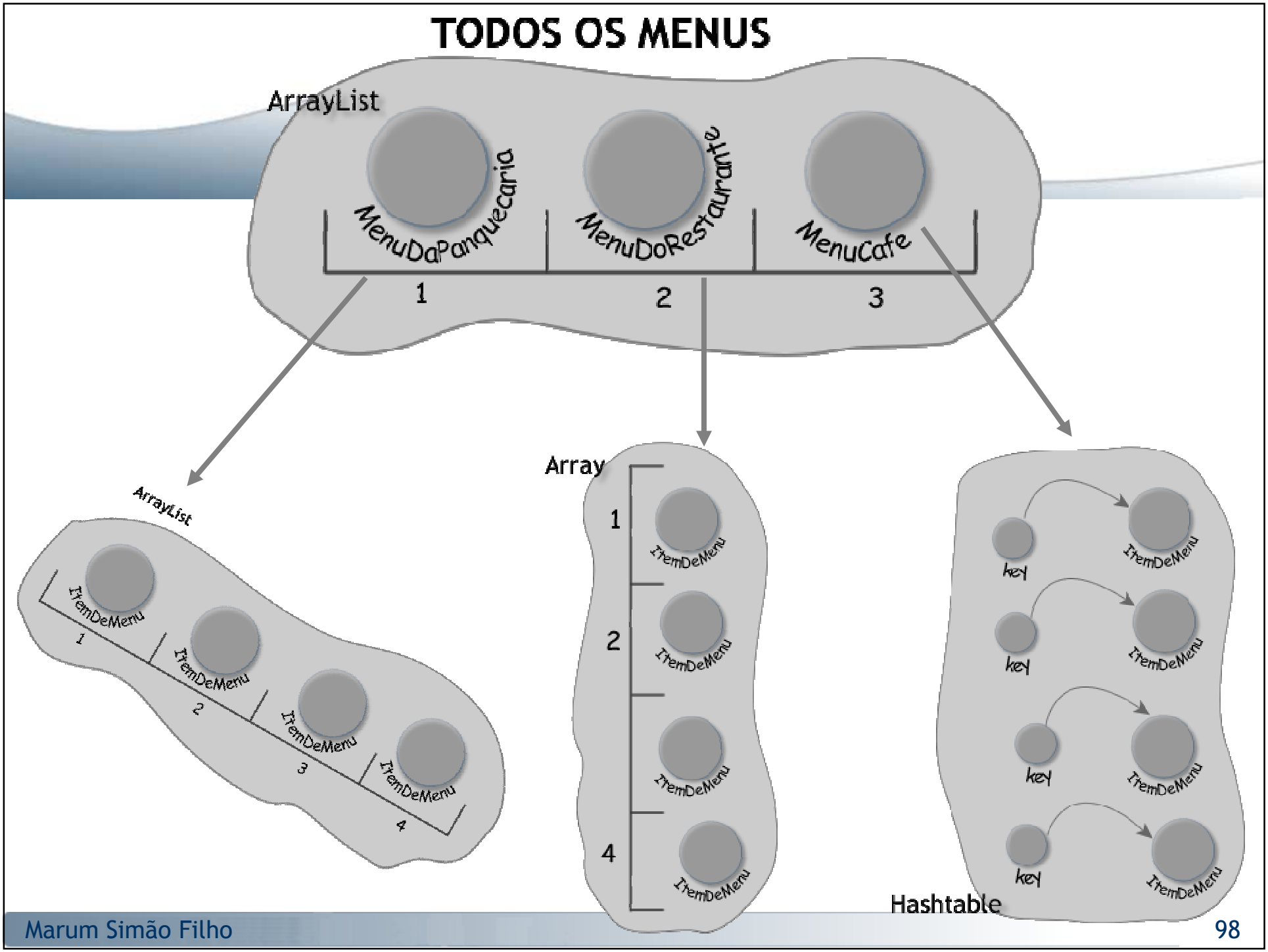
MenuDaPanquecaria
- itensDeMenu : ArrayList<ItemDeMenu>
+ getItensDeMenu() : ArrayList<ItemDeMenu> + adicionarItem(nome : String, descricao : String, vegetariano : boolean, preco : double) : void + MenuDaPanquecaria()

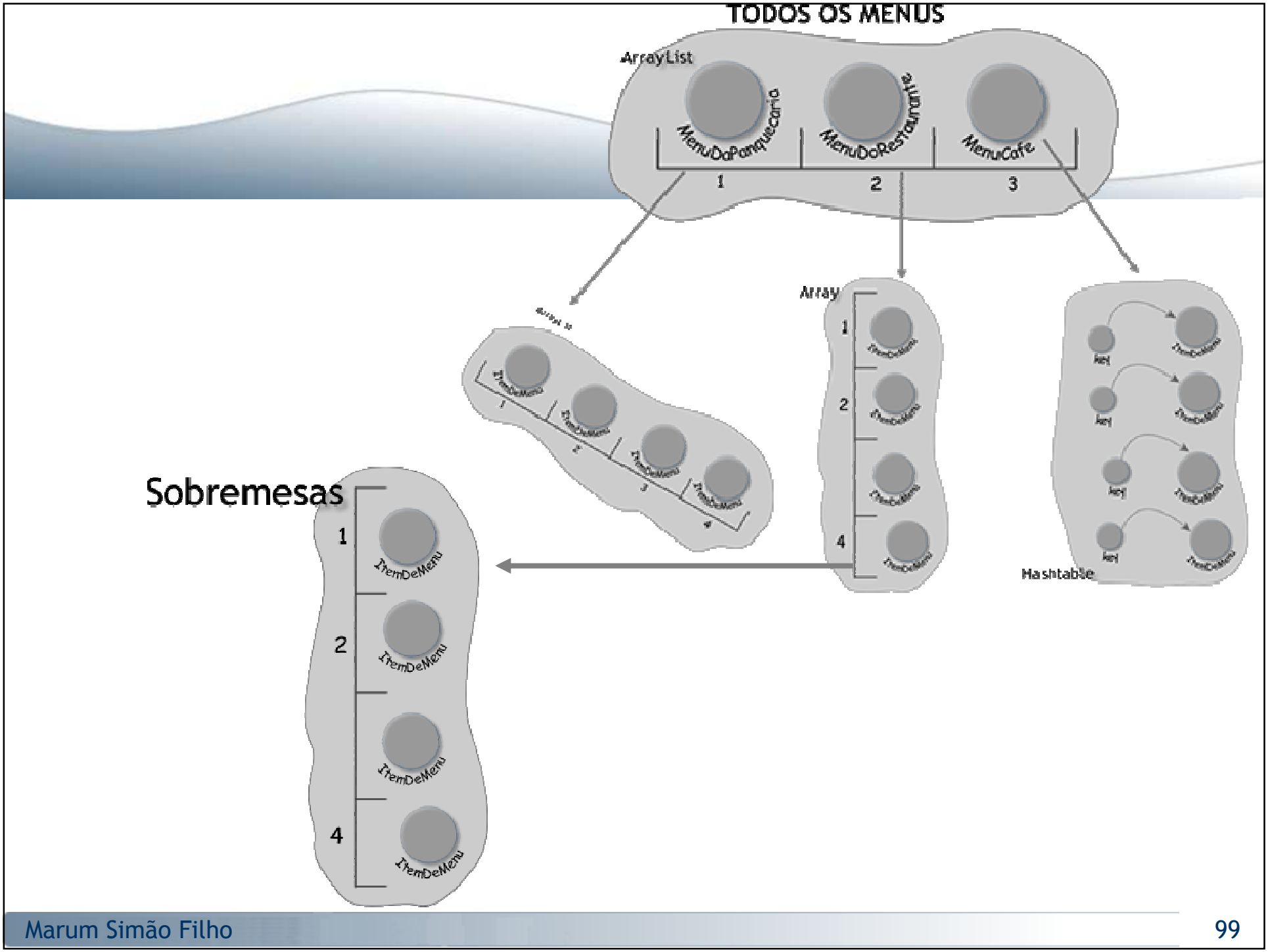
MenuDaPanquecaria.java

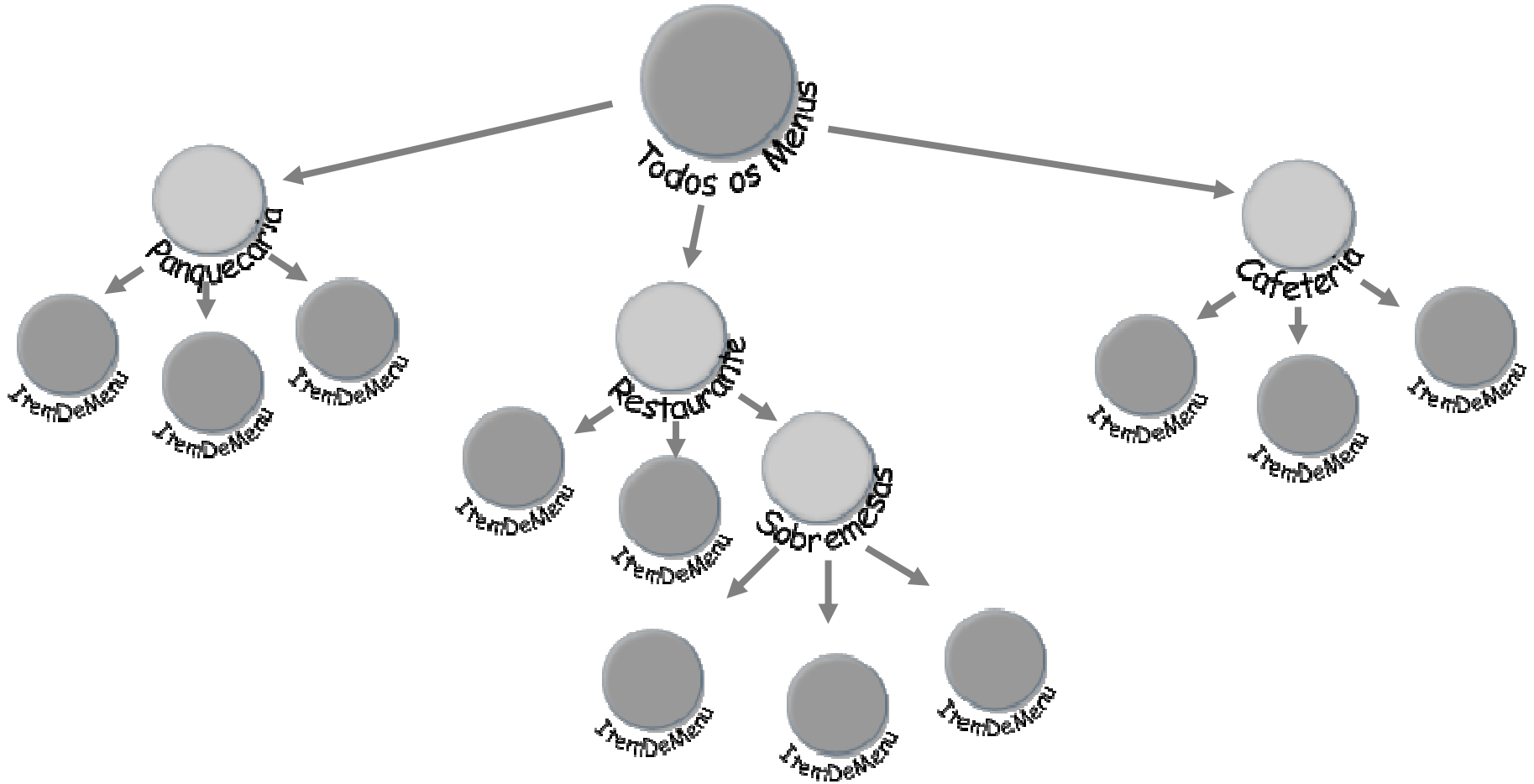
```
public class MenuDaPanquecaria {

  ArrayList itensDeMenu;

  public ArrayList getItensDeMenu() {
    return itensDeMenu;
  }

  public void adicionarItem(
      String nome, String descricao, boolean vegetariano, double preco) {
    ItemDeMenu itemDeMenu = new ItemDeMenu(nome, descricao, vegetariano, preco);
    itensDeMenu.add(itemDeMenu);
  }

  public MenuDaPanquecaria() {
    itensDeMenu = new ArrayList();
    adicionarItem("Panqueca Café da Manhã", "Panqueca com ovos", true, 2.99);
    adicionarItem("Waffles", "Waffles", true, 3.59);
  }

}
```

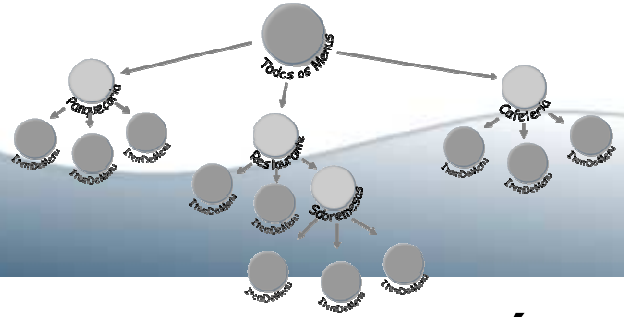
Composite

- Um padrão que permite construir estruturas de objetos na forma de **árvores**.
- A árvore
  - Objetos Individuais
  - Composições de objetos.


TODOS OS MENUS
ArrayList
MenuDaPanquecaria
MenuDoRestaurante
MenuCafe
1
2
3
ArrayList
ItemDeMenu
1
ItemDeMenu
2
ItemDeMenu
3
ItemDeMenu
4
Array
1
ItemDeMenu
2
ItemDeMenu
ItemDeMenu
4
ItemDeMenu
key
ItemDeMenu
key
ItemDeMenu
key
ItemDeMenu
key
ItemDeMenu
Hashtable


TODOS OS MENUS
ArrayList
MenuDoParquecario
MenuDoRestaurante
MenuCafe
1
2
3
Array
1
2
4
Hashtable
Sobremesas
1
2
4
ItemDeMenu


Todos os Menus
Panquecaria
Restaurante
Cafeteria
Sobremesas
ItemDeMenu
ItemDeMenu
ItemDeMenu
ItemDeMenu
ItemDeMenu
ItemDeMenu
ItemDeMenu
ItemDeMenu
ItemDeMenu
ItemDeMenu

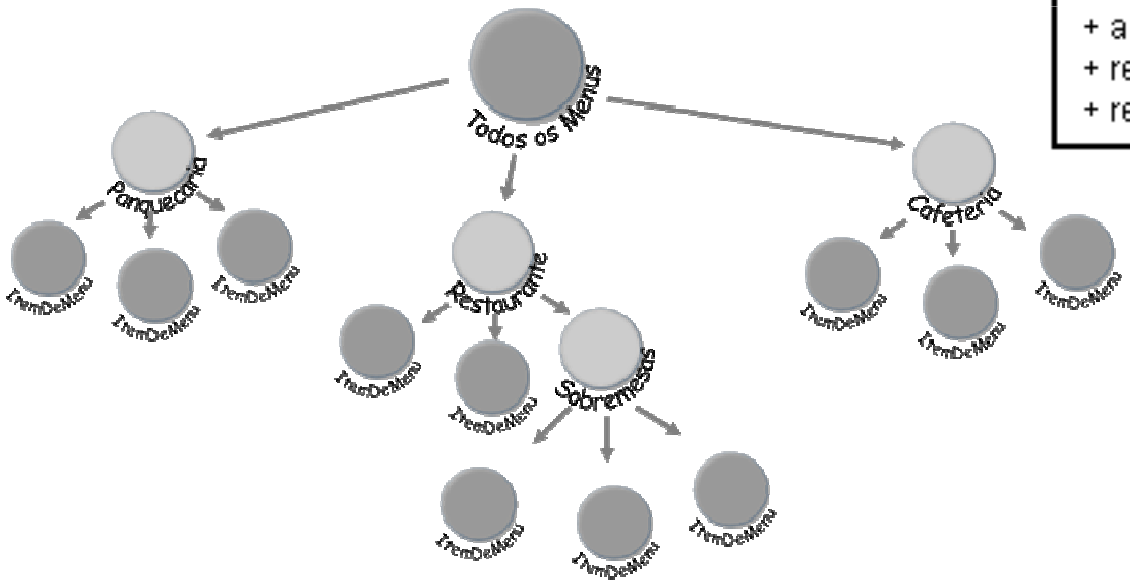


Do que precisaremos?

- Estrutura de árvore
  - Menus
  - Submenus
  - Itens de Menu
- Manter viável iterar sobre os itens de cada menu.
- Ser capaz de percorrer os itens de maneira flexível
  - Submenu
  - Item de Menu
- Acesso individual a todos os elementos da árvore.

O que é importante?

- Uma interface comum para todos os elementos da árvore: **o Componente**



Componente
+ operacao() : void + adicionar(componente : Componente) : void + remover(componente : Componente) : void + recuperarFilho(indice : int) : Componente

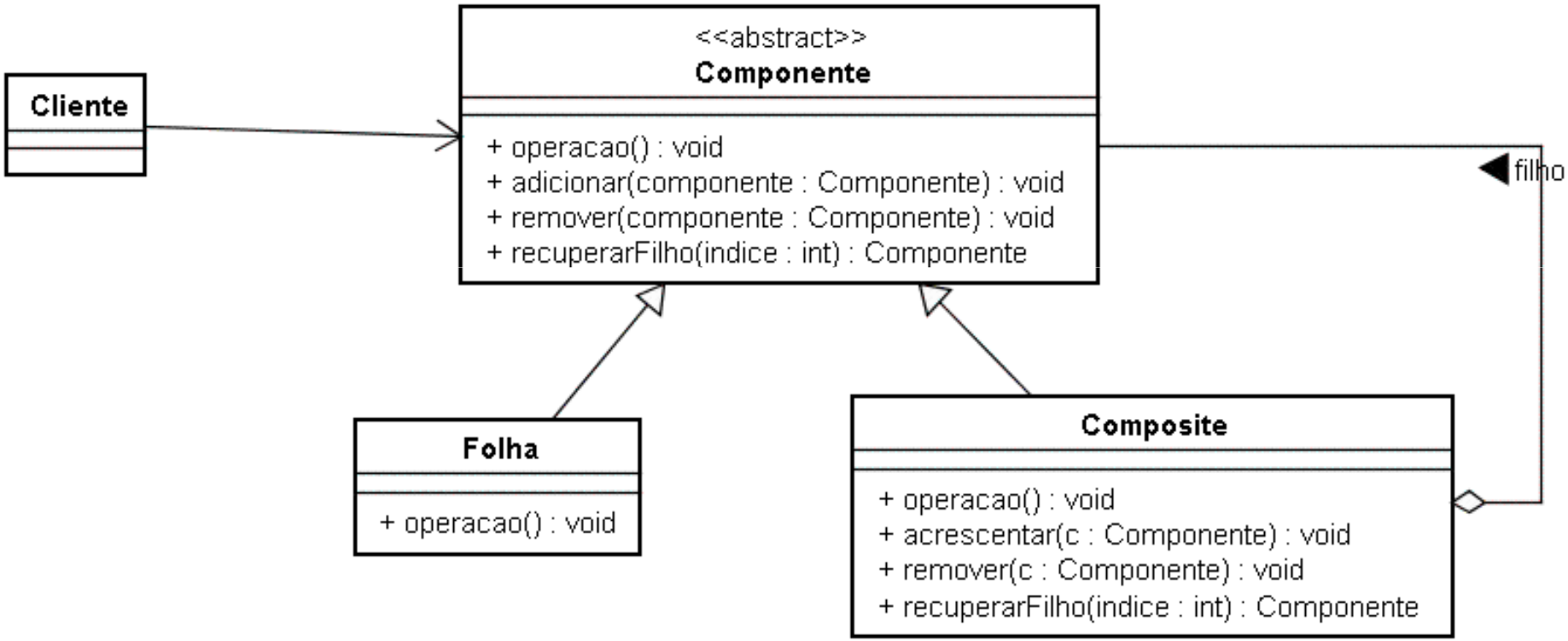
+ 1 Padrão Composite

O **Padrão Composite** permite que você componha objetos em estruturas de árvores para representarem hierarquias **parte-todo**. O Composite permite aos clientes tratarem de maneira **uniforme** objetos individuais e composições de objetos.

Elementos da solução

- Componente, composto e árvores.
- Criar uma interface comum.

Diagrama de classes



Aplicabilidade

- Representação de hierarquias **parte-todo**.
- Necessidade de **transparência** para o cliente.
  - O cliente não sabe se está lidando com um objeto individual ou composto.

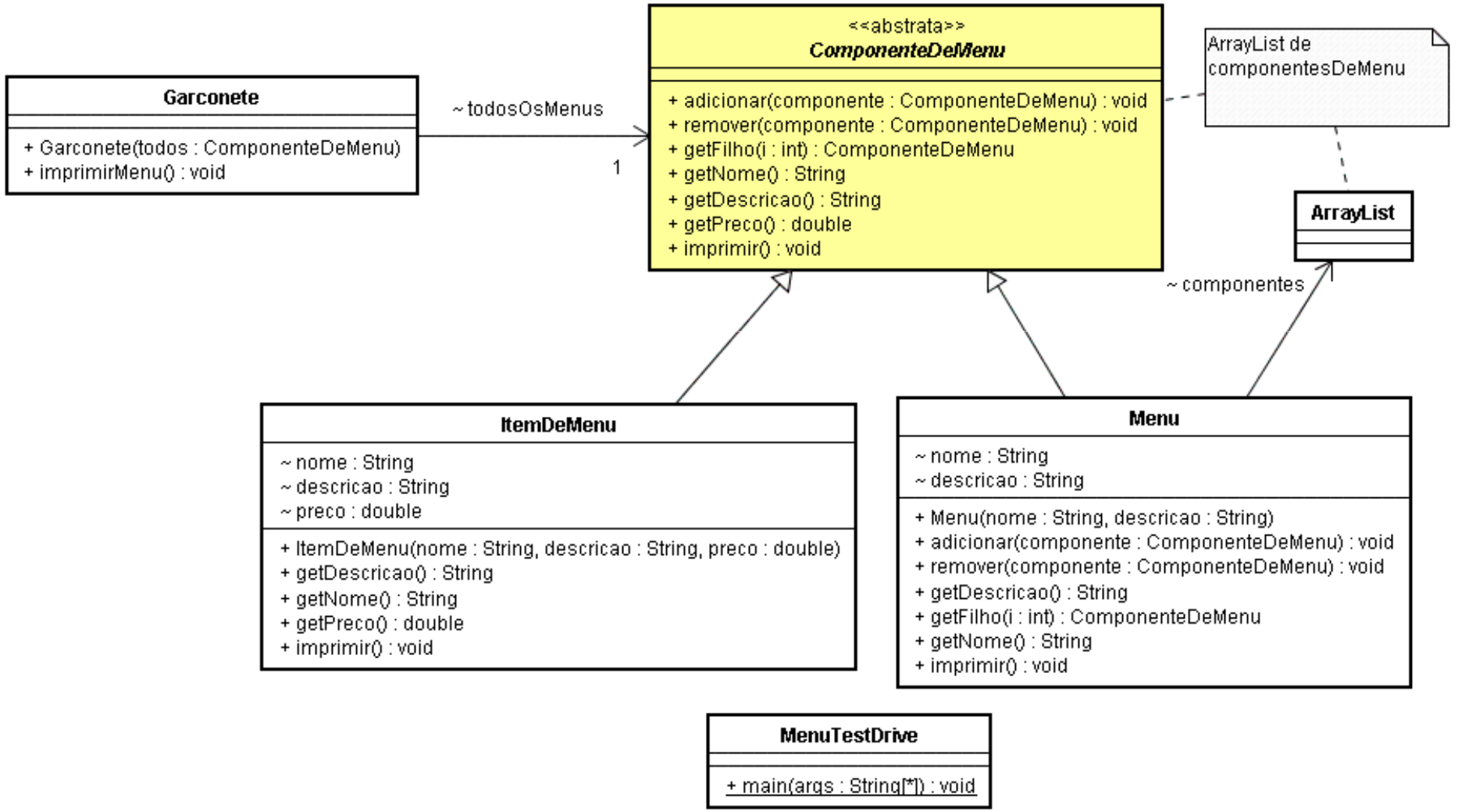
Participantes

- **Componente**
  - Declara a interface comum para os objetos na composição.
  - Implementa o comportamento padrão.
  - Declara uma interface para gerenciar os componentes-filhos.
- **Folha**
  - Representa os objetos folha na composição.
  - Define o comportamento para objetos primitivos.
- **Composite**
  - Define o comportamento para componentes que têm filhos.
  - Armazena os componentes-filho.
  - Implementa as operações relacionadas com os filhos.
- **Cliente**
  - Manipula objetos na composição através da interface do Componente.

Colaborações

- Clientes usam a interface da classe Componente para interagir com objetos na estrutura composta.
- Se o receptor de uma solicitação do cliente é uma Folha:
  - Solicitação é tratada diretamente.
- Se é um Composite:
  - Repasssa a solicitação para os filhos executando operações adicionais.

Os menus e o Composite



ComponenteDeMenu.java (Componente)

```
public abstract class ComponenteDeMenu {
    public void adicionar(ComponenteDeMenu componente) {
       throw new UnsupportedOperationException();
    }
    public void remover(ComponenteDeMenu componente) {
       throw new UnsupportedOperationException();
    }
    public ComponenteDeMenu getFilho( int i ) {
       throw new UnsupportedOperationException();  }
    public String getNome() {
       throw new UnsupportedOperationException();  }
    public String getDescricao() {
       throw new UnsupportedOperationException();  }
    public double getPreco() {
       throw new UnsupportedOperationException();  }
    public void imprimir() {
       throw new UnsupportedOperationException();  }
}
```

ItemDeMenu.java (Folha)

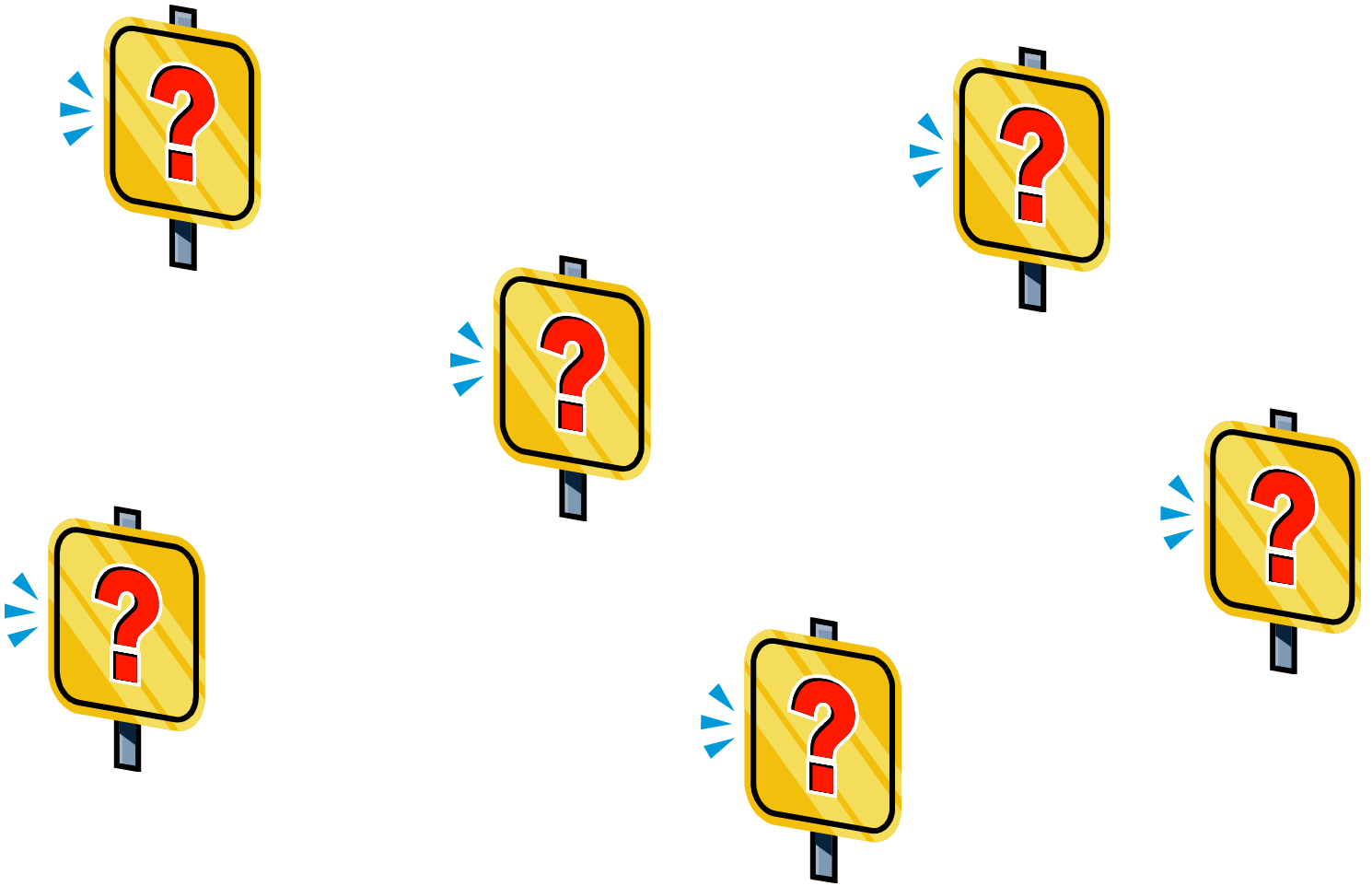
```
public class ItemDeMenu extends ComponenteDeMenu {
  String nome;
  String descricao;
  double preco;
  public ItemDeMenu(String nome, String descricao, double preco)
  {
       this.nome = nome;
       this.descricao = descricao;
       this.preco = preco;
  }
  public String getNome() { return nome; }
  public String getDescricao() { return descricao; }
  public double getPreco() { return preco; }
  public void imprimir() {
       System.out.println ( getNome() + " - " + getDescricao()
                            + " " + getPreco() );
   }
}
```

Menu.java (Composite)

```
public class Menu extends ComponenteDeMenu {
  ArrayList componentesDeMenu = new ArrayList();
  String nome;   String descricao;   double preco;
  public Menu(String nome, String descricao) {
        this.nome = nome;   this.descricao = descricao;
  }
  public void adicionar(ComponenteDeMenu componente) {
        componentesDeMenu.add( componente );
  }
  public void remover(ComponenteDeMenu componente) {
        componentesDeMenu.remove( componente );
  }
  public ComponenteDeMenu getFilho( int i ) {
        return (ComponenteDeMenu) componentesDeMenu.get(i);
  }
   public String getNome() { return nome; }
   public String getDescricao() { return descricao; }
   public void imprimir() {
     for (ComponenteDeMenu  a : componentesDeMenu ) { a.imprimir(); }
   }
```

Consequências

- Com a interface comum, alcançamos:
  - Flexibilidade
  - Transparência
- Caso contrário:
  - Seria necessário saber qual o tipo de cada classe
    - Em Java: condicionais com uso de instanceOf



Obrigado!!!

Agradecimentos:

Prof. Eduardo Mendes

Prof. Régis Simão

Faculdade 7 de Setembro