

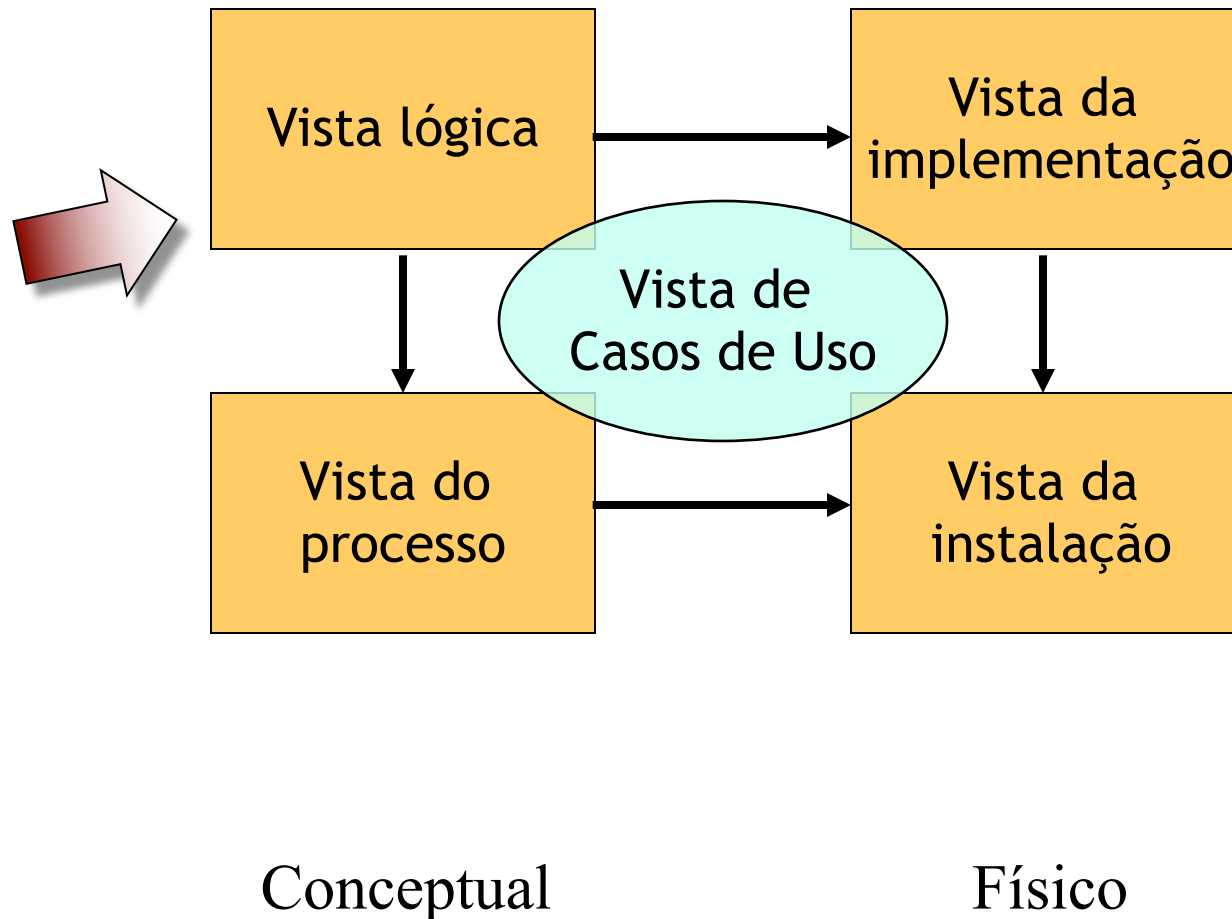


# Desenvolvimento de Sistemas Software

## Modelação Estrutural (Diagramas de Classe)



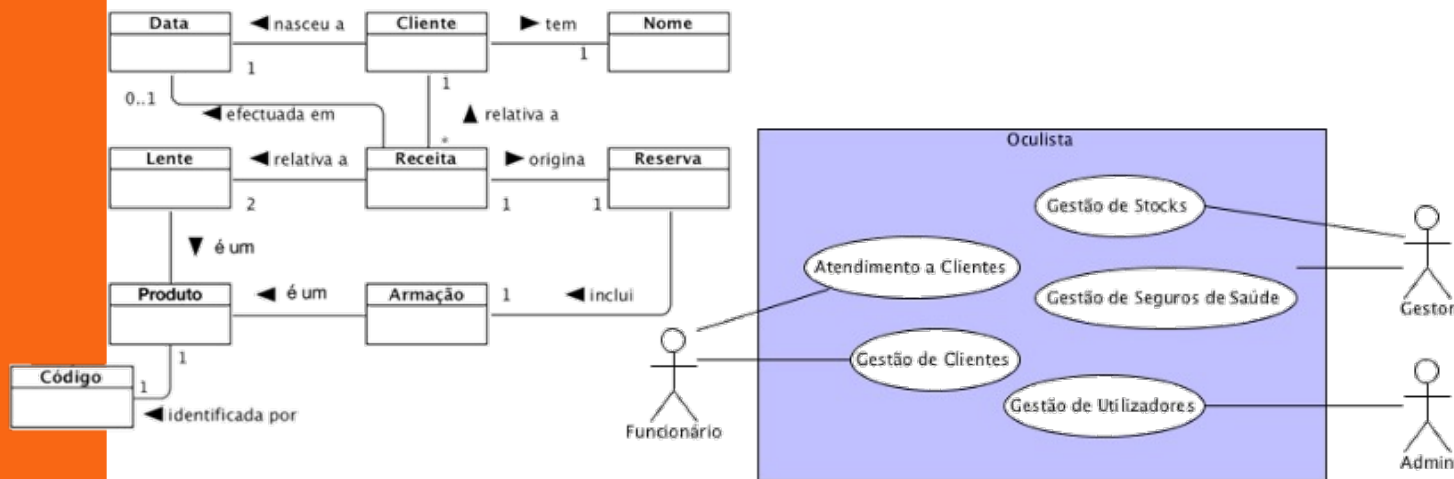
# Onde estamos...



(Kruchten, 1995)



# Em resumo...



## Use Case: Reservar armação e lentes

Descrição: Funcionário regista uma reserva de armação e lentes  
Pós-condição: Reserva fica registada

### Fluxo normal:

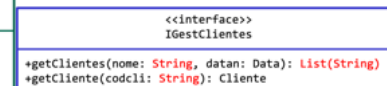
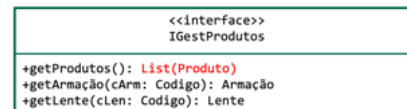
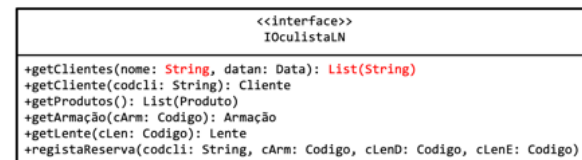
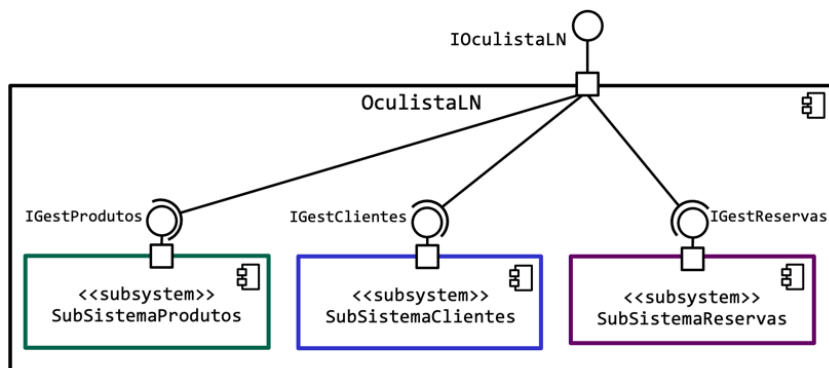
1. Funcionário indica nome e/ou data de nascimento do cliente
2. Sistema procura clientes
3. Sistema apresenta lista de clientes
4. Funcionário selecciona cliente
5. Sistema procura cliente
6. Sistema apresenta detalhes do cliente
7. Funcionário confirma cliente
8. Sistema procura produtos e apresenta lista
9. Funcionário indica Código de armação e lentes
10. Sistema procura detalhes dos produtos
11. Sistema apresenta detalhes dos produtos
12. Funcionário confirma produtos
13. Sistema regista reserva dos produtos
14. <<include>> imprimir talão

### Fluxo alternativo: [lista de clientes tem tamanho 1] (passo 3)

- 3.1. Sistema apresenta detalhes do único cliente da lista
- 3.2. regressa a 7

### Fluxo de excepção: [cliente não quer produto] (passo 12)

- 12.1. Funcionário rejeita produtos
- 12.2. Sistema termina processo





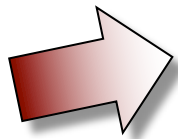
# Fases do ciclo de vida do desenvolvimento de sistemas

## Planeamento

- Decisão de avançar com o projecto
- Gestão do projecto

## Análise

- Análise do domínio do problema
- Análise de requisitos



## Concepção

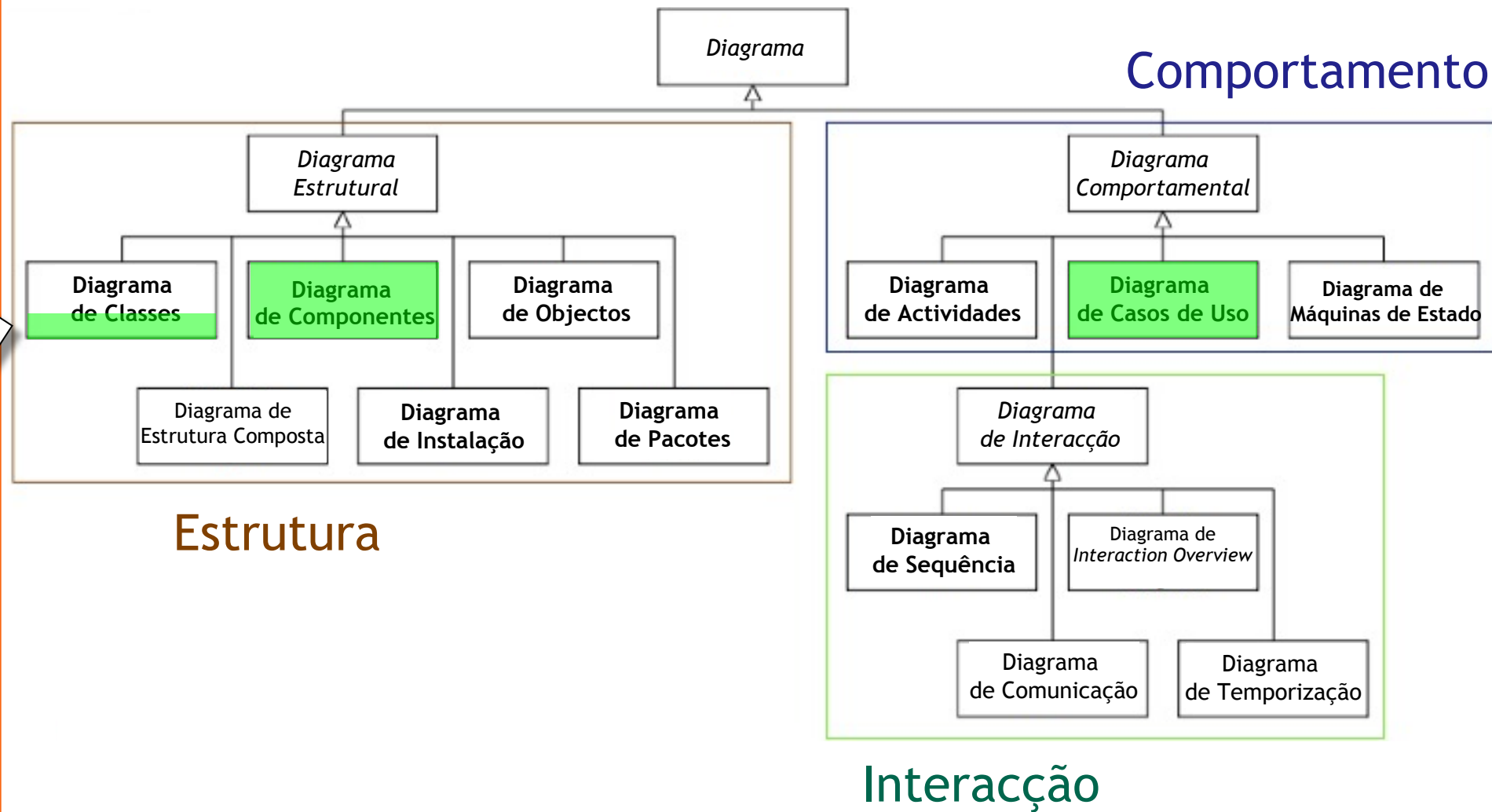
- Concepção da Arquitectura
- Concepção do Comportamento

## Implementação

- Construção
- Teste
- Instalação
- Manutenção



# Diagramas da UML 2.x

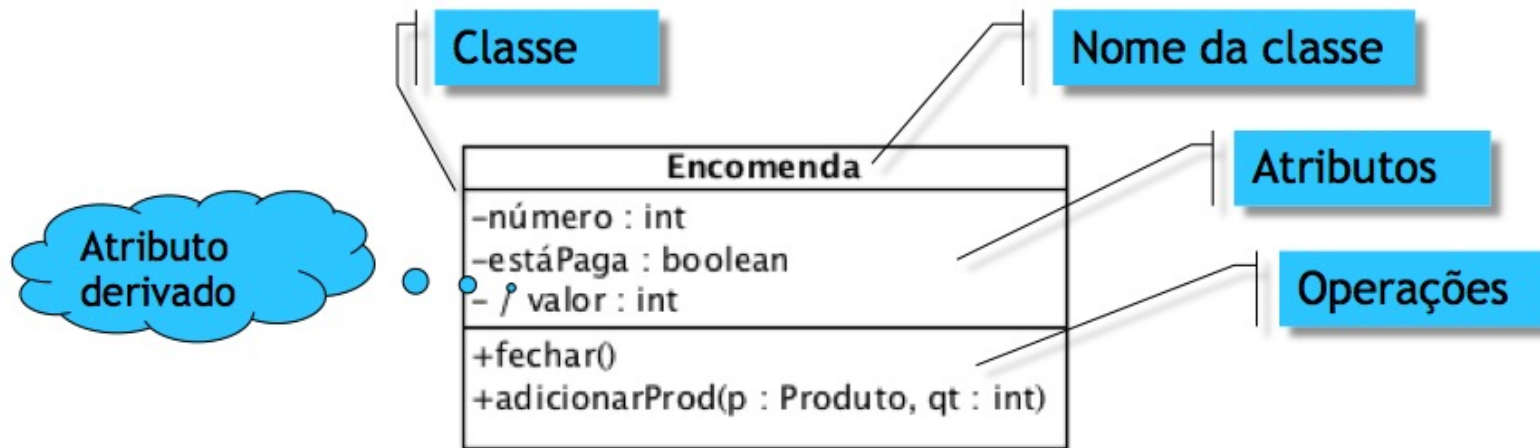




# Revisão do conceito de classe

- A noção de classe é fundamental no paradigma OO
  - tipicamente uma classe representa uma abstração de uma entidade do mundo real.
- Cada classe descreve um conjunto de objectos com a mesma estrutura e comportamento:
  - Estrutura:
    - atributos
    - relações
  - Comportamento:
    - operações
- A organização do código em classes tem dois objectivos fundamentais:
  - facilitar a reutilização — através da reutilização de classes previamente desenvolvidas em novos sistemas;
  - facilitar a manutenção — o sistema deverá ser desenvolvido de forma a que a alteração de uma classe tenha o menor impacto possível no resto do sistema.

# Representação de classes em UML



- Compartimentos pré-definidos
  - Nome da classe – começa com maiúsculas / substantivo
  - Atributos (de instância) – representam propriedades das instâncias desta classe / começam com minúsculas / substantivos
  - Operações (de instância) – representam serviços que podem ser pedidos a instâncias da classe / começam com minúsculas / verbos
- Compartimentos podem ser omitidos – isso não significa que não exista lá informação!



# Níveis de modelação

- Podemos considerar 3 níveis de modelação:
  - Conceptual
  - Especificação (vista lógica)
  - Implementação

- **Nível Conceptual**

- Representação dos conceitos no domínio de análise
- Não corresponde necessariamente a um mapeamento directo para a implementação
- Cf. Modelo de Domínio

(GESTOR DE)  
HORARIO

AULA T  
duracao

AULA TP  
duracao

AULA ?  
duracao

DOCENTE

SALA

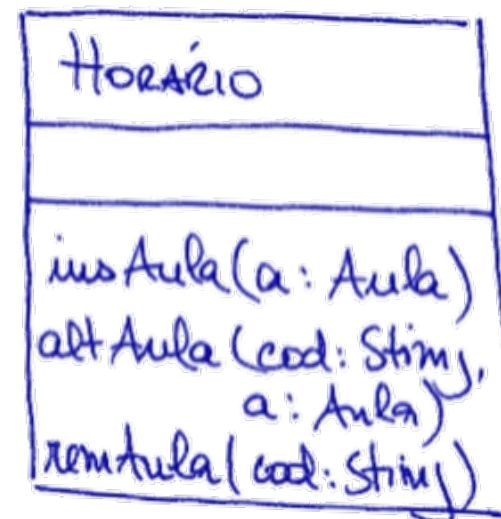
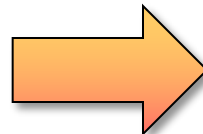
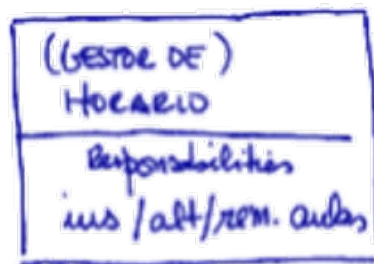
ALTERACAO

CURSO



# Níveis de modelação

- Nível de especificação
  - Definição das interfaces (API's)
  - Identificar responsabilidades e modelá-las com operações/atributos
- Exemplo:

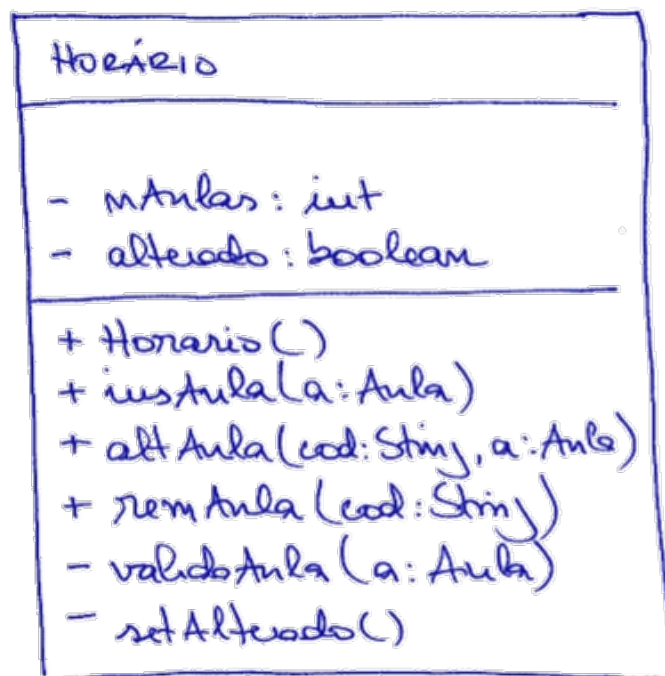




# Níveis de modelação

- Nível de implementação
  - Definição concreta das classes a implementar - geração de código
  - Definição dos relacionamentos estruturais entre as entidades

- Exemplo:





# Visibilidade de atributos e operações

- O nível de visibilidade (acesso) que se pretende para cada atributo/operação é representado com as seguintes anotações:
  - privado — só acessível ao objecto a que pertence (cf. encapsulamento)
  - # protegido — acessível a instâncias das sub-classes (atenção: em Java fica também acessível a instâncias de classes do mesmo *package*!)
  - pacote/*package* — acessível a instâncias de classes do mesmo *package* (nível de acesso por omissão)
  - + público — acessível a todos os objectos no sistema (que conheçam o objecto a que o atributo/operação pertence!)



# Declaração de atributos / operações

Só o nome é obrigatório!

- Atributos

«*esteréotipo*» *visibilidade* / nome : *tipo* [*multiplicidade*] = valorInic {propriedades}

- Exemplos

morada

- morada= “Braga” {addedBy=“jfc”, date=“18/11/2011”}
- morada: String [1..2] {leaf, addOnly, addedBy=“jfc”}

Propriedades comuns:

changeability:

changeable - pode ser alterado (o *default*)

frozen - não pode ser alterado (**final** em Java)

addOnly - para multiplicidades > 1 (só adicionar)

leaf - não pode ser redefinido

ordered - para multiplicidades > 1



# Declaração de atributos / operações

## • Operações

Obrigatório!

in | out | inout | return

«*esteréotipo*» *visibilidade* nome (direção nomeParam : tipo = valorOmiss) : *tipo*

{propriedades}

## • Exemplos

setNome

+ setNome(nome = "SCX") {abstract}

+ getNome() : String {isQuery, risco = baixo}

# getNome(out nome) {isQuery}

«create» + Pessoa()

por omissão é "in"

in - parâmetro de entrada  
out - parâmetro de saída  
inout - parâmetro de entrada/saída  
return - operação retorna o parâmetro como um dos seus valores de retorno

Propriedades comuns:

abstract - operação abstrata

leaf - não pode ser redefinido

isQuery - não altera o estado do objecto



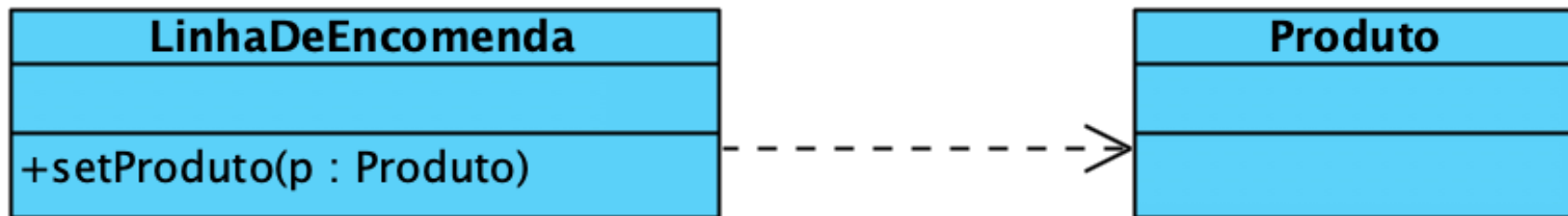
# Relações entre classes

- Três tipos de relações possíveis entre as classes:
  - **Dependência**  
indica que uma classe depende de outra
  - **Associação**  
indica que existe algum tipo de ligação entre objectos das duas classes
  - **Generalização/Especialização**  
relação entre classe mais geral e classe mais específica



# Relações entre classes - Dependência

- Notação:



- Indica que a definição de uma classe está dependente da definição de outra
- Utiliza-se normalmente para mostrar que instâncias da origem utilizam, de alguma forma, instâncias do destino (por exemplo: um parâmetro de um método)
- Uma alteração no destino (quem é usado) pode alterar a origem (quem usa)
- Diminuir o número de dependências deve ser um objectivo.

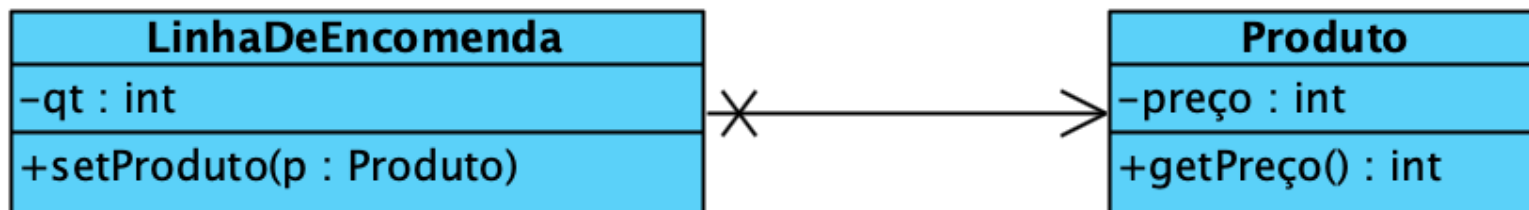


# Relações entre classes - Associação

- Código:
 

```
public class LinhaEncomenda {
    private int qt;
    private Produto p;

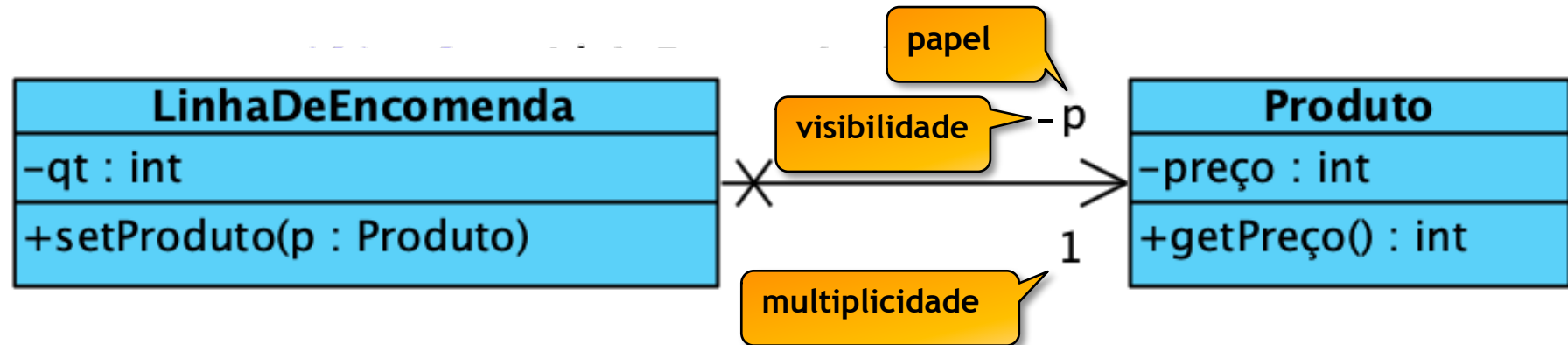
    public void setProduto(Produto p) { ...3 lines }
}
```
- Indica que objectos de uma classe estão ligados a objectos de outra classe – define uma associação entre os objectos
- Indicação de navegabilidade
  - Por omissão navegação é bidireccional (cf. diagramas E-R)
  - pode indicar-se explicitamente o sentido da navegabilidade.







# Relações entre classes - Associação



- Três decorações possíveis:
  - nome** — descreve a natureza da relação (pode ter direcção - cf. Modelos de Domínio)
  - papeis** — indica o papel que cada classe desempenha na relação definida pela associação (usualmente utilizado como alternativa ao nome)
  - multiplicidade** — quantos objectos participam na relação:
    - $*$  — zero ou mais objectos
    - $n$  —  $n$  objectos ( $n \geq 1$ )
    - $n..m$  — entre  $n$  e  $m$  objectos ( $n < m$ )

## Casos particulares:

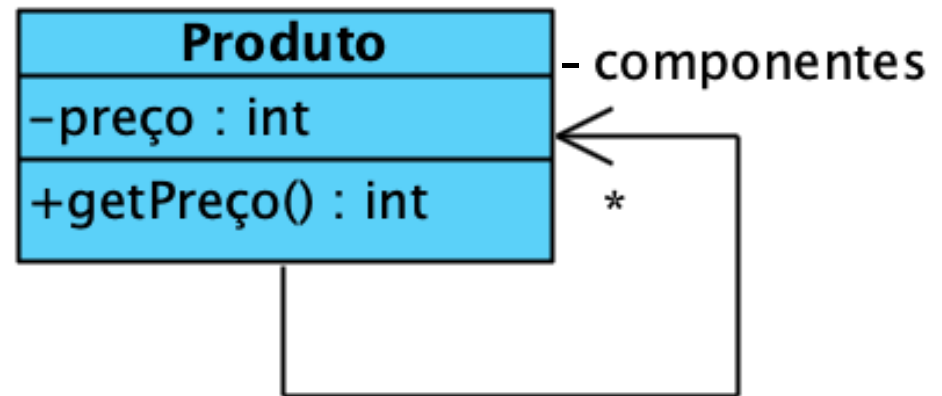
- $1$  — um objecto = objecto obrigatório
- $0..1$  — zero ou um objectos = objecto opcional
- $n..*$  —  $n$  ou mais objectos ( $0..* = *$ )



## Relações entre classes - Associação reflexiva

- Definem uma relação entre objectos da mesma classe

```
class Produto {  
    private int preço;  
    private Collection<Produto> componentes;  
  
    public int getPreço() {  
        return this.preço;  
    }  
}
```

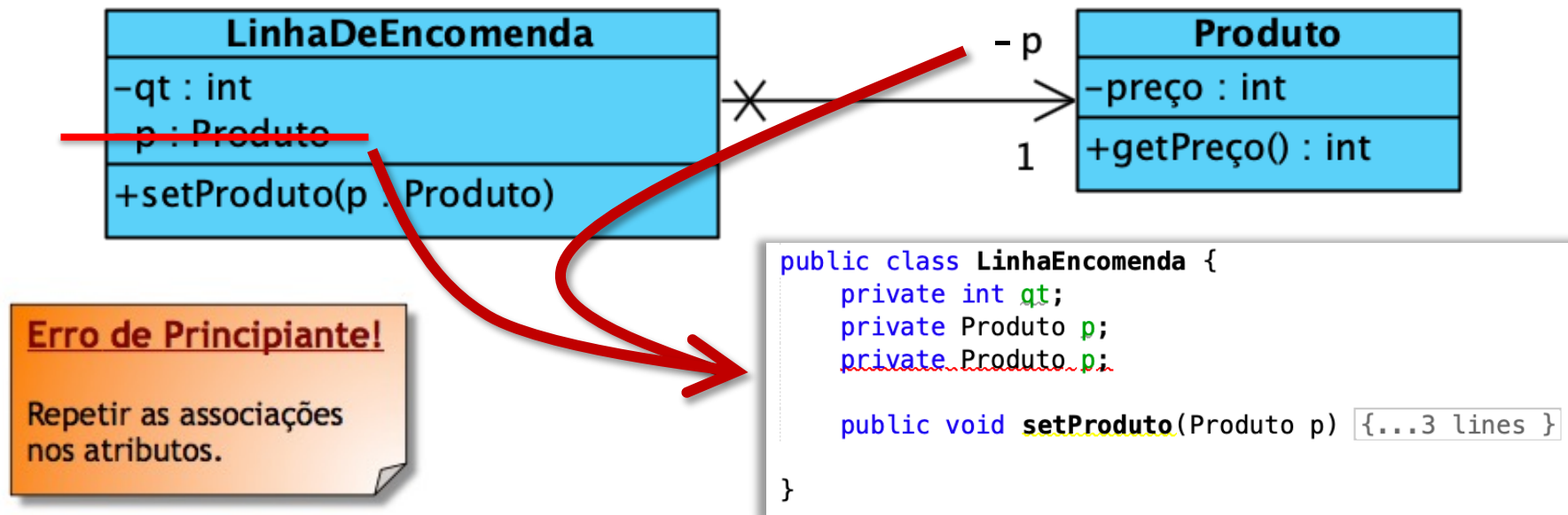


- Um Produto pode ser construído a partir de outros Produtos



# Relações entre classes - Associações vs. Atributos

- Atributos (de instância) representam propriedades das instâncias das classes
  - São codificados como variáveis de instância
- Associações também representam propriedades das instâncias das classes
  - também são codificados como variáveis de instância
- Atributos devem ter tipos simples
- **Utilizar associações para tipos estruturados**



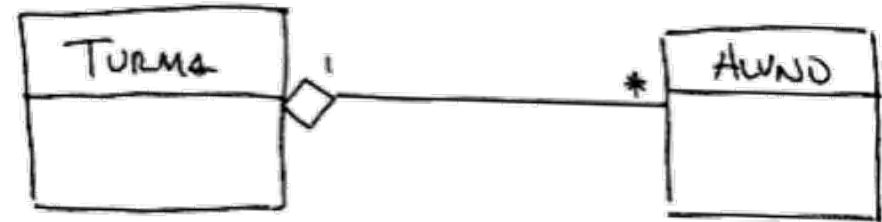


# Relações entre classes - Agregação vs. Composição

- Por vezes a relação entre duas classes implica uma relação todo-parte
  - mais forte que simples associação
  - Exemplo: uma Turma é constituída por Alunos

- **Agregação**

- Os alunos fazem parte da estrutura interna da Turma
- Apesar disso, os Alunos tem existência própria



- **Composição**

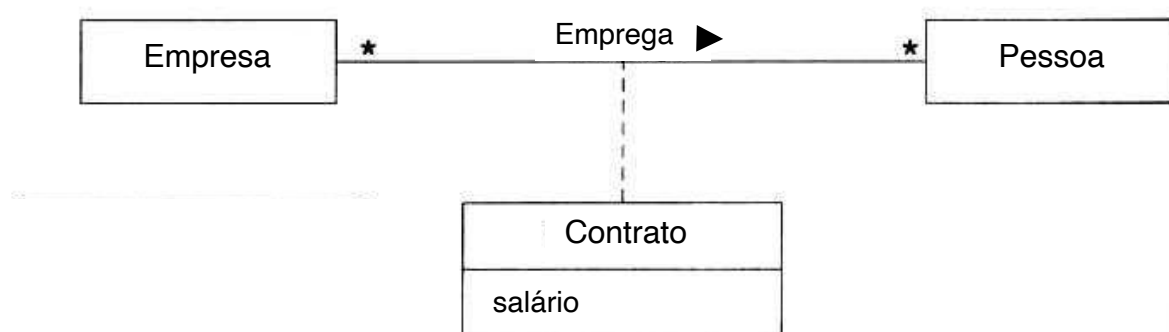
- Os alunos (da Turma) só existem no contexto da Turma
- Os alunos não têm existência para além da existência da Turma (!?)



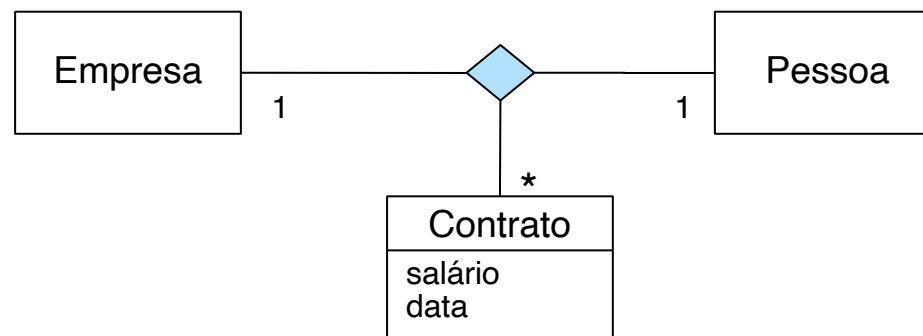


# Relações entre classes

- Não são obrigatoriamente binárias
- Já vimos...
  - Classes de associação:



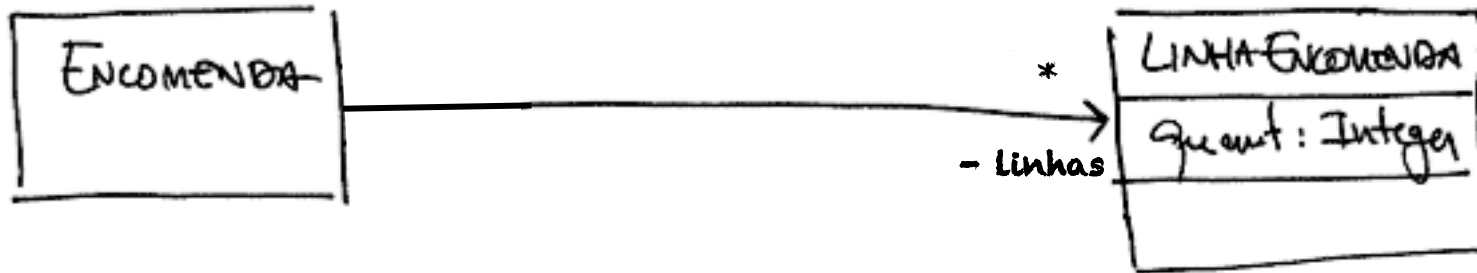
- Associações  $n$ -árias:





# Relações entre classes - Associações qualificadas

- Produto é chave na relação entre Encomenda e LinhaEncomenda
  - Para cada produto p existe (no máximo) uma linha de encomenda



```
public class Encomenda {  
  
    private Map<Produto, LinhaEncomenda> linhas;  
    ...  
    public LinhaEncomenda getLinhaEnc(Produto umProd);  
    public void addLinhaEncomenda(Integer qt,  
                                   Produto umProd);  
    ...  
}
```



# Relações entre classes - Associações qualificadas

- Produto é chave na relação entre Encomenda e LinhaEncomenda
  - Para cada produto p existe (no máximo) uma linha de encomenda

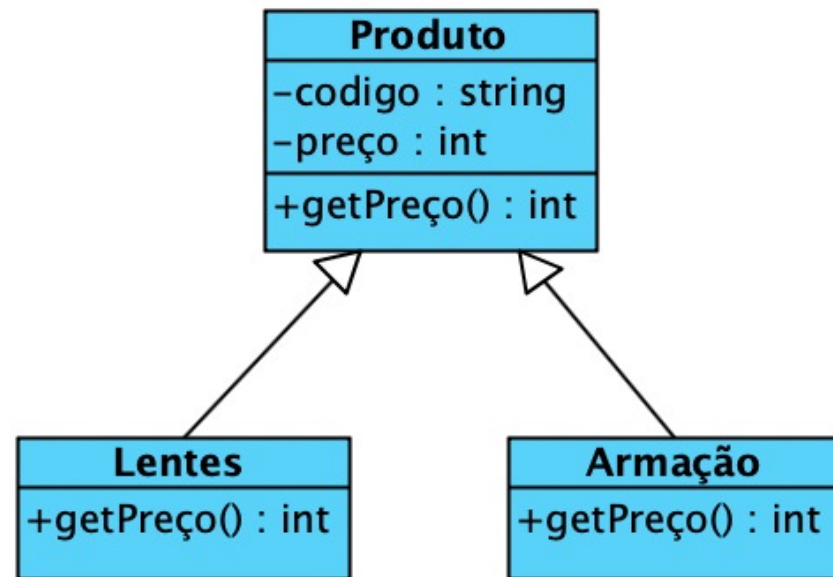


```
public class Encomenda {  
    ...  
    private Map<Produto, LinhaEncomenda> linhas;  
    ...  
    public LinhaEncomenda getLinhaEnc(Produto umProd);  
    public void addLinhaEncomenda(Integer qt,  
                                    Produto umProd);  
    ...  
}
```



# Relações entre classes - Generalização/Especialização

- Indica a relação entre uma classe mais geral (super-classe) e uma classe mais específica (sub-classe).
- Noção de *is-a* – tipagem / substitubilidade
- Polimorfismo – duas sub-classes podem fornecer métodos diferentes para implementar uma operação da super classe.
- *Overriding* – sub-classe pode alterar o método associado a uma operação declarada pela super-classe
- Herança simples vs. herança múltipla
- Notação:





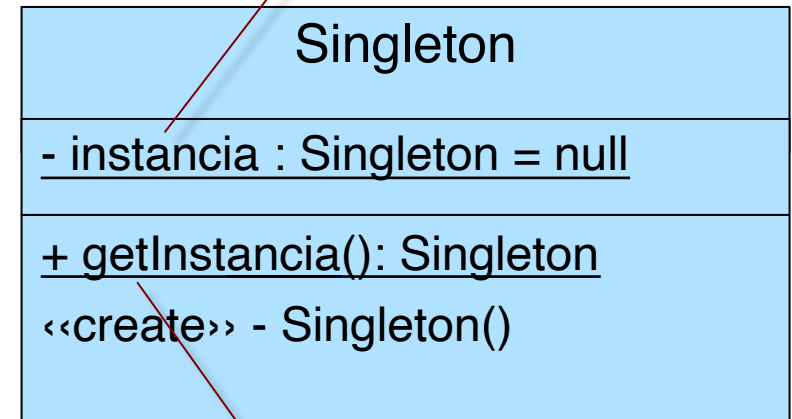


# Operações e variáveis de classe

- Variáveis de classe são globais a todas as instâncias de uma classe.
- Métodos de classe são executados directamente pela classe e não pelas instâncias (logo, não tem acesso directo a variáveis/métodos de instância).
- São representados tal como variáveis/métodos de instância, mas sublinhados.
- Deve evitar-se abusar de operações e variáveis de classe.

```
public class Singleton {  
    private static Singleton instancia = null;  
  
    public static Singleton getInstance() {  
        if (Singleton.instancia==null)  
            Singleton.instancia = new Singleton();  
        return Singleton.instancia;  
    }  
  
    private Singleton() {}  
}
```

Variável de classe  
(static no Java)



Operação de classe  
(static no Java)



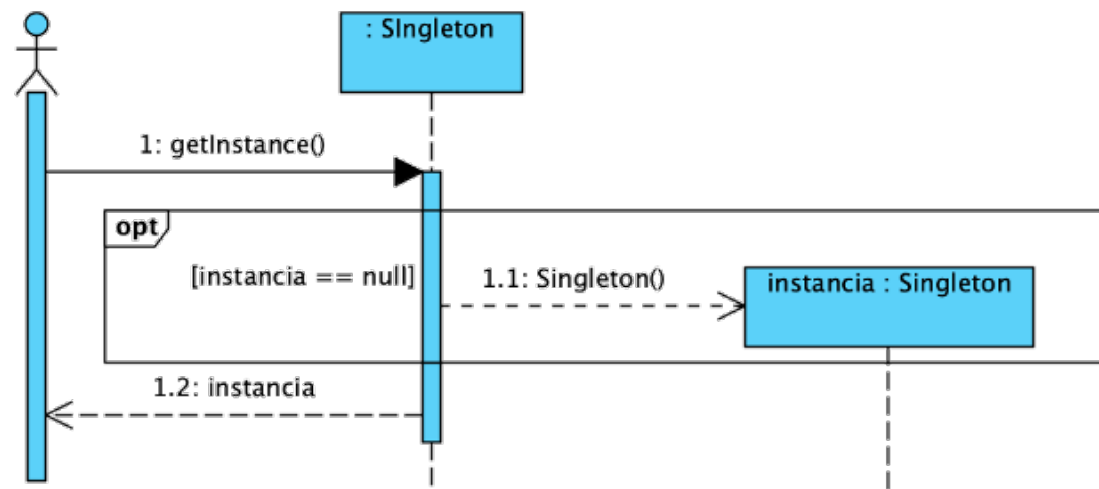
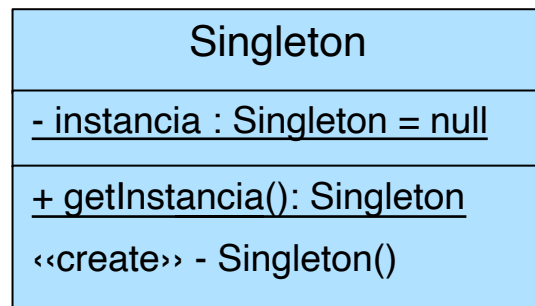
# Singleton Pattern

- Garantir que só é criada uma instância da classe....

```
public class Singleton {
    private static Singleton instancia = null;

    public static Singleton getInstance() {
        if (Singleton.instancia==null)
            Singleton.instancia = new Singleton();
        return Singleton.instancia;
    }

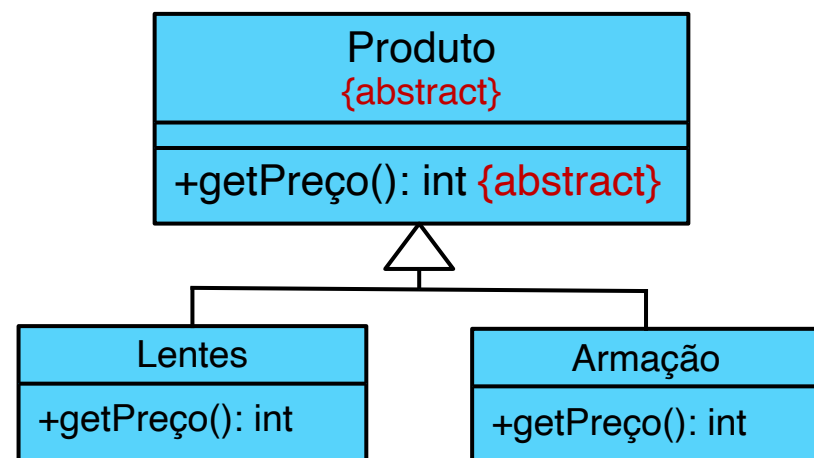
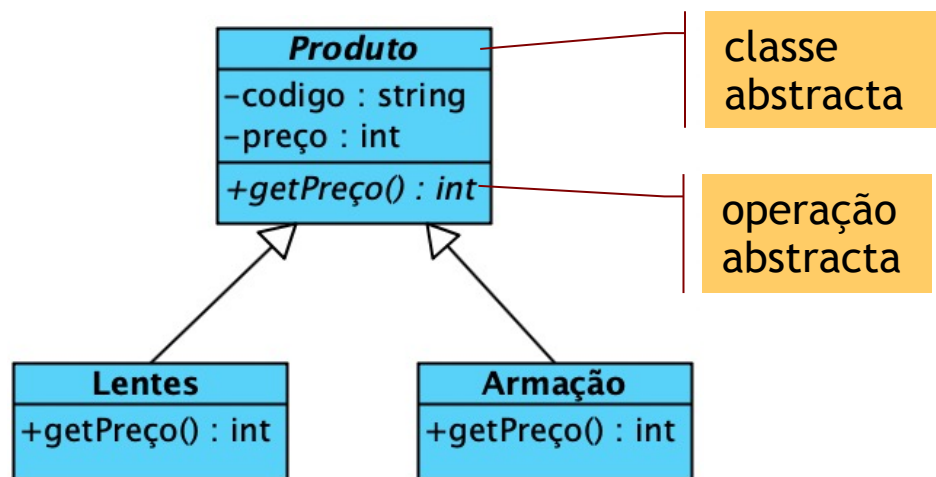
    private Singleton() {}
}
```





# Classes abstractas

- Nem sempre ao nível da super-classe é possível saber qual deverá ser o método associado a uma operação
  - Uma operação abstracta é uma operação que não tem método associado na classe em que está declarada
- Quando se está a utilizar uma hierarquia de classes para representar subtipos, pode não fazer sentido permitir instâncias da super-classe.
  - Uma classe abstracta é uma classe da qual não se podem criar instâncias e pode conter operações abstractas
  - Classes concretas (não abstractas) não podem conter métodos abstractos!
- Notação: em *itálico* ou através da propriedade **{abstract}**





# Classes parametrizadas (*Template classes*)

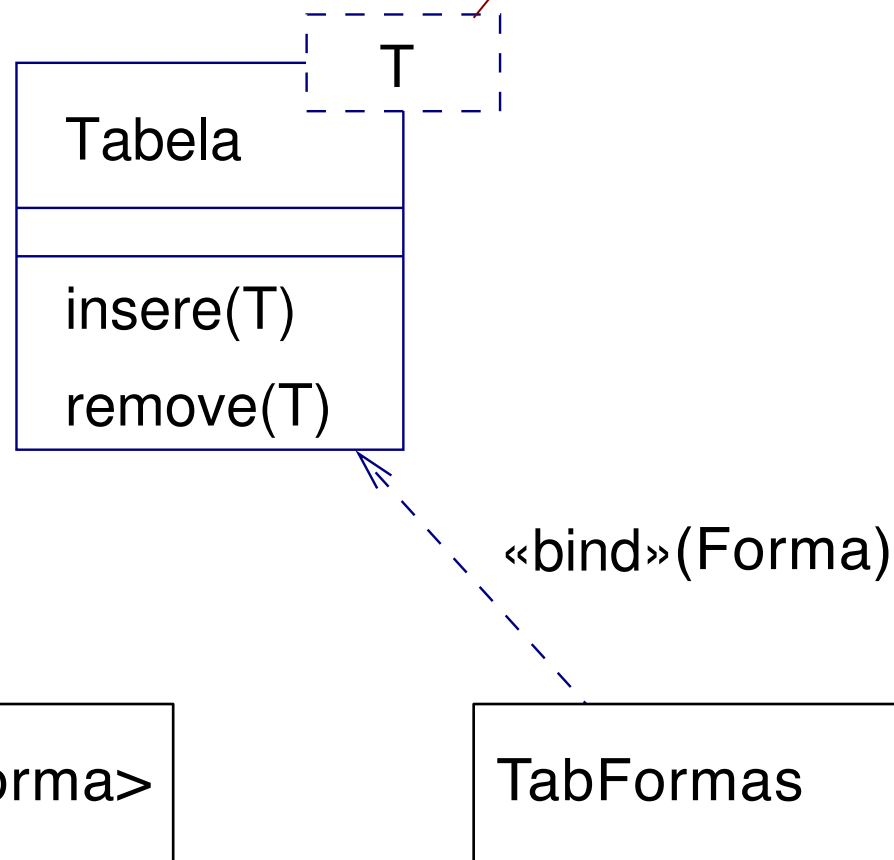
Java Generics!

Parâmetro

```
class Tabela<T> {
    public List<T> elementos;

    public void insere(T t) { ...3 lines }
    public void remove(T t) { ...3 lines }
}
```

```
class TabFormas {
    private Tabela<Forma> tabela;
}
```

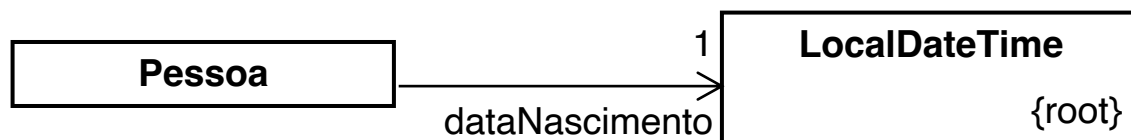




## Outras propriedades

- Classes etiquetadas com a propriedade

- {root} - não podem ser generalizadas



- {active} - são consideradas activas (e.g. *threads*)



- {leaf} - não podem ser especializadas (classes final no Java)





# Mecanismos de extensibilidade

- “Tagged Values” (valores etiquetados)
- Estereótipos
- Restrições (“constraints”)
- Valores Etiquetados
  - Definem novas propriedades das “coisas”
  - Trabalham ao nível dos meta-dados



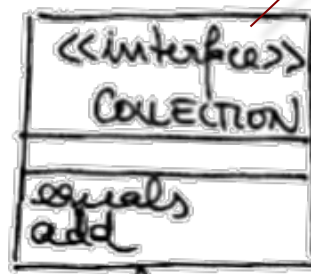
Valores etiquetados



# Mecanismos de extensibilidade

## • Estereótipos

- Permitem a definição de variações dos elementos de modelação existentes (ex: «include», «extend» são estereótipos de dependência)
- Possibilitam a extensão da linguagem de forma controlada
- Cada estereótipo pode ter a si associado um conjunto de valores etiquetados
  - Trabalham ao nível dos meta-dados
- Meta-tipo de dados  $\neq$  Generalização

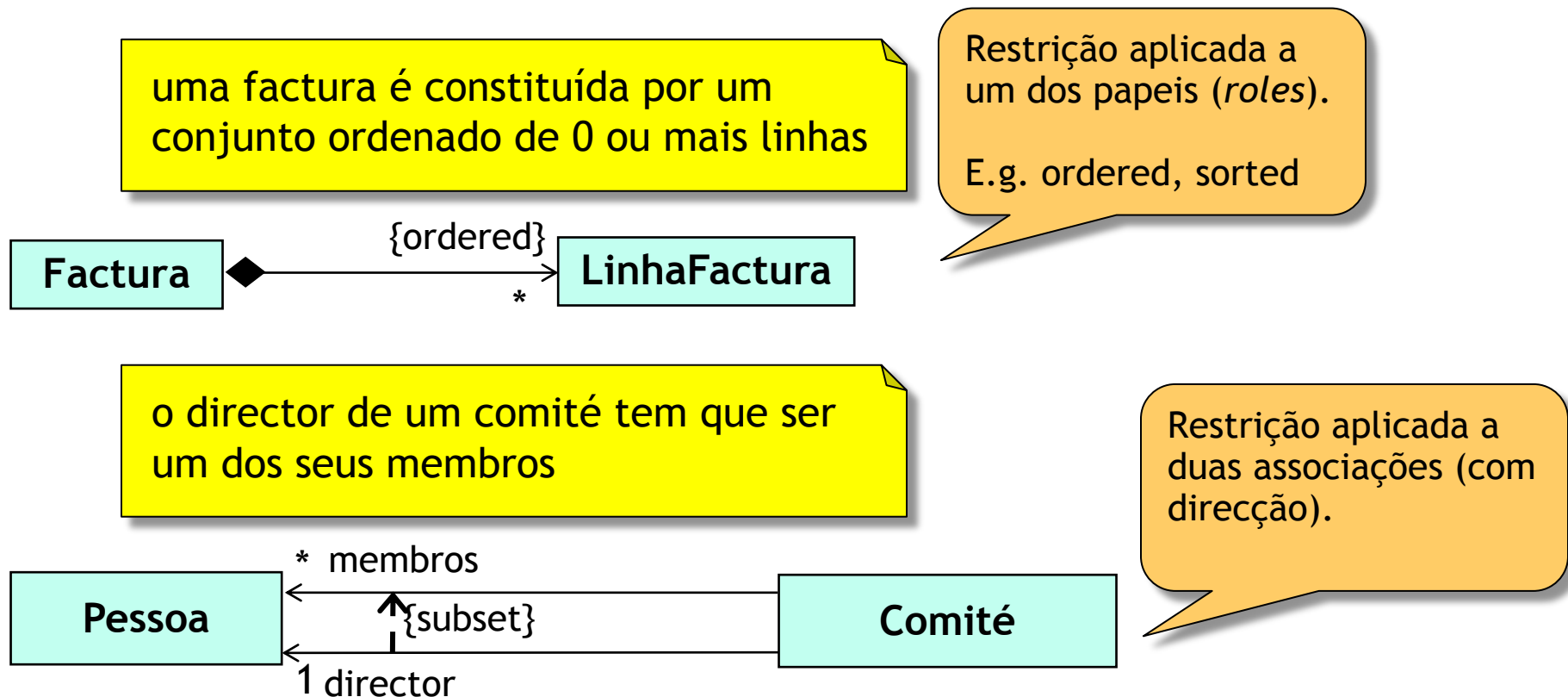


Estereótipo

# Mecanismos de extensibilidade

## • Restrições

- Utiliza-se quando a semântica das construções diagramáticas do UML não é suficiente

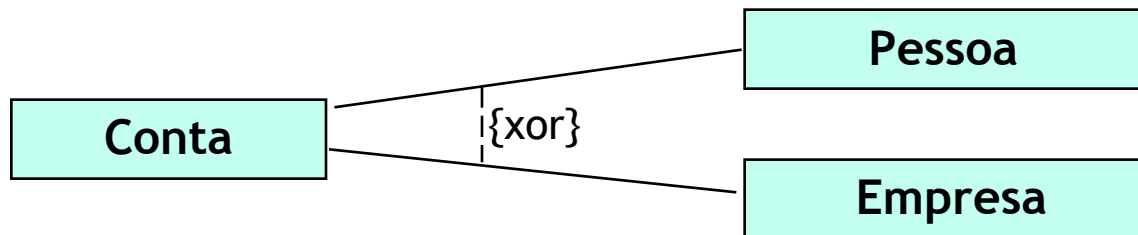






# Restrições às associações

uma conta pode ser de uma pessoa ou de uma empresa (mas não de ambos)



Restrições aplicadas a duas associações (sem direção).  
E.g. associações mutuamente exclusivas.

- Veremos mais sobre restrições quando falarmos de OCL



# Diagramas da UML 2.x

