# Modelagem Conceitual

© Prof. Raul Sidnei Wazlawick

UFSC-CTC-INE

2010

Fonte: Análise e Projeto de Sistemas de Informação Orientados a Objetos, 2ª Edição, Elsevier, 2010.

# Análise de Domínio

- Descoberta das informações que são gerenciadas no sistema: *representação* e *transformação* da informação.
- Ocorre em pelo menos duas fases do Processo Unificado.
  - Na fase de *concepção* pode-se fazer um modelo conceitual preliminar.
  - Na fase de *elaboração* este modelo é refinado e complementado.

# Aspectos da Análise de Domínio

- *Estático* ou *estrutural*, que pode ser representado no *modelo conceitual*.
- *Funcional, que* pode ser representado através dos *contratos de operações* e *consultas de sistema*.

# Caracterização do Modelo Conceitual

- Deve ser independente da solução tecnológica que virá a ser adotada.
- Deve conter apenas elementos referentes ao domínio do problema em questão.
- Os elementos da solução ficam relegados à atividade de projeto :
  - Interfaces.
  - Formas de armazenamento (banco de dados).
  - Segurança de acesso.
  - Comunicação.
  - Etc.

# O Modelo Conceitual é Estático

- Não podem existir no modelo conceitual referências a operações ou aspectos dinâmicos dos sistemas.
- Então, embora o modelo conceitual seja representando pelo diagrama de classes da UML, o analista não deve ainda adicionar métodos a essas classes.

# Elementos do Modelo Conceitual

- **Atributos**: informações alfanuméricas simples, como números, textos, datas, etc.
- **Classes** ou **conceitos**: que são a representação da informação complexa que agrega atributos e que não pode ser descrita meramente por tipos alfanuméricos.
- **Associações**: que consistem em um tipo de informação que liga diferentes conceitos entre si.

# Atributos

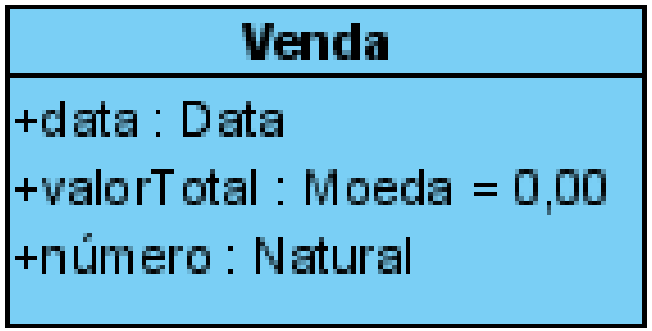
- São os tipos escalares
- NÃO são estruturas de dados como listas, tabelas e *arrays*
- São sempre representados no contexto de uma classe:

# Tipagem

- Atributos podem ter tipos clássicos como string, inteiro, data, etc., ou tipos primitivos definidos pelo analista:

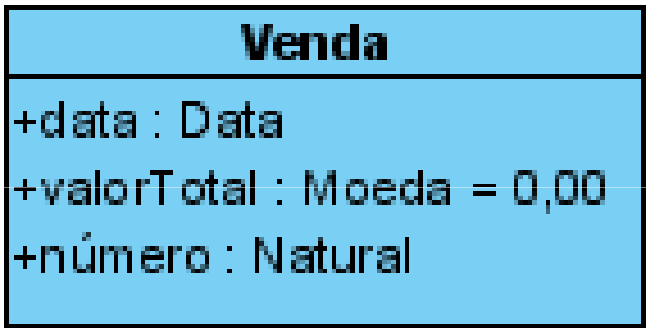
# Valores Iniciais

- Atributos podem ser definidos com valores iniciais.
- Valores iniciais são produzidos no atributo no momento que as instâncias da classe correspondente forem criadas



# OCL – Object Constraint Language

- Pode ser usada, entre outras coisas, para definir atributos iniciais:

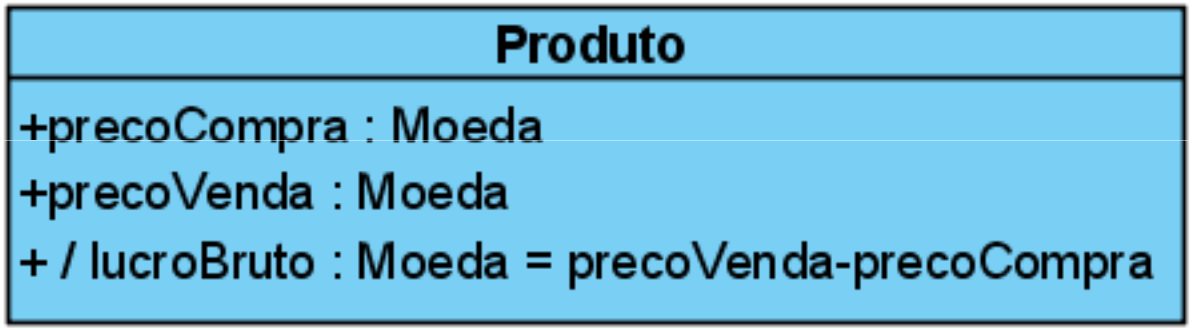



- `Context Venda::valorTotal:Moeda`
  `init: 0,00`

- `Context Venda::valorTotal`
  `init: 0,00`

# Atributos Derivados

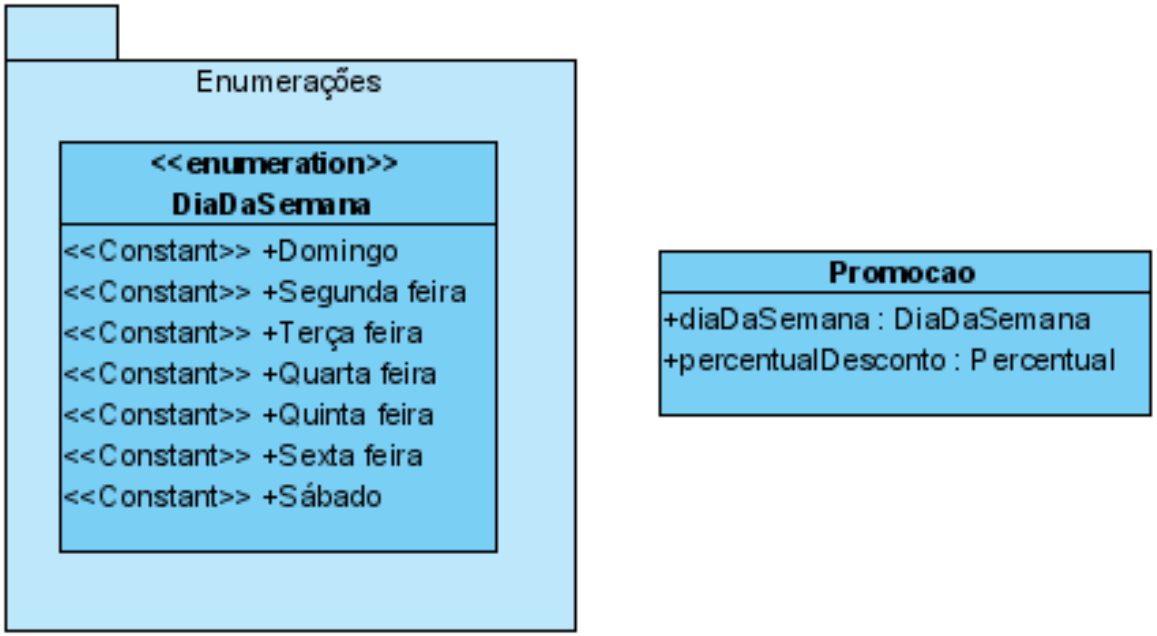
- Não são definidos diretamente, mas calculados




- `Context Produto::lucroBruto`
  **`derive:`**
  `self.precoVenda - self.precoCompra`

# Enumerações

- São um meio termo entre o conceito e o atributo.
- São basicamente *strings* e se comportam como tal, mas há um conjunto predefinido de *strings* válidas que constitui a enumeração.

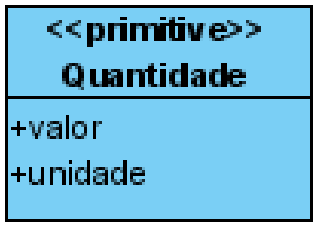


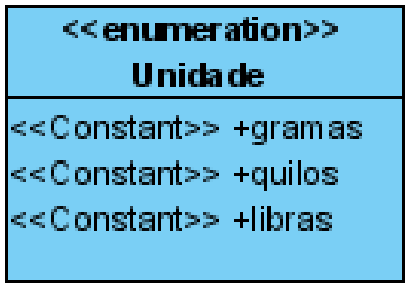
# Características de Enumerações

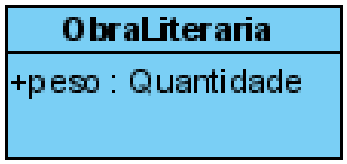
- NÃO podem ter associações com outros elementos.
- NÃO podem ter atributos.
- Se isso acontecer, então não se trata mais de uma enumeração, mas de um conceito complexo.
- Em OCL:
  - DiaDaSemana::Terca_Feira

# Tipos Primitivos

- O analista pode e deve definir *tipos primitivos* sempre que se deparar com atributos que tenham regras de formação, como no caso do CPF.
- Tipos primitivos podem ser classes estereotipadas com <<primitive>>.



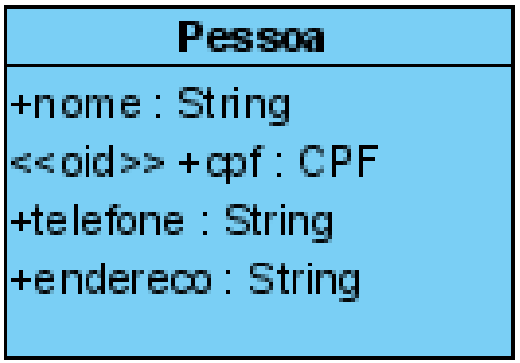



# Conceitos

- Conceitos são mais do que valores alfanuméricos.
- São também mais do que meramente um amontoado de atributos, pois:
  - Eles trazem consigo um significado e
  - Podem estar associados uns com os outros.

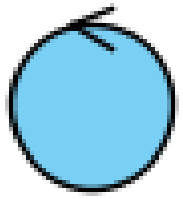
# Identificador

- É um atributo que permite que uma instância de um conceito seja diferenciada de outras.
- Estereótipo: <<oid>> (*Object IDentifier*)
- Não existem duas instâncias do mesmo conceito com o mesmo valor para este atributo.



# Classe Controladora de Sistema

- Representa o Sistema como um todo.
- É o ponto de partida para as conexões das associações.
- Tem uma única instância estereotipada com <<control>> ou na notação de Jacobson:



Livir

# Conceitos Dependentes e Independentes

- *Dependente:* precisa estar associado a outros conceitos para fazer sentido, ou seja, para representar uma informação minimamente compreensível.
  - Usualmente são gerenciados por processos de negócio complexos.
- *Independente*: pode ser compreendido sem estar associado a outros.
  - Usualmente são cadastros ou CRUD.

# Como Encontrar Conceitos e Atributos

- Olhar para o texto dos casos de uso expandidos ou os diagramas de seqüência de sistema.
  - Substantivos, adjetivos, verbos, sintagmas nominais, etc. são candidatos.
  - Agrupar sinônimos.
  - Identificar conceitos e atributos.

**Caso de Uso: Comprar livros**
1. [IN] O comprador informa sua identificação.
2. [OUT] O sistema informa os livros disponíveis para venda (título, capa e preço) e o conteúdo atual do carrinho de compras.
3. [IN] O comprador seleciona os livros que deseja comprar.
4. O comprador decide se finaliza a compra ou se guarda o carrinho:
    4.1 Variante: Finalizar a compra.
    4.2 Variante: Guardar carrinho.

**Variante 4.1: Finalizar a compra**
4.1.1. [OUT] O sistema informa o valor total dos livros e apresenta as opções de endereço cadastradas.
4.1.2. [IN] O comprador seleciona um endereço para entrega.
4.1.3. [OUT] O sistema informa o valor do frete e total geral, bem como a lista de cartões de crédito já cadastrados para pagamento.
4.1.4. [IN] O comprador seleciona um cartão de crédito.
4.1.5. [OUT] O sistema envia os dados do cartão e valor da venda para a operadora.
4.1.6. [IN] A operadora informa o código de autorização.
4.1.7. [OUT] O sistema informa o prazo de entrega.

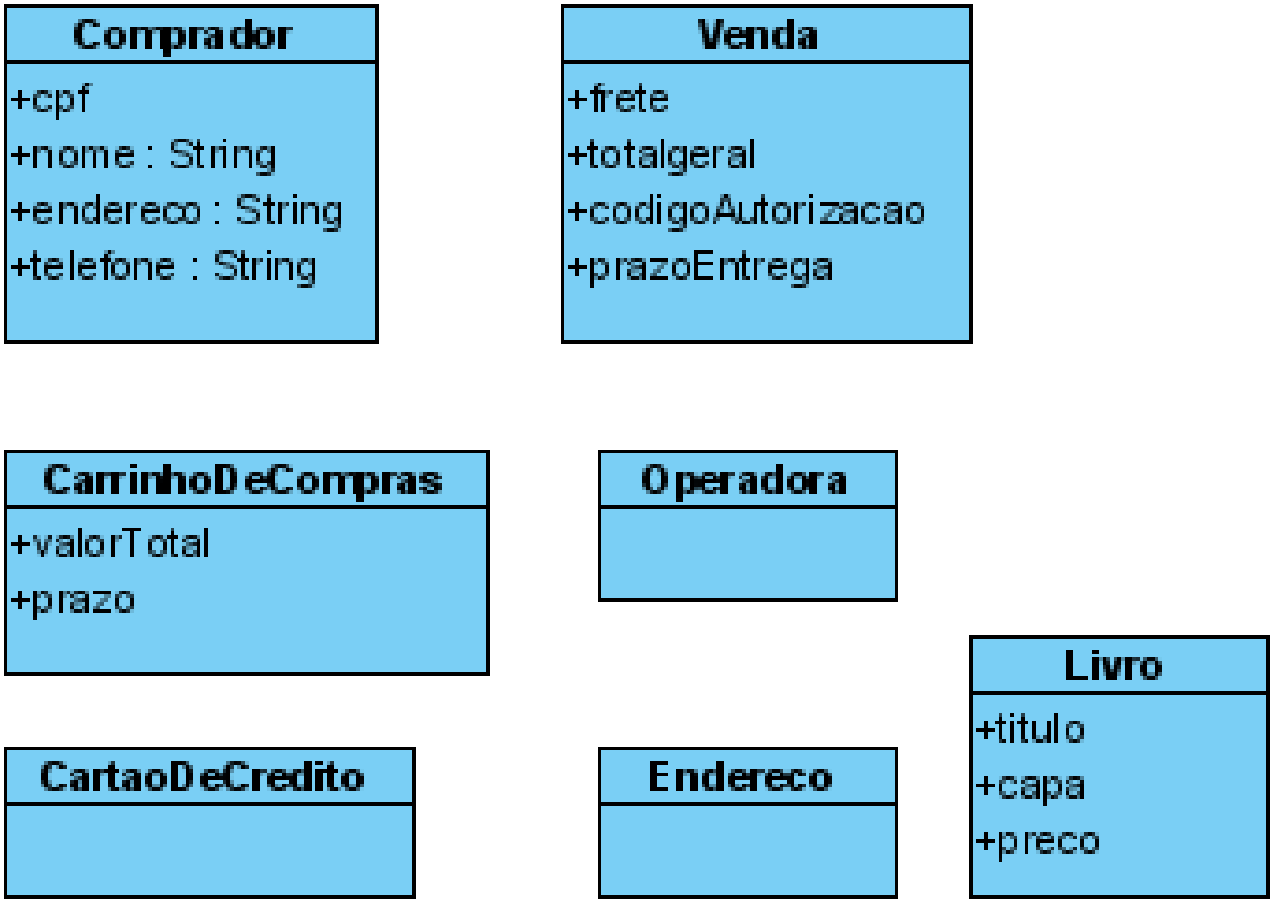
**Variante 4.2: Guardar carrinho**
4.2.1 [OUT] O sistema informa o prazo (dias) em que o carrinho será mantido.

**Exceção 1a: Comprador não cadastrado**
1a.1 [IN] O comprador informa seu CPF, nome, endereço e telefone.
Retorna ao passo 1.

**Exceção 4.1.2a: Endereço consta como inválido**
4.1.2a.1 [IN] O comprador atualiza o endereço.
Avança para o passo 4.1.2.

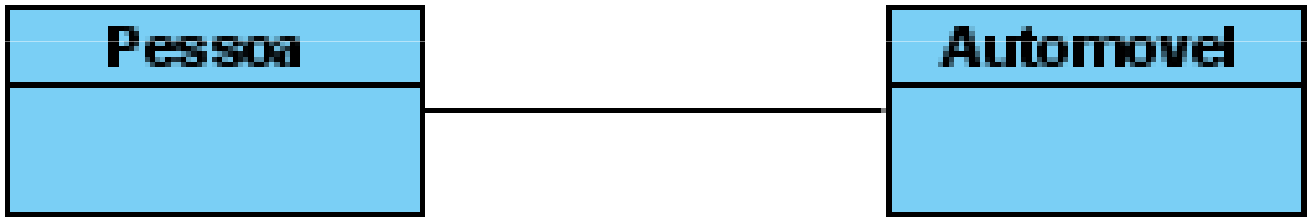
**Exceção 4.1.6a: A operadora não autoriza a venda**
4.1.6a.1 [OUT] O sistema apresenta outras opções de cartão ao comprador.
4.1.6a.2 [IN] O comprador seleciona outro cartão.
Retorna ao passo 4.1.5.

# Resultado


Comprador
+cpf
+nome : String
+endereco : String
+telefone : String
Venda
+frete
+totalgeral
+codigoAutorizacao
+prazoEntrega
CarrinhoDeCompras
+valorTotal
+prazo
Operadora
Livro
+titulo
+capa
+preco
CartaoDeCredito
Endereco

# Associações

- Relacionam dois ou mais conceitos entre si.



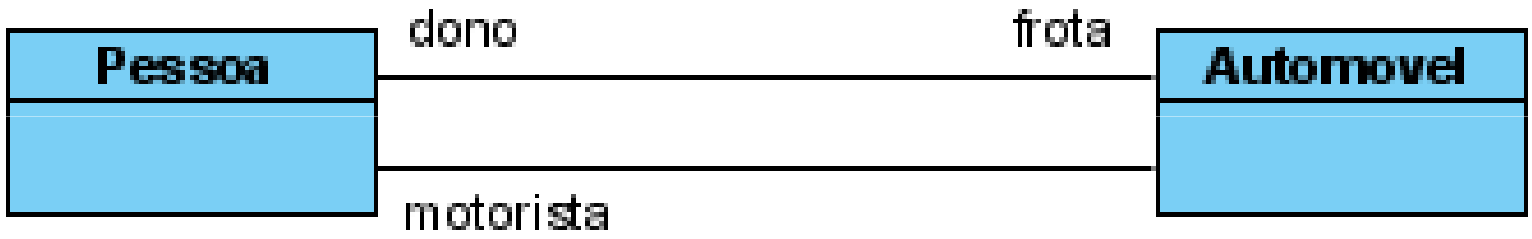
# Associação x Operação

- *Associação* é uma relação estática que pode existir entre conceitos complexos, complementando a informação que se tem sobre eles em um determinado instante, ou referenciando informação associativa nova.
- *Operação* é o ato de transformar a informação, fazendo-a passar de um estado para outro, mudando, por exemplo, a configuração das associações, destruindo e/ou criando novas associações ou objetos, ou modificando o valor dos atributos.

# Papeis

- Correspondem à função que um lado da associação representa em relação aos objetos do lado oposto.

# Múltiplas Associações Demandam Papeis



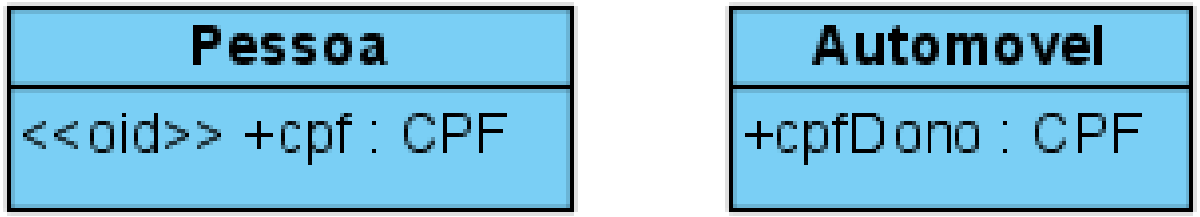
# Como Encontrar Associações

- *Conceitos dependentes* (como Compra) precisam necessariamente estar ligados aos conceitos que os complementam (como Comprador e Item).
- *Informações associativas* só podem ser representadas através de associações.

# Associação x Atributo

- Errado:


- Errado:



# Multiplicidade de Papel

- Indica quantos objetos podem se associar.
- Sempre há um limite inferior.
- Pode haver um limite superior.

# Consideração de Multiplicidade

- O papel é obrigatório ou não?
  - Uma pessoa é obrigada a ter pelo menos um automóvel?
  - Um automóvel deve obrigatoriamente ter um dono?
- A quantidade de instâncias que podem ser associadas através do papel tem um limite conceitual definido?
  - Existe um número máximo ou mínimo de automóveis que uma pessoa pode possuir?

# Armadilha da Obrigatoriedade:

- A toda venda corresponde um pagamento.
- Mas isso não torna a associação obrigatória, pois a venda pode existir sem um pagamento.
- Um dia ela possivelmente será paga, mas ela pode existir sem o pagamento por algum tempo.
- Então esse papel *não* é obrigatório para a venda.

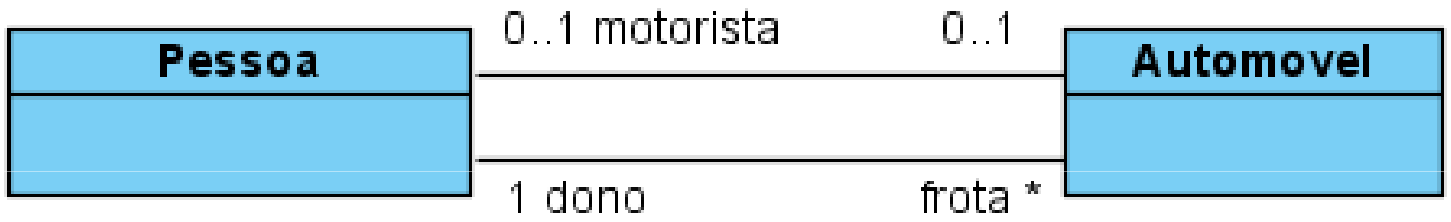
# Armadilha do Limite Máximo

- O número máximo de automóveis que uma pessoa pode possuir é o número de automóveis que existe no planeta.
- Mas à medida que outros automóveis venham a ser construídos, esse magnata poderá possuí-los também.
- Embora exista um limite físico, não há um limite lógico para a posse.
- Então o papel deve ser considerado virtualmente sem limite superior.

# Exemplos de Multiplicidade

- 1 exatamente um.
- 0..1 zero ou um.
- * de zero a infinito.
- 1..* de um a infinito.
- 2..5 de dois a cinco.
- 2,5 dois ou cinco.
- 2,5..8 dois ou de cinco a oito

# Uso no Diagrama



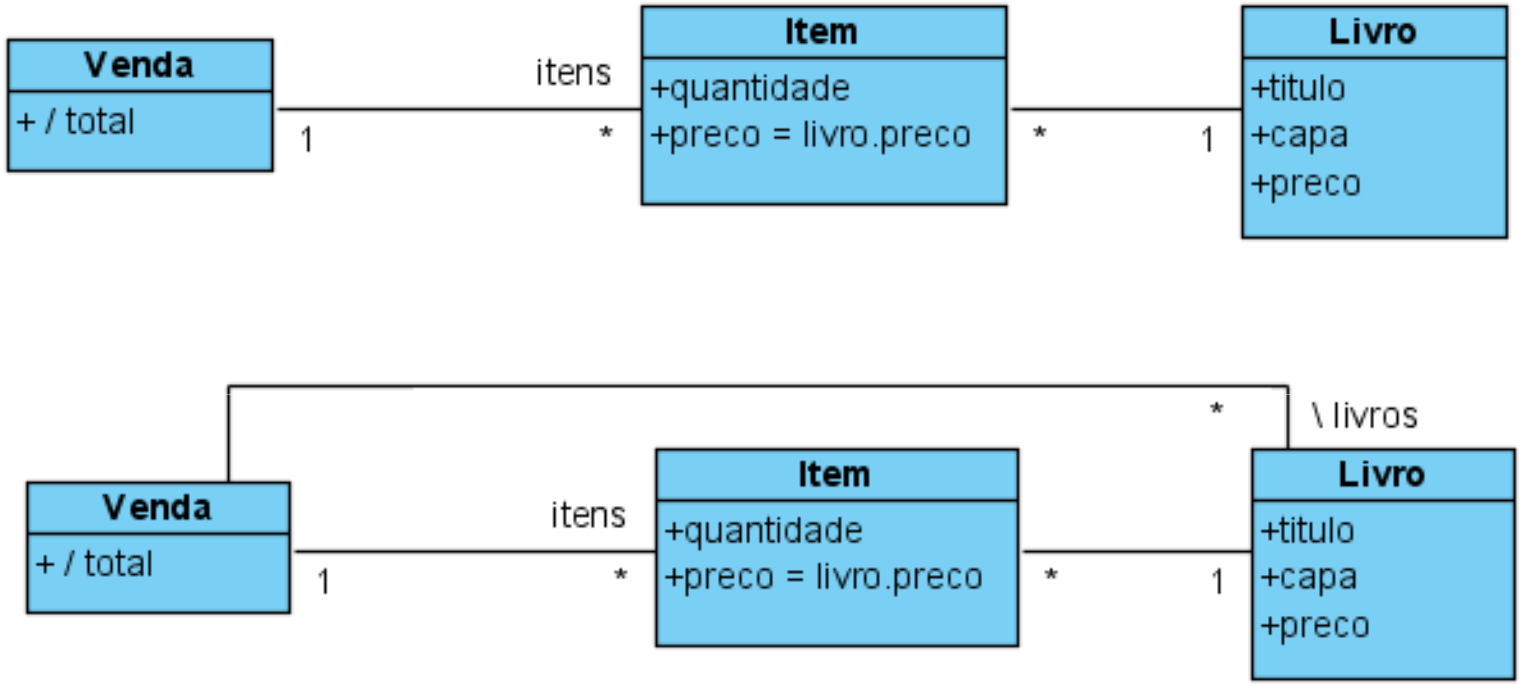
# Direção das Associações

- Uma associação, no modelo conceitual, deve ser *não-direcional*.

# Associação Derivada

- É calculada a partir de outras.

# Exemplo




```
Context Venda::livros
  derive: self.itens.livro
```

# Coleções

- Coleções de objetos são representadas nas associações com papel múltiplo, e não como conceitos.

- Errado:

# Tipos Abstratos de Dados

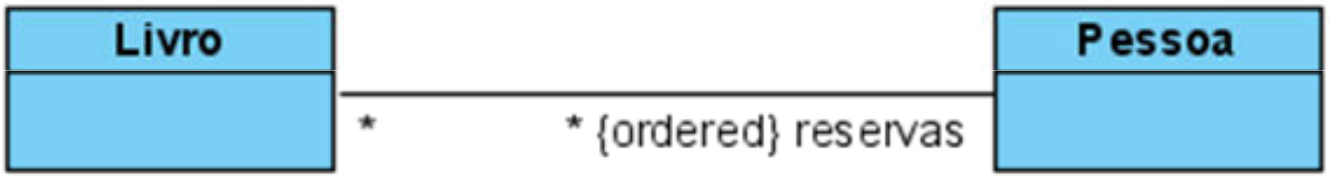
- *
  - Conjunto ou Set
  - Não repete elementos e não tem ordem
- * {ordered}
  - Conjunto Ordenado ou OrderedSet
  - Não repete elementos mas tem ordem
- * {bag}
  - Multiconjunto ou Bag
  - Repete elementos mas não tem ordem
- * {sequence}
  - Lista ou Sequence
  - Repete elementos e tem ordem
- n
  - array

# Conjunto

- Um papel de associação *, na falta de maiores detalhes, representa um *conjunto*, ou seja, elementos não se repetem e não há nenhuma ordem definida entre eles.
- A *frota* é um *conjunto* de automóveis de uma pessoa.
  - Se um mesmo automóvel for adicionado a essa associação para a mesma pessoa, o efeito é nulo, pois ele já pertence ao conjunto.

# Conjunto Ordenado {ordered}

- Existe ordem, mas os elementos não se repetem



# Multiconjunto {bag}

- Elementos podem se repetir, mas a ordem não importa

# Lista {sequence}

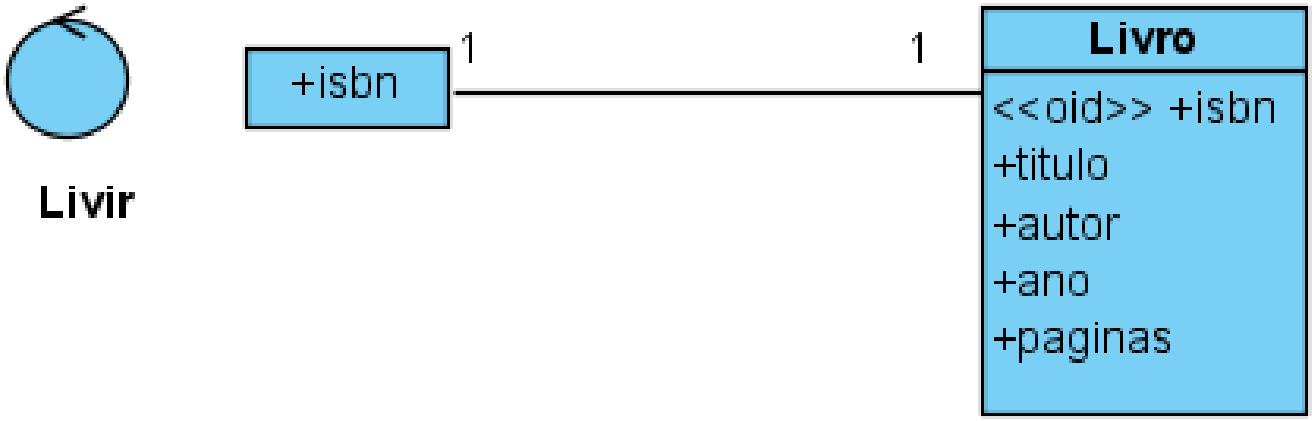
- Há ordem e pode haver repetição

# Casos Especiais de Lista

- Pilha {stack}
- Fila {queue}

# Mapeamento

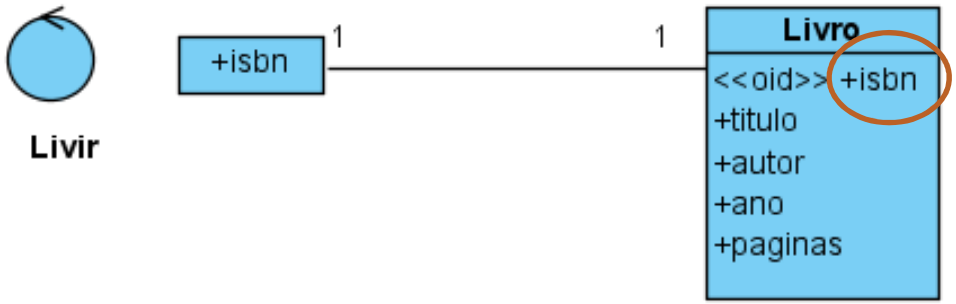
- Associa um valor alfanumérico a um objeto
- Usa-se um *qualificador* na associação



# Tipos de Qualificador

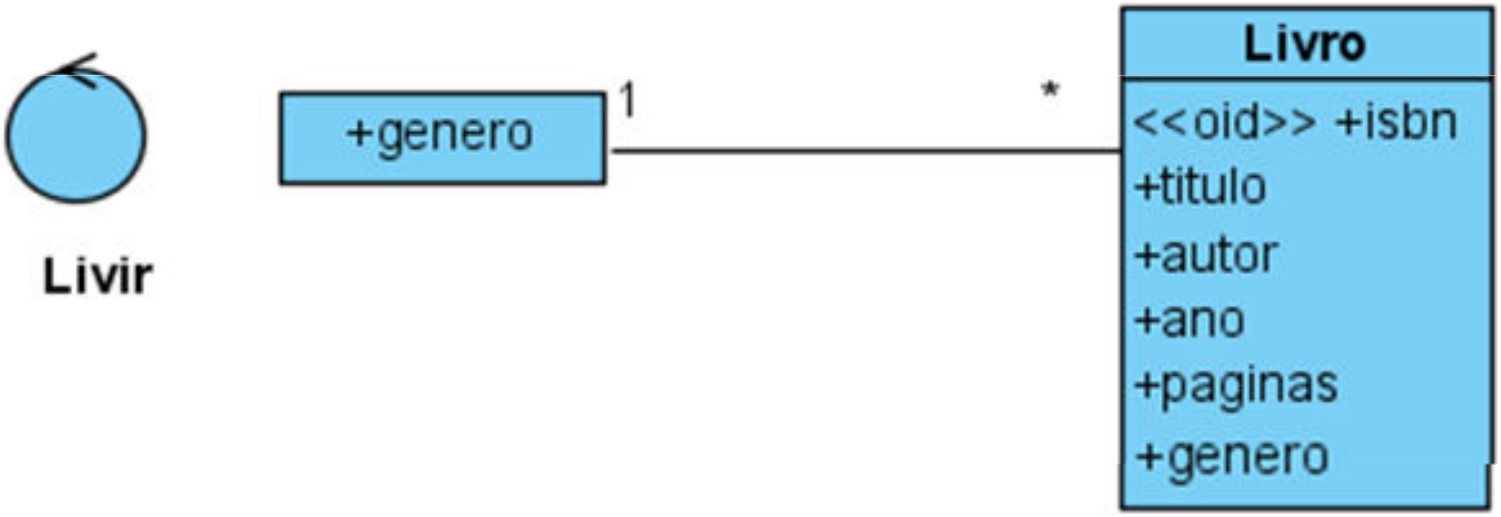
- Interno: é atributo da classe qualificada
- Externo: não é atributo da classe qualificada

- Exemplo de qualificador interno:



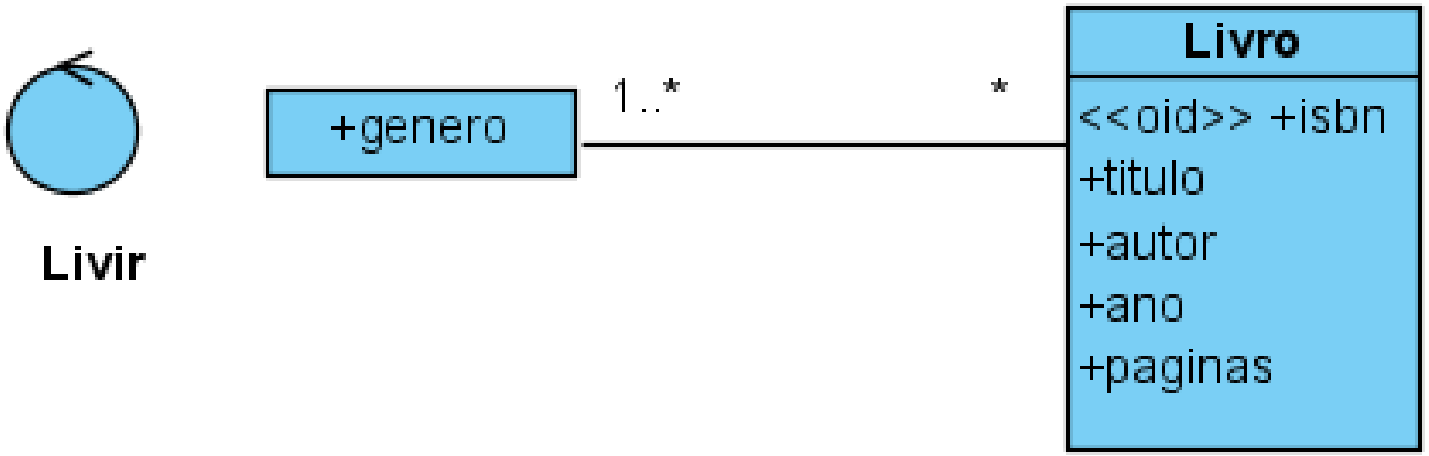
# Partição

- Associa um conjunto a cada qualificador



# Relação

- Associa um conjunto a um qualificador e cada instância pode ser qualificada várias vezes

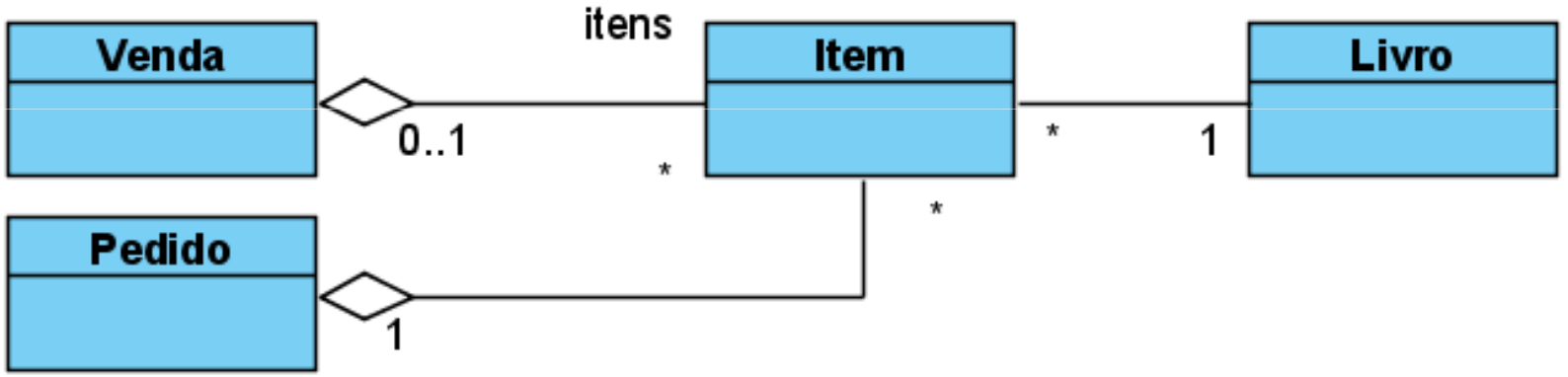


# Composição

- Objetos efetivamente FAZEM PARTE de outros

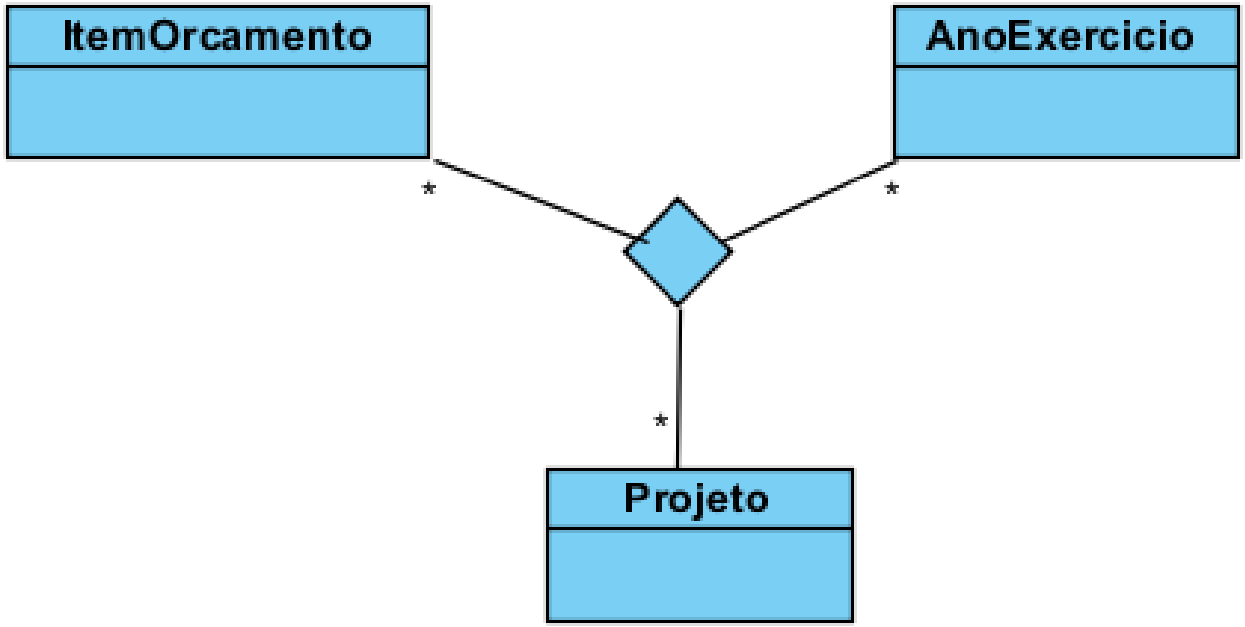
# Agregação

- Objetos formam outros, mas podem ser compartilhados



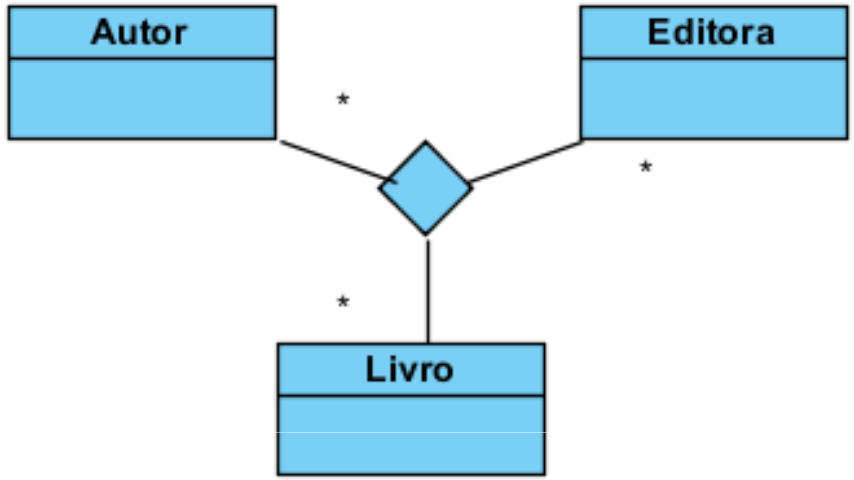
# Associações n-árias

- São raras, mas podem acontecer

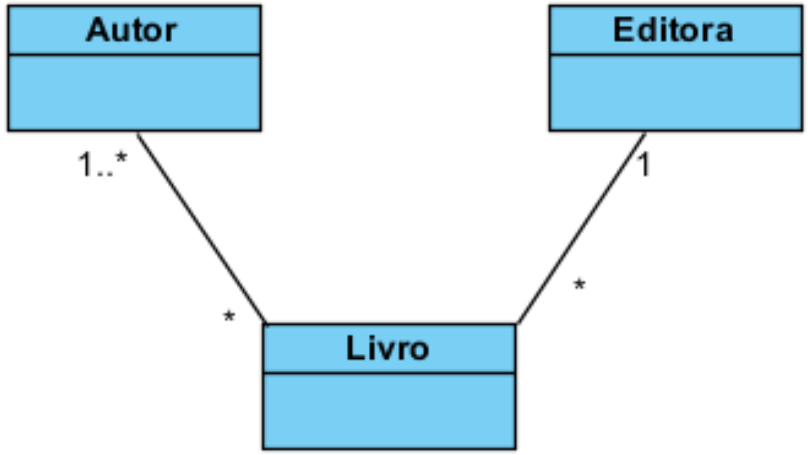


# Armadilha das n-árias

- Errado




- Correto

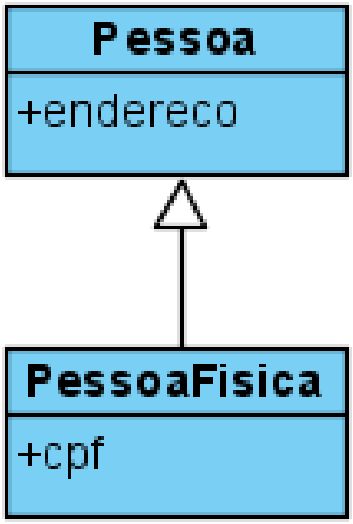


# Organização do Modelo Conceitual

- Técnicas de organização:
  - *Estruturais*: representando relações de generalização estrutural de conceitos, como por exemplo, Pessoa, generalizando PessoaFisica e PessoaJuridica.
  - *Associativas*: representando relações de papéis associativos entre conceitos, como, por exemplo, Pessoa, podendo representar junto a uma empresa o papel de Comprador ou Funcionário.
  - *Temporais*: representando relações entre estados de um conceito como, por exemplo, um Livro e os estados: *encomendado*, *em estoque*, *vendido*, etc.

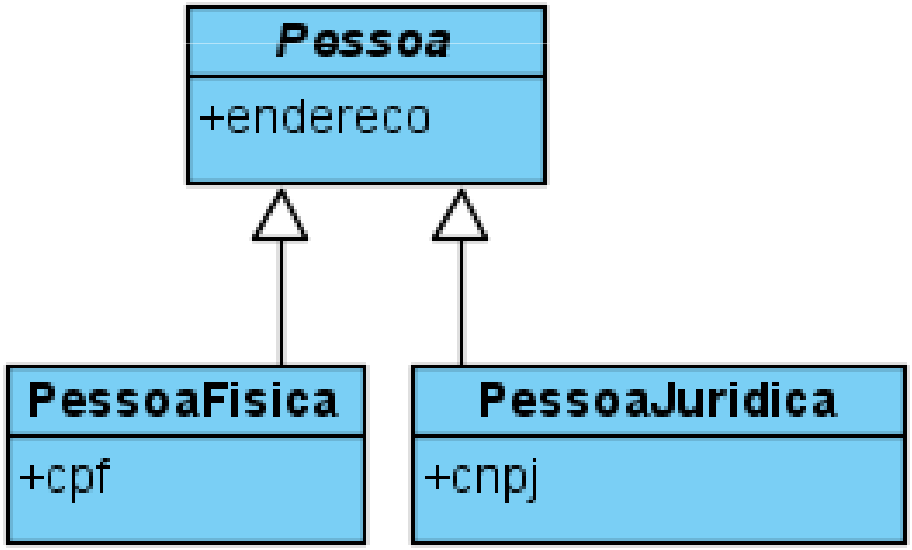
# Herança (Generalização e Especialização)

- Usada para fatorar propriedades.
- Só existe entre classes (não entre instâncias)
- Se A é generalização de B então instâncias de B também são instâncias de A e possuem os mesmos atributos e associações.



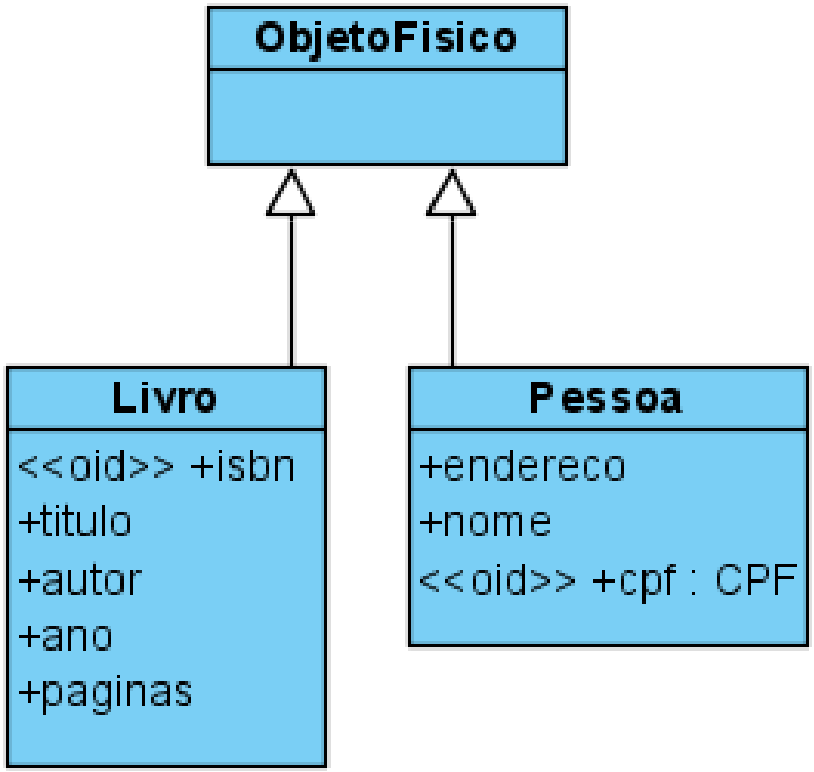
# Classe Abstrata

- Não pode ter instâncias próprias.
- Somente suas subclasses podem ser instanciadas.



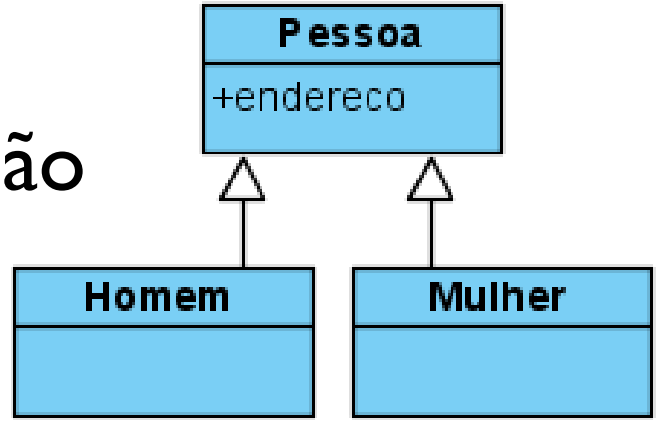
# Armadilha: Superclasse Vazia

- Não se usa generalização neste caso:

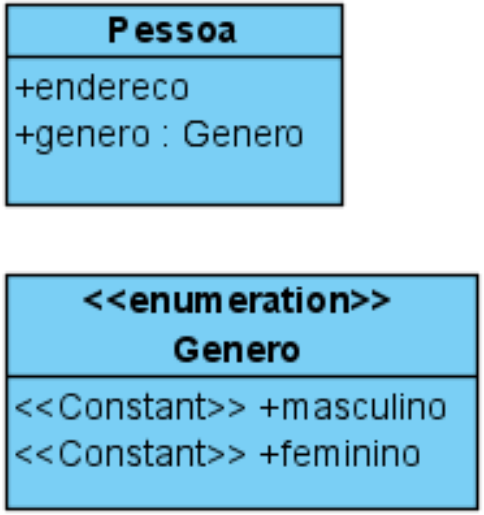


# Armadilha: Subclasses Idênticas ou Vazias

- Não se usa especialização neste caso:

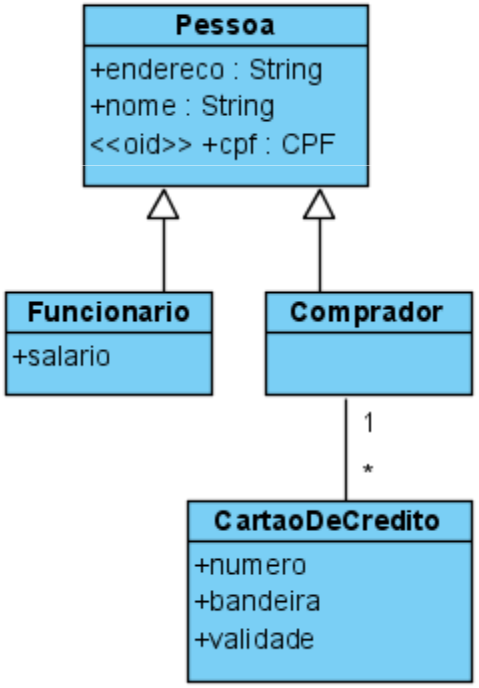



- Solução correta:

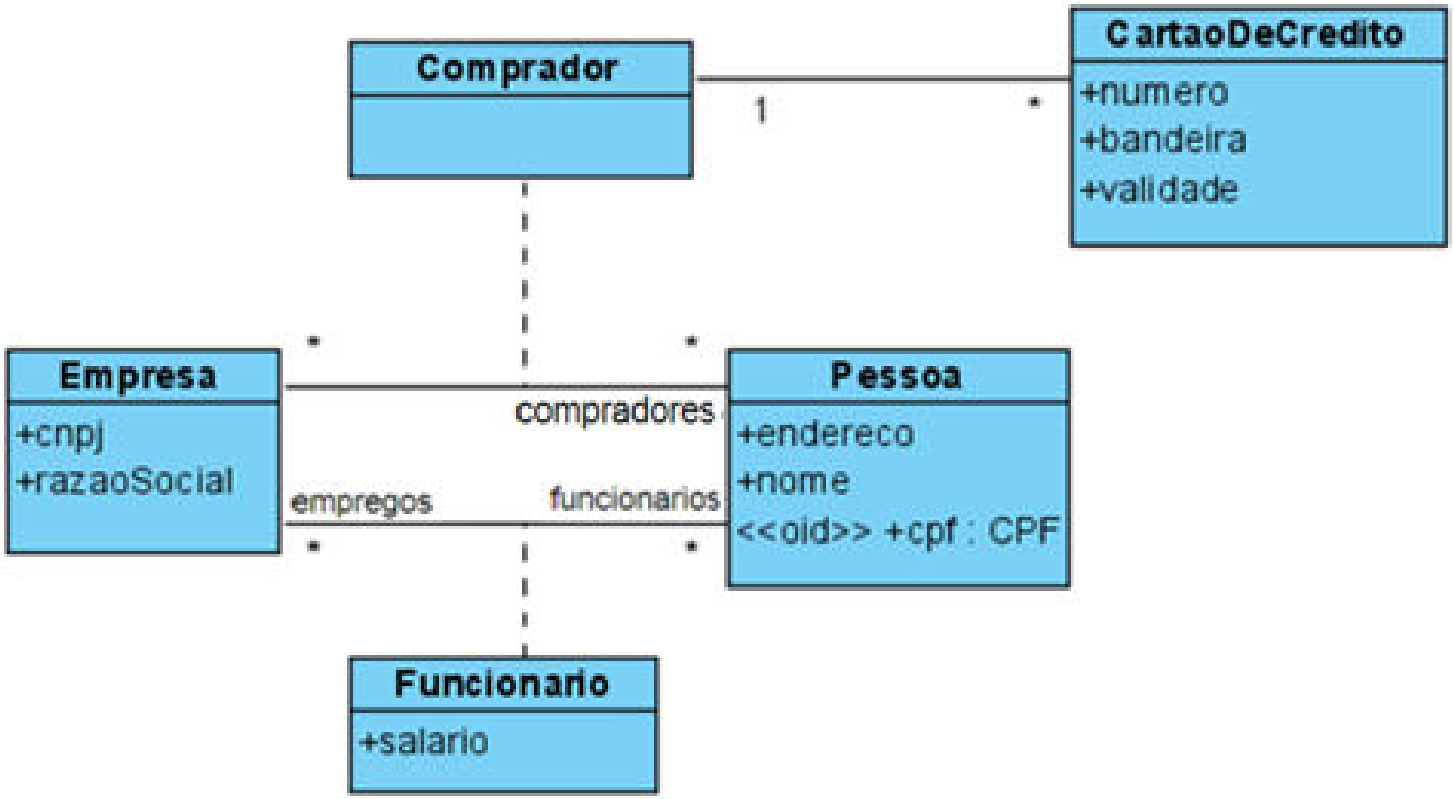


# Técnica Associativa: Classes de associação

- Comprador e Funcionário não são tipos de Pessoa.
- Errado:

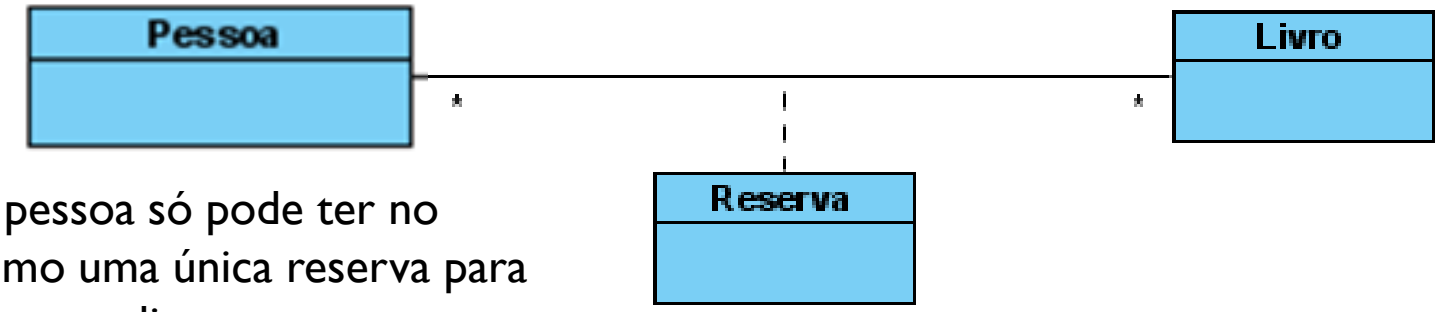


# Representação de Classes de Associação



# Diferença entre Classe de Associação e Conceito Intermediário


uma pessoa pode ter várias reservas para o mesmo livro



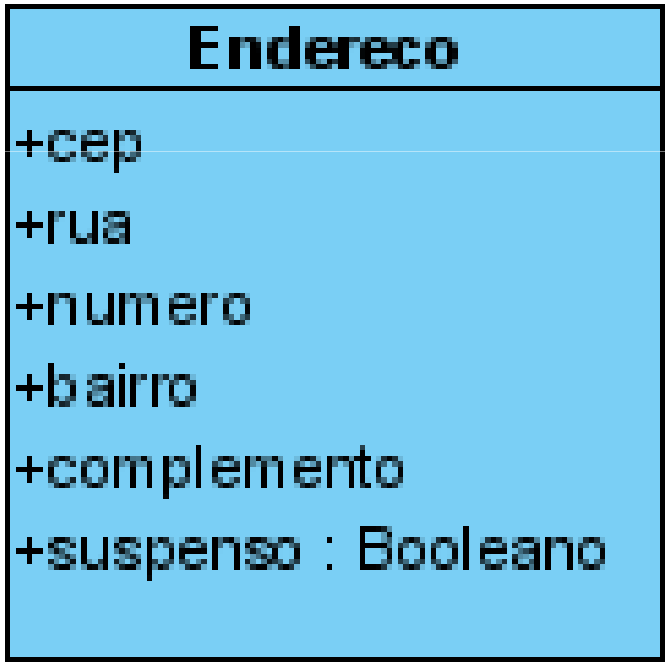

uma pessoa só pode ter no máximo uma única reserva para um mesmo livro

# Técnica Temporal: Classes modais

- Algumas classes definem objetos que mudam de estado.
- Formas de transição:
  - *Estável*: Apenas valores mudam.
  - *Monotônica crescente*: a instância ganha atributos ou associações com o tempo.
  - *Não-monotônica*: a instância pode ganhar ou perder atributos e associações com o tempo;

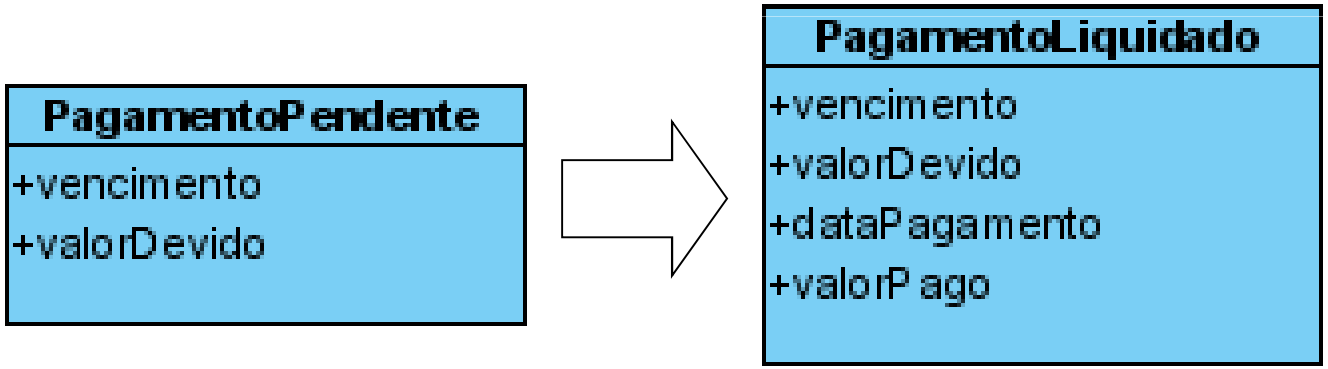
# Transição Estável

- Apenas valores mudam
- Exemplo: Estado "suspenso"

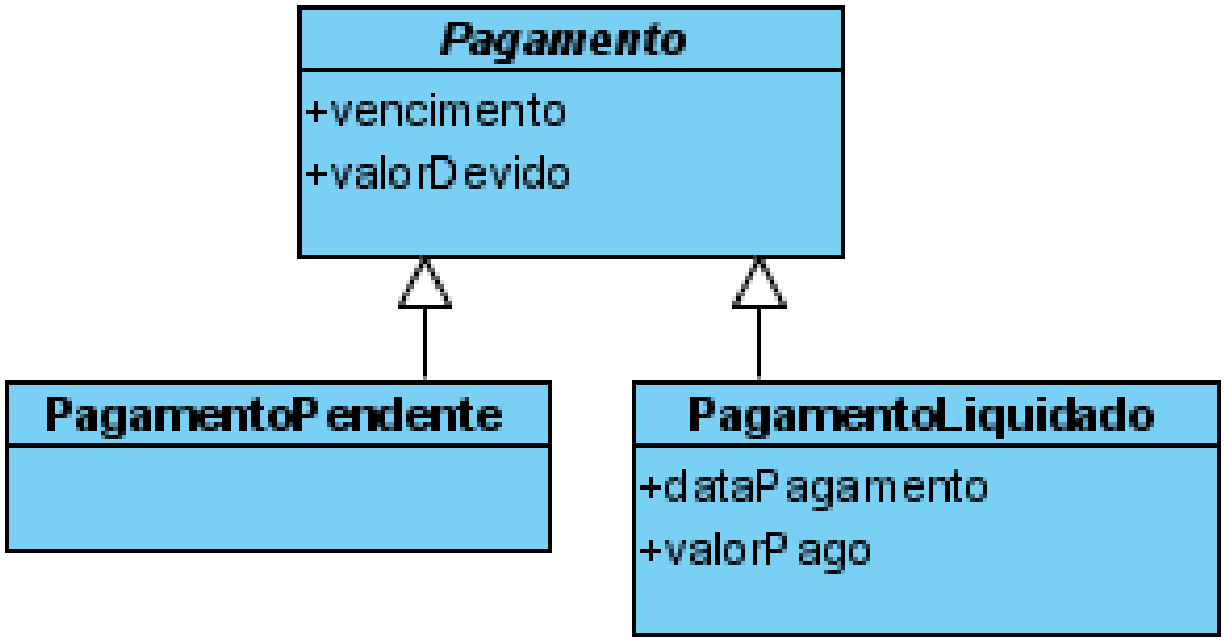


# Transição Monotônica

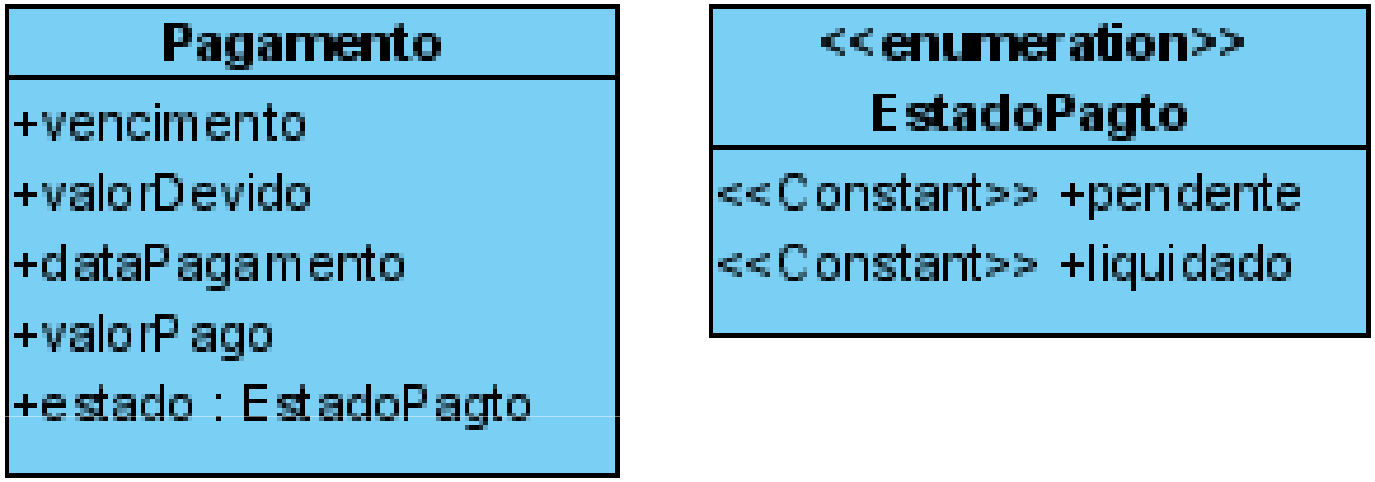
- Há ganho de atributos ou associações com as transições de estado.
- Exemplo:



# Solução Ruim com Herança

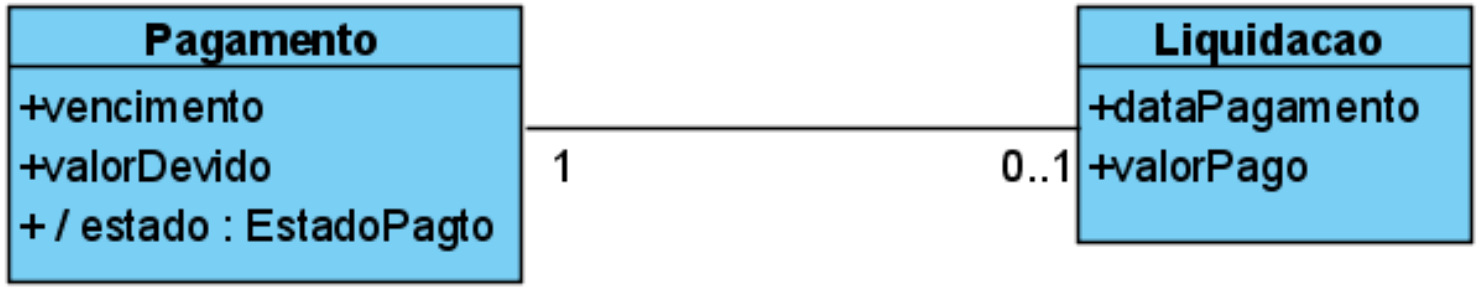


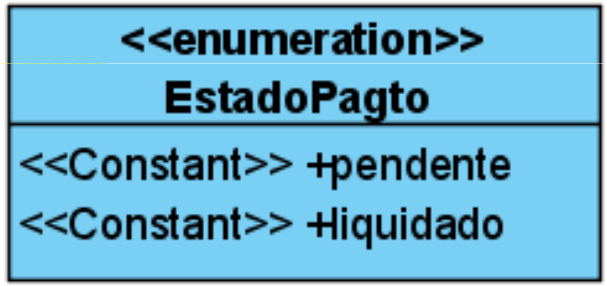
# Solução Ruim com Atributos Nulos




- Essa forma de modelagem ainda não é boa, pois gera classes com baixa coesão, e, portanto, com regras de consistência complexas que devem ser checadas freqüentemente.
- Essas classes com *baixa coesão* são altamente susceptíveis a erros de projeto ou programação.

# Solução Eficaz



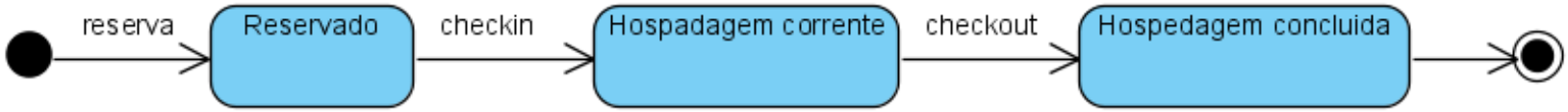

```
Context Pagamento::estado derive:
  if self.liquidacao.isNull() then
    EstadoPagto::pendente
  else
    EstadoPagto::liquidado
  endIf
```

# Transição Não-Monotônica

- Objetos podem ganhar ou perder atributos ou associações quando mudam de estado.
- Usa-se o padrão *State* (Estado)

# Exemplo

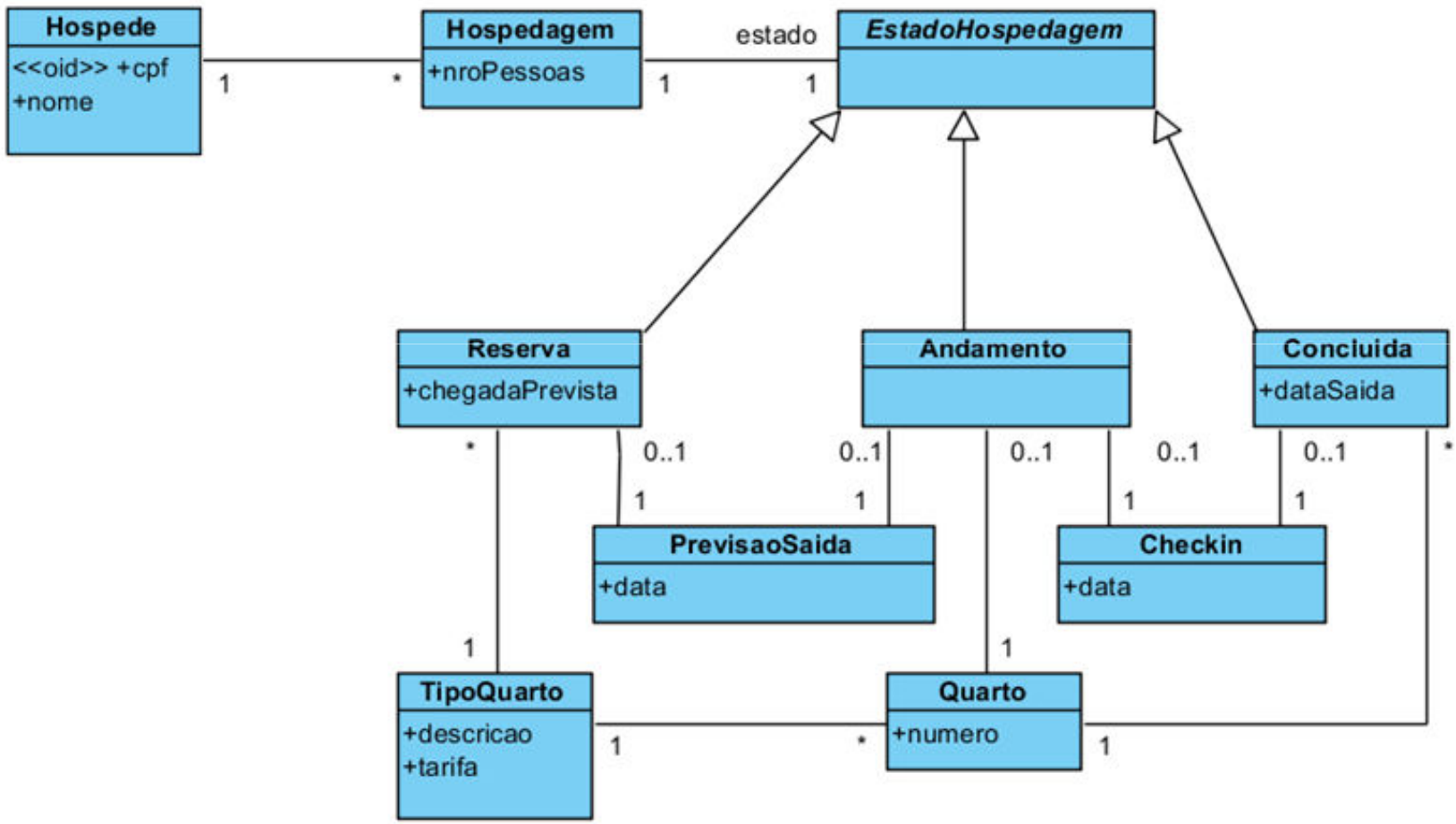
- Reserva/hospedagem em um hotel




- Se o caso fosse monotônico:


- Mas não é...

# Padrão Estado



# Padrões de Análise

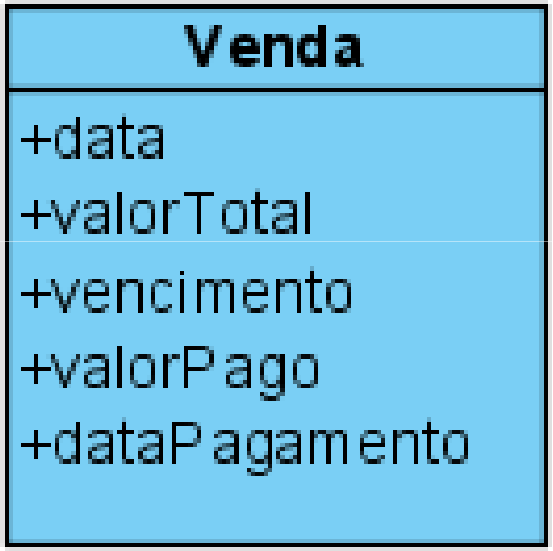
- São um subcaso dos padrões de projeto.
- Aplicam-se ao modelo conceitual.
- São sugestões e boas práticas, não regras.

# Padrão "Coesão Alta"

- Um conceito coeso é mais estável e reusável do que um conceito não coeso, que pode se tornar rapidamente confuso e difícil de manter.

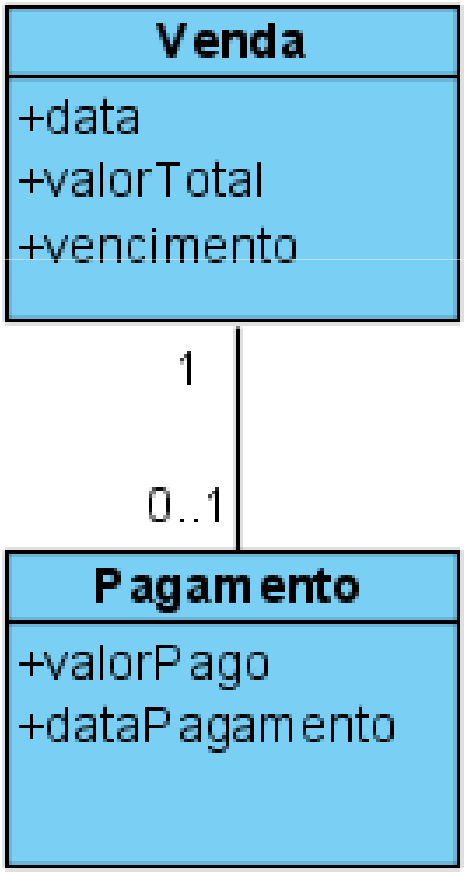
# Exemplo

- Baixa coesão:



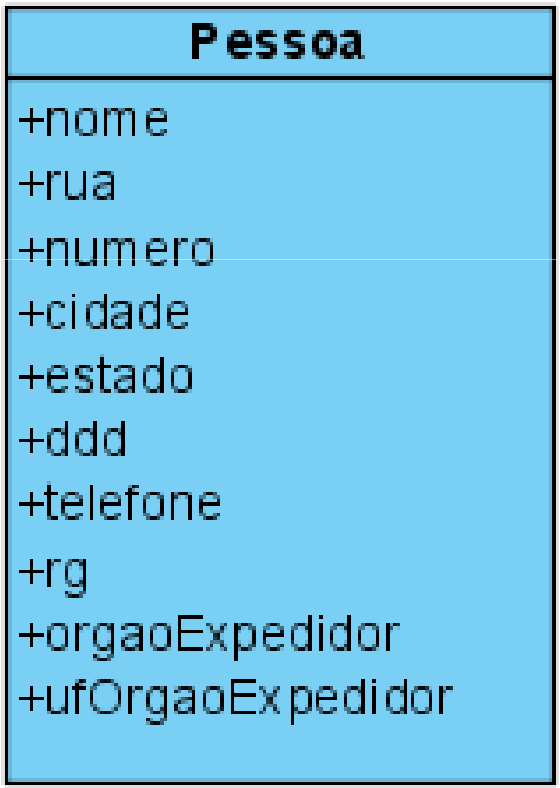

**Atributos dependentes de outros**

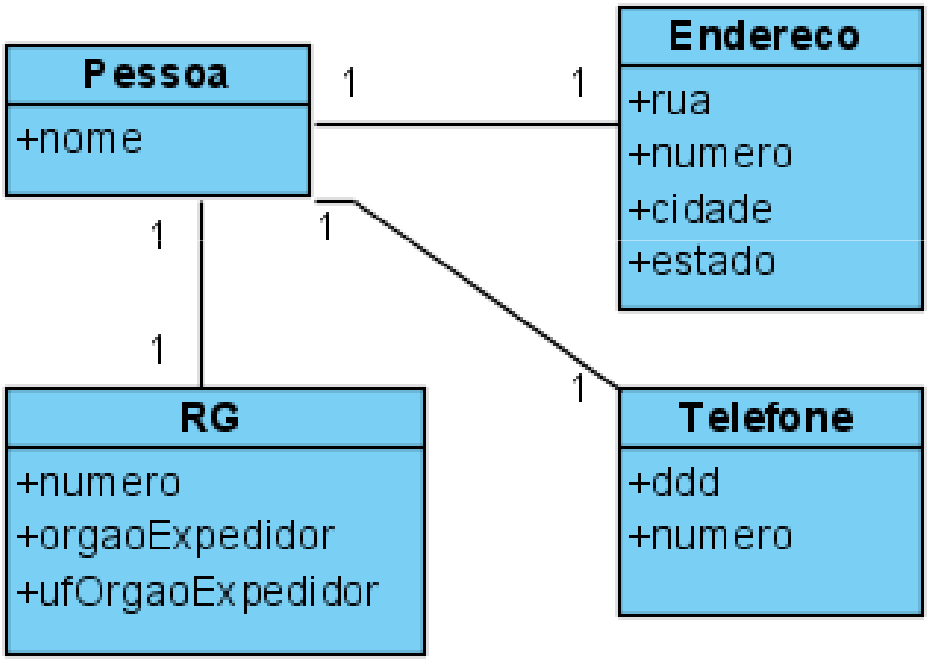
- Maior coesão



# Outro Exemplo

- Baixa coesão
- Maior coesão

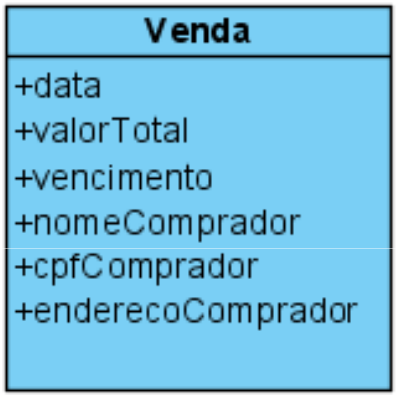


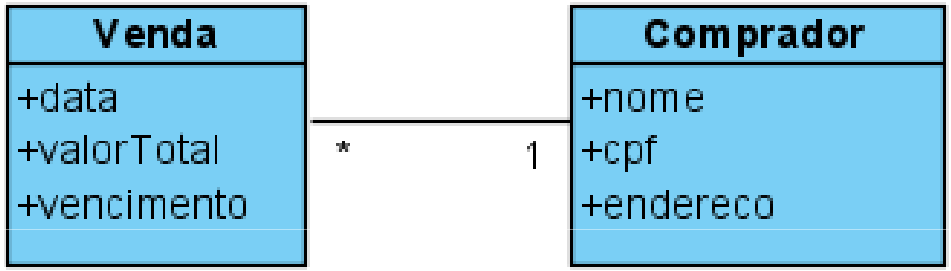




**Grupos de atributos fortemente correlacionados.**

# Mais um Exemplo

- Baixa coesão
- Maior coesão

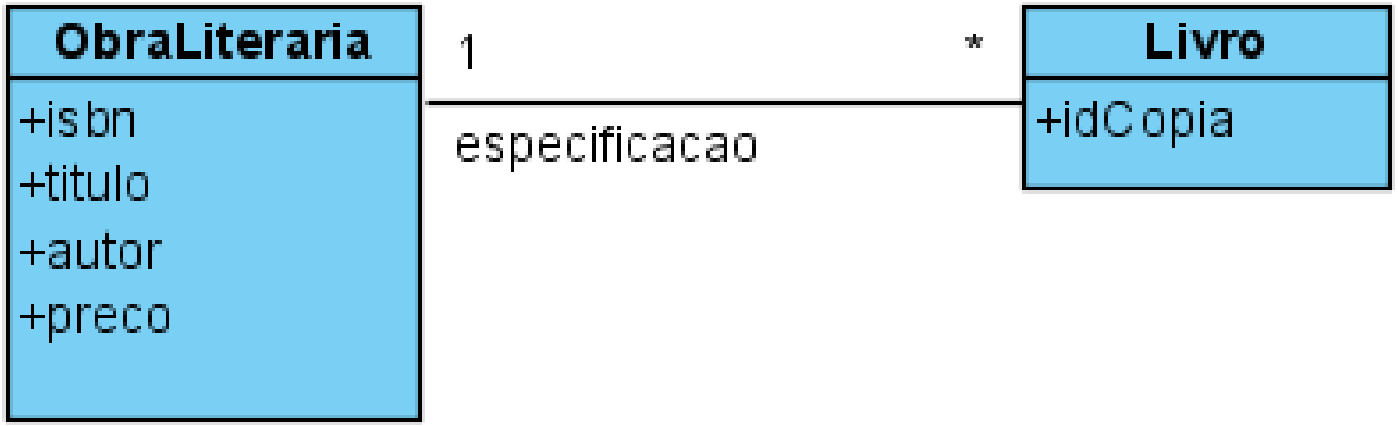


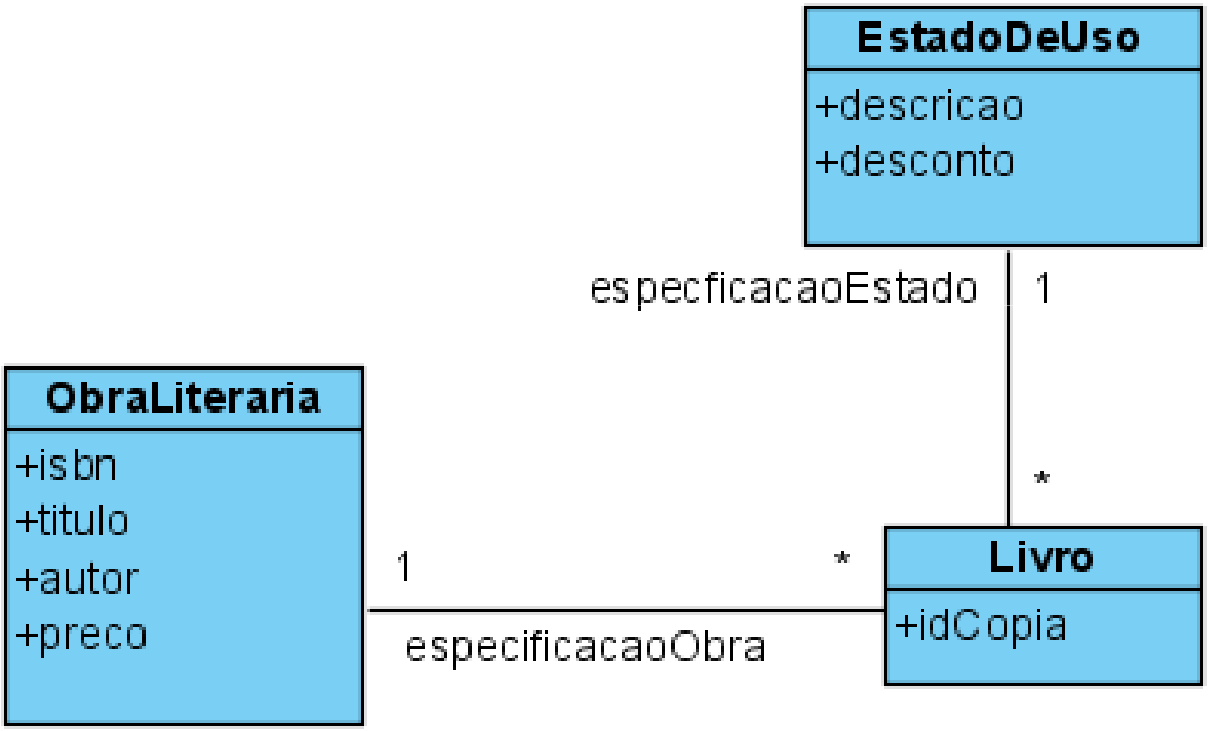

**Atributos que repetem valores nas instâncias**

# Padrão "Classes de Especificação"

- Existem dois domínios de discurso:
  - Nível concreto: as coisas
  - Nível de conhecimento: as descrições das coisas



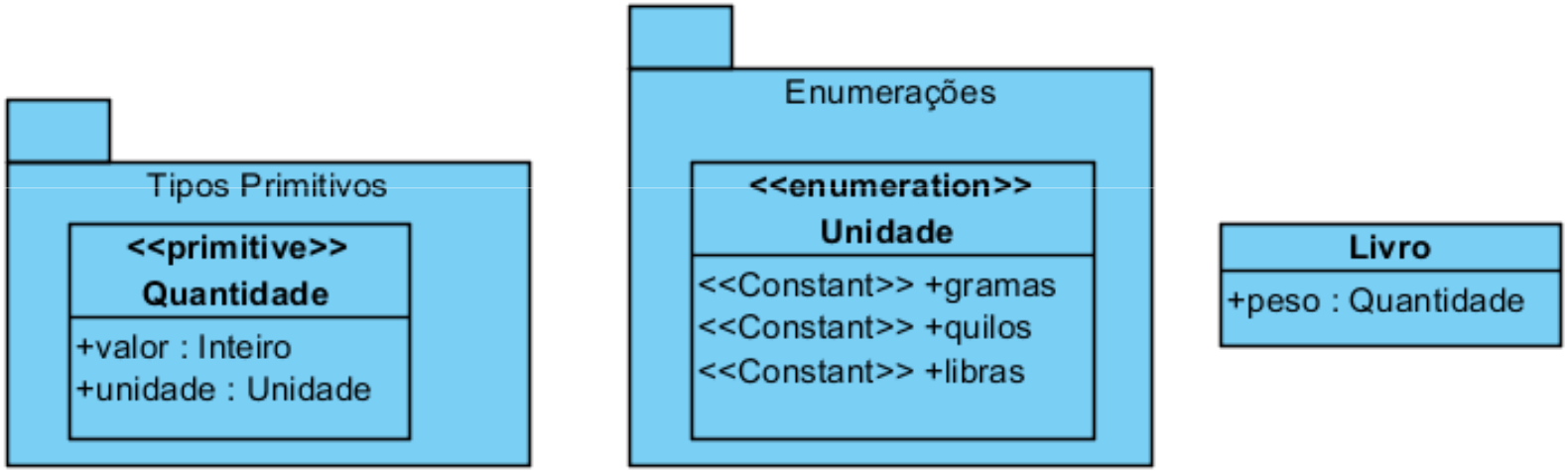
# Exemplo de Dupla Especificação



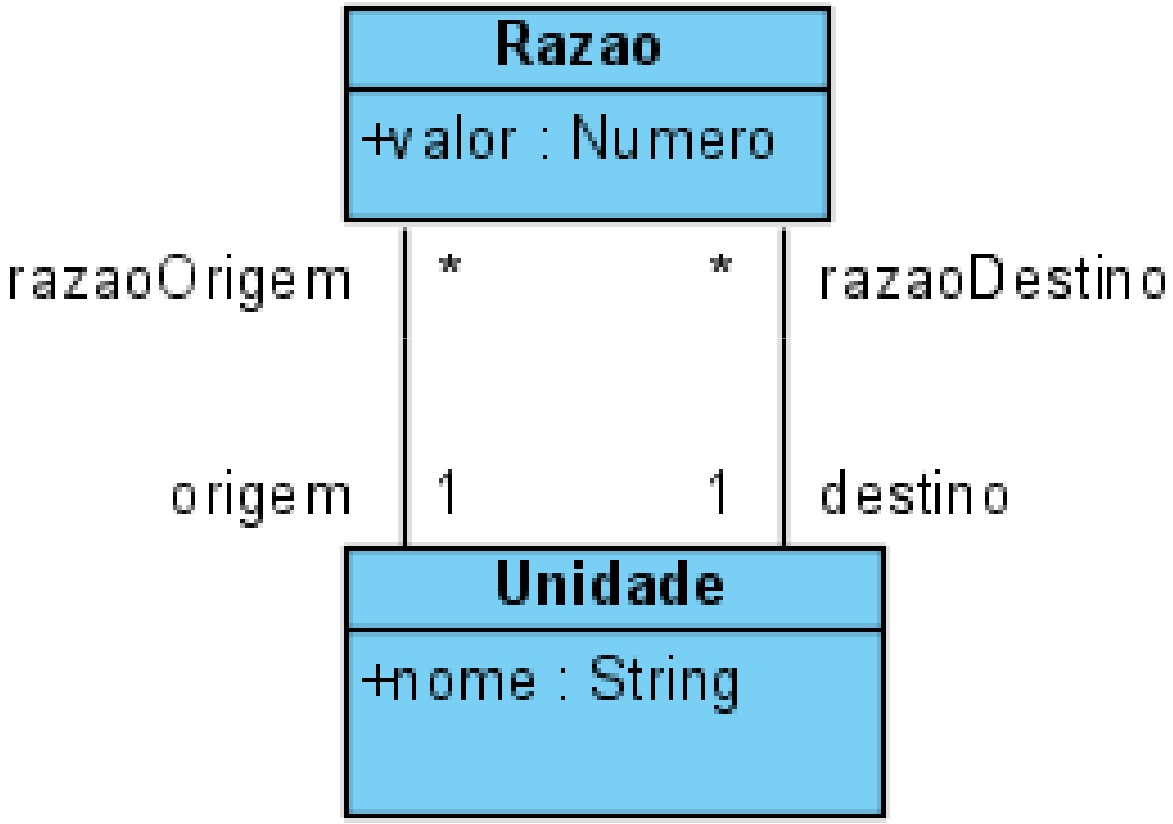
# Padrão "Quantidade"

- Peso de um livro: 400
- 400 o que?
- Solução rasteira:
  - peso:Gramas
- Inflexível

# Definição e Uso de uma Quantidade



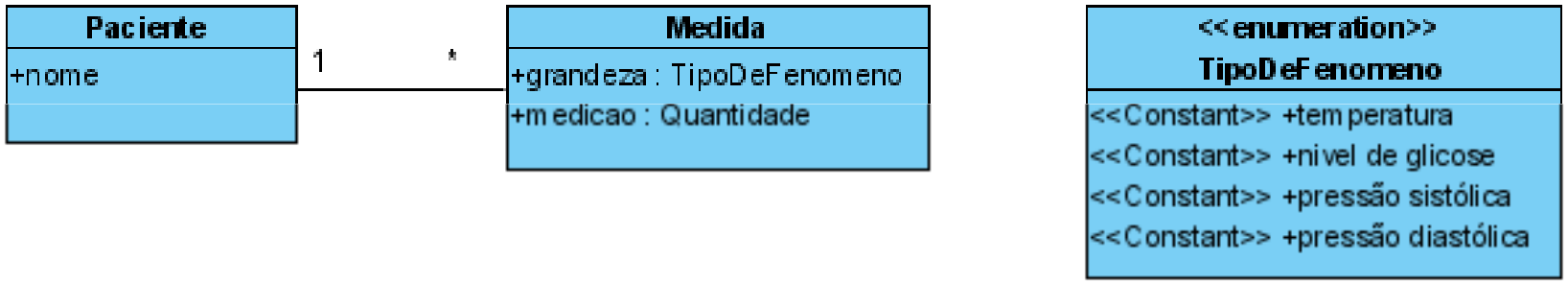
# Quantidade com Razão de Conversão



# Padrão "Medida"

- Um paciente no hospital pode ter inúmera medidas corporais feitas de tempos em tempos.
- Como modelar isso?
- Um *tabelão* esparso?

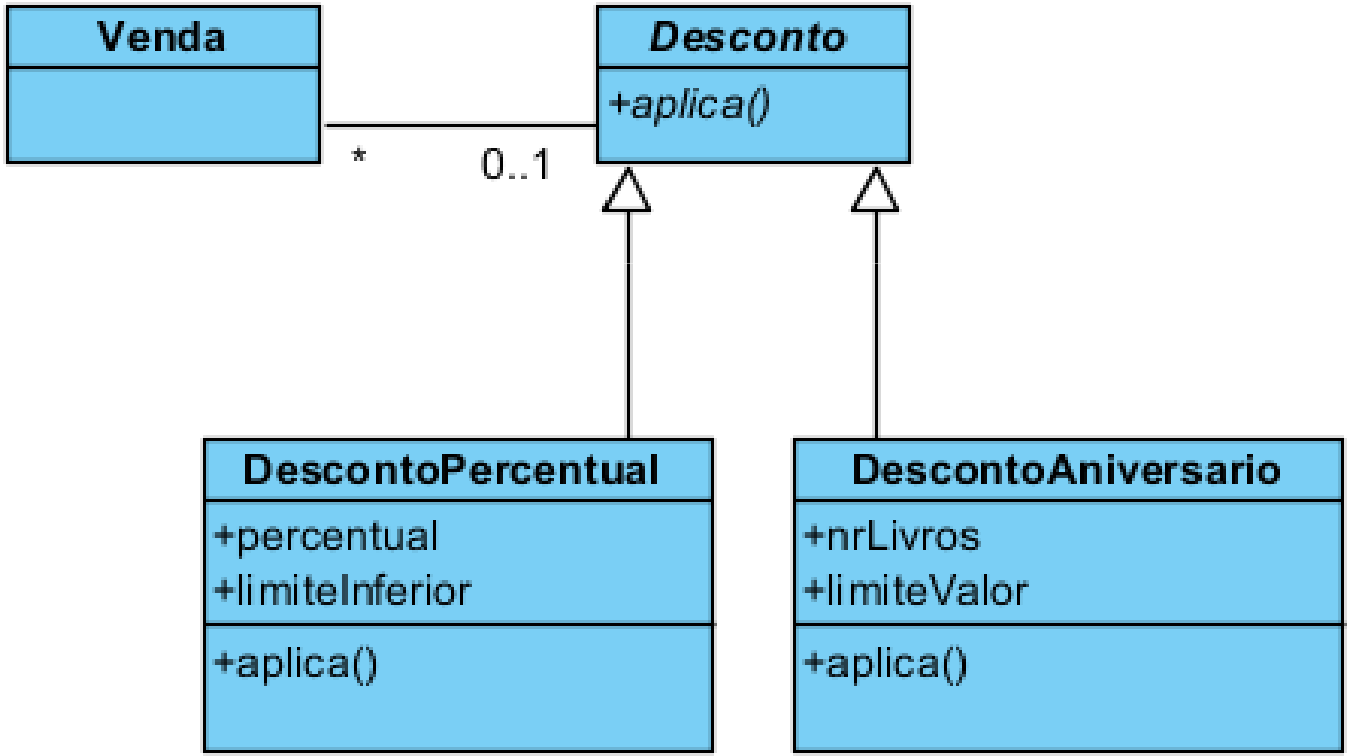
# Exemplo de Uso do Padrão Medida



# Padrão "Estratégia"

- Separar estratégias dos conceitos.
- Formas de calcular impostos mudam.
- Formas de dar desconto mudam.
- Exemplos:
  - Um livro grátis de até 50 reais para compras acima de 300 reais.
  - 20% de desconto em até dois livros no dia do aniversário do comprador.
  - 5% de desconto nos livros de suspense nas sextas-feiras treze.

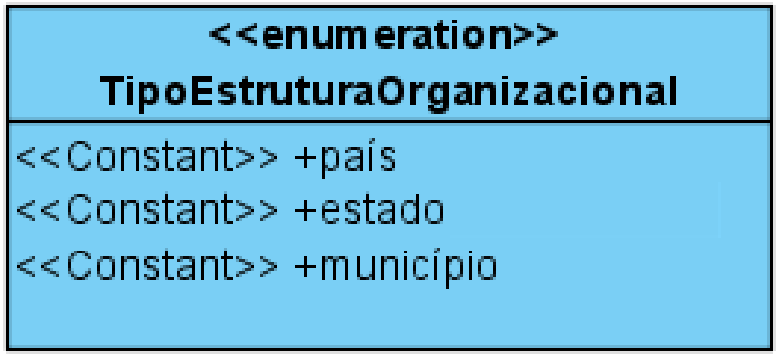
# Exemplo de Aplicação do Padrão Estratégia



# Padrão "Hierarquia Organizacional"

- Hierarquias nem sempre se comportam bem:

# Aplicação de Estrutura Organizacional

# Padrão "Junção de Objetos"

- Por vezes é necessário unir objetos com duas ou mais representações.
- Técnicas:
  - Copiar e Substituir (*copy and replace*)
  - Sucessor (*superseding*)
  - Essência/Aparência (*essence appearance*)

# Copiar e Substituir

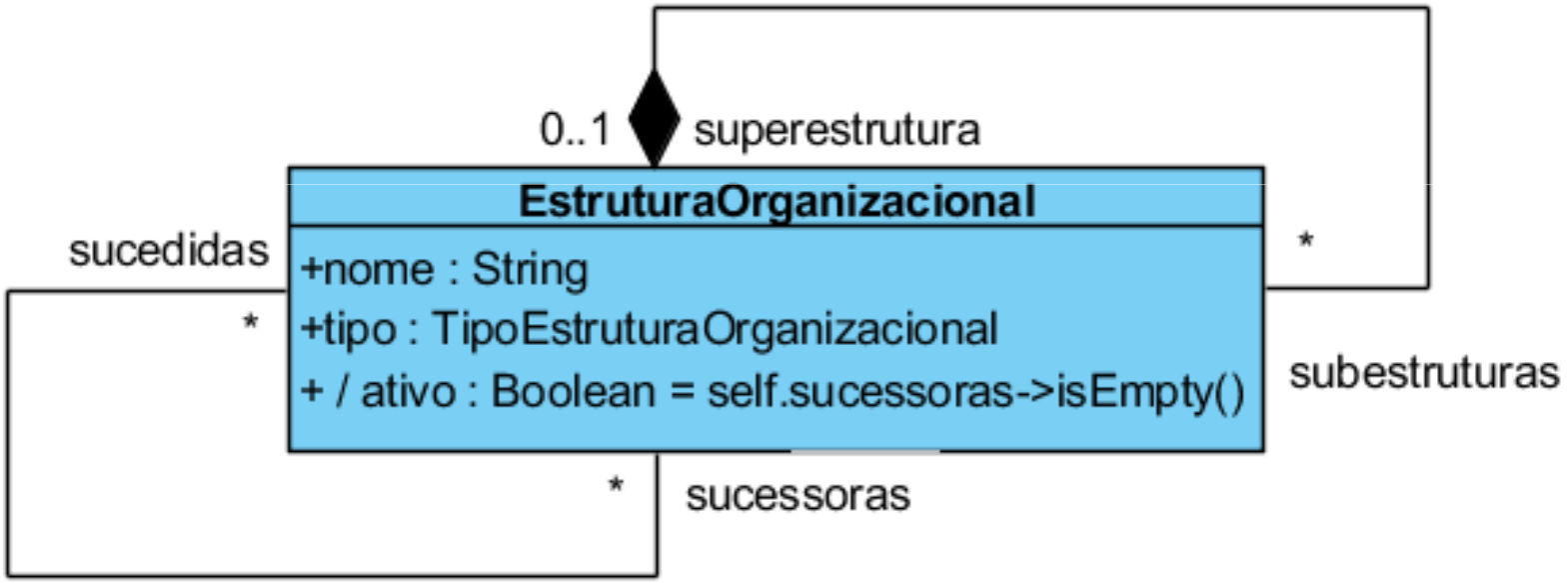
- Aplicável quando um erro de registro cria dois objetos para o mesmo elemento real.
- Copia-se um dos objetos sobre o outro.
- Elimina-se o primeiro objeto.
- Substitui-se referências e associações.

- Os objetos originais são perdidos.

# Sucessor

- Aplicável quando um ou mais objetos sucedem outros no tempo.
- Exemplo, Arena, PDS, PFL, Democratas, etc...
- Exemplo: Estados do Mato Grosso e Mato Grosso do Sul

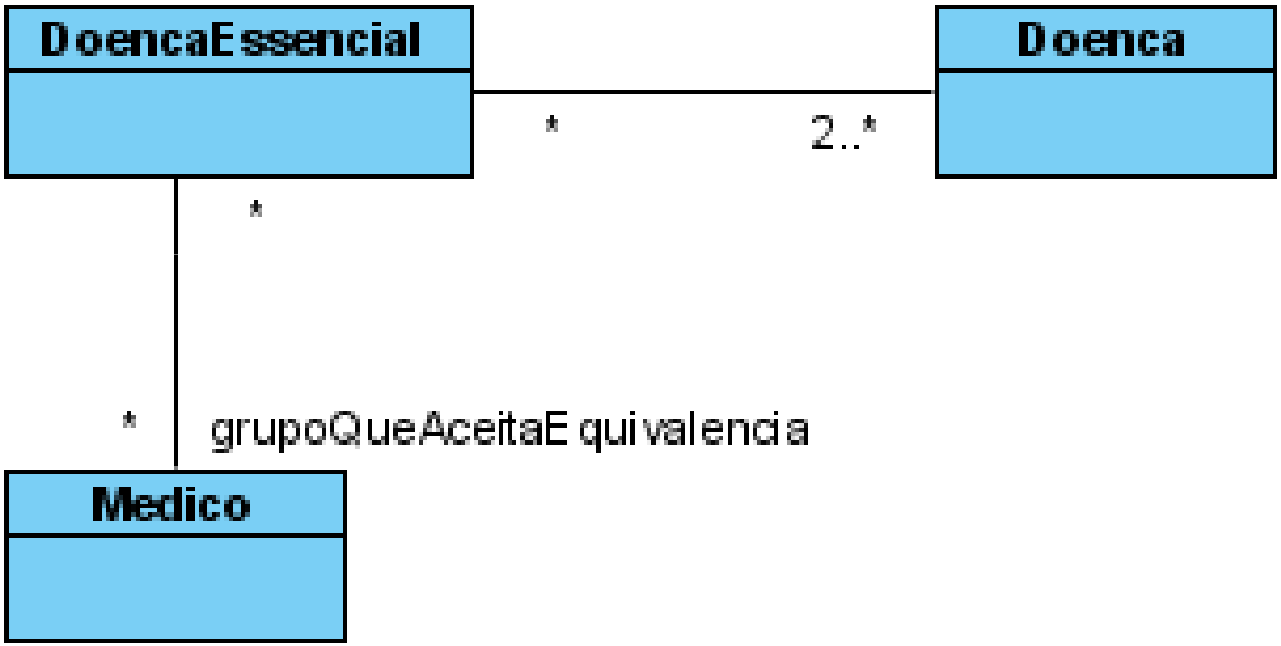
# Aplicação de Sucessor



# Essência/Aparência

- Usado quando objetos são equivalentes mas continuam existindo individualmente.
- Cria-se um objeto essência.
- Exemplo: hierarquias paralelas.

# Variante: Equivalência



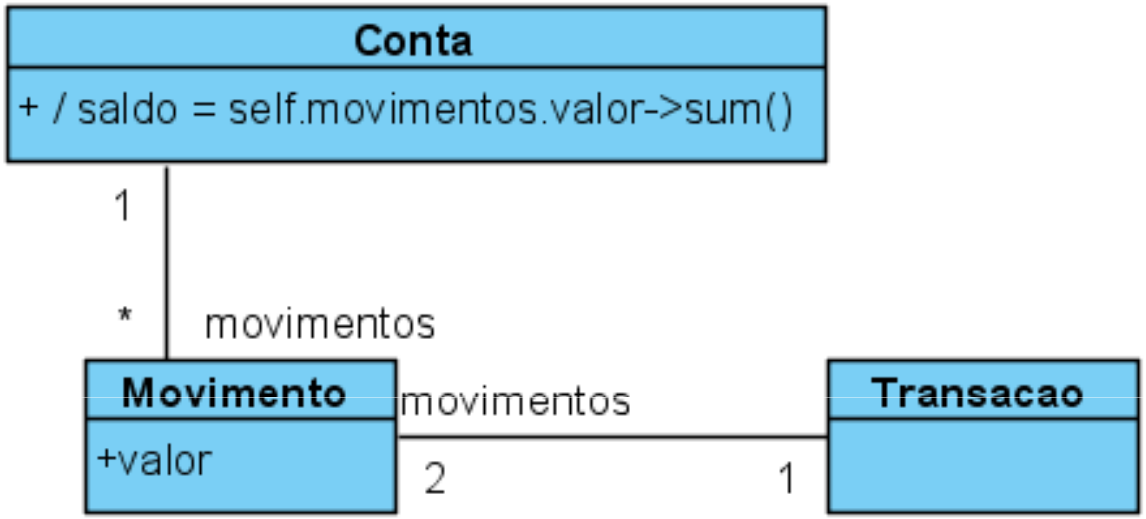
# Desfazendo a Junção

- Copiar e substituir: necessário *log*.
- Sucessor e Essência/Aparência: remover a ligação ou objeto essência.

# Padrão "Conta/Transação"

- Como modelar um controle de estoque com:
  - Contas a pagar e a receber
  - Vendas e comissões
  - Entrada e saída de mercadoria
  - Pedidos de compra e venda pendentes
  - Devoluções
  - Etc.
- Com um único padrão?

# Conta/Transação



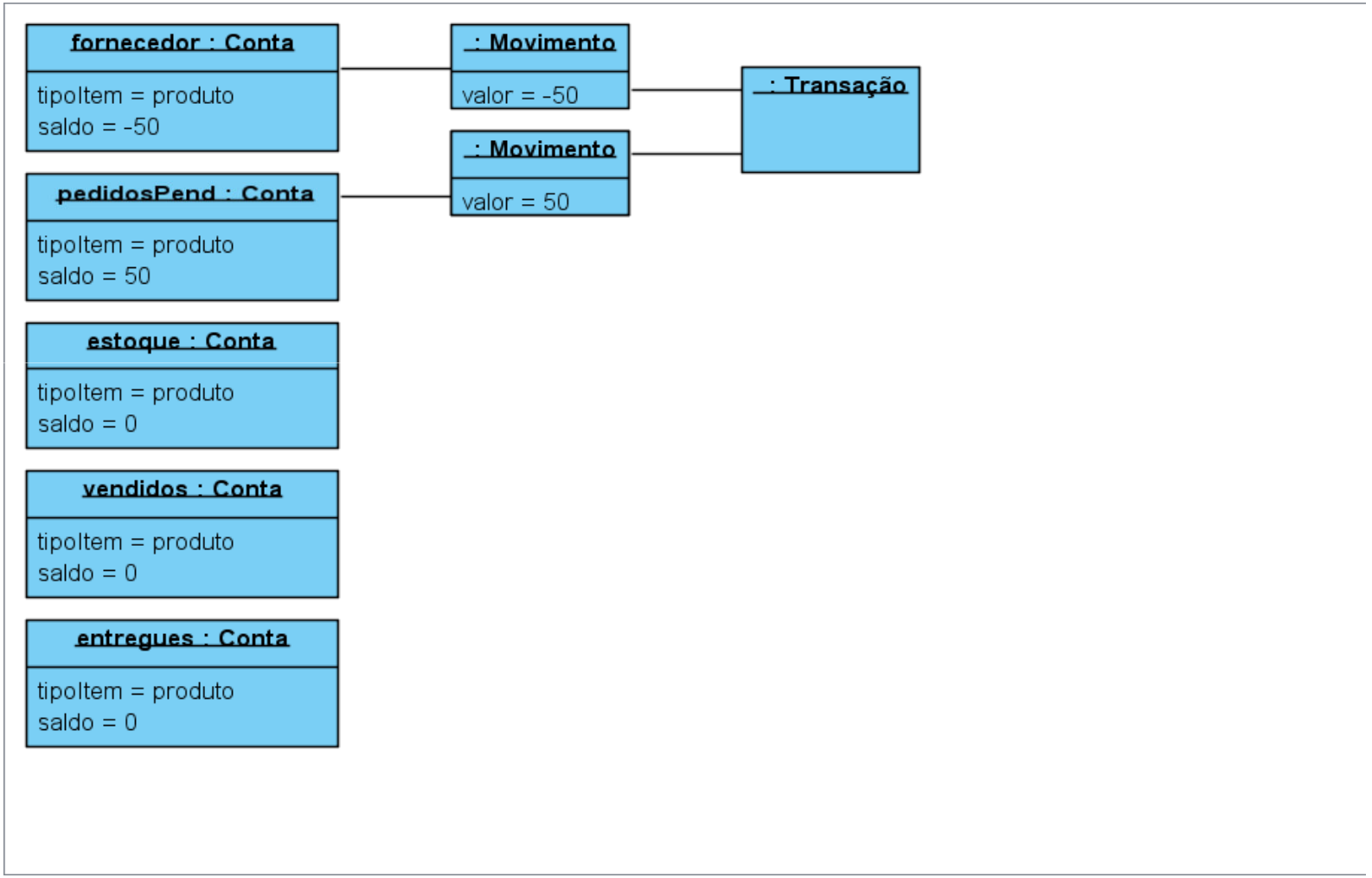

```
Context Transacao inv:
  self.movimentos.valor→sum() = 0
```

# Todas as facetas do problema são instâncias de Conta

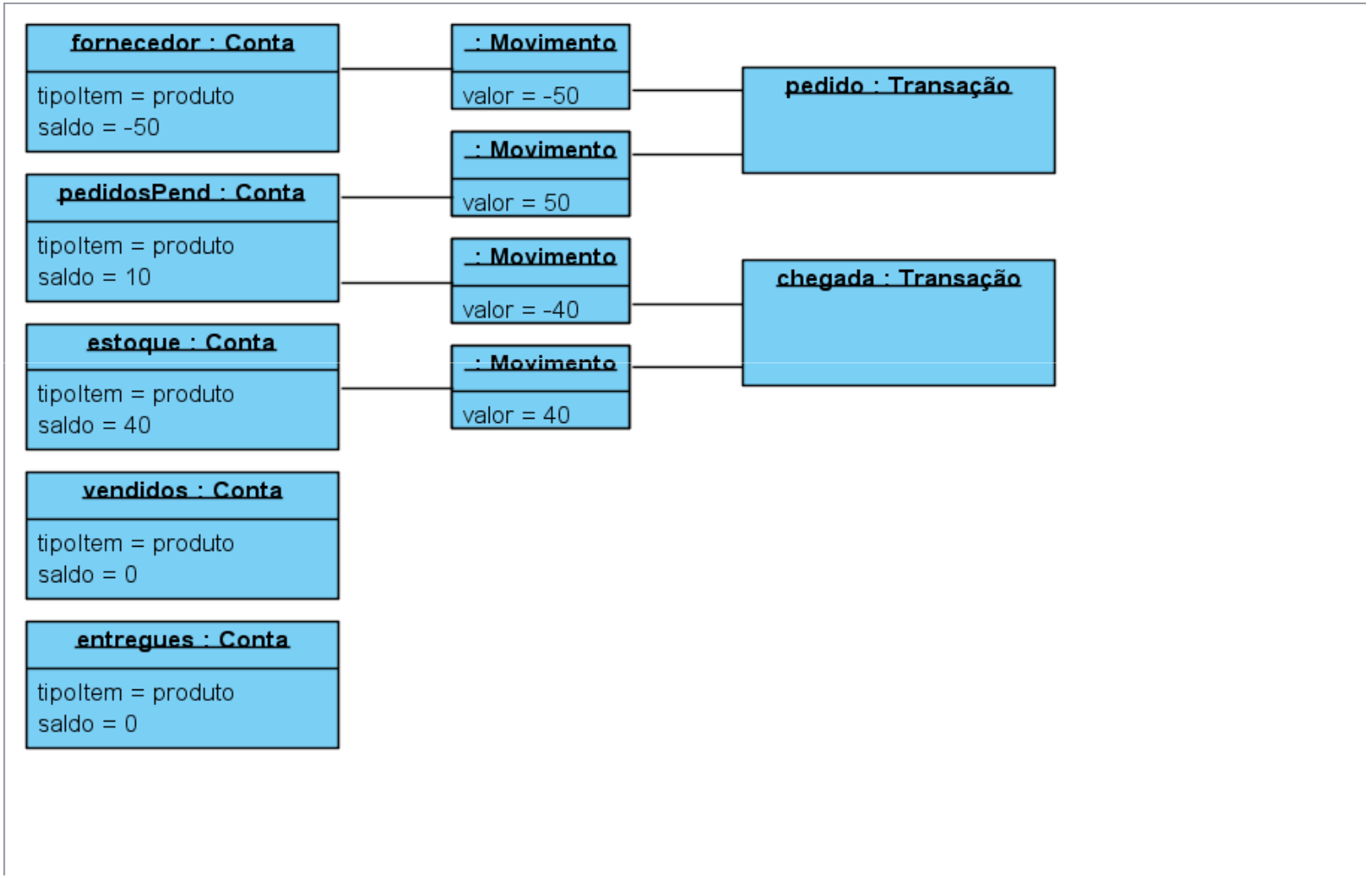
- Conta corrente
- Contas a pagar
- Contas a receber
- Estoque
- Produtos vendidos
- Produtos enviados
- Produtos comprados
- Produtos recebidos
- Produtos devolvidos
- Etc.

# Exemplo: situação inicial

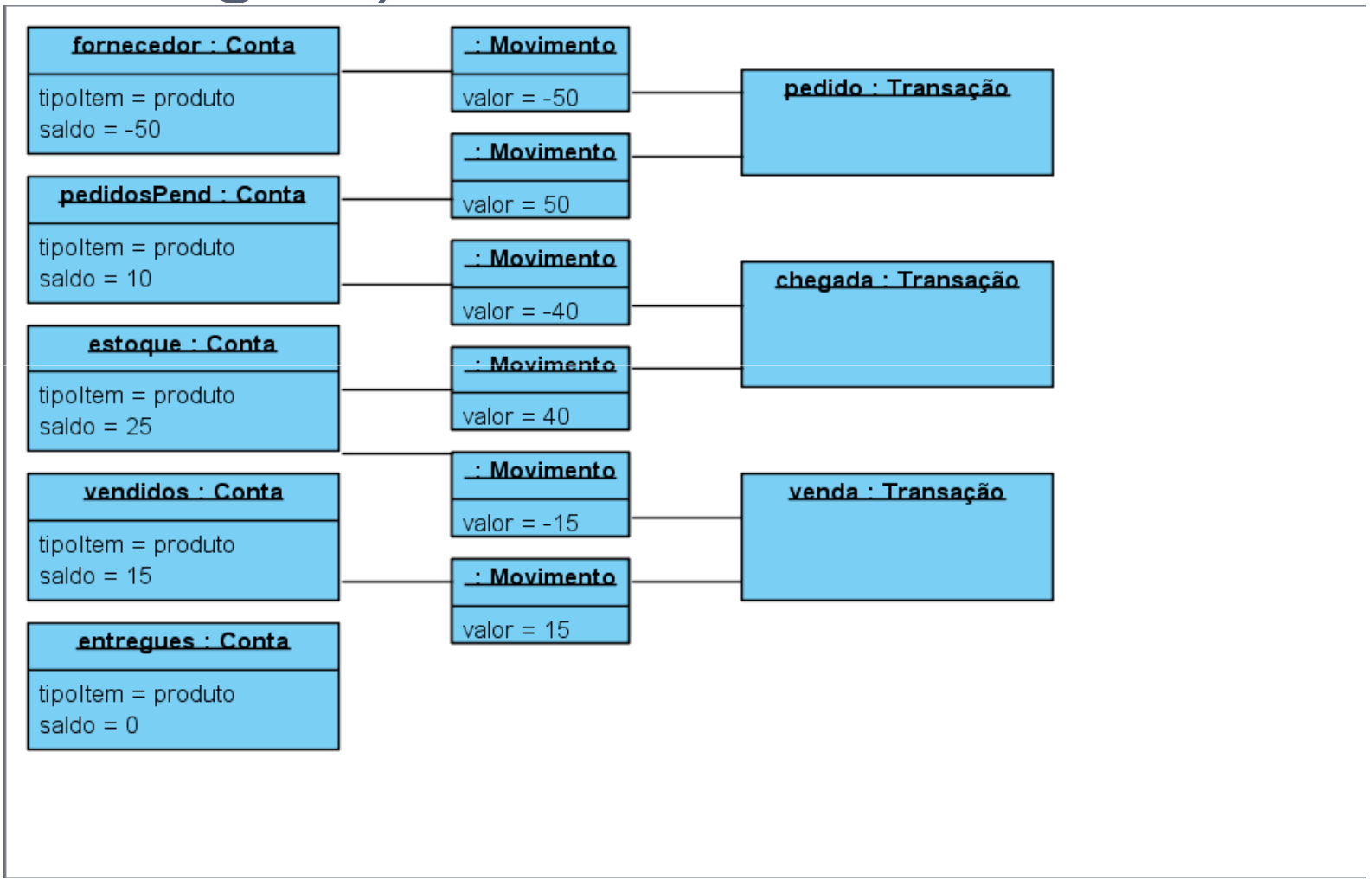
# Após fazer um pedido de 50 itens



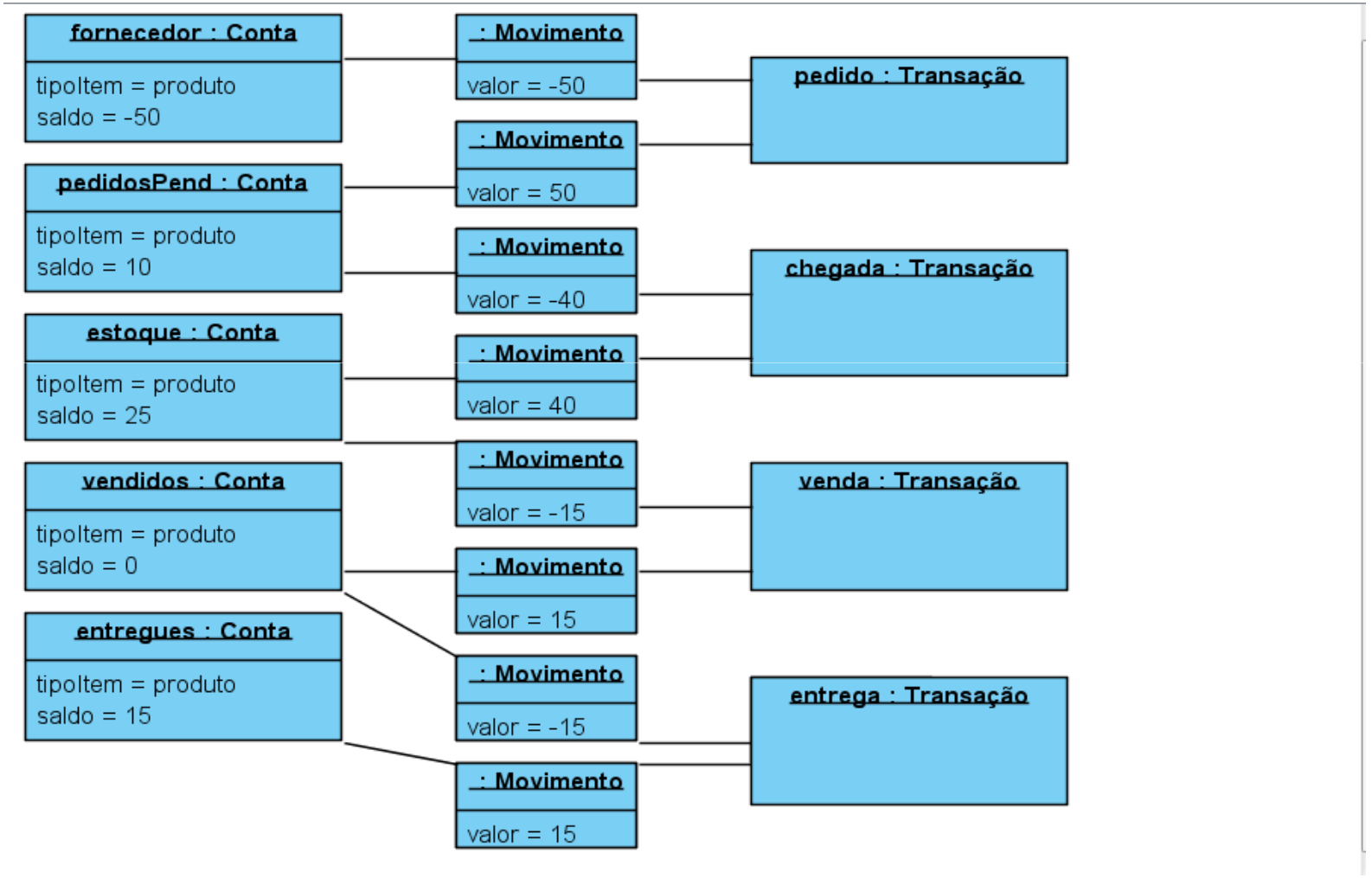
# Apenas 40 itens chegaram



# 15 foram vendidos (mas ainda não entregues)



# 15 produtos são entregues

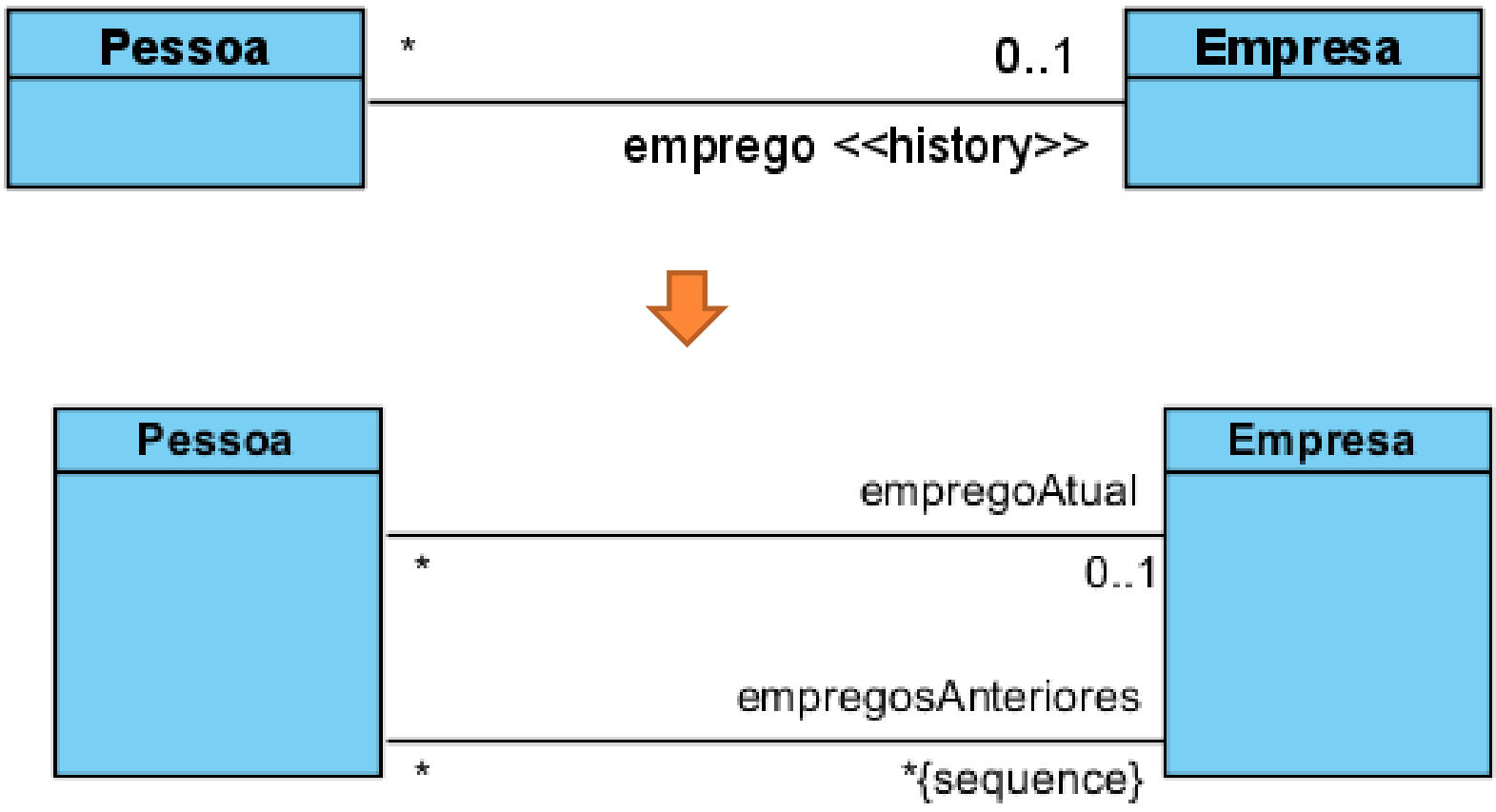


# Padrão "Associação Histórica"

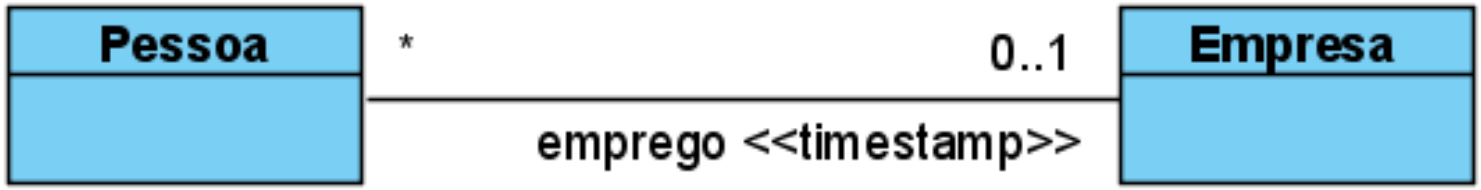
- Associações que representam o presente e o passado.


- *Getters*:
  - getEmprego() -- atual
  - getEmprego(index) -- histórico

# Implementação de <<history>>

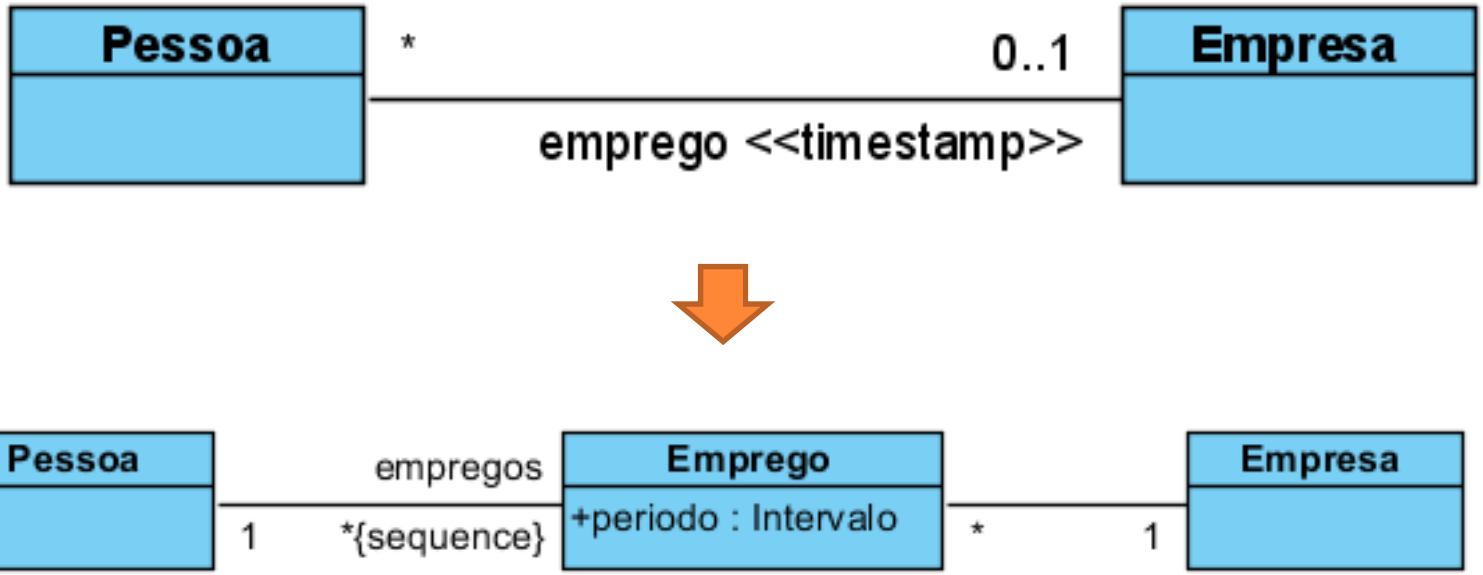


# Variação com Registro de Tempo




- Getters:
  - getEmprego() -- atual
  - getEmprego(indice) -- série histórica
  - getEmprego(time) – ponto no tempo

# Implementação da Variação



# Padrão "Intervalo"

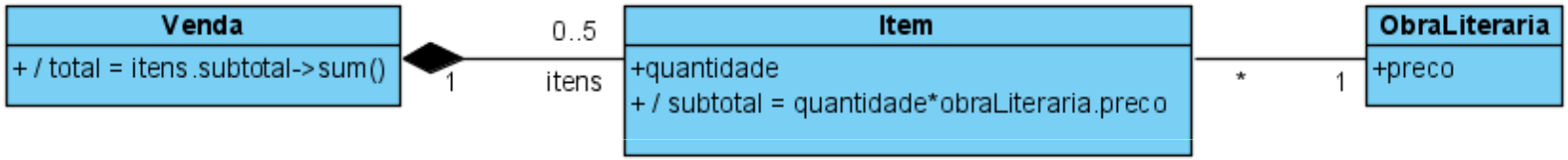
- Não usar atributos de início e fim.
- Usar tipo primitivo "Intervalo".
- Evita baixa coesão.
- Permite reusar operações típicas de intervalos:
  - Verificar se um valor está dentro do intervalo.
  - Verificar se dois intervalos se interceptam.
  - Etc.

# Invariantes

- Existem situações onde a expressividade gráfica do diagrama de classes é insuficiente para representar determinadas regras do modelo conceitual.
- Nestes casos necessita-se fazer uso de invariantes.
- *Invariantes* são restrições sobre as instâncias e classes do modelo

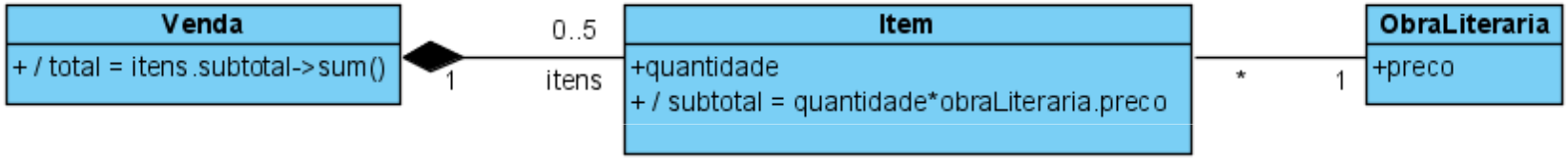
# Uma invariante que pode ser representada graficamente

- Uma venda não pode ter mais de 5 itens:



# Uma invariante que *não* pode ser representada graficamente

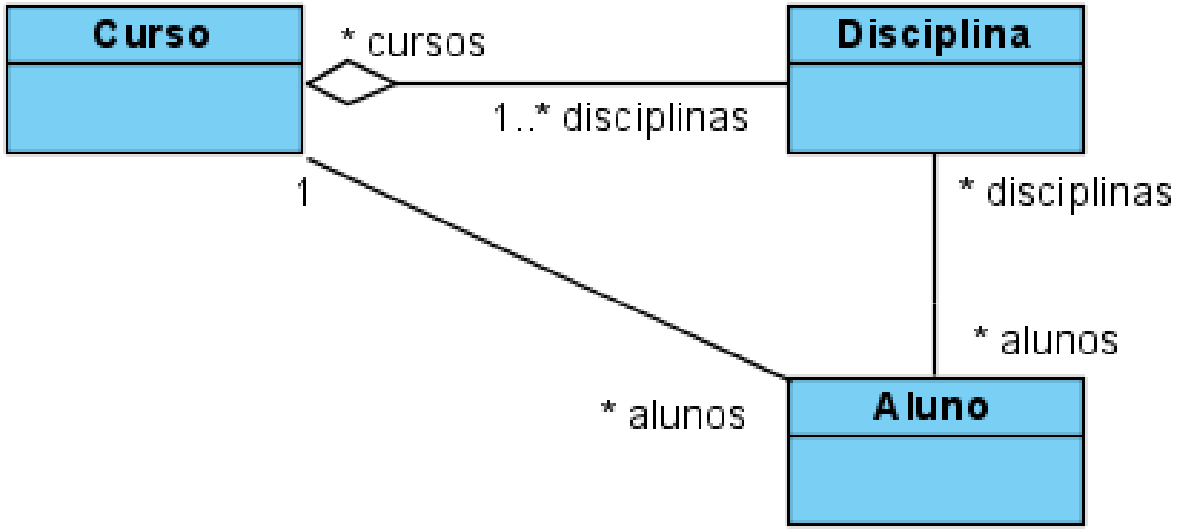
- "nenhuma venda pode ter valor superior a mil reais"




```
Context Venda inv:
  self.total <= 1000,00

Context Venda::total
  derive: self.itens->sum(x|x.subtotal)
Context Item::subtotal
  derive:
self.quantidade*self.obraLiteraria.preco
```

# Uso de invariante para relacionar associações




```
Context Aluno inv:
  self.disciplinas→forAll(d|
    d.cursos→includes(self.curso)
)
```

# Discussão

- Um bom modelo conceitual produz um banco de dados organizado e normalizado.
- Um bom modelo conceitual incorpora regras estruturais que impedem que a informação seja representada de forma inconsistente.

# Discussão

- Um bom modelo conceitual vai simplificar o código gerado porque não será necessário fazer várias verificações de consistência que a própria estrutura do modelo já garante.
- O uso de padrões corretos nos casos necessários simplifica o modelo conceitual e torna o sistema mais flexível e, portanto, lhe dá maior qualidade.

# Discussão

- Apenas é necessário sempre ter em mente que só vale a pena criar um padrão quando os benefícios deste compensam o esforço de registrar sua existência.