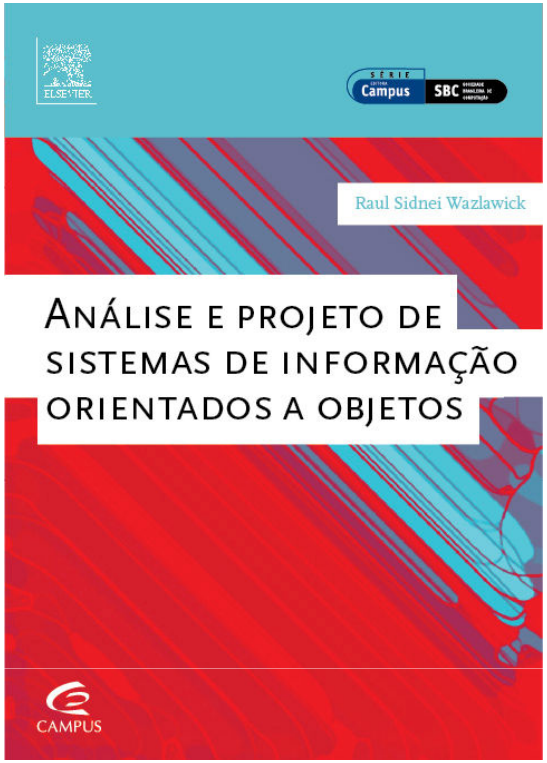


GERAÇÃO AUTOMÁTICA DE CÓDIGO EXECUTÁVEL A PARTIR DE CONTRATOS OCL

Prof. Raul Sidnei Wazlawick

TUTORIAL

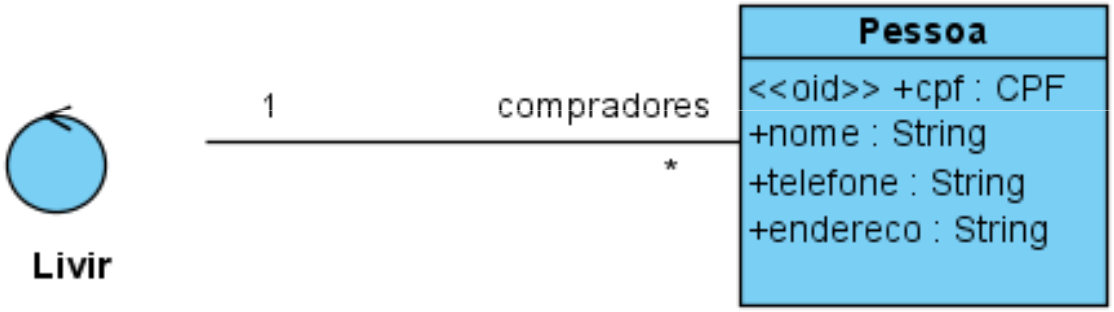
- 1. Geração de código a partir do modelo conceitual
  - Diagrama de Classes
- 2. Geração de código a partir do modelo funcional
  - Contratos de Operação de Sistema

2.1 CONTRATOS

- Operação de Sistema
  - Pré-condições (opcional)
  - Pós-condições
- Consulta de Sistema
  - Pré-condições (opcional)
  - Resultados

2.1.1 Pré-condições

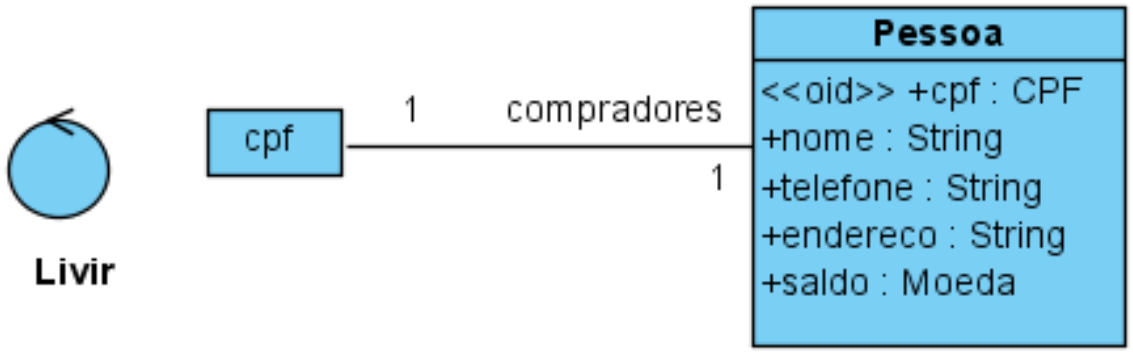
- Estabelecem o que é verdadeiro quando uma operação ou consulta de sistema for executada.




```
Context Livir::identificaComprador(umCpf)

  pre:
    self.compradores→select(p|p.cpf=umCpf)→notEmpty()
```

... COM ASSOCIAÇÃO QUALIFICADA




```
Context Livir::identificaComprador(umCpf)

  pre:
    self.compradores[umCpf]→notEmpty()
```

2.1.2 Retorno de Consulta

- Ex.: Saldo do comprador corrente:

```
Context Livir::saldoCompradorCorrente():Moeda

  body:
    compradorCorrente.saldo
```

2.1.3 Pós-Condições

- As pós-condições estabelecem o que muda nas informações armazenadas no sistema após a execução de uma operação de sistema.

```
Context Livir::operacaoX()
  post:
    <expressão 1> AND
    <expressão 2> AND
       ...
    <expressão n>
```

Tipos de pós-condições

- Modificação de valor de atributo.
- Criação de instância.
- Criação de associação.
- Destruição de instância.
- Destruição de associação.

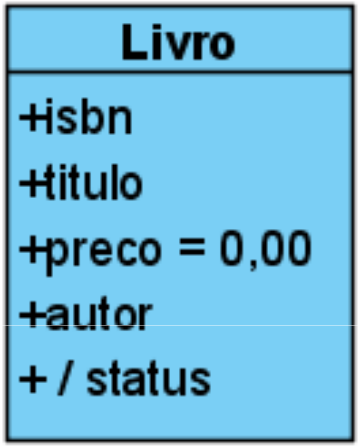
Operações básicas

- São aquelas que em orientação a objetos operam diretamente sobre os elementos básicos do modelo conceitual:
  - Classes
  - Atributos
  - Associações
- Seu significado e comportamento são definidos por padrão.
- Elas realizam as pós-condições.

Modificação de valor de atributo

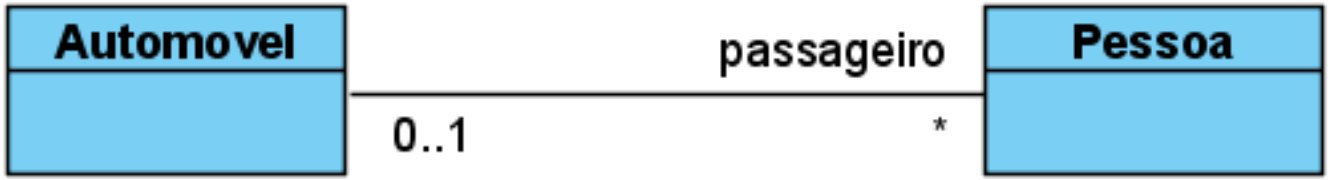
```
pessoa^setDataNascimento(novaData)
```

CRIAÇÃO DE INSTÂNCIA




```
Livro::newInstance()
```

CRIAÇÃO DE ASSOCIAÇÃO

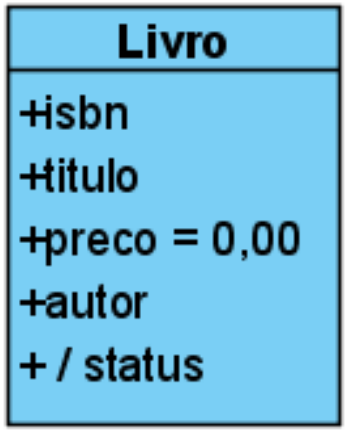



```
umAutomovel^addPassageiro(umaPessoa)

Ou:

umaPessoa^addAutomovel(umAutomovel)
```

EXEMPLO DE CONTRATO COMPLETO:




```
Context Livir::criaLivro(umIsbn, umTitulo, umAutor)

  def: novoLivro =
    Livro::newInstance()

  post:
    self^addLivro(novoLivro) AND
    novoLivro^setIsbn(umIsbn) AND
    novoLivro^setTitulo(umTitulo) AND
    novoLivro^setAutor(umAutor)
```

DESTRUIÇÃO EXPLÍCITA DE INSTÂNCIA

```
objeto^destroy()
```

Assume-se que todas as associações deste objeto também são destruídas com ele

DESTRUIÇÃO DE ASSOCIAÇÃO


```
umaVenda^removePagamento(umPagamento)

Ou:

umPagamento^removeVenda()
```

Duas conseqüências possíveis:

1. O pagamento p1 também é destruído.
2. O pagamento p1 é associado a outra venda.

PÓS-CONDIÇÕES BEM FORMADAS

- Considerando-se que as operações básicas mais elementares não comportam checagem de consistência nos objetos em relação ao modelo conceitual, o conjunto de pós-condições precisa ser verificado para se saber se ao final da execução das operações os objetos estão em um estado consistente com as definições do modelo.

Regra 1

- Uma instância recém criada deve ter pós-condições indicando que todos seus **atributos** foram **inicializados**, exceto:
  - Atributos derivados (que são calculados).
  - Atributos com valor inicial (que já são definidos por uma cláusula *init* no contexto da classe e não precisam ser novamente definidos para cada operação de sistema).
  - Atributos que possam ser nulos (neste caso a inicialização é opcional).

Regra 2

- Uma instância recém criada deve ter sido **associada** a alguma outra que por sua vez possua um caminho de associações que permita chegar à controladora de sistema.
- Caso contrário ela é **inacessível** e não faz sentido criar um objeto que não possa ser acessado por outros.

Regra 3

- Todas as associações afetadas por criação ou destruição de instância ou associação devem estar com seus papeis dentro dos **limites inferior e superior**.

Regra 4

- Todas as **invariantes** afetadas por alterações em atributos, associações ou instâncias devem continuar sendo verdadeiras.

2.2 Modelagem Dinâmica

Responsabilidades de objetos

	Conhecer	Fazer
Sobre si mesmo	Consultar atributos getAtributo()	Modificar atributos setAtributo(valor)
Sobre suas vizinhanças	Consultar associações getPapel()	Modificar associações addPapel(umObjeto) removePapel(umObjeto)
Outros	Consultas gerais e Associações e atributos derivados consultaInformação() getAssociacaoDerivada() getAtributoDerivado()	Métodos delegados -- nomes variam

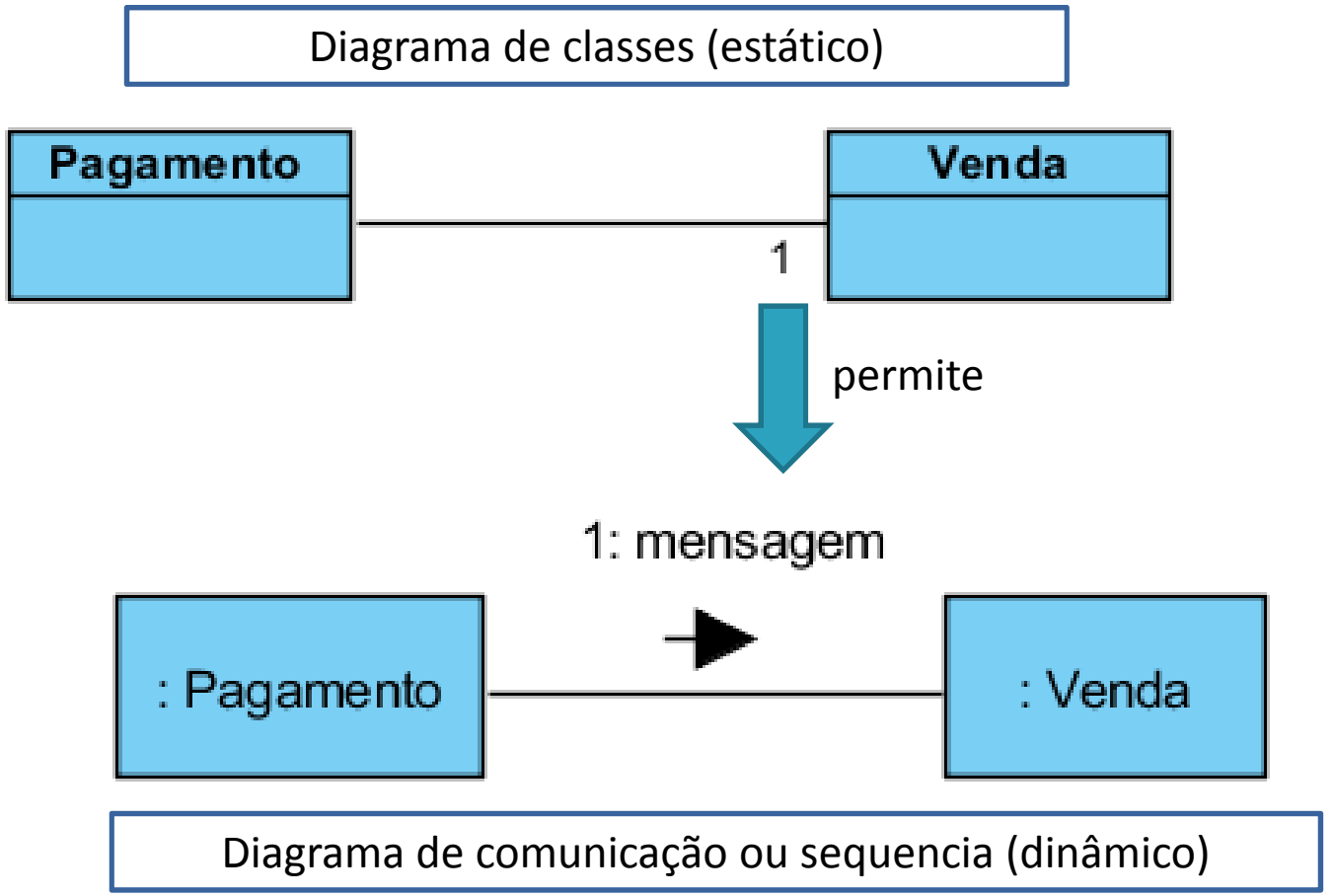
2.2.1 VISIBILIDADE

- Para que dois objetos possam trocar mensagens para realizar responsabilidades derivadas e coordenadas, é necessário que exista *visibilidade* entre eles.
- Existem quatro formas básicas de visibilidade:
  - *Por associação*.
    - Quando existe uma associação entre os dois objetos de acordo com as definições de suas classes no modelo conceitual.
  - *Por parâmetro*.
    - Quando um objeto, ao executar um método, recebe outro como parâmetro.
  - *Localmente declarada*.
    - Quando um objeto, ao executar um método, recebe o outro como retorno de uma consulta.
  - *Global*.
    - Quando um objeto é declarado globalmente.

Visibilidade por Associação

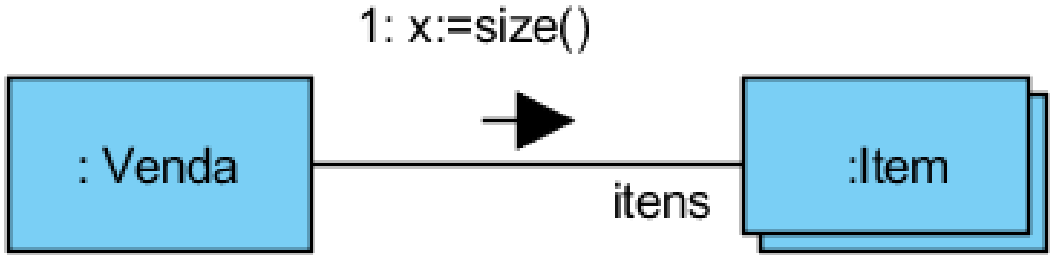
- Somente pode existir *visibilidade por associação* entre dois objetos, quando existir uma associação entre as classes correspondentes no modelo conceitual ou DCP.
- O tipo de visibilidade que se tem varia conforme a multiplicidade de papel e de outras características, como o fato de a associação ser qualificada ou possuir classe de associação.
  - Se a multiplicidade de papel for para 1, então se tem visibilidade diretamente para uma instância.
  - Qualquer outra multiplicidade de papel dá visibilidade a um conjunto de instâncias.

VISIBILIDADE POR ASSOCIAÇÃO PARA 1

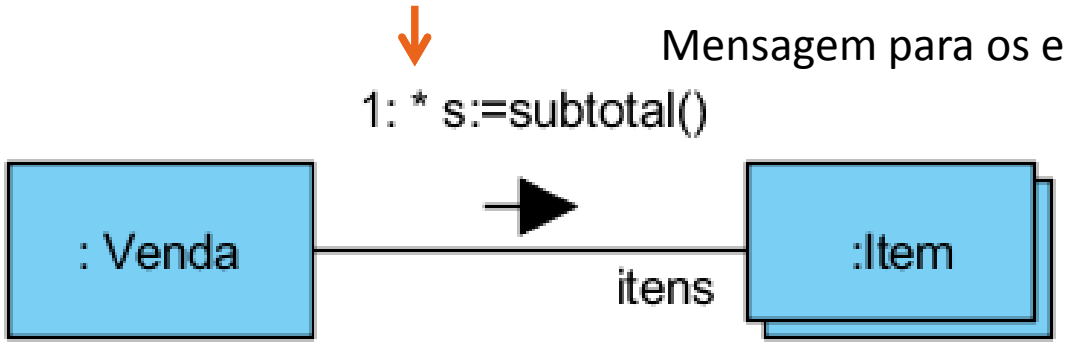


VISIBILIDADE POR ASSOCIAÇÃO PARA MUITOS

Mensagem para a estrutura do conjunto

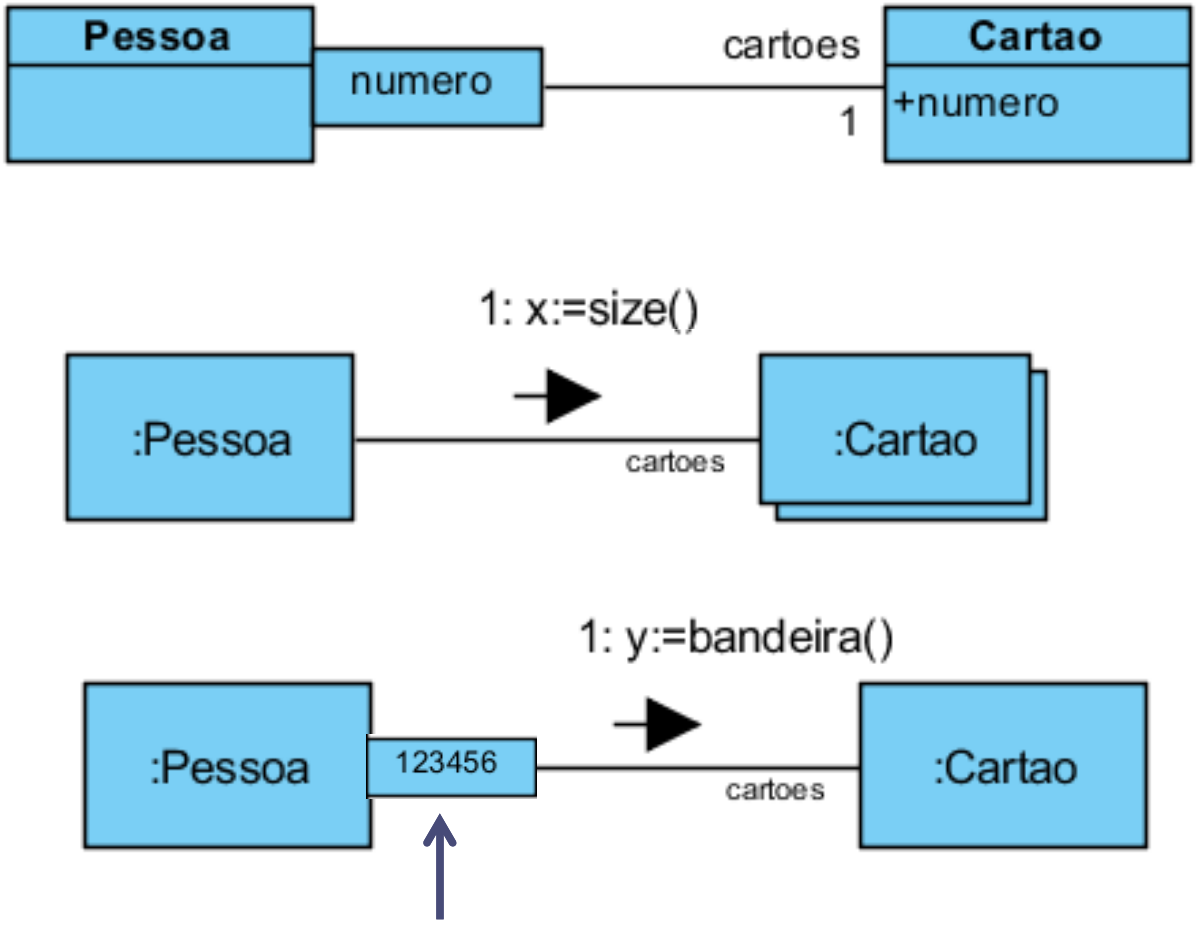


Mensagem para os elementos do conjunto



VISIBILIDADE POR ASSOCIAÇÃO ORDENADA

VISIBILIDADE POR ASSOCIAÇÃO QUALIFICADA



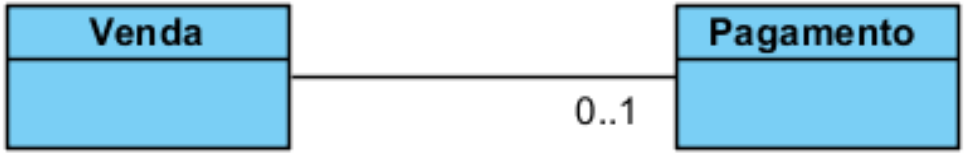
Qualificador conhecido por :Pessoa

Influência das Pré-Condições na Visibilidade por Associação

- Quando uma pré-condição de operação de sistema restringe ainda mais uma multiplicidade de papel, então, no contexto daquela operação é a pré-condição que vale como determinante da visibilidade.

EXEMPLO

Modelo conceitual:



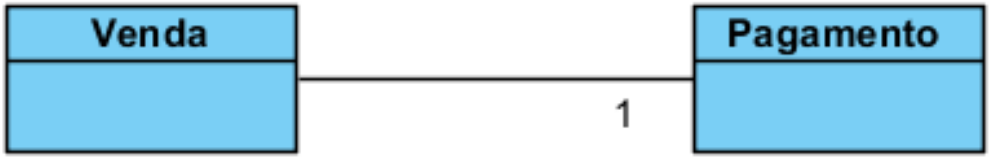
+

Pré-condição:

```
pre: umaVenda.pagamento->size() = 1
```

=

Condição efetiva:

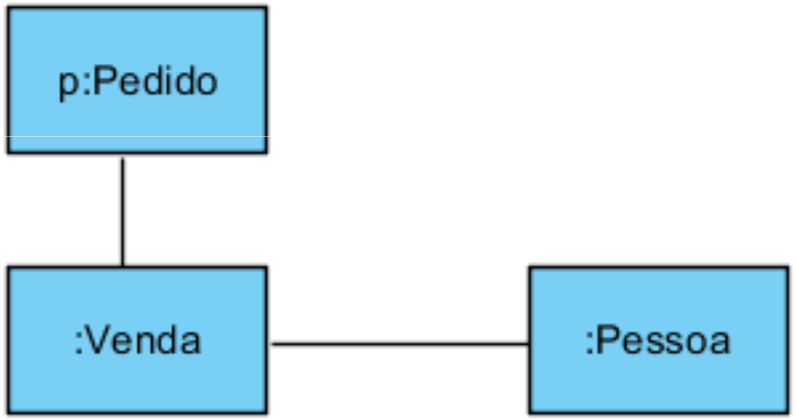


Visibilidade por parâmetro

- A visibilidade por parâmetro é obtida quando um objeto, ao executar um método, recebe outro objeto como parâmetro.

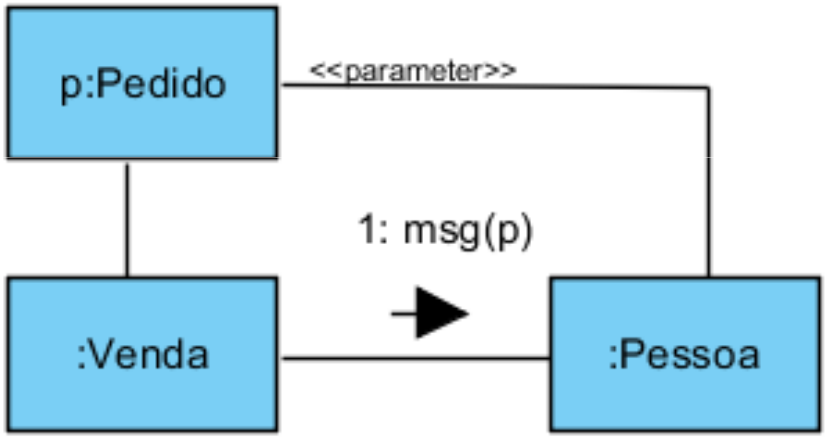
EXEMPLO

- Estado inicial



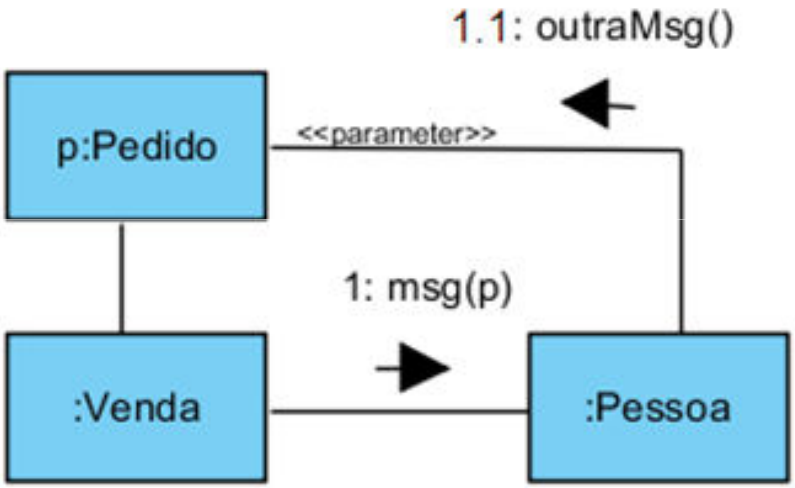
EXEMPLO

- Envio da mensagem parametrizada



EXEMPLO

- A nova visibilidade pode ser usada



Visibilidade Localmente Declarada

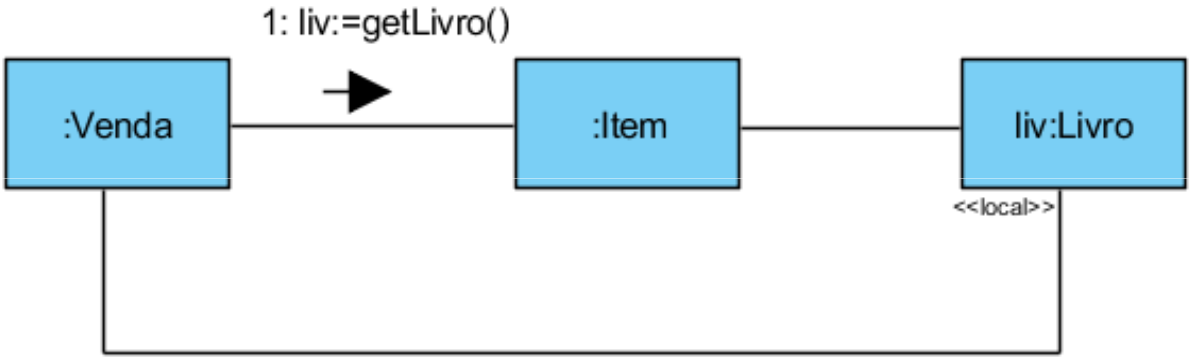
- Ocorre quando um objeto, ao enviar uma consulta a outro recebe *como retorno* um terceiro objeto.

EXEMPLO

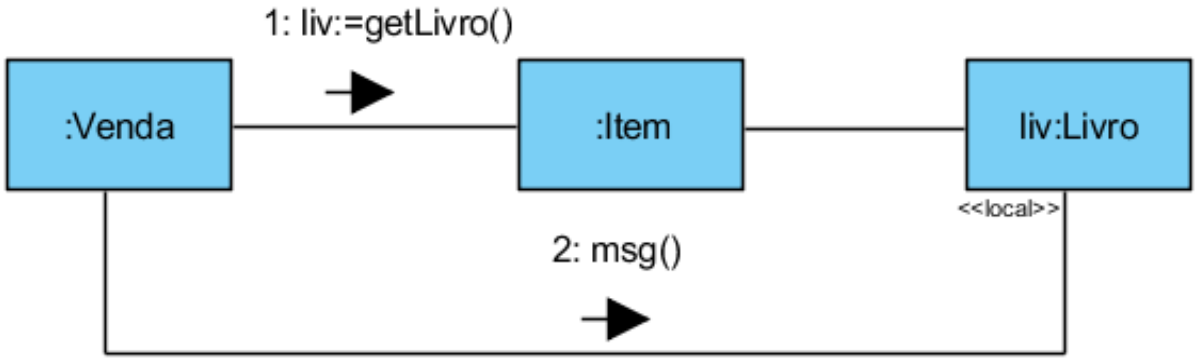
- 1:


:Venda
:Item
:Livro

- 2:

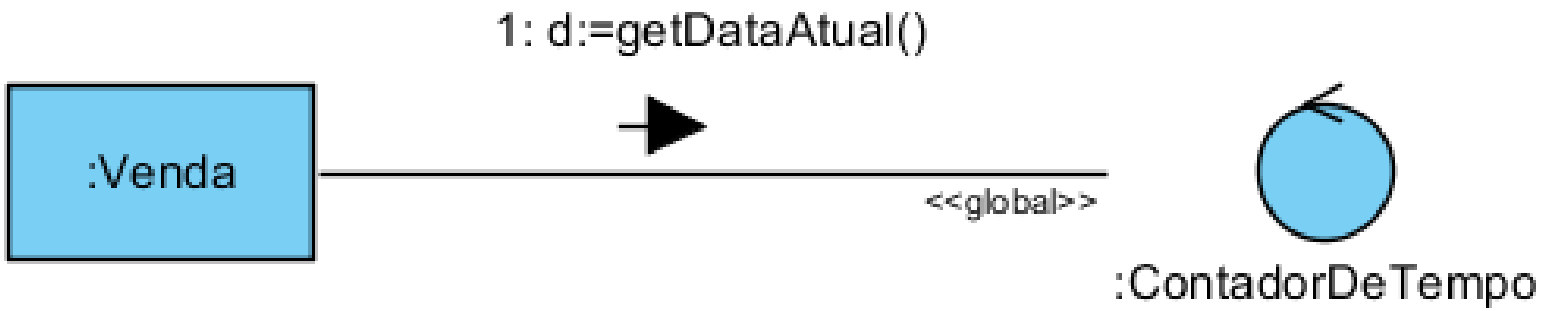

1: liv:=getLivro()
:Venda
:Item
liv:Livro
<<local>>

- 3:


1: liv:=getLivro()
:Venda
:Item
liv:Livro
<<local>>
2: msg()

VISIBILIDADE GLOBAL

- Existe visibilidade global para um objeto quando ele é declarado globalmente.
- O padrão de projeto *Singleton* admite uma instância globalmente visível apenas quando ela é a *única instância possível da sua classe*.



2.2.2 Realização dinâmica das pós-condições

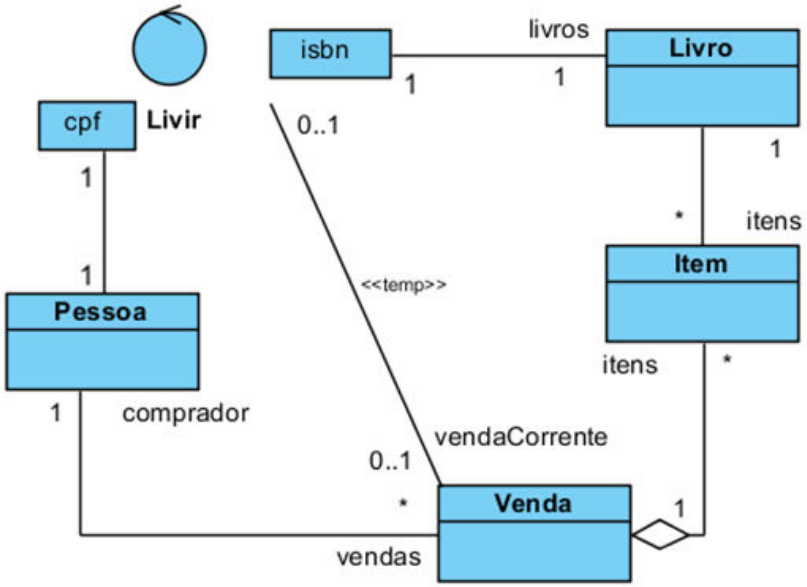
- Os contratos OCL apenas indicam *o que* deve acontecer, sem mostrar como mensagens reais são trocadas entre objetos para realizar tais ações.
- Os diagramas de comunicação ou seqüência podem ser usados exatamente para mostrar como essas trocas são feitas.

PRINCÍPIOS

- A **visibilidade** entre instâncias de objetos é regida pela multiplicidade de papel estabelecida no modelo conceitual e eventuais pré-condições que restringem tal multiplicidade.
- Cada uma das **operações básicas** estabelecidas em pós-condição deve ser realizada no diagrama de comunicação através do envio de uma mensagem básica ao objeto que detém a ***responsabilidade*** de forma mais imediata.
- O fluxo de execução em um diagrama de comunicação ou seqüência que representa uma operação de sistema ***sempre*** **inicia na instância da *controladora de sistema*** recebendo uma mensagem da *interface*.
- Quando o objeto que detém o controle de execução não tiver visibilidade para o objeto que deve executar a operação básica, ele deve **delegar** a responsabilidade (e o controle) a outro objeto que possa fazê-lo ou que esteja mais próximo deste último.

EXEMPLO

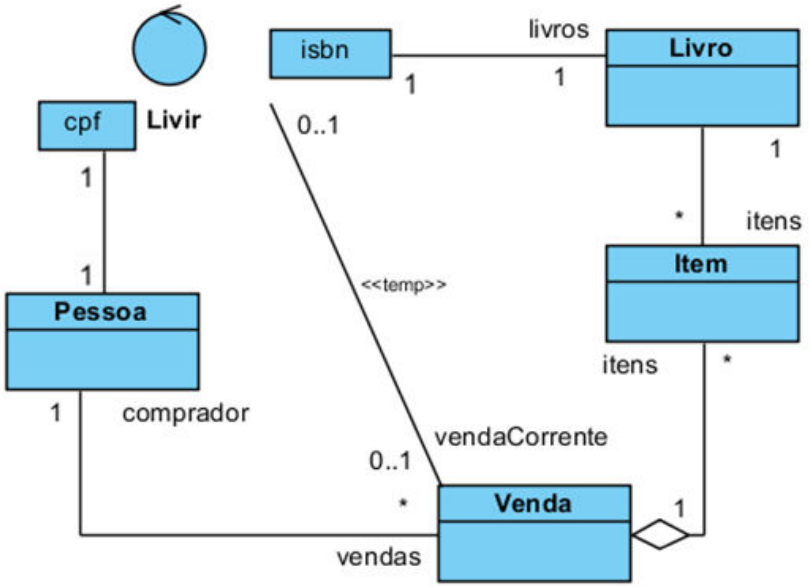
- Um **contrato OCL** completo sendo transformado em **modelo dinâmico** passo a passo




```
Context Livir::adicionaItem(idLivro,quantidade)
  def: item =
    Item::newInstance()
  def: livro =
    livros[idLivro]
  pre:
    vendaCorrente→size()=1
  post:
    vendaCorrente^addItens(item) AND
    item^addLivro(livro) AND
    item^setQuantidade(quantidade)
```

CRIAÇÃO DA INSTÂNCIA DE ITEM

```
Context Livir::adicionaItem(...)
  def: item =
    Item::newInstance()
  pre:
    vendaCorrente→size()=1
  post: ...
```



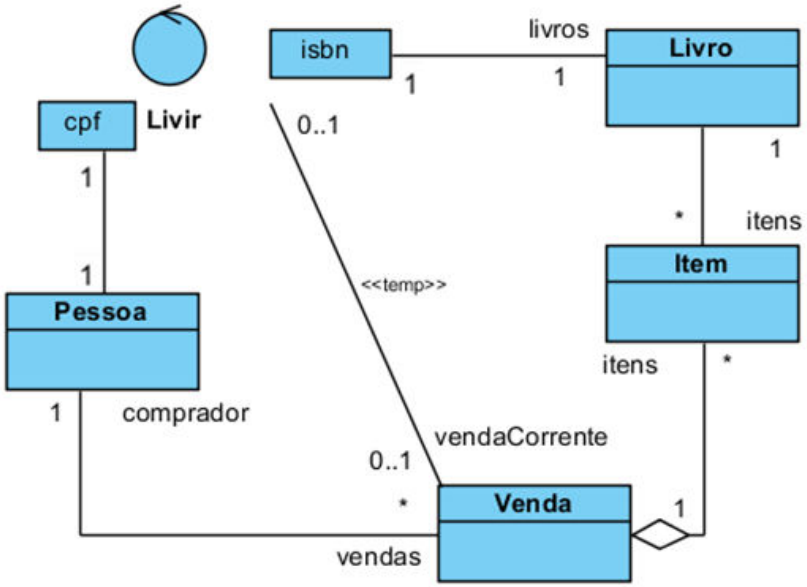
op. de sistema

delegada

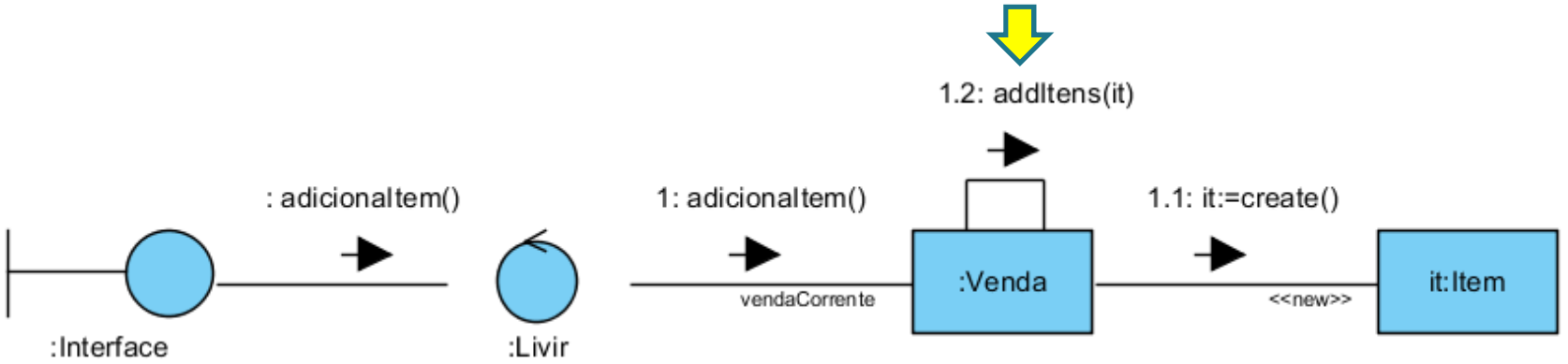
básica

CRIAÇÃO DA ASSOCIAÇÃO ENTRE VENDA E ITEM

```
Context Livir::adicionaItem(...)
  def: item =
    Item::newInstance()
  pre:
    vendaCorrente→size() = 1
  post:
    vendaCorrente^addItens(item)
```

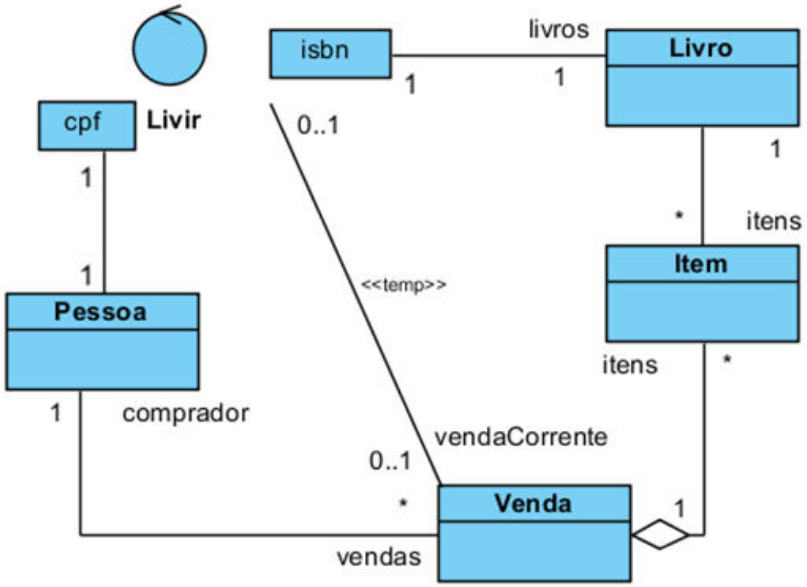


básica

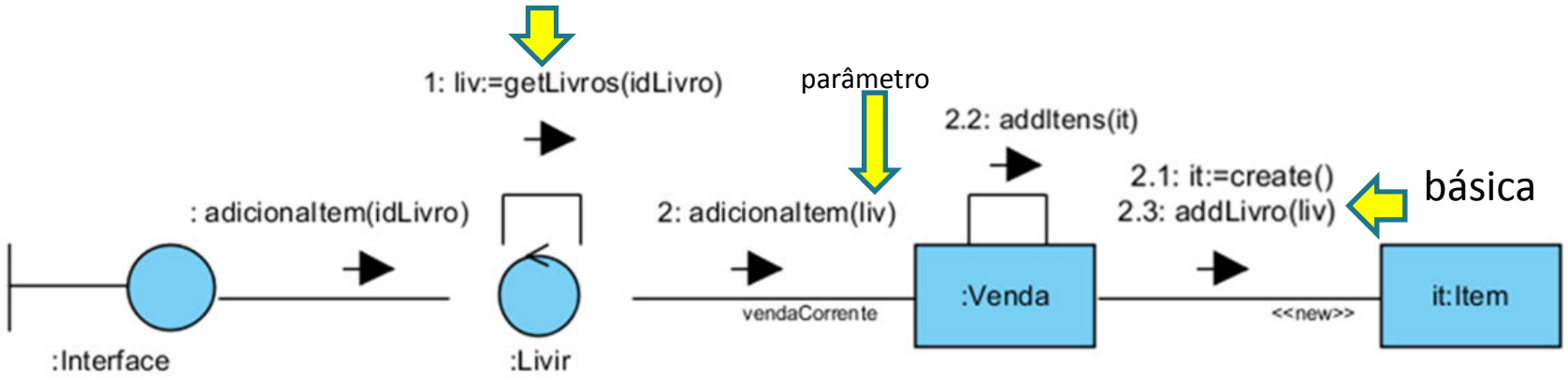


CRIAÇÃO DA ASSOCIAÇÃO ENTRE ITEM E LIVRO

```
Context Livir::adicionaItem(idLivro)
  def: item =
    Item::newInstance()
  def: livro =
    livros[idLivro]
  pre:
    vendaCorrente→size()=1
  post:
    vendaCorrente^addItens(item) AND
    item^addLivro(livro)
```

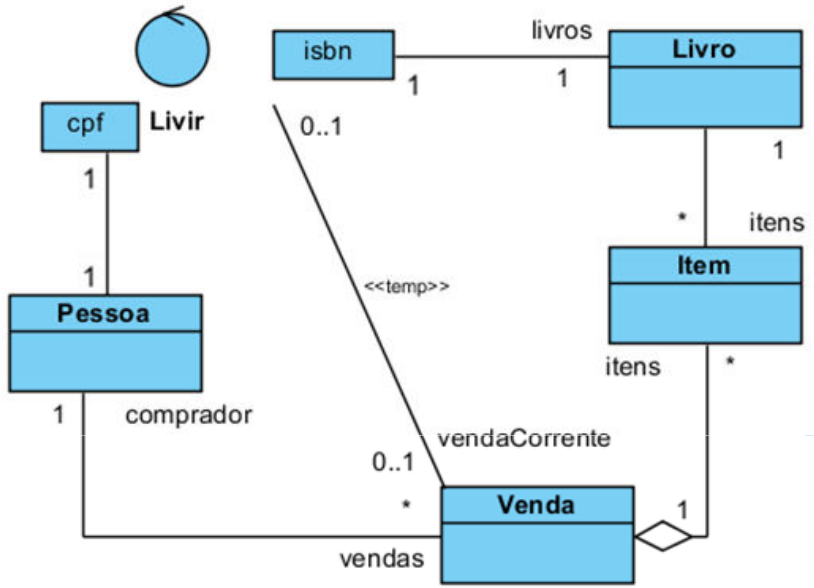


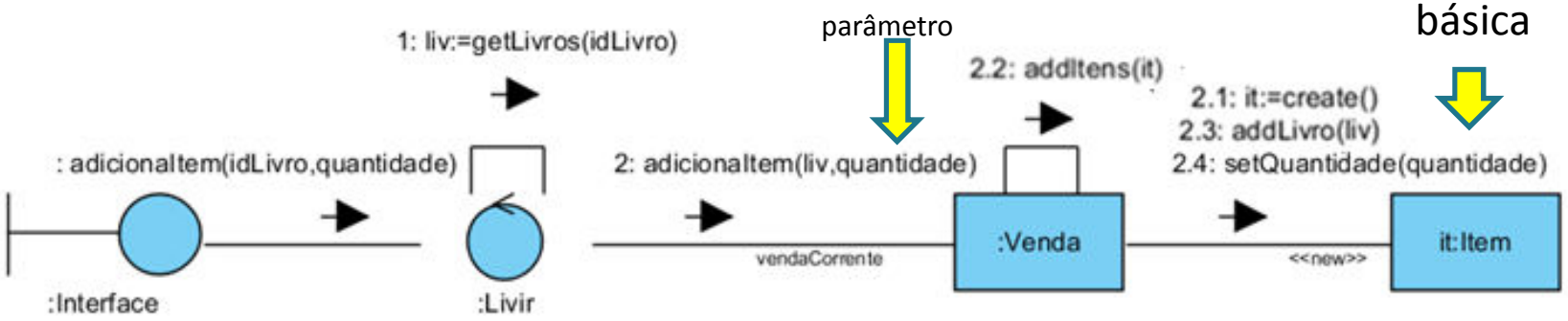
básica



MODIFICAÇÃO DO ATRIBUTO "QUANTIDADE" DO LIVRO

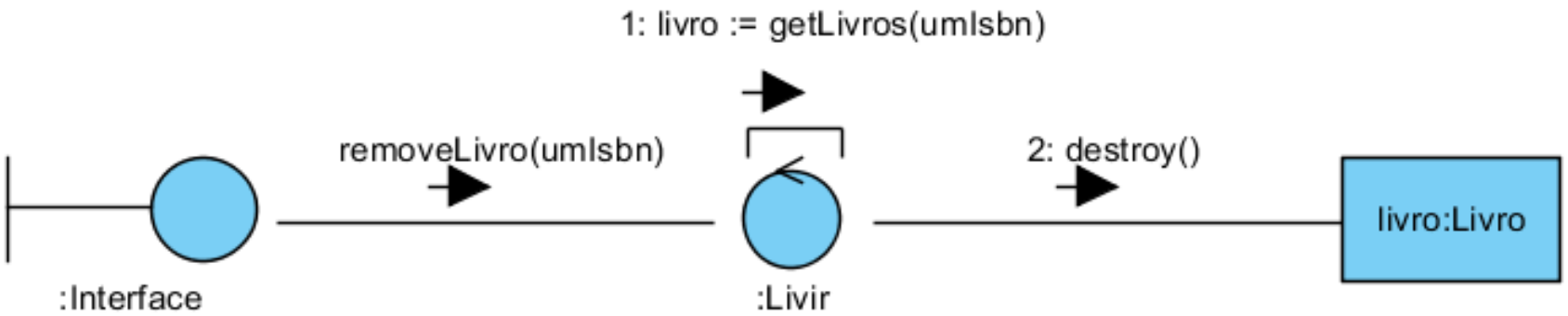
```
Context Livir::adicionaItem(idLivro,quantidade)
  def: item =
    Item::newInstance()
  def: livro =
    livros[idLivro]
  pre:
    vendaCorrente→size()=1
  post:
    vendaCorrente^addItens(item) AND
    item^addLivro(livro) AND
    item^setQuantidade(quantidade)
```





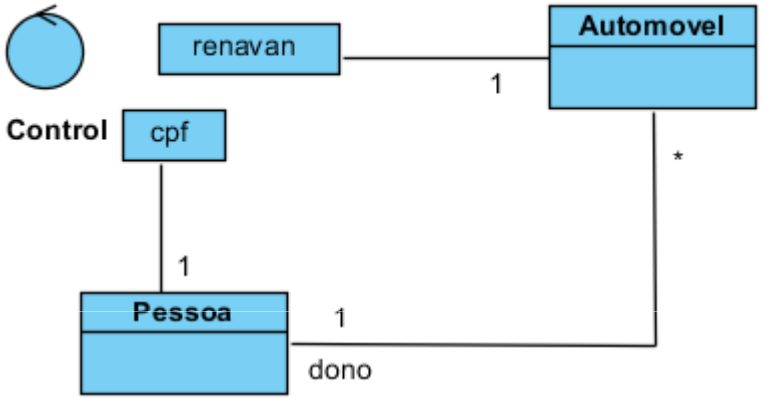
EXEMPLO DE DESTRUIÇÃO DE INSTÂNCIA

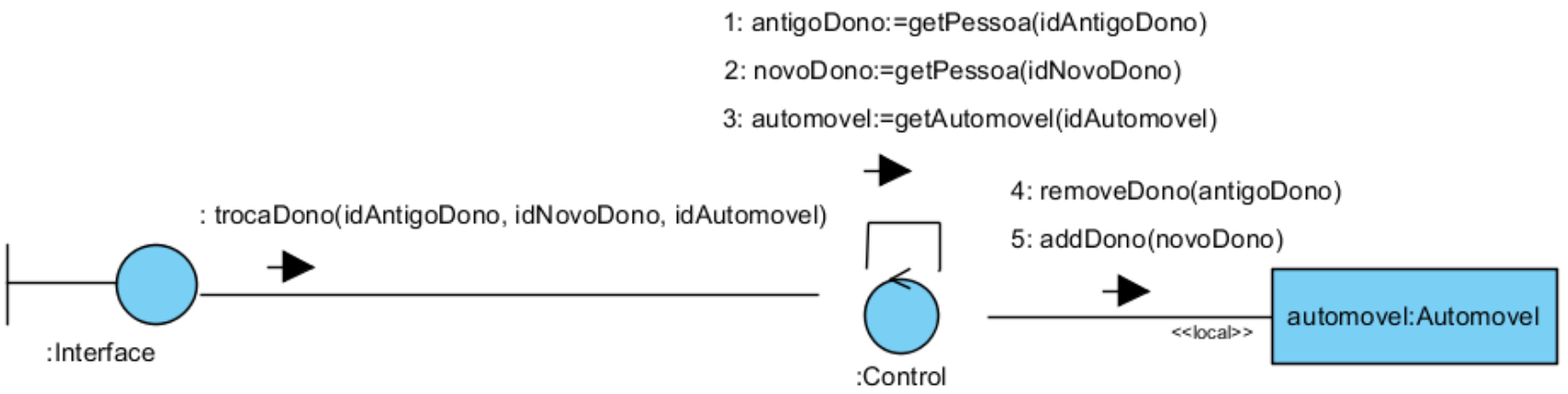
```
Context Livir::removeLivro(umIsbn)
  def:
    livro = livros[umIsbn]
  pre:
    livro.itens→size() = 0
  post:
    livro^destroy()
```



EXEMPLO DE DESTRUIÇÃO DE ASSOCIAÇÃO

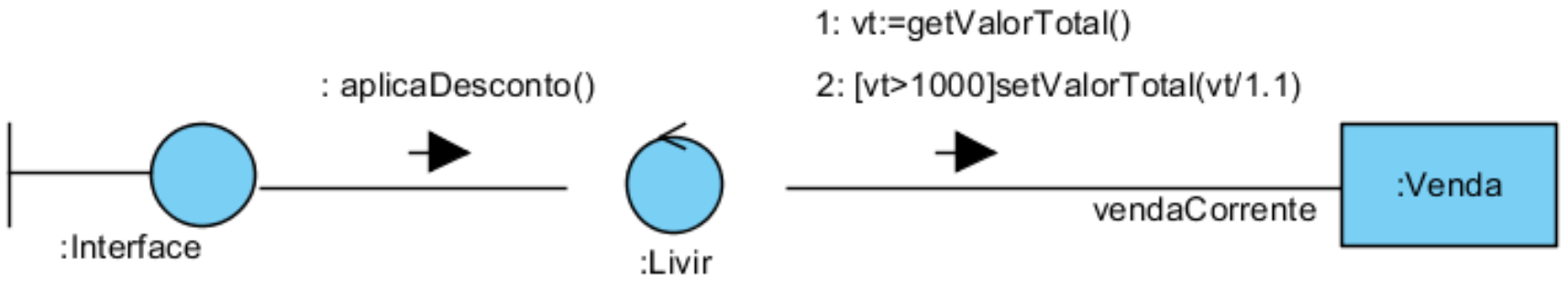
```
Context Control::trocaDono(idAntigoDono, idNovoDono, idAutomovel)
  def: antigoDono =
    pessoas[idAntigoDono]
  def: novoDono =
    pessoas[idNovoDono]
  def: automovel =
    automoveis[idAutomovel]
  pre:
    antigoDono→size() = 1 AND
    novoDono →size() = 1 AND
    automovel→size() = 1
  post:
    automovel^removeDono(antigoDono) AND
    automovel^addDono(novoDono)
```





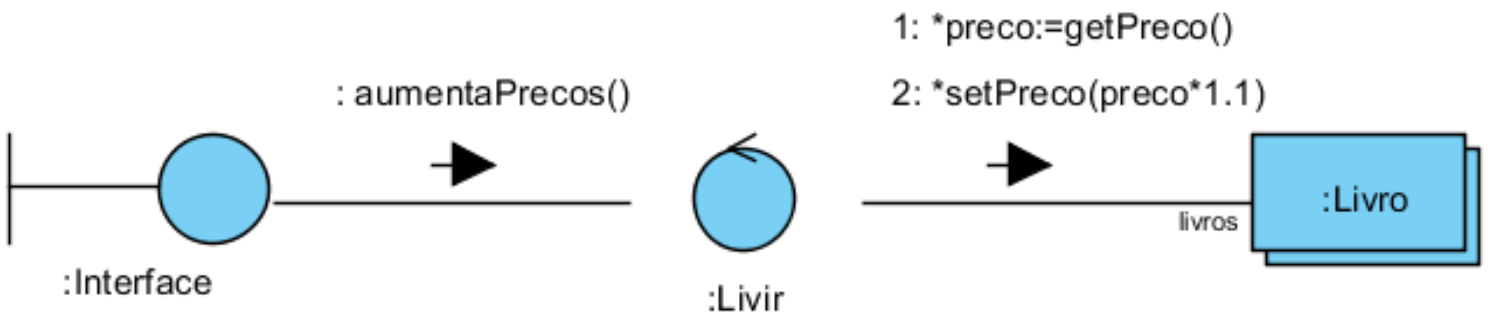
EXEMPLO DE PÓS-CONDIÇÃO CONDICIONAL

```
Context Livir::aplicaDesconto()
  pre:
    vendaCorrente→size() = 1
  post:
    vendaCorrente.valorTotal > 1000 IMPLIES
      vendaCorrente^setValorTotal(vendaCorrente.valorTotal@pre/1.1)
```



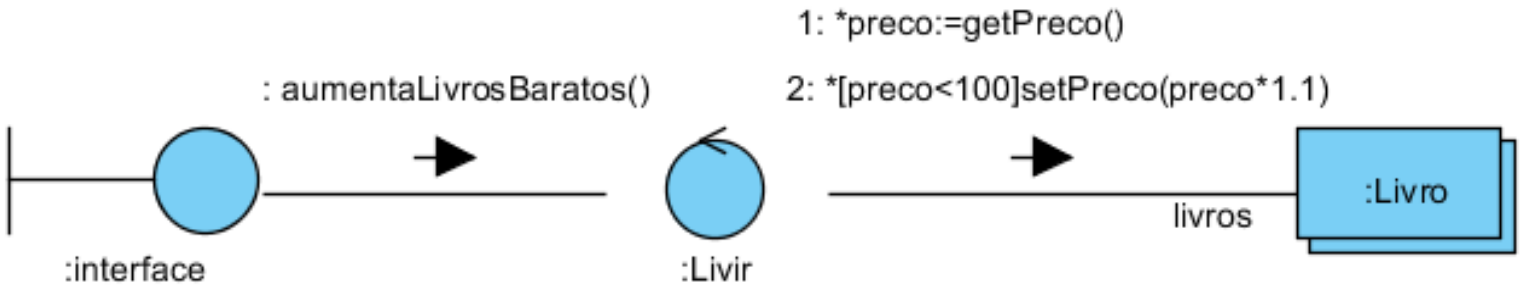
PÓS-CONDIÇÕES SOBRE COLEÇÕES

```
Context Livir::aumentaPrecos()
  post: livros→forAll(livro|
    livro^setPreco(livro.preco@pre * 1.1)
  )
```

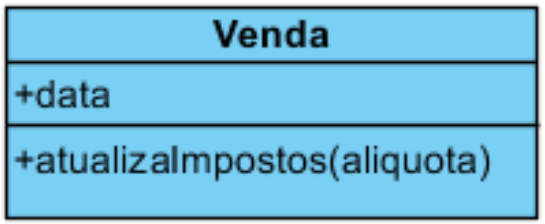


PÓS-CONDIÇÃO SOBRE PARTE DE UMA COLEÇÃO

```
Context Livir::aumentaLivrosBaratos()
  post:
    livros→select(preco@pre<100)→forAll(livro|
      livro^setPreco(livro.preco@pre *1.1)
    )
```

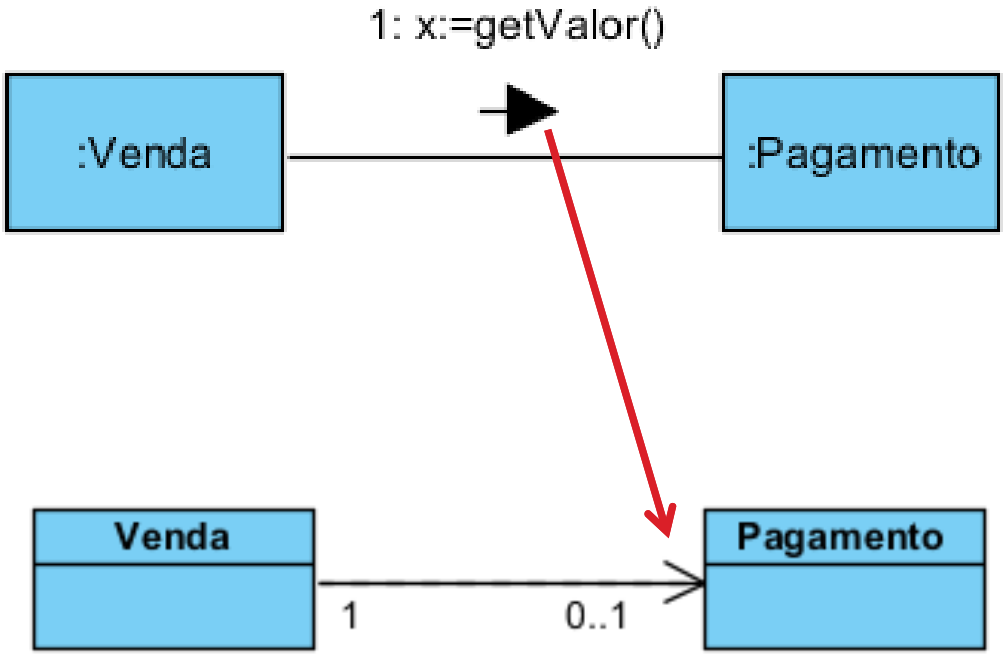


2.2.3 MÉTODOS DELEGADOS

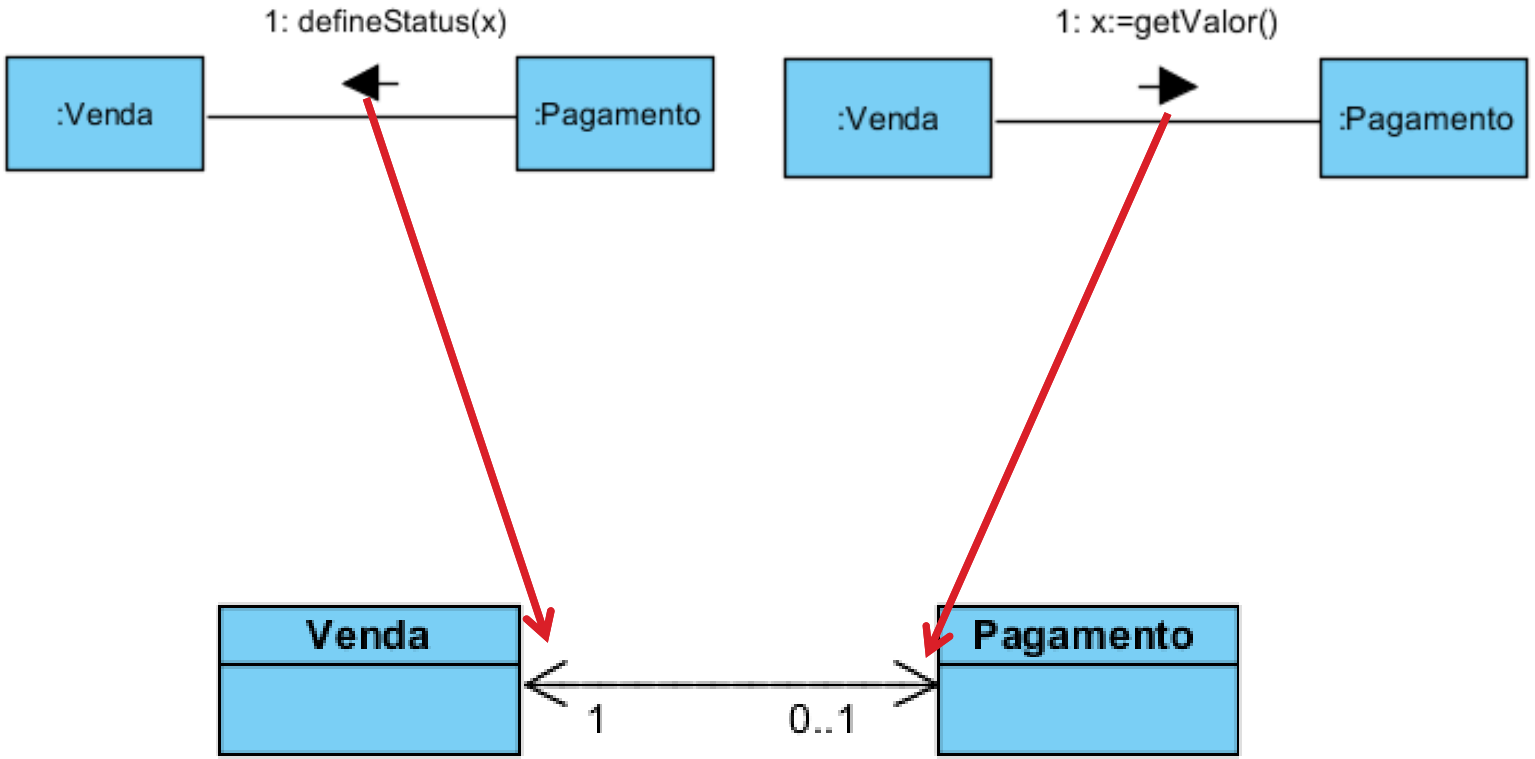


Não é necessário colocar no diagrama de classes as operações básicas e consultas a atributos ou associações, visto que elas podem ser deduzidas pela própria existência das classes, associações e atributos

DETERMINAÇÃO DA DIREÇÃO DAS ASSOCIAÇÕES



DETERMINAÇÃO DA DIREÇÃO DAS ASSOCIAÇÕES



Implementação de um método delegado

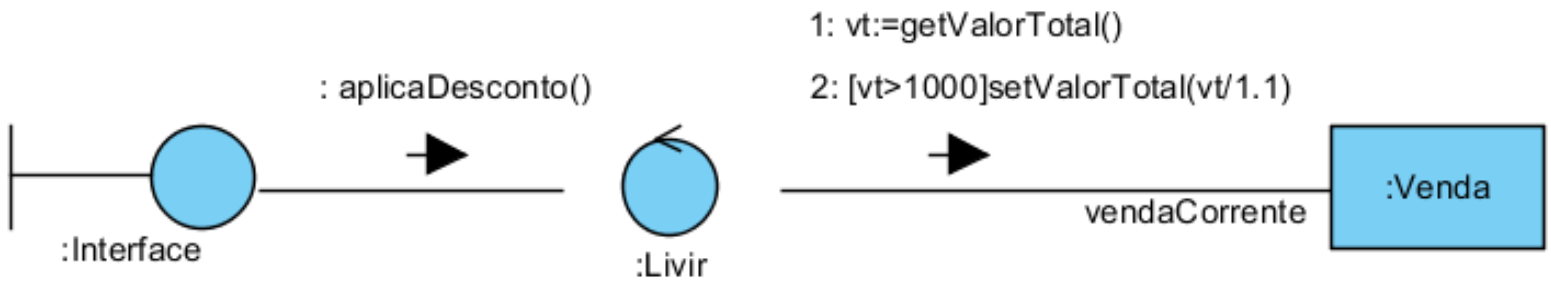
- Toda mensagem delegada com número *x*, que chega a uma instância da classe A deve ser implementada como a seqüência das mensagens *x*.1, *x*.2, ..., *x*.*n*, que saem da instância de A e são enviadas a outros objetos.

Exemplo


```
CLASSE Livir
...
MÉTODO adicionaItem(idLivro:String, quantidade:Inteiro)
        VAR liv:Livro
        liv := self.getLivros(idLivro) --1
        self.getVendaCorrente().adicionaItem(liv,quantidade) --2
FIM MÉTODO
FIM CLASSE
```

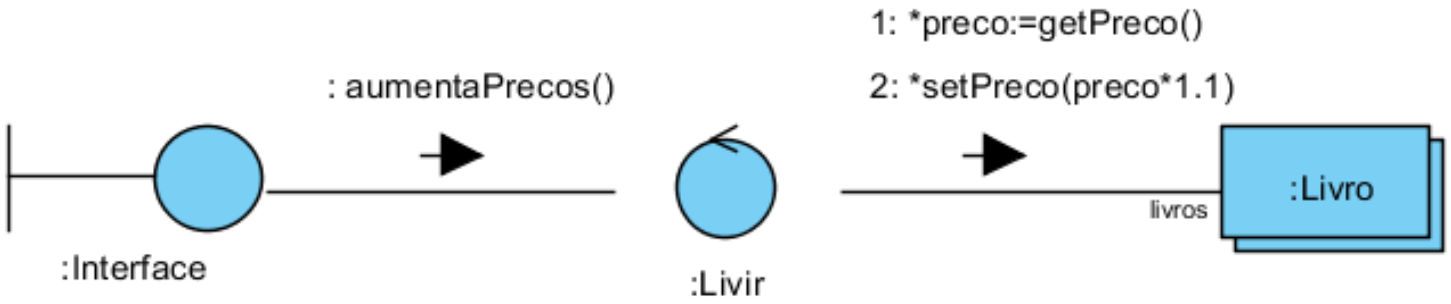
```
CLASSE Venda
...
MÉTODO adicionaItem(liv:Livro, quantidade:Inteiro) --2
        VAR it:Item
        it:=Item.newInstance() --2.1
        self.addItens(it) --2.2
        it.addLivro(liv) --2.3
        it.setQuantidade(quantidade) -2.4
FIM MÉTODO
FIM CLASSE
```

IMPLEMENTAÇÃO DE CONDICIONAL




```
CLASSE Livir
...
MÉTODO aplicaDesconto()
        VAR vt:MOEDA
        vt := self.getValorTotal()--1
        SE vt>1000 ENTÃO
                self.getVendaCorrente().setValorTotal(vt/1.1) --2
        FIM SE
FIM MÉTODO
FIM CLASSE
```

IMPLEMENTAÇÃO DE ITERAÇÃO



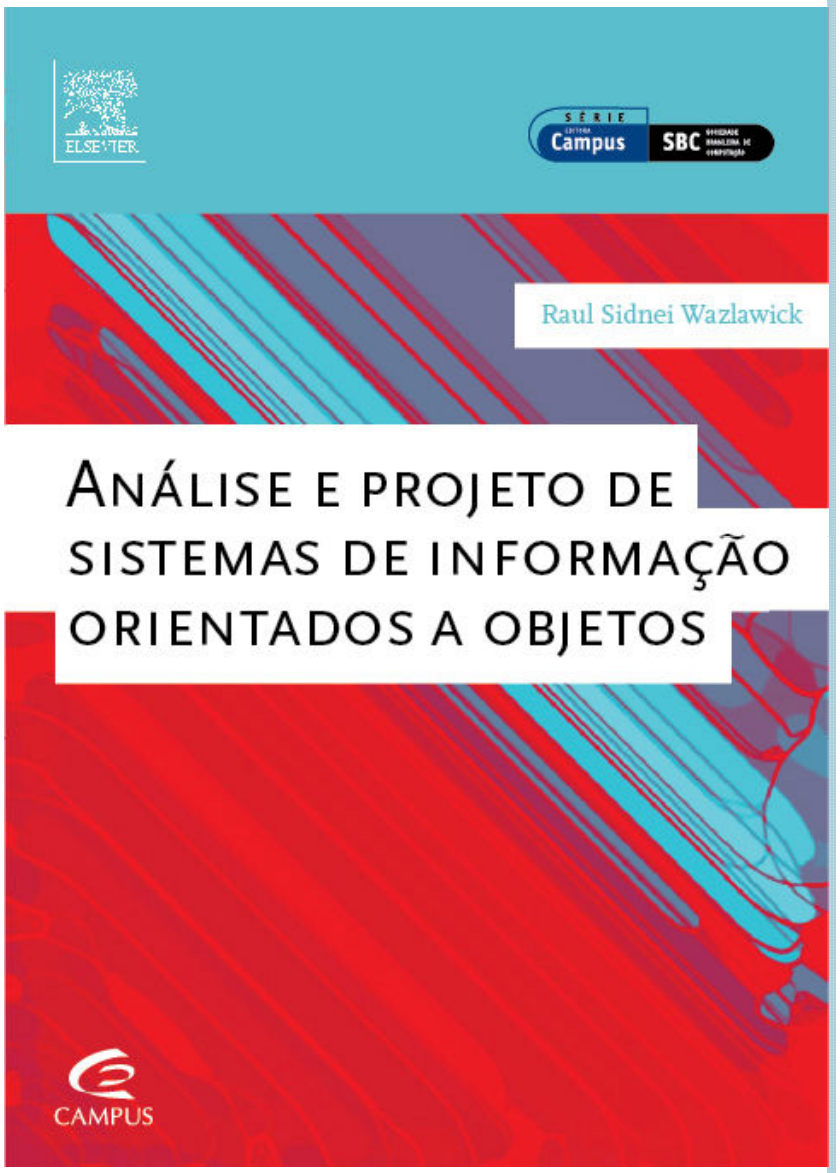

```
CLASSE Livir
...
MÉTODO aumentaPrecos()
        VAR preco:MOEDA
        PARA TODO livro EM self.getLivros() FAÇA
                preco := livro.getPreco() --1
                livro.setPreco(preco*1.1) --2
        FIM PARA
FIM MÉTODO
FIM CLASSE
```

Procedimento geral para geração de código

1. Geração de código padrão para a **estrutura** das classes, atributos e associações.
2. Geração de código padrão para as operações básicas (get, set, add, remove, create e destroy).
3. Geração de código para as operações e consultas de sistema e métodos delegados de acordo com os diagramas de comunicação derivados dos contratos.

OBRIGADO!

- Contato:
  - www.inf.ufsc.br/~raul/



(BIS) 2.3 SALVAMENTO E CARREGAMENTO

- As tarefas de salvamento e carregamento de objetos pode ser totalmente controladas pela própria arquitetura do sistema.
- Para isso é necessário um sistema de carregamento preguiçoso baseado em proxy virtual.

PROXY VIRTUAL

- Um *proxy* virtual é um objeto muito simples que implementa apenas duas responsabilidades:
  - Conhecer o valor da chave primária do objeto real.
  - Repassar ao objeto real todas as mensagens que receber em nome dele.

IMPLEMENTAÇÃO DE UM PROXY VIRTUAL

Classe *Proxy* Virtual

Para qualquer mensagem recebida faça:

Solicite ao BrokerManager o objeto real a partir de sua *pk*.

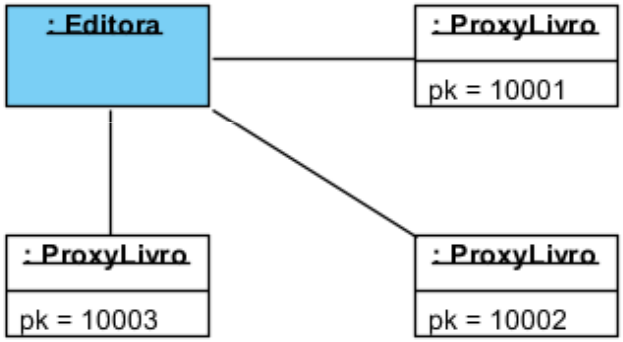
Repasse a mensagem recebida ao objeto real.

Fim

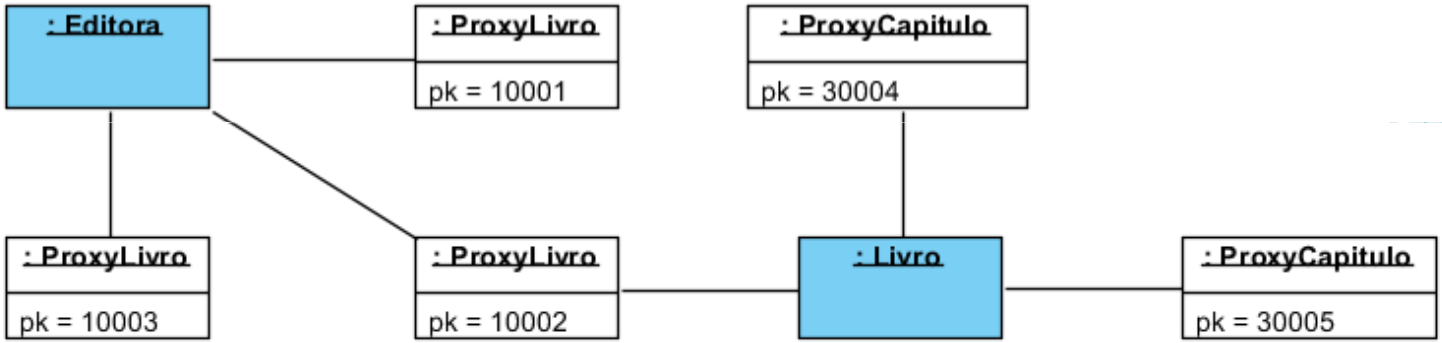
Assim, o projeto poderá determinar que ao invés dos objetos se associarem uns aos outros, eles se associam com seus *proxies*.

EXEMPLO DE LAZZY LOAD

- Estado inicial



EXEMPLO DE LAZZY LOAD



Estruturas de dados virtuais

- A implementação de *proxies* virtuais para cada objeto pode ser bastante ineficiente quando se trata de mapear coleções de objetos.
  - Por exemplo, uma editora associada a 1000 títulos de livro teria que ter 1000 *proxies* instanciados associados a ela quando fosse trazida à memória?
  - A resposta, felizmente é *não*.
- A solução para evitar a instanciação indiscriminada de *proxies* em memória é a implementação de estruturas de dados virtuais pra substituir a implementação das associações.

FUNCIONAMENTO DE UMA EDV

- Ao invés de uma representação física de um conjunto de objetos, terá uma representação física de um conjunto de números inteiros: as *pk* dos objetos.
- O método que adiciona um elemento na estrutura deve adicionar apenas a *pk* do elemento na representação física.
- O método que remove um elemento da estrutura deve apenas remover a *pk* do elemento na representação física.
- Qualquer método que consulte a estrutura para obter um objeto deve tomar a *pk* do objeto da representação física e solicitar ao *BrokerManager* que retorne o objeto real.

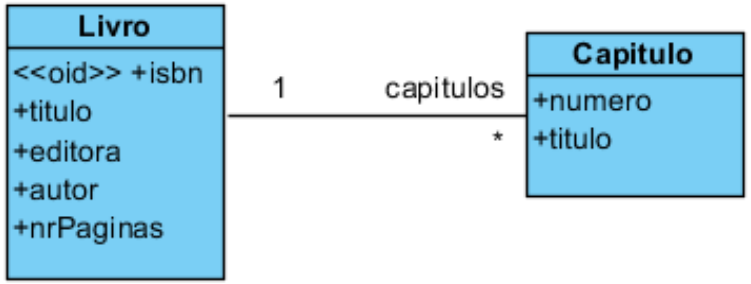
MATERIALIZAÇÃO

- O processo de carregamento de um objeto do banco de dados para a memória principal é denominado de *materialização.*
- A materialização é solicitada pelo BrokerManager a um *broker* especializado sempre que necessário, ou seja, sempre que um *proxy* solicitar acesso a um objeto que ainda não esteja materializado.
- Poderá existir um *broker* especializado para cada classe persistente ou um único *broker* genérico.

FUNCIONAMENTO DO MÉTODO *MATERIALIZA* DE UM BROKER

1. Cria uma instância da classe persistente.
2. Inicializa os valores dos atributos da nova instância com valores da respectiva linha e coluna do banco de dados.
3. Inicializa as estruturas de dados virtuais que implementam as associações do objeto com as chaves primárias dos respectivos objetos associados.

EXEMPLO



- Um broker de Livro:
  - Cria uma instância de Livro.
  - Preenche os atributos isbn, titulo, editora, autor e nrPaginas da nova instância com os valores armazenados nas respectivas colunas da tabela Livro no banco de dados.
  - Buscar na tabela livro_capitulos ocorrências da *pk* do livro na coluna pkLivros.
    - Para todas as ocorrências, adicionar no VirtualSet capitulos da nova instância de Livro os valores da coluna correspondente pkCapitulo.

Caches

- Uma *cache* é uma estrutura de dados na forma de um dicionário (ou Map), que associa valores de pk com objetos reais.

Classificação de objetos em memória

- *Limpos* ou *sujos*
  - dependendo se estão ou não consistentes com o banco de dados.
- *Novos* ou *velhos*
  - dependendo se já existem ou não no banco de dados.
- *Excluídos*
  - dependendo se foram excluídos em memória, mas ainda não do banco de dados.

Quatro combinações são suficientes

- *Old clean cache*
  - onde ficam os objetos que estão consistentes com o banco de dados, ou seja, velhos e limpos.
- *Old dirty cache*
  - onde ficam os objetos que existem no banco de dados, mas foram modificados em memória, isto é, velhos e sujos.
- *New cache*
  - onde ficam os objetos que foram criados em memória, mas ainda não existem no banco de dados.
- *Delete cache*
  - onde ficam os objetos deletados em memória, mas que ainda existem no banco de dados.

COMMIT

- Efetuar um update no banco de dados para os objetos da OldDirtyCache e mover estes objetos para a OldCleanCache.
- Efetuar um insert no banco de dados para os objetos da NewCache e mover esses objetos para a OldCleanCache.
- Efetuar um remove no banco de dados para os objetos da DeleteCache e remover estes objetos da *cache*.

ROLLBACK

- Remover todos os objetos de todas as *caches*, exceto os da OldCleanCache.

ABORDAGENS MULTIUSUÁRIO

1. Não existe compartilhamento de memória no servidor, o qual serve apenas para armazenar os dados no momento em que a transação efetuar um commit.
   - Neste caso, o que trafega pela rede são registros do banco de dados e instruções SQL apenas nos momentos da materialização de objetos ou commit.
   - A desvantagem desta forma de definir a arquitetura é que o nodo cliente fica sobrecarregado, e mecanismos de controle de segurança adicionais devem ser implementados no próprio banco de dados para impedir acessos não autorizados.
2. Implementar no nodo cliente apenas a camada de interface, e deixar no servidor as camadas de domínio e persistência.
   - Neste caso, os objetos existirão em memória apenas no servidor e a comunicação na rede consistirá no envio de mensagens e recebimento de dados.

AGORA SIM!

- Contato:
  - www.inf.ufsc.br/~raul/