```
Context Livir::consultaPrazoEntrega():Date
  pre:
    compraCorrente→size() = 1
  body:
    compraCorrente.enderecoEntrega.cidade.tempoEntrega
```

Observa-se que, na maioria das operações e consultas, a principal alteração entre as estratégias *stateless* e *statefull* foi a troca da expressão, que era definida como compras[idCompra] na estratégia *stateless* e por self.compraCorrente na estratégia *statefull*.

Capítulo

9

Projeto da Camada de Domínio

Ainda durante a fase de *elaboração*, após as atividades de análise, vêm as atividades de projeto. Mas o que é *projeto de software orientado a objetos*? Pode-se dividir as atividades de projeto em dois grandes grupos:

a) o *projeto lógico*, que inclui os diagramas de classe que evoluem a partir do modelo conceitual e os diagramas de modelagem dinâmica que representam a maneira como os objetos interagem para executar as operações e consultas de sistema;

b) o *projeto tecnológico*, que inclui todos os aspectos do problema que são inerentes à tecnologia empregada: interface, armazenamento de dados, segurança, comunicação, tolerância a falhas etc.

O *projeto* do software visa produzir uma *solução* para o problema que, nesse ponto, já deve estar suficientemente esclarecido pela análise. O *problema* está especificado no modelo conceitual, nos contratos e nos diagramas de sequência de sistema ou casos de uso expandidos. Resta agora projetar uma arquitetura de software (e possivelmente hardware) para realizar lógica e tecnologicamente o sistema, ou seja, para apresentar uma solução ao problema enunciado.

O projeto lógico também é conhecido como projeto da *camada de domínio*. Essa camada da arquitetura do software corresponde ao conjunto de

classes que vai realizar toda a lógica de acesso e transformação dos dados do sistema de informação. As demais camadas (persistência, interface, segurança etc.) são derivadas ou dependentes da camada de domínio, servindo para conectar essa lógica pura com os aspectos físicos da computação (redes, interfaces, dispositivos de armazenamento etc.).

O projeto da camada de domínio consiste basicamente em duas atividades que podem ser executadas concomitantemente:

a) *modelagem dinâmica*, que consiste em construir modelos de execução para os contratos de operação e consulta de sistema. Em orientação a objetos, tais modelos devem ser modelos de interação entre objetos, que usualmente são representados através de diagramas de comunicação ou diagramas de sequência da UML ou, ainda, diretamente na forma algorítmica;
b) *elaboração do diagrama de classes de projeto* (*DCP*), que consiste basicamente em adicionar ao modelo conceitual algumas informações que não era possível ou desejável obter na atividade de análise, como, por exemplo, a direção das associações e os métodos a serem implementados nas classes. Esses aspectos só podem ser efetivamente inseridos no diagrama durante a modelagem dinâmica.

As demais fases do projeto que são abordadas neste livro em capítulos subsequentes são:

a) *projeto da camada de interface* (Capítulo 10), em que são vistas técnicas para manter a independência entre a camada de domínio e a interface do software;
b) *projeto da camada de persistência* (Capítulo 11), em que é visto como implementar um sistema de persistência que automatiza o salvamento e a recuperação de dados em memória secundária, retirando do projetista a necessidade de se preocupar com esses aspectos.

O projeto lógico do software pode ser realizado de forma bastante sistemática, desde que o projetista possua dois artefatos da atividade de análise corretamente construídos:

a) o modelo conceitual;
b) os contratos de operações e consultas de sistema (modelo funcional).

O trabalho do projetista consistirá em:

a) construir um diagrama de comunicação (ou de sequência) para cada operação e consulta de sistema, levando em conta o modelo conceitual e o respectivo contrato;
b) construir e aprimorar o DCP a partir do modelo conceitual e dos diagramas de comunicação desenvolvidos.

A modelagem dinâmica, como foi explicado, pode se valer de diagramas de comunicação, diagramas de sequência ou ainda de algoritmos. Cada uma das formas tem vantagens e desvantagens:

a) *algoritmos* são os mais fáceis de fazer, mas é difícil perceber claramente as conexões entre os objetos em simples textos. Assim, algoritmos podem fazer com que o acoplamento entre as classes seja aumentado, prejudicando a qualidade do projeto;
b) *diagramas de comunicação* são melhores do que os algoritmos para visualizar e distribuir responsabilidades e melhores do que os diagramas de sequência para visualizar as dependências de visibilidade entre os objetos. Mas pode ser difícil organizá-los graficamente, no caso de colaborações mais complexas, e algumas vezes são mais difíceis de ler do que os diagramas de sequência;
c) *diagramas de sequência* são mais fáceis de entender, mas não explicitam as ligações de visibilidade entre os objetos, podendo permitir, caso o projetista não esteja atento, comunicações inválidas ou impossíveis.

Inicialmente, este capítulo usará diagramas de comunicação para mostrar a importância de perceber que os objetos se comunicam através de linhas de visibilidade. Posteriormente, serão utilizados diagramas de sequência nos exemplos, por serem aparentemente mais populares (e, talvez por isso, inúmeras vezes mal elaborados).

A distribuição de responsabilidades entre os objetos tem a ver com a questão: *quais métodos devem ficar em cada classe?*

Muitos projetistas têm dificuldade para construir uma solução elegante para esse problema quando tentam simplesmente acrescentar métodos em um diagrama de classes. O uso de diagramas de comunicação e padrões de projeto pode, entretanto, permitir uma forma muito mais eficiente de *descobrir* o local adequado para implementar cada método.

A definição dos métodos é feita na atividade de projeto do sistema, quando se constrói o DCP. Para construir esse diagrama, deve-se observar

inicialmente o modelo conceitual (conceitos e associações que representam a estrutura da informação descoberta na atividade de análise de domínio).

Depois, é necessário observar os diagramas de comunicação ou sequência. Esses diagramas apresentam objetos trocando mensagens para realizar contratos. Existe todo um processo para se definir corretamente esses diagramas. A construção deles para representar os aspectos dinâmicos internos do sistema é, especialmente no caso de projetistas menos experientes, superior ao processo de escrever algoritmos porque, quando um projetista faz um algoritmo, ele pode tender a concentrar todas as responsabilidades em uma única classe, enquanto o uso dos diagramas (especialmente os de comunicação) permite melhor visualização da distribuição espacial dos objetos e suas formas de visibilidade, o que possibilita melhor aplicação dos padrões de projeto para distribuição de responsabilidades.

Quando se trabalha com orientação a objetos sem um método adequado, ou seja, simplesmente fazendo um diagrama de classes e adicionando métodos nas classes, sem uma técnica sistemática e padronizada que dirija essa atividade, as responsabilidades das classes acabam sendo mal distribuídas e o resultado final acaba sendo tão desestruturado quanto os chamados programas *spaghetti*.

Assim, para um sistema ser elegante, as responsabilidades têm de estar bem distribuídas. Se não se usa um método sistemático, pode acontecer que as responsabilidades acabem ficando concentradas na classe que representa o sistema como um todo (como Livir) ou naquelas classes que representam seres humanos, como Comprador ou Funcionário. Então, acabaria acontecendo que classes como Livro, Venda, Pagamento etc. não teriam nenhum método relevante.

Quando uma ou duas classes fazem tudo e as outras são meras pacientes desse processo, não existe propriamente orientação a objetos, mas uma estrutura concentradora. Seria preferível fazer um projeto estruturado bem feito do que um projeto orientado a objetos dessa forma.

Projetistas podem cometer o erro de acreditar que um sistema orientado a objetos é uma simulação do mundo real. Mas isso não é normalmente verdade. O sistema representa as *informações* do mundo real e não as *coisas* propriamente ditas. Há aqui uma diferença sutil, mas importante. Os métodos não correspondem a ações do mundo real, mas à realização interna de contratos de operações e consultas de sistema. Por esse motivo é que os métodos

internos são citados apenas na atividade de projeto e sequer aparecem na atividade de análise.

Sendo assim, projetar software orientado a objetos deve ser compreendido como um processo muito preciso e guiado por padrões já aprendidos, e não simplesmente como o ato de criar classes e associar métodos a elas de forma *ad-hoc*.

9.1. Responsabilidades e Operações Básicas

Para que haja uma boa distribuição de responsabilidades entre os diferentes tipos de objetos, inicialmente será definida uma classificação. Basicamente, há dois grandes grupos de responsabilidades:

a) responsabilidades de *conhecer*, que correspondem às consultas;
b) responsabilidades de *fazer* ou *alterar*, que correspondem às operações.

Ambos os grupos ainda se subdividem em três subgrupos:

a) coisas que objeto conhece ou faz sobre *si mesmo*;
b) coisas que o objeto conhece ou faz a respeito das suas *vizinhanças*;
c) outras coisas que o objeto conhece ou faz não classificadas nos subgrupos anteriores; normalmente *conhecimentos derivados* e *ações coordenadas*.

No caso das responsabilidades de conhecer, os três subgrupos poderiam ser assim caracterizados:

a) *coisas que o objeto conhece sobre si mesmo*: equivale a poder acessar o valor dos atributos do objeto. Tais responsabilidades são incorporadas às classes através de operações básicas de consulta, que são nomeadas com o prefixo get seguido do nome do atributo. Por exemplo, se a classe Pessoa tem um atributo dataNascimento, então o método getDataNascimento() realiza a responsabilidade de conhecer o valor desse atributo;
b) *coisas que o objeto conhece sobre suas vizinhanças*: equivale a poder acessar outros objetos que estão associados diretamente a ele. Essa responsabilidade é então realizada por métodos que acessam o conjunto de objetos associados através de cada uma das associações de um objeto. Tais métodos também são usualmente representados por um prefixo

get seguido do nome de papel da associação. Por exemplo, se a classe Pessoa tem uma associação com Reserva com nome de papel reservas, então o método getReservas() retorna o conjunto de reservas de uma pessoa;

c) *coisas que o objeto conhece de forma derivada*: equivale a conhecimentos que são combinações de outros, por exemplo, uma venda pode saber seu valor total a partir da soma dos preços dos produtos que estão associados a ela. Essa responsabilidade é também identificada pelo prefixo get, seguido do nome da informação (por exemplo, getTotal()), e frequentemente corresponde ao método de acesso de um atributo derivado ou associação derivada.

No caso das responsabilidades de fazer ou alterar informações, os três subgrupos poderiam ser assim caracterizados:

a) *coisas que o objeto faz sobre si mesmo*: corresponde às operações básicas de alteração de atributos, identificadas pelo prefixo set seguido do nome do atributo. Então, se a classe Pessoa tem o atributo dataNascimento, o método setDataNascimento(umaData) realiza essa responsabilidade;

b) *coisas que o objeto faz sobre suas vizinhanças*: corresponde às operações básicas de adição e remoção de associações, identificadas, respectivamente, pelos prefixos add e remove, seguidos do nome de papel da associação. Então, se a classe Pessoa possui associação com Reserva e nome de papel reservas, os métodos addReservas(umaReserva) e removeReservas(umaReserva) realizam essa responsabilidade do ponto de vista da classe Pessoa;

c) *coisas que o objeto faz de forma coordenada*: corresponde a operações múltiplas que alteram objetos e que são coordenados a partir de um objeto que detenha a melhor visibilidade possível para os objetos a serem alterados. Tais operações são conhecidas como *métodos delegados* e são bastante importantes na atividade de projeto porque os demais métodos (básicos) são definidos por padrão, mas os delegados precisam ser elaborados. Por exemplo, se todos os livros de uma determinada venda devem ser marcados como entregues, quem deve coordenar essas atividades de marcação é a própria venda, que possui acesso (ou visibilidade) mais direta para os livros que devem sofrer a operação.

A Tabela 9.1 resume os seis tipos de responsabilidades e os métodos tipicamente associados a cada tipo.

Tabela 9.1: Tipos de Responsabilidades

	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, Associações e atributos derivados consultaInformação() getAtributoDerivado() getAssociaçãoDerivada	Métodos delegados -- nomes variam

Não há um prefixo padrão para métodos derivados, que serão nomeados conforme a operação que executem. Usualmente, esses nomes não devem incluir o nome da classe onde estão implementados. Por exemplo, Livir poderá ter uma operação de sistema encerrarVenda(). Se essa operação for delegada à classe Venda, o nome do método delegado deverá ser simplesmente encerrar(), pois possivelmente será invocado assim: venda.encerrar().

9.2. 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:

a) *por associação*: quando existe uma associação entre os dois objetos de acordo com as definições de suas classes no modelo conceitual;

b) *por parâmetro*: quando um objeto, ao executar um método, recebe outro como parâmetro;

c) *localmente declarada*: quando um objeto, ao executar um método, recebe o outro como retorno de uma consulta;

d) *global*: quando um objeto é declarado globalmente.

Nenhuma das formas de visibilidade é necessariamente simétrica. Ou seja, se x tem visibilidade para y, isso não significa que y tenha necessariamente visibilidade para x.

9.2.1. 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.

Um papel de associação *sempre* pode ser tratado como um conjunto de instâncias. Mas, no caso da multiplicidade para um, ainda é possível interpretar o papel como sendo um objeto individual. Em relação à multiplicidade, pode-se identificar os seguintes tipos:

a) se a multiplicidade de papel for para um, tem-se visibilidade diretamente para uma instância;
b) qualquer outra multiplicidade de papel dá visibilidade a um conjunto de instâncias.

Essa multiplicidade, como será visto, não é restrita apenas ao modelo conceitual. Caso o contrato da operação ou consulta em questão possua uma pre-condição que estabeleça uma multiplicidade mais restrita do que a do modelo conceitual, é a multiplicidade do contrato a que vale.

9.2.1.1. Visibilidade para Um

Na Figura 9.1a, a classe Pagamento tem uma associação *para um* com Venda. Nesse caso, qualquer instância de Pagamento terá visibilidade direta para uma instância de Venda. Dessa forma, quando tais instâncias forem representadas em um diagrama de comunicação, como na Figura 9.1b, a instância de Pagamento poderá enviar mensagens diretamente à instância de Venda.

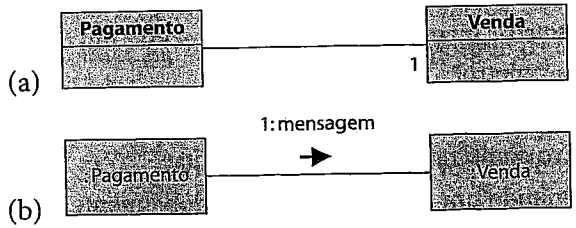


Figura 9.1: (a) Um diagrama de classe com associação para um. (b) Um diagrama de comunicação onde um objeto :Pagamento tem visibilidade por associação para um objeto :Venda.

9.2.1.2. Visibilidade para Muitos

Outras formas de multiplicidade diferentes de *para um*, como "*", "1..*", "0..1", "5" etc. habilitam a visibilidade não para uma, mas para uma coleção de instâncias. O caso 0..1 ainda poderia ser tratado como uma visibilidade *para um* que aceita o objeto *null*, mas por questão de padronização convém tratar esse caso como um conjunto que pode ser unitário ou vazio. A partir de agora, todos esses casos serão tratados genericamente como "*".

Na Figura 9.2a a classe Venda tem uma associação para "*" para a classe Item. Verifica-se que, nesse caso, uma instância de Venda pode enviar mensagens a conjuntos de instâncias de itens. É possível enviar mensagens ao conjunto propriamente dito, como na Figura 9.2b (por exemplo, consultando o tamanho do conjunto ou adicionando um elemento a ele) ou, ainda, enviar mensagens a cada um dos elementos do conjunto, como na Figura 9.2c (por exemplo, solicitando o valor do subtotal de cada item). Ou seja, a mensagem pode ser endereçada a estrutura de dados que contém os elementos ou a cada um dos elementos iterativamente.

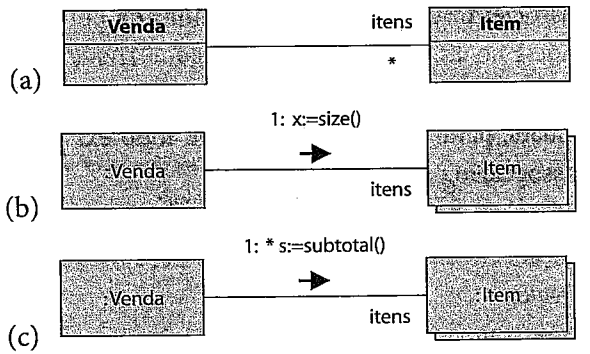


Figura 9.2: (a) Diagrama de classes com associação *para muitos*. (b) Diagrama de comunicação com mensagem para um conjunto. (c) Diagrama de comunicação com mensagem para cada um dos elementos de um conjunto.

No caso da Figura 9.2c, o asterisco antes da mensagem propriamente dita identifica que se trata de uma mensagem enviada a cada um dos elementos do conjunto e não ao conjunto em si, como na Figura 9.2b.

9.2.1.3. Associações Ordenadas

Se a associação for ordenada (OrderedSet e Sequence), além da visibilidade ao conjunto todo, pode-se ter visibilidade a um elemento específico da

associação com base na sua posição: primeiro, ultimo e *n*-ésimo. A Figura 9.3 exemplifica esse caso.

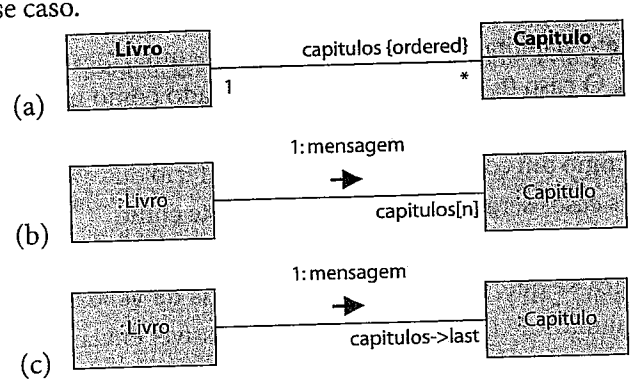


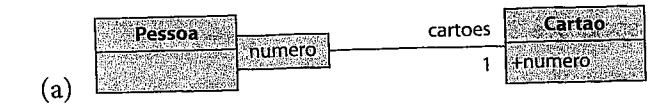
Figura 9.3: Diagrama de classes com associação ordenada. (b) Visibilidade ao *n*-ésimo elemento. (c) Visibilidade ao último elemento.

9.2.1.4. Associações Qualificadas

Se a associação para muitos for qualificada como mapeamento (com multiplicidade 1 no lado oposto ao qualificador), haverá duas formas de visibilidade direta:

a) visibilidade para o conjunto como um todo exatamente como se fosse uma associação para "*";
b) se o objeto do lado do qualificador possuir uma chave para acessar a associação, ele terá visibilidade direta para o elemento qualificado por essa chave.

A Figura 9.4a apresenta um diagrama de classes com uma associação qualificada entre Pessoa e Cartao. A Figura 9.4b demonstra que, nesse caso, uma instância de Pessoa tem visibilidade para o conjunto de seus cartões. A Figura 9.4c mostra que, caso a pessoa em questão possua o valor do qualificador (número do cartão), ela terá visibilidade direta para o elemento qualificado por esse número. Além disso, ela ainda pode acessar todos os elementos da associação, como no caso da Figura 9.2c.



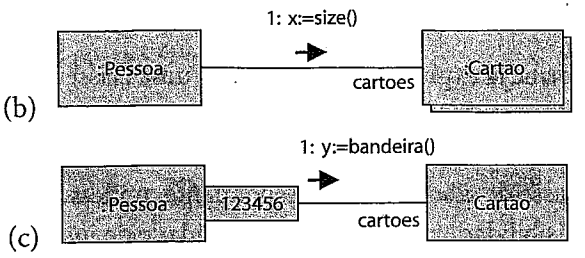


Figura 9.4: (a) Diagrama de classe com associação qualificada. (b) Diagrama de comunicação representando visibilidade para o conjunto. (c) Diagrama de comunicação representando visibilidade para um elemento qualificado.

Por outro lado, se a multiplicidade do lado oposto ao qualificador for diferente de *um*, o que se tem é uma partição, ou seja, uma divisão de um conjunto maior em subconjuntos (Figura 9.5a). Nesse caso, a visibilidade sem o qualificador (Figura 9.5b) representa o conjunto completo (no exemplo, todos os livros) e a visibilidade com o qualificador (Figura 9.5c) representa um subconjunto (no caso, os livros infantis).

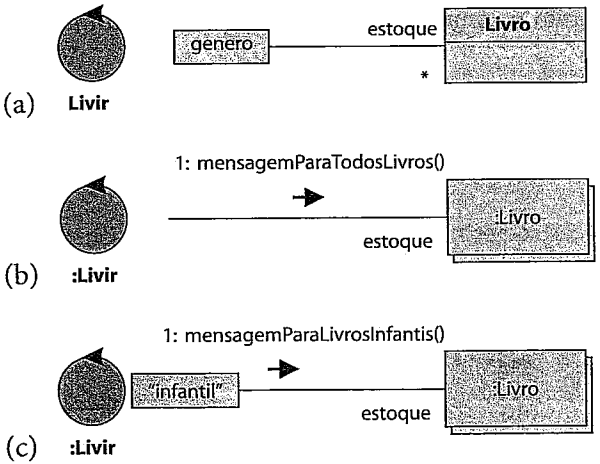


Figura 9.5: (a) Diagrama de classes com qualificador definindo partição. (b) Mensagem para o conjunto como um todo. (c) Mensagem para um subconjunto.

Tanto no caso da Figura 9.5b quanto na 9.5c, trata-se de mensagens enviadas a conjuntos. Para enviar mensagens a cada um dos elementos desses conjuntos, seria necessário prefixar a mensagem com *.

9.2.1.5. Classes de Associação

Conforme já explicado no Capítulo 7, uma classe de associação tem instâncias que são criadas e destruídas conforme associações definidas entre duas outras classes sejam criadas e destruídas. Assim, se A tem uma associação com B, e C é a classe de associação correspondente, pode-se inferir que instâncias de A terão visibilidade para a mesma quantidade de instâncias de B e de C. Simetricamente, instâncias de B terão visibilidade para a mesma quantidade de instâncias de A e C. Por outro lado, uma instância C só terá visibilidade para uma única instância de A e uma única instância de B.

A Figura 9.6a apresenta um diagrama de classe típico com classe de associação. A Figura 9.6b representa a visibilidade que :Pessoa tem para :Empresa (corresponde à multiplicidade do papel emprego, ou seja, uma visibilidade para um conjunto). Já a Figura 9.6c representa a visibilidade que :Pessoa tem para :Emprego. Como uma pessoa tem um emprego para cada empresa, essa visibilidade também corresponde à multiplicidade do papel emprego. Há, então, uma dualidade no papel emprego nesse caso, visto que ele permite que uma :Pessoa acesse tanto o conjunto de empresas quanto o conjunto de empregos.

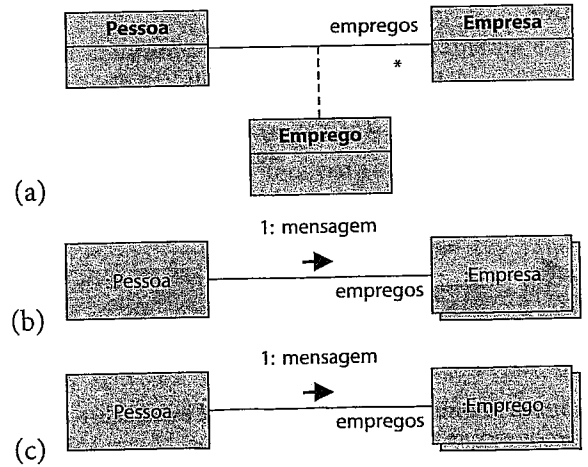


Figura 9.6: (a) Modelo conceitual com classe de associação. (b) Visibilidade usual para um conjunto dado pelo papel. (c) Visibilidade para um conjunto obtido a partir da classe de associação.

A classe de associação também funciona como mapeamento similarmente à associação qualificada. No caso da Figura 9.6a, esse mapeamento equivale a mapear um emprego para cada empresa onde a pessoa trabalha.

Assim, havendo uma empresa dada, é possível determinar de forma única um emprego de uma dada pessoa (Figura 9.7).

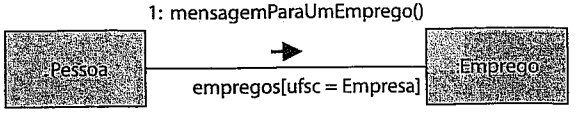


Figura 9.7: Uma representação de visibilidade onde a associação qualificada funciona como um mapeamento.

Ainda na Figura 9.7, observa-se que :ufsc deve ser uma instância da classe Empresa para que o acesso direto ao emprego de :Pessoa seja possível.

Finalmente, a Figura 9.8 mostra a visibilidade que instâncias da classe de associação têm das demais classes participantes. No caso, uma visibilidade necessariamente *para um*.

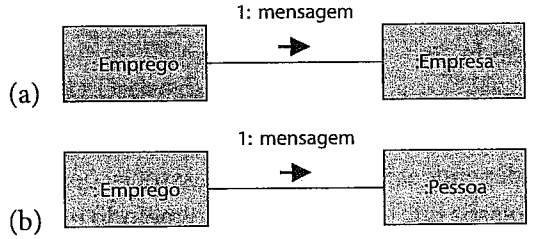


Figura 9.8: Visibilidade da classe de associação para as classes participantes: (a) Visibilidade de :Emprego para :Empresa. (b) Visibilidade de :Emprego para :Pessoa.

9.2.1.6. Influência das Precondições na Visibilidade por Associação

Quando uma precondição de operação ou consulta de sistema restringe ainda mais uma multiplicidade de papel, será sempre a precondição que vai valer como determinante da visibilidade no contexto daquela operação.

Por exemplo, se um diagrama de classe como o da Figura 9.9a define que uma venda pode ter ou não um pagamento, mas a operação sendo executada tem como precondição algo como:

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

então, *no contexto daquela operação*, passa a valer a visibilidade *para um* e não mais *para zero ou um*. É como se, durante a execução dessa operação, o modelo conceitual fosse temporariamente alterado para ficar como na Figura 9.9b.

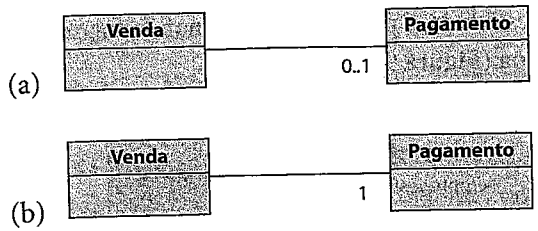
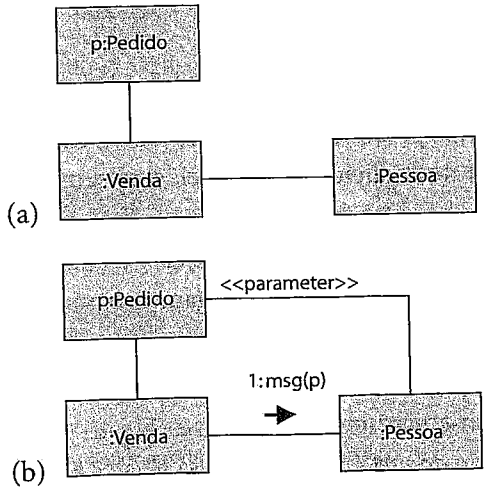


Figura 9.9: (a) Visibilidade opcional de Venda para Pagamento. (b) Como fica a visibilidade temporariamente durante uma operação que tenha como precondição a necessidade da existência do pagamento para uma determinada instância de Venda.

9.2.2. Visibilidade por Parâmetro

A visibilidade por parâmetro é obtida quando um objeto, ao executar um método, recebe outro objeto como parâmetro. Não precisa haver, nesse caso, associação entre as classes para que o primeiro objeto possa se comunicar com o segundo.

Por exemplo, uma instância de Pessoa, ao executar o método msg, recebe como parâmetro uma instância de Pedido identificada pelo argumento p. Nesse caso, independentemente de existir associações entre Pessoa e Pedido, a instância de Pessoa que está executando o método passa a ter visibilidade *por parâmetro* para o pedido p (Figura 9.10).



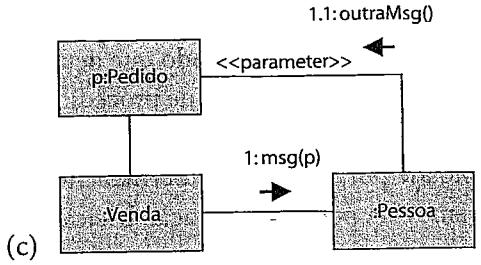


Figura 9.10: (a) Situação inicial, onde :Venda tem visibilidade para p:Pedido e :Pessoa. (b) Após :Venda enviar mensagem msg para :Pessoa passando p:Pedido como parâmetro, :Pessoa adquire visibilidade por parâmetro para p:Pedido. (c) A partir desse ponto, :Pessoa pode se comunicar diretamente com p:Pedido.

No caso da visibilidade por parâmetro, deve-se admitir que o objeto que enviou a mensagem msg para a instância de Venda detinha algum tipo de visibilidade para a instância de Pessoa que enviou como parâmetro.

9.2.3. Visibilidade Localmente Declarada

Outra forma de visibilidade possível ocorre quando um objeto, ao enviar uma consulta a outro, recebe *como retorno* um terceiro objeto, como demonstrado na Figura 9.11.

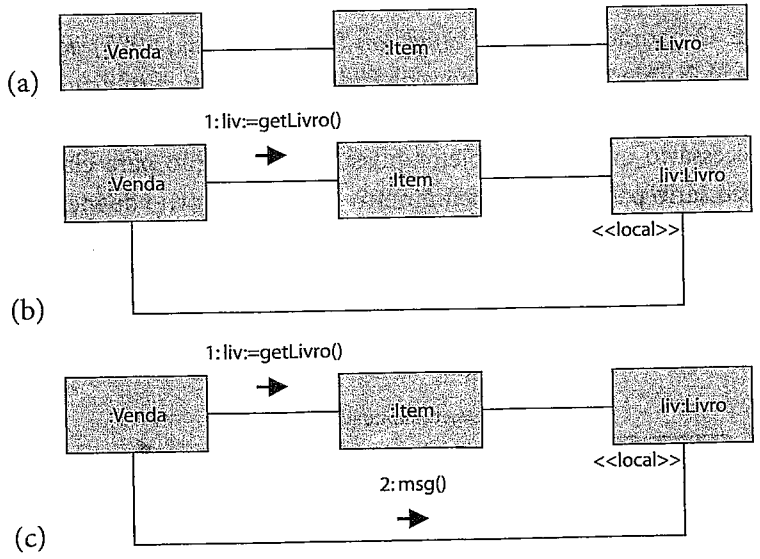


Figura 9.11: (a) Situação inicial. (b) :Venda envia uma mensagem a :Item e recebe liv:Livro como retorno. A partir desse ponto, :Venda adquire visibilidade local para liv:Livro. (c) Agora :Venda pode se comunicar diretamente com liv:Livro.

Tanto a visibilidade local quanto a visibilidade por parâmetro somente são válidas durante a execução do método que deu origem a elas, ou seja, assim como variáveis locais e parâmetros de procedimentos, elas só têm validade dentro da operação onde foram declaradas, desaparecendo após o término desta. Já a visibilidade por associação é permanente, persistindo até que seja explicitamente removida por uma operação básica de destruição de associação.

Deve-se evitar ao máximo usar a visibilidade por parâmetro e a localmente declarada para enviar mensagens a objetos.

Por princípio, deve-se optar por enviar mensagens apenas através de ligações de visibilidade por associação. Objetos recebidos como parâmetro ou retorno devem apenas ser repassados como parâmetro, se possível. Já a visibilidade local deveria, por princípio, ser usada *apenas* quando o método que retorna o objeto efetivamente *cria* o objeto. Esse princípio é conhecido em padrões de projeto como "*Não Fale com Estranhos*" ou "*Law of Demeter*" (Lieberherr & Holland, 1989).

9.2.4. Visibilidade Global

Existe visibilidade global para um objeto quando ele é declarado globalmente. O padrão de projeto *Singleton* (Gamma *et al.*, 1995) admite uma instância globalmente visível apenas quando ela é a *única instância possível da sua classe*. Isso faz sentido porque, se essa classe possui uma única instância, não é necessário que outros objetos possuam associação para ela. Também não é necessário passá-la como parâmetro ou como retorno de métodos.

Um exemplo seriam classes de projeto que não representam conceitos do modelo conceitual, mas serviços, como: ConversorDeMoedas, ContadorDeTempo etc. Haverá uma única instância dessas classes de serviço que podem ser acessadas por qualquer objeto a qualquer tempo (Figura 9.12).

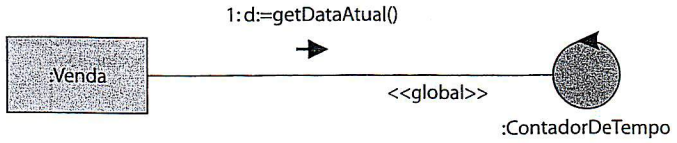


Figura 9.12: Visibilidade global.

9.3. Realização Dinâmica das Pós-condições

Já foi visto que os contratos de operação de sistema apresentam um conjunto de pós-condições que correspondem a certas operações básicas de criação e destruição de instâncias, criação e destruição de associações e modificação de valor de atributos. Os contratos apenas indicam *o que* deve acontecer, sem mostrar como mensagens reais são trocadas entre objetos para realizar tais ações. Os diagramas de comunicação podem ser usados exatamente para mostrar como essas trocas são feitas. Os seguintes princípios devem ser observados:

a) a visibilidade entre instâncias de objetos é regida pela multiplicidade de papel estabelecida no modelo conceitual e eventuais precondições que restringem mais ainda tal multiplicidade;

b) cada uma das operações básicas estabelecidas em pós-condições deve ser realizada no diagrama de comunicação através do envio de uma mensagem básica ao objeto que detém a *responsabilidade* de forma mais imediata;

c) o fluxo de execução em um diagrama de comunicação ou sequência que representa uma operação de sistema *sempre* inicia na instância da *controladora de sistema* recebendo uma mensagem da *interface*;

d) 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.

Aplicam-se, ainda, padrões de projeto que vão auxiliar o projetista a construir métodos que sejam efetivamente reusáveis e com baixo acoplamento. Santos (2007) apresenta uma sistematização desses princípios em um sistema de regras de produção capaz de gerar bons diagramas de comunicação a partir de uma ampla gama de contratos.

9.3.1. Criação de Instância

Quando um contrato estabelece que um instância foi criada, algum objeto deve enviar uma mensagem de criação no respectivo diagrama de comunicação. O padrão "Criador" (Gamma, 1995) estabelece que deva ser prioritariamente:

a) um objeto de uma classe que tenha relação de *composição* ou *agregação* para com o objeto a ser criado;
b) um objeto que tenha relação de *um para muitos* com o objeto a ser criado;
c) um objeto que tenha os *dados de inicialização* do objeto a ser criado e cuja classe esteja associada à classe dele.

Cada regra só se aplica se a anterior não for possível ou quando houver empate.

Considerando o diagrama da Figura 9.13, a criação de uma instância da classe Item deverá ser feita, de acordo com o padrão *Criador*, por uma instância da classe Venda, visto que existe entre elas uma relação de agregação. Se não houvesse essa agregação, a criação poderia ser feita pela classe Livro, mas nunca pelas classes Pessoa ou Livir, que não têm associação com Item.

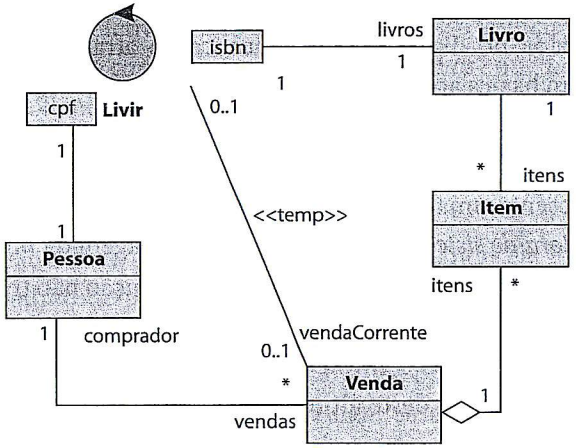


Figura 9.13: Modelo conceitual de referência.

Um fragmento de uma possível operação de sistema com a criação de uma instância de Item é mostrado a seguir:

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


Aplicando-se tais princípios a esse contrato no diagrama de comunicação da Figura 9.14, deve-se iniciar o fluxo pela controladora: a controladora recebe a mensagem referente à operação de sistema diretamente da interface. Essa mensagem não deve ser numerada. Como Livir não tem visibilidade para Item, ela não pode criar essa instância, devendo delegar (mensagem 1) a uma instância de Venda, a venda corrente, que, por sua vez, fará a criação propriamente dita (mensagem 1.1).

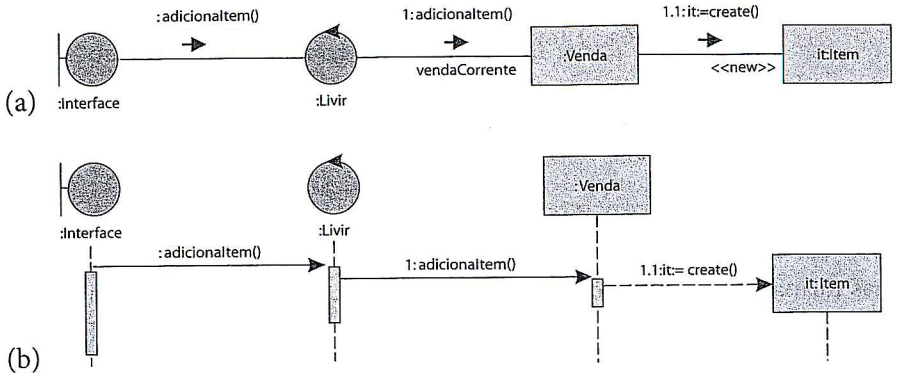


Figura 9.14: Diagrama de comunicação (a) e de sequência (b) com uma mensagem de criação de instância.

É importante observar que os diagramas da Figura 9.14 possuem três tipos de mensagens:

a) mensagem que ativa uma *operação de sistema* entre a interface e a controladora :Livir;
b) mensagem *delegada* entre a controladora e a instância de Venda;
c) mensagem referente a uma *operação básica* entre : Venda e it:Item.

O primeiro e o terceiro tipo de mensagens *sempre* aparecerão nesses diagramas: a operação de sistema uma única vez, consistindo na raiz da árvore de mensagens enviadas. As mensagens básicas aparecerão na mesma quantidade das pós-condições dos contratos (incluindo criação de instância, que usualmente aparece na cláusula def). As mensagens básicas consistem nas folhas da árvore de mensagens. Elas não devem invocar novas mensagens.

Já o segundo tipo, operações delegadas, é opcional, aparecendo sempre que a controladora não for capaz de realizar as pós-condições sem delegar a outros objetos alguma responsabilidade. As operações delegadas são os ramos intermediários da árvore de mensagens e *sempre* precisam ter alguma continuação. O fluxo de mensagens só termina nas mensagens básicas (folhas).