9.3.2. Criação de Associação

Já foi comentado que, quando uma instância é criada, ela deve ser imediatamente associada a alguma outra instância para que seja acessível. Então, o contrato da seção anterior deve ser complementado para realizar também essa pós-condição adicionando o item à venda corrente:

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

Conforme já comentado, a pós-condição que cria uma associação tem duas formas simétricas: no caso anterior, vendaCorrente^addItens(item) também poderia ter sido escrita como item^addVenda(vendaCorrente). Embora ambas produzam o mesmo efeito final, usualmente será mais prático que o objeto que recebe a mensagem seja aquele mais próximo da controladora. No caso, a escolha seria por enviar a mensagem de :Venda para :Item, já que :Venda tem uma ligação direta com a controladora enquanto :Item só tem ligações indiretas (Figura 9.15).

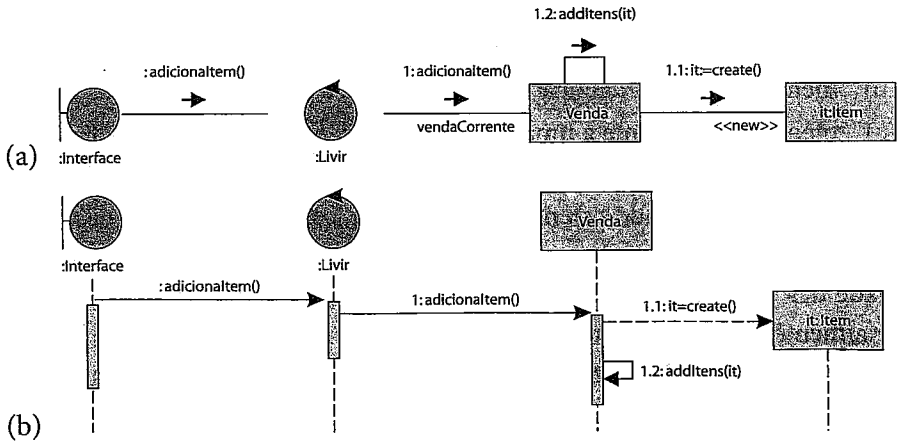


Figura 9.15: Um diagrama de comunicação (a) e de sequência (b) com uma mensagem de criação de associação.

Para que esse contrato fosse ainda mais completo, deveria haver a associação do item com um livro existente, cujo código fosse passado como parâmetro. Para acessar uma instância de Livro a partir de seu código, pode-se usar a mensagem básica de consulta, ou *get*. A mensagem *get* básica retorna todas as instâncias associadas ao papel, ou seja, um conjunto, mas quando a associação é qualificada, pode-se usar uma mensagem *get* com parâmetro correspondendo ao qualificador do objeto, como na Figura 9.16. O contrato completo é apresentado a seguir:

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

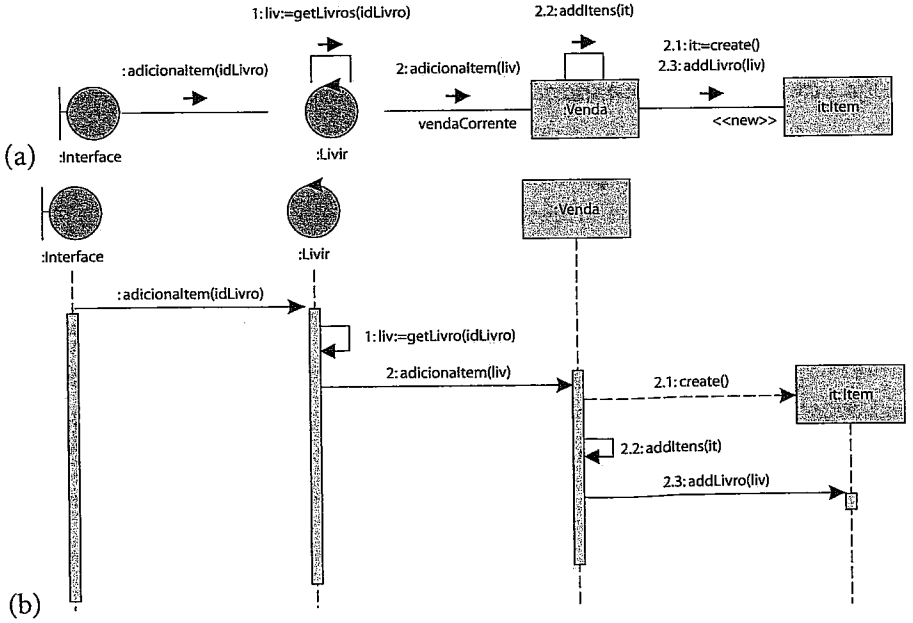


Figura 9.16: Um diagrama de comunicação (a) e de sequência (b) com uma criação de instância e duas operações de criação de associação.

Observa-se, na Figura 9.16, que a operação de sistema recebe como parâmetro um identificador alfanumérico. A controladora que tem acesso imediato ao conjunto dos livros faz o acesso através da consulta básica getLivros, passando o idLivro e recebendo liv como retorno, para o qual passa a ter visibilidade local. Em seguida, a controladora delega à venda a operação de adição, mas passando a instância liv em vez do simples identificador. A venda, por sua vez, tem capacidade para executar diretamente as três operações básicas exigidas no contrato: a criação do item, a associação da venda com o item e a associação do item com o livro, já que sua classe tem associação direta com a classe Item.

9.3.3. Modificação de Valor de Atributo

Outra operação básica é a operação de modificação de valor de atributo, que pode ser enviada pelo objeto que contém o atributo ou por um de seus vizinhos diretos. Continuando o exemplo anterior, adiciona-se ao contrato a necessidade de complementar o item com um valor para o atributo quantidade (que não aparece na Figura 9.13, mas está implícito). Então, além das operações já executadas, a venda ainda poderá alterar o valor desse atributo (recebido como parâmetro) através de uma mensagem básica do tipo *set*, como na Figura 9.17. Já o contrato completo aparece a seguir:

```
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)
```

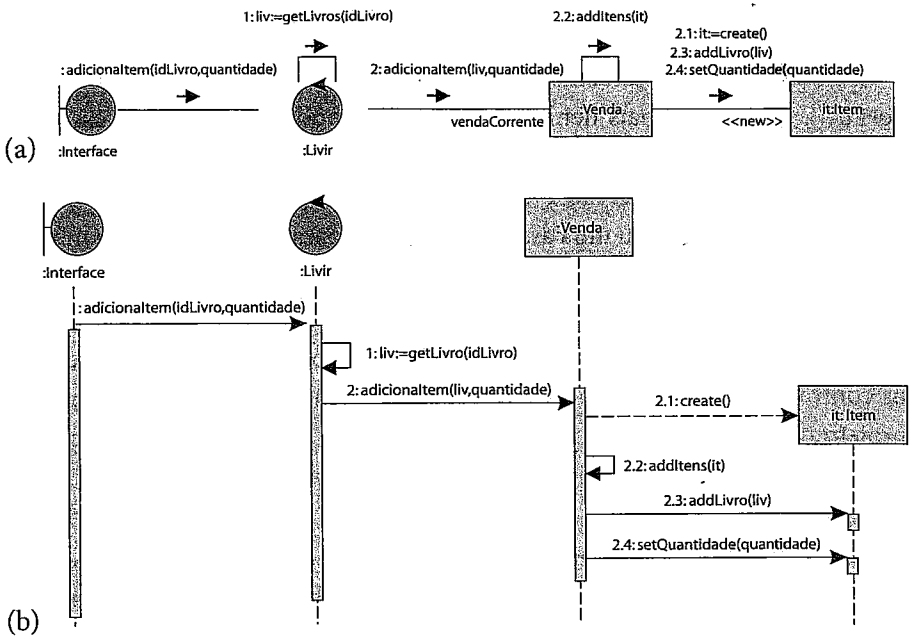



Figura 9.17: Um diagrama de comunicação (a) e de sequência (b) com operação básica de alteração de valor de atributo.

Como se pode observar até aqui, os diagramas de comunicação podem ser substituídos por diagramas de sequência, caso se prefira trabalhar com eles. A principal inconveniência é que os diagramas de sequência não apresentam as linhas de visibilidade explicitamente.

Uma dúvida que poderia ser suscitada nesse ponto é por que a consulta getLivro é enviada da controladora para si própria e não para o conjunto de livros. Isso se deve ao fato de que aqui já se trata de uma operação de consulta padronizada sobre associações. A classe que possui uma associação tem visibilidade para um conjunto de instâncias de outra classe. A forma básica de acessar esse conjunto é através do envio da mensagem get, seguido do nome de papel à própria instância associada. No caso anterior, Livir tem associação com Livro (papel livros), e a forma correta de obter o conjunto de livros é enviando a mensagem getLivros à instância de Livir.

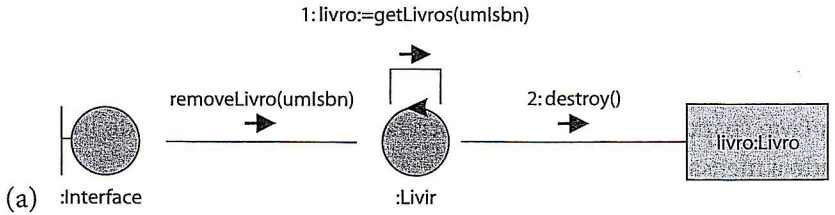
9.3.4. Destruição de Instância

A destruição de uma instância é representada pelo envio da mensagem destroy(). Após isso, mais nenhuma mensagem pode ser enviada a essa instância. Aplicam-se aqui também os mesmos princípios do padrão *Criador*: o objeto que destrói uma instância deve ter agregação ou composição com o objeto a ser destruído ou uma associação de um para muitos, ou, pelo menos, ser associado ao objeto.

Deve-se tomar todos os cuidados estruturais para evitar que uma instância destruída mantenha associações com objetos que precisariam dela. Se os contratos de operação de sistema estiverem bem formados, a atividade de projeto dos diagramas de comunicação só precisa representar as pós-condições já mencionadas, sem necessidade de novas checagens de consistência que já terão sido consideradas.

Na Figura 9.18, apresenta-se um modelo em que uma instância de Livro é removida. A operação considera que o livro em questão não está associado a nenhum item. Segue o respectivo contrato:

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



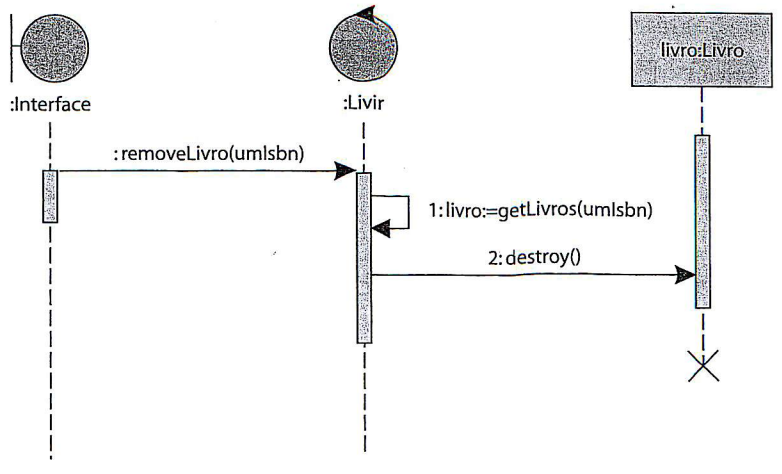


Figura 9.18: Diagrama de comunicação (a) e sequência (b) com operação básica de destruição de instância.

9.3.5. Destruição de Associação

A destruição de uma associação é realizada pela operação básica prefixada por remove, seguida do nome de papel. A Figura 9.19 apresenta um exemplo de remoção de associação para a operação de sistema que faz um automóvel trocar de dono, conforme o contrato a seguir:

```
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)
```

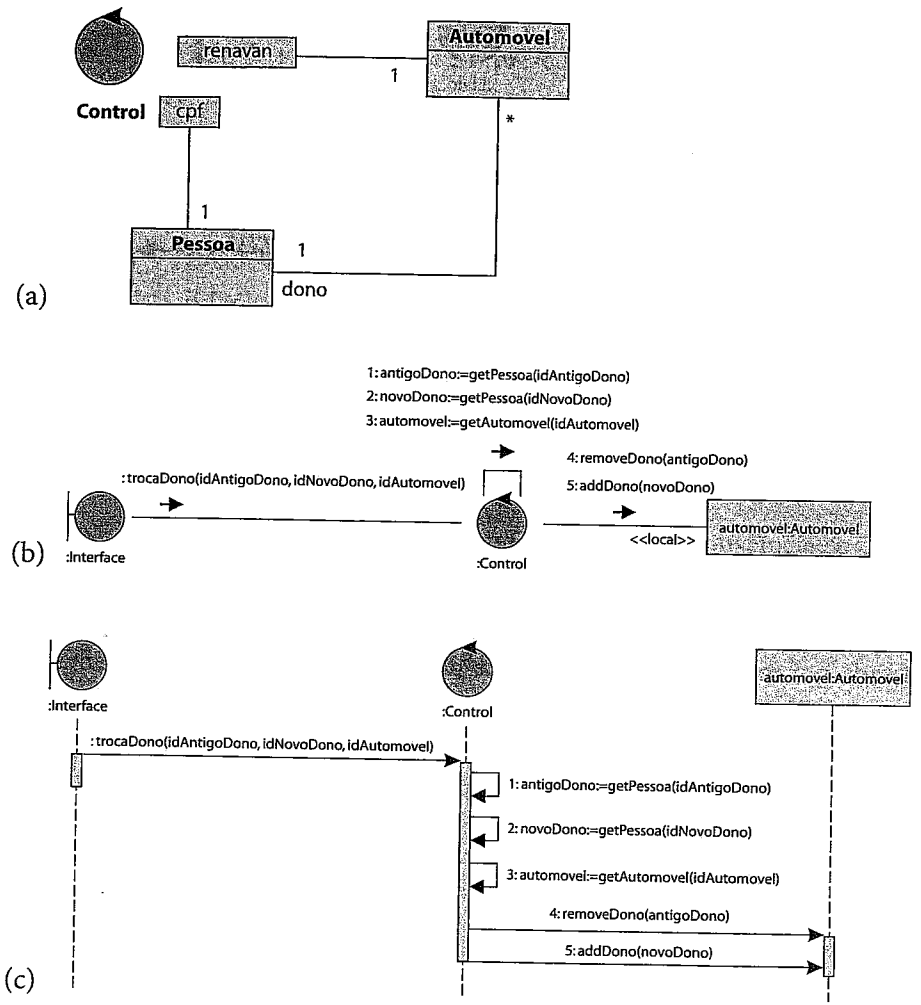


Figura 9.19: (a) Modelo conceitual de referência. Diagrama de comunicação (b) e de sequência (c) com operação básica de remoção de associação.

9.3.6. Pós-condições Condicionais

Conforme já explicado, por vezes pode-se afirmar que uma pós-condição só é obtida quando determinadas condições iniciais são satisfeitas. Nesses casos, é possível usar condicionais nas mensagens do diagrama de sequência de forma semelhante à condição de guarda que existe nos diagramas de atividade e de máquina de estados. Por exemplo, uma operação de sistema sobre

uma venda poderá aplicar um desconto de 10% caso o valor total da venda seja superior a 1.000 reais. O contrato seria algo como:

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

O modelo dinâmico deveria representar a cláusula condicional como na Figura 9.20.

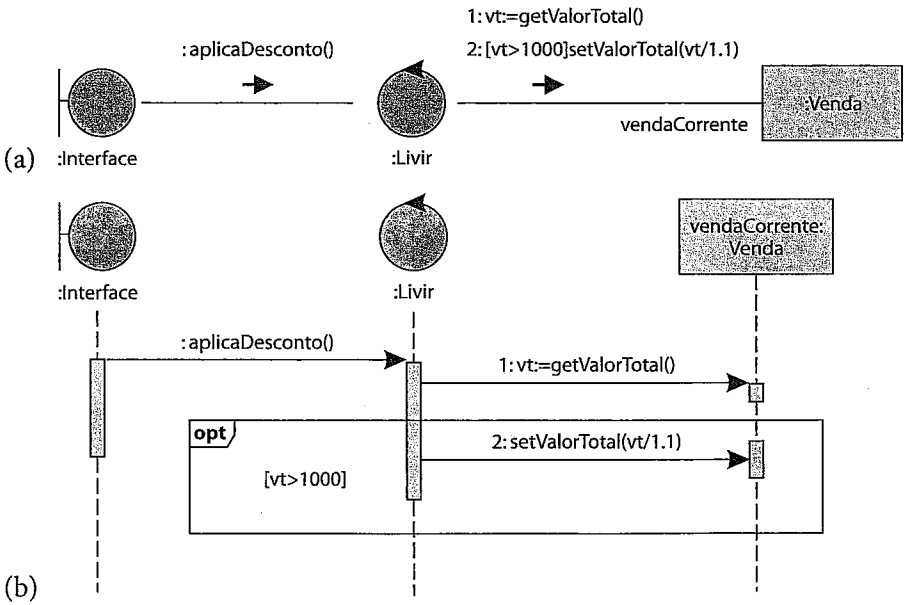


Figura 9.20: Diagrama de comunicação (a) e sequência (b) com mensagem condicional.

Observa-se que, se o valor vt for menor ou igual a 1.000, a operação não produz nenhum resultado.

Caso a condicional envolvesse uma estrutura do tipo *if-then-else-endif*, deveria haver uma condicional para a cláusula *then* e outra condicional negando a primeira para a cláusula *else*.

No caso de estruturas *CASE*, deveria haver uma mensagem condicional para cada um dos casos definidos.

9.3.7. Pós-condições sobre Coleções

Quando operações de sistema têm contratos com pós-condições que especificam alterações em coleções de objetos, pode-se usar a estrutura de repetição "*" para indicar que uma mensagem é enviada a todos os elementos de uma coleção. Por exemplo, a seguinte operação de sistema aumenta o preço de todos os livros em 10%:

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

Os diagramas da Figura 9.21 foram elaborados para esse contrato.

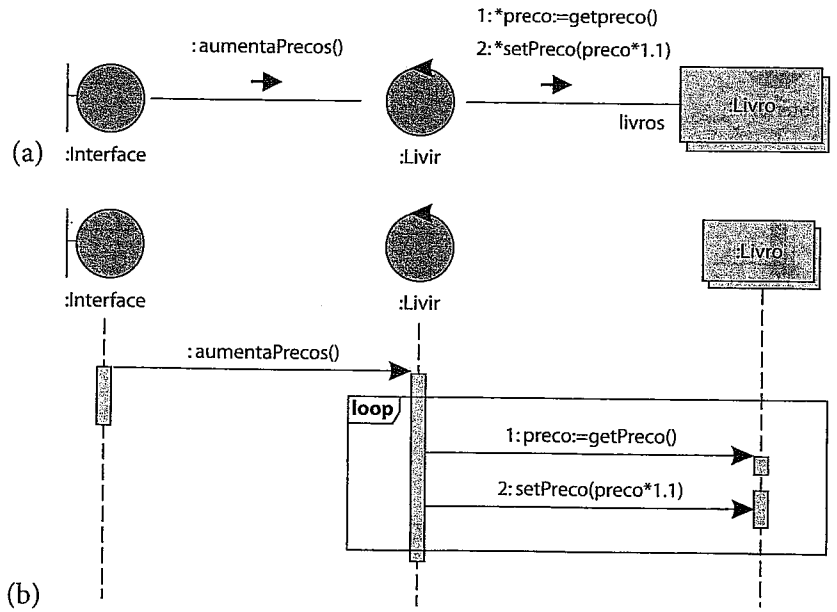


Figura 9.21: Diagrama de comunicação (a) e sequência (b) com iteratividade.

Outra situação comum ocorre quando a iteração não deve ocorrer sobre todos os elementos de uma coleção, mas apenas sobre aqueles que satisfazem um determinado critério. O contrato a seguir apresenta uma operação que majora em 10% apenas os livros cujo preço é inferior a 100 reais:

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

Os diagramas de comunicação e sequência correspondentes são mostrados na Figura 9.22.

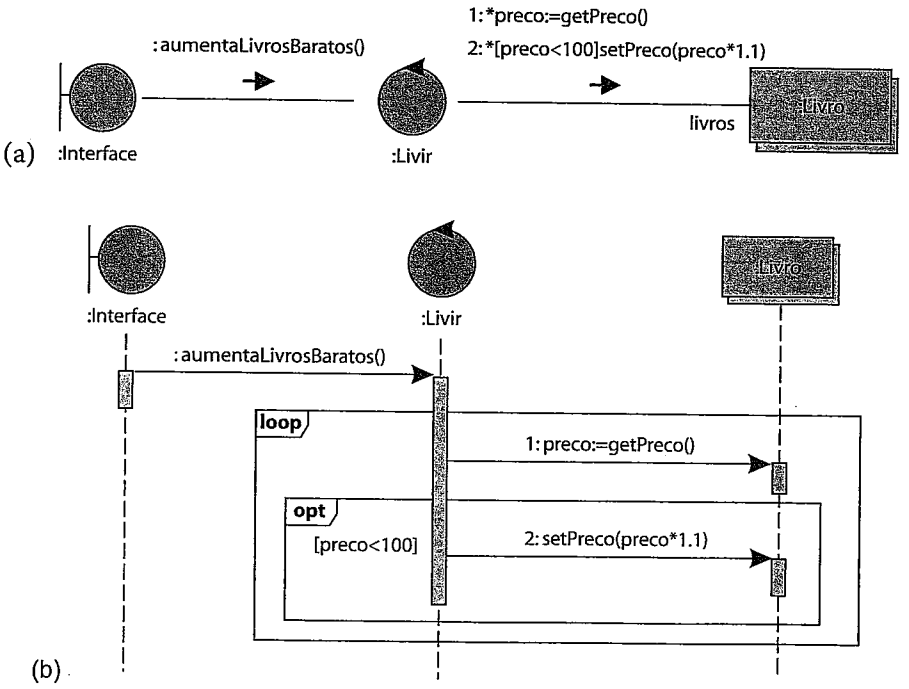


Figura 9.22: Diagrama de comunicação (a) e sequência (b) com iteração e filtro.

9.4. Consultas de Sistema

Em relação às consultas de sistema, pode-se observar no diagrama de comunicação também a presença de três tipos de mensagens:

a) a *consulta de sistema*, que é única em cada diagrama, vem do diagrama de sequência de sistema e inicia as chamadas, sendo a raiz da árvore de mensagens. Ela também é a responsável por retornar o resultado, que deve ser uma estrutura *DTO* ou um alfanumérico;

b) as *consultas básicas*, que são as folhas do diagrama e correspondem às operações *get* básicas sobre atributos ou papéis de associação. Elas correspondem às folhas da árvore de mensagens e não precisam ser mais detalhadas no diagrama;

c) as *consultas intermediárias*, que correspondem aos atributos derivados quando seu retorno é um valor alfanumérico ou a consultas intermediárias propriamente ditas quando o retorno é uma estrutura de dados como uma lista, conjunto ou tupla.

A Figura 9.23 apresenta um diagrama para o contrato de consulta de sistema CRUD para consultar Livro, conforme apresentado no capítulo anterior.

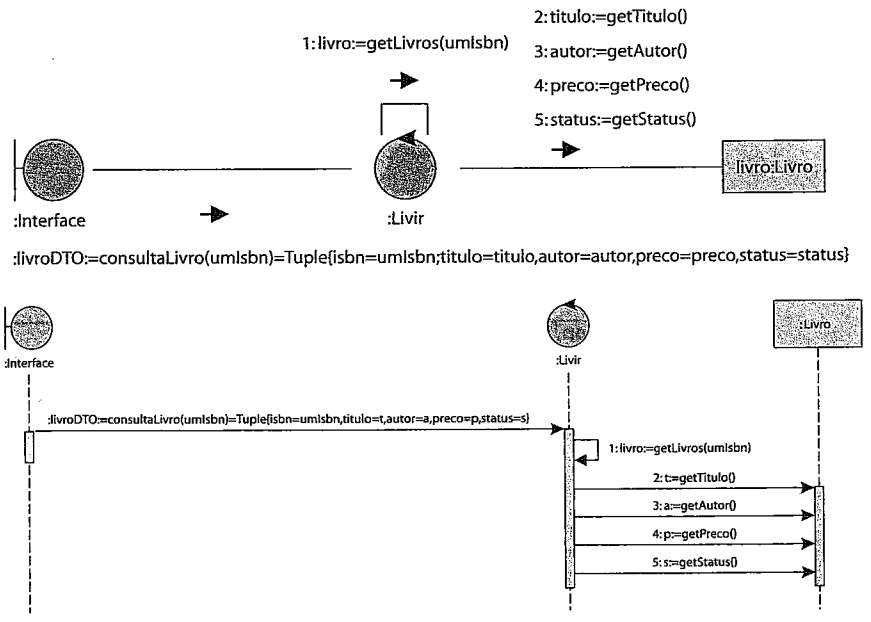


Figura 9.23: Diagrama de comunicação (a) e sequência (b) para consulta de sistema *CRUD-retrieve*.

Porém, nem o diagrama de sequência, nem o diagrama de comunicação possuem estruturas adequadas para representar combinações complexas de valores como funções matemática, filtros etc., o que faz com que não sejam

boas ferramentas para representar tais contratos. No exemplo anterior, o acesso a cada um dos atributos do livro é indicado por uma mensagem individual, e elas são combinadas na chamada da consulta de sistema após o sinal de "=", onde se especifica como a tupla que retorna da consulta de sistema é formada.

Em outras palavras, o trabalho que se tem para criar diagramas de consulta de sistema usualmente não compensa o que se possa aprender de novo com eles. Nesses casos, recomenda-se o uso de algoritmos ou programação direta a partir da especificação do contrato da consulta, tendo sempre em vista o seguinte:

a) sempre que uma consulta intermediária retornar um valor alfanumérico simples, verificar se não faria sentido transformar esse valor em um atributo derivado da classe com grande potencial de reusabilidade;

b) sempre que uma consulta intermediária retornar um objeto ou conjunto de objetos, verificar se não faria sentido representá-la como uma associação derivada;

c) sempre que possível, criar novos padrões de consulta que permitam agrupar várias consultas básicas em estruturas de mais alto nível, como, por exemplo, uma consulta que já aplique um filtro nos resultados ou que transforme os atributos de uma classe diretamente em uma tupla.

9.4.1. Padrões de Consulta com Filtro

Nem sempre o que se deseja é uma simples consulta que retorne todas as instâncias de uma classe ou todas as instâncias associadas a um determinado objeto. Muitas vezes, aplicam-se filtros nas consultas, desejando, por exemplo, apenas as venda do mês de janeiro ou apenas os clientes que moram em apartamento etc.

Existem pelo menos três padrões para tratar essa diversidade de consultas com filtros:

a) implementar na classe que detém a responsabilidade uma única consulta que retorne todos os elementos da associação e deixar que o objeto que solicitou tais elementos faça a filtragem;

b) implementar uma consulta específica para cada filtro possível;

c) implementar uma consulta genérica que tenha como parâmetro um objeto filtro.

Nos casos *a* e *c*, implementa-se uma única consulta, o que deixa a classe que detém a responsabilidade mais simples, mas, por outro lado, gera mais código nas classes que solicitam a informação.

No caso *b*, implementam-se mais métodos na classe que detém a responsabilidade, mas as classes que solicitam a informação farão chamadas mais simples, recebendo a informação pronta.

A escolha entre um padrão ou outro deve ser tomada pelo projetista em função da quantidade de possíveis filtros e da quantidade de possíveis chamadas. Se existem poucas possibilidades de filtros e muitas chamadas, a melhor escolha é o padrão *b*. Se a quantidade de filtros é grande e há poucas chamadas para cada um individualmente, deve-se escolher entre os padrões *a* e *c*. O padrão *a* é mais direto, mas gera muito trabalho na fase de programação: cada vez que um mesmo filtro for usado, o código deve ser repetido na classe que chama a consulta. Já o padrão c requer que se conheça o funcionamento da classe Filtro, mas depois de dominado tende a ser mais simples do que o padrão *a*.

Um *objeto filtro* é um parâmetro que é passado ao método de consulta. Esse objeto deve conter atributos e associações para outros objetos, de forma que o método de consulta faça uma verificação e só retorne as instâncias que correspondem à definição dada pelo objeto filtro.

Por exemplo, no modelo da Figura 9.24, deseja-se fazer uma consulta para retornar os livros cadastrados no sistema Livir.

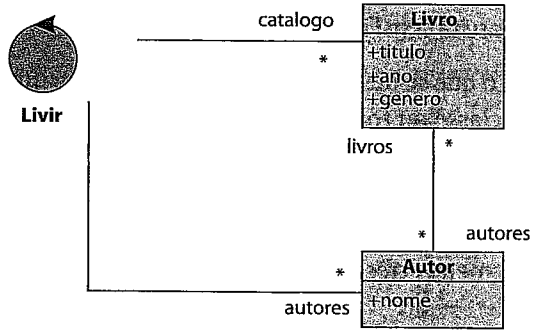


Figura 9.24: Modelo de referência para exemplos de consultas.

Os filtros possíveis são:

a) pelo autor;
b) pelo ano de publicação (inicial e final);
c) pelo gênero.

Aplicando o padrão *a*, implementa-se na classe Livir (que detém a responsabilidade porque tem visibilidade para todo o conjunto de livros) uma

única consulta básica getCatalogo():SET[Livro] , que retorna o conjunto de todos os livros da livraria. Para implementar consultas pelos filtros, deve-se chamar getCatalogo() e aplicar o filtro ao resultado.

Aplicando o padrão *b*, deve-se implementar, na classe Livir, três consultas que podem ser consideradas variantes da consulta básica:

a) getCatalogoPorAutor(umAutor:Autor): SET[Livro]
b) getCatalogoPorAno(umIntervalo:Intervalo): SET[Livro]
c) getCatalogoPorGenero(umGenero:String): SET[Livro]

Cada uma das consultas retorna apenas os elementos que satisfazem os parâmetros passados.

O padrão *c* exige a definição de uma classe para representar o objeto filtro. No caso das consultas derivadas de getCatalogo, será necessário definir uma classe FiltroLivros segundo a definição da Figura 9.24.

Figura 9.25: Uma classe FiltroLivros para consultas de catálogo.

Ainda seria possível implementar essa classe com uma associação à classe Autor para evitar o uso do atributo autorNome. Mas, nesse caso especificamente, pode ser interessante manter a classe FiltroLivro (que não é conceitual) independente das classes conceituais.

O método de consulta getCatalogo(umFiltro:FiltroLivro):SET[Livro] pode então ser implementado assim:

```
CLASSE Livir
  VAR PRIVADA catalogo : SET[Livro]

MÉTODO getCatalogo(umFiltro:FiltroLivro):SET[Livro]
  VAR resultado:SET[Livro]
  PARA CADA livro EM catalogo FAÇA
    SE umFiltro.getNomeAutor = livro.getAutor().getNome
    OU umFiltro.getNomeAutor().isNull() ENTÃO
      SE umFiltro.getIntervalo().contains(livro.getAno))
      OU umFiltro.getIntervalo().isNull() ENTÃO
```

```
            SE umFiltro.getGenero() = livro.getGenero()
            OU umFiltro.getGenero().isNull() ENTÃO
              resultado.add(livro)
            FIM SE
          FIM SE
        FIM SE
      FIM SE
    FIM MÉTODO
FIM CLASSE
```

Então, uma consulta, para retornar os livros entre 2004 e 2006 do autor "raul", deveria ser invocada com a passagem de um objeto filtro devidamente instanciado. Como não se especifica a editora, serão retornados resultados de todas as editoras:

```
...
meuFiltro := FiltroLivro.newInstance()
meuFiltro.setAutorNome("raul")
meuFiltro.setPeriodo(Intervalo[2004..2006])
livros := self.getCatalogo(meuFiltro)
...
```

9.5. Delegação e Acoplamento Baixo

Até aqui, foram vistas algumas técnicas para construção de diagramas de comunicação para realizar contratos. Mas, em várias situações, há mais de uma opção de projeto e mais de um meio de delegar mensagens. A partir de agora, será aprofundada a discussão sobre os padrões de projeto que devem ser usados para produzir um projeto elegante.

Como visto anteriormente, muitas vezes acontece que o objeto que detém o fluxo de controle de execução não tem visibilidade para o objeto com uma responsabilidade a executar (por exemplo, alterar o valor de um atributo). Nesse caso, pode-se identificar duas abordagens diametralmente opostas para o projeto:

a) o objeto que detém o fluxo de informação procura obter uma referência (local) para o objeto que poderia executar a ação e comunica-se diretamente com ele;
b) o objeto delega o envio da mensagem a outro que tenha contato mais direto com o objeto que poderia executar a ação.

Pode-se parodiar essas abordagens da seguinte forma: imagine que João seja chefe de Pedro. João quer contratar um *buffet* para a festa do escritório, mas não conhece nenhuma empresa que faça tal serviço. De alguma maneira, João fica sabendo que Pedro tem contato com uma empresa de *buffet*. Então:

a) na primeira abordagem, João solicita a Pedro que lhe dê o telefone da empresa e, depois, João faz a encomenda direto à empresa. A única coisa que Pedro faz é passar o contato a João;
b) na segunda abordagem, João passa os parâmetros a Pedro (quantas pessoas, que tipo de comida, até quanto pode gastar etc.) e pede a Pedro que contrate o *buffet*. A festa acontece e João nunca fica sabendo qual é o telefone do *buffet*. Ele delegou tarefa a Pedro.

Embora na vida real possa ser interessante que as pessoas tenham muitos contatos e relacionamentos, tal não ocorre em sistemas orientados a objetos. Quanto mais conectados os objetos estiverem, mais complexos serão os sistemas. Então, é necessário evitar ao máximo criar conexões entre objetos que não estejam já conectados pelo modelo conceitual.

A primeira abordagem chama-se *concentração* e faz com que o objeto que detém o fluxo de informação procure obter acesso a todos os objetos necessários e comande ele mesmo as atividades. A segunda abordagem chama-se *delegação*, e faz com que o objeto procure delegar as responsabilidades distribuindo-as entre vários outros objetos, quando necessário. A segunda abordagem normalmente é preferível, pois provê maior potencial de reuso, criando métodos intermediários ou delegados nas classes.

Um exemplo concreto disso é mostrado a partir do modelo conceitual da Figura 9.26 (onde a quantidade de atributos foi reduzida ao mínimo necessário para o exemplo). O contrato de consulta de sistema a seguir tem como objetivo obter o valor total da venda corrente. Esse valor total deve ser calculado a partir do valor e quantidade de cada um dos itens da venda:

```
Context Livir::getTotalVendaCorrente():Moeda
  pre:
```

```
    vendaCorrente→size() = 1
  body:
    vendaCorrente.itens→sum(quantidade*valor)
```

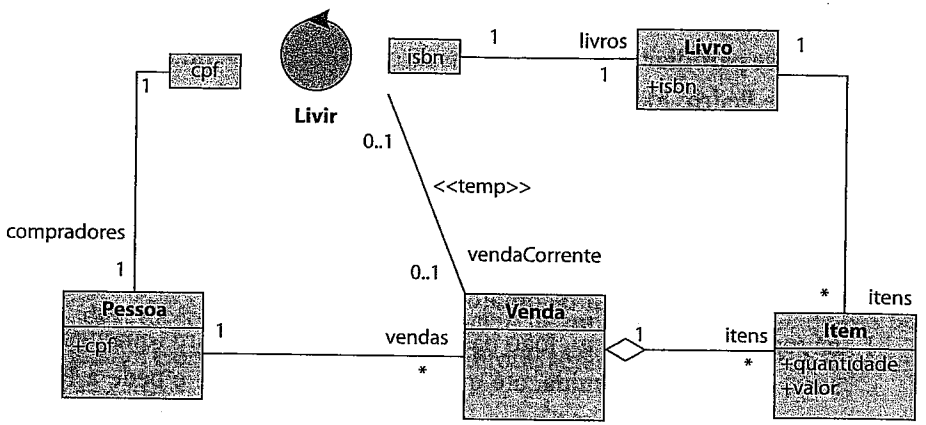


Figura 9.26: Modelo conceitual de referência.

Com a abordagem de concentração, o algoritmo será implementado na classe Livir (aliás, seguindo essa abordagem, *todos* os algoritmos serão implementados nessa classe). Livir tentará obter todos os dados necessários para realizar o cálculo e retornará o resultado. O pseudocódigo poderá ser descrito assim:

```
CLASSE Livir
VAR compradores : MAP[String->Pessoa]
VAR livros : MAP[String->Livro]
VAR vendaCorrente : Venda
MÉTODO getTotalVendaCorrente() : Moeda
 VAR itens : SET[Item]
 VAR total : Moeda
 itens := vendaCorrente.getItens()
 total := 0
 PARA CADA item EM itens FAÇA
    total := total + (item.getValor() * item.getQuantidade())
 FIM PARA
 RETORNA total
```



```
 FIM METODO
FIM CLASSE
```

O que se pode observar com essa abordagem é que:

a) ela resolve o problema e está correta;
b) ela não produz nenhum elemento de software reusável, a não ser a consulta de sistema em si.

A abordagem de delegação pode resolver o problema e, ao mesmo tempo, criar estruturas reusáveis e menos acopladas do que no caso anterior.

Em primeiro lugar, deve-se impedir que a classe Livir tenha acesso a quaisquer instâncias de Item, uma vez que não existe conexão entre essas classes no modelo conceitual. Para fazer isso, Livir vai ter de delegar para a classe Venda o cálculo do resultado. Como se está tratando de consultas e o valor de retorno é escalar, isso equivale a criar um atributo derivado na classe Venda com a seguinte definição OCL:

```
Context Venda::valorTotal:Moeda
  derive:
    itens→sum(quantidade*valor)
```

Além da criação de um atributo derivado na venda, que permite calcular o seu valor total independentemente da operação de sistema que se ocupa disso, poderia ser criado um atributo derivado na classe Item definindo o subtotal, conforme a seguir:

```
Context Item::subtotal:Moeda
  derive:
    quantidade*valor
```

Assim, o atributo valorTotal de Venda passa a ser definido como:

```
Context Venda::valorTotal:Moeda
  derive:
    itens→sum(subtotal)
```

Agora, o pseudocódigo poderia ser definido com responsabilidades distribuídas entre as classes:

```
CLASSE Livir
VAR compradores : MAP[String->Pessoa]
VAR livros : MAP[String->Livro]
```

```
VAR vendaCorrente : Venda
MÉTODO getTotalVendaCorrente():Moeda
 RETORNA vendaCorrente.getValorTotal()
FIM METODO
FIM CLASSE

CLASSE Venda
VAR itens : SET[Item]
MÉTODO getValorTotal() : Moeda
 VAR total : Moeda
 total := 0
 PARA CADA item EM itens FAÇA
       total := total + item.getSubtotal()
 FIM PARA
 RETORNA total
FIM MÉTODO
FIM CLASSE

CLASSE Item
VAR quantidade : Número
VAR valor : Moeda
MÉTODO getSubtotal () : Moeda
 RETORNA quantidade * valor
FIM MÉTODO
FIM CLASSE
```

Agora, há duas estruturas altamente reusáveis: o subtotal da classe Item e o total da classe Venda, que não existiriam com a estratégia concentradora.

No caso de operações de sistema, o padrão de acoplamento baixo também se realiza quando o projetista opta por delegar em vez de retornar o objeto. Considere o modelo conceitual parcial da Figura 9.27a. O diagrama de comunicação da Figura 9.27b mostra a execução da operação mudaData(umaData) da forma concentradora. Já a Figura 9.27c mostra a mesma operação sendo

realizada com delegação, evitando assim o acoplamento desnecessário entre as classes Livir e Venda.

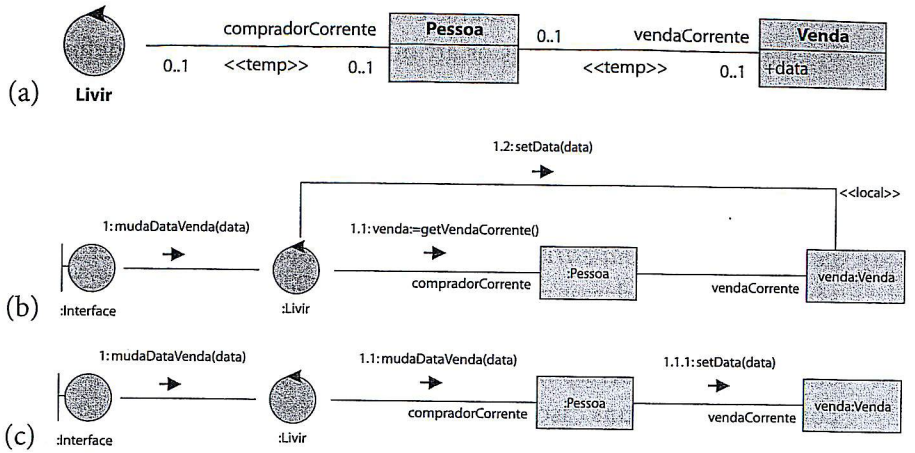


Figura 9.27: (a) Fragmento de modelo conceitual de referência. (b) Estilo de projeto concentrador. (c) Estilo de projeto com delegação.

Pode-se observar claramente, nos diagramas da Figura 9.27, que o estilo concentrador aumenta o número de conexões entre os objetos e, consequentemente, a complexidade do projeto.

9.6. Diagrama de Classes de Projeto

Um dos objetivos do projeto lógico é a construção de um *diagrama de classes de projeto* (*DCP*), que é criado a partir do modelo conceitual e de informações obtidas durante a modelagem dinâmica (construção dos diagramas de comunicação para os contratos).

A primeira versão do DCP constitui uma cópia exata do modelo conceitual. Em seguida, ele vai sendo modificado. As modificações básicas a serem feitas no DCP durante o projeto da camada de domínio são:

a) *adição dos métodos*. Na atividade de análise, apenas as operações e consultas *de sistema* foram determinadas e adicionadas na classe controladora. Na atividade de projeto os métodos delegados encontrados serão adicionados nas das demais classes;

b) *adição da direção das associações*. Na atividade de análise, as associações do modelo conceitual eram não direcionais. Na atividade de projeto

será determinada a direção de navegação das associações em função da direção das mensagens nos diagramas de comunicação;

c) *possível detalhamento dos atributos e associações.* É possível que, na atividade de análise, nem todos os atributos tenham seus tipos definidos. Nesse caso, esses elementos poderão ser adicionados na atividade de projeto, na medida do necessário. Além disso, os tipos abstratos de dados definidos nos papéis de associações poderão ser substituídos por tipos concretos (por exemplo, trocar *lista* por *array* ou *lista encadeada*);

d) *possível alteração na estrutura das classes e associações.* Pode ser necessário criar novas classes para implementar certas estruturas de projeto, como estratégias, por exemplo. Assim, é possível que a estrutura de classes do DCP não corresponda exatamente à mesma estrutura do modelo conceitual em alguns casos;

e) *possível criação de atributos privados ou protegidos.* No modelo conceitual, todos os atributos devem ser públicos porque ali se está representando a informação disponível. Assim, não faz sentido que seja modelada alguma informação que não possa ser acessível fora da classe. Porém, quando se inicia a descrição dos aspectos dinâmicos e de representação interna dos objetos, pode ser necessário trabalhar com atributos privados ou protegidos para encapsular estados internos que determinarão o funcionamento de alguns métodos.

A rigor, o modelo conceitual poderá ser modificado durante a atividade de projeto, mas apenas quando se identificar, nessa atividade, que a modelagem original precisa ser corrigida. Caso contrário, essas modificações são feitas sobre o DCP.

Algumas informações aprendidas durante a construção dos diagramas de comunicação são imediatamente passadas ao DCP. Essas informações são de dois tipos:

a) *métodos delegados*: sempre que um objeto receber uma mensagem delegada, a classe correspondente ao objeto deve registrar a implementação desse método (Figura 9.28);

b) *direção das associações*: a direção das associações no DCP corresponderá à direção do envio das mensagens sobre as ligações de visibilidade baseadas em associações.

Figura 9.28: (a) Uma instância de Venda recebendo uma mensagem delegada. (b) Consequência: a classe Venda deve implementar um método para responder a essa mensagem.

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. Basicamente, cada classe tem predefinidas as seguintes operações:

a) uma operação de criação de instâncias: *create*;

b) uma operação de destruição de instâncias: *destroy*;

c) para cada atributo:
   - uma operação de atualização: *set*;
   - uma consulta: *get*;

d) para cada associação:
   - uma operação de adição: *add*;
   - uma operação de remoção: *remove*;
   - uma consulta: *get*.

Assim, por exemplo, uma classe com seis associações e 15 atributos teria, só de operações e consultas básicas:

a) uma operação de criação de instâncias;

b) uma operação de destruição de instâncias;

c) seis operações de adição de associação;

d) seis operações de remoção de associação;

e) seis operações de consulta de associação;

f) quinze operações de atualização de atributo;

g) quinze operações de consulta de atributo.

Assim, só de operações e consultas básicas essa classe teria 50 métodos declarados! É mais simples assumir que cada classe implementará as operações e consultas citadas para os atributos e associações por padrão e declarar no diagrama de classe apenas os métodos que não podem ser deduzidos pela existência desses elementos, ou seja, os *métodos delegados*.

Deve-se também adicionar ao DCP a *direção* das associações à medida que se verificar a necessidade de um objeto enviar mensagem a outro. As-

sim, as associações sempre terão a direção do envio das mensagens entre as instâncias das respectivas classes. Quando as mensagens trafegarem nas duas direções (mesmo que em diagramas de comunicação distintos), as associações serão bidirecionais (Figura 9.29b).

As Figuras 9.29 e 9.30 mostram como a direção das mensagens pode determinar a direção de navegação das associações no DCP.

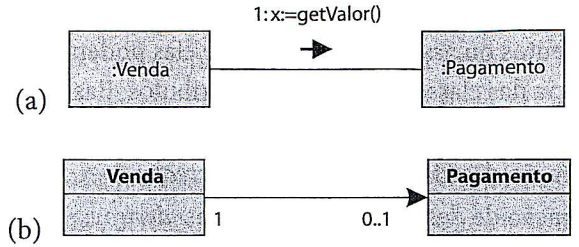


Figura 9.29: (a) Se apenas instâncias de Venda enviam mensagens a instâncias de Pagamento, então a associação entre Venda e Pagamento é unidirecional (b).

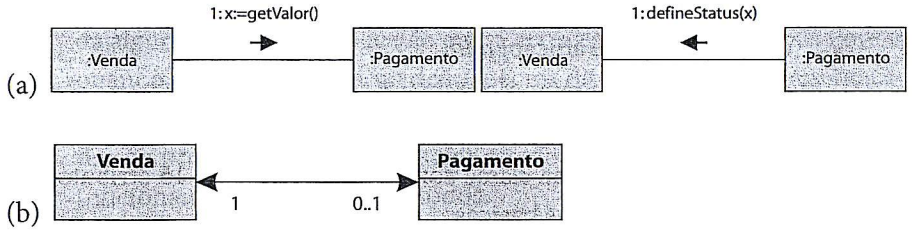


Figura 9.30: (a) Se mensagens são enviadas nas duas direções, então (b) a associação deve ser bidirecional.

A atividade de projeto lógico termina quando o DCP tem informações suficientes para implementar as classes da camada de domínio, isto é, as classes que realizam toda a lógica de transformação e apresentação de dados do sistema. Isso normalmente acontece quando todos os contratos de operação e consulta de sistema foram examinados e seus modelos dinâmicos incorporados ao projeto na forma de diagramas ou algoritmos.

Restam, ainda, os projetos tecnológicos, dentre os quais interface e persistência, que serão tratados respectivamente nos Capítulos 10 e 11, e a fase de construção, incluindo a programação ou geração de código e os testes, a serem tratados no Capítulo 12. Em especial no Capítulo 12, essas estruturas de projeto (DCP e diagramas de comunicação) serão retomadas para mostrar as regras de transformação delas em estruturas de programação.

Capítulo

10

Projeto da Camada de Interface (Web)

O projeto da camada de interface de um sistema depende de alguns artefatos da análise, especificamente dos casos de uso expandidos ou diagramas de sequência de sistema. Será necessário construir um projeto de interfaces que permitam que as operações especificadas possam ser executadas por um usuário que estiver seguindo os fluxos.

Também se deve ter em mente, ao fazer esse projeto, os requisitos não funcionais e suplementares de interface que eventualmente tenham sido levantados.

Há vários tipos de interface: Web, baseada em janelas, baseada em texto, realidade virtual etc. Tendo em vista que muitos sistemas de informação são baseados em interfaces Web, este capítulo dará ênfase à apresentação de uma linguagem de modelagem para esse tipo de interface. Essa linguagem é conhecida como WebML, e consiste em uma extensão UML para modelagem de interfaces Web (Ceri *et al.*, 2003).

O capítulo tem como objetivo apenas apresentar os conceitos fundamentais da WebML mostrando seu potencial de modelagem. Mais informações podem ser encontradas no site oficial www.webml.org. Sugere-se também o uso da ferramenta *WebRatio* (disponível gratuitamente no site para