8 Modelagem Funcional com Contratos

Até o início da modelagem funcional, o processo de análise, na fase de elaboração, deve ter produzido dois artefatos importantes:

a) o *modelo conceitual refinado* (Capítulos 6 e 7), que representa estaticamente a informação a ser gerenciada pelo sistema;

b) os *diagramas de sequência de sistema* (Capítulo 5), que mostram como possíveis usuários trocam informações com o sistema, sem mostrar, porém, como a informação é processada internamente.

Na fase de construção dos diagramas de sequência de sistema devem ter sido identificadas as *operações* e *consultas* de sistema. Cada operação ou consulta desse tipo implica a existência de uma *intenção* por parte do usuário. Essa intenção é capturada pelos contratos de operação de sistema e pelos contratos de consulta de sistema, que correspondem à *modelagem funcional* do sistema.

Um contrato de operação de sistema pode ter três seções:

a) precondições (opcional);

b) pós-condições (obrigatória);

c) exceções (opcional).

Já um contrato para uma consulta de sistema pode ter as seguintes seções:

a) precondições (opcional);

b) resultados (obrigatória);

c) exceções (opcional).

As *precondições* existem nos dois tipos de contratos e devem ser cuidadosamente estabelecidas. Elas complementam o modelo conceitual no sentido de definir o que será verdadeiro na estrutura da informação do sistema quando a operação ou consulta for executada. Isso significa que elas não serão testadas durante a execução, mas algum mecanismo externo deverá garantir sua validade antes de habilitar a execução da operação ou consulta de sistema correspondente.

As *pós-condições* só existem nos contratos de operação de sistema, mas também devem ser muito precisas. Elas estabelecem o que uma operação de sistema muda na informação.

Deve-se tomar cuidado para não confundir as pós-condições com os *resultados* das consultas. As pós-condições só existem nas operações de sistema porque elas especificam alguma *alteração nos dados armazenados*. Assim, pelo princípio de separação entre operação e consulta, não é apropriado que uma operação de sistema retorne algum resultado (exceto em alguns casos consagrados pelo uso). Já as consultas, por definição, devem retornar algum resultado, mas não podem alterar os dados armazenados. Daí os contratos das consultas de sistema terem resultados mas não pós-condições.

Ao contrário das precondições, que devem ser garantidamente verdadeiras durante a execução de uma operação, as *exceções* são situações que usualmente não podem ser garantidas *a priori,* mas serão testadas *durante* a execução da operação. Exceções são eventos que, se ocorrerem, impedem o prosseguimento correto da operação.

O objetivo das exceções em contratos de operação e consulta são distintos. Nos contratos de operação de sistema, as exceções são usadas em situações nas quais se tenta alterar alguma informação com dados que não satisfazem alguma regra de negócio (por exemplo, tentar cadastrar um comprador que já tem cadastro).

Já nas consultas de sistemas as exceções terão uma aplicação bem diferente. Assume-se que como uma consulta não altera dados, então nenhum parâmetro será inválido no sentido de que a consulta possa ou não possa ser feita. Se for consultado, por exemplo, os dados de um CPF que não consta na base, a consulta vai retornar um conjunto vazio de dados, o que é a resposta correta neste caso. Então isso não impede que ela seja executada. Porém, existem situações em que mesmo que uma consulta *possa* ser executada o projetista pode não *querer* que ela seja executada. Por exemplo, se uma consulta na base de livros vai retornar 100 mil livros que deveriam ser listados na tela, essa consulta levaria tempo demais. Então poderia-se adicionar uma exceção a essa consulta estabelecendo que, por exemplo, se o número de livros a ser listado for maior do que 100, a consulta não deve ser executada e uma exceção sinalizada. Não se trata, portanto, de regra de negócio, mas de conveniência de design.

Normalmente, em um contrato, seja de operação ou de consulta, uma exceção pode ser transformada em pré-condição e vice versa. A decisão sobre testar uma condição como precondição ou exceção vai determinar que é o responsável por testar essa condição: a operação que a chama ou a operação chamada.

8.1 Precondições

As precondições estabelecem o que é verdadeiro quando uma operação ou consulta de sistema for executada. Por exemplo, considerando o modelo conceitual de referência da Figura 8.1, se um usuário estiver comprando um livro, poderá ser assumido como precondição que o seu CPF, passado como parâmetro para a operação, corresponde a um comprador válido, ou seja, existe uma instância de Comprador cujo atributo cpf é igual ao CPF passado como parâmetro. Essa expressão pode ser assim representada em OCL:

```
Context Livir::identificarComprador(umCpf:CPF)

  pre:
    comprador→select(cpf=umCpf)→notEmpty()
```

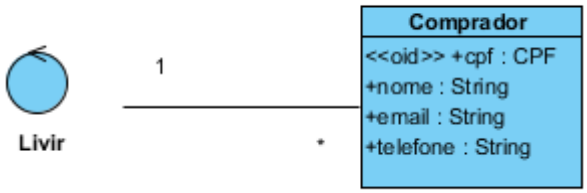

Figura 8.1: Modelo conceitual de referência. FALTOU SALDO NO COMPRADOR

A expressão então afirma que, no contexto do método identificarComprador, que é uma operação de sistema implementada na controladora Livir, há uma precondição que estabelece que o conjunto de compradores filtrado (select) pela condição de que

o atributo cpf do comprador seja igual ao parâmetro umCpf é não vazio, ou seja, há pelo menos um comprador cujo CPF é igual ao parâmetro passado.

Se a associação da Figura 8.1 for qualificada como na Figura 8.2, a precondição de garantia de parâmetro mencionada anteriormente poderá ser escrita de forma mais direta:

```
Context Livir::identificarComprador(umCpf:CPF)

  pre:
    comprador[umCpf]→notEmpty()
```

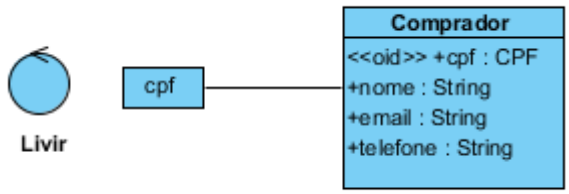

Figura 8.2: Modelo conceitual de referência com associação qualificada. FALTOU SALDO NO COMPRADOR

Quando a associação é qualificada, como na Figura 8.2, pode-se usar um valor para indexar diretamente o mapeamento representado pela associação. Assim, como a operação identificarComprador tem um parâmetro umCpf, a expressão comprador[umCpf] produz o mesmo resultado que a expressão comprador→select(cpf=umCpf).

Para serem úteis ao processo de desenvolvimento de software, as precondições não podem ser expressas de maneira descuidada. Elas devem refletir fatos que possam ser identificados diretamente no modelo conceitual já desenvolvido para o sistema. Isso justifica a utilização de linguagens formais como OCL (Warmer & Kleppe, 1998) para escrever contratos.

Pode-se identificar duas grandes famílias de precondições:

a) *garantia de parâmetros*: precondições que garantem que os *parâmetros* da operação ou consulta correspondem a elementos válidos do sistema de informação, como, por exemplo, que existe cadastro para o comprador cujo CPF corresponde ao parâmetro da operação ou consulta;

b) *restrição complementar*: precondições que restringem ainda mais o modelo conceitual para a execução da operação ou consulta, de forma a garantir que a informação se encontra em uma determinada situação desejada, por exemplo, que o endereço para entrega informado pelo comprador não esteja no estado *inválido*.

Sobre o segundo tipo de precondição, pode-se entender que ela pode estabelecer restrições mais fortes sobre o modelo conceitual. Assim, se o modelo conceitual especifica que uma associação tem multiplicidade de papel 0..1, uma precondição complementar poderá especificar que durante a execução de determinada operação de

sistema esse papel está preenchido (1) ou não (0). Por exemplo, um Pedido pode ter ou não um Pagamento (0..1), mas a operação de efetuar o pagamento de um pedido exige que o pedido que não esteja pago ainda (0).

É necessário lembrar que uma precondição nunca poderá *contradizer* as especificações do modelo conceitual, mas apenas restringi-las ainda mais. Se o modelo conceitual exige 0 ou 1, nenhuma precondição poderá garantir 2.

8.1.1 Garantia de Parâmetros

Existe uma forma sistemática para saber se uma operação deve ter uma pré-condição (ou exceção) de garantia de parâmetros. Essa técnica tem relação com o teste funcional da operação, como será visto mais adiante. Deve-se olhar cada parâmetro da operação e perguntar se existem classes de valores válidos e inválidos dentro do tipo de dados da variável.

Por exemplo, se o parâmetro é tipado com ISBN e a operação é inserirLivro(umIsbn:ISBN, ...), então consideradas as regras de negócio, existe uma classe de valores válidos (isbns que ainda não são cadastrados), e uma classe de valores inválidos (isbns já cadastrados). Por outro lado, se o tipo do parâmetro for String e a operação inserirLivro(umIsbn:String,...), então para esta operação podem ser identificadas duas classes inválidas: isbns de livros já cadastrados e isbns mal formados.

Assim, considerado o tipo do parâmetro e identificadas as classes de valores inválidos, a regra para definir pré-condições de garantia de parâmetros é a seguinte:

> *Para cada classe de valores inválidos de um parâmetro de uma operação de sistema deve ser definida uma pré-condição (ou uma exceção).*

Em relação às precondições de *garantia de parâmetros*, deve-se tomar cuidado para não confundir as precondições que testam os parâmetros *semanticamente* com as simples verificações sintáticas. Para garantir que um parâmetro seja, por exemplo, um número maior do que zero, basta usar *tipagem* (por exemplo, "x:InteiroPositivo"), não sendo necessário escrever isso como precondição. Outro exemplo é ISBN, que tem uma regra sintática de formação. NumeroPrimo poderia ser um tipo? A resposta é sim, porque para saber se um número é primo não é necessário consultar quaisquer dados armazenados.

A *tipagem* deve ser definida na assinatura da operação. Por exemplo, a tipagem é que vai definir que um determinado parâmetro deve ser um número inteiro ou um número maior do que 100, ou mesmo um número primo. Se o tipo não existir, deve-se definir uma classe com o estereótipo <<primitive>> para o novo tipo.

Será considerada precondição semântica apenas uma asserção para a qual a determinação do valor verdade implica verificar os dados gerenciados pelo sistema. Assim, determinar se um número de CPF está bem formado pode ser feito sintaticamente (aplicando-se uma fórmula para calcular os dígitos verificadores), mas verificar se existe um comprador cadastrado com um dado número de CPF é uma

verificação semântica, pois exige a consulta aos dados de compradores. Assim, a primeira verificação deve ser feita por tipagem, e a segunda por precondição.

8.1.2 Restrição Complementar

Uma *restrição complementar* consiste na garantia de que certas restrições mais fortes do que aquelas estabelecidas pelo modelo conceitual são obtidas.

As restrições complementares são mais comuns em designs que utilizam a técnica *statefull* nos diagramas de sequencia de sistema. Isso porque neste caso, cada operação vai estar esperando por informações deixadas por outras operações, como, por exemplo, a identificação do cliente, que é passada por uma operação vai ser usada por outras. Então essas outras operações terão estabelecido como precondição que o cliente foi devidamente identificado (possivelmente por uma associação temporária).

É possível identificar vários tipos de restrições complementares, como por exemplo:

a) fazer uma *afirmação específica* sobre uma instância ou um conjunto de instâncias;

b) fazer uma *afirmação existencial* sobre um conjunto de instâncias;

c) fazer uma *afirmação universal* sobre um conjunto de instâncias.

Um exemplo de *afirmação específica* sobre uma instância, considerando o modelo da Figura 8.2 poderia ser afirmar que o comprador com o CPF 12345678910 tem saldo igual a zero:

```
Context Livir::operacaoQualquer()
  pre:
    comprador[12345678910].saldo = 0
```

Um exemplo de *afirmação existencial* seria dizer que existe pelo menos um comprador com saldo igual a zero (embora não se saiba necessariamente qual):

```
Context Livir::operacaoQualquer()
  pre:
    comprador→exists(c|c.saldo = 0)
```

Um exemplo de *afirmação universal* seria dizer que todos os compradores têm saldo igual a zero:

```
Context Livir::operacaoQualquer()
  pre:
    comprador→forAll(c|c.saldo = 0)
```

Tanto a expressão exists quanto forAll usadas poderiam ser simplificadas para exists(saldo=0) ou forAll(saldo=0), mantendo o mesmo significado.

8.1.3 Garantia das Precondições

Como as precondições não são testadas pela operação admite-se que algum mecanismo externo as garanta. Elas precisam ser testadas não pela operação que as declara, mas pela operação que vai chamar a operação que as declara.

Essa garantia pode ser feita pela chamada explícita de uma consulta que verifique se a condição é verdadeira, antes de chamar a operação ou ainda por mecanismos de interface que impedem que a operação seja chamada com dados inválidos.

Por exemplo, em vez de digitar um CPF qualquer, o usuário terá de selecioná-lo de uma lista de CPFs válidos. Dessa forma, o parâmetro já estará garantidamente validado antes de executar a operação.

8.1.4 Precondição × Invariante

Usam-se invariantes no modelo conceitual para regras que valem sempre, independentemente de qualquer operação. Usam-se precondições para regras que valem apenas quando determinada operação ou consulta está sendo executada.

Quando já existir uma invariante para determinada situação não é necessário escrever precondições para a mesma situação. Por exemplo, se já existir uma invariante na classe Aluno afirmando que ele só pode se matricular em disciplinas do seu curso, não é necessário escrever precondições nas operações de matrícula para verificar isso. Assume-se que o design deva ser efetuado de forma que tanto a invariante quanto as eventuais precondições nunca sejam desrespeitadas.

Mecanismos de teste de design poderão verificar as invariantes e precondições durante a fase de teste do sistema. Caso, em algum momento, as condições sejam falsas, devem ser sinalizadas exceções. Porém, nesses casos, o projetista deve imediatamente corrigir o sistema para que tais erros não venham mais a acontecer. Quando o sistema for entregue ao usuário final deve-se ter garantias de que as precondições e invariantes nunca sejam desrespeitadas.

8.2 Associações Temporárias

Quando se utiliza a estratégia *statefull*, mencionada no Capítulo 6, é necessário que a controladora guarde "em memória" certas informações que não são persistentes, mas que devem ser mantidas durante a execução de um conjunto de operações.

Pode-se, então, definir certas associações ou atributos temporários (ambos estereotipados com <<temp>>) para indicar informações que só são mantidas durante a execução de um determinado caso de uso e descartadas depois.

Por exemplo, para que a controladora guarde a informação sobre quem é o comprador correntemente sendo atendido, pode-se utilizar uma associação temporária para indicar isso no modelo conceitual refinado, como na Figura 8.3.

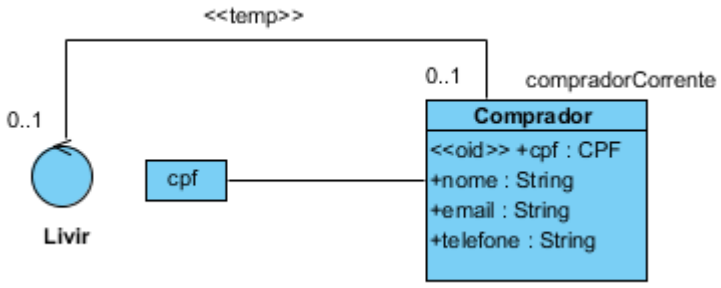

Figura 8.3: Uma associação temporária.

Assim, uma precondição de uma operação, por exemplo, poderia afirmar que já existe um comprador corrente identificado da seguinte forma:

```
Context Livir::operacaoQualquer()
  pre:
    compradorCorrente→notEmpty()
```

8.3 Retorno de Consulta

Conforme mencionado, operações de sistema provocam alterações nos dados, enquanto consultas apenas retornam dados. Os contratos de consultas devem ter obrigatoriamente uma cláusula de retorno, representada em OCL pela expressão body.

As expressões que representam precondições vistas até aqui são todas booleanas, mas expressões utilizadas na cláusula body podem ser de qualquer tipo. Podem retornar *strings*, números, listas, tuplas etc. Os exemplos seguintes são baseados no modelo conceitual da Figura 8.3.

Inicialmente define-se uma consulta de sistema que retorna o *saldo* do *comprador corrente*:

```
Context Livir::saldoCompradorCorrente():Moeda
   body:
     compradorCorrente.saldo
```

As consultas de sistema sempre têm por contexto a controladora. Portanto, nessa expressão, compradorCorrente é uma propriedade da controladora; no caso, um papel de associação.

A consulta a seguir retorna nome e telefone do comprador cujo CPF é dado:

```
Context Livir::nomeTelefoneComprador(umCpf):Tuple
   body:
     Tuple{
       nome = comprador[umCpf].nome,
       telefone = comprador[umCpf].telefone
     }
```

O construtor Tuple é uma das formas de representar *DTOs*[23] em OCL; a tupla funciona como um registro, no caso com dois campos: nome e telefone. Os valores dos campos são dados pelas expressões após o sinal "=".

Para não ter de repetir a expressão comprador[umCpf] ou possivelmente expressões até mais complexas do que essa em contratos OCL, pode-se usar a cláusula def para

[23] DTO, ou Data Transfer Object, é um padrão de design que indica que objetos passados pela controladora para a interface e vice versa não devem ser semânticos, ou seja, devem ser objetos que apenas possuem atributos com seus respectivos *getters* e *setters*, nada mais. O *type Record* em Pascal e a Tupla OCL são exemplos de DTOs.

definir um identificador para a expressão que pode ser reutilizado. Usando a cláusula def, o contrato ficaria assim:

```
Context Livir::nomeTelefoneComprador(cpfComprador):Tuple
  def:
    c = comprador[cpfComprador]
   body:
     Tuple{
       nome = c.nome,
       telefone = c.telefone
     }
```

A expressão dentro da clausula def: indica que o simbolo c passa a referenciar o comprador com o CPF passado como parâmetro.

A expressão a seguir faz uma *projeção*, retornando um conjunto com todos os nomes de compradores:

```
Context Livir::listarNomeCompradores():Set
  body:
    comprador.nome
```

A próxima expressão aplica um *filtro* e uma projeção, retornando os nomes de todos os compradores que têm saldo igual a zero:

```
Context Livir::listarNomeCompradoresSaldoZero():Set
  body:
    comprador→select(saldo=0).nome
```

Como último exemplo, a expressão a seguir retorna CPF, nome e telefone de todos os compradores que têm saldo igual a zero:

```
Context Livir::listarCpfNomeTelefoneCompradoresSaldoZero():Set
  body:
    comprador→select(saldo=0)→collect(c|
        Tuple {
          cpf = c.cpf,
          nome = c.nome,
          telefone = c.telefone
        }
    )
```

A expressão collect é uma forma de obter um conjunto cujos elementos são propriedades ou transformações de outro conjunto. A própria notação "." aplicada

sobre conjuntos é uma forma abreviada de collect. Por exemplo, comprador.nome é equivalente a comprador→collect(nome).

Quando for possível, usa-se a notação ".", por ser mais simples. Mas, no exemplo anterior, a necessidade de criar uma tupla em vez de acessar uma propriedade dos elementos do conjunto impede o uso da notação ".". Assim, a expressão collect tem de ser explicitamente usada nesse caso.

8.4 Pós-condições

As pós-condições estabelecem o que muda nas informações armazenadas no sistema após a execução de uma operação de sistema. As pós-condições também devem ser claramente especificadas em termos que possam ter correspondência nas definições do modelo conceitual. Assim, uma equivalência com as expressões usadas como pós-condição e expressões passíveis de escrita em OCL é altamente desejável para evitar que os contratos sejam ambíguos ou incompreensíveis.

Uma pós-condição em OCL é escrita no contexto de uma operação (de sistema) com o uso da cláusula "post", conforme exemplo a seguir:

```
Context Livir::operacaoX()
  post:
    <expressão OCL>
```

Havendo mais de uma pós-condição que deve ser verdadeira após a execução da operação de sistema, faz-se a combinação das expressões com o operador and[24]:

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

Para se proceder a uma classificação dos tipos de pós-condições possíveis e úteis em contratos de operação de sistema deve-se considerar que o modelo conceitual possui apenas três elementos básicos, que são os *conceitos* (representados pelas classes), as *associações* e os *atributos*. Assim, considerando que instâncias de classes e associações podem ser criadas ou destruídas, e que atributos apenas podem ter seus valores alterados, chega-se a uma classificação com cinco tipos de pós-condições:

a) modificação de valor de atributo;

b) criação de instância;

c) adição de ligação baseada em associação;

d) destruição de instância;

---

[24] O operador and também pode ser usado em quaisquer outras expressões OCL, como pré-condições, invariantes, etc.

e) remoção de ligação.

Para que essas pós-condições possam ser definidas de forma não ambígua é necessário que inicialmente se proceda a uma definição de certas *operações básicas* sobre essas estruturas conceituais e seu comportamento esperado.

As operações consideradas básicas são aquelas que em orientação a objetos operam diretamente sobre os elementos básicos do modelo conceitual. Seu significado e comportamento são definidos por padrão.

Infelizmente, as linguagens de programação não oferecem ainda um tratamento padronizado para as operações básicas. Sua programação muitas vezes é fonte de trabalho braçal para programadores.

As operações, conforme definidas nas subseções seguintes, são efetivamente básicas, no sentido de que não fazem certas verificações de consistência. Por exemplo, uma operação que adiciona uma ligação não vai verificar se o limite máximo de ligações possíveis já foi atingido. Essas verificações de consistência devem ser feitas em relação ao contrato como um todo, ou seja, após avaliar todas as pós-condições é que se vai verificar se os objetos ficaram ou não em um estado consistente.

Por exemplo, suponha que um objeto *A* tenha uma ligação obrigatória com um objeto *B1*, e uma operação de sistema vai trocar essa ligação por outra entre *A* e *B2*. É necessário destruir a ligação original e criar uma nova. No intervalo de tempo entre essas duas operações, o objeto *A* estaria inconsistente (sem a ligação obrigatória), mas, considerando o conjunto das pós-condições do contrato, observa-se que o resultado final é consistente, pois uma foi destruída e outra criada em seu lugar.

A discussão sobre a manutenção de consistência do conjunto de pós-condições de uma operação de sistema será feita na Seção 8.4.6.

8.4.1 Modificação de Valor de Atributo

Um dos tipos de pós-condição consiste em indicar que o valor de um atributo foi alterado. Usualmente essa pós-condição é declarada assim em OCL:

```
objeto.atributo=valor
```

Mas como OCL é uma linguagem declarativa essa expressão tem, na verdade três maneiras diferentes de ser tornada verdadeira:

a) O atributo do objeto pode ser alterado para ficar igual ao valor (o que, normalmente é a interpretação default).
b) O valor pode ser alterado para ficar igual ao atributo do objeto.
c) Tanto o valor quanto o atributo podem ser alterados para ficarem iguais a um terceiro valor.

Isso se constitui, portanto, em um problema de ambiguidade na realização da expressão. Cabot (2007) propõe que se utilize uma semântica default para interpretar tais expressões de forma que a ambiguidade seja reduzida. Porém, acreditamos ser mais prático considerar outra abordagem, com o uso de operações básicas e o predicado "^" da OCL, como será visto na sequencia.

Pode-se indicar que um atributo foi alterado, sem ambiguidade, afirmado-se que um mensagem básica predefinda foi enviada ao objeto, promovendo a alteração. Essa

mensagem, por padrão é denotada pelo prefixo "set" seguido do nome do atributo. O novo valor é passado como parâmetro.

A mensagem setAtributo é enviada a uma instância da classe que contém o atributo. Considere o modelo da Figura 8.4 e considere que c é uma instância de Comprador.

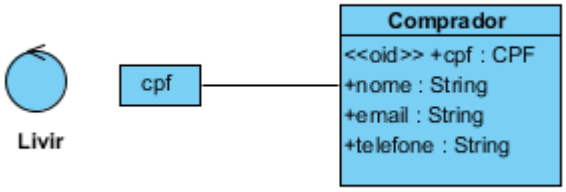

Figura 8.4: Modelo conceitual de referência. mutiplicidade

Neste caso, se uma operação de sistema qualquer vai alterar o atrbuto email de c para um valor novoEmail, possivelmente passado como parâmetro. Isso pode ser expresso na forma usual como:

```
c.email = novoEmail
```

Mas já vimos que essa forma pode ser interpretada de várias maneiras. Então a expressão abaixo expressa a mesma intenção mas sem tal ambiguidade:

```
c^setEmail(novoEmail)
```

A notação "^" usada aqui difere da notação "." no seguinte aspecto: o *ponto* forma uma expressão cujo valor é o retorno da avaliação da expressão, ou *null*, se não houver retorno; já o *circunflexo* apenas indica que a mensagem *foi enviada ao objeto*. O valor de uma expressão com circunflexo, portanto, só pode ser booleano.

O circunflexo é um *predicado*. Ele na verdade substitui o sinal de igual na expressão original. A expressão c.email = novoEmail indica que dois valores são iguais, mas a expressão c^setEmail(novoEmail) indica que um objeto recebeu uma mensagem. Se estiver claro que o efeito dessa mensagem é alterar o valor do atributo, então realizou-se a intenção.

Outra coisa: a expressão c^setEmail(novoEmail) só diz que a instância de pessoa *recebeu* a mensagem, mas não diz quem *enviou*. A decisão sobre qual objeto vai enviar a mensagem é tomada na atividade de design, durante a modelagem dinâmica (Capítulo 9).

8.4.2 Criação de Instância

A operação de criação de instância deve simplesmente criar uma nova instância de uma classe. Embora a OCL não seja uma linguagem imperativa, ela possui um construtor para referenciar que uma nova instância de uma classe dada foi criada. Esta operação, na verdade é uma propriedade que de um objeto que é referenciada assim:

```
objeto.oclIsNew()
```

Seu significado, quando consta em uma pós condição é de que "objeto" foi criado na operação que contém a pós-condição. Essa propriedade pode ser usada em conjunto com outra que declara a classe à qual um objeto pertence:

```
objeto.oclIsTypeOf(classe)
```

Porém, como seria um tanto prolixo ficar sempre declarando a criação de instâncias com duas expressões, sugerimos a criação de uma propriedade única que declara que o objeto foi criado e pertence a uma classe. A propriedade newInstanceOf é então definida da seguinte maneira:

```
objeto.newInstanceOf(classe)  é equivalente a:
objeto.oclIsNew() and  objeto.oclIsTypeOf(classe)
```

Assim, uma pós-condição que declara que foi criada uma nova instância de Livro, conforme a Figura 8.5, poderia ser referenciada assim:

```
post:
    livro.newInstanceOf(Livro)
```

Figura 8.5: Uma classe a ser instanciada. Incluit controladora

Neste caso, não se usa o circunflexo porque não se trata de um mensagem enviada a uma instância de livro. Trata-se de declarar que uma instância de livro foi criada pela operação.

A mensagem newInstanceOf apenas declara que a instância foi criada, mas não fala nada de seus atributos e associações, algumas das quais podem ser obrigatórias. Pode-se assumir que atributos com valores iniciais (definidos pela cláusula init) já sejam definidos automaticamente quando a instância for criada. Então não há necessidade de especificar novamente pós-condições para dar valor a esses atributos.

Além disso, atributos derivados são calculados e não podem ser modificados diretamente. Mas, o que acontece com os demais atributos e associações no momento da criação?

A operação básica de criação de instância simplesmente produz a instância, sem inicializar seus atributos e associações obrigatórias. Nesse caso, a instância poderá ficar inconsistente com sua definição a não ser que o contrato especifique outras pos-condições que garantem que todos os atrbutos e associações obrigatórios seam inicializados. A validação de consistência dos objetos deve ser feita apenas ao final da operação de sistema especificada pelo contrato.

Já vimos até aqui pos-condições de criação de instância e de alteração de valor de atributo. Isso já permite iniciar um contrato para uma operação relacionada ao CRUD de livro. A operação de sistema inserirLivro tem seu contrato entao assim definido, inicialmente:

```
Context Livir::inserirLivro(umIsbn, umTitulo, umAutor)
  post:
    novoLivro.newInstanceOf(Livro) and
    novoLivro^setIsbn(umIsbn) and
    novoLivro^setTitulo(umTitulo) and
    novoLivro^setAutor(umAutor)
```

Nota-se que o atributo preco, que tem valor predefinido, não precisa ser inicializado, bem como o atributo derivado status, que é calculado (por uma clausula "derive" na definição da classe Livro).

Há mais um "porém" aqui: em pós-condições de contratos de nada adianta mencionar a criação de uma instância se ela não for também imediatamente ligada a alguma outra instância, porque a informação inacessível em um sistema simplesmente não é informação. Então, a criação de instância vai ocorrer sempre em conjunto com uma adição de ligação, conforme será visto na seção seguinte.

8.4.3 Adição de Ligação

Como visto, outro tipo de operação básica é aquela que indica que uma *ligação* baseada em uma associação foi criada entre duas instâncias. A criação de ligações pode ser limitada superiormente e inferiormente, dependendo da multiplicidade de papel. Por exemplo, uma associação 0..5 que já tenha cinco objetos não poderá aceitar um sexto objeto. Uma associação *para um* não pode aceitar um segundo elemento, nem o primeiro pode ser removido.

Usuamente, uma pos condição OCL para indicar que uma ligação foi adicionada seria feita desta forma:

```
objeto.papel→includes(outroObjeto)
```

Mas novamente, tal expressão pode ter interpretação ambígua quando se trata de geração de código, pois há pelo menos quatro formas de torna-la verdadeira:

a) Incluir outroObjeto no papel de objeto.
b) Atribuir a outroObjeto um objeto que já esteja no papel do objeto.
c) Incluir um terceiro objeto no papel do objeto e atribuir este terceiro objeto a outroObjeto.
d) Atribuir o conjunto vazio a outroObjeto, pois qualquer que seja o papel do objeto ele vai incluir o conjunto vazio.

Usualmente, claro, a interpretação escolhida é a primeira. Mas novamente preferimos uma forma de escrita que não dê margem a interpretações ambíguas e que sejam mais

palatável para analistas e programadores. Usaremos novamente uma operação básica para indicar que uma ligação foi criada.

Existem vários dialetos para nomear operações que modificam e acessam papeis de associações. Aqui será usado o prefixo "add" seguido do nome de papel para nomear essa operação (outra opção seria usar set, como no caso de atributos).

Assim, considerando a associação entre as classes Automovel e Pessoa, conforme a Figura 8.6, e considerando duas instâncias, respectivamente, jipe e joao, pode-se admitir que uma ligação possa ser criada do ponto de vista do automóvel por:

```
jipe^addPassageiro(joao)
```

ou, do ponto de vista da pessoa, por:

```
joao^addAutomovel(jipe)
```

As duas expressões são simétricas e produzem exatamente o mesmo resultado (a criação da ligação entre as instâncias baseada na associação entre as classes).

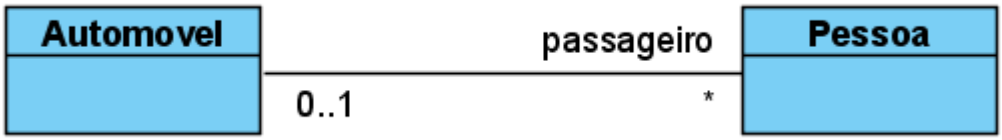

Figura 8.6: Um modelo de referência para operações de adição de ligações.

Associações com papel obrigatório, como na Figura 8.7, normalmente são criadas juntamente com um dos objetos (o do lado não obrigatório). Assim, usualmente esse tipo de pós-condição deve vir combinada, como indicado a seguir:

```
p.newInstanceOf(Pagamento) and
pedido^addPagamento(p)
```

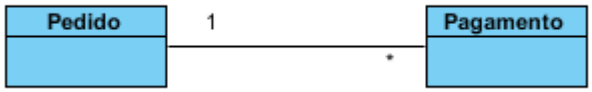

Figura 8.7: Um modelo de referência para operações de criação de associação com papel obrigatório.

A situação se complica mais quando o limite inferior for maior do que 1, o que implica que um objeto já teria de ser criado com vários outros objetos associados. Mas, basta mencionar no contrato a criação do objeto e adicionar ligações até chegar ao limite exigido. A consistência será verificada para o contrato como um todo.

Complementando, então, o exemplo da seção anterior, a expressão a seguir mostra a criação de um novo livro e sua inicialização, inclusive com a criação de uma associação entre a controladora e o novo livro:

```
Context Livir::inserirLivro(umIsbn, umTitulo, umAutor)
  post:
    novoLivro.newInstanceOf(Livro) and
```

```
novoLivro^setIsbn(umIsbn) and
novoLivro^setTitulo(umTitulo) and
novoLivro^setAutor(umAutor) and
self^addLivro(novoLivro)
```

8.4.4 Destruição de Instância

A destruição de objetos deve também ser entendida do ponto de vista declarativo da OCL. Há duas abordagens para indicar que uma instância foi destruída:

a) *explícita*: declara-se que um objeto foi destruído através do envio de uma mensagem explícita de destruição;

b) *implícita*: removem-se todas as ligações para o objeto de forma que ele passe a não ser mais acessível. Em linguagens de programação é possível implementar coletores de lixo (*garbage collection*) para remover da memória objetos que não são mais acessíveis.

Neste livro será assumida a abordagem explícita, visto que ela deixa mais claro qual a real intenção do analista. Um objeto que foi destruído, então, terá recebido uma mensagem básica como:

```
objeto^destroy()
```

O significado dessa expressão em uma pós-condição de operação de sistema é de que o objeto referenciado foi destruído durante a execução da operação.

Assume-se que todas as ligações desse objeto também são removidas com ele. Se algum objeto ligado a ele ficou inconsistente após a remoção da ligação então o contrato deve especificar o que acontece com este objeto.

8.4.5 Remoção de Ligação

A remoção de uma ligação entre dois objetos é referenciada pela operação básica com prefixo "remove" seguida do nome de papel e tendo como parâmetro o objeto cuja ligação deve ser removida (em alguns dialetos poderia ser "unset"). Por exemplo, considerando a Figura 8.7, para remover um pagamento p1 associado ao pedido pedido, pode-se escrever:

```
pedido^removePagamento(p1)
```

Deve-se assumir, neste caso, que, como a multiplicidade de papel de Pagamento para Pedido é obrigatória (igual a 1), a remoção da ligação implicará necessariamente a destruição do pagamento ou a criação posterior de uma nova ligação com outro pagamento, o que deverá ser especificado no mesmo contrato.

Quando a multiplicidade do papel a ser removido é 1 ou 0..1, é opcional informar o parâmetro, pois há uma única possível ligação a ser removida. Observando novamente Figura 8.7, se a remoção da ligação fosse feita a partir do pagamento, a operação poderia ser chamada sem o parâmetro, pois só há um pedido possível a ser removido:

```
p1^removePedido()
```

Novamente, deve-se ter em mente que a remoção dessa ligação obriga à criação de uma nova ligação para o pagamento p1 ou sua destruição no mesmo contrato.

Tentar remover uma ligação inexistente é um erro de design e não pode ser feito em pós-condições bem formadas.

8.4.6 Pós-condições bem Formadas

Considerando-se então que as operações básicas que denotam as pós-condições mais elementares não comportam internamente a checagem de consistência nos objetos em relação ao modelo conceitual, o conjunto de pós-condições é que 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.

Pode-se resumir assim as checagens a serem efetuadas em um contrato:

a) uma instância recém-criada deve ter pós-condições indicando que todos os seus atributos foram inicializados, exceto: (1) atributos derivados (que são calculados), (2) 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) e (3) atributos que possam ser nulos (nesse caso, a inicialização é opcional);

b) 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.

c) todas as associações afetadas por criação ou destruição de instância ou associação devem estar com seus papéis dentro dos limites inferior e superior;

d) todas as invariantes afetadas por alterações em atributos, associações ou instâncias devem continuar sendo verdadeiras.

Foge ao escopo deste livro a definição e um sistema de verificação de restrições, o que seria necessário para implementar automaticamente a checagem de invariantes e limites máximo e mínimo em associações. O analista, ao preparar os contratos, deve estar ciente de que os objetos devem ser deixados em um estado consistente após cada operação de sistema. Havendo a possibilidade de implementar um sistema de checagem automática dessas condições, seria uma grande ajuda à produtividade do analista. Porém, tais sistemas ainda são alvo de pesquisa e suas implementações em ferramentas comerciais são ainda experimentais.

Uma outra questão relacionada a contratos bem formados refere-se aos atributos declarados com valores iniciais. Se um atributo com valor inicial usa em sua definição valores que devem ser obtidos em associações (possivelmente obrigatórias), como o valor inicial será calculado antes que as ligações tenham sido adicionadas? Deve-se considerar então que a inicializacao de atributos com valores iniciais será uma das últimas operações a serem realizadas quando da implementação de um contrato, ou seja, não no momento da criação da instância, mas apenas depois que todas as suas ligações obrigatórias e atributos obrigatórios tenham sido adicionados e definidos.

8.4.7 Combinações de Pós-condições

Cada operação de sistema deve ter um contrato no qual as pós-condições vão estabelecer tudo o que essa operação muda nos objetos, associações e atributos

existentes. Usualmente, uma operação de sistema terá várias pós-condições, que podem ser unidas por operadores and, como mencionado anteriormente. Mas também é possível usar operadores or, que indicam que pelo menos uma das pós-condições ocorreu, mas não necessariamente todas:

```
post:
  <pos-condição 1> or
  <pos-condição 2>
```

Um dos problemas com este tipo de especificação é que ela é inerentemente ambígua em termos de geração de código, e, assim, sua utilização pode ser preterida caso outras interpretações mais determinísticas sejam possíveis.

Outro conector de condições que pode ser usado é o operador implies, com o mesmo significado da implicação lógica. Mas esse operador também pode ser substituído pela forma if-then-endif. Assim, a expressão:

```
post:
  <condição> implies <pos-condição>
```

pode ser escrita como:

```
post:
  if <condição> then
    <pos-condição>
  endIf
```

A vantagem do implies é que ele é mais enxuto. A vantagem do if é que ele ainda tem a forma if-then-else-endif e é mais familiar a programadores.

8.4.8 Valores Anteriores: @pre

Muitas vezes, a poscondição é construída com valores que os atributos tinham antes de a operação ser executada. Esses valores anteriores devem ser anotados com a expressão @pre. Por exemplo, uma pós-condição que estabeleça que *se o saldo do cliente corrente era negativo então o saldo passou a ser 0* poderia ser escrita assim:

```
post:
  if clienteCorrente.saldo@pre < 0 then
    clienteCorrente^setSaldo(0)
  endIf
```

ou:

```
post:
  clienteCorrente.saldo@pre < 0 implies
    clienteCorrente^setSaldo(0)
```

Outra situação em que seria necessário o uso de @pre é quando se quer que um atributo seja modificado de forma relativa ao valor que tinha antes. Por exemplo, se a

operação de sistema deve dobrar o valor total do saldo do cliente corrente, então a pós-condição poderia ser escrita assim:

```
post:
  clienteCorrente^setsaldo(clienteCorrente.saldo@pre*2)
```

O @pre também pode ser aplicado sobre ligações de um papel. Por exemplo, para indicar o conjunto de livros que estava contido em um pedido antes da operação ser executada, pode-se escrever:

```
  pedido.livro@pre
```

Outro exemplo, com uso mais intenso deste operador seria uma expressão que obtém o valor total derivado de um pedido antes da execução da operação. Como tanto as quantidades e preços dos livros podem ter mudado como tambem pode ter mudado o próprio conjunto de livros ligado ao pedido, é necessário usar o operador @pre mais de uma vez:

```
  pedido.livro@pre->sum(preco@pre*quantidade@pre)
```

Se, por outro lado, o analista não tem motivos para acreditar que os valores tenham mudado (pois ele próprio faz o contrato), não necessita usar o @pre.

8.4.9 Pós-condições sobre Coleções de Objetos

É possível com uma única expressão OCL afirmar que todo um conjunto de objetos foi alterado. Por exemplo, para afirmar que o preço de todos os livros foi aumentado em x%, pode-se usar a expressão forAll para indicar que todas as instâncias foram alteradas:

```
Context Livir::aumentaPrecoLivros(x)
    post: self.livro→forAll(li|
          li^setPreco(li.preco@pre * (1-x/100))
    )
```

Se não houver necessidade de relativizar o valor, como no exemplo anterior, isto é, se todos os objetos vão receber um mesmo valor fixo, então é possível abreviar a notação. Por exemplo, se fosse para fixar o preço de todos os livro em x reais, poderia-se escrever:

```
Context Livir::fixaPrecos(x:Moeda)
  post: self.livro→forAll(li|
    li^setPreco(x)
  )
```

Ou, de forma mais abreviada:

```
Context Livir::fixaPrecos(x:Moeda)
  post: livro^setPreco(x)
```

Lembre que livro é um papel da controladora, ou seja, na expressao, livro representa self.livro, que consiste no conjunto de todos os livros.

8.4.10 Poscondições e Eventos do Mundo Real

O processo de criação de contratos está intimamente ligado ao caso de uso expandido e ao modelo conceitual. Deve-se usar o modelo conceitual como referência em todos os momentos porque ele é a fonte de informação sobre quais asserções podem ser feitas.

Os contratos devem ser sempre escritos como expressões interpretáveis em termos dos elementos do modelo conceitual. Assim, asserções como "foi impresso um recibo" dificilmente serão pós-condições de um contrato, visto que não representam informação do modelo conceitual. Tais expressões não podem sequer ser representadas em OCL.

Mesmo que o analista quisesse por algum motivo armazenar a informação de que um recibo foi impresso após a execução de alguma operação, a pós-condição deveria ser escrita de forma a poder ser interpretada como alteração de alguma informação no modelo conceitual, como por exemplo, "o atributo reciboImpresso do emprestimoAberto foi alterado para true" ou, em OCL:

```
post:
  emprestimoAberto^setReciboImpresso(true)
```

8.5 Exceções

As exceções em contratos são estabelecidas como situações de falha que não podem ser evitadas ou verificadas antes de iniciar a execução da operação propriamente dita.

Muitas vezes, situações identificadas como exceções podem ser convertidas em precondições. Sempre que for possível transformar uma exceção em pré-condição é conveniente fazê-lo, pois é preferível limitar a possibilidade de o usuário gerar um erro do que ficar indicando erros em operações que ele já tentou executar e falhou. E isso vale tanto para usuários humanos, quanto para operações que chamam outras operações: quanto antes um erro for localizado melhor.

A OCL não apresenta em sua definição uma clausula específica para indicar exceções em contratos, mas essas podem ser construídas como pós-condições condicionadas, do tipo:

```
post:
  if <condição> then
    <sinaliza exceção>
  else
    <todas as outras pós-condições (inclusive outras exceções)>
```

Como seria muito trabalhoso representar exceções desta forma, especialmente se forem várias (neste caso teriam que ser aninhadas), propomos o uso de uma clausula nova "exception:", que pode ser interpretada como uma reescrita do formato acima:

```
exception:
  <condição> implies <sinaliza exceção>
```

Nos contratos de operação de sistema, então, basta agora indicar a exceção dizendo qual condição que a gera.

Os contratos apenas indicam que a exceção pode ocorrer, mas seu tratamento deve ser feito necessariamente durante a atividade de design da camada de interface do sistema, já que as operações de sistema são implementadas na controladora, a qual é chamada pela interface.

O exemplo a seguir mostra um contrato com uma exceção indicada:

```
Context Livir::identificarComprador(umCpf:CPF)

  def:
    c = comprador→select(cpf = umCpf)

  post:
    self^addCompradorCorrente(c)

  exception:
    c→isEmpty() IMPLIES self^throw("cpf inválido")
```

Aqui, usa-se a mensagem básica throw como sinalizadora de exceção. Quando o código para esta operação for gerado, caso a condição ocorra, a exceção deve ser sinalizada para quem chamou a operação e nenhuma das pós-condições deve ser obtida.

Considera-se como exceção de contrato apenas a situação que não possa ser tratada dentro da própria operação, exigindo possivelmente o retorno do controle da aplicação ao usuário para tentar outras operações. Assim, exceçoes internas que uma operação possa resolver com seus próprios recursos, sem sinalizar para a interface, simplesmente não são mencionadas nos contratos, devendo ser tratadas quando do design do código da própria operação.

Assume-se também que, quando uma exceção ocorre em uma operação de sistema, a operação não é concluída e nenhuma de suas pós-condições é obtida. O sistema de informação deve ser mantido em um estado consistente, mesmo quando ocorrer uma exceção.

Como mencionado, algumas exceções podem ser convertidas em precondições desde que o analista vislumbre algum meio de verificar a condição antes de tentar executar a operação. Assim, o contrato com uma exceção acima poderia ser transformado em:

```
Context Livir::identificaComprador(umCpf:CPF)

  def:
    c = comprador→select(cpf = umCpf)

  pre:
    c→notEmpty()

  post:
```

```
self^addCompradorCorrente(c)
```

Nesse caso não há verificação da condição. Quem chamar a operação identificaComprador deve garantir que o CPF passado como parâmetro seja válido. O diagrama de sequencia de sistema deve deixar isso claro. Isso evita que a operação identificaComprador, quando for implementada faça verificações descessárias e também simplifica o código dela.

Apenas elementos conceituais (conceitos, atributos e associações) podem constar nos contratos de análise. Esses elementos terão necessariamente relação com as regras de negócio do sistema sendo analisado. As exceções aqui referenciadas, portanto, são exceções referentes às regras de negócio e não exceções referentes a problemas de hardware ou de comunicação. As exceções que podem ocorrer nos níveis de armazenamento, comunicação ou acesso a dispositivos externos são tratadas por mecanismos específicos nas camadas arquitetônicas correspondentes na atividade de design, e o usuário normalmente nem toma conhecimento delas.

8.6 Contratos Padrão CRUD

As subseções seguintes apresentam modelos de contratos para as operações típicas CRUD. São três contratos de operação de sistema, e um contrato de consulta de sistema. As operações são executadas sobre a classe Livro, definida segundo o modelo conceitual da Figura 8.8.

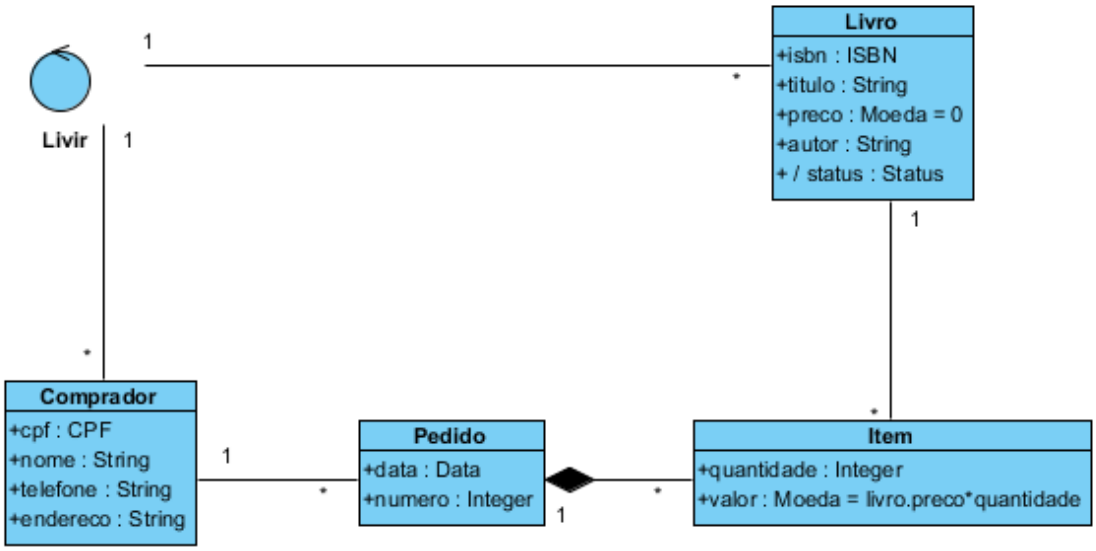

Figura 8.8: Modelo conceitual de referência para contratos de operações CRUD. Qualificar e conferir exemplos depois

8.6.1 Operações de Inserção (*Create*)

Para as operações e consultas ligadas à manutenção de informações (cadastros), os contratos serão mais ou menos padronizados. Inserção (*create*) de informação sempre vai incluir a criação de uma instância, alteração de seus atributos e a adição de pelo menos uma ligação com a controladora do sistema ou com alguma outra classe. Como exemplo, segue um contrato completo para a operação de inserção de um Livro:

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

```
  post:
    novoLivro.newInstanceOf(Livro) and
    self^addLivro(novoLivro) and
    novoLivro^setIsbn(umIsbn) and
    novoLivro^setTitulo(umTitulo) and
    novoLivro^setAutor(umAutor)
```

Como o atributo isbn já está estereotipado com <<oid>> não é necessário estabelecer como exceção a tentativa de inserir um livro cujo isbn já exista, pois essa exceção já é prevista pelo próprio estereótipo. Mas, se ao em vez de exceção esse fato for assumido como precondição, ela deve ficar explícita:

```
Context Livir criaLivro(umIsbn, umTitulo, umAutor)
  pre:
    livro→select(isbn=umIsbn)→isEmpty()
  post:
    novoLivro.newInstanceOf(Livro) and
    self^addLivro(novoLivro) and
    novoLivro^setIsbn(umIsbn) and
    novoLivro^setTitulo(umTitulo) and
    novoLivro^setAutor(umAutor)
```

8.6.2 Operações de Alteração (*Update*)

As alterações de informação vão envolver apenas a alteração de atributos ou, possivelmente, a adição e/ou remoção de links:

```
Context Livir::alteraLivro(umIsbn, novoIsbn, umTitulo, umAutor)
  def:
    li = livro→select(isbn=umIsbn)
  pre:
    li→notEmpty()
  post:
    li^setIsbn(novoIsbn) and
    li^setTitulo(umTitulo) and
    li^setAutor(umAutor)
```

Há dois padrões aqui envolvendo a alteração de atributos marcados com <<oid>>:

a) *não se permite que sejam alterados*. O objeto deve ser destruído e um novo objeto criado;

b) *permite-se a alteração*. Nesse caso, a operação passa dois argumentos: o ISBN anterior e o novo, como foi feito no exemplo anterior. Se o novo ISBN corresponder a um livro já existente haverá uma exceção porque esse atributo foi marcado como oid.

Também há duas opções em relação a verificar se o livro com identificador umISBN existe ou não:

a) entende-se como *exceção* o fato de não existir um livro com o ISBN dado;

b) define-se uma *precondição* que garante que o livro com umISBN existe, como foi feito no exemplo.

8.6.3 Operações de Exclusão (*Delete*)

As operações de sistema que excluem objetos terão de considerar as regras de restrição estrutural do modelo conceitual antes de decidir se um objeto pode ou não ser destruído e, caso possa, verificar se outros objetos também devem ser destruídos junto com ele.

No caso da Figura 8.8, por exemplo, uma instância de Livro não pode ser simplesmente destruída sem que se verifique o que acontece com possíveis instâncias de Item ligadas ao livro, já que do ponto de vista dos itens a associação com um livro é obrigatória.

Para que a exclusão seja feita sem ferir esse tipo de restrição estrutural pode-se escolher uma dentre três abordagens:

a) garantir por *precondição* que o livro sendo excluído não possui nenhum item ligado a ele. Por essa abordagem, apenas livros que não têm itens associados podem ser selecionados para exclusão;

b) garantir por *pós-condição* que todos os itens ligados ao livro também serão excluídos. Usa-se essa abordagem quando se quer que a operação de exclusão se propague para todos os objetos (no caso, itens) que têm associações obrigatórias com o livro. Essa propagação continua atingindo outros objetos em cadeia até que não haja mais ligações baseadas em associações obrigatórias;

c) utilizar uma *exceção* para abortar a operação caso seja tentada sobre um livro que tenha itens associados a ele.

Um possível contrato usando a abordagem de *precondição* teria esse formato:

```
Context Livir::excluiLivro(umIsbn)

  def:
    li = livro→select(isbn=umIsbn)

  pre:
    li→notEmpty() AND
    li.item→isEmpty()

  post:
    li^destroy()
```

Conforme indicado, a mensagem destroy elimina a instância de Livro, bem como todas as suas ligações (com outros objetos que não são instâncias de Item) que, portanto, não precisam ser removidas uma a uma.

Um possível contrato usando a abordagem de *pós-condição*, que propaga a exclusão a todos os objetos ligados ao livro por associações de multiplicidade de papel obrigatória, teria o seguinte formato:

```
Context Livir::excluiLivro(umIsbn)

  def:
    li = livro→select(isbn=umIsbn)

  pre:
    li→size() = 1

  post:
    li.item^destroy() and
    li^destroy()
```

A pós-condição afirma então que, além do livro, todos os itens ligados a ele foram destruídos. Como a classe Item não possui associações obrigatórias vindas de outras classes, a destruição pode parar por aí, mas caso contrário seria necessário destruir outros objetos.

Um possível contrato usando a abordagem de *exceção* teria este formato:

```
Context Livir::excluiLivro(umIsbn)

  def:
    li = livro→select(isbn=umIsbn)

  pre:
    li→size() = 1

  post:
    li^destroy()

  exception:
    li.item→notEmpty() IMPLIES
      self^throw("não é possível excluir este livro")
```

Escolhe-se a abordagem de pós-condição quando se quer propagar a exclusão a todos os elementos dependentes do objeto destruído. Por exemplo, se um comprador for destruído, quaisquer reservas que ele tenha no sistema podem ser destruídas junto.

Mas há situações em que não se quer essa propagação. Por exemplo, a remoção de um livro do catálogo não deveria ser possível se cópias dele já foram vendidas. Nesse caso, deve-se optar pelas abordagens de precondição ou exceção. A primeira vai exigir que se impeça que um livro com itens associados possa sofrer a operação de exclusão. Por exemplo, a lista de livros disponível para a operação de exclusão poderia conter apenas aqueles que não possuem itens associados. Caso não se queira ou não se possa dar essa garantia, resta a abordagem de exceção. Deixa-se o usuário tentar a exclusão, mas, se ela não for possível, uma exceção é criada.

Usualmente, informações cadastrais como essas nunca são removidas de sistemas de informação, mas marcadas com algum atributo booleano que indica se são instâncias ativas ou não. Afinal, não se poderia ter registros históricos de vendas de livros se os livros que saem de circulação fossem simplesmente removidos do sistema de informação.

8.6.4 Consultas (*Retrieve*)

A consulta simples do padrão CRUD implica simplesmente a apresentação de dados disponíveis sobre uma instância de um determinado conceito a partir de um identificador dessa instância. Nessas consultas não se fazem totalizações ou filtragens, ficando essas operações restritas aos relatórios (estereótipo <<rep>>).

Então, uma consulta simples para a classe Livro da Figura 8.7 seria:

```
Context Livir::consultaLivro(umIsbn):Tuple
  def:
    li = livro→select(isbn=umIsbn)
  body:
    Tuple{
      isbn = umIsbn,
      titulo = li.titulo,
      preco = li.preco,
      autor = li.autor,
      status = li.status
    }
```

A consulta do tipo CRUD retorna sempre uma tupla com os dados constantes nos atributos do objeto selecionado por seu identificador.

8.7 Contrato Padrão Listagem

Frequentemente é necessário utilizar operações de listagem de um ou mais atributos de um conjunto de instâncias de uma determinada classe para preencher listas ou menus em interfaces.

Um primeiro contrato de listagem (simples) vai apenas listar os ISBN dos livros catalogados na livraria:

```
Context Livir::listaIsbn():Set
  body:
    livro.isbn
```

Caso se deseje uma lista de múltiplas colunas como, por exemplo, ISBN e título, é necessário retornar uma coleção de tuplas, como:

```
Context Livir::listaIsbnTitulo():Set
  body:
    livro→collect(li|
      Tuple{
```

```
        isbn = li.isbn,
        titulo = li.titulo
      }
    )
```

Por vezes será necessário aplicar um filtro à lista, como, por exemplo, retornando apenas o ISBN e título de livros que não têm nenhum item associado (nunca foram vendidos). Nesse caso aplica-se um select ao conjunto antes de formar as tuplas:

```
Context Livir::listaIsbnTituloNaoVendido():Set
  body:
    livro→select(li|
      li.item→isEmpty()
    )→ collect(li|
      Tuple{
        isbn = li.isbn,
        titulo = li.titulo
      }
    )
```

A maioria das consultas de simples listagem terá apenas estes dois construtores: um select (filtro) seguido de um collect (projeção dos atributos que serão retornados). Mas algumas poderão ser mais complexas. Nesses casos, elas já não se encaixam no padrão Listagem, mas serão relatórios (<<rep>>) mais complexos.

8.8 Contratos para Operações não Padronizadas

Para as operações associadas com casos de uso, frequentemente haverá uma cadeia de execução ao longo de um dos fluxos, em que duas ou mais operações de sistema serão executadas. Possivelmente cada operação poderá deixar informações para as demais. Para melhor construir os contratos dessas operações, uma abordagem possível é seguir a sequência das operações de sistema do caso de uso expandido. Nesse processo, o analista deve se perguntar:

a) qual é o objetivo de cada operação?

b) o que cada uma delas produz?

c) o que cada uma delas espera que tenha sido produzido pelas anteriores?

d) que exceções poderiam ocorrer durante a execução?

e) seus parâmetros possuem classes de valores inválidos?

f) elas seguem algum padrão como CRUD, Listagem ou REP?

Ao responder a essas perguntas, o analista estará construindo contratos que permitirão que as operações sejam executadas de forma consistente no contexto de uma transação. Se for necessário adicionar novas consultas no diagrama de sequência para garantir certas precondições, isso deve ser feito nesse momento.

As subseções seguintes vão apresentar todos os contratos para as operações e consultas de sistema relacionadas ao caso de uso 01 *solicitar livros*.

8.8.1 Contratos para Estratégia *Stateless*

Inicialmente será apresentado um exemplo usando um design de controladora com a estratégia *stateless*. O modelo conceitual de referência é mostrado na Figura 8.9 e o diagrama de sequencia que detalha o caso de uso 01 *solicitar livros* é apresentado na Figura 8.10. Está se considerando, portanto, o caso de uso e modelo conceitual refinados no primeiro ciclo iterativo do projeto Livir. O diagrama de sequência apresenta apenas o fluxo principal para que o exemplo não se torne muito grande.

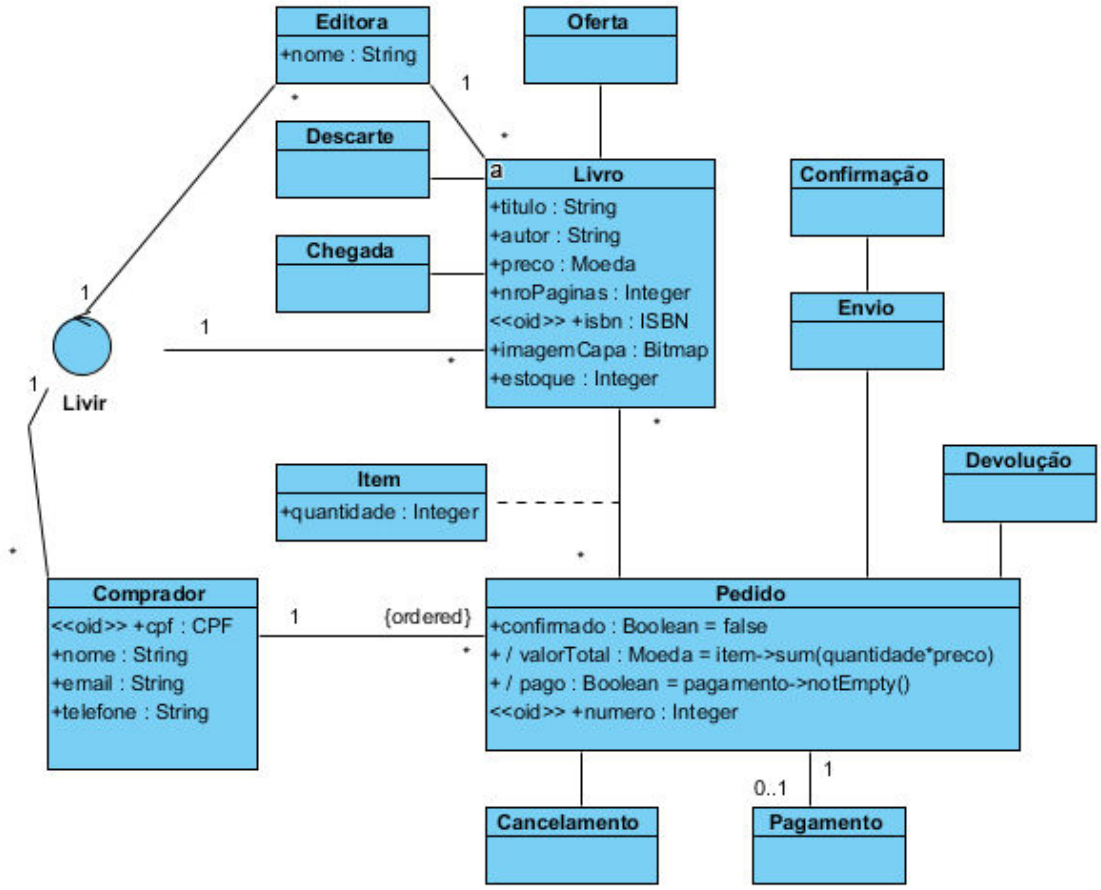

Figura 8.9: Modelo conceitual de referência para o exemplo.

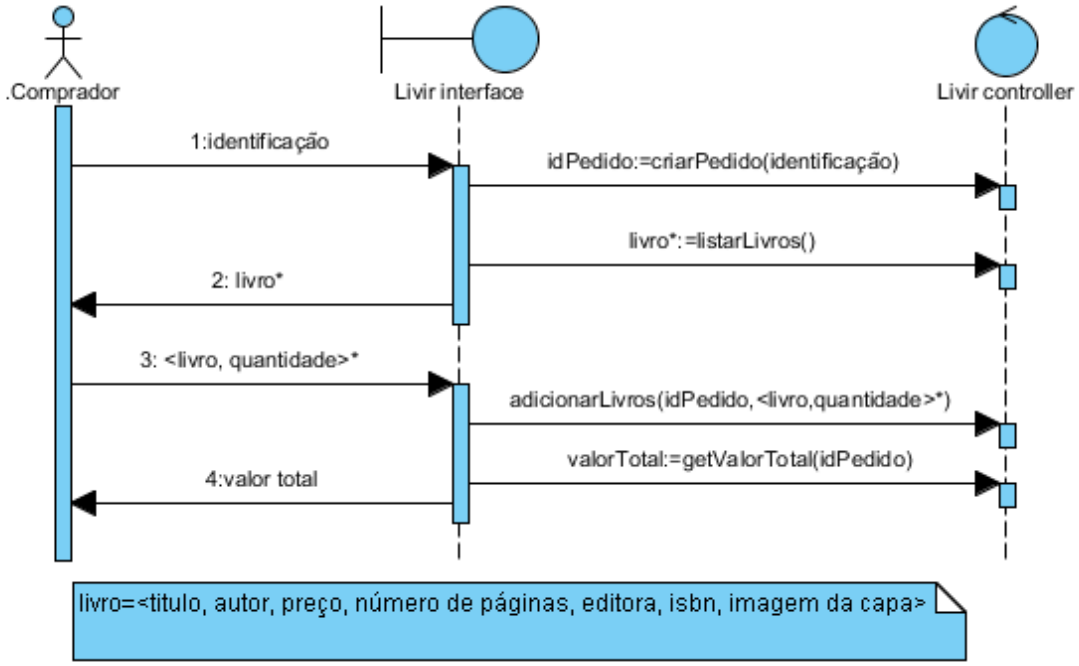

Figura 8.10: Diagrama de sequencia stateless de referência para o exemplo. Isbn ao invés de livro

A primeira operação, na sequencia do diagrama da Figura 8.10 é:

```
idPedido:=criaPedido(identificação)
```

O objetivo desta operação é criar uma instância de Pedido, associar essa instância ao comprador cuja identificação é passada como parâmetro, inicializar seus atributos e retornar o identificador do pedido. O contrato poderia ser definido assim:

```
Context Livir::criaPedido(identificacao:CPF):Integer
  def:
    c = comprador→select(cpf=identificacao)
  post:
    p.newInstanceOf(Pedido) and
    p^addComprador(c) and
    p^setNumero(GeradorIdPedido.getNovoId())
  exception:
    c→isEmpty() implies throw("Comprador inexistente")
  return:
    p.numero
```

Logo no início já foi decidido que o CPF seja o identificador do cliente, pois é o único atributo candidato no momento (<<oid>>).

O número do novo pedido é gerado por um *singleton*, um objeto globalmente acessível cuja única função é gerar o próximo número de pedido. Esse objeto é GeradorIdPedido.

Observando o parâmetro da operação, a única classe de valores inválidos para identificacao seriam CPFs de pessoas que não constam na base como compradores. Como não há nenhuma operação anterior no diagrama de sequencia, não se pode grantir isso como pré-condição. Então, ela será considerada, por ora, como uma exceção. Mais adiante voltaremos a discutir essa questão.

Finalmente, a clausula "return:" indica que um valor é retornado pela operação, no caso, o número do novo pedido. Esse também é o único atributo do pedido que precisa ser inicializado no contrato, já que pago e valorTotal são atributos derivados e confirmado é definido com valore inicial.

A segunda mensagem a ser especificada é uma consulta:

```
livro* = listaLivro()
```

Esta consulta deve retornar vários atributos de cada livro: titulo, autor, preço, numero de paginas, editora, isbn e imagem de capa. Como não há filtro, a princípio, serão listados todos os livros em cadastro. Assim, a consulta será uma simples aplicação de projeção dentro do padrão listagem:

```
Context Livir::listaLivro():Set
  body:
    livro→collect(li|
      Tuple {
        titulo=li.titulo,
        autor=li.autor,
        preco=li.preco,
        nroPaginas=li.nroPaginas,
        editora=li.editora.nome,
        isbn=li.isbn,
        imagemCapa=li.imagemCapa
      }
   )
```

Praticamente todos os campos retornados são atributos do livro, exceto "editora", que é na verdade o nome da instância de Editora associada ao livro.

Pode-se argumentar, com razão, neste ponto, que listar todos os livros existentes no cadastro seria uma operação bastante demorada e desnecessária. O design do caso de uso poderia ser melhorado neste ponto estabelecendo, por exemplo, que o usuário deveria digitar uma palavra chave e que apenas os livros que satisfizessem o critério de busca seriam efetivamente listados. Poderia-se estabelecer uma exceção para essa

consulta caso o número de resultados para a palavra chave exceda um certo limite (por exemplo 100), caso em que seria solicitada uma nova palavra-chave mais específica.

A operação seguinte é: ***tirar infinitivo dos verbos dos nomes de operaçao

```
adicionaLivros(idPedido,<isbn,quantidade>*)
```

Seu objetivo é adicionar no pedido identificado cada um dos livros da lista passada como parâmetro com a respectiva quantidade. Uma das primeiras coisas que o contrato deve estabelecer é identificar qual é o pedido em questão. Temos o idPedido, mas não á uma associação direta entre a controladora e a classe pedido para que esse idPedido possa ser usado. Existem diferentes opções, pelo menos para resolver isso:

a) Criar uma associação direta entre a controladora e a classe Pedido e acessar o pedido com a expressão: p = pedido→select(idPedido).
b) Ao invés do identificador do pedido, passar o identificador do comprador como parâmetro da operação adicionarLivros e acessar o pedido com a expressão: p = comprador→select(CPF=identificador).pedido→last(), ou seja, o último pedido do comprador indicado (que, por definição, deve ser o corrente).
c) Obter o pedido a partir da lista de todos os pedidos de todos os compradores: p = comprador.pedido→select(numero=idPedido).

Optamos pela terceira forma visto que ela não necessita nenhuma alteração no modelo nem na operação, embora possa parecer menos "eficiente". Porém, deve-se lembrar neste ponto que o que se está fazendo aqui é especificar um resultado desejado. Esse resultado poderá ser obtido de diferentes formas por um código de programa e essas formas poderão tão otimizadas quanto se queira.

O contrato poderia ser feito assim:

```
Context Livir::adicionarLivros(idPedido:Integer,lq:Set)
  def:
    p = comprador.pedido→select(numero=idPedido)
  pre:
    p→notEmpty() and
    lq→forAll(par|
      livro→select(isbn=par[1]) →notEmpty()
    )
  post:
    lq→forAll(par|
      p^addLivro(livro→select(isbn=lq[1]) and
      p.item[livro→select(isbn=lq[1])]^setQuantidade(lq[2])
```

```
  exception:
    lq→exists(par|
      livro→select(isbn=lq[1]).estoque<lq[2])) implies
    self^throw("Solicitada quantidade maior do que em
estoque")
```

Em relação às precondições, como o parâmetro idPedido é obtido como retorno da primeira operação e, portanto, válido, pode-se assumir como precondição que ele corresponde a um pedido válido. Já a lista de ISBNs correspondendo à primeira posição de cada elemento do parâmetro lq, é obtida a partir de ISBNs retornados pela consulta de sistema listarLivros. Assim, cada um destes ISBNs corresponde a um livro que existe na base de dados e, portanto, é válido.

A pós-condição percorre todos os valores da lista lq passada como parâmetro. Inicialmente addLivro referencia a criação da associação entre o pedido e o livro. Isso produz um item em função da classe de associação. Então a expressão final p.item[livro→select(isbn=lq[1])]^setQuantidade(lq[2]) seleciona esse item referente ao livro que acaba de ser adicionado ao pedido, e altera o valor da quantidade para o valor correspondente na lista lq.

O contrato acima poderia ser bastante simplificado se a operação de sistema original, ao invés de passar uma lista com todos os livros selecionados passasse um livro de cada vez. Neste caso, o contrato poderia ser assim:

```
Context Livir:adicionarLivro(idPedido,idLivro,quant)
  def:
    p = comprador.pedido→select(numero=idPedido)
    li = livro→select(isbn=idLivro)
  pre:
    p→notEmpty() and
    li→notEmpty()
  post:
    p^addLivro(li) and
    p.item[li]^setQuantidade(quant)
  exception:
    quant > li.estoque
      throw("Quantidade solicitada acima do estoque").
```

Assim, possivelmente seria uma boa opção de design substituir a operação de sistema que adiciona todos os livros de uma vez por uma sequencia de operações que adiciona um livro de cada vez, pois desta forma se obtêm contratos e operações mais simples.

A última mensagem do diagrama de sequencia é uma simples consulta:

```
valorTotal := getValorTotal(idPedido)
```

Seu contrato seria:

```
Context Livir::getValorTotal(idPedido:String):Moeda
  def:
    p = comprador.pedido→select(numero=idPedido)
       pre:
    p→notEmpty()
  body:
    p.valorTotal
```

Trata-se de uma consulta com precondição. Assume-se que a precondição será sempre verdadeira porque o idPedido passado como parâmetro é o mesmo retornado pela operação criarPedido, e assim só pode ser válido.

8.8.2 Contratos para Estratégia *Statefull*

Para o exemplo a seguir será considerado um exemplo de diagrama de sequencia projetado com a estratégia *statefull* e já considerando que a operação de inserção de livros ocorra com um livro de cada vez, o que já foi mostrado, é bem mais simples. O modelo conceitual deverá ser adaptado para que a informação sobre o cliente corrente seja mantida. Ao mesmo tempo, assume-se que o pedido corrente seja o último pedido deste cliente. Essa informação então passará a ser disponível a todas as operações ao longo do diagrama de sequencia. Além disso, adicionamos qualificadores ao modelo conceitual onde possível para simplificar as operações de acesso aos elementos.

O modelo conceitual modificado, então, será como o mostrado na Figura 8.11.

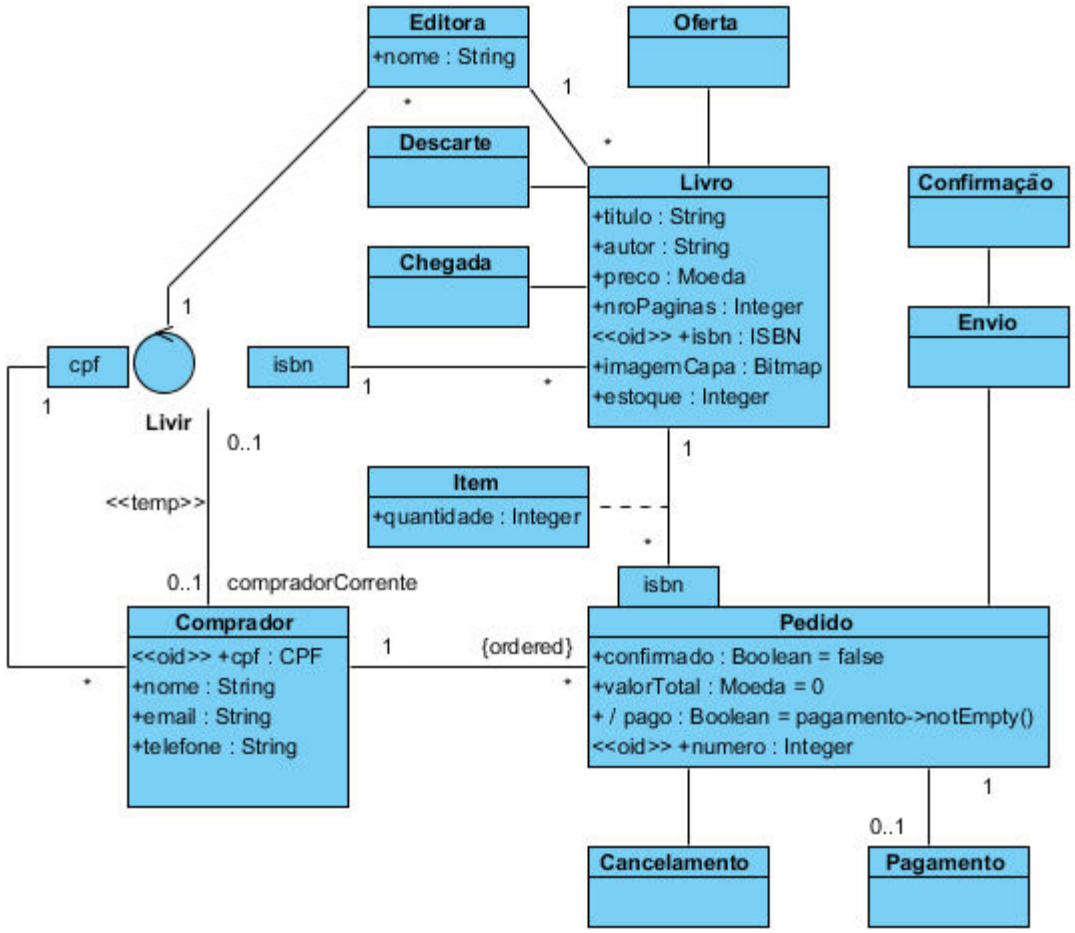

Figura 8.11: Modelo conceitual de referencia. **valor total derivado?

Já o diagrama de sequencia será alterado de acordo com a estratégia *statefull* e a entrada de livros um por vez, como mostrado na Figura 8.12.

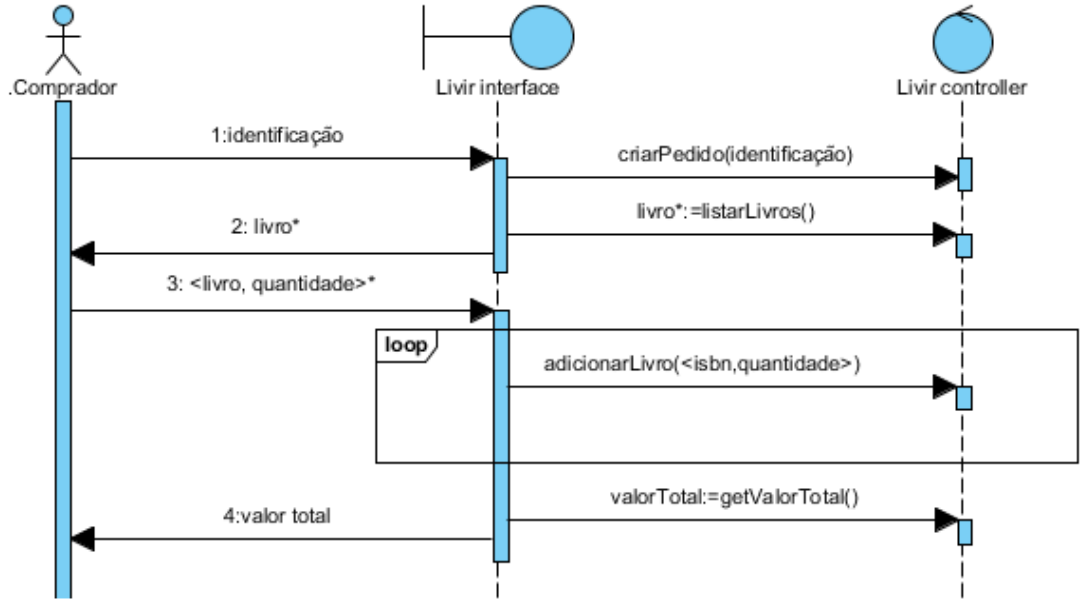

Figura 8.12: Diagrama de sequencia de sistema statefull de referência. Isbn//livro

O primeiro contrato, então, é feito para a operação:

```
criarPedido(identificacao)
```

Esse contrato tem os mesmos objetivos da sua versão *stateless*, mas adicionalmente deve guardar na forma da associação temporária o comprador corrente (aquele identificado pelo parâmetro) e não necessita retornar o *id* do pedido. Assim, o contrato poderia ser:

```
Context Livir::criarPedido(identificacao:CPF)
   def:
    c = comprador[identificacao]
  post:
    p.newInstanceOf(Pedido) and
    p^addComprador(c) and
    p^setNumero(GeradorIdPedido.getNovoId()) and
    self^addCompradorCorrente(c)
  exception:
    c→isEmpty() implies throw("Comprador inexistente")
```

A operação seguinte, listarLivros, que não tem parâmetros, é definida exatamente como na versão *stateless*.

Na sequência, a operação adicionarLivro, não vai receber como parâmetro nem o identificador do cliente, nem o identificador do pedido. Ela deverá acessar o último pedido do cliente corrente e neste pedido adicionar o novo livro informado:

```
Context
Livir:adicionarLivro(idLivro:String,quant:Integer)
  def:
    p = compradorCorrente.pedido→last()
    li = livro[idLivro]
  pre:
    p→notEmpty() and
    li→notEmpty()
  post:
    p^addLivro(li) and
    p.item[li]^setQuantidade(quant)
  exception:
    quant>li.estoque  throw("Quantidade  solicitada  acima
do estoque").
```

Finalmente, a operação getValorTotal, que também não vai receber como parâmetro o número do pedido, deve acessa-lo como sendo o último pedido do comprador corrente:

```
Context Livir::getValorTotal():Moeda
  def:
    p = compradorCorrente.pedido→last()
  pre:
    p→notEmpty()
  body:
    p.valorTotal
```

Poderia-se colocar ao final do diagrama de sequencia uma operação de finalização e *commit* dos dados, que entre outras coisas iria remover a ligação compradorCorrente. Mas como este tipo de finalização iria acontecer em todos os casos de uso, melhor seria deixar implícito e indicar que cada transação representada em um diagrama de sequencia, se chegar ao final, fará *commit* dos dados (vai salva-los em memória persistente) e eliminará da memória principal todas as associações temporárias. Mas isso é uma questão de design, também, e não precisa ser resolvida neste momento ainda.

8.9 O Processo Até Aqui

	Inception	Elaboration	Construction
Business Modeling	Elaborar visão geral: • Construir diagrama de casos de uso de negócio determinando o escopo de automatização. • Construir diagramas de atividade para casos de uso potencialmente automatizáveis. • Construir diagramas de estado para objetos de negócio relevantes.		
Requirements	Elaborar diagrama de casos de uso de sistema. Identificar requisitos não funcionais (anotações nos casos de uso de sistema). Identificar requisitos suplementares.	Detalhamento ou expansão dos casos de uso.	
Analisys and Project	Elaborar o modelo conceitual preliminar.	Elaboração dos diagramas de sequencia de sistema. Refinamento e organização do modelo conceitual. Melhoria do modelo conceitual com aplicação de padrões de análise. Modelagem funcional, com a	

		definição dos contratos de todas as operações e consultas de sistema.	
Implementation			
Test			
Project Management	Estimação de esforço, duração e tamanho médio de equipe para o projeto. Estimação da duração e quantidade de ciclos iterativos. Planejamento inicial das entregas para cada ciclo iterativo.		

8.10 Questões

1. Explique e exemplifique como uma pré-condição pode ser transformada em exceção e vice versa. Qual a diferença entre exceções em contratos de operação e contratos de consulta de sistema?
2. Quais são os cinco tipos possíveis de pos-condições e quais operações básicas as definem? Como podem ser representadas em contratos OCL?
3. Considere o diagrama da Figura 8.12. Como esse diagrama teria que ser modificado caso se quisesse que a exceção da operação criarPedido fosse transformada em uma precondição?