Análise e projeto de sistemas de informação orientados a objetos

Raul Sidnei Wazlawick

Análise e projeto de sistemas de informação orientados a objetos

2ª edição revista e atualizada

ELSEVIER

Série Campus — SBC Sociedade Brasileira de Computação

CAMPUS

Copidesque: Ivone Teixeira
Revisão: Bruno Barrio
Editoração Eletrônica: SBNIGRI Artes e Textos Ltda.

Elsevier Editora Ltda.
Conhecimento sem Fronteiras
Rua Sete de Setembro, 111 – 16º andar
20050-006 – Centro – Rio de Janeiro – RJ – Brasil

Rua Quintana, 753 – 8º andar
04569-011 – Brooklin – São Paulo – SP – Brasil

Serviço de Atendimento ao Cliente
0800-0265340
sac@elsevier.com.br

ISBN 978-85-352-3916-4

Nota: Muito zelo e técnica foram empregados na edição desta obra. No entanto, podem ocorrer erros de digitação, impressão ou dúvida conceitual. Em qualquer das hipóteses, solicitamos a comunicação ao nosso Serviço de Atendimento ao Cliente, para que possamos esclarecer ou encaminhar a questão.

Nem a editora nem o autor assumem qualquer responsabilidade por eventuais danos ou perdas a pessoas ou bens, srcinados do uso desta publicação.

CIP-Brasil. Catalogação-na-fonte.
Sindicato Nacional dos Editores de Livros, RJ

W372a 2.ed. Wazlawick, Raul Sidnei
Análise e projeto de sistemas de informação orientados a objetos / Raul Sidnei Wazlawick. – 2.ed. – Rio de Janeiro: Elsevier, 2011.
(Série SBC, Sociedade Brasileira de Computação)

Apêndice: Sumário OCL
Inclui bibliografia e índice
ISBN 978-85-352-3916-4

1. Métodos orientados a objetos (Computação). 2. UML (Computação). 3. Análise de sistemas. 4. Projeto de sistemas. 5. Software – Desenvolvimento. I. Sociedade Brasileira de Computação. II. Título. III. Série.

10-2632. CDD: 005.117
CDU: 004.414.2

Dedicatória

Este livro é dedicado aos meus pais e antepassados, sem os quais eu não existiria.

Agradecimentos

Desejo agradecer a várias pessoas que, de uma forma ou outra, tornaram este livro possível: ao mestre Luiz Fernando Bier Melgarejo, por apresentar as ideias de orientação a objetos já em 1987; ao colega Marcos Eduardo Casa, por todos os trabalhos desenvolvidos em conjunto nos tempos em que orientação a objetos era "coisa de outro mundo"; ao colega Antônio Carlos Mariani, pelo Mundo dos Atores, ferramenta que tanto usamos para ensinar programação orientada a objetos; ao ex-aluno Leonardo Ataide Minora, por inicialmente me chamar a atenção para o livro de Larman; às empresas e órgãos públicos que possibilitaram a implantação dessas técnicas em ambientes reais de produção de software e especialmente ao engenheiro de software Gilmar Purim, pelas interessantes discussões que muito contribuíram para dar a forma final a este livro; aos ex-alunos Everton Luiz Vieira e Kuesley Fernandes do Nascimento, por terem ajudado a consolidar algumas das técnicas quando da aplicação delas a um interessante sistema Web; ao Departamento de Informática e Estatística da UFSC, pela oportunidade de concretizar este trabalho; e a Dayane Montagna, por digitar o primeiro rascunho deste livro a partir das gravações das minhas aulas.

Agradeço também aos mais de mil ex-alunos, vítimas da minha disciplina de Análise e Projeto de Sistemas Orientados a Objetos – suas dúvidas e dificuldades me fizeram pesquisar e aprender muito mais; ao colega Rogério Cid Bastos, por muitas orientações recebidas; e, finalmente, aos amigos e irmãos, pelos momentos de descontração e higiene mental.

Prefácio

Este livro apresenta, de maneira didática e aprofundada, elementos de análise e projeto de sistemas de informação orientados a objetos.

A área de desenvolvimento de software tem se organizado nos últimos anos em torno da linguagem de modelagem UML (*Unified Modeling Language*) e do processo UP (*Unified Process*), transformados em padrão internacional pela OMG (*Object Management Group*).

Não se procura aqui realizar um trabalho enciclopédico sobre UP ou UML, mas uma apresentação de cunho estritamente prático, baseada em mais de vinte anos de experiência em ensino, prática e consultoria em análise, projeto e programação orientada a objetos.

Este livro diferencia-se da maioria de outros livros da área por apresentar em detalhes as técnicas de construção de contratos de operação e consulta de sistema de forma que esses contratos possam ser usados para efetiva geração de código.

Novos padrões e técnicas de modelagem conceitual são detalhadamente apresentados, técnicas estas também adequadas para uso com contratos e diagramas de comunicação, de forma a garantir geração automática de código; não apenas de esqueletos, mas de código final executável.

Em relação aos casos de uso de análise, o livro apresenta, em detalhes, técnicas para ajudar a decidir o que considerar efetivamente como caso de uso. Essa dificuldade tem sido sistematicamente relatada por analistas de várias partes do Brasil, a partir do contato obtido em cursos ministrados pelo autor.

Ao contrário de outros livros da área, que se organizam em torno da apresentação dos diagramas UML e procuram explicar todos os seus possí-

veis usos, este livro se concentra nas *atividades* com as quais o analista e o projetista de software possivelmente vão se deparar e sugere quais diagramas poderiam ajudá-los e de que forma.

Algumas empresas brasileiras ainda têm dificuldade em conseguir exportar software devido à falta de flexibilidade e manutenibilidade dos sistemas gerados. Este livro apresenta um conjunto de informações e técnicas que pode suprir essa carência. As técnicas em questão foram implementadas com êxito pelo autor na empresa TEClógica Ltda., em Blumenau, no desenvolvimento de um projeto de grande porte em 2004. Posteriormente, as técnicas foram aplicadas e aperfeiçoadas nos departamentos de tecnologia de informação do Ministério Público de Santa Catarina, Tribunal Regional do Trabalho do Mato Grosso e Justiça Federal de Santa Catarina, contendo agora ainda mais orientações e detalhes do que na primeira edição deste livro.

O livro é direcionado a profissionais de computação (analistas, projetistas e programadores) e a estudantes de graduação e pós-graduação das disciplinas de Análise e Projeto de Sistemas e Engenharia de Software. Como conhecimentos prévios são recomendados rudimentos sobre orientação a objetos, notação UML e fundamentos de banco de dados.

Para que o livro pudesse aprofundar ainda mais as informações sobre análise e projeto orientados a objetos sem se tornar demasiadamente longo, foram suprimidas nesta segunda edição algumas informações referentes ao *processo* de Engenharia de Software que estavam presentes na primeira edição. Esses processos serão descritos de forma detalhada pelo autor em um novo livro sobre Engenharia de Software a ser lançado brevemente. Além disso, para ganhar espaço e dinamismo, os exercícios, anteriormente incluídos no livro, passam a estar disponíveis apenas na Internet (www.elsevier.com.br/wazlawick ou www.inf.ufsc.br/~raul/).

Raul Sidnei Wazlawick
Florianópolis, 19 de fevereiro de 2010.

Capítulo

1

Introdução

O principal objetivo deste livro é apresentar um conjunto de informações práticas que possibilite aos desenvolvedores de software a compreensão e utilização da *orientação a objetos* de forma consciente e eficaz.

A justificativa deste trabalho parte da observação de que há uma vasta literatura que visa apenas a apresentar os diagramas da UML (OMG, 2009) de forma sintática (por exemplo, Erickson & Penker, 1998), mas poucos livros que ofereçam informações suficientes para viabilizar a aplicação eficaz da orientação a objetos no desenvolvimento de software no mundo real.

Neste livro, são feitos uma interpretação e um detalhamento de partes do método de análise e projeto apresentado por Larman (2002), o qual é baseado no *Processo Unificado* ou *UP* (Jacobson, Booch & Rumbaugh, 1999; Scott, 2001).

A motivação para o uso do método de Larman como base para este trabalho deve-se ao fato de que Larman apresenta uma abordagem concisa e eficiente para análise e projeto de sistemas orientados a objetos. Nessa abordagem, cada artefato (documento ou diagrama) tem uma razão muito clara para existir, e as conexões entre os diferentes artefatos são muito precisas.

Pode-se até dizer que o método seria inspirado em *Extreme Programming* ou XP (Beck, 2004) no qual, em vez de usar uma linguagem de progra-

mação (como Java ou PHP), utilizam-se diagramas e outros artefatos. Dentro dessa proposta, diagramas e artefatos só fazem sentido se contribuem diretamente para a geração automática de código. Não são usados, portanto, como mera documentação, mas como programação em nível muito alto.

Em relação ao processo descrito por Larman, este livro aprofunda e apresenta conceitos srcinais em vários tópicos, como, por exemplo, o uso de *Object Constraint Language* ou OCL (OMG, 2006) para construção de contratos de operação de sistema, a discussão sobre quais passos são realmente obrigatórios em casos de uso expandidos, a noção de contratos de consultas de sistema, a interconexão entre os contratos e os diagramas de comunicação ou sequência e o projeto da camada de interface com o uso de WebML (Ceri *et al.*, 2003).

Desde o início, o uso dessas práticas vai levando sistematicamente à produção de software de boa qualidade, isto é, bem organizado, baseado em uma arquitetura multicamadas e com possibilidade de incluir novos requisitos e modificar os requisitos existentes.

1.1. Desenvolvimento de Sistemas Orientados a Objetos

Em primeiro lugar, deve-se discutir o que é realmente desenvolver sistemas orientados a objetos. Ao observar a forma como a análise e o projeto de sistemas vêm sendo ensinados e praticados em certos lugares, pode-se verificar que muitos profissionais simplesmente adotam uma linguagem orientada a objetos ou até algum fragmento de processo orientado a objetos, mas sem ter realmente muita noção do que estão fazendo.

O problema de fazer os profissionais migrarem de paradigmas mais antigos para orientação a objetos apresenta situações caricatas. Em determinada ocasião, durante uma palestra, alguém comentou que programava há muitos anos usando a linguagem C e que havia resolvido começar a trabalhar com C++, mas que após alguns meses não notou absolutamente nenhuma vantagem nessa migração. Essa pessoa realmente não viu diferença entre as linguagens porque faltou a ela saber o que havia por trás da nova abordagem, e que a linguagem C++ é mais interessante do que a linguagem C não porque tem mais recursos ou eficiência, mas porque traz consigo uma maneira muito mais sensata de se pensar e organizar sistemas.

b) *é centrado na arquitetura* o processo de desenvolvimento prioriza a construção de uma arquitetura de sistema que permita a realização dos requisitos. Essa arquitetura baseia-se na identificação de uma estrutura de classes, produzida a partir de um modelo conceitual;
c) *é iterativo e incremental*: a cada ciclo de trabalho realizado, novas características são adicionadas à arquitetura do sistema, deixando-a mais completa e mais próxima do sistema final.

O UP comporta, em suas disciplinas as atividades de estudo de viabilidade, análise de requisitos, análise de domínio, projeto etc. Porém, essas atividades aparecem no UP associadas, com maior ou menor ênfase, às quatro grandes fases do UP, que são: concepção, elaboração, construção e transição.

A fase de *concepção* incorpora o estudo de viabilidade, o levantamento dos requisitos e uma parte da sua análise. A fase de *elaboração* incorpora o detalhamento da análise de requisitos, a modelagem de domínio e o projeto. A fase de *construção* corresponde à programação e testes, e a fase de *transição* consiste na instalação do sistema e migração de dados.

A fase de concepção, denominada *inception* em inglês, é a primeira fase do processo unificado, na qual se procura levantar os principais requisitos e compreender o sistema de forma abrangente. Os resultados dessa fase usualmente são um documento de requisitos e riscos, uma listagem de casos de uso de alto nível e um cronograma de desenvolvimento baseado nesses casos de uso.

As fases de elaboração e construção ocorrem em ciclos iterativos. A elaboração incorpora a maior parte da análise e projeto, e a construção incorpora a maior parte da implementação e testes. É durante os ciclos iterativos propriamente ditos que acontece a análise detalhada do sistema, a modelagem de domínio e o projeto do sistema usando os *padrões de projeto*.

Na fase de transição, o sistema, depois de pronto, será implantado substituindo o sistema atual, seja ele manual ou computadorizado.

Apesar de ser um processo prescritivo, o UP pode também ser realizado como um processo ágil, com poucos artefatos e burocracia, permitindo o desenvolvimento de software de forma eficiente, uma vez que o que interessa ao cliente é o software pronto e não uma pilha de documentos justificando por que não ficou pronto.

Para que isso seja obtido, a documentação deve ser dirigida à produção do software. Cada atividade realizada pelo desenvolvedor deve ter um obje-

tivo muito claro e uma utilização precisa visando sempre à produção de um código que atenda aos requisitos da melhor forma possível no menor tempo.

1.4. As Atividades de Análise e Projeto no Contexto do Processo Unificado

As diferentes atividades de análise e projeto não ocorrem de forma estanque em cada uma das fases do processo unificado, mas podem ocorrer com maior ou menor ênfase nas diferentes fases. A Figura 1.1 é a representação clássica da distribuição das atividades de desenvolvimento de sistemas e sua ênfase nas diferentes fases da implementação mais conhecida do UP, denominada RUP, ou *Rational Unified Process* (Kruchten, 2003).

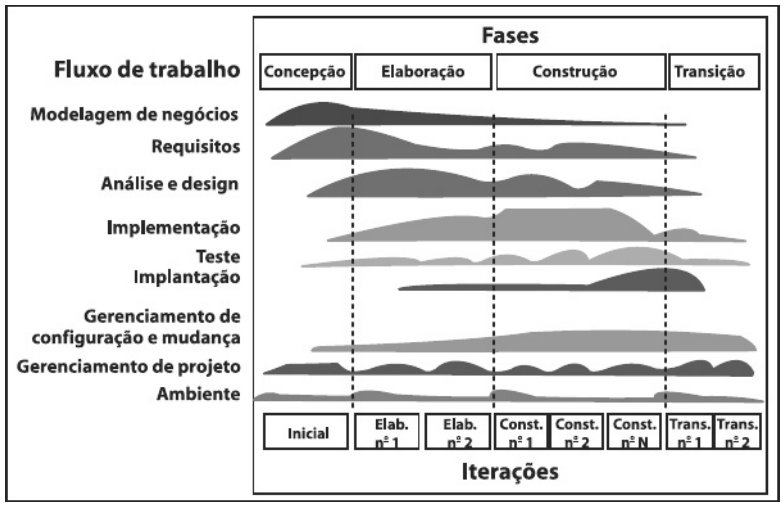


Figura 1.1: As diferentes ênfases das atividades de desenvolvimento ao longo das quatro fases do Processo Unificado (fonte: IBM).

Este livro vai abordar com maior ênfase as atividades típicas de *análise* e *projeto*. Para uma visão maior sobre gerenciamento de projeto e processo deve-se consultar um livro de engenharia de software, como, por exemplo, o de Pressman (2010). Já as atividades de programação são orientadas de acordo com a linguagem escolhida.

Então, com o foco nas atividades de análise e projeto, a fase de concepção vai exigir do analista uma visão inicial e geral do sistema a ser desenvolvido (Capítulo 2). Essa visão pode ser obtida a partir de entrevistas, documentos e sistemas. Para apoiar a modelagem dessa visão geral pode-se usar

diagramas de máquina de estados ou diagramas de atividades da UML, que correspondem, nessa fase, à modelagem de negócios.

A partir dessa compreensão do negócio pode-se analisar mais aprofundadamente cada uma das atividades ou estados para obter os requisitos funcionais e não funcionais do sistema (Capítulo 3).

Ainda na fase de concepção pode-se elaborar com o diagrama de classes um modelo conceitual preliminar (Capítulo 7) para compreensão da estrutura da informação a ser gerenciada pelo sistema.

Esse modelo conceitual preliminar e os requisitos já levantados ajudarão a compreender quais são os processos de negócio e processos complementares da empresa, obtendo-se assim os casos de uso de alto nível (Capítulo 4). Esses casos de uso deverão ser usados como base para planejar o restante do desenvolvimento.

A fase de elaboração inicia com a expansão dos casos de uso de alto nível (Capítulo 5) e posterior representação de seus fluxos através dos diagramas de sequência de sistema (Capítulo 6), quando são descobertas as operações e consultas de sistema.

Na fase de elaboração, o modelo conceitual poderá ser refinado, e mais informações agregadas a ele a partir das descobertas feitas durante a expansão dos casos de uso.

Ainda nessa fase poderão ser feitos os contratos de operação e consulta de sistema (Capítulo 8) que definem a funcionalidade, ou seja, os resultados de cada operação e consulta realizadas.

Posteriormente, com os contratos e modelo conceitual à mão, podem ser criados os diagramas de comunicação ou sequência (Capítulo 9), que, seguindo critérios de delegação e distribuição de responsabilidades, vão mostrar quais métodos devem ser criados em cada classe e como esses métodos devem ser implementados, produzindo assim o diagrama de classes de projeto, ou DCP.

Ainda na fase de elaboração, pode-se passar ao projeto da interface (Capítulo 10). Como grande parte dos sistemas de informação são desenvolvidos para Web ou interfaces compatíveis com modelos Web, este livro apresenta o WebML (Ceri *et al.*, 2003) como opção para a modelagem para esse aspecto do sistema.

A fase de construção inclui a geração de bancos de dados (Capítulo 11) e a geração de código e testes (Capítulo 12). A persistência dos dados usual-

mente não precisa ser modelada, pois pode ser gerada automaticamente. Assim mesmo, o livro mostra como ela ocorre quando se usa um *framework* orientado a objetos para persistência. Também a geração de código é apresentada como um conjunto de regras que podem ser automatizadas. As atividades de teste de software são adaptadas neste livro para as características peculiares da orientação a objetos.

Essas são as questões que devem ser trabalhadas no primeiro momento. Nessa fase, também surge outra questão que os analistas frequentemente esquecem: comprar ou construir? Muitas vezes, o produto que o cliente quer já está pronto, podendo-se comprar um pacote e adaptá-lo para as necessidades da empresa, em vez de construir a partir do zero.

Essas questões devem ser respondidas em um tempo relativamente curto. Por isso, sugere-se que a fase de concepção não dure muito tempo. Por que motivo? Porque, nessa fase, normalmente, o analista e o cliente ainda não têm um contrato fechado; ela é, do ponto de vista do analista, um investimento no futuro.

A *visão geral do sistema* , ou *sumário executivo*, é um documento em formato livre, no qual o analista deve escrever o que ele conseguiu descobrir de relevante sobre o sistema após as conversas iniciais com os clientes e usuários.

Aqui não são propostas regras sobre como deve ser escrito esse documento. Mas sugere-se que ele não seja longo demais. Uma ou duas páginas de texto e alguns diagramas parece ser suficiente para descrever, de forma resumida, a maioria dos sistemas. Com mais do que isso, possivelmente estarão sendo incluídos detalhes que não são relevantes nesse resumo e que deveriam ser tratados em outros documentos, como análise de riscos, requisitos ou casos de uso.

Na Figura 2.1 é apresentada a visão geral de um sistema de livraria virtual fictício, que será usado como exemplo para as técnicas de modelagem ao longo do livro.

Sistema Livir: Livraria Virtual

Visão Geral do Sistema

O sistema deve gerenciar todos os processos de uma livraria virtual, desde a aquisição até a venda dos livros. O acesso dos compradores e gerentes deve ser feito através de um site Web e possivelmente com outras tecnologias. Os compradores fazem as transações pagando com cartão de crédito.

Existem promoções eventuais pelas quais os livros podem ser comprados com desconto.

De início, a livraria vai trabalhar apenas com livros novos a serem adquiridos de editoras que tenham sistema automatizado de aquisição. O sistema a ser desenvolvido deve conectar-se aos sistemas das editoras para efetuar as compras.

O sistema deve calcular o custo de entrega baseado no peso dos livros e na distância do ponto de entrega. Eventualmente pode haver promoções do tipo "entrega gratuita" para determinadas localidades.

O sistema deve permitir a um gerente emitir relatórios de livros mais vendidos e de compradores mais assíduos, bem como sugerir compras para compradores baseadas em seus interesses anteriores.

Quando um livro é pedido, se existe em estoque, é entregue imediatamente, senão o livro é solicitado ao fornecedor, e um prazo compatível é informado ao comprador.

Figura 2.1: Sumário executivo do sistema Livir.

Para permitir a apresentação do sistema no curto espaço disponível neste livro, várias simplificações foram consideradas. A descrição e a modelagem de um sistema real poderiam ser bem mais extensas.

Observa-se que a visão geral do sistema é apenas uma descrição desestruturada. Existem aqui informações de nível gerencial e de nível operacional. Muitas vezes, até detalhes sobre tecnologias a serem empregadas também são descritos.

O que o analista deve ter em mente é que esse documento descreve as principais preocupações do cliente, as quais serão mais bem estruturadas nas fases posteriores do processo de análise.

2.1. Modelagem de Negócio com Diagrama de Atividades

Para melhor compreensão do funcionamento da empresa, pode-se criar um ou mais modelos das atividades de negócio. Para tal, pode ser usado o diagrama de atividades da UML. Os diagramas de atividades podem ser usados para representar processos em nível organizacional, ou seja, de forma muito mais ampla do que a mera visão de requisitos de um sistema informatizado.

O diagrama pode ser dividido em *raias* (*swimlanes*), de forma que cada raia represente um ator ou sistema que participa de um conjunto de atividades. O ator pode ser um ser humano, um departamento ou mesmo uma organização completa.

O *processo*, esquematizado no diagrama da Figura 2.2 deve ter uma pseudoatividade inicial representada por um círculo preto e uma pseudoatividade final representada por um círculo preto dentro de outro círculo.

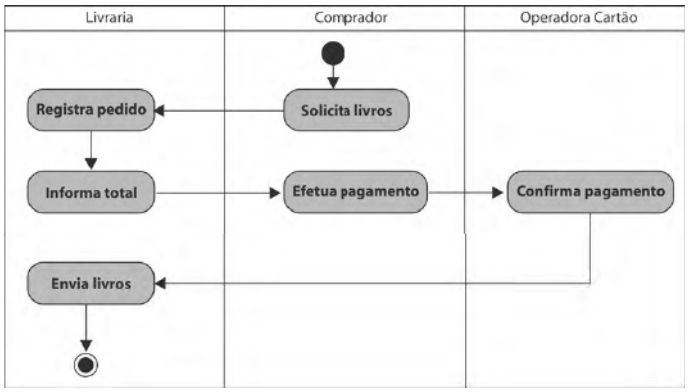


Figura 2.2: Primeira versão de um diagrama de atividades para modelar o processo de venda de livros.

As *atividades* são representadas por figuras oblongas. Quando uma atividade é representada dentro de uma determinada raia, isso significa que o ator ou sistema correspondente àquela raia é o responsável pela sua execução.

Fluxos ou *dependências* entre atividades são representados por setas. Os fluxos normalmente ligam duas atividades indicando precedência entre elas.

Um *caminho* é uma sequência de atividades ligadas por fluxos.

O diagrama de atividades pode ser então usado como ferramenta de visualização do negócio da livraria virtual. A modelagem apresentada na Figura 2.2 mostra uma primeira aproximação do processo de venda de livros. Três entidades participam do processo: a livraria virtual, o comprador (um ator humano) e a empresa de cartão de crédito.

Esse diagrama não tem a intenção de ser um modelo do sistema a ser construído e não deve ser pensado dessa forma. Sua função é ajudar o analista a entender quais são as atividades e os atores envolvidos nos principais processos de negócio da empresa, para que, a partir dessas informações, ele possa efetuar uma captura de requisitos mais eficaz. Assim, cada uma das atividades descritas no diagrama deve estar corretamente encadeada para que o caminho lógico delas seja efetivamente executado. Posteriormente, o analista vai examinar em detalhe como cada uma das atividades deve ser realizada. Se a atividade em questão envolver o sistema a ser desenvolvido, então um levantamento de requisitos mais detalhado deve ser feito referente à atividade.

Duas estruturas de controle de fluxo são usuais nesse diagrama:

a) a estrutura de seleção (*branch* e *merge*), representada por losangos. Do nó *branch* saem fluxos com *condições de guarda* (expressões lógicas entre colchetes). Todos os caminhos devem voltar a se encontrar em um nó *merge*. Dois ou mais fluxos podem sair de uma estrutura de seleção, mas é importante que as condições de guarda sejam mutuamente excludentes, ou seja, exatamente uma delas pode ser verdadeira de cada vez;

b) a estrutura de paralelismo (*fork* e *join*), representada por barras pretas. Caminhos independentes entre os nó *fork* e *join* podem ser executados em paralelo, ou seja, sem dependências entre suas atividades.

O diagrama da Figura 2.3 ainda é um esboço muito rudimentar do real processo de venda de livros. Apenas para ilustrar uma possível evolução desse diagrama, pode-se analisar o que aconteceria se algum dos livros em questão não estivesse em estoque. Seria necessário solicitá-lo a uma das editoras e adicioná-lo ao pedido quando chegasse.

A Figura 2.3 mostra essa situação indicando que, se nem todos os livros estão disponíveis, então, antes de o pedido ser entregue, alguns livros devem ser solicitados às editoras, e apenas após a chegada desses livros é que o pedido é atendido.

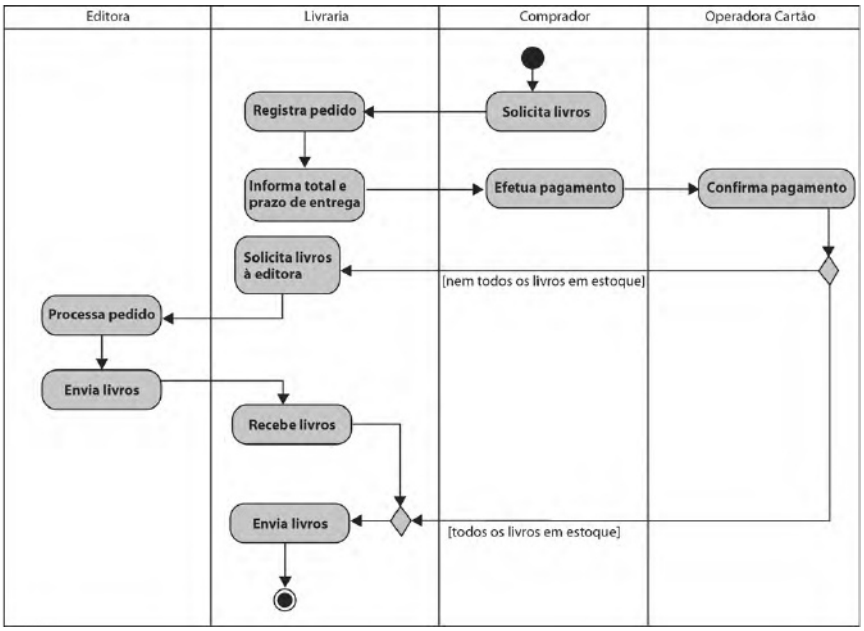


Figura 2.3: O processo de venda considerando a necessidade de comprar livros que não estejam em estoque.

cial a um estado logado se a senha do usuário estiver correta. Graficamente, os eventos são representados nos fluxos por expressões simples e as condições de guarda, por expressões entre colchetes.

Os diagramas de máquina de estado também podem representar ações que são executadas durante uma transição de estado. Por exemplo, a transição ativada pelo evento login e guardada pela condição [senha correta] pode ainda ativar a ação /autorizar acesso, a qual é uma *consequência* da transição (mas não a sua causa nem sua condição). As ações associadas às transições devem ser representadas por expressões iniciadas por uma barra (/).

Para exemplificar, pode-se considerar que o analista interessado em entender melhor o ciclo de vida de um livro na livraria virtual resolve estudar os estados pelos quais ele passa. Assim, ele descobre que, inicialmente, um livro é disponibilizado no catálogo de um fornecedor. A livraria pode então encomendar um conjunto de cópias desse livro. Quando a encomenda chega, o livro é adicionado ao estoque e disponibilizado para venda. Uma vez vendido, o livro é baixado do estoque e vai ao departamento de remessa. Depois de enviado, o livro é considerado definitivamente vendido. Eventualmente, livros enviados podem retornar, por exemplo, em função de endereço incorreto ou ausência de uma pessoa para receber a encomenda. Nesse caso, a livraria entra em contato com o comprador e, conforme for, cancela a venda ou reenvia o material. O livro é considerado definitivamente entregue apenas quando o correio confirma a entrega. Todas essas mudanças de estado (transições) estão representadas na Figura 2.5.

Figura 2.5: Uma primeira modelagem do ciclo de vida de um livro no sistema Livir como máquina de estados.

Porém, o modelo representado na Figura 2.5 ainda é incompleto em relação às possíveis transições de est ados de um livro. Por exemplo, um livro enviado pode retornar danificado devido ao manuseio e, nesse caso, não pode ser reenviado. Isso pode ser representado pela adição de uma *condição de guarda* à transição que vai do estado Devolvido para Enviado, e com a adição de um novo estado final em que o livro é descartado, conforme a Figura 2.6.

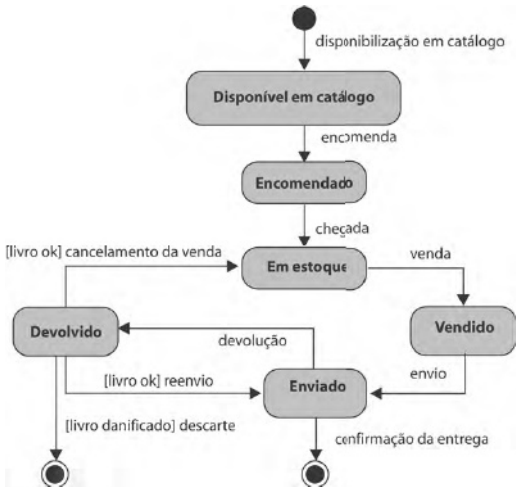


Figura 2.6: Diagrama de máquina de estados com condições de guarda.

Porém, essas condições de guarda ainda são bastante informais. Para ter condições de guarda efetivamente formais seria necessário usar uma linguagem formal como a OCL (*Object Constraint Language*), que será apresentada mais adiante neste livro.

Em algumas situações, é possível que um evento ocorra em mais de um estado, levando a uma transição para outro estado. No exemplo da Figura 2.6 pode-se imaginar que um livro pode ser danificado não apenas a partir do estado Devolvido, mas também a partir dos estados Em estoque e Vendido, pois, estando no depósito da livraria, é possível que venha a ser danificado também. Nos três casos, cabe uma transição para o estado final através do evento descarte. É possível representar um conjunto de transições com o mesmo evento de/ou para o mesmo estado utilizando o conceito de *superestado*.

Na Figura 2.7, qualquer transição de/ou para o superestado No depósito corresponde ao conjunto de transições de cada um de seus três *subestados*. No caso de transições *para* o superestado, é necessário estabelecer entre seus subestados qual seria o inicial. Para isso, basta incluir um pseudoestado inicial dentro do superestado ou fazer as transições diretamente para um dos subestados, como na Figura 2.7.

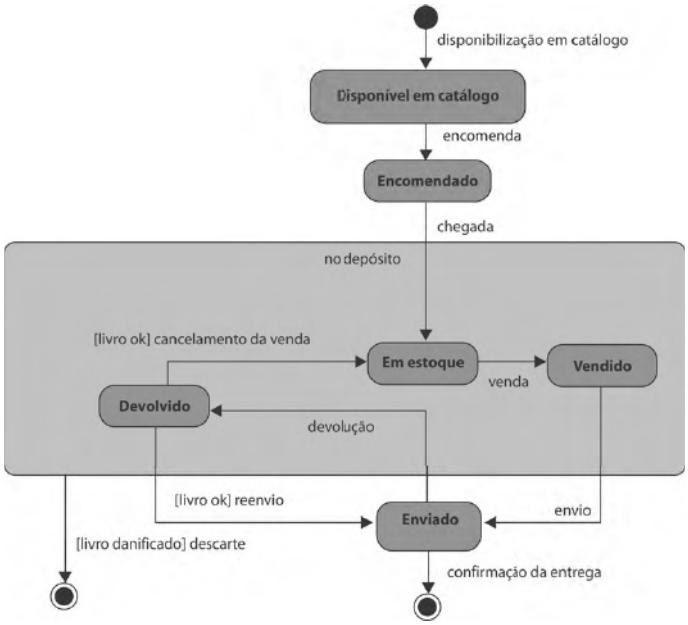


Figura 2.7: Diagrama de máquina de estados com um superestado.

Um objeto pode estar simultaneamente em mais de um estado. Por exemplo, um livro pode estar em oferta ou não desde o momento em que é encomendado até o momento em que seja descartado ou que sua entrega ao comprador seja confirmada. Assim, pode-se modelar um livro com dois estados paralelos: o primeiro estabelece se o livro está em oferta ou não, e o segundo estabelece seu *status* em relação à venda. No diagrama da Figura 2.8 os estados paralelos são representados dentro de *regiões concorrentes* de um superestado. As transições para dentro de regiões concorrentes devem ocorrer através de um nó *fork*, e as transições para fora das regiões concorrentes, através de um nó *join*.

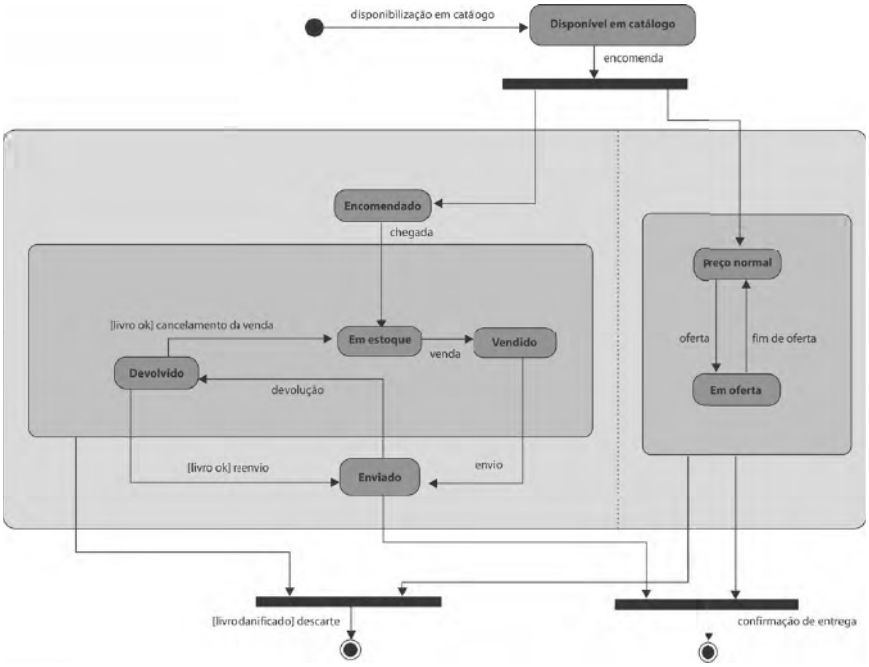


Figura 2.8: Diagrama de máquina de estados com subestados paralelos.

As transições divididas por *fork* e unidas por *join* devem ser rotuladas apenas na parte em que são de fluxo único, ou seja, na entrada, no caso de *fork*, e na saída, no caso de *join*.

Continuando a modelagem, ainda seria possível estabelecer que um livro no subestado vendido, devolvido ou enviado não pode mudar do subestado preço normal para em oferta ou vice-versa. Isso pode ser modelado adicionando-se uma condição de guarda às transições dos subestados preço normal e em oferta: [não está em vendido, devolvido ou enviado].

2.3. Comentários

É importante frisar que o objetivo dessa etapa da análise é obter uma *visão geral do sistema*, e não uma especificação detalhada do seu funcionamento. Então, na maioria dos casos, a preocupação do analista deve se centrar em descobrir informações sobre o sistema e não, ainda, especificar formalmente o seu funcionamento.

Fica a pergunta: para quais elementos do sistema deve-se fazer diagramas de máquina de estados ou de atividades durante essa fase? Não é recomendável criar diagramas para todo e qualquer elemento do futuro sistema porque essa fase demoraria demais e sua objetividade seria prejudicada, já que há pouco conhecimento sobre o sistema para poder realizar tal modelagem. Nesse ponto, é necessário a modelagem de alguns *elementos-chave* para que se possa entender melhor seu funcionamento.

Uma pista para identificar esses elementos-chave é verificar *qual o objeto do negócio*. No caso da livraria, são os livros; no caso de um hotel, são as hospedagens; no caso de um sistema de controle de processos, são os próprios processos. Então, são esses elementos que devem ser modelados para serem mais bem compreendidos nessa fase.

A segunda pergunta seria: devo usar um diagrama de máquina de estados ou um de atividades? A resposta depende da natureza do que vai ser modelado. Observa-se que um estado não comporta necessariamente uma atividade. Uma TV, por exemplo, pode estar no estado desligada quando não está fazendo nada. Um diagrama de atividades é útil quando se trata de pessoas ou sistemas fazendo coisas, como numa linha de produção ou na execução de um processo. Já o diagrama de máquina de estados é mais útil quando a entidade em questão passa por diferentes estados nos quais não está necessariamente realizando alguma atividade.

Além disso, o diagrama de máquina de estados usualmente apresenta diferentes estados de uma única entidade, enquanto o diagrama de atividades apresenta atividades realizadas por um conjunto de pessoas, sistemas ou organizações representados nas *swimlanes*.

Deve-se ter em mente também que os requisitos inevitavelmente mudam durante o desenvolvimento do projeto. Deve-se esperar, então, poder gerenciar a mudança dos requisitos nas demais fases de desenvolvimento.

Outras vezes, os requisitos mudam depois do desenvolvimento. Podem mudar as condições de contexto ou a forma de trabalho ou as políticas da empresa. Embora essas mudanças não possam ser previstas pelo analista, podem ser criados mecanismos que as acomodem ao sistema quando surgirem. Existem padrões de projeto específicos para tratar essas instabilidades do sistema (como, por exemplo, o padrão *Estratégia*).

Essas mudanças são totalmente imprevisíveis. Se o sistema não estiver estruturado para acomodar mudanças nos requisitos, haverá excesso de trabalho para implementá-las.

Esse tipo de situação faz com que os processos de análise e projeto dirigidos por requisitos (Alford, 1991) sejam inadequados para muitos sistemas. Fundamentar a arquitetura de um sistema em seus requisitos é como construir um prédio sobre areia movediça. Quando os requisitos mudam, a estrutura do sistema muda. O UP, porém, propõe que a arquitetura do sistema seja fundamentada em elementos muito mais estáveis: as *classes* (e *componentes*) que encapsulam informação e comportamento.

Essas classes, mais adiante, vão implementar as funcionalidades que, combinadas, permitem a realização dos requisitos. Mudando-se os requisitos, mudam-se as combinações, mas não a estrutura básica. Essa estrutura usualmente segue o *princípio aberto-fechado* (Meyer, 1988), no sentido de que está sempre pronta para funcionar (fechada), mas aberta para incorporar novas funcionalidades.

3.1.3. Requisitos Funcionais

Os requisitos funcionais devem conter basicamente os seguintes elementos:

a) a *descrição* de uma função a ser executada pelo sistema (usualmente entrada, saída ou transformação da informação);
b) a *srcem* do requisito (quem solicitou) e/ou quem vai executar a função em questão (usuário);
c) quais as *informações* que são passadas do sistema para o usuário e vice-versa quando a função for executada;
d) quais *restrições lógicas* (regras de negócio) ou *tecnológicas* se aplicam à função.

Cada requisito funcional deve conter, portanto, uma *função*, que pode ser uma entrada (exemplo, "cadastrar comprador") ou saída (exemplo, "emitir relatório de vendas no mês").

É importante identificar a *srcem* ou *usuário* do requisito, pois sempre é necessário validar os requisitos com essas fontes, verificando se estão bem escritos, completos e consistentes.

Algumas vezes também acontece de pessoas ou departamentos diferentes apresentarem o mesmo requisito de forma discrepante. Nesse caso, é necessário realizar um acordo ou identificar quem teria autoridade para determinar a forma aceitável do requisito.

As *informações de entrada e saída*são importantíssimas para que a análise de requisitos ocorra de forma sistemática. Sem essas informações, os requisitos ficam muito vagos e pouco é aprendido sobre o sistema nessa fase. Dizer simplesmente que "o sistema deve permitir o cadastro de compradores" é muito vago como requisito. O analista deve sempre procurar saber quais movimentos de informação essas funções envolvem. Por exemplo, o cadastro do comprador envolve apenas nome, endereço e telefone? No caso do sistema Livir, não. Deve incluir com certeza dados de um ou mais cartões de crédito, pois serão necessários para efetuar os pagamentos. Essa verificação de informações que entram e saem do sistema é que vai permitir ao analista descobrir a maioria dos conceitos e funções, realizando a pesquisa em extensão no espaço de requisitos para ter uma visão abrangente do todo. No exemplo visto, fica patente, após o detalhamento das informações que entram e saem, a necessidade de definir um requisito funcional para cadastrar cartões de crédito adicionais para o comprador.

3.1.4. Requisitos Não Funcionais

Os requisitos não funcionais aparecem sempre ligados a requisitos funcionais e podem ser basicamente de dois tipos: lógicos ou tecnológicos.

As *restrições lógicas* são as regras de negócio relacionadas à função em questão. Por exemplo, no registro de uma venda, uma série de restrições lógicas poderia ser considerada, como por exemplo: não efetuar a venda caso a operadora de cartão não autorize o pagamento ou não efetuar a venda caso a venda anterior tenha sido cancelada devido a um endereço inválido que ainda não foi corrigido.

As *restrições tecnológicas* dizem respeito à tecnologia para a realização da função, como, por exemplo, a interface (Web, por exemplo), o tipo de protocolo de comunicação, restrições de segurança ou tolerância a falhas etc.

Por exemplo, registrar a venda de um conjunto de livros é um requisito funcional. Estabelecer que o tipo de interface para efetuar uma venda deve seguir um padrão de interface de fluxo sequencial de telas é uma restrição tecnológica (de interface) sobre a forma como essa função é realizada.

Outro exemplo de requisito não funcional seria "a autorização de débito no cartão de crédito não deve levar mais do que 5 segundos". Isso seria uma restrição tecnológica de desempenho e afetaria a forma como o projetista iria pensar no mecanismo de acesso ao sistema da operadora de cartões. Nesse caso, o projeto do sistema teria de considerar seriamente o uso de conexões fixas com as operadoras de cartão em vez de linhas discadas.

3.1.5. Requisitos Suplementares

Os *requisitos suplementares* são todo tipo de restrição tecnológica ou lógica que se aplica ao sistema como um todo e não apenas a funções individuais. Por exemplo, um requisito suplementar pode estabelecer que o sistema deva ser compatível com um banco de dados legado ou ser implementado em uma determinada linguagem de programação, ou ainda, seguir um determinado padrão de interface ou *look and feel*.

Deve-se tomar certo cuidado no enunciado dos requisitos suplementares. Um requisito como "o sistema deve ser fácil de usar" é muito pouco esclarecedor para que possa ser útil. Melhor seria dizer: "o sistema terá ajuda *on-line* em todas as telas e para todas as funções". Isso dá uma ideia mais precisa sobre o que realmente deve ser realizado pelo sistema.

3.1.6. Documento de Requisitos

O *documento de requisitos* registra todos os tópicos relativos ao que o sistema deve fazer e sob quais condições. Esse documento ainda não precisa ser totalmente estruturado, e assume-se que não será completo em profundidade, embora se possa esperar que esteja razoavelmente completo em extensão. Eventuais lacunas desse documento serão preenchidas durante a fase de elaboração.

O documento de requisitos pode ser organizado de forma textual ou na forma de um diagrama (o diagrama de requisitos existente em algumas ferramentas *CASE*, porém, não pertence à especificação da UML).

Os requisitos funcionais podem ainda ser subdivididos em subsistemas, especialmente se a quantidade de funções for muito grande. Essa é uma primeira forma de organização do sistema.

Sugere-se que o documento de requisitos tenha um índice consistindo no nome da função ou requisito suplementar e que o corpo do documento contenha os detalhes mencionados na Seção 3.1.3.

A Figura 3.1 apresenta um exemplo de documento de requisitos para o sistema Livir. Se houvesse subsistemas, eles seriam o primeiro nível de divisão dentro de "Requisitos funcionais", contendo cada divisão um conjunto de requisitos numerados.

Sistema Livir – Documento de Requisitos

Requisitos funcionais

1. Registrar novos títulos a partir do catálogo das editoras.
2. Registrar vendas de livros.
3. Realizar encomendas de livros.
4. Registrar e autorizar pagamentos com cartão de crédito.
5. Registrar e aplicar promoções.
6. Calcular custos de entrega.
7. Emitir relatório de livros mais vendidos.
8. Emitir relatório de compradores mais assíduos.
9. Emitir sugestões de compra para compradores baseadas em compras anteriores.
10. ...

Requisitos suplementares

1. O sistema deve operar via interface Web.
2. Todos os controles de interface devem ter um campo de ajuda associado.
3. ...

Figura 3.1: O índice de um documento de requisitos para o sistema Livir.

O detalhamento de cada requisito (especialmente os funcionais) deve aparecer no corpo do documento. Esse detalhamento deve conter as partes mencionadas na Seção 3.1.3. A Figura 3.2 apresenta um exemplo.

1. Registrar novos títulos a partir do catálogo das editoras

Descrição:

O gerente seleciona as editoras para as quais pretende fazer a atualização. O processo é automático. O sistema consulta os ISBN disponibilizados e os compara com os existentes na base. Havendo novos ISBN, o sistema atualiza a base com as novas informações.

Fontes:

Sr. Fulano de Tal (gerente) e manual técnico da interface de catálogo das editoras.

Usuário:

O próprio gerente.

Informações de entrada:

O gerente informa quais são as editoras para as quais pretende fazer a atualização a partir de uma lista fornecida pelo sistema.

Informações de saída:

- A lista de editoras (nome).
- O relatório de atualizações efetuadas (uma lista contendo: nome da editora, ISBN, título e preço de compra).

Restrições lógicas:

Não há.

Restrições tecnológicas:

1. Deve ser usado o sistema de interface com as editoras, de acordo com o manual XYZ.1234.
2. A seleção feita pelo gerente se dá através de uma lista de seleção múltipla, a qual deve ser ordenada de forma alfabética.
3. ...

Figura 3.2: Detalhamento de um requisito.

Observa-se que o detalhamento das informações que entram e saem do sistema é fundamental para a compreensão mais detalhada dessa função. Através desse detalhamento é que a verdadeira dimensão do sistema vai sendo descoberta.

Sem esse detalhamento, o sistema pode parecer mais simples do que realmente é, o que explica por que, em muitos casos, os analistas esperam desenvolver um sistema em determinado tempo mas levam muito mais tempo, estourando prazos e orçamentos.

3.2. Análise de Requisitos

Na análise de requisitos, o analista vai procurar caracterizar certas propriedades dos requisitos já levantados, conforme as subseções seguintes.

3.2.1. Permanência e Transitoriedade

Uma primeira caracterização considerada importante na análise de requisitos é decidir se determinado requisito não funcional ou suplementar é *permanente* ou *transitório*.

Requisitos não funcionais podem ser considerados permanentes (não vão mudar) ou transitórios (podem mudar) de acordo com uma decisão tomada pelo analista em conjunto com o cliente. O requisito não tem a propriedade de permanência ou transitoriedade por si: ela é decidida de acordo com a conveniência. Um mesmo requisito pode ser considerado permanente ou transitório dependendo do que se queira em relação ao tempo de desenvolvimento ou custo da manutenção do software.

Por exemplo, um requisito suplementar poderia estabelecer que o sistema Livir trabalhe com uma única moeda: o real. Se esse requisito suplementar for considerado *permanente*, todo o sistema será construído de forma que o real seja a única moeda. Se o requisito for considerado *transitório*, o sistema deverá ser construído de forma a poder acomodar futuramente outras moedas ou, ainda, mais de uma moeda de cada vez.

As consequências de decidir que um requisito é permanente são as seguintes:

a) fica mais barato e rápido desenvolver o sistema em si;
b) fica mais caro e demorado mudar o sistema caso o requisito, por algum motivo, mude no futuro (com a implantação de uma nova moeda, por exemplo).

Por outro lado, decidir que um requisito é transitório tem como consequências:

a) fica mais caro e complexo desenvolver o sistema (ele deverá acomodar funcionalidades para a mudança da moeda);

b) fica mais barato e rápido fazer a manutenção no sistema (caso a moeda mude, o sistema já está preparado para acomodar esse fato com uma simples reconfiguração).

Então, a natureza dos requisitos não funcionais não vai decidir se eles são permanentes ou transitórios. O analista é que tem de tomar essa decisão (com o aval do cliente). O ideal seria elencar aqueles requisitos de maior importância (que se espera que possam *mesmo* mudar num futuro próximo e cuja mudança tenha maior *impacto* no sistema) e considerá-los transitórios, deixando os demais como permanentes.

3.2.2. Requisitos Evidentes e Ocultos

Os requisitos funcionais podem ser opcionalmente classificados em evidentes ou ocultos:

a) os *requisitos funcionais evidentes* são funções efetuadas com conhecimento do usuário. Esses requisitos usualmente correspondem a trocas de informação, como consultas e entrada de dados, que ocorrem com o meio exterior através da interface do sistema;

b) os *requisitos funcionais ocultos* são funções efetuadas pelo sistema sem o conhecimento explícito do usuário. Usualmente, são cálculos ou atualizações feitas pelo sistema sem a solicitação explícita do usuário, mas como consequência de outras funções solicitadas por ele.

É importante classificar os requisitos dessa forma porque, posteriormente, eles serão associados aos casos de uso através de *relações de rastreabilidade*. Apenas os requisitos evidentes corresponderão aos passos do caso de uso expandido porque são executados com o conhecimento explícito do usuário. Os requisitos ocultos são executados internamente pelo sistema. Então, embora não apareçam explicitamente nos passos de um caso de uso expandido, esses precisam ser adequadamente associados a aqueles para ser lembrados no momento de projetar e implementar as operações do caso de uso.

Um exemplo de requisito evidente é emitir um relatório de livros mais vendidos por requisição do gerente. Um exemplo de requisito oculto seria aplicar uma política de desconto, se ela existir. Nesse caso, nenhum usuário solicita explicitamente ao sistema para fazer essa aplicação. É uma atividade que o sistema executa independentemente dos usuários e, portanto, de forma oculta.

3.2.3. Requisitos Obrigatórios e Desejados

Os requisitos ainda podem ser classificados em *obrigatórios* e *desejados*, ou seja, aqueles que devem ser obtidos de qualquer maneira e aqueles que podem ser obtidos caso isso não cause maiores transtornos no processo de desenvolvimento.

No caso dos requisitos funcionais, essa classificação indica uma priorização de desenvolvimento. Um bom analista tomaria as funções como base para calcular o tempo de desenvolvimento. Assim, não faria muito sentido falar em funções obrigatórias e desejáveis, mas sim em quanto tempo levaria para desenvolver esse ou aquele conjunto de funções.

Entretanto, nem sempre a coisa funciona assim. No caso de alguns sistemas pode-se querer trabalhar com prioridades, desenvolvendo inicialmente determinadas funções consideradas obrigatórias e, posteriormente (se sobrar tempo), outras funções consideradas desejadas.

Mas os requisitos não funcionais e suplementares são bem mais imprevisíveis do que os funcionais para efeito de estimativa de esforço. Assim, em alguns casos, pode ser necessário efetivamente classificar esses requisitos por graus de prioridade.

Definem-se algumas restrições que devem ser obtidas a qualquer custo e outras que seria desejável obter, desde que isso não extrapole o tempo ou recursos disponibilizados para o projeto.

Por exemplo, no caso do sistema Livir, o requisito de que a interface seja Web poderia ser considerado obrigatório Nesse caso, não se aceita outra coisa. Porém, o acesso através de telefone celular poderiaser um requisito desejável, já que não é absolutamente necessário para o efetivo funcionamento do sistema.

Com a formalização dos contratos de desenvolvimento de software, porém, cada vez menos flexibilidade se tem em relação a requisitos desejáveis. O desenvolvedor deve dizer claramente quais requisitos vai implementar, quanto tempo vai levar e quanto vai custar. Qualquer desvio dessas previsões pode implicar multas ou cancelamento de contratos.

3.2.4. Classificação de Requisitos Não Funcionais e Suplementares

Os requisitos não funcionais e suplementares podem ser classificados por atributo, ou seja, se são requisitos de interface, de implementação, de efi-

ciência, de tolerância a falhas etc. A finalidade principal das classificações de requisitos em categorias é a organização. Algumas sugestões de possíveis categorias são:

a) *usabilidade*: quais fatores humanos estão envolvidos no sistema? Que tipo de ajuda o sistema vai prover? Quais as formas de documentação ou manuais estarão disponíveis? Como esses manuais vão ser produzidos? Que tipo de informação eles vão conter? Seria interessante definir esses tópicos na fase de concepção, visto que o contrato com o cliente deve especificar muitas dessas questões;

b) *confiabilidade*: que tipo de tratamento de falhas o sistema vai ter? O analista não é obrigado a produzir um sistema totalmente tolerante a falhas, mas deve estabelecer que tipo de falhas o sistema será capaz de gerenciar: falta de energia, falha de comunicação, falha na mídia de gravação etc. Não se deve confundir *falha* com erro de programação, visto que erros de programação são elementos que nenhum software deveria conter. As falhas são situações anormais que podem ocorrer mesmo para um software implementado sem nenhum erro de programação;

c) *desempenho*: que tipo de eficiência e precisão o sistema será capaz de apresentar? Pode-se estabelecer, por exemplo, como requisito de eficiência, que nenhuma consulta à base de dados de compradores vai demorar mais de cinco segundos. Na fase de concepção não se define *como* o sistema fará para cumprir o requisito, apenas se diz que de alguma forma ele terá de ser cumprido no projeto. Cabe ao projetista e programador garantir que o requisito seja satisfeito. Se o analista por algum motivo conclui que o requisito dificilmente poderá ser implementado, o requisito passa a ser um *risco* do sistema e eventualmente necessitará de um estudo mais aprofundado ainda na fase de concepção, para verificar a possibilidade de sua realização;

d) *configurabilidade*: o que pode ser configurado no sistema? Deve-se definir os elementos que poderão ser configurados pelo usuário sem que seja necessário recompilar o sistema. Exemplos de itens configuráveis são: o tipo de impressoras, a moeda do país, políticas da empresa, fontes e cores da interface, idioma etc.;

e) *segurança*: quais são os tipos de usuários e que funções cada um pode executar? Sugere-se a implantação de um sistema de permissões, de for-

ma que possam ser criados dinamicamente perfis de usuários com diferentes conjuntos de permissões;

f) *implementação*: qual linguagem deve ser usada? Por que motivo? Que bibliotecas estarão disponíveis? Quais bancos de dados serão acessíveis? Há necessidade de comunicação com sistemas legados?;

g) *interface*: como deve ser a interface? Vai ser seguida alguma norma ergonômica?;

h) *empacotamento*: de que forma o software deve ser entregue ao usuário final?;

i) *legais*: muitas vezes, uma equipe de desenvolvimento deve contar com uma assessoria jurídica para saber se está infringindo direitos autorais ou normas específicas da área para a qual o software está sendo desenvolvido.

Embora essa lista seja extensa, o analista deve ter em mente que se trata apenas de uma forma de classificação para que ele possa mais facilmente avaliar quais requisitos são relevantes para cada um dos tipos listados. Não há necessidade de procurar requisitos que não existem, por exemplo, estabelecendo complicados requisitos de empacotamento para um cliente para o qual não faz a menor diferença a forma como o software será entregue.

Também não se deve perder tempo discutindo se um requisito é desse ou daquele tipo. Mais importante do que classificar é reconhecer que o requisito existe. Esse tipo de discussão, que não acrescenta conhecimento ao estudo do problema, deixa a análise travada, ou seja, não se anda mais para frente.

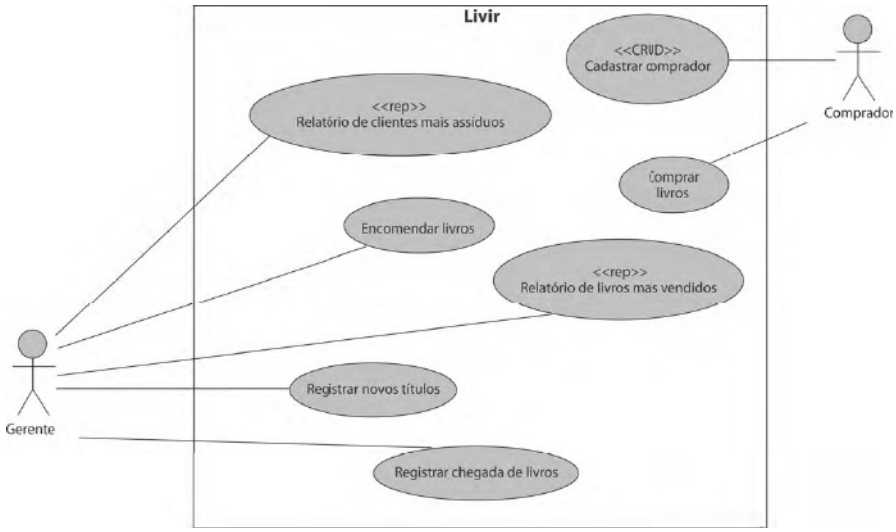


Figura 4.1: Diagrama de casos de uso da UML.

O objetivo de listar os casos de uso é l evantar informações sobre como o sistema interage com possíveis usuários e quais consultas e transformações da informação são necessárias para que processos completos de interação sejam executados. É, portanto, uma forma de sistematizar e organizar os requisitos.

Por exemplo, no caso do sistema Livir, os principais casos de uso (entre outros) são: Comprar livros, Encomendar livros, Registrar novos títulos etc. Funções mais simples (requisitos funcionais), como Calcular custos de entrega e Registrar e autorizar pagamentos com cartão de crédito, possivelmente vão ocorrer apenas como parte dos processos maiores, nunca isoladamente. Casos de uso são processos que podem ocorrer isoladamente. Processos que *só* podem ocorrer juntamente com outros processos são apenas partes de casos de uso, mas não são casos de uso por si.

No caso do sistema Livir, o cálculo de custos de entrega acontecerá durante o processo de venda de livros, nunca como um caso de uso isolado. Já a venda de livros pode ser considerada como um caso de uso porque tem início e fim bem definidos, ocorre em um intervalo de tempo contíguo (sem interrupções) e produz um resultado consistente (a venda registrada).

Cada caso de uso será associado a um conjunto de requisitos funcionais do sistema. Algumas ferramentas *CASE* fornecem recursos para representar essas dependências. Normalmente isso se faz através de relações de *rastreabilidade* ou matriz de relacionamento. Na falta de uma ferramenta desse tipo, basta que o analista procure listar os casos de uso anotando ao lado os códigos dos requisitos funcionais associados. Usualmente, vários requisitos associam-se a um caso de uso, especialmente quando se tratar de um caso de uso complexo. Alguns requisitos, porém, podem estar associados a vários casos de uso. Em alguns casos, também é possível que um requisito corresponda a um único caso de uso e vice-versa.

Para descobrir os casos de uso, deve-se identificar os atores envolvidos com o sistema (funcionários, gerentes, compradores, fornecedores etc.). Após as entrevistas com esses atores, para descobrir seus objetivos, o analista deve descobrir quais os principais processos de negócio de que eles participam. A cada processo possivelmente corresponderá um ou mais casos de uso.

Os casos de uso de alto nível da fase de concepção não têm como intenção definir perfis de segurança para acessar funções do sistema. Portanto, a definição de diferentes atores tem apenas papel ilustrativo.

4.1. Caracterização de Casos de Uso

Existe um diagrama na UML para representar casos de uso e seus atores (Figura 4.1). Nesse diagrama, as elipses representam casos de uso, os bonecos representam atores (usuários) e o retângulo representa a fronteira do sistema ou subsistemas. Não é um dos diagramas mais úteis, mas é bastante popular para mostrar quais funções um sistema efetivamente executa.

Deve-se evitar que o diagrama tenha um conjunto muito grande de elipses, pois, nesse caso, fica inviável compreendê-lo. Assim, deve-se caracterizar muito bem o que são casos de uso para evitar que esse diagrama tenha, por um lado, processos demais, excessivamente detalhados, ou, por outro lado, processos de menos, faltando funcionalidades importantes do sistema.

Via de regra, a solução é representar no diagrama apenas os processos que podem ser executados isoladamente. Processos parciais que são executados necessariamente dentro de outros processos não devem ser representados nesse diagrama.

4.1.1. Monossessão

Um bom caso de uso deve ser *monossessão*. Isso significa que ele deve iniciar e terminar sem ser interrompido. Por exemplo, o registro de uma encomenda de livros é feito em uma única sessão de uso do sistema.

O pedido e a entrega de livros encomendados, embora ocorram necessariamente um após o outro, devem ser considerados casos de uso independentes, pois acontecem em momentos diferentes, havendo uma interrupção da interação com o sistema entre esses dois processos.

Por outro lado, o caso da venda de livros deve ser considerado como um processo ininterrupto. Então, como tratar a situação em que o carrinho de compras é "guardado" pelo comprador para continuar outro dia? Essa situação pode ser pensada assim: o caso de uso de venda de livros pode terminar de duas maneiras alternativas – com a consumação da venda ou com o carrinho sendo guardado. Em uma próxima oportunidade, o caso de uso é iniciado novamente (possivelmente com alguns livros já no carrinho) e novamente pode ser concluído de uma das duas maneiras mencionadas.

4.1.2. Interativo

Um caso de uso também deve ser *interativo*, o que significa que necessariamente deve existir um ator interagindo com o sistema. Processos internos do sistema não são casos de uso.

Por exemplo, "fazer *backup* automático dos dados" não pode ser considerado caso de uso porque é algo que o sistema faz internamente, sem repassar necessariamente informações aos atores ou receber informações deles.

4.1.3. Resultado Consistente

Um caso de uso deve produzir resultado consistente, seja um registro completo produzido ou uma consulta realizada. Ele não pode terminar deixando a informação em estado inconsistente. Por exemplo, um registro de uma venda não pode deixar de identificar o comprador e os livros solicitados, caso contrário a informação ficará inconsistente com as regras de negócio. Não se poderia cobrar o total da venda se não se sabe quais livros foram comprados nem quem os solicitou.

Pode-se pensar assim: somente será um caso de uso um processo completo, no sentido de que um usuário iria ao computador, ligaria o sistema, executaria o processo e em seguida poderia desligar o computador porque o processo estaria completo.

Isso exclui fragmentos como "Calcular custos de entrega" no caso do sistema Livir, porque esses custos são calculados dentro do processo de venda de livros e não como um processo isolado. Isso também exclui operações como *login*, visto que ir ao sistema fazer *login* e em seguida desligar o computador não pode ser visto como um processo completo que produz resultado consistente.

Por outro lado, é possível que casos de uso completos ocorram dentro de outros casos de uso. Por exemplo, o processo de cadastramento de um comprador pode ser considerado um caso de uso completo, e esse processo pode ocorrer dentro do caso de uso de venda de livros quando for a primeira vez que o comprador usa o sistema.

4.2. Complexidade de Casos de Uso

Os casos de uso podem ser classificados de acordo com sua complexidade da seguinte forma:

a) *processos de negócio*. Os principais processos de negócio da empresa que não se encaixam em nenhum dos padrões a seguir possivelmente abarcarão um número considerável de requisitos funcionais. Esses processos são desconhecidos e apresentam alto risco na modelagem de um sistema, pois usualmente são os mais complexos;

b) *CRUD*. A sigla *CRUD* vem do inglês: *Create*, *Retrieve*, *Update* e *Delete*, ou seja, criar, consultar, atualizar e remover, as quatro operações básicas sobre unidades de informação ou conceitos. Assim, em vez de definir cada uma dessas operações como um caso de uso individual, elas devem ser agrupadas em casos de uso do tipo "manter" ou "gerenciar". Essas operações seguem um padrão bem definido, variando apenas as regras de negócio que são específicas do conceito sendo gerenciado ou mantido. Portanto, são casos de uso de médio risco e média complexidade;

c) *relatórios*. Um relatório é um acesso à informação presente no sistema que não altera essa informação (não tem efeito colateral). A consulta (*retrieve*) de CRUD apenas apresenta dados sobre um conceito (p. ex.,

final os relatórios, que vão apenas tabular e totalizar informação que já deve estar disponível.

4.4. Fronteira do Sistema

Uma das decisões que o analista precisa tomar quando está projetando casos de uso é qual a *fronteira do sistema*. Graficamente, a fronteira é apenas um retângulo que aparece no diagrama (ver Figura 4.1), dentro do qual estão os casos de uso e fora do qual estão os atores.

Mas, na prática, decidir sobre essa fronteira nem sempre é tão simples. Um exemplo seria um posto de gasolina onde há um frentista que usa a bomba a pedido do cliente. O ator é o frentista ou o cliente? Ou ambos? O frentista é um ator ou é parte do sistema? Isso o analista deve resolver e permanecer consistente com sua decisão.

Para que um caso de uso seja o mais essencial possível, recomenda-se, porém, que apenas os atores efetivamente interessados no caso de uso sejam considerados. O frentista, no exemplo anterior, é um mero instrumento de ação do cliente. O frentista não tem interesse pessoal no processo de encher o tanque. Ele atua simplesmente como uma ferramenta do sistema ou como parte da tecnologia. Se a bomba fosse operada pelo próprio cliente, como acontece em alguns países, o caso de uso essencial ainda seria o mesmo, pois as mesmas informações seriam trocadas com o sistema, mudando apenas o personagem. Então, nesse caso, recomenda-se que o ator seja o cliente e que o frentista sequer apareça como ator.

Dessa forma, a análise vai produzir casos de uso que são independentes da tecnologia de interface. No caso do sistema Livir, por exemplo, se o ator do caso de uso de venda de livros for apenas o comprador, a descrição do caso de uso pode ser a mesma, quer seja uma livraria virtual ou uma livraria presencial, onde há funcionários atendendo os clientes e operando o sistema.

Alguém poderá perguntar: mas, e se o funcionário deve indicar que foi ele quem fez a venda para receber uma comissão sobre ela? Nesse caso, trata-se de *outro* sistema, com outro caso de uso, que difere do mencionado para o sistema Livir. E, nesse caso, tanto o comprador quanto o funcionário seriam atores com interesse na informação trocada com o sistema.

Capítulo

5

Casos de Uso Expandidos

A fase de *elaboração* do UP comporta as atividades de análise e projeto do sistema. A análise, nessa fase, por sua vez, comporta três subatividades distintas:

a) expansão dos *casos de uso* (capítulo presente) e determinação dos *eventos e respostas de sistema* (Capítulo 6);
b) construção ou refinamento do *modelo conceitual* (Capítulo 7);
c) elaboração dos *contratos das operações e consultas de sistema* (Capítulo 8).

A *expansão dos casos* de uso pode ocorrer em primeiro lugar porque é uma atividade que toma como entradas apenas o caso de uso de alto nível identificado na fase de concepção e o documento de requisitos.

O refinamento do *modelo conceitual* pode ser feito depois disso porque as informações explicitamente trocadas entre o sistema e o mundo externo, conforme a expansão do caso de uso, serão usadas como base para construir e aprimorar o modelo conceitual.

A atividade de *elaboração dos contratos* deve ser realizada por último, já que ela depende da descoberta das operações e consultas de sistema, bem como do modelo conceitual.

A expansão dos casos de uso corresponde ao aprofundamento da análise de requisitos. Já a modelagem conceitual corresponde à análise de domínio

em seus aspectos estáticos. Finalmente, a elaboração dos contratos corresponde à modelagem funcional de domínio, ou seja, ela mostra como a informação é transformada pelo sistema.

Quando se está expandindo um caso de uso de análise, deve-se proceder a um exame detalhado do processo envolvido. Deve-se descrever o caso de uso passo a passo: como ele ocorre e como é a interação entre os atores e o sistema. Deve-se evitar mencionar interfaces ou tecnologia, mas apenas dizer quais informações os atores passam ao sistema e quais informações o sistema passa aos atores.

Essa descrição passo a passo, a princípio, não deve ser estruturada com desvios. Ela deve ser baseada em uma sequência *default*, ou *fluxo principal*, na qual se descreve o que acontece quando tudo dá certo na interação. Esse fluxo também é chamado de "*caminho feliz*", pois nele não se deve prever erros ou exceções.

Depois de descrever o fluxo principal do caso de uso, analisa-se criticamente cada passo e procura-se verificar o que poderia dar errado. A partir da identificação de uma possível exceção, deve-se construir uma descrição de procedimentos para resolver o problema. O caso de uso então passa a possuir *fluxos alternativos*, semelhantes aos *handlers* dos métodos de tratamento de exceções.

Essa descrição do caso de uso na atividade de análise é feita sem considerar a tecnologia de interface. É, portanto, uma descrição *essencial*. Nesse nível da análise, não interessa a forma das interfaces do sistema, mas *quais* informações serão trocadas entre o sistema e o ambiente externo. Deve-se evitar, portanto, mencionar "menus", "janelas" etc. Apenas a *informação* efetivamente trocada deve ser mencionada.

5.1. Caso de Uso Essencial Versus Caso de Uso Real

Todos os casos de uso de análise são do tipo *essencial*. Essencial, nesse contexto, significa que ele é descrito em um nível de discurso em que apenas a "essência" das operações é apresentada, em oposição à sua realização concreta. Em outras palavras, o analista deve descrever "o que" acontece entre o usuário e o sistema, sem, entretanto, informar sobre "como" essa interação ocorre. O analista não deve, portanto, na atividade de análise, tentar descrever a tecno-

logia de interface entre o sistema e o usuário. Isso será feito na atividade de projeto, na qual serão construídos casos de uso *reais*.

Uma dúvida frequente refere-se a *o que* descrever no caso de uso: o sistema atual ou o sistema como vai ficar depois de pronto? Se, no sistema atual, as operações são feitas manualmente e depois serão feitas no computador, qual deve ser a descrição produzida pelo caso de uso? A resposta é: nem uma nem outra. O caso de uso de análise deve descrever a *essência* das operações e não a sua realização concreta. Assim, o analista deve procurar sempre abstrair a tecnologia empregada no processo e se concentrar nas informações trocadas. Em vez de dizer "o funcionário preenche os dados do comprador numa ficha de papel", correspondendo a uma tecnologia manual, empregada normalmente em sistemas não informatizados ou "o comprador preenche seus dados na tela *XYZ*", que corresponde a uma tecnologia informatizada, o analista deve registrar no caso de uso simplesmente "o comprador informa seus dados". Esta última forma é independente de tecnologia ou interface e representa, portanto, a descrição da operação no seu nível essencial.

Porém, recomenda-se também que *sempre* seja deixado explícito quais dados são informados ou recebidos, para maior clareza do caso de uso. Assim, a forma final desse passo seria, por exemplo, "o comprador informa seu nome, CPF, telefone e endereço".

Então, como descrever, por exemplo, o caso de uso "sacar dinheiro" de um caixa automático, sem entrar no nível da tecnologia de interface? Em vez de dizer que o comprador *passa o cartão magnético*, diz-se que ele *informa sua identificação*. Em vez de dizer que o sistema *imprime o extrato*, diz-se apenas que o sistema *apresenta o extrato*. Assim, eliminando as referências à tecnologia, fica-se apenas com a essência das informações. Isso abre caminho para que na atividade de projeto seja possível pensar em diferentes alternativas para implementar as operações e a interface que permitirá realizar um caso de uso.

Cabe ao analista, portanto, estudar os processos correntes da empresa e produzir uma versão essencial deles, correspondendo ao caso de uso expandido essencial. Depois, o projetista vai apresentar uma solução real para essa especificação baseada em uma ou mais tecnologias existentes.

5.2. Fluxo Principal

O *fluxo principal* é a principal seção de um caso de uso expandido. Ele é a descrição do processo quando tudo dá certo, ou seja, quando não ocorre nenhuma exceção. No caso do sistema de caixa automático, seria a situação em que o comprador informa corretamente sua identificação e senha, tem saldo na conta, há dinheiro suficiente na máquina etc. Já os fluxos alternativos, que serão discutidos mais adiante, correspondem ao tratamento das possíveis exceções e variantes identificadas pelo analista.

Uma das grandes dúvidas dos analistas que trabalham com casos de uso costuma ser decidir exatamente o que pode e/ou deve ser colocado como passo dos fluxos de um caso de uso expandido.

De fato, duas pessoas que descrevam o mesmo processo, se não utilizarem um bom padrão, quase sempre vão gerar uma sequência de passos diferente. Em uma livraria tradicional, um analista poderia fazer constar um passo em que o funcionário pergunta o nome do cliente. Outro analista poderia excluir esse passo e dizer que o cliente simplesmente chega ao balcão e se identifica.

A pergunta é: qual das duas abordagens está correta? Se ambas estão corretas, existem descrições incorretas?

Ambas estão corretas, mas uma delas é mais útil. O objetivo do padrão é incentivar os analistas a construírem versões corretas e semelhantes entre si, além de evitar as versões incorretas. Os conceitos de passos obrigatórios, complementares e impróprios existem para ajudar os analistas a estabelecerem essas distinções.

Todo caso de uso tem passos que são *obrigatórios*. Esses passos envolvem informações que passam dos atores para o sistema e do sistema para os atores; sem essas trocas de informação, o caso de uso não faz sentido ou não pode prosseguir. Esses passos devem constar em qualquer caso de uso expandido, e a falta deles faz com que o caso de uso esteja incorretamente descrito.

Outros passos, como "perguntar o nome do comprador" são opcionais ou *complementares*. Eles servem para contextualizar o caso de uso, mas não são fundamentais porque não passam nenhuma informação para dentro ou para fora do sistema.

Além disso, deve-se ter em mente que, como o caso de uso é uma descrição da interação entre os atores e o sistema, deve-se evitar descrever quaisquer processos internos que ocorram no sistema, como "o sistema armazena a informação no banco de dados". Esses passos são considerados como *impróprios* ou não recomendados para a descrição do caso de uso.

5.2.1. Passos Obrigatórios

Na categoria de passos *obrigatórios* estão incluídos todos os passos que indicam de alguma forma que foi trocada informação entre o sistema e um ou mais atores (usuários), conforme ilustrado na Figura 5.1.

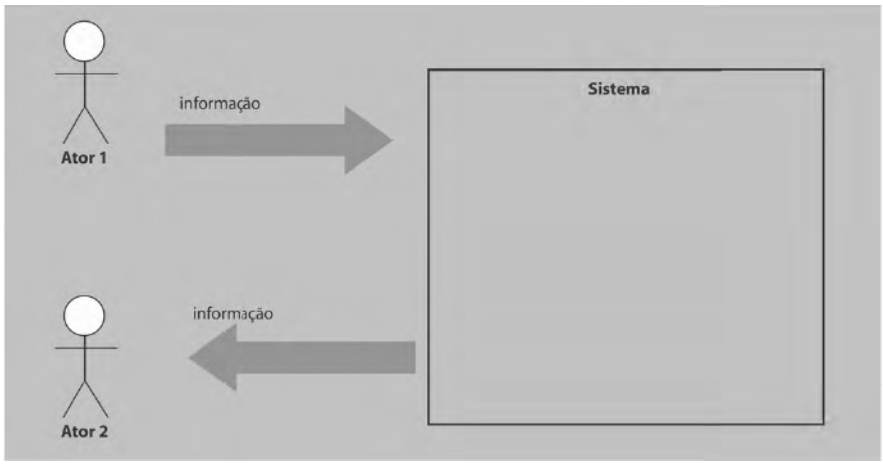


Figura 5.1: Passos obrigatórios implicam trocas de informação entre o sistema e os atores.

Assim, um caso de uso expandido para comprar livros deve obrigatoriamente conter os passos que indicam que o comprador se identifica, identifica os livros que deseja comprar, e o sistema informa o valor dos livros. Por que esses passos são obrigatórios? Porque sem essas informações nenhum sistema seria capaz de registrar adequadamente uma venda. Certamente não seria de grande ajuda um sistema que registrasse uma venda sem indicar quem foi o comprador ou quais foram os livros vendidos. Visto que essas informações são tão importantes para a conclusão do caso de uso, os passos da interação em que os atores passam essa informação ao sistema devem ser obrigatoriamente citados no caso de uso expandido. Também não seria

aceitável se o sistema não informasse o valor dos livros, pois, nesse caso, o comprador poderia pagar aquilo que bem entendesse, já que não haveria o "contrato" de venda.

Em uma descrição de caso de uso, a informação não surge do nada. Ela é transmitida dos atores para o sistema e vice-versa. A ausência de passos de interação que registrem essa troca de informação das fontes para os destinatários deixa os casos de uso sem sentido. A Figura 5.2 mostra um exemplo em que um caso de uso foi mal construído porque uma informação obrigatória foi omitida.

Caso de Uso (mal construído): Comprar Livros

1. O comprador informa seu CPF.
2. O sistema confirma a venda informando o valor total.

Figura 5.2: Um caso de uso em que falta pelo menos um passo obrigatório.

Esse caso de uso está incompleto porque uma venda necessitaria de mais informações do que as que foram trocadas entre o comprador e o sistema. Como o sistema poderia saber quais os livros a serem comprados se o comprador, que é quem detém essa informação, não a transmitiu?

Por outro lado, como o comprador poderia saber quais livros podem ser comprados se o sistema não lhe apresentar as possibilidades? Então, deve também haver um passo em que o sistema apresenta os livros que podem ser comprados, e o comprador seleciona um ou mais dentre eles.

Uma versão melhor desse caso de uso é descrita na Figura 5.3.

Caso de Uso: Comprar livros

1. O comprador informa sua identificação.
2. O sistema informa os livros disponíveis para venda (título, capa e preço).
3. O comprador seleciona os livros que deseja comprar.
4. O sistema informa o valor total dos livros e apresenta as opções de endereço cadastradas.

sistema para os atores. Essa informação deve ser algo que os atores, a princípio, não tenham ou não possam inferir necessariamente sem consultar o sistema.

Por exemplo, o sistema tem de informar o valor total no caso de uso "Comprar livros", caso contrário o comprador não saberá quanto vai pagar. Essa informação até poderia ser calculada pelo comprador se ele somasse os valores individuais de cada livro, mas o comprador não poderia ter certeza sobre quanto seria cobrado, devido a descontos e frete, sem que a livraria explicitamente o informasse disso; a responsabilidade de fornecer essa informação é do *sistema*.

Por outro lado, quando um comprador envia dados ao sistema, a interface pode emitir algum tipo de *feedback* para informar que os dados foram recebidos e corretamente processados. Mas esse retorno (normalmente uma mensagem do tipo "*ok*" ou "operação confirmada") não constitui nova informação sobre livros, cartões ou endereços. É apenas um *feedback* de interface e, portanto, refere-se à tecnologia usada. Não sendo uma informação propriamente dita, não deve ser considerada como passo obrigatório.

Será interessante, para efei to de identificação de operaçõ es e consultas de sistema, que os passos do caso de uso que correspondem a eventos e respostas sejam claramente marcados. Sugere-se o marcador [IN] para eventos de sistema e [OUT] para respostas de sistema. Esses marcadores podem ser colocados logo após o número da linha do passo no caso de uso, como na Figura 5.4.

Caso de Uso: Comprar livros

1. [IN] O comprador informa sua identificação.
2. [OUT] O sistema informa os livros disponíveis para venda (título, capa e preço).
3. [IN] O cliente seleciona os livros que deseja comprar.
4. [OUT] O sistema informa o valor total dos livros e apresenta as opções de endereço cadastradas.
5. [IN] O cliente seleciona um endereço para entrega.
6. [OUT] O sistema informa o valor do frete e total geral, bem como a lista de cartões de crédito já cadastrados para pagamento.
7. [IN] O cliente seleciona um cartão de crédito.

4		Seleciona um cartão de crédito	Envia os dados do cartão e valor da venda para a operadora
5	Informa ao sistema o código de autorização da venda		Informa ao comprador o prazo de entrega

Figura 5.5: Um caso de uso em múltiplas colunas.

Nesse tipo de notação, fica mais claro que o sistema apenas reage às ações dos atores. Inclusive o sistema da operadora de cartão, que aqui é considerado um ator, age de forma autônoma em relação ao sistema Livir.

5.2.2. Passos Complementares

A segunda categoria de passos citada é a dos passos *complementares*, a qual consiste em todos aqueles passos que não apresentam informações trocadas entre o sistema e os atores, mas que ajudam a entender o contexto do caso de uso.

Esse tipo de passo corresponde, normalmente, à comunicação entre os atores (comunicação que não envolve o sistema, conforme ilustrado na Figura 5.6) ou à descrição de suas ações ou atitudes, que também não se configuram como envio de informação para o sistema, como "o comprador acessa o site da livraria", "o comprador decide se compra os itens", "o sistema pede ao comprador que se identifique" ou "o sistema informa que a venda foi concluída com sucesso".

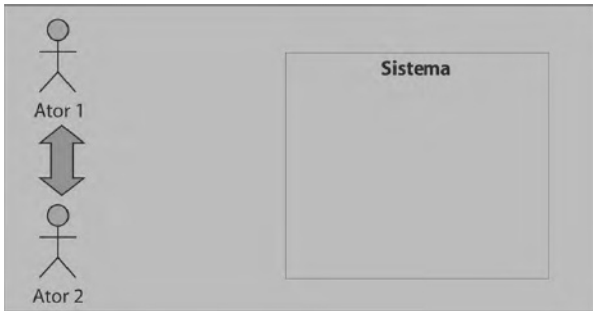


Figura 5.6: Passos complementares em um caso de uso referem-se a ações dos atores que não afetam o sistema.

Os passos complementares não são fundamentais no caso de uso essencial porque não correspondem a eventos nem respostas do sistema, já que não passam informação através da fronteira do sistema.

Alguns deles até poderão, na atividade de projeto, ser mapeados em operações de navegação na interface. Por exemplo, um caso de uso real de projeto poderia ter um passo como "o comprador seleciona a opção 'iniciar compra' no menu de opções". Essa linha não seria necessariamente uma operação do sistema, pois, nesse momento, o comprador ainda não passou nenhuma informação ao sistema (como nome, CPF, título de livro etc.). A ação consiste, então, apenas em uma mudança de estado, que possivelmente corresponderá a uma navegação na interface (uma nova tela é aberta para que o comprador possa iniciar sua compra, por exemplo).

5.2.3. Passos Impróprios

A terceira categoria de passos refere-se aos passos *impróprios*, ou não recomendados. Nessa categoria incluem-se todos os processos considerados internos ao sistema (Figura 5.7).

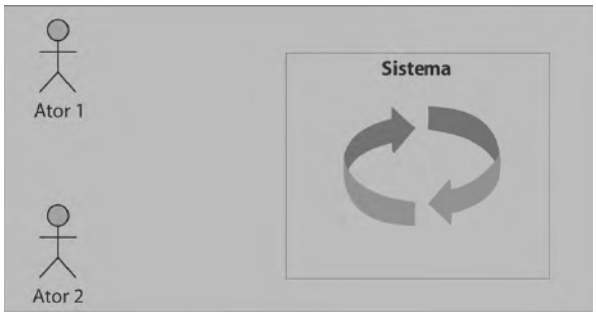


Figura 5.7: Passos não recomendados em um caso de uso são aqueles que descrevem processamento interno do sistema.

O caso de uso é uma ferramenta para descrever ainteração entre usuários e um sistema. Então, não é com esta ferramenta que se deve descrever o processamento interno do sistema.

Esses aspectos serão mais bem descritos na atividade de projeto com ferramentas adequadas (diagramas de comunicação ou sequência). Assim, não seria recomendado, por exemplo, a construção de um caso de uso que tivesse passos como "o sistema registra o nome do comprador no banco de dados" ou "o sistema calculaa média das vendas", pois esses passos correspondem a processamento interno. Porém, um passo como "o sistema *apresenta* a

O caso de uso contém todos os passos obrigatórios que deveriam existir, mas acrescenta dois passos impróprios (passos 3 e 6), que correspondem a processos internos e não a envio ou recebimento de informações.

O fato de que o sistema "calcula a média mensal" não afeta a interação com o usuário. Isso é processamento interno. Ao usuário interessa apenas saber que essa média será apresentada quando necessário. A forma como o sistema vai processar isso é um problema do sistema, não do usuário. Portanto, essa questão será definida mais adiante, nas atividades de projeto e não na atividade de análise.

5.3. Estilos de Escrita

O estilo de escrita dos passos de um caso de uso deve, sempre que possível, seguir um padrão do tipo "ator informa.../sistema informa...". Evita-se escrever "o sistema solicita..." porque se deseja representar, nos passos, apenas fluxos de informação e não as eventuais solicitações que deram srcem a esses fluxos, já que estas nem sempre existem, sendo passos *complementares* e não obrigatórios.

Deve-se tomar cuidado para não colocar, no fluxo principal do caso de uso, testes para verificar exceções. Evita-se, portanto, o uso de seletores como "*se* o usuário está com o cadastro em dia, *então* o sistema apresenta...". Esse tipo de teste é desnecessário porque, como será visto adiante, os tratadores de exceção já estarão associados aos passos do fluxo principal. Então, se uma exceção como essa puder ocorrer, isso estará explícito no fluxo alternativo que corresponde ao tratador da exceção.

Evita-se colocar esses testes no fluxo principal para que ele não acabe se transformando em um complexo fluxograma em vez de uma lista simples de passos. Muitos "se/então" em um fluxo poderão deixá-lo obscuro, e o analista poderá não saber mais qual é exatamente a situação *esperada* (caminho feliz) do processo.

A única situação na qual um teste seria aceitável no fluxo é quando se tratar de um passo opcional, como, por exemplo, "se o usuário desejar, informa também seu celular".

Outra situação a ser observada é evitar, sempre que não houver justificativa, a inclusão de eventos de sistema em sequência ([IN] seguidos de [IN]) ou respostas de sistema em sequência ([OUT] seguidas de [OUT]). A ideia, nesse caso, é normalizar a quantidade de passos que diferentes analistas poderiam atribuir a um caso de uso. Sem essa regra, um analista poderia escrever:

```
1. [IN] O comprador informa seu nome, CPF e telefone.
```

E outro analista poderia escrever:

1. `[IN] O comprador informa seu nome.`
2. `[IN] O comprador informa seu CPF.`
3. `[IN] O comprador informa seu telefone.`

A opção mais útil e correta sob esse ponto e vista é a primeira, até porque, no segundo caso, a sequência estrita exigiria que cada uma das informações fosse apresentada na ordem dos passos; uma vez que uma informação entrou, não se pode mais voltar atrás, a não ser que alguma estrutura explícita permita isso. Então, a primeira opção é mais compatível com a realidade da maioria dos sistemas de informação, que apresentam interfaces em que várias informações podem ser passadas de uma única vez e editadas enquanto não forem definitivamente repassadas ao sistema.

Justificam-se dois passos [IN] em sequência apenas se o primeiro passo puder causar uma exceção que, se ocorrer, deve impedir a execução do segundo passo. Um exemplo disso seria:

1. `[IN] O comprador informa seu CPF.`
2. `[IN] O comprador informa o número, validade e bandeira de seu cartão de crédito.`

Nesse exemplo, quando o comprador informa o CPF pode haver uma exceção, caso o CPF informado esteja incorreto ou se não houver ainda cadastro desse comprador no sistema. Então, a informação dos dados do cartão de crédito deve ser postergada até que o comprador se cadastre. Esse cadastramento pode ser feito na sequência alternativa que trata essa exceção, como será visto na seção seguinte.

5.4. Tratamento de Exceções em Casos de Uso

Depois de descrever o fluxo principal do caso de uso, o analista deve imaginar o que poderia dar errado em cada um dos passos descritos, gerando, dessa forma, os fluxos alternativos que tratam as exceções.

Uma *exceção* (no sentido usado em computação) não é necessariamente um evento que ocorra muito raramente, mas sim um evento que, se não for devidamente tratado, impede o prosseguimento do caso de uso.

Por exemplo, quando uma pessoa vai pagar uma conta, ela pode usar cheque, cartão ou dinheiro. Mesmo que apenas 1% das contas sejam recebidas em dinheiro contra 99% sendo pagas em cheque ou cartão, isso não significa

a) *voltar ao início do caso de uso*, o que não é muito comum nem muito prático, na maioria das vezes, a não ser em sistemas que precisam receber uma sequência de dados em tempo real;
b) *retornar ao início do passo que causou a exceção*: executá-lo novamente, o que é mais comum. Deve-se optar por essa forma quando o passo que causou a exceção eventualmente causar outras exceções diferentes, mesmo que uma delas já tenha sido tratada;
c) *avançar para algum passo posterior*: Isso pode ser feito quando as ações corretivas realizam a operação que o passo ou a sequência de passos posterior deveria ter executado. Porém, deve-se verificar se novas exceções não poderiam ainda ocorrer no passo do fluxo anterior que srcinou a exceção;
d) *abortar o caso de uso*: Nesse caso, não se retorna ao fluxo principal. O caso de uso não atinge seus objetivos. Se for necessário fazer alguma ação corretiva no sentido de desfazer registros intermediários, isso deve ser indicado nos passos do fluxo alternativo (essa forma de tratar uma exceção é conhecida como "pânico organizado").

No caso de uso "Comprar livros" (Figura 5.9), no passo 1, quando o comprador informa o seu identificador (CPF), o sistema pode verificar que ele não possui cadastro (exceção 1a). Nessa eventualidade, o caso de uso não pode prosseguir a não ser que as ações corretivas sejam executadas.

Como foi dito, não se deve colocar essa verificação como uma condicional no fluxo principal (por exemplo, não se deve escrever "2. *Se* o comprador possui cadastro, *então* o sistema informa os livros disponíveis..."), mas como um fluxo alternativo.

Para cada exceção, deve-se tentar identificar possíveis ações corretivas. Não havendo possibilidade ou interesse em efetuar ações corretivas por parte do ator, o caso de uso será abortado.

Exceções genéricas, que podem ocorrer em qualquer passo de qualquer caso de uso, indiferentemente, como, por exemplo, o ator cancelar o processo ou faltar energia no sistema, não devem ser tratadas como exceções nos passos. Esse tipo de situação é tratada através de mecanismos genéricos. No caso do cancelamento, o sistema deverá ter um mecanismo de *rollback* geral.

Considerar essas exceções genéricas em cada passo de um caso de uso criaria um conjunto enorme de informações cujo resultado final é inócuo. Se, em qualquer passo de qualquer transação, o usuário pode efetuar um cancela-

será a continuidade do caso de uso. Isso se traduzirá, na atividade de projeto de interface, em uma operação de controle de navegação e não em uma operação de sistema.

5.6. Casos de Uso Incluídos

Pode ser possível também que dois casos de uso ou mais tenham partes coincidentes. Por exemplo, vários casos de uso podem comportar uma subsequência de pagamento ou um caso de uso pode incluir outro caso de uso completo, como é o caso do cadastramento do comprador, que deve ser feito como sequência alternativa no caso de uso "Comprar livros", caso o comprador não tenha cadastro.

Casos de uso completos podem então ser referenciados dessa forma. No caso dos CRUD, convém informar qual a operação indicada, quando for o caso. Por exemplo, o passo 1a.1 na Figura 5.10 poderia ser escrito como na Figura 5.11.

```
1a.1 Inclui <<CRUD>> Gerenciar Comprador.
```

Figura 5.11: Redação alternativa para um dos passos do caso de uso da Figura 5.10.

Nesse caso, apenas a operação de inserção do caso de uso CRUD pode ser executada para que o comprador insira seus dados.

Tratadores de exceção também podem ser referenciados dessa forma, desde que a ação de finalização seja coerente.

Quando um caso de uso inclui outro, pode-se relacioná-los com a associação de dependência estereotipada por <<include>>, como na Figura 5.12.

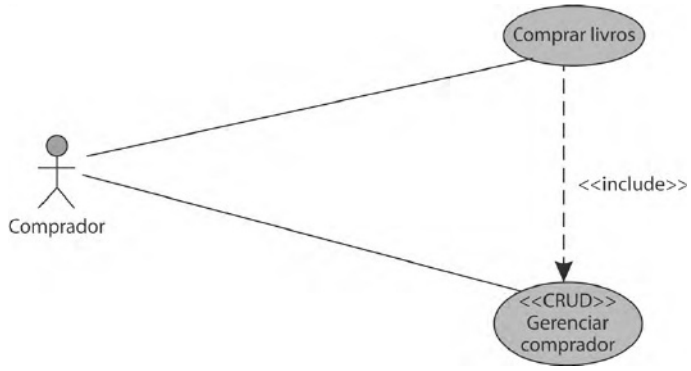


Figura 5.12: Um caso de uso que inclui outro.

Porém, as variantes e tratadores de exceção não têm o *status* de caso de uso, embora sejam representadas pelo mesmo símbolo nos diagramas de caso de uso da UML. Eles não são casos de uso porque não são processos completos, mas partes de outro processo. A rigor, nem deveriam aparecer nos diagramas de caso de uso de análise, mas se for absolutamente necessário mencioná-las, pode-se estereotipar as variantes com <<variant>> e os tratadores de exceção com <<exception>>, para que fique claro que não são processos autônomos, mas apenas partes reusáveis de outros processos.

Deve-se ter em conta, sempre, que o objetivo do caso de uso expandido na análise é o estudo do problema de interação dos atores com o sistema e não a estruturação de um algoritmo para descrever essa estrutura.

5.7. Cenários e Casos de Uso

Um caso de uso pode ser compreendido como uma descrição ou especificação geral que comporta um conjunto de diferentes *cenários*. Cada cenário é uma realização particular ou instância do caso de uso. Usualmente, considera-se que o caso de uso comporta um cenário principal (fluxo principal) e cenários alternativos. Porém, a noção de variantes de fluxo principal normalmente dá margem a dúvidas sobre o que deveria realmente ser um caso de uso. Por exemplo, existe um caso de uso "Comprar livros" com duas variantes em relação à finalização (guardar carrinho ou pagar compra) ou existem dois casos de uso?

Na verdade, o que importa nessa fase da análise é descobrir quais são as informações trocadas entre os atores e o sistema. Assim, não faz muita diferença em relação ao resultado final da análise se as operações são descobertas ou descritas em um único caso de uso com dois cenários alternativos ou em dois casos de uso com um único cenário cada.

A vantagem de unir cenários variantes em um único caso de uso é que, assim, não se precisa repetir a descrição daqueles passos que são coincidentes nos diferentes cenários. É a mesma vantagem que se tem ao fatorar as propriedades de suas classes em uma superclasse pelo uso de herança (Capítulo 7).

Porém, antes de decidir pela criação de casos de uso com variantes a partir da união de dois cenários semelhantes, o analista deve verificar se as sequências variantes efetivamente apresentam passos obrigatórios. Apenas as variantes que contêm passos obrigatórios devem ser consideradas.

Variante 1.1: Inserir

1.1.1 O usuário informa: ...

Variante 1.2: Consultar

1.2.1 O sistema apresenta uma lista de ....

1.2.2 O usuário seleciona um elemento da lista.

1.2.3 O sistema apresenta ... do elemento selecionado.

Variante 1.3: Alterar

1.3.1 Inclui Variante 1.2

1.3.2 O usuário informa novos valores para ...

Variante 1.4: Excluir

1.4.1 O sistema apresenta uma lista de ...

1.4.2 O usuário seleciona um elemento da lista para excluir.

Exceção 1.1.1a Inclusão fere regra de negócio.

1.1.1a.1 O sistema informa a regra que impede a inclusão.

1.1.1a.2 Retorna ao passo 1.1.1 informando novos dados.

Exceção 1.3.2a Alteração fere regra de negócio.

1.3.21a.1 O sistema informa a regra que impede a alteração.

1.3.21a.2 Retorna ao passo 1.3.2 informando novos dados.

Exceção 1.4.2a Exclusão fere regra estrutural ou de negócio.

1.4.2a.1 O sistema informa a regra que impede a exclusão.

1.4.2a.2 Retorna ao passo 1.4.2 para selecionar um novo elemento.

Figura 5.15: Um *template* para CRUD expandido.

Observa-se nesse *template* que a variante 1.3 inclui os passos da variante 1.2. Isso significa que a variante 1.3 na verdade tem quatro passos, identificados no *template* por 1.2.1, 1.2.2, 1.2.3 e 1.3.2, ou seja, o passo 1.3.1 expande-se em 1.2.1, 1.2.2 e 1.2.3.

A forma de tratamento da exceção 1.4.2a (exclusão) abordada aqui é a de impedir que a ação seja executada. No Capítulo 8 esse tema será retomado, e outras duas abordagens possíveis para tratar a exceção de exclusão serão discutidas.

A Figura 5.16 apresenta um exemplo concreto de caso de uso CRUD expandido.

Caso de Uso: <<CRUD>> Gerenciar comprador.

1. O usuário escolhe a operação:
   1.1 Variante "inserir".
   1.2 Variante "consultar".
   1.3 Variante "alterar".
   1.4 Variante "excluir".

Variante 1.1: Inserir

1.1.1 O usuário informa: nome, CPF, endereço e telefone do comprador.

Variante 1.2: Consultar

1.2.1 O sistema apresenta uma lista de CPF e nome ordenada pelo nome.

1.2.2 O usuário seleciona um elemento da lista.

1.2.3 O sistema apresenta nome, CPF, endereço e telefone do comprador selecionado.

Variante 1.3: Alterar

1.3.1 Inclui Variante 1.2

1.3.2 O usuário informa novos valores para nome, CPF, endereço e telefone.

Variante 1.4: Excluir

1.4.1 O sistema apresenta uma lista de CPF e nome ordenada pelo nome.

1.4.2 O usuário seleciona um elemento da lista para excluir.

Exceção 1.1.1a e 1.3.2a CPF já cadastrado

1.1.1a.1 O sistema informa que o CPF já está cadastrado.

1.1.1a.2 Retorna ao passo 1.

Exceção 1.4.2a O comprador tem compras cadastradas em seu nome.

1.4.2a.1 O sistema informa que é impossível excluir o comprador, pois ele já tem compras em seu nome.

1.4.2a.2 O caso de uso é abortado.

Figura 5.16: Um caso de uso CRUD expandido.

5.9.3. Precondições

Por definição, precondições são fatos considerados verdadeiros antes do início do caso de uso. Não se deve confundir as precondições com as exceções, visto que estas últimas não são necessariamente verdadeiras antes do início do caso de uso. As exceções podem ocorrer durante a execução justamente porque não se pode garantir que elas não ocorram. Não é possível, por exemplo, garantir que o comprador terá dinheiro para pagar a dívida antes de iniciar o caso de uso. Portanto, isso é uma exceção. Entretanto, é possível assumir que um comprador não poderá em hipótese alguma comprar um livro que a livraria não tenha disponibilizado no catálogo. Essa disponibilização pode ser então considerada como uma precondição.

Como as pre-condições são dadas como verdadeiras antes do início do caso de uso, resulta que elas não serão testadas durante a execução do caso de uso. Ou seja, as precondições, dessa forma, não gerariam exceções. Simplesmente seria impossível iniciar o caso de uso se a precondição fosse falsa.

5.9.4. Pós-condições de Sucesso

As pós-condições estabelecem normalmente os resultados do caso de uso, ou seja, o que será verdadeiro após sua execução. Por exemplo, o caso de uso "Comprar livros" pode ter como pós-condições os seguintes resultados: "foi criado um registro da venda dos livros para o comprador" e "o setor de entregas foi notificado da venda".

Os resultados de um caso de uso podem ser bem variados em sua natureza, o que difere bastante das pós-condições de operações de sistema, que serão estudadas no Capítulo 7, e que são bem mais formais.

5.9.5. Requisitos Correlacionados

Quando a análise produz um documento estruturado de requisitos (iniciado normalmente na fase de concepção e incrementado ao longo da fase de elaboração), pode ser útil correlacionar esses requisitos aos casos de uso.

A correlação entre requisitos e casos de uso permite ao analista perceber se ainda existem requisitos não abordados.

Para simplificar o processo de associar um requisito a um caso de uso, usualmente coloca-se o código alfanumérico de cada requisito na seção cor-

respondente do caso de uso ou usam-se relações de rastreabilidade (setas tracejadas com o estereótipo <<trace>>).

5.9.6. Variações Tecnológicas

Um caso de uso de análise deve ser descrito no nível essencial e, portanto, não deve tratar de aspectos tecnológicos. Porém, algumas vezes, pode ser interessante registrar, para a atividade de projeto, possíveis variações tecnológicas que poderiam ser utilizadas para realizar o caso de uso.

Por exemplo, o passo do caso de uso Comprar livros, que corresponde à identificação do comprador, pode ter como variações tecnológicas a digitação do CPF ou do nome do comprador em um campo apropriado ou outro código qualquer. Se essas possibilidades estiverem sendo consideradas para o desenvolvimento do sistema, então podem ser listadas na seção "variações tecnológicas".

5.9.7. Questões em Aberto

Muitas vezes, o analista, trabalhando sem a presença do cliente, não sabe como decidir sobre determinado assunto que pode depender de políticas da empresa. Por exemplo, se o usuário pode pagar a dívida a prazo ou se existem promoções para usuários que compram certa quantidade de livros, e assim por diante.

Essas dúvidas devem ser documentadas na seção "questões em aberto" para serem resolvidas no momento em que o cliente estiver disponível. No final da atividade de análise, espera-se que todas as questões em aberto tenham sido resolvidas e incorporadas à descrição do caso de uso expandido.

Capítulo

6

Diagramas de Sequência de Sistema

Na atividade de análise, o texto dos casos de uso expandidos terá basicamente duas utilizações:

a) como fonte de informação para encontrar conceitos para o modelo conceitual (Capítulo 7);
b) como fonte de informação para encontrar as *operações e consultas de sistema*, que darão srcem aos métodos que fazem a interface do sistema com o mundo externo (Capítulo 8).

Operações de sistema são métodos que são ativados a partir de um *evento de sistema*, ou seja, como resposta a uma ação de um usuário. As operações de sistema, por definição, indicam um fluxo de informações do exterior para o interior do sistema e, portanto, de alguma forma, elas alteram as informações gerenciadas pelo sistema.

Consultas de sistema são métodos que correspondem à simples verificação de informação já armazenada. Essa informação pode ser apresentada exatamente como está ou modificada pela aplicação de funções (p. ex., média, total etc.). Mas, por definição, uma consulta de sistema não deve ser responsável por inserir, remover ou alterar informações armazenadas.

Essa separação entre consulta e operação é um princípio antigo em engenharia de software (Meyer, 1988) e justifica-se por permitir melhor reusabi-

lidade do código. Uma consulta que altera dados é menos coesa do que uma consulta sem efeitos colaterais e uma operação que não retorna dados.

Pode-se definir que as operações e consultas de sistema, em conjunto, correspondem à totalidade das funções possíveis do sistema, ou seja, à funcionalidade efetiva total do sistema.

Os casos de uso são excelentes fontes para encontrar operações e consultas de sistema. Nos casos de uso encontram-se operações de sistema a partir da observação de ações do usuário que produzem modificações no sistema (possivelmente estarão marcadas com [IN]), ou seja, as ações que levam informação dos atores para o sistema. Já as consultas de sistema são identificadas por passos que trazem informação do sistema para os atores (possivelmente marcadas por [OUT]).

6.1. Elementos do Diagrama de Sequência

A UML possui um diagrama que pode ser útil para representar a sequência dos eventos do sistema em um cenário de um caso de uso (Figura 6.1). O diagrama de sequência tem como elementos *instâncias de atores*, representados por figuras humanas esquematizadas, e instâncias que representam elementos do sistema. Nessa primeira versão do diagrama, apenas a interface do sistema (camada de aplicação) estará representada.

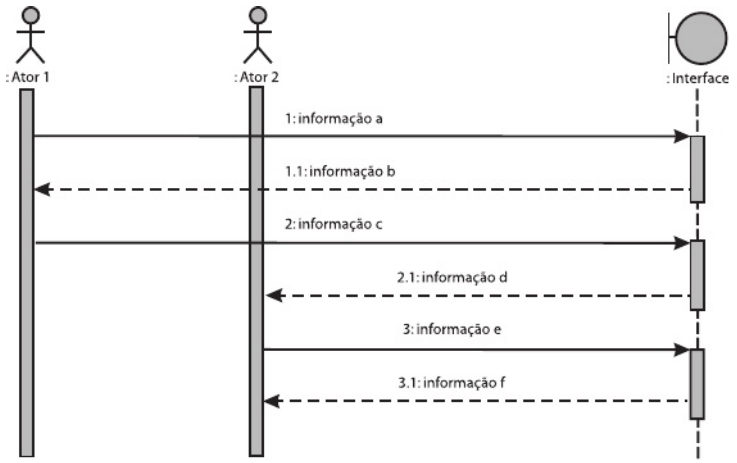


Figura 6.1: Diagrama de sequência de sistema.

As mensagens retornadas pelo sistema são tracejadas porque o sistema apenas reage aos atores, e a mensagem tracejada representa então esse mero retorno de informação a partir de um estímulo provocado por um dos atores. Da mesma forma, a numeração das mensagens pode ser diferente da numeração do caso de uso, visto que os retornos são subordinados à mensagem original. Assim, se o caso de uso numera os passos como 1, 2, 3, 4... , o diagrama de sequência poderá ter os passos equivalentes numerados com 1, 1.1, 2, 2.1....

Em relação à numeração, o caso de uso multicolunas pode ser mais apropriado, pois a cada linha dele corresponde uma mensagem de um ator para o sistema, e a resposta do sistema vem subordinada na mesma linha. Assim, a correspondência entre números de linha do caso de uso multicolunas e do diagrama de sequência de sistema será mais direta.

Como a atividade de análise não considera ainda os objetos internos ao sistema, será necessário representar o sistema como um único objeto, do tipo caixa-preta. Nesse caso, usa-se o símbolo de interface (Figura 6.1), conforme proposto por Jacobson *et al.* (1992). Um ator só pode se comunicar diretamente com a aplicação através de sua interface.

Atores, interfaces e outros elementos possuem, no diagrama de sequência, uma linha de tempo, representada pelas linhas verticais, onde os eventos podem ocorrer. Quando a linha está tracejada, o ator ou sistema está inativo. Quando a linha está cheia, isso significa que o ator ou sistema está ativo (operando ou aguardando o resultado de alguma operação). Atores humanos estão sempre ativos.

As linhas horizontais representam o fluxo de informação. Existem três tipos de envio de informação nesse diagrama:

a) entre atores (comunicação entre atores, correspondendo a passos complementares do caso de uso expandido);
b) dos atores para o sistema (*eventos de sistema* do caso de uso expandido);
c) do sistema para os atores (*respostas do sistema* do caso de uso expandido).

Os envios de informação do tipo "a" não pertencem ao escopo do sistema e apenas são úteis para ilustrar como a informação é trocada entre os atores.

Uma coisa para ter em mente quando se constrói esses diagramas é que a informação normalmente não é criada durante esses processos, mas apenas transferida ou transformada. Um ator ou sistema detém alguma informação

e, para realizar o processo, ele terá de passar essa informação adiante. Para realizar a venda de um livro, é necessário que o sistema tenha o registro da identificação do comprador que está efetuando a compra e o identificador do livro que está sendo vendido. Porém, quem detém essa informação é o comprador. O sistema detém o cadastro de todos os compradores, mas não sabe, até que ele se identifique, quem é o comprador que está nesse momento comprando um livro.

O diagrama de sequência pode ser construído para o fluxo principal do caso de uso e também para alguns fluxos alternativos com passos obrigatórios. Porém, o mais importante nesse momento ainda não é especificar as sequências, mas saber *quais* são as informações repassadas dos atores para o sistema e vice-versa. O analista deve então construir um catálogo com todas as operações e consultas de sistema identificadas nos fluxos principais e nos fluxos alternativos. Mais adiante, ainda no processo de análise, essas informações serão usadas para definir os contratos de operação e consulta de sistema que indicam como o sistema transforma a informação. Ferramentas *CASE* poderão construir esse catálogo automaticamente indicando a implementação de métodos em uma classe chamada *controladora de sistema.*

6.2. Representação de Casos de Uso Expandidos como Diagramas de Sequência de Sistema

O diagrama de sequência de sistema é uma forma de sistematizar o caso de uso expandido e, assim, refiná-lo para obter mais detalhes sobre o funcionamento do sistema.

A representação do caso de uso em um diagrama de sequência de sistema é feita em duas etapas:

a) representação dos passos do caso de uso como troca de informações entre os atores e a interface do sistema;

b) representação de operações e consultas de sistema como troca de mensagens entre a interface e a *controladora-fachada*da camada de domínio do sistema.

A primeira etapa é simples: a cada passo identificado com [IN] equivale um envio de informação de um ator para a interface do sistema, e a cada passo [OUT] equivale um envio de informação do sistema para um ator (Figura 6.2).

Caso de Uso: Comprar livros

1. [IN] O comprador informa sua identificação. (1)
2. [OUT] O sistema informa os livros disponíveis para venda (título, capa e preço). (1.1)
3. [IN] O comprador seleciona os livros que deseja comprar. (2)
4. [OUT] O sistema informa o valor total dos livros e apresenta as opções de endereço cadastradas. (2.1 e 2.2)
5. [IN] O comprador seleciona um endereço para entrega. (3)
6. [OUT] O sistema informa o valor do frete e total geral, bem como a lista de cartões de crédito já cadastrados para pagamento. (3.1, 3.2 e 3.3)
7. [IN] O comprador seleciona um cartão de crédito. (4)
8. [OUT] O sistema envia os dados do cartão e valor da venda para a operadora. (4.1)
9. [IN] A operadora informa o código de autorização. (5)
10. [OUT] O sistema informa ao comprador o prazo de entrega. (5.1)

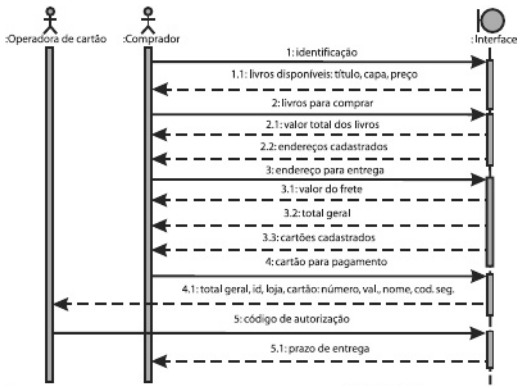


Figura 6.2: Um caso de uso e sua representação como diagrama de sequência de sistema (os números equivalentes do diagrama de sequência estão marcados entre parênteses no caso de uso para facilitar sua identificação).

Ao sistematizar os passos do caso de uso como envios de informação de atores para o sistema e vice-versa, o analista poderá, entre outras coisas, se dar conta de informações faltantes no caso de uso, como, por exemplo, o envio da identificação da loja no passo 8, conforme a Figura 6.2.

6.3. Ligação da Interface com o Domínio

Os eventos de sistema representam ações que o ator efetua contra a interface do sistema. Quando se usa uma interface Web, por exemplo, essas ações consistem no preenchimento de formulários e apertar de botões em páginas Web. Não são, ainda, operações propriamente ditas.

As *operações e consultas de sistema* são procedimentos computacionais, que são executados em função de um evento ou resposta de sistema. Trata-se agora de um componente do sistema que chama outro. No caso, é a interface que envia uma solicitação de execução de operação ou consulta de sistema para a camada de domínio, a qual é responsável pela execução de toda a lógica de acesso e transformação dos dados.

Nem sempre uma resposta de sistema exige parâmetros. Nesse caso, o passo 1 da Figura 6.4 não existiria. Mas a consulta de sistema seria ativada por algum outro evento de interface (qualquer ação do ator, passagem de tempo ou mesmo a inicialização da interface).

Essas regras de derivação de operações e consultas de sistema a partir de eventos e respostas de sistema são apenas uma primeira aproximação do projeto da camada de interface. Posteriormente, técnicas de modelagem e decisões de projeto poderão alterar a forma como essas operações são chamadas.

Existem, portanto, nos diagramas de sequência de sistema, quatro tipos de envio de mensagens:

a) *evento de sistema*: é uma ação realizada por um ator que envia alguma informação ao sistema. No diagrama é representado por uma seta do ator para a interface;
b) *resposta do sistema* : informação que o sistema repassa aos atores, representada no diagrama como uma seta tracejada da interface para os atores;
c) *operação do sistema*: uma chamada de método que o sistema executa internamente em resposta a um evento do sistema. A operação do sistema deve, por definição, alterar alguma informação armazenada. No diagrama é representada por uma seta da interface para a controladora rotulada com uma chamada de operação;
d) *consulta do sistema*: é uma chamada de método cuja execução faz com que o sistema retorne alguma informação que interessa aos atores. As consultas não devem alterar os dados armazenados no sistema, mas apenas retornar dados de uma forma apropriada ao usuário. No diagrama, as consultas são representadas por setas da interface para a controladora rotuladas com uma chamada de função e com valor de retorno explícito. Também seria possível representar os retornos como setas tracejadas da controladora para a interface, mas essa opção gera mais elementos no diagrama de sequência, deixando-o mais complexo, e deve ser evitada.

Também é possível que os diagramas apresentem comunicação entre os atores, representada por setas de um ator para outro, mas essas informações não geram nenhum tipo de consequência direta no sistema.

A regra que exige que as operações não retornem dados (o que equivaleria a uma consulta com efeito colateral) tem uma exceção aceitável consagrada pelo uso e pela praticidade. Se usada de forma controlada, essa exceção não

prejudica a coesão dos métodos. Trata-se da *criação de elementos conceituais*, correspondendo, por exemplo, ao *create* do padrão *CRUD*. Quando uma operação de sistema *cria* um elemento conceitual, admite-se que ela retorne uma *referência* para o elemento criado, conforme pode ser visto na primeira operação de sistema (1.1) da Figura 6.5, que retorna o identificador de uma compra recém-criada.

6.4. Estratégias Statefull e Stateless

Quando se faz o projeto do diagrama de sequência, cada informação é repassada pelos atores para a interface apenas uma vez. Porém, no nível seguinte, várias operações e consultas de sistema podem necessitar da mesma informação. Nesse ponto, o projetista deve decidir se vai considerar que a controladora possui memória temporária para essas informações (estratégia *statefull*) ou se ela é desprovida de memória (estratégia *stateless*), situação na qual cada vez que uma operação ou consulta necessitar de uma informação deverá recebê-la explicitamente da interface.

A Figura 6.5 mostra como ficaria o diagrama completo, feito a partir do exemplo da Figura 6.2, com o uso da estratégia *stateless*. A informação é passada pelo ator à interface apenas uma vez, mas, cada vez que uma operação de sistema necessita da informação, ela deve ser enviada novamente.

Em relação à Figura 6.2, esse diagrama também apresenta mais algumas diferenças. Em primeiro lugar, os descritores das informações passadas dos atores para a interface e vice-versa passaram a ser apresentados como identificadores. Cada informação individual corresponde a um identificador (uma expressão alfanumérica iniciando com letra e sem espaços). Em segundo lugar, o envio de uma sequência de informações (identificadores dos livros) passa a ocorrer dentro de uma estrutura de repetição (*loop*), de forma que cada chamada de operação e consulta de sistema corresponda a uma operação individual que possa ser efetivamente programada.

Como se pode ver nesse diagrama, idComprador e idCompra são passados explicitamente a todas as operações e consultas que precisam desses parâmetros.

No caso da estratégia *statefull*, assume-se que a controladora de alguma maneira possa lembrar os parâmetros já passados. Não se trata de informação persistente, ou seja, informação que vai ser armazenada no banco de dados ou estrutura equivalente, mas de informações temporárias (por exemplo, qual o

comprador corrente, qual a compra atual etc.) que ficam disponíveis apenas durante a execução da transação representada pelo caso de uso. A Figura 6.6 apresenta a mesma situação da Figura 6.5, mas desta vez com a estratégia *statefull*. Observa-se que, entre outras coisas, não é mais necessário retornar o identificador da nova compra, pois a controladora vai lembrar qual é.

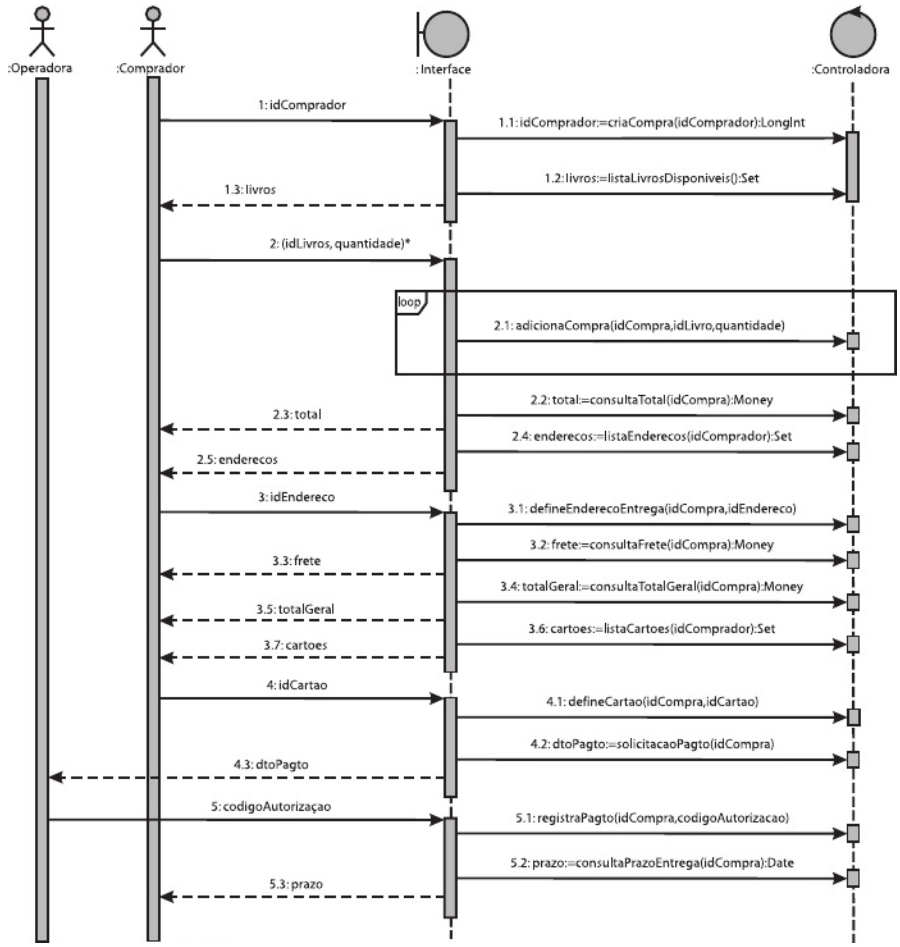


Figura 6.5: Diagrama de sequência completo com estratégia *stateless*.[1]

1 O significado da sigla "dto" nesta figura será explicado na seção 6.6.

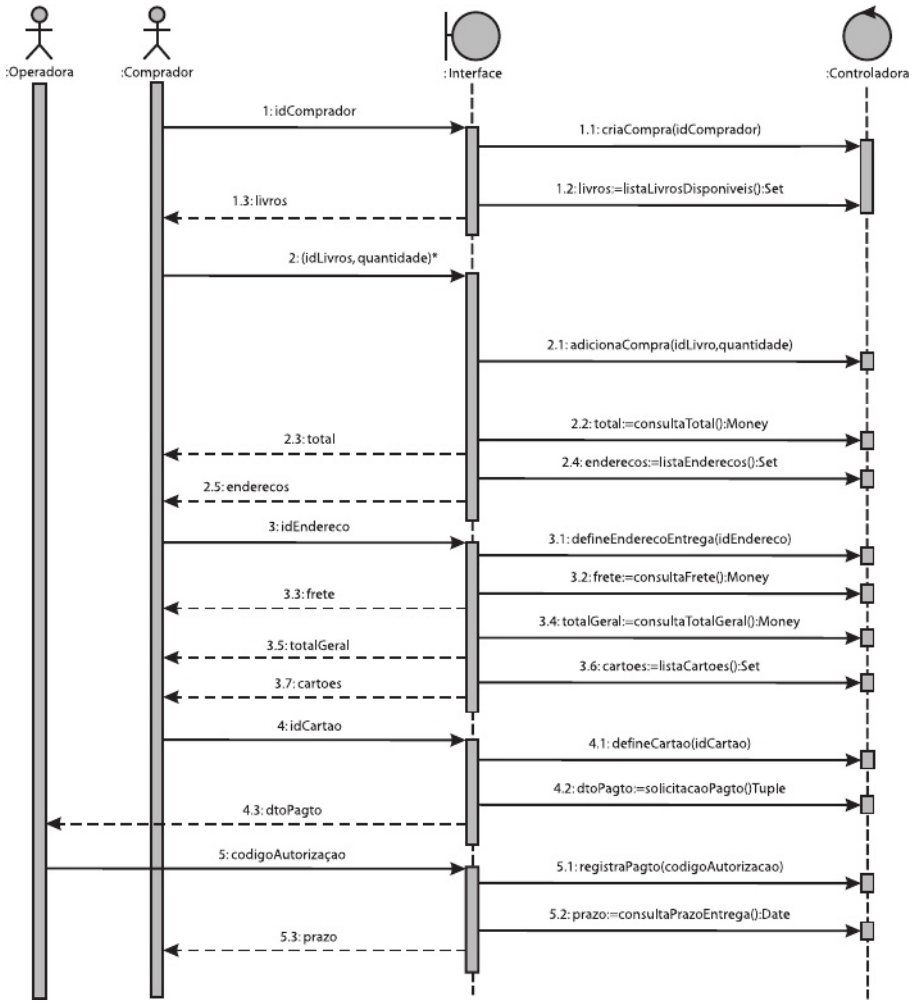


Figura 6.6: Diagrama de sequência completo com estratégia *statefull*.

Cabe ao projetista decidir qual estratégia usar. Os prós e contras de cada uma são os seguintes:

a) a estratégia *statefull* exige a implementação de um mecanismo de memória temporária (não persistente) para lembrar alguns parâmetros (uso de associações temporárias no modelo conceitual, por exemplo). A estratégia *stateless* não exige esse tipo de mecanismo;

b) a estratégia *stateless* exige maior passagem de parâmetros entre a interface e a controladora. Quando se trata de envio de informações pela rede, isso pode ser inconveniente. Com a estratégia *statefull*, cada informação é transmitida uma única vez.

6.5. Exceções em Diagramas de Sequência

Como visto no capítulo anterior, passos em casos de uso, especialmente eventos de sistema, podem ter exceções associadas, cujo tratamento é descrito em um fluxo alternativo do caso de uso.

Uma exceção pode ser modelada no diagrama de sequência como um evento condicional sinalizado que aborta a operação que está sendo tentada. (Figura 6.7).

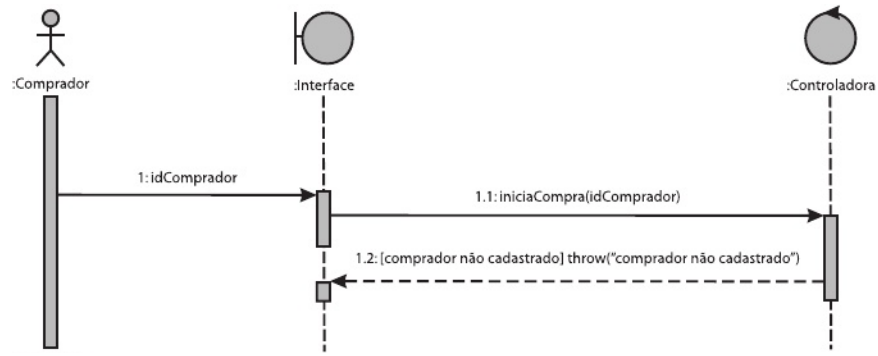


Figura 6.7: Uma operação de sistema com exceção.

As exceções possíveis são aquelas identificadas no caso de uso, ocorrendo nas operações de sistema correspondentes. Usualmente, não ocorrem exceções em consultas porque estas sempre retornam algum tipo de resultado.

Uma vez identificada a exceção, há pelo menos duas formas de tratá-la:

a) pode-se tratar a exceção na interface, emitindo algum tipo e mensagem ao ator e realizando o fluxo alternativo;.
b) pode-se também tentar transformar a exceção em uma precondição, evitando que o erro detectado ocorra na operação, mas que seja evitado antes da operação ser tentada.

No primeiro caso, pode-se representar o fluxo alternativo como uma chamada à operação de sistema criaComprador, que é parte do caso de uso <<CRUD>> "Gerenciar comprador". O resultado é mostrado na Figura 6.8.

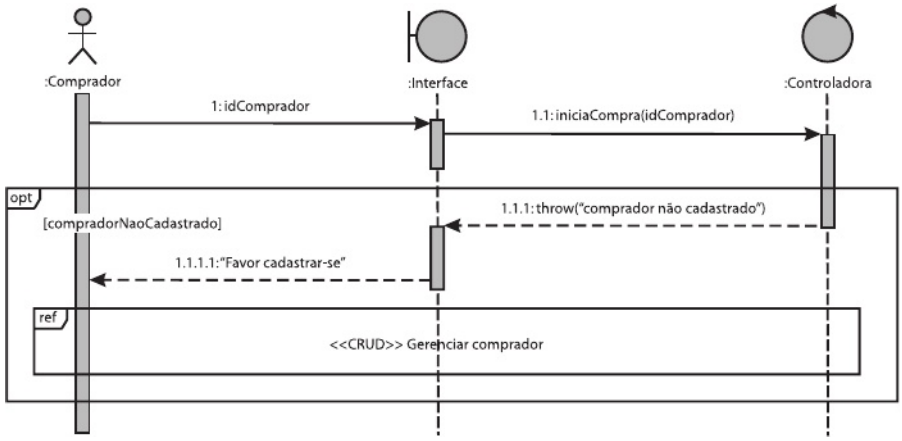


Figura 6.8: Uma exceção tratada em um diagrama de sequência.

Conforme estipulado na Figura 5.10 e complementado na Figura 5.11, o tratamento da exceção 1a do caso de uso Comprar livros exige o cadastramento do comprador. Na Figura 6.8 são usadas duas estruturas de *fragmento* típicas do diagrama de sequência para modelar situações que justamente fogem de uma sequência estrita.

O primeiro fragmento opt indica que os elementos incluídos nele só são executados quando a condição [compradorNaoCadastrado] for verdadeira. Essa exceção, aqui apenas mencionada, pode ser formalmente especificada como uma expressão OCL no contrato da operação de sistema iniciaCompra, conforme será visto no Capítulo 8.

O segundo fragmento, ref, indica uma referência a outro diagrama de sequência cujo nome é dado dentro do fragmento. Ou seja, os passos desse outro diagrama seriam expandidos no local onde a referência aparece.

É recomendável, para que os diagramas de sequência não fiquem demasiadamente complexos, que todas as sequências alternativas sejam definidas como diagramas separados (subordinados ao diagrama do fluxo principal se a ferramenta *CASE* assim o permitir) e referenciados no diagrama do fluxo principal, com fragmentos ref.

Outra maneira de tratar a exceção é evitar que ela ocorra na operação de sistema, transformando-a em precondição. Isso pode ser feito se uma consulta verificasse a condição antes de tentar a operação de sistema. Essa opção é mostrada na Figura 6.9.

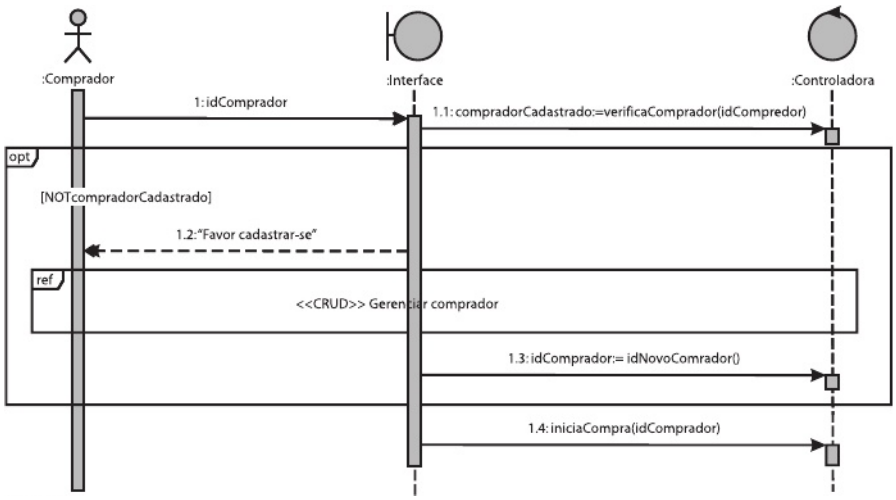


Figura 6.9: Uma exceção transformada em precondição em um diagrama de sequência.

Nesse caso, a operação de sistema iniciaCompra, em vez de ter uma exceção, como no caso da Figura 6.8, terá uma precondição: o idComprador sempre será um identificador válido.

A mensagem 1.3 nesse diagrama será desnecessária se a variante *create* do *CRUD* for definida de forma a já retornar à interface o idComprador.

Uma terceira forma seria possível caso o idComprador fosse selecionado de uma lista de identificadores válidos em vez de ser meramente digitado. Nesse caso, a exceção simplesmente não ocorre mais no caso de uso, sendo transformada em uma precondição do próprio caso de uso. A Figura 6.10 representa essa situação.

Na mensagem 3, a expressão entre chaves {ids→includes(idComprador)} corresponde a uma condição escrita em OCL que exige que idComprador seja um elemento da lista ids retornada pela controladora. Matematicamente, essa expressão poderia ser escrita como idComprador ∈ ids ou, ainda, como ids ∋ idComprador.

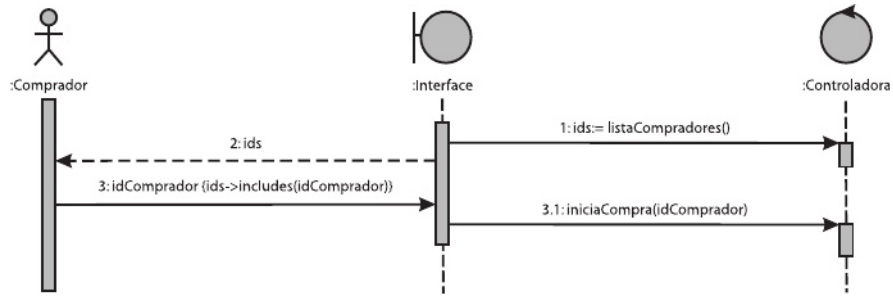


Figura 6.10: Uma exceção eliminada pela sua transformação em precondição.

Porém, pode não ser desejável que a operação seja executada dessa forma nesse caso, pois, assim, qualquer usuário poderia saber quais são os identificadores de compradores válidos. Esse tipo de modelagem seria adequado, entretanto, para selecionar livros, porque nesse caso não há problema de quebra de segurança caso um usuário possa acessar todos os códigos válidos.

6.6. Padrão DTO — Data Transfer Object

Na Figura 6.5, foi usado o conceito de DTO (*Data Transfer Object*). Muitas vezes pode ser inconveniente rotular transições de um diagrama de sequência (ou outros) com uma série de informações ou parâmetros, como nome, endereço, CPF, telefone etc. Assume-se, então, que uma entidade mais complexa pode representar esse conjunto de atributos. No caso da Figura 6.5, usou-se o termo dtoPagto para referenciar uma série de informações alfanuméricas sobre pagamentos.

Mas qual a diferença entre um DTO e as classes do modelo conceitual que serão discutidas no capítulo seguinte? A diferença reside no fato de que uma classe ou conceito tem uma estrutura semântica complexa, com associações, restrições e, mais adiante, quando transformada em classe de implementação, também métodos que serão responsáveis pela transformação e acesso à informação.

Essas classes pertencem à camada de domínio da aplicação, e sua funcionalidade deve ser encapsulada pela controladora-fachada. Não é possível, portanto, que os atores ou a interface (no diagrama de sequência de sistema) façam uso dessas classes.

Assim, a classe DTO é uma espécie de *registro* ou *tupla*, ou seja, uma classe que representa um conjunto coeso de informações alfanuméricas (ou atributos)

e que é capaz apenas de acessar e modificar essas mesmas informações (através de métodos *get* e *set*), não tendo nenhum outro tipo de funcionalidade.

Essas classes servem exatamente como estruturas de dados (registros) que permitem acesso e modificação, e um conjunto de informações alfanuméricas que, de outra forma, consistiria em longas listas de atributos. Um DTO, a princípio, deve ser definido como uma classe estereotipada em um diagrama específico (fora do modelo conceitual): o pacote de DTOs. A Figura 6.11 apresenta um exemplo.

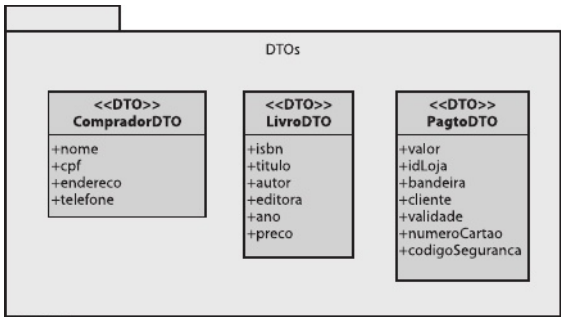


Figura 6.11: Um pacote com DTOs.

Uma vez definido o DTO, seu nome pode ser usado nos diagramas de sequência para representar a lista de atributos que ele define. Sugere-se o uso do sufixo (ou prefixo) DTO sempre para evitar confusão com as classes conceituais que serão estudadas no capítulo seguinte.

Para referenciar um valor específico dentro de um DTO, como, por exemplo, o CPF de um comprador, pode-se escrever: CompradorDTO.cpf .

A vantagem de se ter DTOs paralelamente às classes conceituais é que, dessa forma, diferentes visões de interface sobre a informação gerenciada pelo sistema não afetarão diretamente a organização interna do sistema. Pode-se traçar um paralelismo com a área de banco de dados: as classes conceituais correspondem às tabelas, que são os repositórios persistentes da informação, enquanto os DTOs correspondem às visões, que são diferentes maneiras como esses mesmos dados podem ser vistos e acessados.

Uma forma eficiente de implementar DTOs é pelo uso do padrão *Protection Proxy*, que consiste em utilizar um objeto com uma interface simples (consistindo apenas em *getters* e *setters*) encapsulando um ou mais objetos conceituais.

Capítulo

7

Modelagem Conceitual

A *análise de domínio* está relacionada à descoberta das informações que são gerenciadas no sistema, ou seja, à representação e transformação da informação. Ela ocorre em pelo menos duas fases do processo unificado. Na fase de concepção, pode-se fazer um modelo conceitual preliminar. Na fase de elaboração, esse modelo é refinado e complementado.

No sistema de informações de uma livraria virtual, as informações descobertas na análise de domínio possivelmente seriam relativas aos compradores, livros, editoras, autores, pagamentos, vendas etc.

As informações têm dois aspectos analisados: o *estático* (também denominado *estrutural*, que será estudado neste capítulo) e o *funcional* (estudado no Capítulo 8). O aspecto estático pode ser representado no *modelo conceitual*, e o aspecto funcional pode ser representado através dos *contratos de operações e consultas de sistema*.

Na atividade de análise não existe *modelo dinâmico*, visto que a análise considera apenas a descrição da visão *externa* do sistema. O modelo dinâmico, consistindo nas colaborações entre objetos, é reservado à atividade de projeto (Capítulo 9), pois apenas nessa fase é que se vai tratar dos aspectos internos do sistema. Assim, o modelo funcional da análise apenas especifica o que entra e o que sai do sistema, sem indicar como as transformações ocorrem. Já o mo-

delo dinâmico da atividade de projeto vai ter de mostrar claramente como as transformações ocorrem através das colaborações entre objetos.

O modelo conceitual deve descrever a informação que o sistema vai gerenciar. Trata-se de um artefato do domínio do problema e não do domínio da solução. Portanto, o modelo conceitual não deve ser confundido com a arquitetura do software (representada no DCP, ou diagrama de classes de projeto do Capítulo 9) porque esta, embora inicialmente derivada do modelo conceitual, pertence ao domínio da solução e, portanto, serve a um objetivo diferente.

O modelo conceitual também não deve ser confundido com o modelo de dados (Capítulo 11), pois o modelo de dados enfatiza a representação e a organização dos dados armazenados, enquanto o modelo conceitual visa a representar a compreensão da informação e não a sua representação física. Assim, um modelo de dados relacional é apenas uma possível representação física de um modelo conceitual mais essencial.

O analista deve lembrar que a atividade de análise de domínio considera apenas o mundo exterior ao sistema. Por isso, não faz sentido falar em modelo de dados nessa fase porque os dados estarão representados no interior do sistema. Uma maneira interessante de compreender o modelo conceitual é imaginar que os elementos descritos nele correspondem a informações que inicialmente existem apenas na mente do usuário, como na Figura 7.1, e não em um sistema físico de armazenamento.

Figura 7.1: O modelo conceitual é uma representação da visão que o usuário tem das informações gerenciadas pelo sistema.

O usuário, através das operações e consultas de sistema (Capítulo 6), passa informações ao sistema e recupera informações do sistema. O sistema nem sequer precisa ser considerado como um sistema *computacional* nesse momento. Ou seja, essas informações existem independentemente de um computador para armazená-las.

O objetivo da análise é estudar o problema. Mas o sistema computacional seria uma solução para o problema e, portanto, objeto da atividade de projeto. O sistema-solução poderia também não ser computacional. Seria possível analisar todo um sistema e propor uma solução manual para implementá-lo, na qual os dados são armazenados em fichas de papel e as operações são efetuadas por funcionários da empresa usando lápis, borracha e grampeador.

Assim como os casos de uso essenciais, o modelo conceitual deve ser independente da solução física que virá a ser adotada e deve conter apenas elementos referentes ao domínio do problema em questão, ficando relegados à atividade de projeto os elementos da solução, ou seja, todos os conceitos que se referem à tecnologia, como interfaces, formas de armazenamento (banco de dados), segurança de acesso, comunicação etc.

O modelo conceitual representa somente o aspecto estático da informação. Portanto, não podem existir no modelo conceitual referências a operações ou aspectos dinâmicos dos sistemas. Então, embora o modelo conceitual seja representado pelo diagrama de classes da UML, o analista não deve ainda adicionar métodos a essas classes (o Capítulo 9 vai mostrar como fazer isso de forma adequada na atividade de projeto).

Quando se trabalha modelagem conceitual com o diagrama de classes da UML, existem precisamente três tipos de elementos para modelar a informação:

a) *atributos*, que são informações alfanuméricas simples, como números, textos, datas etc. Exemplos de atributos no sistema Livir são: nome do comprador, data do pagamento, título do livro e valor da venda. Observa-se que um atributo sempre está ligado a um elemento mais complexo: o conceito;
b) *conceitos*, que são a representação da informação complexa que agrega atributos e que não pode ser descrita meramente por tipos alfanuméricos. Exemplos de conceitos no sistema Livir são: livro, comprador, venda e pagamento;

c) *associações*, que consistem em um tipo de informação que liga diferentes conceitos entre si. Porém, a associação é mais do que uma mera ligação: ela própria *é* um tipo de informação. No sistema Livir, devem existir associações entre uma venda e seus itens e entre a venda e seu comprador.

Nas próximas seções, esses elementos serão detalhados. É praticamente impossível falar de um sem mencionar os outros, pois os três se inter-relacionam fortemente.

7.1. Atributos

Atributos são, no modelo conceitual, os tipos escalares, textos e outros formatos derivados populares, como data, moeda, intervalo etc.

Não devem ser consideradas como atributos as estruturas de dados com múltiplos valores como listas, conjuntos, árvores etc. Essas estruturas devem ser modeladas como associações, conforme será visto mais adiante.

Conceitos complexos (classes) também não devem ser modelados como atributos. Por exemplo, um livro não pode ser atributo de uma venda. Pode existir, se for o caso, uma *associação* entre um livro e uma venda, pois ambos são conceitos complexos.

Atributos são sempre representados no seio de uma classe, conforme a Figura 7.2, em que a classe Comprador tem os atributos nome, cpf, endereco e telefone.

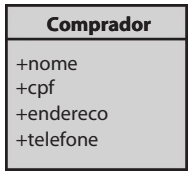


Figura 7.2: Atributos representados dentro de uma classe.

7.1.1. Tipagem

Atributos podem ser tipados, embora o modelo conceitual não exija isso explicitamente. A Figura 7.3 mostra uma versão da mesma classe da Figura 7.2 com atributos tipados.

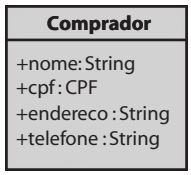


Figura 7.3: Atributos tipados.

O significado dos tipos é o mesmo correntemente atribuído pelas linguagens de programação. Observa-se, na Figura 7.3, a existência de um tipo clássico (String) e um tipo primitivo definido (CPF). Quando um atributo é definido por regras de formação, como é o caso do CPF, convém que se defina um tipo primitivo especialmente para ele, como foi feito na Figura 7.3.

O atributo telefone também poderia ser definido por um tipo especial, já que admite uma formatação específica.

Por outro lado, seria possível argumentar que um telefone é um número. Isso é verdade. Mas dificilmente alguém faria operações matemáticas com números de telefones, a não ser nos cálculos de improbabilidade da "Coração de Ouro" (Adams, 2005), mas isso já é ficção. No mundo real, embora um telefone seja composto apenas por números, ele se comporta mais como uma *string*. É mais comum extrair uma *substring* (código DDD, por exemplo) do que somar um telefone com outro.

Já o endereço é um caso à parte. Trata-se de um atributo ou um conceito complexo? Afinal, um endereço é composto por logradouro, CEP, cidade etc. Esse caso, como muitos outros, define-se pela necessidade de informação do sistema. Se endereços são usados apenas para gerar etiquetas de envelopes, então eles se comportam como atributos e podem ser representados por uma simples *string*. Porém, se endereços são usados para calcular distâncias entre diferentes pontos ou para agrupar compradores por localidade ou proximidade, então eles se comportam como conceitos complexos e devem ser modelados como uma classe com atributos e associações próprias.

7.1.2. Valores Iniciais

Um atributo pode ser declarado com um valor inicial, ou seja, sempre que uma instância do conceito (classe) for criada, aquele atributo receberá o valor inicial definido, que posteriormente poderá ser mudado, se for o caso.

Uma venda, por exemplo, pode ser criada com um valor total que inicialmente (antes que haja algum livro na venda) será zero. Isso pode ser definido no próprio diagrama de classes do modelo conceitual, como mostrado na Figura 7.4.

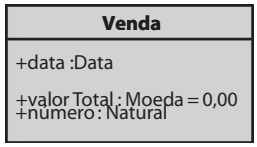


Figura 7.4: Um atributo com valor inicial.

Pode-se usar a linguagem OCL também para definir o valor inicial de um atributo de forma textual. Para isso, é necessário primeiro declarar o contexto da expressão (no caso, o atributo valorTotal, na classe Venda, representado por Venda::valorTotal) e usando a expressão init para indicar que se trata de um valor inicial de atributo. A expressão OCL seria escrita assim:

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

Pode-se omitir o tipo Moeda, pois a OCL não obriga a declaração de tipos nas suas expressões:

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

É possível também que um atributo tenha um valor inicial calculado de forma mais complexa. Mais adiante serão apresentados exemplos de expressões complexas com OCL que podem ser usadas para inicializar atributos com valores como somatórios, quantidade de elementos associados, maior elemento associado etc.

7.1.3. Atributos Derivados

Atributos derivados são valores alfanuméricos (novamente não se admitem objetos nem estruturas de dados como conjuntos e listas) que não são definidos senão através de um cálculo. Ao contrário dos valores iniciais, que são atribuídos na criação do objeto e depois podem ser mudados à vontade, os atributos derivados não admitem qualquer mudança diretamente neles. Em outras palavras, são atributos *read-only*.

Um atributo derivado deve ser definido por uma expressão. No diagrama, representa-se o atributo derivado com uma barra (/) antes do nome do

atributo seguida da expressão que o define. Na Figura 7.5, define-se que o lucro bruto de um produto é a diferença entre seu preço de venda e seu preço de compra.

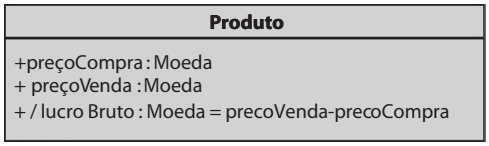


Figura 7.5: Um atributo derivado.

Em OCL, o mesmo atributo derivado poderia ser definido usando a expressão derive:

```
Context Produto::lucroBruto:Moeda
 derive:
  precoVenda – precoCompra
```

Nessa classe, apenas os atributos precoCompra e precoVenda podem ser diretamente alterados por um *setter*. O atributo lucroBruto pode apenas ser consultado. Ele é o resultado do cálculo conforme definido.

Mecanismos de otimização de fase de implementação podem definir se atributos derivados como lucroBruto serão recalculados a cada vez que forem acessados ou se serão mantidos em algum armazenamento oculto e recalculados apenas quando um de seus componentes for mudado. Por exemplo, o lucroBruto poderia ser recalculado sempre que precoCompra ou precoVenda executarem a operação *set* que altera seus valores.

7.1.4. Enumerações

Enumerações são um meio-termo entre o conceito e o atributo. Elas são basicamente *strings* e se comportam como tal, mas há um conjunto predefinido de *strings* válidas que constitui a enumeração. Por exemplo, o dia da semana só pode assumir um valor dentre sete possíveis: "domingo", "segunda-feira", "terça-feira" etc. Assim, um atributo cujo valor pode ser um dia da semana poderá ser tipado com uma enumeração.

A enumeração pode aparecer nos diagramas UML como uma classe estereotipada. Sugere-se que não sejam colocadas no modelo conceitual, mas em um pacote específico para conter enumerações, como mostrado na Figura 7.6.

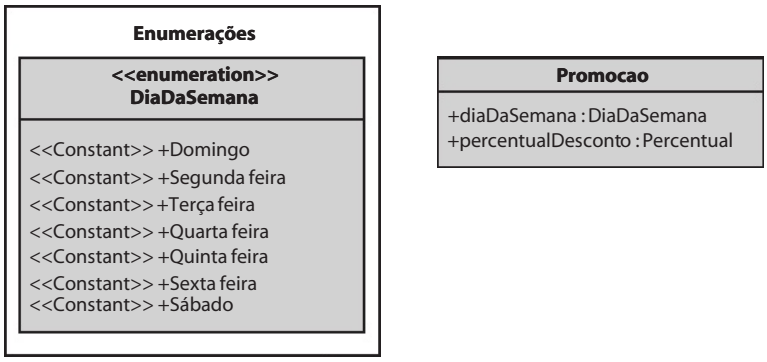


Figura 7.6: Um pacote com uma enumeração e seu uso em uma classe.

Na Figura 7.6, o atributo diaDaSemana, na classe Promocao pode assumir um dentre os sete valores da enumeração DiaDaSemana.

Em hipótese alguma enumerações podem ter associações com outros elementos ou atributos (os valores que aparecem dentro da declaração da enumeração são constantes e não atributos). Se isso acontecer, não se trata mais de uma enumeração, mas de um conceito complexo. Nesse caso, não se deve usar o estereótipo <<enumeration>> nem se pode usar o nome da enumeração como se fosse um tipo, como na Figura 7.6. Quando se trata de um conceito complexo, sua relação com outros conceitos tem de ser feita através de associações, como será mostrado mais adiante.

7.1.5. Tipos Primitivos

O analista pode e deve definir *tipos primitivos* sempre que se deparar com atributos que tenham regras de formação, como no caso do CPF. Tipos primitivos podem ser classes estereotipadas com <<primitive>>, como na Figura 7.50.

Alguns tipos primitivos como Date, Money etc. já são definidos em OCL e na maioria das linguagens de programação. Mas outros tipos podem ser criados e suas regras de formação podem ser definidas pelo analista. É o caso de CPF, CEP, NumeroPrimo e quaisquer outros tipos alfanuméricos cuja regra de formação possa ser analisada sintaticamente, ou seja, avaliando expressões em que não constem dados gerenciados pelo sistema. Por exemplo, CPF pode ser um tipo primitivo, mas CPFDeComprador não, pois para saber se o CPF

pertence a um comprador seria necessário avaliar os dados de compradores gerenciados pelo sistema.

7.2. Conceitos

É impossível falar de atributos sem falar nos conceitos, já que são fortemente ligados. Conceitos são usualmente agrupamentos coesos de atributos sobre uma mesma entidade de informação.

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

7.2.1. Identificadores

Um *identificador* é um atributo que permite que uma instância de um conceito seja diferenciada de outras. Uma vez que um atributo tenha sido estereotipado como identificador (<<oid>> do inglês *Object IDentifier*), não é possível que existam duas instâncias do mesmo conceito com o mesmo valor para esse atributo. Um exemplo de identificador é o número de CPF para os brasileiros (Figura 7.7) porque não existem duas pessoas com o mesmo CPF (pelo menos não oficialmente).

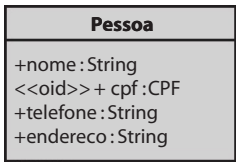


Figura 7.7: Um conceito com identificador.

Alguns dialetos usam o estereótipo <<PK>> para identificadores, derivado do inglês *Primary Key* (chave primária), um conceito de banco de dados que representa um atributo ou campo que não pode ser repetido em duas entidades.

Entretanto, a semelhança entre OID e PK não é perfeita porque um registro em um banco de dados relacional pode ter apenas uma chave primária (algumas vezes composta por mais de um atributo), enquanto um conceito

ceitos independentes). Assim, pode-se dizer "João dormiu" sem acrescentar nenhum complemento ao verbo, e a frase faz sentido. Mas não se pode dizer "João fez" sem que haja um complemento, mesmo que por elipse. Para a frase ter sentido, João tem de ter feito *alguma coisa*.

Mas, para que serve essa distinção? É interessante notar que apenas os conceitos independentes se prestam a ser cadastros, ou seja, eles são operáveis através de um caso de uso *CRUD*. Pode-se ter, então, cadastros de pessoas, de livros, de fornecedores etc., mas, usualmente, os conceitos dependentes não se prestam a esse tipo de operação. Compras, vales-presente, pagamentos não podem simplesmente ser *cadastrados* na forma *CRUD*, mas são operados nos casos de uso que correspondem a processos de negócio, porque possivelmente comportam interações e exceções que fogem ao padrão.

Mais adiante, será visto que o acesso à informação no sistema (modelo dinâmico) inicia sempre na controladora-fachada. Ou seja, qualquer caminho de acesso à informação parte da controladora-fachada. Assim, conceitos diretamente ligados à controladora-fachada são aqueles cujas instâncias podem ser acessadas diretamente. Livros e compradores são cadastros (conceitos independentes) e, portanto, podem ser acessados diretamente.

Como consequência disso, apenas os conceitos independentes estarão inicialmente associados à classe controladora de sistema. Os conceitos dependentes não terão esse tipo de associação a não ser que ela represente algum tipo de informação adicional. Por exemplo, uma associação entre reservas (conceito dependente) poderia estar ligada à controladora-fachada através de uma associação ordenada para indicar a ordem de prioridade entre as reservas, conforme será visto mais adiante.

7.3. Como Encontrar Conceitos e Atributos

O processo de descoberta dos elementos do modelo conceitual pode variar. Porém, uma forma bastante útil é olhar para o texto dos casos de uso expandidos ou os diagramas de sequência de sistema. A partir desses artefatos, pode-se descobrir todos os elementos textuais que eventualmente referenciam informação a ser guardada e/ou processada.

Usualmente, esses elementos textuais são compostos por substantivos, como "pessoa", "compra", "pagamento" etc., ou por expressões que denotam substantivos (conhecidas em linguística como *sintagmas nominais*), como

"autorização de venda". Além disso, no texto, algumas vezes alguns verbos podem indicar conceitos, pois o verbo pode exprimir um ato que corresponde a um substantivo, como por exemplo, "pagar", que corresponde ao substantivo "pagamento"; "comprar", que corresponde ao substantivo "compra" etc.

Via de regra, entretanto, o analista deve ter em mente os objetivos do sistema enquanto procura descobrir os elementos do modelo conceitual. Não é interessante representar, no modelo conceitual, a informação que é irrelevante para o sistema. Assim, nem todos os substantivos e verbos deverão ser considerados no modelo. O analista tem a responsabilidade de compreender quais as verdadeiras necessidades de informação e filtrar as irrelevâncias.

O processo de identificação dos conceitos e atributos, então, consiste em:

a) identificar no texto dos casos de uso as palavras ou sintagmas que correspondem a conceitos sobre os quais se tem interesse em manter informação no sistema;
b) agrupar as palavras ou expressões que são sinônimos, como, por exemplo, "compra" e "aquisição", "comprador" e "cliente" etc.;
c) identificar quais dos itens considerados correspondem a conceitos complexos e quais são meros atributos. Os atributos são aqueles elementos que podem ser considerados alfanuméricos, usualmente: nomes, números em geral, códigos, datas, valores em moeda, valores booleanos (verdadeiro ou falso) etc.

Aplicando essa técnica ao caso de uso da Figura 5.10, são encontrados os principais elementos de informação a serem trabalhados. Na Figura 7.8, os elementos identificados como conceitos ou atributos são grifados no texto.

Caso de Uso: Comprar livros

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

b) *operação* é o ato de transformar a informação, fazendo-a passar de um estado para outro, mudando, por exemplo, a configuração das associações, destruindo e/ou criando novas associações ou objetos, ou modificando o valor dos atributos.

Assim, o texto dos casos de uso está frequentemente repleto de operações, mas não de associações. Quanto se têm, por exemplo, as classes Pessoa e Automovel, como na Figura 7.10, a associação estática que existe entre elas pode indicar a posse de uma pela outra.

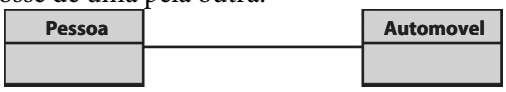


Figura 7.10: Representação de uma associação estática entre dois conceitos.

Porém, existem diferentes formas de associação entre pessoas e automóveis que não meramente a posse. Uma pessoa pode ser *passageira* de um automóvel ou *motorista* dele, ou ainda a associação pode simplesmente representar que uma determinada pessoa *gosta* de um determinado automóvel. Para eliminar tais ambiguidades, é conveniente, em muitos casos, utilizar um *nome de papel* para as associações, o qual pode ser colocado em um ou ambos os lados e deve ser lido como se fosse uma função. Na Figura 7.11, por exemplo, uma pessoa está no papel ou função de motorista de um automóvel.

Figura 7.11: Uma associação com nome de papel.

Diferentes papéis podem ser representados através de associações diferentes, como na Figura 7.12.

Figura 7.12: Múltiplas associações entre conceitos.

Na falta de um nome de papel explícito, o próprio nome da classe associada deve ser considerado como nome de papel. No caso da Figura 7.12, um automóvel explicitamente tem dois tipos de associação com pessoas: dono e motorista. Do ponto de vista inverso, porém, uma pessoa tem dois tipos de

associação com automóveis: aqueles dos quais é dona, correspondendo à sua frota, e aquele do qual é motorista, correspondendo simplesmente ao seu automovel.

Voltando à discussão sobre as associações não serem operações, observa-se que uma pessoa pode comprar um automóvel. Isso é uma operação porque transforma as associações: uma associação de posse deixa de existir e outra é criada em seu lugar. Seria incorreto representar a operação como uma associação, como na Figura 7.13.

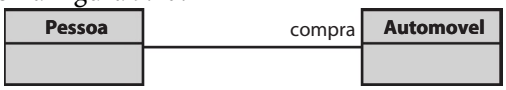


Figura 7.13: Uma operação incorretamente rotulando um papel de associação.

Segundo a definição da UML, associações também podem ter nomes, que são colocados no meio da linha que liga duas classes e não nas extremidades. Porém, tais nomes, além de serem difíceis de atribuir (analistas iniciantes preencherão seus diagramas com associações chamadas "possui" ou sinônimos), não têm qualquer resultado prático. Mais vale trabalhar com bons nomes de papel do que com nomes de associações. Então, é prático simplesmente ignorar que associações tenham nomes e viver feliz com os nomes de papel, muito mais úteis para definir possibilidades de navegação entre conceitos (Capítulos 8 e 9) e para nomear elementos de programação (Capítulo 12).

Algumas vezes, pode ser interessante guardar informações sobre operações ou transações que foram efetuadas. Assim, pode ser interessante para o sistema guardar as informações sobre a transação de compra, como data, valor pago etc. Nesse caso, a transação é tratada no modelo conceitual como um conceito complexo (Figura 7.14) e representa estaticamente a memória da operação que um dia foi efetuada.

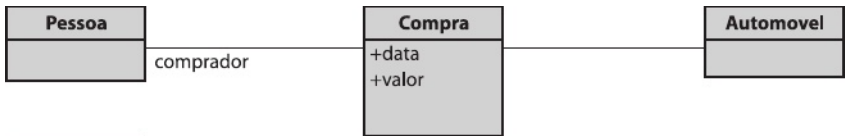


Figura 7.14: Transação representada como conceito.

Porém, como se pode notar, a transação é representada por um conceito, enquanto as associações continuam representando ligações estáticas entre conceitos e não operações ou transformações.

7.4.1. Como Encontrar Associações

Se a informação correspondente aos conceitos e atributos pode ser facilmente encontrada no texto dos casos de uso, o mesmo não ocorre com as associações.

Mas, então, como encontrar as associações entre os conceitos complexos? Há duas regras:

a) conceitos dependentes (como Compra) precisam necessariamente estar ligados aos conceitos que os complementam (como Comprador e Item);
b) informações associativas só podem ser representadas através de associações.

Informações associativas podem até aparecer nos casos de uso como conceitos. Mas quando se afirma que uma pessoa é dona ou motorista de um automóvel, essa informação só pode ser colocada na forma de uma associação. Analistas viciados em modelos de dados relacionais poderão fazer uma modelagem como a da Figura 7.15, mas está incorreta, pois atributos devem ser alfanuméricos e nunca conceitos complexos (para isso existem associações).

Figura 7.15: Um atributo representando *indevidamente* uma associação.

Também é incorreto utilizar chaves estrangeiras (Date, 1982), como na Figura 7.16. Quando dois conceitos complexos se relacionam, o elemento de modelagem é a associação, e fim de conversa!

Figura 7.16: Um atributo como chave estrangeira representando *indevidamente* uma associação.

Uma regra geral de *coesão* a ser usada é que um conceito só deve conter seus próprios atributos e nunca os atributos de outro conceito. Ora, o CPF do dono é um atributo de uma pessoa e não do automóvel. Por isso é inadequado representá-lo como na Figura 7.16. Outro motivo para que se tenham associações visíveis é prático: é muito mais fácil perceber quais objetos efetivamente têm referências para com outros. Mais adiante (Capítulo 9) será visto como essas linhas de *visibilidade* são importantes para que se possa projetar um código efetivamente orientado a objetos de qualidade.

7.4.2. Multiplicidade de Papéis

Na modelagem conceitual, é fundamental que se saiba a quantidade de elementos que uma associação permite em cada um de seus papéis. Por exemplo, na associação entre Pessoa e Automovel da Figura 7.11, quantos automóveis uma pessoa pode dirigir? Quantos motoristas um automóvel pode ter?

A resposta sempre dependerá de um estudo sobre a natureza do problema e sobre o real significado da associação, especialmente se ela representa o presente ou o histórico. Quer dizer, se a associação da Figura 7.11 representa o presente, pode-se admitir que um automóvel tenha um único motorista. Mas, se a associação representa o histórico, pode-se admitir que o automóvel tenha uma série de motoristas, que o dirigiram em momentos diferentes.

Assim, é fundamental que o analista decida claramente o que a associação significa antes de anotar a multiplicidade de seus papéis.

Há, basicamente, duas decisões a tomar sobre a multiplicidade de um papel de associação:

a) se o papel é obrigatório ou não. Por exemplo, uma pessoa é obrigada a ter pelo menos um automóvel? Um automóvel deve obrigatoriamente ter um dono?;
b) se a quantidade de instâncias que podem ser associadas através do papel tem um limite conceitual definido. Por exemplo, existe um número máximo ou mínimo de automóveis que uma pessoa pode possuir?

Deve-se tomar cuidado com algumas armadilhas conceituais. Em relação ao primeiro tópico, por exemplo, espera-se que a toda venda corresponda um pagamento. Mas isso não torna a associação obrigatória, pois a venda pode existir sem pagamento. Um dia ela possivelmente será paga, mas pode existir sem o pagamento por algum tempo. Então, esse papel *não* é obrigatório para a venda.

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

A multiplicidade de papel é representada por uma expressão numérica, onde:

a) "*" representa o infinito;
b) a vírgula (,) significa "e";
c) ponto-ponto (..) significa "até".

A seguir são apresentados alguns exemplos usuais e seu significado:

a) 1 exatamente um
b) 0..1 zero ou um
c) * de zero a infinito
d) 1..* de um a infinito
e) 2..5 de dois a cinco
f) 2,5 dois ou cinco
g) 2,5..8 dois ou de cinco a oito

Os valores de multiplicidade que incluem o zero são opcionais. Já os demais representam papéis obrigatórios.

A Figura 7.17 mostra que uma pessoa pode ter uma frota composta de um número qualquer de automóveis (opcional) e que pode ser motorista de um automóvel (também opcional). Por outro lado, mostra que um automóvel tem um único dono (obrigatório) e pode ter ou não um motorista (opcional).

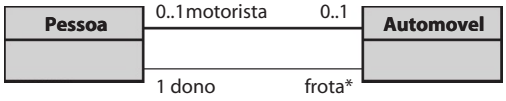


Figura 7.17: Associações com multiplicidade de papel.

7.4.3. Direção das Associações

Uma associação, no modelo conceitual, deve ser *não direcional*, isto é, a srcem e o destino da associação não devem ser estabelecidos. Isso se deve ao fato de que, na atividade de análise, basta saber que dois conceitos estão associados.

Há pouca praticidade em definir prematuramente (antes da atividade de projeto) a direção de uma associação. Associações *unidirecionais* teriam como vantagem apenas o seguinte:

a) não é necessário definir o nome de papel na srcem de uma associação unidirecional;

b) associações unidirecionais podem ser implementadas de forma mais eficiente que as bidirecionais.

Assim, essa decisão, que não afeta significativamente a modelagem conceitual, pode ser tomada com melhor conhecimento de causa (a direção das mensagens trocadas entre objetos) na atividade de projeto, como será visto no Capítulo 9.

7.4.4. Associação Derivada

Assim como algumas vezes pode ser interessante definir atributos derivados, que são calculados a partir de outros valores, pode ser também interessante ter associações derivadas, ou seja, associações que, em vez de serem representadas fisicamente, são calculadas a partir de outras informações que se tenha. Por exemplo, suponha que uma venda, em vez de se associar diretamente aos livros, se associe a um conjunto de itens, e estes, por sua vez, representem uma quantidade e um título de livro específico (Figura 7.18).

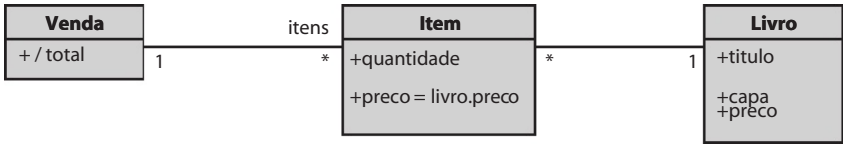


Figura 7.18: Uma venda e seus itens.

Esse tipo de modelagem é necessário quando os itens de venda não são representados individualmente, mas como quantidades de algum produto. Além disso, o livro enquanto produto pode ter seu preço atualizado, mas o item que foi vendido terá sempre o mesmo preço. Daí a necessidade de representar também o preço do item como um atributo com valor inicial igual ao preço do livro.

Porém, a partir da venda, não é mais possível acessar diretamente o conjunto de livros. Seria necessário tomar o conjunto de itens e, para cada item, verificar qual é o livro associado. Criar uma nova associação entre Venda e Livro não seria correto porque estaria representando informação que já existe no modelo (mesmo que de forma indireta). Além disso, uma nova associação entre Venda e Livro poderia associar *qualquer* venda com *qualquer* livro, não apenas aqueles que já estão presentes nos itens da venda, o que permitiria a representação de informações inconsistentes.

A solução de modelagem, nesse caso, quando for relevante ter acesso a uma associação que pode ser derivada de informações que já existem, é o uso de uma associação derivada, como representado na Figura 7.19.

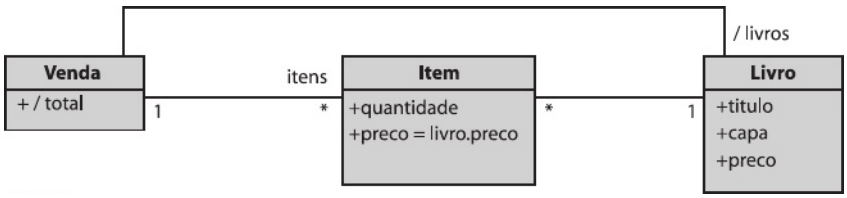


Figura 7.19: Uma associação derivada.

Uma associação derivada só tem papel e multiplicidade em uma direção (no caso, de Venda para Livro). Na outra direção ela é indefinida. Ao contrário de associações comuns que podem ser criadas e destruídas, as derivadas só podem ser consultadas (assim como os atributos derivados, elas são *read-only*). A forma de implementar uma associação derivada varia e otimizações podem ser feitas para que ela não precise ser recalculada a cada vez que for acessada.

Uma associação derivada pode ser definida em OCL. O exemplo da Figura 7.19 poderia ser escrito assim:

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

Em relação a essa expressão OCL, pode-se observar que:

a) o contexto é a própria associação derivada a partir da classe Venda conforme definido no diagrama;
b) usa-se "derive" como no caso de atributos derivados. O que define se é um atributo ou associação derivada é o contexto e o tipo de informação, já que atributos são alfanuméricos e associações definem conjuntos de objetos;
c) "self" denota uma instância do contexto da expressão OCL. No caso, qualquer instância de Venda;
d) "." é uma notação que permite referenciar uma propriedade de um objeto.

Propriedades que podem ser referenciadas pela notação "." são:

a) atributos;
b) associações;
c) métodos.

Na modelagem conceitual, usualmente faz-se referência apenas a atributos e associações. Assim, se o contexto é Venda, então self.total representa o atributo total de uma instância de Venda e self.itens representa um conjunto de instâncias de Item associadas à venda pelo papel itens.

Quando a notação "." é usada sobre uma coleção, ela denota a coleção das propriedades de todos os elementos da coleção srcinal. Assim, no contexto de Venda, a expressão "self.itens.titulo" referencia o conjunto de títulos (*strings*) dos itens de uma venda. Já a expressão self.itens.livro, que aparece na definição da associação derivada da Figura 7.19, representa o conjunto das instâncias de Livro associados às instâncias de Item associados a uma instância de Venda.

7.4.5. Coleções

Coleções de objetos não devem ser representadas como conceitos no modelo conceitual, mas como associações. De fato, uma associação com multiplicidade * representa um conjunto de objetos da classe referenciada. Assim, no exemplo da Figura 7.17, "frota" é um papel que associa um conjunto de automóveis a um dono. A modelagem da Figura 7.20 é, portanto, inadequada e desnecessária.

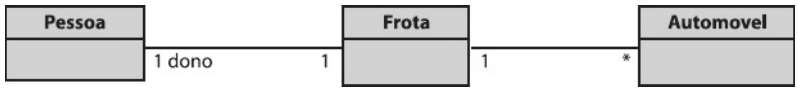


Figura 7.20: Uma coleção inadequadamente representada como conceito.

Assim, a forma correta de representar um conjunto ou coleção de objetos é através de um papel de associação, como na Figura 7.17 e não como um conceito (Figura 7.20).

Associações podem representar mais do que conjuntos; podem representar *tipos abstratos de dados*. Na verdade, podem também representar tipos concretos, mas estes não são importantes na atividade de análise, sendo utilizados apenas na atividade de projeto.

Para lembrar: tipos *abstratos* de dados são definidos apenas pelo seu comportamento. É o caso, por exemplo, do conjunto (*set*), onde elementos não se repetem e onde não há ordem definida; do multiconjunto (*bag*), onde

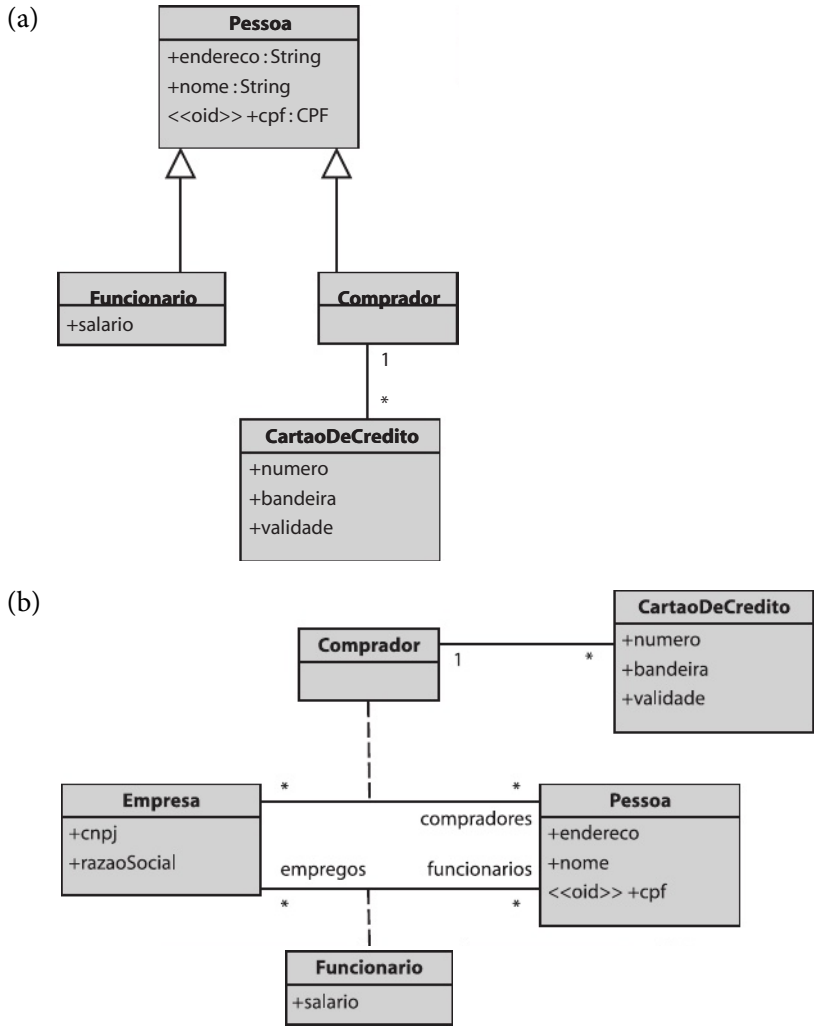


Figura 7.35: (a) Forma inadequada de representar papéis como herança. (b) Forma correta de representar papéis como classes de associação.

Portanto, quando uma mesma entidade pode representar diferentes papéis em relação a outras entidades, não se deve usar subclasses, mas classes de associação como solução de modelagem.

Para diferenciar a situação na qual se usa herança e a situação na qual se usa classe de associação, deve-se verificar se os subtipos do conceito considerado existem em função de um terceiro tipo ou não. Caso o subtipo só exista

O atributo suspenso tem valor booleano e, se for verdadeiro, o endereço não pode ser usado para entregas.

A transição é considerada *estável* porque apenas o valor do atributo muda. A estrutura interna do objeto não é alterada, como nos dois subcasos seguintes.

7.5.3.2. Transição Monotônica

A situação é um pouco mais complexa quando, em função dos diferentes estados, o conceito pode adquirir diferentes atributos ou associações. Por exemplo, pagamentos no estado pendente têm apenas vencimento e valorDevido. Já os pagamentos no estado liquidado têm adicionalmente os atributos dataDePagamento e valorPago (Figura 7.38).

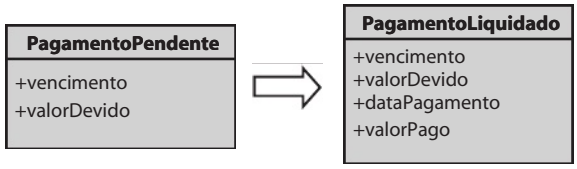


Figura 7.38: Um conceito com transição monotônica.

Diz-se que a transição de estado no caso da Figura 7.38 é monotônica porque novos atributos ou associações são acrescentados, mas nada é retirado. Isso implica que um pagamento liquidado não pode retroceder e se tornar novamente um pagamento pendente.

Seria incorreto modelar essa situação com herança de propriedades, como na Figura 7.39, pois, nesse caso, uma instância de PagamentoPendente só poderia se tornar uma instância de PagamentoLiquidado se fosse destruída e novamente criada, com todos os seus atributos (inclusive os comuns) novamente definidos. Essa forma não é muito prática e exige, quando implementada, mais processamento do que se poderia esperar, visto que, além dos atributos, várias associações poderão ter de ser refeitas.

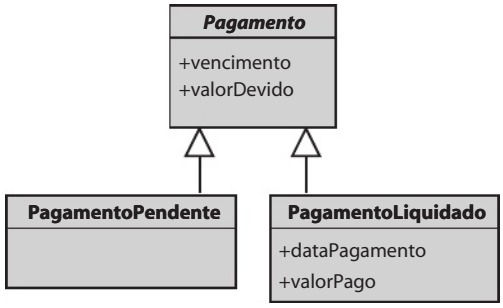


Figura 7.39: Forma inconveniente de modelar estados monotônicos com herança.

Outra solução não muito prática, mas ainda muito usada, consiste em criar uma única classe Pagamento e fazer com que certos atributos sejam nulos até que a classe mude de estado. Essa situação é indicada na Figura 7.40. Usualmente, a verificação da consistência da classe é feita nos métodos que a atualizam. Mas também poderia ser usada uma invariante (conforme explicado adiante) para garantir que nenhuma instância entre em um estado inválido, como, por exemplo, comvalorPago definido e dataDePagamento indefinida.

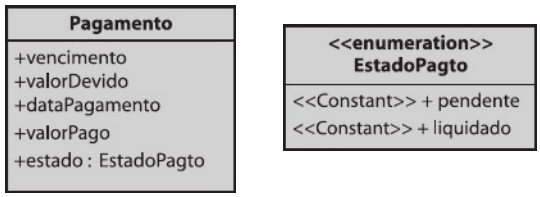


Figura 7.40: Modelagem inconveniente de estados monotônicos com uma única classe com atributos possivelmente nulos.

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

Melhor seria modelar o conceito de pagamento de forma que o controle da consistência do objeto fosse feito através da própria estrutura do modelo. Como se trata de uma transição monotônica, é possível modelar essa situação simplesmente desdobrando o conceito srcinal de pagamento em dois: um que representa o pagamento em aberto e outro que representa apenas os atributos ou associações adicionadas a um pagamento quando ele é liquidado.

Esses dois conceitos são, então, ligados por uma associação simples com multiplicidade de papel 1 no lado do conceito srcinal e 0..1 no lado do conceito que complementa o srcinal (Figura 7.41).

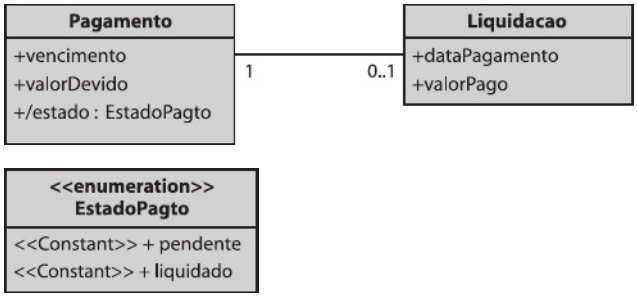


Figura 7.41: Forma eficaz de modelar classes modais com transição monotônica.

Com a modelagem indicada na Figura 7.41, percebe-se que é impossível que um pagamento não liquidado tenha dataDePagamento ou valorPago definidos. Já o estado do pagamento pode ser definido como um atributo derivado da seguinte forma: se existe uma associação com liquidacao a partir da instância de Pagamento, então o estado é liquidado, caso contrário, o estado é pendente. Em OCL:

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

Em relação à notação, verifica-se que:

a) a OCL possui estruturas de seleção na forma *if-then-else-endIf*. Se a expressão após o if for verdadeira, então a expressão toda é avaliada como a parte que vem entre o then e o else, caso contrário ela é avaliada como a parte que vem entre o else e o endIf;

b) a função isNull() aplicada a uma propriedade retorna true se ela é indefinida (ou seja, null) e false caso contrário;

c) a referência a uma constante de enumeração em OCL é feita usando-se o nome da enumeração seguido de "::" e o nome da constante, como EstadoPagto::pendente.

7.5.3.3. Transição Não Monotônica

Na transição monotônica, cada vez que um objeto muda de estado, ele pode adquirir novos atributos ou associações que não possuía antes. Contudo, se, além disso, o objeto puder ganhar e perder atributos ou associações, a transição é dita *não monotônica*.

Felizmente, é raro que em algum sistema se deseje *perder* alguma informação. Mas, às vezes, por questões práticas, isso é exatamente o que precisa acontecer.

Existem várias maneiras de se conceber e modelar um sistema de reservas em um hotel. Uma delas consiste em entender a hospedagem como uma entidade que evolui a partir de uma reserva da seguinte forma:

a) inicialmente, um potencial hóspede faz uma reserva indicando os dias de chegada e saída, o tipo de quarto e o número de pessoas. O hotel lhe informa a tarifa;
b) quando o hóspede faz o *checkin*, é registrado o dia de chegada (pode eventualmente ser diferente do dia previsto na reserva). O hotel lhe atribui um quarto, que eventualmente pode até ser diferente do tipo inicialmente reservado e, se for o caso, informa a nova tarifa. A data de saída prevista continua existindo, embora seu valor possa ser mudado no momento do *checkin;*
c) quando o hóspede faz o *checkout*, deixa de existir a data prevista de saída para passar a existir a data de saída de fato; nesse momento, a conta precisa ser paga.

Esse conjunto de estados poderia ser modelado na fase de concepção por um diagrama de máquina de estados como o da Figura 7.42.

Figura 7.42: Uma máquina de estados para modelar uma hospedagem.

Se a hospedagem apenas adquirisse novos atributos e associações à medida que seus estados evoluem, ela poderia ser representada por uma sequência de conceitos ligados por associações de 1 para 0..1, como na Figura 7.43.

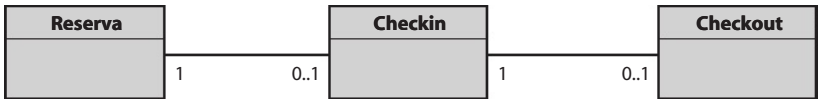


Figura 7.43: Possível modelagem de estados de uma reserva caso fosse monotônica.

7.6.1. Coesão Alta

Um dos padrões mais fundamentais consiste na definição de conceitos de boa qualidade, ou seja, coesos. Um conceito coeso é mais estável e reusável do que um conceito não coeso, que pode se tornar rapidamente confuso e difícil de manter. A maioria dos sistemas poderia ter suas informações representadas em um único "tabelão" com baixíssima coesão, mas isso não seria nada prático.

Já foi mencionado que conceitos não devem ter atributos de outros conceitos (um automóvel não deve ter como atributo o CPF de seu dono). Atributos também não devem ser tipados com estruturas de dados (listas, conjuntos etc.), pois isso é uma evidência de baixa coesão (uma classe com atributos desse tipo estaria representando mais do que um único conceito). Por exemplo, uma Venda não deveria ter um atributo listaDeItens, pois os itens devem aparecer como um conceito separado ligado à Venda por uma associação de 1 para *.

Além disso, é importante que conceitos tenham atributos que sejam efetivamente compostos por uma estrutura simples e coesa. Quando alguns atributos podem ser nulos dependendo do valor de outros atributos, isso é sinal de baixa coesão. Restrições complexas poderão ser necessárias para manter o conceito consistente. Isso equivale a usar fita adesiva para tentar manter juntos os cacos de um vaso quebrado.

(a) (b)

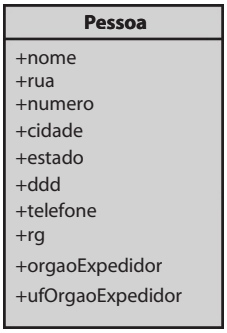
Figura 7.45: (a) Uma classe com baixa coesão por ter atributos dependentes de outros. (b) Uma solução de modelagem com classes mais coesas.

Na Figura 7.45a, os atributos valorPago e dataPagamento são mutuamente dependentes: ou ambos são nulos ou ambos são definidos. Uma restrição ou invariante de classe teria de estabelecer isso como regra para evitar que

instâncias inconsistentes surgissem. Mas uma forma melhor de modelar essa situação é mostrada na Figura 7.45b, na qual os conceitos Venda e Pagamento aparecem individualmente mais coesos. Nesse caso, não há mais atributos dependentes entre si.

Outro problema potencial é a existência de grupos de atributos fortemente correlacionados, como na Figura 7.46a, na qual se observa que grupos de atributos se relacionam mais fortemente entre si do que com outros, como rua, numero, cidade e estado, que compõe um endereço, ou ddd e telefone, que fazem parte de um telefone completo, ou ainda rg, orgaoExpedidor e ufOrgaoExpedidor, que são atributos do documento de identidade da pessoa.

(a)



(b)

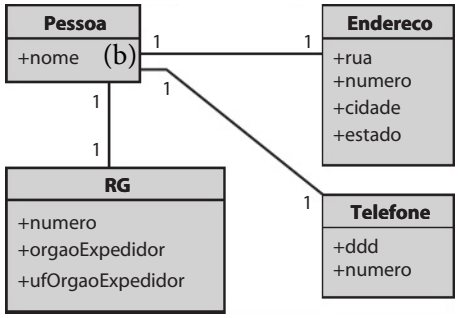
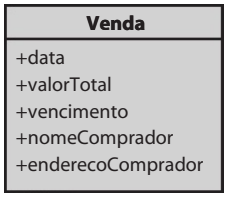


Figura 7.46: (a) Uma classe com baixa coesão por ter grupos de atributos fortemente correlacionados. (b) Uma solução com melhor coesão.

A solução para melhorar a coesão mostrada na Figura 7.46b também abre caminho para outras possibilidades de modelagem, como, por exemplo, permitir que uma pessoa tenha mais de um endereço ou mais de um telefone, caso as associações sejam trocadas por associações de um para muitos.

Outra situação ainda ocorre quando determinados atributos repetem sempre os mesmos valores em diferentes instâncias. A Figura 7.47a apresenta um exemplo.

(a)



(b)

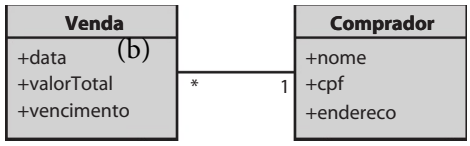


Figura 7.47: (a) Uma classe com baixa coesão por ter atributos que repetem valores nas instâncias. (b) Uma solução com melhor coesão.

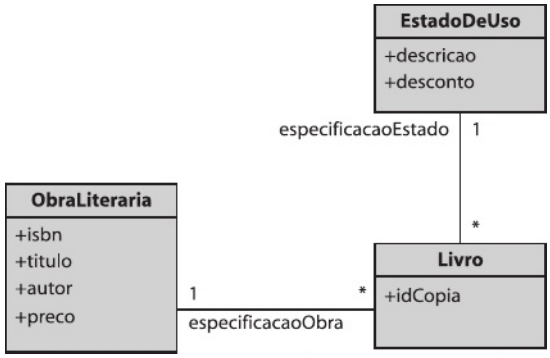


Figura 7.49: Uma classe com duas classes de especificação.

7.6.3. Quantidade

Frequentemente, o analista se depara com a necessidade de modelar quantidades que não são meramente números. O peso de um livro, por exemplo, poderia ser definido como 400. Mas 400 o quê? Gramas? Libras? Quilos? Uma solução é definir um tipo específico para o peso e então usá-lo sempre consistentemente. O atributo, então, seria declarado como peso:Gramas. Mas isso exige que o peso de todos os livros seja expresso em gramas. Se a informação vier em outra unidade, terá de ser convertida ou estará inconsistente.

Em alguns casos, espera-se que seja possível configurar o sistema informatizado para suportar diferentes medidas. Em alguns países se usam gramas, e em outros, libras. Se a classe for modelada com gramas, o sistema terá de ser refeito para aceitar libras.

Porém, o padrão "*Quantidade*" permite que diferentes sistemas de medição coexistam sem conflito e sejam facilmente intercambiáveis. O padrão consiste na criação de um novo tipo de dados primitivo Quantidade, com dois atributos, como mostrado na Figura 7.50.

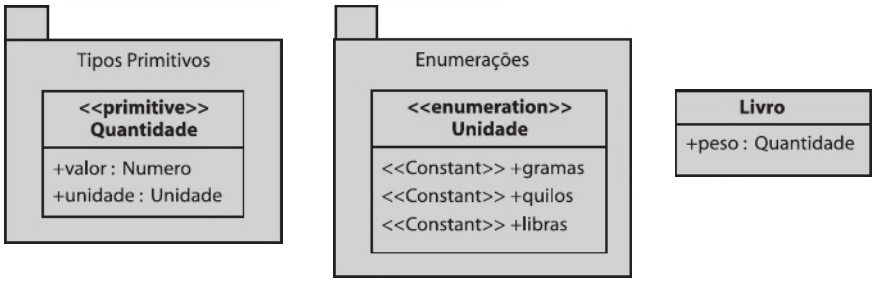


Figura 7.50: Definição e uso de Quantidade.

Dessa forma, o peso de cada livro será especificado como uma quantidade formada por um valor numérico e uma unidade, que corresponde a uma enumeração dos valores quilos, gramas e libras.

Caso se necessite estabelecer *razões de conversão* entre unidades, uma opção seria transformar a enumeração Unidade em uma classe normal e criar uma classe Razao associada a duas unidades: srcem e destino, como na Figura 7.51. Quando uma quantidade da unidade srcem tiver de ser convertida em uma quantidade da unidade destino, divide-se seu valor pelo valor da razão.

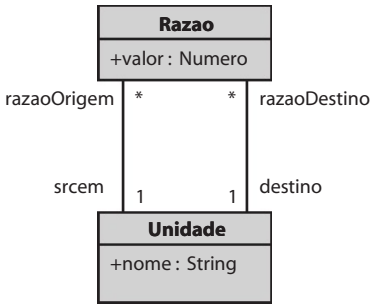


Figura 7.51: Unidades com razão de conversão.

Assim, por exemplo, a instância de Unidade, cujo nome é "gramas" pode estar ligada a uma instância de Razao, que por sua vez liga-se a uma instância de Unidade cujo nome é "quilos". O valor dessa instância de Razao será então 1000 porque, para converter uma quantidade em gramas para uma quantidade em quilos, deve-se dividir por 1.000.

7.6.4. Medida

Uma evolução do padrão *Quantidade* é o padrão *Medida*, que deve ser usado quando for necessário realizar várias medidas diferentes, possivelmente em tempos diferentes a respeito de um mesmo objeto. Por exemplo, uma pessoa em observação em um hospital pode ter várias medidas corporais sendo feitas de tempos em tempos: temperatura, pressão, nível de glicose no sangue etc. Milhares de diferentes medidas poderiam ser tomadas, mas apenas umas poucas serão efetivamente tomadas para cada paciente. Então, para evitar a criação de um conceito com milhares de atributos dos quais a grande maioria permaneceria nulo, a opção é usar o padrão *Medida*, como na Figura 7.52.

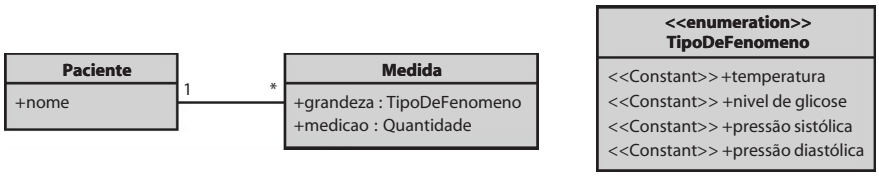


Figura 7.52: Definição e uso do padrão Medida.

Assim, um paciente terá uma série de medidas tomadas, cada uma avaliando um tipo de fenômeno e apresentando um valor que corresponde a uma quantidade (conforme padrão *Quantidade*).

Ainda é possível sofisticar mais uma medida adicionando atributos para indicar o instante do tempo em que a medida foi tomada e, também, o prazo de validade da medida. Por exemplo, o fato de que um paciente tinha febre há duas horas não continua necessariamente sendo verdadeiro no presente, ou seja, a medida já pode estar inválida.

7.6.5. Estratégia

Foi mencionado que um dos desafios dos requisitos é estar preparado para sua mudança. Especialmente os requisitos transitórios (aqueles que se prevê que vão mudar) devem ser acomodados no projeto do sistema de forma que sua mudança, quando ocorrer, minimize o impacto das alterações sobre o sistema e, consequentemente, seu custo.

Alguns casos são relativamente fáceis de tratar. Por exemplo, se houver uma previsão de que a moeda corrente do país poderá mudar (isso aconteceu muitas vezes entre 1980 e 1994), basta usar o padrão *Quantidade* ou, simplesmente, tratar o tipo de moeda como um parâmetro de sistema que pode ser alterado.

Mas há situações mais complexas. Por exemplo, a forma de calcular impostos pode variar muito. Há impostos que são calculados sobre o preço de venda dos produtos, outros são calculados sobre o lucro, outros são calculados sobre a folha de pagamento. As formas variam e, historicamente, uma quantidade significativa de novos impostos é criada ao longo de um ano. Os sistemas devem estar preparados para isso, mas as mudanças são completamente imprevisíveis.

truturas organizacionais em hierarquias diferentes são equivalentes ou, ainda, que uma é sucessora de outra.

As subseções seguintes vão apresentar as principais estratégias para lidar com este tipo de situação.

7.6.7.1. Copiar e Substituir

A primeira estratégia em que se pensa quando é necessário juntar dois objetos que na verdade são um só consiste em copiar os dados de um sobre o outro (*copy and replace* ou *copiar e substituir*). A operação de cópia deve ser definida por contrato (ver Capítulo 8) e o analista deve definir, para cada atributo e cada associação, o que deve acontecer durante a cópia. Regras serão definidas para dizer se um atributo será copiado sobre outro, se seus valores serão somados ou se o maior dentre eles deve permanecer etc. Quanto às associações, o analista deve decidir o que acontece: se uma associação sobrescreve outra, se elas se adicionam e assim por diante.

O registro da data da última inclusão ou alteração de um conceito pode ser uma ferramenta útil para que se tenha como decidir qual atributo manter no caso de conflito. Por exemplo, um comprador cadastrado duas vezes para o qual constam dois endereços diferentes possivelmente deverá manter apenas o registro do endereço mais recente. Por outro lado, todas as compras que esse comprador tenha efetuado devem ser agrupadas na instância resultante.

Depois de efetuar a cópia ou junção dos dados de uma instância sobre a outra, a instância que foi copiada deve ser destruída e quaisquer referências a ela devem ser redirecionadas para a instância que recebeu os dados da cópia.

7.6.7.2. Sucessor

Sucessor (*Superseding*) é uma técnica que pode ser usada quando se pretende manter o objeto srcinal sem destruí-lo. Aplica-se *Sucessor*, por exemplo, no caso de estruturas organizacionais que se sucedem no tempo. Supondo que os departamentos de *venda* e *marketing* sejam unidos em um único departamento de *contato com clientes*, os departamentos srcinais devem ser mantidos mas marcados como não mais ativos, e o novo departamento deve ser marcado como sucessor deles. Implementa-se a estratégia *Sucessor* através de uma associação reflexiva, como na Figura 7.56.

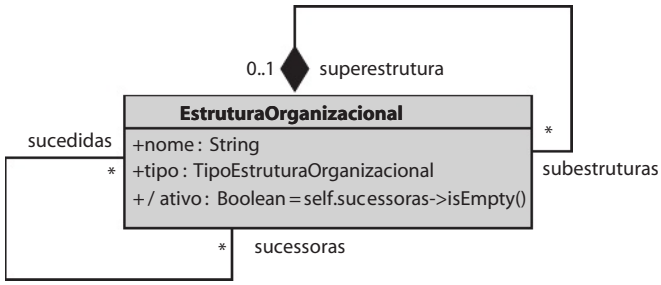


Figura 7.56: Um exemplo de aplicação da estratégia *Sucessor*.

O atributo derivado ativo é true se o conjunto representado pelo papel sucessoras é vazio e false caso contrário.

Manter a estrutura srcinal, mesmo que ela não seja mais ativa, pode ser importante para fins de registro. Algum dia, alguém pode querer saber quanto se gastava por mês no antigo departamento de marketing, quanto se gastava no antigo departamento de vendas e quanto se gasta atualmente com o departamento de contato com clientes.

Pode ser útil adicionar uma classe de associação à associação sucessoras/sucedidas cujos atributos poderiam indicar, entre outras coisas, a data em que houve o evento de sucessão.

7.6.7.3. Essência/Aparência

Outra situação que ainda pode surgir com frequência é a existência de objetos equivalentes dos quais se queira manter a individualidade. Não se trata nesse caso de um objeto que sucede a outro, como em Sucessor, mas de objetos que são considerados equivalentes.

Pode-se ter, em alguns casos, diferentes manifestações de um mesmo objeto, mas uma única essência por trás. Quando algo muda na essência, muda também em todas as manifestações ou aparências.

Essa técnica pode ser modelada com a criação de um *objeto essência* para ser associado a um conjunto de objetos equivalentes. Diferentemente da técnica *Copiar e Substituir*, os objetos srcinais são mantidos, e diferentemente da técnica *Sucessor*, não há um objeto ativo e um objeto sucedido: todos os objetos são equivalentes. A Figura 7.57 mostra um exemplo de modelagem de uma classe que aceita que seus membros tenham objetos essência. Objetos são

Uma maneira de implementar a possibilidade de desfazer junções, bem como quaisquer outras operações, é manter um "*log*" de banco de dados, no qual cada modificação em um registro é anotada, sendo mantido, em uma tabela à parte, o valor anterior e o novo valor de cada campo de cada tabela alterada, bem como a hora exata e o usuário responsável pela modificação. Dessa forma, quaisquer operações podem ser desfeitas, mas ao custo de maior uso de armazenamento de dados.

7.6.8. Conta/Transação

Um padrão de cunho eminentemente comercial, mas de grande aplicabilidade, é o padrão *Conta/Transação* Foi mencionado anteriormente que livros podem ser encomendados, recebidos, estocados, vendidos, entregues, devolvidos, reenviados e descartados. Tais movimentações, bem como as transações financeiras envolvidas, poderiam dar srcem a uma série de conceitos como Pedido, Compra, Chegada, Estoque, Venda, Remessa, Devolução, ContasAReceber, ContasAPagar etc., cada um com seus atributos e associações.

Porém, é possível modelar todos esses conceitos com apenas três classes simples e poderosas.

Uma *conta* é um local onde são guardadas quantidades de alguma coisa (itens de estoque ou dinheiro, por exemplo). Uma conta tem um *saldo* que, usualmente, consiste no somatório de todas as retiradas e depósitos.

Por outro lado, retiradas e depósitos, frequentemente, são apenas movimentações de bens ou dinheiro de uma conta para outra. Assim, uma *transação* consiste em duas movimentações, uma retirada de uma conta e um depósito de igual valor em outra. A Figura 7.58 ilustra essas classes.

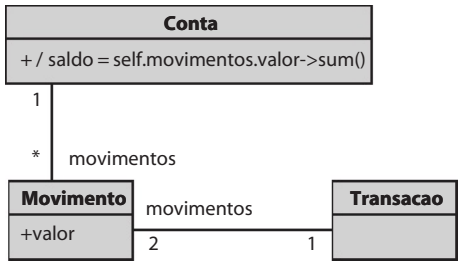


Figura 7.58: Classes do padrão *Conta/Transação.*

Para a classe Transacao ser consistente, é necessário que ela tenha exatamente dois movimentos de mesmo valor absoluto mas sinais opostos. Ou seja, se a transação tira cinco reais de uma conta, ela coloca cinco reais em outra conta. Então, a classe Transacao necessitaria de uma invariante (assunto da Seção 7.7) como a seguinte:

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

Ou seja, para quaisquer instâncias de Transacao, a soma dos dois movimentos associados a ela tem de ser zero.

Por outro lado, o atributo derivado /saldo da classe Conta é definido como o somatório de todos os movimentos daquela conta.

Então, as várias situações relacionadas a pedidos de livros podem ser modeladas a partir de um conjunto de instâncias da classe Conta. Por exemplo:

a) para cada fornecedor (editora) corresponde uma instância de Conta da qual somente são retirados livros, ou seja, essa é uma *conta de entrada* e seu saldo vai ficando cada vez mais negativo à medida que mais e mais livros são encomendados;
b) há uma conta para *saldo de pedidos*, que contém os livros pedidos mas ainda não entregues;
c) há uma conta para *estoque* contendo os pedidos entregues e ainda não vendidos;
d) há uma conta de *remessa* contendo os livros vendidos mas ainda não enviados;
e) há uma conta de *envio*, contendo livros enviados mas cuja entrega ainda não foi confirmada;
f) há uma conta de *venda confirmada* contendo os livros vendidos e cuja entrega foi confirmada pelo correio (possivelmente uma para cada comprador). Essa é uma conta de saída, cujo saldo vai ficando cada vez mais positivo à medida que transações são feitas. Seu saldo representa a totalidade de livros já vendidos.

Paralelamente, há contas para as transações em dinheiro feitas concomitantemente. Haverá contas a receber, contas a pagar, contas recebidas e pagas, investimentos, dívidas, valores separados para pagamento de impostos etc.

O problema com essa abordagem é que apenas *naquele* método seria feita a verificação, mas não fica uma regra geral para ser observada em outros métodos. Até é possível que o analista hoje saiba que a regra deve ser seguida, mas, e se outro analista fizer a manutenção do sistema dentro de cinco ou 10 anos? Ele não saberá necessariamente que essa regra existe, provavelmente não vai consultar o documento de requisitos, já desatualizado, e poderá introduzir erro no sistema se permitir a implementação de métodos que não obedeçam à regra.

Então, todas as regras gerais para o modelo conceitual devem ser explicitadas no modelo para que instâncias inconsistentes não sejam permitidas. Se for possível, as restrições devem ser explicitadas graficamente; caso contrário, através de invariantes.

Outro exemplo, que ocorre com certa frequência, é a necessidade de restringir duas associaçõesque a princípio são independentes. Na Figura 7.65, considera-se que cursos têm alunos, cursos são formados por disciplinas e alunos matriculam-se em disciplinas, mas o modelo mostrado nafigura não estabelece que alunos só podem se matricular nas disciplinas de seu próprio curso.

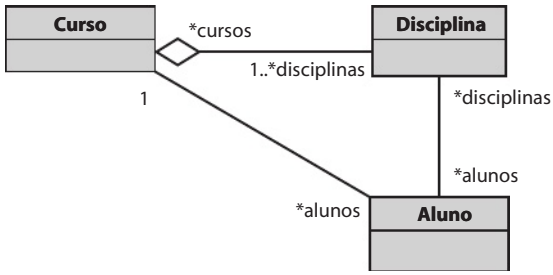


Figura 7.65: Uma situação que necessita de uma invariante para que a consistência entre associações se mantenha.

Para que um aluno só possa se matricular em disciplinas que pertencem ao curso ao qual está associado, é necessário estabelecer uma invariante como:

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

A invariante diz que, para todas as disciplinas (`d`) cursadas por um aluno (`self`), o conjunto de cursos nos quais a disciplina é oferecida contém o curso no qual o aluno está matriculado.

A mensagem forAll afirma que uma expressão lógica é verdadeira para todos os elementos de um conjunto; no caso, o conjunto dado por self.disciplina.

A variável "d" entre parênteses equivale a um *iterador*, ou seja, "d" substitui na expressão lógica cada um dos elementos do conjunto. A mensagem includes corresponde ao símbolo matemático de pertença invertida (∋), ou seja, afirma que um determinado conjunto contém um determinado elemento.

É possível, ainda, simplificar a expressão eliminando a variável self e o iterador d, visto que podem ser inferidos pelo contexto em que se encontram. A expressão anterior poderia, então, ser escrita assim:

```
Context Aluno
 inv:
   disciplinas→forAll(cursos→includes(curso))
```

7.8. Discussão

Um bom modelo conceitual produz um banco de dados organizado e normalizado. Um bom modelo conceitual incorpora regras estruturais que impedem que a informação seja representada de forma inconsistente. Um bom modelo conceitual vai simplificar o código gerado porque não será necessário fazer várias verificações de consistência que a própria estrutura do modelo já garante.

O uso de padrões corretos nos casos necessários simplifica o modelo conceitual e torna o sistema mais flexível e, portanto, lhe dá maior qualidade. É, dessa maneira, uma ferramenta poderosa. Muitos outros padrões existem e os analistas podem descobrir e criar seus próprios padrões. Apenas é necessário sempre ter em mente que só vale a pena criar um padrão quando os seus benefícios compensam o esforço de registrar sua existência.

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 verdadeiros.

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, salvo melhor juízo, tal sistema ainda não está disponível nas ferramentas *CASE* comerciais.

8.4.7. Combinações de Pós-condições

Cada operação de sistema terá 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>
```

Além disso, é possível utilizar 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:
```

Figura 10.33: Modelo WebML parcial para o diagrama da Figura 10.32.

Continuando o exemplo, percebe-se na Figura 10.34 que o passo seguinte, caso a identificação do comprador tenha sido feita com sucesso, é apresentar a lista de livros disponíveis.

Figura 10.34: Continuação do diagrama de sequência.

Isso pode ser feito através de uma *index unit* seguindo o padrão "Listagem com Filtro". São apresentados os principais dados dos livros disponíveis (Figura 10.35). Como o usuário poderá escolher mais de um livro, deve-se usar uma *multi-choice index unit*

Figura 10.35: Continuação da modelagem introduzindo uma multi-choice index unit.

Possivelmente, tal lista será muito longa, e o projetista, nesse ponto, poderá pensar em usar o padrão "Índice Filtrado" para apresentar apenas alguns livros com base em uma escolha de palavra-chave por parte do comprador. Mas, por ora, ficará assimmesmo.

Além disso, a *multi-choice index unit* deverá passar, além do ISBN de cada livro escolhido, para a quantidade solicitada. O padrão *multi-choice index unit* não permite realizar diretamente essa operação. Deveriam ser definidas *entry units* para as quantidades sempre que uma opção fosse selecionada. Considerando-se que esse tipo de interação é bastante comum em sistemas Web, pode-se optar pela definição de um novo estereótipo de *unit*: uma *multi-choice index unit* com quantidade. Ela não será definida aqui, mas pode-se atribuir a ela uma representação simbólica como a da Figura 10.35, no qual aparece com a marca # para indicar que cada elemento selecionado é complementado com uma quantidade.

Na continuação (Figura 10.36), percebe-se que, após o usuário selecionar os livros e quantidades, deve ser executada a operação não padronizada adicionaCompra. E logo após, o total da compra será exibido.

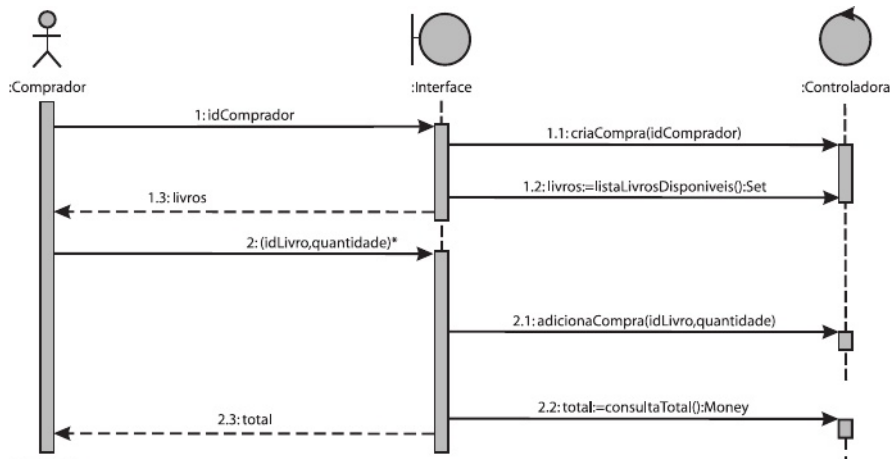


Figura 10.36: Continuação do diagrama de sequência.

Para exibir um atributo derivado da compra, é possível usar uma *data unit*, como na Figura 10.37.

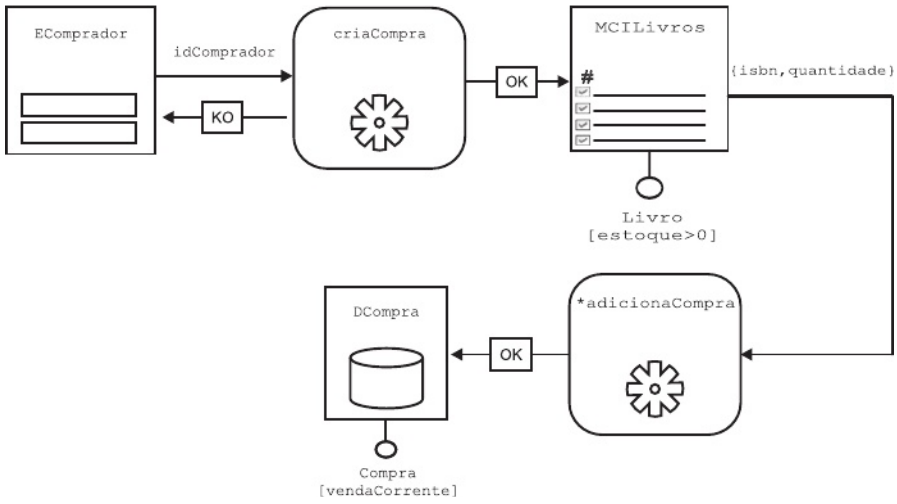


Figura 10.37: Continuação da modelagem introduzindo uma nova *operation unit* e uma *data unit*.

A *unit* DCompra deveria ser definida, a princípio, apenas com o atributo derivado valorTotal:

```
DataUnit DCompra (
 source Compra;
 selector CompraCorrente;
 attributes valorTotal
)
```

E assim por diante. A modelagem continua até que todas as operações e consultas definidas no diagrama de sequência do fluxo principal do caso de uso, bem como dos diagramas de sequência dos fluxos alternativos, estejam representadas.

Em relação à modelagem WebML "pura" e à forma apresentada neste livro, nota-se uma significativa vantagem na segunda: a modelagem WebML pura faz com que as operações básicas individuais (criação e destruição de instância, criação e destruição de associação e modificação de valor de atributo) sejam representadas nos diagramas WebML, o que poderia deixá-los bastante confusos. A abordagem usada aqui sugere que, em vez de representar essas operações básicas individuais no diagrama WebML, se usem as *operation units* genéricas a partir das operações e consultas de sistema identificadas no diagrama de sequência. Essas *operation units* estarão encapsulando todo um conjunto de operações básicas (de acordo com seus contratos) e, dessa forma, deixam os diagramas mais legíveis.

Capítulo

11

Persistência

A disponibilização de *frameworks* de persistência para linguagens comerciais (ver, por exemplo, http://docs.jboss.org/hibernate/stable/annotations/reference/en/pdf/hibernate_annotations.pdf) tornou praticamente secundária uma atividade de projeto de persistência que envolva o projeto especificamente de tabelas relacionais, formato de campos, restrições estruturais etc. Com o uso de ferramentas adequadas, é possível gerar a camada de persistência automaticamente a partir do projeto da camada de domínio. Eventualmente, será necessário efetuar alguns ajustes no mecanismo de persistência para acomodar objetos com características especiais ou para satisfazer requisitos de *performance* ou segurança de dados.

Assim, cabe ao projetista indicar qual a ferramenta de persistência a ser usada e apresentar as indicações necessárias para que essa ferramenta possa ser usada de forma profícua.

O objetivo deste capítulo é, então, indicar ao leitor o que acontece na camada de persistência quando se usa um *framework* desse tipo. Mas os aspectos discutidos aqui não precisam ser necessariamente redefinidos a cada projeto.

Em primeiro lugar, é interessante deixar claro que um mecanismo de persistência se baseia em uma ideia de separação entre interface, domínio e

persistência. Todas as operações e consultas da interface são realizadas através da camada de domínio, e não através de expressões SQL.

Cabe ao mecanismo de persistência, então, garantir que os objetos sejam salvos em um dispositivo de memória secundária e que de lá sejam recarregados quando necessário.

11.1. Equivalência entre Projeto Orientado a Objetos e Modelo Relacional

O DCP permite a geração automática de uma estrutura de banco de dados relacional que reflete, em memória secundária, a informação que os objetos representam em memória primária. Para isso é necessário seguir algumas regras de equivalência que são apresentadas nas próximas subseções.

11.1.1. Classes e Atributos

O primeiro conjunto de regras trata das classes e seus atributos. Cada classe do DCP equivale a uma *tabela relacional*. Cada atributo de uma classe equivale a uma *coluna* na tabela respectiva. Cada instância de uma classe equivale a uma *linha* na tabela respectiva. Identificadores de objetos são representados como *colunas indexadas* que não admitem repetição de elementos, sendo, portanto, marcadas com a expressão unique (Figura 11.1).

(a)

(b)

Tabela: Livro					
pkLivro<<pk>>	**isbn<<unique>>**	**titulo**	**editora**	**autor**	**nrPaginas**
10001	12345	análise e projeto	campus	raul	302
10002	54321	metologia de pesquisa	campus	raul	156
10003	11111	a república	acrópole	platão	205

Figura 11.1: (a) Uma classe. (b) Tabela relacional equivalente a essa classe com três instâncias representadas.

A representação relacional, além dos atributos da classe, terá uma coluna consistindo em uma chave primária (pk, de *primary key*), tratando-se de um código inacessível e sem qualquer significado para a camada de domínio. O valor da chave primária deve ser conhecido, portanto, apenas na camada de persistência.

11.1.2. Associações de Muitos para Muitos

As associações entre as classes (exceto as temporárias) corresponderão a *tabelas associativas* no modelo relacional, ou seja, tabelas com uma chave primária composta pelas chaves primárias de duas outras tabelas.

Conforme o tipo de multiplicidade dos papéis da associação, algumas regras devem ser observadas. Se nenhum dos lados da associação tiver multiplicidade 1, nenhuma das colunas que formam a chave primária será marcada com unique (Figura 11.2). Suponha que, além dos três livros da Figura 11.1, houvesse três pessoas representadas na tabela apropriada (Figura 11.2c). A tabela da Figura 11.2b mostra como se relacionam algumas pessoas com alguns livros de sua lista de desejos.

(a)

(b)

Tabela: listaDesejos_quemDeseja	
pkLivro<<pk>>	**pkPessoa <<pk>>**
10001	20001
10001	20003
10003	20001

(c)

Tabela> Pessoa			
pkPessoa <<pk>>	**cpf <<unique>>**	**nome**	**endereco**
20001	3637283	joão	rua não
20002	3729109	miguel	av das dores
20003	3938204	maria	rua talvez

Figura 11.2: (a) Exemplo de associação de muitos para muitos. (b) Tabela associativa que representa essa associação. (c) Tabela para a classe Pessoa.

Na Figura 11.2b, as duas colunas formam uma chave composta. Isso significa que, individualmente, elas podem repetir valores. Apenas não é possível repetir os pares de valores, pois cada par forma a chave primária da tabela.

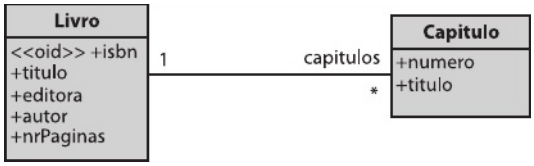
Na figura, a tabela associativa, juntamente com as tabelas das Figuras 11.1b e 11.2c, estabelece que João deseja os livros "análise e projeto" e "a república". Já Maria deseja apenas o livro "análise e projeto", e Miguel não deseja livro algum.

Nota-se que é preferível nomear a tabela associativa com os nomes de papel da associação do que com os nomes das classes, pois pode haver mais de uma associação entre as mesmas classes e, usando os nomes de papel, garante-se que não haverá ambiguidade. Usam-se os nomes das classes apenas na falta do nome de papel explícito, como no caso de expressões OCL.

11.1.3. Associações de Um para Muitos

No caso de associações de um para muitos ou de muitos para um, a tabela associativa terá a condição unique na coluna correspondente à classe do lado "muitos". Isso significa que a coluna correspondente ao lado "muitos" da associação não pode ter seus valores individuais repetidos.

(a)



(b)

Tabela: livro_capitulos	
pkLivro<<pk>>	**pkCapitulo <<pk>> <<unique>>**
10001	30001
10001	30002
10001	30003
10002	30004
10002	30005
10003	30006
10003	30007

Figura 11.3:(a) Associação de um para muitos. (b) Tabela associativa correspondente.

Na Figura 11.3b, a restrição unique na coluna da direita impede que um mesmo capítulo apareça associado a mais do que um livro.

Mais algumas observações:

a) se a associação for estritamente de um para muitos, todos os elementos da tabela do lado "muitos" *devem* aparecer na tabela associativa. No caso, todos os capítulos existentes aparecem na tabela associativa da Figura 10.3b, pois é obrigatório que um capítulo esteja associado a um livro;

b) se a associação fosse de 0..1 para muitos, nem todos os elementos da tabela Capitulo precisariam aparecer na tabela associativa;

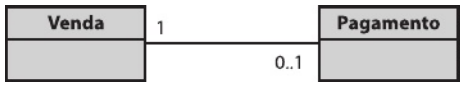
c) se a associação fosse de 1 para 1..*, ela seria obrigatória nas duas direções. Logo, todos os livros e todos os capítulos deveriam aparecer na tabela associativa.

Genericamente, considerado que *A* tem uma associação para *B* e que o limite mínimo do papel de *A* para *B* é *n*, enquanto o limite máximo é *m* (onde *m* pode ser * ou infinito), o número de vezes que cada instância de *A* aparece na tabela associativa é limitado inferiormente por *n* e superiormente por *m*. Por exemplo, se a multiplicidade de papel de *A* para *B* for 2..5, cada instância de *A* deve aparecer na tabela associativa no mínimo duas e no máximo cinco vezes.

11.1.4. Associações de Um para Um

No caso de associações de um para um, um para 0..1, 0..1 para um e 0..1 para 0..1, a tabela associativa terá unique nas duas colunas da chave primária, impedindo, com isso, que qualquer dos elementos associe-se com mais de um elemento da outra classe (Figura 11.4).

(a)

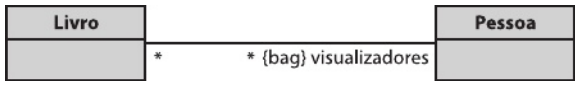


(b)

Tabela: venda_pagamento	
pkVenda <<pk>><<unique>>	**pkPagamento <<pk>><<unique>>**
50001	60001
50003	60002
50005	60003
50011	60004
50016	60005
50021	60006
50030	60007

Figura 11.4: (a) Associação um para 0..1. (b) Tabela associativa correspondente.

(a)



(b)

Tabela: livro_visualizadores		
pkLivro <<pk>>	**pkPessoa <<pk>>**	**quantidade**
10001	20001	1
10001	20002	1
10002	20001	2
10002	20003	6
10003	20001	1

Figura 11.6:(a) Associação multiconjunto (*bag*). (b) Representação dessa associação como tabela relacional.

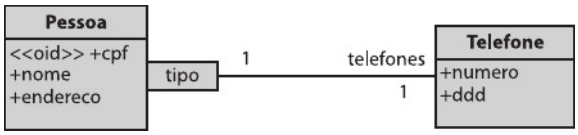
Na Figura 11.6b, Miguel (10002) visualizou o livro "a república" (20003) seis vezes. Já João (10001) visualizou o livro "análise e projeto" (20001) apenas uma vez. Não é necessário representar a quantidade zero. Por exemplo, Maria (20003) nunca visualizou "a república" (10003); então essa combinação simplesmente não deve aparecer na tabela.

11.1.7 Associações Qualificadas

No caso de associação qualificada *para um* com *qualificador interno* (o qualificador é atributo da classe), basta implementar a associação como mera associação para muitos (Figura 11.3), tomando o cuidado de fazer com que a coluna que contém o atributo qualificador seja marcada com unique na tabela que contém o conceito srcinal. Se a ferramenta de banco de dados permitir, pode-se ainda indexar o campo com o atributo qualificador para que o acesso a este seja mais rápido (Date, 1982).

Porém, quando o qualificador for *externo*, é necessário adicionar uma terceira coluna à tabela associativa que permita mapear elementos da classe destino a partir do valor do qualificador (Figura 11.7).

(a)



(b)

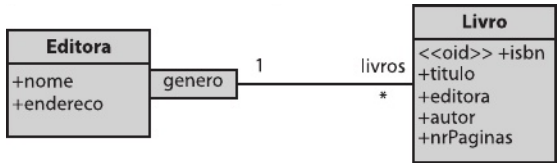
Tabela: pessoa_telefones		
pkPessoa<<pk>>	**tipo<<pk>>**	**pkTelefone <<unique>>**
20001	casa	70001
20001	celular	70002
20002	casa	70003

Figura 11.7:(a) Associação com qualificador externo. (b) Sua representação como tabela relacional.

Nesse caso, a tabela relacional associativa deve ter uma chave primária dupla, formada pelas colunas como na Figura 11.7. A pkPessoa e tipo a coluna pkTelefone não pertence à chave, mas deve ser unique visto que um telefone só pode se associar a uma única pessoa.

A situação de um qualificador externo que define uma partição, como na Figura 11.8, é implementada da mesma maneira que a associação qualificada da Figura 11.7. Porém, no caso da Figura 11.8, a chave primária deve ser tripla e a restrição que impede a repetição de pares de instâncias da classe de srcem e qualificador não deve existir.

(a)



(b)

Tabela: editora_livros		
pkEditora<<pk>>	**genero<<pk>>**	**pkLivro<<pk>><<unique>>**
60001	computação	10001
60001	computação	10002
60002	filosofia	10003

Figura 11.8:(a) Associação qualificada representando uma partição e (b) sua representação como tabela relacional.

No caso de *relações*, como na Figura 7.26, faz-se a mesma implementação, mas elimina-se o unique na coluna que representa a classe destino.

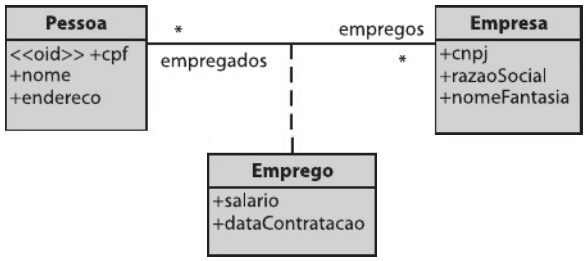
11.1.8. Classes de Associação

Uma classe de associação e sua associação correspondente são representadas em uma única tabela no banco de dados. Em primeiro lugar, cria-se uma tabela associativa para a associação (normalmente é "muitos para muitos"). Os atributos da classe de associação são acrescentados como colunas extras nessa tabela associativa. Porém, como a classe de associação pode ter suas próprias associações, pode ser inconveniente manter uma chave primária dupla para ela. Nesse caso, deve-se criar uma nova chave primária para representar as instâncias da classe de associação. Assim, a tabela associativa terá três tipos de colunas:

a) a sua própria chave primária simples;
b) duas colunas com valores tomados das chaves primárias das tabelas associadas, correspondendo a uma chave candidata;
c) os atributos da classe de associação.

A Figura 11.9 apresenta esquematicamente a equivalência entre uma classe de associação e uma tabela relacional.

(a)



(b)

Tabela: empregados_empregos				
pkEmprego <<pk>>	**pkPessoa**	**pkEmpresa**	**salario**	**dataContratacao**
80001	20001	70001	1500,00	15/02/2008
80002	20001	70005	1200,00	01/03/1998
80003	20002	70001	2000,00	16/04/2005
80004	20003	70002	900,00	17/01/2001

Figura 11.9:(a) Classe de associação e (b) sua representação como tabela associativa.

Na Figura 11.9 observa-se que o par pkPessoa/pkEmpresa é uma chave candidata da tabela, pois seus pares não devem se repetir. Mas a chave primá-

ria efetiva é pkEmprego, para que seja uma chave simples e dessa forma facilite a representação de associações entre a classe de associação Emprego e outras classes.

11.1.9. Associações Temporárias e Associações da Controladora

As associações temporárias *não são* transformadas em tabelas relacionais porque, por sua própria definição, existem apenas em memória primária, não sendo necessário nem desejável torná-las persistentes ao longo do tempo.

Já algumas associações ligadas à controladora não precisam ser persistentes porque a controladora é *singleton*. As associações de *singletons*, caso fossem transformadas em tabelas associativas, teriam sempre o mesmo valor na coluna da chave primária do lado do *singleton*, já que só existe uma única instância dele. Assim, essas tabelas são, a princípio, desnecessárias, e pode-se usar simplesmente a chave primária da tabela que representa a outra classe da associação quando se quer, a partir da controladora, localizar uma de suas instâncias.

Porém, isso só vale para as associações *um para muitos* da controladora para as classes independentes que ligam a controladora a todas as instâncias da classe. Caso se trate de associações opcionais do lado da controladora, elas devem ser implementadas, pois trazem informação nova. Por exemplo, na Figura 11.10, a associação clientes não precisa ser representada como tabela associativa, pois acessa todas as instâncias da classe Cliente e para isso basta acessar diretamente a chave primária da tabela Cliente. Já a associação clientesPremium deve ser implementada, pois nem todos os clientes pertencem a ela. Embora todas as colunas do lado Livir da tabela associativa repitam o mesmo valor (pk de Livir), nem todas as instâncias de Pessoa estarão na tabela, o que justifica que se trata de informação nova não acessível por outros meios.

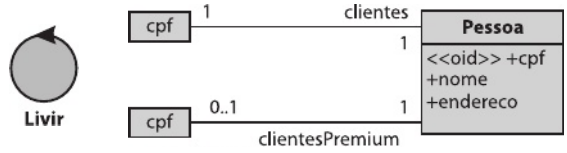


Figura 11.10: Uma associação obrigatória (clientes) e uma associação opcional (clientesPremium) para a controladora.

Além disso, conforme foi dito, as associações obrigatórias da controladora não *precisam* ser implementadas, mas isso não quer dizer que seja proibido implementá-las. Por uma questão de uniformidade de tratamento, o projetista pode optar por implementar todas as associações, inclusive essas, se julgar que será adequado para o projeto.

11.1.10. Herança

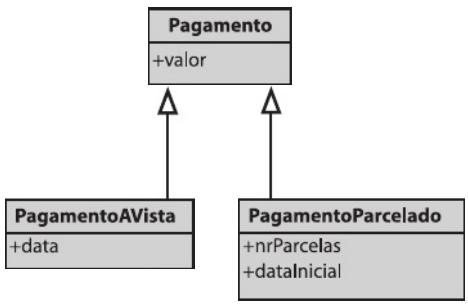
Visto que a herança é principalmente uma maneira de fatorar propriedades na especificação das classes, pode-se implementar tabelas relacionais para as subclasses de forma não fatorada, ou seja, criando cada tabela com todos os atributos da classe e atributos herdados.

Mas essa forma de representação pode ser inconveniente caso se queira fazer muitas operações com o conjunto das instâncias do ponto de vista da superclasse porque, nesse caso, seria necessário unir tabelas heterogêneas. Além disso, poderia ser complicado implementar um mecanismo de controle de tabelas associativas quando existem associações para a superclasse e para as subclasses.

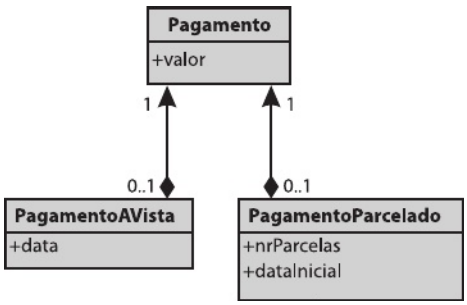
Então, outra opção que surge é a decomposição das instâncias de subclasses em tabelas com os atributos próprios das subclasses e tabelas representando as superclasses com os atributos generalizados. Essa situação (Figura 11.11a) pode ser representada por um equivalente conceitual como o modelo da Figura 11.11b. Adiciona-se ainda, ao modelo da Figura 11.11b, uma invariante na classe Pagamento:

```
Context Pagamento inv:
 pagamentoAVista→size() + pagamentoParcelado→size() = 1
```

(a)



(b)



(c)

Tabela:Pagamento	
pkPagamento< <pk>>	valor
60001	105,00
60002	430,20
60003	28,51
60004	23,22
60005	24,42
60006	345,32
60007	32,85
60008	893,89
60009	326,22

Tabela:PagamentoAVista		
pkPagamentoAVista< <pk>>	data	pkPagamento
61001	20/10/2010	60001
61002	21/10/2010	60004
61003	25/10/2010	60007
61004	25/10/2010	60008

Tabela: PagamentoParcelado			
pkPagamentoParcelado <<pk>>	nrParcelas	dataInicial	pkPagamento
62001	12	20/10/2010	60002
62002	12	21/10/2010	60003
62003	12	22/10/2010	60005
62004	18	24/10/2010	60006
62005	6	30/10/2010	60009

Figura 11.11:(a) Situação conceitual em que existe herança. (b) Equivalente de projeto. (c) Tabelas relacionais equivalentes.

A implementação da associação unidirecional das subclasses para a superclasse *nesse caso* pode ser implementada como uma chave estrangeira na tabela que representa a subclasse porque:

a) não existe nenhuma *associação* efetiva entre as instâncias que tivesse que ser representada à parte;

b) as propriedades da subclasse e da superclasse se complementam. São propriedades de um único objeto representadas em dois lugares diferentes.

A interpretação é que a tabela Pagamento apresenta a *continuação* da definição das tabelas PagamentoAVista e PagamentoParcelado.

11.2. Proxy Virtual

A equivalência em termos de representação entre classes e tabelas é apenas parte do problema de compatibilizar um projeto orientado a objetos com um banco de dados. É preciso ainda decidir como e quando os objetos serão salvos e carregados do banco. O projetista poderá optar por um projeto em que ele insira nos pontos adequados do código de carregamento e salvamento dos objetos à medida que a lógica da aplicação determine essa necessidade. Porém, essa abordagem, por assim dizer, *manual* de salvamento e carregamento, introduz uma carga extra no tempo de projeto, além de possibilitar a introdução de mais erros de lógica além daqueles que já são inerentes ao projeto da camada de domínio.

Além disso, se o projetista é que decide o momento de carregar e salvar objetos, muitas vezes essas operações poderão acabar sendo executadas sem necessidade, por exemplo, carregando objetos que já estão na memória e salvando objetos que não foram alterados. Controlar mais essas características caso a caso, método a método, não é a maneira mais produtiva de se proceder.

O ideal é que as tarefas de salvamento e carregamento de objetos sejam totalmente controladas pela própria arquitetura do sistema, ou seja, o projetista apenas teria de definir que um objeto é persistente, e todo um conjunto de métodos e estruturas de dados seria automaticamente criado em torno dele para permitir que o carregamento e o salvamento ocorram nos momentos mais apropriados.

Para implementar esse esquema, pode ser usado um padrão de projeto denominado proxy *virtual* (Larman, 2001). Um *proxy* virtual é um objeto muito simples que implementa apenas duas responsabilidades:

a) conhecer o valor da chave primária do objeto real. Isso não é problema, pois o *proxy* virtual é uma classe da camada de persistência e por isso pode ter acesso a esse valor;
b) repassar ao objeto real todas as mensagens que receber em nome dele.

O algoritmo da Figura 11.12 representa, de forma geral, o funcionamento de um *proxy* virtual.

Classe *Proxy* Virtual

Para qualquer mensagem recebida faça:
    Solicite ao BrokerManager o objeto real a partir de sua *pk*.
    Repasse a mensagem recebida ao objeto real.
Fim

Figura 11.12: Funcionamento geral de um *proxy* virtual.

Mais adiante, será explicado o que é e como funciona o BrokerManager.

Assim, o projeto poderá determinar que, em vez de os objetos se associarem uns aos outros, eles se associam com seus *proxies*. Dessa forma, será possível trazer para a memória uma instância de Editora sem trazer com ela as instâncias de Livro associadas. Basta associar a instância de Editora aos *proxies* dos livros. Essa atitude econômica, também chamada de *carregamento preguiçoso* (*lazy load*), permite grandes ganhos de eficiência de tempo e memória no sistema implementado.

O carregamento preguiçoso fará com que as instâncias de Livro só sejam carregadas para a memória se forem realmente necessárias. Por exemplo, se a instância de Editora vai apenas alterar seu endereço, não será necessário carregar as instâncias de Livro associadas. Porém, se a instância de Editora quiser saber qual o livro mais caro associado a ela, será necessário carregar as instâncias de livro associadas também.

Para que o projetista não tenha de decidir caso a caso quando os objetos devem ser carregados, o mecanismo de *proxy* virtual deve se interpor a todas as associações persistentes entre objetos. Os objetos reais simplesmente mandam mensagens uns aos outros como se todos estivessem em memória principal. Os *proxies* cuidam do carregamento quando ele se fizer necessário.

A Figura 11.13 exemplifica o funcionamento do mecanismo de *lazy load*. Inicialmente (a) apenas uma instância de Editora está em memória. Ela possui associação para três livros. Porém, em vez de os livros estarem em memória, apenas seus *proxies* estão. Quando a editora precisa solicitar alguma operação ou consulta de um dos livros (b), ela envia a mensagem pela asso-

ciação, que é interceptada pelo *proxy*, que providencia para que o livro seja trazido à memória. O livro teria associação com dois capítulos, mas apenas os proxies dos capítulos são carregados.

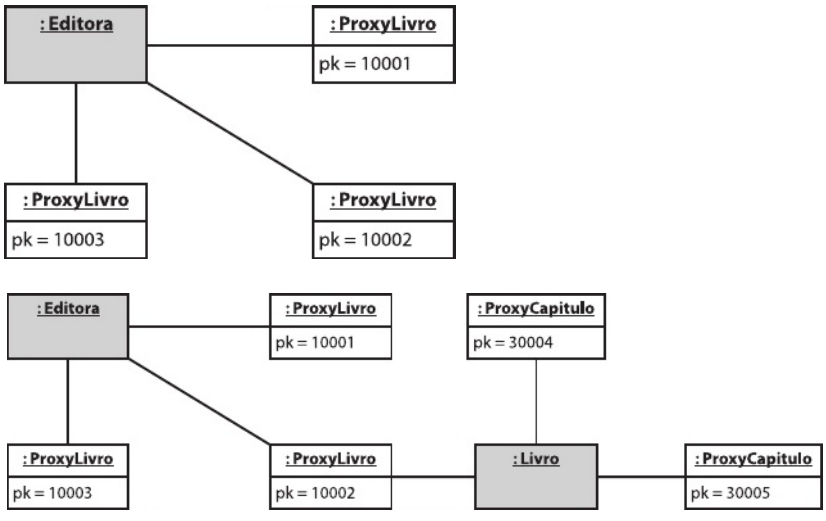


Figura 11.13: Ilustração do método *lazy load*: (a) apenas uma editora e seus proxies em memória e (b) um livro que se tornou necessário é carregado juntamente com seus proxies.

Se, agora, um dos capítulos fosse acessado por uma mensagem enviada do livro, então apenas o capítulo seria trazido à memória, já que não existiriam mais associações a partir dele.

11.2.1. Estruturas de Dados Virtuais

A implementação de *proxies* virtuais para cada objeto pode ser bastante ineficiente quando se trata de mapear coleções de objetos. Por exemplo, uma editora associada a 1.000 títulos de livro teria de ter 1.000 *proxies* instanciados associados a ela quando fosse trazida à memória? A resposta, felizmente, é *não*. A solução para evitar a instanciação indiscriminada de *proxies* em memória é a implementação de estruturas de dados virtuais pra substituir a implementação das associações.

Assim, uma editora não conteria um Set com 1.000 instâncias de *proxies* de livros, mas uma estrutura VirtualSet, que implementa os mesmos métodos de adição, remoção e consulta de um Set normal. Só que o VirtualSet não traz

a) cria uma instância da classe persistente;
b) inicializa os valores dos atributos da nova instância com valores da respectiva linha e coluna do banco de dados;
c) inicializa as estruturas de dados virtuais que implementam as associações do objeto com as chaves primárias dos respectivos objetos associados.

Para obter os valores das chaves primárias dos objetos associados, o *broker* deve saber quais são as associações que saem do objeto em questão e, em seguida, deve buscar nas tabelas associativas correspondentes ocorrências da pk do objeto em questão. A pk associada na tabela será adicionada na estrutura de dados virtual que vai implementar a associação.

Para exemplificar, um BrokerLivro deve materializar instâncias da classe Livro, definida conforme a Figura 11.3. De acordo com essas definições, esse BrokerLivro deverá implementar um método materializa, executando as seguintes operações:

a) criar uma instância de Livro;
b) preencher os atributos isbn, titulo, editora, autor e nrPaginas da nova instância com os valores armazenados nas respectivas colunas da tabela Livro no banco de dados;
c) buscar na tabela livro_capitulos ocorrências da *pk* do livro na coluna pkLivros. Para todas as ocorrências, adicionar no VirtualSet capitulos da nova instância de Livro os valores da coluna correspondente pkCapitulo.

Não se deve confundir a materialização feita pelo Broker com a criação de instância definida nos contratos de operação de sistema. Nos contratos, a criação de uma instância refere-se à inserção de nova informação, independentemente do meio físico no qual ela esteja armazenada (memória principal ou secundária). A materialização feita pelo Broker apenas representa o ato de trazer para a memória principal um objeto que está em memória secundária. A materialização é, pois, uma operação exclusiva da camada de persistência, nada tendo a ver com as regras de negócio.

11.4. Caches

Os objetos em memória principal podem ser classificados em:

a) *limpos* ou *sujos*, dependendo de estarem ou não consistentes com o banco de dados;
b) *novos* ou *velhos*, dependendo de já existirem ou não no banco de dados;

c) *excluídos*, dependendo de terem sido excluídos da memória, mas ainda não do banco de dados.

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

Embora existam oito combinações possíveis, e Larman (2001) trabalhe com seis delas, a prática indica que apenas quatro caches são suficientes para gerenciar objetos em memória primária:

a) *old clean cache*: onde ficam os objetos que estão consistentes com o banco de dados, ou seja, velhos e limpos;
b) *old dirty cache*: onde ficam os objetos que existem no banco de dados, mas foram modificados em memória, isto é, velhos e sujos;
c) *new cache*: onde ficam os objetos que foram criados em memória, mas ainda não existem no banco de dados;
d) *delete cache*: onde ficam os objetos deletados em memória, mas que ainda existem no banco de dados.

Quando se diz que o BrokerManager verifica se um objeto está em memória, ele faz uma consulta às caches existentes e, caso encontre uma referência ao objeto cuja pk foi passada, retorna o objeto ao Proxy. Se o BrokerManager procurar todas as caches e não encontrar o objeto, ele deverá solicitar ao *broker* especializado na classe do objeto que o materialize. O objeto assim materializado é inserido na OldCleanCache.

Se, em algum momento, esse objeto for alterado em memória, ou seja, se algum de seus atributos for mudado ou se alguma associação partindo dele for criada ou destruída, ele será movido da OldCleanCache para a OldDirtyCache.

Para se ter um bom controle do estado de cada objeto, é importante que as variáveis de instância que representam atributos e associações do objeto só possam ser alteradas pelos métodos set, add e remove. Dessa forma, em cada um desses métodos, poderá ser adicionado um comando do tipo BrokerManager.instance().ficouSujo(self). Essa mensagem enviada ao BrokerManager vai fazer com que ele mova o objeto de uma OldCleanCache para uma OldDirtyCache ou que o mantenha na OldDirtyCache, se ele já estiver lá.

Objetos criados em memória, como resultado de uma pós-condição de contrato, devem ser armazenados em uma NewCache. Um objeto em uma NewCache não pode ser movido para a OldDirtyCache, mesmo se tiver seus atributos alterados. Ele só será movido para a OldCleanCache depois do commit

sobre o banco de dados. Antes disso ele permanece na NewCache, mesmo que o seus atributos sejam alterados.

Quando for solicitada a destruição de um objeto em memória, o resultado dependerá de onde ele está. Se ele estiver na OldCleanCache ou na OldDirtyCache será movido para a DeleteCache. Porém, se ele estiver na NewCache, pode ser simplesmente deletado sem maior cerimônia.

11.5. Commit e Rollback

As operações de *commit* e *rollback* são normalmente ativadas pela camada de aplicação para indicar, respectivamente, que uma transação foi bem-sucedida e confirmada ou que foi cancelada. Essas operações são implementadas pelo BrokerManager. No caso do commit, o BrokerManager deve executar o seguinte:

a) efetuar um update no banco de dados para os objetos da OldDirtyCache e mover esses objetos para a OldCleanCache;
b) efetuar um insert no banco de dados para os objetos da NewCache e mover esses objetos para a OldCleanCache;
c) efetuar um remove no banco de dados para os objetos da DeleteCache e remover esses objetos da *cache*.

No caso de um rollback, o BrokerManager deve apenas remover todos os objetos de todas as *caches*, exceto os da OldCleanCache.

Como a OldCleanCache pode crescer indefinidamente, é necessário implementar algum mecanismo para remover dela os objetos mais antigos sempre que seu tamanho atingir algum limite preestabelecido.

As outras *caches* crescem apenas até o momento do commit ou rollback, quando são esvaziadas.

11.6. Controle das Caches em um Servidor Multiusuário

Se mais de um usuário conecta-se ao sistema, é necessário determinar como vai funcionar o compartilhamento de dados em memória. Considerando uma arquitetura cliente/servidor com camadas de interface, domínio e persistência, pelo menos duas abordagens são possíveis:

a) na primeira abordagem, as três camadas são executadas no cliente. Não existe compartilhamento de memória no servidor, o qual serve apenas

para armazenar os dados no momento em que a transação efetuar um commit. Nesse caso, o que trafega pela rede são registros do banco de dados e instruções SQL apenas nos momentos da materialização de objetos ou commit. A desvantagem dessa forma de definir a arquitetura é que o nó cliente fica sobrecarregado, e mecanismos de controle de segurança adicionais devem ser implementados no próprio banco de dados para impedir acessos não autorizados;

b) outra possibilidade é implementar no nó cliente apenas a camada de interface e deixar no servidor as camadas de domínio e persistência. Nesse caso, os objetos existirão em memória apenas no servidor, e a comunicação na rede consistirá no envio de mensagens e recebimento de dados.

Estando os objetos fisicamente no servidor, existem duas possibilidades ainda. No primeiro caso, todos os usuários compartilham as quatro caches, com a desvantagem de que um usuário poderá ter acesso a objetos modificados que ainda não foram confirmados por commit pela aplicação de outro usuário. Essa opção parece ser desaconselhável na maioria das aplicações.

A outra opção é permitir a cada usuário apenas a visualização dos objetos cuja informação é confirmada, ou seja, objetos que estejam na OldCleanCache. Objetos em outras caches são, portanto, objetos que estão em processo de modificação por algum usuário e não devem ser acessíveis.

Assim, o mecanismo de persistência em sistemas multiusuário pode ser implementado da seguinte forma:

a) uma OldCleanCache compartilhada por todos os usuários;
b) cada usuário possuirá individualmente sua própria OldDirtyCache, DeleteCache e NewCache.

Procedendo assim, é possível garantir que nenhum usuário tenha acesso a objetos sendo modificados por outro usuário. É possível, portanto, usar as caches para implementar um mecanismo de lock, ou seja, quando um usuário usa um objeto, outros não podem ter acesso a ele. Quando o usuário que está de posse do objeto efetua um commit ou rollback, o lock é desfeito e outros usuários podem acessar novamente o objeto.

Uma grande vantagem desse método está no uso otimizado da memória do servidor. Os objetos na OldCleanCache, que é a única que cresce de forma indeterminada, são compartilhados por todos os usuários. As outras quatro caches, que são específicas para cada usuário, só crescem durante uma transação. Quando ocorrer commit ou rollback essas caches são esvaziadas.

Capítulo

12

Geração de Código e Testes

A fase de construção do UP prevê a geração e código e testes do sistema. É necessário gerar código tanto para a camada de domínio, resultante do projeto lógico, quanto para as demais camadas, resultantes do projeto tecnológico.

Uma vez definidos os diagramas de comunicação e o DCP, a *geração de código* é uma tarefa passível de automatização. Trata-se neste capítulo da geração de código das classes correspondentes à camada de domínio da aplicação, ou seja, as classes que realizam toda a lógica do sistema a partir das operações e consultas de sistema.

Este capítulo apresenta regras para geração de código a partir do DCP e dos diagramas de comunicação. Os exemplos são apresentados em pseudocódigo, que pode ser traduzido para qualquer linguagem de programação (preferencialmente orientada a objetos).

12.1. Classes e Atributos

Classes do DCP são imediatamente convertidas em classes na linguagem de programação. Os atributos das classes são convertidos em variáveis de instância (privadas) da respectiva classe. Atributos sempre terão tipos alfanu-

Além disso, considerando as diferentes multiplicidades de papel e outras características das associações, haverá algumas distinções a fazer quanto aos métodos associados.

Na geração de código da camada de domínio, não se diferencia associações temporárias e persistentes, pois sua implementação é a mesma.

De forma geral, cada associação deverá implementar pelo menos três métodos:

a) add, tendo como parâmetro o objeto a ser associado;
b) remove, tendo como parâmetro o objeto a ser desassociado;
c) get, retornando uma cópia da coleção de objetos associados, sobre a qual é possível realizar iterações.

As associações derivadas implementam apenas o método get, de acordo com sua definição.

Em relação a essas operações básicas, ainda pode ser possível implementar variações:

a) se a associação for *qualificada*, pode-se ter um get que recebe como parâmetro a chave do qualificador e retorna apenas o objeto qualificado, e não o conjunto todo. Da mesma forma, o add poderá passar como parâmetro adicionalmente o valor do qualificador, especialmente se for um qualificador externo. O método remove poderá remover a associação a partir do valor do qualificador;
b) no caso de associações *ordenadas*, pode-se ter um método get que retorna um elemento conforme sua posição. Da mesma forma, o add poderá adicionar elementos diretamente em uma posição indicada como parâmetro e o remove remover da posição indicada;
c) associações ordenadas também podem ter métodos especiais para acessar, adicionar e remover elementos no início ou no fim da lista;
d) *pilhas* e *filas* terão métodos específicos como *push* e *pop*, que seguem as regras específicas dessas estruturas.

12.2.1. Associação Unidirecional para Um

A associação unidirecional *para um* ou *para* 0..1 pode ser armazenada em uma variável de instância na classe de srcem da associação e seu tipo deve ser a classe de destino. Assim, uma associação unidirecional *para um* de

Pagamento para Venda corresponderá a uma variável de instância na classe Pagamento declarada com tipo Venda.

Em relação aos métodos, observa-se que o método que remove a associação não precisa de parâmetros, pois existe um único objeto a ser removido.

A Figura 12.2 mostra um exemplo de classe com associação unidirecional para um e o respectivo pseudocódigo a ser implementado. Atributos foram suprimidos, pois já foram explicados na seção anterior.

```
Classe Pagamento
VAR PRIVADA
 venda : Venda

MÉTODO addVenda(umaVenda)
     SE self.getVenda() = NULL ENTÃO
      venda := umaVenda
      SENÃO
       self.throw("Já existe uma venda associada")
   FIM SE
FIM MÉTODO

MÉTODO removeVenda ()
 venda := NULL
FIM MÉTODO

MÉTODO getVenda():Venda
 RETORNA venda
FIM MÉTODO
FIM CLASSE
```

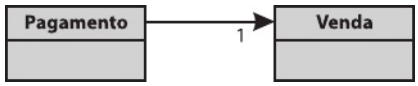


Figura 12.2: Classe com associação unidirecional para um e respectivo pseudocódigo.

Quando a multiplicidade for estritamente para um, a associação pode ser removida, como mencionado no Capítulo 8, mas o objeto ficará temporariamente inconsistente, devendo a associação removida ser substituída por outra ainda no contexto da mesma operação de sistema.

12.2.2. Associação Unidirecional para Muitos

A associação unidirecional para * (ou qualquer outra multiplicidade diferente de um ou 0..1) corresponde à implementação de uma estrutura de dados. Sendo uma associação simples, será implementada como um conjunto (Set).

A Figura 12.3 apresenta um exemplo desse tipo de construção.

```
CLASSE Venda
VAR PRIVADA
 itens : SET[Item]

MÉTODO addItens(umItem:Item)
 itens.add(umItem)
FIM MÉTODO

MÉTODO removeItens(umItem:Item)
 itens.remove(umItem)
FIM MÉTODO

MÉTODO getItens():Set[Item]
 RETORNA itens.copia()
FIM MÉTODO
```

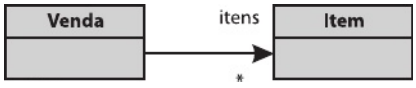


Figura 12.3: Classe com associação unidirecional simples para muitos e respectivo código.

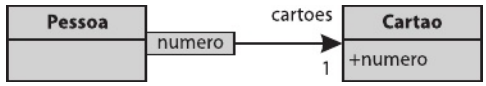
Se a associação for rotulada com {sequence}, {ordered set} ou {bag}, deve-se substituir o tipo de dados da variável de instância Set pelo tipo apropriado de acordo com a linguagem. Podem ser usados também, conforme o caso, tipos concretos de dados como *array* ou árvore binária, por exemplo. No caso de associações para muitos com limite inferior e superior idênticos, inclusive, recomenda-se a implementação como *array*. Por exemplo, uma associação com multiplicidade de papel 5 (ou 5..5) deve ser implementada como um *array* de cinco posições.

Adicionalmente podem ser implementados métodos específicos dessas estruturas, conforme mencionado no início da Seção 12.2.

12.2.3. Associação Unidirecional Qualificada

A associação unidirecional qualificada é implementada de forma semelhante à associação com multiplicidade para muitos. Porém, em vez do tipo de dados Set, usa-se uma estrutura de dicionário ou mapeamento (Map), que associa o atributo qualificador a um objeto ou objetos dependendo da multiplicidade do papel. Nesse caso, será possível implementar um método de consulta que retorne um objeto da coleção a partir de um valor para o qualificador.

A Figura 12.4 apresenta a implementação de uma associação qualificada definindo um mapeamento (para um) com qualificador interno. A Figura 12.5 apresenta o caso de qualificador externo. A Figura 12.6 apresenta o caso de implementação de uma associação qualificada como partição (para *).

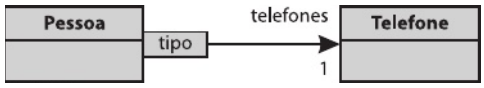



```
Classe Pessoa
VAR PRIVADA
 cartoes : MAP[String->Cartao]

MÉTODO addCartoes(umCartao)
 cartoes.put(umCartao.getNumero(), umCartao)
FIM MÉTODO

MÉTODO removeCartoes (umNumero:String)
 cartoes.removeKey (umNumero)
FIM MÉTODO

MÉTODO getCartoes(umNumero:String):Cartao
 RETORNA cartoes.at(umNumero)
FIM MÉTODO
```

Figura 12.4: Associação qualificada como mapeamento (com qualificador interno) e seu código correspondente.

O atributo numero do Cartao é tipado como String, conforme discutido no Capítulo 7, porque se comporta como tal. Dificilmente serão executadas operações matemáticas com números de cartão.

Na Figura 12.4, pode-se adicionar, ainda, a implementação dos métodos get, add e remove da associação para muitos conforme a Seção 12.2.2.




```
Classe Pessoa
VAR PRIVADA
 telefones : MAP[String->Telefone]

MÉTODO addTelefones(umTipo:String, umTelefone: Telefone)
 telefones.put(umTipo, umTelefone)
FIM MÉTODO

MÉTODO removeTelefones (umTipo:String)
 telefones.removeKey (umTipo)
FIM MÉTODO

MÉTODO getTelefones(umTipo:String):Telefone
 RETORNA cartoes.at(umTipo)
FIM MÉTODO
```

Figura 12.5: Associação qualificada como mapeamento (com qualificador externo) e seu código correspondente.


```
Classe Editora
VAR PRIVADA
 livros MAP[String,SET[Livro]]

MÉTODO addLivros(umGenero:String, umLivro:Livro)
     SE SET livros.at(umGenero) = NULL ENTÃO
        livros.put(umGenero,SET.new())
   FIM SE
     livros.at(umGenero).add(umLivro)
FIM MÉTODO

MÉTODO removeLivros (umGenero:String,umLivro:String)
 livros.at(umGenero).remove(umLivro)
FIM MÉTODO
```

```
MÉTODO removeGenero (umGenero:String)
 livros.removeKey(umGenero)
FIM MÉTODO

MÉTODO getLivros(umGenero:String):SET[Livro]
 RETORNA livros.at(umGenero)
FIM MÉTODO
```

Figura 12.6: Associação qualificada como partição e seu código correspondente.

Aqui foram definidas duas operações de remoção, uma baseada no livro, que remove uma única associação, e outra baseada no gênero, que remove todos os livros do gênero.

O método getLivros, baseado no gênero, retorna um conjunto de livros.

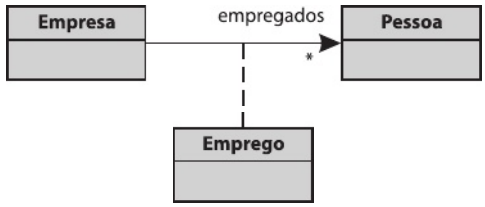
12.2.4. Associação Unidirecional com Classe de Associação

Quando a associação contém uma classe de associação, é necessário implementar a criação e destruição de instâncias dessa classe de associação cada vez que uma associação for adicionada e removida.

Classes de associação podem existir em associações com qualquer multiplicidade. Entretanto, o mais comum é que classes de associação sejam usadas em associações de * para *.

A implementação desse tipo de associação consiste em um Map, associando instâncias do destino da associação com instâncias da classe de associação.

O exemplo da Figura 12.7 mostra uma classe de associação e a implementação da associação unidirecional na classe de srcem.




```
Classe Empresa
VAR PRIVADA
 empregados : MAP[Pessoa->Emprego]
```

```
MÉTODO addEmpregados(umaPessoa:Pessoa)
 empregados.put(umaPessoa, Emprego.newInstance())
FIM MÉTODO

MÉTODO removeEmpregados(umaPessoa)
 empregados.removeKey(umaPessoa)
FIM MÉTODO

MÉTODO getEmpregados():SET[Pessoa]
 RETORNA empregados.copia()
FIM MÉTODO

MÉTODO getEmpregados(umaPessoa):Emprego
 RETORNA empregados.at(umaPessoa)
FIM MÉTODO
```

Figura 12.7: Classe de associação e respectivo código.

Na Figura 12.7, nota-se que, ao adicionar uma nova associação de Empresa com Pessoa, é automaticamente criado um novo Emprego.

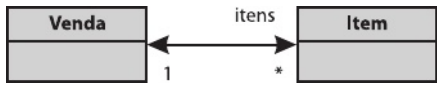
Há dois métodos de acesso implementados: um que retorna todos os empregados (instâncias de Pessoa associadas à empresa) e outro, parametrizado, que retorna um emprego a partir de uma pessoa.

12.3. Associação Bidirecional

Existem pelo menos três padrões para implementação de associações bidirecionais (Fowler, 2003):

a) implementar a associação como duas associações unidirecionais nas duas classes participantes;
b) implementar a associação como unidirecional apenas em uma das classes. A outra classe pode acessar os elementos da associação fazendo uma pesquisa;
c) implementar um objeto intermediário que representa a associação e pode ser identificado através de métodos de localização rápida como *hash*.

direção. A vantagem é o código mais simples e a economia de espaço. A desvantagem é que a navegação na direção oposta será uma operação bem mais lenta do que na direção implementada. A Figura 12.9 apresenta um exemplo no qual a associação bidirecional é implementada apenas na classe Venda.




```
CLASSE Venda
VAR PRIVADA
 itens : SET[Item]

MÉTODO addItens(umItem:Item)
 itens.add(umItem)
FIM MÉTODO

MÉTODO removeItens(umItem:Item)
 itens.remove(umItem)
FIM MÉTODO

MÉTODO getItens():Set[Item]
 RETORNA itens.copia()
FIM MÉTODO

CLASSE Item
-- não se declara aqui a variável como nos casos anteriores

MÉTODO getVenda():Venda
 PARA TODA venda EM Venda.allInstances() FAÇA
     SE venda.itens().includes(self) ENTÃO
       RETORNA venda
    FIM SE
 FIM PARA
FIM MÉTODO
FIM CLASSE
```

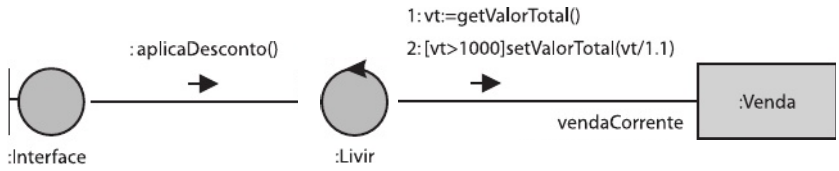
Figura 12.9: Uma associação bidirecional e sua implementação unidirecional.

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

Figura 12.11: Implementação de uma operação de sistema e um método delegado.

Na figura foram omitidas as declarações de atributos, associações e métodos básicos relacionados. Apenas a operação de sistema e o método delegado foram mostrados.

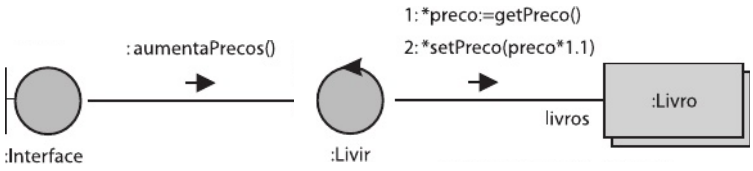
A Figura 12.12 mostra a implementação de uma operação de sistema com mensagem condicionada.




```
CLASSE Livir
...
MÉTODO aplicaDesconto()
 VAR vt:MOEDA
 vt := self.getValorTotal()--1

 SE vt>1000 ENTÃO
     self.getVendaCorrente().setValorTotal(vt/1.1) --2
 FIM SE
FIM MÉTODO
FIM CLASSE
```

Figura 12.12: Implementação de uma operação de sistema com mensagem condicionada.

Finalmente, a Figura 12.13 mostra como seria a implementação de uma operação de sistema com mensagens iterativas.




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

Figura 12.13: Implementação de uma operação de sistema com mensagem condicionada.

O procedimento de geração de código, então, pode ser implementado assim:

a) geração de código para as classes, atributos e associações, conforme definições deste capítulo;
b) geração de código para as operações básicas, também segundo os padrões descritos neste capítulo;
c) geração de código para as operações e consultas de sistema e métodos delegados de acordo com os diagramas de comunicação definidos na atividade de projeto ou seus equivalentes diagramas de sequência ou algoritmos.

12.5. Testes

Por melhores que sejam as técnicas de projeto e por mais ferramentas de geração de código que se tenha, o teste do software continuará a ser sempre uma necessidade devido ao fator "erro humano". Especificações malfeitas, es-

quecimentos, inconsistências podem fazer o software falhar. Cabe ao analista de testes prover as ferramentas para solucionar tais problemas.

Foge ao escopo deste livro apresentar um método completo de teste de software. Existem trabalhos específicos sobre esse assunto, como o livro de Maldonado, Delamaro e Jino (2007). A intenção desta seção é mostrar como o teste de software pode ser concebido quando um conjunto de técnicas como as apresentadas aqui forem usadas.

Embora o teste fosse relegado a segundo plano até os anos 1990, quando o plano de software normalmente se resumia a chamar um programador e dizer "testa aí!", o assunto hoje se reveste de grande importância devido a fatores como qualidade do produto de software, cláusulas contratuais que punem erros e funcionalidades incorretas e a tendência da grande indústria de software a realizar testes independentes.

Sem entrar no mérito de definir famílias técnicas, como testes caixa-branca ou caixa-preta, pode-se classificar os testes de software em relação aos seus *objetivos* nos seguintes grupos:

a) teste de unidade • verificar o funcionamento dos métodos implementados;
b) teste de integração • verificar se a comunicação entre objetos funciona;
c) teste de sistema • verificar execução do sistema do ponto de vista do usuário final;
d) teste de aceitação • teste conduzido pelos usuários finais;
e) teste de operação • teste conduzido pelos administradores do sistema no ambiente final;
f) teste de regressão • aplicado quando novas versões do software são liberadas.

12.5.1. Teste de Unidade

O teste de unidade tem como objetivo verificar o funcionamento dos métodos mais básicos. Já foi mostrado que a camada de domínio do software (que realiza toda a lógica de acesso e transformação de dados) é projetada por diagramas de comunicação nos quais as mensagens se estruturam como em uma árvore, estando a operação de sistema na raiz, os métodos delegados nos ramos e as operações básicas nas folhas.

D

N

O

P

Q

R

S