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 associações que 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 na figura não estabelece que alunos só podem se matricular nas disciplinas de seu próprio curso.

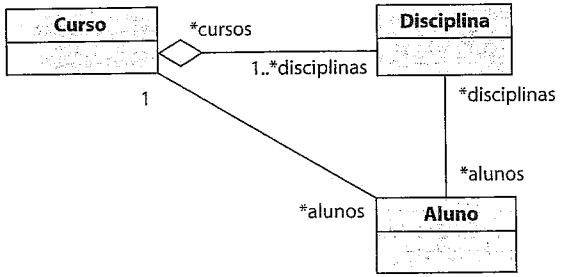

Figura 7.65: Uma situação que necessita de uma invariante para que a consistência entre associações se mantenha.

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.cursos→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.

a) getEmprego(), que retorna o emprego atual ou o conjunto vazio, se ele não existir;
b) getEmprego(index), que retorna um emprego anterior, cujo número de ordem é dado pelo valor inteiro passado como parâmetro, ou o conjunto vazio se tal emprego não existir;
c) getEmprego(time), em que time é um valor que corresponde a uma data/hora válida. Se a pessoa em questão tinha um emprego naquela data/hora, ele será retornado, caso contrário é retornado o conjunto vazio.

A Figura 7.62 apresenta uma possível implementação do estereótipo <<timestamp>>.

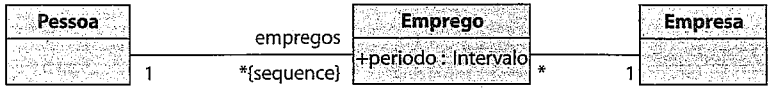

Figura 7.62: Uma possível implementação para o estereótipo <<timestamp>>.

Para que a implementação da Figura 7.62 seja exatamente equivalente à Figura 7.61, é necessário ainda que se estabeleça uma invariante que garanta que a mesma pessoa não tenha dois empregos no mesmo período de tempo. Se o papel **emprego** da Figura 7.61, porém, fosse *, tal invariante não seria necessária.

Observa-se que a classe Emprego, em vez de ter atributos para data inicial e data final, possui um único atributo periodo:Intervalo representando um período contínuo de tempo. Isso leva ao padrão seguinte: *Intervalo*.

7.6.10. Intervalo

Sempre que um objeto qualquer tem pares de atributos representando um início e um fim, como data inicial e final, em vez de representar essa situação como dois atributos, é preferível definir e utilizar um tipo primitivo chamado Intervalo , como na Figura 7.63.

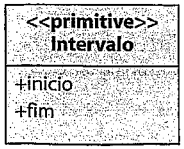

Figura 7.63: Tipo primitivo Intervalo.

Há dois motivos para isso: o primeiro é que a existência dos dois atributos fortemente correlacionados (início e fim) em uma classe fere o princípio de coesão, já explicado. O segundo é que, possivelmente, várias operações específicas sobre intervalos serão necessárias, como, por exemplo, verificar se uma determinada data está dentro de um intervalo ou verificar se dois intervalos se sobrepõem.

É muito mais razoável implementar essas operações uma única vez em uma classe primitiva do que implementá-las inúmeras vezes nas classes conceituais cada vez que houver necessidade delas.

7.7. Invariantes

Existem situações em que a expressividade gráfica do diagrama de classes é insuficiente para representar determinadas regras do modelo conceitual. Nesses casos, necessita-se fazer uso de invariantes.

Invariantes são restrições sobre as instâncias e classes do modelo. Certas restrições podem ser representadas no diagrama: por exemplo, a restrição de que uma venda não pode ter mais do que cinco livros poderia ser representada como na Figura 7.64.

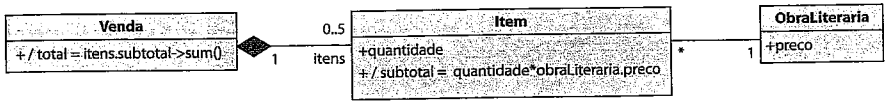

Figura 7.64: Uma restrição que pode ser representada no diagrama.

Mas nem todas as restrições podem ser representadas tão facilmente. Se houvesse uma restrição que estabelecesse que nenhuma venda pode ter valor superior a mil reais, isso não seria passível de representação nas associações nem nos atributos do diagrama da Figura 7.64. Mas seria possível estabelecer tal restrição usando invariantes de classe como a seguir:

```
Context Venda
 inv:
   self.total <= 1000,00
```

Talvez a maioria dos desenvolvedores de software, quando se depara com regras desse tipo acaba incorporando-as nos métodos que fazem algum tipo de atualização nas instâncias da classe. Por exemplo, no caso anterior, poderia ser colocado um teste no método que faz a adição de um novo item em uma venda para verificar se o valor total passaria de 1.000 e, se for o caso, impedir a adição.

Assim, cada uma das possíveis transações de pedido, compra, venda, devolução e quebra de estoque pode ser modelada como instâncias de Transacao. Por exemplo:

a) um *pedido* é uma transação que tira de uma conta de fornecedor e repassa à conta de saldo de pedidos;
b) a *chegada da mercadoria* é uma transação que tira da conta de saldo de pedidos e coloca na conta de estoque;
c) uma *venda* é uma transação que retira da conta de estoque e coloca na conta de remessa;
d) um *registro de envio* é uma transação que retira da conta de remessa e coloca na conta de itens enviados;
e) uma *devolução* é uma transação que retira da conta de itens enviados e coloca novamente na conta de estoque;
f) uma *confirmação de entrega* é uma transação que retira da conta de itens enviados e coloca em uma conta de entrega definitiva.

Esse padrão comporta inúmeras variações e sofisticações, mas é muito interessante ver como uma simples ideia poderosa pode dar conta de tantas situações cuja modelagem ingênua poderia ser bastante complicada.

7.6.9. Associação Histórica

Frequentemente, o analista se defronta com a necessidade de que uma associação tenha memória. Por exemplo, pode-se representar a relação entre pessoas e seus empregos. Mas, quando uma pessoa muda de emprego, se a associação representa apenas a situação presente, a memória dos empregos anteriores se perde.

Caso se queira guardar informações históricas de uma associação, como empregos anteriores, pode-se usar o padrão *Associação Histórica*. Há um estereótipo associado a esse padrão, conforme mostrado na Figura 7.59.

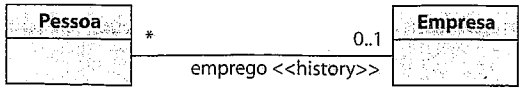

Figura 7.59: Uma associação histórica.

A associação da Figura 7.59 significa que uma pessoa pode ter no máximo um emprego em um dado instante de tempo, mas, se já teve outros empregos antes, eles podem ser recuperados como em uma lista. Ou seja, é possível retornar o emprego anterior, o segundo anterior e assim por diante.

Na prática, tal padrão é implementado a partir de duas associações, a atual e a histórica, como na Figura 7.60. O estereótipo é, então, uma forma de abreviar essa estrutura mais complexa substituindo-a por uma forma mais simples.

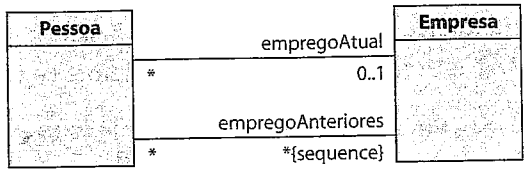

Figura 7.60: Desdobramento físico do estereótipo <<history>>.

Como será visto nos capítulos seguintes, a associação normal tem apenas um método para retornar seus elementos. No caso, a classe Pessoa terá um método getEmprego() que retorna o emprego da pessoa se ele existir ou o conjunto vazio, caso contrário. Se a associação for estereotipada com <<history>>, além desse método padrão, vai existir outro indexado getEmprego(index), onde index=1 representa o emprego atual, index=2 representa o emprego anterior, e assim por diante. Se não houver um emprego para o index dado, a função retorna o conjunto vazio.

Esse tipo de associação, porém, não é capaz de indicar qual o emprego da pessoa em uma determinada data. Ela pode apenas dizer onde a pessoa trabalhava antes, mas não quando saiu de lá. Pode-se optar, então, se necessário, por um padrão em que, além da memória da sequência, a associação tenha memória de tempo, como na Figura 7.61.

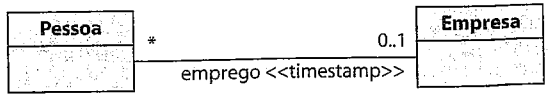

Figura 7.61: Uma associação histórica com registro de tempo.

A associação histórica com registro de tempo, além dos métodos get() e get(index), já mencionados, ainda permite o acesso ao elemento em dado período de tempo pelo método get(time). Seria possível consultar a associação da Figura 7.61 de três formas:

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 origem 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.

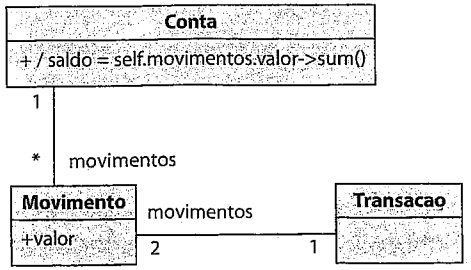

Figura 7.58: Classes do padrão *Conta/Transação*.

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.

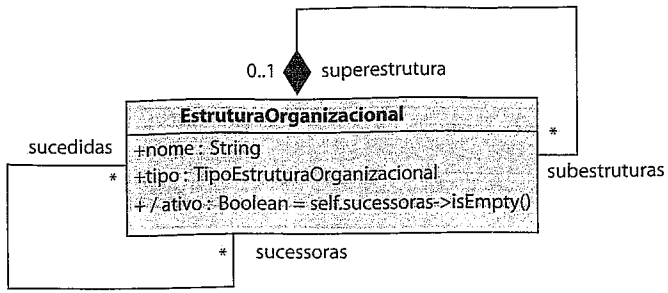

Figura 7.56: Um exemplo de aplicação da estratégia *Sucessor*.

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

Manter a estrutura original, 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 originais 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

considerados equivalentes se estão ligados ao mesmo objeto essência. Adicionalmente, a figura já introduz uma sofisticação que é a definição do conjunto de pessoas que aceitam a equivalência, que não é necessariamente unânime.

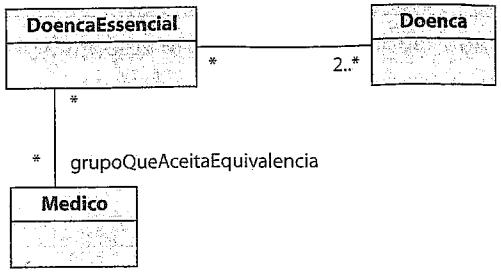

Figura 7.57: Exemplo da aplicação da técnica Essência/Aparência.

O exemplo aplica-se na área da saúde, na qual, em alguns casos, grupos de médicos aceitam que determinadas doenças são, na verdade, uma única doença, mas isso nem sempre é unanimidade.

O objeto essencial existe apenas para ligar objetos; ele não tem outras propriedades.

7.6.7.4. Desfazendo a Junção

Quando parece que o mundo real não pode ficar mais complexo, ele usualmente fica. Então, se existe a possibilidade de se descobrirem dados redundantes que necessitam de junção, também é possível que junções sejam feitas indevidamente e que tenham de ser desfeitas. Novamente, tais operações só são possíveis a partir de uma cirurgia de peito aberto no banco de dados ou a partir de operações bem planejadas disponíveis na interface do sistema. A segunda opção costuma ser menos invasiva.

Deve-se observar que, para que as junções feitas pela técnica *Copiar e Substituir* possam ser desfeitas, seria necessário guardar um *backup* dos objetos originais, pois a técnica destrói um dos objetos e descaracteriza o outro. No caso de *Sucessor*, a desvinculação ente os objetos é feita pela remoção da associação. No caso de *Essência/Aparência*, a junção é desfeita pela eliminação do objeto essencial. Porém, em todos os casos, deve-se decidir como tratar eventuais modificações que um objeto tenha sofrido quando estava ligado a outros.

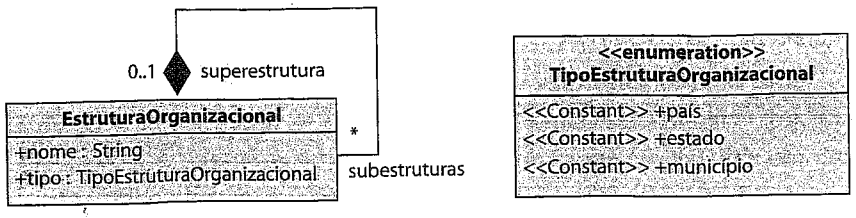

Figura 7.55: Aplicação do padrão Estrutura Organizacional.

Dessa forma, ganha-se flexibilidade em relação a poder lidar simultaneamente com estruturas organizacionais de diferentes países, bem como se pode mais facilmente lidar com suas mudanças, como a criação de novos níveis hierárquicos.

Esse padrão pode ter inúmeras variantes conforme se queira representar hierarquias concorrentes, estruturas sucessoras ou equivalentes e outras situações que podem surgir. Algumas serão comentadas nas seções seguintes.

7.6.7. Junção de Objetos

Um dos pressupostos de analistas que muitas vezes falha é que os usuários, ao operarem o sistema, farão tudo de acordo com o previsto. Isso nem sempre acontece. A falha humana ainda é frequente, apesar dos esforços no sentido de se construir interfaces cada vez mais à prova de falhas. Algum usuário poderia, por exemplo, cadastrar uma nova editora no sistema e, mais adiante, descobrir que ela, na verdade, já era cadastrada. Um código registrado erroneamente ou a impossibilidade de se ter um identificador único para um objeto com significado no mundo real podem causar essa situação.

A solução, quando esse erro de usuário ocorre, é fazer a junção dos objetos, usualmente copiando um sobre o outro. Essa operação pode ser executada diretamente como uma correção no banco de dados, mas em algumas situações pode ser necessário que o sistema esteja preparado para permitir ao próprio usuário fazer essa junção.

Além disso, nem sempre é um *erro* que provoca a necessidade de uma junção. Em algumas situações, a equivalência entre diferentes instâncias de um conceito pode não ser consenso, sendo necessário representar duas ou mais visões contraditórias simultaneamente. Em outros casos, como nas hierarquias organizacionais múltiplas, pode ser necessário indicar que duas estruturas 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 original 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 originais 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.

Outra situação é a política de descontos da empresa. É possível que, no momento da contratação, a livraria aplique uma política de dar desconto de 10% para compras acima de 100 reais. Mas, com o passar do tempo, novas e imprevisíveis políticas podem ser criadas pelos departamentos comerciais, por exemplo:

a) um livro grátis de até 50 reais para compras acima de 300 reais;
b) 20% de desconto em até dois livros no dia do aniversário do comprador;
c) 5% de desconto nos livros de suspense nas sextas-feiras 13.

Além disso, pode ser permitido combinar duas ou mais políticas, caso se apliquem, ou escolher apenas aquela que dá o maior desconto.

O padrão *Estratégia* sugere que, nesses casos, o procedimento deve ser separado dos dados aos quais ele se aplica. Ou seja, se é aplicado um desconto em uma venda, o desconto não deve ser meramente um método da venda a ser alterado quando houver mudanças. O desconto será representado por uma classe abstrata associada à venda. Essa classe abstrata terá subclasses concretas que representarão políticas concretas de desconto (Figura 7.53).

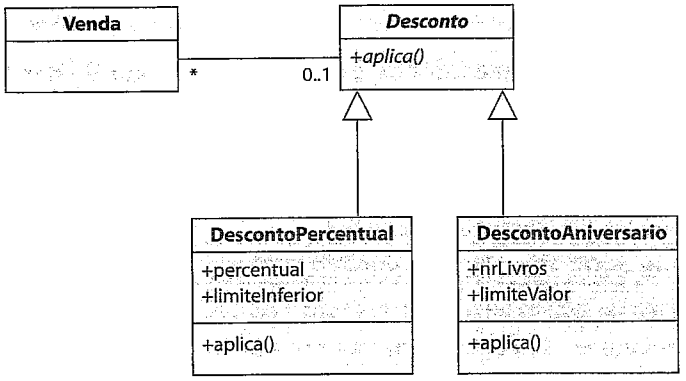

Figura 7.53: Padrão Estratégia.

Esse padrão não é puramente conceitual, pois sua realização envolve a existência de métodos (que pertencem ao domínio do projeto). Então, cada instância de Venda será associada a uma instância de uma das subclasses da estratégia de desconto. A classe abstrata *Desconto* implementa um método abstrato *aplica*(), que é aplicado de forma concreta nas subclasses. Caso alguma das estratégias concretas precise de dados específicos do comprador ou da

venda, eles podem ser acessados através das associações da classe de desconto concreta para as classes que contêm a informação necessária através de Venda.

Esse padrão minimiza dois problemas com a mudança desse tipo de requisito. Primeiro, ele mantém, para cada venda, o registro da estratégia de desconto aplicada na época em que a venda foi feita. Então, essa informação não se perde. Em segundo lugar, se novas estratégias de desconto forem criadas no futuro, basta implementar novas subclasses para Desconto. Isso não afeta o funcionamento das estratégias anteriores.

7.6.6. Hierarquia Organizacional

Outra situação comum consiste na necessidade de representar hierarquias organizacionais que nem sempre se comportam bem. É comum, por exemplo, representar a estrutura administrativa do Brasil com os níveis de estados e municípios, como na Figura 7.54.

Figura 7.54: Representação direta de uma hierarquia organizacional usando classes e composição.

Porém, hierarquias normalmente não se comportam de forma tão simples. Pode-se observar, de início, que essa organização não se repete em muitos países, que adotam outras formas de divisão administrativa. Além disso, reformas políticas e administrativas podem mudar a hierarquia, criando subdivisões ou agrupando diferentes níveis hierárquicos. Aliás, diferentes estruturas organizacionais podem coexistir, por exemplo, com estados sendo divididos em municípios, do ponto de vista do executivo, mas em comarcas, do ponto de vista do judiciário.

Como, então, lidar com toda essa complexidade no modelo conceitual? Usando o padrão *Hierarquia Organizacional*.

A solução consiste em não considerar mais os diferentes tipos de organização como conceitos, mas como instâncias de um conceito único: a *estrutura organizacional*, como na Figura 7.55.

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: origem e destino, como na Figura 7.51. Quando uma quantidade da unidade origem tiver de ser convertida em uma quantidade da unidade destino, divide-se seu valor pelo valor da razão.

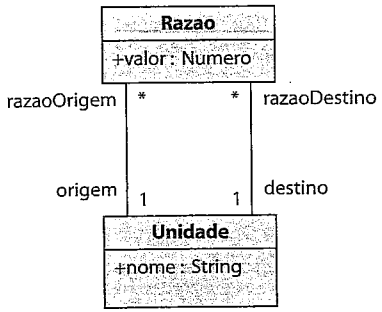

Figura 7.51: Unidades com razão de conversão.

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.

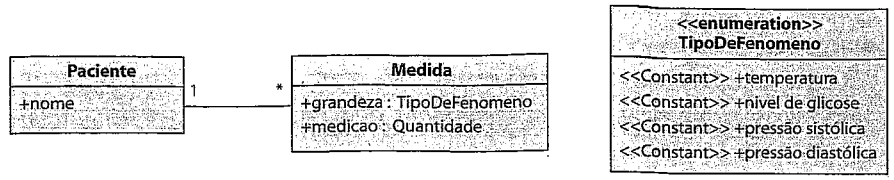

Figura 7.52: Definição e uso do padrão Medida.

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.

Na Figura 7.47a, os atributos nomeComprador, cpfComprador e endereco-Comprador vão se repetir em diferentes compras quando o comprador for o mesmo. Esse tipo de situação pode ser eliminado com a separação do conceito não coeso em dois conceitos associados, como na Figura 7.47b.

7.6.2. Classes de Especificação

Um caso especial de baixa coesão também ocorre quando se confunde um objeto com sua especificação. Até o momento, no sistema Livir, não foi discutida a diferença entre o conceito de obra literária e *cópia* de obra literária. Ambos os conceitos são tratados simplesmente como Livro, sem distinção. Mas ambos são entidades distintas e devem ser diferenciados. Por exemplo, título, ISBN, autor, preço de capa etc. aplicam-se à obra literária, pois se fossem aplicados a cada cópia iriam se repetir nas instâncias. Outra evidência de que se trata de conceitos diferentes é que, quando um livro é meramente reservado por não haver em estoque, reserva-se a obra literária, mas quando um livro é vendido, é vendida a cópia física.

Essa situação é muito frequente: muitas vezes produtos ou itens físicos compartilham uma especificação comum. Especificação e item físico devem ser modelados como dois conceitos separados, unidos por uma associação de um para muitos, como na Figura 7.48.

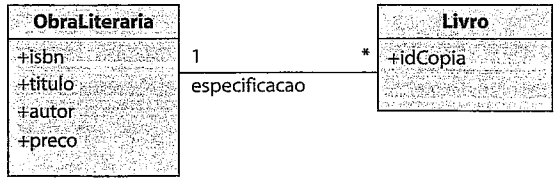

Figura 7.48: Uma classe com sua classe de especificação.

É possível que uma classe possua mais de uma especificação. Por exemplo, cópias de livros, além de serem especificadas pela obra literária, podem ser especificadas pelo seu estado (novo, usado, muito usado etc.). Em função do estado, um percentual de desconto pode ser aplicado ao livro. Assim, estado de uso seria uma classe com algumas instâncias que especificam cópias físicas de livros e definem seus percentuais de desconto, como na Figura 7.49.

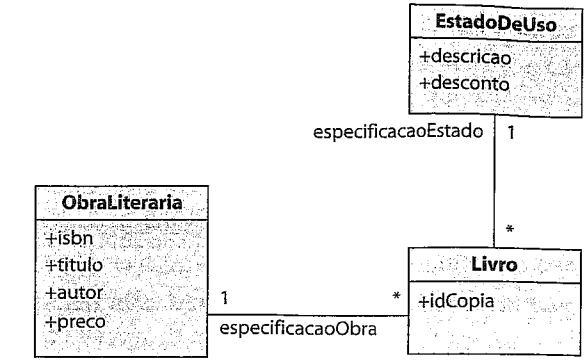

Figura 7.49: Uma classe com duas classes de especificação.

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.

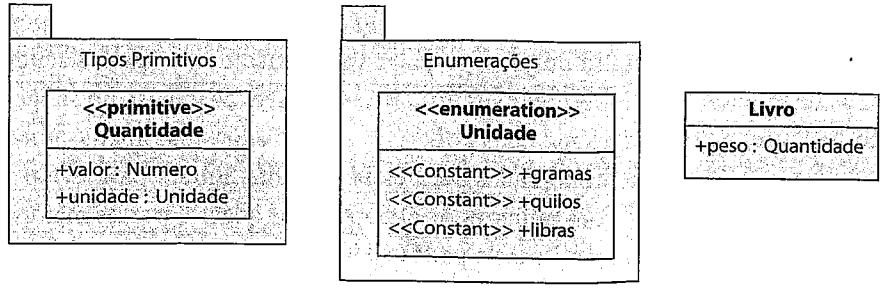

Figura 7.50: Definição e uso de Quantidade.

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.

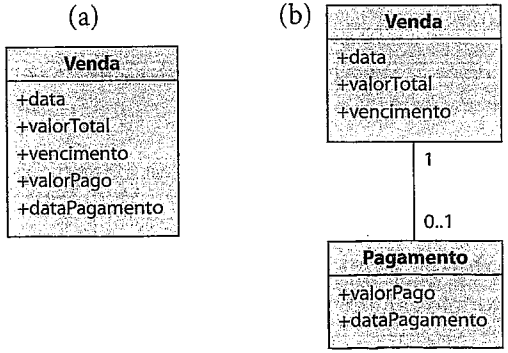

Figura 7.45: (a) Uma classe com baixa coesão por ter atributos dependentes de outros. (b) Uma solução de modelagem com classes mais coesas.

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.

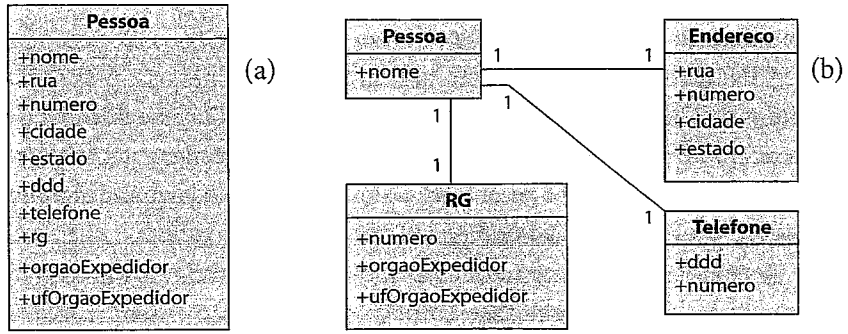

Figura 7.46: (a) Uma classe com baixa coesão por ter grupos de atributos fortemente correlacionados. (b) Uma solução com melhor coesão.

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.

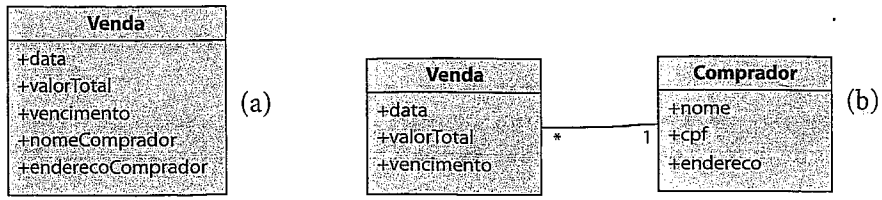

Figura 7.47: (a) Uma classe com baixa coesão por ter atributos que repetem valores nas instâncias. (b) Uma solução com melhor coesão.

Porém, observa-se que, com a transição de estados, alguns atributos e associações deixam de existir, como, por exemplo, as datas previstas, que são substituídas por datas reais, e a associação da reserva com um tipo de quarto, que é substituída pela associação da hospedagem com um quarto real.

Esse fato faz com que seja necessário usar um padrão de projeto (*design pattern*) conhecido como estado (*state*). De acordo com esse padrão, deve-se, primeiramente, separar o conceito de seu estado. Os atributos e associações que são comuns a todos os estados devem ser colocados no conceito original. Já os atributos e associações que são específicos dos estados devem ser modelados apenas nos estados onde devem ocorrer. Esses estados específicos são especializações de uma classe abstrata, como na Figura 7.44.

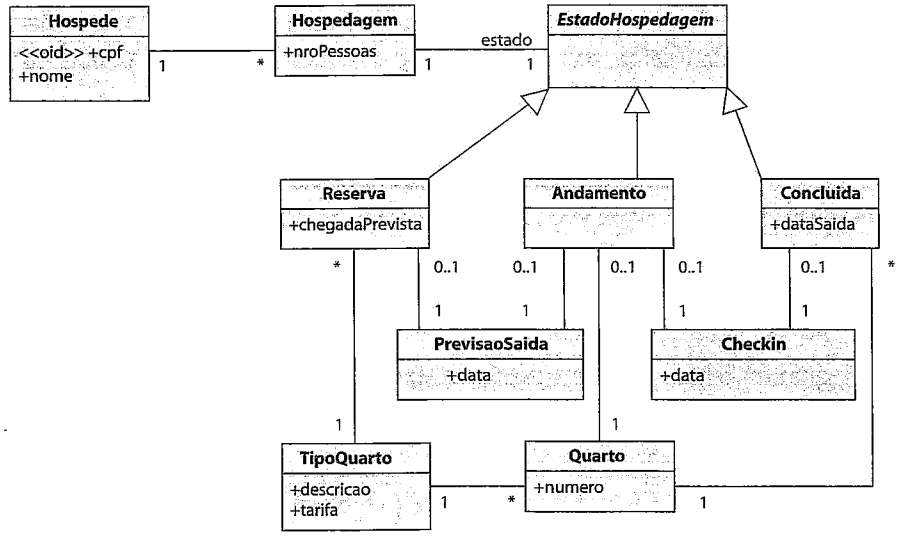

Figura 7.44: Modelagem de estados não monotônicos usando o padrão Estado.

Observa-se, na Figura 7.44, que o atributo nroPessoas é válido em qualquer estado de uma hospedagem, por isso é modelado no conceito Hospedagem. Há, depois, três estados possíveis:

a) Reserva, pelo qual uma hospedagem, além do número de pessoas, tem chegada e saída previstas e o tipo de quarto;
b) Andamento, pelo qual, além do número de pessoas, a hospedagem tem chegada efetiva, saída prevista e um quarto efetivo;
c) Concluida, pelo qual, além do número de pessoas, a hospedagem tem chegada e saída efetivas e um quarto efetivo.

Quando um atributo é específico a um único estado, ele pode ser representado como atributo do estado, como, por exemplo, chegadaPrevista e dataSaida na Figura 7.44, mas atributos comuns a dois ou mais estados devem ser representados como conceitos separados, para que possam ser associados aos diferentes estados. No caso da Figura 7.44, a saída prevista é comum aos estados Reserva e Andamento. Por isso, ela foi transformada em conceito e ligada a ambos os estados por associação de 0..1 para 1. Já a data de chegada é comum aos estados Andamento e Concluida e, portanto, foi representada como um conceito à parte (Checkin) associado a ambos os estados por uma associação de 0..1 para 1.

Para complementar o modelo, ainda seria necessário estabelecer que uma instância de PrevisaoSaida deve se associar a apenas uma instância dentre Reserva e Andamento de cada vez. Igualmente, Checkin só pode se associar a uma instância de Andamento ou uma instância de Concluida de cada vez. Mais adiante, será explicado como fazer isso com invariantes.

Observa-se também que um quarto pode se associar a, no máximo, uma hospedagem em andamento (associação para 0..1), mas pode se associar a um número qualquer de hospedagens concluídas (associação para *). Assim, já fica modelada também a propriedade temporal dessa associação, que tem multiplicidade 1 no presente, mas * no histórico.

7.6. Padrões de Análise

Fazer um modelo conceitual é muito mais do que amontoar conceitos, atributos e associações em um diagrama. Frequentemente percebe-se que a modelagem simplesmente não funciona porque fica complicado demais continuar a enriquecê-la.

Existem técnicas, porém, que diminuem a complexidade desses diagramas e, ao mesmo tempo, aumentam sua expressividade, permitindo que sejam modeladas, de forma simples, situações que, abordadas de forma ingênua, poderiam gerar modelos altamente complexos.

Essas técnicas são chamadas por Fowler (2003) de *padrões de análise* e podem ser concebidas como um caso especial de padrões de projeto (Gamma *et al.*, 1999) aplicados ao modelo conceitual.

Padrões de análise ou projeto não são regras que obrigatoriamente devem ser aplicadas, mas sugestões baseadas em experiências prévias. Cabe ao analista decidir quando aplicar ou não determinado padrão em sua modelagem.

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

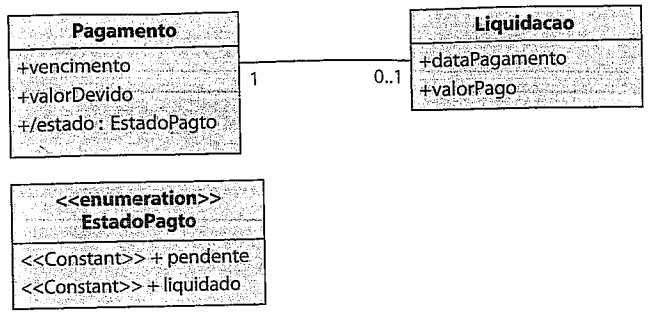

Figura 7.41: Forma eficaz de modelar classes modais com transição monotônica.

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.isEmpty() 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 isEmpty() aplicada a uma propriedade retorna true se o conjunto de objetos no respectivo papel da associação for vazio;

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.

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).

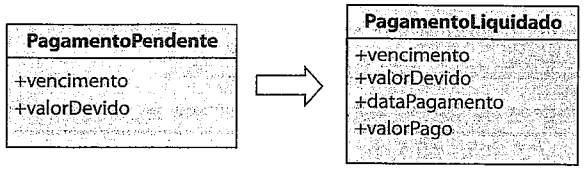

Figura 7.38: Um conceito com transição monotônica.

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.

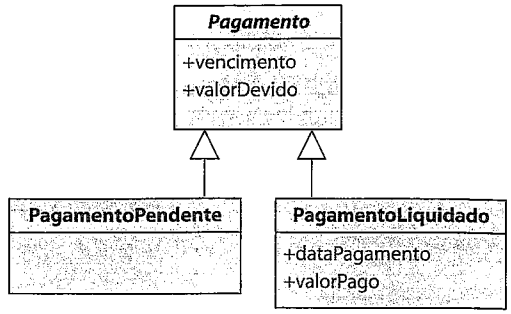

Figura 7.39: Forma inconveniente de modelar estados monotônicos com herança.

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, com valorPago definido e dataDePagamento indefinida.

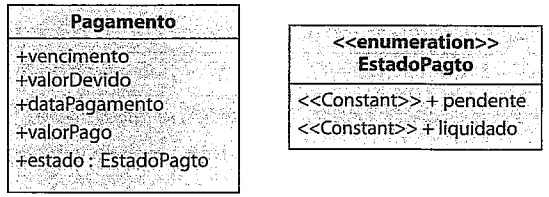

Figura 7.40: Modelagem inconveniente de estados monotônicos com uma única classe com atributos possivelmente nulos.

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 original 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.

exista de forma relativa, então se deve usar classe de associação. Por exemplo, ninguém pode ser *aluno*, simplesmente. Para alguém ser aluno deve haver uma instituição de ensino ou um professor. A pessoa tem de ser aluno *de* alguém ou alguma instituição.

Pode-se verificar, então, que é inadequado criar classes para representar tipos de pessoas que na verdade não são subclasses, mas papéis. Funcionários, professores, alunos, diretores, compradores etc. nunca poderiam ser subclasses de Pessoa. Esses conceitos só fazem sentido quando relacionados a outro conceito como empresa, escola, departamento etc. Deve-se ser professor *de* alguém, comprador *de* uma empresa, diretor *de* um departamento e assim por diante.

A diferença entre classe e associação e o uso de um conceito intermediário (transação) é bastante sutil. A Figura 7.36 mostra dois modelos alternativos parecidos, mas com uma pequena diferença.

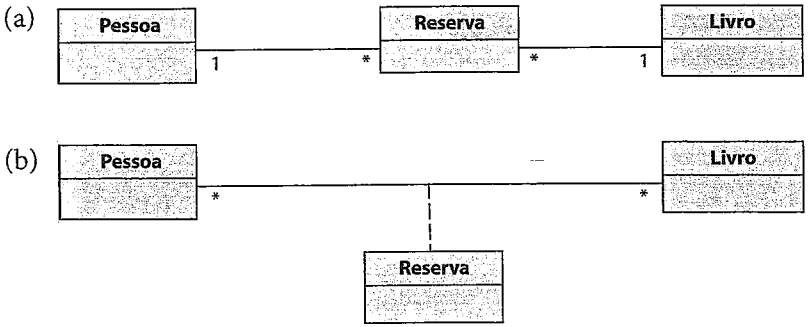

Figura 7.36: Modelagem de uma reserva (a) como um conceito e (b) como classe de associação.

No caso *a* da Figura 7.36, uma reserva associa uma pessoa a um livro. No caso *b* também. Mas, no caso *a*, uma pessoa pode ter várias reservas para o mesmo livro. No caso *b*, uma pessoa só pode ter, no máximo, uma única reserva para um mesmo livro.

7.5.3. Classes Modais

Classes modais ou *classes com estados* são usadas para modelar conceitos cujas instâncias podem mudar de um estado para outro ao longo de sua existência, alterando possivelmente sua estrutura, valores de atributos ou comportamento dos métodos. Embora algumas linguagens de programação, como *Smalltalk* (Goldberg & Robson, 1989), até permitem que instâncias de

objetos mudem de classe dinamicamente, isso não deve ser assumido como um princípio de modelagem, pois tais mudanças podem acarretar problemas estruturais complexos.

A rigor, uma instância, depois de criada, não poderá mudar de classe, visto que, mesmo que isso fosse possível, resta o problema de redefinir atributos e associações da nova instância. Assim, deve-se usar técnicas de modelagem que não exigem que objetos troquem dinamicamente de classe. Quando um objeto muda, é seu *estado* que muda, não sua classe.

São identificadas, aqui, três situações relacionadas à modelagem de estados:

a) *transição estável*: os diferentes estados de um objeto não afetam sua estrutura, mas apenas, possivelmente, valores de atributos;
b) *transição monotônica*: o objeto passa de um estado para outro e à medida que muda de estado vai ganhando novos atributos ou associações;
c) *transição não monotônica*: o objeto pode ganhar ou perder atributos ou associações à medida que muda de estado.

A modelagem da forma estável e da forma monotônica é simples. Já a modelagem da transição não monotônica exigirá a aplicação de um padrão de projeto denominado *Estado*, conforme será visto nas próximas subseções.

7.5.3.1. Transição Estável

Frequentemente, os diferentes estados de um objeto podem ser determinados através de um simples atributo. Por exemplo, o estado de uma Venda poderia ser modelado simplesmente com um atributo estado, tipado como uma enumeração de em andamento, concluída e paga.

Outro exemplo seria o atributo suspenso de um endereço, que indica que houve alguma devolução de mercadoria nesse endereço. O endereço fica nesse estado até que o comprador o atualize, pois possivelmente contém um erro.

Figura 7.37: Um conceito com transição de estado estável.

7.5.2. Classes de Associação

Uma questão frequentemente mal modelada diz respeito a conceitos que no senso comum poderiam ser considerados subtipos, mas que na verdade são papéis. Por exemplo, em uma livraria, poderiam ser identificados conceitos como Comprador e Funcionario. Ao descobrir que ambos têm atributos em comum como nome, endereco etc., um analista menos avisado poderia criar uma superclasse Pessoa para generalizar esses dois conceitos, o que resultaria em um problema muito sério de modelagem, pois não se trata de tipos diferentes de pessoas, mas de papéis que pessoas têm em relação a uma empresa.

Para entender por que isso seria um problema, pode-se supor que um funcionário da livraria deseja comprar livros. Nesse caso, ele estará se comportando como comprador. Assim, a solução errada, mas frequente, acaba sendo criar um segundo cadastro do funcionário, agora como comprador. Como consequência, a mesma pessoa ficará com dois registros no sistema: um como funcionário e outro como comprador. Isso gera redundância nos dados e é uma fonte de inconsistências, visto que, por exemplo, se essa pessoa mudar de endereço, pode ser que seja registrado apenas que o Funcionario mudou de endereço, mantendo-se o endereço antigo para o Comprador.

A solução, nesse caso, é considerar que existe uma Pessoa que pode se relacionar com uma Empresa de pelo menos duas formas: como comprador ou como funcionário. As propriedades específicas de comprador (cartões de crédito, por exemplo) e de funcionário (salário, por exemplo), seriam propriedades da associação e não da pessoa. Para representar essas propriedades, define-se uma classe de associação para cada associação específica, como na Figura 7.35b.

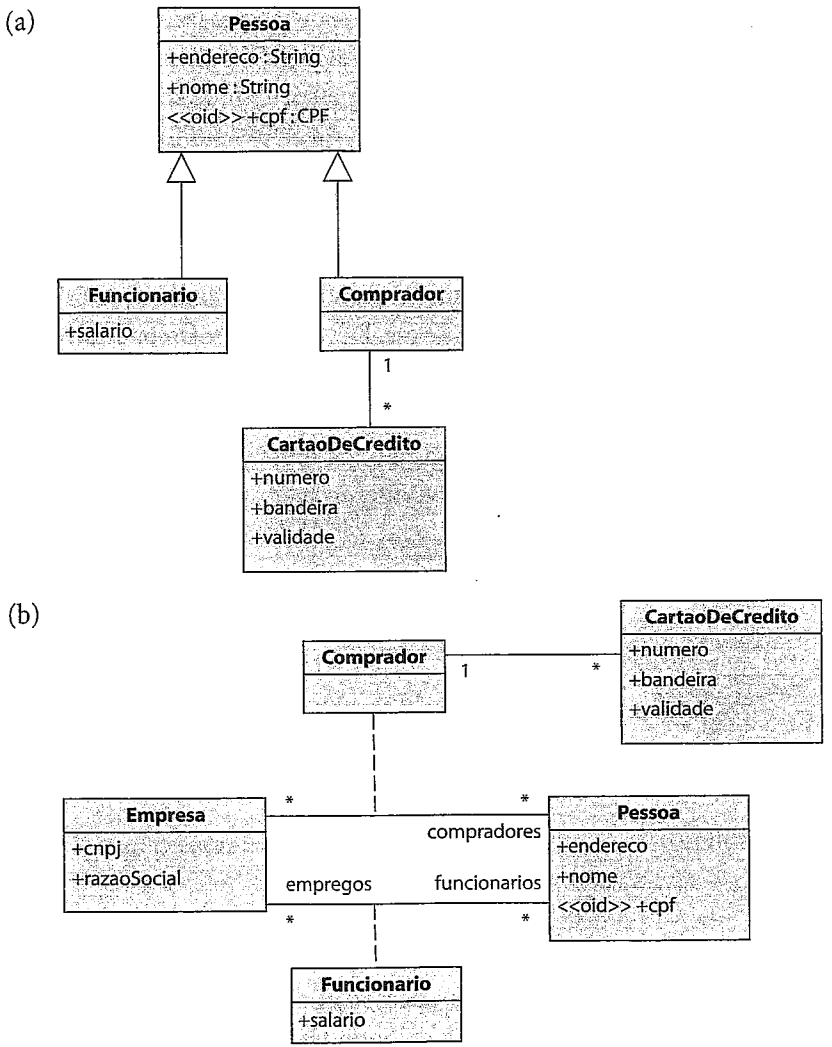

Figura 7.35: (a) Forma inadequada de representar papéis como herança. (b) Forma correta de representar papéis como classes de associação.

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 conceito ou não. Caso o subtipo só

Pode-se verificar então que, embora as associações simples entre classes do modelo conceitual se propaguem para as instâncias, a relação de herança não se propaga. Ela funciona como uma espécie de abreviação ou *macro* na definição das classes.

Assim, a única razão plausível para usar a relação de generalização existe quando é necessário fatorar as informações de duas ou mais classes (por "informações" das classes entende-se atributos, associações e, quando for caso, métodos).

Se a superclasse puder ter suas próprias instâncias, ela é uma classe normal. Porém, se não for possível instanciar diretamente objetos da superclasse, mas apenas das subclasses, então a superclasse é *abstrata*. Classes abstratas são representadas em UML com seu nome escrito em itálico ou com a restrição {abstract}.

A generalização deve ser usada sempre que um conjunto de classes X_1, ..., X_n, possuir diferenças específicas e semelhanças, de forma que as semelhanças possam ser agrupadas em uma *superclasse* X (generalização das subclasses X_1, ..., X_n), e as diferenças mantidas nas *subclasses* X_1, ..., X_n. Essa situação está exemplificada na Figura 7.32.

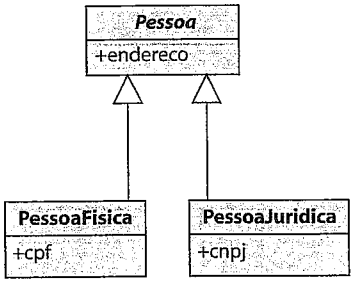

Figura 7.32: Representação esquemática de uma situação em que a herança pode ser usada.

Na Figura 7.32, a classe Pessoa é abstrata e não pode ter instâncias que não sejam instâncias de uma de suas subclasses. Instâncias de PessoaFisica têm cpf e endereco. Já instâncias de PessoaJuridica têm cnpj e endereco.

Não se deve usar a relação de generalização quando a superclasse não pode possuir nenhum atributo ou associação (Figura 7.33).

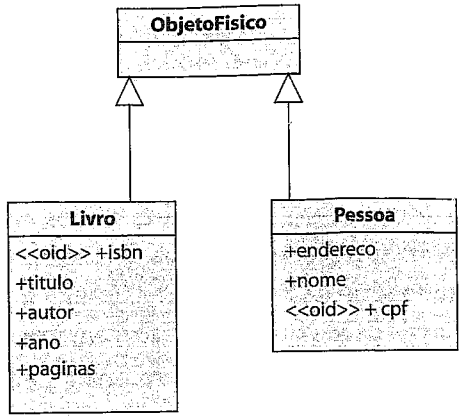

Figura 7.33: Situação em que a herança não deveria ser usada, pois não existem propriedades generalizadas.

Também não se deve usar herança quando as subclasses não possuem atributos ou associações que os diferenciem um do outro, como na Figura 7.34a. Nesses casos, usa-se apenas um atributo para diferenciar os tipos de pessoa, que estruturalmente são idênticos. Esse atributo diferenciador pode ser modelado como uma enumeração, como na Figura 7.34b.

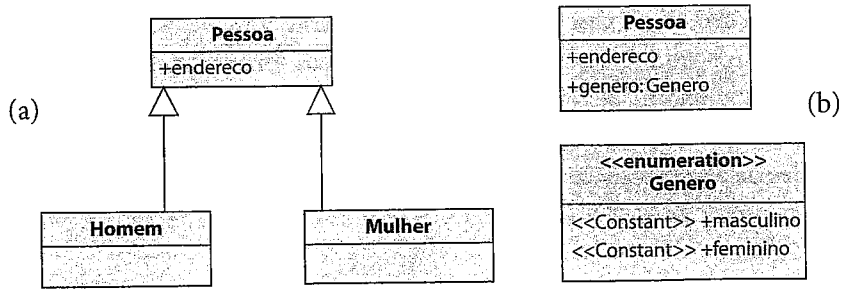

Figura 7.34: (a) Situação em que a herança não deve ser usada, pois não existem propriedades especializadas. (b) Modelagem alternativa mais adequada.

Além dessa regra, deve-se verificar, antes de decidir pelo uso da herança, se a generalização realmente representa uma classificação estrutural dos elementos, e não uma organização associativa ou temporal, como será visto nas seções seguintes.

No caso da Figura 7.30, a associação não deve ser ternária porque não existe uma relação de autor com editora independentemente do livro. Ou seja, o autor se relaciona ao livro e a editora se relaciona ao livro, mas o autor não se relaciona diretamente à editora.

7.5. Organização do Modelo Conceitual

A construção do modelo conceitual envolve mais do que simplesmente juntar conceitos, atributos e associações. Para que o modelo consista em uma representação fiel e organizada da informação gerenciada pelo sistema, é necessário que certas técnicas de modelagem sejam utilizadas.

As técnicas de organização de conceitos seguem três grupos distintos:

a) *estruturais*: representando relações de generalização estrutural de conceitos, como, por exemplo, Pessoa, generalizando PessoaFisica e PessoaJuridica;
b) *associativas*: representando relações de papéis associativos entre conceitos, como, por exemplo, Pessoa, podendo representar junto a uma empresa o papel de Comprador ou Funcionário;
c) *temporais*: representando relações entre estados de um conceito como, por exemplo, um Livro e os estados da Figura 2.6: *encomendado*, *em estoque*, *vendido* etc.

Analistas principiantes e mesmo alguns experientes tendem a pensar que a única forma de fatorar informações é com herança. Mas deve-se saber distinguir quando usar cada uma das técnicas referidas. Só se usa herança quando um conceito efetivamente tiver dois ou mais subtipos. Uma instância nunca pode mudar de um subtipo para outro. Ela *nasce* no subtipo e *morre* no subtipo.

O caso, por exemplo, de Comprador e Funcionario não deve ser modelado com herança, ou seja, essas classes não devem ser subclasses de Pessoa, porque ninguém nasce comprador nem nasce funcionário. Além disso, uma mesma pessoa pode ser simultaneamente comprador e funcionário da mesma empresa ou até de diferentes empresas. Usar herança nesse caso vai causar problemas conceituais muito grandes no sistema (duplicações de cadastros, por exemplo).

As subseções seguintes detalham as três formas de organização de conceitos.

7.5.1. Generalização, Especialização e Herança

Durante anos, o uso de herança foi considerado como o grande mote da orientação a objetos. O mais importante na construção de um sistema era a definição de uma boa hierarquia de classes. Linguagens como *Smalltalk* (Goldberg & Robson, 1989) se estruturam totalmente sobre uma hierarquia de classes.

Com o passar do tempo, essa ênfase foi perdendo força, pois se percebeu que o uso da herança nem sempre era a melhor solução para problemas de modelagem, e hoje a herança é considerada apenas mais uma ferramenta de modelagem que ajuda a fatorar informações que, de outra forma, ficariam repetidas em diferentes conceitos.

A herança, em orientação a objetos é obtida quando duas classes se relacionam através de uma associação especial denominada *generalização* (no sentido da classe mais específica – *subclasse* –, para a mais genérica – *superclasse*) ou *especialização* (no sentido inverso).

A relação de generalização tem uma natureza muito diferente das associações do modelo conceitual. Ela só existe entre as classes, mas não entre as instâncias dessas classes. Se duas classes como Pessoa e Automovel são ligadas por uma associação simples, como na Figura 7.17, então as instâncias de Pessoa serão ligadas às instâncias de Automovel pelas realizações concretas dessa associação. Porém, se duas classes como Pessoa e PessoaFisica são ligadas pela relação de generalização, como na Figura 7.31, então as instâncias de PessoaFisica também *são* instâncias de Pessoa, ou seja, não existem instâncias de PessoaFisica ligadas a supostas instâncias de Pessoa: as instâncias de PessoaFisica são *simultaneamente* instâncias de Pessoa. Assim, toda instância de PessoaFisica tem atributos endereco e cpf. O inverso porém não ocorre, ou seja, nem toda instância de Pessoa é uma instância de PessoaFisica.

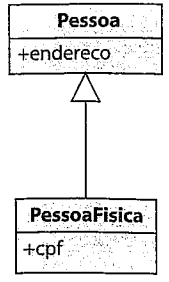

Figura 7.31: Generalização (herança).

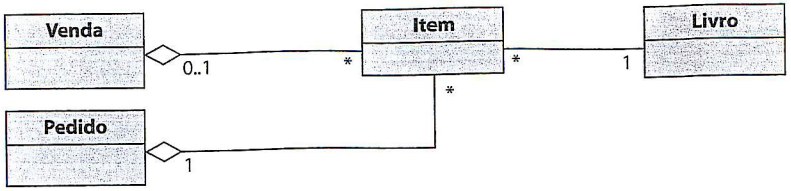

Figura 7.28: Agregação compartilhada.

O losango branco indica uma agregação compartilhada em que o componente pode ser agregado a vários conceitos ao mesmo tempo. Na Figura 7.28, um dado Item necessariamente faz parte de um Pedido, mas também poderá fazer parte de uma Venda.

Quanto à multiplicidade de papel do lado do agregador (lado onde está o losango), deve-se observar não ser possível uma multiplicidade diferente de 1 ou 0..1 quando a associação for de composição.

Agregação e composição são associações especiais e devem ser usadas com muita parcimônia no modelo, ou seja, só se deve usar quando se tem certeza.

Erros comuns são associar por agregação elementos que não são *parte-todo*. Por exemplo, um comprador não é parte de uma venda, visto que um comprador é um ente físico e uma venda é uma transação não física. Agregações e composições devem unir elementos de mesma natureza. Uma venda associa-se a um comprador, mas não é *feita de* comprador.

Existem poucas vantagens práticas na definição de agregações ou composições. Por isso, seu uso deve ser minimizado. Dentre as vantagens, pode-se citar basicamente o fato de que elementos agregados usualmente possuem atributos que se combinam nas partes e são derivados no todo. Por exemplo, o valor total de uma venda é a soma do valor dos seus itens. O peso total de uma encomenda é o peso de seus itens. Quando um automóvel é vendido, todos os seus componentes são vendidos. E assim por diante.

7.4.7. Associações n-árias

Pode-se dizer que a grande maioria das associações é binária. Porém pode haver situações em que devem ser criadas associações ente três ou mais classes. A representação gráfica dessas associações consiste em um polígono ligando por arestas todas as classes. Um exemplo de associação ternária é apresentado na Figura 7.29.

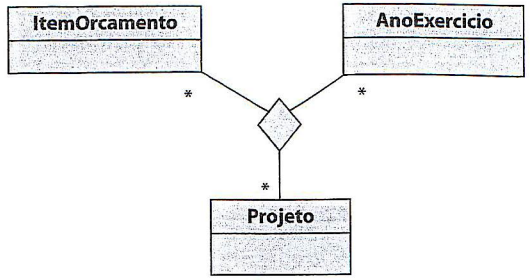

Figura 7.29: Exemplo de associação ternária.

Esse exemplo foi tirado de uma aplicação real de controle de orçamento. Cada projeto associa-se a um item de orçamento e um exercício. Para cada ano-exercício pode haver um variado número de itens de orçamento por projeto. Isso tem de ser representado por uma associação ternária.

Esses casos são *muito raros*, mas podem existir. Antes de decidir usar uma associação n-ária, que pode gerar inconvenientes no projeto e implementação, deve-se verificar se não é o caso de várias associações binárias. Por exemplo, o caso da Figura 7.30a não é uma associação ternária, mas duas associações binárias. A opção binária deve ser sempre preferida.

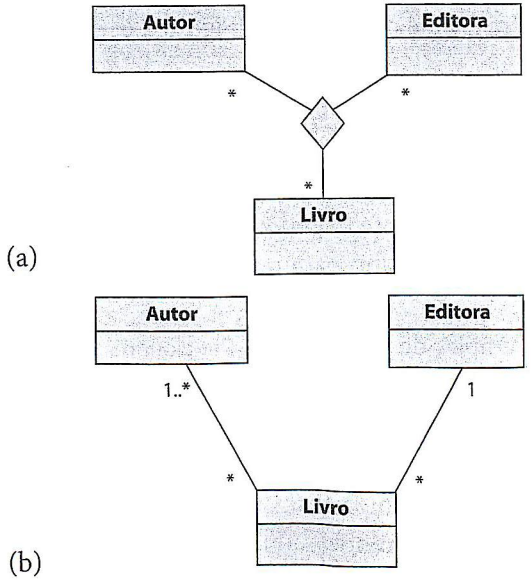

Figura 7.30: (a) Uma associação ternária indevida. (b) A solução correta com duas associações binárias.