Capítulo 6 – Modelagem Conceitual Estrutural

Um modelo conceitual é uma abstração da realidade segundo uma conceituação. Ele pode ser usado para comunicação, aprendizado e análise de aspectos relevantes do domínio subjacente (GUIZZARDI, 2005). O modelo conceitual estrutural de um sistema tem por objetivo descrever as informações que esse sistema deve representar e gerenciar.

A modelagem conceitual é a atividade de descrever alguns dos aspectos do mundo físico e social a nossa volta, com o propósito de entender e comunicar. Os modelos resultantes das atividades de modelagem conceitual são essencialmente destinados a serem usados por pessoas e não por máquinas (MYLOPOULOS, 1992). Assim, modelos conceituais devem ser concebidos com foco no domínio do problema e não no domínio da solução e, por conseguinte, um modelo conceitual estrutural é um artefato do domínio do problema e não do domínio da solução. As informações a serem capturadas em um modelo conceitual estrutural devem existir independentemente da existência de um sistema computacional para tratá-las. Assim, o modelo conceitual deve ser independe da solução computacional a ser adotada para resolver o problema e deve conter apenas os elementos de informação referentes ao domínio do problema em questão. Elementos da solução, tais como interfaces, formas de armazenamento e comunicação, devem ser tratados apenas na fase de projeto (WAZLAWICK, 2004).

Uma vez que requisitos não-funcionais de produto (atributos de qualidade) são inerentes à solução computacional, de maneira geral, eles não são alvo na modelagem conceitual. Ou seja, não se consideram elementos de informação para tratar aspectos como desempenho, segurança, confiabilidade, formas de armazenamento etc. Esses atributos de qualidade do produto são considerados posteriormente, na fase de projeto.

Os elementos de informação básicos da modelagem conceitual estrutural são os tipos de entidades e os tipos de relacionamentos. A identificação de quais os tipos de entidades e os tipos de relacionamentos que são relevantes para um particular sistema de informação é uma meta crucial da modelagem conceitual (OLIVÉ, 2007).

Diferentes tipos de modelos podem ser usados na modelagem conceitual estrutural, tais como Diagramas de Entidades e Relacionamentos (ER) e Diagramas de Classe da UML. Este último é baseado no paradigma orientado a objetos.

Na modelagem conceitual segundo o paradigma orientado a objetos, tipos de entidades são modelados como classes. Tipos de relacionamentos são modelados como atributos e associações. Assim, o propósito da modelagem conceitual estrutural orientada a objetos é definir as classes, atributos e associações que são relevantes para tratar o problema a ser resolvido. Para tal, as seguintes tarefas devem ser realizadas:

- Identificação de Classes
- Identificação de Atributos e Associações
- Especificação de Hierarquias de Generalização/Especialização

É importante notar que essas atividades são dependentes umas das outras e que, durante o desenvolvimento, elas são realizadas de forma paralela e iterativa, sempre visando ao entendimento do domínio do problema, desconsiderando aspectos de implementação.

Este capítulo aborda a modelagem conceitual estrutural quando realizada segundo o paradigma orientado a objetos. A Seção 6.1 apresenta os principais conceitos da orientação a objetos. A Seção 6.2 discute como identificar classes. A Seção 6.3 aborda a identificação de atributos e associações. Finalmente, a Seção 6.4 discute a especificação de hierarquias de generalização/especialização.

6.1 – Conceitos da Orientação a Objetos

Para conduzir a modelagem conceitual estrutural, é necessário adotar um paradigma de desenvolvimento. Um dos paradigmas mais adotados atualmente é a orientação a objetos (OO). Segundo esse paradigma, o mundo é visto como sendo composto por *objetos*, onde um objeto é uma entidade que combina estrutura de dados e comportamento funcional. No paradigma OO, os sistemas são modelados como objetos que interagem.

A orientação a objetos oferece um número de conceitos bastante apropriados para a modelagem de sistemas. Os modelos baseados em objetos são úteis para a compreensão de problemas, para a comunicação com os especialistas e usuários das aplicações, e para a realização das tarefas ao longo do processo de desenvolvimento de software. O paradigma OO utiliza uma perspectiva humana de observação da realidade, incluindo objetos, classificação e compreensão hierárquica.

O mundo real é extremamente complexo. Quanto mais de perto o observamos, mais claramente percebemos sua complexidade. A orientação a objetos tenta gerenciar a complexidade inerente dos problemas do mundo real, abstraindo conhecimento relevante e encapsulando-o em objetos. De fato, alguns princípios básicos gerais para a administração da complexidade norteiam o paradigma orientado a objetos, entre eles abstração, encapsulamento e modularidade.

Uma das principais formas do ser humano lidar com a complexidade é através do uso de **abstrações**. As pessoas tipicamente tentam compreender o mundo, construindo modelos mentais de partes dele. Tais modelos são uma visão simplificada de algo, onde apenas os elementos relevantes são considerados. Modelos, portanto, são mais simples do que os complexos sistemas que eles modelam.

Seja o exemplo de um mapa representando um modelo de um território. Um mapa é útil porque abstrai apenas as características do território que se deseja modelar. Se um mapa incluísse todos os detalhes do território, provavelmente teria o mesmo tamanho do território e, portanto, não serviria a seu propósito.

Da mesma forma que um mapa precisa ser significativamente menor que o território que ele mapeia, incluindo apenas as informações selecionadas, um modelo conceitual deve abstrair apenas as características relevantes de um sistema para seu entendimento. Assim, pode-se definir abstração como sendo o princípio de ignorar aspectos não relevantes de um assunto, segundo a perspectiva de um observador, tornando possível uma concentração maior nos aspectos principais do mesmo. De fato, a abstração consiste na seleção que um observador faz de alguns aspectos de um assunto, em detrimento de outros que não demonstram ser relevantes para o propósito em questão. A Figura 6.1 ilustra o conceito de abstração.

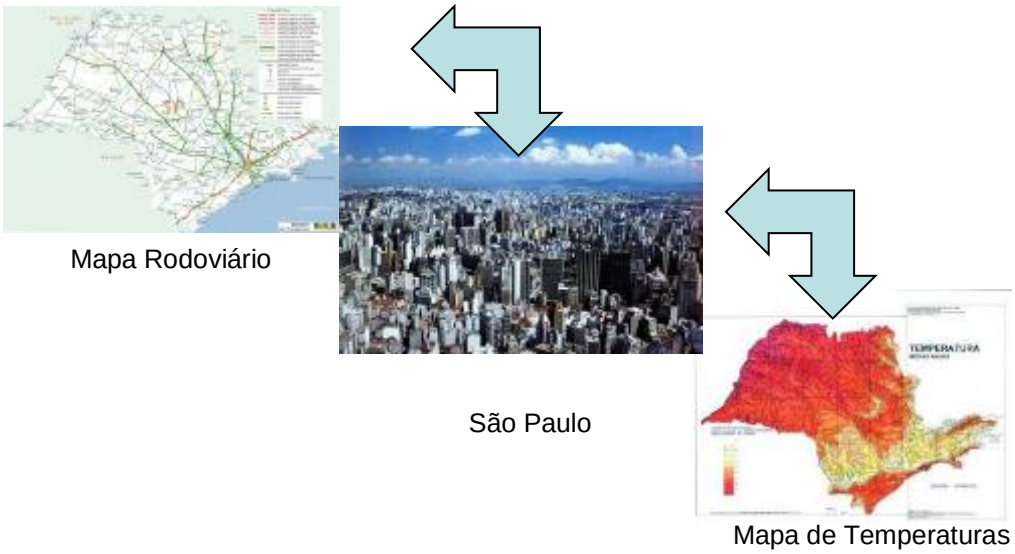

Figura 6.1 – Ilustração do Conceito de Abstração.

No mundo real, um objeto pode interagir com outro sem conhecer seu funcionamento interno. Uma pessoa, por exemplo, utiliza um forno de micro-ondas sem saber efetivamente qual a sua estrutura interna ou como seus mecanismos internos são ativados. Para utilizá-lo, basta saber usar seu painel de controle (a interface do aparelho) para realizar as operações de ligar/desligar, informar o tempo de cozimento etc. Como essas operações produzem os resultados, não interessa ao cozinheiro, como ilustra a Figura 6.2. Na OO, a este princípio de administração da complexidade, dá-se o nome de encapsulamento.

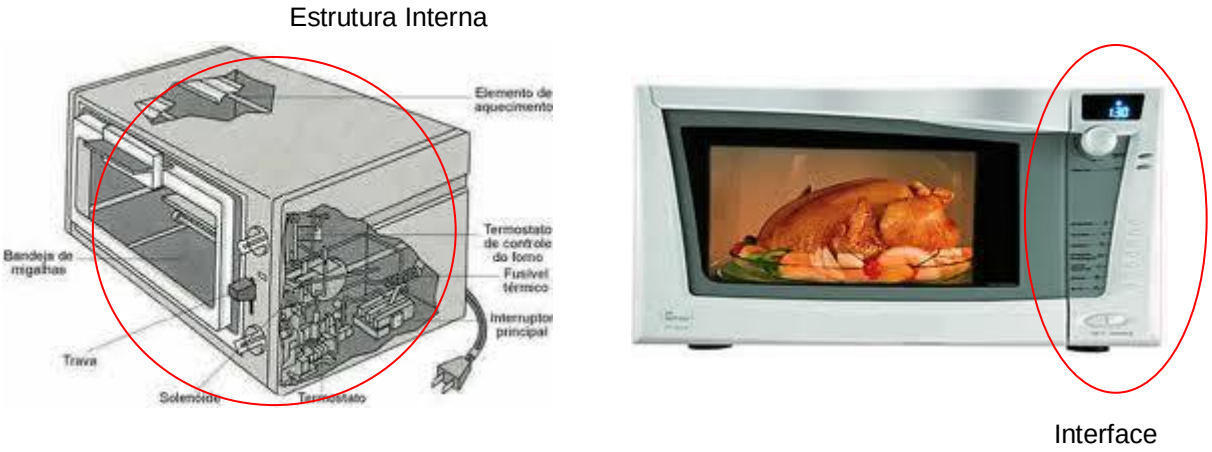

Figura 6.2 – Ilustração do Conceito de Encapsulamento.

O **encapsulamento** consiste na separação dos aspectos externos de um objeto, acessíveis por outros objetos, de seus detalhes internos de implementação, que ficam ocultos dos demais objetos (BLAHA; RUMBAUGH, 2006). A interface de um objeto deve ser definida de forma a revelar o menos possível sobre o seu funcionamento interno.

Abstração e encapsulamento são conceitos complementares: enquanto a abstração enfoca o comportamento observável de um objeto, o encapsulamento oculta a implementação que origina esse comportamento. Encapsulamento é frequentemente conseguido através da ocultação de informação, isto é, escondendo detalhes que não contribuem para suas características essenciais. Tipicamente, em um sistema OO, a estrutura de um objeto e a

implementação de seus métodos são encapsuladas (BOOCH, 1994). Assim, o encapsulamento serve para separar a interface contratual de uma abstração e sua implementação. Os usuários têm conhecimento apenas das operações que podem ser requisitadas e precisam estar cientes apenas do *quê* as operações realizam e não *como* elas estão implementadas.

A principal motivação para o encapsulamento é garantir estabilidade aos sistemas. Um encapsulamento bem feito pode servir de base para a localização de decisões de projeto que necessitam ser alteradas. Uma operação pode ter sido implementada de maneira ineficiente e, portanto, pode ser necessário escolher um novo algoritmo. Se a operação está encapsulada, apenas o objeto que a define precisa ser modificado, garantindo estabilidade ao sistema.

Por fim, os métodos de desenvolvimento de software buscam obter sistemas modulares, isto é, construídos a partir de elementos que sejam autônomos e conectados por uma estrutura simples e coerente. **Modularidade** visa à obtenção de sistemas decompostos em um conjunto de módulos coesos e fracamente acoplados e é crucial para se obter sistemas fáceis de manter e estender. Abstração, encapsulamento e modularidade são princípios sinergéticos, isto é, ao se trabalhar bem com um deles, está-se aperfeiçoando os outros também.

O paradigma OO procura fazer uso dos mecanismos de administração da complexidade para modelar, projetar e implementar sistemas. De maneira simples, o paradigma OO traz uma visão de mundo em que os fenômenos e domínios são vistos como coleções de objetos interagindo entre si. Essa visão simplista do paradigma OO esconde conceitos importantes da orientação a objetos que são a base para o desenvolvimento OO, tais como classes, associações, generalização etc. A seguir os principais conceitos da orientação a objetos são discutidos.

Objetos

O mundo real é povoado por entidades que interagem entre si, onde cada uma delas desempenha um papel específico. Na orientação a objetos, essas entidades são ditas *objetos*. Objetos podem ser coisas concretas ou abstratas, tais como um carro, uma reserva de passagem aérea, uma organização etc.

Do ponto de vista da modelagem de sistemas, um objeto é uma entidade que incorpora uma abstração relevante no contexto de uma aplicação. Um objeto possui um estado (informação), exibe um comportamento bem definido (dado por um número de operações para examinar ou alterar seu estado) e tem identidade única. O estado de um objeto compreende o conjunto de suas propriedades, associadas a seus valores correntes. Propriedades de objetos são geralmente referenciadas como atributos e associações. Portanto, o estado de um objeto diz respeito aos seus atributos/associações e aos valores a eles associados. Operações são usadas para recuperar ou manipular a informação de estado de um objeto e se referem apenas às estruturas de dados do próprio objeto, não devendo acessar diretamente estruturas de outros objetos. Caso a informação necessária para a realização de uma operação não esteja disponível, o objeto terá de colaborar com outros objetos.

A comunicação entre objetos dá-se por meio de *troca de mensagens*. Para requisitar uma operação de um objeto, é necessário enviar uma mensagem para ele. Uma mensagem é composta do nome da operação sendo requisitada e dos argumentos requeridos. Assim, o comportamento de um objeto representa como esse objeto reage às mensagens a ele enviadas. Em outras palavras, o conjunto de mensagens a que um objeto pode responder representa o seu comportamento. Um objeto é, pois, uma entidade que tem seu estado representado por um conjunto de atributos e associações (uma estrutura de informação) e seu comportamento representado por um conjunto de operações.

Cada objeto tem uma identidade própria que lhe é inerente. Todos os objetos têm existência própria, ou seja, dois objetos são distintos, mesmo se seus estados e comportamentos forem iguais. A identidade de um objeto transcende os valores correntes de suas propriedades.

Classes

No mundo real, diferentes objetos desempenham um mesmo papel. Seja o caso de duas cadeiras iguais. Apesar de serem objetos diferentes, elas compartilham uma mesma estrutura e um mesmo comportamento. Entretanto, não há necessidade de se despender tempo modelando cada cadeira. Basta definir, em um único lugar, um modelo descrevendo a estrutura e o comportamento desses objetos. A esse modelo dá-se o nome de *classe*. Uma classe descreve um conjunto de objetos com a mesma estrutura (atributos e associações), o mesmo comportamento (operações) e a mesma semântica. Objetos que se comportam da maneira especificada pela classe são ditos *instâncias* dessa classe.

Todo objeto pertence a uma classe, ou seja, é instância de uma classe. De fato, a orientação a objetos norteia o processo de desenvolvimento através da *classificação de objetos*, isto é, objetos são agrupados em classes, em função de exibirem características similares, sem, no entanto, perda de sua individualidade, como ilustra a Figura 6.3. Assim, a modelagem orientada a objetos consiste, basicamente, na definição de classes. O comportamento e a estrutura de informação de uma instância são definidos pela sua classe. Objetos com propriedades e comportamento idênticos são descritos como instâncias de uma mesma classe, de modo que a descrição de suas propriedades possa ser feita uma única vez, de forma concisa, independentemente do número de objetos que tenham tais propriedades em comum. Deste modo, uma classe captura as características comuns a todas as suas instâncias.

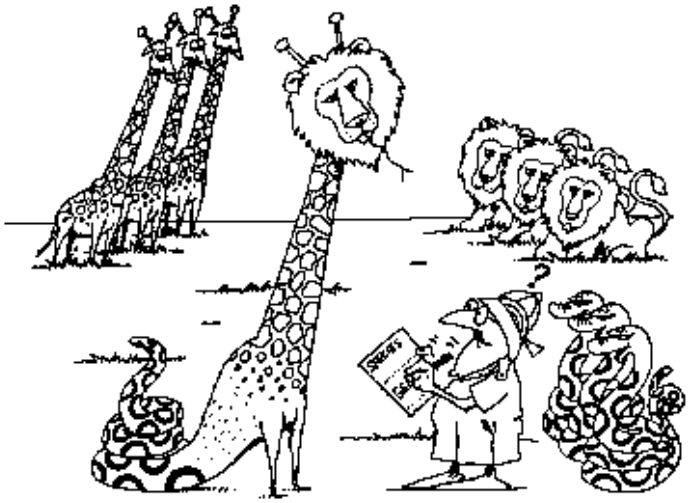

Figura 6.3 – Ilustração do conceito de Classificação (BOOCH, 1994).

Enquanto um objeto individual é uma entidade real, que executa algum papel no sistema como um todo, uma classe captura a estrutura e o comportamento comuns a todos os objetos que ela descreve. Assim, uma classe serve como uma espécie de contrato que deve ser estabelecido entre uma abstração e todos os seus clientes.

Ligações e Associações

Em qualquer sistema, objetos relacionam-se uns com os outros. Por exemplo, em "*o empregado João trabalha no Departamento de Pessoal*", temos um relacionamento entre o objeto *empregado João* e o objeto *Departamento de Pessoal*.

Ligações e associações são meios de se representar relacionamentos entre objetos e entre classes, respectivamente. Uma ligação é uma conexão entre objetos. No exemplo anterior, há uma ligação entre os objetos *João* e *Departamento de Pessoal*. Uma associação, por sua vez, descreve um conjunto de ligações com estrutura e semântica comuns. No exemplo anterior, há uma associação entre as classes *Empregado* e *Departamento*. Todas as ligações de uma associação interligam objetos das mesmas classes e, assim, uma associação descreve um conjunto de potenciais ligações da mesma maneira que uma classe descreve um conjunto de potenciais objetos (BLAHA; RUMBAUGH, 2006).

Generalização / Especialização

Muitas vezes, um conceito geral pode ser especializado, adicionando-se a ele novas características. Seja o exemplo do conceito de *estudante* no contexto de uma universidade. De modo geral, há características que são intrínsecas a quaisquer estudantes da universidade. Entretanto, é possível especializar esse conceito para mostrar especificidades de subtipos de estudantes, tais como estudantes de graduação e estudantes de pós-graduação.

De maneira inversa, pode-se extrair de um conjunto de conceitos características comuns que, quando generalizadas, formam um conceito geral. Por exemplo, ao se avaliar os conceitos de carros, motos, caminhões e ônibus, pode-se notar que eles têm características comuns que podem ser generalizadas em um supertipo *veículo automotor terrestre*.

As abstrações de especialização e generalização são muito úteis para a estruturação de sistemas. Com elas, é possível construir hierarquias de classes. A *herança* é um mecanismo para modelar similaridades entre classes, representando as abstrações de generalização e especialização. Através da herança, é possível tornar explícitos atributos, associações e operações comuns em uma hierarquia de classes. O mecanismo de herança possibilita reutilização, captura explícita de características comuns e definição incremental de classes. No que se refere à definição incremental de classes, a herança permite conceber uma nova classe como um refinamento de outras classes. A nova classe pode herdar as similaridades e definir apenas as novas características.

A herança é, portanto, um relacionamento entre classes (em contraposição às associações que representam relacionamentos entre objetos das classes), no qual uma classe compartilha a estrutura e o comportamento definidos em outras (uma ou mais) classes. A classe que herda características[12] é dita *subclasse* e a que fornece as características, *superclasse*. Desta forma, a herança representa uma hierarquia de abstrações na qual uma subclasse herda de uma ou mais superclasses.

Tipicamente, uma subclasse aumenta ou redefine características de suas superclasses. Assim, se uma classe *B* herda de uma classe *A*, todas as características descritas em *A* tornam-se automaticamente parte de *B*, que ainda é livre para acrescentar novas características para seus propósitos específicos.

A generalização permite abstrair, a partir de um conjunto de classes, uma classe mais geral contendo todas as características comuns. A especialização é a operação inversa e, portanto, permite especializar uma classe em um número de subclasses, explicitando as diferenças entre as novas subclasses. Deste modo é possível compor uma hierarquia de classes. Esses tipos de relacionamento são conhecidos também como relacionamentos "*é um tipo de*",

[12] O termo *característica* é usado aqui para designar estrutura (atributos e associações) e comportamento (operações).

onde um objeto da subclasse também "*é um tipo de*" objeto da superclasse. Neste caso, uma instância da subclasse é dita uma *instância indireta* da superclasse. Quando uma subclasse herda características de uma única superclasse, tem-se *herança simples*. Quando uma classe é definida a partir de duas ou mais superclasses, tem-se *herança múltipla*.

Mensagens, Operações e Métodos

A abstração incorporada por um objeto é caracterizada por um conjunto de operações que podem ser requisitadas por outros objetos, ditos clientes. Métodos são implementações reais de operações. Para que um objeto realize alguma tarefa, é necessário enviar uma mensagem a ele, solicitando a realização de uma operação. Um cliente só pode acessar um objeto através da emissão de mensagens, isto é, ele não pode acessar ou manipular diretamente os dados associados ao objeto. Os objetos podem ser complexos e o cliente não precisa tomar conhecimento de sua complexidade interna. O cliente precisa saber apenas como se comunicar com o objeto e como ele reage. Assim, garante-se o encapsulamento.

As mensagens são o meio de comunicação entre objetos e são responsáveis pela ativação de todo e qualquer processamento. Dessa forma, é possível garantir que clientes não serão afetados por alterações nas implementações de um objeto que não alterem o comportamento esperado de seus serviços.

Classes e Operações Abstratas

Nem todas as classes são projetadas para instanciar objetos. Algumas são usadas simplesmente para organizar características comuns a diversas classes. Tais classes são ditas *classes abstratas*. Uma classe abstrata é desenvolvida basicamente para ser herdada por outras classes. Ela existe meramente para que um comportamento comum a um conjunto de classes possa ser colocado em uma localização comum e definido uma única vez. Assim, uma classe abstrata não possui instâncias diretas, mas suas classes descendentes *concretas*, sim. Uma classe concreta é uma classe passível de instanciação, isto é, que pode ter instâncias diretas. Uma classe abstrata pode ter subclasses também abstratas, mas as classes-folhas na árvore de herança devem ser classes concretas.

Classes abstratas podem ser projetadas de duas maneiras distintas. Primeiro, elas podem prover implementações completamente funcionais do comportamento que pretendem capturar. Alternativamente, elas podem prover apenas a definição de um protocolo para uma operação, sem apresentar um método correspondente. Tal operação é dita uma *operação abstrata*. Neste caso, a classe abstrata não é completamente implementada e todas as suas subclasses concretas são obrigadas a prover uma implementação para suas operações abstratas. Assim, diz-se que uma operação abstrata define apenas a assinatura[13] a ser usada nas implementações que as subclasses deverão prover, garantindo, assim, uma interface consistente. Métodos que implementam uma operação genérica têm de ter a mesma semântica.

Uma classe concreta não pode conter operações abstratas, porque senão seus objetos teriam operações indefinidas. Assim, toda classe que possuir uma operação abstrata não pode ter instâncias diretas e, portanto, obrigatoriamente é uma classe abstrata.

[13] nome da operação, parâmetros e retorno

6.2 – Identificação de Classes

Classificação é o meio pelo qual os seres humanos estruturam a sua percepção do mundo e seu conhecimento sobre ele. Sem ela, não é possível nem entender o mundo a nossa volta nem agir sobre ele. Classificação assume a existência de tipos e de objetos a serem classificados nesses tipos. Classificar consiste, então, em determinar se um objeto é ou não uma instância de um tipo. A classificação nos permite estruturar conhecimento sobre as coisas em dois níveis: tipos e instâncias. No nível de tipos, procuramos encontrar as propriedades comuns a todas as instâncias de um tipo. No nível de instância, procuramos identificar o tipo do qual o objeto é uma instância e os valores particulares das propriedades desse objeto (OLIVÉ, 2007).

Tipos de entidade são um dos mais importantes elementos em modelos conceituais. Definir os tipos de entidade relevantes para um particular sistema de informação é uma tarefa crucial na modelagem conceitual. Um tipo de entidade pode ser definido como um tipo cujas instâncias em um dado momento são objetos individuais identificáveis que se consideram existir no domínio naquele momento. Um objeto pode ser instância de vários tipos ao mesmo tempo (OLIVÉ, 2007). Por exemplo, seja o caso dos tipos *Estudante* e *Funcionário* em um sistema de uma universidade. Uma mesma pessoa, por exemplo João, pode ser ao mesmo tempo um estudante e um funcionário dessa universidade.

Na orientação a objetos, tipos de entidade são representados por classes, enquanto as instâncias de um tipo de entidade são objetos. Assim, uma atividade crucial da modelagem conceitual estrutural segundo o paradigma OO é a identificação de classes. Na UML, classes são representadas por um retângulo com três compartimentos: o compartimento superior é relativo ao nome da classe; o compartimento do meio é dedicado à especificação dos atributos da classe; e o compartimento inferior é dedicado à especificação das operações da classe, como mostra a Figura 6.4.

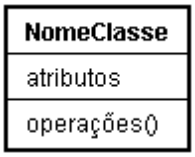

Figura 6.4 – Notação de Classes na UML.

Para nomear classes, sugere-se iniciar com um substantivo no singular, o qual pode ser combinado com complementos ou adjetivos, omitindo-se preposições. O nome da classe deve ser iniciado com letra maiúscula, bem como os nomes dos complementos, sem dar um espaço em relação à palavra anterior. Acentos não devem ser utilizados. Ex.: Cliente, PessoaFisica, ItemPedido.

Tomando por base os requisitos iniciais de cliente, relatórios de atividades de levantamento de requisitos (p.ex., atas de reunião, ou documentos em uma investigação de documentos) e, sobretudo, descrições de casos de uso, é possível iniciar a modelagem conceitual estrutural. Esse trabalho começa com a descoberta de quais classes devem ser incluídas no modelo, já que o cerne de um modelo OO é exatamente o seu conjunto de classes.

Durante a análise de requisitos, tipicamente o analista estuda, filtra e modela o domínio do problema. Dizemos que o analista "filtra" o domínio, pois apenas uma parte desse domínio fará parte das responsabilidades do sistema. Assim, um domínio de problemas pode incluir várias informações, mas as responsabilidades de um sistema nesse domínio podem incluir apenas uma pequena parcela desse conjunto.

As classes de um modelo representam a expressão inicial do sistema. As atividades subsequentes da modelagem estrutural buscam obter uma descrição cada vez mais detalhada, em termos de associações e atributos. Contudo, deve-se observar que, à medida que atributos e associações vão sendo identificados, se ganha maior entendimento a respeito do domínio e naturalmente novas classes podem surgir. Assim, as atividades da modelagem conceitual são iterativas e com alto grau de paralelismo, devendo ser realizadas concomitantemente.

Um aspecto fundamental no processo de modelagem conceitual é a interação constante com os especialistas de domínio. Técnicas de levantamento de requisitos, tais como entrevistas, análise de documentos e reuniões JAD, têm um papel fundamental nesta etapa. Assim, o levantamento de requisitos continua acontecendo paralelamente à modelagem conceitual e os relatórios produzidos nessas atividades são fontes importantíssimas de informação para a elaboração de modelos conceituais.

Uma maneira bastante prática e eficaz de trabalhar a identificação de classes consiste em olhar esses documentos à procura de classes. Diversos autores, dentre eles Jacobson (1992) e Wazlawick (2004), sugerem que uma boa estratégia para identificar classes consiste em ler esses documentos procurando por substantivos. Esses autores argumentam que uma classe é, tipicamente, descrita por um nome no domínio e, portanto, aprender sobre a terminologia do domínio do problema é um bom ponto de partida. Ainda que um bom ponto de partida, essa heurística é ainda muito vaga. Se o analista segui-la fielmente, muito provavelmente ele terá uma extensa lista de potenciais classes, sendo que muitas delas podem, na verdade, se referir a atributos de outras classes. Além disso, pode ser que importantes classes não sejam capturadas, notadamente aquelas que se referem ao registro de eventos de negócio, uma vez que esses eventos, muitas vezes, são descritos na forma de verbos. Seja o seguinte exemplo de uma descrição de um domínio de locação de automóveis: "clientes locam carros". Seriam consideradas potenciais classes: *Cliente* e *Carro*. Contudo, a locação é um evento de negócio importante que precisa ser registrado e, usando a estratégia de identificar classes a partir de substantivos, *Locação* não entraria na lista de potenciais classes.

Assim, neste texto sugere-se que, ao examinar documentos de requisitos de cliente, relatórios de atividades de levantamento de requisitos e descrições de casos de uso, os seguintes elementos sejam considerados como candidatos a classes:

- Agentes: entidades do domínio do problema que têm a capacidade de agir com intenção de atingir uma meta. Em sistemas de informação, há dois tipos principais de agentes: os agentes físicos (tipicamente pessoas) e os agentes sociais (organizações, unidades organizacionais, sociedades etc.). Em relação às pessoas, deve-se olhar para os papéis desempenhados pelas diferentes pessoas no domínio do problema.

- Objetos: entidades sem a capacidade de agir, mas que fazem parte do domínio de informação do problema. Podem ser também classificados em físicos (p.ex., carros, livros, imóveis) e sociais (p.ex., cursos, disciplinas, leis). Entretanto, há também outros tipos de objetos, tais como objetos de caráter descritivo usados para organizar e descrever outros objetos de um domínio (p.ex., modelos de carro). Objetos sociais e de descrição tendem a ser coisas menos tangíveis, mas são tão importantes para a modelagem conceitual quanto os objetos físicos.

- Eventos: representam a ocorrência de ações no domínio do problema que precisam ser registradas e lembradas pelo sistema. Eventos acontecem no tempo e, portanto, a representação de eventos normalmente envolve a necessidade de registrar, dentre outros, quando o evento ocorreu (ponto no tempo ou intervalo de tempo). Deve-se

observar que muitos eventos ocorrem no domínio do problema, mas grande parte deles não precisa ser lembrada. Para capturar os eventos que precisam ser lembrados e, portanto, registrados, devem-se focalizar os principais eventos de negócio do domínio do problema. Assim, em um sistema de locação de automóveis, são potenciais classes de eventos: *Locação*, *Devolução* e *Reserva*. Por outro lado, a ocorrência de eventos cadastrais, tais como os cadastros de clientes e carros, tende a ser de pouca importância, não sendo necessário lembrar a ocorrência desses eventos.

Seja qual for a estratégia usada para identificar classes, é sempre importante que o analista tenha em mente os objetivos do sistema durante a modelagem conceitual. Não se devem representar informações irrelevantes para o sistema e, portanto, a relevância para o sistema é o principal critério a ser adotado para decidir se um determinado elemento deve ou não ser incluído no modelo conceitual estrutural do sistema.

O resultado principal da atividade de identificação de classes é a obtenção de uma lista de potenciais classes para o sistema em estudo. Um modelo conceitual estrutural para uma aplicação complexa pode conter dezenas de classes e, portanto, pode ser necessário definir uma representação concisa capaz de orientar um leitor em um modelo dessa natureza. O agrupamento de classes em subsistemas serve basicamente a este propósito, podendo ser útil também para a organização de grupos de trabalho em projetos extensos. Conforme discutido no Capítulo 5, a base principal para a identificação de subsistemas é a complexidade do domínio do problema. Através da identificação e agrupamento de classes em subsistemas, é possível controlar a visibilidade do leitor e, assim, tornar o modelo mais compreensível. Assim, da mesma maneira que casos de uso são agrupados em pacotes, classes também devem ser.

Quando uma coleção de classes colabora entre si para realizar um conjunto coeso de responsabilidades (casos de uso), elas podem ser vistas como um subsistema. Assim, um subsistema é uma abstração que provê uma referência para mais detalhes em um modelo de análise, incluindo tanto casos de uso quanto classes. O agrupamento de classes em subsistemas permite apresentar o modelo global em uma perspectiva mais alta. Esse nível ajuda o leitor a rever o modelo, bem como constitui um bom critério para organizar a documentação.

Uma vez identificadas as potenciais classes, deve-se proceder uma avaliação para decidir o que efetivamente considerar ou rejeitar. Conforme discutido anteriormente, a relevância para o sistema deve ser o critério principal. Além desse critério, os seguintes critérios devem ser considerados nessa avaliação:

- Estrutura complexa: o sistema precisa tratar informações sobre os objetos da classe? Tipicamente, uma classe deve ter, pelo menos, dois atributos. Se uma classe apresentar apenas um atributo, avalie se não é melhor tratá-la como um atributo de uma classe existente[14].
- Atributos e associações comuns: os atributos e as associações da classe devem ser aplicáveis a todas as suas instâncias, isto é, a todos os objetos da classe.
- Existência de instâncias: toda classe deve possuir instâncias. Uma potencial classe que possua uma única instância não deve ser considerada em um modelo conceitual estrutural. Tipicamente uma classe possui várias instâncias e a população da classe varia ao longo do tempo.

[14] Uma classe que possui um único atributo, mas várias associações, também satisfaz a esse critério.

6.3 – Identificação de Atributos e Associações

Uma classe típica de um modelo conceitual estrutural apresenta estrutura complexa. A estrutura de uma classe corresponde a seus atributos e associações. Conceitualmente, não há diferença entre atributos e associações. Atributos são, na verdade, um tipo de relacionamento binário. Em um tipo de relacionamento binário, há dois participantes. Em alguns tipos de relacionamentos, esses participantes são considerados "colegas", porque eles desempenham funções análogas e nenhum deles é subordinado ao outro. Seja o caso do tipo de relacionamento "*aluno cursa um curso*". Um aluno só é aluno se cursar um curso. Por outro lado, não faz sentido existir um curso se ele não puder ser cursado por alunos. A ordem dos participantes no modelo não implica uma relação de prioridade ou subordinação entre eles (OLIVÉ, 2007). Na orientação a objetos, esse tipo de relacionamento é modelado como uma associação.

Entretanto, há alguns tipos de relacionamentos nos quais usuários e analistas consideram um participante como sendo uma característica do outro. Seja o exemplo do tipo de relacionamento "*filme possui gênero*". *Gênero* é uma característica de *filme* e, portanto, subordinado a este. Esse tipo de relacionamento é modelado como um atributo. Assim, um atributo é um tipo de relacionamento binário em que um participante é considerado uma característica de outro. Por conseguinte, um atributo é igual a uma associação, exceto pelo fato de usuários e analistas adicionarem a interpretação que um dos participantes é subordinado ao outro (OLIVÉ, 2007).

De uma perspectiva mais prática, atributos ligam classes do domínio do problema a tipos de dados. Associações, por sua vez, consistem em um tipo de informação que liga diferentes classes do domínio entre si.

Tipos de dados podem ser primitivos ou específicos de domínio. Os tipos de dados primitivos são aplicáveis aos vários domínios e sistemas, tais como strings, datas, inteiros e reais, e são considerados como sendo predefinidos. Os tipos de dados específicos de um domínio de aplicação, por outro lado, precisam ser definidos. São exemplos de tipos de dados específicos: CPF, ISBN de livros, endereço etc.

Neste texto são considerados os seguintes tipos de dados primitivos:

- *String*: cadeia de caracteres;
- *boolean*: admite apenas os valores verdadeiro e falso;
- *Integer* (ou *int*): números inteiros;
- *Float* (ou *float*): números reais;
- *Currency*: valor em moeda (reais, dólares etc.);
- *Date*: datas, com informação de dia, mês e ano;
- *Time*: horas em um dia, com informação de hora, minuto e segundo;
- *DateTime*: combinação dos dois anteriores;
- *YearMonth*: informação de tempo contendo apenas mês e ano;
- *Year*: informação de tempo contendo apenas ano.

6.3.1 – Atributos

Um atributo é uma informação de estado para a qual cada objeto em uma classe tem o seu próprio valor. Os atributos adicionam detalhes às abstrações e são apresentados na parte central do símbolo de classe. Atributos possuem um tipo de dado, que pode ser primitivo ou específico de domínio. Ao identificar um atributo como sendo relevante, deve-se definir qual o seu tipo de dado. Caso nenhum dos tipos de dados primitivos se aplique, deve-se definir, então, um tipo de dados específico. Por exemplo, em domínios que lidem com livros, é necessário definir o tipo ISBN[15], cujas instâncias são ISBNs válidos. Em domínios que lidem com pessoas físicas e jurídicas, CPF e CNPJ também devem ser definidos como tipos de dados específicos. Usar um tipo de dados primitivo nestes casos, tal como *String* ou *int*, é insuficiente, pois não são quaisquer cadeias de caracteres ou números que se caracterizam como ISBNs, CPFs ou CNPJs válidos.

Tipos de dados específicos podem apresentar propriedades. Por exemplo, CPF é um número de 11 dígitos, que pode ser dividido em duas partes: os 9 primeiros dígitos e os dois últimos, que são dígitos verificadores. Um tipo de dados especial é a enumeração. Na enumeração, os valores do tipo são enumerados explicitamente na forma de literais, como é o caso do tipo *DiaSemana*, que é tipicamente definido como um tipo de dados compreendendo sete valores: {Segunda, Terça, Quarta, Quinta, Sexta, Sábado e Domingo}. É importante observar que tipos de dados enumerados só devem ser usados quando se sabe a priori quais são os seus valores e eles são fixos. Assim, são bons candidatos a tipos enumerados informações como sexo (M/F), estado civil, etc.

Tipos de dados geralmente não são representados graficamente em um modelo conceitual estrutural, de modo a torná-lo mais simples. Na maioria das situações, basta descrever os tipos de dados específicos de domínio no Dicionário de Dados do Projeto. Contudo, se necessário, eles podem ser representados graficamente usando o símbolo de classe estereotipado com a palavra chave <<*dataType*>>. Tipos enumerados também podem ser representados usando o símbolo de classe, mas com o estereótipo <<*enumeration*>> ou <<*enum*>>, sendo que ao invés de apresentar atributos de um tipo de dados, enumeram-se os valores possíveis da enumeração. A Figura 6.5 ilustra a notação de tipos de dados na UML.

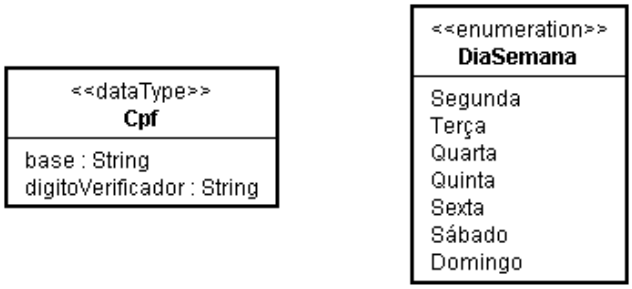

Figura 6.5 – Notação de Tipos de Dados na UML.

Uma dúvida típica e recorrente na modelagem estrutural é se um determinado item de informação deve ser modelado como uma classe ou como um atributo. Para que o item seja considerado uma classe, ele tem de passar nos critérios de inclusão no modelo discutidos na seção anterior. Entretanto, há alguns itens de informação que passam nesses critérios, mas que

[15] O ISBN - *International Standard Book Number* - é um sistema internacional padronizado que identifica numericamente os livros segundo o título, o autor, o país, a editora, individualizando-os inclusive por edição. Utilizado também para identificar software, seu sistema numérico pode ser convertido em código de barras.

ainda assim podem ser melhor modelados como atributos, tendo como tipo um tipo de dado complexo, específico de domínio. Um atributo deve capturar um conceito atômico, i.e., um único valor ou um agrupamento de valores fortemente relacionados que sirva para descrever outro objeto. Além disso, para que um item de estrutura complexa seja modelado como um atributo, ele deve ser compreensível pelos interessados simplesmente pelo seu nome.

É muito importante lembrar também que, uma vez que atributos e associações são tipos de relacionamentos, não devemos incluir na lista de atributos de uma classe, atributos representando associações (ou atributos representando "chaves estrangeiras" como se a classe fosse uma tabela de um banco de dados relacional). Associações já têm sua presença indicada pela notação de associação, ou seja, pelas linhas que conectam as classes que se relacionam.

Um aspecto bastante importante na especificação de atributos é a escolha de nomes. Deve-se procurar utilizar o vocabulário típico do domínio do problema, usando nomes significativos. Para nomear atributos, sugerem-se nomes iniciando com substantivo, o qual pode ser combinado com complementos ou adjetivos, omitindo-se preposições. O nome do atributo deve ser iniciado com letra minúscula, enquanto os nomes dos complementos devem iniciar com letras maiúsculas, sem dar um espaço em relação à palavra anterior. Acentos não devem ser utilizados. Atributos monovalorados devem iniciar com substantivo no singular (p.ex., nome, razaoSocial), enquanto atributos multivalorados devem iniciar com o substantivo no plural (p.ex., telefones).

A sintaxe de atributos na UML, em sua forma plena, é a seguinte (BOOCH; RUMBAUGH; JACOBSON, 2006):

visibilidade nome: tipo [multiplicidade] = valorInicial {propriedades}

A visibilidade de um atributo indica em que situações esse atributo é visível por outras classes. Na UML há quatro níveis de visibilidade, indicados pelos seguintes símbolos:

+ público : o atributo pode ser acessado por qualquer classe;

protegido: o atributo só é passível de acesso pela própria classe ou por uma de suas especializações;

- privado: o atributo só pode ser acessado pela própria classe;

~ pacote: o atributo só pode ser acessado por classes declaradas dentro do mesmo pacote da classe a que pertence o atributo.

A informação de visibilidade é inerente à fase de projeto e não deve ser expressa em um modelo conceitual. Assim, em um modelo conceitual, atributos devem ser especificados sem nenhum símbolo antecedendo o nome.

O *tipo* indica o tipo de dado do atributo, o qual deve ser um tipo de dado primitivo ou um tipo de dado específico de domínio. Tipos de dados específicos de domínio devem ser definidos no Dicionário de Dados do Projeto.

A multiplicidade é a especificação do intervalo permitido de itens que o atributo pode abrigar. O padrão é que cada atributo tenha um e somente um valor para o atributo. Quando um atributo for opcional ou quando puder ter mais do que uma ocorrência, a multiplicidade deve ser informada, indicando o valor mínimo e o valor máximo, da seguinte forma:

valor_mínimo .. valor_máximo

A seguir, são dados alguns exemplos:

- nome: String → instâncias da classe têm obrigatoriamente um e somente um nome.
- cpf: Cpf [0..1] → instâncias da classe têm um ou nenhum cpf.
- telefones: Telefone [0..*] → instâncias da classe têm um ou vários telefones.
- pessoasContato: String [2] → instâncias da classe têm exatamente duas pessoas de contato.

Um atributo pode ter um valor padrão inicial, ou seja, um valor que, quando não informado outro valor, será atribuído ao atributo. O campo *valorInicial* descreve exatamente este valor. O exemplo abaixo ilustra o uso de valor inicial.

- origem: Ponto = (0,0) → a origem, quando não informado outro valor, será o ponto (0,0)

Finalmente, podem ser indicadas propriedades dos atributos. Uma propriedade que pode ser interessante mostrar em um modelo conceitual é a propriedade *readonly*, a qual indica que o valor do atributo não pode ser alterado após a inicialização do objeto. No exemplo abaixo, está-se indicando que o valor do atributo *matricula* de um aluno não pode ser alterado.

- matricula: String {readonly}

Além das informações tratadas na declaração de um atributo seguindo a sintaxe da UML, outras informações de domínio, quando pertinentes, podem ser adicionadas no Dicionário de Dados do Projeto, tais como unidade de medida, intervalo de valores possíveis, limite, precisão etc.

6.3.2 – Associações

Uma associação é um tipo de relacionamento que ocorre entre instâncias de duas ou mais classes. Assim como classes, associações são tipos. Ou seja, uma associação modela um tipo de relacionamento que pode ocorrer entre instâncias das classes envolvidas. Uma instância de uma associação (dita uma ligação) conecta instâncias específicas das classes envolvidas na associação. Seja o exemplo de um domínio em que clientes efetuam pedidos. Esse tipo de relacionamento pode ser modelado como uma associação *Cliente efetua Pedido*. Seja Pedro uma instância de *Cliente* e Pedido100 uma instância de *Pedido*. Se foi Pedro quem efetuou o Pedido100, então a ligação (Pedro, Pedido100) é uma instância da associação *Cliente efetua Pedido*.

Associações podem ser nomeadas. Neste texto sugere-se o uso de verbos conjugados, indicando o sentido de leitura. Ex.: Cliente (classe) *efetua* ▸ (associação) Locação (classe). Cada classe envolvida na associação desempenha um papel, ao qual pode ser dado um nome. Cada classe envolvida na associação possui também uma multiplicidade[16] nessa associação, que indica quantos objetos podem participar de uma instância dessa associação. A notação da UML usada para representar associações em um modelo conceitual é ilustrada na Figura 6.6.

[16] Multiplicidades em uma associação são análogas às multiplicidades em atributos e especificam as quantidades mínima e máxima de objetos que podem participar da associação. Quando nada for dito, o padrão é 1..1 como no caso de atributos. Contudo, para deixar os modelos claros, recomenda-se sempre especificar explicitamente as multiplicidades das associações.

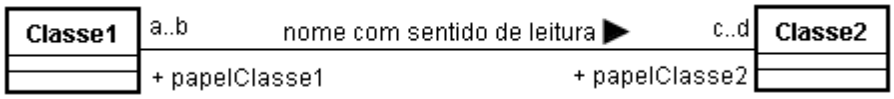

Figura 6.6 – Notação de Associações na UML.

Na ilustração da figura, um objeto da *Classe1* se relaciona com no mínimo *c* e no máximo *d* objetos da *Classe2*. Já um objeto da *Classe2* se relaciona com no mínimo *a* e no máximo *b* objetos da *Classe1*. Objetos da *Classe1* desempenham o papel de "*papelClasse1*" nesta associação, enquanto objetos da *Classe2* desempenham o papel de "*papelClasse2*" nessa mesma associação.

É importante, neste ponto, frisar a diferença entre sentido de leitura (ou direção do nome) de uma associação com a navegação da associação. O sentido de leitura diz apenas em que direção ler o nome da associação, mas nada diz sobre a navegabilidade da associação. A navegabilidade (linha de associação com seta direcionada) é usada para limitar a navegação de uma associação a uma única direção e é um recurso a ser usado apenas na fase de projeto. Em um modelo conceitual, todas as associações navegáveis nos dois sentidos e, portanto, não direcionais.

Ainda que nomes de associações e papéis sejam opcionais, recomenda-se usá-los para tornar o modelo mais claro. Além disso, há algumas situações em que fica inviável ler um modelo se não houver a especificação do nome da associação ou de algum de seus papéis.

Seja o exemplo da Figura 6.7. Em uma empresa, um empregado está lotado em um departamento e, opcionalmente, pode chefiá-lo. Um departamento, por sua vez, pode ter vários empregados nele lotados, mas apenas um chefe. Sem nomear essas associações, o modelo fica confuso. Rotulando os papéis e as associações, o modelo torna-se muito mais claro. Na Figura 6.7, um departamento exerce o papel de *departamento de lotação* do empregado e, neste caso, um empregado tem um e somente um departamento de lotação. No outro relacionamento, um empregado exerce o papel de *chefe* e, portanto, um departamento possui um e somente um chefe.

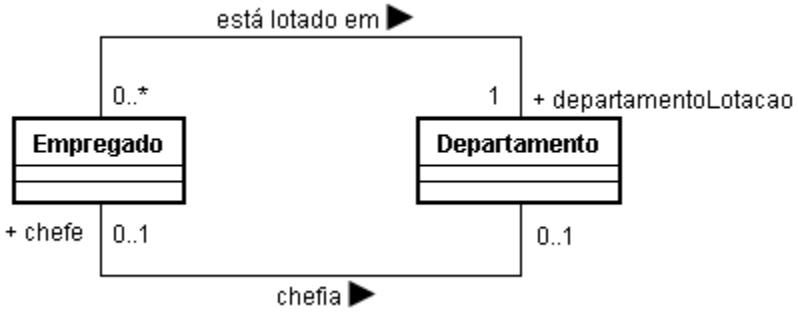

Figura 6.7 – Exemplo: Nomeando Associações.

Ao contrário das classes e dos atributos que podem ser encontrados facilmente a partir da leitura dos textos da descrição do minimundo e das descrições de casos de uso, muitas vezes, as informações sobre associações não aparecem tão explicitamente. Casos de uso descrevem ações de interação entre atores e sistema e, por isso, acabam mencionando principalmente operações. Operações transformam a informação, passando um objeto de um estado para outro, por meio da alteração dos seus valores de atributos e associações. Uma associação, por sua vez, é uma relação estática que pode existir entre duas classes. Assim, as descrições de casos de uso estão repletas de operações, mas não de associações (WAZLAWICK, 2004).

Contudo, conforme discutido na seção anterior, há alguns eventos que precisam ter sua ocorrência registrada e, portanto, são tipicamente mapeados como classes. Esses eventos estão descritos nos casos de uso e podem ter sido capturados como associações. Seja o exemplo de uma concessionária de automóveis. Neste domínio, clientes compram carros, como ilustra a parte (a) da Figura 6.8. Contudo, a compra é um evento importante para o negócio e precisa ser registrada. Neste caso, como ilustra a parte (b) da Figura 6.8, a compra deve ser tratada como uma classe e não como uma associação.

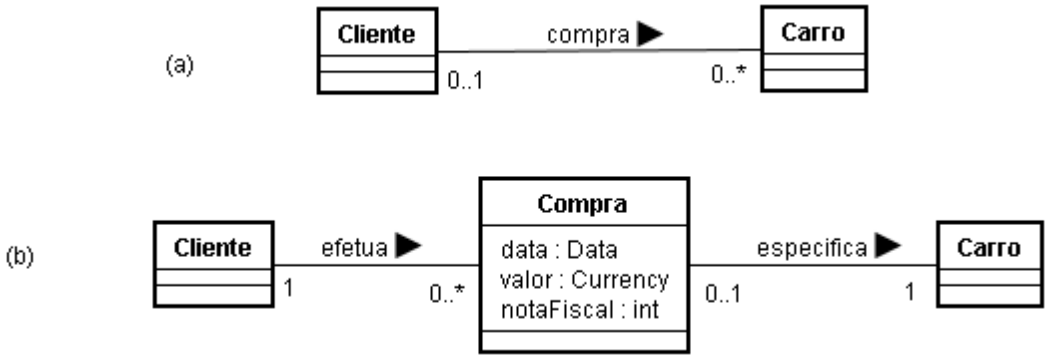

Figura 6.8 – Exemplo: Associação x Classe de Evento Lembrado.

Deve-se notar pelo exemplo acima que o evento é representado por uma classe, enquanto as associações continuam representando relacionamentos estáticos entre as classes e não operações ou transformações (WAZLAWICK, 2004). Assim, deve-se tomar cuidado com a representação de eventos como associações, questionando sempre se aquela associação é relevante para o sistema em questão.

Seja o exemplo da Figura 6.9. Nesse exemplo, o caso de uso aponta que funcionários são responsáveis por cadastrar livros em uma biblioteca. Seria necessário, pois, criar uma associação *Funcionário cadastra Livro* no modelo estrutural? A resposta, na maioria dos casos, é não. Apenas se explicitamente expresso pelo cliente em um requisito que é necessário saber exatamente qual funcionário fez o cadastro de um dado livro (o que é muito improvável de acontecer), é que tal relação deveria ser considerada. Mesmo se houver a necessidade de auditoria de uso do sistema (requisito não funcional relativo a segurança), não há a necessidade de modelar esta associação, pois requisitos não funcionais não devem ser considerados no modelo conceitual, uma vez que soluções bastante distintas à do uso dessa associação poderiam ser adotadas.

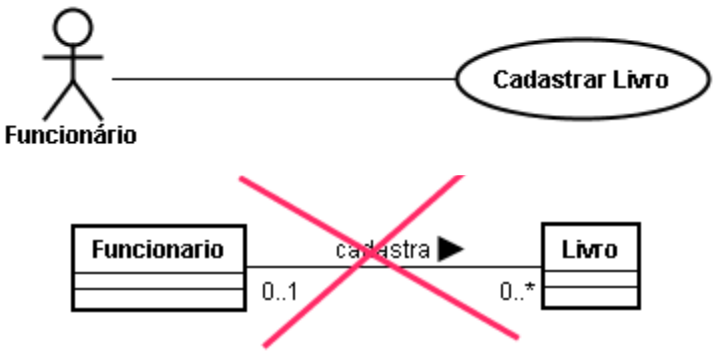

Figura 6.9 – Exemplo: Associação x Caso de Uso.

Na modelagem conceitual é fundamental saber a quantidade de objetos que uma associação admite em cada um de seus papéis, o que é capturado pelas multiplicidades da associação. Esta informação é bastante dependente da natureza do problema e do real significado da associação (o que se quer representar efetivamente), especialmente no que se refere à associação representar apenas o presente ou o histórico (WAZLAWICK, 2004).

Retomemos o exemplo da Figura 6.7, no qual se diz que um empregado está lotado em um departamento e, opcionalmente, pode chefiá-lo. Para definir precisamente as multiplicidades, é necessário investigar os seguintes aspectos: Um empregado pode mudar de lotação? Se sim, é necessário registrar apenas a lotação atual ou é necessário registrar o histórico de lotações dos empregados (ou seja, registrar o evento de lotação de um empregado em um departamento)? Um departamento pode, ao longo do tempo, mudar de chefe? Se sim, é necessário registrar o histórico de chefias do departamento (ou seja, registrar o evento de nomeação do chefe do departamento)?

O modelo da Figura 6.7 representa apenas a situação presente. Se houver mudança de chefe de um departamento ou do departamento de lotação de um empregado, perder-se-á a informação histórica. Na maioria das vezes, essa não é uma solução aceitável. Na maioria dos domínios, as pessoas querem saber a informação histórica. Assim, nota-se que é parte das responsabilidades do sistema registrar a ocorrência dos eventos de nomeação do chefe e de lotação de empregados. Assim, um modelo mais fidedigno a essa realidade é o modelo da Figura 6.10, o qual introduz as classes do tipo "evento lembrado" *NomeacaoChefia* e *Lotacao*.

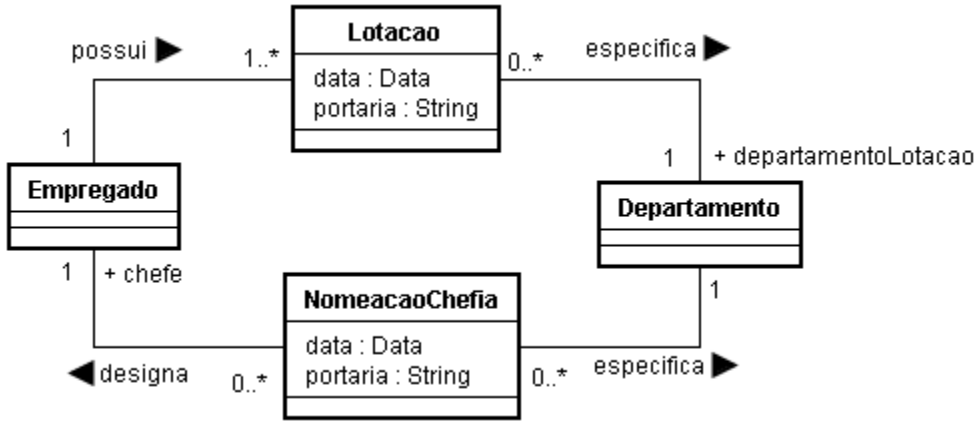

Figura 6.10– Registrando Históricos.

Ainda que este modelo seja mais fidedigno à realidade, ele ainda apresenta problemas. Por exemplo, o modelo diz que um empregado pode ter uma ou mais lotações. Mas o empregado pode ter mais de uma lotação vigente? O mesmo vale para o caso da nomeação de chefia. Um empregado pode ser chefe de mais de um departamento ao mesmo tempo? Um departamento pode ter mais do que um chefe nomeado ao mesmo tempo? Infelizmente, o modelo é incapaz de responder a essas perguntas. Para eliminar essas ambiguidades, é necessário capturar regras de negócio do tipo *restrições de integridade*. No exemplo acima, as seguintes regras se aplicam:

- Um empregado só pode estar lotado em um único departamento em um dado momento.
- Um empregado só pode estar designado como chefe de um único departamento em um dado momento.

- Um departamento só pode ter um empregado designado como chefe em um dado momento.

Observe que, como um departamento pode ter vários empregados nele lotados ao mesmo tempo, não é necessário escrever uma restrição de integridade, pois este é o caso mais geral (sem restrição). Assim, restrições de integridade devem ser escritas apenas para as associações que efetivamente envolvem restrições.

Restrições de integridade são regras de negócio e poderiam ser lançadas no Documento de Definição de Requisitos. Contudo, como elas são importantes para a compreensão e eliminação de ambiguidades do modelo conceitual, é melhor descrevê-las no próprio modelo conceitual.

Além das restrições de integridade relativas às multiplicidades *n*, diversas outras restrições podem ser importantes para tornar o modelo mais fiel à realidade. Ainda no exemplo da Figura 6.10, poder-se-ia querer dizer que um empregado só pode ser nomeado como chefe de um departamento, se ele estiver lotado nesse departamento. Restrições deste tipo são comuns em porções fechadas de um diagrama de classes. Assim, toda vez que se detectar uma porção fechada em um diagrama de classes, vale a pena analisá-la para avaliar se há ali uma restrição de integridade ou não. Havendo, deve-se escrevê-la.

Restrições de integridade também podem falar sobre atributos das classes. Por exemplo, a data da portaria de nomeação de um empregado *e* como chefe de um departamento *d* deve ser igual ou posterior à data da portaria de lotação do empregado *e* no departamento *d*.

Geralmente, restrições de integridade são escritas em linguagem natural, uma vez que não são passíveis de modelagem gráfica. Contudo, conforme já discutido anteriormente, o uso da linguagem natural pode levar a ambiguidades. Visando suprir essa lacuna na UML, o OMG[17] incorporou ao padrão uma linguagem para especificação formal de restrições, a OCL (*Object Constraint Language*). Contudo, restrições escritas em OCL dificilmente serão entendidas por clientes e usuários, o que dificulta a validação das mesmas. Assim, neste texto, sugere-se escrever as restrições de integridade em linguagem natural mesmo.

Vale ressaltar que a UML provê alguns mecanismos para representar restrições de integridade em um modelo gráfico. As próprias multiplicidades são uma forma de capturar restrições de integridade (ditas restrições de integridade de cardinalidade). Além das multiplicidades, a UML provê o recurso de restrições, as quais são representadas entre chaves (**{restrição}**). Restrições podem ser usadas, dentre outros, para restringir a ocorrência de associações. Seja o seguinte exemplo: em uma concessionária de automóveis compras podem ser financiadas ou por financeiras ou por bancos. Para capturar essa restrição, pode-se usar a restrição *xor* da UML, como ilustra a Figura 6.11.

[17] *Object Management Group* (http://www.omg.org/) é uma organização internacional que gerencia padrões abertos relativos ao desenvolvimento orientado a objetos, dentre eles a UML.

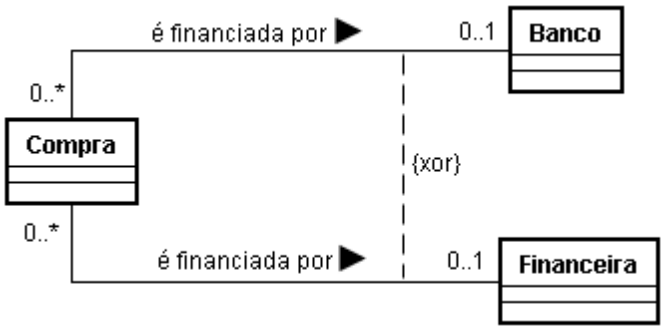

Figura 6.11 – Restrição XOR entre Associações.

Nesta figura, uma compra ou está relacionada a um banco ou a uma financeira. Não é possível que uma compra esteja associada aos dois ao mesmo tempo. Como as multiplicidades mínimas do lado de banco e financeira são zero, uma compra pode não ser financiada.

Ainda em relação às multiplicidades, vale frisar que associações muitos-para-muitos são perfeitamente legais em um modelo orientado a objetos, como ilustra o exemplo da Figura 6.12. Nesse exemplo, está-se dizendo que disciplinas podem possuir vários pré-requisitos e podem ser pré-requisitos para várias outras disciplinas.

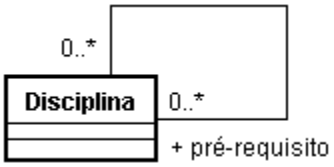

Figura 6.12– Associação Muitos-para-Muitos.

Deve-se observar, no entanto, que muitas vezes, uma associação muitos-para-muitos oculta a necessidade de uma classe do tipo evento a ser lembrado. Seja o seguinte exemplo: em uma organização, empregados são alocados a projetos. Um empregado pode ser alocado a vários projetos, enquanto um projeto pode ter vários empregados a ele alocados. Tomando por base este fato, seria natural se chegar ao modelo da Figura 6.13(a). Contudo, se quisermos registrar as datas de início e fim do período em que o empregado esteve alocado ao projeto, esse modelo é insuficiente e deve ser alterado para comportar uma classe do tipo evento lembrado *Alocacao*, como mostra a Figura 6.13(b).

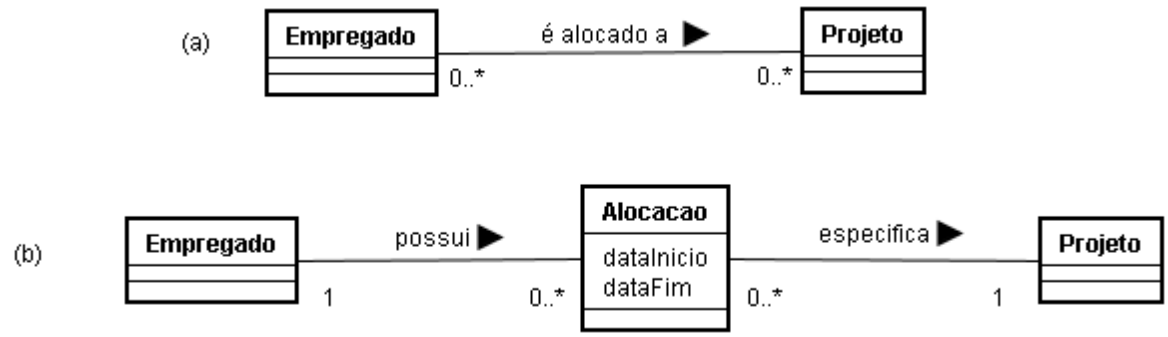

Figura 6.13– Associação Muitos-para-Muitos e Classes de Evento Lembrado.

De fato, o problema por detrás do modelo da Figura 6.13(a) é o mesmo anteriormente discutido na Figura 6.10: a necessidade ou não de se representar informação histórica. Contudo, de maneira mais abrangente, pode-se pensar que, se uma associação apresenta atributos, é melhor tratá-la como uma nova classe. Seja o seguinte exemplo: em uma loja, um cliente efetua um pedido, discriminando vários produtos, cada um deles em uma certa quantidade. O modelo da Figura 6.14(a) procura representar essa situação, mas uma questão permanece em aberto: onde representar a informação da quantidade pedida de cada produto? Essa informação não pode ficar em *Produto*, pois diferentes pedidos pedem quantidades diferentes de um mesmo produto. Também não pode ficar em *Pedido*, pois um mesmo pedido tipicamente especifica diferentes quantidades de diferentes produtos. De fato, quantidade não é nem um atributo da classe *Pedido* nem um atributo da classe *Produto*, mas sim um atributo da associação *especifica*. Assim, a solução consiste em introduzir a classe *ItemPedido*, reificando[18] essa associação, como ilustra a Figura 6.14(b).

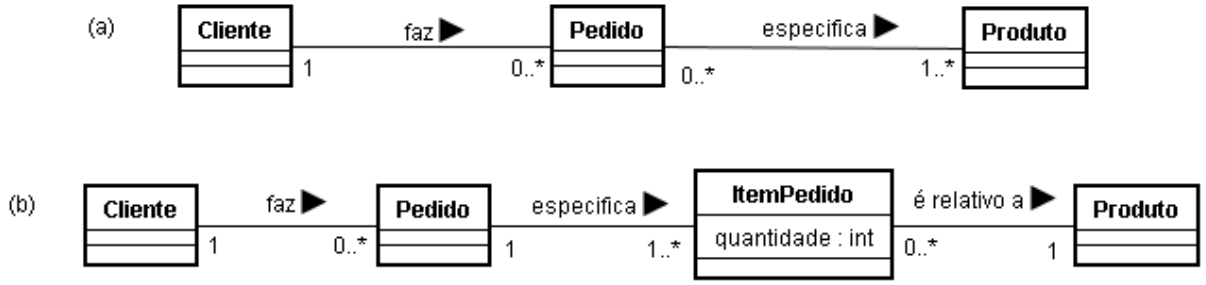

Figura 6.14 – Reificando uma Associação

A UML oferece uma primitiva de modelagem, chamada *classe de associação*, que pode ser usada para reificar associações (OLIVÉ, 2007). Uma classe de associação pode ser vista como uma associação que tem propriedades de classe (BOOCH; RUMBAUGH; JACOBSON, 2006). A Figura 6.15 mostra o exemplo anterior, sendo modelado como uma classe de associação, segundo a notação da UML.

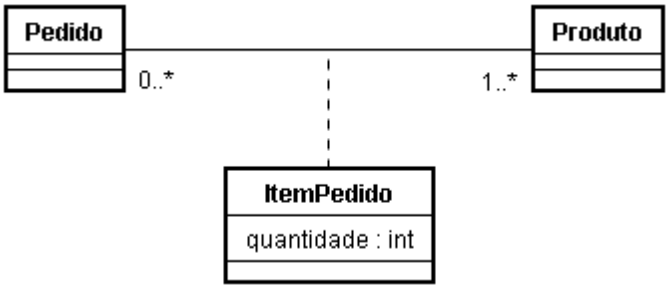

Figura 6.15 – Notação da UML para Classes Associativas.

Classes associativas são ainda representações de associações. Assim como uma instância de uma associação, uma instância de uma classe associativa é um par ordenado conectando duas instâncias das classes envolvidas na associação. Assim, se Pedido100 é uma instância de *Pedido*, Lápis é uma instância de *Produto* e o Pedido100 especifica 5 Lápis, então uma instância de *ItemPedido* é a tupla ((Pedido100, Lápis), 5).

[18] Reificar uma associação consiste em ver essa associação como uma classe. A palavra "reificação" vem da palavra do latim *res*, que significa coisa. Reificação corresponde ao que em linguagem natural se chama nominalização, que basicamente consiste em transformar um verbo em um substantivo (OLIVÉ, 2007).

Classes associativas podem ser usadas também para representar eventos cuja ocorrência precisa ser lembrada, como nos exemplos das figuras 6.10 e 6.13. Entretanto, é importante observar que o uso de classes associativas nesses casos pode levar a problemas de modelagem. Seja o seguinte contexto: em um hospital, pacientes são tratados em unidades médicas. Um paciente pode ser tratado em diversas unidades médicas diferentes, as quais podem abrigar diversos pacientes sendo tratados. A Figura 6.16(a) mostra um modelo que busca representar essa situação usando uma classe associativa. Como uma classe associativa, as instâncias de Tratamento são pares ordenados (Paciente, Unidade Médica). Assim, cada vez que um paciente é tratado em uma unidade médica diferente, tem-se um tratamento. Esta pode, contudo, não ser precisamente a concepção do problema original. Poder-se-ia imaginar que um tratamento é um tratamento de um paciente em várias unidades médicas. A classe de associação não permite representar isso. Assim, um modelo mais fiel ao domínio é aquele que representa Tratamento como uma classe do tipo evento a ser lembrado e que está relacionada com Paciente e Unidade Médica da forma mostrada na Figura 6.16(b).

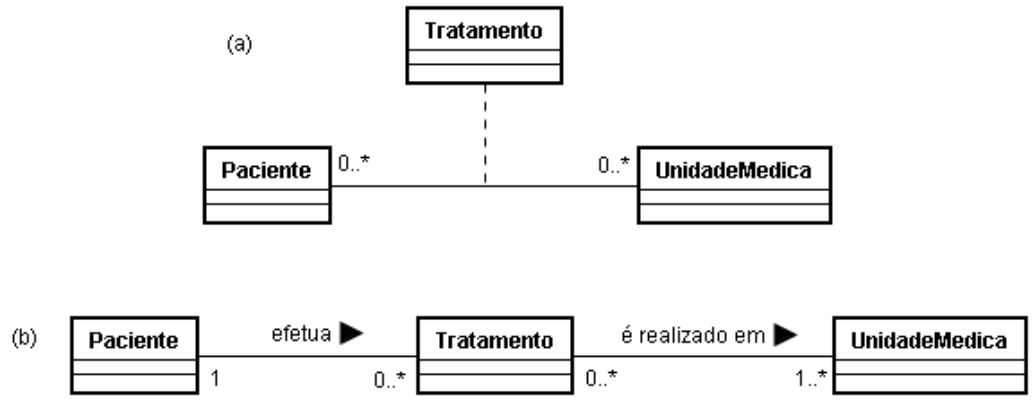

Figura 6.16 – Classes Associativas x Classes do Tipo Evento a Ser Lembrado.

Até o momento, todas as associações mostradas foram associações binárias, i.e., associações envolvendo duas classes. Mesmo o exemplo da Figura 6.12 (Disciplina tem como pré-requisito Disciplina) é ainda uma associação binária, na qual a mesma classe desempenha dois papéis diferentes (disciplina que possui pré-requisito e disciplina que é pré-requisito). Entretanto, associações *n*-árias são também possíveis, ainda que bem menos corriqueiramente encontradas. Uma associação ternária, por exemplo, envolve três classes, como ilustra o exemplo da Figura 6.17. Nesse exemplo, está-se dizendo que fornecedores podem fornecer produtos para certos clientes.

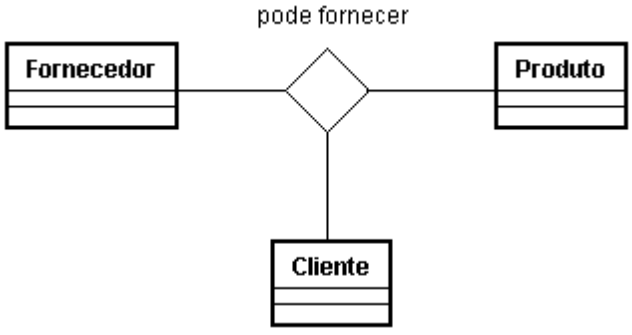

Figura 6.17 – Associação Ternária.

Na UML, associações *n*-árias são mostradas como losangos conectados às classes envolvidas na associação por meio de linhas sólidas, como mostra a Figura 6.17. O nome da

associação é colocado dentro ou em cima do losango, sem direção de leitura. Normalmente, multiplicidades não são mostradas, dada a dificuldade de interpretá-las.

Por fim, uma associação comum entre duas classes representa um relacionamento estrutural entre pares, significando que essas duas classes estão em um mesmo nível, sem que uma seja mais importante do que a outra. Algumas associações, contudo, são consideradas mais fortes, pois indicam que um objeto é composto de outros (relação todo-parte). Para esses casos, a UML considera um tipo especial de associação entre objetos: *agregação*. A agregação é uma forma especial de associação binária que especifica um relacionamento *todo-parte* entre um objeto agregado (o todo) e seus componentes (as partes). Uma agregação pode ser compartilhada ou composta. Na agregação compartilhada, a parte pode ser compartilhada por vários todos. P.ex., uma Comissão (todo) tem vários Membros (parte) e um Membro pode ser parte de mais de uma Comissão. A agregação composta (ou *composição*) é uma forma mais forte de agregação que requer que o objeto parte seja incluído em no máximo um objeto composto por vez. P.ex., um Motor só pode ser parte de um e somente um Carro por vez. Na composição, a parte tem responsabilidade na existência do todo. Se o objeto todo é excluído, todas as suas partes são também excluídas. Note, contudo, que um objeto parte pode ser removido do objeto todo antes que este último seja excluído e, portanto, o objeto parte, nesse caso, não seria excluído junto com o objeto todo (OMG, 2015).

A Figura 6.18 ilustra os casos exemplificados acima. Na Figura 6.18(a), um Carro tem como partes um Motor e quatro ou cinco Rodas. Motor e rodas, ao serem partes de um carro, não podem ser partes de outros carros simultaneamente. Assim, esta é uma relação de composição, a qual impõe que um objeto-parte só pode ser parte de um único todo. Já a agregação não implica nessa exclusividade. O exemplo da Figura 6.15(b) ilustra o caso de comissões compostas por professores. Nesse caso, um professor pode participar de mais de uma comissão simultaneamente e, portanto, trata-se uma relação de agregação. Na UML, um losango branco na extremidade da associação relativa ao todo indica uma agregação. Já um losango preto indica uma composição.

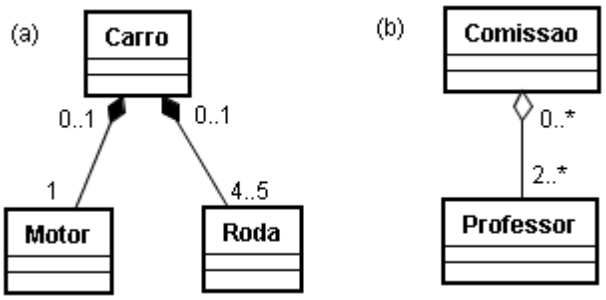

Figura 6.18 – Composição e Agregação.

Relações todo-parte podem ser empregadas em situações como:

- Quando há clareza de que um objeto complexo é composto de outros objetos (componente de). Ex.: Motor é um componente de um carro.
- Para designar membros de coleções (membro de). Ex.: Pesquisadores são membros de Grupos de Pesquisa.

Muitas vezes pode ser difícil perceber a diferença entre uma agregação e uma associação comum. Quando houver essa dúvida, é melhor representar a situação usando uma associação comum, tendo em vista que ela impõe menos restrições.

6.3 – Especificação de Hierarquias de Generalização / Especialização

Um dos principais mecanismos de estruturação de conceitos é a generalização / especialização. Com este mecanismo é possível capturar similaridades entre classes, dispondo-as em hierarquias de classes. No contexto da orientação a objetos, esse tipo de relacionamento é também conhecido como herança.

É importante notar que a herança tem uma natureza bastante diferente das associações. Associações representam possíveis ligações entre instâncias das classes envolvidas. Já a relação de herança é uma relação entre classes e não entre instâncias. Ao se considerar uma classe *B* como sendo uma subclasse de uma classe *A* está-se assumindo que todas as instâncias de *B* são também instâncias de *A*. Assim, ao se dizer que a classe *EstudanteGraduacao* herda da classe *Estudante*, está-se indicando que todos os estudantes de graduação são estudantes. Em resumo, deve-se interpretar a relação de herança como uma relação de subtipo entre classes. A Figura 6.19 mostra a notação da UML para representar herança.

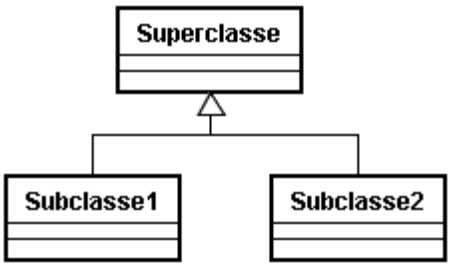

Figura 6.19 – Notação de Herança da UML.

A relação de herança é aplicável quando for necessário fatorar os elementos de informações (atributos e associações) de uma classe. Quando um conjunto de classes possuir semelhanças e diferenças, então elas podem ser organizadas em uma hierarquia de classes, de forma a agrupar em uma superclasse os elementos de informação comuns, deixando as especificidades nas subclasses.

De maneira geral, é desnecessário criar hierarquias de classes quando as especializações (subclasses) não tiverem nenhum elemento de informação diferente. Quando isso ocorrer, é normalmente suficiente criar um atributo *tipo* para indicar os possíveis subtipos da generalização. Seja o caso de um domínio em que se faz distinção entre clientes normais e clientes especiais, dos quais se quer saber exatamente as mesmas informações. Neste caso, criar uma hierarquia de classes, como ilustra a Figura 6.20(a), é desnecessário. Uma solução como a apresentada na Figura 6.20(b), em que o atributo *tipo* pode ser de um tipo enumerado com os seguintes valores {Normal, Especial}, modela satisfatoriamente o problema e é mais simples e, portanto, mais indicada.

Também não faz sentido criar uma hierarquia de classes em que a superclasse não tem nenhum atributo ou associação. Informações de estados pelos quais um objeto passa também não devem ser confundidas com subclasses. Por exemplo, um carro de uma locadora de automóveis pode estar locado, disponível ou em manutenção. Estes são estados e não subtipos de carro.

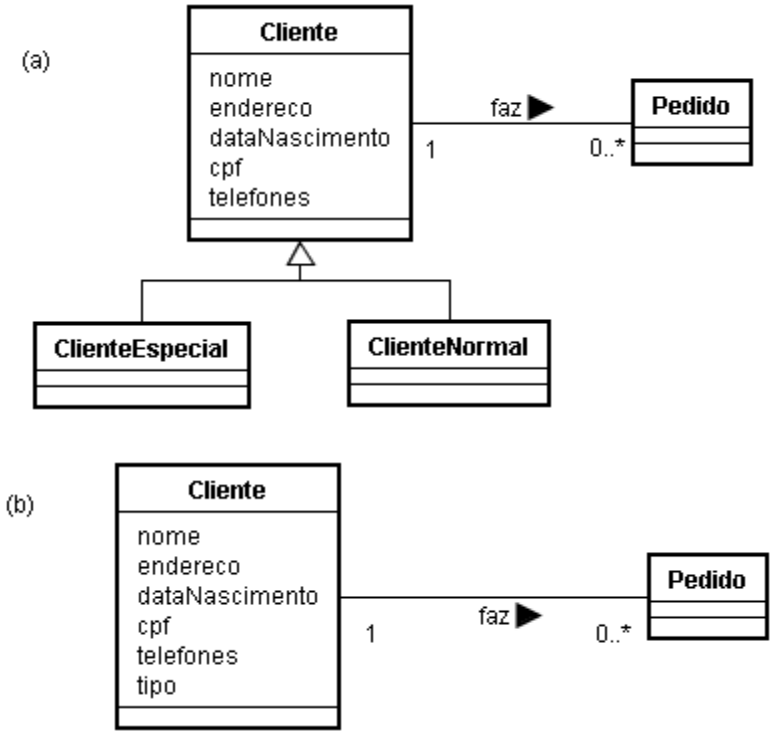

Figura 6.20 – Uso ou não de Herança.

De fato, é interessante considerar alguns critérios para incluir uma subclasse (ou superclasse) em um modelo conceitual. O principal deles é o fato da especialização (ou generalização) estar dentro do domínio de responsabilidade do sistema. Apenas subclasses (superclasses) relevantes para o sistema em questão devem ser consideradas. Além desse critério básico, os seguintes critérios devem ser usados para analisar hierarquias de herança:

- Uma hierarquia de classes deve modelar relações "é-um-tipo-de", ou seja, toda subclasse deve ser um subtipo específico de sua superclasse.
- Uma subclasse deve possuir todas as propriedades (atributos e associações) definidas por suas superclasses e adicionar mais alguma coisa (algum outro atributo, associação ou operação).
- Todas as instâncias de uma subclasse têm de ser também instâncias da superclasse.

Atenção especial deve ser dada à nomeação de classes em uma hierarquia de classes. Cada especialização deve ser nomeada de forma a ser autoexplicativa. Um nome bastante apropriado para a especialização é aquele formado pelo nome de sua superclasse, acompanhado por um qualificador que descreve a natureza da especialização. P.ex., *EstudanteGraduacao* para designar um subtipo de *Estudante*.

Hierarquias de classes não devem ser usadas de forma não criteriosa, simplesmente para compartilhar algumas propriedades. Seja o caso de uma loja de animais, em que se deseja saber as seguintes informações sobre clientes e animais: nome, data de nascimento e endereço. Não faz nenhum sentido considerar que *Cliente* é uma subclasse de *Animal* ou vice-versa, apenas para reusar um conjunto de atributos que, coincidentemente, é igual.

No que se refere à modelagem de superclasses, deve-se observar se uma superclasse é concreta ou abstrata. Se a superclasse puder ter instâncias próprias, que não são instâncias de nenhuma de suas subclasses, então ela é uma classe concreta. Por outro lado, se não for possível

instanciar diretamente a superclasse, ou seja, se todas as instâncias da superclasse são antes instâncias das suas subclasses, então a superclasse é abstrata. Classes abstratas são representadas na UML com seu nome escrito em itálico.

Quando modeladas hierarquias de classes, é necessário posicionar atributos, associações e operações adequadamente. Cada atributo, associação ou operação deve ser colocado na classe mais adequada. Atributos, associações e operações genéricos, que se aplicam a todas as subclasses, devem ser posicionados no topo da estrutura, de modo a serem aplicáveis a todas as especializações. De maneira mais geral, se um atributo, associação ou operação é aplicável a um nível inteiro de especializações, então ele deve ser posicionado na generalização correspondente. Por outro lado, se um atributo, associação ou operação não for aplicável a algumas instâncias, deve-se rever seu posicionamento ou mesmo a estrutura de generalização-especialização adotada.

Inevitavelmente, a designação de atributos e associações a classes conduz a um entendimento mais completo da hierarquia de herança do que era possível em um estágio anterior. Assim, deve-se esperar que o trabalho de reposicionamento de atributos, associações e operações conduza a uma revisão da hierarquia de classes.

Por fim vale a pena mencionar que, durante anos, o mecanismo de herança foi considerado o grande diferencial da orientação a objetos. Contudo, com o passar do tempo, essa ênfase foi perdendo força, pois se percebeu que o uso da herança nem sempre conduz à melhor solução de um problema de modelagem. Hoje a herança é considerada apenas mais uma ferramenta de modelagem, utilizada basicamente para fatorar informações, as quais, de outra forma, ficariam repetidas em diferentes classes (WAZLAWICK, 2004).

Referências do Capítulo

BOOCH, G., RUMBAUGH, J., JACOBSON, I., *UML Guia do Usuário*, 2a edição, Elsevier Editora, 2006.

GUIZZARDI, G., *Ontological Foundations for Structural Conceptual Models*, Telematics Instituut Fundamental Research Series, The Netherlands, 2005.

JACOBSON, I.; *Object-Oriented Software Engineering*, Addison-Wesley, 1992.

MYLOPOULOS, J., "*Conceptual Modeling and Telos*", In: *"Conceptual Modeling, Databases and CASE*", Wiley, 1992.

OLIVÉ, A., *Conceptual Modeling of Information Systems*, Springer, 2007.

OMG, *OMG Unified Modeling Language (OMG UML)*, Version 2.5, March 2015.

WAZLAWICK, R.S., *Análise e Projeto de Sistemas de Informação Orientados a Objetos*, Elsevier, 2004.