APÊNDICE

1 INTRODUÇÃO À UML[1]

CONCEITOS-CHAVE

A UML (*Unified Modeling Language – linguagem de modelagem unificada*) é "uma linguagem-padrão para descrever/documentar projeto de software. A UML pode ser usada para visualizar, especificar, construir e documentar os artefatos de um sistema de software-intensivo" [Boo05]. Em outras palavras, assim como os arquitetos criam plantas e projetos para ser usados por uma empresa de construção, os arquitetos de software criam diagramas UML para ajudar os desenvolvedores de software a construir o software. Se você entender o vocabulário da UML (os elementos visuais do diagrama e seus significados), pode facilmente entender e especificar um sistema e explicar o projeto daquele sistema para outros interessados.

Grady Booch, Jim Rumbaugh e Ivar Jacobson desenvolveram a UML na década de 1990 com muita realimentação da comunidade de desenvolvimento de software. A UML combinou um grupo de notações de modelagem concorrentes usadas pela indústria do software na época. Em 1997, a UML 1.0 foi apresentada ao OMG (Object Management Group), uma associação sem fins lucrativos dedicada a manter especificações para ser usadas pela indústria de computadores. A UML 1.0 foi revisada tornando-se a UML 1.1 e adotada mais tarde naquele ano. O padrão atual é a UML 2.0 e agora é um padrão ISO. Em virtude desse padrão ser tão novo, muitas referências mais antigas, como [Gam95] não usam a notação UML.

A UML 2.0 fornece 13 diferentes diagramas para uso na modelagem de software. Neste apêndice, discutiremos apenas os diagramas de *classe*, *distribuição*, *caso de uso*, *sequência*, *comunicação*, *atividade* e *estado*.

Você notará que há muitas características opcionais em diagramas UML. A linguagem UML proporciona essas opções (às vezes obscuras) para que você possa expressar todos os aspectos importantes de um sistema. Ao mesmo tempo, é possível suprimir partes não relevantes ao aspecto que está sendo modelado para evitar congestionar o diagrama com detalhes irrelevantes. Portanto, a omissão de uma característica particular não significa que esteja ausente, mas sim que foi suprimida. Neste apêndice, *não* apresentamos uma discussão exaustiva de todas as características dos diagramas UML. Em vez disso, nos concentramos nas opções-padrão, especialmente as usadas neste livro.

DIAGRAMAS DE CLASSE

Para modelar classes, incluindo seus atributos, operações e relações e associações com outras classes,[2] a UML tem um *diagrama de classe*. Ele fornece uma visão estática ou estrutural de um sistema, mas não mostra a natureza dinâmica das comunicações entre os objetos das classes no diagrama.

Os elementos principais são caixas, ou seja, ícones usados para representar classes e interfaces. Cada caixa é dividida em partes horizontais. A parte superior contém o nome da classe. A seção do meio lista os atributos da classe. Um *atributo* refere-se a alguma coisa que um objeto daquela classe sabe ou pode fornecer o tempo todo. Atributos são usualmente implementados como campos da classe, mas eles não precisam ser. Poderiam ser valores que a classe calcula a partir de suas variáveis de instância ou valores que a classe pode obter de outros objetos dos quais é composta. Por exemplo, um objeto pode sempre

1 Este apêndice teve a contribuição de Dale Skrien e foi adaptado de seu livro, *An introduction to object-oriented design and design patterns in Java* (McGraw-Hill, 2008). Todo o conteúdo é usado com permissão.

2 Se você não estiver familiarizado com os conceitos orientados a objeto, é apresentada uma breve introdução no Apêndice 2.

saber a hora atual e ser capaz de retorná-la sempre que for solicitado. Portanto, seria apropriado listar a hora atual como um atributo daquela classe de objetos. No entanto, o objeto muito provavelmente não teria a hora armazenada em uma de suas variáveis de instância, porque precisaria continuamente atualizar aquele campo. Em vez disso, o objeto poderia calcular a hora atual (por exemplo, por meio da consulta a objetos de outras classes) no momento em que a hora é requisitada. A terceira seção do diagrama de classes contém as operações ou comportamentos da classe. Uma *operação* refere-se ao que os objetos da classe podem fazer. Usualmente é implementada como um método da classe.

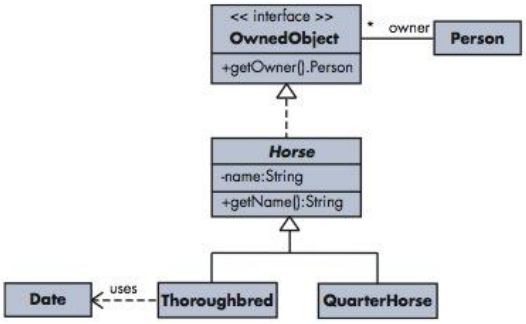
A Figura A1.1 apresenta um simples exemplo de uma classe **Thoroughbred** que modela cavalos puros-sangues. Ela mostra três atributos – **mother**, **father** e **birthyear**. Os diagramas também mostram três operações: *getCurrentAge()*, *getFather()* e *getMother()*. Pode haver outros atributos e operações suprimidos não mostrados no diagrama.

Cada atributo pode ter um nome, um tipo e um nível de visibilidade. O tipo e a visibilidade são opcionais. O tipo vem após o nome e é separado por dois-pontos. A visibilidade é indicada precedendo pelo sinal –, #, ~ ou +, indicando respectivamente, visibilidade *private, protected, package* ou *public*. Na Figura A1.1, todos os atributos têm visibilidade private, conforme indica o sinal de menos (–). Você pode também especificar que um atributo é estático ou de classe usando sublinhado. Cada operação pode também ser mostrada com um nível de visibilidade, parâmetros com nomes e tipos e um tipo de retorno.

Uma classe abstrata ou método abstrato é indicado pelo uso de itálico no nome da classe no diagrama de classes. Como exemplo, veja a classe **Horse** na Figura A1.2. Uma interface é indicada acrescentando-se a frase "«interface»" (chamada de *stereotype*) acima do nome. Veja a interface **OwnedObject** na Figura A1.2. Uma interface também pode ser representada graficamente por um círculo vazio.

FIGURA A1.1
Um diagrama para a classe Thoroughbred

FIGURA A1.2
Um diagrama de classe referente a cavalos



Vale mencionar que o ícone que representa uma classe pode ter outras partes opcionais. Por exemplo, uma quarta seção na parte inferior da caixa de classe pode ser usada para listar as responsabilidades da classe. Essa seção é particularmente útil quando se faz a transição dos cartões CRC (Capítulo 6) para diagramas de classe em que as responsabilidades listadas nos cartões CRC podem ser acrescentadas à quarta seção na caixa da classe no diagrama UML antes de que os atributos e operações que executam essas responsabilidades sejam criados. Essa quarta seção não é mostrada em qualquer uma das figuras neste apêndice.

Os diagramas de classe também podem exibir relações entre classes. Uma classe que seja subclasse de outra classe é conectada a ela por uma seta com uma linha sólida como eixo e com uma ponta triangular vazia. A seta aponta da subclasse para a superclasse. Em UML, uma relação como essa é chamada de *generalização*. Por exemplo, na Figura A1.2, as classes **Thoroughbred** e **QuarterHorse** são exibidas como subclasses da classe abstrata **Horse**. Uma seta tendo como eixo uma linha tracejada indica implementação de interface. Em UML, esse tipo de relação é chamada de *realização*. Na Figura A1.2, a classe **Horse** implementa ou realiza a interface **OwnedObject**.

A *associação* entre duas classes indica que há uma relação estrutural entre elas. Associações são representadas por linhas sólidas. Uma associação tem muitas partes opcionais. Ela pode ser rotulada, assim como cada uma de suas extremidades, para indicar o papel de cada classe na associação. Por exemplo, na Figura A1.2, há uma associação entre **OwnedObject** e **Person** na qual **Person** desempenha o papel de proprietário (*owner*). Setas em qualquer uma ou ambas as extremidades de uma linha de associação indicam navegabilidade. Além disso, cada extremidade da linha de associação pode ter um valor de multiplicidade. Navegabilidade e multiplicidade são explicadas em maior detalhe mais à frente nesta seção. Uma associação pode também conectar uma classe com ela própria, usando um laço. Desse modo, uma associação indica a conexão de um objeto da classe com outros objetos da mesma classe.

A associação com uma seta em uma extremidade indica navegabilidade unidirecional. A seta significa que de uma classe pode-se facilmente acessar a segunda classe associada para a qual aponta a associação. A partir da segunda classe, não se pode necessariamente acessar com facilidade a primeira classe. Outra maneira de pensar sobre isso é que a primeira classe tem conhecimento da segunda classe, enquanto o objeto da segunda classe não necessariamente conhece a primeira classe. Uma associação sem setas em geral indica uma associação bidirecional, que é o que se pretendia na Figura A1.2, mas podia também significar apenas que a navegabilidade não é importante e foi deixada de lado.

Deve-se notar que um atributo de uma classe é muito parecido com uma associação da classe com o tipo de classe do atributo. Isto é, para indicar que uma classe tem uma propriedade chamada "nome" do tipo String, poderíamos mostrar aquela propriedade como um atributo, como na classe **Horse** na Figura A1.2. Como alternativa, poderíamos criar uma associação unidirecional da classe **Horse** para a classe **String,** sendo "nome" o papel da classe **String**. A abordagem de atributo é melhor para tipos de dados primitivos, enquanto a abordagem associação muitas vezes é melhor se a classe da propriedade desempenha um papel importante no projeto, caso em que é muito bom ter uma caixa de classe para aquele tipo.

Uma relação de *dependência* representa outra conexão entre classes e é indicada por uma linha tracejada (com setas opcionais nas extremidades e com rótulos opcionais). Uma classe depende de outra se alterações na segunda classe podem requerer alterações na primeira classe. Uma associação de uma classe para outra automaticamente indica uma dependência. Não é necessária uma linha pontilhada entre classes se já houver uma associação entre elas. No entanto, para uma relação transiente (uma classe que não mantém relação de longo prazo com outra classe, mas usa aquela classe ocasionalmente), podemos colocar uma linha tracejada da primeira classe para a segunda. Por exemplo, na Figura A1.2, a classe **Thoroughbred** usa a classe **Date** sempre que é invocado o método *getCurrentAge()*, e assim a dependência é chamada de "uses".

A *multiplicidade* de extremidade de uma associação indica o número de objetos daquela classe associados a outra classe. Uma multiplicidade é especificada por um inteiro não negativo ou

por um intervalo de inteiros. Uma multiplicidade especificada como "0..1" significa que há 0 ou 1 objeto na extremidade da associação. Por exemplo, cada pessoa no mundo tem um número de Seguro Social ou não tem tal número (especialmente se não forem cidadãos americanos), e assim uma multiplicidade de 0..1 poderia ser usada em uma associação entre uma classe **Person** e uma classe **SocialSecurityNumber** no diagrama de classes. A multiplicidade especificada por "1..*" significa um ou mais, e a multiplicidade especificada por "0..*" ou apenas "*" significa zero ou mais. Usou-se um * como multiplicidade na extremidade **OwnedObject** da associação com a classe **Person** na Figura A1.2 porque uma **Person** pode possuir zero ou mais objetos.

Se a extremidade de uma associação apresenta multiplicidade maior do que 1, os objetos da classe aos quais se faz referência na extremidade da associação estão provavelmente armazenados em uma coleção, como um conjunto ou uma lista ordenada. Poderíamos também incluir a própria classe de coleção no diagrama UML, mas uma classe desse tipo usualmente é desconsiderada e assume-se, implicitamente, que esteja lá devido à multiplicidade da associação.

Uma *agregação* é um tipo especial de associação representada por um losango vazio em uma extremidade do ícone. Ela indica uma relação "todo/parte", em que a classe para a qual a seta aponta é considerada uma "parte" da classe na extremidade do losango da associação. Uma *composição* é uma agregação indicando forte relação de propriedade entre as partes. Em uma composição, as partes vivem e morrem com o proprietário porque não têm um papel a desempenhar no sistema de software independente do proprietário. Veja na Figura A1.3 exemplos de agregação e composição.

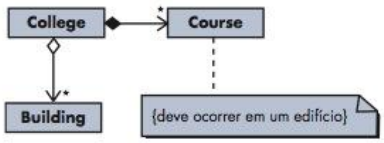
Uma classe **College** tem uma agregação de objetos **Building**, que representam os edifícios que formam o *campus*. A classe **College** tem também uma coleção de cursos. Mesmo que a universidade fechasse, os edifícios ainda continuariam a existir (supondo que a universidade não fosse fisicamente destruída) e poderiam ser usados para outros propósitos, mas um objeto **Course** não tem utilidade fora da universidade na qual está sendo oferecido. Se a universidade (**College**) deixasse de existir como uma entidade de negócios, o objeto **Course** não teria mais utilidade e, portanto, também deixaria de existir.

Outro elemento comum em diagramas de classes é uma anotação (*note*) representada por um caixa com um canto dobrado e é conectada a outros ícones por uma linha pontilhada. Ela pode ter conteúdo arbitrário (texto e gráficos) e é similar aos comentários nas linguagens de programação. Pode ter comentários sobre o papel de uma classe ou restrições que todos os objetos daquela classe devem satisfazer. Se o conteúdo for uma restrição, estará entre chaves. Observe a restrição anexada à classe **Course** na Figura A1.3.

Diagramas de Disponibilização

Diagramas de distribuição focalizam a estrutura de um sistema de software e são úteis para mostrar a distribuição física de um sistema de software entre plataformas de hardware e ambientes de execução. Por exemplo, suponha que você esteja desenvolvendo um pacote de renderização gráfica baseado na Web. Os usuários do seu pacote de software usarão o navegador Web para acessar o seu site e introduzir as informações de renderização. O seu site irá renderizar uma imagem gráfica de acordo com as especificações do usuário e a enviará de volta ao usuário. Como a renderização gráfica pode ser cara em termos de computação, você decide mudar a renderização para fora do servidor Web, colocando-a em uma plataforma separada. Portanto,

Figura A1.3
As relações entre Colleges, Courses e Buildings



haverá três dispositivos de hardware envolvidos no seu sistema: o cliente Web (o computador do usuário executando um navegador), o computador que está hospedando o servidor Web e o computador que está hospedando o dispositivo de renderização.

A Figura A1.4 mostra o diagrama de distribuição para um pacote de software como esse. Neles, os componentes de hardware são desenhados como caixas com o título "«device»". Os caminhos de comunicação entre os componentes de hardware são traçados com linhas com títulos opcionais. Na Figura A1.4, os caminhos são identificados com o protocolo de comunicação e o tipo de rede usada para conectar os dispositivos.

Cada nó em um diagrama de distribuição pode também ser anotado com detalhes sobre o dispositivo. Por exemplo, na Figura A1.4, é desenhado o navegador cliente para mostrar que contém um artefato, que é o software do navegador Web. Artefato é tipicamente um arquivo contendo software executando em um dispositivo. Você pode também especificar valores rotulados, como mostra a Figura A1.4 no nó do servidor Web. Esses valores definem o fornecedor do servidor Web e o sistema operacional usado pelo servidor.

Diagramas de distribuição podem também mostrar os nós do ambiente de execução, desenhados em caixas contendo o rótulo "«ambiente de execução»". Esses nós representam sistemas, como os sistemas operacionais, que podem hospedar outro software.

Diagramas de Casos de Uso

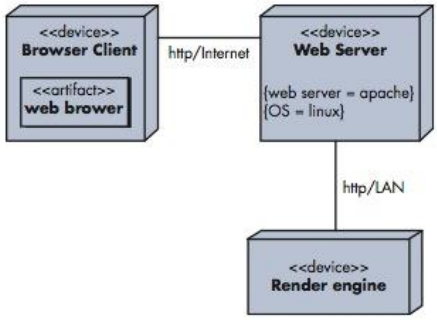
Casos de uso (Capítulos 5 e 6) e o *diagrama de caso de uso* ajudam a determinar a funcionalidade e as características do software sob o ponto de vista do usuário. Para uma ideia de como os casos de uso e os diagramas de caso de uso funcionam, vamos criar alguns deles para um aplicativo de software para gerenciar arquivos de música digital, similar ao software iTunes da Apple.

Algumas das coisas que o software pode fazer são:

- Baixar um arquivo de música MP3 e armazená-lo na biblioteca do aplicativo.
- Capturar a música e armazená-la na biblioteca do aplicativo.
- Gerenciar a biblioteca do aplicativo (por exemplo, excluir músicas ou organizá-las em listas de execução).
- Gravar em um CD uma lista de músicas da biblioteca.
- Carregar uma lista de músicas da biblioteca para um iPod ou MP3 player.
- Converter uma música do formato MP3 para o formato AAC e vice-versa.

Essa não é uma lista completa, mas suficiente para entendermos a função dos casos de uso e diagramas de caso de uso.

Figura A1.4
Diagrama de distribuição



Um caso de uso descreve como um usuário interage com o sistema definindo os passos necessários para atingir um objetivo específico (por exemplo, gravar uma lista de músicas em um CD). Variações na sequência de passos descrevem vários cenários (por exemplo, o que acontece se as músicas da lista não couberem em um CD?).

Um diagrama UML de caso de uso é uma visão geral de todos os casos de uso e como estão relacionados. Fornece uma visão geral da funcionalidade do sistema. Um diagrama de caso de uso para o aplicativo de música digital é mostrado na Figura A1.5.

Neste diagrama, a figura do usuário representa um *ator* (Capítulo 5) que está associado a uma categoria de usuário (ou outro elemento de interação). Sistemas complexos tipicamente possuem mais de um ator. Por exemplo, um aplicativo "vending machine" pode ter três atores representando clientes, pessoal de manutenção e os fornecedores que abastecem a máquina.

No diagrama de caso de uso, estes são mostrados como ovais. Os atores são conectados por linhas aos casos de uso que eles executam. Note que nenhum dos detalhes dos casos de uso é incluído no diagrama e precisa ser armazenado separadamente. Observe também que os casos de uso são colocados em um retângulo, mas os atores não. Esse retângulo serve para lembrar visualmente as fronteiras do sistema e que os atores estão fora do sistema.

Alguns casos de uso em um sistema podem estar relacionados uns com os outros. Por exemplo, há passos similares para gravar uma lista de músicas para um CD e para carregar uma lista de músicas para um iPod. Em ambos os casos, o usuário primeiro cria uma lista vazia e, em seguida, acrescenta as músicas da biblioteca para a lista. Para evitar duplicação, normalmente é melhor criar um novo caso de uso representando a atividade duplicada, e depois deixar que os outros casos incluam esse novo caso de uso com um de seus passos. A inclusão é indicada nos diagramas de caso de uso, como mostra a Figura A1.6, por meio de uma seta tracejada identificada como «include» conectando um caso de uso a outro.

FIGURA A1.5
Um diagrama de caso de uso para o sistema de música

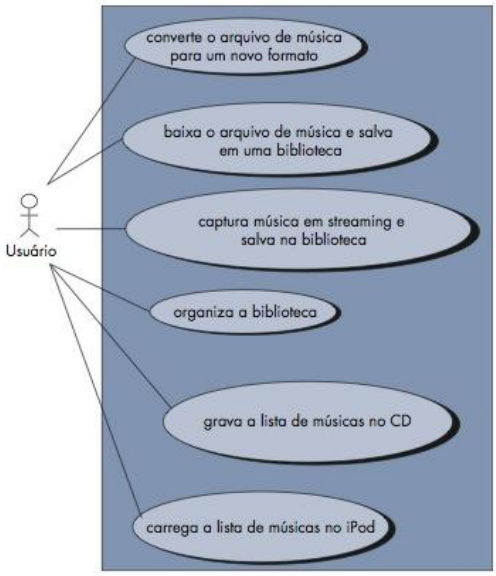
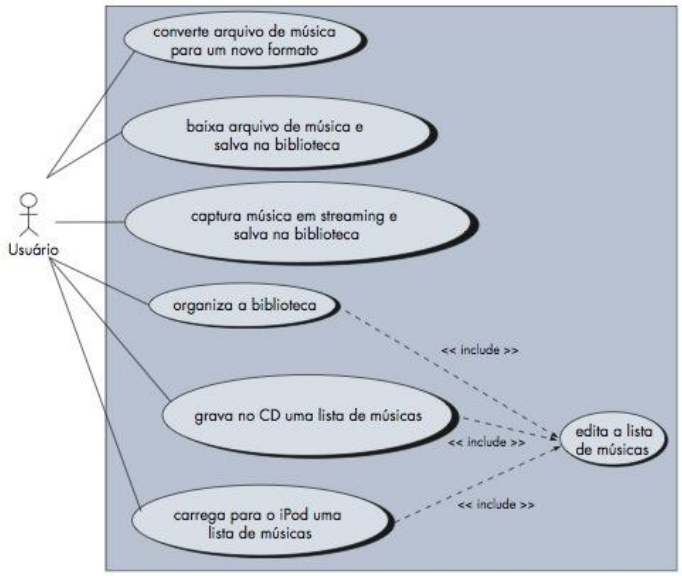


FIGURA A1.6

Um diagrama de caso de uso com casos de uso incluídos



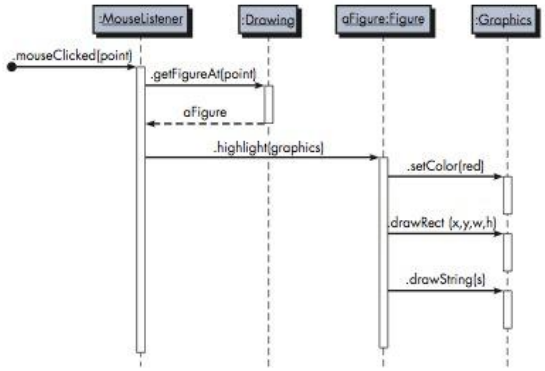
Um diagrama de caso de uso, por mostrar todos os casos, é um bom auxílio para assegurar inclusão de toda a funcionalidade do sistema. No organizador de música digital, certamente você iria querer ter mais casos de uso, como, por exemplo, um para tocar uma música da biblioteca. Mas tenha em mente que a maior contribuição dos casos de uso para o processo de desenvolvimento de software é a descrição textual de cada caso e não o diagrama geral de caso de uso [Fow04b]. É por meio das descrições que você consegue formar uma ideia clara dos objetivos do sistema que está desenvolvendo.

DIAGRAMAS DE SEQUÊNCIA

Em contraste com os diagramas de classe e os de distribuição, que mostram a estrutura estática de um componente de software, o *diagrama de sequência* é utilizado para indicar as comunicações dinâmicas entre objetos durante a execução de uma tarefa. Ele mostra a ordem temporal na qual as mensagens são enviadas entre os objetos para executar aquela tarefa. Podemos usar um diagrama de sequência para mostrar as interações em um caso de uso ou em um cenário de um sistema de software.

Na Figura A1.7, há um diagrama de sequência para um programa de desenho. O diagrama mostra os passos envolvidos para destacar uma figura em um desenho quando é clicada. Cada caixa na linha do topo do diagrama em geral corresponde a um objeto, embora seja possível fazer as caixas modelarem outras coisas, como por exemplo, classes. Se a caixa representa um objeto (como é o caso em todos os nossos exemplos), dentro da caixa pode-se opcionalmente declarar o tipo do objeto precedido de dois-pontos. Você pode também colocar antes dos dois-pontos e do tipo o nome do objeto, conforme mostra a terceira caixa na Figura A1.7. Abaixo de cada caixa há uma linha tracejada chamada de *linha de vida* do objeto. O eixo vertical no diagrama de sequência corresponde ao tempo, e o tempo aumenta à medida que se caminha para baixo.

FIGURA A1.7
Exemplo de um diagrama de sequência



Um diagrama de sequência mostra chamadas de método usando setas horizontais do *chamador* para o *chamado*, identificadas com o nome do método e opcionalmente incluindo seus parâmetros, seus tipos e o tipo de retorno. Por exemplo, na Figura A1.7, **MouseListener** chama o método *getFigureAt() de* **Drawing**. Quando um objeto está executando um método (quando tem uma ativação de um método na pilha), você pode opcionalmente mostrar uma barra branca, conhecida como *barra de ativação*, ao longo da linha de vida do objeto. Na Figura A1.7, há barras de ativação para todas as chamadas de método. O diagrama também pode mostrar opcionalmente o retorno de uma chamada de método com uma seta pontilhada e um rótulo opcional. Na Figura A1.7, o retorno da chamada do método *getFigureAt()* é identificado com o nome do objeto retornado. Uma prática comum, como fizemos na Figura A1.7, é omitir a seta de retorno quando um método não retorna nada (*void*), porque isso complicaria o diagrama fornecendo informações de pouca importância. Um círculo preto com uma seta indica uma *mensagem encontrada* cuja origem é desconhecida ou irrelevante.

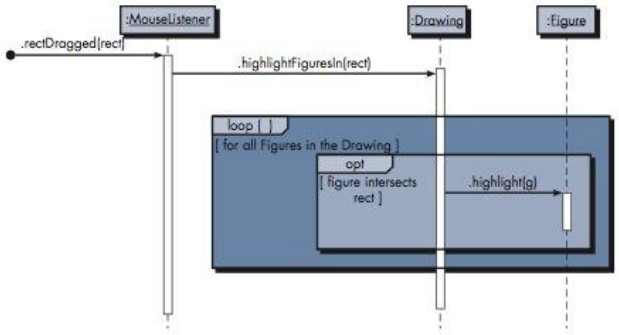
Você será capaz de entender agora a tarefa que a Figura A1.7 está mostrando. Uma origem desconhecida chama o método *mouseClicked()* de um **MouseListener**, passando como argumento o ponto onde o clique ocorreu. O **MouseListener** por sua vez chama o método *getFigureAt()* de um **Drawing**, que retorna um **Figure.** O **MouseListener** então chama o método destacado de **Figure**, passando um objeto **Graphics** como argumento. Em resposta, **Figure** chama três métodos do objeto **Graphics** para traçar a figura em vermelho.

O diagrama na Figura A1.7 é muito claro e não contém condicionais ou laços. Se forem necessárias estruturas lógicas de controle, provavelmente será melhor traçar um diagrama de sequência separado para cada caso. Isto é, se o fluxo de mensagem puder tomar dois caminhos diferentes dependendo de uma condição, trace dois diagramas de sequência separados, um para cada possibilidade.

Se você ainda quer incluir laços, condicionais e outras estruturas de controle em um diagrama de sequência, use frames de interação (*interaction frames)*, que são retângulos que envolvem as partes do diagrama e são identificados com o tipo de estruturas de controle que representam. A Figura A1.8 ilustra isso, mostrando o processo envolvido para destacar todas as figuras em um retângulo. Para o **MouseListener** é enviada a mensagem **rectDragged**. O **MouseListener** então manda o desenho destacar todas as figuras no retângulo chamando o método *highlightFigures()*, passando o retângulo como argumento. O método passa em laço por todos os objetos **Figure** no objeto **Drawing** e, se o objeto **Figure** intercepta o retângulo, é solicitado a **Figure** que se destaque. As frases entre colchetes são chamadas de *guardas*, que são condições booleanas que devem ser verdadeiras se a ação dentro da frame de interação deve continuar.

FIGURA A 1.8

Um diagrama de sequência com dois frames de interação



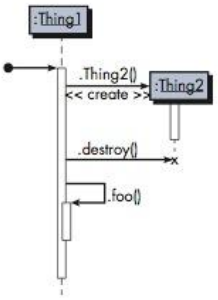
Há muitas outras características que podem ser incluídas em um diagrama de sequência. Por exemplo:

1. Você pode distinguir entre mensagens síncronas e assíncronas. Mensagens síncronas são exibidas com pontas de setas sólidas, enquanto mensagens assíncronas são mostradas com pontas de seta em traço.
2. Você pode mostrar um objeto fazendo-o enviar a si próprio uma mensagem com uma seta saindo do objeto, virando para baixo e, em seguida, apontando de volta para o mesmo objeto.
3. Você pode mostrar a criação do objeto traçando uma seta identificada de forma apropriada (por exemplo, com um rótulo «create») para a caixa de um objeto. Nesse caso, a caixa aparecerá no diagrama abaixo das caixas que correspondem a objetos que já existiam quando a ação começa.
4. Você pode representar a destruição de um objeto com um X grande no fim da linha de vida do objeto. Outros objetos podem destruir um objeto e, nesse caso, uma seta aponta do outro objeto para o X. Um X é útil também para indicar que um objeto não é mais utilizável e está pronto para ser enviado à coleta de lixo.

As três últimas características são mostradas no diagrama de sequência na Figura A1.9.

FIGURA A1.9

Criação, destruição e laços em diagramas de sequência



DIAGRAMAS DE COMUNICAÇÃO

O *diagrama de comunicação* UML (chamado de "diagrama de colaboração" em UML 1.X) fornece outra indicação da ordem temporal das comunicações, mas dá ênfase às relações entre os objetos e classes em vez da ordem temporal.

O diagrama de comunicação, ilustrado na Figura A1.10, exibe as mesmas ações do diagrama de sequência da Figura A1.7.

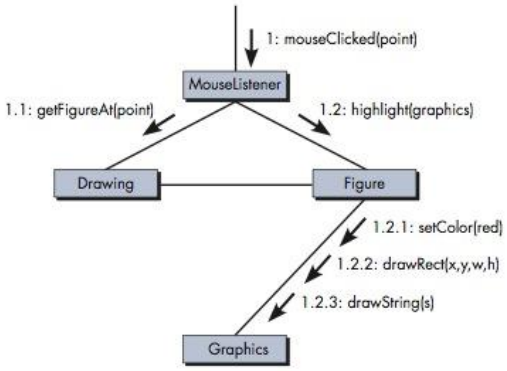
Em um diagrama de comunicação, os objetos que interagem são representados por retângulos. Associações entre objetos são representadas por linhas ligando os retângulos. Há tipicamente uma seta apontando para um objeto no diagrama, que inicia a sequência de passagem de mensagens. A seta é identificada com um número e um nome de mensagem. Se a mensagem, que chega, for identificada com o número 1 e se ela faz o objeto receptor invocar outras mensagens em outros objetos, aquelas mensagens são representadas por setas do emissor para o receptor com uma linha de associação e recebem números 1.1, 1.2 e assim por diante, na ordem em que são chamadas. Se aquelas mensagens por sua vez invocam outras, é acrescentado outro ponto e outro número ao número que as identifica, para indicar novo aninhamento da mensagem passada.

Na Figura A1.10, você vê que a mensagem **mouseClicked** chama os métodos *getFigureAt()* e depois *highlight()*. A mensagem *highlight()* chama três outras mensagens: *setColor(), drawRect()* e *drawstring()*. A numeração em cada rótulo mostra os aninhamentos, bem como a natureza sequencial de cada mensagem.

Há muitas características opcionais que podem ser acrescentadas aos rótulos das setas. Por exemplo, você pode colocar uma letra na frente do número. Uma seta chegando poderia ser marcada como **A1: mouseClicked(point).** indicando a execução de uma sequência de comandos (thread), A. Se outras mensagens são executadas em outras threads, o rótulo seria precedido por uma letra diferente. Por exemplo, se o método *mouseClicked()* é executado na thread A, mas ele cria uma nova thread B e chama *highlight()* naquela thread, então a seta de **MouseListener** para **Figure** seria rotulada como **1.B2: highlight(graphics)**.

Se você estiver interessado em mostrar as relações entre os objetos além das mensagens que estão sendo enviadas entre eles, o diagrama de comunicação é provavelmente uma opção melhor do que o diagrama de sequência. Se você estiver mais interessado na ordem temporal da mensagem enviada, o diagrama de sequência provavelmente será melhor.

FIGURA A1.10
Um diagrama de comunicação UML



Diagramas de Atividade

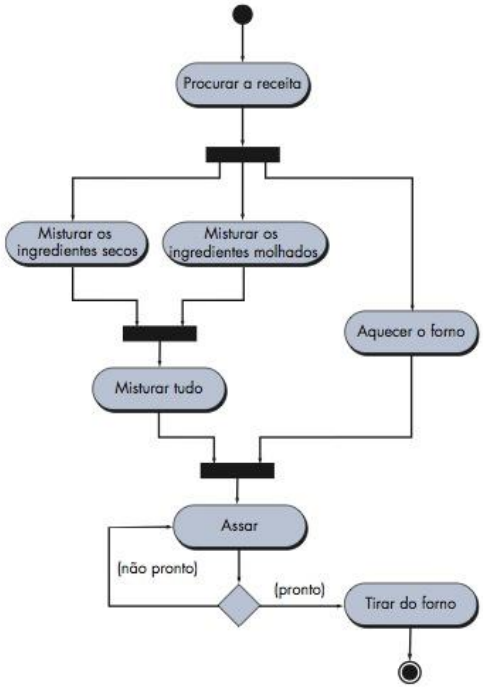
O *diagrama de atividade* mostra o comportamento dinâmico de um sistema ou parte de um sistema através do fluxo de controle entre ações que o sistema executa. Ele é similar a um fluxograma exceto que pode mostrar fluxos concorrentes.

O componente principal de um diagrama de atividade é um nó *ação*, representado por um retângulo arredondado, que corresponde a uma tarefa executada por um sistema de software. Setas que vão de um nó ação para outro indicam o fluxo de controle. Isto é, uma seta entre dois nós ação significa que depois que a primeira ação é completada a segunda ação começa. Um ponto preto sólido forma o *nó inicial* que representa o ponto inicial da atividade. Um ponto preto envolvido por um círculo preto é o *nó final* indicando o fim da atividade.

Um *fork* representa a separação de atividades em duas ou mais atividades concorrentes. Ela é desenhada como uma barra preta horizontal com uma seta apontando para ela e duas ou mais setas apontando para fora dela. Cada seta de saída representa um fluxo de controle que pode ser executado concorrentemente com os fluxos que correspondem a outras setas que saem. Essas atividades concorrentes podem ser executadas em um computador por meio de diferentes threads ou mesmo usando diferentes computadores.

A Figura A1.11 mostra um exemplo de um diagrama de atividade envolvendo a confecção de um bolo. O primeiro passo é procurar a receita. Uma vez encontrada, os ingredientes secos e molhados podem ser medidos e misturados e o forno pode ser pré-aquecido. A mistura dos ingredientes secos pode ser feita em paralelo com a mistura dos ingredientes molhados e com o pré-aquecimento do forno.

Figura A1.11
Um diagrama de atividade UML demonstrando como fazer um bolo



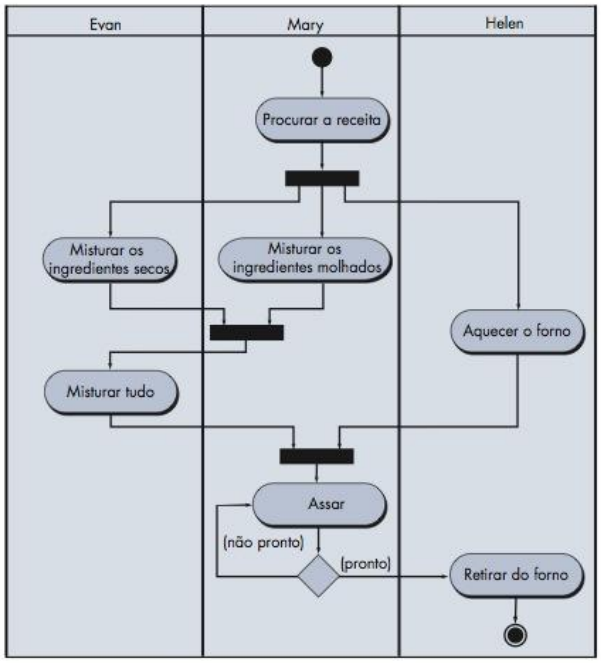
Uma *junção* (*join*) é uma maneira de sincronizar fluxos de controle concorrentes. Ela é representada por uma barra preta horizontal com duas ou mais setas chegando e uma seta saindo. O fluxo de controle representado pela seta que sai não pode iniciar a execução até que todos os outros fluxos representados pelas setas que chegam tenham sido completados. Na Figura A1.11, temos uma junção antes da ação de mistura dos ingredientes secos e molhados. Ela indica que todos os ingredientes secos devem ser misturados e todos os ingredientes molhados devem ser misturados antes que as duas misturas possam ser combinadas. A segunda junção na figura informa que, antes de começar a assar o bolo, todos os ingredientes devem estar misturados e o forno deve estar na temperatura correta.

Um nó *decisão* corresponde a uma ramificação no fluxo de controle baseada em uma condição. Um nó desse tipo é mostrado como um triângulo branco com uma seta chegando e duas ou mais saindo. Cada seta que sai é identificada com uma condição entre colchetes (*guard*). O fluxo de controle segue a seta que sai cuja condição é verdadeira (*true*). É bom certificar- se de que as condições abrangem todas as possibilidades de forma que exatamente uma delas seja verdadeira sempre que for encontrado um nó de decisão. A Figura A1.11 apresenta um nó de decisão após assar o bolo. Se o bolo estiver pronto, ele é removido do forno. Caso contrário, permanece no forno por mais um tempo.

Uma das coisas que o diagrama de atividades da Figura A1.11 não mostra é quem executa cada uma das ações. Muitas vezes, a divisão exata do trabalho não importa. Mas se você quiser indicar como as ações são divididas entre os participantes, decore o diagrama de atividades com raias (*swimlanes*), como mostra a Figura A1.12. *As raias*, como o nome diz, são formadas

FIGURA A1.12

O diagrama de atividades da confecção do bolo com a inclusão de raias



dividindo-se o diagrama em tiras ou "faixas", e cada uma dessas faixas corresponde a um dos participantes. Todas as ações em uma faixa são executadas pelo participante correspondente. Na Figura A1.12, Evan é responsável pela mistura dos ingredientes secos e depois pela mistura dos ingredientes secos e molhados juntos, Helen é responsável por aquecer o forno e tirar o bolo e Mary por todo o restante.

Diagramas de Estado

O comportamento de um objeto em determinado instante frequentemente depende do estado do objeto, ou seja, os valores de suas variáveis naquele instante. Como um exemplo trivial, considere um objeto com uma variável de instância booleana. Quando solicitado a executar uma operação, o objeto pode realizar algo se a variável for verdadeira *(true)* e realizar outra coisa se for falsa *(false)*.

Um *diagrama de estado* modela os estados de um objeto, as ações executadas dependendo daqueles estados e as transições entre os estados do objeto.

Como exemplo, considere o diagrama de estado para uma parte de um compilador Java. A entrada do compilador é um arquivo de texto, que pode ser considerado uma longa sequência (string) de caracteres. O compilador lê os caracteres, um de cada vez, e a partir deles determina a estrutura do programa. Uma pequena parte desse processo consiste em ignorar caracteres "espaço em branco" (por exemplo, os caracteres *espaço, tabulação, nova linha,* e *"return"*) e caracteres dentro de um comentário.

Suponha que o compilador delegue ao objeto **WhiteSpaceAndCommentEliminator** a tarefa de avançar sobre os caracteres espaço em branco e caracteres dentro de comentários. A tarefa desse objeto é ler os caracteres de entrada até que todos os espaços em branco e os caracteres entre comentários tenham sido lidos. Nesse ponto ele retorna o controle para o compilador para ler e processar caracteres que não são espaços em branco e não são caracteres de comentários. Pense em como o objeto **WhiteSpaceAndCommentEliminator** lê os caracteres e determina se o próximo caractere é um espaço em branco ou se é parte de um comentário. O objeto pode verificar espaços em branco testando o próximo caractere para saber se é " ", "\t", "\n", e "\r". Mas como ele determina se o próximo caractere é parte de um comentário? Por exemplo, quando vê uma "/" pela primeira vez, ele ainda não sabe se aquele caractere representa um operador de divisão, parte do operador /= ou o início de uma linha ou comentário de bloco. Para determinar isso, **WhiteSpaceAndCommentEliminator** precisa registrar o fato de que viu uma "/" e então passar para o próximo caractere. Se o caractere que vem depois da "/" for uma outra "/" ou um "*", então **WhiteSpaceAndCommentEliminator** saberá que ele está agora lendo um comentário e pode avançar até o fim sem processar ou salvar qualquer caractere. Se o caractere que vem em seguida à primeira "/" for alguma coisa diferente de uma "/" ou um "*", o objeto **WhiteSpaceAndCommentEliminator** saberá que a "/" representa o operador divisão ou parte do operador /= e assim ele para de avançar sobre os caracteres.

Resumindo, à medida que o objeto **WhiteSpaceAndCommentEliminator** lê os caracteres, ele precisa controlar vários aspectos, incluindo se o caractere atual é um espaço em branco, se o caractere anterior que ele leu era uma "/", se ele está no momento lendo caracteres de um comentário, se chegou ao fim de um comentário e assim por diante. Tudo isso corresponde a diferentes estados do objeto **WhiteSpaceAndCommentEliminator**. Em cada um desses estados, **WhiteSpaceAndCommentEliminator** se comporta diferentemente com relação ao próximo caractere a ser lido.

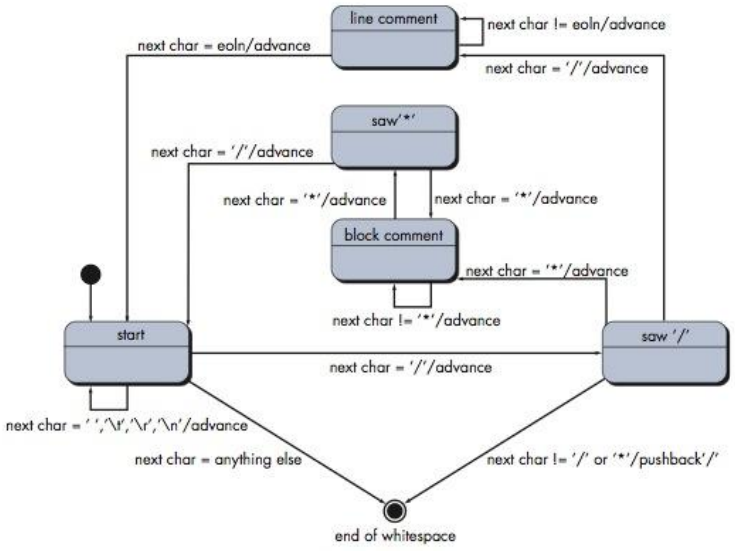
Para ajudar a visualizar todos os estados desse objeto e como ele muda o estado, você pode usar um diagrama de estado conforme indica a Figura A1.13. Um diagrama de estado mostra os estados através de retângulos arredondados, cada um dos quais tem um nome em sua metade superior. Há também um círculo preto chamado de "pseudoestado inicial", que não é realmente

um estado, mas apenas pontos para o estado inicial. Na Figura A1.13, o estado **start** é o estado inicial. Setas de um estado para outro representam transições ou mudanças no estado do objeto. Cada transição é identificada com um evento disparo, uma barra (/), e uma atividade. Todas as partes dos rótulos de transição são opcionais nos diagramas de estado. Se o objeto está em um estado e ocorre o disparo de evento, para uma de suas transições, a atividade daquela transição é executada e o objeto assume o novo estado indicado pela transição. Por exemplo, na Figura A1.13, se o objeto **WhiteSpaceAndCommentEliminator** estiver no estado **start** e o próximo caractere for "/", **WhiteSpaceAndCommentEliminator** avança além daquele caractere e muda para o estado **saw '/'**. Se o caractere depois de "/" for outra "/", o objeto avança para o estado **line comment** e permanece lá até ler um caractere fim de linha. Se por outro lado, o próximo caractere após a "/" for um "*", o objeto avança para o estado **block comment** e permanece lá até encontrar outro "*" seguido de uma "/", que indica o fim de um comentário de bloco. Estude o diagrama para ter certeza de que o entendeu. Note que, após avançar além do espaço em branco ou de um comentário, o objeto **WhiteSpaceAndCommentEliminator** volta para o estado **start** e começa tudo novamente. Esse comportamento é necessário, já que pode haver vários comentários ou caracteres de espaço em branco sucessivos antes de encontrar quaisquer outros caracteres do código-fonte Java.

O objeto pode fazer uma transição para um estado final, indicado por um círculo preto com um círculo branco ao redor dele, que informa que não há mais transições. Na Figura A1.13, o objeto **WhiteSpaceAndCommentEliminator** é encerrado quando o próximo caractere não é um espaço em branco ou parte de um comentário. Note que todas as transições, exceto as duas que levam ao estado final, têm atividades que consistem em avançar para o próximo caractere. As duas transições para o estado final não avançam sobre o próximo caractere porque o pró-

FIGURA A1.13

Um diagrama de estado para avançar além do espaço em branco e comentários em Java



ximo é parte de uma palavra ou símbolo de interesse para o compilador. Note que se o objeto estiver no estado **saw '/'** mas o próximo caractere não é uma "/" ou "*", então o caractere "/" é um operador divisão ou parte do operador /= e então não queremos avançar. Na verdade, queremos retroceder um caractere para tornar a "/" o próximo caractere, para que assim o caractere "/" possa ser usado pelo compilador. Na Figura A1.13, essa atividade de retrocesso é chamada de pushback '/'.

Um diagrama de estado o ajudará a descobrir situações perdidas ou inesperadas. Com um diagrama de estado, é relativamente fácil garantir que todos os eventos possíveis para todos os estados possíveis foram levados em conta. Na Figura A1.13, você pode facilmente verificar que cada estado tenha incluído transições para todos os caracteres possíveis.

Os diagramas de estado podem conter muitas outras características não incluídas na Figura A1.13. Por exemplo, quando um objeto está em um estado, ele usualmente não faz nada e espera até que ocorra um evento. No entanto, há um tipo especial de estado, denominado *estado de atividade*, no qual o objeto executa alguma atividade, chamada de *do-activity*, enquanto ele está naquele estado. Para indicarmos que um estado é um estado de atividade no diagrama, incluímos na metade inferior do retângulo arredondado de estado a frase "do/" seguida da atividade que deve ser executada enquanto estiver naquele estado. A do-activity pode terminar antes que ocorram quaisquer transições de estado, o estado atividade se comporta como um estado de espera normal. Se ocorrer uma transição fora do estado atividade, antes que a do-activity tenha terminado a do-activity é interrompida.

Devido a um evento ser opcional quando ocorre uma transição, é possível que nenhum evento possa estar listado como parte do rótulo de uma transição. Em casos assim para estados de espera normais, o objeto fará imediatamente uma transição daquele estado para o novo estado. Para estados de atividade, uma transição dessas ocorre logo que do-activity termina.

A Figura A1.14 ilustra essa situação usando os estados para um telefone comercial. Quando uma chamada é colocada em espera, vai para o estado **on hold with music** (música de espera toca por 10 segundos). Após 10 segundos, a do-activity do estado é completada e o estado se comporta como um estado normal de não atividade. Se a pessoa que está chamando pressionar a tecla # quando a chamada está no estado **on hold with music**, a chamada faz uma transição para o estado **canceled**, e o telefone passa imediatamente para o estado **dial tone**. Se a tecla # for pressionada antes de completar os 10 segundos de música de espera, a do-activity é interrompida e a música para imediatamente.

FIGURA A1.14

Um diagrama de estado com um estado de atividade e uma transição "triggerless"

OBJECT CONSTRAINT LANGUAGE – UMA VISÃO GERAL

A grande variedade de diagramas disponíveis como parte da UML proporciona um excelente conjunto de formas de representação para o modelo de projeto. No entanto, as representações gráficas muitas vezes não são suficientes. Você pode precisar de um mecanismo para representar explícita e formalmente informações que restringem alguns elementos do modelo de projeto. É possível, naturalmente, descrever as restrições em uma linguagem natural como o inglês, mas essa abordagem invariavelmente leva a inconsistência e ambiguidade. Por essa razão, parece ser apropriada uma linguagem mais formal – que use a teoria dos conjuntos e linguagens de especificação formal (veja o Capítulo 21), mas tenha de certa forma a sintaxe menos matemática de uma linguagem de programação.

A *Object Constraint Language* (OCL) complementa a UML, permitindo que você use uma gramática e sintaxe formais para construir instruções não ambíguas sobre vários elementos do modelo de projeto (por exemplo, classes e objetos, eventos, mensagens, interfaces). As instruções OCL mais simples são criadas em quatro partes: (1) um *contexto* que define a situação limitada na qual a instrução é válida, (2) uma *propriedade* que representa algumas características do contexto (por exemplo, se o contexto é uma classe, uma propriedade pode ser um atributo), (3) uma *operação* (por exemplo, aritmética, orientada a conjunto) que manipula ou qualifica uma propriedade, e (4) palavras-chave (por exemplo, **if, then, else, and, or, not, implies**) usadas para especificar expressões condicionais.

Como um exemplo simples de uma expressão OCL, considere o sistema de impressão discutido no Capítulo 10. A condição de guarda (*guard condition*) colocada no evento **jobCostAccepted** que causa uma transição entre os estados *computingJobCost* e *formingJob* dentro do diagrama de estado (*statechart diagram*) para o objeto **PrintJob** (Figura 10.9). No diagrama (Figura 10.9), a condição de guarda é expressa em linguagem natural e implica que a autorização só pode ocorrer se o cliente está autorizado para aprovar o custo do trabalho. Em OCL, a expressão pode assumir a forma:

```
customer
   self.authorizationAuthority = 'yes'
```

em que um atributo booleano, **authorizationAuthority, da classe** (na realidade uma instância específica da classe) denominada **Customer** deve ser definida como "yes" para que a condição de guarda seja satisfeita.

Na medida em que o modelo de projeto é criado, muitas vezes há instâncias nas quais pré ou pós-condições devem ser satisfeitas antes de completar alguma ação especificada pelo projeto. A OCL proporciona uma poderosa ferramenta para especificar pré e pós-condições de uma maneira formal. Como exemplo, considere uma ampliação do sistema de impressão (discutido como exemplo no Capítulo 10), no qual o cliente estabelece um limite superior de custo para a impressão e uma data final de entrega no instante em que são especificadas outras características da impressão. Se as estimativas de custo e prazos de entrega exceder esses limites, o trabalho não é aceito e o cliente deve ser notificado. Em OCL, um conjunto de pré e pós-condições pode ser especificado da seguinte maneira:

```
context PrintJob::validate(upperCostBound : Integer, custDeliveryReq :
   Integer)
pre: upperCostBound > 0
   and custDeliveryReq > 0
   and self.jobAuthorization = 'no'
post: if self.totalJobCost <= upperCostBound
     and self.deliveryDate <= custDeliveryReq
   then
    self.jobAuthorization = 'yes'
   endif
```

Essa instrução OCL define uma invariante **(inv)** – condições que devem existir antes de (pré) e após (pós) algum comportamento. Inicialmente, uma pré-condição estabelece que um limite de custo e de data de entrega deve ser especificado pelo cliente, e a autorização deve ser definida como "no". Depois que os custos e o prazo de entrega são determinados, é aplicada a pós-condição especificada. Deve-se notar também que para a expressão:

```
self.jobAuthorization = 'yes'
```

não é atribuído o valor "yes", mas está declarando que **jobAuthorization** deve ser definido como "yes" quando a operação terminar. Uma descrição completa da OCL está além do escopo deste apêndice. A especificação OCL completa pode ser obtida no site **www.omg.org/technology/documents/formal/ocl.htm**.

Leituras e Fontes de Informação Complementares

Há dezenas de livros discutindo UML. Aqueles que tratam da última versão incluem: Miles e Hamilton (*Learning UML 2.0*, O'Reilly Media, Inc., 2006); Booch, Rumbaugh e Jacobson (*Unified Modeling Language User Guide*, 2d ed., Addison-Wesley, 2005), Ambler (*The Elements of UML 2.0 Style*, Cambridge University Press, 2005) e Pilone e Pitman (*UML 2.0 in a Nutshell*, O'Reilly Media, Inc., 2005).

Há disponível na Internet uma ampla variedade de fontes de informação sobre o uso da UML na modelagem de engenharia de software. Uma lista atualizada das referências da Web pode ser encontrada em "análise" e "projeto" no site **www.mhhe.com/engcs/compsci/ pressman/professional/olc/ser.htm**.