Esta edição abrange a versão UML 2.0 OMG

3ª Edição

UML Essencial

Um breve guia para a linguagem-padrão de modelagem de objetos

MARTIN FOWLER

Apresentações de **Cris Kobryn, Grady Booch, Ivar Jacobson e Jim Rumbaugh**

bookman®

O AUTOR

Martin Fowler é cientista-chefe da ThoughtWorks, empresa de desenvolvimento de aplicações empresariais. Aplica técnicas orientadas a objeto no desenvolvimento de *software* empresarial há mais de uma década. Muito conhecido por seu trabalho com padrões, UML, refatoração e métodos ágeis, Martin vive em Melorose, Massachusetts, com sua mulher, Cindy, e um gato muito estranho. O endereço de sua página na Web é http://martinfowler.com.

F787u Fowler, Martin
UML essencial [recurso eletrônico] : um breve guia para a linguagem-padrão de modelagem de objetos / Martin Fowler ; tradução João Tortello. – 3. ed. – Dados eletrônicos. – Porto Alegre : Bookman, 2007.

Esta edição abrange a versão UML 2.0 OMG.
Editado também como livro impresso em 2005.
ISBN 978-85-60031-38-2

1. Computação – Linguagem – UML. I. Título.

CDU 004.438UML

Catalogação na publicação: Júlia Angst Coelho – CRB 10/1712

MARTIN FOWLER

Esta edição abrange a versão UML 2.0 OMG

3ª Edição

UML Essencial

Um breve guia para a linguagem-padrão de modelagem de objetos

Tradução:
JOÃO TORTELLO

Consultoria, supervisão e revisão técnica desta edição:
ENG. ANA M. DE ALENCAR PRICE
Doutora em Ciência da Computação pela University of Sussex, Grã-Bretanha
Mestre em Informática pela PUC-RJ
Professora do Instituto de Informática da UFRGS na área de qualidade de *software* e linguagens de programação

Versão impressa desta obra: 2005

bookman®

2007

Obra originalmente publicada sob o título:
UML Distilled: A Brief Guide to the Standard Object Modeling Language, 3/e

ISBN 0-321-19368-7

Tradução autorizada do original em língua inglesa publicado por
Pearson Education, Inc, sob o selo Addison Wesley Professional.

Capa:
Mário Röhnelt

Leitura final:
Fabio Grespan Godinho

Supervisão editorial:
Arysinha Jacques Affonso

Editoração eletrônica:
AGE – Assessoria Gráfica e Editorial Ltda.

Para Cindy

Apresentação da Terceira Edição

Desde os tempos antigos, os mais talentosos arquitetos e projetistas conhecem a lei da parcimônia. Seja ela formulada como um paradoxo ("menos é mais") ou como um mantra ("a mente Zen é a mente do iniciante"), sua sabedoria é eterna: reduzir tudo a sua essência para que a forma harmonize com a função. Das pirâmides ao Opera House de Sydnei, das arquiteturas de von Neumann ao UNIX e Smaltalk, os melhores arquitetos e projetistas têm se esmerado para seguir esse princípio universal e eterno.

Reconhecendo o valor de barbear com a Navalha de Occam[1], quando eu arquiteto e leio, procuro projetos e livros que obedecem a lei da parcimônia. Conseqüentemente, aplaudo o livro que você está lendo agora.

À primeira vista, você pode achar meu último comentário surpreendente. Eu sou freqüentemente associado às especificações volumosas e densas que definem a UML (Unified Modeling Language). Essas especificações permitem que os fornecedores de ferramentas implementem a UML e que os metodologistas a apliquem. Por sete anos, presidi grandes equipes de padronização internacional para especificar a UML 1.1 e a UML 2.0, assim como diversas revisões secundárias intermediárias. Durante esse tempo, a UML amadureceu em expressividade e precisão, mas também a ela foi acrescentada uma complexidade gratuita, como resultado do processo de padronização. Lamentavelmente, os processos de padronização são mais conhecidos pelos compromissos do projeto realizado pelo comitê do que pela elegância parcimoniosa.

O que um especialista em UML familiarizado com as minúcias misteriosas da especificação pode aprender com o refinamento da UML 2.0 feito por Martin? Muito, assim como você. Para começar, Martin reduz habilmente uma linguagem grande e complexa a um subconjunto pragmático que ele tem provado ser eficiente com a prática. Ele tem resistido ao caminho fácil de anexar páginas adicionais à última edição de seu livro. À medida que a linguagem cresce, Martin se mantém fiel ao seu objetivo de procurar "a fração mais útil de UML" e dizer exatamente isso a você. A fração a que ele se refere são os míticos 20% da UML que ajudam você a fazer 80% do seu trabalho. Capturar e domesticar essa fera arisca é uma façanha considerável!

É ainda mais impressionante que Martin atinja esse objetivo enquanto escreve em um estilo coloquial maravilhosamente envolvente. Compartilhando suas opiniões e anedotas conosco, ele torna este livro divertido e nos lembra que definir a arquitetura de sistemas e projetá-los deve ser uma atividade criativa e produtiva. Se buscarmos o mantra

* N. de R.T.: Occam's Razor é um princípio lógico atribuído ao filósofo medieval William of Occam, o qual afirma que não devemos fazer mais suposições do que o mínimo necessário.

da parcimônia a todo custo, acharemos que os projetos modelados com a UML são tão agradáveis quanto aquelas aulas de desenho e de pintura com os dedos na escola primária. A UML deve ser o pára-raios da nossa criatividade, assim como um *laser* para especificar precisamente projetos de sistemas, de modo que outros possam orçar e construir esses sistemas. Este último é a prova dos nove de qualquer linguagem genuína de projeto.

Assim, embora este possa ser um livro pequeno, ele não é trivial. Com a estratégia de modelagem de Martin, você pode aprender tanto quanto aprende com suas explicações sobre a UML 2.0.

Tive o prazer de trabalhar com Martin para melhorar a seleção e a corretude dos recursos da linguagem UML 2.0 explicados nesta revisão. Precisamos ter em mente que todas as linguagens ativas, tanto naturais quanto sintetizadas, devem evoluir ou perecer. As escolhas das novas características feitas por Martin, junto com suas preferências e as de outros profissionais, são uma parte fundamental do processo de revisão da UML. Elas mantêm a linguagem viva e a ajudam a evoluir através da seleção natural no mercado.

Muito trabalho desafiador resta antes que o desenvolvimento orientado por modelos se torne comum, mas sou estimulado por livros como este, que explicam claramente os fundamentos da modelagem com UML e os aplicam pragmaticamente. Espero que você aprenda com ele como eu aprendi e use suas novas idéias para melhorar suas próprias práticas de modelagem de *software*.

CRIS KOBRYN
Presidente, U2 Partner's UML 2.0 Submission Team
Tecnólogo-chefe, Telelogic

Apresentação da Primeira Edição

Quando começamos a moldar a UML (Unified Modeling Language), esperávamos produzir um meio padronizado de expressar projetos que não somente refletisse as melhores práticas do setor, mas que também ajudasse a desmistificar o processo de modelagem de sistemas de *software*. Acreditávamos que a disponibilidade de uma linguagem de modelagem padronizada estimularia mais desenvolvedores a modelar os seus sistemas de *software*, antes de construí-los. A adoção rápida e muito difundida da UML demonstra que os benefícios da modelagem são, de fato, bem conhecidos da comunidade de desenvolvedores de *software*.

A própria criação da UML foi um processo iterativo e incremental, muito semelhante à modelagem de um grande sistema de *software*. O resultado final é um padrão baseado nas muitas idéias e contribuições feitas por diversas pessoas e empresas da comunidade. Nós começamos o trabalho sobre a UML, mas muitos outros contribuíram para uma conclusão de sucesso; somos muito gratos pelas suas contribuições.

Criar e entrar em acordo sobre uma linguagem de modelagem padrão por si só já é um grande desafio. Educar a comunidade de desenvolvedores e apresentar a UML a ela de uma forma acessível e no contexto do processo de desenvolvimento de *software*, também é um grande desafio. Neste livro aparentemente pequeno, atualizado para refletir as mudanças mais recentes na UML, Martin Fowler superou, certamente, esse desafio.

Com um estilo claro e acessível, Martin não somente introduz aspectos principais da UML, mas também demonstra claramente o papel que a UML desempenha no processo de desenvolvimento. Ao longo da leitura, recebemos grandes doses de sabedoria e conhecimento de modelagem, decorrentes dos mais de 12 anos de experiência que o autor acumulou em projeto e em modelagem.

Como resultado, temos um livro que apresenta a linguagem a milhares de desenvolvedores, aguçando seus interesses em explorar melhor os vários benefícios da modelagem utilizando UML, agora um padrão.

Recomendamos o livro para qualquer modelador ou desenvolvedor interessado em conhecê-la e em obter uma visão do importante papel que ela desempenha no processo de desenvolvimento.

GRADY BOOCH
IVAR JACOBSON
JAMES RUMBAUGH

Prefácio

Tenho tido sorte de várias maneiras em minha vida; um de meus maiores golpes de sorte foi estar no lugar certo e com o conhecimento certo para escrever a primeira edição deste livro, em 1997. Naquela época, o mundo caótico da modelagem orientada a objetos (OO) estava apenas começando a ser unificado sob a UML (Unified Modeling Language). Depois disso, a UML se tornou o padrão para a modelagem gráfica de *software*, não apenas para objetos. Minha sorte é que este livro é o mais popular sobre a UML, vendendo mais de 250 mil cópias.

Bem, isso foi ótimo para mim, mas você deve comprar este livro?

Gosto de enfatizar que este é um livro breve. Ele não se destina a dar os detalhes sobre cada faceta da UML, que vem crescendo a cada dia nos últimos anos. Minha intenção é encontrar aquela fração da UML que é mais útil e dizer a você exatamente isso. Embora um livro maior forneça mais detalhes, também demora mais para ser lido. E seu tempo é o maior investimento que você fará em um livro. Mantendo este livro pequeno, gastei meu tempo escolhendo os melhores pontos para evitar que você mesmo tenha que fazer essa seleção. (Infelizmente, ser menor não significa ser proporcionalmente mais barato; há um certo custo fixo para produzir um livro técnico de qualidade.)

Um motivo para ter este livro é começar a aprender sobre a UML. Como este é um livro pequeno, ele fará com que você comece a aprender rapidamente os fundamentos da UML. Com isso, você pode entrar em mais detalhes sobre a UML, com livros maiores, tais como o *User Guide* [Booch, UML user] ou o *Reference Manual* [Rumbaugh, UML Reference].

Este livro também pode atuar como uma referência útil para as partes mais comuns da UML. Embora ele não aborde tudo, é muito mais leve para carregar do que a maioria dos outros livros sobre UML.

Ele também é um livro dogmático. Trabalho com objetos há muito tempo e tenho idéias precisas sobre o que funciona e o que não funciona. Todo livro reflete as opiniões de seu autor, e eu não tento esconder as minhas. Assim, se você estiver procurando algo que tenha um gosto de objetividade, talvez queira escolher outro.

Embora muitas pessoas tenham me dito que este livro é uma boa introdução a objetos, não o escrevi tendo isso em mente. Se você está atrás de uma introdução ao projeto orientado a objetos, sugiro o livro de Craig Larman [Larman].

Muitas pessoas interessadas na UML estão usando ferramentas. Este livro se concentra no padrão e na utilização convencional da UML, e não entra em detalhes do que as várias ferramentas suportam. Embora a UML tenha resolvido a torre de Babel das notações anteriores ao seu aparecimento, muitas diferenças maçantes permanecem entre o que as ferramentas mostram e permitem fazer, ao se desenhar diagramas UML.

Neste livro, não falo muito sobre MDA (Model Driven Architecture). Embora muitas pessoas considerem os dois como sendo a mesma coisa, muitos desenvolvedores utilizam a UML sem estarem interessados na MDA. Se você quiser aprender mais sobre MDA, eu começaria primeiro com este livro, para ter uma visão geral da UML, e depois mudaria para um livro mais específico sobre MDA.

Embora o objetivo principal deste livro seja a UML, incluí também algum material sobre técnicas, como os cartões CRC, que são valiosas para o projeto orientado a objetos. A UML é apenas uma parte do que você precisa para ter sucesso com objetos, e acho que é importante apresentar algumas outras técnicas.

Em um livro breve como este, é impossível entrar nos detalhes sobre como a UML se relaciona com o código-fonte, particularmente porque não existe nenhuma maneira padronizada de fazer essa correspondência. Entretanto, destaco técnicas de codificação comuns para implementar partes da UML. Meus exemplos de código são em Java e C#, pois descobri que essas linguagens são as mais amplamente entendidas. Não presuma que eu prefiro essas linguagens; já trabalhei muito com Smaltalk para dizer isso!

POR QUE UTILIZAR A UML?

As notações gráficas de projeto existem há algum tempo. Para mim, seu principal valor está na comunicação e no entendimento. Um bom diagrama freqüentemente pode ajudar a transmitir idéias sobre um projeto, particularmente quando você quer evitar muitos detalhes. Os diagramas também podem ajudá-lo a entender um sistema de *software* ou um processo do negócio. Como parte de uma equipe tentando descobrir algo, os diagramas ajudam toda a equipe tanto a entender como comunicar esse entendimento. Embora eles não sejam substitutos, pelo menos ainda, para as linguagens de programação textuais, eles são um útil assistente.

Muitas pessoas acreditam que, no futuro, as técnicas gráficas desempenharão um papel preponderante no desenvolvimento de *software*. Sou mais cético do que isso, mas certamente é útil ter uma idéia do que essas notações podem e não podem fazer.

Dessas notações gráficas, a importância da UML é proveniente de seu uso amplo e da padronização dentro da comunidade de desenvolvimento orientado a objetos. A UML se tornou não somente a notação gráfica dominante dentro do mundo orientado a objetos, como também uma técnica popular nos círculos não-orientados a objetos.

ESTRUTURA DO LIVRO

O Capítulo 1 fornece uma introdução à UML: o que ela é, os diferentes significados que ela tem para diferentes pessoas e de onde ela veio.

O Capítulo 2 discute o processo de desenvolvimento de *software*. Embora isso seja rigorosamente independente da UML, acho que é fundamental entender o processo para ver o contexto de algo como a UML. Em particular, é importante entender o papel do desenvolvimento iterativo, que tem sido a estratégia subjacente ao processo para a maioria da comunidade orientada a objetos.

Organizei o restante do livro em torno dos tipos de diagramas da UML. Os Capítulos 3 e 4 discutem as duas partes mais úteis da UML: diagramas de classes (básicos) e diagramas de seqüência. Mesmo que este livro seja fino, acredito que você pode obter o

que há de mais valioso na UML usando as técnicas sobre as quais falo nesses capítulos. A UML é grande e está crescendo, mas você não precisa de tudo que ela tem.

O Capítulo 5 entra nos detalhes sobre as partes menos essenciais, porém ainda úteis, dos diagramas de classes. Os Capítulos 6 a 8 descrevem três diagramas úteis, que esclarecem melhor a *estrutura* de um sistema: diagramas de objetos, diagramas de pacotes e diagramas de distribuição.

Os Capítulos 9 a 11 mostram três técnicas *comportamentais* bastante úteis: casos de uso, diagramas de estados (embora oficialmente conhecidos como diagramas de máquina de estados, geralmente eles são chamados de diagramas de estados) e diagramas de atividades. Os Capítulos 12 a 17 são muito breves e abordam os diagramas que geralmente são menos importantes; portanto, para eles, forneci apenas um rápido exemplo e uma breve explicação.

A contracapa resume as partes mais úteis da notação. Freqüentemente, tenho ouvido as pessoas dizerem que essas capas são a parte mais valiosa do livro. Você provavelmente achará útil se referir a elas quando estiver lendo alguma das outras partes do livro.

MUDANÇAS DA TERCEIRA EDIÇÃO

Se você tem as edições anteriores deste livro, provavelmente está se perguntando o que há de diferente e, o mais importante, se deve adquirir a nova edição.

A principal causa para a terceira edição foi a aparição da UML 2. Ela acrescentou muita coisa nova, incluindo vários tipos novos de diagrama. Até os diagramas já conhecidos têm muita notação nova, como os quadros de interação nos diagramas de seqüência. Se você quiser saber o que aconteceu, mas não deseja percorrer a especificação (eu certamente não recomendo isso!), este livro deve lhe dar uma boa visão geral.

Também aproveitei essa oportunidade para reescrever completamente a maior parte do livro, atualizando o texto e os exemplos. Incorporei grande parte do que aprendi ensinando e usando a UML nos últimos cinco anos. Assim, embora o espírito deste livro ultra-fino sobre UML esteja intacto, a maioria das palavras é nova.

Com o passar dos anos, trabalhei arduamente para manter este livro o mais atualizado possível. À medida que a UML passou por alterações, fiz o melhor que pude para acompanhar o ritmo. Este livro é baseado nos rascunhos da UML 2, que foram aceitos pelo comitê pertinente, em junho de 2003. É improvável que mais mudanças ocorram entre essa votação e outras mais formais; portanto, acho que a UML 2 agora encontra-se estável o suficiente para que minha revisão seja publicada. Vou divulgar informações sobre quaisquer atualizações em minha página na Web (**http://martinfowler.com**).

AGRADECIMENTOS

Durante muitos anos, muitas pessoas fizeram parte do sucesso deste livro. Agradeço primeiramente a Carter Shanklin e Kendall Scott. Carter foi o editor da Addison-Wesley que sugeriu este livro para mim. Kendall Scott me ajudou a fazer as duas primeiras edições, trabalhando no texto e nos desenhos. Eles fizeram o impossível para publicar a primeira edição em um tempo muito curto, enquanto mantinham a alta qualidade que as pessoas esperam da Addison-Wesley. Eles também ficavam fazendo as alterações, durante os primeiros dias da UML, quando nada parecia estável.

Jim Odell foi meu mentor e guia em grande parte do início de minha carreira. Ele também esteve profundamente envolvido com os problemas técnicos e pessoais para fazer metodologistas dogmáticos ajustarem suas diferenças e concordarem com um padrão comum. Sua contribuição para este livro foi profunda e difícil de medir, e aposto que isso também vale para a UML.

A UML é uma criatura de padrões, mas sou alérgico aos organismos de padrões. Assim, para saber o que está havendo, preciso de uma rede de espiões que possa me manter atualizado sobre todas as maquinações dos comitês. Sem esses espiões, incluindo Conrad Bock, Steve Cook, Cris Kobryn, Jim Odell, Guus Ramackers e Jim Rumbaugh, eu estaria arruinado. Todos eles me deram dicas úteis e responderam perguntas estúpidas.

Grady Booch, Ivar Jacobson e Jim Rumbaugh são conhecidos como os Três Amigos. A despeito das brincadeiras que tenho feito nesses anos, eles me deram muito apoio e estímulo para este livro. Nunca esqueçam que minhas brincadeiras normalmente são reflexo de um profundo apreço.

Os revisores são o segredo da qualidade de um livro e aprendi com Carter que você nunca pode ter revisores demais. Os revisores das edições anteriores desde livro foram Simmi Kochhar Bhargava, Grady Booch, Eric Evans, Tom Hadfield, Ivar Jacobson, Ronald E. Jeffries, Joshua Kerievsky, Helen Klein, Jim Odell, Jim Rumbaugh e Vivek Salgar.

A terceira edição também teve um excelente grupo de revisores:

Conrad Bock
Andy Carmichael
Alistair Cockburn
Steve Cook
Luke Hohmann
Pavel Hruby
Jon Kern
Cris Kobryn
Craig Larman
Steve Mellor
Jim Odell
Alan O'Callaghan
Guus Ramackers
Jim Rumbaugh
Tim Seltzer

Todos esses revisores gastaram tempo lendo o manuscrito e cada um deles encontrou pelo menos um erro gritante. Meus sinceros agradecimentos a todos eles. Todos os erros gritantes que permanecem são de minha inteira responsabilidade. Vou divulgar uma folha de errata na seção de livros do *site* **martinfowler.com**, quando eu os encontrar.

A equipe que projetou e escreveu a especificação UML é composta por Don Baisley, Morgan Björkander, Conrad Bock, Steve Cook, Phillipe Desfray, Nathan Dykman, Anders Ek, David Frankel, Eran Gery, Oystein Haugen, Sridhar Iyengar, Cris Kobryn, Birger Moller-Pedersen, James Odell, Gunnar Övergaard, Karin Palmkvist, Guus Ramackers, Jim Rumbaugh, Bran Selic, Thomas Weigert e Larry Williams. Sem eles, eu não teria nada sobre o que escrever.

Pavel Hruby desenvolveu alguns modelos excelentes no Visio, que eu utilizo muito para diagramas UML; você pode obtê-los no endereço **http://phruby.com**.

Muitas pessoas têm entrado em contato comigo, pela Internet e pessoalmente, com sugestões e perguntas, e para apontar erros. Não consigo lembrar de todos, mas meus agradecimentos não são menos sinceros.

Ao pessoal da minha livraria técnica predileta, SoftPro, em Burlington, Massachusetts, EUA, que me deixou passar muitas horas lá, olhando seu estoque para ver como as pessoas usam a UML na prática, e que sempre me serviu um bom café.

Na terceira edição, o editor de aquisições foi Mike Hendrickson. Kim Arney Mulcahy gerenciou o projeto, assim como fez o *layout* e a limpeza dos diagramas. John Fuller, da Addison-Wesley, foi o editor de produção, enquanto Evelyn Pyle e Rebecca Rider ajudaram na edição de cópia e na revisão do livro. Agradeço a todos eles.

Cindy tem ficado comigo, enquanto eu insisto em escrever livros. Então, ela usa o produto no jardim.

Meus pais me deram uma boa educação, a partir da qual tudo brota.

MARTIN FOWLER
Melrose, Massachusetts, EUA
http://martinfowler.com

Sumário

Figuras

Capítulo 1

Introdução

O QUE É UML?

UML (Unified Modeling Language) é uma família de notações gráficas, apoiada por um metamodelo único, que ajuda na descrição e no projeto de sistemas de *software*, particularmente daqueles construídos utilizando o estilo orientado a objetos (OO). Essa definição é um tanto simplificada. Na verdade, para diferentes pessoas a UML tem significados diferentes. Isso ocorre devido à sua própria história e às diferentes maneiras de ver o que compõe um processo de engenharia de *software* eficaz. Como resultado, em grande parte deste capítulo, minha tarefa é armar o cenário para este livro, explicando as diferentes maneiras pelas quais as pessoas vêem e utilizam a UML.

As linguagens gráficas de modelagem existem há muito tempo na indústria do *software*. O propulsor fundamental por trás de todas elas é que as linguagens de programação não estão em um nível de abstração suficientemente alto para facilitar as discussões sobre projeto.

Apesar de as linguagens gráficas de modelagem existirem há muito tempo, há uma enorme controvérsia sobre seu papel na indústria de *software*. Essas controvérsias afetam diretamente o modo como as pessoas percebem o papel da UML em si.

A UML é um padrão relativamente aberto, controlado pelo OMG (Object Management Group), um consórcio aberto de empresas. O OMG foi formado para estabelecer padrões que suportassem interoperabilidade, especificamente a de sistemas orientados a objetos. Talvez, o OMG seja mais conhecido pelos padrões CORBA (Common Object Request Broker Architecture).

A UML nasceu da unificação das muitas linguagens gráficas de modelagem orientadas a objetos que floresceram no final dos anos oitenta, início dos noventa. Desde sua aparição, em 1997, ela fez com que essa torre de Babel fosse resolvida. Trata-se de um serviço pelo qual eu e muitos outros desenvolvedores estamos profundamente agradecidos.

MANEIRAS DE USAR A UML

No centro do papel da UML no desenvolvimento de software estão as diferentes maneiras pelas quais as pessoas querem utilizá-la, diferenças que sobraram de outras linguagens gráficas de modelagem. Essas diferenças levam a argumentos longos e difíceis sobre como a UML deve ser utilizada.

Para desemaranhar isso, Steve Mellor e eu propusemos, de forma independente, uma caracterização dos três modos pelos quais as pessoas utilizam a UML: esboço, projeto e linguagem de programação. De longe, o mais comum dos três, pelo menos de acordo com minha opinião tendenciosa, é utilizar a **UML como esboço**. Nessa utilização, os desenvolvedores usam a UML para ajudar a transmitir alguns aspectos de um sistema. Assim como no caso de projetos, você pode utilizar esboços no desenvolvimento* e na engenharia reversa. No **desenvolvimento**, desenha-se um diagrama UML antes de se escrever o código, enquanto a **engenharia reversa** constrói um diagrama UML a partir de um código já existente, para ajudar em seu entendimento.

A essência dos esboços é a seletividade. No esboço para desenvolvimento, você delineia alguns problemas em código que você está prestes a escrever, normalmente discutindo-os com um grupo de pessoas de sua equipe. Seu objetivo é usar os esboços para ajudar a transmitir as idéias e alternativas sobre o que está prestes a fazer. Você não fala sobre todo o código que vai escrever, mas apenas sobre as questões importantes que quer passar primeiro para seus colegas ou seções do projeto que deseja visualizar antes de iniciar a programação. Sessões como essa podem ser muito curtas: uma sessão de 10 minutos para discutir algumas horas de programação ou um dia para discutir uma iteração de duas semanas.

Na engenharia reversa, você usa esboços para explicar o funcionamento de alguma parte de um sistema. Você não mostra cada classe, mas apenas aquelas que são interessantes e sobre as quais vale a pena falar, antes de se aprofundar no código.

Como os esboços são muito informais e dinâmicos, você precisa fazê-los rapidamente e com colaboração; portanto, uma mídia comum é um quadro branco (*whiteboard*). Os esboços também são úteis em documentos, no caso em que o foco é a comunicação, em vez da perfeição. As ferramentas usadas para fazer esboços são ferramentas de desenho leves, e freqüentemente as pessoas não são muito exigentes a respeito de manter cada regra restrita da UML. A maioria dos diagramas UML mostrada em livros, tais como nos meus próprios, é constituída de esboços. Sua ênfase está na comunicação seletiva, em vez da especificação completa.

Em contraste, a **UML como projeto** tem como foco a completeza. No desenvolvimento, a idéia é de que os projetos são desenvolvidos por um projetista, cujo trabalho é construir um projeto detalhado para um programador codificar. Esse projeto deve ser suficientemente completo, no sentido de que todas as decisões estejam expostas, e o programador deve ser capaz de segui-lo como uma atividade simples e direta, que exija poucas considerações. O projetista pode ser também o programador, mas normalmente é um desenvolvedor mais experiente, que trabalha em uma equipe de programadores. A inspiração para essa estratégia provém de outras formas de engenharia, nas quais engenheiros profissionais criam desenhos de engenharia que são distribuídos para empresas de construção edificarem.

Os desenhos podem ser usados para todos os detalhes ou um projetista pode desenhá-los para uma área em particular. Uma estratégia comum é um projetista desenvolver modelos em nível de projeto, no que diz respeito às interfaces de subsistemas, mas deixando que os desenvolvedores trabalhem nos pormenores da implementação desses detalhes.

Na engenharia reversa, os projetos têm como objetivo transmitir informações detalhadas sobre o código em documentos em papel ou via um navegador gráfico interativo. Os projetos podem mostrar, de forma gráfica, cada detalhe sobre uma classe, que é mais fácil para os desenvolvedores entenderem.

Os projetos precisam de ferramentas muito mais sofisticadas do que os esboços, para manipular os detalhes exigidos para cada tarefa. As ferramentas CASE (engenharia

* N. de R.T.: O autor usou o termo “forward engineering”.

de *software* auxiliada por computador) especializadas caem nessa categoria, embora o termo CASE tenha se tornado um palavrão e agora os vendedores tentem evitá-lo. As ferramentas de desenvolvimento suportam desenhos de diagramas e os armazenam em um repositório para manter as informações. As ferramentas de engenharia reversa lêem o código-fonte, o interpretam a partir do repositório e geram diagramas. Ferramentas de desenvolvimento e de engenharia reversa como essas são referidas como ferramentas de **ida e volta**.

Algumas ferramentas usam o próprio código-fonte como repositório e utilizam diagramas como uma porta de visualização gráfica do código. Essas ferramentas estão muito mais ligadas à programação e freqüentemente se integram diretamente com os editores de programação. Costumo denominar essas ferramentas de **estáticas**.

A linha entre os projetos e os esboços é bastante tênue, mas a distinção, acho eu, repousa no fato de que os esboços são informações deliberadamente incompletas que evidenciam informações importantes, enquanto que os projetos pretendem ser abrangentes, freqüentemente com o objetivo de reduzir a programação a uma atividade simples e completamente mecânica. De forma sucinta, eu diria que os esboços são exploratívos, enquanto os projetos são definitivos.

À medida que você trabalha com a UML e a programação fica cada vez mais mecânica, torna-se evidente que esta deve ser automatizada. Na verdade, muitas ferramentas CASE realizam alguma geração de código, o que automatiza a construção de uma parte significativa de um sistema. Finalmente, você chega em um ponto em que todo o sistema pode ser especificado na UML e, assim, chega à **UML como linguagem de programação**. Nesse ambiente, os desenvolvedores desenham diagramas UML que são compilados diretamente para o código executável e a UML se torna o código-fonte. Obviamente, essa utilização da UML exige ferramentas particularmente sofisticadas. (Além disso, as noções de engenharia direta e reversa não fazem nenhum sentido para esse modo, pois a UML e o código-fonte são a mesma coisa.)

MDA e UML Executável

Quando as pessoas falam sobre a UML, freqüentemente também falam sobre a **MDA** (**Model Driven Architecture**) [Kleppe et. al.]. Basicamente, a MDA é uma estratégia padrão para usar a UML como linguagem de programação; o padrão é controlado pelo OMG, assim como é a UML. Produzindo um ambiente de modelagem de acordo com a MDA, os fornecedores podem criar modelos que também podem trabalhar com outros ambientes compatíveis com a MDA.

Freqüentemente, fala-se simultaneamente da MDA e da UML, pois a primeira utiliza a segunda como linguagem de modelagem básica. Mas, é claro, você não precisa estar usando MDA para utilizar UML.

A MDA divide o trabalho de desenvolvimento em duas áreas principais. Os modeladores representam uma aplicação em particular, por meio da criação de um **PIM** (**Platform Independent Model – modelo independente da plataforma**). O PIM é um modelo da UML independente de qualquer tecnologia específica. Ferramentas podem então transformar o PIM em um **PSM** (**Platform Specific Model – modelo específico de plataforma**). O PIM é um modelo de um sistema destinado a um ambiente de execução específico. Assim, mais ferramentas pegam o PSM e geram código para essa plataforma. O PSM poderia ser feito em UML, mas isso não é obrigatório.

Assim, se você quiser construir um sistema de armazenagem usando MDA, você começará criando um PIM simples de seu sistema de armazenagem. Se desejar que esse sistema de armazenagem seja executado em J2EE e em .NET, deve utilizar as ferramentas de algum fornecedor para criar dois PSMs: um para cada plataforma. Então, mais ferramentas gerarão código para as duas plataformas.

Se o processo de passagem do PIM para o PSM e daí para o código final for completamente automatizado, teremos a UML como linguagem de programação. Se qualquer uma das etapas for manual, teremos os projetos.

Steve Mellor trabalha há bastante tempo nisso e, recentemente, utilizou o termo **UML executável** [Mellor e Balcer]. A UML executável é semelhante à MDA, mas utiliza termos ligeiramente diferentes. Analogamente, você começa com um modelo independente de plataforma que é equivalente ao PIM da MDA. Entretanto, a etapa seguinte consiste em utilizar um compilador de modelos para transformar esse modelo UML em um sistema que possa ser distribuído em um único passo; portanto, não há necessidade do PSM. Conforme o termo *compilador* sugere, essa etapa é completamente automática.

Os compiladores de modelos são baseados em arquétipos reutilizáveis. Um **arquétipo** descreve como pegar um modelo de UML executável e transformá-lo para uma plataforma de programação em particular. Assim, para o exemplo de armazenagem, você compraria um compilador de modelos e dois arquétipos (J2EE e .NET). Execute cada arquétipo em seu modelo de UML executável e você terá suas duas versões do sistema de armazenamento.

A UML executável não usa o padrão UML completo; muitas construções da UML são consideradas desnecessárias e, portanto, não são usadas. Como resultado, a UML executável é mais simples do que a UML completa.

Tudo isso parece bom, mas o quanto é realista? No meu ponto de vista, existem dois problemas aqui. Primeiro, há a questão das ferramentas: elas são maduras o suficiente para o trabalho? Isso é algo que muda com o passar do tempo; certamente, no momento em que eu escrevia isto, elas não eram amplamente utilizadas e não tenho visto muitas delas em ação.

Uma questão mais básica é a noção da UML como linguagem de programação. Em minha opinião, é interessante usar a UML como linguagem de programação apenas se isso resultar em algo significativamente mais produtivo do que utilizar outra linguagem de programação. Não estou convencido de que seja, baseado em vários ambientes gráficos de desenvolvimento com que trabalhei no passado. Mesmo que seja mais produtivo, ainda é necessário obter uma massa crítica de usuários para que seja considerado de uso comum. Só isso já é uma barreira enorme. Assim como muitos usuários antigos de Smalltalk, eu considero esta linguagem muito mais produtiva do que as linguagens de uso comum hoje. Mas, como Smalltalk está agora relegada a segundo plano, não vejo muitos projetos que a utilizam. Para evitar a sina da Smalltalk, a UML precisa ter mais sorte, mesmo sendo superior.

Uma das questões interessantes sobre a UML como linguagem de programação é como modelar lógica comportamental. A UML 2 oferece três espécies de modelagem comportamental: diagramas de interação, diagramas de estado e diagramas de atividade. Todas têm suas propostas de programação. Se a UML ganhar popularidade como linguagem de programação, será interessante ver quais dessas técnicas se tornará bem-sucedida.

Outra maneira pela qual as pessoas vêem a UML é a variação entre utilizá-la para modelagem conceitual e para a modelagem de *software*. A maioria das pessoas está familiarizada com o uso da UML para modelagem de *software*. Nessa **perspectiva de *software***, os elementos da UML são mapeados diretamente nos elementos de um sistema de *software*. Conforme veremos, o mapeamento não é consagrado, mas quando usamos a UML, estamos falando a respeito de elementos de *software*.

Na **perspectiva conceitual**, a UML representa uma descrição dos conceitos de um domínio de estudo. Aqui, não estamos falando a respeito de elementos de *software*, tanto quanto estamos construindo um vocabulário para falarmos sobre um domínio em particular.

Não existem regras rígidas e diretas sobre perspectiva; conforme se verifica, existe uma gama de utilização muito grande. Algumas ferramentas transformam automaticamente código-fonte em diagramas UML, tratando a UML como um modo de visualização alternativo do código-fonte. Isso se parece muito com uma perspectiva de *software*. Se você usar diagramas UML para tentar entender os vários significados do termo fundo de bens com vários contadores, você estará com uma disposição de espírito muito mais conceitual.

Nas edições anteriores deste livro, eu dividi a perspectiva de *software* em especificação (interface) e implementação. Na prática, descobri que era muito difícil traçar uma linha precisa entre as duas; portanto, achei que não era mais interessante fazer essa distinção. Entretanto, em meus diagramas, estou sempre propenso a enfatizar interfaces, em vez de implementação.

Essas diferentes maneiras de usar a UML levam a muitos argumentos sobre o que significam os diagramas UML e qual é sua relação com a realidade. Em particular, isso afeta o relacionamento entre a UML e o código-fonte. Algumas pessoas sustentam que a UML deve ser utilizada para criar um projeto que seja independente da linguagem de programação usada para a implementação. Outras acreditam que o projeto independente de linguagem é como reunir palavras aparentemente contraditórias, com forte ênfase na contradição.

Outra diferença nos pontos de vista é quanto a essência da UML. Eu acho que a maioria dos usuários da UML, particularmente os profissionais que fazem esboços, vê a essência da UML como sendo os diagramas. Entretanto, os criadores da UML vêem os diagramas como secundários; a essência da UML é o metamodelo. Os diagramas são simplesmente uma apresentação do metamodelo. Essa visão também faz sentido para os profissionais que fazem projetos UML e para os usuários de UML como linguagem de programação.

Então, sempre que você ler algo que envolva a UML, é importante entender o ponto de vista do autor. Somente então você poderá entender os argumentos freqüentemente ferozes que a UML estimula.

Dito isso, preciso tornar claras minhas preferências. Quase sempre utilizo a UML para fazer esboços. Acho os esboços UML úteis para desenvolvimento e engenharia reversa, tanto na perspectiva conceitual como na de *software*.

Não sou adepto dos projetos de desenvolvimento detalhados; acredito que é muito difícil fazê-los bem feitos, e eles retardam o trabalho de desenvolvimento. É razoável fazer projetos em nível de interfaces de subsistema, mas mesmo assim, você deve esperar alterações nessas interfaces, a medida que os desenvolvedores implementarem as interações nelas. O valor dos projetos de engenharia reversa é dependente do funcionamento da ferramenta. Se ela for usada como um navegador dinâmico, pode ser muito útil; se ela gerar um documento grande, estará apenas causando devastação.

Como linguagem de programação, eu vejo a UML como uma ótima idéia, mas duvido que tenha uso significativo. Não estou convencido de que as formas gráficas

sejam mais produtivas do que as textuais para a maioria das tarefas de programação e, mesmo que sejam, é muito difícil que uma linguagem seja amplamente aceita.

Como resultado de minhas preferências, este livro focaliza muito mais o uso da UML para fazer esboços. Felizmente, isso faz sentido para um guia breve. Em um livro deste tamanho, não posso fazer justiça à UML em seus outros modos, mas um livro assim é uma boa introdução para outros que o fazem. Portanto, se você estiver interessado nos outros modos da UML, sugiro que trate este livro como uma introdução e consulte outros quando precisar. Se você estiver interessado apenas em esboços, este livro pode ser tudo que você precisa.

COMO CHEGAMOS À UML

Vou admitir que sou aficionado por história. Minha concepção predileta de leitura leve é um bom livro de história. Mas também sei que essa não é a idéia de diversão de todo mundo. Falo sobre história aqui porque acho que, de muitas formas, é difícil compreender onde a UML está, sem entender a história de como ela chegou até aqui.

Na década de oitenta, os objetos começaram a sair dos laboratórios de pesquisa e a dar seus primeiros passos em direção ao mundo "real". Smalltalk estabilizou-se em uma plataforma confiável e surgiu a linguagem C++. Naquela época, várias pessoas começaram a pensar a respeito das linguagens gráficas de projeto orientadas a objetos.

Os livros essenciais sobre linguagens gráficas de modelagem orientadas a objetos surgiram entre 1988 e 1992. Os principais profissionais incluíam Grady Booch [Booch, OOAD]; Peter Coad [Coad, OOA], [Coad, OOD]; Ivar Jacobson (Objectory) [Jacobson, OOSE); Jim Odell [Odell]; Jim Rumbaugh (OMT) [Rumbaugh, idéias], [Rumbaugh, OMT]; Sally Shlaer e Steve Mellor [Shlaer e Steve Mellor, dados], [Shlaer e Steve Mellor, estados]; e Rebecca Wirfs-Brock (Responsibility Driven Design)[Wirfs-Brock].

Cada um desses autores estava, então, liderando informalmente um grupo de profissionais que gostavam daquelas idéias. Todos esses métodos eram muito parecidos, apesar de conterem várias pequenas diferenças entre si. Os mesmos conceitos básicos apareciam em muitas notações diferentes, o que causava confusão para os meus clientes.

Durante aquela época agitada, qualquer conversa sobre padronização era ignorada. Uma equipe do OMG tentou analisar uma padronização, mas recebeu uma carta aberta de protesto de todos os metodologistas importantes. (Isso me lembra uma velha piada: Qual é a diferença entre um metodologista e um terrorista? Resposta: com um terrorista você pode negociar.)

O evento cataclísmico que iniciou a UML se deu quando Jim Rumbaugh deixou a General Electric e se uniu a Grady Brooch, da Rational (agora pertencente à IBM). A aliança entre Booch e Rumbaugh foi vista desde o início como aquela que poderia obter uma massa crítica da fatia de mercado. Grady e Jim proclamaram que "a guerra de métodos acabou – nós vencemos"–, declarando basicamente que eles iriam alcançar a padronização "à maneira da Microsoft". Vários outros metodologistas sugeriram formar uma Coalizão Anti-Booch.

Para a conferência de OOPSLA 95, Grady e Jim haviam preparado a sua primeira descrição pública de seu método unificado: a versão 0.8 da documentação do *Método*

Unificado. Mais importante ainda, eles anunciaram que a Rational Software comprara a Objectory e que, portanto, Ivar Jacobson iria unir-se à equipe. A Rational fez uma festa para comemorar o lançamento do rascunho 0.8, a qual foi muito concorrida. (O destaque da festa foi a primeira aparição pública de Jim Rumbaugh cantando; todos esperamos que também tenha sido a última.)

O ano seguinte viu emergir um processo mais aberto. O OMG, que de modo geral tinha ficado à parte, desempenhava agora um papel ativo. A Rational teve que incorporar as idéias de Ivar e também dispender algum tempo com outros parceiros. O mais significativo foi que o OMG decidiu desempenhar um papel importante.

Neste ponto, é importante compreender porque o OMG se envolveu. Os metodologistas, assim como os autores, gostam de pensar que eles são importantes, mas eu não acho que os gritos dos autores de livros sejam ouvidos pelo OMG. O que fez o OMG se envolver foram os gritos dos fornecedores de ferramentas, todos os quais estavam com medo de que um padrão controlado pela Rational desse uma vantagem competitiva desleal à essa empresa. Como resultado, os fornecedores incitaram o OMG a fazer algo a respeito, sob a bandeira da interoperabilidade da ferramenta CASE. Essa bandeira era importante, pois o OMG estava totalmente voltado à interoperabilidade. A idéia era criar uma UML que permitisse às ferramentas CASE trocar modelos livremente.

Mary Loomis e Jim Odell assumiram a liderança inicial do trabalho. Odell deixou claro que estava preparado para desistir da sua metodologia em favor de um padrão, mas ele não queria um padrão imposto pela Rational. Em janeiro de 1997, várias organizações submeteram propostas para um padrão de métodos a fim de facilitar a troca de modelos. A Rational colaborou com diversas outras organizações e lançou a versão 1.0 da documentação UML como sua proposta, a primeira criatura a responder pelo nome de Unified Modeling Language.

Seguiu-se, então, um pequeno período de quebra de braço, enquanto várias propostas eram unificadas. O OMG adotou a versão 1.1 resultante como um padrão oficial do OMG. Posteriormente, foram feitas algumas revisões. A revisão 1.2 foi somente para melhorar as aparências, mas a 1.3 foi mais significativa. A revisão 1.4 acrescentou vários conceitos detalhados, relativos a componentes e a perfis. A revisão 1.5 adicionou semântica de ação.

Quando as pessoas falam sobre a UML, elas creditam principalmente a Grady Booch, Ivar Jacobson e Jim Rumbaugh como seus criadores. Geralmente, eles são chamados de "os três amigos", embora os piadistas gostem de omitir a primeira sílaba da segunda palavra. Embora eles sejam os mais reconhecidos em relação à UML, acho um tanto desleal dar a eles o crédito principal. A notação UML foi concebida pela primeira vez no método unificado de Booch/Rumbaugh. Desde então, grande parte do trabalho tem sido liderado pelos comitês do OMG. Durante esses estágios posteriores, Jim Rumbaugh foi o único dos três a ter se comprometido de forma intensa. Eu acho que são esses membros do comitê do OMG relativo a UML que merecem o crédito principal por ela.

NOTAÇÕES E METAMODELOS

No seu estado atual, a UML define uma notação e um metamodelo. A **notação** é o material gráfico que você vê nos modelos; ela é a sintaxe gráfica da linguagem de modelagem. Por exemplo, a notação de diagrama de classes define como são representados os itens e conceitos, como classe, associação e multiplicidade.

Certamente, isso leva à questão sobre o que queremos dizer exatamente com uma associação ou multiplicidade ou mesmo uma classe. O uso comum sugere algumas definições informais, mas algumas pessoas querem mais rigor.

A idéia de linguagens rigorosas de especificação e de projeto prevalece mais no campo dos métodos formais. Em tais técnicas, projetos e especificações são representados usando alguns derivativos do cálculo de predicados. Tais definições são matematicamente rigorosas, não permitindo ambigüidade. Entretanto, o valor dessas definições não é de forma alguma universal. Mesmo que você possa provar que um programa satisfaz uma especificação matemática, não há maneira de provar que a especificação matemática satisfaça realmente os requisitos reais do sistema.

A maioria das linguagens gráficas de modelagem tem muito pouco rigor; suas notações apelam para a intuição, em vez de para uma definição formal. Em geral, isso parece não ter causado muito prejuízo. Essas metodologias podem ser informais, mas muitas pessoas ainda as consideram úteis – e é a utilidade que interessa.

No entanto, os metodologistas estão procurando maneiras de melhorar o rigor das metodologias, sem sacrificar sua utilidade. Uma maneira de fazer isso é definir um **metamodelo**: um diagrama, geralmente um diagrama de classes, que define os conceitos da linguagem.

A Figura 1.1 é uma pequena parte do metamodelo UML que mostra o relacionamento entre as características. (Este segmento aparece aí para dar uma idéia do que são os metamodelos. Não vou nem tentar explicá-lo.)

Quanto o metamodelo afeta um usuário da notação de modelagem? A resposta depende principalmente do modo de utilização. Um profissional que trabalhe com esboços normalmente não se preocupa muito; outro que trabalhe com projetos deve se preocupar bem mais. Isso é fundamentalmente importante para aqueles que utilizam a UML como linguagem de programação, pois ela define a sintaxe abstrata dessa linguagem.

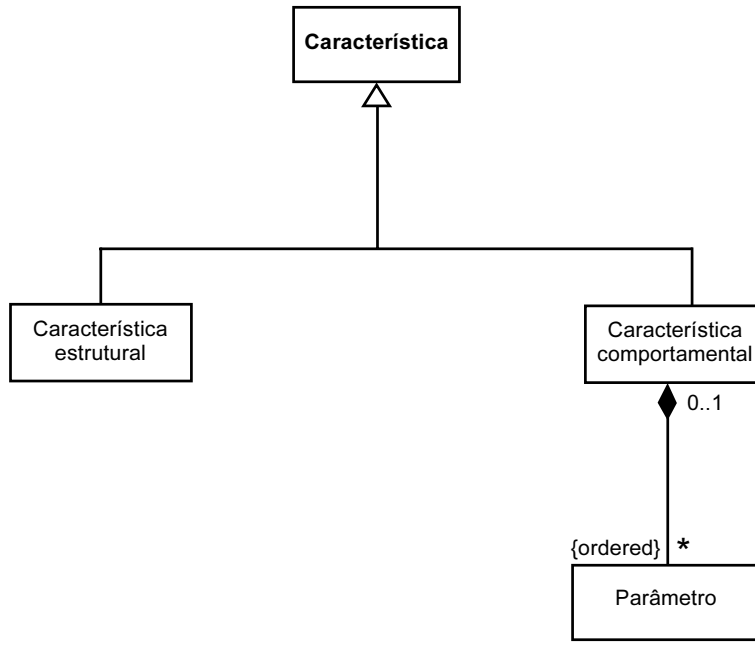


FIGURA 1.1 Uma pequena parte do metamodelo UML.

Muitas das pessoas que estão envolvidas como desenvolvimento em andamento da UML estão interessadas principalmente no metamodelo, particularmente porque isso é importante para a utilização da UML e de uma linguagem de programação. Os problemas de notação freqüentemente ficam em segundo plano, o que é importante lembrar, se você tentar se familiarizar com os documentos padrões propriamente ditos.

À medida que você se aprofunda na utilização mais detalhada da UML, percebe que precisa de muito mais do que a notação gráfica. É por isso que as ferramentas de UML são tão complexas.

Não sou muito rigoroso neste livro. Eu prefiro os camihos dos métodos tradicionais e, de modo geral, apelo para a intuição do leitor. Isso é natural para um livro pequeno como este, escrito por um autor principalmente inclinado a utilização de esboços. Se você quiser um maior rigor, então deve ler livros mais detalhados.

DIAGRAMAS UML

A UML 2 descreve 13 tipos de diagramas oficiais, listados na Tabela 1.1 e classificados conforme indicado na Figura 1.2. Embora esses tipos de diagrama sejam a maneira como muitas pessoas encaram a UML e como eu organizei este livro, os autores da UML não vêem os diagramas como a parte central da UML. Como resultado, os tipos de diagrama não são particularmente rígidos. Freqüentemente, você pode utilizar legalmente elementos de um tipo de diagrama em outro diagrama. O padrão UML indica que certos elementos normalmente são desenhados em determinados tipos de diagrama, mas isso não é uma regra.

TABELA 1.1 Tipos de diagrama oficiais da UML

Diagrama	Capítulos do livro	Objetivo	Linhagem
Atividades	11	Comportamento procedimental e paralelo	Na UML 1
Classes	3,5	Classe, características e relacionamentos	Na UML 1
Comunicação	12	Interação entre objetos; ênfase nas ligações	Diagrama de colaboração da UML 1
Componentes	14	Estrutura e conexão de componentes	Na UML 1
Estruturas compostas	13	Decomposição de uma classe em tempo de execução	Novidade da UML 2
Distribuição	8	Distribuição de artefatos nos nós	Na UML 1
Visão geral da interação	16	Mistura de diagrama de seqüência e de atividades	Novidade da UML 2
Objetos	6	Exemplo de configurações de instâncias	Extra-oficialmente na UML 1
Pacotes	7	Estrutura hierárquica em tempo de compilação	Extra-oficialmente na UML 1
Seqüência	4	Interação entre objetos; ênfase na seqüência	Na UML 1
Máquinas de estado	10	Como os eventos alteram um objeto no decorrer de sua vida	Na UML 1
Sincronismo	17	Interação entre objetos; ênfase no sincronismo	Novidade da UML 2
Casos de uso	9	Como os usuários interagem com um sistema	Na UML 1

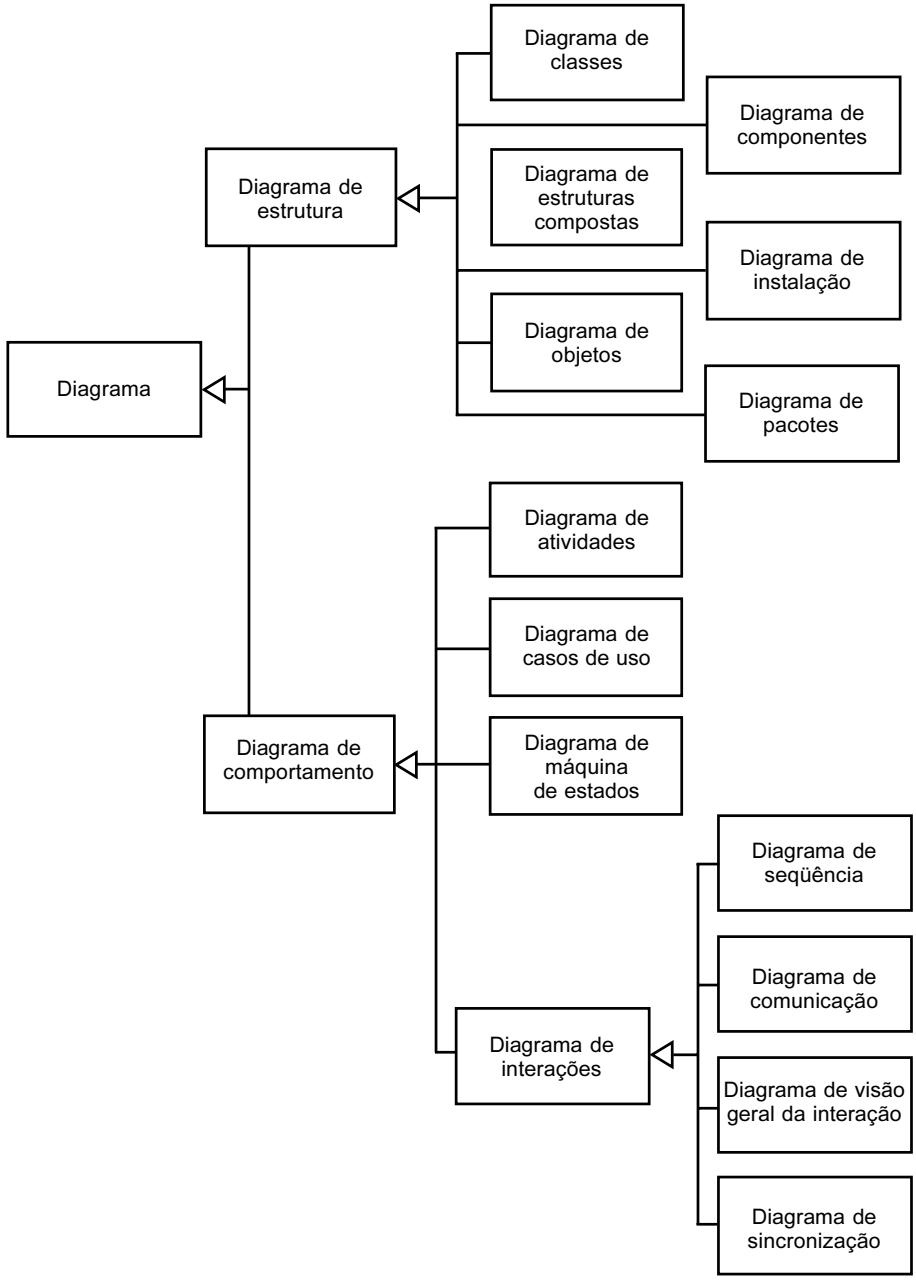


FIGURA 1.2 Classificação dos tipos de diagrama da UML.

O QUE É UML VÁLIDA?

À primeira vista, essa deve ser uma pergunta simples de responder: UML válida é o que é definido como bem-formado na especificação. Na prática, entretanto, a resposta é um pouco mais complicada.

Uma parte importante dessa questão é se a UML possui regras descritivas ou prescritivas. Uma linguagem com **regras prescritivas** é controlada por um organismo oficial que diz o que é ou não válido na linguagem e qual significado você dá às declarações nessa linguagem. Uma linguagem com **regras descritivas** é aquela na qual você entende suas regras examinando como as pessoas utilizam a linguagem na prática. As linguagens de programação tendem a ter regras prescritivas estabelecidas por um comitê formador de padrões ou por um fornecedor dominante, enquanto as linguagens naturais, como o inglês, tendem a ter regras descritivas, cujo significado é estabelecido por convenção.

A UML é uma linguagem bastante precisa; portanto, você pode esperar que ela tenha regras prescritivas. Mas a UML é freqüentemente considerada como sendo o equivalente de *software* aos projetos em outras disciplinas da engenharia, e esses projetos não são notações prescritivas. Nenhum comitê diz quais são os símbolos válidos em um desenho estrutural de engenharia; a notação tem sido aceita por convenção, de forma semelhante a uma linguagem natural. Simplesmente ter um organismo padronizador também não resolve, pois as pessoas do setor podem não seguir todos os padrões definidos por ele; basta perguntar aos franceses sobre a Académie Française. Além disso, a UML é tão complexa que o padrão é freqüentemente aberto a múltiplas interpretações. Mesmo os líderes da UML que revisaram este livro discordariam quanto à interpretação do padrão UML.

Essa questão é importante tanto para mim, que estou escrevendo este livro, quanto para você, que está usando a UML. Se você quiser compreender um diagrama UML, é importante perceber que entender o padrão UML não é tudo. As pessoas adotam convenções, amplamente na indústria e dentro de um projeto particular. Como resultado, embora o padrão possa ser a principal fonte de informação sobre a UML, ele não pode ser o único.

Minha postura é a de que, para a maioria das pessoas, a UML possui regras descritivas. O padrão UML é a maior influência sobre o significado da UML, mas não é o único. Acho que isso se tornará particularmente verdade com a UML 2, que apresenta algumas convenções de notação que entram em conflito com a definição da UML 1 ou com a utilização convencional da UML, assim como acrescenta ainda mais complexidade à UML. Neste livro, portanto, estou tentando resumir a UML conforme eu a percebo: com os padrões e com a utilização convencional. Quando eu tiver que fazer uma distinção neste livro, usarei o termo **uso convencional** para indicar algo que não está no padrão, mas que creio ser amplamente usado. Para algo que esteja de acordo com o padrão, usarei os termos **padrão** ou **normativo**. (Normativo é o termo que os profissionais que trabalham com padrões utilizam para indicar uma declaração que você deve obedecer para ser válida no padrão. Assim, UML não-normativa é uma maneira elegante de dizer que algo é rigorosamente inválido, de acordo com o padrão UML.)

Quando você estiver examinando um diagrama UML, deve se lembrar de que um princípio geral na UML é que qualquer informação pode ser **suprimida** de um diagrama em particular. Essa supressão pode ocorrer geralmente – ocultar todos os atributos – ou especificamente – não exibir essas três classes. Em um diagrama, portanto, você nunca pode inferir algo por sua ausência. Se estiver faltando uma multiplicidade, você não pode inferir qual valor ela poderia ter. Mesmo que o metamodelo UML tenha um padrão estabelecido, como [1] para atributos, se você não vir a informação no diagrama, isso pode ser porque se trata do padrão estabelecido ou porque ele foi suprimido.

Dito isso, existem algumas convenções gerais, como as propriedades que possuem vários valores como conjuntos. Vou apontar essas convenções padronizadas no texto.

É importante não colocar muita ênfase na obtenção de UML válida, caso você seja um profissional que faz esboços ou esquemas. É mais importante ter um bom projeto para seu sistema; e eu preferiria ter um bom projeto com UML inválida, ao invés de uma UML válida, mas com um projeto deficiente. Obviamente, bom e válido é melhor, mas é mais interessante dispender sua energia na obtenção de um bom projeto do que se preocupar com os mistérios da UML. (É claro que você precisa utilizar UML válida como linguagem de programação, senão seu programa não funcionará corretamente!)

O SIGNIFICADO DE UML

Um dos problemas difíceis sobre a UML é que, embora a especificação descreva com bastante detalhe o que é UML bem-formada, ela não tem muito a dizer a respeito do que significa UML fora do mundo refinado dos metamodelos UML. Não existe nenhuma definição formal sobre como a UML é mapeada para qualquer linguagem de programação específica. Você não pode examinar um diagrama UML e dizer *exatamente* como seria o código equivalente. Entretanto, você pode ter uma *idéia aproximada* de como ficaria o código. Na prática, isso é suficiente para ser útil. As equipes de desenvolvimento freqüentemente estabelecem suas convenções locais para isso e você precisará se familiarizar com aquelas que estejam sendo usadas.

UML NÃO É SUFICIENTE

Embora a UML forneça um conjunto considerável de diversos diagramas que ajudam a definir uma aplicação, de modo algum é uma lista completa de todos os diagramas úteis que você poderia querer usar. Em muitos lugares, diferentes diagramas podem ser úteis, e você não deve hesitar em usar um diagrama que não seja feito com UML, se nenhum diagrama da UML atender seu propósito.

A Figura 1.3, um diagrama de fluxo de tela, mostra as várias telas de uma interface com o usuário e como você se move entre elas. Eu tenho visto e utilizado esses diagramas de fluxo de tela há muitos anos. Nunca vi mais do que uma definição muito grosseira do que eles significam; não há nada como isso na UML; apesar disso, acho que são diagramas muito úteis.

A Tabela 1.2 mostra um outro diagrama favorito: a tabela de decisão. As tabelas de decisão são uma boa maneira de mostrar condições lógicas complicadas. Você pode fazer isso com um diagrama de atividades, mas quando vai além dos casos mais simples, a tabela é mais compacta e clara. Novamente, existem muitas formas de tabelas de decisão. A Tabela 1.2 divide-se em duas seções: as condições estão acima da linha dupla e as conseqüências estão abaixo dela. Cada coluna mostra como uma combinação particular de condições leva a um conjunto específico de seqüências.

Você encontrará vários tipos dessas coisas em vários livros. Não hesite em experimentar técnicas que pareçam apropriadas para seu projeto. Se elas funcionarem bem, utilize-as. Caso contrário, descarte-as. (É claro que esse mesmo conselho vale para os diagramas UML.)

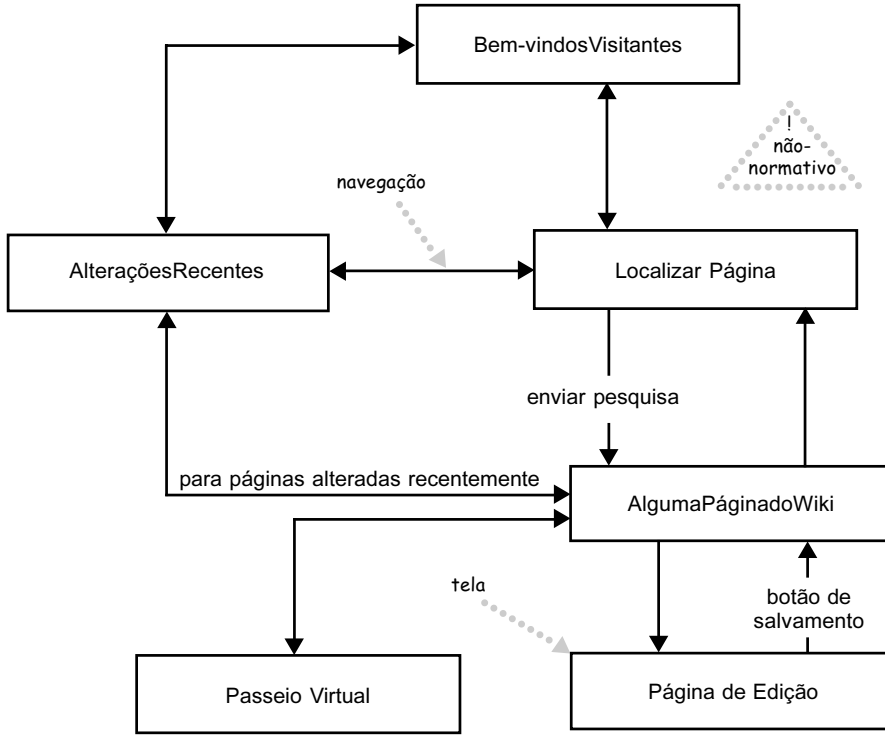


FIGURA 1.3 Um diagrama informal de fluxo de tela para parte do hipertexto Wiki (http://c2.com/cgi/wiki).

TABELA 1.2 Uma tabela de decisão

Cliente de 1ª classe	X	X	Y	Y	N	N
Ordem de prioridade	Y	N	Y	N	Y	N
Pedido internacional	Y	Y	N	N	N	N
Taxa	$150	$100	$70	$50	$80	$60
Relatório de alerta	•	•	•			

ONDE COMEÇAR COM A UML

Ninguém, nem mesmo seus criadores, entende ou utiliza tudo que há na UML. A maioria das pessoas utiliza um pequeno subconjunto da UML e trabalha com isso. Você precisa encontrar o subconjunto da UML que funcione para seu caso e para o de seus colegas.

Se você estiver apenas começando, sugiro que se concentre primeiro nas formas básicas de diagramas de classes e de diagramas de seqüência. Esses são os tipos de diagramas mais comuns e, creio eu, os mais úteis.

Quando você tiver dominado esses diagramas, poderá começar a usar alguma notação de diagrama de classes mais avançada e dar uma olhada nos outros tipos de diagra-

mas. Faça experiências com os diagramas e veja quão úteis eles são para você. Não tenha medo de eliminar tudo que não pareça útil para seu trabalho.

ONDE ENCONTRAR MAIS INFORMAÇÕES

Este livro não é uma referência completa e definitiva para UML, muito menos para análise e projeto orientados a objetos. Há muito material publicado e muitas coisas que valem a pena ser lidas. À medida que eu discutir os tópicos individuais, também mencionarei outros livros aos quais você deve recorrer para obter informações mais aprofundadas. Aqui estão indicados alguns livros gerais sobre a UML e sobre o projeto orientado a objetos.

Assim como em todas as recomendações de livros, talvez você precise verificar para qual versão da UML eles foram escritos. Em junho de 2003, nenhum livro publicado usava UML 2.0, o que não é surpresa, pois o padrão acabava de ser lançado. Os livros que sugiro são muito bons, mas não posso dizer quando eles serão atualizados, se é que serão utilizadas, para o padrão UML 2.

Se você é iniciante na área de objetos, recomendo o meu livro introdutório favorito: [Larman]. A forte estratégia do autor direcionada à responsabilidade para projetos vale a pena ser seguida.

Para um texto conclusivo sobre a UML, você deve ver os documentos do padrão oficial; mas, lembre-se de que eles são escritos para metodologistas que estão de acordo, na privacidade de seus próprios cubículos. Para uma versão muito mais digerível do padrão, dê uma olhada em [Rumbaugh, UML Reference].

Para informações mais detalhadas sobre projeto orientado a objetos, você aprenderá muito com [Martin].

Sugiro também que você leia livros sobre padrões para material que o levará além do conhecimento básico. Agora que a guerra de métodos acabou, é nos padrões de projeto (página 46) que aparece o material mais interessante sobre análise e projeto.

Capítulo 2

Processo de Desenvolvimento

Conforme já mencionei, a UML originou-se a partir de diversas metodologias de análise e projeto orientados a objetos. Até certo ponto, todos eles misturaram uma linguagem gráfica de modelagem com um processo que descrevia como proceder no desenvolvimento de *software*.

É interessante notar que, assim que a UML foi concebida, os vários participantes de sua formação descobriram que, embora pudessem concordar com uma linguagem de modelagem, quase certamente não concordariam com um processo. Como resultado, eles concordaram em deixar qualquer acordo sobre o processo para depois e restringir a UML a uma linguagem de modelagem.

O título deste livro é *UML Essencial*; portanto, eu poderia seguramente ignorar o processo. Entretanto, não acredito que as técnicas de modelagem tenham algum sentido sem que se saiba como elas se encaixam no processo. A maneira como você usa a UML depende muito do estilo de processo utilizado.

Como resultado, acho que é importante falar primeiro sobre processo, para que você possa ver o contexto da utilização da UML. Não vou entrar em muitos detalhes sobre qualquer processo específico; quero simplesmente fornecer informações suficientes sobre esse contexto e indicar onde encontrar mais informações sobre isso.

Quando as pessoas discutem a UML, freqüentemente você as ouve falar a respeito do RUP (Rational Unified Process). O RUP é um processo – ou, mais rigorosamente, uma estrutura de processo – que você pode utilizar com a UML. Mas, a não ser pelo envolvimento comum de várias pessoas da Rational e pelo nome "unified" (unificado), ele não tem nenhum relacionamento especial com a UML. A UML pode ser usada com qualquer processo. O RUP é uma estratégia popular e será discutido mais adiante.

PROCESSOS ITERATIVO E EM CASCATA

Um dos debates mais intensos sobre processo é aquele entre os estilos em cascata e iterativo. Os termos são freqüentemente empregados de forma errada, particularmente porque o processo iterativo é visto como elegante, enquanto que o processo em cascata parece vestir calça xadrez. Como resultado, muitos projetos dizem ser iterativos, mas na verdade são em cascata.

A diferença essencial entre os dois é o modo como você subdivide um projeto em segmentos menores. Se você tiver um projeto que acha que levará um ano, poucas pessoas se sentirão à vontade para dizer à equipe que vá trabalhar por um ano e volte quan-

do o trabalho estiver pronto. Alguma interrupção é necessária, para que as pessoas possam encarar o problema e controlar o andamento.

O estilo **em cascata** subdivide um projeto com base nas atividades. Para construir *software*, você precisa realizar certas atividades: análise dos requisitos, projeto, codificação e teste. Assim, nosso projeto de um ano poderia ter uma fase de análise de dois meses, seguida de uma fase de projeto de quatro meses, após a qual viria uma fase de codificação de três meses, seguida de uma fase de teste de mais três meses.

O estilo **iterativo** subdivide um projeto em subconjuntos de funcionalidade. Você poderia pegar um ano e dividi-lo em iterações de três meses. Na primeira iteração, você pegaria um quarto dos requisitos e faria o ciclo de vida do *software* completo para esse quarto: análise, projeto, código e teste. No final da primeira iteração, você teria um sistema que faria um quarto da funcionalidade necessária. Então, você faria uma segunda iteração tal que, no final de seis meses, tivesse um sistema que fizesse metade da funcionalidade. É claro que o exposto é uma descrição simplificada, mas é a essência da diferença. Na prática, evidentemente, vazam algumas impurezas no processo.

No desenvolvimento em cascata, normalmente existe alguma espécie de transferência formal entre cada fase, mas freqüentemente existem refluxos. Durante a codificação, pode surgir algo que o faça rever a análise e o projeto. Você certamente não deve supor que todo o projeto esteja concluído, quando a codificação começa. É inevitável que as decisões de análise e projeto tenham de ser revistas nas fases posteriores. Entretanto, esses refluxos são exceções e devem ser minimizados o máximo possível.

No caso do estilo iterativo, você normalmente vê alguma forma de atividade exploratória, antes que as iterações reais comecem. No mínimo, isso fornecerá uma visão de alto nível dos requisitos: pelo menos o suficiente para subdividi-los nas iterações que se seguirão. Algumas decisões de projeto de alto nível também podem ocorrer durante a exploração. Por outro lado, embora cada iteração deva gerar *software* integrado pronto para a produção, freqüentemente ela não chega a esse ponto e precisa de um período de estabilização para eliminar os últimos erros. Além disso, algumas atividades, como o treinamento dos usuários, são deixadas para o final.

Você pode não colocar o sistema em produção ao final de cada iteração, mas ele deve ter qualidade de produção. Freqüentemente, entretanto, você pode colocar o sistema em produção em intervalos regulares; isso é bom, porque você avalia o sistema antecipadamente e obtém um retorno de melhor qualidade. Nessa situação, muitas vezes você ouve falar de um projeto com múltiplas **versões**, cada uma das quais subdividida em várias **iterações**.

O desenvolvimento iterativo tem aparecido com muitos nomes: incremental, espiral, evolutivo e jacuzzi, por exemplo. Várias pessoas fazem distinções entre eles, mas não há um acordo generalizado a respeito, nem são tão importantes, comparados à dicotomia do processo iterativo/em cascata.

Você pode ter estratégias híbridas. [McConnell] descreve o ciclo de vida da **entrega por etapas**, por meio da qual a análise e o projeto de alto nível são realizados primeiro, no estilo em cascata, e depois a codificação e o teste são divididos em iterações. Tal projeto poderia ter quatro meses de análise e projeto, seguidos de quatro construções iterativas do sistema, de dois meses cada uma.

A maioria dos autores envolvidos com processos de *software*, especialmente da comunidade orientada a objetos, não gosta da estratégia em cascata. Dos muitos motivos para isso, o mais importante é que, com o processo em cascata, é muito difícil dizer se o projeto está realmente progredindo a contento. É fácil cantar vitória nas primei-

ras fases e ocultar um deslize no planejamento. Normalmente, a única maneira de saber se você está no caminho certo é produzir *software* integrado testado. Fazendo isso repetidamente, um estilo iterativo fornece a você um alerta melhor, caso algo esteja indo mal.

Somente por esse motivo, recomendo veementemente que os projetos não utilizem uma estratégia em cascata pura. Você deve usar pelo menos a entrega por etapas, se não usar uma técnica iterativa mais pura.

Há muito tempo a comunidade orientada a objetos é a favor do desenvolvimento iterativo e é seguro dizer que todos os que estão envolvidos na construção da UML são a favor de pelo menos alguma forma de desenvolvimento iterativo. Entretanto, minha percepção da prática do setor indica que o desenvolvimento em cascata ainda é a estratégia mais comum. Um motivo para isso é o que eu refiro como desenvolvimento pseudo-iterativo: as pessoas dizem que estão fazendo desenvolvimento iterativo, mas na verdade estão usando o processo em cascata. Os sintomas comuns disso são:

- "Estamos fazendo uma iteração de análise seguida de duas iterações de projeto..."
- "O código dessa iteração está repleto de erros, mas vamos limpá-lo no final."

É particularmente importante que cada iteração produza código integrado testado, que seja o mais próximo possível da qualidade de produção. O teste e a integração são as atividades mais difíceis de estimar; portanto, é importante não ter uma atividade aberta como essa no final do projeto. O teste deve ser tal que qualquer iteração que não esteja programada para ser lançada possa sê-lo, sem um trabalho substancial extra de desenvolvimento.

Uma técnica comum no caso das iterações é usar **quadro de tempo**. Isso obriga uma iteração a ocorrer em um período de tempo fixo. Se você achar que não vai conseguir fazer tudo o que pretendia durante uma iteração, deve deslocar alguma funcionalidade dela; você não deve deslocar a data da iteração. A maioria dos projetos que utilizam desenvolvimento iterativo usam o mesmo período de iteração por todo o projeto; desse modo, você consegue um ritmo de construções regular.

Gosto do quadro de tempo, pois as pessoas normalmente têm dificuldade em deslocar funcionalidade. Praticando a função de deslocamento de funcionalidade regularmente, elas ficam em uma melhor posição para fazer uma escolha inteligente, em um grande lançamento, entre deslocar uma data e deslocar uma função. O deslocamento de funções durante as iterações também é eficaz para ajudar as pessoas a aprender quais são as prioridades reais dos requisitos.

Uma das preocupações mais comuns a respeito do desenvolvimento iterativo é a questão de refazer o trabalho. O desenvolvimento iterativo supõe explicitamente que você vai refazer o trabalho e excluir o código já existente, durante as iterações posteriores de um projeto. Em muitos domínios, como na manufatura, refazer o trabalho é visto como desperdício de tempo. Mas o software não é como a manufatura; como resultado, freqüentemente é mais eficaz refazer um código já existente do que corrigir um código mal projetado. Diversas práticas técnicas podem ajudar muito a refazer o trabalho de forma mais eficiente.

- **Testes de regressão automatizados** ajudam, permitindo que você detecte rapidamente quaisquer defeitos que possam ter sido introduzidos na alteração de algo. A família xUnit de estruturas de teste é uma ferramenta particularmente valiosa para a construção de testes de unidade automatizados. Começando com o endereço da JUnit original, **gttp://junit.org**, agora existem portas para praticamente todas as

linguagens imagináveis (veja o endereço **http://www.xprogramming.com/software.htm**). Uma boa regra geral é a de que o tamanho do código de sua unidade em teste deve ser aproximadamente igual ao tamanho do seu código de produção.

- ***Refactoring*** é uma técnica disciplinada para alteração de *software* já existente [Fowler, refactoring]. O *refactoring* trabalha usando uma série de pequenas transformações na base de código, que preservam o comportamento. Muitas dessas transformações podem ser automatizadas (veja o endereço **http://www.refactoring.com**).
- A **integração contínua** mantém uma equipe em sincronismo para evitar ciclos de integração complicados [Fowler e Foemmel]. No centro disso reside um processo de construção totalmente automatizado que pode ser disparado automaticamente, quando qualquer membro da equipe verifica o código na base de código. Espera-se que os desenvolvedores façam verificações diariamente; portanto, as construções automatizadas são feitas muitas vezes por dia. O processo de construção inclui executar um grande grupo de testes de regressão automatizados, tal que quaisquer discrepâncias sejam capturadas rapidamente, e que possam ser corrigidas facilmente.

Todas essas práticas técnicas foram popularizadas recentemente pela Extreme Programming [Beck], embora já tenham sido usadas anteriormente e possam (e devam) ser utilizadas esteja você usando ou não XP ou outro processo ágil.

PLANEJAMENTOS PREDITIVO E ADAPTATIVO

Um motivo pelo qual o processo em cascata resiste é o desejo de previsibilidade no desenvolvimento de *software*. Nada é mais frustrante do que não ter uma idéia clara do quanto custará desenvolver algum *software* e quanto tempo demorará para construí-lo.

Uma estratégia preditiva procura fazer o trabalho antecipadamente no projeto, a fim de gerar um maior entendimento do que precisa ser feito posteriormente. Dessa maneira, você pode chegar a um ponto onde a última parte do projeto pode ser estimada com um grau de precisão razoável. No **planejamento preditivo**, um projeto tem dois estágios. O primeiro estágio sugere planos e é difícil de prever, mas o segundo estágio é muito mais previsível, pois os planos estão estabelecidos.

Isso não é necessariamente um negócio preto-no-branco. À medida que o projeto prossegue, você obtém gradualmente muito mais previsibilidade. Mesmo quando você tem um plano preditivo, as coisas podem dar errado. Você simplesmente espera que os desvios se tornem menos significativos, quando um plano sólido estiver estabelecido.

Entretanto, há uma discussão considerável sobre se muitos projetos de *software* podem ser previsíveis. No centro dessa questão está a análise de requisitos. Uma das únicas fontes de complexidade nos projetos de *software* é a dificuldade de entender os requisitos de um sistema de *software*. A maioria dos projetos de *software* passa por uma **revolução de requisitos** significativa: alterações nos requisitos nos estágios posteriores do projeto. Essas alterações arruinam as bases de um plano preditivo. Você pode combater essas alterações congelando os requisitos antecipadamente e não permitindo mudanças, mas isso acarreta o risco de distribuir um sistema que não atende mais às necessidades de seus usuários.

Esse problema leva a duas reações muito diferentes. Um caminho é elaborar melhor o processo de especificação dos requisitos em si. Dessa maneira, você pode obter um conjunto de requisitos mais preciso, o que reduzirá a revolução.

Outra escola sustenta que a revolução de requisitos é inevitável, que para muitos projetos é difícil demais estabilizar os requisitos suficientemente para utilizar um plano preditivo. Isso pode ser devido à tremenda dificuldade de imaginar o que o *software* pode fazer ou porque as condições do mercado impõem alterações imprevisíveis. Essa escola de pensamento defende o **planejamento adaptativo**, no qual a previsibilidade é vista como uma ilusão. Em vez de nos enganarmos com uma previsibilidade ilusória, devemos encarar a realidade da alteração constante e utilizar uma estratégia de planejamento que trata a alteração como uma constante em um projeto de *software*. Essa alteração é controlada para que o projeto gere o melhor *software* possível; mas, embora o projeto seja controlável, ele não é previsível.

A diferença entre um projeto preditivo e um projeto adaptativo aparece nas muitas maneiras pelas quais as pessoas falam a respeito de como o projeto está se desenvolvendo. Quando as pessoas falam que um projeto vai bem, porque está de acordo com o planejado, essa é uma forma preditiva de pensar. Você não pode dizer "de acordo com o planejado" em um ambiente adaptativo, pois o plano está sempre mudando. Isso não significa que os projetos adaptativos não têm planejamento; normalmente eles têm, e muito, mas o plano é tratado como uma linha de base para avaliar as conseqüências da alteração, em vez de tratá-lo como uma previsão do futuro.

Em um plano preditivo, você pode fechar um contrato a preço fixo e abrangência fixa. Tal contrato diz exatamente o que deve ser construído, quanto custará e quando será entregue. Tal determinação não é possível em um plano adaptativo. Você pode fixar um orçamento e um tempo para entrega, mas não pode estabelecer a funcionalidade que será entregue. Um contrato adaptativo pressupõe que os usuários irão colaborar com a equipe de desenvolvimento para reavaliar regularmente a funcionalidade que precisa ser construída e cancelará o projeto, caso o progresso acabe sendo lento demais. Desse modo, um processo de planejamento adaptativo pode ter preço fixo e abrangência variável.

Naturalmente, a estratégia adaptativa é menos desejável, pois qualquer pessoa preferiria previsibilidade em um projeto de *software*. No entanto, a previsibilidade depende de um conjunto de requisitos preciso, exato e estável. Se você não puder estabilizar seus requisitos, o plano preditivo não terá uma boa base e serão altas as chances de que o projeto saia do curso. Isso leva a duas recomendações importantes.

1. Não faça um plano preditivo até ter requisitos precisos e exatos e até estar confiante de que eles não mudarão significativamente.
2. Se você não puder obter requisitos precisos, exatos e estáveis, utilize um estilo de planejamento adaptativo.

A previsibilidade e a adaptabilidade estabelecem a escolha do ciclo de vida. Um plano adaptativo exige absolutamente um processo iterativo. O planejamento preditivo pode ser feito de qualquer uma das maneiras, embora seja mais fácil ver como ele funciona com uma estratégia de distribuição em cascata ou em estágios.

PROCESSOS ÁGEIS

Nos últimos anos, tem havido muito interesse nos processos ágeis de *software*. *Ágil* é um termo abrangente que compreende muitos processos que compartilham um conjunto

comum de valores e de princípios, conforme definido pelo Manifesto of Agile Software Development (**http://agileManifesto.org**). Exemplos desses processos são Extreme Programming (XP), Scrum, FDD (Feature Driven Development), Crystal e DSDM (Dynamic Systems Development Method).

Nos termos de nossa discussão, os processos ágeis são fortemente adaptativos por natureza. Eles também são processos muito voltados às pessoas. As estratégias ágeis presumem que o fator mais importante no sucesso de um projeto é a qualidade das pessoas que estão envolvidas nele e o quão bem elas trabalham juntas, em termos humanos. Os processos e as ferramentas que elas utilizam são estritamente efeitos de segunda ordem.

As metodologias ágeis tendem a utilizar iterações curtas e com quadro de tempo estabelecido, freqüentemente de um mês ou menos. Como não associam muito peso aos documentos, as estratégias ágeis desprezam o uso da UML no projeto. A maioria utiliza a UML no modo de esboço, com alguns defendendo seu uso como linguagem de programação.

Os processos ágeis tendem a ter pouca **formalidade**. Um processo muito formal ou pesado tem muitos documentos e pontos de controle durante o projeto. Os processos ágeis consideram que a formalidade torna mais difícil fazer alterações e vão contra a natureza das pessoas talentosas. Como resultado, os processos ágeis são freqüentemente caracterizados como **leves**. É importante perceber que a falta de formalidade é uma conseqüência da adaptabilidade e da orientação das pessoas, em vez de ser uma propriedade fundamental.

RATIONAL UNIFIED PROCESS

Embora o RUP (Rational Unified Process) seja independente da UML, freqüentemente os dois são mencionados juntos. Assim, acho que é interessante dizer algumas coisas a respeito dele.

Embora o RUP seja chamado de processo, na verdade trata-se de uma estrutura de processo, fornecendo um vocabulário e uma vaga estrutura para falar sobre processos. Quando você usa o RUP, a primeira coisa que precisa fazer é escolher um **caso de desenvolvimento**: o processo que você vai utilizar no projeto. Os casos de desenvolvimento podem variar bastante; portanto, não ache que seu caso de desenvolvimento será parecido com algum outro. A escolha de um caso de desenvolvimento exige alguém que esteja antecipadamente familiarizado com o RUP: alguém que possa personalizar o RUP para as necessidades de um projeto em particular. Como alternativa, existe um conjunto crescente de casos de desenvolvimento empacotados, para dar partida.

Qualquer que seja o caso de desenvolvimento, o RUP é basicamente um processo iterativo. Um estilo em cascata não é compatível com a filosofia do RUP, embora, infelizmente, não seja incomum encontrar projetos que utilizam um processo de estilo em cascata e se fantasiam de RUP.

Todos os projetos RUP devem seguir quatro fases.

1. A **concepção** faz uma avaliação inicial de um projeto. Normalmente, na concepção, você decide se vai comprometer fundos para realizar uma fase de elaboração.

2. A **elaboração** identifica os casos de uso principais do projeto e elabora o *software* em iterações para reorganizar a arquitetura do sistema. Ao final da elaboração,

você deve ter uma boa idéia dos requisitos e um esqueleto funcional do sistema, que atue como semente de desenvolvimento. Em particular, você deve ter encontrado e resolvido os principais riscos do projeto.

3. A **construção** continua o processo, desenvolvendo funcionalidade suficiente para o lançamento.
4. A **transição** inclui várias atividades de último estágio que você não faz de forma iterativa. Isso pode incluir a distribuição para o centro de dados, treinamento dos usuários e coisas parecidas.

Há muita imprecisão entre as fases, especialmente entre a elaboração e a construção. Para alguns, a mudança para a construção é o ponto em que você pode passar para um modo de planejamento preditivo. Para outros, ela apenas indica o ponto em que você tem uma visão ampla dos requisitos e uma arquitetura que você acha que vai durar pelo resto do projeto.

Às vezes, o RUP é referido como UP (Unified Process – processo unificado). Isso é feito normalmente por organizações que desejam utilizar a terminologia e o estilo global do RUP, sem usar os produtos licenciados da Rational Software. Você pode considerar o RUP como o produto da Rational baseado no UP ou pode considerar o RUP e o UP como idênticos. Seja lá como for, você encontrará pessoas que concordam com sua idéia.

COMO ADEQUAR UM PROCESSO A UM PROJETO

Os projetos de *software* diferem muito uns dos outros. A maneira de proceder em um desenvolvimento de *software* depende de muitos fatores: do tipo de sistema que está sendo construído, da tecnologia utilizada, do tamanho e da distribuição da equipe, da natureza dos riscos, das conseqüências de um fracasso, dos estilos de trabalho da equipe e da cultura da organização. Como resultado, você nunca deve esperar que haja um processo único que funcione para todos os projetos.

Conseqüentemente, você sempre tem de adaptar um processo, de acordo com seu ambiente particular. Uma das primeiras coisas que você precisa fazer é examinar seu projeto e considerar quais processos parecem mais próximos ao desejado. Isso deve fornecer a você uma lista de processos a considerar.

Você deve considerar então quais adaptações precisa fazer para adequá-las ao seu projeto. Você precisa ser muito cuidadoso com isso. Muitos processos são difíceis de avaliar completamente, até que você tenha trabalhado com eles. Nesses casos, freqüentemente é interessante utilizar um processo já pronto para fazer algumas iterações, até você saber como ele funciona. Então, você pode começar a modificar o processo. Se você estiver mais familiarizado com o funcionamento do processo desde o início, poderá modificá-lo já a partir desse ponto. Lembre-se de que normalmente é mais fácil começar com pouco e acrescentar itens do que começar com muitas coisas e jogá-las fora.

Por mais confiante que você esteja com seu processo, é fundamental aprender à medida que avança. Na verdade, uma das maiores vantagens do desenvolvimento iterativo é que ele suporta melhorias freqüentes no processo.

Ao final de cada iteração, realize uma **retrospectiva de iteração**, na qual a equipe se reúne para para considerar como as coisas correram e como elas podem ser aprimoradas.

Padrões

A UML diz como expressar um projeto orientado a objetos. Ao contrário, os padrões focam os resultados do processo: eles oferecem exemplos de projetos.

Muitas pessoas comentam que os projetos têm problemas porque as pessoas envolvidas não estavam inteiradas de projetos bem conhecidos de pessoas mais experientes. Os padrões descrevem maneiras comuns de fazer as coisas e são coletados por pessoas que identificam temas repetitivos em projetos. Essas pessoas examinam cada tema e o descrevem de modo que outras pessoas possam ler o padrão e ver como aplicá-lo.

Vamos ver um exemplo. Digamos que você tenha alguns objetos sendo executados em um processo na sua máquina de trabalho e que eles precisam se comunicar com outros objetos sendo executados em outro processo. Talvez, esse processo também esteja em sua área de trabalho; talvez esteja em outro lugar. Você não quer que os objetos do seu sistema tenham que se preocupar em encontrar outros objetos na rede ou executar chamadas de procedimentos remotos.

O que você pode fazer é criar um objeto *proxy* para o objeto remoto, dentro de seu processo local. O *proxy* tem a mesma interface que o objeto remoto. Os seus objetos locais se se comunicam com o *proxy*, utilizando envios normais de mensagens internas de processo. O *proxy*, então, é responsável por passar todas as mensagens para o objeto real, onde ele estiver.

Os *proxies* são uma técnica comum, utilizada em redes e em outros sistemas. Profissionais têm muita experiência no uso de *proxies*, sabem como podem ser utilizados, quais vantagens podem trazer, suas limitações e como implementá-los. Livros de metodologia como este não discutem esse conhecimento; eles discutem apenas como você pode diagramar um *proxy*, que, apesar de ser útil, não o é tanto quanto a discussão sobre a experiência envolvendo *proxies*.

No início dos anos 90, algumas pessoas começaram a captar essas experiências. Elas formaram uma comunidade interessada em escrever padrões. Essas pessoas patrocinam conferências e têm produzido vários livros.

O livro mais famoso sobre padrões que surgiu desse grupo foi o livro [Gangue dos Quatro], que discute em detalhes 23 padrões de projeto. Se você quiser saber sobre *proxies*, esse livro desenvolve o assunto em dez páginas, dando detalhes sobre como os objetos funcionam em conjunto, benefícios e limitações do padrão, variações comuns e dicas de implementação.

Um padrão é muito mais do que um modelo. Um padrão também deve incluir a razão pela qual ele é o que é. Freqüentemente diz-se que um padrão é uma solução para um problema. Um padrão deve identificar o problema claramente, explicar porque ele resolve o problema e também explicar em quais circunstâncias ele funciona ou não.

Os padrões são importantes porque são o próximo estágio além da compreensão do básico de uma linguagem ou de uma técnica de modelagem. Os padrões fornecem uma série de soluções e também mostram o que faz um bom modelo e como você deve proceder para construir um modelo. Eles ensinam por meio de exemplos.

Quando comecei, eu me perguntava por que tinha de inventar as coisas a partir do nada. Por quê eu não possuía manuais para me mostrar como fazer coisas comuns? A comunidade envolvida com padrões está tentando elaborar esses manuais.

Existem, agora, muitos livros sobre padrões no mercado e sua qualidade varia bastante. Meus prediletos são [Gangue dos Quatro], [POSA1], [POSA2], [Core J2EE Patterns], [Pont] e, modéstia a parte, [Fowler, AP] e [Fowler, P of EAA]. Você também pode dar uma olhada na *homepage* sobre padrões: **http://www.hillsite.net/patterns**.

Duas horas é suficiente, caso suas iterações sejam curtas. Uma boa maneira de realizar isso é fazer uma lista com três categorias:

1. *Manter*: coisas que funcionaram bem, as quais você quer que continuem a funcionar assim.
2. *Problemas*: áreas que não estão funcionando bem.
3. *Tentativa*: alterações em seu processo para aprimorá-lo.

Você pode começar cada retrospectiva de iteração após a primeira, examinando os itens da sessão anterior e vendo como as coisas mudaram. Não se esqueça da lista de coisas a manter; é importante controlar o que está funcionando. Se você não fizer isso, poderá perder a perspectiva do projeto e, possivelmente, parar de prestar atenção nas práticas de sucesso.

Ao final de um projeto ou de um lançamento importante, talvez você queira considerar uma **retrospectiva de projeto** mais formal, que durará dois dias; veja o endereço **http://www.retrospectives.com/** e [Kerth], para obter mais detalhes. Uma de minhas maiores irritações é a respeito de como as organizações constantemente deixam de aprender a partir de suas próprias experiências e acabam cometendo erros dispendiosos repetidamente.

COMO ENCAIXAR A UML EM UM PROCESSO

Ao examinar as linguagens gráficas de modelagem, as pessoas normalmente as consideram no contexto de um processo em cascata. Um processo em cascata normalmente possui documentos que atuam como transferências entre as fases de análise, projeto e codificação. Os modelos gráficos freqüentemente podem formar uma parte importante desses documentos. Na verdade, muitas das metodologias estruturadas dos anos 70 e 80 falam muito sobre modelos de análise e projeto como esse.

Utilizando ou não uma estratégia em cascata, você ainda executa as atividades de análise, projeto, codificação e teste. Você pode executar um projeto iterativo com iterações de uma semana, com cada semana sendo uma mini-cascata.

Usar a UML não implica necessariamente no desenvolvimento de documentos ou na alimentação de uma ferramenta CASE complexa. Muitas pessoas desenham diagramas UML em quadros brancos, somente durante uma reunião, para ajudar a transmitir as suas idéias.

ANÁLISE DE REQUISITOS

A atividade de análise de requisitos procura descobrir o quê os usuários e clientes de um produto de *software* querem que o sistema faça. Várias técnicas de UML são úteis aqui:

- Casos de uso, que descrevem como as pessoas interagem com o sistema.
- Um diagrama de classes desenhado a partir da perspectiva conceitual, o qual pode ser uma boa maneira de construir um vocabulário rigoroso do domínio.
- Um diagrama de atividades, o qual pode exibir o fluxo de trabalho da organização, mostrando como o *software* e as atividades humanas interagem. Um diagra-

ma de atividades pode mostrar o contexto dos casos de uso e também os detalhes sobre como um caso de uso complicado funciona.

- Um diagrama de estados, o qual pode ser útil, caso um conceito tenha um ciclo de vida interessante, com vários estados e eventos que mudam esses estados.

Ao trabalhar na análise de requisitos, lembre-se de que o mais importante é a comunicação com seus usuários e clientes. Normalmente, eles não são profissionais de *software* e não estarão familiarizados com a UML ou qualquer outra técnica. Mesmo assim, eu tive sucesso em usar essas técnicas com pessoal não-técnico. Para fazer isso, lembre-se de que é importante manter a notação em um mínimo. Não introduza nada que seja específico da implementação de *software*.

Esteja preparado para violar as regras da UML a qualquer momento, caso isso o ajude a se comunicar melhor. O maior risco ao utilizar a UML na análise é desenhar diagramas que os especialistas do domínio não entendem completamente. Um diagrama que não é entendido pelas pessoas que conhecem o domínio é simplesmente inútil; ele apenas desenvolve um falso sentido de confiança para a equipe de desenvolvimento.

Projeto

Quando você está fazendo um projeto, pode ter diagramas mais técnicos. Você pode utilizar mais notação e ser mais preciso a respeito dela. Algumas técnicas úteis são:

- Diagramas de classes a partir de uma perspectiva de *software*. Eles mostram as classes presentes no *software* e como elas se relacionam.
- Diagramas de seqüência para cenários comuns. Uma estratégia valiosa é escolher os cenários mais importantes e interessantes dos casos de uso e utilizar cartões CRC ou diagramas de seqüência para descobrir o que acontece no *software*.
- Diagramas de pacotes para mostrar a organização em larga escala do *software*.
- Diagramas de estados para classes com históricos de vida complexos.
- Diagramas de distribuição para mostrar o *layout* físico do *software*.

Muitas dessas mesmas técnicas podem ser usadas para documentar o *software*, quando ele já tiver sido codificado. Isso pode ajudar às pessoas a se orientar no *software*, caso elas tenham que trabalhar nele e não estejam familiarizadas com o código.

Com um ciclo de vida em cascata, você faria esses diagramas e essas atividades como parte das fases. Os documentos de final de fase normalmente incluem os diagramas UML apropriados para essa atividade. Um estilo em cascata normalmente implica que a UML seja usada como projeto.

Em um estilo iterativo, os diagramas UML podem ser utilizados como um projeto ou como um esboço. No caso de um projeto, os diagramas de análise normalmente serão construídos na iteração anterior àquela que cria a funcionalidade. A iteração não começa do nada; em vez disso, ela modifica o texto dos documentos existente, destacando as alterações na nova iteração.

Os desenhos de projeto são normalmente feitos antecipadamente na iteração e podem ser feitos em partes para diferentes funcionalidades destinadas à iteração. Novamente, uma iteração implica em fazer alterações em um modelo já existente, em vez de criar um novo modelo a cada vez.

Usar a UML no modo de esboço implica em um processo mais fluido. Uma estratégia é passar uns dois dias, no início de uma iteração, esboçando o projeto para essa iteração. Você também pode fazer sessões curtas de projeto em qualquer ponto durante a iteração, fazendo uma rápida reunião de meia hora, sempre que um desenvolvedor começar a atacar uma função complicada.

No modo de projeto, você espera que a implementação do código acompanhe os diagramas. Uma alteração no projeto é um desvio que precisa de revisão por parte dos projetistas que o elaboraram. Normalmente, um esboço é tratado mais como um corte de caminho no projeto. Se, durante a codificação, as pessoas acharem que o esboço não está exatamente correto, elas devem se sentir livres para alterar o projeto. Os implementadores precisam utilizar seu julgamento para identificar se a alteração precisa de uma discussão mais ampla para entenderem todas as ramificações.

Uma de minhas preocupações com os projetos é que, mesmo para um bom projetista, é muito mais difícil fazê-los corretamente. Freqüentemente, verifico que meus próprios projetos não sobrevivem intactos ao contato com uma codificação. Ainda considero meus esboços UML úteis, mas não creio que eles possam ser tratados como absolutos.

Nos dois modos, faz sentido explorar várias alternativas de projeto. Normalmente é melhor explorar alternativas no modo de esboço, para que você possa gerar e alterar as alternativas rapidamente. Quando você escolher um projeto para executar, pode usar esse esboço ou detalhá-lo em uma planta de projeto.

Documentação

Quando você tiver construído o *software*, poderá utilizar a UML para ajudá-lo a documentar o que foi feito. Para isso, acho os diagramas UML úteis para se obter um entendimento global de um sistema. Ao se fazer isso, no entanto, devo enfatizar que não acredito na produção de diagramas detalhados do sistema inteiro. Para citar Cunningham [Cunningham]:

> *Memorandos bem escritos e cuidadosamente selecionados podem facilmente substituir uma documentação de projeto abrangente tradicional. Esta última raramente brilha, exceto em pontos isolados. Eleve esses pontos... e esqueça o resto. (p. 384)*

Acredito que uma documentação detalhada deva ser gerada a partir do código – como, por exemplo, JavaDoc. Você deve escrever documentação adicional para salientar conceitos importantes. Pense neles como sendo um primeiro passo para o leitor, antes que ele entre nos detalhes do código. Gosto de estruturá-los como textos curtos o suficiente para serem lidos enquanto tomo um café, utilizando diagramas UML para ajudar a ilustrar a discussão. Eu prefiro os diagramas como esboços que destacam as partes mais importantes do sistema. Obviamente, o escritor do documento precisa decidir o que é importante e o que não é, mas ele está muito melhor equipado do que o leitor para fazer isso.

Um diagrama de pacotes é um bom mapa lógico do sistema. Esse diagrama me ajuda a compreender as partes lógicas do sistema, ver as dependências e mantê-las sob controle. Um diagrama de distribuição (veja o Capítulo 8), que mostra a visão física de alto nível, também pode se mostrar útil neste estágio.

Dentro de cada pacote, gosto de ver um diagrama de classes. Não mostro todas as operações em cada classe. Mostro apenas as características importantes que me ajudam a

entender o que existe lá dentro. Esse diagrama de classes funciona como um sumário gráfico.

O diagrama de classes deve ser suportado por diversos diagramas de interação, que mostrem as interações mais importantes no sistema. Novamente, a seletividade é importante aqui; lembre-se de que, nesse tipo de documento, a abrangência é inimiga da compreensibilidade.

Se uma classe tem um comportamento de ciclo de vida complexo, eu desenho um diagrama de máquina de estados (veja o Capítulo 10) para descrevê-lo. Faço isso somente se o comportamento for suficientemente complexo, o que não acontece com muita freqüência.

Incluirei, freqüentemente, algum código importante, escrito em um estilo erudito de programação. Se um algoritmo particularmente complexo estiver envolvido, eu considero a utilização de um diagrama de atividades (veja o Capítulo 11), mas somente se ele proporcionar maior compreensão do que o próprio código.

Se encontro conceitos que vêm aparecendo repetidamente, uso padrões (página 27) para captar as idéias básicas.

Um dos itens mais importantes a ser documentado são as alternativas de projeto que você não adotou e o por que não as tomou. Essa é, freqüentemente, a documentação externa mais útil, porém mais esquecida, que você pode fornecer.

Como Entender o Código Legado

A UML pode ajudá-lo a entender um emaranhado de códigos desconhecidos, de duas maneiras. A construção de um esboço dos fatos principais pode funcionar como um mecanismo gráfico de anotações, que o ajuda a capturar informações importantes, à medida que você as aprende. Os esboços das classes principais de um pacote e suas interações mais importantes podem ajudar a esclarecer o que está acontecendo.

Com ferramentas modernas, você pode gerar informações detalhadas para as partes fundamentais de um sistema. Não utilize essas ferramentas para gerar grandes relatórios em papel; em vez disso, utilize-as para sondar áreas importantes, quando você estiver explorando o código em si. Um recurso particularmente interessante é a geração de um diagrama de seqüência para ver como os vários objetos colaboram no tratamento de um método complexo.

COMO ESCOLHER UM PROCESSO DE DESENVOLVIMENTO

Sou francamente favorável aos processos de desenvolvimento iterativo. Conforme já mecionei neste livro, você deve utilizar desenvolvimento iterativo somente em projetos em que queira ter sucesso.

Talvez isso não seja muito sério, mas à medida que fico mais velho, me torno mais agressivo quanto à utilização de desenvolvimento iterativo. Bem-feita, trata-se de uma técnica fundamental, que você pode usar para expor cedo os riscos e para obter um melhor controle sobre o desenvolvimento. Isso não é o mesmo que não ter nenhum gerenciamento, embora, para ser justo, devo salientar que algumas pessoas o tenham utilizado dessa maneira. O desenvolvimento iterativo precisa ser bem planejado, mas é um enfoque sólido e todo livro sobre desenvolvimento orientado a objetos estimula a sua utilização – por boas razões.

Você não deve ficar surpreso em ouvir que, como um dos autores do Manifesto for Agile Software Development, sou fã ardoroso das estratégias ágeis. Também tive muitas experiências positivas com a Extreme Programming e você certamente deve levar suas práticas muito a sério.

ONDE ENCONTRAR MAIS INFORMAÇÕES

Livros sobre processo de *software* sempre foram comuns e o surgimento do desenvolvimento ágil de *software* levou a muitas novas publicações. De modo geral, meu livro predileto sobre processos em geral é o [McConnell]. Ele fornece uma cobertura ampla e prática de muitos dos problemas envolvidos no desenvolvimento de *software* e uma longa lista de práticas úteis.

Da comunidade que trabalha com desenvolvimento ágil de *software*, [Cockburn, agile] e [Highsmith] fornecem uma boa visão geral. Para boas orientações sobre a aplicação da UML num processo de maneira ágil, veja [Ambler].

Uma das metodologias ágeis mais populares é a Extreme Programming (XP), na qual você pode se aprofundar por intermédio de páginas da Web como **http://xprogramming.com** e **http://www.extremeprogramming.org**. A XP gerou muitos livros e esse é o motivo pelo qual eu agora me refiro a ela como metodologia anteriormente leve. O ponto de partida usual é [Beck].

Embora seja escrito para XP, [Beck e Fowler] fornece mais detalhes sobre o planejamento de um projeto iterativo. Grande parte disso também é abordada por outros livros sobre XP, mas se você estiver interessado apenas no aspecto do planejamento, essa seria uma boa escolha.

Para obter mais informações sobre o Rational Unified Process, minha introdução predileta é [Kruchten].

Capítulo 3

Diagramas de Classes: Os Elementos Básicos

Se alguém chegar perto de você em um beco escuro e disser: "Psiu, quer ver um diagrama UML?", esse provavelmente seria um diagrama de classes. A maioria dos diagramas UML que vejo é composta por diagramas de classes.

O diagrama de classes não é apenas amplamente usado, mas também está sujeito à maior variação de conceitos de modelagem. Embora os elementos básicos sejam necessários para todo mundo, os conceitos avançados são utilizados com menos freqüência. Portanto, dividi a minha exposição sobre diagramas de classes em duas partes: a básica (este capítulo) e a avançada (Capítulo 5).

Um **diagrama de classes** descreve os tipos de objetos presentes no sistema e os vários tipos de relacionamentos estáticos existentes entre eles. Os diagramas de classes também mostram as propriedades e as operações de uma classe e as restrições que se aplicam à maneira como os objetos estão conectados. A UML utiliza a palavra **característica** como um termo geral que cobre as propriedades e operações de uma classe.

A Figura 3.1 mostra um modelo de classes simples que não surpreenderia ninguém que já tivesse trabalhado com processamento de pedidos. As caixas do diagrama são classes, as quais estão divididas em três compartimentos: o nome da classe (em negrito), seus atributos e suas operações. A Figura 3.1 também mostra dois tipos de relacionamentos entre as classes: associações e generalizações.

Propriedades

As **propriedades** representam as características estruturais de uma classe. Como uma primeira aproximação, você pode considerar as propriedades como correspondentes aos campos de uma classe. A realidade é bem mais complicada, conforme veremos, mas essa é uma consideração razoável, para começar.

As propriedades são um conceito simples, mas elas aparecem em duas notações bastante distintas: atributos e associações. Embora elas pareçam bastante diferentes em um diagrama, na realidade elas são a mesma coisa.

Atributos

A notação de **atributo** descreve uma propriedade como uma linha de texto dentro da caixa de classe em si. A forma completa de um atributo é:

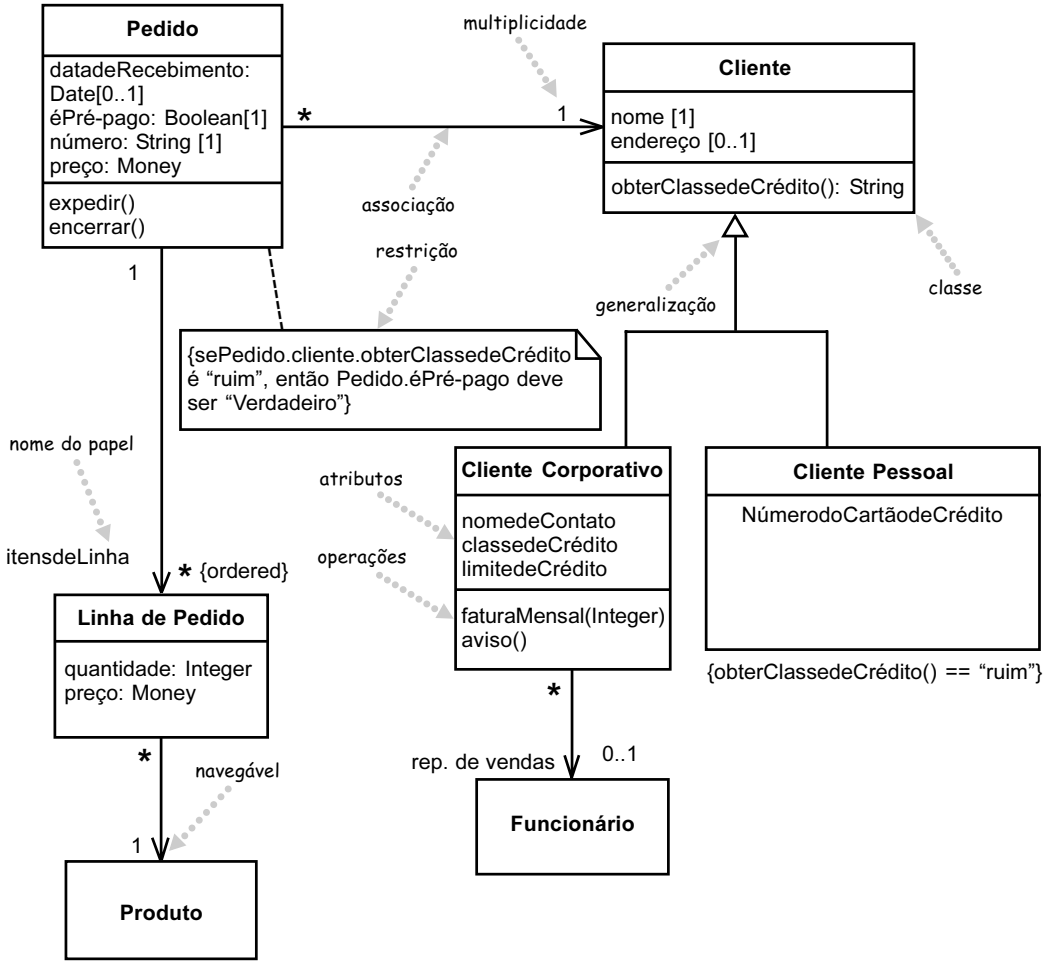


FIGURA 3.1 Um diagrama de classes simples.

```
visibilidade nome: tipo multiplicidade = valor-por-omissão {lista de propriedades}
```

Um exemplo disso aparece a seguir:

```
- nome: String [1] = "Sem título" {readOnly}
```

Somente o `nome` é necessário.

- Esse marcador de `visibilidade` indica se o atributo é público (+) ou privado (-); vamos discutir outras visibilidades na página 92.
- O `nome` do atributo – como a classe se refere ao atributo – corresponde aproximadamente ao nome de um campo em uma linguagem de programação.
- O `tipo` do atributo indica uma restrição sobre o tipo de objeto que pode ser colocado no atributo. Você pode considerar isso como o tipo de um campo em uma linguagem de programação.
- Vou explicar a `multiplicidade` na página 54.

- O `valor-por-omissão` é o valor do objeto novo criado, caso o atributo não seja especificado durante a criação.
- O item `{lista de propriedades}` permite que você indique propriedades adicionais para o atributo. No exemplo, usei `{readOnly}` para indicar que os clientes não podem modificar a propriedade. Se isso estiver ausente, você normalmente pode supor que o atributo é modificável. Vou descrever outras propriedades à medida que prosseguirmos.

Associações

A outra maneira de anotar uma propriedade é como uma associação. Praticamente as mesmas informações que você pode exibir em um atributo aparecem em uma associação. As Figuras 3.2 e 3.3 mostram as mesmas propriedades representadas nas duas notações diferentes.

Uma **associação** é uma linha cheia entre duas classes, direcionada da classe de origem para a classe de destino. O nome da propriedade fica no destino final da associação, junto com sua multiplicidade. O destino final da associação vincula à classe que é o tipo da propriedade.

Embora grande parte das mesmas informações apareça nas duas notações, alguns itens são diferentes. Em particular, as associações podem mostrar multiplicidades nas duas extremidades da linha.

Com duas notações para a mesma coisa, a pergunta óbvia é: por quê você deve usar uma ou a outra? Em geral, eu tendo a utilizar atributos para coisas pequenas, como datas ou valores booleanos – normalmente, tipos de valor (página 73) – e associações para classes mais significativas, como clientes e pedidos. Eu também tendo a preferir o uso de caixas para aquelas classes que são significativas para o diagrama, o que leva ao uso de associações, e atributos para coisas menos importantes desse diagrama. A escolha está muito mais relacionada à ênfase do que a qualquer significado subjacente.

MULTIPLICIDADE

A **multiplicidade** de uma propriedade é uma indicação de quantos objetos podem preencher a propriedade. As multiplicidades que você verá mais comumente são:

- **1** (Um pedido deve ter exatamente um cliente.)
- **0..1** (Um cliente corporativo pode ter ou não um único representante de vendas.)
- * (Um cliente não precisa fazer um Pedido e não existe nenhum limite superior para o número de Pedidos que um Cliente pode fazer – zero ou mais pedidos.)

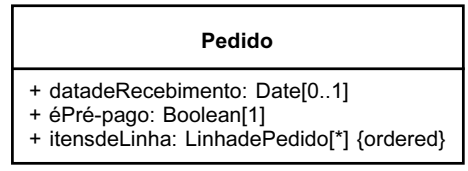


FIGURA 3.2 Mostrando as propriedades de um pedido como atributos.

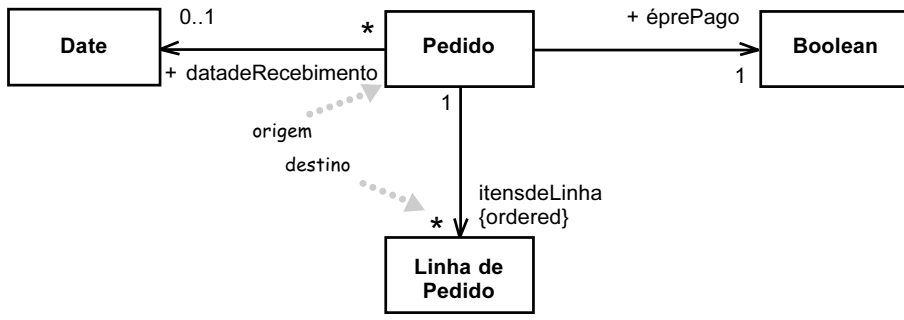


FIGURA 3.3 Mostrando as propriedades de um pedido como associações.

Geralmente, as multiplicidades são definidas com um limite inferior e um limite superior, como 2..4 para jogadores de canastra. O limite inferior pode ser qualquer número positivo ou zero; o limite superior é qualquer número positivo ou * (para ilimitado). Se os limites inferior e superior forem os mesmos, você pode usar um único número; assim, 1 é equivalente a 1..1. Como se trata de um caso comum, * é a abreviatura de 0..*.

Nos atributos, você encontra vários termos que se referem à multiplicidade.

- **Opcional** significa um limite inferior igual a 0.
- **Obrigatório** significa um limite inferior igual a 1 ou possivelmente mais.
- **Valor único** significa um limite superior igual a 1.
- **Valores múltiplos** significa um limite superior maior que 1: normalmente, *.

Se eu tiver uma propriedade de valores múltiplos, prefiro utilizar uma forma plural para seu nome.

Caso não sejam especificados, os elementos em uma multiplicidade de valores múltiplos formam um conjunto; portanto, se você solicitar os pedidos de um cliente, eles não voltarão em qualquer ordem. Se a ordem dos pedidos na associação for significativa, você precisará adicionar `{ordered}` na ponta da associação. Se você quiser permitir a existência de duplicatas, adicione `{nonunique}`. (Se você quiser mostrar o padrão explicitamente, pode usar `{unordered}` e `{unique}`.) Você também poderá ver nomes orientados à coleção, como `{bag}` para não-ordenado e não-exclusivo.

A UML 1 permitia multiplicidades descontínuas, como 2, 4 (significando 2 ou 4, como nos carros, no tempo em que não havia minivans). As multiplicidades descontínuas não eram muito comuns e a UML 2 as eliminou.

A multiplicidade padrão de um atributo é [1]. Embora isso seja verdade no metamodelo, você não pode supor que um atributo em um diagrama que não possui multiplicidade tenha um valor igual a [1], pois o diagrama pode estar suprimindo a informação de multiplicidade. Como resultado, eu prefiro indicar explicitamente uma multiplicidade [1], se isso for importante.

INTERPRETAÇÃO DE PROPRIEDADES EM PROGRAMAS

Assim como acontece com tudo na UML, não existe uma maneira única de interpretar propriedades no código. A representação de *software* mais comum é a de um campo ou

de uma propriedade em sua linguagem de programação. Assim, a classe Linha de Pedido da Figura 3.1 correspondesse a algo como o seguinte em Java:

```
public class LinhadePedido...
  private int quantidade;
  private Dinheiro preço;
  private Pedido pedido;
  private Produto produto;
```

Em uma linguagem como C#, que possui propriedades, ela corresponderia a:

```
public class LinhadePedido...
  private int quantidade;
  private Dinheiro Preço;
  private Pedido Pedido;
  private Produto Produto;
```

Note que um atributo normalmente corresponde às propriedades públicas em uma linguagem que suporta propriedades, mas a campos privados, em um linguagem que não suporta. Em uma linguagem sem propriedades, você poderá ver os campos expostos por intermédio de métodos acessores (de leitura e modificação). Um atributo somente de leitura não terá nenhum método de modificação (com campos) nem ação de conjunto (para propriedades). Note que, se você não der um nome para uma propriedade, é comum utilizar o nome da classe de destino.

A utilização de campos privados é uma interpretação muito focada na implementação do diagrama. Em vez disso, uma interpretação mais voltada à interface poderia se concentrar nos métodos de leitura, em vez de usar os dados subjacentes. Neste caso, poderíamos ver os atributos da Linha de Pedido correspondendo aos seguintes métodos:

```
public class LinhadePedido...
  private int quantidade;
  private Produto produto;
  public int obterQuantidade() {
    return quantidade;
  }
public void configurarQuantidade(int quantidade) {
  this.quantidade = quantidade;
  }
public Dinheiro obterPreço() {
  return produto.obterPreço().multiply(quantidade);
  }
```

Nesse caso, não existe nenhum campo de dados para preço; em vez disso, ele é um valor calculado. Mas, no que diz respeito aos clientes da classe Linha de Pedido, ele é igual a um campo. Os clientes não conseguem diferenciar o que é um campo e o que é calculado. Essa ocultação de informações é a essência do encapsulamento.

Se um atributo tem valores múltiplos, isso significa que os dados envolvidos são uma coleção. Assim, uma classe Pedido faria referência a uma coleção de Linhas de Pedido. Como essa multiplicidade é ordenada, essa coleção deve ser ordenada (como um elemento List em Java ou um elemento IList em .NET). Se a coleção não é ordenada,

rigorosamente ela não deveria ter nenhuma ordem significativa e, assim, ser implementada como um conjunto, mas a maioria das pessoas implementa atributos não-ordenados também como listas. Alguns utilizam *arrays*, mas a UML requer um limite superior ilimitado; portanto, eu quase sempre utilizo uma coleção como estrutura de dados.

As propriedades de valores múltiplos produzem um tipo diferente de interface para propriedades de valor único (em Java):

```
class Pedido {
  private Set itensdeLinha = new HashSet();
  public Set obterItensdeLinha() {
    return Coleções.unmodifiableSet(itensdeLinha);
  }
public void adicionarItemdeLinha(ItemdoPedido arg) {
  itensdeLinha.add (arg);
  }
public void removerItemdeLinha(ItemdoPedido arg) {
  itensdeLinha.remove(arg);
  }
```

Na maioria dos casos, você não faz atribuição a uma propriedade de valores múltiplos; em vez disso, você atualiza com os métodos *add* e *remove*. Para controlar sua propriedade Itens de Linha, o pedido deve controlar a participação como membro dessa coleção; como resultado, ele não deve passar a coleção vazia. Neste caso, eu usei um *proxy* de proteção para fornecer um *wrapper* somente de leitura para a coleção. Você também pode fornecer um iterador não-atualizável ou fazer uma cópia. Os clientes podem modificar os objetos membro, mas não devem alterar diretamente a coleção em si.

Como os atributos de valores múltiplos implicam em coleções, você quase nunca vê classes de coleção em um diagrama de classes. Você os exibiria somente em diagramas de implementação de nível muito baixo, das próprias coleções.

Você deve evitar classes que não são outra coisa a não ser uma coleção de campos e seus métodos acessores. O projeto orientado a objetos tem como tema o fornecimento de objetos que têm um comportamento rico; portanto, eles não devem estar simplesmente fornecendo dados para outros objetos. Se você estiver fazendo chamadas repetidas a dados utilizando acessores, esse é um sinal de que algum comportamento deve ser movido para o objeto que possui os dados.

Esses exemplos também reforçam o fato de que não existe uma correspondência rígida e direta entre a UML e código, apesar de haver uma semelhança. Dentro de uma equipe de projeto, as convenções levarão a uma correspondência mais próxima.

Se uma propriedade é implementada como um campo ou como um valor calculado, isso representa algo que um objeto sempre pode fornecer. Você não deve usar uma propriedade para modelar um relacionamento transitório, como um objeto que é passado como parâmetro durante uma chamada de método e utilizado apenas dentro dos limites dessa interação.

ASSOCIAÇÕES BIDIRECIONAIS

As associações que vimos até aqui são chamadas de unidirecionais. Outro tipo comum de associação é a bidirecional, como mostra a Figura 3.4.

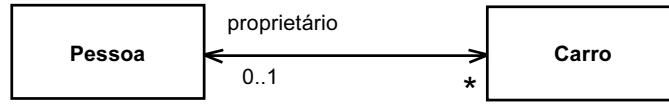


FIGURA 3.4 Uma associação bidirecional.

Uma associação bidirecional é um par de propriedades inversamente vinculadas. A classe Carro tem a propriedade `proprietário:Pessoa[1]` e a classe Pessoa tem uma propriedade `carros:Carro[*]`. (Observe como chamei a propriedade de "carros", na forma plural do tipo da propriedade, uma convenção comum, mas não normativa.)

O vínculo inverso entre elas significa que, se você seguir as duas propriedades, deverá retornar a um conjunto que contém seu ponto de partida. Por exemplo, se eu começar com um MG Midget particular, encontrar seu proprietário e depois ver os carros de seu proprietário, esse conjunto deverá conter o Midget a partir do qual comecei.

Como uma alternativa à denominação de uma associação por meio de uma propriedade, muitas pessoas, particularmente se tiverem experiência em modelagem de dados, gostam de rotular uma associação utilizando um verbo (Figura 3.5) para que o relacionamento possa ser usado em uma frase. Isso é válido e você pode adicionar uma seta na associação para evitar ambiguidade. A maioria dos modeladores de objeto prefere usar um nome de propriedade, pois isso corresponde melhor às responsabilidades e operações.

Algumas pessoas nomeiam cada associação de alguma maneira. Eu prefiro nomear uma associação somente quando isso melhorar o entendimento. Já vi muitas associações com nomes tais como "tem" ou "está relacionado a".

Na Figura 3.4, a natureza bidirecional da associação fica evidente pelas **setas de navegabilidade** nas duas extremidades da associação. A Figura 3.5 não tem setas; a UML permite usar essa forma para indicar uma associação bidirecional ou quando você não está mostrando navegabilidade. Minha preferência é usar a seta de duas pontas da Figura 3.4, quando você quer tornar claro que tem uma associação bidirecional.

Freqüentemente é um pouco complicado implementar uma associação bidirecional em uma linguagem de programação, pois você precisa certificar-se de que as duas propriedades se mantenham sincronizadas. Usando C#, utilizo um código nestes moldes para implementar uma associação bidirecional:

```
class Carro...
  public Pessoa Proprietário {
    get { return _proprietário;}
    set {
      if (_proprietário != null) _proprietário.amigoCarros().Remove(this);
      _proprietário = valor;
      if (_proprietário != null) _proprietário.amigoCarros().Add(this);
    }
  }
private Pessoa_proprietário;
...
```

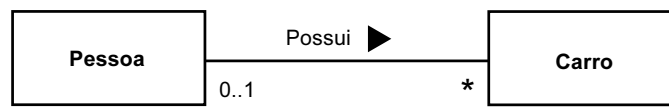


FIGURA 3.5 Usando um verbo para nomear uma associação.

```
class Pessoa...
  public IList Carros {
    get {return ArrayList.ReadOnly(_carros);}
  }
  public void AddCarro(Carro arg) {
    arg.Proprietário = this;
  }
  private IList _carros = new ArrayList();
  internal IList AmigoCarros() {
    //só deve ser usado por Carro.Proprietário
    return _carros;
  }
....
```

O principal é permitir que um lado da associação – um lado de valor único, se possível – controle o relacionamento. Para que isso funcione, o lado escravo (Pessoa) precisa deixar vazar o encapsulamento de seus dados para o lado mestre. Isso acrescenta um método inoportuno na classe escrava, que realmente não deveria estar lá, a não ser que a linguagem tenha um controle de acesso refinado. Eu usei aqui a convenção de atribuição de nomes "amigo" como uma aprovação da linguagem C++, onde o método de modificação do mestre seria mesmo um amigo (friend). Assim como grande parte do código de propriedade, esse material é bastante padronizado, que é o motivo pelo qual muitas pessoas preferem utilizar alguma forma de geração de código para produzi-lo.

Em modelos conceituais, a navegabilidade não é uma questão importante; portanto, não mostro quaisquer setas de navegabilidade nesses modelos.

OPERAÇÕES

Operações são as ações que uma classe sabe realizar. As operações correspondem claramente aos métodos presentes em uma classe. Normalmente, você não mostra as operações que simplesmente manipulam propriedades, porque elas podem ser, na maioria das vezes, inferidas.

A sintaxe completa da UML para as operações é

```
visibilidade nome (lista-de-parâmetros): tipo-de-retorno {lista-de-propriedades}
```

- O marcador `visibilidade` é público (+) ou privado (–); outros na página 92.
- O `nome` é uma seqüência de caracteres.
- A `lista-de-parâmetros` é a lista de parâmetros da operação.
- O `tipo-de-retorno` é o tipo do valor retornado, se houver um.
- A `lista-de-propriedades` indica os valores de propriedade que se aplicam à operação dada.

Os parâmetros da lista de parâmetros são anotados de maneira semelhante aos atributos. A forma é:

```
direção nome: tipo = valor-por-omissão
```

- O `nome`, `o tipo` e `o valor-por-omissão` são iguais aos dos atributos.
- A `direção` indica se o parâmetro é de entrada (`in`), saída (`out`) ou ambos (`inout`). Se nenhuma direção for mostrada, assume-se que ela seja `in`.

Um exemplo de operação de uma conta poderia ser:

```
+ saldoEm (data: Date): Dinheiro
```

Nos modelos conceituais, você não deve usar operações para especificar a interface de uma classe. Em vez disso, utilize-as para indicar as responsabilidades principais de cada classe, empregando, talvez, algumas palavras para resumir responsabilidades no estilo CRC (página 77).

Geralmente, considero útil fazer distinção entre as operações que alteram o estado do sistema daquelas que não alteram. A UML define uma **consulta** como uma operação que obtém um valor de uma classe, sem mudar o estado do sistema – em outras palavras, sem efeitos colaterais. Você pode marcar tal operação com a palavra de propriedade `{query}`. Refiro-me às operações que alteram o estado como **modificadores**, também chamadas de comandos.

Rigorosamente falando, a diferença entre consulta e modificadores é se elas alteram o estado observável [Meyer]. O estado observável é o que pode ser percebido do exterior. Uma operação que atualiza uma memória cache alteraria o estado interno, mas não teria nenhum efeito observável no exterior.

Considero útil destacar as consultas, pois você pode mudar a ordem de execução delas e não alterar o comportamento do sistema. Uma convenção comum é tentar escrever operações de modo que os modificadores não retornem nenhum valor; desse modo, você pode contar com o fato de que as operações que retornam valores são consultas. [Meyer] se refere a isso como o princípio da separação Comando-Consulta. Às vezes é difícil fazer isso o tempo todo, mas você deve fazê-lo o máximo que puder.

Outros termos que você às vezes encontra são métodos de leitura e métodos de modificação. Um **método de leitura** retorna um valor de um campo (e não faz mais nada). Um **método de modificação** coloca um valor em um campo (e não faz mais nada). Do lado externo, um cliente não deve ser capaz de dizer se uma consulta é um método de leitura ou se um modificador é um método de modificação. O conhecimento dos métodos de leitura e de modificação é completamente interno à classe.

Outra distinção é entre operação e método. Uma **operação** é algo que é chamado em um objeto (a declaração de procedimento), enquanto um **método** é o corpo de um procedimento. Os dois são diferentes quando você tem polimorfismo. Se você tem um supertipo com três subtipos, cada um dos quais sobrepondo a operação obterPreço do supertipo, então tem uma operação e quatro métodos que a implementam.

As pessoas usam, normalmente, *operação* e *método* como sinônimos, mas existem ocasiões em que é útil ser preciso sobre a diferença.

GENERALIZAÇÃO

Um exemplo típico de **generalização** é o que envolve clientes e pessoas físicas e jurídicas de uma empresa. Elas têm diferenças mas também muitas semelhanças. As semelhanças podem ser colocadas em uma classe geral Cliente (o supertipo), com Cliente Pessoa Física e Cliente Pessoa Jurídica como subtipos.

Esse fenômeno também está sujeito a várias interpretações em diferentes perspectivas da modelagem. Conceitualmente, podemos dizer que Cliente Pessoa Jurídica é um subtipo de Cliente, se todas as instâncias de Cliente Pessoa Jurídica também o são, por definição, instâncias de Cliente. Um Cliente Pessoa Jurídica é, então, um tipo especial de Cliente. A idéia principal é a de que tudo que dissermos sobre um Cliente – associações, atributos, operações – também é verdadeiro para um Cliente Pessoa Jurídica.

Em uma perspectiva de *software*, a interpretação óbvia é a herança: o Cliente Pessoa Jurídica é uma subclasse de Cliente. Nas principais linguagens orientadas a objetos, a subclasse herda todos os recursos da superclasse e pode sobrepor todos os métodos da superclasse.

Um princípio importante para utilizar a herança eficientemente é a **capacidade de substituição**. Eu devo ser capaz de substituir um Cliente Pessoa Jurídica dentro de qualquer código que exija um Cliente e tudo deve funcionar bem. Basicamente, isso significa que, se você escrever um código supondo que tem um Cliente, então pode usar livremente qualquer subtipo de Cliente. O Cliente Pessoa Jurídica pode responder a certos comandos diferentemente de outro Cliente (usando polimorfismo), mas o chamador não precisa se preocupar com a diferença. (Para mais informações sobre isso, veja o Princípio da Substituição de Liskov (LSP), em [Martin].)

Embora a herança seja um mecanismo poderoso, ela traz muita bagagem que nem sempre é necessária para se obter a capacidade de substituição. Um bom exemplo disso é o que acontecia nos primórdios da linguagem Java, quando muitas pessoas não gostavam da implementação da classe interna Vector e queriam substituí-la por algo mais leve. Entretanto, a única maneira pela qual elas podiam produzir uma classe que fosse substituível por Vector era por meio de sua subtipagem e isso significava herdar muitos dados e comportamento indesejados.

Muitos outros mecanismos podem ser usados para fornecer classes substituíveis. Como resultado, muitas pessoas gostam de fazer distinção entre subtipagem (ou herança de interface) e subclassificação (ou herança de implementação). Uma classe é um **subtipo** se ela é substituível por seu `supertipo`, utilize herança ou não. A **subclassificação** é usada como sinônimo de herança regular.

Existem muitos outros mecanismos que permitem ter subtipos sem subclasses. Exemplos disso são a implementação de uma interface (página 69) e muitos dos padrões de projeto [Gangue dos Quatro].

NOTAS E COMENTÁRIOS

Notas são comentários nos diagramas. As notas podem ser isoladas ou vinculadas, com uma linha tracejada, aos elementos que estão sendo comentados (Figura 3.6). Elas podem aparecer em qualquer tipo de diagrama.

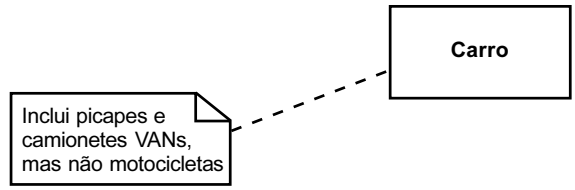


FIGURA 3.6 Uma nota é usada como comentário sobre um ou mais elementos do diagrama.

Às vezes, a linha tracejada pode ser incômoda, pois você não pode localizar exatamente onde essa linha termina. Assim, uma convenção comum é colocar um pequeno círculo aberto no final da linha. Às vezes, é útil ter um comentário em linha em um elemento do diagrama. Você pode fazer isso prefixando o texto com dois traços: – –.

DEPENDÊNCIA

Uma **dependência** entre dois elementos existe se mudanças na definição de um elemento (o **fornecedor**) podem causar mudanças ao outro (o **cliente**). Nas classes, as dependências existem por várias razões: uma classe envia uma mensagem para outra; uma classe tem outra como parte de seus dados; uma classe menciona uma outra como um parâmetro de uma operação. Se uma classe muda a sua interface, qualquer mensagem enviada para essa classe pode não ser mais válida.

À medida que os sistemas de computador crescem, você precisa se preocupar cada vez mais com o controle das dependências. Se as dependências sairem de controle, cada alteração em um sistema terá um amplo efeito de propagação à medida que mais coisas tiverem que mudar. Quanto maior a propagação, mais difícil é a alteração de qualquer coisa.

A UML permite representar dependências entre todos os tipos de elementos. Você usa dependências quando quer mostrar como as mudanças em um elemento poderiam alterar outros elementos.

A Figura 3.7 mostra algumas dependências que você poderia encontrar em um aplicativo de mútiplas camadas. A classe Plano de Benefícios – uma interface com o usuário ou classe de **apresentação** – é dependente da classe Funcionários: um **objeto do domínio** que captura o comportamento essencial do sistema – neste caso, regras do negócio. Isso significa que, se a classe Funcionário mudar sua interface, a classe Plano de Benefícios talvez tenha que mudar.

O importante aqui é que a dependência é apenas em uma direção e vai da classe de apresentação para a classe do domínio. Desse modo, sabemos que podemos alterar livremente a classe Plano de Benefícios sem que essas alterações tenham qualquer efeito sobre a classe Funcionários ou outros objetos do domínio. Descobri que uma separação rigorosa entre a apresentação e a lógica do domínio, com a apresentação dependendo do domínio, mas não o contrário, tem sido uma regra valiosa a ser seguida.

Um segundo detalhe notável nesse diagrama é que não existe nenhuma dependência direta da classe Plano de Benefícios para as duas classes Entrada de Dados. Se essas classes mudarem, a classe Funcionários talvez tenha que mudar. Mas se a alteração for

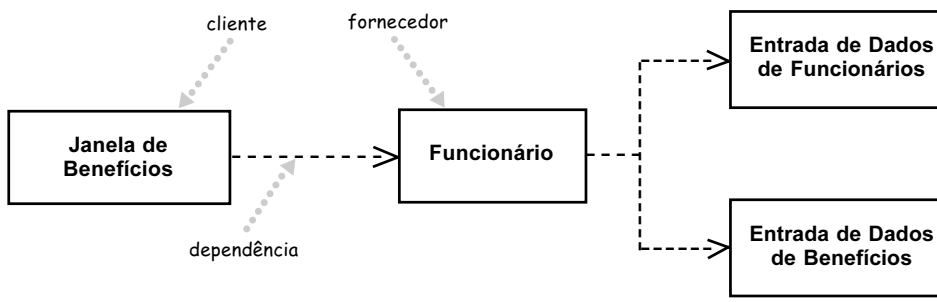


FIGURA 3.7 Exemplos de dependências.

apenas na implementação da classe Funcionário e não em sua interface, a alteração vai parar aí.

A UML tem muitas variedades de dependência, cada uma com semântica e palavras-chave particulares. A dependência básica que destaquei aqui é a que considero mais útil e normalmente a utilizo sem palavras-chave. Para adicionar mais detalhes, você pode acrescentar uma palavra-chave apropriada (Tabela 3.1).

A dependência básica não é um relacionamento transitivo. Um exemplo de relacionamento **transitivo** é o relacionamento "barba maior". Se Jim tem barba maior do que Grady e Grady tem barba maior do que Ivar, podemos deduzir que Jim tem barba maior do que Ivar. Alguns tipos de dependências, como a substituta, são transitivas, mas na maioria dos casos existe uma diferença significativa entre as dependências diretas e indiretas, como se vê na Figura 3.7.

Muitos relacionamentos da UML implicam em uma dependência. A associação navegável de Pedido para Cliente, na Figura 3.1, significa que o Pedido é dependente do Cliente. Uma subclasse é dependente de sua superclasse, mas não o contrário.

Sua regra geral deve ser minimizar as dependências, particularmente quando elas atravessam grandes áreas de um sistema. Em particular, você deve tomar cuidado com os ciclos, pois eles podem levar a um ciclo de alterações. Não sou muito exigente quanto a isso. Não ligo para dependências mútuas entre classes fortemente relacionadas, mas tento eliminar ciclos em um nível mais amplo, particularmente entre pacotes.

Tentar mostrar todas as dependências em um diagrama de classes é uma perda de tempo; existem muitas e elas mudam demais. Seja seletivo e mostre dependências somente quando elas forem diretamente relevantes para o assunto específico que você quer transmitir. Para entender e controlar dependências, é melhor utilizá-las com diagramas de pacotes (página 96).

O caso mais comum em que utilizo dependências com classes é ao ilustrar um relacionamento transitório, como quando um objeto é passado para outro como parâmetro. Você pode ver isso usado com as palavras-chave «`parameter`», «`local`» e «`global`». Você também pode ver essas palavras-chave em associações nos modelos da UML 1,

TABELA 3.1 Palavras-chave de dependência selecionadas

Palavra-chave	Significado
«`call`»	A origem chama uma operação no destino.
«`create`»	A origem cria instâncias do destino.
«`derive`»	A origem é derivada do destino.
«`instantiate`»	A origem é uma instância do destino. (Note que, se a origem é uma classe, a própria classe é uma instância de classe da classe; isto é, a classe de destino é uma metaclasse.)
«`permit`»	O destino permite que a origem acesse seus recursos privados.
«`realize`»	A origem é uma implementação de uma especificação ou interface definida pelo destino (página 80).
«`refine`»	O refinamento indica um relacionamento entre diferentes níveis semânticos; por exemplo, a origem poderia ser uma classe de projeto e o destino, a classe de análise correspondente.
«`substitute`»	A origem é substituível pelo destino (página 61).
«`trace`»	Usada para controlar coisas como requisitos de classes ou como alterações em um modelo se vinculam a alterações em outro lugar.
«`use`»	A origem exige o destino para sua implementação.

no caso em que elas indicam vínculos transitórios e não propriedades. Essas palavras-chave não fazem parte da UML 2.

As dependências podem ser determinadas examinando-se o código; portanto, ferramentas são ideais para se fazer análise de dependência. Conseguir uma ferramenta para fazer engenharia reversa nos relacionamentos de dependência é a maneira mais útil de utilizar essa parte da UML.

REGRAS DE RESTRIÇÃO

Grande parte do que você faz quando desenha diagramas de classes é indicar restrições. A Figura 3.1 indica que um Pedido pode ser feito apenas por um único Cliente. O diagrama também implica que cada Item de Linha é considerado separadamente: no Pedido, você diz "40 elementos de janela marrons, 40 elementos de janela azuis e 40 elementos de janela vermelhos" e não "120 coisas". Além disso, o diagrama diz que Cliente Pessoa Jurídica tem limite de crédito, mas Cliente Pessoa Física, não.

As construções básicas de associação, atributo e generalização fazem muito para especificar restrições importantes, mas elas não conseguem indicar todas as restrições. Essas restrições ainda precisam ser capturadas; o diagrama de classes é um bom lugar para fazê-lo.

A UML permite que você use qualquer coisa para descrever restrições. A única regra é que você as coloque entre chaves ({}). Você pode utilizar linguagem natural, uma linguagem de programação ou a linguagem formal de restrições de objetos de UML (OCL – Object Constraint Language) [Warmer e Kleppe], que é baseada no cálculo de predicados. O uso de uma notação formal evita o risco de uma interpretação errônea devido a uma linguagem natural ambígua. No entanto, ela introduz o risco de interpretação errônea devida ao fato de os escritores e leitores não entenderem realmente a OCL. Portanto, a não ser que você tenha leitores que se sintam à vontade com o cálculo de predicados, sugiro utilizar linguagem natural.

Opcionalmente, você pode nomear uma restrição colocando o primeiro nome, seguido de dois-pontos; por exemplo, {proibir incesto: marido e esposa não devem ser irmãos}.

Projeto por Contrato (*Design by Contract*)

Projeto por Contrato é uma técnica de projeto desenvolvida por Bertrand Meyer [Meyer]. A técnica é uma característica central da linguagem Eiffel desenvolvida por ele. Entretanto, o projeto por contrato não é específico da linguagem Eiffel; é uma técnica valiosa que pode ser utilizada com qualquer linguagem de programação.

No coração do Projeto por Contrato está a asserção. Uma **asserção** é uma declaração booleana que nunca deve ser falsa e, portanto, só será falsa devido a um erro. Normalmente, as asserções são verificadas apenas durante a depuração e não são verificadas durante a execução em produção. De fato, um programa não deve supor que as asserções estão sendo verificadas.

O Projeto por Contrato usa três tipos particulares de asserções: pós-condições, pré-condições e invariantes. As pré-condições e pós-condições se aplicam às operações. Uma **pós-condição** é uma declaração de como o mundo deve parecer depois da execução de uma operação. Por exemplo, se definirmos a operação "raiz quadrada" de um número, a pós-condição assumiria a forma *entrada = resultado * resultado,*

onde *resultado* é a saída e *entrada* é o valor de entrada. A pós-condição é uma forma útil de dizer o que fazemos, sem dizer como o fazemos – em outras palavras, a maneira de separar a interface da implementação.

Uma **pré-condição** é uma declaração de como esperamos que o mundo seja antes de executarmos uma operação. Poderíamos definir uma pré-condição para a operação "raiz quadrada" igual a *entrada* > = *0*. Tal pré-condição diz que é um erro executar "raiz quadrada" de um número negativo e que as conseqüências disso são indefinidas.

Em um primeiro instante, isso parece ser uma má idéia, pois deveríamos colocar alguma verificação em algum lugar, para assegurar que a "raiz quadrada" seja executada adequadamente. A pergunta importante é: quem é responsável por fazer isso?

A pré-condição torna explícito que o chamador é responsável pela verificação. Sem essa declaração de responsabilidades explícita, podemos obter pouca verificação (pois as duas partes presumem que a outra é responsável) ou verificação demais (as duas partes verificam). Verificação demais não é bom, pois ela leva a muito código de verificação duplicado, o que pode aumentar significativamente a complexidade de um programa. Ser explícito a respeito de quem é responsável ajuda a reduzir a complexidade. O perigo do chamador esquecer de verificar é reduzido pelo fato de que as asserções são normalmente verificadas durante a depuração e durante os testes.

A partir dessas definições de pré-condição e pós-condição, podemos ter uma forte definição do termo **exceção**. Uma exceção ocorre quando uma operação é executada com sua pré-condição satisfeita, apesar de não poder retornar com sua pós-condição satisfeita.

Uma **invariante** é uma asserção a respeito de uma classe. Por exemplo, uma classe Conta pode ter uma invariante que diz que *saldo == sum(entradas.valor())*. A invariante é "sempre" verdadeira para todas as instâncias da classe. Aqui, "sempre" significa "quando o objeto estiver disponível para ter uma operação executada nele".

Basicamente, isso significa que a invariante é acrescentada às pré-condições e às pós-condições associadas a todas as operações públicas de determinada classe. A invariante pode se tornar falsa durante a execução de um método, mas deve voltar a ser verdadeira no momento em que outro objeto puder fazer algo no receptor.

As asserções podem desempenhar um papel único na subclassificação. Um dos perigos da herança é que você pode redefinir as operações de uma subclasse, tornando-a inconsistente com as operações da superclasse. As asserções reduzem as chances de se fazer isso. As invariantes e as pós-condições de uma classe devem ser aplicadas a todas as subclasses. As subclasses podem optar por fortalecer essas asserções, mas não podem enfraquecê-las. A pré-condição, por outro lado, não pode ser fortalecida, mas pode ser enfraquecida.

À primeira vista, isso parece estranho, mas é importante para permitir ligação dinâmica. Você sempre deve ser capaz de tratar um objeto de subclasse como se ele fosse uma instância da superclasse (pelo princípio da capacidade de substituição). Se uma subclasse fortalecesse sua pré-condição, então uma operação de superclasse poderia falhar quando aplicada à subclasse.

QUANDO UTILIZAR DIAGRAMAS DE CLASSES

Os diagramas de classes são a espinha dorsal da UML; portanto, você irá utilizá-los o tempo todo. Este capítulo aborda os conceitos básicos; o Capítulo 5 discutirá muitos dos conceitos avançados.

O problema com os diagramas de classes é que eles são tão ricos que podem ser complexos demais para usar. Aqui estão algumas dicas.

- Não tente utilizar todas as notações de que você dispõe. Comece com o material simples deste capítulo: classes, associações, atributos, generalização e restrições. Introduza as outras notações do Capítulo 5 somente quando você necessitá-las.
- Considero os diagramas de classes conceituais muito úteis na exploração da linguagem de negócio. Para que isso funcione, você precisa trabalhar arduamente para manter o *software* fora da discussão e manter a notação muito simples.
- Não desenhe modelos para tudo; em vez disso, concentre-se nas áreas principais. É melhor ter poucos diagramas que você utiliza e os mantém atualizados do que ter muitos modelos esquecidos e obsoletos.

O maior perigo com os diagramas de classes é que você pode focalizar exclusivamente a estrutura e ignorar o comportamento. Portanto, ao desenhar diagramas de classes para entender *software*, sempre faça-os em conjunto com alguma forma de técnica comportamental. Se você estiver indo bem, vai ficar trocando entre as técnicas freqüentemente.

ONDE ENCONTRAR MAIS INFORMAÇÕES

Todos os livros sobre UML que mencionei no Capítulo 1 abordam mais detalhadamente os diagramas de classes. O gerenciamento de dependências é uma característica fundamental de projetos maiores. O melhor livro sobre esse assunto é [Martin].

Capítulo 4

Diagramas de Seqüência

Os **diagramas de interação** descrevem como grupos de objetos colaboram em algum comportamento. A UML define várias formas de diagrama de interação, das quais a mais comum é o diagrama de seqüência.

Normalmente, um diagrama de seqüência captura o comportamento de um único cenário. O diagrama mostra vários exemplos de objetos e mensagens que são passadas entre esses objetos dentro de um caso de uso.

Para começar a discussão, considerarei um cenário simples. Temos um pedido e vamos executar um comando sobre ele para calcular seu preço. Para fazer isso, o pedido precisa examinar todos os itens de linha nele presentes e determinar seus preços, os quais são baseados nas regras de composição de preços dos produtos da linha do pedido. Tendo feito isso para todos os itens de linha, o pedido precisa então calcular um desconto global, que é baseado nas regras vinculadas ao cliente.

A Figura 4.1 é um diagrama de seqüência que mostra uma implementação desse cenário. Os diagramas de seqüência mostram a interação, exibindo cada participante com uma linha de vida, que corre verticalmente na página, e a ordem das mensagens, lendo a página de cima para baixo.

Uma das coisas interessantes a respeito de um diagrama de seqüência é que quase não é preciso explicar a notação. Você pode ver que uma instância do pedido envia mensagens `obterQuantidade` e `obterProduto` para a linha do pedido. Você também pode ver como mostramos o pedido chamando um método dele mesmo e como esse método envia `obterInformaçãodeDesconto` para uma instância de cliente.

O diagrama, entretanto, não mostra tudo muito bem. A seqüência de mensagens `obterQuantidade`, `obterProduto`, `obterDetalhesdoPreço` e `calcularInformaçãodeDesconto` precisa ser feita para cada linha de pedido no pedido, enquanto `calcularDesconto` é chamado apenas uma vez. Você não pode saber isso a partir desse diagrama, embora apresentemos mais alguma notação para tratar disso, posteriormente.

A maior parte das vezes, você pode considerar os participantes de um diagrama de interação como objetos, conforme eles eram, na realidade, na UML 1. Mas, na UML 2, seus papéis são muito mais complicados e, explicar isso completamente está fora dos objetivos deste livro. Assim, uso o termo **participantes**, uma palavra que não é utilizada formalmente na especificação da UML. Na UML 1, os participantes eram objetos e, assim, seus nomes eram sublinhados; mas, na UML 2, eles devem ser mostrados sem o sublinhado, conforme fizemos aqui.

Nesses diagramas, dei nome aos participantes utilizando o estilo `umPedido`. Isso funciona bem na maior parte das vezes. Uma sintaxe mais completa é `nome : Classe`,

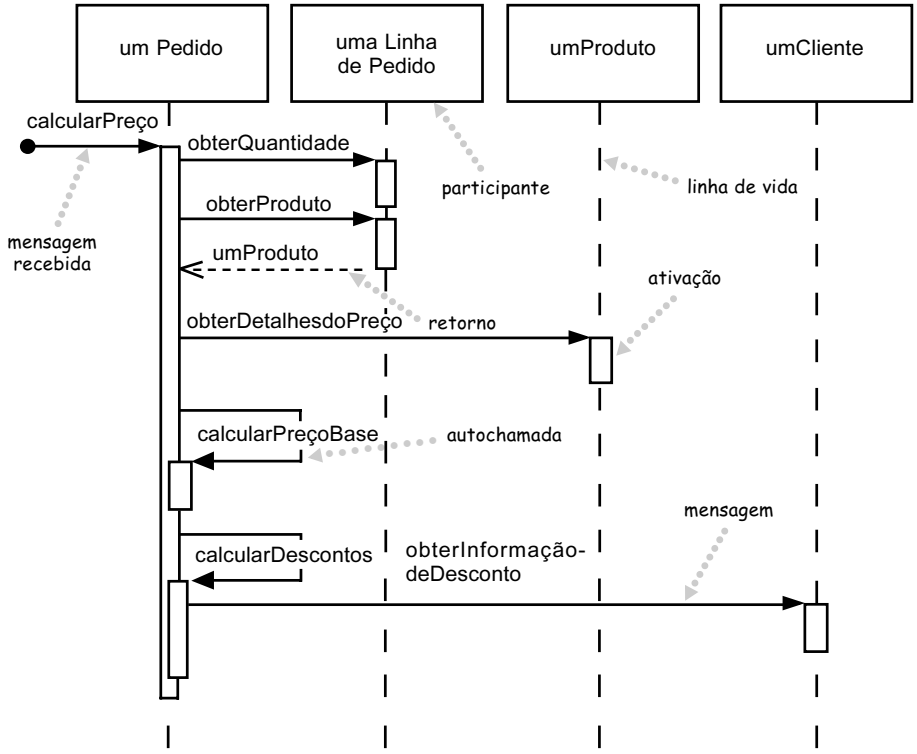


FIGURA 4.1 Um diagrama de seqüência para controle centralizado.

onde o nome e a classe são opcionais, mas você deve manter os dois-pontos, se usar a classe. (A Figura 4.4, mostrada na página 58, usa esse estilo.)

Cada linha de vida tem uma barra de ativação que mostra quando o participante está ativo na interação. Isso corresponde aos métodos de um dos participantes entrando na pilha. As barras de ativação são opcionais na UML, mas as considero extremamente valiosas no esclarecimento do comportamento. A única exceção é ao explorar um projeto durante uma sessão de projeto, pois elas são incômodas de desenhar em quadros brancos.

Freqüentemente, a atribuição de nomes é útil para correlacionar participantes no diagrama. A chamada `obterProduto` aparece retornando `umProduto`, o qual tem o mesmo nome e, portanto, é o mesmo participante, que o objeto `umProduto` para o qual a chamada de `obterDetalhesdoPreço` é enviada. Note que usei uma seta de retorno apenas para essa chamada; fiz isso para mostrar a correspondência. Algumas pessoas utilizam retornos para todas as chamadas, mas prefiro usá-los somente onde eles acrescentam informações; caso contrário, eles apenas congestionam o diagrama. Mesmo nesse caso, você provavelmente poderia omitir o retorno, sem confundir seu leitor.

A primeira mensagem não tem um participante que a enviou, pois ela é proveniente de uma fonte indeterminada. Ela é chamada de **mensagem recebida**.

Para outra abordagem desse cenário, dê uma olhada na Figura 4.2. O problema básico ainda é o mesmo, mas a maneira como os participantes colaboram para implementá-lo é muito diferente. O Pedido pede para que cada Linha de Pedido calcule seu próprio Preço. A própria Linha de Pedido transmite o cálculo para o Produto; observe como mostramos a passagem de um parâmetro. Analogamente, para calcular o descon-

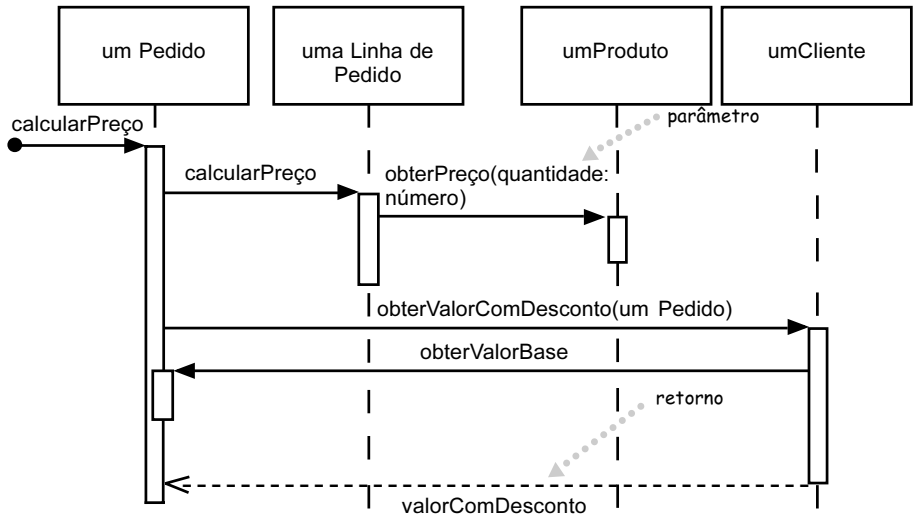


FIGURA 4.2 Um diagrama de seqüência para controle distribuído.

to, o Pedido chama um método do Cliente. Como ele precisa de informações do Pedido para fazer isso, o Cliente faz uma chamada reentrante (`obterValorBase`) no Pedido para obter os dados.

O primeiro detalhe a ser notado a respeito desses dois diagramas é a clareza com que o diagrama de seqüência indica as diferenças na interação dos participantes. Essa é a grande vantagem dos diagramas de interação. Eles não são bons para mostrar detalhes dos algoritmos, como laços e comportamento condicional, mas tornam as chamadas entre os participantes cristalinas e fornecem uma visão excepcional a respeito de quais participantes estão realizando quais processamentos.

O segundo detalhe a ser notado é a clara diferença nos estilos entre as duas interações. A Figura 4.1 é um **controle centralizado**, com um único participante realizando todo o processamento e com os outros participantes lá para fornecer dados. A Figura 4.2 utiliza **controle distribuído**, no qual o processamento é dividido entre muitos participantes, cada um executando um pequeno trecho do algoritmo.

Os dois estilos têm suas forças e suas fraquezas. A maioria das pessoas, particularmente aquelas iniciantes na tecnologia de objetos, está mais acostumada com o controle centralizado. De muitas formas, ele é mais simples, pois todo o processamento é feito em um só lugar; em comparação, no controle distribuído, você tem a sensação de correr atrás dos objetos, tentando encontrar o programa.

Apesar disso, os fanáticos por objetos como eu dão forte preferência ao controle distribuído. Um dos principais objetivos de um bom projeto é localizar os efeitos de alterações. Os dados e o comportamento que acessam esses dados freqüentemente mudam juntos. Portanto, colocar os dados e o comportamento que os utiliza juntos em um único local é a primeira regra do projeto orientado a objetos.

Além disso, com o controle distribuído, você gera mais oportunidades para usar polimorfismo, em vez de utilizar lógica condicional. Se os algoritmos da composição de preços do produto são diferentes para tipos de produto diversos, o mecanismo de controle distribuído nos permite usar subclasses do produto para tratar dessas variações.

Em geral, o estilo orientado a objetos utiliza muitos objetos pequenos com muitos métodos pequenos que nos fornecem muitos pontos de ligação para sobreposição e variação. Esse estilo é muito confuso para as pessoas acostumadas com procedimentos longos; na verdade, essa mudança é o coração da **mudança de paradigma** da orientação a objetos. Isso é algo muito difícil de ensinar. Parece que a única maneira de realmente entender isso é trabalhar por algum tempo em um ambiente orientado a objetos, com controle fortemente distribuído. Muitas pessoas relatam que, repentinamente, exclamam "aha!", quando o estilo faz sentido. Nesse ponto, o cérebro delas foi religado e elas começam a achar que o controle descentralizado é mesmo mais fácil.

COMO CRIAR E EXCLUIR PARTICIPANTES

Os diagramas de seqüência exibem alguma notação extra para criar e excluir participantes (Figura 4.3). Para criar um participante, você desenha a seta da mensagem diretamente para a caixa do participante. Um nome para a mensagem é opcional aqui, caso você esteja usando um construtor, mas normalmente a identifico com "nova", de qualquer forma. Se o participante faz algo imediatamente, quando é criado, como executar o comando de consulta, você inicia uma ativação logo após a caixa do participante.

A exclusão de um participante é indicada por um X grande. Uma seta de mensagem chegando ao X indica um participante explicitamente excluindo outro; um X no final de uma linha de vida mostra um participante excluindo a si mesmo.

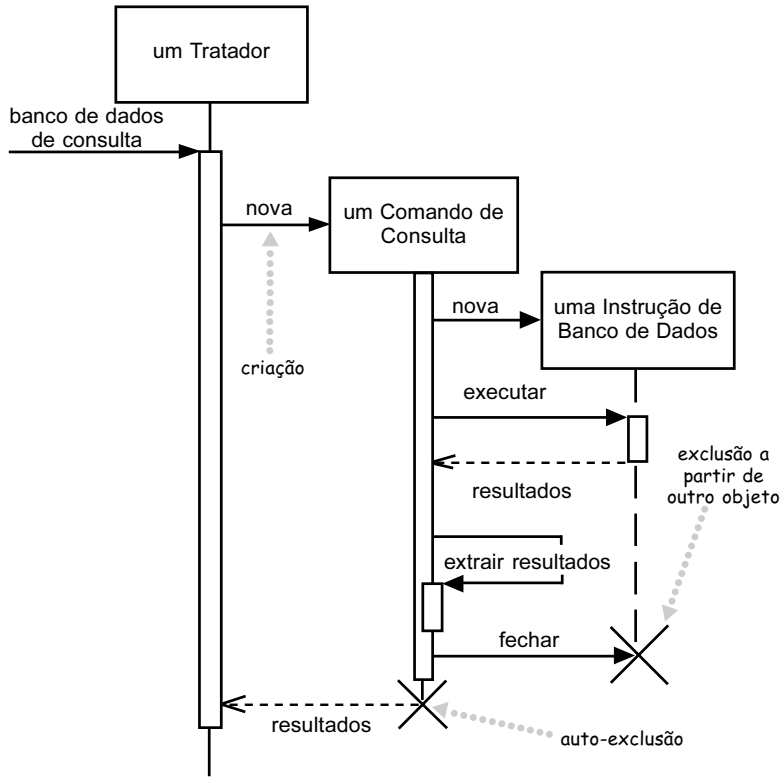


FIGURA 4.3 Criação e exclusão de participantes.

Em um ambiente de coleta de lixo, você não exclui objetos diretamente, mas ainda assim é interessante usar o X para indicar quando um objeto não é mais necessário e está pronto para ser coletado. Isso também é apropriado para operações de fechamento, indicando que o objeto não pode mais ser utilizado.

LAÇOS, CONDICIONAIS, ETC.

Um problema comum dos diagramas de seqüência é como mostrar laços (*loops*) e comportamento condicional. O primeiro detalhe a apontar é que os diagramas de seqüência não servem muito bem para isso. Se você quiser mostrar estruturas de controle como essas, é melhor utilizar um diagrama de atividades ou, na verdade, o próprio código. Trate os diagramas de seqüência como uma visualização de como os objetos interagem, em vez de tratá-los como uma maneira de modelar lógica de controle.

Dito isso, aqui está a notação a ser usada. Os laços e as condicionais usam **quadros de interação**, que são maneiras de demarcar uma parte de um diagrama de seqüência. A Figura 4.4 mostra um algoritmo simples baseado no pseudo-código a seguir:

```
procedure despachar
  foreach (itemdelinha)
    if (produto.valor > $10K)
      cuidadoso.despachar
    else
      regular.despachar
    end if
  enf for
  if (precisaConfirmação) mensageiro.confirmar
end procedure
```

Em geral, os quadros consistem em alguma região de um diagrama de seqüência que é dividida em um ou mais fragmentos. Cada quadro tem um operador e cada fragmento pode ter uma sentinela. (A Tabela 4.1 lista os operadores comuns para quadros de interação.) Para mostrar um laço, você utiliza o operando `loop` com um único fragmento e coloca a base da iteração na sentinela. Para lógica condicional, você pode usar um operador `alt` e colocar uma condição em cada fragmento. Somente o fragmento cuja sentinela é verdadeira será executado. Se você tiver apenas uma região, existe um operador `opt`.

Os quadros de interação são novos na UML 2. Como resultado, você pode ver diagramas preparados antes da existência da UML 2 e que usam uma estratégia diferente. Além disso, algumas pessoas não gostam dos quadros e preferem alguma das convenções mais antigas. A Figura 4.5 mostra algumas dessas otimizações extraoficiais.

A UML 1 utilizava marcadores de iteração e sentinelas. Um **marcador de iteração** é um asterisco (*) adicionado ao nome da mensagem. Você pode adicionar algum texto entre colchetes, para indicar a base da iteração. As **sentinelas** são expressões condicionais colocadas entre colchetes e indicam que a mensagem é enviada somente se a sentinela é verdadeira. Embora essas notações tenham sido eliminadas dos diagramas de seqüência na UML 2, elas ainda são válidas nos diagramas de comunicação.

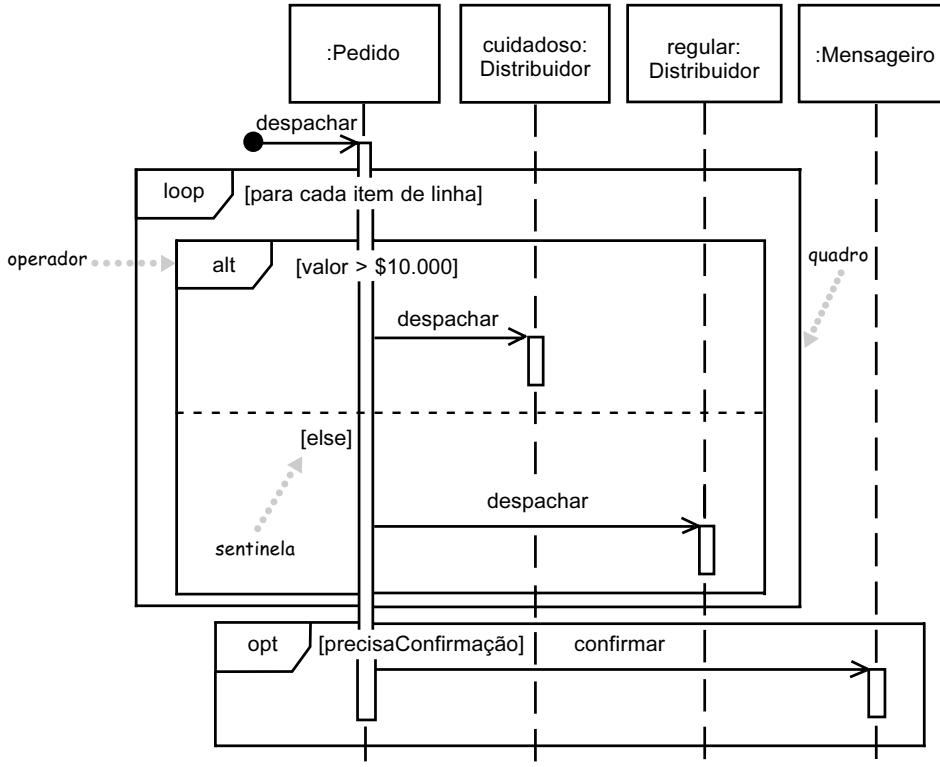


FIGURA 4.4 Quadros de interação.

Embora os marcadores de iteração e as sentinelas possam ajudar, eles têm suas fraquezas. As sentinelas não conseguem indicar que um conjunto delas é mutuamente exclusivo, como as duas da Figura 4.5. As duas notações só funcionam com um envio de

TABELA 4.1 Operadores comuns para quadros de interação

Operador	Significado
`alt`	Múltiplos fragmentos alternativos; somente aquele cuja condição for verdadeira será executado (Figura 4.4).
`opt`	Opcional; o fragmento é executado somente se a condição fornecida for verdadeira. Equivalente a um `alt`, com apenas um caminho (Figura 4.4).
`par`	Paralelo; cada fragmento é executado em paralelo.
`loop`	Laço; o fragmento pode ser executado várias vezes e a sentinela indica a base da iteração (Figura 4.4).
`region`	Região crítica; o fragmento pode ter apenas uma linha de execução ativa por vez.
`neg`	Negativo; o fragmento mostra uma interação inválida.
`ref`	Referência; refere-se a uma interação definida em outro diagrama. O quadro é desenhado de forma a abordar as linhas de vida envolvidas na interação. Você pode definir parâmetros e um valor de retorno.
`sd`	Diagrama de seqüência; usado para circundar um diagrama de seqüência inteiro, se você quiser.

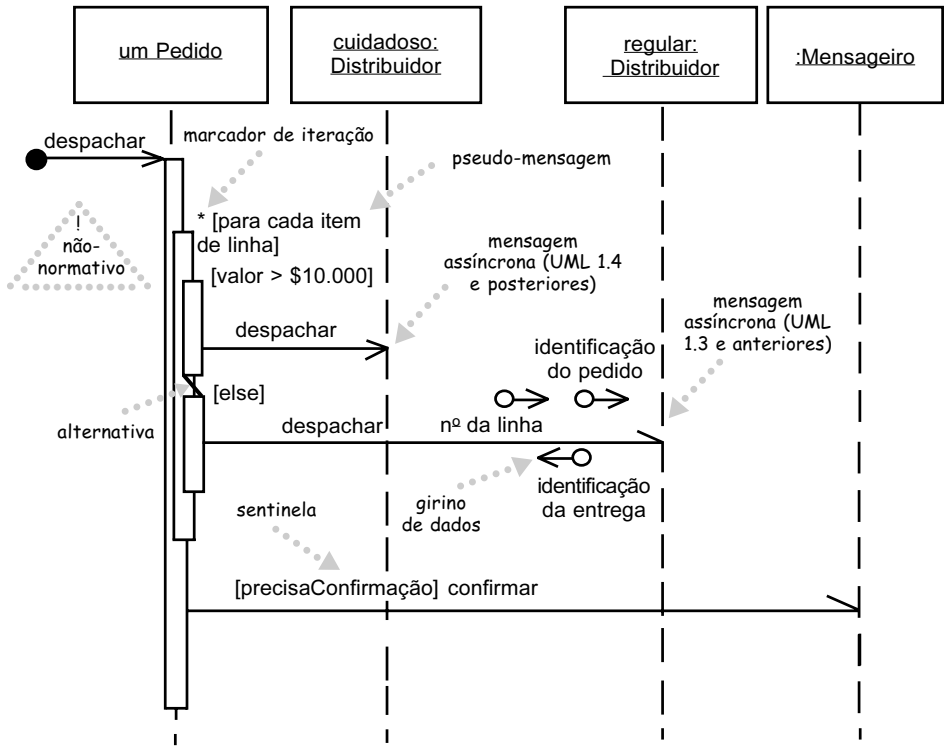


FIGURA 4.5 Convenções mais antigas para lógica de controle.

mensagem e não funcionam bem quando várias mensagens provenientes de uma única ativação estão dentro do mesmo laço ou bloco condicional.

Para contornar este último problema, uma convenção extra-oficial que se tornou popular é usar uma **pseudo-mensagem**, com a condição de laço ou a sentinela em uma variação da notação de autochamada. Na Figura 4.5, mostrei isso sem uma seta de mensagem; algumas pessoas incluem uma seta de mensagem, mas omiti-la ajuda a reforçar que essa não é uma chamada real. Alguns também gostam de sombrear a barra de ativação da pseudo-mensagem. Se você tem comportamento alterador, pode mostrá-lo com um marcador alternativo entre as ativações.

Embora eu considere as ativações muito úteis, elas não acrescentam muito no caso do método `despachar`, por intermédio do qual você envia uma mensagem e nada mais acontece dentro da ativação do receptor. Uma convenção comum, que mostrei na Figura 4.5, é suprimir a ativação para essas chamadas simples.

O padrão UML não tem nenhum dispositivo gráfico para mostrar a passagem de dados; em vez disso, ela é mostrada por meio de parâmetros no nome da mensagem e de setas de retorno. Em muitos métodos, existem **girinos de dados** para indicar o movimento dos dados, e muitas pessoas ainda gostam de utilizá-los com a UML.

No todo, embora vários esquemas possam acrescentar notação para lógica condicional a diagramas de seqüência, não acho que eles funcionem melhor do que o código ou, pelo menos, do que o pseudo-código. Em particular, considero os quadros de interação muito pesados, ocultando o objetivo principal do diagrama; portanto, prefiro as pseudo-mensagens.

CHAMADAS SÍNCRONAS E ASSÍNCRONAS

Se você estiver excepcionalmente alerta, terá notado que as pontas das setas dos últimos dois diagramas são diferentes das dos anteriores. Essa pequena diferença é muito importante na UML 2. Na UML 2, as pontas de seta preenchidas mostram uma mensagem síncrona, enquanto as pontas de seta tipo "pé de galinha" mostram uma mensagem assíncrona.

Se um chamador enviar uma **mensagem síncrona**, ele deve esperar até que ela seja concluída, tal como a chamada a uma sub-rotina. Se um chamador envia uma **mensagem assíncrona**, ele pode continuar o processamento e não precisa esperar por uma resposta. Você vê chamadas assíncronas em aplicativos de múltiplas linhas de execução e em *middleware** orientado por mensagens. A falta de sincronização fornece melhor correspondência e reduz o acoplamento temporal, mas é mais difícil de depurar.

A diferença na ponta da seta é muito sutil; na verdade, sutil demais. Ela também foi uma mudança na UML 1.4, incompatível com as versões anteriores, nas quais uma mensagem assíncrona era mostrada com a seta tipo "meio pé de galinha", como na Figura 4.5.

Acho que essa distinção na ponta da seta é sutil demais. Se você quiser destacar mensagens assíncronas, recomendo utilizar a ponta de seta tipo "meio pé de galinha" obsoleta, que chama muito mais atenção para uma distinção importante. Se você estiver lendo um diagrama de seqüência, cuidado ao fazer suposições a respeito do sincronismo a partir das pontas de seta, a não ser que tenha certeza de que o autor está fazendo a distinção intencionalmente.

QUANDO UTILIZAR DIAGRAMAS DE SEQÜÊNCIA

Você deve utilizar diagramas de seqüência quando quiser observar o comportamento de vários objetos dentro de um único caso de uso. Os diagramas de seqüência são bons para mostrar as colaborações entre os objetos, mas não são tão bons para uma definição precisa do comportamento.

Se você quiser observar o comportamento de um único objeto em muitos casos de uso, utilize um diagrama de estados (veja o Capítulo 10). Se quiser observar o comportamento em muitos casos de uso ou em muitas linhas de execução, considere um diagrama de atividades (veja Capítulo 11).

Se você quiser explorar múltiplas interações alternativas rapidamente, talvez seja melhor usar cartões CRC, pois isso evita desenhar e apagar muitas vezes. Freqüentemente é útil ter uma sessão de cartão CRC para explorar alternativas de projeto e depois utilizar diagramas de seqüência para capturar todas as interações a que você queira fazer referência posteriormente.

Outras formas úteis de diagramas de interação são os diagramas de comunicação, para mostrar conexões, e os diagramas de temporização, para mostrar restrições de temporização.

* N. de R.T.: *Middleware* é a camada de *software* entre os aplicativos e a rede.

Cartões CRC

Uma das técnicas mais valiosas para propor um bom projeto orientado a objetos é explorar as interações dos objetos, pois isso enfoca o comportamento, em vez dos dados. Os diagramas CRC (Class-Responsibility-Collaboration – Classe-Responsabilidade-Colaboração), inventados por Ward Cunningham no final dos anos 80, têm resistido ao teste do tempo como uma maneira altamente eficiente para se fazer isso (Figura 4.6). Embora não façam parte da UML, eles representam uma técnica muito popular entre os projetistas de objeto experientes.

Para usar cartões CRC, você e seus colegas devem se reunir em torno de uma mesa. Peguem diversos cenários e representem-nos com os cartões, escolhendo-os aleatoriamente, quando eles estiverem ativos, movendo-os para sugerir o modo como eles enviam mensagens uns para os outros, e fazendo-os circular. Essa técnica é quase impossível de descrever em um livro, apesar de ser facilmente demonstrada; a melhor maneira de aprendê-la é pedir para alguém que já a tenha utilizado para mostrar a você.

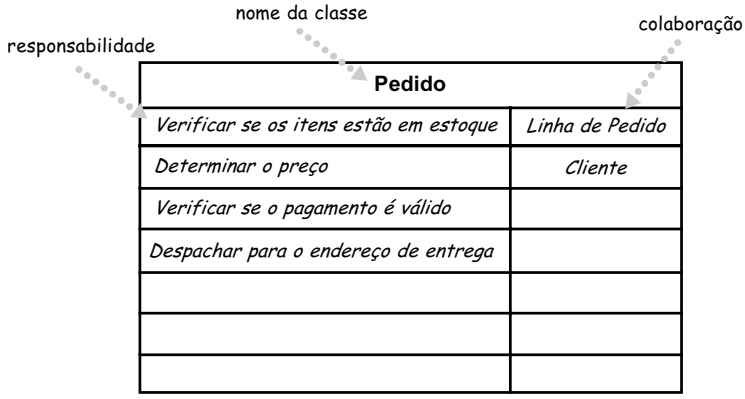


FIGURA 4.6 Um exemplo de cartão CRC.

Uma parte importante do modo de pensar com relação aos cartões CRC é a identificação de responsabilidades. Uma **responsabilidade** é uma frase curta que resume algo que um objeto deve fazer: uma ação que o objeto executa, algum conhecimento que o objeto conserva ou algumas decisões importantes que o objeto toma. A idéia é que você deve ser capaz de pegar qualquer classe e resumi-la com algumas responsabilidades. Fazer isso pode ajudá-lo a pensar mais claramente a respeito do projeto de suas classes.

O segundo C refere-se aos **colaboradores**: as outras classes com as quais essa classe precisa trabalhar. Isso dá uma idéia dos vínculos entre as classes – ainda em um nível alto.

Um dos maiores benefícios dos cartões CRC é que eles estimulam animadas discussões entre os desenvolvedores. Quando você estiver trabalhando com um caso de uso para ver como as classes o implementarão, os diagramas de interação deste capítulo poderão ser demorados de desenhar. Normalmente, você precisa considerar alternativas e com os diagramas, perde-se muito tempo desenhando-as e apagando-as.

Com os cartões CRC, você modela a interação escolhendo-os e movendo-os. Isso permite que se considere rapidamente diversas alternativas.

À medida que faz isso, você forma idéias sobre as responsabilidades e as escreve nos cartões. É importante pensar sobre as responsabilidades, porque isso desvia você da noção de classes como simples portadores de dados e facilita aos membros da equipe compreenderem o comportamento de cada classe em um nível mais elevado. Uma responsabilidade pode corresponder a uma operação, a um atributo ou (mais provavelmente) a um aglomerado indeterminado de atributos e operações.

Um erro comum que vejo as pessoas cometerem é gerar longas listas de responsabilidades de baixo nível. Fazendo isso perde-se o rumo. As responsabilidades devem caber facilmente em um cartão. Você deve questionar se a classe deve ser dividida ou se as responsabilidades seriam mais bem formuladas se englobadas em declarações de nível mais alto.

Muitas pessoas salientam a importância de desempenhar papéis; cada pessoa de uma equipe desempenha o papel de uma ou mais classes. Nunca vi Ward Cunningham fazer isso, e acho que desempenhar papéis atrapalha.

Já foram escritos livros sobre CRC, mas verifiquei que eles nunca chegam realmente ao centro da técnica. O trabalho original sobre CRC, escrito com Kent Beck, é [Beck e Cunnigham]. Para aprender mais sobre cartões CRC e sobre responsabilidades no projeto, dê uma olhada em [Wirfs-Brock].

Capítulo 5

Diagramas de Classes: Conceitos Avançados

Os conceitos descritos no Capítulo 3 correspondem às notações-chave nos diagramas de classes. Esses conceitos são os primeiros com os quais você deve familiarizar-se e compreender, porque eles abrangerão 90% do seu trabalho na construção de diagramas de classes.

A técnica do diagrama de classes, entretanto, promoveu a criação de dezenas de notações de conceitos adicionais. Na realidade, não as utilizo o tempo todo, mas elas são úteis, quando apropriadas. Eu as discutirei uma de cada vez e salientarei alguns dos aspectos de suas utilizações.

Você provavelmente considerará este capítulo um pouco pesado. A boa nova é que, durante sua primeira passada por este livro, você pode seguramente pular este capítulo e retomá-lo mais tarde.

PALAVRAS-CHAVE

Um dos desafios de uma linguagem gráfica é que você precisa lembrar do significado dos símbolos. Existindo muitos deles, os usuários acham muito difícil lembrar o que todos os símbolos significam. Assim, a UML freqüentemente tenta reduzir o número de símbolos e, em seu lugar, utilizar palavras-chave. Se você verificar que precisa de uma construção de modelagem que não está na UML, mas que é semelhante a algo que está, utilize o símbolo da construção UML existente, mas marque-o com uma palavra-chave para mostrar que é algo diferente.

Um exemplo disso é a interface. Uma **interface** da UML (página 69) é uma classe que possui apenas operações públicas, sem nenhum corpo de método. Isso corresponde às interfaces da linguagem Java, COM (Component Object Module) e CORBA. Como se trata de um tipo especial de classe, ela é mostrada usando-se o ícone de classe com a palavra-chave «interface».

Normalmente, as palavras-chave são mostradas como texto entre os caracteres « e ». Como uma alternativa a palavras-chave, você pode usar ícones especiais, mas então terá o problema de todo mundo ter que lembrar o que eles significam.

Algumas palavras-chave aparecem entre chaves, tais como {abstract}. Nunca fica realmente claro o que deve, tecnicamente, estar entre os caracteres «e» e o que deve estar entre chaves. Felizmente, se você errar, apenas os perfeccionistas da UML notarão – ou se preocuparão.

Algumas palavras-chave são tão comuns que freqüentemente são abreviadas: «interface» é muitas vezes abreviada como «I» e {abstract}, como {A}. Tais abreviações são

muito úteis, particularmente em quadros brancos, mas não são padrão; portanto, se você utilizá-las, certifique-se de encontrar um lugar para esclarecer o que elas significam.

Na UML 1, os caracteres « e » eram usados principalmente para **estereótipos**. Na UML 2, os estereótipos são definidos muito rigidamente e descrever o que é e o que não é um estereótipo está fora dos objetivos deste livro. No entanto, por causa da UML 1, muitas pessoas usam o termo *estereótipo* com o mesmo significado de *palavra-chave*, embora isso não seja mais correto.

Os estereótipos são usados como parte dos perfis. Um **perfil** (*profile*) pega uma parte da UML e a estende com um grupo coerente de estereótipos para um propósito específico, como a modelagem de negócio. A semântica completa dos perfis está fora dos objetivos deste livro. A não ser que você esteja em um projeto sério de metamodelo, é improvável que precise criar um perfil. É mais provável que você utilize um já criado para um propósito de modelagem específico, mas felizmente o uso de um perfil não exige o conhecimento de todos os detalhes sobre como eles são vinculados ao metamodelo.

RESPONSABILIDADES

Freqüentemente, é útil mostrar as responsabilidades (página 75) de uma classe em um diagrama de classes. A melhor maneira de mostrá-las é como frases de comentário em seu próprio compartimento na classe (Figura 5.1). Se quiser, você pode dar um nome ao compartimento, mas eu normalmente não faço isso, pois raramente existirá alguma confusão.

OPERAÇÕES E ATRIBUTOS ESTÁTICOS

A UML se refere a uma operação ou a um atributo que se aplica a uma classe, em vez de a uma instância, como **estática**. Isso equivale aos membros estáticos em linguagens baseadas em C. As propriedades estáticas são sublinhadas em um diagrama de classes (veja Figura 5.2).

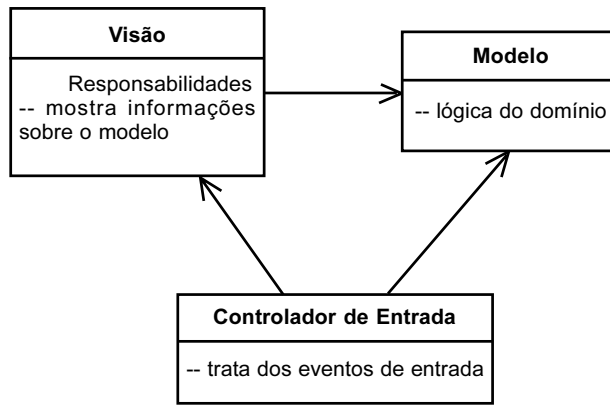


FIGURA 5.1 Mostrando responsabilidades em um diagrama de classes.

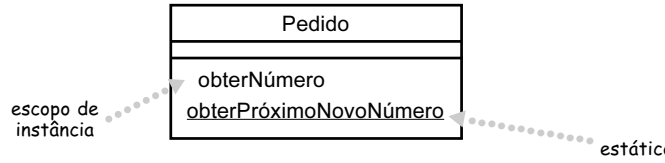


FIGURA 5.2 Notação de propriedade estática.

AGREGAÇÃO E COMPOSIÇÃO

Uma das fontes de confusão mais freqüentes na UML é a agregação e a composição. É fácil explicá-las superficialmente: **agregação** é o relacionamento "parte de". É como dizer que um carro tem um motor e rodas como suas partes. Isso soa bem, mas o difícil é considerar qual é a diferença entre agregação e associação.

Antes do surgimento da UML, as pessoas eram, geralmente, muito vagas a respeito do que era agregação e do que era associação. Vagas ou não, elas sempre eram inconsistentes com as idéias dos outros. Como resultado, muitos projetistas consideram a agregação importante, embora por razões diferentes. Então, a UML incluiu a agregação (Figura 5.3), mas com quase nenhuma semântica. Como Jim Rumbaugh diz: "Pense nisto como um placebo de modelagem" [Rumbaugh, UML Reference].

Assim como para a agregação, a UML tem a propriedade mais definida da **composição**. Na Figura 5.4, uma instância de Ponto pode fazer parte de um polígono ou pode ser o centro de um círculo, mas não pode ser ambos. A regra geral é que, embora uma classe possa ser um componente de muitas outras classes, toda instância deve ser um componente de apenas um proprietário. O diagrama de classes pode mostrar várias classes de proprietários em potencial, mas toda instância tem apenas um único objeto como seu proprietário.

Você notará que não mostro as multiplicidades inversas na Figura 5.4. Na maioria dos casos, como aqui, é 0..1. Seu único outro valor possível é 1, para casos em que a classe de componentes é projetada de modo a poder ter apenas uma outra classe como sua proprietária.

A regra do "não compartilhamento" é a chave da composição. Outra suposição é a de que, se você excluir o polígono, isso deve garantir automaticamente que todos os Pontos a ele pertencentes também sejam excluídos.

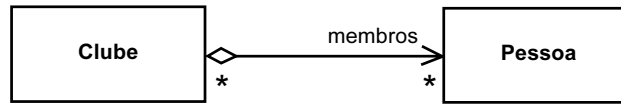


FIGURA 5.3 Agregação.

FIGURA 5.4 Composição.

A composição é uma boa maneira de mostrar propriedades que têm valor, propriedades para objetos de valor (página 84) ou propriedades que têm uma participação como membro forte e um tanto exclusiva de outros componentes específicos. A agregação é estritamente sem significado; como resultado, recomendo que você a ignore em seus diagramas. Se você a encontrar nos diagramas de outras pessoas, precisará ir mais a fundo para descobrir o que elas querem dizer como isso. Diferentes autores e equipes a utilizam para propósitos muito distintos.

PROPRIEDADES DERIVADAS

As **propriedades derivadas** podem ser calculadas com base em outros valores. Quando pensamos a respeito de um intervalo de datas (Figura 5.5), podemos considerar três propriedades: a data de início, a data de término e o número de dias no período. Esses valores são vinculados; portanto, podemos pensar na duração como sendo derivada dos outros dois valores.

Na perspectiva do *software*, a derivação pode ser interpretada de duas maneiras diferentes. Você pode usar derivação para indicar a diferença entre um valor calculado e um valor armazenado. Nesse caso, interpretaríamos a Figura 5.5 como indicando que o início e o término são armazenados, mas que a duração é calculada. Embora esse seja um uso comum, não gosto muito dele, pois ele revela muito dos detalhes internos de `IntervalodeDatas`.

Meu modo de pensar preferido é que ela indica uma restrição entre valores. Neste caso, estamos dizendo que a restrição entre os três valores é válida; no entanto, não é importante qual deles é calculado. Nesse caso, a escolha do atributo a ser marcado como derivado é arbitrária e rigorosamente desnecessária, mas é útil para ajudar a lembrar as pessoas sobre a restrição. Essa utilização também faz sentido nos diagramas conceituais.

A derivação também pode ser aplicada em propriedades, usando-se notação de associação. Nesse caso, você simplesmente marca o nome com uma barra normal (/).

INTERFACES E CLASSES ABSTRATAS

Uma **classe abstrata** é uma classe que não pode ser instanciada diretamente. Em vez disso, você instancia uma instância de uma subclasse. Normalmente, uma classe abstrata tem uma ou mais operações abstratas. Uma **operação abstrata** não tem nenhuma implementação; ela é uma declaração pura para que os clientes possam se ligar à classe abstrata.

A maneira mais comum de indicar uma classe ou operação abstrata na UML é colocar o nome em *itálico*. Você também pode tornar propriedades abstratas indicando uma

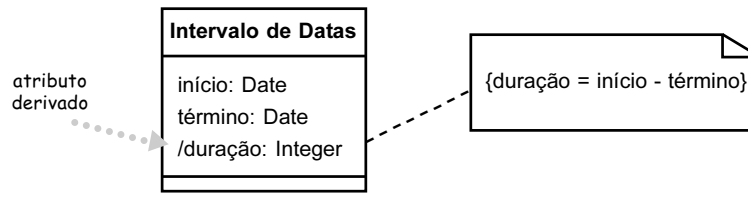


FIGURA 5.5 Atributo derivado em um período de tempo.

propriedade abstrata ou métodos de acesso. As palavras em itálico são chatas de escrever em um quadro branco; portanto, você pode usar o rótulo: {abstract}.

Uma interface é uma classe que não tem nenhuma implementação; isto é, todas as suas propriedades são abstratas. As interfaces correspondem diretamente às interfaces das linguagens C# e Java e são um dialeto comum em outras linguagens tipadas. Você identifica uma interface com a palavra-chave «interface».

As classes têm dois tipos de relacionamentos com as interfaces: fornecimento e exigência. Uma classe **fornece uma interface** se ela é substituível pela interface. Em Java e em .NET, uma classe pode fazer isso implementando a interface ou implementando um subtipo da interface. Em C++, você cria uma subclasse da classe que é a interface.

Uma classe **exige uma interface** se precisa de uma instância dessa interface para funcionar. Basicamente, isso significa ter uma dependência da interface.

A Figura 5.6 mostra esses relacionamentos em ação, baseados em algumas classes de coleção da linguagem Java. Eu poderia escrever uma classe Pedido que tivesse uma lista de itens de linha. Como estou usando uma lista, a classe Pedido é dependente da interface List. Vamos supor que ela use os métodos equals, add e get. Quando os objetos se conectarem, a classe Pedido usará na verdade uma instância de ArrayList, mas não precisará saber disso para utilizar esses três métodos, pois todos eles fazem parte da interface List.

O próprio ArrayList é uma subclasse da classe AbstractList. AbstractList fornece parte da implementação (mas não toda) do comportamento de Lista. Em particular, o

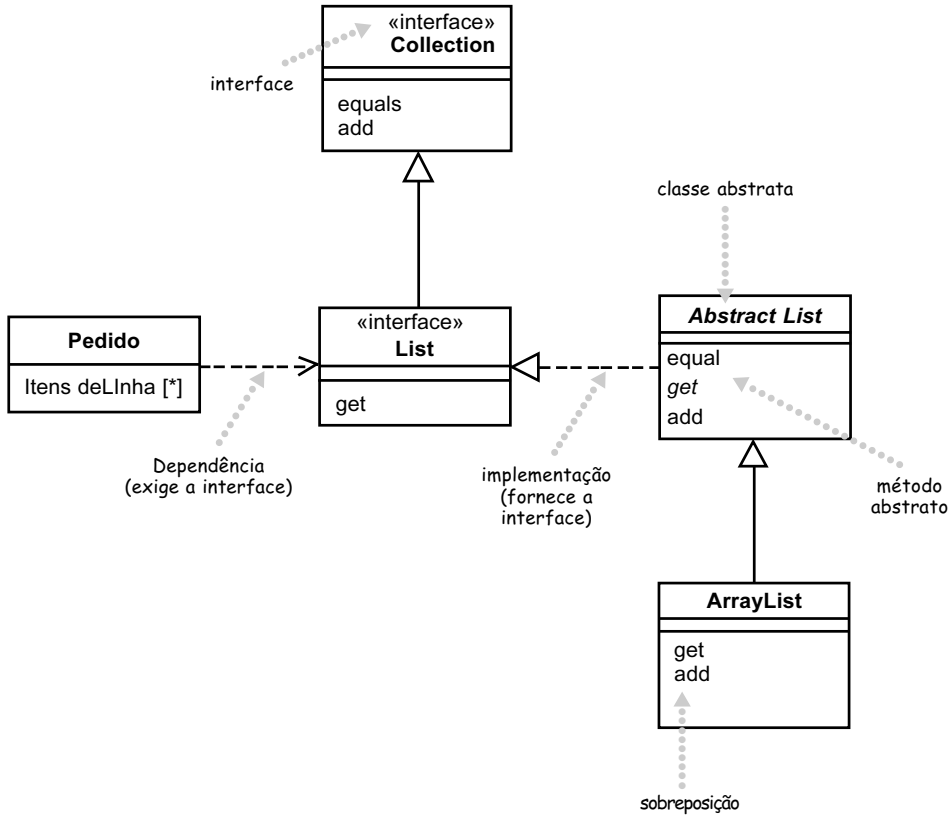


FIGURA 5.6 Um exemplo de interfaces e de uma classe abstrata na linguagem Java.

método `get` é abstrato. Como resultado, `ArrayList` implementa o método `get`, mas também sobrepõe algumas das outras operações em `AbstractList`. Neste caso, ele também sobrepõe o método `add`, mas fica feliz em herdar a implementação do método `equals`.

Por quê não evitar isso simplesmente e fazer `Pedido` usar `ArrayList` diretamente? Utilizando a interface, tenho a vantagem de tornar mais fácil alterar implementações posteriormente, se for necessário. Outra implementação pode fornecer melhorias de desempenho, alguns recursos de interação de banco de dados ou outros benefícios. Programando com interfaces, em vez da implementação direta, não tenho que alterar todo o código, caso precise de uma implementação diferente de `List`. Você sempre deve tentar programar com uma interface como essa; use sempre o tipo mais geral que puder.

Também devo chamar a atenção para uma situação pragmática nisso. Quando os programadores utilizam uma coleção como essa, eles normalmente iniciam a coleção com uma declaração, como a seguinte:

```
private List itensdeLinha = new ArrayList();
```

Note que, rigorosamente, isso introduz uma dependência de `Pedido` para o `ArrayList` concreto. Teoricamente, isso é um problema, mas na prática as pessoas não se preocupam com ele. Como o tipo de `itensdeLinha` é declarado como `List`, nenhuma outra parte da classe `Pedido` é dependente de `ArrayList`. Se alterarmos a implementação, haverá apenas essa única linha de código de inicialização com a qual precisamos nos preocupar. É bastante comum fazer referência a uma classe concreta uma vez, durante a criação, mas posteriormente, utilizar apenas a interface.

A notação completa da Figura 5.6 é uma maneira de indicar interfaces. A Figura 5.7 mostra uma notação mais compacta. O fato de que `ArrayList` implementa `List` e `Collection` é mostrado fora dele por meio de ícones de círculos, freqüentemente referidos como pirulitos. O fato de que `Pedido` exige uma interface `List` é mostrado pelo ícone de soquete. A conexão é bastante evidente.

A UML tem usado a notação de pirulito há algum tempo, mas a notação de soquete é nova na UML 2. (Acho que essa é minha adição de notação favorita.) Você provavelmente verá diagramas mais antigos usando o estilo da Figura 5.8, onde uma dependência resiste à notação de soquete.

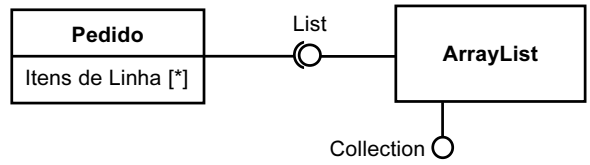


FIGURA 5.7 Notação de bola e de soquete.

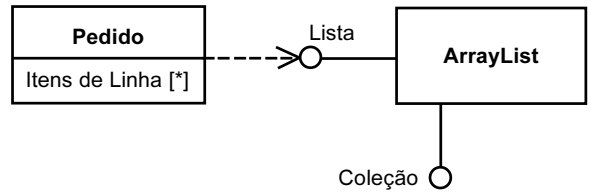


FIGURA 5.8 Dependências mais antigas com pirulitos.

Toda classe é uma mistura de interface e implementação. Portanto, freqüentemente podemos ver um objeto utilizado através da interface de uma de suas superclasses. Rigorosamente falando, não seria válido usar a notação de pirulito para uma superclasse, pois a superclasse é uma classe e não uma interface pura. Mas, em nome da clareza, eu violo essas regras.

Assim como acontece nos diagramas de classes, as pessoas têm considerado os pirulitos úteis em outros contextos. Um dos problemas perenes dos diagramas de interação é que eles não fornecem uma visualização muito boa de comportamento polimórfico. Embora não seja de utilização obrigatória, você pode indicar isso de acordo com a Figura 5.9. Aqui, podemos ver que, embora tenhamos uma instância de Vendedor, que é usada como tal pela Calculadora de Bônus, o objeto Período de Pagamento utiliza o Vendedor somente por intermédio de sua interface Funcionário. (Você pode fazer o mesmo truque com diagramas de comunicação.)

READ-ONLY E *FROZEN*

Na página 54, descrevi a palavra-chave {`readOnly`}. Você usa essa palavra-chave para identificar uma propriedade que só pode ser lida pelos clientes e que não pode ser atualizada. Semelhante, embora diferente, é a palavra-chave {`frozen`} da UML 1. Uma propriedade é ***frozen*** (**congelada**) se não pode ser mudada durante o tempo de vida de um objeto; tais propriedades são freqüentemente chamadas de imutáveis. Embora tenha sido retirado da UML 2, {`frozen`} é um conceito muito útil; portanto, eu continuaria a

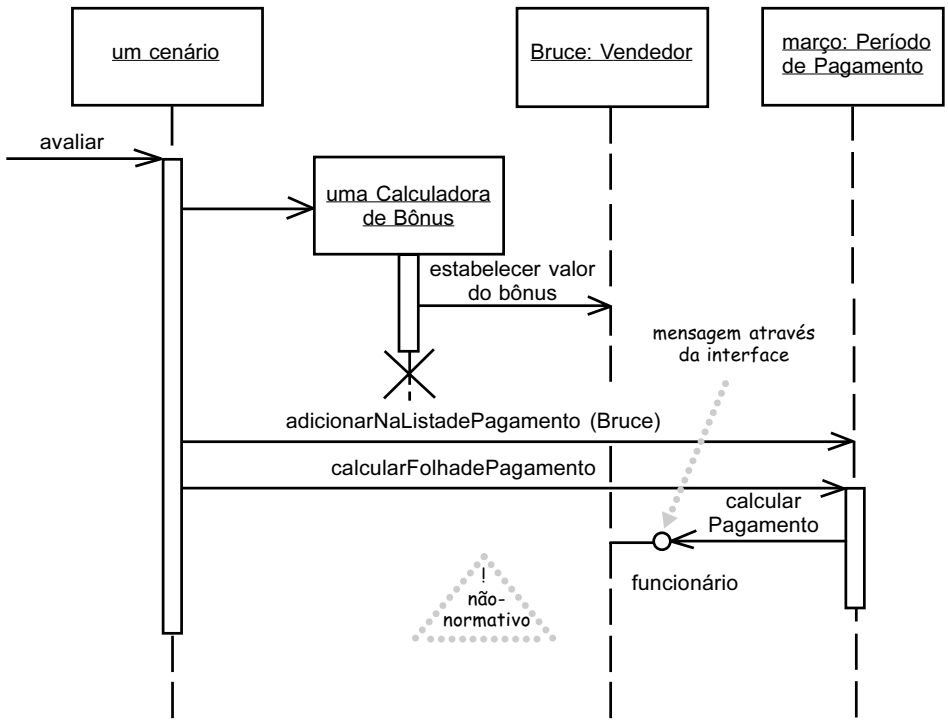


FIGURA 5.9 Usando um pirulito para mostrar polimorfismo em um diagrama de seqüência.

utilizá-lo. Assim como você pode marcar propriedades individuais como *frozen*, também pode aplicar a palavra-chave a uma classe para indicar que todas as propriedades de todas as instâncias estão congeladas. (Ouvi dizer que o conceito de congelamento poderá ser reintegrado em breve.)

OBJETOS DE REFERÊNCIA E OBJETOS DE VALOR

Uma das coisas comumente ditas a respeito dos objetos é que eles têm identidade. Isso é verdade, mas não é tão simples assim. Na prática, você verifica que a identidade é importante para objetos de referência, mas nem tanto para objetos de valor.

Os **objetos de referência** são coisas como um Cliente. Aqui, a identidade é muito importante, pois você normalmente só quer um objeto de *software* para designar um cliente no mundo real. Qualquer objeto que faça referência a um objeto Cliente fará isso por intermédio de uma referência ou ponteiro. Todos os objetos que fazem referência a esse Cliente farão referência ao mesmo objeto de *software*. Desse modo, as alterações em um Cliente estarão disponíveis para todos os usuários do Cliente.

Se você tem duas referências para um Cliente e quer ver se elas são as mesmas, normalmente você compara suas identidades. As cópias podem ser proibidas; se elas forem permitidas, tenderão a ser feitas raramente, talvez para propósitos de arquivamento ou para duplicação através de uma rede. Se forem feitas cópias, você precisará escolher como vai sincronizar as alterações.

Os **objetos de valor** são coisas como Date. Freqüentemente, você tem vários objetos de valor representando o mesmo objeto no mundo real. Por exemplo, é normal ter centenas de objetos que designam 1-Jan-04. Todos eles são cópias intercambiáveis. Novas datas são criadas e destruídas freqüentemente.

Se você tem duas datas e quer ver se elas são as mesmas, não examine as suas identidades, mas sim os valores que representam. Isso normalmente significa que você precisa escrever um operador de teste de igualdade, o qual, para datas, faria um teste do ano, do mês e do dia – ou qualquer que seja a representação interna. Cada objeto que faz referência a 1-Jan-04 normalmente possui seu próprio objeto dedicado, mas você também pode compartilhar datas.

Os objetos de valor devem ser imutáveis; em outras palavras, você não deve poder pegar um objeto de data 1-Jan-04 e mudar o mesmo objeto de data para 2-Jan-04. Em vez disso, você deve criar um novo objeto 2-Jan-04 e usá-lo. O motivo é que, se a data fosse compartilhada, você atualizaria a data de um outro objeto de uma maneira imprevisível, um problema conhecido como **bug de nome alternativo** (***aliasing***).

Antigamente, a diferença entre objetos de referência e objetos de valor era mais clara. Os objetos de valor eram os valores internos do sistema de tipos. Agora, você pode estender o sistema de tipos com suas próprias classes; portanto, esse problema exige mais reflexão.

A UML utiliza a noção de **tipo de dados**, que é mostrada como uma palavra-chave no símbolo de classe. Rigorosamente falando, o tipo de dados não é o mesmo que objeto de valor, pois os tipos de dados não podem ter identidade. Os objetos de valor podem ter uma identidade, mas não a utilizam para comparações de igualdade. Em Java, as primitivas seriam tipos de dados, mas as datas não seriam, embora elas sejam objetos de valor.

Se é importante destacá-las, eu uso composição ao associar com um objeto de valor. Você também pode utilizar uma palavra-chave em um tipo de valor; as convencionais comuns que vejo são «`value`» e «`struct`».

ASSOCIAÇÕES QUALIFICADAS

Associação qualificada é o equivalente da UML para um conceito de programação conhecido como *arrays* associativos, mapeamentos, *hashing* e dicionários. A Figura 5.10 mostra uma maneira de usar um qualificador para representar a associação entre as classes Pedido e Linha de Pedido. O qualificador diz que, em conexão com um Pedido, pode haver uma Linha de Pedido para cada instância de Produto.

Da perspectiva do *software*, essa associação qualificada implicaria em uma interface de acordo com a seguinte:

```
class Pedido...
  public LinhadePedido obterItemdeLinha(Produto umProduto);
  public void adicionarItemdeLinha(Número quantidade, Produto paraProduto);
```

Assim, todo acesso a determinada Linha de Pedido exige um Produto como argumento, sugerindo uma implementação que use uma chave e uma estrutura de dados de valor.

É comum as pessoas ficarem confusas quanto às multiplicidades de uma associação qualificada. Na Figura 5.10, um Pedido pode ter muitos Itens de Linha, mas a multiplicidade da associação qualificada é a multiplicidade no contexto do qualificador. Assim, o diagrama diz que um Pedido tem 0..1 Itens de Linha por Produto. Uma multiplicidade igual a 1 indicaria que Pedido deveria ter um Item de Linha por instância de Produto. Um * indicaria que você teria vários Itens de Linha por Produto mas o acesso aos Itens de Linha seria indexado por Produto.

Na modelagem conceitual, eu uso a construção de qualificador somente para mostrar restrições de acordo com "uma Linha de Pedido por Produto no Pedido".

CLASSIFICAÇÃO E GENERALIZAÇÃO

Ouço, com freqüência, pessoas falarem sobre subtipagem como o relacionamento *é um*. Peço que você tome cuidado com essa maneira de pensar. O problema é que a expressão *é um* pode significar coisas diferentes.

Considere as frases a seguir.

1. Shep é um Border Collie.
2. Um Border Collie é um Cachorro.
3. Cachorros são Animais.
4. Border Collie é uma Raça.
5. Cachorro é uma Espécie.

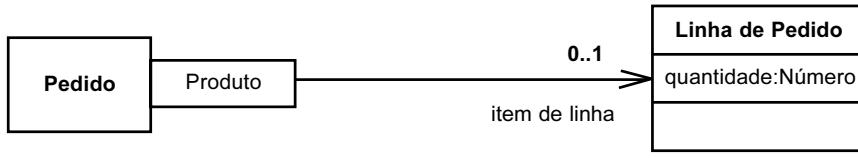


FIGURA 5.10 Associação qualificada.

Agora, tente combinar as frases. Se eu combino as frases 1 e 2, tenho "Shep é um Cachorro"; as frases 2 e 3 juntas dão "Border Collies são Animais". A frase 1 mais a 2 mais a 3 resulta em "Shep é um Animal". Até aqui, tudo bem. Agora, tente as frases 1 e 4: "Shep é uma Raça". A combinação das frases 2 e 5 é "Um Border Collie é uma Espécie". Estas não estão corretas.

Por que posso combinar algumas destas frases e outras não? A razão é que algumas são **classificações** – o objeto Shep é uma instância do tipo Border Collie –, e algumas são **generalizações** – o tipo Border Collie é um subtipo do tipo Cachorro. A generalização é transitiva; a classificação, não. Posso combinar a classificação seguida de uma generalização, mas não o contrário.

Faço questão de mostrar isso para que você desconfie do *é um*. Utilizá-lo pode levar a um uso inadequado da subclassificação e à confusão de responsabilidades. Testes melhores de subtipagem nesse caso seriam as frases "Cachorros são tipos de Animais" e "Toda instância de um Border Collie é uma instância de Cachorro".

A UML utiliza o símbolo de generalização para mostrar generalização. Se você precisa mostrar uma classificação, use uma dependência com a palavra-chave «`instantiate`».

CLASSIFICAÇÃO MÚLTIPLA E DINÂMICA

Classificação refere-se ao relacionamento entre um objeto e o seu tipo. As principais linguagens de programação presumem que um objeto pertença a uma única classe. Mas existem mais opções de classificação, além dessa.

Na **classificação única**, um objeto pertence a um único tipo, que pode herdar de supertipos. Na **classificação múltipla**, um objeto pode ser descrito por vários tipos, que não estão necessariamente conectados por herança.

A classificação múltipla é diferente da herança múltipla. A herança múltipla diz que um tipo pode ter muitos supertipos, mas que um tipo único deve ser definido para cada objeto. A classificação múltipla permite tipos múltiplos para um objeto, sem definir um tipo específico para esse propósito.

Por exemplo, considere uma pessoa subtipada como homem ou como mulher, médico ou enfermeira, paciente ou não (veja Figura 5.11). A classificação múltipla permite que um objeto tenha quaisquer destes tipos atribuídos a ele, em qualquer combinação possível, sem a necessidade de os tipos serem definidos para todas as combinações válidas.

Se você utiliza classificação múltipla, tem que ter certeza de que deixa claro quais combinações são válidas. A UML 2 faz isso colocando cada relacionamento de generalização em um **conjunto de generalização**. No diagrama de classes, você rotula a ponta de seta de generalização com o nome do conjunto de generalização, o qual, na UML 1, era chamado de discriminador. A classificação simples corresponde a um único conjunto de generalização, sem denominação.

Quando não houver especificação, os conjuntos de generalização são separados: qualquer instância do supertipo pode ser uma instância de apenas um dos subtipos dentro desse conjunto. Se você envolver as generalizações em uma única seta, todas elas deverão fazer parte do mesmo conjunto de generalização, como mostrado na Figura 5.11. Como alternativa, você pode ter várias setas com o mesmo rótulo de texto.

Para ilustrar, observe as seguintes combinações válidas de subtipos no diagrama: (Feminino, Paciente, Enfermeira); (Masculino, Fisioterapeuta); (Feminino, Paciente); e

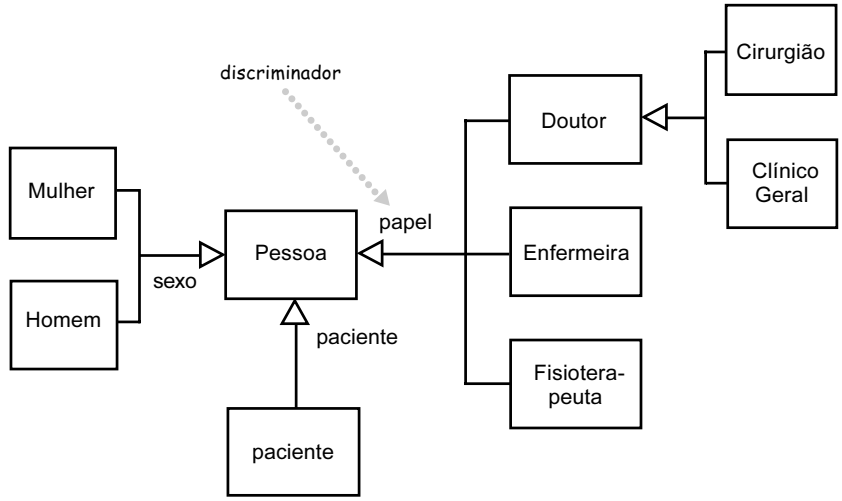


FIGURA 5.11 Classificação múltipla.

(Feminino, Médico, Cirurgião). A combinação (Paciente, Médico, Enfermeira) é inválida, pois ela contém dois tipos do conjunto de generalização de papel.

Outra questão é se um objeto pode mudar a sua classe. Por exemplo, quando uma conta bancária está sem fundos, ela muda substancialmente o seu comportamento. Especificamente, várias operações, incluindo "retirar" e "fechar", são sobrescritas.

A **classificação dinâmica** permite que os objetos mudem de classe dentro da estrutura de subtipagem; a **classificação estática** não admite isso. Na classificação estática, é feita uma separação entre tipos e estados; a classificação dinâmica combina essas noções.

Deve-se utilizar classificação múltipla e dinâmica? Considero-a útil para modelagem conceitual. Da perspectiva do *software*, entretanto, a distância entre ela e as implementações é enorme. Na grande maioria dos diagramas em UML, você verá apenas a classificação estática simples; portanto, esse deve ser seu padrão.

CLASSE DE ASSOCIAÇÃO

As **classes de associação** permitem que você acrescente atributos, operações e outras características a associações, como mostrado na Figura 5.12. Podemos ver pelo diagrama que uma pessoa pode participar de muitas reuniões. Precisamos manter informações sobre o quanto essa pessoa estaria desperta; podemos fazer isso adicionando o atributo atenciosidade à associação.

A Figura 5.13 mostra outra maneira de representar essa informação: tornar Comparecimento uma classe completa. Observe como as multiplicidades mudaram.

Que vantagens você obtém com a classe de associação para compensar a notação extra que precisa lembrar? A classe de associação acrescenta uma restrição a mais, no sentido de que pode haver apenas uma instância da classe de associação entre quaisquer dois objetos participantes. Sinto a necessidade de outro exemplo.

Dê uma olhada nos dois diagramas da Figura 5.14. Esses diagramas têm praticamente a mesma forma. Entretanto, podemos imaginar uma Empresa desempenhando diferentes papéis no mesmo Contrato, mas é difícil imaginar uma Pessoa tendo múlti-

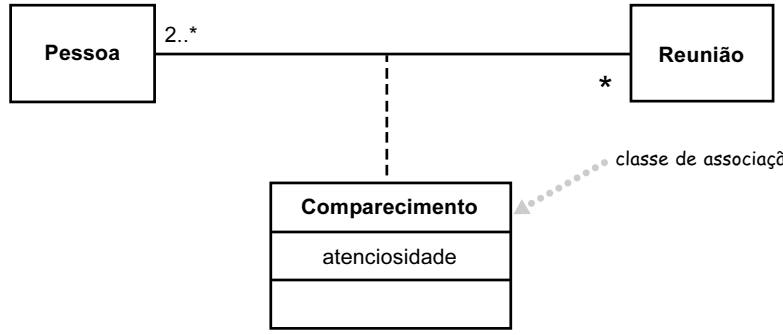


FIGURA 5.12 Classe de associação.

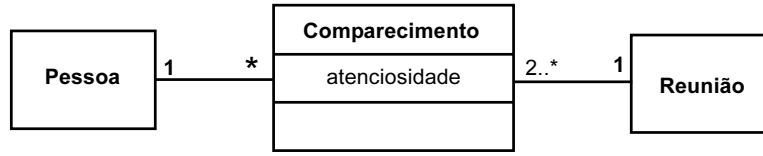


FIGURA 5.13 Promovendo uma classe de associação para uma classe completa.

plas competências na mesma habilidade; na verdade, você provavelmente consideraria isso um erro.

Na UML, apenas o último caso é válido. Você pode ter apenas uma competência para cada combinação de Pessoa e Habilidade. O diagrama superior na Figura 5.14 não permitiria que uma Empresa tivesse mais de um Papel em um único contrato. Se você precisa permitir isso, então é necessário tornar Papel uma classe completa, no estilo da Figura 5.13.

A implementação de classes de associação não é terrivelmente óbvia. Meu conselho é implementar uma classe de associação como se ela fosse uma classe completa, mas fornecer métodos que obtenham informações para as classes vinculadas pela classe de associação. Assim, para a Figura 5.12, veríamos os seguintes métodos em Pessoa:

```
class Pessoa
  List obterComparecimentos()
  List obterReuniões()
```

Desse modo, um cliente de Pessoa pode controlar as pessoas na reunião; se elas quiserem detalhes, podem obter os Comparecimentos por si próprias. Se você fizer isso, lembre-se de impor a restrição de que só pode haver um objeto Comparecimento para qualquer par de Pessoa e Reunião. Você deve colocar uma verificação em qualquer método que crie objetos Comparecimento.

Você encontra freqüentemente esse tipo de construção com informações históricas, como na Figura 5.15. No entanto, acho que criar classes extras ou classes de associações pode tornar o modelo difícil de entender, assim como inclina a implementação em uma direção específica, que muitas vezes é inapropriada.

Se eu tenho esse tipo de informação temporal, uso a palavra-chave «temporal» na associação (veja a Figura 5.16). O modelo indica que uma Pessoa pode trabalhar

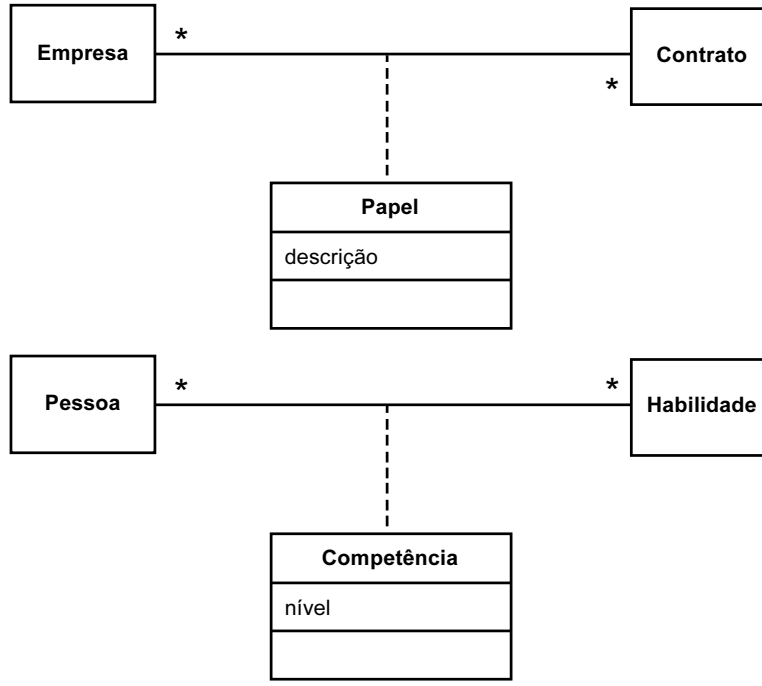


FIGURA 5.14 Sutilezas da classe de associação (Papel provavelmente não deveria ser uma classe de associação).

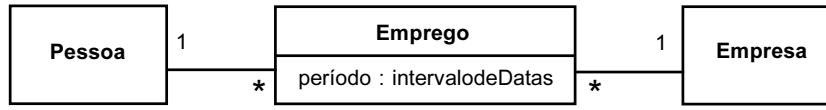


FIGURA 5.15 Usando uma classe para um relacionamento temporal.

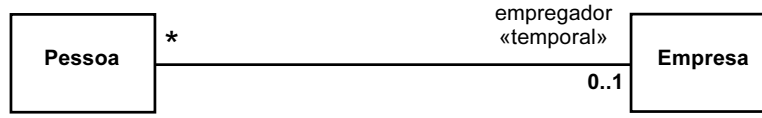


FIGURA 5.16 A palavra-chave «temporal» para associações.

apenas para uma única Empresa por vez. Com o passar do tempo, entretanto, uma Pessoa pode trabalhar para várias Empresas. Isso sugere uma interface de acordo com o seguinte:

```
class Pessoa
  Empresa obterEmpregador();//obtém o empregador atual
  Empresa obterEmpregador(Date);//obtém empregador em determinada data
  void trocarde mudarEmpregador(Empresa novoEmpregador, Date datadaTroca);
  void deixarEmpregador (Date datadaTroca);
```

A palavra-chave «temporal» não faz parte da UML, mas eu a menciono aqui por dois motivos. Primeiro, trata-se de uma notação que considerei útil em várias ocasiões em minha carreira de modelagem. Segundo, ela mostra como você pode usar palavras-chave para estender a UML. Você pode ler muito mais sobre isso no endereço **http://martinfowler.com/ap2/timeNarrative.html**.

CLASSE *TEMPLATE* (PARAMETRIZADA)

Várias linguagens, mais notadamente C++, têm a noção de **classe parametrizada** ou ***template***. (Os *templates* estão na lista para serem incluídos em Java e em C# em um futuro próximo.)

Esse conceito, obviamente, é mais útil para trabalhar com coleções em uma linguagem fortemente tipada. Dessa forma, você pode definir comportamento para conjuntos em geral, definindo uma classe *template* Conjunto.

```
class Conjunto <T> {
  void insert (T novoElemento);
  void remove (T umElemento);
```

Tendo feito isso, você pode usar a definição geral para fazer classes Conjunto para elementos mais específicos.

```
Conjunto <Funcionário> conjuntodeFuncionários;
```

Você declara uma classe *template* na UML usando a notação mostrada na Figura 5.17. O T no diagrama é um lugar reservado para o parâmetro de tipo. (Você pode ter mais de um.)

Um uso de uma classe parametrizada tal como `Conjunto<Funcionário>`, é chamado de **derivação**. Você pode mostrar uma derivação de duas maneiras. A primeira espelha a sintaxe da linguagem C++ (veja a Figura 5.18). Você descreve a expressão de derivação entre colchetes, na forma `<nome-parâmetro::valor-parâmetro>`. Se houver apenas um

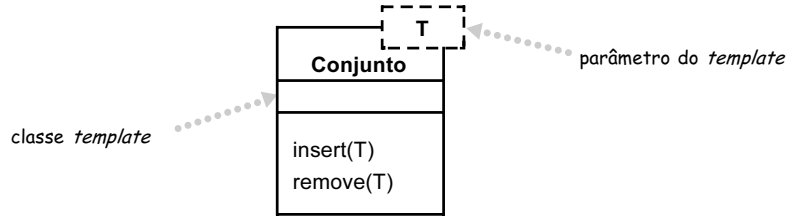


FIGURA 5.17 Classe *template*.

FIGURA 5.18 Elemento de amarração (versão 1).

parâmetro, o uso convencional freqüentemente omite o nome do parâmetro. A notação alternativa (veja a Figura 5.19) reforça o elo para o *template* e permite que você mude o nome do elemento de amarração.

A palavra-chave «bind» é um estereótipo no relacionamento de refinamento. Esse relacionamento indica que ConjuntodeFuncionários estará de acordo com a interface de Conjunto. Você pode considerar ConjuntodeFuncionários como um subtipo de Conjunto. Isso se encaixa na outra forma de implementar coleções de tipo específico, que é declarar todos os subtipos apropriados.

No entanto, o uso de uma derivação *não* é o mesmo que subtipagem. Você não pode acrescentar propriedades no elemento de amarração, que é completamente especificado pelo seu *template*; você está acrescentando somente uma informação de tipo restrito. Se quiser acrescentar propriedades, você deve criar um subtipo.

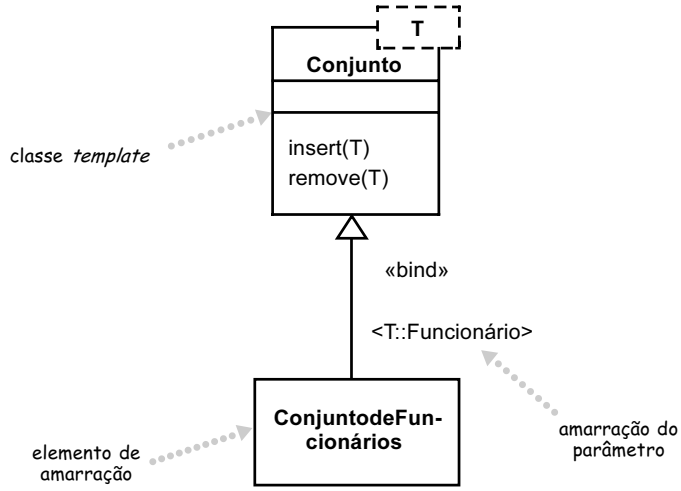


FIGURA 5.19 Elemento de amarração (versão 2).

ENUMERAÇÕES

As enumerações (Figura 5.20) são usadas para mostrar um conjunto fixo de valores que não possuem quaisquer propriedades além de seu valor simbólico. Elas são mostradas como a classe com a palavra-chave «enumeration».

FIGURA 5.20 Enumeração.

CLASSE ATIVA

Uma **classe ativa** tem instâncias, cada uma das quais executa e controla sua própria linha de execução de controle. Chamadas de método podem ser executadas em uma linha de execução do cliente ou em uma linha de execução do objeto ativo. Um bom exemplo disso é um processador de comandos que aceita objetos comando do exterior e, então, executa os comandos dentro de sua própria linha de execução de controle.

A notação para classes ativas mudou da UML 1 para a UML 2, como mostrado na Figura 5.21. Na UML 2, uma classe ativa tem linhas verticais extras na lateral; na UML 1, uma classe ativa tinha uma borda grossa e era chamada de objeto ativo.

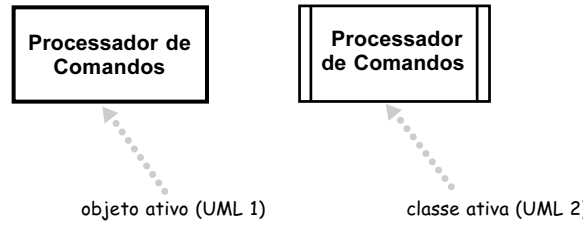


FIGURA 5.21 Classe ativa.

VISIBILIDADE

A **visibilidade** é um assunto que é simples em princípio, mas tem sutilezas complexas. A idéia básica é que qualquer classe tem elementos públicos e elementos privados. Os elementos públicos podem ser usados por qualquer outra classe; os elementos privados podem ser usados somente pela classe proprietária. Entretanto, cada linguagem tem suas próprias regras. Embora muitas linguagens usem termos como *public* (público), *private* (privado) e *protected* (protegido), eles têm significados distintos nas diferentes linguagens. Essas diferenças são pequenas, mas causam confusão, especialmente para aquelas pessoas que utilizam mais de uma linguagem.

A UML tenta abordar o tema sem entrar em uma terrível confusão. Basicamente, dentro da UML, você pode rotular qualquer atributo ou operação com um indicador de visibilidade. Você pode usar o marcador que quiser, e seu significado é dependente da linguagem. Entretanto, a UML fornece quatro abreviações para visibilidade; + (público), – (privado), ~ (pacote) e # (protegido). Esses quatro níveis são usados dentro do metamodelo da UML e são definidos dentro dele, mas suas definições variam sutilmente daquelas das outras linguagens.

Quando você estiver aplicando visibilidade, utilize as regras da linguagem na qual está trabalhando. Quando você estiver examinando um modelo da UML de qualquer outra origem, seja cuidadoso com o significado dos marcadores de visibilidade e esteja ciente de como esses significados podem mudar de uma linguagem para outra.

Na maioria das vezes, não desenho marcadores de visibilidade nos diagramas; eu os utilizo apenas se preciso destacar as diferenças na visibilidade de certos recursos. Mesmo assim, posso me virar com + e –, que pelo menos são fáceis de lembrar.

MENSAGENS

A UML padrão não mostra nenhuma informação sobre chamadas de mensagem nos diagramas de classe. Entretanto, às vezes, tenho visto diagramas convencionais como o da Figura 5.22.

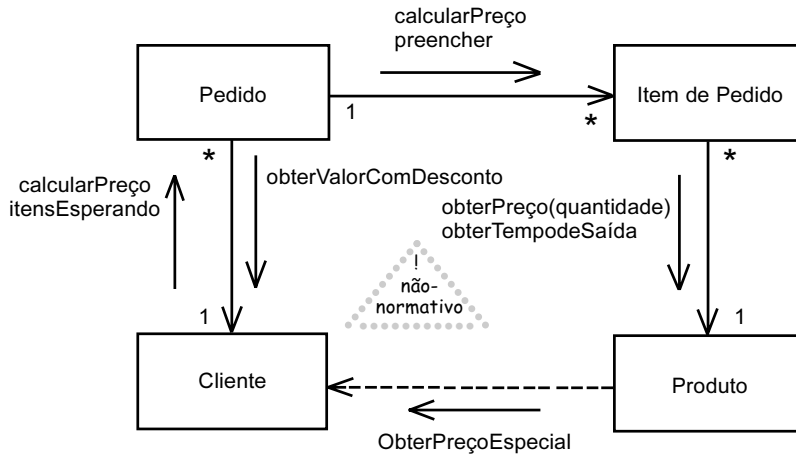


FIGURA 5.22 Classes com mensagens.

Eles adicionam setas ao longo das associações. As setas são rotuladas com as mensagens que um objeto envia para outro. Como você não precisa de uma associação com uma classe para enviar uma mensagem a ela, talvez também precise adicionar uma seta de dependência para mostrar as mensagens entre classes que não estão associadas.

Essas informações de mensagem abrangem múltiplos casos de uso; portanto, não são numeradas para mostrar seqüências, ao contrário dos diagramas de comunicação.

Capítulo 6

Diagramas de Objetos

Um **diagrama de objetos** é um instantâneo dos objetos em um sistema em um determinado ponto no tempo. Como ele mostra instâncias, em vez de classes, um diagrama de objetos é freqüentemente chamado de diagrama de instâncias.

Você pode usar um diagrama de objetos para mostrar um exemplo de configuração de objetos. (Veja a Figura 6.1, que mostra um conjunto de classes, e a Figura 6.2, que mostra um conjunto de objetos associados.) Este último uso é muito útil, quando as conexões possíveis entre os objetos são complicadas.

Você pode dizer que os elementos da Figura 6.2 são instâncias porque os nomes estão sublinhados. Cada nome assume a forma `nome de instância : nome de classe`. As duas partes do nome são opcionais; portanto, `John`, `:Pessoa` e `umaPessoa` são nomes válidos. Se você usar apenas o nome da classe, deve incluir os dois pontos. Você pode mostrar valores para atributos e vínculos, como na Figura 6.2.

Rigorosamente falando, os elementos de um diagrama de objetos são especificações de instâncias, em vez de serem instâncias verdadeiras. O motivo é que é válido deixar atributos obrigatórios vazios ou mostrar especificações de instâncias de classes abstratas. Você pode considerar uma **especificação de instância** como uma instância parcialmente definida.

Outra maneira de ver um diagrama de objetos é como um diagrama de comunicação (página 129) sem mensagens.

QUANDO USAR DIAGRAMAS DE OBJETOS

Os diagramas de objetos são úteis para mostrar exemplos de objetos interligados. Em muitas situações, você pode definir uma estrutura precisamente, com um diagrama de classes, mas a estrutura ainda é difícil de entender. Nessas situações, dois exemplos de diagrama de objetos podem fazer a diferença.

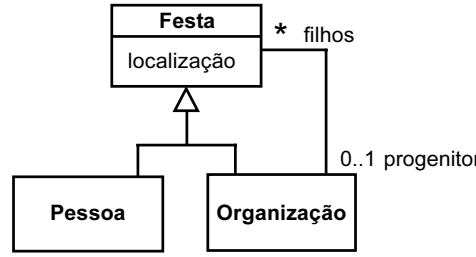


FIGURA 6.1 Diagrama de classes da estrutura de composição Festa.

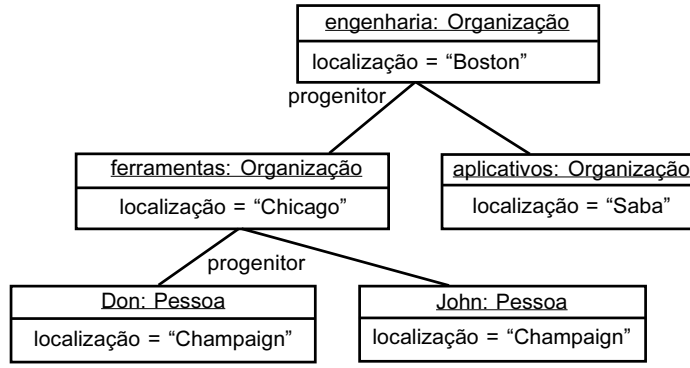


FIGURA 6.2 Diagrama de objetos mostrando exemplos de instâncias de Festa.

Capítulo 7

Diagramas de Pacotes

As classes representam a forma básica de estruturação de um sistema orientado a objetos. Embora elas sejam maravilhosamente úteis, você precisa de algo mais para estruturar sistemas grandes, os quais podem ter centenas de classes.

Um **pacote** é uma construção de agrupamento que permite a você pegar qualquer construção na UML e agrupar seus elementos em unidades de nível mais alto. Seu uso mais comum é o agrupamento de classes e é dessa maneira que o estou descrevendo aqui, mas lembre-se de que você também pode usar pacotes para todos os outros elementos da UML.

Em um modelo da UML, cada classe é membro de um único pacote. Os pacotes também podem ser membros de outros pacotes, de modo que você obtém uma estrutura hierárquica na qual os pacotes de nível superior são divididos em subpacotes que possuem seus próprios subpacotes e assim por diante, até que a hierarquia chegue nas classes. Um pacote pode conter subpacotes e classes.

Em termos de programação, os pacotes correspondem a contruções de agrupamento como pacotes (em Java) e espaços de nomes (em C++ e .NET).

Cada pacote representa um **espaço de nomes**, o que significa que toda classe deve ter um nome exclusivo dentro do pacote a que pertence. Se eu quiser criar uma classe chamada Date e já houver uma classe Date no pacote System, posso ter minha classe Date, desde que a coloque em um pacote separado. Para tornar claro qual é qual, posso usar um **nome totalmente qualificado**; isto é, um nome que mostra a estrutura de pacotes ao qual pertence. Para indicar nomes de pacote na UML, você usa dois pontos duplos; portanto, as datas poderiam ser `System::Date e MartinFowler::Util::Date`.

Nos diagramas, os pacotes são mostrados por uma pasta com guia, como se vê na Figura 7.1. Você pode mostrar simplesmente o nome do pacote ou mostrar também o conteúdo. Em qualquer ponto, você pode usar nomes totalmente qualificados ou apenas nomes normais. Exibir o conteúdo com ícones de classe permite a você mostrar todos os detalhes de uma classe, podendo apresentar até mesmo um diagrama de classes dentro do pacote. Listar simplesmente os nomes faz sentido quando tudo que você quer fazer é indicar quais classes estão em quais pacotes.

É muito comum ver uma classe rotulada com algo como Date (`de java.util`), em vez da forma totalmente qualificada. Esse estilo é uma convenção que foi muito utilizada pela Rational Rose; ele não faz parte do padrão.

A UML permite que as classes de um pacote sejam públicas ou privadas. Uma classe pública faz parte da interface do pacote e pode ser usada por classes de outros pacotes; uma classe privada fica oculta. Diferentes ambientes de programação têm regras distintas sobre visibilidade entre suas construções de empacotamento; você deve seguir a

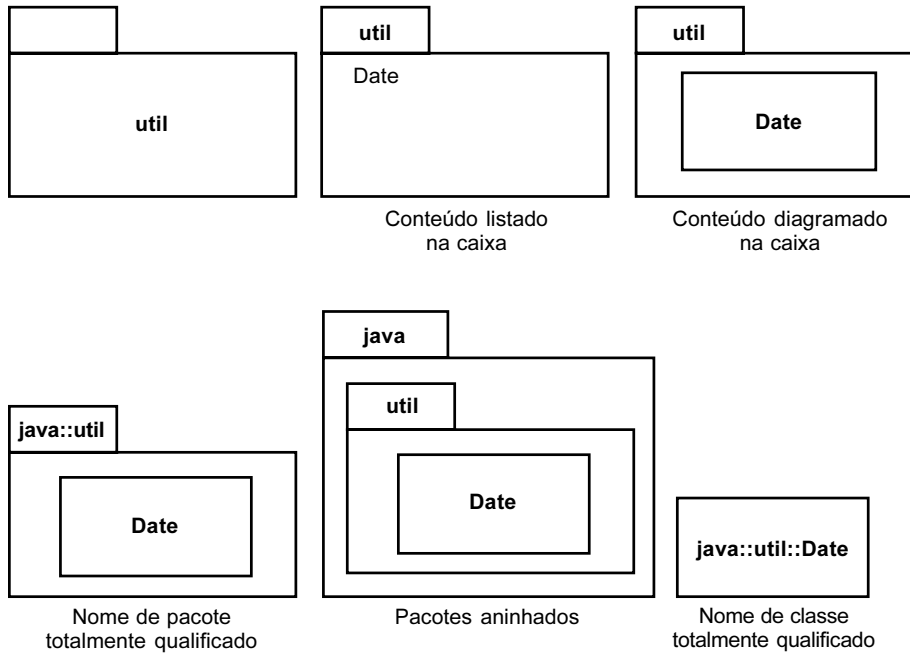


FIGURA 7.1 Maneiras de mostrar pacotes em diagramas.

convenção de seu ambiente de programação, mesmo que isso signifique violar as regras da UML.

Uma técnica útil, aqui, é reduzir a interface do pacote, exportando apenas um pequeno subconjunto das operações associadas às classes públicas do pacote. Você pode fazer isso fornecendo visibilidade privada a todas as classes, para que elas possam ser vistas apenas por outras classes do mesmo pacote, e adicionando classes públicas extras para o comportamento público. Então, essas classes extras, chamadas *Facades* [Gangue dos Quatro], delegam operações públicas às suas companheiras mais tímidas do pacote.

Como você escolhe que classes vai colocar em quais pacotes? Na verdade, essa é uma questão bastante complicada, que precisa de uma boa habilidade com projetos para ser respondida. Dois princípios úteis são o Princípio do Fechamento Comum e o Princípio da Reutilização Comum [Martin]. O Princípio do Fechamento Comum diz que as classes de um pacote devem precisar de alteração por motivos semelhantes. O Princípio da Reutilização Comum diz que todas as classes de um pacote devem ser reutilizadas juntas. Muitos dos motivos para o agrupamento de classes em pacotes estão relacionados às dependências entre os pacotes, que é o que veremos a seguir.

PACOTES E DEPENDÊNCIAS

Um **diagrama de pacotes** mostra pacotes e suas dependências. Eu apresentei a noção de dependência na página 47. Se você tem pacotes de apresentação e de domínio, então tem uma dependência do pacote de apresentação para o pacote de domínio, caso qualquer classe no pacote de apresentação tenha uma dependência de qualquer classe no pacote de

domínio. Desse modo, as dependências entre os pacotes resumem as dependências entre seus conteúdos.

A UML tem muitas variedades de dependências, cada uma com uma semântica e um estereótipo em particular. Acho mais fácil começar com a dependência não-estereotipada e usar as dependências mais particulares somente se for necessário, o que dificilmente acontece.

Em um sistema médio ou grande, representar um diagrama de pacotes pode ser uma das coisas mais valiosas que você pode fazer para controlar a estrutura de larga escala do sistema. De preferência, esse diagrama deve ser gerado a partir da própria base de código, para que você possa ver o que realmente há no sistema.

Uma boa estrutura de pacotes tem um fluxo claro das dependências, um conceito difícil de definir, mas freqüentemente mais fácil de reconhecer. A Figura 7.2 mostra um exemplo de diagrama de pacotes para uma aplicação comercial, o qual é bem estruturado e tem um fluxo claro.

Freqüentemente, você pode identificar um fluxo claro porque todas as dependências seguem uma única direção. Embora esse seja um bom indicador de um sistema bem-estruturado, os pacotes de mapeamento de dados da Figura 7.2 mostram uma exceção a essa regra geral. Os pacotes de mapeamento de dados atuam como uma camada de isolamento entre os pacotes de domínio e de banco de dados, um exemplo do padrão Mapper [Fowler, P de EAA].

Muitos autores dizem que não devem existir ciclos nas dependências (o Princípio da Dependência Acíclica [Martin]). Não trato isso como uma regra absoluta, mas acho que os ciclos devem ser localizados e, em particular, que você não deve ter ciclos que cruzem camadas.

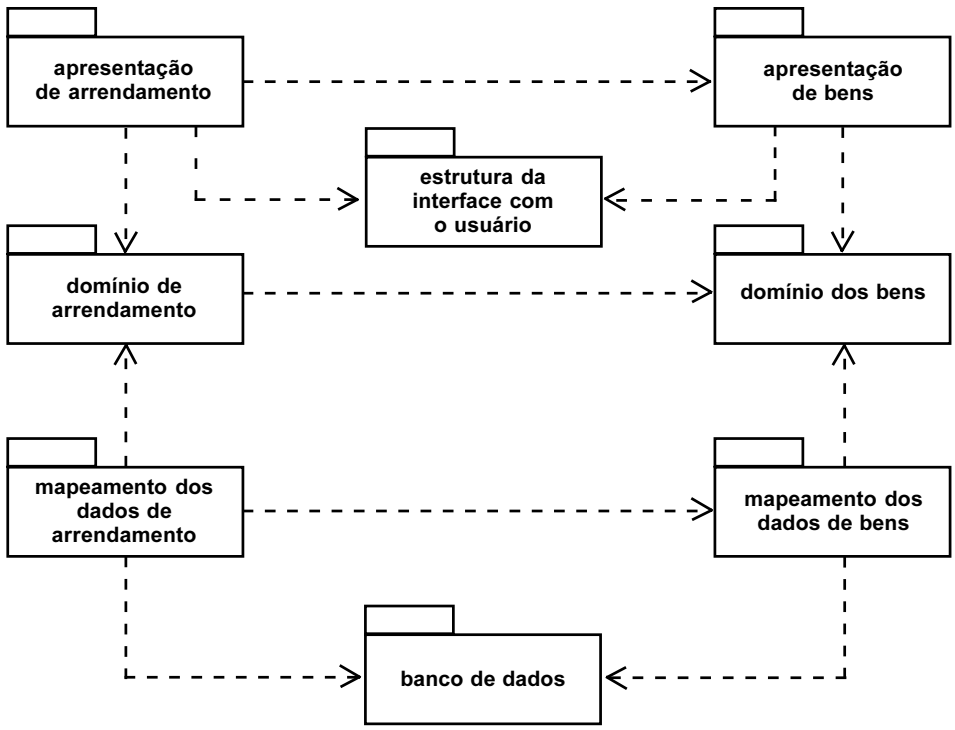


FIGURA 7.2 Diagrama de pacotes para uma aplicação comercial.

Quanto mais dependências entram em um pacote, mais estável a interface do pacote precisa ser, pois qualquer alteração em sua interface será propagada para todos os pacotes que são dependentes dela (o Princípio das Dependências Estáveis [Martin]). Assim, na Figura 7.2, o pacote do domínio dos bens precisa de uma interface mais estável do que o pacote de mapeamento de dados de arrendamento. Freqüentemente, você verá que os pacotes mais estáveis tendem a ter uma proporção mais alta de interfaces e classes abstratas (o Princípio das Abstrações Estáveis [Martin]).

Os relacionamentos de dependência não são transitivos (página 48). Para ver por que isso é importante para dependências, veja novamente a Figura 7.2. Se uma classe no pacote do domínio de bens muda, talvez tenhamos que alterar as classes dentro do pacote do domínio de arrendamento. Mas essa alteração não se propaga necessariamente para a apresentação de arrendamento. (Ela só se propaga se o domínio de arrendamento muda sua interface.)

Alguns pacotes são usados em tantos lugares, que seria uma confusão desenhar todas as linhas de dependência para eles. Nesse caso, uma convenção é usar uma palavra-chave, como «`global`», no pacote.

Os pacotes da UML também definem construções para permitir a importação e mesclagem de classes de um pacote para outro, usando dependências com palavras-chave para denotar isso. Entretanto, as regras para esse tipo de coisa variam muito com as linguagens de programação. Em geral, considero a noção de dependências bem mais útil na prática.

ASPECTOS DOS PACOTES

Se você pensar a respeito da Figura 7.2, perceberá que o diagrama tem dois tipos de estruturas. Uma delas é a estrutura de camadas na aplicação: apresentação, domínio, mapeamento de dados e banco de dados. A outra é uma estrutura de áreas de assunto: arrendamento e bens.

Você pode tornar isso mais aparente, separando os dois aspectos, como na Figura 7.3. Com esse diagrama, você pode ver claramente cada aspecto. Entretanto, esses dois aspectos não são pacotes verdadeiros, pois você não pode atribuir classes a um único pacote. (Você teria que escolher uma de cada aspecto.) Esse problema espelha o problema dos espaços de nomes hierárquicos nas linguagens de programação. Embora diagramas como o da Figura 7.3 não sejam UML padrão, freqüentemente eles são muito úteis na explicação da estrutura de uma aplicação complexa.

COMO IMPLEMENTAR PACOTES

Freqüentemente, você verá um caso em que um pacote define uma interface, que pode ser implementada por vários outros pacotes, como mostra a Figura 7.4. Nesse caso, o relacionamento de realização indica que o *gateway* do banco de dados define uma interface e que as outras classes de *gateway* fornecem uma implementação. Na prática, isso significaria que o pacote de *gateway* de banco de dados contém interfaces e classes abstratas que são totalmente implementadas pelos outros pacotes.

É muito comum que uma interface e sua implementação estejam em pacotes separados. Na verdade, um pacote de cliente freqüentemente contém uma interface para

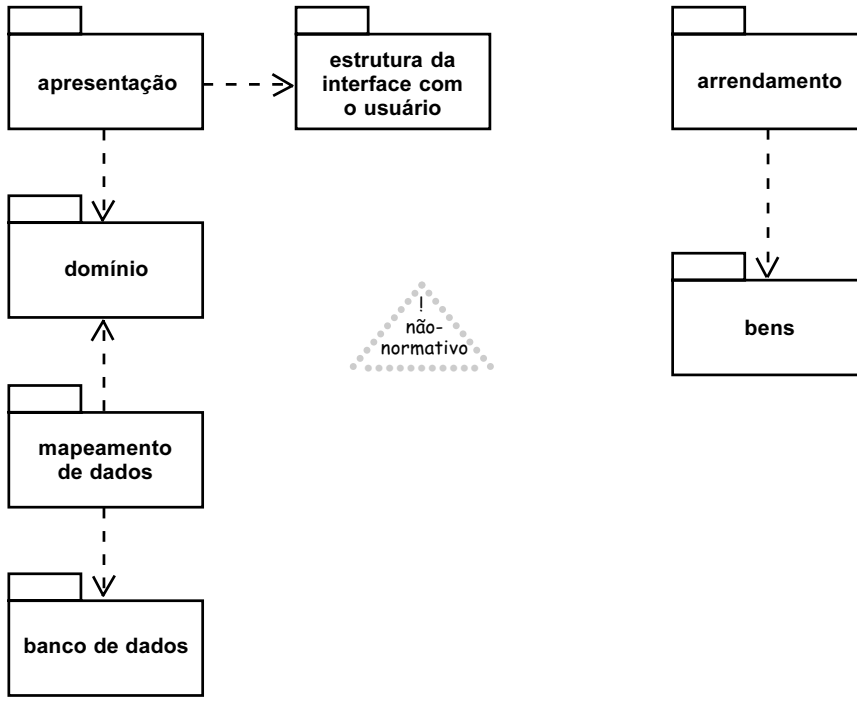


FIGURA 7.3 Separando a Figura 7.3 em dois aspectos.

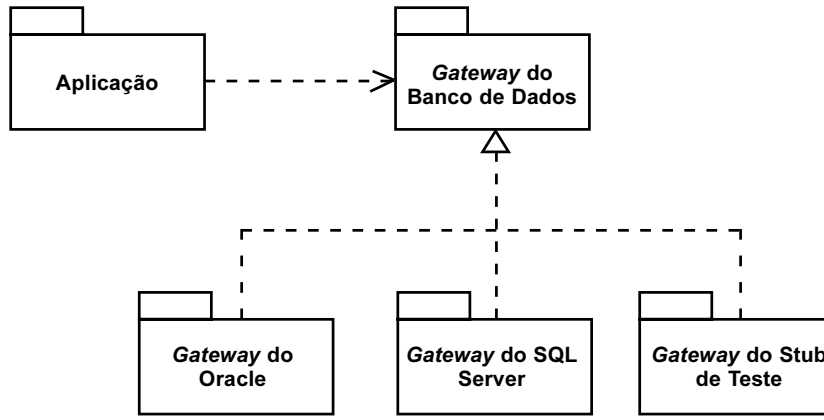


FIGURA 7.4 Um pacote implementado por outros pacotes.

um outro pacote implementar: a mesma noção de interface obrigatória que discuti na página 70.

Imagine que queiramos fornecer alguns controles de interface com o usuário (IU) para ligar e desligar coisas. Queremos que isso funcione com muitas coisas diferentes, como aquecedores e luzes. Os controles da interface com o usuário precisam executar métodos no aquecedor, mas não queremos que os controles tenham uma dependência do aquecedor. Podemos evitar essa dependência definindo, no pacote de controles, uma

interface que seja, então, implementada por qualquer classe que queira trabalhar com esses controles, como se vê na Figura 7.5. Esse é um exemplo do padrão Separated Interface [Fowler, P de EAA].

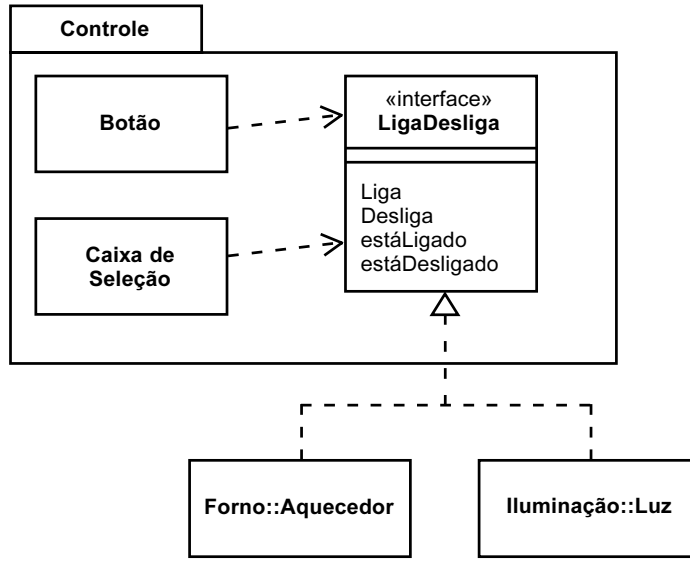


FIGURA 7.5 Definindo uma interface requerida em um pacote de cliente.

QUANDO USAR DIAGRAMAS DE PACOTES

Considero os diagramas de pacotes extremamente úteis em sistemas de grande porte, para obter uma visão das dependências entre os principais elementos de um sistema. Esses diagramas correspondem bem às estruturas usuais de programação. A representação de diagramas de pacotes e dependências o ajuda a manter as dependências de uma aplicação sob controle.

Os diagramas de pacotes representam um mecanismo de agrupamento em tempo de compilação. Para mostrar como os objetos são compostos em tempo de execução, use um diagrama de estrutura composta (página 132).

ONDE ENCONTRAR MAIS INFORMAÇÕES

A melhor discussão que conheço sobre pacotes e como utilizá-los é [Martin]. Há algum tempo, Robert Martin tem uma obsessão quase patológica com dependências e escreve muito bem sobre como prestar atenção a elas para que você possa controlá-las e minimizá-las.

Capítulo 8

Diagramas de Instalação

Os diagramas de instalação mostram o *layout* físico de um sistema, revelando quais partes do *software* são executadas em quais partes do *hardware*. Os diagramas de distribuição são muito simples; daí, a brevidade deste capítulo.

A Figura 8.1 é um exemplo simples de um diagrama de distribuição. Os itens principais do diagrama são nós conectados por caminhos de comunicação. Um **nó** é algo que pode conter algum *software*. Os nós aparecem em duas formas. Um **dispositivo** é *hardware*; ele pode ser um computador ou uma peça de *hardware* mais simples conectada a um sistema. Um **ambiente de execução** é *software* que contém a si mesmo ou contém outro *software*; exemplos desse são um sistema operacional ou um processo contêiner.

Os nós contêm **artefatos**, que são as manifestações físicas de *software*: normalmente, arquivos. Esses arquivos podem ser executáveis (como arquivos.exe, binários, DLLs, arquivos JAR, programas em linguagem *assembly* ou *scripts*) ou arquivos de dados, arquivos de configuração, documentos HTML etc. A listagem de um artefato dentro de um nó mostra que ele está instalado nesse nó do sistema que está em execução.

Você pode mostrar artefatos como caixas de classe ou listando o nome dentro de um nó. Se você os mostrar como caixas de classe, poderá adicionar um ícone de documento ou a palavra-chave «`artifact`». Você pode rotular nós ou artefatos com valores para indicar diversas informações interessantes a respeito do nó, tais como fornecedor, sistema operacional, localização ou qualquer coisa que você desejar.

Freqüentemente, você terá vários nós físicos executando a mesma tarefa lógica. Você pode mostrar isso com várias caixas de nó ou declarar o número como um valor afixado. Na Figura 8.1, usei o rótulo "`número instalado`" para indicar três servidores físicos de Web, mas não existe nenhum rótulo padrão para isso.

Muitas vezes, os artefatos são a implementação de um componente. Para mostrar isso, você pode usar um valor indicado na caixa do artefato.

Os caminhos de comunicação entre os nós indicam como as coisas se comunicam. Você pode rotular esses caminhos com informações sobre os protocolos de comunicação utilizados.

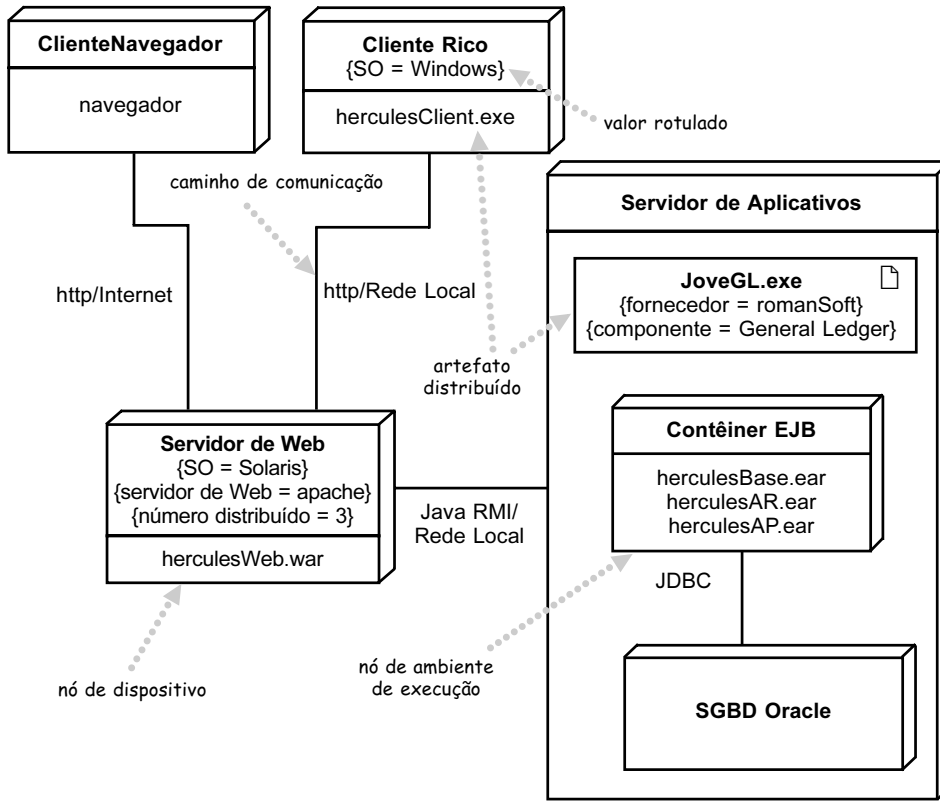


FIGURA 8.1 Exemplo de diagrama de instalação.

QUANDO USAR DIAGRAMAS DE INSTALAÇÃO

Não deixe que a brevidade deste capítulo o faça pensar que os diagramas de instalação não devem ser usados. Eles são muito úteis para mostrar o que é instalado e onde; portanto, qualquer instalação mais complicada pode fazer bom uso deles.

Capítulo 9

Casos de Uso

Os casos de uso são uma técnica para captar os requisitos funcionais de um sistema. Eles servem para descrever as interações típicas entre os usuários de um sistema e o próprio sistema, fornecendo uma narrativa sobre como o sistema é utilizado.

Em vez de descrever os casos de uso de início, acho mais fácil rodeá-los e começar descrevendo cenários. Um **cenário** é uma seqüência de passos que descreve uma interação entre um usuário e um sistema. Assim, se tivermos uma loja *on-line* baseada na Web (loja virtual), podemos ter um cenário de Compra de um Produto que diria:

> *O cliente navega no catálogo de itens e adiciona os itens desejados à sua cesta de compras. Quando o cliente deseja pagar, descreve o endereço de entrega, fornece as informações do cartão de crédito e confirma a venda. O sistema verifica a autorização do cartão de crédito e confirma a venda imediatamente e com um* e-mail *subseqüente.*

Esse cenário é uma alternativa que pode acontecer. No entanto, a autorização do cartão de crédito pode falhar, o que seria um outro cenário. Em um outro caso, você poderia ter um cliente regular de quem não precisa captar o endereço de entrega e as informações do cartão de crédito, o que seria um terceiro cenário.

Todos esses cenários são diferentes, embora semelhantes. A essência de sua similaridade é que, em todos, o usuário tem o mesmo objetivo: comprar um produto. Nem sempre ele tem sucesso, mas o objetivo permanece. Esse objetivo do usuário é a chave dos casos de uso: um **caso de uso** é um conjunto de cenários amarrados por um objetivo comum de usuário.

No jargão dos casos de uso, os usuários são referidos como atores. Um **ator** é um papel que um usuário desempenha com relação ao sistema. Os atores podem ser o cliente, representante de serviço ao cliente, gerente de vendas e analista de produto. Os atores realizam os casos de uso. Um único ator pode realizar muitos casos de uso; inversamente, um caso de uso pode ter vários atores executando-o. Normalmente, você tem muitos clientes; portanto, muitas pessoas podem ser o ator cliente. Além disso, uma pessoa pode atuar como mais de um ator, como um gerente de vendas que executa tarefas de representante de serviço ao cliente. Um ator não precisa ser um ser humano. Se o sistema realiza um serviço para outro sistema de computador, esse outro sistema é um ator.

Na realidade, *ator* não é o termo correto; *papel* seria muito melhor. Aparentemente, houve uma tradução errada do sueco; *ator* é o termo que a comunidade dos casos de uso utiliza.

Os casos de uso são reconhecidos como uma parte importante da UML. Entretanto, a surpresa é que, de muitas maneiras, a definição de casos de uso na UML é muito

rala. Nada na UML descreve como você deve capturar o conteúdo de um caso de uso. O que a UML descreve é um diagrama de casos de uso, que mostra como utilizar casos relacionados entre si. Mas praticamente todo o valor dos casos de uso reside no conteúdo e o diagrama é de valor bastante limitado.

CONTEÚDO DE UM CASO DE USO

Não existe nenhuma maneira padronizada para escrever o conteúdo de um caso de uso e diferentes formatos funcionam bem em diferentes casos. A Figura 9.1 mostra um estilo comum de uso. Você começa escolhendo um dos cenários como sendo o **cenário principal de sucesso (CPS)**. Dá início ao corpo do caso de uso escrevendo o cenário principal de sucesso como uma seqüência de passos numerados. Então, pega os outros cenários e os escreve como **extensões**, descrevendo-os em termos de variações em relação ao cenário principal de sucesso. As extensões podem ser bem-sucedidas – o usuário atinge o objetivo, como em 3a – ou falhas, como em 6a.

Cada caso de uso tem um ator principal, que pede ao sistema para que execute um serviço. O ator principal é aquele cujo objetivo o caso de uso está tentando satisfazer e, normalmente (mas nem sempre) é o iniciador do caso de uso. Podem existir outros atores com os quais o sistema se comunica enquanto executa o caso de uso. Eles são conhecidos como atores secundários.

Cada passo em um caso de uso é um elemento da interação entre um ator e o sistema. Cada passo deve ser uma declaração simples e mostrar claramente quem está executando o passo. O passo deve mostrar a intenção do ator e não os mecanismos do que o ator faz. Conseqüentemente, você não descreve a interface com o usuário no caso de uso. Na verdade, a escrita do caso de uso normalmente precede o projeto da interface com o usuário.

Uma extensão dentro do caso de uso nomeia uma condição que resulta em diferentes interações daquelas descritas no cenário principal de sucesso e informa quais são essas

Compra de um Produto

Nível do Objetivo: Nível do Mar

Cenário Principal de Sucesso:

1. O cliente navega pelo catálogo e seleciona itens para comprar
2. O cliente vai para o caixa
3. O cliente preenche o formulário da remessa (endereço de entrega; opção de entrega imediata ou em três dias)
4. O sistema apresenta a informação completa do faturamento, incluindo a remessa
5. O cliente preenche a informação de cartão de crédito
6. O sistema autoriza a compra
7. O sistema confirma imediatamente a venda
8. O sistema envia uma confirmação para o cliente por *e-mail*

Extensões:

3a: Cliente regular
.1: O sistema mostra a informação atual da remessa, a informação de preço e a informação de cobrança
.2: O cliente pode aceitar ou escrever por cima desses padrões, retornando ao CPS, no passo 6

6a. O sistema falha na autorização da compra a crédito
.1: O cliente pode inserir novamente a informação do cartão de crédito ou cancelar

Figura 9.1 Exemplo de texto de caso de uso.

diferenças. Inicie a extensão dando um nome ao passo em que a condição é detectada e forneça uma breve descrição da condição. Após a condição, coloque passos numerados, no mesmo estilo que o do cenário principal de sucesso.

Conclua esses passos descrevendo onde você volta para o cenário principal de sucesso, caso volte.

A estrutura do caso de uso é uma excelente maneira de criar alternativas ao cenário principal de sucesso. Para cada passo, pergunte: como isso poderia ser feito de uma forma diferente? E em particular, pergunte: o que poderia dar errado? Normalmente é melhor pensar em todas as condições de extensão primeiro, antes de se aprofundar na solução das conseqüências. Dessa maneira, você provavelmente considerará mais condições, o que se traduz em menos erros a serem capturados posteriormente.

Um passo complicado em um caso de uso pode ser um outro caso de uso. Em termos de UML, dizemos que o primeiro caso de uso **inclui** o segundo. Não existe nenhuma maneira padronizada para mostrar no texto um caso de uso incluído, mas acho que o sublinhado, que sugere um elo de hipertexto, funciona muito bem e, em muitas ferramentas, será realmente isso. Assim, na Figura 9.1, o primeiro passo inclui o caso de uso "navega pelo catálogo e seleciona itens para comprar".

Os casos de uso incluídos podem ser úteis em um passo complexo que congestionaria o cenário principal ou em passos que são repetidos em vários casos de uso. No entanto, não tente subdivir os casos de uso em sub-casos de uso e sub-sub-casos de uso, utilizando decomposição funcional. Tal decomposição é uma boa maneira de perder muito tempo.

Assim como acontece com os passos nos cenários, você pode adicionar algumas outras informações comuns a um caso de uso.

- Uma **pré-condição** descreve o que o sistema deve garantir como verdadeiro, antes de permitir que o caso de uso comece. Isso é útil para dizer aos programadores quais condições eles não precisam verificar no código.
- Uma **garantia** descreve o que o sistema irá assegurar no final do caso de uso. As garantias de sucesso se mantêm após um cenário bem-sucedido; as garantias mínimas se mantêm após qualquer cenário.
- Um **gatilho** especifica o evento que inicia o caso de uso.

Quando você estiver considerando a adição de elementos, seja cético. É melhor fazer muito pouco do que fazer demais. Além disso, faça o máximo para manter o caso de uso breve e fácil de ler. Descobri que os casos de uso longos e detalhados não são lidos, o que anula seu objetivo.

O volume de detalhes necessário em um caso de uso depende do risco nesse caso de uso. Freqüentemente, você precisa de detalhes apenas no início de alguns poucos casos de uso importantes; os outros podem ser considerados imediatamente antes de você implementá-los. Você não precisa escrever todos os detalhes; a comunicação verbal é freqüentemente muito eficaz, particularmente dentro de um ciclo iterativo em que as necessidades são rapidamente atendidas por meio da execução do código.

DIAGRAMAS DE CASOS DE USO

Conforme eu disse anteriormente, a UML nada diz sobre o conteúdo de um caso de uso, mas fornece um formato de diagrama para mostrá-lo, como se vê na Figura 9.2. Embora

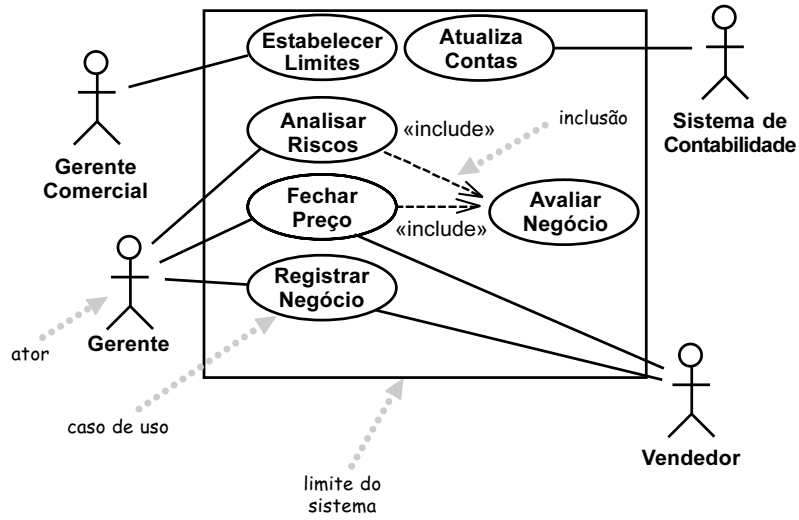


FIGURA 9.2 Diagrama de casos de uso.

o diagrama às vezes seja útil, ele não é obrigatório. Em seu trabalho com casos de uso, não se esmere muito no diagrama. Em vez disso, concentre-se no conteúdo textual dos casos de uso.

A melhor maneira de pensar um diagrama de caso de uso é como um sumário gráfico do conjunto de casos de uso. Ele também é semelhante ao diagrama de contexto usado nos métodos estruturados, pois mostra o limite do sistema e as interações com o mundo exterior. O diagrama de casos de uso mostra os atores, os casos de uso e os relacionamentos entre eles:

- Quais atores realizam quais casos de uso
- Quais casos de uso incluem outros casos de uso

A UML inclui outros relacionamentos entre os casos de uso, além da inclusão simples, como «extend». Sugiro que você os ignore. Tenho visto muitas situações em que as equipes podem ficar terrivelmente atrasadas ao usar diferentes relacionamentos de caso de uso e muita energia é desperdiçada. Em vez disso, concentre-se na descrição textual de um caso de uso; é aí que reside o valor real da técnica.

NÍVEIS DE CASOS DE USO

Um problema comum que pode acontecer com casos de uso é que, concentrando-se na interação entre um usuário e o sistema, você pode negligenciar situações nas quais uma mudança no processo do negócio pode ser a melhor maneira de lidar com o problema. Freqüentemente, as pessoas falam sobre casos de uso de sistema e casos de uso de negócio. Os termos não são precisos, mas o uso geral é que um **caso de uso de sistema** é uma interação com o *software*, enquanto um **caso de uso de negócio** examina como a aplicação responde ao cliente ou a um evento.

[Cockburn, use cases] sugere um esquema de níveis de casos de uso. Os casos de uso básicos estão "no nível do mar". Normalmente, os casos de uso no **nível do mar** representam uma interação distinta entre um ator principal e o sistema. Tais casos de uso transmitirão algo de valor para o ator principal e, normalmente, este levará de alguns minutos a meia hora para terminar. Os casos de uso existentes apenas porque foram incluídos pelos casos de uso de nível do mar estão **em nível de peixe**. Os casos de uso de nível mais alto (**em nível de pássaro**) mostram como os casos de uso de nível do mar se encaixam nas interações do negócio mais amplas. Os casos de uso em nível de pássaro normalmente são casos de uso de negócio, enquanto os casos de níveis do mar e de peixe são casos de uso de sistema. Você deverá ter a maioria de seus casos de uso em nível do mar. Eu prefiro indicar o nível no início do caso de uso, como na Figura 9.1.

CASOS DE USO E FUNCIONALIDADES (OU HISTÓRIAS)

Muitas estratégias utilizam as funcionalidades de um sistema – a Extreme Programming os chama de histórias de usuário – para ajudar a descrever requisitos. Uma questão comum é como funcionalidades e casos de uso se correlacionam.

As funcionalidades constituem uma boa maneira de repartir um sistema para planejar um projeto iterativo, pelo qual cada iteração implementa várias funcionalidades. Os casos de uso fornecem uma narrativa de como os atores utilizam o sistema. Então, embora as duas técnicas descrevam requisitos, seus propósitos são diferentes.

Embora você possa passar diretamente à descrição das funcionalidades, muitas pessoas acham interessante desenvolver primeiro os casos de uso e depois gerar uma lista de funcionalidades. Um recurso pode ser um caso de uso inteiro, um cenário em um caso de uso, um passo em um caso de uso ou algum comportamento variante, como a adição de um outro método de depreciação para suas avaliações de bens, que não apareça em uma narrativa de caso de uso. Normalmente, os recursos acabam sendo mais refinados do que os casos de uso.

QUANDO UTILIZAR CASOS DE USO

Os casos de uso são uma ferramenta valiosa para ajudar no entendimento dos requisitos funcionais de um sistema. Uma primeira passagem nos casos de uso deve ser feita no início. Versões mais detalhadas dos casos de uso devem ser elaboradas apenas antes do desenvolvimento desse caso de uso.

É importante lembrar que casos de uso representam uma visão *externa* do sistema. Como tal, não espere quaisquer correlações entre eles e as classes dentro do sistema.

Quanto mais eu observo os casos de uso, menos valioso parece ser o diagrama de casos de uso. Com os casos de uso, você concentra sua energia no texto e não no diagrama. A despeito do fato de que a UML nada tem a dizer sobre o texto do caso de uso, é esse texto que contém todo o valor da técnica.

Um grande perigo dos casos de uso é que as pessoas os tornam complicados demais e não conseguem prosseguir. Normalmente, você terá menos problemas fazendo pouco do que fazendo demais. Umas duas páginas por caso de uso está bom, para a maioria das situações. Se você tiver muito pouco, pelo menos terá um documento curto e legível,

que será um ponto de partida para perguntas. Se você tiver demais, dificilmente alguém o lerá e o entenderá.

ONDE ENCONTRAR MAIS INFORMAÇÕES

Os casos de uso foram popularizados originalmente por Ivar Jacobson [Jacobson, OOSE].

Embora os casos de uso já existam há algum tempo, havia pouca padronização para seu uso. A UML nada diz sobre o importante conteúdo de um caso de uso e tem padronizado apenas os diagramas, muito menos importantes. Como resultado, você pode encontrar uma variedade de opiniões divergentes a respeito dos casos de uso.

Nos últimos anos, entretanto, [Cockburn, use cases] tornou-se o livro padrão sobre o assunto. Neste capítulo, segui a terminologia e as recomendações desse livro, pelo excelente motivo de que, quando discordamos no passado, no final acabei aceitando a opinião de Alistair Cockburn. Ele também mantém uma página na Web, no endereço **http://usecases.org**. [Constantine e Lockwood] fornecem um convincente processo para derivar interfaces com o usuário a partir de casos de uso; veja também o endereço **http://foruse.com**.

Capítulo 10

Diagramas de Máquina de Estados

Os **diagramas de máquina de estados** são uma técnica conhecida para descrever o comportamento de um sistema. Existem várias formas de diagramas de estados desde os anos 60 e as mais antigas técnicas orientadas a objetos os adotaram para mostrar comportamento. Nas estratégias orientadas a objetos, você desenha um diagrama de máquina de estados para uma única classe, para mostrar o comportamento do ciclo de vida de um único objeto.

Quando as pessoas escrevem sobre máquina de estados, inevitavelmente os exemplos são controles de navegação ou máquina de vendas. Como estou um pouco enjoado desses exemplos, decidi usar um controlador para um painel secreto em um castelo gótico. Nesse castelo, quero manter meus objetos de valor em um cofre que seja difícil de encontrar. Assim, para revelar o cadeado do cofre, tenho que remover uma vela estratégica de seu castiçal, mas isso mostrará o cadeado somente enquanto a porta estiver fechada. Uma vez que eu consiga ver o cadeado, posso inserir minha chave para abrir o cofre. Como uma medida de segurança extra, garanto que só posso abrir o cofre se primeiro substituir a vela. Se um ladrão se esquecer dessa precaução, eu soltarei um monstro asqueroso para devorá-lo.

A Figura 10.1 mostra um diagrama de máquina de estados da classe do controlador que dirige meu sistema de segurança incomum. O diagrama de estados começa com o estado do objeto controlador, quando ele é criado: na Figura 10.1, o estado Esperar. O diagrama indica isso com o **pseudo-estado inicial**, que não é um estado, mas tem uma seta que aponta para o estado inicial.

O diagrama mostra que o controlador pode estar em três estados: Esperar, Trancar e Abrir. Ele também fornece as regras por meio das quais o controlador muda de um estado para outro. Essas regras estão na forma de transições: as linhas que conectam os estados.

A **transição** indica um movimento de um estado para outro. Cada transição tem um rótulo que possui três partes: `assinatura-do-gatilho [sentinela]/atividade`. Todas as partes são opcionais. A `assinatura-do-gatilho` normalmente é um único evento que dispara uma mudança de estado em potencial. A `sentinela`, se estiver presente, é uma condição booleana que deve ser verdadeira para que a transição ocorra. A `atividade` é algum comportamento executado durante a transição. Pode ser qualquer expressão comportamental. A forma completa de uma `assinatura-do-gatilho` pode incluir múltiplos eventos e parâmetros. Assim, na Figura 10.1, você lê a transição externa do estado Esperar como: "No estado Esperar, se a vela for removida na condição em que a porta esteja aberta, você revelará o cadeado e mudará para o estado Trancar".

Todas as três partes de uma transição são opcionais. Uma atividade ausente indica que você não faz nada durante a transição. Uma sentinela ausente indica que você sem-

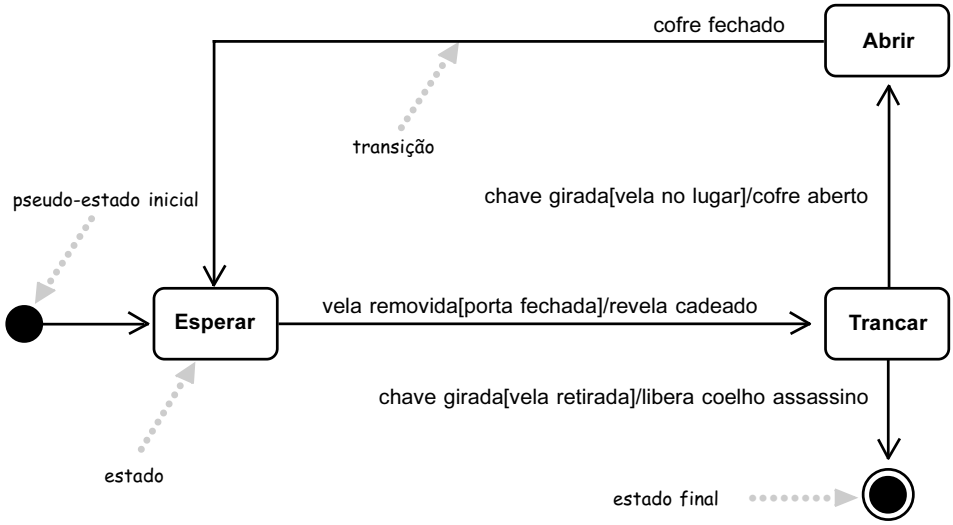


FIGURA 10.1 Um diagrama simples de máquina de estados.

pre faz a transição, caso o evento ocorra. Uma assinatura-do-gatilho ausente é rara, mas ocorre. Isso indica que a transição é feita imediatamente, o que você vê principalmente com estados de atividade, que veremos daqui há pouco.

Quando ocorre um evento em um estado, você pode fazer apenas uma transição a partir dele. Assim, se você usar múltiplas transições com o mesmo evento, como no caso do estado Trancar da Figura 10.1, as sentinelas deverão ser mutuamente exclusivas. Se ocorrer um evento e nenhuma transição for válida – por exemplo, um evento de cofre fechado no estado Esperar ou um evento de vela removida com a porta fechada – , o evento será ignorado.

O estado final indica que a máquina de estados está completa, implicando a exclusão do objeto controlador. Assim, se alguém for tão descuidado para cair em minha armadilha, o objeto controlador terminará, de modo que precisarei colocar o coelho em sua jaula, limpar o chão e reiniciar o sistema.

Lembre-se de que a máquina de estados pode mostrar apenas o que o objeto observa ou ativa diretamente. Assim, embora você possa esperar que eu coloque ou retire coisas do cofre quando ele estiver aberto, não coloco isso no diagrama de estados, pois o controlador não poderá identificar.

Quando os desenvolvedores falam sobre objetos, eles freqüentemente se referem ao estado dos objetos como o significado da combinação de todos os dados presentes nos campos dos objetos. Entretanto, o estado em um diagrama de máquina de estados é uma noção mais abstrata; basicamente, diferentes estados implicam maneiras diferentes de reagir aos eventos.

ATIVIDADES INTERNAS

Os estados podem reagir a eventos sem transição, usando **atividades internas**: colocar o evento, a sentinela e a atividade dentro da própria caixa de estado.

A Figura 10.2 mostra um estado com atividades internas do caractere e eventos de ajuda, tal como você poderia encontrar em um campo de texto de uma interface com o usuário. Uma atividade interna é semelhante a uma **autotransição**: uma transição que volta para o mesmo estado. A sintaxe das atividades internas segue a mesma lógica do evento, da sentinela e do procedimento.

A Figura 10.2 também mostra duas atividades especiais: as atividades de entrada e de saída. A **atividade de entrada** é executada quando você entra em um estado; a **atividade de saída** é executada quando você sai dele. Entretanto, as atividades internas não disparam as atividades de entrada e saída; essa é a diferença entre as atividades internas e as autotransições.

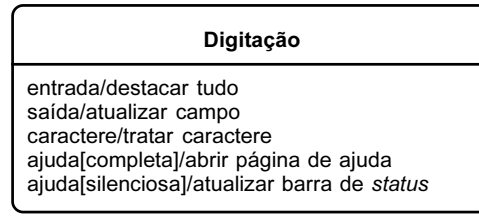


FIGURA 10.2 Eventos internos mostrados com o estado de digitação de um campo de texto.

ESTADOS DE ATIVIDADES

Nos estados que descrevi até aqui, o objeto está quieto e esperando pelo próximo evento, antes de fazer alguma coisa. No entanto, você pode ter estados nos quais o objeto está realizando algum trabalho.

O estado Pesquisar, na Figura 10.3, é um **estado de atividade**: a atividade em andamento é identificada com `realizar`/; daí, o termo **realizar-atividade**. Uma vez terminada a pesquisa, todas as transições sem atividade, como aquela para exibir o novo *hardware*, são realizadas. Se ocorrer o evento cancelar durante a atividade, a atividade realizar-atividade será interrompida sem cerimônia e voltaremos ao estado Atualizar Janela de *Hardware*.

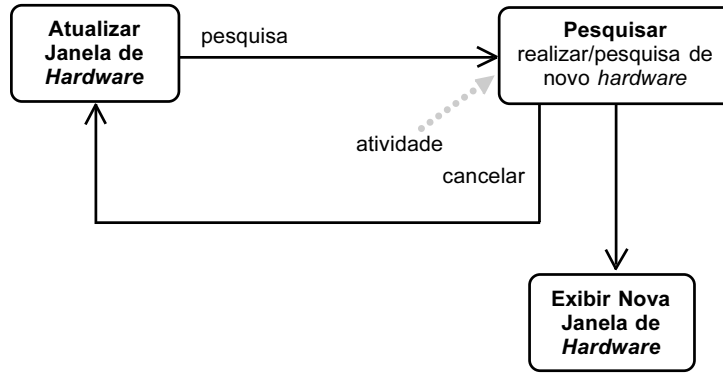


FIGURA 10.3 Um estado com uma atividade.

Tanto realizar-atividades como as atividades normais representam a execução de algum comportamento. A diferença fundamental entre as duas é que as atividades normais ocorrem "instantaneamente" e não podem ser interrompidas por eventos regulares, enquanto as do tipo realizar-atividades podem demorar um tempo finito e podem ser interrompidas, como na Figura 10.3. O termo instantaneamente significará diferentes coisas para sistemas distintos; para sistemas de *hardware* em tempo real, poderia significar algumas instruções de máquina, mas para *software* executando localmente poderia ser vários segundos.

A UML 1 usava o termo **ação** para atividades normais e usava atividades somente para a modalidade realizar-atividades.

SUPERESTADOS

Freqüentemente, você verá que vários estados compartilham transições e atividades internas comuns. Nesses casos, você pode transformá-los em subestados e mover o comportamento compartilhado para um superestado, como na Figura 10.4. Sem o superestado, você teria que desenhar uma transição de cancelamento para todos os três estados dentro do estado Inserir Detalhes da Conexão.

ESTADOS CONCORRENTES

Os estados podem ser divididos em vários diagramas de estados ortogonais que executam concorrentemente. A Figura 10.5 mostra um despertador absurdamente simples, que pode tocar CDs ou ligar o rádio e mostrar a hora atual ou a hora de alarme.

As escolhas CD/rádio e hora atual/de alarme são ortogonais. Se você quisesse representar isso com um diagrama de estados não-ortogonal, precisaria de um diagrama confuso, que ficaria fora de controle se fossem necessários mais estados. Separar as duas áreas de comportamento em diagramas de estados distintos torna isso muito mais claro.

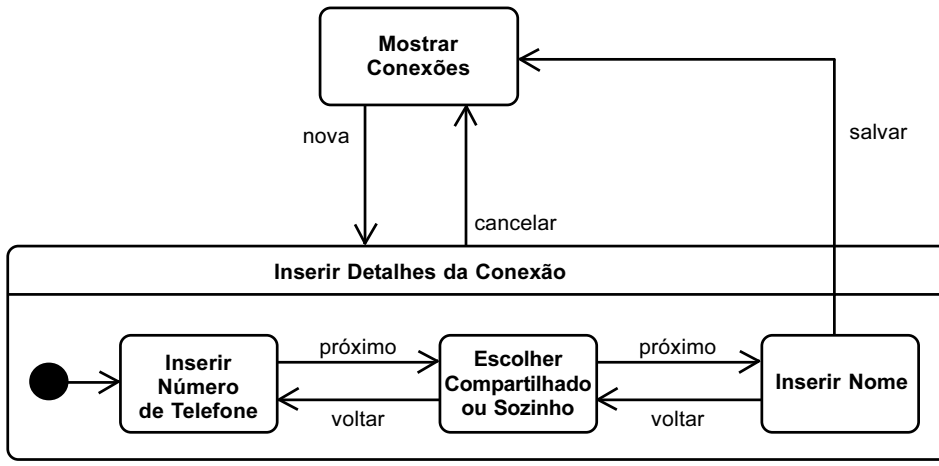


FIGURA 10.4 Superestado com subestados aninhados.

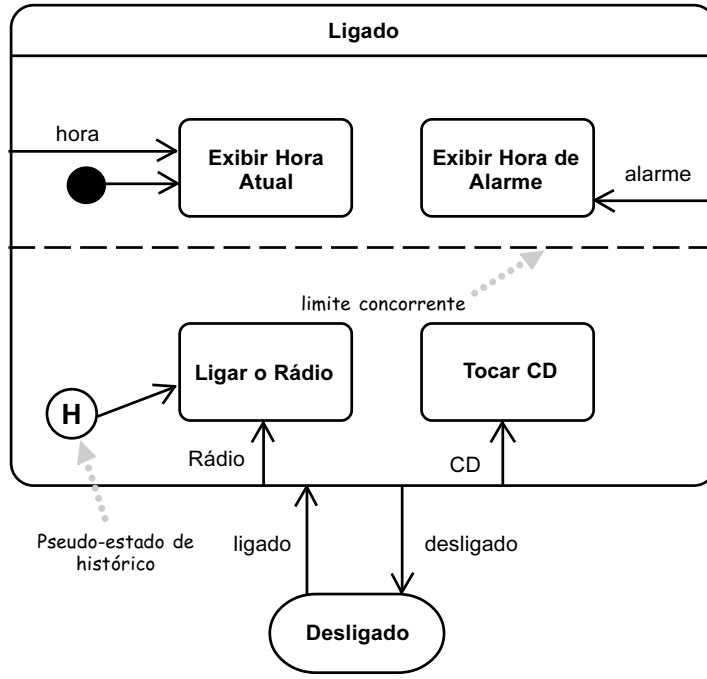


FIGURA 10.5 Estados concorrentes ortogonais.

A Figura 10.5 também inclui um **pseudo-estado de histórico**. Isso indica que, quando o despertador é ligado, a escolha de rádio/CD volta para o estado em que o despertador estava quando foi desligado. A seta do pseudo-estado de histórico indica em qual estado estará na primeira vez, quando ainda não houver histórico.

COMO IMPLEMENTAR DIAGRAMAS DE ESTADOS

Um diagrama de estados pode ser implementado de três maneiras principais: comando *switch* aninhado, o padrão State e tabelas de estado. A estratégia mais direta para lidar com um diagrama de estados é por meio de uma instrução *switch* aninhada, como se vê na Figura 10.6. Embora seja direta, ela é cansativa, mesmo para este caso simples. Essa estratégia também sai muito facilmente de controle, de modo que não gosto de usá-la, mesmo para casos simples.

O **padrão State** [Gangue dos Quatro] cria uma hierarquia de classes de estado para manipular o comportamento dos estados. Cada estado no diagrama tem uma subclasse de estado. O controlador tem métodos para cada evento, os quais simplesmente encaminham para a classe do estado. O diagrama de estados da Figura 10.1 produziria uma implementação indicada pelas classes da Figura 10.7.

O topo da hierarquia é uma classe abstrata que implementa todos os métodos de tratamento de eventos para não fazer nada. Para cada estado concreto, você simplesmente reescreve os métodos de eventos específicos para os quais o estado possui transições.

A estratégia da **tabela de estados** captura as informações do diagrama de estados como dados. Assim, a Figura 10.1 poderia acabar sendo representada como a Tabela 10.1.

```
public void TratarEvento (EventodePainel umEvento) {
   switch (EstadoCorrente) {
      case EstadodoPainel.Abrir:
         switch (umEvento) {
            case EventodePainel.CofreFechado:
               EstadoCorrente = EstadodoPainel.Esperar
               break;
         }
         break;
      case EstadodoPainel.Esperar:
         switch (umEvento) {
            case EventodePainel.VelaRemovida:
               if (aPortaEstáAberta) {
                  RevelaCadeado();
                  EstadoCorrente = EstadodoPainel.Trancar;
               }
               break;
         }
         break;
      case EstadodoPainel.Trancar:
         switch (umEvento) {
            case EventodePainel.ChaveGirada:
               if (aVelaEstáPresente) {
                  AbrirCofre();
                  EstadoCorrente = EstadodoPainel.Abrir;
               } else {
                  SoltarCoelhoAssassino();
                  EstadoCorrente = EstadodoPainel.Final;
               }
               break;
         }
         break;
   }
  }
}
```

FIGURA 10.6 Uma instrução *switch* aninhada da linguagem C# para manipular a transição de estados da Figura 10.1.

Construímos, então, um interpretador que usa a tabela de estados em tempo de execução ou um gerador de código que gera classes com base na tabela de estados.

Obviamente, a tabela de estados dá mais trabalho para fazer, mas então você pode usá-la sempre que tiver um problema de estados para resolver. Uma tabela de estados de tempo de execução também pode ser modificada sem recompilação, o que é muito útil em alguns contextos. O padrão State é mais fácil construir, quando você precisa dele e, embora ele necessite de uma nova classe para cada estado, trata-se de um pequeno volume de código a ser escrito em cada caso.

Essas implementações são mínimas, mas devem dar a você uma idéia de como proceder na implementação de diagramas de estados. Em cada caso, a implementação de modelos de estado leva a um código bastante padronizado; portanto, normalmente é melhor usar alguma forma de geração de código para fazê-lo.

QUANDO UTILIZAR DIAGRAMAS DE ESTADOS

Os diagramas de estados são bons para descrever o comportamento de um objeto por intermédio de vários casos de uso. No entanto, esses diagramas não são muito bons para

descrever um comportamento que envolva vários objetos em colaboração. Para tal, é útil combinar diagramas de estados com outras técnicas. Por exemplo, os diagramas de interação (veja o Capítulo 4) são bons para descrever o comportamento de vários objetos em um único caso de uso e os diagramas de atividades (veja Capítulo 11) são bons para mostrar a seqüência geral de atividades para vários objetos e casos de uso.

Nem todo mundo considera que os diagramas de estados sejam naturais. Fique atento ao modo como as pessoas trabalham com eles. Pode ser que sua equipe não considere os diagramas de estados úteis para seu modo de trabalhar. Isso não é um grande problema; como sempre, você deve lembrar de usar uma combinação das técnicas que funcionem para seu caso.

Se você utilizar diagramas de estados, não tente desenhá-los para cada classe presente no sistema. Embora essa estratégia seja freqüentemente utilizada por perfeccionistas de muita formalidade, ela é quase sempre um desperdício de trabalho. Use diagramas de estados somente para aquelas classes que exibem comportamento interessante, para as quais a construção do diagrama de estados ajude a compreender o que está acontecendo. Muitas pessoas acreditam que a interface com o usuário e os objetos de controle têm o tipo de comportamento que é útil representar com um diagrama de estados.

TABELA 10.1 Uma tabela de estados para a Figura 10.1

Estado de Origem	Estado de Destino	Evento	Sentinela	Procedimento
Esperar	Trancar	Vela removida	Porta aberta	Revelar cadeado
Trancar	Abrir	Chave girada	Vela presente	Abrir cofre
Trancar	Final	Chave girada	Vela removida	Soltar coelho assassino
Abrir	Esperar	Cofre fechado		

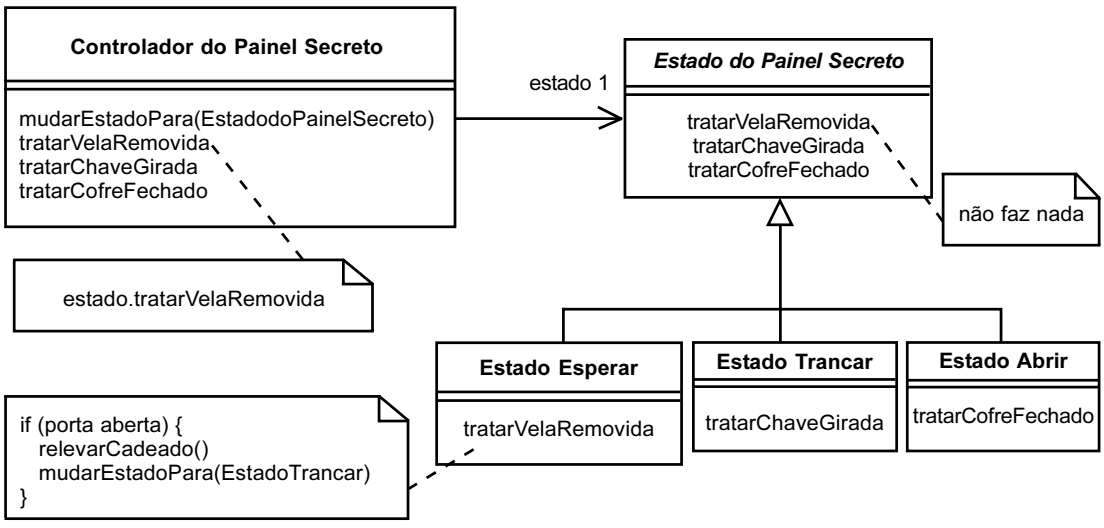


FIGURA 10.7 Uma implementação de padrão de estado para a Figura 10.1.

ONDE ENCONTRAR MAIS INFORMAÇÕES

Tanto o *User Guide* [Booch, UML user] como o *Reference Manual* [Rumbaugh, UML Reference] têm mais informações sobre diagramas de estados. Os projetistas de tempo real tendem a usar muito os modelos de estado; portanto, não é de surpreender que [Douglass] tenha muito a dizer sobre diagramas de estados, incluindo informações sobre como implementá-los. [Martin] contém um capítulo muito bom sobre as várias maneiras de implementar diagramas de estados.

Capítulo 11

Diagramas de Atividades

Os diagramas de atividades são uma técnica para descrever lógica de procedimento, processo de negócio e fluxo de trabalho. De várias formas, eles desempenham um papel semelhante aos fluxogramas, mas a principal diferença entre eles e a notação de fluxograma é que os diagramas suportam comportamento paralelo.

Os diagramas de atividades têm recebido as maiores modificações nas diversas versões da UML; portanto, não surpreende que tenham sido significativamente estendidos e novamente alterados para a UML 2. Na UML 1, os diagramas de atividades eram vistos como casos especiais dos diagramas de estados. Isso causou muitos problemas para os modeladores de fluxos de trabalho, para os quais os diagramas de atividades eram muito convenientes. Na UML 2, esse vínculo foi eliminado.

A Figura 11.1 mostra um exemplo simples de diagrama de atividades. Começamos na ação do **nó inicial** e depois executamos a ação Receber Pedido. Uma vez feito isso, encontramos uma separação. Uma **separação** tem um fluxo de entrada e vários fluxos concorrentes de saída.

A Figura 11.1 diz que Preencher Pedido, Enviar Fatura e as ações subseqüentes ocorrem em paralelo. Basicamente, isso significa que a seqüência entre elas é irrelevante. Eu poderia preencher o pedido, enviar a fatura, entregar e depois receber o pagamento; ou então, poderia enviar a fatura, receber o pagamento, preencher o pedido e depois entregar: você entende a situação.

Essas atividades também podem ser executadas intercaladamente. Por exemplo, pego o primeiro item de linha do depósito, escrevo a fatura, pego o segundo item de linha, coloco a fatura em um envelope e assim por diante. Ou então, poderia fazer alguma dessas coisas simultaneamente: escrever a fatura com uma das mãos, enquanto pego o próximo item com a outra. Qualquer uma dessas seqüências está correta, de acordo com o diagrama.

O diagrama de atividades permite que quem está seguindo o processo escolha a ordem na qual fazer as coisas. Em outras palavras, ele simplesmente determina as regras essenciais de seqüência que se deve seguir. Isso é importante para modelagem de negócios, pois os processos freqüentemente ocorrem em paralelo. Isso também é útil para algoritmos concorrentes, nos quais linhas de execução independentes podem fazer coisas em paralelo.

Quando você tem comportamento paralelo, precisa sincronizar. Não fechamos um pedido até que ele seja entregue e pago. Mostramos isso com a **junção**, antes da ação Fechar Pedido. Com uma junção, o fluxo de saída é executado somente quando todos os fluxos de entrada chegarem à junção. Assim, você somente pode fechar o pedido quando tiver recebido o pagamento e entregue o pedido.

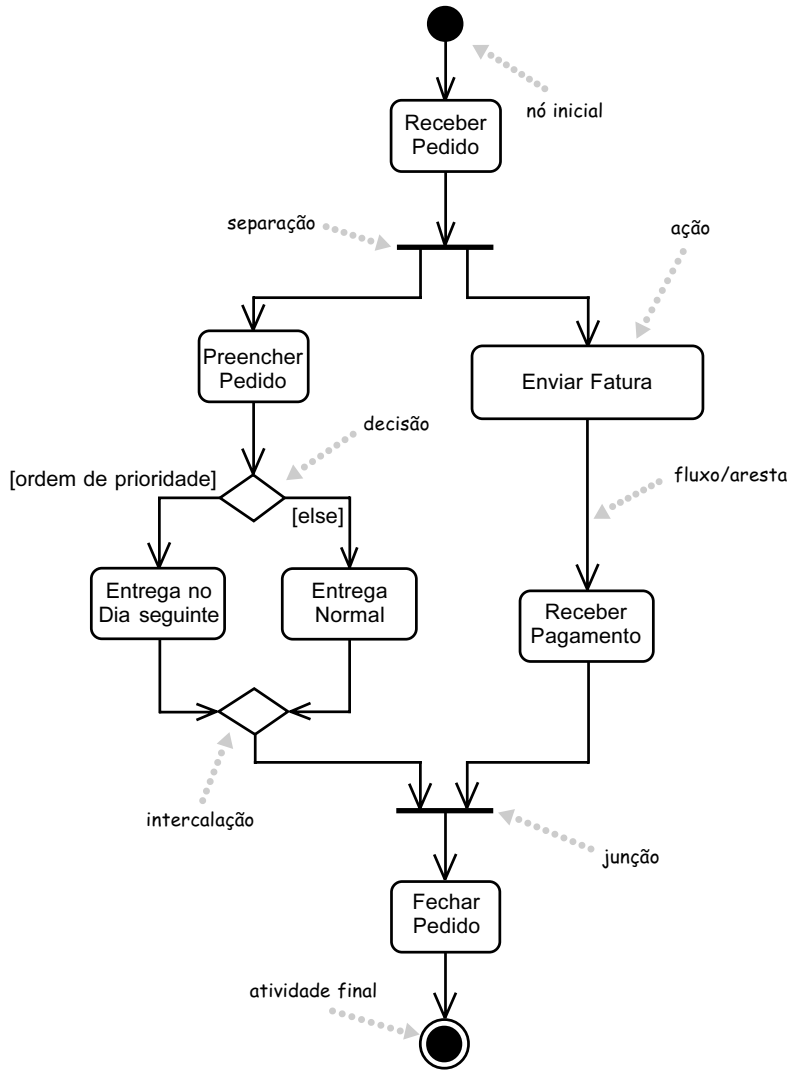


FIGURA 11.1 Um diagrama de atividades simples.

A UML 1 tinha regras particulares para harmonizar separações e junções, pois os diagramas de atividades eram casos especiais dos diagramas de estados. Na UML 2, tal harmonização não é mais necessária.

Você notará que os nós em um diagrama de atividades são chamados de ações e não de atividades. Rigorosamente falando, uma atividade se refere a uma seqüência de ações; portanto, o diagrama mostra uma atividade constituída de ações.

O comportamento condicional é delineado por decisões e intercalações. Uma **decisão**, denominada *desvio* na UML 1, tem um único fluxo de entrada e vários fluxos de saída vigiados. Cada fluxo de saída tem uma sentinela: uma condição booleana colocada entre colchetes. Sempre que você chega em uma decisão, pode seguir apenas um dos fluxos de saída, de modo que as sentinelas devem ser mutuamente exclusivas. A utiliza-

ção de [else] como sentinela indica que o fluxo [else] deve ser usado, caso todas as outras sentinelas da decisão sejam falsas.

Na Figura 11.1, depois que um pedido é preenchido, há uma decisão. Se você tiver um pedido urgente, fará uma Entrega no Dia Seguinte; caso contrário, fará uma Entrega Normal.

Uma **intercalação** tem vários fluxos de entrada e uma única saída. Uma intercalação marca o final de um comportamento condicional iniciado por uma decisão.

Em meus diagramas, cada ação tem um único fluxo de entrada e um único fluxo de saída. Na UML 1, múltiplos fluxos de entrada tinham uma intercalação implícita. Isto é, sua ação seria executada se qualquer fluxo fosse disparado. Na UML 2, isso mudou, de modo que, em vez disso, há uma junção implícita; assim, a ação é executada somente se todos os fluxos dispararem. Como resultado dessa mudança, recomendo que você use apenas um fluxo de entrada e um fluxo de saída para uma ação e mostre todas as junções e intercalações explicitamente; isso evitará confusão.

COMO DECOMPOR UMA AÇÃO

As ações podem ser decompostas em sub-atividades. Posso pegar a lógica de entrega da Figura 11.1 e defini-la como sua própria atividade (Figura 11.2). Então, posso chamá-la como uma ação (Figura 11.3, na página 121).

As ações podem ser implementadas como sub-atividades ou como métodos nas classes. Você pode mostrar uma sub-atividade utilizando o símbolo de ancinho. Uma chamada de método pode ser mostrada com a sintaxe `nome-da-classe::nome-do-método`. Você também pode escrever um fragmento de código no símbolo de ação, se o comportamento executado não for uma única chamada de método.

PARTIÇÕES

Os diagramas de atividades dizem o que acontece, mas não dizem quem faz o que. Em programação, isso significa que o diagrama não comunica qual classe é responsável por cada ação. Na modelagem do processo de negócio, isso não comunica qual parte de uma organização executa qual ação. Isso não é necessariamente um problema; freqüentemente, faz sentido se concentrar no que é feito, em vez de em quem realiza quais partes do comportamento.

Se você quiser mostrar quem faz o que, pode dividir um diagrama de atividades em **partições**, que mostram quais ações uma classe ou unidade da organização executa. A Figura 11.4 (na página 122) tem um exemplo simples disso, mostrando como as ações envolvidas no processamento do pedido podem ser separadas entre vários departamentos.

O particionamento da Figura 11.4 é unidimensional simples. Esse estilo é freqüentemente referido como **raias de piscina**, por motivos óbvios, e era a única forma usada na UML 1.x. Na UML 2, você pode usar uma grade bidimensional; portanto, a metáfora da raia de piscina não vale mais. Você também pode pegar cada dimensão e dividir as linhas e colunas hierarquicamente.

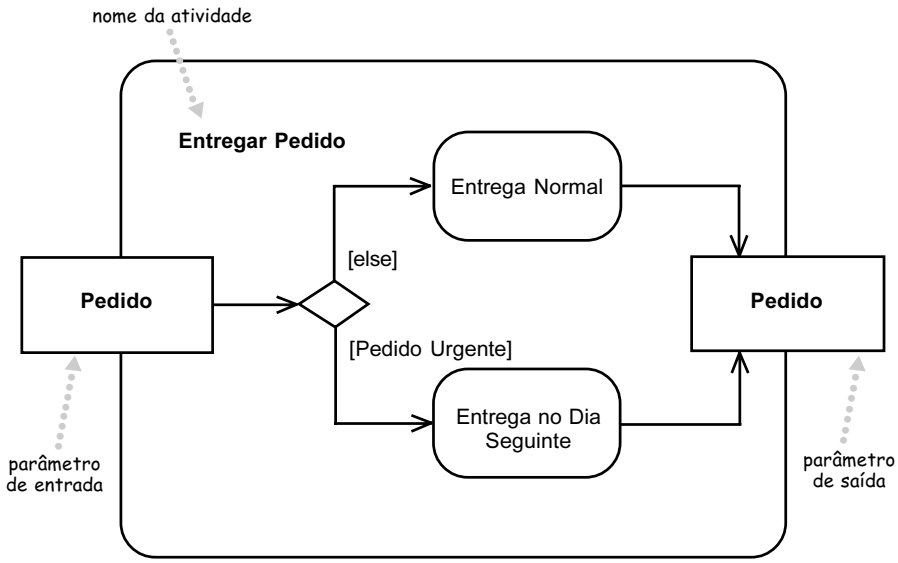


FIGURA 11.2 Um diagrama de atividades auxiliar.

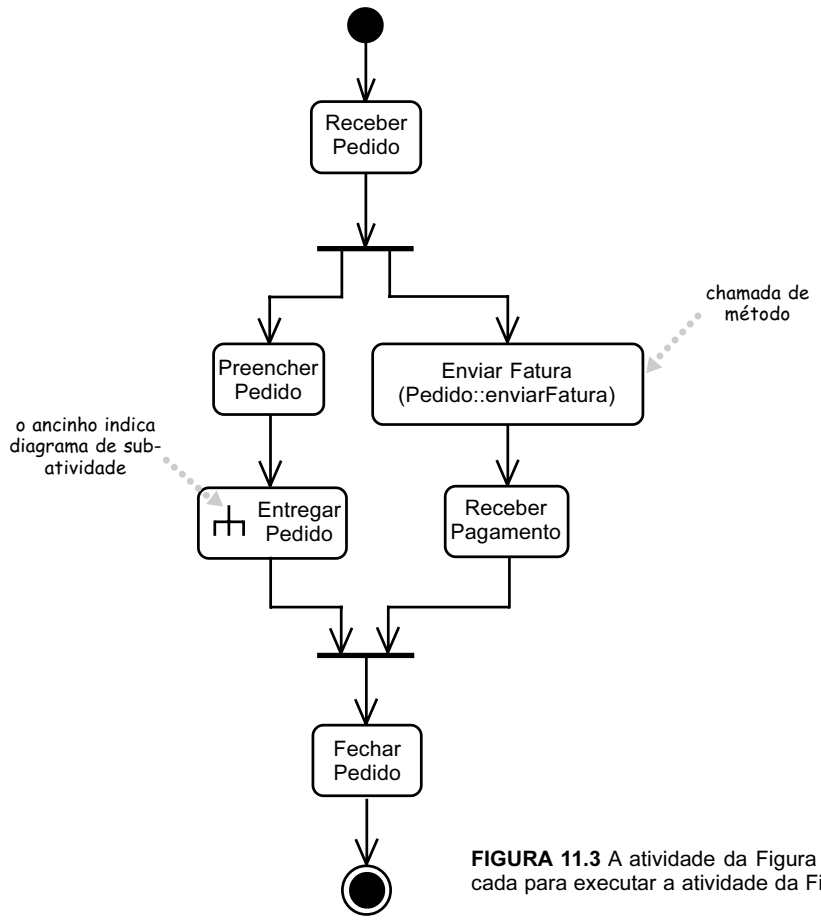


FIGURA 11.3 A atividade da Figura 11.1, modificada para executar a atividade da Figura 11.2.

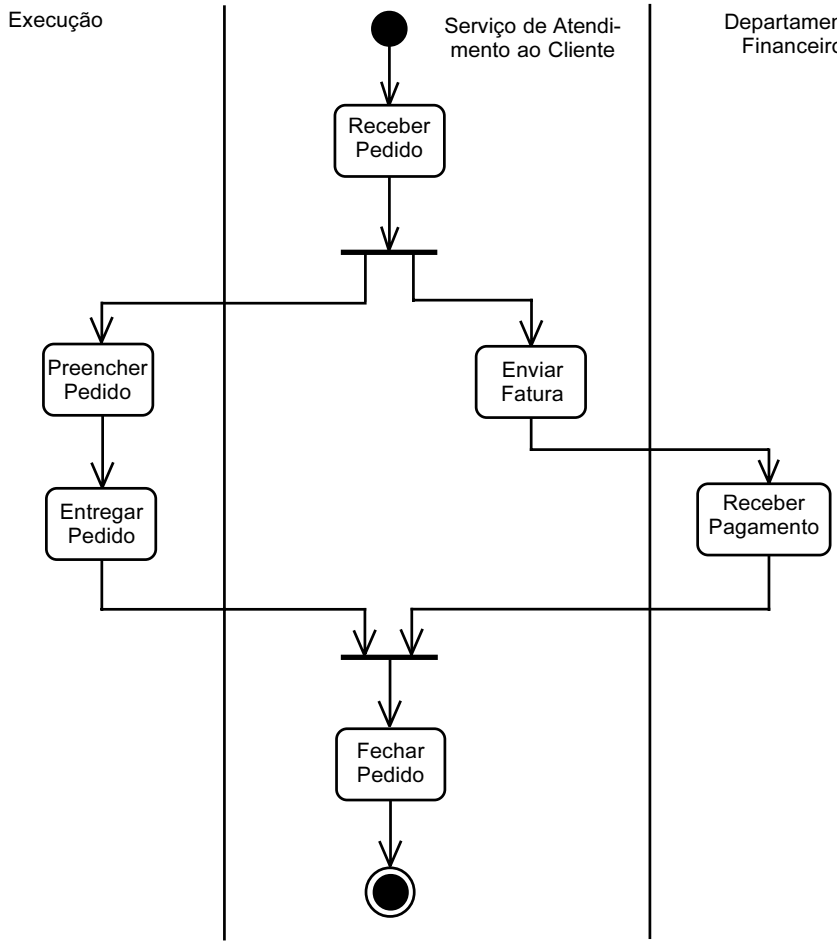


FIGURA 11.4 Partições em um diagrama de atividades.

SINAIS

No exemplo simples da Figura 11.1, os diagramas de atividades têm um ponto de partida claramente definido, que corresponde a uma chamada de um programa ou de uma rotina. As ações também podem responder a sinais.

Um **sinal de tempo** ocorre devido à passagem do tempo. Tais sinais poderiam indicar o final de um mês em um período financeiro ou cada microssegundo em um controlador de tempo real.

A Figura 11.5 mostra uma atividade que espera receber dois sinais. Um **sinal** indica que a atividade recebe um evento de um processo externo. Isso indica que a atividade capta esses sinais constantemente, e o diagrama define como a atividade reage.

No caso da Figura 11.5, duas horas antes de meu vôo sair, preciso começar a fazer minhas malas. Se eu for rápido ao fazê-las, ainda não posso sair enquanto o táxi não chegar. Se o táxi chegar antes de minhas malas estarem prontas, ele precisará esperar que eu termine, antes de sairmos.

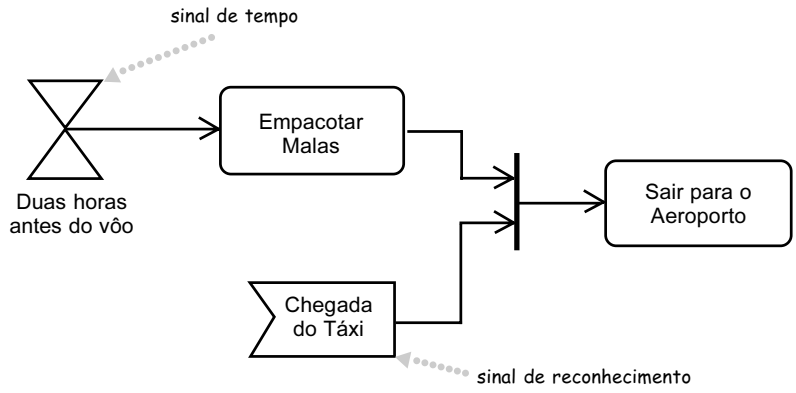


FIGURA 11.5 Sinais em um diagrama de atividades.

Assim como podemos aceitar sinais, também podemos enviá-los. Isso é útil quando precisamos enviar uma mensagem e depois esperar pela resposta, antes de podermos continuar. A Figura 11.6 mostra um bom exemplo disso, com um jargão comum da temporização. Note que os dois fluxos estão em uma disputa: o primeiro a chegar ao estado final vencerá e terminará o outro fluxo.

Embora os reconhecimentos normalmente sejam apenas esperar por um evento externo, também podemos mostrar um fluxo em sua direção. Isso indica que não começamos a captar até que o fluxo dispare o reconhecimento.

TOKENS

Se você for suficientemente audaz para se aventurar nas profundezas demoníacas da especificação da UML, verá que a seção sobre atividades fala muito sobre *tokens* e sua produção e consumo. O nó inicial cria um *token*, o qual é passado para a próxima ação, que é executada e depois passa o *token* para a seguinte. Em uma separação, um *token* entra e a separação produz um *token* em cada um de seus fluxos de saída. Inversamente,

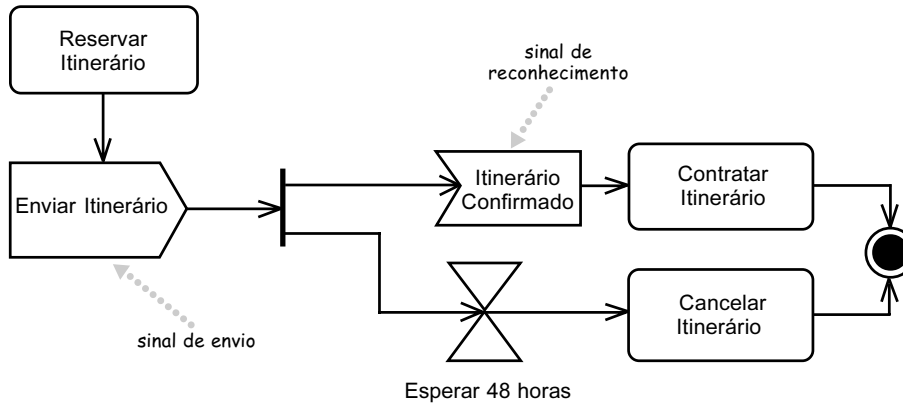


FIGURA 11.6 Enviando e recebendo sinais.

em uma junção, quando cada *token* de entrada chega, nada acontece até que todos os tokens apareçam na junção; então, um *token* é produzido no fluxo de saída.

Você pode visualizar os *tokens* com moedas ou contadores se movendo pelo diagrama. Quando você vê exemplos mais complicados de diagramas de atividades, os *tokens* freqüentemente tornam mais fácil visualizar as coisas.

FLUXOS E ARESTAS

A UML 2 utiliza os termos **fluxo** e **aresta** como sinônimos para descrever as conexões entre duas ações. O tipo mais simples de aresta é a seta simples entre duas ações. Se você quiser, pode dar um nome a uma aresta, mas na maioria das vezes uma seta simples bastará.

Se você estiver com dificuldade para traçar o caminho das linhas, pode usar conectores, que simplesmente evitam o desenho de uma linha pela distância inteira. Quando você usa conectores, deve fazê-lo em pares: um com o fluxo de entrada, outro com um fluxo de saída e ambos com o mesmo rótulo. Se possível, eu tento evitar o uso de conectores, pois eles confundem a visualização do fluxo de controle.

As arestas mais simples passam um *token* que não tem outro significado a não ser controlar o fluxo. Entretanto, você também pode passar objetos através das arestas; os objetos desempenham, então, o papel de *tokens*, assim como transportam dados. Se você estiver passando um objeto através de uma aresta, pode mostrar isso colocando uma caixa de classe na aresta ou usar pinos nas ações, embora os pinos impliquem mais algumas sutilezas, que descreverei em breve.

Todos os estilos mostrados na Figura 11.7 são equivalentes; você deve usar o que transmitir melhor o que está tentando comunicar. Na maioria das vezes, a seta simples é suficiente.

PINOS E TRANSFORMAÇÕES

As ações podem ter parâmetros, exatamente como acontece com os métodos. Você não precisa mostrar informações sobre parâmetros no diagrama de atividades, mas, se quiser,

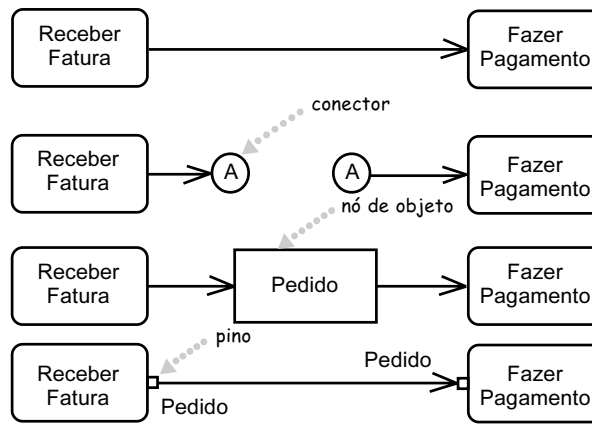


FIGURA 11.7 Quatro maneiras de mostrar uma aresta.

pode mostrá-los com **pinos**. Se você estiver decompondo uma ação, os pinos correspondem às caixas de parâmetro no diagrama decomposto.

Quando você está desenhando rigorosamente um diagrama de atividades, precisa certificar-se de que os parâmetros de saída de uma ação de saída correspondam aos parâmetros de entrada de outra. Se eles não corresponderem, você pode indicar uma **transformação** (Figura 11.8) para ir de um para outro. A transformação deve ser uma expressão isenta de efeitos colaterais: basicamente, uma consulta sobre o pino de saída que fornece um objeto do tipo correto para o pino de entrada.

Você não precisa mostrar pinos em um diagrama de atividades. Eles são mais indicados quando você quer ver os dados necessários e produzidos pelas várias ações. Na modelagem de processo de negócio, você pode usar pinos para mostrar os recursos produzidos e consumidos por ações.

Se você usa pinos, é seguro mostrar vários fluxos entrando na mesma ação. A notação de pino reforça a junção implícita, e a UML 1 não tinha pinos; portanto, não há confusão com as suposições anteriores.

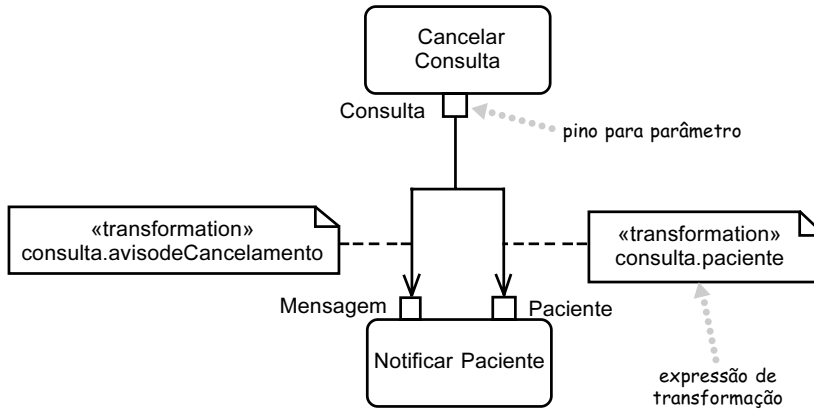


FIGURA 11.8 Transformação em um fluxo.

REGIÕES DE EXPANSÃO

Com os diagramas de atividades, você freqüentemente se depara com situações em que a saída de uma ação dispara múltiplas execuções de outra. Existem várias maneiras de mostrar isso, mas a melhor é usar uma região de expansão. Uma **região de expansão** marca uma área do diagrama de atividades onde as ações ocorrem uma vez para cada item de uma coleção.

Na Figura 11.9, a ação Escolher Tópicos gera uma lista de tópicos como saída. Cada elemento dessa lista se torna, então, um *token* de entrada para a ação Escrever Artigo. Analogamente, cada ação Examinar Artigo gera um único artigo, que é adicionado à lista de saída da região de expansão. Quando todos os *tokens* da região de expansão acabam na coleção de saída, a região gera um único *token* para a lista, que é passado para Publicar Boletim.

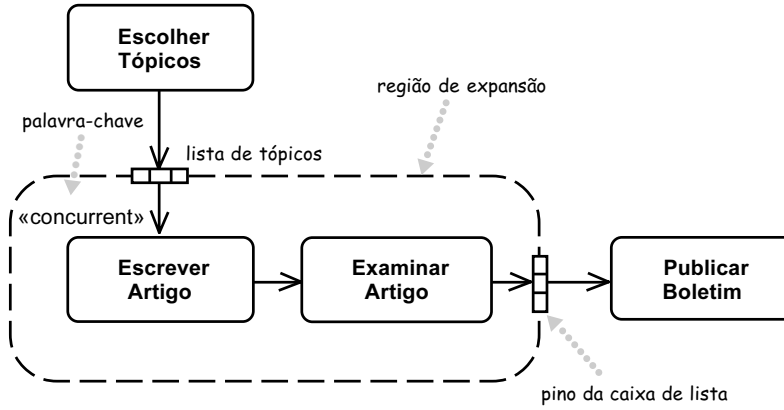


FIGURA 11.9 Região de expansão.

Neste caso, você tem o mesmo número de itens na coleção de saída e na coleção de entrada. Entretanto, você poderia ter menos, caso em que a região de expansão atuaria como um filtro.

Na Figura 11.9, todos os artigos são escritos e examinados em paralelo, o que é marcado pela palavra-chave «`concurrent`». Você também pode ter uma região de expansão iterativa. As regiões iterativas devem processar completamente cada elemento de entrada, um por vez.

Se você tem apenas uma ação que precisa de chamada múltipla, use a abreviação da Figura 11.10. A abreviação assume expansão concorrente, pois essa é a mais comum. Essa notação corresponde ao conceito de concorrência dinâmica da UML 1.

FIGURA 11.10 Abreviação para uma única ação em uma região de expansão.

FINAL DE FLUXO

Quando você tem múltiplos *tokens*, como em uma região de expansão, freqüentemente você tem fluxos que param mesmo quando a atividade como um todo não termina. Um **final de fluxo** indica o término de um fluxo em particular, sem terminar a atividade inteira.

A Figura 11.11 mostra isso por meio de uma modificação do exemplo da Figura 11.9, para permitir que artigos sejam rejeitados. Se um artigo é rejeitado, o *token* é destruído pelo final de fluxo. Ao contrário de um final de atividade, o restante da atividade pode continuar. Essa estratégia permite que as regiões de expansão atuem como filtros, por meio dos quais a coleção de saída fica menor do que a coleção de entrada.

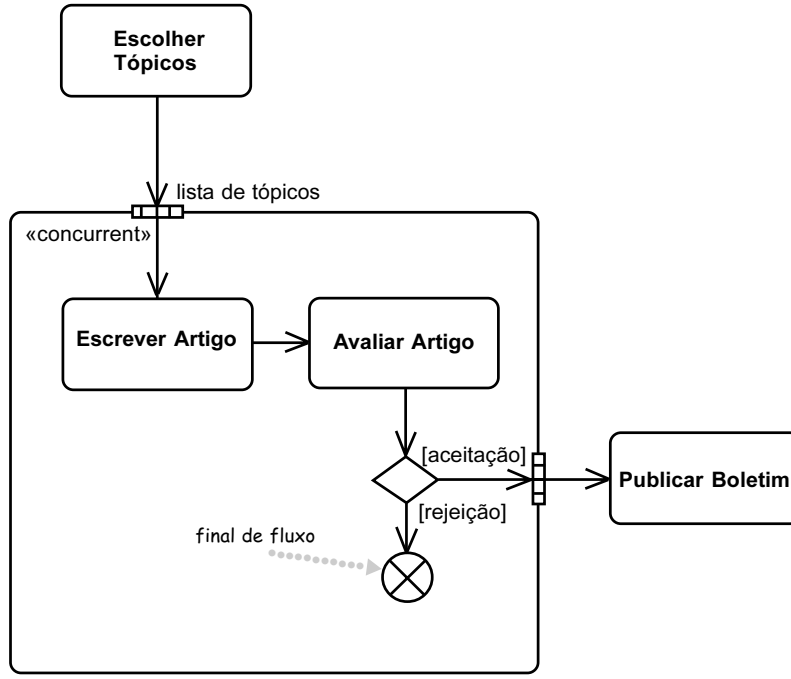


FIGURA 11.11 Finais de fluxo em uma atividade.

ESPECIFICAÇÕES DE JUNÇÃO

Por padrão, uma junção permite que a execução passe por seu fluxo de saída quando todos os seus fluxos de entrada tiverem chegado nela. (Ou, em uma linguagem mais formal, ela emite um *token* em seu fluxo de saída, quando um *token* tiver chegado em cada fluxo de entrada.) Em alguns casos, particularmente quando você tem um fluxo com múltiplos *tokens*, é interessante ter uma regra mais complicada.

Uma **especificação de junção** é uma expressão booleana ligada a uma junção. Sempre que um *token* chega à junção, a especificação da junção é avaliada e, se for verdadeira, um *token* de saída é emitido. Assim, na Figura 11.12, quando seleciono uma bebida ou insiro uma moeda, a máquina avalia a especificação de junção. A máquina satisfaz meu desejo somente se eu tiver colocado dinheiro suficiente. Se, como neste caso, você quiser

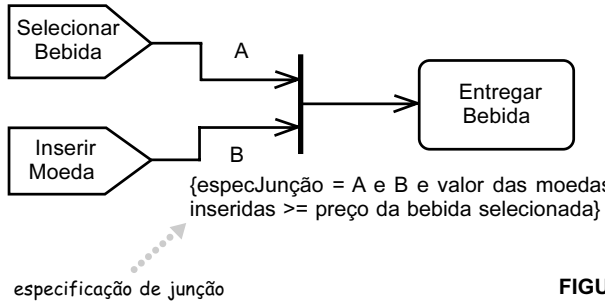


FIGURA 11.12 Especificação de junção.

indicar que recebeu um *token* em cada fluxo de entrada, rotule os fluxos e os inclua na especificação de junção.

E HÁ MAIS

Devo salientar que este capítulo apenas toca a superfície dos diagramas de atividades. Assim como para grande parte da UML, você poderia escrever um livro inteiro apenas sobre essa técnica. Na verdade, acho que os diagramas de atividades se constituiriam em um assunto muito conveniente para um livro que entrasse realmente a fundo na notação e como usá-la.

A questão fundamental é se seu uso seria difundido. Os diagramas de atividades não são a técnica da UML mais amplamente utilizada no momento, e seus progenitores de modelagem de fluxo também não eram muito usados. As técnicas de diagramação ainda não se tornaram muito populares para descrever comportamento dessa maneira. Por outro lado, existem sinais em várias comunidades de uma demanda reprimida que uma técnica padrão ajudaria a satisfazer.

QUANDO UTILIZAR DIAGRAMAS DE ATIVIDADES

A maior qualidade dos diagramas de atividades está no fato de que eles suportam e estimulam o comportamento paralelo. Isso os torna uma excelente ferramenta para modelagem de fluxos de trabalho e de processos. Na verdade, grande parte do avanço da UML 2 é devido às pessoas envolvidas com fluxo de trabalho.

Você também pode usar um diagrama de atividades como um fluxograma compatível com a UML. Embora isso permita fazer fluxogramas nos moldes da UML, não se trata de algo muito interessante. Em princípio, você pode usufruir das vantagens das separações e junções para descrever algoritmos paralelos para programas concorrentes. Embora eu não participe de grupos que trabalhem com programação concorrente, não tenho visto muitas evidências de pessoas os utilizando nisso. Acho que o motivo é que grande parte da complexidade da programação concorrente é evitar a disputa de dados, e os diagramas de atividades não ajudam muito nesse sentido.

A principal vantagem de fazer isso pode ser para pessoas que usam a UML como linguagem de programação. Nesse caso, os diagramas de atividades representam uma técnica importante para representar lógica comportamental.

Freqüentemente, tenho visto diagramas de atividades utilizados para descrever um caso de uso. O perigo dessa estratégia é que, muitas vezes, os especialistas no domínio não os acompanham facilmente. Se assim for, é melhor usar a forma textual normal.

ONDE ENCONTRAR MAIS INFORMAÇÕES

Embora os diagramas de atividades sempre tenham sido bastante complicados e estejam ainda mais na UML 2, nunca houve um bom livro que os descrevesse com profundidade. Espero que essa lacuna seja preenchida algum dia.

Várias técnicas orientadas a fluxos têm estilo semelhante aos diagramas de atividades. Uma das melhores – mas dificilmente conhecida – são as Redes Petri, para as quais o endereço **http://www.daimi.au.dk/PetriNets/** é uma boa página na Web.

Capítulo 12

Diagramas de Comunicação

Os **diagramas de comunicação**, um tipo de diagrama de interação, enfatiza os vínculos de dados entre os vários participantes na interação. Em vez de desenhar cada participante como uma linha de vida e mostrar a seqüência de mensagens por meio da direção vertical, como fazem os diagramas de seqüência, o diagrama de comunicação permite livre posicionamento dos participantes, permite desenhar vínculos para mostrar como eles se conectam e usa numeração para mostrar a seqüência de mensagens.

Na UML 1.x, esses diagramas eram chamados de **diagramas de colaboração**. Esse nome pegou bem e suspeito que as pessoas demorarão algum tempo para se acostumar com o novo nome. (Eles são diferentes das Colaborações [página 143]; daí a mudança de nome.)

A Figura 12.1 mostra um diagrama de comunicação para a mesma interação de controle centralizado da Figura 4.2. Com um diagrama de comunicação, podemos mostrar como os participantes estão vinculados.

Assim como mostrar vínculos que são instâncias de associações, podemos mostrar também vínculos transitórios, que surgem somente no contexto da interação. Neste caso, o vínculo «`local`» de Pedido para Produto é uma variável local; outros vínculos transitórios são «`parameter`» e «`local`». Essas palavras-chave eram usadas na UML 1, mas estão ausentes na UML 2. Como elas são úteis, espero que permaneçam em uso convencional.

O estilo de numeração da Figura 12.1 é simples e comumente usado, mas na verdade não é válido na UML. Para estar de acordo com a UML, você precisa usar um esquema de numeração decimal aninhada, como na Figura 12.2.

O motivo para os números decimais aninhados é resolver a ambiguidade com as autochamadas. Na Figura 4.2, você pode ver claramente que `obterInformaçãodeDesconto` é chamado dentro do método `calcularDesconto`. Com a numeração simples da Figura 12.1, entretanto, você não pode identificar se `obterInformaçãodeDesconto` é chamado dentro de `calcularDesconto` ou dentro do método global `calcularPreço`. O esquema de numeração aninhada resolve esse problema.

Apesar de sua ilegalidade, muitas pessoas preferem um esquema de numeração simples. Os números aninhados podem ficar muito confusos, particularmente quando as chamadas ficam aninhadas demais, levando a números de seqüência como 1.1.1.2.1.1. Nesses casos, a cura da ambiguidade pode ser pior do que a doença.

Assim como números, você também pode ver letras nas mensagens; essas letras indicam diferentes linhas de execução de controle. Assim, as mensagens A5 e B2 estariam em diferentes linhas de execução; as mensagens 1a1 e 1b1 seriam diferentes linhas de execução aninhadas dentro da mensagem 1. Você também vê letras de linha

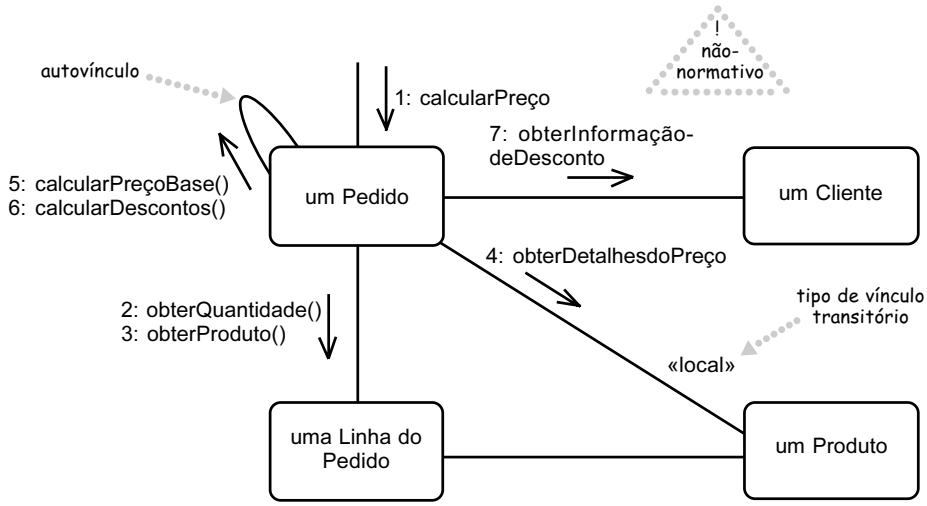


FIGURA 12.1 Diagrama de comunicação para controle centralizado.

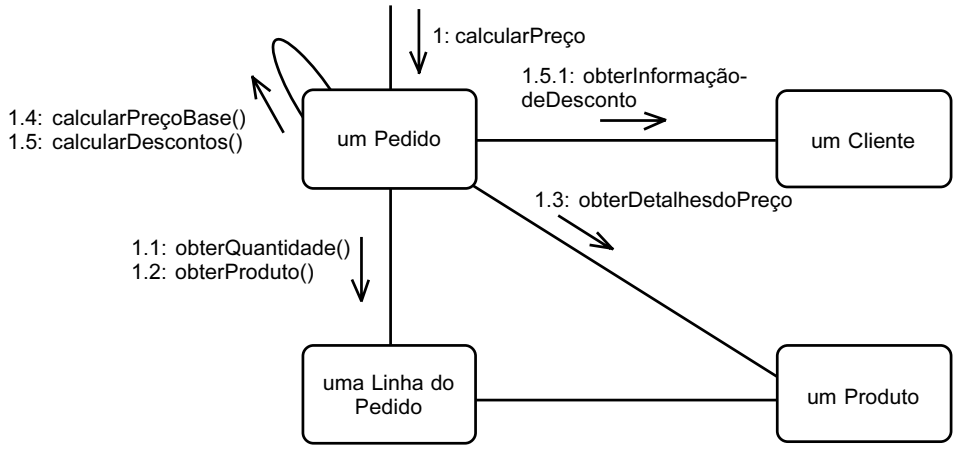


FIGURA 12.2 Diagrama de comunicação com numeração decimal aninhada.

de execução em diagramas de seqüência, embora isso não transmita visualmente a concorrência.

Os diagramas de comunicação não têm qualquer notação precisa para lógica de controle. Eles permitem que você use marcadores de iteração e sentinelas (página 72), mas não permitem especificar totalmente a lógica de controle. Não há nenhuma notação especial para criar ou destruir objetos, mas as palavras-chave «`create`» e «`delete`» são convenções comuns.

QUANDO USAR DIAGRAMAS DE COMUNICAÇÃO

A principal questão com os diagramas de comunicação é quando usá-los em lugar dos diagramas de seqüência, mais comuns. Grande parte da decisão é questão de preferência

pessoal: algumas pessoas gostam mais de um do que de outro. Freqüentemente, isso direciona a escolha mais do que tudo. Em geral, a maioria das pessoas parece preferir os diagramas de seqüência e, pelo menos uma vez, estou com a maioria.

Uma abordagem mais racional diz que os diagramas de seqüência são melhores quando você quer salientar a seqüência de chamadas e que os diagramas de comunicação são melhores quando quer salientar os vínculos. Muitas pessoas acham que os diagramas de comunicação são mais fáceis de alterar em um quadro branco, de modo que eles são uma boa estratégia para explorar alternativas, embora nesses casos, eu freqüentemente prefira os cartões CRC.

Capítulo 13

Estruturas Compostas

Um dos recursos novos mais significativos da UML 2 é a capacidade de decompor hierarquicamente uma classe em uma estrutura interna. Isso permite que você pegue um objeto complexo e divida-o em partes.

A Figura 13.1 mostra uma classe Visualizador de TV com suas interfaces fornecidas e exigidas (página 69). Mostrei isso de duas maneiras: usando a notação de bola-e-soquete e listando-as internamente.

A Figura 13.2 mostra como essa classe é decomposta internamente em duas partes e quais partes suportam e exigem as diferentes interfaces. Cada parte tem um nome da forma `nome: classe`, com os dois elementos individualmente opcionais. As partes não são especificações de instância, de modo que aparecem em negrito e não sublinhadas.

Você pode mostrar quantas instâncias de uma parte estão presentes. A Figura 13.2 informa que cada Visualizador de TV contém uma parte geradora e uma parte de controles.

Para mostrar uma parte implementando uma interface, você desenha um conector de delegação a partir dessa interface. Similarmente, para mostrar que a parte necessita de uma interface, você mostra um corretor de delegação para a interface. Você também pode mostrar conectores entre as partes com uma linha simples, conforme fiz aqui, ou com notação de bola-e-soquete (página 82).

Você pode adicionar portas (Figura 13.3) à estrutura externa. As portas permitem que você agrupe as interfaces exigidas e fornecidas em interações lógicas que um componente possui com o mundo externo.

QUANDO USAR ESTRUTURAS COMPOSTAS

As estruturas compostas são novas na UML 2, embora alguns métodos mais antigos tivessem algumas idéias semelhantes. Uma boa maneira de pensar a respeito da diferença entre pacotes e estruturas compostas é que os pacotes são um agrupamento em tempo de compilação, enquanto que as estruturas compostas mostram agrupamentos em tempo de execução. Desse modo, elas servem naturalmente para mostrar componentes e como eles são divididos em partes; assim, grande parte dessa notação é usada em diagramas de componentes.

Como as estruturas compostas são novas na UML, é muito cedo para dizer o quanto elas se mostrarão eficazes na prática; muitos membros do comitê da UML acham que esses diagramas se tornarão uma adição muito valiosa.

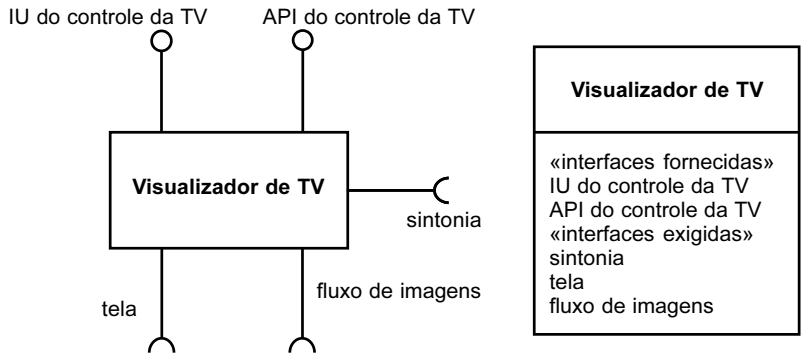


FIGURA 13.1 Duas maneiras de mostrar um visualizador de TV e suas interfaces.

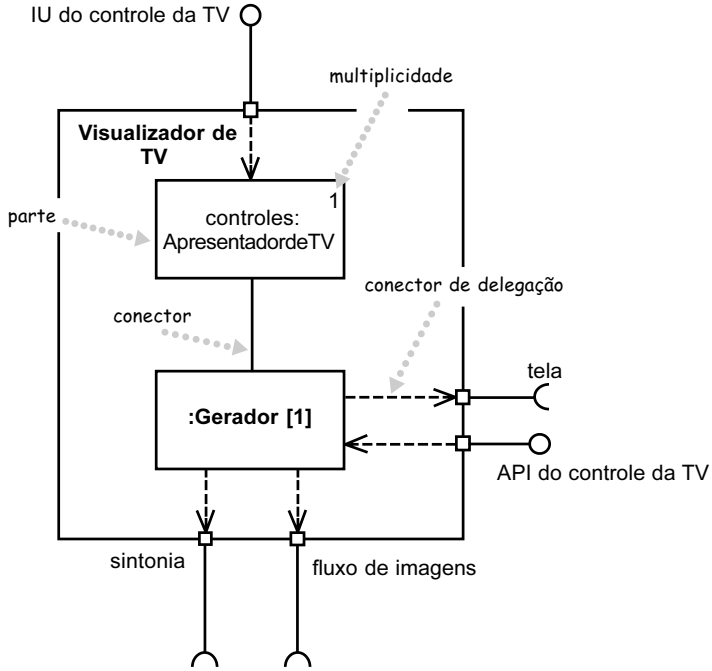


FIGURA 13.2 Visão interna de um componente (exemplo sugerido por Jim Rumbaugh).

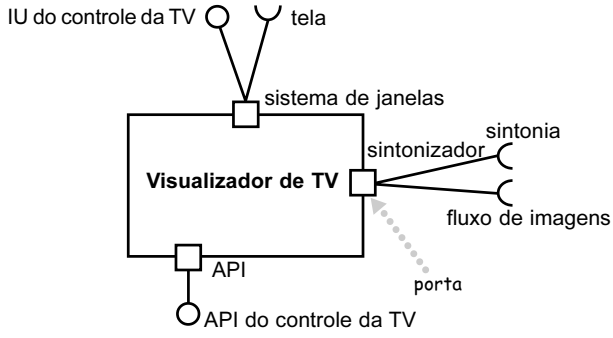


FIGURA 13.3 Um componente com várias portas.

Capítulo 14

Diagramas de Componentes

Uma polêmica que sempre vai longe na comunidade orientada a objetos é a diferença existente entre um componente e uma classe regular qualquer. Esse não é um debate que quero estabelecer aqui, mas posso mostrar a notação que a UML utiliza para distinguí-los.

A UML 1 tinha um símbolo característico para um componente (Figura 14.1). A UML 2 retirou esse ícone, mas permite que você anote uma caixa de classe com um ícone de aparência semelhante. Como alternativa, você pode usar a palavra-chave «`component`».

Além do ícone, os componentes não introduzem nenhuma notação que ainda não tenhamos visto. Os componentes são conectados por meio de interfaces implementadas e exigidas, freqüentemente usando a notação de bola-e-soquete (página 71), exatamente como para diagramas de classes. Você também pode decompor os componentes usando diagramas de estrutura composta.

A Figura 14.2 mostra um exemplo de diagrama de componentes. Nesse exemplo, uma caixa registradora pode se conectar com um componente servidor de vendas, usando uma interface de mensagens de vendas. Como a rede não é confiável, um componente fila de mensagens é introduzido para que a caixa possa se comunicar com o servidor, quando a rede estiver ativa, e se comunicar com uma fila, quando a rede estiver desativada; a fila se comunicará então com o servidor, quando a rede se tornar disponível. Como resultado, a fila de mensagens fornece a interface de mensagens de vendas para se comunicar com a caixa e exige essa interface para se comunicar com o servidor. O servidor é dividido em dois componentes principais. O processador de transações implementa a interface de mensagens de vendas e o *driver* de contabilidade se comunica com o sistema de contabilidade.

Conforme eu já disse, a questão do que é um componente é um assunto de discussões intermináveis. Uma das declarações mais úteis que encontrei é a seguinte:

> *Componentes não são uma tecnologia. O pessoal técnico parece achar isso difícil de entender. Os componentes dizem como os clientes querem se relacionar com o* software. *Eles querem comprar seu* software *uma peça de cada vez e atualizá-lo, exatamente como fazem com seus equipamentos estereofônicos. Eles querem que as novas peças funcionem transparentemente com as peças mais antigas e querem atualizar em seu próprio ritmo, e não de acordo com a programação do fabricante. Eles querem misturar e combinar peças de vários fabricantes. Essa é uma exigência bastante razoável. É apenas difícil de satisfazer.*
>
> Ralph Johnson, **http://www.c2.com/cgi/wiki?DoComponentsExist**

O ponto importante é que os componentes representam peças que podem ser adquiridas e atualizadas independentemente. Como resultado, dividir um sistema em componentes é tanto uma decisão de *marketing* quanto uma decisão técnica, para a qual [Hohmann] é um guia excelente. Também é um lembrete para tomar cuidado com componentes demasiadamente refinados, pois tendo-se componentes demais torna-se difícil gerenciar, especialmente quando o controle de versão mostra sua cabeça horrenda; por isso, o termo "inferno da DLL".

Nas versões anteriores da UML, os componentes eram usados para representar estruturas físicas, como as DLLs. Isso não é mais verdade; para essa tarefa, agora você usa artefatos (página 97).

QUANDO USAR DIAGRAMAS DE COMPONENTES

Use diagramas de componentes quando você estiver dividindo seu sistema em componentes e quiser mostrar seus relacionamentos por intermédio de interfaces ou a decomposição de componentes em uma estrutura de nível mais baixo.

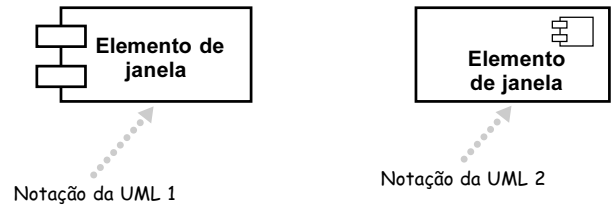


FIGURA 14.1 Notação para componentes.

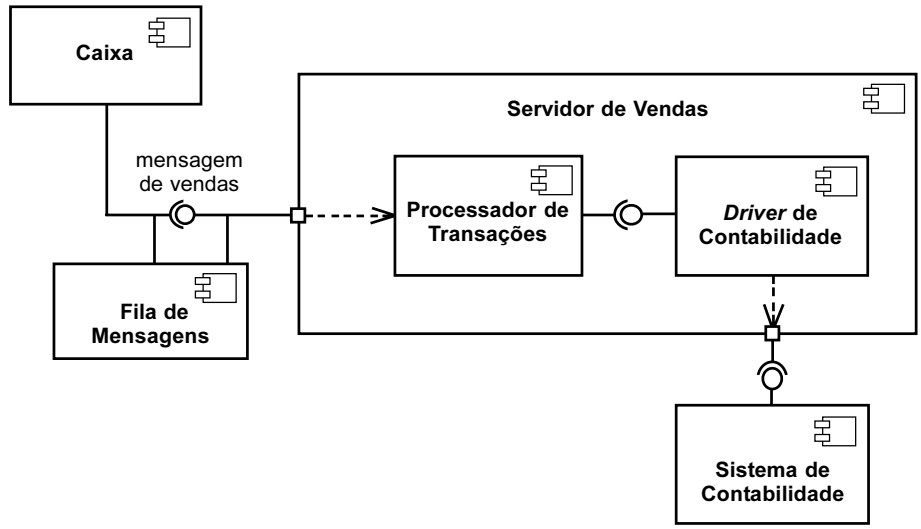


FIGURA 14.2 Um exemplo de diagrama de componentes.

Capítulo 15

Colaborações

Ao contrário dos outros capítulos deste livro, este não corresponde a um diagrama oficial da UML 2. O padrão discute colaborações como parte de estruturas compostas, mas o diagrama é bastante diferente e foi usado na UML 1 sem qualquer vínculo com as estruturas compostas. Assim, achei melhor discutir as colaborações em seu próprio capítulo.

Vamos considerar a noção de leilão. Em qualquer leilão, devemos ter um vendedor, alguns compradores, lotes de mercadorias e algumas ofertas para a venda. Podemos descrever esses elementos em termos de um diagrama de classes (Figura 15.1) e, talvez, alguns diagramas de interação (Figura 15.2).

A Figura 15.1 não é um diagrama de classes normal. Para começar, o diagrama está circundado pela elipse tracejada, que representa a colaboração leilão. Segundo, as assim chamadas classes na colaboração não são classes propriamente ditas, mas **papéis** que serão desempenhados quando a colaboração for aplicada – daí o fato de seus nomes não iniciarem por letras maiúsculas. Não é incomum ver interfaces reais ou classes que correspondam aos papéis da colaboração, mas você não precisa tê-las.

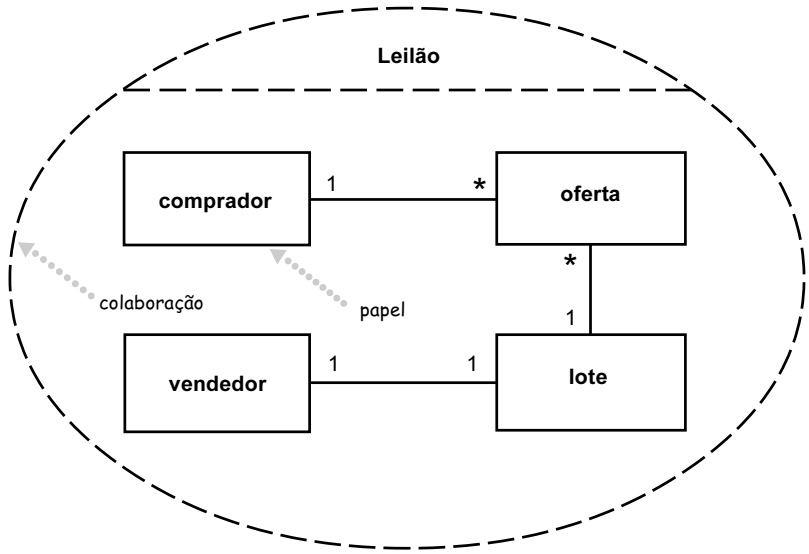


FIGURA 15.1 Uma colaboração com seu diagrama de classes de papéis.

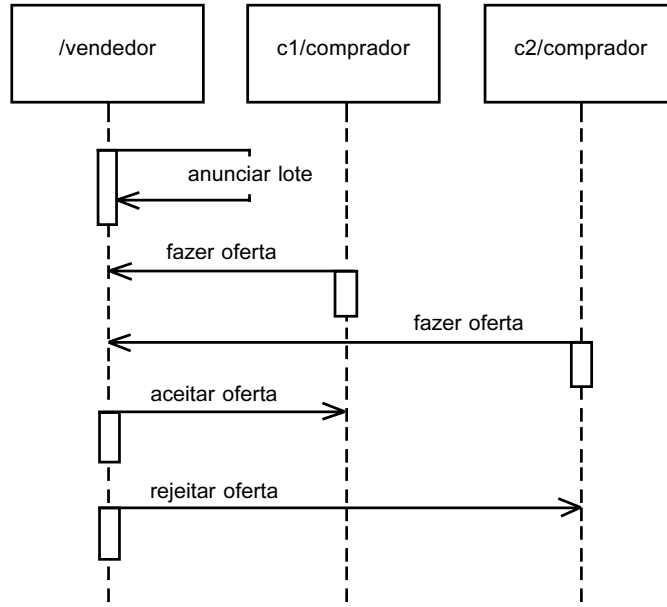


FIGURA 15.2 Um diagrama de seqüência para a colaboração leilão.

No diagrama de interação, os participantes são rotulados de modo ligeiramente diferente do caso normal. Em uma colaboração, o esquema de atribuição de nomes é `nome-do-participante / nome-do-papel: nome-da-classe`. Como sempre, todos esses elementos são opcionais.

Quando você usa uma colaboração, pode mostrar isso colocando uma ocorrência de colaboração em um diagrama de classes, como na Figura 15.3, um diagrama de algu-

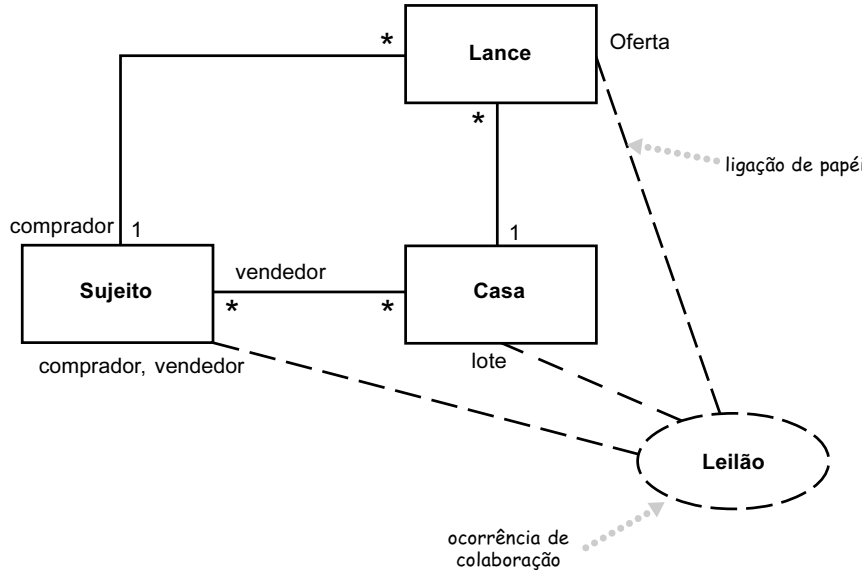


FIGURA 15.3 Uma ocorrência de colaboração.

mas das classes da aplicação. Os vínculos da colaboração para essas classes indicam como as classes desempenham os vários papéis definidos na colaboração.

A UML sugere que você pode usar a notação de ocorrência de colaboração para mostrar o uso de padrões, mas dificilmente os autores de padrões têm feito isso. Erich Gamma desenvolveu uma excelente notação alternativa (Figura 15.4). Os elementos do diagrama são rotulados com o nome do padrão ou com uma combinação de `padrão:papel`.

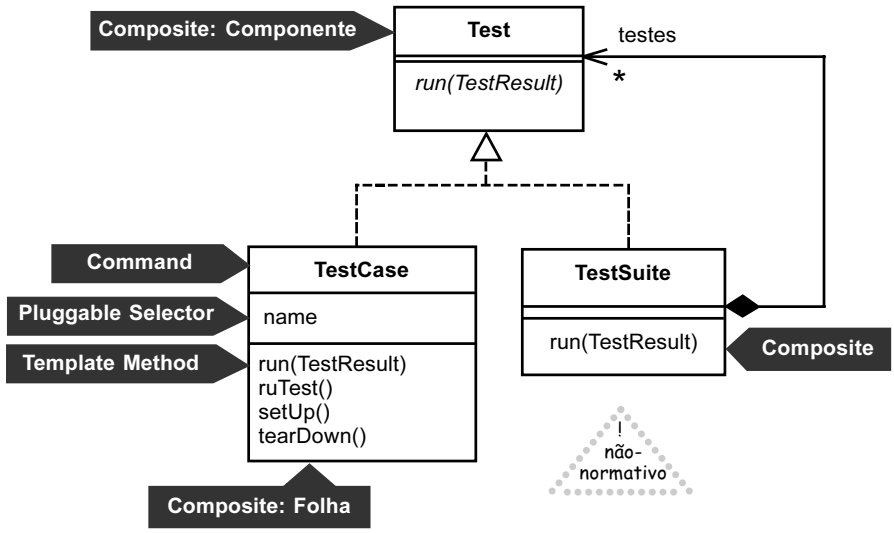


FIGURA 15.4 Uma maneira não-padronizada de mostrar o uso de padrões em JUnit (junit.org).

QUANDO USAR COLABORAÇÕES

As colaborações já existem desde a UML 1, mas admito que dificilmente as tenho utilizado, mesmo quando escrevo sobre padrões. As colaborações fornecem uma maneira de agrupar trechos de comportamento de interação, quando os papéis são desempenhados por classes diferentes. Na prática, entretanto, não considero que eles sejam um tipo de diagrama atraente.

Capítulo 16

Diagramas de Visão Geral da Interação

Os diagramas de visão geral da interação são uma mistura de diagramas de atividades e diagramas de seqüência. Você pode considerar os diagramas de visão geral da interação como diagramas de atividades nos quais as atividades são substituídas por pequenos diagramas de seqüência ou como um diagrama de seqüência fragmentado, com a notação de diagrama de atividades usada para mostrar o fluxo de controle. De qualquer modo, eles fazem uma mistura bastante estranha.

A Figura 16.1 mostra um exemplo de diagrama simples; a notação é conhecida, a partir do que você já viu nos capítulos sobre diagramas de atividades e sobre diagramas de seqüência. Nesse diagrama, queremos produzir e formatar um relatório de resumo de pedidos. Se o cliente é externo, obtemos as informações em XML; se é interno, as obtemos de um banco de dados. Pequenos diagramas de seqüência mostram as duas alternativas. Quando obtemos os dados, formatamos o relatório; neste caso, não mostramos o diagrama de seqüência, mas simplesmente fazemos referência a ele com um quadro de interação de referência.

QUANDO USAR DIAGRAMAS DE VISÃO GERAL DA INTERAÇÃO

Esses diagramas são novos na UML 2 e é cedo demais para se ter uma idéia de como eles funcionarão na prática. Não gosto muito deles, pois acho que eles misturam dois estilos que não se encaixam muito bem. Desenhe um diagrama de atividades ou use um diagrama de seqüência, dependendo do que melhor atender seu propósito.

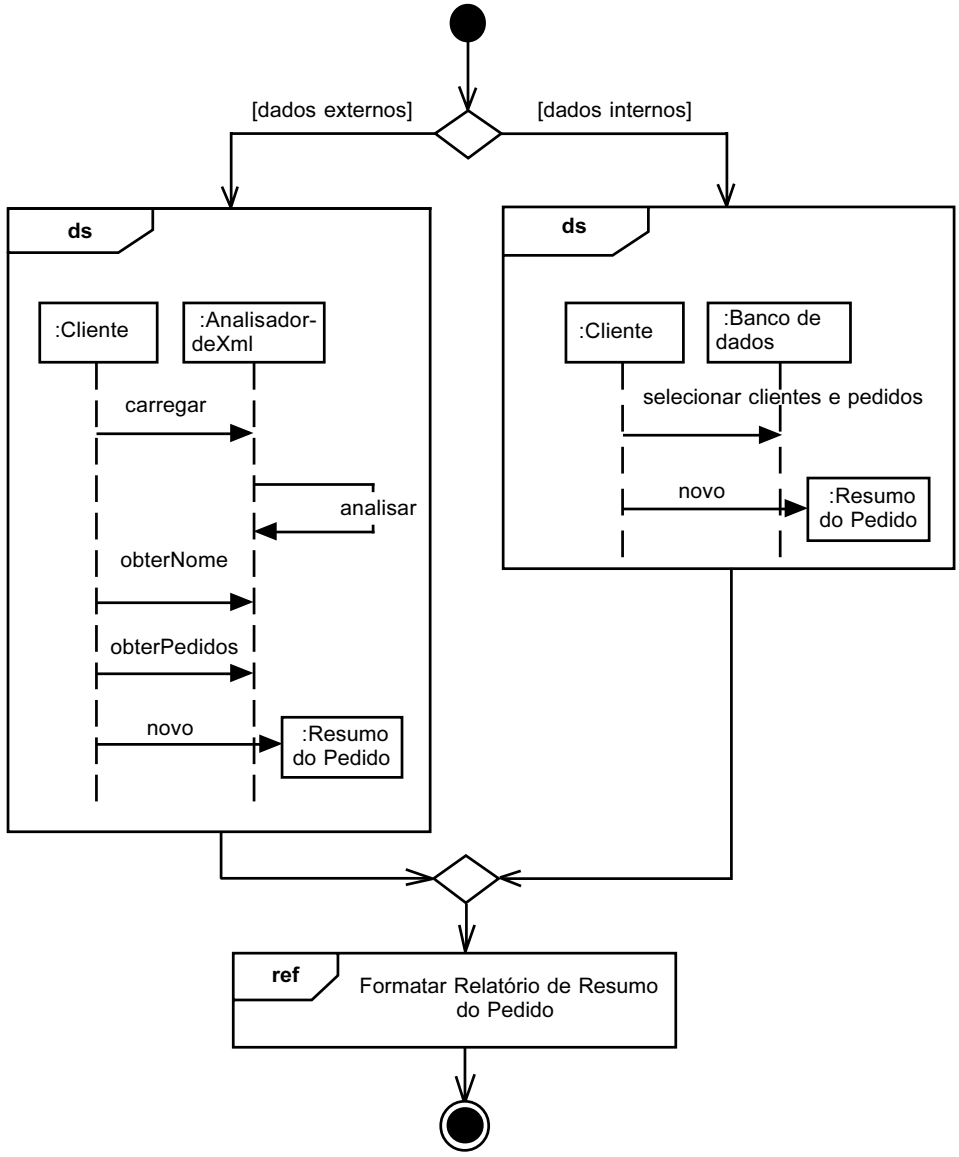


FIGURA 16.1 Diagrama de resumo de interação.

Capítulo 17

Diagramas de Temporização

Após sair da escola secundária, comecei a fazer engenharia eletrônica, antes de mudar para a computação. Assim, sinto certa familiaridade angustiosa quando vejo a UML definir diagramas de temporização como um de seus diagramas padrão. Os diagramas de temporização existem na engenharia eletrônica há muito tempo e parecem nunca ter precisado da ajuda da UML para definir seu significado. Mas como eles estão na UML, merecem uma breve menção.

Os diagramas de temporização são outra forma de diagrama de interação, nos quais o foco está nas restrições de temporização: ou para um único objeto ou, de forma mais útil, para vários. Vamos considerar um cenário simples, baseado na bomba e na chapa elétrica de uma cafeteira. Vamos imaginar uma regra que diga que pelo menos 10 segundos devem passar entre o acionamento da bomba e o aquecimento da chapa. Quando o reservatório de água se esvazia, a bomba desliga, e a chapa não pode permanecer ligada por mais de 15 minutos depois disso.

As figuras 17.1 e 17.2 são maneiras alternativas de mostrar essas restrições de temporização. Os dois diagramas mostram a mesma informação básica. A principal diferença é que a Figura 17.1 mostra as mudanças de estado, movendo-se de uma linha horizontal para outra, enquanto a Figura 17.2 mantém a mesma posição horizontal, mas mostra as mudanças de estado com uma cruz. O estilo da Figura 17.1 funciona melhor quando existem apenas alguns estados, como neste caso, e a Figura 17.2 é melhor quando existem muitos estados para se lidar.

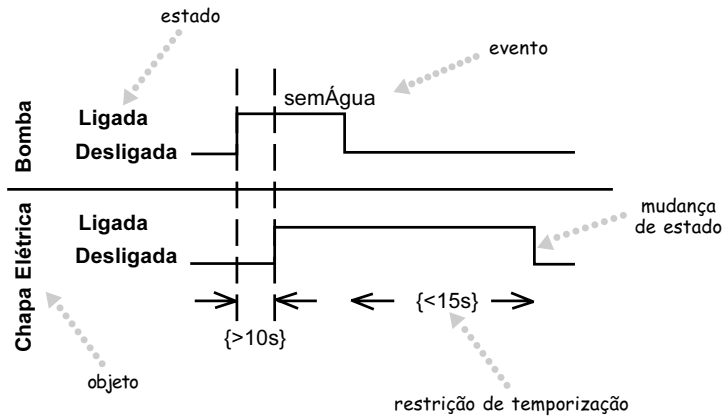


FIGURA 17.1 Diagrama de temporização mostrando os estados como linhas.

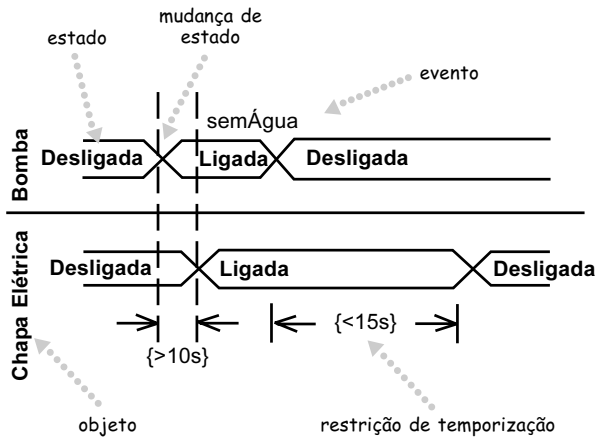


FIGURA 17.2 Diagrama de temporização mostrando os estados como áreas.

As linhas tracejadas que usei nas restrições {>10s} são opcionais. Utilize-as se você achar que elas ajudam a esclarecer exatamente quais eventos a temporização restringe.

QUANDO USAR DIAGRAMAS DE TEMPORIZAÇÃO

Os diagramas de temporização são úteis para mostrar restrições de temporização entre mudanças de estado em diferentes objetos. Os diagramas são particularmente conhecidos dos engenheiros de *hardware*.

Apêndice

Modificações nas Versões da UML

Quando a primeira edição deste livro foi para as livrarias, a UML estava na versão 1.0. Grande parte dela parecia ter se estabilizado e estava no processo de reconhecimento pelo OMG. Desde então, várias revisões têm sido feitas. Neste apêndice, descrevo as modificações significativas que ocorreram desde a versão 1.0 e como essas mudanças afetam o material deste livro.

Este apêndice resume as modificações para que você possa se manter atualizado, caso tenha uma edição mais antiga. Fiz alterações no livro para acompanhar a UML; portanto, se você tem uma edição mais antiga, ela descreve a situação na data da publicação.

REVISÕES NA UML

O lançamento público mais antigo do que veio a ser a UML foi a versão 0.8 do Método Unificado lançado na OOPSLA, em outubro de 1995. Esse foi um trabalho de Booch e Rumbaugh, pois Jacobson ainda não era membro da Rational naquela época. Em 1996, a Rational lançou as versões 0.9 e 0.91, que incluíam o trabalho de Jacobson. Depois dessa última versão, eles mudaram o nome para UML.

A Rational e um grupo de parceiros submeteram a versão 1.0 da UML para a Analysis and Design Task Force do OMG, em janeiro de 1997. Subseqüentemente, a parceria da Rational e os outros remetentes combinaram seu trabalho e submeteram uma proposta única para o padrão OMG, em setembro de 1997, da versão 1.1 da UML. Esta foi adotada pelo OMG no final de 1997. Entretanto, devido a uma confusão, o OMG chamou esse padrão de versão 1.0. Então, a UML era tanto a versão 1.0 do OMG como a versão 1.1 da Rational, para não ser confudida com a versão 1.0 da Rational. Na prática, todos chamam esse padrão de versão 1.1.

Daí em diante, houve vários outros desenvolvimentos na UML. A UML 1.2 apareceu em 1998, a 1.3, em 1999, a 1.4, em 2001 e a 1.5, em 2002. A maior parte das alterações entre as versões 1.x foi muito profunda na UML, exceto quanto à UML 1.3, o que causou algumas mudanças bastante visíveis, especialmente nos casos de uso e nos diagramas de atividades.

À medida que a série UML 1 continuou, os desenvolvedores da UML se concentraram em uma revisão importante na UML, com a versão 2. Os primeiros RFPs (Request for Proposals) foram publicados em 2000, mas a UML 2 não começou a se estabilizar propriamente até 2003.

Desenvolvimentos futuros na UML certamente ocorrerão. Normalmente, o UML Forum (**http://uml-forum.com**) é um bom lugar para procurar mais informações. Eu

também mantenho algumas informações sobre a UML em minha página (**http://martinfowler.com**).

MUDANÇAS NO *UML ESSENCIAL*

À medida que essas revisões acontecem, venho tentando acompanhá-las, revisando o livro *UML Essencial* com impressões subseqüentes*. Também tive a oportunidade de corrigir erros e fazer esclarecimentos.

O período mais dinâmico de acompanhamento foi durante a primeira edição do *UML Essencial*, quando tínhamos que fazer atualizações freqüentes entre as impressões para acompanhar o padrão UML emergente. UML 1.0 foi usada como base da primeira à quinta impressão. As mudanças na UML entre essas impressões foram mínimas. A sexta impressão levou em consideração a UML 1.1.

Da sétima à décima impressão, a UML 1.2 foi usada como base; a décima-primeira impressão foi a primeira a usar a UML 1.3. As edições baseadas em versões da UML posteriores a 1.0 têm o número de versão da UML na capa.

A primeira até a sexta impressão da segunda edição foram baseadas na versão 1.3. A sétima impressão foi a primeira a levar em conta as pequenas alterações da versão 1.4.

A terceira edição foi lançada para atualizar o livro com a UML 2 (veja a Tabela A.1). No restante deste apêndice, faço um resumo das principais mudanças na UML, das versões 1.0 para 1.1, da 1.2 para 1.3 e da 1.x para a 2.0. Não discuto todas as alterações que ocorreram, mas somente aquelas que mudam algo que eu disse no *UML Essencial* ou que representam recursos importantes que eu teria discutido no livro.

Continuo a seguir o espírito do *UML Essencial*: discutindo os elementos principais da UML, conforme eles afetam a aplicação de UML em projetos do mundo real. Como sempre, as escolhas e os conselhos são meus. Se houver algum conflito entre o que eu digo e os documentos oficiais da UML, os documentos é que devem ser seguidos. (Mas me informem, para que eu possa fazer correções).

TABELA A.1 *UML Essencial* e versões correspondentes da UML

UML Essencial	Versões da UML
1ª edição	UML 1.0-1.3
2ª edição	UML 1.3-1.4
3ª edição	UML 2.0 em diante

Também aproveitei a oportunidade para indicar erros e omissões importantes das edições anteriores. Agradeço aos leitores que me mostraram esses erros.

MUDANÇAS DA UML 1.0 PARA 1.1

Tipo e Classe de Implementação

Na primeira edição do *UML Essencial*, abordei perspectivas e como elas alteravam a maneira como as pessoas desenham e interpretam modelos – particularmente diagramas

* N. de R.: As reimpressões a que se refere o autor dizem respeito à obra original, publicada nos EUA.

de classes. Agora, a UML leva isso em consideração, dizendo que todas as classes em um diagrama de classes podem ser especializadas tanto como tipos quanto como classes de implementação.

Uma **classe de implementação** corresponde a uma classe no ambiente de *software* no qual você está programando. Um **tipo** é ainda mais nebuloso; ele representa uma abstração não tão amarrada à implementação. Esse poderia ser um tipo CORBA, uma perspectiva de especificação de uma classe ou uma perspectiva conceitual. Se necessário, você pode acrescentar estereótipos para diferenciar ainda mais.

Você pode determinar que, para um diagrama em particular, todas as classes sigam um estereótipo específico. Isso é o que você faria ao desenhar um diagrama a partir de uma perspectiva determinada. A perspectiva de implementação usaria classes de implementação, enquanto as perspectivas conceituais e de especificação usariam tipos.

Você usa o relacionamento de realização para indicar que uma classe de implementação implementa um ou mais tipos.

Existe uma distinção entre tipo e interface. Uma interface tem como objetivo corresponder diretamente a uma interface de estilo COM ou Java. Portanto, as interfaces só têm operações e não atributos.

Você pode usar somente classificação estática e simples com classes de implementação, mas pode usar classificação dinâmica e múltipla com tipos. (Acredito que isso ocorre porque as linguagens orientadas a objetos mais importantes seguem a classificação estática e simples. Se um dia você usar uma linguagem que suporte classificação dinâmica ou múltipla, essa restrição não deverá ser aplicada.)

Restrições Discriminadoras Completas e Incompletas

Nas edições anteriores do *UML Essencial*, afirmei que uma restrição {`complete`} em uma generalização indicava que todas as instâncias do supertipo também devem ser uma instância de um subtipo dentro dessa partição. Em vez disso, a UML 1.1 define que {`complete`} indica que todos os subtipos dentro de uma partição foram especificados, o que não é exatamente a mesma coisa. Encontrei alguma inconsistência na interpretação dessa restrição; portanto, você deve ser cauteloso quanto a isso. Se você quiser indicar que todas as instâncias de um supertipo devem ser uma instância de um dos subtipos, sugiro o uso de uma outra restrição para evitar confusão. Atualmente, estou usando {`mandatory`}.

Composição

Na UML 1.0, o uso da composição implicava que o vínculo fosse imutável (ou *frozen*), pelo menos para componentes de valor único. Essa restrição não faz mais parte da definição.

Imutabilidade e Congelamento

A UML estabelece a restrição {`frozen`} para definir a imutabilidade em papéis de associação. Conforme ela está atualmente definida, isso não parece se aplicar a atributos ou a classes. No meu trabalho, uso agora o termo ***frozen***, em vez de imutabilidade, e aplico a restrição a papéis de associações, classes e atributos.

Retornos em Diagramas de Seqüência

Na UML 1.0, um retorno em diagramas de seqüência era diferenciado pelo uso de uma ponta de seta, em vez de uma seta cheia (veja as edições anteriores). Isso era um pouco complicado, pois a distinção era muito sutil e fácil de passar despercebida. A UML 1.1 usa uma seta tracejada para um retorno, o que me agrada, pois isso torna os retornos muito mais óbvios. (Quando usei retornos tracejados no livro *Analysis Patterns* [Fowler, AP], isso também me faz parecer influente). Você pode dar nome ao que é retornado para uso posterior, usando a forma `estoqueSuficiente:= verificar()`.

Uso do Termo "Papel"

Na UML 1.0, o termo **papel** indicava, principalmente, uma direção em uma associação (veja as edições anteriores). A UML 1.1 se refere a esse uso como **papel de associação**. Existe também um **papel de colaboração**, que é o papel que uma instância de uma classe desempenha em uma colaboração. A UML 1.1 dá muito mais ênfase às colaborações e parece que esse uso de "papel" pode se tornar o principal.

MUDANÇAS DA UML 1.2 (E 1.1) PARA 1.3 (E 1.5)

Casos de Uso

As mudanças nos casos de uso envolvem novos relacionamentos entre eles. A UML 1.1 tem dois relacionamentos de caso de uso: «`uses`» e «`extends`», sendo os dois estereótipos de generalização. A UML 1.3 oferece três relacionamentos.

- A construção «`include`» é um estereótipo de dependência. Isso indica que o caminho de um caso de uso está incluído em um outro. Normalmente, isso ocorre quando poucos casos de uso compartilham passos em comum. O caso de uso incluído pode fatorar o comportamento comum. Um exemplo de um Caixa Automático poderia ser que tanto Retirar Dinheiro quanto Fazer Transferência usem Validar Cliente. Isso substitui o uso comum de «`uses`».
- A **generalização** de casos de uso indica que um caso de uso é uma variação de um outro. Portanto, poderíamos ter um caso para Retirar Dinheiro (o caso de uso de base) e um caso de uso separado para lidar com o caso de quando a retirada é recusada devido à falta de fundos. A recusa poderia ser tratada como um caso de uso que especializa o caso de uso de retirada. (Você também poderia tratar disso simplesmente como um outro cenário dentro do caso de uso Retirar Dinheiro.) Uma especialização de caso de uso como essa pode mudar qualquer aspecto do caso de uso de base.
- A construção «`extend`» é um estereótipo de dependência. Isso fornece uma forma mais controlada de extensão do que o relacionamento de generalização. Aqui, o caso de uso de base declara vários pontos de extensão. O caso de uso estendido pode alterar o comportamento somente nesses pontos de extensão. Então, se você está comprando um produto *on-line*, pode ter um caso de uso para comprar um produto com pontos de extensão, para capturar a informação

de expedição e a informação de pagamento. Esse caso de uso poderia, então, ser estendido para um cliente regular, para o qual essas informações seriam obtidas de forma diferente.

Existe certa confusão sobre a relação entre os relacionamentos antigos e os novos. A maioria das pessoas usava «uses» da maneira como «includes» é usado na versão 1.3; portanto, para a maioria das pessoas, podemos dizer que «includes» substitui «uses». E a maioria das pessoas usava «extends» da versão 1.1 tanto na maneira controlada da palavra-chave «extends» da versão 1.3 quanto como uma sobreposição geral no estilo da generalização da versão 1.3. Assim, você pode pensar que «extends» da versão 1.1 foi dividida em «extend» e na generalização da versão 1.3.

Agora, embora essa explicação cubra a maioria dos usos da UML que já vi, não é a maneira estritamente correta de usar esses antigos relacionamentos. Entretanto, a maioria das pessoas não seguiu o uso estrito e não quero entrar nesse assunto aqui.

Diagramas de Atividades

Quando a UML chegou à versão 1.2, havia algumas questões em aberto sobre a semântica dos diagramas de atividades. Assim, o trabalho da 1.3 envolveu muito ajuste nessa semântica.

Para comportamento condicional, agora você pode usar a atividade de decisão, em forma de losango, para uma intercalação de comportamentos, bem como para uma ramificação. Embora nem intercalações nem ramificações sejam necessárias para descrever comportamento condicional, mostrá-las é um estilo cada vez mais comum, para que você possa colocar o comportamento condicional entre colchetes.

A barra de sincronização é agora referida como **separação** (ao dividir o controle) ou como **junção** (ao sincronizar o controle). Entretanto, você não pode mais acrescentar condições arbitrárias nas junções. Além disso, você deve seguir regras de correspondência para garantir que separações e junções combinem. Basicamente, isso significa que cada separação deve ter uma junção correspondente que una as linhas de execução iniciadas por uma separação. Você pode aninhar separação e junções, e pode eliminá-las do diagrama, quando as linhas de execução forem diretamente de uma separação para outra (ou de uma junção para outra).

As junções são acionadas somente quando todas as linhas de execução de entrada se completam. No entanto, você pode ter, em uma linha de execução, uma condição proveniente de uma separação. Se essa condição é falsa, essa linha de execução é considerada concluída para propósito de junção.

O recurso de disparo múltiplo não está mais presente. No seu lugar, você pode ter concorrência dinâmica em uma atividade, mostrada com um asterisco (*) dentro de uma caixa de atividade. Tal atividade pode ser chamada várias vezes em paralelo; todas as suas chamadas devem ser completadas, antes que qualquer transição de saída possa ser tomada. Isso é vagamente equivalente, embora menos flexível, a um disparo múltiplo e à condição de sincronização correspondente.

Essas regras reduzem um pouco a flexibilidade dos diagramas de atividade, mas garantem que eles sejam realmente casos especiais de máquinas de estados. O relacionamento entre diagramas de atividades e máquinas de estados foi assunto de debate no RTF. Revisões futuras da UML (posteriores à versão 1.4) podem muito bem tornar os diagramas de atividades uma forma de diagrama completamente diferente.

MUDANÇAS DA UML 1.3 PARA 1.4

A mudança mais visível na UML 1.4 é a adição de **perfis**, que permitem que um grupo de extensões sejam coletadas em um conjunto coerente. A documentação da UML inclui dois exemplos de perfis. Junto com isso, há um maior formalismo envolvido na definição de estereótipos, e elementos de modelo agora podem ter múltiplos estereótipos; na UML 1.3, eles eram limitados a um só.

Artefatos foram acrescentados na UML. Um artefato é uma manisfestação física de um componente; assim, por exemplo, Xerces é um componente e todas aquelas cópias de arquivos jar Xerces em meu disco rígido são artefatos que implementam o componente Xerces.

Antes da versão 1.3, não havia nada no metamodelo da UML para tratar a **visibilidade de pacote** da linguagem Java. Agora há, e o símbolo é "~".

A UML 1.4 também transformou a ponta de seta vazada nos diagramas de interação em um sinal assíncrono, uma incômoda mudança, incompatível com as versões anteriores. Isso surpreendeu algumas pessoas, incluindo a mim.

MUDANÇAS DA UML 1.4 PARA 1.5

A principal mudança aqui foi a adição da semântica de ação na UML, um passo necessário para tornar a UML uma linguagem de programação. Isso foi feito para permitir que as pessoas trabalhassem nisso sem esperar pela UML 2.0 completa.

MUDANÇAS DA UML 1.X PARA 2.0

A UML 2 representa a maior mudança que aconteceu na UML. Todos os tipos de coisas mudaram com essa revisão e muitas alterações afetaram o livro *UML Essencial*.

Dentro da UML, houve mudanças profundas no metamodelo. Embora essas mudanças não afetem as discussões deste livro, elas são muito importantes para alguns grupos.

Uma das alterações mais evidentes é a introdução de novos tipos de diagrama. Os diagramas de objetos e os diagramas de pacotes já eram amplamente desenhados, mas não eram tipos de diagrama oficiais; agora, eles são. A UML 2 mudou o nome dos diagramas de colaboração para diagramas de comunicação. Ela também introduziu novos tipos de diagrama: diagramas de visão geral da interação, diagramas de temporização e diagramas de estrutura composta.

Muitas alterações não foram abordadas no *UML Essencial*. Omiti a discussão sobre construções como extensões de máquina de estados, portas em diagramas de interação e tipos de poder em diagramas de classes.

Assim, nesta seção, estou discutindo apenas as alterações que entraram no *UML Essencial*. Trata-se de alterações em coisas que discutimos nas edições anteriores ou novidades que comecei a discutir nesta edição. Como as mudanças são bastante espalhadas, as organizei de acordo com os capítulos deste livro.

Diagramas de Classes: os Elementos Básicos (Capítulo 3)

Os atributos e associações unidirecionais são agora, principalmente, apenas notações diferentes para o mesmo conceito subjacente de propriedade. As multiplicidades descontínuas, como [2, 4], foram eliminadas. A propriedade de congelamento (*frozen*) foi eliminada. Adicionei uma lista de palavras-chave de dependência comuns, diversas das quais são novas na UML 2. As palavras-chave «`parameter`» e «`local`» foram retiradas.

Diagramas de Seqüência (Capítulo 4)

A grande alteração aqui é a notação de quadro de interação para diagramas de seqüência, para lidar com controles de comportamento iterativos, condicionais e vários outros. Agora, isso permite que você expresse algoritmos de forma bastante completa em diagramas de seqüência, embora eu não esteja convencido de que eles sejam mais claros do que o código. Os antigos marcadores de iteração e sentinelas de mensagens foram eliminados dos diagramas de seqüência. Os cabeçalhos das linhas de vida não são mais instâncias; eu utilizo o termo **participante** para me referir a eles. Os diagramas de colaboração da UML 1 foram renomeados para diagramas de comunicação, na UML 2.

Diagramas de Classes: Conceitos (Capítulo 5)

Os estereótipos agora são mais rigidamente definidos. Como resultado, agora me refiro às palavras entre os sinais de menor e maior como palavras-chave, apenas algumas das quais são estereótipos. Nos diagramas de objetos, as instâncias são especificações de instância. As classes podem exigir interfaces, assim como fornecê-las. A classificação múltipla utiliza conjuntos de generalização para agrupar as generalizações. Os componentes não são mais desenhados com seu símbolo especial. Os objetos ativos têm linhas verticais duplas, em vez de linhas grossas.

Diagramas de Máquina de Estados (Capítulo 10)

A UML 1 separava as ações de vida curta das atividades de vida longa. A UML 2 chama as duas de atividades e usa o termo realizar-atividade para as atividades de vida longa.

Diagramas de Atividades (Capítulo 11)

A UML 1 tratava os diagramas de atividades como um caso especial do diagrama de estados. A UML 2 quebrou esse vínculo e, como resultado, removeu as regras de correspondência de separações e junções que os diagramas de atividades da UML 2 tinham que manter. Como resultado, eles são melhor entendidos pelo fluxo de *tokens* do que pela transição de estado. Assim, apareceu muita notação nova, incluindo sinais de tempo e de aceitação, parâmetros, especificações de junção, pinos, transformações de fluxo, ancinhos de subdiagrama, regiões de expansão e finais de fluxo.

Uma alteração simples, porém incômoda, é que a UML 1 tratava fluxos múltiplos de entrada para uma atividade como uma intercalação implícita, enquanto a UML 2 os trata como uma junção implícita. Por isso, aconselho usar uma intercalação ou junção explícita, quando fizer diagramas de atividades.

As raias de piscina agora podem ser multidimensionais e, de modo geral, são chamadas de partições.

Bibliografia

[**Ambler**]
Scott Ambler, *Agile Modeling*, Wiley, 2002

[**Beck**]
Kent Beck, *Extreme Programming Explained: Embrace Change.* AddisonWesley, 2000.

[**Beck**]
Kent Beck e Martin Flowler, *Planning Extreme Programming*, AddisonWesley, 2000.

[**Beck e Cunningham**]
Kent Beck e Ward Cunningham, "A Laboratory for Teaching ObjectOriented Thinking", *Proceedings of OOPSLA 89*, 24, (10): 1-6. http://c2.com/doc/oopsla89/paper.html

[**Booch, OOAD**]
Grady Booch, *Object-Oriented Analysis and Design with Applications, 2ª ed.* Addison-Wesley, 1994.

[**Booch, UML user**]
Grady Booch, James Rumbaugh e Ivar Jacobson, *UML User Guide.* Addison-Wesley, 1999.

[**Coad, OOA**]
Peter Coad e Edward Yourdon, *Object-Oriented Analysis.* Yourdon, 1991.

[**Coad, OOD**]
Peter Coad e Edward Yourdon, *Object-Oriented Design.* Yourdon, 1991.

[**Cockburn, agile**]
Alistair Cockburn, *Agile Software Development.* Addison-Wesley, 2001.

[**Cockburn, use cases**]
Alistair Cockburn: *Writing Effective Use Cases.* Addison-Wesley, 2001.

[**Constantine e Lockwood**]
Larry Constantine e Lucy Lockwood, *Software for Use*, Addison-Wesley, 2000.

[**Cook e Daniels**]
Steve Cook e John Daniels, *Designing Object Systems: Object-Oriented Modeling with Syntropy*, Prentice-Hall, 1994.

[**Core J2EE Patterns**]
Deepak Alur, John Crupi e Dan Malks, *Core J2EE Patterns*, Prentice-Hall, 2001.

[**Cunningham**]
Ward Cunningham, "EPISODES: A Pattern Language of Competitive Development". Em *Pattern Languages of Program Design 2*,Vlissides, Coplien e Kerth, Addison-Wesley, 1996, pgs. 371-388.

[**Douglass**]
Bruce Powel Douglass, *Real-Time UML.* Addison-Wesley, 1999.

[**Fowler, AP**]
Martin Fowler, *Analysis Patterns: Reusable Object Models.* Addison-Wesley, 1997.

[**Fowler, new methodology**]
Martin Fowler, "The New Methodology", **http://martinfowler.com/articles/newMethodology.html**

[**Fowler e Foemmel**]
Martin Fowler e Matthew Foemmel, "Continuous Integration", **http://martinfowler.com/articles/continuousIntegration.html**

[**Fowler, P of EAA**]
Martin Fowler, *Patterns of Enterprise Application Architecture*, Addison-Wesley, 2003

[**Fowler, refactoring**]
Martin Fowler, *Refactoring: Improving the Design of Existing Programs.* Addison-Wesley, 1999.

[**Gangue dos Quatro**]
Erich Gamma, Richard Helm, Ralph Johnson e John Viissides, *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[**Highsmith**]
Jim Highsmith, *Agile Software Development Ecosystems.* Addison-Wesley, 2002.

[**Hohmann**]
Luke Hohmann, *Beyond Software Architecture.* Addison-Wesley, 2003.

[**Jacobson, OOSE**]
Ivar Jacobson, Magnus Christerson, Patrik Jonsson e Gunnar Övergaard, *Object-Oriented Software Engineering: A Use Case Driven Approach.* Addison-Wesley, 1992.

[**Jacobson, UP**]
Ivar Jacobson, Maria Ericsson e Agneta Jacobson: *The Object Advantage: Business Process Reengineering with Object Technology.* Addison-Wesley, 1995.

[**Kerth**]
Norm Kerth, *Project Retrospectives.* Dorset House, 2001.

[**Kleppe et al.**]
Anneke Kleppe, Jos Warmer e Wim Bast, *MDA Explained*, Addison-Wesley, 2003.

[**Kruchten**]
Philippe Kruchten: *The Rational Unified Process: An Introduction.* Addison-Wesley, 1999.

[**Larman**]
Craig Larman, *Applying UML and Patterns.* 2ª ed., Prentice-Hall, 2001.

[**Martin**]
Robert Cecil Martin, *The Principles, Patterns, and Practices of Agile Software Development.* Prentice-Hall, 2003.

[**McConnell**]
Steve McConnell: *Rapid Development: Taming Wild Software Schedules.* Microsoft Press, 1996.

[**Mellor e Balcer**]
Steve Mellor e Marc Balcer, *Executable UML*, Addison-Wesley, 2002.

[**Meyer**]
Bertrand Meyer, *Object-Oriented Software Construction.* Prentice Hall, 2000.

[**Odell**]
James Martin e James J. Odell, *Object-Oriented Methods: A Foundation (UML Edition)*, Prentice Hall, 1998.

[**Pont**]
Michael Pont, *Patterns for Time-Triggered Embedded Systems*, Addison-Wesley, 2001.

[**POSA1**]
Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad e Michael Stal, *Pattern-Oriented Software Architecture: A System of Patterns*, Wiley, 1996.

[**POSA2**]
Douglas Schmit, Michael Stal, Hans Rohnert e Frank Buschmann, *Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects*, Wiley, 2000.

[**Rumbaugh, insights**]
James Rumbaugh, *OMT Insights.* SIGS Books, 1996.

[**Rumbaugh, OMT**]
James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy e William Lorenzen, *Object-Oriented Modeling and Design*, Prentice-Hall, 1991.

[**Rumbaugh, UML Rerefence**]
James Rumbaugh, Ivar Jacobson e Grady Booch, *The Unified Modeling Language Reference Manual.* Addison-Wesley, 1999.

[**Shlaer e Mellor, data**]
Sally Shlaer e Stephen J. Mellor, *Object-Oriented Systems Analysis: Modeling the World in Data.* Yourdon Press, 1989.

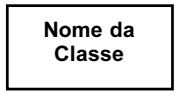
[**Shlaer e Mellor, states**]
Sally Shlaer e Stephen J. Mellor, *Object Lifecycles: Modeling the World in States.* Yourdon Press, 1991.

[**Warmer e Kleppe**]
Jos Warmer e Anneke Kleppe, *The Object Constraint Language: Precise Modeling with UML.* Addison-Wesley, 1998.

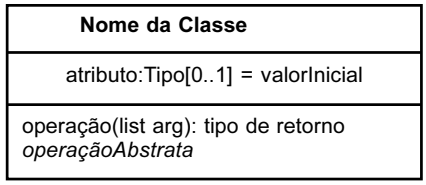
[**Wirfs-Brock**]
Rebecca Wirfs-Brock e Alan McKean, *Object Design: Roles Responsabilities and Collaborations.* Prentice Hall, 2003.

Índice

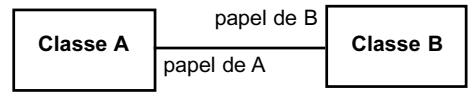
Classe



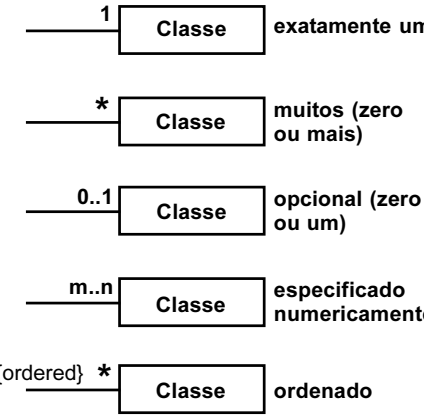
p.52



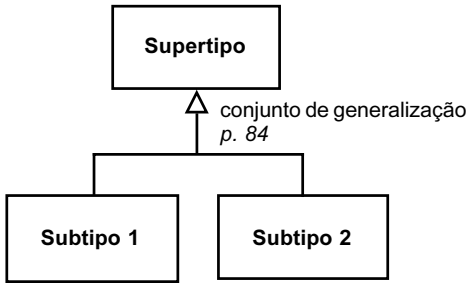
Associação *p. 54*



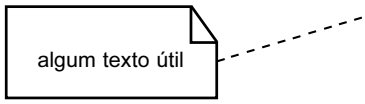
Multiplicidades *p.54*



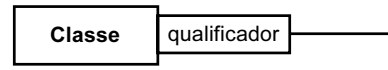
Generalização *p. 60*



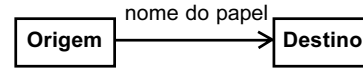
Nota *p. 61*



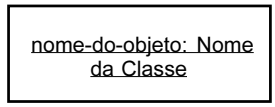
Associação Qualificada *p. 85*



Navegabilidade *p. 58*



Especificação de Instância *p. 94*



Dependência *p. 62*

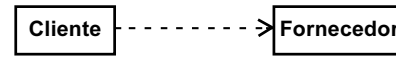
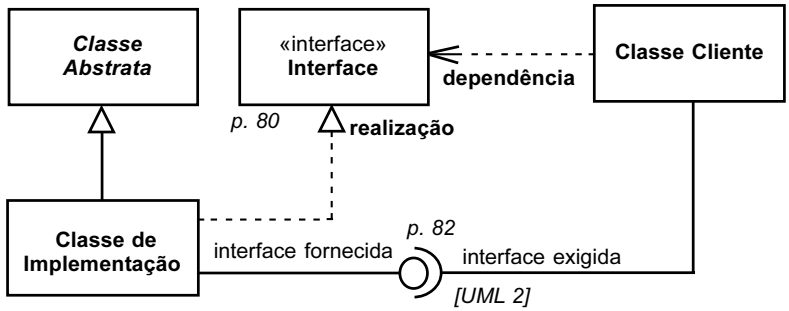
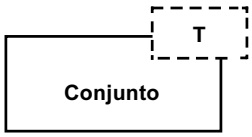
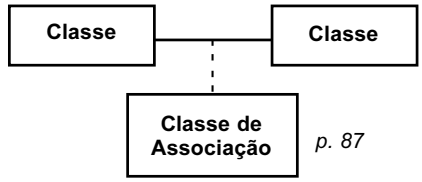


Diagrama de Classes

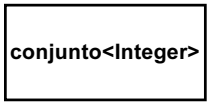


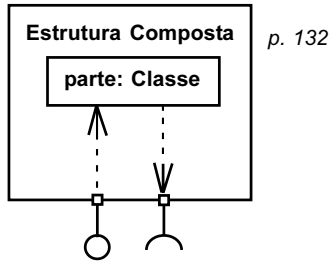
classe *template* *p. 90*





elemento de ligação





p. 134

Diagrama de Comunicação *p. 129*

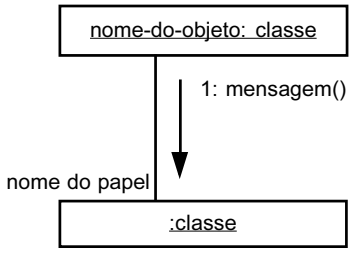


Diagrama de Casos de Uso *p. 104*

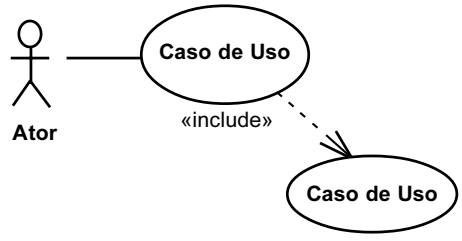


Diagrama de Pacotes *p. 96*

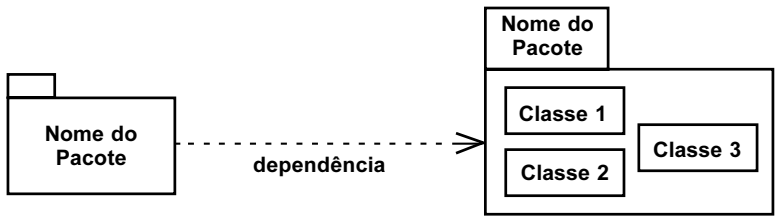
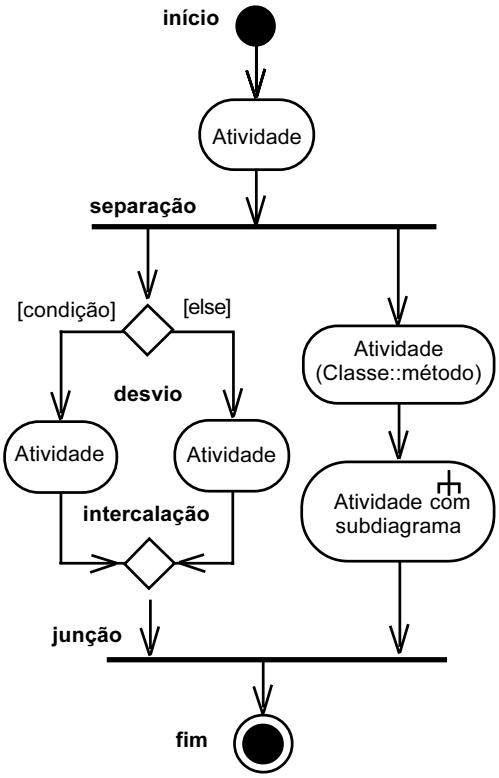
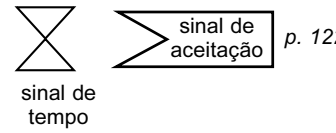


Diagrama de Atividades *p. 118*



final de fluxo *p. 126*



p. 122

Região de Expansão *p. 126*

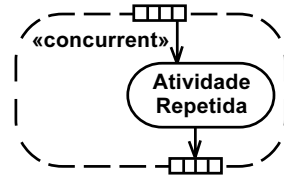


Diagrama de Instalação *p. 102*

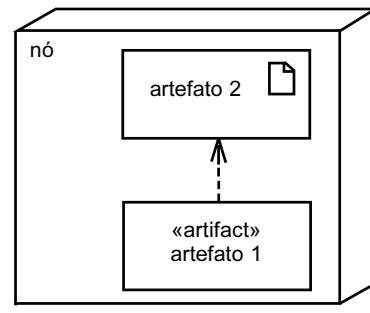
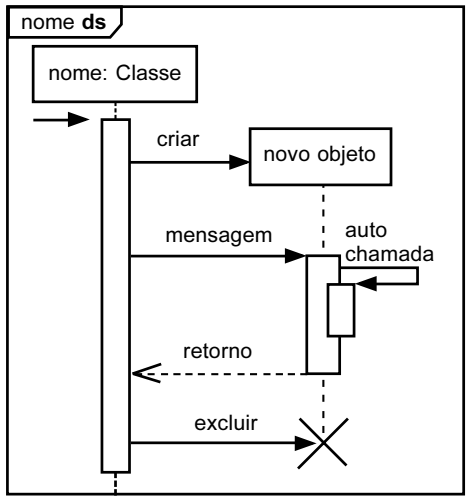
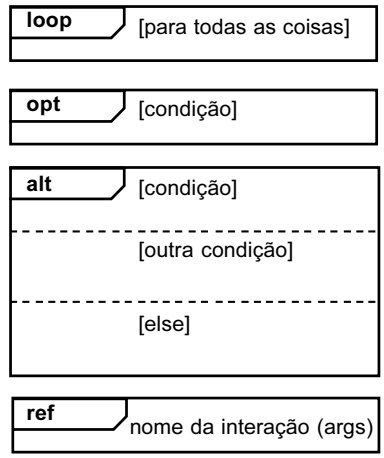


Diagrama de Seqüência *p. 67*





p. 70

síncrono *p. 74*

assíncrono *[UML >= 1.4]*

assíncrono *[UML <= 1.3]*

*:mensagem interativa ()
[condição] mensagem ()

[UML 1] p. 72

Diagrama de Estados *p. 110*