Padrões de Arquitetura de Aplicações Corporativas

Martin Fowler
com a colaboração de
David Rice,
Matthew Foemmel,
Edward Hieatt,
Robert Mee,
Randy Stafford

bookman®

Martin Fowler é o pesquisador chefe da ThoughtWorks, empresa de desenvolvimento e integração de aplicações corporativas. Foi pioneiro no uso da tecnologia de objetos na criação de aplicações corporativas multicamadas na década de 1980. É o autor de *UML Essencial* e *Refatoração*, traduzidos e publicados no Brasil pela Bookman Editora, e de *Planning Extreme Programming* e *Analysis Patterns*, publicados pela Addison-Wesley.

A634a Fowler, Martin

Padrões de arquitetura de aplicações corporativas [recurso eletrônico] / Martin Fowler com a colaborapção de David Rice ... [et al.] ; tradução Acauan Fernandes. – Dados eletrônicos. – Porto Alegre : Bookman, 2007.

Editado também como livro impresso em 2006.
ISBN 978-85-7780-064-3

1. Computação – Projeto – Sistema. 2. Computadores. – Arquitetura. I. Título.

CDU 004.41

Catalogação na publicação: Juliana Lagôas Coelho – CRB 10/1798

Martin Fowler
com a colaboração de
David Rice,
Matthew Foemmel,
Edward Hieatt,
Robert Mee,
Randy Stafford

Padrões de Arquitetura de Aplicações Corporativas

Tradução:
Acauan Fernandes
Mestre em Ciência da Computação pela UFRGS

Consultoria, supervisão e revisão técnica desta edição:
Jonas Knopman
D. Sc. COPPE/UFRJ
Analista, Pesquisador e Professor do NCE/UFRJ

Versão impressa
desta obra: 2006

bookman®

2007

Obra originalmente publicada sob o título
Patterns of Enterprise Application Architecture, 1st Edition

ISBN 0-321-12742-0

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

Capa: *Gustavo Demarchi*

Leitura final: *Mareci Pedron de Oliveira*

Supervisão editorial: *Arysinha Jacques Affonso e Rachel Garcia Valdez*

Editoração eletrônica: *Laser House*

IMPRESSO NO BRASIL
PRINTED IN BRAZIL

Para Denys William Fowler, 1922-2000
in memoriam

— Martin

Esta página foi deixada em branco intencionalmente.

Prefácio

No verão de 1999, fui para Chicago prestar consultoria em um projeto em execução pela ThoughtWorks, uma empresa de desenvolvimento de aplicações pequena, mas em rápido crescimento. Tratava-se de um daqueles projetos ambiciosos de aplicações corporativas: um sistema *de apoio* para contratos de *leasing*. O sistema lida, basicamente, com tudo o que acontece com o *leasing* após você ter assinado na linha pontilhada: enviar contas, lidar com a atualização de um dos bens do *leasing*, procurar pessoas que não pagam suas contas em dia e descobrir o que acontece quando alguém devolve os bens mais cedo. Isso não parece muito difícil até que você perceba que contratos de *leasing* têm variações infinitas e são terrivelmente complicados. A "lógica" de negócio raramente se ajusta a algum padrão lógico porque, antes de qualquer coisa, ela é escrita por pessoas de negócio com a finalidade de capturar negócios, em que pequenas variações estranhas pode fazer toda a diferença em fechar ou não uma operação. Cada uma dessas pequenas conquistas acrescenta ainda mais complexidade ao sistema.

Este é o tipo de coisa que me deixa entusiasmado: como pegar toda essa complexidade e propor um sistema de objetos que possa tornar o problema mais tratável. De fato, acredito que o benefício principal dos objetos está em tornar tratável uma lógica complexa. Desenvolver um bom *Modelo do Domínio* (126) para um problema de negócio complexo é difícil, mas maravilhosamente gratificante.

Mas este não é o fim do problema. Nosso modelo de domínio tinha que ser persistido em um banco de dados e, como em muitos projetos, estávamos usando um banco de dados relacional. Também tínhamos que conectar esse modelo a uma interface de usuário, fornecer suporte para permitir a aplicações remotas usarem nosso *software* e integrá-lo com pacotes de terceiros. Tudo isso em uma nova tecnologia chamada J2EE, a qual ninguém no mundo tinha experiência real de uso.

Embora esta tecnologia fosse nova, tínhamos o benefício da experiência. Eu vinha trabalhando com esse tipo de coisa há muito tempo com C++, Smalltalk e CORBA. Muitos dos membros da ThoughtWorks tinham muita experiência com Forte.

Nós já tínhamos as idéias centrais da arquitetura em nossas cabeças, e só precisávamos descobrir como aplicá-las a J2EE. Olhando para o projeto três anos mais tarde, vemos que não está perfeito, mas suportou muito bem a passagem do tempo.

É para este tipo de situação que este livro foi escrito. Ao longo dos anos tenho visto muitos projetos de aplicações corporativas. Estes projetos muitas vezes contêm idéias semelhantes de projeto que se mostraram efetivas na manipulação da complexidade inevitável que as aplicações corporativas possuem. Este livro é o ponto de partida para capturar essas idéias na forma de padrões.

O livro é organizado em duas partes, sendo a primeira um conjunto de capítulos descritivos sobre vários tópicos importantes sobre o projeto de aplicações corporativas. Esses capítulos introduzem vários problemas da arquitetura de aplicações corporativas e suas soluções. No entanto, eles não entram em muitos detalhes a respeito dessas soluções. Os detalhes das soluções estão na segunda parte, organizada na forma de padrões. Esses padrões são uma referência, e não espero que você os leia do começo ao fim. Meu objetivo é que você leia todos os capítulos descritivos da Parte 1 para ter uma idéia global do livro. Então você mergulha nos capítulos de padrões da Parte 2, na medida dos seus interesses e necessidades. Assim, este é, ao mesmo tempo, um livro descritivo curto e um livro de referência mais longo.

O tema da obra é o projeto de aplicações corporativas. Aplicações corporativas dizem respeito à exibição, manipulação e armazenamento de grandes quantidades de dados, freqüentemente complexos, e ao suporte ou automação dos processos de negócio que fazem uso desses dados. Como os sistemas de reserva, sistemas financeiros, cadeia de suprimentos e muitos outros que suportam os modernos ambientes de negócios. As aplicações corporativas têm seus próprios desafios e soluções e são diferentes dos sistemas embutidos, dos sistemas de controle, de telecomunicação ou do *software* para *desktops*. Assim, se você trabalhar nesses outros campos, não há realmente nada neste livro que lhe interesse (a menos que queira apenas saber o que são aplicações corporativas.) Para um livro geral sobre arquitetura de *software*, eu recomendaria [POSA].

Há muitos aspectos referentes à arquitetura na construção de aplicações corporativas. Receio que este livro não seja um guia abrangente de todos eles. Na construção de *software*, acredito firmemente no desenvolvimento iterativo. No âmago do desenvolvimento iterativo está a noção de que você deve distribuir versões do *software* assim que tiver algo útil para o usuário, ainda que estas distribuições não estejam completas. Embora haja muitas diferenças entre escrever um livro e escrever *software*, penso que esta noção é compartilhada em ambos os casos. Isso dito, este livro é um compêndio incompleto, porém (acredito eu) útil, de conselhos sobre arquitetura de aplicações corporativas. Os principais tópicos sobre os quais eu falo são:

- Dispor aplicações corporativas em camadas
- Estruturar lógica de domínio (negócio)
- Estruturar uma interface Web de usuário
- Conectar módulos em memória (particularmente objetos) com um banco de dados relacional
- Manipular o estado da sessão em ambientes sem estado
- Princípios de distribuição

A lista de coisas sobre as quais não falo é longa. Eu realmente gostaria de escrever sobre a organização de validações, incorporação de mensagens e comunicação assíncrona, segurança, manipulação de erros, clusterização, integração de aplicações, refatoração de arquitetura, estruturação de interfaces de usuário do tipo cliente rico, entre outros tópicos. Entretanto, devido a restrições de espaço e tempo e falta de reflexão suficiente, você não irá encontrá-los neste livro. Espero ver alguns padrões para este trabalho em um futuro próximo. Talvez eu escreva um segundo volume deste livro e trate esses tópicos, ou talvez alguma outra pessoa preencha esta e outras lacunas.

Destas, a comunicação baseada em mensagens é uma questão particularmente importante. As pessoas que integram múltiplas aplicações estão cada vez mais fazendo uso de abordagens de comunicação baseada em mensagens assíncronas. Há muito a ser dito a favor de seu uso dentro de uma aplicação também.

Este livro não pretende ser específico para nenhuma plataforma de *software* em particular. Deparei-me pela primeira vez com estes padrões quando estava trabalhando com Smalltalk, C++ e CORBA no final da década de 80 e início da de 90. No final da década de 90, comecei a realizar trabalhos em Java e descobri que estes padrões se aplicavam bem tanto nos sistemas antigos Java/CORBA quanto no trabalho mais recente baseado em J2EE. Mais recentemente fiz um trabalho introdutório com a plataforma .NET da Microsoft e descobri que novamente os padrões se aplicavam. Meus colegas de ThoughtWorks também apresentaram suas experiências, especialmente com Forte. Não posso afirmar que os padrões são genéricos para todas as plataformas já usadas ou ainda em uso com aplicações corporativas, mas até agora estes padrões têm mostrado recorrência suficiente para serem úteis.

Forneci exemplos de código para a maioria dos padrões. Minha escolha de linguagem para eles foi baseada no que acho que a maioria dos leitores provavelmente será capaz de ler e entender. Java é uma boa escolha aqui. Qualquer um que possa ler C ou C++ pode ler Java e, além disso, Java é menos complexa que C++. Basicamente, a maioria dos programadores C++ consegue ler programas escritos em Java, mas o contrário não é verdadeiro. Sou um entusiasta de objetos, então inevitavelmente tendo para uma linguagem OO. Em conseqüência, a maioria dos exemplos de código estão em Java. Enquanto eu trabalhava no livro, a Microsoft começou a firmar seu ambiente .NET, e sua linguagem C# tem a maioria das propriedades de Java que a tornam interessante para um autor. Assim, escrevi alguns dos exemplos de código também em C#, embora isso introduzisse certo risco já que os desenvolvedores não têm muita experiência com .NET, e as técnicas para usá-lo eficientemente são menos maduras. Ambas são linguagens baseadas em C, de modo que se você conseguir ler uma deve ser capaz de ler ambas, mesmo se não gostar profundamente dessa linguagem ou plataforma. Meu objetivo era usar uma linguagem que a grande maioria dos desenvolvedores de *software* pudesse ler, ainda que esta não fosse a sua linguagem principal ou preferida. (Minhas desculpas àqueles que gostam de Smalltalk, Delphi, Visual Basic, Perl, Python, Ruby, COBOL, ou qualquer outra linguagem. Eu sei que vocês acham que conhecem uma linguagem melhor do que Java ou C#. Tudo o que posso dizer é que eu também!)

Os exemplos estão lá para inspirar e explicar as idéias nos padrões. Eles não são soluções enlatadas. Em todos os casos você precisará efetuar uma quantidade razoável de trabalho para ajustá-los à sua aplicação. Padrões são pontos de partida úteis, e não destinos.

Para Quem é Este Livro

Escrevi este livro para programadores, projetistas e arquitetos que estejam construindo aplicações corporativas e que queiram melhorar sua compreensão sobre questões arquiteturais ou sua comunicação sobre elas.

Presumo que a maioria dos meus leitores se dividirá em dois grupos: aqueles com necessidades moderadas que querem construir seu próprio *software* e leitores com necessidades maiores que usarão uma ferramenta. Para aqueles com necessidades moderadas, meu objetivo é que estes padrões sirvam como uma iniciação. Em muitas áreas, você precisará de mais do que padrões, mas lhe darei uma vantagem inicial neste campo muito maior do que a que tive. Aos usuários de ferramentas espero que este livro dê alguma idéia do que acontece por baixo dos panos e também ajude-os a escolher quais padrões suportados por ferramentas usar. Por exemplo, se você usa uma ferramenta para o mapeamento objeto-relacional, ainda assim tem que tomar decisões sobre como mapear certas situações. Ler os padrões deve lhe dar alguma orientação nesse sentido.

Há uma terceira categoria, aqueles exigentes e que querem construir seu próprio *software*. A primeira coisa que eu diria neste caso é para considerarem com atenção a opção de usar ferramentas. Já vi mais de um projeto virar um longo exercício de construção de *frameworks*, o que não era o objetivo inicial do projeto. Se você ainda não estiver convencido, vá em frente. Lembre-se nesse caso de que muitos dos exemplos de código neste livro são deliberadamente simples para ajudar a compreensão, e você descobrirá que precisará fazer muitos ajustes para lidar com as demandas maiores que enfrentar.

Já que padrões são soluções comuns para problemas recorrentes, há uma boa chance de que você já tenha se deparado com alguns deles. Se você vem trabalhando com aplicações corporativas há algum tempo, pode conhecer a maioria deles. Não reivindico a apresentação de nada novo neste livro. Na verdade, afirmo o oposto – este é um livro de (para nossa indústria) idéias antigas. Se você é novo neste campo, espero que o livro o ajude a aprender sobre estas técnicas. Se você estiver familiarizado com as técnicas, espero que o livro o ajude a comunicar e ensiná-las a outros. Uma parte importante dos padrões é tentar construir um vocabulário comum, de modo que você possa dizer que essa classe é uma *Fachada Remota* (368) e outros projetistas saibam o que você quer dizer.

Agradecimentos

O que está escrito aqui deve-se muito às pessoas que trabalharam comigo de diversas formas no decorrer dos anos. Muitas pessoas ajudaram de várias maneiras diferentes. Muitas vezes não consigo lembrar coisas importantes que as pessoas disseram e que estão neste livro, mas posso reconhecer aquelas contribuições de que me lembro.

Começarei com meus colaboradores. David Rice, um dos meus colegas na ThoughtWorks, deu uma grande contribuição – cerca de um décimo do livro. Enquanto trabalhávamos duro para atender o prazo (ao mesmo tempo que ele dava suporte a um cliente), tivemos várias conversas tarde da noite, via troca de mensagens, em que ele confessou finalmente entender por que escrever um livro era, ao mesmo tempo, tão difícil e tão compulsivo.

Matt Foemmel é outro colega da ThoughtWorks, e embora o Ártico vá precisar de ar condicionado antes que ele escreva prosa por diversão, foi um grande colaborador de exemplos de código (assim como um crítico bastante sucinto do livro.) Fiquei satisfeito por Randy Stafford colaborar com *Camada de Serviço* (141), pois ele é um firme defensor dela. Também gostaria de agradecer Edward Hieatt e Rob Mee por suas colaborações, que se originaram da percepção de Rob de uma lacuna no texto enquanto fazia revisão. Ele se tornou meu revisor favorito: ele não apenas percebe que algo está faltando como também escreve uma seção para consertá-lo!

Como de costume, devo mais do que posso dizer à minha equipe de revisores oficiais de primeira classe:

John Brewer	Rob Mee
Kyle Brown	Gerard Meszaros
Jens Coldewey	Dirk Riehle
John Crupi	Randy Stafford
Leonard Fenster	David Siegel
Alan Knight	Kai Yu

Eu poderia quase listar a lista telefônica da ThoughtWorks aqui, devido à quantidade de colegas que me ajudaram, conversando sobre seus próprios projetos e experiências. Muitos padrões se formaram na minha mente porque tive a oportunidade de conversar com projetistas talentosos, de modo que tenho de agradecer à companhia inteira.

Kyle Brown, Raquel Reinitz e Bobby Woolf não mediram esforços para ter longas e detalhadas sessões de revisão comigo na Carolina do Norte. Seu pente fino introduziu todo o tipo de bom senso aqui. Apreciei em especial as diversas e longas conversas telefônicas com Kyle que contribuiu mais do que posso registrar.

No início de 2000, preparei uma palestra para a Java One, com Alan Knight e Kai Yu, que foi a gênese deste material. Além de agradecê-los, também devo agradecer Josh Mackenzie, Rebecca Parsons e Dave Rice pela ajuda em refinar essas palestras e pelas contribuições posteriores. Jim Newkirk foi de grande valia ao ajudar a me familiarizar com o mundo .NET.

Aprendi muito com as pessoas que trabalham nesta área por meio de boas conversas e colaborações. Gostaria de agradecer em especial a Colleen Roe, David Muirhead e Randy Stafford por compartilharem seu trabalho no exemplo do Foodsmart na Gemstone. Também tive ótimas conversas no *workshop* em Crested Butte que Bruce Eckel organizou e devo agradecer a todas as pessoas que compareceram àquele evento nos últimos anos. Joshua Kerievsky não teve tempo de fazer uma revisão completa, mas foi um excelente consultor de padrões.

Como sempre, tive a notável ajuda do grupo de leitura da UIUC. Meus agradecimentos: Ariel Gertzenstein, Bosko Zivaljevic, Brad Jones, Brian Foote, Brian Marick, Federico Balaguer, Joseph Yoder, John Brant, Mike Hewner, Ralph Johnson e Weerasak Witthawaskul.

Dragos Manolescu, *ex-hitman* da UIUC, juntou seu próprio grupo para me dar retorno. Meus agradecimentos a Muhammad Anan, Brain Doyle, Emad Ghosheh, Glenn Graessle, Daniel Hein, Prabhaharan Kumarakulasingam, Joe Quint, John Reinke, Kevin Reynolds, Sripriya Srinivasan e Tirumala Vaddiraju.

Kent Beck me deu muitas boas idéias. Lembro de que foi ele que deu o nome ao padrão *Caso Especial* (462). Jim Odell foi o responsável por me introduzir no

mundo da consultoria, ensino e escrita – nenhum agradecimento jamais fará justiça a essa ajuda.

Enquanto escrevia este livro, coloquei esboços na Web. Durante esse tempo, muitas pessoas me enviaram *e-mails* apontando problemas, fazendo perguntas ou falando sobre alternativas. Essas pessoas incluem Michael Banks, Mark Bernstein, Graham Berrisford, Bjorn Beskow, Bryan Boreham, Sean Broadley, Peris Brodsky, Paul Campbell, Chester Chen, John Coakley, Bob Corrick, Pascal Costanza, Andy Czerwonka, Martin Diehl, Daniel Drasin, Juan Gomez Duaso, Don Dwiggins, Peter Foreman, Russell Freeman, Peter Gassmann, Jason Gorman, Dan Green, Lars Gregori, Rick Hansen, Tobin Harris, Russel Healey, Christian Heller, Richard Henderson, Kyle Hermenean, Carsten Heyl, Akira Hirasawa, Eric Kaun, Kirk Knoernschild, Jesper Ladegaard, Chris Lopez, Paolo Marino, Jeremy Miller, Ivan Mitrovic, Thomas Neumann, Judy Obee, Paolo Parovel, Trevor Pinkney, Tomas Restrepo, Joel Rieder, Matthew Roberts, Stefan Roock, Ken Rosha, Andy Schneider, Alexandre Semenov, Stan Silvert, Geoff Soutter, Volker Termath, Christopher Thames, Volker Turau, Knut Wannheden, Marc Wallace, Stefan Wenig, Brad Wiemerslage, Mark Windholtz, Michael Yoon.

Há muitos outros colaboradores cujos nomes eu nunca soube ou não consigo lembrar, mas meus agradecimentos não são menos sinceros.

Meu maior agradecimento é, como sempre, para minha esposa Cindy, cuja companhia aprecio muito mais do que alguém poderá apreciar este livro.

Colofão

Este é o primeiro livro que escrevi usando XML e tecnologias relacionadas. O texto principal foi escrito como uma série de documentos XML usando o fiel TextPad. Também usei um DTD feito em casa. Enquanto estava trabalhando, usei XSLT para gerar as páginas Web para o *site* HTML. Para os diagramas, baseei-me no velho amigo Visio, usando os excelentes gabaritos UML de Pavel Hruby (muito melhores que aqueles com a ferramenta. Tenho um *link* no meu Web *site*, se você os quiser.) Escrevi um pequeno programa que importava automaticamente os exemplos de código para a saída, o que me poupou do pesadelo costumeiro de recortar e colar código. No meu primeiro esboço, experimentei XSL-FO com Apache FOP. Na ocasião, ele não se mostrou muito apropriado para a tarefa, então para trabalhos posteriores escrevi *scripts* em XSLT e em Ruby para importar o texto para o FrameMaker.

Usei diversas ferramentas *open source* enquanto trabalhava neste livro – em especial, JUnit, NUnit, ant, Xerces, Xalan, Tomcat, Jboss, Ruby e Hsql. Meus agradecimentos para os muitos desenvolvedores dessas ferramentas. Também houve uma longa lista de ferramentas comerciais. Em especial, baseei-me no Visual Studio for .NET e no excelente Idea da IntelliJ – a primeiro IDE que me entusiasmou desde Smalltalk – para Java.

O livro foi adquirido para a Addison-Wesley por Mike Hendrickson que, assistido por Ross Venables, supervisionou sua publicação. Comecei o trabalho no manuscrito em novembro de 2000 e lancei o esboço final para produção em junho de 2002. Enquanto escrevo isto, o livro está pronto para o lançamento em novembro de 2002 na OOPSLA.

Sarah Weaver foi a editora de produção, coordenando a edição, composição, revisão, indexação e produção dos arquivos finais. Dianne Wood foi a editora de cópia,

executando o complicado trabalho de polir meu inglês sem introduzir nenhum refinamento desagradável. Kim Arney Mulcahy compôs o livro no formato que você vê aqui, organizou os diagramas, configurou o texto no Sabon e preparou os arquivos Framemaker finais para a impressão. O formato do texto é baseado no formato que usamos para *Refatoração**. Cheryl Ferguson fez a revisão das páginas e desentocou todos os erros que tinham escorregado pelas fendas. Irv Hershman preparou o índice.

A foto da capa

Durante os dois anos em que escrevi este livro uma construção mais importante era feita em Boston. A ponte Leonard P. Zakim Bunker Hill (tente colocar esse nome em uma placa de sinalização à beira da estrada) vai substituir a feia ponte de dois andares usada pela rodovia interestadual 93 para atravessar o rio Charles. A ponte Zakim é uma ponte estaiada, um estilo incomum nos EUA mas muito utilizado na Europa. A ponte Zakim não é longa, mas é a mais longa do gênero no mundo e a primeira dos EUA com um projeto assimétrico. É linda, o que não impede que eu brinque com Cindy sobre a previsão de Henry Petroski de que vamos ter uma grande falha, em breve, em uma ponte estaiada.

Martin Fowler, Melrose, Massachussets, agosto de 2002
HTTP://martinfowler.com

* N. de R.T.: Refatoração, Aperfeiçoando o Projeto de Código Existente, Bookman, 2004.

Esta página foi deixada em branco intencionalmente.

Sumário

Esta página foi deixada em branco intencionalmente.

Introdução

Caso você não tenha percebido, criar sistemas computacionais é uma tarefa difícil. À medida que a complexidade do sistema aumenta, a tarefa de construir o *software* fica muito mais difícil. Assim como em qualquer profissão, só progredimos aprendendo, tanto com os nossos erros quanto com nossos sucessos. Este livro descreve parte desse aprendizado de forma que, espero, o ajudará a aprender essas lições mais rapidamente do que aprendi, ou para se comunicar com os outros de modo mais eficaz do que eu pude fazer antes de descrever estes padrões.

Nesta introdução, quero estabelecer o escopo do livro e fornecer alguns dos fundamentos que servirão de base para as idéias aqui descritas.

Arquitetura

A indústria de *software* se delicia em pegar palavras e estendê-las em uma miríade de significados sutilmente contraditórios. Uma das maiores sofredoras é "arquitetura". Vejo "arquitetura" com uma daquelas palavras que soam impressionantes, usadas principalmente para indicar que estamos falando algo importante. Contudo, sou pragmático o suficiente para não deixar que meu cinismo atrapalhe o desafio de atrair as pessoas para meu livro. :-)

"Arquitetura" é um termo que muitas pessoas, com pouca concordância entre si, tentam definir. Existem dois elementos comuns: um é a decomposição em alto nível de um sistema em suas partes; o outro são decisões difíceis de alterar. Percebe-se também, cada vez mais, que não há apenas um único modo de especificar a arquitetura de um sistema; ao contrário, existem diversas arquiteturas em um sistema, e a visão do que é significativo em termos de arquitetura pode mudar durante o ciclo de vida de um sistema.

De tempos em tempos, Ralph Johnson envia uma mensagem verdadeiramente notável para uma lista de correio, e ele enviou uma dessas quando eu estava termi-

nando o esboço deste livro. Nessa mensagem, ele levantou a questão de que a arquitetura é subjetiva, uma compreensão do projeto de um sistema compartilhada pelos desenvolvedores experientes em um projeto. Esta compreensão compartilhada freqüentemente se apresenta na forma dos componentes mais importantes do sistema e de como eles interagem. Também diz respeito a decisões, pois os desenvolvedores gostariam de tomar as decisões certas desde o início, já que elas são vistas como difíceis de alterar. A subjetividade aparece aqui também porque, se você descobrir que algo é mais fácil de alterar do que você supôs, então isso não é mais arquitetural. No final, a arquitetura se resume a coisas importantes – seja lá o que isso signifique.

Neste livro, apresento minha percepção das partes mais importantes de uma aplicação corporativa e das decisões que eu gostaria de poder tomar acertadamente logo no princípio do projeto. O padrão arquitetural de que mais gosto é o de camadas, o qual descrevo em mais detalhes no Capítulo 1. Este livro, é portanto, sobre como decompor uma aplicação corporativa em camadas e como estas camadas trabalham juntas. A maioria das aplicações corporativas não-triviais usa alguma forma de arquitetura em camadas, mas, em algumas situações, outras abordagens, tais como canais e filtros, são valiosas. Não discuto essas situações, focalizando em vez disso o contexto de uma arquitetura em camadas porque esta é a mais amplamente útil.

Alguns dos padrões deste livro podem, de modo aceitável, ser chamados de padrões arquiteturais, visto que representam decisões significativas sobre essas partes. Outros são mais sobre projeto e o ajudam a conceber essa arquitetura. Não faço nenhuma tentativa decidida de separar os dois, visto que é tão subjetivo o que venha a ser ou não um padrão arquitetural.

Aplicações Corporativas

Muitas pessoas escrevem *software* para computadores, e chamamos a tudo isso de desenvolvimento de *software*. Entretanto, existem diferentes tipos de *software*, cada qual com seus próprios desafios e complexidades. Isso torna-se evidente quando converso com alguns dos meus amigos do ramo das telecomunicações. Em alguns aspectos, as aplicações corporativas são muito mais fáceis do que o *software* para telecomunicações - não temos problemas de *multithreading* muito complicados e não temos a integração de *hardware* e *software*. Em outros aspectos, no entanto, elas são muito mais difíceis. As aplicações corporativas muitas vezes têm dados complexos – e uma quantidade grande deles – para trabalhar, aliados a regras de negócio que não passam em nenhum teste de raciocínio lógico. Embora algumas técnicas e padrões sejam relevantes para todos os tipos de *software*, muitos são relevantes apenas para um ramo específico.

Em minha carreira, tenho me concentrado em aplicações corporativas, de modo que todos os meus padrões aqui são sobre elas. (Outros termos para aplicações corporativas incluem "sistemas de informação" ou, para aqueles com boa memória, "processamento de dados".) Mas o que eu quero dizer com o termo "aplicação corporativa"? Não consigo dar uma definição precisa, mas posso dar algumas pistas do meu significado.

Começarei com exemplos. As aplicações corporativas incluem folhas de pagamento, registros de pacientes, rastreamento de remessas, análise de custos, pontuação de crédito, seguro, cadeia de suprimentos, contabilidade, serviço de atendimento ao cliente e comércio internacional. As aplicações corporativas não incluem a inje-

ção de combustível em automóveis, processadores de texto, controladores de elevadores, controladores de fábricas químicas, chaves telefônicas, sistemas operacionais, compiladores e jogos.

As aplicações corporativas normalmente envolvem **dados persistentes**. Os dados são persistentes porque precisam estar disponíveis entre múltiplas execuções do programa - de fato, eles normalmente precisam persistir por vários anos. Além disso, durante este tempo haverá muitas alterações nos programas que usam estes dados. Eles freqüentemente durarão mais do que o *hardware* que originalmente os criou e sobreviverão aos sistemas operacionais e compiladores. Durante esse período, haverá muitas alterações na estrutura dos dados a fim de armazenar novas informações sem perturbar as já existentes. Ainda que haja uma alteração fundamental, e a companhia instale uma aplicação completamente nova para lidar com o trabalho, os dados têm que ser migrados para a nova aplicação.

Normalmente existe **uma grande quantidade de dados** – um sistema moderado terá mais de 1 GB de dados organizados em dezenas de milhões de registros – tantos, que a gerência destes dados é uma parte importante do sistema. Os sistemas mais antigos usavam estruturas de arquivos indexados, tais como ISAM e VSAM da IBM. Os sistemas modernos normalmente usam bancos de dados, na sua maioria, relacionais. O projeto e carga desses bancos de dados se tornou por si só uma profissão.

Normalmente muitas pessoas **acessam os dados concorrentemente.** Em muitos sistemas isso pode ser menos do que uma centena de pessoas, mas para sistemas baseados na Web que conversam pela Internet, isso aumenta em ordens de magnitude. Com tantas pessoas acessando o sistema, existem questões bem-definidas para assegurar que todas elas possam fazê-lo apropriadamente. Mas mesmo sem tantas pessoas, ainda existem problemas para garantir que duas pessoas não acessem os mesmos dados ao mesmo tempo de uma forma que possa causar erros. As ferramentas para o controle de transações absorvem um pouco dessa carga, mas muitas vezes é impossível esconder esse problema dos desenvolvedores das aplicações.

Com tantos dados, normalmente há uma grande quantidade de **telas de interface com o usuário** para lidar com eles. Não é raro haver centenas de telas diferentes. Os usuários das aplicações corporativas vão dos ocasionais àqueles que usam regularmente o sistema e, em geral, eles têm pouco conhecimento técnico. Assim, os dados têm que ser apresentados de várias formas diferentes para atender a diferentes objetivos. Os sistemas muitas vezes têm uma grande quantidade de processamento em lote, o que é fácil de esquecer quando enfocamos os casos de uso que enfatizam a interação com o usuário.

As aplicações corporativas raramente vivem isoladas. Elas freqüentemente precisam **falar com outras aplicações corporativas** espalhadas pela corporação. Os diversos sistemas são construídos em épocas diferentes com diferentes tecnologias e até mesmo os mecanismos de colaboração são diferentes: arquivos de dados em COBOL, CORBA, sistemas de mensagens, etc. De vez em quando a empresa tenta integrar seus diferentes sistemas usando uma tecnologia comum de comunicação. É claro que o trabalho quase nunca chega ao fim, de modo que diversos esquemas de integração diferentes coexistem ao mesmo tempo. Isso fica ainda pior quando a empresa tenta se integrar também aos seus parceiros de negócio.

Mesmo se uma companhia unificar a tecnologia para a integração, ela ainda terá problemas com as diferenças nos processos de negócios e com a **dissonância conceitual** com os dados. Uma divisão da companhia pode achar que um cliente é alguém com quem ela tem um contrato vigente; outra divisão também conta aqueles clientes com os quais a companhia já teve um contrato, mas que não tem mais; uma

outra conta a venda de produtos mas não a venda de serviços. Pode parecer fácil lidar com essas diferenças, mas quando você tem centenas de registros nos quais cada campo pode ter um significado sutilmente diferente, a grandeza do problema torna-se um desafio – mesmo se a única pessoa que saiba exatamente o que o campo significa ainda esteja na companhia. (E, é claro, tudo isso muda sem aviso prévio.) A conseqüência disso é que os dados têm de ser constantemente lidos, alterados e gravados em todos os tipos de formatos sintática e semanticamente diferentes.

A seguir, temos a questão do que está por trás do termo "lógica de negócio." Acho este um termo curioso porque existem poucas coisas que são menos lógicas do que a lógica de negócio. Quando você cria um sistema operacional, esforça-se para manter toda a coisa lógica. No entanto, as regras de negócio são simplesmente impostas a você e, sem um grande esforço político, não há nada que você possa fazer para alterá-las. Você tem que lidar com um conjunto aleatório de condições estranhas que muitas vezes interagem umas com as outras de formas surpreendentes. É claro que elas são assim por alguma razão: algum vendedor negociou receber uma prestação anual dois dias mais tarde do que o usual porque isso se ajustava ao fluxo de caixa do cliente e, assim, ele conseguiu ganhar alguns milhões de dólares no negócio. Alguns poucos milhares desses casos especiais é o que leva à complexa **"ilógica" de negócio,** que torna tão difícil o *software* de negócios. Nesta situação você tem que organizar a lógica de negócio tão eficazmente quanto puder, porque a única coisa certa é que ela mudará com o decorrer do tempo.

Para algumas pessoas, o termo "aplicação corporativa" sugere um sistema grande. Entretanto, é importante lembrar que nem todas as aplicações corporativas são grandes, ainda que elas possam agregar muito valor ao negócio. Muitas pessoas entendem que não vale a pena se preocupar com sistemas pequenos, e, até certo, ponto elas tem razão. Se um sistema pequeno falhar, ele normalmente faz menos barulho do que um sistema grande. Ainda assim, acredito que essa forma de pensar tende a não dar tanta atenção ao efeito cumulativo da coexistência de vários projetos pequenos. Se você puder fazer coisas que melhorem os projetos pequenos, o efeito cumulativo pode ser muito significativo para o negócio, especialmente porque projetos pequenos muitas vezes têm um valor desproporcional. Uma das melhores coisas que você pode fazer é transformar um projeto grande em um pequeno simplificando a arquitetura e o processo.

Tipos de Aplicações Corporativas

Quando discutimos como projetar aplicações corporativas e quais padrões usar, é importante perceber que as aplicações corporativas são inteiramente diferentes umas das outras e que problemas diferentes levam a maneiras diferentes de fazer as coisas. Tenho um conjunto de campainhas de alarme que disparam quando as pessoas dizem "Sempre faça isso". Para mim, muito do desafio (e interesse) do projeto está em conhecer as alternativas e julgar os compromissos envolvidos em usar esta ou aquela. Há muitas alternativas, mas escolherei aqui três pontos nesta área enorme.

Considere um varejista *online* B2C (negócio direto com o cliente): as pessoas navegam e – com sorte e um carrinho de compras – compram. Em um sistema como esse, precisamos ser capazes de lidar com um grande volume de usuários, então nossa solução precisa não apenas ser razoavelmente eficiente em termos de recursos usados, mas também escalável, de modo que você possa aumentar a carga adicionando

mais *hardware*. A lógica de domínio para tal aplicação pode ser bastante direta: obter pedido, alguns cálculos relativamente simples de preço e remessa e notificação de envio. Queremos que qualquer um consiga acessar o sistema facilmente, então isto sugere uma apresentação Web bastante genérica que possa ser usada pela maior faixa possível de navegadores. A fonte de dados inclui um banco de dados para armazenar pedidos e talvez alguma comunicação com um sistema de estoque para ajudar com informações sobre disponibilidade e entrega.

Compare isso com um sistema que automatize o processo de contratos de *leasing*. De certo modo, este é um sistema muito mais simples do que o do varejista B2C porque há muito menos usuários – não mais do que em torno de uma centena de cada vez. Ele é mais complicado na lógica de negócio. Calcular contas mensais em um *lease*, tratar eventos tais como retornos antecipados e pagamentos atrasados e validar dados quando um *lease* é reservado com antecedência, são todas tarefas complicadas, já que muito da competição na indústria de *leasing* vem na forma de pequenas variações sobre negócios feitos no passado. Um domínio de negócio complexo como esse é desafiador porque as regras são arbitrárias.

Tal sistema apresenta ainda uma interface com o usuário mais complexa. No mínimo, isso significa uma interface HTML muito mais elaborada com telas mais complicadas e em maior número. Com freqüência esses sistemas têm demandas de interface com o usuário que levam os usuários a querer uma apresentação mais sofisticada do que o permitido por um *front end* HTML, assim é necessária uma interface mais convencional, do tipo cliente rico. Uma interação mais complexa com o usuário também leva a um comportamento transacional mais complicado: Reservar com antecedência um *lease* pode levar uma hora ou duas, período durante o qual o usuário está em uma transação lógica. Também observamos um esquema de banco de dados complexo com talvez duas centenas de tabelas e conexões com pacotes para avaliação de bens e determinação de preços.

Um terceiro exemplo é um sistema simples de registro de gastos de uma pequena companhia. Tal sistema tem poucos usuários e lógica simples, podendo ser disponibilizado facilmente para a companhia com uma apresentação HTML. A única fonte de dados são algumas poucas tabelas em um banco de dados. Tão simples quanto possa parecer, um sistema como este não é desprovido de desafios. Você tem que construí-lo muito rapidamente e tem que ter em mente que ele pode crescer à medida que as pessoas queiram calcular reembolsos, incluí-los na folha de pagamento, entender implicações de impostos, fornecer relatórios para o diretor financeiro, conectar o sistema aos serviços Web de reserva de passagens aéreas, e assim por diante. A tentativa de usar a arquitetura de qualquer um dos dois outros exemplos irá retardar o desenvolvimento deste. Se um sistema traz benefícios para o negócio (como todas as aplicações corporativas deveriam trazer), atrasar esses benefícios custa dinheiro. Entretanto, você não quer tomar decisões agora que dificultem uma futura expansão do sistema. Contudo, se você acrescentar flexibilidade agora e o fizer de forma equivocada, a complexidade adicionada objetivando a flexibilidade pode, em verdade, tornar mais difícil estender o sistema no futuro e pode atrasar o desenvolvimento atual retardando assim os benefícios. Embora tais sistemas sejam pequenos, a maioria das empresas têm muitos deles, de modo que o efeito cumulativo de uma arquitetura inapropriada pode ser significativo.

Cada um desses três exemplos de aplicações corporativas tem suas dificuldades e todas elas são diferentes. Como conseqüência, é difícil propor uma arquitetura única que seja apropriada para as três situações. Escolher uma arquitetura significa entender os problemas específicos do seu sistema e definir um projeto apropriado ba-

seado nesse entendimento. Eis por que, neste livro, não dou uma solução única para suas necessidades corporativas. Em vez disso, muitos dos padrões se referem a escolhas e alternativas. Mesmo quando você escolher um padrão específico, terá que modificá-lo para satisfazer suas demandas. Você não pode criar *software* corporativo sem pensar, e tudo que qualquer livro pode fazer é lhe dar mais informações sobre as quais basear suas decisões.

Se isso se aplica a padrões, também se aplica a ferramentas. Embora obviamente faça sentido escolher um conjunto tão pequeno quanto possível de ferramentas para desenvolver aplicações, você também tem que reconhecer que cada ferramenta diferente é melhor para uma tarefa diferente. Tome cuidado para não usar uma ferramenta que seja apropriada para um tipo diferente de aplicação – pode atrapalhar mais do que ajudar.

Pensando em Desempenho

Muitas decisões relacionadas à arquitetura dizem respeito ao desempenho. Para a maioria das questões relacionadas ao desempenho, prefiro obter um sistema no ar e rodando, instrumentá-lo e então usar um processo disciplinado de otimização baseado em medições. Entretanto, algumas decisões de arquitetura afetam o desempenho de um modo tal que é difícil consertá-lo com otimizações posteriores. E mesmo quando é fácil consertá-lo mais tarde, as pessoas envolvidas no projeto se preocupam desde cedo com tais decisões.

É sempre difícil falar sobre desempenho em um livro como este. A dificuldade advém do fato de que qualquer conselho sobre desempenho não deve ser tratado como verdade absoluta até que seja avaliado na sua própria configuração. Tenho visto muitas vezes projetos aproveitados ou rejeitados devido a considerações de desempenho que se tornam falsas assim que alguém faz algumas medidas na configuração real usada para a aplicação.

Dou algumas diretrizes neste livro, incluindo minimizar o número de chamadas remotas, o qual tem sido um bom conselho relativo a desempenho há bastante tempo. Mesmo assim, você deve verificar cada dica avaliando-o em sua própria aplicação. De modo semelhante, há diversas ocasiões em que os exemplos de código neste livro sacrificam o desempenho para tornar mais fácil a compreensão do que se quer demonstrar. Mais uma vez, é sua tarefa aplicar as otimizações no seu ambiente. No entanto, toda vez que você fizer uma otimização de desempenho, deve medir o ganho obtido, caso contrário pode apenas estar tornando seu código mais difícil de ler.

Há uma conseqüência importante disso: uma alteração significativa na configuração pode invalidar quaisquer fatos referentes ao desempenho. Assim, se você atualizar sua máquina virtual, seu *hardware*, seu banco de dados ou praticamente qualquer outra coisa, você deve refazer suas otimizações de desempenho e assegurar-se de que elas ainda estão ajudando. Em muitos casos, uma nova configuração pode alterar as coisas. Você pode descobrir que uma otimização que fez no passado para melhorar o desempenho, traz prejuízos no novo ambiente.

Um outro problema ao falar sobre desempenho é o fato de que muitos termos são usados de forma inconsistente. A mais famosa vítima é a "escalabilidade," usada regularmente com meia dúzia de significados diferentes. Aqui estão os termos que eu uso.

O **tempo de resposta** é a quantidade de tempo que o sistema leva para processar uma solicitação externa. Pode ser uma ação na interface com o usuário, como o pressionamento de um botão, ou uma chamada de API do servidor.

A **agilidade de resposta** diz respeito ao quão rapidamente o sistema reconhece uma solicitação (em oposição ao tempo que leva para processá-la). Isso é importante em muitos sistemas porque os usuários podem ficar frustrados se um sistema demorar a responder a uma solicitação, ainda que seu tempo de resposta seja bom. Se o seu sistema esperar durante toda a solicitação, então sua agilidade de resposta e seu tempo de resposta são os mesmos. Entretanto, se você indicar que recebeu a solicitação antes de completá-la, então sua agilidade de resposta é melhor. Fornecer uma barra de progresso durante uma cópia de arquivo melhora a agilidade de resposta da sua interface com o usuário, embora não melhore o tempo de resposta.

A **latência** é o tempo mínimo requerido para obter qualquer forma de resposta, mesmo se o trabalho a ser feito for inexistente. Geralmente é a grande questão em sistemas remotos. Se eu pedir a um programa para não fazer nada, mas para me avisar quando tiver terminado de fazer nada, então devo receber uma resposta quase instantânea se o programa rodar no meu *laptop*. Entretanto, se o programa rodar em um computador remoto, pode demorar alguns segundos devido ao tempo gasto para que a solicitação e a resposta cheguem aos seus destinos através da conexão. Como desenvolvedor de aplicações, geralmente, nada posso fazer para melhorar a latência. A latência é também o motivo pelo qual você deve minimizar chamadas remotas.

O ***throughput*** é a quantidade de coisas que você pode fazer em uma dada quantidade de tempo. Se você estiver contabilizando o tempo gasto na cópia de um arquivo, o *throughput* poderia ser medido em *bytes* por segundo. Para aplicações corporativas, uma medida típica é o número de transações por segundo (tps), mas o problema é que isso depende da complexidade da sua transação. Para seu sistema específico, você deve escolher um conjunto usual de transações.

Nesta terminologia, **desempenho** pode significar tanto *throughtput* quanto tempo de resposta - o que for mais importante para você. Às vezes, pode ser difícil falar sobre desempenho quando uma técnica melhora o *throughput*, mas piora o tempo de resposta. Assim, é melhor usar o termo mais preciso. Da perspectiva de um usuário, a agilidade de resposta pode ser mais importante do que o tempo de resposta, então melhorar a agilidade de resposta em detrimento do tempo de resposta ou do *throughput* aumentará o desempenho.

A **carga** é uma medida da pressão a que o sistema está submetido, que poderia ser medida pelo número de usuários a ele conectados em um determinado instante de tempo. A carga é geralmente um contexto para alguma outra medida, como um tempo de resposta. Assim, você pode dizer que o tempo de resposta para alguma solicitação é de 0,5 segundo com 10 usuários e de 2 segundos com 20 usuários.

A **sensibilidade de carga** é uma medida de como o tempo de resposta varia com a carga. Digamos que o sistema A tenha um tempo de resposta de 0,5 segundo para um número de usuários entre 10 e 20, e o sistema B tenha um tempo de resposta de 0,2 segundo para 10 usuários que aumenta para 2 segundos com 20 usuários. Neste caso o sistema A tem uma sensibilidade de carga menor do que o sistema B. Poderíamos também usar o termo **degradação** para dizer que o sistema B degrada mais do que o sistema A.

A **eficiência** é o desempenho dividido pelos recursos. Um sistema que obtenha 30 tps com duas CPUs é mais eficiente que um que obtenha 40 tps com quatro CPUs idênticas.

A **capacidade** de um sistema é uma indicação do máximo *throughput* efetivo ou máxima carga efetiva. Este poderia ser um máximo absoluto ou um ponto a partir do qual o desempenho caia abaixo de um limite aceitável.

A **escalabilidade** é uma medida de como o acréscimo de recursos (normalmente *hardware*) afeta o desempenho. Um sistema escalável é aquele que lhe permite adicionar *hardware* e obter uma melhora de desempenho proporcional, como dobrar o número de servidores disponíveis para dobrar o *throughput*. O **escalabilidade vertical**, ou **escalar para cima**, significa adicionar mais poder a um único servidor (por exemplo, acrescentar memória). O **escalabilidade horizontal**, ou **escalar para fora**, significa adicionar mais servidores.

O problema aqui é que as decisões de projeto não afetam todos esses fatores de desempenho de forma igual. Digamos que temos dois sistemas de *software* rodando em um servidor: a capacidade do Peixe-Espada é de 20 tps, enquanto que a do Camelo é de 40 tps. Qual dos dois tem melhor desempenho? Qual é mais escalável? Não podemos responder a questão da escalabilidade com estes dados, e a única coisa que podemos dizer é que o Camelo é mais eficiente em um único servidor. Se adicionarmos outro servidor, percebemos que o Peixe-Espada agora trata 35 tps, e o Camelo 50 tps. A capacidade do Camelo ainda é melhor, mas parece que o Peixe-Espada pode ter melhor escalabilidade. Se continuarmos adicionando servidores, descobriremos que o Peixe-Espada obtém 15 tps por servidor adicional e que o Camelo obtém 10. Com esses dados, podemos dizer que o Peixe-Espada tem uma melhor escalabilidade horizontal, ainda que o Camelo seja mais eficiente com menos de cinco servidores.

Ao criar aplicações corporativas, muitas vezes faz sentido visar à escalabilidade de *hardware* em vez da capacidade ou mesmo eficiência. A escalabilidade lhe dá a opção de melhor desempenho, se você precisar dela. A escalabilidade pode também ser mais fácil de obter. Os projetistas muitas vezes fazem coisas complicadas que aumentam a capacidade de uma plataforma específica de *hardware* quando, em verdade, poderia ser mais barato comprar mais *hardwares.* Se o Camelo tem um custo maior do que o Peixe-Espada, e esse custo maior é equivalente a um par de servidores, então o Peixe-Espada acaba sendo mais barato, mesmo se você precisar de apenas 40tps. Está na moda reclamar por ter que depender de *hardware* melhor para fazer nosso *software* rodar apropriadamente, e me junto a este coro sempre que tenho que atualizar meu *laptop* apenas para poder rodar a versão mais nova do Word. No entanto, adquirir *hardware* mais novo é com freqüência mais barato do que fazer o software rodar em sistemas menos poderosos. De forma semelhante, adicionar mais servidores é freqüentemente mais barato do que adicionar mais programadores - desde que um sistema seja escalável.

Padrões

Os padrões estão em cena há bastante tempo, então parte de mim não quer ficar repetindo sua história mais uma vez. No entanto, esta é uma oportunidade para dar a minha visão de padrões e o que os torna uma abordagem valiosa para descrever projetos.

Não existe uma definição universalmente aceita de padrão, mas talvez o melhor seja começar por Cristopher Alexander, uma inspiração para muitos entusiastas de padrões: "Cada padrão descreve um problema que ocorre repetidamente no nosso ambiente e então descreve a essência da solução desse problema, de tal forma que você possa usar essa solução um milhão de vezes, sem jamais fazê-lo exatamente da

mesma forma" [Alexander *et al.*]. Alexander é um arquiteto, de modo que ele estava falando de edifícios, mas a definição funciona muito bem para *software* também. O foco do padrão é uma solução específica, uma que seja, ao mesmo tempo, usual e efetiva na abordagem de um ou mais problemas recorrentes. Uma outra maneira de ver é que um padrão é um conjunto de conselhos, e a arte de criar padrões consiste em separar os conselhos em grupos relativamente independentes de modo que você possa se referir a eles e discuti-los mais ou menos independentemente.

Um elemento-chave dos padrões é que eles estão enraizados na prática. Você descobre padrões vendo o que as pessoas fazem, observando as coisas que funcionam e então buscando a "essência da solução." Não é um processo fácil, mas uma vez que você tenha encontrado alguns bons padrões, eles se tornam algo valioso. Para mim, o seu valor reside no fato de me possibilitarem criar um livro que serve como uma referência. Você não precisa ler este livro, ou qualquer outro livro sobre padrões, do início ao fim, para achá-lo útil. Você só precisa ler o suficiente para ter uma noção do que os padrões são, quais problemas eles resolvem e como eles os resolvem. Você não precisa conhecer todos os detalhes, mas apenas o suficiente, de modo que, se você se deparar com um dos problemas, possa encontrar o padrão no livro. Somente então você precisa realmente entender o padrão em profundidade.

Uma vez que você precise do padrão, tem que descobrir como aplicá-lo ao seu contexto. Uma coisa importante sobre padrões é que você não pode simplesmente aplicar cegamente a solução, que é o motivo pelo qual as ferramentas de padrões têm sido um fracasso total. Gosto de dizer que padrões são "meio assados," significando que você tem sempre que terminar de assá-los no forno do seu próprio projeto. Sempre que uso um padrão, adapto-o aqui e ali. Você vê a mesma solução muitas vezes, mas ela nunca é exatamente a mesma.

Cada padrão é relativamente independente, mas padrões não isolados uns dos outros. Freqüentemente, um padrão leva a outro ou um ocorre apenas se outro estiver presente. Assim, você normalmente só verá uma *Herança de Tabela de Classes* (276) se houver um *Modelo de Domínio* (126) no seu projeto. As fronteiras entre os padrões são naturalmente pouco claras, mas tentei tornar cada padrão tão autocontido quanto possível. Se alguém disser "Use uma *Unidade de Trabalho* (187)," você pode dar uma olhada e ver como aplicá-la sem ter que ler o livro inteiro.

Se você for um projetista experiente de aplicações corporativas, provavelmente descobrirá que a maioria destes padrões lhe é familiar. Espero que você não fique muito desapontado (tentei avisá-lo no Prefácio). Os padrões não são idéias originais; eles são, em grande medida, observações das coisas que ocorrem na prática. Em conseqüência, nós, autores de padrões, não dizemos que "inventamos" um padrão, mas, que "descobrimos" um. Nosso papel é perceber a solução usual, procurar sua essência e então escrever o padrão resultante. Para um projetista experiente, o valor do padrão não é lhe dar uma idéia nova; seu valor reside no fato de ele lhe ajudar a comunicar sua idéia. Se você e todos os seus colegas sabem o que é uma *Fachada Remota* (368), você pode dizer muito simplesmente afirmando "Esta classe é uma *Fachada Remota*." Isso também lhe permite dizer a alguém mais novo, "Use um *Objeto de Transferência de Dados* (380) para isso" e eles podem vir a este livro fazer a consulta. O resultado é que os padrões criam um vocabulário sobre projeto, que é o motivo pelo qual a atribuição de nomes é uma questão tão importante.

Embora a maioria destes padrões seja verdadeiramente para aplicações corporativas, aqueles no capítulo de padrões básicos (Capítulo 18) são mais gerais e localizados. Eu os incluo porque me refiro a eles em discussões sobre os padrões de aplicações corporativas.

A Estrutura dos Padrões

Cada autor tem que escolher seu formato de descrição de padrões. Alguns baseiam seus formatos em um livro clássico de padrões como [Alexander *et al.*], [Gang of Four] ou [POSA]. Outros criam seus próprios formatos. Tenho lutado há tempos para descobrir o melhor formato. Por um lado, não quero algo tão pequeno quanto o formato do GOF; por outro, preciso ter seções que dêem suporte a um livro de referência. Assim, isso é o que usei neste livro.

O primeiro item é o nome do padrão. Nomes de padrões são cruciais, porque parte do propósito do padrão é criar um vocabulário que permita aos projetistas se comunicarem mais eficazmente. Assim, se eu lhe disser que meu servidor Web é construído em torno de um *Controlador Frontal* (328) e uma *Vista de Transformação* (343) e você conhecer esses padrões, você terá uma idéia clara da arquitetura do meu servidor Web.

A seguir vêm dois itens que estão sempre juntos: a intenção e o esboço. A intenção resume o padrão em uma frase ou duas. O esboço é uma representação visual do padrão, muitas vezes, mas nem sempre, um diagrama da UML. A idéia é criar um breve lembrete do que se trata o padrão de modo que você possa rapidamente relembrá-lo. Se você já "tem o padrão", significando que você conhece a solução ainda que não saiba o seu nome, então a intenção e o esboço devem ser tudo o que você precisa para saber o que é o padrão.

A próxima seção descreve um problema que motiva o padrão. Este pode não ser o único problema que o padrão resolve, mas acredito que seja o que melhor justifica o padrão.

Como Funciona descreve a solução. Aqui coloco a discussão de questões de implementação e variações com as quais me deparei. A discussão é tão independente quanto possível de qualquer plataforma específica - onde houver seções específicas para alguma plataforma, distanciei-as das margens de modo que você possa pulá-las facilmente. Onde julguei útil, coloquei diagramas da UML para ajudar a explicar os padrões.

Quando Usá-lo descreve quando o padrão deve ser usado. Aqui falo os compromissos que o levam a selecionar uma solução em comparação com outras. Muitos dos padrões deste livro são alternativas, como o *Controlador de Página* (318) e o *Controlador Frontal* (328). Poucos padrões são sempre a escolha certa, então toda vez que encontro um padrão eu sempre me pergunto "Quando eu não o usaria?" Essa questão freqüentemente me leva a padrões alternativos.

A seção *Leitura Adicional* mostra-lhe outras discussões sobre o padrão sendo apresentado. Esta não é uma bibliografia completa. Limitei minhas referências a textos que acho importantes para ajudá-lo a entender o padrão, por isso eliminei qualquer discussão que, segundo meu entendimento, não acrescentasse muito ao que escrevi. Além disso eliminei, é claro, discussões sobre padrões que não li. Também não mencionei itens que acho que serão difíceis de encontrar, ou *links* Web instáveis que, temo, possam já ter desaparecido quando você estiver lendo este livro.

Gosto de acrescentar um ou mais *exemplos*. Cada um é um exemplo *simples* do padrão em uso, ilustrado com algum código em Java ou C#. Escolhi essas linguagens porque elas parecem ser as que o maior número de programadores profissionais sabe ler. É absolutamente essencial compreender que o exemplo não é o padrão. Quando você usar o padrão, ele não se parecerá exatamente com o exemplo, então não o trate como algum tipo de macro completa. Deliberadamente mantive o exemplo tão simples quanto possível, de modo que você possa ver o padrão da forma mais clara

que consegui imaginar. Várias questões que foram ignoradas, se tornarão, no entanto, importantes quando você for usar o padrão, mas elas são específicas do seu próprio ambiente. Eis por que você sempre tem que adaptar o padrão.

Uma das conseqüências disso é que tive de trabalhar duro para manter cada exemplo tão simples quanto pude e, no entanto, ainda transmitindo sua mensagem essencial. Assim, muitas vezes escolhi um exemplo que fosse simples e explícito, em vez de um que demonstrasse como um padrão funciona com os muitos ajustes requeridos em um sistema de produção. É um equilíbrio complicado entre o simples e o simplista, mas também é verdade que muitas questões por demais realistas porém periféricas, podem tornar mais difícil entender o pontos-chave de um padrão.

Foi também por este motivo que usei exemplos simples e independentes em vez de exemplos executáveis conectados. Exemplos independentes são mais fáceis de entender separadamente, mas dão menos orientação sobre como juntá-los. Um exemplo conectado mostra como as coisas se ajustam, mas é difícil entender um padrão sem entender todos os outros envolvidos no exemplo. Embora na teoria seja possível produzir exemplos que sejam conectados e, ainda assim, compreensíveis independentemente, fazer isso é muito difícil – ou pelo menos, muito difícil para mim – então escolhi o caminho independente.

O código nos exemplos é escrito com foco em tornar as idéias compreensíveis. Em conseqüência, muitas coisas ficaram de fora – em particular o tratamento de erros, ao qual não presto muita atenção uma vez que ainda não desenvolvi nenhum padrão nessa área. Eles estão lá meramente para ilustrar o padrão. Eles não se propõem a mostrar como modelar qualquer problema específico de negócio.

Por esses motivos, o código não está disponível para *download* no meu *site* Web. Cada exemplo de código neste livro é cercado com tantos andaimes para simplificar as idéias básicas, que eles não têm valor algum em um ambiente de produção.

Nem todas as seções aparecem em todos os padrões. Para o que não consegui pensar em um bom exemplo ou texto de motivação, deixei-os de fora.

Limitações Destes Padrões

Como mencionei no Prefácio, esta coleção de padrões não é, de forma alguma, um guia completo para o desenvolvimento de aplicações corporativas. Meu teste para este livro não é se ele é completo, mas simplesmente se é útil. O campo é muito extenso para uma única mente, que dirá para um único livro.

Os padrões aqui são todos padrões com os quais me deparei na prática, mas não vou dizer que entendo completamente todas as suas ramificações e inter-relacionamentos. Este livro reflete minha compreensão atual, e essa compreensão se desenvolveu à medida que eu escrevia o livro. Espero que ela continue a se desenvolver muito após este livro ter sido impresso. Uma certeza sobre o desenvolvimento de *software* e suas técnicas é que ele nunca estaciona.

Quando você considerar a utilização dos padrões, nunca se esqueça de que são um ponto de partida, não um destino final. Não existe como um autor descrever todas as variações que os projetos de *software* podem ter. Escrevi estes padrões para ajudar a fornecer um início, de modo que você possa ler sobre as lições que eu e as pessoas que observei aprendemos fazendo e lutando. Além dessas, você terá suas próprias lutas. Lembre-se sempre de que cada padrão está incompleto e que você tem a responsabilidade, e a diversão, de completá-lo no contexto do seu próprio sistema.

Esta página foi deixada em branco intencionalmente.

PARTE I

AS NARRATIVAS

Esta página foi deixada em branco intencionalmente.

CAPÍTULO 1

Criando Camadas

A criação de camadas é uma das técnicas mais comuns que os projetistas de *software* usam para quebrar em pedaços um sistema complexo de *software*. Você encontra esta técnica em arquiteturas de máquinas, em que as camadas descendem de uma linguagem de programação com chamadas do sistema operacional aos *drivers* de dispositivos e conjuntos de instruções da CPU, e às portas lógicas dentro de *chips*. Redes de computadores têm FTP como uma camada sobre TCP, o qual, por sua vez, está sobre o IP, que está sobre a ethernet.

Ao pensar em um sistema em termos de camadas, você imagina os subsistemas principais no *software* dispostos de forma parecida com camadas de um bolo, em que cada camada repousa sobre uma camada mais baixa. Nesse esquema, a camada mais alta usa vários serviços definidos pela camada mais baixa, mas a camada mais baixa ignora a existência da camada mais alta. Além disso, cada camada normalmente esconde suas camadas mais baixas das camadas acima, então a camada 4 usa os serviços da camada 3, a qual usa os serviços da camada 2, mas a camada 4 ignora a existência da camada 2. (Nem todas as arquiteturas de camadas são opacas como essa, mas a maioria é.)

Dividir um sistema em camadas tem uma série de benefícios importantes:

- Você pode compreender uma única camada como um todo coerente sem saber muito sobre as outras camadas. Você pode compreender como construir um serviço FTP sobre TCP sem conhecer os detalhes de como funciona o protocolo ethernet.
- Você pode substituir camadas por implementações alternativas dos mesmos serviços básicos. Um serviço FTP pode rodar sem modificações sobre ethernet, PPP ou seja lá o que o for usado pelo provedor.
- Você minimiza as dependências entre as camadas. Se o provedor alterar o sistema físico de transmissão, não precisamos alterar nosso serviço FTP.

- Camadas são bons lugares para padronização. TCP e IP são padrões porque eles definem como suas camadas devem operar.
- Uma vez que você tenha construído uma camada, ela pode ser usada por muitos serviços de nível mais alto. Desta forma, TCP/IP é usado por FTP, telnet, SSH e HTTP. De outra maneira, todos esses protocolos de nível mais alto teriam que escrever seus próprios protocolos de nível mais baixo.

O uso de camadas é uma técnica importante, mas há aspectos negativos:

- As camadas encapsulam bem algumas coisas, mas não todas. Isso, às vezes, resulta em alterações em cascata. O exemplo clássico disto em uma aplicação corporativa em camadas é o acréscimo de um campo que precise ser mostrado na interface com o usuário e deva estar no banco de dados e assim deva também ser acrescentado a cada camada entre elas.
- Camadas extras podem prejudicar o desempenho. Em cada camada os dados precisam, tipicamente, ser transformados de uma representação para outra. O encapsulamento de uma função subjacente, no entanto, muitas vezes lhe dá ganhos de eficiência que mais do que compensam esse problema. Uma camada que controla transações pode ser otimizada e então tornará tudo mais rápido.

Contudo, a parte mais difícil de uma arquitetura em camadas é decidir quais camadas são necessárias e quais as responsabilidades que cada uma deve receber.

A Evolução das Camadas nas Aplicações Corporativas

Embora eu seja jovem demais para ter feito algum trabalho nos velhos tempos dos sistemas em *batch*, não sinto que as pessoas pensassem muito em camadas naquele tempo. Você escrevia um programa que manipulava algum tipo de arquivo (ISAM, VSAM, etc.) e essa era sua aplicação. Não era necessário aplicar nenhuma camada.

A noção de camadas se tornou mais visível nos anos 1990, com o advento dos sistemas **cliente-servidores**. Estes eram sistemas em duas camadas: o cliente mantinha a interface com o usuário e um ou outro código da aplicação, e o servidor era normalmente um banco de dados relacional. Ferramentas comuns para o lado cliente eram, por exemplo, VB, Powerbuilder e Delphi. Essas ferramentas tornaram particularmente fácil criar aplicações que faziam uso intensivo de dados, uma vez que elas disponibilizavam componentes visuais que trabalhavam com SQL. Assim, você podia criar uma tela arrastando controles para uma área de desenho e então usando páginas de propriedades para conectar os controles ao banco de dados.

Se a aplicação tivesse somente de exibir e fazer atualizações simples em dados relacionais, então esses sistemas cliente-servidor funcionavam muito bem. O problema surgiu com a lógica de domínio: regras de negócio, validações, cálculos, e assim por diante. Normalmente, as pessoas escreviam essa lógica no cliente, mas isso era desajeitado e, normalmente, feito embutindo-se a lógica diretamente nas telas da interface com o usuário. À medida que a lógica do domínio se tornava mais complexa, ficava muito difícil trabalhar com este código. Além disso, embutir lógica nas telas

facilitava a duplicação de código, o que significava que alterações simples resultavam em buscas de código semelhante em muitas telas.

Uma alternativa era colocar a lógica de domínio no banco de dados na forma de procedimentos armazenados (*stored procedures*). Estes, no entanto, forneciam mecanismos limitados de estruturação, o que mais uma vez levava a código desajeitado. Além disso, muitas pessoas gostavam de bancos de dados relacionais porque SQL era um padrão, o que lhes permitia a qualquer tempo mudar o fornecedor do banco de dados. Ainda que poucas pessoas realmente fizessem isso, muitas gostavam de ter a opção de mudar de fornecedor sem que isso implicasse em custos de migração altos demais. Por serem todos proprietários, os procedimentos armazenados eliminavam essa opção.

Ao mesmo tempo em que a arquitetura cliente-servidor estava ganhando popularidade, o mundo orientado a objetos estava ascendendo. A comunidade de objetos tinha a resposta para o problema da lógica de domínio: migrar para um sistema em três camadas. Nesta abordagem, você tinha uma camada de apresentação para a sua interface com o usuário, uma camada de domínio para a sua lógica de domínio e uma camada de dados. Desta maneira, você poderia tirar toda a complexa lógica de domínio da interface com o usuário e colocá-la em uma camada na qual você poderia estruturá-la apropriadamente utilizando objetos.

Apesar disso, a popularidade dos objetos fez pouco progresso. A verdade era que muitos sistemas eram simples, ou pelo menos começavam simples. Embora a abordagem em três camadas tivesse muitos benefícios, o ferramental para o desenvolvimento cliente-servidor era convincente se o seu problema fosse simples. Além disso, as ferramentas para o desenvolvimento cliente-servidor eram difíceis, ou mesmo impossíveis, de usar em uma configuração com três camadas.

Acho que o abalo sísmico aqui foi o advento da Web. De repente as pessoas passaram a querer instalar aplicações cliente-servidor usando um navegador Web. No entanto, se toda a sua lógica de negócio estivesse enterrada em um cliente rico, então toda ela precisaria ser refeita para ter uma interface Web. Um sistema bem-projetado, em três camadas, poderia simplesmente acrescentar uma nova camada de apresentação e estaria pronto. Além disso, com a linguagem Java, vimos uma linguagem orientada a objetos ocupar a cena sem pudores. As ferramentas que surgiram para criar páginas Web eram muito menos amarradas à linguagem SQL e, assim, mais adaptáveis a uma terceira camada.

Quando as pessoas discutem a criação de camadas, freqüentemente há confusão entre os termos *layer* e *tier*. Muitas vezes os dois são usados como sinônimos, mas a maioria das pessoas considera que o termo *tier* implica uma separação física. Os sistemas cliente-servidor são freqüentemente descritos como sistemas em duas camadas (*two-tier systems*), e a separação é física: o cliente é um *desktop* e o servidor é um servidor. Uso *layer* para enfatizar o fato de que você não tem que rodar as camadas em máquinas diferentes. Uma camada distinta de lógica de domínio muitas vezes pode rodar tanto em um *desktop* como no servidor de banco de dados. Nessa situação você tem dois nodos, mas três camadas distintas. Com um banco de dados local, posso executar todas as três camadas em um único *laptop*, mas ainda assim haverá três camadas distintas.

As Três Camadas Principais

Neste livro estou centralizando minha discussão em torno da arquitetura contendo três camadas principais: apresentação, domínio e fonte de dados. (Estou seguindo os nomes usados em [Brown *et al.*]). A Tabela 1.1 resume estas camadas.

A lógica de **apresentação** diz respeito a como tratar a interação entre o usuário e o *software*. Isso pode ser tão simples quanto uma linha de comando ou um menu baseado em texto, porém, hoje é mais provável que seja uma interface gráfica em um cliente rico ou um navegador com interface baseada em HTML. (Neste livro, uso **cliente rico** significando uma interface com o usuário ao estilo Windows ou Swing, em vez de um navegador HTML.) As responsabilidades primárias da camada de apresentação são exibir informações para o usuário e traduzir comandos do usuário em ações sobre o domínio e a camada de dados.

A lógica da **camada de dados** diz respeito à comunicação com outros sistemas que executam tarefas no interesse da aplicação. Estes podem ser monitores de transações, outras aplicações, sistemas de mensagens e assim por diante. Para a maioria das aplicações corporativas, a maior parte da lógica de dados é um banco de dados responsável, antes de mais nada, pelo armazenamento de dados persistentes.

O resto é a **lógica de domínio**, também chamada de lógica de negócio. Este é o trabalho que esta aplicação tem de fazer para o domínio com o qual você está trabalhando. Envolve cálculos baseados nas entradas e em dados armazenados, validação de quaisquer dados provenientes da camada de apresentação e a compreensão exata de qual lógica de dados executar, dependendo dos comandos recebidos da apresentação.

Às vezes as camadas são organizadas de modo que a camada de domínio esconda completamente a camada de dados da camada de apresentação. Com maior freqüência, contudo, a apresentação acessa diretamente o armazenamento de dados. Embora isso seja menos puro, tende a funcionar melhor na prática. A apresentação pode interpretar um comando do usuário, usar a camada de dados para trazer os dados relevantes do banco de dados e então deixar a lógica de domínio manipular esses dados antes de apresentá-los na tela.

Uma única aplicação freqüentemente pode ter múltiplos pacotes de cada uma dessas três áreas. Uma aplicação, projetada para ser manipulada não apenas por usuários finais por meio de uma interface de cliente rico, mas também por meio de uma linha de comando teria duas apresentações: uma para a interface de cliente rico e uma para a linha de comando. Múltiplos componentes de dados podem estar presentes para bancos de dados diferentes, mas, supostamente, estariam presentes par-

Tabela 1.1 Três Camadas Principais

Camada	Responsabilidades
Apresentação	Fornecimento de serviços, exibição de informações (p. ex., em Windows ou HTML, tratamento de solicitações do usuário (cliques com o *mouse*, pressionamento de teclas), requisições HTTP, chamadas em linhas de comando, API em lotes)
Domínio	Lógica que é o real propósito do sistema
Fonte de Dados	Comunicação com os bancos de dados, sistemas de mensagens, gerenciadores de transações, outros pacotes

ticularmente para a comunicação com os pacotes existentes. Mesmo o domínio pode ser quebrado em áreas distintas relativamente separadas umas das outras. Certos pacotes de dados só podem ser usados por determinados pacotes do domínio.

Até agora falei sobre um único usuário. Isso naturalmente levanta a questão do que acontece quando não há um ser humano guiando o *software*. Pode ser algo novo e moderno como um serviço Web ou algo banal e útil como um processo em lote. Neste último caso, o usuário é o programa cliente. Neste ponto torna-se evidente que há muita semelhança entre as camadas de apresentação e de dados visto que ambas são relacionadas à conexão com o mundo externo. Esta é a lógica por trás do padrão *Arquitetura Hexagonal* [wiki] de Alistair Cockburn, que enxerga qualquer sistema como um núcleo cercado de interfaces para sistemas externos. Na *Arquitetura Hexagonal*, tudo que é externo ao sistema é tratado essencialmente como uma interface externa. Consiste, portanto, em uma visão simétrica ao invés do meu esquema assimétrico de camadas.

No entanto, considero essa assimetria útil, porque há uma boa distinção a ser feita entre uma interface que você fornece como um serviço para outros e o serviço que você utiliza de outra pessoa qualquer. No fundo, esta é a distinção real que faço entre a apresentação e a camada de dados. A apresentação é uma interface externa para um serviço que o seu sistema oferece a outra pessoa, seja um humano complexo ou um programa remoto simples. A camada de dados é a interface para coisas que estão provendo um serviço para você. É benéfico pensar nelas de forma diferente, porque a diferença nos clientes altera a maneira de pensar sobre o serviço.

Embora possamos identificar as três camadas usuais, apresentação, domínio e fonte de dados, em cada aplicação corporativa, a maneira como você as separa depende do quão complexa é a aplicação. Um roteiro simples para extrair dados de um banco de dados e exibi-los em uma página Web pode, todo ele, consistir de um único procedimento. Eu ainda me esforçaria para separar as três camadas, mas, neste caso, poderia fazê-lo simplesmente colocando o comportamento de cada camada em sub-rotinas separadas. Se o sistema se tornasse mais complexo, quebraria as três camadas em classes separadas. Se a complexidade aumentasse ainda mais, dividiria as classes em pacotes separados. Meu conselho geral é escolher a forma mais apropriada de separação para o seu problema mas estar seguro de que você tenha algum tipo de separação – ao menos ao nível de sub-rotinas.

Junto com a separação, há também uma regra estabelecida sobre dependências: o domínio e a camada de dados nunca devem ser dependentes da apresentação, isto é, não deve haver chamadas de sub-rotinas da apresentação a partir do código do domínio ou da camada de dados. Essa regra torna mais fácil utilizar apresentações diferentes sobre a mesma base e torna mais fácil modificar a apresentação sem ramificações sérias mais abaixo. O relacionamento entre o domínio e a camada de dados é mais complexo e depende dos padrões arquiteturais usados para a fonte de dados.

Uma das partes mais difíceis de trabalhar com lógica de domínio parece ser que as pessoas freqüentemente acham difícil reconhecer o que é lógica de domínio e o que são outras formas de lógica. Um teste informal que gosto de aplicar é imaginar a adição de uma camada radicalmente diferente a uma aplicação, como uma interface em linha de comando a uma aplicação Web. Se, para fazer isso, você tiver que duplicar alguma funcionalidade, é um sinal de que a lógica de domínio vazou para a apresentação. De maneira semelhante, você tem que duplicar lógica para substituir um banco de dados relacional por um arquivo XML?

Falaram-me certa vez sobre um sistema que era um bom exemplo disso: ele continha uma lista de produtos em que todos que tivessem um acréscimo de vendas de 10% ou mais, em relação ao mês anterior, eram coloridos de vermelho. Para fazer isso, os desenvolvedores colocaram lógica na camada de apresentação que comparava as vendas do mês atual com as do mês anterior e, se a diferença fosse maior do que 10%, eles a coloriam de vermelho.

O problema é que essa solução coloca lógica de domínio na apresentação. Para separar corretamente as camadas, você precisa de um método na camada de domínio para indicar se um produto aumentou suas vendas. Este método faz a comparação entre os dois meses e retorna um valor booleano. A camada de apresentação então, simplesmente chama este método booleano e, se o valor retornado é verdadeiro, destaca o produto em vermelho. Desta forma o processo é dividido em suas duas partes: decidir se há algo a ser destacado e decidir como destacá-lo.

Preocupo-me a estar sendo demasiadamente categórico a esse respeito. Ao rever este livro, Alan Knight comentou que ele estava "em dúvida se colocar lógica do negócio na interface com o usuário é o primeiro passo em direção a uma rampa escorregadia para o inferno ou uma coisa perfeitamente razoável a fazer a que apenas um purista categórico faria objeção". A razão pela qual nos sentimos desconfortáveis é porque ambos são verdadeiros!

Escolhendo Onde Rodar suas Camadas

Na maior parte deste livro estarei falando sobre camadas lógicas – isto é, a divisão de um sistema em pedaços separados para reduzir o acoplamento entre as diferentes partes do sistema. A separação entre camadas é útil mesmo se todas as camadas estiverem rodando na mesma máquina física. Entretanto, há lugares onde a estrutura física de um sistema faz diferença.

Na maioria das aplicações de sistemas de informação, a decisão diz respeito a rodar o processamento em um cliente, em uma máquina *desktop*, ou em um servidor.

Muitas vezes, a solução mais simples é rodar tudo em servidores. Um *front end** HTML usando um navegador Web é uma boa maneira de fazer isso. A grande vantagem de rodar no servidor é que, por estar em uma quantidade limitada de lugares, tudo é fácil de atualizar e reparar. Você não tem que se preocupar com a distribuição para muitos *desktops* e em mantê-los todos em sincronia com o servidor. Você também não tem que se preocupar com compatibilidades com outros *softwares* de *desktop*.

O argumento geral a favor da execução em um cliente gira em torno da capacidade de resposta ou operação desconectada. Qualquer lógica que rode no servidor precisa de uma viagem de ida e volta ao servidor para responder a qualquer coisa que o usuário faça. Se esse usuário quiser brincar com alguma coisa e receber retorno imediato, essa ida e volta atrapalha. Ela também precisa de uma conexão de rede para rodar. A rede pode estar por toda parte, mas, enquanto digito isso, ela não está a 10.000 metros. Em breve, poderá estar em todo lugar, mas há pessoas que querem trabalhar agora, sem ter de esperar que a cobertura sem fios chegue a lugares muito distantes. As operações desconectadas trazem desafios específicos, e lamento tê-los colocado fora do escopo deste livro.

* N. de R. T.: Parte do programa responsável pela interface com o usuário.

Com essas forças gerais posicionadas, podemos considerar as opções, camada por camada. A camada de dados quase sempre roda em servidores. A exceção são as situações em que você poderia duplicar a funcionalidade do servidor em um cliente convenientemente poderoso, geralmente quando você quer uma operação desconectada. Neste caso, as alterações na fonte de dados do cliente desconectado precisam ser sincronizadas com o servidor. Como mencionei anteriormente, decidi deixar essas questões para outro dia – ou outro autor.

A decisão sobre onde rodar a apresentação depende, em sua maior parte, do tipo de interface de usuário que você quer. Um cliente rico implica, quase certamente, em rodar a apresentação no cliente. Uma interface Web implica, quase certamente, em rodá-la no servidor. Existem exceções – por exemplo, a operação remota de um *software* cliente (tais como servidores X no mundo Unix) rodando um servidor Web no *desktop* – mas estas exceções são raras.

Se você estiver criando um sistema B2C, você não tem escolha. Qualquer usuário, Tom, Dick ou Harriet, pode estar se conectando aos seus servidores, e você não quer mandar ninguém embora porque eles insistem em fazer suas compras *online* com um TRS-80. Neste caso, você faz todo o processamento no servidor e devolve HTML para o *browser* manipular. A limitação da opção HTML é que cada pedaço de uma tomada de decisão precisa de uma viagem de ida e volta do cliente para o servidor, e isso pode diminuir a capacidade de resposta. Você pode reduzir parte do atraso com *scripts* no navegador e *applets*, mas eles reduzem a compatibilidade do seu navegador e tendem a acrescentar outras dores de cabeça. Quanto mais puro for seu HTML, mas fácil será sua vida.

Esta vida fácil é atraente mesmo se cada uma das suas áreas de trabalho for construída à mão, com todo o carinho, pelo seu departamento de sistemas de informação. Manter clientes atualizados e evitar erros de compatibilidade com outros *softwares* são problemas que mesmo os mais simples sistemas baseados em clientes ricos têm.

A razão principal pela qual as pessoas desejam uma apresentação baseada em clientes ricos é que algumas tarefas são complicadas para os usuários realizarem e, nesses casos, para ter uma aplicação amigável, eles precisarão de mais do que uma interface Web pode dar. Cada vez mais, entretanto, as pessoas estão se acostumando às técnicas para construir *front ends* Web mais amigáveis, e isso reduz a necessidade de uma apresentação em um cliente rico. Hoje, enquanto escrevo este livro, sou muito favorável à apresentação Web, se você puder; e ao cliente rico, se você precisar.

Isso nos deixa com a lógica do domínio. Você pode rodar toda a lógica de negócio no servidor, toda ela no cliente, ou você pode dividi-la. Uma vez mais, toda a lógica no servidor é a melhor escolha para facilitar a manutenção. A motivação em movê-la para o cliente visa a melhorar a capacidade de resposta ou possibilitar o uso desconectado.

Se você tiver que rodar alguma lógica no cliente, deve considerar a possibilidade de executá-la toda lá – pelo menos assim ela fica toda em um único lugar. Normalmente esse cenário ocorre de mãos dadas com um cliente rico – rodar um servidor Web em uma máquina cliente não irá ajudar muito a capacidade de resposta, embora possa ser um modo de tratar operações desconectadas. Neste caso, você ainda pode manter sua lógica do domínio em módulos separados da apresentação, com um *Roteiro de Transação* (110) ou um *Modelo do Domínio* (116). O problema em colocar toda a lógica do domínio no cliente é que você tem mais coisas para atualizar e manter.

Dividir a lógica de domínio entre o *desktop* e o servidor parece ser o pior de dois mundos, porque você não sabe onde cada parte da lógica vai estar. O motivo principal para fazer isso é se apenas uma pequena parte da lógica precisar rodar no cliente. O truque então é isolar esta parte da lógica em um módulo que não seja dependente de nenhuma outra parte do sistema. Dessa forma, você pode rodar esse módulo no cliente ou no servidor. Isso irá requerer uma boa quantidade de código malicioso, mas é um bom modo de realizar esse trabalho.

Uma vez que você tenha escolhido seus nós de processamento, você deve tentar manter todo o código em um único processo, seja em um único nó ou copiado em diversos nós em um *cluster*. Não tente separar as camadas em processos discretos a menos que seja absolutamente necessário. Fazer isso irá degradar o desempenho e acrescentar complexidade ao sistema, uma vez que você terá que acrescentar coisas como *Fachadas Remotas* (368) e *Objetos de Transferência de Dados* (380).

É importante lembrar que muitas dessas coisas são o que Jens Coldewey se refere como **impulsionadores de complexidade** – distribuição, *multithreading* explícito, mudanças de paradigma (como objeto/relacional), desenvolvimento multiplataforma e requisitos extremos de desempenho (como mais de 100 transações por segundo). Tudo isso carrega um alto custo. Certamente há ocasiões em que você tem que fazê-lo, mas nunca se esqueça de que cada um deles carrega um custo tanto no desenvolvimento quanto na manutenção.

CAPÍTULO 2

Organizando a Lógica do Domínio

De modo a organizar a lógica do domínio, eu a separei em três padrões principais: *Roteiro de Transação* (120), *Modelo do Domínio* (126) e *Módulo Tabela* (134).

A abordagem mais simples para armazenar lógica de domínio é o *Roteiro de Transação* (120). Um *Roteiro de Transação* (120) é, essencialmente, um procedimento que recebe os dados de entrada da camada de apresentação, efetua o processamento com validações e cálculos, armazena dados no banco de dados e invoca quaisquer operações necessárias em outros sistemas. Ele então responde com mais dados para a apresentação, talvez executando mais cálculos para ajudar a organizar e formatar a resposta. A organização fundamental é a de um único procedimento para cada ação que o usuário possa querer executar. Assim, podemos pensar neste padrão como sendo um roteiro para uma ação ou transação de negócio. Ele não tem de ser constituído, necessariamente, de um único procedimento. As partes podem ser separadas em sub-rotinas, e estas podem ser compartilhadas por diferentes *Roteiros de Transação* (120). Entretanto, a motivação ainda é a de um procedimento para cada ação, de modo que um sistema de varejo pode ter *Roteiros de Transação* (120) para a checagem de saída, para acrescentar algo ao carrinho de compras, para mostrar a situação atual da entrega e assim por diante.

Um *Roteiro de Transação* (120) oferece diversas vantagens:

- É um modelo procedural simples que a maioria dos desenvolvedores compreende.
- Funciona bem com uma camada de dados simples usando o *Gateway de Linha de Dados* (158) ou o *Gateway de Tabelas de Dados* (151).
- O modo de estabelecer as fronteiras da transação é uma tarefa óbvia: comece abrindo uma transação e termine fechando-a. É fácil para as ferramentas fazer isso por trás dos panos.

Infelizmente, existem também muitas desvantagens, que tendem a aparecer à medida que a complexidade da lógica do domínio aumenta. Freqüentemente, haverá código duplicado, uma vez que diversas transações precisam fazer coisas parecidas. Este problema pode, em parte, ser tratado fatorando-se o código comum em sub-rotinas, mas, mesmo assim, muito da duplicação é complicada de ser removida e mais difícil ainda de ser localizada. A aplicação resultante pode acabar sendo uma teia confusa de sub-rotinas sem uma estrutura clara.

É claro que a lógica complexa é o cenário ideal para os objetos, e a maneira pela qual a orientação a objetos lida com este problema é por meio de um *Modelo de Domínio* (126). Com um *Modelo de Domínio* (126), construímos um modelo de nosso domínio o qual, pelo menos em uma primeira aproximação, é organizado primariamente em torno dos nomes do domínio. Assim, um sistema de *leasing* teria classes para o arrendamento, propriedades, e assim por diante. A lógica para lidar com validações e cálculos seria colocada neste modelo de domínio, de modo que um objeto remessa poderia conter a lógica para calcular o preço da remessa para uma entrega. Poderia ainda haver rotinas para calcular uma fatura, mas tal procedimento logo delegaria para um método do *Modelo de Domínio* (126).

Usar um *Modelo de Domínio* (126), em oposição a um *Roteiro de Transação* (120), é a essência da mudança de paradigma que as pessoas de orientação a objetos falam tanto. Em vez de uma rotina contendo toda a lógica para uma ação do usuário, cada objeto realiza uma parte da lógica que seja relevante para ele. Se você não estiver acostumado com um *Modelo de Domínio* (126), aprender a trabalhar com um pode ser muito frustrante quando você começa a correr de objeto em objeto tentando descobrir em qual está o comportamento.

É difícil capturar a essência da diferença entre os dois padrões com um exemplo simples, mas nas discussões sobre os padrões tentei fazer isso criando um pedaço simples de lógica de domínio de ambas as maneiras. A maneira mais fácil de ver a diferença é olhar os diagramas de seqüência das duas abordagens (Figuras 2.1 e 2.2). O problema essencial é que diferentes tipos de produtos têm algoritmos diferentes para o lançamento de receitas em um dado contrato (veja o Capítulo 9, para uma melhor fundamentação). O método de cálculo tem que determinar a que tipo de produ-

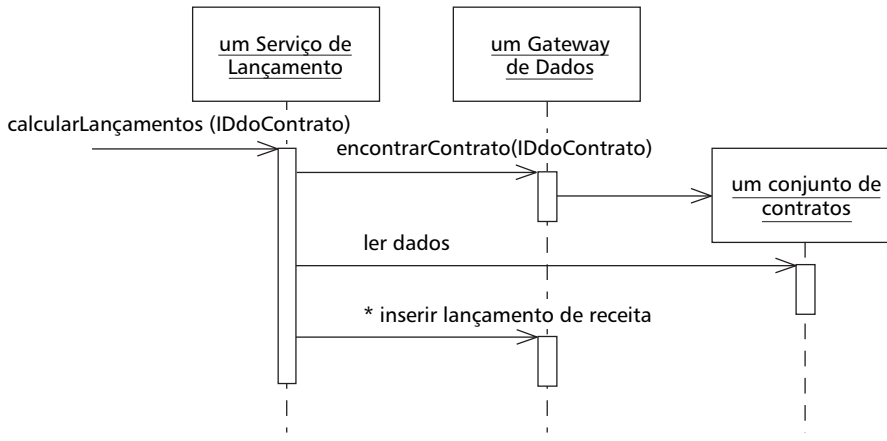


Figura 2.1 Um *Roteiro de Transação* (120) para o lançamento de receitas.

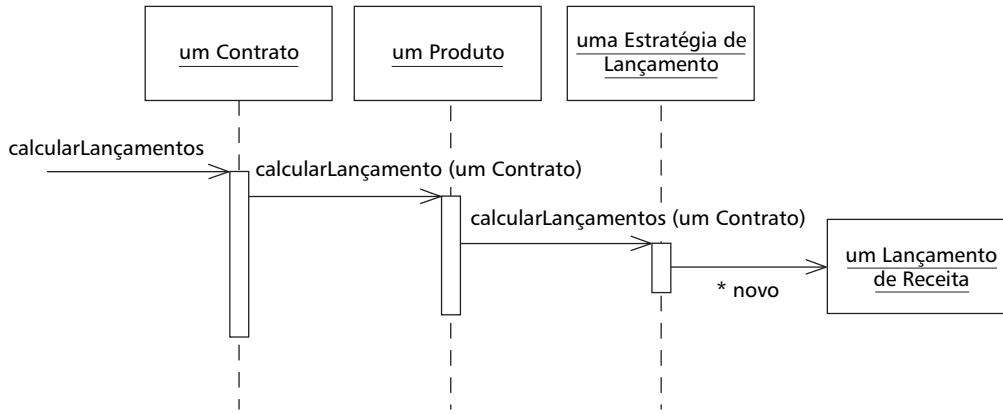


Figura 2.2 Um *Modelo de Domínio* (126) para o lançamento de receitas.

to um dado contrato se refere, aplicar o algoritmo correto e então criar os objetos lançamento de receita para capturar os resultados do cálculo. (Para simplificar estou ignorando as questões relativas à interação com o banco de dados).

Na Figura 2.1, o método no *Roteiro de Transação* (120) faz todo o trabalho. Os objetos subjacentes são apenas *Gateways de Tabela de Dados* (151) e tudo o que eles fazem é passar dados para o roteiro de transação.

Ao contrário, a Figura 2.2 mostra múltiplos objetos, cada um passando parte do comportamento para outro até que um objeto de estratégia crie os resultados.

O valor de um *Modelo de Domínio* (126) é que, depois que você se acostuma, há muitas técnicas que lhe permitem lidar com lógica cada vez mais complexa de uma forma bem-organizada. À medida que mais e mais algoritmos para calcular o lançamento de receitas vão sendo adicionados, podemos acrescentá-los adicionando novos objetos contendo a estratégia de lançamento. Com o *Roteiro de Transação* (120), teremos de adicionar mais condições à lógica condicional do roteiro. Assim que sua mente estiver tão voltada para objetos como a minha, você descobrirá que prefere um *Modelo de Domínio* (126) mesmo em casos bastante simples.

Os custos de um *Modelo de Domínio* (126) decorrem da complexidade de usá-lo e da complexidade da sua camada de dados. Leva tempo para que pessoas novatas em modelos ricos de objetos se acostumem a um *Modelo de Domínio* (126) rico. Os desenvolvedores freqüentemente precisam gastar diversos meses trabalhando em um projeto que usa este padrão antes que seus paradigmas mudem. No entanto, quando você se acostuma ao *Modelo de Domínio* (126), geralmente está contagiado para o resto da vida e torna-se fácil trabalhar com ele no futuro – é assim que fanáticos por objetos, como eu, são criados. No entanto, uma minoria significativa de desenvolvedores parece não ser capaz de dar o salto.

Mesmo depois de ter dado o salto, você ainda tem que lidar com o mapeamento do banco de dados. Quanto mais rico for o seu *Modelo de Domínio* (126), mais complexo será o mapeamento para um banco de dados relacional (geralmente com um *Mapeador de Dados* (170)). Uma sofisticada camada de dados se aproxima bastante de um custo fixo – obter uma boa camada de dados custa uma quantidade razoável de dinheiro (se você comprar) ou tempo (se você fizer), mas assim que você tiver uma, pode fazer muito com ela.

Há uma terceira escolha para estruturar a lógica do domínio, o *Módulo Tabela* (134). À primeira vista, o *Módulo Tabela* (134) se parece com o *Modelo de Domínio* (126), já que ambos têm classes para contratos, produtos e lançamento de receitas. A diferença vital é que um *Modelo de Domínio* (126) tem uma instância de contrato para cada contrato no banco de dados, enquanto um *Módulo Tabela* (134) tem apenas uma instância. Um *Módulo Tabela* é projetado para trabalhar com um *Conjunto de Registros* (473). Assim, o cliente de um *Módulo Tabela* (134) referente a contratos, irá primeiro fazer consultas ao banco de dados para formar um *Conjunto de Registros* (473), então criará um objeto contrato e passará a ele o *Conjunto de Registros* (473) como parâmetro. O cliente pode então invocar as operações no contrato para fazer diversas coisas (Figura 2.3). Se ele quiser fazer algo com um contrato individual, deve passar o ID desse contrato.

Um *Módulo Tabela* (134) é de muitas maneiras um andar intermediário entre um *Roteiro de Transação* (120) e um *Modelo de Domínio* (126). Organizar a lógica do domínio ao redor de tabelas em vez de procedimentos seqüenciais fornece mais estrutura e torna mais fácil encontrar e remover duplicação. No entanto, você não pode usar várias das técnicas que um *Modelo de Domínio* (126) usa para estruturar a lógica em granularidade mais fina, tais como herança, estratégias e outros padrões OO.

A maior vantagem de um *Módulo Tabela* (134) é como ele se adapta ao resto da arquitetura. Muitos ambientes usando uma interface gráfica com o usuário (GUI) são criados para trabalhar sobre os resultados de uma consulta SQL organizados em um *Conjunto de Registros* (473). Uma vez que um *Módulo Tabela* (134) também trabalha sobre um *Conjunto de Registros* (473), você pode facilmente executar uma consulta, manipular os resultados no *Módulo Tabela* (134) e passar os dados manipulados para apresentação na interface gráfica. Você também pode usar o *Módulo Tabela* (134) no sentido contrário para realizar mais validações e cálculos. Várias plataformas, particularmente Microsoft COM e .NET, usam esse estilo de desenvolvimento.

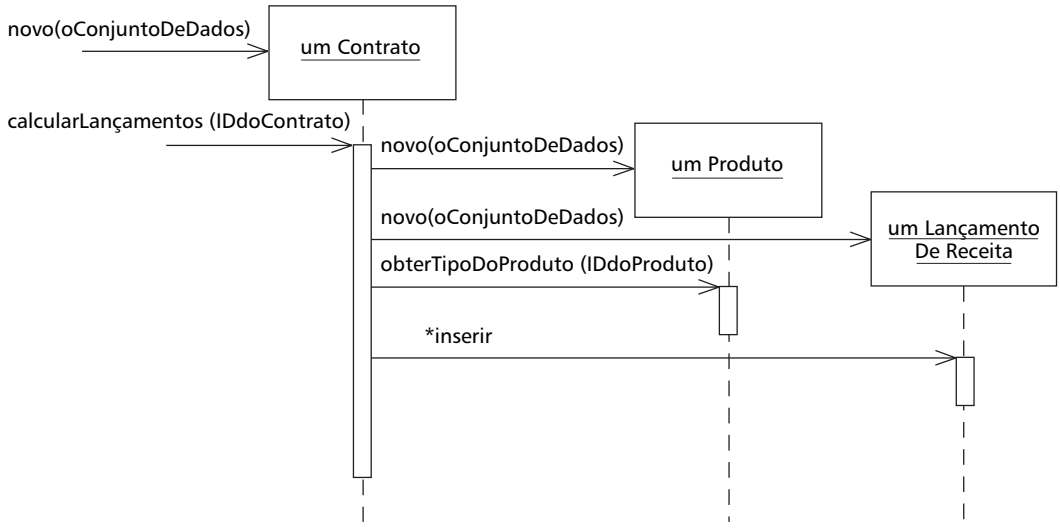


Figura 2.3 Calculando lançamentos de receitas com um *Módulo Tabela* (134).

Fazendo uma Escolha

Então, como você escolhe entre os três padrões? Não é uma escolha fácil e depende muito do quão complexa é sua lógica de domínio. A Figura 2.4 é um daqueles gráficos não-científicos que realmente me irritam em apresentações PowerPoint porque têm eixos sem quantificação alguma. No entanto, o gráfico ajuda a visualizar minha percepção de como comparar os três. O *Modelo de Domínio* (126) é menos atrativo com lógica de domínio simples porque o custo de compreendê-lo e a complexidade da camada de dados adicionam muito esforço ao desenvolvimento, esforço esse que não terá retorno. No entanto, à medida que a complexidade da lógica do domínio aumenta, as outras abordagens tendem a atingir uma barreira em que acrescentar mais características se torna muito mais difícil.

Seu problema, claro, é descobrir onde nesse eixo X sua aplicação se encontra. A boa notícia é que posso dizer que você deveria usar um *Modelo de Domínio* (126) sempre que a complexidade da sua lógica de domínio for maior que 7.42. A má notícia é que ninguém sabe como medir a complexidade da lógica do domínio. Na prática, então, tudo que o você pode fazer é encontrar pessoas experientes que possam fazer uma análise inicial dos requisitos e fazer um julgamento.

Há alguns fatores que alteram um pouco as curvas. Uma equipe familiarizada com o *Modelo de Domínio* (126) diminuirá o custo inicial do uso deste padrão. Isso não irá rebaixá-lo ao mesmo ponto inicial dos outros padrões devido à complexidade da camada de dados. Ainda assim, quanto melhor for a equipe, maior minha inclinação a usar o *Modelo de Domínio* (126).

A atratividade de um *Módulo de Tabela* (134) depende muito do suporte existente em seu ambiente a uma estrutura do tipo *Conjunto de Registros* (473) comum. Se você tiver um ambiente como .NET ou Visual Studio, onde várias ferramentas traba-

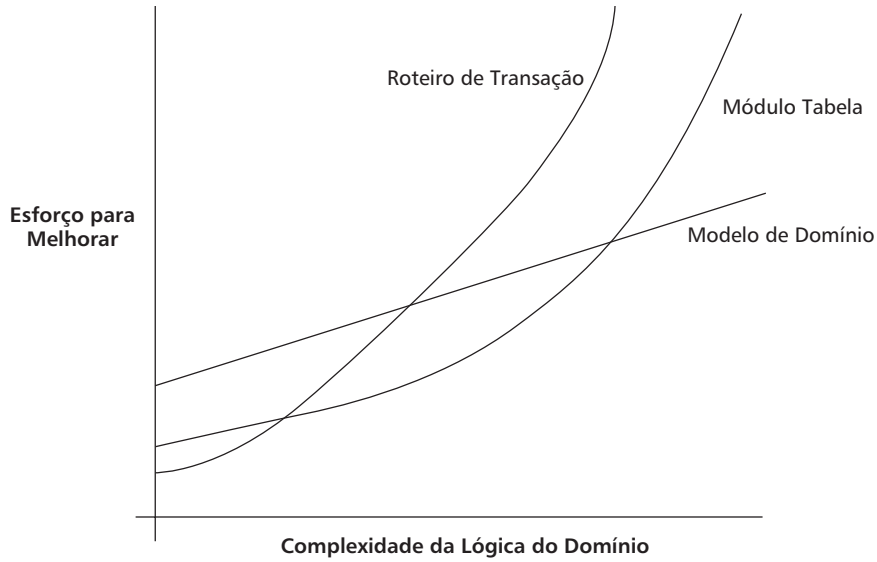


Figura 2.4 Um sentimento sobre o relacionamento entre complexidade e esforço para diferentes estilos de lógica de domínio.

lham ao redor de um *Conjunto de Registros* (473), então isso torna um *Módulo Tabela* (134) muito mais atrativo. De fato, não vejo uma boa razão para usar um *Roteiro de Transação* (120) em um ambiente .NET. Assim, se não houver ferramentas especiais para *Conjuntos de Registros* (473), eu não perderia tempo com o Módulo Tabela (134).

Sua decisão, uma vez que você a tenha tomado, não estará completamente gravada em pedra, mas será mais complicado alterá-la posteriormente. Dessa forma, vale a pena realizar alguma reflexão inicial para decidir qual caminho tomar. Se você descobrir que tomou o caminho errado, então, se você começou com um *Roteiro de Transação* (120), não hesite em refatorar em direção a um *Modelo de Domínio* (126). No entanto, se você começou com um *Modelo de Domínio* (126), geralmente vale menos a pena ir para um *Roteiro de Transação* (120), a não ser que você possa simplificar sua camada de dados.

Estes três padrões não são escolhas mutuamente excludentes. Na verdade, é muito comum usar um *Roteiro de Transação* (120) para uma parte da lógica do domínio e um *Módulo Tabela* (134) ou *Modelo de Domínio* (126) para o resto.

Camada de Serviço

Uma abordagem comum para lidar com a lógica do domínio é dividir a camada de domínio em duas. Uma *Camada de Serviço* (141) é colocada sobre um *Modelo de Domínio* (126) ou *Módulo Tabela* (134) subjacente. Normalmente, você só consegue isso com um *Modelo de Domínio* (126) ou *Módulo Tabela* (134), uma vez que uma camada de domínio que use apenas um *Roteiro de Transação* (120) não é complexa o suficiente para justificar uma camada separada. A lógica da apresentação interage com a de domínio puramente por meio da *Camada de Serviço* (141), a qual age como uma API para a aplicação.

Além de fornecer uma API clara, a *Camada de Serviço* (141) é também um bom local para colocar coisas como controle de transação e segurança. Isso lhe dá um modelo simples para pegar cada método na *Camada de Serviço* (141) e descrever suas características transacionais e de segurança. Uma escolha comum para isso consiste em um arquivo separado de propriedades, mas os atributos .NET fornecem uma maneira elegante de fazer isso diretamente no código.

Quando você tenta uma *Camada de Serviço* (141), uma decisão-chave é quanto comportamento colocar nela. O caso mínimo é tornar a *Camada de Serviço* (141) uma fachada, de modo que todo o comportamento real esteja em objetos subjacentes. Dessa forma, tudo o que a *Camada de Serviço* (141) faz é encaminhar as chamadas à fachada para os objetos de mais baixo nível. Neste caso, a *Camada de Serviços* (141) fornece uma API mais fácil de usar porque, tipicamente, ela é orientada aos casos de uso. Ela também é um ponto conveniente para acrescentar invólucros transacionais e verificações de segurança.

No outro extremo, a maior parte da lógica de negócio é colocada nos *Roteiros de Transação* (120), dentro da *Camada de Serviço* (141). Os objetos de domínio subjacentes são muito simples; se for um *Modelo de Domínio* (126), eles terão uma correspondência um-para-um com o banco de dados e, assim, você pode usar uma camada de dados mais simples, tal como um *Registro Ativo* (165).

A meio caminho entre essas alternativas está uma combinação mais regular de comportamento: o estilo **controle-entidade**. Este nome vem de uma prática comum, muito influenciada por [Jacobson *et al.*]. O principal aqui é colocar a lógica que seja

particular a uma única transação ou caso de uso em *Roteiros de Transação* (120), os quais comumente são referidos como controles ou serviços. Estes são controles diferentes do controle de entrada no *Modelo Vista Controle* (315) ou *Controle de Aplicação* (360) que encontraremos mais adiante, por isso uso o termo **controle do caso de uso.** O comportamento usado em mais de um caso de uso vai para os objetos de domínio, os quais são chamados de entidades.

Ainda que a abordagem controle-entidade seja comum, nunca gostei muito dela. Os controles de casos de uso, como qualquer *Roteiro de Transação* (120), tendem a encorajar o código duplicado. Minha opinião é que, se você decidir usar um *Modelo de Domínio* (126), realmente deveria ir fundo e fazê-lo dominante. A exceção a isso é se você começou com um projeto que usa o *Roteiro de Transação* (120) com um *Gateway de Linha de Dados* (134). Neste caso, faz sentido mover o comportamento duplicado para os *Gateways de Linhas de Dados* (158), o que os tornará em um *Modelo de Domínio* (126) simples usando *Registro Ativo* (165). No entanto, eu não começaria dessa forma. Eu faria isso apenas para melhorar um projeto que esteja mostrando falhas.

Não estou dizendo que você nunca deva ter objetos de serviço que contenham lógica de domínio, mas sim que você não deve, necessariamente, fazer deles uma camada fixa. Objetos de serviço procedurais podem, às vezes, ser uma maneira muito útil de fatorar lógica, mas tendo a usá-los quando necessário, em vez de usá-los como uma camada da arquitetura.

Minha preferência é, portanto, ter a mais fina *Camada de Serviço* (141) que você puder, se realmente necessitar de uma. Minha abordagem normal é supor que eu não preciso de uma e apenas acrescentá-la se parecer que a aplicação precisa dela. Entretanto, conheço muitos bons projetistas que sempre usam uma *Camada de Serviço* (141) com razoável quantidade de lógica, então sinta-se à vontade para não me dar atenção a esse respeito. Randy Stafford tem tido muito sucesso com uma *Camada de Serviço* (141) rica. Por este motivo pedi a ele que escrevesse o padrão *Camada de Serviço* (141) para este livro.

CAPÍTULO 3

Mapeando para Bancos de Dados Relacionais

O papel da camada de fonte de dados é a comunicação com as diversas partes da infra-estrutura que uma aplicação precisa para executar sua tarefa. Uma parte proeminente deste problema é se comunicar com o banco de dados, o que, na maioria dos sistemas criados hoje, significa um banco de dados relacional. Certamente ainda há muitos dados em formatos de armazenamento mais antigos, como arquivos ISAM e VSAM de *mainframes*, mas a maior parte das pessoas que está construindo sistemas hoje se preocupa em trabalhar com um banco de dados relacional.

Uma das maiores razões para o sucesso dos bancos de dados relacionais é a presença da SQL, a linguagem mais padronizada para comunicação com bancos de dados. Embora a SQL seja cheia de acréscimos aborrecidos e complicados específicos de vendedores, o núcleo da sua sintaxe é comum e bem compreendido.

Padrões de Arquitetura

O primeiro conjunto de padrões abrange os padrões de arquitetura, que direcionam o modo pelo qual a lógica de domínio se comunica com o banco de dados. A escolha que você faz aqui tem amplas conseqüências sobre o seu projeto e, desta forma, dificulta a refatoração, de modo que é uma decisão na qual você deve prestar certa atenção. É também uma escolha fortemente afetada pelo modo como você projeta sua lógica de domínio.

Apesar de toda a difusão do uso da SQL em *software* corporativo, ainda há armadilhas. Muitos desenvolvedores de aplicação não entendem bem SQL e, como resultado, têm problemas para definir consultas e comandos eficazes. Embora existam várias técnicas para embutir SQL em uma linguagem de programação, elas são todas um pouco deselegantes. Seria melhor acessar os dados usando mecanismos que se encaixem com a linguagem de desenvolvimento da aplicação. Administradores de

bancos de dados (DBAs) também gostam de acessar a SQL que acessa uma tabela de modo que possam compreender como melhor ajustá-la e como organizar os índices.

Por essas razões, é sábio separar o acesso SQL da lógica de domínio e colocá-la em classes separadas. Uma boa maneira de organizar essas classes é baseá-las na estrutura da tabela do banco de dados de modo que você tenha uma classe por tabela do banco de dados. Essas classes então formam um *Gateway* (436) para a tabela. O resto da aplicação não precisa saber nada da SQL, e é fácil de encontrar toda a SQL que acessa o banco de dados. Os desenvolvedores que se especializam no banco de dados têm um claro lugar aonde ir.

Há duas maneiras principais por meio das quais você pode usar um *Gateway* (136). A mais óbvia é ter uma instância dela para cada linha retornada por uma consulta (Figura 3.1). Este *Gateway de Linhas de Dados* (158) é uma abordagem que naturalmente se ajusta em um modo orientado a objetos de pensar sobre os dados.

Muitos ambientes fornecem um *Conjunto de Registros* (473) – ou seja, uma estrutura de dados genérica de tabelas e linhas que simula a natureza tabular de um banco de dados. Devido ao fato de um *Conjunto de Registros* (473) ser uma estrutura de dados genérica, os ambientes podem usá-la em muitas partes de uma aplicação. É bastante comum que ferramentas de interfaces gráficas com o usuário (GUI) tenham controles que trabalhem com um *Conjunto de Registros* (473). Se você usar um *Conjunto de Registros* (473), só precisa de um objeto para cada tabela no banco de dados. Este *Gateway de Tabelas de Dados* (151) (veja a Figura 3.2) fornece métodos para pesquisar o banco de dados que retornam um *Conjunto de Registros* (473).

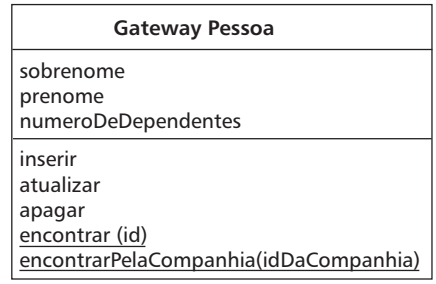


Figura 3.1 Um *Gateway de Linhas de Dados* (158) tem uma instância por linha retornada por uma consulta.

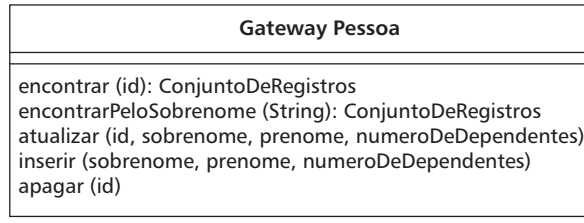


Figura 3.2 Um *Gateway de Tabelas de Dados* (151) tem uma instância por tabela.

Até mesmo em aplicações simples, tendo a usar um dos padrões Gateway. Uma olhada rápida nos meus roteiros Ruby e Python confirmará isso. Considero muito útil a separação clara entre SQL e lógica do domínio.

O fato do *Gateway de Tabelas de Dados* (151) se ajustar tão bem ao *Conjunto de Registros* (473) torna-o a escolha óbvia se você estiver usando *Módulo Tabela* (134). Também é um padrão que você pode usar para pensar em organizar *procedures* armazenadas (*stored procedures*). Muitos projetistas gostam de executar todo seu acesso de banco de dados por meio de procedimentos armazenados em vez de SQL explícita. Neste caso, você pode pensar na coleção de procedimentos armazenados como definindo um *Gateway de Tabelas de Dados* (151) para uma tabela. Eu ainda teria um *Gateway de Tabelas de Dados* (151) em memória para encapsular as chamadas aos procedimentos armazenados, uma vez que isso mantém encapsulados os mecanismos de chamada ao procedimento armazenado.

Se você estiver usando um *Modelo de Domínio* (126), há mais algumas opções. Você certamente pode usar um *Gateway de Linhas de Dados* (158) ou *Gateway de Tabelas de Dados (*151) com um *Modelo de Domínio* (126). Para o meu gosto, entretanto, isso pode ser ou indireção demais, ou não o suficiente.

Em aplicações simples, o *Modelo de Domínio* (126) é uma estrutura sem complicação que em verdade corresponde bastante à estrutura do banco de dados, com uma classe de domínio por tabela do banco de dados. Tais objetos de domínio muitas vezes têm lógica de negócio apenas moderadamente complexa. Nesse caso faz sentido deixar cada objeto do domínio ser responsável por carregar e salvar no banco de dados, o que caracteriza o *Registro Ativo* (165) (veja a Figura 3.3). Outra forma de pensar no *Registro Ativo* (165) é que você começa com um *Gateway de Linhas de Dados* (158) e então acrescenta lógica de domínio à classe, especialmente quando vê código repetido em diversos *Roteiros de Transação* (120).

Neste tipo de situação, a indireção adicionada de um *Gateway* (436) não fornece grande valor. À medida que a lógica de domínio ficar mais complicada e você começar a ir na direção de um *Modelo de Domínio* (126) rico, a abordagem simples de um *Registro Ativo* (165) começa a não funcionar. A associação um-para-um das classes do domínio com as tabelas começa a falhar quando você fatora a lógica de domínio em classes menores. Bancos de dados relacionais não lidam com herança, de modo que se torna difícil usar estratégias [Gang of Four] e outros padrões elegantes OO. À medida que a lógica de domínio fica pior, você irá querer testá-la sem ter que se comunicar com o banco de dados todo o tempo.

Todas essas forças lhe empurram para a indireção à medida que seu *Modelo de Domínio* (126) se torna mais rico. Nesse, caso o *Gateway* (436) pode resolver alguns

Figura 3.3 No *Registro Ativo* (165), um objeto de domínio cliente sabe como interagir com tabelas do banco de dados.

problemas, mas ainda lhe deixa com o *Modelo de Domínio* (126) acoplado ao esquema do banco de dados. Como resultado, há algumas transformações dos campos do *Gateway* (436) para os campos dos objetos do domínio, transformações estas que complicam seus objetos de domínio.

Um caminho melhor é isolar completamente o *Modelo de Domínio* (126) do banco de dados, tornando sua camada de indireção inteiramente responsável pelo mapeamento entre os objetos do domínio e as tabelas do banco de dados. Este *Mapeador de Dados* (170) (veja a Figura 3.4) lida com toda a carga e armazenamento entre o banco de dados e o *Modelo do Domínio* (126) e permite a ambos variar independentemente. É a mais complicada das arquiteturas de mapeamento de banco de dados, mas seu benefício é o completo isolamento das duas camadas.

Não recomendo o uso de um *Gateway* (436) como mecanismo principal de persistência para um *Modelo de Domínio* (126). Se a lógica do domínio for simples e você tiver uma correspondência próxima entre classes e tabelas, o *Registro Ativo* (165) é a solução mais simples; se você tiver algo mais complicado, é do *Mapeador de Dados* (170) que você precisa.

Esses padrões não são mutuamente excludentes. Em boa parte desta discussão estamos pensando no mecanismo principal de persistência, isto é, o modo como você salva os dados de algum tipo de modelo em memória para o banco de dados. Para tal, escolheremos um destes padrões. Você não irá querer misturá-los, porque isso acaba ficando muito confuso. Entretanto, mesmo se estiver usando o *Mapeador de Dados* (170) como mecanismo principal de persistência, você poderá usar um *Gateway* (436) de dados para encapsular tabelas ou serviços que estejam sendo tratados como interfaces externas.

Na minha discussão a respeito dessas idéias, tanto aqui quanto nos próprios padrões, tendo a usar a palavra "tabela". Todavia, a maioria dessas técnicas pode ser aplicada igualmente em visões, consultas encapsuladas por meio de procedimentos armazenados, e consultas dinâmicas freqüentemente usadas. Infelizmente, não há um termo amplamente usado para tabela/visão/consulta/procedimento armazenado, de modo que uso "tabela" porque representa uma estrutura de dados tabular. Geralmente penso em visões como tabelas virtuais, que, é claro, é como também SQL pensa nelas. A mesma sintaxe é usada para pesquisar visões e tabelas.

A atualização é obviamente mais complicada com visões e consultas, já que você nem sempre pode atualizar uma visão diretamente, mas, em vez disso, tem que manipular as tabelas que dão suporte a ela. Nesse caso, encapsular a visão/consulta com um padrão apropriado é uma maneira muito boa de implementar a lógica de atualização em apenas um lugar, o que torna o uso das visões mais simples e confiável.

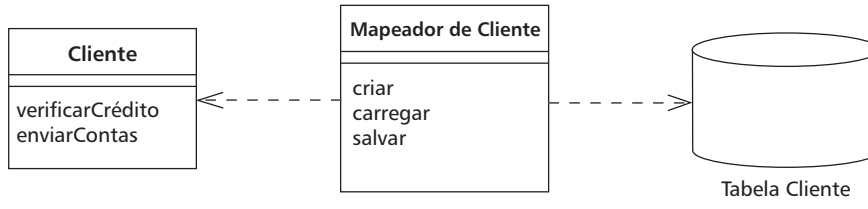


Figura 3.4 Um *Mapeador de Dados* (170) isola os objetos do domínio do banco de dados.

Um dos problemas do uso de visões e consultas dessa forma é que ele pode levar a inconsistências que surpreendam os desenvolvedores que não entendem como uma visão é formada. Eles podem executar atualizações em duas estruturas diferentes, ambas atualizando as mesmas tabelas de suporte em que a segunda atualização sobrescreve uma atualização feita pela primeira. Estabelecendo que a lógica de atualização execute a validação apropriada, você não deveria obter dados inconsistentes, mas pode surpreender seus desenvolvedores.

Também devo mencionar a maneira mais simples de realizar persistência mesmo com o mais complexo *Modelo de Domínio* (126). Nos primórdios dos objetos, muitas pessoas perceberam que havia um "desacordo de impedância" fundamental entre objetos e relações. Assim, seguiu-se um grande esforço em bancos de dados orientados a objetos, os quais essencialmente trouxeram o paradigma OO ao armazenamento em disco. Com um banco de dados OO, você não tem que se preocupar com mapeamento. Você trabalha com uma grande estrutura de objetos interconectados, e o banco de dados descobre quando mover objetos dos discos e para os discos. Além disso, você pode usar transações para agrupar atualizações e permitir compartilhamento dos dados armazenados. Para os programadores isso parece uma quantidade infinita de memória transacional que é suportada transparentemente pelo armazenamento em disco.

A maior vantagem dos bancos de dados OO é que eles aumentam a produtividade. Embora eu não esteja ciente de testes controlados, observações informais estimam o esforço de mapeamento para um banco de dados relacional em cerca de um terço do esforço de programação – um custo que continua durante a manutenção.

No entanto, a maior parte dos projetos não usa bancos de dados OO. A razão principal contra eles é o risco. Bancos de dados relacionais são uma tecnologia bastante conhecida e provada, suportada por vendedores grandes existentes já há bastante tempo. A SQL fornece uma interface relativamente padrão para todos os tipos de ferramentas. (Se você estiver preocupado com desempenho, tudo que posso dizer é que não tenho visto quaisquer dados conclusivos de comparações entre o desempenho de OO e o de sistemas relacionais.)

Mesmo se você não puder usar um banco de dados OO, deve considerar seriamente a compra de uma ferramenta de mapeamento O/R se tiver um *Modelo de Domínio* (116). Embora os padrões neste livro lhe digam muito sobre como construir um *Mapeador de Dados* (170), esta ainda é uma tarefa complicada. Vendedores de ferramentas passaram muitos anos trabalhando sobre esse problema, e ferramentas comerciais de mapeamento O/R são muito mais sofisticadas do que qualquer coisa que possa razoavelmente ser feita à mão. Embora as ferramentas não sejam baratas, você tem que comparar seu preço com o custo considerável de você mesmo escrever e manter tal camada.

Há ações a realizar para fornecer uma camada estilo banco de dados OO que possa trabalhar com bancos de dados relacionais. A JDO é uma proposta para a solução deste problema no mundo Java, mas ainda é muito cedo para dizer se se sairá bem. Ainda não tive experiência suficiente com ela para chegar a qualquer conclusão para este livro.

Contudo, mesmo se você comprar uma ferramenta, é uma boa idéia ter ciência destes padrões. Boas ferramentas O/R lhe dão muitas opções no mapeamento para um banco de dados, e estes padrões irão lhe ajudar a entender quando usar as diferentes opções. Não parta do princípio de que uma ferramenta acaba com todo o esforço. Ela o reduz consideravelmente, mas você descobrirá que usar e ajustar uma ferramenta O/R demanda um pequeno, porém significativo, trabalho.

O Problema Comportamental

Quando as pessoas falam sobre mapeamento O/R, normalmente enfocam os aspectos estruturais – como você relaciona tabelas e objetos. Entretanto, descobri que a parte mais difícil do exercício são seus aspectos arquiteturais e comportamentais. Já falei sobre as principais abordagens arquiteturais; o próximo passo é pensar no aspecto comportamental.

O problema comportamental é como fazer os diversos objetos carregarem e gravarem a si próprios no banco de dados. À primeira vista, isto não parece ser um grande problema. Um objeto cliente pode ter métodos para carregar e gravar que executem essa tarefa. Com certeza, com *Registro Ativo* (165), este é um caminho óbvio a tomar.

Se você carregar um punhado de objetos para a memória e modificá-los, você tem que manter registrado quais modificou e se assegurar de gravá-los de volta no banco de dados. Se você carregar apenas alguns objetos, isso é fácil. À medida que carrega mais e mais objetos, o trabalho aumenta, especialmente quando você cria algumas linhas e modifica outras, já que precisará das chaves das linhas criadas antes que possa modificar as linhas que se referem a elas. Este é um problema um pouco traiçoeiro a ser resolvido.

Enquanto você lê e modifica objetos, tem que se assegurar de que o estado do banco de dados com o qual está trabalhando permaneça consistente. Se você ler alguns objetos, é importante garantir que a leitura seja isolada, de modo que nenhum outro processo altere qualquer um dos objetos que leu enquanto estiver trabalhando com ele. De outra forma, você poderia ter dados inconsistentes e inválidos em seus objetos. Esta é a questão da concorrência, que é um problema traiçoeiro a ser resolvido; falaremos sobre ele no Capítulo 5.

Um padrão essencial à solução de ambos os problemas é a *Unidade de Trabalho* (187). Uma *Unidade de Trabalho* (187) mantém registro de todos os objetos lidos do banco de dados, junto com todos os objetos modificados de alguma maneira. Ela também trata a maneira como as atualizações são feitas no banco de dados. Em vez do programador da aplicação chamar explicitamente métodos de gravação, ele manda a unidade de trabalho confirmar (*commit*). Essa unidade de trabalho então organiza em uma seqüência todo o comportamento apropriado para o banco de dados, colocando todo o complexo processamento de confirmação em um local. A *Unidade de Trabalho* (187) é um padrão essencial sempre que as interações comportamentais com o banco de dados se tornarem difíceis de manejar.

Uma boa maneira de pensar na *Unidade de Trabalho* (187) é como um objeto que age como o controlador do mapeamento do banco de dados. Sem uma *Unidade de Trabalho* (187), normalmente a camada do domínio age como o controlador, decidindo quando ler e gravar no banco de dados. A *Unidade de Trabalho* (187) resulta da fatoração do comportamento do controlador de mapeamento do banco de dados em seu próprio objeto.

À medida que você carrega objetos, tem que ser cuidadoso para não carregar o mesmo objeto duas vezes. Se você fizer isso, terá dois objetos na memória que correspondem a uma única linha do banco de dados. Atualize ambos, e tudo fica muito confuso. Para lidar com isso, você precisa manter um registro de cada linha que lê em um *Mapa de Identidade* (196). Cada vez que você lê algum dado, primeiro deve verificar o *Mapa de Identidade* (196) para se assegurar de que já não o tenha lido. Se o dado já estiver carregado, você pode retornar uma segunda referência para ele. Dessa maneira, quaisquer atualizações serão apropriadamente coordenadas. Como benefício,

você também pode conseguir evitar uma chamada ao banco de dados já que o *Mapa de Identidade* (196) também funciona como um cache para o banco de dados. Não esqueça, entretanto, que o propósito principal de um *Mapa de Identidade* (196) é manter as identidades corretas, e não aumentar o desempenho.

Se você estiver usando um *Modelo de Domínio* (126), normalmente irá arranjar as coisas de modo que objetos conectados sejam carregados juntos de tal maneira que a leitura de um objeto Pedido carrega também seu objeto Cliente associado. Entretanto, com muitos objetos conectados, qualquer leitura de um objeto pode trazer um enorme grafo de objetos do banco de dados. Para evitar tais ineficiências, você precisa reduzir o que traz de volta, mas manter a porta aberta para pegar mais dados se precisar deles mais tarde. A *Carga Tardia* (200) se baseia na existência de um um recipiente para uma referência a um objeto. Há diversas variações sobre esse tema, mas todas elas modificam a referência ao objeto de modo que, em vez de apontar para o objeto real, elas referenciam o recipiente. Apenas se você tentar seguir, a conexão fará com que o objeto real seja trazido do banco de dados. Usando *Carga Tardia* (200) em pontos convenientes, você pode trazer do banco de dados apenas o suficiente em cada chamada.

Lendo Dados

Ao ler dados, gosto de pensar nos métodos como **buscadores** que encapsulam comandos SQL *select* com uma interface estruturada na forma de um método. Assim, você poderia ter métodos como *buscar(id)* ou *buscarPorCliente(cliente)*. Esses métodos claramente podem se tornar muito difíceis de lidar se você tiver 23 cláusulas diferentes no comando *Select*; mas felizmente isso é raro.

O lugar onde você coloca seus métodos de busca depende do padrão de interface usado. Se suas classes de interação com o banco de dados forem baseadas em tabela – ou seja, você tem uma instância da classe por tabela no banco de dados – então pode combinar os métodos de busca com as inserções e atualizações. Se suas classes de interação forem baseadas em linhas – ou seja, você tem uma classe de interação por linha no banco de dados – isso não funciona.

Com classes baseadas em linhas você pode tornar estáticas as operações de busca, mas fazer isso impedirá que você torne as operações de banco de dados substituíveis. Isso significa que você não pode, com o propósito de teste, substituir o banco de dados por um *Stub de Serviço* (469). Para evitar esse problema a melhor abordagem é ter objetos de busca separados. Cada classe de busca tem muitos métodos que encapsulam uma consulta SQL. Quando você executa a consulta, o objeto de busca retorna uma coleção dos objetos apropriados baseados em linhas.

Algo a ser observado em métodos de busca é que eles trabalham sobre o estado do banco de dados, não sobre o estado do objeto. Se você executar uma consulta no banco de dados para encontrar todas as pessoas em um clube, lembre-se de que quaisquer objetos pessoa que você tiver acrescentado ao clube na memória não serão pegos pela consulta. Como resultado disso, normalmente é sensato executar as consultas no início.

Ao ler dados, questões relativas a desempenho podem muitas vezes surgir. Isso leva a algumas regras básicas.

Tente trazer diversas linhas de uma vez. De modo especial, nunca execute consultas repetidas sobre a mesma tabela para obter diversas linhas. Quase sempre é

melhor pegar dados demais do que de menos (embora você tenha que ser cuidadoso em relação ao bloqueio de linhas demais com controle de concorrência pessimista). Assim, considere uma situação em que você precisa obter 50 pessoas que pode identificar por uma chave primária no seu modelo de domínio, mas só pode construir uma pesquisa que obtenha 200 pessoas, sobre a qual irá executar alguma lógica adicional para isolar as 50 de que você precisa. Normalmente, é melhor usar uma consulta que traga linhas desnecessárias do que realizar 50 consultas individuais.

Outra forma de evitar ir ao banco de dados mais do que uma vez é usar junções de modo que você possa trazer diversas tabelas com uma única consulta. O conjunto de registros resultante parece estranho, mas pode realmente acelerar as coisas. Neste caso, você pode ter um *Gateway* (436) que tenha dados oriundos da junção de diversas tabelas ou um *Mapeador de Dados* (170) que carregue diversos objetos de domínio com uma única chamada.

Entretanto, se estiver usando junções, tenha em mente que bancos de dados são otimizados para lidar com até três ou quatro junções por consulta. Acima disso, o desempenho sofre, embora você possa restaurar uma boa parte dele com visões em cache.

Muitas otimizações são possíveis no banco de dados. Elas envolvem agrupar dados freqüentemente referenciados, o uso cuidadoso de índices e a habilidade do banco de dados de usar cache em memória. Isso está fora do escopo deste livro, mas dentro do escopo de um bom DBA.

Em todos os casos, você deve moldar sua aplicação com seu banco de dados e dados específicos. Regras gerais podem guiar seu pensamento, mas suas circunstâncias particulares irão sempre ter suas próprias variações. Sistemas de bancos de dados e servidores de aplicação freqüentemente têm esquemas de cache sofisticados, e não há como eu predizer o que acontecerá em sua aplicação. Para cada regra que uso, tenho visto exceções surpreendentes, então reserve tempo para testes e ajustes de desempenho.

Padrões de Mapeamento Estrutural

Quando as pessoas falam em mapeamento objeto-relacional, na maioria das vezes se referem aos padrões de mapeamento estruturais, os quais você usa ao efetuar o mapeamento entre objetos na memória e tabelas do banco de dados. Estes padrões geralmente não são relevantes para o *Gateway de Tabelas de Dados* (151), mas você pode usar alguns deles se usar o *Gateway de Linhas de Dados* (158) ou o *Registro Ativo* (165). Você provavelmente precisará usar todos eles para o *Mapeador de Dados* (170).

Mapeando Relacionamentos

A questão central aqui é o modo diferente como objetos e relações lidam com associações, o que leva a dois problemas. Primeiro, há uma diferença na representação. Os objetos lidam com associações armazenando referências que são mantidas em tempo de execução ou por ambientes gerenciados por memória ou por endereços de memória. Os bancos de dados relacionais lidam com associações, criando uma chave para outra tabela. Segundo, os objetos podem usar facilmente coleções para lidar com referências múltiplas a partir de um único campo, enquanto que a normalização força todas as associações da relação a terem um único valor. Isso leva a inversões da estrutu-

ra de dados entre objetos e tabelas. Um objeto pedido evidentemente tem uma coleção de objetos Item de pedido que não precisam de qualquer referência de volta ao pedido. Entretanto, a estrutura da tabela é diferente – o Item deve incluir uma chave estrangeira para o pedido, já que o pedido não pode ter um campo multivalorado.

A maneira de lidar com o problema da representação é manter a identidade relacional de cada objeto como um *Campo Identidade* (226) no objeto e buscar estes valores para realizar o mapeamento nos dois sentidos entre as referências dos objetos e as chaves relacionais. É um processo tedioso, mas não tão difícil, uma vez que você compreenda a técnica básica. Quando você lê objetos do disco, usa um *Mapa de Identidade* (196) como tabela de busca das chaves relacionais para objetos. Cada vez que você se deparar com uma chave estrangeira na tabela, use *Mapeamento de Chave Estrangeira* (233) (veja a Figura 3.5) para ligar a referência interobjetos apropriada. Se você não tiver a chave no *Mapa de Identidade* (196), você precisa ir ao banco de dados para obtê-la ou usar *Carga Tardia* (200). Cada vez que você grava um objeto, você o faz em uma linha com a chave correta. Qualquer referência interobjetos é substituída pelo campo ID do objeto alvo.

Estabelecidos esses fundamentos, a manipulação da coleção requer uma versão mais complexa do *Mapeamento de Chave Estrangeira* (233) (veja a Figura 3.6). Se um objeto tiver uma coleção, você precisará executar outra busca para encontrar todas as linhas que referenciam o ID do objeto fonte (ou você pode agora evitar a consulta com *Carga Tardia* (200)). Cada objeto que retorna é criado e acrescentado à coleção. Salvar a coleção envolve salvar cada objeto nela e assegurar de que tenha uma chave estrangeira para o objeto fonte. Isso se torna confuso, especialmente quando você tem que detectar objetos adicionados ou removidos da coleção. Pode se tornar repetitivo quando você domina o processo, e é por isso que alguma forma de abordagem baseada em metadados torna-se um procedimento óbvio para sistemas maiores (entrarei em mais detalhes sobre isso mais tarde). Se a coleção de objetos não for usada fora do escopo do dono dessa coleção, você pode usar o *Mapeamento Dependente (*256) para simplificar o mapeamento.

Um caso diferente ocorre com um relacionamento muitos-para-muitos, o qual tem uma coleção em ambos as extremidades. Um exemplo é uma pessoa que tenha

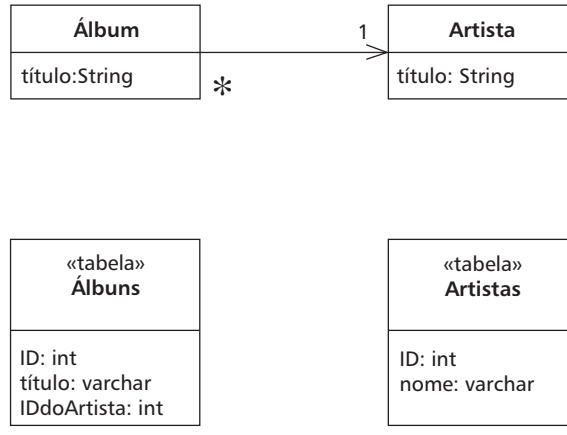


Figura 3.5 Use um *Mapeamento de Chave Estrangeira* (233) para mapear um campo monovalorado.

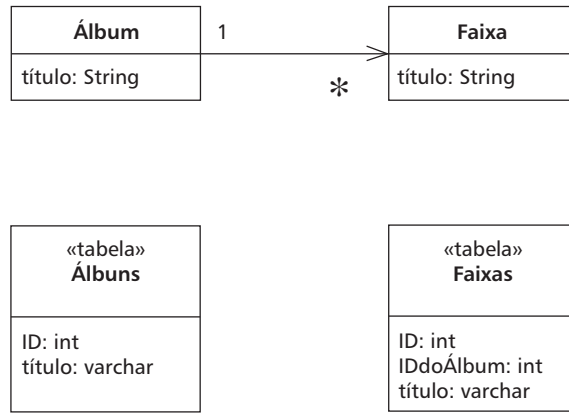


Figura 3.6 Use um *Mapeamento de Chave Estrangeira* (233) para mapear um campo coleção.

muitas habilidades e cada habilidade conheça as pessoas que a possuem. Bancos de dados relacionais não podem lidar com isso diretamente, de modo que você usa um *Mapeamento de Tabela de Associação* (244) (veja a Figura 3.7) para criar uma nova tabela relacional apenas para lidar com a associação muitos-para-muitos.

Ao trabalhar com coleções, um erro freqüente é se basear na ordenação dentro delas. Em linguagens OO, é comum o uso de coleções ordenadas, como listas e vetores – com certeza, isso freqüentemente torna os testes mais fáceis. Entretanto, é muito difícil manter uma coleção ordenada arbitrariamente quando salva em um banco de dados relacional. Por este motivo, vale a pena considerar o uso de conjuntos desordenados para armazenar coleções. Outra opção é escolher uma forma de ordenação sempre que executar uma pesquisa em uma coleção, embora isso possa ser bastante custoso.

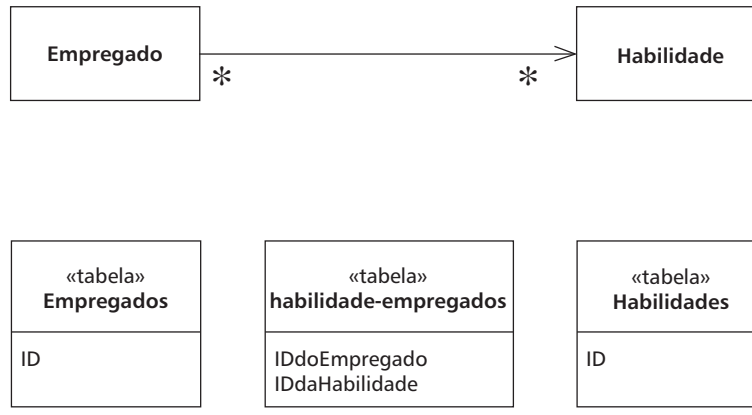


Figura 3.7 Use um *Mapeamento de Tabela de Associação* (244) para mapear uma associação muitos-para-muitos.

Em alguns casos, a integridade referencial pode tornar as atualizações mais complexas. Os sistemas modernos permitem que você protele a verificação da integridade referencial para o final da transação. Se isso for possível, não há motivo para não fazê-lo. De outro modo, o banco de dados irá verificar a cada gravação. Nesse caso, você tem que ser cuidadoso para realizar sua atualização na ordem correta. Como fazer isso está fora do escopo deste livro, mas outra técnica é realizar uma ordenação topológica de suas atualizações. Outra é codificar explicitamente quais tabelas são escritas em qual ordem. Isso pode às vezes reduzir problemas de *deadlock* dentro do banco de dados que, muito freqüentemente, fazem com que as transações sejam desfeitas *(roll back).*

Campo Identidade (226) é usado para referências interobjetos que viram chaves estrangeiras, mas nem todos os relacionamentos de um objeto precisam ser persistidos dessa forma. Pequenos *Objetos Valor* (453), como intervalos de datas e objetos do tipo moeda claramente não devem ser representados como sua própria tabela no banco de dados. Em vez disso, pegue todos os campos do *Objeto Valor* (453) e coloque-os dentro do objeto associado como um *Valor Embutido* (261). Já que os *Objetos Valor* (486) têm semântica de valor, você pode facilmente criá-los cada vez que obtiver uma leitura, e não precisa se incomodar com um *Mapa de Identidade* (196). Gravá-los também é fácil – simplesmente retire a referência para o objeto e jogue seus campos para a tabela proprietária.

Você pode fazer esse tipo de coisa em uma escala maior pegando um conjunto de objetos e gravando-os em uma única coluna em uma tabela como um *LOB Serializado* (264). LOB quer dizer "Large OBject" (Objeto Grande), que pode ser binário (BLOB) ou textual (CLOB – Character Large OBject). Serializar um grupo de objetos como um documento XML é um caminho óbvio para uma estrutura hierárquica de objetos. Dessa forma você pode pegar um grupo de pequenos objetos associados em uma única leitura. Muitas vezes os bancos de dados têm desempenho pobre com objetos pequenos altamente interconectados – onde você gasta muito tempo fazendo muitas chamadas pequenas a banco de dados. Estruturas hierárquicas como organogramas e contas de materiais são no que um *LOB Serializado* (264) pode economizar muitas idas e vindas a um banco de dados.

O lado negativo é que a SQL não está ciente do que está acontecendo, de modo que você não pode fazer consultas portáveis sobre a estrutura do banco de dados. Mais uma vez, XML pode ser sua salvação aqui, permitindo a você embutir expressões de pesquisa XPath dentro de chamadas SQL, embora embutir não seja de modo algum padrão no momento. Como resultado, *LOB Serializado* (264) é melhor usado quando você não quiser pesquisar as partes da estrutura armazenada.

Geralmente um *LOB Serializado* (264) é melhor para um grupo relativamente isolado de objetos que fazem parte de uma aplicação. Se você usá-lo demais, acaba transformando seu banco de dados em pouco mais do que um sistema de arquivos transacional.

Herança

Nas hierarquias acima, estou falando sobre hierarquias composicionais, como uma árvore de partes, o que tradicionalmente não é o forte dos sistemas relacionais. Há outro tipo de hierarquia que causa dores de cabeça relacionais: uma hierarquia de classes associadas por herança. Já que não há um modo padrão de tratar herança em SQL, novamente temos que executar um mapeamento. Para qualquer estrutura de

herança há basicamente três opções. Pode haver uma tabela para todas as classes na hierarquia: *Herança de Tabela Única* (269) (veja a Figura 3.8); uma tabela para cada classe concreta: *Herança de Tabela Concreta* (283) (veja Figura 3.9); ou uma tabela por classe na hierarquia: *Herança de Tabela de Classes* (276) (veja a Figura 3.10).

Os compromissos são todos entre duplicação da estrutura de dados e velocidade de acesso. A *Herança de Tabela de Classes* (276) é o relacionamento mais simples en-

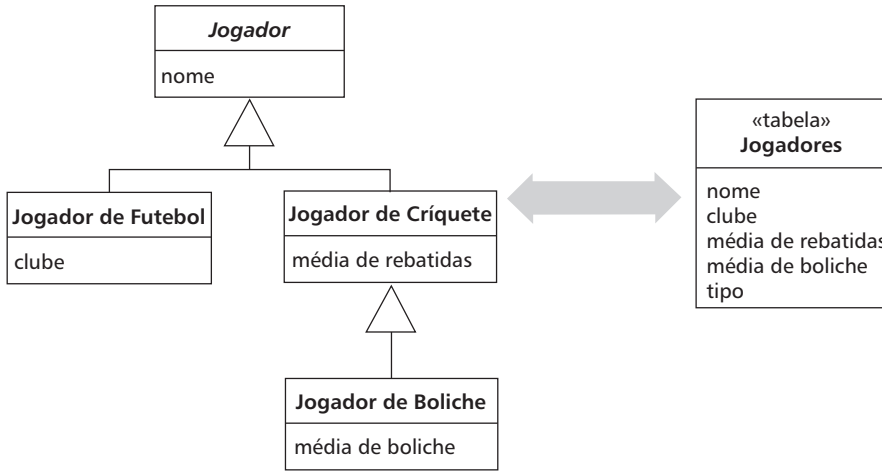


Figura 3.8 A *Herança de Tabela Única* (269) usa uma única tabela para armazenar todas as classes em uma hierarquia.

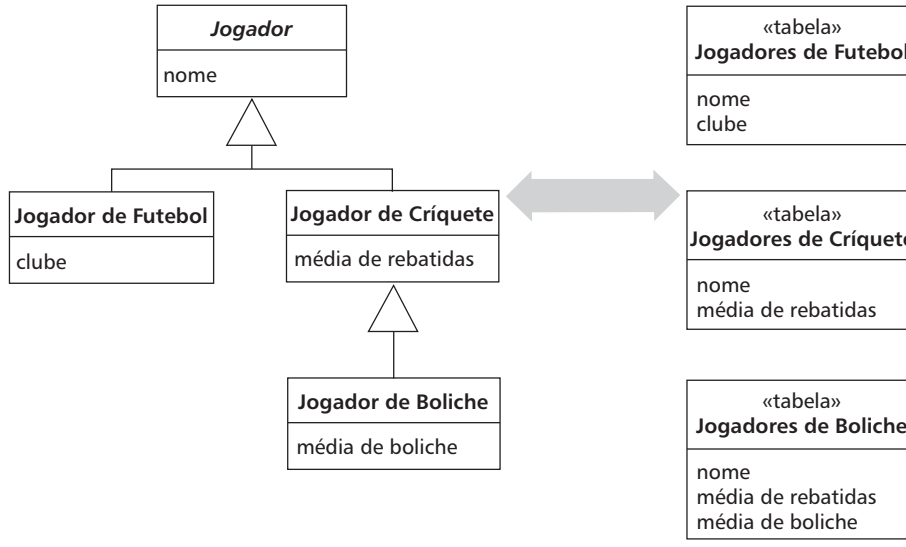


Figura 3.9 A *Herança de Tabela Concreta* (283) usa uma única tabela para armazenar cada classe concreta em uma hierarquia.

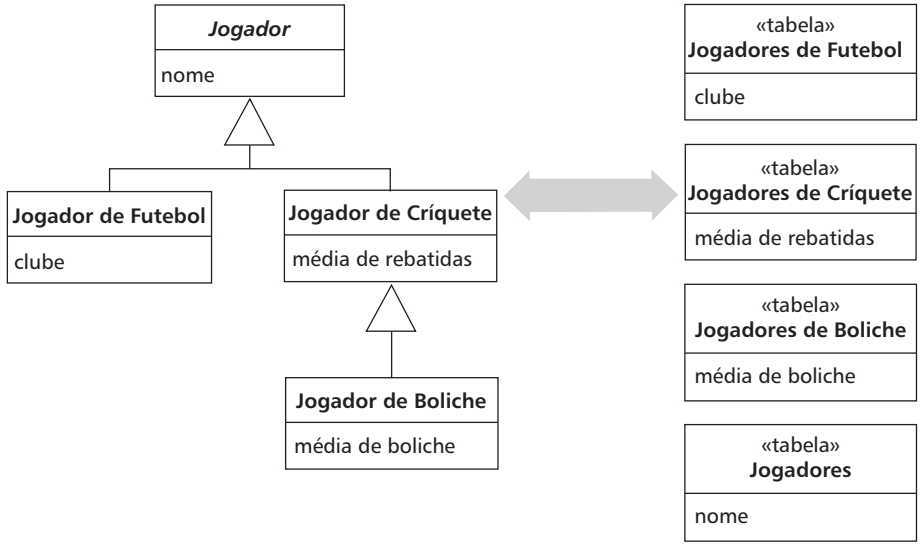


Figura 3.10 A *Herança de Tabela de Classes* (276) usa uma única tabela para cada classe em uma hierarquia.

tre as classes e as tabelas, mas precisa de junções múltiplas para carregar um único objeto, o que geralmente reduz o desempenho. A *Herança de Tabela Concreta* (283) evita as junções, permitindo que você pegue um único objeto de uma tabela, mas é sensível a alterações. Com qualquer alteração em uma superclasse você tem que se lembrar de alterar todas as tabelas (e o código de mapeamento). Alterar a própria hierarquia pode causar alterações ainda maiores. Além disso, a falta de uma tabela da superclasse pode tornar o gerenciamento das chaves difícil e atrapalhar a integridade referencial, embora reduza a disputa de bloqueios na tabela da superclasse. Em alguns bancos de dados, o maior problema da *Herança de Tabela Única* (269) é o desperdício de espaço, já que cada linha precisa ter colunas para todos os subtipos possíveis, e isso leva a colunas vazias. No entanto, muitos bancos de dados fazem um bom trabalho ao comprimir o espaço desperdiçado em tabelas. Outro problema com a *Herança de Tabela Única* (269) é seu tamanho, tornando-a um gargalo para os acessos. Sua grande vantagem é que ela coloca todas as coisas em um único lugar, o que torna a modificação mais fácil e evita junções.

As três opções não são mutuamente exclusivas e, em uma hierarquia, você pode misturar padrões. Por exemplo, você poderia ter diversas classes lidas juntas com *Herança de Tabela Única* (269) e usar *Herança de Tabela de Classes* (276) para alguns casos pouco comuns. É claro que misturar padrões acrescenta complexidade.

Não há um vencedor nítido aqui. Você deve levar em consideração suas próprias circunstâncias e preferências, como com todo o resto desses padrões. Minha primeira escolha tende a ser a *Herança de Tabela Única* (269), já que é fácil de realizar e resiliente a muitas refatorações. Tendo a usar as outras duas quando necessário para ajudar a resolver os problemas inevitáveis de colunas irrelevantes e desperdiçadas. Muitas vezes, o melhor é conversar com os DBAs. Eles freqüentemente têm bons conselhos sobre o tipo de acesso que faz mais sentido no banco de dados.

Todos os exemplos recém-descritos, e os padrões, usam herança única. Embora herança múltipla esteja ficando menos em uso atualmente e a maioria das linguagens esteja cada vez mais evitando-a, a questão ainda aparece em mapeamentos O/R quando você usa interfaces, como em Java e .NET. Os padrões aqui não entram nesse tópico especificamente, mas na essência você lida com herança múltipla usando variações do trio de padrões de herança. A *Herança de Tabela Única* (269) coloca todas as superclasses e interfaces em uma grande tabela, a *Herança de Tabela de Classes* (276) cria uma tabela separada para cada interface e superclasse, enquanto que a *Herança de Tabela Concreta* (283) inclui todas as interfaces e superclasses em cada tabela concreta.

Construindo o Mapeamento

Quando mapeia para um banco de dados relacional, você encontra essencialmente três situações:

- Você mesmo escolhe o esquema.
- Você tem que mapear para um esquema preexistente, o qual não pode ser alterado.
- Você tem que mapear para um esquema preexistente, porém alterações nele são negociáveis.

O caso mais simples ocorre quando você mesmo está criando o esquema e tem complexidade de pequena a moderada na sua lógica de domínio, resultando em um *Roteiro de Transação* (120) ou em *Módulo Tabela* (134). Nesse caso, você pode projetar as tabelas em torno dos dados usando técnicas clássicas de projeto de banco de dados. Use um *Gateway de Linha de Dados* (158) ou um *Gateway de Tabela de Dados* (151) para retirar toda SQL da lógica do domínio.

Se você estiver usando um *Modelo de Domínio* (126), deve tomar cuidado com um projeto que se pareça com um projeto de banco de dados. Neste caso, construa seu *Modelo de Domínio* (126) sem considerar o banco de dados, de modo que você possa simplificar da melhor forma a lógica do domínio. Trate o projeto do banco de dados como uma maneira de persistir os dados dos objetos. O *Mapeador de Dados* (170) lhe dá a maior flexibilidade aqui, porém é mais complexo. Se o projeto de um banco de dados isomórfico ao *Modelo de Domínio* (126) fizer sentido, você pode considerar um *Registro Ativo* (165) em vez disso.

Embora criar primeiramente o modelo seja uma forma razoável de pensar nele, o conselho só se aplica dentro de ciclos interativos pequenos. Passar seis meses criando um *Modelo de Domínio* (126) sem bancos de dados e então decidir persisti-lo assim que tiver terminado é altamente arriscado. O perigo é que o projeto resultante tenha problemas de desempenho fraco que demandem refatoração demais para serem consertados. Em vez disso, construa o banco de dados a cada iteração, em não mais de seis semanas de duração, preferencialmente menos. Dessa forma, você terá retorno rápido e contínuo sobre como suas interações de banco de dados funcionam na prática. Dentro desta tarefa particular, você deve pensar no *Modelo de Domínio* (126) primeiro, mas integre cada parte do *Modelo de Domínio* (126) no banco de dados à medida que avança.

Quando o esquema já está lá, suas escolhas são similares, mas o processo é ligeiramente diferente. Com lógica de domínio simples você cria classes *Gateway de Linhas*

de Dados (158) ou *Gateway de Tabelas de Dados* (151) que imitem o banco de dados, e coloca a camada da lógica do domínio sobre elas. Com lógica de domínio mais complexa você precisará de um *Modelo de Domínio* (126), o qual é altamente improvável de casar com o desenho do banco de dados. Portanto, construa um *Modelo de Domínio* (126) gradualmente e inclua *Mapeadores de Dados* (170) para persistir os dados no banco de dados preexistente.

Mapeamento Duplo

Ocasionalmente me deparo com situações em que o mesmo tipo de dados precisa ser trazido de mais de uma fonte. Pode haver múltiplos bancos de dados com os mesmos dados, porém com pequenas diferenças no esquema devido a alguma reutilização feita com "copiar e colar". (Nesta situação, o tamanho do aborrecimento é inversamente proporcional ao tamanho da diferença.) Outra possibilidade é usar mecanismos diferentes, armazenando os dados, às vezes, em um banco de dados e, às vezes, em mensagens. Você pode querer trazer dados similares de uma combinação de mensagens XML, transações CICS e tabelas relacionais.

A opção mais simples é ter diversas camadas de mapeamento, uma para cada fonte de dados. Entretanto, se os dados forem muito semelhantes, isso pode levar a muita duplicação. Nesse caso você poderia considerar um mapeamento em dois passos. O primeiro passo converte os dados do esquema em memória para um esquema lógico de armazenamento de dados. O esquema lógico de armazenamento de dados é projetado para maximizar as similaridades dos formatos das fontes de dados. O segundo passo mapeia do esquema lógico de armazenamento de dados para o esquema de armazenamento de dados físico real. Este segundo passo contém as diferenças.

O passo extra só vale a pena quando você tem muitas similaridades, de modo que você somente deve usá-lo quando você tem depósitos físicos de dados similares, porém irritantemente diferentes. Trate o mapeamento do depósito lógico para o depósito físico dos dados como um *Gateway* (436) e use qualquer uma das técnicas de mapeamento para mapear da lógica da aplicação para o depósito lógico dos dados.

Usando Metadados

Neste livro, a maior parte dos meus exemplos usa código escrito à mão. Com mapeamento simples e repetitivo isso pode levar a código simples e repetitivo – e código repetitivo é um sinal de algo errado com o projeto. Há muito que você pode fazer fatorando comportamentos comuns com herança e delegação – práticas OO boas e honestas – mas também há uma abordagem mais sofisticada usando *Mapeamento em Metadados* (295).

O *Mapeamento em Metadados* (295) é baseado na condensação do mapeamento em um arquivo de metadados que detalha como as colunas no banco de dados são mapeadas em campos nos objetos. O objetivo disto é que, assim que tenha os metadados, você pode evitar o código repetitivo usando geração de código ou programação reflexiva.

Com uma pequena quantidade de metadados você obtém uma grande expressividade. Uma linha de metadados pode dizer algo como

```
<nomeCampo = cliente classeAlvo = "Cliente", colunaBD = "IDCliente", tabelaAlvo = "clientes"
limiteInferior = "1" limiteSuperior = "1" metodoGravacao = "carregarCliente"/>
```

A partir disso, você pode definir o código de leitura e gravação, gerar automaticamente junções para este caso, executar toda SQL, forçar a multiplicidade do relacionamento e até fazer coisas como computar ordens de gravação sob a presença da integridade referencial. É por isso que ferramentas de mapeamento O/R tendem a usar metadados.

Quando usa *Mapeamento em Metadados* (295), você tem a fundamentação necessária para criar consultas baseadas em objetos em memória. Um *Objeto de Pesquisa* (304) permite-lhe criar suas consultas baseadas em objetos e dados em memória de um modo que os desenvolvedores não precisem conhecer nem SQL nem os detalhes do esquema relacional. O *Objeto de Pesquisa* (304) pode então usar o *Mapeamento em Metadados* (295) para traduzir expressões baseadas em campos de objetos para a SQL apropriada.

Leve isso adiante o suficiente e você pode formar um *Repositório* (309) que esconda amplamente o banco de dados da vista. Quaisquer consultas ao banco de dados podem ser feitas em um *Repositório* (309) pelos *Objetos de Pesquisa* (304), e os desenvolvedores não conseguem perceber se os objetos foram trazidos da memória ou do banco de dados. O *Repositório* (309) funciona bem com sistemas de *Modelo de Domínio* (126) ricos.

Apesar das muitas vantagens dos metadados, neste livro enfoquei exemplos escritos à mão porque acredito que são mais fáceis de entender. Assim que você dominar os padrões e conseguir escrevê-los à mão para sua aplicação, poderá descobrir como usar metadados para tornar as coisas mais simples.

Conexões de Bancos de Dados

A maior parte das interfaces de bancos de dados baseia-se em algum tipo de objeto de conexão de banco de dados para atuar como um *link* entre o código da aplicação e o banco de dados. Normalmente uma conexão deve ser aberta antes de você poder executar comandos sobre o banco de dados. Em geral, você precisa de uma conexão explícita para criar e executar um comando. Durante todo o tempo em que você executa este comando a conexão deve estar aberta. Buscas retornam um *Conjunto de Registros* (473). Algumas interfaces fornecem *Conjuntos de Registros* (473) desconectados, os quais podem ser manipulados após a conexão ser fechada. Outras interfaces fornecem apenas *Conjuntos de Registros* (473) conectados, o que significa que a conexão deve permanecer aberta enquanto o *Conjunto de Registros* (473) é manipulado. Se você estiver executando dentro de uma transação, normalmente ela está conectada a uma conexão específica e esta deve permanecer aberta enquanto aquela estiver acontecendo.

Em muitos ambientes, é custoso criar uma conexão, o que faz com que valha a pena criar um *pool* de conexões. Nesta situação os desenvolvedores solicitam uma conexão do *pool* e a liberam quando tiverem terminado, em vez de criar e fechar a conexão. Atualmente, a maioria das plataformas lhe dá esta possibilidade de *pooling*, de modo que raramente você mesmo terá que criá-lo. Se tiver, primeiro verifique se o *pooling* vai realmente ajudar no desempenho. Cada vez mais os ambientes tornam mais rápido criar uma nova conexão, de modo que não há necessidade de você usar um *pool.*

Ambientes que fornecem *pooling* muitas vezes o colocam por trás de uma interface que se parece com a criação de uma nova conexão. Dessa forma, você não sabe

se está obtendo uma conexão nova ou uma alocada de um *pool*. Esta é uma coisa boa, já que a escolha de usar um *pool* ou não está encapsulada. De maneira semelhante, fechar uma conexão pode não fechá-la realmente, mas apenas retorná-la ao *pool* para que outra pessoa a use. Nesta discussão usarei "abrir" e "fechar", que podem ser substituídos por "obter" do *pool* e "liberar" de volta para o *pool*.

Custosas de criar ou não, as conexões precisam de gerenciamento. Já que elas são recursos custosos de gerenciar, devem ser fechadas assim que você tiver terminado de usá-las. Além disso, se você estiver usando uma transação, normalmente precisa se assegurar de que cada comando dentro de uma transação específica use a mesma conexão.

O conselho mais comum é obter uma conexão explicitamente, usando uma chamada para um *pool* ou um gerente de conexão, e então fornecê-la para cada comando de banco de dados que você quiser executar. Assim que você não precisar mais da conexão, feche-a. Este conselho leva a algumas questões: assegurar-se de que você tenha a conexão e todos os locais onde precisar dela e garantir que não se esquecerá de fechá-la ao final.

Para se assegurar de que você tenha uma conexão onde precisa, há duas escolhas. Uma é passar a conexão como um parâmetro explícito. O problema nisso é que, por exemplo, a conexão é adicionada a todas as chamadas de método onde seu único propósito é ser passada para algum outro método cinco camadas abaixo da pilha de chamadas. É claro que esta é a situação que demanda um *Registro* (448). Já que você não quer diversas *threads* usando a mesma conexão, irá querer um *Registro* (448) com escopo de *thread*.

Se você for esquecido como eu, o fechamento explícito não é uma boa idéia. É muito fácil esquecer de fazê-lo quando deveria. Você também não pode fechar a conexão com cada comando porque pode estar sendo executado dentro de uma transação, e o fechamento irá normalmente fazer com que a transação seja desfeita.

Como uma conexão, a memória é um recurso que precisa ser liberado quando você não estiver usando. Ambientes modernos atualmente fornecem gerenciamento automático de memória e coleta de lixo (*garbage collection*), de modo que uma maneira de se assegurar de que as conexões sejam fechadas é usar um coletor de lixo. Nesta abordagem, a própria conexão ou um objeto que se refira a ela fecha a conexão durante a coleta de lixo. O bom disso é que usa o mesmo esquema de gerenciamento usado para a memória sendo dessa forma conveniente e familiar ao mesmo tempo. O problema é que o fechamento da conexão só acontece quando o coletor realmente reivindica a memória, e isto pode ocorrer um pouco depois do momento em que a conexão tiver perdido sua última referência. O resultado é que conexões sem referência podem continuar existindo por um certo tempo antes de serem fechadas. Se isso é um problema ou não depende muito do seu ambiente específico.

No geral, não gosto de depender da coleta de lixo. Outros esquemas – até o fechamento explícito – são melhores. Ainda assim, a coleta de lixo é um bom *backup* no caso do esquema normal falhar. Afinal, é melhor ter as conexões fechadas em algum momento do que tê-las perdurando para sempre.

Já que as conexões são tão atreladas a transações, uma boa maneira de gerenciá-las é vinculando-as a uma transação. Abra uma conexão quando você começar uma transação e feche-a quando você confirmá-la ou desfazê-la. Faça com que a transação saiba qual conexão está usando de modo que você possa ignorar a conexão completamente e lidar apenas com a transação. Já que a conclusão da transação tem um efei-

to visível, é mais fácil lembrar de confirmá-la e localizá-la se você esquecer. Uma *Unidade de Trabalho* (184) se encaixa com perfeição para gerenciar tanto a transação quanto a conexão.

Se você fizer coisas fora de uma transação, como uma leitura de dados invariáveis, use uma conexão nova para cada comando. *Poolings* podem lidar com quaisquer questões relacionadas à criação de conexões de curta duração.

Se você estiver usando um *Conjunto de Registros* (473) desconectado, pode abrir uma conexão para colocar os dados no conjunto de registros e fechá-la enquanto manipula os dados do *Conjunto de Registros* (473). A seguir, quando você já tiver feito o que precisava com os dados, pode abrir uma nova conexão, e transação, para gravar os dados. Se você fizer isto, precisará se preocupar com os dados alterados durante a manipulação do *Conjunto de Registros* (473). Este é um tópico sobre o qual falarei junto com o controle de concorrência.

As especificidades do gerenciamento de conexão são características do seu *software* de interação com o banco de dados, de modo que a estratégia que você usa é muitas vezes ditada pelo seu ambiente.

Questões Diversas

Você perceberá que alguns dos códigos exemplo usam declarações `select` na forma `select * from`, enquanto que outros usam colunas com nomes. Usar `select *` pode ter sérios problemas em alguns *drivers* de bancos de dados, que podem falhar se uma nova coluna for acrescentada ou se uma coluna for reordenada. Embora ambientes mais modernos não sofram isso, não é sábio usar `select *` se você estiver usando índices posicionais para obter informações das colunas, pois o reordenamento de uma coluna irá fazer com que o código falhe. Não há problema em usar índices de nomes de colunas com um `select *` e de fato estes são mais claros de ler. Entretanto eles podem ser mais lentos, embora isso provavelmente não vá fazer muita diferença dado o tempo da chamada SQL. Como sempre, meça para se certificar.

Se você usar índices de números de coluna, precisa se assegurar de que os acessos ao conjunto resultante sejam muito próximos à definição da declaração SQL de modo que não saiam de sincronia caso as colunas sejam reordenadas. Conseqüentemente, se você estiver usando *Gateway de Tabelas de Dados* (151), deve usar índices de nomes de colunas, pois o conjunto resultante é usado por todo código que executa uma operação de busca nesse *gateway*. O resultado disso é que geralmente vale a pena ter testes de caso simples de criação/leitura/remoção para cada estrutura de mapeamento de banco de dados que você usar. Isso ajudará a pegar casos quando sua SQL sai de sincronia com seu código.

Sempre vale a pena o esforço de usar SQL estática que possa ser pré-compilada, em vez de SQL dinâmica que tem que ser compilada a cada vez. A maioria das plataformas lhe dá um mecanismo para pré-compilar SQL. Uma boa regra é evitar o uso de concatenação de *strings* para formar pesquisas SQL.

Muitos ambientes lhe dão a habilidade de fazer um lote com diversas pesquisas SQL em uma única chamada ao banco de dados. Não fiz isso nestes exemplos, mas é certamente uma tática que você deve usar no código de produção. Como você faz isso varia com a plataforma.

Para conexões nestes exemplos, apenas os invoco com uma chamada a um objeto "DB", o qual é um *Registro* (448). Como você obtém uma conexão dependerá de

seu ambiente, de modo que você irá substituir isto pelo que precisar fazer. Não envolvi transações em nenhum dos padrões além daqueles sobre concorrência. Mais uma vez, você precisará combinar com o que o seu ambiente precisar.

Leitura Adicional

O mapeamento objeto-relacional é um fato para a maioria das pessoas, então não é surpresa que haja muita coisa escrita sobre isso. A surpresa é que não há um único livro coerente, completo e atualizado, motivo pelo qual dedico tanto deste a este assunto complicado, porém interessante.

O bom do mapeamento de banco de dados é que existem por aí muitas idéias para serem roubadas. As fontes intelectuais mais vitimadas são [Brown and Whitenack], [Ambler], [Yoder] e [Keller and Coldewey]. Eu certamente o encorajaria a dar uma boa olhada nesse material para suplementar os padrões deste livro.

Capítulo 4

Apresentação Web

Uma das maiores mudanças nas aplicações corporativas nos últimos anos foi o surgimento das interfaces com o usuário baseadas em navegadores Web. Elas trazem muitas vantagens: nenhum *software* cliente para instalar, uma abordagem comum para a interface com o usuário e um acesso universal fácil. Além disso, uma série de ambientes tornam fácil criar uma aplicação Web.

A preparação de uma aplicação Web começa com o próprio *software* servidor. Normalmente ele tem algum tipo de arquivo de configuração que indica quais URLs devem ser manipuladas por quais programas. Freqüentemente, um único servidor Web pode lidar com muitos tipos de programas. Estes programas podem ser dinâmicos e podem ser acrescentados a um servidor colocando-os em um diretório apropriado. O trabalho do servidor Web é interpretar a URL de um pedido e passar o controle para um programa servidor. Há duas formas principais de estruturar um programa em um servidor Web: como um *roteiro* ou como uma página servidora (*server page*).

O roteiro é um programa, geralmente com funções ou métodos para tratar a chamada HTTP. Exemplos incluem roteiros CGI e servlets Java. O texto do programa pode fazer quase tudo o que um programa pode fazer, e o roteiro pode ser divido em sub-rotinas, além de poder criar e usar outros serviços. Ele obtém dados da página Web examinando a requisição HTTP, uma *string*. Em alguns ambientes isso é feito por meio de uma busca usando expressões regulares na *string* de requisição – a facilidade oferecida por Perl para fazer isso a torna uma escolha popular para roteiros CGI. Outras plataformas, como servlets Java, fazem essa busca por programa, o que permite ao programador acessar as informações da requisição por meio de uma interface com palavras-chave. Isso, no mínimo, significa um número menor de expressões regulares para criar problemas. A saída do servidor Web é outra *string* – a resposta – na qual o roteiro pode escrever usando as operações comuns da linguagem de gravação em *streams*.

Escrever uma resposta HTML por meio de comandos de *streams* é incômodo para os programadores e quase impossível para não-programadores que, no entanto, sentiriam-se confortáveis preparando páginas HTML. Isso levou à idéia de páginas servidoras, em que o programa é estruturado em torno da página-texto de retorno. Você escreve a página de retorno em HTML e insere *scriplets* no HTML para executar código em determinados pontos. Exemplos desta abordagem incluem PHP, ASP e JSP.

A abordagem da página servidora funciona bem quando há pouco processamento da resposta, como em "Mostre-me os detalhes do álbum # 1234". As coisas ficam bem mais confusas quando você tem que tomar decisões baseadas na entrada, como diferentes formatos de apresentação para álbuns clássicos ou de jazz.

Devido ao fato do estilo roteiro funcionar melhor para interpretar a requisição e o estilo página servidora funcionar melhor para formatar a resposta, existe a opção óbvia de usar um roteiro para interpretar a requisição e uma página servidora para formatar a resposta. Esta separação é uma idéia antiga que apareceu primeiro em interfaces com o usuário usando o padrão *Modelo Vista Controlador* (315). Combine-a com a noção essencial de que lógica não-relacionada à apresentação deve ser dela extraída, e temos uma ótima base para os conceitos deste padrão.

O *Modelo Vista Controlador* (315) (veja a Figura 4.1) é um padrão amplamente referenciado, mas freqüentemente mal-compreendido. Antes das aplicações Web entrarem em cena, a maior parte das demonstrações do *Modelo Vista Controlador* (315) que examinei o compreendiam erradamente. Uma das principais razões para a confusão era o uso da palavra "controlador." Controlador é usado em vários contextos diferentes, e geralmente eu a encontrava usada de uma maneira diferente daquela descrita no *Modelo Vista Controlador* (315). Por esse motivo, prefiro usar o termo **controlador de entrada** para o controlador no *Modelo Vista Controlador* (315).

Uma requisição chega a um controle de entrada, o qual extrai as informações dessa requisição. Ele então encaminha a lógica de negócio para um objeto de domínio apropriado. Esse objeto conversa com a fonte de dados e realiza todas as tarefas indicadas pela requisição, assim como reúne informações para a resposta. Quando tiver terminado, ele devolve o controle para o controlador de entrada, o qual analisa os resultados e decide qual vista é necessária para exibir a resposta. Ele então passa o controlador, juntamente com os dados da resposta, para a vista. O repasse do controlador para a vista, muitas vezes, não é uma chamada direta e, freqüentemente, envolve a colocação dos dados em um local pré-combinado, alguma forma de objeto de sessão HTTP compartilhado entre o controlador de entrada e a vista.

A primeira e mais importante razão para aplicar o *Modelo Vista Controlador* (315) é assegurar que os modelos estejam completamente separados da apresentação Web. Isso torna mais fácil modificar a apresentação, assim como acrescentar apresentações adicionais mais tarde. Colocar o processamento em objetos *Roteiro de Transação* (120) ou *Modelo de Domínio* (126) também tornará mais fácil testá-los. Isto é especialmente importante se você estiver usando uma página servidora como sua vista.

Neste ponto, chegamos a um segundo uso da palavra "controlador". Muitos projetos de interface separam os objetos da apresentação dos do domínio com uma camada intermediária de objetos do tipo *Controlador de Aplicação* (360). O objetivo de um *Controlador de Aplicação* (360) é lidar com o fluxo de uma aplicação, decidindo quais telas devem aparecer em qual ordem. Ele pode aparecer como parte da camada de apresentação ou você pode pensar nele como uma camada separada que realiza a mediação entre as camadas de apresentação e de domínio. Os *Controladores de*

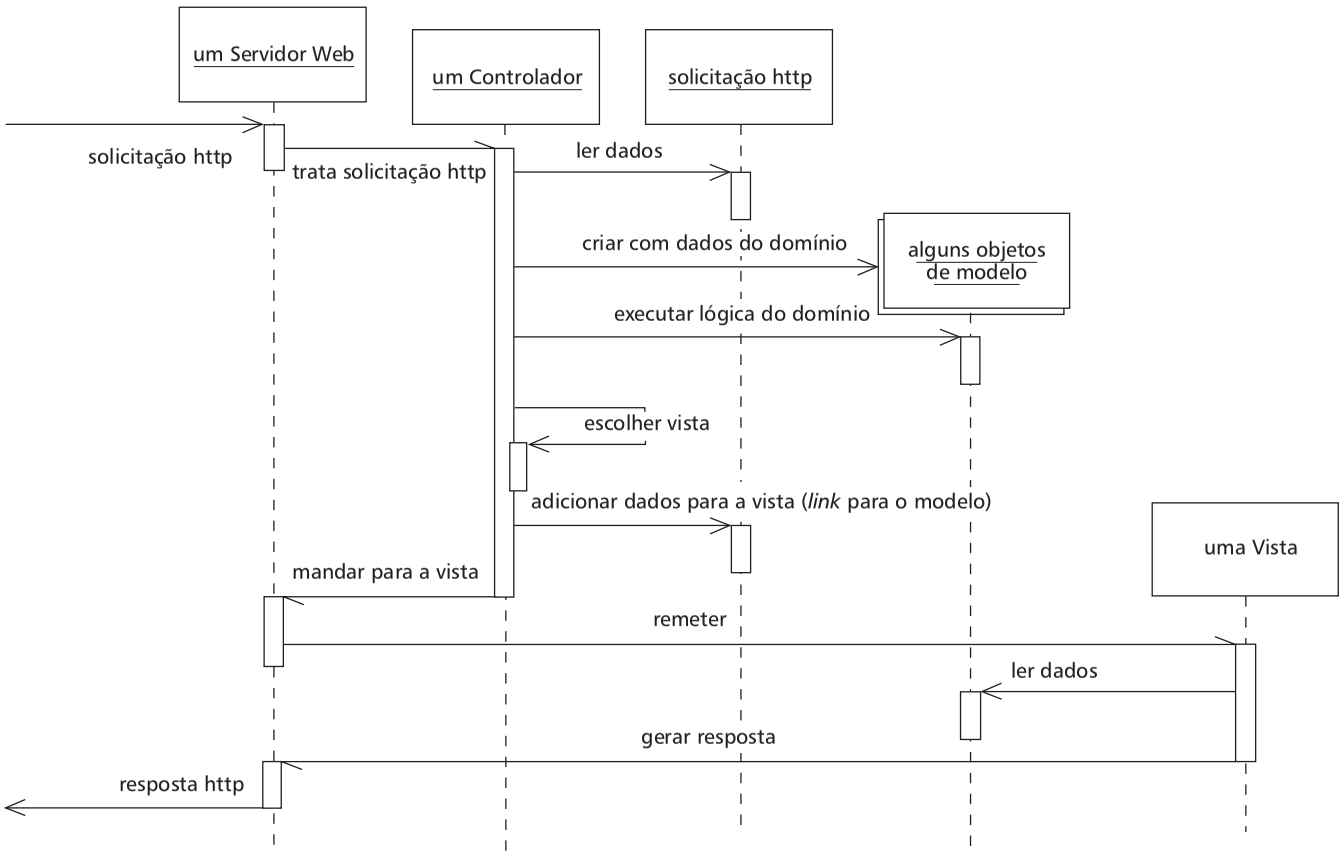


Figura 4.1 Uma visão geral de como os papéis do modelo, vista e controle de entrada trabalham juntos em um servidor Web. O controlador trata a requisição, faz com que o modelo execute a lógica do domínio e então faz com que a vista crie uma resposta baseada no modelo.

Aplicação (360) podem ser escritos para serem independentes de qualquer apresentação em particular. Nesse caso, eles podem ser reutilizados entre as aplicações. Isso funciona bem se você tiver diferentes apresentações com os mesmos fluxo e navegação básicos, embora, muitas vezes, seja melhor prover um fluxo diferente para cada apresentação.

Nem todos os sistemas precisam de um *Controlador de Aplicação* (360). Eles são úteis se o seu sistema tiver muita lógica relacionada à ordem das telas e à navegação entre elas. Eles são úteis também se você não tiver um mapeamento simples entre suas páginas e as ações no domínio. No entanto, se as telas puderem ser vistas em praticamente qualquer ordem, você provavelmente terá pouca necessidade de um *Controlador de Aplicação* (360). Um bom teste é este: se a máquina estiver no controle do fluxo de telas, você precisa de um *Controlador de Aplicações* (360); se o usuário estiver no controle, você não precisa.

Padrões de Vista

No lado da vista há três padrões a se considerar: *Vista de Transformação* (343), *Vista Modelo* (333) e *Vista em Duas Etapas* (370). Esses, essencialmente, dão origem a duas escolhas: usar *Vista de Transformação* (343) ou *Vista Modelo* (333) e, em ambos os casos, se eles usam um único estágio ou uma *Vista em Duas Etapas* (370). Os padrões básicos para *Vista de Transformação* (343) e *Vista Modelo* (333) são um único estágio. A *Vista em Duas Etapas* (370) é uma variação que você pode aplicar em ambos os casos.

Começarei com a escolha entre a *Vista de Transformação* (343) e a *Vista Modelo* (333). A *Vista Modelo* (333) lhe permite escrever a apresentação na estrutura da página e embutir marcadores na página para indicar onde é necessário o conteúdo dinâmico. Algumas plataformas populares são baseadas neste padrão, muitas das quais são tecnologias de páginas servidoras (ASP, JSP, PHP) que lhe permitem colocar uma linguagem de programação completa na página. Isso claramente fornece muito poder e flexibilidade, mas, infelizmente, também leva a código bastante confuso e difícil de manter. Como conseqüência, se você usar tecnologia de páginas servidoras, deve ser muito disciplinado para manter a lógica da programação fora da estrutura da página, muitas vezes usando um objeto auxiliar.

A *Vista de Transformação* (343) usa um estilo de programação de transformação. O exemplo habitual é XSLT. Isso pode ser muito eficaz se você estiver trabalhando com dados do domínio que estejam no formato XML ou que possam ser facilmente convertidos para XML. Um controle de entrada pega a folha de estilo XSLT apropriada e a aplica ao XML montado a partir do modelo.

Se você usa roteiros procedurais como sua vista, você pode escrever o código em ambos os estilos, *Vista de Transformação* (343) ou *Vista Modelo* (333), ou mesmo em alguma mistura interessante dos dois. Percebi que a maioria dos roteiros segue um destes dois padrões como sua forma principal.

A segunda decisão é sobre usar um estágio único ou usar a *Vista em Duas Etapas* (370). Uma vista de um único estágio, na maioria das vezes, tem um componente de vista para cada tela na aplicação. A vista pega dados orientados ao domínio e os representa em HTML. Digo "na maioria das vezes" porque telas lógicas similares podem compartilhar visões. Mesmo assim, na maior parte do tempo, você pode pensar em uma vista por tela.

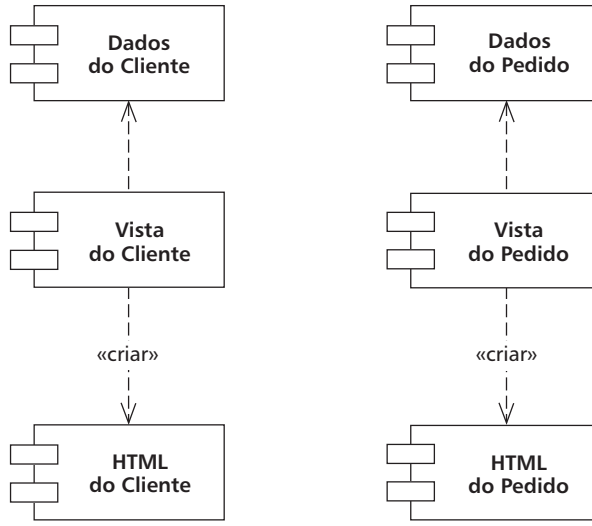


Figura 4.2 Uma vista de estágio único.

Uma vista em dois estágios (Figura 4.3) divide esse este processo em dois estágios, produzindo uma tela lógica a partir dos dados do domínio e então os representando em HTML. Há uma vista de primeiro estágio para cada tela mas apenas uma vista de segundo estágio para a aplicação inteira.

A vantagem da *Vista em Duas Etapas* (370) é que ela coloca a decisão de qual HTML usar em um único local. Isto facilita as alterações globais no código HTML, uma vez que só há um objeto para alterar se quisermos alterar todas as telas no *site*. É claro que você só obtém esta vantagem se sua apresentação lógica permanecer a mesma, de modo que isto funciona melhor com *sites* onde telas diferentes usam o mesmo *layout* básico. Será difícil sugerir uma boa estrutura de tela lógica em *sites* com *design* elaborado.

A *Vista em Duas Etapas* (370) funciona ainda melhor se você tiver uma aplicação Web em que os serviços estejam sendo usados por múltiplos clientes, com diferentes interfaces com o usuário, como diversas companhias aéreas usando o mesmo sistema de reservas. Dentro dos limites da tela lógica, cada interface com o usuário pode ter uma aparência diferente usando um segundo estágio diferente. De modo semelhante você pode usar uma *Vista em Duas Etapas* (370) para lidar com diferentes dispositivos de saída, com segundos estágios separados para um navegador Web normal e para um palmtop. Mais uma vez, a limitação é que você tem de fazer os dois compartilharem uma mesma tela lógica, o que pode não ser possível se as interfaces de usuário forem muito diferentes, como em um navegador e um telefone celular.

Padrões de Controlador de Entrada

Há dois padrões para o controlador de entrada. O mais comum é existir um objeto Controlador de entrada para cada página no seu *site*. No caso mais simples, este *Controlador de Página* (318) pode ser a própria página servidora, combinando os papéis de vista e controlador de entrada. Em muitas implementações separar o controlador de

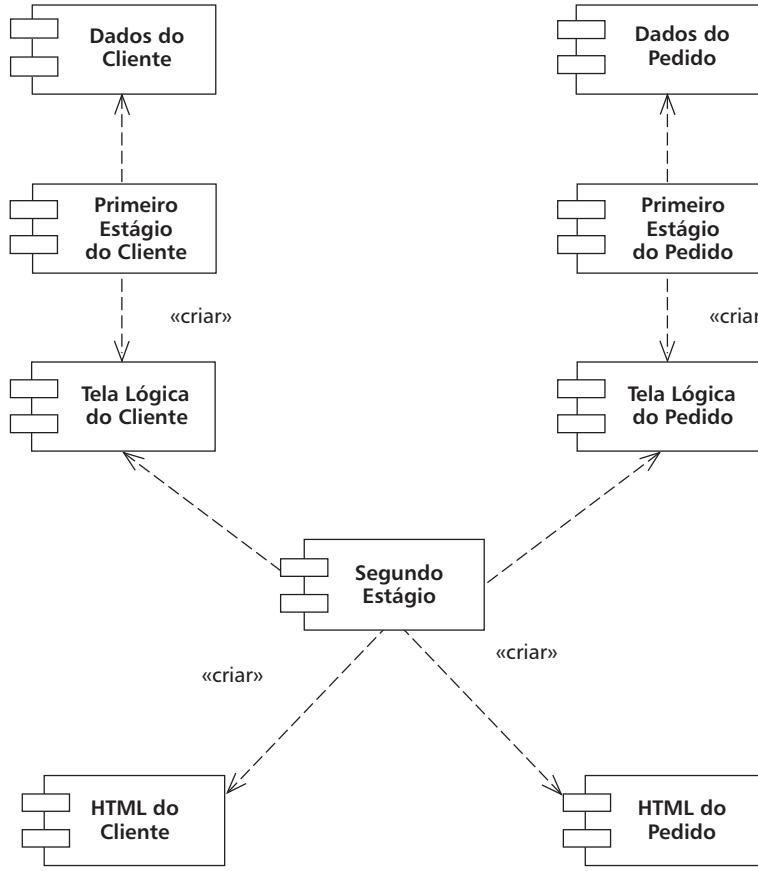


Figura 4.3 Uma vista em dois estágios.

entrada em um objeto separado torna as coisas mais fáceis. O controlador de entrada pode então criar modelos apropriados para executar o processamento e instanciar uma vista para retornar o resultado. Freqüentemente, você perceberá que não há exatamente um relacionamento um-para-um entre *Controladores de Página* (318) e vistas. Um pensamento mais preciso é que você tem um *Controlador de Página* (318) para cada ação, em que uma ação é um botão ou *link*. Na maior parte do tempo as ações correspondem a páginas, mas ocasionalmente isso não acontece – tal como um *link* que pode levar a diferentes páginas, dependendo de alguma condição.

Em qualquer controlador de entrada há duas responsabilidades – tratar a requisição HTTP e decidir o que fazer com ela – e muitas vezes faz sentido separá-las. Uma página servidora pode tratar a requisição, delegando a um objeto auxiliar separado a decisão do que fazer com ela. O *Controlador Frontal* (328) aprofunda esta separação tendo apenas um objeto para lidar com todas as solicitações. Esse objeto único interpreta a URL para descobrir que tipo de requisição ele está tratando e cria um objeto separado para processá-la. Dessa forma você pode centralizar todo o tratamento HTTP em um único objeto, evitando a necessidade de reconfigurar o servidor Web toda vez que alterar a estrutura de ação do *site*.

Leitura Adicional

A maior parte dos livros sobre tecnologias de servidores Web fornece um capítulo ou dois sobre bons projetos de servidores, embora estes estejam freqüentemente imersos em descrições tecnológicas. Uma discussão excelente de projeto Web em Java é o Capítulo 9 de [Brown *et al.*]. A melhor fonte para outros padrões é [Alur *et al.*]; a maior parte destes padrões pode ser usada fora do ambiente em situações não Java. Roubei a terminologia de separar controladores de aplicação e de entrada de [Knight and Dai].

Capítulo 5

Concorrência

por Martin Fowler e David Rice

A concorrência é um dos aspectos mais traiçoeiros do desenvolvimento de *software*. Sempre que você tiver múltiplos processos ou *threads* manipulando os mesmos dados, depara-se com problemas de concorrência. Pensar a respeito de concorrência já é difícil, uma vez que não é fácil enumerar os possíveis cenários que podem lhe trazer problemas. Não importa o que você faça, sempre parece haver algo que você esqueceu. Além disso, a concorrência é difícil de testar. Somos fãs de um grande número de testes automatizados atuando como base para o desenvolvimento de *software*, mas é difícil obter testes que nos dêem a segurança de que precisamos em relação a problemas de concorrência.

Uma das grandes ironias do desenvolvimento de aplicações corporativas é que poucos ramos do desenvolvimento de *software* estão usando a concorrência, e menos ainda se preocupando com ela. O motivo pelo qual desenvolvedores deste tipo de aplicação podem ter esta visão ingênua de concorrência são os gerenciadores de transação. As transações fornecem um *framework* que ajuda a evitar muitos dos aspectos mais traiçoeiros da concorrência em uma aplicação corporativa. Enquanto você executar toda sua manipulação de dados dentro de uma transação, nada de muito ruim irá lhe acontecer.

Infelizmente, isso não significa que possamos ignorar completamente problemas de concorrência, pela razão básica de que muitas das interações com um sistema não podem ser colocadas dentro de uma única transação com banco de dados. Isso nos força a gerenciar a concorrência em situações em que os dados atravessam várias transações. O termo que usamos é **concorrência *offline***, ou seja, controle de concorrência para dados que sejam manipulados durante múltiplas transações com bancos de dados.

A segunda área em que a concorrência mostra seu lado negativo para os desenvolvedores corporativos são os servidores de aplicações – suportar múltiplas *threads* em um servidor de aplicações. Gastamos muito menos tempo nisso, porque isso é

muito mais simples de lidar. De fato, você pode usar plataformas de servidores que tomem conta de grande parte disso para você.

Infelizmente, para entender essas questões, você precisa compreender pelo menos alguns dos conceitos gerais de concorrência. Assim, começamos este capítulo examinando tais questões. Não temos a pretensão de que este capítulo seja um tratamento geral da concorrência no desenvolvimento de *software* – para tanto precisaríamos de pelo menos um livro inteiro. O que este capítulo faz é introduzir questões de concorrência para o desenvolvimento de aplicações corporativas. Assim que tivermos feito isso, introduziremos os padrões para lidar com concorrência *offline* e diremos breves palavras sobre concorrência em servidores de aplicações.

Em grande parte deste capítulo ilustraremos as idéias com exemplos de uma área com a qual esperamos que você esteja familiarizado – os sistemas de controle de código fonte usados por equipes para coordenar alterações em um código base. Fazemos isto porque eles são relativamente fáceis de entender e também bastante conhecidos. Afinal, se você não estiver familiarizado com sistemas de controle de código fonte, você não deve estar desenvolvendo aplicações corporativas.

Problemas de Concorrência

Começaremos examinando os problemas básicos da concorrência. Chamamos de básicos porque são os problemas fundamentais que os sistemas de controle de concorrência devem tentar impedir. Eles não são os únicos problemas de concorrência, porque os mecanismos de controle, muitas vezes, criam um novo conjunto de problemas nas suas soluções! Todavia, eles enfocam o ponto básico do controle de concorrência.

Atualizações perdidas são a idéia mais simples de entender. Suponha que Martin edite um arquivo para fazer algumas alterações no método `verificarConcorrência` – uma tarefa que leva alguns minutos. Enquanto ele está fazendo isso, David altera o método `atualizarParâmetroImportante` no mesmo arquivo. David começa e termina sua alteração muito rapidamente, tão rapidamente que, embora tenha começado depois de Martin, termina antes deste. Isso é desastroso. Quando Martin leu o arquivo ele não incluía a atualização do David, de modo que quando Martin gravar o arquivo ele gravará por cima da versão que David atualizou e essas atualizações serão perdidas para sempre.

Uma **leitura inconsistente** ocorre quando você lê duas informações que são corretas, mas não ao mesmo tempo. Suponha que Martin deseja saber quantas classes estão no pacote de concorrência, o qual contém dois subpacotes para bloqueio e multifase. Martin olha o pacote de bloqueio e vê sete classes. Neste momento ele recebe uma ligação do Roy a respeito de alguma questão confusa. Enquanto Martin está falando ao telefone, David termina de consertar aquela falha ridícula no código de bloqueio em quatro fases e adiciona duas classes ao pacote de bloqueio e três classes às cinco que estavam no pacote de multifases. Após o término de seu telefonema, Martin olha o pacote multifase para ver quantas classes há e vê oito, produzindo um total geral de quinze.

Infelizmente, quinze classes nunca foi a resposta correta. A resposta correta era doze antes da atualização do David e dezessete depois. Cada uma destas respostas teria sido correta, mesmo se não fosse a resposta correta correntemente, mas quinze nunca foi correto. Este problema é chamado de leitura inconsistente, porque os dados que Martin leu estavam inconsistentes.

Ambos os problemas causam uma falha de **correção** (ou segurança) e resultam em comportamento incorreto que não teria ocorrido sem duas pessoas tentando trabalhar com os mesmos dados ao mesmo tempo. Entretanto, se a correção fosse a única questão, esses problemas não seriam tão sérios. Afinal, podemos arranjar as coisas de modo que apenas um de nós possa trabalhar nos dados de cada vez. Embora isso ajude a correção, reduz a capacidade de fazer coisas concorrentemente. O problema essencial de qualquer programação concorrente é que não basta se preocupar com a correção, você também tem que se preocupar com a vivacidade do sistema: quanta atividade concorrente pode ser executada. Muitas vezes as pessoas precisam sacrificar um pouco de correção para ganhar mais vivacidade, dependendo da seriedade e da probabilidade de falhas e da necessidade de as pessoas trabalharem nos seus dados concorrentemente.

Estes não são todos os problemas que você tem com concorrência, mas consideramos estes como sendo os básicos. Para resolvê-los, usamos diversos mecanismos de controle. Infelizmente, não existe comida de graça. As soluções introduzem seus próprios problemas, embora estes problemas sejam menos sérios que os originais. Ainda assim, isso nos traz uma questão importante: se você puder tolerar os problemas, pode evitar qualquer forma de controle de concorrência; é raro, mas ocasionalmente você encontra circunstâncias que o permitam.

Contextos de Execução

Toda vez que ocorre processamento em um sistema, ele ocorre em algum contexto e normalmente em mais de um. Não há terminologia padrão para contextos de execução, de modo que aqui iremos definir as que estamos pressupondo neste livro.

Da perspectiva da interação com o mundo externo, dois contextos importantes são a solicitação e a sessão. Uma **solicitação** corresponde a uma única chamada do mundo externo sobre a qual o *software* trabalha e para a qual opcionalmente retorna uma resposta. Durante uma solicitação, o processamento ocorre amplamente no lado do servidor e pressupõe-se que o cliente deva esperar por uma resposta. Alguns protocolos permitem que o cliente interrompa uma solicitação antes de obter uma resposta, mas isso é bastante raro. Com maior freqüência um cliente pode enviar outra solicitação que pode interferir com a que foi recém-enviada. Assim, um cliente pode solicitar permissão para colocar um pedido e então enviar uma solicitação separada para cancelar aquele pedido. Do ponto de vista do cliente, as duas solicitações podem estar obviamente conectadas, mas, dependendo do seu protocolo, isso pode não ser tão óbvio para o servidor.

Uma **sessão** é uma interação longa entre um cliente e um servidor. Pode consistir em uma única solicitação, mas, mais comumente, ela consiste em uma série de solicitações que o usuário considera como uma seqüência lógica consistente. É comum uma sessão começar com um usuário se logando e executando algum trabalho que pode envolver algumas pesquisas e uma ou mais transações de negócio (a serem discutidas em breve). No final da sessão, o usuário se desconecta ou pode simplesmente ir embora e assumir que o sistema interprete isso como uma desconexão.

O *software* servidor em uma aplicação corporativa vê tanto solicitações quanto sessões de dois ângulos, como o servidor do cliente e como o cliente para outros sistemas. Assim, você muitas vezes verá diversas sessões: sessões HTTP do cliente e sessões de banco de dados com diversos bancos de dados.

Dois termos muito importantes de sistemas operacionais são os processos e os *threads*. O **processo** é um contexto de execução normalmente pesado que fornece bastante isolamento para os dados internos sobre os quais trabalha. A ***thread*** é um agente ativo de menor peso que é configurado de modo que diversas ***threads*** possam operar em um único processo. As pessoas gostam de *threads* porque elas suportam múltiplas solicitações dentro de um único processo – o que é uma boa utilização de recursos. As *threads* entretanto normalmente compartilham memória, e tal compartilhamento leva a problemas de concorrêncìa. Alguns ambientes permitem que você controle que dados uma *thread* pode acessar, permitindo que você tenha ***threads* isoladas** que não compartilham memória.

A dificuldade com contextos de execução vem quando eles se não alinham tão bem quanto gostaríamos. Na teoria, cada sessão teria um relacionamento exclusivo com um processo por todo seu ciclo de vida. Uma vez que processos são apropriadamente isolados entre si, isso ajudaria a reduzir conflitos de concorrência. Atualmente não conhecemos quaisquer ferramentas de servidor que lhe permitam trabalhar assim. Uma alternativa próxima é começar um novo processo para cada solicitação, que era o modo comum nos primeiros sistemas Web Perl. As pessoas tendem a evitar isso agora porque iniciar processos retém muitos recursos, mas é muito comum que sistemas façam um processo lidar com apenas uma solicitação de cada vez – e isso pode evitar muitas dores de cabeça causadas por concorrência.

Quando você estiver lidando com bancos de dados, há outro contexto importante – uma **transação**. As transações juntam diversas solicitações que o cliente quer que sejam tratadas como se fossem uma única. Elas podem ocorrer de uma aplicação para o banco de dados (uma transação de sistema) ou do usuário para uma aplicação (uma transação de negócio). Iremos nos aprofundar nesses termos mais adiante.

Isolamento e Imutabilidade

Os problemas de concorrência têm ocorrido há um certo tempo, e as pessoas que lidam com *software* têm encontrado diversas soluções. Para aplicações corporativas, duas soluções são particularmente importantes: isolamento e imutabilidade.

Problemas de concorrência ocorrem quando mais de um agente ativo, como um processo ou uma *thread*, tem acesso a uma mesma parte dos dados. Uma maneira de lidar com isso é por meio do isolamento: particionar os dados de modo que qualquer parte deles só possa ser acessada por um agente ativo. Os processos trabalham assim na memória de sistemas operacionais: o sistema operacional aloca memória exclusivamente para um único processo, e apenas esse processo pode ler ou gravar os dados ligados a ele. De modo similar, você encontra bloqueios de arquivo em muitas aplicações de produtividade populares. Se Martin abrir um arquivo, ninguém mais pode abri-lo. Eles podem abrir uma cópia apenas para leitura do arquivo no estado em que este se encontrava no momento em que Martin o abriu, mas não podem alterá-lo e não conseguem ver o arquivo entre suas alterações.

O isolamento é uma técnica vital porque reduz a chance de erros. Temos visto com muita freqüência pessoas se envolverem em problemas, porque usam uma técnica que força todos a se preocuparem com concorrência durante o tempo todo. Com isolamento, você deixa as coisas de um modo que os programas entram em uma área isolada, dentro da qual você não tem que se preocupar com concorrência. Um bom projeto de concorrência é, desta forma, encontrar modos de criar tais áreas e garantir que tanta programação quanto possível seja feita em uma delas.

Você só terá problemas de concorrência se os dados que estiver compartilhando puderem ser modificados. Dessa forma, uma maneira de evitar conflitos de concorrência é reconhecer dados **imutáveis**. Obviamente, não podemos tornar todos os dados imutáveis, já que o objetivo de muitos sistemas é a modificação de dados. Contudo, identificando alguns dados como imutáveis, ou pelo menos imutáveis quase todo o tempo, podemos relaxar nossas preocupações com concorrência em relação a eles e compartilhá-los amplamente. Outra opção é separar aplicações que apenas lêem dados e fazê-las usar fontes de dados copiados, nas quais podemos relaxar todos os controles de concorrência.

Controles de Concorrência Otimista e Pessimista

O que acontece quando temos dados mutáveis que não podemos isolar? De um modo geral, há duas formas de controle de concorrência que podemos usar: otimista e pessimista.

Suponhamos que Martin e David queiram editar o arquivo Clientes ao mesmo tempo. Com **bloqueio otimista**, ambos podem fazer uma cópia do arquivo e editá-la livremente. Se David for o primeiro a terminar, ele pode gravar seu trabalho sem problema. O controle de concorrência entra quando Martin tentar perpetrar suas alterações. Nesse momento, o sistema de controle de código fonte detecta um conflito entre as alterações de Martin e as de David. A confirmação de Martin é rejeitada e é responsabilidade dele descobrir como lidar com a situação. Com **bloqueio pessimista**, qualquer pessoa que pegue o arquivo antes evita que outra possa editá-lo. Dessa maneira, se Martin for o primeiro a pegar o arquivo, David não poderá trabalhar com esse arquivo até que Martin tenha terminado e perpetrado suas alterações.

Uma boa maneira de pensar sobre isso é que um bloqueio otimista se preocupa com a detecção de conflitos enquanto que um bloqueio pessimista se preocupa com a prevenção de conflitos. Os sistemas de controle de código fonte podem usar ambos os tipos, embora atualmente mais desenvolvedores de código fonte prefiram trabalhar com bloqueios otimistas. (Há um argumento razoável, segundo o qual o bloqueio otimista não é realmente um bloqueio, mas consideramos a terminologia bastante conveniente e difundida demais para ser ignorada.)

Ambas as abordagens têm prós e contras. O problema com o bloqueio pessimista é que ele diminui a concorrência. Enquanto Martin está trabalhando em um arquivo ele o bloqueia, de modo que as outras pessoas têm de esperar. Se você já tiver trabalhado com sistemas de controle de código fonte pessimistas, sabe o quão frustrante isso pode ser. Com dados corporativos, é freqüentemente, pior porque, se alguém estiver editando dados, ninguém mais pode lê-los, muito menos editá-los.

Os bloqueios otimistas permitem às pessoas fazerem mais progresso, porque o bloqueio só acontece durante a confirmação. O problema com eles é o que acontece quando você tem um conflito. Basicamente, todas as pessoas após a confirmação do David têm que verificar a versão do arquivo que David gravou, descobrir como juntar suas alterações com as do David e então gravar a nova versão. Com código fonte isso não é muito difícil. De fato, em muitos casos o sistema de controle de código fonte pode fazer a junção automaticamente para você e, mesmo quando ele não puder fazê-lo, ferramentas podem tornar muito mais fácil ver as diferenças. Contudo, dados de negócio são normalmente difíceis demais de sofrerem junção automaticamente, de modo que freqüentemente o que você poderá fazer é jogar tudo fora e começar de novo.

A essência da escolha entre bloqueios otimista e pessimista é a freqüência e severidade dos conflitos. Se estes forem suficientemente raros ou se as conseqüências não forem graves, você deve normalmente escolher bloqueios otimistas porque eles lhe dão maior concorrência e são geralmente mais fáceis de implementar. Entretanto, se os resultados de um conflito forem dolorosos para os usuários, você precisará usar uma técnica de bloqueio pessimista.

Nenhuma dessas abordagens está livre de problemas. Usando-as, você pode facilmente introduzir problemas que causem quase tanto prejuízo quanto os problemas básicos de concorrência que você está tentando resolver. Deixaremos uma discussão detalhada dessas ramificações para um livro apropriado sobre concorrência, mas aqui estão alguns fatos importantes a serem lembrados.

Evitando Leituras Inconsistentes

Considere esta situação. Martin edita a classe Clientes, a qual efetua chamadas para a classe Pedidos. Enquanto isso, David edita a classe Pedidos e altera a interface. David compila e grava. Martin então compila e grava. Agora o código compartilhado está estragado porque Martin não percebeu que a classe Pedidos foi alterada. Alguns sistemas de controle de código fonte irão localizar esta leitura inconsistente, mas outros requerem algum tipo de disciplina manual para impor consistência, como atualizar seus arquivos antes de gravá-los.

Na essência, este é o problema da inconsistência de leitura e, muitas, vezes é fácil não vê-lo, porque a maioria das pessoas tende a focar as atualizações perdidas como o problema essencial na concorrência. Os bloqueios pessimistas têm um jeito muito usado de lidar com este problema por meio de bloqueios de leitura e de gravação. Para ler dados você precisa de um bloqueio de leitura (ou bloqueio compartilhado). Para gravar dados você precisa de um bloqueio de gravação (ou bloqueio exclusivo). Muitas pessoas podem ter bloqueios de leitura sobre um mesmo dado ao mesmo tempo, mas, se alguém tiver um bloqueio de leitura, ninguém pode obter um bloqueio de gravação. Inversamente, assim que alguém tiver um bloqueio de gravação, ninguém mais poderá ter qualquer tipo de bloqueio. Com esse sistema, você consegue evitar leituras inconsistentes com bloqueios pessimistas.

Os bloqueios otimistas normalmente baseiam sua detecção de conflitos em algum tipo de marcador de versão sobre os dados. Este pode ser um *timestamp* ou um contador seqüencial. Para detectar atualizações perdidas, o sistema compara o marcador de versão da sua atualização com o marcador de versão dos dados compartilhados. Se eles forem o mesmo, o sistema permite a atualização e atualiza o marcador de versão.

Detectar uma leitura inconsistente é semelhante: neste caso cada parte dos dados que foi lida precisa ter seu marcador de versão comparado com os dados compartilhados. Qualquer diferença indica um conflito.

Controlar o acesso a cada parte dos dados que é lida, muitas vezes, causa problemas desnecessários devido a conflitos ou esperas dos dados que não importam tanto. Você pode reduzir esse fardo separando dados que *usou* dos dados que meramente leu. Em uma lista de escolha de produtos, não importa se um novo produto aparecer após você começar suas alterações. Entretanto, uma lista de pagamentos que você esteja resumindo para uma conta pode ser mais importante. A dificuldade é que isso requer uma análise cuidadosa do motivo pelo qual é usado. Um código postal em um endereço de um cliente pode parecer inócuo, porém se um cálculo de taxa for basea-

do em onde alguém mora, esse endereço tem que ser controlado quanto à concorrência. Como você pode ver, descobrir o que é e o que não é preciso controlar é um exercício intrincado, não importa qual forma de controle de concorrência você use.

Outra maneira de lidar com problemas de leitura inconsistente é usar **Leituras Temporais**. Elas assinalam cada leitura de dados com algum tipo de *timestamp* ou rótulo imutável, e o banco de dados retorna os dados como eles estavam de acordo com aquele tempo ou rótulo. Muito poucos bancos de dados têm algo assim, mas os desenvolvedores muitas vezes se deparam com isso em sistemas de controle de código fonte. O problema é que a fonte de dados precisa fornecer uma história temporal completa das alterações, o que gasta tempo e espaço para processar. Isso é razoável para código fonte, porém é mais difícil e custoso para bancos de dados. Você pode precisar fornecer essa capacidade para áreas específicas da sua lógica de domínio: veja em [Snodgrass] e [Fowler TP] idéias de como fazer isso.

Deadlocks

Um problema particular com técnicas pessimistas é o ***deadlock***. Suponha que Martin comece a editar o arquivo Clientes, e David comece a editar o arquivo Pedidos. David percebe que, para completar sua tarefa, precisa editar o arquivo Clientes também, porém Martin tem o bloqueio deste arquivo, de forma que ele tem que esperar. Martin então percebe que tem que editar o arquivo Pedidos, o qual foi bloqueado por David. Eles estão agora em *deadlock* – nenhum dos dois pode avançar até que o outro termine. Descrito assim, os *deadlocks* parecem fáceis de evitar, mas eles podem ocorrer com muitas pessoas envolvidas em uma cadeia complexa, e isso os torna mais traiçoeiros.

Há várias técnicas usadas para lidar com *deadlocks*. Uma é ter *software* que consiga detectar um *deadlock* quando ele ocorrer. Nesse caso você escolhe uma **vítima**, que tem que jogar fora seu trabalho e seus bloqueios de modo que os outros possam avançar. A detecção de *deadlocks* é muito difícil e custosa para as vítimas. Uma abordagem similar é dar a cada bloqueio um limite de tempo. Assim que você chegar nesse limite, perde seus bloqueios e seu trabalho – basicamente se tornando uma vítima. Mecanismos de tempo limite são mais fáceis de implementar do que um mecanismo de detecção de *deadlocks*, porém, se alguém segurar um bloqueio por certo tempo, algumas pessoas serão vitimadas quando não houver realmente *deadlocks*.

Os tempos limite e detecção de *deadlocks* lidam com um *deadlock* quando ele ocorre. Outras abordagens tentam evitar que os *deadlocks* ocorram. *Deadlocks* ocorrem basicamente quando pessoas que já têm bloqueios tentam obter mais (ou passar de bloqueios de leitura para de gravação.) Assim, um modo de preveni-los é forçar as pessoas a obter todos os seus bloqueios de uma só vez, no início do seu trabalho e, então, evitar que elas obtenham mais.

Você pode forçar uma ordem pela qual todos obtenham seus bloqueios. Um exemplo seria ser obter sempre os bloqueios sobre os arquivos em ordem alfabética. Desta maneira, já que David tinha um bloqueio sobre o arquivo Pedidos, ele não pode tentar obter um bloqueio sobre o arquivo Clientes, porque este é anterior na seqüência. Nesse momento, ele basicamente se torna uma vítima.

Você também pode fazer com que, se Martin tentar obter um bloqueio e David já tiver um, Martin automaticamente se torne uma vítima. É uma técnica drástica, mas é simples de implementar, e, em muitos casos, tal esquema funciona bem.

Se você for muito conservador, pode usar múltiplos esquemas. Por exemplo, você força todos a obter todos os seus bloqueios no início, mas acrescenta um tempo limite para o caso de algo dar errado. Isso pode parecer como usar um cinto e suspensórios, mas tal conservadorismo é muitas vezes sábio com relação a *deadlocks*, porque estes são coisas desagradáveis que facilmente dão errado.

É muito fácil pensar que você tem um esquema à prova de *deadlocks* e depois descobrir alguma cadeia de eventos que não considerou. O resultado disso é que preferimos esquemas muito simples e conservadores para o desenvolvimento de aplicações corporativas. Elas podem causar vítimas desnecessárias, mas é geralmente muito melhor do que as conseqüências de esquecer algum cenário de *deadlock.*

Transações

A principal ferramenta para lidar com concorrência em aplicações corporativas é a transação. A palavra "transação" muitas vezes traz à mente uma troca de dinheiro ou de bens. Ir até um caixa eletrônico, digitar seu código e retirar dinheiro é uma transação. Pagar $3 de taxa na Golden Gate Bridge é uma transação. Comprar uma cerveja no bar local é uma transação.

Olhar para negócios financeiros típicos como esses fornece uma boa definição para o termo. Primeiro, uma transação é uma seqüência limitada de trabalho, com pontos de início e fim bem definidos. Uma transação em um caixa eletrônico começa quando o cartão é inserido e termina quando o dinheiro é entregue ou uma falta de fundos é descoberta. Em segundo lugar, todos os recursos participantes estão em estado consistente tanto quando a transação começa como quando ela termina. Um homem comprando uma cerveja tem algum dinheiro a menos na sua carteira, mas tem uma bela cerveja na sua frente. O somatório dos seus bens não mudou. O mesmo vale para o bar – servir cerveja de graça não seria uma maneira de conduzir um negócio.

Além disso, cada transação deve ser completada em uma base de tudo ou nada. O banco não pode subtrair do saldo de uma conta a não ser que o caixa eletrônico realmente entregue o dinheiro. Embora o elemento humano pudesse tornar esta última propriedade opcional durante as transações acima, não há razão para que o *software* não possa criar uma garantia.

ACID

As transações de *software* são freqüentemente descritas em termos de propriedades ACID:

- **Atomicidade:** Cada passo na seqüência de ações executadas no contexto de uma transação deve ser completado com sucesso ou então todo o trabalho deve ser desfeito. A conclusão parcial não é um conceito transacional. Assim, se Martin estiver transferindo algum dinheiro de sua conta poupança para sua conta corrente, e o servidor sofrer uma pane após ter retirado o dinheiro da conta poupança, o sistema se comporta com se nunca tivesse executado essa retirada. Confirmar (*committing*) diz que ambas as coisas ocorreram. Desfazer (*roll back*) significa que nenhuma aconteceu. Tem que ser as duas coisas ou nenhuma.

- **Consistência:** Os recursos de um sistema devem estar em um estado consistente e não corrompido tanto no início quanto no final de uma transação.
- **Isolamento:** O resultado de uma transação individual não deve ser visível para outras transações abertas até que a transação confirme sua execução com sucesso.
- **Durabilidade:** Qualquer resultado de uma transação confirmada deve ser permanente. Isso é traduzido para "deve sobreviver a qualquer tipo de falha".

Recursos Transacionais

A maior parte das aplicações corporativas se depara com transações no uso de bancos de dados. Porém há muitas outras coisas que podem ser controladas pelo uso de transações, como filas de mensagens, impressoras e ATMs. O resultado disso é que discussões técnicas sobre transações usam o termo "recurso transacional" para qualquer coisa que seja transacional – ou seja, que use transações para controle de concorrência. "Recurso transacional" é um termo um pouco longo, então usamos apenas "banco de dados", já que esse é o caso mais comum. Contudo, quando dizemos "banco de dados", o mesmo se aplica para qualquer outro recurso transacional.

Para obter a maior transferência de dados possível, sistemas transacionais modernos são projetados para manter as transações tão curtas quanto possível. Como conseqüência disso, o conselho geral é nunca fazer uma transação durar por múltiplas solicitações. Uma transação que dura por múltiplas solicitações é geralmente conhecida como uma transação longa.

Por essa razão, uma abordagem comum é começar uma transação no início de uma solicitação e completá-la ao final. Esta **transação de solicitação** é um ótimo modelo simples, e vários ambientes tornam fácil fazê-la declarativamente, simplesmente identificando métodos como transacionais.

Uma variação é abrir uma transação o mais tarde possível. Com uma **transação tardia**, você pode executar todas as leituras fora dela e apenas abri-la quando executar as atualizações. Isso tem a vantagem de minimizar o tempo gasto em uma transação. Se houver um tempo longo entre a abertura da transação e a primeira escrita, isso pode melhorar a vivacidade. Entretanto, significa que você não tem nenhum controle de concorrência até que comece a transação, o que lhe deixa sujeito a leituras inconsistentes. Como conseqüência, normalmente não vale a pena fazer isso, a menos que você tenha uma disputa muito grande ou o esteja fazendo devido a transações de negócio que atravessam múltiplas solicitações (o que é o próximo tópico).

Quando você usa transações, precisa de alguma forma estar ciente do que exatamente está bloqueado. Para muitas ações do banco de dados, o sistema de transações bloqueia as linhas envolvidas, o que permite que múltiplas transações acessem a mesma tabela. Entretanto, se uma transação bloquear muitas linhas de uma tabela, o banco de dados tem mais bloqueios do que consegue lidar e amplia o bloqueio para a tabela inteira – bloqueando outras transações. Esta **ampliação de bloqueio** pode ter efeitos sérios sobre a concorrência, e é exatamente por isso que você não deve ter algumas tabelas "objeto" para dados no nível *Camada Supertipo* (444) do domínio. Tal tabela é uma forte candidata à ampliação do bloqueio, e bloquear esta tabela impede o acesso de todos os outros ao banco de dados.

Reduzindo o Isolamento Transacional para Aumentar a Vivacidade

É comum restringir a proteção completa das transações, de modo que você possa obter mais vivacidade. Isso é particularmente o caso ao lidar com isolamento. Se você tiver isolamento integral, obtém transações serializáveis. As transações são **serializáveis** se puderem ser executadas concorrentemente e você obtiver um resultado igual ao que obteria da execução serial dessas transações. Assim, se pegarmos nosso exemplo anterior do Martin contando seus arquivos, a capacidade de serializar garante que ele obtenha um resultado que corresponde a concluir sua transação completamente antes que a transação do David comece (doze) ou completamente após ela terminar (dezessete). Capacidade de serializar não pode garantir o resultado, como neste caso, mas pelo menos garante um resultado correto.

A maior parte dos sistemas transacionais usa o padrão SQL, o qual define quatro níveis de isolamento. O serializável é o nível mais forte, e cada nível abaixo permite que um tipo particular de leitura inconsistente entre em cena. Nós os exploraremos com o exemplo do Martin contando arquivos enquanto David os modifica. Há dois pacotes: bloqueio e multifase. Antes do David atualizá-los há sete arquivos no pacote de bloqueio e cinco no de multifase. Após sua atualização, há nove no de bloqueio e oito no multifase. Martin olha o pacote de bloqueio e David então atualiza ambos. A seguir, Martin olha o pacote multifase.

Se o nível de isolamento for serializável, o sistema garante que a resposta do Martin seja doze ou dezessete, ambas corretas. A capacidade de serializar não pode garantir que toda execução deste cenário dê o mesmo resultado, mas este sempre obtém o número antes da atualização do David ou o número depois.

O primeiro nível de isolamento abaixo da serialização é a **leitura repetível**, que permite **fantasmas**. Os fantasmas ocorrem quando você adiciona alguns elementos a uma coleção, e o leitor vê apenas alguns deles. O caso aqui é que Martin vê os arquivos no pacote de bloqueio e enxerga sete. David então confirma sua transação, após o que, Martin vê o pacote multifase e enxerga oito. Conseqüentemente, Martin obtém um resultado incorreto. Os fantasmas ocorrem porque eles são válidos para uma parte da transação de Martin, mas não para toda ela, e eles são sempre coisas que foram inseridas.

A seguir, na lista, está o nível de isolamento **leitura confirmada**, que permite **leituras não repetíveis**. Imagine que Martin olhe um total em vez dos arquivos reais. Uma leitura não-repetível permite a ele ler um total de sete para bloqueio. A seguir, David confirma a gravação e ele então lê um total de oito para multifase. É chamada leitura não-repetível porque, se Martin fosse reler o total do pacote de bloqueio após David ter confirmado a gravação, ele obteria o novo número nove. Sua leitura original do número sete não pode ser repetida após a atualização de David. É mais fácil para bancos de dados localizar leituras não-repetíveis do que fantasmas, de modo que a leitura repetível lhe dá mais correção do que leituras confirmadas, mas menos concorrência.

O menor nível de isolamento é a **leitura não-confirmada**, que permite **leituras sujas**. Na leitura não-confirmada, você pode ler dados que outra transação ainda não confirmou. Isso causa dois tipos de erros. Martin poderia ver o pacote de bloqueio enquanto David adicionava o primeiro dos seus arquivos, mas antes que ele acrescentasse o segundo. O resultado disso é que ele vê oito arquivos no pacote de bloqueio. O segundo tipo de erro ocorre se David adiciona seus arquivos, mas desfaz sua transação – neste caso Martin vê arquivos que nunca estiveram realmente lá.

A Tabela 5.1 lista os erros de leitura causados por cada nível de isolamento.

Para estar seguro da correção, você deve sempre usar o nível de isolamento serializável. O problema é que escolher o serializável atrapalha a vivacidade de um sistema, tanto que muitas vezes você tem que reduzir a capacidade de serializar para aumentar a quantidade de dados transferidos. Você tem que decidir que riscos quer tomar e fazer seu próprio balanço entre erros e desempenho.

Você não tem que usar o mesmo nível de isolamento para todas as transações, de modo que deve ver cada transação e decidir como balancear vivacidade *versus* correção.

Transações de Negócio e de Sistema

O que nós falamos até agora, e a maior parte do que as pessoas falam, é o que chamamos de transações de sistema, ou transações suportadas por sistemas de gerenciamento de bancos de dados relacionais (RDBMS) e monitores de transações. Uma transação de banco de dados é um grupo de comandos SQL delimitados por instruções para começar e terminá-lo. Se a quarta declaração na transação resultar em uma violação de restrição de integridade, o banco de dados deve desfazer os efeitos das primeiras três declarações e notificar o solicitante de que a transação falhou. Se todas as quatro declarações tivessem sido completadas com sucesso, todas teriam sido tornadas visíveis para os outros usuários ao mesmo tempo, e não uma de cada vez. Gerenciadores de transações em sistemas RDBMS e em servidores de aplicações são tão comuns que podem não receber tanta atenção. Eles funcionam bem e são bem compreendidos pelos desenvolvedores de aplicações.

Todavia, uma transação de sistema não tem significado para o usuário de um sistema de negócios. Para o usuário de um sistema de banco *online* uma transação consiste em se logar, selecionar uma conta, configurar alguns pagamentos e finalmente clicar no botão OK para pagar as contas. Isso é o que chamamos de uma **transação de negócio**, e parece uma expectativa razoável que ela apresente as mesmas propriedades ACID como uma transação de sistema. Se o usuário cancelar antes de pagar as contas, quaisquer alterações feitas nas telas anteriores devem ser canceladas. Fazer pagamentos não deve resultar em uma alteração de saldo visível para o sistema até que o botão OK seja pressionado.

A resposta óbvia para suportar as propriedades ACID de uma transação de negócios é executar essa transação inteira dentro de uma única transação de sistema. Infelizmente as transações de negócios muitas vezes precisam de diversas solicitações para serem completadas, de modo que usar uma única transação de sistema para implementá-la resulta em uma transação de sistema longa. A maioria dos sistemas de transação não funciona muito eficientemente com transações longas.

Tabela 5.1 Níveis de Isolamento e os Erros de Leitura Inconsistente que Permitem

Nível de Isolamento	Leitura Suja	Leitura Não-Repetível	Fantasmas
Leitura Não-Confirmada	Sim	Sim	Sim
Leitura Confirmada	Não	Sim	Sim
Leitura Repetível	Não	Não	Sim
Serializável	Não	Não	Não

Isso não significa que você nunca deva usar transações longas. Se o seu banco de dados tiver somente necessidades moderadas de concorrência, elas podem funcionar para você. E, se elas funcionarem para você, sugerimos usá-las. Entretanto, a aplicação não será escalável, porque transações longas transformam o banco de dados em um gargalo importante. Além disso, a refatoração de transações longas para curtas é complexa e mal compreendida.

Por este motivo muitas aplicações corporativas não podem arriscar transações longas. Nesse caso, você tem que dividir a transação de negócio em uma série de transações curtas. Isso significa que você terá que dar seu jeito para suportar as propriedades ACID de transações de negócio entre transações de sistema – um problema que chamamos de **concorrência *offline***. As transações de sistema ainda fazem parte da cena. Sempre que transações de negócio interagem com um recurso transacional, como um banco de dados, essa interação irá executar dentro de uma transação de sistema para que se mantenha a integridade desse recurso. Contudo, como você lerá a seguir, não é suficiente conectar uma série de transações de sistema para suportar apropriadamente uma transação de negócio. A aplicação de negócio deve conectar as transações de sistema.

A atomicidade e a durabilidade são as propriedades ACID mais facilmente suportadas em transações de negócios. Ambas são suportadas executando-se a fase de confirmação da transação de negócio, quando o usuário pressiona "Gravar", dentro de uma transação de sistema. Antes que a sessão tente gravar todas as alterações para o conjunto de registros, ela primeiro abre uma transação de sistema. A transação de sistema garante que as alterações sejam gravadas como uma unidade e as torna permanentes. A única parte potencialmente traiçoeira aqui é a manutenção de um conjunto preciso de alterações durante a vida da transação de negócio. Se a aplicação usar um *Modelo de Domínio* (126), uma *Unidade de Trabalho* (187) pode rastrear as alterações com precisão. Colocar lógica de negócio em um *Roteiro de Transação* (120) requer um rastreamento manual das alterações, mas isso provavelmente não é um problema já que o uso de roteiros de transação sugere transações simples de negócios.

A propriedade ACID mais traiçoeira de garantir em transações de negócios é o isolamento. Falhas de isolamento levam a falhas de consistência. A consistência determina que uma transação de negócio não deixe o conjunto de registros em um estado inválido. Dentro de uma única transação, a responsabilidade da aplicação no suporte à consistência é forçar o respeito a todas as regras de negócio disponíveis. Ao longo de múltiplas transações, a responsabilidade da aplicação é assegurar que uma sessão não atropele as alterações de outra sessão, deixando o conjunto de registros no estado inválido de ter perdido o trabalho de um usuário.

Assim como o problema óbvio da colisão de atualizações, há problemas mais sutis de leituras inconsistentes. Quando dados são lidos por diversas transações de sistema, não há garantia de que estarão consistentes. As diferentes leituras podem até introduzir dados na memória que estejam suficientemente inconsistentes para causar falhas na aplicação.

As transações de negócios estão intimamente ligadas a sessões. Na visão do usuário, cada sessão é uma seqüência de transações de negócios (a não ser que eles estejam apenas lendo dados), de modo que normalmente pressupomos que todas as transações de negócio executam em uma única sessão cliente. Embora certamente seja possível projetar um sistema que tenha múltiplas sessões para uma transação de negócio, este é um modo muito bom de se confundir – de maneira que pressupomos que você não irá fazer isso.

Padrões para o Controle de Concorrência *Offline*

Tanto quanto possível, você deve deixar o seu sistema de transações lidar com problemas de concorrência. Tratar controle de concorrência que atravessa várias transações de sistema faz com que você mergulhe nas águas turvas de ter você mesmo que lidar com a concorrência. Essas águas estão cheias de tubarões virtuais, águas-vivas, piranhas e outras criaturas menos amigáveis. Infelizmente, o desacordo entre transações de negócios e de sistema significa que às vezes você tem que suar a camisa. Os padrões que fornecemos são algumas técnicas que achamos úteis para lidar com controle de concorrência que atravessa transações de sistema.

Lembre-se de que essas são técnicas que você só deve usar se tiver que fazê-lo. Se puder fazer com que todas as suas transações de negócios se ajustem em uma transação de sistema, assegurando-se de que elas caibam dentro de uma única solicitação, então faça isso. Se você puder se virar com transações longas renunciando à escalabilidade, então faça isso. Deixando o controle de concorrência nas mãos do seu *software* de transações, você evitará uma grande quantidade de problemas. Estas técnicas são o que você tem que usar quando não puder fazer isso. Devido à natureza traiçoeira da concorrência, temos que enfatizar novamente que os padrões são um ponto de partida, não um destino. Descobrimos que eles são úteis, mas não alegamos ter encontrado uma cura para todos os males da concorrência.

Nossa primeira escolha para lidar com problemas de concorrência *offline* é o *Bloqueio Offline Otimista* (392), que basicamente usa controle de concorrência otimista ao longo das transações de negócios. Gostamos desta como primeira escolha porque é uma abordagem mais fácil de programar e fornece a melhor vivacidade. A limitação do *Bloqueio Offline Otimista* (392) é que você só descobre que uma transação de negócio irá falhar quando tenta confirmá-la e, em algumas circunstâncias, o custo desta descoberta tardia é muito alto. Usuários podem ter perdido uma hora de trabalho entrando com detalhes sobre um empréstimo e se você tiver muitas falhas eles perderão sua confiança no sistema. Sua alternativa é o *Bloqueio Offline Pessimista* (401), com o qual você descobre cedo se tem problemas, mas sai perdendo porque é mais difícil de programar e reduz a vivacidade.

Com qualquer dessas abordagens, você pode diminuir consideravelmente a complexidade se não tentar gerenciar os bloqueios em todos os objetos. Um *Bloqueio de Granularidade Alta* (412) permite que você gerencie simultaneamente a concorrência de um grupo de objetos. Outra maneira de você tornar a vida dos desenvolvedores de aplicações mais fácil é usar *Bloqueio Implícito* (449), o qual os poupa de ter que gerenciar os bloqueios diretamente. Isso não apenas poupa trabalho como também evita falhas quando as pessoas esquecem – e essas falhas são difíceis de serem encontradas.

Uma declaração comum sobre concorrência é que ela é uma decisão puramente técnica que pode ser tomada após os requisitos estarem completos. Nós discordamos. A escolha de controles otimista ou pessimista afeta toda a experiência do usuário com o sistema. Um projeto inteligente do *Bloqueio Offline Pessimista* (401) precisa de muita informação a respeito do domínio proveniente dos usuários do sistema. De modo semelhante, conhecimento do domínio é necessário para escolher bons *Bloqueios de Granularidade Alta* (412).

Brincar com concorrência é uma das tarefas de programação mais difíceis. É muito difícil testar código concorrente com confiança. Falhas de concorrência são difíceis de serem reproduzidas e muito difíceis de rastrear. Os padrões que descreve-

mos funcionaram para nós até agora, mas este é um território particularmente difícil. Se você precisar ir por esse caminho, vale a pena obter um pouco de ajuda experiente. Pelo menos consulte os livros mencionados no final deste capítulo.

Concorrência em Servidores de Aplicação

Até agora falamos sobre concorrência, principalmente em termos de múltiplas sessões rodando sobre uma fonte de dados compartilhada. Outra forma de concorrência é a do processo do próprio servidor de aplicações: como esse servidor lida com diversas solicitações concorrentemente e como isso afeta o projeto da aplicação no servidor? A grande diferença das outras questões de concorrência sobre as quais falamos é que a concorrência do servidor de aplicações não envolve transações, de modo que trabalhar com elas significa um afastamento do relativamente controlado mundo transacional.

A programação *multithread* explícita, com bloqueios e blocos de sincronização, é complicada de ser bem feita. É fácil introduzir defeitos que são muito difíceis de descobrir – falhas de concorrência são quase impossíveis de reproduzir – resultando em um sistema que funciona corretamente 99% do tempo, mas falha aleatoriamente. Tal *software* é incrivelmente frustrante de usar e depurar, de modo que nossa política é evitar tanto quanto possível a necessidade de manipulação explícita de sincronização e bloqueios. Os desenvolvedores de aplicações não devem quase nunca ter que lidar com esses mecanismos explícitos de concorrência.

A maneira mais simples de lidar com isso é usar **processo-por-sessão**, em que cada sessão roda no seu próprio processo. Sua grande vantagem é que o estado de cada processo é completamente isolado dos outros processos, de modo que os programadores de aplicações não têm que se preocupar com *multithreading*. Até onde o isolamento de memória abranger, é quase que igualmente eficaz fazer cada solicitação iniciar um novo processo ou ter um processo associado à sessão que estiver ociosa entre as solicitações. Muitos dos primeiros sistemas Web iniciavam um novo processo Perl para cada solicitação.

O problema do processo-por-sessão é que ele usa muitos recursos, uma vez que os processos são custosos. Para ser mais eficiente, você pode fazer um *pool* de processos, de modo que cada um lide apenas com uma única solicitação de cada vez mas possa lidar com múltiplas solicitações de diferentes sessões em uma seqüência. Essa abordagem de um *pool* de **processos-por-solicitação** usará muito menos processos para suportar um dado número de sessões. Seu isolamento é quase tão bom: você não tem muitas das complicadas questões de *multithreading*. O problema principal do processo por solicitação em relação ao processo-por-sessão é que você tem que assegurar que quaisquer recursos usados para manipular uma solicitação sejam liberados ao final da solicitação. A versão atual do Apache *mod-perl* usa este esquema, assim como fazem muitos dos sistemas de processamento de transação de larga escala.

Até mesmo o processo-por-solicitação irá precisar de muitos processos rodando para lidar com uma carga razoável. Você pode melhorar ainda mais a quantidade de dados transferidos fazendo um único processo executar múltiplas *threads*. Com esta abordagem ***thread*-por-solicitação**, cada solicitação é manipulada por uma única *thread* dentro de um processo. Já que as *threads* usam muito menos recursos do servidor do que um processo, você pode lidar com mais solicitações com menos *hardware*, de modo que seu servidor fica mais eficiente. O problema com o uso de *thread*-por-solicitação é que não há isolamento entre as *threads*, e qualquer *thread* pode pegar alguma parte dos dados a qual tiver acesso.

Na nossa visão, há muito a ser dito sobre o uso de processo-por-solicitação. Embora seja menos eficiente do que *thread*-por-solicitação, usar processo-por-solicitação é igualmente escalável. Você também obtém melhor robustez – se uma das *threads* sair de controle, ela pode trazer abaixo o processo inteiro, de forma que usar processo-por-solicitação limita o estrago. Especialmente com uma equipe menos experiente, a redução dos problemas causados por *threads* (e o tempo e custo de consertar falhas) justifica os custos extras de *hardware*. Acreditamos que poucas pessoas realmente executam algum teste de desempenho para estimar os custos relativos de *thread*-por-solicitação e processo-por-solicitação para suas aplicações.

Alguns ambientes fornecem um meio-termo permitindo que áreas isoladas de dados sejam designadas a uma única *thread*. COM faz isso com apartamentos de uma única *thread*, e J2EE faz isso com Enterprise Java Beans (e no futuro fará com isolamentos). Se sua plataforma tiver disponível algo parecido com isso, ela pode permitir que você faça e coma seu próprio bolo – seja lá o que isso signifique.

Se você usar *thread*-por-solicitação, o mais importante é criar e entrar em uma zona isolada onde os desenvolvedores de aplicações podem na maior parte das vezes, ignorar questões relativas a *multithreading*. O modo usual de fazer isso é levar a *thread* a criar novos objetos quando ela inicia o tratamento da solicitação e assegurar que esses objetos não sejam colocados em algum lugar (como em uma variável estática) onde outras *threads* possam vê-los. Dessa maneira, os objetos ficam isolados porque outras *threads* não têm como referenciá-los.

Muitos desenvolvedores estão preocupados com a criação de novos objetos porque lhes foi dito que a criação de objetos é um processo custoso. O resultado disso é que eles muitas vezes criam um *pool* de objetos. O problema com a criação de *pools* é que você tem que sincronizar o acesso aos objetos do *pool* de alguma maneira. Contudo o custo da criação de objetos depende muito da máquina virtual e das estratégias de gerenciamento de memória. Em ambientes modernos, a criação de objetos é realmente bastante rápida [Peckish]. (Quantos objetos Java para datas você acha que podemos criar em um segundo no P3 600 Mhz do Martin com Java 1.3? Nós lhe diremos mais adiante.) Criar objetos novos para cada sessão evita muitas falhas de concorrência e pode realmente aumentar a escalabilidade.

Embora essa tática funcione em muitos casos, ainda há algumas áreas que os desenvolvedores precisam evitar. Uma são as variáveis estáticas baseadas em classe ou variáveis globais, porque qualquer uso delas precisa ser sincronizado. Isso também é verdade para *singletons*. Se você precisar de algum tipo de memória global, use um *Registro* (448), o qual você pode implementar de tal modo que ele se pareça com uma variável estática, mas que na verdade use armazenamento específico para *thread*.

Mesmo se você puder criar objetos para a sessão, e assim criar uma zona comparativamente segura, alguns objetos são custosos de criar e, portanto, precisam ser manipulados diferentemente – o exemplo mais comum é uma conexão de banco de dados. Para lidar com isso, você pode colocar esses objetos em um *pool* explícito em que possa obter uma conexão enquanto precisar de uma e retorná-la quando terminar. Estas operações precisarão ser sincronizadas.

Leitura Adicional

De muitas formas, este capítulo apenas arranha a superfície de um tópico muito mais complexo. Para maior investigação, sugerimos começar com [Bernstein and Newcomer], [Lea] e [Schmidt *et al.*].

CAPÍTULO 6

Estado da Sessão

Quando falamos sobre concorrência, levantamos a questão da diferença entre transações de sistema e transações de negócio (Capítulo 5). Além de afetar a concorrência, essa diferença afeta também a forma de armazenar os dados usados em uma transação de negócio, mas que ainda não estão prontos para serem gravados de forma definitiva (*commit*) no banco de dados.

As diferenças entre transações de negócio e transações de sistema sustentam muito do debate sobre sessões sem estado contra sessões com estado. Muito tem sido escrito sobre esta questão, mas o problema básico muitas vezes está disfarçado atrás das questões técnicas de sistemas servidores com e sem estado. A questão fundamental é perceber que algumas sessões são inerentemente com estado e então decidir o que fazer a respeito do estado.

O Valor de Não Possuir Estado

O que as pessoas querem dizer com um servidor sem estado? O ponto principal a respeito de objetos, é óbvio, é que eles combinam estado (dados) com comportamento. Um objeto verdadeiramente sem estado é um objeto sem atributos. Tais "animais" aparecem de tempos em tempos, mas, francamente, eles são bastante raros. Na verdade, você pode argumentar, com razão, que um objeto sem estado caracteriza um projeto ruim.

Verifica-se, entretanto, que isso não é o que a maioria das pessoas quer dizer quando fala sobre ausência de estado em uma aplicação corporativa distribuída. Quando as pessoas se referem a um servidor sem estado, querem dizer um objeto que não retém estado entre solicitações. Tal objeto pode muito bem ter atributos, mas quando você invoca um método em um servidor sem estado os valores dos atributos são indefinidos.

Um exemplo de um objeto servidor sem estado pode ser um servidor que retorne uma página Web contendo informações a respeito de um livro. Você faz uma chamada a este servidor acessando uma URL – o objeto pode ser um documento ASP ou um servlet. Na URL você fornece um número ISBN que o servidor usa para gerar a resposta HTTP. Durante a interação, o objeto servidor, antes de gerar a página HTML, pode armazenar o ISBN, o título e o preço do livro em atributos do objeto, quando ele os obtém do banco de dados. Talvez ele execute alguma lógica de negócio para determinar quais informações complementares mostrar para o usuário. Assim que tiver terminado seu trabalho, entretanto, estes valores se tornam inúteis. O próximo ISBN é outra história completamente diferente, e o objeto servidor será provavelmente reinicializado para limpar quaisquer valores antigos evitando assim a ocorrência de erros.

Agora imagine que você queira guardar todos os ISBNs visitados por um determinado endereço IP cliente. Você pode guardá-los em uma lista mantida pelo objeto servidor. Contudo, esta lista precisa persistir entre as solicitações e assim você tem um objeto servidor com estado. Essa mudança de sem estado para com estado é muito mais do que uma mudança de palavras. Para muitas pessoas, servidores com estado são nada menos do que desastrosos. Por que isso?

A questão básica diz respeito a recursos do servidor. Qualquer objeto servidor com estado precisa manter todo o seu estado enquanto espera que o usuário medite sobre a página Web enviada. Um objeto servidor sem estado, contudo, pode processar outras solicitações de outras sessões. Considere um exemplo completamente irreal, porém ilustrativo. Suponha que tenhamos uma centena de pessoas que queiram obter informações a respeito de livros e que processar uma requisição sobre um livro leve um segundo. Cada pessoa faz uma solicitação a cada dez segundos, e todas as solicitações são perfeitamente balanceadas. Se quisermos guardar as solicitações de um usuário em um objeto servidor com estado, temos que ter um objeto servidor por usuário: cem objetos. Entretanto, durante 90% do tempo, estes objetos ficam parados sem nada para fazer. Se desistirmos de rastrear o ISBN e apenas usarmos objetos servidores sem estado para responder às solicitações, podemos nos sair bem com apenas dez objetos servidores, integralmente utilizados, durante todo o tempo.

A questão é que, se não tivermos estados entre chamadas de métodos, não importa que objeto responde à solicitação, mas, se armazenarmos o estado, então precisamos sempre pegar o mesmo objeto. A ausência de estado nos permite fazer um *pool* com nossos objetos de modo que precisemos de menos objetos para lidar com mais usuários. Quanto mais usuários inativos tivermos, mais valiosos serão os servidores sem estado. Como você pode imaginar, servidores sem estado são muito úteis em Web *sites* com muito tráfego. A ausência de estado também é apropriada para a Web, já que o HTTP é um protocolo sem estado.

Então, tudo deveria ser sem estado, certo? Bem, deveria, se isso fosse possível. O problema é que muitas interações de clientes são inerentemente com estado. Considere a metáfora do carrinho de compras presente em milhares de aplicações de comércio eletrônico. A interação do usuário consiste em navegar pelas páginas de diversos livros e escolher os que deseja comprar. O conteúdo do carrinho de compras precisa ser lembrado durante toda a sessão do cliente. Basicamente, temos uma transação de negócio com estado, o que significa que a sessão tem que ter estado. Se eu apenas procuro livros e não compro nada, minha sessão não tem estado, mas se eu comprar, ela tem. Não podemos evitar o estado a não ser que queiramos permanecer na pobreza. Em vez disso, temos que decidir o que fazer com ele.

A boa notícia é que podemos usar um servidor sem estado para implementar uma sessão com estado. A novidade interessante é que, ainda que isso seja possível, podemos não querê-lo.

Estado da Sessão

Os itens no carrinho de compras são o **estado da sessão**, o que quer dizer que os dados no carrinho são relevantes apenas para aquela sessão em particular. Este estado está dentro de uma transação de negócio, o que significa que está separado de outras sessões e suas transações de negócio. (Continuarei a pressupor, nesta discussão, que cada transação de negócio roda em uma sessão apenas e que cada sessão executa apenas uma transação de negócio de cada vez). O estado de sessão é diferente do que chamo de **registro de dados**, que são os dados persistentes, de longo alcance, armazenados no banco de dados e visíveis para todas as sessões. Para se tornar um registro de dados, o estado da sessão precisa ser armazenado de forma persistente (*commited*).

Uma vez que o estado da sessão está dentro de uma transação de negócio, ele possui muitas das propriedades que as pessoas normalmente associam a transações, tais como as propriedades ACID (atomicidade, consistência, isolamento e durabilidade). As conseqüências disso nem sempre são compreendidas.

Uma conseqüência interessante é o efeito sobre a consistência. Enquanto o cliente está editando uma apólice de seguros, o estado corrente da apólice pode não ser válido. O cliente altera um valor, usa uma solicitação para enviá-lo para o sistema e o sistema responde indicando valores inválidos. Esses valores são parte de um estado de sessão, mas não são válidos. Este fato muitas vezes ocorre com o estado da sessão – ele não satisfaz as regras de validação enquanto está sendo trabalhado. Isso irá ocorrer apenas quando a transação de negócio for armazenada de maneira persistente no banco (*commited*).

A questão maior no que diz respeito ao estado da sessão é lidar com o isolamento. Devido ao grande número de aspectos envolvidos, diversas coisas podem ocorrer enquanto o cliente edita uma apólice. A mais óbvia consiste em duas pessoas editando a mesma apólice ao mesmo tempo. No entanto, o problema não são apenas as alterações. Considere a existência de dois registros, a própria apólice e o registro do cliente. A apólice tem um valor de risco que depende parcialmente do código postal no registro do cliente. O cliente começa a editar a apólice e, após dez minutos, faz algo que abre o registro do cliente de modo que ele possa ver o código postal. Entretanto, durante esse período, outra pessoa alterou o código postal e o valor do risco – levando a uma leitura inconsistente. Veja, na página 89-90, uma estratégia para lidar com isso.

Nem todos os dados armazenados pela sessão contam como estado da sessão. A sessão pode armazenar em cache alguns dados que, em verdade, não precisariam ser persistidos entre solicitações, mas que são armazenados para aumentar o desempenho. Uma vez que você pode perder o cache sem perder o comportamento correto, isto é diferente do estado da sessão, que deve ser armazenado entre solicitações para que o comportamento seja o correto.

Modos de Armazenar o Estado da Sessão

Como você pode armazenar o estado da sessão uma vez que descobriu que precisará dele? Divido as opções em três escolhas não muito distintas, porém básicas.

O *Estado da Sessão no Cliente* (427) armazena os dados no cliente. Há diversos modos de fazer isso: codificando os dados em uma URL para uma apresentação Web, usando *cookies*, serializando os dados em algum campo escondido em um formulário Web ou armazenando os dados em objetos em um cliente rico.

O *Estado da Sessão no Servidor* (429) pode ser tão simples quanto armazenar os dados na memória entre solicitações. Normalmente, entretanto, há um mecanismo para armazenar o estado da sessão em algum lugar mais durável, como um objeto serializado. O objeto pode ser armazenado no sistema de arquivos local do servidor de aplicações ou colocado em uma fonte de dados compartilhada. Esta poderia ser uma simples tabela em um banco de dados com um ID da sessão como chave e um objeto serializado como valor.

O *Estado da Sessão no Banco de Dados* (432) também é armazenamento no lado servidor, porém envolve a separação dos dados em tabelas e campos e o armazenamento deles no banco de dados, de modo semelhante ao que você faria com dados mais duráveis.

Há algumas questões envolvidas na escolha da opção. Primeiro falarei sobre as necessidades de largura da banda entre o cliente e o servidor. Usar *Estado da Sessão no Cliente* (427) significa que os dados da sessão precisam ser transferidos pela rede em cada solicitação. Se estivermos falando de apenas alguns campos, isso não é grande coisa, mas quantidades maiores de dados resultam em transferências maiores. Em uma aplicação, estes dados chegaram a quase um *megabyte* ou, como um dos membros da nossa equipe colocou, o tamanho de três peças de Shakespeare. É verdade que estávamos usando XML entre os dois, o que não é a forma mais compacta de transmissão de dados, porém mesmo assim havia muitos dados para se trabalhar.

É claro que alguns dados precisarão ser transferidos porque eles têm que ser vistos na apresentação, porém, usar *Estado da Sessão no Cliente* (427) significa que, em cada solicitação, você tem que transferir todos os dados que o servidor usa, mesmo que o cliente não precise deles para fins de exibição. Tudo isso significa que você não irá querer usar *Estado da Sessão no Cliente* (427) a não ser que a quantidade de estado de sessão que você precisa armazenar seja pequena. Você também tem que se preocupar com segurança e integridade. A menos que você criptografe os dados, deve ter em mente que qualquer usuário mal-intencionado pode editar seus dados de sessão, o que poderia levá-lo a uma versão inteiramente nova do conceito de "diga o seu próprio preço."

Os dados da sessão têm que ser isolados. Na maior parte dos casos o que está acontecendo em uma sessão não deve afetar o que está acontecendo em outra. Se reservarmos um itinerário de vôo, isso não deve ter nenhum efeito sobre nenhum outro usuário até que o vôo seja confirmado. De fato, parte do significado de dados de sessão é que eles são invisíveis fora da sessão. Isso se torna uma questão complicada se você usar *Estado da Sessão no Banco de Dados* (432), porque você tem que trabalhar duro para isolar os dados da sessão dos registros de dados armazenados no banco de dados.

Se você tiver muitos usuários, deve considerar a utilização de *clusters* para melhorar a taxa de transferência de dados (*throughput*). Neste caso, você deve considerar se você precisa de migração de sessão. A **migração de sessão** permite que uma sessão se mova de um servidor a outro quando um servidor trata uma solicitação da sessão e outros servidores tratam as demais. Seu oposto é a **afinidade de servidor**, que força um servidor a tratar todas as solicitações para uma sessão em particular. A migração de servidor leva a um melhor balanceamento dos seus servidores, especial-

mente se suas sessões forem longas. Entretanto, isso pode ser complicado se você estiver usando *Estado da Sessão no Servidor* (429), porque, muitas vezes, apenas a máquina que lida com a sessão pode encontrar esse estado facilmente. Há maneiras de contornar isso, mas estas maneiras obscurecem a distinção entre o *Estado da Sessão no Banco de Dados* (432) e o *Estado da Sessão no Servidor* (429).

A afinidade com o servidor pode levar a problemas maiores do que você poderia inicialmente imaginar. Na tentativa de garantir a afinidade com o servidor, o *cluster* de máquinas não pode a todo momento inspecionar as chamadas para ver de qual sessão elas fazem parte. Como resultado, ele irá aumentar a afinidade de modo que todas as chamadas de um cliente sejam encaminhadas para o mesmo servidor de aplicações. Muitas vezes, isso é feito pelo endereço IP do cliente. Se o cliente estiver atrás de um *proxy*, isso poderia significar que muitos clientes estejam usando o mesmo endereço IP e assim associados a um servidor em particular. Isso pode ser muito ruim se você chegar a uma situação em que a maior parte do seu tráfego é manipulado por um único servidor responsável pelo endereço IP da AOL!

Se o servidor for usar o estado da sessão, precisa obtê-lo em um formato que possa ser usado rapidamente. Se você usar o *Estado da Sessão no Servidor* (429), o estado da sessão já está lá. Se você usar o *Estado da Sessão no Cliente* (427), ele está lá, mas freqüentemente precisa antes ser colocado na forma apropriada. Se você usar o *Estado da Sessão no Banco de Dados* (432), precisa ir ao banco de dados para obtê-lo (e, talvez, também executar alguma transformação). Isso significa que cada abordagem pode ter diferentes efeitos sobre a resposta do sistema. O tamanho e a complexidade dos dados terão efeito nesta hora.

Se você tiver um sistema público de venda a varejo, provavelmente não terá tantos dados em cada sessão, mas terá muitos usuários na maior parte do tempo inativos. Por este motivo, o *Estado da Sessão no Banco de Dados* (432) pode funcionar bem em termos de desempenho. Em um sistema de *leasing*, você corre o risco de arrastar uma grande quantidade de dados de e para o banco de dados em cada solicitação. É nessa situação que o *Estado da Sessão no Servidor* (429) pode lhe dar um desempenho melhor.

Um dos grandes bichos-papões, em muitos sistemas, é quando o usuário cancela uma sessão e diz "pode deixar, esqueça". Isso é especialmente complicado com aplicações B2C, porque o usuário normalmente não diz "pode deixar, esqueça", ele simplesmente desaparece e não volta mais. O *Estado da Sessão no Cliente* (427) certamente ganha aqui, porque você pode facilmente esquecer esse usuário. Nas outras abordagens, você precisa limpar o estado da sessão quando percebe que ela foi cancelada, assim como configurar um sistema que lhe permita cancelar a sessão após a passagem de um certo período de tempo (*timeout*). Boas implementações do *Estado da Sessão no Servidor* (429) permitem um *timeout* automático.

Assim como o cancelamento pelo usuário, considere o que acontece quando um sistema cancela: um cliente pode falhar, um servidor pode ter problemas e uma conexão de rede pode desaparecer. O *Estado da Sessão no Banco de Dados* (432) normalmente lida muito bem com as três situações. O *Estado da Sessão no Servidor* (429) pode ou não sobreviver, dependendo de o objeto da sessão ter *backup* em armazenamento não-volátil e onde esse armazenamento é mantido. O *Estado da Sessão no Cliente* (427) não irá sobreviver a uma falha no cliente, mas deve sobreviver à queda dos demais.

Não se esqueça do esforço de desenvolvimento envolvido nestes padrões. Nesse sentido, o *Estado da Sessão no Servidor* (429) é, normalmente, o mais fácil de implementar, especialmente se você não tiver que persistir o estado da sessão entre requi-

sições. O *Estado da Sessão no Banco de Dados* (432) e o *Estado da Sessão no Cliente* (427) normalmente demandarão código para transformar o formato interno de um banco de dados ou um formato de transporte em um formato apropriado para os objetos de sessão. Esse tempo extra significa que você não conseguirá efetuar o mesmo trabalho tão rapidamente quanto o faria com o *Estado da Sessão no Servidor* (429), especialmente se os dados forem complexos. À primeira vista, o *Estado da Sessão no Banco de Dados* (432) poderia não parecer tão complexo se você já estiver efetuando o mapeamento para tabelas no banco de dados, porém o esforço extra de desenvolvimento vem da necessidade de manter os dados da sessão isolados de todas as outras utilizações do banco de dados.

As três abordagens não são mutuamente excludentes. Você pode usar uma mistura de duas ou três delas para armazenar diferentes partes do estado da sessão. Contudo, isso normalmente torna as coisas mais complicadas, já que você nunca tem certeza de que parte do estado vai em que parte do sistema. Se, apesar disso, você usar algo em conjunto com o *Estado da Sessão no Cliente* (427), terá que manter pelo menos um identificador de sessão no *Estado da Sessão no Cliente* (427) mesmo se o resto do estado for mantido usando os outros padrões.

Minha preferência é pelo *Estado da Sessão no Servidor* (429), especialmente se o estado da sessão for armazenado remotamente, de modo que possa sobreviver a uma falha no servidor. Também gosto do *Estado da Sessão no Cliente* (427) para IDs e dados de sessões que sejam muito pequenos. Não gosto do *Estado da Sessão no Banco de Dados* (432), a menos que você precise de *failover* e *clusters* e você não possa armazenar mementos remotos ou se o isolamento entre sessões não for um problema para você.

CAPÍTULO 7

Estratégias de Distribuição

Os objetos têm sido usados já há um certo tempo e às vezes parece que, desde sua criação, as pessoas têm querido distribuí-los. Entretanto, a distribuição de objetos tem muito mais armadilhas do que muitas pessoas percebem [Waldo *et al.*], especialmente quando elas estão sob a influência dos livretos dos vendedores. Este capítulo é sobre algumas dessas duras lições – lições que tenho visto muitos dos meus clientes aprenderem de modo difícil.

O Fascínio dos Objetos Distribuídos

Há uma apresentação recorrente que eu costumava ver duas ou três vezes por ano durante revisões de projetos. Orgulhosamente, o arquiteto de sistemas de um novo sistema OO mostra seu plano para um novo sistema de objetos distribuídos – vamos fazer de conta que é algum sistema de pedidos. Ele me mostra um projeto que se parece com a Figura 7.1, com objetos remotos separados para clientes, pedidos, produtos e entregas. Cada um deles é um componente separado que pode ser colocado em um nó de processamento separado.

Pergunto "Por que você faz isso?"

"Desempenho, é claro", o arquiteto retruca, olhando para mim de um modo um pouco estranho. "Podemos rodar cada componente em uma caixa separada. Se um componente ficar ocupado demais, acrescentamos caixas extras para ele, de modo que possamos balancear a carga da nossa aplicação". O olhar agora é curioso como se ele estivesse tentando imaginar se eu realmente conheço alguma coisa sobre objetos distribuídos.

Enquanto isso, deparo-me com um dilema interessante. Será que eu digo que este projeto é ruim e me mostram a porta de saída imediatamente? Ou tento aos poucos esclarecer o meu cliente? Essa última opção é mais lucrativa, mas bem mais difícil, já que o cliente geralmente está bastante satisfeito com sua arquitetura, e é duro desistir de um sonho do qual se gosta muito.

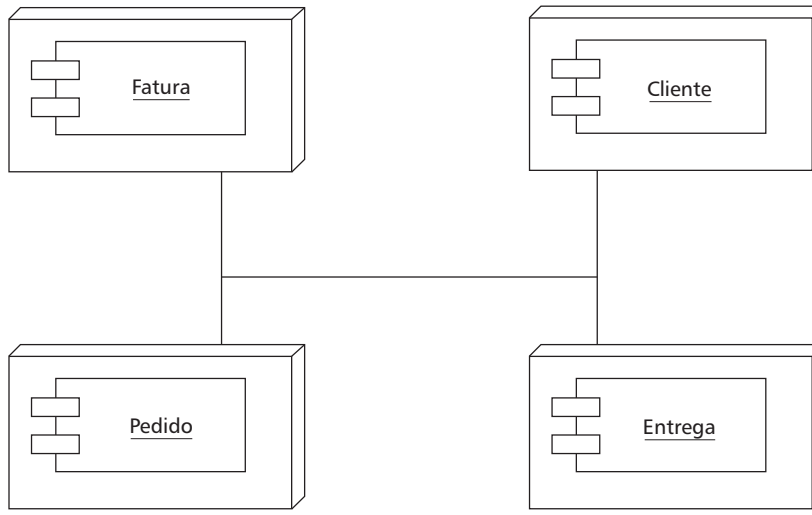


Figura 7.1 Distribuindo uma aplicação, colocando diferentes componentes em diferentes nós (não recomendado!).

Assim, pressupondo que você não tenha mostrado a porta de saída para este livro, acho que irá querer saber por que esta arquitetura distribuída é ruim. Afinal de contas, muitos vendedores de ferramentas lhe dirão que o motivo principal do uso de objetos distribuídos é que você pode pegar um monte de objetos e posicioná-los como quiser em nós de processamento. Além disso, seu *middleware* poderoso fornece transparência. A transparência permite que os objetos chamem uns aos outros dentro de um processo ou entre processos sem ter que saber se o objeto chamado está no mesmo processo, em outro processo ou em outra máquina.

A transparência é valiosa, no entanto, ainda que muitas coisas possam ser tornadas transparentes em objetos distribuídos, o desempenho normalmente não é uma delas. Embora nosso arquiteto idealizado estivesse distribuindo objetos do modo que vinha fazendo por motivos de desempenho, na verdade seu projeto irá prejudicar o desempenho ou tornar o sistema muito mais difícil de construir e distribuir ou ainda, o que é mais freqüente, terá ambos os defeitos.

Interfaces Locais e Remotas

A principal razão pela qual a distribuição por modelo de classe não funciona tem a ver com um fato fundamental a respeito de computadores. Uma chamada de procedimento dentro de um mesmo processo é muito, muito rápida. Uma chamada de um procedimento entre dois processos separados ordens de magnitude mais devagar. Rode esse processo em uma outra máquina e você pode acrescentar mais uma ou duas ordens de magnitude, dependendo da topografia de rede envolvida.

O resultado disso é que a interface de um objeto a ser usado remotamente deve ser diferente daquela de um objeto usado dentro do mesmo processo.

Uma interface local fica melhor como uma interface de granularidade baixa. Assim, se eu tiver uma classe endereço, uma boa interface terá métodos separados pa-

ra ler a cidade, o estado, gravar a cidade, gravar o estado, e assim por diante. Uma interface de granularidade baixa é boa porque segue o princípio OO, que preconiza muitos pedaços pequenos de *software* que podem ser combinados e sobrescritos de várias maneiras para estender o projeto no futuro.

Uma interface de granularidade baixa não funciona bem quando é remota. Quando as chamadas de métodos são lentas, você quer obter ou atualizar a cidade, estado e código postal em uma chamada, em vez de em três. A interface resultante tem granularidade alta, projetada não para maior flexibilidade e extensibilidade mas para minimizar as chamadas. Aqui você verá uma interface lidar com todos os detalhes de uma leitura ou atualização de endereços. É muito mais complicado de programar, mas por motivos de desempenho, você precisará deste tipo de interface.

É claro, que os vendedores irão lhe dizer é que não há *overhead* no uso do seu *middleware* para chamadas locais e remotas. Se for uma chamada local, ela é feita com a velocidade de uma chamada local. Se for uma chamada remota, ela é feita mais devagar. Assim, você só paga o preço de uma chamada remota quando precisar de uma. Isso é, até certo ponto, verdadeiro, porém não evita a questão essencial de que qualquer objeto que possa ser usado remotamente deva ter uma interface de granularidade alta enquanto que cada objeto que não for usado remotamente deve ter uma interface de granularidade baixa. Sempre que dois objetos se comunicam, você deve escolher qual delas usar. Se o objeto puder, alguma vez, estar em um processo separado, você tem que usar a interface de granularidade alta e pagar o preço do modelo de programação mais difícil. Obviamente, só faz sentido pagar esse preço quando você precisar, então você deve minimizar a quantidade de colaborações interprocessos.

Por esses motivos você não pode simplesmente pegar um grupo de classes que projetou para um ambiente de um único processo, atirar CORBA ou algo semelhante nelas e disto resultar um modelo distribuído. Um projeto distribuído é mais do que isso. Se você basear sua estratégia de distribuição em classes, acabará com um sistema que executa muitas chamadas remotas e, desta forma, precisa de interfaces deselegantes, com granularidade alta. No final, mesmo com interfaces de granularidade alta em toda classe remota, você ainda terá como bônus, demasiadas chamadas remotas e um sistema complicado para modificar.

Assim, chegamos à minha **Primeira Lei do Projeto de Objetos Distribuídos**: não distribua seus objetos!

Como, então, usar efetivamente múltiplos processadores? Na maior parte dos casos, o caminho apropriado é a clusterização (veja a Figura 7.2). Coloque todas as classes em um único processo e então execute múltiplas cópias desse processo nos diversos nós de processamento. Desta forma, cada processo usa chamadas locais para executar o trabalho e assim o faz mais rápido. Além disso, você pode usar interfaces de granularidade baixa em todas as classes dentro do processo e assim obter maior facilidade de manutenção com um modelo de programação mais simples.

Onde Você Tem que Distribuir

Você quer minimizar as fronteiras da distribuição e utilizar a clusterização de seus nós de processamento tanto quanto possível. O obstáculo é que há limites nessa abordagem – isto é, situações onde você precisará separar os processos. Se você for sensato, lutará como um rato encurralado para eliminar tantas fronteiras de distribuição quanto puder, mas não as eliminará por completo.

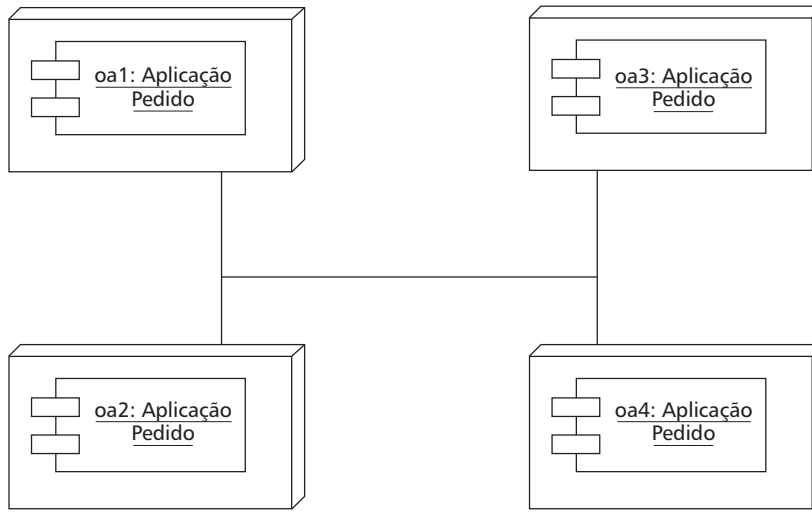


Figura 7.2 A utilização de *clusters* envolve a colocação de diversas cópias da mesma aplicação em diferentes nós.

- Uma separação óbvia é entre o lado cliente e o lado servidor nas aplicações de negócio. Os PCs nas mesas dos usuários são nós de processamento diferentes daqueles onde residem os repositórios de dados compartilhados. Já que eles são máquinas diferentes, você precisa de processos separados que se comuniquem. A divisão cliente-sevidor é uma típica divisão interprocessos.
- Uma segunda divisão freqüentemente ocorre entre o lado servidor da aplicação (o servidor de aplicações) e o banco de dados. É claro que você não é obrigado a fazer isso. Você pode rodar todo o lado servidor da aplicação no próprio processo do banco de dados usando coisas tais como procedimentos armazenados. Mas, na maioria das vezes, isso não é tão prático, de modo que você tem que ter processos separados. Eles até podem rodar na mesma máquina mas, apenas por trabalhar com processos separados, você terá que pagar a maior parte do custo de chamadas remotas. Felizmente, a linguagem SQL é projetada como uma interface remota, de modo que você normalmente pode arranjar as coisas de modo a minimizar esse custo.
- Outra separação em processos pode ocorrer em um sistema Web entre o servidor Web e o servidor de aplicação. Se tudo for igual, é melhor rodar os servidores Web e de aplicações em um único processo, mas nem tudo é sempre igual.
- Você pode ter que separar devido a diferenças entre vendedores. Se você estiver usando um pacote de *software*, ele muitas vezes rodará no seu próprio processo, de modo que mais uma vez você estará distribuindo. Um bom pacote terá, no mínimo, uma interface de granularidade alta.
- E, finalmente, pode haver alguma razão genuína para você ter que dividir seu *software* servidor de aplicações. Você deve até mesmo vender seus próprios avós para evitar isso, mas algumas vezes há certos casos em que não é possível evitá-lo. Nesses casos, você tem de tampar o nariz e dividir seu *software* em componentes remotos com granularidade alta.

O tema recorrente, segundo a memorável frase de Colleen Roe, é ser "parcimonioso com a distribuição de objetos". Venda primeiro sua avó favorita, se for possível.

Trabalhando com as Fronteiras da Distribuição

Durante o projeto de seu sistema, você precisa limitar a distribuição tanto quanto possível, mas onde ela for necessária, você tem de levar em conta as fronteiras de distribuição. Toda chamada remota é o equivalente cibernético de uma viagem em uma carruagem puxada a cavalos. Todos os locais no sistema mudarão de formato para minimizar chamadas remotas. Este é o preço esperado.

Entretanto, dentro de um único processo, você ainda pode projetar usando objetos de granularidade baixa. A chave é usar esses objetos internamente e colocar objetos de granularidade alta nas fronteiras de distribuição, cujo único papel é fornecer uma interface remota para os objetos de granularidade baixa. Os objetos de granularidade alta agem simplesmente como uma fachada para os objetos de granularidade baixa. A existência desta fachada é motivada apenas por questões de distribuição – daí o nome de *Fachada Remota* (368).

O uso de uma *Fachada Remota* (368) ajuda a minimizar as dificuldades que as interfaces de granularidade alta introduzem. Dessa forma, apenas os objetos que realmente precisam de um serviço remoto invocam o método de granularidade alta e fica óbvio para os desenvolvedores que eles estão pagando esse preço. A transparência tem suas virtudes, mas você não quer ser transparente em relação a uma potencial chamada remota.

Mantendo as interfaces de granularidade alta como meras fachadas, você permite que as pessoas usem os objetos de granularidade baixa sempre que elas souberem que estão rodando no mesmo processo. Isso torna toda a política de distribuição muito mais explícita. De mãos dadas com a *Fachada Remota* (368) está o *Objeto de Transferência de Dados* (380). Você precisa não apenas de métodos de granularidade alta, mas também transferir objetos de granularidade alta. Quando você solicita um endereço, precisa enviar essa informação em um bloco. Normalmente você não pode enviar o próprio objeto do domínio, porque ele está preso a uma rede de referências locais entre objetos de granularidade baixa. Assim, você deve pegar todos os dados que o cliente precisa e os empacotar em um objeto específico para a transferência – daí o termo *Objeto de Transferência de Dados* (380). (Muitas pessoas na comunidade *enterprise Java* usam o termo *objeto valor* para isso, mas causa um conflito com outros significados do termo *Objeto Valor* (453)). O *Objeto de Transferência de Dados* (380) aparece nos dois lados da conexão, de modo que é importante que ele não faça nenhuma referência a nada que não seja compartilhado através da conexão. Isto resulta no fato de que um *Objeto de Transferência de Dados* (380) normalmente apenas referencia outros *Objetos de Transferência de Dados* (380) e objetos fundamentais tais como *strings*.

Outro caminho para a distribuição é ter um intermediário que migre objetos entre os processos. A idéia aqui é usar um esquema de *Carga Tardia* (200) em que, em vez de efetuar uma leitura tardia de um banco de dados, você move objetos através da conexão. A parte complicada é assegurar que você não acabe com demasiadas chamadas remotas. Ainda não vi ninguém tentar isso em uma aplicação, mas algumas ferramentas de mapeamento O/R (p. ex., TOPLink) têm esta facilidade e tenho ouvido alguns relatos interessantes a esse respeito.

Interfaces para Distribuição

Tradicionalmente, as interfaces de componentes distribuídos têm sido baseadas em chamadas a procedimentos remotos, seja por meio de procedimentos globais ou como métodos nos objetos. Nos últimos anos, entretanto, temos começado a ver interfaces baseadas em XML sobre HTTP. A forma mais comum dessa interface será provavelmente o SOAP, mas muitas pessoas efetuaram experimentos na área durante muitos anos.

A comunicação HTTP baseada em XML é útil por diversos motivos. Ela permite facilmente que uma grande quantidade de dados seja enviada de forma estruturada, em uma única viagem de ida e volta. Isso é bom, uma vez que o número de chamadas remotas precisa ser minimizado. O fato de XML ser um formato comum, com *parsers* disponíveis em muitas plataformas, permite que sistemas construídos em plataformas inteiramente diferentes se comuniquem, acrescido ao fato do protocolo HTTP ser universal. O fato do XML ser textual torna fácil ver o que está acontecendo através da conexão. E ainda é fácil passar HTTP através de *firewalls* quando questões políticas e de segurança tornam difícil uma alternativa.

Ainda assim, uma interface orientada a objetos de classes e métodos também tem valor. Mover todos os dados transferidos para estruturas XML e *strings* pode adicionar uma sobrecarga considerável à chamada remota. As aplicações certamente têm visto uma significativa melhora de desempenho substituindo uma interface baseada em XML por uma chamada remota. Se ambos os lados da conexão usarem o mesmo mecanismo binário, uma interface XML não traz muito mais do que um conjunto vistoso de acrônimos. Se você tiver dois sistemas construídos com a mesma plataforma, estará melhor servido se usar o mecanismo de chamadas remotas nessa plataforma. Serviços Web se tornam úteis quando você quer que plataformas diferentes conversem. Minha posição é usar serviços Web baseados em XML apenas quando uma abordagem mais direta não for possível.

É claro que você pode ter o melhor dos dois mundos colocando uma camada HTTP sobre uma interface orientada a objetos. Todas as chamadas para o servidor Web são traduzidas por esta camada em chamadas para uma interface orientada a objetos correspondente. Até certo ponto, isso lhe dá o melhor dos dois mundos, mas aumenta a complexidade, já que irá precisar tanto do servidor Web quanto do mecanismo para uma interface remota OO. Portanto, você só deve fazer isso se precisar de uma API HTTP, bem como de uma API remota OO, ou se as facilidades da API OO remota para segurança e gerenciamento de transações tornar mais fácil lidar com essas questões do que usando objetos não-remotos.

Nas minhas discussões aqui, parti do pressuposto de uma interface síncrona baseada em RPC. Entretanto, embora isso seja o que descrevi, realmente não considero que seja sempre a melhor maneira de lidar com um sistema distribuído. Cada vez mais, minha preferência é por uma abordagem inerentemente assíncrona, baseada em mensagens. A exploração de padrões para trabalhos baseados em mensagens é um tópico relativamente grande por si só, por isso a evitei neste livro. Espero que um livro sobre isso apareça em um futuro próximo, mas, por enquanto, tudo o que posso fazer é encorajá-lo a considerar abordagens assíncronas, baseadas em mensagens. Particularmente, acho que elas são o melhor uso dos serviços Web, ainda que a maior parte dos exemplos publicados até agora sejam síncronos.

Capítulo 8

Juntando Tudo

Até agora estas narrativas olharam um aspecto de um sistema e exploraram as diversas opções para tratá-lo. Agora é hora de reunir tudo e começar a responder às questões traiçoeiras de quais padrões usar ao projetar uma aplicação corporativa.

O conselho, neste capítulo, é de muitas maneiras uma repetição do conselho dado em capítulos anteriores. Devo admitir que fiquei na dúvida se este capítulo seria necessário. Contudo, pensei que seria bom contextualizar toda a discussão agora que, espero eu, você tem pelo menos as linhas gerais do escopo completo dos padrões deste livro.

Enquanto escrevo isso, estou inteiramente ciente das limitações do meu conselho. Frodo disse em *O Senhor dos Anéis*: "Não peça conselhos aos Elfos, porque eles dirão não e sim." Embora eu não esteja reivindicando qualquer conhecimento imortal, certamente entendo sua resposta de que conselhos são muitas vezes um presente perigoso. Se estiver lendo isto para tomar decisões referentes à arquitetura do seu projeto, pense que você sabe muito mais sobre seu projeto do que eu. Uma das maiores frustrações em ter bastante conhecimento sobre algo é que as pessoas muitas vezes vêm a mim em uma conferência ou enviam uma mensagem pelo correio eletrônico pedindo conselhos sobre suas decisões de processo ou arquitetura. Não há como você dar conselhos específicos baseado em uma descrição de cinco minutos. Escrevo este capítulo com ainda menos conhecimento do seu problema.

Assim, leia com o espírito com o qual ele é apresentado. Não conheço todas as respostas e certamente não conheço suas questões. Use este conselho para estimular sua reflexão, mas não o use em substituição à sua reflexão. No final, você tem que tomar suas próprias decisões e viver com elas.

Uma coisa boa é que suas decisões não têm de ficar para sempre gravadas em pedra. A refatoração arquitetural é difícil, e ainda não conhecemos todos os seus custos, mas ela não é impossível. Aqui o melhor conselho que posso dar é que, mesmo se você não gostar de nada da programação extrema [Beck XP], ainda assim deveria

considerar seriamente três práticas técnicas: integração contínua [Fowler CI], desenvolvimento conduzido por testes [Beck TDD] e refatoração [Fowler Refactoring]. Essas técnicas não são uma panacéia, mas tornarão muito mais fácil mudar de idéia, quando você descobrir que precisa fazê-lo. E você irá precisar, a menos que tenha mais sorte ou mais habilidade do que qualquer pessoa que eu já tenha conhecido.

Começando com a Camada de Domínio

O começo do processo é decidir qual abordagem de lógica de domínio usar. Os três principais competidores são o *Roteiro de Transação* (120), o *Módulo Tabela* (134) e o *Modelo de Domínio* (126).

Conforme ressaltei no Capítulo 2 (página 45-46), a maior força que o conduz por este trio é a complexidade da lógica do domínio, algo totalmente impossível de quantificar, ou até mesmo de qualificar, com algum grau de precisão. Contudo outros fatores também influem na decisão, em particular a dificuldade de conexão com um banco de dados.

O mais simples dos três padrões é o *Roteiro de Transação* (120). Ele se ajusta ao modelo procedural com o qual a maioria das pessoas ainda se sente confortável. Eficientemente encapsula a lógica de cada transação do sistema em um roteiro facilmente compreensível. Além disso, ele é fácil para construir um *Roteiro de Transação* sobre um banco de dados relacional. Sua maior falha é não tratar adequadamente a lógica de negócio complexa, sendo especialmente suscetível a código duplicado. Se você tiver apenas uma aplicação simples de catálogo, com pouco mais do que um carrinho de compras rodando sobre uma estrutura básica de preços, então o *Roteiro de Transação* (120) será perfeitamente suficiente. Entretanto, à medida que sua lógica ficar mais complicada, suas dificuldades se multiplicam.

No outro lado da escala está o *Modelo de Domínio* (126). Fanáticos radicais por objetos, assim como eu, não enxergam uma aplicação de outra forma. Afinal, se uma aplicação é simples o suficiente para que possa ser escrita com o *Roteiro de Transação* (120), por que nossos imensos intelectos deveriam se incomodar com um problema tão sem valor? Além disso, minha experiência me leva a não ter dúvidas de que nada pode lidar melhor com o inferno da lógica de domínio realmente complexa do que um rico *Modelo de Domínio* (126). Uma vez que você tenha se acostumado a trabalhar com um *Modelo de Domínio* (126), até mesmo problemas simples podem ser tratados com facilidade.

Ainda assim, o *Modelo de Domínio* (126) tem suas falhas. Nos primeiros lugares da lista está a dificuldade de aprender a como usar um modelo de domínio. Fanáticos por objetos muitas vezes torcem seus narizes para pessoas que simplesmente não conseguem entender objetos, mas a verdade é que, se for para ser bem feito, um *Modelo de Domínio* (126) requer habilidade – mal feito ele é um desastre. A segunda grande dificuldade de um *Modelo de Domínio* (126) é a sua conexão a um banco de dados relacional. É claro que um verdadeiro fanático por objetos usa de artimanhas para resolver este problema com um empurrãozinho de um banco de dados orientado a objetos. Mas, por muitas razões, a maioria delas não-técnicas, um banco de dados orientado a objetos não é uma escolha possível para aplicações corporativas. O resultado é uma conexão confusa com um banco de dados relacional. Vamos falar a verdade: modelos de objetos e modelos relacionais não se encaixam. O resultado é a complexidade de muitos dos padrões de mapeamento O/R que descrevo.

O *Módulo Tabela* (134) representa um meio-termo interessante entre esses dois pólos. Ele pode tratar melhor a lógica de domínio do que os *Roteiros de Transação* (120). Além disso, embora não possa se comparar a um verdadeiro *Modelo de Domínio* (126) para tratar lógica de domínio complexa, ele se encaixa verdadeiramente bem com um banco de dados relacional – e muitas outras coisas também. Se você tiver um ambiente como o .NET, onde muitas ferramentas orbitam em torno do todo poderoso *Conjunto de Registros* (473), então o *Módulo Tabela* (134) funciona muito bem, aproveitando os pontos fortes dos bancos de dados relacionais e ainda representando uma razoável fatoração da lógica do domínio.

Analisando esse argumento, vemos que as ferramentas que você tem também afetam sua arquitetura. Às vezes você pode escolher as ferramentas baseado na arquitetura e, em teoria, é isso o que você tem de fazer. Na prática, entretanto, muitas vezes você tem que adaptar sua arquitetura às suas ferramentas. Dos três padrões, o *Módulo Tabela* (134) é aquele cuja estrela mais brilha quando você tem ferramentas que se adaptam a ele. É uma escolha particularmente apropriada para ambientes .NET, já que tanto da plataforma é centrado ao redor do *Conjunto de Registros* (473).

Se você leu a discussão sobre a lógica do domínio no Capítulo 2, muito disso parecerá familiar. Ainda assim, vale a pena repetir essa discussão aqui, porque realmente penso que esta é a decisão central. Daqui, prosseguimos para a camada do banco de dados, mas agora as decisões são moldadas pelo contexto da sua escolha da lógica do domínio.

Descendo para a Camada de Dados

Uma vez que você tenha escolhido sua camada de domínio, tem que descobrir como conectá-la às suas fontes de dados. Suas decisões são baseadas na sua escolha da camada de domínio. Por esse motivo, abordarei essa questão em seções separadas, dirigidas por essa escolha.

Camada de Dados para o *Roteiro de Transação* (120)

Os *Roteiros de Transação* (120) mais simples contêm sua própria lógica de banco de dados, mas eu evito isso mesmo nos casos mais simples. Separar o banco de dados delimita duas partes que fazem sentido separadas, de modo que faço a separação mesmo nas aplicações mais simples. Os padrões de bancos de dados a serem escolhidos aqui são o *Gateway de Linhas de Dados* (158), e o *Gateway de Tabela de Dados* (151).

A escolha entre os dois depende muito dos recursos da sua plataforma de implementação e para onde você espera que a aplicação vá no futuro. Com um *Gateway de Linhas de Dados* (158), cada registro é lido para um objeto com uma interface clara e explícita. Com um *Gateway de Tabela de Dados* (151), você pode ter menos código para escrever, uma vez que você não precisa de todo o código de acesso para chegar aos dados, mas você acaba com uma interface muito mais implícita que se baseia no acesso a uma estrutura do tipo conjunto de registros que é pouco mais que um mapa.

A decisão-chave, entretanto, recai sobre o resto da sua plataforma. Se você tem uma plataforma que fornece várias ferramentas que trabalham bem com um *Conjunto de Registros* (473), especialmente ferramentas de interface com o usuário ou conjuntos de registros transacionais desconectados, faz com que você se incline decididamente na direção de um *Gateway de Tabela de Dados* (151).

Você normalmente não precisa de nenhum dos outros padrões de mapeamento O/R neste contexto. As questões referentes ao mapeamento estrutural estão praticamente ausentes aqui uma vez que a estrutura em memória mapeia tão bem para a estrutura do banco de dados. Você poderia considerar a utilização de uma *Unidade de Trabalho* (187), mas normalmente é fácil rastrear o que mudou no roteiro. Você não precisa se preocupar com a maior parte das questões relacionadas à concorrência, porque o roteiro freqüentemente corresponde quase que exatamente a uma transação do sistema.Você pode, portanto, simplesmente encapsular todo o roteiro em uma única transação. A exceção usual é quando uma requisição extrai dados para editá-los, e a próxima requisição tenta gravar as alterações. Neste caso, o *Bloqueio Offline Otimista* (392) é quase sempre a melhor escolha. Ele não apenas é mais fácil de implementar, como também geralmente satisfaz as expectativas dos usuários e evita o problema de uma sessão pendurada deixar todo tipo de recurso bloqueado.

Camada de Dados para o *Módulo Tabela* (134)

A principal razão para se escolher um *Módulo Tabela* (134) é a presença de um bom *framework* baseado em *Conjuntos de Registros* (473). Neste caso, você irá querer um padrão de mapeamento de banco de dados que funcione bem com um *Conjunto de Registros* (473), e isso o leva inexoravelmente na direção de um *Gateway de Tabela de Dados* (151). Esses dois padrões se ajustam tão bem que parecem ter sido feitos um para o outro.

De fato, com este padrão, não há mais nada que você precise acrescentar no lado da fonte de dados. Nos melhores casos, o *Conjunto de Registros* (473) tem algum tipo de mecanismo de controle de concorrência embutido, o que efetivamente o torna uma *Unidade de Trabalho* (187), reduzindo ainda mais a perda de cabelos.

Camada de Dados para o *Modelo de Domínio* (126)

Agora as coisas ficam interessantes. Por muitos motivos, a grande fraqueza do *Modelo de Domínio* (126) é que a conexão ao banco de dados é complicada. O grau de complicação depende da complexidade deste padrão.

Se o seu *Modelo de Domínio* (126) for simples, digamos uma ou duas dúzias de classes que sejam bastante semelhantes às tabelas no banco de dados, então um *Registro Ativo* (165) faz sentido. Se você quiser desacoplar um pouco as coisas, você pode usar para isso tanto um *Gateway de Tabela de Dados* (151) quanto um *Gateway de Linha de Dados* (158). Quer você separe, quer não, qualquer das alternativas não é muito complicada.

Como as coisas se tornam mais complicadas, você precisará considerar a utilização de um *Mapeador de Dados* (170). Esta é a abordagem que promete manter seu *Modelo de Domínio* (126) tão independente quanto possível de todas as outras camadas. Mas o *Mapeador de Dados* (170) é também o mais complicado padrão para implementar. A menos que você tenha uma equipe forte ou consiga encontrar algumas simplificações que tornem o mapeamento mais fácil, eu recomendaria fortemente o uso de uma ferramenta de mapeamento.

Uma vez que você tenha escolhido o *Mapeador de Dados* (170), a maioria dos padrões na seção de mapeamento O/R se aplicam. Em particular, recomendo de todo o coração a *Unidade de Trabalho* (187), que funciona como um ponto central para o controle da concorrência.

A Camada de Apresentação

Por diversos motivos, a apresentação é relativamente independente da escolha das camadas mais baixas. Sua primeira questão é entre fornecer uma interface rica de cliente ou uma interface HTML em um *browser*. Um cliente rico lhe dará uma interface com o usuário mais bonita, mas, neste caso, você precisa de um certo controle sobre a distribuição de seus clientes. Minha preferência é usar um navegador HTML se for possível, e um cliente rico em caso contrário. Clientes ricos normalmente demandarão um esforço maior de programação, mas isso ocorre porque eles tendem a ser mais sofisticados, nem tanto devido às complexidades inerentes à tecnologia.

Não explorei nenhum padrão de cliente rico neste livro, de modo que se você escolher um, não tenho realmente nada mais a dizer.

Se você escolher o caminho HTML, tem que decidir como estruturar sua aplicação. Certamente, recomendo o padrão *Modelo Vista Controlador* (315) como a fundação do seu projeto. Feito isso, você ainda tem duas decisões a tomar, uma para o controlador e outra para a vista.

Suas ferramentas podem muito bem fazer a escolha para você. Se você usar o Visual Studio, a maneira mais fácil é o *Controlador de Página* (318) e a *Vista Padrão* (333). Se você usar Java, tem uma variedade de *frameworks* Web a considerar. No momento, o Struts é uma escolha popular, o que o levará a um *Controlador Frontal* (328) e a uma *Vista Padrão* (333).

Se for possível uma maior liberdade na escolha, eu recomendaria um *Controlador de Página* (318) se o seu *site* for mais orientado a documentos, especialmente se você tiver uma mistura de páginas estáticas e dinâmicas. A navegação e interface com o usuário mais complexas levam à direção de um *Controlador Frontal* (328).

No lado da vista, a escolha entre a *Vista Padrão* (333) e a *Vista de Transformação* (343) depende de se sua equipe usa páginas servidoras ou XSLT na programação. As *Vistas Padrão* (333) têm no momento a dianteira, embora eu goste da facilidade de testes da *Vista de Transformação* (343). Se você tiver necessidade de mostrar um mesmo *site* com múltiplas aparências e funcionalidades, deve considerar a utilização de uma *Vista em Duas Etapas* (347).

O modo como você se comunica com as camadas inferiores depende do tipo de camada e se elas estarão sempre no mesmo processo. Minha preferência é fazer com que tudo rode em um único processo, se possível – desta maneira, você não tem que se preocupar com as lentas chamadas interprocessos. Se você não puder fazer isso, deve encapsular sua camada de domínio com uma *Fachada Remota* (368) e usar um *Objeto de Transferência de Dados* (380) para se comunicar com o servidor Web.

Alguns Conselhos Específicos para Determinadas Tecnologias

Na maior parte deste livro, tenho tentado trazer a experiência comum de criar projetos em várias plataformas diferentes. A experiência com Forte, CORBA e Smalltalk se traduz muito efetivamente no desenvolvimento com Java e .NET. A única razão pela qual tenho me concentrado em ambientes Java e .NET é que eles parecem ser as plataformas mais comuns para o desenvolvimento de aplicações corporativas no futuro. (Ainda que eu fosse gostar se as linguagens de *script* dinamicamente tipadas, em especial Python e Ruby, tivessem uma chance.)

Nesta seção, quero aplicar os conselhos acima nestas duas plataformas. Tão logo tenha feito isso, contudo, corro o risco de atribuir a mim mesmo um prazo de validade. As tecnologias mudam muito mais rapidamente do que estes padrões, de modo que enquanto você lê lembre-se de que estou escrevendo no início de 2002, quando todos estão dizendo que a recuperação econômica chegará logo.

Java e J2EE

Atualmente o grande debate no mundo Java é exatamente o quão valiosos são os Enterprise Java Beans. Depois de tantas "versões preliminares" finais quanto concertos de despedida do *The Who*, a especificação EJB 2.0 finalmente apareceu. Mas você não *precisa* de EJB para criar uma boa aplicação J2EE, apesar do que os fornecedores EJB lhe dizem. Você pode fazer bastante com os velhos e bons objetos Java (POJOs)* e JDBC.

As alternativas de projeto para J2EE variam em função dos padrões que você estiver usando e, mais uma vez, começam pela lógica do domínio.

Se você usar um *Roteiro de Transação* (120) sobre alguma das formas de um *Gateway* (436), a abordagem usual com EJBs, atualmente, é usar *session beans* como um *Roteiro de Transação* (120) e *entity beans* como um *Gateway de Linha de Dados* (158). Esta é uma arquitetura bastante razoável se a sua lógica de domínio for suficientemente modesta. Entretanto, um problema com tal abordagem usando *beans* é que é difícil se livrar do servidor EJB se você descobrir que não precisa dele e não quiser gastar com taxas de licenças. A abordagem não-EJB consiste de um POJO para o *Roteiro de Transação* (120) sobre um *Gateway de Linha de Dados* (158) ou um *Gateway de Tabela de Dados* (151). Se os conjuntos de linhas do JDBC 2.0 obtiverem maior aceitação, esta será uma razão para usá-los como um *Conjunto de Dados* (473) e isso leva a um *Gateway de Tabela de Dados* (151). Se você não estiver seguro sobre a utilização de EJBs, pode usar a abordagem não-EJB e encapsular os *entity beans* com os *session beans* atuando como fachadas remotas (368).

Se você estiver usando um *Modelo de Domínio* (126), a ortodoxia corrente é usar *entity beans*. Se o seu *Modelo de Domínio* (126) for bastante simples e tiver uma boa correspondência com as tabelas no banco de dados, isso faz bastante sentido e seus *entity beans* serão então *Registros Ativos* (165). Ainda é uma boa prática encapsular seus *entity beans* com *session beans* agindo como *Fachadas Remotas* (368) (embora você também possa pensar na *Persistência Gerenciada pelo Conteiner* (CMP**) como um *Mapeador de Dados* (170)). Entretanto, se o seu *Modelo de Domínio* (126) for mais complexo, você irá querer que ele seja inteiramente independente da estrutura do EJB de modo que você possa gravar, executar e testar sua lógica de domínio sem ter que lidar com os caprichos do contâiner EJB. Nesse contexto, eu usaria POJOs para o *Modelo do Domínio* (126) e os encapsularia com *session beans* agindo como *Fachadas Remotas* (368). Se você escolher não usar EJBs, eu executaria toda a aplicação no servidor Web e evitaria chamadas remotas entre a apresentação e o domínio. Se você estiver usando um *Modelo de Domínio* (126) POJO, eu também usaria POJOs para os *Mapeadores de Dados* (170) – seja usando uma ferramenta para o mapeamento O/R, seja fazendo eu mesmo, se eu achasse que conseguiria.

* N. de R.T.: *Plain and Old Java Objects.*

** N. de R.T.: *Container Managed Persistence*.

Se você usar *entity beans* em qualquer contexto, evite dar a eles uma interface remota. Nunca entendi o motivo de dar a um *entity bean* uma interface remota. Os *entity beans* são normalmente usados como *Modelos de Domínio* (126) ou como *Gateways de Linhas de Dados* (158). Em ambos os casos, eles precisam de uma interface de granularidade baixa para executar bem esses papéis. No entanto, como espero ter introduzido em sua mente, uma interface remota deve sempre ter uma granularidade alta. Por esse motivo, mantenha seus *entity beans* apenas como locais. (A exceção a isso é o padrão *Entity Composto* de [Alur *et al.*], o qual é uma forma diferente de usar *entity beans* e que, ainda assim, não me parece muito útil.)

No presente momento, o *Módulo Tabela* (134) não é muito comum no mundo Java. Será interessante ver se mais ferramentas surgirão em torno do conjunto de linhas (*row set*) JDBC – se isso acontecer, este padrão pode se tornar uma abordagem viável. Neste caso, a abordagem POJO se adapta melhor, embora você também possa encapsular o *Módulo Tabela* (134) com *session beans* atuando como *Fachadas Remotas* (368) e retornando *Conjuntos de Registros* (473).

.NET

Observando o .NET, o Visual Studio e a história do desenvolvimento de aplicações no mundo Microsoft, o padrão dominante é o *Módulo Tabela* (134). Ainda que os fanáticos por objetos costumem dizer que isso significa apenas que os seguidores da Microsoft não conhecem orientação a objetos, o fato é que o *Módulo Tabela* (134) oferece um meio-termo valioso entre um *Roteiro de Transação* (120) e um *Modelo de Domínio* (126), com um impressionante conjunto de ferramentas que tiram proveito do onipresente conjunto de dados atuando como um *Conjunto de Registros* (473).

A conseqüência disso é que o *Módulo Tabela* (134) tem que ser a escolha padrão para esta plataforma. Em verdade, não vejo sentido algum em usar *Roteiros de Transação* (120), exceto nos casos mais simples e, mesmo assim, eles deveriam ser bem-comportados e retornar conjuntos de dados.

Isso não significa que você não possa usar um *Modelo de Domínio* (126). Com certeza, você pode construir um *Modelo de Domínio* (126) tão facilmente em .NET quanto em qualquer outro ambiente OO. Todavia, as ferramentas não lhe dão a ajuda extra que elas dão em *Módulos Tabela* (134), de modo que eu toleraria uma complexidade maior antes de sentir a necessidade de mudar para um *Modelo de Domínio* (126).

Existe atualmente uma publicidade excessiva referente ao uso de serviços Web em .NET. No entanto, eu não usaria serviços Web dentro de uma aplicação. Eu os usaria, assim como em Java, na camada de apresentação para permitir a integração de aplicações. Não existe nenhuma boa razão em uma aplicação .NET para separar o servidor Web e a lógica do domínio em processos distintos. Desse modo, a *Fachada Remota* (368) é menos útil aqui.

Procedimentos Armazenados

Ocorrem normalmente uma quantidade razoável de discussões a respeito de procedimentos armazenados. Freqüentemente, eles são o modo mais rápido de fazer as coisas, uma vez que eles rodam no mesmo processo do seu banco de dados e, desta forma, reduzem as preguiçosas chamadas remotas. Entretanto, a maioria dos ambientes de procedimentos armazenados não lhe fornece bons mecanismos para a estruturação de seus procedimentos armazenados. Além disso, os procedimentos ar-

mazenados o manterão preso a um vendedor de banco de dados específico. (Uma boa maneira de evitar estes problemas é a abordagem Oracle de permitir que você rode aplicações Java dentro do processo do seu banco de dados. Isso é o equivalente a colocar toda sua camada de lógica do domínio dentro do banco de dados. Por enquanto, isso ainda o deixa um pouco amarrado a um vendedor, mas pelo menos reduz os custos de portar o banco de dados para um outro gerenciador.)

Por questões de modularidade e portabilidade, muitas pessoas evitam usar procedimentos armazenados para a lógica de negócio. Tendo a me alinhar com esta visão a menos que haja um grande ganho de desempenho a ser obtido, o que, para ser sincero, freqüentemente ocorre. Nesse caso, pego um método da camada do domínio e, alegremente, o transformo em um procedimento armazenado. Faço isso apenas em áreas com claros problemas de desempenho, tratando-o como um passo da otimização, e não como um princípio arquitetural. ([Nilsson] apresenta um bom argumento a favor do uso de procedimentos armazenados mais extensamente.)

Um modo comum de usar procedimentos armazenados é para controlar o acesso a um banco de dados, segundo as diretrizes de um *Gateway de Tabelas de Dados* (151). Não tenho nenhuma opinião fechada quanto a fazer isso ou não, e, pelo que tenho visto, não há fortes argumentos de nenhum dos lados. De todo modo, prefiro usar os mesmos padrões para isolar o acesso ao banco de dados, quer este seja controlado por procedimentos armazenados ou por SQL normal.

Serviços Web

Quando escrevo isso, o consenso entre os especialistas é o de que os serviços Web farão da reutização uma realidade e tirarão os integradores de sistemas do mercado, mas não estou assim tão entusiasmado. Os serviços Web não desempenham um papel muito importante na utilização destes padrões, porque sua finalidade é a integração de aplicações, e não a construção das mesmas. Você não deveria tentar separar uma aplicação única em serviços Web que conversem entre si, a menos que você realmente precise fazê-lo. Em vez disso, construa sua aplicação e exponha as diversas partes dela como serviços Web, tratando esses serviços como *Fachadas Remotas* (368). Acima de tudo, não deixe que todo o burburinho sobre o quão fácil é criar serviços Web faça você esquecer a Primeira Lei do Projeto de Objetos Distribuídos (página 100-101).

Embora a maioria dos exemplos publicados que eu tenha visto use serviços Web de forma síncrona, em vez de usá-los como uma chamada RPC XML, prefiro usá-los de forma assíncrona e baseados em mensagens. Embora não apresente aqui nenhum padrão para isso (este livro já é grande o suficiente do jeito que está), espero que possamos ver nos próximos anos alguns padrões para a troca assíncrona de mensagens.

Outros Esquemas de Camadas

Construí minha discussão em torno de três camadas principais, mas a minha abordagem para a criação de camadas não é a única que faz sentido. Outros bons livros sobre arquitetura têm esquemas de camadas, e todos eles têm valor. Vale a pena olhar esses outros esquemas e compará-los àqueles que apresento aqui. Você pode achar que eles fazem mais sentido para a sua aplicação.

O primeiro destes padrões é o que chamarei de modelo de Brown, discutido em [Brown *et al.*] (veja a Tabela 8.1). Esse modelo tem cinco camadas: apresentação, controlador/mediador, domínio, mapeamento de dados e fonte de dados. Basicamente ele coloca camadas adicionais de mediação entre as três camadas básicas. O controlador/mediador realiza a mediação entre as camadas de apresentação e domínio, enquanto que a camada de mapeamento de dados realiza a mediação entre a camada de domínio e a camada da fonte de dados.

Acho que as camadas de mediação são úteis durante parte do tempo mas não durante todo o tempo, de forma que as descrevo em termos de padrões. O *Controlador de Aplicação* (360) é o mediador entre a apresentação e o domínio, e o *Mapeador de Dados* (170) é o mediador entre a camada de dados e o domínio. Para organizar este livro, descrevi o *Controlador de Aplicação* (360) na seção da apresentação (Capítulo 14) e o *Mapeador de Dados* (170) na seção da camada de dados (Capítulo 10).

Para mim então, a adição das freqüentemente (mas nem sempre) úteis camadas de mediação, representa um suplemento opcional no projeto. Minha abordagem é sempre pensar nas três camadas básicas, verificar se alguma delas está se tornando muito complexa e, em caso afirmativo, acrescentar a camada de mediação para separar a funcionalidade.

Outro bom esquema de camadas para J2EE aparece nos padrões CoreJ2EE [Alur *et al.*] (veja a Tabela 8.2). Aqui as camadas são: cliente, apresentação, negócio, integração e recursos. Correspondências simples existem para as camadas de negócio e integração. A camada de recursos compreende os serviços externos aos quais a camada de integração se conecta. A principal diferença é que eles separam a camada de apresentação entre a parte que roda no cliente (cliente) e a parte que roda em um servidor (apresentação). Essa separação é muitas vezes útil, mas, novamente, ela não é necessária durante todo o tempo.

A arquitetura Microsoft DNA [Kirtland] define três camadas: apresentação, negócio e acesso a dados, que correspondem quase que diretamente às três camadas que uso aqui (veja a Tabela 8.3). A maior mudança ocorre no modo pelo qual os dados são passados das camadas de acesso a dados. No Microsoft DNA, todas as camadas operam sobre conjuntos de registros que resultam de consultas SQL executadas pela camada de acesso a dados. Isso introduz um aparente acoplamento, uma vez que, tanto a camada de negócio quanto a de apresentação conhecem o banco de dados.

O modo como vejo isso é que no DNA o conjunto de registros age como um *Objeto de Transferência de Dados* (380) entre as camadas. A camada de negócio pode modificar o conjunto de registros durante o trajeto deste para a camada de apresentação ou mesmo ela própria criar um conjunto de registros (o que é raro). Embora

Tabela 8.1 As Camadas de Brown

Brown	Fowler
Apresentação	Apresentação
Controlador/mediador	Apresentação (*Controlador de Aplicação* (360))
Domínio	Domínio
Mapeamento de dados	Fonte de dados (*Mapeador de Dados* (170))
Fonte de dados	Fonte de dados

Tabela 8.2 Camadas CoreJ2EE

CoreJ2EE	Fowler
Cliente	Apresentação que roda no cliente (p.ex., sistemas de clientes ricos)
Apresentação	Apresentação que roda no servidor (p.ex., processos que tratam requisições HTTP, páginas servidoras)
Negócio	Domínio
Integração	Fonte de Dados
Recursos	Recursos externos com os quais a fonte de dados se comunica

Tabela 8.3 Camadas Microsoft DNA

Microsoft DNA	Fowler
Apresentação	Apresentação
Negócio	Domínio
Acesso a dados	Fonte de Dados

esta forma de comunicação seja, por muitos motivos, incômoda, ela tem a grande vantagem de permitir que a apresentação use controles ligados aos dados (*data aware*) na interface com o usuário, mesmo com dados que tenham sido modificados pela camada de negócio.

Neste caso, a camada de domínio é estruturada na forma de *Módulos Tabela* (134), e a camada de dados usa *Gateways de Tabelas de Dados* (151).

[Marinescu] tem cinco camadas (veja a Tabela 8.4). A apresentação é dividida em duas camadas, refletindo a separação de um *Controlador de Aplicação* (380). O domínio é também dividido, com uma *Camada de Serviço* (141) construída sobre um *Modelo de Domínio* (126), refletindo o senso comum de dividir uma camada de domínio em duas partes. Esta é uma abordagem comum, reforçada pelas limitações de EJB como um *Modelo de Domínio* (126) (veja a página 127-128).

A idéia de separar uma camada de serviços de uma camada de domínio é baseada na separação da lógica do fluxo de trabalho da lógica de domínio pura. A camada de serviços tipicamente inclui lógica específica de um único caso de uso e também alguma comunicação com outras infra-estruturas, como a de mensagens. Ter ou não camadas separadas de serviço e de domínio é motivo de algum debate. Tendo a encarar isso como algo ocasionalmente útil, em vez de obrigatório, mas vários projetistas que respeito não concordam comigo.

[Nilsson] usa um dos esquemas de camadas mais complexos (veja a Tabela 8.5). O mapeamento para este esquema de camadas se torna um pouco mais complexo pelo fato de Nilsson usar extensivamente procedimentos armazenados e encorajar, por motivos de desempenho, a inclusão de lógica de domínio neles. Não me sinto confortável em colocar lógica de domínio em procedimentos armazenados, porque isso pode tornar uma aplicação muito mais difícil de manter. Ocasionalmente, entretanto, essa é uma técnica de otimização valiosa. As camadas de procedimentos armazenados de Nilsson contêm tanto a fonte de dados quanto a lógica do domínio.

Tabela 8.4 Camadas de Marinescu

Marinescu	Fowler
Apresentação	Apresentação
Aplicação	Apresentação (Controlador de Aplicação (360))
Serviços	Domínio (Camada de Serviço (141))
Domínio	Domínio (Modelo de Domínio (126))
Persistência	Fonte de Dados

Tabela 8.5 Camadas de Nilsson

Nilsson	Fowler
Consumidor	Apresentação
Auxiliar do Consumidor	Apresentação (Controlador de Aplicação (360))
Aplicação	Domínio (Camada de Serviço (141))
Domínio	Domínio (Modelo de Domínio (126))
Acesso Persistente	Fonte de Dados
Procedimentos Armazenados Públicos	Fonte de Dados (pode incluir algum domínio)
Procedimentos Armazenados Privados	Fonte de Dados (pode incluir algum domínio)

Assim como [Marinescu], Nilsson usa camadas de aplicação e de domínio separadas para a lógica de domínio. Ele sugere que você pode pular a camada de domínio em um sistema pequeno, o que é semelhante à minha visão de que um *Modelo de Domínio* (126) tem menos valor para sistemas pequenos.

Esta página foi deixada em branco intencionalmente.

Parte II

Os Padrões

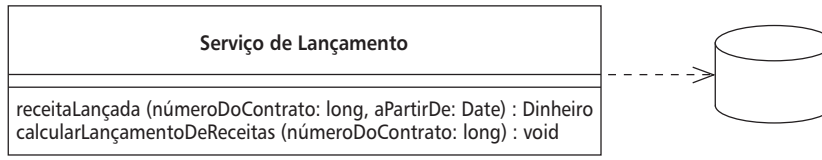
Esta página foi deixada em branco intencionalmente.

CAPÍTULO 9

Padrões de Lógica de Domínio

Roteiro de Transação (Transaction Script)

Organiza a lógica de negócio em procedimentos onde cada procedimento lida com uma única solicitação da apresentação.



A maioria das aplicações podem ser percebidas como uma série de transações. Uma transação pode enxergar alguma informação organizada de alguma forma particular, outra transação pode fazer alterações nesta informação. Cada interação entre um sistema cliente e um sistema servidor contém uma certa quantidade de lógica. Em alguns casos isso pode ser tão simples quanto mostrar informações armazenadas no banco de dados. Em outros, pode envolver muitos passos de validações e cálculos.

Um *Roteiro de Transação* organiza toda esta lógica primariamente como um único procedimento, fazendo chamadas diretas ao banco de dados ou usando um fino envoltório para acesso ao banco de dados. Cada transação terá seu próprio *Roteiro de Transação*, embora subtarefas comuns possam ser separadas em procedimentos.

Como Funciona

Com o *Roteiro de Transação*, a lógica do domínio é organizada primariamente pelas transações que você executa usando o sistema. Se a sua necessidade for reservar o quarto de um hotel, a lógica para verificar a disponibilidade de quartos, calcular taxas e atualizar o banco de dados é encontrada dentro do procedimento Reservar Quarto de Hotel.

Para casos simples não há muito a dizer sobre como organizar isso. É claro que, assim como com qualquer outro programa, você deve estruturar o código em módulos de uma maneira que faça sentido. A menos que a transação seja especialmente complicada, isso não será um grande desafio. Um dos benefícios desta abordagem é que você não precisa se preocupar com o que outras transações estão fazendo. Sua tarefa é ler a entrada de dados, inquirir o banco de dados, trabalhar sobre os dados e gravar seus resultados no banco de dados.

Onde você deve colocar o *Roteiro de Transação* dependerá de como você organiza suas camadas. Pode ser em uma página servidora, um *script* CGI ou um objeto de sessão distribuído. Minha preferência é separar os *Roteiros de Transação* o máximo que você puder. Coloque-os no mínimo em sub-rotinas distintas; melhor ainda, coloque-os em classes separadas daquelas que lidam com a apresentação e a fonte de dados. Além disso, não permita chamadas dos *Roteiros de Transação* para qualquer lógica de apresentação. Isso facilitará a modificação do código e os testes dos *Roteiros de Transação*.

Você pode organizar seus *Roteiros de Transação* em classes de duas maneiras distintas. A mais comum é ter diversos Roteiros de Transação em uma única classe, onde cada classe define um assunto comum para *Roteiros de Transação* relacionados. Isso é simples e direto e é a melhor aposta para a maior parte dos casos. A outra maneira é ter cada *Roteiro de Transação* em sua própria classe (Figura 9.1), usando o padrão

Comando [Gang of Four]. Neste caso você define um supertipo para seus comandos que define algum método para execução no qual a lógica dos *Roteiros de Transação* se encaixa. A vantagem desta abordagem é que ela permite manipular instâncias de roteiros como objetos em tempo de execução, embora eu raramente tenha visto necessidade de fazer isso com os tipos de sistemas que usam *Roteiros de Transação* para organizar a lógica de domínio. É claro que, em muitas linguagens, você pode ignorar as classes completamente e usar apenas funções globais. Contudo, você descobrirá muitas vezes que instanciar um novo objeto ajuda em questões relacionadas a *threads*, pois torna mais fácil isolar os dados.

Eu uso o termo *Roteiro de Transação*, porque na maior parte do tempo você terá um *Roteiro de Transação* para cada transação no banco de dados. Isso não é uma regra para 100% dos casos, mas é verdade em uma primeira aproximação.

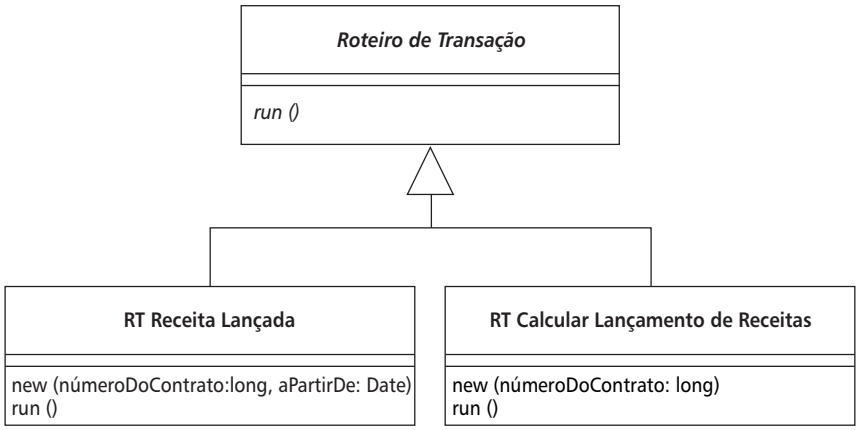


Figura 9.1 Usando comandos para *Roteiro de Transação*.

Quando Usá-lo

A beleza do *Roteiro de Transação* é sua simplicidade. Organizar a lógica desta maneira é natural para aplicações com apenas uma pequena quantidade de lógica, e envolve muito pouco *overhead*, seja no desempenho, seja na compreensão.

No entanto, à medida que a lógica de negócio fica mais complicada, torna-se cada vez mais difícil mantê-la em um estado que possa ser caracterizado como um bom projeto. Um problema específico a observar é a duplicação de código entre transações. Já que o ponto central é lidar com uma transação por roteiro, qualquer código em comum tende a ser duplicado.

Uma fatoração cuidadosa pode minorar muitos destes problemas, no entanto domínios de negócio mais complexos precisam construir um *Modelo de Domínio* (126). Um *Modelo de Domínio* (126) lhe dará muito mais opções na estruturação do código, aumentando a legibilidade e diminuindo a duplicação.

É difícil quantificar o ponto de corte entre as duas estratégias, especialmente quando você está mais familiarizado com um padrão do que com outro. Você pode refatorar o projeto de um *Roteiro de Transação* para o de um *Modelo de Domínio* (126), mas é uma alteração mais difícil do que de outra forma precisaria ser. Uma escolha inicial acertada é, portanto, a melhor maneira de seguir adiante.

Entretanto, ainda que você se torne um fanático por objetos, não descarte os *Roteiros de Transação*. Há muitos problemas simples por aí, e uma solução simples lhe levará adiante mais rápido.

O Problema do Lançamento de Receitas

Para este padrão, e outros que falam de lógica do domínio, usarei o mesmo problema como ilustração. Para não ter de descrever o problema diversas vezes, eu o farei apenas uma vez aqui.

O lançamento de receitas é um problema comum em sistemas de negócios. O problema central é quando você pode realmente contabilizar em seus livros o dinheiro que você recebe. Se eu lhe vender uma xícara de café, esta é uma questão simples: eu lhe dou o café, pego seu dinheiro e contabilizo o dinheiro nos livros no mesmo nanossegundo. Entretanto, para muitas coisas isso fica complicado. Digamos que você me dê um adiantamento para que eu fique disponível durante esse ano. Ainda que você me pague uma remuneração ridícula hoje, posso não poder lançá-la imediatamente em meus livros, porque o serviço será executado no decorrer de um ano. Uma solução poderia ser contabilizar apenas uma duodécima parte dessa remuneração para cada mês do ano, desde que você pudesse rescindir o contrato após um mês, quando percebesse que escrever atrofiou minhas habilidades de programação.

As regras para lançamento de receitas são muitas, variadas e voláteis. Algumas são estabelecidas por regulamentos, algumas por padrões profissionais e outras por políticas da companhia. Rastrear receitas acaba sendo um problema bastante complexo.

Não quero me aprofundar nesta complexidade neste momento, assim, em vez disso, vamos imaginar uma companhia que venda três tipos de produtos: processadores de texto, bancos de dados e planilhas. De acordo com as regras, quando você assina um contrato de um processador de texto, pode lançar as receitas nos livros imediatamente. Se for uma planilha, pode colocar um terço hoje, um terço em sessenta dias e um terço em noventa dias. Se for um banco de dados, pode colocar um terço hoje, um terço em trinta dias e um terço em sessenta dias. Não há base para estas regras além da minha própria imaginação febril. Disseram-me que as regras reais são tão racionais quanto essas.

Exemplo: Lançamento de Receitas (Java)

Este exemplo usa dois roteiros de transação: um para calcular o lançamento de receitas para um contrato e outro para dizer quanto da receita de um contrato foi lançado até uma determinada data. A estrutura do banco de dados tem três tabelas: uma para os produtos, uma para os contratos e uma para os lançamentos de receitas.

```
CREATE TABLE produtos (ID int primary key, nome varchar, tipo varchar)
CREATE TABLE contratos (ID int primary key, produto int, receita decimal, dataAssinatura date)
CREATE TABLE lançamentosDeReceitas (contrato int, quantia decimal, lançadaEm date,
                                    PRIMARY KEY (contrato, lançadaEm))
```

O primeiro roteiro calcula o total de lançamentos até uma determinada data. Posso fazer isso em duas etapas: na primeira seleciono as linhas apropriadas na tabela de lançamentos de receitas; na segunda, somo as quantias.

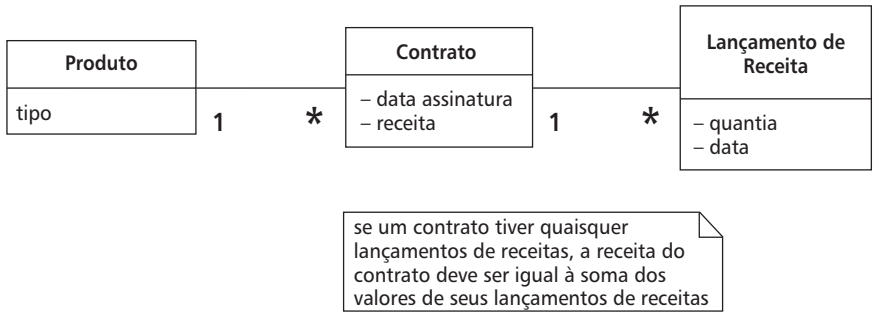


Figura 9.2 Um modelo conceitual para o lançamento simplificado de receitas. Cada contrato tem múltiplos lançamentos de receitas que indicam quando as várias parcelas da receita devem ser lançadas.

Muitos projetos de *Roteiros de Transação* têm roteiros que operam diretamente no banco de dados, inserindo código SQL no procedimento. Aqui estou usando um simples *Gateway de Tabela de Dados* (151) para encapsular as consultas SQL. Já que este exemplo é tão simples, usarei um único *gateway* para todas as tabelas. Posso definir um método de busca apropriado no *gateway*.

```
class Gateway...

    public ResultSet buscaLançamentoPor (long IDdoContrato, MfDate aPartirDe) throws SQLException {
        PreparedStatement dec = db.prepareStatement (declaraçãoBuscaLançamentos);
        dec.setLong(1, IDdoContrato);
        dec.setDate(2, aPartirDe.toSqlDate( ));
        ResultSet resultado = dec.executeQuery( ));
        return resultado;
    }
    private static final String declaraçãoBuscaLançamentos =
        "SELECT quantia " +
        "  FROM lançamentosDeReceitas " +
        "  WHERE contrato = ? AND lançadoEm <= ?";
    private Connection db;
```

Eu então uso o roteiro para somar baseado no conjunto de resultados devolvido pelo *gateway*.

```
class ServiçoDeLançamento...

    public Dinheiro receitaLançada(long númeroDoContrato, MfDate aPartirDe) {
        Dinheiro resultado = Dinheiro.dollars(0);
        try {
            ResultSet rs = db.buscaLançamentoPor(númeroDoContrato, aPartirDe);
            while (rs.next()) {
                resultado = resultado.add(Dinheiro.dollars(rs.getBigDecimal("quantia")));
            }
            return resultado;
        } catch (SQLException e) {throw new AplicationException(e);
        }
    }
```

Quando o cálculo for simples como este, você pode substituir o roteiro em memória por uma chamada a uma declaração SQL que use uma função de agregação para somar as quantias.

Para calcular as lançamentos de receitas em um contrato existente, uso uma separação similar. O roteiro no serviço se encarrega da lógica de negócio.

```
class ServiçoDeLançamento...

    public void calcularLançamentoDeReceitas(long númeroDoContrato) {
        try {
            ResultSet contratos = db.encontrarContrato(númeroDoContrato);
            contratos.next();
            Dinheiro receitaTotal = Dinheiro.dollars(contratos.getBigDecimal("receita"));
            MfDate dataLançamento = new MfDate(contratos.getDate("dataAssinatura"));
            String tipo = contratos.getString("tipo");
            if(tipo.equals("P")) // planilha {
                Dinheiro alocação[] = receitaTotal.alocar(3);
                db.inserirLançamento
                    (númeroDoContrato,alocação[0],dataLançamento);
                db.inserirLançamento
                    (númeroDoContrato,alocação[1],dataLançamento.adicionaDias(60));
                db.inserirLançamento
                    (númeroDoContrato,alocação[2],dataLançamento.adicionaDias(90));
            } else if (tipo.equals("T")) // processador de textos {
                db.inserirLançamento(númeroDoContrato, receitaTotal, dataLançamento);
            } else if (tipo.equals("B")) // banco de dados {
                Dinheiro alocaçao[] = receitaTotal.alocar(3);
                db.inserirLançamento
                    (númeroDoContrato,alocaçao[0],dataLançamento);
                db.inserirLançamento
                    (númeroDoContrato,alocaçao[1],dataLançamento.adicionaDias(30));
                db.inserirLançamento
                    (númeroDoContrato,alocaçao[2],dataLançamento.adicionaDias(60));
            }
        } catch (SQLException e) { throw new AplicationException(e);
        }
    }
```

Perceba que estou usando *Dinheiro* (455) para executar a alocação. Ao dividir uma quantia em três é fácil perder alguns centavos.

O *Gateway de Tabela de Dados* (151) fornece suporte ao SQL. Primeiro há um método de busca para um contrato.

```
class Gateway...

    public ResultSet encontrarContrato(long IDdoContrato) throws SQLException {
        PreparedStatement dec = db.prepareStatement (declaraçãoBuscaContrato);
        dec.setLong(1, IDdoContrato);
        ResultSet resultado = dec.executeQuery( ));
        return resultado;
    }
    private static final String declaraçãoBuscaContrato =
        "SELECT * " +
        "  FROM contratos c, produtos p " +
        "  WHERE c.ID = ? AND c.produto = p.ID";
```

Em segundo lugar, há um encapsulamento para a inserção.

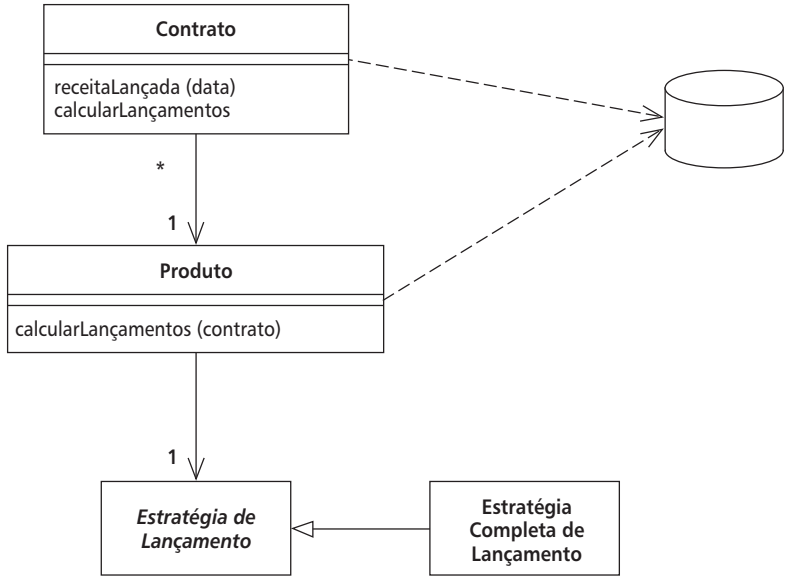
```
class Gateway...

    public void inserirLançamento(long IDdoContrato, Dinheiro quantia, MfDate aPartirDe)
        throws SQLException {
        PreparedStatement dec = db.prepareStatement(declaraçãoInserirLançamento);
        dec.setLong(1, IDdoContrato);
        dec.setBigDecimal(2, quantia.quantia());
        dec.setDate(3, aPartirDe.toSqlDate());
        dec.executeUpdate();
    }
    private static final String declaraçãoInserirLançamento =
        "INSERT INTO lançamentosDeReceitas VALUES (?, ?, ?)";
```

Em um sistema Java, o serviço de lançamento poderia ser uma classe regular ou um *session bean*.

Quando você comparar este padrão ao exemplo no *Modelo do Domínio* (126), a menos que sua mente seja tão torcida quanto a minha, provavelmente pensará que este é muito mais simples. O pior cenário a se considerar é quando as regras ficam mais complicadas. As regras típicas de lançamento de receitas são muito complexas, variando não somente com o produto, mas também com a data (se o contrato foi assinado antes de 15 de abril, esta regra se aplica...). É difícil manter um projeto coerente com o *Roteiro de Transação*, quando as coisas ficam tão complicadas, e este é o motivo pelo qual fanáticos por objetos, como eu, preferem usar o *Modelo de Domínio* (126) nestas circunstâncias.

Modelo de Domínio (Domain Model)

Um modelo de objetos do domínio que incorpora tanto o comportamento quanto os dados.



No pior caso, a lógica de negócios pode ser muito complexa. As regras e a lógica de negócio descrevem muitos casos diferentes e muitas variações de comportamento. Os objetos foram projetados justamente para lidar com essa complexidade. Um *Modelo de Domínio* cria uma rede de objetos interconectados em que cada um representa algum conceito significativo, que pode ser tão grande quanto uma corporação, ou tão pequeno quanto uma linha em um formulário de pedidos.

Como Funciona

Colocar um *Modelo de Domínio* em uma aplicação envolve a inserção de uma camada inteira de objetos que modelam a área de negócios com a qual você está trabalhando. Você encontrará objetos que representam os dados do negócio e objetos que capturam as regras usadas pelo negócio. Em sua maior parte, dados e processos são combinados para juntar os processos aos dados com os quais eles trabalham.

Um modelo de domínio OO freqüentemente se parece com um modelo de banco de dados, embora muitas diferenças persistam. Um *Modelo de Domínio* mistura dados e processos, tem atributos multivalorados, uma complexa rede de associações e usa herança.

Em conseqüência, consigo enxergar dois estilos de *Modelo de Domínio*. Um *Modelo de Domínio* simples se parece muito com o projeto de um banco de dados, tendo, na maior parte das vezes, um objeto de domínio para cada tabela do banco de dados. Um *Modelo de Domínio* rico pode parecer diferente do projeto de um banco de dados,

com herança, estratégias e outros padrões [Gang of Four], e complexas redes de pequenos objetos interconectados. Um *Modelo de Domínio* rico é melhor para a lógica mais complexa, mas é mais difícil de mapear para o banco de dados. Um *Modelo de Domínio* simples pode usar o *Registro Ativo* (165), enquanto que um *Modelo de Domínio* rico requer o *Mapeador de Dados* (170).

Uma vez que o comportamento do negócio é sujeito a muitas alterações, é importante poder modificar, criar e testar esta camada facilmente. Em conseqüência, quanto menor o acoplamento do *Modelo de Domínio* a outras camadas no sistema, melhor. Você perceberá que uma linha mestra de muitos padrões de camadas é manter tão poucas dependências entre o modelo de domínio e outras partes do sistema quanto for possível.

Existem vários escopos diferentes que você pode usar com um *Modelo de Domínio*. O caso mais simples é uma aplicação monousuário na qual todo o conjunto de objetos é lido de um arquivo e trazido para a memória. Uma aplicação *desktop* pode funcionar desta maneira, porém isso é menos comum para um sistema de informações multicamadas, simplesmente porque há objetos demais. Trazer todos os objetos para a memória demora muito e consome memória demais. A beleza dos bancos de dados orientados a objetos é que eles dão a impressão de fazer isso enquanto movem objetos entre a memória e o disco.

Sem um banco de dados OO, você mesmo tem que fazer isso. Normalmente, uma sessão envolve trazer para a memória todos os objetos nela envolvidos. Isso certamente não significa trazer todos os objetos e, na maior parte das vezes, nem mesmo todas as classes. Assim, se você estiver olhando um grupo de contratos, poderia trazer apenas os produtos referenciados pelos contratos dentro do seu conjunto de trabalho. Se você estiver apenas efetuando cálculos nos contratos e em objetos de lançamento de receitas, não precisa trazer para a memória nenhum objeto produto. O que exatamente você trará para a memória é controlado pelos seus objetos de mapeamento de banco de dados.

Se você precisar do mesmo conjunto de objetos entre chamadas ao servidor, você tem de salvar o estado do servidor em algum lugar, o que é o assunto da seção sobre gravação do estado do servidor (página 93-94).

Uma preocupação comum com a lógica de domínio são os objetos de domínio inchados. Quando você criar uma tela para manipular pedidos perceberá que uma parte do comportamento do pedido só é necessário para esta tela. Se você colocar essas responsabilidades no pedido, o risco é que a classe Pedido se torne grande demais porque está cheia de responsabilidades que só são usadas em um único caso de uso. Essa preocupação leva as pessoas a considerarem se uma parte da responsabilidade é geral, em cujo caso ela deve ficar na classe Pedido ou específica, em cujo caso ela deve ficar em alguma classe de uso específico, que poderia ser um *Roteiro de Transação* (120) ou talvez a própria apresentação.

O problema em separar comportamento específico de uma situação de uso é que isso pode levar a duplicação. O comportamento separado do pedido é difícil de achar, de modo que as pessoas tendem a não encontrá-lo e o duplicam. A duplicação pode levar rapidamente a mais complexidade e inconsistência, mas descobri que inchaços ocorrem com muito menor freqüência do que esperado. Se realmente eles ocorrerem, são relativamente fáceis de ver e não são difíceis de consertar. Meu conselho é não separar comportamento específico de uma situação de uso. Coloque tudo no objeto que parece ser o mais adequado. Conserte o inchaço quando, e se, ele se tornar um problema.

Implementação Java

As discussões sobre o desenvolvimento de um *Modelo de Domínio* em J2EE são sempre acaloradas. Muitos dos materiais de ensino e dos livros introdutórios de J2EE sugerem que você use *entity beans* para desenvolver um modelo de domínio, mas há sérios problemas com esta abordagem, pelo menos com a especificação corrente (2.0).

Os *entity beans* são mais úteis quando você usa CMP (Container Managed Persistence). Na verdade, eu diria que não há muito sentido em usar *entity beans* sem CMP. Entretanto, CPM é uma forma limitada de mapeamento objeto-relacional, e não consegue suportar muitos dos padrões que você precisa em um *Modelo de Domínio* rico.

Os *entity beans* não devem ser reentrantes. Isso quer dizer que, se você fizer uma chamada de um *entity bean* para outro objeto, esse outro objeto (ou qualquer objeto que ele chamar) não pode chamar o primeiro *entity bean*. Um *Modelo de Domínio* rico freqüentemente usa reentrância, de modo que esta é uma limitação importante. Ela fica pior pelo fato de que é difícil localizar comportamento reentrante. Em conseqüência, algumas pessoas dizem que um *entity bean* nunca deve chamar outro. Embora isso evite reentrância, diminui significativamente as vantagens de usar um *Modelo de Domínio*.

Um *Modelo de Domínio* deve usar objetos de granularidade baixa, com interfaces de granularidade baixa. Os *entity beans* podem ser remotos (antes da versão 2.0 eles tinham que ser). Se você tiver objetos remotos com interfaces de granularidade baixa, obterá desempenhos péssimos. Você pode evitar facilmente esse problema, usando apenas interfaces locais para seus *entity beans* em um *Modelo de Domínio*.

Para rodar os *entity beans* você precisa de um contâiner e um banco de dados conectado. Isso irá aumentar o tempo de criação e também o de teste, já que estes têm que ser executados sobre um banco de dados. Os *entity beans* são também complicados para depurar.

A alternativa é usar objetos Java normais, embora isso muitas vezes cause uma reação de surpresa – é incrível como muitas pessoas pensam que você não pode rodar objetos Java normais em um contâiner EJB. Cheguei à conclusão de que as pessoas se esquecem dos objetos Java normais porque eles não têm um nome bonito. É por isso que, durante a preparação para uma palestra em 2000, Rebecca Parsons, John Mackenzie e eu lhes demos um: POJO (Plain Old Java Objects – velhos e bons objetos Java). Um modelo de domínio POJO é fácil de ser reunido e construído, pode rodar e ser testado fora de um contâiner EJB e é independente do EJB (talvez seja por isso que os vendedores de EJB não lhe encorajem a usá-los).

Minha visão global é que usar *entity beans* como um *Modelo de Domínio* funciona se você tiver uma lógica de domínio bastante simples. Neste caso, você pode construir um *Modelo de Domínio* que tenha um relacionamento simples com o banco de dados: na sua maior parte apenas um *entity bean* por tabela do banco de dados. Se você tiver uma lógica de domínio mais rica, com herança, estratégias e outros padrões mais sofisticados, você estará melhor servido com um modelo de domínio POJO e um *Mapeador de Dados* (170), usando uma ferramenta comercial ou uma camada desenvolvida em casa.

A maior frustração para mim no uso de EJB é que considero um *Modelo de Domínio* complicado o suficiente para lidar, e quero ficar tão independente quanto possível dos detalhes do ambiente de implementação. O EJB faz com que você tenha de pensar nele quando pensa no *Modelo de Domínio*, o que significa que tenho que me preocupar tanto com o domínio quanto com o ambiente EJB.

Quando usá-lo

Se o *como* para um *Modelo de Domínio* é difícil porque é um assunto extenso, o *quando* é difícil devido tanto à falta de clareza quanto à simplicidade do conselho. Tudo se resume à complexidade do comportamento do seu sistema. Se você tiver regras de negócio complicadas e em constante mudança, envolvendo validação, cálculos e derivações, é bastante possível que você queira um modelo de objetos que lide com elas. Por outro lado, se você tiver verificações simples para fazer (*not null*) e apenas algumas somas para calcular, um *Roteiro de Transação* (120) é uma aposta melhor.

Um fator importante aqui é o quão confortável a equipe de desenvolvimento se sente com objetos de domínio. Aprender como projetar e usar um *Modelo de Domínio* é um exercício significativo – que já proporcionou vários artigos sobre a "mudança de paradigma" representado pelo uso de objetos. Certamente é necessário prática e orientação para que alguém se acostume a um *Modelo de Domínio*, mas descobri que, uma vez acostumadas, muito poucas pessoas querem voltar a um *Roteiro de Transação* (120), a não ser para problemas muito simples.

Se você estiver usando um *Modelo de Domínio*, minha primeira escolha para a interação com o banco de dados é o *Mapeador de Dados* (170). Isso manterá seu *Modelo de Domínio* independente e é a melhor abordagem para lidar com casos em que o *Modelo de Domínio* e o esquema do banco de dados divergem.

Quando você usa o *Modelo de Domínio*, talvez possa considerar a *Camada de Serviço* (141) para dar ao seu *Modelo de Domínio* uma API mais separada.

Leitura Adicional

Qualquer livro sobre projeto OO falará sobre *Modelos de Domínio*, já que a maior parte do que as pessoas chamam de desenvolvimento OO é centrado sobre seu uso.

Se você estiver procurando por um livro introdutório sobre projeto OO, atualmente meu favorito é [Larman]. Para exemplos de *Modelo de Domínio*, veja [Fowler AP]. [Hay] também dá bons exemplos em um contexto relacional. Para construir um bom *Modelo de Domínio*, você deve ter uma boa compreensão sobre os conceitos relacionados ao uso de objetos. Para isso sempre gostei de [Martin e Odell]. Para compreender os padrões que você verá em um *Modelo de Domínio rico*, ou em qualquer outro sistema OO, você deve ler [Gang of Four].

Eric Evans está escrevendo atualmente um livro [Evans] sobre a criação de *Modelos de Domínio*. Enquanto escrevo isto, vi apenas um manuscrito inicial, mas ele parece bastante promissor.

Exemplo: Lançamento de Receita (Java)

Uma das maiores frustrações em descrever um *Modelo de Domínio* é o fato de que qualquer exemplo que eu mostre tem de ser necessariamente simples para que você possa entendê-lo. Contudo, esta simplicidade esconde a força do *Modelo de Domínio*. Você só aprecia essa força quando tem um domínio realmente complicado.

Mas ainda que o exemplo possa não fazer justiça ao porquê de você querer um *Modelo de Domínio*, pelo menos lhe dará uma boa idéia de como ele se parece. Por esta razão, estou usando o mesmo exemplo que usei para o *Roteiro de Transação* (120), uma pequena questão de lançamento de receitas.

Uma coisa a ser percebida imediatamente é que cada classe, neste pequeno exemplo (Figura 9.3) contém tanto comportamento quanto dados. Mesmo a modesta classe Lançamento de Receitas contém um método simples para descobrir se o valor do objeto pode ser lançado em determinada data.

```
public class LançamentoDeReceitas...

    private Dinheiro quantia;
    private MfDate data;
    public LançamentoDeReceitas(Dinheiro quantia, MfDate data) {
        this.quantia = quantia;
        this.data = data;
    }
    public Dinheiro lerQuantia() {
        return quantia;
    }
    boolean éLançávelEm(MfDate aPartirDe) {
        return aPartirDe.after(data) || aPartirDe.equals(data);
    }
```

Calcular o quanto de uma receita já foi lançada em determinada data, envolve tanto a classe Contrato quanto o LançamentoDeReceitas.

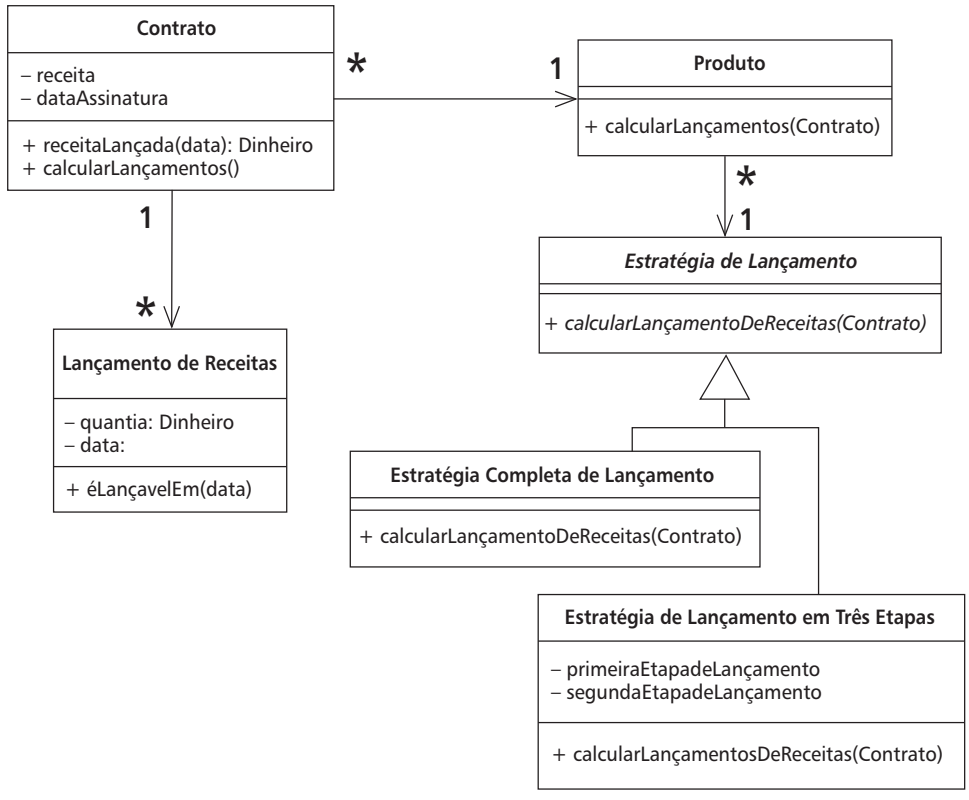


Figura 9.3 Diagrama de classes para as classes do exemplo de um *Modelo de Domínio*.

```
class Contrato …

    private List lançamentoDeReceitas = new ArrayList();
    public Dinheiro receitaLançada(MfDate aPartirDe) {
        Dinheiro resultado = Dinheiro.dollars(0);
        Iterator it = lançamentoDeReceitas.iterator();
        while (it.hasNext()) {
            LançamentoDeReceitas r = (LançamentoDeReceitas) it.next();
            if (r.éLançávelEm(aPartirDe))
                resultado = resultado.add(r.lerQuantia());
        }
        return resultado;
    }
```

Uma coisa comum que você descobre em modelos de domínio é como múltiplas classes têm de interagir para executar até as tarefas mais simples. Isso é o que leva à reclamação freqüente de que em programas OO perde-se muito tempo passeando pelas classes, tentando encontrá-las. Esta reclamação tem bastante fundamento. O benefício surge quando a decisão sobre se algo é lançável ou não em uma certa data fica mais complexa, e outros objetos precisam saber. Manter o comportamento no objeto que precisa saber evita a duplicação de código e reduz o acoplamento entre os diferentes objetos.

Olhar para o cálculo e a criação destes objetos de lançamento de receitas reforça o conceito de sistemas constituídos de muitos objetos pequenos. Neste caso, o cálculo e a criação começam com o cliente e são encaminhados através do produto para uma hierarquia de estratégias. O padrão hierarquia [Gang of Four] é um padrão OO conhecido que permite combinar um grupo de operações em uma pequena hierarquia de classes. Cada instância de um produto é conectada a uma única instância de estratégia que determina o algoritmo usado para calcular o lançamento de receitas. Neste caso, temos duas subclasses de estratégia de lançamento para os dois diferentes casos. A estrutura do código tem esta aparência:

```
class Contrato...

    private Produto produto;
    private Dinheiro receita;
    private MfDate dataAssinatura;
    private Long id;
    public Contrato(Produto produto, Dinheiro receita, MfDate dataAssinatura) {
        this.produto = produto;
        this.receita = receita;
        this.dataAssinatura = dataAssinatura;
    }

class Produto...

    private String nome;
    private EstratégiaLançamento estratégiaLançamento;
    public Produto(String nome, EstratégiaLançamento estratégiaLançamento) {
        this.nome = nome;
        this.estratégiaLançamento = estratégiaLançamento;
    }
    public static Produto novoProcessadorDeTexto(String nome) {
        return new Produto(nome, new EstratégiaLançamentoTotal());
```

```
    }
    public static Produto novaPlanilha(String nome) {
        return new Produto(nome, new EstratégiaLançamentoEmTrêsEtapas(60, 90));
    }
    public static Produto novoBancoDeDados(String nome) {
        return new Produto(nome, new EstratégiaLançamentoEmTrêsEtapas(30, 60));
    }
}
abstract class EstratégiaLançamento {

    abstract void calcularLançamentoDeReceitas(Contrato contrato); }

class EstratégiaLançamentoTotal extends EstratégiaLançamento {
    void calcularLançamentoDeReceitas(Contrato contrato) {
        contrato.adicionarLançamentoDeReceitas(new LançamentoDeReceitas(contrato.lerReceita(),
                                                                        contrato.lerDataAssinatura()));
    }

class EstratégiaLançamentoEmTrêsEtapas extends EstratégiaLançamento {

    private int primeiraEtapaDeLançamento;
    private int segundaEtapaDeLançamento;
    public EstratégiaLançamentoEmTrêsEtapas(int primeiraEtapaDeLançamento,
                    int segundaEtapaDeLançamento)
    {
        this.primeiraEtapaDeLançamento = primeiraEtapaDeLançamento;
        this.segundaEtapaDeLançamento = segundaEtapaDeLançamento;
    }
    void calcularLançamentoDeReceitas(Contrato contrato) {
        Dinheiro[] alocação = contrato.lerReceita().alocar(3);
        contrato.adicionarLançamentoDeReceitas(new LançamentoDeReceitas
                (alocação[0], contrato.lerDataAssinatura()));
        contrato.adicionarLançamentoDeReceitas(new LançamentoDeReceitas
                (alocação[1], contrato.lerDataAssinatura().adicionaDias(primeiraEtapaDeLançamento)));
        contrato.adicionarLançamentoDeReceitas(new LançamentoDeReceitas
                (alocação[2], contrato.lerDataAssinatura().adicionaDias(segundaEtapaDeLançamento)));
    }
```

O grande valor das estratégias é que elas fornecem pontos de conexão bem limitados para estender a aplicação. Adicionar um novo algoritmo de lançamento de receita exige apenas a criação de uma nova subclasse que sobrescreva o método *calcularLançamentoDeReceitas*. Isso torna fácil estender o comportamento algorítmico da aplicação.

Quando você cria produtos, os conecta com os objetos de estratégia pertinentes. Farei isto em meu código de teste.

```
class Teste...

    private Produto processador = Produto.novoProcessadorDeTexto("Thinking Word");
    private Produto planilha = Produto.novaPlanilha("Thinking Calc");
    private Produto db = Produto.novoBancoDeDados("Thinking DB");
```

Quando tudo estiver pronto, não é necessário conhecer as subclasses de estratégias para calcular os lançamentos.

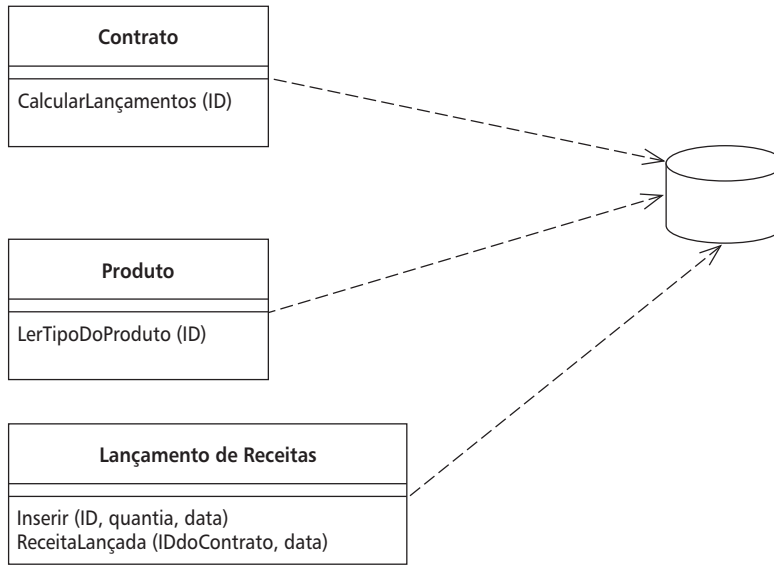
```
class Contrato...

    public void calcularLançamentos () {
        produto.calcularLançamentoDeReceitas (this);
    }

class Produto...

    void calcularLançamentoDeReceitas (Contrato contrato) {
        estratégiaLançamento.calcularLançamentoDeReceitas (contrato);
    }
```

A prática OO de mandar mensagens de um objeto para outro não só move o comportamento para o objeto mais qualificado a lidar com ele, mas também resolve muito do comportamento condicional. Você perceberá que não há instruções condicionais neste cálculo. Você estabelece o caminho da decisão quando cria os produtos com a estratégia apropriada. Uma vez que tudo esteja no lugar, os algoritmos simplesmente seguem o trilho. Modelos de domínio funcionam muito bem quando você tem expressões condicionais parecidas, porque estas expressões podem ser fatoradas para a própria estrutura do objeto. Isso tira a complexidade dos algoritmos e a coloca nos relacionamentos entre objetos. Quanto mais parecida for a lógica, mais vezes você encontrará a mesma rede de relacionamentos usada em diferentes partes do sistema. Qualquer algoritmo que seja dependente do tipo de cálculo de lançamento pode seguir esta rede específica de objetos.

Perceba neste exemplo que não mostrei nada sobre como os objetos são trazidos do banco de dados ou nele armazenados. Há algumas razões para isso. Em primeiro lugar, mapear um *Modelo de Domínio* em um banco de dados é sempre um pouco difícil, de modo que me acovardei e não forneci um exemplo. Em segundo lugar, sobre muitos aspectos, a questão central a respeito de um *Modelo de Domínio* é esconder o banco de dados, tanto das camadas superiores quanto das pessoas trabalhando no próprio *Modelo de Domínio*. Assim, escondê-lo aqui reflete a realidade do que é programar neste ambiente.

Módulo Tabela (Table Module)

Uma única instância que trata a lógica de negócio para todas as linhas de uma tabela ou visão de um banco de dados.



Um dos conceitos-chave da orientação a objetos é empacotar os dados com o comportamento que os usam. A abordagem orientada a objetos tradicional é baseada em objetos identificados, junto com as diretrizes de *Modelo de Domínio* (126). Assim, se tivermos uma classe Empregado, qualquer instância dela corresponde a um empregado em particular. Este esquema funciona bem porque, uma vez que tenhamos uma referência para um empregado, podemos executar operações neste empregado, seguir relacionamentos e obter informações sobre ele.

Um dos problemas com o *Modelo de Domínio* (126) é a interface com bancos de dados relacionais. Sob vários aspectos, esta abordagem trata o banco de dados relacional como uma tia louca trancada no sótão sobre a qual ninguém quer falar. Em conseqüência, muitas vezes você precisa de uma considerável ginástica de programação para extrair e gravar dados no banco de dados, realizando a transformação entre as duas diferentes representações dos dados.

Um *Módulo Tabela* organiza a lógica do domínio com uma classe por tabela do banco de dados, e uma única instância de uma classe contém os vários procedimentos que atuarão sobre os dados. A distinção primária do *Modelo de Domínio* (126) é que, se você tiver muitos pedidos, um *Modelo de Domínio* (126) terá um objeto Pedido por pedido, enquanto que um *Módulo Tabela* terá um objeto para tratar todos os pedidos.

Como Funciona

A força do *Módulo Tabela* é que ele permite empacotar juntos os dados e o comportamento e, ao mesmo tempo, tira proveito do poder de um banco de dados relacional. Na superfície, o *Módulo Tabela* se parece muito com um objeto regular. A principal di-

ferença é que ele não tem o conceito de identidade dos objetos com os quais está trabalhando. Assim, se você quiser obter o endereço de um empregado, deve usar um método como `umMóduloEmpregado.lerEndereço(long IDdoEmpregado)`. Cada vez que você quiser fazer algo com um empregado específico, tem que passar algum tipo de identificação deste empregado. Freqüentemente, esta será a chave primária usada no banco de dados.

Normalmente, você usa o *Módulo Tabela* com uma estrutura de suporte orientada a tabelas. Os dados tabulares são geralmente o resultado de uma chamada SQL e são armazenados em um *Conjunto de Registros* (473) que imita uma tabela SQL. O *Módulo Tabela* dá a você uma interface explicitamente baseada em métodos que age sobre esses dados. Juntar o comportamento à tabela dá a você muitos dos benefícios do encapsulamento, na medida em que o comportamento está próximo aos dados sobre os quais ele atua.

Com freqüência, você precisará do comportamento de múltiplos *Módulos Tabela* de modo a executar algum trabalho útil. Muitas vezes você encontrará múltiplos *Módulos Tabela* operando sobre o mesmo *Conjunto de Registros* (473) (Figura 9.4).

O caso mais simples é a utilização de um *Módulo Tabela* para cada tabela no banco de dados. No entanto, se você tiver consultas e visões interessantes no banco de dados, pode ter também *Módulos Tabelas* para elas.

O *Módulo Tabela* pode ser uma instância ou uma coleção de métodos estáticos. A vantagem de uma instância é que ela permite a você inicializar o *Módulo Tabela* com um conjunto de dados existente, talvez o resultado de uma consulta ao banco. Você pode então usar a instância para manipular as linhas no conjunto de registros. As instâncias também tornam possível a utilização de herança, de modo que podemos escrever um módulo de um contrato urgente que contenha comportamento adicional ao do contrato normal.

O *Módulo Tabela* pode incluir consultas ao banco como métodos fábrica. A alternativa é um *Gateway para Tabelas de Dados* (151), mas uma desvantagem disso é ter

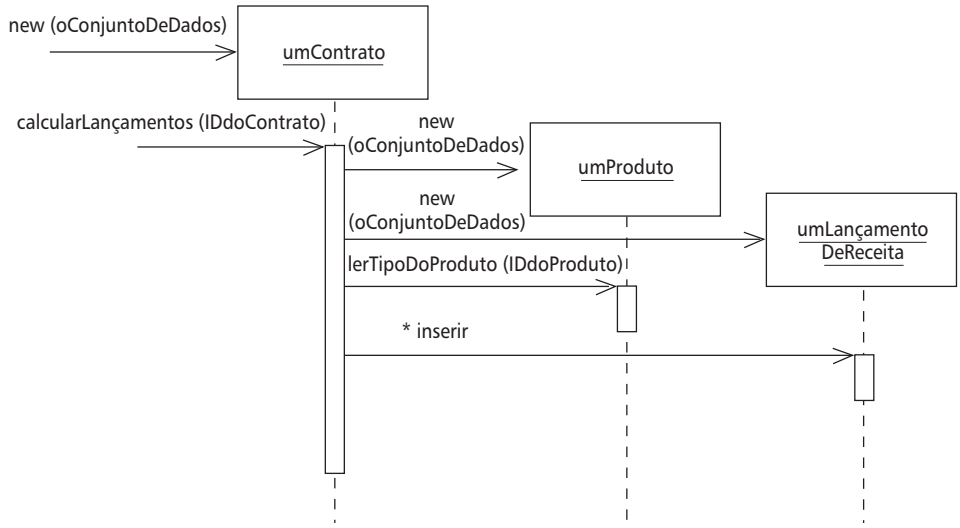


Figura 9.4 Diversos *Módulos Tabela* podem colaborar com um único *Conjunto de Registros* (473).

uma classe e mecanismo *Gateway para Tabelas de Dados* (151) extra no projeto. A vantagem é que você pode usar um único *Módulo Tabela* nos dados de diferentes fontes de dados, uma vez que você estará usando *Gateways para Tabelas de Dados* (151) diferentes para cada fonte de dados.

Quando você usa um *Gateway para Tabelas de Dados* (151), a aplicação primeiro usa o *Gateway para Tabelas de Dados* (151) para inserir dados em um *Conjunto de Registros* (473). Você então cria um *Módulo Tabela* com o *Conjunto de Registros* (473) como um argumento. Se você precisar do comportamento de múltiplos *Módulos Tabela*, pode criá-los com o mesmo *Conjunto de Registros* (473). O *Módulo Tabela* pode então executar a lógica do negócio no *Conjunto de Registros* (473) e passar esse *Conjunto de Registros* (473) modificado para a apresentação para que seja exibido e editado usando componentes que trabalham com tabelas. Esses componentes não sabem dizer se os *conjuntos de registros* vieram diretamente do banco de dados relacional ou se um *Módulo Tabela* manipulou os dados no caminho. Após as modificações na interface com o usuário, o conjunto de dados volta para o *Módulo Tabela* para validação antes de ser gravado no banco de dados. Um dos benefícios deste estilo é que você pode testar o *Módulo Tabela* criando um *Conjunto de Registros* (473) em memória, sem ir ao banco de dados.

A palavra "tabela" no nome do padrão sugere que você tenha um *Módulo Tabela* por tabela no banco de dados. Embora isso seja verdadeiro em uma primeira aproximação, não é completamente verdadeiro. Também é útil ter um *Módulo Tabela* para visões ou outras pesquisas comumente usadas. Na verdade, a estrutura do *Módulo Tabela* não depende da estrutura real das tabelas no banco de dados, e sim das tabelas virtuais percebidas pela aplicação, incluindo visões e pesquisas.

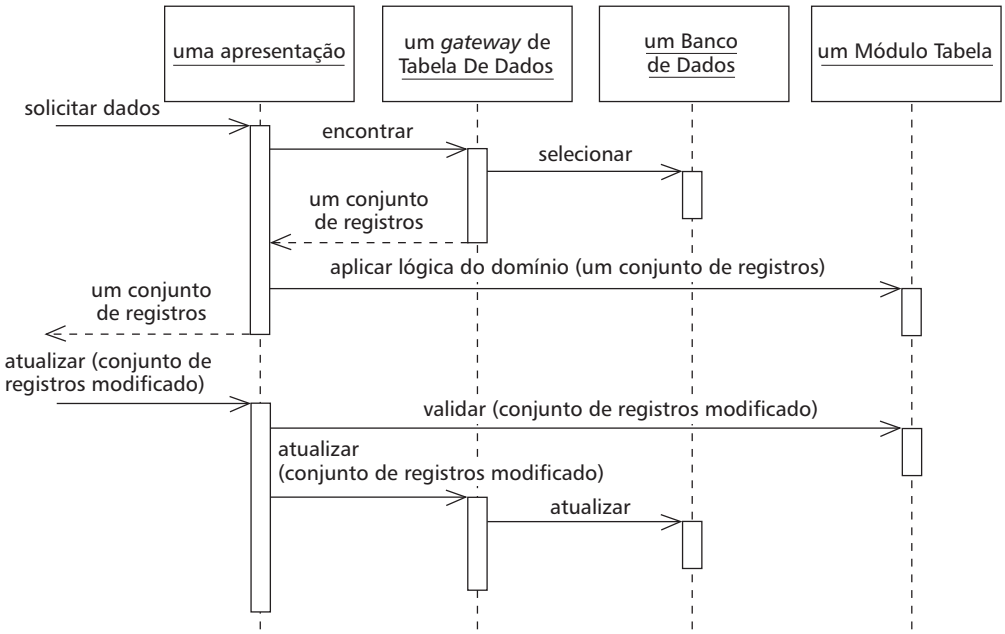


Figura 9.5 Interações típicas das camadas em torno de um *Módulo Tabela*.

Quando Usá-lo

O *Módulo Tabela* é fortemente baseado em dados orientados a tabelas, de modo que, obviamente, o seu uso faz sentido quando você está acessando dados tabulares usando um *Conjunto de Registros* (473). Além disso, o *Módulo Tabela* coloca essa estrutura de dados bem no centro do código, então é desejável ainda que o modo de acesso à estrutura de dados seja razoavelmente direta.

No entanto, o *Módulo Tabela* não dá a você todo o poder dos objetos na organização de lógica complexa. Você não pode ter relacionamentos diretos de instância com instância, e o polimorfismo não funciona muito bem. Assim, para lidar com lógica de domínio complexa, um *Modelo de Domínio* (126) é uma escolha melhor. Essencialmente, a escolha entre um e outro padrão é um compromisso entre a habilidade do *Modelo de Domínio* (126) em lidar com lógica complexa e a facilidade de integração do *Módulo Tabela* com suas estruturas de suporte orientadas a tabelas.

Se os objetos em um *Modelo de Domínio* (126) e as tabelas no banco de dados forem relativamente semelhantes, pode ser melhor usar um *Modelo de Domínio* (126) que use *Registros Ativos* (165). O *Módulo Tabela* funciona melhor do que a combinação *Modelo de Domínio* (126) e *Registros Ativos* (165) quando outras partes da aplicação forem baseadas em uma estrutura comum, baseada em tabelas. É por esse motivo que você não vê muito o *Módulo Tabela* em um ambiente Java, embora isso possa mudar na medida em que conjuntos de linhas se tornem mais largamente usados.

A situação mais conhecida em que me deparei com este padrão foi nos projetos utilizando o Microsoft COM. Em COM (e .NET) o *Conjunto de Registros* (473) é o repositório primário de dados em uma aplicação. Conjuntos de registros podem ser passados para a interface com o usuário, onde componentes *data aware* mostram as informações. As bibliotecas ADO da Microsoft fornecem um bom mecanismo para acessar os dados relacionais como conjuntos de registros. Nesta situação, o *Módulo Tabela* permite a você introduzir a lógica do negócio na aplicação de uma maneira bastante organizada, sem perder o modo como os diversos elementos atuam sobre os dados tabulares.

Exemplo: Lançamento de Receitas com um Módulo Tabela (C#)

É hora de revisitar o exemplo do lançamento de receitas que usei nos outros padrões para modelagem de domínio, desta vez com um *Módulo Tabela*. Apenas para recapitular, nossa missão é lançar receitas referentes a pedidos em que as regras de lançamento variam de acordo com o tipo do produto. Neste exemplo, temos regras diferentes para processadores de texto, planilhas e banco de dados.

O *Módulo Tabela* é baseado em um esquema de dados de algum tipo, normalmente um modelo de dados relacional (embora no futuro possamos ver um modelo XML usado de forma semelhante). Neste caso, usarei o esquema relacional da Figura 9.6.

As classes que manipulam estes dados têm quase que a mesma forma; há uma classe *Módulo Tabela* para cada tabela. Na arquitetura .NET, um objeto conjunto de dados (*data set*) fornece uma representação em memória de uma estrutura do banco de dados. Assim, faz sentido criar classes que operem sobre estes *data sets*. Cada classe *Módulo Tabela* tem um atributo do tipo tabela de dados (*data table*), que é a classe .NET correspondente a uma tabela dentro do *data set*. A capacidade de ler uma tabela é comum a todos os *Módulos Tabela* e por isso pode aparecer em um *Supertipo de Camada* (444).

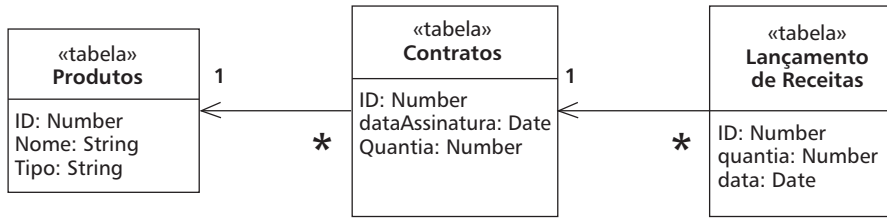


Figura 9.6 Esquema do banco de dados para o lançamento de receitas.

```
class MóduloTabela...

        protected DataTable tabela;
        protected MóduloTabela (DataSet ds, String nomeDaTabela) {
            tabela = ds.Tables[nomeDaTabela];
        }
```

O construtor da subclasse chama o construtor da superclasse com o nome correto da tabela.

```
class Contrato...

        public Contrato (DataSet ds): base (ds, "Contratos") {}
```

Isso permite a você criar um novo *Módulo Tabela* simplesmente passando o conjunto de dados (*data set*) para o construtor do Módulo Tabela

```
            contrato = new Contrato (dataset);
```

o qual mantém o código que cria o *data set* separado dos *Módulos Tabela*, seguindo as diretrizes do ADO.NET.

Uma característica útil é o indexador C#, que localiza uma linha específica no *data table*, dada a chave primária.

```
class Contrato...

        public DataRow this [long chave] {
        get {
            String filtro = String.format("ID = {0}", chave);
            return tabela.Select (filtro)[0];
            }
        }
```

A primeira parte da função calcula o lançamento da receita de um contrato, atualizando a tabela de lançamento de receitas de acordo. A quantia lançada depende do tipo de produto que temos. Já que este comportamento usa principalmente dados da tabela Contrato, decidi adicionar o método na classe Contrato.

```
class Contrato...

        public void CalcularLançamentos (long IDdoContrato) {
            DataRow linhaDoContrato = this[IDdoContrato];
            Decimal quantia = (Decimal)linhaDoContrato["quantia"];
```

```
        LançamentoDeReceitas lr = new LançamentoDeReceitas (tabela.DataSet);
        Produto produto = new Produto (tabela.DataSet);
        long IDdoProduto = lerIDdoProduto(IDdoContrato);
        if (produto.LerTipoDoProduto(IDdoProduto) == TipoDoProduto.PT) {
            lr.Inserir (IDdoContrato, quantia, (DateTime) LerDataAssinatura(IDdoContrato));
        } else if (produto.LerTipoDoProduto(IDdoProduto) == TipoDoProduto.P) {
            Decimal [] alocação = alocar(quantia, 3);
            lr.Inserir(IDdoContrato, alocação[0], (DateTime) LerDataAssinatura(IDdoContrato));
            lr.Inserir(IDdoContrato, alocação[1], (DateTime)
                LerDataAssinatura(IDdoContrato). adicionaDias(60));
            lr.Inserir(IDdoContrato, alocação[2], (DateTime)
                LerDataAssinatura(IDdoContrato). adicionaDias (90));
        } else if (produto.LerTipoDoProduto(IDdoProduto) == TipoDoProduto.BD) {
            Decimal [] alocação = alocar(quantia, 3);
            lr.Inserir(IDdoContrato, alocação[0], (DateTime) LerDataAssinatura(IDdoContrato));
            lr.Inserir(IDdoContrato, alocação[1], (DateTime)
                LerDataAssinatura(IDdoContrato). adicionaDias (30));
            lr.Inserir(IDdoContrato, alocação[2], (DateTime)
                LerDataAssinatura(IDdoContrato). adicionaDias (60));
        } else throw new Exception ("identificação inválida de produto");
    }
    private Decimal[] alocar (Decimal quantia, int quantos) {
        Decimal resultadoInferior = quantia / quantos;
        resultado inferior = Decimal.Round(resultadoInferior, 2);
        Decimal resultadoSuperior = resultadoInferior + 0.01m;
        Decimal[] resultados = new Decimal [quantos];
        int resto = (int) quantia % quantos;
        for (int i = 0; i < resto; i++) resultados[i] = resultadoSuperior;
        for (int i = resto; i < quantos; i++) resultados[i] = resultadoInferior;
        return resultados;
    }
```

Normalmente eu usaria a classe *Dinheiro* (455) aqui, mas para variar um pouco usarei dessa vez um decimal. Uso aqui um método de alocação similar ao que uso em *Dinheiro* (455).

Para isso funcionar, precisamos de parte do comportamento definido em outras classes. O produto precisa ser capaz de nos dizer de que tipo ele é. Podemos fazer isso com uma enumeração para o tipo do produto e um método de busca.

```
    public enum TipoDoProduto {PT, P, BD};

class Produto...

    public TipoDoProduto LerTipoDoProduto (long id) {
        String enumeração = (String) this[id]["tipo"];
        return (TipoDoProduto) Enum.Parse (typeof(TipoDoProduto), enumeração);
    }
```

`LerTipoDoProduto` encapsula os dados no *data table*. Há um bom argumento para fazer isso para todas as colunas de dados, em oposição a acessá-los diretamente como fiz com a quantia no contrato. Embora o encapsulamento geralmente seja uma boa coisa, não o uso aqui porque ele não combina com a suposição de um ambiente em que diferentes partes do sistema acessem o conjunto de dados diretamente. Não há encapsulamento quando o conjunto de dados se move para a interface com o usuário,

de modo que funções de acesso a colunas só fazem sentido quando há funcionalidade adicional a ser executada, tal como converter uma *string* em um tipo do produto.

Esta também é uma boa hora para mencionar que, embora eu esteja usando aqui um *data set* não tipado, porque estes são mais comuns em diferentes plataformas, há um argumento forte (página 473-474) para o uso de *data set* .NET fortemente tipados.

O outro comportamento adicional consiste em inserir um novo registro de lançamento de receitas.

```
class LançamentoDeReceitas...

       public long Inserir (long IDdoContrato, Decimal quantia, DateTime data) {
           DataRow novaLinha = tabela.NewRow( );
           long id = GetNextID ( );
           novaLinha["ID"] = id;
           novaLinha["IDdoContrato"] = IDdoContrato;
           novaLinha["quantia"] = quantia;
           novaLinha["data"] = String.Format("{0:s}", data);
           tabela.Rows.Add(novaLinha);
           return id;
       }
```

Uma vez mais, o ponto central deste método é menos encapsular os *datarows* e mais ter um método, em vez de diversas linhas de código repetidas.

A segunda parte da funcionalidade é somar todas as receitas lançadas em um contrato até uma determinada data. Uma vez que esta funcionalidade usa a tabela de lançamento de receitas, faz sentido definir o método lá.

```
class LançamentoDeReceitas...

       public Decimal ReceitaLançada (long IDdoContrato, DateTime aPartirDe) {
           String filtro = String.Format ("IDdoContrato = {0} AND data <= #{1:d}#", IDdoContrato,
aPartirDe);
           DataRow [] linhas = tabela.Select(filtro);
           Decimal resultado = 0m;
           foreach (DataRow linha in linhas) {
               resultado += (Decimal) linha["quantia"];
           }
           return resultado;
       }
```

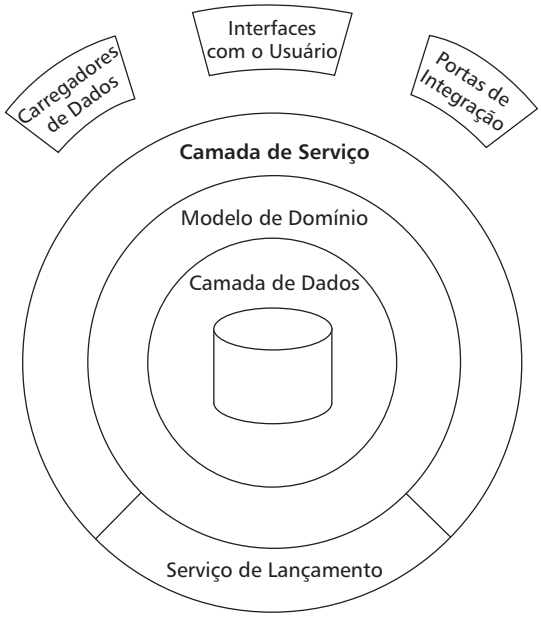
Este fragmento tira proveito da ótima característica do ADO.NET que lhe permite definir uma cláusula *where* e então selecionar um subconjunto da tabela de dados para manipular. De fato, você pode ir além e usar uma função agregada.

```
class LançamentoDeReceitas...

       public Decimal ReceitaLançada2 (long IDdoContrato, DateTime aPartirDe) {
           String filtro = String.Format ("IDdoContrato = {0} AND data <= #{1:d}#", IDdoContrato,
aPartirDe);
           String expressãoAComputar = "sum(quantia)";
           Object soma = tabela.Compute(expressãoAComputar, filtro);
           return (soma is System.DBNull) ? 0: (Decimal) soma);
       }
```

Camada de Serviço (Service Layer)

por Randy Stafford

Define os limites de uma aplicação com uma camada de serviços que estabelece um conjunto de operações disponíveis e coordena a resposta da aplicação em cada operação.



As aplicações corporativas normalmente requerem diferentes tipos de interfaces para os dados armazenados e para a lógica implementada: carregadores de dados, interfaces com usuários, *gateways* de integração e outros. Apesar de seus objetivos diferentes, estas interfaces freqüentemente requerem interações comuns com a aplicação para acessar e manipular os dados e invocar a lógica de negócio. As interações podem ser complexas, envolvendo transações por meio de múltiplos recursos e a coordenação de diversas respostas a uma ação. Codificar a lógica das interações separadamente em cada interface acarreta em muita duplicação de código.

Uma *Camada de Serviço* define a fronteira de uma aplicação [Cockburn PloP] e seu conjunto de operações disponíveis, a partir da perspectiva das camadas de interface dos clientes. Ela encapsula a lógica de negócio da aplicação, controlando as transações e coordenando as respostas na implementação de suas operações.

Como Funciona

Uma *Camada de Serviço* pode ser implementada de algumas formas diferentes, sem violar as características de definição expressas acima. As diferenças aparecem na alocação de responsabilidades por trás da interface da *Camada de Serviço*. Antes que eu me aprofunde nas diversas possibilidades de implementação, deixe-me apresentar alguns princípios básicos.

Tipos de "Lógica de Negócio" Assim como no *Roteiro de Transação* (120) e no *Modelo de Domínio* (126), a *Camada de Serviço* é um padrão para organizar a lógica do negócio. Muitos projetistas, incluindo a mim, gostam de dividir a "lógica de negócio" em dois tipos: "lógica do domínio", relacionada estritamente ao domínio do problema (tais como as estratégias para calcular o lançamento de receitas de um contrato) e "lógica da aplicação", relacionada às responsabilidades da aplicação [Cockburn UC] (tais como notificar os administradores de contratos e as aplicações integradas dos cálculos de lançamento de receitas). A lógica da aplicação é algumas vezes chamada de "lógica de fluxo de trabalho" (*workflow logic*), embora pessoas diferentes tenham interpretações diferentes para "fluxo de trabalho".

Os *Modelos de Domínio* (126) são preferíveis aos *Roteiros de Transação* (120) para evitar a duplicação da lógica de domínio e para gerenciar a complexidade usando padrões clássicos de projeto. Contudo, colocar lógica da aplicação em objetos de domínio puros traz algumas conseqüências indesejáveis. Primeiro, os objetos de domínio tornam-se menos reutilizáveis em outras aplicações se eles implementarem lógica específica da aplicação e dependerem de pacotes específicos da aplicação. Em segundo lugar, misturar ambos os tipos de lógica nas mesmas classes torna mais difícil reimplementar a lógica da aplicação, digamos, em uma ferramenta de fluxo de trabalho (*workflow*) se isso se tornar desejável. Por estas razões, a *Camada de Serviço* fatora cada tipo de lógica de negócio em uma camada separada, produzindo os benefícios costumeiros da disposição em camadas e tornando os objetos de domínio puros, mais reutilizáveis de uma aplicação para outra.

Variações de Implementação As duas variantes básicas de implementação são as abordagens da fachada de domínio e a do roteiro de operação. Na abordagem da **fachada de domínio** uma *Camada de Serviço* é implementada como um conjunto de fachadas finas sobre um *Modelo de Domínio* (126). As classes que implementam as fachadas não implementam nenhuma lógica de negócio. Em vez disso, o *Modelo de Domínio* (126) implementa toda a lógica de negócio. As fachadas finas estabelecem uma fronteira e um conjunto de operações por meio do qual as camadas clientes interagem com a aplicação, exibindo as características que definem uma *Camada de Serviço*.

Na abordagem **roteiro de operação**, uma *Camada de Serviço* é implementada como um conjunto de classes maiores que implementam diretamente a lógica da aplicação, mas que delegam a lógica de domínio para as classes de objetos de domínio encapsulados. As operações disponíveis para clientes de uma *Camada de Serviço* são implementadas como roteiros, vários deles organizados em cada classe definindo uma área relacionada de lógica. Cada uma dessas classes, constitui um "serviço" da aplicação, e é comum haver nomes de tipos de serviços que começam com "Serviço". Uma *Camada de Serviço* é composta destas classes de serviços da aplicação, que devem estender um *Supertipo de Camada* (444), abstraindo suas responsabilidades e comportamentos comuns.

Ser ou Não Ser Remoto?A interface de uma classe *Camada de Serviço* tem granularidade alta quase que por definição, uma vez que ela declara um conjunto de operações de aplicação destinado à interface com as camadas clientes. As classes *Camada de Serviço* são, portanto, apropriadas para invocação remota, da perspectiva da granularidade da interface.

No entanto, a invocação remota traz embutido o custo de lidar com a distribuição de objetos. Isso provavelmente impõe muito trabalho adicional para fazer as as-

sinaturas dos métodos da sua *Camada de Serviços* lidar com *Objetos de Transferência de Dados* (380). Não subestime o custo deste trabalho, especialmente se você tiver um *Modelo de Domínio* (126) complexo e interfaces de usuário sofisticadas para complexos casos de uso de atualização! É significativo e trabalhoso – talvez ele perca apenas para o custo e trabalho necessários para o mapeamento objeto-relacional. Lembre-se da Primeira Lei do Projeto de Objetos Distribuídos (página 100-101).

Meu conselho é começar com uma *Camada de Serviço* invocada localmente, cujas assinaturas dos métodos lidem com objetos do domínio. Acrescente chamadas remotas quando precisar (se precisar) colocando *Fachadas Remotas* (368) na sua *Camada de Serviço* ou fazendo os objetos da sua *Camada de Serviço* implementarem interfaces remotas. Se a sua aplicação tiver uma interface com o usuário baseada na Web ou um *gateway* de integração baseado em serviços Web, não há lei que diga que a sua lógica de negócio tenha que rodar em um processo separado das páginas no servidor e serviços Web. De fato, você pode economizar algum esforço de desenvolvimento e tempo de resposta durante a execução, sem sacrificar escalabilidade, começando com uma abordagem que as coloque lado a lado.

Identificando Serviços e Operações Identificar as operações necessárias na fronteira de uma Camada de Serviço é bastante claro. Elas são determinadas pelas necessidades dos clientes da *Camada de Serviço*, a mais significativa (e primeira) das quais é geralmente uma interface com o usuário. Uma vez que uma interface com o usuário é projetada para suportar os casos de uso que os atores querem realizar com uma aplicação, o ponto de partida para identificar as operações da *Camada de Serviço* é o modelo de caso de uso e o projeto da interface com o usuário da aplicação.

Por mais desapontador que seja, muitos dos casos de uso em uma aplicação corporativa são simples e maçantes casos de uso "CRUD" (iniciais em inglês para criar, ler, atualizar e excluir) sobre objetos do domínio – criar um destes, ler um conjunto daqueles, atualizar este outro. Minha experiência é de que quase sempre existe uma correspondência um-para-um entre casos de uso CRUD e operações da *Camada de Serviço*.

As responsabilidades da aplicação em levar a cabo esses casos de uso, todavia, podem ser tudo, menos maçantes. Validações à parte, criação, atualização ou exclusão de um objeto do domínio em uma aplicação, cada vez mais requerem notificações a outras pessoas e aplicações integradas. Essas respostas devem ser coordenadas em transações atômicas por operações da *Camada de Serviço*.

Seria bom se fosse assim tão fácil identificar as abstrações da *Camada de Serviço* para agrupar operações relacionadas. Não há uma lei obrigatória nesta área, apenas heurísticas vagas. Para uma aplicação suficientemente pequena, pode ser o bastante ter apenas uma abstração, com o mesmo nome da aplicação. Segundo minha experiência, aplicações maiores são particionadas em diversos "subsistemas", cada um dos quais inclui uma fatia vertical completa da pilha de camadas da arquitetura. Neste caso, prefiro uma abstração por subsistema, cujo nome provém desse subsistema. Outras possibilidades incluem abstrações refletindo partições importantes em um modelo de domínio, caso estas sejam diferentes das partições do subsistema (p. ex., `ServiçoDeContratos`, `ServiçoDeProdutos`) e abstrações cujo nome deriva de comportamentos temáticos da aplicação (p. ex., `ServiçoDeLançamento`).

Implementação Java

Tanto na abordagem fachada de domínio quanto na abordagem roteiro de operação, uma classe da *Camada de Serviço* pode ser implementada como um POJO

ou um *session bean* sem estado. O compromisso é entre a facilidade de teste e a facilidade de controle de transação. Os POJOs poderiam ser mais fáceis de testar, já que não têm que ser carregados em um contâiner EJB para rodar, mas é mais difícil para uma *Camada de Serviço* baseada em POJOs se adaptar a um ambiente de serviços de transações distribuídas gerenciados por contâineres, especialmente em chamadas inter-serviços. Os EJBs, por outro lado, já vêm com o potencial para transações distribuídas gerenciadas por contâineres, mas têm que ser carregados em um contâiner antes que possam ser executados e testados. Escolha seu veneno.

Minha maneira preferida de aplicar uma *Camada de Serviço* em J2EE é com *session beans* EJB 2.0 sem estado, usando interfaces locais, e a abordagem do roteiro de operação, delegando a objetos POJO de classes do domínio. É muito conveniente implementar uma *Camada de Serviço* usando *session beans* sem estado, por causa das transações distribuídas gerenciadas pelo contâiner fornecidas pelo EJB. Além disso, com as interfaces locais introduzidas no EJB 2.0, uma *Camada de Serviço* pode explorar os valiosos serviços de transação ao mesmo tempo em que evita as espinhosas questões envolvendo distribuição de objetos.

Em uma observação específica para Java, deixe-me diferenciar a *Camada de Serviço* do padrão *Fachada de Sessão* (*Session Facade*) documentado na literatura de padrões J2EE [Alur *et al.*] e [Marinescu]. A *Fachada de Sessão* foi motivada pelo desejo de evitar a perda de desempenho causado pelo excesso de chamadas remotas em *entity beans*. Nesse sentido, ela recomenda o uso de *session beans* atuando como fachada para *entity beans*. Em vez disso, a motivação da *Camada de Serviço* é fatorar responsabilidade para evitar duplicação e promover a reutilização. É um padrão arquitetural que transcende a tecnologia. De fato, o padrão fronteira da aplicação [Cockburn PloP] que inspirou a *Camada de Serviços* precede o EJB em três anos. A *Fachada de Serviço* pode estar no espírito da *Camada de Serviço*, mas, como correntemente chamada, definida em seu escopo e apresentada, os padrões não são os mesmos.

Quando Usá-la

O benefício da *Camada de Serviço* é que ela define um conjunto comum de operações da aplicação disponível para muitos tipos de clientes e coordena uma resposta da aplicação em cada operação. A resposta pode envolver lógica de aplicação que precise ser executada atomicamente por diversos recursos transacionais. Assim, em uma aplicação com mais de um tipo de cliente da sua lógica de negócio e respostas complexas aos casos de uso, envolvendo múltiplos recursos transacionais, faz bastante sentido incluir uma *Camada de Serviço* com transações gerenciadas por contâiners, mesmo em uma arquitetura não-distribuída.

A questão mais fácil de responder é provavelmente quando não usá-la. Você provavelmente não precisa de uma Camada de Serviço se a lógica de negócio da sua aplicação só tiver um tipo de cliente – digamos, uma interface com o usuário – e as respostas aos seus casos de uso não envolverem múltiplos recursos transacionais. Neste caso, seus Controladores de Página podem controlar manualmente as transações e coordenar qualquer resposta que seja requerida, talvez delegando diretamente à camada da Fonte de Dados (*Data Source*).

No entanto, se você prever um segundo tipo de cliente ou um segundo recurso transacional em respostas de casos de usos, vale a pena projetar uma *Camada de Serviço* desde o início.

Leitura Adicional

Não há muita coisa anterior sobre a *Camada de Serviço*, cuja inspiração é o padrão fronteira da aplicação de Alistair Cockburn [Cockburn PloP]. Na área de serviços remotos [Alpert *et al.*] discutem o papel das fachadas em sistemas distribuídos. Compare este trabalho com as várias apresentações de *Fachadas de Seção* [Alur *et al.*] e [Marinescu]. No tópico sobre responsabilidades da aplicação que devem ser coordenadas dentro de operações da *Camada de Serviço*, é muito útil a descrição de Cockburn de casos de uso como um contrato de comportamento [Cockburn UC]. Uma referência anterior sobre fundamentos é a metodologia *Fusion* para a identificação de "operações do sistema" [Coleman *et al.*].

Exemplo: Lançamento de Receitas (Java)

Este exemplo continua o exemplo de lançamento de receitas dos padrões *Roteiro de Transação* (120) e *Modelo de Domínio* (126), demonstrando como a *Camada de Serviços* é usada para fazer um roteiro da lógica de aplicação e delegar a lógica do domínio em uma operação da *Camada de Serviço*. Ele usa a abordagem do roteiro da operação para implementar uma *Camada de Serviço*, primeiro com POJOs e depois com EJBs.

Para realizar a demonstração, expandimos o cenário para incluir um pouco de lógica de aplicação. Suponha que os casos de uso da aplicação requeiram que, quando o lançamento de receitas de um contrato for calculado, a aplicação deva responder enviando um *e-mail*, notificando este evento a um administrador de contrato designado e publicando uma mensagem, usando um *middleware* orientado a mensagens, para notificar outras aplicações integradas.

Começamos alterando a classe `ServiçoDeLançamento` do exemplo do *Roteiro de Transação* (120) para estender uma *Camada Supertipo* (444) e para usar alguns *Gateways* (436) na execução da lógica de aplicação. Isso produz o diagrama de classes da Figura 9.7. ServiçoDeLançamento se torna uma implementação POJO de um serviço de aplicação *Camada de Serviço*, e seus métodos representam duas das operações disponíveis na fronteira da aplicação.

Os métodos da classe ServiçoDeLançamento constroem o roteiro da lógica de aplicação das operações, delegando a lógica do domínio para os objetos pertencentes às classes do domínio (do exemplo do *Modelo de Domínio* (126)).

```
public class ServiçoDeAplicação {
    protected GatewayDeEmail obterGatewayDeEmail() {
        //retorna uma instancia de GatewayDeEmail
    }
    protected GatewayDeIntegracao obterGatewayDeIntegracao() {
        //retorna uma instancia de GatewayDeIntegracao
    }
}
public interface GatewayDeEmail {
    void enviarEmail(String endereco, String assunto, String corpoDaMensagem);
}
public interface GatewayDeIntegracao {
    void publicarCálculoDeLançamentoDeReceita(Contrato contrato);
}
public class ServiçoDeLançamento
extends ServiçoDeAplicação {
```

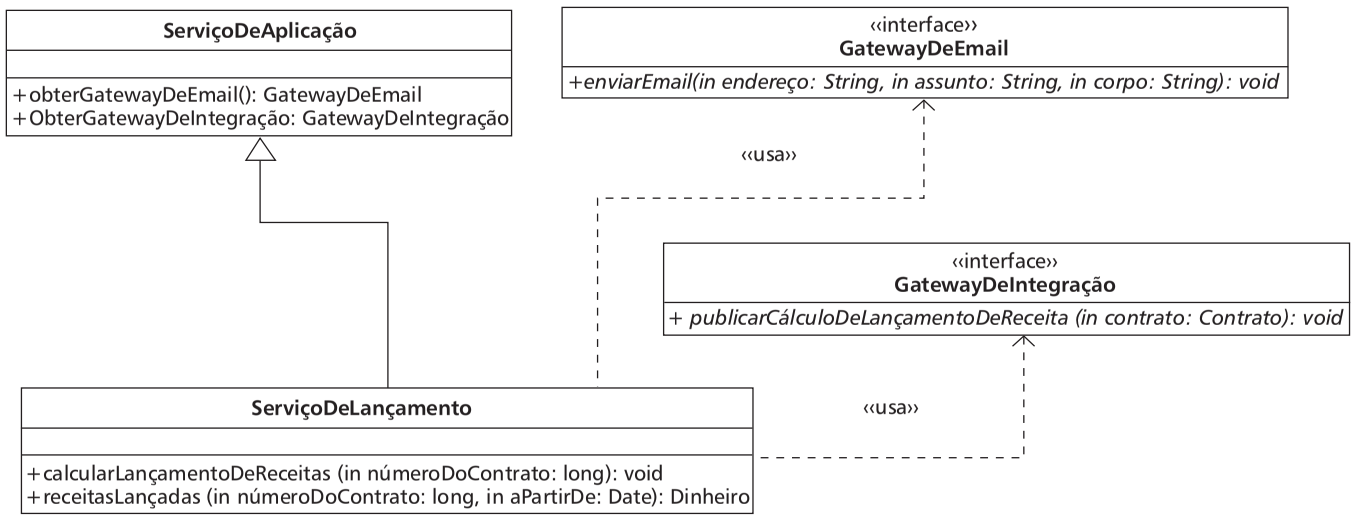


Figura 9.7 Diagrama de Classes POJO ServiçoDeLançamento.

```
    public void calcularLançamentoDeReceitas(long númeroDoContrato) {
        Contrato contrato = Contrato.lerParaAtualização(númeroDoContrato);
        contrato.calcularLançamentos();
        obterGatewayDeEmail().enviarEmail(
            contrato.lerEmailDoAdministrador(),
            "RE: Contrato #" + númeroDoContrato,
            contrato + " teve o lançamento de receitas calculado.");
        obterGatewayDeIntegracao().publicarCálculoDeLançamentoDeReceita(contrato);
    }
    public Dinheiro receitasLançadas(long númeroDoContrato, Date aPartirDe)  {
        return Contrato.ler(númeroDoContrato).receitaLançada(aPartirDe);
    }
}
```

Os detalhes de persistência são novamente deixados fora do exemplo. É suficiente dizer que a classe Contrato implementa métodos estáticos para ler contratos da camada de dados, identificados pelos seus números. Um desses métodos tem um nome que revela uma intenção de atualizar o contrato lido, o que permite a um *Mapeador de Dados* (170) relacionado registrar o(s) objeto(s) lido(s) com uma *Unidade de Trabalho* (187), por exemplo.

Detalhes de controle de transação são também deixados de fora do exemplo. O método `calcularLançamentoDeReceitas()` é inerentemente transacional porque, durante sua execução, objetos persistentes referentes a contratos são modificados por meio da adição de lançamentos de receitas, mensagens são enfileiradas em um *middleware* orientado a mensagens e *e-mails* são enviados. Todas essas respostas devem ser executadas atomicamente, no contexto de uma transação, porque não queremos enviar *e-mails* e publicar mensagens para outras aplicações se as alterações do contrato não puderem ser gravadas.

Na plataforma J2EE podemos deixar o contêiner EJB gerenciar as transações distribuídas implementando serviços de aplicação (e *Gateways* (436)) como *session beans* sem estado que usam contextos de transação. A Figura 9.8 mostra o diagrama de classes de uma implementação do `ServiçoDeLançamento` que usa interfaces locais EJB 2.0 e o idioma "interface de negócio". Nesta implementação uma *Camada Supertipo* (444) é ainda usada, fornecendo implementações padrão dos métodos de *beans* requeridos pelo EJB, além dos métodos específicos da aplicação. Se partirmos do princípio de que as interfaces `GatewayDeEmail` e `GatewayDeIntegração` também são "interfaces de negócio" para seus respectivos *session beans* sem estado, então o controle da transação distribuída é obtido declarando-se os métodos `calcularLançamentoDeReceitas`, `enviarEmail` e `publicarCálculoDeLançamentoDeReceita` como sendo transacionais. Os métodos do ServiçoDeLançamento do exemplo POJO são movidos sem alterações para a classe `ImplementaçãoDoBeanServiçoDeLançamento`.

O ponto importante sobre o exemplo é que a *Camada de Serviço* usa conjuntamente um roteiro de operação e objetos de classes do domínio na coordenação da resposta transacional da operação. O método calcularLançamentosDeReceitas contrói o roteiro da lógica de aplicação da resposta requerida pelos casos de uso da aplicação, mas delega a lógica do domínio para os objetos das classes do domínio. Ele também apresenta algumas técnicas para combater a lógica duplicada dentro dos roteiros de operação de uma *Camada de Serviço*. As responsabilidades são fatoradas em diferentes objetos, (p. ex., *Gateways* (436)) que podem ser reutilizados por meio de delegação. Uma *Camada Supertipo* (444) fornece acesso adequado a esses outros objetos.

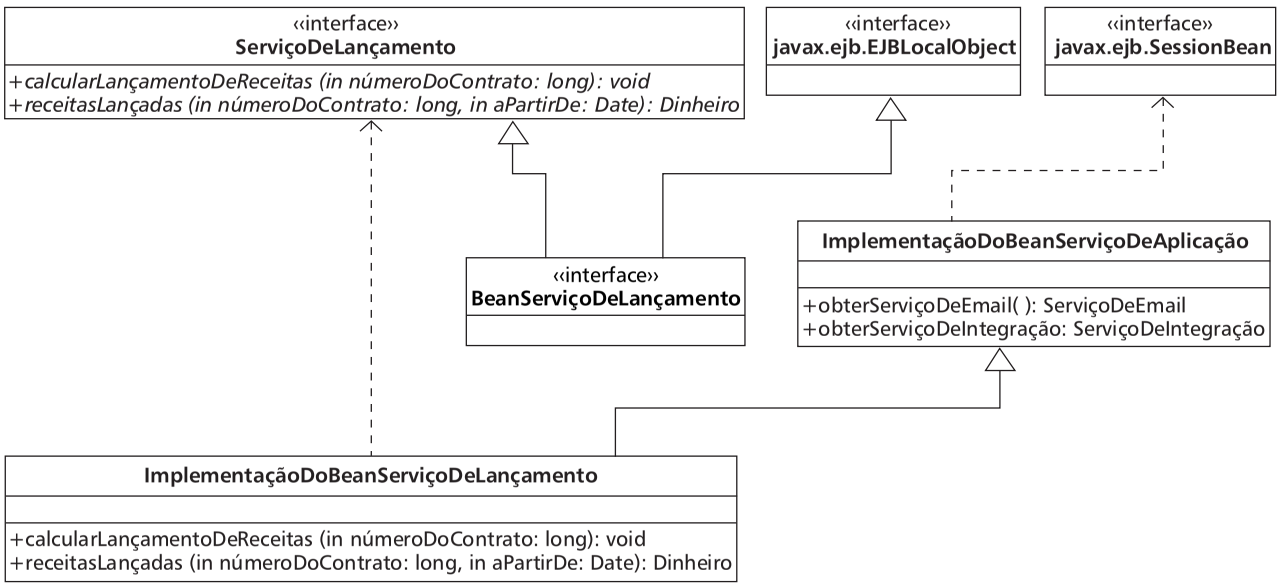


Figura 9.8 Diagrama de classe EJB para o ServiçoDeLançamento.

Alguns podem argumentar que uma implementação mais elegante do roteiro de operação usaria o padrão Observer [Gang of Four], mas o Observer é difícil de implementar em uma *Camada de Serviço* sem estado e com múltiplas *threads*. Na minha opinião, o código aberto do roteiro de operação é mais claro e mais simples.

Alguns poderiam ainda argumentar que as responsabilidades da lógica de aplicação poderiam ser implementadas em métodos de objetos do domínio, tais como `Contrato.calcularLançamentoDeReceitas()`, ou até mesmo na camada de dados, eliminando com isso a necessidade de uma Camada de Serviço separada. Acredito no entanto, essas atribuições de responsabilidades indesejáveis por uma série de motivos. Primeiro, as classes de domínio são menos reutilizáveis em outras aplicações se implementarem lógica específica da aplicação (e dependerem de *Gateways* (436) específicos da aplicação, e assim por diante). Elas deveriam modelar as partes do domínio do problema que são de interesse da aplicação, o que não significa todas as responsabilidades dos casos de uso da aplicação. Segundo, encapsular lógica de aplicação em uma camada "mais alta" dedicada a esse propósito (o que a camada de dados não é) facilita a alteração da implementação dessa camada – para usar talvez um mecanismo de fluxo de trabalho (*workflow*).

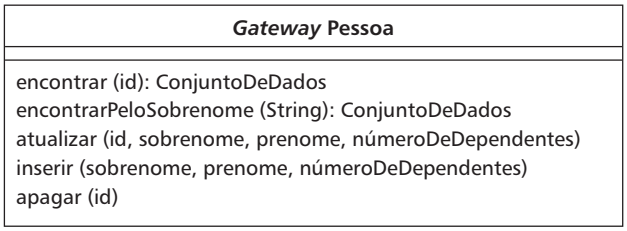
Como um padrão para a organização da camada de lógica de uma aplicação corporativa, a *Camada de Serviço* combina roteiros e classes de objetos do domínio, alavancando os melhores aspectos de ambos. Diversas variações são possíveis na implementação de uma *Camada de Serviço* – por exemplo, fachadas de domínio ou roteiros de operação, POJOs ou *session beans*, ou uma combinação de ambos. A *Camada de Serviço* pode ser projetada para chamadas locais, chamadas remotas ou ambas. Mais importante de tudo, independentemente dessas variações, este padrão estabelece a fundação para a implementação encapsulada da lógica de negócio de uma aplicação e para a chamada consistente dessa lógica por seus diversos clientes.

CAPÍTULO 10

Padrões Arquiteturais de Fontes de Dados (*Data Source*)

Gateway de Tabela de Dados (Table Data Gateway)

Um objeto que atua como um Gateway (436) para uma tabela do banco de dados. Uma instância lida com todas as linhas na tabela.



Misturar SQL com a lógica da aplicação pode causar vários problemas. Muitos desenvolvedores não se sentem confortáveis com SQL, e muitos dos que se sentem, podem não escrevê-lo bem. Os administradores de bancos de dados precisam poder encontrar facilmente as instruções SQL de modo que eles possam descobrir como ajustar e aprimorar o banco de dados.

Um *Gateway de Tabela de Dados* armazena todo o SQL utilizado para acessar uma única tabela ou visão: seleções, inserções, atualizações e exclusões. Os outros códigos chamam os métodos do *Gateway* para toda a interação necessária com o banco de dados.

Como Funciona

Um *Gateway de Tabela de Dados* tem uma interface simples, consistindo normalmente de diversos métodos de busca para obter dados do banco de dados e métodos para atualizar, inserir e excluir dados. Cada método mapeia os parâmetros de entrada em uma chamada SQL e executa o SQL sobre a conexão com o banco de dados. O *Gateway de Tabela de Dados* normalmente não retém estado, já que seu papel é simplesmente enviar dados do banco e trazer dados para o banco.

O mais complicado em um *Gateway de Tabela de Dados* é como ele deve retornar informações de uma consulta. Mesmo uma consulta simples, do tipo recuperar um registro dada a chave primária, retornará diversos itens de dados. Em ambientes onde você pode retornar múltiplos itens, você pode usar esta característica para retornar uma única linha da tabela, mas muitas linguagens permitem apenas um único valor de retorno, e muitas buscas retornam múltiplas linhas.

Uma alternativa é retornar alguma estrutura de dados simples, tal como um mapa. Um mapa funciona, mas força a cópia dos dados do conjunto de registros que retorna da consulta ao banco de dados para o mapa. Penso que o uso de mapas para passar dados é uma prática ruim, porque ele frustra a verificação em tempo de compilação e não é uma interface muito explícita, levando a falhas quando as pessoas não entendem o que está no mapa. Uma alternativa melhor é usar um *Objeto de Transferência de Dados* (380). É mais um objeto que tem de ser criado, mas é um objeto que pode muito bem ser usado em todo lugar.

Para evitar tudo isso, você pode retornar o *Conjunto de Registros* (473) resultante da consulta SQL. Conceitualmente, isso é confuso, uma vez que, idealmente, o objeto na memória não tem que saber nada a respeito da interface SQL. Isso também pode tornar difícil substituir o banco de dados por um arquivo, se você não puder

criar conjuntos de registros com facilidade no seu próprio código. Apesar disso, em muitos ambientes que fazem uso extensivo de *Conjuntos de Registros* (473), tal como o .NET, esta é uma abordagem muito efetiva. Desta forma, um *Gateway de Tabela de Dados,* combina muito bem com o *Módulo Tabela* (134). Se todas as suas atualizações forem feitas por meio do *Gateway de Tabela de Dados*, os dados retornados podem ser baseados em visões, em vez de nas tabelas reais, o que reduz o acoplamento entre o seu código e o banco de dados.

Se você estiver usando um *Modelo de Domínio* (126), pode fazer o *Gateway de Tabela de Dados* retornar o objeto de domínio apropriado. O problema disso é que você passa a ter dependências bidirecionais entre os objetos do domínio e o *gateway*. Os dois são intimamente conectados, de modo que isso não é necessariamente algo terrível, mas é algo que sempre reluto em fazer.

Na maioria das vezes em que você usar um *Gateway de Tabela de Dados*, terá um para cada tabela no banco de dados. Para casos muito simples, entretanto, você pode ter um único *Gateway de Tabela de Dados* que lide com todos os métodos para todas as tabelas. Você também pode ter um *gateway* para visões ou até mesmo para pesquisas interessantes que não são mantidas no banco de dados como visões. Obviamente, muitas vezes, os *Gateway de Tabela de Dados* baseados em visões não são atualizáveis e, assim, não terão o comportamento de atualização. Contudo, se você puder fazer atualizações nas tabelas associadas, então encapsular essas atualizações por trás de operações de atualização no *Gateway de Tabela de Dados* é uma excelente técnica.

Quando Usá-lo

Assim como com o *Gateway de Linha de Dados* (158), a decisão relativa ao *Gateway de Tabela de Dados* é, em primeiro lugar, usar ou não um *Gateway* (436) e, depois, qual deles.

Acho que o *Gateway de Tabela de Dados* é provavelmente o padrão de interface de banco de dados mais simples de usar, na medida em que ele mapeia tão bem tanto para uma tabela de em banco de dados quanto para um tipo registro. Ele também consiste em um lugar natural para encapsular a exata lógica de acesso da fonte de dados. Eu o uso menos com o *Modelo de Domínio* (126) porque acredito que o *Mapeador de Dados* (170) provê um isolamento melhor entre o *Modelo de Domínio* (126) e o banco de dados.

O *Gateway de Tabela de Dados* funciona especialmente bem com o *Módulo Tabela* (134), no qual ele gera um conjunto de registros para ser usado pelo *Módulo Tabela* (134). Em verdade, não consigo imaginar qualquer outra abordagem de mapeamento de banco de dados para o *Módulo Tabela* (134).

Assim como o *Gateway de Linha de Dados,* o *Gateway de Tabela de Dados* é muito apropriado para *Roteiros de Transação* (120). A escolha entre os dois se reduz a como eles lidam com múltiplas linhas de dados. Muitas pessoas gostam de usar um *Objeto de Transferência de Dados* (380), mas isso me parece não compensar o trabalho, a menos que o mesmo *Objeto de Transferência de Dados* (380) seja usado em algum outro lugar. Prefiro o *Gateway de Tabela de Dados* quando a representação do conjunto resultante for conveniente para ser manipulado pelo *Roteiro de Transação* (120).

É interessante observar como muitas vezes faz sentido fazer os *Mapeadores de Dados* (170) se comunicarem com o banco de dados por meio de *Gateways de Tabela de Dados*. Embora isso não seja útil quando tudo é codificado à mão, pode ser muito efetivo se você quiser usar metadados para os *Gateways de Tabela de Dados*, mas preferir codificar à mão o mapeamento real para os objetos do domínio.

Um dos benefícios na utilização de um *Gateway de Tabela de Dados* é que a mesma interface pode funcionar tanto na utilização de SQL para manipular o ban-

co de dados quanto na utilização de procedimentos armazenados. De fato, os próprios procedimentos armazenados são muitas vezes organizados como *Gateways de Tabela de Dados*. Dessa forma, os procedimentos armazenados para inserção e atualização encapsulam a estrutura real da tabela. Neste caso, os procedimentos de pesquisa podem retornar visões, o que ajuda a esconder a estrutura das tabelas correspondentes.

Leitura Adicional

[Alur *et al.*] discutem o padrão *Objeto de Acesso a Dados*, que é um *Gateway de Tabela de Dados*. Eles mostram como retornar uma coleção de *Objetos de Transferência de Dados* (401) nos métodos de consulta. Não fica claro se eles vêem este padrão como sendo sempre baseado em tabelas. A intenção e a discussão parecem sugerir ou um *Gateway de Tabela de Dados* ou um *Gateway de Linha de Dados* (158).

Usei um nome diferente, em parte porque vejo este padrão como um uso particular do conceito mais genérico de *Gateways* (436) e quero que o nome do padrão reflita isso. Além do mais, o termo *Objeto de Acesso a Dados* (*Data Access Object*) e sua abreviação DAO tem seu próprio significado específico dentro do mundo Microsoft.

Exemplo: O Gateway Pessoa (C#)

Um *Gateway de Tabela de Dados* é a forma usual de acesso a banco de dados no mundo Windows, então faz sentido ilustrar seu uso com C#. Tenho que enfatizar, no entanto, que esta forma clássica de *Gateway de Tabela de Dados* não se ajusta perfeitamente ao ambiente .NET, já que ela não tira proveito do conjunto de dados (*data set*) ADO.NET. Em vez disso, ela usa o leitor de dados (*data reader*), que é uma interface do tipo cursor para registros de bancos de dados. O leitor de dados é a escolha certa para manipular quantidades maiores de informação quando você não quiser trazer tudo para a memória de uma só vez.

Para o exemplo, estou usando uma classe Gateway Pessoa que conecta a uma tabela pessoa em um banco de dados. O Gateway Pessoa contém o código de busca, retornando um leitor de dados ADO.NET para acessar os dados retornados.

```
class GatFewayPessoa...

    public IDataReader EncontrarTodos( ) {
        String sql = "SELECT * FROM pessoa";
        return new OleDbCommand(sql, DB.Connection).ExecuteReader( );
    }
    public IDataReader EncontrarPeloSobrenome (String sobrenome) {
        String sql = "SELECT * FROM pessoa WHERE sobrenome = ?";
        IDbCommand comm = new OleDbCommand(sql, DB.Connection);
        comm.Parameters.Add(new OleDbParameter("sobrenome", sobrenome));
        return comm.ExecuteReader( );
    }
    public IDataReader EncontrarComCondição (String cláusulaCondicional) {
        String sql = String.Format("SELECT * FROM pessoa WHERE {0}", cláusulaCondicional);
        return new OleDbCommand(sql, DB.Connection).ExecuteReader( );
    }
```

Na maioria das vezes, você desejará enviar de volta um conjunto de linhas com um leitor. Em raras ocasiões, você pode desejar pegar uma linha de dados individual:

```
class GatewayPessoa...

    public Object[ ] EncontrarLinha (long chave) {
        String sql = "SELECT * FROM pessoa WHERE id = ?";
        IDbCommand comm = new OleDbCommand(sql, DB.Connection);
        comm.Parameters.Add (new OleDbParameter("chave", chave));
        IDataReader leitor = comm.ExecuteReader( );
        leitor.Read( );
        Object[ ] resultado = new Object[leitor.FieldCount];
        leitor.GetValues(resultado);
        leitor.Close( );
        return resultado;
    }
```

Os métodos de atualização e inserção recebem os dados necessários em parâmetros e chamam as rotinas SQL apropriadas.

```
class GatewayPessoa...

    public void atualizar (long chave, String sobrenome, String prenome, long númeroDeDependentes) {
        String sql = @"
            UPDATE pessoa
                SET sobrenome = ?, prenome = ?, númeroDeDependentes = ?
                WHERE id = ?";
        IDbCommand comm = new OleDbCommand(sql, DB.Connection);
        comm.Parameters.Add (new OleDbParameter ("sobrenome", sobrenome));
        comm.Parameters.Add (new OleDbParameter ("prenome", prenome));
        comm.Parameters.Add (new OleDbParameter ("numDep", númeroDeDependentes));
        comm.Parameters.Add (new OleDbParameter ("chave", chave));
        comm.ExecuteNonQuery( );
    }
```

```
class GatewayPessoa...

    public long inserir (String sobrenome, String prenome, long númeroDeDependentes) {
        String sql = "INSERT INTO pessoa VALUES (?, ?, ?, ?);
        long chave = GetNextID( );
        IDbCommand comm = new OleDbCommand (sql, DB.Connection);
        comm.Parameters.Add(new OleDbParameter("chave", chave);
        comm.Parameters.Add (new OleDbParameter ("sobrenome", sobrenome));
        comm.Parameters.Add (new OleDbParameter ("prenome", prenome));
        comm.Parameters.Add (new OleDbParameter ("numDep", númeroDeDependentes));
        comm.ExecuteNonQuery( );
        return key;
    }
```

O método de exclusão precisa apenas de uma chave.

```
class GatewayPessoa...

    public void excluir (long chave) {
        String sql = "DELETE FROM pessoa WHERE id = ?";
        IDbCommand comm = new OleDbCommand (sql, DB.Connection);
        comm.Parameters.Add(new OleDbParameter("chave", chave);
        comm.ExecuteNonQuery( );
    }
```

Exemplo: Usando Conjuntos de Dados ADO.NET (C#)

O *Gateway de Tabela de Dados* genérico trabalha com quase todo tipo de plataforma, já que ele não é nada além de um envoltório para declarações SQL. Com o .NET você usa mais freqüentemente conjuntos de dados (*data sets*), mas o *Gateway de Tabela de Dados* ainda é útil, embora ele assuma uma forma diferente.

Um conjunto de dados precisa de adaptadores de dados para nele carregar os dados e para atualizar esses dados. Considero útil definir um repositório para o conjunto de dados e os adaptadores. Um *gateway* usa então o repositório para armazená-los. Muito deste comportamento é genérico e pode ser feito em uma superclasse.

O repositório indexa os conjuntos de dados e os adaptadores pelo nome da tabela.

```
class RepositórioDeConjuntosDeDados...

    public DataSet dados = new DataSet( );
    private Hashtable adaptadoresDeDados = new Hashtable( );
```

O *gateway* armazena o repositório e expõe o conjunto de dados para os seus clientes.

```
class GatewayDeDados...

    public RepositórioDeConjuntosDeDados repositório;
    public DataSet dados {
        get { return repositório.dados;}
    }
```

O *gateway* pode atuar sobre um repositório existente ou pode criar um novo repositório.

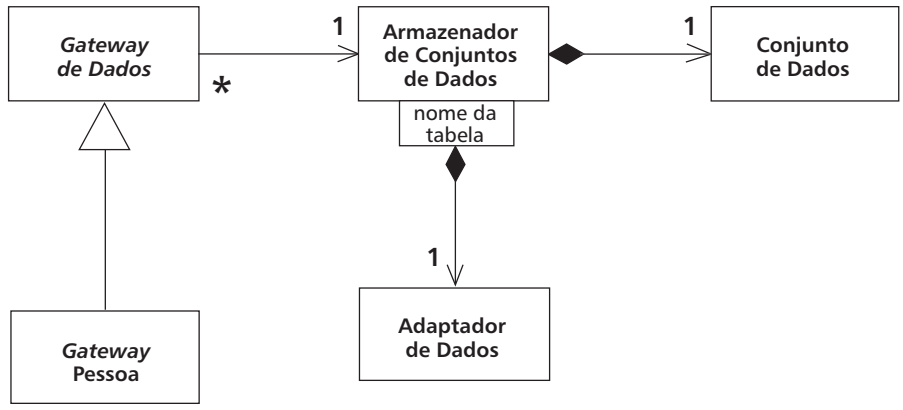


Figura 10.1 Diagrama de classes de um *gateway* orientado a conjunto de dados e o repositório de dados de apoio.

```
class GatewayDeDados...

    protected GatewayDeDados ( ) {
        repositório = new RepositórioDeConjuntosDeDados ( );
    }
    protected GatewayDeDados (RepositórioDeConjuntosDeDados repositório) {
        this.repositório = repositório;
    }
```

O código de busca pode ter um comportamento um pouco diferente aqui. Um conjunto de dados é um contâiner de dados orientados a tabelas e pode armazenar dados provenientes de diversas tabelas. Por esse motivo, é melhor carregar os dados em um conjunto de dados.

```
class GatewayDeDados...

    public void CarregarTudo( ) {
        String comando = String.Format( "select * from {0}", NomeDaTabela);
        repositório.PreencherDados (comando, NomeDaTabela);
    }
    public void CarregarCondicional (String cláusulaCondicional) {
        String comando =
            String.Format("select * from {0} where {1}", NomeDaTabela, cláusulaCondicional);
        repositório.PreencherDados(comando, NomeDaTabela);
    }
    abstract public String NomeDaTabela {get;}

class GatewayPessoa...

    public override String NomeDaTabela {
        get {return "Pessoa";}
    }

class RepositórioDeConjuntosDeDados...

    public void PreencherDados (String consulta, String nomeDaTabela) {
        if (adaptadoresDeDados.Contains(nomeDaTabela)) throw new MultipleLoadException( );
        OleDbDataAdapter ad = new OleDbDataAdapter(consulta, DB.connection);
        OleDbCommandBuilder construtor= new OleDbCommandBuilder(ad);
        ad.Fill(dados, nomeDaTabela);
        adaptadoresDeDados.Add(nomeDaTabela, ad);
    }
```

Para atualizar os dados, você manipula o conjunto de dados diretamente no código cliente.

```
        pessoa.CarregarTudo( );
        pessoa[chave]["sobrenome"] = "Odell";
        pessoa.repositório.Atualizar( );
```

O *gateway* pode ter um indexador para tornar mais fácil obter linhas específicas.

```
class GatewayDeDados...

    public DataRow this [long chave] {
        get{
            String filtro = String.Format("id = {0}", chave);
```

```
            return Table.Select(filtro)[0];
        }
    }
    public override DataTable Table {
        get{ return Dados.Tables[NomeDaTabela];}
    }
```


A atualização dispara o código de atualização no repositório.

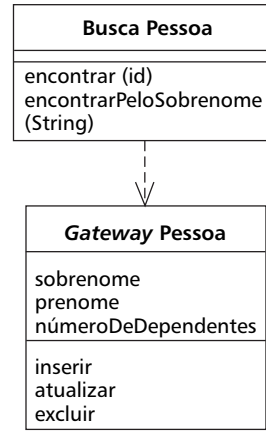
```
class RepositórioDeConjuntosDeDados...

    public void Atualizar( ) {
        foreach (String tabela in adaptadoresDeDados.Keys)
            ((OleDbDataAdapter)adaptadoresDeDados[tabela]).Atualizar(Dados, tabela);
    }
    public DataTable this [String nomeDaTabela] {
        get {return Dados.Tables[nomeDaTabela];}
    }
```

A inserção pode ser feita essencialmente da mesma forma: obtenha um conjunto de dados, insira uma nova linha na tabela de dados e preencha cada coluna. No entanto, um método de atualização pode fazer a inserção em uma única chamada.

```
class GatewayDeDados...

    public long Inserir (String sobrenome, String prenome, int númeroDeDependentes) {
        long chave = new PersonGatewayDS().GetNextID( );
        DataRow novaLinha = Table.NewRow( );
        novaLinha["id"] = chave;
        novaLinha["sobrenome"] = sobrenome;
        novaLinha["prenome"] = prenome;
        novaLinha["númeroDeDependentes"] = númeroDeDependentes;
        Table.Rows.Add (novaLinha);
        return chave;
    }
```

Gateway de Linha de Dados (Row Data Gateway)

Um objeto que age como um Gateway (436) para um único registro em uma fonte de dados. Haverá uma instância por linha.



Embutir código de acesso ao banco de dados em objetos na memória pode lhe trazer algumas desvantagens. Para começar, se os seus objetos na memória tiverem sua própria lógica de negócio, acrescentar o código de manipulação do banco de dados aumenta a complexidade. Os testes também ficam complicados uma vez que, se os seus objetos na memória estiverem ligados a um banco de dados, a execução dos testes fica mais lenta devido à sobrecarga do acesso ao banco de dados. Você pode ter de acessar múltiplos bancos de dados com todas aquelas pequenas e irritantes variações de SQL.

Um *Gateway de Linha de Dados* fornece objetos que parecem exatamente com o registro na sua estrutura de registros, mas que podem ser acessados com os mecanismos normais da sua linguagem de programação. Todos os detalhes do acesso à fonte de dados ficam escondidos atrás desta interface.

Como Funciona

Um *Gateway de Linha de Dados* age como um objeto que encapsula exatamente um único registro, tal como uma linha de um banco de dados. Nesta estrutura, cada coluna no banco de dados torna-se um campo. Normalmente, o *Gateway de Linha de Dados* faz qualquer conversão de tipo necessária dos tipos da fonte de dados para os tipos na memória, porém esta conversão é muito simples. Este padrão armazena os dados de uma linha de modo que um cliente possa então acessar diretamente o *Gateway de Linha de Dados*. O *gateway* age como uma boa interface para cada linha de dados. Essa abordagem funciona especialmente bem em *Roteiros de Transações* (120).

Com o *Gateway de Linha de Dados*, você se depara com as questões relativas a onde colocar as operações de busca que geram este padrão. Você pode usar métodos estáticos de busca, mas eles impedem o polimorfismo, se você quiser substituir diferentes métodos de busca para diferentes fontes de dados. Neste caso, muitas vezes, faz sentido ter objetos de busca separados de modo que cada tabela em um banco de dados relacional tenha uma classe de busca e uma classe *gateway* para os resultados (Figura 10.2).

Muitas vezes é difícil perceber a diferença entre um *Gateway de Linha de Dados* e um *Registro Ativo* (165). O ponto crucial da questão é se há alguma lógica de domínio presente. Se houver, você tem um *Registro Ativo* (165). Um *Gateway de Linha de Dados* deve conter apenas a lógica de acesso ao banco de dados e nenhuma lógica de domínio.

Do mesmo modo que com qualquer outra forma de encapsulamento tabular, você pode usar um *Gateway de Linha de Dados* com uma visão ou pesquisa assim como com uma tabela. Neste padrão, muitas vezes, as atualizações acabam por ser mais complicadas, uma vez que você tem que atualizar as tabelas correspondentes. Além disso, se você tiver dois *Gateways de Linha de Dados* que operam sobre as mesmas tabelas, você pode descobrir que o segunda *Gateway de Linha de Dados* que você atualiza desfaz as alterações do primeiro. Não há um modo genérico de evitar isso. Simplesmente, os desenvolvedores precisam estar conscientes de como os *Gateways de Linha de Dados* virtuais são formados. Afinal, a mesma coisa pode acontecer com visões atualizáveis. É claro que você pode decidir não fornecer operações de atualização.

Os *Gateways de Linha de Dados* tendem a ser um pouco tediosos para codificar, mas são bons candidatos para a geração de código baseada em um *Mapeamento em Metadados* (295). Desta forma, todo o código de acesso ao banco de dados pode ser automaticamente criado para você durante o seu processo automatizado de construção.

Quando Usá-lo

A escolha do *Gateway de Linha de Dados* muitas vezes exige dois passos: primeiro, usar ou não um *gateway* e, segundo, usar um *Gateway de Linha de Dados* ou um *Gateway de Tabela de Dados* (151).

Uso o *Gateway de Linha de Dados* com maior freqüência quando estou usando um *Roteiro de Transação* (120). Neste caso, ela decompõe eficientemente o código de acesso ao banco de dados e permite que este seja reutilizado facilmente em diferentes *Roteiros de Transação* (120).

Não uso um *Gateway de Linha de Dados* quando estou usando um *Modelo de Domínio* (126). Se o mapeamento for simples, o *Registro Ativo* (165) faz o mesmo trabalho sem uma camada adicional de código. Se o mapeamento for complexo, o *Mapeador de Dados* (170) funciona melhor, na medida em que ele é melhor para desacoplar a estrutura de dados dos objetos do domínio uma vez que estes objetos não precisam conhecer o *layout* do banco de dados. É claro que você pode usar o *Gateway de Linha de Dados* para isolar os objetos do domínio da estrutura do banco de dados. Isso é uma boa coisa se, quando você estiver usando o *Gateway de Linha de Dados*, estiver alterando a estrutura do banco de dados e não quiser alterar a lógica do domínio. Todavia, fazer isso em larga escala lhe leva a três representações de dados: uma na lógica do domínio, uma no *Gateway de Linha de Dados* e uma no banco de dados – e isso é demais. Por essa razão, normalmente, faço o *Gateway de Linha de Dados* espelhar a estrutura do banco de dados.

Tenho visto com interesse *Gateways de Linha de Dados* usados muito eficientemente com *Mapeadores de Dados* (170). Embora isso pareça demandar um trabalho adicional, pode ser eficaz se os *Gateway de Linha de Dados* forem gerados automaticamente a partir de metadados enquanto que os *Mapeadores de Dados* (170) são feitos à mão.

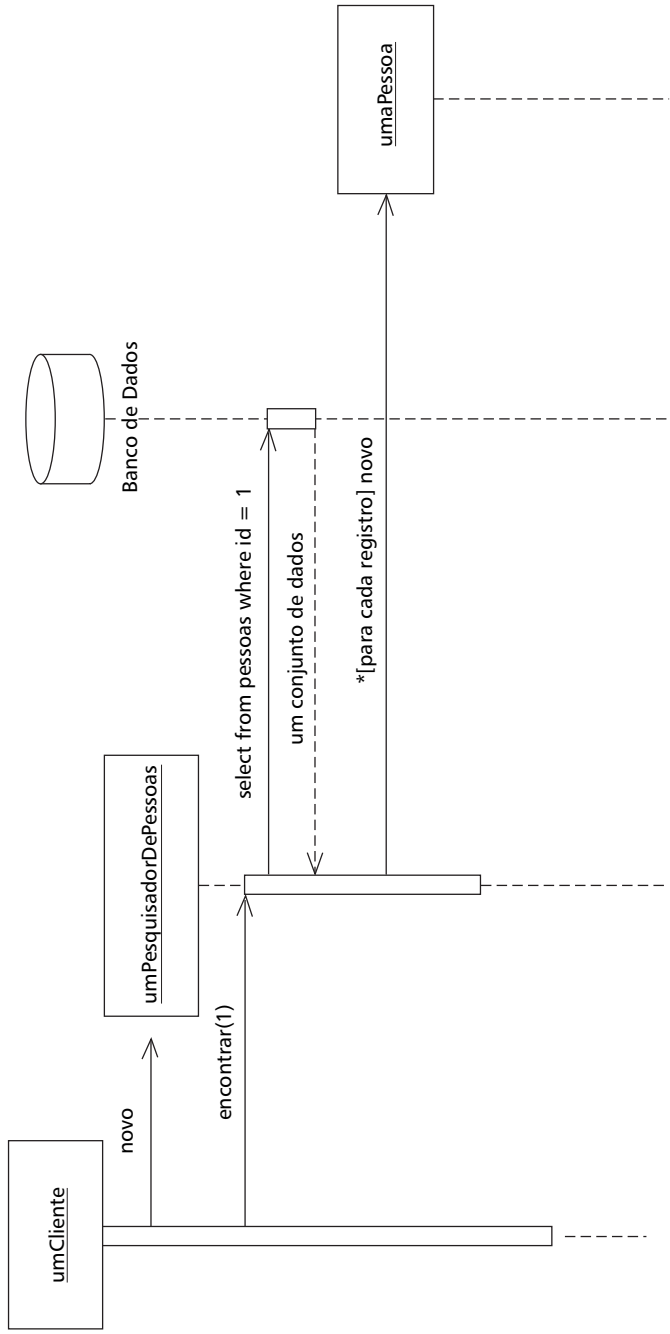


Figura 10.2 Interações para uma busca com uma *Gateway de Linhas de Dados* baseada em linhas.

Se você usar o *Roteiro de Transação* (120) com o *Gateway de Linha de Dados*, você pode perceber que existe lógica de negócio repetida em multiplos roteiros, lógica esta que faria sentido no *Gateway de Linha de Dados*. Mover esta lógica irá gradualmente transformar seu *Gateway de Linha de Dados* em um *Registro Ativo* (165), o que freqüentemente é bom, uma vez que reduz a duplicação na lógica de negócio.

Exemplo: Um Registro Pessoa (Java)

Aqui está um exemplo de um *Gateway de Linha de Dados*. É uma tabela simples, Pessoa.

```
create table pessoas (ID int primary key, sobrenome varchar,
                      prenome varchar, númeroDeDependentes int)
```

`GatewayPessoa` é um *gateway* para a tabela. Ele começa com atributos e métodos de acesso.

```
class GatewayPessoa...

    private String sobrenome;
    private String prenome;
    private int númeroDeDependentes;
    public String lerSobrenome( ) {
        return sobrenome;
    }
    public void gravarSobrenome (String sobrenome) {
        this.sobrenome = sobrenome;
    }
    public String lerPrenome ( ) {
        return prenome;
    }
    public void gravarPrenome (String prenome) {
        this.prenome = prenome;
    }
    public int lerNúmeroDeDependentes ( ) {
        return númeroDeDependentes;
    }
    public void gravarNúmeroDeDependentes (int númeroDeDependentes) {
        this.númeroDeDependentes = númeroDeDependentes;
    }
```

A classe *gateway* pode, ela própria, lidar com atualizações e inserções.

```
class GatewayPessoa...

    private static final String stringDaDeclaraçãoDeAtualização =
            "UPDATE pessoas "+
            "  set sobrenome = ?, prenome = ?, númeroDeDependentes = ? " +
            "  where id = ?";
    public void atualizar ( ) {
        PreparedStatement declaraçãoDeAtualização = null;
        try {
            declaraçãoDeAtualização = DB.prepare(stringDaDeclaraçãoDeAtualização);
            declaraçãoDeAtualização.setString(1, sobrenome);
```

```
            declaraçãoDeAtualização.setString(2, prenome);
            declaraçãoDeAtualização.setInt(3, númeroDeDependentes);
            declaraçãoDeAtualização.setInt(4, lerID( ).intValue( ));
            declaraçãoDeAtualização.execute( );
        } catch (Exception e) {
            throw new ApplicationException (e);
        } finally {DB.cleanUp(declaraçãoDeAtualização);
        }
    }
    private static final String stringDaDeclaraçãoDeInserção =
            "INSERT INTO pessoas VALUES(?, ?, ?, ?)";
    public Long inserir ( ) {
        PreparedStatement declaraçãoDeInserção = null;
        try {
            declaraçãoDeInserção = DB.prepare(stringDaDeclaraçãoDeInserção);
            setID(descobreProximoIDBancoDeDados( ));
            declaraçãoDeInserção.setInt (1, lerID.intValue( ));
            declaraçãoDeInserção.setString(2, sobrenome);
            declaraçãoDeInserção.setString(3, prenome);
            declaraçãoDeInserção.setInt(4, númeroDeDependentes);
            declaraçãoDeInserção.execute( );
            Registro.adicionarPessoa(this);
            return lerID( );
        } catch (SQLException e) {
            throw new ApplicationException (e);
        } finally {DB.cleanUp(declaraçãoDeInserção);
        }
    }
```

Para buscar pessoas no banco de dados, temos uma classe separada `PesquisadorDePessoas`. Esta trabalha junto com portão *gateway* para criar novos objetos *gateway*.

```
class PesquisadorDePessoas...

    private static final String stringDaDeclaraçãoDeBusca =
        "SELECT id, sobrenome, prenome, númeroDeDependentes " +
        "  FROM pessoas "+
        "  WHERE id = ? ";
    public GatewayPessoa procurar(Long id ) {
        GatewayPessoa resultado = (GatewayPessoa) Registro.lerPessoa(id);
        if (resultado != null) return resultado;
        PreparedStatement declaraçãoDeBusca = null;
        ResultSet rs = null;
        try {
            declaraçãoDeBusca = DB.prepare(stringDaDeclaraçãoDeBusca);
            declaraçãoDeBusca.setLong(1, id.longValue( ));
            rs = declaraçãoDeBusca.executeQuery( );
            rs.next( );
            resultado = GatewayPessoa.carregar(rs);
            return resultado;
        } catch (SQLException e) {
            throw new ApplicationException (e);
        } finally {DB.cleanUp(declaraçãoDeBusca, rs);
        }
```

```
    }
    public GatewayPessoa procurar (long id) {
        return procurar(new Long(id));
    }
```

```
class GatewayPessoa...

    public static GatewayPessoa carregar (ResultSet rs) throws SQLException {
        Long id = new Long(rs.getLong(1));
        GatewayPessoa resultado = (GatewayPessoa) Registro.lerPessoa(id);
        if (resultado != null) return resultado;
        String parâmetroSobrenome = rs.getString(2);
        String parâmetroPrenome = rs.getString(3);
        int parâmetroNúmeroDeDependentes = rs.getInt(4);
        resultado = new GatewayPessoa(id, parâmetroSobrenome, parâmetroPrenome, parâmetroNú
    meroDeDependentes);
        Registro.adicionarPessoa(resultado);
        return resultado;
    }
```

Para pesquisar mais de uma pessoa de acordo com algum critério, podemos fornecer um método de busca apropriado.

```
class PesquisadorDePessoas...

    private static final String declaraçãoDeBuscaDeResponsáveis =
            "SELECT id, sobrenome, prenome, númeroDeDependentes "+
            "  FROM pessoas "+
            "  WHERE númeroDeDependentes > 0";
    public List encontrarResponsáveis ( ) {
        List resultado = new ArrayList( );
        PreparedStatement dec = null;
        ResultSet rs = null;
        try {
            dec = DB.prepare(declaraçãoDeBuscaDeResponsáveis);
            rs = dec.executeQuery( );
            while (rs.next( )) {
                resultado.add(GatewayPessoa.carregar(rs));
            }
            return resultado;
        } catch (SQLException e) {
            throw new ApplicationException (e);
        } finally {DB.cleanUp(dec, rs);
        }
    }
```

O pesquisador de pessoas usa um *Registro* (448) para armazenar *Mapas de Identidade* (196).

Podemos agora usar os *gateways* a partir de um *Roteiro de Transação* (120).

```
        PesquisadorDePessoas pesquisador = new PesquisadorDePessoas ( )
        Iterator pessoas = pesquisador.encontrarResponsáveis( ).iterator( );
        StringBuffer resultado = new StringBuffer( );
        while (pessoas.hasNext( )) {
            GatewayPessoa cada = (GatewayPessoa) pessoas.next( );
```

```
            resultado.append (cada.lerSobrenome( ));
            resultado.append (" ");
            resultado.append (cada.lerPrenome( ));
            resultado.append (" ");
            resultado.append (String.valueOf(cada.lerNúmeroDeDependentes( ));
            resultado.append ("\n");
        }
        return resultado.toString( );
```

Exemplo: Um Armazenador de Dados para um Objeto de Domínio (Java)

Na maior parte das vezes, uso o *Gateway de Linha de Dados* com *Roteiros de Transação* (120). Se quisermos usar o *Gateway de Linha de Dados* a partir de um *Modelo de Domínio* (126), os objetos do domínio precisam chegar aos dados a partir do *gateway*. Em vez de copiarmos os dados para o objeto do domínio, podemos usar o *Gateway de Linha de Dados* como um armazenador de dados para o objeto do domínio.

```
class Pessoa...

    private GatewayPessoa dados;
    public Pessoa (GatewayPessoa dados){
        this.dados = dados;
    }
```

Métodos de acesso na lógica de domínio podem então delegar os dados para o *gateway*.

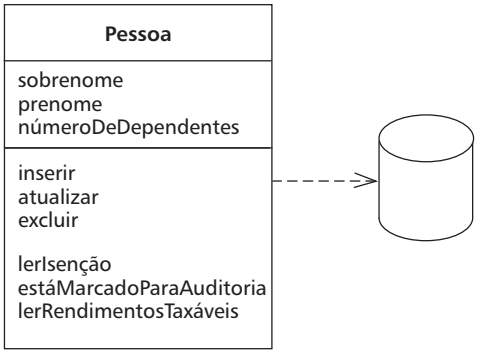
```
class Pessoa...

    public int lerNúmeroDeDependentes ( ) {
        return dados.lerNúmeroDeDependentes( );
    }
```

A lógica de domínio usa os métodos de leitura (*getters*) *para* trazer os dados do *gateway*.

```
class Pessoa...

    public Dinheiro lerIsenção( ) {
        Dinheiro baseDeIsenção = Dinheiro.dollars(1500);
        Dinheiro isençãoPorDependente = Dinheiro.dollars(750);
        return baseDeIsenção.adicionar(isençãoPorDependente.multiplicar(this.lerNúmeroDeDependentes( )));
    }
```

Registro Ativo (Active Record)

Um objeto que encapsula uma linha de uma tabela ou visão de um banco de dados, o acesso ao banco de dados e adiciona lógica de domínio a esses dados.



Um objeto carrega tanto dados quanto comportamento. Muitos desses dados são persistentes e precisam ser armazenados em um banco de dados. O *Registro Ativo* usa a abordagem mais óbvia, colocando a lógica de acesso aos dados no objeto do domínio. Dessa maneira todos os objetos pessoa sabem como ler e gravar seus dados do banco de dados e no banco de dados.

Como Funciona

A essência de um *Registro Ativo* é um *Modelo de Domínio* (126) no qual as classes são muito semelhantes à estrutura das tabelas correspondentes. Cada *Registro Ativo* é responsável por salvar e buscar seus dados no banco de dados e também por qualquer lógica de domínio que atue sobre os dados. Esta pode ser toda a lógica de domínio na aplicação, ou você pode descobrir que alguma lógica de domínio é mantida em *Roteiros de Transação* (120) com o código compartilhado e orientado a dados no *Registro Ativo*.

A estrutura de dados do *Registro Ativo* deve corresponder exatamente àquela do banco de dados: um atributo na classe para cada coluna na tabela. O tipo dos atributos deve ser o mesmo dos dados entregues a você pela interface SQL – não faça nenhuma conversão neste estágio. Você pode considerar o uso do *Mapeamento de Chave Estrangeira* (233), mas também pode deixar as chaves estrangeiras como estiverem. Você pode usar o *Registro Ativo* com visões ou tabelas, ainda que as atualizações por meio de visões sejam obviamente mais difíceis. Visões são particularmente úteis para a geração de relatórios.

A classe *Registro Ativo* tem, tipicamente, métodos que realizam as seguintes funções:

- Constroem uma instância do *Registro Ativo* a partir de uma linha de um conjunto de dados resultantes de uma consulta SQL.
- Constroem uma nova instância para posterior inserção na tabela.

- Encapsulam consultas SQL, comumente usadas, e retornam objetos do tipo *Registro Ativo*, pelos métodos de busca estáticos.
- Atualizam o banco de dados e nele inserem os dados do *Registro Ativo*.
- Lêem e gravam os atributos.
- Implementam alguns fragmentos de lógica de negócio.

Os métodos de leitura e gravação podem fazer algumas outras coisas inteligentes, tais como converter os tipos orientados a SQL para tipos mais apropriados para armazenamento em memória. Além disso, se você solicitar uma tabela relacionada, o método de leitura pode retornar o *Registro Ativo* apropriado, mesmo se você não estiver usando o *Campo Identidade* (215) na estrutura de dados (executando um *lookup*).

Nesse padrão as classes são convenientes, mas não escondem o fato de que um banco de dados relacional está presente. Como conseqüência, você normalmente encontrará uma quantidade menor dos outros padrões de mapeamento objeto-relacional presentes quando estiver usando o *Registro Ativo*.

O *Registro Ativo* é muito semelhante ao *Gateway de Linha de Dados* (158). A principal diferença é que o *Gateway de Linha de Dados* (158) contém apenas métodos de acesso ao banco de dados, enquanto que um *Registro Ativo* contém tanto lógica de domínio quanto acesso aos dados. Como a maioria das fronteiras em *software*, a linha divisória entre os dois padrões não é muito precisa, mas é útil.

Devido ao grande acoplamento entre o *Registro Ativo* e o banco de dados, encontro com maior freqüência métodos de busca estáticos neste padrão. Entretanto, nada existe que o impeça de separar os métodos de busca em uma classe separada, como discuti em *Gateway de Linha de Dados* (158), e isso facilita os testes.

Assim como com os outros padrões tabulares, você pode usar o *Registro Ativo* com uma visão ou pesquisa, bem como com uma tabela.

Quando Usá-lo

O *Registro Ativo* é uma boa escolha para a lógica de domínio que não seja muito complexa, como criações, leituras, atualizações e exclusões. Derivações e validações baseadas em um único registro funcionam bem nesta estrutura.

Em um projeto inicial para um *Modelo de Domínio* (126), a escolha principal recai entre o *Registro Ativo* e o *Mapeador de Dados* (170). O *Registro Ativo* tem a vantagem primária da simplicidade. É fácil construir *Registros Ativos*, e eles são fáceis de entender. O principal problema relacionado ao seu uso é que eles trabalham bem somente se os objetos do tipo *Registro Ativo* corresponderem diretamente às tabelas no banco de dados em um esquema isomórfico. Se a sua lógica de negócio for complexa, você logo irá desejar usar relacionamentos diretos entre seus objetos, coleções, herança e assim por diante. Estas técnicas não são facilmente mapeadas em *Registros Ativos*, e adicioná-las uma a uma torna-se muito confuso. Isso é o que lhe levará ao uso alternativo dos *Mapeadores de Dados* (170).

Outro argumento contra o *Registro Ativo* é o fato de que ele acopla o projeto dos objetos ao projeto do banco de dados. Isso torna mais difícil refatorar esses projetos à medida que o projeto prossegue.

O *Registro Ativo* é um bom padrão a considerar se você estiver usando o *Roteiro de Transação* (120) e estiver começando a sentir os problemas da duplicação de códi-

go e a dificuldade, muitas vezes, presente nos *Roteiros de Transação* (120) para atualizar roteiros e tabelas. Neste caso, você pode começar gradualmente a criar *Registros Ativos* e então lentamente neles refatorar o comportamento. Muitas vezes, é útil primeiro encapsular as tabelas como um *Gateway* (436) e só depois começar a mover o comportamento de modo que as tabelas evoluam para um *Registro Ativo*.

Exemplo: Uma Pessoa Simples (Java)

Este é um exemplo simples, até mesmo simplista, para mostrar como o âmago do Registro Ativo funciona. Começamos com uma classe Pessoa básica.

```
class Pessoa...

    private String sobrenome;
    private String prenome;
    private int númeroDeDependentes;
```

Também há um campo ID na superclasse.

O banco de dados tem exatamente a mesma estrutura.

```
create table pessoas (ID int primary key, sobrenome varchar,
                      prenome varchar, númeroDeDependentes int)
```

Para carregar um objeto, a classe Pessoa localiza o objeto e efetua a carga. Para isso são usados métodos estáticos na classe Pessoa.

```
class Pessoa...

    private static final String stringDaDeclaraçãoDeBusca =
            "SELECT id, sobrenome, prenome, númeroDeDependentes " +
            "  FROM pessoas "+
            "  WHERE id = ? ";
    public static Pessoa procurar(Long id ) {
        Pessoa resultado = (Pessoa) Registro.lerPessoa(id);
        if (resultado != null) return resultado;
        PreparedStatement declaraçãoDeBusca = null;
        ResultSet rs = null;
        try {
            declaraçãoDeBusca = DB.prepare(stringDaDeclaraçãoDeBusca);
            declaraçãoDeBusca.setLong(1, id.longValue( ));
            rs = declaraçãoDeBusca.executeQuery( );
            rs.next( );
            resultado = carregar(rs);
            return resultado;
        } catch (SQLException e) {
            throw new ApplicationException (e);
        } finally {
            DB.cleanUp(declaraçãoDeBusca, rs);
        }
    }
    public static Pessoa procurar (long id) {
        return procurar(new Long(id));
    }
```

```
    public static Pessoa carregar (ResultSet rs) throws SQLException {
        Long id = new Long(rs.getLong(1));
        Pessoa resultado = (Pessoa) Registro.lerPessoa(id);
        if (resultado != null) return resultado;
        String parâmetroSobrenome = rs.getString(2);
        String parâmetroPrenome = rs.getString(3);
        int parâmetroNúmeroDeDependentes = rs.getInt(4);
        resultado = new Pessoa (id, parâmetroSobrenome, parâmetroPrenome, parâmetroNúmeroDeDependentes);
        Registro.adicionarPessoa(resultado);
        return resultado;
    }
```

Para atualizar um objeto basta um simples método de instância.

```
class Pessoa...

    private static final String stringDaDeclaraçãoDeAtualização =
            "UPDATE pessoas "+
            "  SET sobrenome = ?, prenome = ?, númeroDeDependentes = ? " +
            "  WHERE id = ?";
    public void atualizar ( ) {
        PreparedStatement declaraçãoDeAtualização = null;
        try{
            declaraçãoDeAtualização = DB.prepare(stringDaDeclaraçãoDeAtualização);
            declaraçãoDeAtualização.setString(1, sobrenome);
            declaraçãoDeAtualização.setString(2, prenome);
            declaraçãoDeAtualização.setString(3, númeroDeDependentes);
            declaraçãoDeAtualização.setString(4, lerID( ).intValue( ));
            declaraçãoDeAtualização.execute( );
        } catch (Exception e) {
            throw new ApplicationException (e);
        } finally {
            DB.cleanUp(declaraçãoDeAtualização);
        }
    }
```

Inserções são, na sua maioria, também muito simples.

```
class Pessoa...

    private static final String stringDaDeclaraçãoDeInserção =
            "INSERT INTO pessoas VALUES (?, ?, ?, ?)";
    public Long inserir ( ) {
        PreparedStatement declaraçãoDeInserção = null;
        try {
            declaraçãoDeInserção = DB.prepare(stringDaDeclaraçãoDeInserção);
            setID(encontrarProximoIDdaBaseDeDados ( ));
            declaraçãoDeInserção.setInt (1, lerID.intValue( ));
            declaraçãoDeInserção.setString(2, sobrenome);
            declaraçãoDeInserção.setString(3, prenome);
            declaraçãoDeInserção.setInt(4, númeroDeDependentes);
            declaraçãoDeInserção.execute( );
            Registro.adicionarPessoa( this);
            return lerID( );
```

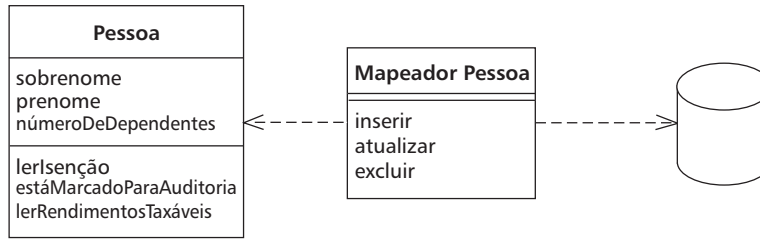
```
        } catch (SQLException e) {
            throw new ApplicationException (e);
        } finally {
            DB.cleanUp(declaraçãoDeInserção);
        }
    }
```

Qualquer lógica de negócio, tal como o cálculo de uma isenção, vai diretamente para a classe Pessoa.

```
class Pessoa...

    public Dinheiro lerIsenção( ) {
        Dinheiro baseDeIsenção = Dinheiro.dollars(1500);
        Dinheiro isençãoPorDependente = Dinheiro.dollars(750);
        return baseDeIsenção.adicionar(isençãoPorDependente.multiplicar(this.lerNúmeroDeDependentes( )));
    }
```

Mapeador de Dados (Data Mapper)

Uma camada de Mapeadores (442) que move dados entre os objetos e um banco de dados mantendo-os independentes um do outro e do próprio mapeador.



Objetos e bancos de dados relacionais usam mecanismos diferentes para estruturar os dados. Muitas partes integrantes de um objeto, tais como coleções e herança, não estão presentes em bancos de dados relacionais. Quando você cria um modelo de objetos com muita lógica de negócio, o uso destes mecanismos é valioso para melhor organizar os dados e o comportamento associado. Isso leva a esquemas diversos, isto é, o esquema dos objetos e o esquema relacional não combinam.

Ainda assim, você precisa transferir dados entre os dois esquemas, e esta transferência se torna, por si só, complexa. Se os objetos na memória conhecem a estrutura do banco de dados relacional, alterações em um tendem a se propagar para o outro.

O *Mapeador de Dados* é uma camada de *software* que separa os objetos na memória do banco de dados. Sua responsabilidade é transferir dados entre os dois e também isolá-los um do outro. Com o *Mapeador de Dados*, os objetos na memória não precisam nem mesmo saber que há um banco de dados presente. Eles não precisam conter comandos SQL e certamente não precisam ter nenhum conhecimento do esquema do banco de dados. (O esquema do banco de dados sempre ignora os objetos que o usam.) Uma vez que o *Mapeador de Dados* é uma forma de *Mapeador* (442), ele próprio não é conhecido pela camada de domínio.

Como Funciona

A separação entre o domínio e a fonte de dados é a principal função de um *Mapeador de Dados*, mas existe uma grande quantidade de detalhes que têm que ser tratados para que isso aconteça. Existe ainda uma diversidade grande de formas pelas quais camadas de mapeamento são criadas. Muitas das observações aqui são bastante gerais, porque estou tentando lhe dar uma visão geral do que você precisa para separar o joio do trigo.

Começaremos com um exemplo bastante básico de um *Mapeador de Dados*. Este é o estilo mais simples que você pode ter para esta camada e pode parecer não valer a pena usá-lo. Quando usamos exemplos simples de mapeamento de banco de dados, outros padrões são normalmente mais simples e, portanto, melhores. Você normalmente usará o *Mapeador de Dados* em casos mais complicados. No entanto, é mais fácil explicar as idéias se começamos de modo simples em um nível bastante básico.

Um caso simples envolveria, por exemplo, uma classe Pessoa e uma classe Mapeador de Pessoa. Para carregar uma pessoa do banco de dados, um cliente chama-

ria um método de busca no mapeador (Figura 10.3). O mapeador usa um *Mapa de Identidade* (196) para verificar se a pessoa já está carregada; em caso contrário, ele a carrega.

As atualizações são mostradas na Figura 10.4. Um cliente solicita que o mapeador grave um objeto do domínio. O mapeador extrai os dados do objeto de domínio e os envia para o banco de dados.

Toda a camada do *Mapeador de Dados* pode ser substituída, para efetuar testes, ou para permitir que uma única camada de domínio trabalhe com diferentes bancos de dados.

Um *Mapeador de Dados* simples apenas mapearia, campo a campo, uma tabela do banco de dados em uma classe equivalente em memória. É claro que as coisas normalmente não são simples. Os *Mapeadores* precisam de uma série de estratégias para lidar com classes que se transformam em campos múltiplos, classes que têm múltiplas tabelas, classes com herança, além de ter de conectar os objetos uns aos outros assim que eles tiverem sido extraídos do banco. Os vários padrões para o mapeamento objeto-relacional apresentados neste livro lidam todos com essas estratégias. É normalmente mais fácil desenvolver esses padrões com um *Mapeador de Dados* do que com as outras alternativas de organização.

Quando se trata de inserções e atualizações, a camada de mapeamento do banco de dados precisa saber que objetos mudaram, quais novos objetos foram criados e quais foram destruídos. Ela também tem que encaixar toda essa carga de trabalho em um *contexto* transacional. O padrão *Unidade de Trabalho* (184) é uma boa maneira de organizar isso.

A Figura 10.3 sugere que uma única requisição a um método de busca resulta em uma única consulta SQL. Isso não é sempre verdade. Carregar um pedido típico, com múltiplas linhas de pedido pode envolver a carga das linhas também. A requisição do cliente normalmente leva a um conjunto de objetos sendo carregados: o projetista do mapeador decide exatamente quanto pegar de cada vez. A essência aqui é minimizar as consultas ao banco de dados, assim, tipicamente, os métodos de pesquisa precisam conhecer um pouco sobre a forma como os clientes usam os objetos de modo a poder tomar as melhores decisões na carga dos dados.

O exemplo apresentado leva a casos em que é necessário carregar diversas classes de objetos de domínio a partir de uma única consulta. Se você quiser carregar pedidos e linhas de pedidos, normalmente será mais rápido executar uma única consulta que junte as tabelas dos pedidos e das linhas dos pedidos. Você então usa o conjunto resultante para carregar as instâncias do pedido e das linhas (página 239-240).

Uma vez que os objetos são bastante interconectados, você, em algum ponto, tem de parar de trazer os dados. De outra forma, você possivelmente irá trazer todo o banco de dados como resultado de uma única consulta. Mais uma vez, as camadas de mapeamento têm uma série de técnicas para lidar com este problema ao mesmo tempo em que, usando a *Carga Tardia* (200), minimizam o impacto sobre os objetos na memória. Por esse motivo, os objetos na memória não podem desconhecer completamente a camada de mapeamento. Eles podem precisar conhecer os métodos de pesquisa e alguns outros mecanismos.

Uma aplicação pode ter um ou vários *Mapeadores de Dados*. Se você estiver codificando explicitamente seus mapeadores, é melhor usar um para cada classe do domínio ou raiz de uma hierarquia de domínio. Se você estiver usando um *Mapeamento de Metadados* (295), você pode ter êxito com apenas uma única classe de mapeamento. Neste último caso, o problema limitante são os métodos de pesquisa. Em uma

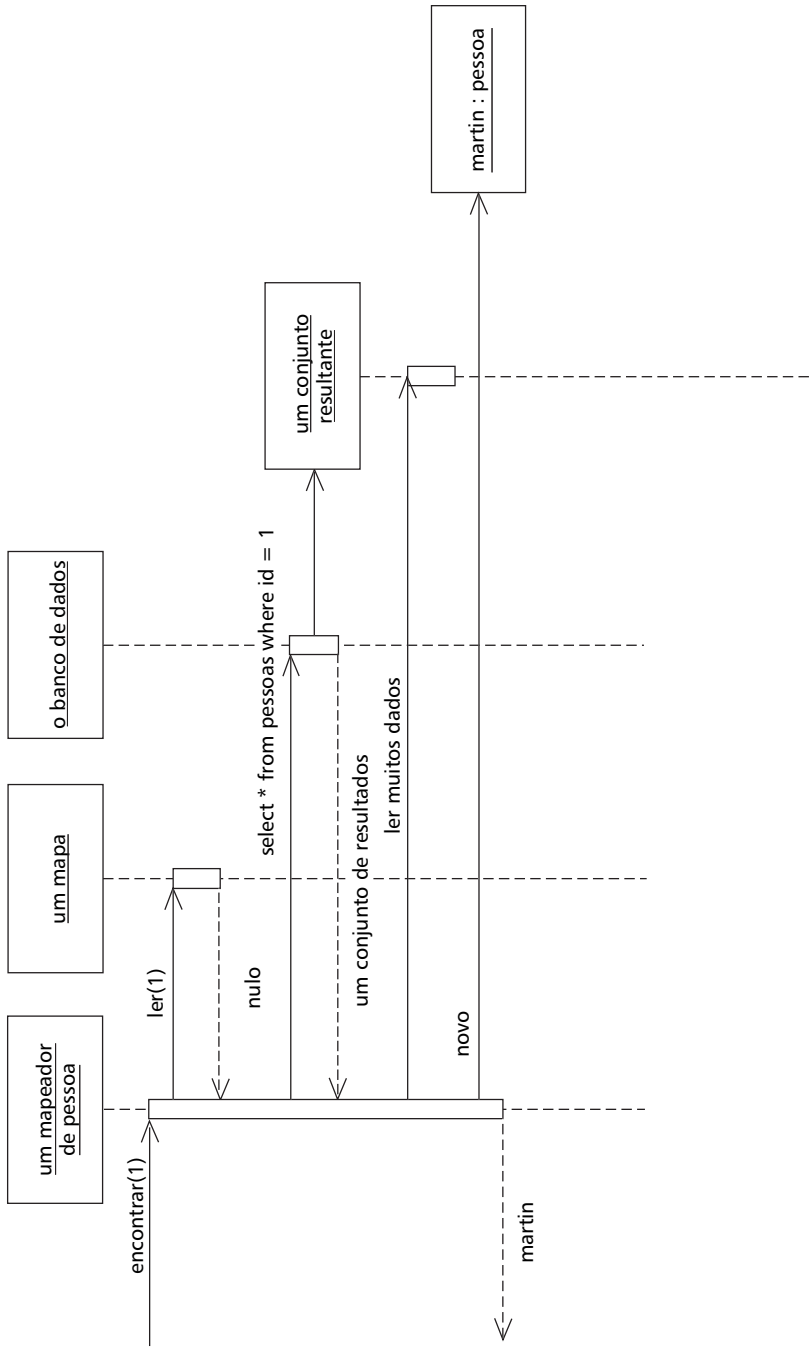


Figura 10.3 Trazendo dados de um banco de dados.

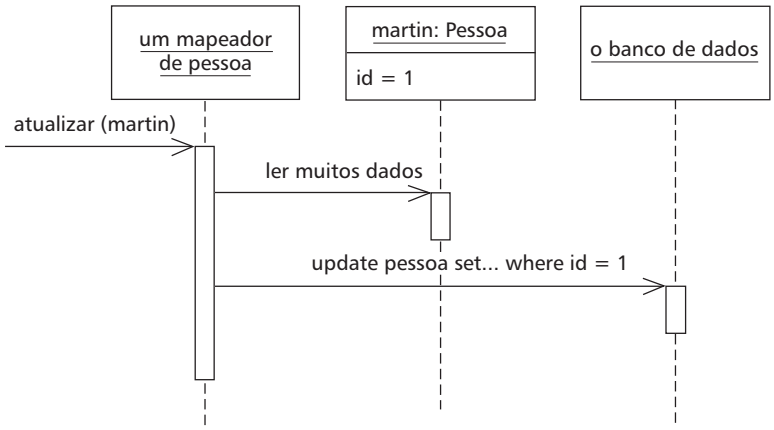


Figura 10.4 Atualizando dados.

aplicação grande, pode ser excessivo ter um único mapeador com muitos métodos de busca, de modo que faz sentido dividir estes métodos por classe de domínio ou raiz de hierarquia de domínio. Desse modo, você obtém um número grande de pequenas classes de busca, mas é mais fácil para um desenvolvedor localizar o método de busca de que ele precisa.

Assim como com qualquer procedimento de busca em um banco de dados, os métodos de busca precisam usar um *Mapa de Identidade* (196) a fim de persistir a identidade dos objetos lidos do banco de dados. Você pode usar um *Registro* (448) ou *Mapas de Identidade* (196), ou você pode fazer cada método de busca armazenar um *Mapa de Identidade* (196) (supondo que exista apenas um método de busca por classe por sessão).

Manipulando Métodos de Pesquisa Para trabalhar com um objeto, você tem que carregá-lo do banco de dados. Normalmente a camada de apresentação dá início ao processo carregando alguns objetos iniciais. O controle passa então para a camada de domínio, momento em que o fluxo de controle do código passa a se mover de objeto em objeto usando as associações entre eles. Este mecanismo funciona efetivamente desde que a camada de domínio tenha todos os objetos de que precisa carregados na memória, ou então que você use a *Carga Tardia* (200) para carregar objetos adicionais quando necessário.

Ocasionalmente, você pode precisar que os objetos do domínio invoquem métodos de busca no *Mapeador de Dados*. No entanto, descobri que, com uma boa *Carga Tardia* (200), você pode completamente evitar isso. Para aplicações mais simples, no entanto, pode não valer a pena tentar gerenciar tudo com associações e *Carga Tardia* (200). Ainda assim, não é desejável introduzir uma dependência dos seus objetos de domínio com o seu *Mapeador de Dados*.

Você pode resolver esse dilema usando uma *Interface Separada* (445). Ponha todos os métodos de busca necessários para o código do domínio em uma classe de interface que você pode colocar no pacote do domínio.

Mapeando Dados em Atributos do Domínio Os mapeadores precisam ter acesso aos atributos nos objetos do domínio. Isso, freqüentemente pode ser um problema, porque você precisa de métodos públicos para dar suporte aos mapeadores que não são necessários para a lógica do domínio. (Estou pressupondo que você não irá cometer o pecado capital de tornar os atributos públicos.) Não existe uma resposta fácil para este problema. Você poderia usar um nível mais baixo de visibilidade empacotando os mapeadores próximo aos objetos do domínio (o que corresponderia em Java a colocá-los no mesmo pacote), mas isso confunde o cenário dependências, porque você não quer que outras partes do sistema que conheçam os objetos do domínio conheçam também os mapeadores. Você pode usar reflexão, que freqüentemente pode contornar as regras de visibilidade da linguagem. Isso é mais lento, mas a velocidade menor pode ser tão insignificante quanto um erro de arredondamento, comparada ao tempo gasto pela chamada SQL. Você pode ainda usar métodos públicos, mas protegê-los com um campo de *status* de modo que eles gerem uma exceção se forem usados fora do contexto de uma carga do banco de dados. Neste caso, nomeie-os de modo que não sejam confundidos com métodos de acesso regulares (*getters* e *setters*).

Associada a esta questão, está a da criação do objeto. Em essência, você tem duas opções. Uma é criar o objeto com um **construtor rico** de modo que ele seja criado, ao menos, com todos os dados obrigatórios. A outra opção é criar um objeto vazio e então povoá-lo com os dados obrigatórios. Normalmente prefiro a primeira opção, uma vez que é bom ter um objeto bem formado desde o início. Isso também significa que, se você tiver um campo imutável, você pode garantir que ele não será alterado, simplesmente não fornecendo nenhum método para alterar seu valor.

O problema com um construtor rico é que você tem de estar ciente das referências cíclicas. Se você tiver dois objetos que referenciam um ao outro, cada vez que você tentar carregar um deles, ele tentará carregar o outro, que por sua vez tentará carregar o primeiro, e assim por diante, até que você fique sem espaço na pilha. Para evitar esse problema, é necessário um código condicional especial, freqüentemente usando a *Carga Tardia* (200). A codificação deste código condicional especial é confusa, de modo que vale a pena tentar passar sem isso. Você pode fazê-lo criando um **objeto vazio**. Use um construtor sem parâmetros para criar um objeto em branco e insira esse objeto vazio imediatamente no *Mapa de Identidade* (196). Desta forma, se você tiver um ciclo, o *Mapa de Identidade* (196) retornará um objeto de modo a interromper a carga recursiva.

Usar um objeto vazio como esse significa que você pode precisar de alguns métodos de escrita para os valores que forem verdadeiramente imutáveis quando o objeto for carregado. Uma combinação de uma convenção de nomes apropriada e, talvez, algumas sentinelas para verificação de *status* pode resolver isso. Você também pode usar reflexão para a carga de dados.

Mapeamentos Baseados em Metadados Uma das decisões que você precisa tomar com respeito ao armazenamento da informação é como os atributos nos objetos do domínio são mapeados para colunas no banco de dados. A maneira mais simples, e muitas vezes a melhor, de fazer isso é com código explícito, o que requer uma classe de mapeamento para cada objeto do domínio. O mapeador realiza o mapeamento por meio de atribuições e tem campos (usualmente *strings* constantes) para armazenar a consulta SQL para acesso ao banco de dados. Uma alternativa é usar o *Mapeamento de Metadados* (295), que armazena os metadados como dados, em uma classe

ou em um arquivo separado. A grande vantagem dos metadados é a de que todas as variações nos mapeadores podem ser tratadas por meio de dados sem a necessidade de código fonte adicional, seja pela geração de código ou por meio de programação reflexiva.

Quando Usá-lo

A principal ocasião para usar o *Mapeador de Dados* é quando você quiser que o esquema do banco de dados e o modelo de objetos evoluam independentemente. A situação mais comum para isso é com um *Modelo de Domínio* (126). O principal benefício do *Mapeador de Dados* é que, quando você estiver trabalhando no modelo do domínio, você pode ignorar o banco de dados, tanto na fase de projeto quanto na de construção e teste. Os objetos do domínio não têm qualquer conhecimento da estrutura do banco de dados porque toda a correspondência é feita pelos mapeadores.

Isso lhe ajuda no código porque você pode entender e trabalhar com os objetos do domínio sem ter que entender como eles estão armazenados no banco de dados. Você pode modificar o *Modelo do Domínio* (126) ou o banco de dados sem ter que alterar o outro. Em mapeamentos complicados, especialmente aqueles envolvendo bancos de dados já existentes, isso é muito valioso.

O preço, é claro, é a camada adicional não-existente se você usa o *Registro Ativo* (165), de modo que a chave para o uso ou não deste padrão é a complexidade da lógica de negócio. Se sua lógica de negócio for razoavelmente simples, você provavelmente não precisa de um *Modelo de Domínio* (126) ou de um *Mapeador de Dados*. Lógica mais complicada leva-o a um *Modelo de Domínio* (126) e, conseqüentemente, a um *Mapeador de Dados*.

Eu não escolheria um *Mapeador de Dados* sem um *Modelo de Domínio* (126), mas posso usar o *Modelo de Domínio* (126) sem um *Mapeador de Dados*? Se o modelo do domínio for bastante simples e o banco de dados estiver sob o controle do desenvolvedor do modelo do domínio, então é razoável que os objetos do domínio acessem diretamente o banco de dados com o *Registro Ativo* (165). Efetivamente, isso coloca a lógica de mapeamento descrita aqui nos próprios objetos do domínio. À medida que as coisas se tornam mais complicadas, é melhor refatorar o código de acesso ao banco de dados em uma camada separada.

Lembre-se de que você não tem que criar uma camada completa de mapeamento de dados. Esta é uma tarefa bastante complicada, e existem produtos disponíveis que fazem isso para você. Na maioria dos casos, recomendo comprar uma camada para o mapeamento no banco de dados em vez de criar uma você mesmo.

Exemplo: Um Mapeador Simples de Banco de Dados (Java)

Este exemplo mostra um uso absurdamente simples de um *Mapeador de Dados* para lhe dar uma idéia da sua estrutura básica. Nosso exemplo consiste em uma classe pessoa e uma tabela isomórfica pessoa.

```
class Pessoa...

    private String sobrenome;
    private String prenome;
    private int númeroDeDependentes;
```

O esquema do banco de dados se parece com:

```
create table pessoas (ID int primary key, sobrenome varchar,
                      prenome varchar, númeroDeDependentes int)
```

Usaremos o caso simples aqui onde a classe *Mapeador de Pessoa* também implementa o método de busca e o *Mapa de Identidade* (196). Entretanto, acrescentei um mapeador abstrato *Camada Supertipo* (444) para indicar para onde posso extrair algum comportamento comum. A carga envolve a verificação de que o objeto já não esteja no *Mapa de Identidade* (196) e, então, a busca dos dados a partir do banco de dados.

O comportamento de busca começa no *Mapeador de Pessoa*, o qual encapsula as chamadas para um método abstrato de busca pelo ID.

```
class MapeadorDePessoa...

    protected String declaraçãoDeBusca ( ) {
        return "SELECT " + COLUNAS +
            "  FROM pessoas" +
            "  WHERE id = ?";
    }
    public static final String COLUNAS = " id, sobrenome, prenome, númeroDeDependentes ";
    public Pessoa buscar (Long id)
        return (Pessoa) buscaAbstata (id);
    }
    public Pessoa buscar (long id) {
        return buscar(new Long(id));
    }

class MapeadorAbstrato...

    protected Map mapaCarregado = new HashMap( );
    abstract protected String declaraçãoDeBusca( );
    protected ObjetoDoDomínio buscaAbstrata(Long id) {
        ObjetoDoDomínio resultado = (ObjetoDoDomínio) mapaCarregado.ler(id);
        if (resultado != null) return resultado;
        PreparedStatement declaraçãoDeBusca = null;
        try{
            declaraçãoDeBusca = DB.prepare(declaraçãoDeBusca( ));
            declaraçãoDeBusca.setLong(1, id.longValue( ));
            ResultSet rs = declaraçãoDeBusca.executeQuery( );
            rs.next( );
            resultado = carregar(rs);
            return resultado;
        } catch (SQLException e) {
            throw new ApplicationException (e);
        } finally {
            DB.cleanUp(declaraçãoDeBusca);
        }
    }
```

O método de busca chama o método de carga, o qual é dividido entre o mapeador abstrato e o mapeador de pessoa. O mapeador abstrato verifica o ID, extraindo-o dos dados e registrando o novo objeto no *Mapa de Identidade* (196).

```
class MapeadorAbstrato...

    protected ObjetoDoDomínio carregar (ResultSet rs) throws SQLException {
        Long id = new Long(rs.getLong(1));
        if (mapaCarregado.containsKey(id)) return (ObjetoDoDomínio) mapaCarregado.ler(id);
        ObjetoDoDomínio resultado = fazerCarga(id, rs);
        mapaCarregado.put(id, resultado);
        return resultado;
    }
    abstract protected ObjetoDoDomínio fazerCarga (Long id, ResultSet rs)
    throws SQLException;

class MapeadorDePessoa...

    protected ObjetoDoDomínio fazerCarga (Long id, ResultSet rs) throws SQLException {
        String parâmetroSobrenome = rs.getString(2);
        String parâmetroPrenome = rs.getString(3);
        int parâmetroNúmeroDeDependentes = rs.getInt(4);
        return new Pessoa (id, parâmetroSobrenome, parâmetroPrenome,
        parâmetroNúmeroDeDependentes);
    }
```

Perceba que o *Mapa de Identidade* (196) é verificado duas vezes, uma pelo método `buscaAbstrata` e outra método `carregar`. Há uma razão para esta loucura.

Preciso verificar o mapa no método de busca porque, se o objeto já estiver lá, posso evitar uma ida ao banco de dados – sempre quero evitar essa ida ao banco de dados, se puder. Contudo, também preciso verificar na carga porque é possível que eu tenha consultas que não posso ter certeza de resolver no *Mapa de Identidade* (196). Suponha que eu queira encontrar todas as pessoas cujo sobrenome satisfaça algum padrão de busca. Não posso ter certeza de que eu já tenha todas essas pessoas carregadas, de modo que tenho que ir ao banco de dados e executar uma consulta.

```
class MapeadorDePessoa...

    private static String declaraçãoEncontrarSobrenome =
            "SELECT " + COLUNAS +
            "  FROM pessoas " +
            "  WHERE UPPER(sobrenome) like UPPER(?)" +
            "  ORDER BY sobrenome";
    public List encontrarPeloSobrenome (String nome) {
        PreparedStatement dec = null;
        ResultSet rs = null;
        try {
            dec = DB.prepare(declaraçãoEncontrarSobrenome);
            dec.setString (1, nome);
            rs = dec.executeQuery( );
            return carregarTodos(rs);
        } catch (SQLException e) {
            throw new ApplicationException (e);
        } finally {
            DB.cleanUp (dec, rs);
        }
    }
```

```
class MapeadorAbstato...

    protected List carregarTodos (ResultSet rs) throws SQLException {
        List resultado = new ArrayList( );
        while (rs.next( ))
            resultado.add(carregar(rs));
        return resultado;
    }
```

Quando faço isso posso extrair algumas linhas do conjunto resultante que correspondam a pessoas que eu já tenha carregado. Tenho que me assegurar de que eu não faça uma duplicata, então tenho que verificar o *Mapa de Identidade* (196) novamente.

Escrever um método de busca dessa forma em cada subclasse que precise dele implica em uma codificação básica, porém repetitiva, o que posso eliminar fornecendo um método genérico.

```
class MapeadorAbstrato...

    public List encontrarMuitos (OrigemDaDeclaração origem) {
        PreparedStatement dec = null;
        ResultSet rs = null;
        try {
            dec = DB.prepare(origem.sql( ));
            for(int i =0; i< origem.parâmetros( ).length; i++)
                dec.setObject(i+1, origem.parâmetros( )[i]);
            rs = dec.executeQuery( );
            return carregarTodos(rs);
        } catch (SQLException e) {
            throw new ApplicationException (e);
        } finally {
            DB.cleanUp (dec, rs);
        }
    }
```

Para isto funcionar, preciso de uma interface que encapsule tanto a *string* SQL quanto a carga dos parâmetros na declaração preparada.

```
interface OrigemDaDeclaração...

    String sql;
    Object [ ] parâmetros ( );
```

Posso, então, usar esse recurso fornecendo uma implementação apropriada como uma classe interna.

```
class MapeadorDePessoa...

    public List encontrarPeloSobrenome2 (String padrão) {
        return encontrarMuitos (new EncontrarPeloSobrenome (padrão));
    }
    static class EncontrarPeloSobrenome implements OrigemDaDeclaração{
        private String sobrenome;
        public EncontrarPeloSobrenome (String sobrenome) {
            this.sobrenome = sobrenome;
```

```
        }
        public String sql ( ) {
            return
                "SELECT " + COLUNAS +
                "  FROM pessoas " +
                "  WHERE UPPER (sobrenome) like UPPER(?)" +
                "  ORDER BY sobrenome";
        }
        public Object[ ] parâmetros ( ) {
            Object[ ] resultado = {sobrenome};
            return resultado;
        }
    }
```

Esse tipo de trabalho pode ser executado em outros lugares onde houver código repetitivo de chamada de declarações. De modo geral, fiz os exemplos aqui mais diretos para torná-los mais fáceis de seguir. Se você perceber que está escrevendo uma grande quantidade de código direto repetitivo, deve considerar fazer algo semelhante.

Com a atualização, o código JDBC é específico para o subtipo.

```
class MapeadorDePessoa...

    private static final String stringDaDeclaraçãoDeAtualização =
            "UPDATE pessoas "+
            "  SET sobrenome = ?, prenome = ?, númeroDeDependentes = ? " +
            "  WHERE id = ?";
    public void atualizar (Pessoa sujeito) {
        PreparedStatement declaraçãoDeAtualização = null;
        try{
            declaraçãoDeAtualização = DB.prepare(stringDaDeclaraçãoDeAtualização);
            declaraçãoDeAtualização.setString(1, sujeito.lerSobrenome());
            declaraçãoDeAtualização.setString(2, sujeito.lerPrenome());
            declaraçãoDeAtualização.setInt(3, sujeito.lerNúmeroDeDependentes());
            declaraçãoDeAtualização.setInt(4, sujeito.lerID( ).intValue( ));
            declaraçãoDeAtualização.execute( );
        } catch (Exception e) {
            throw new ApplicationException (e);
        } finally {
            DB.cleanUp(declaraçãoDeAtualização);
        }
    }
```

Para a inserção, algum código pode ser fatorado para a *Camada Supertipo* (444).

```
class MapeadorAbstrato...

    public Long inserir (ObjetoDoDomínio sujeito) {
        PreparedStatement declaraçãoDeInserção = null;
        try {
            declaraçãoDeInserção = DB.prepare(declaraçãoDeInserção( ));
            sujeito. gravarID (encontrarPróximoIdNoBancoDeDados( ));
            declaraçãoDeInserção.setInt (1, sujeito.lerID( ).intValue( ));
            fazerInserção (sujeito, declaraçãoDeInserção);
```

```
            declaraçãoDeInserção.execute( );
            mapaCarregado.put (sujeito.lerID( ), sujeito);
            return sujeito.lerID( );
        } catch (SQLException e) {
            throw new ApplicationException (e);
        } finally {
            DB.cleanUp(declaraçãoDeInserção);
        }
    }
    abstract protected String declaraçãoDeInserção( );
    abstract protected void fazerInserção (ObjetoDoDomínio sujeito, PreparedStatement declaração-
DeInserção)
        throws SQLException;

class MapeadorDePessoa...

    protected String declaraçãoDeInserção ( ) {
        return "INSERT INTO pessoas VALUES (?, ?, ?, ?)";
    }
    protected void fazerInserção (
            ObjetoDoDomínio sujeitoAbstrato,
            PreparedStatement dec)
            throws SQLException
    {
        Pessoa sujeito = (Pessoa) sujeitoAbstrato;
        dec.setString (2, sujeito.lerSobrenome( ));
        dec.setString (3, sujeito.lerPrenome( ));
        dec.setInt (4, sujeito.lerNúmeroDeDependentes( ));
    }
```

Exemplo: Separando os Métodos de Busca (Java)

Para permitir que os objetos do domínio invoquem o comportamento de busca, posso usar uma *Interface Separada* (445) para separar as interfaces de busca dos mapeadores (Figura 10.5). Posso colocar estas interfaces de busca em um pacote separado que seja visível à camada de domínio ou, como neste caso, posso colocá-las na própria camada do domínio.

Uma das pesquisas mais comuns é aquela que busca um objeto pelo ID que o identifica. A maior parte deste processamento é bastante genérico, de modo que ele pode ser tratado por uma *Camada Supertipo* (444) apropriada. Tudo que é preciso é uma *Camada Supertipo* (444) para objetos do domínio que conheça os IDs.

A interface para pesquisas reside na interface para buscas. Normalmente, é melhor não torná-la genérica porque você precisa conhecer o tipo de retorno.

```
interface buscadorDeArtista...

    Artista buscar (Long id);
    Artista buscar (long id);
```

É melhor declarar a interface para buscas no pacote do domínio com os métodos de busca armazenados em um *Registro* (448). Neste caso, fiz com que a classe mapeadora implementasse a interface para buscas.

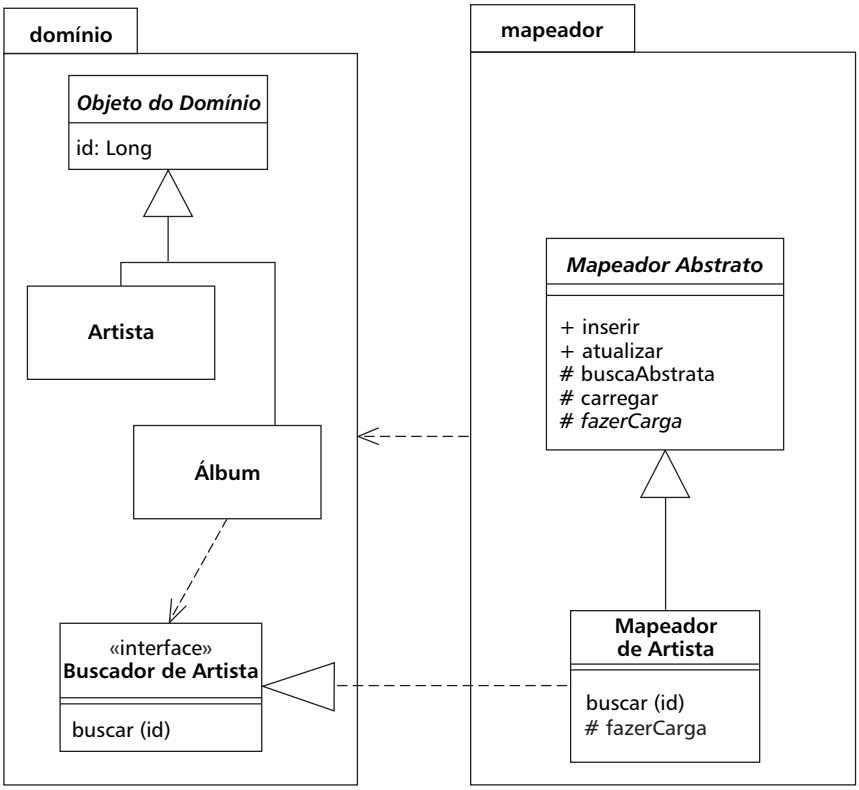


Figura 10.5 Definindo uma interface de busca no pacote do domínio.

```
class MapeadorDeArtista implements buscadorDeArtista...

    public Artista buscar (Long id) {
        return (Artist) buscaAbstrata (id);
    }
    public Artista buscar (long id) {
        return buscar(new Long(id));
    }
```

A parte principal do método de busca é executada pela *Camada Supertipo* (444) do mapeador, que verifica o *Mapa de Identidade* (196) para ver se o objeto já está na memória. Se não estiver, ele completa um *prepared statement* (carregado pelo mapeador de artista) e o executa.

```
class MapeadorAbstrato...

    abstract protected String declaraçãoDeBusca ( );
    protected Map mapaCarregado = new HashMap( );
    protected ObjetoDoDomínio buscaAbstrata (Long id) {
        ObjetoDoDomíno resultado = (ObjetoDoDomínio) mapaCarregado.get(id);
        if (resultado != null) return resultado;
        PreparedStatement dec = null;
        ResultSet rs = null;
```

```
        try {
            dec = DB.prepare(declaraçãoDeBusca( ));
            dec.setLong(1, id.longValue( ));
            rs = dec.executeQuery( );
            rs.next( );
            resultado = carregar(rs);
            return resultado;
        } catch (SQLException e) {
            throw new ApplicationException (e);
        } finally {cleanUp(dec, rs);
        }
    }
```

```
class MapeadorDeArtista...

    protected String declaraçãoDeBusca ( ) {
        return "select " + LISTA_DE_COLUNAS + " from artistas art where ID = ?";
    }
    public static String LISTA_DE_COLUNAS = "art.ID, art.nome";
```

A parte do comportamento referente à busca diz respeito à obtenção de um objeto existente ou de um novo. A parte referente à carga diz respeito a colocar os dados do banco de dados em um novo objeto.

```
class MapeadorAbstrato...

    protected ObjetoDoDomínio carregar (ResultSet rs) throws SQLException {
        Long id = new Long (rs.getLong (id"));
        if (mapaCarregado.containsKey (id)) return (ObjetoDoDomínio) mapaCarregado.get(id);
        ObjetoDoDomínio resultado = fazerCarga (id, rs);
        mapaCarregado.put (id, resultado);
        return resultado;
    }
    abstract protected ObjetoDoDomínio fazerCarga(Long id, ResultSet rs)
    throws SQLException;
```

```
class MapeadorDeArtista...

    protected ObjetoDoDomínio fazerCarga (Long id, ResultSet rs) throws SQLException {
        String nome = rs.getString("nome");
        Artista resultado = new Artista (id, nome);
        return resultado;
    }
```

Observe que o método de carga também verifica o *Mapa de Identidade* (196). Embora, neste caso, isso seja redundante, a carga pode ser chamada por outros métodos de busca que ainda não fizeram esta verificação. Neste esquema, tudo que uma subclasse tem que fazer é desenvolver um método `fazerCarga` para carregar os dados reais necessários, além de retornar uma declaração apropriada a partir do método `declaraçãoDeBusca`.

Você também pode fazer uma busca baseada em uma consulta. Suponha que tenhamos um banco de dados de faixas e álbuns e queiramos um método de busca que encontre todas as faixas em um álbum específico. Novamente, a interface declara os métodos de busca.

```
interface buscadorDeFaixas...

    Faixa buscar (Long id);
    Faixa buscar (long id);
    List buscarDoAlbum (Long idDoÁlbum);
```

Uma vez que este é um método de busca específico para esta classe, ele é implementado em uma classe específica, tal como a classe mapeadora de faixas, em vez de em uma *Camada Supertipo* (444). Assim como com qualquer método de busca, existem dois métodos para a implementação. Um deles estabelece a expressão preparada (*prepared statement*) e o outro encapsula a chamada para a expressão preparada e interpreta os resultados.

```
class MapeadorDeFaixa...

    public static final String declaraçãoBuscaDoÁlbum =
            "SELECT ID, seq, idDoÁlbum, título "+
            "FROM faixas "+
            "WHERE IDdoÁlbum = ? ORDER BY seq";
    public List buscarPorÁlbum (Long idDoÁlbum) {
        PreparedStatement dec = null;
        ResultSet rs = null;
        try {
            dec = DB.prepare (declaraçãoBuscaDoÁlbum );
            dec.setLong (1, idDoÁlbum.longValue( ));
            rs = dec.executeQuery( );
            List resultado = new ArrayList( );
            while (rs.next( ))
                resultado.add(carregar(rs));
            return resultado;
        } catch (SQLException e) {
            throw new ApplicationException (e);
        } finally {cleanUp(dec, rs);
        }
    }
```

O método de busca chama um método de carga para cada linha no conjunto resultante. Este método tem a responsabilidade de criar o objeto na memória e carregá-lo com os dados. Como no exemplo anterior, parte disso pode ser tratada em uma *Camada Supertipo* (444), incluindo a verificação do *Mapa de Identidade* (196) para ver se algo já está carregado.

Exemplo: Criando um Objeto Vazio (Java)

Há duas abordagens básicas para carregar um objeto. Uma é criar um objeto completamente válido com um construtor, que é o que fiz nos exemplos acima. Isso resulta no seguinte código construtor:

```
class MapeadorAbstrato...

    protected ObjetoDoDomínio carregar (ResultSet rs) throws SQLException {
        Long id = new Long(rs.getLong(1));
        if (mapaCarregado.containsKey(id)) return (ObjetoDoDomínio) mapaCarregado.get(id);
        ObjetoDoDomínio resultado = fazerCarga(id, rs);
```

```
        mapaCarregado.put (id, resultado);
        return resultado;
    }
    abstract protected ObjetoDoDomínio fazerCarga (Long id, ResultSet rs) throws SQLException;
```

```
class MapeadorDePessoa...

    protected ObjetoDoDomínio fazerCarga (Long id, ResultSet rs) throws SQLException {
        String parâmetroSobrenome = rs.getString(2);
        String parâmetroPrenome = rs.getString(3);
        int parâmetroNúmeroDeDependentes = rs.getInt(4);
        return new Pessoa (id, parâmetroSobrenome, parâmetroPrenome, parâmetroNúmeroDeDependentes);
    }
```

A alternativa é criar um objeto vazio e, mais tarde, carregá-lo com os métodos de gravação.

```
class MapeadorAbstrato...

    protected ObjetoDoDomínioEL carregar (ResultSet rs) throws SQLException {
        Long id = new Long(rs.getLong(1));
        if (mapaCarregado.containsKey(id)) return (ObjetoDoDomínioEL) mapaCarregado.get(id);
        ObjetoDoDomínioEL resultado = criarObjetoDoDomínio( );
        resultado.gravarID(id);
        mapaCarregado.put(id, resultado);
        fazerCarga (resultado, rs);
        return resultado;
    }
    abstract protected ObjetoDoDomínioEL criarObjetoDoDomínio( );
    abstract protected void fazerCarga (ObjetoDoDomínioEL obj, ResultSet rs) throws SQLException;
```

```
class MapeadorDePessoa...

    protected ObjetoDoDomínioEL criarObjetoDoDomínio( ) {
        return new Pessoa( );
    }
    protected void fazerCarga (ObjetoDoDomínioEL obj, ResultSet rs) throws SQLException {
        Pessoa pessoa = (Pessoa) obj;
        pessoa.carregarSobrenomeBD (rs.getString(2));
        pessoa.gravarPrenome(rs.getString(3));
        pessoa.gravarNúmeroDeDependentes(rs.getInt(4));
    }
```

Perceba que estou usando um tipo diferente de objeto de domínio, *Camada Supertipo* (444) aqui, porque quero controlar o uso de métodos de gravação. Digamos que eu queira que o sobrenome de uma pessoa seja um campo imutável. Neste caso, não quero alterar o valor do campo, uma vez que este tenha sido gravado, então acrescento um campo *status* no objeto do domínio.

```
class ObjetoDoDomínioEL...

    private int estado = CARREGANDO;
    private static final int CARREGANDO = 0;
    private static final int ATIVO = 1;
    public void fiqueAtivo ( ) {
        estado = ATIVO;
    }
```

Posso, então, verificar o valor deste durante uma carga.

```
class Pessoa...

    public void carregarSobrenomeBD (String sobrenome) {
        assertEstadoÉCarregando ( );
        this.sobrenome = sobrenome;
    }

class ObjetoDoDomínioEL...

    void assertEstadoÉCarregando ( ) {
        Assert.isTrue (estado == CARREGANDO);
    }
```

O que me desagrada nisso é termos agora um método na interface que a maioria dos clientes da classe Pessoa não pode usar. Este é um argumento para o mapeador usando reflexão gravar o campo, ignorando assim completamente os mecanismos de proteção Java.

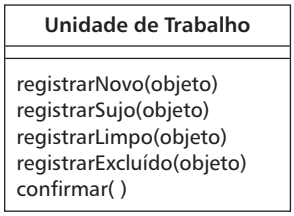
A condição guarda baseada no *status* compensa o problema? Não estou inteiramente certo. Por um lado isso irá pegar falhas causadas por pessoas chamando métodos de atualização na hora errada. Por outro lado, estas falhas são tão severas que compensam o custo do mecanismo? No momento não tenho uma opinião sólida a favor de qualquer das opções.

CAPÍTULO 11

Padrões Comportamentais Objeto-Relacionais

Unidade de Trabalho (Unit of Work)

Mantém uma lista de objetos afetados por uma transação de negócio e coordena a gravação das alterações e a resolução de problemas de concorrência.



Quando você está enviando dados de e para um banco de dados, é importante manter registro do que você alterou, senão esses dados não serão gravados de volta no banco de dados. De maneira similar, você tem que inserir novos objetos que criou e remover quaisquer objetos que apagou.

Você pode alterar o banco de dados a cada mudança no seu modelo de objetos, mas isso pode levar a muitas chamadas pequenas ao banco de dados, o que acaba sendo muito lento. Além disso, requer que você tenha uma transação aberta para a interação inteira, o que não é prático se você tiver uma transação que envia diversas solicitações. A situação é até pior se você precisar manter registro dos objetos que leu para que possa evitar leituras inconsistentes.

Uma *Unidade de Trabalho* mantém registro de tudo o que você faz durante uma transação de negócio que possa afetar seu banco de dados. Quando você tiver terminado, ela descobre tudo o que precisa ser feito para alterar o banco de dados como resultado do seu trabalho.

Como Funciona

As coisas óbvias que fazem você lidar com o banco de dados são alterações: novos objetos criados e objetos já existentes atualizados ou excluídos. A *Unidade de Trabalho* é um objeto que mantém registro dessas coisas. Assim que você começa a fazer algo que possa afetar um banco de dados, você cria uma *Unidade de Trabalho* para manter registro das alterações. Cada vez que você cria, altera ou exclui um objeto, informa a *Unidade de Trabalho*. Você também pode informá-la sobre objetos que leu para que ela possa verificar leituras inconsistentes checando se nenhum dos objetos foi alterado no banco de dados durante a transação de negócio.

A chave da *Unidade de Trabalho* é que, na hora de confirmar o trabalho, ela decide o que fazer. Abre uma transação, executa qualquer verificação de concorrência (usando *Bloqueio Offline Pessimista* (401) ou *Bloqueio Offline Otimista* (392)) e grava as alterações no banco de dados. Os programadores de aplicações nunca chamam explicitamente métodos para atualização do banco de dados. Dessa forma, eles não têm que manter registro do que foi alterado ou se preocupar a respeito de como a integridade referencial afeta a ordem na qual precisam fazer as coisas.

É claro que para isso funcionar a *Unidade de Trabalho* precisa saber de quais objetos deve manter registro. Você pode obter isso com o solicitante do objeto realizando esse trabalho ou fazendo o objeto informar à *Unidade de Trabalho*.

Com o **registro do solicitante** (Figura 11.1), o usuário de um objeto tem que se lembrar de registrar esse objeto na *Unidade de Trabalho* para atualizações. Quaisquer objetos que não estiverem registrados não serão gravados na hora da confirmação. Embora isso permita que o esquecimento cause problemas, dá flexibilidade permitindo que as pessoas façam alterações na memória que elas não queiram que sejam gravadas. Não obstante, eu argumentaria que causará muito mais confusão do que valeria a pena. É melhor fazer uma cópia explícita para esse propósito.

Com o **registro do objeto** (Figura 11.2), o ônus é removido do solicitante. O truque comum aqui é colocar métodos de registro em métodos de objetos. Carregar um objeto do banco de dados registra o objeto como limpo; os métodos de gravação registram o objeto como sujo. Para este esquema funcionar, a *Unidade de Trabalho* precisa ser passada para o objeto ou estar em um lugar bem conhecido. Passar a *Unidade de Trabalho* é tedioso, mas normalmente não há problemas em tê-la presente em algum tipo de objeto de sessão.

Mesmo o registro do objeto deixa algo a ser lembrado, ou seja, o desenvolvedor do objeto tem que se lembrar de acrescentar uma chamada de registro no lugar certo. A consistência se torna habitual, mas ainda é uma falha complicada quando perdida.

Este é um lugar natural para a geração de código gerar chamadas apropriadas, mas só funciona quando você puder separar claramente código gerado de código não-gerado. Este problema é especialmente apropriado para a programação orienta-

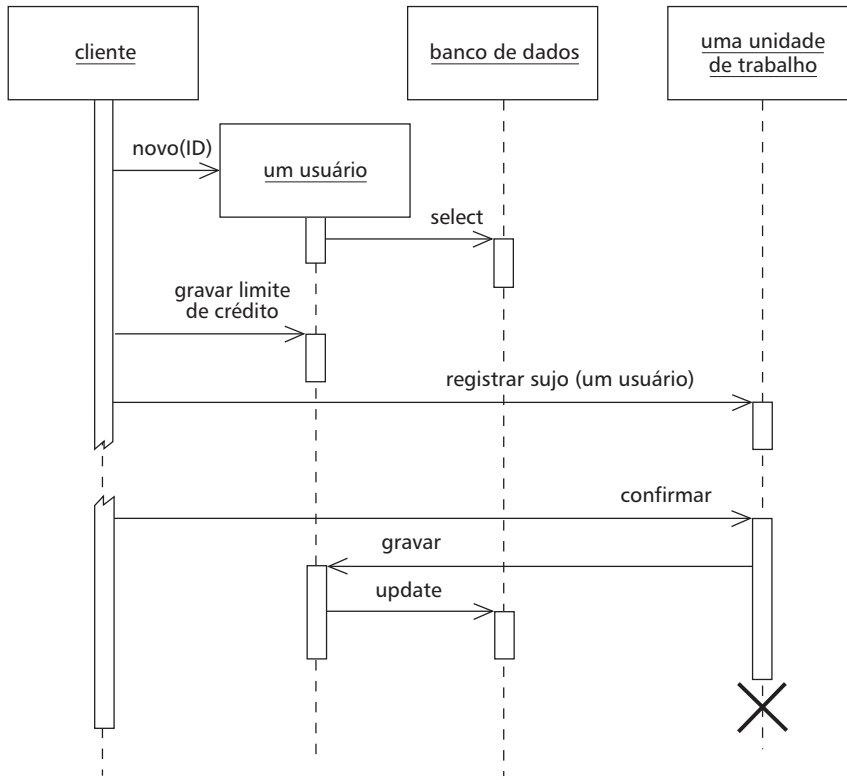


Figura 11.1 Fazendo o solicitante registrar um objeto alterado.

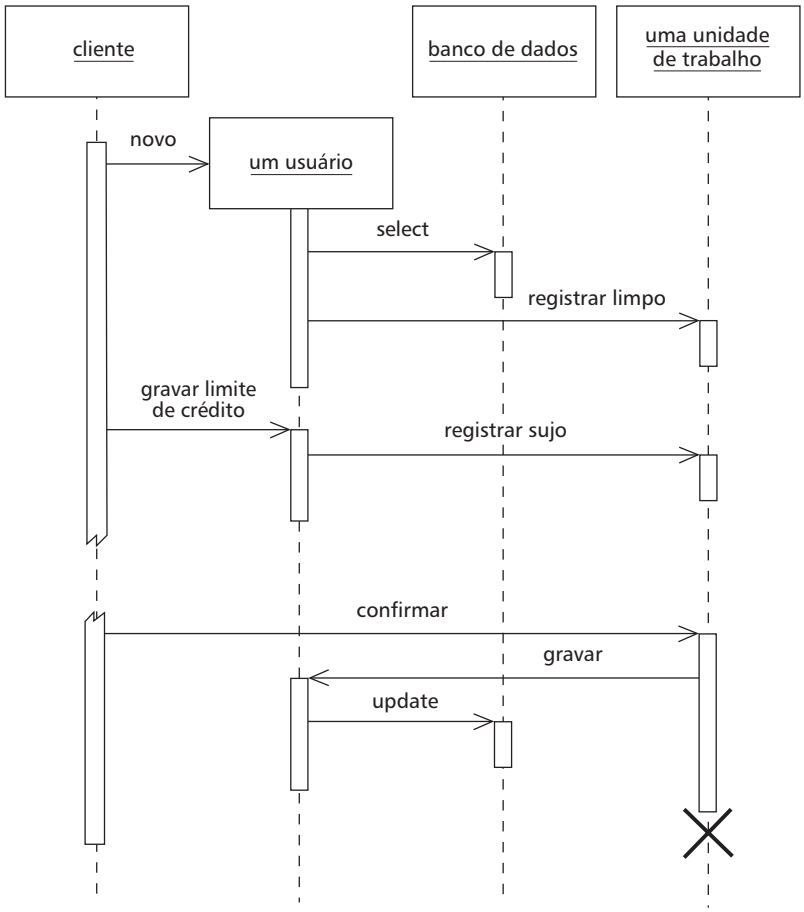


Figura 11.2 Fazer com que o objeto receptor registre a si próprio.

da a aspectos. Também me deparei com pós-processamento dos arquivos objeto para resolver isso. Neste exemplo, um pós-processador examinou todos os arquivos Java *.class*, procurou pelos métodos apropriados e inseriu chamadas de registro no *byte code*. Todo esse procedimento parece deselegante, mas ele separa o código do banco de dados do código comum. A programação orientada a aspectos fará isso de modo mais limpo com o código fonte e, quando suas ferramentas se tornarem mais comuns, espero ver esta estratégia sendo usada.

Outra técnica que vi é o **controlador de unidade de trabalho** (Figura 11.3), que o produto TOPLink usa. Aqui a *Unidade de Trabalho* lida com todas as leituras do banco de dados e registra objetos limpos toda vez que eles são lidos. Em vez de marcar objetos como sujos, a *Unidade de Trabalho* faz uma cópia na hora da leitura e então compara o objeto na hora da confirmação do trabalho. Embora isso acrescente *overhead* ao processo de confirmação, permite uma atualização seletiva de apenas aqueles campos que foram realmente modificados. Também evita chamadas de registro nos objetos do domínio. Uma abordagem híbrida é fazer cópias apenas dos objetos modificados. Isso requer registro, mas suporta atualização seletiva e reduz grandemente o *overhead* da cópia se houver muito mais leituras do que atualizações.

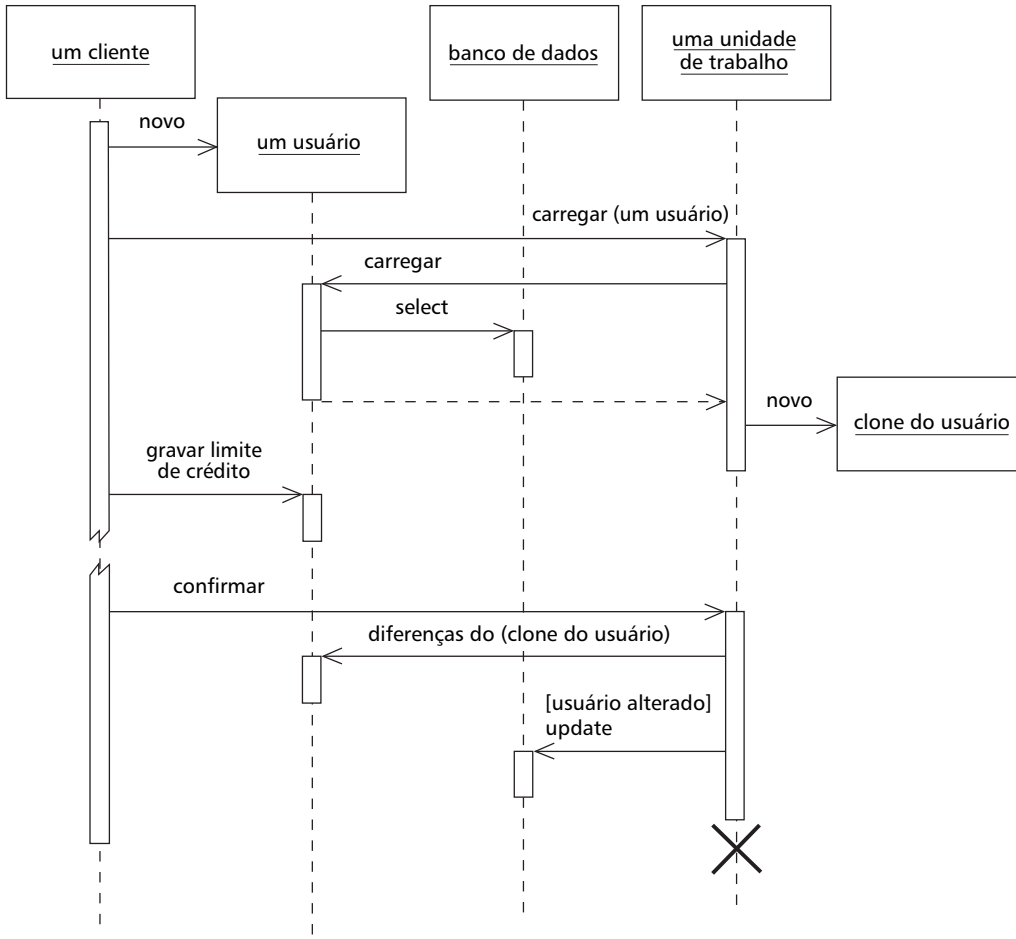


Figura 11.3 Usando a Unidade de Trabalho como controlador do acesso ao banco de dados.

A criação de objetos é freqüentemente um momento especial para considerar o registro do solicitante. Não é incomum as pessoas criarem objetos que apenas supostamente são transitórios. Um bom exemplo disso ocorre nos testes de objetos de domínio, que rodam muito mais rapidamente sem as gravações no banco de dados. O registro do solicitante pode tornar isso perceptível. Entretanto, há outras soluções, como fornecer um construtor transitório que não registre na *Unidade de Trabalho* ou, melhor ainda, fornecer um *Caso Especial* (462) de *Unidade de Trabalho* que não faça nada na hora da confirmação. Outra área onde uma *Unidade de Trabalho* pode ser útil é na atualização quando um banco de dados usa integridade referencial. Na maior parte do tempo, você pode evitar essa questão assegurando que o banco de dados apenas verifique a integridade referencial quando a transação é confirmada, em vez de em cada chamada SQL. A maioria dos bancos de dados permite isso, e se estiver disponível não há um bom motivo para não fazê-lo. Se não estiver, a *Unidade de Trabalho* é o lugar natural para lidar com a atualização. Em sistemas menores isso pode ser feito com código explícito que contenha detalhes sobre quais tabelas gravar primeiro baseado nas dependências das chaves estrangeiras. Em uma aplicação maior é

melhor usar metadados para descobrir em qual ordem gravar no banco de dados. Como você faz isso está além do escopo deste livro e é uma razão comum para usar uma ferramenta comercial. Se você mesmo tiver que fazê-lo, fui informado de que a chave do enigma é um arranjo topológico.

Você pode usar uma técnica semelhante para minimizar os *deadlocks*. Se cada transação usar a mesma seqüência de tabelas para editar, você reduz grandemente o risco de *deadlocks*. A *Unidade de Trabalho* é um lugar ideal para armazenar uma seqüência fixa de gravações em tabelas para que você sempre acesse as tabelas na mesma ordem.

Os objetos precisam ser capazes de encontrar sua *Unidade de Trabalho* corrente. Uma boa maneira de fazer isso é com um *Registro* com o escopo de uma *thread*. Outra maneira é passar a *Unidade de Trabalho* para os objetos que necessitem dela, seja em uma chamada de método ou quando você cria um objeto. Em ambos os casos, assegure-se de que apenas uma *thread* possa acessar uma *Unidade de Trabalho* – aí reside o caminho para a loucura.

A *Unidade de Trabalho* torna óbvia a questão de lidar com atualizações em lotes. A idéia por trás de uma **atualização em lote** é enviar diversos comandos SQL como uma única unidade de modo que eles possam ser processados em uma única chamada remota. Isso é especialmente importante quando muitas atualizações, inserções e exclusões são enviadas em uma rápida sucessão. Diferentes ambientes fornecem diferentes graus de suporte a atualizações em lote. JDBC tem um recurso que permite a você colocar em lote declarações individuais. Se você não tiver este recurso, pode simulá-lo criando uma *string* que tenha diversas declarações SQL e, então, submetendo-as como uma única declaração. [Nilsson] descreve um exemplo disso para plataformas Microsoft. Entretanto, se você fizer isso verifique se interfere com a pré-compilação das declarações.

A *Unidade de Trabalho* funciona com qualquer recurso transacional, não apenas bancos de dados, de modo que você também pode usá-la para coordenar filas de mensagens e monitores transacionais.

Implementação .NET

Em .NET a *Unidade de Trabalho* é feita pelo conjunto de dados desconectado. Isso a torna um padrão levemente diferente da variedade clássica. A maioria das *Unidades de Trabalho* com as quais me deparei registram e rastreiam as alterações nos objetos. .NET lê dados do banco de dados para um conjunto de dados, o qual é uma série de objetos dispostos como tabelas de um banco de dados, linhas e colunas. O conjunto de dados é basicamente uma imagem na memória do resultado de uma ou mais consultas SQL. Cada linha de dados tem o conceito de uma versão (corrente, original, proposta) e um estado (inalterado, acrescentado, excluído, modificado), o que, junto com o fato de que o conjunto de dados simula a estrutura do banco de dados, torna direta a gravação de alterações no banco de dados.

Quando Usá-la

O problema fundamental com o qual a *Unidade de Trabalho* lida é manter registro dos vários objetos que você manipulou, de modo que você saiba quais precisa considerar para sincronizar seus dados na memória com o banco de dados. Se você for capaz de fazer todo seu trabalho dentro de uma transação de sistema, os únicos objetos com os quais precisa se preocupar são aqueles que você altera. Embora a *Unidade de Trabalho* seja geralmente a melhor maneira de fazer isso, há alternativas.

Talvez a alternativa mais simples seja gravar explicitamente qualquer objeto toda vez que o alterar. O problema aqui é que você pode ter muito mais chamadas ao banco de dados do que quer já que, se você alterar um objeto em três momentos diferentes no seu trabalho, tem três chamadas em vez de uma no seu estado final.

Para evitar diversas chamadas ao banco de dados, você pode deixar todas as suas atualizações para o final. Para tanto, você precisa manter registro de todos os objetos que alterou. Você pode usar variáveis, no seu código, para isso, mas logo elas se tornam não-gerenciáveis, já que você tem mais do que apenas algumas. As variáveis muitas vezes trabalham bem com um *Roteiro de Transação* (120), mas elas podem ficar muito difíceis com um *Modelo de Domínio* (126).

Em vez de guardar objetos em variáveis, você pode dar a cada objeto uma *flag*, indicando-o como sujo, que é configurada quando o objeto é alterado. Então, você precisa encontrar todos os objetos sujos no final da sua transação e gravá-los. O valor desta técnica depende de quão fácil é encontrar os objetos sujos. Se todos eles estiverem em uma única hierarquia, então você pode percorrer a hierarquia e gravar qualquer um que tenha sido alterado. Entretanto, uma rede de objetos mais geral, tal como um *Modelo de Domínio* (126), é mais difícil de percorrer.

A grande força da *Unidade de Trabalho* é que ela mantém toda esta informação em um único lugar. Assim que você a tiver trabalhando para si, não terá mais que se lembrar de fazer muita coisa para manter o registro das suas alterações. Além disso, a *Unidade de Trabalho* é uma plataforma sólida para situações mais complicadas, como lidar com transações de negócio que se estendem por diversas transações de sistema usando *Bloqueio Offline Otimista* (392) e *Bloqueio Offline Pessimista* (401).

Exemplo: *Unidade de Trabalho* com Registro de Objeto (Java)

por David Rice

Aqui está uma *Unidade de Trabalho* que pode manter registro de todas as alterações em uma dada transação de negócio e então perpetrá-las (*commit*) no banco de dados quando instruída a fazê-lo. Nossa camada de domínio tem uma *Camada Supertipo* (444), `ObjetoDoDomínio`, com a qual a *Unidade de Trabalho* irá interagir. Para armazenar o conjunto de alterações usamos três listas: objetos do domínio novos, sujos e excluídos.

```
class UnidadeDeTrabalho...

    private List objetosNovos = new ArrayList( );
    private List objetosSujos = new ArrayList( );
    private List objetosExcluídos = new ArrayList( );
```

Os métodos de registro mantêm o estado destas listas. Eles devem executar asserções básicas tais como assegurar que um ID não seja nulo ou que um objeto sujo não esteja sendo registrado como um novo.

```
class UnidadeDeTrabalho...

    public void registrarNovo (ObjetoDoDomínio obj) {
        Assert.notNull("id não nulo", obj.lerId( ));
        Assert.isTrue("objeto não sujo", !objetosSujos.contains(obj));
        Assert.isTrue("objeto não excluído", !objetosExcluídos.contains(obj));
        Assert.isTrue("objeto ainda não registrado como novo", !objetosNovos.contains(obj));
```

```
        objetosNovos.add(obj);
    }
    public void registrarSujo (ObjetoDoDomínio obj) {
        Assert.notNull("id não nulo", obj.lerId( ));
        Assert.isTrue("objeto não excluído", !objetosExcluídos.contains(obj));
        if (!objetosSujos.contains(obj) && !objetosNovos.contains(obj)) {
            objetosSujos.add(obj);
        }
    }
    public void registrarRemovido (ObjetoDoDomínio obj) {
        Assert.notNull("id não nulo", obj.lerId( ));
        if (objetosNovos.remove(obj)) return;
        objetosSujos.remove(obj);
        if (!objetosRemovidos.contains(obj) {
            objetosRemovidos.add(obj);
        }
    }
    public void registrarLimpo (ObjetoDoDomínio obj) {
        Assert.notNull("id não nulo", obj.lerId( ));
    }
```

Perceba que `registrarLimpo( )` não faz nada aqui. Uma prática comum é colocar um *Mapa de Identidade* (196) dentro de uma *Unidade de Trabalho*. Um *Mapa de Identidade* (196) é necessário quase sempre que você armazenar o estado de objetos do domínio na memória porque múltiplas cópias do mesmo objeto resultariam em comportamento indefinido. Estivesse um *Mapa de Identidade* (196) no lugar, `registrarLimpo( )` colocaria o objeto registrado nele. Da mesma forma, `registrarNovo( )` colocaria o novo objeto no mapa e `registrarExcluído( )` removeria um objeto excluído do mapa. Sem o *Mapa de Identidade* (196), você tem a opção de não incluir `registrarLimpo( )` na sua *Unidade de Trabalho*. Já vi implementações deste método que removem objetos alterados da lista de sujos, mas desfazer parcialmente alterações é sempre complicado. Tenha cuidado ao reverter qualquer estado no conjunto de alterações.

`commit( )` localizará o *Mapeador de Dados* (170) para cada objeto e chamará o método de mapeamento apropriado. Os métodos `atualizarSujos( )` e `apagarRemovidos( )` não são mostrados, mas eles se comportariam como `inserirNovos( )`, o que é esperado.

```
class UnidadeDeTrabalho...

    public void commit( ) {
        inserirNovo( );
        atualizarSujos( );
        apagarRemovidos( );
    }
    private void inserirNovos( ) {
        for (Iterator objetos = objetosNovos.iterator( ); objetos.hasNext( );) {
            ObjetoDoDomínio obj = (ObjetoDoDomínio) objetos.next( );
            RegistroDoMapeador.lerMapeador(obj.getClass( )).insert(obj);
        }
    }
```

Não está incluído nesta *Unidade de Trabalho* o rastreamento de quaisquer objetos que tenhamos lido e queiramos verificar erros de leitura inconsistente na hora da confirmação. Isso é abordado pelo *Bloqueio Offline Otimista* (392).

A seguir, precisamos facilitar o registro de objetos. Primeiro cada objeto do domínio precisa encontrar a *Unidade de Trabalho* servindo a transação de negócio corrente. Já que essa *Unidade de Trabalho* será necessária para todo o modelo de domínio passá-la como um parâmetro, é provavelmente pouco razoável. Já que cada transação de negócio é executada dentro de uma única *thread*, podemos associar a *Unidade de Trabalho* com a *thread* correntemente em execução usando a classe `java.lang.ThreadLocal`. Mantendo as coisas simples, adicionaremos esta funcionalidade usando métodos estáticos na nossa classe *Unidade de Trabalho*. Se já tivermos algum tipo de objeto de sessão associado à *thread* da execução da transação de negócio, devemos colocar a *Unidade de Trabalho* corrente nesse objeto de sessão, em vez de adicionar o *overhead* de gerenciamento do mapeamento de outra *thread*. Além disso, a *Unidade de Trabalho* pertence logicamente à sessão.

```
class UnidadeDeTrabalho...

    private static ThreadLocal corrente = new ThreadLocal( );
    public static void novoCorrente ( ) {
        gravarCorrente(new UnidadeDeTrabalho( ));
    }
    public static void gravarCorrente (UnidadeDeTrabalho udt) {
        corrente.set(udt);
    }
    public static UnidadeDeTrabalho lerCorrente ( ) {
        return (UnidadeDeTrabalho) corrente.get( );
    }
```

Agora podemos dar ao nosso objeto de domínio abstrato os métodos de marcação para registrar a si próprio na *UnidadeDeTrabalho* corrente.

```
class ObjetoDoDomínio...

    protected void marcarNovo( ) {
        UnidadeDeTrabalho.lerCorrente( ).registrarNovo(this);
    }
    protected void marcarLimpo( ) {
        UnidadeDeTrabalho.lerCorrente( ).registrarLimpo(this);
    }
    protected void marcarSujo( ) {
        UnidadeDeTrabalho.lerCorrente( ).registrarSujo(this);
    }
    protected void marcarExcluído ( ) {
        UnidadeDeTrabalho.lerCorrente( ).registrarExcluído(this);
    }
```

Os objetos de domínio concretos precisam se lembrar de marcar a si mesmos como novos e sujos quando apropriado.

```
class Álbum...
    public static Álbum criar (String nome) {
        Álbum obj = new Álbum(geradorID.novoID( ), nome);
        obj.marcarNovo( );
        return obj;
    }
```

```
    public void gravarTítulo (String título) {
        this.título = título;
        marcarSujo( );
    }
```

Não está mostrado que o registro de objetos removidos pode ser feito por um método `remover( )` no objeto de domínio abstrato. Além disso, se você tiver implementado `registrarLimpo( )`, seus *Mapeadores de Dados* (170) precisarão registrar qualquer objeto recém-carregado como limpo.

A parte final é registrar e perpetrar (*commit*) a *Unidade de Trabalho* onde for apropriado. Isso pode ser feito explícita ou implicitamente. Aqui está como o gerenciamento explícito da *Unidade de Trabalho* se parece:

```
class RoteiroDeEdiçãoDeÁlbum...

    public static void atualizarTítulo (Long IdDoÁlbum, String título) {
        UnidadeDeTrabalho.novoCorrente( );
        Mapeador mapeador = RegistroDoMapeador.lerMapeador(Álbum.class);
        Álbum álbum = (Álbum) mapeador.buscar(IdDoÁlbum);
        álbum.gravarTítulo(título);
        UnidadeDeTrabalho.lerCorrente( ).commit( );
    }
```

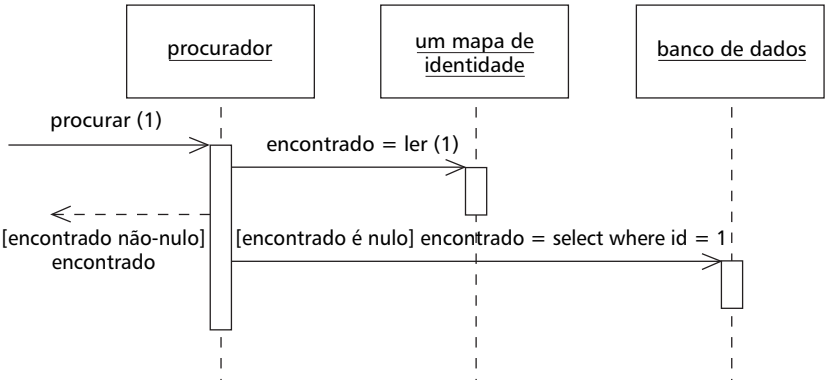
Excetuando-se as aplicações muito simples, o gerenciamento implícito da *Unidade de Trabalho* é mais apropriado, pois evita a codificação repetitiva e tediosa. Aqui está um *servlet Camada Supertipo* (444) que registra e grava (*commit*) a *Unidade de Trabalho* para os seus subtipos concretos. Os subtipos implementarão `tratarLeitura( )` em vez de sobrescreverem `executarLeitura( )`. Qualquer código em execução dentro de `tratarLeitura( )` terá uma *Unidade de Trabalho* com a qual trabalhar.

```
class ServletUnidadeDeTrabalho...

    final protected void executarLeitura (HttpServletRequest solicitação, HttpServletResponse resposta)
            throws ServletException, IOException {
        try{
            UnidadeDeTrabalho.novoCorrente( );
            tratarLeitura(solicitação, resposta);
            UnidadeDeTrabalho.lerCorrente( ).commit( );
        } finally {
            UnidadeDeTrabalho.gravarCorrente(null);
        }
    }
abstract void tratarLeitura (HttpServletRequest solicitação, HttpServletResponse resposta)
    throws ServletException, IOException;
```

O exemplo de *servlet* acima é obviamente um pouco simplista, pois omite o controle de transação de sistema. Se você estivesse usando um *Controlador Frontal* (328), você mais provavelmente encapsularia a gerência da *Unidade de Trabalho* nos seus comandos, em vez de `executarLeitura( )`. Um encapsulamento similar pode ser feito com quase qualquer contexto de execução.

Mapa de Identidade (Identity Map)

Assegura que cada objeto seja carregado apenas uma vez, mantendo cada objeto carregado em um mapa. Procura objetos usando o mapa quando se referindo a eles.



Um provérbio antigo diz que um homem com dois relógios nunca sabe que horas são. Se dois relógios são confusos, você pode entrar em uma confusão ainda maior ao carregar objetos de um banco de dados. Se você não for cuidadoso poderá carregar os dados do mesmo registro do banco de dados em dois objetos diferentes. Então, quando você atualizar ambos os objetos, passará por momentos interessantes gravando as alterações no banco de dados corretamente.

Relacionado a isso está um problema óbvio de desempenho. Se você carregar os mesmos dados mais de uma vez, estará incorrendo em um custo alto em chamadas remotas. Assim, não carregar os mesmos dados duas vezes não apenas ajuda na consistência, mas pode também aumentar a velocidade da sua aplicação.

Um *Mapa de Identidade* mantém um registro de todos os objetos que foram lidos do banco de dados em uma única transação de negócio. Sempre que quiser um objeto, você verifica o *Mapa de Identidade* primeiro para ver se já o tem.

Como Funciona

A idéia básica por trás do *Mapa de Identidade* é ter uma série de mapas contendo objetos que foram trazidos do banco de dados. Em um caso simples, com um esquema isomórfico, você terá um mapa por tabela do banco de dados. Quando você carregar um objeto do banco de dados, primeiro verifica o mapa. Se houver um objeto nele que corresponda ao que você estiver carregando, retorne-o. Caso contrário, você vai ao banco de dados, colocando os objetos no mapa para referência futura quando carregá-los.

Há um número de escolhas na implementação com as quais se preocupar. Além disso, já que *Mapas de Identidade* interagem com gerenciamento de concorrência, você deveria considerar também o *Bloqueio Offline Otimista* (392).

Escolha das Chaves A primeira coisa a considerar é a chave do mapa. A escolha óbvia é a chave primária da tabela correspondente do banco de dados. Isso funciona

bem se a chave for uma coluna única e imutável. Uma chave primária substituta se adapta bem nesta abordagem porque você pode usá-la como a chave no mapa. A chave será normalmente um tipo de dados simples de modo que o comportamento de comparação funcionará bem.

Explícito ou Genérico Você tem que escolher entre tornar um *Mapa de Identidade* explícito ou genérico. Um *Mapa de Identidade* explícito é acessado com métodos distintos para cada tipo de objeto que você precisar: tal como `encontrarPessoa(1)`. Um mapa genérico usa um único método para todos os tipos de objetos, com talvez um parâmetro para indicar qual o tipo do objeto que você precisa, tal como em `encontrar("Pessoa", 1)`. A vantagem óbvia é que você pode suportar um mapa genérico com um objeto genérico e reutilizável. É fácil criar um *Registro* (448) reutilizável que funcione para todos os tipos de objetos e não precise de atualização quando você adicionar um novo mapa.

Contudo, prefiro um *Mapa de Identidade* explícito. Para começar, isso lhe dá verificação em tempo de compilação em uma linguagem fortemente tipada, mas, mais do que isso, tem todas as outras vantagens de uma interface explícita: é mais fácil ver quais mapas estão disponíveis e como eles se chamam. Isso significa acrescentar um método cada vez que você acrescentar um novo mapa, mas é um *overhead* pequeno em razão da clareza.

O tipo da sua chave afeta a escolha. Você só pode usar um mapa genérico se todos os seus objetos tiverem o mesmo tipo de chave. Este é um bom argumento para encapsular diferentes tipos de chaves de bancos de dados por trás de um único objeto chave. (Veja mais detalhes em *Campo Identidade* (215).)

Quantos Aqui a decisão varia entre um mapa por classe e um mapa para a sessão inteira. Um único mapa para a sessão funciona apenas se você tiver chaves únicas de banco de dados (veja a discussão em *Campo Identidade* (215) sobre os compromissos envolvidos.) Uma vez que você tenha um *Mapa de Identidade*, o benefício é que você tem apenas um lugar para ir e nenhuma decisão complicada sobre herança.

Se você tiver diversos mapas, o caminho óbvio é um mapa por classe ou por tabela, o que funciona bem se o esquema do seu banco de dados e modelos de objetos forem o mesmo. Se eles forem diferentes, é geralmente mais fácil basear os mapas nos seus objetos em vez de nas suas tabelas, já que os objetos não devem conhecer as complexidades do mapeamento.

A herança aparece aqui como um sério complicador. Se você tiver carros como subtipo de veículos, você tem um mapa ou mapas separados? Mantê-los separados pode tornar referências polimórficas muito mais complicadas, já que qualquer busca precisa saber procurar em todos os mapas. Como conseqüência, prefiro usar um único mapa para cada árvore de herança, mas isso significa que você também deve tornar suas chaves únicas pelas árvores de herança, o que pode ser difícil se você usar *Herança de Tabela Concreta* (283).

Uma vantagem de um único mapa é que você não tem que acrescentar novos quando adiciona tabelas de banco de dados. Todavia, vincular seus mapas aos seus *Mapeadores de Dados* (170) (veja a seguir) não será nenhum fardo extra.

Onde Colocá-los Os *Mapas de Identidade* precisam estar em algum lugar onde sejam fáceis de encontrar. Eles também estão vinculados ao contexto do processo no qual você está trabalhando. Você precisa assegurar que cada sessão tenha sua própria instância que seja isolada de qualquer outra instância de sessão. Assim, você precisa co-

locar o *Mapa de Identidade* em um objeto específico de sessão. Se você estiver usando *Unidade de Trabalho* (187), este é de longe o melhor local para os *Mapas de Identidade* já que a *Unidade de Trabalho* (187) é o principal lugar para manter registro dos dados vindos do banco de dados ou indo para o banco de dados. Se você não tiver uma *Unidade de Trabalho* (187), a melhor aposta é um *Registro* (448) que seja vinculado à sessão.

Como sugeri aqui, você normalmente vê um único *Mapa de Identidade* para uma sessão, caso contrário precisará fornecer proteção transacional para o seu mapa, o que dá mais trabalho do que qualquer desenvolvedor sensato quer executar. Todavia, há algumas exceções. A maior de todas é usar um banco de dados orientado a objetos como um cache transacional, mesmo se você usar um banco de dados relacional para armazenar os dados. Embora eu não tenha visto qualquer estudo independente sobre performance, as chances são de que vale a pena dar uma olhada. Muitas pessoas que respeito são grandes fãs de cache transacional como uma forma de melhorar o desempenho.

A outra exceção são objetos que sejam apenas de leitura em todas as situações. Se um objeto não puder nunca ser modificado, não há necessidade de se preocupar com ele sendo compartilhado pelas sessões. Em sistemas de desempenho intensivo, pode ser muito benéfico carregar todos os dados apenas de leitura uma única vez e mantê-los disponíveis para o processo inteiro. Neste caso, você tem seus *Mapas de Identidade* apenas de leitura em um contexto de processo e seus *Mapas de Identidade* atualizáveis em um contexto de sessão. Isso também se aplica a objetos que não sejam completamente apenas de leitura, mas que sejam atualizados tão raramente que você não se importe de limpar o *Mapa de Identidade* do processo e potencialmente voltar ao servidor quando isso acontecer.

Mesmo se você estiver inclinado a ter apenas um *Mapa de Identidade*, você pode dividi-lo em dois: linhas apenas para leitura e linhas atualizáveis. Você pode evitar que os clientes saibam qual é qual fornecendo uma interface que verifique ambos os mapas.

Quando Usá-lo

Geralmente você usa um *Mapa de Identidade* para gerenciar qualquer objeto trazido de um banco de dados e modificado. A razão principal é que você não deseja ter uma situação na qual dois objetos na memória correspondam a um único registro no banco de dados – você poderia modificar os dois registros inconsistentemente e assim confundir o mapeamento do banco de dados.

Outra importância do *Mapa de Identidade* é que ele atua como um cache para as leituras do banco de dados, o que significa que você pode evitar ir ao banco de dados cada vez que precisar de algum dado.

Você pode não precisar de um *Mapa de Identidade* para objetos imutáveis. Se você não pode alterar um objeto, então não tem que se preocupar com anomalias causadas por modificações. Já que *Objetos Valor* (453) são imutáveis, resulta que você não precisa de *Mapas de Identidade* para eles. Ainda assim, *Mapas de Identidade* apresentam vantagens aqui, a mais importante das quais é a de desempenho do cache. Outra é que eles ajudam a prevenir o uso das formas errôneas de testes de igualdade, um problema importante em Java, onde você não pode sobrescrever o operador ==.

Você não precisa de um *Mapa de Identidade* para um *Mapeamento Dependente* (256). Já que objetos dependentes têm sua persistência controlada pelos pais, não há necessidade de um mapa para armazenar identidade. Contudo, embora você não

precise de um mapa, talvez queira fornecer um se precisar acessar o objeto por meio de uma chave de banco de dados. Neste caso, o mapa é meramente um índice, de modo que é discutível se ele realmente conta como um mapa.

O *Mapa de Identidade* ajuda a evitar conflitos de atualização dentro de uma mesma sessão, mas não faz nada para lidar com conflitos que atravessem sessões. Este é um problema complexo que discutimos mais no *Bloqueio Offline Otimista* (392) e no *Bloqueio Offline Pessimista* (401).

Exemplo: Métodos para um *Mapa de Identidade* (Java)

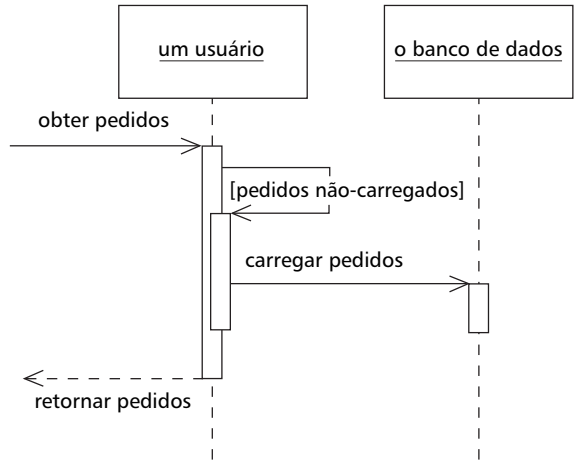
Para cada Mapa de Identidade, temos um campo mapa e métodos de acesso.

```
private Map pessoas = new HashMap( );
public static void adicionarPessoa (Pessoa parâmetro) {
    instânciaÚnica.pessoas.put(parâmetro.lerID( ), parâmetro);
}
public static Pessoa lerPessoa (Long chave) {
    return (Pessoa) instânciaÚnica.pessoas.get(chave);
}
public static Pessoa lerPessoa (long chave) {
    return lerPessoa(new Long(chave));
}
```

Um dos aborrecimentos de Java é o fato de que `long` não é um objeto, de modo que você não pode usá-lo como índice para um mapa. Isso não é tão aborrecedor quanto poderia ter sido, já que não fazemos nenhuma operação aritmética no índice. A única situação em que isso realmente atrapalha é quando você quer resgatar um objeto com um literal. Você quase nunca precisa fazer isso em código de produção, mas muitas vezes o faz no código de teste, de forma que incluí um método de leitura que recebe um `long` para tornar o teste mais fácil.

Carga Tardia (Lazy Load)

Um objeto que não contém todos os dados de que você precisa, mas sabe como obtê-los.



Para carregar dados de um banco de dados para a memória, é conveniente projetar as coisas de modo que quando você carregar um objeto de interesse também carregue os objetos que sejam relacionados a ele. Isso torna a carga mais fácil para o desenvolvedor que estiver usando o objeto, que de outra forma tem que carregar explicitamente todos os objetos de que precisa.

Todavia, se você levar isso ao seu final, chegará ao ponto em que carregar um objeto pode ter o mesmo efeito de carregar um grande número de objetos relacionados – algo que prejudica o desempenho quando apenas alguns dos objetos são realmente necessários.

Uma *Carga Tardia* interrompe este processo de carga por um tempo, deixando um marcador na estrutura do objeto de modo que se os dados forem necessários podem ser carregados apenas quando forem usados. Como muitas pessoas sabem, se você deixar as coisas para mais tarde vai se dar bem quando descobrir que não precisava realmente fazê-las.

Como Funciona

Há quatro maneiras principais pelas quais você pode implementar a *Carga Tardia*: inicialização tardia, *proxy* virtual, armazenador de valor e fantasma.

Inicialização tardia [Padrões Beck] é a abordagem mais simples. A idéia básica é que cada acesso ao campo verifique primeiro para ver se ele é nulo. Se for, ele calcula o valor do campo antes de retornar esse mesmo campo. Para fazer isso funcionar, você tem que assegurar que o campo seja auto-encapsulado, o que quer dizer que todo acesso ao campo, mesmo de dentro da classe, é feito por meio de um método de leitura.

Usar um nulo para sinalizar um campo que não foi carregado ainda funciona bem, a menos que nulo seja um valor legal de campo. Neste caso, você precisa de al-

go a mais para sinalizar que esse campo não foi carregado ou usar um *Caso Especial* (462) para o valor nulo.

Usar inicialização tardia é simples, mas tende a forçar uma dependência entre o objeto e o banco de dados. Por esta razão, ela funciona melhor com *Registro Ativo* (165), *Gateway de Tabelas de Dados* (151) e *Gateway de Linhas de Dados* (158). Se você estiver usando um *Mapeador de Dados* (170), precisará de uma camada adicional de indireção, o que você pode obter usando um **proxy virtual** [Gang of Four]. Um *proxy* virtual é um objeto que parece com o objeto que deveria estar no campo, mas, na verdade não contém nada. Apenas quando um dos seus métodos é chamado, ele carrega o objeto correto a partir do banco de dados.

O bom do *proxy* virtual é que ele parece exatamente com o objeto que deve estar lá. O ruim é que ele não é esse objeto, então você pode facilmente se deparar com um desagradável problema de identidade. Além disso, você pode ter mais de um *proxy* virtual para o mesmo objeto real. Todos esses *proxies* terão diferentes identidades de objeto, mas ainda assim representam o mesmo objeto conceitual. Você tem que, no mínimo, sobrescrever o método de igualdade e lembrar-se de usá-lo em vez de um método identidade. Sem isso, e disciplina, você irá se deparar com algumas falhas difíceis de rastrear.

Em alguns ambientes, outro problema é que você acaba tendo que criar muitos *proxies* virtuais, um para cada classe na qual estiver usando *proxy.* Você pode normalmente evitar isso em linguagens tipadas dinamicamente, mas em linguagens tipadas estaticamente, as coisas muitas vezes ficam confusas. Mesmo quando a plataforma fornece recursos convenientes, como os *proxies* de Java, outros obstáculos podem aparecer.

Esses problemas não lhe atingem se você apenas usar *proxies* virtuais para classes coleções, tais como listas. Já que coleções são *Objetos Valor* (453), suas identidades não importam. Adicionalmente, você tem apenas umas poucas classes coleção para escrever coleções virtuais.

Com classes do domínio, você pode contornar esses problemas usando um **armazenador de valor**. Este conceito, com o qual me deparei pela primeira vez em Smalltalk, é um objeto que encapsula algum outro objeto. Para obter o objeto subjacente, você solicita seu valor ao armazenador de valor, mas apenas no primeiro acesso ele pega os dados do banco de dados. As desvantagens do armazenador de valor são que a classe precisa saber que ele existe e que você perde a característica explícita da forte verificação de tipos. Você pode evitar problemas de identidade assegurando que o armazenador de valor nunca seja passado para além da classe a qual ele pertence.

Um **fantasma** é o objeto real em um estado parcial. Quando você carrega o objeto do banco de dados, ele contém apenas seu ID. Sempre que você tentar acessar um campo, ele carrega seu estado completo. Pense em um fantasma como um objeto, onde cada campo é inicializado tardiamente de uma só vez, ou como um *proxy* virtual, onde o objeto é seu próprio *proxy* virtual. É claro que não há necessidade de carregar todos os dados de uma vez só. Você pode reuni-los em grupos que sejam comumente usados juntos. Se você usar um fantasma, pode colocá-lo imediatamente no seu *Mapa de Identidade* (196). Dessa forma, você mantém a identidade e evita todos os problemas causados por referências cíclicas durante a leitura de dados.

Um *proxy* virtual/fantasma não precisa ser completamente destituído de dados. Se você tiver alguns dados que sejam rápidos de pegar e comumente usados, pode fazer sentido carregá-los quando você carregar o *proxy* ou o fantasma. (Isso é às vezes referido como um "objeto leve.")

A herança muitas vezes coloca um problema com a *Carga Tardia*. Se você for usar fantasmas, precisará saber que tipos de fantasmas criar, o que você muitas vezes não sabe dizer sem carregar a coisa apropriadamente. Os *proxies* virtuais podem sofrer do mesmo problema em linguagens tipadas estaticamente.

Outro perigo com a *Carga Tardia* é que ela pode facilmente causar mais acessos ao banco de dados do que você precisa. Um bom exemplo dessa **onda de cargas** é se você preencher um conjunto com *Cargas Tardias* e então olhar um de cada vez. Isso fará com que você vá ao banco de dados uma vez para cada objeto em vez de lê-los todos de uma só vez. Já vi ondas de cargas prejudicar o desempenho de uma aplicação. Uma maneira de evitar isso é nunca ter uma coleção de *Cargas Tardias*, mas, em vez disso, tornar a própria coleção uma *Carga Tardia* e, quando você carregá-la, carregar todos os conteúdos. A limitação desta tática é quando a coleção é muito grande, como todos os endereços IP do mundo. Estes normalmente não são conectados através de associações no modelo de objetos, de modo que não ocorrem com muita freqüência, mas, quando ocorrem, você precisará de um *Manipulador de Listas de Valores* [Alur *et al.*].

A *Carga Tardia* é uma boa candidata à programação orientada a aspectos. Você pode colocar o comportamento da *Carga Tardia* em um aspecto separado, o que lhe permite alterar a estratégia de carga tardia separadamente assim como liberar os desenvolvedores do domínio de ter que lidar com questões de carga tardia. Já vi também um projeto de pós-processador de *byte codes* Java para implementar a *Carga Tardia* de forma transparente.

Muitas vezes, você irá se deparar com situações nas quais diferentes casos de uso funcionam melhor com diferentes variedades de carga tardia. Alguns precisam de um subconjunto do grafo de objetos, outros precisam de outro subconjunto. Para uma eficiência máxima, você quer carregar o subgrafo correto para o caso de uso correto.

A maneira de lidar com isso é ter objetos de interação com o banco de dados separados para os diferentes casos de uso. Assim, se você usar *Mapeador de Dados* (170) pode ter dois objetos mapeadores de pedidos: um que carrega as linhas dos itens imediatamente e outro que os carrega tardiamente. O código da aplicação escolhe o mapeador apropriado dependendo do caso de uso. Uma variação disso é ter o mesmo objeto carregador básico, mas delegar para um objeto estratégia a decisão sobre o padrão de carga. Isto é um pouco mais sofisticado, mas pode ser uma maneira melhor de fatorar comportamento.

Teoricamente você poderia querer uma faixa de diferentes graus de carga tardia, mas na prática você precisa realmente de apenas dois: uma carga completa e uma suficiente para o propósito de identificação em uma lista. Acrescentar mais, normalmente, adiciona mais complexidade do que vale a pena.

Quando Usá-la

Quando usar a *Carga Tardia* é uma decisão relacionada a quanto você quer trazer do banco de dados quando carrega um objeto e quantas chamadas ao banco de dados isso irá requerer. Normalmente, não tem sentido usar a *Carga Tardia* em um campo que é armazenado na mesma linha do resto do objeto, porque na maioria das vezes não custa mais trazer dados adicionais em uma chamada, mesmo se o campo for muito grande – como um *LOB Serializado* (264). Isso quer dizer que normalmente só vale a pena considerar *Carga Tardia* se o campo requerer uma chamada de banco de dados extra para ser acessado.

Em termos de desempenho, trata-se de decidir quando você quer pagar o preço de trazer de volta os dados. Muitas vezes, é uma boa idéia trazer tudo de que você irá precisar em uma única chamada de forma que você tenha logo tudo no lugar, especialmente se isso corresponder a uma única interação com uma interface gráfica com o usuário. O melhor momento para usar a *Carga Tardia* é quando ela envolver uma chamada extra, e os dados que você estiver chamando não forem usados quando o objeto principal for usado.

Acrescentar a *Carga Tardia* aumenta um pouco a complexidade do programa, por isso minha preferência é não usá-la a menos que efetivamente ache que precisarei fazê-lo.

Exemplo: Inicialização Tardia (Java)

A essência da inicialização tardia é um código como este:

```
class Fornecedor...

    public List lerProdutos( ) {
        if (produtos == null) produtos = Produto.buscarPorFornecedor (lerID( ));
        return produtos;
    }
```

Desta forma, o primeiro acesso ao campo *produtos* faz com que os dados sejam carregados do banco de dados.

Exemplo: *Proxy* Virtual (Java)

A chave para o *proxy* virtual é fornecer uma classe que se pareça com a classe real que você usa normalmente mas que na verdade mantenha um simples envoltório em torno dessa classe real. Assim, a lista de produtos de um fornecedor seria armazenada em um campo de lista comum.

```
class LVFornecedor...

    private Lista produtos;
```

O mais complicado na criação de um *proxy* de lista como este é configurá-lo de modo que você possa fornecer uma lista subjacente que seja criada apenas quando for acessada. Para fazer isso, precisamos passar o código necessário para criar a lista na lista virtual quando esta for instanciada. A melhor maneira de fazer isso em Java é definir uma interface para o comportamento de carga.

```
public interface CarregadorDeListaVirtual {
    List carregar( );
}
```

Podemos então instanciar a lista virtual com um carregador que chama o método de mapeamento apropriado.

```
class MapeadorDeFornecedor...

    public static class CarregadorDeProduto implements CarregadorDeListaVirtual {
        private Long id;
```

```
        public CarregadorDeProduto (Long id) {
            this.id = id;
        }
        public List carregar ( ) {
            return MapeadorDeProduto.criar( ).buscarPorFornecedor(id);
        }
    }
```

Durante o método de carga, atribuímos o carregador de produto ao campo lista.


```
class MapeadorDeFornecedor...

    protected ObjetoDoDomínio fazerCarga (Long id, ResultSet rs) throws SQLException {
        String parâmetroNome = rs.getString(2);
        LVFornecedor resultado = new LVFornecedor (id, parâmetroNome);
        resultado.gravarProdutos (new ListaVirtual (new CarregadorDeProduto(id)));
        return resultado;
    }
```

A lista fonte da lista virtual é auto-encapsulada e avalia o carregador na primeira referência:

```
class ListaVirtual...

    private List fonte;
    private CarregadorDeListaVirtual carregador;
    public ListaVirtual (CarregadorDeListaVirtual carregador) {
        this.carregador = carregador;
    }
    private List lerFonte ( ) {
        if (fonte == null) fonte = carregador.carregar( );
        return fonte;
    }
```

Os métodos comuns da lista para os quais delegar são implementados na lista fonte.

```
class ListaVirtual...

    public int size ( ) {
        return lerFonte( ).size( );
    }
    public boolean isEmpty ( ) {
        return lerFonte( ).isEmpty( );
    }
    // ... e assim por diante para o resto dos métodos de lista
```

Dessa maneira, a classe do domínio não sabe nada a respeito de como a classe mapeadora executa a *Carga Tardia*. Na verdade, a classe do domínio não sabe nem que há uma *Carga Tardia*.

Exemplo: Usando um Armazenador de Valor (Java)

Um armazenador de valor pode ser usado como uma *Carga Tardia* genérica. Neste caso o tipo do domínio está ciente de que algo está acontecendo, já que o campo produto é tipado como um armazenador de valor. Esse fato pode ser escondido dos clientes do fornecedor pelo método de leitura.

```
class AVFornecedor...

    private ArmazenadorDeValor produtos;
    public List lerProdutos ( ) {
        return (List) produtos.lerValor( );
    }
```

O próprio armazenador de valor executa o comportamento da *Carga Tardia*. Ele precisa receber o código necessário para carregar seu valor quando acessado. Podemos fazer isso definindo uma interface carregadora.

```
class ArmazenadorDeValor...

    private Object valor;
    private CarregadorDeValor carregador;
    public ArmazenadorDeValor (CarregadorDeValor carregador) {
        this.carregador = carregador;
    }
    public Object lerValor ( ) {
        if (valor == null) valor = carregador.carregar( );
        return valor;
    }
public interface CarregadorDeValor {
    Object carregar( );
}
```

Um mapeador pode configurar o armazenador de valor criando uma implementação do carregador e colocando-a no objeto fornecedor.

```
class MapeadorDeFornecedor...

    protected ObjetoDoDomínio fazerCarga (Long id, ResultSet rs) throws SQLException {
        String parâmetroNome = rs.getString(2);
        AVFornecedor resultado = new AVFornecedor (id, parâmetroNome);
        resultado.gravarProdutos (new ArmazenadorDeValor (new CarregadorDeProduto(id)));
        return resultado;
    }
    public static class CarregadorDeProduto implements CarregadorDeValor {
        private Long id;
        public CarregadorDeProduto (Long id);
            this.id = id;
        }
        public Object carregar ( ) {
            return MapeadorDeProduto.criar( ).buscarPorFornecedor(id);
        }
    }
```

Exemplo: Usando Fantasmas (C#)

Muito da lógica para tornar objetos fantasmas pode ser criada em *Camadas Supertipo* (444). Como conseqüência, se você usar fantasmas, tende a vê-los usados em todo lugar. Começarei nossa exploração de fantasmas olhando alguns objetos do domínio *Camada Supertipo* (444). Cada objeto do domínio sabe se é um fantasma ou não.

```
class ObjetoDoDomínio...

    CarregarEstado Estado;
    public ObjetoDoDomínio (long chave) {
        this.Chave = chave;
    }
    public Boolean ÉFantasma {
        get {return Estado == CarregarEstado.FANTASMA);}
    }
    public Boolean EstáCarregado {
        get {return Estado == CarregarStatus.CARREGADO);}
    }
    public void MarcarCarregando ( ) {
        Debug.Assert(ÉFantasma);
        Estado = CarregarEstado.CARREGANDO;
    }
    public void MarcarCarregado ( ) {
        Debug.Assert(Estado == CarregarEstado.CARREGANDO);
        Estado = CarregarEstado.CARREGADO;
    }
  enum CarregarEstado {FANTASMA, CARREGANDO, CARREGADO};
```

Os objetos do domínio podem estar em três estados: fantasma, carregando e carregados. Gosto de encapsular a informação do estado com propriedades apenas de leitura e métodos explícitos de alteração de estado.

O elemento mais intrusivo dos fantasmas é que cada método de acesso precisa ser modificado de modo que provoque uma carga se o objeto realmente for um fantasma.

```
class Empregado...

    public String Nome {
        get {
            Carregar( );
            return _nome;
        }
        set {
            Carregar( );
            _nome = valor;
        }
    }
    String _nome;

class ObjetoDoDomínio...

    protected void Carregar( ) {
        if (ÉFantasma)
            FonteDeDados.Carregar(this);
    }
```

Essa necessidade é um alvo ideal para programação orientada a aspectos para pós-processamento de *byte code*.

Para que a carga funcione, o objeto do domínio precisa chamar o mapeador correto. Entretanto, minhas regras de visibilidade dizem que o código do domínio pode não ver o código do mapeador. Para evitar a dependência, preciso usar uma combinação interessante de *Registro* (448) e *Interface Separada* (445) (Figura 11.4). Defino um *Registro* (448) para o domínio para operações na fonte de dados.

```
class FonteDeDados...

    public static void Carregar (ObjetoDoDomínio obj) {
        instance.Carregar(obj);
    }
```

A instância da fonte de dados é definida usando uma interface.

```
class FonteDeDados...

    public interface IFonteDeDados {
        void Carregar (ObjetoDoDomínio obj);
    }
```

O registro de mapeadores, definido na camada de fonte de dados, implementa a interface da fonte de dados. Neste caso, coloquei os mapeadores em um dicionário indexado pelo tipo de domínio. O método de carga encontra o mapeador correto e lhe manda carregar o objeto do domínio apropriado.

```
class RegistroMapeador: IFonteDeDados...

    public void Carregar (ObjetoDoDomínio obj) {
        Mapeador(obj.GetType( )).Carregar(obj);
    }
    public static Mapeador Mapeador(Type tipo) {
        return (Mapeador) instance.mapeadores[tipo];
    }
    IDictionary mapeadores = new Hashtable( );
```

O código anterior mostra como os objetos do domínio interagem com a fonte de dados. A lógica da fonte de dados usa *Mapeadores de Dados* (170). A lógica de atualização nos mapeadores é a mesma do caso sem fantasmas – o comportamento interessante para este exemplo reside no comportamento de busca e de carga.

Classes mapeadoras concretas têm seus próprios métodos de busca que usam um método abstrato e realizam *downcast* no resultado.

```
class MapeadorDeEmpregado...

    public Empregado Buscar (long chave) {
        return (Empregado) BuscaAbstrata (chave);
    }

class Mapeador...

    public ObjetoDoDomínio BuscaAbstrata (long chave) {
        ObjetoDoDomínio resultado;
        resultado = (ObjetoDoDomínio) mapaCarregado[chave];
```

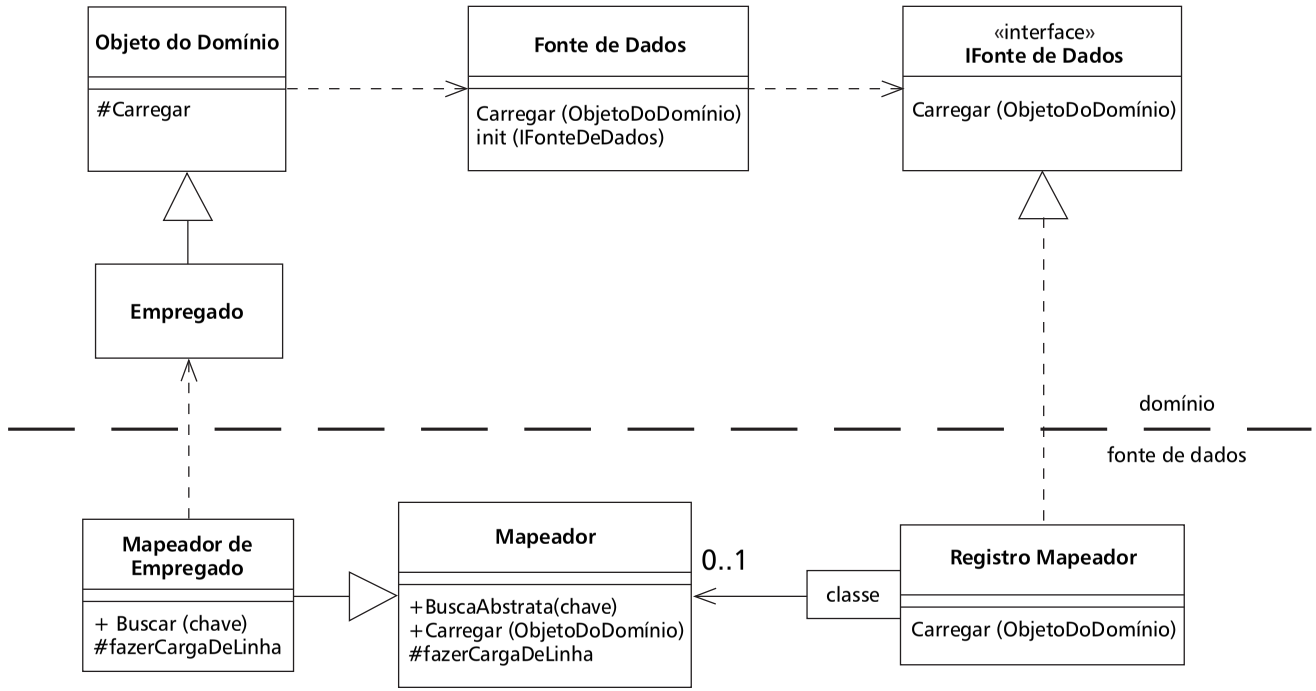


Figura 11.4 Classes envolvidas na carga de um fantasma.

```
        if (resultado == null) {
            resultado = CriarFantasma(chave);
            mapaCarregado.Add(chave, resultado);
        }
        return resultado;
    }
    IDictionary mapaCarregado = new Hashtable( );
    public abstract ObjetoDoDomínio CriarFantasma(long chave);

class MapeadorDeEmpregado...

    public override ObjetoDoDomínio CriarFantasma(long chave) {
        return new Empregado(chave);
    }
```


Como você pode ver, o método de busca retorna um objeto no seu estado de fantasma. Os dados reais não vêm do banco de dados até que a carga seja disparada pelo acesso a uma propriedade no objeto de domínio.

```
class Mapeador...

    public void Carregar (ObjetoDoDomínio obj) {
        if (! obj.ÉFantasma) return;
        IDBCommand comm = new OleDbCommand (declaraçãoDeBusca( ), DB.connection);
        comm.Parameters.Add (new OleDbParameter("chave", obj.Chave));
        IDataReader leitor = comm.ExecuteReader( );
        leitor.Read( );
        CarregarLinha (leitor, obj);
        leitor.Close( );
    }
    protected abstract String declaraçãoDeBusca( );
    public void CarregarLinha (IDataReader leitor, ObjetoDoDomínio obj) {
        if (obj.ÉFantasma) {
            obj.MarcarCarregando( );
            fazerCargaDaLinha (leitor, obj);
            obj.MarcarCarregado( );
        }
    }
    protected abstract void fazerCargaDaLinha (IDataReader leitor, ObjetoDoDomínio obj);
```

Como é comum nestes exemplos, a *Camada Supertipo* (444) lida com todo o comportamento abstrato e então chama um método abstrato para uma subclasse específica executar seu papel. Para este exemplo, usei um leitor de dados, uma abordagem baseada em cursor que é a mais comum para as diversas plataformas no momento. Deixarei para você estender isso para um conjunto de dados, o que seria realmente mais apropriado para a maior parte dos casos em .NET.

Para este objeto Empregado, mostrarei três tipos de propriedades: um nome que é um valor simples, um departamento que é uma referência a um outro objeto e uma lista de registros de horários que mostra o caso de uma coleção. Todos são carregados juntos na implementação da subclasse do método associado.

```
class MapeadorDeEmpregado...

    protected override void fazerCargaDaLinha (IDataReader leitor, ObjetoDoDomínio obj) {
        Empregado empregado = (Empregado) obj;
        empregado.Nome = (String) leitor["nome"];
        MapeadorDeDepartamento mapDep =
            (MapeadorDeDepartamento) RegistroMapeador.Mapeador (typeof (Departamento));
        empregado.Departamento = mapDep.Buscar( (int) leitor["IdDoDepartamento"]);
        carregarRegistrosDeTempo (empregado);
    }
```


O valor do nome é carregado simplesmente lendo a coluna apropriada do cursor corrente do leitor de dados. O departamento é lido usando o método de busca no objeto mapeador de departamento. Isso acabará configurando a propriedade para um fantasma do departamento. Os dados do departamento serão lidos apenas quando o próprio objeto departamento for acessado.

A coleção é o caso mais complicado. Para evitar carregamento em ondas, é importante carregar todos os registros de tempo em uma única busca. Para isso, precisamos de uma implementação de lista especial que atue como uma lista fantasma. Esta lista é apenas um envoltório fino em torno de um objeto lista real, para o qual todo o comportamento real é apenas delegado. A única coisa que o fantasma faz é assegurar que qualquer acesso à lista real provoque uma carga.

```
class ListaDoDomínio...

    IList dados {
        get {
            Carregar( );
            return _dados;
        }
        set { _dados = valor;}
    }
    IList _dados = new ArrayList( );
    public int Count {
        get { return dados.Count;}
    }
```

A classe da lista do domínio é usada pelos objetos do domínio e é parte da camada do domínio. A carga real precisa acessar comandos SQL, então uso delegação para definir uma função de carga que possa ser fornecida pela camada mapeadora.

```
class ListaDoDomínio...

    public void Carregar( ) {
        if (ÉFantasma) {
            MarcarCarregando( );
            ExecutarCarregador(this);
            MarcarCarregado( );
        }
    }
    public delegate void Carregador (ListaDoDomínio lista);
    public Carregador ExecutarCarregador;
```

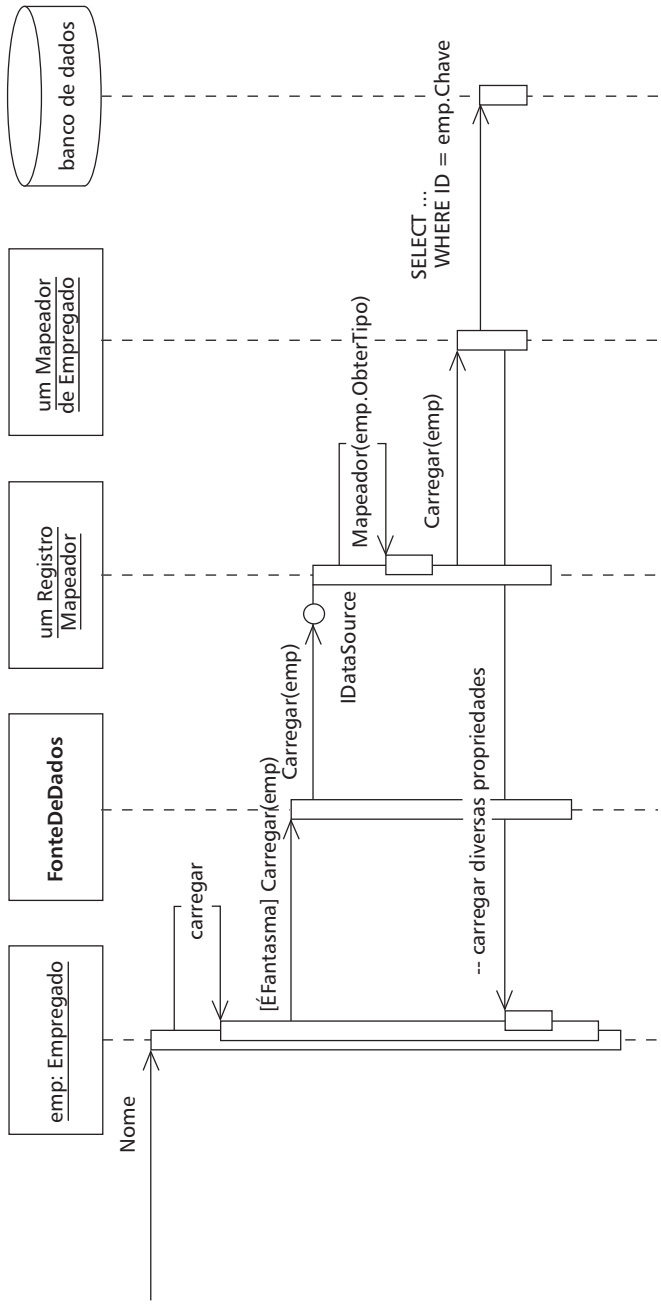


Figura 11.5 A seqüência de carga de um fantasma.

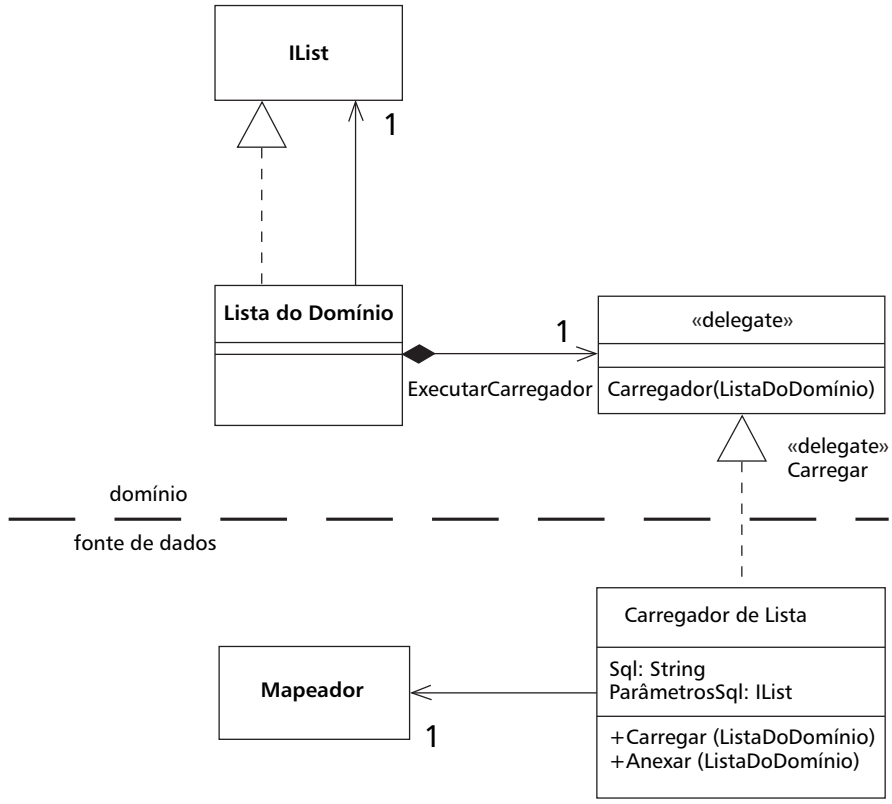


Figura 11.6 Classes para uma lista fantasma. Como ainda não há um padrão aceito para mostrar delegação em modelos UML, esta é minha abordagem corrente.

Pense em delegação como uma variedade especial de *Interface Separada* (445) para uma única função. De fato, declarar uma interface com uma única função é uma alternativa razoável de fazer isso.

O próprio carregador tem propriedades para especificar o SQL para a carga e o mapeador para usar para mapear os registros de tempo. O mapeador de empregados configura o carregador quando carrega o objeto empregado.

```
class MapeadorDeEmpregado...

    void carregarRegistrosDeTempo (Empregado empregado) {
        CarregadorDeLista carregador = new CarregadorDeLista( );
        carregador.Sql = MapeadorDeRegistroDeTempo.SQL_BUSCAR_PELO_EMPREGADO;
        carregador.ParâmetrosSql.Add(empregado.Chave);
        carregador.Mapeador = RegistroMapeador.Mapeador(typeof(RegistroDeTempo));
        carregador.Anexar((ListaDeDomínio) empregado.RegistrosDeTempo);
    }

class CarregadorDeLista...

    public String Sql;
    public IList ParâmetrosSql = new ArrayList( );
    public Mapeador Mapeador;
```

Já que a sintaxe para atribuição de delegação é um pouco complicada, dei ao carregador um método anexar.

```
class CarregadorDeLista...

    public void Anexar (ListaDoDomínio lista) {
        lista.ExecutarCarregador = new ListaDoDomínio.Carregador(Carregar);
    }
```

Quando o empregado é carregado, a coleção de registros de tempo permanece em um estado de fantasma até que um dos métodos de acesso dispare o carregador. Nesse momento, o carregador executa a consulta para preencher a lista.


```
class CarregadorDeLista...

    public void Carregar (ListaDoDomínio lista) {
        lista.EstáCarregada = true;
        IDbCommand comm = new OleDbCommand (Sql, DB.connection);
        foreach (Object param in ParâmetrosSql)
            comm.Parameters.Add(new OleDbParameter(param.ToString( ), param));
        IDataReader leitor = comm.ExecuteReader( );
        while (leitor.Read( )) {
            ObjetoDoDomínio obj = FantasmaParaLinha(leitor);
            Mapeador.CarregarLinha(leitor, obj);
            lista.Adicionar(obj);
        }
        leitor.Close( );
    }
    private ObjetoDoDomínio FantasmaParaLinha (IDataReader leitor) {
        return Mapeador.BuscaAbstrata((System.Int32)leitor[Mapeador.NomeDaColunaChave]);
    }
```

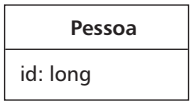
Usar listas de fantasmas como essa é importante para reduzir a onda de cargas. Ela não a elimina completamente, já que há outros casos em que ela aparece. Nesse exemplo, um mapeamento mais sofisticado poderia carregar os dados do departamento em uma única consulta com os empregados. Entretanto, carregar sempre todos os elementos juntos em uma coleção ajuda a eliminar os piores casos.

CAPÍTULO 12

Padrões Estruturais Objeto-Relacionais

Campo Identidade (Identity Field)

Guarda o campo ID de um banco de dados em um objeto para manter a identidade entre um objeto na memória e uma linha do banco de dados.



Os bancos de dados relacionais diferenciam uma linha de outra usando uma chave – em particular, a chave primária. Entretanto, objetos na memória não precisam de tal chave, já que o sistema de objetos, por trás dos panos, assegura a identidade correta (ou, no caso de C++, com posições de memória absolutas). Não há problemas para ler dados de um banco de dados, mas, para gravá-los de volta, você precisa vincular o banco de dados ao sistema de objetos em memória.

Em essência, o *Campo Identidade* é muito simples. Tudo o que você faz é armazenar a chave primária da tabela do banco de dados relacional nos campos dos objetos.

Como Funciona

Embora a noção básica do *Campo Identidade* seja muito simples, há muitas questões complicadas que vêm à tona.

Escolhendo a chave A primeira questão é que tipo de chave escolher no seu banco de dados. É claro que nem sempre esta escolha se apresenta, visto que, muitas vezes, você está lidando com um banco de dados já existente que já tem suas estruturas de chaves estabelecidas. Há muita discussão e material sobre isso na comunidade de banco de dados. Ainda assim, o mapeamento para objetos adiciona algumas preocupações à sua decisão.

A primeira preocupação é se devem ser usadas chaves com ou sem significado. Uma **chave com significado** é como o número do seguro social nos Estados Unidos para identificar uma pessoa. Uma **chave sem significado** é basicamente um número randômico que o banco de dados inventa e que não se destina ao uso de seres humanos. O perigo de uma chave com significado é que, embora em teoria elas sejam boas chaves, na prática não o são. Para simplesmente funcionar, as chaves precisam ser únicas. Para funcionar bem, elas precisam ser imutáveis. Embora os números atribuídos sejam supostamente únicos e imutáveis, erros humanos muitas vezes fazem com que eles não sejam nem uma coisa nem outra. Se você digitar errado trocando meu número do Seguro Social pelo da minha esposa, o registro resultante não é nem único e nem imutável (presumindo que você quisesse consertar o erro.) O banco de dados deve detectar o problema da unicidade, mas ele só pode fazer isso após meu registro ir para o sistema e é claro que isso só poderia acontecer após o erro. Como conseqüência, deve-se desconfiar de chaves com significado. Para sistemas pequenos e/ou casos muito estáveis, você pode se sair bem com o seu uso, mas, em geral, você deve tomar uma firme posição em favor da falta de significado da chave.

A próxima preocupação são as chaves simples *versus* as chaves compostas. Uma **chave simples** usa apenas um campo do banco de dados. Uma **chave composta** usa mais de um. A vantagem de uma chave composta é que ela, freqüentemente, é mais

fácil de usar quando uma tabela faz sentido no contexto de outra. Um bom exemplo são os pedidos e as linhas de itens, em que uma boa chave para a linha do item é uma chave composta pelo número do pedido e um número seqüencial que identifica a linha do pedido. Embora freqüentemente as chaves compostas façam sentido, há muito a ser dito sobre a maior uniformidade das chaves simples. Se você usar as chaves simples em todo lugar, você pode usar o mesmo código para toda manipulação de chaves. As chaves compostas requerem tratamento especial em classes concretas. (Com geração de código, isso não é problema). As chaves compostas também carregam um pouco de significado, então, ao usá-las, seja cuidadoso com a regra da unicidade e, especialmente, com a regra da imutabilidade.

Você tem que escolher o tipo da chave. A operação mais comum que você fará com uma chave é o teste de igualdade, de modo que você quer um tipo com uma operação rápida de igualdade. A outra operação importante é a obtenção da próxima chave. Assim, um tipo inteiro longo é freqüentemente a melhor aposta. As *strings* também podem funcionar, mas a verificação de igualdade pode ser mais lenta, e incrementar *strings* é um pouco mais difícil. As preferências do DBA podem decidir a questão.

(Cuidado com o uso de datas ou horas em chave. Elas não apenas têm significado como também levam a problemas de portabilidade e consistência. As datas são particularmente vulneráveis a isso, porque elas freqüentemente são armazenadas para uma precisão de frações de segundos, que podem facilmente sair de sincronismo e levar a problemas de identidade.)

Você pode ter chaves que são únicas na tabela ou em todo o banco de dados. Uma **chave única na tabela** é única nessa tabela, o que, afinal, é o mínimo que você precisa de uma chave. Uma **chave única no banco de dados** é única para todas as linhas de todas as tabelas do banco de dados. Uma chave única na tabela geralmente é boa, mas uma chave única no banco de dados é freqüentemente mais fácil de gerar e lhe permite usar um único *Mapa de Identidade* (196). Com os valores modernos sendo o que são, é bastante improvável que você fique sem números para novas chaves. Se você realmente insistir, pode recuperar as chaves dos objetos excluídos com um roteiro simples no banco de dados que compacte o espaço das chaves – ainda que rodar este roteiro vá requerer que você tire a aplicação do ar. Todavia, se você usar chaves de 64 *bits* (o que você deveria fazer), é improvável que você precise disso.

Seja cauteloso com herança quando você usar chaves únicas por tabela. Se você estiver usando *Herança de Tabela Concreta* (283) ou *Herança de Tabela de Classe* (276), a vida fica muito mais fácil com chaves que sejam únicas na hierarquia em vez de únicas em cada tabela. Ainda uso o termo "única na tabela", ainda que, a rigor, isso deveria ser como "única no grafo da herança."

O tamanho da sua chave pode afetar o desempenho, especialmente com índices. Isso depende do seu sistema de banco de dados e/ou de quantas linhas você tem, mas vale a pena fazer uma verificação por alto antes de tomar uma decisão.

Representando o *Campo Identidade* em um Objeto A forma mais simples de um *Campo Identidade* é um campo que corresponda ao tipo da chave no banco de dados. Assim, se você usar uma chave inteira simples, um campo inteiro funcionará muito bem.

As chaves compostas são mais problemáticas. A melhor aposta com elas é criar uma classe chave. Uma classe chave genérica pode armazenar uma seqüência de objetos que atuam como os elementos da chave. O comportamento chave para o objeto chave (tenho uma cota de trocadilhos por livro a preencher) é a igualdade. Também é útil pegar partes da chave quando você estiver mapeando para o banco de dados.

Se você usar a mesma estrutura básica para todas as chaves, pode executar toda a manipulação de chaves em uma *Camada Supertipo* (444). Você pode colocar comportamento padrão, que funcionará para a maioria dos casos na *Camada Supertipo* (444) e estendê-la para os casos excepcionais nos subtipos particulares.

Você pode ter uma única classe chave, que compreende uma lista genérica de objetos chave, ou uma classe chave para cada classe do domínio com campos explícitos para cada parte da chave. Normalmente prefiro ser explícito, porém, neste caso, não estou certo de que compense tanto assim. Você acaba com muitas classes pequenas que não fazem nada interessante. O maior benefício é que você pode evitar os erros causados por usuários, colocando os elementos da chave na ordem errada, mas isso não parece ser um grande problema na prática.

Se é provável que você vá importar dados de diferentes instâncias de bancos de dados, você precisa se lembrar que terá colisões de chaves a menos que tenha algum esquema para separar as chaves entre os diferentes bancos de dados. Você pode resolver este problema com algum tipo de migração de chaves nas importações, mas isso pode facilmente se tornar confuso.

Obtendo Uma Nova Chave Para criar um objeto, você precisará de uma chave. Isso parece ser uma questão simples, mas, muitas vezes, pode ser um problema considerável. Você tem três escolhas básicas: deixar o banco de dados gerá-la automaticamente, usar uma GUID ou gerar a sua própria.

Deixar o banco de dados gerar a chave deveria ser o caminho mais fácil. Cada vez que você insere dados no banco de dados, este gera uma chave primária única sem que você tenha de fazer nada. Parece bom demais para ser verdade, e, infelizmente, muitas vezes é. Nem todos os bancos de dados fazem isso da mesma maneira. Muitos dos que fazem lidam com isso de um modo que causa problemas para o mapeamento objeto-relacional.

O método de geração automática mais comum é declarar um **campo auto-gerado**, o qual, toda vez que você inserir uma linha, é incrementado para um novo valor. O problema com este esquema é que você não consegue determinar facilmente qual valor foi criado para a chave. Se você quiser inserir um pedido e diversas linhas de itens, você precisa da chave do novo pedido de modo que você possa colocar o valor na chave estrangeira da linha do item. Além disso, você precisa desta chave antes que a transação seja confirmada, de modo que possa gravar tudo dentro da transação. Infelizmente, os bancos de dados normalmente não lhe dão esta informação, de modo que você geralmente não pode usar este tipo de geração automática em qualquer tabela na qual precise inserir objetos associados.

Uma abordagem alternativa à geração automática é um **contador do banco de dados**, o qual o Oracle usa com a sua seqüência. Uma seqüência Oracle funciona enviando um comando *select* que referencia uma seqüência. O banco de dados então retorna um conjunto de registros SQL consistindo no próximo valor na seqüência. Você pode configurar o incremento de uma seqüência para qualquer valor inteiro, o que lhe permite obter diversas chaves de uma só vez. A pesquisa da seqüência é automaticamente executada em uma transação separada, de forma que acessar a seqüência não bloqueará outras transações inserindo registros no banco ao mesmo tempo. Um contador do banco de dados como este é perfeito para nossas necessidades, mas não é padrão e não está disponível em todos os bancos de dados.

Um **GUID** (Identificador Globalmente Único) é um número gerado em uma máquina que tem a garantia de ser único em todas as máquinas no espaço e no tempo. Muitas vezes plataformas lhe dão a API para gerar um GUID. O algoritmo é in-

teressante e envolve endereços de placas ethernet, hora do dia em nanossegundos, números de identificação dos *chips* e provavelmente o número de fios de cabelo no seu pulso esquerdo. Tudo que importa é que o número resultante é completamente único e desse modo uma chave segura. A única desvantagem do GUID é que a chave resultante é grande e isso pode ser igualmente um grande problema. Sempre há ocasiões em que alguém tem que digitar uma chave em uma janela ou expressão SQL, e chaves longas são difíceis tanto de digitar quanto de ler. Elas também podem levar a problemas de desempenho, especialmente com índices.

A última opção é gerar a sua própria chave. Um mecanismo simples para sistemas pequenos é fazer uma **varredura de tabela** usando a função SQL *max* para encontrar a maior chave na tabela e então incrementá-la de uma unidade para usá-la. Infelizmente, essa leitura bloqueia a tabela inteira enquanto estiver sendo executada, o que significa que funciona bem se as inserções forem raras, mas seu desempenho será muito diminuído se você tiver inserções rodando concorrentemente com atualizações na mesma tabela. Você também tem que assegurar-se de que possui um completo isolamento entre as transações, caso contrário pode acabar com diversas transações obtendo o mesmo valor de ID.

Uma abordagem melhor é usar uma **tabela de chaves** separada. Esta tabela tipicamente tem duas colunas: nome e próximo valor disponível. Se você usar chaves únicas no banco de dados, terá apenas uma linha nesta tabela. Se usar chaves únicas por tabela, terá uma linha para cada tabela no banco de dados. Para usar esta tabela de chaves, tudo o que você precisa fazer é ler essa linha e anotar o número, incrementar o número e gravá-lo de volta na linha. Você pode pegar muitas chaves de uma só vez adicionando um número apropriado quando atualizar a tabela de chaves. Isso diminui as custosas chamadas ao banco de dados e reduz a disputa na tabela de chaves.

Se você usar uma tabela de chaves, é uma boa idéia projetá-la de modo que o acesso a ela esteja em uma transação separada daquela que atualiza a tabela na qual você esteja inserindo. Digamos que eu esteja inserindo um pedido na tabela de pedidos. Para fazer isso, precisarei bloquear a linha da tabela de pedidos na tabela de chaves com um bloqueio de gravação (já que estou atualizando). Esse bloqueio irá durar enquanto durar a transação na qual estou, bloqueando com isso qualquer outra pessoa que queira uma chave. Para chaves únicas por tabela, isso significa qualquer pessoa inserindo na tabela de pedidos. Para chaves únicas no banco de dados, isso significa qualquer pessoa inserindo em qualquer lugar.

Colocando o acesso à tabela de chaves em uma transação separada, você só bloqueia a linha para essa transação, que é muito mais curta. O aspecto negativo é que, se você desfizer sua inserção na tabela de pedidos, a chave que você obteve da tabela de chaves fica perdida para todos. Felizmente números são baratos, então este não é um grande problema. Usar uma transação separada também lhe permite obter o ID assim que você cria o objeto na memória, o que muitas vezes ocorre um pouco antes de você abrir a transação para confirmar a transação de negócio.

Usar uma tabela de chaves afeta a escolha entre chave única por tabela ou por banco de dados. Se você usar uma chave única por tabela, tem que adicionar uma linha na tabela de chaves toda vez que acrescentar uma tabela no banco de dados. Isto é mais trabalhoso, porém reduz a disputa pela linha. Se você mantiver seus acessos à tabela de chaves em uma transação separada, a disputa não é um problema tão grande, especialmente se você obtém várias chaves em uma única chamada. Todavia se você não conseguir fazer com que a atualização da tabela de chaves fique em uma transação separada, você terá um forte argumento contra chaves únicas por banco de dados.

É bom separar o código para a obtenção de uma nova chave na sua própria classe, pois isso facilita a criação de um *Stub de Serviço* (469) para propósitos de testes.

Quando Usá-lo

Use um *Campo Identidade* quando houver um mapeamento entre os objetos na memória e as linhas em um banco de dados. Isso ocorre normalmente quando você usa um *Modelo de Domínio* (126) ou um *Gateway de Linhas de Dados* (158). Você não precisa deste mapeamento se estiver usando um *Roteiro de Transação* (120), um *Módulo Tabela* (134) ou um *Gateway de Tabela de Dados* (151).

Para um objeto pequeno com semântica de valor, como um objeto do tipo dinheiro ou uma faixa de datas que não terão sua própria tabela, é melhor usar um *Valor Embutido* (261). Para um grafo complexo de objetos que não precise ser pesquisado dentro do banco de dados relacional, o *LOB Serializado* (264) é normalmente mais fácil de escrever e apresenta um desempenho melhor.

Uma alternativa ao *Campo Identidade* é estender o *Mapa de Identidade* (196) para manter a correspondência. Isso pode ser usado para sistemas no qual você não queira armazenar um *Campo Identidade* no objeto em memória. O *Mapa de Identidade* (196) precisa procurar em ambas as direções: dê-me a chave de um objeto ou o objeto correspondente a uma chave. Não costumo ver isso com muita freqüências porque geralmente é mais fácil armazenar a chave no objeto.

Leitura Adicional

[Marinescu] discute várias técnicas para a geração de chaves.

Exemplo: Chave Integral (C#)

A forma mais simples de um *Campo Identidade* é um campo inteiro no banco de dados que mapeia para um campo inteiro em um objeto na memória.

```
class ObjetoDoDomínio...

    public const long PROCURADOR_ID = -1;
    public long Id = PROCURADOR_ID;
    public Boolean éNovo( ) {return Id == PROCURADOR_ID;}
```

Um objeto que tenha sido criado na memória mas que não tenha sido salvo no banco de dados não terá um valor para a sua chave. Para um objeto .NET este valor é um problema, uma vez que valores .NET não podem ser nulos. Eis por que o valor do procurador.

A chave se torna importante em dois locais: na busca e na inserção. Para uma busca você precisa formar uma consulta usando uma chave na cláusula *WHERE*. Em .NET você pode carregar muitas linhas em um conjunto de dados e então selecionar uma delas em particular com uma operação de busca.

```
class MapeadorDeJogadorDeCríquete...

    public JogadorDeCríquete Buscar (long id) {
        return (JogadorDeCríquete) BuscaAbstrata(id);
    }
```

```
class Mapeador...

    protected ObjetoDoDomínio BuscaAbstrata (long id) {
        DataRow linha = BuscarLinha (id);
        return (linha == null) ? null: Buscar(linha);
    }
    protected DataRow BuscarLinha (long id) {
        String filtro = String.Format("id = {0}", id);
        DataRow[ ] resultados = tabela.Select(filtro);
        return (resultados.Length == 0) ? null: resultados[0];
    }
    public ObjetoDoDomínio Buscar (DataRow linha) {
        ObjetoDoDomínio resultado = CriarObjetoDoDomínio( );
        Carregar(resultado, linha);
        return resultado;
    }
    abstract protected ObjetoDoDomínio CriarObjetoDoDomínio( );
```


A maior parte desse comportamento pode ficar na *Camada Supertipo* (444), mas freqüentemente você terá de definir o método de busca na classe concreta apenas para encapsular o *downcast*. É claro que você pode evitar isso em uma linguagem que não use a tipagem em tempo de compilação.

Com um simples *Campo Identidade* inteiro, o comportamento de inserção também pode ser armazenado na *Camada Supertipo* (444).

```
class Mapeador...

    public virtual long Inserir (ObjetoDoDomínio arg) {
        DataRow linha = tabela.NewRow( );
        arg.Id = lerPróximoID( );
        linha["id'] = arg.Id;
        Gravar (arg, linha);
        tabela.Rows.Add(linha);
        return arg.Id;
    }
```

Essencialmente, a inserção envolve a criação de uma nova linha e o uso da próxima chave nela. Assim que você a tiver, você pode gravar os dados do objeto localizado na memória nessa nova linha.

Exemplo: Usando uma Tabela de Chaves (Java)

por Matt Foemmel e Martin Fowler

Se o seu banco de dados suportar um contador e você não estiver preocupado com o fato de ficar dependente de um SQL específico de um determinado banco de dados, deve usar o contador. Mesmo se você estiver preocupado com o fato de ficar dependente de um banco de dados, ainda assim você deve considerá-lo – desde que o seu código de geração de chaves esteja bem encapsulado, você sempre poderá alterá-lo para um algoritmo portável mais tarde. Você pode até mesmo ter uma estratégia [Gang of Four] para usar contadores quando você os tiver e fazer os seus próprios em caso contrário.

Por enquanto, vamos supor que temos que fazer isso do modo difícil. A primeira coisa de que precisamos é uma tabela de chaves no banco de dados.

```
CREATE TABLE chaves (nome varchar primary key, próximoID int)
INSERT INTO chaves ('pedidos', 1)
```

Esta tabela contém uma linha para cada contador no banco de dados. Neste caso inicializamos a chave com 1. Se você estiver pré-carregando dados no banco de dados, precisará configurar o contador para um número apropriado. Se você quiser chaves únicas para todo o banco de dados, precisará de apenas uma linha. Se quiser chaves únicas por tabela, precisará de uma linha por tabela.

Você pode encapsular todo o seu código de geração de chaves na sua própria classe. Dessa maneira é mais fácil usá-lo mais extensamente em uma ou mais aplicações, e é mais fácil colocar a reserva de chave na sua própria transação.

Construímos um gerador de chaves com sua própria conexão de banco de dados, juntamente com a informação sobre quantas chaves pegar do banco de dados de cada vez.


```
class GeradorDeChaves...

    private Connection con;
    private String nomeDaChave;
    private long próximoId;
    private long máximoId;
    private int incrementarPor;
    public GeradorDeChaves (Connection con, String nomeDaChave, int incrementarPor) {
        this.con = con;
        this.nomeDaChave = nomeDaChave;
        this.incrementarPor = incrementarPor;
        próximoId = máximoId = 0;
        try {
            conn.setAutoCommit(false);
        } catch (SQLException exc) {
            throw new ApplicationException("Incapaz de desligar a confirmação
            automática", exc);
        }
    }
```

Precisamos assegurar que nenhuma confirmação automática está acontecendo, já que é fundamental ter a seleção e a atualização operando em uma mesma transação.

Quando pedimos uma nova chave, o gerador verifica se existe uma em cache uma vez de ir ao banco de dados.

```
class GeradorDeChaves...

    public synchronized Long próximaChave ( ) {
        if (próximoId == máximoId) {
            reservarIds( );
    }
        return new Long(próximoId++);
    }
```

Se o gerador não tiver uma chave em cache, ele precisa ir ao banco de dados.

```
class GeradorDeChaves...

    private void reservarIds( ) {
        PreparedStatement dec = null;
        ResultSet rs = null;
        long próximoIdNovo;
        try {
            dec = con.prepareStatement("SELECT próximoId FROM chaves WHERE nome = ?
            FOR UPDATE");
            dec.setString(1, nomeDaChave);
            rs = dec.executeQuery( );
            rs.next( );
            próximoIdNovo= rs.getLong(1);
        }
        catch (SQLException exc) {
            throw new ApplicationException("Incapaz de gerar ids", exc);
        }
        finally {
            DB.cleanUp(dec, rs);
        }
        long máximoIdNovo = próximoIdNovo + incrementarPor;
        dec = null;
        try {
            dec = con.prepareStatement("UPDATE chaves SET próximoId = ? WHERE nome = ? ");
            dec.setLong(1, máximoIdNovo);
            dec.setString(2, nomeDaChave);
            dec.executeUpdate( );
            conn.commit( );
            próximoId = próximoIdNovo;
            máximoId = máximoIdNovo;
        }
        catch (SQLException exc) {
            throw new ApplicationException("Incapaz de gerar ids", exc);
        }
        finally {
            DB.cleanUp(dec);
        }
    }
```

Nesse caso, usamos `SELECT ... FOR UPDATE` para dizer ao banco de dados para manter um bloqueio de gravação sobre a tabela de chaves. Este é um comando específico do Oracle, de modo que sua milhagem irá variar se você estiver usando algo diferente. Se você não puder efetuar um bloqueio de gravação no `SELECT`, corre o risco da transação falhar se outra pessoa chegar lá antes de você. Nesse caso, contudo, você pode executar novamente *reservarIds* com segurança, até que você obtenha um conjunto imaculado de chaves.

Exemplo: Usando uma Chave Composta (Java)

Usar uma chave inteira simples é uma solução boa e simples, porém muitas vezes você precisa de chaves de outros tipos ou de chaves compostas.

Uma Classe Chave Assim que você precisar de algo a mais, vale a pena criar uma classe chave. Uma classe chave precisa ser capaz de armazenar múltiplos elementos da chave e ser capaz de dizer se duas chaves são iguais.

```
class Chave...

    private Object[ ] campos;
    public boolean éIgual (Object obj) {
        if (! (obj instanceof Chave)) return false;
        Chave outraChave = (Chave) obj;
        if (this.campos.length != outraChave.campos.length) return false;
        for (int i = 0; i< campos.length; i++)
            if(! this.campos[i].equals(outraChave.campos[i])) return false;
        return true;
    }
```

O modo mais elementar de criar uma chave é com um *array* de parâmetros.

```
class Chave...

    public Chave (Object[ ] campos) {
        verificarChaveNãoNula (campos);
        this.campos = campos;
    }
    private void verificarChaveNãoNula (Object[ ] campos) {
        if (campos == null) throw new IllegalArgumentException("Não pode haver uma chave nula");
        for ( int i = 0; i< campos.length; i++)
            if (campos[i] == null)
                throw new IllegalArgumentException("Não pode haver um elemento nulo na chave");
    }
```

Se você descobrir que freqüentemente cria chaves com certos elementos, você pode acrescentar construtores convenientes. Exatamente quais construtores devem ser criados, vai depender dos tipos de chaves que sua aplicação possui.

```
class Chave...

    public Chave (long arg) {
        this.campos = new Object[1];
        this.campos[0] = new Long(arg);
    }
    public Chave (Object campo) {
        if (campo == null) throw new IllegalArgumentException("Não pode haver uma chave nula");
        this.campos = new Object[1];
        this.campos[0] = campo;
    }
    public Chave (Object arg1, Object arg2) {
        this.campos = new Object[2];
        this.campos[0] = arg1;
        this.campos[1] = arg2;
        verificarChaveNãoNula (campos);
    }
```

Não tenha receio de adicionar esses métodos por conveniência. Afinal de contas, conveniência é importante para todos que usam as chaves.

De modo similar, você pode adicionar métodos de acesso para ler partes das chaves. A aplicação precisará disso para os mapeamentos.

```
class Chave...

    public Object valor (int i) {
        return campos[i];
    }
    public Object valor ( ) {
        verificarSeÉChaveSimples( );
        return campos[0];
    }
    private void verificarSeÉChaveSimples ( ) {
        if (campos.length > 1)
            throw new IllegalStateException("Não posso pegar o valor de uma chave composta");
    }
    public long valorLongo( ) {
        verificarSeÉChaveSimples( );
        return valorLongo(0);
    }
    public long valorLongo(int i){
        if (! (campos[i] instanceof Long) )
        throw new IllegalStateException("Não posso usar valorLongo sobre uma chave que não seja do
tipo long");
        return ((Long) campos[i]).longValue( );
    }
```

Neste exemplo mapearemos para uma tabela de pedidos e uma de linhas de itens. A tabela de pedidos tem uma chave primária inteira simples, a chave primária da tabela de linhas de itens é uma chave composta pela chave primária da tabela de pedidos e um número seqüencial.

```
CREATE TABLE pedidos (ID int primary key, cliente varchar)
CREATE TABLE linhas_itens (idDoPedido int, seq int, quantia int, produto varchar,
                           primary key (idDoPedido, seq))
```

A *Camada Supertipo* (475) para objetos do domínio precisa ter um campo chave.

```
class ObjetoDoDomínioComChave...

    private Chave chave;
    protected ObjetoDoDomínioComChave (Chave ID) {
        this.chave = ID;
    }
    protected ObjetoDoDomínioComChave( ) {
    }
    public Chave lerChave( ){
        return chave;
    }
    public void gravarChave (Chave chave) {
        this.chave = chave;
    }
```

Lendo Assim como em outros exemplos neste livro, dividi o comportamento em busca (que chega à linha correta no banco de dados) e carregar (que carrega dados dessa linha para o objeto do domínio). Ambas as responsabilidades são afetadas pelo uso de um objeto chave.

A diferença principal entre estes e os outros exemplos neste livro (que usam chaves inteiras simples) é que temos que fatorar certas partes do comportamento que são sobrescritas por classes que têm chaves mais complexas. Neste exemplo, pressuponho que a maioria das tabelas usa chaves inteiras simples. Entretanto, algumas usam algo mais, de modo que tornei padrão o caso do inteiro simples e embuti o comportamento para isso no mapeador *Camada Supertipo* (444). A classe dos pedidos é um desses casos simples. Aqui está o código para o comportamento de busca:

```
class MapeadorDePedido...

    public Pedido buscar (Chave chave) {
        return (Pedido) buscaAbstrata(chave);
    }
    public Pedido buscar(Long id) {
        return buscar (new Chave(id) );
    }
    protected String stringDoComandoDeBusca ( ) {
        return "SELECT id, cliente FROM pedidos WHERE id = ?";
    }
```

```
class MapeadorAbstrato...

    abstract protected String stringDoComandoDeBusca ( );
    protected Map mapaCarregado = new HashMap( );
    public ObjetoDoDomínioComChave buscaAbstrata (Chave chave) {
        ObjetoDoDomínioComChave resultado = (ObjetoDoDomínioComChave) mapaCarregado.get (chave);
        if (resultado != null) return resultado;
        ResultSet rs = null;
        PreparedStatement comandoDeBusca = null;
        try {
            comandoDeBusca = DB.prepare(stringDoComandoDeBusca ( ));
            carregarComandoDeBusca(chave, comandoDeBusca);
            rs = comandoDeBusca.executeQuery( );
            rs.next( );
            if (rs.isAfterLast( ) ) return null;
            resultado = carregar(rs);
            return resultado;
        } catch (SQLException e) {
            throw new ApplicationException (e);
        } finally {
            DB.cleanUp(comandoDeBusca, rs);
        }
    }
    // método gancho para chaves que não sejam inteiros simples
    protected void carregarComandoDeBusca (Chave chave, PreparedStatement buscador)
    throws SQLException {
        buscador.setLong(1, chave.valorLongo( ));
    }
```

Extraí a construção do comando de busca, já que isso requer que parâmetros diferentes sejam passados para o comando preparado. A linha de item é uma chave composta, então ela precisa sobrescrever esse método.

```
class MapeadorDeLinhaDeitem...

    public LinhaDeItem buscar (long idDoPedido, long seq) {
        Chave chave = new Chave (new Long(idDoPedido), new Long(seq));
        return (LinhaDeItem) buscaAbstrata (chave);
    }
    public LinhaDeItem buscar (Chave chave) {
        return (LinhaDeItem) buscaAbstrata (chave);
    }
    public String stringDoComandoDeBusca( ) {
        return
            "SELECT idDoPedido, seq, quantia, produto "+
            "  FROM linhas_itens "+
            "  WHERE (idDoPedido = ?) AND (seq = ?)";
    }
    // métodos gancho sobrescritos para a chave composta
    protected void carregarComandoDeBusca (Chave chave, PreparedStatement buscador) throws SQLException {
        buscador.setLong (1, idDoPedido(chave));
        buscador.setLong (2, númeroDaSeqüência(chave));
    }
    // métodos auxiliares para extrair valores apropriados da chave da linha de item
    private static long idDoPedido (Chave chave) {
        return chave.longValue(0);
    }
    private static long númeroDaSeqüência (Chave chave) {
        return chave.longValue (1);
    }
```

Além de definir a interface para os métodos de busca e fornecer uma *string* SQL para o comando de busca, a subclasse precisa sobrescrever o método gancho para permitir a inclusão de dois parâmetros no comando SQL. Também escrevi dois métodos auxiliares para extrair os pedaços da informação da chave. Isso cria código mais claro do que aquele que eu obteria apenas colocando métodos explícitos de acesso com índices numéricos da chave. Esses índices literais cheiram a problemas.

O comportamento de carga mostra uma estrutura semelhante – o comportamento padrão na *Camada Supertipo* (444) para chaves inteiras simples, sobrescrito para os casos mais complexos. Neste caso, o comportamento da carga do pedido se parece com:

```
class MapeadorAbstrato...

    protected ObjetoDoDomínioComChave carregar (ResultSet rs) throws SQLException {
        Chave chave = criarChave(rs);
        if (mapaCarregado.containsKey(chave)) return (ObjetoDoDomínioComChave)
        mapaCarregado.get(chave);
        ObjetoDoDomínioComChave resultado = fazerCarga (chave, rs);
        mapaCarregado.put (chave, resultado);
        return resultado;
    }
    abstract protected ObjetoDoDomínioComChave fazerCarga (Chave id, ResultSet rs) throws SQLException;
```

```
    // método gancho para chaves que não sejam inteiros simples
    protected Chave criarChave (ResultSet rs) throws SQLException {
        return new Chave(rs.getLong(1));
    }
```

```
class MapeadorDePedido...

    protected ObjetoDoDomínioComChave fazerCarga (Chave chave, ResultSet rs)
    throws SQLException {
        String usuário = rs.getString("usuário");
        Pedido resultado = new Pedido(chave, usuário);
        RegistoMapeador.linhaDeItem( ).carregarTodasAsLinhasDeItemDo (resultado);
        return resultado;
    }
```

A linha de item precisa sobrescrever o método gancho para criar uma chave baseada em dois campos.


```
class MapeadorDeLinhaDeItem...

    protected ObjetoDoDomínioComChave fazerCarga (Chave chave, ResultSet rs) throws SQLException {
        Pedido oPedido = RegistroMapeador.pedido( ).busca(idDoPedido(chave));
        return fazerCarga (chave, rs, oPedido);
    }
    protected ObjetoDoDomínioComChave fazerCarga (Chave chave, ResultSet rs, Pedido pedido)
            throws SQLException
    {
        LinhaDeItem resultado;
        int quantia = rs.getInt("quantia");
        String produto = rs.getString("produto");
        resultado = new LinhaDeItem(chave, quantia, produto);
        pedido.adicionarLinhaDeItem(resultado); //conecta ao pedido
        return resultado;
    }
    //sobrescreve o caso padrão
    protected Chave criarChave (ResultSet rs) throws SQLException {
        Chave chave = new Chave (new Long(rs.getLong("idDoPedido")), new Long(rs.getLong("seq")));
        return chave;
    }
```

A linha de item também tem um método de carga separado para usar quando estiver carregando todas as linhas do pedido.

```
class MapeadorDeLinhaDeItem...

    public void carregarTodasAsLinhasDeItemDo (Pedido arg) {
        PreparedStatement dec = null;
        ResultSet rs = null;
        try {
            dec = DB.prepare(stringDaBuscaDoPedido);
            dec.setLong(1, arg.lerChave( ).longValue( ));
            rs = dec.executeQuery( );
            while (rs.next( ))
                carregar(rs, arg);
        } catch (SQLException e) {
            throw new ApplicationException (e);
```

```
        } finally {DB.cleanUp(dec, rs);
        }
    }
    private final static String stringDaBuscaDoPedido =
           "SELECT idDoPedido, seq, quantia, produto " +
               "FROM linhas_itens " +
               "WHERE idDoPedido = ?";
    protected ObjetoDoDomínioComChave carregar (ResultSet rs, Pedido pedido) throws SQLException {
        Chave chave = criarChave(rs);
        if (mapaCarregado.containsKey(chave)) return (ObjetoDoDomínioComChave) mapaCarregado.get(chave);
        ObjetoDoDomínioComChave resultado = fazerCarga(chave, rs, pedido);
        mapaCarregado.put(chave, resultado);
        return resultado;
    }
```


Você precisa do tratamento especial porque o objeto Pedido só é colocado no *Mapa de Identidade* (196) do pedido depois de criado. Criar um objeto vazio e inseri-lo diretamente no *Campo Identidade* evitaria a necessidade disso (página 173-174).

Inserção Assim como a leitura, a inserção tem uma ação padrão para uma chave inteira simples e os ganchos para serem sobrescritos para chaves mais interessantes. No supertipo do mapeador forneci uma operação para atuar como a interface, junto com um método modelo para executar o trabalho da inserção.

```
class MapeadorAbstrato...

    public Chave inserir (ObjetoDoDomínioComChave sujeito) {
        try {
            return executarInserção(sujeito, objetoBuscarPróximaChaveNoBancoDeDados( ));
        } catch (SQLException e) {
            throw new ApplicationException(e);
        }
    }
    protected Chave executarInserção (ObjetoDoDomínioComChave sujeito, Chave chave) throws SQLException {
        sujeito.gravarChave(chave);
        PreparedStatement dec = DB.prepare(stringDoComandoDeInserção( ));
        inserirChave (sujeito, dec);
        inserirDados(sujeito, dec);
        dec.execute( );
        mapaCarregado.put(sujeito.lerChave( ), sujeito);
        return sujeito.lerChave( ));
    }
    abstract protected String stringDoComandoDeInserção( );

class MapeadorDePedido...

    protected String stringDoComandoDeInserção ( ) {
        return "INSERT INTO pedidos VALUES (?,?)";
    }
```

Os dados do objeto vão para o comando *insert* por meio de dois métodos que separam os dados da chave dos dados básicos do objeto. Faço isso porque posso fornecer uma implementação padrão para a chave que funcionará para qualquer classe, tal como Pedido, que usa a chave inteira simples padrão.

```
class MapeadorAbstrato...

    protected void inserirChave (ObjetoDoDomínioComChave sujeito, PreparedStatement dec)
            throws SQLException
    {
        dec.setLong(1, sujeito.lerChave( ).longValue( ));
    }
```

O resto dos dados para o comando *insert* é dependente da subclasse em particular, então este comportamento é abstrato na superclasse.

```
class MapeadorAbstrato...

    abstract protected vos inserirDados (ObjetoDoDomínioComChave sujeito, PreparedStatement dec)
        throws SQLException;

class MapeadorDePedido...

    protected void inserirDados (ObjetoDoDomínioComChave sujeitoAbstrato, PreparedStatement dec)
throws SQLException {
        try {
            Pedido sujeito = (Pedido) sujeitoAbstrato;
            dec.setString(2, sujeito.lerCliente( ));
        } catch (SQLException e) {
            throw new ApplicationException(e);
        }
    }
```

A linha de item sobrescreve esses dois métodos. Ela extrai dois valores para a chave.

```
class MapeadorDeLinhaDeItem...

    protected String stringDoComandoDeInserção ( ) {
        return "INSERT INTO linhas_itens VALUES (?, ?, ?, ?)";
    }
    protected void inserirChave (ObjetoDoDomínioComChave sujeito, PreparedStatement dec)
            throws SQLException
    {
        dec.setLong(1, idDoPedido(sujeito.lerChave( )));
        dec.setLong(2, númeroDaSeqüência(sujeito.lerChave( )));
    }
```

Ela também fornece sua própria implementação do comando `insert` para o resto dos dados.

```
class MapeadorDeLinhaDeItem...

    protected void inserirDados (ObjetoDoDomínioComChave sujeito, PreparedStatement dec)
            throws SQLException
    {
        LinhaDeItem item = (LinhaDeItem) sujeito;
        dec.setInt(3, item.lerQuantia( ));
        dec.setInt(4, item.lerProduto( ));
    }
```

Colocar a carga dos dados no comando `insert` só vale a pena se a maior parte das classes usarem o mesmo campo simples como chave. Se houver mais variação na manipulação de chaves, então ter apenas um comando para inserir a informação é provavelmente mais fácil.

Gerar a próxima chave do banco de dados também é algo que eu posso separar em um caso padrão e um sobrescrito. Para o caso padrão posso usar o esquema da tabela de chaves do qual falei anteriormente. Todavia, para a linha de item nos deparamos com um problema. A chave da linha de item usa a chave do pedido como parte de sua chave composta. Entretanto, não existe uma referência da classe linha de item para a classe pedido, de modo que é impossível mandar uma linha de item inserir a si mesma no banco de dados sem fornecer também o pedido correto. Isso leva à sempre confusa abordagem de implementar o método na superclasse com uma exceção de operação não suportada.


```
class MapeadorDeLinhaDeItem...

    public Chave inserir (ObjetoDoDomínioComChave sujeito) {
        throw new UnsupportedOperationException
                ("Deve fornecer um pedido ao inserir uma linha de item");
    }
    public Chave inserir (LinhaDeItem item, Pedido pedido) {
        try {
            Chave chave = new Chave(pedido.lerChave( ).value( ), lerPróximonúmeroDaSeqüência(pedido));
            return executarInserção(item, chave);
        } catch (SQLException e) {
            throw new ApplicationException(e);
        }
    }
```

É claro que podemos evitar isso tendo uma conexão de retorno da linha de item para o pedido, tornando assim, efetivamente, a associação entre as duas classes bidirecional. Decidi não fazer isso aqui para ilustrar o que fazer quando você não tem essa conexão.

Fornecendo o pedido, é fácil obter da chave a parte referente ao pedido. O próximo problema é obter um número seqüencial para a linha do pedido. Para encontrar esse número, precisamos descobrir qual é o próximo número seqüencial disponível para um pedido, o que podemos fazer tanto com uma consulta em SQL usando *max* ou olhando as linhas de pedido dos pedidos em memória. Para este exemplo, usarei esta segunda opção.

```
class MapeadorDeLinhaDeItem...

    private Long lerPróximoNúmeroDaSeqüência (Pedido pedido) {
        carregarTodasAsLinhasDeItemDo (pedido);
        Iterator it = pedido.lerItens( ).iterator( );
        LinhaDeItem candidata = (LinhaDeItem) it.next( );
        while (it.hasNext( )) {
            LinhaDeItem esteItem = (LinhaDeItem) it.next( );
            if (esteItem.lerChave( ) == null) continue;
            if (númeroDaSeqüência(esteItem) > númeroDaSeqüência(candidata)) candidata = esteItem;
        }
        return new Long(númeroDaSeqüência(candidata) + 1);
    }
```

```
    private static long númeroDaSeqüência (LinhaDeItem li) {
        return númeroDaSeqüência (li.lerChave( ));
    }
    //o comparador não funciona bem aqui devido a chaves nulas não gravadas
    protected String linhaDaTabelaDeChaves ( ) {
        throw new UnsupportedOperationException( );
    }
```

Esse algoritmo seria muito melhor se eu usasse o método `Collections.max`, mas uma vez que podemos (e de fato iremos) ter pelo menos uma chave nula, esse método falharia.

Atualizações e Exclusões Após tudo isso, atualizações e exclusões são praticamente inofensivas. Mais uma vez, temos métodos abstratos para o suposto caso costumeiro e métodos sobrescritos para os casos especiais.

As atualizações funcionam desta forma:

```
class MapeadorAbstrato ..

    public void atualizar (ObjetoDoDomínioComChave sujeito) {
        PreparedStatement dec = null;
        try {
            dec = DB.prepare(stringDoComandoDeAtualização( ));
            carregarComandoDeAtualização(sujeito, dec);
            dec.execute( );
        } catch (SQLException e) {
            throw new ApplicationException(e);
        } finally {
            DB.cleanUp(dec);
        }
    }
    abstract protected String stringDoComandoDeAtualização( );
    abstract protected void carregarComandoDeAtualização(ObjetoDoDomínioComChave sujeito,
                          PreparedStatement dec)
            throws SQLException;

class MapeadorDePedido...

    protected void carregarComandoDeAtualização (ObjetoDoDomínioComChave sujeito, PreparedStatement dec)
            throws SQLException
    {
        Pedido pedido = (Pedido) sujeito;
        dec.setString(1, pedido.lerCliente( ));
        dec.setLong(2, pedido.lerChave( ).longValue( ));
    }
    protected String stringDoComandoDeAtualização ( ) {
        return "UPDATE pedidos SET cliente = ? WHERE id = ?";
    }

class MapeadorDeLinhaDeitem...

    protected String stringDoComandoDeAtualização ( ) {
        return
            "UPDATE linhas_itens " +
            "  SET quantia = ?, produto = ? " +
            "  WHERE idDoPedido = ? AND seq = ?";
```

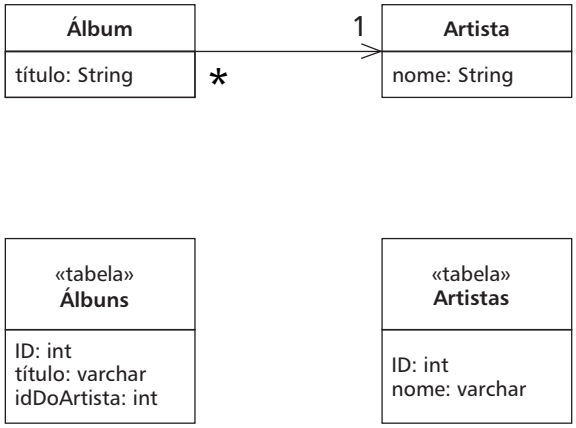
```
    }
    protected void carregarComandoDeAtualização (ObjetoDoDomínioComChave sujeito, PreparedStatement dec)
            throws SQLException
    {
        dec.setLong(3, idDoPedido(sujeito.lerChave( ));
        dec.setLong(4, númeroDaSeqüência(sujeito.lerChave( ));
        LinhaDeItem li = (LinhaDeitem) sujeito;
        dec.setInt(1, li.lerQuantia( ));
        dec.setString(2, li.lerProduto( ));
    }
```

As exclusões funcionam assim:

```
class MapeadorAbstrato...

    public void excluir(ObjetoDoDomínioComChave sujeito) {
        PreparedStatement dec = null;
        try {
            dec = DB.prepare(stringDoComandoDeExclusão( ));
            carregarComandoDeExclusão(sujeito, dec);
            dec.execute( );
        } catch (SQLException e) {
            throw new ApplicationException(e);
        } finally {
            DB.cleanUp(dec);
        }
    }
    abstract protected String stringDoComandoDeExclusão( );
    protected void carregarComandoDeExclusão(ObjetoDoDomínioComChave sujeito, PreparedStatement dec)
            throws SQLException
    {
        dec.setLong(1, sujeito.lerChave( ).longValue( ));
    }

class MapeadorDePedidos...

    protected String stringDoComandoDeExclusão ( ) {
        return "DELETE FROM pedidos   WHERE id = ?";
    }

class MapeadorDeLinhaDeItem...

    protected String stringDoComandoDeExclusão ( ) {
        return "DELETE FROM linhas_itens WHERE idDoPedido = ? AND seq = ?";
    }
    protected void carregarComandoDeExclusão(ObjetoDoDomínioComChave sujeito, PreparedStatement dec)
            throws SQLException
    {
        dec.setLong(1, idDoPedido(sujeito.lerChave( )));
        dec.setLong(2, númeroDaSeqüência(sujeito.lerChave( )));
    }
```

Mapeamento de Chave Estrangeira (Foreign Key Mapping)

Mapeia uma associação entre objetos para uma referência de chave estrangeira entre tabelas.



Os objetos podem se referir uns aos outros diretamente por meio de referências. Mesmo o sistema orientado a objetos mais simples conterá um pequeno grupo de objetos conectados entre si por todos os tipos de meios interessantes. Para gravar esses objetos em um banco de dados, é vital gravar essas referências. Contudo, já que os dados das referências são específicos à instância particular do programa sendo executado, você não pode simplesmente gravar os valores de dados em estado bruto. Outra complicação é o fato de que os objetos podem facilmente possuir coleções de referências para outros objetos. Tal estrutura viola a primeira forma normal dos bancos de dados relacionais.

Um *Mapeamento de Chave Estrangeira* mapeia uma referência a um objeto como uma chave estrangeira no banco de dados.

Como Funciona

A chave óbvia para esse problema é o *Campo Identidade* (215). Cada objeto contém a chave do banco de dados da tabela do banco de dados apropriada. Se dois objetos são conectados com uma associação, esta associação pode ser substituída por uma chave estrangeira no banco de dados. Colocado de forma simples, quando você salvar um álbum no banco de dados, você grava o ID do artista ao qual o álbum está associado no registro do álbum, como na Figura 12.1.

Esse é o caso simples. Um caso mais complicado surge quando você tem uma coleção de objetos. Você não pode gravar uma coleção no banco de dados, então você tem que inverter a direção da referência. Assim, se você tiver uma coleção de faixas no álbum, você coloca a chave estrangeira do álbum no registro da faixa, como nas Figuras 12.2 e 12.3. A complicação acontece quando você tem uma atualização. Atualizações significam que faixas podem ser acrescentadas ou removidas da coleção de um álbum. Como você pode saber quais alterações colocar no banco de dados? Basicamente, você tem três opções: (1) excluir e inserir, (2) adicionar um ponteiro reverso e (3) diferenciar a coleção.

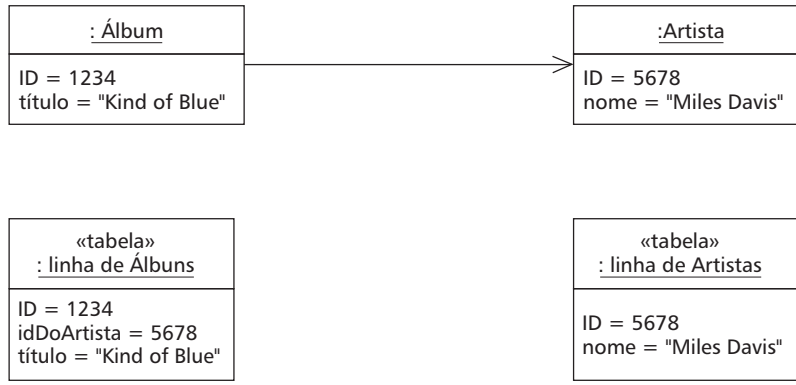


Figura 12.1 Mapeando uma coleção para uma chave estrangeira.

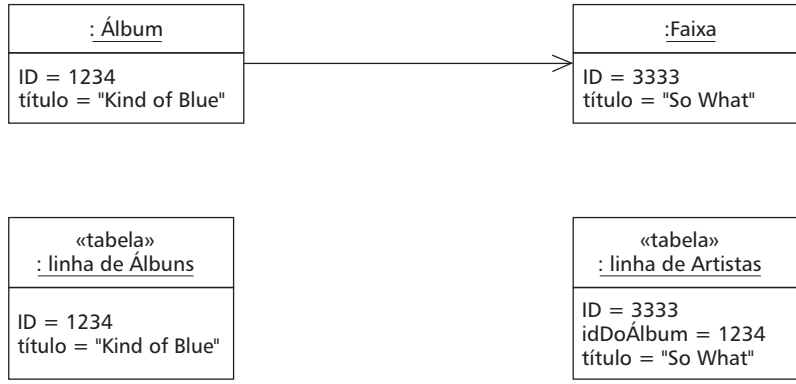


Figura 12.2 Mapeando uma coleção para uma chave estrangeira.

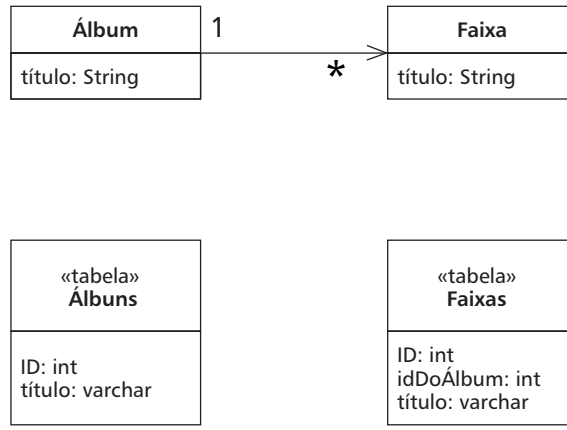


Figura 12.3 Classes e tabelas para uma referência multivalorada.

Com a exclusão e a inserção, você exclui todas as faixas do banco de dados vinculadas ao álbum e, então, insere todas as correntemente no álbum. À primeira vista isso parece um tanto espantoso, especialmente se você não tiver alterado nenhuma faixa. Contudo, a lógica é fácil de implementar e, como tal, funciona muito bem comparada com as alternativas. A desvantagem é que você só pode fazer isso se as faixas forem *Mapeamentos Dependentes* (256), o que significa que elas devem pertencer ao álbum e não podem ser referenciadas de fora dele.

Adicionar um ponteiro reverso coloca um vínculo da faixa de volta para o álbum, efetivamente tornando a associação bidirecional. Isso altera o modelo de objetos, mas, por outro lado, você agora pode tratar a atualização usando a técnica simples para campos univalorados.

Se nenhuma dessas opções for atrativa, você pode fazer uma diferenciação. Existem duas possibilidades aqui: diferenciar em relação ao estado corrente do banco de dados ou diferenciar em relação ao que você leu na primeira vez. Diferenciar em relação ao banco de dados envolve reler coleção do banco de dados e então compará-la com a coleção no álbum. Qualquer coisa no banco de dados que não estiver no álbum, obviamente foi removida. Qualquer coisa no álbum que não estiver no disco é obviamente um novo item a ser acrescentado. Observe então a lógica da aplicação para decidir o que fazer com cada item.

Diferenciar em relação ao que você leu na primeira vez significa que você tem que guardar o que você leu. Isso é melhor já que evita outra leitura do banco de dados. Você pode também ter de diferenciar em relação ao banco de dados se estiver usando um *Bloqueio Offline Otimista* (392).

No caso geral, qualquer coisa que tenha sido adicionada à coleção precisa ser primeiro verificada para ver se é um objeto novo. Você pode fazer isso vendo se ele tem uma chave. Se não tiver, ele precisa ser adicionado ao banco de dados. Esse passo é tornado muito mais simples com uma *Unidade de Trabalho* (187), porque, dessa maneira, qualquer objeto novo será primeiro inserido automaticamente. Em qualquer caso, você então encontra a linha associada no banco de dados e atualiza sua chave estrangeira para apontar para o álbum corrente.

Para a remoção, você tem que saber se a faixa foi movida para outro álbum, se ela não tem um álbum ou se foi completamente excluída. Se ela tiver sido movida para outro álbum, ela deve ser atualizada quando você atualizar esse outro álbum. Se ela não tiver um álbum, você precisa colocar um *null* na chave estrangeira. Se a faixa foi excluída, então ela deveria ser apagada. Tratar exclusões é muito mais fácil se o vínculo reverso for obrigatório, como ele é aqui, onde toda faixa deve estar em um álbum. Dessa maneira, você não tem que se preocupar em detectar itens removidos da coleção, já que eles serão atualizados quando você processar o álbum ao qual eles foram adicionados.

Se esse vínculo for imutável, significando que você não pode alterar o álbum de uma faixa, então a adição sempre significa inserção, e a remoção sempre significa exclusão. Isso torna as coisas ainda mais simples.

Uma das coisas com que deve-se tomar cuidado é a existência de ciclos nas suas associações. Digamos que você precise carregar um pedido, que tem uma associação com um cliente (o qual você carrega). O cliente tem um conjunto de pagamentos (os quais você carrega), e cada pagamento tem pedidos os quais ele está pagando, o que poderia incluir o pedido original que você está tentando carregar. Por conseguinte, você carrega o pedido (agora volte ao início deste parágrafo.)

Para evitar se perder em ciclos, você tem duas escolhas que, no final das contas, resumem-se à forma como você cria seus objetos. Normalmente, é uma boa idéia pa-

ra um método de criação incluir dados que lhe darão um objeto completamente formado. Se você fizer isso, precisará colocar a *Carga Tardia* (200) em pontos apropriados para quebrar os ciclos. Se você perder algum, você terá um estouro de pilha, mas se os seus testes forem bons o suficiente, você poderá gerenciar esse fardo.

A outra escolha é criar objetos vazios e imediatamente colocá-los em um *Mapa de Identidade* (196). Dessa maneira, quando o seu ciclo voltar ao início, o objeto já estará carregado, e o ciclo terminará. Os objetos que você cria não estão completamente formados, mas deveriam estar ao final do procedimento de carga. Isso evita ter que tomar decisões em casos especiais sobre o uso de *Carga Tardia* (200) apenas para fazer uma carga correta.

Quando Usá-lo

Um *Mapeamento de Chave Estrangeira* pode ser usado para quase todas as associações entre classes. O caso mais comum em que não é possível usá-lo é com associação muitos-para-muitos. As chaves estrangeiras são valores únicos, e a primeira forma normal diz que você não pode armazenar diversas chaves estrangeiras em um único campo. Em vez disso, você precisará usar um *Mapeamento de Tabela Associativa* (244).

Se você tem um campo do tipo coleção sem nenhum ponteiro reverso, você deverá considerar se o lado "muitos" da associação deve ser um *Mapeamento Dependente* (256). Se esse for o caso, isso pode simplificar o tratamento da coleção.

Se o objeto relacionado é um *Objeto Valor* (453), então você deveria usar um *Valor Embutido* (261).

Exemplo: Referência Univalorada (Java)

Este é o caso mais simples, em que um álbum tem uma única referência para um artista.

```
class Artista...

    private String nome;
    public Artista (Long ID, String nome) {
        super (ID);
        this.nome = nome;
    }
    public String lerNome( ) {
        return nome;
    }
    public void gravarNome(String nome) {
        this.nome = nome;
    }

class Álbum...

    private String título;
    private Artista artista;
    public Álbum (Long ID, String título, Artista artista) {
        super (ID);
        this.título = título;
        this.artista = artista;
    }
    public String lerTítulo ( ) {
```

```
        return título;
    }
    public void gravarTítulo (String título) {
        this.título = título;
    }
    public Artista lerArtista ( ) {
        return artista;
    }
    public void gravarArtista (Artista artista) {
        this.artista = artista;
    }
```

A Figura 12.4 mostra como você pode carregar um álbum. Quando o mapeador de um álbum é instruído a carregar um determinado álbum, ele consulta o banco de dados e traz o conjunto resultante dessa pesquisa. Ele então procura no conjunto resultante por campo chave estrangeira e descobre esse objeto. Ele agora pode criar o álbum com os objetos apropriados encontrados. Se o objeto Artista já estava na memória, ele seria trazido do cache. Caso contrário, ele seria carregado do banco de dados da mesma maneira.

A operação de busca usa comportamento abstrato para manipular um *Mapa de Identidade* (196).


```
class MapeadorDeÁlbum...

    public Álbum buscar (Long id) {
        return (Álbum) buscaAbstrata(id);
    }
    protected String comandoDeBusca ( ) {
        return "SELECT ID, título, idDoArtista FROM álbuns WHERE ID = ?";
    }

class MapeadorAbstrato...

    abstract protected String comandoDeBusca ( );
    protected ObjetoDoDomínio buscaAbstrata (Long id) {
        ObjetoDoDomínio resultado = (ObjetoDoDomínio) mapaCarregado.get(id);
        if (resultado != null) return resultado;
        PreparedStatement dec = null;
        ResultSet rs = null;
        try {
            dec = DB.prepare(comandoDeBusca( ));
            dec.setLong(1, id.longValue( ));
            rs = dec.executeQuery( );
            rs.next( );
            resultado = carregar (rs);
            return resultado;
        } catch (SQLException e) {
            throw new ApplicationException (e);
        } finally {DB.cleanUp(dec, rs);}
    }
    private Map mapaCarregado = new HashMap( );
```

A operação de busca chama uma operação de carga para efetivamente carregar os dados para o álbum.

```
class MapeadorAbstrato...

    protected ObjetoDoDomínio carregar (ResultSet rs) throws SQLException {
        Long id = new Long(rs.getLong(1));
        if (mapaCarregado.containsKey(id)) return (ObjetoDoDomínio) mapaCarregado.get(id);
        ObjetoDoDomínio resultado = fazerCarga (id, rs);
        fazerRegistro (id, resultado);
        return resultado;
    }
    protected void fazerRegistro (Long id, ObjetoDoDomínio resultado) {
        Assert.isFalse (mapaCarregado.containsKey(id));
        mapaCarregado.put(id, resultado);
    }
    abstract protected ObjetoDoDomínio fazerCarga (Long id, ResultSet rs) throws SQLException;

class MapeadorDeÁlbum...

    protected ObjetoDoDomínio fazerCarga (Long id, ResultSet rs) throws SQLException {
        String título = rs.getString(2);
        long idDoArtista = rs.getLong(3);
        Artista artista = RegistroMapeador.artista( ).buscar(idDoArtista);
        Álbum resultado = new Álbum(id, título, artista);
        return resultado;
    }
```


Para atualizar um álbum, o valor da chave estrangeira é extraído do objeto artista associado.

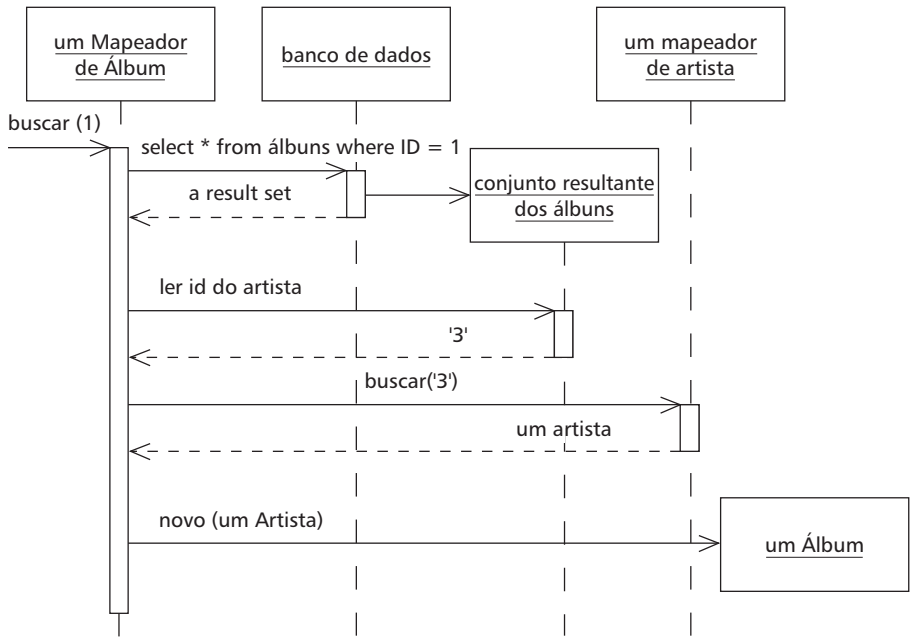


Figura 12.4 Seqüência para carregar um campo univalorado.

```
class MapeadorAbstrato...

    abstract public void atualizar (ObjetoDoDomínio arg);

class MapeadorDeÁlbum...

    public void atualizar (ObjetoDoDomínio arg) {
        PreparedStatement dec = null;
        try {
            dec = DB.prepare(
                   "UPDATE álbuns SET título = ?, idDoArtista = ? WHERE id = ?";
            dec.setLong(3, arg.lerID( ).longValue( ));
            Álbum álbum = (Álbum) arg;
            dec.setString(1, álbum.lerTítulo( ));
            dec.setLong(2, álbum.lerArtista( ).lerID( ).longValue( ));
            dec.execute( );
        } catch (SQLException e) {
            throw new ApplicationException(e);
        } finally {
            cleanUp(dec);
        }
    }
```


Exemplo: Busca Multitabelas (Java)

Embora seja conceitualmente claro executar uma pesquisa por tabela, muitas vezes isso é ineficiente, já que SQL consiste de chamadas remotas, e estas chamadas são lentas. Por conseguinte, freqüentemente vale a pena encontrar maneiras de recuperar informações de diversas tabelas em uma única pesquisa. Posso modificar o exemplo anterior para usar uma única consulta para obter as informações tanto do álbum quanto do artista com uma única chamada SQL. A primeira alteração é a do SQL do comando de busca.

```
class MapeadorDeÁlbum...

    public Álbum buscar (Long id) {
        return (Álbum) buscaAbstrata(id);
    }
    protected String comandoDeBusca ( ) {
        return "SELECT a.ID, a.título, a.idDoArtista, r.nome "+
            " FROM álbuns a, artistas r, " +
            " WHERE ID = ? AND a.idDoArtista = r.ID";
    }
```

A seguir uso um método de carga diferente que carrega tanto a informação do álbum quanto a do artista.

```
class MapeadorDeÁlbum...

    protected ObjetoDoDomínio fazerCarga (Long id, ResultSet rs) throws SQLException {
        String título = rs.getString(2);
        long idDoArtista = rs.getLong(3);
        MapeadorDeArtista mapeadorDeArtista = RegistroMapeador.artista( );
        Artista artista;
        if (mapeadorDeArtista.estáCarregado(idDoArtista))
```

```
            artista = mapeadorDeArtista.buscar(idDoArtista);
        else
            artista = carregarArtista(idDoArtista, rs);
        Álbum resultado = new Álbum(id, título, artista);
        return resultado;
    }
    private Artista carregarArtista (long id, ResultSet rs) throws SQLException {
        String nome = rs.getString(4);
        Artista resultado = new Artista(new Long(id), nome);
        RegistroMapeador.artista( ).registrar(resultado.lerID( ), resultado);
        return resultado;
    }
```

Há alguma animosidade envolvendo a decisão sobre onde colocar o método que mapeia o resultado da consulta SQL para o objeto artista. Por um lado é melhor colocá-lo no mapeador de artista, já que esta é a classe que normalmente carrega o artista. Por outro lado, o método de carga é intimamente associado ao SQL e, dessa maneira, deveria ficar com a consulta SQL. Neste caso, votei na segunda opção.

Exemplo: Coleção de Referências (C#)

O caso da coleção de referências ocorre quando você tem um campo que constitui uma coleção. Usarei aqui um exemplo de equipes e jogadores em que iremos supor que não podemos tornar um jogador um *Mapeamento Dependente* (256) (Figura 12.5).

```
class Equipe...

    public String Nome;
    public IList Jogadores; {
        get {return ArrayList.ReadOnly(dadosDosJogadores);}
        set {dadosDosJogadores = new ArrayList(valor);}
    }
    public void AdicionarJogador(Jogador arg) {
        dadosDosJogadores.Add(arg);
    }
    private IList dadosDosJogadores = new ArrayList( );
```

No banco de dados, isso será manipulado com o registro do jogador tendo uma chave estrangeira para a equipe (Figura 12.6).

```
class MapeadorDeEquipe...

    public Equipe Buscar(long id) {
        return (Equipe) BuscaAbstrata(id);
    }
```

Figura 12.5 Uma equipe com vários jogadores.

Figura 12.6 Estrutura do banco de dados para uma equipe com vários jogadores.

```
class MapeadorAbstrato...

    protected ObjetoDoDomínio BuscaAbstrata (long id) {
        Assert.True (id != ObjetoDoDomínio.PROCURADOR_ID);
        DataRow linha = BuscarLinha(id);
        return (linha == null) ? null: Carregar(linha);
    }
    protected DataRow BuscarLinha (long id) {
        String filtro = String.Format("id = {0}", id);
        DataRow [ ] resultados = tabela.Select(filtro);
        return (resultados.Length == 0) ? null: resultados[0];
    }
    protected DataTable tabela {
        get {return acd.Dados.Tables[NomeDaTabela];}
    }
    public ArmazenadorDoConjuntoDeDados acd;
    abstract protected String NomeDaTabela {get;}

class MapeadorDeEquipe...

    protected override String NomeDaTabela {
        get {return "Equipes";}
    }
```

O armazenador do conjunto de dados é uma classe que armazena o conjunto de dados em uso, junto com os adaptadores necessários para atualizá-los no banco de dados.

```
class ArmazenadorDoConjuntoDeDados...

    public DataSet Dados = new DataSet( );
    private Hashtable AdaptadoresDeDados = new Hashtable( );
```

Para esse exemplo, iremos supor que a classe já foi povoada por algumas consultas apropriadas.

O método de busca chama um método de carga para efetivamente carregar os dados no novo objeto.

```
class MapeadorAbstrato...

    protected ObjetoDoDomínio Carregar (DataRow linha) {
        long id = (int) linha["id"];
        if (mapaDeIdentidade[id] != null) return (ObjetoDoDomínio) mapaDeIdentidade[id];
        else {
            ObjetoDoDomínio resultado = CriarObjetoDoDomínio( );
            resultado.Id = id;
```

```
            mapaDeIdentidade.Add (resultado.Id, resultado);
            fazerCarga (resultado, linha);
            return resultado;
        }
    }
    abstract protected ObjetoDoDomínio CriarObjetoDoDomínio ( );
    private IDictionary mapaDeIdentidade = new Hastable( );
    abstract protected void fazerCarga (ObjetoDoDomínio obj, DataRow linha);

class mapeadorDeEquipe...

    protected override void fazerCarga (ObjetoDoDomínio obj, DataRow linha) {
        Equipe equipe = (Equipe) obj;
        equipe.Nome = (String) linha["nome"];
        equipe.Jogadores = RegistroMapeador.Jogador.BuscarPorEquipe(equipe.Id);
    }
```


Para trazer os jogadores, executo um método especializado de busca sobre o mapeador de jogadores.

```
class MapeadorDeJogador...

    public IList BuscarPorEquipe (long id) {
        String filtro = String.Format("idDaEquipe {0}", id);
        DataRow[ ] linhas = tabela.Select(filtro);
        IList resultado = new ArrayList( );
        foreach (DataRow linha in linhas) {
            resultado.Add(Carregar (linha));
        }
        return resultado;
    }
```

Para a atualização, a equipe grava seus próprios dados e delega ao mapeador de jogadores a gravação dos dados na tabela de jogadores.

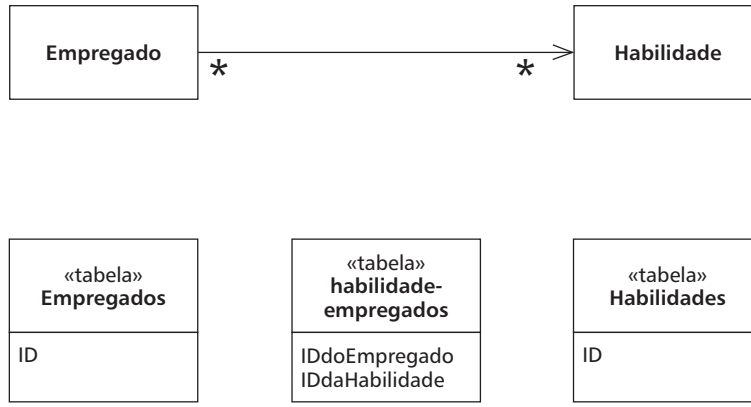
```
class MapeadorAbstrato...

    public virtual void Atualizar (ObjetoDoDomínio arg) {
        Gravar (arg, BuscarLinha(arg.Id));
    }
    abstract protected void Gravar (ObjetoDoDomínio obj, DataRow linha);

class MapeadorDeEquipe...

    protected override void Gravar (ObjetoDoDomínio obj, DataRow linha) {
        Equipe equipe = (Equipe) obj;
        linha["nome"] = equipe.Nome;
        gravarJogadores (equipe);
    }
    private void gravarJogadores (Equipe equipe) {
        foreach (Jogador j in equipe.Jogadores) {
            RegistroMapeador.Jogador.AssociaçãoEquipe (j, equipe.Id);
        }
    }
```

```
class MapeadorDeJogador...

    public void AssociaçãoEquipe (Jogador jogador, long idDaEquipe) {
        DataRow linha = BuscarLinha (jogador.Id);
        linha["idDaEquipe"] = idDaEquipe;
    }
```

O código de atualização torna-se muito mais simples pelo fato da associação do jogador com a equipe ser obrigatória. Se movermos um jogador de uma equipe para outra, desde que atualizemos ambas as equipes, não temos que fazer uma diferenciação complicada para lidar com os jogadores. Deixarei esse caso como exercício para o leitor.

Mapeamento de Tabela Associativa (Association Table Mapping)

Grava uma associação como uma tabela com chaves estrangeiras para as tabelas que são vinculadas pela associação.



Os objetos podem manipular campos multivalorados com bastante facilidade usando coleções como valores de campos. Os bancos de dados relacionais não têm essa característica e são restritos apenas a campos univalorados. Quando você estiver mapeando uma associação um-para-muitos, pode manipulá-la usando um *Mapeamento de Chave Estrangeira* (233), basicamente usando uma chave estrangeira para o lado univalorado da associação. Contudo, uma associação muitos-para-muitos não pode fazer isso porque não há lado univalorado para armazenar a chave estrangeira.

A resposta é a solução clássica que tem sido há décadas usada pela comunidade de dados relacionais: criar uma tabela extra para armazenar o relacionamento. Depois use um *Mapeamento de Tabela Associativa* para mapear o campo multivalorado para essa tabela de vinculação.

Como Funciona

A idéia básica por trás do *Mapeamento de Tabela Associativa* é usar uma tabela de vinculação para armazenar a associação. Essa tabela tem apenas os IDs das chaves estrangeiras para as duas tabelas que são vinculadas. Ela possui uma linha para cada par de objetos associados.

A tabela de vinculação não tem um objeto correspondente em memória. A conseqüência disso é que ela não tem ID. Sua chave primária é a composição das duas chaves primárias das tabelas que são associadas.

Em termos simples, para carregar dados da tabela de vinculação, você executa duas consultas. Considere carregar as habilidades de um empregado. Neste caso, pelo menos conceitualmente, você executa consultas em duas etapas. A primeira etapa consulta a tabela `habilidadesEmpregados` para encontrar todas as linhas que referenciam o empregado que você quer. A segunda etapa, para cada linha do conjunto resultante, encontra o objeto habilidade correspondente usando o ID relacionado.

Se toda a informação já estiver em memória, este esquema funciona bem. Se não estiver, ele pode ser terrivelmente custoso no que diz respeito às consultas, já que você tem de executar uma consulta para cada habilidade na tabela associativa. Você po-

de evitar este custo juntando a tabela de habilidades com a tabela associativa, o que lhe permite obter todos os dados em uma única consulta, embora ao custo de tornar o mapeamento um pouco mais complicado.

Atualizar os dados referentes à associação das tabelas envolve muitas das questões da atualização de campos multivalorados. Felizmente a questão é tornada muito mais simples já que você pode, de muitas maneiras, tratar a tabela associativa como um *Mapeamento Dependente* (256). Nenhuma outra tabela deve referenciar a tabela associativa, de modo que você pode livremente criar e destruir vínculos na medida em que eles forem necessários.

Quando Usá-lo

O caso canônico de *Mapeamento de Tabela Associativa* é uma associação muitos-para-muitos, já que realmente não há alternativas para essa situação.

O *Mapeamento de Tabela Associativa* também pode ser usada para qualquer outra forma de associação. Entretanto, devido ao fato de ele ser mais complexo que o *Mapeamento de Chave Estrangeira* (233) e envolver uma junção (*join*) extra, ele, usualmente, não é a melhor escolha. Ainda assim, em dois casos, o *Mapeamento de Tabela Associativa* é apropriado para uma associação mais simples. Ambos os casos envolvem bancos de dados em que você tem menos controle sobre o esquema. Às vezes, você pode precisar associar duas tabelas existentes, mas não pode adicionar colunas a essas tabelas. Neste caso, você pode criar uma nova tabela e usar o *Mapeamento de Tabela Associativa*. Outras vezes, um esquema existente usa uma tabela associativa, mesmo quando ela não é realmente necessária. Neste caso, freqüentemente é mais fácil usar um *Mapeamento de Tabela Associativa* do que simplificar o esquema do banco de dados.

Em um projeto de banco de dados relacional, você pode muitas vezes ter tabelas associativas que também carregam informação sobre o relacionamento. Um exemplo é uma tabela associativa pessoa/companhia que também contenha informações sobre o emprego da pessoa na companhia. Neste caso, a tabela pessoa/companhia realmente corresponde a um verdadeiro objeto do domínio.

Exemplo: Empregados e Habilidades (C#)

Aqui está um exemplo simples usando o modelo do esboço. Temos uma classe Empregado com um conjunto de habilidades, cada uma das quais pode aparecer associada a mais de um empregado.

```
class Empregado...

    public IList Habilidades {
        get {return ArrayList.ReadOnly(dadosDasHabilidades);}
        set {dadosDasHabilidades = new ArrayList(valor);}
    }
    public void AdicionarHabilidade (Habilidade arg) {
        dadosDasHabilidades.Add(arg);
    }
    public void removerHabilidade (Habilidade arg) {
        dadosDasHabilidades.Remove(arg);
    }
    private IList dadosDasHabilidades = new ArrayList( );
```

Para carregar um empregado do banco de dados, precisamos pegar as habilidades usando um mapeador de empregado. Cada classe mapeadora de empregados tem um método de busca que cria um objeto empregado. Todos os mapeadores são subclasses da classe mapeadora abstrata que junta serviços comuns para os mapeadores.

```
class MapeadorDeEmpregado...

    public Empregado Buscar (long id) {
        return (Empregado) BuscaAbstrata (id);
    }

class MapeadorAbstrato...

    protected ObjetoDoDomínio BuscaAbstrata (long id) {
        Assert.isTrue (id != ObjetoDoDomínio.PROCURADOR_ID);
        DataRow linha = BuscarLinha(id);
        return (linha == null) ? null: Carregar(linha);
    }
    protected DataRow BuscarLinha (long id) {
        String filtro = String.Format("id = {0}", id);
        DataRow[ ] resultados = tabela.Select(filtro);
        return (resultados.Length == 0) ? null: resultados[0];
    }
    protected DataTable tabela {
        get {return acd.Dados.Tables[NomeDaTabela];}
    }
    public ArmazenadorDoConjuntoDeDados acd;
    abstract protected String NomeDaTabela {get;}

class MapeadorDeEmpregado...

    protected override String NomeDaTabela {
        get {return "Empregados";}
    }
```

O armazenador do conjunto de dados é um objeto simples que contém um conjunto de dados ADO.NET e os adaptadores relevantes para gravar este conjunto no banco de dados.

```
class ArmazenadorDoConjuntoDeDados...

    public DataSet Dados = new DataSet( );
    private Hashtable AdaptadoresDeDados = new Hashtable( );
```

Para tornar este exemplo simples – na verdade, simplista – iremos supor que o conjunto de dados já foi carregado com todos os dados de que precisamos.

O método de busca chama os métodos de carga para carregar os dados do empregado.

```
class MapeadorAbstrato...

    protetced ObjetoDoDomínio Carregar (DataRow linha) {
        long id = (int) linha["id"];
        if (mapaDeIdentidade[id] != null return (ObjetoDoDomínio) mapaDeIdentidade[id];
        else {
            ObjetoDoDomínio resultado = CriarObjetoDoDomínio( );
```

```
            resultado.Id = id;
            mapaDeIdentidade.Add(resultado.Id, resultado);
            fazerCarga(resultado, linha);
            return resultado;
        }
    }
    abstract protected ObjetoDoDomínio CriarObjetoDoDomínio( );
    private IDictionary mapaDeIdentidade = new Hashtable( );
    abstract protected void fazerCarga (ObjetoDoDomínio obj, DataRow linha);
```

```
class MapeadorDeEmpregado...

    protected override void fazerCarga (ObjetoDoDomínio obj, DataRow linha) {
        Empregado emp = (Empregado) obj;
        emp.Nome = (String) linha["nome"];
        carregarHabilidades(emp);
    }
```


Carregar as habilidades é suficientemente complicado para justificar um método separado para realizar o trabalho.

```
class MapeadorDeEmpregado...

    private IList carregarHabilidades (Empregado emp) {
        DataRow[ ] linhas = linhasHabilidadesAssociadas(emp);
        IList resultado = new ArrayList( );
        foreach (DataRow linha in linhas) {
            long idDaHabilidade = (int) linha["idDaHabilidade"];
            emp.AdicionarHabilidade(RegistroMapeador.Habilidade.Buscar(idDaHabilidade));
        }
        return resultado;
    }
    private DataRow[ ] linhasHabilidadesAssociadas (Empregado emp) {
        String filtro = String.Format("idDoEmpregado = {0}", emp.Id);
        return tabelaHabilidadesAssociadas.Select(filtro);
    }
    private DataTable tabelaHabilidadesAssociadas {
        get {return acd.Dados.Tables["habilidadesEmpregados"];}
    }
```

Para tratar as alterações nas informações sobre as habilidades, usamos um método de atualização no mapeador abstrato.

```
class MapeadorAbstrato...

    public virtual void Atualizar (ObjetoDoDomínio arg) {
        Gravar (arg, BuscarLinha(arg.Id));
    }
    abstract protected void Gravar (ObjtetoDoDomínio arg, DataRow linha);
```

O método de atualização chama um método de gravação na subclasse.

```
class MapeadorDeEmpregado...

    protected override void Gravar (ObjetoDoDomínio obj, DataRow linha) {
        Empregado emp = (Empregado) obj;
```

```
        linha["nome"] = emp.Nome;
        gravarHabilidades(emp);
    }
```

Mais uma vez, criei um método separado para gravar as habilidades.

```
class MapeadorDeEmpregado...

    private void gravarHabilidades (Empregado emp) {
        excluirHabilidades(emp);
        foreach (Habilidade h in emp.Habilidades) {
            DataRow linha = tabelaHabilidadesAssociadas.NewRow( );
            linha["idDoEmpregado"] = emp.Id;
            linha["idDaHabilidade"] = h.Id;
            tabelaHabilidadesAssociadas.Rows.Add(linha);
        }
    }
    private void excluirHabilidades (Empregado emp) {
        DataRow[ ] linhasDasHabilidades = linhasHabilidadesAssociadas(emp);
        foreach (DataRow li in linhasDasHabilidades) li.Delete( );
    }
```


A lógica aqui executa a ação simples de apagar todas as linhas existentes na tabela associativa e criar novas linhas. Isso me poupa de ter que descobrir quais foram adicionadas e quais foram excluídas.

Exemplo: Usando SQL Direto (Java)

Uma das coisas boas de ADO.NET é que ele me permite discutir o básico de um mapeamento objeto-relacional sem me prender aos detalhes pegajosos relativos à minimização de consultas. Com outros esquemas de mapeamento relacional, você fica mais próximo do SQL e tem que levar muito disso em conta.

Quando você está indo diretamente ao banco de dados, é importante minimizar as consultas. Para a minha primeira versão, pegarei o empregado e todas as suas habilidades em duas consultas diferentes ao banco de dados. Isso é fácil de acompanhar, mas não é a melhor maneira de fazer, então, tenha paciência comigo.

Aqui está a DDL para as tabelas:

```
create table empregados (ID int primary key, prenome varchar, sobrenome varchar)
create table habilidades (ID int primary key, nome varchar)
create table empregadoHabilidades (idDoEmpregado int, idDaHabilidade int, primary key
(idDoEmpregado, idDaHabilidade))
```

Para carregar um único Empregado, seguirei uma abordagem semelhante à que utilizei antes. O mapeador de empregado define um envoltório simples para um método abstrato de busca na *Camada Supertipo* (444).

```
class MapeadorDeEmpregado...

    public Empregado buscar (long chave) {
        return buscar (new Long(chave));
    }
    public Empregado buscar (Long chave) {
```

```
        return (Empregado) buscaAbstrata (chave);
    }
    protected String comandoDeBusca ( ) {
        return
            "SELECT " + LISTA_DE_COLUNAS +
            "  FROM empregados" +
            "  WHERE ID = ?";
    }
    public static final String LISTA_DE_COLUNAS = " ID, sobrenome, prenome ";
```

```
class MapeadorAbstrato...

    protected ObjetoDoDomínio buscaAbstrata (Long id) {
        ObjetoDoDomínio resultado = (ObjetoDoDomínio) mapaCarregado.get(id);
        if (resultado != null) return resultado;
        PreparedStatement dec = null;
        ResultSet rs = null;
        try {
            dec = DB.prepare(comandoDeBusca( ));
            dec.setLong(1, id.longValue( ));
            rs = dec.executeQuery( );
            rs.next( );
            resultado = carregar(rs);
            return resultado;
        } catch (SQLException e) {
            throw new ApplicationException(e);
        } finally {DB.cleanUp(dec, rs);
        }
    }
    abstract protected String comandoDeBusca( );
    protected Map mapaCarregado = new HashMap( );
```


Os métodos de busca chamam então os métodos de carga. Um método abstrato de carga trata a carga do ID enquanto os dados reais do empregado são carregados no mapeador de empregados.

```
class MapeadorAbstrato...

    protected ObjetoDoDomínio carregar (ResultSet rs) throws SQLException {
        Long id = new Long(rs.getLong(1));
        return carregar (id, rs);
    }
    public ObjetoDoDomínio carregar (Long id, ResultSet rs) throws SQLException {
        if (foiCarregado(id)) return (ObjetoDoDomínio) mapaCarregado.get(id);
        ObjetoDoDomínio resultado = fazerCarga(id, rs);
        mapaCarregado.put(id, resultado);
        return resultado;
    }
    abstract protected ObjetoDoDomínio fazerCarga (Long id, ResultSet rs) throws SQLException;
```

```
class MapeadorDeEmpregado...

    protected ObjetoDoDomínio fazerCarga (Long id, ResultSet rs) throws SQLException {
        Empregado resultado = new Empregado(id);
        resultado.gravarPrenome(rs.getString("prenome");
        resultado.gravarSobrenome(rs.getString("sobrenome");
```

```
        resultado.gravarHabilidades(carregarHabilidades(id));
        return resultado;
    }
```

O empregado precisa executar outra consulta para carregar as habilidades, mas ele pode facilmente carregar todas as habilidades em uma única consulta. Para fazer isso, ele chama o mapeador de habilidades para carregar os dados de uma habilidade específica.

```
class MapeadorDeEmpregado...

    protected List carregarHabilidades (Long idDoEmpregado) {
            PreparedStatement dec = null;
            ResultSet rs = null;
            try {
                List resultado = new ArrayList( );
                dec = DB.prepare(comandoDeBuscaDeHabilidades);
                dec.setObject(1, idDoEmpregado);
                rs = dec.executeQuery( );
                while (rs.next( )) {
                    Long idDaHabilidade = new Long(rs.getLong(1));
                    resultado.Add( (Habilidade) RegistroMapeador.habilidade( ).carregarLinha
(idDaHabilidade, rs));
            }
            return resultado;
            } catch (SQLException e) {
                throw new ApplicationException(e);
            } finally {DB.cleanUp(dec, rs);
            }
    }
    private static final String comandoDeBuscaDeHabilidades =
        "SELECT habilidade.ID, " + MapeadorDeHabilidade.LISTA_DE_COLUNAS +
        "  FROM habilidades habilidade, empregadoHabilidades eh " +
        "  WHERE eh.idDoEmpregado = ? AND habilidade.ID = eh.idDaHabilidade";

class MapeadorDeHabilidade...

    public static final String LISTA_DE_COLUNAS = " habilidade.nome nomeDaHabilidade ";

class MapeadorAbstrato...

    protected ObjetoDoDomínio carregarLinha (Long id, ResultSet rs) throws SQLException {
        return carregar(id, rs);
    }

class MapeadorDeHabilidade...

    protected ObjetoDoDomínio fazerCarga (Long id, ResultSet rs) throws SQLException {
        Habilidade resultado = new Habilidade (id);
        resultado.gravarNome(rs.getString("nomeDaHabilidade"));
        return resultado;
    }
```

O mapeador abstrato também pode ajudar a encontrar empregados.

```
class MapeadorDeEmpregado...

    public List buscarTodos ( ) {
```

```
        return buscarTodos (comandoBuscarTodos);
    }
    private static final String comandoBuscarTodos =
            "SELECT " + LISTA_DE_COLUNAS +
            "  FROM empregados empregado" +
            "  ORDER BY empregado.sobrenome";

class MapeadorAbstrato...

    protected List buscarTodos (String sql) {
        PreparedStatement dec = null;
        ResultSet rs = null;
        try {
            List resultado = new ArrayList( );
            dec = DB.prepare(sql);
            rs = dec.executeQuery( );
            while (rs.next( ))
                resultado.add(carregar(rs));
            return resultado;
        } catch (SQLException e) {
            throw new ApplicationException(e);
        } finally {DB.cleanUp(dec, rs);}
        }
    }
```


Tudo isso funciona muito bem e é muito simples de acompanhar. Ainda assim, há um problema no número de pesquisas, e isso se deve ao fato de que cada empregado gasta duas consultas SQL para ser carregado. Embora possamos carregar os dados básicos de um empregado para muitos empregados em uma única consulta, ainda precisamos de uma consulta por empregado para carregar as habilidades. Assim, carregar 100 empregados exige 101 consultas.

Exemplo: Usando uma Única Consulta para Vários Empregados (Java)

É possível trazer do banco vários empregados, juntamente com suas habilidades, em uma única busca. Este é um bom exemplo de otimização de consultas em mais de uma tabela, o que certamente é mais complicado. Por esse motivo, faça isso somente quando precisar, em vez de em todas as vezes. É melhor colocar mais energia em acelerar as suas buscas lentas do que em muitas consultas que sejam menos importantes.

O primeiro caso que investigaremos é um caso simples no qual trazemos do banco todas as habilidades de um empregado na mesma consulta que traz os dados básicos. Para fazer isso, usarei um comando SQL mais complexo que realiza a junção das três tabelas.

```
class MapeadorDeEmpregado...

    protected String comandoDeBusca ( ) {
        return
            "SELECT " + LISTA_DE_COLUNAS +
            "  FROM empregados empregado, habilidades habilidade, empregadoHabilidades eh" +
            "  WHERE empregado.ID = eh.idDoEmpregado AND habilidade.ID = eh.idDaHabilidade AND
empregado.ID = ?";
    }
```

```
    public static final String LISTA_DE_COLUNAS =
           " empregado.ID, empregado.sobrenome, empregado.prenome, " +
           "  eh.idDaHabilidade, eh.idDoEmpregado, " +
              "habilidade.ID, habilidade.nome ";
```

Os métodos `buscaAbstrata` e `carregar` da superclasse são os mesmos do exemplo anterior, de modo que não irei repeti-los aqui. O mapeador de empregados carrega seus dados de maneira diferente para tirar proveito das múltiplas linhas de dados.

```
class MapeadorDeEmpregado...

    protected ObjetoDoDomínio fazerCarga (Long id, ResultSet rs) throws SQLException {
        Empregado resultado = (Empregado) carregarLinha (id, rs);
        carregarDadosDasHabilidades (resultado, rs);
        while (rs.next( )) {
            Assert.isTrue(aLinhaÉParaOMesmoEmpregado(id, rs));
            carregarDadosDasHabilidades(resultado, rs);
        }
        return resultado;
    }
    protected ObjetoDoDomínio carregarLinha(Long id, ResultSet rs) throws SQLException {
        Empregado resultado = new Empregado(id);
        resultado.gravarPrenome(rs.getString("prenome");
        resultado.gravarSobrenome(rs.getString("sobrenome");
        return resultado;
    }
    private boolean aLinhaÉParaOMesmoEmpregado (Long id, ResultSet rs) throws SQLException {
        return id.equals(new Long(rs.getLong(1)));
    }
    private void carregarDadosDasHabilidades (Empregado pessoa, ResultSet rs) throws SQLException {
        Long idDaHabilidade = new Long(rs.getLong("idDaHabilidade");
        pessoa.adicionarHabilidade ((Habilidade) RegistroMapeador.habilidade( ).carregarLinha
(idDaHabilidade, rs));
    }
```

Neste caso, o método de carga do mapeador de empregados, em verdade percorre o resto do conjunto resultante para carregar todos os dados.

Tudo é simples quando estamos carregando os dados de um único empregado. Entretanto, o benefício real desta consulta em mais de uma tabela aparece quando queremos carregar muitos empregados. Obter a leitura correta pode ser complicado, especialmente quando não queremos forçar o agrupamento do conjunto resultante por empregados. Neste ponto é útil introduzir uma classe auxiliar para percorrer o conjunto resultante focando na própria tabela associativa, carregando os empregados e habilidades à medida que prossegue.

Começarei com o SQL e a chamada para a classe carregadora especial.

```
class MapeadorDeEmpregado...

    public List buscarTodos ( ) {
        return buscarTodos(comandoBuscarTodos);
    }
    private static final String comandoBuscarTodos =
           "SELECT " + LISTA_DE_COLUNAS +
           "  FROM empregados empregado, habilidades habilidade, empregadoHabilidades eh" +
```

```
            "  WHERE empregado.ID = eh.idDoEmpregado AND habilidade.ID = eh.idDaHabilidade" +
            "  ORDER BY empregado.sobrenome";
    protected List buscarTodos(String sql) {
        CarregadorDaTabelaAssociativa carregador = new CarregadorDaTabelaAssociativa(this, new
AdicionadorDeHabilidade( ));
        return carregador.rodar(comandoBuscarTodos);
    }
```

```
class CarregadorDaTabelaAssociativa...

    private MapeadorAbstrato mapeadorFonte;
    private Adicionador adicionadorAlvo;
    public CarregadorDaTabelaAssociativa (MapeadorAbstrato mapeadorPrincipal, Adicionador
adicionadorAlvo) {
        this.mapeadorFonte = mapeadorPrincipal;
        this.adicionadorAlvo = adicionadorAlvo;
    }
```


Não se preocupe com o `adicionadorDeHabilidade` – mais tarde ele se tornará um pouco mais claro. Por enquanto, perceba que construímos o carregador com uma referência ao mapeador e então o mandamos fazer uma carga com uma consulta apropriada. Essa é a estrutura típica de um objeto método. Um **objeto método** [Beck Patterns] é uma maneira de transformar um método complicado em um objeto por si só. A grande vantagem disso é que lhe permite colocar valores em campos em vez de passá-los por parâmetros. A maneira comum de usar um objeto método é criá-lo, dispará-lo e então deixá-lo morrer assim que o seu trabalho estiver terminado.

O comportamento de carga se dá em três passos.

```
class CarregadorDaTabelaAssociativa...

    protected List rodar (String sql) {
        carregarDados(sql);
        adicionarTodosObjetosNovosAoMapaDeIdentidade( );
        return formarResultado;
    }
```

O método `carregarDados` constrói a chamada SQL, executa-a e efetua um laço no conjunto resultante. Já que este é um objeto método, coloquei o conjunto resultante em um campo de modo que não o tenha de passar por parâmetro.

```
class CarregadorDaTabelaAssociativa...

    private ResultSet rs = null;
    private void carregarDados (String sql) {
        PreparedStatement dec = null;
        try {
            dec = DB.prepare(sql);
            rs = dec.executeQuery( );
            while (rs.next( ))
                carregarLinha( );
        } catch (SQLException e) {
            throw new ApplicationException(e);
        } finally {DB.cleanUp(dec, rs);
        }
    }
```

O método carregarLinha carrega os dados de uma única linha no conjunto resultante. Ele é um pouco complicado.

```
class CarregadorDaTabelaAssociativa...

    private List idsResultantes = new ArrayList( );
    private Map emProgresso = new HashMap( );
    private void carregarLinha ( ) throws SQLException {
        Long ID = new Long(rs.getLong(1));
        if (!idsResultantes.contains(ID)) idsResultantes.add(ID);
        if (!mapeadorFonte.carregou(ID)) {
            if (!emProgresso.keySet( ).contains(ID))
                emProgresso.put(ID, mapeadorFonte.carregarLinha(ID, rs));
            adicionadorAlvo.adicionar( (ObjetoDoDomínio) emProgresso.get(ID), rs);
        }
    }

class MapeadorAbstrato...

    boolean carregou(Long id) {
        return mapaCarregado.containsKey(id);
    }
```


O carregador preserva qualquer ordenação presente no conjunto resultante, de modo que a lista de saída de empregados estará na mesma ordem inicial. Então, mantenho uma lista de IDs na ordem em que os vejo. Assim que tiver o ID, vejo se ele já está totalmente carregado no mapeador – normalmente por uma consulta anterior. Caso não esteja, carrego os dados que tiver e guardo-os em uma lista "em progresso". Preciso de tal lista já que diversas linhas serão combinadas para reunir todos os dados do empregado e posso não chegar a essas linhas consecutivamente.

A parte mais delicada deste código é assegurar que posso adicionar a habilidade que estou carregando na lista de habilidades dos empregados e ainda assim manter genérico o carregador de modo que ele não dependa de empregados e habilidades. Para alcançar isso, preciso procurar fundo no meu baú de truques para encontrar uma interface interna – o Adicionador.

```
class CarregadorDaTabelaAssociativa...

    public static interface Adicionador {
        void adicionar (ObjetoDoDomínio hospedeiro, ResultSet rs) throws SQLException;
    }
```

O solicitante original tem que fornecer uma implementação para a interface, para especializá-la para as necessidades específicas do empregado e habilidade.

```
class MapeadorDeEmpregado...

    private static class AdicionadorDeHabilidade implements CarregadorDaTabelaAssociativa.Adicionador {
        public void adicionar (ObjetoDoDomínio hospedeiro, ResultSet rs) throws SQLException {
            Empregado emp = (Empregado) hospedeiro;
            Long idDaHabilidade = new Long(rs.getLong("idDaHabilidade"));
            emp.adicionarHabilidade( (Habilidade) RegistroMapeador.habilidade( ).carregarLinha
(idDaHabilidade, rs));
        }
    }
```

Este é o tipo de coisa que fica mais natural em linguagens que possuem ponteiros para funções ou fechamentos (*closures*), mas pelo menos a classe e a interface conseguem executar o trabalho. (Elas não têm que ser internas neste caso, mas isso ajuda a enfatizar seu escopo estreito.)

Você pode ter percebido que tenho um método `carregar` e um método `carregarLinha` definidos na superclasse e a implementação de `carregarLinha` é uma chamada ao método `carregar`. Fiz isso porque existem ocasiões em que você quer ter certeza de que uma ação de carga não avançará o conjunto resultante. A carga faz o que ela tiver de fazer para carregar um objeto, mas `carregarLinha` garante a carga de dados de uma linha sem alterar a posição do cursor. Na maior parte do tempo, esses dois métodos são a mesma coisa, mas no caso deste mapeador de empregados, eles são diferentes.

Agora todos os dados vieram do conjunto resultante. Tenho duas coleções: uma lista dos IDs de todos os empregados que estavam no conjunto resultante, na ordem da sua aparição inicial e uma lista de novos objetos que ainda não haviam aparecido no *Mapa de Identidade* (196) do mapeador de empregados.

O próximo passo é colocar todos os novos objetos no *Mapa de Identidade* (196).

```
class CarregadorDaTabelaAssociativa...

    private void adicionarTodosObjetosNovosAoMapaDeIdentidade ( ) {
        for (Iterator it = emProgresso.values( ).iterator( ); it.hasNext( );)
            mapeadorFonte.colocarComoCarregado((ObjetoDoDomínio) it.next( ));
    }

class MapeadorAbstrato...

    void colocarComoCarregado (ObjetoDoDomínio obj) {
        mapaCarregado.put (obj.lerID( ), obj);
    }
```

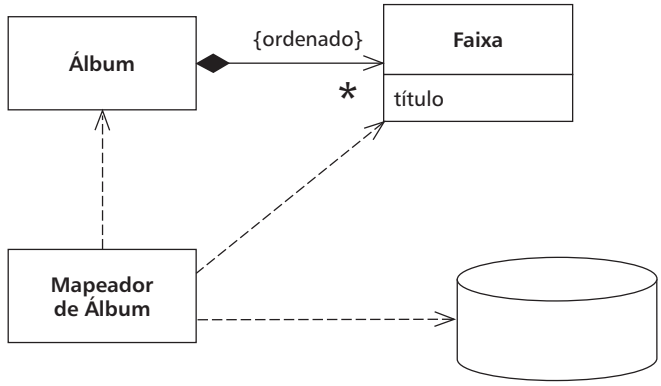
O passo final é montar a lista resultante procurando os IDs no mapeador.

```
class CarregadorDaTabelaAssociativa...

    private List formarResultado( ) {
        List resultado = new ArrayList( );
        for ( Iterator it = idsResultantes.iterator( ); it.hasNext( );) {
            Long id = (Long) it.next( );
            resultado.add(mapeadorFonte.procurar(id));
        }
        return resultado;
    }

class MapeadorAbstrato...

    protected ObjetoDoDomínio procurar (Long id) {
        return (ObjetoDoDomínio) mapaCarregado.get(id);
    }
```

Tal código é mais complexo do que o código de carga usual, mas esse tipo de coisa pode ajudar a diminuir o número de consultas. Já que é complicado, isso é algo a ser usado com parcimônia, quando você tiver partes lentas de interação com o banco de dados. Entretanto, é um exemplo de como o *Mapeador de Dados* (170) pode fornecer boas consultas sem que a camada do domínio tome conhecimento da complexidade envolvida.

Mapeamento Dependente (Dependent Mapping)

Faz uma classe executar o mapeamento do banco de dados para uma classe filha.



Alguns objetos aparecem naturalmente no contexto de outros objetos. As faixas de um álbum podem ser carregadas ou gravadas sempre que o álbum correspondente for carregado ou gravado. Se elas não forem referenciadas por nenhuma outra tabela no banco de dados, você pode simplificar o procedimento de mapeamento fazendo o mapeador do álbum executar também o mapeamento das faixas – ou seja, tratando este mapeamento como um *mapeamento dependente*.

Como Funciona

A idéia básica por trás do *Mapeamento Dependente* é que uma classe (a **dependente**) depende de alguma outra (a **proprietária**) para a sua persistência no banco de dados. Cada dependente deve ter uma e apenas uma proprietária.

Isso se manifesta em termos das classes que executam o mapeamento. Para o *Registro Ativo* (165) e o *Gateway de Linhas de Dados* (158), a classe dependente não conterá qualquer código de mapeamento para o banco de dados. Seu código de mapeamento reside na proprietária. Com o *Mapeador de Dados* (170) não há um mapeador para a dependente. O código de mapeamento reside no mapeador da proprietária. Em um *Gateway de Tabelas de Dados* (151), tipicamente não haverá nenhuma classe dependente, todo tratamento da dependente é feito na proprietária.

Na maioria dos casos, cada vez que você carrega uma proprietária, você também carrega as dependentes. Se for custoso carregar as dependentes e elas não são usadas com freqüência, você pode usar uma *Carga Tardia* (200) para evitar carregar as dependentes até que você precise delas.

Uma propriedade importante de uma dependente é que ela não tem um *Campo Identidade* (215) e, portanto, ela não é armazenada em um *Mapa de Identidade* (196). Por conseguinte, ela não pode ser carregada por um método de busca que procure um ID. De fato, não existe um método de busca para uma dependente, já que todas as buscas são feitas com a proprietária.

A própria dependente pode ser a proprietária de outra dependente. Neste caso, a proprietária da primeira dependente é também responsável pela persistência da se-

gunda dependente. Você pode ter toda uma hierarquia de dependentes controlada por uma única classe proprietária.

Normalmente é mais fácil para a chave primária no banco de dados ser uma chave composta que inclua a chave primária da proprietária. Nenhuma outra tabela deveria ter uma chave estrangeira apontando para a tabela da dependente, a menos que que esse objeto tenha a mesma classe proprietária. O resultado é que nenhum objeto na memória além da proprietária ou suas dependentes deve ter uma referência para uma dependente. A rigor, você pode relaxar essa regra desde que a referência não seja persistida no banco de dados, no entanto, uma referência não persistente é por si mesma uma boa fonte de confusão.

Em um modelo UML, é apropriado usar composição para mostrar o relacionamento entre uma proprietária e suas dependentes.

Já que a escrita e a gravação das dependentes são deixadas a cargo da proprietária e não há referências externas, as atualizações nas dependentes podem ser tratadas por meio de exclusões e inserções. Assim, se você quiser atualizar a coleção de dependentes, você pode seguramente excluir todas as linhas que apontam para a proprietária e então inserir novamente todas as dependentes. Isso evita que você tenha de fazer uma análise dos objetos adicionados ou removidos da coleção da proprietária.

As dependentes são, de muitas formas, parecidas com *Objetos Valor* (453), ainda que, freqüentemente, elas não precisem de todos os mecanismos que você usa para tornar algo em um *Objeto Valor* (453) (tal como sobrescrever o método *equals*). A principal diferença é que não há nelas, estritamente no que diz respeito à memória, nada de especial. A natureza dependente dos objetos, em verdade, deve-se apenas ao comportamento de mapeamento para o banco de dados.

O uso do *Mapeamento Dependente* torna mais complicado descobrir se a proprietária foi alterada. Qualquer alteração em uma dependente precisa marcar a proprietária como alterada a fim de que a proprietária grave as alterações no banco de dados. Você pode simplificar isso consideravelmente tornando a dependente imutável, a fim de que qualquer alteração nela tenha de ser feita removendo-a e adicionando uma nova. Isso pode tornar o modelo na memória mais difícil de trabalhar, mas simplifica o mapeamento para o banco de dados. Embora, em teoria, o mapeamento em memória e o mapeamento no banco de dados devessem ser independentes quando você está usando um *Mapeador de Dados* (170), na prática você tem que, ocasionalmente, aderir a essa solução conciliatória.

Quando Usá-lo

Você usa um *Mapeamento Dependente* quando você tem um objeto que é referenciado apenas por um único objeto, o que normalmente ocorre quando um objeto tem uma coleção de dependentes. O *Mapeamento Dependente* é uma boa forma de lidar com a situação complicada em que a classe proprietária tem uma coleção de referências para as suas dependentes, mas não há um ponteiro reverso. Desde que os muitos objetos não precisem de suas próprias identidades, o uso do *Mapeamento Dependente* torna mais fácil gerenciar sua persistência.

Para que o *Mapeamento Dependente* funcione, existem várias pré-condições que devem ser satisfeitas.

- Uma classe dependente deve ter uma e apenas uma proprietária.

- Nenhum outro objeto, com exceção da classe proprietária, deve referenciar as dependentes.

Há uma corrente de projeto OO que usa a noção de objetos entidades e objetos dependentes ao projetar um *Modelo de Domínio* (126). Tendo a pensar no *Mapeamento Dependente* como uma técnica para simplificar o mapeamento para o banco de dados em vez de como uma parte essencial do projeto OO. Em particular, evito grandes grafos de dependentes. O problema com eles é que é impossível referenciar uma dependente de fora do grafo, o que muitas vezes leva a complexos esquemas de busca baseados na proprietária raiz.

Não recomendo o *Mapeamento Dependente* se você estiver usando uma *Unidade de Trabalho* (187). A estratégia de remoção e reinserção absolutamente não ajuda se você tiver uma *Unidade de Trabalho* (187) cuidando das coisas. Também pode levar a problemas, uma vez que a *Unidade de Trabalho* (187) não estará controlando as dependentes. Mike Rettig contou-me a respeito de uma aplicação em que uma *Unidade de Trabalho* (187) cuidaria das linhas inseridas para teste e então as excluiria ao terminar. Devido a ela não manter registro das dependentes, linhas órfãs apareceram e causaram falhas nos testes.

Exemplo: Álbuns e Faixas (Java)

Neste modelo de domínio (Figura 12.7), um álbum mantém uma coleção de faixas. Esta quase que inútil aplicação simples não precisa de mais nada para referenciar uma faixa, de modo que é uma candidata óbvia para um *Mapeamento Dependente*. (De fato, qualquer um pensaria que o exemplo foi deliberadamente criado para o padrão.)

Esta faixa tem apenas o atributo título. Eu a defini como uma classe imutável.

```
class Faixa...

    private final String título;
    public Faixa (String título) {
        this.título = título;
    }
    public String lerTítulo ( ) {
        return título;
    }
```

As faixas são armazenadas na classe álbum.

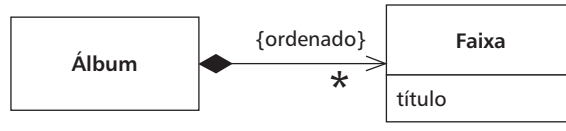


Figura 12.7 Um álbum com faixas que podem ser manipuladas usando um *Mapeamento Dependente*.

```
class Álbum...

    private List faixas = new ArrayList( );
    public void adicionarFaixa (Faixa arg) {
        faixas.add (arg);
    }
    public void removerFaixa (Faixa arg) {
        faixas.remove (arg);
    };
    public void removerFaixa(int i) {
        faixas.remove(i);
    }
    public Faixa[ ] lerFaixas ( ) {
        return (Faixa[ ]) faixas.toArray(new Faixa[faixas.size( )]);
    }
```

A classe mapeadora de álbuns manipula todo o SQL para as faixas e, portanto, define os comandos SQL que acessam a tabela de faixas.

```
class MapeadorDeÁlbum...

    protected String comandoDeBusca ( ) {
        return
            "SELECT ID, a.título, f.título as tituloDaFaixa"+
            "  FROM álbuns a, faixas f"+
            "  WHERE a.ID = ? AND f.idDoÁlbum = a.ID" +
            "  ORDER BY f.seq";
    }
```

As faixas são carregadas no álbum sempre que o álbum for carregado.

```
class MapeadorDeÁlbum...

    protected ObjetoDoDomínio fazerCarga (Long id, ResultSet rs) throws SQLException {
        String título = rs.getString(2);
        Álbum resultado = new Álbum(id, título);
        carregarFaixas(resultado, rs);
        return resultado;
    }
    public void carregarFaixas (Álbum arg, ResultSet rs) throws SQLException {
        arg.adicionarFaixa (novaFaixa(rs));
        while (rs.next( )){
            arg.adicionarFaixa(novaFaixa(rs));
        }
    }
    private Faixa novaFaixa (ResultSet rs) throws SQLException {
        String título = rs.getString(3);
        Faixa novaFaixa = new Faixa (título);
        return novaFaixa;
    }
```

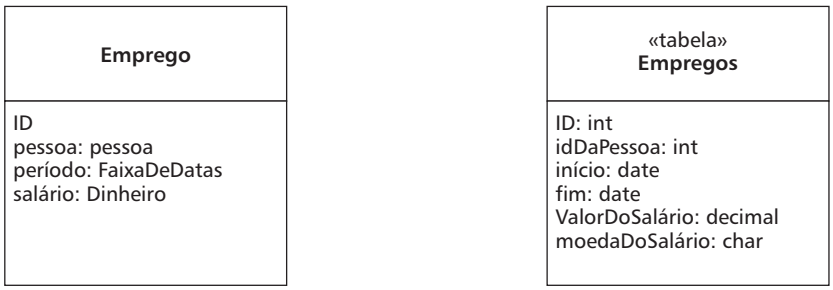
Para aumentar a clareza, executei a carga de faixas em uma consulta separada. Por questões de desempenho, você poderia considerar carregá-las na mesma consulta de acordo com a linha seguida no exemplo da página 239-240.

Quando o álbum é atualizado, todas as faixas são apagadas e reinseridas.

```
class MapeadorDeÁlbum...

    public void atualizar (ObjetoDoDomínio arg) {
        PreparedStatement comandoDeAtualização = null;
        try {
            comandoDeAtualização = DB.prepare("UPDATE álbuns SET título = ? WHERE id = ?");
            comandoDeAtualização.setLong(2, arg.lerID( ).longValue( ));
            Álbum álbum = (Álbum) arg;
            comandoDeAtualização.setString(1, álbum.lerTítulo( ));
            comandoDeAtualização.execute( );
            atualizarFaixas(álbum);
        } catch (SQLException e){
            throw new ApplicationException(e);
        } finally {DB.cleanUp(comandoDeAtualização);
        }
    }
    public void atualizarFaixas (Álbum arg) throws SQLException {
        PreparedStatement comandoExcluirFaixas = null;
        try {
            comandoExcluirFaixas = DB.prepare("DELETE FROM faixas WHERE idDoÁlbum = ?");
            comandoExcluirFaixas.setLong(1, arg.lerID( ).longValue( ));
            comandoExcluirFaixas.execute( );
            for (int i = 0; i < arg.lerFaixas( ).length; i++) {
                Faixa faixa = arg.lerFaixas( )[i];
                inserirFaixa(faixa, i+1, arg);
            }
        } finally {DB.cleanUp(comandoExcluirFaixas);
        }
    }
    public void inserirFaixa (Faixa faixa, int seq, Álbum álbum) throws SQLException {
        PreparedStatement comandoInserirFaixas = null;
        try {
            comandoInserirFaixas =
                DB.prepare("INSERT INTO faixas (seq, idDoÁlbum, título) VALUES (?, ?, ?)");
            comandoInserirFaixas.setInt(1, seq);
            comandoInserirFaixas.setLong(2, álbum.lerID( ).longValue( ));
            comandoInserirFaixas.setString(3, faixa.lerTítulo( ));
            comandoInserirFaixas.execute( );
        } finally {DB.cleanUp(comandoInserirFaixas);
        }
    }
```

Valor Embutido (Embedded Value)

Mapeia um objeto em diversos campos da tabela de um outro objeto.



Muitos objetos pequenos que fazem sentido em um sistema OO não fazem sentido como tabelas em um banco de dados. Exemplos incluem objetos dinheiro que conhecem sua moeda e faixas de datas. Embora o pensamento padrão seja gravar um objeto como uma tabela, nenhuma pessoa sensata iria querer uma tabela de valores monetários.

Um *Valor Embutido* mapeia os valores de um objeto em campos do registro do proprietário do objeto. No desenho temos um objeto emprego com referências para um objeto faixa de datas e um objeto dinheiro. Na tabela resultante, os campos referentes àqueles objetos são mapeados para campos na tabela empregos em vez de eles mesmos, criarem novos registros.

Como Funciona

Este exercício é, em verdade, muito simples. Quando o objeto proprietário (emprego) é carregado ou gravado, os objetos dependentes (faixa de datas e dinheiro) são carregados e gravados ao mesmo tempo. As classes dependentes não terão seus próprios métodos de persistência já que toda a persistência é feita pelo proprietário. Você pode pensar no *Valor Embutido* como um caso especial do *Mapeamento Dependente* (256), em que o valor é um único objeto dependente.

Quando Usá-lo

Este é um dos padrões no qual a execução é bastante direta, mas saber quando usá-lo é um pouco mais complicado.

Os casos mais simples de *Valor Embutido* são os *Objetos Valor* (453) claros e simples, como dinheiro e faixa de datas. Uma vez que os *Objetos Valor* (453) não têm identidade, você pode criá-los e destruí-los facilmente sem se preocupar com coisas como *Mapas de Identidade* (196) para mantê-los todos em sincronismo. De fato, todos os *Objetos Valor* (453) deveriam ser persistidos como um *Valor Embutido*, já que você nunca iria querer uma tabela para eles.

Uma questão nebulosa é se vale a pena armazenar objetos de referência, tal como um pedido e uma remessa, usando um *Valor Embutido*. A principal questão aqui é se os dados da remessa têm alguma relevância fora do contexto do pedido. Uma questão é a carga e a gravação. Se você só carregar os dados da remessa na memória

quando você carregar o pedido, este é um argumento para gravar ambos na mesma tabela. Uma outra questão é se você vai querer acessar os dados da remessa separadamente por meio de SQL. Isso pode ser importante se estiver fazendo relatórios por meio de SQL e não tiver um banco de dados separado para relatórios.

Se você estiver mapeando para um esquema existente, você pode usar um *Valor Embutido* quando uma tabela contiver dados que você divide em mais de um objeto na memória. Isso pode ocorrer porque você quer um objeto separado para fatorar algum comportamento no modelo de objetos, mas, no banco de dados, ele ainda é uma única entidade. Neste caso, você tem que tomar cuidado para que qualquer alteração nos dependentes marque o proprietário como sujo – o que não é um problema com *Objetos Valor* (453) que são substituídos no proprietário.

Na maioria dos casos, você somente usará um *Valor Embutido* em um objeto de referência quando a associação entre eles tiver um único valor em ambas as extremidades (uma associação um-para-um). Ocasionalmente você pode usá-lo se houver vários candidatos a dependentes e seu número for pequeno e fixo. Neste caso, você terá campos numerados para cada valor. Isso é um projeto confuso de tabelas, e muito ruim para consultar com SQL, mas pode trazer benefícios no que se refere ao desempenho. Se, entretanto, este for o caso, o *LOB Serializado* (264) é normalmente a melhor escolha.

Uma vez que muito da lógica para decidir quando usar um *Valor Embutido* é a mesma para um *LOB Serializado* (264), existe a questão óbvia de escolher qual dos dois usar. A grande vantagem do *Valor Embutido* é que ele permite que sejam feitas consultas SQL nos valores do objeto dependente. Ainda que o uso de XML como técnica de serialização, junto com extensões do SQL que permitam consultas baseadas em XML, possa alterar isso no futuro, no momento você realmente precisa de um *Valor Embutido* se quiser usar valores dependentes em uma consulta. Isso pode ser importante para mecanismos de relatórios separados no banco de dados.

Um *Valor Embutido* só pode ser usado para dependentes razoavelmente simples. Ele funciona bem para um único dependente solitário, ou para alguns poucos dependentes separados. O *LOB Serializado* (264) funciona melhor com estruturas mais complexas, incluindo subgrafos de objetos potencialmente grandes.

Leitura Adicional

O *Valor Embutido*, em sua história, tem sido chamado por vários nomes diferentes. O TOPLink se refere a ele como *mapeamento agregado*. Visual Age se refere a ele como *compositor*.

Exemplo: Objeto Valor Simples (Java)

Este é o exemplo clássico de um objeto valor mapeado com um *Valor Embutido*. Começaremos com uma classe simples de oferta de produto com os seguintes campos.

```
class OfertaDeProduto...

    private Produto produto;
    private Dinheiro custoBásico;
    private Integer ID;
```

Nesse campos o ID é um *Campo Identidade* (215), e o produto é um mapeamento de registros comum. Mapearemos o custo básico usando um *Valor Embutido*. Para manter as coisas simples, faremos o mapeamento global com um *Registro Ativo* (165).

Uma vez que estamos usando um *Registro Ativo* (165), precisamos de procedimentos de gravação e carga. Estes procedimentos simples estão na classe de oferta de produtos porque ela é a proprietária. A classe dinheiro não tem nenhum comportamento de persistência. Aqui está o método de carga.

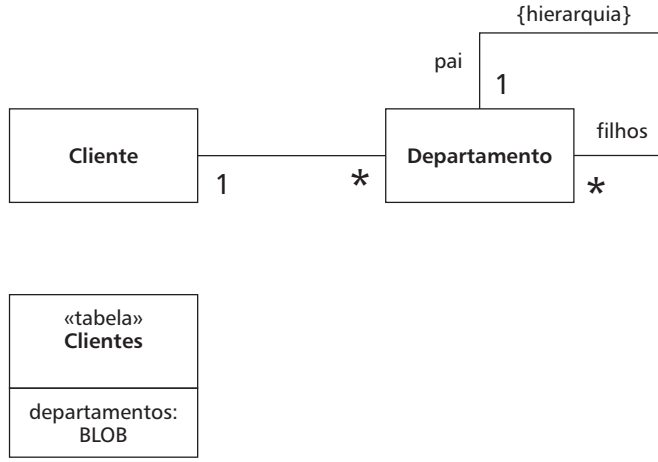
```
class OfertaDeProduto...

    public static OfertaDeProduto carregar (ResultSet rs) {
        try {
            Integer id = (Integer) rs.getObject("ID");
            BigDecimal valorCustoBásico = rs.getBigDecimal("valor_custo_básico");
            Currency moedaCustoBásico =
            Registro.lerMoeda(rs.getString("moeda_custo_básico"));
            Dinheiro custoBásico = new Dinheiro (valorCustoBásico, moedaCustoBásico);
            Integer idDoProduto = (Integer) rs.getObject ("produto");
            Produto produto = Produto.buscar( (Integer) rs.getObject("produto"));
                return new OfertaDeProduto(id, produto, custoBásico);
        } catch (SQLException e){
            throw new ApplicationException(e);
        }
    }
```

Aqui está o comportamento de atualização. Mais uma vez, é uma simples variação das atualizações.

```
class OfertaDeProduto...

    public void atualizar ( ) {
        PreparedStatement dec = null;
        try {
            dec = DB.prepare(stringDoComandoDeAtualização);
            dec.setBigDecimal(1, custoBásico.quantia( ));
            dec.setString(2, custoBásico.moeda( ).código( ));
            dec.setInt(3, ID.intValue( ));
            dec.execute( );
        } catch (Exception e) {
            throw new ApplicationException(e);
        } finally {DB.cleanUp(dec);}
    }
    private String stringDoComandoDeAtualização =
            "UPDATE oferta_produtos" +
            "  SET valor_custo_básico = ?, moeda_custo_básico = ? " +
            "  WHERE id = ?";
```

LOB Serializado (Serialized LOB)

Grava um grafo de objetos serializando-os em um único objeto grande (LOB – Large OBject), o qual ele armazena em um campo do banco de dados.



Os modelos de objetos muitas vezes contêm grafos complicados de pequenos objetos. A maior parte da informação nestas estruturas não está nos objetos, mas nas associações entre eles. Considere o armazenamento da hierarquia de organização de todos os seus clientes. Um modelo de objetos muito naturalmente apresenta o padrão composição para representar hierarquias organizacionais, e você pode facilmente adicionar métodos que lhe permitem obter ancestrais, irmãos, descendentes e outros relacionamentos comuns.

Não é tão fácil colocar tudo isso em um esquema relacional. O esquema básico é simples – uma tabela organização com uma chave estrangeira origem, entretanto, sua manipulação do esquema requer muitas junções, o que é lento e complicado.

Os objetos não têm que ser persistidos como linhas de tabelas relacionadas umas às outras. Outra forma de persistência é a serialização, onde todo um grafo de objetos é gravado como um único objeto grande (LOB) em uma tabela. Este LOB Serializado então se torna uma forma de memento [Gang of Four].

Como Funciona

Há duas maneiras por meio das quais você pode executar a serialização: como um binário (**BLOB**) ou como caracteres texto (**CLOB**). O BLOB é, freqüentemente, o mais simples de criar, uma vez que muitas plataformas incluem a habilidade de serializar automaticamente um grafo de objetos. Gravar o grafo é uma simples questão de aplicar a serialização em um *buffer* e gravar esse *buffer* no campo pertinente.

As vantagens do BLOB são que ele é simples de programar (se a sua plataforma suportá-lo) e que ele usa o mínimo de espaço. As desvantagens são que o seu banco de dados deve suportar um tipo binário de dados para ele e que você não pode reconstruir o gráfico sem o objeto, de modo que o campo é completamente impenetrável a uma passada de olhos. O problema mais sério, contudo, é com o controle de versões. Se você alterar a classe departamento, você pode não conseguir ler todas as

suas serializações anteriores. Uma vez que os dados podem residir no banco de dados por um longo período, a quantidade de dados pode não ser pequena.

A alternativa é um CLOB. Neste caso, você serializa o grafo do departamento em uma *string* de texto que carrega toda a informação de que você precisa. Esta *string* pode ser lida facilmente por um humano vendo a linha, o que ajuda em uma passada de olhos pelo banco de dados. Todavia a abordagem do texto normalmente precisará de mais espaço, e você pode precisar criar seu próprio analisador (*parser*) para o formato textual que você estiver usando. Também é provável que ele seja mais lento do que uma serialização binária.

Muitas das desvantagens dos CLOBs podem ser superadas com o uso de XML. Vários analisadores XML estão disponíveis como *software* livre, de modo que você não precisa escrever o seu próprio. Além disso, XML é um padrão amplamente suportado, de modo que, para fazer manipulações adicionais, você pode tirar proveito das ferramentas à medida que elas se tornam disponíveis. A desvantagem que o XML não resolve é a questão do espaço. De fato, ele torna a questão do espaço muito pior porque ele é um formato bastante prolixo. Uma maneira de lidar com isso é usar um XML zipado como o seu BLOB – você perde a legibilidade humana direta, mas é uma opção, se o espaço for realmente um problema.

Quando você usa um *LOB Serializado*, cuidado com problemas de identidade. Digamos que você queira usar um *LOB Serializado* para armazenar os detalhes do cliente em um pedido. Para isso, não coloque o LOB do cliente na tabela de pedidos, caso contrário os dados do cliente serão copiados a cada pedido, o que torna a atualização um problema. (Isso pode ser uma coisa boa, entretanto, se você quiser armazenar um instantâneo dos dados do cliente como eles eram no instante da colocação do pedido – isso evita relacionamentos temporais.) Se você quiser que os dados do seu cliente sejam atualizados em cada pedido no sentido relacional clássico, precisa colocar o LOB em uma tabela de clientes de modo que muitos pedidos podem ser associados a ela. Não há nada de errado com uma tabela que tenha apenas um ID e um único campo LOB como seus dados.

De um modo geral, tenha cuidado com a duplicação de dados ao usar este padrão. Freqüentemente, não é todo um *LOB Serializado* que é duplicado, mas parte de um que se sobrepõe com parte de outro. O que se tem a fazer é prestar muita atenção aos dados que são armazenados no *LOB Serializado* e estar certo de que eles não possam ser alcançados de nenhum outro lugar, mas apenas de um único objeto que atua como o proprietário do *LOB Serializado*.

Quando Usá-lo

O *LOB Serializado* não é considerado tão freqüentemente quanto poderia. XML o torna muito mais atrativo, uma vez que ele fornece uma abordagem textual fácil de implementar. Sua principal desvantagem é que você não pode consultar a estrutura usando SQL. Algumas extensões SQL que têm surgido se propõem a alcançar os dados XML dentro de um campo, mas ainda não é a mesma coisa (ou portável).

Este padrão funciona melhor quando você pode extrair um pedaço do modelo de objetos e usá-lo para representar o LOB. Pense em um LOB como uma maneira de pegar um grupo de objetos que provavelmente não serão pesquisados de fora da aplicação usando SQL. Este grafo pode então ser enganchado no esquema SQL.

Um *LOB Serializado* tem um desempenho sofrível quando objetos fora do LOB referenciam objetos enterrados dentro dele. Para lidar com esse caso você tem que descobrir algum tipo de esquema de referências que suporte referências a objetos

dentro de um LOB – não é de forma alguma impossível, mas é complicado o suficiente para geralmente não valer a pena. Mais uma vez XML, ou mais exatamente XPath, reduz um pouco esta complicação.

Se você estiver usando um banco de dados separado para os relatórios, e todas as outras consultas SQL forem executadas sobre esse banco de dados, você pode transformar o LOB em uma estrutura de tabela apropriada. O fato de que um banco de dados para relatórios é normalmente não-normalizado significa que as estruturas apropriadas para um *LOB Serializado* são freqüentemente apropriadas também para um banco de dados separado para relatórios.

Exemplo: Serializando uma Hierarquia de Departamentos em XML (Java)

Para este exemplo, pegaremos a noção de clientes e departamentos do desenho e mostraremos como você poderia serializar todos os departamentos em um CLOB XML. No momento que escrevo isto, a manipulação Java de XML é um tanto primitiva e volátil, de modo que, quando você estiver lendo este código, ele poderá parecer um pouco diferente daquilo a que você está habituado (também estou usando uma versão antiga de JDOM).

O modelo de objetos do esboço dá origem às seguintes estruturas de classe:

```
class Cliente...

    private String nome;
    private List departamentos = new ArrayList( );

class Departamento...

    private String nome;
    private List subsidiárias = new ArrayList( );
```

O banco de dados para isso tem apenas uma tabela.

```
create table clientes (ID int primary key, nome varchar, departamentos varchar)
```

Trataremos o cliente como um *Registro Ativo* (165) e ilustraremos a gravação dos dados com o comportamento de inserção.

```
class Cliente...

    public Long inserir ( ) {
        PreparedStatement comandoDeInserção = null;
        try {
            comandoDeInserção = DB.prepare(stringDoComandoDeInserção);
            gravarID (encontrarOPróximoIDdoBancoDeDados( ));
            comandoDeInserção.setInt(1, lerID( ).intValue( ));
            comandoDeInserção.setString(2, nome);
            comandoDeInserção.setString(3, XmlStringer.write(departamentosParaElementoXml( )));
            comandoDeInserção.execute( );
            Registro.adicionarCliente(this);
            return lerID( );
        } catch (SQLException e) {
            throw new ApplicationException(e);
        } finally {DB.cleanUp(comandoDeInserção);
        }
```

```
    }
    public Element departamentosParaElementoXml ( ) {
        Element raiz = new Element("listaDeDepartamentos");
        Iterator i = departamentos.iterator( );
        while (i.hasNext( )) {
            Departamento dep = (Departamento) i.next( );
            raiz.addContent(dep.paraElementoXml( ));
        }
        return raiz;
    }
```

```
class Departamento...

    Element paraElementoXml ( ) {
        Elemento raiz = new Element("Departamento");
        raiz.setAttribute("nome", nome);
        Iterator i = subsidiárias.iterator( );
        while (i.hasNext( )) {
            Departamento dep = (Departamento) i.next( );
            raiz.addContent(dep.paraElementoXml( ));
        }
        return raiz;
    }
```


O cliente tem um método para serializar seu campo departamentos em um único DOM XML. Cada departamento tem um método para serializar a si próprio (e recursivamente às suas subsidiárias) também em um DOM. O método de inserção então pega o DOM dos departamentos, converte-o em uma *string* (por meio de uma classe utilitária) e a coloca no banco de dados. Não estamos particularmente preocupados com a estrutura da *string*. Ela é legível para humanos, mas não iremos olhar para ela regularmente.

```
    <?xml version="1.0" encoding="UTF-8"?>
    <listaDeDepartamentos>
        <departamento nome="US">
            <departamento nome="New England">
                <departamento nome="Boston /">
                <departamento nome="Vermont /">
            </departamento>
            <departamento nome="California /">
            <departamento nome="Mid-West /">
        </departamento>
        <departamento nome="Europe" />
    </listaDeDepartamentos>
```

Ler os dados de volta é razoavelmente simples. Basicamente é o reverso desse processo.

```
class Cliente...

    public static Cliente carregar (ResultSet rs) throws SQLException {
        Long id = new Long(rs.getLong("id"));
        Cliente resultado = (Cliente) Registro.lerCliente(id);
        if (resultado != null) return resultado;
        String nome = rs.getString("nome");
```

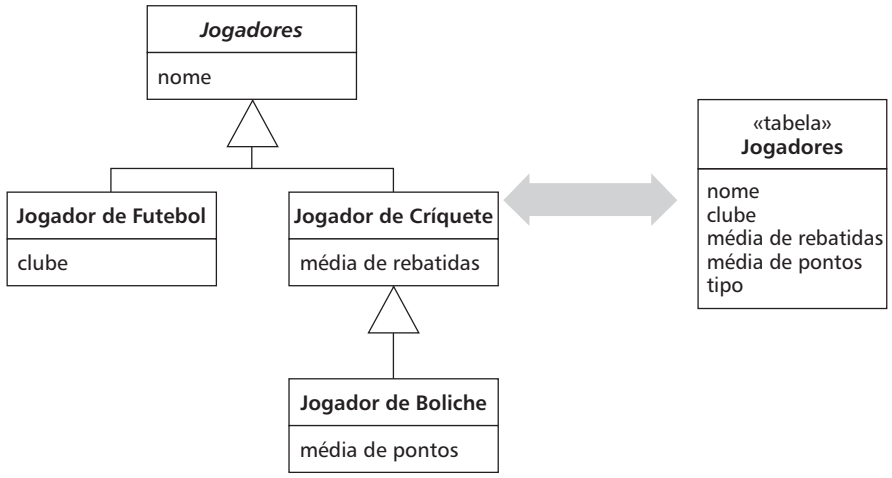
```
        String lobDepartamento = rs.getString("departamentos");
        resultado = new Cliente(nome);
        resultado.lerDepartamentos(XmlStringer.read(lobDepartamento));
        return resultado;
    }
    void lerDepartamentos (Element fonte) {
        List resultado = new ArrayList( );
        Iterator it = fonte.getChildren("departamento").iterator( );
        while (it.hasNext( ))
        adicionarDepartamento(Departamento.lerXml((Elemento) it.next( )));
    }
class Departamento...
    static Departamento lerXml (Element fonte) {
        String nome = fonte.getAttributeValue("nome");
        Departamento resultado = new Departamento(nome);
        Iterator it = fonte.getChildren("departamento").iterator( );
        while (it.hasNext( ))
            resultado.adicionarSubsidiária(lerXml((Element) it.next( )));
        return resultado;
    }
```


O código de carga é obviamente uma imagem espelhada do código de inserção. O departamento sabe como criar a si próprio (e suas subsidiárias) a partir de um elemento XML, e o cliente sabe como pegar em elemento XML e criar a lista de departamentos a partir dele. O método de carga usa uma classe utilitária para transformar a string do banco de dados em um elemento utilitário.

Um perigo óbvio aqui é que alguém pode tentar editar à mão o XML no banco de dados e estragar o XML, tornando-o ilegível para o procedimento de leitura. Ferramentas mais sofisticadas que suportassem a adição de um campo contendo um DTD ou esquema XML com o propósito de validação obviamente ajudaria.

Herança de Tabela Única (Single Table Inheritance)

Representa uma hierarquia de herança de classes como uma única tabela que tem colunas para todos os campos das diversas classes da hierarquia.



Os bancos de dados relacionais não suportam herança, então ao mapear de objetos para os bancos de dados, temos que considerar como representar nossas belas estruturas de herança como tabelas relacionais. Ao mapear para um banco de dados relacional, tentamos minimizar as junções (*joins*) que podem crescer rapidamente ao processar uma estrutura de herança em diversas tabelas. A Herança de Tabela Única mapeia todos os campos de todas as classes de uma estrutura de herança em uma única tabela.

Como Funciona

Neste esquema de mapeamento de herança, temos uma única tabela que contém todos os dados de todas as classes da hierarquia de herança. Cada classe armazena os dados que sejam relevantes para ela em uma única linha da tabela. Quaisquer colunas no banco de dados que não sejam relevantes são deixadas em branco. O comportamento básico de mapeamento segue o esquema geral dos *Mapeadores de Herança* (291).

Ao carregar um objeto na memória, você precisa saber qual classe instanciar. Para isso você tem um campo na tabela que indica qual classe deveria ser usada. Este pode ser o nome da classe ou um campo com um código. Um campo codificado precisa ser interpretado por algum código para mapeá-lo para a classe pertinente. Este código precisa ser estendido quando uma nova classe é adicionada à hierarquia. Se você embutir o nome da classe na tabela, você pode usá-lo diretamente para instanciar uma instância da classe. O nome da classe, entretanto, demandará mais espaço e pode ser menos fácil de ser processado por aqueles usando diretamente a estrutura da tabela do banco de dados. Além disso, o nome da classe pode acoplar mair fortemente a estrutura da classe ao esquema do banco de dados.

Ao carregar os dados, você primeiro lê o código para descobrir qual subclasse instanciar. Ao salvar os dados, o código precisa ser gravado pela superclasse na hierarquia.

Quando Usá-la

A *Herança de Tabela Única* é uma das opções para mapear os campos de uma hierarquia de herança para um banco de dados relacional. As alternativas são a *Herança de Tabela de Classes* (276) e a *Herança de Tabela Concreta* (283).

Estes são os pontos fortes da *Herança de Tabela Única*:

- Só há uma única tabela com a qual se preocupar no banco de dados.
- Não há junções na recuperação dos dados.
- Qualquer refatoração que mova campos para cima ou para baixo na hierarquia não requer que você altere o banco de dados.

Os pontos fracos da *Herança de Tabela Única* são:

- Algumas vezes os campos são relevantes, algumas vezes não, o que pode ser confuso para as pessoas usando diretamente as tabelas.
- As colunas usadas apenas por algumas subclasses levam a um desperdício de espaço no banco de dados. O quanto isso é realmente um problema depende das características específicas dos dados e o quão bem o banco de dados comprime as colunas vazias. O Oracle, por exemplo, é muito eficiente em eliminar espaço desperdiçado, especialmente se você mantiver suas colunas opcionais no lado direito da tabela do banco de dados. Cada banco de dados tem os seus próprios artifícios para fazer isso.
- A tabela única pode acabar ficando grande demais, com muitos índices e com bloqueios freqüentes, o que pode prejudicar o desempenho. Você pode evitar isso tendo tabelas de índices separadas que listem as chaves das linhas que tenham uma certa propriedade ou que copiem um subconjunto de campos relevantes para um índice.
- Você tem um único espaço de nomes (*namespace*) para os campos, de modo que você tem de estar certo de que não usa o mesmo nome para diferentes campos na hierarquia. Nomes compostos, com o nome da classe como prefixo ou sufixo, ajudam aqui.

Lembre-se de que você não precisa usar uma única forma de mapeamento de herança para toda a sua hierarquia. É perfeitamente correto mapear meia dúzia de classes similares em uma única tabela, desde que você use a *Herança de Tabela Concreta* (283) para quaisquer classes que tenham uma quantidade grande de dados específicos.

Exemplo: Uma Tabela Única para Jogadores (C#)

Como nos outros exemplos de herança, baseei este aqui nos *Mapeadores de Herança* (291), usando as classes da Figura 12.8. Cada mapeador precisa ser associado a uma tabela de dados em um conjunto de dados ADO.NET. Esta associação pode ser feita genericamente na superclasse mapeadora. A propriedade dos dados do *gateway* é um conjunto de dados que pode ser carregado por uma consulta.

```
class Mapeador...

    protected DataTable tabela {
        get {return Gateway.Data.Tables[NomeDaTabela];}
    }
    protected Gateway Gateway;
    abstract protected String NomeDaTabela {get;}
```

Uma vez que só há uma única tabela, esta pode ser definida pelo mapeador de jogador abstrato.

```
class MapadorDeJogadorAbstrato...

    protected override String NomeDaTabela {
        get {return "Jogadores";}
    }
```

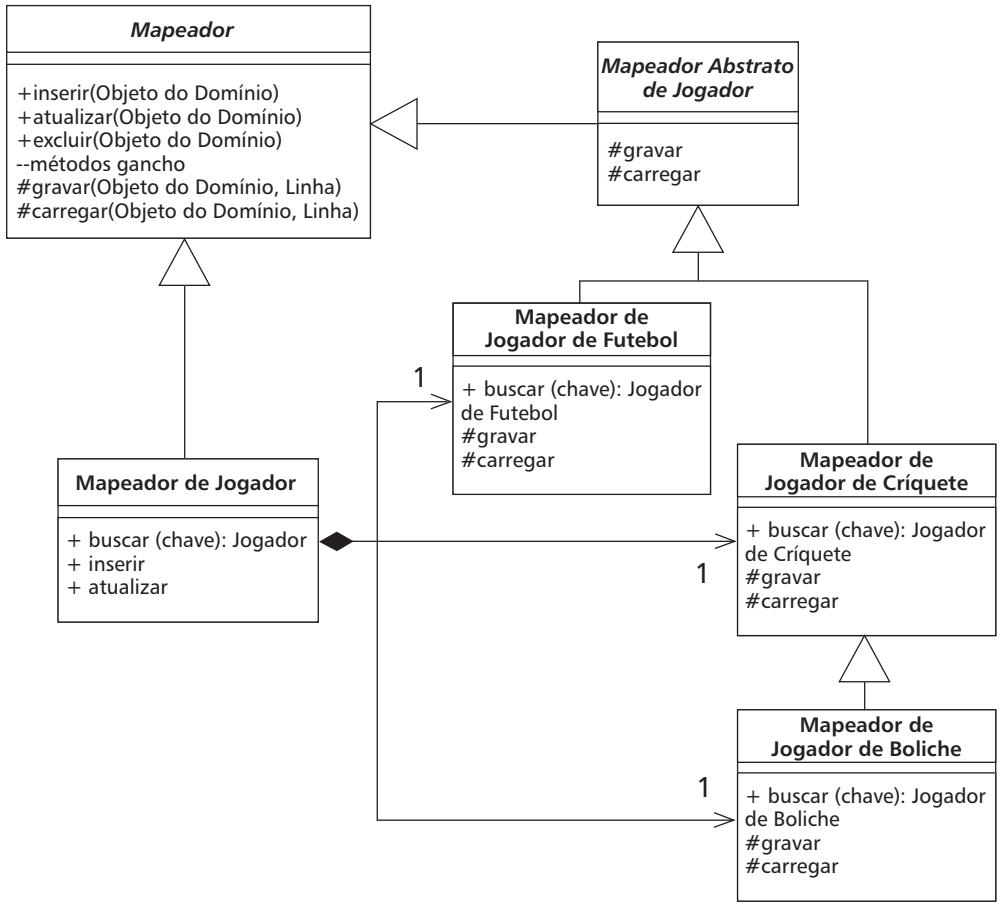



Figura 12.8 O diagrama de classes genérico dos *Mapeadores de Herança* (291).

Cada classe precisa de um código de tipo para auxiliar o código do mapeador a descobrir com que tipo de jogador ele está lidando. O código do tipo é definido na superclasse e implementado nas subclasses.

```
class MapeadorAbstratoDeJogador...

    abstract public String CódigoDoTipo {get;}

class MapeadorDeJogadorDeCríquete...

    public const String CÓDIGO_DO_TIPO = "C";
    public override String CódigoDoTipo {
        get {return CÓDIGO_DO_TIPO;}
    }
```

O mapeador de jogador tem campos para cada uma das três classes mapeadoras concretas.

```
class MapeadorDeJogador...

    private MapeadorDeJogadorDeBoliche mapeadorB;
    private MapeadorDeJogadorDeCríquete mapeadorC;
    private MapeadorDeJogadorDeFutebol mapeadorF;
    public private MapeadorDeJogador (Gateway gateway): base (gateway) {
        mapeadorB = new MapeadorDeJogadorDeBoliche (Gateway);
        mapeadorC = new MapeadorDeJogadorDeCríquete (Gateway);
        mapeadorF = new MapeadorDeJogadorDeFutebol (Gateway);
    }
```

Carregando um Objeto do Banco de Dados

Cada classe mapeadora concreta tem um método de busca que traz um objeto dos dados.

```
class MapeadorDeJogadorDeCríquete...

    public JogadorDeCríquete Buscar (long id) {
        return (JogadorDeCríquete) BuscaAbstrata(id);
    }
```

Isso invoca um comportamento genérico para buscar um objeto.

```
class Mapeador...

    protected ObjetoDoDomínio BuscaAbstrata (long id) {
        DataRow linha = BuscarLinha (id);
        return (linha == null) ? null: Buscar(linha);
    }
    protected DataRow BuscarLinha (long id) {
        String filtro = String.Format("id = {0}", id);
        DataRow[ ] resultados = tabela.Select (filtro);
        return (resultados.Length == 0) ? null: resultados[0];
    }
    public ObjetoDoDomínio Buscar (DataRow linha) {
        ObjetoDoDomínio resultado = CriarObjetoDoDomínio ( );
        Carregar (resultado, linha);
```

```
        return resultado;
    }
    abstract protected ObjetoDoDomínio CriarObjetoDoDomínio( );
```

```
class MapeadorDeJogadorDeCríquete...

    protected override ObjetoDoDomínio CriarObjetoDoDomínio ( ) {
        return new JogadorDeCríquete( );
    }
```

Carrego os dados no novo objeto com uma série de métodos de carga, um em cada classe na hierarquia.

```
class MapeadorDeJogadorDeCríquete...

    protected override void Carregar (ObjetoDoDomínio obj, DataRow linha) {
        base.Carregar (obj, linha);
        JogadorDeCríquete jogadorDeCríquete = (JogadorDeCríquete) obj;
        jogadorDeCríquete.médiaDeRebatidas = (double) linha["médiaDeRebatidas"];
    }
```

```
class MapeadorAbstratoDeJogador...

    protected override void Carregar (ObjetoDoDomínio obj, DataRow linha) {
        base.Carregar (obj, linha);
        Jogador jogador = (Jogador) obj;
        jogador.nome = (String) linha["nome"];
    }
```

```
class Mapeador...

    protected virtual void Carregar (ObjetoDoDomínio obj, DataRow linha) {
        obj.Id = (int) linha["id"];
    }
```

Também posso carregar um jogador por meio do mapeador de jogador. Ele precisa ler os dados e usar o código de tipo para determinar qual mapeador concreto usar.

```
class MapeadorDeJogador...

    public Jogador Buscar (long chave) {
        DataRow linha = BuscarLinha (chave);
        if (linha == null) return null;
        else {
            String códigoDeTipo = (String) linha["tipo"];
            switch (códigoDeTipo) {
                case MapeadorDeJogadorDeBoliche.CÓDIGO_DE_TIPO:
                    return (Jogador) mapeadorB.Buscar(linha);
                case MapeadorDeJogadorDeCríquete.CÓDIGO_DE_TIPO:
                    return (Jogador) mapeadorC.Buscar(linha);
                case MapeadorDeJogadorDeFutebol.CÓDIGO_DE_TIPO:
                    return (Jogador) mapeadorF.Buscar(linha);
                default:
                    throw new Exception ("tipo desconhecido");
            }
        }
    }
```

Atualizando um Objeto A operação básica para a atualização é a mesma para todos os objetos, de modo que posso defini-la na superclasse mapeadora.

```
class Mapeador...

    public virtual void Atualizar (ObjetoDoDomínio arg) {
        Gravar (arg, BuscarLinha(arg.id));
    }
```

O método de gravação é similar ao método de carga – cada classe o define para gravar os dados que contém.

```
class MapeadorDeJogadorDeCríquete...

    protected override void Gravar (ObjetoDoDomínio obj, DataRow linha) {
        base.Gravar(obj, linha);
        JogadorDeCríquete jogadorDeCríquete = (JogadorDeCríquete) obj;
        linha["médiaDeRebatidas"] = jogadorDeCríquete.médiaDeRebatidas;
    }

class MapeadorAbstratoDeJogador...

    protected override void Gravar (ObjetoDoDomínio obj, DataRow linha) {
        Jogador jogador = (Jogador) obj;
        linha["nome"] = jogador.nome;
        linha["tipo"] = CódigoDoTipo;
    }
```

O mapeador de jogadores transfere para o mapeador concreto apropriado.

```
class MapeadorDeJogador...

    public override void Atualizar (ObjetoDoDomínio obj) {
        MapeadorPara(obj).Atualizar (obj);
    }
    private Mapeador MapeadorPara (ObjetoDoDomínio obj) {
        if (obj is JogadorDeFutebol)
            return mapeadorF;
        if (obj is JogadorDeBoliche)
            return mapeadorB;
        if (obj is JogadorDeCríquete)
            return mapeadorC;
        throw new Exception ("Nenhum mapeador disponível");
    }
```

Inserindo um Objeto As inserções são semelhantes às atualizações. A única diferença real é que uma nova linha precisa ser criada na tabela antes da gravação.

```
class Mapeador...

    public virtual long Inserir (ObjetoDoDomínio arg) {
        DataRow linha = tabela.NewRow( );
        arg.Id = LerPróximoID( );
        linha["id"] = arg.Id;
        Gravar(arg, linha);
```

```
        tabela.Rows.Add(linha);
        return arg.Id;
    }
```

```
class mapeadorDeJogador...

    public override long Inserir (ObjetoDoDomínio obj) {
        return MapeadorPara(obj).Inserir(obj);
    }
```

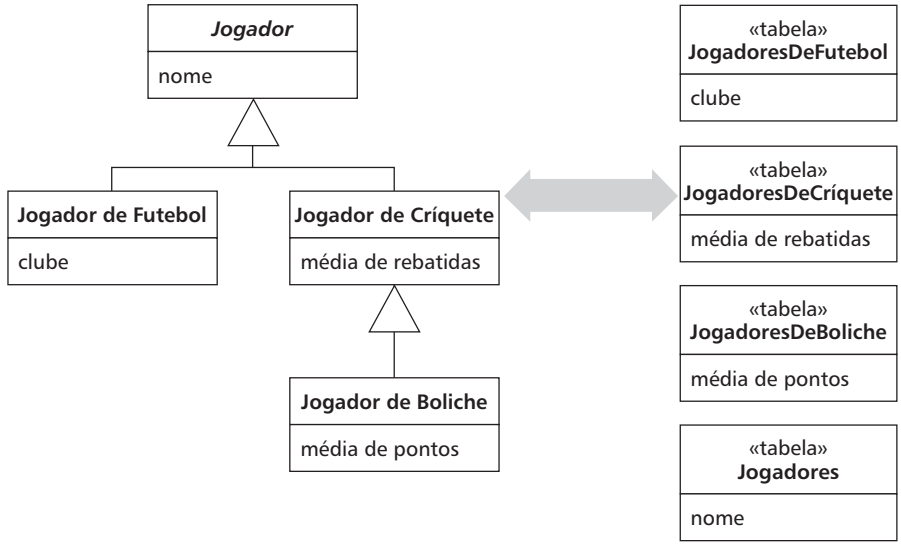
Excluindo um Objeto As exclusões são bastante simples. Elas são definidas no nível do mapeador abstrato ou no envoltório do jogador.

```
class Mapeador...

    public virtual void Excluir (ObjetoDoDomínio obj) {
        DataRow linha = BuscarLinha (obj.Id);
        linha.Delete( );
    }
```

```
class MapeadorDeJogador...

    public override void Excluir (ObjetoDoDomínio obj) {
        MapeadorPara(obj).Excluir(obj);
    }
```

Herança de Tabela de Classes (Class Table Inheritance)

Representa uma hierarquia de herança de classes com uma tabela para cada classe.



Um aspecto muito visível da incompatibilidade objeto-relacional é o fato de que os bancos de dados relacionais não suportam herança. Você quer estruturas de bancos de dados que mapeiem claramente para os objetos e que permitam associações em qualquer lugar da estrutura de herança. A *Herança de Tabela de Classes* suporta isso, usando uma tabela no banco de dados por classe da estrutura de herança.

Como Funciona

A característica evidente da *Herança de Tabela de Classes* é que ela possui uma tabela por classe no modelo do domínio. Os campos nas classes do domínio são mapeados diretamente para campos nas tabelas correspondentes. Assim como com os outros mapeamentos de herança, aqui também se aplica a abordagem básica dos *Mapeadores de Herança* (291).

Uma questão é como associar as linhas correspondentes das tabelas do banco de dados. Uma solução possível é usar um valor de chave primária comum, de modo que, digamos, a linha da chave 101 na tabela de jogadores de futebol e a linha da chave 101 na tabela de jogadores correspondam ao mesmo objeto do domínio. Visto que a tabela da superclasse tem uma linha para cada linha nas outras tabelas, se você usar este esquema as chaves primárias serão únicas por todas as tabelas. Uma alternativa é deixar cada tabela ter suas próprias chaves primárias e usar chaves estrangeiras apontando para a tabela da superclasse para conectar as linhas.

O maior problema na implementação da *Herança de Tabela de Classes* é como recuperar os dados a partir de diversas tabelas de uma maneira eficiente. Obviamente, fazer uma chamada para cada tabela não é bom, já que você teria diversas chamadas para o banco de dados. Você pode evitar isso fazendo uma junção (*join*) pelas diversas ta-

belas componentes. Não obstante, junções para mais de três ou quatro tabelas tendem a ser lentas devido ao modo pelo qual os bancos de dados executam suas otimizações.

Além disso, há o problema de que freqüentemente você não sabe exatamente quais tabelas juntar em uma consulta. Se você estiver procurando um jogador de futebol, você sabe usar a tabela de jogadores de futebol, mas se você estiver procurando por um grupo de jogadores, quais tabelas você usa? Para realizar junções de maneira eficaz quando algumas tabelas não têm dados, você precisará fazer uma junção externa (*outer join*), a qual não é padrão e, freqüentemente, é lenta. A alternativa é ler primeiro a tabela raiz e então usar um código para descobrir quais tabelas ler a seguir, mas isso envolve múltiplas consultas.

Quando Usá-la

Herança de Tabela de Classes, *Herança de Tabela Única* (269) e *Herança de Tabela Concreta* (283) são as três alternativas a considerar para o mapeamento de herança.

Os pontos fortes da *Herança de Tabela de Classes* são:

- Todas as colunas são relevantes para todas as linhas, de modo que as tabelas são mais fáceis de compreender e não desperdiçam espaço.
- O relacionamento entre o modelo do domínio e o banco de dados é bastante direto.

Os pontos fracos da *Herança de Tabela de Classes* são:

- Você precisa alcançar diversas tabelas para carregar um objeto, o que significa uma junção ou diversas consultas para depois costurar os pedaços em memória.
- Qualquer refatoração de campos para cima ou para baixo na hierarquia gera alterações no banco de dados.
- As tabelas dos supertipos podem se tornar um gargalo porque elas têm de ser acessadas freqüentemente.
- A alta normalização pode torná-la difícil de entender em consultas *ad hoc*.

Você não tem que escolher apenas um padrão de mapeamento de herança para uma hierarquia de classes. Você pode usar *Herança de Tabela de Classes* para as classes no topo da hierarquia e uma coleção de *Heranças de Tabela Concreta* (283) para aquelas mais abaixo.

Leitura Adicional

Vários textos da IBM referem-se a este padrão como Mapeamento Raiz-Folha (*Root-Leaf Mapping*) [Brown *et al.*].

Exemplo: Jogadores e Assemelhados (C#)

Aqui está a implementação para a descrição. Novamente seguirei o tema familiar (talvez um pouco tedioso) de jogadores e assemelhados, usando *Mapeadores de Herança* (291) (Figura 12.9).

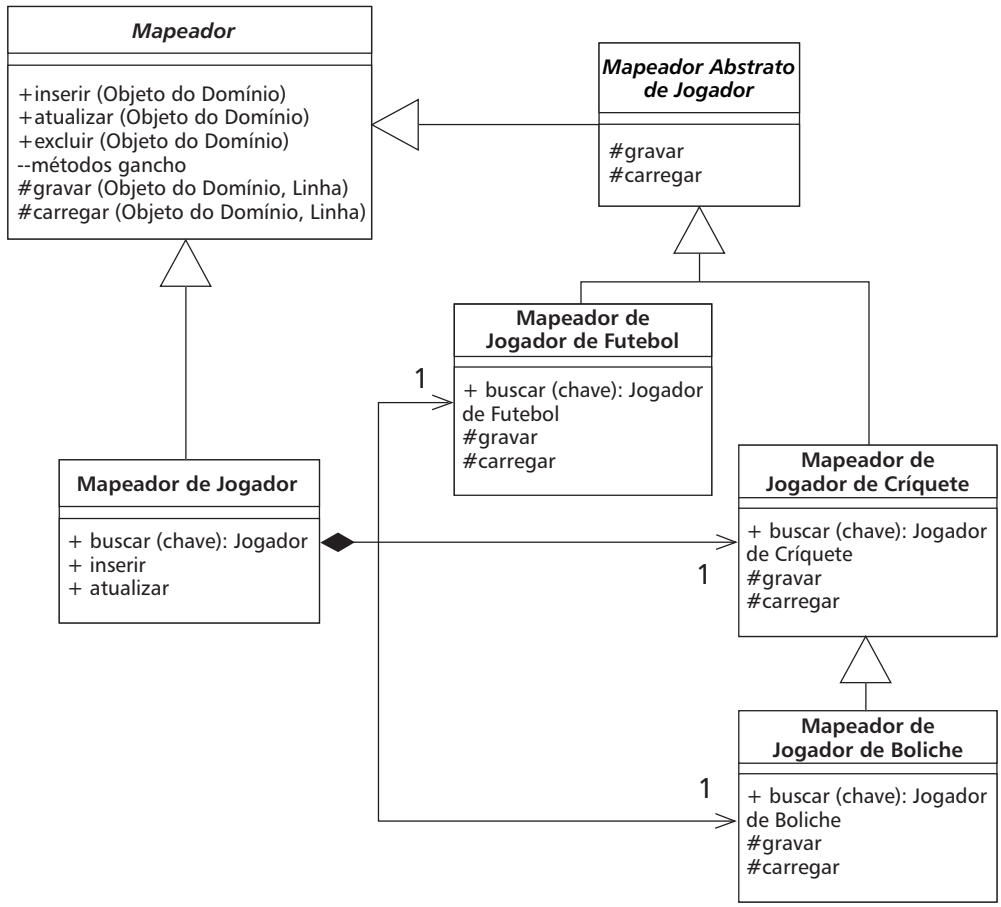


Figura 12.9 O diagrama de classes genérico dos *Mapeadores de Herança* (291).

Cada classe precisa definir a tabela que armazena os seus dados e um código de tipo para ela.

```
class MapeadorAbstratoDeJogador...

    abstract public String CódigoDoTipo {get;}
    protected static String NOME_DA_TABELA = "Jogadores";

class MapeadorDeJogadorDeFutebol...

    public override String CódigoDoTipo {
        get {return "F";}
    }
    protected new static String NOME_DA_TABELA = "JogadoresDeFutebol";
```

Diferentemente dos outros exemplos de herança, este não tem um nome de tabela sobrescrito porque temos de ter o nome da tabela para esta classe mesmo quando a instância é uma instância da subclasse.

Carregando um Objeto Se você tiver lido os outros mapeamentos, você sabe que o primeiro passo é o método de busca nos mapeadores concretos.

```
class MapeadorDeJogadorDeFutebol...

    public JogadorDeFutebol Buscar (long id) {
        return (JogadorDeFutebol) BuscaAbstrata (id, NOME_DA_TABELA);
    }
```

O método abstrato de busca procura uma linha que seja igual à chave e, se encontrar, cria um objeto do domínio e chama o seu método de carga.

```
class Mapeador...

    public ObjetoDoDomínio BuscaAbstrata (long id, String nomeDaTabela) {
        DataRow linha = BuscarLinha (id, tabelaPara(nomeDaTabela));
        if (linha == null) return null;
        else {
            ObjetoDoDomínio resultado = CriarObjetoDoDomínio( );
            resultado.Id = id;
            Carregar (resultado);
            return resultado;
        }
    }
    protected DataTable tabelaPara(String nome) {
        return Gateway.Data.Tables[nome];
    }
    protected DataRow BuscarLinha (long id, DataTable tabela) {
        String filtro = String.Format("id = {0}", id);
        DataRow [ ] resultados = tabela.Select (filtro);
        return (resultados.Length == 0) ? null: resultados[0];
    }
    protected DataRow BuscarLinha (long id, String nomeDaTabela) {
        return BuscarLinha (id, tabelaPara(nomeDaTabela));
    }
    protected abstract ObjetoDoDomínio CriarObjetoDoDomínio ( );

class MapeadorDeJogadorDeFutebol...

    protected override ObjetoDoDomínio CriarObjetoDoDomínio ( ) {
        return new JogadorDeFutebol( );
    }
```

Existe um método de carga para cada classe que carrega os dados definidos por essa classe.

```
class MapeadorDeJogadorDeFutebol...

    protected override void Carregar (ObjetoDoDomínio obj) {
        base.Carregar(obj);
        DataRow linha = BuscarLinha (obj.Id, tabelaPara (NOME_DA_TABELA));
        JogadorDeFutebol jogadorDeFutebol = (JogadorDeFutebol) obj;
        jogadorDeFutebol.clube = (String) linha["clube"];
    }
```

```
class MapeadorAbstratoDeJogador...

    protected override void Carregar (ObjetoDoDomínio obj) {
        DataRow linha = BuscarLinha (obj.Id, tabelaPara(NOME_DA_TABELA));
        Jogador jogador = (Jogador) obj;
        jogador.nome = (String) linha["nome"];
    }
```

Assim com o outro exemplo de código, porém de forma muito mais perceptível aqui, estou contando com o fato de que o conjunto de dados ADO.NET trouxe os dados do banco de dados e os colocou em cache em memória. Isso me permite fazer diversos acessos à estrutura de dados baseada em tabelas sem um alto custo em desempenho. Se você for diretamente ao banco de dados, precisará reduzir essa carga. Neste exemplo, você poderia fazer isso criando uma junção por todas as tabelas e manipulando o resultado.

O mapeador de jogadores determina que tipo de jogador ele tem que buscar e então delega ao mapeador concreto correto.


```
class MapeadorDeJogador...

    public Jogador Buscar (long chave) {
        DataRow linha = BuscarLinha (chave, tabelaPara (NOME_DA_TABELA));
        if (linha == null) return null;
        else {
            String códigoDoTipo = (String) linha["tipo"];
            if (códigoDoTipo == mapeadorB.CódigoDoTipo)
                return mapeadorB.Buscar(chave);
            if (códigoDoTipo == mapeadorC.CódigoDoTipo)
                return mapeadorC.Buscar(chave);
            if (códigoDoTipo == mapeadorF.CódigoDoTipo)
                return mapeadorF.Buscar(chave);
            throw new Exception("tipo desconhecido");
        }
    }
    protected static String NOME_DA_TABELA = "Jogadores";
```

Atualizando um Objeto O método de atualização aparece na superclasse mapeadora.

```
class Mapeador...

    public virtual void Atualizar (ObjetoDoDomínio arg) {
        Gravar(arg);
    }
```

Ele é implementado por meio de uma série de métodos de gravação, um para cada classe na hierarquia.

```
class MapeadorDeJogadorDeFutebol...

    protected override void Gravar (ObjetoDoDomínio obj) {
        base.Gravar(obj);
        DataRow linha = BuscarLinha (obj.Id, tabelaPara(NOME_DA_TABELA));
        JogadorDeFutebol jogadorDeFutebol = (JogadorDeFutebol) obj;
        linha["clube"] = jogadorDeFutebol.clube;
    }
```

```
class MapeadorAbstratoDeJogador...

    protected override void Gravar (ObjetoDoDomínio obj) {
        DataRow linha = BuscarLinha (obj.Id, tabelaPara(NOME_DA_TABELA));
        Jogador jogador = (Jogador) obj;
        linha["nome"] = jogador.nome;
        linha["tipo"] = CódigoDoTipo;
    }
```

O método de atualização do mapeador de jogadores redefine o método geral para transferir para o mapeador concreto correto.

```
class MapeadorDeJogador...

    public override void Atualizar (ObjetoDoDomínio obj) {
        MapeadorPara(obj).Atualizar(obj);
    }
    private Mapeador MapeadorPara (ObjetoDoDomínio obj) {
        if (obj is JogadorDeFutebol)
            return mapeadorF;
        if (obj is JogadorDeBoliche)
            return mapeadorB;
        if (obj is JogadorDeCríquete)
            return mapeadorC;
        throw new Exception ("Nenhum mapeador disponível");
    }
```


Inserindo um Objeto O método para inserir um objeto é declarado na superclasse mapeadora. Ele tem dois estágios: criar novas linhas no banco de dados e, então, usar os métodos de gravação para atualizar estas linhas em branco com os dados necessários.

```
class Mapeador...

    public virtual long Inserir (ObjetoDoDomínio obj) {
        obj.Id = LerPróximoID();
        AdicionarLinha(obj);
        Gravar(obj);
        return obj.Id;
    }
```

Cada classe insere uma linha em sua tabela.

```
class MapeadorDeJogadorDeFutebol...

    protected override void AdicionarLinha (ObjetoDoDomínio obj) {
        base.AdicionarLinha (obj);
        InserirLinha (obj, tabelaPara(NOME_DA_TABELA));
    }

class MapeadorAbstratoDeJogador...

    protected override void AdicionarLinha (ObjetoDoDomínio obj) {
        InserirLinha (obj, tabelaPara(NOME_DA_TABELA));
    }
```

```
class Mapeador...

    abstract protected void AdicionarLinha (ObjetoDoDomínio obj);
    protected virtual void InserirLinha ((ObjetoDoDomínio arg, DataTable tabela) {
        DataRow linha = tabela.NewRow( );
        linha["id"] = arg.Id;
        tabela.Rows.Add(linha);
    }
```

O mapeador de jogadores delega ao mapeador concreto apropriado.

```
class MapeadorDeJogador...

    public override long Inserir (ObjetoDoDomínio obj) {
        return MapeadorPara(obj).Inserir(obj);
    }
```


Excluindo um Objeto Para excluir um objeto, cada classe exclui uma linha da tabela correspondente no banco de dados.

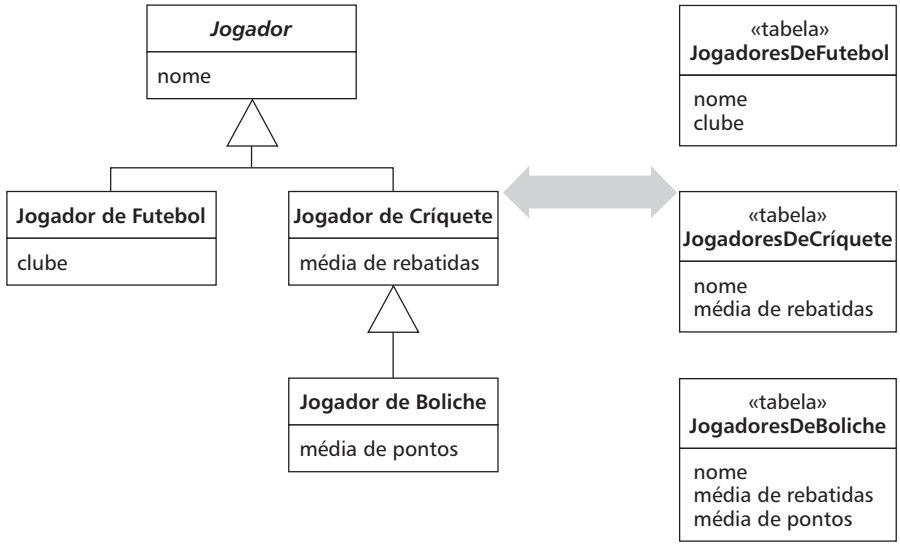
```
class MapeadorDeJogadorDeFutebol...

    public override void Excluir (ObjetoDoDomínio obj) {
        base.Excluir (obj);
        DataRow linha = BuscarLinha (obj.Id, NOME_DA_TABELA);
        linha.Delete( );
    }

class MapeadorAbstratoDeJogador...

    public override void Excluir (ObjetoDoDomínio obj) {
        DataRow linha = BuscarLinha (obj.Id, tabelaPara(NOME_DA_TABELA));
        linha.Delete( );
    }

class Mapeador...

    public abstract void Excluir (ObjetoDoDomínio obj);
```

O mapeador de jogadores novamente foge do trabalho duro e apenas delega para o mapeador concreto.

```
class MapeadorDeJogador...

    public override void Excluir (ObjetoDoDomínio obj) {
        MapeadorPara(obj).Excluir(obj);
    }
```

Herança de Tabela Concreta (Concrete Table Inheritance)

Representa um hierarquia de herança de classes com uma tabela por classe concreta na hierarquia.



Como qualquer purista da orientação a objetos lhe dirá, os bancos de dados relacionais não suportam herança – um fato que complica o mapeamento objeto-relacional. Do ponto de vista da instância de um objeto, um caminho sensato para pensar em tabelas é pegar cada objeto na memória e mapeá-lo para uma única linha no banco de dados. Isso sugere a *Herança de Tabela Concreta*, na qual existe uma tabela para cada classe concreta na hierarquia de herança.

Confesso que tive alguma dificuldade para dar um nome para este padrão. A maioria das pessoas pensa nele como orientado a folhas, já que você normalmente tem uma tabela por classe folha em uma hierarquia. Seguindo essa lógica, eu poderia chamar este padrão de herança de tabela folha, e o termo "folha" é freqüentemente usado para este padrão. A rigor, entretanto, uma classe concreta que não seja uma folha normalmente também tem uma tabela, de modo que resolvi escolher o termo mais correto, ainda que menos intuitivo.

Como Funciona

A *Herança de Tabela Concreta* usa uma tabela do banco de dados para cada classe concreta na hierarquia. Cada tabela contém colunas para a classe concreta e todos os seus ancestrais, de modo que qualquer campo na superclasse é duplicado pelas tabelas das subclasses. Assim como com todos estes esquemas de herança, o comportamento básico usa *Mapeadores de Herança* (291).

Com este padrão, você precisa prestar atenção às chaves. Fazendo um trocadilho, a chave é assegurar que as chaves sejam únicas não apenas para uma tabela, mas para todas as tabelas de uma hierarquia. Um exemplo clássico de onde você precisa disto é se você tiver uma coleção de jogadores e estiver usando um *Campo Identidade*

(215) com chaves com escopo de tabela. Se as chaves puderem ser duplicadas entre as tabelas que mapeiam as classes concretas, você terá diversas linhas para um valor particular da chave. Assim, você realmente precisa de um sistema de alocação de chaves que fique de olho no uso das chaves pelas tabelas. Além disso, você não pode depender do mecanismo de unicidade de chaves primárias do banco de dados.

Isso se torna especialmente complicado se você estiver preso a bancos de dados usados por outros sistemas. Em muitos desses casos, você não pode garantir a unicidade das chaves pelas tabelas. Nesta situação, ou você evita usar os campos da superclasse ou usa uma chave composta que envolva um identificador de tabela.

Você pode evitar alguns destes problemas não tendo campos do tipo da superclasse, mas isto obviamente compromete o modelo de objetos. Uma alternativa é ter métodos de acesso para o supertipo na interface mas usar vários campos privados para cada tipo concreto na implementação. A interface então combina os valores dos campos privados. Se a interface pública for um único valor, ela pega qualquer um dos valores privados que não forem nulos. Se a interface pública for uma coleção, ela responde com a união dos valores dos campos da implementação.

Para chaves compostas você pode usar um objeto chave especial como seu campo ID para o *Campo Identidade* (215). Esta chave usa tanto a chave primária da tabela como o nome da tabela para garantir unicidade

Relacionados a isso, estão os problemas com a integridade referencial no banco de dados. Considere um modelo de objetos como o da Figura 12.10. Para implementar integridade referencial, você precisa de uma tabela associativa que contenha colunas de chaves estrangeiras para a função caridade e para o jogador. O problema é que não existe uma tabela para o jogador, de modo que você não pode montar uma restrição de integridade referencial para o campo da chave estrangeira que recebe tanto jogadores de futebol quanto de críquete. A sua escolha é entre ignorar a integridade referencial ou usar múltiplas tabelas associativas, uma para cada uma das tabelas concretas no banco de dados. Além disso, você terá problemas se não puder garantir a unicidade das chaves.

Se você estiver procurando por jogadores com um comando *select*, precisa olhar todas as tabelas para ver quais contêm o valor apropriado. Isso significa usar múltiplas consultas ou usar uma junção externa (*outer join*), ambas opções ruins no que se refere ao desempenho. Você não sofre o problema do desempenho quando sabe de qual classe precisa, mas você tem que usar a classe concreta para melhorar o desempenho.

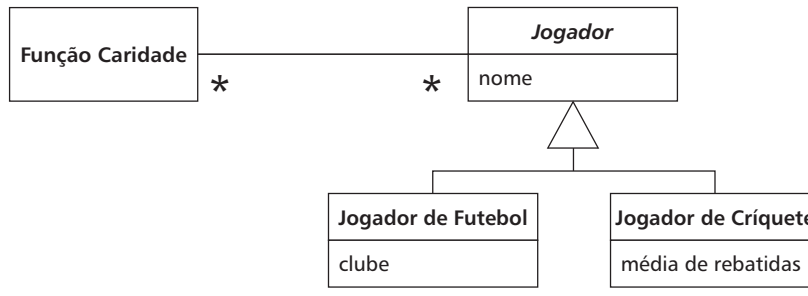


Figura 12.10 Um modelo que causa problemas de integridade referencial para a *Herança de Tabela Concreta*.

Muitas vezes, este padrão é mencionado como sendo alinhado com o padrão **herança de tabela de folhas.** Algumas pessoas preferem uma variação na qual você tem uma tabela por classe folha em vez de uma tabela por classe concreta. Se você não tiver nenhuma superclasse concreta na hierarquia, isso acaba sendo a mesma coisa. Mesmo se você tiver superclasses concretas, a diferença é bem pequena.

Quando Usá-la

Ao buscar a solução para mapear herança, as alternativas são: *Herança de Tabela Concreta*, *Herança de Tabela de Classes* (276) e *Herança de Tabela Única* (269) .

Os pontos fortes da *Herança de Tabela Concreta* são:

- Cada tabela é autocontida e não tem campos irrelevantes. A conseqüência é que ela faz bastante sentido quando usada por outras aplicações que não estejam usando os objetos.
- Não há junções a realizar durante a leitura dos dados dos mapeadores concretos.
- Cada tabela é acessada apenas quando a classe é acessada, o que pode desconcentrar a carga de acesso.

Os pontos fracos da *Herança de Tabela Concreta* são:

- Pode ser difícil tratar as chaves primárias.
- Você não pode forçar relacionamentos no banco de dados para classes abstratas.
- Se os campos das classes do domínio subirem ou descerem na hierarquia, você tem que alterar as definições das tabelas. Você não tem que fazer tantas alterações como com a *Herança de Tabela de Classes* (276), mas você não pode ignorar esse fato, como fizemos na *Herança de Tabela Única* (269).
- Se um campo da superclasse for alterado, você precisa alterar cada tabela que tenha este campo porque os campos da superclasse são duplicados pelas tabelas.
- Uma busca na superclasse obriga você a verificar todas as tabelas, o que leva a múltiplos acessos ao banco de dados (ou a uma junção esquisita).

Lembre-se de que o trio de padrões de herança pode coexistir em uma única hierarquia. Assim, você poderia usar a *Herança de Tabela Concreta* para uma ou duas subclasses e *Herança de Tabela Única* (269) para o resto.

Exemplo: Jogadores Concretos (C#)

Aqui mostrarei uma implementação para a descrição do padrão. Assim como em todos os exemplos de herança neste capítulo, estou usando o projeto básico das classes de *Mapeadores de Herança* (291), mostrado na Figura 12.11.

Cada mapeador é associado à tabela do banco de dados que é a fonte dos dados. Em ADO.NET um conjunto de dados (*data set*) armazena os dados da tabela.

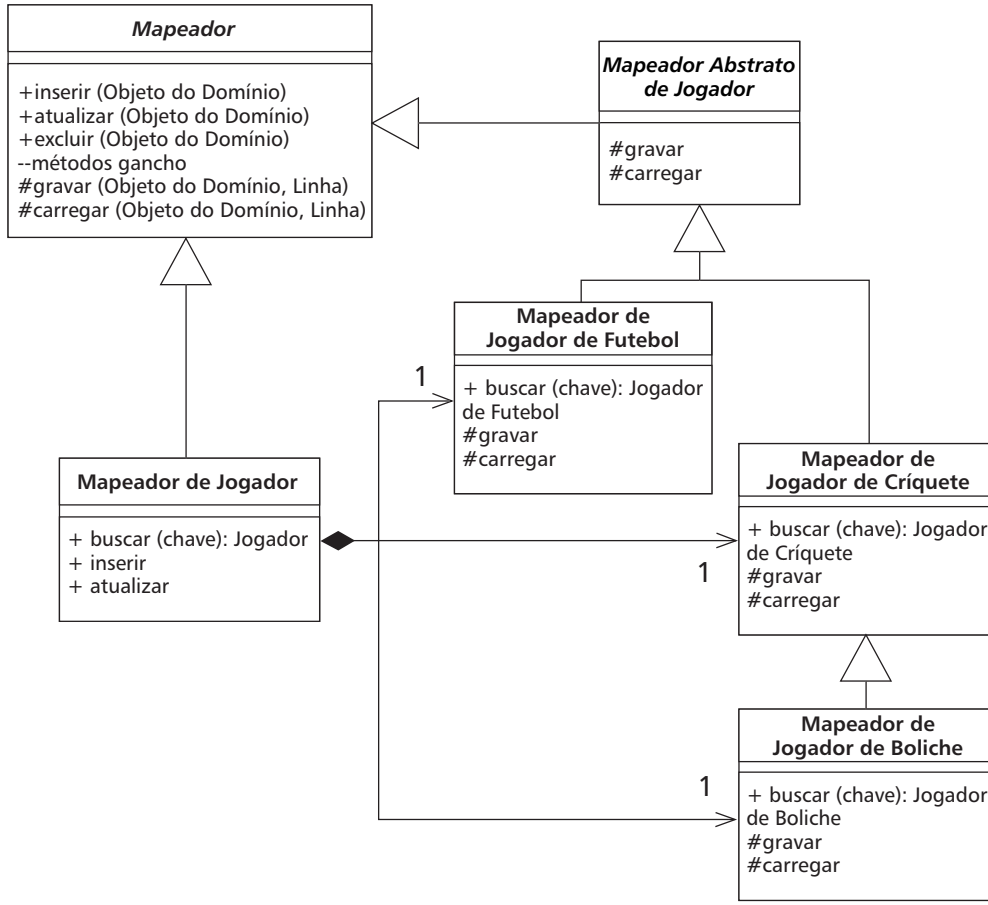


Figura 12.11 O diagrama de classes genérico dos *Mapeadores de Herança* (291).

```
class Mapeador...

    public Gateway Gateway;
    private IDictionary mapaDeIdentidade = new Hashtable( );
    public Mapeador (Gateway gateway) {
        this.Gateway = gateway;
    }
    private DataTable tabela {
        get {return Gateway.Data.Tables[NomeDaTabela];}
    }
    abstract public String NomeDaTabela {get;}
```

A classe *gateway* armazena o conjunto de dados dentro de sua propriedade de dados. Os dados podem ser carregados fornecendo-se consultas apropriadas.

```
class Gateway...

    public DataSet Dados = new DataSet ( );
```

Cada mapeador concreto precisa definir o nome da tabela que armazena seus dados.

```
class MapeadorDeJogadorDeCríquete...

       public override String NomeDaTabela {
           get {return "JogadoresDeCríquete";}
    }
```

O mapeador de jogadores tem campos para cada mapeador concreto.

```
class MapeadorDeJogador...

    private MapeadorDeJogadorDeBoliche mapeadorB;
    private MapeadorDeJogadorDeCríquete mapeadorC;
    private MapeadorDeJogadorDeFutebol mapeadorF;
    public MapeadorDeJogador (Gateway gateway): base (gateway) {
        mapeadorB = new MapeadorDeJogadorDeBoliche (Gateway);
        mapeadorC = new MapeadorDeJogadorDeCríquete (Gateway);
        mapeadorF = new MapeadorDeJogadorDeFutebol (Gateway);
    }
```


Carregando um Objeto do Banco de Dados Cada classe mapeadora concreta tem um método de busca que retorna um objeto dado um valor da chave.

```
class MapeadorDeJogadorDeCríquete...

    public JogadorDeCríquete Buscar (long id) {
        return (JogadorDeCríquete) BuscaAbstrata(id);
    }
```

O comportamento abstrato na superclasse busca a linha correta do banco de dados para o ID, cria um novo objeto do domínio do tipo correto e usa o método de carga para carregá-lo (descreverei a carga logo em seguida).

```
class Mapeador...

    public ObjetoDoDomínio BuscaAbstrata (long id) {
        DataRow linha = BuscarLinha (id);
        if (linha == null) return null;
        else{
            ObjetoDoDomínio resultado = CriarObjetoDoDomínio( );
            Carregar(resultado, linha);
            return resultado;
        }
    }
    private DataRow BuscarLinha (long id) {
        String filtro = String.Format("id = {0}", id);
        DataRow[ ] resultados = tabela.Select(filtro);
        if (resultados.Length == 0) return null;
        else return resultados[0];
    }
    protected abstract ObjetoDoDomínio CriarObjetoDoDomínio( );
```

```
class MapeadorDeJogadorDeCríquete...

    protetected override ObjetoDoDomínio CriarObjetoDoDomínio ( ) {
        return new JogadorDeCríquete ( );
    }
```

A carga real de dados do banco de dados é feita pelo método de carga, ou então por diversos métodos de carga: um de cada para a classe mapeadora e para todas as suas superclasses.

```
class MapeadorDeCríquete...

    protected override void Carregar (ObjetoDeDomínio obj, DataRow linha) {
        base.Carregar(obj, linha);
        JogadorDeCríquete jogadorDeCríquete = (JogadorDeCríquete) obj;
        jogadorDeCríquete.médiaDeRebatidas = (double) linha["médiaDeRebatidas"];
    }

class MapeadorAbstratoDeJogador...

    protected override void Carregar (ObjetoDoDomínio obj, DataRow linha) {
        base.Carregar(obj, linha);
        Jogador jogador = (Jogador) obj;
        jogador.nome = (String) linha["nome"];
    }

class Mapeador...

    protected virtual void Carregar (ObjetoDoDomínio obj, DataRow linha) {
        obj.Id = (int) linha["id"];
    }
```

Esta é a lógica para encontrar um objeto, usando um mapeador para uma classe concreta. Você também pode usar um mapeador para a superclasse: o mapeador de jogadores, o qual precisa buscar um objeto de qualquer tabela em que ele esteja residindo. Uma vez que todos os dados já estão no conjunto de dados em memória, posso fazer isso como:

```
class MapeadorDeJogador...

    public Jogador Buscar (long chave) {
        Jogador resultado;
        resultado = mapeadorF.Buscar(chave);
        if (resultado != null) return resultado;
        resultado = mapeadorB.Buscar(chave);
        if (resultado != null) return resultado;
        resultado = mapeadorC.Buscar(chave);
        if (resultado != null) return resultado;
        return null;
    }
```

Lembre-se de que isso é razoável apenas porque os dados já estão na memória. Se você precisar ir ao banco de dados três vezes (ou mais para mais subclasses), isso ficará lento. Pode ajudar a executar uma junção por todas as tabelas concretas, o que lhe permitirá acessar os dados em uma única chamada ao banco de dados. Entretanto, junções grandes são, por direito, freqüentemente lentas, de modo que você preci-

será executar alguns testes com sua própria aplicação para descobrir o que funciona e o que não funciona no seu caso. Além disso, esta será uma junção externa (*outer join*), a qual, além de tornar lenta a consulta, usa uma sintaxe não-portável e freqüentemente obscura.

Atualizando um Objeto O método de atualização pode ser definido na superclasse mapeadora.

```
class Mapeador...

    public virtual void Atualizar (ObjetoDoDomínio arg) {
        Gravar (arg, BuscarLinha (arg.Id));
    }
```

De forma semelhante à carga, usamos uma seqüência de métodos de gravação para cada classe mapeadora.

```
class MapeadorDeJogadorDeCríquete...

    protected override void Gravar (ObjetoDoDomínio obj, DataRow linha) {
        base.Gravar (obj, linha);
        JogadorDeCríquete jogadorDeCríquete = (JogadorDeCríquete) obj;
        linha["médiaDeRebatidas"] = jogadorDeCríquete.médiaDeRebatidas;
    }

class MapeadorDeJogadorAbstrato...

    protected override void Gravar (ObjetoDeDomínio obj, DataRow linha) {
        Jogador jogador = (Jogador) obj;
        linha["nome"] = jogador.nome;
    }
```

O mapeador de jogadores precisa encontrar o mapeador concreto correto a ser usado e então delegar a chamada de atualização.

```
class MapeadorDeJogador...

    public override void Atualizar (ObjetoDoDomínio obj) {
        MapeadorPara(obj).Atualizar(obj);
    }
    private Mapeador MapeadorPara (ObjetoDoDomínio obj) {
        if (obj is JogadorDeFutebol)
            return mapeadorF;
        if (obj is JogadorDeBoliche)
            return mapeadorB;
        if (obj is JogadorDeCríquete)
            return mapeadorC;
        throw new Exception ("Nenhum mapeador disponível");
    }
```

Inserindo um Objeto A inserção é uma variação da atualização. O comportamento extra é a criação de uma nova linha, o que pode ser feito na superclasse.

```
class Mapeador...

    public virtual long Inserir (ObjetoDoDomínio arg) {
        DataRow linha = tabela.NewRow( );
        arg.Id = LerPróximoID( );
        linha["id"] = arg.Id;
        Gravar(arg, linha);
        tabela.Rows.Add (linha);
        return arg.Id;
    }
```

Mais uma vez, a classe jogador delega para o mapeador apropriado.

```
class MapeadorDeJogador...

    public override long Inserir (ObjetoDoDomínio obj) {
        return MapeadorPara (obj).Inserir(obj);
    }
```


Excluindo um Objeto A exclusão é bastante direta. Como antes, temos um método definido na superclasse.

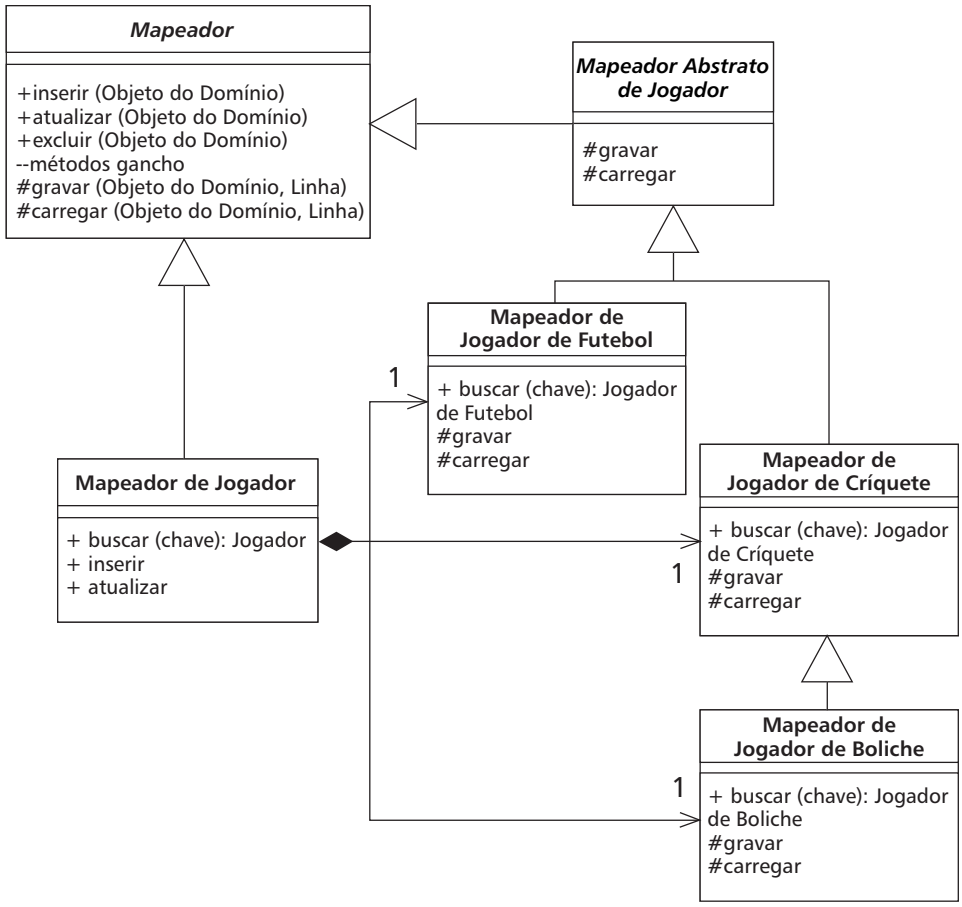
```
class Mapeador...

    public virtual void Excluir (ObjetoDoDomínio obj) {
        DataRow linha = BuscarLinha (obj.Id);
        linha.Delete( );
    }
```

e um método no mapeador de jogadores que executa a delegação.

```
class MapeadorDeJogador...

    public override void Excluir (ObjetoDoDomínio obj) {
        MapeadorPara(obj).Excluir(obj);
    }
```

Mapeadores de Herança (Inheritance Mappers)

Uma estrutura para organizar os mapeadores de bancos de dados que lidam com hierarquias de herança.



Quando você mapeia de uma hierarquia de herança orientada a objetos em memória para um banco de dados relacional, tem que minimizar a quantidade de código necessária para gravar e carregar os dados do banco de dados. Você também quer fornecer tanto comportamento abstrato de mapeamento quanto comportamento concreto que lhe permitam gravar ou carregar uma superclasse ou uma subclasse.

Ainda que os detalhes deste comportamento variem com o esquema de mapeamento de herança utilizado (*Herança de Tabela Única* (269), *Herança de Tabela de Classes* (276) e *Herança de Tabela Concreta* (283)), a estrutura geral funciona da mesma maneira para todos eles.

Como Funciona

Você pode organizar os mapeadores com uma hierarquia a fim de que cada classe do domínio tenha um mapeador que grave e carregue os dados para ela. Dessa forma, você tem um ponto em que pode alterar o mapeamento. Esta abordagem funciona bem com mapeadores concretos que sabem como mapear os objetos concretos na hierarquia. Existem ocasiões, no entanto, em que você também precisa de mapeadores para as classes abstratas. Estes podem ser implementados com mapeadores que, na verdade, estão fora da hierarquia básica, mas delegam para os mapeadores concretos apropriados.

Para explicar melhor como isso funciona, começarei com os mapeadores concretos. No exemplo, os mapeadores concretos são os mapeadores de jogadores de futebol, de críquete e de boliche. Seu comportamento básico inclui as operações de busca, inserção, atualização e exclusão.

Os métodos de busca são declarados nas subclasses concretas porque retornarão uma classe concreta. Portanto, o método de busca no MapeadorDeJogadorDeBoliche deveria retornar um jogador de boliche, não uma classe abstrata. As linguagens OO comuns não deixam você alterar o tipo de retorno declarado de um método, de modo que não é possível herdar a operação de busca e ainda declarar um tipo específico de retorno. Você pode, é claro, retornar um tipo abstrato, mas isso obriga o usuário da classe a executar um *downcast* – o que é melhor evitar. (Uma linguagem com atribuição dinâmica de tipos não tem este problema.)

O comportamento básico do método de busca é encontrar a linha apropriada no banco de dados, instanciar um objeto do tipo correto (uma decisão que é tomada pela subclasse) e então carregar o objeto com os dados do banco de dados. O método de carga é implementado por cada mapeador na hierarquia, o qual carrega o comportamento para o seu objeto do domínio correspondente. Isso significa que o método de carga do mapeador de jogador de boliche carrega os dados específicos para a classe jogador de boliche e chama o método da superclasse para carregar os dados específicos do jogador de críquete, o qual chama o método da sua superclasse e assim por diante.

Os métodos de inserção e atualização operam de forma semelhante, usando um método de gravação. Aqui você pode definir a interface na superclasse – na verdade, em uma *Camada Supertipo* (444). O método de inserção cria uma nova linha e então grava os dados do objeto do domínio usando os ganchos para os métodos de gravação. O método de atualização apenas grava os dados, também usando os ganchos para os métodos de gravação. Estes métodos operam de forma similar aos ganchos para os métodos de carga, com cada classe armazenando os seus dados específicos e chamando o método de gravação da superclasse.

Este esquema torna fácil escrever os mapeadores apropriados para gravar as informações necessárias para uma parte específica da hierarquia. O próximo passo é suportar a carga e gravação de uma classe abstrata – neste exemplo, um jogador. Ainda que o primeiro pensamento seja colocar métodos apropriados no mapeador da superclasse, isso, na verdade, se torna complicado. Embora as classes mapeadoras concretas possam apenas usar os métodos de inserção e atualização do mapeador abstrato, a inserção e atualização do mapeador de jogadores precisam sobrescrevê-los para, em seu lugar, invocar um mapeador concreto. O resultado é uma dessas combinações de generalização e composição que dão um nó na sua cabeça.

Prefiro separar os mapeadores em duas classes. O mapeador abstrato de jogadores é responsável por carregar e gravar os dados do jogador específico no banco de dados. Esta é uma classe abstrata cujo comportamento é usado apenas pelos objetos mapeadores concretos. Uma classe mapeadora de jogadores distinta é usada como interface para as operações no nível do jogador. O mapeador de jogadores fornece um método de busca e sobrescreve os métodos de inserção e atualização. A responsabilidade de todos eles é descobrir qual mapeador concreto deveria tratar a tarefa e delegar para ele.

Ainda que um esquema genérico como este faça sentido para todos os tipos de mapeamento de herança, os detalhes de fato variam. Portanto, não é possível mostrar um exemplo de código para este caso. Você pode encontrar bons exemplos em cada uma das seções dos padrões de mapeamento de herança: *Herança de Tabela Única* (269), *Herança de Tabela de Classes* (276) e *Herança de Tabela Concreta* (283).

Quando Usá-los

Este esquema geral faz sentido para qualquer mapeamento de banco de dados baseado em herança. As alternativas envolvem coisas como duplicar o código de mapeamento da superclasse entre os mapeadores concretos e incluir a interface do jogador na classe mapeadora abstrata de jogadores. O primeiro é um crime hediondo, e o último é possível, mas leva a uma classe mapeadora de jogadores deselegante e confusa. Em geral, então, é difícil pensar em uma boa alternativa para este padrão.

CAPÍTULO 13

Padrões de Mapeamento em Metadados Objeto-Relacionais

Mapeamento em Metadados (Metadata Mapping)

Armazena os detalhes do mapeamento objeto-relacional em metadados.

Muito do código que lida com mapeamento objeto-relacional descreve como os campos no banco de dados correspondem aos campos dos objetos na memória. O código resultante tende a ser tedioso e repetitivo de escrever. Um *Mapeamento em Metadados* permite aos desenvolvedores definir os mapeamentos em uma forma tabular simples, a qual então pode ser processada por código genérico para realizar os detalhes de leitura, inserção e atualização dos dados.

Como Funciona

A maior decisão a ser tomada no uso do *Mapeamento em Metadados* é como as informações nos metadados se expressam em termos de código em execução. Há dois caminhos principais a trilhar: geração de código e programação reflexiva.

Com a **geração de código** você escreve um programa cuja entrada são os metadados e cuja saída é o código fonte das classes que executam o mapeamento. Essas classes parecem ter sido escritas à mão, mas são inteiramente geradas durante o processo de construção, normalmente, imediatamente antes da compilação. As classes de mapeamento resultantes são distribuídas com o código servidor.

Se você usar a geração de código, você deve se assegurar de que ela é completamente integrada ao seu processo de construção (*build*) seja qual for o roteiro de construção que você estiver usando. As classes geradas nunca devem ser editadas à mão e, desta forma, não precisam ter o seu código fonte controlado.

Um **programa reflexivo** pode solicitar um método chamado `gravarNome` a um objeto e então executar uma chamada a esse método passando o parâmetro apropriado. Tratando métodos (e campos) como dados, o programa reflexivo pode ler os nomes de campos e métodos de um arquivo de metadados e usá-los para realizar o mapeamento. Normalmente, desaconselho a reflexão, em parte porque ela é lenta, mas principalmente porque ela freqüentemente produz código difícil de depurar. Ainda assim, a reflexão é, na verdade, bastante apropriada para o mapeamento em banco de dados. Uma vez que você está lendo os nomes dos campos e dos métodos de um arquivo, você está tirando total proveito da flexibilidade da reflexão.

A geração de código é uma abordagem menos dinâmica, já que qualquer alteração no mapeamento requer a recompilação e a redistribuição de pelo menos parte do *software*. Com uma abordagem reflexiva, você pode apenas alterar o arquivo de dados do mapeamento, e as classes existentes usarão os novos metadados. Você pode fazer isso até mesmo em tempo de execução, relendo os metadados quando ocorrer alguma interrupção específica. Alterações nos mapeamentos deveriam ser bastante raras, já que implicam alterações no banco de dados ou no código. Os modernos ambientes de desenvolvimento facilitam ainda a tarefa de redistribuir parte de uma aplicação.

A programação reflexiva muitas vezes padece com a velocidade, embora o problema aqui dependa muito do ambiente que você estiver usando – em alguns, uma chamada reflexiva pode ser uma ordem de grandeza mais lenta. Lembre-se, contudo, de que a reflexão está sendo executada no contexto de uma chamada SQL, então, sua velocidade mais lenta pode não fazer tanta diferença assim, considerando a baixa velocidade da chamada remota. Como com qualquer aspecto relacionado ao desempenho, você precisa efetuar medidas dentro do seu ambiente para descobrir o quanto este é um fator importante para você.

Ambas as abordagens podem ser um pouco complicadas para depurar. A comparação entre elas depende muito do quão habituados estão os desenvolvedores com código gerado e código reflexivo. O código gerado é mais explícito, de modo que você pode ver o que está acontecendo no depurador. Como conseqüência, normalmente, prefiro a geração à reflexão e penso que ela é normalmente mais fácil para desenvolvedores menos sofisticados (o que, suponho, faz de mim um desenvolvedor não-sofisticado).

Na maioria das ocasiões, você mantém os metadados em um formato de arquivo separado. XML é, atualmente, uma escolha popular uma vez que fornece uma estrutura hierárquica ao mesmo tempo em que o livra de escrever seus próprios analisadores (*parsers*) e outras ferramentas. Uma etapa de carga pega esses metadados e os transforma em uma estrutura de linguagem de programação, a qual então conduz o processo de geração de código ou o mapeamento reflexivo.

Nos casos mais simples, você pode dispensar o arquivo externo e criar a representação dos metadados diretamente no código fonte. Isso lhe poupa de ter que realizar a análise sintática (*parse*), mas torna a edição dos metadados um pouco mais difícil.

Uma alternativa é manter as informações do mapeamento no próprio banco de dados, que as mantém junto aos dados. Se o esquema do banco de dados for alterado, a informação do mapeamento já está lá.

Quando você estiver decidindo de que modo armazenar as informações em metadados, você pode, quase que completamente, esquecer as questões relativas ao desempenho no acesso e na análise sintática (*parsing*). Se você usar a geração de código, o acesso e a análise ocorrem apenas durante a construção e não durante a execução. Se você estiver usando a programação reflexiva, você, tipicamente, efetuará o acesso e a análise durante a execução, mas apenas uma única vez, durante a inicialização do sistema; você pode então manter a representação em memória.

O quão complexo tornar seus metadados é uma das suas maiores decisões. Quando você se depara com um problema genérico de mapeamento relacional, existe uma série de fatores diferentes para manter nos metadados, mas muitos projetos podem arranjar-se com muito menos do que um esquema geral completo, e, assim, seus metadados podem ser muito mais simples. No geral, vale mais a pena desenvolver seu projeto à medida que as suas necessidades aumentarem, já que não é difícil adicionar novas capacidades a um *software* orientado a metadados.

Um dos desafios dos metadados é que, embora um esquema simples de metadados freqüentemente funcione bem durante 90% do tempo, há casos especiais que tornam a vida muito mais complicada. Para lidar com esta minoria de casos, você freqüentemente tem que adicionar uma grande complexidade aos metadados. Uma alternativa útil é sobrescrever o código genérico com subclasses nas quais o código especial é escrito à mão. Essas subclasses de casos especiais são subclasses tanto do código gerado quanto das rotinas reflexivas. Uma vez que esses casos especiais são... bem... especiais, não é fácil descrever em termos gerais como você deve organizar as

coisas para sustentar a sobrescrita. Meu conselho é tratá-las caso a caso. Quando você precisar da sobrescrita, altere o código gerado/reflexivo para isolar o método que deve ser sobrescrito e então sobrescreva-o no seu caso especial.

Quando Usá-lo

O *Mapeamento em Metadados* pode reduzir muito a quantidade de trabalho necessária para tratar o mapeamento em um banco de dados. Entretanto, algum trabalho de configuração é requerido para preparar o *framework* do *Mapeamento em Metadados*. Além disso, ainda que freqüentemente seja fácil tratar a maioria dos casos com *Mapeamento em Metadados*, você pode encontrar exceções que realmente complicam os metadados.

Não é surpresa que as ferramentas comerciais para o mapeamento objeto-relacional usem o *Mapeamento em Metadados* – ao vender um produto, produzir um *Mapeamento em Metadados* sofisticado sempre vale o esforço.

Se você estiver construindo seu próprio sistema, você mesmo deve avaliar os compromissos. Compare a adição de novos mapeamentos usando código escrito à mão com o uso do *Mapeamento em Metadados*. Se você usar reflexão, investigue seus efeitos sobre o desempenho. Às vezes, ela causa uma diminuição na velocidade, mas, às vezes, não. Sua própria medição revelará se este é um problema para você.

O trabalho adicional de codificar à mão pode ser grandemente reduzido pela criação de uma boa *Camada Supertipo* (444) que trate todo o comportamento comum. Desta maneira, você precisará apenas adicionar algumas poucas rotinas para cada mapeamento. Normalmente, o *Mapeamento em Metadados* pode reduzir ainda mais este número.

O *Mapeamento em Metadados* pode interferir com a refatoração, especialmente se você estiver usando ferramentas automatizadas. Se você alterar o nome de um campo privado, pode fazer uma aplicação falhar inesperadamente. Mesmo as ferramentas automatizadas de refatoração não conseguirão descobrir o nome do campo escondido em um arquivo XML de dados de um mapeamento. Usar a geração de código é um pouco mais fácil, uma vez que os mecanismos de busca podem descobrir a convenção corrente. Ainda assim, qualquer atualização automática será perdida quando você regerar o código. Uma ferramenta pode lhe avisar sobre um problema, mas é sua responsabilidade alterar os metadados. Se você usar reflexão, você nem mesmo receberá o aviso.

Por outro lado, o *Mapeamento em Metadados* pode tornar a refatoração do banco de dados mais fácil, já que os metadados representam uma declaração da interface do esquema do seu banco de dados. Dessa forma, alterações no banco de dados pode ser contingenciadas por alterações no *Mapeamento em Metadados*.

Exemplo: Usando Metadados e Reflexão (Java)

A maioria dos exemplos neste livro usa código explícito porque é o mais fácil de entender. Entretanto, isso leva a uma programação bastante tediosa, e programação tediosa é um sinal de que algo está errado. Você pode remover bastante da programação tediosa usando metadados.

Armazenando os metadados A primeira questão a ser respondida sobre os metadados é como eles serão mantidos. Aqui os estou mantendo em duas classes. O mapa de dados corresponde ao mapeamento de uma classe para uma tabela. Este é um mapeamento simples, mas será suficiente para o exemplo.

```
class MapaDeDados...

    private Class classeDoDomínio;
    private String nomeDaTabela;
    private List mapasDeColunas = new ArrayList( );
```

O mapa de dados contém uma coleção de mapas de colunas que mapeiam colunas na tabela para campos.

```
class MapaDeColunas...

    private String nomeDaColuna;
    private String nomeDoCampo;
    private Field campo;
    private MapaDeDados mapaDeDados;
```

Este não é um mapeamento extremamente sofisticado. Estou apenas usando os mapeamentos de tipos padrão de Java, o que significa que não há conversão de tipos entre campos e colunas. Também estou forçando um relacionamento um-para-um entre tabelas e classes.

Estas estruturas armazenam os mapeamentos. A próxima questão é como elas são povoadas. Para este exemplo irei povoá-las com código Java em classes específicas de mapeamento. Isso pode parecer um pouco estranho, mas proporciona a maioria dos benefícios dos metadados – evitando código repetitivo.


```
class MapeadorDePessoa...

    protected void carregarMapaDeDados ( ) {
        mapaDeDados = new MapaDeDados(Pessoa.class, "pessoas");
        mapaDeDados.adicionarColuna("sobrenome", "varchar", "sobreNome");
        mapaDeDados.adicionarColuna("prenome", "varchar", "preNome");
        mapaDeDados.adicionarColuna("número_de_dependentes", "int", "númeroDeDependentes");
    }
```

Durante a construção do mapeador de colunas, crio a conexão com o campo. A rigor, isso é uma otimização, de modo que você pode não ter que calcular os campos. Entretanto, fazer isso reduz os acessos subseqüentes em uma ordem de magnitude no meu pequeno *laptop*.

```
class MapaDeColunas...

    public MapaDeColunas (String nomeDaColuna, String nomeDoCampo, MapaDeDados mapaDeDados) {
        this.nomeDaColuna = nomeDaColuna;
        this.nomeDoCampo = nomeDoCampo;
        this.mapaDeDados = mapaDeDados;
        inicializarCampo( );
    }
    private void inicializarCampo ( ) {
        try {
            campo = mapaDeDados.lerClasseDoDomínio( ).getDeclaredField (lerNomeDoCampo( ));
            campo.setAccessible(true);
        } catch (Exception e) {
            throw new ApplicationException ("Não foi possível configurar o campo: " + nomeDoCampo, e);
        }
    }
```

Não é um desafio muito grande escrever uma rotina para carregar o mapa de um arquivo XML ou dos metadados de um banco de dados. Embora esse desafio não seja grande, declinarei dele e o deixarei para você.

Agora que os mapeamentos estão definidos, posso fazer uso deles. O poder da abordagem de metadados é que todo o código que em verdade manipula alguma coisa está em uma superclasse, de modo que não tenho que escrever o código de mapeamento que tive de escrever nos casos explícitos.

Busca pelo ID Começarei com o método de busca pelo ID.

```
class Mapeador...

    public Object buscarObjeto (Long chave) {
        if (udt.estáCarregado(chave)) return udt.lerObjeto(chave);
        String sql = "SELECT" + mapaDeDados.listaDeColunas( ) + " FROM "+ mapaDeDados.
            lerNomeDaTabela( ) + " WHERE ID = ?";
        PreparedStatement dec = null;
        ResultSet rs = null;
        ObjetoDoDomínio resultado = null;
        try {
            dec = DB.prepare(sql);
            dec.setLong(1, chave.longValue( ));
            rs = dec.executeQuery( );
            rs.next( );
            resultado = carregar(rs);
        } catch (Exception e) { throw new ApplicationException (e);
        } finally {DB.cleanUp(dec, rs);
        }
     return resultado;
    }
    private UnidadeDeTrabalho udt;
    protected MapaDeDados mapaDeDados;

class DataMap...

    public String listaDeColunas ( ) {
        StringBuffer resultado = new StringBuffer(" ID");
        for (Iterator it = mapasDeColunas.iterator( ); it.hasNext( ); ) {
            resultado.append(",");
            MapaDeColunas mapaDeColunas = (MapaDeColunas) it.next( );
            resultado.append(mapaDeColunas.lerNomeDaColuna( ));
        }
        return resultado.toString( );
    }
    public String lerNomeDaTabela ( ) {
        return nomeDaTabela;
    }
```

O comando SELECT é construído mais dinamicamente do que os outros exemplos, mas ainda assim vale a pena prepará-lo de modo que seja possível à sessão do banco de dados colocá-la em cache. Se isso for um problema, a lista de colunas pode ser calculada durante a construção e colocada em cache, uma vez que não há nenhuma chamada para atualização de colunas durante a vida do mapa de dados. Para este exemplo estou usando uma *Unidade de Trabalho* (187) para lidar com a sessão do banco de dados.

Como é comum nos exemplos neste livro, separei a carga da busca, de modo que podemos usar o mesmo método de carga a partir de outros métodos de busca.

```
class Mapeador...

    public ObjetoDoDomínio carregar (ResultSet rs)
            throws InstantiationException, IllegalAccessException, SQLException
    {
        Long chave = new Long (rs.getLong("ID"));
        if (udt.estáCarregado(chave)) return udt.lerObjeto(chave);
        ObjetoDoDomínio resultado =
            (ObjetoDoDomínio) mapaDeDados.lerClasseDoDomínio( ).newInstance( );
        resultado.gravarID(chave);
        udt.registrarLimpo(resultado);
        carregarCampos(rs, resultado);
        return resultado;
    }
    private void carregarCampos (ResultSet rs, ObjetoDoDomínio resultado) throws SQLException {
        for (Iterator it = mapaDeDados.lerColunas( ); it.hasNext( ); ) {
            MapaDeColunas mapaDeColunas = (MapaDeColunas) it.next( );
            Object valorDaColuna = rs.getObject(mapaDeColunas.lerNomeDaColuna( ));
            mapaDeColunas.gravarCampo(resultado, valorDaColuna);
        }
    }

class MapaDeColunas...

    public void gravarCampo (Object resultado, Object valorDaColuna) {
        try {
            campo.set(resultado, valorDaColuna);
        } catch (Exception e) { throw new ApplicationException ("Erro na gravação " + nomeDoCampo, e);
        }
    }
```

Este é um programa reflexivo clássico. Passamos por cada um dos mapas de colunas e os usamos para carregar o campo no objeto do domínio. Separei o método `carregarCampos` para mostrar como podemos estendê-lo para casos mais complicados. Se tivermos uma classe e uma tabela onde a hipótese simples dos metadados não vale, posso simplesmente sobrescrever o método `carregarCampos` em uma subclasse mapeadora e colocar aí o código arbitrariamente complexo. Esta é uma técnica comum com metadados – fornecer um lugar em que sobrescrever em casos menos comuns. Normalmente, é muito mais fácil sobrescrever os casos incomuns com subclasses do que criar metadados sofisticados o suficiente para armazenar alguns casos especiais e raros.

É claro que, se tivermos uma subclasse, poderíamos também usá-la para evitar *downcasting*.

```
class MapeadorDePessoa...

    public Pessoa buscar(Long chave) {
        return (Pessoa) buscarObjeto(chave);
    }
```

Gravando no Banco de Dados Para atualizações uso uma única rotina.

```
class Mapeador...

    public void atualizar (ObjetoDoDomínio obj) {
        String sql = "UPDATE " + mapaDeDados.lerNomeDaTabela( ) + mapaDeDados.listaDeAtualização( ) +
" WHERE ID = ?";
        PreparedStatement dec = null;
        try {
            dec = DB.prepare(sql);
            int contadorDeParâmetros = 1;
            for (Iterator it = mapaDeDados.lerColunas( ); it.hasNext( ); ) {
                MapaDeColunas col = (MapaDeColunas) it.next( );
                dec.setObject (contadorDeParâmetros++, col.lerValor(obj));
            }
            dec.setLong(contadorDeParâmetros, obj.lerID( ).longValue( ));
            dec.executeUpdate( );
            } catch (SQLException e) { throw new ApplicationException (e);
            } finally {DB.cleanUp(dec);
            }
    }

class MapaDeDados...

    public String listaDeAtualização ( ) {
        StringBuffer resultado = new SttringBuffer(" SET ");
        for (Iterator it = mapasDeColunas.iterator( ); it.hasNext( ); ) {
            MapaDeColunas mapaDeColunas = (MapaDeColunas) it.next( );
            resultado.append(mapaDeColunas.lerNomeDaColuna( ));
            resultado.append("=?,");
        }
        resultado.setLength(resultado.length( ) - 1);
        return resultado.toString( );
    }
    public Iterator lerColunas( ) {
        return Collections.unmodifiableCollection(mapasDeColunas).iterator( );
    }

class MapaDeColunas...

    public Object lerValor(Object sujeito) {
        try {
            return campo.get(sujeito);
        }catch (Exception e) {
            throw new ApplicationException (e);
        }
    }
```

Inserções usam um esquema semelhante.

```
class Mapeador...

    public Long inserir (ObjetoDoDomínio obj) {
        String sql = "INSERT INTO " + mapaDeDados.lerNomeDaTabela( ) + " VALUES
                    (?" + mapaDeDados.listaDeInserção( ) + ")";
        PreparedStatement dec = null;
        try {
```

```
            dec = DB.prepare(sql);
            dec.setObject (1, obj.lerID( ));
            int contadorDeParâmetros = 2;
            for (Iterator it = mapaDeDados.lerColunas( ); it.hasNext( ); ) {
                MapaDeColunas col = (MapaDeColunas) it.next( );
                dec.setObject (contadorDeParâmetros++, col.lerValor(obj));
            }
            dec.executeUpdate( );
        } catch (SQLException e) { throw new ApplicationException (e);
        } finally {DB.cleanUp(dec);
        }
        return obj.lerID( );
    }
```

```
class mapaDeDados...

    public String listaDeInserção ( ) {
        StringBuffer resultado = new StringBuffer( );
        for (int i = 0; i < mapasDeColunas.size( ); i++) {
            result.append(",");
            result.append("?");
        }
        return resultado.toString( );
    }
```


Buscas Multiobjeto Há alguns caminhos que você pode seguir para obter diversos objetos com uma única consulta. Se você quiser uma capacidade de consulta genérica no mapeador genérico, você pode ter uma consulta que receba uma cláusula SQL *where* como parâmetro.

```
class Mapeador...

    public Set encontrarObjetosOnde (String cláusulaWhere) {
        String sql = "SELECT" + mapaDeDados.listaDeColunas( ) + " FROM "+ mapaDeDados.
            lerNomeDaTabela( ) + " WHERE " + cláusulaWhere;
        PreparedStatement dec = null;
        ResultSet rs = null;
        Set resultado = new HashSet( );
        try {
            dec = DB.prepare(sql);
            rs = dec.executeQuery( );
            resultado = carregarTudo(rs);
        } catch (Exception e) {
            throw new ApplicationException (e);
        } finally {DB.cleanUp(dec, rs);
        }
    return resultado;
    }
    public Set carregarTudo (ResultSet rs) throws SQLException, InstantiationException, Il
legalAccessException {
        Set resultado = new HashSet ( );
        while (rs.next( )) {
            ObjetoDoDomínio novoObj = (ObjetoDoDomínio) mapaDeDados.lerClasseDoDomínio( ).newInstance( );
            novoObj = carregar(rs);
```

```
            resultado.add(novoObj);
        }
        return resultado;
    }
```

Uma alternativa é fornecer métodos de busca para casos especiais nos subtipos do mapeador.

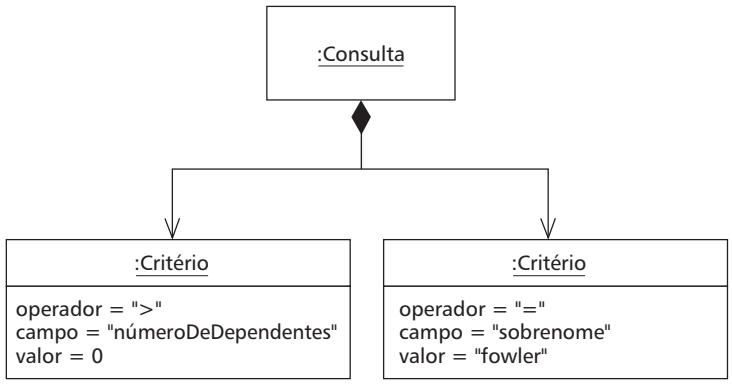
```
class MapeadordePessoa...

    public Set encontrarSobrenomesComo (String padrão) {
        String sql =
                "SELECT " + mapaDeDados.listaDeColunas( ) +
                " FROM " + mapaDeDados.lerNomeDaTabela( ) +
                " WHERE UPPER(sobrenome) like UPPER(?)";
        PreparedStatement dec = null;
        ResultSet rs = null;
        try {
            dec = DB.prepare(sql);
            dec.setString(1, padrão);
            rs = dec.executeQuery( );
            return carregarTudo(rs);
        } catch (Exception e) { throw new ApplicationException (e);
        } finally { DB.cleanUp(dec, rs);
        }
    }
```

Uma alternativa adicional para seleções gerais é um *Objeto de Pesquisa* (304).

No geral, a grande vantagem da abordagem de metadados é que posso agora adicionar novas tabelas e classes ao meu mapeamento de dados, e tudo o que tenho que fazer é fornecer um método `carregarMapaDeDados` e quaisquer métodos de busca especializados que eu possa conceber.

Objeto de Pesquisa (Query Object)

Um objeto que representa uma consulta ao banco de dados.



A SQL pode ser uma linguagem complicada, e muitos desenvolvedores não estão particularmente familiarizados com ela. Além disso, você precisa conhecer o esquema do banco de dados para formular consultas. Você pode evitar isso criando métodos de pesquisa especializados que escondam a SQL dentro de métodos parametrizados, mas isso torna difícil formular consultas mais especializadas. Isso também leva a duplicação nas declarações SQL se o esquema do banco de dados mudar.

Um *Objeto de Pesquisa* é um interpretador [Gang of Four], ou seja, uma estrutura de objetos que pode transformar-se em uma consulta SQL. Você pode criar esta consulta fazendo referência a classes e campos em vez de tabelas e colunas. Dessa maneira, aqueles que escrevem as consultas podem fazê-lo independentemente do esquema do banco de dados, e alterações no esquema podem ser localizadas em um único lugar.

Como Funciona

Um *Objeto de Pesquisa* é uma aplicação do padrão Interpretador destinado a representar uma consulta SQL. Suas principais tarefas são permitir a um cliente formular consultas de vários tipos e transformar essas estruturas de objetos na *string* SQL apropriada.

Para representar qualquer consulta, você precisa de um *Objeto de Pesquisa* flexível. Muitas vezes, no entanto, as aplicações podem se virar com muito menos do que a capacidade total do SQL, caso em que seu *Objeto de Pesquisa* pode ser mais simples. Ele não será capaz de representar todas as coisas, mas pode satisfazer as suas necessidades específicas. Além disso, normalmente, não é mais trabalhoso melhorá-lo quando for preciso mais capacidade do que é criar um *Objeto de Pesquisa* inteiramente apto desde o princípio. Como conseqüência, você deve criar um *Objeto de Pesquisa* minimamente funcional para suas necessidades correntes e aperfeiçoá-lo à medida que essas necessidades aumentarem.

Uma característica comum do *Objeto de Pesquisa* é que ele pode representar consultas na linguagem dos objetos na memória em vez da linguagem do esquema do banco de dados. Isso significa que, em vez de usar nomes de tabelas e colunas, você pode usar nomes de objetos e campos. Embora isso não seja importante se os seus ob-

jetos e banco de dados tiverem a mesma estrutura, pode ser muito útil se você tiver variações entre os dois. Para efetuar essa mudança de visão, o *Objeto de Pesquisa* precisa saber como a estrutura do banco de dados é mapeada para a estrutura dos objetos, uma habilidade que realmente precisa do *Mapeamento em Metadados* (295).

Para bancos de dados múltiplos, você pode projetar seu *Objeto de Pesquisa* de modo que ele produza *strings* SQLs diferentes dependendo do banco de dados sobre o qual a pesquisa está sendo executada. No caso mais simples, isso pode dar conta das aborrecidas diferenças na sintaxe SQL que continuam surgindo. Em um nível mais ambicioso, um Objeto de Pesquisa pode usar diferentes mapeamentos para lidar com as mesmas classes sendo armazenadas em diferentes esquemas de banco de dados.

Um uso especialmente sofisticado do *Objeto de Pesquisa* é para eliminar pesquisas redundantes sobre um banco de dados. Se você perceber que executou a mesma consulta anteriormente em uma sessão, você pode usá-la para selecionar objetos do *Mapa de Identidade* (196) evitando assim uma ida ao banco de dados. Uma abordagem mais sofisticada pode detectar se uma pesquisa é um caso particular de uma pesquisa anterior, tal como uma consulta que é idêntica a uma consulta anterior a menos de uma cláusula adicional acrescentada por um "AND".

A maneira exata de obter essas características mais sofisticadas está além do escopo deste livro, mas elas são o tipo de característica que as ferramentas de mapeamento O/R podem fornecer.

Uma variação do *Objeto de Pesquisa* consiste em permitir que uma consulta seja especificada por um exemplo de objeto de domínio. Assim, você poderia ter um objeto pessoa cujo sobrenome fosse gravado como Fowler, mas todos os outros atributos fossem gravados como nulos. Você poderia tratá-lo como um exemplo de consulta o qual é processado como o *Objeto de Pesquisa* do estilo Interpretador. Isso retorna todas as pessoas no banco de dados cujo sobrenome seja Fowler, e é muito simples e conveniente de usar. Entretanto, isso não funciona com pesquisas complexas.

Quando Usá-lo

Os *Objetos de Pesquisa* consistem em um padrão bastante sofisticado para construir, então, a maioria dos projetos não os usa, se eles tiverem uma camada de fonte de dados criada à mão. Você só precisa realmente deles quando você estiver usando um *Modelo de Domínio* (126) e um *Mapeador de Dados* (170). Você realmente também precisa de um *Mapeamento em Metadados* (295) para fazer um uso sério dos *Objetos de Pesquisa*.

Mesmo nesses casos os *Objetos de Pesquisa* não são sempre necessários, já que muitos desenvolvedores se sentem confortáveis com SQL. Você pode esconder muitos dos detalhes do esquema do banco de dados atrás de métodos específicos de busca.

As vantagens do *Objeto de Pesquisa* vêm com necessidades mais sofisticadas: manter esquemas de bancos de dados encapsulados, suportar múltiplos bancos de dados, suportar múltiplos esquemas e otimizar para evitar múltiplas consultas. Alguns projetos com uma equipe de fonte de dados particularmente sofisticada poderia querer construir ela mesma estas capacidades, mas a maior parte das pessoas que usam *Objeto de Pesquisa* o fazem com uma ferramenta comercial. Tendo a achar que quase sempre você fica melhor comprando uma ferramenta.

Dito tudo isto, você pode descobrir que um *Objeto de Pesquisa* limitado satisfaz suas necessidades, e não é difícil construir em um projeto que não justifique uma versão integral do mesmo. O artifício aqui é reduzir a funcionalidade a não mais do aquilo que você realmente usa.

Leitura Adicional

Você pode encontrar um exemplo de *Objeto de Pesquisa* em [Alpert *et al.*] na discussão dos interpretadores. O *Objeto de Pesquisa* também está intimamente ligado ao padrão Especificação em [Evans e Fowler] e [Evans].

Exemplo: Um Objeto de Pesquisa Simples (Java)

Este é um exemplo simples de um Objeto de Pesquisa – na verdade menos do que seria útil na maior parte das situações, mas o suficiente para lhe dar uma idéia do que é um *Objeto de Pesquisa*. Ele pode pesquisar uma única tabela baseado em um conjunto de critérios ligados por um conectivo "AND" (em uma linguagem ligeiramente mais técnica, ele pode tratar uma conjunção de predicados elementares).

O *Objeto de Pesquisa* é configurado usando a linguagem dos objetos do domínio em vez daquela da estrutura da tabela. Assim, uma pesquisa sabe a que classe ela pertence e um conjunto de critérios que correspondem às cláusulas de uma cláusula *where*.

```
class ObjetoDePesquisa...

    private Class classe;
    private List critério = new ArrayList( );
```

Um critério simples é o que pega um campo e um valor e um operador SQL para compará-los.

```
class Critérios...

    private String operadorSql;
    protected String campo;
    protected Object valor;
```

Para tornar mais fácil criar os critérios corretos, posso fornecer um método de criação adequado.

```
class Critérios...

    public static Critérios maiorQue (String nomeDoCampo, int valor) {
        return Critérios.maiorQue (nomeDoCampo, new Integer(valor));
    }
    public static Critérios maiorQue (String nomeDoCampo, Object valor) {
        return new Critérios(" > ", nomeDoCampo, valor);
    }
    private Critérios (String sql, String campo, Object valor) {
        this.operadorSql = sql;
        this.campo = campo;
        this.valor = valor;
    }
```

Isso me permite encontrar todas as pessoas com dependentes formando uma pesquisa tal como

```
class Critérios...

    ObjetoDePesquisa pesquisa = new ObjetoDePesquisa (Pessoa.class);
    pesquisa.adicionarCritérios (Critérios.maiorQue("númeroDeDependentes", 0));
```

Assim, se eu tiver um objeto pessoa como este:

```
class Pessoa...

    private String sobrenome;
    private String prenome;
    private int númeroDeDependentes;
```

Posso solicitar todas as pessoas com dependentes, criando uma pesquisa para pessoas e adicionando um critério.

```
ObjetoDePesquisa pesquisa = new ObjetoDePesquisa (Pessoa.class);
pesquisa.adicionarCritérios (Critérios.maiorQue("númeroDeDependentes", 0));
```

Isso é suficiente para descrever a pesquisa. Agora essa pesquisa precisa ser executada transformando-se a si mesma em um *select* SQL. Neste caso, parto do princípio de que minha classe mapeadora suporta um método que busque objetos baseado em uma *string* que corresponda a uma cláusula *where*.

```
class ObjetoDePesquisa...

    public Set executar (UnidadeDeTrabalho udt) {
        this.udt = udt;
        return udt.lerMapeador (classe).encontrarObjetosOnde (geraCláusulaWhere( ));
    }

class Mapeador...

    public Set encontrarObjetosOnde (String cláusulaWhere) {
        String sql = "SELECT" + mapaDeDados.listaDeColunas( ) + " FROM " + mapaDeDados.
            lerNomeDaTabela( ) + " WHERE " + cláusulaWhere;
        PreparedStatement dec = null;
        ResultSet rs = null;
        Set resultado = new HashSet( );
        try {
            dec = DB.prepare(sql);
            rs = dec.executeQuery( );
            resultado = carregarTudo(rs);
        } catch (Exception e) {
            throw new ApplicationException (e);
        } finally {DB.cleanUp(dec, rs);
        }
    return resultado;
    }
```

Aqui estou usando uma *Unidade de Trabalho* (187) que armazena mapeadores indexados pela classe e um mapeador que usa *Mapeamento em Metadados* (295). O código é igual àquele do exemplo do *Mapeamento em Metadados* (295), para evitar a repetição de código nesta seção.

Para gerar a cláusula *where*, a consulta percorre cada um dos critérios e faz com que cada um imprima a si próprio, juntando-os com um conectivo "AND".

```
class ObjetoDePesquisa...

    private String gerarCláusulaWhere ( ) {
```

```
        StringBuffer resultado = new StringBuffer ( );
        for (Iterator it = critérios.iterator( ); it.hasNext( ); ) {
            Critérios c = (Critérios) it.next( );
            if (resultado.length( ) !=0)
                resultado.append(" AND ");
            resultado.append("c.gerarSql (udt.lerMapeador(classe).lerMapaDeDados( )));
        }
        return resultado.toString( );
    }

class Critérios...

    public String gerarSql (MapaDeDados mapaDeDados) {
        return mapaDeDados.lerColunaParaCampo(campo) + operadorSql + valor;
    }

class MapaDeDados...

    public String lerColunaParaCampo (String nomeDoCampo) {
        for (Iterator it = lerColunas( ); it.hasNext( ); ) {
            MapaDeColunas mapaDeColunas = (MapaDeColunas) it.next( );
            if (mapaDeColunas.lerNomeDoCampo( ).equals(nomeDoCampo))
                return mapaDeColunas.lerNomeDaColuna( );
        }
        throw new ApplicationException ("Não foi encontrada a coluna para " + nomeDoCampo);
    }
```


Assim como critérios com operadores SQL simples, podemos criar classes de critérios mais complexas que façam um pouco mais. Considere uma busca por um padrão de letras que não diferencie letras maiúsculas de minúsculas, como a que busca todas as pessoas cujos sobrenomes comecem com a letra F. Podemos formar um objeto de pesquisa para encontrar todas as pessoas com tais dependentes.

```
ObjetoDePesquisa pesquisa = new ObjetoDePesquisa(Pessoa.class);
pesquisa.adicionarCritérios(Critérios.maiorQue("númeroDeDependentes", 0));
pesquisa.adicionarCritérios(Critérios.éIgual("sobrenome", "f%"));
```

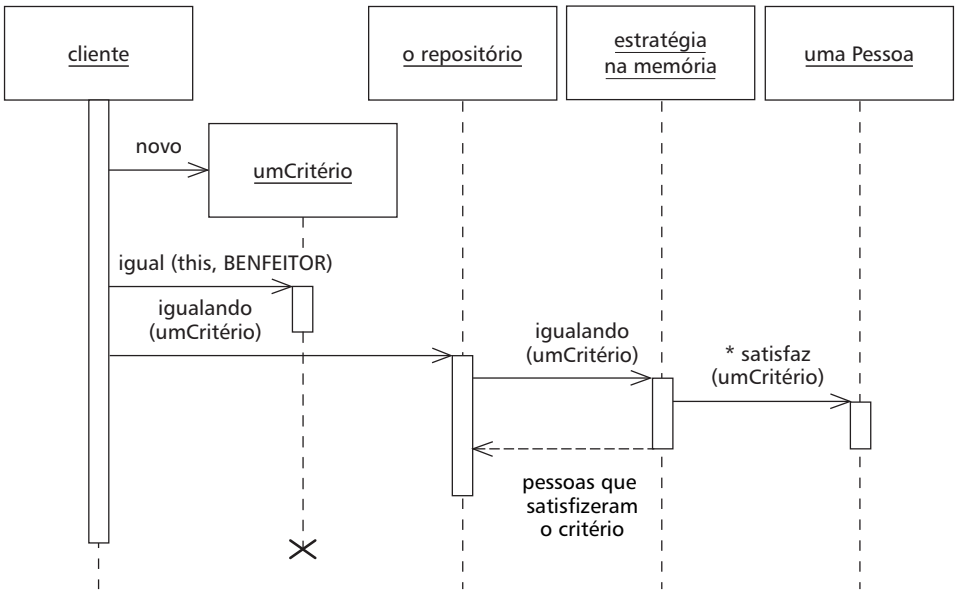
O *Objeto de Pesquisa* usa agora uma classe de critérios diferente que forma uma cláusula mais complexa na declaração *where*.

```
class Critérios...

    public static Critérios éIgual (String nomeDoCampo, String padrão) {
        return new CritérioDeIgualdade(nomeDoCampo, padrão);
    }

class CritérioDeIgualdade extends Critérios...

    public String gerarSql (MapaDeDados mapaDeDados) {
        return "UPPER(" + mapaDeDados.lerColunaParaCampo(campo) + ") LIKE UPPER('" + valor + "')";
    }
```

Repositório (Repository)

por Edward Hieatt e Rob Mee

Faz a mediação entre as camadas de domínio e de mapeamento de dados usando uma interface de tipo coleção para acessar objetos do domínio.



Um sistema com um modelo de domínio complexo muitas vezes se beneficia de uma camada, como a fornecida pelo *Mapeador de Dados* (170), que isola os objetos do domínio dos detalhes do código para acesso ao banco de dados. Em tais sistemas, pode valer a pena construir uma camada adicional de abstração sobre a camada de mapeamento em que o código de criação de consultas fica concentrado. Isso se torna mais importante quando há um grande número de classes do domínio ou consultas pesadas. Nestes casos, em especial, a adição desta camada ajuda a minimizar a lógica de consulta duplicada.

Um *Repositório* realiza a mediação entre as camadas mapeadoras de dados e as camadas de domínio, agindo como uma coleção de objetos de domínio em memória. Os objetos clientes criam especificações de consultas declarativamente e as submetem ao *Repositório* para que sejam satisfeitas. Objetos podem ser adicionados e removidos do *Repositório*, assim como o podem ser de uma simples coleção de objetos, e o código de mapeamento encapsulado pelo *Repositório* executará as operações apropriadas por trás dos panos. Conceitualmente, um *Repositório* encapsula o conjunto de objetos persistidos e as operações executadas sobre eles, fornecendo uma visão mais orientada a objetos da camada de persistência. O *Repositório* também dá suporte ao objetivo de alcançar uma separação limpa e uma dependência unidirecional entre as camadas de domínio e as camadas mapeadoras de dados.

Como Funciona

Repositório é um padrão sofisticado que faz uso de um número razoável de outros padrões descritos neste livro. De fato, ele se parece com um pequeno banco de dados orientado a objetos e, neste sentido, ele é semelhante ao *Objeto de Pesquisa* (304), o qual é mais provável de ser encontrado pelas equipes de desenvolvimento em uma ferramenta comercial para o mapeamento objeto-relacional do que construir um elas mesmas. Entretanto, se uma equipe aceitou o desafio e construiu ela mesma um *Objeto de Pesquisa* (304), não é um passo muito grande adicionar a capacidade de um *Repositório*. Quando usado em conjunto com o *Objeto de Pesquisa* (304), o *Repositório* acrescenta sem muito esforço, uma grande quantidade de aplicabilidade à camada de mapeamento objeto-relacional.

Apesar de toda o mecanismo por trás dos panos, o *Repositório* apresenta uma interface simples. Os clientes criam um objeto critérios especificando as características dos objetos que eles querem que sejam retornados por uma pesquisa. Por exemplo, para encontrar objetos pessoa pelo nome, primeiro criamos um objeto critérios, configurando cada critério individual apropriadamente: `critérios.éIgual(Pessoa.SOBRENOME, "Fowler")` e `critérios.como(Pessoa.PRENOME, "M")`. Então chamamos `repositório.atendendo(critérios)` para retornar uma lista de objetos do domínio representando pessoas com o sobrenome Fowler e um prenome começando com M. Diversos métodos convenientes semelhantes a atendendo (critérios) podem ser definidos em um repositório abstrato. Por exemplo, quando apenas uma combinação é esperada `únicoAtendendo(critérios)` poderia retornar o objeto encontrado em vez de uma coleção. Outros métodos comuns incluem `peloIDdoObjeto(id)`, o qual pode ser implementado trivialmente usando `únicoAtendendo`.

Para o código que usa um *Repositório*, ele se parece com uma coleção simples de objetos do domínio em memória. O fato dos objetos do domínio geralmente não serem armazenados diretamente no *Repositório* não é exposto ao código cliente. É claro que o código que usa o *Repositório* deve estar ciente de que esta coleção aparente de objetos pode muito bem mapear uma tabela produto com centenas de milhares de registros. Invocar o método `todos( )` no `RepositórioDeProdutos` de um sistema de catálogo pode não ser uma boa idéia.

O *Repositório* substitui métodos especializados de busca em classes do tipo *Mapeador de Dados* (170) por uma abordagem baseada em especificações para a seleção de objetos [Evans e Fowler]. Compare isso com o uso direto do *Objeto de Pesquisa* (304), no qual o código no cliente pode construir um objeto critérios (um exemplo simples do padrão especificação), `adicionar( )` o mesmo diretamente ao *Objeto de Pesquisa* (304) e executar a consulta. Com um *Repositório*, o código cliente constrói os critérios e então os passa para o *Repositório*, pedindo a ele para selecionar dentre os seus objetos, aqueles que satisfaçam aos critérios. Da perspectiva do código do cliente, não existe a noção de "execução" de uma consulta. Em vez disso, ocorre a seleção dos objetos apropriados por meio da "satisfação" da especificação da pesquisa. Isso pode parecer uma distinção acadêmica, mas ilustra o sabor declarativo da interação dos objetos com o *Repositório*, o que é uma parte signgificativa do seu poder conceitual.

Por trás dos panos, o *Repositório* combina o *Mapeamento em Metadados* (295) com um *Objeto de Pesquisa* (304) para gerar automaticamente código SQL a partir dos critérios. Se os critérios sabem como adicionar a si mesmos a uma consulta, se o *Objeto de Pesquisa* (304) sabe como incorporar objetos critérios, ou se o próprio *Mapeamento em Metadados* (295) controla a interação, é um detalhe de implementação.

A fonte de objetos para o *Repositório* não precisa necessariamente ser um banco de dados relacional, o que é bom pois o *Repositório* é bastante apropriado para a substituição do componente de mapeamento de dados por objetos de estratégia especializados. Por esta razão, ele pode ser especialmente útil em sistemas com múltiplos esquemas de bancos de dados ou múltiplas fontes de objetos de domínio, assim como durante os testes, quando o uso de objetos exclusivamente em memória é desejável por motivos de velocidade.

O *Repositório* pode ser um bom mecanismo para melhorar a legibilidade e a clareza em um código que faz uso extensivo de consultas. Por exemplo, um sistema baseado em um navegador, contendo muitas páginas de pesquisa, precisa de um mecanismo perfeito para transformar objetos `HttpRequest` em resultados de consultas. O código manipulador da solicitação pode, geralmente, sem muita complicação, converter o `HttpRequest` em um objeto critérios. Submeter os critérios ao *Repositório* apropriado deve requerer apenas uma linha ou duas de código adicional.

Quando Usá-lo

Em um sistema grande, com muitos tipos de objetos de domínio e muitas pesquisas possíveis, o *Repositório* reduz a quantidade de código necessário para lidar com todas as consultas que ocorrem. O *Repositório* promove o padrão Especificação (na forma do objeto critérios, nos exemplos mostrados aqui), o qual encapsula a consulta a ser realizada de uma forma puramente orientada a objetos. Portanto, todo o código para configurar um *Objeto de Pesquisa* em casos específicos pode ser removido. Os clientes nunca precisam pensar em SQL e podem escrever código puramente em termos de objetos.

Entretanto, em situações com diversas fontes de dados, é onde realmente vemos o *Repositório* ter sua oportunidade de mostrar o quanto é útil. Suponha, por exemplo, que às vezes estejamos interessados em usar um armazenamento de dados simples em memória, comumente quando queremos executar um conjunto de testes de unidade inteiramente em memória para obter melhor desempenho. Sem o acesso a um banco de dados, muitos conjuntos de testes longos rodam significativamente mais rápido. Criar um artefato para testes de unidade pode também ser mais direto se tudo o que tivermos que fazer for criar alguns objetos do domínio e atirá-los em uma coleção em vez de ter que gravá-los no banco de dados no início e excluí-los no final.

Também é possível, quando a aplicação estiver rodando normalmente, que certos tipos de objetos do domínio devam sempre ser armazenados em memória. Um exemplo de tal situação são os objetos de domínio imutáveis (aqueles que não podem ser alterados pelo usuário) os quais, uma vez em memória, devem permanecer lá e nunca ser trazidos novamente. Como veremos mais adiante neste capítulo, uma extensão simples do padrão *Repositório* permite que diferentes estratégias de consulta sejam empregadas dependendo da situação.

Outro exemplo, no qual o *Repositório* poderia ser útil, é quando um alimentador de dados é usado como fonte de objetos do domínio – digamos, um fluxo XML pela Internet, talvez usando SOAP, poderia estar disponível como fonte. Uma `EstratégiaDeRepositórioAlimentadaPorXML` poderia ser implementada para ler dados do fluxo e criar objetos do domínio a partir do XML.

Leitura Adicional

O padrão Especificação não tem ainda uma boa fonte de referência. A melhor descrição publicada até o momento é [Evans e Fowler]. Uma descrição melhor está atualmente sendo preparada por [Evans].

Exemplo: Encontrando os Dependentes de uma Pessoa (Java)

Da perspectiva do objeto cliente, usar o *Repositório* é simples. Para recuperar seus dependentes do banco de dados, um objeto pessoa cria um objeto critérios representando os critérios de busca e os envia ao *Repositório* apropriado.

```
public class Pessoa {

    public List dependentes ( ) {
        Repositório repositório = Registro.repositórioPessoa( );
        Critérios critérios = new Critérios( );
        critérios.igual(Pessoa.BENFEITOR, this);
        return repositório.satisfazendo(critérios);
    }
}
```


Pesquisas comuns podem ser supridas com subclasses especializadas do *Repositório*. No exemplo anterior, poderíamos criar uma subclasse de *Repositório*, `RepositórioPessoa`, e mover a criação do critério de pesquisa para o próprio *Repositório*.

```
public class RepositórioPessoa extends Repositório {
    public List dependentesDe (Pessoa umaPessoa) {
        Critérios critérios = new Critérios( );
        critérios.igual(Pessoa.BENFEITOR, umaPessoa);
        return satisfazendo(critérios);
    }
}
```

O objeto pessoa então chama o método `dependentes( )` diretamente no seu *Repositório*.

```
public class Pessoa {

    public List dependentes( ) {
        return Registro.repositórioPessoa( ).dependentesDe(this);
    }
}
```

Exemplo: Trocando Estratégias de Repositório (Java)

Uma vez que a interface do *Repositório* isola a camada do domínio da fonte de dados, podemos refatorar a implementação do código da consulta dentro do *Repositório* sem alterar quaisquer chamadas dos clientes. De fato, o código do domínio não precisa se preocupar com a fonte ou com o destino dos objetos do domínio. No caso do armazenamento em memória, queremos alterar o método `satisfazendo( )` para selecionar, a partir de uma coleção, os objetos do domínio que satisfaçam os critérios. Entretan-

to, não estamos interessados em alterar permanentemente a fonte de dados usada, mas, em vez disso, em poder escolher à vontade entre diversas fontes de dados. Vem daí a necessidade de alterar a implementação do método `satisfazendo( )` para delegar a um objeto estratégia o qual executa a pesquisa. O poder disso, é claro, é que podemos ter diversas estratégias e podemos estabelecer a estratégia conforme desejado. No nosso caso, é apropriado ter duas delas: uma `EstratégiaRelacional`, que busca no banco de dados, e uma `EstratégiaEmMemória`, que busca na coleção de objetos do domínio em memória. Cada estratégia implementa a interface `EstratégiaDeRepositório`, que expõe o método `satisfazendo( )`, de modo que obtemos a seguinte implementação da classe *Repositório*:

```
abstract class Repositório {
    private EstratégiaDeRepositório estratégia;
    protected List satisfazendo (Critérios algunsCritérios) {
        return estratégia.satisfazendo(algunsCritérios);
    }
}
```

Uma `EstratégiaRelacional` implementa `satisfazendo( )` criando um *Objeto de Pesquisa* a partir dos critérios e, então, pesquisando o banco de dados usando este *Objeto de Pesquisa*. Podemos configurá-lo com os campos e valores apropriados conforme definidos pelos critérios, pressupondo aqui que o Objeto de Pesquisa sabe como povoar a si próprio a partir dos critérios.

```
public class EstratégiaRelacional implements EstratégiaDeRepositório {
    protected List satisfazendo(Critérios critérios) {
        Pesquisa pesquisa = new Pesquisa (classeDoMeuObjetoDoDomínio( ))
        pesquisa.adicionarCritérios(critérios);
        return pesquisa.executar(unidadeDeTrabalho( ));
    }
}
```

Uma `EstratégiaEmMemória` implementa percorrendo a coleção de objetos do domínio e perguntando aos critérios para cada objeto do domínio se estes são satisfeitos por este objeto. Os critérios podem implementar o código que valida os objetos usando reflexão para inquirir os objetos do domínio sobre os valores de campos específicos. O código para fazer a seleção se parece com este:

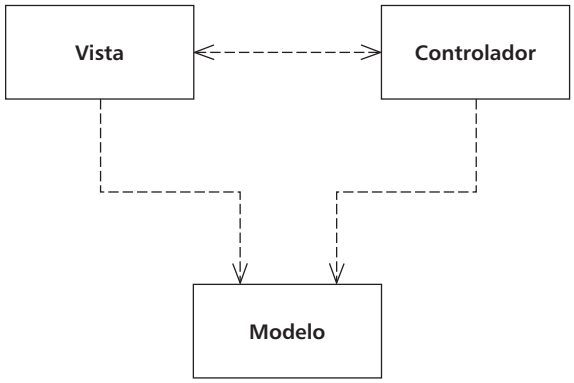
```
public class EstratégiaEmMemória implements EstratégiaDeRepositório {
    private Set objetosDoDomínio;
    protected List satisfazendo (Critérios critérios) {
        List resultados = new ArrayList( );
        Iterator it = objetosDoDomínio.iterator( );
        while (it.hasNext( )) {
            ObjetoDoDomínio cada = (ObjetoDoDomínio) it.next( );
            if (critérios.sãoSatisfeitosPor(cada))
                resultados.add(cada);
        }
         return resultados;
    }
}
```

CAPÍTULO 14

Padrões de Apresentação Web

Modelo Vista Controlador (Model View Controller)

Divide a interação da interface com o usuário em três papéis distintos.



O *Modelo Vista Controlador* (MVC) é um dos padrões mais citados (e mais citados indevidamente). Ele começou como um *framework* desenvolvido por Trygve Reenskaug para a plataforma Smalltalk no final dos anos 70. Desde então ele tem exercido um papel de influência na maioria dos *frameworks* para a interface com o usuário e no pensar sobre o projeto de interfaces com o usuário.

Como Funciona

O MVC considera três papéis. O modelo é um objeto que representa alguma informação sobre o domínio. É um objeto não-visual contendo todos os dados e comportamento que não os usados pela interface de usuário. Na sua forma OO mais pura, o modelo é um objeto dentro de um *Modelo de Domínio* (126). Você também poderia pensar em um *Roteiro de Transação* (120) como o modelo, desde que ele não contenha nenhum mecanismo de interface com o usuário. Tal definição amplia a noção de modelo, mas se adapta à divisão de papéis do MVC.

A vista representa a exibição do modelo na interface com o usuário. Assim, se nosso modelo for um objeto cliente nossa vista poderia ser um *frame* cheio de controles para a interface com o usuário ou uma página HTML com informações do modelo. A vista diz respeito apenas à apresentação de informações, quaisquer alterações nessas informações são manipuladas pelo terceiro membro da tríade MVC: o controlador. O controlador recebe a entrada do usuário, manipula o modelo e faz com que a vista seja atualizada apropriadamente. Dessa forma, a interface de usuário é uma combinação da vista e do controlador.

Quando penso sobre o MVC, vejo duas separações principais: separar a apresentação do modelo e separar o controle da vista.

Destas, **separar a apresentação do modelo** é uma das heurísticas mais fundamentais do bom projeto de *software*. Esta separação é importante por diversas razões.

- Fundamentalmente, a apresentação e o modelo referem-se a preocupações diferentes. Quando você está desenvolvendo uma vista, você está pensando nos mecanismos de interface com o usuário e em como construir uma boa in-

terface com o usuário. Quando você está trabalhando com um modelo, você está pensando em políticas de negócio, talvez em interações com bancos de dados. Certamente você usará bibliotecas diferentes, muito diferentes, ao trabalhar com a vista ou com o modelo. Muitas vezes, as pessoas preferem uma área à outra e se especializam em um dos lados da linha.

- Dependendo do contexto, os usuários querem ver as mesmas informações básicas do modelo de diferentes formas. Separar o modelo da vista permite que você desenvolva diversas apresentações – de fato, interfaces completamente diferentes – e ainda use o mesmo código do modelo. O mais notável é que isso poderia estar fornecendo o mesmo modelo com um cliente rico, um navegador Web, uma API remota e uma interface em linha de comando. Mesmo dentro de uma única interface Web você poderia ter diferentes páginas de clientes em diferentes pontos de uma aplicação.
- Os objetos não-visuais são geralmente mais fáceis de testar do que os visuais. Separar a apresentação do modelo permite que você teste facilmente toda a lógica do domínio, sem recorrer a coisas tais como complicadas ferramentas de roteiros (*scripting*) para a interface com o usuário.

Um ponto-chave nesta separação é a direção das dependências: a apresentação depende do modelo, mas o modelo não depende da apresentação. As pessoas programando o modelo não deveriam ter conhecimento algum de qual apresentação está sendo usada, o que ao mesmo tempo simplifica a sua tarefa e torna mais fácil a adição de novas apresentações mais tarde. Isso também significa que as alterações na apresentação podem ser feitas livremente sem alterar o modelo.

Este princípio introduz uma questão comum. Com uma interface de cliente rico com diversas janelas, é provável que existam diversas apresentações de um modelo em uma tela de uma só vez. Se um usuário fizer uma alteração no modelo a partir de uma apresentação, as outras precisam mudar também. Para fazer isso sem criar uma dependência, você normalmente precisa de uma implementação do padrão Observer [Gang of Four], tal como propagação de evento ou um *listener*. A apresentação atua como o observador do modelo: toda vez que o modelo muda, ele dispara um evento e as apresentações atualizam a informação.

A segunda divisão, a **separação da vista e do controlador**, é menos importante. De fato, a ironia é que quase todas as versões de Smalltalk não fizeram realmente uma separação vista/controlador. O exemplo clássico de por que você iria querer separá-los é para suportar comportamento editável e não-editável, o que você pode fazer com uma vista e dois controladores para os dois casos, em que os controladores são estratégias [Gang of Four] para a vista. Na prática, entretanto, a maioria dos sistemas tem apenas um controlador por vista, de modo que esta separação geralmente não é feita. Ela voltou a estar em voga com interfaces Web nas quais ela se torna útil para novamente separar o controlador e a vista.

O fato da maioria dos *frameworks* para a interface com o usuário combinar vista e controle levou a muitas citações errôneas do padrão MVC. O modelo e a vista são óbvios, mas onde está o controle? A idéia comum é que ele se situa entre o modelo e a vista, como no *Controlador de Aplicação* (360) – o fato da palavra "controlador" ser usada em ambos os contextos não ajuda. Sejam quais forem os méritos de um *Controlador de Aplicação* (360), trata-se de algo muito diferente de um controle MVC.

Para os propósitos deste conjunto de padrões, estes princípios são realmente tudo o que você precisa saber. Se você quiser se aprofundar mais em MVC a melhor referência disponível é [POSA].

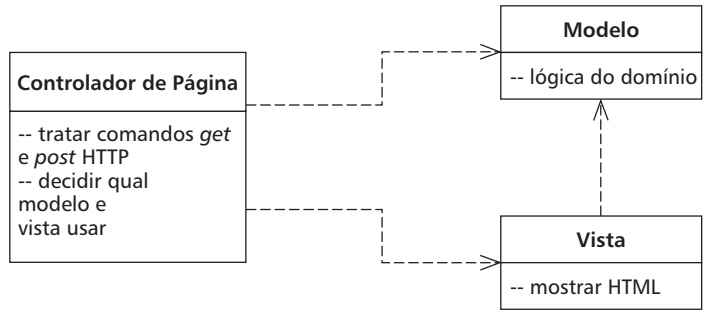
Quando Usá-lo

Como eu disse, o valor do MVC está nas suas duas separações. Destas, a separação entre a apresentação e o modelo é um dos mais importantes princípios do projeto de *software*, e a única vez em que você não deve segui-lo é em sistemas muito simples em que o modelo não tem nele nenhum comportamento real. Assim que você tiver alguma lógica não-visual, deve aplicar a separação. Infelizmente, muitos dos *frameworks* para interface com o usuário tornam isso difícil, e aqueles que não o fazem, muitas vezes, são ensinados sem enfatizar a separação.

A separação entre a vista e o controlador é menos importante, de modo que eu só a recomendaria quando fosse realmente útil. Para sistemas com clientes ricos, isso acaba sendo quase nunca, embora seja comum em *front ends* Web no qual o controle é separado. A maioria dos padrões sobre o projeto Web é baseada nesse princípio.

Controlador de Página (Page Controller)

Um objeto que trata uma solicitação para uma página ou ação específica em um site Web.



A experiência Web básica da maioria das pessoas é com páginas HTML estáticas. Quando você solicita um HTML estático, você passa para o servidor Web o nome e o caminho de um documento HTML nele armazenado. A noção-chave é que cada página em um *site* Web é um documento separado no servidor. Com páginas dinâmicas, as coisas podem ficar muito mais interessantes já que há um relacionamento muito mais complexo entre nomes de caminhos e o arquivo que atende à solicitação. Entretanto, a abordagem de um caminho levando a um arquivo que trata a solicitação é um modelo simples de entender.

O resultado é que o *Controlador de Página* tem um controlador de entrada para cada página lógica do *site* Web. Esse controlador pode ser a própria página, como freqüentemente ocorre em ambientes de páginas servidoras, ou pode ser um objeto separado que corresponda àquela página.

Como Funciona

A idéia básica por trás de um *Controlador de Página* é fazer com que um módulo no servidor Web aja como um controlador para cada página no *site* Web. Na prática, ele não funciona exatamente como um módulo por página, já que, às vezes, você pode clicar em um *link* e obter páginas diferentes, dependendo da informação dinâmica. Mais precisamente, os controladores se associam a cada ação, que pode ser clicar em um *link* ou em um botão.

O *Controlador de Página* pode ser estruturado tanto como um roteiro (*script* CGI, *servlet*, etc), quanto como uma página servidora (ASP, PHP, JSP, etc). Usar uma página servidora geralmente combina o *Controlador de Página* e uma *Vista Padrão* (333) no mesmo arquivo. Isso funciona bem para a *Vista Padrão* (333), mas não tão bem para o *Controlador de Página*, porque é mais complicado estruturar apropriadamente o módulo. Se a página for uma apresentação simples, isso não é problema. Entretanto, se houver lógica envolvida para extrair dados da solicitação ou decidir qual vista mostrar, então você pode acabar com um código *scriptlet* complicado na página servidora.

Um modo de lidar com o código *scriptlet* é usar um objeto auxiliar. Neste caso, a primeira coisa que a página servidora faz é chamar o objeto auxiliar para tratar to-

da a lógica. O objeto auxiliar pode retornar o controle para a página servidora original ou ele pode transferi-lo para uma outra página servidora diferente para que esta atue como a vista. Neste caso, a página servidora trata a solicitação, mas a maior parte da lógica do controlador fica no objeto auxiliar.

Outra abordagem é tornar o controlador e o manipulador um *script*. O servidor Web passa o controle para o *script*, este executa as responsabilidades do controlador e finalmente transfere o controle para uma vista apropriada para que esta mostre os resultados.

As responsabilidades básicas de um *Controlador de Página* são:

- Decodificar a URL e extrair quaisquer dados do formulário para ter disponíveis todos os dados para a ação.
- Criar e chamar quaisquer objetos do modelo para processar os dados. Todos os dados relevantes da solicitação HTML devem ser passados para o modelo, de modo que os objetos do modelo não precisem de nenhuma conexão com a solicitação HTML.
- Determinar qual vista deve mostrar a página de resposta e transferir a informação do modelo para ela.

O Controlador de Página não precisa ser uma única classe, mas pode chamar objetos auxiliares. Isso é particularmente útil se diversos manipuladores tiverem que executar tarefas semelhantes. Uma classe auxiliar pode então ser um bom lugar para colocar qualquer código que, de outra forma, seria duplicado.

Não há nenhum motivo para que você não possa ter algumas URLs tratadas por páginas servidoras e algumas por *scripts*. Quaisquer URLs que tenham pouca ou nenhuma lógica de controle são melhor manipuladas por uma página servidora, uma vez que ela fornece um mecanismo simples que é fácil de entender e modificar. Quaisquer URLs com lógica mais complicada vão para um *script*. Já me deparei com equipes que queriam tratar tudo da mesma forma: tudo são páginas servidoras ou tudo é um roteiro. Quaisquer vantagens de consistência em tal aplicação são geralmente contrabalançadas pelos problemas de páginas servidoras carregadas de *scriptlets* ou um número grande de *scripts* que servem apenas como uma simples passagem.

Quando Usá-lo

A principal decisão a tomar é quando usar um *Controlador de Página* ou um *Controlador Frontal* (328). Dos dois, o *Controlador de Página* é o mais rotineiro para se trabalhar e leva a um mecanismo natural de estruturação no qual ações particulares são tratadas por páginas servidoras particulares ou classes scripts. Seu compromisso é, portanto, a maior complexidade do *Controlador Frontal* (328) contra as várias vantagens do *Controlador Frontal*, a maior parte das quais faz diferença em *sites* Web que tenham maior complexidade de navegação.

O Controlador de Página funciona especialmente bem em um *site* onde a maior parte da lógica de controle seja bastante simples. Neste caso, a maioria das URLs pode ser tratada com uma página servidora e os casos mais complicados com objetos auxiliares. Quando a sua lógica de controle é simples, o *Controlador Frontal* (328) adiciona um grande *overhead*.

Não é raro encontrar um *site* onde algumas solicitações são tratadas por *Controladores de Página* e outras por *Controladores Frontais* (328), especialmente quando uma equipe estiver refatorando de uma solução para a outra. Na realidade, os dois padrões se misturam sem muito problema.

Exemplo: Apresentação Simples com um Controlador Servlet e uma Vista JSP (Java)

Um exemplo simples de um *Controlador de Página* exibe algumas informações a respeito de alguma coisa. Aqui o mostraremos exibindo algumas informações sobre um artista. A URL é executada em `http://www.thingy.com/recordingApp/artista?name=daniela-Mercury`.

O servidor Web precisa ser configurado para reconhecer **/artista** como uma chamada para o `ControladorDeArtista`. Em Tomcat, você faz isso com o seguinte código no arquivo `web.xml`:

```
<servlet>
    <servlet-name>artista</servlet-name>
    <servlet-class>actionController.ControladorDeArtista</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>artista</servlet-name>
    <url-pattern>/artista</url-pattern>
<servlet-mapping>
```

O controlador de artistas precisa implementar um método para tratar a solicitação.

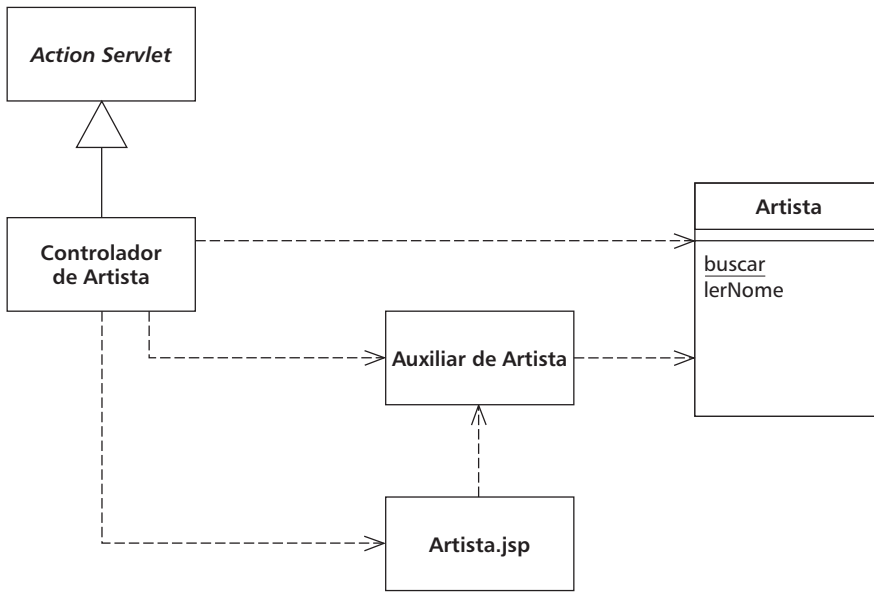


Figura 14.1 Classes envolvidas em uma exibição simples com um *servlet Controlador de Página* e uma vista JSP.

```
class ControladorDeArtista...

    public void executarLeitura (HttpServletRequest solicitação, HttpServletResponse resposta)
            throws IOException, ServletException {
        Artista artista = Artista.buscarChamado(solicitação.getParameter("nome"));
        if (artista == null)
            forward("/NomeDoArtistaFaltando.jsp", solicitação, resposta);
        else {
            solicitação.setAttribute("auxiliar", new AuxiliarDeArtista (artista));
            forward ("/artista.jsp", solicitação, resposta);
        }
    }
```

Embora este seja um caso muito simples, ele cobre os pontos mais importantes. Primeiro, o controlador precisa criar os objetos do modelo necessários para executar suas tarefas; no caso do exemplo, apenas encontrar o objeto do modelo correto para a apresentação. Em segundo lugar, ele coloca a informação correta na solicitação HTTP a fim de que a JSP possa exibi-la apropriadamente. Neste caso, ele cria um objeto auxiliar e coloca-o na solicitação. Finalmente, ele transfere o controle para a *Vista Padrão* (333) para que esta trate a apresentação. A transferência de controle é um comportamento comum, logo, ele naturalmente se situa em uma superclasse para todos os *Controladores de Página*.

```
class ActionServlet...

    protected void forward (String alvo,
                    HttpServletRequest solicitação,
                    HttpServletResponse resposta)
            throws IOException, ServletException
    {
            RequestDispatcher despachante = getServletContext( ).getRequestDispatcher(alvo);
            despachante.forward(solicitação, resposta);
    }
```


O ponto principal de acoplamento entre a *Vista Padrão* (333) e o *Controlador de Página* são os nomes dos parâmetros usados na solicitação (*request*) para passar quaisquer objetos de que a JSP precise.

A lógica do controlador aqui é realmente muito simples, mas podemos continuar a usar o *servlet* como um controlador mesmo quando ela fica mais complexa. Podemos ter um comportamento semelhante para álbuns, com a ressalva de que os álbuns clássicos têm um objeto do modelo diferente e são apresentados com uma JSP diferente. Para obter este comportamento, podemos novamente usar uma classe controladora.

```
class ControladorDeÁlbum...

    public void executarLeitura (HttpServletRequest solicitação, HttpServletResponse resposta)
            throws IOException, ServletException
    {
        Álbum álbum = Álbum.buscar (solicitação.getParameter("id"));
        if (álbum == null) {
            forward("/erroÁlbumFaltando.jsp", solicitação, resposta);
            return
    }
```

```
        solicitação.setAttribute("auxiliar", álbum);
        if (álbum instanceof ÁlbumClássico)
            forward ("/álbumClássico.jsp", solicitação, resposta);
        else
            forward ("/álbum.jsp", solicitação, resposta);
    }
```

Perceba que neste caso estou usando os objetos do modelo como auxiliares em vez de criar uma classe auxiliar separada. Vale a pena fazer isso se uma classe auxiliar for apenas um despachante burro para a classe do modelo. Porém, se você fizer isso, assegure-se de que a classe do modelo não contém qualquer código dependente do *servlet*. Qualquer código que seja dependente do *servlet* deve estar em uma classe auxiliar separada.

Exemplo: Usando uma JSP como Manipulador (Java)

Usar um *servlet* como controlador é um caminho possível, mas o caminho mais comum é fazer da própria página servidora o controlador. O problema com esta abordagem é que ela resulta em código *scriptlet* no início da página servidora, e, como você deve ter percebido, penso que o código *scriplet* tem a mesma relação com um *software* bem-projetado que o *wrestling* profissional tem com o esporte.

Apesar disso, você pode tratar solicitações com uma página servidora como um manipulador, enquanto delega o controle para o objeto auxiliar para que este efetivamente execute a função de controlador. Isso preserva a propriedade simples de ter sua URL disponibilizada por páginas servidoras. Farei isso para a exibição do álbum, usando a URL na forma `http://localhost:8080/isa/album.jsp?id=zero`. A maioria dos álbuns é apresentada diretamente com a JSP de álbum, mas gravações de clássicos requerem uma apresentação diferente, uma JSP de álbum clássico.

O comportamento deste controlador aparece em uma classe auxiliar para o JSP. A classe auxiliar é configurada na própria JSP do álbum.


```
álbum.jsp...

<jsp:useBean id="auxiliar" class="actionController.AuxiliarCtrlÁlbum"/>
<%auxiliar.init(solicitação, resposta); %>
```

A chamada ao método *init* configura o auxiliar para executar o comportamento do controlador.

```
class AuxiliarCtrlÁlbum extends ControladorAuxiliar...

    public void init(HttpServletRequest solicitação, HttpServletResponse resposta) {
        super.init(solicitação, resposta);
        if (lerÁlbum( ) == null) forward ("erroFaltandoÁlbum.jsp", solicitação, resposta);
        if (lerÁlbum( ) instanceof ÁlbumClássico) {
            solicitação.setAttribute("auxiliar", lerÁlbum( ));
            forward("/álbumClássico.jsp", solicitação, resposta);
        }
    }
```

O comportamento comum do auxiliar fica, naturalmente, em uma superclasse auxiliar.

```
class ControladorAuxiliar...

    public void init (HttpServletRequest solicitação, HttpServletResponse resposta) {
        this.solicitação = solicitação;
        this.resposta = resposta;
    }
    protected vois forward (String alvo,
                    HttpServletRequest solicitação,
                    HttpServletResponse resposta)
    {
        try {
            RequestDispatcher despachante = solicitação.getRequestDispatcher(alvo);
            if (despachante == null) resposta.sendError(resposta.SC_NO_CONTENT);
            else despachante.forward (solicitação, resposta);
        } catch (IOException e) {
            throw new ApplicationException (e);
        } catch (ServletException e) {
            throw new ApplicationException (e);
        }
    }
```

A principal diferença entre o comportamento do controlador aqui e aquele quando do uso de um *servlet* é que a JSP manipuladora também é a vista *default* e, a menos que o controlador transfira o controle para uma outra JSP, este retorna para o manipulador original. Isso é uma vantagem quando você tem páginas na qual, na maior parte do tempo, a JSP atua diretamente como a vista e, portanto, não há nenhuma transferência a ser feita. A inicialização do objeto auxiliar atua para dar a partida em qualquer comportamento do modelo e configurar as coisas para a vista mais tarde. É um modelo simples de seguir, já que as pessoas geralmente associam uma página Web com a página servidora que atua como sua vista. Freqüentemente isso também se adapta naturalmente à configuração do servidor Web.

A chamada para inicializar o manipulador é um pouco deselegante. Em um ambiente JSP essa complicação pode ser muito melhor tratada com uma etiqueta personalizada (*custom tag*). Tal etiqueta pode criar automaticamente um objeto apropriado, colocá-lo na solicitação e inicializá-lo. Com isso, tudo o que você precisa é de uma etiqueta simples na página JSP.

```
<auxiliar:init name: "actionController.AuxiliarCtrlÁlbum"/>
```

A implementação da etiqueta personalizada executa então o trabalho.

```
class EtiquetaDeInicializaçãoDoAuxiliar extends EtiquetaDoAuxiliar...

    private String nomeDaClasseAuxiliar;
    public void gravarNome (String nomeDaClasseAuxiliar) {
        this.nomeDaClasseAuxiliar = nomeDaClasseAuxiliar;
    }
    public int iniciarEtiqueta ( ) throws JspException {
        ControladorAuxiliar auxiliar = null;
        try {
            auxiliar = (ControladorAuxiliar)
            Class.forName(nomeDaClasseAuxiliar).newInstance( );
        } catch (Exception e) {
```

```
            throw new ApplicationException ("Não foi possível instanciar" + nomeDaClasseAuxiliar, e);
        }
        iniciarAuxiliar(auxiliar);
        pageContext.setAttribute(AUXILIAR, auxiliar);
        return SKIP_BODY;
    }
    private void iniciarAuxiliar (ControladorAuxiliar auxiliar) {
        HttpServletRequest solicitação = (HttpServletRequest) pageContext.getRequest( );
        HttpServletResponse resposta = (HttpServletResponse) pageContext.getResponse( );
        auxiliar.init(solicitação, resposta);
    }
```

```
class EtiquetaAuxiliar...

    public static final String AUXILIAR = "auxiliar";
```

Se eu for usar etiquetas personalizadas dessa forma, eu poderia ainda criá-las para acessar também as propriedades.

```
class EtiquetaAuxiliarDeLeitura extends EtiquetaAuxiliar...

    private String nomeDaPropriedade;
    public void gravarPropriedade (String nomeDaPropriedade) {
        this.nomeDaPropriedade = nomeDaPropriedade;
    }
    public int iniciarEtiqueta ( ) throws JspException {
        try {
            pageContext.getOut( ).print(lerPropriedade(nomeDaPropriedade));
        } catch (IOException e) {
            throw new JspException("Não foi possível imprimir em um writer");
        }
        return SKIP_BODY;
    }
```

```
class EtiquetaAuxiliar...

    protected Object lerPropriedade(String propriedade) throws JspException {
        Object auxiliar = lerAuxiliar( );
        try {
            final Method leitor = auxiliar.getClass( ).getMethod(lendoMétodo(propriedade), null);
            return leitor.invoke(auxiliar, null);
        } catch (Exception e) {
            throw new JspException
                ("Não foi possível chamar "+ lendoMétodo(propriedade) + " - " + e.getMessage( ));
        }
    }
    private Object lerAuxiliar ( ) throws JspException {
        Object auxiliar = pageContext.getAttribute(AUXILIAR);
        if (auxiliar == null) throw new JspException("Auxiliar não encontrado");
        return auxiliar;
    }
    private String lendoMétodo(String propriedade) {
        String nomeDoMétodo = "ler" + propriedade.substring(0,1).toUpperCase( ) +
                propriedade.substring(1);
        return nomeDoMétodo;
    }
```

(Você pode achar que é melhor usar o mecanismo de Java Beans do que apenas chamar um método de leitura usando reflexão. Se este for o caso, provavelmente você está certo... e, provavelmente, você também é inteligente o bastante para descobrir como alterar o método para fazer isso.)

Com a etiqueta de leitura definida, posso usá-la para extrair informações do objeto auxiliar. A etiqueta é mais curta e elimina qualquer chance de eu digitar "auxiliar" errado.

```
<B><auxiliar:get property = "título"/></B>
```

Exemplo: Manipulador de Página com um Código Por Trás (C#)

O sistema Web em .NET é projetado para trabalhar com os padrões *Controlador de Página* e *Vista Padrão* (333), ainda que você certamente possa decidir tratar eventos Web com uma abordagem diferente. Neste próximo exemplo, usarei o estilo preferido de .NET, construindo a camada de apresentação sobre um domínio usando um *Módulo Tabela* (134) e usando conjuntos de dados (*data sets*) como os principais transportadores de informação entre as camadas.

Desta vez, teremos uma página que mostra pontos marcados e a média de pontos para um turno de uma partida de críquete. Como sei que terei muitos leitores aflitos com a falta de experiência nesta forma de arte, deixe-me resumir dizendo que os pontos marcados são os pontos do rebatedor, e a média de pontos é o número de pontos que ele marca dividido pelo número de bolas arremessadas em sua direção. Os pontos marcados e as bolas arremessadas estão no banco de dados. A média de pontos precisa ser calculada pela aplicação – um pedaço de lógica de domínio pequeno, mas pedagogicamente útil.

O manipulador neste projeto é uma página Web ASP.NET, capturada em um arquivo .aspx. Assim como com outras construções de páginas servidoras, este arquivo permite que você insira lógica de programação diretamente nas páginas como *scriptlets*. Já que você sabe que eu preferiria beber cerveja ruim a escrever *scriptlets*, você sabe que há pouca chance de eu fazer isso. Meu salvador neste caso é o mecanismo de **código por trás** da ASP.NET que permite que você associe um arquivo e classe comuns à página aspx, sinalizado no cabeçalho da página aspx.

```
<%@ Page language = "C#" Codebehind="bat.aspx.cs" AutoEventWireup="false" trace="False"
    Inherits="rebatedores.PáginaDeRebatidas" %>
```

A página é configurada como uma subclasse da classe código "por trás", e desta forma pode usar todas as suas propriedades e métodos do tipo *protected*. O objeto página é o manipulador da solicitação, e o código por trás pode definir o tratamento definindo um método Carregar_Página. Se a maioria das páginas seguir um fluxo comum, posso definir uma *Camada Supertipo* (444) que tenha um método padrão [Gang of Four] para isso.

```
class PáginaDeCríquete...

    protected void Carregar_Página (object remetente, System.EventArgs e) {
        db = new OleDbConnection(DB.ConnectionString);
        if(háParâmetrosFaltando( ))
            transfereErro (mensagemDeFaltaDeParâmetros);
        DataSet ds = lerDados( );
```

```
        if(nãoHáDados(ds))
            transfereErro ("Nenhum dado corresponde à sua solicitação");
        aplicarLógicaDoDomínio(ds);
        DataBind ( );
        prepararInterfaceComUsuário(ds);
    }
```

O método padrão divide o tratamento da solicitação em vários passos comuns. Desta maneira, podemos definir um fluxo único comum para tratar solicitações Web, ao mesmo tempo em que permitimos que cada *Controlador de Página* forneça implementações para os passos específicos. Se você fizer isso, depois que tiver escrito uns poucos *Controladores de Páginas*, saberá que fluxo comum usar no método padrão. Se alguma página precisar fazer algo completamente diferente, ela sempre poderá sobrescrever o método de carga da página.

A primeira tarefa é fazer a validação dos parâmetros chegando à página. Em um exemplo mais realista, isso poderia requerer a verificação de valores de várias formas, mas neste caso estamos apenas decodificando uma URL na forma `http://localhost/rebatedores/bat.aspx?equipe=Inglaterra&período=2&jogo=905`. A única validação neste exemplo é que os vários parâmetros requeridos para a consulta ao banco de dados estejam presentes. Como de costume, tenho sido extremamente simplista na manipulação de erros até que alguém escreva um bom conjunto de padrões sobre validação – então, aqui, a página especificada define um conjunto de parâmetros obrigatórios, e a *Camada Supertipo* (444) tem a lógica para verificá-los.


```
class PáginaDeCríquete...

    abstract protected String[ ] parâmetrosObrigatórios( );
    private Boolean háParâmetrosFaltando ( ) {
        foreach (String parâmetro in parâmetrosObrigatórios( ))
            if (Request.Params[parâmetro] == null) return true;
        return false;
    }
    private String mensagemDeFaltaDeParâmetros ( ) {
        get {
            String resultado = "<p>Estão faltando parâmetros obrigatórios nesta
                                página: <p>";
            resultado += "<UL>";
            foreach (String parâmetro in parâmetrosObrigatórios( ))
                if (Request.Params[parâmetro] == null)
                    resultado += String.Format("<LI>{0}</LI>", parâmetro);
            resultado += "</UL>"
            return resultado;
        }
    }
    protected void transfereErro (String mensagem) {
        Context.Items.Add("mensagemDeErro", mensagem);
        Context.Server.Transfer("Erro.aspx");
    }

class PáginaDeRebatidas...

    override protected String[ ] parâmetrosObrigatórios ( ) {
        String[ ] resultado = {"equipe", "período", "jogo"};
        return resultado;
    }
```

O próximo estágio é extrair os dados do banco de dados e colocá-los em um objeto do tipo conjunto de dados (*data set*) ADO.NET desconectado. Aqui está uma única consulta para a tabela de rebatidas.

```
class PáginaDeCríquete...

    abstract protected DataSet lerDados( );
    protected Boolean nãoHáDados (DataSet ds) {
        foreach (DataTable tabela in ds.Tables)
            if(tabela.Rows.Count != 0) return false;
        return true;
    }

class PáginaDeRebatidas...

    override protected DataSet lerDados ( ) {
        OleDbCommand comando = new OleDbCommand(SQL, db);
        comando.Parameters.Add (new OleDbParameter("equipe", equipe));
        comando.Parameters.Add (new OleDbParameter("período", período));
        comando.Parameters.Add (new OleDbParameter("jogo", jogo));
        OleDbAdapter da = new OleDbAdapter(comando);
        DataSet resultado = new DataSet( );
        da.Fill(resultado, Rebatidas.TABLE_NAME);
        return resultado;
    }
    private const String SQL =
        @"SELECT * from rebatidas
            WHERE equipe = ? AND período = ? AND idDoJogo = ?
            ORDER BY ordemDeRebatidas";
```

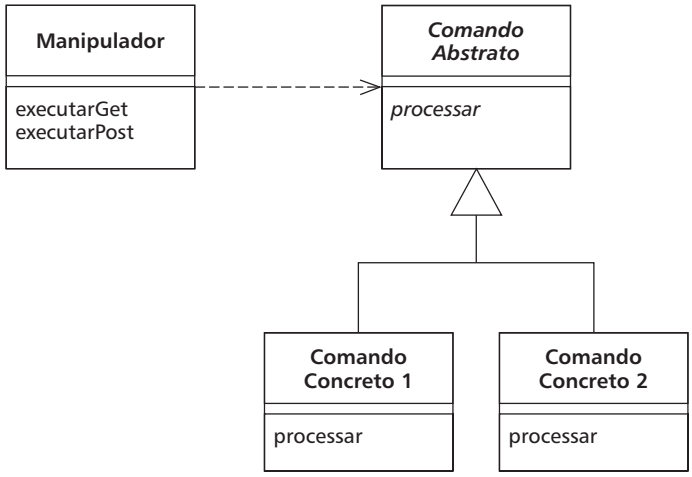
Agora é a hora da lógica do domínio, organizada como um *Módulo Tabela* (134). O controlador passa o conjunto de dados encontrado para o *Módulo Tabela* (134) para processamento.

```
class PáginaDeCríquete...

    protected virtual void aplicarLógicaDoDomínio (DataSet ds) { }

class PáginaDeRebatidas...

    override protected void aplicarLógicaDoDomínio (DataSet conjuntoDeDados) {
        rebatidas = new Rebatidas(conjuntoDeDados);
        rebatidas.CalcularMédias( );
    }
```

Neste ponto, a parte controladora do manipulador da página está terminada. Quero dizer com isso, em termos clássicos do *Modelo Vista Controle* (315), que o controlador deve agora transferir a apresentação para a vista. Neste projeto, a `PáginaDeRebatidas` atua tanto como controle quanto como vista, e a última chamada a `prepararInterfaceComUsuário`, é parte do comportamento da vista. Posso agora dizer adeus a este exemplo deste padrão. Entretanto, suspeito que você pensará que falta um certo fechamento dramático, então você poderá encontrar a continuação do exemplo mais adiante (página 333).

Controlador Frontal (Front Controller)

Um controlador que trata todas as solicitações para um site Web.



Em um *site* Web complexo, há muitas coisas semelhantes que você precisa fazer ao tratar uma solicitação. Estas coisas incluem segurança, internacionalização e fornecimento de apresentações particulares para determinados usuários. Se o comportamento do controlador de entrada for espalhado por vários objetos, muito deste comportamento pode acabar duplicado. Além disso, é difícil alterar comportamento em tempo de execução.

O *Controlador Frontal* consolida todo o tratamento de solicitações canalizando-as através de um único objeto manipulador. Este objeto pode executar o comportamento comum, o qual pode ser modificado em tempo de execução com decoradores. O manipulador então despacha para objetos do tipo comando que possuem comportamento específico relacionado a uma solicitação.

Como Funciona

Um *Controlador Frontal* trata todas as chamadas a um *site* Web, e é normalmente estruturado em duas partes: um manipulador Web e uma hierarquia de comandos. O manipulador Web é o objeto que efetivamente recebe as solicitações *post* ou *get* do servidor Web. Ele extrai apenas as informações necessárias da URL e da solicitação para decidir que tipo de ação iniciar e então delega a um objeto comando para executar a ação (veja a Figura 14.2).

O manipulador Web é quase sempre implementado como uma classe em vez de como uma página servidora, já que ele não produz nenhuma resposta. Os comandos também são classes em vez de páginas servidoras e, na verdade, não precisam de nenhum conhecimento sobre o ambiente Web, embora freqüentemente a informação HTTP seja passada a eles. O manipulador Web é, ele próprio, normalmente um programa razoavelmente simples que não faz nada além de decidir qual comando executar.

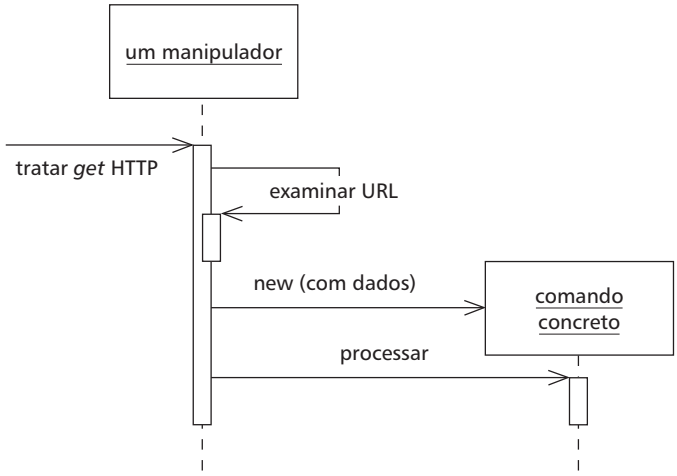


Figura 14.2 Como o *Controlador Frontal* funciona.

O manipulador Web pode decidir qual comando executar tanto estática quanto dinamicamente. A versão estática envolve analisar a URL e usar lógica condicional. A versão dinâmica envolve pegar um pedaço padrão da URL e usar instanciação dinâmica para criar uma classe comando.

O caso estático tem a vantagem da lógica explícita, verificação de erros no despacho em tempo de compilação e muita flexibilidade na aparência das suas URLs. O caso dinâmico permite a você adicionar novos comandos sem alterar o manipulador Web.

Com a invocação dinâmica você pode colocar o nome da classe comando na URL ou você pode usar um arquivo de propriedades que liga as URLs aos nomes das classes comando. O arquivo de propriedades é outro arquivo para editar, mas torna mais fácil alterar os nomes de suas classes sem ser necessária uma grande pesquisa pelas suas páginas Web.

Um padrão particularmente útil a ser usado com o *Controlador Frontal* é o *Filtro Interceptador*, descrito em [Alur *et al.*]. Este é basicamente um decorador que encapsula o manipulador do controlador frontal permitindo a você criar uma *cadeia de filtros* (ou *pipeline* de filtros) para lidar com questões tais como autenticação, *logging* e identificação de local. O uso de filtros permite que você, dinamicamente, prepare os filtros a usar em tempo de configuração.

Rob Mee mostrou-me uma variação interessante do *Controlador Frontal* usando um manipulador Web de dois estágios, dividido em um manipulador Web degenerado e em um despachante. O manipulador Web degenerado extrai os dados básicos dos parâmetros http e os passa para o despachante de tal modo que o despachante é completamente independente do *framework* do servidor Web. Isso torna os testes mais fáceis, porque o código de teste pode alimentar diretamente o despachante sem ter que rodar em um servidor Web.

Lembre-se de que tanto o manipulador quanto os comandos são parte do controlador. Em função disso, os comandos podem (e devem) escolher que vista usar para a resposta. A única responsabilidade do manipulador é escolher qual comando executar. Uma vez que isso tiver sido feito, ele participa mais dessa solicitação.

Quando Usá-lo

O *Controlador Frontal* é um projeto mais complicado do que sua contraparte óbvia, o *Controlador de Página* (318). Ele precisa portanto apresentar algumas vantagens para justificar o esforço.

Apenas um único *Controlador Frontal* tem que ser configurado no servidor Web; o manipulador Web faz o resto do trabalho de despacho (*dispatching*). Isso simplifica a configuração do servidor Web, o que é uma vantagem se ele for complicado de configurar. Com objetos comando dinâmicos, você pode adicionar novos comandos sem alterar nada. Eles também tornam mais fácil portar a aplicação, já que você tem apenas que registrar o manipulador em um modo específico do servidor Web utilizado.

Devido ao fato de você criar novos objetos comando em cada solicitação, você não tem que se preocupar em tornar as classes dos comandos seguras a *threads*. Dessa maneira, você evita as dores de cabeça da programação *multithread*, entretanto você tem de se assegurar de que não compartilha quaisquer outros objetos, tais como os objetos do modelo.

Uma vantagem comumente mencionada de um *Controlador Frontal* é que ele permite que você fatore código que, de outra forma, seria duplicado no *Controlador de Página* (318). Contudo, para ser justo, você também pode fazer muito disso com uma superclasse *Controlador de Página* (318).

Existe apenas um único controlador, então você, usando decoradores [Gang of Four], pode facilmente melhorar o seu comportamento em tempo de execução. Você pode ter decoradores para autenticação, codificação de caracteres, internacionalização, e assim por diante, e adicioná-los usando um arquivo de configuração ou, até mesmo, enquanto o servidor estiver rodando. ([Alur *et al.*] descrevem esta abordagem em detalhe sob o nome *Filtro Interceptador*.)

Leitura Adicional

[Alur *et al.*] fornecem uma descrição detalhada de como implementar o *Controlador Frontal* em Java. Eles também descrevem o *Filtro Interceptador,* que combina muito bem com o *Controlador Frontal*.

Vários *frameworks* Web Java usam este padrão. Um exemplo excelente aparece em [Struts].

Exemplo: Exibição Simples (Java)

Aqui está um caso simples de uso de *Controlador Frontal* para a tarefa original e inovadora de exibição de informações sobre um artista. Usaremos comandos dinâmicos com uma URL da forma `http://localhost:8080/isa/musica?nome=malFunciona&comando=Artista`. O parâmetro comando diz ao manipulador Web qual comando usar.

Começaremos com o manipulador, que implementei como um *servlet*.

```
class ServletFrontal...

    public void executarGet (HttpServletRequest solicitação, HttpServletResponse resposta)
            throws IOException, ServletException {
        ComandoFrontal comando = lerComando (solicitação);
        comando.init (getServletContext( ), solicitação, resposta);
        comando.processar( );
    }
```

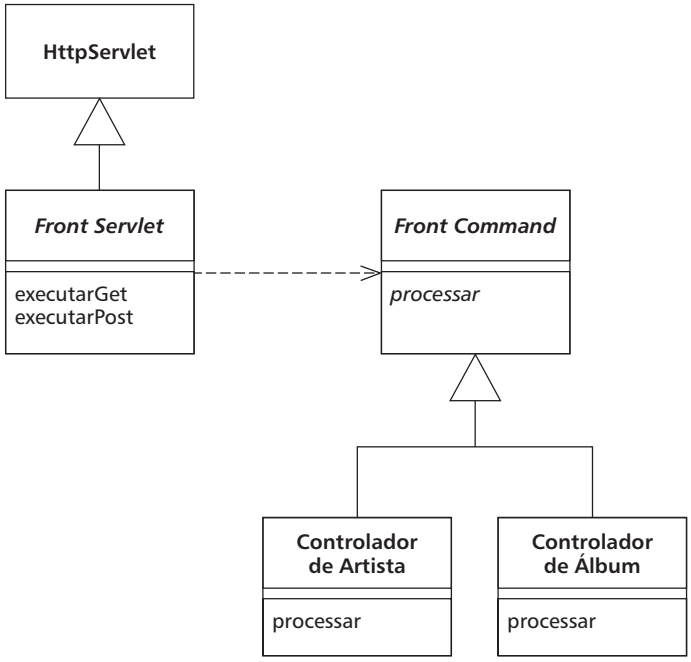


Figura 14.3 As classes que implementam o *Controlador Frontal*.


```
private ComandoFrontal lerComando (HttpServletRequest solicitação) {
    try {
        return (ComandoFrontal) lerClasseDoComando(solicitação).newInstance( );
    } catch (Exception e) {
        throw new ApplicationException (e);
    }
}
private Class lerClasseDoComando (HttpServletRequest solicitação) {
    Class resultado;
    final String nomeDaClasseDoComando =
        "controladorFrontal." + "Comando" + (String) solicitação.getParameter("comando");
    try {
        resultado = Class.forName(nomeDaClasseDoComando);
    } catch (ClassNotFoundException e) {
        resultado = UnknowCommand.class;
    }
    return resultado;
}
```

A lógica é direta. O manipulador tenta instanciar uma classe cujo nome é formado pela concatenação do nome do comando e a palavra "Comando". Assim que ele obtém o novo comando, ele o inicializa com as informações necessárias provenientes do servidor HTTP. Passei o que precisava para este exemplo simples. Você pode precisar de mais, como a sessão HTTP. Se você não conseguir encontrar um comando, usei o padrão *Caso Especial* (462) e retornei um comando desconhecido. Como é muitas ve-

zes o caso, o *Caso Especial* (462) permite que você evite muita verificação adicional de erros.

Os comandos compartilham uma quantidade razoável de dados e comportamento. Todos eles precisam ser inicializados com informações do servidor Web.

```
class ComandoFrontal...

    protected ServletContext contexto;
    protected HttpServletRequest solicitação;
    protected HttpServletResponse resposta;
    public void init(ServletContext contexto,
                   HttpServletRequest solicitação,
                   HttpServletResponse resposta )
    {
        this.contexto = contexto;
        this.solicitação = solicitação;
        this.resposta = resposta;
    }
```

Eles também podem fornecer comportamento comum, como um método para passar adiante, e definir um método abstrato de processamento para os comandos reais sobrescreverem.

```
class FrontCommand...

    abstract public void process( ) throws ServletException, IOException;
    protected forward (String alvo) throws ServletException, IOException
    {
        RequestDispatcher despachante = contexto.getRequestDispatcher(alvo);
        despachante.forward(solicitação, resposta);
    }
```


O objeto comando é muito simples, pelo menos neste caso. Ele apenas implementa o método de processamento, o que envolve chamar o comportamento apropriado nos objetos do modelo, colocando a informação necessária para a vista na solicitação e passando adiante para uma *Vista Padrão* (333).

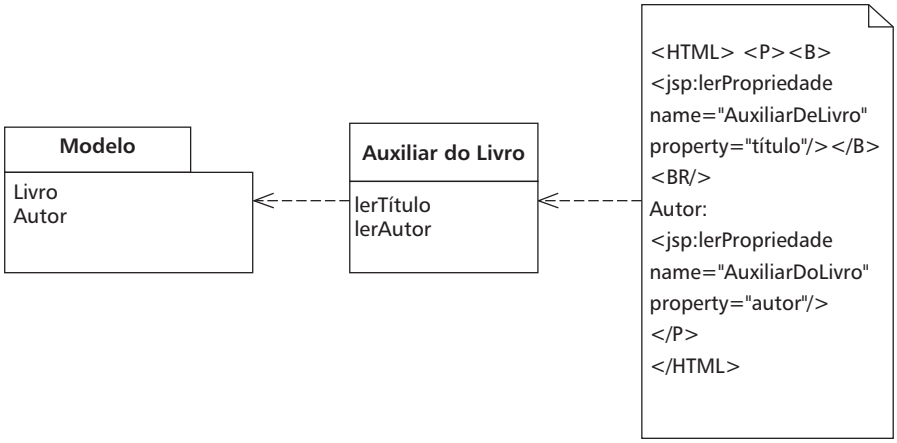
```
class ComandoArtista...

    public void process( ) throws ServletException, IOException {
        Artista artista = Artista.buscarPeloNome(solicitação.getParameter("nome"));
        solicitação.setAttribute("auxiliar", new AuxiliarDeArtista(artista));
        forward("/artista.jsp");
    }
```

O comando desconhecido apenas traz uma entediante página de erro.

```
class UnknowCommand...

    public void process( ) throws ServletException, IOException {
        forward("/desconhecido.jsp");
    }
```

Vista Padrão (Template View)

Representa informações em HTML
inserindo marcadores em uma página HTML.



Escrever um programa que produza HTML é muitas vezes mais difícil do que você poderia imaginar. Embora linguagens de programação estejam melhores na criação de texto do que costumavam ser (alguns de nós lembram a manipulação de caracteres em Fortran e Pascal padrão), criar e concatenar *strings* ainda é trabalhoso. Se não houver muito a ser feito, isso não é tão ruim, mas uma página HTML inteira é muita manipulação de texto.

Com páginas HTML estáticas – aquelas que não se alteram de solicitação para solicitação – você pode usar bons editores WYSIWYG. Mesmo aqueles que gostam de editores de texto primitivos acham mais fácil simplesmente digitar o texto e identificadores em vez de mexer com concatenação de *strings* em uma linguagem de programação.

É claro que o problema é com páginas Web dinâmicas – aquelas que pegam o resultado de algo como pesquisas em bancos de dados e as inserem em HTML. A página parece diferente em cada resultado, por isso editores HTML comuns não são apropriados para o trabalho.

A melhor maneira de trabalhar é organizar a página Web dinâmica como você faz com uma página estática, mas colocar marcadores que podem ser transformados em chamadas para juntar informações dinâmicas. Já que a parte estática da página atua como um padrão para a resposta específica, chamo isso de *Vista Padrão*.

Como Funciona

A idéia básica da *Vista Padrão* é inserir marcadores em uma página HTML estática quando ela é escrita. Quando a página é usada para atender uma solicitação, os marcadores são substituídos pelo resultado de alguma computação, como uma pesquisa em um banco de dados. Dessa forma a página pode ser mostrada do modo costumeiro, muitas vezes com editores WYSIWYG, muitas vezes por pessoas que não são programadoras. Os marcadores então se comunicam com programas reais para enviar os resultados.

Muitas ferramentas usam a *Vista Padrão*. O resultado é que este padrão não diz respeito a como construir você mesmo uma *Vista Padrão*, mas sim como usá-la eficazmente e qual é a alternativa.

Inserindo os Marcadores Há um número de maneiras pelas quais os marcadores podem ser colocados no HTML. Uma é usar etiquetas HTML-*like*. Isto funciona bem com editores WYSIWYG porque eles percebem que qualquer coisa entre os colchetes em ângulo (<>) é especial e a ignora ou a trata de maneira diferente. Se os identificadores seguirem as regras de XML bem-formada, você também pode usar ferramentas XML no documento resultante (desde que seu HTML seja *XHTML*, é claro).

Outra maneira de fazer isso é usando marcadores de texto especiais no corpo do texto. Editores WYSIWYG tratam isso como texto comum, ainda ignorando-o mas provavelmente fazendo coisas aborrecidas tais como verificação ortográfica. A vantagem é que a sintaxe pode ser mais fácil do que a sintaxe pesada de HTML/XML.

Muitos ambientes fornecem o conjunto de etiquetas que você usa, mas mais e mais plataformas agora lhe dão a habilidade de definir suas próprias etiquetas e marcadores, de modo que você pode projetá-los para se ajustarem às suas necessidades específicas.

Uma das formas mais populares de *Vista Padrão* é uma **página servidora** como ASP, JSP ou PHP. Estas realmente vão um passo à frente em relação à forma básica de uma *Vista Padrão*, visto que permitem que você insira na página lógica de programação arbitrária, referida como ***scriptlet***. Entretanto, no meu modo de ver, esta característica é em verdade um grande problema e, ao usar tecnologia de páginas servidoras, você estaria melhor limitando-se ao comportamento básico da *Vista Padrão*.

A desvantagem mais óbvia de colocar muitos *scriptlets* em uma página é que isso elimina a possibilidade de não-programadores editarem a página. Isso é especialmente importante quando você estiver usando projetistas gráficos para o projeto da página. Entretanto, os maiores problemas de inserir *scriptlets* na página decorrem do fato de que uma página é um módulo pobre para um programa. Mesmo com uma linguagem orientada a objetos a construção na página faz com que você perca a maior parte das características estruturais que tornam possível fazer um projeto modular quer seja em OO quer em um estilo procedural.

Pior ainda, colocar muitos *scriptlets* na página torna fácil demais misturar as diferentes camadas de uma aplicação corporativa. Quando a lógica do domínio começa a aparecer em páginas servidoras, torna-se muito difícil estruturá-la bem e muito fácil duplicá-la em diferentes páginas servidoras. Tudo considerado, o pior código que vi nos últimos anos foi o código de páginas servidoras.

Objeto Auxiliar A chave para evitar *scriptlets* é fornecer um objeto comum como um **auxiliar** para cada página. Este auxiliar tem toda a lógica real de programação. A página só tem chamadas para ele, o que a simplifica e a torna uma *Vista Padrão* mais pura. A simplicidade resultante permite a não-programadores editarem a página e aos programadores concentrarem-se no objeto auxiliar. Dependendo da ferramenta que você estiver usando, você muitas vezes pode reduzir todos os padrões em uma página para etiquetas HTML/XML, o que mantém a página mais consistente e mais acessível a suporte com ferramentas.

Isso soa como um princípio simples e recomendável, mas, como sempre, algumas questões tornam as coisas mais complicadas. Os marcadores mais simples são aqueles que obtêm informações do resto do sistema e as colocam no lugar correto na

página. Eles são facilmente traduzidos em chamadas para o objeto auxiliar que resultam em texto (ou algo trivialmente transformado em texto), e o mecanismo coloca o texto na página.

Exibição Condicional Uma questão mais complicada é o comportamento condicional da página. O caso mais simples é a situação em que algo é mostrado apenas se uma condição for verdadeira. Poderia ser algum tipo de etiqueta condicional do tipo `<IF condição = "$quedaDePreço > 0.1">... mostrar alguma coisa </IF>`. O problema é que, quando você começa a ter etiquetas condicionais como essa, você começa a ir pelo caminho de transformar os padrões em uma linguagem de programação . Isso o leva aos mesmos problemas com os quais se depara quando insere *scriptlets* na página. Se você precisar mesmo de uma linguagem de programação, até poderia usar *scriptlets*, mas você sabe o que eu penso dessa idéia!

Como conseqüência disso, vejo etiquetas puramente condicionais como indício de problemas, algo que você deveria tentar evitar. Você não pode evitá-las sempre, mas deve tentar alguma coisa mais focado do que uma etiqueta `<IF>` de propósito geral.

Se você estiver mostrando algum texto condicionalmente, uma opção é mover a condição para o objeto auxiliar. A página então irá sempre inserir o resultado da chamada no objeto auxiliar. Se a condição não for verdadeira o objeto auxiliar irá enviar de volta uma *string* vazia, mas, desta forma, ele armazena toda a lógica. A abordagem funciona melhor se não houver *markups* para o texto retornado ou se for suficiente retornar um *markup* vazio que é ignorado pelo navegador.

Isto não funciona se, digamos, você quiser realçar em uma lista itens que apresentam boas vendagens colocando seus nomes em negrito. Em tal situação, você sempre precisa dos nomes exibidos, mas, às vezes, quer o *markup* especial. Uma maneira de obter isso é fazer o objeto auxiliar gerar o *markup.* Isso mantém toda a lógica fora da página, ao custo de retirar a escolha do mecanismo de destaque do projetista da página e de dá-la ao código de programação.

Para manter a escolha do HTML nas mãos do projeto da página, você precisa de algum tipo de etiqueta condicional. Entretanto, é importante olhar para além de um simples `<IF>`. Um bom caminho a escolher é uma etiqueta focada, de modo que, em vez de uma etiqueta que se parece com

```
<IF expressão = "estáVendendoBastante( )"><B></IF>
<property name = "preço"/>
<IF expressão = "estáVendendoBastante( )"></B></IF>
```

você tem uma como

```
<destacar se condição = "estáVendendoMuito" style = "bold">
    <property name = "preço"/>
</destacar>
```

Em ambos os casos é importante que a condição seja feita baseada em uma única propriedade Boleana do objeto auxiliar. Colocar alguma expressão mais complexa na página é na verdade colocar a lógica na própria página.

Outro exemplo seria colocar informações em uma página que dependa do local no qual o sistema esteja sendo executado. Considere algum texto que só deva ser mostrado nos Estados Unidos ou no Canadá, o qual, em vez de

```
<IF expressão = "local = 'US' || 'CA' " > .. texto especial </IF>
```

Seria algo como

```
<local includes = "US, CA">... texto especial </local>
```

Iteração Iterar sobre uma coleção apresenta problemas semelhantes. Se você quiser uma tabela em que cada linha corresponda a uma linha de item em um pedido, você precisa de uma construção que permita a exibição fácil de informação para cada linha. Aqui é difícil evitar uma iteração geral sobre uma etiqueta de coleção, mas isso geralmente funciona com simplicidade suficiente para se ajustar bem.

É claro que os tipos de etiquetas com as quais você tem que trabalhar muitas vezes são limitados pelo ambiente no qual você está. Alguns ambientes lhe dão um conjunto fixo de padrões, caso em que você pode estar mais limitado do que gostaria, seguindo estes tipos de diretrizes. Em outros ambientes, entretanto, você pode ter mais escolha nas etiquetas a usar. Muitos deles permitem até que você defina suas próprias bibliotecas de etiquetas.

Quando Processar O nome *Vista Padrão* salienta o fato de que a função primária deste padrão é executar o papel de vista no padrão *Modelo Vista Controlador* (315). Para muitos sistemas a *Vista Padrão* deveria apenas executar o papel de vista. Em sistemas mais simples, pode ser razoável que ela execute o papel do controle, e possivelmente até o de modelo, embora eu fosse lutar para separar o processamento do modelo tanto quanto possível. Onde a *Vista Padrão* tem outras responsabilidades além da vista, é importante assegurar que essas responsabilidades são manipuladas pelo objeto auxiliar, não pela página. Responsabilidades de modelo e de controle envolvem lógica de programação que, como qualquer lógica de programação, deveria ficar no objeto auxiliar.

Qualquer sistema padrão demanda um processamento extra pelo servidor Web. Isso pode ser feito compilando-se a página após ela ser criada, compilando-a na sua primeira solicitação ou interpretando-a a cada solicitação. Obviamente esta última opção não é uma boa idéia, se a interpretação tomar algum tempo.

Uma coisa a observar com a *Vista Padrão* são as exceções. Se uma exceção chegar ao contâiner Web, você pode se descobrir com uma página tratada pela metade, que, em vez de um redirecionamento, fornece alguma saída estranha para o navegador solicitante. Você precisa investigar como o seu servidor Web manipula exceções. Se ele fizer algo estranho, capture você mesmo todas as exceções na classe auxiliar (outra razão para desdenhar os *scriptlets*.)

Usando *Scripts* Embora páginas servidoras sejam uma das formas mais comuns de *Vista Padrão* atualmente, você pode escrever *scripts* no estilo *Vista Padrão*. Tenho visto uma quantidade razoável de Perl feita desta maneira. Mais notadamente demonstrada pelo CGI.pm de Perl, o truque é evitar concatenar *strings* tendo chamadas de funções que tenham como saída os identificadores apropriados à resposta. Dessa forma, você pode escrever o *script* na sua linguagem de programação e evitar a confusão de misturar a impressão de *strings* com lógica de programação.

Quando Usá-la

Para implementar a vista no *Modelo Vista Controlador* (315), a escolha principal é entre *Vista Padrão* e *Vista de Transformação* (343). A força da *Vista Padrão* é que ela permi-

te que você forme o conteúdo da página olhando a estrutura da mesma. Isso parece ser mais fácil de fazer e aprender para a maioria das pessoas. Em especial, ela suporta muito bem a idéia de um projetista gráfico compondo uma página com um programador trabalhando no auxiliar.

A *Vista Padrão* tem duas fraquezas significativas. Primeiro, as implementações comuns tornam fácil demais colocar lógica complicada na página, tornando-a assim difícil de sofrer manutenção, especialmente por não-programadores. Você precisa de muita disciplina para manter a página simples e orientada à exibição, colocando a lógica no auxiliar. A segunda fraqueza é que a *Vista Padrão* é mais difícil de testar do que a *Vista de Transformação* (343). A maioria das implementações de *Vista Padrão* são projetadas para trabalhar dentro de um servidor Web e são muito difíceis ou impossíveis de testar de outra forma. Implementações de *Vista de Transformação* (343) são muito mais fáceis de inserir em uma rotina de testes e testar sem rodar um servidor Web.

Ao pensar sobre uma vista, você também precisa considerar *Vista em Duas Etapas* (347). Dependendo de seu esquema do padrão, você pode conseguir implementar este padrão usando identificadores especializados. Entretanto, você pode achar mais fácil implementá-lo baseado em uma *Vista de Transformação* (343). Se você vai precisar de *Vista em Duas Etapas* (347), você precisará levar isso em consideração na sua escolha.

Exemplo: Usando uma JSP como uma Vista com um Controle Separado (Java)

Quando se usa uma JSP como uma vista apenas, ela é sempre chamada a partir de um controle em vez de diretamente do contâiner de *servlet*. Assim, é importante passar para a JSP qualquer informação de que ela vá precisar para descobrir o que exibir. Uma boa maneira de fazer isso é fazer o controlador criar um objeto auxiliar e passá-lo para a JSP usando a solicitação HTTP. Mostraremos isso com o exemplo de exibição simples do *Controlador de Página* (318). O método de manipulação Web para o *servlet* parece com o seguinte:

```
class ControladorDeArtista...

    public void executarGet (HttpServletRequest solicitação, HttpServletResponse resposta)
            throws IOException, ServletException {
        Artista artista = Artista.buscarPeloNome(solicitação.getParameter("nome"));
        if (artista == null)
            forward("/ErroArtistaNãoEncontrado.jsp", solicitação, resposta);
        else {
            solicitação.setAttribute("auxiliar", new AuxiliarDeArtista (artista));
            forward ("/artista.jsp", solicitação, resposta);
        }
    }
```

Quando se considera a *Vista Padrão*, o comportamento importante é criar o auxiliar e colocá-lo na solicitação. A página servidora pode agora alcançar o auxiliar com o identificador `useBean`.

```
<jsp:useBean id="auxiliar" type="actionController.auxiliarDeArtista" scope="solicitação"/>
```

Com o auxiliar no lugar, podemos usá-lo para acessar as informações que precisamos exibir. As informações do modelo de que o auxiliar precisa foram passadas para ele quando foi criado.

```
class AuxiliarDeArtista...

    private Artista artista;
    private AuxiliarDeArtista (Artista artista) {
        this.artista = artista;
    }
```

Podemos usar o auxiliar para obter informações apropriadas sobre o modelo. No caso mais simples, fornecemos um método para obter dados simples, como o nome do artista.

```
class AuxiliarDeArtista...

    public String lerNome( ) {
        return artista.lerNome( );
    }
```

Então acessamos esta informação por meio de uma expressão Java.

```
    <B> <%=auxiliar.lerNome( )%></B>
```

ou uma propriedade

```
    <B><jsp:lerPropriedade name="auxiliar" property="nome".></B>
```

A escolha entre propriedades ou expressões depende de quem está editando a JSP. Os programadores acham as expressões fáceis de ler e mais compactas, mas editores HTML podem não ser capazes de manipulá-las. Não-programadores irão provavelmente preferir identificadores, já que eles se adaptam à forma geral de HTML e deixam menos espaço para erros que confundem.

Usar um auxiliar é um modo de remover código *scriptlet* complicado. Se você quiser mostrar uma lista de álbuns de um artista, você precisa executar um laço, o que você pode fazer com um *scriptlet* na página servidora.

```
<UL>
<%
    for (Iterator it = auxiliar.lerÁlbuns( ).iterator( ); it.hasNext( );) {
        Álbum álbum = (Álbum) it.next( );%>
    <LI><%=álbum.lerTítulo( )%></LI>
<% } %>
</UL>
```

Francamente, esta mistura de Java e HTML é realmente horrível de ler. Uma alternativa é mover o laço do *for* para o auxiliar.

```
class AuxiliarDeArtista...

    public String lerListaDeÁlbuns( ) {
        StringBuffer resultado = new StringBuffer( );
        resultado.append("<UL>");
```

```
    for (Iterator it = lerÁlbuns( ).iterator( ); it.hasNext( );) {
        Álbum álbum = (Álbum) it.next( );
        resultado.append("<LI>");
        resultado.append(álbum.lerTítulo( ));
        resultado.append("</LI>");
    }
        resultado.append("</UL>");
        return resultado.toString( );
    }
    public List lerÁlbuns( ) {
        return artista.lerÁlbuns( );
    }
```

Considero isso mais fácil de acompanhar porque a quantidade de HTML é bem pequena. Isso também permite que você use uma propriedade para ler a lista. Muitas pessoas não gostam de colocar código HTML em auxiliares. Embora eu prefira não fazê-lo, havendo a escolha entre HTML e *scriptlets*, eu escolho o primeiro em auxiliares em qualquer momento.

O melhor caminho a seguir é um identificador especializado para iteração.

```
<UL><tag:forEach host = "auxiliar" collection = "álbuns" id = "cada">
    <LI><jsp:lerPropriedade name="cada" property = "título"/></LI>
</tag:forEach></UL>
```

Esta é uma alternativa muito melhor, pois mantém os *scriptlets* fora da JSP e HTML fora do auxiliar.

Exemplo: Página Servidora ASP.NET (C#)

Este exemplo continua o que comecei no *Controlador de Página* (318) (página 325). Recordando para você, ele mostra os pontos feitos por rebatedores em um único período de uma partida de críquete. Para aqueles que pensam que críquete (*cricket*, grilo) é um inseto pequeno e barulhento, omitirei as longas rapsódias sobre o esporte mais imortal do mundo e resumirei tudo no fato de que a página exibe três informações essenciais:

- Um número de ID para referenciar a partida.
- Os pontos de quais equipes estão mostrados e de quais períodos eles são.
- Uma tabela mostrando o nome de cada rebatedor e a média de pontos (o número de bolas que ele recebeu dividido pelo número de pontos que ele marcou).

Se você não entende o que estas estatísticas significam, não se preocupe. O críquete é cheio de estatísticas – talvez sua maior contribuição à humanidade seja fornecer estatísticas estranhas para artigos excêntricos.

A discussão sobre o *Controlador de Página* (318) abordou como uma solicitação Web é manipulada. Para resumir, o objeto que atua como o controlador e como a vista é a página ASP.NET aspx. Para manter o código do controle fora de um *scriptlet*, você define um código separado atrás da classe.

```
<%@ Page language = "C#" Codebehind="bat.aspx.cs" AutoEventWireup="false" trace="False"
    Inherits="rebatedores.PáginaDeRebatidas" %>
```

A página pode acessar os métodos e propriedades do código atrás da classe diretamente. Além disso, esse código pode definir um método `Carregar_Página` para manipular a solicitação. Neste caso, defini `Carregar_Página` como um método padrão [Gang of Four] em uma *Camada Supertipo* (444).

```
class PáginaDeCríquete...

    protected void Carregar_Página (Object sender, System.EventArgs e) {
        db = new OleDbConnection(DB.ConnectionString);
        if(háParâmetrosFaltando( ))
            erroDeTransferência (mensagemDeFaltaDeParâmetros);
        DataSet ds = lerDados( );
        if(nãoHáDados(ds))
            erroDeTransferência ("Nenhum dado corresponde à sua solicitação");
        aplicarLógicaDoDomínio(ds);
        LigarDados( );
        PrepararInterfaceDeUsuário(ds);
    }
```

Para os propósitos da *Vista Padrão*, eu posso ignorar tudo menos as duas últimas linhas da carga da página. A chamada a `LigarDados` permite que diversas variáveis de página sejam ligadas apropriadamente a suas fontes de dados correspondentes. Isso será suficiente nos casos mais simples, mas para os mais complicados a última linha chama um método no código separado da página específica para preparar quaisquer objetos para seu uso.

O número de ID, equipe e período do jogo são valores únicos para a página, todos eles vindos para a página como parâmetros na solicitação HTTP. Posso fornecer esses valores usando propriedades no código atrás da classe

```
class PáginaDeRebatidas...

    protected String equipe {
        get {return Request.Params["equipe"];}
    }
    protected String jogo {
        get {return Request.Params["jogo"];}
    }
    protected String período {
        get {return Request.Params["período"];}
    }
    protected String perídoOrdinal {
        get {return (período == "1") ? "1º": "2º";}
    }
```

Com as propriedades definidas, posso usá-las no texto da página.

```
    <P>
        Id do Jogo:
        <asp:label id = "rótuloJogo" Text="<%# jogo %>" runat="server" font-bold="True">
        </asp:label> 
```

```
    </P>
    <P>
        <asp:label id = "rótuloEquipe" Text="<%# equipe %>" runat="server" font-bold="True">
        </asp:label> 
        <asp:label id = "rótuloPeríodoOrdinal" Text="<%# períodoOrdinal %>" runat="server"
font-bold="True">
        </asp:label> período</P>
    <P>
```

A tabela é um pouco mais complicada, mas realmente funciona facilmente na prática, devido aos recursos gráficos de projeto no *Visual Studio*. O *Visual Studio* fornece um controle de *grid* de dados que pode ser ligado a uma única tabela de um conjunto de dados. Posso fazer isso ligando o método `prepararInterfaceDeUsuário` que é chamado pelo método `Carregar_Página`.

```
class PáginaDeRebatidas...

    override protected void prepararInterfaceDeUsuário (DataSet ds) {
        DataGrid1.DataSource = ds;
        DataGrid1.DataBind( );
    }
```

A classe de rebatidas é um *Módulo Tabela* (134) que fornece lógica de domínio para a tabela de rebatidas no banco de dados. A propriedade de seus dados são os dados dessa tabela enriquecidos pela lógica de domínio do *Módulo Tabela* (134). Aqui o enriquecimento é a média de pontos, que é calculada em vez de armazenada no banco de dados.

Com o *grid* de dados ASP.NET, você pode selecionar quais colunas da tabela você quer exibir na página Web, junto com informações sobre a aparência da tabela. Neste caso, podemos selecionar as colunas do nome, pontos e média.

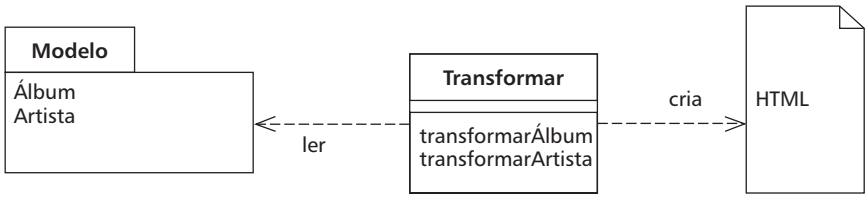

```
        <asp:DataGrid id="DataGrid1" runat="server" Width="480px" Height="171px"
                BorderColor="#336666" BorderStyle="Double" BorderWidth= "3px" BackColor="White"
                CellPadding="4" GridLines="Horizontal" AutoGeneratedColumns="False">
            <SelectedItemStyle Font-Bold="True" ForeColor="White" BackColor="#339966"></
                SelectedItemStyle>
            <ItemStyle ForeColor="#333333" BackColor="White"></ItemStyle>
            <HeaderStyle Font-Bold="True" ForeColor="White" BackColor="#336666"></HeaderStyle>
            <FooterStyle ForeColor="#333333" BackColor="White"></FooterStyle>
            <Columns>
                <asp:BoundColumn DataField= "nome" HeaderText="Rebatedor">
                    <HeaderStyle Width="70px"></HeaderStyle>
                </asp:BoundColumn>
                <asp:BoundColumn DataField= "pontos" HeaderText="Pontos">
                    <HeaderStyle Width="30px"></HeaderStyle>
                </asp:BoundColumn>
                <asp:BoundColumn DataField= "stringMédia" HeaderText="Média">
                    <HeaderStyle Width="30px"></HeaderStyle>
                </asp:BoundColumn>
            </Columns>
            <PagerStyle HorizontalAlign="Center" ForeColor="White" BackColor="#336666"
                Mode="NumericPages"></PagerStyle>
        </asp:DataGrid></P>
```

O HTML para este *grid* de dados parece intimidador, mas no *Visual Studio* você não o manipula diretamente, mas sim por meio de folhas de propriedades no ambiente de desenvolvimento, como você faz para uma boa parte do resto da página.

Esta habilidade de ter controles de formulário Web na página Web que compreendem as abstrações de ADO.NET dos conjuntos de dados e tabelas de dados é a força e a limitação deste esquema. A força é que você transfere informações por meio de conjuntos de dados, graças ao tipo de ferramentas que o *Visual Studio* fornece. A limitação é que isso só funciona sem alterações quando você usa padrões como *Módulo Tabela* (134). Se você tiver lógica de domínio muito complexa, então um *Modelo de Domínio* (126) se torna útil. Para tirar proveito das ferramentas, o *Modelo de Domínio* (126) precisa criar seu próprio conjunto de dados.

Vista de Transformação (Transform View)

Uma vista que processa dados do domínio elemento por elemento e os transforma em HTML.



Quando você envia solicitações de dados para o domínio e camadas de fontes de dados, recebe de volta todos os dados de que precisa para satisfazê-las, mas sem a formatação necessária para criar uma página Web apropriada. O papel da vista no *Modelo Vista Controle* (315) é representar esses dados em uma página Web. Usar *Vista de Transformação* significa pensar nisso como uma transformação em que você tem os dados do modelo como entrada e seu HTML como saída.

Como Funciona

A noção básica de *Vista de Transformação* é escrever um programa que olha os dados orientados ao domínio e os converte para HTML. O programa percorre a estrutura dos dados do domínio e, quando reconhece cada forma de dado do domínio, grava a parte específica de HTML para ela. Se você pensar nisso de uma forma imperativa, poderá ter um método chamado `exibirCliente` que recebe um objeto cliente e o exibe em HTML. Se o cliente contiver muitos pedidos, este método percorre os pedidos chamando `exibirPedido`.

A diferença-chave entre *Vista de Transformação* e *Vista Padrão* (333) é o modo pelo qual a vista é organizada. Uma *Vista Padrão* (333) é organizada em torno da saída. Uma *Vista de Transformação* é organizada em torno de transformações separadas para cada tipo de elemento de entrada. A transformação é controlada por algo como um laço simples que olha cada elemento de entrada, encontra a transformação apropriada para esse elemento e então chama a transformação sobre ele. As regras de uma *Vista de Transformação* típica podem ser arrumadas em qualquer ordem sem afetar a saída resultante.

Você pode escrever uma *Vista de Transformação* em qualquer linguagem. No momento, entretanto, a escolha dominante é XSLT. O interessante disso é que XSLT é uma linguagem de programação funcional, similar a Lisp, Haskell e outras linguagens que nunca chegaram a ser *mainstream.* Como tal, ela tem um tipo diferente de estrutura para isso. Por exemplo, em vez de chamar rotinas explicitamente, XSLT reconhece elementos nos dados do domínio e então chama as transformações para exibição apropriadas.

Para executar uma transformação XSLT, precisamos começar com alguns dados XML. A maneira mais simples pela qual isso pode acontecer é se o tipo de retorno natural da lógica de domínio for ou XML ou algo transformável automaticamente em XML – por exemplo, um objeto .NET. Se isso falhar, precisamos nós mesmos produzir o XML, talvez povoando um *Objeto de Transferência de Dados* (380) que possa seria-

lizar a si próprio em XML. Dessa forma, os dados podem ser juntados usando uma API conveniente. Em casos mais simples, um *Roteiro de Transação* (110) pode retornar XML diretamente.

O XML que alimenta a transação não tem que ser uma *string*, a menos que uma forma de *string* seja necessária para passar por uma linha de comunicação. É geralmente mais rápido e mais fácil produzir um DOM e passá-lo para a transformação.

Assim que tivermos o XML, passamos o mesmo para um mecanismo XSLT, o qual está se tornando cada vez mais disponível comercialmente. A lógica para a transformação é capturada em uma *style sheet* XSLT, a qual também passamos para o transformador. Este, então, aplica a *style sheet* na entrada XML para produzir a saída HTML, a qual podemos gravar diretamente na resposta HTTP.

Quando Usá-la

A escolha entre uma *Vista de Transformação* e uma *Vista Padrão* (333) na sua maior parte recai sobre qual ambiente a equipe trabalhando no *software* de vista prefere. A presença de ferramentas é um fator-chave aqui. Há mais e mais editores HTML que você pode usar para escrever *Vistas Padrão* (333). Ferramentas para XSLT são, pelo menos até agora, muito menos sofisticadas. Além disso, XSLT pode ser uma linguagem complicada de dominar, devido ao seu estilo de programação funcional acoplado com sua sintaxe XML complicada.

Uma das forças de XSLT é sua portabilidade a quase que qualquer plataforma Web. Você pode usar o mesmo XSLT para transformar XML criado a partir de J2EE ou .NET, o que pode ajudar a colocar uma vista comum HTML sobre dados de fontes diferentes.

O XSLT também é muitas vezes mais fácil se você estiver construindo uma vista sobre um documento XML. Outros ambientes geralmente requerem que você transforme tal documento em um objeto ou aceite passar o DOM XML, o que pode ser complicado. XSLT se ajusta naturalmente a um mundo XML.

Vista de Transformação evita dois dos maiores problemas com *Vista Padrão* (333). É mais fácil manter a transformação focada apenas em representar HTML, desta forma evitando ter muita outra lógica na vista. Também é fácil executar *Vista de Transformação* e capturar a saída para teste. Isto torna mais fácil testar a vista e você não precisa de um servidor Web para rodar os testes.

A *Vista de Transformação* transforma diretamente de XML orientado a domínio para HTML. Se você precisar alterar a aparência geral de um *site* Web, isso pode forçá-lo a alterar diversos programas de transformação. Usar transformações comuns, como as que incluem XSLT, ajuda a reduzir este problema. De fato, é muito mais fácil chamar transformações comuns usando *Vista de Transformação* do que usando *Vista Padrão* (333). Se você precisar fazer alterações globais facilmente ou suportar diversas aparências para os mesmos dados, você poderia considerar *Vista em Duas Etapas* (347), a qual usa um processo de duas etapas.

Exemplo: Transformação Simples (Java)

Estabelecer uma transformação simples envolve o preparo de código Java para chamar a *style sheet* correta para compor a resposta. Também envolve preparar a *style sheet* para formatar a resposta. A maior parte da resposta a uma página é bastante genérica, de modo que faz sentido usar *Controlador Frontal* (328). Descreverei apenas o

comando aqui, e você deve olhar o *Controlador Frontal* (328) para ver como o objeto comando se ajusta ao resto da manipulação de resposta da solicitação.

Tudo o que o objeto comando faz é chamar os métodos no modelo para obter um documento XML de entrada e, então, passar esse documento XML através de um processador XML.

```
class ComandoÁlbum...

    public void processar( ) {
        try{
            Álbum álbum = Álbum.buscarPeloNome(solicitação.getParameter("nome"));
            Assert.notNull(álbum);
            PrintWriter saída = resposta.getWriter( );
            XsltProcessor processador = new SingleStepXsltProcessor("álbum.xsl");
            saída.print(processador.getTransformation(álbum.toXmlDocument( )));
        } catch (Exception e) {
            throw new ApplicationException (e);
        }
    }
```

O documento XML pode se parecer um pouco com isto:

```
<álbum>
    <título>Stormcock</título>
    <artista>Roy Harper</artista>
    <listaDeFaixas>
        <faixa><título>Hors d'Oeuvres</título><tempo>8:37<cell><row>
        <faixa><título>The Same Old Rock</título><tempo>12:24<cell><row>
        <faixa><título>One Man Rock and Roll</título><tempo>7:23<cell><row>
        <faixa><título>Me and My Woman</título><tempo>13:01<cell><row>
    </listaDeFaixas>
</álbum>
```

A tradução do documento XML é feita por um programa XSLT. Cada padrão se enquadra a uma parte específica do XML e produz uma saída HTML apropriada para esta página. Neste caso, mantive a formatação em um nível excessivamente simples para mostrar apenas o essencial. As cláusulas do padrão a seguir enquadram os elementos básicos do arquivo XML.

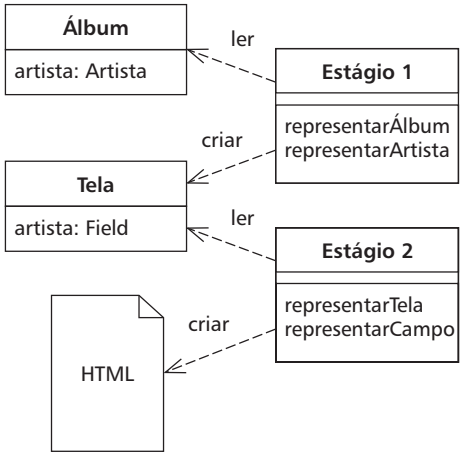
```
<xsl:template match="álbum">
    <HTML><BODY bgcolor="white">
    <xsl:apply-templates/>
    </BODY></HTML>
</xsl:template>
<xsl:template match="álbum/título">
    <h1><xsl:apply-templates/></h1>
</xsl:template>
<xsl:template match="álbum/artista">
    <P><B>Artista: </B><xsl:apply-templates/></P>
</xsl:template>
```

Estes padrões manipulam a tabela, que aqui tem linhas alternadas destacadas em cores diferentes. Este é um bom exemplo de algo que não é possível com fluxos de *style sheets*, mas é razoável com XML.

```
<xsl: template match="listaDeFaixas">
    <table><xsl: aaply-templates/></table>
</xsl: template>
<xsl: template match="faixa">
    <xsl:variable name="bgcolor">
        <xls:choose>
            <xsl:when test="(position( ) mod2) = 1"> linen</xsl:when>
            <xsl:otherwise>white</xsl:otherwise>
        </xsl:choose>
    </xsl:variable>
    <tr bgcolor="{$bgcolor}"><xsl:apply-templates/></tr>
</xsl: template>
<xsl: template match="faixa/título">
    <td><xsl:apply-templates/></td>
</xsl: template>
<xsl: template match="faixa/tempo">
    <td><xsl:apply-templates/></td>
</xsl: template>
```

Vista em Duas Etapas (Two Step View)

Transforma dados do domínio em HTML em duas etapas: primeiro formando algum tipo de página lógica e depois representando essa página lógica em HTML.



Se você tem uma aplicação Web com muitas páginas, você geralmente quer uma aparência e organização consistentes no *site*. Se cada página tiver uma aparência diferente, você acaba com um *site* que os usuários acham confuso. Você também pode querer fazer alterações globais na aparência do *site* com facilidade, porém técnicas comuns usando *Vista Padrão* (333) ou *Vista de Transformação* (343) tornam isso difícil, porque decisões relativas a apresentação estão muitas vezes duplicadas em diversas páginas ou módulos de transformação. Uma alteração global pode obrigá-lo a alterar diversos arquivos.

A *Vista em Duas Etapas* lida com este problema dividindo a transformação em dois estágios. O primeiro transforma os dados do modelo em uma apresentação lógica sem qualquer formatação específica. O segundo converte essa apresentação lógica na formatação real necessária. Dessa forma, você pode fazer uma alteração global alterando o segundo estágio, ou pode suportar diversas aparências de *output* cada uma com um segundo estágio.

Como Funciona

A chave deste padrão é fazer da transformação para HTML um processo em dois estágios. O primeiro junta a informação em uma estrutura de tela lógica que é uma indicação de que os elementos de exibição ainda não contêm HTML. O segundo estágio pega essa estrutura orientada a apresentação e a representa em HTML.

Esta forma intermediária é um tipo de tela lógica. Seus elementos podem incluir coisas como campos, cabeçalhos, rodapés, tabelas, seleções, e assim por diante. Como tal, ela é certamente orientada a apresentação e certamente força a tela a seguir um estilo preciso. Você pode pensar no modelo orientado a apresentação como um

que define os diversos dispositivos que você pode ter e os dados que eles contêm, mas que não especifica a aparência HTML.

A estrutura orientada a apresentação é montada por um código específico escrito para cada tela. A responsabilidade do primeiro estágio é acessar um modelo orientado a domínio, ou um banco de dados, um modelo real de domínio ou um *Objeto de Transferência de Dados* (380) orientado a domínio, extrair as informações relevantes para essa tela e então colocar essas informações em uma estrutura orientada a apresentação.

O segundo estágio transforma a estrutura orientada a apresentação em HTML. Ele conhece cada elemento da estrutura orientada a apresentação e sabe como mostrá-los como HTML. Assim, um sistema com muitas telas pode ser representado em HTML por um único segundo estágio de modo que todas as decisões de formatação de HTML são tomadas em um lugar. É claro que a restrição é que a tela resultante deve ser derivável de uma estrutura orientada a apresentação.

Há diversas maneiras de criar uma *Vista em Duas Etapas*. Talvez a mais fácil seja com XSLT em duas etapas. O XSLT de uma etapa segue a abordagem de *Vista de Transformação* (343), na qual cada página tem uma *style sheet* XSLT que transforma o XML orientado a domínio em HTML. Na abordagem de duas etapas, há duas *style sheets* XSLT. A *style sheet* da primeira etapa transforma XML orientado a domínio em XML orientado a apresentação, enquanto que a da segunda etapa representa esse XML em HTML.

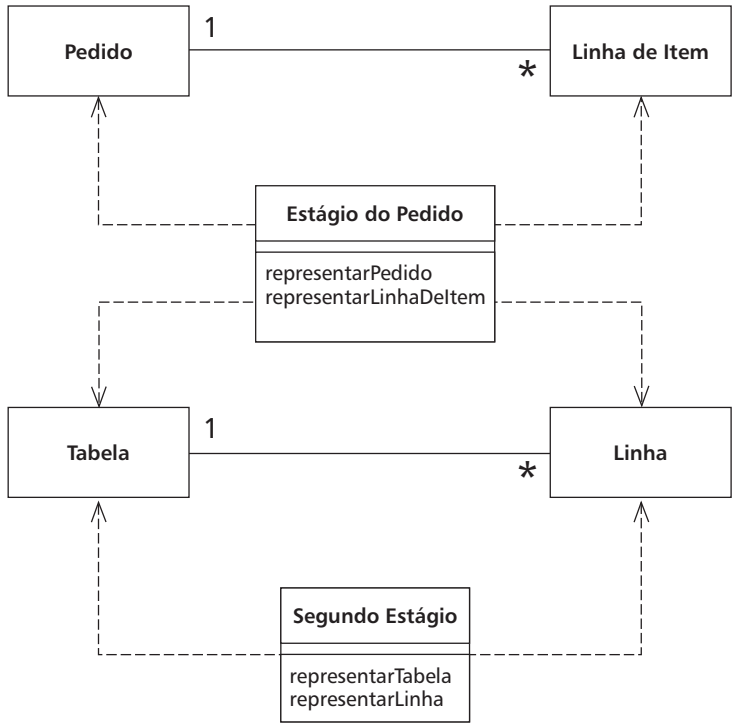


Figura 14.4 Classes exemplo para representação em duas etapas.

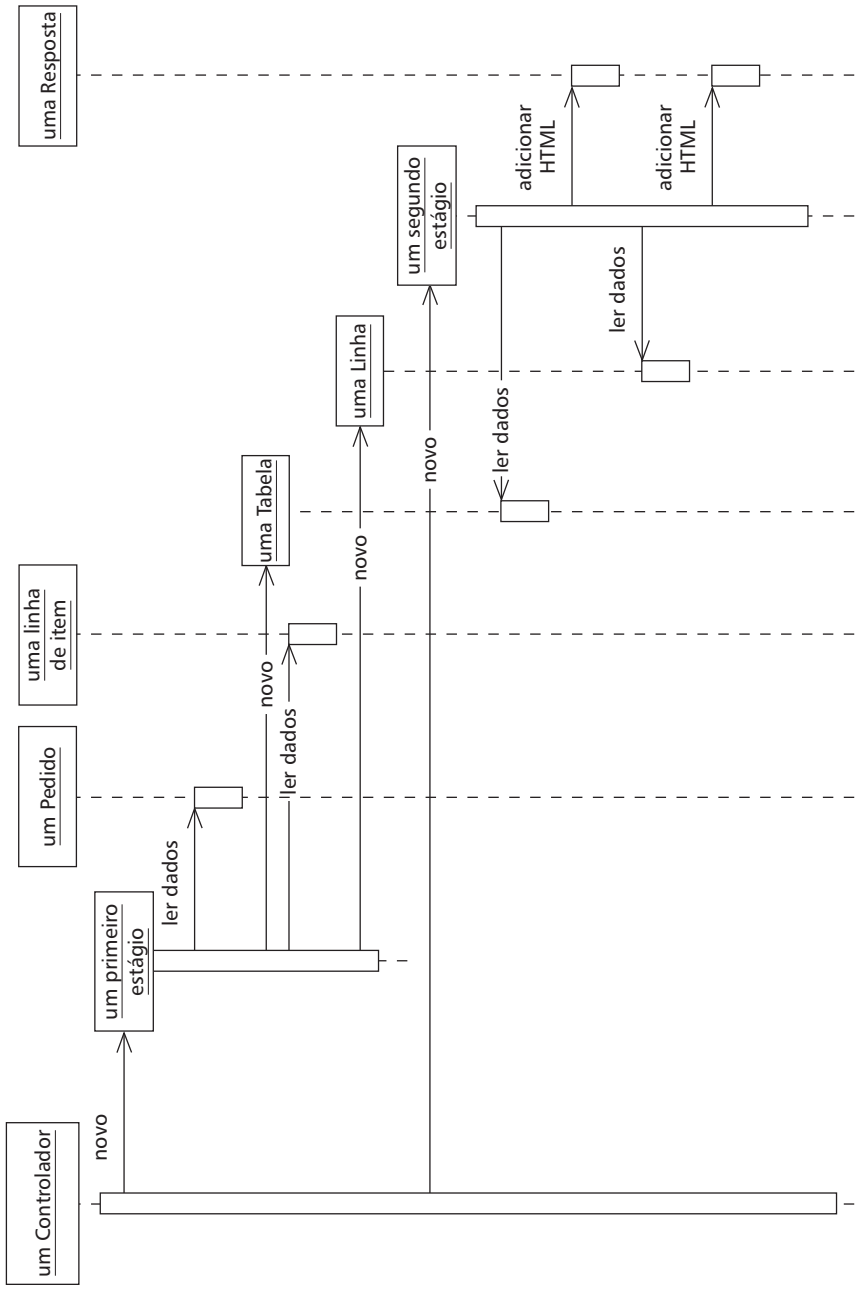


Figura 14.5 Diagrama de seqüência para representação em duas etapas.

Outra maneira é usando classes. Aqui você define a estrutura orientada a apresentação como um conjunto de classes: com uma classe tabela, uma classe linha, e assim por diante. O primeiro estágio pega informações do domínio e instancia estas classes em uma estrutura que modela uma tela lógica. O segundo estágio representa as classes em HTML, seja pegando cada classe orientada a apresentação para gerar HTML por si mesmo, seja tendo uma classe separada responsável por representar em HTML.

Ambas as abordagens são baseadas em *Vista de Transformação* (343). Você também pode usar uma abordagem baseada em *Vista Padrão* (333), na qual escolhe padrões baseados na idéia de uma tela lógica – por exemplo:

```
<field label = "Nome" value = "lerNome" />
```

O sistema padrão converte então estes identificadores lógicos em HTML. Neste esquema, a definição da página não inclui HTML, mas apenas estes identificadores de tela lógica. O resultado disso é que ele provavelmente será um documento XML, o que é claro significa que você perde a habilidade de usar editores WYSIWYG HTML.

Quando Usá-la

O valor-chave da *Vista em Duas Etapas* vem da separação do primeiro e do segundo estágios, permitindo que você faça alterações globais mais facilmente. Ela ajuda a pensar em duas situações: aplicações Web com diversas aparências ou com uma única aparência. Aplicações Web com diversas aparências são mais raras, mas estão aumentando. Nelas a mesma funcionalidade básica é fornecida por diversas organizações, e cada organização tem sua própria aparência distinta. Um exemplo atual são *sites* de viagens aéreas, nos quais, quando você os olha, pode perceber pela aparência e projeto que são todas variações de um *site* básico. Suspeito que muitas linhas aéreas querem essa mesma funcionalidade, mas com uma aparência distintamente individual.

Aplicações de uma única aparência são mais comuns. Apenas uma organização faz sua parte frontal, e elas querem uma aparência consistente em todo o *site.* Isso as torna o caso mais fácil para considerar primeiro.

Com uma vista de único estágio (*Vista Padrão* (333) ou *Vista de Transformação* (343)), você constrói um módulo vista por página Web (veja a Figura 14.6). Com uma *Vista em Duas Etapas*, você tem dois estágios: um módulo de primeiro estágio por página e um de segundo estágio para a aplicação inteira (Figura 14.7). Sua compensação por usar *Vista em Duas Etapas* é que qualquer alteração na aparência do *site* no segundo estágio é muito mais fácil de fazer, já que uma alteração no segundo estágio afeta o *site* como um todo.

Com uma aplicação de diversas aparências, esta vantagem é aumentada porque você tem uma vista de único estágio para cada combinação de tela e aparência (Figura 14.8). Assim, dez telas e três aparências requerem trinta módulos vista de único estágio. Usando *Vista em Duas Etapas*, entretanto (veja a Figura 14.9), você pode conseguir seu intento com os dez primeiros estágios e os três segundos estágios. Quanto mais telas e aparências você tiver, maior a economia.

Apesar disso, sua habilidade de ser bem-sucedido depende de quão bem você conseguir criar a estrutura orientada a apresentação para realmente servir as neces-

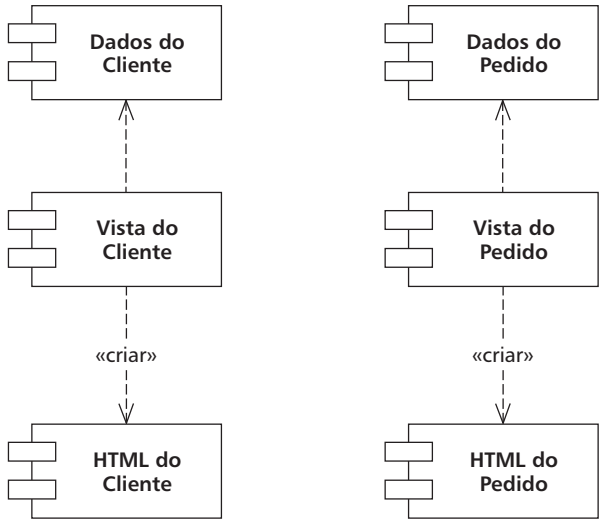


Figura 14.6 Vista de único estágio com uma aparência.

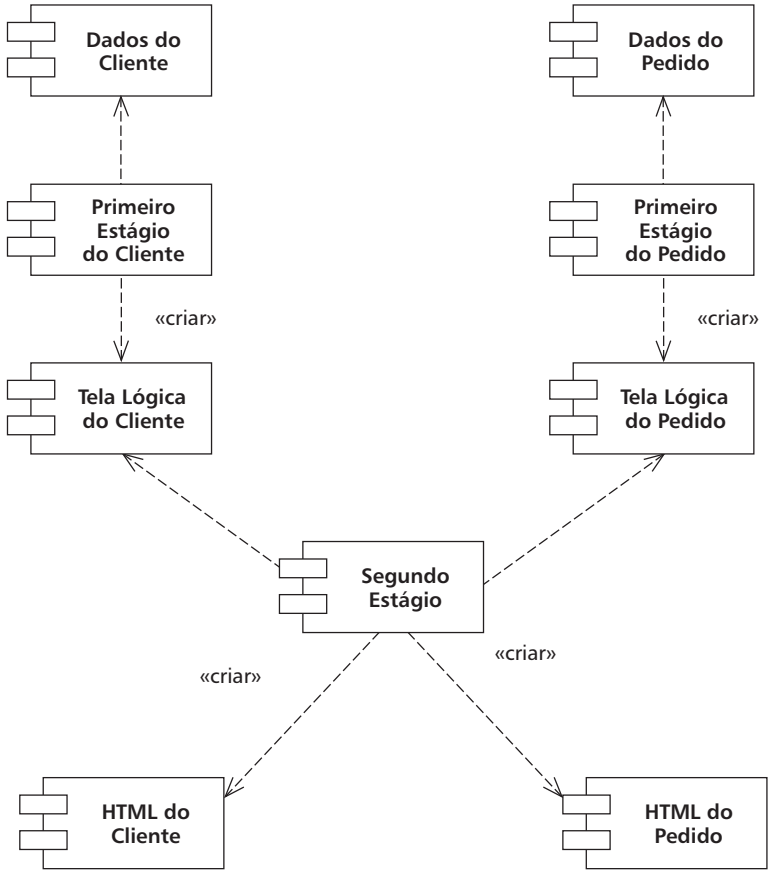


Figura 14.7 Vista de dois estágios com uma aparência.

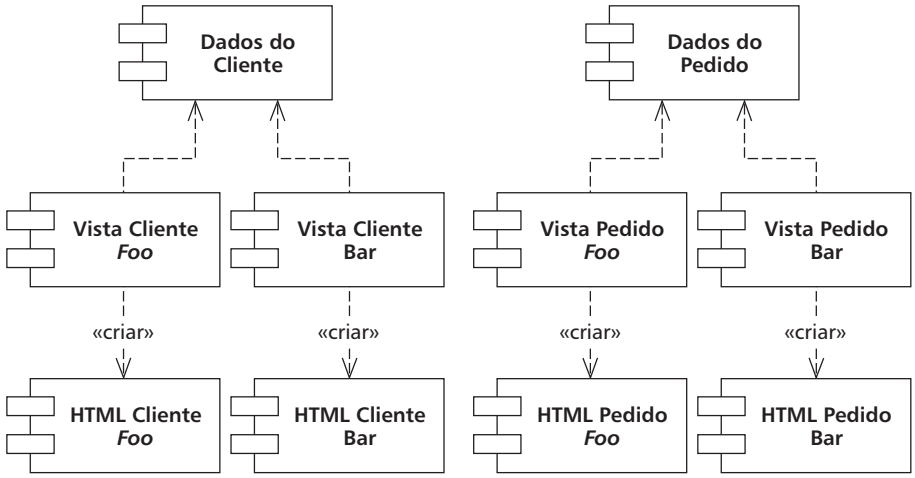


Figura 14.8 Vista de único estágio com duas aparências.

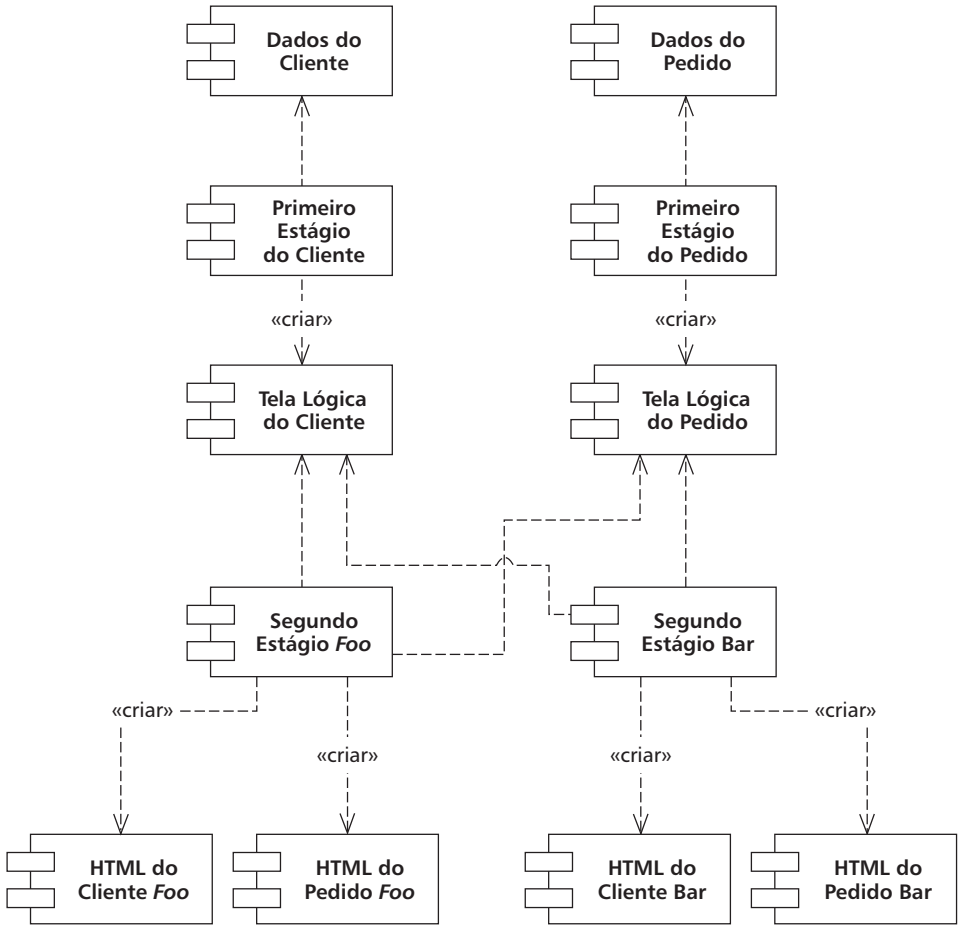


Figura 14.9 Vista de dois estágios com duas aparências.

sidades da aparência. Um *site* com projeto pesado, no qual cada página deva parecer diferente, não funcionará bem com *Vista em Duas Etapas*, porque é difícil encontrar coisas em comum em quantidade suficiente entre as telas para obter uma estrutura orientada a apresentação simples o suficiente. Basicamente o projeto do *site* é limitado pela estrutura orientada a apresentação, e para muitos *sites* é uma limitação demasiada.

Outro empecilho de *Vista em Duas Etapas* são as ferramentas requeridas para usá-la. Há muitas ferramentas para projetistas sem habilidade de programação criarem páginas HTML usando *Vista Padrão* (333), mas *Vista em Duas Etapas* obriga os programadores a escreverem os objetos representadores e controladores. Assim, os programadores têm que estar envolvidos em qualquer alteração no projeto.

Também é verdade que a *Vista em Duas Etapas*, com suas múltiplas camadas, apresenta um modelo de programação mais difícil de aprender, embora assim que você esteja acostumado com ele não é difícil, e pode ajudar a reduzir código repetitivo.

Uma variação no tema das aparências múltiplas é fornecer segundos estágios diferentes para dispositivos diferentes, de modo que você possa ter um segundo estágio para um navegador e outro para um PDA. A limitação costumeira aqui é que ambas as aparências devem seguir a mesma tela lógica e, para dispositivos muito diferentes, isso pode ser pedir muito.

Exemplo: XSLT em Duas Etapas (XSLT)

Esta abordagem de *Vista em Duas Etapas* usa uma transformação XSLT de dois estágios. O primeiro estágio transforma XML específica do domínio em XML de tela lógica. O segundo estágio transforma XML de tela lógica em HTML.

O XML inicial orientado a domínio se parece com o seguinte:

```
<álbum>
    <título>Zero Hour</title>
    <artista>Astor Piazzola</artist>
    <listaDeFaixas>
        <faixa><título>Tanguedia III</título><tempo>4:39<cell><row>
        <faixa><título>Milonga del Angel</título><tempo>6:30<cell><row>
        <faixa><título>Concierto Para Quinteto</título><tempo>9:00<cell><row>
        <faixa><título>Milonga Loca</título><tempo>3:05<cell><row>
        <faixa><título>Michelangelo '70</título><tempo>2:50<cell><row>
        <faixa><título>Contrabajisimo</título><tempo>10:18<cell><row>
        <faixa><título>Mumuki</título><tempo>9:32<cell><row>
    </listaDeFaixas>
</álbum>
```

O processador XSLT da primeira página a transforma em XML orientado a tela:

```
<screen>
    <título>Zero Hour</title>
    <field label="Artista">Astor Piazzola</field>
    <table>
        <row><cell>Tanguedia III</cell><cell>4:39<cell><row>
        <row><cell>Milonga del Angel</cell><cell>6:30<cell><row>
        <row><cell>Concierto Para Quinteto</cell><cell>9:00<cell><row>
```

```
        <row><cell>Milonga Loca</cell><cell>3:05<cell><row>
        <row><cell>Michelangelo '70</cell><cell>2:50<cell><row>
        <row><cell>Contrabajisimo</cell><cell>10:18<cell><row>
        <row><cell>Mumuki</cell><cell>9:32<cell><row>
    </table>
</screen>
```

Para fazer isso, precisamos do seguinte programa XSLT:

```
<xsl:template match="album">
    <screen><xsl:apply-templates/></screen>
</xsl:template>
<xsl:template match="album/title">
    <title><xsl:apply-templates/></title>
</xsl:template>
<xsl:template match="artist">
    <field label="Artist"><xsl:apply-templates/></field>
</xsl:template>
<xsl:template match="trackList">
    <table><xsl:apply-templates/></table>
</xsl:template>
<xsl:template match="track">
    <row><xsl:apply-templates/></row>
</xsl:template>
<xsl:template match="track/title">
    <cell><xsl:apply-templates/></cell>
</xsl:template>
<xsl:template match="track/time">
    <cell><xsl:apply-templates/></cell>
</xsl:template>
```


O XML orientado a tela é bastante simples. Para transformá-lo em HTML usamos um programa XSLT de segundo estágio.

```
<xsl: template match="screen">
    <HTML><BODY bgcolor="white">
    <xsl:apply-templates/>
    </BODY></HTML>
</xsl: template>
<xsl: template match="título">
    <h1><xsl:apply-templates/></h1>
</xsl: template><xsl: template match="field">
    <P><B><xsl:value-of select = "@label"/>: </B><xsl:apply-templates/></P>
</xsl: template>
<xsl: template match="table">
    <table><xsl:apply-templates/></table>
</xsl: template>
<xsl: template match="table/row">
    <xsl:variable name="bgcolor">
        <xsl:choose>
        <xsl:when test="(position( ) mod 2) = 1">linen</xsl:when>
    <xsl:otherwise>white</xsl:otherwise>
</xsl:choose>
<xsl:variable>
```

```
        <tr bgcolor="{$bgcolor}"><xsl:apply-templates/></tr>
    </xsl: template>
    <xsl: template match="table/row/cell">
        <td><xsl:apply-templates/></td>
    </xsl: template>
```

Ao montar as duas partes, usei *o Controlador Frontal* (328) para ajudar a separar o código que executa o trabalho.

```
class AlbumCommand...

    public void process ( ) {
        try {
            Álbum álbum = Álbum.buscarPeloNome(solicitação.getParameter("nome"));
            álbum = Álbum.buscarPeloNome("1234");
            Assert.notNull(álbum);
            PrintWriter saída = resposta.getWriter( );
            XsltProcessor processador = new TwoStepXsltProcessor("álbum2.xsl","segundo.xsl");
            saída.print(processador.getTransformation(álbum.toXmlDocument( )));
        } catch (Exception e) {
            throw new ApplicationException (e);
        }
    }
```

É útil comparar esta com a abordagem de estágio único em *Vista de Transformação* (343). Se você quiser alterar as cores das linhas alternadas, a *Vista de Transformação* (343) requer a edição de cada programa XSLT, mas com *Vista em Duas Etapas* apenas o único programa XSLT segunda etapa precisa ser alterado. Poderia ser possível usar padrões que possam ser chamados para fazer algo semelhante, mas isso precisa de um pouco de ginástica XSLT para ser bem-sucedido. O aspecto negativo da *Vista em Duas Etapas* é que o HTML final é muito limitado pelo XML orientado a tela.

Exemplo: JSP e Identificadores Customizados (Java)

Embora o caminho de XSLT seja conceitualmente a forma mais fácil de pensar em implementar *Vista em Duas Etapas*, existem muitas outras maneiras. Para este exemplo, usarei JSPs e identificadores customizados. Embora eles sejam complicados e menos poderosos que XSLT, mostram como o padrão pode se expressar de diferentes modos. Estou sendo um pouco audacioso neste exemplo, pois não vi isto feito em campo. Contudo, acho que um exemplo um pouco especulativo lhe dará uma idéia do que seria possível.

A regra chave de *Vista em Duas Etapas* é que escolher o que exibir e escolher o HTML que o fará estejam totalmente separados. Para este exemplo, meu primeiro estágio é manipulado por uma página JSP e seu auxiliar, e meu segundo estágio por um conjunto de identificadores customizados. A parte interessante do primeiro estágio é a página JSP.

```
<%@ taglib uri = "2passo.tld" prefix = "2passo" %>
<%@ page session = "false"%>
<jsp:useBean id = "auxiliar" class="actionController.AuxiliarConÁlbum"/>
<%auxiliar.init(solicitação, resposta);%>
<2passo:screen>
```

```
<2passo:título><jsp:lerPropriedade name = "auxiliar" property = "título"/></2passo:título>
<2passo:field label = "Artista"><jsp:lerPropriedade name = "auxiliar" property =
"artista"/></2passo:field>
<2passo:table host = "auxiliar" collection = "listaDeFaixas" columns = "título, tempo"/>
</2passo:screen>
```

Estou usando *Controlador de Página* (318) para a página JSP com um objeto auxiliar. Você pode checar *Controlador de Página* (318) para ler mais sobre isso. O importante aqui é olhar os identificadores que fazem parte do `namespace 2passo`. São eles que estou usando para chamar o segundo estágio. Perceba também que não há HTML na página JSP. Os únicos identificadores presentes são ou identificadores do segundo estágio ou identificadores de manipulação de *beans* para ler valores do auxiliar.

Cada identificador de segundo estágio tem uma implementação para obter o HTML necessário para esse elemento da tela lógica. O mais simples deles é o título.

```
class IdentificadorDoTítulo...

    public int executarIdentificadorInicial ( ) throws JspException {
        try {
            pageContext.getOut( ).print("<H1>");
        } catch (IOException e) {
            throw new JspException ("não foi possível imprimir o início");
        }
        return EVAL_BODY_INCLUDE;
    }
    public int executarIdentificadorFinal ( ) throws JspException {
        try {
            pageContext.getOut( ).print("</H1>");
        } catch (IOException e) {
            throw new JspException ("não foi possível imprimir o final");
        }
        return EVAL_PAGE;
    }
```

Para aqueles que não permitiram, um identificador customizado funciona implementando métodos associados chamados, no início e no final do texto, identificado. Este identificador simplesmente envolve o conteúdo do seu corpo com um identificador `<H1>`. Um identificador mais complexo, como o campo, pode receber um atributo. Esse atributo é vinculado na classe do identificador usando um método de gravação.

```
class IdentificadorDoCampo...

    private String rótulo;
    public void gravarRótulo (String rótulo) {
        this.rótulo = rótulo;
    }
```

Assim que o valor tiver sido gravado, você pode usá-lo na saída.

```
class IdentificadorDoCampo...

    public int executarIdentificadorInicial ( ) throws JspException {
        try {
            pageContext.getOut( ).print("<P>" + rótulo + ": <B>");
```

```
        } catch (IOException e) {
            throw new JspException ("não foi possível imprimir o início");
        }
        return EVAL_BODY_INCLUDE;
    }
    public int executarIdentificadorFinal ( ) throws JspException {
        try {
            pageContext.getOut( ).print("</B></P>");
        } catch (IOException e) {
            throw new JspException ("não foi possível imprimir o final");
        }
        return EVAL_PAGE;
    }
```

O `Table` é o mais sofisticado dos identificadores. Além de permitir ao escritor JSP escolher quais colunas colocar na tabela, ele destaca linhas alternadas. A implementação do identificador atua como o segundo estágio, então o destaque é feito lá, de modo que uma alteração no sistema inteiro possa ser feita globalmente.

O identificador `Table` recebe atributos para o nome da propriedade *collection*, o objeto no qual a propriedade *collection* fica, e uma lista de nomes de colunas separados por vírgulas.

```
class IdentificadorDaTabela...

    private String nomeDoConjunto;
    private String nomeDoHospedeiro;
    private String colunas;
    public void gravarConjunto (String nomeDoConjunto) {
        this.nomeDoConjunto = nomeDoConjunto;
    }
    public void gravarHospedeiro (String nomeDoHspedeiro) {
        this.nomeDoHospedeiro = nomeDoHospedeiro;
    }
    public void gravarColunas (String colunas) {
        this.colunas= colunas;
    }
```

Criei um método auxiliar para ler uma propriedade de um objeto. Há um bom argumento para usar as diversas classes que suportam Java *beans*, em vez de apenas chamar um método "lerAlgumaCoisa", mas isso ainda servirá para o exemplo.

```
class IdentificadorDaTabela...

    private Object lerPropriedade (Object obj, String propriedade) throws JspException {
        try{
            String nomeDoMétodo = "ler" + propriedade.substring(0,1).toUpperCase( ) +
                propriedade.substring(1);
            Object resultado = obj.getClasse( ).getMethod(nomeDoMétodo, null).invoke(obj, null);
            return resultado;
        } catch(Exception e) {
            throw new JspException("não foi possível ler a propriedade " + propriedade +
" do objeto " + obj);
        }
    }
```

Este identificador não tem um corpo. Quando ele é chamado, pega o conjunto nomeado da propriedade da solicitação e itera por meio desse conjunto para gerar as linhas da tabela.

```
class IdentificadorDaTabela...

    public int executarIdentificadorInicial ( ) throws JspException {
        try {
            JspWriter saída = pageContext.getOut( );
            saída.print("<table>");
            Collection coll = (Collection) lerPropriedadeDoAtributo(nomeDoHospedeiro, nomeDoConjunto);
            Iterator linhas = coll.iterator( );
            int númeroDaLinha = 0;
            while (linhas.hasNext( )) {
                saída.print("<tr");
                if ( (númeroDaLinha++ % 2) == 0) saída.print(" bgcolor = " + COR_DESTAQUE);
                saída.print(">");
                imprimirCélulas (linhas.next( ));
                saída.print("</tr>");
            }
            saída.print("</table>");
        } catch(IOException e) {
            throw new JspException ("não foi possível imprimir");
        }
        return SKIP_BODY;
    }
    private Object lerPropriedadeDoAtributo (String atributo, String propriedade)
            throws JspException
    {
        Object objetoHospedeiro = pageContext.findAttribute(atributo);
        if (objetoHospedeiro == null)
            throw new JspException ("Atributo " + atributo + " não encontrado.");
        return lerPropriedade(objetoHospedeiro, propriedade);
    }
    public static final String COR_DESTAQUE = "'linen'";
```

Durante a iteração, ele configura uma linha sim outra não com a cor de fundo usada para dar destaque a ela.

Para imprimir as células de cada linha, uso os nomes das colunas como sendo os valores da propriedade nos objetos do conjunto.

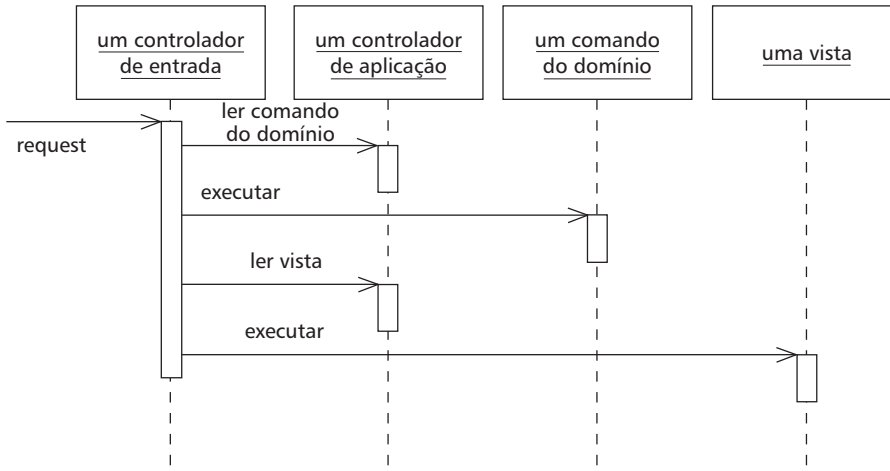
```
class IdentificadorDeTabela...

    private void imprimirCélulas (Object obj) throws IOException, JspException {
        JspWriter saída = pageContext.getOut( );
        for (int i = 0; i< lerListaDeColunas( ).length; i++) {
            saída.print("<td>");
            saída.print(lerPropriedade(obj, lerListaDeColunas( ) [i] ));
            saída.print("</td>");
        }
    }
    private String [ ] lerListaDeColunas ( ) {
        StringTokenizer tk = new StringTokenizer (colunas, ",");
        String [ ] resultado = new String[tk.countTokens( )];
```

```
        for (int i = 0; tk.hasMoreTokens; i++)
            resultado[i] = tk.nextToken( );
        return resultado;
    }
```

Comparada com a implementação XSLT, esta solução é menos limitante quanto à uniformidade da aparência do *site*. Um autor de uma página, querendo mover algum HTML individual para ela, achará isso fácil de ser feito. É claro que, embora permita o ajuste de páginas de projeto pesado, isso também está exposto ao uso inapropriado por pessoas não-familiarizadas com o modo como as coisas funcionam. Às vezes, restrições ajudam a prevenir erros. É um balanceamento que uma equipe tem que decidir por si própria.

Controlador de Aplicação (Application Controller)

Um ponto centralizado para manipular navegação de tela e o fluxo de uma aplicação.



Algumas aplicações contêm uma quantidade significativa de lógica relativa a telas a serem usadas em diferentes pontos, o que pode envolver a chamada de certas telas em determinados momentos em uma aplicação. Este é o estilo especialista de interação, em que o usuário é levado por meio de uma série de telas em uma determinada ordem. Em outros casos, podemos ver telas que só são trazidas sob determinadas condições, ou escolhas entre diferentes telas que dependem de entradas anteriores.

Até certo ponto, os vários controladores de entrada do *Modelo Vista Controlador* (315) podem tomar algumas dessas decisões, mas, à medida que a aplicação fica mais complexa, isso pode levar a código duplicado pois diversos controladores para diferentes telas precisam saber o que fazer em uma determinada situação.

Você pode remover esta duplicação colocando toda a lógica do fluxo em um *Controlador de Aplicação*. Controladores de entradas solicitam, então, ao *Controlador de Aplicação* os comandos apropriados para execução sobre um modelo e a vista correta a usar dependendo do contexto da aplicação.

Como Funciona

Um *Controlador de Aplicação* tem duas responsabilidades principais: decidir qual lógica de domínio executar e a vista com a qual exibir a resposta. Para fazer isso, geralmente ele armazena dois conjuntos estruturados de referências de classes, um para comandos do domínio executarem na camada do domínio e um de vistas. (Figura 14.10).

Tanto para os comandos do domínio quanto para as vistas, o controlador de aplicação precisa de uma maneira de armazenar algo que possa chamar. Um Comando [Gang of Four] é uma boa escolha, já que permite pegar e executar facilmente um bloco de código. Linguagens que podem manipular funções podem armazenar refe-

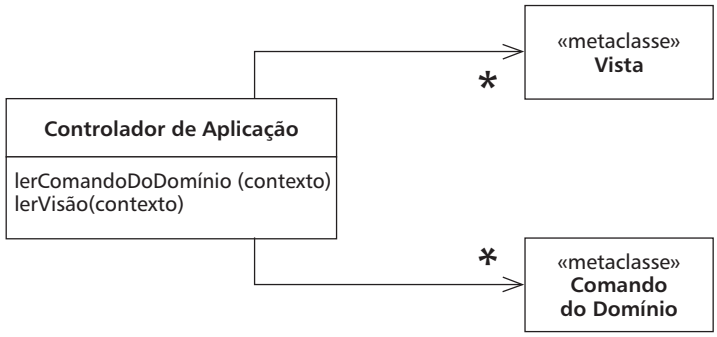


Figura 14.10 Um controlador de aplicação tem dois conjuntos de referências a classes, um para a lógica de domínio e outro para as visões.

rências a elas. Outra opção é armazenar uma *string* que possa ser usada para chamar um método usando reflexão.

Os comandos do domínio podem ser objetos comando que são parte da camada do *Controlador de Aplicação*, ou podem ser referências a um *Roteiro de Transação* (120) ou métodos de objetos do domínio na camada de domínio.

Se você estiver usando páginas servidoras como suas vistas, pode usar o nome da página servidora. Se estiver usando uma classe, um comando ou uma *string* para uma chamada reflexiva faz sentido. Você também poderia usar uma transformação XSLT, para a qual o *Controlador de Aplicação* pode armazenar uma *string* como uma referência.

Uma decisão que você precisará tomar é como separar o *Controlador de Aplicação* do resto da apresentação. A princípio, esta decisão se manifesta na forma de se o *Controlador de Aplicação* tem dependências com o mecanismo da interface de usuário. Talvez ele acesse diretamente os dados da sessão HTTP, passe adiante para uma página servidora ou chame métodos em uma classe cliente rica.

Embora eu tenha visto *Controladores de Aplicação* diretos, minha preferência é que os *Controladores de Aplicação* não tenham conexões para o mecanismo de interface de usuário. Para começar, isso torna possível testar o *Controlador de Aplicação* independentemente da interface de usuário, o que é um grande benefício. Também é importante fazer isso se você for usar o mesmo *Controlador de Aplicação* com diversas apresentações. Por esses motivos, muitas pessoas gostam de pensar no *Controlador de Aplicação* como uma camada intermediária entre a apresentação e o domínio.

Uma apresentação pode ter diversos *Controladores de Aplicação* para manipular cada uma de suas diferentes partes. Isso permite a você dividir lógica complexa em diversas classes. Neste caso, geralmente faço com que o trabalho seja dividido em áreas gerais da interface do usuário e crio *Controladores de Aplicação* separados para cada área. Em uma aplicação mais simples, eu poderia precisar de apenas um *Controlador de Aplicação*.

Se você tiver diversas apresentações, como um *front end* Web, um cliente rico e um PDA, pode ser capaz de usar o mesmo *Controlador de Aplicação* para cada apresentação, mas não fique muito ávido. Muitas vezes, diferentes interfaces de usuário precisam de um fluxo de tela diferente para obter uma interface de usuário realmente utilizável. Contudo, reutilizar um único *Controlador de Aplicação* pode reduzir o trabalho de desenvolvimento, e isso pode valer o custo de uma interface de usuário mais complexa.

Uma maneira comum de pensar em uma interface de usuário é como uma máquina de estados, em que determinados eventos disparam diferentes respostas dependendo do estado de determinados objetos chave da aplicação. Neste caso, o *Controlador de Aplicação* é particularmente acessível ao uso de metadados para representar o fluxo de controle da máquina de estados. Os metadados podem ser configurados por chamadas de linguagem de programação (o modo mais simples) ou podem ser armazenados em um arquivo de configuração separado.

Você pode encontrar lógica de domínio específica de uma solicitação localizada em um *Controlador de Aplicação* (360). Como você pode suspeitar, não sou muito favorável a esta noção. Entretanto, o limite entre lógica de domínio e de aplicação fica muito obscuro. Digamos que eu esteja lidando com aplicações de seguro e precise mostrar uma tela separada com perguntas apenas se o requerente for fumante. Isso é lógica de aplicação ou lógica de domínio? Se eu tiver apenas alguns casos desse tipo, provavelmente posso colocar este tipo de lógica no *Controlador de Aplicação* (360), mas se ele ocorrer em muitos lugares, preciso projetar o *Modelo de Domínio* (126) de uma maneira a conduzir isso.

Quando Usá-lo

Se o fluxo e a navegação de sua aplicação forem simples o suficiente para que qualquer pessoa possa visitar qualquer tela em qualquer ordem, há pouco valor em um *Controlador de Aplicação*. A força de um *Controlador de Aplicação* vem de regras definidas sobre a ordem na qual páginas devem ser visitadas e vistas diferentes, dependendo do estado dos objetos.

Um bom sinal para usar um *Controlador de Aplicação* é se você se encontrar tendo que fazer alterações similares em muitos lugares diferentes quando o fluxo de sua aplicação muda.

Leitura Adicional

A maioria das idéias que baseiam a escrita deste padrão vieram de [Knight and Dai]. Embora suas idéias não sejam exatamente novas, considero suas explanações notavelmente claras e atraentes.

Exemplo: *Controlador de Aplicação* Modelo de Estados (Java)

Os modelos de estados são um modo comum de pensar em interfaces de usuário. Eles são especialmente apropriados quando você precisa reagir de maneira diferente a eventos dependendo do estado de algum objeto. Neste exemplo, tenho um modelo de estados simples para alguns comandos em um bem (Figura 14.11). Os especialistas de *leasing* da ThoughtWork desmaiariam com o excesso de simplificação virulenta deste modelo, mas ele servirá como exemplo de um *Controlador de Aplicação* baseado em estados.

No que se refere ao código, nossas regras são as seguintes:

- Quando recebemos um comando de retorno e estamos no estado Em *Lease*, exibimos uma página para capturar informações sobre o retorno do bem.
- Um evento de retorno no estado Em Estoque é um erro, então mostramos uma página de ação ilegal.

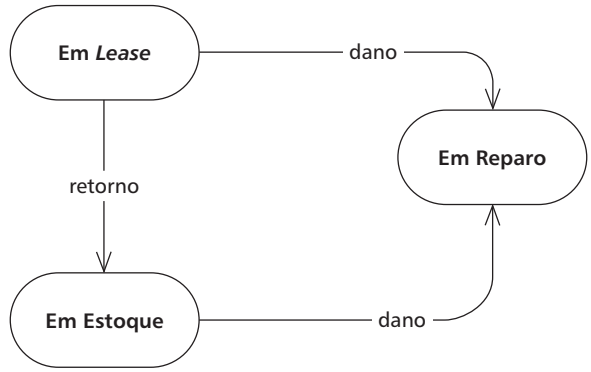


Figura 14.11 Um diagrama de estados simples para um bem.

- Quando recebemos um comando de dano, mostramos diferentes páginas, dependendo de se o bem está no estado Em Estoque ou no Em *Lease*.

O controlador de entrada é um *Controlador Frontal* (328). Ele serve a solicitação desta forma:

```
class FrontServlet...

    public void executarServiço (HttpServletRequest solicitação, HttpServletResponseresposta)
            throws IOException, ServletException
    {
        ControladorDeAplicação contApl = lerControladorDeAplicação (solicitação);
        String stringDoComando = (String) solicitação.getParameter("comando");
        ComandoDeDomínio com =
            contApl.lerComandoDeDomínio(stringDoComando, lerMapaDeParâmetros(solicitação));
        com.executar(lerMapaDeParâmetros(solicitação));
        String páginaVisão =
            "/" + contApl.lerVisão(stringDoComando, lerMapaDeParâmetros(solicitação)) + ".jsp";
        forward(páginaVisão, solicitação, resposta);
    }
```

O fluxo do método de serviço é bastante direto: descobrimos o controlador de aplicação correto para uma dada solicitação, solicitamos ao controlador de aplicação o comando do domínio, executamos esse comando do domínio, solicitamos ao controlador de aplicação uma vista e, finalmente, passamos adiante para a vista.

Neste esquema, estou pressupondo um número de *Controladores de Aplicação*, os quais implementam a mesma interface.

```
interface ControladorDeAplicação...

    ComandoDeDomínio lerComandoDeDomínio (String stringDoComando, Map parâmetros);
    String lerVisão (String stringDoComando, Map parâmetros);
```

Para nossos comandos o *Controle de Aplicaçã*o apropriado é um controle de aplicação para bens. Ele usa uma classe de resposta para armazenar as referências a vistas e comandos de domínio. Para o comando de domínio uso uma referência a uma

classe. Para a vista, uso uma *string*, a qual o controlador de aplicação transformará em uma URL para uma JSP.

```
class Resposta...

    private Class comandoDeDomínio;
    private String UrlDaVisão;
    public Resposta (Class comandoDeDomínio, String UrlDaVisão) {
        this.comandoDeDomínio = comandoDeDomínio;
        this.UrlDaVisão = UrlDaVisão;
    }
    public ComandoDeDomínio lerComandoDeDomínio( ) {
        try {
            return (ComandoDeDomínio) comandoDeDomínio.newInstance( );
        } catch (Exception e) { throw new ApplicationException (e);
        }
    }
    public String lerUrlDaVisão( ) {
        return UrlDaVisão;
    }
```

O controlador de aplicação segura as respostas, usando um mapa de mapas indexados pela *string* do comando e a situação atual do bem (Figura 14.12).

```
class ControladorDeAplicaçãoParaBens...

    private Resposta getResponse (String strindDoComando, SituaçãoDoBem estado) {
        return (Response) lerMapaDeRespostas(strindDoComando).get(estado);
    }
    private Map lerMapaDeRespostas (String chave) {
        return (Map) eventos.get(chave);
    }
    private Map eventos = new HashMap( );
```

Ao ser solicitado por um comando do domínio, o controlador olha a solicitação para descobrir o ID do bem, vai ao domínio para determinar a situação desse bem, procura a classe apropriada do comando do domínio, instancia essa classe e retorna o novo objeto.

```
class ControladorDeAplicaçãoParaBens...

    public ComandoDeDomínio lerComandoDeDomínio (String stringDoComando, Map parâmetros) {
        Resposta resposta = getResponse (stringDoComando, lerSituaçãoDoBem(parâmetros));
        return resposta.lerComandoDeDomínio( );
    }
    private SituaçãoDoBem lerSituaçãoDoBem (Map parâmetros) {
        String id = lerParâmetros("IDdoBem", parâmetros);
        Bem bem = Bem.buscar(id);
        return bem.lerSituação( );
    }
    private String lerParâmetros (String chave, Map parâmetros) {
        return ((String[ ]) parâmetros.get(chave))[0];
    }
```

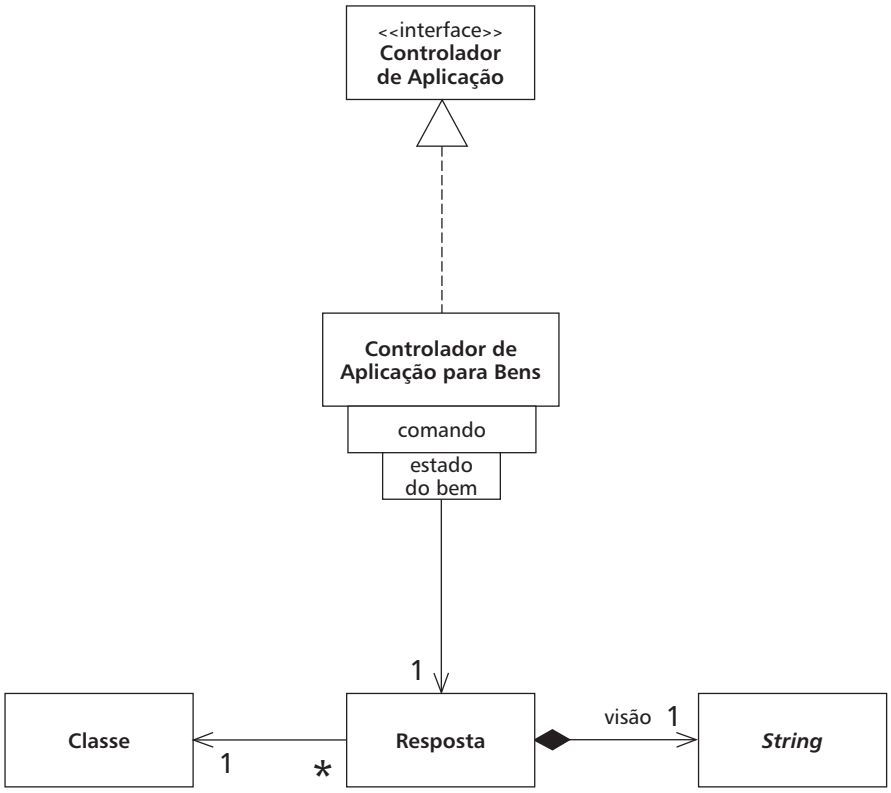


Figura 14.12 Como o controlador de aplicação para bens armazena suas referências a comandos do domínio e visões.

Todos os comandos de domínio seguem uma interface simples que permite ao controlador frontal executá-los.

```
interface ComandoDeDomínio...

    abstract public void executar(Map parâmetros);
```

Assim que o comando do domínio tiver feito o que precisa, o *Controlador de Aplicação* entra em ação novamente quando a vista lhe é solicitada.

```
class ControladorDeAplicaçãoParaBens...

    public String lerVisão (String stringDoComando, Map parâmetros) {
        return getResponse (stringDoComando, lrPosiçãoDoBem(parâmetros)).lerUrlDaVisão( );
    }
```

Neste caso, o *Controlador de Aplicação* não retorna a URL inteira para a JSP. Ele retorna uma *string* que o controlador frontal transforma em uma URL. Faço isso para evitar duplicar os caminhos da URL nas respostas. Isso também torna fácil adicionar mais indireção, mais tarde, se eu precisar.

O *Controlador de Aplicação* pode ser carregado para uso com código.

```
class ControladorDeAplicaçãoParaBens...

    public void adicionarResposta (String evento, Objeto estado, Class comandoDeDomínio, String visão) {
    Resposta novaResposta = new Resposta (comandoDeDomínio, visão);
    if ( !eventos.containsKey(evento))
        eventos.put(evento, new HashMap( ));
    lerMapaDeRespostas(evento).put(estado, novaResposta);
    }
    private static void carregarControladorDeAplicação(ControladorDeAplicaçãoParaBens contApl) {
        contApl = ControladorDeAplicaçãoParaBens.getDefault( );
        contApl.adicionarResposta("retorno", SituaçãoDoBem.EM_LEASE,
                          ComandoJuntarDetalhesDoRetono.class, "retorno");
        contApl.adicionarResposta("retorno", SituaçãoDoBem.EM_ESTOQUE,
                          ComandoBemNulo.class, "açãoIlegal");
        contApl.adicionarResposta("retorno", SituaçãoDoBem.EM_LEASE,
                          ComandoDanosEstoque.class, "danoLease");
        contApl.adicionarResposta(retorno", SituaçãoDoBem.EM_ESTOQUE,
                          ComandoDanoNoLease.class, "DanoEstoque");
    }
```

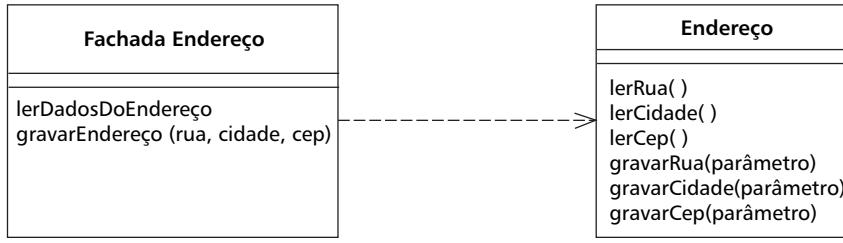
Fazer isso a partir de um arquivo não é muito difícil, mas, mesmo assim, deixarei para você.

CAPÍTULO 15

Padrões de Distribuição

Fachada Remota (Remote Façade)

Fornece uma fachada de granularidade alta sobre objetos de granularidade baixa para melhorar a eficiência em uma rede.



Em um modelo orientado a objetos, você obtém melhor desempenho com objetos pequenos que tenham métodos pequenos. Isso lhe dá muitas oportunidades de controle e substituição de comportamento e de usar nomes sugestivos que deixem uma aplicação mais fácil de ser entendida. Uma das conseqüências deste comportamento de granularidade baixa é que geralmente há bastante interação entre os objetos, e esta interação normalmente requer muitas chamadas de métodos.

Dentro de um espaço de endereçamento único, interação com granularidade baixa funciona bem, mas não ocorre quando você executa chamadas entre processos. Chamadas remotas são muito mais custosas porque há muito a ser feito: dados podem ter que ser preparados, a segurança pode precisar ser verificada, pacotes podem necessitar ser roteados por meio de *switches*. Se os dois processos estiverem rodando em máquinas em lados opostos do globo, a velocidade da luz pode ser um fator. A verdade brutal é que qualquer chamada interprocessos é ordem de magnitude mais custosa do que uma chamada interna ao processo – mesmo se ambos os processos estiverem na mesma máquina. Esse efeito no desempenho não pode ser ignorado, mesmo por quem acredita em otimização tardia.

O resultado é que qualquer objeto o qual se pretenda usar como objeto remoto precisa de uma interface de granularidade alta que minimize o número de chamadas necessárias para que algo seja realizado. Isso não afeta apenas suas chamadas de métodos, como também seus objetos. Em vez de solicitar um pedido e os itens desse pedido individualmente, você precisa acessar e atualizar o pedido e seus itens em uma única chamada, o que afeta a estrutura inteira do seu objeto. Você desiste da intenção clara e do controle de granularidade baixa que obtém com objetos e métodos pequenos. Programar se torna mais difícil, e sua produtividade diminui.

A *Fachada Remota* é uma fachada de granularidade alta [Gang of Four] em uma rede de objetos de granularidade baixa. Nenhum dos objetos de granularidade baixa tem uma interface remota, e a *Fachada Remota* não contém lógica de domínio. Tudo o que a *Fachada Remota* faz é traduzir métodos de granularidade alta para os objetos de granularidade baixa associados.

Como Funciona

A *Fachada Remota* lida com o problema de distribuição da abordagem OO padrão de separar responsabilidades distintas em diferentes objetos e como resultado tem se tornado o padrão mais adotado para este problema. Reconheço que objetos de gra-

nularidade baixa são a resposta correta para a lógica complexa, então asseguro-me de que cada lógica complexa seja colocada em objetos de granularidade baixa que são projetados para colaborar dentro de um único processo. Para permitir acesso remoto eficiente a eles, crio um objeto fachada separado que atua como uma interface remota. Como o nome sugere, a fachada é meramente uma camada fina que muda de uma interface de granularidade alta para uma de granularidade baixa.

Em um caso simples, como um objeto para endereços, uma *Fachada Remota* substitui todos os métodos de gravação e leitura do objeto de endereços normal por um método de gravação e um de leitura, muitas vezes, chamados de **métodos de acesso abrangentes**. Quando um cliente chama um método de gravação abrangente, a fachada de endereços lê os dados do método de gravação e chama os métodos de acesso individuais no objeto de endereços real (veja a Figura 15.1) e não faz mais nada. Dessa forma, toda a lógica de validação e computação permanece no objeto de endereços no qual pode ser fatorada de forma limpa e pode ser usada por outros objetos de granularidade baixa.

Em um caso mais complexo, uma única *Fachada Remota* pode atuar como um *gateway* remoto para muitos objetos de granularidade baixa. Por exemplo, uma fachada para pedidos pode ser usada para obter e atualizar informações sobre um pedido, todos seus itens e talvez alguns dados do cliente também.

Ao transferir dados em volumes como este, você precisa de que eles estejam em um formato que possa ser movido facilmente pela conexão. Se suas classes de granularidade baixa estiverem presentes nos dois lados da conexão e forem serializáveis, você pode transferi-las diretamente fazendo uma cópia. Neste caso, um método `lerDadosDoEndereço` cria uma cópia do objeto endereço original. O `gravarDadosDoEndereço` recebe um objeto endereço e o usa para atualizar os dados do objeto endereço real. (Isso pressupõe que o objeto endereço original precisa preservar sua identidade e, assim, não pode simplesmente ser substituído pelo novo endereço.)

Entretanto, muitas vezes, você não pode fazer isso. Você pode não querer duplicar suas classes de domínio em diversos processos, ou pode ser difícil serializar um segmento de um modelo do domínio devido a sua estrutura de relacionamentos

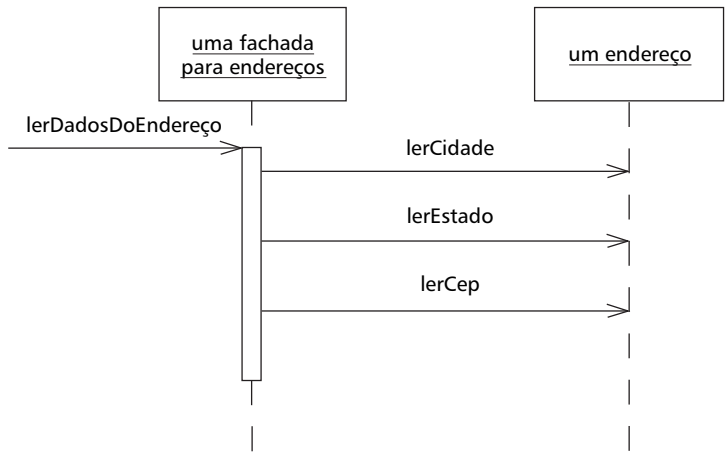


Figura 15.1 Uma chamada para uma fachada produz diversas chamadas da fachada para o objeto do domínio.

complicada. O cliente pode não querer o modelo inteiro, mas apenas um subconjunto simplificado dele. Nestes casos, faz sentido usar um *Objeto de Transferência de Dados* (380) como base da transferência.

No esboço, mostrei uma *Fachada Remota* que corresponde a um único objeto do domínio. Isso não é incomum e é fácil de entender, mas não é o caso mais comum. Uma única Fachada Remota teria vários métodos, cada um projetado para passar informações de diversos objetos. Assim, `lerDadosDoEndereço` e `gravarDadosDoEndereço` seriam métodos definidos em uma classe como `ServiçoCliente`, a qual também teria métodos como `lerHistóricoDeCompras` e `atualizarDadosDeCrédito`.

A granularidade é uma das questões mais traiçoeiras em *Fachadas Remotas*. Algumas pessoas gostam de criar *Fachadas Remotas* razoavelmente pequenas, como uma por caso de uso. Prefiro uma estrutura de granularidade mais grossa com muito menos *Fachadas Remotas*. Mesmo para uma aplicação de tamanho moderado eu poderia ter apenas uma e, até para uma aplicação grande, posso ter apenas meia dúzia. Isso significa que cada *Fachada Remota* tem muitos métodos, mas já que esses métodos são pequenos, não vejo isso como um problema.

Você projeta uma *Fachada Remota* baseado nas necessidades de uso de um cliente em particular – mais comumente a necessidade de visualizar e atualizar informações através de uma interface de usuário. Neste caso, você poderia ter uma única *Fachada Remota* para um grupo de telas, para cada uma das quais um método de acesso abrangente transporta e grava os dados. Pressionar botões em uma tela, digamos para alterar a situação de um pedido, chama métodos de comando na fachada. Muitas vezes, você terá diferentes métodos na *Fachada Remota* que fazem quase que a mesma coisa nos objetos correspondentes. Isso é comum e razoável. A fachada é projetada para tornar a vida de usuários externos mais simples, não para o sistema interno, então se o processo cliente pensa nele como um comando diferente, é um comando diferente, mesmo se internamente tudo for direcionado para o mesmo comando.

A *Fachada Remota* pode manter seu estado ou não. Uma *Fachada Remota* sem estados pode ser usada na criação de um *pool*, o que pode melhorar a utilização e a eficiência dos recursos, especialmente em uma situação B2C*. Entretanto, se a interação envolver estado por meio de uma sessão, então a Fachada Remota precisa armazenar o estado da sessão em algum lugar, usando *Estado da Sessão no Cliente* (427) ou *Estado da Sessão no Banco de Dados* (432), ou uma implementação de *Estado da Sessão no Servidor* (429). Se a *Fachada Remota* tiver que armazenar seu próprio estado, ela será facilmente implementável por um *Estado da Sessão no Servidor* (429), mas isso pode levar a questões de desempenho quando você tiver milhares de usuários simultâneos.

Assim como fornecer uma interface de granularidade alta, diversas outras responsabilidades podem ser acrescentadas a uma *Fachada Remota*. Por exemplo, seus métodos são um ponto natural no qual aplicar segurança. Uma lista de controle de acesso pode dizer quais usuários podem chamar quais métodos. Os métodos da *Fachada Remota* também são um ponto natural no qual aplicar controle transacional. Um método da *Fachada Remota* pode começar uma transação, executar todo o trabalho interno e, então, confirmar a transação no final. Cada chamada constitui uma transação completa porque você não quer uma transação aberta quando o retorno

* N de R. T.: Business to Consumer.

volta para o cliente, já que transações não são criadas para serem eficientes em casos de execução tão longa.

Um dos maiores erros que vejo em uma *Fachada Remota* é colocar lógica de domínio nela. Repita comigo três vezes: "*Fachada Remota* não tem lógica de domínio." Qualquer fachada deve ser uma camada fina que tenha apenas responsabilidades mínimas. Se você precisar de lógica de domínio para fluxo de trabalho ou coordenação, coloque-a nos seus objetos de granularidade baixa ou crie um *Roteiro de Transação* (120) separado e não-remoto para contê-la. Você deve poder executar a aplicação inteira localmente sem usar as *Fachadas Remotas* ou ter que duplicar qualquer código.

***Fachada Remota* e Fachada de Sessão** Nos últimos anos, o padrão Fachada de Sessão [Alur *et al.*] tem aparecido na comunidade J2EE. Nos meus primeiros esboços, considerei Fachada de Sessão o mesmo padrão de *Fachada Remota* e usei o nome Fachada de Sessão. Na prática, entretanto, há uma diferença crucial. *Fachada Remota* diz respeito a ter uma camada fina remota – por esta razão, minha crítica contra lógica de domínio nela. Ao contrário, a maior parte das descrições de Fachada de Sessão envolvem a colocação de lógica nela, normalmente de um tipo de fluxo de trabalho. Uma grande parte disso é devido à abordagem comum de usar *session beans* J2EE para encapsular *entity beans*. Qualquer coordenação de *entity beans* tem que ser feita por outro objeto, já que eles não podem ser reentrantes.

Como resultado disso, vejo uma Fachada de Sessão como colocar diversos *Roteiros de Transação* (120) em uma interface remota. É uma abordagem razoável, mas não é a mesma coisa que uma *Fachada Remota*. De fato, eu argumentaria que, como Fachada de Sessão contém lógica de domínio, não deveria ser chamada de fachada!

Camada de Serviço Um conceito familiar a fachadas é uma *Camada de Serviço* (141). A principal diferença é que uma camada de serviço não tem que ser remota e deste modo não precisa ter apenas métodos de granularidade baixa. Ao simplificar o *Modelo de Domínio* (126), você muitas vezes acaba com métodos de granularidade alta, mas isso por clareza, não por eficiência da rede. Além disso, não há necessidade de uma camada de serviço usar *Objetos de Transferência de Dados* (380). Normalmente, ela pode facilmente retornar objetos do domínio reais para o cliente.

Se um *Modelo do Domínio* (126) for ser usado dentro do processo e também remotamente, você pode ter uma *Camada de Serviço* (141) e colocar uma *Fachada Remota* separada sobre ela. Se o processo só é usado remotamente, provavelmente é mais fácil incluir a *Camada de Serviço* (141) na *Fachada Remota*, supondo que a *Camada de Serviço* (141) não tenha lógica de aplicação. Se houver alguma lógica de aplicação nela, então eu faria da *Fachada Remota* um objeto separado.

Quando Usá-la

Use *Fachada Remota* sempre que precisar de acesso remoto a um modelo de objetos de granularidade baixa. Você obtém as vantagens de uma interface de granularidade alta enquanto mantém a vantagem dos objetos de granularidade baixa, dando-lhe o melhor dos dois mundos.

O uso mais comum deste padrão é entre uma apresentação e um *Modelo de Domínio* (126), onde os dois podem rodar em processos diferentes. Você obterá isso entre uma interface de usuário *Swing* e modelo de domínio no servidor ou com um *ser-*

vlet e um modelo de objetos no servidor se a aplicação e os servidores Web forem processos diferentes.

Mais freqüentemente, você se depara com isso com diferentes processos em diferentes máquinas, mas o custo de uma chamada entre processos no mesmo contâiner acaba sendo suficientemente grande para você precisar de uma interface de granularidade alta para qualquer comunicação entre processos, independentemente de onde eles estejam.

Se todo seu acesso estiver dentro de um único processo, você não precisa desse tipo de conversão. Assim, eu não usaria este padrão para comunicação entre um cliente *Modelo de Domínio* (126) e sua apresentação ou entre um *script* CGI e um *Modelo de Domínio* (126) rodando em um servidor Web. Você não vê *Fachada Remota* usada com um *Roteiro de Transação* (120) como regra, já que *Roteiro de Transação* (120) tem inerentemente granularidade alta.

As *Fachadas Remotas* sugerem um estilo de distribuição síncrono – ou seja, uma chamada de procedimento remoto. Muitas vezes, você pode melhorar bastante a capacidade de resposta de uma aplicação com comunicação remota assíncrona e baseada em mensagens. De fato, uma abordagem assíncrona tem muitas vantagens atrativas. Infelizmente, a discussão de padrões assíncronos está fora do escopo deste livro.

Exemplo: Usando um *Session Bean* Java como *Fachada Remota* (Java)

Se você estiver trabalhando com a plataforma Java Enterprise, uma boa escolha para uma fachada distribuída é um *session bean*, porque é um objeto remoto que pode ter ou não estado. Neste exemplo, executarei alguns POJOs (velhos e bons objetos Java) dentro de um contâiner EJB e irei acessá-los remotamente através de um *session bean* que é projetado como uma *Fachada Remota*. *Session beans* não são especialmente complicados, então tudo deve fazer sentido mesmo se você nunca tenha trabalhado com eles antes.

Sinto a necessidade de algumas observações aqui. Primeiro, fui surpreendido pelo número de pessoas que parecem acreditar que você não pode rodar objetos comuns dentro de um contâiner EJB em Java. Ouço as perguntas "Os objetos do domínio são *entity beans*?". A resposta é que eles podem, mas não precisam sê-lo. Os objetos Java simples funcionam bem, como neste exemplo.

Minha segunda observação é apenas para destacar que esta não é a única maneira de usar *session beans*. Eles também podem ser usados para hospedar *Roteiros de Transação* (120).

Neste exemplo, olharei interfaces remotas para acessar informações sobre álbuns musicais. O *Modelo de Domínio* (126) consiste em objetos de granularidade baixa que representam um artista, álbum e faixas. Em torno disso, estão diversos outros pacotes que fornecem as fontes de dados para a aplicação (veja a Figura 15.2).

Na figura, o pacote *otd* contém *Objetos de Transferências de Dados* (380) que ajudam a mover os dados pela conexão até o cliente. Eles têm comportamento simples de acesso simples e também a habilidade de serializar a si mesmos em formato binário ou XML textual. No pacote remoto, estão objetos montadores que movem dados entre os objetos do domínio e os *Objetos de Transferência de Dados* (380). Se você estiver interessado em como isso funciona, veja a discussão sobre *Objetos de Transferência de Dados* (380).

Para explicar a fachada, irei pressupor que posso mover dados de e para os *Objetos de Transferência de Dados* (380) e concentrarei nas interfaces remotas. Um

único *session bean* lógico Java tem três classes reais. Duas delas fazem a API remota (e na verdade são interfaces Java). A outra é a classe que implementa a API. As duas interfaces são `ÁlbumService` e o objeto `ÁlbumHome`. Este objeto *home* é usado pelo serviço de nomeação para obter acesso à fachada distribuída, mas este é um detalhe EJB que pularei aqui. Nosso interesse é na própria *Fachada Remota*, `ÁlbumService`. Sua interface é declarada no pacote API para ser usada pelo cliente e é apenas uma lista de métodos.

```
class ÁlbumService...

    String tocar (String id) throws RemoteException;
    String lerXmlÁlbum (String id) throws RemoteException;
    ÁlbumOTD lerÁlbum (String id) throws RemoteException;
    void criarÁlbum (String id, String xml) throws RemoteException;
    void criarÁlbum (String id, ÁlbumOTD otd) throws RemoteException;
    void atualizarÁlbum (String id, String xml) throws RemoteException;
    void atualizarÁlbum (String id, ÁlbumOTD otd) throws RemoteException;
    void adicionarArtistaChamado (String id, String nome) throws RemoteException;
    void adicionarArtista (String id, String xml) throws RemoteException;
    void adicionarArtista (String id, ArtistaOTD otd) throws RemoteException;
    ArtistaOTD lerArtista (String id) throws RemoteException;
```

Perceba que, mesmo nesse exemplo curto, vejo métodos para duas classes diferentes no *Modelo de Domínio* (126): artista e álbum. Também vejo variações menores

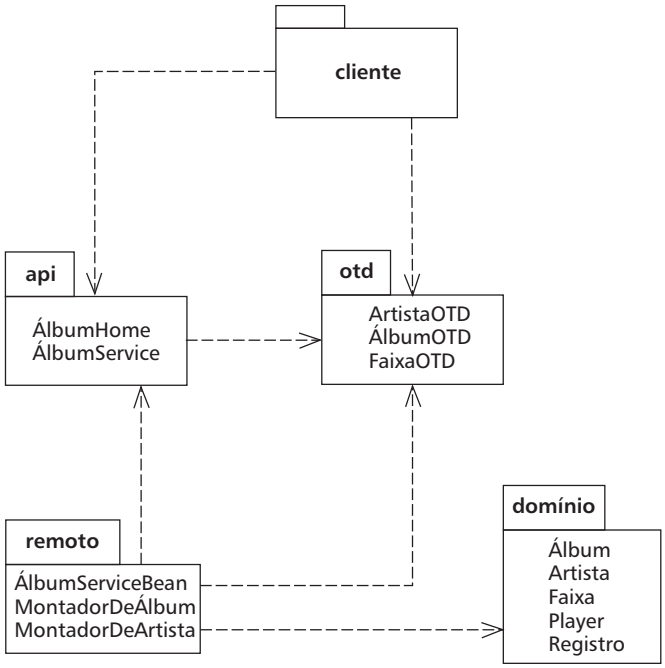


Figura 15.2 Pacotes das interfaces remotas.

no mesmo método. Os métodos têm variantes que usam o *Objeto de Transferência de Dados* (380) ou uma *string* XML para mover dados para o serviço remoto. Isso permite que o cliente escolha qual forma usar dependendo da natureza do cliente e da conexão. Como você pode ver, mesmo para uma aplicação pequena isso pode levar a muitos métodos em `ÁlbumService`.

Felizmente, os próprios métodos são muito simples. Aqui estão os métodos para manipular álbuns:

```
class ÁlbumServiceBean...

    public ÁlbumOTD lerÁlbum(String id) throws RemoteException {
        return new MontadorDeÁlbum( ).gravarOTD(Registro.encontrarÁlbum(id));
    }
    public String lerXmlÁlbum(String id) throws RemoteException {
        ÁlbumOTD otd = new MontadorDeÁlbum( ).gravarOTD(Registro.encontrarÁlbum(id));
        return otd.toXmlString( );
    }
    public void criarÁlbum (String id, ÁlbumOTD otd) throws RemoteException {
        new MontadorDeÁlbum( ).criarÁlbum(id, otd);
    }
    public void criarÁlbum (String id, String xml) throws RemoteException {
        ÁlbumOTD otd = ÁlbumOTD.lerStringXml(xml);
        new MontadorDeÁlbum( ).criarÁlbum(id, otd);
    }
    public void atualizarÁlbum (String id, ÁlbumOTD otd) throws RemoteException {
        new MontadorDeÁlbum( ).atualizarÁlbum(id, otd);
    }
    public void criarÁlbum (String id, String xml) throws RemoteException {
        ÁlbumOTD otd = ÁlbumOTD.lerStringXml(xml);
        new MontadorDeÁlbum( ).atualizarÁlbum(id, otd);
    }
```


Como você pode ver, cada método realmente não faz nada mais do que delegar a outro objeto, de modo que têm o tamanho de apenas uma linha ou duas. Este fragmento ilustra muito bem como uma fachada distribuída deve se parecer: uma longa lista de métodos muito curtos com muito pouca lógica neles. A fachada então não é nada mais do que um mecanismo de empacotamento, que é como deveria ser.

Iremos terminar com algumas palavras sobre teste. É muito útil poder executar tanto teste quanto possível em um único processo. Neste caso, posso escrever testes para a implementação do *session bean* diretamente: estes podem ser executados sem distribuir para o contâiner EJB.

```
class testadorXml...

    private ÁlbumOTD kob;
    private ÁlbumOTD novokob;
    private ÁlbumServiceBean fachada = new ÁlbumServiceBean( );
    protected void configurar ( ) throws Exception {
        fachada.inicializarParaTeste( );
        kob = fachada.lerÁlbum("kob");
        Writer buffer = new StringWriter( );
```

```
        kob.toXmlString(buffer);
        newkob = ÁlbumOTD.lerStringXml(new StringReader(buffer.toString( )));
    }
    public void testarArtista( ) {
        assertEquals(kob.lerArtista( ), newkob.lerArtista( ));
    }
```

Este foi um dos testes JUnit a serem executados na memória. Ele mostrou como posso criar uma instância do *session bean* fora do contâiner e executar testes nele, permitindo que o tempo para testes seja mais rápido.

Exemplo: Serviço Web (C#)

Estava falando sobre este livro com Mike Hendrickson, meu editor na Addison-Wesley. Sempre alerta a novas palavras da moda, ele me perguntou se eu tinha alguma coisa sobre serviços Web nele. Sou avesso a sair correndo a cada nova moda – afinal, dada a velocidade lenta da publicação de livros, qualquer coisa sobre a qual eu escreva classificada como "última moda" parecerá estranha quando você ler. Ainda assim, é um bom exemplo de como padrões essenciais mantêm com tanta freqüência seu valor mesmo com as últimas mudanças tecnológicas.

Um serviço Web é basicamente nada mais do que uma interface para uso remoto (com um passo lento de análise de *strings* acrescentado como procedimento vantajoso). Como tal, o conselho da *Fachada Remota* permanece: construa sua funcionalidade de uma maneira que apresente granularidade baixa e, então, coloque uma *Fachada Remota* sobre o modelo de granularidade baixa para manipular os serviços Web.

Para o exemplo, usarei o mesmo problema básico que descrevi anteriormente, mas concentrarei apenas na solicitação de informações sobre um único álbum. A Figura 15.3 mostra as diversas classes que participam: serviço de álbum, a *Fachada Remota*, dois *Objetos de Transferência de Dados* (380), três objetos em um *Modelo de Domínio* (126) e um montador para trazer dados do *Modelo de Domínio* (126) para os *Objetos de Transferência de Dados* (380).

O *Modelo de Domínio* (126) é absurdamente simples. De fato, para este tipo de problema você está melhor servido usando um *Gateway de Tabelas de Dados* (151) para criar os *Objetos de Transferência de Dados* (380) diretamente. Entretanto, isso poderia estragar o exemplo de uma *Fachada Remota* colocada sobre um modelo de domínio.

```
class Álbum...

    public String Título;
    public Artista Artista;
    public IList Faixas {
        get {return ArrayList.ReadOnly(dadosDasFaixas);}
    }
    public void AdicionarFaixa (Faixa arg) {
        dadosDasFaixas.Add(arg);
    }
    public void RemoverFaixa (Faixa arg) {
        dadosDasFaixas.Remove(arg);
    }
    private IList dadosDasFaixas = new ArrayList( );
```

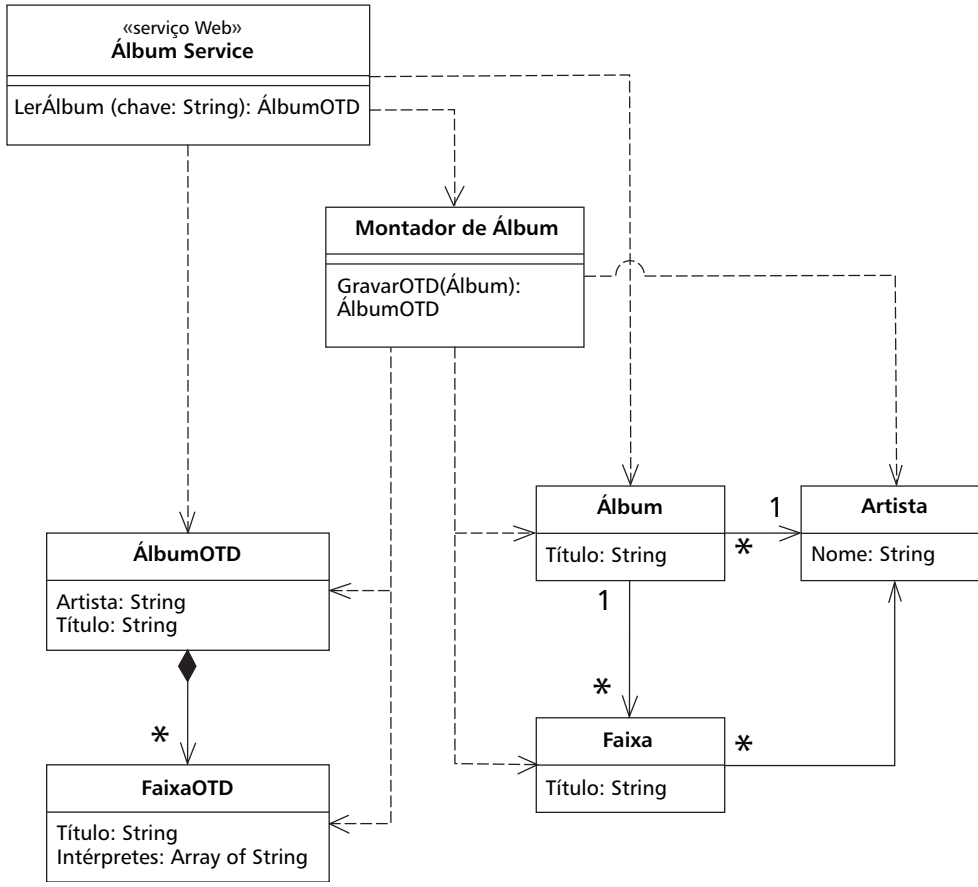


Figura 15.3 Classes do serviço Web para Álbum.


```
class Artista...

    public String Nome;

class Faixa...

    public String Título;
    public IList Intérpretes {
        get { return ArrayList.ReadOnly(dadosDosIntérpretes);}
    }
    public void AdicionarIntérprete (Artista arg) {
        dadosDosIntérpretes.Add(arg);
    }
    public void RemoverIntérprete (Artista arg) {
        dadosDosIntérpretes.Remove(arg);
    }
    private IListа dadosDosIntérpretes = new ArrayList( );
```

Uso *Objetos de Transferência de Dados* (380) para passar os dados através da conexão. Estes são apenas armazenadores de dados que aplainam a estrutura para os propósitos do serviço Web.

```
class ÁlbumOTD...

    public String Título;
    public String Artista;
    public FaixaOTD[ ] Faixas;

class FaixaOTD...

    public String Título;
    public String[ ] Intérpretes;
```

Já que isso é .NET, não preciso escrever nenhum código para serializar e restaurar para XML. O *framework* .NET vem com a classe serializadora apropriada para fazer o trabalho.

Este é um serviço Web, então preciso também declarar a estrutura dos *Objetos de Transferência de Dados* (380) em WSDL. As ferramentas do Visual Studio gerarão o WSDL para mim, e sou um cara do tipo preguiçoso, então deixarei que ele faça isso. Aqui está a definição do Esquema XML que corresponde aos *Objetos de Transferência de Dados* (380):

```
<s:complexType name = "ÁlbumOTD">
    <s:sequence>
        <s:element minOccurs = "1" maxOccurs = "1" name = "Título" nillable = "true" type = "s:string" />
        <s:element minOccurs = "1" maxOccurs = "1" name = "Artista" nillable = "true" type = "s:string" />
         <s:element minOccurs = "1" maxOccurs = "1" name = "Faixas"
              nillable = "true" type = "s0:"ArrayDeFaixasOTD" />
    <s/:sequence>
</s:complexType>
<s:complexType name = "ArrayDeFaixasOTD">
    <s:sequence>
        <s:element minOccurs = "0" maxOccurs = "unbounded" name = "FaixaOTD"
              nillable = "true" type = "s0:FaixaOTD" />
    <s/:sequence>
</s:complexType>
<s:complexType name = "FaixaOTD">
    <s:sequence>
        <s:element minOccurs = "1" maxOccurs = "1" name = "Título" nillable = "true" type = "s:string" />
        <s:element minOccurs = "1" maxOccurs = "1" name = "Intérpretes"
              nillable = "true" type = "s0:ArrayDeStrings" />
    <s/:sequence>
</s:complexType>
<s:complexType name = "ArrayDeStrings">
    <s:sequence>
        <s:element minOccurs = "0" maxOccurs = "unbounded" name = "string"
              nillable = "true" type = "s:string" />
    <s/:sequence>
</s:complexType>
```

Por se tratar de XML, é uma definição de estrutura de dados particularmente prolixa, mas ela executa o trabalho.

Para obter os dados do *Modelo do Domínio* (126) para o *Objeto de Transferência de Dados* (380), preciso de um montador.

```
class MontadorDeÁlbum...

    public ÁlbumOTD GravarOTD (Álbum sujeito) {
        ÁlbumOTD resultado = new ÁlbumOTD( );
        resultado.Artista = sujeito.Artista.Nome;
        resultado.Título = sujeito.Título;
        ArrayList listaDeFaixas = new ArrayList( );
        foreach ( Faixa f in sujeito.Faixas)
            listaDefaixas.Add (GravarFaixa(f));
        resultado.Faixas = (FaixaOTD[ ]) listaDeFaixas.ToArray(typeof(FaixaOTD));
        return resultado;
    }
    public FaixaOTD GravarFaixa (Faixa sujeito) {
        FaixaOTD resultado = new FaixaOTD( );
        resultado.Título = sujeito.Título;
        resultado.Intérpretes = new String[sujeito.Intérpretes.Count];
        ArrayList listaDeIntérpretes = new ArrayList( );
        foreach (Artista a in sujeito.Intérpretes)
            listaDeIntérpretes.Add (a.Nome);
        resultado.Intérpretes = (String [ ]) listaDeIntérpretes.ToArray(typeof (String));
        return resultado;
    }
```

A última parte de que precisamos é a própria definição do serviço. Esta vem primeiro da classe C#.

```
class ÁlbumService...

    [ MétodoWeb ]
    public ÁlbumOTD LerÁlbum (String chave) {
        Álbum resultado == new BuscadorDeÁlbum( ) [chave];
        if (resultado == null)
            throw new SoapException ("não foi possível encontrar um álbum com chave: " +
                        chave, SoapException.ClientFaultCode);
        else return new MontadorDeÁlbum( ).GravarOTD(resultado);
    }
```

É claro que esta não é a definição da interface real – isso vem do arquivo WSDL. Aqui estão os pedaços relevantes:

```
<porType name = "ÁlbumServiceSoap">
    <operation name = "LerÁlbum">
        <input message = "s0:LerÁlbumSoapIn" />
        <output message = "s0:LerÁlbumSoapOut" />
    </operation>
</portType>
<message name = "LerÁlbumSoapIn">
    <part name = "parameters" element = "s0:LerÁlbum" />
</message>
<message name = "LerÁlbumSoapOut">
    <part name = "parameters" element = "s0:LerRespostaÁlbum" />
```

```
</message>
<s:element name = "LerÁlbum: >
    <s:complexType>
        <s:sequence>
            <s:element minOccurs = "1" maxOccurs="1" name = "key" nillable = "true" type = "s:string" />
        </s:sequence>
    </s:complexType>
</s:element>
<s:element name = "LerRespostaÁlbum: >
    <s:complexType>
        <s:sequence>
            <s:element minOccurs = "1" maxOccurs="1" name = "lerResultadoÁlbum"
                  nillable = "true" type = "s0:ÁlbumOTD" />
        </s:sequence>
    </s:complexType>
</s:element>
```

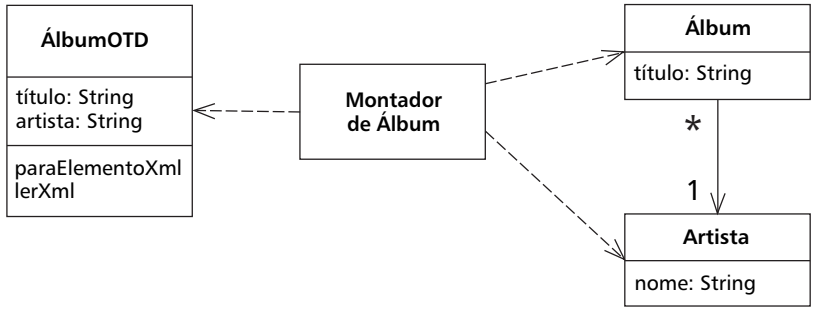
Como esperado, o WSDL é mais tagarela do que a maioria dos políticos, mas ao contrário de tantos deles, consegue executar seu trabalho. Posso agora chamar o serviço enviando uma mensagem SOAP do formato

```
<?xml version = "1.0" encoding = "utf-9"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xmlns:xsd="http://www.w3.org/2001/XMLSchema"
           xmlns:soap= "http://schemas.xmlsoap.org/soap/envelope/">
    <soap:Body>
        <LerÁlbum xmlns= "http://martinfowler.com">
            <key>umaStringChave</key>
        </LerÁlbum>
    </soap:Body>
</soap:Envelope>
```

O importante a ser lembrado a respeito desse exemplo não são as particularidades do SOAP e .NET, mas a abordagem fundamental de camadas. Projete uma aplicação sem distribuição e então coloque a capacidade de distribuição sobre ela com *Fachadas Remotas* e *Objetos de Transferência de Dados* (380).

Objeto de Transferência de Dados (Data Transfer Object)

Um objeto que transporta dados entre processos para reduzir o número de chamadas de métodos.



Quando você está trabalhando com uma interface remota, como uma *Fachada Remota* (368), cada chamada a ela é custosa. O resultado é que você precisa reduzir o número de chamadas, o que significa que você precisa transferir mais dados em cada chamada. Uma maneira de fazer isso é usar muitos parâmetros. Todavia, muitas vezes é complicado de programar – de fato, é freqüentemente impossível com linguagens como Java que retornam apenas um único valor.

A solução é criar um *Objeto de Transferência de Dados* que possa armazenar todos os dados da chamada. Ele precisa ser serializável para passar pela conexão. Normalmente um montador é usado no lado servidor para transferir dados entre o OTD e objetos do domínio.

Muitas pessoas na comunidade Sun usam o termo "Objeto Valor" para este padrão. Eu o uso para outra coisa. Veja a discussão na página 454.

Como Funciona

De muitas formas, um *Objeto de Transferência de Dados* é um daqueles objetos que nossas mães disseram para nunca escrevermos. É freqüentemente pouco mais do que alguns campos e os métodos de gravação e de leitura para eles. O valor coisa detestável é que lhe permite mover diversos pedaços de informação por uma rede em uma única chamada – um artifício que é essencial para sistemas distribuídos.

Sempre que um objeto remoto precisa de alguns dados, pede por um *Objeto de Transferência de Dados* apropriado. O *Objeto de Transferência de Dados* normalmente transporta muito mais dados do que o objeto remoto solicitou, mas deve transportar todos os dados que o objeto remoto precisará por um certo tempo. Devido aos custos de latência das chamadas remotas, é melhor errar enviando dados demais do que ter que fazer diversas chamadas.

Um único *Objeto de Transferência de Dados* normalmente contém mais do que apenas um único objeto do servidor. Ele agrega dados de todos os objetos do servidor que o objeto remoto provavelmente irá querer. Assim, se um objeto remoto solicitar dados sobre um objeto pedido, o *Objeto de Transferência de Dados* retornado conterá dados do pedido, do cliente, dos itens dos pedidos e de seus respectivos produtos e informações sobre a entrega – todos os tipos de coisas.

Você geralmente não pode transferir objetos de um *Modelo de Domínio* (126). Isto porque os objetos estão normalmente conectados em uma rede complexa que é difícil, senão impossível, serializar. Além disso, você normalmente não quer as classes dos objetos do domínio no cliente, o que é igual a copiar o *Modelo de Domínio* (126) inteiro lá. Em vez disso, você tem que transferir uma forma simplificada dos dados dos objetos do domínio.

Os campos em um *Objeto de Transferência de Dados* são razoavelmente simples, sendo classes primitivas simples como *strings* e datas, ou outros *Objetos de Transferência de Dados*. Qualquer estrutura entre objetos de transferência de dados deve ser grafo simples – normalmente uma hierarquia – em oposição às estruturas de grafos mais complexas que você vê em um *Modelo de Domínio* (126). Mantenha estes atributos simples, porque eles têm que ser serializáveis e precisam ser entendidos pelos dois lados da conexão. O resultado disso é que as classes dos *Objetos de Transferência de Dados* e quaisquer classes que elas referenciam devem estar presentes em ambos os lados.

Faz sentido projetar os *Objetos de Transferência de Dados* em torno das necessidades de um cliente em particular. É por isso que você muitas vezes os vê correspondendo a páginas Web ou telas de interface gráfica com o usuário. Você também pode ver diversos *Objetos de Transferência de Dados* para um pedido, dependendo da tela em particular. É claro que, se diferentes apresentações solicitarem dados semelhantes, então faz sentido usar um único *Objeto de Transferência de Dados* para lidar com todos eles.

Uma questão relacionada a considerar é o uso de um único *Objeto de Transferência de Dados* para uma interação completa contra o uso de diferentes objetos desses para cada solicitação. *Objetos de Transferência de Dados* diferentes tornam mais fácil ver quais dados são transferidos em cada chamada, mas isso leva a muitos *Objetos de Transferência de Dados*. Um é menos trabalhoso de escrever, mas torna mais difícil ver como cada chamada transfere informações. Sou inclinado a usar apenas um se houver muitas coisas em comum nos dados, mas não hesito em usar *Objetos de Transferência de Dados* diferentes se uma solicitação em especial sugerir isso. É uma daquelas coisas para as quais você não pode criar uma regra, então eu poderia usar um *Objeto de Transferência de Dados* para a maior parte da interação e usar diferentes *Objetos de Transferência de Dados* para algumas solicitações e respostas.

Uma questão similar é ter um único *Objeto de Transferência de Dados* tanto para a solicitação quanto para a resposta, ou objetos separados para cada uma. Novamente, não há uma regra. Se os dados em cada caso forem bastante semelhantes, uso um. Se forem bastante diferentes, eu uso dois.

Algumas pessoas gostam de tornar *Objetos de Transferência de Dados* imutáveis. Neste esquema, você recebe um *Objeto de Transferência de Dados* do cliente, cria e envia de volta um diferente, mesmo se for da mesma classe. Outras pessoas alteram o *Objeto de Transferência de Dados* da solicitação. Não tenho uma opinião decidida sobre estas maneiras, mas de modo geral prefiro um *Objeto de Transferência de Dados* mutável porque é mais fácil inserir os dados gradualmente, mesmo se você criar um novo objeto para a resposta. Alguns argumentos a favor de *Objetos de Transferência de Dados* imutáveis têm a ver com a confusão de nomes com *Objeto Valor* (453).

Uma forma comum de *Objeto de Transferência de Dados* é aquela de um *Conjunto de Registros* (473), ou seja, um conjunto de registros tabulares – exatamente o que você recebe de volta de uma consulta SQL. De fato, um *Conjunto de Registros* (473) é o *Objeto de Transferência de Dados* para um banco de dados SQL. Arquiteturas fre-

qüentemente o usam durante o projeto. Um modelo de domínio pode gerar um *Conjunto de Registros* (473) de dados para transferir para um cliente, o qual o cliente trata como se estivesse vindo diretamente do SQL. Isto é útil se o cliente tiver ferramentas amarradas a estruturas de *Conjunto de Registros* (473). O *Conjunto de Registros* (473) pode ser criado inteiramente pela lógica do domínio, mas mais provavelmente é gerado a partir de uma consulta SQL e modificado pela lógica do domínio antes que seja passado para a apresentação. Esta maneira presta-se para *Módulo Tabela* (134).

Outra forma de *Objeto de Transferência de Dados* é uma estrutura de dados do tipo coleção genérica. Tenho visto *arrays* usados para isso, mas desaconselho porque os índices do *array* deixam o código mais obscuro. A melhor coleção é um dicionário, porque você pode usar *strings* significativas como chaves. O problema é que você perde a vantagem de uma interface explícita e tipificação forte. Pode valer a pena usar um dicionário para casos *ad hoc*, quando você não tem um gerador à mão, pois é mais fácil manipular um do que escrever um objeto explícito manualmente. Entretanto, com um gerador, penso que você fica melhor com uma interface explícita, especialmente quando considera que ela está sendo usada como protocolo de comunicação entre componentes diferentes.

Serializando o Objeto de Transferência de Dados Além dos métodos de gravação e leitura, o *Objeto de Transferência de Dados* normalmente também é responsável por serializar a si mesmo em algum formato próprio para transmissão pela conexão. Qual formato depende do que está em cada lado da conexão, o que pode passar pela própria conexão e o nível de facilidade da serialização. Várias plataformas fornecem serialização embutida para objetos simples. Por exemplo, Java tem um serialização binária embutida, e .NET tem serializações embutidas binária e XML. Se houver serialização embutida, ela normalmente funciona direito porque *Objetos de Transferência de Dados* são estruturas simples que não lidam com as complexidades com as quais você se depara com objetos em um modelo de domínio. O resultado disso é que sempre uso o mecanismo automático se puder.

Se não tiver um mecanismo automático, geralmente pode criar você mesmo um. Tenho visto diversos geradores de código que recebem simples descrições de registros e geram classes apropriadas para armazenar os dados, fornecer métodos de acesso e ler e gravar as serializações de dados. O importante é tornar o gerador não mais complicado do que você precisa que ele seja, e não tentar colocar características que você apenas acredita que precisará. Pode ser uma boa idéia escrever as primeiras classes à mão e então usá-las para lhe ajudar a escrever o gerador.

Você também pode usar programação reflexiva para lidar com serialização. Dessa maneira, você só tem que escrever as rotinas de serialização e desserialização uma vez e colocá-las em uma superclasse. Pode haver um custo de desempenho nisso. Você terá que medi-lo para descobrir se esse custo é significativo.

Você tem que escolher um mecanismo com o qual os dois lados da conexão trabalharão. Se controlar os dois lados, pegue o mais fácil, caso contrário, pode conseguir fornecer um conector no lado que não é seu. Você pode então usar um *Objeto de Transferência de Dados* simples em ambos os lados da conexão e usar o conector para adaptar o componente externo.

Uma das questões mais comuns com que você se depara com *Objetos de Transferência de Dados* é usar uma forma de serialização em modo texto ou binário. As serializações em modo texto são fáceis de ler para descobrir o que está sendo comunica-

do. A XML é popular porque você pode obter ferramentas facilmente para criar e analisar documentos XML. As grandes desvantagens com o modo texto é que ele precisa de mais largura de banda para enviar os mesmos dados (algo especialmente verdadeiro sobre XML) e muitas vezes há prejuízo de desempenho, o que pode ser bastante significativo.

Um fator importante para serialização é a sincronização do *Objeto de Transferência de Dados* em cada lado da conexão. Na teoria, sempre que o servidor alterar a definição do *Objeto de Transferência de Dados*, o cliente atualiza também, mas na prática isso pode não acontecer. Acessar um servidor com um cliente desatualizado sempre leva a problemas, mas o mecanismo de serialização pode tornar os problemas mais ou menos penosos. Com uma serialização puramente binária de um *Objeto de Transferência de Dados*, o resultado será que sua comunicação é inteiramente perdida, já que qualquer alteração na sua estrutura normalmente causa um erro na desserialização. Mesmo uma alteração inofensiva, como a adição de um campo opcional, terá esse efeito. O resultado é que a serialização binária direta pode introduzir muita fragilidade às linhas de comunicação.

Outros esquemas de serialização podem evitar isso. Um é a serialização XML, que pode normalmente ser escrita de uma maneira que torne as classes mais tolerantes a alterações. Outro é uma abordagem binária mais tolerante, como serializar os dados usando um dicionário. Embora eu não goste de usar um dicionário como *Objeto de Transferência de Dados*, pode ser uma maneira útil de executar uma serialização binária dos dados, já que isso introduz alguma tolerância na sincronização.

Montando um *Objeto de Transferência de Dados* a partir de Objetos de Domínio

Um *Objeto de Transferência de Dados* não sabe como se conectar a objetos do domínio, porque ele deve ser distribuído nos dois lados da conexão. Por este motivo, não quero que o *Objeto de Transferência de Dados* seja dependente do objeto do domínio. Também não quero que os objetos do domínio sejam dependentes do *Objeto de Transferência de Dados*, já que a estrutura do *Objeto de Transferência de Dados* mudará quando eu alterar os formatos da interface. Como regra, quero manter o modelo do domínio independente das interfaces externas.

O resultado disso é que gosto de criar um objeto montador separado responsável pela criação de um *Objeto de Transferência de Dados* a partir do modelo do domínio e pela atualização do modelo a partir dele (Figura 15.4). O montador é um exemplo de um *Mapeador* (442), visto que ele mapeia entre o *Objeto de Transferência de Dados* e os objetos do domínio.

Também posso fazer diversos montadores compartilhar o mesmo *Objeto de Transferência de Dados*. Um caso comum para isso são diferentes semânticas de atualização em cenários diferentes usando os mesmos dados. Outra razão para separar o montador é que o *Objeto de Transferência de Dados* pode facilmente ser gerado automaticamente a partir de uma descrição de dados simples. Gerar o montador é mais difícil e, muitas vezes, impossível.

Quando Usá-lo

Use um *Objeto de Transferência de Dados* sempre que precisar transferir diversos itens de dados entre dois processos em uma única chamada de método.

Há algumas alternativas a *Objeto de Transferência de Dados*, embora eu não seja fã delas. Uma é não usar um objeto, mas simplesmente um método de gravação

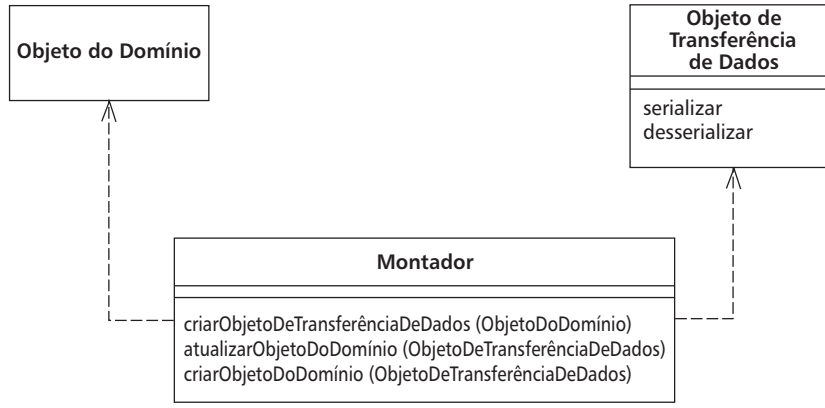


Figura 15.4 Um objeto montador pode manter o modelo do domínio e os objetos de transferência de dados independentes um do outro.

com muitos parâmetros ou um método de leitura com diversos parâmetros passados por referência. O problema é que muitas linguagens, como Java, permitem apenas o retorno de objetos, então, embora isso possa ser usado para atualizações, não pode ser usado para trazer informações sem correr riscos com chamadas de retorno.

Uma alternativa é usar diretamente algum tipo de representação de *string*, sem um objeto atuando como interface para isso. Aqui o problema é que tudo o mais fica acoplado à representação da *string*. É bom esconder a representação exata por trás de uma interface explícita. Dessa forma, se você quiser alterar a *string* ou substituí-la por uma estrutura binária, não tem que alterar mais nada.

Em particular, vale a pena criar um *Objeto de Transferência de Dados* quando você quiser comunicar entre componentes usando XML. O DOM XML é difícil de manipular e é muito melhor usar um *Objeto de Transferência de Dados* que o encapsule, especialmente pelo *Objeto de Transferência de Dados* ser tão fácil de gerar.

Outro propósito comum do *Objeto de Transferência de Dados* é atuar como uma fonte comum de dados para diversos componentes em camadas diferentes. Cada componente faz algumas alterações no *Objeto de Transferência de Dados* e então o passa para a próxima camada. O uso de *Conjunto de Registros* (473) em COM e .NET é um bom exemplo disso, em que cada camada sabe como manipular dados baseados em conjuntos de registros, quer tenham vindo diretamente de um banco de dados SQL, quer tenham sido modificados por outras camadas. .NET expande isso fornecendo um mecanismo embutido para serializar conjuntos de registros para XML.

Embora este livro enfoque sistemas síncronos, há um uso assíncrono interessante para *Objeto de Transferência de Dados*. Este é quando você quiser usar uma interface tanto síncrona quanto assincronamente. Retorne um *Objeto de Transferência de Dados* como de costume para o caso síncrono. Para o caso assíncrono, crie uma *Carga Tardia* (200) do *Objeto de Transferência de Dados* e a retorne. Conecte a *Carga Tardia* (200) onde quer que os resultados da chamada assíncrona devam aparecer. O usuário do *Objeto de Transferência de Dados* bloqueará apenas quando este tentar acessar os resultados da chamada.

Leitura Adicional

[Alur *et al.*] discutem este padrão sob o nome de *Objeto Valor*, o que eu disse anteriormente ser equivalente ao meu *Objeto de Transferência de Dados*. Meu *Objeto Valor* (453) é um padrão inteiramente diferente. Esta é uma colisão de nomes. Muitas pessoas têm usado "Objeto Valor" no sentido em que eu o uso. Até onde eu saiba, seu uso significando o que eu chamo de *Objeto de Transferência de Dados* ocorre apenas dentro da comunidade J2EE. Como resultado, segui o uso mais geral.

O *Montador de Objeto Valor* [Alur *et al.*] é uma discussão sobre o montador. Decidi não torná-lo um padrão separado, embora use o nome "montador" em vez de um nome baseado no *Mapeador* (442).

[Marinescu] discute *Objeto de Transferência de Dados* e diversas variantes de implementações. [Riehle *et al.*] discutem maneiras flexíveis de serializar, incluindo a alternância entre diferentes formas de serialização.

Exemplo: Transferindo Informações Sobre Álbuns (Java)

Para este exemplo, usarei o modelo do domínio da Figura 15.5. Os dados que quero transferir são os dados sobre estes objetos vinculados, e a estrutura para os objetos de transferência de dados é a da Figura 15.6.

Os objetos de transferência de dados simplificam bastante esta estrutura. Os dados relevantes da classe Artista são movidos para o Álbum OTD, e os intérpretes de uma faixa são representados como um *array* de *strings*. Isso é típico da compactação de estruturas que você vê para um objeto de transferência de dados. Há dois objetos de transferência de dados presentes, um para o álbum e um para cada faixa. Neste caso, não preciso de um para o artista, já que todos os dados estão presentes em um dos outros dois. Só tenho a Faixa como objeto de transferência porque há diversas faixas no álbum e cada uma contém mais do que um item de dados.

Aqui está o código para gravar um *Objeto de Transferência de Dados* do modelo do domínio. O montador é chamado por qualquer objeto que esteja lidando com a interface remota, como uma *Fachada Remota* (368).

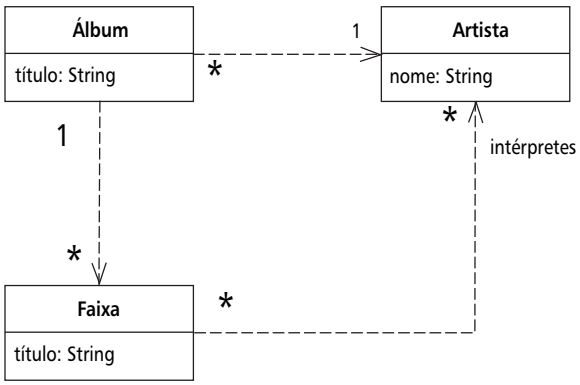


Figura 15.5 Um diagrama de classes de artistas e álbuns.

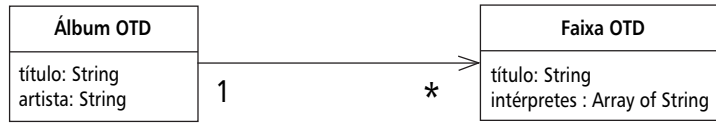


Figura 15.6 Um diagrama de classes de artistas e álbuns.

```
class MontadorDeÁlbum...

    public ÁlbumOTD gravarOTD (Álbum sujeito) {
        ÁlbumOTD resultado = new ÁlbumOTD( );
        resultado.gravarTítulo(sujeito.lerTítulo( ));
        resultado.gravarArtista(sujeito.lerArtista( ).lerNome());
        gravarFaixas(resultado, sujeito);
        return resultado;
    }
    private void gravarFaixas (ÁlbumOTD resultado, Álbum sujeito) {
        List novasFaixas = new ArrayList ( );
        Iterator it = sujeito.lerFaixas( ).iterator( );
        while (it.hasNext( )) {
            FaixaOTD novoOTD = new FaixaOTD( );
            Faixa estaFaixa = (Faixa) it.next( );
            novoOTD.gravarTítulo (estaFaixa.lerTítulo( ));
            gravarIntérpretes(novoOTD, estaFaixa);
            novasFaixas.add(novoOTD);
        }
        resultado.gravarFaixas( (FaixaOTD[ ]) novasFaixas.toArray(new FaixaOTD[0]));
    }
    private void gravarIntérpretes (FaixaOTD otd, Faixa sujeito) {
        List resultado = new ArrayList( );
        Iterator it = sujeito.lerIntérpretes( ).iterator( );
        while (it.hasNext( )) {
            Artista cada = (Artista) it.next( );
            resultado.add (cada.lerNome( ));
        }
        otd.gravarIntérpretes ( (String[ ]) resultado.toArray(new String[0]));
    }
```


Atualizar o modelo a partir do *Objeto de Transferência de Dados* é normalmente mais confuso. Para este exemplo há uma diferença entre criar um novo álbum e atualizar um existente. Aqui está o código para criação:

```
class MontadorDeÁlbum...

    public void criarÁlbum (String id, ÁlbumOTD fonte) {
        Artista artista = Registry. encontrarArtistaChamado (fonte.lerArtista( ));
        if (artista == null)
            throw new RuntimeException ("Nenhum artista chamado " + fonte.lerArtista( ));
        Álbum álbum = new Álbum (fonte.lerTítulo( ), artista);
        criarFaixas(fonte.lerFaixas( ), álbum);
        Registro.adicionarÁlbum(id, álbum);
    }
    private void criarFaixas (FaixaOTD[ ] faixas, Álbum álbum) {
```

```
        for (int i = 0; i< faixas.length; i++) {
            Faixa novaFaixa = new Faixa(faixas[i].lerTítulo( ));
            álbum.adicionarFaixa(novaFaixa);
            criarIntérpretes(novaFaixa, faoxas[i].lerIntérpretes( ));
        }
    }
    private void criarIntérpretes (Faixa novaFaixa, String[ ] arrayDeIntérpretes) {
        for (int i = 0; i < arrayDeIntérpretes.length; i++) {
            Artista intérprete = Registro. encontrarArtistaChamado (arrayDeIntérpretes [i]) {
            if (intérprete == null)
                throw new RuntimeException ("Nenhum artista chamado " + arrayDeIntérpretes [i]);
            novaFaixa.adicionarIntérprete(intérprete);
        }
    }
```

Ler o OTD envolve algumas decisões. Perceptível aqui é como lidar com os nomes dos artistas à medida que eles chegam. Meus requisitos são que os artistas já devem estar em um *Registro* (448) quando eu crio o álbum de modo que, se eu não conseguir encontrar um artista, isso é um erro. Um método de criação diferente poderia decidir criar artistas quando eles forem mencionados no *Objeto de Transferência de Dados*.

Para este exemplo tenho um método diferente para atualizar um álbum existente.

```
class MontadorDeÁlbum...

    public void atualizarÁlbum (String id, ÁlbumOTD fonte) {
        Álbum atual = Registro. encontrarÁlbum (id);
        if (atual == null)
            throw new RuntimeException ("Álbum não existe: " + fonte.lerTítulo( ));
        if (fonte.lerTítulo( ) != atual.lerTítulo( )) atual.gravarTítulo(fonte.lerTítulo( ));
        if (fonte.lerArtista( ) != atual.lerArtista( ).lerNome( )) {
            Artista artista = Registro. encontrarArtistaChamado (fonte.lerArtista( ));
            if (artista == null)
                throw new RuntimeException ("Nã há artista chamado " + fonte.lerArtista( ));
            atual.gravarArtista(artista);
        }
        atualizarFaixas(fonte, atual);
    }
    private void atualizarFaixas (ÁlbumOTD fonte, Álbum atual) {
        for (int i = 0; i < fonte.lerFaixas( ).length; i++) {
            atual.lerFaixa(i).gravarTítulo(fonte.lerFaixaOTD(i).lerTítulo( ));
            atual.lerFaixa(i). apagarIntérpretes ( );
            criarIntérpretes(atual.lerFaixa(i), fonte.lerFaixaOTD(i).lerIntérpretes( ));
        }
    }
```

Quanto a atualizações, você pode decidir atualizar o objeto do domínio existente ou destruí-lo e substituí-lo por um novo. A questão aqui é se você tem outros objetos se referindo ao objeto que quer atualizar. Neste código, estou atualizando o álbum já que tenho outros objetos se referindo a ele e a suas faixas. Entretanto, para o título e os intérpretes de uma faixa, apenas substituo os objetos que estão lá.

Outra questão diz respeito à alteração de artistas. Isso está alterando o nome do artista existente ou mudando o artista ao qual o álbum está vinculado? Novamente,

estas questões têm que ser decididas caso a caso, e estou lidando com ela vinculando-a a um novo artista.

Neste exemplo, usei uma serialização binária nativa, o que significa que tenho que tomar cuidado para que as classes dos *Objetos de Transferência de Dados* em ambos os lados da conexão sejam mantidas em sincronia. Se eu fizer uma alteração na estrutura de dados do *Objeto de Transferência de Dados* servidor e não alterar o cliente, terei erros na transferência. Posso tornar a transferência mais tolerante usando um mapa como minha serialização.

```
class FaixaOTD...

    public Map Fpa( ) {
        Map resultado = new HashMap( );
        resultado.put ("título", título);
        resultado.put("intérpretes", intérpretes);
        return resultado;
    }
    public static FaixaOTD lerMapa (Mapa arg) {
        FaixaOTD resultado = new Faixa OTD( );
        resultado.título = (String) arg.get("título");
        resultado.intérpretes = (String [ ]) arg.get("intérpretes");
        return resultado;
    }
```

Agora, se eu adicionar um campo ao servidor e usar o cliente antigo, embora o novo campo não vá ser pego pelo cliente, o resto dos dados serão transferidos corretamente.

É claro que escrever as rotinas de serialização e desserialização desta forma é tedioso. Posso evitar muito desse tédio usando uma rotina reflexiva como esta na *Camada Supertipo* (444):


```
class ObjetoDeTransferênciaDeDados...

    public Map gravarMapaReflexão ( ) {
        Map resultado = null;
        try {
            Field [ ] campos = this.getClass( ).getDeclaredFields( );
            resultado = new HashMap( );
            for (int i = 0; i< campos.length; i++)
                resultado.put (campos[i].getName( ), campos[i].get(this));
        } catch (Exception e) {throw new ApplicationException (e);
        }
        return resultado;
    }
    public static FaixaOTD lerMapaComReflxão (Map arg) {
        FaixaOTD resultado = new FaixaOTD( );
        try {
            Field[ ] campos = resultado.getClass( ).getDeclaredFields( );
            for (int i = 0; i< campos.length; i++)
                campos[i].set(resultado, arg.get(campos[i].getName( ));
        } catch (Exception e) {throw new ApplicationException (e);
        }
        return resultado;
    }
```

Tal rotina irá lidar muito bem com a maioria dos casos (embora você vá ter que adicionar código extra para lidar com primitivas).

Exemplo: Serializando Usando XML (Java)

Enquanto escrevo isto, a manipulação de XML por Java está em contínua mudança e APIs, ainda voláteis, estão de modo geral melhorando. Quando você ler, esta seção pode estar desatualizada ou completamente irrelevante, mas o conceito básico da conversão para XML é quase que o mesmo.

Primeiro, leio a estrutura de dados do *Objeto de Transferência de Dados*, então preciso decidir como serializá-la. Em Java, você obtém serialização binária simplesmente usando uma interface de marcador. Isso funciona de maneira completamente automática com um *Objeto de Transferência de Dados*, então, é minha primeira escolha. Entretanto, muitas vezes, é necessário serialização baseada em texto. Para este exemplo, então, usarei XML.

Neste exemplo, estou usando JDOM, já que isso torna o trabalho com XML muito mais fácil do que usar as interfaces padrão W3C. Escrevo métodos para ler e gravar um elemento XML para representar essa classe em cada classe *Objeto de Transferência de Dados*.

```
class ÁlbumOTD...

    Element paraElementoXML ( ) {
        Element raiz = new Element ("álbum");
        raiz.setAttribute ("título", título);
        raiz.setAttribute("artista", artista);
        for (int i = 0; i < faixas.length; i++)
            raiz.addContent(faixas[i].paraElementoXML( ));
        return raiz;
    }
    static ÁlbumOTD lerXML (Element fonte) {
        ÁlbumOTD resultado = new ÁlbumOTD( );
        resultado.gravarTítulo(fonte.getAttributeValue("título");
        resultado.gravarArtista(fonte.getAttributeValue("artista");
        List listaDeFaixas = new ArrayList( );
        Iterator it = fonte.getChildren("faixa").iterator( );
        while (it.hasNext( ))
            listaDeFaixa.add(FaixaOTD.lerXML((Element) it.next( )));
        resultado.gravarFaixas((FaixaOTD[ ] ) listaDeFaixas.toArray(new FaixaOTD[0]));
        return resultado;
    }

class FaixaOTD...

    Element paraElementoXML ( ) {
        Element resultado = new Element ("faixa");
        resultado.setAttribute ("título", título);
        for (int i = 0; i < intérpretes.length; i++) {
            Element elementoIntérprete = new Element ("intérprete");
            elementoIntérprete.setAttribute("nome", intérpretes[i]);
            resultado.addContent(elementoIntérprete);
        }
        return resultado;
    }
```

```
    static FaixaOTD lerXml (Element arg) {
        FaixaOTD resultado = new FaixaOTD( );
        resultado.gravarTítulo(arg.getAttributeValue("título");
        Iterator it = arg.getChildren("intérprete").iterator( );
        List buffer = new ArrayList( );
        while (it.hasNext( )) {
            Element cadaElemento = (Element) it.next( );
            buffer.add(cadaElemento.getAttributeValue("nome");
        }
        resultado.gravarIntérpretes( (String[ ]) buffer.toArray(new String[0]));
        return resultado;
    }
```

É claro que estes métodos apenas criam os elementos no DOM XML. Para executar a serialização, preciso ler e escrever texto. Já que a faixa é transferida apenas no contexto do álbum, preciso apenas gravar este código do álbum.

```
class Álbum OTD...

    public void paraStringXML (Writer saída) {
        Element raiz = paraElementoXml( );
        Document doc = new Document (raiz);
        XMLOutputter gravador = new XMLOutputter( );
        try {
            gravador.output(doc, saída);
        } catch (IOException e) {
            e.printStackTrace( );
        }
    }
    public static ÁlbumOTD lerStringXml (Reader entrada) {
        try {
            SAXBuilder construtor = new SAXBuilder( );
            Document doc = construtor.build(entrada);
            Element raiz = doc.getRootElement( );
            ÁlbumOTD resultado = lerXML(raiz);
                return resultado;
        } catch (Exception e) {
            e.printStackTrace( );
            thow new RuntimeException ( );
        }
    }
```


Embora este não seja um conhecimento sofisticado, ficarei feliz quando JAXB tornar este tipo de coisa desnecessário.

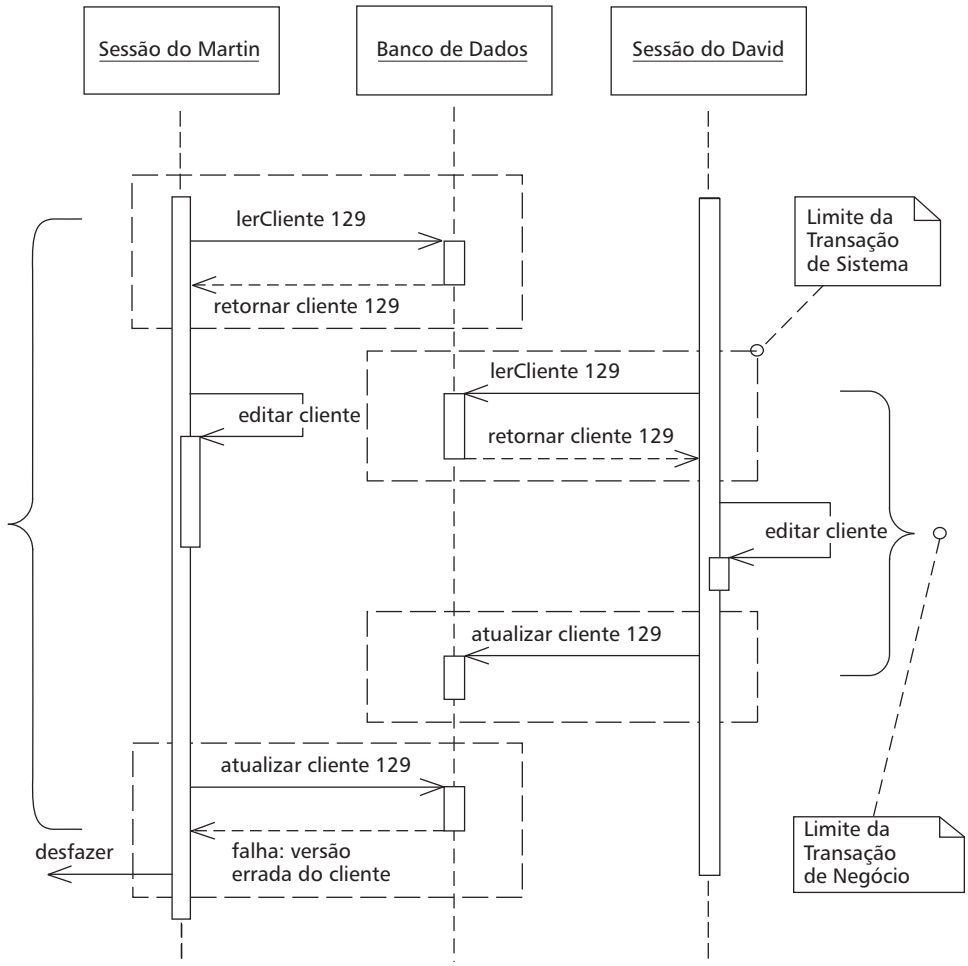
CAPÍTULO 16

Padrões de Concorrência *Offline*

Bloqueio *Offline* Otimista (Optimistic Offline Lock)

por David Rice

Previne conflitos entre transações de negócio concorrentes detectando um conflito e desfazendo a transação.



Muitas vezes, uma transação de negócio é executada por meio de uma série de transações de sistema. Uma vez fora de uma transação de sistema, não podemos contar apenas com nosso gerenciador de banco de dados para assegurar que a transação de negócio deixará os dados em um estado consistente. A integridade dos dados é um risco, assim que duas sessões começam a trabalhar nos mesmos registros, e atualizações perdidas são bastante possíveis. Além disso, com uma sessão editando dados que outra esteja lendo, uma leitura inconsistente torna-se provável.

O *Bloqueio Offline Otimista* resolve esse problema ratificando que as alterações a serem gravadas por uma sessão não estejam em conflito com as alterações de ou-

tra sessão. Uma validação com sucesso antes da confirmação (*commit*) significa, de certo modo, obter um bloqueio indicando que não há problema em ir em frente com as alterações nos dados dos registros. Enquanto a validação e as atualizações ocorrerem dentro de uma única transação de sistema, a transação de negócio exibirá consistência.

Enquanto que o *Bloqueio Offline Pessimista* (401) pressupõe que a chance de conflito na sessão é alta e, portanto, limita a concorrência do sistema, o *Bloqueio Offline Otimista* supõe que a chance de conflito é baixa. A expectativa de que um conflito na sessão não seja provável permite que diversos usuários trabalhem com os mesmos dados ao mesmo tempo.

Como Funciona

Um *Bloqueio Offline Otimista* é obtido garantindo que, no tempo decorrido desde, que uma sessão carregou um registro, outra sessão não o alterou. Ele pode ser obtido a qualquer momento, mas é válido apenas durante a transação de sistema na qual foi obtido. Assim, para que uma transação de negócio não corrompa dados gravados ela deve obter um *Bloqueio Offline Otimista* para cada membro de seu conjunto de alterações, durante a transação de sistema na qual ela aplica as alterações no banco de dados.

A implementação mais comum é associar um número de versão com cada registro no seu sistema. Quando um registro é carregado, esse número é mantido pela sessão junto com todo o estado dela. Obter o *Bloqueio Offline Otimista* é uma questão de comparar a versão armazenada nos dados da sua sessão com a versão atual nos dados gravados. Assim que a verificação tenha sucesso, todas as alterações, incluindo o incremento da versão, podem ser gravadas. O incremento da versão é o que evita dados gravados inconsistentemente, já que a sessão com uma versão antiga não pode obter o bloqueio.

Com dados em um SGBDR a verificação é uma questão de adicionar o número da versão ao critério de qualquer declaração SQL usada para atualizar ou apagar um registro. Uma única declaração SQL pode tanto obter o bloqueio quanto atualizar os dados gravados. O passo final é a transação de negócio inspecionar o contador de linha retornado pela execução do SQL. Um contador de linha igual a um indica sucesso, zero indica que o registro foi alterado ou excluído. Com um contador de linha igual a zero, a transação de negócio deve desfazer a transação de sistema para evitar que alterações sejam feitas nos dados gravados. Neste ponto, a transação de negócio deve abortar ou tentar resolver o conflito e tentar de novo.

Além de um número de versão para cada registro, armazenar informações sobre quem modificou esse registro da última vez e quando isso foi feito pode ser muito útil durante o gerenciamento de conflitos de concorrência. Ao informar um usuário a respeito de uma falha de atualização devido a violação de concorrência, uma aplicação correta dirá quando o registro foi alterado e por quem. É uma idéia ruim usar o *timestamp* da modificação em vez de um contador de versão para suas verificações otimistas, porque os *clocks* do sistema são simplesmente não-confiáveis, especialmente se você estiver coordenando por meio de diversos servidores.

Em uma implementação alternativa, a cláusula *WHERE* na atualização inclui cada campo da linha. A vantagem aqui é que você pode usar essa cláusula sem usar algum tipo de campo versão, o que pode ser útil se você não puder adicionar um campo versão alterando as tabelas do banco de dados. O problema é que isso com-

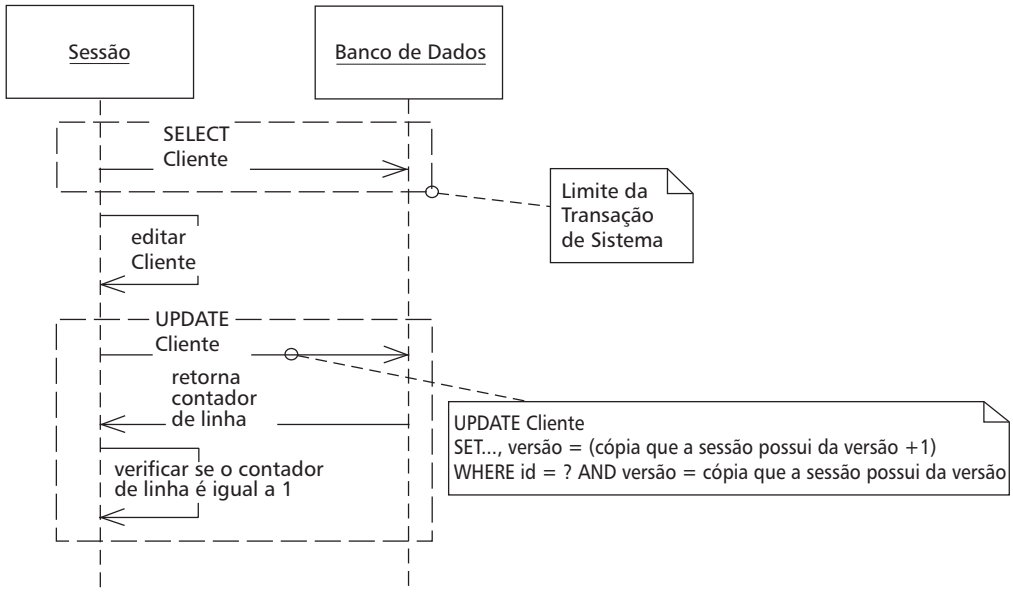


Figura 16.1 Verificação otimista de UPDATE.

plica a declaração UPDATE com uma cláusula *WHERE* potencialmente grande, o que pode também ter um impacto no desempenho dependendo do quão hábil o banco de dados é em relação ao uso do índice da chave primária.

Muitas vezes, a implementação do *Bloqueio Offline Otimista* é feita por meio da inclusão da versão nas declarações UPDATE e DELETE, porém isso não resolve o problema de uma leitura inconsistente. Pense em um sistema de faturas que crie uma cobrança e calcule a taxa de venda apropriada. Uma sessão cria a cobrança e então procura pelo endereço do cliente para calcular a taxa, mas durante a sessão de geração da cobrança uma sessão separada de manutenção de clientes edita o endereço do cliente. Como a taxa depende da localização, o valor calculado pela sessão de geração da cobrança poderia ser inválido, mas já que a sessão de geração da cobrança não fez nenhuma alteração no endereço, o conflito não será detectado.

Não há motivo para que o *Bloqueio Offline Otimista* não possa ser usado para detectar uma leitura inconsistente. No exemplo acima, a sessão de geração da cobrança precisa reconhecer que sua correção depende do valor do endereço do cliente. Ela deve, portanto, executar também uma verificação de versão no endereço, talvez adicionando o endereço ao conjunto de alterações ou mantendo uma lista separada de itens a terem sua versão checada. Esta última opção requer um pouco mais de trabalho para ser configurada, mas resulta em código que declara suas intenções mais claramente. Se você estiver verificando uma consistência de leitura simplesmente relendo a versão em vez de uma alteração artificial, esteja ciente do nível de isolamento da sua transação de sistema. A releitura da versão só funcionará com leituras repetíveis ou isolamento maior. Qualquer coisa mais fraca do que isso requer um incremento da versão.

Uma verificação de versão poderia ser demais para determinados problemas de leitura inconsistente. Muitas vezes, uma transação depende apenas da presença de um registro ou talvez do valor de apenas um de seus campos. Neste caso, você pode-

ria melhorar a esperteza de seu sistema verificando condições em vez de versão, já que menos atualizações concorrentes resultarão na falha das transações de negócio que estão competindo. Quanto mais você entender os problemas de concorrência, melhor poderá gerenciá-los no seu código.

O *Bloqueio de Granularidade Alta* (412) pode ajudar com algumas leituras inconsistentes tratando um grupo de objetos como um único item bloqueável. Outra opção é simplesmente executar todos os passos das transações de negócio problemáticas dentro de uma única transação longa. A facilidade de implementação poderia provar que o benefício de usar algumas transações longas em algumas ocasiões compensa o custo.

A detecção de uma leitura inconsistente fica um pouco difícil quando sua transação é dependente dos resultados de uma pesquisa dinâmica em vez da leitura de registros específicos. É possível que você tenha que gravar os resultados iniciais e compará-los aos resultados da mesma pesquisa na hora da confirmação como um meio de obter um *Bloqueio Offline Otimista*.

Assim como com todos os esquemas de bloqueio, o *Bloqueio Offline Otimista* por si só não fornece soluções adequadas para alguns dos problemas mais complicados de concorrência e alguns problemas temporais em uma aplicação de negócio. Nunca é demais enfatizar que em uma aplicação de negócio o gerenciamento de concorrência é uma questão tanto de domínio quanto técnica. O cenário do endereço de cliente acima é realmente um conflito? Poderia ser admissível que eu tivesse calculado a taxa com uma versão mais antiga do cliente, mas qual versão eu deveria realmente estar usando? Esta é uma questão de negócio. Ou então considere uma coleção. Se duas sessões adicionassem itens simultaneamente em uma coleção? O esquema de *Bloqueio Offline Otimista* típico não evitaria isso embora pudesse muito bem ser uma violação de regras de negócio.

Há um sistema usando *Bloqueio Offline Otimista* com o qual todos nós deveríamos estar familiarizados: gerenciamento de código fonte (GCF). Quando um sistema GCF detecta um conflito entre programadores, normalmente pode descobrir a consolidação correta e tentar gravar novamente. Uma boa estratégia de consolidação torna o *Bloqueio Offline Otimista* bastante poderoso não apenas porque a concorrência do sistema é bastante alta, mas também porque os usuários raramente têm que refazer algum trabalho. É claro que a grande diferença entre um sistema GCF e uma aplicação corporativa de negócio é que o GCF deve implementar apenas um tipo de consolidação, enquanto que o sistema de negócio poderia implementar centenas. Alguns poderiam ser de tal complexidade que não valeria o custo da codificação. Outros poderiam ser de tal valor para o negócio que a consolidação deveria ser codificada de qualquer jeito. Embora sendo feita com pouca freqüência, a consolidação de objetos de negócio é possível. Na verdade, consolidar dados de negócio é em si um padrão. Pararei por aqui em vez de esgotar o assunto, mas compreenda o poder que a consolidação acrescenta ao *Bloqueio Offline Otimista*.

O *Bloqueio Offline Otimista* só nos informa se uma transação de negócio será realmente gravada durante a última transação de sistema. Contudo, ocasionalmente é útil conhecer com maior antecedência se ocorreu um conflito. Para isso, você pode fornecer um método `verificarAtual` que verifica se mais alguém atualizou os dados. Isso não garante que você não vá ter um conflito, mas pode valer a pena parar um processo complicado se você puder perceber de antemão que ele não será gravado. Use este `verificarAtual` sempre que descobrir cedo que uma falha possa ser útil, mas lembre-se de que isso nunca garante que não haverá falha na hora da gravação.

Quando Usá-lo

O gerenciamento otimista de concorrência é apropriado quando a chance de conflito entre duas transações de negócio quaisquer for baixa. Quando os conflitos forem prováveis, não é razoável anunciá-los apenas quando o usuário tiver concluído seu trabalho e estiver pronto para a confirmação . No final, ele irá pressupor a falha das transações de negócio e parará de usar o sistema. O *Bloqueio Offline Pessimista* (401) é mais apropriado quando a chance de conflito for alta ou o custo de um conflito for inaceitável.

Como um bloqueio otimista é muito mais fácil de implementar e não está propenso aos mesmos defeitos e erros em tempo de execução do *Bloqueio Offline Pessimista* (401), considere usá-lo como abordagem padrão de gerenciamento de conflito de transações de negócio em qualquer sistema que você construir. A versão pessimista funciona bem como um complemento ao seu correlato otimista, então em vez de perguntar quando usar uma abordagem otimista para evitar conflito, pergunte quando a abordagem otimista sozinha não é boa o suficiente. A abordagem correta para a gerência de concorrência maximizará o acesso concorrente aos dados ao mesmo tempo em que minimiza os conflitos.

Exemplo: Camada de Domínio com *Mapeadores de Dados* (170) (Java)

O exemplo mais curto de *Bloqueio Offline Otimista* envolveria apenas uma tabela de banco de dados com uma coluna para a versão e declarações *UPDATE* e *DELETE* que usam essa versão como parte de seu critério de atualização. É claro que você criará aplicações mais sofisticadas, então apresento a implementação usando um *Modelo de Domínio* (126) e *Mapeadores de Dados* (170). Isso revelará mais questões que surgem comumente durante a implementação de *Bloqueio Offline Otimista*.

Uma das primeiras coisas a fazer é assegurar-se de que sua *Camada Supertipo* (444) do domínio é capaz de armazenar qualquer informação necessária para implementar o *Bloqueio Offline Otimista* – a saber, dados da modificação e da versão.

```
class ObjetoDoDomínio...

    private Timestamp modificado;
    private String modificadoPor;
    private int versão;
```


Nossos dados são armazenados em um banco de dados relacional, então, cada tabela deve também armazenar dados sobre versão e modificação. Aqui está o esquema para uma tabela cliente assim como o SQL CRUD (criação, leitura, atualização e exclusão) padrão necessário para suportar o *Bloqueio Offline Otimista*.

```
tabela cliente...

    create table cliente (id bigint primary key, nome varchar, criadoPor varchar,
        criado datetime, modificadoPor varchar, modificado datetime, versão int)

CRUD SQL cliente...

    INSERT INTO cliente VALUES (?, ?, ?, ?, ?, ?, ?)
    SELECT * FROM cliente WHERE id = ?
    UPDATE cliente SET nome = ?, modificadoPor = ?, modificado = ?, versão = ?
        WHERE id = ? AND versão = ?
    DELETE FROM cliente WHERE id = ? AND versão = ?
```

Assim que você tiver mais do que algumas tabelas e objetos do domínio, irá querer introduzir uma *Camada Supertipo* (444) para seus *Mapeadores de Dados* (170) que lide com os segmentos tediosos e repetitivos de mapeamento O/R. Isso não apenas economiza muito trabalho durante a escrita de *Mapeadores de Dados* (170) como também permite o uso de um *Bloqueio Implícito* (422) para evitar que um desenvolvedor estrague uma estratégia de bloqueio esquecendo de codificar uma parte dos mecanismos de bloqueio.

A primeira parte a ser movida para seu mapeador abstrato é a construção SQL. Isso requer que você forneça mapeadores com um pouco de metadados sobre suas tabelas. Uma alternativa ao mapeador para construir SQL em tempo de execução é gerá-lo com código. Entretanto, deixarei a construção de declarações SQL como um exercício para o leitor. No mapeador abstrato abaixo, você verá que fiz algumas suposições sobre os nomes das colunas e posições de nossos dados para modificação. Isso se torna menos factível com dados legados. O mapeador abstrato irá provavelmente requerer que um pouco de metadados de colunas seja fornecido por cada mapeador concreto.

Uma vez que o mapeador abstrato tenha declarações SQL, pode gerenciar as operações CRUD. Aqui está como um método de busca é executado:

```
class MapeadorAbstrato...

    public MapeadorAbstrato (String tabela, String [ ] colunas) {
        this.tabela = tabela;
        this.colunas = colunas;
        criarDeclarações( );
    }
    public ObjetoDoDomínio buscar (Long id) {
        ObjetoDoDomínio obj = GerenciadorDeSessãoDeAplicação.lerSessão( ).lerMapaDeIdentidade( ).ler(id);
        if (obj == null) {
            Connection con = null;
            PreparedStatement dec = null;
            ResultSet rs = null;
            try {
                con = ConnectionManager.INSTANCE.getConnection( );
                dec = con.prepareStatement(SQLCarregar);
                dec.setLong(1, id.longValue( ));
                rs = dec.executeQuery( );
                if (rs.next( )) {
                    obj = carregar(id, rs);
                    String modificadoPor = rs.getString(colunas.length + 2);
                    Timestamp modificado = rs.getTimestamp(colunas.length + 3);
                    int versão = rs.getInt(colunas.length + 4);
                    obj.gravaCamposSistema(modificado, modificadoPor, versão);
                    GerenciadorDeSessãoDeAplicação.lerSessão( ).lerMapaDeIdentidade( ).grava(obj);
                } else {
                    throw new SystemException (tabela + " " + id + " não existe");
                }
            } catch (SQLException sqlEx) {
                throw new SystemException("erro não esperado buscando " + tabela + " " + id);
            } finally {
                liberaRecursosBD (rs, con, dec);
            }
        }
```

```
        return obj;
    }
    protected abstract ObjetoDoDomínio carregar (Long id, ResultSet rs) throws SQLException;
```

Há alguns itens a perceber aqui. Primeiro, o mapeador verifica um *Mapa de Identidade* (196) para se assegurar de que o objeto já não está carregado. Não usar um *Mapa de Identidade* (196), poderia resultar em versões diferentes de um objeto sendo carregadas em momentos diferentes em uma transação de negócio, levando a comportamento indefinido na sua aplicação, assim como criando uma confusão em qualquer verificação de versão. Assim que o conjunto resultante seja obtido, o mapeador transfere para um método abstrato de carga que cada mapeador concreto deve implementar para extrair seus campos e retornar um objeto ativado. O mapeador chama `gravarCamposDeSiystema( )` para gravar a versão e os dados de modificação do objeto do domínio abstrato. Embora um construtor pudesse parecer o meio mais apropriado de passar estes dados, fazer assim empurraria parte da responsabilidade pelo armazenamento da versão para cada mapeador concreto e objeto do domínio e dessa forma enfraqueceria o *Bloqueio Implícito* (422).

Aqui está como um método `carregar( )` concreto se parece:

```
class MapeadorDeCliente extends MapeadorAbstrato...

    protected ObjetoDoDomínio carregar(Long id, ResultSet rs) throws SQLException {
        String nome = rs.getString(2);
        return Cliente.ativar(id, nome, endereços);
    }
```

O mapeador abstrato irá gerenciar de maneira semelhante a execução das operações de atualização e exclusão. O trabalho aqui é verificar que a operação do banco de dados retorne um contador de linha igual a um. Se nenhuma linha tiver sido atualizada, o bloqueio otimista não pode ser obtido, e o mapeador precisa então levantar uma exceção de concorrência. Aqui está a operação de exclusão:

```
class MapeadorAbstrato...

    public void apagar(ObjetoDoDomínio objeto) {
        GerenciadorDeSessãoDeAplicação.lerSessão( ).lerMapaDeIdentidade( ).remove(objeto.lerId( ));
        Connection con = null;
        PreparedStatement dec = null;
        try {
            con = ConnectionManager.INSTANCE.getConnection( );
            dec = con.prepareStatement(SQLExcluir);
            dec.setLong(1, objeto.lerId( ).longValue( ));
            int contLinha = dec.executeUpdate( );
            if (contLinha == 0) {
                dispararExceçãoDeConcorrência(objeto);
            }
        } catch (SQLException e) {
            throw new SystemException ("erro inesperado na exclusão");
        } finally {
            liberaRecursosBD (con, dec);
        }
    }
```

```
protected void dispararExceçãoDeConcorrência (ObjetoDoDomínio objeto) throws SQLException {
    Connection con = null;
    PreparedStatement dec = null;
    ResultSet rs = null;
    try {
        con = ConnectionManager.INSTANCE.getConnection( );
        dec = con.prepareStatement(SQLVerificaVersão);
        dec.setInt(1, (int) objeto.lerId( ).longValue( ));
        rs = dec.executeQuery( );
        if (rs.next( )) {
            int versão = rs.getInt(1);
            String modificadoPor = rs.getString(2);
            Timestamp modificado = rs.getTimestamp(3);
            if (versão > objeto.lerVersão( )) {
                String quando = DateFormat.getDateTimeInstance( ).format(modificado);
                throw new ConcurrencyException (tabela + " " + objeto.lerId( ) +
                        " modificada por " + modificadoPor + " em " + quando);
            } else {
                throw new SystemException("erro inesperado ao checar o timestamp");
            }
        } else {
            throw new ConcurrencyExcception(tabela + " " + objeto.lerId( ) +
                    " foi apagado");
        }
    } finally {
        liberaRecursosBD (rs, com, dec);
    }
}
```

A declaração SQL usada para verificar a versão em uma exceção de concorrência também precisa ser conhecida pelo mapeador abstrato. Seu mapeador deveria construí-la quando constrói o SQL CRUD. Ela se parecerá com algo assim:

```
SQLdeVerificaçãoDeVersão...

    SELECT versão, modificadoPor, modificado FROM cliente WHERE id = ?
```

Este código não dá uma idéia perfeita das diversas partes executando por meio de diversas transações de sistema dentro de uma única transação de negócio. O mais importante a lembrar é que a aquisição de *Bloqueios Offline Otimistas* deve ocorrer dentro da mesma transação de sistema em que ocorre a confirmação das suas alterações para manter a consistência dos dados gravados. Com a verificação embutida nas declarações UPDATE e DELETE, isso não será um problema.

Dê uma olhada no uso de um objeto versão no código exemplo de *Bloqueio de Granularidade Alta* (412). Embora *Bloqueio de Granularidade Alta* (412) possa resolver alguns problemas de leitura inconsistente, um simples objeto versão não compartilhado pode ajudar a detectar leituras inconsistentes porque é um lugar conveniente para acrescentar comportamento de verificação otimista como `incrementar( )` ou `verificarSeAVersãoÉAMaisRecente( )`. Aqui está uma *Unidade de Trabalho* (187) na qual adicionamos verificações de leitura consistente no nosso processo de confirmação de gravação, por meio da medida mais drástica de incrementar a versão, porque não sabemos qual o nível de isolamento:

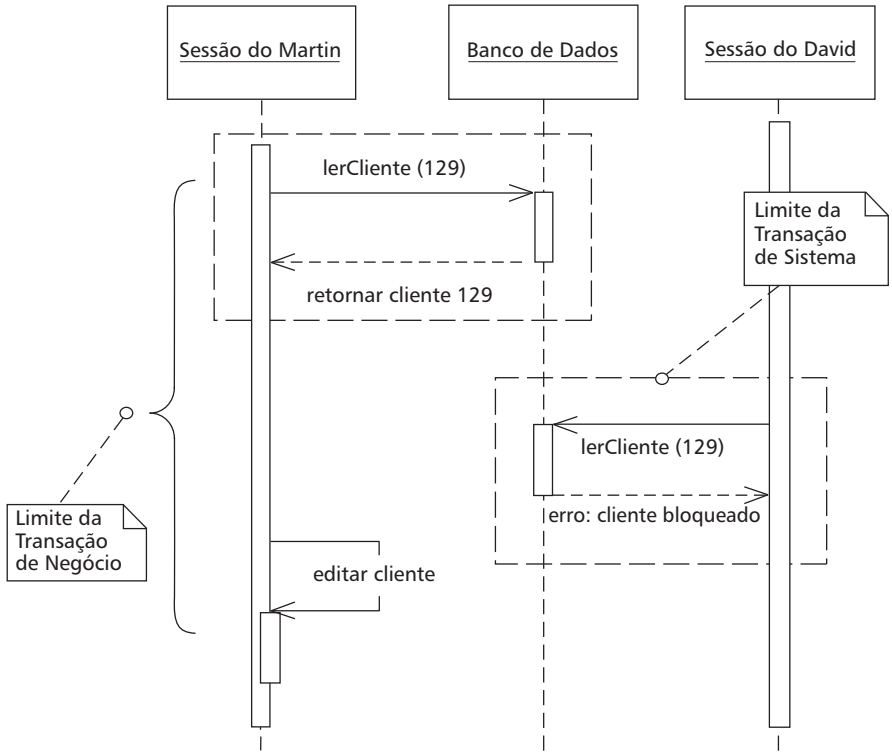
```
class UnidadeDeTrabalho...

    private List leituras = new ArrayList( );
    public void registrarLeitura (ObjetoDoDomínio objeto) {
        leituras.add(objeto);
    }
    public void confirmar ( ) {
        try {
            verificarLeiturasConsistentes( );
            inserirNovo( );
            excluirRemovido( );
            atualizarSujo( );
        } catch (ConcurrencyException e) {
            desfazTransaçãoDeSistema ( );
            throw e;
        }
    }
    public void verificarLeiturasConsistentes( ) {
        for (Iterator iterator = leituras.iterator( ); iterator.hasNext( ))
            ObjetoDoDomínio dependente = (ObjetoDoDomínio) iterator.next( );
            dependente.lerVersão( ).incrementar( );
        }
    }
```

Perceba que a *Unidade de Trabalho* (184) desfaz a transação de sistema quando detecta uma violação de concorrência. Você provavelmente decidiria desfazer a transação se houvesse qualquer exceção durante a confirmação. Não esqueça este passo! Como alternativa a objetos versão, você pode adicionar verificações de versão à sua interface mapeadora.

Bloqueio *Offline* Pessimista (Pessimistic Offline Lock)

por David Rice

Previne conflitos entre transações de negócio concorrentes permitindo que apenas uma transação de negócio acesse os dados de cada vez.



Já que a concorrência *offline* envolve manipulação de dados para uma transação de negócio que perdura por diversas solicitações, a abordagem mais simples pareceria ser a de ter uma transação de sistema aberta durante toda a transação de negócio. Infelizmente, isso não funciona sempre bem porque sistemas de transações não são equipados para trabalhar com transações longas. Por essa razão, você tem que usar diversas transações de sistema, em que você fica por conta própria para gerenciar o acesso concorrente aos seus dados.

A primeira abordagem a tentar é o *Bloqueio Offline Otimista* (392). Entretanto, esse padrão tem seus problemas. Se diversas pessoas acessarem os mesmos dados dentro de uma transação de negócio, uma delas confirmará a gravação com facilidade, mas as outras entrarão em conflito e falharão. Como o conflito só é detectado no final da transação de negócio, as vítimas farão todo o trabalho da transação apenas para descobrir no último minuto que tudo falhará e seu tempo terá sido desperdiçado. Se isso acontecer, com muita freqüência em transações de negócio longas, o sistema logo se tornará bastante impopular.

O *Bloqueio Offline Pessimista* previne conflitos evitando-os completamente. Ele força uma transação de negócio a obter um bloqueio sobre um fragmento de dados antes de começar a usá-lo, de modo que, na maior parte do tempo, assim que você começa uma transação de negócio, pode ter bastante certeza de que irá completá-la sem ser interrompido pelo controle de concorrência.

Como Funciona

Você implementa o *Bloqueio Offline Pessimista* em três fases: determinando de que tipo de bloqueios precisa, criando um gerenciador de bloqueios e definindo procedimentos para uma transação de negócios para usar os bloqueios. Além disso, se você estiver usando *Bloqueio Offline Pessimista* como complemento para o *Bloqueio Offline Otimista* (392), precisa determinar quais tipos de registros bloquear.

Em relação aos tipos de bloqueios, a primeira opção é um **bloqueio exclusivo de gravação**, que requer apenas que uma transação de negócios obtenha um bloqueio para editar dados da sessão. Isso evita conflito por não permitir que duas transações de negócio façam alterações no mesmo registro, ao mesmo tempo. O que este esquema de bloqueio ignora é a leitura dos dados, de modo que se não for crucial que uma sessão de visualização tenha os dados mais recentes, esta estratégia será suficiente.

Se for muito importante que uma transação de negócio tenha sempre os dados mais recentes, independentemente de suas intenções de editá-los, use o **bloqueio exclusivo de leitura**. Este requer que uma transação de negócio obtenha um bloqueio simplesmente para carregar o registro. Tal estratégia tem claramente o potencial de restringir severamente a concorrência de um sistema. Na maioria das aplicações corporativas, o bloqueio exclusivo de gravação vai permitir muito mais acesso concorrente aos registros do que este tipo de bloqueio.

Uma terceira estratégia combina os dois tipos para fornecer o bloqueio restritivo do bloqueio exclusivo de leitura, assim como o aumento de concorrência do bloqueio exclusivo de gravação. Chamado de **bloqueio de leitura/gravação**, é um pouco mais complexo do que os dois primeiros. O relacionamento dos bloqueios de leitura e de gravação é a chave para obter o melhor dos dois mundos:

- Bloqueios de leitura e de gravação são mutuamente exclusivos. Um registro não pode ser bloqueado para gravação se alguma outra transação de negócio possuir um bloqueio de leitura nele. Ele não pode ser bloqueado para gravação se alguma outra transação de negócio possuir um bloqueio de gravação nele.
- Bloqueios de leitura concorrentes são aceitáveis. A existência de um único bloqueio de leitura evita que alguma transação de negócio edite o registro, então não há mal em permitir qualquer número de sessões como leitores, uma vez que uma delas tenha recebido permissão de leitura.

Permitir diversos bloqueios de leitura é o que aumenta a concorrência do sistema. O aspecto negativo deste esquema é que é um pouco desagradável de implementar e apresenta mais desafios para os especialistas no domínio se preocuparem quando estiverem modelando o sistema.

Durante a escolha do tipo correto de bloqueio, pense em maximizar a concorrência do sistema, satisfazer às necessidades do sistema e minimizar a complexidade

do código. Tenha em mente também que a estratégia de bloqueio deve ser compreendida pelos analistas e modeladores do domínio. O bloqueio não é só um problema técnico. O tipo errado de bloqueio, simplesmente bloqueando todos os registros, ou bloquear os tipos errados de registros pode resultar em uma estratégia de *Bloqueio Offline Pessimista* ineficaz. Uma estratégia de *Bloqueio Offline Pessimista* ineficaz não evita conflitos no começo da transação de negócio ou degrada a concorrência de seu sistema multiusuário de tal maneira que ele pareça mais como um sistema monousuário. A estratégia de bloqueio errada não pode ser salva por uma implementação técnica apropriada. Na verdade, não é uma má idéia incluir o *Bloqueio Offline Pessimista* no seu modelo de domínio.

Assim que você tiver decidido seu tipo de bloqueio, defina seu gerenciador de bloqueios. O trabalho do gerenciador de bloqueio é dar ou negar qualquer solicitação de uma transação de negócio para obter ou liberar um bloqueio. Para executar seu trabalho, ele precisa saber o que está sendo bloqueado, assim como o pretendente a dono do bloqueio – a transação de negócio. É bastante possível que o seu conceito de uma transação de negócio não seja alguma *coisa* que possa ser identificada de forma única, o que torna um pouco difícil passar uma transação de negócio para o gerenciador de bloqueios. Neste caso, considere seu conceito de uma sessão, já que é mais provável que você tenha um objeto de sessão à sua disposição. Os termos "sessão" e "transação de negócio" são razoavelmente intercambiáveis. Desde que as transações de negócio sejam executadas serialmente dentro de uma sessão, esta sessão ficará bem como dona de um *Bloqueio Offline Pessimista*. O código exemplo deve lançar alguma luz sobre essa idéia.

O gerenciador de bloqueios não deve consistir de muito mais do que uma tabela que mapeia os bloqueios a seus donos. Um gerenciador de bloqueios simples poderia envolver uma tabela *hash* na memória ou poderia ser uma tabela de banco de dados. Não importa qual, mas você deve ter uma e apenas uma tabela de bloqueios, então, se ela estiver na memória, assegure-se de usar um *singleton* [Gang of Four]. Se o seu servidor de aplicação for distribuído em *clusters*, uma tabela de bloqueios na memória não funcionará, a menos que seja restrita a uma única instância no servidor. O gerenciador de bloqueios baseado em banco de dados é provavelmente mais apropriado uma vez que você esteja em um ambiente que use um servidor de aplicação distribuído em *clusters*.

O bloqueio, quer seja implementado como um objeto ou como um SQL sobre uma tabela de banco de dados, deve permanecer privado para o gerenciador de bloqueios. Transações de negócio devem interagir apenas com o gerenciador de bloqueios, nunca com um objeto de bloqueio.

Agora é hora de definir o protocolo de acordo com o qual uma transação de negócio deve usar o gerenciador de bloqueios. Este protocolo deve incluir o que bloquear e quando, quando liberar um bloqueio e como agir quando um bloqueio não puder ser obtido.

O que bloquear depende de quando bloquear, então vejamos o quando primeiro. Geralmente, a transação de negócio deve obter um bloqueio antes de carregar os dados, pois não há por que obter um bloqueio sem uma garantia de que você terá a versão mais recente do item bloqueado. Já que estamos obtendo bloqueios dentro de uma transação de sistema, há circunstâncias em que a ordem do bloqueio e da carga não importará. Dependendo do tipo do seu bloqueio, se você estiver usando transações de leitura repetíveis ou serializáveis, a ordem na qual você carrega objetos e obtém bloqueios pode não importar. Uma opção é executar uma verificação otimista so-

bre um item após obter o *Bloqueio Offline Pessimista*. Você deve estar bastante seguro de que tem a versão mais recente de um objeto após tê-lo bloqueado, o que geralmente se traduz na obtenção do bloqueio antes de carregar os dados.

Agora, o que estamos bloqueando? Parece que estamos bloqueando objetos ou registros ou apenas quase tudo, mas o que geralmente bloqueamos é realmente o ID, ou a chave primária, que usamos para buscar esses objetos. Isso nos permite obter o bloqueio antes de carregá-los. Bloquear o objeto funciona bem desde que não o obrigue a quebrar a regra de que um objeto seja a versão mais atual após você obter seu bloqueio.

A regra mais simples para liberar bloqueios é fazê-lo quando a transação de negócio for completada. Liberar um bloqueio antes disso poderia ser admissível, dependendo do seu tipo de bloqueio e da sua intenção de usar esse objeto novamente dentro da transação. Ainda assim, a menos que você tenha uma razão muito específica para liberar prematuramente o bloqueio, tal como algum problema especialmente desagradável de capacidade de resposta do sistema, continue executando a liberação no momento da conclusão da transação de negócio.

A ação mais fácil para uma transação de negócio que não consiga obter um bloqueio é abortá-la. O usuário deveria achar isso aceitável, já que o *Bloqueio Offline Pessimista* deveria resultar em falha bem antes na transação. O desenvolvedor e o projetista podem certamente ajudar nessa situação, não esperando até tarde na transação para obter um bloqueio especialmente disputado. Se for possível, obtenha todos os seus bloqueios antes de o usuário começar a trabalhar.

Para qualquer determinado item que você pretenda bloquear, o acesso à tabela de bloqueios deve ser serializado. Com uma tabela de bloqueios na memória é mais fácil serializar o acesso ao gerenciador de bloqueios inteiro com qualquer construção que sua linguagem de programação fornecer. Se você precisar de maior concorrência do que isso permite, esteja ciente de que está entrando em um território complexo.

Se a tabela de bloqueios for armazenada em um banco de dados, a primeira regra, é claro, é interagir com ela dentro de uma transação de sistema. Tire todo o proveito das capacidades de serialização que um banco de dados fornece. Com os bloqueios exclusivo de leitura e exclusivo de gravação, a serialização é uma simples questão de fazer o banco de dados reforçar uma restrição de unicidade na coluna que armazena o ID do item bloqueável. Armazenar bloqueios de leitura/gravação em um banco de dados torna as coisas um pouco mais difíceis, já que a lógica requer leituras da tabela de bloqueios além de inserções e desta maneira se torna imperativo evitar leituras inconsistentes. Uma transação de sistema com um nível de isolamento de serialização fornece a maior segurança, já que garante que não há leituras inconsistentes. Usar transações serializáveis por todo o nosso sistema poderia nos trazer problemas de desempenho, mas uma transação de sistema serializável separada para obtenção de bloqueios e um nível de isolamento menos rígido para outro uso poderia aliviar este problema. Outra opção é investigar se um procedimento armazenado poderia ajudar no gerenciamento de bloqueios. O gerenciamento de concorrência pode ser difícil, então não tenha receio de transferi-lo para seu banco de dados em momentos críticos.

A natureza serial do gerenciamento de bloqueios anuncia um gargalo de desempenho. Uma consideração importante aqui é a granularidade do bloqueio, já que quanto menos bloqueios requeridos, menor o gargalo que você terá. Um *Bloqueio de Granularidade Alta* (412) pode cuidar da disputa pela tabela de bloqueios.

Com um esquema de bloqueio pessimista de transações de sistema, como "SELECT FOR UPDATE..." ou EJBs de entidade, *deadlocks* são uma possibilidade que não pode ser ignorada, pois estes mecanismos de bloqueio irão esperar até que um bloqueio se torne disponível. Pense em um *deadlock* dessa forma. Dois usuários precisam dos recursos A e B. Se um obtém o bloqueio de A e o outro o de B, ambas as transações poderiam esperar para sempre pelo outro bloqueio. Considerando que estamos perdurando por diversas transações de sistema, esperar por um bloqueio não faz muito sentido, especialmente porque uma transação de negócio poderia durar 20 minutos. Ninguém quer esperar por esses bloqueios. E isto é bom, porque codificar uma espera envolve *timeouts* e logo se torna complicado. Simplesmente faça com que seu gerenciador de bloqueios levante uma exceção assim que um bloqueio esteja indisponível. Isto elimina o fardo de lidar com *deadlocks*.

Um requisito final é gerenciar *timeouts* dos bloqueios de sessões perdidas. Se uma máquina cliente pára de funcionar no meio de uma transação, essa transação perdida não consegue ser concluída e nem liberar seus bloqueios obtidos. Isso é algo sério em uma aplicação Web na qual sessões são abandonadas regularmente pelos usuários. O ideal é você ter um mecanismo de *timeout* gerenciado pelo seu servidor de aplicação, em vez de fazer com que sua aplicação trate *timeouts*. Servidores de aplicações Web fornecem uma sessão HTTP para isso. Os *timeouts* podem ser implementados registrando um objeto utilitário que libera todos os bloqueios quando a sessão HTTP se torna inválida. Outra opção é associar um *timestamp* a cada bloqueio e considerar inválido qualquer bloqueio mais velho que uma determinada idade.

Quando Usá-lo

O *Bloqueio Offline Pessimista* é apropriado quando a chance de conflitos entre sessões concorrentes for alta. Um usuário nunca deveria ter que jogar trabalho fora. Bloquear também é apropriado quando o custo de um conflito for alto demais independentemente de sua probabilidade. Bloquear cada entidade de um sistema quase que certamente criará grandes problemas de disputa de dados, então lembre-se de que o *Bloqueio Offline Pessimista* é bastante complementar ao *Bloqueio Offline Otimista* (392) e só use o *Bloqueio Offline Pessimista* quando for realmente necessário.

Se você tiver que usar o *Bloqueio Offline Pessimista*, deveria também considerar uma transação longa. Transações longas nunca são boas, mas em algumas situações elas podem não ser mais prejudiciais do que o *Bloqueio Offline Pessimista* e muito mais fáceis de programar. Faça algum teste de carga antes de decidir.

Não use estas técnicas se as suas transações de negócio couberem dentro de uma transação de sistema. Muitas técnicas de bloqueio pessimista de transações de sistema já estão nos servidores de aplicações e de banco de dados que você já está usando, entre eles a declaração SQL "SELECT FOR UPDATE" para bloqueio de banco de dados e o EJB de entidade para bloqueio de servidor de aplicações. Por que se preocupar com *timeouts*, visibilidade de bloqueio e coisas desse tipo, quando não há necessidade? Compreender esses tipos de bloqueios pode com certeza agregar muito valor à sua implementação de *Bloqueio Offline Pessimista*. Compreenda, entretanto, que o inverso não é verdadeiro! O que você leu aqui não irá lhe preparar para escrever um monitor de transações ou gerenciador de banco de dados. Todas as técnicas de bloqueio *offline* apresentadas neste livro dependem do seu sistema ter um monitor de transações real próprio.

Exemplo: Gerenciador de Bloqueios Simples (Java)

Neste exemplo, iremos primeiro construir um gerenciador de bloqueios para bloqueios exclusivos de leitura – lembre-se de que você precisa destes bloqueios para ler ou editar um objeto. Demonstraremos então como o gerenciador de bloqueios poderia ser usado em uma transação de negócios que perdura diversas transações de sistema.

O primeiro passo é definir a interface do nosso gerenciador de bloqueios.

```
interface GerenciadorDeBloqueiosExclusivosDeLeitura...

    public static final GerenciadorDeBloqueiosExclusivosDeLeitura INSTANCE =
        (GerenciadorDeBloqueiosExclusivosDeLeitura)
            Plugins.getPlugin(GerenciadorDeBloqueiosExclusivosDeLeitura.class);
        public void obterBloqueio (Long bloqueável, String dono) throws ConcurrencyException;
        public void liberarBloqueio (Long bloqueável, String dono);
        public void liberarTodosBloqueios (String dono);
```

Perceba que estamos identificando `bloqueável` com o tipo long e o `dono` como *string*. `Bloqueável` é do tipo long porque cada tabela no nosso banco de dados usa uma chave primária desse tipo que é única em todo o sistema e, desta maneira, serve como um bom ID bloqueável (que deve ser único por todos os tipos manipulados pela tabela de bloqueios). O ID do `dono` é uma *string* porque o exemplo será uma aplicação Web, e o ID da sessão HTTP é um bom dono de bloqueio dentro dela.

Escreveremos um gerenciador de bloqueios que interage diretamente com uma tabela de bloqueios no nosso banco de dados, em vez de com um objeto de bloqueio. Perceba que esta é a nossa própria tabela chamada `bloqueio`, como qualquer tabela de aplicação, e não parte do mecanismo interno de bloqueio do banco de dados. Obter um bloqueio é uma questão de inserir com sucesso uma linha na tabela de bloqueios. Liberá-lo é uma questão de apagar essa linha. Aqui está o esquema para a tabela de bloqueios e parte da implementação do gerenciador de bloqueios:

```
tabela bloqueio...

    create table bloqueio (idBloqueável bigint primary key, idDono bigint)

class ImplementaçãoBDGerenciadorDeBloqueiosExclusivosDeLeitura
    implements GerenciadorDeBloqueiosExclusivos...

    private static final String SQL_INSERIR =
        "insert into bloqueio values(?, ?)";
    private static final String SQL_EXCLUIR_UNICO =
        "delete from bloqueio where idBloqueável = ? and idDono = ?";
    private static final String SQL_EXCLUIR_TODOS =
        "delete from bloqueio where idDono = ?";
    private static final String SQL_VERIFICAR =
        "select idBloqueável from bloqueio where idBloqueável = ? and idDono = ?";
    public void obterBloqueio (Long bloqueável, String dono) throws ConcurrencyException {
        if (! temBloqueio(bloqueável, dono)) {
            Connection con = null;
            PreparedStatement dec = null;
            try {
                con = ConnectionManager.INSTANCE.getConnection( );
                dec = con.prepareStatement(SQL_INSERIR);
```

```
            dec.setLong(1, bloqueável.longValue( ));
            dec.setString(2, dono);
            dec.executeUpdate( );
        } catch (SQLEXception sqlEx) {
            throw new ConcurrencyException("não foi possível bloquear "+ bloqueável);
        } finally {
            liberaRecursosBD (con, dec);
        }
    }
}
public void liberarBloqueio (Long bloqueável, String dono) {
    Connection con = null;
    PreparedStatement dec = null;
    try {
        con = ConnectionManager.INSTANCE.getConnection( );
        dec = con.prepareStatement(SQL_EXCLUIR_UNICO);
        dec.setLong(1, bloqueável.longValue( ));
        dec.setString(2, dono);
        dec.executeUpdate( );
    } catch (SQLEXception sqlEx) {
        throw new SystemException("erro inesperado ao liberar bloqueio sobre "+ bloqueável);
    } finally {
        liberaRecursosBD (con, dec);
    }
}
```

O método público `liberarTodosBloqueios( )` e o método privado `temBloqueio( )` não são mostrados no gerenciador de bloqueios. O método `liberarTodosBloqueios( )` faz exatamente o que seu nome dá a entender e libera todos os bloqueios de um dono. O método `temBloqueio( )` consulta o banco de dados para verificar se um dono já possui um bloqueio. Não é incomum que um código de sessão tente obter um bloqueio que ele já possui. Isso significa que `obterBloqueio( )` deve primeiro verificar se o dono já não tem o bloqueio antes de tentar inserir a linha de bloqueio. Como a tabela de bloqueios é geralmente um ponto de disputa de recursos, estas leituras repetitivas podem degradar o desempenho da aplicação. Pode ser necessário que você coloque em memória cache bloqueios que já foram obtidos em nível de sessão para verificação de propriedade. Tenha cuidado ao fazer isso.

Agora iremos montar uma aplicação Web simples para realizar a manutenção de registros dos clientes. Primeiro, estabeleceremos um pouco de infra-estrutura para facilitar o processamento da transação de negócio. Alguns conceitos de uma sessão de usuário serão necessários para as camadas abaixo da camada Web, de modo que não poderemos depender apenas da sessão HTTP. Vamos nos referir a essa nova sessão como a sessão de aplicação para distingui-la da sessão HTTP. Sessões de aplicação armazenarão seus IDs, um nome de usuário e um *Mapa de Identidade* (196) para colocar em memória cache os objetos carregados ou criados durante a transação de negócio. Eles serão associados à *thread* correntemente em execução para que sejam encontrados.

```
class SessãoDeAplicação...

    private String usuário;
    private String id;
```

```
    private MapaDeIdentidade mpIdentidade;
    public SessãoDeAplicação (String usuário, String id, MapaDeIdentidade mpIdentidade) {
        this.usuário = usuário;
        this.mpIdentidade = mpIdentidade;
        this.id = id;
    }
```

```
class GerenciadorDeSessãoDeAplicação...

    private static ThreadLocal atual = new ThreadLocal( );
    public static SessãoDeAplicação lerSessão( ) {
        return (SessãoDeAplicação) atual.get( );
    }
    public static void gravarSessão(SessãoDeAplicação sessão) {
        atual.set(sessão);
    }
```

Iremos usar um *Controlador Frontal* (328) para lidar com as solicitações, de modo que precisaremos definir um comando. A primeira coisa que cada comando deve fazer é indicar a sua intenção de iniciar uma nova transação de negócio ou continuar uma que já exista. Isso é uma questão de estabelecer uma nova sessão de aplicação ou encontrar a atual. Aqui temos um comando abstrato que fornece métodos convenientes para estabelecer o contexto da transação de negócio.

```
interface Comando...

    public void init(HttpServletRequest solicitação, HttpServletResponse resposta);
    public void processar( ) throws Exception;
```

```
abstract class ComandoDeTransaçãoDeNegócio implements Comando...

    public void init (HttpServletRequest solicitação, HttpServletResponse resposta) {
        this.solicitação = solicitação;
        this.resposta = resposta;
    }
    protected void iniciarNovaTransaçãoDeNegócio ( ) {
        HttpSession sessãoHttp = lerSolicitação( ).getSession(true);
        SessãoDeAplicação sessãoDeAplicação = (SessãoDeAplicação) sessãoHttp.getAttribute(APP_SESSION);
        if (sessãoDeAplicação != null) {
            GerenciadorDeBloqueiosExclusivosDeLeitura.INSTANCE.liberarTodosBloqueios
                (sessãoDeAplicação.getId( ));
        }
        sessãoDeAplicação = new SessãoDeAplicação(lerSolicitação( ).getRemoteUser( ),
                            sessãoHttp.getId( ), new MapaDeIdentidade( ));
        GerenciadorDeSessãoDeAplicação.gravarSessão(sessãoDeAplicação);
        sessãoHttp.setAttribute(APP_SESSION, sessãoDeAplicação);
        sessãoHttp.setAttribute(LOCK_REMOVER,
                    new RemovedorDeBloqueio(sessãoDeAplicação.getID( )));
    }
    protected void continuarTransaçãoDeNegócio( ){
        HttpSession sessãoHttp = lerSolicitação( ).getSession( );
        SessãoDeAplicação sessãoDeAplicação = (SessãoDeAplicação) sessãoHttp.getAttribute(APP_SESSION);
        GerenciadorDeSessãoDeAplicação.gravarSessão(sessãoDeAplicação);
    }
    protected HttpServletRequest lerSolicitação ( ) {
        return solicitação;
```

```
    }
    protected HttpServletResponse lerResposta ( ) {
        return resposta;
    }
```

Perceba que, quando estabelecemos uma nova sessão de aplicação, removemos os bloqueios de qualquer uma existente. Também adicionamos um *listener* para os eventos associados à sessão HTTP que removerão quaisquer bloqueios possuídos por uma sessão de aplicação quando a sessão HTTP correspondente expirar.

```
class RemovedorDeBloqueios implements HttpSessionBindingListener...

    private String idSessão;
    public RemovedorDeBloqueios (String idSessão) {
        this.idSessão = idSessão;
    }
    public void valorSolto (HttpSessionBindingEvent evento) {
        try {
            iniciarTransaçãoDeSistema( );
            GerenciadorDeBloqueiosExclusivosDeLeitura.INSTANCE.liberarTodosBloqueios(this.idSessão);
            confirmarTransaçãoDeSistema( );
        } catch (Exception e) {
            tratarErroSério(e);
        }
    }
```

Nossos comandos contêm tanto lógica de negócio padrão quanto gerenciamento de bloqueio, e cada comando deve ser executado dentro dos limites de uma única transação de sistema. Para garantirmos isso, podemos decorá-lo [Gang of Four] com um objeto comando transacional. Assegure-se de que todo o bloqueio e negócio de domínio padrão ocorram dentro de uma única transação de sistema. Os métodos que definem os limites da transação de sistema dependem do seu contexto de distribuição. É obrigatório desfazer a transação do sistema quando uma exceção de concorrência, ou qualquer outra exceção for detectada, pois isso irá evitar que quaisquer alterações sejam inseridas nos dados permanentes quando um conflito ocorre.


```
class ComandoTransacional implements Comando...

    public ComandoTransacional (Comando impl) {
        this.impl = impl;
    }
    public void processar( ) throws Exception {
        iniciarTransaçãoDeSistema( );
        try {
            impl.processar( );
            confirmarTransaçãoDeSistema( );
        } catch (Exception e) {
            desfazerTransaçãoDeSistema( );
            throw e;
        }
    }
```

Agora é uma questão de escrever o *servlet* controlador e os comandos concretos. O *servlet* controlador tem a responsabilidade de encapsular cada comando com um controle de transação. Os comandos concretos devem estabelecer o contexto da transação de negócio, executar lógica de domínio e obter e liberar bloqueios onde for apropriado.

```
class ServletControlador extends HttpServlet...

    protected void doGet (HttpServletRequest solicitação, HttpServletResponse resposta)
                        throws ServletException, IOException {
        try {
            String nomeDoComando solicitação.getParameter("comando");
            Comando com = lerComando(nomeDoComando);
            comando.init(solicitação, resposta);
            comando.processar( );
        } catch (Exception e) {
            gravarExceção(e, resposta.getWriter( ));
        }
    }
    private Comando lerComandos(String nome) {
        try {
            String nomeDaClasse = (String) comandos.get(nome);
            Comando com = (Comando) Class.forName(nomeDaClasse).newInstance( );
            return new ComandoTransacional(com);
        } catch (Exception e) {
            e.printStackTrace( );
            throw new SystemException ("não foi possível criar objeto comando para " + nome);
        }
    }
```

```
class ComandoEditarCliente extends Comando TransaçãoDeNegócio...

    public void processar ( ) throws Exception {
        iniciarNovaTransaçãoDeNegócio( );
        Long idCliente = new Long(lerSolicitação( ).getParameter("id_cliente");
        GerenciadorDeBloqueiosExclusivosDeLeitura.INSTANCE.obterBloqueio(
                    idCliente, GerenciadorDeSessãoDeAplicação.lerSessão( ).getId( ));
        Mapeador mapeadorDeCliente = RegistroMapeador.INSTANCE.lerMapeador(Cliente.class);
        Cliente cliente = (Cliente) mapeadorDeCliente.buscar(idCliente);
        lerSolicitação( ).lerSessão( ).setAttribute("cliente", cliente);
        forward("/editarCliente.jsp");
    }
```

Bloqueio *Offline* Pessimista

```
class ComandoGravarCliente extends Comando TransaçãoDeNegócio...

    public void processar ( ) throws Exception {
        continuarTransaçãoDeNegócio( );
        Cliente cliente = (Cliente) lerSolicitação( ).lerSessão( ).getAttribute("cliente");
        String nome = lerSolicitação( ).getParameter("nomeDoCliente");
        cliente.gravarNome(nome);
        Mapeador mapeadorDeCliente = RegistroMapeador.INSTANCE.lerMapeador(Cliente.class);
        mapeadorDoCliente.atualizar(cliente);
        GerenciadorDeBloqueiosExclusivosDeLeitura.INSTANCE.liberarBloqueio(cliente.lerID( ),
                                    GerenciadorDeSessãoDeAplicação.lerSessão( ).getID( ));
        forward("/clienteGravado.jsp");
    }
```

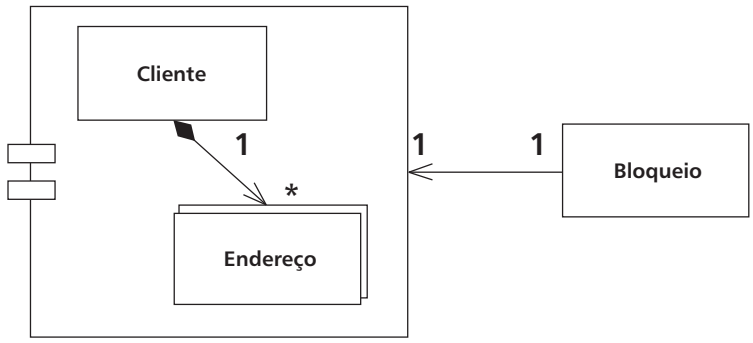
Os comandos recém-mostrados evitarão que duas sessões quaisquer trabalhem com o mesmo cliente ao mesmo tempo. Qualquer outro comando na aplicação que trabalhar com um objeto cliente deve se assegurar de obter o bloqueio ou de trabalhar apenas com um cliente bloqueado por um comando anterior na mesma transação de negócio. Visto que temos uma verificação `temBloqueio( )` no gerenciador de bloqueios, poderíamos simplesmente obter o bloqueio em cada comando. Isso poderia ser ruim para o desempenho, mas certamente garantiria que temos um bloqueio. O *Bloqueio Implícito* (422) discute outras abordagens seguras de mecanismos de bloqueio.

A quantidade de código no *framework* poderia parecer um pouco fora de proporção em relação à quantidade de código de domínio. De fato, o *Bloqueio Offline Pessimista* requer, no mínimo, sincronizar a atuação de uma sessão de aplicação, uma transação de negócio, um gerenciador de bloqueio e uma transação de sistema, o que é claramente um desafio. Esse exemplo serve mais como inspiração do que como modelo de arquitetura, pois ele sofre de falta de robustez em muitas áreas.

Bloqueio de Granularidade Alta (Coarse-Grained Lock)

por David Rice e Matt Foemmel

Bloqueia um conjunto de objetos relacionados utilizando para isso um único bloqueio.



Os objetos muitas vezes podem ser editados em grupos. Talvez você tenha um cliente e seu conjunto de endereços. Se este for o caso, ao usar a aplicação, faz sentido bloquear todos esses itens se você quiser bloquear qualquer um deles. Ter um bloqueio separado para objetos individuais apresenta um número de desafios. Primeiro, qualquer um que for manipulá-los tem que escrever código que busque a todos para poder bloqueá-los. Isso é suficientemente fácil com um cliente e seus endereços, mas fica bastante complicado uma vez que se tem mais grupos para bloquear. E se os grupos ficarem complicados? Onde fica este comportamento quando seu *framework* está gerenciando a obtenção de bloqueios? Se a sua estratégia de bloqueios requerer que um objeto seja carregado para que seja bloqueado, como com o *Bloqueio Offline Otimista* (392), bloquear um grupo grande afetará o desempenho. Além disso, com o *Bloqueio Offline Pessimista* (401), um conjunto de bloqueios grande é uma dor de cabeça para gerenciar e aumenta a disputa pela tabela de bloqueios.

Um *Bloqueio de Granularidade Alta* é um único bloqueio que cobre muitos objetos. Ele não apenas simplifica o próprio ato de bloquear como também o libera de ter que carregar todos os membros de um grupo para poder bloqueá-los.

Como Funciona

O primeiro passo na implementação do *Bloqueio de Granularidade Alta* é criar um único ponto de disputa para bloquear um grupo de objetos. Isso faz com que apenas um bloqueio seja necessário para bloquear todo o conjunto. Você fornece então o caminho mais curto possível para encontrar esse ponto único de bloqueio de modo a minimizar os membros do grupo que devem ser identificados e possivelmente carregados na memória no processo de obtenção desse bloqueio.

Com o *Bloqueio Offline Otimista* (392), fazer com que cada item do grupo compartilhe uma versão (veja a Figura 16.2) cria o ponto único de disputa, o que significa compartilhar a *mesma* versão, não uma versão *igual*. Incrementar essa versão bloqueará todo o grupo com um **bloqueio compartilhado**. Configure seu modelo para indicar cada membro do grupo na versão compartilhada, e você terá certamente minimizado o caminho para o ponto de disputa.

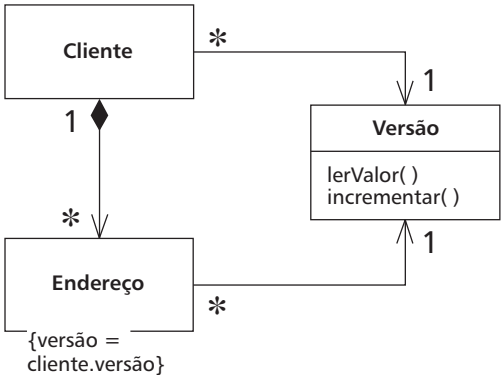


Figura 16.2 Compartilhando uma versão.

Um *Bloqueio Offline Pessimista* (401) compartilhado requer que cada membro do grupo compartilhe algum tipo de *token* bloqueável, sobre o qual ele deve então ser obtido. Como o *Bloqueio Offline Pessimista* (401) é muitas vezes usado como um complemento do *Bloqueio Offline Otimista* (392), uma versão de objeto compartilhada é uma candidata excelente para o papel de *token* bloqueável (Figura 16.3).

Eric Evans e David Siegel [Evans] definem um **agregado** como um *cluster* de objetos associados que tratamos como uma unidade na hora de alterar dados. Cada agregado tem uma **raiz** que fornece o único ponto de acesso aos membros do conjunto e um **limite** que define o que está incluído no conjunto. As características do agregado pedem um *Bloqueio de Granularidade Alta*, já que trabalhar com qualquer um de seus membros requer bloquear todos eles. Bloquear um agregado é uma alternativa a um bloqueio compartilhado que eu chamo de **bloqueio de raiz** (veja a Figura 16.4). Por definição bloquear a raiz bloqueia todos os membros do agregado. O bloqueio da raiz nos dá um único ponto de disputa.

Usar um bloqueio de raiz como *Bloqueio de Granularidade Alta* torna necessário implementar navegação para a raiz no grafo de objetos. Isso permite a um mecanismo de bloqueio, quando solicitado a bloquear qualquer objeto no agregado, navegar

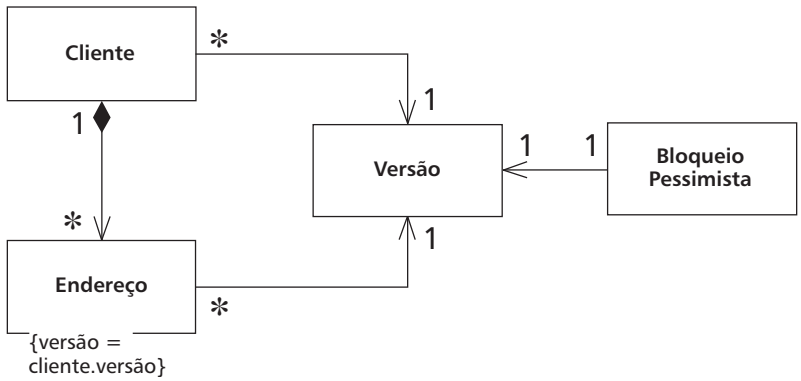


Figura 16.3 Bloqueando uma versão compartilhada.

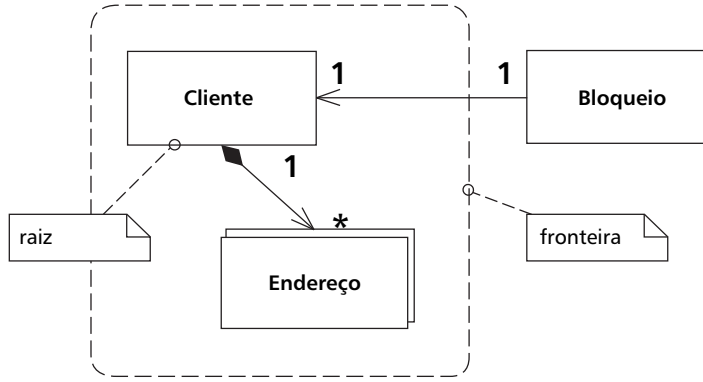


Figura 16.4 Bloqueando a raiz.

até a raiz e bloqueá-la em vez do objeto. Esta navegação pode ser realizada de algumas formas. Você pode manter uma navegação direta para a raiz em cada objeto no agregado, ou pode usar uma seqüência de relacionamentos intermediários. Por exemplo, em uma hierarquia a raiz óbvia é o ancestral no topo, ao qual você pode conectar os descendentes diretamente. Alternativamente, você pode dar a cada nó uma conexão ao seu ancestral imediato e navegar nessa estrutura para alcançar a raiz. Em um grafo grande, esta última estratégia poderia causar problema de desempenho, já que cada ancestral deve ser carregado para determinar se ele próprio também tem um ancestral. Assegure-se de usar uma *Carga Tardia* (200) ao carregar os objetos que constituem o caminho para a sua raiz. Isso não apenas evita que os objetos sejam carregados antes de serem necessários como evita um laço de mapeamento infinito quando você mapeia um relacionamento bidirecional. Esteja alerta para o fato de que *Cargas Tardias* (200) para um único agregado podem ocorrer por diversas transações de sistema e, deste modo, você pode acabar com um agregado construído com partes inconsistentes. É claro que isso não é bom.

Perceba que um bloqueio compartilhado também funciona para bloqueio de agregados, já que bloquear qualquer objeto no agregado irá simultaneamente bloquear a raiz.

As implementações do bloqueio compartilhado e do bloqueio da raiz do *Bloqueio de Granularidade Alta* têm ambos seus compromissos. Ao usar um banco de dados relacional, o bloqueio compartilhado carrega o fardo de que quase todas as suas seleções irão requerer um *join* com a tabela versão. Contudo, carregar objetos ao navegar para a raiz pode causar um problema de desempenho também. O bloqueio da raiz e o *Bloqueio Offline Pessimista* (401) talvez sejam uma combinação estranha. Quando você navega para a raiz e a bloqueia, pode precisar recarregar alguns objetos para garantir que estejam atualizados e, como sempre, construir um sistema sobre dados legados colocará numerosas restrições sobre sua escolha de implementação. Implementações de bloqueios existem em abundância, e as sutilezas são até mais numerosas. Assegure-se de chegar a uma implementação que satisfaça a suas necessidades.

Quando Usá-lo

A razão mais óbvia para usar um *Bloqueio de Granularidade Alta* é para satisfazer requisitos de negócio. Este é o caso ao bloquear um agregado. Considere um objeto *leasing* que possua uma coleção de bens. Provavelmente não faz sentido para o negócio que um usuário edite o objeto de *leasing* e outro usuário edite simultaneamente um bem. Bloquear um bem ou o *leasing* deveria resultar no bloqueio do *leasing* e de todos os seus bens.

Uma conseqüência muito positiva de usar o *Bloqueio de Granularidade Alta* é que obter e liberar bloqueios é menos custoso. Esta é certamente uma motivação legítima para usá-los. O bloqueio compartilhado pode ser usado além do conceito do agregado [Evans], mas seja cauteloso ao trabalhar com requisitos não-funcionais como desempenho. Cuidado com a criação de relacionamentos não-naturais entre objetos para facilitar o *Bloqueio de Granularidade Alta*.

Exemplo: Bloqueio *Offline* Otimista (392) Compartilhado (Java)

Para este exemplo temos um modelo de domínio com uma *Camada Supertipo* (444), um banco de dados relacional como nosso armazenamento persistente e *Mapeadores de Dados* (170).

A primeira coisa a fazer é criar uma tabela e uma classe de versão. Para manter as coisas simples, criaremos uma classe de versão bastante versátil que irá não apenas armazenar seu valor, mas também terá um método estático de busca. Perceba que estamos usando um mapa de identidade para colocar em memória cache versões para uma sessão. Se os objetos compartilharem uma versão, é crucial que todos apontem para exatamente a mesma instância dela. Como a classe de versão é uma parte do nosso modelo de domínio, é provavelmente uma forma pobre colocar código de banco de dados nele, de modo que deixarei a versão com separação de código de banco de dados na camada de mapeamento como um exercício para você.

```
tabela versão...

    create table versão (id bigint primary key, valor bigint,
        modificadoPor varchar, modificado datetime)

class Versão...

    private Long id;
    private long valor;
    private String modificadoPor;
    private Timestamp modificado;
    private boolean bloqueado;
    private boolean éNova;
    private static final String SQL_ATUALIZAR =
        "UPDATE versão SET valor = ?, modificadoPor = ?, modificado = ?, " +
        "WHERE id = ? AND valor = ?";
    private static final String SQL_EXCLUIR =
        "DELETE FROM versão WHERE id = ? AND valor = ?";
    private static final String SQL_INSERIR =
        "INSERT INTO versão VALUES (?, ?, ?, ?)";
    private static final String SQL_CARREGAR=
        "SELECT id, valor, modificadoPor, modificado FROM versão WHERE id = ? ";
    public static Versão buscar(Long id) {
```

```
        Versão versão = GerenciadorDeSessãoDeAplicação.lerSessão( ).lerMapaDeIdentidade( ).lerVersão(id);
        if (versão == null) {
            versão = carregar(id);
        }
        return versão;
    }
    private static Versão carregar(Long id) {
        ResultSet rs = null;
        Connection con = null;
        PreparedStatement dec = null;
        Versão versão = null;
        try {
            con = ConnectionManager.INSTANCE.getConnection( );
            dec = con.prepareStatement(SQL_CARREGAR);
            dec.setLong(1, id.longValue( ));
            rs = dec.executeQuery( );
            if (rs.next( )) {
                long valor = rs.getLong(2);
                String modificadoPor = rs.getString(3);
                Timestamp modificado = rs.getTimestamp(4);
                versão = new Versão(id, valor, modificadoPor, modificado);
                GerenciadorDeSessãoDeAplicação.lerSessão( ).lerMapaDeIdentidade( ).gravarVersão(versão);
            } else {
                throw new ConcurrencyException ("versão " + id + " não encontrada.");
            }
        } catch (SQLException sqlEx) {
            throw new SystemException ("erro de sql inesperado ao carregar a versão", sqlEx);
        } finally {
            liberarRecursosBD(rs, con, dec);
        }
        return versão;
    }
```

A versão também sabe como criar a si mesma. A inserção no banco de dados é separada da criação para permitir o adiamento da inserção até que pelo menos um dono seja inserido no banco de dados. Cada um dos nossos *Mapeadores de Dados* (170) do domínio pode chamar a inserção na versão com segurança durante a inserção do objeto do domínio correspondente. A versão executa um rastreamento verificando se ele é novo para se assegurar de que será inserido apenas uma vez.


```
class Versão...

    public static Versão criar ( ) {
        Versão versão = new Versão (GeradorDeId.INSTANCE.próximaId( ), 0,
                GerenciadorDeSessãoDeAplicação.lerSessão( ).lerUsuário( ), now ( ));
        versão.éNova = true;
        return versão;
    }
    public void inserir ( ) {
        if( éNova( )) {
            Connection con = null;
            PreparedStatement dec = null;
            try {
                con = ConnectionManager.INSTANCE.getConnection( );
```

```
            dec = con.prepareStatement(SQL_INSERIR);
            dec.setLong(1, this.lerId( ).longValue( ));
            dec.setLong(2, this.lerValor( ));
            dec.setString(3, this.lerModificadoPor( ));
            dec.setTimestamp(4, this.lerModificado( ));
            dec.executeUpdate( );
            GerenciadorDeSessãoDeAplicação.lerSessão( ).lerMapaDeIdentidade( ).gravarVersão(this);
            éNova = false;
        } catch (SQLException sqlEx) {
            throw new SystemException ("erro de sql inesperado ao inserir versão", sqlEx);
        } finally {
            cleanupDBResources(con, dec);
        }
    }
}
```

A seguir, temos um método `incrementar( )` que incrementa o valor da versão na linha correspondente no banco de dados. É provável que diversos objetos em um conjunto de alterações compartilhem a mesma versão, de modo que a versão primeiro se assegura de que já não esteja bloqueada antes de incrementar a si mesma. Após chamar o banco de dados, o método `incrementar( )` deve verificar se a linha da versão foi realmente atualizada. Se este método retornar um contador de linha igual a zero, ele detectou uma violação de concorrência e levanta uma exceção.

```
class Versão...

    public void incrementar ( ) throws ConcurrencyException {
        if (! estáBloqueado( )) {
            Connection con = null;
            PreparedStatement dec = null;
            try {
                con = ConnectionManager.INSTANCE.getConnection( );
                dec = con.prepareStatement(SQL_ATUALIZAR);
                dec.setLong(1, valor + 1);
                dec.setString(2, this.lerModificadoPor( ));
                dec.setTimestamp(3, this.lerModificado( ));
                dec.setLong(4, id.longValue( ));
                dec.setLong(5, valor);
                int contadorDeLinha = dec.executeUpdate( );
                if (contadorDeLinha == 0) {
                    levantarExceçãoDeConcorrência( );
                }
                valor ++;
                bloqueado = true;
                } catch (SQLException sqlEx) {
                    throw new SystemException ("erro de sql inesperado ao incrementar versão", sqlEx);
                } finally {
                    liberarRecursosBD(con, dec);
            }
        }
    }
    private void levantarExceçãoDeConcorrência( ) {
        Versão versãoAtual = carregar(this.lerId( ));
```

```
            throw new ConcurrencyException (
                "versão modificada por " + versãoAtual.modificadoPor + " em " +
                DateFormat.getDateTimeInstance( ).format(versãoAtual.lerModificado( ) ) );
        }
```

Com este código assegure-se de chamar `incrementar` apenas na transação de sistema na qual você confirmar sua transação de negócio. O *flag* `estáBloqueado` faz isso de modo que incrementar em transações anteriores resultará em obtenção falsa de bloqueio durante a transação de confirmação. Isso não é problema, porque toda a questão de um bloqueio otimista é que você apenas obtém o bloqueio quando confirma a gravação (*commit*).

Quando você usa este padrão, pode querer ver se os seus dados ainda estão atuais com o banco de dados em uma transação de sistema anterior. Você pode fazer isso acrescentando um método `verificarAtual` na classe versão que simplesmente verifique se um *Bloqueio Offline Otimista* (392) está disponível sem realizar nenhuma atualização.

O método de exclusão que executa o SQL para remover a versão do banco de dados não é mostrado. Se o contador de linha retornado for zero, uma exceção de concorrência é levantada. Acontece assim, porque o *Bloqueio Offline Otimista* (392) provavelmente não foi obtido durante a exclusão do último dos objetos usando esta versão. Isso nunca deveria acontecer. O verdadeiro truque é saber quando não há problema em excluir uma versão compartilhada. Se você estiver compartilhando uma versão por um agregado, simplesmente apague-a após você apagar a raiz do agregado. Outros cenários tornam as coisas muito mais problemáticas. Uma possibilidade é que o objeto versão mantenha um contador de referência de seus donos e apague a si próprio quando o contador chegar a zero. Esteja avisado de que isso poderia contribuir para um objeto versão bastante sofisticado. Assim que sua versão ficar complicada, você poderia considerar torná-la um objeto de domínio pleno. Isso faz sentido, mas, é claro, seria um objeto de domínio especial sem uma versão.

Agora vamos ver como usamos a versão compartilhada. A *Camada Supertipo* (444) do domínio contém um objeto versão, em vez de um contador simples. Cada *Mapeador de Dados* (170) pode gravar a versão ao carregar o objeto do domínio.

```
class ObjetoDoDomínio...

    private Long id;
    private Timestamp modificado;
    private String modificadoPor;
    private Versão versão;
    public void gravarCamposDeSistema (Versão versão, Timestamp modificado, String modificadoPor) {
        this.versão = versão;
        this.modificado = modificado;
        this.modificadoPor = modificadoPor;
    }
```

Para a criação, vamos olhar um agregado que consiste em uma raiz cliente e seus endereços. O método de criação do cliente criará a versão compartilhada. Clientes terão um método `acrescentarEndereço( )` que cria um endereço passando a versão do cliente. Nosso mapeador abstrato de banco de dados inserirá a versão antes de inserir os objetos de domínio correspondentes. Lembre-se de que a versão garantirá que será inserida apenas uma vez.

```
class Cliente extends ObjetoDoDomínio...

    public static Cliente criar(String nome) {
        return new Cliente(GeradorDeId.INSTANCE.próximaId( ), Versão.criar( ), nome, new ArrayList( ));
    }
```

```
class Cliente extends ObjetoDoDomínio...

    public Endereço acrescentarEndereço(String linha1, String cidade, String estado) {
        Endereço endereço = Endereço.criar(this, lerVersão( ), linha1, cidade, estado);
        endereços.add(endereço);
        return endereço;
    }
```

```
class Endereço extends ObjetoDoDomínio...

    public static Endereço criar (Cliente cliente, Versão versão,
        String linha1, String cidade, String estado) {
        return new Endereço (GeradorDeId.INSTANCE.próximaId( ), versão, cliente,
                    linha1, cidade, estado);
    }
```

```
class MapeadorAbstrato...

    public void inserir (ObjetoDoDomínio objeto) {
        objeto.lerVersão( ).inserir( );
```

O incremento deveria ser chamado em uma versão pelo *Mapeador de Dados* (165) antes que ele atualize ou exclua um objeto.

```
class MapeadorAbstrato...

    public void atualizar (ObjetoDoDomínio objeto) {
        objeto.lerVersão( ).incrementar( );
```

```
class MapeadorAbstrato...

    public void excluir (ObjetoDoDomínio objeto) {
        objeto.lerVersão( ).incrementar( );
```

Como este é um agregado, excluímos os endereços quando excluímos o cliente. Isso nos permite excluir a versão imediatamente após.

```
class MapeadorDeCliente extends MapeadorAbstrato...

    public void excluir (ObjetoDoDomínio objeto) {
        Cliente cli = (Cliente) objeto;
        for (Iterator iterator = cli.lerEndereços( ).iterator( ); iterator.hasNext( );) {
            Endereço endereço = (Endereço) iterator.next( );
            RegistroMapeador.lerMapeador( Endereço.class).excluir(endereço);
        }
        super.excluir(objeto);
        cli.lerVersão( ).excluir( );
    }
```

Exemplo: Bloqueio *Offline* Pessimista (401) Compartilhado (Java)

Precisamos de algum tipo de *token* bloqueável que possamos associar a todos os objetos no conjunto relacionado. Como discutido anteriormente, usaremos *Bloqueio Offline Pessimista* (401) como um complemento do *Bloqueio Offline Otimista* (392), a fim de que possamos usar como *token* bloqueável a versão compartilhada. Usaremos todo o mesmo código para chegar a uma versão compartilhada.

A única questão é que alguns dos nossos dados devem ser carregados a fim de obter a versão. Se obtivermos o *Bloqueio Offline Pessimista* (401) após carregarmos seus dados, como saberemos que os dados são os atuais? Algo que podemos fazer facilmente é incrementar a versão dentro da transação de sistema na qual obtivemos o *Bloqueio Offline Pessimista* (401). Assim que essa transação de sistema for confirmada, nosso bloqueio pessimista é válido, e sabemos que temos a cópia mais recente de qualquer dado compartilhando essa versão, independentemente de onde carregamos dentro da transação de sistema.

```
class ComandoCarregarCliente...

    try {
        Cliente cliente = (Cliente) RegistroMapeador.lerMapeador(Cliente.class).buscar(id);
        GerenciadorDeBloqueiosExclusivosDeLeitura.INSTANCE.obterBloqueio
            (cliente.lerId( ), GerenciadorDeSessãoDeAplicação.lerSessão( ).lerId( ));
        cliente.lerVersão( ).incrementar( );
        GerenciadorDeTransações.INSTANCE.confirmar( );
    } catch (Exception e) {
        GerenciadorDeTransações.INSTANCE.desfazer( );
        throw e;
    }
```

Você pode ver que o incremento da versão poderia ser algo que você iria querer construir no seu gerenciador de bloqueios. Pelo menos você quer decorar [Gang of Four] seu gerenciador de bloqueios com código que incremente a versão. Seu código de produção irá, é claro, requerer a manipulação de exceções e o controle de transações mais robustos do que o exemplo mostra.

Exemplo: Bloqueio *Offline* Otimista (392) de Raiz (Java)

Este exemplo faz a maioria das mesmas suposições dos exemplos anteriores, incluindo uma *Camada Supertipo* (444) e *Mapeadores de Dados* (170) de domínio. Há um objeto versão, mas neste caso ele não será compartilhado. Ele simplesmente fornece um método **incrementar( )** conveniente para permitir mais facilmente a obtenção do Bloqueio *Offline Otimista* (392) fora do *Mapeador de Dados* (170). Também estamos usando uma Unidade de Trabalho (187) para rastrear nosso conjunto de alterações.

Nosso agregado contém relacionamentos pai-filho, de modo que usaremos navegação filho-para-pai para encontrar a raiz. Precisaremos acomodar isso nos nossos modelos de domínio e de dados.

```
class ObjetoDoDomínio...

    private Long id;
    private ObjetoDoDomínio pai;
    public ObjetoDoDomínio (Long id, ObjetoDoDomínio pai) {
```

```
        this.id = id;
        this.pai = pai;
    }
```

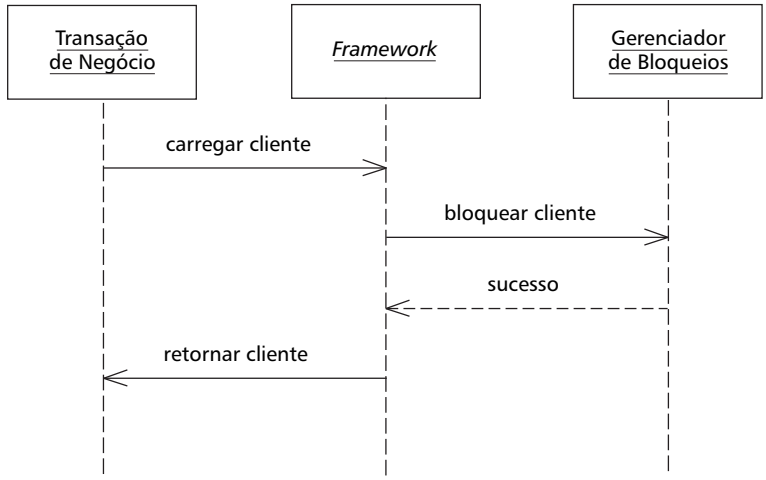
Assim que tivermos nossos donos, podemos obter nossos bloqueios de raiz antes de confirmarmos a gravação da *Unidade de Trabalho*.

```
class UnidadeDeTrabalho...

    public void confirmar ( ) throws SQLException {
        for (Iterator iterator = _objetosModificados.iterator( ); iterator.hasNext( );) {
            ObjetoDoDomínio objeto = (ObjetoDoDomínio) iterator.next( );
            for(ObjetoDoDomínio dono = objeto; dono != null; dono = dono.lerPais( )) {
                dono.lerVersão( ).incrementar( );
            }
        }
        for (Iterator iterator = _objetosModificados.iterator( ); iterator.hasNext( ); ) {
            ObjetoDoDomínio objeto = (ObjetoDoDomínio) iterator.next( );
            Mapeador mapeador = RegistroMapeador.lerMapeador(objeto.getClass( ));
            mapeador.atualizar(objeto);
        }
    }
```

Bloqueio Implícito (Implicit Lock)

por David Rice

Permite ao código de uma camada supertipo ou de um framework obter bloqueios offline.



A chave de qualquer esquema de bloqueio é que não haja lacunas no seu uso. Esquecer de escrever uma única linha de código que requeira um bloqueio pode produzir todo um esquema de bloqueio *offline* inútil. Falhar na recuperação de um bloqueio de leitura em que outras transações usam bloqueios de gravação significa que você poderia obter dados de sessão não-atualizados. Falhar no uso apropriado de um contador de versão pode resultar na gravação não-intencional por cima das alterações de alguém. Geralmente, se um item pode ser bloqueado *em qualquer lugar*, deve ser bloqueado *em todo lugar*. Ignorar a estratégia de bloqueio da sua aplicação permite a uma transação de negócio criar dados inconsistentes. Não liberar bloqueios não irá corromper seus dados, mas acabará trazendo sérios prejuízos à produtividade. Devido ao fato de o gerenciamento de concorrência *offline* ser difícil de testar, tais erros poderiam não ser detectados por todos os seus conjuntos de testes.

Uma solução é não permitir que os desenvolvedores cometam esse erro. As tarefas de bloqueio que não podem ser negligenciadas não deveriam ser manipuladas explicitamente pelos desenvolvedores, mas implicitamente pela aplicação. O fato de que a maioria das aplicações corporativas faz uso de alguma combinação de *framework*, *Camada Supertipo* (444) e geração de código nos fornece uma vasta oportunidade para facilitar o *Bloqueio Implícito*.

Como Funciona

Implementar *Bloqueio Implícito* é uma questão de fatorar seu código de modo que qualquer mecanismo de bloqueio que *realmente não possa* ser pulado possa ser executado pelo seu *framework* da aplicação. Pela falta de uma palavra melhor usaremos

"*framework*" significando uma combinação de *Camadas Supertipo* (444), classes de *framework* e qualquer outro código auxiliar. Ferramentas de geração de código são uma outra via para forçar o bloqueio apropriado. Percebo que isso de forma alguma significa uma idéia inovadora. É muito provável que você siga esse caminho, assim que tiver codificado o mesmo mecanismo de bloqueio algumas vezes pela sua aplicação. Ainda assim, tenho visto isso não muito bem feito com freqüência suficiente para que ele mereça uma breve olhada.

O primeiro passo é montar uma lista de quais tarefas são obrigatórias para que uma transação de negócio trabalhe dentro da sua estratégia de bloqueio. Para o *Bloqueio Offline Otimista* (392), essa lista conterá itens como armazenar um contador de versão para cada registro, incluindo a versão no critério de atualização SQL, e armazenar uma versão incrementada ao alterar o registro. A lista do *Bloqueio Offline Pessimista* (401) incluirá itens junto com as linhas para obter qualquer bloqueio necessário para carregar uma certa quantidade de dados – normalmente o bloqueio exclusivo de leitura ou a parte de leitura do bloqueio de leitura/gravação – e liberar todos os bloqueios quando a transação de negócio ou sessão for completada.

Perceba que a lista do *Bloqueio Offline Pessimista* (401) não inclui a obtenção de qualquer bloqueio necessário apenas para editar uma determinada quantidade de dados – ou seja, bloqueio exclusivo de gravação e a parte de gravação do bloqueio de leitura/gravação. Sim, esses são obrigatórios se sua transação de negócio quiser editar os dados, mas obtê-los implicitamente apresenta, se os bloqueios estiverem indisponíveis, algumas dificuldades. Primeiro, os únicos pontos nos quais poderíamos obter implicitamente um bloqueio de gravação, como no registro de um objeto sujo dentro de uma *Unidade de Trabalho* (187), não nos oferecem nenhuma garantia de que, se os bloqueios estiverem disponíveis, a transação irá abortar assim que o usuário comece a trabalhar. A aplicação não consegue descobrir por si mesma quando é um bom momento para obter esses bloqueios. Uma transação que não falhe rapidamente, conflita com um dos objetivos de um *Bloqueio Offline Pessimista* (401) – que um usuário não tenha que executar o mesmo trabalho duas vezes.

Segundo, e também importante, é que esses tipos de bloqueio limitam muito a concorrência do sistema. Evitar um *Bloqueio Implícito* aqui nos ajuda a pensar sobre como afetamos a concorrência tirando esta questão da área técnica e colocando-a no domínio do negócio. Ainda assim, temos que nos assegurar de que os bloqueios necessários para a gravação sejam obtidos antes que as alterações sejam confirmadas. O que o seu *framework* pode fazer é assegurar que um bloqueio de gravação já tenha sido obtido antes de confirmar qualquer alteração. Não ter obtido o bloqueio até o momento da confirmação é um erro de programação, e o código deve pelo menos levantar uma falha de asserção. Aconselho pular a declaração e levantar uma exceção de concorrência aqui, já que você não quer tais erros no seu sistema de produção quando as asserções são desativadas.

Uma palavra de precaução sobre o uso de *Bloqueio Implícito*. Embora ele permita aos desenvolvedores ignorar muito dos mecanismos de bloqueio, não lhes permite ignorar as conseqüências. Por exemplo, se os desenvolvedores estiverem usando *Bloqueio Implícito* com um esquema de bloqueio pessimista que espere pelos bloqueios, eles ainda precisam pensar sobre as possibilidades de *deadlock*. O perigo do *Bloqueio Implícito* é que as transações de negócio podem falhar de maneiras inesperadas assim que os desenvolvedores pararem de pensar sobre bloqueio.

Fazer o bloqueio funcionar é uma questão de determinar o melhor modo de fazer com que seu *framework* execute implicitamente os mecanismos de bloqueio. Veja

em *Bloqueio Offline Otimista* (392) exemplos de manipulação implícita desse tipo de bloqueio. As possibilidades de uma implementação de *Bloqueio Implícito* de qualidade são numerosas demais para demonstrá-las todas aqui.

Quando Usá-lo

O *Bloqueio Implícito* deve ser usado em todas as aplicações não-triviais que não tenham o conceito de *framework*. O risco de um único bloqueio esquecido é grande demais.

Exemplo: Bloqueio *Offline* Pessimista (401) Implícito (Java)

Vamos considerar um sistema que usa um bloqueio exclusivo de leitura. Nossa arquitetura contém um *Modelo de Domínio* (126) e estamos usando *Mapeadores de Dados* (170) para realizar a mediação entre nossos objetos de domínio e nosso banco de dados relacional. Com o bloqueio exclusivo de leitura, o *framework* deve obter um bloqueio sobre um objeto de domínio antes de permitir a uma transação de negócio fazer qualquer coisa com ele.

Qualquer objeto de domínio usado em uma transação de negócio é localizado pelo método `buscar( )` em um mapeador. Isso é verdadeiro se a transação de negócio usar o mapeador diretamente chamando `buscar( )` ou indiretamente navegando pelo grafo de objetos. Agora é possível para nós decorar [Gang of Four] nossos mapeadores com a funcionalidade de bloqueio requerida. Escreveremos um mapeador de bloqueio que obtém um bloqueio antes de tentar encontrar um objeto.

```
interface Mapeador...

    public ObjetoDoDomínio buscar(Long id);
    public void inserir(ObjetoDoDomínio obj);
    public void atualizar(ObjetoDoDomínio obj);
    public void excluir(ObjetoDoDomínio obj);

class MapeadorDeBloqueio implements Mapeador...

    private Mapeador impl;
    public MapeadorDeBloqueio (Mapeador impl) {
        this.impl = impl;
    }
    public ObjetoDoDomínio buscar (Long id) {
        GerenciadorDeBloqueioExclusivosDeLeitura.INSTANCE.obterBloqueio(
            id, GerenciadorDeSessãoDeAplicação.lerSessão( ).lerId( ));
        return impl.buscar(id);
    }
    public void inserir (ObjetoDoDomínio obj) {
        impl.inserir(obj);
    }
    public void atualizar (ObjetoDoDomínio obj) {
        impl.atualizar(obj);
    }
    public void excluir (ObjetoDoDomínio obj) {
        impl.excluir(obj);
    }
```

Por ser muito comum procurar por um objeto mais de uma vez em uma sessão, para que o código acima funcione, o gerenciador de bloqueio deve verificar primeiro se a sessão já não tem o bloqueio antes que ele obtenha um. Se estivéssemos usando um bloqueio exclusivo de gravação em vez de um bloqueio exclusivo de leitura, escreveríamos um decorador para mapeador que verificasse a obtenção prévia de bloqueios na atualização e exclusão, em vez de realmente obter um bloqueio.

Uma das coisas boas relacionadas a decoradores é que o objeto sendo envolvido nem sabe que sua funcionalidade está sendo aumentada. Aqui podemos envolver os mapeadores no nosso registro:

```
RegistroDeMapeadorDeBloqueios implements RegistroMapeador...

    private Map mapeadores = new HashMap( );
    public void mapeadorDeRegistro(Class classe, Mapeador mapeador) {
        mapeadores.put(classe, new MapeadorDeBloqueio(mapeador));
    }
    public Mapeador lerMapeador(Class classe) {
        return (Mapeador) mapeadores.get(classe);
    }
```

Quando a transação de negócio põe suas mãos em um mapeador, pensa que está para chamar um método de atualização padrão, mas o que realmente acontece é mostrado na Figura 16.5.

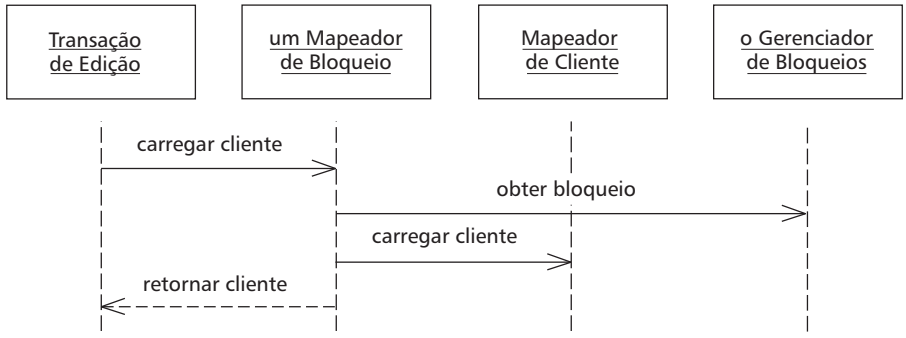


Figura 16.5 Mapeador de bloqueio.

CAPÍTULO 17

Padrões de Estado de Sessão

Estado da Sessão no Cliente (Client Session State)

Armazena o estado da sessão no cliente.

Como Funciona

Até mesmo os projetos mais orientados a servidor precisam de pelo menos um pequeno *Estado da Sessão no Cliente*, no mínimo para armazenar um identificador da sessão. Em algumas aplicações você pode considerar a colocação de todos os dados da sessão no cliente. Neste caso, o cliente envia todo o conjunto de dados de sessão em cada solicitação, e o servidor os envia de volta a cada resposta. Isso permite um servidor completamente sem estado.

Na maior parte do tempo você irá querer usar um *Objeto de Transferência de Dados* (380) para tratar a transferência de dados. O *Objeto de Transferência de Dados* (380) pode serializar a si próprio através da conexão permitindo assim que até mesmo dados complexos sejam transmitidos.

O cliente também precisa armazenar os dados. Se for uma aplicação com cliente rico, ele pode fazer isso dentro de suas próprias estruturas, como os campos na sua interface – embora eu preferisse beber Budweiser a ter de fazer isso. Um conjunto de objetos não-visuais são, muitas vezes, uma escolha melhor, tais como o próprio *Objeto de Transferência de Dados* (380) ou um modelo de domínio. De uma maneira ou de outra, isso normalmente não é um grande problema.

Com uma interface HTML, as coisas ficam um pouco mais complicadas. Existem três maneiras usuais de implementar o *Estado da Sessão no Cliente*: parâmetros URL, campos ocultos e *cookies*.

Os parâmetros URL são os mais fáceis de trabalhar, para uma quantidade pequena de dados. Essencialmente, todas as URLs, em qualquer página de resposta, trazem o estado da sessão como um parâmetro. O limite claro desta técnica é que o tamanho máximo de uma URL é limitado, mas se você tiver apenas alguns poucos itens de dados ela funciona bem, sendo por isso uma escolha popular para algo como uma identidade de sessão. Algumas plataformas fazem automaticamente a reescrita da URL para adicionar uma identidade de sessão. A alteração na URL pode ser um problema com *bookmarks*, de modo que este é um argumento contra o uso de parâmetros URL para *sites* de consumidores.

Um campo oculto é um campo enviado para o navegador que não é mostrado na página Web. Você o obtém com uma *tag* na forma `<INPUT type = "hidden">`. Para fazer um campo oculto funcionar, você, quando constrói uma resposta, serializa nele o estado da sessão e o lê de volta em cada solicitação. Você precisará de um formato para colocar os dados no campo oculto. A XML é uma escolha óbvia, mas é claro que ela é bastante prolixa. Você pode também codificar os dados com algum esquema de codificação baseado em texto. Lembre-se de que um campo oculto é oculto apenas para a página exibida. Qualquer um pode ver os dados olhando o código fonte da página.

Tome cuidado com um *site* confuso que tenha páginas velhas ou que tenham sido consertadas. Você pode perder todos os dados da sessão se navegar até essas páginas.

A última, e algumas vezes controversa, escolha são os *cookies*, os quais são enviados de um lado para o outro automaticamente. Assim como com um campo oculto, você pode usar um *cookie* serializando nele o estado da sessão. Você fica limitado

ao tamanho máximo que o *cookie* pode ter. Além disso, muitas pessoas não gostam de *cookies* e os desativam. Se eles fizerem isso, seu *site* irá parar de funcionar. No entanto, atualmente, cada vez mais *sites* são dependentes de *cookies*, de modo que esse problema deverá ocorrer cada vez com menos freqüência. Além disso, isso certamente não é um problema para um sistema puramente interno à empresa.

Perceba que os *cookies* não são mais seguros que quaisquer das alternativas, então tenha em mente que podem ocorrer bisbilhotices de todo tipo. Além disso, os *cookies* só funcionam dentro de um único nome de domínio, de modo que se o seu *site* estiver disperso em diferentes nomes de domínio, os *cookies* não serão movidos entre eles.

Algumas plataformas conseguem detectar se os *cookies* estão habilitados e, se não estiverem, podem usar a reescrita de URL. Isso pode tornar o *Estado da Sessão no Cliente* bastante fácil para pequenas quantidades de dados.

Quando Usá-lo

O *Estado da Sessão no Cliente* tem uma série de vantagens. Em especial, ele suporta bem objetos servidores sem estado com máxima clusterização e resiliência a falhas. É claro que, se o cliente falhar, tudo está perdido, mas isso, muitas vezes, já é esperado pelo usuário.

Os argumentos contrários ao *Estado da Sessão no Cliente* crescem muito com a quantidade de dados envolvida. Com apenas alguns poucos campos, tudo funciona muito bem. Com grandes quantidades de dados, as questões sobre onde armazenar os dados e o custo do tempo de transferência em cada solicitação tornam-se proibitivos. Isso é especialmente verdadeiro se sua constelação incluir um cliente http.

Existe também a questão da segurança. Qualquer dado enviado para o cliente é vulnerável a ser visto e alterado. A criptografia é a única maneira de evitar isso, mas criptografar e decriptografar a cada solicitação é uma sobrecarga para o desempenho. Sem criptografia, você tem que estar seguro de que não está enviando algo que preferiria esconder de olhos intrometidos. Os dedos podem também bisbilhotar, então, não pressuponha que o que foi enviado é o mesmo que foi recebido. Qualquer dado voltando precisará ser completamente revalidado.

Quase sempre, você tem que usar o *Estado da Sessão no Cliente* para identificação de sessão. Felizmente, isso deve ser apenas um número, o qual não será um fardo para nenhum dos esquemas acima. Você deve ainda se preocupar com o roubo da sessão, que é o que acontece quando um usuário malicioso altera a sua identificação de sessão para tentar entrar na sessão de outra pessoa. A maioria das plataformas, para reduzir esse risco, incorpora uma identificação de sessão randômica. Se este não for o caso, gere uma identificação simples de sessão por meio de um *hash*.

Estado da Sessão no Servidor (Server Session State)

Mantém o estado da sessão serializado no servidor.

Como Funciona

Na forma mais simples deste padrão, um objeto de sessão é armazenado na memória em um servidor de aplicação. Você pode ter algum tipo de mapa na memória que armazene estes objetos de sessão usando como chave uma identificação de sessão. Tudo o que o cliente precisa fazer é fornecer essa identificação e, então, o objeto de sessão pode ser trazido do mapa para processar a solicitação.

Este cenário básico pressupõe, é claro, que o servidor de aplicação tenha memória suficiente para executar esta tarefa. Ele pressupõe também que haja apenas um servidor de aplicação – isto é, que não se estejam utilizando *clusters* – e que, se o servidor de aplicação falhar, seja apropriado que a sessão seja abandonada e todo o trabalho feito até o momento seja perdido.

Para muitas aplicações este conjunto de suposições não é de fato um problema. Entretanto, para outras, pode ser problemático. Existem formas de lidar com os casos em que as suposições não sejam válidas, e estes introduzem variações comuns que acrescentam complexidade a um padrão essencialmente simples.

A primeira questão é a manipulação dos recursos de memória mantidos pelos objetos de sessão. De fato, esta é a objeção mais comum ao *Estado da Sessão no Servidor*. A resposta, é claro, é não manter recursos na memória, mas, em vez disso, serializar todo o estado da sessão em um memento [Gang of Four] para um armazenamento persistente. Isso apresenta duas questões: em que formato você persiste o *Estado da Sessão no Servidor*, e onde você o persiste?

O formato a ser usado é normalmente o formato mais simples possível, já que a marca do *Estado da Sessão no Servidor* é sua simplicidade de programação. Diversas plataformas fornecem um mecanismo de serialização binária simples que permitem facilmente serializar um grafo de objetos. Outro caminho é serializar em algum outro formato, tal como texto – tão moderno quanto um arquivo XML.

O formato binário é normalmente mais fácil, uma vez que requer pouca programação, enquanto que a forma textual requer, geralmente, ao menos um pouco de código. Serializações binárias também requerem menos espaço em disco. Ainda que o espaço total em disco raramente seja um problema, grandes grafos de objetos serializados demorarão mais tempo para serem ativados na memória.

Existem dois problemas recorrentes com a serialização binária. Em primeiro lugar, a forma serializada não é legível por um ser humano – o que é um problema se as pessoas quiserem lê-lo. Segundo, pode haver problemas com o controle de versões. Se você modificar uma classe, digamos, acrescentando um campo após tê-la serializado, poderá não conseguir lê-la novamente. É claro que não é provável que muitas sessões convivam com uma atualização do *software* do servidor, a menos que este seja um servidor 24/7 no qual você pode ter um *cluster* de máquinas rodando, algumas atualizadas e outras não.

Isso nos remete à questão de onde armazenar o *Estado da Sessão no Servidor*. Uma possibilidade óbvia é armazená-lo no próprio servidor de aplicação, no sistema de arquivos ou em um banco de dados local. Este é um caminho simples, mas pode não suportar a utilização eficiente de *clusters* ou a recuperação automática de falhas. Para suportar estas características, o *Estado da Sessão no Servidor* precisa estar em algum lugar acessível de modo geral, tal como um servidor compartilhado. Isso irá su-

portar a utilização de *clusters* e a recuperação automática de falhas ao custo de um tempo maior necessário para ativar o servidor – ainda que a utilização de uma memória cache possa diminuir bastante esse custo.

Esta linha de raciocínio pode levar, ironicamente, ao armazenamento do *Estado da Sessão no Servidor* serializado no banco de dados, usando uma tabela de sessões indexada pelo ID da sessão. Essa tabela iria requerer um *LOB Serializado* (264) para armazenar o *Estado da Sessão no Servidor* serializado. Quando se trata de manipular objetos grandes, o desempenho do banco de dados pode variar muito, então, os aspectos de desempenho desta alternativa são bastante dependentes do banco de dados utilizado.

Neste ponto estamos exatamente na fronteira entre o *Estado da Sessão no Servidor* e o *Estado da Sessão no Banco de Dados* (432). Esta fronteira é completamente arbitrária, mas tracei a linha no ponto onde você converte os dados no *Estado da Sessão no Servidor* para um formato tabular.

Se você estiver armazenando o *Estado da Sessão no Servidor* em um banco de dados, você terá de se preocupar com o tratamento de sessões que desaparecem, especialmente em uma aplicação voltada aos consumidores. Um caminho é ter um *daemon* que procure sessões antigas e as exclua, mas isso pode levar a uma grande contenção na tabela de sessões. Kai Yu me falou sobre uma abordagem que ele usava com sucesso: dividir a tabela de sessões em doze segmentos no banco de dados e, a cada duas horas, rodar os segmentos, apagando tudo no segmento mais antigo e, então, direcionando todas as inserções para ele. Embora signifique que qualquer sessão que permanecer ativa por vinte e quatro horas será sumariamente jogada fora, isto é suficientemente raro para não se constituir em um problema.

Todas essas variações custam mais e mais trabalho para serem implementadas, mas a boa notícia é que os servidores de aplicação vêm, cada vez mais, suportando estas capacidades automaticamente. Assim, pode bem ser que estas passem a ser preocupações exclusivamente dos vendedores de servidores de aplicação.

Implementação Java

As duas técnicas mais comuns para o *Estado da Sessão no Servidor* são usar a sessão http e usar um *session bean* com estado. A sessão http é o caminho mais simples e faz com que os dados da sessão sejam armazenados pelo servidor Web. Na maioria dos casos, isto leva à afinidade com o servidor, e não é possível lidar com a recuperação automática de falhas. Alguns vendedores vêm implementando uma sessão http compartilhada que permite armazenar os dados da sessão http em um banco de dados disponível para todos os servidores de aplicação. (Você também pode fazer isso manualmente, é claro.)

O outro caminho usual é através de um *session bean* com estado, o que requer um servidor EJB. O contêiner EJB trata a persistência e a passivação*, de modo que isso o torna muito fácil de programar. A principal desvantagem é que a especificação não exige que o servidor de aplicação evite a afinidade com um servidor. Alguns servidores de aplicação, no entanto, fornecem esta capacidade. Um deles, o WebSphere da IBM, pode serializar um *session bean* com estado em um BLOB no DB2, o que permite que múltiplos servidores de aplicação acessem este estado.

Várias pessoas dizem que, uma vez que os *session beans* sem estado têm um desempenho melhor, você deveria *sempre* usá-los em vez de usar *beans* com estado.

* N. de R. T.: A passivação é o ato de separar a instância de um *bean* com estado de seu objeto EJB e salvar o seu estado.

Francamente, isso é conversa fiada. Primeiro, faça um teste de carga com o seu ambiente para ver se a diferença de velocidade entre *beans* com estado e *beans* sem estado faz alguma diferença para a sua aplicação. A ThoughtWorks tem feito testes de carga em aplicações com algumas centenas de usuários concorrentes e não encontrou nenhum problema de desempenho devido à utilização de *beans* com estado para tal nível de carga de usuário. Se o ganho de desempenho não é significativo para a sua aplicação, e *beans* com estado são mais fáceis, então você deveria usá-los. Há outras razões para ser cauteloso com *beans* com estado – a recuperação automática de falhas pode ser mais problemática dependendo do seu vendedor, mas a diferença de desempenho só aparece sob carga pesada.

Uma alternativa é usar um *entity bean*. Em geral, não tenho dado muita atenção a *entity beans*, mas você pode usar um para armazenar um *LOB Serializado* (264) de dados de sessão. Isso é bastante simples e é pouco provável que dê margem à ocorrência de alguns dos muitos problemas que geralmente envolvem a utilização de *entity beans*.

Implementação .NET

O *Estado da Sessão no Servidor* é fácil de implementar com a capacidade embutida do estado de sessão. Por meio de *default*.NET, armazena os dados da sessão no próprio processo servidor. Você pode ainda acomodar o armazenamento usando um serviço de estado, o qual pode residir na máquina local ou em qualquer outra máquina na rede. Com um serviço de estado separado, você pode reinicializar o servidor Web e ainda assim reter o estado da sessão. Você faz a escolha entre o estado de sessão embutido no processo e o serviço de estado em um arquivo de configuração, de modo que você não tem que alterar a aplicação.

Quando Usá-lo

A grande atração do *Estado da Sessão no Servidor* é a sua simplicidade. Em vários casos você não tem que executar nenhuma programação para fazê-lo funcionar. Se ele vai ou não resolver seu problema, depende de você conseguir resolver a implementação em memória ou, em caso contrário, do nível de suporte que sua plataforma servidora de aplicação lhe dá.

Mesmo sem isso você bem pode descobrir que o esforço de que precisa é pequeno. Serializar um BLOB para uma tabela de banco de dados pode acabar exigindo muito menos esforço do que converter os objetos servidores para o formato tabular.

Onde o esforço de programação entra em cena é na manutenção da sessão, especialmente se você tiver que prover seu próprio suporte para permitir a utilização de *clusters* e a recuperação automática de falhas. Pode ocorrer de isso ser mais problemático do que as outras opções, especialmente se você não tiver muitos dados de sessão para tratar, ou se seus dados de sessão podem ser facilmente convertidos para o formato tabular.

Estado da Sessão no Banco de Dados (Database Session State)

Armazena dados de sessão como dados gravados no banco de dados.

Como Funciona

Quando uma chamada sai do cliente para o servidor, o objeto servidor primeiro traz do banco de dados os dados solicitados pela requisição. Em seguida, executa o trabalho necessário e salva novamente todos os dados solicitados no banco de dados.

Para trazer as informações do banco de dados, o objeto servidor precisará de algumas informações sobre a sessão, o que requer que, ao menos, um número identificador da sessão seja armazenado no cliente. Normalmente, contudo, esta informação não é nada mais do que o conjunto apropriado de chaves necessárias para encontrar a quantidade apropriada de dados no banco de dados.

Os dados envolvidos são geralmente uma mistura de dados de sessão, que são locais apenas para a interação corrente, e dados gravados, que são relevantes para todas as interações.

Um dos aspectos-chave a considerar aqui é o fato de que os dados de sessão são geralmente considerados locais à sessão e não deveriam afetar outras partes do sistema até que a sessão como um todo seja gravada no banco de dados. Assim, se você estiver trabalhando em um pedido em uma sessão e quiser gravar seu estado intermediário no banco de dados, normalmente precisa tratá-lo de forma diferente de um pedido que é confirmado ao final da sessão. Isso se deve ao fato de que você não quer que pedidos pendentes apareçam em consultas ao banco de dados, buscando coisas como disponibilidade de livros e receita diária.

Então, como você separa os dados de sessão? Adicionar um campo a cada linha do banco de dados que possa ter dados de sessão é um caminho. A maneira mais simples disso requer apenas um campo booleano `estáPendente`. No entanto, uma maneira melhor é armazenar uma identificação de sessão como um campo pendente, o que torna muito mais fácil encontrar todos os dados de uma determinada sessão. Todas as consultas que querem apenas dados permanentes precisam ser modificadas com uma cláusula `IDdaSessão is not NULL`, ou precisam de uma visão que filtre esses dados.

O uso de um campo contendo a identificação da sessão é uma solução muito invasiva, porque todas as aplicações que acessam o banco de dados precisam conhecer o significado do campo para evitar obter dados de sessão. As Visões, algumas vezes resolvem esse problema, mas elas freqüentemente impõem seus próprios custos.

Uma segunda alternativa é separar conjuntos de tabelas pendentes. Assim, se você já tiver tabelas para pedidos e para itens de pedidos no seu banco de dados, você simplesmente adicionaria tabelas para pedidos pendentes e itens de pedidos pendentes. Dados pendentes de sessão seriam armazenados nas tabelas pendentes. Quando eles se tornarem registros permanentes, você os grava nas tabelas reais. Isso remove muita da invasividade. No entanto, você precisará acrescentar ao seu código de mapeamento de banco de dados a lógica apropriada para a seleção de tabelas, o que certamente adicionará algumas complicações.

Freqüentemente, os dados permanentes terão regras de integridade que não se aplicam a dados pendentes. Neste caso, as tabelas pendentes lhe permitem abrir mão das regras, quando não as quiser, mas forçar o seu uso, quando isso for conveniente. Regras de validação geralmente também não são aplicadas ao salvar dados penden-

tes. Você pode se deparar com diferentes regras de validação, dependendo de onde você estiver na sessão, mas isso geralmente aparece na lógica do objeto servidor.

Se você usar tabelas pendentes, elas devem ser clones exatos das tabelas reais. Dessa forma, você pode manter sua lógica de mapeamento tão parecida quanto possível. Use o mesmo nome de campo nas duas tabelas, mas adicione um campo de identificação da sessão nas tabelas pendentes de modo que você possa encontrar facilmente todos os dados de uma sessão.

Você precisará de um mecanismo para limpar os dados de sessão se uma sessão for cancelada ou abandonada. Usando um identificador de sessão, você pode encontrar todos os dados relacionados à sessão e apagá-los. Se os usuários abandonarem a sessão sem lhe informar, você irá precisar de algum tipo de mecanismo de *timeout* para lidar com essa situação. Um *daemon* que rode a cada poucos minutos pode procurar por dados de sessões antigos. Isso requer uma tabela no banco de dados que registre a hora da última interação do usuário com a sessão.

As atualizações tornam o processo de desfazer uma transação (*rollback*) muito mais complicado. Se você atualiza um pedido existente em uma sessão que permite o *rollback* de toda a sessão, como você executa esse *rollback*? Uma opção é não permitir o cancelamento de uma sessão assim. Quaisquer atualizações em registros permanentes existentes tornam-se parte dos dados permanentes ao final da solicitação. Isto é simples e freqüentemente se encaixa na visão de mundo dos usuários. A alternativa é complicada, quer você use campos pendentes, quer use tabelas pendentes. É fácil copiar todos os dados que podem ser modificados para tabelas pendentes, lá modificá-los e gravá-los de volta nas tabelas permanentes ao final da sessão. Você pode fazer isso com um campo pendente, mas somente se o identificador da sessão for parte da chave. Dessa forma, você pode manter juntos, na mesma tabela e ao mesmo tempo, identificadores de sessão novos e antigos, o que pode ser bastante confuso.

Se você for usar tabelas pendentes separadas que são lidas apenas por objetos que tratam a sessão, então pode haver uma motivação pequena para colocar os dados em uma forma tabular. É melhor usar um *LOB Serializado* (264). Neste ponto, cruzamos a fronteira para o *Estado da Sessão no Servidor* (429).

Você pode evitar toda essa confusão de dados pendentes simplesmente não tendo nenhum dado pendente. Isto é, você projeta o seu sistema de modo que todos os dados sejam considerados dados permanentes. Isso, é claro, nem sempre é possível, e, ainda que seja, pode ser tão complicado que os projetistas fariam melhor se pensassem sobre dados pendentes explícitos. Ainda assim, se você tiver esta opção, isso torna muito mais fácil trabalhar com o *Estado da Sessão no Banco de Dados*.

Quando Usá-lo

O *Estado da Sessão no Banco de Dados* é uma das alternativas para tratar o estado da sessão. Ele deve ser comparado ao *Estado da Sessão no Servidor* (429) e ao *Estado da Sessão no Cliente* (427).

O primeiro aspecto a considerar neste padrão é o desempenho. Você ganhará usando objetos sem estado no servidor, possibilitando assim a utilização da técnica de *pooling* e a fácil utilização de *clusters*. No entanto, você pagará um preço pelo tempo necessário para trazer os dados do banco de dados e para o banco de dados a cada solicitação. Você pode reduzir este custo colocando o objeto servidor em memória cache, de modo que você não tenha que ler os dados do banco de dados quando estes estiverem no cache. No entanto, você ainda terá de pagar pelos custos de escrita.

A segunda questão importante diz respeito ao esforço de programação, a maior parte do qual está relacionada ao tratamento do estado da sessão. Se você não tiver estado de sessão algum e puder gravar todos os seus dados como registros permanentes a cada solicitação, este padrão é uma escolha óbvia porque você não perde nada, quer seja em esforço, quer seja em desempenho (se você colocar em cache seus objetos servidores).

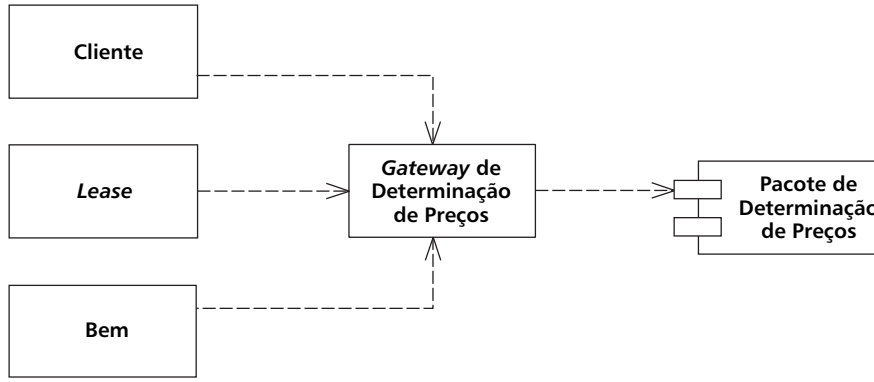
Em uma escolha entre o *Estado da Sessão no Banco de Dados* e o *Estado da Sessão no Servidor* (429), a questão mais importante pode ser o quão fácil é suportar a adoção de *clusters* e a recuperação automática de falhas com o *Estado da Sessão no Servidor* (429) no seu servidor de aplicação. A utilização de *clusters* e a recuperação automática de falhas com o *Estado da Sessão no Banco de Dados*, são geralmente mais claras, pelo menos com as soluções normais.

CAPÍTULO 18

Padrões Básicos

Gateway

Um objeto que encapsula o acesso a um sistema ou recurso externo.



O *software* interessante raramente vive isolado. Até mesmo o sistema orientado a objetos mais puro muitas vezes tem que lidar com coisas que não são objetos, como tabelas de bancos de dados relacionais, transações CICS e estruturas de dados XML.

Ao acessar recursos externos como esses, você geralmente obterá APIs para eles. Entretanto, estas APIs naturalmente serão um pouco complicadas porque elas levam em consideração a natureza do recurso. Qualquer pessoa que precise entender um recurso precisa entender sua API – sejam JDBC e SQL para bancos de dados relacionais ou W3C ou JDOM para XML. Isso não apenas torna o *software* mais difícil de entender, como também muito mais difícil de alterar se você precisar trocar alguns dados de um banco de dados relacional para uma mensagem XML em algum ponto, no futuro.

A resposta é tão comum que mal vale a pena dizer. Encapsule todo o código especial de API em uma classe cuja interface se pareça com um objeto normal. Os outros objetos acessam o recurso através deste *Gateway*, que traduz as chamadas simples de métodos na apropriada API especializada.

Como Funciona

Na verdade, este é um padrão muito simples de encapsulamento. Pegue o recurso externo. O que a aplicação precisa fazer com ele? Crie uma API simples para seu uso e use o *Gateway* para traduzir para a fonte externa.

Um dos usos-chave de uma *Gateway* é como um bom ponto no qual aplicar um *Stub de Serviço* (473). Você pode muitas vezes alterar o projeto do *Gateway* para tornar mais fácil a aplicação de um *Stub de Serviço* (473). Não tenha medo de fazer isso – *Stubs de Serviço* (473) bem-colocados podem tornar um sistema muito mais fácil de testar e, assim, muito mais fácil de escrever.

Mantenha um *Gateway* tão simples quanto possível. Enfoque os papéis básicos da adaptação do serviço externo e do fornecimento de um bom ponto para *stub*. O *Gateway* deve ser tão mínimo quanto possível e ainda assim capaz de lidar com essas tarefas. Qualquer lógica mais complexa deve ficar nos clientes do *Gateway*.

Muitas vezes, é uma boa idéia usar geração de código para criar *Gateways*. Definindo a estrutura do recurso externo, você pode gerar uma classe *Gateway* para encapsulá-lo. Você poderia usar metadados relacionais para criar uma classe que encapsule uma tabela relacional, ou um esquema XML ou DTD para gerar código para um *Gateway* para XML. Os *Gateways* resultantes são burros, mas resolvem o problema. Outros objetos podem executar manipulações mais complicadas.

Às vezes, uma boa estratégia é criar um *Gateway* em termos de mais de um objeto. O formato óbvio é usar dois objetos: um frontal e um na retarguarda (*front end, back end*). O da retaguarda age como uma camada mínima para o recurso externo e absolutamente não simplifica a API do recurso. O frontal então transforma essa API complicada em uma mais conveniente para sua aplicação usar. Esta abordagem é boa se o encapsulamento do serviço externo e a adaptação às suas necessidades forem razoavelmente complicados, porque cada responsabilidade é manipulada por uma única classe. De modo inverso, se o encapsulamento do serviço externo for simples, uma classe pode lidar com ele e com qualquer adaptação que for necessária.

Quando Usá-lo

Você deve considerar um *Gateway* toda vez que tiver uma interface complicada para algo que parece externo. Em vez de deixar a complicação se espalhar pelo sistema inteiro, use um *Gateway* para contê-la. Quase nunca há aspectos negativos na criação do *Gateway*, e o código, em algum outro lugar do sistema, torna-se muito mais fácil de ler.

Os *Gateways* geralmente tornam um sistema mais fácil de testar, dando a você um ponto claro no qual distribuir *Stubs de Serviço* (473). Mesmo se a interface do sistema externo for boa, um *Gateway* é útil como um primeiro movimento na aplicação de um *Stub de Serviço* (473).

Um benefício claro do *Gateway* é que ele também torna mais fácil para você trocar algum tipo de recurso por outro. Qualquer alteração nos recursos significa que você só tem que alterar a classe *Gateway* – a alteração não se propaga pelo resto do sistema. O *Gateway* é uma forma simples e poderosa de variação protegida. Em muitos casos argumentar sobre esta flexibilidade é o foco do debate sobre o uso do *Gateway*. Todavia, não se esqueça de que, mesmo que você não pense que o recurso vai ser alterado, pode se beneficiar da simplicidade e da testabilidade que o *Gateway* lhe dá.

Quando você tem alguns subsistemas como este, outra escolha para desacoplá-los é um *Mapeador* (442). Contudo, o *Mapeador* (442) é mais complicado do que o *Gateway*. Como conseqüência, uso *Gateway* para a maioria dos meus acessos a recursos externos.

Devo admitir que relutei um pouco entre fazer deste um novo padrão, em vez de referenciar padrões existentes como Fachada e Adaptador [Gang of Four]. Decidi separá-lo desses outros padrões porque acho que há uma distinção útil a ser feita.

- Embora a *Fachada* simplifique uma API mais complexa, isso geralmente é feito por quem escreve o serviço de uso geral. Um *Gateway* é escrito pelo cliente para seu uso particular. Além disso, uma Fachada sempre significa uma interface diferente para o que ela está cobrindo, enquanto que um *Gateway* pode copiar inteiramente a fachada envolvida, sendo usada para substituição ou testes.

- Os *Adaptadores* alteram a interface de uma implementação para que se ajuste a outra interface com a qual você precisa trabalhar. Com o *Gateway*, geralmente não há uma interface existente, embora você possa usar um adaptador para mapear uma implementação para a interface de um *Gateway*. Neste caso, o adaptador é parte da implementação do *Gateway*.
- O *Mediador* geralmente separa diversos objetos de modo que eles não conheçam uns aos outros, mas conheçam o mediador. Com um *Gateway*, geralmente, há apenas dois objetos envolvidos, e o recurso que está sendo envolvido não conhece o *Gateway*.

Exemplo: Um *Gateway* para um Serviço de Mensagens Proprietário (Java)

Estava falando sobre este padrão com meu colega, Mike Retting, e ele me descreveu a forma como costuma usá-lo em interfaces com *software* do tipo Enterprise Application Integration (EAI). Decidimos que isso seria uma boa inspiração para um exemplo de *Gateway*.

Para manter as coisas no nível costumeiro de enorme simplicidade, criaremos um *gateway* para uma interface que apenas envia uma mensagem usando o serviço de mensagens. A interface é apenas um único método.

```
int enviar (String tipoDaMensagem, Object[ ] parâmetros);
```

O primeiro parâmetro é uma *string* indicando o tipo da mensagem. O segundo são os parâmetros da mensagem. O sistema de mensagens lhe permite enviar qualquer tipo de mensagem, então ele precisa de uma interface genérica com esta. Quando você configura o sistema de mensagens, especifica os tipos de mensagens que o sistema enviará e o número e tipo de parâmetros para eles. Assim, poderíamos configurar a mensagem de confirmação com a *string* `"CNFRM"` e ter parâmetros para um número de identificação como uma *string*, uma quantidade inteira e uma *string* para o código do registrador. O sistema de mensagens verifica os tipos dos parâmetros para nós e gera um erro se enviarmos uma mensagem errada ou a mensagem certa com os parâmetros errados.

Isso é louvável, e provê a necessária flexibilidade, mas a interface genérica é complicada de usar porque não é explícita. Você não pode perceber olhando a interface quais são os tipos legais de mensagens ou quais parâmetros são necessários para um determinado tipo de mensagem. O que precisamos em vez disso é de uma interface com métodos como este:

```
public void enviarConfirmação (String IDdoPedido, int quantia, String símbolo);
```

Desta maneira, se quisermos que um objeto do domínio envie uma mensagem, ele pode fazer isso desta forma:

```
class Pedido...

    public void confirmar ( ) {
        if (éValido( ))Ambiente.lerGatewayDeMensagens( ).enviarConfirmação(id, quantia, símbolo);
}
```

Aqui o nome do método nos diz qual mensagem ele está enviando e os parâmetros são tipados e recebem nomes. Este é um método muito mais fácil de chamar do

que o método genérico. É papel do *gateway* criar uma interface mais conveniente. Isso significa, porém, que cada vez que adicionarmos ou alterarmos um tipo de mensagem no sistema de mensagens, precisamos alterar a classe *gateway*, mas teríamos que alterar o código da chamada de qualquer maneira. Pelo menos, dessa forma, o compilador pode nos ajudar a encontrar os clientes e a verificar erros.

Há outro problema. Quando temos um erro com esta interface, ela nos informa retornando um código de erro. Um zero indica sucesso, qualquer outra coisa indica uma falha, e números diferentes indicam diferentes erros. Esta é uma maneira natural de um programador C trabalhar, mas não é a maneira pela qual Java faz as coisas. Em Java, você levanta uma exceção para indicar um erro, então, os métodos do *Gateway* devem levantar exceções em vez de retornar códigos de erros.

A faixa completa de erros possíveis é algo que iremos naturalmente ignorar. Focarei apenas dois: enviar uma mensagem com um tipo desconhecido de mensagem e enviar uma mensagem em que um dos parâmetros é nulo. Os códigos de retorno são definidos na interface do sistema de mensagens.

```
public static final int PARÂMETRO_NULO = -1;
public static final int TIPO_DESCONHECIDO_DE_MENSAGEM = -2;
public static final int SUCESSO = 0;
```

Os dois erros têm uma diferença significativa. O erro referente ao tipo desconhecido de mensagem indica um erro na classe *gateway*. Já que qualquer cliente está apenas chamando um método completamente explícito, este erro nunca deveria ser gerado. Entretanto, eles poderiam passar um nulo e assim ver o erro do parâmetro nulo. Este erro não é uma exceção verificada já que indica um erro do programador – não é algo para o qual você escreveria um manipulador específico. O *gateway*, em verdade, poderia ele mesmo verificar os nulos, mas se o sistema de mensagens for levantar o mesmo erro, isso provavelmente não valerá a pena.

Por essas razões, o *gateway* tem que traduzir da interface explícita para a interface genérica e traduzir os códigos de retorno em exceções.

```
class GatewayDeMensagens...

    protected static final String CONFIRMAR = "CNFRM";
    private RemetenteDeMensagens remetente;
    public void enviarConfirmação (String IDdoPedido, int quantia, String símbolo) {
        Object [ ] parâmetros = new Object [ ] { IDdoPedido, new Integer(quantia), símbolo};
        enviar (CONFIRMAR, parâmetros);
    }
    private void enviar(String mensagem, Object [ ] parâmetros) {
        int códigoDeRetorno = executarEnvio(mensagem, parâmetros);
        if (códigoDeRetorno == RemetenteDeMensagens.PARÂMETRO_NULO)
            throw new NullPointerException ("Parâmetro nulo passado para tipo de mensagem: " + mensagem);
        if (códigoDeRetorno != RemetenteDeMensagens.SUCESSO)
            throw new IllegalStateException (
                    "Erro inesperado do sistema de mensagens #: " + códigoDeRetorno)
    }
    protected int executarEnvio (String mensagem, Object [ ] parâmetros) {
        Assert.notNull(remetente);
        return remetente.enviar(mensagem, parâmetros);
    }
```

Até aqui é difícil ver o propósito do método `executarEnvio`, mas ele está lá por outro papel-chave de um *gateway* – o teste. Podemos testar objetos que usam o *gateway* sem o serviço de envio de mensagens estar presente. Para fazer isso, precisamos criar um *Stub de Serviço* (473). Neste caso, o *stub* do *gateway* é uma subclasse do *gateway* real e sobrescreve o método `executarEnvio`.

```
class StubDoGatewayDeMensagens...

    protected int executarEnvio (String tipoDaMensagem, Object [ ] parâmetros) {
        int códigoDeRetorno = éMensagemVálida(tipoDaMensagem, parâmetros);
        if (códigoDeRetorno == RemetenteDeMensagens.SUCESSO) {
            mensagensEnviadas++;
        }
        return códigoDeRetorno;
    }
    private int éMensagemVálida (String tipoDaMensagem, Object [ ] parâmetros) {
        if (todasAsMensagensDevemFalhar) return -999;
        if (! tiposVálidosDeMensagens( ).contains(tipoDaMensagem))
            return RemetenteDeMensagens.TIPO_DESCONHECIDO_DE_MENSAGEM;
        for (int i = 0; i < parâmetros.length; i++) {
            Object parâmetro = parâmetros[i];
            if (parâmetro == null) {
                return RemetenteDeMensagens.PARÂMETRO_NULO;
            }
        }
        return RemetenteDeMensagens.SUCESSO;
    }
    public static List tiposVálidosDeMensagens ( ) {
        List resultado = new ArrayList( );
        resultado.add(CONFIRMAR);
        return resultado;
    }
    private boolean todasAsMensagensDevemFalhar = false;
    public void falharTodasAsMensagens( ) {
        todasAsMensagensDevemFalhar = true;
    }
    public int lerNúmeroDeMensagensEnviadas ( ) {
        return mensagensEnviadas;
    }
```

Capturar o número de mensagens enviadas é uma maneira simples de nos ajudar a testar se a porta funciona corretamente com testes como estes.

```
class TestadorDoGateway...

    public void testarEnvioDeParâmetroNulo ( ) {
        try {
            porta( ).enviarConfirmação(null, 5, "US");
            falhar("Não detectou parâmetro nulo");
        } catch (NullPointerException esperada) {
        }
        assertEquals(0, porta( ).lerNúmeroDeMensagensEnviadas( ));
    }
    private StubDoGatewayDeMensagens porta ( ) {
```

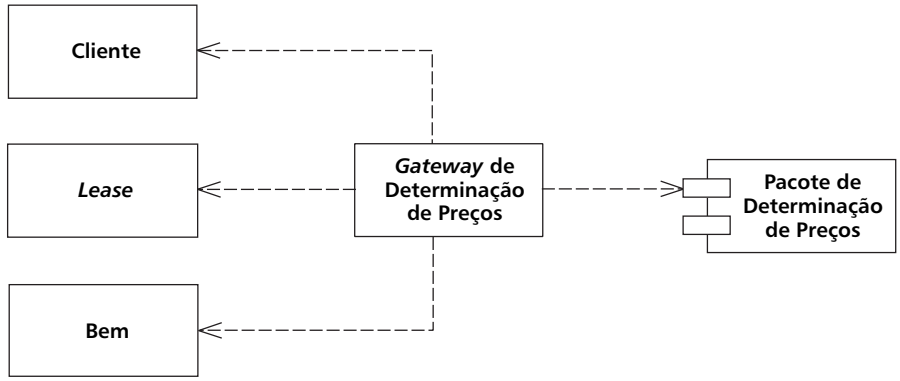
```
        return (StubDoGatewayDeMensagens) Ambiente.lerGatewayDeMensagens( );
    }
    protected void configurar( ) throws Exception {
        Ambiente.testarInicialização( );
    }
```

Você geralmente configura o *Gateway* de modo que as classes possam encontrá-lo a partir de um lugar bem conhecido. Aqui usei uma interface de ambiente estática. Você pode trocar entre o serviço real e o *stub* em tempo de configuração usando um *Plugin* (465), ou pode fazer as rotinas de configuração de teste inicializarem o ambiente para usar o *Stub de Serviço* (473).

Neste caso, usei uma subclasse do *gateway* para servir de *stub* para o serviço de mensagens. Outro caminho é criar uma subclasse do próprio serviço ou reimplementá-lo. Para testar, você conecta o *gateway* no *Stub de Serviço* (473) remetente. Isso funciona se reimplementar o serviço não for muito difícil. Você sempre tem a escolha de fazer um *stub* para o serviço ou para o *gateway*. Em alguns casos, é até útil fazer para os dois, usando o do *gateway* para testar clientes do *gateway* e o do serviço para testar o próprio *gateway*.

Mapeador (Mapper)

Um objeto que estabelece uma comunicação entre dois objetos independentes.



Às vezes, você precisa estabelecer comunicações entre dois subsistemas que ainda assim precisam ignorar um do outro. Isso pode acontecer porque você não pode modificá-los, ou pode, mas não quer criar dependências entre os dois ou até mesmo entre eles e o elemento isolante.

Como Funciona

Um mapeador é uma camada isolante entre subsistemas. Ele controla os detalhes da comunicação entre eles sem que nenhum desses subsistemas tenha ciência disso.

Um mapeador, muitas vezes, leva dados de uma camada para outra. Uma vez ativado para esse transporte, é bastante fácil ver como ele funciona. A parte complicada de usar um mapeador é decidir como invocá-lo, já que ele não pode ser invocado diretamente por nenhum dos subsistemas entre os quais ele está mapeando. Às vezes, um terceiro subsistema conduz o mapeamento e também invoca o mapeador. Uma alternativa é fazer do mapeador um observador [Gang of Four] de um ou outro subsistema. Dessa maneira, ele pode ser ativado escutando os eventos em um deles.

O funcionamento de um mapeador depende do tipo de camadas que ele estiver mapeando. O caso mais comum de um mapeamento de camadas com que nos deparamos é um *Mapeador de Dados* (170), então, veja lá mais detalhes sobre como um *Mapeador* é usado.

Quando Usá-lo

Basicamente, um *Mapeador* desacopla diferentes partes de um sistema. Quando você quiser fazer isso, tem que escolher entre um *Mapeador* e um *Gateway* (436). O *Gateway* (436) é de longe a escolha mais comum, porque ele é muito mais simples de usar um *Gateway* (436) do que um *Mapeador*, tanto na escrita do código quanto no seu uso posterior.

A conseqüência disso é que você deveria usar um *Mapeador* somente quando precisar se assegurar de que nenhum subsistema tem uma dependência em relação à esta interação. A única ocasião em que isso é realmente importante é quando a inte-

ração entre os subsistemas é especialmente complicada e, de alguma forma, independente do propósito principal de ambos os sistemas. Assim, em aplicações corporativas, na maior parte das vezes, encontramos um *Mapeador* usado para interações com um banco de dados, como num *Mapeador de Dados* (170).

O *Mapeador* é semelhante ao Mediador [Gang of Four] visto que ele é usado para separar elementos diferentes. Entretanto, os objetos que usam um mediador estão cientes disso, mesmo se não estiverem cientes uns dos outros. Os objetos que um *Mapeador* separa não estão nem mesmo cientes do mapeador.

Camada Supertipo (Layer Supertype)

Um tipo que atua como o supertipo para todos os tipos em sua camada.

Não é incomum que todos os objetos em uma camada tenham métodos que você não quer ter duplicados por todo o sistema. Você pode mover todo este comportamento para uma *Camada Supertipo* compartilhada.

Como Funciona

A *Camada Supertipo* é uma idéia simples que leva a um padrão bastante curto. Tudo o que você precisa é de uma superclasse para todos os objetos em uma camada – por exemplo, uma superclasse Objeto do Domínio para todos os objetos do domínio em um *Modelo de Domínio* (126). Características comuns, como o armazenamento e a manipulação de *Campos Identidade* (215), podem estar lá. De modo similar, todos os *Mapeadores de Dados* (170) na camada de mapeamento podem ter uma superclasse que conte com o fato de que todos os objetos do domínio têm uma superclasse em comum.

Se você tiver mais de um tipo de objeto em uma camada, é útil ter mais do que uma *Camada Supertipo*.

Quando Usá-la

Use uma *Camada Supertipo* quando você tiver características comuns a todos os objetos em uma camada. Eu freqüentemente faço isso automaticamente, porque uso muito características comuns.

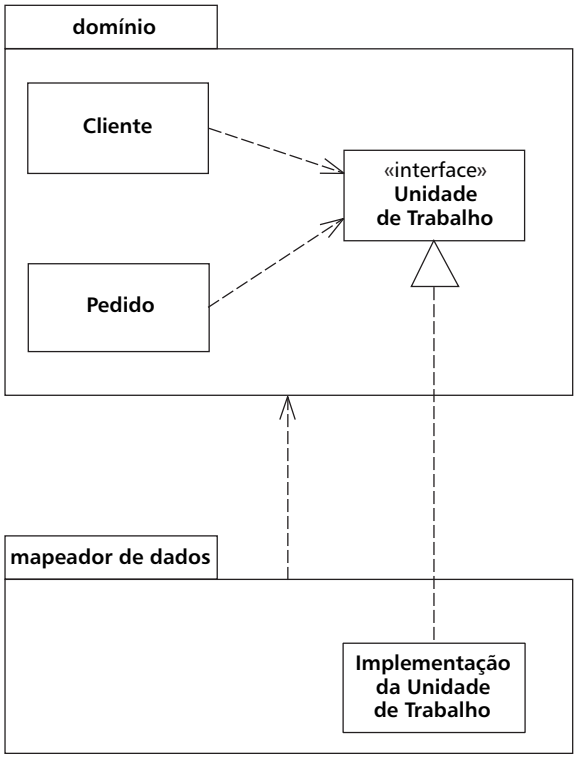
Exemplo: Objeto do Domínio (Java)

Os objetos do domínio podem ter uma superclasse comum para manipulação da identidade.

```
class ObjetoDoDomínio...

    private Long ID;
    public Long lerID( ) {
        return ID;
    }
    public void gravarID(Long ID) {
        Assert.notNull("Não posso configurar um ID nulo", ID);
        this.ID = ID;
    }
    public ObjetoDoDomínio (Long ID) {
        this.ID = ID;
    }
    public ObjetoDoDomínio( ) {
    }
```

Interface Separada (Separated Interface)

Define uma interface em um pacote separado da sua implementação.



Implementação da Unidade de Trabalho

À medida que você desenvolve um sistema, você pode melhorar a qualidade do seu projeto, reduzindo o acoplamento entre as partes dele. Uma boa maneira de fazer isso é agrupar as classes em pacotes e controlar as dependências entre eles. Você pode então seguir as regras sobre como classes em um pacote podem chamar classes em outro – por exemplo, uma que diz que classes na camada do domínio não podem chamar classes no pacote de apresentação.

Entretanto, você poderia precisar chamar métodos que contradizem a estrutura geral de dependência. Se isso for verdade, use uma *Interface Separada* para definir uma interface em um pacote, mas implementá-la em outro. Dessa maneira, um cliente que precisa da dependência com a interface pode ficar completamente ignorante a respeito da implementação. A *Interface Separada* fornece um bom ponto de conexão para o *Gateway* (436).

Como Funciona

Este padrão é muito simples de empregar. Basicamente, ele tira proveito do fato de que uma implementação é dependente de sua interface, mas o contrário não é verdadeiro. Isso significa que você pode colocar a interface e a implementação em pacotes separados, e o pacote da implementação tem uma dependência em relação ao pacote da interface. Outros pacotes podem depender do pacote da interface sem depender do pacote da implementação.

É claro que, em tempo de execução, o *software* não funcionará sem alguma implementação da interface. Isso pode se dar tanto em tempo de compilação, usando um pacote separado que ligue os dois, quanto em tempo de configuração, usando um *Plugin* (465).

Você pode colocar a interface no pacote do cliente (como no esboço) ou em um terceiro pacote (Figura 18.1). Se houver apenas um cliente para a implementação, ou todos os clientes estiverem no mesmo pacote, então você poderia também colocar a interface com o cliente. Uma boa maneira de pensar nisso é que os desenvolvedores do pacote cliente são responsáveis pela definição da interface. Basicamente o pacote cliente indica que ele trabalhará com qualquer outro pacote que implemente a interface que ele define. Se você tiver diversos pacotes clientes, um terceiro pacote de interface é melhor. Ele também é melhor se você quiser mostrar que a definição da interface não é a responsabilidade dos desenvolvedores do pacote cliente. Este seria o caso se os desenvolvedores da implementação fossem responsáveis por isso.

Você tem que considerar que característica da linguagem usar para a interface. Para linguagens que suportam a idéia de interfaces, como Java e C#, a palavra-chave interface é a escolha óbvia. Entretanto, ela pode não ser a melhor. Uma classe abstrata pode ser uma boa interface, porque nela você pode ter implementações de comportamento comuns, mas opcionais.

Uma das coisas complicadas a respeito de interfaces separadas é como instanciar a implementação. Isso geralmente requer conhecimento da classe da implementação. A abordagem comum é usar um objeto fábrica separado, em que novamente há uma *Interface Separada* para a fábrica. Você ainda tem que ligar uma implementação à fábrica, e o *Plugin* (465) é uma boa maneira de fazê-lo. Isso não significa apenas

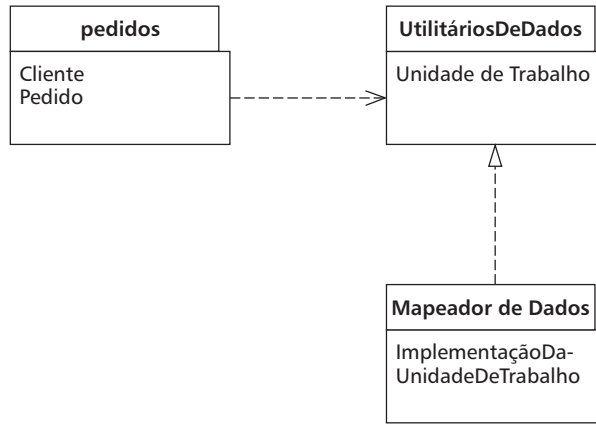


Figura 18.1 Colocando a *Interface Separada* em um terceiro pacote.

que não há dependência, mas também adia a decisão sobre a classe da implementação para o tempo de configuração.

Se você não quiser ir até o fim com um *Plugin* (465), uma alternativa mais simples é deixar que um outro pacote, que conheça tanto a interface quando a implementação, instancie os objetos corretos no início da aplicação. Quaisquer objetos que usem uma *Interface Separada* podem, eles mesmos, serem instanciados ou ter fábricas instanciadas no início da aplicação.

Quando Usá-la

Você usa uma *Interface Separada* quando precisa quebrar a dependência entre duas partes do sistema. Aqui estão alguns exemplos:

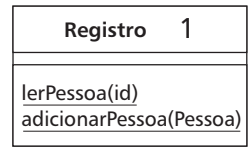
- Você construiu algum código abstrato para casos comuns em um pacote *framework* que precisa chamar algum código específico da aplicação.
- Você tem algum código em uma camada que precisa chamar código em outra camada que ele não deveria ver, tal como código de domínio chamando um *Mapeador de Dados* (170).
- Você precisa chamar funções desenvolvidas por outro grupo de desenvolvimento, mas não quer uma dependência com suas APIs.

Deparo-me com muitos programadores que têm interfaces separadas para cada classe que escrevem. Considero que isso é excessivo, especialmente para o desenvolvimento de aplicações. Manter separadas interfaces e implementações é um trabalho extra, especialmente porque, freqüentemente, você precisa também de classes fábricas (com interfaces e implementações). Para aplicações, recomendo o uso de uma interface separada apenas se você quiser quebrar um dependência ou quiser ter diversas implementações independentes. Se você colocar a interface e a implementação juntas e precisar separá-las mais tarde, esta é uma refatoração simples que pode ser adiada até que você precise fazê-la.

Existe um ponto em que esta forma diligente de gerenciamento de dependências pode se tornar um pouco tola. Ter apenas uma dependência para criar um objeto e, a partir daí, usar apenas a interface, é normalmente suficiente. O problema vem quando você quer reforçar regras de dependência, como fazer uma verificação de dependência em tempo de construção. Então, todas as dependências têm que ser removidas. Para um sistema menor reforçar regras de dependência é um problema menor, mas para sistemas maiores é uma disciplina que vale muito a pena.

Registro (Registry)

Um objeto conhecido que os outros objetos podem usar para encontrar objetos e serviços comuns.



Quando você quer encontrar um objeto, você normalmente começa com outro objeto que tenha uma associação com ele e usa essa associação para navegar até ele. Assim, se você quiser encontrar todos os pedidos de um cliente, comece com o objeto cliente e use um método nele para chegar aos pedidos. Entretanto, em alguns casos, você não terá um objeto apropriado com o qual começar. Você pode saber o número do ID do cliente, mas não ter uma referência para ele. Neste caso, você precisa de algum tipo de método de busca – um buscador – mas a questão permanece: Como você chega até o buscador?

O *Registro* é basicamente um objeto global, ou pelo menos ele se parece com um – mesmo se ele não for tão global quanto possa parecer.

Como Funciona

Como com qualquer objeto, você tem que pensar no projeto do *Registro* em termos de interface e implementação. Como muitos objetos, as duas são bastante diferentes, embora, muitas vezes, as pessoas cometam o erro de pensar que elas devam ser o mesmo.

A primeira coisa na qual pensar é a interface e, para os *Registros*, minha interface preferida são os métodos estáticos. Um método estático em uma classe é fácil de encontrar em qualquer lugar da aplicação. Além disso, você pode encapsular qualquer lógica que quiser dentro do método estático, incluindo delegação para outros métodos, quer sejam estáticos, quer de instâncias.

Entretanto, só porque seus métodos são estáticos, isso não significa que seus dados devam estar em campos estáticos. De fato, quase nunca uso campos estáticos a menos que eles sejam constantes.

Antes que você decida sobre como armazenar seus dados, pense no escopo dos mesmos. Os dados de um *Registro* podem variar com diferentes contextos de execução. Alguns deles são globais por todo o processo; alguns, globais em uma *thread*; e alguns, globais em uma sessão. Escopos diferentes pedem implementações diferentes, mas não pedem interfaces diferentes. O programador da aplicação não tem que saber se uma chamada a um método estático produz dados cujo escopo é o processo ou a *thread*. Você pode ter diferentes *Registros* para diferentes escopos, mas também pode ter um único *Registro* no qual diferentes métodos têm diferentes escopos.

Se os seus dados forem comuns a todo um processo, um campo estático é uma opção. Entretanto, raramente, uso campos estáticos mutáveis, porque eles não per-

mitem a substituição. Pode ser extremamente útil ser capaz de, com alguma finalidade específica, substituir um *Registro*, especialmente para testes (o *Plugin* (465) é uma boa maneira de fazer isso).

Dessa forma, para um Registro com escopo de processo, a opção usual é um *singleton* [Gang of Four]. A classe *Registro* contém um único campo estático que armazena uma instância do *Registro*. Quando as pessoas usam um *singleton*, elas freqüentemente obrigam o cliente a explicitamente acessar os dados subjacentes (`Registro.instânciaÚnica( ).lerBobo( ))`, mas eu prefiro um método estático que esconda o objeto *singleton* de mim (`Registro.lerBobo( ))`. Isso funciona especialmente bem, já que as linguagens baseadas em C permitem aos métodos estáticos acessar dados privados de instâncias.

Os *Singletons* são largamente utilizados em aplicações com uma única *thread*, mas podem ser um problema em aplicações com diversas delas. Isso ocorre porque é muito fácil acontecer de diversas *threads* manipularem o mesmo objeto de formas imprevisíveis. Você pode conseguir resolver o problema com sincronização, mas a dificuldade de escrever o código de sincronização provavelmente o levará a um hospício antes que você consiga eliminar todos os erros. Por esta razão, não recomendo o uso de um *singleton* para dados mutáveis em um ambiente de *multi-thread*. Isso funciona bem com dados imutáveis, já que qualquer coisa que não possa ser alterada não terá problemas causados pelo choque de *threads*. Assim, algo como uma lista de todos os estados dos Estados Unidos é uma boa candidata para um *Registro* com escopo de processo. Tais dados podem ser carregados, quando um processo inicia, e nunca precisam de alteração, ou podem ser atualizados raramente com algum tipo de interrupção do processo.

Um tipo comum de dados de *Registro* são dados com escopo de *thread*. Um bom exemplo é uma conexão a um banco de dados. Neste caso, muitos ambientes lhe dão algum tipo de armazenamento específico de *thread*, como a *thread* local de Java. Outra técnica é um dicionário no qual chave é uma *thread* cujo valor é um objeto de dados apropriado. Uma solicitação de uma conexão resulta em uma busca nesse dicionário pela *thread* corrente.

O importante a lembrar sobre dados com escopo de *thread* é que eles não parecem diferentes dos dados com escopo de processo. Ainda posso usar um método como `Registro.lerConexãoBD( )`, que é a mesma forma de quando estou acessando dados com escopo de processo.

Uma busca no dicionário também é uma técnica que você pode usar para dados com escopo de sessão. Aqui você precisa de uma identificação de sessão, mas ela pode ser colocada em um registro com escopo de *thread* quando uma solicitação tem início. Quaisquer acessos subseqüentes aos dados da sessão podem procurar os dados em um mapa cujas chaves são sessões usando a identificação da sessão mantida no armazenamento específico de *thread*.

Se você estiver usando um *Registro* com escopo de *thread* com métodos estáticos, pode ter problemas de desempenho com diversas *threads* passando por estes métodos. Neste caso, o acesso direto à instância da *thread* evitará o gargalo.

Algumas aplicações podem ter um único *Registro*; outras podem ter vários. Os *Registros* são normalmente classificados pela camada do sistema ou pelo contexto de execução. Minha preferência é classificá-los pelo modo como são usados, e não pela sua implementação.

Quando Usá-lo

Apesar do encapsulamento de um método, um *Registro* consiste ainda de dados globais e, como tal, é algo com o que não me sinto confortável. Quase sempre vejo alguma forma de *Registro* em uma aplicação, mas sempre tento acessar os objetos por meio de referências normais inter-objetos. Basicamente, você só deve usar um *Registro* como último recurso.

Existem alternativas ao uso de um *Registro*. Uma é passar por parâmetros quaisquer dados necessários em vários pontos da aplicação. O problema é que os parâmetros têm de ser adicionados a chamadas de métodos em que eles não são necessários ao método invocado, mas a algum outro método que é chamado diversas camadas abaixo na árvore de chamadas. Passar um parâmetro adiante quando, durante 90% do tempo ele não é necessário, é o que me leva a usar um *Registro*.

Uma alternativa que tenho visto ao uso do *Registro* é adicionar aos objetos, no momento em que eles são criados, uma referência aos dados comuns. Embora isso leve a um parâmetro extra em um construtor, ele pelo menos é usado apenas por esse construtor. Ainda assim, é mais trabalho do que muitas vezes vale a pena, mas se você tiver dados que são usados apenas por um subconjunto das classes, esta técnica permite a você restringir as coisas deste modo.

Um dos problemas com um *Registro* é que ele tem que ser modificado cada vez que você adiciona novos dados. É por isso que algumas pessoas preferem usar um mapa como seu armazenador de dados globais. Prefiro a classe explícita porque ela mantém os métodos explícitos, de modo que não há confusão sobre qual chave você usa para encontrar alguma coisa. Com uma classe explícita, você pode apenas olhar o código fonte ou a documentação gerada para ver o que está disponível. Com um mapa você tem que encontrar lugares no sistema onde os dados são lidos ou gravados no mapa para descobrir qual chave é usada, ou depender da documentação que rapidamente se torna desatualizada. Uma classe explícita também permite a você manter a segurança de tipos em uma linguagem estaticamente tipada, assim como encapsular a estrutura do *Registro*, de modo que você possa refatorá-lo à medida que o sistema cresce. Além disso, um mapa simples não é encapsulado, o que torna mais difícil esconder a implementação. Isso é especialmente complicado se você tiver que alterar o escopo de execução dos dados.

Assim, existem situações em que é correto usar um *Registro*, mas lembre-se de que quaisquer dados globais são sempre culpados, até que se prove o contrário.

Exemplo: Um Registro *Singleton* (Java)

Considere uma aplicação que leia dados de um banco de dados e, a seguir, faça modificações nestes dados para transformá-los em informações. Bem, imagine um sistema razoavelmente simples que use *Gateways de Linhas de Dados* (158) para o acesso a dados. Este sistema tem objetos de busca para encapsular as consultas ao banco de dados. Os buscadores ficam melhor em instâncias porque, desta forma, podemos substituí-los para criar um *Stub de Serviço* (469) com o propósito de testes. Precisamos de um lugar para colocá-los. Um *Registro* é a escolha óbvia.

Um registro *singleton* é um exemplo muito simples do padrão Singleton [Gang of Four]. Você tem uma variável estática para a instância única.

```
class Registro...

    private static Registro lerInstância ( ) {
```

```
        return únicaInstância;
    }
    private static Registro únicaInstância = new Registro ( );
```

Tudo que é armazenado no registro é armazenado na instância.

```
class Registro...

    protected BuscadorDePessoa buscadorDePessoa = new BuscadorDePessoa ( );
```

No entanto, para tornar o acesso mais fácil, torno os métodos públicos estáticos:

```
class Registro...

    public static BuscadorDePessoa buscadorDePessoa ( ) {
        return lerInstância( ).buscadorDePessoa;
    }
```

Posso reinicializar o registro simplesmente criando uma nova instância única.

```
class Registro...

    public static void inicializar ( ) {
        únicaInstância = new Registro( );
    }
```

Se eu quiser usar *Stubs de Serviço* (469) para testes, uso em vez disso uma subclasse.

```
class StubDoRegistro extends Registro...

    public StubDoRegistro ( ) {
        buscadorDePessoa = new StubDoBuscadorDePessoa( );
    }
```

O buscador do *Stub de Serviço* (469) apenas retorna instâncias codificadas explicitamente do *Gateway de Linhas de Dados* (158) da pessoa.

```
class StubDoBuscadorDePessoa...

    public Pessoa buscar(long id) {
        if (id == 1) {
            return new Pessoa("Fowler", "Martin", 10);
        }
        throw new IllegalArgumentException("Não foi possível encontrar id: " + String.valueOf(id));
        }
```

Coloco um método no registro para inicializá-lo no modo *stub*, mas, mantendo todo o comportamento do *stub* na subclasse, posso separar todo o código necessário para testes.

```
class Registro...

    public static void inicializarStub ( ) {
        únicaInstância = new StubDoRegistro ( );
    }
```

Exemplo: Registro à Prova de *Thread* (Java)

Matt Foemmel e Martin Fowler

O simples exemplo anterior não funcionará em uma aplicação *multi-thread* na qual as diferentes *threads* precisam de seu próprio registro. Java fornece o mecanismo de variáveis de armazenamento específicas de *thread* [Schmidt], que são locais a uma *thread*, apropriadamente, chamadas de variáveis locais de *threads*. Você pode usá-las para criar um registro que seja único para uma *thread*.

```
class RegistroDeThreadLocal...

    private static ThreadLocal instâncias = new ThreadLocal ( );
    public static RegistroDeThreadLocal lerInstância ( ) {
        return (RegistroDeThreadLocal) instâncias.get( );
    }
```

O *Registro* precisa ser configurado com métodos para a obtenção e liberação deste registro. Normalmente, você faz isso no contexto de uma transação ou chamada de sessão.

```
class RegistroDeThreadLocal...

    public static void início( ) {
        Assert.isTrue(instâncias.get( ) == null);
        instâncias.set(new RegistroDeThreadLocal ( ));
    }
    public static void fim ( ) {
        Assert.notNull(lerInstância ( ));
        instâncias.set(null);
    }
```

Você pode então, como antes, armazenar buscadores de pessoas.

```
class RegistroDeThreadLocal...

    private BuscadorDePessoa buscadorDePessoa = new BuscadorDePessoa( );
    public static BuscadorDePessoa buscadorDePessoa ( ) {
        return lerInstância( ).buscadorDePessoa;
}
```

As chamadas de fora encapsulam o seu uso de um registro nos métodos início e fim.

```
    try{
        RegistroDeThreadLocal.inicio( );
        BuscadorDePessoa b1 = RegistroDeThreadLocal.buscadorDePessoa( );
        Pessoa martin = b1.buscar(1);
        assertEquals("Fowler", martin.lerSobrenome( ) );
    } finally {RegistroDeThreadLocal.fim( );
    }
```

Objeto Valor (Value Object)

Um objeto pequeno e simples, como dinheiro ou uma faixa de datas, cuja igualdade não é baseada na identidade.

Com sistemas de objetos de vários tipos, descobri que é útil distinguir entre objetos referência e *Objetos Valor*. Dos dois, um *Objeto Valor* é geralmente o menor. Ele é semelhante aos tipos primitivos encontrados em muitas linguagens que não são puramente orientadas a objetos.

Como Funciona

Definir a diferença entre um objeto referência e um *Objeto Valor* pode ser um pouco complicado. De maneira geral, gostamos de pensar que *Objetos Valor* são pequenos objetos, tais como um objeto dinheiro ou uma data, enquanto que objetos referência são grandes, tais como um pedido ou um cliente. Tal definição é útil, mas irritantemente informal.

A principal diferença entre objetos de referência e objetos valor reside em como eles lidam com a igualdade. Um objeto referência usa a identidade como base para a igualdade – talvez a identidade dentro do sistema de programação, tal como a identidade embutida das linguagens de programação OO, ou talvez algum tipo de número de identificação, como a chave primária em um banco de dados relacional. Um *Objeto Valor* baseia sua noção de igualdade em valores de campos dentro da classe. Assim, dois objetos data podem ser o mesmo objeto se seus valores de dia, mês e ano forem os mesmos.

Essa diferença se expressa na maneira como você lida com elas. Uma vez que *Objetos Valor* são pequenos e facilmente criados, eles freqüentemente são passados por valor em vez de por referência. Você não se importa realmente com quantos objetos 18 de março de 2001 existem no seu sistema. Tampouco se importa se dois objetos compartilham o mesmo objeto data físico ou se eles têm cópias diferentes, porém iguais.

A maioria das linguagens não têm facilidades especiais para objetos valor. Para que os objetos valor funcionem apropriadamente nestes casos, é uma boa idéia torná-los imutáveis – ou seja, uma vez criados nenhum de seus campos muda. A razão para isso é evitar problemas com apelidos (*aliasing*). Tais problemas podem ocorrer quando dois objetos compartilham o mesmo objeto valor e um dos donos altera os valores nele. Assim, se Martin foi contratado no dia 18 de março e sabemos que Cindy foi contratada no mesmo dia, podemos estabelecer a data de contratação de Cindy como sendo a mesma de Martin. Se, a seguir, Martin altera o mês da sua data de contratação para maio, a data de contratação de Cindy muda também. Quer seja correto, quer não, não é o que as pessoas esperam. Normalmente, com valores pequenos como este as pessoas esperam alterar uma data de contratação substituindo o objeto data existente por um novo. Tornar os *Objeto Valor* imutáveis satisfaz esta expectativa.

Os *Objetos Valor* não devem ser persistidos como registros completos. Em vez disso, use um *Valor Embutido* (261) ou um *LOB Serializado* (264). Uma vez que os *Objetos Valor* são pequenos, um *Valor Embutido* (261) é geralmente a melhor escolha porque ele também permite consultas SQL usando os dados em um *Objeto Valor*.

Se você estiver executando uma quantidade grande de serialização binária, pode descobrir que otimizar a serialização dos *Objetos Valor* pode melhorar o desempenho, especialmente em linguagens como Java que não tratam *Objetos Valor* de modo especial.

Para um exemplo de um *Objeto Valor*, veja *Dinheiro* (455).

Implementação .NET

.NET tem um tratamento de primeira classe para *Objetos Valor*. Em C# um objeto é marcado como um *Objeto Valor* declarando-o como uma *struct* em vez de como uma classe. O ambiente então o trata com semântica de valor.

Quando Usá-lo

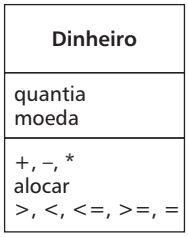
Trate algo como um *Objeto Valor* quando estiver baseando as igualdades em algo que não seja uma identidade. Vale a pena considerar isso para qualquer objeto pequeno que seja fácil de criar.

Colisões de Nomes Tenho visto o termo *Objeto Valor* usado para este padrão já há algum tempo. Infelizmente, vi recentemente a comunidade J2EE [Alur *et al.*] usar o termo "objeto valor" para significar um *Objeto de Transferência de Dados* (380), o que causou uma tempestade em um copo d'água na comunidade de padrões. Esta é apenas uma daquelas colisões sobre nomes que acontecem o tempo todo neste negócio. Recentemente [Alur *et al.*] decidiram usar o termo *objeto de transferência*.

Continuo usando *Objeto Valor* desta maneira, neste texto. Senão por nenhum outro motivo, ele me permite ser consistente com meus escritos anteriores!

Dinheiro (Money)

Representa um valor monetário.



Uma grande proporção dos computadores no mundo manipulam dinheiro, então sempre me intrigou que dinheiro não seja realmente um tipo de dados de primeira classe em nenhuma das principais linguagens de programação. A falta de um tipo causa problemas, os mais óbvios envolvendo moedas. Se todos os seus cálculos forem feitos em uma única moeda corrente, este não é um grande problema, mas uma vez que você envolva diversas moedas, você quer evitar adicionar seus dólares aos seus ienes sem levar as diferenças de moeda em consideração. O problema mais sutil é com o arredondamento. Cálculos monetários são muitas vezes arredondados para a menor unidade da moeda. Quando você faz isso, é fácil perder alguns centavos (ou o seu equivalente local) devido aos erros de arredondamento.

O bom a respeito da programação orientada a objetos é que você pode consertar estes problemas criando uma classe Dinheiro que lide com eles. É claro que ainda é surpreendente que, em verdade, nenhuma das bibliotecas principais de classes básicas faça isso.

Como Funciona

A idéia básica é ter uma classe Dinheiro com campos para a quantidade numérica e a moeda corrente. Você pode armazenar a quantidade como um tipo inteiro ou um tipo decimal fixo. O tipo decimal é mais fácil para algumas manipulações, o integral para outras. Você deve evitar completamente qualquer tipo de ponto flutuante, pois isso introduzirá o tipo de problemas de arredondamento cuja finalidade de *Dinheiro* é evitar. Na maior parte do tempo, as pessoas querem valores monetários arredondados para a menor unidade da moeda, como os centavos no dólar. Entretanto, há vezes em que unidades fracionárias são necessárias. É importante deixar claro com que tipo de dinheiro você está trabalhando, especialmente em uma aplicação que usa ambos os tipos. Faz sentido ter diferentes tipos para os dois casos, pois eles se comportam de forma bastante diferente no que diz respeito à aritmética.

O dinheiro é um *Objeto Valor* (453), então deve ter suas operações de igualdade e código *hash* sobrescritas para serem baseadas na moeda corrente e na quantia.

O dinheiro precisa de operações aritméticas, de modo que você possa usar objetos desse tipo tão facilmente quanto usa números. Contudo, operações aritméticas com dinheiro têm algumas diferenças importantes em relação às operações com dinheiro em números. A mais óbvia, qualquer adição ou subtração precisa ter ciência da moeda, de modo que você possa reagir se tentar adicionar diferentes tipos de moedas. A resposta mais simples, e mais comum, é tratar a adição de moedas incom-

patíveis como um erro. Em algumas situações mais sofisticadas, você pode usar a idéia de Ward Cunningham de um saco de dinheiro. Esse é um objeto que contém moedas de tipos diferentes juntas em um objeto. Este objeto pode então participar de cálculos como qualquer objeto dinheiro. Ele também pode ser avaliado em uma moeda em particular.

A multiplicação e a divisão acabam sendo mais complicadas devido a problemas de arredondamento. Quando você multiplica dinheiro, faça isso com uma grandeza escalar. Se você quiser adicionar uma taxa de 5% a uma conta, multiplique-a por 0,05, de modo que você se preocupe apenas com a multiplicação por tipos numéricos normais.

A complicação vem com o arredondamento, especialmente ao alocar dinheiro entre lugares diferentes. Aqui está o simples enigma de Matt Foemmel. Suponha que eu tenha uma regra de negócio que diga que tenho de alocar a totalidade de uma importância em dinheiro entre duas contas: 70% para uma e 30% para outra. Tenho 5 centavos para alocar. Se eu fizer o cálculo matemático, acabo com 3,5 centavos e 1,5 centavo. Não importa de que maneira eu arredonde essas quantias, arrumo um problema. Se eu fizer o arredondamento costumeiro para o mais próximo, então 1,5 vira 2 e 3,5 vira 4. Assim eu acabo ganhando 1 centavo. Arredondar para baixo me dá 4 centavos e arredondar para cima me dá 6. Não há um esquema geral de arredondamento que eu possa aplicar que tanto evite a perda quanto o ganho de 1 centavo.

Tenho visto diversas soluções para esse problema.

- Talvez a mais comum seja ignorá-lo – afinal, é apenas 1 centavo aqui e ali. Entretanto isso tende a deixar os contadores compreensivelmente nervosos.
- Durante a alocação você sempre executa a última alocação subtraindo do que você já alocou até o momento. Isso evita a perda de centavos, mas você pode ter o acúmulo de uma quantia significativa de centavos na última alocação.
- Permita aos usuários de uma classe Dinheiro declarar o esquema de arredondamento quando eles chamam o método. Isso permite a um programador dizer que o caso dos 70% arredonda para cima e o dos 30% para baixo. As coisas ficam mais complicadas quando você tem de fazer a alocação por dez contas em vez de duas. Você também tem que se lembrar de arredondar. Para encorajar as pessoas a lembrar, tenho visto algumas classes Dinheiro forçarem a existência de um parâmetro para arredondamento na operação de multiplicação. Isso não apenas obriga o programador a pensar sobre qual arredondamento precisa, mas também poderia lembrá-lo dos testes a escrever. Entretanto, isso se torna confuso se você tiver muitos cálculos de taxas que sejam todos arredondados da mesma maneira.
- Minha solução favorita: tenha uma função de alocação na classe Dinheiro. O parâmetro para o alocador é uma lista de números, representando a proporção a ser alocada (ele se pareceria com algo como `umDinheiro.alocar[7,3])`. O alocador retorna uma lista de objetos dinheiro, garantindo que nenhum centavo seja perdido espalhando-os pelos objetos dinheiro alocados de um modo que, de fora, parece pseudo-randômico. O alocador tem suas falhas: você tem que se lembrar de usá-lo e quaisquer regras precisas sobre para onde devem ir os centavos são difíceis de impingir.

A questão fundamental aqui é entre usar multiplicação para determinar o valor de uma cobrança proporcional (como um imposto) e usá-la para alocar uma soma de dinheiro em diversos lugares. A multiplicação funciona bem no primeiro caso, mas um alocador funciona melhor para este último. O importante é considerar sua intenção ao usar multiplicação ou divisão em um valor monetário.

Você pode querer converter de um tipo de moeda para outro com um método como `umDinheiro.converterPara(Moeda.DOLLARS)`. O modo óbvio de fazer isso é procurar uma taxa de câmbio e multiplicar por ela. Embora funcione em muitas situações, há casos em que não funciona – novamente devido ao arredondamento. As regras de conversão entre as moedas européias tinham arredondamentos específicos aplicados que faziam uma simples multiplicação não funcionar. Assim, é sensato ter um objeto conversor para encapsular o algoritmo.

As operações de comparação permitem-lhe ordenar objetos de dinheiro. Assim como a operação de adição, as conversões precisam ter ciência da moeda. Você pode escolher ou levantar uma exceção ao comparar moedas diferentes ou fazer uma conversão entre elas.

Uma classe *Dinheiro* pode encapsular o comportamento de impressão. Isso torna muito mais fácil fornecer uma boa apresentação em interfaces com o usuário e relatórios. Uma classe Dinheiro também pode analisar uma *string* para fornecer um mecanismo de entrada que conheça moedas, o que, novamente, é muito útil para a interface com o usuário. Aqui, as bibliotecas da sua plataforma podem ajudar. Cada vez mais as plataformas fornecem suporte à globalização com formatadores de números explícitos para países específicos.

Armazenar um objeto *Dinheiro* em um banco de dados sempre traz um problema, uma vez que os bancos de dados também não parecem entender que dinheiro é importante (embora seus vendedores entendam.) O caminho óbvio a tomar é usar um *Valor Embutido* (261), o que resulta em armazenar uma moeda para cada objeto dinheiro. Isso pode ser exagerado quando, por exemplo, uma conta possa ter todos os seus lançamentos em libras. Neste caso, você pode armazenar a moeda na conta e alterar o mapeamento do banco de dados para trazer a moeda da conta sempre que você carregar os lançamentos.

Quando Usá-lo

Uso *Dinheiro* para quase todo cálculo numérico em ambientes orientados a objetos. A razão principal é encapsular a manipulação do comportamento de arredondamento, o que ajuda a reduzir os problemas de erros de arredondamento. Outra razão para usar *Dinheiro* é tornar muito mais fácil o trabalho com diversas moedas. A objeção mais comum a *Dinheiro* é o desempenho, embora raramente eu tenha ouvido falar de casos em que ele fez qualquer diferença perceptível, e mesmo nesses casos o encapsulamento, muitas vezes, torna mais fácil a otimização.

Exemplo: Uma Classe Dinheiro (Java)

por Matt Foemmel e Martin Fowler

A primeira decisão é que tipo de dados usar para a quantia. Se alguém precisar ser convencido de que um número de ponto flutuante é uma má idéia, peça a ele para rodar este código.

```
double val = 0.00;
for(int i = 0; i< 10; i++) val += 0.10;
System.out.println(val == 1.00);
```

Com os números em ponto flutuante descartados, a escolha fica entre inteiros e decimais de ponto fixo, o que em Java se resume a `BigDecimal`, `BigInteger` e `long`. Usar um valor inteiro de fato torna a matemática interna mais fácil, e se usarmos `long` podemos usar primitivas e, assim, ter expressões matemáticas legíveis.

```
class Dinheiro...
    private long quantia;
    private Currency moeda;
```

Estou usando uma quantia inteira, ou seja, a quantidade da menor unidade básica da moeda, a qual chamo de centavos no código porque este é um nome tão bom quanto qualquer outro. Com um tipo *long* obtemos um erro de estouro (*overflow*) se o número ficar grande demais. Se você nos der $92.233.720.368.547.758,09 escreveremos para você uma versão que use `BigIntegers`.

É útil fornecer construtores de diversos tipos numéricos.

```
    public Dinheiro (double quantia, Currency moeda) {
        this.moeda = moeda;
        this.quantia = Math.round(quantia * fatorCentavos( ));
    }
    public Dinheiro (long quantia, Currency moeda) {
        this.moeda = moeda;
        this.quantia = quantia * fatorCentavos( );
    }
    private static final int[ ] centavos = new int[ ] {1, 10, 100, 1000};
    private int fatorCentavos ( ) {
        return centavos[moeda.getDefaultFractionDigits( )];
    }
```

Diferentes moedas têm diferentes quantias fracionárias. A classe Currency de Java 1.4 lhe informará o número de dígitos fracionários em uma classe. Podemos determinar quantas unidades menores há em uma unidade maior elevando dez à potência, mas isso é tão trabalhoso em Java que o *array* é mais fácil (e provavelmente mais rápido). Estamos preparados para aceitar o fato de que este código falha se alguém usar quatro dígitos fracionários.

Embora, na maior parte do tempo, você vá querer usar diretamente as operações na classe dinheiro, há ocasiões em que você precisará acessar os dados encapsulados.

```
class Dinheiro...

    public BigDecimal quantia ( ) {
        return BigDecimal.valueOf(quantia, moeda.getDefaultFractionDigits( ));
    }
    public Currency moeda ( ) {
        return moeda;
    }
```

Você deve sempre questionar seu uso de métodos de acesso. Quase sempre há uma maneira melhor que não quebrará o encapsulamento. Um exemplo que não poderíamos evitar é o mapeamento de banco de dados, como em *Valor Embutido* (261).

Se você usar muito freqüentemente uma moeda para quantias literais, um construtor auxiliar pode ser útil.

```
class Dinheiro...

    public static Dinheiro dólares (double quantia) {
        return new Dinheiro (quantia, Currency.USD);
    }
```

Como *Dinheiro* é um *Objeto Valor* (453), você precisará definir *equals*.

```
class Dinheiro...

    public boolean equals (Object outro) {
        return (outro instanceof Dinheiro) && equals( (Dinheiro)outro);
    }
    public boolean equals (Dinheiro outro) {
        return moeda.equals(outro.moeda) && (quantia == outro.quantia);
    }
```

E sempre que houver um método *equals* deveria haver um *hash*.

```
class Dinheiro...

    public int hashCode( ) {
        return (int) (quantia ^ (quantia >>> 32));
    }
```

Começaremos pela aritmética com a adição e a subtração.

```
class Dinheiro...

    public Dinheiro adicionar(Dinheiro outro) {
        asserçãoMesmaMoedaQue(outro);
        return novoDinheiro (quantia + outro.quantia);
    }
    private void asserçãoMesmaMoedaQue(Dinheiro parâmetro) {
        Assert.equals ("erro de comparação de moedas", moeda, parâmetro.moeda);
    }
    private Dinheiro novoDinheiro(long quantia) {
        Dinheiro dinheiro = new Dinheiro( );
        dinheiro.moeda = this.moeda;
        dinheiro.quantia = quantia,
        return dinheiro;
    }
```

Perceba aqui o uso de um método fábrica privado que não faz a conversão usual para a quantia baseada em centavos. Usaremos isso algumas vezes dentro do próprio código da classe *Dinheiro*.

Com a adição definida, a subtração é fácil.

```
class Dinheiro...

    public Dinheiro subtrair(Dinheiro outro) {
        asserçãoMesmaMoedaQue(outro);
        return novoDinheiro(quantia - outro.quantia);
    }
```

O método base para a comparação é `compararCom`.

```
class Dinheiro...

    public int compararCom(Objeto outro) {
        return compararCom((Dinheiro) outro);
    }
    public int compararCom(Dinheiro outro) {
        asserçãoMesmaMoedaQue (outro);
        if (quantia < outro.quantia) return -1;
        else if (quantia == outro.quantia) return 0;
        else return 1;
    }
```

Embora, atualmente, isso seja tudo o que você consegue na maioria das classes Java, achamos que o código fica mais legível com os outros métodos de comparação tais como estes.

```
class Dinheiro...

    public boolean maiorDoQue(Dinheiro outro) {
        return (compararCom(outro) > 0);
    }
```

Agora estamos prontos para olhar a multiplicação. Estamos fornecendo um modo de arredondamento padrão, mas você pode também configurar o seu próprio.

```
class Dinheiro...

    public Dinheiro multiplicar(double quantia) {
        return multiplicar(new BigDecimal(quantia));
    }
    public Dinheiro multiplicar(BigDecimal quantia) {
        return multiplicar(quantia, BigDecimal.ROUND_HALF_EVEN);
    }
    public Dinheiro multiplicar(BigDecimal quantia, int modoDeArredondamento) {
        return new Dinheiro(quantia( ).multiplicar(quantia), moeda, modoDeArredondamento);
    }
```

Se você quiser alocar uma importância em dinheiro entre muitos destinatários e não quiser perder centavos, irá precisar de um método de alocação. O mais simples deles aloca a mesma quantia (quase) entre vários destinatários.

```
class Dinheiro...

    public Dinheiro[ ] alocar(int n) {
        Dinheiro resultadoBaixo = novoDinheiro(quantia / n);
        Dinheiro resultadoAlto = novoDinheiro(resultadoBaixo.quantia +1);
```

```
        Dinheiro[ ] resultados = new Dinheiro[n];
        int resto = (int) quantia % n;
        for (int i = 0; i < resto; i++) resultados[i] = resultadoAlto;
        for (int i = resto; i < n; i++) resultados[i] = resultadoBaixo;
        return resultados;
    }
```

Um algoritmo de alocação mais sofisticado pode lidar com qualquer razão.

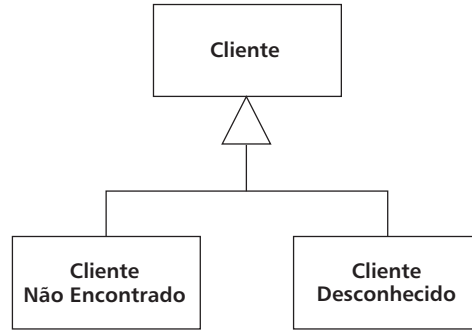
```
class Dinheiro...

    public Dinheiro[ ] alocar (long[ ] razões) {
        long total = 0;
        for (int i = 0; i < razões.length; i++) total += razões[i];
        long resto = quantia;
        Dinheiro [ ] resultados = new Dinheiro[razões.length];
        for (int i = 0; i< resultados.length; i++) {
            resultados[i] = novoDinheiro(quantia * razões[i] / total);
            resto -= resultados[i].quantia;
        }
        for (int i = 0; i < resto; i++) {
            resultados[i].quantia++;
        }
        return resultados;
    }
```

Você pode usar isso para resolver o Enigma de Foemmel.

```
class Dinheiro...

    public void testeAlocar2( ) {
        long[ ] alocação = {3,7};
        Dinheiro[ ] resultado = Dinheiro.dólares(0.05).alocar(alocação);
        assertEquals(Dinheiro.dólares(0.02), resultado[0]);
        assertEquals(Dinheiro.dólares(0.03), resultado[1]);
    }
```

Caso Especial (Special Case)

Uma subclasse que fornece comportamento especial para casos particulares.



Os nulos são coisas complicadas em programas orientados a objetos porque eles frustram o polimorfismo. Normalmente, você pode invocar o método `bobo` livremente em uma variável que seja uma referência para um determinado tipo sem se preocupar se o item pertence a este tipo ou uma subclasse. Com uma linguagem fortemente tipada, você pode até mesmo fazer o compilador verificar se a chamada é correta. Entretanto, já que uma variável pode conter um nulo, você pode se deparar com um erro em tempo de execução ao invocar uma mensagem sobre um nulo, o que lhe dará um belo e amigável rastreamento de pilha.

Se for possível que uma variável seja nula, você tem que lembrar de cercá-la com código para testar a condição nula, de modo que você possa fazer a coisa certa se um nulo estiver presente. Muitas vezes, a coisa certa é a mesma em muitos contextos, então você acaba escrevendo código similar em muitos lugares – cometendo o pecado da duplicação de código.

Os nulos são um exemplo comum de tais problemas, e outros aparecem regularmente. Em sistemas numéricos você precisa lidar com o infinito, que tem regras especiais para coisas, como a adição, que quebram as invariantes costumeiras dos números reais. Uma das minhas primeiras experiências em *software* de negócio foi com um cliente de um serviço que não era completamente conhecido, referido como "ocupante." Todos estes sugerem a alteração do comportamento usual do tipo.

Em vez de retornar nulo, ou algum valor estranho, retorne um *Caso Especial* que tenha a mesma interface da que o solicitante espera.

Como Funciona

A idéia básica é criar uma subclasse para tratar o *Caso Especial*. Assim, se você tiver um objeto Cliente e quiser evitar verificações de nulos, cria um objeto Cliente nulo. Pegue todos os métodos do Cliente e sobrescreva-os no *Caso Especial* para fornecer algum comportamento inofensivo. Então, sempre que você tiver um nulo, coloque uma instância do cliente nulo no seu lugar.

Geralmente, não existe nenhuma razão para distinguir entre instâncias diferentes do cliente nulo, de modo que, freqüentemente, você pode implementar um *Caso Especial* com um peso-mosca [Gang of Four]. Você não pode fazer isso todo o tempo.

Para um serviço, você pode acumular débitos de um ocupante, mesmo que você não possa sempre entregar a conta, de modo que é importante manter os ocupantes separados.

Um nulo pode significar coisas diferentes. Um cliente nulo pode significar a não-existência de um cliente ou pode significar que existe um cliente mas não sabemos quem ele é. Em vez de usar somente um cliente nulo, considere ter *Casos Especiais* separados para clientes ausentes ou desconhecidos.

Um modo comum de um *Caso Especial* sobrescrever métodos é retornar outro *Caso Especial*, então, se você pedir a um cliente desconhecido a sua última conta, pode bem obter uma conta desconhecida.

A aritmética de ponto flutuante IEEE 754 oferece bons exemplos de *Casos Especiais* com infinito negativo, infinito positivo e não-é-um-número (NaN). Se você dividir por zero, em vez de obter uma exceção a qual terá de tratar, o sistema simplesmente retorna NaN, e NaN participa de expressões aritméticas como qualquer outro número de ponto flutuante.

Quando Usá-lo

Use o *Caso Especial* sempre que, em diversos lugares do sistema, você tiver o mesmo comportamento após uma verificação condicional da ocorrência de uma instância particular de uma classe, ou o mesmo comportamento após uma verificação da ocorrência de nulos.

Leitura Adicional

Ainda não vi o *Caso Especial* escrito como um padrão, mas o **Objeto Nulo** foi descrito em [Woolf]. Se você perdoar o trocadilho irresistível, vejo o Objeto Nulo como um caso especial do *Caso Especial*.

Exemplo: Um Objeto Nulo Simples (C#)

Aqui está um exemplo simples de um *Caso Especial* usado como um objeto nulo.

Temos um empregado regular.

```
class Empregado...

    public virtual String Nome {
        get{return _nome;}
        set{_nome = valor);
    }
    private String _nome;
        public virtual Decimal TotalAtéOMomento {
        get{return calcularTotalDoPeríodo(0);}
    }
    public virtual Contrato Contrato {
        get{return _contrato;}
    }
    private Contrato _contrato;
```

As características da classe poderiam ser sobrescritas por um empregado nulo.

```
class EmpregadoNulo: Empregado, INull...

    public override String Nome {
        get {return "Empregado Nulo";}
        set { }
    }
    public override Decimal TotalAtéOMomento {
        get{return 0m;}
    }
    public override Contrato Contrato {
        get{return Contrato.NULL;}
    }
```

Perceba que quando você pede a um empregado nulo seu contrato, obtém de volta um contrato nulo.

Os valores padrão aqui evitam muitos testes para a verificação de nulos se eles acabam com os mesmos valores nulos. Os valores nulos repetidos são manipulados pelo objeto nulo por *default*. Você também pode testar a ocorrência de nulos explicitamente dando ao cliente um método `éNulo` ou usando uma verificação de tipo para uma interface de sinalização.

Plugin

por David Rice e Matt Foemmel

Conecta classes durante a configuração em vez de na compilação.

A *Interface Separada* (445) é freqüentemente usada quando o código da aplicação roda em múltiplos ambientes de execução, cada um deles requerendo implementações diferentes de um comportamento específico. A maioria dos desenvolvedores fornece a implementação correta escrevendo um método fábrica. Suponha que você defina seu gerador de chave primária com uma *Interface Separada* (445), de modo que você possa usar um contador simples em memória para testes de unidade, mas uma seqüência gerenciada pelo banco de dados para código de produção. Seu método fábrica conterá provavelmente uma declaração condicional que olha uma variável de ambiente local, determina se o sistema está em modo de teste e retorna o gerador de chaves correto. Uma vez que você tenha algumas fábricas, tem uma bagunça nas mãos. Estabelecer uma nova configuração de distribuição – digamos "execute testes de unidade contra a base de dados na memória sem controle de transações" ou "execute no modo de produção em um banco de dados DB2 com controle de transações integral" – requer editar declarações condicionais em várias fábricas, reconstruir e redistribuir. A configuração não deveria ser dispersa por toda a sua aplicação, nem deveria requerer uma reconstrução ou redistribuição. Um *Plugin* resolve ambos os problemas fornecendo a configuração em tempo de execução e centralizada.

Como Funciona

A primeira coisa a fazer é definir com uma *Interface Separada* (445) quaisquer comportamentos que terão implementações diferentes baseadas no ambiente de execução. Além disso, usamos o padrão fábrica básico, apenas com alguns requisitos especiais. A fábrica *Plugin* requer que suas instruções de conexão sejam declaradas em um único ponto externo para que a configuração possa ser gerenciada com facilidade. Adicionalmente, a conexão com as implementações devem ocorrer dinamicamente em tempo de execução, em vez de durante a compilação, de modo que a reconfiguração não irá requerer uma reconstrução.

Um arquivo texto funciona bastante bem como meio de declaração de regras de conexão. A fábrica *Plugin* irá simplesmente ler o arquivo texto, buscar uma entrada especificando a implementação de uma interface requerida e retornar essa implementação.

O *Plugin* funciona melhor em uma linguagem que suporte reflexão porque a fábrica pode construir implementações sem dependências de tempo de compilação. Ao usar reflexão, o arquivo de configuração deve conter os mapeamentos dos nomes na interface para os nomes na classe de implementação. A fábrica pode permanecer independentemente em um pacote *framework* e não precisa ser alterada quando você adiciona novas implementações às suas opções de configuração.

Mesmo quando você não estiver usando uma linguagem que suporte reflexão, ainda vale a pena estabelecer um ponto central de configuração. Você pode até mes-

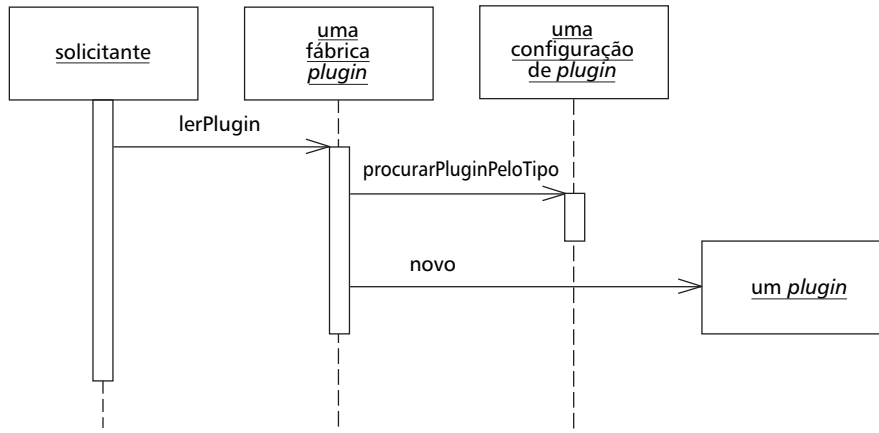


Figura 18.2 Um solicitante obtém uma implementação *Plugin* de uma interface separada.

mo usar um arquivo texto para estabelecer as regras de conexão, com a única diferença de que sua fábrica usará lógica condicional para mapear uma interface para a implementação desejada. Cada tipo de implementação deve ser contemplada na fábrica – na prática, isso não é nada demais. Apenas acrescente outra opção dentro do método fábrica sempre que você adicionar uma nova implementação ao código base. Para forçar as dependências de camada e de pacote com uma verificação em tempo de construção, coloque esta fábrica no seu próprio pacote para evitar interromper seu processo de construção.

Quando Usá-lo

Use *Plugin* toda vez que tiver comportamentos que requeiram implementações diferentes baseadas no ambiente de execução.

Exemplo: Um Gerador de Identidades (Java)

Como discutido acima, a geração de chave ou IDs é uma tarefa cuja implementação poderia variar entre ambientes de distribuição (Figura 18.3).

Primeiro, escreveremos a *Interface Separada* (445) GeradorDeId assim como quaisquer implementações necessárias.

```
interface GeradorDeId...

    public Long próximaId( );

class GeradorDeIdOracle implements GeradorDeId...

    public GeradorDeIdOracle ( ) {
        this.seqüência= Environment.getProperty("id.sequence");
        this.fontededados = Environment.getProperty("id.source");
    }
```

No GeradorDeIdOracle, `próximoId( )` seleciona o próximo número disponível de uma seqüência definida de uma fonte de dados definida.

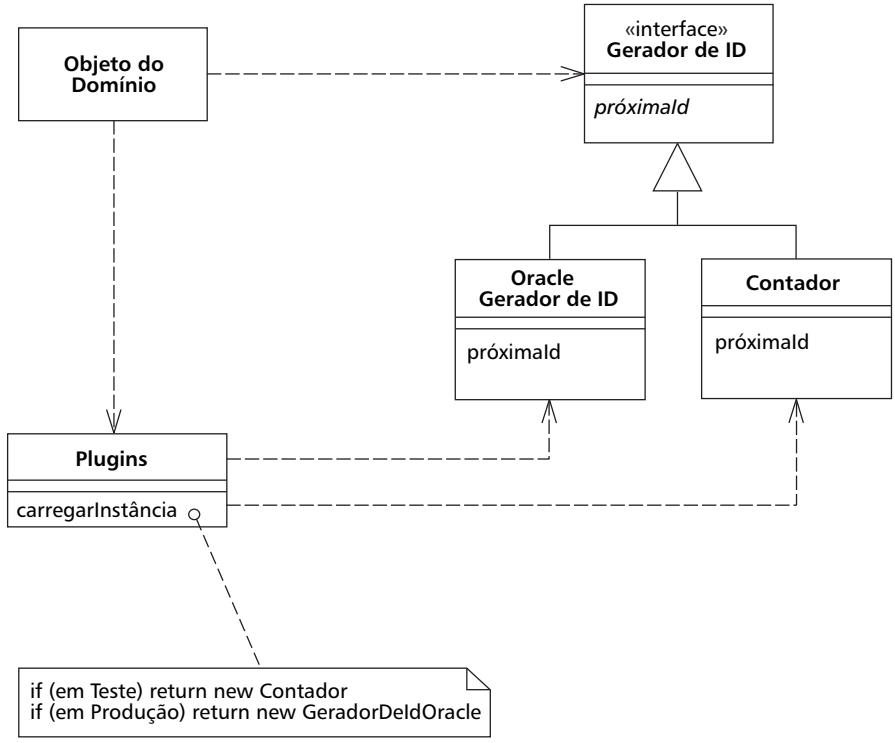


Figura 18.3 Múltiplos geradores de identidade.

```
class Contador implements GeradorDeId...

    private long contador = 0;
    public synchronized Long próximoId( ) {
        return new Long(contador++);
    }
```

Agora que temos algo para construir, vamos escrever a fábrica *plugin* que realizará os mapeamentos correntes da interface para a implementação.

```
class FábricaPlugin...
    private static Properties props = new Properties( );

    static {
        try {
            String arquivoDePropriedades = System.getProperty("plugins");
            propriedades.load (new FileInputStream(arquivoDePropriedades));
        } catch (Exception ex) {
            throw new ExceptionInInitializerError(ex);
        }
    }
    public static Object lerPlugin (Class iface) {
        String nomeDaImplementação = propriedades.getProperty(iface.getName( ));
        if (nomeDaImplementação == null) {
            throw new RuntimeException ("implementação não especificada para " +
```

```
                                iface.getName( ) + " nas propriedades da FábricaPlugin.");
        }
        try {
            return Class.forName(nomeDaImplementação).newInstance( );
        } catch (Exception ex) {
            throw new RuntimeException("fábrica incapaz de criar instância de " +
                                    iface.getName( ));
        }
    }
```

Perceba que estamos carregando a configuração procurando por uma propriedade de sistema chamada *plugins* que localizará o arquivo contendo nossas instruções de conexão. Existem muitas opções para definir e armazenar instruções de conexão, mas achamos que um simples arquivo de propriedades é a mais fácil. Usar a propriedade de sistema para encontrar o arquivo, em vez de olhar no *classpath*, torna simples especificar uma nova configuração em qualquer lugar da sua máquina. Isso pode ser muito conveniente ao mover construções (*builds*) entre ambientes de desenvolvimento, teste e produção. Aqui está como dois arquivos diferentes de configuração, um para teste e um para produção, poderiam se parecer:

```
arquivo de configuração teste.propriedades...

    #configuração para teste
    GeradorDeId = GeradorDeIdTeste

arquivo de configuração produção.propriedades...

    #configuração para produção
    GeradorDeId = GeradorDeIdOracle
```

Vamos voltar à interface GeradorDeId e adicionar um membro estático `INSTANCE` que é configurado por uma chamada à fábrica *Plugin*. Ela combina o *Plugin* com o padrão *singleton* para fornecer uma chamada extremamente simples e legível para obter um ID.

```
interface GeradorDeId...

    public static final GeradorDeId INSTANCIA =
        (GeradorDeId) FábricaPlugin.lerPlugin(GeradorDeId.class);
```

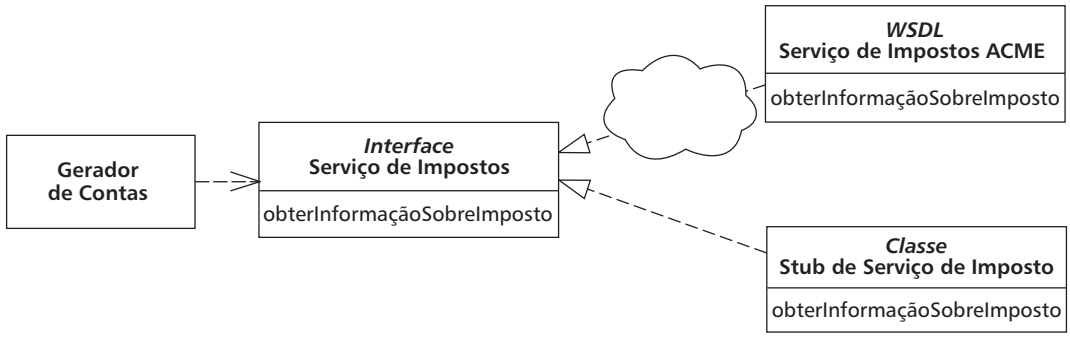
Podemos agora fazer essa chamada sabendo que obteremos o ID correto para o ambiente correto.

```
class Cliente extends ObjetoDoDomínio...

    private Cliente (String nome, Long id) {
        super(id);
        this.nome = nome;
    }
    public Cliente criar (String nome) {
        Long novaIdDoObjeto = GeradorDeId.INSTANCIA.próximaId( );
        Cliente obj = new Cliente(nome, novaIdDoObjeto);
        obj.marcarNovo( );
        return obj;
    }
```

Stub de Serviço (Service Stub)

por David Rice

Remove dependência de serviços problemáticos durante os testes.



Sistemas corporativos, muitas vezes, dependem do acesso a serviços de terceiros, tais como pontuação de crédito, busca por valores de taxas e mecanismos de determinação de preços. Qualquer desenvolvedor que já tenha criado tais sistemas pode falar da frustração de depender de recursos completamente fora de seu controle. A entrega dessas características é imprevisível, e como esses serviços são muitas vezes remotos, a confiabilidade e o desempenho podem sofrer também.

Esses problemas, no mínimo, diminuem a velocidade do processo de desenvolvimento. Os desenvolvedores ficam esperando pelo serviço voltar a operar ou talvez coloquem algo no código para compensar por características ainda por serem entregues. Muito pior, e bastante provável, tais dependências levarão a vezes em que os testes não poderão ser executados. Quando os testes não podem ser executados, o processo de desenvolvimento é interrompido.

Substituir durante os testes o serviço por um *Stub de Serviço* que rode em memória, rápida e localmente melhora a sua experiência de desenvolvimento.

Como Funciona

O primeiro passo é definir o acesso ao serviço com um *Gateway* (436). O *Gateway* (436) não deve ser uma classe, mas sim uma *Interface Separada* (445) de modo que você possa ter uma implementação que chame o serviço real e, pelo menos, uma que seja apenas um *Stub de Serviço*. A implementação desejada do *Gateway* (436) deve ser carregada usando um *Plugin* (465). A chave para escrever um *Stub de Serviço* é que você o mantenha tão simples quanto possível – a complexidade frustrará o seu intento.

Vamos investigar o processo de criar um *stub* para um serviço de impostos sobre vendas que, dado um endereço, tipo do produto e quantia vendida, fornece os valores e faixas dos impostos estaduais sobre vendas. A maneira mais simples de fornecer um *Stub de Serviço* é escrever duas ou três linhas de código que usem um valor de tributação simples que satisfaça todas as solicitações.

As leis sobre impostos não são tão simples, é claro. Determinados produtos são isentos de impostos em certos estados, então confiaremos que o nosso serviço real de impostos saiba quais combinações de estado e produto são isentas. Entretanto, mui-

to da funcionalidade da nossa aplicação depende de se os impostos são cobrados, então precisamos acomodar a isenção de impostos no nosso *Stub de Serviço*. O meio mais simples de adicionar este comportamento ao *stub* é por meio de uma declaração condicional que isente uma combinação específica de endereço e produto e, a seguir, use esses mesmos dados em quaisquer casos de teste relevantes. O número de linhas de código no nosso *stub* pode ainda ser contado com uma única mão.

Um *Stub de Serviço* mais dinâmico mantém uma lista de combinações isentas de produtos e estados, permitindo aos casos de teste acréscimo a essa lista. Mesmo aqui temos em torno de 10 linhas de código. Estamos mantendo as coisas simples dado nosso objetivo de aumentar a velocidade do processo de desenvolvimento.

O *Stub de Serviço* dinâmico traz uma questão interessante relacionada à dependência entre ele e os casos de teste. O *Stub de Serviço* depende de um método de configuração para adicionar isenções que não estejam na interface *Gateway* (436) original de serviço de impostos. Para tirar proveito de um *Plugin* (465) para carregar o *Stub de Serviço*, este método deve ser adicionado ao *Gateway* (436), o que é bom, pois ele não acrescenta muito ruído ao seu código e é feito em nome dos testes. Assegure-se de que a implementação do *Gateway* (436) que chama o serviço real levante falhas de asserção dentro dos métodos de teste.

Quando Usá-lo

Use o Stub de Serviço sempre que descobrir que a dependência de um determinado serviço esteja retardando seu desenvolvimento e testes.

Muitos adeptos da Programação Extrema usam o termo **Objeto Falso** para um *Stub de Serviço*. Permanecemos com *Stub de Serviço* porque ele tem estado em uso há mais tempo.

Exemplo: Serviço de Impostos Sobre Vendas (Java)

Nossa aplicação usa um serviço de impostos que é distribuído como um serviço Web. O primeiro item do qual cuidaremos é definir um *Gateway* (436), de modo que nosso código do domínio não seja forçado a lidar com os mistérios dos serviços Web. O *Gateway* (436) é definido como uma interface para facilitar a carga de quaisquer *Stubs de Serviço* que viermos a escrever. Usaremos um *Plugin* (465) para carregar a implementação correta do serviço de impostos.

```
interface ServiçoDeImpostos...

    public static final ServiçoDeImpostos INSTANCIA =
        (ServiçoDeImpostos) FábricaPlugin.lerPlugin(ServiçoDeImpostos.class);
    public InformaçãoSobreImposto obterInformaçãoSobreImpostoSobreVendas
        (String códigoDoProduto, Endereço endereço, Dinheiro quantiaVendida);
```

O *Stub de Serviço* para um cálculo simples de tributação se pareceria com:

```
class ServiçoDeImpostosTaxaPlana implements ServiçoDeImpostos...

    private static final BigDecimal TAXA_PLANA = new BigDecimal ("0.0500");
    public InformaçãoSobreImposto obterInformaçãoSobreImpostoSobreVendas
           (String códigoDoProduto, Endereço endereço, Dinheiro quantiaVendida) {
        return new InformaçãoSobreImposto (TAXA_PLANA, quantiaVendida.multiplica(TAXA_PLANA));
    }
```

Aqui está um *Stub de Serviço* que fornece isenções de impostos para uma combinação específica de endereço e produto:

```
class ServiçoDeImpostosProdutoIsento implements ServiçoDeImpostos...

    private static final BigDecimal TAXA_ISENTO = new BigDecimal ("0.0000");
    private static final BigDecimal TAXA_PLANA = new BigDecimal ("0.0500");
    private static final String ESTADO_ISENTO = "IL";
    private static final String PRODUTO_ISENTO= "12300";
    public InformaçãoSobreImposto obterInformaçãoSobreImpostoSobreVendas
                (String códigoDoProduto, Endereço endereço, Dinheiro quantiaVendida) {
        if (códigoDoproduto.equals(PRODUTO_ISENTO) &&
                endereço.lerCódigoDoEstado( ).equals(ESTADO_ISENTO)) {
            return new InformaçãoSobreImposto(TAXA_ISENTO, quantiaVendida.multiplica(TAXA_ISENTO));
        } else {
            return new InformaçãoSobreImposto(TAXA_PLANA, quantiaVendida.multiply(TAXA_PLANA));
        }
    }
```

Agora aqui está um *Stub de Serviço* mais dinâmico, cujos métodos permitem que um caso de teste adicione e remova combinações com isenção. Uma vez que achemos necessário adicionar estes métodos de teste, precisamos voltar e adicioná-los aos nossos *Stubs de Serviço* mais simples assim como à implementação que chama o serviço Web real para o cálculo de impostos. Os métodos de teste não usados deveriam todos causar falhas de asserção.

```
class TesteDeServiçoDeImpostos implements ServiçoDeImpostos...

    private static Set isenções = new HashSet( );
    public InformaçãoSobreImposto obterInformaçãoSobreImpostoSobreVendas
                (String códigoDoProduto, Endereço endereço, Dinheiro quantiaVendida) {
        BigDecimal taxa = lerTaxa(códigoDoProduto, endereço);
        return new InformaçãoSobreImposto (taxa, quantiaVendida.multiplica(taxa));
    }
    public static void adicionarIsenção(String códigoDoProduto, String códigoDoEstado) {
        isenções.add(lerChaveDaIsenção(códigoDoProduto, códigoDoEstado));
    }
    public static void zerar( ) {
        isenções.clear( );
    }
    private static BigDecimal lerTaxa(String códigoDoProduto, Endereço endereço) {
        if(isenções.contains(lerChaveDaIsenção(códigoDoproduto, endereço.lerCódigoDoEstado( )))) {
            return TAXA_ISENTO;
        } else {
            return TAXA_PLANA;
        }
    }
```

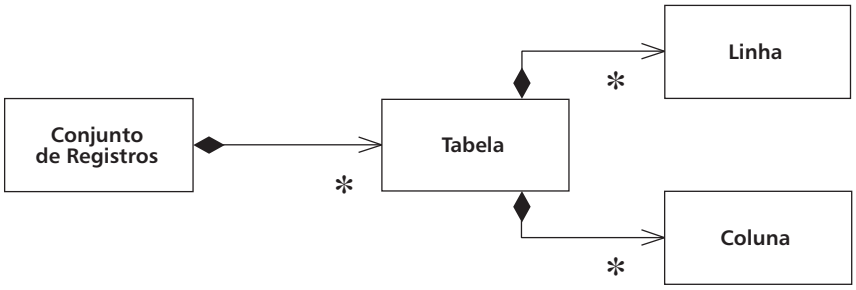
Não é mostrada a implementação que chama o serviço Web que fornece os dados reais sobre os impostos, a qual a configuração do *Plugin* (465) de produção ligaria a interface do serviço de impostos. As configurações do nosso *Plugin* (465) de teste apontariam para o *Stub de Serviço* apropriado acima.

Finalmente, qualquer solicitante do serviço de impostos deve acessar esse serviço através do *Gateway* (436). Temos aqui um gerador de contas que cria contas padrão e então chama o serviço de impostos para criar quaisquer impostos correspondentes.

```
class GeradorDeConta...

    public Conta[ ] calcularContas (ProgramaçãoDeFaturamento programação) {
        List contas= new ArrayList( );
        Conta contaBásica = new Conta(programação.lerQuantiaFaturada( ), false);
        contas.add(contaBásica);
        InformaçãoSobreImposto informação =
                ServiçoDeImpostos.INSTANCIA.obterInformaçãoSobreImpostoSobreVendas(
                programação.lerProduto( ), programação.lerEndereço( ),
                programação.lerQuantiaFaturada( ));
        if(informação.lerTaxaEstadual( ).compareTo(new BigDecimal(0)) > 0) {
            Conta impostoDaConta = new Conta(informação.lerQuantiaDoEstado( ), true);
            contas.add(impostoDaConta);
        }
        return (Conta[ ]) contas.toArray(new Conta[contas.size( )]);
    }
```

Conjunto de Registros (Record Set)

Uma representação em memória de dados tabulares.



Nos últimos vinte anos, o modo dominante de representar dados em um banco de dados tem sido a forma relacional tabular. Apoiado pelas grandes e pequenas companhias de banco de dados e uma linguagem de consultas razoavelmente padrão, quase todo novo desenvolvimento que vejo usa dados relacionais.

Além disso, tem surgido uma profusão de ferramentas para a construção rápida de interfaces com o usuário. Esses *frameworks* baseiam-se no fato de que os dados correspondentes são relacionais, e eles fornecem controles conectados aos dados (*data aware*) de vários tipos para a interface com o usuário. Tais controles permitem a visualização e manipulação desses dados quase que sem programação.

O lado ruim destes ambientes é que, embora tornem a exibição e as atualizações simples de dados ridiculamente fáceis, elas não oferecem um bom lugar onde colocar a lógica de negócio. Quaisquer validações além de "esta é uma data válida?", e quaisquer cálculos ou regras de negócio não têm um bom local para ir. Elas ficam ou jogadas no banco de dados como procedimentos armazenados ou misturadas com o código da interface com o usuário.

A idéia do *Conjunto de Registros* é dar a você o bolo e deixá-lo comer, fornecendo uma estrutura em memória que se pareça exatamente com o resultado de uma pesquisa SQL, mas que possa ser gerada e manipulada por outras partes do sistema.

Como Funciona

Um *Conjunto de Registros* é geralmente algo que você não irá construir sozinho, sendo fornecido pelo vendedor da plataforma de *software* com a qual você está trabalhando. Exemplos incluem o conjunto de dados do ADO.NET e o conjunto de linhas do JDBC 2.0.

O primeiro elemento essencial de um *Conjunto de Registros* é que ele se parece exatamente com o resultado de uma consulta a um banco de dados. Isso significa que você pode usar a abordagem clássica em duas camadas de emitir uma consulta e jogar os dados diretamente em uma interface com o usuário diretamente conectada aos dados (*data aware*) com toda a comodidade que essas ferramentas de duas camadas lhe dão. O segundo elemento essencial é que você mesmo pode criar facilmente um *Conjunto de Registros* ou pegar um que tenha resultado de uma consulta ao banco de dados e manipulá-lo facilmente com código de lógica de domínio.

Embora as plataformas, muitas vezes, lhe dêem um *Conjunto de Registros*, você mesmo pode criar um. O problema é que não há muito sentido nisso sem as ferramentas para as interfaces com o usuário conectadas aos dados (*data aware*) que, neste caso, você mesmo precisaria criar. De qualquer forma é razoável dizer que criar uma estrutura de *Conjunto de Registros* como uma lista de mapas, o que é comum em linguagens de *scripts* dinamicamente tipadas, é um bom exemplo deste padrão.

A habilidade de desconectar um *Conjunto de Registros* de sua conexão à fonte de dados é bastante valiosa. Isso lhe permite passar o *Conjunto de Registros* adiante em uma rede sem ter que se preocupar com as conexões ao banco de dados. Além disso, se você puder serializar facilmente o *Conjunto de Registros*, ele pode também atuar como um *Objeto de Transferência de Dados* (380) para uma aplicação.

A desconexão traz a questão do que acontece quando você atualiza o *Conjunto de Registros*. Cada vez mais, as plataformas estão permitindo que o *Conjunto de Registros* seja uma forma de *Unidade de Trabalho* (187), de modo que você pode modificá-lo e então retorná-lo à fonte de dados para que seja gravado. Uma fonte de dados pode tipicamente usar um *Bloqueio Offline Otimista* (392) para verificar se existe algum conflito e, em caso contrário, gravar as alterações no banco de dados.

Interface Explícita A maioria das implementações de *Conjuntos de Registros* usam uma **interface implícita**. Isso significa que, para extrair informações do *Conjunto de Registros*, você invoca um método genérico com um argumento para indicar qual campo você deseja. Por exemplo, para obter o passageiro de uma reserva de passagem aérea, você pode usar uma expressão como `umaReserva["passageiro"]`. Uma interface explícita requer uma classe Reserva real com métodos e propriedades definidos. Com uma reserva explícita, a expressão para o passageiro poderia ser `umaReserva.passageiro`.

As interfaces implícitas são flexíveis, visto que você pode usar um *Conjunto de Registros* genérico para qualquer tipo de dados. Isso lhe poupa de ter que escrever uma nova classe cada vez que definir um novo tipo de *Conjunto de Registros*. Em geral, no entanto, considero interfaces implícitas uma Coisa Ruim. Se estou programando uma reserva, como sei como obter o passageiro? A *string* apropriada é "passageiro" ou "convidado" ou "viajante"? O único modo pelo qual posso descobrir é andar pelo código base tentando encontrar onde as reservas são criadas e usadas. Se eu tiver uma interface explícita, posso olhar a definição da reserva para ver a propriedade de que preciso.

Este problema é exacerbado em linguagens tipadas estaticamente. Se eu quiser o sobrenome do passageiro, tenho que recorrer a alguma expressão horrível tal como `((Pessoa)umaReserva["passageiro"]).sobrenome`, mas então o compilador perde toda a informação sobre os tipos e tenho que entrar manualmente com ela para obter a informação que quero. Uma interface explícita pode manter a informação sobre os tipos, então posso usar `umaReserva.passageiro.sobrenome`.

Por esses motivos, geralmente franzo as sobrancelhas para interfaces implícitas (e também para seu primo malvado, passar dados em dicionários). Também não sou muito fã deles com *Conjuntos de Registros*, mas o que se salva aqui é que o *Conjunto de Registros* geralmente carrega nele informação sobre as colunas válidas. Além disso, os nomes de colunas são definidos pelo SQL que cria o *Conjunto de Registros*, então não é muito difícil encontrar as propriedades quando você precisa delas.

Contudo, é bom dar um passo adiante e ter uma interface explícita. ADO.NET fornece isso com seus conjuntos de dados fortemente tipados, classes geradas que

fornecem uma interface explícita e completamente tipada para um *Conjunto de Registros*. Já que um conjunto de dados ADO.NET pode conter muitas tabelas e os relacionamentos entre elas, conjuntos de dados fortemente tipados também fornecem propriedades que podem usar essas informações sobre relacionamentos. As classes são geradas a partir do XSD da definição do conjunto de dados.

As interfaces implícitas são mais comuns, então usei conjuntos de dados não-tipados nos exemplos deste livro. Para código de produção em ADO.NET, todavia, sugiro o uso de conjuntos de dados que sejam tipados. Em um ambiente diferente do ADO.NET, sugiro o uso de geração de código para seus próprios *Conjuntos de Registros* explícitos.

Quando Usá-lo

Na minha opinião, o valor de um *Conjunto de Registros* vem de se ter um ambiente que se baseia nele como uma forma comum de manipular dados. Várias ferramentas de interface com o usuário usam *Conjuntos de Registros*, e esta é uma razão convincente para usá-los você também. Se você tiver um desses ambientes, deve usar um *Módulo Tabela* (134) para organizar a sua lógica de domínio: Obtenha um *Conjunto de Registros* do banco de dados, passe-o para um *Módulo Tabela* (134) para calcular informações derivadas, passe-o para uma interface com o usuário para exibi-lo e editar seus dados e passe-o de volta para um *Módulo Tabela* (134) para validação. A seguir grave as atualizações no banco de dados.

De diversas maneiras, as ferramentas que tornam o *Conjunto de Registros* tão valioso apareceram devido à onipresença dos bancos de dados relacionais e SQL e à ausência de quaisquer alternativas reais de estrutura e linguagem de pesquisa. Agora, é claro, existe o XML, que tem uma estrutura amplamente padronizada e uma linguagem de pesquisa em XPath, e acredito ser provável que venhamos a ver ferramentas que usem uma estrutura hierárquica da mesma forma que as ferramentas atuais usam *Conjuntos de Registros*. Talvez isso seja na verdade um caso particular de um padrão mais genérico: algo como *Estrutura de Dados Genérica*. Mas deixarei de pensar nesse padrão até lá.

Esta página foi deixada em branco intencionalmente.

Referências

[Alexander et al.]
Alexander, et al. A *Pattern Language*. Oxford, 1997.
Uma inspiração para muitas pessoas no movimento de padrões. Sou menos fascinado por ele do que a maioria, mas penso que vale a pena dar uma olhada para entender uma abordagem de onde tantos extraem tanto.

[Alpert et al.]
Alpert, Brown and Woolf. *Design Patterns Smalltalk Companion.* Addison-Wesley, 1998
Pouco conhecido fora da comunidade Smalltalk, este livro espande e explica muitos dos padrões clássicos.

[Alur et al.]
Alur, Culpi and Malks. *Core J2EE Patterns: Best Practices and Design Strategies.* Prentice Hall, 2001.
Pertence à nova onda de livros sobre padrões que dão nova vida à área. Embora os padrões sejam expressos especificamente para a plataforma J2EE, a maioria deles também faz sentido em outros lugares.

[Ambler]
http://www.ambysoft.com/mappingObjetct.html
Uma fonte de idéias úteis a respeito do mapeamento objeto-relacional.

[Beck XP 2000]
Beck, *Extreme Programming Explained. Addison-Wesley, 2000.**
O manifesto da Programação Extrema. Deve ser lido por qualquer pessoa interessada em processos de *software*.

[Beck Patterns]
Beck. *Smalltalk Best Practice Patterns.* Prentice Hall, 1997.
De forma injusta, é pouco lido devido à sua base Smalltalk. Tem mais bons conselhos para outras linguagens OO do que a maioria dos livros que são escritos especificamente para elas. O único aspecto negativo é que você vai perceber o quanto todos nós perdemos por não programar em Smalltalk.

* N. de R. As obras assinaladas nesta lista foram publicadas no Brasil pela Bookman Editora.

[Beck TDD]
Beck. *Test-Driven Development: By Example.* Addison-Wesley, 2003.
Deve ser lançado no mesmo dia deste livro. TDD é o guia do Kent para o apertado ciclo de teste e refatoração que pode envolver um projeto.

[Bernstein e Newcomer]
Bernstein and Newcomer. *Principles of Transaction Processing.* Morgan Kaufmann, 1997.
Uma excelente introdução ao complicado mundo das transações.

[Brown et al.]
Brown et al. *Enterprise Java Programming with IBM Websphere.* Addison-Wesley, 2001.
Embora dois terços deste livro sejam um manual de *software*, o outro terço contém uma quantidade maior de bons conselhos sobre projeto do que a maioria dos livros inteiros dedicados ao assunto.

[Brown and Whitenack]
http://members.aol.com/kgb1001001/Chasms.html
Um dos primeiros e melhores artigos sobre mapeamento objeto-relacional.

[Cockburn UC]
Cockburn. *Writing Effective Use Cases**. Addison-Wesley, 2001.
De longe a melhor referência sobre casos de uso.

[Cockburn PloP]
Cockburn, "Prioritizing Forces in Software Design," in [PLoPD 2].
Uma discussão sobre as fronteiras de uma aplicação.

[Coleman et al.]
Coleman, Arnold and Bodoff. *Object-Oriented Development: The Fusion Method, Second Edition.* Prentice Hall, 2001.
Embora muito deste livro pré-UML seja principalmente de interesse histórico, sua discussão do modelo de interface é muito útil para quem esteja projetando uma camada de serviço.

[Evans and Fowler]
http://martinfowler.com/apsupp/spec.pdf
Uma discussão sobre o padrão Especificação.

[Evans]
Evans. *Domain Driven.* Addison-Wesley, em preparação.
Um livro sobre o desenvolvimento de modelos de domínio. Embora eu normalmente não goste de referenciar livros ainda não publicados, o manuscrito promete uma discussão fascinante sobre um aspecto importante e difícil do desenvolvimento de aplicações corporativas.

[Fowler Temporal Patterns]
http://martinfowler.com/ap2/timeNarrative.html
Padrões que lidam com histórias de objetos que mudam com o tempo.

[Fowler AP]
Fowler. *Padrões de Análise*. Addison-Wesley, 1997.
Padrões para o modelo de domínio.

[Fowler Refactoring]
Fowler, *Refactoring**. Addison-Wesley, 1999.
Uma técnica para aperfeiçoar o projeto de uma base de código existente.

[Fowler CI]
http://martinfowler.com/articles/continuousIntegration.html
Um ensaio que explica como criar *software*, automaticamente, diversas vezes por dia.

[Gang of Four]
Gamma, Helm, Johnson e Vlissides. *Design Patterns*.* Addison-Wesley, 1995.
O livro seminal sobre padrões.

[Hay]
Hay. *Data Model Patterns*. Dorset House, 1995.
Padrões de modelos conceituais a partir de uma perspectiva relacional.

[Jacobson et al.]
Jacobson et al. *Object-Relational Software Engeneering*. Addison-Wesley, 1992.
Um dos primeiros livros sobre o projeto OO. Introduz casos de uso e a abordagem de projeto interface-controlador-entidade.

[Keller and Coldeway]
http://www.objectarchitects.de/ObjectArchitects/orpatterns/index.htm
Uma referência excelente sobre mapeamento objeto-relacional

[Kirtland]
Kirtland. *Designing Component-Based Applications*. Microsoft Press, 1998.
Descrição da arquitetura DNA.

[Knight and Dai]
Knight and Dai. *"Objects and the Web." IEEE Software*, March/April 2002.
Um excelente artigo sobre o padrão modelo-vista-controle, sua evolução e uso em aplicações Web.

[Larman]
Larman. *Applying UML and Patterns, Second Edition*.* Prentice Hall, 2001.
Atualmente meu favorito para uma introdução ao projeto OO.

[Lea]
Lea. *Concurrent Programming in Java, Second Edition*. Addison-Wesley, 2000.
Se você quiser programar com múltiplas *threads*, precisa entender este livro primeiro.

[Marinescu]
Marinescu. *EJB Design Patterns*.* New York: John Wiley, 2002.
Um livro recente sobre padrões para Java EJBs.

[Martin and Odell]
Martin and Odell. *Object Oriented Methods: A Foundation (UML Edition)*. Prentice Hall, 1998.
Modelagem de objetos de uma perspectiva conceitual, assim como uma investigação sobre os fundamentos de que trata a modelagem.

[Nilsson]
Nilsson. *.NET Enterprise Design with Visual Basic .NET and SQL Server 2000*. Sams, 2002.
Um livro sólido sobre arquitetura para a plataforma da Microsoft.

[PLoPD 2]
Vlissides, Coplien and Kerth (eds.). *Pattern Languages of Program Design 2*. Addison-Wesley, 1996.
Compêndio de artigos sobre padrões.

[PLoPD 3]
Martin, Buschmann and Rielhe (eds.). *Pattern Languages of Program Design 3*. Addison-Wesley, 1998.
Compêndio de artigos sobre padrões.

[POSA]
Buschmann et al. *Pattern-Oriented Software Architecture*. Wiley, 2000.
O melhor livro sobre padrões de arquitetura mais amplos.

[Rielhe at al.]
Rielhe, Siberski, Baumer, Megert and Zullighoven. "Serializer," in [PLoPD 3].
Descrição detalhada das estruturas de serialização de objetos, especialmente quando você precisa serializá-los em diferentes formas.

[Schmidt]
Schmidt, Stal, Rohnert and Buschmann. *Pattern-Oriented Software Architecture, Volume 2.* New York: John Wiley, 2000.
Padrões para sistemas concorrentes e distribuídos. Mais para pessoas que projetam servidores de aplicação do que para aquelas que os usam, mas é bom ter algum conhecimento destas idéias quando você usa os resultados.

[Snodgrass]
Snodgrass. *Developing Time-Oriented Database Applications in SQL.* Morgan-Kaufmann, 1999.
Como lidar com o rastreamento de informações históricas em bancos de dados relacionais.

[Struts]
http://jakarta.apache.org/struts/
Um *framework* para a construção de aplicações Web em Java que vem crescendo em popularidade.

[Waldo et al.]
Waldo, Wyant, Wollrath and Kendall. *A Note on Distributed Computing.* SMLI TR-94-29, *http://research.sun.com/technical-reports/1994/smli_tr-94-29.pdf,* Sun Microsystems, 1994.
Um artigo clássico sobre por que "objetos distribuídos transparentes" são um paradoxo perigoso.

[wiki]
http://c2.com/cgi/wiki
O *wiki* web original, desenvolvido por Ward Cunningham. Um *site* Web sinuoso, porém fascinante, onde todos os tipos de pessoas compartilham todos os tipos de idéias.

[Woolf]
Woolf. "Null Object," in [PLoPD 3].
Uma descrição do padrão Objeto Nulo.

[Yoder]
http://www.joeyoder.com/Research/objectmappings
Uma boa fonte de padrões objeto-relacionais.

Índice

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W

X

Lista de Padrões

Bloqueio de Granularidade Alta (Coarse-Grained Lock) (412): Bloqueia um conjunto de objetos relacionados utilizando para isso um único bloqueio.

Bloqueio Implícito (Implicit Lock) (422): Permite ao código de uma camada supertipo ou de um *framework* obter bloqueios *offline*.

Bloqueio *Offline* Otimista (Optimistic Offline Lock) (392): Previne conflitos entre transações de negócio concorrentes detectando um conflito e desfazendo a transação.

Bloqueio *Offline* Pessimista (Pessimistic Offline Lock) (401): Previne conflitos entre transações de negócio concorrentes permitindo que apenas uma transação de negócio acesse os dados de cada vez.

Camada de Serviço (Service Layer) (141): Define a fronteira de uma aplicação com uma camada de serviços que estabelece um conjunto de operações disponíveis e coordena a resposta da aplicação em cada operação.

Camada Supertipo (Layer Supertype) (444): Um tipo que atua como o supertipo para todos os tipos nesta camada.

Campo Identidade (Identity Field) (215): Guarda o campo ID de um banco de dados em um objeto para manter a identidade entre um objeto na memória e uma linha do banco de dados.

Carga Tardia (Lazy Load) (200): Um objeto que não contém todos os dados de que você precisa, mas que sabe como obtê-los.

Caso Especial (Special Case) (462): Uma subclasse que fornece comportamento especial para casos particulares.

Conjunto de Registros (Record Set) (473): Uma representação em memória de dados tabulares.

Controlador de Aplicação (Application Controller) (360): Um ponto centralizado para manipular a navegação de tela e o fluxo de uma aplicação.

Controlador de Página (Page Controller) (318): Um objeto que trata uma requisição para uma página ou ação específica em um *site* Web.

Controlador Frontal (Front Controller) (328): Um controlador que trata todas as requisições para um *site* Web.

Dinheiro (Money) (455): Representa um valor monetário.

Estado da Sessão no Banco de Dados (Database Session State) (432): Armazena dados de sessão no banco de dados.

Estado da Sessão no Cliente (Client Session State) (427): Armazena o estado da sessão no cliente.

Estado da Sessão no Servidor (Server Session State) (429): Mantém o estado da sessão em um sistema servidor de forma serializada.

Fachada Remota (Remote Façade) (368): Fornece uma fachada de granularidade alta sobre objetos de granularidade baixa para melhorar a eficiência em uma rede.

Gateway (436): Um objeto que encapsula o acesso a um sistema ou recurso externo.

Gateway de Linha de Dados (Row Data Gateway) (158): Um objeto que atua como um Gateway (436) para um único registro em uma fonte de dados. Existe uma instância por linha.

Gateway de Tabela de Dados (Table Data Gateway) (151): Um objeto que atua como um Gateway (436) para uma tabela de um banco de dados. Uma instância lida com todas as linhas na tabela.

Herança de Tabela Concreta (Concrete Table Inheritance) (283): Representa um hierarquia de herança de classes com uma tabela por classe concreta na hierarquia.

Herança de Tabela de Classes (Class Table Inheritance) (276): Representa uma hierarquia de herança de classes com uma tabela por classe.

Herança de Tabela Única (Single Table Inheritance) (269): Representa uma hierarquia de herança de classes como uma única tabela que tem colunas para todos os campos das diversas classes.

Interface Separada (Separated Interface) (445): Define uma interface em um pacote separado de sua implementação.

LOB Serializado (Serialized LOB) (264): Grava um grafo de objetos serializando-os em um único objeto grande (LOB – Large OBject), o qual é armazenado em um campo do banco de dados.

Mapa de Identidade (Identity Map) (196): Garante que cada objeto seja carregado apenas uma vez mantendo cada objeto carregado em um mapa. Procura pelos objetos usando o mapa, quando há uma referência para eles.

Mapeador (Mapper) (442): Um objeto que estabelece uma comunicação entre dois objetos independentes.

Mapeador de Dados (Data Mapper) (170): Uma camada de *Mapeadores* (442) que move dados entre objetos e um banco de dados enquanto os mantém independentes uns dos outros e do próprio mapeador.

Mapeadores de Herança (Inheritance Mappers) (291): Uma estrutura para organizar mapeadores de bancos de dados que tratam hierarquias de herança.

Mapeamento de Chave Estrangeira (Foreign Key Mapping) (233): Mapeia uma associação entre objetos para uma referência de chave estrangeira entre tabelas.

Mapeamento Dependente (Dependent Mapping) (256): Faz uma classe executar o mapeamento no banco de dados de uma classe filha.

Mapeamento em Metadados (Metadata Mapping) (295): Armazena os detalhes do mapeamento objeto-relacional em metadados.

Mapeamento de Tabela Associativa (Association Table Mapping) (244): Grava uma associação como uma tabela com chaves estrangeiras para as tabelas que estejam vinculadas pela associação.

Modelo de Domínio (Domain Model) (126): Um modelo de objetos do domínio que incorpora tanto o comportamento quanto os dados.

Modelo Vista Controlador (Model View Controller) (315): Divide a interação da interface com o usuário em três papéis distintos.

Módulo Tabela (Table Module) (134): Uma instância única que trata a lógica de negócio para todas as linhas de uma tabela ou visão em um banco de dados.

Objeto de Pesquisa (Query Object) (304): Um objeto que representa uma consulta ao banco de dados.

Objeto de Transferência de Dados (Data Transfer Object) (380): Um objeto que transporta dados entre processos de modo a reduzir o número de chamadas de métodos.

Objeto Valor (Value Object) (453): Um objeto pequeno e simples, como dinheiro ou um intervalo de datas, cuja igualdade não é baseada em identidade.

Plugin (465): Conecta classes durante a configuração, em vez de na compilação.

Registro (Registry) (448): Um objeto conhecido que outros objetos podem usar para encontrar objetos e serviços comuns.

Registro Ativo (Active Record) (165): Um objeto que encapsula uma linha, em uma tabela ou visão de um banco de dados e o acesso ao banco de dados e adiciona lógica de domínio a esses dados.

Repositório (Repository) (309): Faz a mediação entre as camadas de domínio e de mapeamento de dados usando uma interface do tipo coleção para acessar os objetos do domínio.

Roteiro de Transação (Transaction Script) (120): Organiza a lógica de negócio em procedimentos em que cada procedimento trata uma única solicitação da apresentação.

Stub de Serviço (Service Stub) (469): Remove a dependência de serviços problemáticos durante os testes.

Unidade de Trabalho (Unit of Work) (187): Mantém uma lista de objetos afetados por uma transação de negócio e coordena a gravação das alterações e a resolução de problemas de concorrência.

Valor Embutido (Embedded Value) (261): Mapeia um objeto em diversos campos da tabela de outro objeto.

Vista de Transformação (Transform View) (343): Uma visão que processa dados do domínio elemento por elemento e os transforma em HTML.

Vista em Duas Etapas (Two Step View) (347): Transforma dados do domínio em HTML em duas etapas: primeiro, formando algum tipo de página lógica e, depois, representando essa página lógica em HTML.

Vista Padrão (Template View) (333): Representa informações em HTML inserindo marcadores em uma página HTML.

Cola

Aviso: Esta discussão de questões está muito simplificada!

Como estruturo minha lógica de domínio?

A lógica é simples → *Roteiro de Transação* (120)
A lógica é complexa → *Modelo de Domínio* (126)
A lógica é moderada e há boas ferramentas para um *Conjunto de Registros* (473) → *Módulo Tabela* (134)

Como disponibilizo uma API distinta para minha lógica de domínio?

→ *Camada de Serviço* (141)

Como estruturo uma apresentação Web?

→ *Modelo Vista Controlador* (315)

Como organizo meu processamento de solicitações HTTP?

Tenho um fluxo de aplicação simples, em sua maioria URLs diretas → *Controlador de Página* (318)
Tenho um fluxo complicado → *Controlador Frontal* (328)
Tenho necessidade de internacionalização ou de políticas flexíveis de segurança → *Controlador Frontal* (328)

Como controlo a formatação das minhas páginas Web?

Gosto de editar a página e colocar ganchos para dados dinâmicos → *Vista Padrão* (333)
Vejo a página como uma transformação de dados do domínio – provavelmente em XML → *Vista de Transformação* (343)
Preciso fazer alterações gerais na aparência do meu *site* → *Vista em Duas Etapas* (347)
Preciso de múltiplas apresentações para o mesmo formato lógico de tela → *Vista em Duas Etapas* (347)

Como controlo um fluxo de aplicação complexo?

→ *Controlador de Aplicação* (360)

Como interajo com o banco de dados?

Estou usando um *Roteiro de Transação* (120) → *Gateway de Linha de Dados* (158)
Estou usando um *Roteiro de Transação* (120) e minha plataforma tem um bom suporte para um *Conjunto de Registros* (473) → *Gateway de Tabela de Dados* (151)
Tenho um *Modelo de Domínio* (126) que tem forte correspondência com minhas tabelas de banco de dados → *Registro Ativo* (165)
Tenho um *Modelo de Domínio* (126) rico → *Mapeador de Dados* (170)

Como posso assegurar que não atualizarei os mesmos dados do banco de dados em diferentes lugares da memória?

→ *Mapa de Identidade* (196)

Como mantenho meus objetos de domínio conectados a registros do banco de dados?

→ *Campo Identidade* (215)

Como minimizo o código para mapear meus dados de domínio para o banco de dados?

→ *Mapeamento de Metadados* (295)

Como posso pesquisar o banco de dados em termos do meu Modelo de Domínio (116)?

→ *Objeto de Pesquisa* (304)

Como armazeno associações entre os objetos no banco de dados?

Tenho uma única referência para um objeto → *Mapeamento de Chave Estrangeira* (233)

Tenho um conjunto de objetos → *Mapeamento de Chave Estrangeira* (233)

Tenho um relacionamento muitos para muitos → *Mapeamento de Tabela de Associação* (244)

Tenho uma coleção de objetos que não são compartilhados com outros → *Mapeamento Dependente* (256)

Tenho um campo que é um *Objeto Valor* (453) → *Valor Embutido* (261)

Tenho uma rede complexa de objetos que não são usados por outras partes do banco de dados → *LOB Serializado* (264)

Como evito carregar todo o banco de dados na memória?

→ *Carga Tardia* (200)

Como armazeno estruturas de herança em um banco de dados relacional?

→ *Herança de Tabela Única* (269)

Uma única tabela agirá como um gargalo em acessos ao banco de dados → *Herança de Tabela de Classes* (276)

Uma única tabela desperdiçará espaço no banco de dados → *Herança de Tabela de Classes* (276)

Terei muitas junções e mesmo assim não quero uma tabela única → *Herança de Tabela Concreta* (283)

Como mantenho registro dos objetos que li ou alterei?

→ *Unidade de Trabalho* (187)

Como posso confirmar (*commit*) minhas alterações no banco de dados dentro de uma única requisição do cliente?

→ *Bloqueio Offline Otimista* (392)

Não posso deixar um usuário perder seu trabalho → *Bloqueio Offline Pessimista* (401)

Como posso bloquear (*lock*) um conjunto de objetos relacionados usando um único bloqueio?

→ *Bloqueio de Granularidade Alta* (412)

Como posso evitar comprometer minha estratégia de bloqueio?

→ *Bloqueio Implícito* (422)

Como posso acessar remotamente objetos de granularidade baixa?

→ *Fachada Remota* (368)

Como passo dados de muitos objetos em uma única chamada remota?

→ *Objeto de Transferência de Dados* (380)

Como armazeno estado no meio de uma transação de negócio com estado?

→ Não há muitos estados → *Estado da Sessão no Cliente* (427)

→ Existem muitos estados → *Estado da Sessão no Servidor* (429)

→ Posso gravar o trabalho em progresso no banco de dados → *Estado da Sessão no Banco de Dados* (432)