Object Oriented SELECTION

Patterns of Enterprise Application Architecture

エンタープライズアプリケーションアーキテクチャパターン

頑強なシステムを実現するためのレイヤ化アプローチ

マーチン・ファウラー 著
長瀬嘉秀 監訳
株式会社テクノロジックアート 訳

SE SHOEISHA

本書内容に関するお問い合わせについて

このたびは翔泳社の書籍をお買い上げいただき、誠にありがとうございます。弊社では、読者の皆様からのお問い合わせに適切に対応させていただくため、以下のガイドラインへのご協力をお願い致しております。下記項目をお読みいただき、手順に従ってお問い合わせください。

●ご質問される前に

弊社 Web サイトの「正誤表」をご参照ください。これまでに判明した正誤や追加情報が掲載されています。

正誤表　　http://www.seshop.com/book/errata/

●ご質問方法

弊社 Web サイトの「出版物 Q&A」をご利用ください。

出版物 Q&A　　http://www.seshop.com/book/qa/

インターネットをご利用でない場合は、FAX または郵便にて、下記“編集部読者サポート係”までお問い合わせください。

電話でのご質問は、お受けしておりません。

●回答について

回答は、ご質問いただいた手段によってご返事申し上げます。ご質問の内容によっては、回答に数日ないしはそれ以上の期間を要する場合があります。

●ご質問に際してのご注意

本書の対象を越えるもの、記述個所を特定されないもの、また読者固有の環境に起因するご質問等にはお答えできませんので、予めご了承ください。

●郵便物送付先および FAX 番号

送付先住所：〒 160-0006 東京都新宿区舟町 5

FAX 番号：03-5362-3818

宛先：（株）翔泳社 編集部読者サポート係

※ 本書に記載された URL 等は予告なく変更される場合があります。

※ 本書の出版にあたっては正確な記述につとめましたが、著者、訳者、監訳者、出版社などのいずれも、本書の内容に対してなんらかの保証をするものではなく、内容やサンプルに基づくいかなる運用結果に関してもいっさいの責任を負いません。

※ 本書に掲載されているサンプルプログラムやスクリプト、および実行結果を記した画面イメージなどは、特定の設定に基づいた環境にて再現される一例です。

※ 本書に記載されている会社名、製品名はそれぞれ各社の商標および登録商標です。

※ 本書では ® および ™ は割愛させていただいております。

Patterns of Enterprise Application Architecture by Martin Fowler

Copyright © 2003 by Pearson Education, Inc.

Japanese translation rights arranged with Pearson Education, Inc. through Japan UNI Agency, Inc., Tokyo.

デニス・ウィリアム・ファウラー（1922 − 2000）に捧げる

日本の読者の皆様へ

短い滞在期間ではありますが、日本への訪問をいつも楽しんでいます。特に、ソフトウェア開発に携わる日本の方々のあいだで私の仕事が広く認められ、役立っていると実感することができて、とても嬉しく思います。今回、私の最新の書籍がとうとう日本でも出版されると聞いて喜びにたえません。本書の日本での刊行に尽力してくださったすべての方々、本書を読んでくださる読者の方々にお礼を申し上げます。

マーチン・ファウラー

監訳者まえがき

本書の原著（英語）が2002年に発行されたときには衝撃が走るほど米国のオブジェクト指向業界にインパクトを与えた。オブジェクト指向だけにとどまらず、アーキテクチャとしてJ2EEや.NETといったミドルウェアを利用している開発者にも注目されている。

本書の内容はシステム分析による作業と実装作業のギャップを埋める方法であり、誰もが悩んでいた技術である。このノウハウをマーチン・ファウラー氏がパターンとしてまとめ上げた。ファウラー氏は、UMLからエクストリームプログラミングまで幅広い活動を行っている。このため、本書の著者は様々な方面から集まっている。まさに、バランスのとれたファウラー氏ならではの人選である。

ファウラー氏と私は、2000年の日本での講演のアテンドを担当した頃からのつきあいである。ファウラー氏とはUMLによるモデリング、パターン、アジャイルプロセスについて、多くの議論を行ってきた。これからも彼（または彼の所属しているThoughtWorks社）とは共同作業をすることもあるであろう。

本書の翻訳にあたって、日本での発売出版社が変更になったり、翻訳会社にお手伝いをお願いしたりと多くの時間を費やしてしまった。日本語版の出版が遅くなってしまったことについて、誠に申し訳なく思っている。

最後に、本書を全体的にチェックした武田知子氏、テクニカルなレビューを行った永田渉氏、坂本武志氏、藤川幸一氏に感謝する。また、株式会社翔泳社の佐藤善昭氏をはじめとするスタッフの人たちには、難しい編集作業をしていただいた。本書によりシステム開発の悩みが少しでも減るように願っている。

2005年2月

長瀬 嘉秀

まえがき

1999年の春、私はシカゴに飛んだ。規模は小さいが成長著しいアプリケーション開発会社ThoughtWorks社でのプロジェクトについて打ち合わせをするためだ。そのプロジェクトとは、最近よくある野心的なエンタープライズアプリケーションプロジェクトで、バックエンドリーシングシステムだった。基本的に、リース契約後に生じるすべての事例を処理するためのもので、請求書の送付、リース物件の更新処理、リース料を期日までに支払わない顧客の追跡、物件を予定より前に返却してきた場合の処理などを行う。これだけであれば、さほど面倒なものには聞こえないのだが、実際には、リース契約の内容は契約ごとに異なり、恐ろしいぐらいに複雑だ。ビジネス「ロジック」がロジックパターンにあてはまることはめったにない。このロジックはビジネスを成功させようとする人によって作られたものであり、ビジネスではわずかな条件の変化でも契約の獲得が大きく左右されることがあるからだ。したがって、契約を勝ち取るたびにそのシステムはさらに複雑なものになっていくことになる。

このような状況は私のやる気を奮い立たせた。複雑な事情を踏まえ、さまざまな問題をスムーズに処理できるシステムを構築するのはやりがいのある仕事だからだ。実際、複雑なロジックをより扱いやすいものにすることには意義がある。複雑なビジネス処理に対応した適正な**ドメインモデル**を開発するのは、非常に困難ではあるが、とてもやりがいのあることである。

しかし、問題はこれだけではなかった。このドメインモデルではデータベースとの連携が必須で、その他多くのプロジェクト同様、リレーショナルデータベースが使用されていた。また、このモデルをユーザインタフェースに接続して、中のソフトウェアをリモートアプリケーションから利用できるようにすることと、サードパーティのパッケージとこのソフトウェアを統合させる必要もあった。これらすべてを新しい技術であるJ2EEを使って処理するわけで、その時点では、世界中のどこにもこの技術を実践的に導入している事例はなかった。

この技術そのものは新しいものだったが、私たちにはそれなりの経験があった。私自身は、この種のプロジェクトをC++、Smalltalk、CORBAなどで長年手掛けており、

ThoughtWorkers 社の社員の多くは Forte に関する経験が豊富だった。私たちの頭の中にはキーとなるアーキテクチャ構想がすでにあり、後は、それらをどうやって J2EE に適用するかを考えればいいだけだった。あれから 3 年経った今、振り返ると、設計は決して完全なものではないが、その時点では立派に機能していたのだ。

本書の目的は、このような状況に対処することである。長年に渡って数多くのエンタープライズアプリケーションプロジェクトを見てきたが、これらのプロジェクトの多くには、エンタープライズアプリケーションが抱える複雑性への対処に一定の効果を上げている共通した設計構想が含まれている。本書は、これらの設計構想をパターンとしてとらえるための出発点となるものだ。

本書は 2 部構成になっており、第 1 部「概論」の各章では、エンタープライズアプリケーションを設計する上で重要なトピックの数々を紹介している。これらの章では、エンタープライズアプリケーションのアーキテクチャに関するさまざまな問題とその解決策を解説している。第 2 部では、その解決策について詳細に解説し、パターン化してまとめている。これらのパターンはあくまでも参考資料であり、すべてのパターンを順番に読む必要はない。本書の正しい使い方は、まず、第 1 部「概論」の各章を最初から最後まで順番に読み、本書が対象としている内容の概略を理解し、興味または必要に応じて第 2 部の「パターン」に進むという方法である。つまり、本書は、短い概論と長いリファレンスを一冊にまとめたものだと考えてもらえればいい。

本書の内容は、エンタープライズアプリケーション設計についてである。エンタープライズアプリケーションとは、膨大で、ときに複雑なデータの表示・処理・格納と、それらのデータを使ったビジネスプロセスのサポートまたは自動化を行うためのものだ。紹介している例には、予約システム、財務システム、サプライチェーンシステム、その他最新のビジネス環境で使用される多くのシステムが含まれている。エンタープライズアプリケーションには固有の課題と解決策があり、組み込みシステムや制御システム、電気通信、またはオフィスアプリケーションとは異なっている。これらの分野の業務をしている人にとっては、本書はあまり役に立たないかもしれない（エンタープライズアプリケーションがどんなものなのか感じをつかみたいという場合は別だが）。ソフトウェアアーキテクチャ全般に関する本としては、[POSA]を推奨したい。

エンタープライズアプリケーションの構築過程では、さまざまなアーキテクチャ面の課題がある。本書は、決してそれらすべてを網羅しているわけではない。ソフトウェア構築においては、私はイテレーティブ開発論者である。イテレーティブ開発の根底には、顧客にとって有効と思われるものが得られたら、たとえ完全でなくてもソフトウェアを配信するという考え方がある。本を書くことと、ソフトウェアを作成することには多くの違いがあるが、この考え方に限っては両者は共通していると思っている。言い換えれば、つまり、本書は完全なものではないが、エンタープライズアプリケーションアーキテクチャに不可欠なアドバイ

スの数々が書き込まれている（と私は信じている）。本書で紹介している主な項目は次のとおりである。

- エンタープライズアプリケーションのレイヤ化
- ドメイン（ビジネス）ロジックの構築
- Web ユーザインタフェースの構築
- インメモリモジュール（特にオブジェクト）とリレーショナルデータベースのリンク
- ステートレス環境におけるセッションステートの扱い
- 配布の原則

本書では触れていない項目も多い。妥当性の整理、メッセージングと非同期通信の統合、セキュリティ、エラー処理、クラスタリング、アプリケーション統合、アーキテクチャのリファクタリング、リッチクライアントユーザインタフェースの構築など、書きたいことはいくらでもある。ただし、紙面と時間の制約、それに着想力不足から、これらの項目には本書では触れることができなかった。今はただ、いくつかのパターンが近い将来実現することを祈るばかりである。続編でこれらの項目を解説するという手もあるが、もしかしたら他の誰かがこれらの空白を埋めるような本を書いてくれるかもしれない。

この中でも、メッセージベースの通信は、特に重要な項目であると言えるだろう。複数のアプリケーションを統合して使う場合、非同期メッセージベース通信を使うケースが増えるからだ。1 つのアプリケーション内でのこれらの使用についても語ることはたくさんあるだろう。

本書は、特定のソフトウェアプラットフォームを前提にしたものではない。私が初めてこれらのパターンに出会ったのは、80 年代後半から 90 年代初頭のことで、Smalltalk、C++、CORBA などで作業をしていたときだ。90 年代後半になると、Java を使った作業の割合が格段と大きくなり、これらのパターンが初期の Java/CORBA システムや J2EE ベースのものにもスムーズに適用できることを発見する。最近になって、Microsoft 社の.NET プラットフォームでの作業もするようになったが、ここでもパターンは適用できることがわかった。ThoughtWorks の仲間からは、主に彼らの Forte での経験を教わった。エンタープライズアプリケーションで使われた、あるいは使われるであろう仕様の中で、すべてのプラットフォームで通用するという汎用性を保証することはできないが、少なくとも今までのところ、これらのパターンは汎用性に優れていることを実証している。

ほとんどのパターンで例としてコードを紹介している。使用する言語には、ほとんどの読者にとって読みやすく理解しやすいと思われるものを選択した。本書では Java を選択した。

C、C++ を読める人であればJava は読めるわけで、しかもC++ よりはるかに簡単だからだ。つまり、ほとんどのC++ プログラマはJava を読めるが、その逆は必ずしもそうではないわけだ。私はかなりのオブジェクト指向主義者なので、必然的にオブジェクト指向言語を使うことになる。その結果、ほとんどのコード例はJava で書かれている。本書の執筆中に、Microsoft は.NET 環境の安定化に着手し始め、そこで使われているC# 言語の特徴は、Java とほとんど同じである。そこで、コード例のいくつかをC# でも書いてみたが、これには多少のリスクがあった。なぜなら、開発者は.NET での経験が浅く、これを使うという発想そのものが時期尚早だからだ。いずれもC をベースにした言語なので、いずれか1つを読めれば、たとえその言語やプラットフォームに精通していなくても、両方を読むことができるはずだ。私の狙いは、最も多くのソフトウェア開発者が読めると思われる言語を使用することだった。たとえ、それが彼らのメインのまたは好みの言語ではないとしてもだ。(Smalltalk、Delphi、Visual Basic、Perl、Python、Ruby、COBOL などの言語を好んで使っている開発者には前もって謝罪しておく。Java やC# よりもっと優れた言語があるという意見も当然あるだろう。1つだけ言えるのは、私も同感だということだ!)

紹介している例は、あくまでもパターン内のアイデアを紹介・説明するためのものである。実践ですぐに利用できる解決策ではない。すべてのケースにおいて、実践で使うにはかなりの修正を必要とすることだろう。つまり、パターンは出発点としては有効だが、必ずしもそれがそのまま適用できるというものではないということだ。

本書の対象読者

本書は、エンタープライズアプリケーションの構築に携わるプログラマ、設計者、およびアーキテクトのなかでも、そのアーキテクチャに関する問題をより深く理解し、それらの問題についてより効果的にコミュニケーションしたいと思っている人を対象に書かれている。

本書の読者は、大きく2つのグループに分かれる。独自のソフトウェアを構築しようとしているニーズ的には比較的低い人達、そしてツールを使う、よりニーズの高い人達だ。比較的ニーズの低い人達にとって、本書で紹介するパターンは出発点として利用できることだろう。多くの分野で、パターンが提供する以上のものが必要となるだろうが、より優位に作業を始められるようになるはずだ。ツールユーザにとって、本書は水面下で何が起こっているかを把握し、ツールがサポートするパターンの選択などに役立つこととなるはずだ。ただし、たとえば、オブジェクトリレーショナルマッピングツールで、どのように対応付けるかは読者自身が決定を下さなくてはならない。パターンを読めば、何を選択するべきかの指針になるはずだ。

第3のグループとして、独自のソフトウェアを構築しようとしている人で高いニーズを

持っている人達もあるかもしれない。まず言えることは、使用するツールをよく知るということだ。過去に、フレームワークを構築するのに膨大な時間を要したプロジェクトをいくつか見てきたが、もちろんプロジェクトそのものの目的はそれではない。それでも納得できるという場合は、構わないが。ただし、覚えておいてほしいのは、本書で紹介しているコード例は、理解しやすいように簡略化されたものであり、実践で生じるさまざまなニーズに対応させるには、相当な作業を要するということだ。

パターンは、再現性のある問題に対する共通解決策なので、いくつかのパターンはすでに見たことがあるという人がいても不思議ではない。エンタープライズアプリケーションでの作業を続けていれば、いずれはほとんどのパターンに遭遇することになるだろう。本書で何か新しいことを書いたつもりはない。反対に、本書は（業界内における）古くからの構想をまとめたものであると言えるだろう。この分野は初めてという人には、本書は、これらのテクニックを学ぶのに役立つだろう。これらのテクニックに精通している人には、本書が他の人とのコミュニケーションやそれらの人達に教えるのに役立つものとなれば幸いだ。共通のボキャブラリを構築するのもパターンの重要な部分だ。たとえば、このクラスは**リモートファサード**だと言ったときに、設計者がその意味をわかるようにした方がいいからだ。

謝辞

本書に書かれている内容は、長年に渡って多くの人達といろいろな形で共同作業した中から学んだものである。多くの人がさまざまな形で貢献してくれている。中には、誰がいつ言ったかは覚えていないことも含まれているが、思い出せる限り、この場を借りてその貢献に対し謝意を述べたいと思う。

まずは、同僚の貢献者から始めよう。David Rice は、ThoughtWorks での同僚で、本書の 10 分の 1 は彼の貢献によるものと言える。締め切りに間に合わせるためによく二人で深夜にインスタントメッセージをやり取りしたが（彼は、その間、クライアントのサポートもしていた）、その中で彼は「本を書くことがいかに大変で面白いことかがわかったよ」と告白したぐらいだ。

Matt Foemmel は、ThoughtWorks の同僚で、エアコンがないところでは文章を書くのを嫌がったが、コード部分で大きく貢献してくれた（本書に対する簡潔明瞭な批評も）。**サービスレイヤ**では、この分野に強い Randy Stafford の貢献が大きかった。Edward Hieatt と Rob Mee の貢献にも謝意を表したい。これは、Rob がレビューしている段階であるギャップに気付いたところから始まる。彼は、私のお気に入りのレビューアになった。単に不足しているものに気付くだけでなく、どう修正するかでも助けてくれたからだ。

いつものことだが、本書のレビューをしてくれた次の優秀なレビューア達には、言葉では

言い尽くせない恩義を感じている。

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

ThoughtWorks の内線一覧表を載せてもいいぐらい、多くの同僚達が設計や経験などを語ってくれたおかげでこのプロジェクトは実現したようなものだ。多くのパターンを思いつくのに至ったのも、社内の優秀な設計者との会話があったからで、謝意は会社全体に対して表するのが適切だろう。

Kyle Brown、Rachel Reinitz、それに Bobby Woolf は、ノースカロライナにて惜しみなく時間を割いて詳細なレビューをしてくれた。彼らの徹底したレビューによって、さまざまな知恵が注入された。特に、Kyle との長電話は楽しかっただけでなく、ここではすべてを挙げられないぐらいの貢献度があった。

2000 年初頭、Alan Knight と Kai Yu を交えて Java One について話をしたのが本書の出発点となっている。彼らおよび後にその会話の内容やアイデアを整理するのを手伝ってくれた Josh Mackenzie、Rebecca Parsons、Dave Rice に感謝したい。.NET という新しい世界を知るのをサポートしてくれた Jim Newkirk にも謝意を表したい。

この分野の業務に携わる人達との会話と共同作業から、とても多くのことを学習させてもらった。特に、Gemstone での Foodsmart サンプルシステムの構築に協力してくれた Colleen Roe、David Muirhead、Randy Stafford に謝意を表したい。また、Bruce Eckel がホストした Crested Butte ワークショップでの会話も大いに役立っている。ここ数年間、あのイベントに参加していた人達にも、ここで謝意を表したい。Joshua Kerievsky は、レビューそのものには余り多くの時間は割けなかったにもかかわらず、パターンコンサルタントとして貢献してくれた。

いつものように、UIUC リーディンググループの徹底したオーディオレビューには、大いに助けてもらった。次の方々に謝意を表する。Ariel Gertzenstein、Bosko Zivaljevic、Brad Jones、Brian Foote、Brian Marick、Federico Balaguer、Joseph Yoder、John Brant、Mike Hewner、Ralph Johnson、Weerasak Witthawaskul。

Dragos Manolescu は、元 UIUC ヒットマンで現在は自身のグループを持っており、今回も色々なフィードバックをしてくれた。次の方々にも謝意を表する。Muhammad Anan、Brian Doyle、Emad Ghosheh、Glenn Graessle、Daniel Hein、Prabhaharan

Kumarakulasingam、Joe Quint、John Reinke、Kevin Reynolds、Sripriya Srinivasan、Tirumala Vaddiraju

Kent Beck は、覚えきれないほどの優れたアイデアを提供してくれた。思い出すのは、**スペシャルケース**の名前を思いついてくれたことだ。Jim Odell は、私をコンサルティングと教育の世界に引きずり込んだ張本人で、この謝辞から外すわけにはいかない。

本書は、執筆過程において、下書きを Web にアップしていた。この際、多くの人が電子メールで問題点を指摘してくれたり、質問したり、代替案などを提案してくれた。これには、次の方々が含まれる。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

ほかにも、名前も知らない、あるいは失念してしまった方々から多くの意見を頂いている。その方々にもこの場を借りて感謝したい。

最後に、最大の感謝を妻の Cindy に捧げる。彼女がいてくれたからこそ、本書を書き上げることができたのだ。

マーチン・ファウラー

2002 年 8 月

マサチューセッツ州メルローズにて

http://martinfowler.com

目　次

はじめに

コンピュータシステムを構築するには、十分な理解が必要である。システムが複雑になればなるほど、ソフトウェア構築の負荷は飛躍的に大きくなる。どんな職務にあっても、私たちは失敗と成功の両方から学ぶことでしか、向上することはできない。本書はそのような教訓のいくつかをまとめたものだ。本書の読者が、かつての私のような苦労をしなくても経験則が得られ、他の人に効果的な意思伝達ができるようになることを願っている。

ここでは、本書の目的を示し、本書の考え方の背後にあるものを説明したい。

アーキテクチャ

ソフトウェア業界では、言葉を拡大解釈して、微妙に矛盾した無数の意味を持たせることが好まれる。最も被害を受けている言葉の1つが「アーキテクチャ（architecture）」である。私はどうしてもこの言葉から重要そうな響きを感じてしまう。いかにも重要な話をしていると思わせるために使用されているような気がするのだ。だが、嫌味を言うのはこのくらいにしておこう。

アーキテクチャは、多くの人が定義しようとして、なかなか同意に至らない用語である。この用語には誰もが認める2つの要素がある。1つは、システムから個々のパーツへとどこまでもブレークダウンできるということ、もう1つは、簡単には変更できない決定事項だということである。また、次第に理解されてきたことだが、システムのアーキテクチャのあり方は1つだけでなく、1つのシステムには複数のアーキテクチャがあり、アーキテクチャにとって重要なことはシステムの存続期間の中で変わることがある。

Ralph Johnsonはメーリングリストに実に素晴らしい投稿を行うことがある。私が本書の草稿を書き終えたときにも、アーキテクチャに関して投稿していた。彼はその中で、アーキテクチャとは主観的要素であり、プロジェクト内の熟練開発者がシステム設計に関して共通に理解していることにすぎないと述べている。この共通理解はふつう、システムの主要なコンポー

ネントはどれで、それらのコンポーネントはどのように相互作用するかということに関するものである。また、開発者が早く確認したいと願っている決定事項に関しての共通理解でもある。この決定事項は後で変更するのは難しいと考えられているからだ。ここで主観性が問題になるのは、思っていたよりも変更が容易であるとわかったら、それはもはやアーキテクチャではないからである。つまりアーキテクチャとは、何であれ重要な要素なのである。

本書で私が示したいのは、エンタープライズアプリケーションの主要なパーツと、早めに確認したい決定事項に関する私なりの知見である。私が最も気に入っているアーキテクチャパターンは、第1章で説明するレイヤに関するパターンである。したがって本書は、エンタープライズアプリケーションをレイヤに分割する方法と、これらのレイヤを互いに連携させる方法についての書籍になる。重要なエンタープライズアプリケーションの多くは特定の形式のレイヤ化アーキテクチャを使用するが、場合によってはパイプやフィルタなど他の手法が有効なこともある。本書ではそのような場合についての説明は避け、最も広範囲に役立つレイヤ化アーキテクチャに焦点を絞ることにする。

本書に紹介するパターンのいくつかは、パーツについての重要な決定を表しているという点でアーキテクチャ的である。一方、どちらかというと設計に関連し、アーキテクチャを理解するために役立つパターンもある。何がアーキテクチャ的で何がアーキテクチャ的でないかはとても主観的なことので、これら2つを区別することはやめておこう。

エンタープライズアプリケーション

多くの人がコンピュータソフトウェアを作成し、私たちはそれらをすべてソフトウェア開発と呼んでいる。しかし、ソフトウェアにもいろいろなものがあり、それぞれ目的も違えば複雑さの度合いも違う。電気通信業界の友人と話をしたときに気づいたのは、エンタープライズアプリケーションは電気通信ソフトウェアよりもある意味で扱いやすいということだ。厄介なマルチスレッドの問題は存在せず、ハードウェアとソフトウェアの統合も不要である。しかしとても難しい面もある。エンタープライズアプリケーションでは、複雑なデータに対処しなければならない場合が多い。しかも論理的とはいいがたいビジネスルールに従いながらである。あらゆる種類のソフトウェアに適した技法やパターンもあるにはあるが、たいていは特定の種類のソフトウェアにだけ適している場合が多い。

私は主にエンタープライズアプリケーションに関わってきたので、本書で紹介するパターンはすべてエンタープライズアプリケーションに関するものである。「エンタープライズアプリケーション」という用語が何を意味するか、明確に定義することはできないにしても、私なりにその意味するところを示すことはできる（エンタープライズアプリケーションを指す他の用語には「情報システム」とか、長期間データを格納するアプリケーションであれば

「データ処理システム」がある)。

エンタープライズアプリケーションには、給与計算、診療記録、出荷管理、コスト分析、信用調査、保険、サプライチェーン、会計、顧客サービス、外国為替取引などが含まれる。一方、自動車用燃料噴射、ワープロ、エレベータ制御、化学プラント制御、電話交換、OS、コンパイラ、ゲームはエンタープライズアプリケーションには含まれない。

エンタープライズアプリケーションは、通常**永続データ**を伴う。「永続」というのは、プログラムが実行されていない間もそのデータが必要だからである。しかもたいていは何年も保存しなければならない。その間にこのデータを使用するプログラムには多くの変更が行われる。データを作成したハードウェアよりも長い期間、そのデータが存在することが多く、OS やコンパイラよりも長く存在することさえある。その間、古い情報を失うことなく新しい情報を格納するために、データ構造に多くの変更が行われる。抜本的な変更のためにまったく新しい業務アプリケーションをインストールするときでも、データを新しいアプリケーションに移行して使用できなければいけないのである。

世の中には通常**たくさんのデータ**があるので(平均的なシステムでも 1GB を超えるデータがあり、数千万のレコードに体系化されている)、これを管理することがシステムの重要な役割になる。古いシステムでは IBM の VSAM や ISAM などのインデックス付きのファイル構造が使われていたが、最新のシステムではデータベース(主にリレーショナルデータベース)を使用することが多い。これらのデータベースの設計と入力は、それ自体が 1 つの専門分野である。

通常、多くのユーザが**同時にデータにアクセスする**。同時にアクセスするユーザ数が 100 に満たないシステムも多いが、インターネットに公開される Web ベースのシステムではユーザ数は大幅に増加する。ユーザ数がどんなに多くても、全員がシステムに適切にアクセスできなければいけない。また、ユーザ数が少ない場合でも、2 人のユーザが同時に同じデータに、エラーを引き起こさないようにアクセスできなければならない。トランザクション管理ツールを使えばこの問題にはある程度対処できるとしても、アプリケーション開発者がこの問題を考慮せずにいることはほとんど不可能である。

通常、多くのデータがある場合は、データを処理するために**多くのユーザインタフェース画面**がある。何百もの異なる画面があることは珍しくない。エンタープライズアプリケーションのユーザには、定期的に利用しないユーザもいれば常時利用するユーザもいるが、技術的な知識を持たない人が多い。このため、ユーザの多様な目的に合わせてさまざまな方法でデータを表示できなければいけない。システムには多くのバッチ処理が存在することもあるが、ユーザとの相互作用に重きを置くユースケースのことばかり考えていると、バッチ処理を忘れがちになる。

エンタープライズアプリケーションが孤立して存在することはほとんどない。通常は企業の周辺に点在する**他のエンタープライズアプリケーションと統合する**必要がある。システム

が異なれば使用される技術も異なり、さらにCOBOLデータファイル、CORBA、メッセージングシステムなど協調メカニズムも異なるだろう。企業では、共通の通信技術を使用してさまざまなシステムを統合しようとすることが多い。そうするともちろん作業を完了できず、複数の異なる統合計画が並存することになる。ビジネスパートナーとビジネスを統合させようとすると、さらに事態は悪化する。

企業が統合のための技術を1つにしたとしても、ビジネスプロセスの相違と、データの**概念の不一致**に突き当たる。企業の部門によっては、現在契約中の人だけを顧客と考えるところもあれば、かつて契約を締結していたけれど今は契約していない人も顧客と考えるところもある。製品の売り上げを重視し、サービスの売り上げは重視しない部門もある。これらの違いは容易に区別できるように思うかもしれないが、何百ものレコードがあり、その各フィールドが微妙に異なる意味を持つ場合は、極めて大きな問題となる。たとえ、各フィールドが何を意味するかを熟知している社員がいたとしてもだ（もちろん、これらのデータはすべて予告なしに変更される）。結果として、構文的にも意味的にも異なるあらゆる種類のフォーマットで、データの読み込み、変更、書き込みが行われなければいけない。

「ビジネスロジック」という問題がある。奇妙な用語である。ビジネスロジックよりも論理的でないことはほとんどないからだ。たとえば、OSを構築するときは、すべてが論理的でなければいけない。しかし、ビジネスルールというものがあり、これを変更するには多大な政治的努力が必要だ。驚くべき方法で相互作用する計画性のない一連の奇妙な状況に対処しなければならない。もちろん、そのルールを採用するにはそれなりの理由があるだろう。たとえば、営業マンが顧客の会計期間に合わせて通常より2日遅い決済を行うように交渉し、その結果200～300万ドルの利益を得るといったことがある。このような1回限りの特殊事例が多々あり、これがビジネスソフトウェアの構築をとても困難にする複雑な**ビジネスロジック（非論理）**につながっている。このような状況では、ビジネスロジックをできる限り効率よく組織化しなければいけない。ロジックは時間とともに変化するということだけが、唯一確実なことだからである。

「エンタープライズアプリケーション」という用語は、大規模システムを意味すると考える人もいる。エンタープライズアプリケーションは企業に多くのメリットを提供するが、だからといってすべてのエンタープライズアプリケーションが大規模なわけではない。小規模システムは重要ではないと思い込んでいる人も多いが、小規模システムにもある程度の利点はある。たとえば、小規模システムがダウンしても、大規模システムほど深刻な損害にはならない。しかし、このような考え方のせいで、小規模プロジェクトの多くでは累積効果が正しく評価されていないように思える。小規模プロジェクトは見掛け以上の価値を持つことが多いので、小規模プロジェクトの改善が可能であれば、その累積効果は企業にとって極めて重要である。アーキテクチャとプロセスを単純化して大規模なプロジェクトを小規模に変えることが最善の策なのである。

エンタープライズアプリケーションの種類

エンタープライズアプリケーションの設計方法とそこで採用するパターンについて検討するときに重要なのは、エンタープライズアプリケーションはそれぞれ異なり、問題が異なれば実行方法も異なると認識することである。私は人が「いつもこうする」と言うのを聞くと警戒心が働く。私にとって設計に関する課題（と関心）の大半は、代替案をいくつか考えておくことであり、1つの代替案を採用することのトレードオフを判断することである。代替案の範囲は広いが、ここでは3つの要点を述べることにしよう。

B2C（business to customer）のオンライン小売業を考えてみよう。この小売業では、Webページで商品を閲覧し、気に入ったものが見つかればショッピングカートに入れ、購入する。このようなシステムでは多数のユーザを扱うことが必須であるため、リソースを効率的に使用するだけでなく、多くの負荷に耐えられるようにハードウェアを追加できるだけの拡張性を持たせておくことが必要となる。このようなアプリケーション用のドメインロジックは、受注、比較的単純な価格計算と出荷計算、出荷の通知など、とてもわかりやすい。簡単にアクセスできるようにしたいので、広く使用されているブラウザーで表示できるような一般的なWebプレゼンテーションにする。データソースには注文を記録するデータベースが含まれ、在庫情報や配送情報を入手するために在庫管理システムとやりとりすることもある。

このB2C小売業のシステムと、リース契約処理を自動化するシステムを対比してみよう。ある意味で、このシステムの方がB2C小売業のシステムよりもシンプルである。それは、ユーザの数が少ないからである（一度に使用するユーザは100人ほど）。しかし、ビジネスロジックはリース契約処理システムの方が複雑である。月々のリース料の計算、期限前の返品や支払遅延などのイベントの扱い、リース予約が入ったときのデータの妥当性確認などは、すべて複雑な処理である。なぜ処理が複雑になるかというと、リース業界の競争の多くが過去の取引の小さなバリエーションとして行われるからである。このような複雑なビジネスドメインは、規則があまりにも任意であるため、大変な難題となる。

また、このようなシステムではUI（ユーザインタフェース）も複雑になる。少なくとも複雑な画面が入り組んだHTMLインタフェースが必要になる。このようなシステムは、ユーザがHTMLのフロントエンドよりも洗練されたプレゼンテーションを望むというUI要求を持つことが多いので、リッチクライアントのインタフェースが必要になる。また、ユーザ同士の複雑な相互作用は、複雑なトランザクションの振る舞いにつながる。たとえば、リースの予約には1、2時間かかることがあり、その間ユーザの論理的なトランザクションは継続される。また、200ものテーブルと資産評価や価格決定のパッケージへの接続を持つ複雑なデータベーススキーマさえある。

3つめの例は、小規模な会社のシンプルな経費管理システムである。このシステムのユーザは少数で、ロジックもシンプルなので、HTMLプレゼンテーションを用いて社内の至る

場所から容易にシステムへアクセスすることができる。データソースはデータベースにある少数のテーブルだけである。一見シンプルに見えるが、このようなシステムにも難題がある。それは極めて短期間で構築しなければいけないことだ。また、ユーザの要求が増えるにつれてシステムが大規模になっていくことも考えておく必要がある。払い戻し小切手の計算をしたり、それを給与計算システムに提供したり、税額の計算をしたり、CFO（財務最高責任者）への報告書を作成したり、Webの航空券予約サービスへリンクしたり、などといったことを行う必要が生じるたびに、システムを拡張していかなければならないのである。他の2つの例で述べたシステムのいずれかのアーキテクチャを使用しようとすると、このシステムの開発が遅くなる。システムがビジネスにとってメリットになる場合は（すべてのエンタープライズアプリケーションがそうなるべきだが）、メリットの獲得が遅れるとコストが発生する。だからといって、将来の成長を妨げるような決定は誰も行いたくないだろう。しかし、ここでシステムに下手に柔軟性を加えても、柔軟性を持たせるために追加された複雑性によって、実際には将来的にシステムを発展させることが難しくなり、配置を遅らせることになるので、なかなかメリットが得られない。このようなシステムは、小規模ではあってもほとんどの企業が多用しているので、不適切なアーキテクチャの累積効果は深刻である。

これら3つのエンタープライズアプリケーションの例には、それぞれ異なる問題がある。このため3つの例すべてに適切な1つのアーキテクチャを見つけることはできない。アーキテクチャの選択とは、それぞれのシステム固有の問題を理解し、その理解に従って適切な設計を選択しなければならないことを意味する。これが、本書で企業のニーズに対して1つの解決策を提示しない理由である。その代わり、選択肢や代替案として役立つ多くのパターンを提示したい。特定のパターンを選択した場合でも、要求に合わせて修正しなければならないことがある。熟考なくしてエンタープライズソフトウェアを構築することはできない。ソフトウェア構築に対するあらゆる書籍の役割は、決定を下すのに役立つ情報を提供することである。

この考え方は、パターンとツールの両方に当てはまる。アプリケーションを開発するためにできるだけ小規模なツールを選ぶことは当然だが、目的ごとに最適なツールは何かを判断しなければいけない。アプリケーションの種類ごとに適切なツールを使用するように注意してほしい。ツールを適切に使用しないと、支援ではなく妨げになる場合があるからだ。

パフォーマンスに関する考察

アーキテクチャ上の決定の多くはパフォーマンスに関するものである。パフォーマンスの問題の大半に対しては、システムを起動させ、実行して測定し、その結果を基にして規則正しい最適化処理を行うことが好ましい。しかし、最適化では解決できない形でパフォーマン

スに影響を及ぼすアーキテクチャ上の決定もある。容易に解決できる場合でも、プロジェクトに関わる人は早い段階でこの決定について悩む。

本書のような書籍でパフォーマンスについて説明することは難しい。その理由は、パフォーマンスに関するアドバイスは、現実のシステム構成でパフォーマンスの測定を行ってみるまでは、当てにならないからである。パフォーマンスを考慮したために、設計の採用／不採用が行われるのを目にすることがとても多い。しかも、実際のアプリケーション設定でパフォーマンスを測定してみると、そうした考慮が無意味だったことが判明したりするのである。

本書では、リモートコールを最小限にすることなどの指針をいくつか紹介する。このリモートコールについての指針は、パフォーマンスについての優れたアドバイスとされてきたものである。それでも、実際にアプリケーションで測定して、すべてのアドバイスが適切なものかどうかを確認したほうがよい。同じく、本書で提示するコードの例は、わかりやすくするためにパフォーマンスを犠牲にしている場合がある。もう一度言うが、最適化を行うべきかどうかは、個々の環境によって異なる。パフォーマンスの最適化を行う場合は常に、最適化の前後で測定を行ってみるべきである。この測定を行わなければコードを読むことは難しい。

測定に関して重要なことがある。システム構成に重要な変更を行うと、パフォーマンスは白紙の状態に戻ってしまう。このため、仮想マシン、ハードウェア、データベースまたはその他のものをバージョンアップする場合は、再度パフォーマンスの最適化を行い、効果があるかどうかを確認しなければいけない。多くの場合、システム構成を新しくするとさまざまな変化が発生する。パフォーマンスを改善するために過去に行った最適化が、新しい環境のパフォーマンスを損なう場合もありうるのだ。

パフォーマンスに関するもう１つの問題は、多くの用語が一貫性なく使用されていることである。最も被害を受けているのは「拡張性（scalability）」という言葉であり、さまざまなことを表すために定期的に使用されている。ここで、私が使う用語の定義を示そう。

レスポンス時間は、システムが外部からのリクエストの処理に要する時間である。ボタンを押すなどのUIアクションやサーバAPI呼び出しがそのようなリクエストの例だ。

レスポンス性は、システムがリクエスト処理に対し、どれだけ早くレスポンスを返すかを表す。システムのレスポンス性が低いとユーザは欲求不満になるので、レスポンス性は多くのシステムにとって重要である。全体のリクエストが完了するまでシステムが待っている場合は、レスポンス時間とレスポンス性は同じである。しかし、完了する前にリクエストを受け取ったという反応を返すのであれば、レスポンス性は優れている。たとえば、ファイルをコピーしている間に進行状況を表すバーを表示させると、レスポンス時間は改善されないが、ユーザインタフェースのレスポンス性は改善される。

待ち時間は、実行すべき作業が存在しなくても、いずれかのレスポンスを得るために必要な最小時間である。一般に、この待ち時間はリモートシステムでは重要な問題である。プロ

グラムに対し何も実行しないよう要求し、その要求が完了したら知らせるようにした場合、プログラムをノートパソコン上で実行しているのであれば即座にレスポンスが返ってくるべきである。しかし、プログラムがリモートコンピュータ上で実行されている場合は、リクエストとレスポンスの間に、ネットワークを通過する時間が数秒間必要だろう。アプリケーションの開発者が待ち時間を改善するためにできることはない。また、待ち時間はリモートコールを最小化すべき理由でもある。

スループットは、所定の時間内にどのくらいのことを行えるかを表した数字である。ファイルのコピー時間を決める場合は、スループットは1秒当たりのバイト数で測定される。エンタープライズアプリケーションの標準的な単位はtps（1秒当たりのトランザクション処理件数）だが、問題は、これが個々のトランザクションの複雑性によって決まってくることである。特定のシステムに対しては、共通のトランザクションを選ぶ必要がある。

私の用語一覧では、**パフォーマンス**は、スループットかレスポンス時間のどちらかを意味する（状況に応じて読み分けてほしい）。ある技法によってスループットが改善されてもレスポンス時間が低下する場合、パフォーマンスについて説明するのは難しい。このような場合は、より正確な用語を使うべきである。ユーザから見ればレスポンス時間よりもレスポンス性の方が重要なので、レスポンス時間やスループットの改善に手間を掛けるより、レスポンス性を改善したほうがパフォーマンスを向上できる。

負荷は、システムがどのくらいストレスを受けているかを表す。負荷はたとえば現在接続しているユーザ数で測定される。この概念はふつうレスポンス時間などの測定に関して使われる。あるリクエストのレスポンス時間は、10人のユーザがいた場合は0.5秒で、20人のユーザでは2秒になる、といった言い方をする。

負荷感度は、レスポンス時間が負荷によってどのように異なるかを表す。たとえば、システムAのレスポンス時間は10～20人のユーザでは0.5秒になり、システムBのレスポンス時間は10人のユーザでは0.2秒、20人のユーザでは2秒になるとする。この場合、システムAは、システムBよりも負荷感度は低い。また、**劣化**という用語を使用して、システムAよりもシステムBの方が劣化すると言う。

効率性は、リソースごとに分割されたパフォーマンスである。2つのCPUで30tpsになるシステムは、同じ4つのCPUで40tpsになるシステムより効率がよい。

システムの**容量**は、実効性のある最大限のスループットまたは負荷を表す。パフォーマンスが許容範囲のしきい値を割り込む地点、または最大絶対値を意味する。

拡張性は、リソース（通常、ハードウェア）の追加がパフォーマンスにどう影響するかを示す基準である。システムに拡張性があれば、ハードウェアを追加すると、それに比例してパフォーマンスが改善される。たとえば、サーバの台数を2倍にすると、スループットが2倍になる。**垂直方向の拡張性**すなわち**スケールアップ**は、メモリの増設など1台のサーバにパワーを追加することを指し、**水平方向の拡張性**すなわち**スケールアウト**は、サーバを追加

することを指す。

ここで問題なのは、設計に関する決定がこれらすべてのパフォーマンス要因に均しく影響するのではないということである。たとえば、1台のサーバで実行している2つのソフトウェアシステムがあるとする。Camelというシステムの容量が40tpsであるのに対し、Swordfishというシステムの容量は20tpsである。どちらのパフォーマンスが優れているか。どちらの方が拡張性があるか。このデータだけでは、拡張性についての質問に答えることはできない。唯一言えるのは、Camelの方が1台のサーバでは効率的だということである。もう1台のサーバを追加すると、Swordfishは35tpsを処理し、Camelは50tpsを処理することになったとしよう。Camelの方は容量に優れ、Swordfishの方はスケールアウトに優れていることになる。サーバを追加し続けると、Swordfishの容量は追加のサーバ当たり15tps増加し、Camelの容量は10tps増加する。このデータを考慮すると、サーバが5台未満であればCamelの方が効率的だが、Swordfishの方が水平方向の拡張性に優れていることがわかる。

エンタープライズアプリケーションを構築するときは、ハードウェアの容量や効率性ではなく、拡張性を構築することに意味があると考える場合が多い。拡張性があれば、必要に応じてパフォーマンスを向上させることができる。また、拡張性を持たせることは容易である。設計者は、ハードウェアを購入する方が実際には安い場合でも、現在のハードウェアプラットフォームの容量を改善するために複雑な作業を行うことが多い。Camelのコストが Swordfishよりも高く、そのコストがサーバ2台を追加するコストと同じである場合、40tpsだけが必要なら、Swordfishの方が安くなる。ソフトウェアを適切に実行させるためには優れたハードウェアに依存するしかないことに対しての不満が広まっている。私は、Wordの最新版を実行するためだけにノートパソコンをアップグレードしなければならないという不満なら大いに同感である。しかし、新しいハードウェアの方が、非力なシステムでソフトウェアを実行するよりも安く済むことが多い。同様に、システムに拡張性があるのなら、サーバを追加する方がプログラマを増やすよりも安く済むことが多いのである。

パターン

パターンには長い歴史があり、ここでその歴史を繰り返し語りたくはない。しかし本書は、パターンに対する私なりの知見と、パターンを設計記述手法として価値あるものにする方法を提供するための本である。

広く認められているパターンの定義はないが、出発点として、多くの熱狂的パターン主義者を鼓舞してきたChristopher Alexanderの定義を採り上げよう。「各パターンは、環境の中で繰り返し起こる問題であり、その問題に対する解決策の核心となるものである。この解

決策は、毎回違った形で何百万回でも使用できる」[Alexander et al.]。Alexanderは建築士なのでここでは建築物について述べているが、ソフトウェアに関してもその定義はとてもよく当てはまる。パターンが焦点を当てるのは特定の解決策であり、繰り返し発生する1つ以上の問題に対して共通かつ効果的に対処するためのものである。別の見方では、パターンはアドバイスの塊であり、パターンを作成するにはいくつものアドバイスを比較的独立した塊に分割する。こうすることで、パターンを参照し、個別に検討することができる。

パターンの重要な点は、それが実践に根差していることである。人の行動を考察し、物事の動き方を観察し、「解決策の核心」を探すことでパターンを発見する。容易なプロセスではないが、優れたパターンを発見すればとても役立つ。私にとってはリファレンス書籍を作ることができるという点で有用である。本書やパターンに関する他の書籍を全部読み、それが役立つかを判断する必要はない。必要なのは、書籍を読んで、何がパターンで、パターンが解決するのはどんな問題か、パターンは問題をどのように解決するか、などについての感覚を十分に持つことである。すべての詳細を知る必要はないが、問題にぶつかった場合に、本書からパターンを探せるようになる必要がある。そのときにそのパターンを徹底的に理解すればよいのである。

パターンが必要になったときには、その状況にパターンをどう適用するかを考えなければならない。パターンに関して重要な点は、やみくもに解決策を適用するだけではいけないということである。これはパターンツールを使用しても悲惨な失敗を招く理由である。パターンは常に「生焼け」であり、常に実際のプロジェクトというオーブンで完成させなければならない。私がパターンを使用するときは、常にところどころ微調整する。何度も同じ解決策を目にしているようでも、厳密には同じではないのである。

各パターンは比較的独立しているが、分離してはいない。あるパターンが他のパターンに関係したり、別のパターンがあってはじめて存在するパターンがあったりする場合がある。たとえば**クラステーブル継承**が存在するのは、通常、**ドメインモデル**が設計に使われている場合だけである。パターンの間の境界は曖昧だが、本書では、各パターンをできる限り独立した状態にしようとした。**ユニットオブワーク**の使用と誰かが言ったら、その部分だけを調べれば本書をすべて読まなくても適用方法を理解してもらえると思う。

エンタープライズアプリケーションに熟練した設計者であれば、本書に載っているパターンの多くに見覚えがあるだろう。「まえがき」にも書いたが、あまり失望しないでほしい。パターンはオリジナルな考え方ではなく、各分野で起こっていることについて観察した結果に過ぎない。そのため、パターンについて書く私たちのような物書きは、パターンを「発明する」とは言わず、「発見する」と言う。私たちの役割は、共通の解決策の指摘、解決策の核心の考察、その結果として生じるパターンの記述を行うことである。熟練の設計者にとってパターンの価値は、新しい考え方を示すのではなく、自らの考え方を伝達できるようになることである。あなたやあなたの同僚が**リモートファサード**とは何かを知っていれば、「こ

のクラスはリモートファサードだよ」と言うだけで多くのことを伝えられる。また、新人に「データ変換オブジェクトを使ってこれを行ってくれないか」と言えば、彼は本書を読んで調べることができるだろう。パターンは設計についての語彙を創造することになる。このため、名前の付け方は大事な問題である。

これら多くのパターンがエンタープライズアプリケーション用であるのに対し、基礎パターンの章（第18章）にあるパターンは、より一般的かつ局所的である。このパターンを含めた理由は、エンタープライズアプリケーションのパターンを論じるときに参照するためである。

パターンの構造

パターンを書く者は、パターンの形式を選択しなければならない。[Alexander et al.]、[Gang of Four]、[POSA]などの古典的なパターン書籍に基づく場合もあれば、独自の形式を作り上げる場合もある。私はGOFのような細かい形式は使いたくないが、リファレンスを充実させる項目が必要である。そこで、本書で使う項目を以下に挙げることにしよう。

1つ目の項目は、パターン名である。パターン名はとても重要だが、それは、設計者が効率的にやり取りするための語彙を作成することがパターンの目的の一部だからである。たとえば、私のWebサーバは**フロントコントローラ**と**トランスフォームビュー**で構築されていると言えば、これらのパターンについて知っている人なら、このWebサーバのアーキテクチャについてとても正確な考え方を持つことになる。

次は目的とスケッチという、互いに関連する項目である。目的とは2、3行からなるパターンの要約である。スケッチとはパターンのビジュアル表現で、UMLダイアグラムであることが多いが、常にそうだとは限らない。これらの項目はパターンがどのようなものであるかを示し、これらの項目によって速やかにパターンを思い出すことができる。すでに「パターンがある」場合、パターン名を知らなくても解決策がわかっているので、そのパターンがどのようなものかを知るためには目的とスケッチを見るだけでよい。

さらに次の項目では、そのパターンの使用を動機付ける問題を記述する。パターンが解決できるのはその問題だけではないが、この問題はそのパターンを使用する動機となる最も大きいものである。

動作方法では、解決策を説明する。ここでは、実装の問題と、私が目にしたさまざまなバリエーションについて説明する。特定のプラットフォームにはなるべく依存しないようにしている。プラットフォームに固有の部分は字下げしてあるので見つけやすく、読み飛ばすこともできるだろう。必要と思われるところにはUMLダイアグラムを挿入してある。

使用するタイミングでは、パターンをいつ使用すべきかを説明する。別の解決策と比較してこの解決策を選択することのトレードオフについて説明する。本書の多くのパターンは、

ページコントローラやフロントコントローラなどのように、選択肢の1つにすぎない。常に適切なパターンが選択されるとは限らないので、私はパターンを発見すると、常に「これを使用しないのはどのようなときか」と自問することにしている。この問いかけにより、他の選択肢となるパターンを見つけることができるのである。

参考文献では、そのパターンについて解説している他の書籍を紹介する。完全な文献リストではもちろんない。私はパターンを理解するのに必要な箇所だけを参照しているので、私が記述した内容の理解に直接役立たないものは取り上げず、また私が読んだことのないものも除外してある。さらに、見つけにくい文献や、いずれなくなる恐れのあるWebリンクも記載していない。

各パターンには1つ以上の**例**を挙げてある。いずれも、使用するパターンの**シンプルな例**であり、JavaやC#のコードを使用している。これらの言語を選択した理由は、専門のプログラマの間でおそらくもっとも使用人口の多い言語だからである。ここで重要なのは、例とパターンは同一ではないということである。実際にパターンを使用するときは、この例とまったく同じようになることはないので、マクロか何かのようにとらえないでいただきたい。例はできるだけシンプルに示しているので、最も明快な形のパターンを見ることができる。実際にパターンを使用するときに重要視しなければならないさまざまな問題は、ここでは扱っていない。しかし、それらの問題は環境に固有の問題であり、これが常にパターンを微調整しなければいけない理由である。

そのため、核となるメッセージはできるだけ明確にしながら、各例をシンプルにしたのである。私が本書の例として選択したのは、本番システムで必要となる各種のアイデアとパターンとがどう関連するかを示す例ではなく、シンプルで明確な例である。単純化すればいいというものではないが、現実的な周辺問題がパターンの要点を理解しにくくすることも確かである。

また、各例をシンプルにしたのは、関連した連続的な例ではなく独立した例を目指したためでもある。独立した例はそれだけで理解しやすいが、どのように複数の例を1つにまとめるかについての指針には欠ける。関連した例は物事がどうまとめられているかを示すことはできるが、その例に関わるパターンすべてを理解せずにどれか1つのパターンを理解することは難しい。互いに関連していながら個別に理解可能な例を作れなくはないが、少なくとも私にとっては難題である。だから独立した例を選択することにした。

例で使用するコードは、考え方をわかりやすくすることに焦点を当てて記述する。その結果、特にエラー処理（この領域ではパターンを開発したことがないので私はあまり注意していない）などのいくつかの処理を犠牲にすることになる。純粋にパターンを説明するためにコードを記述しているのであって、特定のビジネス問題をモデル化する方法を示しているのではない。

こうした理由で、私のWebサイトからコードをダウンロードしてもらうわけにはいかな

かった。本書に記載するコードの例は、基本的な考えを単純化して見せるためのもので、そのまま本番の設定に役立つというようなものではないからだ。

すべてのパターンに上記の項目すべてを記述しているわけではない。良い例や動機付けが思い浮かばない場合は省略した。

本書のパターンの限界

「まえがき」に書いたように、パターンを集めただけでは、エンタープライズアプリケーションの包括的なガイドとはならない。本書に関する私の判断基準は、パターンの収集が完全かどうかではなく、それらが役立つかどうかである。パターンという分野は、1冊の書籍はもちろん1人の仕事にも大きすぎる。

本書のパターンはすべて現場で目にしたパターンだが、私は派生的に起こる問題や相互関係のすべてを完全に理解しているわけではない。本書は、私の最新の理解を反映し、その理解は本書を書き進むにつれ発展してきた。本書が印刷された後も継続して発展させたいと願っている。ソフトウェア開発においては、何事も常に変化するからだ。

パターンの使用を検討するときは、そこが終点ではなく出発点だということを忘れてはいけない。いかなる著者も、ソフトウェアプロジェクトのすべてのバリエーションに精通することはできない。私がこれらのパターンを紹介するのは、読者に出発点を提供するためであり、私や私の周囲の人が悪戦苦闘したことから得た教訓を読者に読みとっていただくためである。どうかこれらの教訓を踏まえた上で、読者の方々それぞれの課題に挑んでいただきたい。どのパターンも完全ではなく、それぞれのシステムの環境に合わせて完成させることに、責任と喜びを持って取り組んでいただきたい。

第1部

概論

第1章

レイヤ化

レイヤ化は、ソフトウェア設計者が複雑なソフトウェアのシステムを分割するために使用する一般的な技法である。コンピュータの構造で言えば、OSのシステムコールのあるプログラミング言語のレイヤの下にデバイスドライバとCPUの命令群のレイヤがあり、さらにその下にチップ内のロジックゲートのレイヤがある。ネットワークでは、FTPのレイヤの下にTCPがあり、その下にIPがあり、さらにその下にはイーサネットがある。

レイヤの観点からシステムを考えた場合、ソフトウェアのサブシステムは、レイヤの積み重なったものとして思い描くことができる。そこでは、各レイヤはその下のレイヤに依存する。上位のレイヤは下位のレイヤで定義されたさまざまなサービスを使用するが、下位のレイヤは上位のレイヤを意識しない。さらに、各レイヤは上位のレイヤから下位のレイヤを隠ぺいすることが多いので、レイヤ4はレイヤ3を利用し、レイヤ3はレイヤ2を利用するが、レイヤ4はレイヤ2を意識することはない（レイヤ化アーキテクチャのレイヤすべてがこのように互いに不透明なわけではないが、多くは不透明である）。

システムをレイヤに分割することには次のように重要なメリットがいくつかある。

- 他のレイヤをよく知らなくても、1つのレイヤを全体として考えることができる。たとえば、イーサネットの動作方法を知らなくても、TCPの上にあるFTPサービスを構築する方法を理解できる。
- 同じ基本サービスの代替実装でレイヤを置き換えることができる。たとえば、FTPサービスは、イーサネット、PPP、またはケーブル会社が変わっても、その実行に支障はない。
- レイヤ間の依存を最小限にできる。たとえば、ケーブル会社がIPを動作させる物理的な伝送システムを変更しても、FTPサービスを修正する必要はない。
- レイヤは標準化に適している。TCPとIPは、レイヤの動作方法を定義しているので標準である。

- レイヤを構築すれば、多くの高水準のサービスがそのレイヤを使用できる。このため、TCP/IP は FTP、telnet、SSH、HTTP によって使用される。TCP/IP がなければ、高水準のプロトコルは独自の低水準のプロトコルを作成しなければいけない。

レイヤ化は重要な技法だが、以下のような弱点もある。

- レイヤは一部の物事をカプセル化してしまう。その結果、連鎖的な変更が起こる場合がある。レイヤ化されたエンタープライズアプリケーションにおける典型的な例は、UI（ユーザインタフェース）上で表示すべきフィールドの追加である。このフィールドはデータベースに存在する必要があり、したがって UI からデータベースまでのすべてのレイヤにそのフィールドを追加しなければいけない。
- レイヤを追加するとパフォーマンスを損ねる場合がある。一般的には、レイヤごとに表示を変換する必要がある。しかし、基盤となる機能をカプセル化すると、そのようなパフォーマンスの損失を補う以上の効率性をもたらすことが多い。トランザクションを制御するレイヤは最適化することができ、その結果すべての機能を高速にすることができる。

しかし、レイヤ化アーキテクチャにおいて最も難しいのは、必要なレイヤと、各レイヤが行うべき内容を決定することである。

1.1 エンタープライズアプリケーションのレイヤの発展

私は初期のバッチシステム開発には乗り遅れた世代だが、その頃はレイヤについて誰もあまり考えていなかったと思う。何らかのファイル形式（ISAM、VSAM など）を操作するプログラムを作成すれば、それがアプリケーションになり、レイヤを適用する必要はなかったのだ。

レイヤの概念は 90 年代の**クライアント／サーバシステム**の発展とともに明確になった。クライアント／サーバシステムは 2 レイヤのシステムであった。クライアントは UI とその他のアプリケーションコードを保持し、通常サーバはリレーショナルデータベースであった。一般的なクライアントツールは VB、Powerbuilder、Delphi だった。これらのツールは SQL を認識する UI ウィジェットを持っていたので、データ量の多いアプリケーションを容易に構築することができた。デザイン領域にコントロールをドラッグすることで画面を構築

し、プロパティシートを使ってそのコントロールをデータベースに接続することができたのである。

アプリケーションが表示とリレーショナルデータのシンプルな更新だけを行う場合、クライアント／サーバシステムは適切に動作した。問題は、ビジネスルール、妥当性、計算などのドメインロジックを伴ったときである。通常はクライアント側でこれらを作成するが、その際ロジックをUI画面に直接組み込むというやっかいな方法が採用される。ドメインロジックが複雑になればなるほど、コードの扱いが難しくなる。さらに、画面にロジックを組み込むことでコードが重複しやすくなる。そのため、シンプルな変更を行う場合でも、多くの画面に散らばる同様のコードを見つけて変更しなければいけないことになる。

この問題の解決案として、ドメインロジックをストアドプロシージャとしてデータベースに格納することが挙げられる。しかし、ストアドプロシージャには弱点があり、その構造化メカニズムには限度があるため、再びやっかいなコードの作成につながってしまう。さらに、リレーショナルデータベースを好む人が多いのはSQLが標準であるためデータベースベンダーを変更しやすいからであるが、ストアドプロシージャだけは各データベースベンダー固有のものなので、ベンダー変更に際しての選択肢がなくなってしまうという問題もある（実際にデータベースベンダーを変更する人はほとんどいないのに、なぜか多くの人は移植に高いコストをかけずにベンダーを変更できる選択肢を持ちたがるのだ）。

クライアント／サーバが人気を博したのと同じ頃、オブジェクト指向も注目を集めていた。オブジェクトコミュニティは、「3レイヤのシステムに移行する」ことでドメインロジックに関する問題を解決した。3レイヤシステムには、UI用のプレゼンテーションレイヤ、ドメインロジック用のドメインレイヤ、データソースのレイヤがある。これによって、すべての複雑なドメインロジックをUIから、オブジェクトによって適切に構造化できるレイヤに移すことができたのである。

だが、オブジェクト指向は主流にならなかった。多くのシステムはまだあまりにシンプルだった——少なくとも開発時は。3レイヤの手法には多くのメリットがあったが、問題がシンプルなものであれば、クライアント／サーバシステム用のツールを使用することで解決できた。クライアント／サーバシステム用のツールは、3レイヤの構成で使用するには扱いにくく、まったく使えないこともあった。

その後、Webの登場によって激震が走った。急に誰も彼もがWebブラウザーでクライアント／サーバアプリケーションを展開することを望みだした。しかし、ビジネスロジックがすべてリッチクライアントに埋め込まれている場合は、すべてのビジネスロジックを作り直してWebインタフェースを持たせなければならない。適切に設計された3レイヤシステムなら、新しいプレゼンテーションレイヤを加えるだけでこれを行うことができる。そして、Javaの登場によってオブジェクト指向言語が堂々と主流を占めるようになった。Webページ構築用に登場したツールは、SQLに拘束されることが少ないので、3つ目のレイヤで使い

やすいのである。

レイヤ化について語るとき、**レイヤ**（layer）と**ティア**（tier）がよく混同される。この2つの用語は同義語として使用されることが多いが、多くの人は**ティア**を物理的に区別されるものととらえている。クライアント／サーバシステムは、2ティアシステムとして説明されることが多い。クライアントはデスクトップであり、サーバはサーバであり、両者の区別は物理的なものだからだ。本書では、異なるマシンでレイヤを実行する必要がないことを強調するために**レイヤ**という用語を使用している。ドメインロジックのレイヤはデスクトップかデータベースサーバのどちらかで実行されることが多い。この状況ではノードは2つだが、レイヤは3つである。ローカルのデータベースであれば、1つのノートパソコンで3つのレイヤすべてを実行できるが、それらはあくまで3つの異なるレイヤである。

1.2 | 3つの主なレイヤ

本書では、プレゼンテーション、ドメイン、データソースという3つの主要なレイヤからなるアーキテクチャに焦点を絞る（これらの呼び名は[Brown et al.]に従っている）。表1.1にこれらのレイヤを要約する。

表1.1 ── 3つのレイヤ

レイヤ	役割
プレゼンテーションレイヤ	サービスの提供、情報の表示（たとえばWindowsやHTMLの場合なら、ユーザのリクエスト（マウスのクリック、キーボードの押下）、HTTPリクエスト、コマンドライン呼び出し、バッチAPIを処理する）
ドメインレイヤ	システムそのものであるロジック
データソースレイヤ	データベース、メッセージングシステム、トランザクションモニタ、その他のパッケージとの通信

プレゼンテーションロジックは、ユーザとソフトウェアとの相互作用を扱う。このロジックは、コマンドラインやテキストベースのメニューシステムと同じくらいシンプルな場合もあるが、現在はリッチクライアントのグラフィックUIやHTMLベースのブラウザーUIになる傾向が強い（本書では、HTMLブラウザーではなく、WindowsやSwingなどのファットクライアントのUIを意味する用語として**リッチクライアント**を使う）。プレゼンテーションレイヤは、主にユーザに情報を表示したり、ユーザからのコマンドをドメインやデータソースでの動作に変換したりする。

データソースロジックは、他のシステムとの通信を行い、アプリケーションのためにタスクを実行する。トランザクションモニタ、他のアプリケーション、メッセージングシステム

などと通信する。エンタープライズアプリケーションの多くでは、データソースロジックは永続的なデータを格納するデータベースであることが多い。

残るドメインロジックは、ビジネスロジックとも呼ばれる。現在作業中のドメインに対してアプリケーションが行うべき作業である。入力データと格納データに基づく計算、プレゼンテーションから送信されたデータの妥当性確認、プレゼンテーションレイヤから受信したコマンドに応じてどのデータソースロジックを呼び出すかの決定などが、この作業に含まれる。

レイヤの配置において、ドメインレイヤがプレゼンテーションレイヤからデータソースレイヤを完全に隠ぺいするように配置されている場合がある。しかし、もっと多いのは、プレゼンテーションからデータストアに直接アクセスする場合である。これはあまり美しい方法ではないが、実際には適切に動作することが多い。プレゼンテーションレイヤはユーザからのコマンドを解読し、データソースを使ってデータベースレイヤから関連データを抽出する。そして、ドメインロジックにそのデータを操作させ、画面に表示させる。

1つのアプリケーションには、この3つの分野ごとに複数のパッケージが存在することが多い。リッチクライアントのインタフェースを利用するエンドユーザだけでなく、コマンドラインを利用するユーザも操作できるように設計されたアプリケーションには、2つのプレゼンテーションがある。1つはリッチクライアントのインタフェース用、もう1つはコマンドライン用のプレゼンテーションである。さまざまなデータベースに対して複数のデータソースコンポーネントが存在するが、これは特に、既存のパッケージと通信するためである。ドメインは互いに関連のない領域に分割される可能性がある。データソースパッケージは特定のドメインパッケージにしか使用されないということもある。

ここまではユーザを中心に説明してきた。そこで、ソフトウェアを動かす人間がいないときに何が起こるかという疑問が湧いてくる。おそらく、それはWebサービスのような新しい流行のものか、またはバッチ処理のような日常的で実用的なものだろう。後者の場合、ユーザはクライアントプログラムである。この点で、プレゼンテーションレイヤとデータソースレイヤには多くの類似点があるように見える。どちらも外部との接続に関するレイヤだからだ。これは、Alistair Cockburnの**ヘキサゴナルアーキテクチャ**（Hexagonal Architecture、六角形のアーキテクチャ）パターン[wiki]の背後にあるロジックである。ヘキサゴナルアーキテクチャは、外部システムへのインタフェースで囲まれた核としてシステムを視覚化したものである。**ヘキサゴナルアーキテクチャ**では、外部にあるものは基本的にすべて外側のインタフェースであり、したがってこれは対称的なビューであって、私の言う非対称のレイヤ化スキーマではない。

しかし、私はこの非対称性こそ有用だと思っている。他へのサービスとして提供するインタフェースと、他のサービスを使用することとの間には、歴然とした違いがあるからだ。核心に触れていえば、これこそが私の言うプレゼンテーションとデータソースとの違いなのである。プレゼンテーションは、他者に対してシステムが提供するサービスの外部インタ

フェースである。この場合、提供する相手が複雑な人間であるか、または単純なリモートプログラムであるかは問わない。データソースは、サービスを提供するものへのインタフェースである。クライアントが異なるとサービスの考え方が変わるので、これらを分けて考えることは有用であると私は考える。

どのエンタープライズアプリケーションにもプレゼンテーション、ドメイン、データソースという3つのレイヤとそれぞれの役割があることを見てきたが、これらのレイヤをどう分けるかは、アプリケーションがどれだけ複雑であるかによって違ってくる。データベースからデータを抽出し、Webページに表示するというシンプルなスクリプトなら、すべて1つの手続きに収まるだろう。3つのレイヤに分けるとしても、各レイヤの振る舞いをそれぞれサブルーチンとするだけで十分だろう。システムがもっと複雑になれば、3つのレイヤを個々のクラスに分割する。さらに複雑になった場合は、クラスをいくつかのパッケージに分割する。私の一般的なアドバイスは、「状況に応じて最も適切な分割の仕方を選択すること、ただし最低限サブルーチンレベルの分割は行ったほうがよい」というものである。

分割だけでなく、依存性についても確かなルールがある。ドメインとデータソースはプレゼンテーションに依存してはいけない、言い換えると、ドメインコードやデータソースコードからプレゼンテーションコード内のサブルーチンを呼び出してはならない、というものだ。このルールによって、同じ土台の上で別のプレゼンテーションに置き換えやすくなり、深刻な問題を派生させずにプレゼンテーションを修正できるようになる。ドメインとデータソースとの関係はより複雑で、データソースのアーキテクチャパターンによって異なる。

ドメインロジックの扱いで最も難しいのは、人々が思っているように、何がドメインロジックで何が他のロジックかを見極めることだろう。私が好きな非公式のテストは、Webアプリケーションにコマンドラインインタフェースを追加するときのように、まったく異なるレイヤをアプリケーションに追加することを想像するというものである。この追加を行うときに機能を複製する必要があれば、それはドメインロジックがプレゼンテーションの中にはみ出していることを示している。同様に、リレーショナルデータベースをXMLファイルで置き換えるために、ロジックを複製する必要があるかどうか考えてみるとよい。

その良い例として、製品一覧を表示するシステムを考えてみよう。この一覧では、前月よりも販売が10パーセント増となったすべての製品が赤い字で表示される。これを実現するため、開発者は当月の売り上げと前月の売り上げを比較するロジックをプレゼンテーションレイヤに配置し、前月と当月の差異が10パーセント以上ある場合は赤で強調する。

しかし、ここで問題なのは、ドメインロジックがプレゼンテーションに入り込んでいることである。レイヤを適切に分離するためには、売り上げが増加しているかどうかを示すメソッドがドメインレイヤ上に必要である。このメソッドが2ヶ月の売り上げを比較し、論理値を返す。プレゼンテーションレイヤは、この論理メソッドを呼び出して、真であれば製品名を赤で強調する。こうすれば、強調表示すべきものがあるかどうかを判断する部分と、ど

のように強調表示するかを選択する部分とにプロセスを分割できる。

分割の仕方に関して、私は教条的すぎないだろうかという不安もある。本書の原稿を読んでもらったとき、Alan Knight はこう言った。「これを UI に押しつけるのは、地獄への緩やかな下り坂の第一歩なのか、純粋な教条主義者だけが反対する完全に合理的な行動なのか、悩んでいる」。私たちが不安になるのは、どちらもあり得るからである。

1.3 レイヤの実行場所の選択

本書の大半は、論理的なレイヤについての説明である。つまり、システムをいくつかに分割して、システムの異なる部分間の結合を減らすことの説明である。レイヤ間の分割は、すべてのレイヤが 1 台の物理的なマシンで実行されていたとしても役立つことである。しかし、システムの物理的な構造によって違いがもたらされる場所がある。

多くの IS アプリケーションにとって、クライアントであるデスクトップマシンとサーバのどちらで処理を実行すべきかは大きな問題である。

最もシンプルなのは、サーバですべてを実行することである。Web ブラウザーを使用する HTML フロントエンドは、これを実行するための優れた方法である。サーバ側で実行することの最大の利点は、サーバは限られた範囲内にあるので、アップグレードと修正が容易に行えることである。無数のデスクトップマシンに配備したり、サーバと同期を取ったりすることをあれこれ悩む必要はない。他のデスクトップソフトウェアとの互換性についても心配する必要はない。

クライアント側で実行することの利点は、レスポンス性や切断時の操作である。サーバで実行するロジックには、クライアントからサーバへの往復が必要である。ユーザが何か操作して即座のフィードバックを求めるとき、この往復は障害になる。ネットワーク接続も必要である。ネットワークなんてどこにでもあるかもしれないが、私が本書を書いている時点では、高度 9000 メートルの上空にはない。いずれどんな場所でもネットワークが使えるようになるとしても、無線の受信範囲が無限大に広がるまで待てず、今すぐに作業を行いたいユーザもいるのである。切断時の操作については別の問題があるが、残念ながら本書では触れる余裕がない。

さて、これらを基にレイヤごとの選択肢を考えてみる。データソースは、常にサーバでだけ実行される。例外は、切断時にも操作を行いたいときに、サーバの機能を適切かつ強力なクライアントに複製する場合である。この場合、切断されたクライアントからデータソースに対して加えられた変更は、サーバと同期させる必要がある。すでに述べたとおり、これらの問題は別の機会または別の執筆者に委ねることにしよう。

プレゼンテーションを実行する場所は、UI の種類によって決まることが多い。リッチク

ライアントの実行は、クライアントでプレゼンテーションを実行するのとほとんど同義である。Web インタフェースはたいていサーバ上で実行される。いくつか例外があり、稀にではあるが、たとえばクライアントソフトウェアを遠隔操作したり（Unix の X サーバなど）、デスクトップで Web サーバを実行したりすることがある。

B2C システムを構築する場合は、選択肢はない。誰かが TRS-80 でオンラインショッピングを行いたいと思っているときに、それを拒否するわけにはいかないだろう。このような場合は、サーバですべての処理を行い、ブラウザーで扱える HTML を提供する。HTML を使用すると、すべての意思決定にクライアントからサーバへの往復が必要となり、これによってレスポンス性が損なわれる。ブラウザーのスクリプトやダウンロード可能なアプレットである程度遅延を緩和できるとしても、そのためにブラウザーの互換性が減少し、他の頭痛の種が生じるかもしれない。HTML が純粋なほど、精神衛生上は楽である。

IS 部門が一台一台愛情込めてデスクトップを手作りしたとしても、精神的な安らぎは大きな魅力である。クライアントを最新の状態に保つことと、他のソフトウェアとの互換性によるエラーを回避することは、シンプルなリッチクライアントシステムも抱える問題である。

リッチクライアントのプレゼンテーションをユーザが望むのは、ユーザが実行するには複雑すぎるタスクがあり、使いやすいアプリケーションにするためには、Web GUI が提供する以上の機能を必要とするからである。しかし、ユーザは Web フロントエンドを使いやすくする方法に徐々に慣れ、リッチクライアントのプレゼンテーションの必要性は少なくなってきている。今のところ私としては、できるだけ Web プレゼンテーションを使用し、他に選択肢がないときだけリッチクライアントを使用することを勧めたい。

最後にドメインロジックが残っている。ビジネスロジックは、すべてサーバで実行することも、すべてクライアントで実行することも、サーバとクライアントに分割して実行することもできる。繰り返しになるが、すべてサーバで実行することが、保守を容易にするための最善の選択である。クライアントへの移行が必要になるのは、レスポンス性か切断時の操作を優先するときである。

クライアントでロジックの一部を実行する必要がある場合は、クライアントでそのロジックすべてを実行すること（少なくともロジックを一箇所に存在させるようにすること）を考えてみよう。これはリッチクライアントには適している。クライアントマシンで Web サーバを実行しても、切断時の動作に対処する方法にはなるが、レスポンス性の向上には役立たない。このような場合は、**トランザクションスクリプト**か**ドメインモデル**のどちらかを使用して、プレゼンテーションから分離したモジュールにドメインロジックを保持することができる。クライアントにすべてのドメインロジックを持ち込むと、バージョンアップと保守の作業が増えるという問題がある。

デスクトップとサーバの両方に分割すると、どこにロジックがあるのかわからなくなるので、クライアントにとってもサーバにとっても最悪の事態となる。このような分割を行うの

は、クライアントで実行すべきドメインロジックが少ししかないときである。この場合の秘訣は、このロジックをシステムの他の部分に依存しない内蔵モジュールに隔離することである。この方法で、クライアントまたはサーバのモジュールを実行できる。ややこしい操作を必要とするが、目的を成し遂げるための優れた方法ではある。

処理ノードを一度選択すると、1つのノードで動かすにせよ、クラスタ内の複数のノードで動かすにせよ、単一プロセスですべてのコードを保持する必要がある。絶対に必要な場合を除いて、レイヤを別々のプロセスに分離してはいけない。別々のプロセスに分離すると、**リモートファサード**や**データ変換オブジェクト**のような処理を追加する必要があるので、パフォーマンスが低下し、複雑性も大きくなる。

このような処理をJens Coldeweyが**複雑性のブースター**（complexity booster）と呼んでいることを覚えておくとよい。複雑性のブースターには、分散、明示的なマルチスレッド化、パラダイムの食い違い（オブジェクトかリレーショナルか）、マルチプラットフォームの開発、（1秒当たり100トランザクション以上といったような）極めて高いパフォーマンス要求などがある。これらのコストはすべて高い。どうしても避けられない場合も無論あるが、開発と保守の両面でコストがかさむことを忘れてはいけない。

第2章

ドメインロジックの構築

ドメインロジックの構築について、私はこれを**トランザクションスクリプト**、**ドメインモデル**、**テーブルモジュール**という3つの主要なパターンに分類している。

ドメインロジックを格納するための最もシンプルな手法は、**トランザクションスクリプト**である。**トランザクションスクリプト**は、本質的にはプレゼンテーションからの入力を取得するための手続きであり、妥当性確認と計算によって入力を処理し、データベースにデータを格納し、他のシステムの操作を呼び出す。そして、多くのデータを使ってプレゼンテーションに応答し、さらに計算を行ってその応答を体系的に書式化する。ユーザが望んでいるアクションのための手続きを1つずつ構築するのが基本である。そのため、このパターンをアクション用のスクリプト、またはビジネストランザクションとして考えることができる。単一のインラインコードである必要はなく、いくつかのサブルーチンに分割され、これらのサブルーチンは異なる**トランザクションスクリプト**間で共有することができる。しかし、このパターンは各アクションの手続きにより起動するので、たとえば小売システムなら、チェックアウト、ショッピングカートへの追加、配送状況の表示のために、それぞれ**トランザクションスクリプト**があるだろう。

トランザクションスクリプトには以下のような長所がある。

- 開発者なら大半の人が理解できるシンプルな手続き型モデルである。
- **行データゲートウェイ**や**テーブルデータゲートウェイ**を使用するシンプルなデータソースレイヤと適切に連携する。
- トランザクションの境界を設定する方法がはっきりしている。トランザクションをオープンして開始し、それをクローズして終了する。ツールを使えば容易にバックグラウンドで処理できる。

残念ながら短所も多い。ドメインロジックが複雑になると、短所が見えてくる。いくつか

のトランザクションが同じ動作を行う必要があるので、コードが重複することが多い。その重複の一部は共通のサブルーチンを抜き出すことで対処できるが、多くは削除するのも発見するのも難しい。結局は、明確な構造を持たないルーチンの絡まり合いのようなアプリケーションができあがることになる。

もちろん複雑なロジックはオブジェクトが関わるところであり、オブジェクト指向ではこの問題を**ドメインモデル**で対処する。**ドメインモデル**では、少なくとも概要レベルで、主にドメイン内の名詞について体系化したモデルを構築する。このため、たとえば賃貸システムなら、賃貸借契約、資産などのクラスがあるだろう。妥当性確認や計算のためのロジックはドメインモデルに配置されるので、出荷オブジェクトには、配送料を計算するためのロジックが含まれるだろう。さらに請求書計算のルーチンもあるだろうが、このような手続きは**ドメインモデル**のメソッドに速やかに委譲される。

トランザクションスクリプトではなく**ドメインモデル**を使用することは、オブジェクト指向のユーザが言うところのパラダイムシフトのエッセンスである。1つのルーチンがユーザアクションのすべてのロジックを持つのではなく、各オブジェクトが自身に関連する部分のロジックを担当する。**ドメインモデル**に慣れていないユーザは、振る舞いがどこで行われているかを見つけるために各オブジェクトを探し回らなければならないので、ドメインモデルの使い方を学ぶ際にフラストレーションを感じることが多い。

この2つのパターンの本質的な違いを、シンプルな例を使って理解するのは難しい。それでも私は、パターンの説明をする際に、シンプルなドメインロジックを両方のパターンで構築し、それらの相違を把握しようとしたことがある。違いを見つけるための最も容易な方法は、それぞれのパターンに従ってシーケンス図を描いてみることだ（図2.1と図2.2）。ここでの重要な問題は、製品（Product）によって、所定の契約（Contract）の収益を認識するためのアルゴリズムが異なることである（詳細は第9章を参照）。計算メソッドは、その契約がどの製品のものであるかを決定し、正しいアルゴリズムを適用してから、計算結果を把握するための収益認識（Revenue Recognition）オブジェクトを作成しなければいけない（話を単純にするために、データベースの相互作用の問題はここでは考えない）。

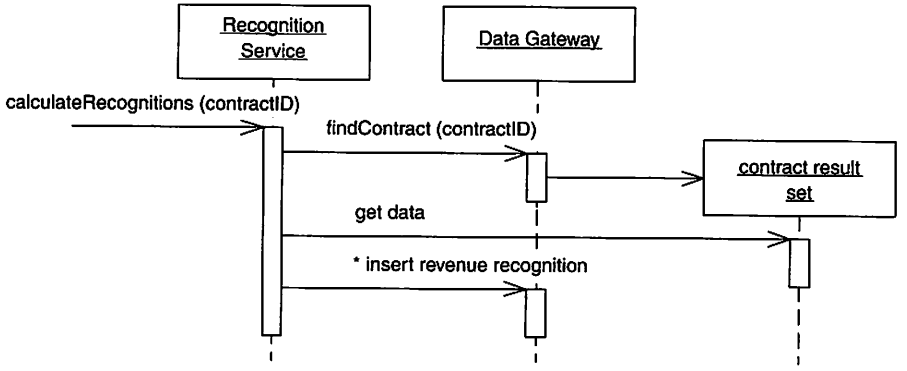


図 2.1 ——トランザクションスクリプトによる収益認識の計算

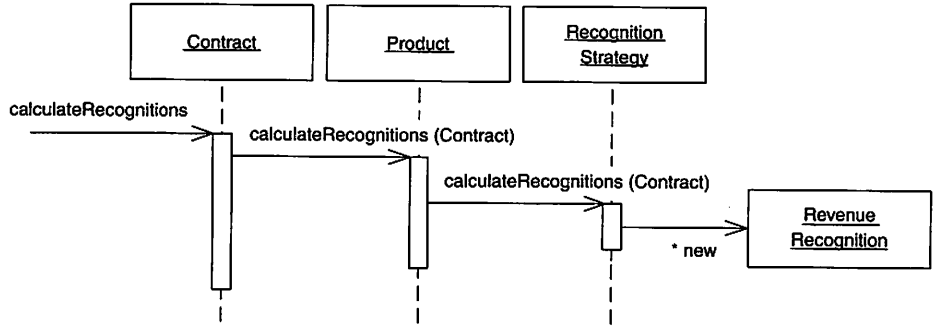


図 2.2 ——ドメインモデルによる収益認識の計算

図 2.1 では、**トランザクションスクリプト**のメソッドがすべての作業を行う。基礎となるオブジェクトは**テーブルデータゲートウェイ**で、これはトランザクションスクリプトにデータを渡すだけである。

これに対し、図 2.2 には複数のオブジェクトが存在し、Recognition Strategy（認識ストラテジー）オブジェクトが結果を作成するまで、各オブジェクトは振る舞いの一部を別のオブジェクトへ転送している。

ドメインモデルが持つ価値は、いったん慣れてしまえば、ロジックが複雑になっても十分に体系化された方法で対処できる多くの技法があることである。収益認識の計算のアルゴリズムが増えても、新しい Recognition Strategy オブジェクトを追加することでこのアルゴリズムを追加できる。**トランザクションスクリプト**の場合は、スクリプトの条件分岐ロジックを追加して、アルゴリズムを追加することになる。私のようにオブジェクトに傾倒している者なら、かなりシンプルなケースでも**ドメインモデル**の方を選びたくなるだろう。

ドメインモデルのコストは、それを使用するときの複雑性とデータソースレイヤの複雑性

から生じる。リッチなオブジェクトモデルを使用したことのない新規のユーザが、リッチな**ドメインモデル**に慣れるまでには時間がかかる。開発者でさえこのパターンを使用するプロジェクトで数カ月間作業してみてはじめてパラダイムシフトを経験することが多い。けれども、いったん**ドメインモデル**を使うことに慣れてしまえば、もう離れられなくなり、その代わり将来の仕事は楽になる。こうして、私のようなオブジェクトしか好きになれない偏屈者ができるわけだ。しかし中には、このパラダイムシフトを経験しない開発者も少ないながらいるようである。

パラダイムシフトを経験できたとしても、データベースへのマッピングがまだ残っている。**ドメインモデル**がリッチになるにつれ、リレーショナルデータベースへのマッピングは複雑になる（ふつうは**データマッパー**を使用する）。洗練されたデータソースレイヤは、固定費のようなものだ。優れたレイヤを得るには多額の経費（購入する場合）、または時間（構築する場合）がかかるが、レイヤが取得できれば多くのことを行える。

ドメインロジックを構築するための3つ目の選択肢は、**テーブルモジュール**である。**テーブルモジュール**と**ドメインモデル**は、一見するとどちらもContract（契約）、Product（製品）、Revenue Recognition（収益認識）のクラスを持っているので、似ているように見える。大きな違いは、**ドメインモデル**には契約ごとにContractのインスタンスがあるのに対し、**テーブルモジュール**には1つのインスタンスしかないことである。**テーブルモジュール**は、**レコードセット**に対処するように設計されている。このため、**テーブルモジュール**というContractのクライアントは、データベースに最初にクエリーを発行して、**レコードセット**を作成する。また、Contractオブジェクトを作成し、**レコードセット**を引数として渡す。こうして、クライアントは契約に関する操作を呼び出していろいろなことが行えるようになる（図2.3）。個別の契約に対して何かを行いたい場合は、IDを渡さなければいけない。

テーブルモジュールはさまざまな意味で**トランザクションスクリプト**と**ドメインモデル**の中間に位置する。直線的手続きではなく、テーブルを中心にドメインロジックを構築することで、構造がわかりやすく、重複の発見と除去が容易になる。しかし、精度の高いロジック構造を得るために**ドメインモデル**が用いる多くの技法（継承やストラテジー、その他のオブジェクト指向パターン）は使用できない。

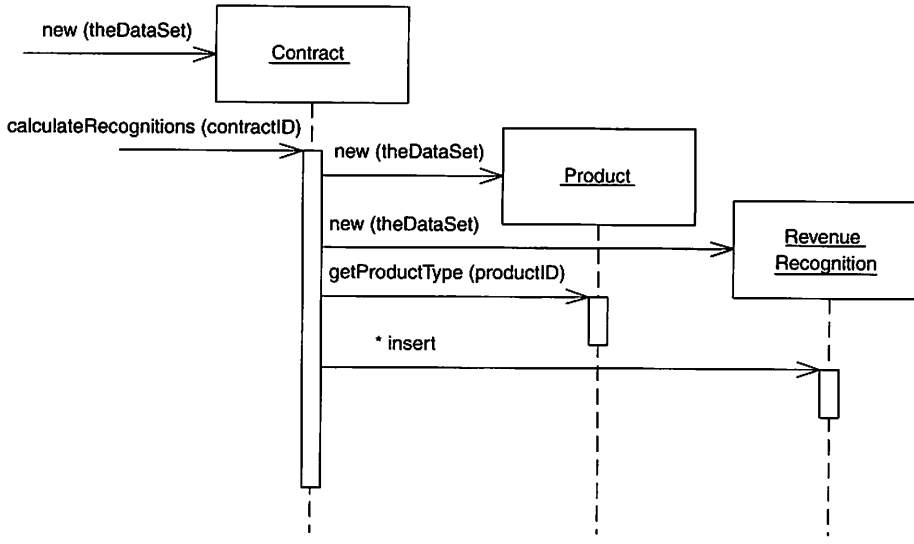


図 2.3 ——テーブルモジュールによる収益認識の計算

テーブルモジュールの最大の長所は、他のアーキテクチャにうまく適合することである。多くの GUI 環境は、レコードセットにまとめられた SQL クエリーの結果に従って動作するように構築されている。テーブルモジュールはレコードセット上で動作するので、クエリーの実行、テーブルモジュールの結果の操作、表示用の操作データの GUI への送信などを容易に行うことができる。また、さらに詳細な妥当性確認と計算を行うために、テーブルモジュールを使用することもできる。多くのプラットフォーム、特に Microsoft の COM と.NET は、この開発スタイルを採用している。

2.1 選択

この 3 つのパターンからどうやって選ぶか。これは容易な選択ではなく、ドメインロジックがどれだけ複雑かによって異なる。図 2.4 は PowerPoint のプレゼンテーションでよく見る類のあまり科学的ではないグラフで、座標軸にまったく数値が示されていないのが残念だが、それでも 3 つのパターンに関する私の見解を視覚化するには役立つ。ドメインモデルはシンプルなドメインロジックに対して使用するには魅力的ではない。このパターンを理解するのは難しくデータソースも複雑なので、開発に多くの努力が必要になり、しかもその見返りが小さいからである。しかし他の手法には、ドメインロジックが複雑になると機能を追加するのが飛躍的に難しくなるという難点がある。

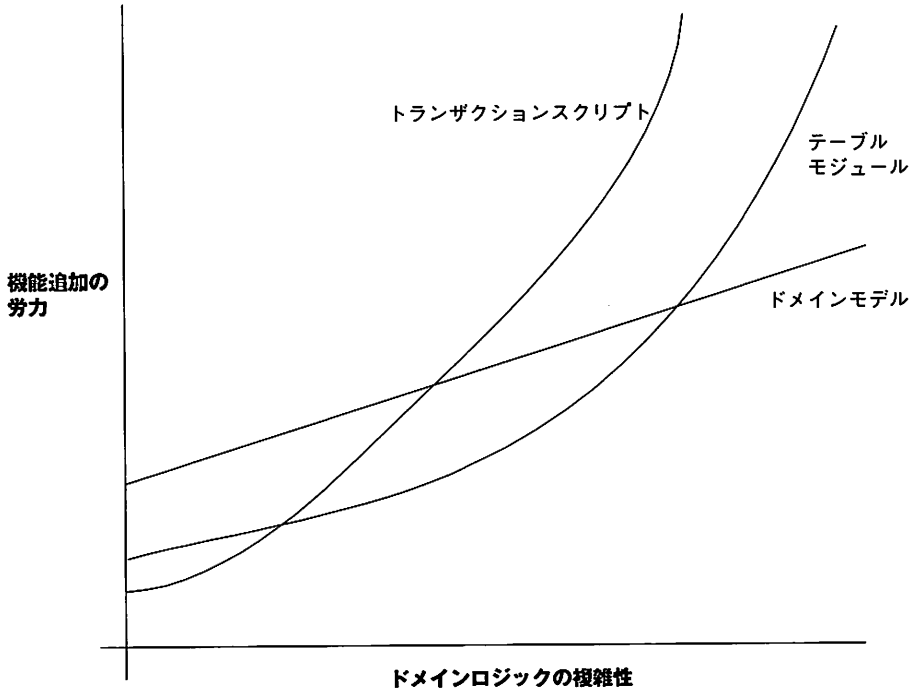


図 2.4 ——ドメインロジックの各形式の複雑性と労力の関係

肝心なのはもちろん、自分のアプリケーションがx軸のどこに位置するかを見極めることである。これを見極めるためには、ドメインロジックの複雑性が7.42を超えたら、**ドメインモデル**を使用すべきだ。しかし残念なことに、ドメインロジックの複雑性を測定する方法は誰も知らないのである。そこで実際には、初期の要求分析を行える熟練した人を見つけ、その人に判定してもらわなければならない。

この図の曲線にはいくつかの修正要因がある。**ドメインモデル**を熟知したプロジェクトチームなら、このパターンの使い方を習得するための初期コストはかからない。データソースの複雑性について言えば、他の手法ほどコストは削減できない。それでも私は、優秀なプロジェクトチームには**ドメインモデル**を使用するように勧めることが多い。

テーブルモジュールの魅力は、共通の**レコードセット**構造を環境がサポートしているかどうかによって違ってくる。**レコードセット**用の多くのツールが動作する.NETやVisual Studioのような環境であれば、**テーブルモジュール**はとても魅力的である。.NET環境で**トランザクションスクリプト**を使用する理由はない。しかし、**レコードセット**用の特別なツールがないときは、わざわざ**テーブルモジュール**は使用しないだろう。

一度行った決定はまったく変えられないわけではないとしても、変更するのはなかなか難しいものだ。このため、どのような決定を行うかについては前もって十分に思案しなければ

ならない。決定を間違えていることに気付いた場合、**トランザクションスクリプト**を使用しているのであれば、**ドメインモデル**を使用する方向へのリファクタリングをためらわずに行うべきだ。しかし、ドメインモデルを使用していてデータソースレイヤを簡略化できないのであれば、**トランザクションスクリプト**へ移行するのはあまり意味がない。

これら3つのパターンは相互に排他的な選択肢ではない。実際にドメインロジックの一部に**トランザクションスクリプト**を使用し、それ以外に**テーブルモジュール**または**ドメインモデル**を使用することは珍しくない。

2.2 サービスレイヤ

ドメインロジックを扱うときの共通の手法は、ドメインレイヤを2つに分割することである。**サービスレイヤ**は、**ドメインモデル**または**テーブルモジュール**の上に配置する。**サービスレイヤ**はふつう**ドメインモデル**か**テーブルモジュール**とともに使用する。**トランザクションスクリプト**だけを使用するドメインレイヤは、レイヤを分割するほど複雑ではないからだ。プレゼンテーションロジックは、アプリケーションのAPIの役割を果たす**サービスレイヤ**だけを通してドメインとやりとりする。

サービスレイヤは明確なAPIを提供するだけでなく、トランザクションの制御やセキュリティの確保のために最適な場所である。これは、**サービスレイヤ**に各メソッドを置いてトランザクションとセキュリティの特性を記述するシンプルなモデルである。個別のプロパティファイルを使用するのが共通の選択肢だが、.NETの属性はコードで直接記述することができる。

サービスレイヤでは、どの程度の振る舞いをそこに加えるかを決めることが重要である。最小限にする場合は、**サービスレイヤ**をファサードにする。これにより、実際の振る舞いはすべて下位のオブジェクトにあり、**サービスレイヤ**は下位のオブジェクトにファサードへの呼び出しを転送するだけになる。この場合、**サービスレイヤ**はユースケース中心の配置になるので、使いやすいAPIを提供できる。また、トランザクションラッパーとセキュリティチェックを追加するのにも便利である。

これに対して、多くのビジネスロジックは**サービスレイヤ**内の**トランザクションスクリプト**に置かれる。下位のドメインオブジェクトはとてもシンプルである。それが**ドメインモデル**であれば、データベースと一対一の関係になるので、**アクティブレコード**などのシンプルなデータソースレイヤを使用することができる。

これらの選択肢の中間には、**コントローラ－エンティティ**という振る舞いの組み合わせがある。この名前は、[Jacobson et al.]に大きな影響を受けた共通的なプラクティスに由来している。ここでの要点は、**トランザクションスクリプト**内に配置される1つのトランザク

ションまたはユースケースに特有のロジックを持つことである。そのロジックは通常コントローラまたはサービスと呼ばれる。これらは、後で説明する**モデルビューコントローラ**や**アプリケーションコントローラ**の入力コントローラとは異なるコントローラなので、**ユースケースコントローラ**と呼ぶことにする。複数のユースケースで使用される振る舞いは、ドメインオブジェクトに移り、エンティティと呼ばれる。

コントローラーエンティティ手法は一般的だが、私はあまり好きではない。ユースケースコントローラは、**トランザクションスクリプト**と同じように、コードの重複を促進する傾向がある。私の考えでは、いったん**ドメインモデル**を使用すると決めたらそれを貫き、このモデルを中心にすべてを進めるべきである。唯一の例外は、**行データゲートウェイ**とともに**トランザクションスクリプト**を使用する設計から始めた場合である。重複する振る舞いを**行データゲートウェイ**に移行することには意味があり、これにより**アクティブレコード**を使用するシンプルな**ドメインモデル**になる。しかし、私ならこのように始めたりはしない。このように行うのは、欠陥のある設計を改善する場合だけである。

私はビジネスロジックを含むサービスオブジェクトを持つべきでないと言っているのではない。サービスオブジェクトの固定レイヤを必ずしも作成しなくてもよいと言っているのである。手続き型サービスオブジェクトはロジックを抽出するのに役立つこともあるが、私個人はアーキテクチャレイヤとしてではなく、必要なときにだけ使用することが多い。

私が気に入っている方法は、**サービスレイヤ**が必要な場合でも、できる限り薄いレイヤにすることである。私のいつもの方法では、本来なら**サービスレイヤ**は1つもいらないという前提で、アプリケーションに必要だと思われる場合にだけ追加する。ただし、常に大量のロジックとともに**サービスレイヤ**を使用する優れた設計者はたくさんいるので、私のこの方法は無視してもらって結構である。Randy Stafford はリッチな**サービスレイヤ**を使用して多くの成功を収めているので、彼には**サービスレイヤ**のパターンを本書のために書いてくれるように依頼した。

第3章

リレーショナルデータベースへのマッピング

データソースレイヤの役割は、アプリケーションが動作するために必要なさまざまなインフラと通信することである。その際に発生する問題の大部分はデータベースとのやり取りである。現在のシステムの大半において、データベースとはリレーショナルデータベースのことを指す。メインフレームのISAMファイルやVSAMファイルなど、旧式の格納フォーマットのデータもまだ多く残っているとはいえ、今日システム構築に携わる人の苦労の種となっているのはリレーショナルデータベースの扱いである。

リレーショナルデータベースが成功した最大の理由は、データベースと通信する際の標準言語となったSQLの存在である。SQLには、面倒で複雑なベンダー固有の拡張も多いが、基本的な構文は共通的かつ汎用的である。

3.1 アーキテクチャに関するパターン

最初に取り上げるパターンは、ドメインロジックがデータベースとやり取りするためのアーキテクチャに関するパターンである。ここで行う選択は設計全般に影響し、やり直しが難しいので注意が必要である。逆に、この選択はドメインロジックの設計の仕方によって大きく影響されるものでもある。

エンタープライズソフトウェアではSQLが広範囲で使われているが、その使用には落とし穴がある。多くのアプリケーション開発者はSQLを十分理解していない。その結果、効果的なクエリーとコマンドを定義できないという問題が生じる。SQLをプログラミング言語に組み込む技法はいろいろあるが、いずれもあまり優れたものではない。アプリケーション開発言語に適合したメカニズムを使用してデータにアクセスした方がましである。DBA（データベース管理者）はテーブルにアクセスするSQLを好むが、その方が最善のチューニング方法やインデックスの配置方法を理解しやすいからである。

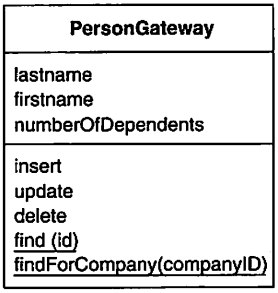


図 3.1 ——クエリーから返された行ごとに 1 つのインスタンスを持つ行データゲートウェイ

これらの理由から、ドメインロジックと SQL アクセスを分離し、異なるクラスに配置することが賢明である。これらのクラスを組織化する適切な方法は、データベースのテーブル構造に基づいて、データベーステーブルごとに 1 つのクラスを持つようにすることである。これらのクラスはそのテーブルに対し**ゲートウェイ**を形成する。アプリケーションの他の部分は SQL について知る必要はなく、データベースにアクセスする SQL はすべて容易に見つけられる。データベースを専門とする開発者には、何を行うべきかは明確である。

ゲートウェイを使用できる主な方法は 2 つある。最も明白な方法は、クエリーから返された行ごとにインスタンスを持つことである（図 3.1）。この**行データゲートウェイ**は、データに関するオブジェクト指向の考え方に自然に適合する手法である。

多くの環境では、**レコードセット**、つまりデータベースのテーブルの性質を持つ、テーブルと行で構成された包括的なデータ構造が提供される。また、**レコードセット**は包括的なデータ構造なので、アプリケーションの多くの部分に**レコードセット**を使用できる。GUI ツールが**レコードセット**と連携するコントロールを持つことはとても一般的である。**レコードセット**を使用する場合は、データベースの各テーブルに対して 1 つのクラスがあればよい。**テーブルデータゲートウェイ**（図 3.2 参照）は、データベースを検索して**レコードセット**を返すメソッドを提供する。

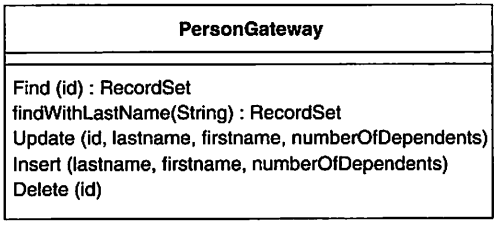


図 3.2 ——テーブルごとに 1 つのインスタンスを所持するテーブルデータゲートウェイ

私はシンプルなアプリケーションにもゲートウェイパターンの1つを使用することが多い。私のRubyとPythonのスクリプトを見れば、それがわかるだろう。SQLとドメインロジックの明確な分離がとても役立つと私は思っている。

テーブルデータゲートウェイが**レコードセット**と見事に適合しているので、**テーブルモジュール**を使用している場合、**テーブルデータゲートウェイ**を選択すべきなのは明らかである。また、**テーブルデータゲートウェイ**はストアドプロシージャの構築について考える際に使用できるパターンでもある。多くの設計者は、明示的なSQLではなく、ストアドプロシージャですべてのデータベースのアクセスを行うことを好む。この場合、ストアドプロシージャのコレクションをテーブル用の**テーブルデータゲートウェイ**の定義と考えることができる。私の方法では、カプセル化されたストアドプロシージャ呼び出しのメカニズムを保持するために、メモリ上の**テーブルデータゲートウェイ**でストアドプロシージャへの呼び出しをラップする。

ドメインモデルを使用している場合は、さらにいくつかの選択肢がある。もちろん**ドメインモデル**と一緒に、**行データゲートウェイ**または**テーブルデータゲートウェイ**を使用することができる。しかし、私個人の好みとしては、この方法は間接的すぎるか十分でないかのどちらかだ。

シンプルなアプリケーションでは、**ドメインモデル**の構造は複雑ではない。実際、その構造はデータベースの構造と厳密に一致し、データベーステーブルごとに1つのドメインクラスがある。このようなドメインオブジェクトは、適度に複雑なビジネスロジックだけを持っていることが多い。この場合、各ドメインオブジェクトがデータベースからの読み込みと保存を行い、**アクティブレコード**になることには意味がある（図3.3参照）。**アクティブレコード**についての別の考え方は、特に複数の**トランザクションスクリプト**に繰り返し現れるコードがある場合、**行データゲートウェイ**から開始して、クラスにドメインロジックを追加することである。

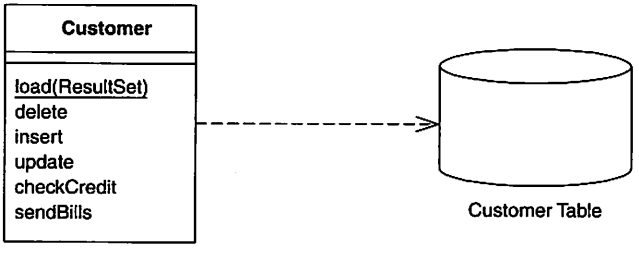


図3.3——アクティブレコードでは、Customerドメインオブジェクトがデータベーステーブルとの通信方法を知っている。

このような状況では、**ゲートウェイ**の余分な間接化は意味がない。ドメインロジックが複雑になり、豊富な**ドメインモデル**に移行するにつれ、**アクティブレコード**のシンプルな手法は崩壊し始める。小さいクラスにドメインロジックを組み込むので、テーブルとドメインクラスとの一対一の組み合わせは失敗し始める。リレーショナルデータベースは継承に対応しないので、ストラテジー [Gang of Four]や他の素晴らしいオブジェクト指向のパターンを使用することが困難になる。ドメインロジックが面倒になると、常にデータベースに通信しなくてもテストできるようにしたくなる。

ドメインモデルが豊富になると、これらの圧力すべてによって間接化が強要される。この場合、**ゲートウェイ**で問題の一部を解決できるが、データベースのスキーマと結合している**ドメインモデル**が残っている。結果として、**ゲートウェイ**のフィールドからドメインオブジェクトのフィールドに変換されるので、ドメインオブジェクトは複雑になる。

これに対するより優れた方法は、ドメインオブジェクトとデータベーステーブルとのマッピングを間接参照のレイヤに行わせ、データベースから**ドメインモデル**を完全に分離させることである。**データマッパー**（図3.4参照）は、データベースと**ドメインモデル**間の読み込みと保存のすべてに対処し、両方の変更を独立して行うことができる。データベースを対応付けるアーキテクチャの中で**データマッパー**は最も複雑だが、そのメリットは2つのレイヤを完全に分離することである。

ゲートウェイを**ドメインモデル**用の主な永続メカニズムとして使用することはお勧めできない。ドメインロジックがシンプルで、クラスとテーブル間に密接な対応がある場合は、**アクティブレコード**がシンプルな方法である。さらに複雑な場合は、**データマッパー**が必要になる。

これらのパターンは、完全に相互排他的というわけではない。ここでは、メモリ上のデータをデータベース内に保存する方法として、主な永続メカニズムを考えている。このため、これらのパターンの1つを選択することになるだろう。これらを混用しても、混乱を招く結果に終わるだけなので避けたいところだ。**データマッパー**を主な永続メカニズムとして使用していても、外部インタフェースとして扱われているテーブルまたはサービスをラップするために**ゲートウェイ**データを使用するだろう。

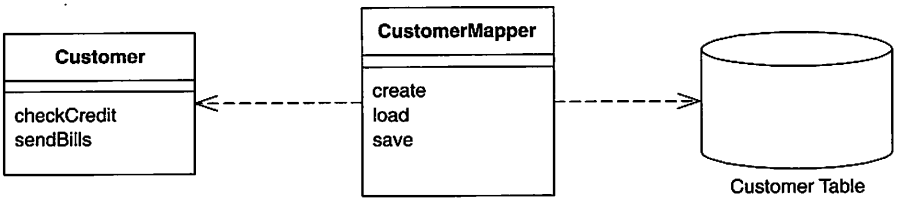


図3.4 ——ドメインオブジェクトとデータベースを分離するデータマッパー

ここでの考え方とパターン自体における考え方の説明では、私は「テーブル」という用語を使用することが多い。しかし、これらの技法の多くは、ビュー、ストアドプロシージャでカプセル化したクエリー、一般的によく使用される動的なクエリーにも同様に適用できる。残念なことに、テーブル、クエリー、ストアドプロシージャに対して幅広く使用される用語はないので、テーブルのデータ構造を意味する「テーブル」を使用する。私はビューを仮想テーブルとして考えることが多い。この仮想テーブルは、SQL がビューを考える方法でもある。テーブルの検索と同様にビューを検索するためには、この仮想テーブルで使用される構文と同じ構文が使用される。

ビューとクエリーに対する更新は明らかに複雑である。その理由は、常にビューを直接更新できるわけではなく、その基盤となるテーブルを操作しなければいけないからである。この場合、ビューとクエリーを適切なパターンでカプセル化することは、1 箇所で更新ロジックを実装するための優れた方法であり、その結果シンプルで信頼性のあるビューを使用できるようになる。

この方法でビューとクエリーを使用することについての問題の 1 つは、ビューを形成する方法を理解していない開発者を驚かせる矛盾が生じてしまうことである。ビューやクエリーの形成方法を知らない開発者は 2 つの異なる構造で更新を実行するので、基盤となる同じテーブルが更新され、最初の更新を 2 度目の更新で上書きすることになる。更新ロジックが適切な妥当性を確認するので、このような矛盾したデータを取得することはないが、開発者は驚くことになるだろう。

また、最も複雑な**ドメインモデル**の永続化のための最もシンプルな方法も説明すべきだろう。オブジェクトを扱い始めて間もないころ、多くの人がオブジェクトとリレーショナルデータベースとの間には根本的な「インピーダンス・ミスマッチ」があると気付いた。このため、オブジェクト指向データベースに関する多くの取り組みが行われ、ディスク記憶装置にオブジェクト指向のパラダイムがもたらされた。オブジェクト指向データベースに関しては、マッピングの心配をする必要はない。相互に関連したオブジェクトの大規模な構造に対処し、データベースはオブジェクトをディスクから出し入れするタイミングを割り出す。また、トランザクションを使用して、更新をひとつに集めることができ、データ格納の共有が可能になる。プログラマにとって、これはディスク記憶装置が透過的に支援する無限のトランザクションメモリのように感じられる。

オブジェクト指向データベースの主な長所は、生産性の向上である。対照試験は行っていないが、事例を観察した結果、プログラミング労力（保守に要する費用）の約 3 分の 1 はリレーショナルデータベースのマッピングに費やされる。

しかし、多くのプロジェクトはオブジェクト指向データベースを使用しない。その主な理由はリスクである。リレーショナルデータベースは、長い歴史のある大手のベンダーが支援する、広く理解された実績ある技術である。SQL は、あらゆる種類のツールに比較的標準

のインタフェースを提供する（パフォーマンスに関して私が唯一言えるのは、リレーショナルシステムのパフォーマンスとオブジェクト指向のパフォーマンスを比較する決定的なデータは見たことがないということである）。

オブジェクト指向データベースを使用できなくても、**ドメインモデル**がある場合は、O/R（オブジェクト/リレーショナル）マッピングツールの購入を真剣に考えるべきである。本書のパターンは、**データマッパー**を構築する方法を多く説明しているが、それでも複雑な取り組みである。ツールベンダーは長年この問題に取り組んでいて、手動で合理的に行われる処理よりも商用のO/Rマッピングツールの方が洗練されている。ツールは安くはないので、レイヤの作成と保守にかかるコストを考慮して価格を比較しなければいけない。

リレーショナルデータベースを扱えるオブジェクト指向データベース形式のレイヤを提供するという動きがある。JavaにおいてJDOは素晴らしいが、それがどのように機能するかを説明するのは早すぎる。本書の中で結論を出せるだけのJDOの経験が私にはない。

しかし、ツールを購入したとしても、パターンを理解しておく方がよい。適切なO/Rマッピングツールによって、データベースへのマッピングにおけるさまざまな選択肢を使用でき、パターンは異なる選択を使用するタイミングを理解するために役立つ。ツールがすべての労力をなくしてくれると思い込んではいけない。ツールは大部分の作業を行うが、O/Rマッピングツールの使用と調整には、少ない量ではあるが重要な作業を行う必要がある。

3.2 振る舞いに関する問題

O/Rマッピングについて話をする際は、一般的に構造的な側面（テーブルとオブジェクトを関連付ける方法）に焦点が当てられる。しかし、これを実行する際に最も困難なのは、アーキテクチャと振る舞いに関する側面である。アーキテクチャに関する主な手法はすでに説明した。次に考えるべきことは、振る舞いに関する問題である。

振る舞いに関する問題は、さまざまなオブジェクトをデータベースに読み込ませる方法と保存させる方法である。一見、大した問題ではないように思える。たとえば、顧客オブジェクトは、このタスクを行う読み込みメソッドと保存メソッドを持つことができる。実際、**アクティブレコード**では、このメソッドを持つという方法を採用すべきである。

メモリにオブジェクト群を読み込んで修正する場合、修正したオブジェクトを記録し、それらすべてをデータベースに書き戻さなければいけない。2、3のレコードを読み込むだけであれば、これは容易である。しかし、多くのオブジェクトを読み込むほど行うべきことが多くなる。これは特に、行の作成や修正を行う場合に、参照する行を修正する前に作成された行のキーが必要になるからである。これは、解決するには少し扱いにくい問題である。

オブジェクトを読み込んで修正する際は、処理するデータベースが一貫性のある状態であ

るようにしなければいけない。いくつかのオブジェクトを読み込んだ場合は、オブジェクトを処理している間に他のプロセスが同じオブジェクトを読み込んで変更することがないように、必ず読み込みを分離させることが重要である。分離させないと、オブジェクトに関して互いに矛盾した無効なデータを持つことになる。これは、並行性の問題であり、解決すべきではあるがとても扱いにくい問題である。これについては第5章で詳しく説明する。

これらの問題を解決するために不可欠なパターンは**ユニットオブワーク**である。**ユニットオブワーク**は、少しでも修正されたすべてのオブジェクトとともに、データベースから読み込んだすべてのオブジェクトを記録する。また、データベースへの更新方法にも対処する。プログラマは、明示的に保存メソッドを呼び出すアプリケーションプログラマの代わりに、コミットする処理単位を告げる。そして、この処理単位はすべてのデータベースに対する適切な振る舞いを順序付け、1箇所にすべての複雑なコミット処理を配置する。データベースとの振る舞いの相互作用がうまくいかなくなった場合は、**ユニットオブワーク**が必須のパターンになる。

ユニットオブワークについては、データベースマッピングのコントローラとしての役割を果たすオブジェクトとして考えることである。**ユニットオブワーク**がなければ、ドメインレイヤは一般的にコントローラとしての役割を果たし、データベースへの読み込みと書き込みを行うタイミングを決定する。**ユニットオブワーク**は、データベースのマッピングコントローラの振る舞いを1つのオブジェクトへ抜き出すことによって生じる。

オブジェクトを読み込む際は、同じオブジェクトを2度読み込まないように注意しなければいけない。2度読み込むと、1つのデータベース行に対応するオブジェクトがメモリ上に2つ存在することになる。両方を更新すると、すべてがとてもわかりにくくなる。これに対処するために、**一意マッピング**で読み込んだ各行の記録を保持する必要がある。データを読み込むごとに、最初に**一意マッピング**をチェックして、すでにデータを持っていないかを確認する。データがすでに読み込まれている場合は、そのデータに対する2つ目の参照を返すことができる。こうすれば、どの更新も適切に調整される。メリットとして、**一意マッピング**はデータベースのキャッシュを持つので、データベース呼び出しを回避することができるだろう。しかし、**一意マッピング**の主な目的は、パフォーマンスの改善ではなく、正確な一意性を維持することである。

ドメインモデルを使用している場合、注文オブジェクトの読み込みが関連する顧客オブジェクトを読み込むというように、リンクされたオブジェクトが一緒に読み込まれるような配置にすることが多い。しかし、多くのオブジェクトが連結している場合、どのオブジェクトの読み込みも膨大なオブジェクトのグラフをデータベースから抜き出すことになる。この無駄を回避するために、抽出するデータは減らすが、その後の必要に応じて、データを抜き出すためのドアを開けておく必要がある。**レイジーロード**はオブジェクト参照用のプレースホルダーがあることを期待する。このテーマに関しては多少のバリエーションがあるが、そ

のバリエーションすべてがオブジェクト参照を修正し、実際のオブジェクトを指す代わりに、プレースホルダーであるというマークを付けている。リンクをたどろうとする場合にだけ、実際のオブジェクトがデータベースから呼び出される。適切な場面で**レイジーロード**を使用すると、各呼び出しによってデータベースから過不足のないデータ抽出を行える。

3.3 データの読み込み

データの読み込み時には、メソッドを **find メソッド**として考えることを私は好む。このメソッドは、メソッドの構造を持つインタフェースで SQL の select 文をラップする。`find(id)` や `findForCustomer (customer)` などのメソッドがその例だろう。select 文に 23 の異なる句がある場合、これらのメソッドはとても扱いにくいが、幸いそのような例は稀である。

find メソッドを置く場所は、使用するインタフェースのパターンに依存する。データベースとの相互作用を行うクラスがテーブルベース（データベースのテーブルごとにクラスのインスタンスを 1 つ持つ）であれば、find メソッドと挿入や更新を組み合わせることができる。相互作用クラスが行ベース（データベースの行ごとに相互作用クラスを 1 つ持つ）の場合、この方法は利用できない。

行ベースのクラスでは find 操作を静的にすることができるが、データベースを替えることができなくなる。これは、**サービススタブ**のテスト目的のためにデータベースを交換できないことを意味する。この問題を回避するための最善の手法は、個別の find オブジェクトを持つことである。各 find クラスには、SQL クエリーをカプセル化する多くのメソッドがある。クエリーを実行すると、find オブジェクトは適切な行ベースのオブジェクトのコレクションを返す。

find メソッドに関して注意すべきことは、オブジェクトではなく、データベースで動作することである。たとえば、クラブにいるすべての人を検索するために、データベースに対してクエリーを発行する場合、メモリ上のクラブに追加した人のオブジェクトがクエリーによって参照されることはない。結果として、通常、最初にクエリーを発行することが賢明である。

データの読み込み時にパフォーマンスの問題が浮上することが多い。これに対しては、経験則による方法がいくつかある。

まず、1 度に複数の行の取得を行うようにした方がよい。特に、複数の行を取得するために同じテーブルに繰り返しクエリーを発行してはいけない。少なすぎるデータよりも多すぎるデータを取得する方が常に適切な場合が多い（悲観的な並行性制御によって多すぎる行をロックしてしまうことには注意しなければいけない）。このため、ドメインモデルにおいて主キーで特定可能な 50 人を取得しなければならない場合、200 人を取得するようなクエ

リーを構築し、そこから必要な50人を分離させるための何かのロジックを実行すればよい。通常、50の個別のクエリーを発行するよりも、不必要な行を戻す1つのクエリーを使用する方が適切である。

何度もデータベースを参照することを回避する別の方法は、ジョインを使用し、1つのクエリーで複数のテーブルを取得できるようにすることである。その結果返されるレコードセットは奇妙に見えるかもしれないが、高速化が可能になる。この場合、複数のジョインテーブルからのデータがあるゲートウェイを使用するか、1つの呼び出しでいくつかのドメインオブジェクトを読み込むデータマッパーを使用することになるだろう。

しかし、ジョインを使用している場合、クエリーごとに3つか4つのジョインに対応するようにデータベースが最適化されていることに注意すべきである。それ以上のジョインを行うと、キャッシュされたビューでこの多くを復元できるが、パフォーマンスが損なわれる。

データベースでは多くの最適化を行うことができる。これらの処理には、共通して参照されるデータのクラスタ化、インデックスの慎重な使用、データベースのメモリキャッシュ機能などがある。これらは本書の範囲外だが、優れたDBAなら守備範囲内である。

すべての場合において、独自のデータベースとデータを使用してアプリケーションのプロファイルを取得する必要がある。一般的な原則を考え方の指針とすることができるが、状況によって、常に独自のバリエーションがある。データベースシステムとアプリケーションサーバは、キャッシュのスキーマを洗練することが多い。また、アプリケーションに起こることを予測する方法はない。私は経験則を使用するたびに、驚くべき例外を知らされるので、パフォーマンスの調査と調整の時間は別に設けることにしている。

3.4 構造的なマッピングに関するパターン

一般的なオブジェクトとリレーショナルデータベースのマッピングに関する話題は、メモリ上のオブジェクトとデータベーステーブルとのマッピング時に使用する構造的なマッピングに関するパターンであることが多い。これらのパターンは、通常、テーブルデータゲートウェイには関連していないが、行データゲートウェイまたはアクティブレコードを使用する場合は、いくつかのパターンが必要になるだろう。おそらく、データマッパーに対しては、すべてのパターンを使用する必要がある。

3.4.1 関係のマッピング

ここでの中心的課題は、オブジェクトとリレーショナルデータベースがリンクに対処する方法の違いであり、2つの問題が生じる。1つ目の問題は、表現の相違である。オブジェクトは、メモリ管理環境か実行時のメモリアドレスによって保持される参照を格納することで

リンクに対処する。リレーショナルデータベースは、他のテーブルにキーを形成することでリンクに対処する。2つ目の問題は、オブジェクトが1つのフィールドからの複数の参照を処理するために容易にコレクションを使用できるのに対し、正規化はすべての関連リンクを単一の値にすることを強要する。これは、オブジェクトとテーブル間のデータ構造の逆転につながる。注文オブジェクトは、注文を参照する必要のない明細オブジェクトのコレクションを自然に持っている。しかし、テーブル構造はその逆で、注文は複数値フィールドを持つことができないので、明細が注文への外部キーの参照を持たなければいけない。

この表現の問題に対処する方法は、各オブジェクトのリレーショナルな一意性を**一意フィールド**としてオブジェクトに保持し、オブジェクト参照とリレーショナルキーとの間を双方向にマッピングするために、これらの値を参照することである。単調なプロセスだが、基本的な技法を理解すれば困難ではない。ディスクからオブジェクトを読み込む際は、**一意マッピング**をリレーショナルキーからオブジェクトへの照合テーブルとして使用する。テーブルの外部キーに出会うたびに、**外部キーマッピング**（図3.5参照）を使用して適切な相互のオブジェクト参照をつなげる。**一意マッピング**にキーがなければ、データベースを使用して取得するか、**レイジーロード**を使用する必要がある。オブジェクトを保存するごとに、適切なキーを使用して行に保存する。相互のオブジェクト参照は、対象オブジェクトのIDフィールドに置き換えられる。

これを基盤に、コレクション管理には、**複雑な外部キーマッピング**（図3.6参照）が必要になる。オブジェクトにコレクションがある場合は、他のクエリーを発行して、ソースオブジェクトのIDにリンクするすべての行を検索する必要がある（または、**レイジーロード**を使用してクエリーを回避することができる）。クエリーから返されたオブジェクトがそれぞれ作成され、コレクションに追加される。コレクションの保存には、各オブジェクトを保存し、ソースオブジェクトのための外部キーを確実に持つことが必要になる。特に、追加されたオブジェクトを発見したり、コレクションから削除したりしなければいけない場合は乱雑になる。これは要領を得れば繰り返すことができるようになる。これがメタデータベースのような形式の手法が大規模なシステムに採用される理由である（これについては後で説明する）。コレクションオブジェクトがコレクションのオーナの範囲外では使用されない場合、**依存マッピング**を使用してマッピングを単純化できる。

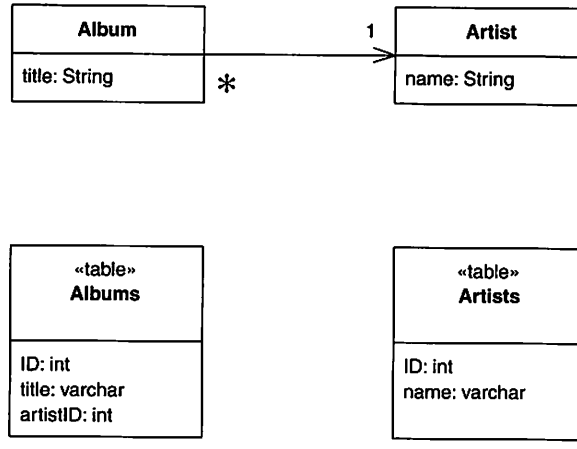


図 3.5 ——外部キーマッピングを使用した単一値フィールドのマッピング

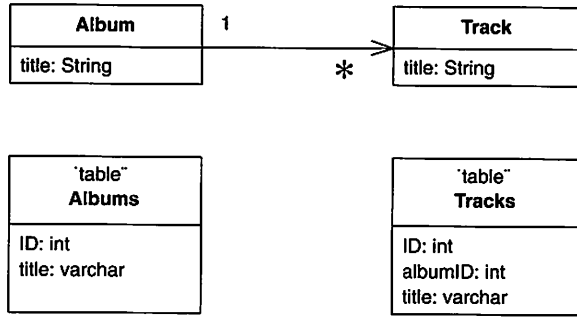


図 3.6 ——外部キーマッピングを使用したコレクションフィールドのマッピング

別の例では、両端にコレクションを持つ多対多の関係が使用される。たとえば、ある人は多くのスキルを持ち、各スキルは使用する人が誰だかわかっているとする。リレーショナルデータベースは、この関係に直接対処することはできないので、**関連テーブルマッピング**（図 3.7 参照）を使用し、多対多の関連に対処するために新規のリレーショナルテーブルを作成する。

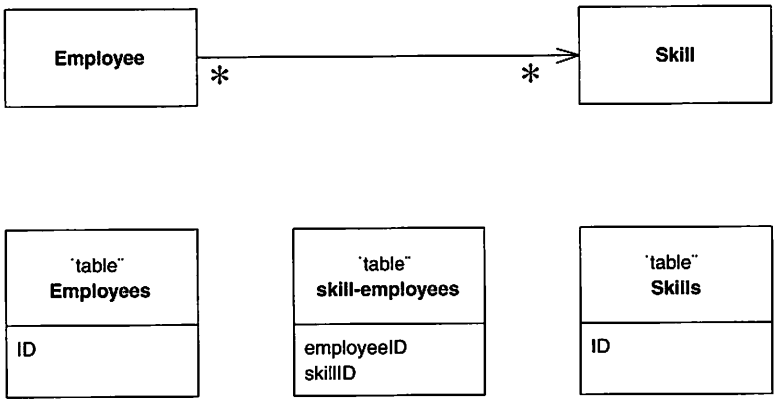


図 3.7 ——関連テーブルマッピングを使用した多対多の関連のマッピング

コレクションを扱う際の共通の認識は、コレクション内の順番への依存である。オブジェクト指向言語では、リストや配列などの順番の付いたコレクションを使用することが一般的である。実際、順番の付いたコレクションの使用によってテストが容易になる場合が多い。それでも、リレーショナルデータベースへの保存時に任意の順番の付いたコレクションを維持することはとても困難である。この理由から、コレクションを格納する際には順番の付いていない集合の使用を考慮した方がよい。他の選択肢としては、とても資源を消費するが、コレクションクエリーの実行時は常に、ソートの順序を決定することである。

参照整合性が更新を複雑にしてしまう場合もある。最新のシステムでは、トランザクションが終了するまで参照整合性の照合を保留することができる。この機能があれば使用すべきである。この機能がなければ、データベースは書き込みごとに照合を行う。この場合、適切な順序で更新を行うように注意しなければいけない。これを行う方法は本書の範囲外だが、更新のトポロジカルソートを行うことが1つの技法である。他の技法としては、どの順序でどのテーブルに書き込みを行うかを直接書くことである。そうすることで、トランザクションの過度のロールバックを生じさせるデータベース上のデッドロック問題を減少できる場合がある。

一意フィールドは、外部キーとしてオブジェクト参照に使用されるが、すべてのオブジェクトの関係がこのように永続化される必要はない。日付範囲や貨幣オブジェクトなどの小さな**バリューオブジェクト**には、データベース内で単独のテーブルとなるだけの情報は存在しない。**バリューオブジェクト**の全フィールドは、**組込バリュー**として**バリューオブジェクト**をリンクしているオブジェクトの中に格納すればよい。**バリューオブジェクト**は値なので、各フィールドを読み込むたびにオブジェクト生成を行えばよいのであって、わざわざ**一意マッピング**を行う必要はない。また、**バリューオブジェクト**を書き込むのも容

易で、各フィールドを取り出して、対象となるテーブルに設定するだけである。

オブジェクトのクラスタ全体を取得し、**シリアライズLOB**のようにテーブルに1つの列として保存することで、大規模にこれを行うことができる。LOBは、「Large OBject：大きいオブジェクト」の略語であり、バイナリ（BLOB）にも、テキスト（CLOB：Character Large Object）にもなる。オブジェクトの一群をXML文書として直列化することは、階層的オブジェクト構造の代替になる明確な方法である。これを行うことで、一度の読み込みでリンクされた小さなオブジェクトの集まりすべてを取得できる。データベースは相互に強く関連した小さいオブジェクトの扱いに慣れていないことが多く、小さいデータベース呼び出しに多くの時間を費やす。組織表や部品表などの階層的構造で、**シリアライズLOB**は多くのデータベースラウンドトリップ（データのつながりが巡回すること）を節約できる。

シリアライズLOBの短所は、何が起こっているかをSQLが認識していないので、データ構造に対し移植可能なクエリーを作成できないことである。再度述べるが、XMLはここでの救援方法となるだろう。今のところ非標準的であるものの、SQL呼び出し内のXPathクエリー表現の組み込みができるようになるだろう。結果として、**シリアライズLOB**は、格納された構造の部分の検索が不要である場合に使用することが最善である。

通常、**シリアライズLOB**は、アプリケーションの一部を構成するオブジェクトの比較的分離した集まりにとって最適である。使用しすぎると、データベースはただのトランザクション可能なファイルシステムになってしまう。

3.4.2 | 継承

先に述べた階層では、リレーショナルシステムが昔から苦手とするパーツツリーなどの複合階層について説明した。リレーショナル問題を引き起こす他の種類の階層化には、継承によってリンクされたクラス階層がある。SQLには継承を行うための標準的な方法がないので、マッピングを遂行する。継承構造には、基本的に3つの選択肢がある。**シングルテーブル継承**（図3.8参照）では、階層にあるすべてのクラスに対し1つのテーブルがある。**具象テーブル継承**（図3.9参照）では、具象クラスごとに1つのテーブルがある。**クラステーブル継承**（図3.10参照）では、階層にあるクラスごとに1つのテーブルがある。

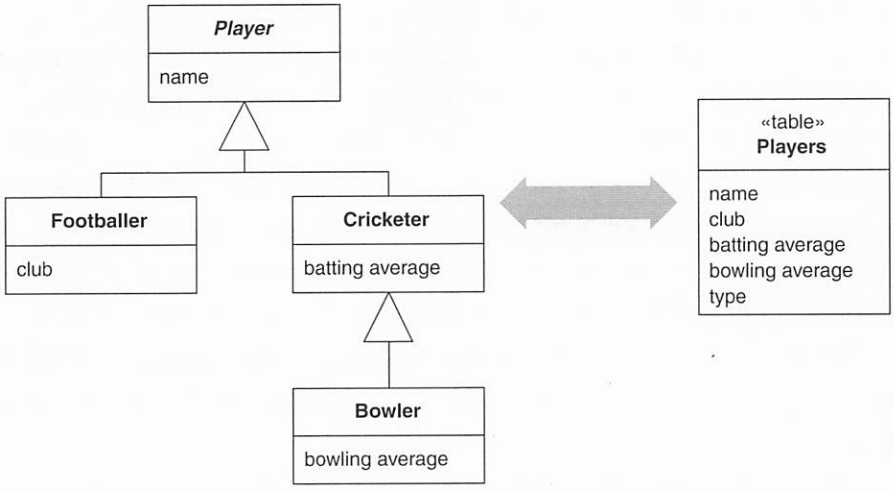


図 3.8 ——階層にあるすべてのクラスを格納するために 1 つのテーブルを使用するシングルテーブル継承

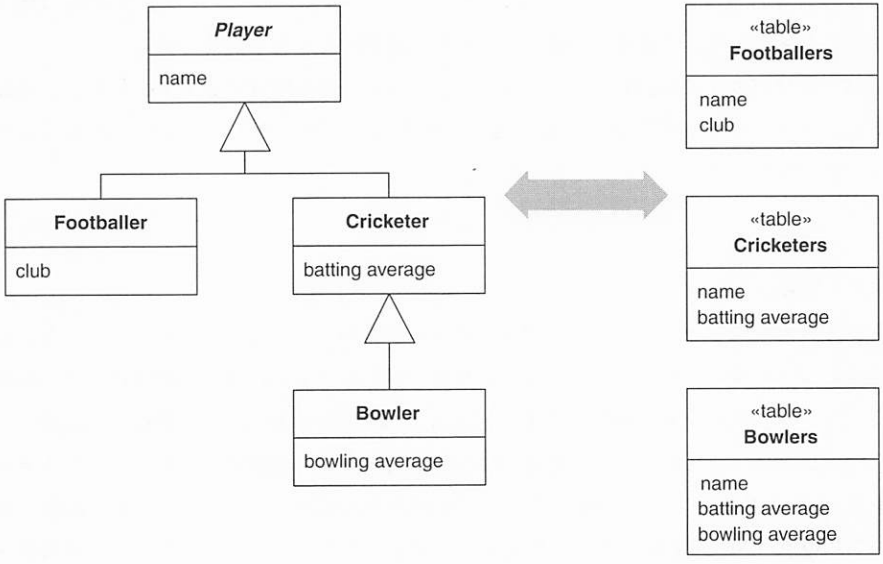


図 3.9 ——階層にある各具象クラスを格納するために 1 つのテーブルを使用する具象テーブル継承

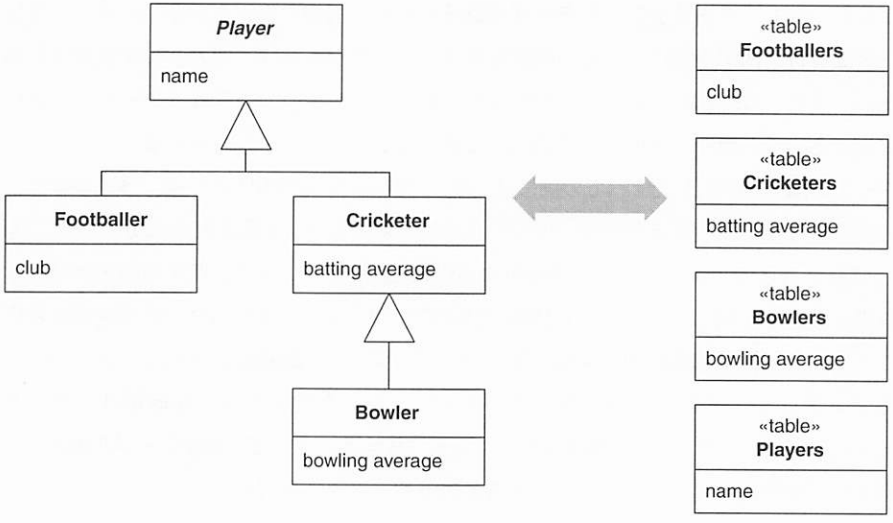


図 3.10 ——階層にあるクラスごとに 1 つのテーブルを使用するクラステーブル継承

すべてはデータ構造の重複とアクセス速度とのトレードオフである。**クラステーブル継承**は、クラスとテーブルとの最もシンプルな関係だが、1 つのオブジェクトをロードするために複数のジョインが必要になる。これにより、通常、パフォーマンスが低下する。**具象テーブル継承**は、このジョインを回避し、1 つのテーブルから 1 つのオブジェクトを抽出することを可能にするが、変更に対して脆弱である。スーパークラスへの変更に関しては、すべてのテーブル（とマッピングコード）を修正しなければいけない。階層自体を修正すれば、さらに大きな変更が発生するだろう。また、スーパークラスのテーブルがないので、スーパークラスのテーブルでのロック競合は減少するが、キーの管理が困難になり、参照整合性の妨げになる。**シングルテーブル継承**の最大の短所は、いくつかのデータベースでは無駄なスペースができることである。各行はすべての可能な部分型に対応する列を持つ必要があり、これにより空の列ができてしまう。しかし、多くのデータベースでは、無駄なテーブルのスペースを見事に圧縮できる。**シングルテーブル継承**はその大きさのために別の問題が発生し、アクセスする際の障害になる。しかし、大きなメリットは、すべての要素を同じところに配置できることである。これにより修正が容易になり、ジョインを回避することができる。

この 3 つの選択肢は、相互排他的ではなく、1 つの階層にパターンを混合できる。たとえば、**シングルテーブル継承**でいくつかのクラスをまとめることができ、いくつかの例外的な状況には**クラステーブル継承**を使用できる。当然、パターンを混合すると複雑性は増す。

この問題に関する明確な成功法はない。他のすべてのパターンと同様に、それぞれの状況と趣向を考慮する必要がある。私はこの場合、容易に行うことができ、多くのリファクタリ

ングに対して弾力性がある**シングルテーブル継承**を最初に選択することが多い。また、関係のない無駄な行に関する避けられない問題を解決する必要があれば、他の2つを使用する傾向がある。また、DBA に相談することが最善である場合が多い。彼らは、データベースに最も有用であるこの種のアクセス方法に関して適切なアドバイスをしてくれる。

パターンに関して説明したすべての例には、単一の継承だけを使用している。多重継承は、今日、流行ではなくなってきていて、多くの言語はこれを徐々に回避しているが、それでも Java や.NET などでインタフェースを使用する際に O/R マッピングに関する問題が発生する。本書のパターンはこのテーマを具体的に扱わないが、3つの継承パターンの変形を使用することで、多重継承に本質的に対処する。**シングルテーブル継承**は、すべてのスーパークラスとインタフェースを1つの大きなテーブルに入れ、**クラステーブル継承**は、インタフェースとスーパークラスごとに個別のテーブルを作成する。そして、**具象テーブル継承**は、各具象テーブル内にすべてのインタフェースとスーパークラスを含む。

3.5 マッピングの構築

リレーショナルデータベースへのマッピングを行う際、必然的に以下の3つの状況に直面する。

- スキーマを自分で選択する。
- 変更できない既存のスキーマへのマッピングを行わなければいけない。
- 既存のスキーマへのマッピングを行わなければいけないが、このための変更には交渉の余地がある。

最もシンプルなのは、自分でスキーマを作成し、ドメインロジックの複雑性を普通よりも少なくして、**トランザクションスクリプト**の設計または**テーブルモジュール**の設計を行えるようにすることである。この場合、従来のデータベース設計技法を用いたデータのテーブルを設計できる。**行データゲートウェイ**または**テーブルデータゲートウェイ**を使用して、ドメインロジックから SQL を分離する。

ドメインモデルを使用している場合は、データベース設計のような設計に注意する必要がある。この場合、ドメインロジックを適切に簡略化できるように、データベースを無視して**ドメインモデル**を構築する。データベース設計をオブジェクトのデータを永続化する方法として扱う。**データマッパー**は最も柔軟性を提供するが、複雑である。**ドメインモデル**と同じデータベース設計が意味を成す場合は、代わりに**アクティブレコード**を考慮するだろう。

最初にモデルを構築することは妥当な方法だが、これは短いイテレーティブなサイクル内

にだけ適用される。データベースを使用しない**ドメインモデル**の構築に6ヶ月を費やし、完了後に永続化を決定することは高いリスクを伴う。長い期間をかけた設計には、重大なパフォーマンスの問題があり、修正するためには膨大なリファクタリングが必要になる。そのかわり、6週間を越さない程度の期間で、データベースをイテレーションごとに構築する。これを行うことで、データベースの相互作用がどのくらい効果的であるかについての迅速で継続的なフィードバックを得ることができる。特定のタスク内で、最初に**ドメインモデル**について考えるべきだが、使用するデータベース内に**ドメインモデル**の各要素を統合する必要がある。

スキーマがすでにある場合、選択は似ているが、プロセスは少し異なる。シンプルなドメインロジックでは、データベースを持つ**行データゲートウェイ**または**テーブルデータゲートウェイ**のクラスを構築し、その上にドメインロジックをレイヤ化する。複雑なドメインロジックでは、データベース設計に適合する可能性がほとんどない**ドメインモデル**が必要である。このため、徐々に**ドメインモデル**を構築し、既存のデータベース内へデータを永続化するために**データマッパー**を含める。

3.5.1 ｜ 2重のマッピング

2つ以上のソースから同じようなデータが戻される状況を目にすることがある。コピーアンドペーストするので、同じデータを保持するが、少しだけスキーマの異なる複数のデータベースがあるためだろう（この状況の場合、苛立ちの度合いは相違の割合に反比例する）。他の可能性としては、異なるメカニズムを使用していることと、データベースの場合もあればメッセージの場合もあるデータを格納していることがあげられる。XMLメッセージ、CICSトランザクション、リレーショナルテーブルから同様のデータを抽出しようとしたこともあるだろう。

最もシンプルな選択肢は、複数のマッピングレイヤ（データソースごとに1つ）を持つことである。しかし、データが酷似している場合は、多くの重複を引き起こすことにつながる。この状況では、2ステップマッピング案を検討する。1ステップ目は、メモリ上のスキーマから論理的なデータソーススキーマにデータを変換する。論理的なデータ格納スキーマは、データソースフォーマット間で類似性を最大限に高めるように設計され、2ステップ目で相違を含める。

追加のステップは、多くの共通性がある場合にだけ有効なので、類似性はあるが異なる物理的なデータを格納する場合は追加のステップを使用すべきである。論理的なデータ格納から物理的なデータ格納へのマッピングを**ゲートウェイ**として処理し、マッピング技法を使用してアプリケーションロジックから論理的なデータ格納にマッピングを行う。

3.6 メタデータの使用

本書では、ほとんどの例に手書きのコードを使用する。シンプルで繰り返されるマッピングを使用すると、シンプルで繰り返されるコードを作ることにつながる。同じコードの繰り返しがあるということは、設計に間違いがあることを意味する。共通の振る舞いを継承や委譲で抜き出すことなど（有効で正当なオブジェクト指向のプラクティスを駆使して）多くのことを行えるが、**メタデータマッピング**を使用する洗練された手法も存在する。

メタデータマッピングは、データベースの列をオブジェクトのフィールドにマッピングする方法を詳述したメタファイルへマッピングを集約することに基づいている。メタデータがあれば、コードの生成か自己反映的なプログラミングのどちらかを使用して同じコードの繰り返しを回避することができる。

メタデータの使用によって、少ないメタデータから多くの表現力を得られる。1行のメタデータは以下のようになる。

```
<field name = customer targetClass = "Customer", dbColumn = "custID", targetTable =
"customers" lowerBound = "1" upperBound = "1" setter = "loadCustomer"/>
```

このメタデータからコードの読み込みと書き込みの定義、任意のジョインの自動生成、すべてのSQLの実行、関係の多重度の強制、参照整合性がある場合における書き込み順序の計算のような手の込んだ処理の実行を行える。これが、商用のO/Rマッピングツールがメタデータを使用することが多い理由である。

メタデータマッピングを使用する際は、メモリ上のオブジェクトを用いてクエリーを構築するために必要な基盤がある。**クエリーオブジェクト**でメモリ上のオブジェクトとデータに関するクエリーを構築することができ、開発者はSQLやリレーショナルスキーマの詳細を把握する必要はない。**クエリーオブジェクト**は、**メタデータマッピング**を使用して、適切なSQLへオブジェクトフィールドに基づく表現を変換できる。

ここまで行えば、データベースをビューから大幅に隠ぺいする**リポジトリ**を形成できる。データベースへのクエリーは、**リポジトリ**に対する**クエリーオブジェクト**として作成でき、開発者はオブジェクトがメモリから検索されたのか、データベースから検索されたのかはわからない。**リポジトリ**は、豊富な**ドメインモデル**システムとともに使用すると効率的である。

メタデータには多くのメリットがあるが、本書では、最初に理解しやすいので、手書きの例に焦点を当てる。パターンの使用法を理解し、アプリケーション用のパターンを手書きできるようになれば、メタデータを使用する方法を解釈し、処理を容易にすることもできるだろう。

3.7 データベース接続

多くのデータベースインタフェースは、アプリケーションコードとデータベースとのリンクとしての役割を果たすデータベース接続オブジェクトに依存する。一般的に、接続はデータベースに対してコマンドを実行する前にオープンにしなければいけない。実際、コマンドを作成し、実行するためには明示的な接続が必要な場合が多い。コマンドの実行時は常にこれと同じ接続をオープンする必要がある。クエリーは**レコードセット**を返す。いくつかのインタフェースは、切断された**レコードセット**を提供するが、これは接続を閉じた後に処理されるものだ。**レコードセット**が処理される間、接続をオープンにしたままにしなければいけない、接続された**レコードセット**だけを提供するインタフェースもある。トランザクション内で実行している場合は、トランザクションは必ず特定の接続で行い、実行中の接続はオープンにしたままでなければいけない。

多くの環境では、接続の作成にかかるコストは高いので、接続プールを作成する方がよい。この状況では、開発者は接続の作成と終了を行うのではなく、プールから接続を要求し、終了時に解放する。今日、多くのプラットフォームでプールを提供するので、自分で行わなければいけないことは稀である。自分で行わなければいけない場合は、最初に、プールが実際にパフォーマンスに役立つかどうかを確かめてほしい。新しい接続の生成速度が速くなっているなら、プールする必要はない。

プールを提供する環境では、新しい接続を作成するようなインタフェースの裏側にプールを置くことが多い。このため、最新の接続なのか、プールから割り当てられた接続なのかはわからない。しかし、プールの選択の可否は適切にカプセル化されているので、特に問題はない。同様に、接続の終了は、実際にクローズしたのではなく、他のユーザが使用できるようにプールに返しているだけかもしれない。本書では「オープン」と「クローズ」という言い方をするが、プールからの「取得」とプールへの「解放」と言い換えることができる。

作成のコストが高いかどうかに関わらず、接続には管理が必要である。管理するリソースは高価なので、使用が終了したらすぐにクローズしなければいけない。さらに、トランザクションを使用している場合は、一般的に、特定のトランザクション内にあるすべてのコマンドが同じ接続で実行されるようにする必要がある。

最も一般的な対処法は、プールまたは接続マネージャへの呼び出しを使用して、明示的に接続を取得し、行いたいデータベースコマンドごとにその接続を提供することである。接続を終了したらすぐにクローズしなければいけない。この方法では、2、3の問題が生じるので、必要であれば接続し、終了時には忘れずにクローズすべきだ。

必要なところで必ず接続されるようにするためには、2つの選択肢がある。1つ目は、接続を明示的な引数として渡すことである。これに関する問題は、呼び出しスタックの5レイヤ下にある他のメソッドに渡されることだけを目的として、あらゆる種類のメソッド呼び

出しに接続が追加されることである。当然、これは**レジストリ**を使用すべき状況である。同じ接続を複数のスレッドで使用する必要はないので、スレッドスコープの**レジストリ**が必要になる。

忘れっぽい人は、明示的にクローズすることは適さない。必要な場合に終了することをすぐ忘れてしまう。また、トランザクション内で実行している場合もあるので、コマンドごとに接続をクローズすることもできない。通常、クローズするとトランザクションのロールバックを引き起こす。

接続と同じように、メモリは未使用時に解放する必要があるリソースである。今日、最新の環境は、自動メモリ管理とガベージコレクタを提供するので、接続がクローズされたことを確実にする1つの方法は、ガベージコレクタを使用することである。この手法では、接続自体、または、その接続を参照するあるオブジェクトが、ガベージコレクション中に接続をクローズする。この優れた点は、メモリと同じ管理方式を使用することと、使いやすいことだ。問題は、ガベージコレクタが実際にメモリを再要求する場合にだけ接続がクローズすることであり、接続が最後の参照を失ってからかなり後にクローズが行われる場合がある。結果として、接続がクローズされる前にしばらくの間、未参照接続が残ることになる。これが問題かどうかは、それぞれの環境に依存する。

総合的に見ると、ガベージコレクションを使用することは推奨しない。明示的にクローズするなどの他の手段を使用する方が適切である。通常の手段が失敗した場合は、ガベージコレクションが優れたバックアップとなる。結局は、接続を永遠に残すよりも最終的に接続をクローズさせる方がよい。

接続はトランザクションと強く結び付いているので、優れた管理方法は、トランザクションと接続を関連付けることである。トランザクションの開始時に接続をオープンし、コミットまたはロールバック時に接続をクローズする。トランザクションがどの接続を使用しているかわかっているので、読者は接続を完全に無視し、トランザクションに対処できる。トランザクションの完了は目に見える効果なので忘れずにコミットし、仮に忘れたとしても見分けることが容易である。**ユニットオブワーク**によって、トランザクションと接続の両方を管理することに自然に適合できる。

不変データの読み込みなど、トランザクション以外の処理を行う場合は、コマンドごとに新しい接続を使用する。プールは、一時的な接続の作成に関する問題に対処する。

切断された**レコードセット**を使用している場合は、レコードセットにデータを加えるために接続をオープンし、**レコードセット**データを処理している間にクローズすることができる。そして、データの処理を終了すると、新しい接続やトランザクションをオープンしてデータを書き込むことができる。これを行う場合、**レコードセット**が処理されている間にデータが変更されているか注意する必要がある。これに関しては、並行性制御とともに取り上げる。

接続管理の詳細は、データベースの相互作用ソフトウェアの機能なので、使用するストラ

テジーは、環境ごとに決定される場合が多い。

3.8 その他の要点

あるコードでは、他のコードが列名を使用するのに対し、select * from という形式でselect 文を使用している。select * を使用すると、ある種のデータベースドライバでは、新しい列が追加されるか、列が並び換えられた場合にコードが壊れるという深刻な問題が発生する。これは、最新の環境では起こらないが、列の並び換えはコードを破壊するので、列から情報を取得するために位置インデックスを使用している場合は、select * を使用することは賢明ではない。列名インデックスを select * とともに使用することは問題ない。実際、列名インデックスは読みやすい。しかし、SQL 呼び出しに要する時間はほとんど同じでも、列名インデックスだと遅くなるだろう。通常どおり、測定が必要だ。

列番号インデックスを使用する場合は、列が並び換えられても同期が取れなくなることがないように、必ず結果群（result set）へのアクセスを SQL 文の近くで行うことが必要である。このため、**テーブルデータゲートウェイ**を使用している場合、ゲートウェイで検索操作を実行するコードごとに結果群が用いられるので、列名インデックスを使うべきである。結果として、通常、使用する構造をマッピングするデータベースごとにシンプルな、作成、読み出し、更新、削除（create, read, update, delete）テストケースを作成することが必要だ。これは、SQL とコードとの同期が取れなくなった際の状況の把握に役立つ。

毎回コンパイルしなければいけない動的な SQL ではなく、事前にコンパイルできる静的な SQL を使用するために労力を費やすのは常に価値がある。多くのプラットフォームには、SQL を事前にコンパイルするためのメカニズムがある。優れた経験的な方法は、SQL クエリーをまとめるために、文字列の連結を使用しないようにすることである。

多くの環境には、複数の SQL クエリーを 1 つのデータベースバッチ呼び出しにまとめる機能がある。例には使用していないが、本番稼動するコードで使用すべき確実な戦術である。実行方法は、プラットフォームによって異なる。

これらの例の接続に関しては、**レジストリ**である「DB」オブジェクトへの呼び出しだけを想定している。接続を得る方法は環境に依存するので、環境に合わせて、行う必要がある処理を置き換える。並行性に関する以外のパターンにはトランザクションを結び付けていない。再度説明するが、それぞれの環境に必要な処理を投入する必要がある。

3.9 参考文献

オブジェクトリレーショナルマッピングは多くの人にとって避けられない話題なので、この題材に関する文書は多くある。しかし意外なことに、一貫した、完全な、最新の書籍がない。この理由から、本書の多くをこの扱い難く興味深い題材に割り当てた。

データベースのマッピングに関する素晴らしい点は、拝借して利用できる多くの考え方があることである。最も有益なライブラリは、[Brown and Whitenack]、[Ambler]、[Yoder]、[Keller and Coldewey]である。本書のパターンを補足するために、ぜひこれらの内容を参照していただきたい。

第4章

Webプレゼンテーション

ここ数年間のエンタープライズアプリケーションにおける最大の変化の1つは、Webブラウザーベースのユーザインタフェースの発展である。Webブラウザーベースのユーザインタフェースには、クライアントソフトウェアをインストールする必要がない、共通のUI手法である、世界中へのアクセスが容易に行えるなど、多くのメリットがある。

Webアプリケーションの作成は、サーバソフトウェア自体から開始する。通常、Webアプリケーションを作成する場合は、どのURLがどのプログラムで処理されるかを示すような構成ファイルがある。多くの場合、1つのWebサーバは、さまざまなプログラムに対処することができる。これらのプログラムは、動的であり、適切なディレクトリに配置することでサーバに追加することができる。Webサーバは、リクエストのURLを解釈し、制御をWebサーバのプログラムに渡すことを行う。Webサーバにおけるプログラムの構造化には、スクリプトとしての構造化と、サーバページとしての構造化という2つの主な方法がある。

スクリプト形式はプログラムであり、通常、HTTP呼び出しを処理するための機能または方法がある。例として、CGIスクリプトやJavaサーブレットがある。プログラムテキストはプログラムが行えるほとんどのことを行うことができ、スクリプトはサブルーチンに分割され、他のサービスを作成および使用することができる。また、文字列であるHTTPリクエストオブジェクトを調査することでWebページからデータを取得する。環境によっては、リクエスト文字列の正規表現を検索してデータの取得を行う場合もある（Perlを使用するとこれを容易に行えるので、CGIスクリプトにおいて一般的な選択肢になっている）。Javaサーブレットなどの他のプラットフォームは、この解析をプログラマのために行うので、プログラマはキーワードインタフェースを用いてリクエストからの情報にアクセスできる。これは、扱うべき正規表現が少ないことを意味する。Webサーバの出力は、他の文字列、つまりレスポンスであり、スクリプトはこの文字列に対して言語における一般的なストリーム操作を使用して書き込みを行える。

ストリームコマンドによるHTMLレスポンスの作成は、プログラマには不自然である。

また、別の方法でHTMLページを作成することに満足しているプログラマ以外の人にはほぼ不可能である。これは、テキストページを返すことを中心にプログラムが構造化されるサーバページの考え方につながる。レスポンスとして返すページをHTMLとして記述し、その中にスクリプトを挿入して、ある時点で実行する。この手法の例としては、PHP、ASP、JSPなどがある。

サーバページ手法は、「アルバム#1234の詳細を示せ」など、レスポンスの処理が少ない場合に有効である。クラシックとジャズでアルバムの表示形式が異なる場合などのように、入力に基づいた決定を行わなければいけない場合は、混乱が生じる。

スクリプトはリクエストの解釈に最も効果があり、サーバページはレスポンスの書式化に最も効果があるので、リクエストの解釈のためにスクリプトを使用し、レスポンスの書式化にサーバページを使用するというように選択が分かれている。実際には、この区別は、**モデルビューコントローラ**パターンを使用するユーザインタフェースで最初に表面化する古い考え方である。この考え方と、プレゼンテーション以外のロジックが抜き出される必要があるという不可欠な概念を組み合わせることで、**モデルビューコントローラ**パターンの概念に適合する。

モデルビューコントローラは、広く参照されるパターンだが、誤解されがちである。実際、Webアプリケーションが登場する以前は、**モデルビューコントローラ**の多くのプレゼンテーションは適切なものではなかった。この混乱の主な理由は、「コントローラ」という言葉の使用であった。コントローラは、多くの異なるコンテキストで使用される。また普通は、**モデルビューコントローラ**で使用されるのとは異なる意味で使用されるようである。そのため私は、**モデルビューコントローラ**のコントローラには「入力コントローラ」という用語を使用することを好んでいる。

リクエストは、リクエストから情報を抜き取る入力コントローラに送信されてくる。そして、ビジネスロジックを適切なモデルオブジェクトに送る。モデルオブジェクトは、データソースと通信し、レスポンス用の情報の収集だけでなく、リクエストによって指示されたすべてを行う。終了すると、入力コントローラに制御を返す。入力コントローラは、結果を確認し、レスポンスを表示するために必要なビューを決定する。そして、レスポンスデータとともに制御をビューに渡す。入力コントローラをビューへ渡すことは、常に直線的な呼び出しであるとは限らない。むしろ、入力コントローラとビューに共有されるある形式のHTTPセッションオブジェクトに、決まった場所に配置されたデータとともに転送されることが多い。

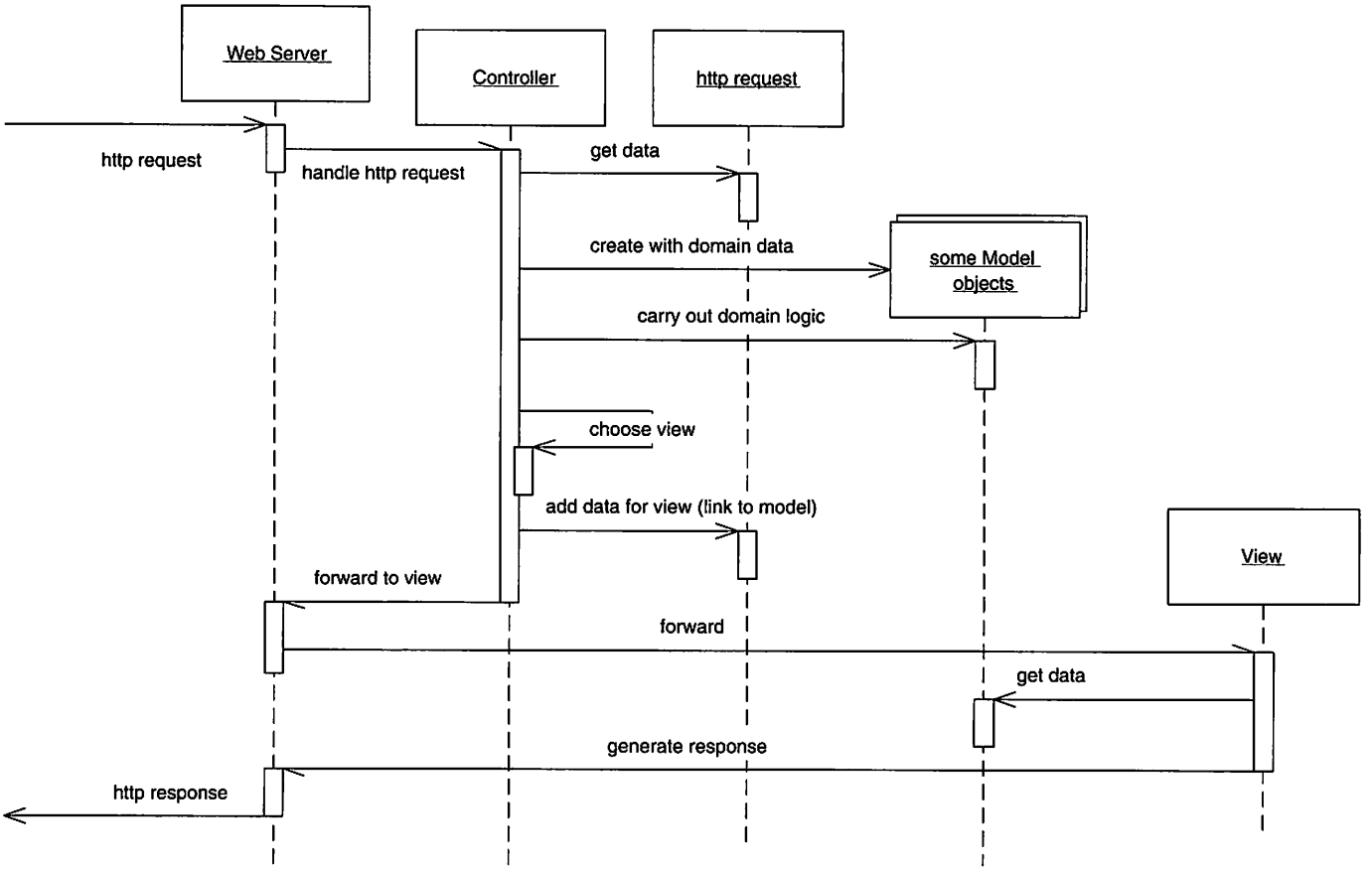


図 4.1 ——モデル、ビュー、入力コントローラの役割が Web サーバで連携する方法の概略図。コントローラはリクエストを処理し、ドメインロジックをモデルに実行させ、モデルに基づくレスポンスをビューに作成させる。

モデルビューコントローラを適用する第1の、そして最も重要な理由は、モデルをWebプレゼンテーションから完全に分離することが確実にできるからである。モデルをWebプレゼンテーションから分離することで、別のプレゼンテーションを後から追加することが容易になるだけでなく、プレゼンテーションを修正することも容易になる。処理を別々の**トランザクションスクリプト**や**ドメインモデル**オブジェクトに置くと、テストを行うことも容易になる。これは、ビューとしてサーバページを使用している場合、特に重要である。

ここで、「コントローラ」という言葉の2つ目の使い方が出てくる。多くのユーザインタフェースの設計は、**アプリケーションコントローラ**オブジェクトの中間レイヤによってプレゼンテーションオブジェクトとドメインオブジェクトを分離する。**アプリケーションコントローラ**の目的はアプリケーションのフローを処理することであり、どの順番でどの画面を表示するかを決定する。プレゼンテーションレイヤの一部として表示することもあれば、プレゼンテーションレイヤとドメインレイヤとを仲介する分離したレイヤとして考えることもできる。**アプリケーションコントローラ**は、特定のプレゼンテーションに関係なく書き込まれる。この場合、**アプリケーションコントローラ**は、プレゼンテーション間で再度使用できる。異なるプレゼンテーションに異なるフローを持たせることが最善である場合が多いが、同じ基本フローと遷移を用いた異なるプレゼンテーションがある場合にこれは有効である。

すべてのシステムに**アプリケーションコントローラ**が必要なわけではない。システムに画面の順序とその間の遷移に関する多くのロジックがある場合、**アプリケーションコントローラ**は有用である。また、ドメイン上でページとアクションがシンプルにマッピングしていない場合にも役立つ。しかし、任意の順序で画面を見ることが多い場合は、おそらく**アプリケーションコントローラ**の必要性はほとんどない。判断基準としては、画面フローをマシンが管理しているのであれば**アプリケーションコントローラ**が必要で、ユーザが管理しているのであれば必要ないと考えるべきである。

4.1 ビューに関するパターン

ビューに関しては、**トランスフォームビュー**、**テンプレートビュー**、**ツーステップビュー**という3つのパターンが考えられる。この3つのパターンにより2種類の選択が生じる。1つは、**トランスフォームビュー**と**テンプレートビュー**のどちらを使用するかという選択、もう1つは、これらのパターンのいずれかが、シングルステージビューと**ツーステップビュー**のどちらを使用するかという選択である。**トランスフォームビュー**と**テンプレートビュー**の基本パターンはシングルステージビューである。**ツーステップビュー**は、どちらにも適用できるバリエーションである。

テンプレートビューと**トランスフォームビュー**との選択から始めよう。**テンプレートビュー**

を使用すると、ページの構造にプレゼンテーションを書き込むことができ、ページにマークを付けて動的な内容が必要となるところを示すことができる。人気の高いプラットフォームの多くがこのパターンに基づいている。これらのプラットフォームは、完全なプログラミング言語をページに入れることができるサーバページ技術（ASP、JSP、PHP）であることが多い。このサーバページ技術は、多くの能力や柔軟性を提供する。残念なことに、これは維持することが困難なとても複雑なコードにもつながる。結果として、サーバページ技術を使用するのであれば、多くの場合ヘルパーオブジェクトを使用して、ページ構造から切り離してプログラミングロジックを保持することを守らなければいけない。

トランスフォームビューはプログラムの変換方式を使用する。通常の例は、XSLT である。これは、XML フォーマット、または容易にそれに変換できるドメインデータを扱っているのであれば、とても有効である。入力コントローラは、適切な XSLT スタイルシートを選別し、モデルから収集された XML に適用する。

ビューとして手続き型スクリプトを使用しているのであれば、**トランスフォームビュー**か**テンプレートビュー**のどちらか一方の方式、またはこの２つの興味深い組み合わせでコードを作成することができる。私は、ほとんどのスクリプトが主な形式としてこの２つのパターンのいずれかに従っていることに気付いた。

２つ目の選択は、シングルステージビュー（図 4.2 参照）にするか、**ツーステップビュー**を使用するかということである。ほとんどの場合、シングルステージビューには、アプリケーションの画面ごとに１つのビューコンポーネントがある。ビューは、ドメイン指向データを受け取り、そのデータを HTML に変換する。「ほとんどの場合」と説明した理由は、同じような論理的な画面がビューを共有する場合があるからである。そうであったとしても、画面ごとに１つのビューとして考えることがほとんどである。

ツーステップビュー（図 4.3 参照）は、このプロセスを２つの段階に分割し、ドメインデータから論理的な画面を作成してから、HTML に変換する。画面ごとに１つの第１段階ビューがあるが、全体のアプリケーションの第２段階ビューは１つだけしかない。

ツーステップビューのメリットは、１箇所で使用する HTML を決定することである。サイトのすべての画面を修正するために変更するオブジェクトは１つだけなので、HTML への広範囲の変更が容易になる。当然、論理的なプレゼンテーションが同じ状態の場合にだけそのメリットを得ることができるので、異なる画面が同じ基本レイアウトを使用するサイトを用いることが最も効果的である。デザインに凝っているサイトは、優れた論理的な画面構造を作り出すことができないだろう。

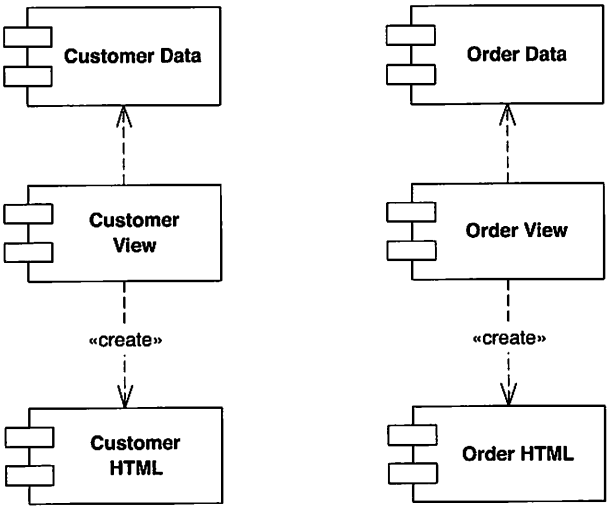


図 4.2——シングルステージビュー

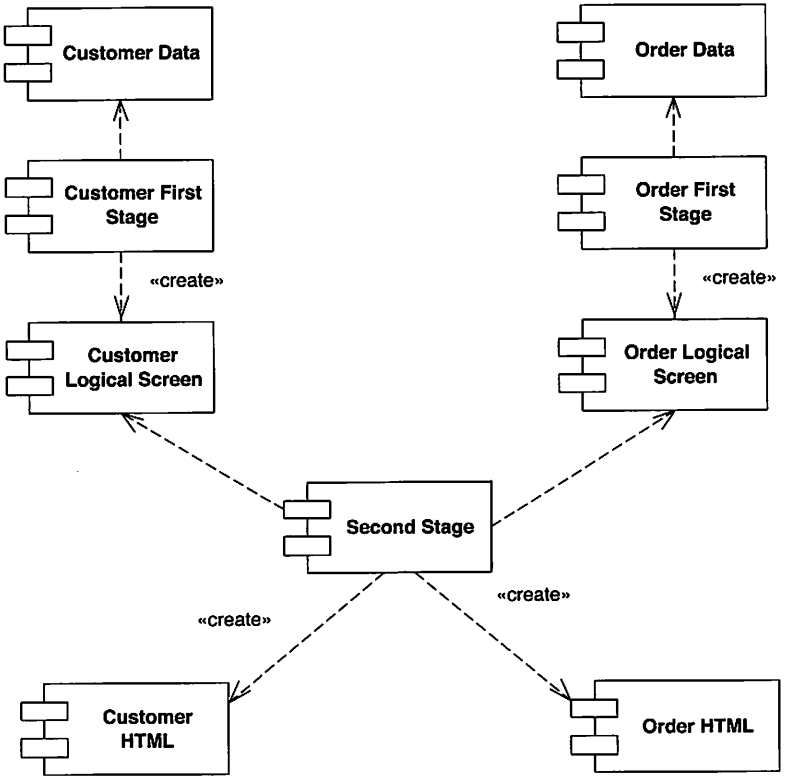


図 4.3——ツーステップビュー

ツーステップビューは、同じ基本的な予約システムを使用する複数の航空会社など、複数のフロントエンドの顧客にサービスが使用されているWebアプリケーションを使用する場合に効率的である。論理的な画面の限度内で、異なる第2段階を使用することによって、フロントエンドごとにさまざまな外見にすることができる。同様の方法で、**ツーステップビュー**を使用し、通常のWebブラウザーと、パームトップに対し別々の第2段階を用いて異なる出力デバイスに対処できる。再度説明するが、同じ論理的な画面はUIが共通の2つで使用することが限度であり、ブラウザーと携帯電話などのようにUIがまったく異なる場合は不可能だろう。

4.2 入力コントローラに関するパターン

入力コントローラには2つのパターンがある。Webサイトのページごとの入力コントローラオブジェクトが最も一般的である。最もシンプルな場合、この**ページコントローラ**は、サーバページ自体になり、ビューと入力コントローラの役割を併せ持つ。多くの実装では、入力コントローラを別々のオブジェクトに分離すると、処理が容易になる。入力コントローラは、適切なモデルを生成して処理を行い、結果を返すためにビューのインスタンスを作成して結果を返すことができる。多くの場合、**ページコントローラ**とビューとは完全に一対一の関係ではない。ボタンやリンクであったりする1つのアクションごとに**ページコントローラ**があるというのが正しい考え方である。アクションはページに対応することが多いが、状況に応じて2、3の異なるページに行くリンクなどの場合は、ページに対応しないことがある。

入力コントローラには、HTTPリクエストの処理と行うべきことの決定という2つの責任があり、この責任を分離することは意味がある場合が多い。サーバページはリクエストを処理することができ、別々のヘルパーオブジェクトを委譲して、行うことを決定する。**フロントコントローラ**は、すべてのリクエストを処理する1つのオブジェクトだけを持つことでこの分離を推進する。この1つのオブジェクトは、URLを解釈して、処理するリクエストの種類を考え、別々のオブジェクトを作成してリクエストを処理する。この方法は、1つのオブジェクト内で処理するすべてのHTTPに焦点を当て、サイトのアクション構造を変更する際は、常にWebサーバを再構成する必要を回避できる。

4.3 参考文献

Webサーバ技術に関する書籍のほとんどは、優れたサーバ設計の解説に1、2章を割いている。とは言っても、技術的な説明に埋没しがちなものが多い。[Brown et al.]の第9章には、Java Webデザインに関する素晴らしい解説がある。より多くのパターンに関する最良の情報源は[Alur et al.]である。これらのパターンの多くは、Java以外で使用することができる。私は、入力コントローラとアプリケーションコントローラを切り離すことに関する用語を[Knight and Dai]から使わせていただいた。

第5章

並行性

(by Martin Fowler, David Rice)

並行性は、ソフトウェア開発において最も扱いにくい部分である。複数のプロセスやスレッドが同じデータを処理するときは常に並行性が問題になる。並行性を説明することは簡単ではない。トラブルを引き起こす場合を考えるのが難しいからだ。何かを見落とすことはどんな場合にも起こりうるだろう。それに、並行性はテストするのが容易ではない。ソフトウェア開発の基礎段階でさまざまな自動化テストを行うのは簡単である。しかし、並行性から生じるトラブルを防ぐためのテストを実行するのはとても難しい。

皮肉なことに、エンタープライズアプリケーション開発において、並行性を使用することがなく、そのことを問題にしていないソフトウェア開発部門もある。エンタープライズ系の開発者が並行性をあまり気にしない理由は、トランザクション管理にある。トランザクションが提供するフレームワークは、エンタープライズアプリケーションのなかで最も扱いにくい並行性の側面の多くを吸収している。データ処理をトランザクション内で行っている限り、特に問題は発生しないのである。

しかし残念ながら、トランザクションによって並行性の問題を完全に無視できるわけではない。それは、システムとの相互作用の多くは1つのデータベーストランザクション内に配置することはできないからだ。つまり、データが複数のトランザクションにまたがる場合は、並行性の管理が必要である。このように複数のデータベーストランザクション間で処理されるデータの並行性を制御することを、私は**オフライン並行性**と呼んでいる。

エンタープライズ系開発者にとって並行性が問題となるもう一つの分野は、アプリケーションサーバである。アプリケーションサーバは、1つのアプリケーションサーバシステム内で複数のスレッドをサポートしている。これは比較的単純なケースなので、本書では多く説明しない。適切なサーバプラットフォームを使用することで対応できる問題である。

これらの問題を理解するには、並行性についての一般的な概念を理解しておく必要がある。そこで本章では、まずこれらの問題について見ていくことにしよう。ただし本章では、ソフ

トウェア開発における並行性の処理については説明しない。それだけで本が一冊書けてしまうからだ。本章では、あくまでもエンタープライズアプリケーション開発における並行性についてだけ採り上げる。その後で、オフライン並行性を処理するパターンを紹介し、アプリケーションサーバの並行性についても簡単に解説したい。

本章では概念を説明するために、主に読者に馴染みのあると思われる分野の例を使っている。つまり、ソースコードに対する変更を調整するために開発チームが用いるソースコード管理システムの例を使用して説明する。この例を使用するのは、単に馴染みがあるというだけでなく、比較的理解しやすいからだ。エンタープライズアプリケーションの開発をするためには、ソースコード管理システムに関する知識が必要である。

5.1 並行性の問題

まず、並行性の本質的な問題について説明しよう。なぜ本質的かというと、それらは並行性制御システムが避けようとしている基本的な問題だからだ。しかし、これらだけが並行性の問題のすべてではない。制御メカニズムが、トラブルを解決しようとしてかえって新たなトラブルを生じさせることも多いからだ。ただ、これら本質的な問題にこそ並行性制御の重要なポイントがある。

最もわかりやすい例が**更新結果が失われること**だろう。たとえば、Martin がファイルを編集して `checkConcurrency` メソッドを変更するとしよう。この作業は数分で終えることができる。しかし同時に David も同じファイル内の `updateImportantParameter` メソッドを編集しているとしよう。David は Martin よりも後に作業を始めたのだが、Martin よりも作業が早く終わったとする。ここで不幸が生じる。Martin がファイルを開いた時点では、David の更新は反映されていない。そのため、Martin がファイルの編集を終えたとき、その前に David が更新したファイルに上書きすることになり、David による更新は無に帰してしまう。

一貫性のない読み込みという例もある。読み込んだ2つの情報は間違っていないが、正しいものではないという場合である。たとえば、Martin が、locking と multiphase の2つのサブパッケージを含む並行性パッケージの中にクラスがいくつあるか調べているとする。locking パッケージを見ると、そこには7つのクラスがある。このとき Roy から電話がかかってきて、難しい質問を聞かれたとしよう。Martin が電話に答えている間に、David が4フェーズロックコード内のバグを修正し、7つのクラスがあった locking パッケージに2つ、5つのクラスがあった multiphase パッケージに3つのクラスを追加する。Martin が電話を終えて multiphase パッケージを見ると、そこには8つのクラスがある。したがって Martin の計算ではクラスの数は 7+8 の 15 になる。

これは正しい数ではない。Davidが修正を加える前のクラスの数は7+5の12で、修正後の数は9+8の17である。どちらもその時点では正しい数である。しかし、15という数はどの時点でも正しい数ではない。Martinは一貫していないデータを読み込んだわけだ。このような問題のことを「一貫性のない読み込み」と呼ぶ。

これらの問題は**正確さ**（または安全性）を脅かし、2人が同時に同じデータで作業しなければ起こらなかったかもしれない不正確な振る舞いを引き起こす。ただし、正確さだけが重要であれば、それほど深刻な問題ではない。同じデータで2人が同時に作業できないようにするだけで解決する。しかし、それで正確さを確保できるとしても、並行して何かを行うことはできない。並行性プログラミングの重要な条件として、正確さだけでは十分でなく、**即応性**も必要である。アクティビティをいくつ並行して処理できるかということも大事だ。エラーの重要度や頻度、データの並行処理の必要性にもよるが、即応性を重視して正確さを犠牲にすることもある。

これらは並行性に関する問題のすべてではないが、基本的なものだ。これらを解決するために、私たちはいろいろな制御メカニズムを使用する。しかし残念ながら、どんな問題にも対応できる策は存在しない。基本的な問題に比べればそれほど深刻ではないが、解決策から新たな問題が生じることもある。そこで、ここから重要なポイントが浮かび上がる。問題が許容できるのであれば、どんな形式の並行性制御もいらないということだ。とても稀なケースだが、そのような場合がないわけではない。

5.2 実行コンテキスト

システム内で処理が行われるときは、常に何らかのコンテキスト内、それも複数のコンテキスト内で行われる。実行コンテキストに関しては標準的な用語がないので、ここでいくつか定義をしておこう。

外部との相互作用という観点から見れば、リクエストとセッションが2つの重要なコンテキストである。**リクエスト**とは、ソフトウェアが動作する世界の外側からの呼び出しのことであり、ソフトウェアはこれに対してレスポンスを返す。リクエストの処理の大半はサーバ内で行われ、クライアントはレスポンスが返ってくるのを待つ。プロトコルによっては、レスポンスがくる前にクライアントがリクエストを中断できるものもあるが、これは稀なケースである。クライアントが別のリクエストを発行し、その前に送信したリクエストに割り込む場合の方が多い。たとえば、クライアントがOrder（注文）した後に、そのOrderをキャンセルする別のOrderを送るといった場合である。クライアント側から見れば2つのリクエストは明らかに関連しているが、プロトコルによっては、サーバはその関連性を意識していない。

セッションとは、クライアントとサーバが一定時間内に相互作用することである。セッションは1つのリクエストだけで構成されることもあるが、ユーザからは一貫した論理シーケンスに見える一連のリクエストから構成されていることが多い。一般に、セッションはユーザのログインで始まり、クエリーやビジネストランザクションの発行などといったさまざまな処理を行う。セッションの最後にユーザはログアウトする。または、ユーザが別の処理を始めると、システムはそれをログアウトしたと解釈する。

エンタープライズアプリケーションで使用されるサーバソフトウェアは、リクエストとセッションの両方を、クライアントからサーバへという角度と、別のシステムのクライアントとなる角度から見る。そのため、クライアントからのHTTPセッションと各種データベースのセッションなど、複数のセッションが同時に進行することになる。

OSから見た重要な2つの用語がプロセスとスレッドである。**プロセス**は重量級の実行コンテキストであり、プロセスで扱う内部データを分離する。**スレッド**は軽量級のアクティブエージェントであり、1つのプロセス内で複数のスレッドが動作することができる。1つのプロセス内で複数のリクエストに対応でき、リソースを効率的に使用できるスレッドの方が好まれている。しかし、スレッドはメモリを共有するのが一般的で、これが並行性の問題を引き起こす。しかし、環境によっては、メモリを共有しない**分離されたスレッド**を使用でき、スレッドがどのデータにアクセスするかを制御できる場合もある。

実行時の難点は、望ましい順序にそれらが並ばないことである。理論的には、各セッションはその存続期間中、プロセスとは排他的な関係にある。プロセスはそれぞれが適切に分離されるため、並行性におけるコンフリクトは少なくなるはずである。しかし、今のところそのような用途に使えるサーバツールはない。このようなツールに近いものとしては、初期のPerl Webシステムでよく使われていた、リクエストごとに新しいプロセスを開始するというモードがあった。しかし、プロセスを開始するには多くのリソースが必要なため、最近ではあまり使われなくなり、1つのプロセスで1つのリクエストだけを処理するシステムが一般的である。おかげで並行性についての悩みが軽減されたことになる。

データベースの場合は、**トランザクション**という重要なコンテキストがある。トランザクションは、クライアントからの複数のリクエストをあたかも1つのリクエストであるかのように扱う。トランザクションは、アプリケーションからデータベース（システムトランザクション）、またはユーザからアプリケーション（ビジネストランザクション）の間で行われる。これらについては、後で詳しく説明する。

5.3 分離性および不変性

並行性の問題は今に始まったことではなく、ソフトウェアに携わる人たちがこれまでさま

ざまな解決策を考えてきた。エンタープライズアプリケーションの場合は、分離性と不変性という2つの問題をいかに解決するかが特に重要である。

並行性の問題は、プロセスやスレッドなど複数のアクティブエージェントが同じデータに同時にアクセスしたときに発生する。その1つの対処法が分離で、データを区切ってそこに1つのアクティブエージェントしかアクセスできないようにする方法である。OSのメモリ内ではプロセスは、次のような方法で処理されている。つまり、OSは1つのプロセスに対しメモリを排他的に割り当て、そのプロセスがリンクしているデータの読み書きを実行できるようにしている。このような例としては、現在普及している多くのアプリケーション製品でもファイルがロックされるのを読者は経験したことがあるだろう。たとえば、Martinがあるファイルを開いているときは、そのファイルは他の人は開けられなくなる。開けた場合でも、Martinが作業を始める前のファイルのコピーを読み取り専用で開くだけで変更することはできず、Martinが加えている変更もそのファイルでは見ることができないのと同じである。

分離は、エラーを起こりにくくするという意味でもとても重要な技法である。常に並行性に注意していなければならない技法を使用したために、問題が起こっている場合がとても多い。分離を行えば、プログラムが分離された領域内に挿入される。この領域内では並行性について懸念する必要がない。つまり、優れた並行性設計とは、このような領域を作成する方法を模索して、できる限り多くのプログラミングをいずれかの領域で行えるようにすることである。

並行性の問題は、共有しているデータが修正できる場合にしか発生しない。つまり、**不変データ**を認知することが、並行性によるコンフリクトを回避する方法の1つとなる。だが、多くのシステムではデータの修正を主な目的としているため、すべてのデータを不変なものにすることはなかなかできない。そこで、不変データ、または不変であると認知したデータを特定することで、そのデータの並行性を考慮しなくても安心して共有できるようになる。別の方法としては、データを読み込むだけのアプリケーションを分離して、そのアプリケーションにデータソースのコピーを使用させることで、並行性の制御の懸念をなくすこともできる。

5.4 軽い並行性制御および重い並行性制御

では、変更できても分離できないデータの場合はどうなるのだろうか。並行性制御には大きく分けて軽い制御と重い制御の2つの形態がある。

MartinとDavidの両者がCustomerというファイルを同時に編集しようとした場合を想定しよう。**軽ロック**では両者ともファイルのコピーを作成し、自由に編集することができ

るので、Davidが先に編集を終えた場合、彼は自分の作業を問題なくチェックインできる。並行性制御が機能し始めるのは、Martinが変更をコミットしようとしたときだ。この時点で、ソースコード管理システムが、Martinが加えた変更とDavidの変更とのコンフリクトを検出して、Martinによるコミットは拒否され、その状況の処理はMartinの判断に任せられる。一方の**重ロック**では、最初にファイルをチェックアウトした人以外は編集が行えなくなる。つまり、先にMartinがファイルをチェックアウトした場合、Davidは、Martinが修正を終えて変更をコミットするまで、そのファイルでの作業はできないことになる。

簡単に表現すると、軽ロックではコンフリクトが検出され、重ロックではコンフリクトが発生しないようになる。実際のソースコード管理システムでは両方のタイプを使うことができるが、現時点では、ほとんどのソースコード開発者は、軽ロックの方を好んで使用している（「軽ロックは実際にはロックではない」という合理的な意見も多いようだが、用語として便利なうえ広く受け入れられているため、ここではそれらの意見は考慮の対象外とさせていただく）。

この2つの手法にはそれぞれ長所と短所がある。重ロックの場合は、並行性が低下する。たとえば、Martinが作業している間ファイルはロックされるため、他の人は彼が終了するまで待たなくてはならない。重いソースコード管理メカニズムを使った環境で作業をしたことがあれば、これがいかに不満に感じられるかがわかるだろう。だが企業データの場合は、この方が適していることが多いのだ。誰かがデータを編集している間は、他の人は見ることができないようにした方が混乱を生じさせないからだ。

軽ロックでは、ロックが実際に機能するのはコミットする段階だけなので、ストレスを感じることなく作業を進めることができるだろう。問題は、コンフリクトが生じた場合の対処の仕方である。基本的に、Davidのコミット後にコミットしようとしたすべての人は、Davidがチェックインしたバージョンのファイルをチェックアウトし、自分が加えた変更をDavidが加えた変更と一緒にしてから、新しいバージョンにチェックインする必要がある。ソースコードではこの方法は、それほど難しくはない。多くの場合、ソースコード管理システムが自動的に結合を行う。自動的な結合を行わない場合でも、ツールを使えば簡単に違いを見つけることができる。しかしビジネスデータの場合は、簡単に自動結合できるわけではないため、たいていは変更をすべて破棄して最初からやり直すことになる。

軽い手法と重い手法のどちらの並行性制御でロックを選択するかは、コンフリクトの頻度と重要性に左右される。コンフリクトの頻度がとても低い場合や、たとえ発生した場合でもその影響が大きくないのであれば、軽ロックを使用すべきである。より高い並行性を維持でき、一般的に実装も簡単だからである。一方で、コンフリクトがユーザに与える影響が大きい場合は、軽い手法ではなく重い手法を使用する必要がある。

しかし、いずれの手法も万全と言えるものではない。これらを使用することで生じる問題は、最初に解決しようとしていた並行性の基本的な問題と同じくらい、トラブルの原因にな

りかねない。詳細については並行性に関する専門書に委ねることにするが、以下にいくつかの留意すべき点を挙げておこう。

5.4.1 ｜一貫性のない読み込みの防止

次のような状況を想定してみよう。Martin が Order クラスを呼び出す Customer クラスを編集し、同時に David が Order クラスを編集し、インタフェースを変更する。そして、先に David がコンパイルしてチェックインし、続いて Martin がコンパイルしてチェックインする。Martin が気付かない間に Order クラスが変更されているので、すでにこの時点で共有コードは壊れている。ソースコード管理システムの中には、このような一貫性のない読み込みを察知できるものもあるが、そうでない場合は、チェックインする前にファイルを更新しておくなどの方法で、手動で一貫性を実現させる必要がある。

本質的には、これは一貫性のない読み込みの問題であるが、更新を喪失してしまうことが並行性の問題点であると考える人が多いため、見落とされる場合が多い。重ロックは、読み書きのロックという従来からの方法でこの問題に対処する。データを読み込むには読み込み（または共有）ロックが、書き込むには書き込み（または排他的）ロックが必要になる。同時に複数のユーザが読み込みロックを行うことができるが、誰かが読み込みロックを実行した時点で他の人は書き込みロックを行えなくなる。逆に誰かが書き込みロックを行った時点で、他の人はいずれのロックも行えなくなる。このようにこのシステムでは、重ロックによって一貫性のない読み込みが回避されている。

軽ロックでは、データのバージョンを示す何らかのマークによってコンフリクトを検出する。ここで言うマークとは、タイムスタンプまたはカウンタなどである。喪失した更新を検出するため、システムは更新されたデータのバージョンマークと共有データのバージョンマークをチェックし、もしそれらが同一だった場合は、更新が許可されバージョンマークも更新される。

一貫性のない読み込みの検出も基本的には同様の動作を行うが、この場合、読み込まれたデータすべてのビットのバージョンマークが共有データのバージョンマークと比較され、少しでも違うとコンフリクトが発生していると判断される。

読み込まれるデータすべてのビットへのアクセスを制御すると、コンフリクトやあまり重要でないデータの待機などといった問題が起こる場合が多い。このような不要な負荷は、使用したデータとほとんど使用しないデータを分離することで軽減させることができる。製品の選択リストの場合は、データの変更を開始した後に新しい製品がリストに加えられてもあまり影響はないが、たとえば請求書を書こうとして料金一覧をまとめているときに、この一覧に変更が加えられた場合は重大である。この判断が難しいところは、用途を注意深く分析しなくてはならないという点だ。また、顧客の住所の郵便番号は特に重要ではないかもしれ

ないが、税金の計算を割り出す時など所在地に基づいて行うときには、住所の並行性は制御しなければならない。このように何を制御するべきかを判断するのは、どの並行性制御を採用するにしても手間を必要とするのである。

一貫性のない読み込みへの対処法として**一時的読み込み**を使用する方法もある。この手法では、読み込まれた各データにタイムスタンプか不変ラベルを付け、データベースは、そのタイムスタンプの日時かラベルを基準にデータを返す。データベースシステムでこの方法を使うことは稀だが、ソースコード管理システムにはしばしば使われる。この手法の問題は、データソースが完全な変更の一時履歴を提供しなくてはならないため、処理に時間とスペースが必要になるという点にある。ソースコードでは問題ではないかもしれないが、データベースではより複雑で負荷の高いものとなってしまう。ドメインロジック内の特定の領域では、この機能を必要とする場合もある。実際の手順については、[Snodgrass]と[Fowler TP]を参照してほしい。

5.4.2 | デッドロック

重い技法特有の問題に**デッドロック**というものがある。たとえば、Martin が Customer ファイルを、David が Order ファイルをそれぞれ編集し始めたとしよう。途中で、David は自分の作業を完了するには Customer ファイルも編集しなくてはならないことに気付くが、Martin がロックしているため待たなければならない。その後、Martin も Order ファイルを編集する必要があることに気付くが、その時点では David がロックしている。この状態がデッドロックだ。一方が終了しない限り両方とも先に進めなくなった状態をいう。この例だけを見れば、デッドロックは簡単に回避できるように見えるかもしれないが、さらに大勢の人が関わる場合には、かなり扱いにくい状態になるのは明白である。

デッドロックへの対処方法としてはいくつかの技法が考えられる。1つにはデッドロックを検出するソフトウェアを使用する方法である。この場合「犠牲」になる人を選択し、その人の変更を破棄して作業を終了させてロックを解除し、他の人が先に進めるようにする。デッドロックの検出はとても難しく、「犠牲者」にとっては非情なものである。同様の手法に、すべてのロックに制限時間を設定するというものがある。この手法では、制限時間に達した時点でロックが解除されるため、それまでの作業が犠牲になる可能性もある。タイムアウトはデッドロック検出よりも簡単に実装できるが、もし誰かがロックした状態を維持している場合には、デッドロックが生じていなくても他の誰かが犠牲になることもある。

タイムアウトと検出はデッドロックが起こったときの対処法だが、他にもデッドロックそのものが起こらないようにする手法もある。デッドロックは、基本的にすでにロックされている人が、さらに他のデータも処理しようとした際に（または、読み込みロックから書き込みロックに変更しようとした場合に）生じる。つまり、デッドロックを防止する方法の1つ

として、作業を開始した時点で強制的にすべてのロックを取得するようにして、それ以上のロックは行えないようにしておくという方法がある。

全員に順番でロックを取得させる方法を指定できる。たとえば、ファイルを常にアルファベット順にロックするようにすることも可能である。この場合、David が Order ファイルのロックを取得した時点では、Customer ファイルのロックはアルファベット順では Order ファイルより前になるため取得できない。この時点で、David は潜在的犠牲者となる。

また、David がすでに持っているロックを Martin が取得しようとしたときに、Martin が自動的に犠牲候補になるように設定することもできる。多少荒っぽい技法ではあるが、簡単に実装でき十分有効な手法である場合が多い。

より慎重に行いたい場合は、複数の手法を同時に使うことができる。たとえば、最初にすべてのロックを全員に強制的に与えておき、何かが起こったときのために、さらに時間制限を加えるという方法である。念には念を入れている方法であるが、この慎重さは賢明だと言える。デッドロックは、簡単にその影響が拡大してしまう恐れがある面倒な問題だからである。

万全のデッドロックメカニズムを実行していると思っていても、予期せぬ出来事が次々に起こることもある。エンタープライズアプリケーション開発では、できるだけシンプルかつ慎重な手法を使用することを推奨する。場合によっては誰かが犠牲になることがあっても、デッドロック対策を怠ったときに起こり得る事態に比べれば、はるかに良い方法である。

5.5 トランザクション

エンタープライズアプリケーションで並行性を処理する最も有効なツールは、トランザクションである。「トランザクション」という言葉からは、金銭や商品の交換を連想することが多い。ATM で暗証番号を入力して現金を引き出すのも、有料道路で通行料を支払うのも、スーパーでビールを買うのもすべてトランザクションである。

このような金銭取り扱いの例から、この用語の的確な定義を引き出すことができる。第1に、トランザクションは開始時点と終了時点が明確に定義された一連の連続した動作である。ATM でのトランザクションはカードを挿入した時点で始まり、現金が引き出されたか残高不足が判明した時点で終了する。第2に、関係するリソースがトランザクションの開始時点と終了時点で一貫していることである。ビールを買った人は、財布の中身は少し減ったが、その分ビールを手に入れている。つまり、彼にとっての資産価値の合計は変わっていないわけである。販売する側も同様で、無料でビールを配っていたら商売は成り立たない。

さらに各トランザクションは、必ずオール・オア・ナッシング（全か無か）で完了する。銀行は、実際に ATM から現金が引き落とされない限り、引き落とし額を口座の残高から差

し引くことはない。人という要素が介在する場合は、このトランザクション中の最後のプロパティ（残高の更新）が忘れられることも起こりうるが、ソフトウェアなら確実に行うことができる。

5.5.1 | ACID

ソフトウェアトランザクションは、以下の用語の頭文字を使用したACIDプロパティという用語で表現されることが多い。

- **原子性（Atomicity）**：トランザクションの中で実行される一連のアクションの各ステップは必ず完了しなくてはいけない。完了しない場合はすべてが開始前の状態に戻る。トランザクションには、部分成立という概念は存在しない。たとえば、Martinが自分の預金口座から小切手口座にいくらか移動しようとしたとき、預金口座からお金を引き出した後にサーバがクラッシュした場合、システムは最初からお金が引き出されていないかのように振舞う。コミットした場合は預金口座からの引き出しと小切手口座への記入の両方が実行され、ロールバックした場合はどちらも実行されない。必ず両方かまたは無かのいずれかである。
- **一貫性（Consistency）**：トランザクションの開始時点から終了時点まで、システムのリソースは必ず一貫した正常な状態でなくてはならない。
- **分離性（Isolation）**：個々のトランザクションの結果は、そのトランザクションが問題なくコミットされるまで、その他のオープンなトランザクションから見えてはならない。
- **耐久性（Durability）**：コミットされたトランザクションの結果は、永続的である必要がある。言い換えれば、「どのような種類のクラッシュが起こっても存続しなければならない」。

5.5.2 | トランザクション可能なリソース

エンタープライズアプリケーションにおけるトランザクションは、データベースに関することが多い。しかし他にも、メッセージキュー、プリンタ、ATMなどトランザクションを使って制御できるリソースは数多くある。このことからもトランザクションを技術的に論じる場合、「トランザクション可能なリソース」という言葉が使われる。「トランザクション可能な」とは、トランザクションを使って並行性を制御できるということである。「トランザクション可能なリソース」では多少無理があり、呼びにくいので、ここでは通例として

「データベース」という言い方をするが、これはトランザクション可能なあらゆるリソースを指していると考えていただきたい。

現在のトランザクションシステムは、スループットを最大限に高めるため、トランザクションができる限り短時間に実行されるように設計されている。そのため、複数のリクエストを含むようなトランザクションは避けたほうがよい。複数のリクエストを含むトランザクションは、一般的に**ロングトランザクション**（疎結合トランザクション）と呼ばれている。

このことから、トランザクションをリクエストの先頭から開始し、その末尾で完了させる手法が一般に使用されている。**リクエストトランザクション**はシンプルなわかりやすいモデルであり、多くの環境でメソッドをトランザクション可能なものとしてタグ付けするだけで容易に指定することができる。

このバリエーションの1つに、トランザクションのオープンをできるだけ遅らせるという方法がある。この**遅延トランザクション**では、すべての読み込みをトランザクション以外の部分で行い、更新するときにだけトランザクションをオープンする。この方法は、トランザクションに要する時間を最小限に抑えることができる。トランザクションのオープンから最初の書き込みまでに時間差があると、即応性は向上するかもしれないが、トランザクションが始まるまで並行性の制御ができないため、一貫性のない読み込みが生じる可能性がある。よほど明確な意図があり、複数のリクエストにまたがるビジネストランザクション（次項を参照）があるのでない限り、そのような構成にするメリットはあまりない。

トランザクションを使用する場合は、何がロックされるかを明確にしておくことが必要だ。多くのデータベースアクションでは、トランザクションシステムによって関連する行がロックされるため、複数のトランザクションから同じテーブルにアクセスできるようになる。ただし、トランザクションによってテーブル内の多くの行がロックされると、データベースが処理できる以上のロックが存在することになり、テーブル全体がロックされてしまうことになる（その他のトランザクションがロックアウトされる）。この**ロックエスカレーション**は並行性に大きく影響することから、ドメインの**レイヤスーパータイプ**レベルには、データ用の「オブジェクト」テーブルを置くべきではない。テーブルは、ロックエスカレーションを引き起こす最有力候補となり、テーブルがロックされるとその他すべての人がデータベースから締め出されてしまうことになる。

5.5.3 ｜即応性に対するトランザクション分離性の制限

通常は、トランザクションの完全な保護を制限して、即応性を向上させる手法を使用している。この手法は、特に分離レベルを定義するときに適用されている。完全な分離レベルを使用する場合には、トランザクションを直列化して使用することも考えられる。トランザクションを並行して実行し、トランザクションを連続して実行したときとまったく同じ結果が

得られる場合、**直列化可能**となる。つまり、先に紹介したMartinがファイル数を数えた例では、直列化可能なレベルでは、彼のトランザクションがDavidのトランザクションが始まる前（12）か後（17）に完了したときと同じ結果が必ず得られることになる。直列化可能なレベルでは、この場合のようにいずれの結果が得られるかは保証できないが、少なくとも正しい数を得ることは保証されるわけだ。

ほとんどのトランザクションシステムは、分離性を4つのレベルで定義しているSQL標準を採用している。直列化可能なレベルは最も強力なレベルであり、それ以下の各レベルでは、特定の一貫性のない読み込みが起こることを容認している。Davidがファイルを修正しているときにMartinがファイル数を数える例を使って、各レベルを解説することにしよう。lockingとmultiphaseの2つのパッケージが存在している。Davidが更新する前はlockingパッケージ内には7つのファイルが、multiphaseパッケージ内には5つのファイルがあったが、Davidが修正を加えた結果、lockingパッケージ内のファイル数は9、multiphaseパッケージ内のファイル数は8になってしまった。Martinは、まずlockingパッケージのファイル数を確認し、その後Davidが両方のパッケージを更新した後にmultiphaseパッケージのファイル数をチェックする。

分離レベルが直列化可能な場合、Martinの答えは12か17になることが保証される。いずれも更新前と更新後のファイル数という意味で正しい数字だ。このように、直列化可能なレベルの分離では同じシナリオで常に同じ結果が得られるとは保証できないが、更新前か更新後いずれかの正確な数を得ることは可能である。

直列化可能なレベルのすぐ下の分離レベルが**繰り返し可能な読み込み**であり、ここでは**幻像**が発生する可能性がある。コレクションにいくつかの要素を追加したにも関わらず、読み込む側がその一部しか見えないときに、幻像が発生している。この例では、Martinがlockingパッケージを見たときファイル数は7である。しかしMartinがmultiphaseパッケージを見るのは、Davidが自分のトランザクションをコミットした後であり、このときmultiphaseパッケージのファイル数は8になっている。Martinは、計15ファイルという誤った答えを得ることになる。幻像は、Martinのトランザクション例のように一部だけが有効でその他は無効な場合に発生するものであり、データを挿入した結果として発生する。

もう1つ下の分離レベルが**繰り返し不可能な読み込み**を発生させる**コミットされた読み込み**である。Martinは実際に個々のファイルを見ているわけではなく、全体の数を見ているとしよう。

繰り返し不可能な読み込みでは、lockingでは7という数が読み込まれる。Davidがトランザクションをコミットすると、multiphaseでは8という数が読み込まれる。繰り返し不可能な読み込みと呼ぶのは、Davidがコミットした後にMartinがlockingパッケージを再度読み込めば、新しいファイル数である9が得られるはずであり、Martinが最初に読み取った7という数は、Davidの更新後には繰り返すことができないからだ。データベース側

から見れば、幻像読み込みよりも繰り返し不可能な読み込みの方が検出しやすい。そのため、繰り返し可能な読み込みはコミットされた読み込みレベルよりも精度は高くなるが、即応性は低くなる。

最も低い分離レベルは、**不確定な読み込み**が生じる**コミットされていない読み込み**である。コミットされていない読み込みレベルでは、他のトランザクションが実際にはまだコミットしていないデータも読むことができる。このことは、2種類のエラーの原因となり得る。Martin が locking パッケージを見たとき、David は最初のファイルを追加したが、2つ目のファイルはまだ追加していなかったとしよう。この場合 Martin は、locking パッケージ内に8つのファイルを見ることになる。これが1つ目のエラーだ。2つ目のエラーは、もし David がファイルを追加した後にトランザクションをロールバックしたときに生じる。この場合 Martin は、存在していないファイルを見たことになってしまう。

表 5.1 は、各分離レベルで発生する読み込みエラーを示している。

精度を確保したい場合は、常に直列化可能な分離レベルを使用するべきである。直列化可能なレベルの短所は、システムの即応性が著しく低下してしまうことにある。場合によっては、スループットを高めるために直列化機能を低下させる必要もある。どのようなリスクを取るか、またエラー排除とパフォーマンス向上のどちらを優先するかはすべて開発者の判断次第である。

すべてのトランザクションで同じ分離レベルを使う必要はないため、即応性と正確性のバランスを維持するためにどのレベルを設定するかは個々のトランザクションごとに判断する必要がある。

表 5.1 ——分離レベルと各レベルで生じる一貫性のない読み込みエラー

分離レベル	不確定な読み込み	繰り返し不可能な読み込み	幻像読み込み
コミットされていない読み込み	発生する	発生する	発生する
コミットされた読み込み	発生しない	発生する	発生する
繰り返し可能な読み込み	発生しない	発生しない	発生する
直列化可能な読み込み	発生しない	発生しない	発生しない

5.5.4 | ビジネストランザクションとシステムトランザクション

ここまで説明してきた内容の大部分は他の人も説明していることだが、システムトランザクションと呼ばれているもの、つまり RDBMS システムとトランザクションモニタによってサポートされているトランザクションについてである。データベーストランザクションとは、開始と終了の命令によって区切られている一連の SQL コマンドグループである。トランザクション内の4番目のステートメントが整合性の制約に違反している場合、データベースはその前の3つのステートメントで実行した内容をロールバックし、呼び出し元にトラン

ザクションが失敗したことを通知する。4つすべてのステートメントが問題なく完了した場合は、1つずつではなくすべてが同時にその他のユーザから見えるようになる。RDBMSシステムとアプリケーションサーバトランザクションマネージャーは、もはや一般的であり、当然のものと考えられている。これらはとても有効であるため、アプリケーション開発者はこれらを熟知している。

だがシステムトランザクションは、ビジネスシステムを使うユーザにとっては意味のないものである。たとえば、オンラインバンキングにおけるユーザのトランザクションは、ログイン、口座の選択、支払いの設定、そして最後に確定して支払うというステップから構成されている。このステップは**ビジネストランザクション**と呼ばれ、システムトランザクションで必要とされたのと同様に、ビジネストランザクションでもACIDプロパティが必要になる。ユーザが支払いをする前にキャンセルした場合、その前の画面で行った変更はすべてキャンセルされるべきであり、支払いの指定をしただけで残高が見えるべきではなく、あくまでもOKボタンを押して実際に支払われた後でだけ見えるようにすべきである。

ビジネストランザクションのACIDプロパティをサポートするには、ビジネストランザクション全体を1つのシステムトランザクション内で実行させるというのが最も明解な解答である。しかし、状況によっては、複数のリクエストが完了しないと処理できないビジネストランザクションもあるので、ロングシステムトランザクション内で1つの結果を実装するために、1つのシステムトランザクションを使用する。ほとんどのトランザクションシステムでは、ロングトランザクションは効率的に動作しない。

これは、決してロングトランザクションを使うべきではないと言っているわけではない。データベースで並行性の必要性がそれほど求められていない場合は、使用することが可能である。むしろ、このようなケースでは使用することを勧める。ロングトランザクションを使用すると、多くの面倒な問題を回避できる。ただし、ロングトランザクションではデータベースがボトルネックとなるため、アプリケーションは拡張性のあるものにはならない。さらに、ロングトランザクションからショートトランザクションへのリファクタリングは、複雑であり十分理解されていない部分でもある。

このような理由から、多くのエンタープライズアプリケーションではロングトランザクションの危険を冒すことができない傾向がある。この場合、ビジネストランザクションを一連のショートトランザクションに分割する。つまり、システムトランザクション間でのビジネストランザクションのACIDプロパティに対応するため、独自のデバイスを使用する必要がある。このような状態を**オフライン並行性**と呼んでいるが、この構成でもシステムトランザクションは数多く存在している。ビジネストランザクションがデータベースなどのトランザクション可能なリソースと相互作用するときは必ず、相互作用がシステムトランザクション内で順番どおりに実行され、リソースの整合性を維持している。ただし後に解説するように、一連のシステムトランザクションを結合しただけでは、適切にビジネストランザク

ションをサポートしているとは言えない。ビジネスアプリケーションは、システムトランザクションとビジネストランザクションの間を結合する別の何かを提供しなければならない。

原子性と耐久性は、ビジネストランザクションでも最も簡単にサポートできるACIDプロパティだ。これらは、ユーザが「保存」を選択したときにシステムトランザクション内でビジネストランザクションのコミットフェーズを実行することでサポートできる。セッションが加えられたすべての変更をレコードセットにコミットする前に、まずシステムトランザクションがオープンになる。システムトランザクションによって、変更が1つのユニットとしてコミットされ永続的なものになることが保証される。唯一扱いにくい部分は、ビジネストランザクションが存続している間は、正確な変更セットを維持するという点である。アプリケーションが**ドメインモデル**を使用している場合は、**ユニットオブワーク**によって変更を正確に記録できる。**トランザクションスクリプト**内にビジネスロジックを配置するには、手動で変更を記録する必要があるが、トランザクションスクリプトを使用するということ自体、そのビジネストランザクションが比較的シンプルなものであることを意味しているので、これが問題になる可能性は少ない。

ビジネストランザクションで使用するときに扱いにくいACIDプロパティが、分離性だ。分離性が確保されていないと、一貫性が失われてしまう。一貫性チェックでは、ビジネストランザクションがレコードセットを無効な状態のままにしないように指示する。1つのトランザクション内で一貫性を維持するためにアプリケーションに課せられる責任は、使用できるすべてのビジネスルールを適用するということである。複数のトランザクションにおけるアプリケーションの責任は、1つのセッションが、レコードセットをユーザの作業が失われるような無効な状態のままにして、別のセッションが読み込み中のデータを変更しないようにすることである。

更新を無効にしてしまうというわかりやすい問題だけでなく、一貫性のない読み込みに関するわかりにくい問題も潜んでいる。データが複数のシステムトランザクションで読み込まれている場合、その一貫性は保証されなくなり、異なる読み込みにより一貫性を失ってしまい、程度によってはメモリ内のデータがアプリケーションエラーを引き起こしてしまうことにもつながりかねない。

ビジネストランザクションは、セッションととても密接な関係にある。ユーザから見れば、各セッションはビジネストランザクションが連なったものとなり（データを単に読み込んでいるだけの場合を除く）、このことから、すべてのビジネストランザクションはシングルクライアントセッション内で実行されるということを前提にしている。1つのビジネストランザクションに対して複数のセッションを持つようなシステムを設計することはできるが、混乱を招くのでするべきではないだろう。

5.6 オフライン並行性制御のためのパターン

並行性の問題は、可能な限りトランザクションシステムに対処させるべきである。システムトランザクションまで含めた並行性制御を扱うのは、並行性を自分で処理しなければならないので、泥水の中に突き落とされるようなものだ。その泥水の中には、仮想のサメや仮想の毒クラゲ、仮想のピラニアなど、あまり近付きたくない生き物がいっぱい生息していて嫌な環境だ。

残念なことに、ビジネストランザクションとシステムトランザクションの不一致によって、その泥水の中を歩いていかなければならない状況に追い込まれることもある。ここで解説するパターンは、システムトランザクションを含む並行性制御に有効と思われるいくつかの技法を盛り込んでいる。

これらの技法は、あくまでも必要な場合に限り使うべきだということを覚えておいてほしい。すべてのビジネストランザクションを1つのリクエスト内に含めることができ、かつシステムトランザクションに一致する場合は、そうするべきである。拡張性を気にせずロングトランザクションを使いたいのであれば、なおさら使うべきである。並行性制御をトランザクションソフトウェアに任せれば、多くの問題を回避することができる。これらの技法は、回避ができないときに限り使うべきものである。並行性が抱える複雑さを踏まえた上でもう一度念を押しておくが、これらのパターンはあくまでも出発点であり、決してゴールではない。間違いなく有効なものである一方で、並行性のすべての病を完治させることはできないのである。

最初にオフライン並行性問題に対応するものとして選んだのは、**軽オフラインロック**だ。基本的に、軽い並行性制御をビジネストランザクション全体で使用するというものである。これを最初に選んだのは、プログラムの手法としては簡単で、最適な即応性を提供してくれるからだ。**軽オフラインロック**の限界は、ビジネストランザクションが失敗しそうだということが、コミットしようとした段階で初めてわかるという点だろう。状況によっては、この遅れが許し難いものになる場合もある。たとえば、ユーザが何時間もかけてリースに関する詳細を入力したにも関わらず、多くの障害があったのでは、ユーザのシステムに対する信頼は失われるだろう。代替案としては**重オフラインロック**がある。このパターンでは、トラブルは早い段階で知ることができるが、プログラミングが大変なだけでなく即応性も低下するので、あまり勧められない。

いずれの手法でも、すべてのオブジェクトのロックを管理する必要がないため、複雑性は大幅に軽減されることになる。**緩ロック**では、オブジェクトグループの並行性をまとめて管理することができる。アプリケーション開発者の作業を楽にできるという意味では、ロックを直接管理しなくても済む**暗黙ロック**を使用するという方法もある。これは単に作業を軽減させるだけでなく、うっかり忘れるという人的なバグも回避することができる（これらのバ

グは見つけるのが大変である)。

並行性についてよく言われるのは、要件がすべて揃った時点で、純粋に技術的観点から決定するというものであるが、私はこれに賛同しかねる。軽い手法か重い手法のどちらで制御するかは、システム全体におけるユーザ体験に影響してくる。**重オフラインロック**によるインテリジェントな設計では、システムのユーザのドメインについて膨大な量の入力が必要になるし、しかも優れた**緩ロック**にするには、ドメインに関する知識が必要となる。

並行性の扱いは、プログラミングタスクの中でも最も難しいとされているものの1つだ。確信をもって並行性コードをテストするのは大変困難な作業となる。並行性のバグは再現するのが難しく、とても追跡しにくい。ここで解説しているパターンは現時点では問題なく動作しているが、この分野は特に難しい分野だと言える。

この道を通る必要がある場合は、できる限り経験者のサポートを得られるようにするべきである。最低限、本章の最後で紹介している書籍は読んでおくべきだろう。

5.7 アプリケーションサーバの並行性

ここまでは、主に共有データに対する複数セッションでの並行性について話してきたが、もう1つ別の形態の並行性もある。アプリケーションサーバにおけるプロセスの並行性だ。サーバはどのようにして複数のリクエストを同時に処理し、またサーバ上のアプリケーション設計にどのように影響するのだろうか。ここまで説明してきた並行性と大きく異なるのは、アプリケーションサーバにおける並行性ではトランザクションが絡んでいないことだ。つまり、これらを考えるときトランザクションの制御は切り離すことができる。

ロックと同期ブロックを伴う明示的なマルチスレッドプログラミングは、とても複雑なものとなる。簡単には見つからない欠陥が突然出てくることがあるため(並行性のバグはほとんど再現不可能なものである)、99%正常に動作しているシステムでも、突発的に不具合が生じる可能性があることになる。ソフトウェアの使用やデバッグはとてつもないストレスを伴うので、私は同期やロックを明示的に処理する必要性を可能な限り避けることにしている。アプリケーション開発者はこれらの明示的な並行性メカニズムと対峙するような状況に陥るべきではない。

最も簡単にこれを処理する方法は、**1つのセッションに1つのプロセス**の手法だ。この場合、各セッションは、それぞれ独自のプロセス内で実行されることになる。この手法の最大のメリットは、各プロセスの状態が完全に分離されていることにある。つまり、アプリケーション開発者は、マルチスレッド処理について心配する必要は一切ない。メモリの分離という点でも、各リクエストが新しいプロセスを開始する、またはリクエスト間に1つのプロセスが待機中の1つのセッションと結び付いているのはとても効率的であると言える。初期の

Webシステムの多くでは、各リクエストに対して新たなPerlプロセスが開始されていた。

この1つのセッションに対して1つのプロセスという手法の問題点は、プロセスがいわば浪費家として例えられ多くのリソースを使うという点にある。より効率的にしたければ、それぞれのプロセスが同時に処理するリクエストを1つに限定しながら、別のセッションからの複数のリクエストを順次処理できるように、プロセスをプールさせることである。**1つのリクエストに1つのプロセス**をプールする手法では、より少ないプロセスで所定のセッション数をサポートできる。分離性の点でもほとんど問題がない。理由は、厄介なマルチスレッドに対処する必要があまりないからである。1つのセッションに1つのプロセスの手法でリクエストを処理する場合の最大の問題点は、リクエストを処理するために使用したリソースを、リクエストが終了した時点で解放しなくてはならないという点である。現行のApache mod-perlではこの仕組みが使用され、多くの大規模かつ重要なトランザクション処理システムでも使われている。

1つのリクエストに1つのプロセスの手法でも、一定の負荷に対処するためには多くのプロセスを必要とする。1つのプロセスで複数のスレッドを実行するようにすれば、スループットをさらに向上させることができる。**1つのリクエストに1つのスレッド**の手法では、各リクエストは、1つのプロセス内の1つのスレッドで処理されることになる。スレッドが使用するサーバリソースはプロセスよりもはるかに少ないため、少ないハードウェアリソースでより多くのリクエストを扱うことができ、サーバもより効率的になる。一方1つのリクエストに1つのスレッドの手法の問題点は、スレッド間に分離性がないため、どのデータにも任意のスレッドからアクセスできてしまうという点である。

私の見解では、1つのリクエストに1つのプロセスの手法について説明したいことがたくさんある。1つのリクエストに1つのスレッドの手法と比べれば、効率という点では劣るが、拡張性という点では同程度である。強固性も向上する。つまり1つのスレッドが壊れるとプロセス全体がダウンするため、1つのリクエストに1つのプロセスの手法では、壊れたスレッドの影響を制限することができる。特に、経験の浅いチームでは、スレッド処理での頭痛の種（バグ修正に要する時間とコストも含む）を減らせれば、ハードウェアのコストが多少アップすることよりも有効であるだろう。1つのリクエストに対して1つのスレッドと、1つのリクエストに対して1つのプロセスの相対コストを算出するため、開発中のアプリケーションのパフォーマンステストを行っているという例は、ごくわずかしかない。

環境によっては、1つのスレッドに割り当てられるデータを分離する場所を中間に設定している場合もある。COMでは、シングルスレッドアパートメント（STA：single-threaded apartment）によってこのような設定を行っている。J2EEもEnterprise Jave Beansも同様である（将来的には分離する可能性がある）。使用しているプラットフォームで似たような設定を行うことができる場合は、（これが何を意味しているかはさておいて）一挙両得である。

1つのリクエストに対して1つのスレッドの手法を使用する場合に最も重要なことは、アプリケーション開発者がマルチスレッドを気にしなくて済む場所に、分離された領域を作成し、そこで処理を行うことである。一般的な方法としては、スレッドがリクエストの処理を開始する際に、新たなオブジェクトを作成するようにし、オブジェクトが他のスレッドから見える場所（たとえば静的変数内など）に置かれないようにするという方法がある。これによって他のスレッドから参照できなくなるため、オブジェクトは分離されることになる。

開発者の多くは、新しいオブジェクトを作成することにあまり積極的ではない。オブジェクトの作成が、プロセスに負荷をかけるという定説が頭にあるからだ。結果、彼らはオブジェクトをプールする手法を使用することが多い。このプールの手法の問題点は、プールされているオブジェクトへのアクセスを何らかの方法で同期させなくてはならない点である。一方で、オブジェクトの作成に要するコストは、仮想マシンとメモリ管理ストラテジーに大きく依存するものである。最近の環境では、オブジェクトの作成はとても速く行えるようになっている[Peckish]（たとえば、Martinの600Mhz P3とJava 1.3で1秒間にいくつのJava Dateオブジェクトを作成できるか想像してみてほしい。解答は後ほど説明する）。新しいオブジェクトをセッションごとに作成すれば、並行性関連の多くのバグを回避できると同時に拡張性も向上する。

この戦術は多くのケースで利用できるが、開発者が避けなくてはならない領域は他にもある。1つは静的クラスベースの変数、またはグローバル変数だ。これらを使用するためには同期が必要となるからである。これは、シングルトンの場合でも当てはまる。

ある種のグローバルメモリが必要な場合は、**レジストリ**の使用を勧める。このパターンでは、静的変数に見えるものでも、実際にはスレッド特化のストレージを使用するように実装できる。

セッションのためのオブジェクトを作成でき、比較的安全なゾーンを作ることができたとしても、オブジェクトによっては作成の負担が大きくなり、別の方法で処理しなくてはならないものもある（最も一般的なものとしては、データベース接続がこれに相当する）。この場合、オブジェクトを明示的プール内に置き、接続が必要なときに取得し、終了したら返すようにするとよい。その場合、これらの動作は同期している必要がある。

5.8 参考文献

本章で解説している内容は本来とても複雑なものであり、ここでは表面に触れたに過ぎない。さらに深く掘り下げたい場合は、[Bernstein and Newcomer]、[Lea]、[Schmidt et al.]などから始めることを勧める。

第6章

セッションステート

並行性についての説明で、ビジネストランザクションとシステムトランザクションの違いについても触れた（第5章、P77）。この違いは並行性だけでなく、ビジネストランザクション内部で使用されたが汎用データベースレコードにコミットする前のデータをどう格納するかにも影響してくることになる。

ビジネストランザクションとシステムトランザクションのそもそもの違いは、ステートレスセッション対ステートフルセッションの違いにある。この件に関しては多くの人が説明しているが、私の見解では、基本的な問題はステートレスとステートフルサーバシステムの技術的な問題の中に隠されることもある。まず考える必要があるのは、いくつかのセッションの中には本質的にステートレスなものがあること、さらに状態をどう扱うかを決定するということである。

6.1 ステートレス性の価値

ステートレスサーバとは何を意味しているのだろうか。オブジェクトであることの意義はもちろん、それらが状態（データ）と振る舞いを組み合わせたものであるという点にある。本来ステートレスオブジェクトとは、フィールドを持たないオブジェクトのことを指す。このようなオブジェクトは実在し、時折り出没するが、お目にかかることはあまりない。私の見解では、ステートレスオブジェクトは悪しき設計と言ってもいいだろう。

しかし、分散エンタープライズアプリケーションにおけるステートレスについて述べる場合、一般的に言われているステートレスサーバとはリクエスト間に状態を保持していないオブジェクトのことであるが、このようなオブジェクトはフィールドを持っていることもある。ただし、ステートレスサーバ上のメソッドを呼び出しても、フィールドの値は定義されていない。

たとえば、書籍に関する情報を含むWebページを返すオブジェクトなどは、ステートレ

スサーバオブジェクトであると言えるだろう。最初にURLにアクセスして呼び出すが、ここでのオブジェクトはASPドキュメントやサーブレットの可能性がある。このとき、サーバはURL内に提示したISBN番号を使ってHTTPレスポンスを生成することになる。処理中の場合には、サーバオブジェクトはHTMLを生成する前に、書籍のISBN、Title（タイトル）、Price（価格）などの情報をデータベースから取得した時点でフィールドに一時保管する場合もあり、またはユーザに表示するレビューについてのビジネスロジックを動作させる場合もある。ただし、HTMLが生成されると、これらの値は意味のないものになり、次のISBNはまったく別のものとなる。サーバオブジェクトは、誤りを避けるために再び初期化して古い値を消去することになる。

仮に、特定のクライアントIPアドレスがアクセスしたすべてのISBNを記録しておく場合を考えてみよう。サーバオブジェクトが保持するリスト内にこれらのISBNを記録するが、リストはリクエストの間も保持されていなければならないので、サーバオブジェクトはステートフルなものになる。ステートレスからステートフルへのシフトは、「レス」と「フル」という文言を変えればいいというような簡単なものではない。多くの人にとって、ステートフルサーバは頭痛の種以外何ものでもない。なぜだろうか。

最大の問題はサーバのリソースである。ステートフルサーバオブジェクトは、ユーザがWebページを閲覧している間、すべての状態を維持しておく必要があるのに対して、ステートレスサーバオブジェクトは、他のセッションからの別のリクエストも処理することができる。以下の例はあり得ないことかもしれないが、概要を理解するには役立つ例である。書籍について調べたいユーザが100人いて、1冊の書籍に対するリクエストの処理時間には1秒だった場合を考えてみる。各ユーザは10秒ごとにリクエストを出し、すべてのリクエストが完全に同じように動作する。ステートフルサーバオブジェクトでユーザのリクエストを追跡するには、ユーザ1人あたり1つのサーバオブジェクトが必要となる。つまり100個のオブジェクトが必要になるわけだ。しかし、これらのオブジェクトは、全体の9割の時間帯は何の動作も行わず待機しているだけである。一方、ISBNを追跡せずにステートレスサーバオブジェクトを使用すると、10個のオブジェクトがフル稼働しつづけるのである。

つまり、メソッド呼び出しの間に状態がなければ、どのオブジェクトがリクエストに応じるかは関係なくなるが、状態を格納する場合は、常に同じオブジェクトを使う必要がある。ステートレスの場合はオブジェクトをプールできるので、より少ないオブジェクトでより多くのユーザに対応できる。待機中のユーザが多ければ多いほど、ステートレスサーバの利用価値は高くなる。このことから、ステートレスサーバがトラフィックの多いWebサイトでは有効だということが容易に想像できる。ステートレスは、HTTPがステートレスプロトコルであることからもWebには十分適している。

では、すべてをステートレスにするべきなのだろうか。可能な場合はそうするべきだ。問題は、クライアントとの相互作用の多くが本質的にステートフルであるという点にある。た

とえば、何千というｅコマースアプリケーションで使われているショッピングカートシステムを見てみよう。ユーザの相互作用には、複数の書籍を閲覧し、その中から実際に購入する商品を選択することが含まれる。ショッピングカートの内容は、ユーザのセッションが完了するまで保持されている必要がある。つまりステートフルなビジネストランザクションになるので、セッションもステートフルなものになることを意味している。単に書籍を閲覧するだけで何も購入しない場合はステートレスだが、購入する場合はステートフルになる。つまりこの例では、本を買う金がある限り、状態を避けることはできない。そこでその対処法を考えてみると、実はステートレスサーバでステートフルセッションを使用するという素晴らしい方法もあるが、この興味深い方法を取り入れるべきかどうかには疑問が残る。

6.2 セッションステート

ショッピングカートの中身は**セッションステート**になっている。つまり、カート内のデータは特定のセッションだけに関連したものになっている。状態はビジネストランザクション内にあるため、他のセッションとビジネストランザクションとは分離されている（さらに言うと、各ビジネストランザクションは１つのセッション内でだけ実行され、各セッションは同時に１つのビジネストランザクションしか処理しないということが前提である）。セッションステートと、私が**レコードデータ**と呼んでいるものは同じものではない。レコードデータは、データベースに長期間保持されるデータのことで、すべてのセッションから見えている。セッションステートがレコードデータとなるには、コミットされる必要がある。

セッションステートはビジネストランザクション内にあるので、トランザクションで一般的に考えられている ACID（原子性、一貫性、分離性、耐久性）などのプロパティの多くを持っていることにもなるが、現状ではこの因果関係が必ずしも理解されているとは言えない。

興味深い関連性として一貫性への影響を挙げることができる。顧客が保険ポリシーを編集しているとき、ポリシーの現行の状態が顧客によっては規定に合わない可能性もある。顧客が値を変更すると、それがリクエストとしてシステムに送られ、システムは無効値であると応答する。これらの値はセッションステートの一部となるわけだが、有効な値とはならない。セッションステートではこのような状況が生じる場合が多く、処理中は妥当性確認ルールと合致せず、ビジネストランザクションがコミットされた場合にだけ合致するのである。

セッションステートで最も懸念される問題が、分離性の処理である。多くの人が関わっているので、顧客がポリシーを編集している間は何が起こってもおかしくない。最も分かりやすい例を挙げると、２人が同時にポリシーを編集している場合である。ここでの問題は、直接変更するということだけではない。たとえば、ポリシーそのものと Customer（顧客）レコードという２つのレコードがあるとすると、ポリシーには、Customer（顧客）レコード

のZip Code（郵便番号）に依存するリスク値がある。顧客がポリシーの編集を開始して10分後に何らかの操作でCustomer（顧客）レコードをオープンし、Zip Code（郵便番号）が見える状態になったとする。そこでその10分の間に他の誰かがZip Code（郵便番号）とリスク値を変更したとしたら、一貫性のない読み込みが起こることとなる。このような場合の対処法については、第5章を参照してほしい。

セッションで保持されているすべてのデータが、セッションステートとしてカウントされるわけではない。セッションは、実際にはリクエスト間には必要のないデータも、パフォーマンスを向上させるためにキャッシュして格納していることもある。キャッシュは振る舞いに影響を与えずに削除できるので、適切な振る舞いのためにリクエスト間に格納される必要があるセッションステートとは異なるものである。

6.2.1 | セッションステートを格納する方法

格納する必要があるとわかったセッションステートは、どのように格納すればいいのだろうか。考えられる選択肢を、基本的な3つに分けてみよう。

クライアントセッションステートでは、データをクライアント上に格納する。これを行う方法はいくつかある。つまり、Webプレゼンテーションのためにデータを URL にエンコードするクッキーを使用する、Webフォームのいくつかの隠しフィールドにデータを直列化する、リッチクライアント上のオブジェクト内にデータを保持するなどである。

サーバセッションステートは、リクエストの間にメモリにデータを保持させるというシンプルな方法である。ただし、直列化されたオブジェクトのようなより耐久性の優れた場所に、セッションステートを格納するメカニズムを保持している。オブジェクトは、アプリケーションサーバのローカルファイルシステムに格納したり、共有データソース内に配置したりすることもできるが、その場所は、キーとしてセッション ID を持つシンプルなデータベーステーブルや、直列化されたオブジェクトが値になるデータベーステーブルになる可能性がある。

データベースセッションステートもサーバサイドストレージだが、ここでは、データをテーブルとフィールドに分割し、より永続性の優れたデータを格納する方法を取り上げて説明する。

どの選択肢を使うかを決めるにあたっては、考慮すべき点がいくつかある。まず、クライアントとサーバ間で必要なバンド幅について考える。**クライアントセッションステート**では、すべてのリクエストによってセッションデータが回線を通じて転送される必要があるが、フィールドが数個しかない場合は問題にならないとしても、データ量が多くなれば転送量も多くなる。あるアプリケーションではデータ量は1メガバイト程度かもしれないし、私たちのチームのようにシェークスピアの戯曲3作にも匹敵する量になることもある。私たちは転

送にXMLを使用していて、決して最適なデータ伝送形式であるとは言えないが、それでも大量のデータを処理できる。

もちろん、プレゼンテーションで見せなければならないデータを転送する場合もあるが、**クライアントセッションステート**では、すべてのリクエストにおいて、たとえクライアント側で表示する必要がなくても、サーバが使用するデータはすべて転送されなくてはならない。つまり、格納する必要があるセッションステートの量が極端に少なくない限りは、**クライアントセッションステート**を使うべきではない。その上セキュリティと整合性にも注意する必要がある。データを暗号化していない限り、悪意のあるユーザがセッションデータを改ざんする可能性もある。

セッションデータは分離される必要がある。ほとんどの場合、あるセッションで行われていることが別のセッションの実行に影響を与えてはいけないからである。たとえば、飛行機の予約を入れたとしても、それが確定されるまでは他のユーザに影響を及ぼすべきではない。セッションデータであることの意味には、セッション外からは一切見えないということもあるが、これは**データベースセッションステート**を使用するときは厄介なものになる。その理由は、セッションデータをデータベース内に存在するレコードデータから分離しなくてはいけないからである。

ユーザの数が多いときは、クラスタ化してスループットを向上させることも考慮すべきであり、セッションを移行させる必要があるかどうかも考えた方がよいだろう。**セッション移行**によってセッションをサーバからサーバへと移行することができ、1つのサーバで1つのリクエストを、別のサーバで別のリクエストを処理できるようにする。これの反対に位置するのが、特定のセッションのためのすべてのリクエストを1つのサーバで処理する**サーバアフィニティ**だ。サーバ移行は、サーバ間のバランスも保つことができ、特にセッションが長い場合は有効であるが、**サーバセッションステート**を使用する場合は少し厄介なものになる。理由は、セッションを処理するマシンだけが状態を容易に見つけられることが多いからである。これを解決する方法として、**データベースセッションステート**と**サーバセッションステート**の中間に位置させる方法がある。

サーバアフィニティは一見すると気付かないかもしれないが、これは大きな問題になる可能性もある。サーバアフィニティを保証しようとしても、クラスタリングシステムでは、呼び出しがどのセッションの一部なのかを検査できないことがある。その結果、アフィニティが強くなり、1つのクライアントからのすべての呼び出しが同じアプリケーションサーバに送信される。これはクライアントのIPアドレスで判断される場合が多いが、クライアントがプロキシーを介していると、複数のクライアントが同じIPアドレスを使っていることになり、すべて特定のサーバに集中してしまうことになる。この状況は、1つのサーバで処理するほとんどのトラフィックがAOL（アメリカオンライン）で使用するIPアドレスをまとめたものだった場合などは、もっと深刻な問題となるだろう。

サーバでセッションステートを使う場合は、すぐに使用できる形式である必要がある。**サーバセッションステート**の場合、セッションステートはサーバが保持していて**クライアントセッションステート**もサーバ内部にあるため、希望する形式で配置しておく必要があることが多い。一方**データベースセッションステート**の場合、データベースまでアクセスし、取得する必要がある（さらに何らかの変換が必要になる場合もある）。つまり、それぞれの手法は、それぞれ異なる方式でシステムに応答する必要があるため、データのサイズや複雑さもここでは影響してくることになる。

一般向け小売システムを運営している場合、各セッションは大量のデータを扱わないが、多くの待機中のユーザを持つことになる。そのため、パフォーマンスを考慮すると**データベースセッションステート**が適切ということになる。一方リースシステムでは、膨大な量のデータをリクエストごとにデータベースへ挿入あるいはデータベースから抽出するという面倒なリスクを抱えなければならないため、この場合は**サーバセッションステート**の方がより良いパフォーマンスを実現できる。

多くのシステムにとって最大の悩みの種といえば、ユーザがセッションをキャンセルして「なかったことにしてくれ」と言ったときの対処法である。これは、特にB2Cアプリケーションでは厄介な問題となる。その理由は、ユーザは「なかったことにしてくれ」とも言わずにいなくなってしまい、そのまま帰ってこないからだ。このケースで有効なのが、ユーザを容易に忘れることができる**クライアントセッションステート**である。別の手法では、一定の時間が経過した時点でキャンセルするようにシステムを設定して、キャンセルされたと分かった時点でセッションステートを破棄できるようにしておく必要がある。適切に実装された**サーバセッションステート**では、自動タイムアウトを使ってこの機能を実行できる。

ユーザがキャンセルしたときのことだけではなく、システムが逆にキャンセルしなければならないような状態のときも考えておく必要がある。つまり、クライアントがクラッシュしたり、サーバがダウンしたり、ネットワーク接続が切断したりしてしまうような場合だ。**データベースセッションステート**ではこれら3つのケースに適切に対処することができる。**サーバセッションステート**は、セッションオブジェクトが不揮発性のメディアに格納されているか、またはそのメディアがどこにあるかによって存続できるかどうかが決まる。**クライアントセッションステート**はクライアントがクラッシュした場合は存続できないが、その他の障害では存続できるようにしなければならない。

また、これらのパターンに必要な開発労力も忘れてはならない。開発リソースから考えると最も簡単なのは**サーバセッションステート**である。リクエストの間にセッションステートを保存する必要がない場合には、極めて容易である。しかし**データベースセッションステート**と**クライアントセッションステート**では、データベースから変換するコードやセッションオブジェクトが使用するフォームにフォーマットを転送するコードを含んでいるので、これらの対処の作業を必要とすることから、**サーバセッションステート**のように容易には多くの

機能を構築することはできない。特に、データが複雑な場合はなおさらである。一見すると、すでにデータベーステーブルへのマッピングが済んでいる場合は、**データベースセッションステート**は複雑なものに見えないかもしれないが、他の目的でのデータベースの使用をセッションデータから分離する場合には、より以上の開発労力が必要となる。

これら3つの手法は、相互に排他的なものではない。2つまたは3つの手法を組み合わせて、セッションステートの異なる部分を格納することもできる。ただし、通常はこのように組み合わせることで作業はより複雑になっていく。状態のどの部分が、システムのどの部分に送られるのかが確実にわかってはいないからである。それでも、**クライアントセッションステート**以外のパターンを使用する場合は、他の状態が別のパターンを使って保持されていたとしても、少なくとも**クライアントセッションステート**内のセッション識別子は保持しておくべきである。

私は**サーバセッションステート**を好んでいる。サーバのクラッシュに耐えられるようにメメントがリモートで格納されている場合などは特に好ましい。また、**クライアントセッションステート**も、セッションIDとセッションデータが極度に少量の場合にも好んで使用する。しかし、**データベースセッションステート**は、フェイルオーバーとクラスタリングが不要な場合で、さらにリモートでメメントを格納できず、セッション間の分離が重要視されていない場合でなければ、使用したいとは思わない。

第7章

分散ストラテジー

オブジェクトという概念が現れてしばらく経つが、最初から開発者はオブジェクトを分散したがっていたように思える。ただし、オブジェクトに限らず分散させるということには、多くの人が気付かない落とし穴が潜んでいる[Waldo et al]。特に、ベンダーが配布する小奇麗なパンフレットに頼るような場合には注意が必要である。本章では、このいくつかの厳しい教訓について紹介する。それは、私の顧客の多くが苦労して身につけた教訓である。

7.1 分散オブジェクトの誘惑

デザインレビューの際に、年に数回、繰り返し目にするプレゼンテーションがある。新しいなんとかというシステムのシステム設計者が、誇らしげに新しい分散オブジェクトシステムのプランを説明する。ここでは仮に、ある種の受発注システムとしよう。その設計図は、図7-1のようなものだ。Cutomer（顧客）、Order（注文）、Product（製品）、Delivery（配送）の各リモートオブジェクトがそれぞれ個別に用意されている。各オブジェクトは個別のコンポーネントになっていて、各コンポーネントは個別の処理ノード上に配置してある。

「なぜこうなっているのですか」と質問してみると、その設計者は少しけげんそうな顔をして、「もちろん、パフォーマンスを考慮したからですよ」と答える。「各コンポーネントを別々のボックスで実行できるので、あるコンポーネントに処理が集中する場合は、ボックスを追加することでアプリケーションへの負荷を分散できるのです。」 この時点で、私を見る彼の目は奇異になり、あたかも私が分散オブジェクトについて何も知らないのではないかという顔をする。

このような場合、私は妙なジレンマに陥る。「このような設計ではだめだ！」と叫んで部屋から追い出されるべきだろうか。または顧客の理解を得るべく懇々と説得すべきだろうか。商売上は絶対に後者を選択するべきだが、顧客が自画自賛している設計を諦めさせるのは想像以上に骨の折れる仕事でもある。

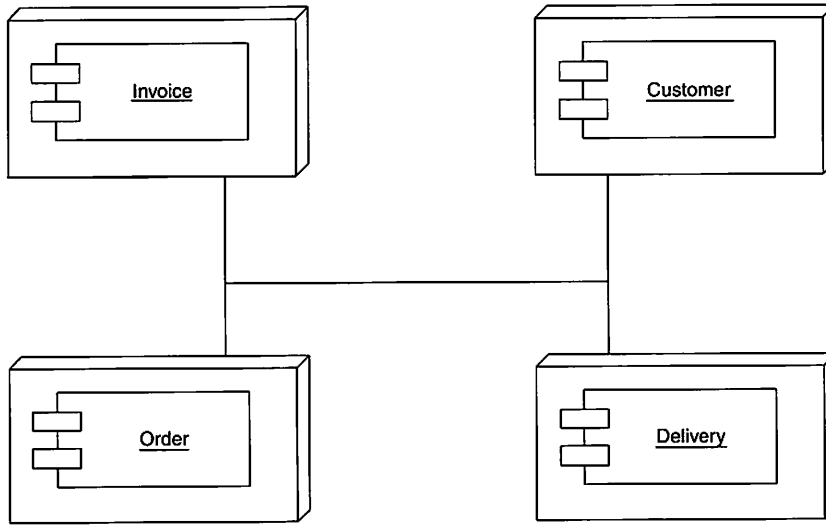


図 7.1 ——異なるノードに別々のコンポーネントを置くことによるアプリケーションの分散

ここまで読み続けてくれた読者は、私が分散設計をまったく評価しない理由を知りたいはずだ。多くのツールベンダーによれば、分散オブジェクトの良い点はいくつものオブジェクトを好きなように処理ノードに配置できることにある。また、ツールベンダーが提供する強力なミドルウェアでは透過性も提供され、透過性によりプロセス内、またはプロセス間で、オブジェクト同士で呼び出し合うことができるようになる。この際、呼び出される側が同じプロセス内にあるのか、別のプロセスあるいはマシンにあるのかを知っている必要はない。

透過性は有益で、分散オブジェクト内では多くのものを透過させることができるが、パフォーマンスは例外である。この設計者は、パフォーマンスを高めるためにオブジェクトを分散しているが、彼の設計はパフォーマンスを低下させるか、システムの設計および導入をとても難しくするか、または多くの場合その両方の問題を引き起こすこととなる。

7.2 リモートインタフェースとローカルインタフェース

クラスモデルによる分散がうまく動作しない主な原因は、使用するコンピュータに基本的に大きく関係している。プロセス内の手続き呼び出しはとても高速で行われるが、2つの別個のプロセス間での手続き呼び出しの処理速度は大幅に低下する。そのプロセスが別のマシンで実行されている場合は、さらに大幅に低下する（ネットワークの形態によって低下幅は違ってくる）。

結果として、リモートで使用されるオブジェクト用のインタフェースは、同じプロセス内

でローカルで使用されるオブジェクト用のインタフェースとは違うものにする必要がある。

ローカルインタフェースは、細かい粒度のインタフェースであるほど優れたものである。したがって、Addressクラスがある場合、優れたインタフェースでは、都市の取得、州の取得、都市の設定、州の設定などのように、それぞれ個別のメソッドを持つ。このような細かい粒度のインタフェースが優れているのは、多くの小さな部品で構成され、さまざまな方法でそれらが組み合わされオーバーライドすることで、将来的に拡張できる設計という一般的オブジェクト指向原則に従っているからである。

ただし、この細かい粒度のインタフェースはリモートでは効率が悪くなってしまう。メソッドの呼び出しが遅い場合は、都市、州、郵便番号の取得または更新を3つではなく1つの呼び出しで行いたいはずであり、結果としてインタフェースの粒度は粗くなり、柔軟性や拡張性ではなく、呼び出しを最小限に抑えるために設計されたものとなる。ここでは、住所の詳細の取得や、住所の詳細の更新などというインタフェースが必要となる。プログラムするにはかなり難しいものではあるが、パフォーマンスから見ると必要なものである。

当然、ベンダーは、彼らのミドルウェアではリモートコールにもローカルコールにもオーバーヘッドはないと言うだろう。ローカルコールの場合は、ローカルコールの速度で処理されることになるが、リモートコールになると速度は低下する。つまり、リモートコールに関する代償は、必要なときにだけ払えばよいということになる。これはある程度当てはまることで、リモートで使用される可能性があるオブジェクトは粗い粒度のインタフェースを持つ必要があり、リモートでは使われないオブジェクトは細かい粒度のインタフェースを持つ必要があるという基本構造は変わらない。2つのオブジェクトが通信するとき、必ずどちらを使うかを選択しなくてはならない。オブジェクトが別のプロセス内に行く可能性がある場合は、粗い粒度のインタフェースを使う必要があり、難易度の高いプログラミングモデルが要求される。言うまでもなく、代償は必要なときにだけ払えばよいので、プロセス間での共同作業はできるだけ少なくする必要があるということになる。

このことから、シングルプロセスの世界で設計したクラスのグループに対して、CORBAなどをあてがうだけで分散モデルにすることはできない。分散設計はもっと複雑なものだからだ。クラスをベースにした分散ストラテジーでは、リモートコールが頻繁に発生するシステムになり、その結果粗い粒度のインタフェースが必要となる。最終的には、すべてのリモート可能なクラスに粗い粒度のインタフェースを持たせたとしても、膨大な数のリモートコールを発生させ、おまけに修正するのも難しいシステムになる。

以上のことから、私が唱える「分散オブジェクト設計の第一法則」にたどり着く。つまり、オブジェクトを分散してはいけないのである。

では、複数のプロセッサを効率的に使うにはどうすればよいのだろうか。ほとんどの場合、クラスタリングを使うことである（図7.2参照）。すべてのクラスを1つのプロセスに入れ、そのプロセスの複数のコピーをそれぞれのノード上で実行する。この方法では、各プロセス

はローカルコールを使うので処理が高速になる。プロセス内のすべてのクラスで細かい粒度のインタフェースを使うようにし、プログラミングモデルをシンプルにして、より保守しやすくすることもできる。

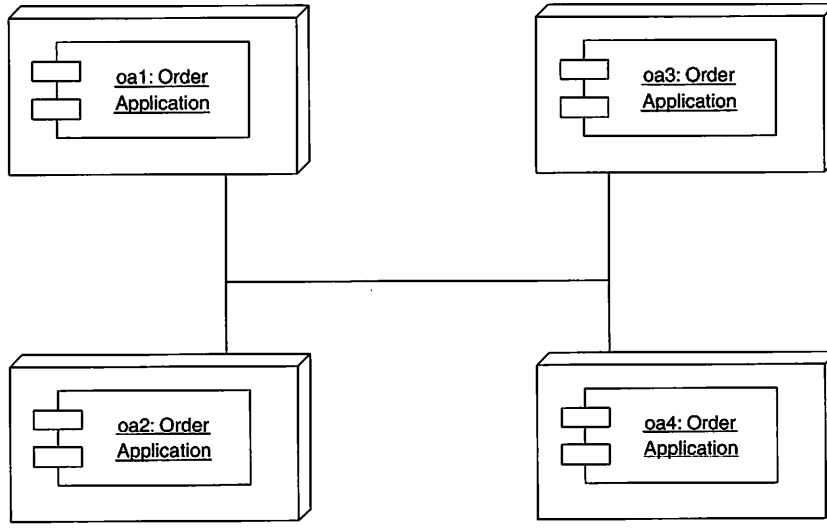


図 7.2 ——同じアプリケーションのコピーを別々のノードに配置したクラスタリング

7.3 | 分散が必要な場所

分散境界を最小限に抑え、可能な限りクラスタリングを介してノードを使うようにしたい。しかしこの手法にもそれなりの制限が存在する。つまり、プロセスを分離しなければならない場所が存在することである。鋭い人であれば、追い詰められたネズミの行動のように奮闘し、できる限りこれらの問題を解決しようと試みるだろうが、すべてを解決することはできない。

- 明確に分離できるのは、従来のクライアントとビジネスソフトウェアのサーバである。ユーザのデスクトップにある PC のノードは、データの共有リポジトリのものとは異なる。それらは別々のマシンなので、通信するプロセスは別に用意する必要がある。クライアントとサーバの分離は、このプロセス間分離の典型的なものである。
- 2 つ目の分離としてよく見られるのが、サーバベースアプリケーションソフトウェア（アプリケーションサーバ）とデータベース間においてである。もちろ

ん、この分離は必ずしもする必要がない。ストアドプロシージャのようなものを使って、すべてのアプリケーションソフトウェアをデータベースプロセス内で実行させることもできる。ただし、この方法はあまり実践的ではないため、やはり別々のプロセスを持たせる必要がある。これらのプロセスを同じマシン上で実行させることは可能だが、別々のプロセスが発生した際は、ほとんどのコストをリモートコールの方に向ける必要がある。幸いなことに、SQL はリモートインタフェースとして設計されているので、コストを最小限に抑えるように配分できる。

- プロセスの分離は、Web システム内や Web サーバとアプリケーションサーバの間でも起こることがある。すべてが同等の場合、Web サーバとアプリケーションサーバを 1 つのプロセス内で実行するのが好ましいが、常にこれらが同等であるとは限らない。
- ベンダーの違いにより分離させなければならないこともある。ソフトウェアパッケージを使っている場合、大抵は自身のプロセス内で実行されるので、分散が生じる。ただし、優秀なパッケージであれば、粗い粒度のインタフェースを持っているはずだ。
- 最終的には、どうしてもアプリケーションサーバソフトウェアを分離せざるをえないような場合もある。これを本能的に回避する手段をとることは理解できるが、それでもどうしようもない場合もある。このような場合は、覚悟を決めてソフトウェアをリモートの粗い粒度のコンポーネントに分割するしかない。

Colleen Roe 氏の忘れがたい言葉を借りると、ここで最も重要なテーマは「オブジェクト分散では可能な限り倹約する」ということだろう。可能であるなら、覚悟を決めて手放すことだ。

7.4 分散境界の扱い

システムを設計するときは分散境界をできる限り制限しなければならないが、分散境界が存在する以上はそれらを考慮する必要があり、すべてのリモートコールはインターネット経由で運ばれることになる。システム内のあらゆる個所が、その形を変えてリモートコールを最小限に抑えようとするが、このコストはある程度計算できるものである。

一方、1 つのプロセス内では粗い粒度のオブジェクトを使った設計もできる。このとき重要なのは、粗い粒度のオブジェクトを分散境界に配置させ、細かい粒度のオブジェクト向けのリモートインタフェースとして使用するということである。粗い粒度のオブジェクトは、

権限を委任する以外は特に何もしないので、細かい粒度のオブジェクト用のファサードとして機能する。このファサードは分散が目的なので、その名前も**リモートファサード**になる。

リモートファサードを使えば、粗い粒度のインタフェースがもたらすさまざまな問題を最小限に抑えることができる。この方法では、リモートサービスを本当に必要としているオブジェクトだけが粗い粒度のメソッドを使うことになり、このような代償を払わなければならないことは開発者にも明らかである。透過性にはメリットがあるが、潜在的なリモートコールに関しては透過性が求められていない。

ただし、粗い粒度のインタフェースを単なるファサードとして使うことで、同じプロセス内で実行中だとわかっていればいつでも、細かい粒度のオブジェクトが利用できることになる。こうすれば、分散ポリシー全体がより明示的なものとなる。**リモートファサード**と密接な関係にあるのが**データ変換オブジェクト**である。粗い粒度のメソッドが必要なだけでなく、粗い粒度のオブジェクトを転送する必要もある。アドレスを催促する場合、その情報は1つのブロックで送信される必要がある。通常は、ドメインオブジェクトそのものは送信できない。これは、Web内でドメインオブジェクトが細かい粒度のローカル相互オブジェクト参照と結び付いているからであり、そこでクライアントが必要なすべてのデータを取り出し、特定の転送用オブジェクトにバンドルさせる。**データ変換オブジェクト**という呼び名の由来はここにある（エンタープライズJavaコミュニティの人たちの多くは、これを**バリューオブジェクト**と呼んでいるが、それでは別の意味での**バリューオブジェクト**と区別できない）。**データ変換オブジェクト**は回線の両側にあるので、それが回線上で共有されていないものを参照しないようにしておくことが重要になる。つまり、**データ変換オブジェクト**は、別の**データ変換オブジェクト**か、文字列のような基本的なオブジェクトしか参照しないようにしておくのが一般的なのである。

分散の別の手法として、プロセス間でオブジェクトを移行させる仲介役を設けるという方法が考えられる。つまり、**レイジーロード**の仕組みを使い、データベースからレイジーに（ダラダラと）読み込む代わりに、回線上ではオブジェクトを移動させる。この方法の難しい点は、最終的に多くのリモートコールが発生しないようにすることである。実際にアプリケーションでこの方法を取り入れた例は見たことがないが、O/Rマッピングツール（例：TOPLink）の中にはこの機能を備えているものもあり、高い評価を得ているようだ。

7.5 分散用インタフェース

従来、分散コンポーネント用のインタフェースは、グローバル手続きかオブジェクト上のメソッドを伴うリモート手続き呼び出しをベースにしたものだった。しかし、ここ数年は、XML over HTTPをベースにしたインタフェースも出現してきた。この種のインタフェー

スとしてはSOAPが最も一般的になりつつあるが、これについては多くの人がすでに何年もの経験を積んでいる。

XMLベースのHTTP通信は便利で、その理由はいくつかある。まず、1回の往復だけで構造化された形式の大量のデータを容易に送ることができる。リモートコールの数はできるだけ減らしたいので、この点は都合がいい。XMLは多くのプラットフォームで現在出回っているパーサーの共通フォーマットとして使われているため、まったく異なるプラットフォームで構築されたシステム間での通信が可能になり、HTTPも今では国際仕様として定着している。XMLは、回線上のどこで何が起こっているかを簡単に見られるテキストであると言ってもよい。HTTPはまた、セキュリティまたは政治上の理由で他のポートが開きにくい場合でも、簡単にファイアウォールを通過することができる。

事実、オブジェクト指向のクラスのインタフェースとメソッドも値は持っている。転送されたすべてのデータをXML構造と文字列に移動することで、リモートコールには相当な負荷がかかる。確かにアプリケーションは、XMLベースのインタフェースをリモートコールに置き換えれば、パフォーマンスが劇的に向上する。回線の両側が同じバイナリ構造を持っていた場合、XMLインタフェースには多くの頭字語以外にあまり得るものがない。同じプラットフォームで構築された2つのシステムがある場合は、そのプラットフォームに備わっているリモートコール構造を使うべきである。逆に異なるプラットフォーム間で互いに通信したいときは、Webサービスが便利である。より直接的な手法がない場合に限り、XML Webサービスを使うというのが私のスタンスだ。

もちろん、HTTPインタフェースをオブジェクト指向インタフェース上に覆いかぶせ、両方の利点を採用するという方法もある。Webサーバへのすべての呼び出しを、その下層のオブジェクト指向インタフェースへの呼び出しに変換する。ある意味では、これは両方の利点を採用するということになるが、Webサーバとリモートオブジェクト指向インタフェース用の機構が必要となるため、複雑性は増すことになる。したがって、HTTPとリモートオブジェクト指向APIが必要な場合か、セキュリティやトランザクション処理用のリモートオブジェクト指向APIの設備の方が非リモートオブジェクトを使うよりも問題を簡単に処理できる場合に限り、この方法を使うべきである。

ここでは、同期RPCベースインタフェースを前提に話を進めてきたが、必ずしもこれが分散システムを扱う最高の手段であるとは思っていない。最近では、私はメッセージベースの手法、つまり本質的に非同期なものを好むようになりつつある。メッセージベース用のパターンについて掘り下げていくと、それだけで一冊の本になるので、本書ではこれ以上述べないことにする。いずれそのような本が出版されることを願いつつ、今のところ私にできることは、非同期メッセージベース手法について調べて見るよう勧めることである。特に、現在まで公開されている例のほとんどが同期であっても、非同期の環境でWebサービスを使うのが最も有効だと思っている。

第8章

まとめ

ここまでの章では、システムの一側面をとらえて扱うためのさまざまなオプションについて説明してきたが、本章ではそれらのことを踏まえ、エンタープライズアプリケーションを設計する際にどのパターンを使うべきかという難問に答えていくことにする。

本章で述べるアドバイスは、多くの点で前章までに述べてきたことの反復である。正直に言うと本章が必要かどうか悩んだのだが、今までの内容をまとめることで、本書で紹介しているパターンが目指しているものの概略だけでも把握できるのではないかと考えて付け加えることにした。

私は、自分でアドバイスできる限界をはっきり意識している。『ロード・オブ・ザ・リング』の中で、フロドも「エルフ達に相談しに行っても、良いとも悪いとも言ってくれない」と言っていた。私は底知れぬ知識を持っているわけではないが、エルフ達の対応は理解できる。アドバイスは押しつけがましい危険な贈り物になることが多いからである。本書を読んでプロジェクトの設計上の判断を下そうとしている人は、そのプロジェクトに関して私よりもはるかによく理解しているはずだ。賢者の大きな悩みの一つは、会議や電子メールなどで、設計やプロセスの判断に関するアドバイスを求められることである。たった5分の説明を聞いて的確なアドバイスをするのは不可能である。本章は、読者が置かれている苦しい状況を考慮せずに書かれたものだ。

したがって、本章は、あるがままのものとして読んでいただきたい。私はすべての解答を知っているわけではなく、読者の質問を聞くこともできない。ここに書かれている内容は、読者のアイデアを引き出すためのものであり、決してそのアイデアに代わるものではない。最終的な決断を下しプロジェクトを進めていくのは読者自身である。

幸い、たとえ決断を下したとしても、永遠に不変のものではない。アーキテクチャのリファクタリングを行うのは困難で、全体のコストがどのくらいになるかもわからないが、決して不可能なことではない。現時点で読者に提供できる唯一のアドバイスは、たとえエクストリームプログラミング[Beck XP]に関する話が好きではなくても、3つの技術上の実践—

—常時結合[Fowler CI]、テスト駆動開発[Beck TDD]、およびリファクタリング[Fowler Refactoring]——については納得いくまで検討すべきであるということだ。決して万能の解決策ではないが、考え直しを迫られた局面においては大いに役立つものである。そして、よほど運が良いか、私が今まで会ったこともないほどの才能の持ち主でない限り、いずれは考え直しを迫られることになる。

8.1 ドメインレイヤからの開始

プロセスの出発点は、どのドメインロジック手法で行くかを決定することである。有力候補は3つあり、**トランザクションスクリプト**、**テーブルモジュール**、**ドメインモデル**である。

第2章でも述べたように、これら3つのパターンからどれを用いるか決めるのに最も強い影響を与えるのは、ドメインロジックの複雑性である。複雑性の量も質も、現時点ではどのような尺度を使用しても測り難い。しかし、決定に影響を及ぼす要因はほかにもある。特にデータベースとの複雑な接続の難しさが挙げられよう。

3つのパターンの中で最もシンプルなのは、**トランザクションスクリプト**である。これは、依然として多くの人が慣れ親しんでいる手続き型モデルといえる。各システムトランザクションは、理解しやすいスクリプト内に正しくカプセル化されていて、リレーショナルデータベース上での構築も簡単に行える。このパターンの最大の欠点は、複雑なビジネスロジックをうまく扱えない点にある。特に、重複コードが起こりやすいという問題がある。つまり、価格を表示する基本的な構造にショッピングカートがついた程度のシンプルなカタログアプリケーションであれば、**トランザクションスクリプト**でも十分ニーズを満たすが、ロジックがさらに複雑になっていくと難易度は飛躍的に増大する。

対極に位置するのが**ドメインモデル**である。私のようなガチガチのオブジェクト主義者は、これ以外の方法でアプリケーションを設計することはない。アプリケーションが**トランザクションスクリプト**で書けてしまうほどシンプルなものであるなら、悩む必要はない。また、私の経験では、本当に複雑なドメインロジックの扱いに関しては、優れた**ドメインモデル**の右に出るものはない。**ドメインモデル**での作業に慣れてくれば、シンプルな問題にも簡単に対処できるようになるというものだ。

しかし、**ドメインモデル**にも欠点はある。中でも最も厄介なのが、その使い方を覚えるのがとても難しいという点だろう。オブジェクト主義者は、オブジェクトのことをよく理解していない人を見下す傾向が強いが、**ドメインモデル**を適切に設計するには相応のスキルが必要であり、これが不足していると悲惨な結果になるのはまちがいない。**ドメインモデル**が抱える2つ目のハードルは、リレーショナルデータベースとの接続が難しいということである。もちろん熱狂的なオブジェクト主義者であれば、この問題もオブジェクトデータベースを巧

みに操って難なく処理する。しかし主に非技術的な理由から、オブジェクトデータベースをエンタープライズアプリケーションで使うのは有効な選択ではない。結果的にリレーショナルデータベース接続が繁雑になるからだ。認めざるを得ないことだが、オブジェクトモデルとリレーショナルモデルはうまく適合しない。その結果が、ここで紹介するO/Rマッピングパターンの大半の複雑さなのである。

テーブルモジュールは、上記2つの対極にあるパターンのちょうど中間に位置する。ドメインロジックの扱いに関しては、明らかに**トランザクションスクリプト**よりは優れている。また、**ドメインモデル**のように複雑なドメインロジックを扱うことはできないが、リレーショナルデータベースやその他多くのものともうまく適合する。すべてを見通せる**レコードセット**の周囲に多くのツールが揃っている.NETのような環境がある場合、**テーブルモジュール**はうまく機能する。リレーショナルデータベースの力を活かしつつ、ドメインロジックを合理的に分離することである。

このとき使用するツールもアーキテクチャに影響する。アーキテクチャを基準にツールを選択できる場合もあり、理論的にはそうするべきだが、アーキテクチャの方をツールに合わせなくてはならないことの方が多い。上記3つのパターンの中でも、適合するツールを持っている場合は、**テーブルモジュール**が群を抜いている。特に、プラットフォームそのものが**レコードセット**との相性が良い.NET環境では最適な選択肢となる。

第2章のドメインロジックに関する説明を読んだ方であればすでに理解されていると思うが、最も重要な決定事項なので、繰り返しておく価値はあるだろう。ここからは1つ下の階層のデータベースレイヤに進むことになるが、これから先の決定はドメインレイヤで何を選択するかによって変わってくる。

8.2 | データソースレイヤに進む

ドメインレイヤを選択したら、次にデータソースに接続する方法を考える。ここから先の決定は、選択したドメインレイヤによって異なるので選択肢ごとに分けて紹介する。

8.2.1 | トランザクションスクリプト用のデータソース

最もシンプルな**トランザクションスクリプト**であっても独自のデータベースロジックが含まれている。しかし、どんなシンプルなものであっても、私はこれを避ける。データベースを分離すると、2つの範囲に区切られて境界線がわかりやすくなるので、私は最もシンプルと思われるアプリケーションでも分離する。選択できるデータベースパターンには、**行データゲートウェイ**と**テーブルデータゲートウェイ**の2つがある。

どちらを選択するかは、実装されているプラットフォームとアプリケーションが最終的に配置される場所によって決まる。**行データゲートウェイ**では、各レコードは明示的なインタフェースを持つオブジェクトに読み込まれる。**テーブルデータゲートウェイ**の場合は、データを取得するために必ずしもすべてのアクセッサーのコードを必要としないので、記述するコードは少なくて済むが、最終的にはるかに暗黙的なインタフェースを多用することになる。そしてこのインタフェースは、整然と並べられたマッピングにすぎないレコードセット構造へのアクセスに依存する。

ただし、最終的な決定は、使用しているプラットフォームに依存する。**レコードセット**と適合するツール、特に、UI ツールやトランザクション可能な切断されたレコードセットなどが数多く提供されているプラットフォームを使っている場合は、**テーブルデータゲートウェイ**が最有力となる。

通常、コンテキスト内では、その他の O/R マッピングパターンは一切必要としない。メモリ内構造がデータベース構造と適切にマッピングされるので、構造的マッピングに関して留意する必要はほとんどない。**ユニットオブワーク**の検討もよいが、更新管理はスクリプト内で行うほうが簡単である。並行性に関してはほとんど心配する必要はない。それは、スクリプトがシステムトランザクションとほぼ同等のものとなることが多いからである。したがって、スクリプト全体を 1 つのトランザクション内にラップする。一般的な例外として、ある要求がデータを編集用に抽出し、次の要求が変更を保存しようとするケースがあるが、この場合**軽オフラインロック**が常に最適な選択肢となる。実装を簡素化できるだけでなく、ユーザが予測する通りの結果が得られ、さまざまなものがロックされたままセッションがハングしてしまうという問題も回避できる。

8.2.2 | データソース「テーブルモジュール」

テーブルモジュールを選択する最大の理由は、優れた**レコードセット**フレームワークの存在にある。このケースでは、**レコードセット**と適合するデータベースマッピングパターンが必要なので、**テーブルデータゲートウェイ**を選択する以外ない。これら 2 つのパターンは、まるで一体型のようにスムーズに適合する。

このパターンでは、データソース側には追加すべきものは他に何もない。最適なケースでは、**レコードセット**にある種の並行性制御メカニズムが組み込まれていて、事実上**ユニットオブワーク**になることもあり、余計なストレスを減らせるようになる。

8.2.3 | ドメインモデル用のデータソース

面白くなるのはここからだ。いろいろな点から見て、**ドメインモデル**の最大の弱点はデー

タベースとの接続が複雑なことである。複雑さの度合いは、パターンの複雑さによって違ってくる。

データベースに隣接する20～30のクラスで構成されているような、比較的シンプルな**ドメインモデル**の場合、**アクティブレコード**が適切である。もう少し切り離したい場合は、**テーブルデータゲートウェイ**か**行データゲートウェイ**のいずれかを使うこともできる。いずれの方法でも、切り離すかどうかは大きな問題ではない。

もう少し複雑になった場合は、**データマッパー**についても検討する必要があるだろう。この手法は、**ドメインモデル**をできる限り他のレイヤに依存しないものにすることを約束する。一方で、**データマッパー**は最も実装が複雑なものの1つでもある。優秀なチームを抱えているか、マッピングを簡素化する何らかの知識がない限り、できるだけマッピングツールを入手することを勧める。

データマッパーを選択した時点で、O/Rマッピングセクションのほとんどのパターンが関与する。特に、並行性制御の中心的役割を果たす**ユニットオブワーク**を強く推奨したい。

8.3 プレゼンテーションレイヤ

多くの意味で、プレゼンテーションは下層レイヤの選択に依存する度合いが小さい。最初に決めるべきことは、リッチクライアントインタフェースとHTMLブラウザーインタフェースのどちらを使うかである。リッチクライアントは、見栄えがよいUIを提供するが、クライアントの制御と導入にそれなりの労力が必要である。私は、リッチクライアントである必要がないのであれば、HTMLブラウザーを使う。通常、リッチクライアントのプログラミングにはより多くの労力が必要だが、これは技術的に複雑だからと言うよりは、より洗練されたものにするためである。

本書ではリッチクライアントを使ったパターンを1つも紹介していない。したがって、このパターンを使う場合、本書はあまり役立たないだろう。

HTMLインタフェースを選択した場合は、まず、アプリケーションをどのような構造にするかを決める必要がある。ここでは、**モデルビューコントローラ**を設計の基盤として使うことを勧める。そうすれば、決定事項はあと2つ、コントローラとビューだけになる。

これらの選択は、使用しているツールによって決まることがある。Visual Studioを使っている場合は、**ページコントローラ**と**テンプレートビュー**を使うのが最も簡単な方法である。Javaを使っている場合は、Webフレームワークの選択も考慮する必要がある。現時点で最も人気があるのはStrutsだが、この場合は**フロントコントローラ**と**テンプレートビュー**になる。

自由に選択するなら、サイトがドキュメント指向で特に静的ページと動的ページが混在する場合は、**ページコントローラ**を推奨したい。一方、より複雑な遷移やUIを含む場合は、

フロントコントローラということになる。

ビューに関しては、**テンプレートビュー**と**トランスフォームビュー**のどちらを選ぶかは、開発チームがサーバページと XSLT のどちらを使ってプログラミングしているかで異なる。現時点では、**テンプレートビュー**の方が優勢である。ただし私は**トランスフォームビュー**が持つテスト機能の方が好きである。1つの共通サイトを複数のルック＆フィールで表示させたい場合は、**ツーステップビュー**も検討するべきである。

下位レイヤとの通信方法は、それらがどの種類のレイヤか、また、常に同じプロセス内にいるかどうかによって異なる。私は、できる限りすべてを1つのプロセス内で実行するようにしている。こうしておけば、プロセス間の遅い呼び出しについて心配する必要がない。これができない場合は、ドメインレイヤを**リモートファサード**でラップし、**データ変換オブジェクト**を使って Web サーバと通信するべきである。

8.4 | 技術上のアドバイス

本書では、多くの異なるプラットフォーム間でプロジェクトを進める場合の共通した経験を述べるように心がけている。Forte、CORBA、Smalltalk などでの経験は、Java や.NET における開発にも十分活かせるものである。ここで Java と.NET 環境にこだわるのは、これらが将来にかけてエンタープライズアプリケーション開発の共通プラットフォームになる可能性が高いからである（ただし、個人的には Python や Ruby などの動的に型付けされるスクリプティング言語を検討したいし、競争も必要だと思う）。

ここからは、今まで述べてきたアドバイスを2つのプラットフォームに適用してみよう。しかし期間限定のものになってしまう恐れがある。技術はこれらのパターンよりもはるかに速く変化しているので、本書が2002年初頭、景気回復の兆しが見え始めたと言われた頃に書かれたものであることを頭に入れておいてほしい。

8.4.1 | Java と J2EE

現在 Java の世界では、EJB（Enterprise Java Beans）にどれほどの価値があるかということが大きな論争の的となっている。The Who が解散コンサートを開いたときのように、EJB 2.0 仕様はいくつもの改訂を経てやっと登場するに至った。しかし、EJB ベンダーが何を言うかはさておき、優れた J2EE アプリケーションを構築するのに必ずしも EJB が必要なわけではない。POJO（plain old Java objects: 標準の古い Java オブジェクト）と JDBC だけでも十分機能するものを構築できる。

J2EE での設計の選択肢は、使用しているパターンによって違い、ここでもドメインロジックがその中心を担っている。

何らかの形態の**ゲートウェイ**の上で**トランザクションスクリプト**を使っている場合、現時点でのEJBにおける一般的な手法は、セッションBeanを**トランザクションスクリプト**として、エンティティBeanを**行データゲートウェイ**として使うというものである。ドメインロジックが適度なサイズの場合は、適切なアーキテクチャである。ただし、Beanを使った手法の1つの問題点は、必要なくなったと判断しても簡単にはEJBサーバを切り離せないという点にあり、その結果、余計なライセンス使用料を取られてしまうこともある。EJBを使わない手法としては、**行データゲートウェイ**か**テーブルデータゲートウェイ**のいずれかの上でPOJOを**トランザクションスクリプト**として使う方法がある。JDBC 2.0の行セットの容量が大きい場合、**レコードセット**として使えるので、**テーブルデータゲートウェイ**ということになる。

ドメインモデルを使っている場合、エンティティBeanを使うのが現行では定説となっている。使用している**ドメインモデル**が比較的シンプルでデータベースとも相性が良い場合、十分納得の行く手法であり、エンティティBeanは、**アクティブレコード**になる。ここでも、エンティティBeanをセッションBeanでラップして、**リモートファサード**として機能させるという手法は有効である（CMP（Container Managed Persistance）を**データマッパー**として考えるという方法もある）。ただし、**ドメインモデル**がより複雑な場合は、EJB構造とは完全に独立させ、ドメインロジックの記述、実行およびテストを、EJBコンテナと切り離して行えるようにした方がよい。私なら、このモデルではPOJOを**ドメインモデル**として使い、セッションBeanでラップして**リモートファサード**として機能させるだろう。また、EJBを使わない場合は、アプリケーション全体をWebサーバ上で実行させ、プレゼンテーションとドメイン間でリモートコールが発生しないようにする。また、POJO**ドメインモデル**を使う場合は、POJOを**データマッパー**として使う（O/Rマッピングツールを使うか、何かしら自分で用意する）。

いずれのコンテキストにおいても、エンティティBeanを使う場合は、リモートインタフェースを採用するべきではない。私にとっては、エンティティBeanにリモートインタフェースを与えること自体理解できないことである。エンティティBeanは、**ドメインモデル**として、または**行データゲートウェイ**として使われるが、いずれのケースでも細かい粒度のインタフェースが必要になる。リモートインタフェースは必ず粗い粒度のものである必要があるので、エンティティBeanはローカルだけで使うべきである（例外として、[Alur et al.]の **Composite Entity** パターンがあるが、エンティティBeanの使用方法が異なっていて、あまり有効なものとは思えない）。

現時点では、**テーブルモジュール**はJavaの世界ではあまり一般的ではない。JDBC行セット用のツールがもっと増えれば面白いと思う。そのときは、パターンも有効な手法となるはずである。このケースではPOJOが最適な手法だが、**テーブルモジュール**をセッションBeanでラップして**リモートファサード**として使い、**レコードセット**を返させるという方法もある。

8.4.2 | .NET

.NET、Visual Studio など、Microsoft のアプリケーション開発環境の歴史を辿ってみると、大半が**テーブルモジュール**をパターンとして使っていることに気付く。このことからも言えることだが、オブジェクト主義者は Microsoft 愛好者がオブジェクトを使わないと結論付けてしまいがちである。しかし、**テーブルモジュール**には、**トランザクションスクリプト**と**ドメインモデル**の妥協点のようなものが含まれ、偏在するデータセットを**レコードセット**として活かせる優秀なツールが数多く含まれている。

つまり、このプラットフォームでは**テーブルモジュール**がデフォルトの選択肢となる。ごくシンプルなケースを除き、**トランザクションスクリプト**を使うことには意味を見出せない。たとえシンプルなケースでもデータセットに対して作用し、データセットを返すようにするべきである。

これは、決して**ドメインモデル**が使えないと言っているわけではない。事実、.NET でもその他のオブジェクト指向環境同様、簡単に**ドメインモデル**を構築することができる。ただし、**テーブルモジュール**のときのように役立つツールが少ないので、**ドメインモデル**へのシフトが必要だと感じる以前に、**ドメインモデル**が他の環境よりもよほど複雑でなければ行わない。

.NET といえば Web サービスというような考え方が浸透しているようだが、私だったらアプリケーション内で Web サービスは使わない。Java のときと同じように、アプリケーション同士が統合できるプレゼンテーションとして使う。.NET アプリケーションでは、Web サーバとドメインロジックを切り離す理由も特にないので、**リモートファサード**はここではあまり有効なものとはならない。

8.4.3 | ストアドプロシージャ

ストアドプロシージャについてはさまざまな議論がある。データベースと同じプロセス内で実行されるので、足を引っ張るリモートコールが減ることから、最も高速な手段であると言われる。ただし、ほとんどのストアドプロシージャ環境では、実際に使用するストアドプロシージャ用に優れた構造メカニズムが提供されることはなく、特定のデータベースベンダーに依存せざるをえなくなることが多い（これらの問題をうまく回避しているのが、データベースプロセス内で Java アプリケーションを実行できるようにしている Oracle の手法である。これはドメインロジックレイヤ全体をデータベース内に置くのと同じことになる。特定ベンダーに依存せざるをえないという点では同じだが、少なくとも移植コストは削減できる）。

モジュール性と移植性からみて、ほとんどの人はビジネスロジックではストアドプロシージャを使うことは避けている。私も、パフォーマンス面で大幅な改善でも見られない限り、ストアドプロシージャを使用しないようにしているが、改善はよく行われているのも事実で

ある。パフォーマンス面での改善があれば、喜んでメソッドをドメインレイヤからストアドプロシージャに移す。ただし、明らかにパフォーマンスに問題のある部分だけに抑え、アーキテクチャそのものではなく、最適化手順として扱う（ストアドプロシージャの扱いに関しては、[Nilsson]がより広い視野から掘り下げた議論を展開している）。

ストアドプロシージャは、**テーブルデータゲートウェイ**の中でデータベースへのアクセスを制御するものとして使うのが一般的な使用法であるが、これを扱う方法に関して私は特に意見はない。今まで見てきたものの中にも、どちらにするかを決定する理由は見つからない。いずれの場合でも、同じパターンでのデータベースアクセスは切り離すべきだろう。ストアドプロシージャを介したものでも、より一般的なSQLを介したものであってもである。

8.4.4 | Webサービス

この本の執筆時点での識者の一般的見解は、今後Webサービスを再利用できるようになるので、システム統合ビジネスはいずれ成り立たなくなるということだ。しかし、これらのパターンに対してWebサービスの役割は決して大きいものではない。Webサービスとはアプリケーションを統合したものであり、アプリケーションの構築そのものではないからである。1つのアプリケーションを複数のWebサービスに分解し、互いに交信し合うような状態にすることは、本当に必要な場合を除いてするべきではない。アプリケーションを構築し、それぞれの部分をWebサービスとして公開するべきであり、Webサービスを**リモートファサード**として扱う。Webサービスの構築は簡単だと言われているが、それらの言葉に惑わされて「分散オブジェクト設計の第一法則」（第7章）を忘れないように気をつけてほしい。

これまで公開された例のほとんどはXML RPCのような同期Webサービスを使っているが、私には非同期でメッセージベースの方が好ましい。本書ではそのようなパターンは紹介していないが（本書は本題だけですでにかなりのボリュームになっているので）、数年のうちに非同期メッセージング用のパターンが紹介されることを期待したい。[訳注]

8.5 | その他のレイヤ化スキーム

ここでは3つの主なレイヤを中心に話を進めているが、これ以外のレイヤ化方法も存在する。アーキテクチャに関する他の書籍にもレイヤ化スキームが紹介されていて、それぞれに価値がある。できればこれらのスキームにも目を通し、ここで紹介しているものと比較してみてほしい。読者が実際に携わっているアプリケーションには、そちらの方が適切であるこ

【訳注】このパターンに関しては、『Enterprise Integration Patterns』を参照のこと。

ともある。

その1番手は、私がBrownモデルと呼んでいるもので、[Brown et al.]で紹介されている（表8.1参照）。このモデルでは、プレゼンテーション、コントローラ／メディエータ、ドメイン、データマッピング、データソースの5つのレイヤを使っている。基本的に、3つの基本レイヤの間に中間レイヤが配置された構成である。コントローラ／メディエータは、プレゼンテーションとドメインの間を、データマッピングは、ドメインとデータソースレイヤの間を取り次ぐものである。

中間レイヤは便利な場合もあるが、常に便利というわけではないので、本書ではパターンとして紹介している。**アプリケーションコントローラ**はプレゼンテーションとドメインの間を、そして**データマッパー**はデータソースとドメインの間をそれぞれ取り次ぐものである。本書では、**アプリケーションコントローラ**はプレゼンテーションの章（第14章）で、**データマッパー**はデータソースの章（第10章）でそれぞれ紹介している。

便利な場合が多いが常に便利とは限らない中間レイヤは、設計におけるオプション的なものに思える。つまり、3つの基本レイヤを軸にし、どれかが複雑になり過ぎたときに中間レイヤを追加して機能を分散させるという手法である。

J2EE用の優れたレイヤ化スキームは、CoreJ2EEパターンでも紹介されている[Alur et al.]（表8.2参照）。ここでは、クライアント、プレゼンテーション、ビジネス、統合、リソースの各レイヤが使われている。この中でビジネスレイヤと統合レイヤは、シンプルに一致するものがある。リソースレイヤは、統合レイヤが接続した先の外部サービスによって構成されている。一番大きな違いは、プレゼンテーションレイヤが、クライアント側で実行する部分（クライアント）と、サーバ側で実行する部分（プレゼンテーション）に分かれている点にある。これは便利な場合もあるが、常に必要であるとは限らない。

表8.1 —— Brownレイヤ

Brown	Fowler
プレゼンテーション	プレゼンテーション
コントローラ／メディエータ	プレゼンテーション（**アプリケーションコントローラ**）
ドメイン	ドメイン
データマッピング	データソース（**データマッパー**）
データソース	データソース

表 8.2 —— Core J2EE レイヤ

Core	J2EE Fowler
クライアント	クライアント側で実行されるプレゼンテーション （例：リッチクライアントシステム）
プレゼンテーション	サーバ側で実行されるプレゼンテーション （例： HTTP ハンドラ、サーバページ）
ビジネス	ドメイン
統合	データソース
リソース	データソースが通信する外部リソース

Microsoft の DNA 設計者[Kirtland]は、プレゼンテーション、ビジネス、データアクセスという 3 つのレイヤを定義しているが、これらはほぼそのまま本書で紹介している 3 つのレイヤと一致する（表 8.3 参照）。最も大きな違いは、データアクセスレイヤからのデータの渡され方にある。Microsoft DNA では、データアクセスレイヤから発行された SQL クエリーの結果となるレコードセット上ですべてのレイヤが作動するようになっている。その結果、ビジネスレイヤとプレゼンテーションレイヤの両方がデータベースについて知っているという結合が発生している。

私の見方では、DNA におけるレコードセットはレイヤ間での**データ変換オブジェクト**のような役割を果たしているように見える。ビジネスレイヤは、プレゼンテーションレイヤ上に移動する途中でレコードセットを修正できるだけでなく、稀なケースだが、新たなものを作成することもできる。この通信形態はいろいろな意味で扱いにくいが、プレゼンテーションがデータを認識している GUI 制御を利用できるという大きな利点もある。このデータには、ビジネスレイヤが修正したものも含まれるのである。

このケースでは、ドメインレイヤは**テーブルモジュール**形式の構造を持ち、データソースレイヤは**テーブルデータゲートウェイ**を使う。

[Marinescu]も 5 つのレイヤを使っている（表 8.4 参照）。ここでは、プレゼンテーションが 2 つのレイヤに分割されているが、**アプリケーションコントローラ**の分離を反映している。ドメインも**ドメインモデル**上で構築された**サービスレイヤ**で分割されているが、ドメインレイヤを 2 つに分割するという一般的なアイデアを反映したものである。これは、**ドメインモデル**のように EJB の制限から強いられる一般的な手法である（P125 参照）。

表 8.3 —— Microsoft DNA レイヤ

Microsoft DNA	Fowler
プレゼンテーション	プレゼンテーション
ビジネス	ドメイン
データアクセス	データソース

表 8.4 —— Marinescu レイヤ

Marinescu	Fowler
プレゼンテーション	プレゼンテーション
アプリケーション	プレゼンテーション（アプリケーションコントローラ）
サービス	ドメイン（サービスレイヤ）
ドメイン	ドメイン（ドメインモデル）
永続	データソース

サービスレイヤをドメインレイヤから切り離すという考え方は、ワークフローロジックを純粋なドメインロジックから分離するという考えに基づいている。サービスレイヤには特定の場合だけ使用するロジックや、メッセージングなど他のインフラと通信するためのロジックが含まれている。サービスとドメインを別々のレイヤにするかどうかについては、いろいろな意見がある。私は必須のものというより、場合によっては便利なものと考えているが、私が尊敬する設計者はこの意見に賛同していない。

[Nilsson]は最も複雑なレイヤ化スキームの１つを使っている（表8.5 参照）。Nilsson はストアドプロシージャを多用し、パフォーマンス目的でその中にドメインロジックを入れ込んでいるので、このスキームへのマッピングはかなり複雑なものになる。ストアドプロシージャ内にドメインロジックを入れると、アプリケーションの維持が大幅に困難になるので、私にとってはこの手法は難しく感じられる。ただし、ある特定の状況においては最適化手法として有効である。Nilsson のストアドプロシージャレイヤには、データソースとドメインロジックの両方が含まれている。

また、[Marinescu]同様、Nilsson もドメインロジック用にアプリケーションとドメインの別々のレイヤを使っている。小規模なシステムではドメインレイヤを省略するように指示しているが、これは**ドメインモデル**の価値が小規模システムでは低くなるという私の考え方に共通するものがある。

表 8.5 —— Nilsson レイヤ

Nilsson	Fowler
コンシューマ	プレゼンテーション
コンシューマヘルパー	プレゼンテーション（アプリケーションコントローラ）
アプリケーション	ドメイン（サービスレイヤ）
ドメイン	ドメイン（ドメインモデル）
永続アクセス	データソース
パブリックストアドプロシージャ	データソース（ドメインを含む場合もある）
プライベートストアドプロシージャ	データソース（ドメインを含む場合もある）

第2部

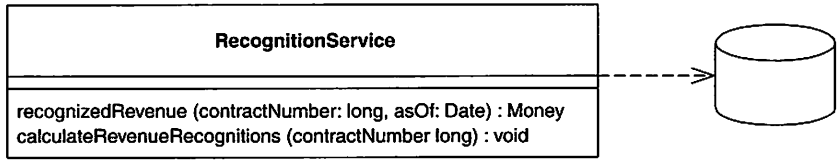
パターン

第9章

ドメインロジックパターン

9.1 | トランザクションスクリプト

ビジネスロジックを一連の手続きで構築して、その各手順でプレゼンテーションからの1つの要求を処理する。



ビジネスアプリケーションの大半は、一連のトランザクションとして考えることができる。トランザクションには、ある特定の方法で体系化した情報を表示するものや変更するものがある。クライアントシステムとサーバシステム間の相互作用には、一定量のロジックが含まれている。ロジックには、データベース内の情報を表示するシンプルなものから、いくつもの妥当性の確認と計算の手順を含むものもある。

トランザクションスクリプトは、主にロジックを1つの手続きにまとめてデータベースを直接呼び出すか、薄いデータベースラッパーを介して呼び出す。各トランザクションには独自の**トランザクションスクリプト**があるが、共通のサブタスクはサブ手続きに分割できる。

9.1.1 | 動作方法

トランザクションスクリプトでは、ドメインロジックはシステムで実行するトランザクションで主に構築される。たとえばホテルの部屋を予約するというニーズがある場合、ホテルの部屋を予約する手続きには、空室をチェックするロジック、宿泊費を計算するロジック、

そしてデータベースを更新するロジックが含まれることになる。

シンプルな場合には、ロジックをどのように体系化するのかまでは必要ない。もちろん、どのようなプログラムでも、ある程度合理的なモジュールとしてコードを構成する必要がある。トランザクションが特に複雑なものでなければ、プログラムのコード化は決して難しいことではないはずだ。プログラムのコード化では、他のトランザクションの動作を気にする必要がないというメリットがある。入力データを取得し、データベースに問い合わせて変更を行い、さらに結果をデータベースに保存するというのが、実行するタスクになる。

また、**トランザクションスクリプト**をどこに置くかは、レイヤをどのように体系化するかによって異なる。サーバページ、CGI スクリプト、または分散セッションオブジェクトのいずれの中にでも置くことができる。私はできる限り**トランザクションスクリプト**を分離させることを好んでいる。少なくとも**トランザクションスクリプト**を個別のサブルーチン内に置き、プレゼンテーションやデータソースを処理する**トランザクションスクリプト**とは異なるクラスに置くことにしている。さらに、**トランザクションスクリプト**からプレゼンテーションロジックを呼び出さないようにする。これでコードの修正も、**トランザクションスクリプト**のテストも簡単に行えるようになる。

トランザクションスクリプトを複数のクラスに体系化する方法は2つある。最も一般的な方法は、複数の**トランザクションスクリプト**を1つのクラスに入れ、各クラスが関連する**トランザクションスクリプト**の対象エリアを定義する方法である。この方法は最も簡単で一般的な手法である。もう1つの方法は、**トランザクションスクリプト**ごとに独自のクラスを持たせ（図9.1）、「コマンドパターン」[Gang of Four]を使うというものだ。この場合、**トランザクションスクリプト**ロジックに適合する実行メソッドを指定するために、コマンドのスーパータイプを定義することになる。この手法のメリットは、スクリプトのインスタンスを実行時にオブジェクトとして扱える点であるが、**トランザクションスクリプト**を使ってドメインロジックを体系化するようなシステムでは、このメリットを活かす必要性はほとんどない。もちろん多くの言語では、クラスを完全に無視してグローバル関数だけを使うこともできる。しかし、新たなオブジェクトをインスタンス化することでスレッドの問題が解決できる場合もある。データの分離が簡単になるからだ。

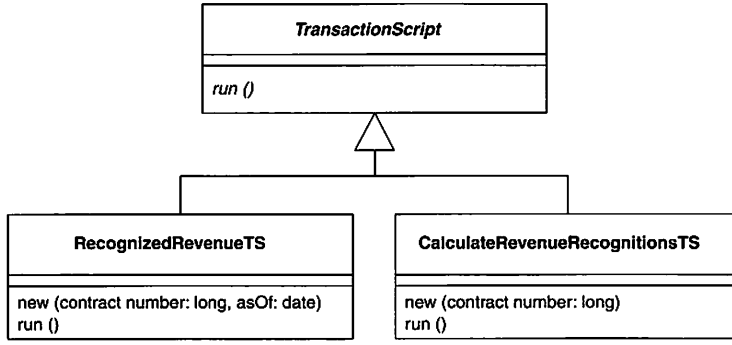


図 9.1 ――トランザクションスクリプトのコマンドの使用

私がトランザクションスクリプトという用語を使っているのは、ほとんどの場合、データベーストランザクションごとに 1 つのトランザクションスクリプトを使用しているからである。これは必ずしも 100 %確実なルールではないが、それに近いものであることは間違いない。

9.1.2 | 使用するタイミング

トランザクションスクリプトの素晴らしい点はシンプルさにある。このようなロジックの体系化は、ごくわずかなロジックを使ったアプリケーションでは一般的であり、パフォーマンスの面でもその解釈の面でも、オーバーヘッドはほとんどない。

しかし、ビジネスロジックが複雑になるにつれて、優れた設計を維持するのは大変になる。特に注意すべきことは、トランザクション間での重複の問題である。

1 つのトランザクションを処理することが重要事項となるので、共通するどのコードも重複しやすい状況になる。

きめ細かにファクタリングすると問題の多くは軽減できるが、より複雑なビジネスドメインでは、ドメインモデルの構築が必要となる。ドメインモデルでは、豊富なオプションを使ってコードを形成できるので、読みやすさが増すと同時に重複も減らすことができる。

パターンを切り替えるレベルを定めるのは容易ではない。特に特定のパターンの場合はなおさらだ。トランザクションスクリプト設計からドメインモデル設計にリファクタリングすることはできるが、実際には思っている以上に難しい変更となる。したがって、初期段階での対応が最良の対処方法である。

しかし、頑固なオブジェクト指向主義者であっても、トランザクションスクリプトをすべて除外するべきではない。シンプルな問題は数多く存在するし、シンプルな解決策の方が素早く対策を講じられるからである。

9.1.3 | RevenueRecognition の問題

トランザクションスクリプトを含め、ドメインロジックを説明するパターンでは、例として同じ問題を扱うことにする。問題のステートメントを繰り返し入力しなくてすむように、以降のパターンでは省略してある。

ビジネスシステムでよく見られる問題に、RevenueRecognition（収益認識）の問題がある。帳簿に収入を計上する際に起こる問題である。コーヒーを1杯売るのなら、簡単なことである。コーヒーを渡し代金を得る。そしてすぐに帳簿に計上する。しかし、多くの場合はもっと複雑である。たとえば、開発者である私のもとに、顧客がその年の依頼料を支払いに来たとする。多額の報酬をこの時点で受領しても、実際のサービスは1年を通じて行われるのですぐに帳簿に計上できない場合がある。1つの手法として、支払われた額の1/12を各月の報酬として計上するという方法がある。こうすれば、本書を刊行したときに私のプログラミングスキルが大したことないのがわかって、顧客の方からContract（契約）を取り下げたときにも対処することができる。

RevenueRecognitionのルールは、多種多様でしかも定石がない。法規や業界基準および会社のポリシーなどによって定められている場合がある。収益の追跡は、かなり難しく複雑な問題となる。

ここで複雑性について掘り下げていくつもりはない。そのかわり、ワープロ、データベース、それに表計算という3種類のProduct（製品）を販売している会社を想定してみることにする。その会社のルールでは、ワープロはContract（契約）にサインした時点で、すべての収益を直ちに計上することができる。表計算の場合は、その時点で1/3、60日で1/3、90日で1/3、データベースの場合はその時点で1/3、30日で1/3、60日で1/3をそれぞれ計上できるようになっている。これらのルールはあくまでも私の勝手な想定で、どこかで使われているものをベースにしているわけではない。実際のルールはもっと合理的なはずである。

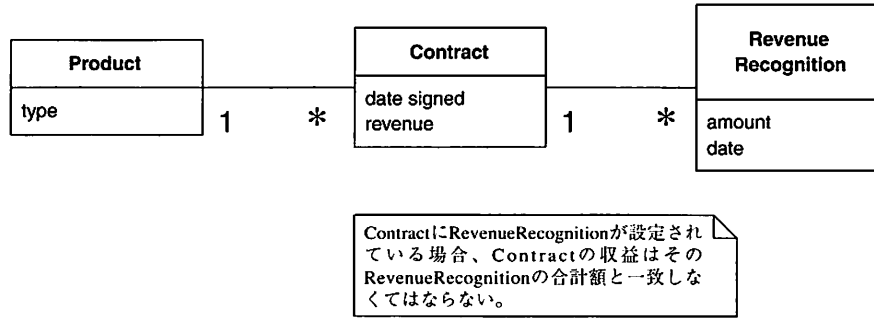


図 9.2 ——簡略化された RevenueRecognition 用の概念モデル。Contract ごとに複数の収益認識を持ち、さまざまな収益を認識する時間を示している。

9.1.4 ｜例：RevenueRecognition（Java）

この例では、2つのトランザクションスクリプトを使用する：1つは、Contractのための RevenueRecognitionを計算するためのものであり、もう1つは、Contractによる収益の何割が特定日までに認識できるかを示すものである。データベース構造には、Product用、Contract用、RevenueRecognition用の3つのテーブルがある。

```
CREATE TABLE products (ID int primary key, name varchar, type varchar)
CREATE TABLE contracts (ID int primary key, product int,
   revenue decimal, dateSigned date)
CREATE TABLE revenueRecognitions (contract int, amount decimal,
   recognizedOn date, PRIMARY KEY (contract, recognizedOn))
```

1つ目のスクリプトは、特定日までに認識できる額を算出するものだ。これは2つの段階で行うことができる。まず、RevenueRecognitionsテーブル内の適切な行を選択し、次に額の合計を算出する。

多くのトランザクションスクリプト設計では、データベース上で直接動作するスクリプトを使用し、SQLコードを手続きに入れている。ここでは、簡単なテーブルデータゲートウェイを使ってSQLクエリーをラップしている。また、例はとてもシンプルであるため、テーブルごとに1つずつゲートウェイを使用するのではなく、全体で1つのゲートウェイを使用している。ゲートウェイ上に適切なfindメソッドを定義することができる。

```
class Gateway...

   public ResultSet findRecognitionsFor(long contractID,
      MfDate asof) throws SQLException{
      PreparedStatement stmt = db.prepareStatement
         (findRecognitionsStatement);
      stmt.setLong(1, contractID);
      stmt.setDate(2, asof.toSqlDate());
      ResultSet result = stmt.executeQuery();
      return result;
   }
   private static final String findRecognitionsStatement =
      "SELECT amount " +
      "  FROM revenueRecognitions " +
      "  WHERE contract = ? AND recognizedOn <= ?";
   private Connection db;
```

ここでスクリプトを使い、ゲートウェイから渡された結果セットを基に合計を計算する。

```
class RecognitionService...

   public Money recognizedRevenue(long contractNumber, MfDate asOf) {
      Money result = Money.dollars(0);
      try {
         ResultSet rs = db.findRecognitionsFor(contractNumber, asOf);
         while (rs.next()) {
            result = result.add(Money.dollars(rs.getBigDecimal
            ("amount")));
         }
         return result;
      } catch (SQLException e) {throw new ApplicationException (e);
      }
   }
```

このようなシンプルな計算であれば、メモリ上のスクリプトの代わりに、金額を合計する集合関数を使うSQL文の呼び出しを使うこともできる。

既存のContract（契約）におけるRevenueRecognition（収益認識）の計算でも、似たような分割を使っている。サービス上のスクリプトがビジネスロジックを実行しているのである。

```
class RecognitionService...

   public void calculateRevenueRecognitions(long contractNumber) {
      try {
         ResultSet contracts = db.findContract(contractNumber);
         contracts.next();
         Money totalRevenue = Money.dollars
            (contracts.getBigDecimal("revenue"));
         MfDate recognitionDate = new MfDate(contracts.getDate
            ("dateSigned"));
         String type = contracts.getString("type");
         if (type.equals("S")){
            Money[] allocation = totalRevenue.allocate(3);
            db.insertRecognition
               (contractNumber, allocation[0], recognitionDate);
            db.insertRecognition
               (contractNumber, allocation[1],
                  recognitionDate.addDays(60));
            db.insertRecognition
```

```
                (contractNumber, allocation[2],
                   recognitionDate.addDays(90));
            } else if (type.equals("W")){
               db.insertRecognition(contractNumber, totalRevenue,
                  recognitionDate);
            } else if (type.equals("D")) {
               Money[] allocation = totalRevenue.allocate(3);
               db.insertRecognition
                  (contractNumber, allocation[0], recognitionDate);
               db.insertRecognition
                  (contractNumber, allocation[1],
                     recognitionDate.addDays(30));
               db.insertRecognition
                  (contractNumber, allocation[2],
                     recognitionDate.addDays(60));
            }
        } catch (SQLException e) {throw new ApplicationException (e);
        }
    }
```

ここでは、マネーを使って割り当てを実行していることに注目してほしい。金額を3つに分割すると、額が合わなくなる可能性が高くなる。

テーブルデータゲートウェイではSQLもサポートしている。まずは、Contract用のfindテーブルについてである。

```
class Gateway...

    public ResultSet findContract (long contractID)
        throws SQLException{
        PreparedStatement stmt = db.prepareStatement
            (findContractStatement);
        stmt.setLong(1, contractID);
        ResultSet result = stmt.executeQuery();
        return result;
    }
        private static final String findContractStatement =
            "SELECT * " +
            " FROM contracts c, products p " +
            " WHERE ID = ? AND c.product = p.ID";
```

次に、挿入用のラッパーについてである。

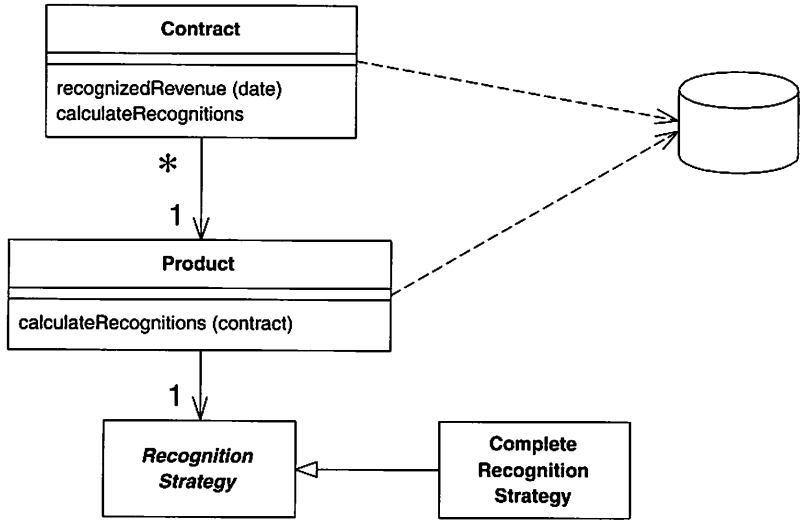
```
class Gateway...

   public void insertRecognition (long contractID, Money amount,
      MfDate asof) throws SQLException {
      PreparedStatement stmt = db.prepareStatement
         (insertRecognitionStatement);
      stmt.setLong(1, contractID);
      stmt.setBigDecimal(2, amount.amount());
      stmt.setDate(3, asof.toSqlDate()); stmt.executeUpdate();
   }
   private static final String insertRecognitionStatement =
      "INSERT INTO revenueRecognitions VALUES (?, ?, ?)";
```

Java システムでは、認識サービスは通常のクラスかセッション Bean となる。

これを**ドメインモデル**で紹介している例と比べたとき、読者から見ると、この例の方がはるかにシンプルなものだと思うはずである。すぐには想像がつかないかもしれないが、仮にルールがもっと複雑になるとどうなるのか。一般に使われる RevenueRecognition ルールは、実際にはもっと複雑で Product ごとに違いがあるだけでなく、日付によっても異なる(4 月 15 日以前の Contract の場合はあるルールが適応され、それ以降の場合は別のルールが適応されるなどである)。ルールが複雑になると、**トランザクションスクリプト**では、一貫した設計を維持できなくなってしまう。そのため、私のようなオブジェクト指向主義者は、このような状況で**ドメインモデル**を好んで使うのである。

9.2 | ドメインモデル

振る舞いとデータの両方を一体化させたドメインのオブジェクトモデル。



ビジネスロジックにおける最悪の場合とは、そのロジックがとても複雑になっている場合である。ルールやロジックは、多くの異なる事例やさまざまな傾向の振る舞いを記述しているが、このような複雑性に対処できるようにオブジェクトは設計されている。**ドメインモデル**は、相互に関連するオブジェクトが絡み合った関係を作成する。各オブジェクトは、企業のような大きなオブジェクトも、1行だけのOrder（注文）フォームのような小さなオブジェクトも、それぞれ意味がある個体を成す。

9.2.1 | 動作方法

アプリケーションでの**ドメインモデル**の使用は、対象となるビジネスエリアをモデル化したオブジェクトのレイヤ全体を挿入するということである。ビジネスにおけるデータをモデル化するオブジェクトもあれば、ビジネスで使用するルールを把握するオブジェクトもある。ほとんどの場合、データとプロセスを結合し、対象となるデータと関係しているプロセスをクラスタ化する。

オブジェクト指向ドメインモデルは、データベースモデルと似ているように見える場合がある。しかしここには大きな違いがある。**ドメインモデル**ではデータとプロセスが一体化され、複数値属性や複雑に絡み合う関係があり、さらに継承が使われるのである。

その結果、**ドメインモデル**には2つの形式があることがわかる。シンプルな**ドメインモデル**は、見た目はデータベース設計とよく似ていて、ほとんどがデータベーステーブルごとに1つのドメインオブジェクトを持つ形式になっている。一方、豊富な**ドメインモデル**は見た目もデータベース設計とは異なり、継承、ストラテジー、その他の[Gang of Four]パターンや、さらには相互に関連する小さなオブジェクトが複雑に絡み合う構造を持っている。豊富な**ドメインモデル**は、複雑なロジックを扱うのには適しているが、データベースにマッピングするのは難しい。シンプルな**ドメインモデル**では**アクティブレコード**を利用できるが、豊富な**ドメインモデル**では**データマッパー**が必要となる。

ビジネスの振る舞いは頻繁に変更されるものなので、レイヤを簡単に修正、構築、テストできるということは重要である。そのため、**ドメインモデル**とシステム内の他のレイヤとの結合は最小限に抑えることになる。多くのレイヤ化パターンでも示されているように、ドメインモデルとシステム内のその他の部分の依存性はできる限り少なくしておくべきである。

ドメインモデルは、いくつかの異なる領域で使用することができる。最もシンプルなスコープはシングルユーザアプリケーションである。アプリケーションでは、オブジェクトグラフ全体がファイルから読み取られ、メモリ内に読み込まれる。デスクトップアプリケーションであればこの方法も良いが、多層のISアプリケーションでは一般的ではない。単にオブジェクトの数が多すぎるからである。すべてのオブジェクトをメモリに入れると、メモリ消費量が多くなると同時に時間もとてもかかる。オブジェクト指向データベースが優れているところは、メモリとディスク間でオブジェクトを移動しているときに、オブジェクトをメモリに入れる作業を行っているかのような印象を与える点である。

オブジェクト指向データベースがない場合は、各自で実行する必要がある。セッションでは、関連するすべてのオブジェクトのオブジェクトグラフが使われる。このオブジェクトグラフは、確実にすべてのオブジェクトを含むわけではなく、またすべてのクラスを含むことも普通はない。したがって、一連のContractは、作業セット内のContractによって参照するProductだけを対象にしている。ContractとRevenueRecognitionオブジェクトに関する計算を実行する場合は、Productオブジェクトは対象にならない場合がある。何をメモリに入れるかはデータベースマッピングオブジェクトによって決まる。

サーバの呼び出しの間に同じオブジェクトグラフが必要な場合、サーバステートをどこかに保存する必要がある。この点についてはサーバステートの保存についての項目（第6章）で述べている。

ドメインロジックを使用する場合に気をつけることは、ドメインオブジェクトの拡大である。Orderを処理する画面を構築する際、Orderの振る舞いの中には、画面のためだけに必要なものを含んでしまうことに注意すべきである。Orderの中に入れてしまうと、Orderクラスが肥大化してしまうリスクを背負い込むことになる。特定の状況でしか使わないもので膨らんでしまうので、何らかの責任が一般的か否かを検討する。一般的な場合、その責任

はOrderクラスに置かれ、そうでない場合、責任は特定用途向けのクラスに置かれることになる。特定用途向けのクラスは、**トランザクションスクリプト**またはプレゼンテーション自体の場合がある。

特定の状況での振る舞いを別にすると、重複の問題が発生する可能性がある。Orderとは違う振る舞いは目に付きにくく、気付かずに重複させてしまうことがある。重複により、複雑性を増すだけでなく一貫性も失われる。しかし、肥大化は思ったほど頻繁には起こらない。起こっても、比較的簡単に見つけることができ、修復も難しいものではない。特定の状況での振る舞いを別にしないことを推奨する。通常のオブジェクトにすべての特定の状況での振る舞いを入れ、肥大化が問題となった場合は、それを修復すればよいのである。

Javaでの実装

J2EEでの**ドメインモデル**の開発に関しては、常に熱い議論が起こる。J2EEの解説書や教材の多くでは、エンティティBeanを使ってドメインモデルを開発するように推奨しているが、この手法には深刻な問題がある。少なくとも現行の（バージョン2.0）仕様は問題である。

エンティティBeanは、CMP（Container Managed Persistence：コンテナ管理の永続化機構）を使用する場合、大変便利である。エンティティBeanをCMPなしで使うのはほとんど意味がないと言ってよい。しかし、CMPはオブジェクトリレーショナルマッピングが制限された形態のものであり、豊富な**ドメインモデル**で必要なパターンの多くはサポートできないというのが現実である。

エンティティBeanは、再入可能ではない。つまり、あるエンティティBeanから別のオブジェクトを呼び出した場合、そのオブジェクトが最初のエンティティBeanを呼び出すことはできないということである。豊富な**ドメインモデル**では、この再入可能性をよく使うので、これがないと不利になる。再入可能な振る舞いを見つけにくいというのも悪条件である。このことから、あるエンティティBeanから別のエンティティBeanを呼び出すべきではないという人もいる。呼び出しをしないと再入可能性の問題は回避できるが、**ドメインモデル**を使うメリットは大いに損なわれる。

ドメインモデルは、細かいインタフェースを持つ細かいオブジェクトを使うべきである。エンティティBeanは、リモート呼び出し可能な場合もある（バージョン2.0より前のバージョンはそうであった）。細かいインタフェースを使ってリモートオブジェクトを取得すると、パフォーマンスは劣化する。**ドメインモデル**では、エンティティBeanに対してローカルインタフェースだけを使用して、簡単に問題を回避できる。

エンティティBeanを使うには、コンテナおよび接続されているデータベースが

必要である。これにより構築に要する時間が長くなるだけでなく、テストはデータベースに対して行う必要があるので、テストに要する時間も長くなる。さらにエンティティ Bean はデバッグしにくい。

通常の Java オブジェクトを使うという方法もある。この方法には多くの人が驚くようだが、EJB コンテナ内では通常の Java オブジェクトを使えないと思っている人が意外にも多いのには驚かされる。Java オブジェクトの影が薄い理由は、名称が悪いことだ。そこで、2000 年に講演の原稿をまとめているときに、Rebecca Parsons、Josh Mackenzie と私で考え出したのが POJO（plain old Java objects）だ。「POJO ドメインモデル」は簡単に統合でき、構築も簡単で、テストも EJB コンテナの外で行え、しかも EJB に依存していない（これが、EJB ベンダーがこの EJB 形態を推奨しない理由かもしれない）。

私の見解では、比較的ニーズの低いドメインロジックを使用する場合、エンティティ Bean を**ドメインモデル**として使うことは可能である。この場合、データベースとシンプルな関係にある**ドメインモデル**を構築できる。継承、ストラテジー、洗練されたパターンで構成される複雑なドメインロジックがある場合、POJO ドメインモデルと**データマッパー**や商用ツール、あるいは自作のレイヤを使う方が賢明である。

EJB を使う上での最大の悩みは、複雑な**ドメインモデル**を処理しづらいことと、実装環境の詳細からの独立が困難であることである。EJB を考慮に入れて、**ドメインモデル**を考えなければいけない。つまり、ドメインと EJB 環境について配慮する必要があるのだ。

9.2.2 | 使用するタイミング

ドメインモデルの使用法が大きな課題であるために難しいのであれば、いつ使用すべきかという問題も単純で漠然としたアドバイスにしかならないため、答えるのが難しい。すべてはシステム内の振る舞いの複雑性による。妥当性確認、計算、派生などを含め、複雑で頻繁に変更されるビジネスルールを使っている場合、オブジェクトモデルで対処することがある。一方で、簡単な非ヌルチェックと 2 つの合計を計算する場合、**トランザクションスクリプト**の方が適切である。

開発チームが、ドメインオブジェクトを扱うスキルがあるか否かも重要な要素となってくる。**ドメインモデル**の設計と使用方法を学習するのは有意義なことで、オブジェクト使用の**パラダイムシフト**に関して多くの論文が執筆されることになった。**ドメインモデル**に慣れるまでには相当の訓練と学習を必要とするが、一旦慣れると、よほどシンプルなケースでない限り**トランザクションスクリプト**に戻ろうとするユーザはいないことがわかる。

ドメインモデル を使う場合、データベースの相互作用で私がまず選択するのがデータマッパーである。これで、ドメインモデルがデータベースに依存していない状態を維持できるため、ドメインモデルとデータベースのスキーマが異なる場合に最適な手法になる。

ドメインモデルを使う際は、サービスレイヤの使用を検討し、ドメインモデルが固有のAPI を取得できるようにする。

9.2.3 | 参考文献

オブジェクト指向設計に関するほとんどの書籍でドメインモデルについて触れている。オブジェクト指向開発として参照されるのは、大半がドメインモデルの使用法についてだからである。

オブジェクト指向設計の入門書的なものを探しているのなら、[Larman]を推奨する。ドメインモデルの例を探しているのなら、[Fowler AP]が参考になるだろう。[Hay]にもリレーショナルコンテキストの優れた例が紹介されている。優れたドメインモデルを構築するには、オブジェクトに関する概念的な考え方を理解しておく必要がある。これについては、[Martin and Odell]を勧める。豊富なドメインモデルやその他のオブジェクト指向システムで見られるパターンを理解するには、必ず[Gang of Gour]を読んでおくべきである。

Eric Evans も、最近では、ドメインモデルの構築に関する書籍[Evans]を執筆中だ【訳注】。私は本書執筆中のためまだ初期段階の原稿しか見ていないが、内容はかなり充実しているようである。

9.2.4 | 例：RevenueRecognition（Java）

ドメインモデルを説明する際の最大の悩みの 1 つは、読者が理解できるようにシンプルな例を示すことである。しかし、シンプルさのためにドメインモデルの力が隠れてしまうことがある。その力を評価できるのは、複雑なドメインを扱ったときである。

しかし、紹介する例ではドメインモデルを使用する理由を正しく評価できなくても、大体どんなものかは伝えられるだろう。したがって、トランザクションスクリプトで使用したときと同じ、RevenueRecognition の例を使うことにする。

この簡単な例（図 9.3）では、すべてのクラスに振る舞いとデータの両方が含まれているのにまず気付くだろう。シンプルな RevenueRecognition クラスでさえ、オブジェクト値が特定日に認識されるかどうかがわかる簡単なメソッドを含んでいる。

【訳注】2003 年に「Domain-Driven Design」という書名で刊行された。

```
class RevenueRecognition...

   private Money amount;
   private MfDate date;
   public RevenueRecognition(Money amount, MfDate date) {
      this.amount = amount;
      this.date = date;
   }
   public Money getAmount() {
      return amount;
   }
   boolean isRecognizableBy(MfDate asOf) {
      return asOf.after(date) || asOf.equals(date);
   }
```

どれだけの収益が特定日に認識できるかの計算では、Contract と RevenueRecognition の両方のクラスが使用される。

```
class Contract...

   private List revenueRecognitions = new ArrayList();
   public Money recognizedRevenue(MfDate asOf) {
      Money result = Money.dollars(0);
      Iterator it = revenueRecognitions.iterator();
      while (it.hasNext()) {
         RevenueRecognition r = (RevenueRecognition) it.next();
         if (r.isRecognizableBy(asOf))
            result = result.add(r.getAmount());
      }
      return result;
   }
```

ドメインモデルで共通して見られるのは、複数のクラスが最もシンプルなタスクでも相互作用する方法である。この方法では、オブジェクト指向プログラムで膨大な時間を掛けてクラスからクラスへと探し続けなくてはならないというストレスを引き起こすことがある。ただし、これには多くのメリットがある。特定の日までに何かを認識できるかどうかという決定がより複雑になったとき、または他のオブジェクトがその決定を知る必要が出てきたときに、価値が生まれる。知る必要があるオブジェクトに振る舞いを含めておけば、重複を防ぐことができ、異なるオブジェクト間での結合も軽減する。

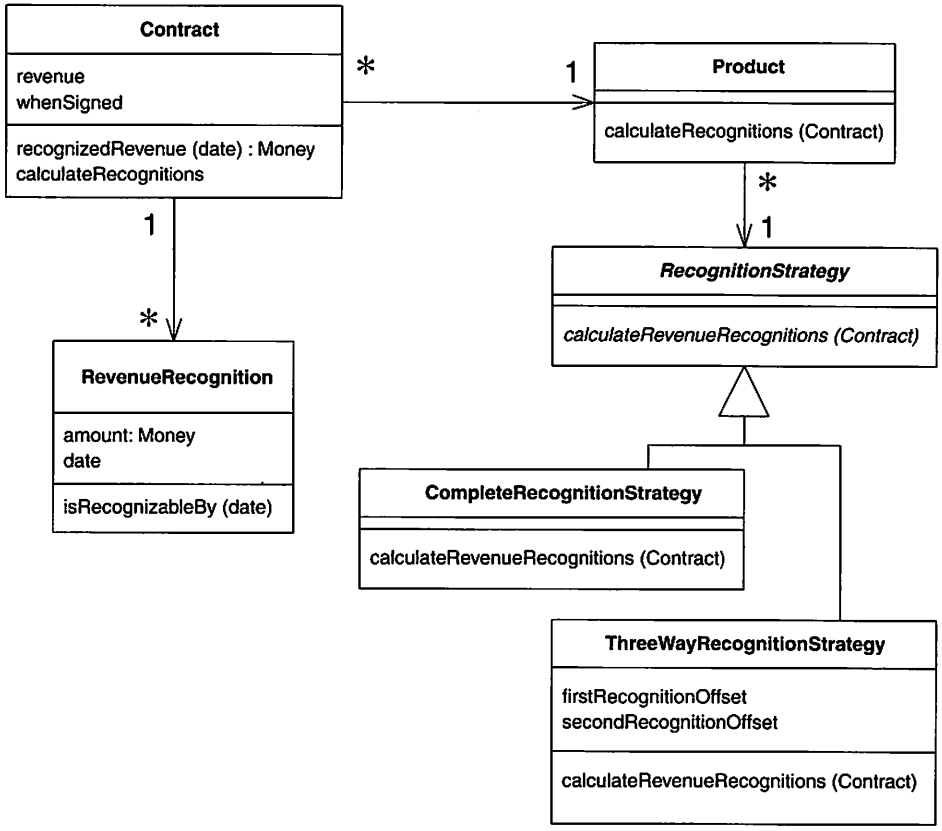


図 9.3 ——ドメインモデル用クラス例のクラス図

これら RevenueRecognition オブジェクトの計算と作成を見ると、多くの小さなオブジェクトという考えがさらに見えてくる。この場合、計算と作成は顧客で始まり、Product を通じてストラテジー階層へと渡される。このストラテジーパターン[Gang of Four]はよく知られているオブジェクト指向パターンで、複数の操作を小さなクラス階層で組み合わせることができる。Product の各インスタンスは RecognitionStrategy の 1 つのインスタンスと繋がっていて、RevenueRecognition を計算するのに使用するアルゴリズムを決定する。2 つの異なるケースのために、2 つの RecognitionStrategy のサブクラスが用意されている。コードの構造は次のようになる。

```
class Contract...

   private Product product;
   private Money revenue;
```

```
    private MfDate whenSigned;
    private Long id;
    public Contract(Product product, Money revenue, MfDate whenSigned) {
        this.product = product;
        this.revenue = revenue;
        this.whenSigned = whenSigned;
    }

class Product...

    private String name;
    private RecognitionStrategy recognitionStrategy;
    public Product(String name, RecognitionStrategy recognitionStrategy) {
        this.name = name;
        this.recognitionStrategy = recognitionStrategy;
    }
    public static Product newWordProcessor(String name) {
        return new Product(name, new CompleteRecognitionStrategy());
    }
    public static Product newSpreadsheet(String name) {
        return new Product(name, new ThreeWayRecognitionStrategy
            (60, 90));
    }
    public static Product newDatabase(String name) {
        return new Product(name, new ThreeWayRecognitionStrategy
            (30, 60));
    }

class RecognitionStrategy...

    abstract void calculateRevenueRecognitions(Contract contract);

class CompleteRecognitionStrategy...

    void calculateRevenueRecognitions(Contract contract) {
        contract.addRevenueRecognition (new RevenueRecognition
        (contract.getRevenue(), contract.getWhenSigned()));
    }

class ThreeWayRecognitionStrategy...

    private int firstRecognitionOffset;
    private int secondRecognitionOffset;
    public ThreeWayRecognitionStrategy
```

```
    (int firstRecognitionOffset, int secondRecognitionOffset)
{
    this.firstRecognitionOffset = firstRecognitionOffset;
    this.secondRecognitionOffset = secondRecognitionOffset;
}
void calculateRevenueRecognitions(Contract contract) {
    Money[] allocation = contract.getRevenue().allocate(3);
    contract.addRevenueRecognition(new RevenueRecognition
        (allocation[0], contract.getWhenSigned()));
    contract.addRevenueRecognition(new RevenueRecognition
        (allocation[1], contract.getWhenSigned().addDays
        (firstRecognitionOffset)));
    contract.addRevenueRecognition(new RevenueRecognition
        (allocation[2], contract.getWhenSigned().addDays
        (secondRecognitionOffset)));
}
```

ストラテジーの大きな価値は、アプリケーションを拡張できるプラグポイントを、整理された状態で提供するという点である。新たなRevenueRecognitionアルゴリズムを追加するということは、新たなサブクラスを作成し、calculateRevenueRecognitionsメソッドをオーバーライドすることになる。アプリケーションのアルゴリズムの振る舞いを拡張する際、簡単に行えるようにするものである。

Productを作成したら、該当するストラテジーオブジェクトに接続させればよい。接続はテストコードの中で行っている。

```
class Tester...

    private Product word = Product.newWordProcessor("Thinking Word");
    private Product calc = Product.newSpreadsheet("Thinking Calc");
    private Product db = Product.newDatabase("Thinking DB");
```

一度すべてをセットアップしてしまえば、認識される収益の計算ではStrategyサブクラスに関する知識は必要なくなる。

```
class Contract...

    public void calculateRecognitions() {
        product.calculateRevenueRecognitions(this);
    }
```

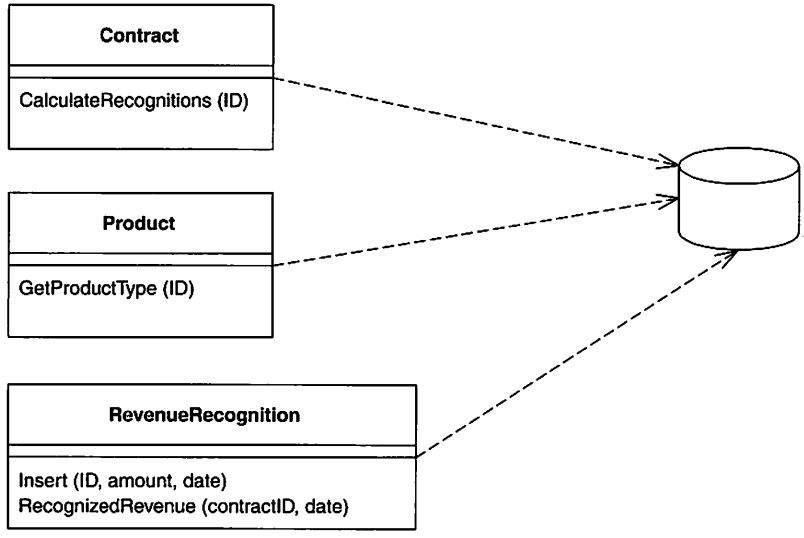
```
class Product...

   void calculateRevenueRecognitions(Contract contract) {
      recognitionStrategy.calculateRevenueRecognitions(contract);
   }
```

オブジェクトからオブジェクトへと連続的に連携するオブジェクト指向の習性により、振る舞いは処理に最も適しているオブジェクトへと移動するが、これにより条件分岐を持つ振る舞いの多くも解決することができる。この計算式には条件分岐が含まれていないことがわかる。決定パスの設定は、適切なストラテジーでProductを作成する際に行う。すべてがこのように結合されると、アルゴリズムは単純にパスに従うようになる。類似の複数の条件分岐がある場合、**ドメインモデル**はとても適していると言える。類似条件がオブジェクト構造自体に抜き出されるからである。これにより、複雑性はアルゴリズムからオブジェクト間の関係に移行することになる。ロジックが似ているほど、システムの異なる部分で使用される同じネットワークの関係が明らかになる。どのような収益認識用のアルゴリズムも、この特定のオブジェクトネットワークに従うことができる。

例では、オブジェクトがデータベースからどのように取得され作成されるかについて一切触れていないが、それにはいくつかの理由がある。まず、**ドメインモデル**をデータベースにマッピングするのは通常難しいので、ここでは例を紹介しない。次に、多くの点で**ドメインモデル**では、上位レイヤや**ドメインモデル**で作業するユーザからデータベースを隠すことに意義がある。したがって、ここでデータベースを隠すことは、このような環境でプログラムすることを表している。

9.3 テーブルモジュール

データベーステーブルかビューのすべての行に関するビジネスロジックを扱うシングルインスタンス。



オブジェクト指向の鍵となるメッセージの1つは、メッセージを使用する振る舞いとデータをバンドルしているという点にある。従来のオブジェクト指向の手法は、**ドメインモデル**のように、一意性を持つオブジェクトをベースにしている。つまり、Employee（従業員）クラスがある場合、クラスのどんなインスタンスも特定のEmployeeと一致するのである。あるEmployeeを参照すると、操作の実行、関係の追跡、そのEmployeeについてのデータの収集を行うことができるので、この仕組みは便利なものとなる。

ドメインモデルが抱える問題の1つに、リレーショナルデータベースとのインタフェースがある。この手法ではたいていリレーショナルデータベースを無視している。その結果、データベースからのデータの出し入れに相当なプログラミングが必要となる場合があり、2つの異なるデータ表現を変換することになる。

テーブルモジュールは、データベース内のテーブル1つにつき1つのクラスを持つドメインロジックを構築し、クラスの1つのインスタンスにはデータに対して動作する各種の手続きが含まれる。**ドメインモデル**との主な違いは、多くのOrderがある場合、**ドメインモデル**では1つのOrderに1つのOrderオブジェクトが使われるが、**テーブルモジュール**では、1つのオブジェクトですべてのOrderを扱うという点である。

9.3.1 | 動作方法

テーブルモジュールのメリットは、データと振る舞いを1つのパッケージにできると同時に、リレーショナルデータベースのメリットを活かせるという点にある。表面的には**テーブルモジュール**はほとんどオブジェクトと同様に見える。最大の違いは、作業対象となるオブジェクトに対する一意性の概念がないという点である。つまり、あるEmployeeの住所（address）を取得したい場合、`anEmployeeModule.getAddress(long employeeID)`といったメソッドを使うことになる。特定のEmployeeに対して処理を行う場合は、常に何らかの一意性のある参照を渡す必要がある。この一意性のある参照はデータベース内で使用されているプライマリキーであることが多い。

テーブルモジュールはテーブル指向のバッキングデータ構造で使用する。テーブル形式のデータは、通常SQL呼び出しの結果であり、SQLテーブルに似た**レコードセット**内に格納される。**テーブルモジュール**では、データに対して動作する明示的なメソッドベースのインタフェースを利用できる。振る舞いとテーブルのグループ化により、カプセル化の多くのメリットが得られる。カプセル化では、振る舞いが作業対象のデータに近くなるからである。

有効な作業を行うために、複数の**テーブルモジュール**からの振る舞いが必要になることもある。多くの場合、複数の**テーブルモジュール**が、同じ**レコードセット**を操作することになる（図9.4）。

テーブルモジュールの最も明確な例は、データベース内のテーブルごとに1つずつこの**テーブルモジュール**を使うことである。しかし、データベース内で凝ったクエリーやビューを使っている場合は、複数の**テーブルモジュール**を使うこともできる。

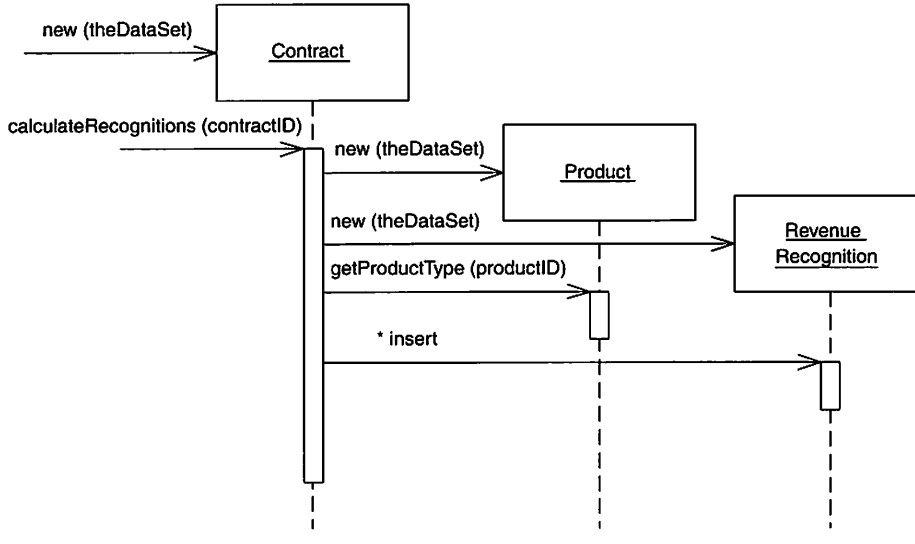


図9.4 ——複数のテーブルモジュールを1つのレコードセットと協調することができる。

テーブルモジュールは、インスタンスの場合もあれば、静的メソッドが集まったものの場合もある。インスタンスのメリットは、クエリーの結果など既存のレコードセットで**テーブルモジュール**を初期化できる点である。その後、インスタンスを使ってレコードセット内の行を操作できる。インスタンスは継承の使用も可能にするので、通常の Contract に振る舞いを追加して急ぎの Contract モジュールを作成することもできる。

テーブルモジュールでは、クエリーをファクトリメソッドとして含むことができる。**テーブルデータゲートウェイ**を使うという方法もあるが、この場合、設計内に余計な**テーブルデータゲートウェイ**クラスとメカニズムが含まれてしまうことになる。一方、異なるデータソースからのデータに対して 1 つの**テーブルモジュール**を使えるというメリットもある。データソースに対して異なる**テーブルデータゲートウェイ**が使われるからである。

テーブルデータゲートウェイを使用する場合、アプリケーションはまず**テーブルデータゲートウェイ**を使って**レコードセット**内のデータをまとめる。次に、**レコードセット**を引数にした**テーブルモジュール**を作成する。複数の**テーブルモジュール**からの振る舞いが必要な場合も、同じ**レコードセット**から作成することができる。作成した**テーブルモジュール**は、**レコードセット**上でビジネスロジックを実行し、修正された**レコードセット**をプレゼンテーションに渡して表示したり、テーブルを認識するウィジェットで編集したりできるようにする。

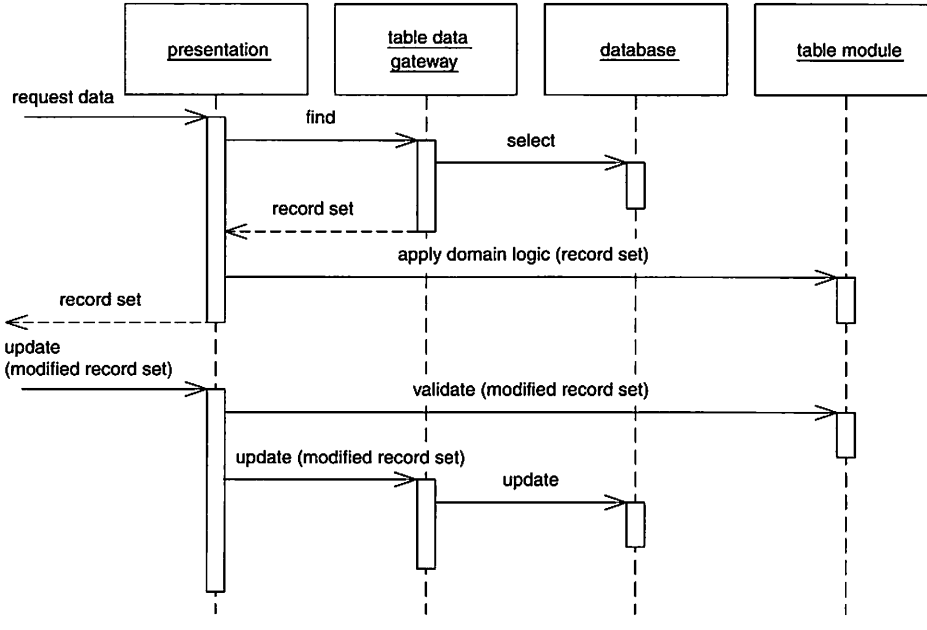


図 9.5 ——テーブルモジュールの周りのレイヤに対する一般的な相互作用

ウィジェットは、レコードセットがリレーショナルデータベースから直接来たのか、**テーブルモジュール**が途中でデータを操作したのかは判断できない。GUIでの修正後、データセットは妥当性確認のために**テーブルモジュール**に戻ってから、データベースに保存される。このスタイルのメリットの1つに、データベースに行かなくてもメモリ内に**レコードセット**を作成して**テーブルモジュール**をテストできるという点がある。

パターン名にテーブルが使われているように、データベース内の1つのテーブルに対して1つの**テーブルモジュール**を使用する。これは最初の概算ではあるが、厳密には完全に正しいというわけではない。一般に使用されるビューやその他のクエリーでも**テーブルモジュール**は便利なものとなる。実際、**テーブルモジュール**の構造は、データベース内のテーブル構造よりも、ビューやクエリーなども含めたアプリケーションで認識される仮想テーブルに依存したものとなっている。

9.3.2 | 使用するタイミング

テーブルモジュールは、ほとんどがテーブル指向のデータをベースにしたものなので、**レコードセット**を使ってテーブル形式のデータにアクセスしている場合に使うのは理にかなっていると言える。また、データ構造がコードのほぼ中心に位置しているので、データ構造へのアクセス方法もできるだけシンプルなものにするべきである。

ただし、**テーブルモジュール**では、複雑なロジックを構築する際にオブジェクトが持つ強力なパワーは得られない。インスタンス間での直接的な関係を築くことはできず、ポリモーフィズムもうまく機能しない。つまり、複雑なドメインロジックでは、**ドメインモデル**の方が適しているということになる。複雑なロジックを扱える**ドメインモデル**の機能性を取るか、基盤となるテーブル指向のデータ構造と簡単に統合できる**テーブルモジュール**の統合性を取るか、という問題になる。

ドメインモデル内のオブジェクトとデータベーステーブルが比較的類似している場合、**アクティブレコード**を使う**ドメインモデル**の方が適しているだろう。**ドメインモデル**と**アクティブレコード**の組み合わせよりも**テーブルモジュール**の方が適しているのは、アプリケーションのその他の部分が、共通のテーブル指向のデータ構造をベースにしている場合である。Java環境で**テーブルモジュール**をほとんど見かけないのはこのためである。しかし、行セットがもっと広範に使われるようになれば、この状況も変わるかもしれない。

これまで目にしたこのパターンを使った最も有名な例は、Microsoft COMの設計である。COM（およびNET）では、**レコードセット**がアプリケーション内のデータの主要なリポジトリになっている。レコードセットはUIに渡すことができ、データを認識するウィジェットによって情報が表示される。MicrosoftのADOライブラリは、リレーショナルデータにレコードセットとしてアクセスしやすいメカニズムを提供している。この状況で**テーブルモ**

ジュールを使えば、テーブル形式のデータに対する各種の要素の操作性を失うことなく、上手く体系化された方法でビジネスロジックをアプリケーションに組み入れることができる。

9.3.3 | 例：テーブルモジュールでの RevenueRecognition（C#）

ここで再びこれまでのドメインモデリングパターンで使用した RevenueRecognition の例が登場する。今回はテーブルモジュールを使用する。簡単に復習しておこう。ここでの課題は、Product の種類によってルールが異なる場合での Order の RevenueRecognition を計算することである。この例では、ワープロ、表計算、データベースの Product ごとに異なるルールが定義されている。

テーブルモジュールは、ある種のデータスキーマをベースにしていて、通常これはリレーショナルデータモデルである（将来は、XML モデルが似たような方法で使用されるようになるかもしれない）。ここでは、図 9.6 のリレーショナルスキーマを使用する。

このデータを処理するクラスは、どれもほとんど同じ形式になっていて、テーブルごとに 1 つのテーブルモジュールクラスが用意されている。.NET アーキテクチャでは、データセットオブジェクトは、データベース構造をメモリ上で表すものとなる。つまり、データセット上で動作するクラスを作成するというのは理にかなっている。各テーブルモジュールクラスはデータテーブルのデータメンバを持っている。これは、データセット内のテーブルに相当する.NET システムクラスである。テーブル読み取り機能は、すべてのテーブルモジュールに共通しているのでレイヤスーパータイプでも使われている。

```
class TableModule...

      protected DataTable table;
      protected TableModule(DataSet ds, String tableName) {
         table = ds.Tables[tableName];
      }
```

サブクラスコンストラクタは、正しいテーブル名でスーパークラスコンストラクタを呼び出す。

```
class Contract...

      public Contract (DataSet ds) : base (ds, "Contracts") {}
```

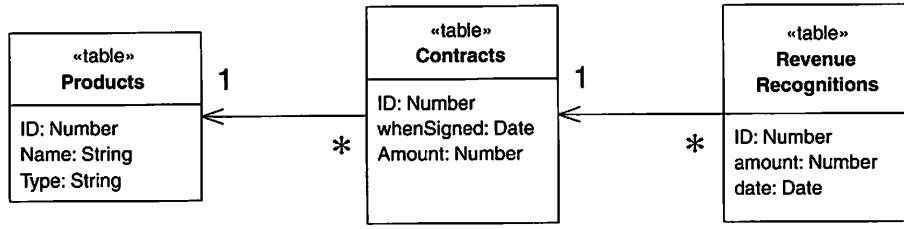


図 9.6 —— RevenueRecognition 用データベーススキーマ

これで、テーブルモジュールのコンストラクタにデータセットを渡すだけで新しいテーブルモジュールが作成できるようになる。

```
contract = new Contract(dataset);
```

ADO.NET のガイドラインによると、データセットを作成するコードはテーブルモジュールから隔離される。

便利なのは、所定のプライマリキーでデータテーブル内の特定行に行ける C# インデクサーである。

```
class Contract...

    public DataRow this [long key] {
    get {
       String filter = String.Format("ID = {0}" , key);
       return table.Select(filter)[0];
       }
    }
```

最初の機能では、Contract の RevenueRecognition を計算し、これにより RevenueRecognitions テーブルを更新する。認識される額は、Product の種類によって違う。この振る舞いは、主に Contracts テーブルからのデータを使っているので、Contract クラスにメソッドを追加する。

```
class Contract...

    public void CalculateRecognitions (long contractID) {
       DataRow contractRow = this[contractID];
       Decimal amount = (Decimal)contractRow["amount"];
       RevenueRecognition rr = new RevenueRecognition (table.DataSet);
```

```
        Product prod = new Product(table.DataSet);
        long prodID = GetProductId(contractID);
        if (prod.GetProductType(prodID) == ProductType.WP) {
            rr.Insert(contractID, amount, (DateTime)
                GetWhenSigned(contractID));
        } else if (prod.GetProductType(prodID) == ProductType.SS) {
            Decimal[] allocation = allocate(amount,3);
            rr.Insert(contractID, allocation[0], (DateTime)
                GetWhenSigned(contractID));
            rr.Insert(contractID, allocation[1], (DateTime)
                GetWhenSigned(contractID).AddDays(60));
            rr.Insert(contractID, allocation[2], (DateTime)
                GetWhenSigned(contractID).AddDays(90));
        } else if (prod.GetProductType(prodID) == ProductType.DB) {
            Decimal[] allocation = allocate(amount,3);
            rr.Insert(contractID, allocation[0], (DateTime)
                GetWhenSigned(contractID));
            rr.Insert(contractID, allocation[1], (DateTime)
                GetWhenSigned(contractID).AddDays(30));
            rr.Insert(contractID, allocation[2], (DateTime)
                GetWhenSigned(contractID).AddDays(60));
        } else throw new Exception("invalid product id");
    }
    private Decimal[] allocate(Decimal amount, int by) {
        Decimal lowResult = amount / by;
        lowResult = Decimal.Round(lowResult,2);
        Decimal highResult = lowResult + 0.01m;
        Decimal[] results = new Decimal[by];
        int remainder = (int) amount % by;
        for (int i = 0; i < remainder; i++) results[i] = highResult;
        for (int i = remainder; i < by; i++) results[i] = lowResult;
        return results;
    }
```

通常はマネーを使うが、バリエーションを持たせるために、ここではDecimalを使った例を示している。割り当てメソッドは、マネーで使用しているものと似ている。

実行するには、他のクラス上で定義されているいくつかの振る舞いが必要となる。Product（製品）はタイプを知らせる必要があるが、Product（製品）用のenumと照合テーブルを使って対処できる。

```
public enum ProductType {WP, SS, DB};
```

```
class Product...

    public ProductType GetProductType (long id) {
        String typeCode = (String) this[id]["type"];
        return (ProductType) Enum.Parse(typeof(ProductType), typeCode);
    }
```

GetProductType は、データテーブル内のデータをカプセル化している。カプセル化をすべてのデータ列に対してするのか、それとも Contract の金額のように直接アクセスするかについては、議論が分かれる。一般的にカプセル化が推奨されているが、ここではシステムのさまざまな部分からデータセットに直接アクセスするという環境を前提にしているため適さないので使用しない。データセットが UI 上に移動する際には、カプセル化は行わない。したがって、列アクセス機能は、文字列の Product 種類への変更などの追加機能が必要でない限り、合理的ではない。

もう 1 つここで触れておきたいことがある。それは、ここでは型指定されていないデータセットを使っているという点だ。多くのプラットフォームで一般的だからである。当然、.NET のしっかりと型指定されたデータセットを使うべきだと言う意見もある。

もう 1 つ追加する振る舞いは、新たな RevenueRecognition レコードの挿入である。

```
class RevenueRecognition...

    public long Insert (long contractID, Decimal amount,
        DateTime date) {
        DataRow newRow = table.NewRow();
        long id = GetNextID(); newRow["ID"] = id;
        newRow["contractID"] = contractID;
        newRow["amount"] = amount;
        newRow["date"]= String.Format("{0:s}", date);
        table.Rows.Add(newRow); return id;
    }
```

データ行のカプセル化を行うためでなく、コードの繰り返しを防ぐために、このメソッドは作成されている。

2 つ目の関数は、所定の日付での Contract（契約）の RevenueRecognition を合計するというものである。これは RevenueRecognitions テーブルを使うので、RevenueRecognition クラスでメソッドを定義している。

```
class RevenueRecognition...

      public Decimal RecognizedRevenue (long contractID,
         DateTime asOf) {
         String filter = String.Format("ContractID = {0} AND
            date <= #{1:d}#", contractID,asOf);
         DataRow[] rows = table.Select(filter);
         Decimal result = 0m;
         foreach (DataRow row in rows) {
            result += (Decimal)row["amount"];
         }
         return result;
      }
```

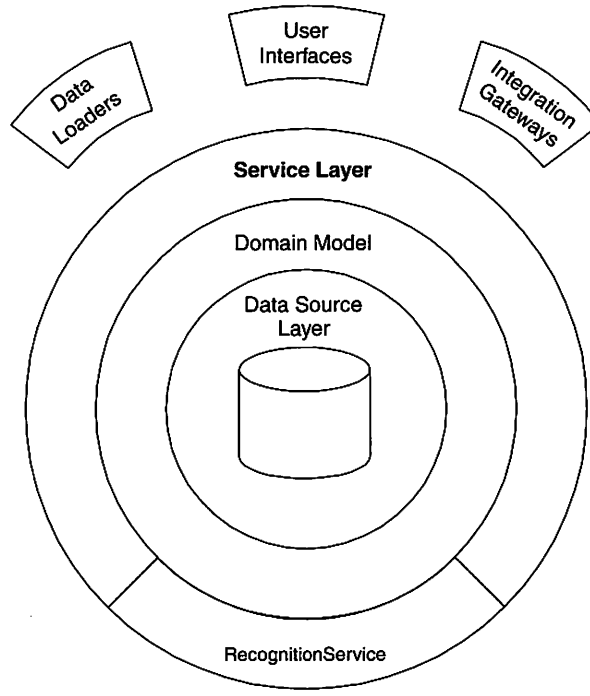
上記の一部では、where 句を定義し、操作するデータテーブルのサブセットを選択できる ADO.NET の機能を効果的に活用している。実際、さらに掘り下げて集合関数を使うこともできる。

```
class RevenueRecognition...

      public Decimal RecognizedRevenue2 (long contractID,
         DateTime asOf) {
         String filter = String.Format("ContractID = {0} AND
            date <= #{1:d}#", contractID,asOf);
         String computeExpression = "sum(amount)";
         Object sum = table.Compute(computeExpression, filter);
         return (sum is System.DBNull) ? 0 : (Decimal) sum;
      }
```

9.4 サービスレイヤ

(by Randy Stafford)

サービスのレイヤとアプリケーションの境界を定義する。サービスは利用できる操作セットを設定し、各操作に対するアプリケーションのレスポンスを調整するものである。



エンタープライズアプリケーションでは、格納しているデータや実装しているロジックに対して異なる種類のインタフェースを必要とする。たとえば、データローダー、ユーザインタフェース、統合ゲートウェイなどである。目的は異なるが、インタフェースは、データへのアクセスと操作やビジネスロジックの呼び出しにおいてアプリケーションと共通した相互作用を必要とする場合がある。この相互作用は、複数のリソースにまたがるトランザクションやアクションに対する複数のレスポンスの調整など複雑なものになることもある。相互作用ロジックを各インタフェース内で個別にエンコードした場合、多くの重複が生じてしまう。

サービスレイヤは、アプリケーションの境界[Cockburn PloP]と利用できる操作セットをクライアントレイヤとのインタフェースという観点から定義している。**サービスレイヤ**はアプリケーションのビジネスロジックをカプセル化し、トランザクションを制御し、操作の実装におけるレスポンスを調整する。

9.4.1 | 動作方法

サービスレイヤは、上述の特性を侵害することなく、いくつかの異なる方法で実装することができる。実装方法の違いは、**サービスレイヤ**インタフェースに対する責任の割り当てに現れている。各種の実装方法を紹介する前に、少し背景を紹介する。

9.4.1.1 ■ ビジネスロジックの種類

トランザクションスクリプトや**ドメインモデル**同様、**サービスレイヤ**もビジネスロジックを構成するパターンの1つである。私も含めて設計者の多くは、「ビジネスロジック」を2種類に分類する傾向がある。1つは、純粋に問題ドメイン（Contract の RevenueRecognition を算出するストラテジーなど）を扱う「ドメインロジック」で、もう1つは、アプリケーションの責任[Cockburn UC]（RevenueRecognition 計算における Contract 管理者への通知やアプリケーションの統合など）を扱う「アプリケーションロジック」である。アプリケーションロジックは「ワークフローロジック」と呼ばれることもあるが、「ワークフロー」の解釈については人によって意見が分かれる。

ドメインモデルは、ドメインロジックの重複を避けるという点と従来の設計パターンを使って複雑性を管理するという点で、**トランザクションスクリプト**よりも好ましいとされる。しかし、アプリケーションロジックを純粋なドメインオブジェクトクラスに入れることは、いくつかの好ましくない事態を招くことにもなる。まず、ドメインオブジェクトクラスがアプリケーション固有のロジックを実装し、アプリケーション固有のパッケージに依存すると、アプリケーション間での再利用の可能性が低下する。次に、両方の種類のロジックを同じクラス内に混在させると、必要とされるワークフローツールなどへのアプリケーションロジックの再実装が難しくなる。これらの理由から、**サービスレイヤ**では多様な種類のビジネスロジックを個別のレイヤに組み込み、レイヤ化の通常のメリットを引き出し、純粋なドメインオブジェクトクラスをアプリケーション間で再利用できるようにする。

9.4.1.2 ■ 実装バリエーション

2つの基本的な実装バリエーションには、ドメインファサード的手法と操作スクリプト手法がある。**ドメインファサード**手法では、**サービスレイヤ**は**ドメインモデル**上の薄いファサードのセットとして実装される。ファサードを実装するクラスにはビジネスロジックは一切実装されていない。ビジネスロジックはすべて**ドメインモデル**によって実装される。薄いファサードは、境界およびクライアントレイヤがアプリケーションと相互作用を行うための操作群を設定する。それが**サービスレイヤ**の特徴である。

操作スクリプト手法では、厚いクラス群として**サービスレイヤ**が実装され、アプリケーションロジックは直接実装されるが、ドメインロジックはカプセル化されたドメインクラスに委譲される。**サービスレイヤ**のクライアントが利用できる操作はスクリプトとして実装さ

れ、関連するロジックの対象エリアを定義するクラスに一部の操作が体系化される。各クラスはアプリケーション「サービス」を形成するが、これは名前の最後に「サービス」が付く種類のサービスでは一般的なものである。**サービスレイヤ**はこれらのアプリケーションサービスクラスによって構成され、**レイヤスーパータイプ**を拡張する。さらに一定のルールや共通な振る舞いを抽出する。

9.4.1.3 ■ リモートにするべきか否か

サービスレイヤクラスのインタフェースは、クライアントレイヤが利用できるアプリケーション操作セットを宣言するので、粗い粒度になっている。インタフェースの粒度から見ると、**サービスレイヤ**クラスはリモート呼び出しに適したものになる。

しかし、リモート呼び出しにはオブジェクトの分散を扱うという犠牲が伴う。**サービスレイヤ**のメソッドシグネチャを**データ変換オブジェクト**で扱えるようにするには、相当な追加作業が必要となるため、作業工数を軽視できない。特に、複雑な**ドメインモデル**と複雑な更新ユースケース用の高度な編集 UI がある場合はなおさらである。この作業は、重要で困難でもある。おそらくオブジェクトリレーショナルマッピングの次にコストが掛かる困難な作業であると言える。分散オブジェクトデザインの第一法則（P95 参照）を忘れないでほしい。

まず、ローカルで呼び出すことができ、メソッドシグネチャがドメインオブジェクト内で処理できる**サービスレイヤ**から始めることを推奨する。リモート性は（本当に）必要になった時点で、**リモートファサード**を**サービスレイヤ**上に置くか、**サービスレイヤ**オブジェクトにリモートインタフェースを実装することにより付加する。アプリケーションが Web ベースの UI か Web サービスベースの統合ゲートウェイを備えている場合、ビジネスロジックをサーバページや Web サービスとは切り離されたプロセスで実行しなくてはならないという決まりがあるわけではない。事実、同じ場所に配置する方法で始めれば、拡張性を犠牲にすることなく、開発労力と実行時のレスポンス時間を短縮することができる。

9.4.1.4 ■ サービスと操作の特定

サービスレイヤ境界で必要な操作の特定はいたって簡単である。これらは、**サービスレイヤ**クライアントのニーズによって決定されるが、最も重要なもの（最初のものでもある）はユーザインタフェースになる。ユーザインタフェースは、アクタがアプリケーションで実行したいユースケースをサポートするように設計されているので、**サービスレイヤ**操作の特定はユースケースモデルとアプリケーション固有のユーザインタフェースから始めるべきである。

残念だが実情は、エンタープライズアプリケーションのユースケースの多くは、ドメインオブジェクトに対する退屈な「CRUD」（create、read、update、delete）ユースケースである。つまり、ドメインオブジェクトを作成し、集まったものを読み取り、更新するという

ことである。私の経験では、ほとんどの場合、CRUD ユースケースとサービスレイヤ操作の間には 1 対 1 で一致するものが必ず存在する。

こうしたユースケースを実行するというアプリケーションの責任は重要である。妥当性確認は別にしても、アプリケーションにおけるドメインオブジェクトの作成、更新、削除は、他のユーザや他の統合アプリケーションへの通知を必要とする。サービスレイヤ操作によって確実に調整、実行しなくてはならない。

サービスレイヤの抽象化を特定して、関連する操作をグループ化するのが簡単であればよいのだが、このエリアにおける厳格な規定はない。単なる曖昧なヒューリスティックスに過ぎない。とても小さなアプリケーションでは、アプリケーションそのものの名前を持つ 1 つの抽象化だけで十分な場合がある。私の経験では、大きなアプリケーションはいくつかの「サブシステム」に分割され、それぞれがアーキテクチャレイヤの階層を垂直方向に切り取ったものを含んでいる。この場合、1 つのサブシステムに 1 つの抽象化を持たせ、名前もサブシステムにちなんだものにするのが好ましい。その他の可能性としては、ドメインモデル内の主なパーティションがサブシステムのパーティションと異なる場合は、反映した抽象化（例：`ContractsService`、`ProductsService`）や、主だったアプリケーションの振る舞いの名前にちなんだ抽象化などがある。

Java での実装

ドメインファサード手法と操作スクリプト手法のいずれの場合でも、サービスレイヤクラスは POJO またはステートレスセッション Bean として実装することができる。ここでのトレードオフは、テストの容易性とトランザクション制御の容易性になる。POJO は、実行するために EJB コンテナ内に配置する必要がないのでテストするのは簡単であるが、POJO サービスレイヤを分散コンテナ管理トランザクションサービスに組み込むのは困難である。特に、サービス間での呼び出し内では大変難しい。一方の EJB は、コンテナ管理分散トランザクション用の基本的な能力は備えているが、テストや実行ではコンテナ内に導入しなくてはならない。どちらでも好きな方を選択すればいい。

私なら、サービスレイヤを J2EE で適用する際は、EJB 2.0 ステートレスセッション Bean によりローカルインタフェースを使用する、または、操作スクリプト手法により POJO ドメインオブジェクトクラスに委譲させる。EJB により提供される分散コンテナ管理トランザクションがあるので、ステートレスセッション Bean を使ってサービスレイヤを実装するのは、とても便利なものとなる。また、EJB 2.0 で登場したローカルインタフェースにより、サービスレイヤで厄介なオブジェクトの分散問題を避けて価値あるトランザクションサービスを実行できる。

Java に関連して 1 つ触れておきたいことがある。サービスレイヤと J2EE パ

ターンの文献[Alur et al.]と[Marinescu]で紹介されている**セッションファサード**のパターンとの違いである。**セッションファサード**は、エンティティBeanに対するリモート呼び出しが多すぎることによるパフォーマンス低下の回避をベースにしている。したがって、ファサードエンティティBeanをセッションBeanと記述している。一方の**サービスレイヤ**は、規定の条件を抜き出すことで重複を避けて再利用性を高めることをベースにしている。それは、高度なアーキテクチャパターンである。**サービスレイヤ**に影響を与えたApplication Boundaryパターン[Cockburn PloP]は、EJBよりも3年も前に登場している。**セッションファサード**も本来は**サービスレイヤ**と同類かもしれないが、現在の名前や用途や提示の仕方が示しているように、同じものではない。

9.4.2 | 使用するタイミング

サービスレイヤのメリットは、多様なクライアントが利用できるアプリケーションの共通の操作セットを定義する点、および各操作でのアプリケーションのレスポンスを調整するという点である。レスポンスには、複数のトランザクションリソースにまたがって、しっかりと処理される必要があるアプリケーションロジックも含まれる。つまり、ビジネスロジックで複数の種類のクライアントを使用したアプリケーションと複数のトランザクションリソースを含むユースケースでの複雑なレスポンスでは、**サービスレイヤ**をコンテナ管理トランザクションと一緒に含むのは理にかなったことである。非分散アーキテクチャでも同様である。

いつ使うべきでないかは、いとも簡単に答えられる。アプリケーションのビジネスロジックが、(ユーザインタフェースなど)1種類のクライアントしか持たず、ユースケースのレスポンスにマルチトランザクションリソースが関連していない場合は、**サービスレイヤ**は不要である。この場合、ページコントローラを使って、トランザクションの手動制御やレスポンスが必要な場合の調整やデータソースレイヤへの直接委譲などを行うことができる。

しかし、違う種類のクライアントやユースケースのレスポンスにおける別のトランザクションリソースの考えが出てきたら、最初から**サービスレイヤ**で設計した方が効率的である。

9.4.3 | 参考文献

サービスレイヤに関してはそれほど多くの先例があったわけではないが、生みの親はAlistair CockburnのApplication Boundaryパターン[Cockburn PloP]である。リモート可能なサービスに関しては、[Alpert, et al.]で分散システムにおけるファサードの役割について紹介している。これを、[Alur et al.]と[Marinescu]のセッションファサードのプレゼンテーションと比較検討してみると面白い。**サービスレイヤ**操作内で調整する必要がある

アプリケーションの責任に関しては、ユースケースを振る舞いのContractとしてCockburnが解説している[Cockburn UC]が大変役立つ。初期段階におけるバックグラウンドのリファレンスは、Fusion方法論による「システム操作」[Coleman et al.]の認識である。

9.4.4 | 例：RevenueRecognition（Java）

ここでは**トランザクションスクリプト**と**ドメインモデル**のパターンと同じRevenue Recognition例を引き続き使って、**サービスレイヤ**を使用してどのようにしてアプリケーションロジックを記述し、**サービスレイヤ**操作でのドメインロジックを委譲するかを紹介する。ここでは操作スクリプト手法を使用し、まずはPOJOを、次にEJBで**サービスレイヤ**を実装する。

デモンストレーションを作るにあたって、シナリオを拡大していくつかのアプリケーションロジックを含めている。アプリケーションのユースケースの場合を考えてみよう。ContractのRevenueRecognitionが計算された際に、アプリケーションがそのイベントの通知を電子メールで指定されたContract管理者に送り、メッセージ指向のミドルウェアを使ってメッセージを発行し、その他の統合アプリケーションに通知するという形で応答する必要がある。

まず**トランザクションスクリプト**の例のRecognitionServiceクラスを変更して**レイヤスーパータイプ**を拡張し、いくつかの**ゲートウェイ**を使ってアプリケーションロジックを実行する。クラス図は図9.7のようになる。RecognitionServiceは**サービスレイヤ**アプリケーションサービスのPOJO実装となり、メソッドはアプリケーションの境界で利用できる2つの操作を表している。

RecognitionServiceクラスのメソッドは操作のアプリケーションロジックを記述し、ドメインロジックのためのドメインオブジェクトクラス（**ドメインモデル**の例からの）を委譲する。

```
public class ApplicationService {
   protected EmailGateway getEmailGateway() {
      //return an instance of EmailGateway
   }
   protected IntegrationGateway getIntegrationGateway() {
      //return an instance of IntegrationGateway
   }
}
public interface EmailGateway {
   void sendEmailMessage(String toAddress, String subject,
```

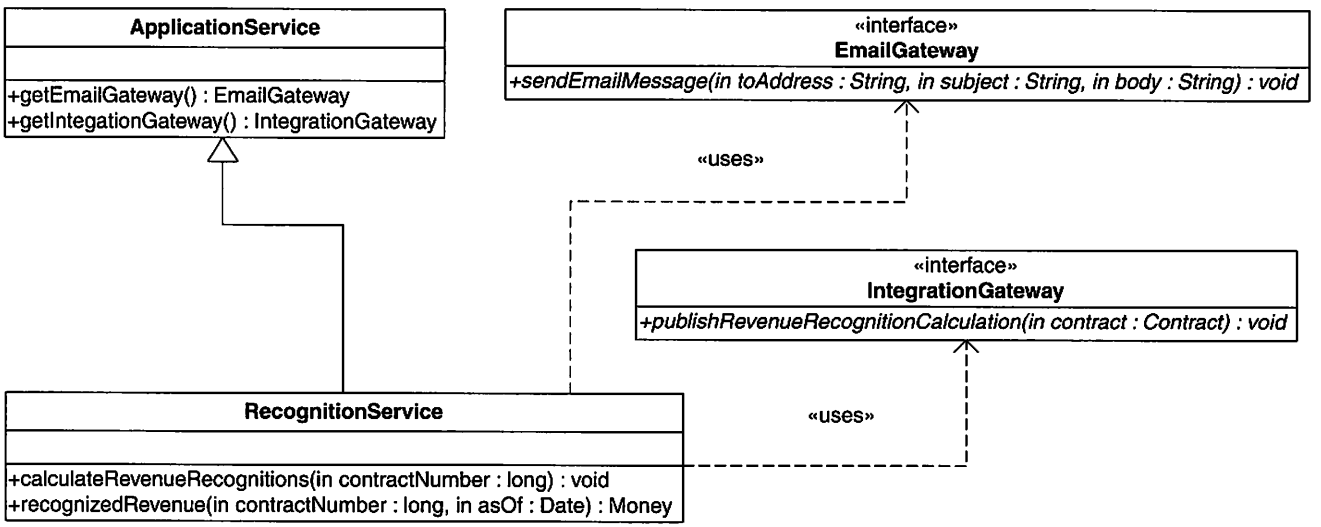


図 9.7 —— RecognitionService POJO クラスダイアグラム

```
        String body)
}
public interface IntegrationGateway {
    void publishRevenueRecognitionCalculation(Contract contract);
}
public class RecognitionService
    extends ApplicationService {
public void calculateRevenueRecognitions(long
    contractNumber) {
    Contract contract = Contract.readForUpdate (contractNumber);
    contract.calculateRecognitions();
    getEmailGateway().sendEmailMessage(
        contract.getAdministratorEmailAddress(),
        "RE: Contract #" + contractNumber,
        contract + " has had revenue recognitions calculated.");
    getIntegrationGateway().publishRevenueRecognitionCalculation
        (contract);
}
    public Money recognizedRevenue(long contractNumber,
        Date asOf) {
        return Contract.read(contractNumber).recognizedRevenue
            (asOf);
    }
}
```

例には詳細が含まれていない。Contract クラスがデータソースレイヤから契約番号で Contract を読み取る静的メソッドが実装されているというだけで十分である。メソッドの1つは読み取った Contract を更新することを意図した名前を持ち、下層の**データマッパー**が読み取ったオブジェクトをたとえば**ユニットオブワーク**で登録できるようになっている。

トランザクション制御に関する詳細もこの例では省略する。calculateRevenueRecognitions() メソッドは、本質的にはトランザクション的なものである。それは、実行中に永続的 Contract オブジェクトが RevenueRecognition の追加により修正され、メッセージがメッセージ指向のミドルウェア内のキューに入れられ、電子メールメッセージが送信されることになるからである。これらのレスポンスは、必ず正確に処理される必要がある。Contract の変更が永続的なものにならなかった場合、電子メールの送信や他のアプリケーションへのメッセージの発行は行わないからである。

J2EE プラットフォームでは、トランザクションリソースを使うステートレスセッション Bean としてアプリケーションサービス（および**ゲートウェイ**）を実装することで、EJB コンテナに分散トランザクションを管理させることができる。図 9.8 は、EJB 2.0 ローカルインタフェースと「ビジネスインタフェース」言語を使う RecognitionService の実装のクラ

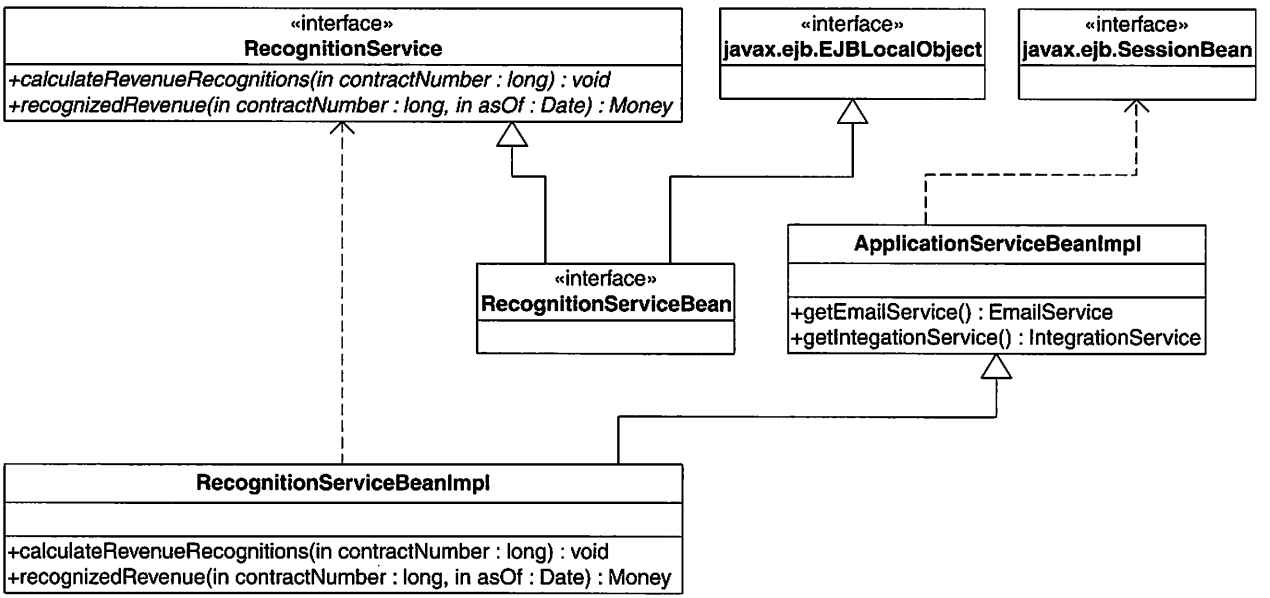


図 9.8 —— RecognitionService EJB クラスダイアグラム

スダイアグラムを示している。この実装でも**レイヤスーパータイプ**が使われているが、これはEJBではアプリケーション固有メソッドに加え、bean実装クラスメソッドのデフォルト実装が必要だからである。EmailGatewayとIntegrationGatewayインタフェースもそれぞれのステートレスセッションBeanに対して「ビジネスインタフェース」であると想定すれば、分散トランザクションの制御はcalculateRevenueRecognitionsとsendEmailMessage、それにpublishRevenueRecognitionCalculationメソッドをトランザクション可能なものとして宣言すれば得られることになる。POJO例からのRecognitionServiceはそのままRecognitionServiceBeanImplになる。

この例で重要なのは、**サービスレイヤ**が操作のトランザクションのレスポンスを調整するのに操作スクリプトとドメインオブジェクトクラスの両方を使っている点である。calculateRevenueRecognitionsメソッドは、アプリケーションのユースケースで必要なレスポンスのアプリケーションロジックを記述しているが、ドメインロジックはドメインオブジェクトクラスに委譲している。ここでは、**サービスレイヤ**の操作スクリプト内での重複ロジックを防ぐためのいくつかの技法も示している。責任は、委譲を通じて再利用できるように別々のオブジェクト（**ゲートウェイ**など）に組み込まれている。**レイヤスーパータイプ**は、こうしたその他のオブジェクトへのアクセスを簡単なものにしている。

中には、オブザーバーパターン[Gang of Four]を使った方がより洗練された状態で操作スクリプトを実装できるという人もいるかもしれないが、ステートレスでマルチスレッドな**サービスレイヤ**へのオブザーバーの実装は難しいものとなる。私の考えでは、操作スクリプトのオープンコードは、読みやすくシンプルである。

また、アプリケーションロジックの責任は、Contract.calculateRevenueRecognitions()のようなドメインオブジェクトメソッド内や、データソースレイヤ内に実装できるので、個別に**サービスレイヤ**を設ける必要はないという意見もあるようである。しかし、このような責任の配置はいくつかの理由からあまり好ましいものだとは思わない。まず、ドメインオブジェクトクラスはアプリケーション固有のロジックに実装されると（そしてアプリケーション固有の**ゲートウェイ**に依存する場合）、アプリケーション間での再利用性が低下する。これらはアプリケーションが関心のある問題ドメインの一部をモデル化するべきである。これは、アプリケーションのすべてのユースケース責任を意味しているわけではない。次に、アプリケーションロジックをその目的だけのために（データソースレイヤではない）「上位」レイヤにカプセル化すれば、レイヤの実装の変更（たとえば、ワークフローエンジンを使用するなど）が容易になる。

エンタープライズアプリケーションにおけるロジックレイヤの体系化パターンである**サービスレイヤ**は、スクリプティングとドメインオブジェクトクラスを組み合わせ、両方の適切な部分を活用できるようにしている。**サービスレイヤ**の実装においては、さまざまなバリエーションがある。たとえば、ドメインファサードや操作スクリプト、POJOか

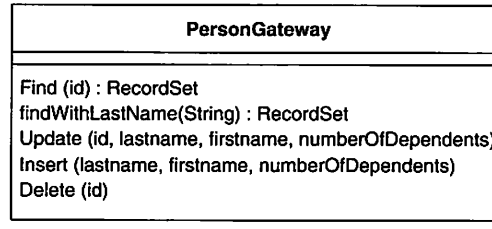
セッションBean、または、それら両方の組み合わせである。**サービスレイヤ**は、ローカル呼び出し、リモート呼び出し、またはその両方に対応するように設計できる。最も重要なのは、このようなバリエーションに関係なく、パターンがアプリケーションのビジネスロジックのカプセル化実装の基礎になり、ロジックを各種クライアントから一貫して呼び出せるという点である。

第10章

データソースのアーキテクチャに関するパターン

10.1 | テーブルデータゲートウェイ

データベーステーブルに対して「ゲートウェイ」の役割を果たすオブジェクト。1 つのインスタンスがテーブル内のすべての行を処理する。



アプリケーションロジックと SQL を組み合わせる場合、さまざまな問題が引き起こされる可能性がある。数多くの開発者は SQL に不慣れであり、たとえ慣れていたとしても、正しく記述できるとは限らない。データベース管理者は、データベースの調整方法と拡張方法を把握するため、SQL を簡単に見つけ出せなければならない。

テーブルデータゲートウェイは、1 つのテーブルまたはビューにアクセス（選択、挿入、更新、削除）するための SQL をすべて保持している。他のコードはそれぞれのメソッドを呼び出すことで、データベースとの相互作用を行う。

10.1.1 | 動作方法

テーブルデータゲートウェイは、データベースからデータを取得するための数種類の find メソッドと、update、insert、delete の各メソッドから構成されるシンプルなインタフェースを備えている。各メソッドは入力引数を SQL 呼び出しにマッピングし、データベース接続に対してその SQL を実行する。**テーブルデータゲートウェイ**は、データの受け

渡しがその役割であるためステートレスである。

テーブルデータゲートウェイの特殊な点は、クエリーからの情報を返す方法にある。シンプルなIDを利用した検索（find-by-ID）クエリーでさえ、複数のデータ項目を返すことができる。複数の項目を返すことが可能な環境においては、単一の行に対してもクエリーを発行することができる。多くの言語では単一の値だけを返すが、ほとんどのクエリーは複数の行を返す。

1つの選択肢としては、マッピングのようなシンプルなデータ構造を返すという方法が考えられる。しかし、マッピングは有効ではあるが、データベースから取り出したレコードセットのデータを強制的にマッピングにコピーしてしまう。マッピングを使ってデータを渡すことは、コンパイル時のチェックを無効にしてしまう上、明示的なインタフェースではないため、ユーザがマッピングの内容の綴りを間違えることでバグにつながる可能性があると、私は考えている。より良い選択肢として、**データ変換オブジェクト**の使用がある。これも作成が必要なオブジェクトではあるが、さまざまな場所で使用することができる。

SQLクエリーによって取り出される**レコードセット**を返すことで、上記2つの選択肢で必要になる労力を省くことが可能になる。理論上、メモリ上のオブジェクトがSQLインタフェースについて一切知る必要がないため、クエリーを発行する方法は概念的にはわかりにくい。また、独自のコードでレコードセットを簡単に作成できない場合、データベースをファイルに置き換えることが難しくなってしまう可能性もある。ただし、.NETなど**レコードセット**を幅広く使用する多くの環境では、これはとても有効な手法である。そのため、**テーブルデータゲートウェイ**は**テーブルモジュール**にとても適している。すべての更新が**テーブルデータゲートウェイ**を介して行われる場合、返されるデータは、実際のテーブルではなくビューに基づいたものになるため、コードとデータベース間の結合が緩和される。

ドメインモデルを使用する場合は、**テーブルデータゲートウェイ**によって適切なドメインオブジェクトを返すことができる。この場合の問題点は、ドメインオブジェクトとゲートウェイの間に双方向の依存性が構成されてしまうことである。双方がとても密接に結び付いているため、私はいつもこの方法をあまり使用したくないと思ってしまう。

テーブルデータゲートウェイを使用する場合のほとんどは、データベースのテーブルごとに**テーブルデータゲートウェイ**を使用することになるだろう。しかし、とてもシンプルなケースに対しては、単一の**テーブルデータゲートウェイ**で、テーブルのすべてのメソッドを処理することができる。さらに、ビューに対して、あるいはデータベースにビューとして保持されていないクエリーに対して、**テーブルデータゲートウェイ**を使用することも可能である。ビューベースの**テーブルデータゲートウェイ**は更新することができず、更新の振る舞いを行わないことが明らかな場合がある。しかし、基盤となるテーブルの更新が可能であれば、**テーブルデータゲートウェイ**上での更新操作の裏側でその更新をカプセル化するという技法は、とても有効である。

10.1.2 | 使用するタイミング

行データゲートウェイと同様に、**テーブルデータゲートウェイ**については、まず**ゲートウェイ**の手法を使用するかどうかを判断し、次にどの手法に移行するかを判断する。

テーブルデータゲートウェイは、データベーステーブルまたはレコードタイプとのマッピングに優れているという点で、最もシンプルなデータベースインタフェースパターンである。また、データソースへの的確なアクセスロジックを自然にカプセル化することができる。私が**テーブルデータゲートウェイ**を使用する場合、必ず**ドメインモデル**とともに使用しているのは、**データマッパー**が**ドメインモデル**とデータベースとの分離を、より確実に実現することに気付いたからである。

テーブルデータゲートウェイは、特に**テーブルモジュール**に最適であり、**テーブルモジュール**が動作するレコードセットデータ構造を作成する。私は**テーブルモジュール**に対して、**テーブルデータゲートウェイ**以外のデータベースのマッピング手法は思い付かない。

行データゲートウェイと同様に、**テーブルデータゲートウェイ**は**トランザクションスクリプト**にとても適している。どちらを選択するかは、それらが複数の行のデータをどう処理するかに左右される。**データ変換オブジェクト**の使用を好む人は少なくないが、同じ**データ変換オブジェクト**が他のどこかで使用されていない限り、あまり意味がないように思える。そのため、私としては結果群が**トランザクションスクリプト**の処理に適している場合には、**テーブルデータゲートウェイ**を好んで使用している。

テーブルデータゲートウェイを介して、**データマッパー**をデータベースと通信させることも、とても興味深い手法である。すべてが手動でコード化されている場合には役に立たないが、**テーブルデータゲートウェイ**にメタデータを使用したいけれど、ドメインオブジェクトへの実際のマッピングには手動コーディングを好むという場合には、とても有効である。

テーブルデータゲートウェイを使用してデータベースのアクセスをカプセル化するメリットの1つとして、同じインタフェースが、SQL によってデータベースを処理する場合にも、ストアドプロシージャを使用する場合にも機能する点が挙げられる。実際、ストアドプロシージャ自体が**テーブルデータゲートウェイ**として体系化される場合も多い。このように、insert および update ストアドプロシージャは、実際のテーブル構造をカプセル化し、find プロシージャはビューを返すことができる。これによって基盤となるテーブル構造を隠ぺいできる。

10.1.3 | 参考文献

[Alur et al.]では**データアクセスオブジェクト**パターンについて紹介しているが、これは**テーブルデータゲートウェイ**のことである。query メソッドを使用して**データ変換オブジェクト**のコレクションを返す方法を示している。このパターンを常にテーブルの基盤として見

ているかどうかは明らかではないが、目的および解説はテーブルデータゲートウェイまたは行データゲートウェイが前提となっていると思われる。

私も異なる名称を使用したことがあるが、それはパターンを、より一般的なゲートウェイの概念の1つを特定の方法で使用した形態と考えたからであり、パターン名にその考えを反映させたかったためである。また、データアクセスオブジェクトという用語および短縮形であるDAOに対して、Microsoftは独自の定義を採用している。

10.1.4 ｜例：PersonGateway（C＃）

テーブルデータゲートウェイは、Windowsの世界でデータベースアクセスによく使用される手法であるため、C#で例を紹介することには意味がある。ただし、典型的なテーブルデータゲートウェイは、ADO.NETデータセットを使用せず、代わりにデータリーダー（データベースレコードのインタフェースとなるカーソル）を使用しているため、.NET環境にはまったく適応しないことは強調しておかなければならない。データリーダーは、メモリに処理のすべてを置かず、大量の情報を処理する場合の優れた選択肢である。

この例では、データベースのpersonテーブルに接続されたPersonGatewayクラスを使用している。PersonGatewayクラスにはFindコードが含まれていて、テーブル内のデータにアクセスするADO.NETのデータリーダーを返している。

```
class PersonGateway...

      public IDataReader FindAll() {
         String sql = "SELECT * FROM person";
         return new OleDbCommand(sql, DB.Connection).ExecuteReader();
      }
      public IDataReader FindWithLastName(String lastName) {
         String sql = "SELECT * FROM person WHERE lastname = ?";
         IDbCommand comm = new OleDbCommand(sql, DB.Connection);
         comm.Parameters.Add(new OleDbParameter("lastname", lastName));
         return comm.ExecuteReader();
      }
      public IDataReader FindWhere(String whereClause) {
         String sql = String.Format("SELECT * FROM person WHERE {0}" ,
            whereClause);
         return new OleDbCommand(sql, DB.Connection).EcecuteReader();
      }
```

ほとんどの場合、リーダーを使用して行の集合を取得しようとする。しかし、稀に行に適したメソッドによって、個々のデータ行を取得したい場合もある。

```
class PersonGateway...

    public Object[] FindRow (long key) {
        String sql = "SELECT * FROM person WHERE id = ?";
        IDbCommand comm = new OleDbCommand(sql, DB.Connection);
        comm.Parameters.Add(new OleDbParameter("key",key));
        IDataReader reader = comm.ExecuteReader();
        reader.Read();
        Object [] result = new Object[reader.FieldCount];
        reader.GetValues(result);
        reader.Close();
        return result;
    }
```

Update および Insert メソッドは必要なデータを引数として取得し、適切な SQL ルーチンを呼び出す。

```
class PersonGateway...

    public void Update (long key, String lastname, String firstname,
        long numberOfDependents){
        String sql = @"
            UPDATE person
                SET lastname = ?, firstname = ?, numberOfDependents = ?
                WHERE id = ?";
        IDbCommand comm = new OleDbCommand(sql, DB.Connection);
        comm.Parameters.Add(new OleDbParameter ("last", lastname));
        comm.Parameters.Add(new OleDbParameter ("first", firstname));
        comm.Parameters.Add(new OleDbParameter ("numDep",
            numberOfDependents));
        comm.Parameters.Add(new OleDbParameter ("key", key));
        comm.ExecuteNonQuery();
    }

class PersonGateway...

    public long Insert(String lastName, String firstName, long
        numberOfDependents) {
        String sql = "INSERT INTO person VALUES (?,?,?,?)";
        long key = GetNextID();
        IDbCommand comm = new OleDbCommand(sql, DB.Connection);
        comm.Parameters.Add(new OleDbParameter ("key", key));
        comm.Parameters.Add(new OleDbParameter ("last", lastName));
```

```
        comm.Parameters.Add(new OleDbParameter ("first", firstName));
        comm.Parameters.Add(new OleDbParameter ("numDep",
            numberOfDependents));
        comm.ExecuteNonQuery();
        return key;
    }
```

Delete メソッドではキーだけを必要とする。

```
class PersonGateway...

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

10.1.5 ｜例：ADO.NET データセット（C#）の使用

汎用な**テーブルデータゲートウェイ**は、SQL 文のラッパーに過ぎないため、どのような種類のプラットフォーム上でも動作する。.NET では、データセットをより頻繁に使用するが、**テーブルデータゲートウェイ**も異なる形式で使用できる。

データセットは、データを読み込んだり、データを更新したりするためのデータアダプターを必要とする。find メソッドでは、データセットとアダプター用のホルダーを定義することが重要である。その後、ゲートウェイはホルダーを使ってデータセットとアダプターを保存する。振る舞いの多くは汎用であり、スーパークラスで実行される。

ホルダーは、テーブル名ごとにデータセットとアダプターをインデックス化する。

```
class DataSetHolder...

    public DataSet Data = new DataSet();
    private Hashtable DataAdapters = new Hashtable();
```

ゲートウェイはホルダーを保存し、クライアントへデータセットを公開する。

```
class DataGateway...

    public DataSetHolder Holder;
    public DataSet Data {
```

```
        get {return Holder.Data;}
    }
```

ゲートウェイは既存のホルダー上でも動作するが、新しいホルダーを作成することもできる。

```
class DataGateway...

    protected DataGateway() {
        Holder = new DataSetHolder();
    }
    protected DataGateway(DataSetHolder holder) {
        this.Holder = holder;
    }
```

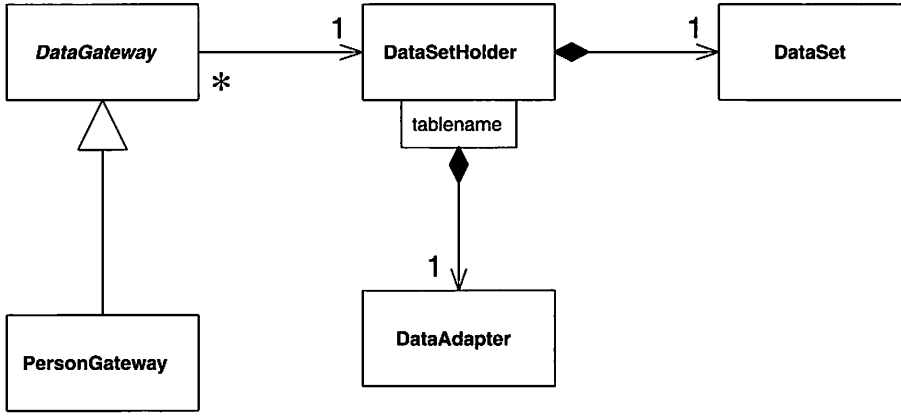


図 10.1 ——データセット指向ゲートウェイ、およびサポートするデータホルダーのクラス図

若干異なったfindの振る舞いを行うこともできる。データセットはテーブル指向データのためのコンテナであり、複数のテーブルのデータを保持することができる。そのため、データをデータセットに読み込むことを勧める。

```
class DataGateway...

    public void LoadAll() {
        String commandString = String.Format("SELECT * FROM
            {0}", TableName);
        Holder.FillData(commandString, TableName);
```

```
    }
    public void LoadWhere(String whereClause) {
        String commandString = String.Format("SELECT * FROM {0}
            where {1}", TableName,whereClause);
        Holder.FillData(commandString, TableName);
    }
    abstract public String TableName {get;}

class PersonGateway...

    public override String TableName {
        get {return "person";}
    }

class DataSetHolder...

    public void FillData(String query, String tableName) {
        if (DataAdapters.Contains(tableName)) throw new
            MutlipleLoadException();
        OleDbDataAdapter da = new OleDbDataAdapter(query,
            DB.Connection);
        OleDbCommandBuilder builder = new OleDbCommandBuilder(da);
        da.Fill(Data, tableName);
        DataAdapters.Add(tableName, da);
    }
```

データを更新するには、クライアントコードでデータセットを直接処理する。

```
person.LoadAll();
person[key]["lastname"] = "Odell";
person.Holder.Update();
```

ゲートウェイは、特定の行へと容易に行けるようにするためのインデクサーを持つことができる。

```
class DataGateway...

    public DataRow this[long key] {
        get {
            String filter = String.Format("id = {0}", key);
            return Table.Select(filter)[0];
        }
```

```
    }
    public override DataTable Table {
       get { return Data.Tables[TableName];}
    }
```

Update メソッドは、ホルダー上での更新動作のトリガーとなる。

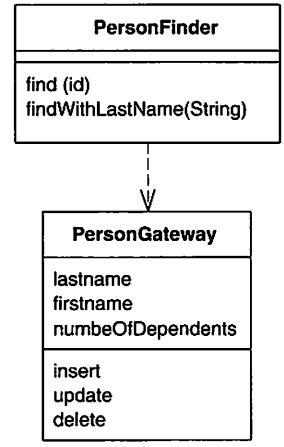
```
class DataSetHolder...

    public void Update() {
       foreach (String table in DataAdapters.Keys)
          ((OleDbDataAdapter)DataAdapters[table]).Update(Data,
          table);
    }
    public DataTable this[String tableName] {
       get {return Data.Tables[tableName];}
    }
```

挿入もまったく同様の方法で行うことが可能である。データセットを取得し、データテーブルに新しい行を挿入し、各列に入力する。ただし、Update メソッドは 1 度の呼び出しで挿入を実行することができる。

```
class DataGateway...

    public long Insert(String lastName, String firstname,
       int numberOfDependents) {
       long key = new PersonGatewayDS().GetNextID();
       DataRow newRow = Table.NewRow();
       newRow["id"] = key;
       newRow["lastName"] = lastName;
       newRow["firstName"] = firstname;
       newRow["numberOfDependents"] = numberOfDependents;
       Table.Rows.Add(newRow);
       return key;
    }
```

10.2 | 行データゲートウェイ

データソース内の1つのレコードに対してゲートウェイの役割を果たすオブジェクト。行ごとに1つのインスタンスが存在する。



メモリ上のオブジェクトにデータベースアクセスコードを組み込むことで、いくつかの弱点が生じることもある。まず、メモリ上のオブジェクトが独自のビジネスロジックを持つ場合、データベース操作コードの追加によって複雑性が増す。メモリ上のオブジェクトがデータベースと結び付いている場合、すべてのデータベースアクセスのため、テストの実行スピードが遅くなり、テストにとても手間がかかる。さらにそれぞれのSQL上の小さな相違によっても、複数のデータベースにアクセスする必要が生じる場合もある。

行データゲートウェイは、レコード構造内のレコードそのもののように見えるが、通常のプログラミング言語のメカニズムを使用してアクセス可能なオブジェクトを提供する。データソースアクセスの詳細はすべてインタフェースの裏側に隠ぺいされる。

10.2.1 | 動作方法

行データゲートウェイは、1つのデータベース行などの、1つのレコードを完全に模倣するオブジェクトとしての役割を果たす。オブジェクト内で、データベースの各列は1つのフィールドとなる。**行データゲートウェイ**は、データソースタイプからメモリ上のタイプへのあらゆる変換を行うが、この変換は極めてシンプルである。パターンは行に関するデータを保持するため、クライアントは**行データゲートウェイ**に直接アクセスできるようになる。ゲートウェイは、データの各行への優れたインタフェースとしての役割を果たす。この手法は、特に**トランザクションスクリプト**に最適である。

行データゲートウェイを使用する場合、このパターンを生成するfind（検索）動作をどこに置くのかという問題に直面する。静的なfindメソッドを使用することは可能だが、ポリモーフィズムを妨げるため、異なるデータソースに対しては別のfindメソッドに置き換えたいと考えるかもしれない。この場合、独立したFindオブジェクトを持つことはできるが、リレーショナルデータベースの各テーブルが、1つのFindクラスと結果に対する1つのGatewayクラスを保持するようになる（図10.2）。

行データゲートウェイと**アクティブレコード**間の相違点を説明することは難しい場合が多い。最も重要なポイントは、何らかのドメインロジックが存在するかどうかである。もし存在するなら**アクティブレコード**である。**行データゲートウェイ**には、データベースアクセスロジックだけが含まれていて、ドメインロジックは含まれていない。

その他のテーブル形式のカプセル化と同様、**行データゲートウェイ**は、テーブルだけでなくビューやクエリーとともに使用できる。更新の場合、基盤となるテーブルの更新が必要となるため、動作はさらに複雑になることが多い。また、同じテーブルに対して2つの**行データゲートウェイ**を実行する場合、更新される2つ目の**行データゲートウェイ**は最初の変更の結果を取り消すことになる。これを防止するための一般的な方法はなく、開発者は仮想的な**行データゲートウェイ**がいかに形成されるかを認識しておくだけである。更新可能なビューでも同様の問題が起こる可能性があるが、もちろん更新操作を実行しないという選択もある。

行データゲートウェイには記述が面倒という傾向があるが、**メタデータマッピング**に基づいてコード生成を実行する場合には優れた選択肢になる。この方法を使用することによって、すべてのデータベースアクセスコードを、自動化されたビルドプロセスで自動的に構築できる。

10.2.2 | 使用するタイミング

以下の2つのステップによって、**行データゲートウェイ**を使用するかどうかを判断する場合が多い。1つ目のステップでゲートウェイを使用するかどうかを判断し、2つ目のステップでは**行データゲートウェイ**と**テーブルデータゲートウェイ**のどちらを使用するかを決定する。

私は、**トランザクションスクリプト**を使用する場合に、**行データゲートウェイ**を使用する頻度が最も高い。この場合、**行データゲートウェイ**でデータベースアクセスコードを適切に抜き出し、別の**トランザクションスクリプト**で容易に再使用できるようにする。

私は、**ドメインモデル**を使用する場合には**行データゲートウェイ**を使用しない。シンプルなマッピングを実行する場合には、コードレイヤを追加しなくても、**アクティブレコード**が同じ役割を果たす。マッピングが複雑な場合には、**データマッパー**が適している。理由は、ドメインオブジェクトはデータベースのレイアウトを知る必要がなく、**データマッパー**がドメインオブジェクトからデータ構造を分離することに優れているからである。もちろん、

データベース構造からドメインオブジェクトを隠ぺいするために**行データゲートウェイ**を使用することもできる。これは、**行データゲートウェイ**を使用してデータベース構造を変更するが、ドメインロジックは変更したくないという場合に有効である。しかし、大規模に行った場合には、ビジネスロジック内のデータ、**行データゲートウェイ**内のデータ、データベー

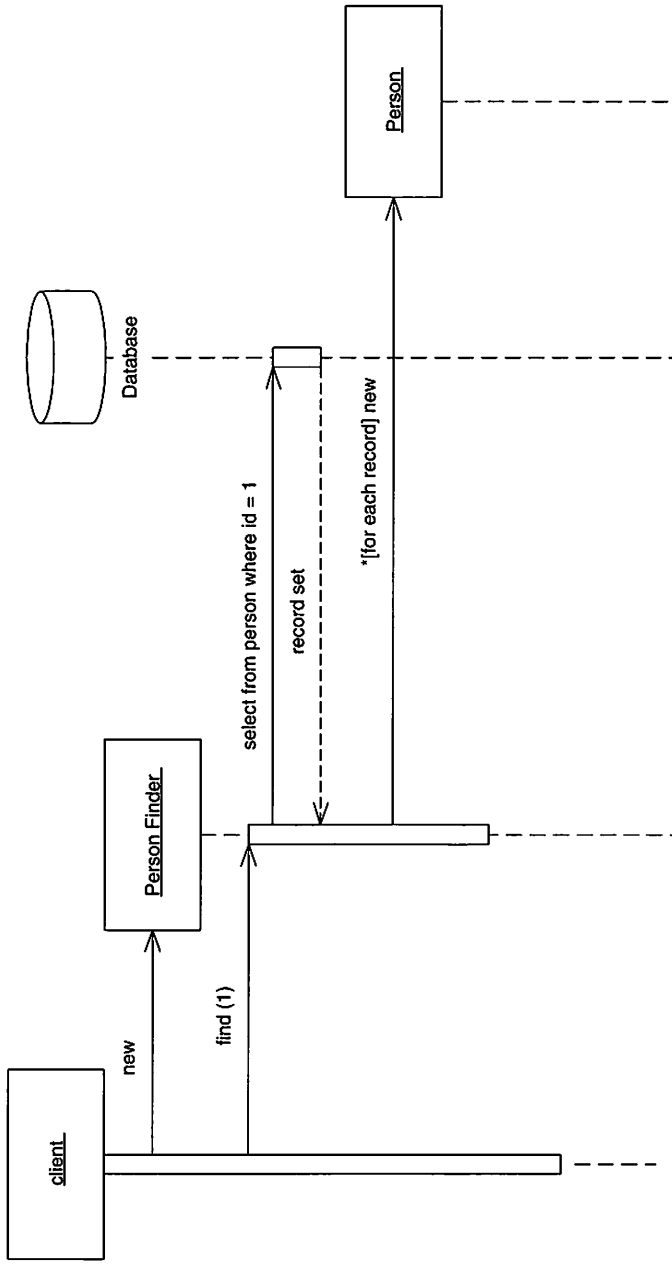


図 10.2——行ベースの行データベースゲートウェイによる検索操作の相互作用

ス内のデータ（極めて大量のデータ）の3種類のデータが表示される。この理由から私は、データベース構造をミラー化する**行データゲートウェイ**を使用する。

面白いことにこれまで私は、**行データゲートウェイ**を**データマッパー**とともに使用することはとても有効であると考えてきた。作業が増えると思われるかもしれないが、**データマッパー**が手動で実行され、**行データゲートウェイ**がメタデータから自動的に生成される場合は、この組み合わせは効果的である。

トランザクションスクリプトを**行データゲートウェイ**とともに使用する場合、複数のスクリプトで繰り返されるビジネスロジックこそが、**行データゲートウェイ**に必要なロジックであることがわかるだろう。ロジックを移動することによって、**行データゲートウェイ**は段階的に**アクティブレコード**へと変化し、ビジネスロジックの重複を軽減する効果をもたらす。

10.2.3 | 例：Person レコード（Java）

以下に**行データゲートウェイ**の例を示す。これはシンプルな Person テーブルである。

```
create table people (ID int primary key, lastname varchar,
                     firstname varchar, number_of_dependents int)
```

PersonGateway は、テーブル用のゲートウェイで、データフィールドとアクセッサーから始まっている。

```
class PersonGateway...

   private String lastName;
   private String firstName;
   private int numberOfDependents;
   public String getLastName() {
      return lastName;
   }
   public void setLastName(String lastName) {
      this.lastName = lastName;
   }
   public String getFirstName() {
      return firstName;
   }
   public void setFirstName(String firstName) {
      this.firstName = firstName;
   }
   public int getNumberOfDependents() {
```

```
        return numberOfDependents;
    }
    public void setNumberOfDependents(int numberOfDependents) {
        this.numberOfDependents = numberOfDependents;
    }
```

Gateway クラス自体が更新と挿入を処理する。

```
class PersonGateway...

    private static final String updateStatementString =
            "UPDATE people " +
            " SET lastname = ?, firstname = ?, number_of_dependents = ? " +
            " WHERE id = ?";
    public void update() {
        PreparedStatement updateStatement = null;
        try {
            updateStatement = DB.prepare(updateStatementString);
            updateStatement.setString(1, lastName);
            updateStatement.setString(2, firstName);
            updateStatement.setInt(3, numberOfDependents);
            updateStatement.setInt(4, getID().intValue());
            updateStatement.execute();
        } catch (Exception e) {
            throw new ApplicationException(e);
        } finally {DB.cleanUp(updateStatement);
        }
    }
    private static final String insertStatementString =
            "INSERT INTO people VALUES (?, ?, ?, ?)";
    public Long insert() {
        PreparedStatement insertStatement = null;
        try {
            insertStatement = DB.prepare(insertStatementString);
            setID(findNextDatabaseId());
            insertStatement.setInt(1, getID().intValue());
            insertStatement.setString(2, lastName);
            insertStatement.setString(3, firstName);
            insertStatement.setInt(4, numberOfDependents);
            insertStatement.execute();
            Registry.addPerson(this);
            return getID();
        } catch (SQLException e) {
```

```
            throw new ApplicationException(e);
        } finally { DB.cleanUp(insertStatement);
        }
    }
```

データベースから複数のPersonを取得するために、独立したPersonFinderを使用する。このPersonFinderをゲートウェイとともに使用することで、新しいGatewayオブジェクトが作成される。

```
class PersonFinder...

    private final static String findStatementString =
            "SELECT id, lastname, firstname, number_of_dependents " +
            " FROM people " +
            " WHERE id = ?";
    public PersonGateway find(Long id) {
        PersonGateway result = (PersonGateway) Registry.getPerson(id);
        if (result != null) return result;
        PreparedStatement findStatement = null;
        ResultSet rs = null;
        try {
            findStatement = DB.prepare(findStatementString);
            findStatement.setLong(1, id.longValue());
            rs = findStatement.executeQuery();
            rs.next();
            result = PersonGateway.load(rs);
            return result;
        } catch (SQLException e) {
            throw new ApplicationException(e);
        } finally {DB.cleanUp(findStatement, rs);
        }
    }
    public PersonGateway find(long id) {
        return find(new Long(id));
    }

class PersonGateway...

    public static PersonGateway load(ResultSet rs) throws SQLException {
            Long id = new Long(rs.getLong(1));
            PersonGateway result = (PersonGateway) Registry.getPerson(id);
            if (result != null) return result;
            String lastNameArg = rs.getString(2);
```

```
            String firstNameArg = rs.getString(3);
            int numDependentsArg = rs.getInt(4);
            result = new PersonGateway(id, lastNameArg,
               firstNameArg, numDependentsArg);
            Registry.addPerson(result);
            return result;
        }
```

いくつかの条件に従って複数のPersonを検索するには、該当するfindメソッドを提供する。

```
class PersonFinder...

   private static final String findResponsibleStatement =
         "SELECT id, lastname, firstname, number_of_dependents " +
         " FROM people " +
         " WHERE number_of_dependents > 0";
   public List findResponsibles() {
      List result = new ArrayList();
      PreparedStatement stmt = null;
      ResultSet rs = null;
      try {
         stmt = DB.prepare(findResponsibleStatement);
         rs = stmt.executeQuery();
         while (rs.next()) {
            result.add(PersonGateway.load(rs));
         }
         return result;
      } catch (SQLException e) {
         throw new ApplicationException(e);
      } finally {DB.cleanUp(stmt, rs);
      }
   }
```

findメソッドはレジストリを使用して、一意マッピングを保持することで、トランザクションスクリプトからこのゲートウェイを使用することができる。

```
      PersonFinder finder = new PersonFinder();
      Iterator people = finder.findResponsibles().iterator();
      StringBuffer result = new StringBuffer();
      while (people.hasNext()) {
```

```
        PersonGateway each = (PersonGateway) people.next();
        result.append(each.getLastName());
        result.append("");
        result.append(each.getFirstName());
        result.append("");
        result.append(String.valueOf(each.getNumberOfDependents()));
        result.append("

    }
    return result.toString();
```

10.2.4 | 例：ドメインオブジェクト用のデータホルダー（Java）

私はトランザクションスクリプトでは、主に行データゲートウェイを使用する。ドメインモデルから行データゲートウェイを使用したい場合、ドメインオブジェクトはゲートウェイからデータを取得する必要がある。データをドメインオブジェクトにコピーする代わりに、行データゲートウェイをドメインオブジェクト用のデータホルダーとして使用することができる。

```
class Person...

   private PersonGateway data;
   public Person(PersonGateway data) {
      this.data = data;
   }
```

その後、ドメインロジックのアクセッサーは、データのゲートウェイに委譲される。

```
class Person...

   public int getNumberOfDependents() {
      return data.getNumberOfDependents();
   }
```

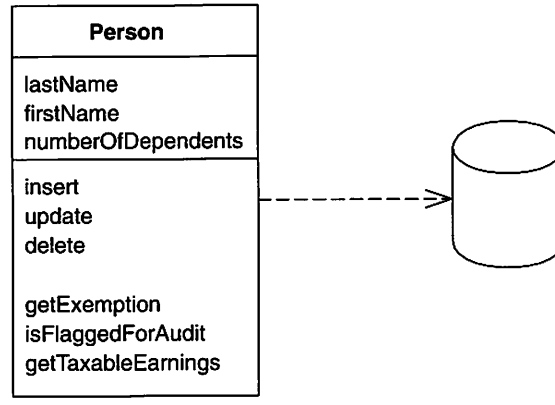
ドメインロジックは get メソッドを使用して、ゲートウェイからデータを取得する。

```
class Person...

   public Money getExemption() {
      Money baseExemption = Money.dollars(1500);
```

```
        Money dependentExemption = Money.dollars(750);
        return baseExemption.add(dependentExemption.multiply
            (this.getNumberOfDependents()));
    }
```

10.3 | アクティブレコード

データベーステーブルまたはビューの行をラップし、データベースアクセスをカプセル化してデータにドメインロジックを追加するオブジェクト。



オブジェクトはデータと振る舞いの両方を持つ。データの大部分は永続データであり、データベースに保存されなければならない。**アクティブレコード**はこれを使ってデータアクセスロジックをドメインオブジェクトに配置する。すべてのユーザがデータベースからデータを読み込み、データベースへデータを書き込む方法を把握できるようになる。

10.3.1 | 動作方法

アクティブレコードの本質は**ドメインモデル**であり、**アクティブレコード**内のクラスは、基盤となるデータベースレコード構造とほぼ一致している。それぞれの**アクティブレコード**はデータベースへの保存や読み込みを行い、またデータに適用されるドメインロジックとしての役割も果たす。ドメインロジックは、アプリケーション内のドメインロジックの場合もあるが、一部のドメインロジックが**アクティブレコード**内のカンマおよびデータ指向のコードとともに、**トランザクションスクリプト**に保持される場合もある。

アクティブレコードのデータ構造は、データベースのデータ構造と完全に一致している必

要がある（つまり、クラス内の1つのフィールドはテーブル内の各列と一致する）。SQL インタフェースからデータを取得する方法を、フィールドに入力しておく（この時点では、いかなる変換も行ってはいけない）。**外部キーマッピング**が気になるかもしれないが、外部キーはそのままにしておいても構わない。ビューを介した更新はかなり困難ではあるが、**アクティブレコード**とともにビューまたはテーブルを使用することができる。特に、ビューはレポートを目的としている場合には有効である。

アクティブレコードのクラスは、以下の処理を行うメソッドを持っている。

- SQL の結果群の行から**アクティブレコード**のインスタンスを構築する。
- 後でテーブルへの挿入を行うため、新しいインスタンスを構築する。
- 静的な find メソッドを使用して、共通の SQL クエリーをラップし、**アクティブレコード**オブジェクトを返す。
- データベースを更新し、更新したデータベースに**アクティブレコード**のデータを挿入する。
- フィールドを取得し、設定する。
- ビジネスロジックの一部を実装する。

get メソッドと set メソッドでは、SQL 指向タイプからより優れたメモリ上のタイプへの変換などの、より高度な操作を行うことも可能である。また、関連テーブルを要求する場合、このデータ構造で**一意フィールド**を使用しなくても、（照合によって）get メソッドは適切な**アクティブレコード**を返す。

このパターンでは、クラスは便利な存在ではあるが、リレーショナルデータベースがあるという事実は隠さない。結果として**アクティブレコード**を使用する場合には、通常、その他のオブジェクトリレーショナルマッピングパターンを目にすることはほとんどなくなる。

アクティブレコードは**行データゲートウェイ**にとても類似している。最大の相違は、**行データゲートウェイ**にはデータベースアクセスだけが含まれ、**アクティブレコード**にはデータソースとドメインロジックの両方が含まれる点である。もっとも、ほとんどのソフトウェア境界のように、両方の間に完全な境界線を設定することは難しくまた意味もない。

アクティブレコードとデータベースとが密接に結合しているため、パターンでは静的な find メソッドを使用することが多い。しかし、**行データゲートウェイ**の項でも述べたとおり、find メソッドを独立したクラスに分離できないという訳ではなく、テストでは分離した方が有効である。

他のテーブル形式パターンと同様、**アクティブレコード**は、テーブルだけでなくビューやクエリーとともに使用できる。

10.3.2 | 使用するタイミング

アクティブレコードは作成、読み込み、更新、削除など、それほど複雑ではないドメインロジックに対しては優れた選択肢である。構造内では、1つのレコードに基づいた派生操作や妥当性確認を効果的に行える。

ドメインモデルの初期設計で最初に判断するべき選択は、**アクティブレコード**か**データマッパー**のいずれを使用するかである。**アクティブレコード**の最大のメリットは、シンプルな構造である。**アクティブレコード**の構築は容易であり、また理解もしやすい。最大の問題は、**アクティブレコード**が有効であるのが、**アクティブレコード**オブジェクトがデータベーステーブルと直接対応している（同一構造スキーム）場合だけという点である。

ビジネスロジックが複雑な場合には、オブジェクトの直接的な関係、コレクション、継承などを使用したいとまず考えるだろう。しかし、これらの部品は簡単には**アクティブレコード**にマッピングできず、また、断片的に追加すると状況はより複雑になる。以上の理由から、**データマッパー**の使用を考えるようになる。

アクティブレコードに関する別の問題点として、オブジェクト設計とデータベース設計とが結合しているので、プロジェクトが進行するにつれて、それぞれの設計分析が困難になってしまうことがある。

トランザクションスクリプトを使用したがコードの重複による弊害が大きく、**トランザクションスクリプト**で行われる多くのスクリプトやテーブルの更新に困難を感じるようになった場合には、**アクティブレコード**は考慮すべき優れたパターンである。この場合には、**アクティブレコード**の作成を段階的に開始した後、ゆっくりとその振る舞いを分析することができる。この方法は**ゲートウェイ**としてテーブルをラップするために役立つことが多く、次に振る舞いの移動を開始してテーブルが**アクティブレコード**へと発展する際に役立つ。

10.3.3 | 例： シンプルな Person（Java）

これは**アクティブレコード**の構造がどのように動作するかを示すシンプルな例である。まずは、基本的な Person クラスから始まる。

```
class Person...

   private String lastName;
   private String firstName;
   private int numberOfDependents;
```

また、スーパークラスには一意フィールドも存在している。データベースはまったく同じ構造に設定されている。

```
create table people (ID int primary key, lastname varchar,
                     firstname varchar, number_of_dependents int)
```

オブジェクトを読み込むために、Person クラスは find メソッドとしての役割を果たし、また読み込みも実行する。この例では、Person クラス上で静的メソッドを使用している。

```
class Person...

   private final static String findStatementString =
         "SELECT id, lastname, firstname, number_of_dependents" +
         " FROM people" +
         " WHERE id = ?";
   public static Person find(Long id) {
      Person result = (Person) Registry.getPerson(id);
      if (result != null) return result;
      PreparedStatement findStatement = null;
      ResultSet rs = null;
      try {
         findStatement = DB.prepare(findStatementString);
         findStatement.setLong(1, id.longValue());
         rs = findStatement.executeQuery();
         rs.next();
         result = load(rs);
         return result;
      } catch (SQLException e) {
         throw new ApplicationException(e);
      } finally { DB.cleanUp(findStatement, rs);
      }
   }
   public static Person find(long id) {
      return find(new Long(id));
   }
   public static Person load(ResultSet rs) throws SQLException {
      Long id = new Long(rs.getLong(1));
      Person result = (Person) Registry.getPerson(id);
      if (result != null) return result;
      String lastNameArg = rs.getString(2);
      String firstNameArg = rs.getString(3);
      int numDependentsArg = rs.getInt(4);
      result = new Person(id, lastNameArg, firstNameArg, numDependentsArg);
      Registry.addPerson(result);
      return result;
   }
```

オブジェクトの更新には、シンプルなインスタンスメソッドを使用している。

```
class Person...

   private final static String updateStatementString =
         "UPDATE people" +
         " SET lastname = ?, firstname = ?, number_of_dependents = ?" +
         " WHERE id = ?";
   public void update() {
      PreparedStatement updateStatement = null;
      try {
         updateStatement = DB.prepare(updateStatementString);
         updateStatement.setString(1, lastName);
         updateStatement.setString(2, firstName);
         updateStatement.setInt(3, numberOfDependents);
         updateStatement.setInt(4, getID().intValue());
         updateStatement.execute();
      } catch (Exception e) {
         throw new ApplicationException(e);
      } finally {
         DB.cleanUp(updateStatement);
      }
   }
```

挿入もまた極めてシンプルである。

```
class Person...

   private final static String insertStatementString =
         "INSERT INTO people VALUES (?, ?, ?, ?)";
   public Long insert() {
      PreparedStatement insertStatement = null;
      try {
         insertStatement = DB.prepare(insertStatementString);
         setID(findNextDatabaseId());
         insertStatement.setInt(1, getID().intValue());
         insertStatement.setString(2, lastName);
         insertStatement.setString(3, firstName);
         insertStatement.setInt(4, numberOfDependents);
         insertStatement.execute();
         Registry.addPerson(this);
         return getID();
      } catch (Exception e) {
```

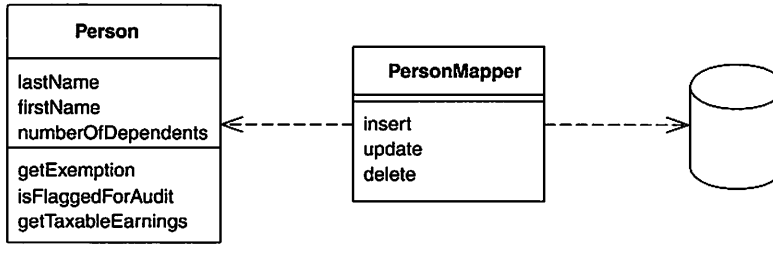
```
            throw new ApplicationException(e);
        } finally {
            DB.cleanUp(insertStatement);
        }
    }
```

控除の計算などのビジネスロジックは、Person クラスに直接置かれている。

```
class Person...

    public Money getExemption() {
        Money baseExemption = Money.dollars(1500);
        Money dependentExemption = Money.dollars(750);
        return baseExemption.add(dependentExemption.multiply
            (this.getNumberOfDependents()));
    }
```

10.4 | データマッパー

オブジェクト、データベース、およびマッパー自体の独立性を保ちつつ、オブジェクトとデータベース間でデータを移動するマッパーのレイヤ。



データの構造化に対して、オブジェクトとリレーショナルデータベースは異なるメカニズムを持っている。コレクションや継承など、オブジェクトの部品の多くはリレーショナルデータベースにはない。多くのビジネスロジックを持ったオブジェクトモデルを構築する場合、これらのメカニズムを使用して、データとそのデータに適用される振る舞いを適切に体系化することは重要である。体系化することによって、さまざまなスキーマも整理することができ、オブジェクトスキーマとリレーショナルスキーマを混同することがなくなる。

2つのスキーマ間でデータを変換する必要があり、データ変換はそれ自体が状況を複雑化する原因である。メモリ上のオブジェクトがリレーショナルデータベース構造を知っている

場合、1つのオブジェクトの変換が他のオブジェクトにも波及する傾向がある。

データマッパーは、メモリ上のオブジェクトをデータベースから分離するソフトウェアのレイヤである。**データマッパー**の役割は、オブジェクトとリレーショナルデータベースの間でデータを変換することであり、両方の独立性も保つことでもある。**データマッパー**を使用すれば、メモリ上のオブジェクトはデータベースが存在するかどうかさえも知る必要はなく、SQL インタフェースコードを使用したり、データベーススキーマの知識を保持したりする必要も一切ない（データベーススキーマは、使用するオブジェクトについて何も知らない）。**データマッパー**は**マッパー**の一種であるため、それ自体がドメインレイヤにとっては未知の存在である。

10.4.1 ｜動作方法

ドメインとデータソースの分離は**データマッパー**の主要な機能であるが、実現するためには、説明しなければならない細かな問題が数多く存在する。また、マッピングレイヤの構築にもさまざまな方法がある。双方を分離するための必要事項についての概要を紹介することが目的であるため、解説する内容も多岐に渡っている。

まずは、基本的な**データマッパー**から始めよう。**データマッパー**は最もシンプルなスタイルのレイヤであり、あまり価値があるようには思えないかもしれない。このシンプルなデータベースマッピングの例では、通常**データマッパー**以外のパターンはよりシンプルで、より優れたものになっている。本格的に**データマッパー**を使用するなら、より複雑なケースが必要になるだろう。しかし、こうした基本的なレベルから解説を始めることで、基本理念の解説が容易となる。

最もシンプルなケースでは、Person クラスと PersonMapper クラスを構築する。データベースから Person を読み込むため、クライアントはマッパーの find メソッドを呼び出す（図 10.3）。マッパーは**一意マッピング**を使用して、その Person がすでに読み込まれているかどうかを判断し、読み込まれていなければ読み込む。

図 10.4 に、更新操作を示す。クライアントはマッパーに対して、ドメインオブジェクトの保存を要求する。マッパーはドメインオブジェクトからデータを取得し、データベースへと戻す。

テストのため、あるいは1つのドメインレイヤが別のデータベースとともに動作するように、**データマッパー**のレイヤ全体を置き換えることができる。

シンプルな**データマッパー**では、データベーステーブルをフィールド単位で、対応するメモリ上のクラスにマッピングするだけである。もちろん、常にシンプルというわけではない。複数のフィールドに変化するクラス、複数のテーブルを持つクラス、継承を持つクラスを処理し、一度これらのクラスを整理した後に、オブジェクトと連結させようとする場合には、

マッパーにさまざまなストラテジーが必要となる。本書で紹介するさまざまなオブジェクトリレーショナルマッピングパターンはすべて、このストラテジーに関するものである。**データマッパー**によるこうしたパターンの配置は、体系化するために別の選択肢を使用するよりも容易である。

挿入や更新を行う場合、データベースマッピングレイヤは、どのオブジェクトが変化し、どのオブジェクトが新たに作成され、どのオブジェクトが破壊されたかを知っておく必要がある。さらにはすべての作業負荷が、トランザクションに関するフレームワークに適応していなければならない。**ユニットオブワーク**パターンは、こうした体系化を行うための優れた方法である。

図 10.3 には、find メソッドへの 1 つのリクエストが、1 つの SQL クエリー内に結果を返す方法を示している。この図の方法が常に正しいとは限らない。複数の Order Line（注文品目）を持った典型的な Order の読み込みには、Order Line の読み込みも含まれる可能性がある。通常クライアントからのリクエストは、マッパーの設計者が 1 回に返す量を決めることで、読み込むオブジェクトの量を調整することができるようになる。重要な点は最小限のデータベースクエリーを設定することである。実現するには一般的に、データの最適な抽出方法を選択するため、find メソッドはクライアントがオブジェクトをどのように使用するかを詳しく知る必要がある。

この例は、1 つのクエリーからドメインオブジェクトの複数クラスを読み込むケースに発展することができる。Order および Order Line を読み込みたいのならば、Orders テーブルと Order Lines テーブルとのジョインを行う 1 つのクエリーを発行したほうが迅速に行える。次に、この結果群を使用して、Order と Order Line インスタンスを読み込む(P261-262 参照)。

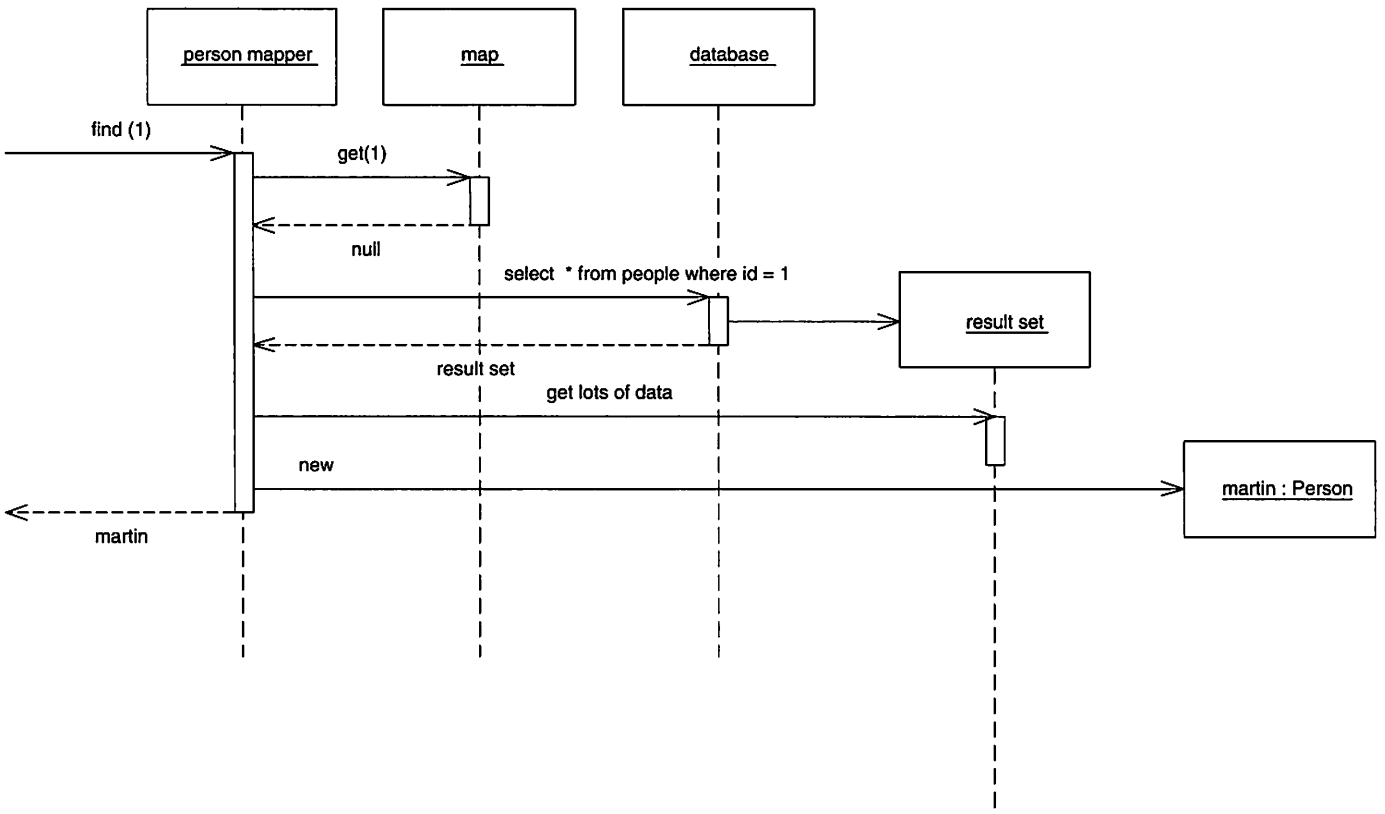


図10.3 ―― 1つのデータベースでのデータの検索

オブジェクトは相互に関連付けられているため、ある時点でデータの抽出を中止しなければならない。そうでない場合は、1つのリクエストによってデータベース全体を抜き出すことになってしまう。マッピングレイヤは、**レイジーロード**と呼ばれる技法によって、メモリ上のオブジェクトへの影響を最小限に抑えながら、抽出に関する問題に対応する。解説してきたとおり、メモリ上のオブジェクトはマッピングレイヤを完全に無視することはできない。マッピングレイヤは、find メソッドや別のいくつかのメカニズムについても知っておく必要がある。

アプリケーションは、1つまたは複数の**データマッパー**を持つことができる。マッパーをハードコーディングする場合、各ドメインクラスまたはドメイン階層構造のルートに、1つのマッパーを使用することが最も有効である。**メタデータマッパー**を使用する場合には、1つの Mapper クラスで対応することもできる。後者の場合、問題になるのは find メソッドである。大規模なアプリケーションの場合、1つのマッパーに多くの find メソッドを持たせることは難しいため、メソッドを各ドメインクラスまたはドメイン階層構造のルートごとに分割する方が意味がある。小規模な Find クラスを数多く持つことになるが、開発者にとって、必要な find メソッドの検索は難しいことではない。

あらゆるデータベース検索の振る舞いと同様、find メソッドがデータベースから読み込まれるオブジェクトの一意性を維持するためには、**一意マッピング**を使用する必要がある。**一意マッピングのレジストリ**を使用することも、あるいは各 find メソッドに**一意マッピング**を保持させることも可能である（1セッションのクラスごとに1つの find メソッドだけが存在する場合）。

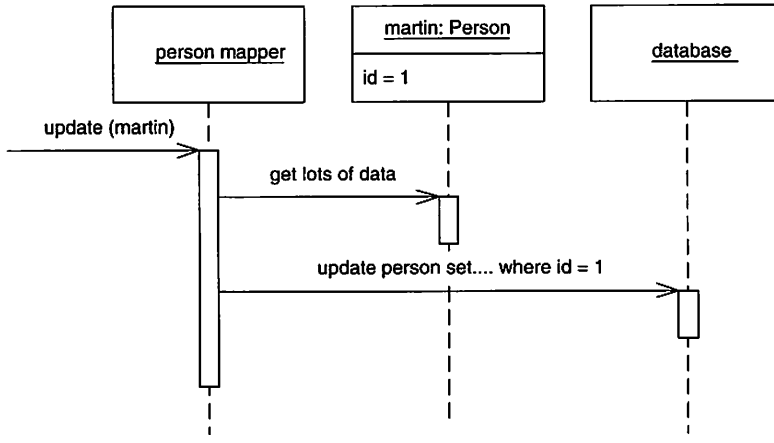


図 10..4 ——データの更新

10.4.1.1 ■ find メソッドの処理

オブジェクトを処理するためには、データベースからオブジェクトを読み込む必要がある。一般的に、プレゼンテーションレイヤはいくつかの初期オブジェクトを読み込むことで初期化を行う。次に、制御はドメインレイヤに移動し、コードは主にオブジェクト間の関連を使用してそのオブジェクトの間を移動する。この方法は、メモリへの読み込みが必要なすべてのオブジェクトをドメインレイヤが持っているか、あるいは必要な時点で追加オブジェクトを読み込む**レイジーロード**を使用している場合には効果的である。

データマッパーで find メソッドを呼び出すためにドメインオブジェクトを必要とする場合がある。しかし私は、優れた**レイジーロード**を使用した、ドメインオブジェクトを一切必要としない方法があることに気付いている。ただしよりシンプルなアプリケーションの場合、関連性と**レイジーロード**を使用して、すべてを管理することに意味がない場合がある。それでもなお、ドメインオブジェクトの依存性を**データマッパー**に追加したいとは考えないであろう。

このジレンマは、**セパレートインタフェース**を使用することによって解決できる。ドメインコードが必要とする任意の find メソッドを、ドメインパッケージに配置できるインタフェースクラスに追加するのである。

10.4.1.2 ■ ドメインフィールドへのデータのマッピング

マッパーはドメインオブジェクトのフィールドにアクセスする必要がある。しかし、マッパーをドメインロジックに対して動作させたくないため、サポートする public メソッドが必要となるので、このマッピングは問題になる場合が多い。(フィールドをパブリックにするといった大罪は問題にしないことにしよう)。この問題の回答は、簡単には見つけられない。Java の同一パッケージのように、ドメインオブジェクトの近くにマッパーをパッケージングするといった、低いレベルの可視性を使用することは可能である。しかし、ドメインオブジェクトを知っているシステムの別の部分にマッパーを知らせたくないため、依存性の問題がより重大になってくる。言語の可視性の規則を回避するためにリフレクションを使用することができる。速度は遅くなるが、この速度の遅さは SQL 呼び出しに要する時間と比較した場合、丸め誤差とすることができる。あるいは public メソッドを使用することもできるが、public メソッドがデータベースの読み込みというコンテキストの外から呼び出されてしまった場合に例外を発生させるには、ステータスフィールドによる保護が必要となる。この場合には、通常の get メソッドや set メソッドが間違わないような名前を付けておく。

問題となるのは、オブジェクトをいつ作成するかという点である。実際には、2 つの選択肢が考えられる。1 つ目の方法は **rich constructor** によってオブジェクトを作成する方法であり、この場合、少なくとも必須データとともにオブジェクトが作成される。もう 1 つの方法は、空のオブジェクトを作成し、そこに必須データを配置する方法である。私は前者を好

むが、その理由は最初から完成度の高いオブジェクトを作成できるからである。さらに不変フィールドが存在する場合、その値を変更するメソッドを提供しないことで、強制的に変更させないというメリットもある。

rich constructor の問題は、循環参照に注意しなければならないという点である。相互に参照し合う2つのオブジェクトがある場合、いずれか1つを読み込もうとするともう1つを読み込むことになり、その結果再び最初の1つを読み込むことになり、最終的にスタックスペースが不足してしまうのである。こうした状況を回避するには特定の case コードが必要となるため、**レイジーロード**が使用される場合が多い。この特定の case コードの記述は困難な作業であり、できれば避ける方が望ましい。記述作業を避けるためには、**空のオブジェクト**を作成する。引数なしの constructor を使って空白のオブジェクトを作成し、その空のオブジェクトを即座に**一意マッピング**に挿入するのである。この方法によって循環が存在する場合でも、**一意マッピング**がオブジェクトを返し、再帰的な読み込みを中止することができる。

空のオブジェクトを使用するということは、オブジェクトを読み込む際に、完全に不変であるいくつかの値にも set メソッドが必要になる。命名規則と、いくつかのステータスチェックによる保護を組み合わせることで、この問題を解決できる。また、データ読み込みにはリフレクションも使用することができる。

10.4.1.3 ■ メタデータベースのマッピング

決定すべき判断の1つとして、ドメインオブジェクトのフィールドをデータベース列にマッピングする方法に関する情報の保存が挙げられる。そのための最もシンプルかつ最善（多くの場合）の方法とは、明示的なコードを使用して、ドメインオブジェクトごとに Mapper クラスを要求することである。マッパーは引数を介してマッピングを行い、フィールド（通常は定数文字列）にデータベースアクセス用の SQL を保存させる。その他の方法としては**メタデータマッピング**が挙げられる、クラスまたは独立したファイルにメタデータをデータとして保存するという方法である。メタデータのメリットとは、コード生成またはリフレクティブプログラミングを使用することで、データを介してマッパーのすべての変数を操作でき、ソースコードを必要としない点である。

10.4.2 | 使用するタイミング

データマッパーを使用する最も一般的な状況は、データベーススキーマとオブジェクトモデルを個別に発展させたい場合である。その最も一般的なケースが**ドメインモデル**である。**データマッパー**の第一のメリットはドメインモデルに対して動作する場合、設計、構築、テストプロセスのいずれにおいても、データベースを無視できる点にある。すべての通信はマッパーによって行われるため、ドメインオブジェクトはデータベース構造については一切

関知していない。

ドメインオブジェクトがデータベースにどのように保存されているかを理解していなくても、ドメインオブジェクトを理解し対処できるため、この状況はコード内で役に立つ。**ドメインモデル**およびデータベースを変更することなく、修正を行うことができるのである。マッピングが複雑な場合、特に既存のデータベースがあるときにはとても有効である。

もちろん**アクティブレコード**では不要である別のレイヤの分の代償が必要になるため、これらのパターンを使用するかどうかはビジネスロジックの複雑性に左右される。ビジネスロジックがとてもシンプルなら、**ドメインモデル**も**データマッパー**も必要ないかもしれない。ビジネスロジックの複雑性によって、**ドメインモデル**やさらには**データマッパー**が必要かどうか決まる。

私は、**ドメインモデル**を使用することなく**データマッパー**を選ぶことはないだろう。しかし、**データマッパー**なしで**ドメインモデル**を使用できるだろうか。ドメインモデルがシンプルで、そのドメインモデルの設計者がデータベースを制御できているなら、**アクティブレコード**を使用して、ドメインオブジェクトがデータベースに直接アクセスすることも十分考えられる。

直接アクセスすることによって、ここで解説したマッパーの振る舞いがドメインオブジェクト自体に効果的に動作するようになる。状況がより複雑な場合、データベースの振る舞いを独立したレイヤにリファクタリングすることを勧める。

完全な機能を備えたデータベースマッパーレイヤを構築する必要がないことは覚えておくべきである。このレイヤは構築するには極めて複雑である上、代用可能な製品も存在している。通常の場合、自身で構築するよりは、データベースマッピングレイヤを購入することを勧めたい。

10.4.3 ｜例： シンプルなデータマッパー（Java）

解説するのは**データマッパー**の極めてシンプルな使用例であり、目的は基本構造を伝えることにある。この例は、同一の People テーブルを持つ Person である。

```
class Person...

   private String lastName;
   private String firstName;
   private int numberOfDependents;
```

データテーブルスキーマは、以下のとおりである。

```
create table people (ID int primary key, lastname varchar,
                     firstname varchar, number_of_dependents int)
```

PersonMapperクラスがfindメソッドと**一意マッピング**を実装するという、シンプルなケースを使用している。ただし、共通の振る舞いをどこで引き出せるかを示すため、抽象マッパーである**レイヤスーパータイプ**を追加した。読み込みには、オブジェクトがすでに**一意マッピング**に含まれていることのチェックと、データベースからのデータの抽出も含まれている。

findメソッドの振る舞いはPersonMapper内で開始される。PersonMapperは、IDで検索するための抽象的なfindメソッドの呼び出しをラップする。

```
class PersonMapper...

   protected String findStatement() {
      return "SELECT " + COLUMNS +
         " FROM people" +
         " WHERE id = ?";
   }
   public static final String COLUMNS = " id, lastname,
      firstname, number_of_dependents ";
   public Person find(Long id) {
      return (Person) abstractFind(id);
   }
   public Person find(long id) {
      return find(new Long(id));
   }

class AbstractMapper...

   protected Map loadedMap = new HashMap();
   abstract protected String findStatement();
   protected DomainObject abstractFind(Long id) {
      DomainObject result = (DomainObject) loadedMap.get(id);
      if (result != null) return result;
      PreparedStatement findStatement = null;
      try {
         findStatement = DB.prepare(findStatement());
         findStatement.setLong(1, id.longValue());
         ResultSet rs = findStatement.executeQuery();
         rs.next();
         result = load(rs);
```

```
            return result;
        } catch (SQLException e) {
            throw new ApplicationException(e);
        } finally {
            DB.cleanUp(findStatement);
        }
    }
```

find メソッドは load メソッドを呼び出し、load メソッドは AbstractMapper と PersonMapper とに分割されている。AbstractMapper は ID をチェックし、ID をデータから抽出し、**一意マッピング**に新たなオブジェクトを登録する。

```
class AbstractMapper...

    protected DomainObject load(ResultSet rs) throws SQLException {
        Long id = new Long(rs.getLong(1));
        if (loadedMap.containsKey(id)) return (DomainObject)
            loadedMap.get(id);
        DomainObject result = doLoad(id, rs);
        loadedMap.put(id, result);
        return result;
    }
    abstract protected DomainObject doLoad(Long id, ResultSet rs)
        throws SQLException;

class PersonMapper...

    protected DomainObject doLoad(Long id, ResultSet rs) throws
        SQLException {
        String lastNameArg = rs.getString(2);
        String firstNameArg = rs.getString(3);
        int numDependentsArg = rs.getInt(4);
        return new Person(id, lastNameArg, firstNameArg,
            numDependentsArg);
    }
```

一意マッピングが2回チェックされている点に注目してほしい（1回目は abstractFind、2回目は load によって行われる）。このような方法を採るには理由がある。

オブジェクトがすでに存在している場合は、データベースのアクセスを省略できるため、find メソッドによってマッピングをチェックしている（常に、私はこの方法で不要なアクセスを回避している）。ただし、**一意マッピング**で解決できるとは限らないクエリーが存在す

るかもしれないので、loadメソッドにおいても同様のチェックを行わなければならない。たとえば、姓（last name）と検索パターンとが一致するすべてのPersonを検索するとしよう。この場合、その複数のPersonがすべて読み込まれているかどうかは分からないため、データベースにアクセスして、クエリーを発行する必要がある。

```
class PersonMapper...

   private static String findLastNameStatement =
         "SELECT " + COLUMNS +
         " FROM people " + " WHERE UPPER(lastname) LIKE UPPER(?)" +
         " ORDER BY lastname";
   public List findByLastName(String name) {
      PreparedStatement stmt = null;
      ResultSet rs = null;
      try {
         stmt = DB.prepare(findLastNameStatement);
         stmt.setString(1, name);
         rs = stmt.executeQuery();
         return loadAll(rs);
      } catch (SQLException e) {
         throw new ApplicationException(e);
      } finally {
         DB.cleanUp(stmt, rs);
      }
   }

class AbstractMapper...

   protected List loadAll(ResultSet rs) throws SQLException {
      List result = new ArrayList();
      while (rs.next())
         result.add(load(rs));
      return result;
   }
```

しかし、その結果として、結果群にはすでに読み込まれている複数のPersonに対応する行も含まれる可能性もある。このような重複がないかどうかをチェックするために、一意マッピングをもう一度チェックする必要がある。

この方法で必要なfindメソッドを各サブクラスに記述することによって、いくつかの基

本コードが繰り返し含まれてしまうが、繰り返しは一般的なメソッドを使用することで解決できる。

```
class AbstractMapper...

   public List findMany(StatementSource source) {
      PreparedStatement stmt = null;
      ResultSet rs = null;
      try {
         stmt = DB.prepare(source.sql());
         for (int i = 0; i < source.parameters().length; i++)
            stmt.setObject(i+1, source.parameters()[i]);
         rs = stmt.executeQuery();
         return loadAll(rs);
      } catch (SQLException e) {
         throw new ApplicationException(e);
      } finally {
         DB.cleanUp(stmt, rs);
      }
   }
```

このコードを機能させるには、SQL 文字列とパラメータの双方の読み込みをあらかじめ準備されたステートメントにラップするインタフェースが必要となる。

```
interface StatementSource...

   String sql();
   Object[] parameters();
```

次に、内部クラスとして適切に実装することで動作が機能するようになる。

```
class PersonMapper...

   public List findByLastName2(String pattern) {
      return findMany(new FindByLastName(pattern));
   }
   static class FindByLastName implements StatementSource {
      private String lastName;
      public FindByLastName(String lastName) {
         this.lastName = lastName;
      }
```

```
    public String sql() {
        return
            "SELECT " + COLUMNS +
            " FROM people " +
            " WHERE UPPER(lastname) LIKE UPPER(?)" +
            " ORDER BY lastname";
    }
    public Object[] parameters() {
        Object[] result = {lastName};
        return result;
    }
}
```

上記のような方法は、繰り返されるステートメント呼び出しコードが存在する他の部分にも応用できる。以下に示す例は、内容が一目でわかるように極めてシンプルな構造になっている。繰り返される単純な大量のコードを記述する場合、上記の方法を検討すべきだろう。

update メソッドでの JDBC コードは、このサブタイプ固有のものである。

```
class PersonMapper...

    private static final String updateStatementString =
            "UPDATE people " +
            " SET lastname = ?, firstname = ?, number_of_dependents = ? " +
            " WHERE id = ?";
    public void update(Person subject) {
        PreparedStatement updateStatement = null;
        try {
            updateStatement = DB.prepare(updateStatementString);
            updateStatement.setString(1, subject.getLastName());
            updateStatement.setString(2, subject.getFirstName());
            updateStatement.setInt(3, subject.getNumberOfDependents());
            updateStatement.setInt(4, subject.getID().intValue());
            updateStatement.execute();
        } catch (Exception e) {
            throw new ApplicationException(e);
        } finally {
            DB.cleanUp(updateStatement);
        }
    }
```

insert メソッドでは、一部のコードを**レイヤスーパータイプ**に組み込むことができる。

```
class AbstractMapper...

   public Long insert(DomainObject subject) {
      PreparedStatement insertStatement = null;
      try {
         insertStatement = DB.prepare(insertStatement());
         subject.setID(findNextDatabaseId());
         insertStatement.setInt(1, subject.getID().intValue());
         doInsert(subject, insertStatement);
         insertStatement.execute();
         loadedMap.put(subject.getID(), subject);
         return subject.getID();
      } catch (SQLException e) {
         throw new ApplicationException(e);
      } finally {
         DB.cleanUp(insertStatement);
      }
   }
   abstract protected String insertStatement();
   abstract protected void doInsert(DomainObject subject,
      PreparedStatement insertStatement) throws SQLException;

class PersonMapper...

   protected String insertStatement() {
      return "INSERT INTO people VALUES (?, ?, ?, ?)";
   }
      protected void doInsert(
      DomainObject abstractSubject,
      PreparedStatement stmt)
      throws SQLException
   {
      Person subject = (Person) abstractSubject;
      stmt.setString(2, subject.getLastName());
      stmt.setString(3, subject.getFirstName());
      stmt.setInt(4, subject.getNumberOfDependents());
   }
```

10.4.4 ｜例：findメソッドの分離（Java）

ドメインオブジェクトがfindメソッドの振る舞いを呼び出せるようにするには、**セパレートインタフェース**を使って、findインタフェースをマッパーから分離する（図10.5）。findインタフェースを、ドメインレイヤから見える独立したパッケージに配置することは

可能であり、この例ではドメインレイヤ自身に配置する。

最も汎用な find メソッドの 1 つは、特定の代理 ID に応じてオブジェクトを検索するものである。処理の大部分は汎用であり、適切な**レイヤスーパータイプ**によって処理することができる。この場合には、ID について知っているドメインオブジェクトの**レイヤスーパータイプ**が必要になる。

検索のためのインタフェースは、find インタフェース内に存在する。通常は、どんなタイプが返されるかを知る必要があるため、汎化しないことが望まれる。

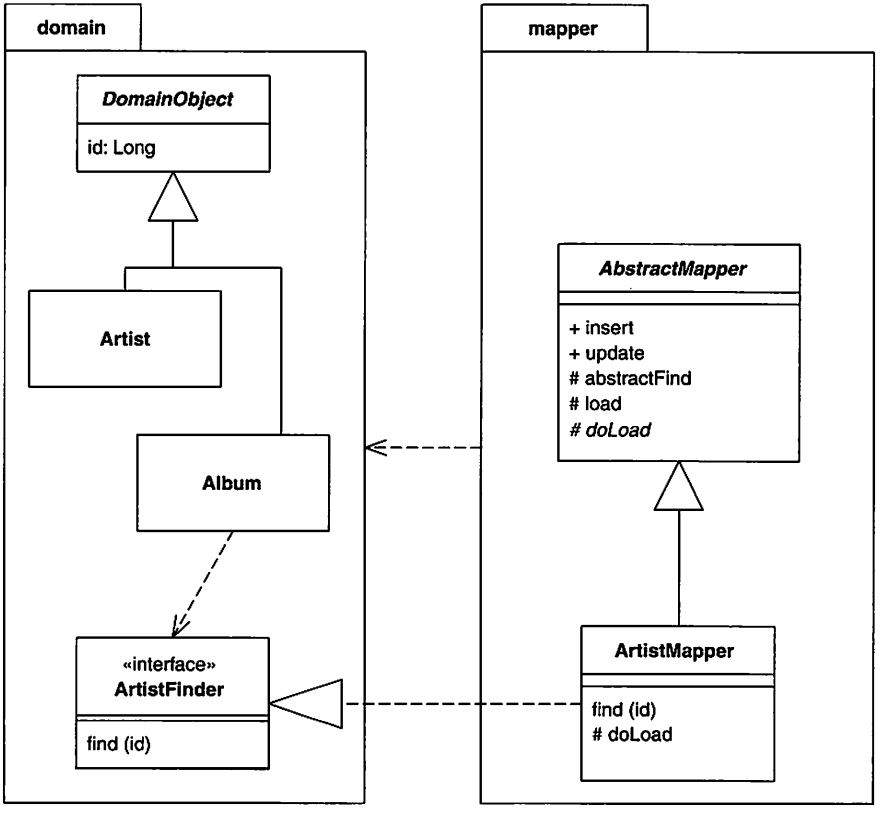


図 10.5 ——ドメインパッケージでの find インタフェースの定義

```
interface ArtistFinder...

   Artist find(Long id);
   Artist find(long id);
```

最適なのは、findインタフェースを**レジストリ**に保持されるfindメソッドとともに、ドメインパッケージ内で宣言することである。この例では、Mapperクラスによってfindインタフェースを実装している。

```
class ArtistMapper implements ArtistFinder...

   public Artist find(Long id) {
      return (Artist) abstractFind(id);
   }
   public Artist find(long id) {
      return find(new Long(id));
   }
```

findメソッドの大部分は、**一意マッピング**をチェックしてオブジェクトがすでにメモリ上にあるかどうかをチェックする、マッパーの**レイヤスーパータイプ**を使用して動作する。**レイヤスーパータイプ**が存在しない場合は、ArtistMapperによって読み込まれるあらかじめ準備されたステートメントを実行して、それを完了する。

```
class AbstractMapper...

   abstract protected String findStatement();
   protected Map loadedMap = new HashMap();
   protected DomainObject abstractFind(Long id) {
      DomainObject result = (DomainObject) loadedMap.get(id);
      if (result != null) return result;
      PreparedStatement stmt = null;
      ResultSet rs = null;
      try {
         stmt = DB.prepare(findStatement());
         stmt.setLong(1, id.longValue());
         rs = stmt.executeQuery();
         rs.next();
         result = load(rs);
         return result;
      } catch (SQLException e) {
         throw new ApplicationException(e);
      } finally {cleanUp(stmt, rs);
      }
   }

class ArtistMapper...
```

```
    protected String findStatement() {
        return "SELECT " + COLUMN_LIST + " FROM artists art WHERE ID = ?";
    }
    public static String COLUMN_LIST = "art.ID, art.name";
```

この振る舞いのfindの部分では、既存のオブジェクトまたは新しいオブジェクトを取得する。一方、loadの部分は、データベースのデータを新しいオブジェクトに配置する。

```
class AbstractMapper...

    protected DomainObject load(ResultSet rs) throws SQLException {
        Long id = new Long(rs.getLong("id"));
        if (loadedMap.containsKey(id)) return (DomainObject)
            loadedMap.get(id);
        DomainObject result = doLoad(id, rs);
        loadedMap.put(id, result);
        return result;
    }
    abstract protected DomainObject doLoad(Long id, ResultSet rs)
        throws SQLException;

class ArtistMapper...

    protected DomainObject doLoad(Long id, ResultSet rs) throws
        SQLException {
        String name = rs.getString("name");
        Artist result = new Artist(id, name);
        return result;
    }
```

このloadメソッドも**一意マッピング**をチェックしている点に注目してほしい。例では重複してしまっているが、loadメソッドを、チェックをまだ終えていない他のfindメソッドから呼び出すことができる。スキームですべてのサブクラスがするべきことは、doLoadメソッドを作成して必要な実際のデータを読み込み、findStatementメソッドからあらかじめ用意された適切なステートメントを返すことである。

また、クエリーを基盤にした検索を実行することもできる。たとえば、Track（曲目）とAlbum（アルバム）のデータベースがあり、指定されたAlbumのすべてのTrackを検索するfindメソッドが必要だとしよう。このインタフェースは再びfindメソッドを定義する。

```
interface TrackFinder...

   Track find(Long id);
   Track find(long id);
   List findForAlbum(Long albumID);
```

これはクラスに限定されたfindメソッドであるため、**レイヤスーパータイプ**ではなく、TrackMapperクラスのような特定のクラスに実装される。あらゆるfindメソッド同様に、実装には2種類のメソッドが存在している。1つはあらかじめ準備されたステートメントを設定し、もう1つはあらかじめ準備されたステートメントの呼び出しをラッピングし結果を解釈する。

```
class TrackMapper...

   public static final String findForAlbumStatement =
         "SELECT ID, seq, albumID, title " +
         "FROM tracks " +
         "WHERE albumID = ? ORDER BY seq";
   public List findForAlbum(Long albumID) {
      PreparedStatement stmt = null;
      ResultSet rs = null;
      try {
         stmt = DB.prepare(findForAlbumStatement);
         stmt.setLong(1, albumID.longValue());
         rs = stmt.executeQuery();
         List result = new ArrayList();
         while (rs.next()) result.add(load(rs));
            return result;
      } catch (SQLException e) {
         throw new ApplicationException(e);
      } finally {cleanUp(stmt, rs);
      }
   }
```

このfindメソッドは、結果群の各行ごとにloadメソッドを呼び出す。メソッドは、メモリ上のオブジェクトの作成と、データを伴ったオブジェクトの読み込みを担当している。前の例と同様、すでに何かが読み込まれているかどうかをチェックするための**一意マッピング**のチェックなど、一部分は**レイヤスーパータイプ**で処理することが可能である。

10.4.5 | 例：空のオブジェクトの作成（Java）

オブジェクトの読み込みには、2種類の手法がある。1つはコンストラクタを使って完全なオブジェクトを作成する方法であり、前述の例ではこの方法を使用している。結果は以下のような読み込みコードになる。

```
class AbstractMapper...

   protected DomainObject load(ResultSet rs) throws SQLException {
      Long id = new Long(rs.getLong(1));
      if (loadedMap.containsKey(id)) return (DomainObject)
         loadedMap.get(id);
      DomainObject result = doLoad(id, rs);
      loadedMap.put(id, result);
      return result;
   }
   abstract protected DomainObject doLoad(Long id, ResultSet rs)
      throws SQLException;

class PersonMapper...

   protected DomainObject doLoad(Long id, ResultSet rs) throws
      SQLException {
      String lastNameArg = rs.getString(2);
      String firstNameArg = rs.getString(3);
      int numDependentsArg = rs.getInt(4);
      return new Person(id, lastNameArg, firstNameArg,
         numDependentsArg);
   }
```

もう1つの手法は空のオブジェクトを作成し、setメソッドによって後に読み込む方法である。

```
class AbstractMapper...

   protected DomainObjectEL load(ResultSet rs) throws SQLException {
      Long id = new Long(rs.getLong(1));
      if (loadedMap.containsKey(id)) return (DomainObjectEL)
         loadedMap.get(id);
      DomainObjectEL result = createDomainObject();
      result.setID(id);
      loadedMap.put(id, result);
```

```
        doLoad (result, rs);
        return result;
    }
    abstract protected DomainObjectEL createDomainObject();
    abstract protected void doLoad(DomainObjectEL obj, ResultSet rs)
        throws SQLException;

class PersonMapper...

    protected DomainObjectEL createDomainObject() {
        return new Person();
    }
    protected void doLoad(DomainObjectEL obj, ResultSet rs)
        throws SQLException {
        Person person = (Person) obj;
        person.dbLoadLastName(rs.getString(2));
        person.setFirstName(rs.getString(3));
        person.setNumberOfDependents(rs.getInt(4));
    }
```

setメソッドの使用を制御したいため、異なる種類のドメインオブジェクト**レイヤスーパータイプ**を使用している点に注目してほしい。Personの姓（last name）を不変フィールドにする場合を解説する。例では、一度読み込まれた後にフィールドの値を変更したくないため、ドメインオブジェクトにステータスフィールドを追加している。

```
class DomainObjectEL...

    private int state = LOADING;
    private static final int LOADING = 0;
    private static final int ACTIVE = 1;
    public void beActive() {
        state = ACTIVE;
    }
```

その後、読み込み時にその値をチェックすることが可能となる。

```
class Person...

    public void dbLoadLastName(String lastName) {
        assertStateIsLoading();
        this.lastName = lastName;
    }
```

```
class DomainObjectEL...

   void assertStateIsLoading() {
      Assert.isTrue(state == LOADING);
   }
```

適切ではないと感じるのは、Person クラスのほとんどのクライアントが使用できないインタフェース内に、メソッドを持っているという点である。これはフィールドを設定するためのリフレクションを使用するマッパーに対する引数であり、Java の保護メカニズムは完全に使用できなくなる。

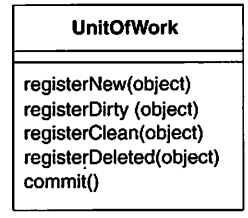
ステータスベースの保護は、行うだけの価値があるのだろうか。残念ながら、私にもわからない。一方では、複数のユーザが、間違ったタイミングで update メソッドを呼び出すというバグが検出される。しかしこのバグは、メカニズムのコストに釣り合うほど深刻なものだろうか。現時点では、私はいずれについても明確な意見は持っていない。

第11章

オブジェクトリレーショナル振る舞いパターン

11.1 | ユニットオブワーク

ビジネストランザクションの影響を受けるオブジェクトのリストを保持しつつ、変更点の書き込みと並行性の問題の解決を調整する。



データベースに対してデータの挿入と抽出を行うとき、変更点を記録しておくことは重要である。そうしないと、データはデータベースに反映されない。同時に、作成した新しいオブジェクトは挿入し、削除するオブジェクトはすべて消去しておく必要がある。

オブジェクトモデルが変化するごとにデータベースを変更することは可能だが、そうするととても小規模なデータベースの呼び出しばかりになってしまい、処理速度が低下してしまう。さらに、トランザクションをすべての相互作用に対してオープンにしておかなければならないので、ビジネストランザクションが複数のリクエストにまたがっている場合は実際的ではない。一貫性のない読み込みを回避できるように、読み込んだオブジェクトの変更点を記録しなければならない場合、さらに状況は難しくなる。

ユニットオブワークは、データベースに影響を与えるビジネストランザクション中に行われるすべての作業を記録する。その結果、作業終了時には、データベースを変更するために行うべきあらゆる作業が明らかになるのである。

11.1.1 | 動作方法

データベースへの対処が必要となる明確な原因は、新しいオブジェクトの作成や、既存オブジェクトの更新や削除といった変更である。**ユニットオブワーク**は、それらの変更の結果を記録するオブジェクトである。データベースに影響を与えることを行うときは、まず**ユニットオブワーク**を作成して変更を記録する。オブジェクトの作成、変更、削除を行うたびに**ユニットオブワーク**に通知する。また、すでに読み込んだオブジェクトについても通知するため、ビジネストランザクション中にデータベース上の一切のオブジェクトに変更がないことを検証することで、一貫性のない読み込みをチェックすることができる。

ユニットオブワークの特長は、コミットするときに何をすべきかを自ら決定する点である。**ユニットオブワーク**は、トランザクションをオープンし、あらゆる並行性チェックを行い（**重オフラインロック**または**軽オフラインロック**を使用）、データベースの変更を書き込む。アプリケーションのプログラマは、データベース更新のためのメソッドを明示的に呼び出さないので、何が変更されたかを記録する必要はなく、参照整合性が処理の順番にどう影響するかを心配する必要もない。

もちろん、こうした機能を有効にするには、記録対象となるオブジェクトがどれであるかを**ユニットオブワーク**が知っている必要がある。そのためには呼び出しを使用するか、オブジェクトから**ユニットオブワーク**に指示を出すようにする。

caller registration（呼び出し元登録）では（図11.1）、オブジェクトのユーザは変更があった場合、**ユニットオブワーク**を使用してオブジェクトを登録する必要がある。登録されていないオブジェクトは、コミット時に書き込まれないからである。

登録を忘れてトラブルを起こす可能性もあるが、書き込みたくない変更をメモリ上で行えるという柔軟性も持っている。ただ私としては、余計な混乱を招く可能性があることは指摘しておきたい。混乱を招かないためには、明示的なコピーを作成する方法をお勧めする。

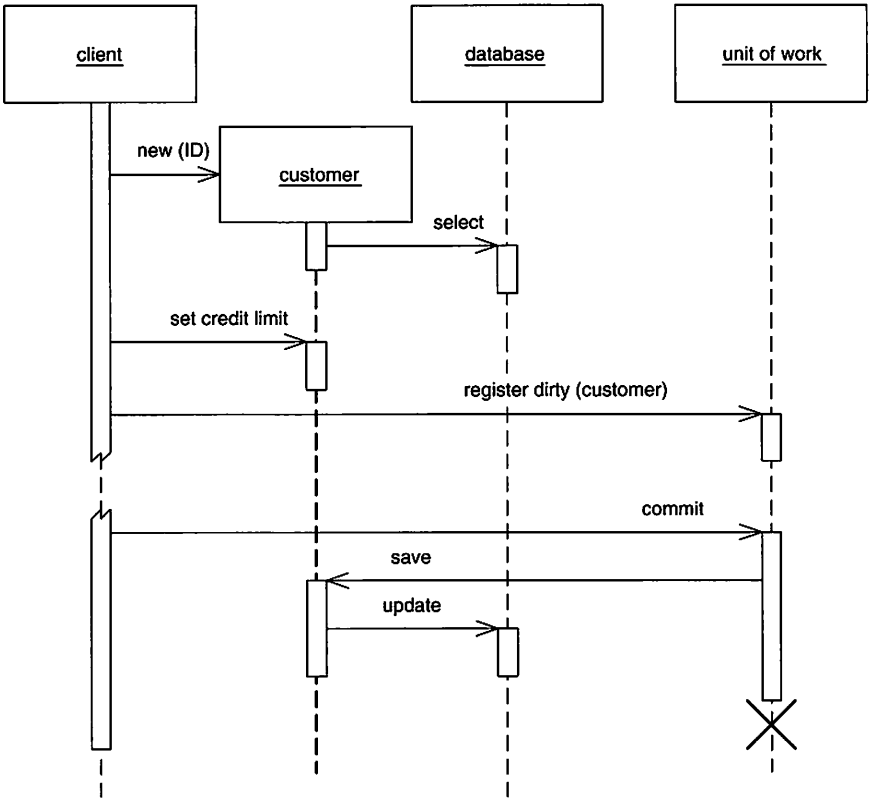


図 11.1 ——呼び出し元による変更されたオブジェクトの登録

object registration（オブジェクト登録）では（図 11.2）、呼び出し元に負荷はかからない。オブジェクトメソッドに registration（登録）メソッドを配置するからである。データベースからオブジェクトを読み込むときは、オブジェクトを確定したオブジェクトとして登録し、set メソッドはオブジェクトを不確定なものとして登録する。このスキームが機能するためには、**ユニットオブワーク**をオブジェクトに渡すか、既知の場所に配置しておく必要がある。

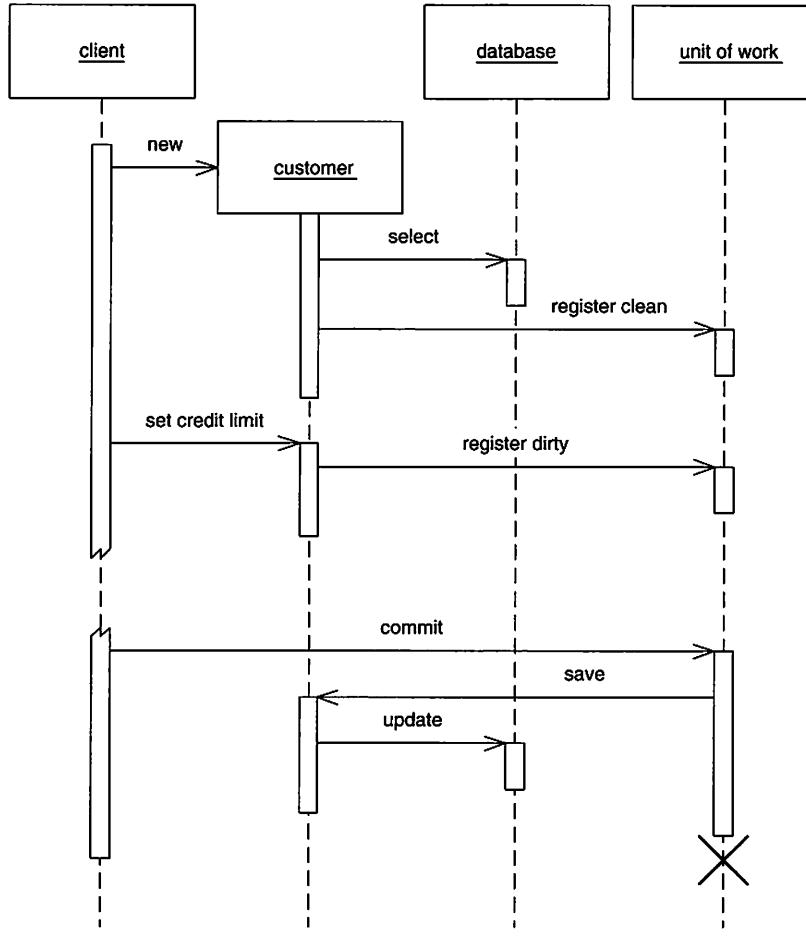


図 11.2 ——受け取り先オブジェクトのオブジェクト自体による登録

ユニットオブワークの受け渡しは面倒だが、何らかのセッションオブジェクト内に置いておくことには特に問題はない。

オブジェクト登録にしても、覚えておかなければならないことがある。オブジェクトの開発者は、登録のための呼び出しを、適切な場所に追加しておかなければならない。こうしたことは習慣化できるものだが、忘れるとかなり難しいバグとなるので注意してほしい。

適切な呼び出しを生成するのはコード生成の仕事だが、これは、生成されるコードと生成されないコードが明確に分かれている場合に実行可能である。この問題はとりわけアスペクト指向プログラミングに当てはまるものである。私はこの問題を解決するために、オブジェクトファイルの後処理（ポストプロセッシング）という方法を思いついた。この例では、ポストプロセッサはすべての Java .class ファイルをチェックし、メソッドを探して、バイト

コードに登録呼び出しを挿入している。こうした凝った方法は洗練されていないように感じられるかもしれないが、これはデータベースに関係するコードを通常のコードから分離するのである。アスペクト指向プログラミングはソースコードを使って処理をもっときちんと行う。ツールが次第に一般的となって来たため、今後はこうした手法がますます使用されるだろうと私は予想している。

私が知っている他の技法としては、TOPLink 製品の**ユニットオブワークコントローラ**（図 11.3 参照）が挙げられる。**ユニットオブワークコントローラ**では、**ユニットオブワーク**がデータベースからのあらゆる読み込み処理をし、確定したオブジェクトが読み込まれたときにそれを登録する。**ユニットオブワーク**は、オブジェクトを不確定なものとしてマーキングするのではなく、読み込み時にコピーを取得し、コミット時にオブジェクトと照合するのである。これによってコミット処理にオーバーヘッドが生じるが、実際に変更されたフィールドだけを更新でき、ドメインオブジェクト内での登録呼び出しは回避できる。こうしたハイブリッドな手法によって、変更されたオブジェクトだけコピーを取得するようにする。登録は必要だが、選択的に更新することができるため、更新以上に読み込みが多い場合には、コピーによるオーバーヘッドを大幅に軽減できる。

オブジェクト作成時には、呼び出し元登録を考慮すべきであるが、一時的なオブジェクトを作成する人にとっては、特別目新しいことではない。その最も良い例がドメインオブジェクトのテストである。この場合、データベース書き込みがない分テストの実行速度が高速化される。呼び出し元登録でこのメリットを実現できる。しかしこの他にも、**ユニットオブワーク**による登録を行わない一時コンストラクタを提供したり、コミットには何も関知しない**スペシャルケースのユニットオブワーク**を提供したりするといった解決方法もある。

また、**ユニットオブワーク**の有効なもう 1 つの領域は、データベースが参照整合性を行う際の更新の順番である。SQL 呼び出しごとではなく、トランザクションがコミットする時点でデータベースが、確実に参照整合性をチェックし、更新の順番を気にしなくてもよくなる。ほとんどのデータベースでこの処理を行うことができるので、できる限りこの方法を使用することを勧める。使用できない場合には、**ユニットオブワーク**こそが、こうした更新の順序を整理するには最適である。小規模なシステムでは、依存性外部キーに基づいたテーブルへの詳細な書き込み順序を含んだ明示的なコードによってこの処理を実行できる。一方、大規模アプリケーションでは、メタデータを使用して、データベースへの書き込み順序を決めることを勧める。この手順は本書が扱う範囲ではないので、商用ツールを参考にすること。自分自身でメタデータを使用しなければならない場合、そのパズルの鍵はトポロジカルソートであると私は認識している。

同様のテクニックを使って、デッドロックの最小化もできる。すべてのトランザクションが同じ順番でテーブルを編集して、デッドロックのリスクはかなり軽減される。**ユニットオブワーク**は、テーブル書き込みの決まった順序を保持するには最適な場所であり、常に同じ

順番でテーブルにアクセスできる。

オブジェクトは現行の**ユニットオブワーク**を見つけられなくてはならない。もっとも有効な手段は、スレッドスコープされている**レジストリ**を使う方法である。もう1つの方法は、メソッド呼び出しあるいはオブジェクトの作成時に、**ユニットオブワーク**を必要とするオブジェクトに渡す方法である。いずれの場合も、複数のスレッドが**ユニットオブワーク**に同時にアクセスできないようにしておく（この点がかなり面倒である）。

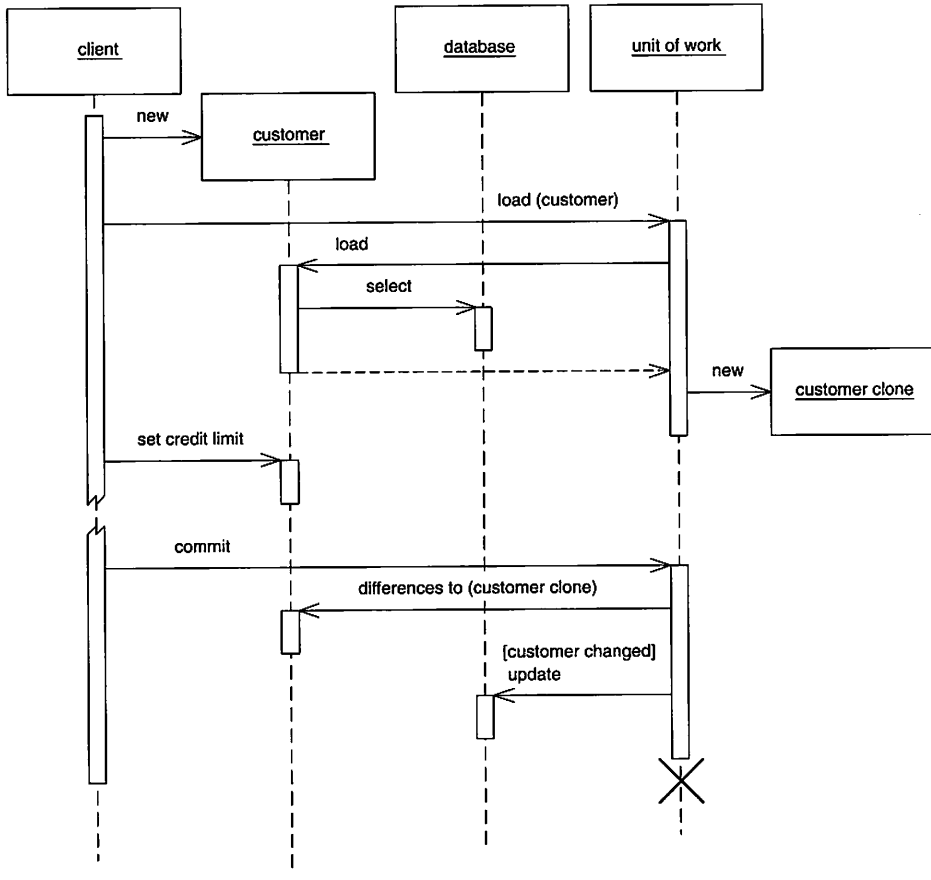


図11.3——データベースアクセスのためのコントローラとしてのユニットオブワークの使用

ユニットオブワークでは、バッチ更新の処理が重視されている。**batch update**（バッチ更新）の基本的な考え方は、1つのリモート呼び出しで処理できるように複数のSQLコマンドを1つのユニットとして送信することである。更新、挿入および削除が立て続けに送信される場合、とくに重要になる。違う環境ではバッチ更新に対して、異なるレベルのサポートが提供される。JDBCには、個々のステートメントをバッチ化する機能が用意されている。

また、こうした機能を持たない場合、複数のSQL文から構成される文字列を作成し、1つのステートメントとして送ることで、同じ動作を行うことができるようになる。[Nilsson]では、Microsoftプラットフォーム用のサンプルを紹介している。しかし、実際に行う場合、ステートメントのプリコンパイルに支障をきたすことがないかどうかを確認しておこう。

ユニットオブワークは、データベースだけでなく、あらゆるトランザクションリソースに使用できるため、メッセージキューやトランザクション監視との関係を変えることもできる。

.NETの実装

.NETでは、**ユニットオブワーク**が非接続型のデータセットによって処理されるため、従来のパターンとは少し異なるパターンとなる。私がこれまでに出会ったほとんどの**ユニットオブワーク**は、オブジェクトへの変更を登録し記録している。.NETはデータをデータベースからデータセットに読み込む。データセットとは、データベースのテーブル、行、列のように配置された一連のオブジェクトであり基本的に、1つまたは複数のSQLクエリーのメモリ上のミラーイメージである。各データ行は、バージョン(カレント、オリジナル、提示された)と状態(未変更、追加、削除、修正)という概念を持っていて、データセットによるデータベース構造の模倣と、この概念によってデータベースへの簡単な変更の書き込みが可能になる。

11.1.2 | 使用するタイミング

ユニットオブワークの基本動作は、処理されたさまざまなオブジェクトを記録することで、メモリ上のデータとデータベースを同期するのに、どのオブジェクトが必要であるかを知ることができる。すべての作業をシステムトランザクション内で行うことができるとしたら、注意すべき唯一のオブジェクトは変更を加えたオブジェクトということになる。**ユニットオブワーク**はそのための最も有効な方法の1つだが、他の選択肢もある。

最もシンプルな選択肢は、変更を行った時点でオブジェクトを明示的に保存する方法である。この場合の問題点は、作業中に3箇所で1つのオブジェクトに変更を加えている場合、それぞれに3回の呼び出しを行わなければならないので、最後に1回の呼び出しを行う場合よりもデータベースの呼び出し回数が予想以上に増えてしまうことである。複数のデータベース呼び出しを避ける方法は、すべての更新を最後に行うことだ。そのためには、変更が加えられたすべてのオブジェクトを記録しておく必要がある。記録にはコード中の変数を使うこともできるが、変数の数が一定以上に増えると、管理不能となってしまう。**トランザクションスクリプト**では変数は有効に機能する場合が多いが、**ドメインモデル**ではとても難しい。

オブジェクトを変数として保持する代わりに、オブジェクトごとに不確定フラグを使用し、

オブジェクトに変更があった時点で、フラグを設定するという方法も考えられる。その場合、トランザクションの最後にすべての不確定なオブジェクトを見つけて書き出す必要がある。この技法の使い勝手は、不確定なオブジェクトを簡単に見出せる構造になっているかどうかに拠る。すべてのオブジェクトが1つの階層構造に収まっていれば、その階層を縦断して、変更されたすべてのオブジェクトを書き出すことができるが、**ドメインモデル**のような一般的なオブジェクトネットワークにおいては、縦断は容易ではない。

ユニットオブワークの最大の強みは、すべての情報を1箇所に保存する点である。使い始めると、後は変更の順序についてほとんど記憶する必要はない。また**ユニットオブワーク**は、**軽オフラインロック**や**重オフラインロック**を使用している、複数のシステムトランザクションにわたるビジネストランザクションの処理といった、より複雑な状況に適した確実なプラットフォームでもある。

11.1.3 | 例：オブジェクト登録を備えたユニットオブワーク（Java）

(by David Rice)

特定のビジネストランザクションに対するすべての変更を記録し、その後指示があった時点でデータベースにそれらをコミットする**ユニットオブワーク**を解説する。このドメインレイヤは**レイヤスーパータイプ**、ドメインオブジェクトを持ち、**ユニットオブワーク**と相互作用する。変更セットの保存には、3つのリスト（新規、不確定、削除ドメインオブジェクト）を使用する。

```
class UnitOfWork...

   private List newObjects = new ArrayList();
   private List dirtyObjects = new ArrayList();
   private List removedObjects = new ArrayList();
```

登録メソッドは、リストの状態を保持し、IDがnull（設定なし）でないことや、不確定なオブジェクトが新規として登録されていないことをチェックするような基本的なアサーションを実行する。

```
class UnitOfWork...

   public void registerNew(DomainObject obj) {
      Assert.notNull("id not null", obj.getId());
      Assert.isTrue("object not dirty", !dirtyObjects.contains(obj));
      Assert.isTrue("object not removed", !removedObjects.contains(obj));
```

```
        Assert.isTrue("object not already registered new",
            !newObjects.contains(obj));
        newObjects.add(obj);
    }
    public void registerDirty(DomainObject obj) {
        Assert.notNull("id not null", obj.getId());
        Assert.isTrue("object not removed",
            !removedObjects.contains(obj));
        if (!dirtyObjects.contains(obj) &&
            !newObjects.contains(obj)) { dirtyObjects.add(obj);
        }
    }
    public void registerRemoved(DomainObject obj) {
        Assert.notNull("id not null", obj.getId());
        if (newObjects.remove(obj)) return;
        dirtyObjects.remove(obj);
        if (!removedObjects.contains(obj)) {
            removedObjects.add(obj);
        }
    }
    public void registerClean(DomainObject obj) {
        Assert.notNull("id not null", obj.getId());
    }
```

registerClean()は何も処理しない点に注目してほしい。通例では、**ユニットオブワーク**内に**一意マッピング**を配置する。同じオブジェクトの複数コピーが未定義の振る舞いとなることもあるため、**一意マッピング**はドメインオブジェクトの状態をメモリ上に格納する場合には必要不可欠である。**一意マッピング**が配置される場合、registerClean()は登録されたオブジェクトをそこに格納する。同様にregisterNew()は新規オブジェクトをマッピングに置き、registerRemoved()は削除されたオブジェクトをマッピングから取り除く。**一意マッピング**がない場合は、**ユニットオブワーク**にregisterClean()を含まないという選択肢もあるが、私は、不確定リストから変更オブジェクトを削除するといったメソッドの実装を目にしたことがある。しかし変更の部分的なロールバックは常に扱い難いものである。変更セットの状態を修正する場合には、とくに注意が必要である。

commit()は各オブジェクトの**データマッパー**を検索し、適切なマッピングメソッドを実行する。updateDirty()とdeleteRemoved()は示されていないがinsertNew()のように振る舞う。

```
class UnitOfWork...

   public void commit() {
      insertNew();
      updateDirty();
      deleteRemoved();
   }
   private void insertNew() {
      for (Iterator objects = newObjects.iterator(); objects.hasNext();) {
         DomainObject obj = (DomainObject) objects.next();
         MapperRegistry.getMapper(obj.getClass()).insert(obj);
      }
   }
```

すでに読み込まれ、コミット時に一貫性のない読み込みエラーをチェックしたいオブジェクトの記録は、**ユニットオブワーク**には含まれていない。こうした処理は、**軽オフラインロック**で実行されることになる。

次に、オブジェクト登録を始める。まずは、各ドメインオブジェクトが、現在のビジネストランザクションを担当する**ユニットオブワーク**を見つけることである。すべてのドメインモデルが**ユニットオブワーク**を必要とするため、パラメータとしてそれを渡す方法は合理的ではない場合が多い。各ビジネストランザクションが1つのスレッド内で実行される場合、java.lang.ThreadLocal クラスを使って**ユニットオブワーク**を現在実行中のスレッドに関連付ける。シンプルにするため、**ユニットオブワーク**クラスで静的メソッドを使用して、この機能を追加する。ビジネストランザクション実行スレッドに関連付けられた何らかのセッションオブジェクトがすでにある場合、管理オーバーヘッドが生じる別のスレッドマッピングを追加するよりは、セッションオブジェクトに現行の**ユニットオブワーク**を配置する方が得策である。また、**ユニットオブワーク**は論理上セッションに属している。

```
class UnitOfWork...

   private static ThreadLocal current = new ThreadLocal();
   public static void newCurrent() {
      setCurrent(new UnitOfWork());
   }
   public static void setCurrent(UnitOfWork uow) {
      current.set(uow);
   }
   public static UnitOfWork getCurrent() {
      return (UnitOfWork) current.get();
   }
```

これで現行の**ユニットオブワーク**を使用して、抽象ドメインオブジェクト自体を登録するためのマーキングメソッドを提供できる。

```
class DomainObject...

   protected void markNew() {
      UnitOfWork.getCurrent().registerNew(this);
   }
   protected void markClean() {
      UnitOfWork.getCurrent().registerClean(this);
   }
   protected void markDirty() {
      UnitOfWork.getCurrent().registerDirty(this);
   }
   protected void markRemoved() {
      UnitOfWork.getCurrent().registerRemoved(this);
   }
```

具象ドメインオブジェクトは、必要に応じて、忘れずに新規および不確定とマーキングする必要がある。

```
class Album...

   public static Album create(String name) {
      Album obj = new Album(IdGenerator.nextId(), name);
      obj.markNew();
      return obj;
   }
   public void setTitle(String title) {
      this.title = title;
      markDirty();
   }
```

削除オブジェクトの登録が、抽象ドメインオブジェクト上の `remove()` メソッドによって処理されるとは限らない。また、`registerClean()` を実装している場合、**データマッパー**は新たに読み込まれたオブジェクトを確定したものとして登録する必要がある。

最後に、必要に応じて**ユニットオブワーク**を登録しコミットする。この処理は、明示的にも暗黙的にも実行できる。明示的な**ユニットオブワーク**管理の例を以下に示す。

```
class EditAlbumScript...

   public static void updateTitle(Long albumId, String title) {
      UnitOfWork.newCurrent();
      Mapper mapper = MapperRegistry.getMapper(Album.class);
      Album album = (Album) mapper.find(albumId);
      album.setTitle(title);
      UnitOfWork.getCurrent().commit();
   }
```

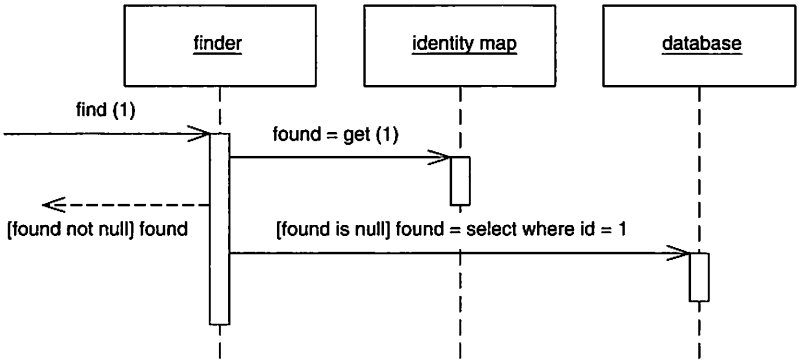
単調で繰り返しの多いコーディングを避けたい場合には、どんなにシンプルなアプリケーションよりも、暗黙的な**ユニットオブワーク**管理がよい。具象サブタイプに対して**ユニットオブワーク**を登録し、コミットするサーブレット**レイヤスーパータイプ**を以下に示す。サブタイプは doGet() をオーバーライドするのではなく、handleGet() を実装する。handleGet() の内部で実行されるすべてのコードには、ともに動作する**ユニットオブワーク**を持つことになる。

```
class UnitOfWorkServlet...

   final protected void doGet(HttpServletRequest request,
         HttpServletResponse response) throws ServletException,
         IOException {
      try {
         UnitOfWork.newCurrent();
         handleGet(request, response);
         UnitOfWork.getCurrent().commit();
      } finally {
         UnitOfWork.setCurrent(null);
      }
   }
   abstract void handleGet(HttpServletRequest request,
         HttpServletResponse response) throws ServletException,
         IOException;
```

上記のサーブレットサンプルは、システムトランザクション制御をスキップしている点で、やや単純化されている。**フロントコントローラ**を使用する場合、doGet() ではなく、コマンドに**ユニットオブワーク**をラップするため、あらゆる実行コンテキストで、同様のラップが実行できる。

11.2 | 一意マッピング

読み込まれたすべてのオブジェクトを1つのマッピングに保存することで、各オブジェクトが確実に一度だけ読み込まれるようにする。オブジェクトを参照する場合には、マッピングを使って参照する。



古い格言に、『時計を2つ持つものは決して時間を知るすべを知らない』というのがある。2つの時計が合っていないと、データベースからのオブジェクトの読み込みにおいては、さらに大きな混乱を招くだろう。注意を怠れば、同じデータベースレコードから2つの異なるオブジェクトにデータを読み込んでしまうこともあるからだ。その後、双方を更新すると、データベースにそれぞれの変更が書き込まれるという奇妙な結果が生じてしまう。

こうした状況はパフォーマンスにも影響する。同じデータを2回以上読み込むことで、リモート呼び出しに負荷がかかる。つまり、同じデータを2回読み込まないことで、正確さを確保でき、アプリケーションの処理速度も向上するのである。

一意マッピングは、1つのビジネストランザクションにおいて、データベースから読み込まれたすべてのオブジェクトを記録する。オブジェクトが必要になったときには、まずは**一意マッピング**を確認してオブジェクトがすでに読み込み済みであるかどうかを判断できるのである。

11.2.1 | 動作方法

一意マッピングの基本的な考え方は、データベースから取得されたオブジェクトを含む一連のマッピングを用意することである。シンプルなケースでは、同じ形式のスキーマによって、データベーステーブルごとに1つのマッピングを持つようにする。データベースからオブジェクトを読み込む場合、まずはマッピングをチェックする。その中に読み込もうとしているオブジェクトがある場合、**一意マッピング**はそれを返す。ない場合はデータベースに移

動し、オブジェクトを読み込むと同時に、今後のためにマッピングにも格納する。

注意しなければいけないのは、実装の選択肢である。また、**一意マッピング**は並行性管理によって相互作用するため、**軽オフラインロック**についても考慮しなくていけない。

11.2.1.1 ■ キーの選択

最初に考慮すべきは、マッピングのキーである。まず考えられる選択肢は、対応するデータベーステーブルのプライマリキーである。キーが1つの列で不変である場合は、うまくいく。代理プライマリキーは、マッピングのキーとして使用することもできるため、この手法にとても適している。キーはシンプルなデータタイプとなるため、比較する場合も有効に機能する。

11.2.1.2 ■ 明示的あるいは汎用

一意マッピングは、明示的あるいは汎用のいずれかを選択しなければならない。明示的な**一意マッピング**は、必要とするオブジェクト固有のメソッド（例：`findPerson(1)`）を使用してアクセスされる。一方、汎用マッピングは、あらゆるオブジェクトに対して1つのメソッドを使用し、必要とするオブジェクトの種類をパラメータで指定する（例：`find("Person", 1)`）。明確なメリットは、汎用で再使用可能なオブジェクトを使用して汎用マッピングをサポートできる点である。あらゆる種類のオブジェクトに機能し、新しいマッピングの追加時にも更新する必要がない再使用可能な**レジストリ**をたやすく構築できる。

しかし、私の好みは明示的な**一意マッピング**である。まず第一に、厳密に型付けされた言語によるコンパイル時チェック機能を持っている。さらに、明示的なインタフェースのあらゆるメリットを持っている。どのマッピングが使用可能であり、何と呼ばれているかを簡単に知ることができる。この場合、新しいマッピングを追加するごとにメソッドが追加されるが、明示的であることのメリットに比べればたいした問題ではない。

選択を左右するのはキーのタイプである。すべてのオブジェクトが同じタイプのキーを持つ場合は、汎用マッピングを使用する。キーのタイプこそが、1つのキーオブジェクトの裏側にさまざまな種類のデータベースキーをカプセル化するための最適な引数になる（詳しくは**一意フィールド**参照）。

11.2.1.3 ■ マッピングの数

クラスごとに1つのマッピングを使用するか、セッション全体に1つのマッピングを使用するかを決定する必要がある。データベース固有のキーを持っている場合は、セッションを対象とした1つのマッピングが機能する（この点のトレードオフについては**一意フィールド**の解説を参照）。1つの**一意マッピング**を持つ場合のメリットとしては、参照先を1つに限定し、継承についての面倒な決定が不要となる点が挙げられる。

複数のマッピングを持つ場合は、クラスごとに1つのマッピングか、テーブルごとに1つのマッピングを使用することになるが、データベーススキーマとオブジェクトモデルが同じ場合には有効である。両方が異なるようであれば、オブジェクトはマッピングの複雑さを知る必要がないため、テーブルよりもオブジェクトをマッピングのベースにする方が簡単である。

継承という難しい問題がある。Vehicle（乗り物）のサブタイプとしてのCar（自動車）を使用する場合、1つのマッピングを持つだろうか、それとも別々のマッピングを持つだろうか。マッピングを分離、参照するときは、すべてのマッピングを照合しなければならないので、ポリモーフィズムによる参照はより複雑になってしまう。こうした理由から、私は継承ツリーごとに1つのマッピングを使用する方法を好む。その場合は、継承ツリーにまたがった一意のキーを使用するが、**具象テーブル継承**を使用する場合、より面倒になると思う。

1つのマッピングのメリットは、データベースを追加しても、新しいマッピングを追加する必要がない点である。ただし、**データマッパー**にマッピングを結び付けることは、それほど大きな作業負荷ではないが。

11.2.1.4 ■ どこに配置するか

一意マッピングはできる限り見つけやすい場所に置く必要がある。また、それらはプロセスコンテキストに結び付けることがある。各セッションは他のあらゆるセッションのインスタンスとは分離した独自のインスタンスを持つようにしておく。つまり、**一意マッピング**はセッション固有のオブジェクトに配置する必要があるということである。**ユニットオブワーク**を使用する場合、**ユニットオブワーク**がデータベースに挿入またはデータベースから抽出するデータを記録するためにも最も適した場所であるため、**一意マッピング**にとっても明らかに最良の配置場所である。一方、**ユニットオブワーク**を使用しない場合は、セッションと結び付いた**レジストリ**が最良の場所となる。

ここで記述したように、セッションに対して1つの**一意マッピング**を使用することが多い。そうでない場合は、マッピングに対するトランザクション上の保護を提供する必要があるが、どんな開発者にもより多くの作業が要求される。しかし、2、3の例外もある。その1つは、たとえレコードデータにリレーショナルデータベースを使用しても、オブジェクトデータベースをトランザクション上のキャッシュとして使用している場合である。私はこの方法だけを使用した場合のパフォーマンスの調査を目にしたことはないが、注目に値するものはいくつもある。事実、私が尊敬する多くの人が、パフォーマンスの改善の手段として、このトランザクション上のキャッシュに注目している。

もう1つの例外は、どんな場合でも読み込みだけしかできない読み取り専用オブジェクトである。オブジェクトが絶対に修正できないとしたら、セッションにまたがって共有されたとしても、心配する必要はまったくない。パフォーマンス重視のシステムにおいては、すべての読み取り専用データを一度に読み込み、プロセス中ずっと使用可能な状態にしておくと

よいのである。その場合、読み取り専用の**一意マッピング**をプロセスコンテキストに保持し、更新可能な**一意マッピング**はセッションコンテキストに保持しておく。この方法は、完全な読み取り専用ではないが、更新される可能性のほとんどない（万が一、そうした更新が発生したときには、プロセスワイドの**一意マッピング**を消去して、サーバを切り離しても惜しくないような）オブジェクトにも適用できる。

ただ1つの**一意マッピング**を使う状況であっても、読み取り専用ラインと更新可能ラインの2つに分割しておくことである。両方のマッピングをチェックするインタフェースを提供することで、クライアントは両方の違いを意識しなくてもすむようになるからである。

11.2.2 | 使用するタイミング

一般的に、**一意マッピング**はデータベースから抽出され、修正されたオブジェクトの管理に使用される。主な理由は、2つのメモリ上のオブジェクトが1つのデータベースレコードに対応しているという状態を回避するためである。2つのレコードを並行して修正した結果、データマッピングが混乱することもある。

一意マッピングのもう1つのメリットとしては、データベース読み込みのキャッシュとして機能する点が挙げられる。何らかのデータが必要になるたびに、その都度データベースにアクセスする必要がなくなるのである。

不変オブジェクトに対しては、**一意マッピング**は必要ないかもしれない。オブジェクトが変更できない場合、修正の間違いについては心配ない。**バリューオブジェクト**は不変であるため、それらに対しては**一意マッピング**は不要である。また**一意マッピング**にはいくつかのメリットがあり、中でも一番重要なものは、キャッシュによるパフォーマンス上のメリットである。その他にも、Java において問題となっている、== をオーバーライドできないために発生する、間違った形式の等価テストの使用を回避することにも役立つ点を挙げることができる。

依存マッピングには**一意マッピング**は必要ない。依存オブジェクトの持続性は親によって制御されるため、一意性を維持するためのマッピングは必要ないのである。しかし、マッピングは必要なくても、データベースキーを介してオブジェクトにアクセスする必要がある場合には、一意マッピングの提供が求められる。その場合マッピングは単なるインデックスであるが、それをマッピングと見なすかどうかについては議論の余地がある。

一意マッピングは、1つのセッション内での更新競合の回避について役立つが、セッションにまたがる競合の処理においてはまったく効果がない。これは込み入った問題であり、詳しくは**軽オフラインロック**および**重オフラインロック**の項で解説する。

11.2.3 ｜例：一意マッピングのためのメソッド（Java）

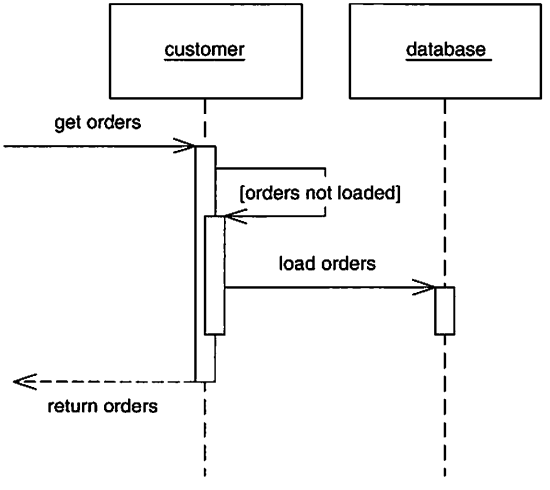
それぞれの一意マッピングごとに、マッピングフィールドとアクセッサーを使用する。

```
private Map people = new HashMap();
public static void addPerson(Person arg) {
   soleInstance.people.put(arg.getID(), arg);
}
public static Person getPerson(Long key) {
   return (Person) soleInstance.people.get(key);
}
public static Person getPerson(long key) {
   return getPerson(new Long(key));
}
```

Java の弱点の１つとして、long がオブジェクトではないため、マッピングのインデックスとして使用できないという点がある。ただ、インデックスでは計算を一切行わないため、実際にはそれほど困ることではない。唯一問題となるのは、リテラルとともにオブジェクトを抽出したい場合である。プロダクションコードにおいてはほとんど必要なくても、テストコードでは頻繁に必要とされることがある。そのため私自身もテストを容易にするために、long を使用した get メソッドを含めたことがある。

11.3 ｜レイジーロード

必要なすべてのデータは含まないが、その取得方法を知っているオブジェクト。



データベースからメモリにデータを読み込む場合、対象となるオブジェクトを読み込むと同時に、関連したオブジェクトも読み込むように設計しておくことは有用である。すべてのオブジェクトを明示的に読み込む必要がなくなる分、オブジェクトを使用する開発者にとって読み込みが容易になるからである。

しかし、こうした考え方を論理的に突き詰めると、1つのオブジェクトの読み込みが無数の関連オブジェクトの読み込みに影響する可能性もあり、必要なオブジェクトがほんの2つか3つという場合には、パフォーマンス上に問題が生じてしまう。

レイジーロードは、こうした読み込みプロセスをとりあえず中断し、オブジェクト構造におけるマークをそのままにすることで、データが必要な場合には、使用するときだけ読み込むことができるようになる。多くの人が知っているように、ある作業を怠けていたとしても、結局その作業をする必要がなくなると、得をすることになるのである。

11.3.1 | 動作方法

レイジーロードを実装するには以下の4つの方法がある。それはレイジーイニシャライズ、仮想プロキシー、バリューホルダーおよびゴーストである。

レイジーイニシャライズ [Beck Patterns]は、最もシンプルな手法である。基本的な考え方は、フィールドへのアクセスごとにチェックを行い、null（設定なし）かどうかを確認するというものである。その場合は、フィールドを返す前にフィールドの値を計算する。可能にするには、フィールド自身を確実にカプセル化しておく、つまり、クラス内のものも含めて、フィールドへのすべてのアクセスがgetメソッドを介して行われるようにする必要がある。

null値がフィールドの正規の値でない限り、null値を使ってまだ読み込まれていないフィールドを示す方法が有効である。nullが有効な価である場合は、フィールドがまだ読み込まれていないことを示すために何か他の処理を行うか、あるいは、null値に対しては**スペシャルケース**を使用する必要がある。

レイジーイニシャライズの使用はシンプルだが、オブジェクトとデータベース間の依存が強くなる傾向がある。こうした理由から、レイジーイニシャライズは**アクティブレコード**、**テーブルデータゲートウェイ**、**行データゲートウェイ**にとても適している。**データマッパー**を使用する場合、インダイレクションの追加レイヤが必要となるが、それを入手するためには**仮想プロキシー**[Gang of Four]を使用する。仮想プロキシーは、フィールドに必須なオブジェクトのように見えるが何も含んでいない。メソッドの1つが呼び出されたときだけデータベースから正しいオブジェクトを読み込む。

仮想プロキシーのメリットとは、まるでそこにあるべきオブジェクトのように見える点である。一方、欠点は、オブジェクトではないためにとても難しい一意性の問題を引き起こし

やすい点である。さらに、同様なオブジェクトに対しても、複数の仮想プロキシーを使用することができる点である。すべてのプロキシーは異なるオブジェクトIDを持つが、同じ概念上のオブジェクトを表しているのである。少なくとも、equality（等価）メソッドをオーバーライドする必要があり、identity（識別）メソッドの代わりに、忘れずに使用する必要がある。これを怠ると、とても追跡しにくいバグが生じることになる。

ある環境においては、大量の仮想プロキシー（プロキシー化するクラスごとに1つずつ）を作成しなければならない状況に追い込まれることもある。通常、動的に型付けされた言語においてはこうした状況を回避できるが、静的に型付けされた言語においては、状況が最悪の状態となることがある。プラットフォームに、Javaのプロキシーのような便利な機能が用意されている場合でも、不具合が生じる可能性がある。

リストのようなコレクションクラスに対してだけ仮想プロキシーを使用する場合には、こうした問題もとくに表面化することはない。コレクションクラスは**バリューオブジェクト**であるため、一意性は問題にならない。また、バーチャルコレクションを記述する必要のあるCollection（コレクション）クラスはそれほど多くない。

ドメインクラスの場合、**バリューホルダー**を使用することで、こうした問題を回避できる。この概念（私は最初、Smalltalkで出会った）は、他のいくつかのオブジェクトをラップするオブジェクトであると言えよう。基盤となるオブジェクトを取得するには、バリューホルダーに対して値を要求するのだが、データベースからデータを取得するのは、最初のアクセスにおいてだけである。バリューホルダーの欠点は、クラスがその存在を知っていなくてはいけないことと、強力な型付けの明示性が失われる点である。一意性の問題を回避するには、その所有クラスを超えてバリューホルダーの受け渡しをしないようにすることである。

ゴーストは不完全な状態のオブジェクトである。データベースからオブジェクトを読み込む場合、オブジェクトには、それ自身のIDだけが含まれている。フィールドにアクセスしようとするときに、完全な状態を読み込むのである。ゴーストとは、すべてのフィールドがまとめてレイジーイニシャライズされるオブジェクトあるいは、オブジェクト自身が独自の仮想プロキシーとなる仮想プロキシーと考えることができる。もちろん、すべてのデータを一挙に読み込む必要はなく、一緒に使用されるグループにまとめられる。ゴーストを使用する場合、即座に**一意マッピング**に挿入することはできるので、一意性を確保し、データ読み込み時の循環参照によって生じるあらゆる問題を回避するのである。

仮想プロキシー／ゴーストは、完全にデータが欠けているというわけではない。すぐに入手できて頻繁に使用されるデータがある場合は、プロキシーまたはゴーストの読み込み時に読み込むことは有効である（これは「軽オブジェクト」と呼ばれることもある）。

継承は、**レイジーロード**に問題を引き起こす場合が多い。ゴーストを使用する場合、どんなタイプのゴーストを作成するかを知っておく必要があるが、正しく読み込みを行わないと、わからないことがあるのだ。仮想プロキシーは、静的に型付けされた言語の場合にも同様の

問題がおこる。

もう1つ**レイジーロード**で気をつけなくてはいけないことは、必要以上にデータベースアクセスを発生させてしまう点である。このような「リプルローディング」の例として、複数の**レイジーロード**を1つのコレクションにまとめた後、それらを1つの**レイジーロード**と見なす場合が挙げられる。この場合、すべてをまとめて読み込む代わり、オブジェクトごとに毎回データベースにアクセスすることになる。私は、リプルローディングがアプリケーションのパフォーマンスを損なう例を目にしたことがある。これを回避する1つの方法としては、**レイジーロード**のコレクションを用意しないというのではなく、コレクション自身を**レイジーロード**にする代わりに、読み込む時点で、すべてのコンテンツを読み込む方法が考えられる。この方法の限界は、コレクションがとても大きい場合（たとえば、世界中のIPアドレスなど）である。これらはオブジェクトモデルの関連性を介してリンクされることはないため、それほど頻繁にはありえないが、そうした状況になった場合には、バリューリストハンドラ [Alur et al] が必要となるだろう。

アスペクト指向プログラミングにおいて、**レイジーロード**は有効な選択肢である。独立したアスペクトに**レイジーロード**の振る舞いを追加することによって、レイジーロードストラテジーを個別に変更するだけでなく、レイジーロードの問題解決からドメイン開発者を開放できる。私は、**レイジーロード**を透過的な方法で実装する、プロジェクト後処理Javaバイトコードを目にしたこともある。

異なるユースケースが、異なるレイジー性を使用して適切に動作している状況を目にすることも多い。オブジェクトグラフのサブセットを必要とするものもあれば、別のサブセットを必要とするものもある。最大の効率性を得るには、正しいユースケースに正しいサブグラフを読み込むことである。

そのための方法としては、異なるユースケースごとに、個別のデータベース相互作用オブジェクトを使用するという方法が挙げられる。**データマッパー**を使用する場合、即座に明細を読み込むマッパーと、後に読み込みを行うマッパーの2種類のOrderマッパーオブジェクトを使用する方法である。アプリケーションコードは、ユースケースに応じて適切なマッパーを選択する。バリエーションとしては、同じ基本ローダーオブジェクトを使用するが、読み込みパターンの決定は、どのように行うかと言うストラテジーオブジェクトに従う方法もある。こちらの方がより洗練されているが、振る舞いの抽出では優れた方法であるとは限らない。

理論的にはさまざまなレイジー性が求められるが、必要なのは完全な読み込みか、リストの識別のために十分な読み込みの2種類である。それ以上追加すると複雑性が増すだけで、効果は相殺されてしまう。

11.3.2 | 使用するタイミング

レイジーロードをいつ使用するかは、オブジェクトの読み込み時にデータベースからどれくらいの量を引き出すか、また、それを必要とするデータベース呼び出しが既にあるかによって決まってくる。残りのオブジェクトと同じ行にあるフィールドに対する**レイジーロード**はほとんど意味をもたない。なぜなら、データフィールドがとても大きい場合でも、1回の呼び出しで追加データを元に戻すのに、ほとんど負荷はかからないからである（例：**シリアライズLOB**）。つまり、**レイジーロード**を考慮するのは、フィールドがアクセスの追加データベース呼び出しを必要とする場合である。

パフォーマンスという点では、どの時点でデータを元に戻したいかが判断の決め手となる。一度の呼び出しで必要な要素をすべて取り込み、所定の場所に配置することをユーザーインタフェースへの1回の操作で行えるとしたら、とても有効なことである。追加呼び出しを含んで呼び出そうとしているデータが、メインオブジェクトの使用中には使用されていないときこそ、**レイジーロード**を使用するのである。

レイジーロードの追加は、プログラムが若干複雑になるため、私としては本当に必要なとき以外はあまり使いたくない。

11.3.3 | 例：レイジーイニシャライズ（Java）

以下のコードは、レイジーイニシャライズの基本を示している。

```
class Supplier...

   public List getProducts() {
      if (products == null) products = Product.findForSupplier(getID());
      return products;
   }
```

このように、Productフィールドの最初のアクセスで、データはデータベースから読み込まれる。

11.3.4 | 例：仮想プロキシー（Java）

仮想プロキシーのキーポイントは、使用するクラスのように見えるが、シンプルなラッパーを持つクラスを提供するのである。つまり、サプライヤーのためのProductリストは、リストフィールドによって保持されることになる。

```
class SupplierVL...

   private List products;
```

このようなリストプロキシーの作成に関して最も複雑なことは、アクセス時にだけ作成される基礎的なリストを提供できる設定を行うことである。それには、インスタンスの作成時に、リストを作成するために必要なコードを仮想リストへと渡さなければならない。Java でそれを実現する最良の方法は、読み込みの振る舞いにインタフェースを定義することである。

```
public interface VirtualListLoader {
   List load();
}
```

これによって、適切なマッパーメソッドを呼び出すローダーによって仮想リストをインスタンス化できるようになる。

```
class SupplierMapper...

   public static class ProductLoader implements VirtualListLoader {
      private Long id;
      public ProductLoader(Long id) {
         this.id = id;
      }
      public List load() {
         return ProductMapper.create().findForSupplier(id);
      }
   }
```

load メソッドが実行している間、リストフィールドに ProductLoader を割り当てる。

```
class SupplierMapper...

   protected DomainObject doLoad(Long id, ResultSet rs) throws
      SQLException {
      String nameArg = rs.getString(2);
      SupplierVL result = new SupplierVL(id, nameArg);
      result.setProducts(new VirtualList(new ProductLoader(id)));
      return result;
   }
```

仮想リストのソースリスト自体はカプセル化され、最初の参照時にローダーを評価する。

```
class VirtualList...

   private List source;
   private VirtualListLoader loader;
   public VirtualList(VirtualListLoader loader) {
      this.loader = loader;
   }
   private List getSource() {
      if (source == null) source = loader.load();
      return source;
   }
```

委譲を行うlistメソッドは、その後ソースリストに実装される。

```
class VirtualList...

   public int size() {
      return getSource().size();
   }
   public boolean isEmpty() {
      return getSource().isEmpty();
   }
   // ... and so on for rest of list methods
```

このようにドメインクラスは、Mapperクラスが**レイジーロード**をどのように行うかは一切知ることはなく、事実**レイジーロード**があるかどうかさえも関知しないのである。

11.3.5 ｜例：バリューホルダーの使用（Java）

バリューホルダーは、汎用**レイジーロード**として使用できる。例では、Productフィールドがバリューホルダーとしてタイプ化されているため、ドメインタイプは何が進行中であるかわかる。しかし、getメソッドによってサプライヤーのクライアントからこれを隠すことができる。

```
class SupplierVH...

   private ValueHolder products;
   public List getProducts() {
```

```
        return (List) products.getValue();
    }
```

バリューホルダー自身は、**レイジーロード**の振る舞いを行う。アクセスされた時点でバリューホルダーが**レイジーロード**の振る舞いを行うための値を読み込むため、コードが渡される必要がある。そのためローダーインタフェースを定義する。

```
class ValueHolder...

    private Object value;
    private ValueLoader loader;
    public ValueHolder(ValueLoader loader) {
        this.loader = loader;
    }
    public Object getValue() {
        if (value == null) value = loader.load();
        return value;
    }
    public interface ValueLoader {
        Object load();
    }
```

マッパーは、ローダーの実装を作成し、サプライヤーオブジェクトに挿入することで、バリューホルダーを設定する。

```
class SupplierMapper...

    protected DomainObject doLoad(Long id, ResultSet rs) throws
        SQLException {
        String nameArg = rs.getString(2);
        SupplierVH result = new SupplierVH(id, nameArg);
        result.setProducts(new ValueHolder(new ProductLoader(id)));
        return result;
    }
    public static class ProductLoader implements ValueLoader {
        private Long id;
        public ProductLoader(Long id) {
            this.id = id;
        }
        public Object load() {
            return ProductMapper.create().findForSupplier(id);
        }
    }
```

11.3.6 | 例：ゴーストの使用（C#）

オブジェクトをゴーストにするロジックを、**レイヤスーパータイプ**に組み込むことができる。結果として、ゴーストを使用する場合、それはあらゆるところに現れる。そこで、ゴーストに関する検証を、まずはドメインオブジェクト**レイヤスーパータイプ**から開始する。各ドメインオブジェクトは、それがゴーストであるかどうかを認知している。

```
class Domain Object...

      LoadStatus Status;
      public DomainObject (long key) {
         this.Key = key;
      }
      public Boolean IsGhost {
         get {return Status == LoadStatus.GHOST;}
      }
      public Boolean IsLoaded {
         get {return Status == LoadStatus.LOADED;}
      }
      public void MarkLoading() {
         Debug.Assert(IsGhost);
         Status = LoadStatus.LOADING;
      }
      public void MarkLoaded() {
         Debug.Assert(Status == LoadStatus.LOADING);
         Status = LoadStatus.LOADED;
      }
   enum LoadStatus {GHOST, LOADING, LOADED};
```

ドメインオブジェクトには、ゴースト、読み込み中および読み込み済みの3つの状態がある。私は、読み取り専用プロパティと明示的なステータス変更メソッドで、ステータス情報をラップする方法を好んで用いている。

ゴーストの最も煩わしい点は、オブジェクトがゴーストである場合、読み込みのトリガとなるためすべてのアクセス関数を修正する必要がある点である。

```
class Employee...

      public String Name {
         get {
            Load();
            return _name;
         }
```

```
            set {
                Load();
                _name = value;
            }
        }
        String _name;

class Domain Object...

        protected void Load() {
            if (IsGhost) DataSource.Load(this);
        }
```

覚えるのは難しいが、バイトコードの後処理をするアスペクト指向プログラミングにとっては典型的なターゲットである。

読み込みが機能するためには、ドメインオブジェクトは該当するマッパーを呼び出す必要がある。しかし、私の経験からするとドメインコードは、マッパーコードを確認できないこともある。こうしたある特定したものに依存しないためにも、**レジストリ**と**セパレートインタフェース**の組み合わせを使うことである（図11.4）。私は、データソース操作のためにドメインに**レジストリ**を定義する。

```
class DataSource...

        public static void Load (DomainObject obj) {
            instance.Load(obj);
        }
```

データソースのインスタンスは、インタフェースを使って定義される。

```
class DataSource...

    public interface IDataSource {
        void Load (DomainObject obj);
    }
```

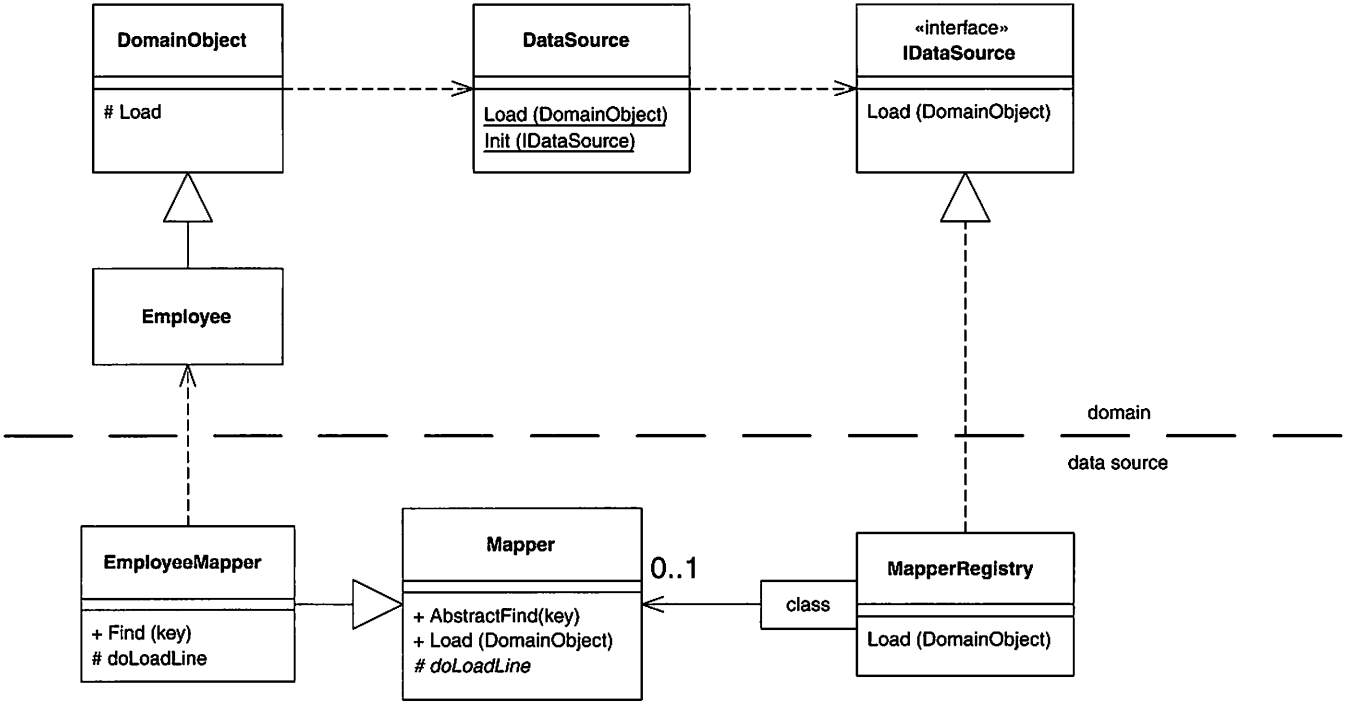


図 11.4 ——ゴーストの読み込みに含まれるクラス

データソースレイヤに定義されたマッパーのレジストリは、Data Source（データソース）インタフェースを実装する。この場合私は、マッパーをドメインタイプによってインデックス化されたディクショナリに配置してきた。Load メソッドは、正しいマッパーを検出して適切なドメインオブジェクトを読み込むように指示する。

```
class MapperRegistry : IDataSource...

      public void Load (DomainObject obj) {
         Mapper(obj.GetType()).Load (obj);
      }
      public static Mapper Mapper(Type type) {
         return (Mapper) instance.mappers[type];
      }
      IDictionary mappers = new Hashtable();
```

上記のコードは、ドメインオブジェクトとデータソースとの相互作用を示している。データソースロジックは**データマッパー**を使用する。マッパーの更新ロジックは、ゴーストがない場合と同様である。この例では、振る舞いが一連の検索および読み込みの振る舞いの中で行われている。

具象 Mapper クラスは、抽象メソッドで結果をダウンキャストする独自の Find メソッドを持っている。

```
class EmployeeMapper...

      public Employee Find (long key) {
         return (Employee) AbstractFind(key);
      }

class Mapper...

      public DomainObject AbstractFind (long key) {
         DomainObject result;
         result = (DomainObject) loadedMap[key];
         if (result == null) {
            result = CreateGhost(key);
            loadedMap.Add(key, result);
         }
         return result;
      }
      IDictionary loadedMap = new Hashtable();
```

```
    public abstract DomainObject CreateGhost(long key);

class EmployeeMapper...

    public override DomainObject CreateGhost(long key) {
        return new Employee(key);
    }
```

この例からわかるように、Find メソッドはオブジェクトを Ghost（ゴースト）状態で返す。実際のデータは、ドメインオブジェクトのプロパティへのアクセスがきっかけとなり読み込まれるまでは、データベースから取り出されない。

```
class Mapper...

    public void Load (DomainObject obj) {
        if (! obj.IsGhost) return;
        IDbCommand comm = new OleDbCommand(findStatement(),
            DB.connection);
        comm.Parameters.Add(new OleDbParameter("key",obj.Key));
        IDataReader reader = comm.ExecuteReader();
        reader.Read();
        LoadLine (reader, obj);
        reader.Close();
    }
    protected abstract String findStatement();
    public void LoadLine (IDataReader reader, DomainObject obj) {
        if (obj.IsGhost) {
            obj.MarkLoading();
            doLoadLine (reader, obj);
            obj.MarkLoaded();
        }
    }
    protected abstract void doLoadLine (IDataReader reader,
        DomainObject obj);
```

例によって**レイヤスーパータイプ**は、すべての抽象的な振る舞いを処理した後、抽象メソッドを呼び出してサブクラスごとに特定の動作を行わせる。今回の例ではデータリーダーを使用したが、これはさまざまなプラットフォームにおいて最も一般的なカーソルベースの手法である。またデータセットにも拡張できるようにしているが、.NET のほとんどのケースでより適していると思われる。

Employee オブジェクトでは、シンプルな値の Name（名前）、他のオブジェクト参照の

Department（部署）、そしてコレクションのケースを示すTimesheet Record（タイムシートレコード）のリストといった3つのプロパティを包含し、サブクラスのフックメソッドの実装において同時に読み込まれる。

```
class EmployeeMapper...

      protected override void doLoadLine (IDataReader reader,
         DomainObject obj) { Employee employee = (Employee) obj;
         employee.Name = (String) reader["name"];
         DepartmentMapper depMapper =
            (DepartmentMapper) MapperRegistry.Mapper(typeof(Department));
         employee.Department = depMapper.Find((int)
            reader["departmentID"]);
         loadTimeRecords(employee);
      }
```

Nameの値は、データリーダーの現行カーソルから列を読み取ることで簡単に読み込まれる。Departmentは、DepartmentのMapperオブジェクトのFindメソッドを使って読み込まれる。プロパティにはゴーストが設定され、実際のデータはDepartmentオブジェクト自体にアクセスしたときに読み込まれる。

コレクションは最も複雑なケースである。リプルローディングを回避するには、すべてのTime Record（タイムレコード）を1度のクエリーで読み込むことが重要である。そのためにはGhost（ゴースト）リストとして機能する特別なリストの実装が必要となる。このリストは、オブジェクトを包むラッパーであり、すべての振る舞いを委譲する。ゴーストが行う唯一のことは、リストへのアクセスを読み込みのトリガにすることである。

```
class DomainList...

      IList data {
         get {
            Load();
            return _data;
         }
         set {_data = value;}
      }
      IList _data = new ArrayList();
      public int Count {
         get {return data.Count;}
      }
```

DomainList クラスは、ドメインオブジェクトで使用され、ドメインレイヤの一部である。読み込みには SQL コマンドへのアクセスが必要であるが、私はマッピングレイヤではなく委譲を使って読み込み機能を定義している。

```
class DomainList...

      public void Load () {
         if (IsGhost) {
            MarkLoading();
            RunLoader(this);
            MarkLoaded();
         }
      }
      public delegate void Loader(DomainList list);
      public Loader RunLoader;
```

委譲を、1 つの機能に対する**セパレートインタフェース**の特定のバリエーションと考える。実際に、1 つの機能を備えたインタフェースを宣言することは、委譲を実行する有効な選択肢の 1 つである。

ローダー自身は、読み込みの SQL を指定するプロパティと、タイムレコードをマッピングするマッパーを持っている。Employee オブジェクトを読み込むとき、Employee（従業員）のマッパーはローダーを設定する。

```
class EmployeeMapper...

      void loadTimeRecords(Employee employee) {
         ListLoader loader = new ListLoader();
         loader.Sql = TimeRecordMapper.FIND_FOR_EMPLOYEE_SQL;
         loader.SqlParams.Add(employee.Key);
         loader.Mapper = MapperRegistry.Mapper(typeof(TimeRecord));
         loader.Attach((DomainList) employee.TimeRecords);
      }

class ListLoader...

      public String Sql;
      public IList SqlParams = new ArrayList();
      public Mapper Mapper;
```

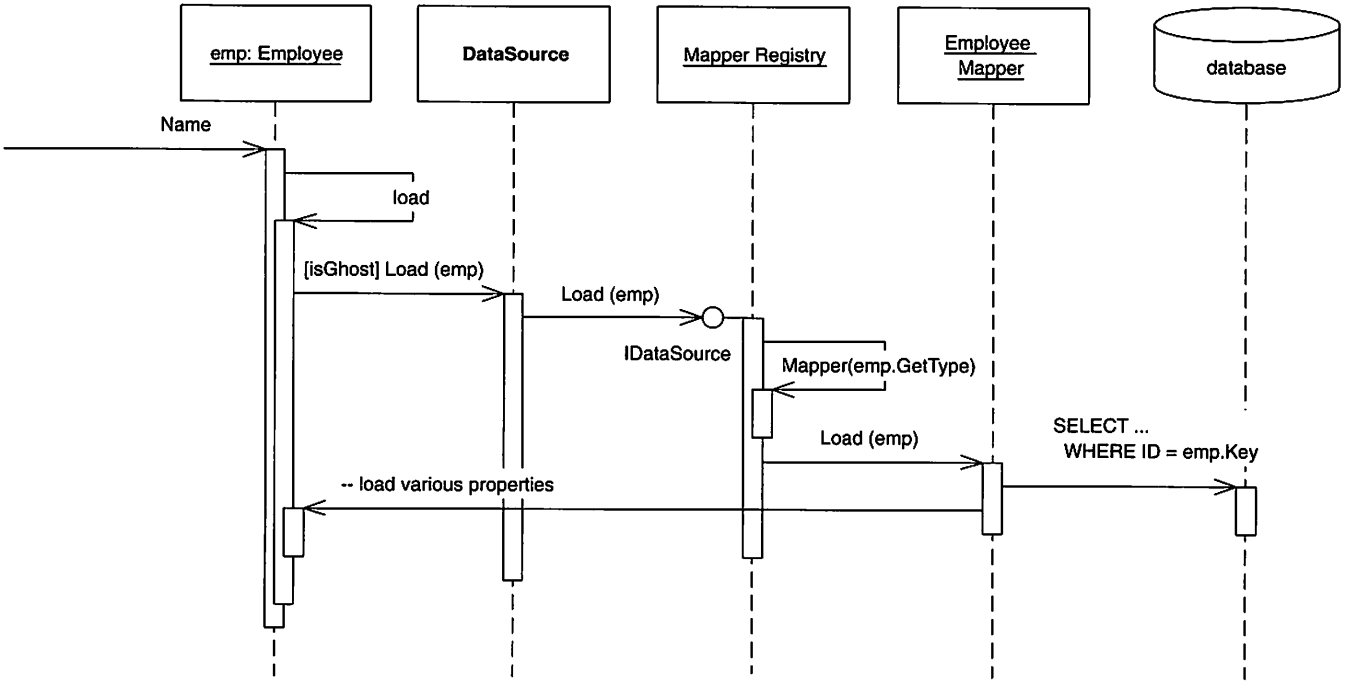


図 11.5 ——ゴーストの読み込みシーケンス

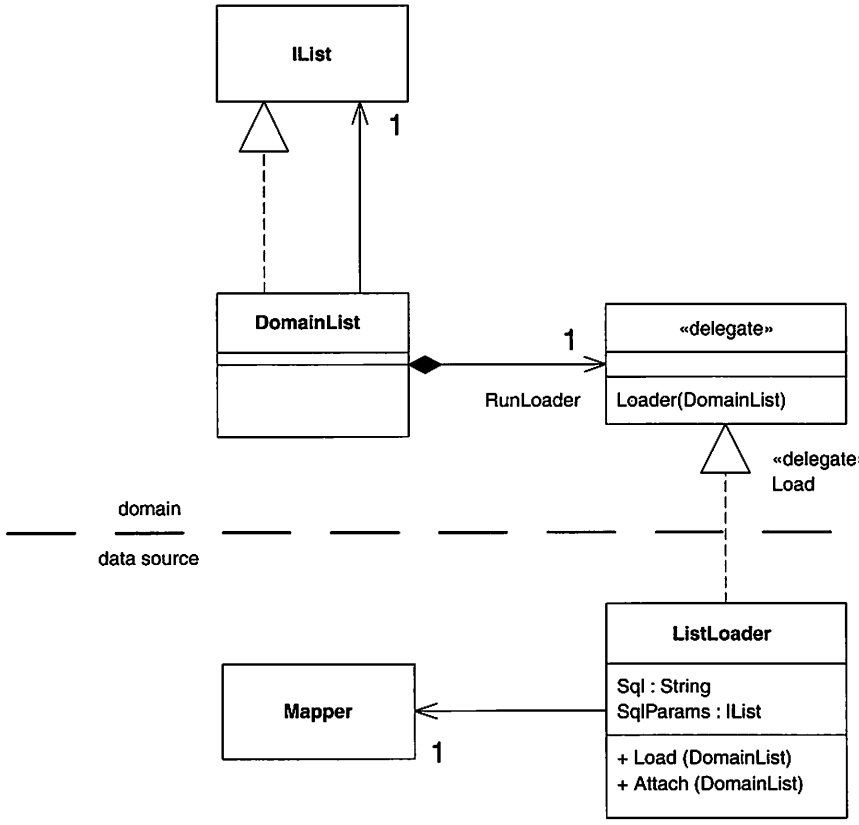


図 11.6 —— Ghost リストのクラス　現時点では、UML モデル内で委譲を示す標準的な規格はない。上記は私の現行の手法である。

委譲の割り当ての構文は複雑であるため、私はローダーに Attach メソッドを使用することにしている。

```
class ListLoader...

      public void Attach (DomainList list) {
         list.RunLoader = new DomainList.Loader(Load);
      }
```

Employee が読み込まれると、Time Record コレクションは、アクセスメソッドがローダーにトリガを発動するまで Ghost 状態となり、ローダーがクエリーを発行しリストへ記入する。

```
class ListLoader...

    public void Load (DomainList list) {
        list.IsLoaded = true;
        IDbCommand comm = new OleDbCommand(Sql, DB.connection);
        foreach (Object param in SqlParams)
            comm.Parameters.Add(new OleDbParameter(param.ToString(),param));
        IDataReader reader = comm.ExecuteReader();
        while (reader.Read()) {
            DomainObject obj = GhostForLine(reader);
            Mapper.LoadLine(reader, obj);
            list.Add (obj);
        }
        reader.Close();
    }
    private DomainObject GhostForLine(IDataReader reader) {
    return Mapper.AbstractFind((System.Int32)reader
        [Mapper.KeyColumnName]);
    }
```

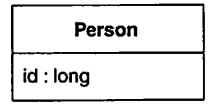
このようにGhostリストを使用することは、リプルローディングを軽減する上で重要である。しかし、他のケースでもリプルローディングは生じるため完全にゼロにすることはできない。例では、より洗練されたマッピングを使用して1つのクエリーでEmployeeのデータとともにDepartmentのデータを読み込むようにしてある。このように常にコレクション内のすべての要素をまとめて読み込むことは、混乱を回避する手助けとなるのである。

第12章

オブジェクトリレーショナル構造パターン

12.1 一意フィールド

データベースIDフィールドをオブジェクトに保存することで、メモリ上のオブジェクトとデータベース行との一意性を維持する。



リレーショナルデータベースは、キー（特にプライマリキー）によって個々の行を識別する。しかし、オブジェクトシステムでは、内部に正しいIDが確保されるため（C++の場合はローメモリの場所）、メモリ上のオブジェクトはキーを必要としない。この場合、データベースからの読み込みにはまったく問題はないが、データを書き戻すためにはデータベースとメモリ上のオブジェクトシステムとを結び付ける必要がある。

一意フィールドは大変シンプルであり、リレーショナルデータベーステーブルのプライマリキーをオブジェクトのフィールドに格納するだけである。

12.1.1 動作方法

一意フィールドの基本概念は、とてもシンプルだがさまざまな問題がある。

12.1.1.1 ■ キーの選択

最初の問題は、データベース内でどの種類のキーを選ぶかである。もちろん、必ずこのような選択が問題となるとは限らない。すでに独自のキー構造を持つ既存のデータベースが対象となる場合が多いからである。データベースコミュニティには問題に関する議論や題材が多くあり、オブジェクトへのマッピングによっては、決定までに何らかの懸案事項が追加さ

れることもある。

最初の懸案事項は、意味のあるキーと意味のないキーのどちらを使うかである。**意味のあるキー**とは、個人を識別するための米国社会保障番号のようなもので【訳注】、一方**意味のないキー**とは、データベースが作り出す無作為の数値であり、使用することを前提としていない。意味のあるキーで注意すべきことは、理論上はキーとして問題がなくても、実はそうでない点である。キーが完全に機能するためにはキーが一意でなければいけないので、正しく機能するためには不変でなければならない。しかし、割り当てられた番号は一意かつ不変であるはずだが、人為的ミスによってどちらにも当てはまらなくなってしまうことが多い。たとえば、私の妻の社会保障番号をタイプミスした場合、生成されるレコードは一意でも不変でもなくなる（ミスを修正しなければの話だ）。データベースは一意性の問題を検出するが、私のデータがシステムへ取り込まれた後であり、もちろん問題が発生するのはミスが検出された後からである。結果として、意味のあるキーは信頼を失う結果となる。小規模なシステムや安定したケースでは、有効な場合もあるかもしれないが、普通は、意味のないキーを使用する方が賢明である。

次は、単純キーと複合キーのどちらを使うかである。**単純キー**は1つのデータベースフィールドだけを使用し、**複合キー**は複数のフィールドを使用する。複合キーのメリットは、あるテーブルが他のテーブルのコンテキスト内で有効である場合に使用しやすい点にある。たとえば、Order および Line Item（明細）などはわかりやすい例であり、注文番号と連番の組み合わせは明らかにそれに当たる。複合キーはわかりやすいというメリットがある一方で、単純キーの均一性にはさまざまなメリットがある。あらゆる設定で単純キーを使用する場合、すべてのキー操作に同じコードを使用できるが、複合キーでは、具象クラスで1つずつ特定の処理を行う必要がある（コード生成では問題とはならない）。また、複合キーは意味を与えるため、一意性、特に不変性に関するルールには注意しなくてはいけない。

キー型も選択しなければならない。キーに対して頻繁に行われる操作は、等価チェックであるため、等価チェックの操作が速い型が求められる。その他の重要な操作として次のキーの取得が挙げられるが、大抵の場合は長整数型が最も無難な選択となる。文字列型も有効だが等価チェックは遅くなりインクリメントにやや負荷がかかり、そのため DBA のパフォーマンスも当然影響を受ける。

（キーに日付や時刻を使用する場合は注意が必要である。これらは意味を持つだけでなく、移植性や一貫性に影響を与えることがある。この点で日付は脆弱である。理由は、日付データには秒以下の値も格納されるが、その値が不正確になりやすく、ID の問題につながるからである）。

【訳注】米国では、全国民に社会保障番号が付与され、あらゆる局面で国民番号として使用されている。

テーブルに対して一意キー、またはデータベース全体での一意キーの使用も可能である。**テーブル一意キー**はテーブルにおいて一意であり、あらゆるケースでのキーとなる。一方、**データベース一意キー**は、データベースすべてのテーブル全体の行において一意である。普通はテーブル一意キーで十分だが、データベースキーの方が簡単に使用できる場合が多く、1つの**一意マッピング**を使用できるようになる。現在使用できる数値の桁数では、新しいキーのための数値を使い果たすことは稀である。これに不安がある場合、キーの領域をコンパクトにするシンプルなデータベーススクリプトで、削除されたオブジェクトからキーを再生することもできる。ただし、スクリプトを実行するには、アプリケーションをオフラインにする必要がある。しかし、64ビットキーを使用する場合（私はこれを推奨する）、このようなことは必要ではない。

テーブル一意キーを使用する場合は、継承に注意しなければならない。**具象テーブル継承**または**クラステーブル継承**を使用する場合、各テーブルではなく、階層構造に一意キーを使うと作業をはるかに簡素化することができる。また、**継承グラフ一意**の場合も私はあえて**テーブル一意**を使用することにしている。

キーのサイズは、特にインデックス付きの場合、パフォーマンスに影響を与える。その度合いは、データベースシステムやどのくらいの行数を管理しているかによって違うが、おおまかに確認しておくことを勧める。

12.1.1.2 ■ オブジェクトにおける一意フィールドの表記方法

最もシンプルな形式の**一意フィールド**は、データベース型のキーのフィールドである。シンプルな整数型キーを使用している場合、整数型フィールドが有効である。

複合キーはもう少し複雑で、最善の対処法はキークラスを作成することである。汎用キークラスは、キーの要素として一連のオブジェクトを格納できる。キーオブジェクトにとってキーの振る舞いは等価的である。また、データベースにマッピングする時にキーの一部分を取得する場合にも有効である。

すべてのキーに対して同じ基本構造を使用する場合、キー操作を**レイヤスーパータイプ**で行う。多くの場合、有効なデフォルトの振る舞いは**レイヤスーパータイプ**に配置し、例外的なケースに対する振る舞いは特定のサブタイプに配置するのである。

キーオブジェクトの汎用リスト上で1つのキークラスを持つと、キーの各部分に対し明示的なフィールドの各ドメインクラスのキークラスを持つこととなる。私は通常、明示的な方を好むが、今回のケースでは有効かどうかは判断できない。この場合、結局小さなクラスを数多く持つことになる。その場合の第1のメリットは、ユーザがキーの要素を間違った順序で入力してもエラーを回避できることであるが、それが大きな問題とは思えない。

異なるデータベースインスタンス間でデータをインポートする場合、覚えておく必要があるのは、異なるデータベース間でキーを分離するための手法を用意しない限り、キーの衝突

が発生することである。この問題は、インポート時におけるキーの書き換えによって解決できるが、処理はとても煩雑になりやすい。

12.1.1.3 ■ 新規キーの取得

オブジェクトを作成するには、キーが必要である。単純なことのように思われるが、新規キーの取得でもさまざまな問題が生じることがある。3つの基本的な方法は、データベースによる自動生成、GUIDの使用および独自生成である。

この中では、自動生成手法が最も容易である。データベースにデータを挿入するたびに、データベースが一意のプライマリキーを生成するため何も行う必要がない。簡単だが必ずしもすべてのデータベースに採用されている訳ではない。このようにすると、オブジェクトリレーショナルマッピングに問題が発生するデータベースも少なくない。

最も一般的な自動生成方法は、**自動生成フィールド**を宣言することである。自動生成フィールドは、行が挿入されるごとに新しい値が1ずつ増える。この手法の問題点は、どんな値がキーとして作成されるかを容易に決定できない点にある。Orderといくつかの Line Itemを挿入したい場合、Line Itemの外部キーに値を入力するには、新しいOrderのキーが必要となる。また、トランザクション内のすべてを保存するには、トランザクションのコミット前にキーが必要となる。残念ながら、データベースからはこのような情報が得られないため、関連オブジェクトを挿入したいテーブルでは自動生成を使用できない。

自動生成の代替手法は、Oracleがシーケンスに使用している**データベースカウンタ**である。Oracleのシーケンスは、シーケンスを参照するselect文を送信して機能する。送信後、データベースは次のシーケンス値から成るSQLレコードセットを返す。シーケンスを任意の整数ずつ増分するように設定して複数のキーを一度に取得することができる。シーケンスクエリーは、独立したトランザクションにおいて自動的に実行されるため、シーケンスへのアクセスは、同時に挿入を行う他のトランザクションをロックすることはない。このようなデータベースカウンタは私たちのニーズに合うが、非標準機能でありすべてのデータベースで使用できるとは限らない。

GUID（Globally Unique IDentifier）とは、1台のマシンで生成される数値で、あらゆるマシン上において一意であることが保証される。プラットフォームには、GUIDを生成するAPIが用意されている。アルゴリズムは、イーサネットカードアドレス、ナノ秒単位の時間帯、チップID番号他にも応用することができる。重要なのは、数値が完全一意であり、安全なキーとなる点である。GUIDの唯一の短所は、得られるキー文字列が長くなることである。ウィンドウやSQL式にキー入力が必要な時、長いキーの場合、入力読み込みが大変で、パフォーマンス上の問題（特にインデックスが存在する場合）も発生する。

これ以外に考えられる方法としては、自分自身で作成することである。小規模システムの基本は、SQLのmax関数を使用した**テーブルスキャン**によってテーブルでの最大のキーを

見つけ、キーを追加して使用することである。データの挿入が頻繁でない場合には有効であるが、残念ながら、処理中はテーブル全体が読み込みロックされてしまう。そのため、同一テーブル上で更新と同時に挿入を行うときには、パフォーマンスは極めて低下するのである。さらにトランザクションを互いに完全に分離しておく必要がある。完全に分離していないと複数のトランザクションが同じID値を取得してしまうからである。

また、追加するアプローチとしては、独立した**キーテーブル**の使用である。このテーブルは2つの列から構成されていて、一方の列は名前に、もう一方は次に使用可能な値に使われる。データベースの一意キーを使用する場合、テーブルには1行だけ含まれる。一方、テーブルの一意キーを使用する場合、データベースのテーブルごとに1行が含まれる。キーテーブルを使用する場合は、その1行を読み込み、値を増やした上で行に書き戻すだけでよい。キーテーブルを更新する場合、数字を追加することで一度に複数のキーを取得できる。これによりデータベースの呼び出しにかかる負荷が削減され、キーテーブルでの競合は回避される。

キーテーブルを使用する場合、キーテーブルへのアクセスが、挿入先のテーブルを更新するトランザクションから独立したトランザクションで行われるように設計することをお勧めする。たとえば、Orderテーブルに Orderを挿入するとしよう。それには、(更新を行うため) 書き込みロックによってキーテーブルのOrder行をロックする必要がある。このロック状態は私が関わるトランザクション全体に継続し、キーを求める他のユーザをすべてロックする。これは、テーブルの一意キーの場合、Orderテーブルへの挿入を行うすべてのユーザに影響し、データベースの一意キーの場合、あらゆる場所で挿入を行うすべてのユーザに影響する。

独立したトランザクションでキーテーブルへのアクセスを行うことで、必要な行がロックされトランザクションも短縮される。この方法の短所は、注文への挿入をロールバックする場合、キーテーブルから取得したキーが、二度と使用されなくなってしまう点である。しかし、数値はいくらでもあるので、大した問題ではない。分離したトランザクションを使用してメモリ上のオブジェクトを作成するとすぐにIDを取得することができるようになる。ビジネストランザクションをコミットするためにトランザクションをオープンする前に行うことが多い。

キーテーブルを使用するかどうかは、データベースの一意キーとテーブルの一意キーの選択にも影響する。テーブルの一意キーを使用する場合、データベースにテーブルを追加するたびに、キーテーブルに行を追加しなければならない。これは手間のかかる作業だが、その行での競合を避ける効果がある。キーテーブルアクセスを異なるトランザクションに保持する場合、競合はそれほど大きな問題にはならない。一度の呼び出しで複数のキーを取得する場合にも特に問題にならない。しかし、キーテーブルの更新を分離したトランザクションとして設定できない場合、データベースの一意キーを使用する強い理由となる。

テスト目的の**サービススタブ**の構築が容易となるため、新しいキーを取得するためのコードを、独立したクラスに分離することをお勧めする。

12.1.2 | 使用するタイミング

メモリ上のオブジェクトとデータベース行の間にマッピングがある場合は**一意フィールド**を使用する。**ドメインモデル**または**行データゲートウェイ**を使用するときには、これが一般的である。しかし**トランザクションスクリプト**、**テーブルモジュール**、**テーブルデータゲートウェイ**を使用する場合は、このようなマッピングは必要ない。

値のセマンティックスを持つ小さなオブジェクトの場合（独自のテーブルを持たないMoney（貨幣）やDate Range（日付範囲）オブジェクトなど）、**組込バリュー**の使用を勧める。リレーショナルデータベース内でクエリーを発行する必要のないオブジェクトの複雑なグラフの場合、**シリアライズLOB**への書き込みが容易であり高速なパフォーマンスを提供できる。

一意フィールドの代替案としては、**一意マッピング**による対応が挙げられ、メモリ上のオブジェクトに**一意フィールド**を格納したくないシステムで使用できる。**一意マッピング**はオブジェクトのためのキーを与えるか、逆にキーのためのオブジェクトを与えるかのいずれかで参照を行う必要がある。オブジェクトにキーを格納するほうが容易なので、私はこの方法をあまり目にすることはない。

12.1.3 | 参考文献

[Marinescu]は、キー生成のためのいくつかの技法を解説している。

12.1.4 | 例：整数型キー（C#）

一意フィールドの最もシンプルな形式は、データベースにおける整数型フィールドであり、メモリ上のオブジェクトの整数型フィールドにマッピングされる。

```
class DomainObject...

      public const long PLACEHOLDER_ID = -1;
      public long Id = PLACEHOLDER_ID;
      public Boolean isNew() {return Id == PLACEHOLDER_ID;}
```

メモリ上には作成されるがデータベースに保存されないオブジェクトは、キーの値を持つことはない。.NETの値オブジェクトの場合、これは問題となる。なぜなら.NETの値は

null（設定なし）となれないからである。したがって、プレースホルダー値である。

キーは、検索と挿入の２つの用途で重要である。検索の場合、.NET では where 句の中でキーを使用するクエリーを作成し、データセットに複数の行を読み込んでから、検索操作によって特定の行を選択する。

```
class CricketerMapper...

      public Cricketer Find(long id) {
         return (Cricketer) AbstractFind(id);
      }

class Mapper...

      protected DomainObject AbstractFind(long id) {
         DataRow row = FindRow(id);
         return (row == null) ? null : Find(row);
      }
      protected DataRow FindRow(long id) {
         String filter = String.Format("id = {0}", id);
         DataRow[] results = table.Select(filter);
         return (results.Length == 0) ? null : results[0];
      }
      public DomainObject Find (DataRow row) {
         DomainObject result = CreateDomainObject();
         Load(result, row);
         return result;
      }
      abstract protected DomainObject CreateDomainObject();
```

この振る舞いの大部分は**レイヤスーパータイプ**によりできるが、ダウンキャストをカプセル化するためだけでも、具象クラスに find メソッドを定義しなければならない場合が多い。当然だが、コンパイル時に型付けしないプログラミング言語においては、このようなことは必要でない。

シンプルな整数型**一意フィールド**によって、挿入の振る舞いもまた**レイヤスーパータイプ**で保持される。

```
class Mapper...

      public virtual long Insert (DomainObject arg) {
         DataRow row = table.NewRow();
         arg.Id = GetNextID();
```

```
        row["id"] = arg.Id;
        Save (arg, row);
        table.Rows.Add(row);
        return arg.Id;
    }
```

基本的には、挿入は新しい行を作成することと、その行のための次のキーを使用することである。キーを取得した後は、新しい行にメモリ上のオブジェクトのデータを保存する。

12.1.5 | 例：キーテーブルの使用（Java）

(by Matt Foemmel, Martin Fowler)

使用するデータベースがデータベースカウンタをサポートし、SQL に依存するのであればカウンタを使用すべきである。データベースへ依存しない場合であっても、考慮する価値はある。キー生成コードが問題なくカプセル化されている限り、いつでも移植可能なアルゴリズムへと変更できるからである。キーがすでにある場合はカウンタを使用し、ない場合は独自に作成するストラテジー[Gang of Four]を採用することである。

ここでは、難しい方法でカウンタを使用すると仮定しよう。最初に必要となるのは、データベースのキーテーブルである。

```
CREATE TABLE keys (name varchar primary key, nextID int)
INSERT INTO keys VALUES ('orders', 1)
```

テーブルには、データベース内の各カウンタに対応する行が 1 行ずつ含まれている。ここでは、キーの初期値は 1 に設定している。データベースのデータを事前ローディングする場合は、カウンタを最適値に設定する必要がある。データベースの一意キーが必要な場合、1 行だけ必要であり、テーブルの一意キーが必要な場合、テーブルごとに 1 行が必要となる。

すべてのキー生成コードを独自のクラスにラップする。これで 1 つ以上のアプリケーションにおいて幅広く使用できるようになり、独自のトランザクションにキー予約を登録することが容易となる。

ここでは、一度にデータベースからいくつのキーを取り出すかという情報に加えて、独自のデータベースとの接続によってキー生成コードを構築している。

```
class KeyGenerator...

    private Connection conn;
    private String keyName;
```

```
    private long nextId;
    private long maxId;
    private int incrementBy;
    public KeyGenerator(Connection conn, String keyName, int incrementBy) {
        this.conn = conn;
        this.keyName = keyName;
        this.incrementBy = incrementBy;
        nextId = maxId = 0;
        try {
            conn.setAutoCommit(false);
        } catch(SQLException exc) {
            throw new ApplicationException("Unable to turn off
            autocommit", exc);
        }
    }
```

選択および更新操作を間違いなく1つのトランザクションで行うため、自動コミットが動作しないようにする必要がある。

新しいキーを要求する場合、生成コードはデータベースにアクセスするのではなく、キャッシュされているキーがあるかどうかを確認する。

```
class KeyGenerator...

    public synchronized Long nextKey() {
        if (nextId == maxId) {
            reserveIds();
        }
        return new Long(nextId++);
    }
```

キャッシュされているキーが得られない場合は、データベースにアクセスする。

```
class KeyGenerator...

    private void reserveIds() {
        PreparedStatement stmt = null;
        ResultSet rs = null;
        long newNextId;
        try {
            stmt = conn.prepareStatement("SELECT nextID FROM keys
                WHERE name = ? FOR UPDATE");
```

```
            stmt.setString(1, keyName);
            rs = stmt.executeQuery();
            rs.next();
            newNextId = rs.getLong(1);
        }
        catch (SQLException exc) {
            throw new ApplicationException("Unable to generate ids", exc);
        }
        finally {
            DB.cleanUp(stmt, rs);
        }
        long newMaxId = newNextId + incrementBy;
        stmt = null;
        try {
            stmt = conn.prepareStatement("UPDATE keys SET nextID = ?
                WHERE name = ?");
            stmt.setLong(1, newMaxId);
            stmt.setString(2, keyName);
            stmt.executeUpdate();
            conn.commit();
            nextId = newNextId;
            maxId = newMaxId;
        }
        catch (SQLException exc) {
            throw new ApplicationException("Unable to generate ids", exc);
        }
        finally {
            DB.cleanUp(stmt);
        }
    }
```

この例では、SELECT... FOR UPDATEを使って、データベースに対してキーテーブルの書き込みロックを保持するよう指示している。これはOracle独自のステートメントであり、他のデータベースを使用している場合には、別の方法が必要となる。選択時に書き込みロックができないと、割り込みがあった場合にトランザクションが失敗するという危険性がある。この場合は、新しいキーセットを取得するまで、reserveIdsを繰り返し実行しても問題はない。

12.1.6 ｜例：複合キーの使用（Java）

シンプルな整数型キーの使用は、簡単で優れた解決法ではあるが、他のタイプまたは複合キーが必要となる場合も多い。

12.1.6.1 ■ キークラス

他の方法が必要となった場合には、すぐにキークラスの作成を検討することは有効である。キークラスは複数のキーの要素を格納するため、2つのキーが等しいかどうかを判断できなければならない。

```
class Key...

   private Object[] fields;
   public boolean equals(Object obj) {
      if (!(obj instanceof Key)) return false;
      Key otherKey = (Key) obj;
      if (this.fields.length != otherKey.fields.length) return false;
      for (int i = 0; i < fields.length; i++)
         if (!this.fields[i].equals(otherKey.fields[i])) return false;
      return true;
   }
```

最も基本的なキーの作成方法は、配列パラメータによる方法である。

```
class Key...
   public Key(Object[] fields) {
      checkKeyNotNull(fields);
      this.fields = fields;
   }
   private void checkKeyNotNull(Object[] fields) {
      if (fields == null) throw new IllegalArgumentException("Cannot
         have a null key");
      for (int i = 0; i < fields.length; i++)
         if (fields[i] == null) throw new IllegalArgumentException
            ("Cannot have a null element of key");
   }
```

一般的に、ある要素によってキーを作成する場合、使いやすいコンストラクタを追加することができる。該当するキーは、アプリケーションが持つキーの種類に依存する。

```
class Key...

   public Key(long arg) {
      this.fields = new Object[1];
      this.fields[0] = new Long(arg);
   }
```

```
    public Key(Object field) {
        if (field == null) throw new IllegalArgumentException
            ("Cannot have a null key");
        this.fields = new Object[1];
        this.fields[0] = field;
    }
    public Key(Object arg1, Object arg2) {
        this.fields = new Object[2];
        this.fields[0] = arg1;
        this.fields[1] = arg2;
        checkKeyNotNull(fields);
    }
```

コンビニエンスメソッドは、積極的に追加すべきである。使いやすさは、すべてのユーザにとって最も重要なのである。

同様に、キーの一部を取得するアクセッサー機能を追加することができる。アプリケーションでは、マッピングの際にこれを行う必要がある。

```
class Key...

    public Object value(int i) {
        return fields[i];
    }
    public Object value() {
        checkSingleKey();
        return fields[0];
    }
    private void checkSingleKey() {
        if (fields.length > 1)
            throw new IllegalStateException("Cannot take value on
                composite key");
    }
    public long longValue() {
        checkSingleKey();
        return longValue(0);
    }
    public long longValue(int i) {
        if (!(fields[i] instanceof Long))
            throw new IllegalStateException("Cannot take longValue
                on non long key");
        return ((Long) fields[i]).longValue();
    }
```

この例では、Orders テーブルおよびLine Items テーブルに対してマッピングしている。Orders テーブルは、シンプルな整数型プライマリキーを持ち、Line Items テーブルのプライマリキーは、Orders テーブルのプライマリキーと連続した番号の組み合わせとなる。

```
CREATE TABLE orders (ID int primary key, customer varchar)
CREATE TABLE line_items (orderID int, seq int, amount int, product varchar,
                         primary key (orderID, seq))
```

ドメインオブジェクトの**レイヤスーパータイプ**は、キーフィールドを持つ必要がある。

```
class DomainObjectWithKey...

   private Key key;
   protected DomainObjectWithKey(Key ID) {
      this.key = ID;
   }
   protected DomainObjectWithKey() {
   }
   public Key getKey() {
      return key;
   }
   public void setKey(Key key) {
      this.key = key;
   }
```

12.1.6.2 ■ 読み込み

他の例と同様、私は、振る舞いを（データベースの正しい行へアクセスする）find メソッドと、（その行のデータをドメインオブジェクトに読み込む）load メソッドに分割している。いずれの役割も、キーオブジェクトによって影響を受ける。

ここに示す例と、（シンプルな整数型キーを使用した）その他の例との最大の違いは、より複雑なキーを持つクラスによってオーバーライドされる振る舞いの特定の部分を分析できる点である。例の場合、ほとんどのテーブルは、シンプルな整数型キーを使用することを前提としている。しかし、一部ではその他のキーも使用されているため、デフォルトのケースをシンプルな整数型として、その振る舞いをマッパー**レイヤスーパータイプ**に組み込んでいる。Order クラスは、このようなシンプルなケースの 1 つである。find メソッドの振る舞いのコードは以下のとおりになる。

```
class OrderMapper...

   public Order find(Key key) {
      return (Order) abstractFind(key);
   }
   public Order find(Long id) {
      return find(new Key(id));
   }
   protected String findStatementString() {
      return "SELECT id, customer FROM orders WHERE id = ?";
   }

class AbstractMapper...

   abstract protected String findStatementString();
   protected Map loadedMap = new HashMap();
   public DomainObjectWithKey abstractFind(Key key) {
      DomainObjectWithKey result = (DomainObjectWithKey) loadedMap.get(key);
      if (result != null) return result;
      ResultSet rs = null; PreparedStatement findStatement = null;
      try {
         findStatement = DB.prepare(findStatementString());
         loadFindStatement(key, findStatement);
         rs = findStatement.executeQuery();
         rs.next();
         if (rs.isAfterLast()) return null;
         result = load(rs);
         return result;
      } catch (SQLException e) {
         throw new ApplicationException(e);
      } finally {
         DB.cleanUp(findStatement, rs);
      }
   }
   // hook method for keys that aren' t simple integral
   protected void loadFindStatement(Key key, PreparedStatement
      finder) throws SQLException {
      finder.setLong(1, key.longValue());
   }
```

ここではfind文の構造部分を抽出しているが、これはあらかじめ用意されたステートメントに異なるパラメータを渡す必要があるためである。Line Itemsは、複合キーであるため、このメソッドをオーバーライドする。

```
class LineItemMapper...

   public LineItem find(long orderID, long seq) {
      Key key = new Key(new Long(orderID), new Long(seq));
      return (LineItem) abstractFind(key);
   }
   public LineItem find(Key key) {
      return (LineItem) abstractFind(key);
   }
   protected String findStatementString() {
   return
      "SELECT orderID, seq, amount, product " +
      " FROM line_items " +
      " WHERE (orderID = ?) AND (seq = ?)";
   }
   // hook methods overridden for the composite key
   protected void loadFindStatement(Key key, PreparedStatement
      finder) throws SQLException {
      finder.setLong(1, orderID(key));
      finder.setLong(2, sequenceNumber(key));
   }
   //helpers to extract appropriate values from line item' s key
   private static long orderID(Key key) {
      return key.longValue(0);
   }
   private static long sequenceNumber(Key key) {
      return key.longValue(1);
   }
```

このサブクラスはfindメソッド用のインタフェースを定義し、find文にSQL文字列を提供するだけでなく、2つのパラメータをSQL文に送るため、フックメソッドをオーバーライドする必要がある。ここでも私は、キー情報の一部を抽出するために、2つのヘルパーメソッドを記述している。これはキーからの数値インデックスを持った明示的なアクセッサーを追加するよりも明確なコードの記述に役立つ。ただし、インデックスにはリスクがある。

loadメソッドの振る舞いも同様の構造をしている。シンプルな整数型キーに対しては、**レイヤスーパータイプ**のデフォルトの振る舞いを用意し、複雑なケースではオーバーライドを実行する。このケースでは、Orderのloadメソッドの振る舞いは以下のとおりである。

```
class AbstractMapper...

   protected DomainObjectWithKey load(ResultSet rs) throws SQLException {
```

```
        Key key = createKey(rs);
        if (loadedMap.containsKey(key)) return
            (DomainObjectWithKey) loadedMap.get(key);
        DomainObjectWithKey result = doLoad(key, rs);
        loadedMap.put(key, result);
        return result;
    }
    abstract protected DomainObjectWithKey doLoad(Key id,
        ResultSet rs) throws SQLException;
    // hook method for keys that aren't simple integral
    protected Key createKey(ResultSet rs) throws SQLException {
        return new Key(rs.getLong(1));
    }

class OrderMapper...

    protected DomainObjectWithKey doLoad(Key key, ResultSet rs)
        throws SQLException {
        String customer = rs.getString("customer");
        Order result = new Order(key, customer);
        MapperRegistry.lineItem().loadAllLineItemsFor(result);
        return result;
    }
```

LineItem は、2 つのフィールドに基づいたキーを作成するため、フックメソッドをオーバーライドする必要がある。

```
class LineItemMapper...

    protected DomainObjectWithKey doLoad(Key key, ResultSet rs)
        throws SQLException {
        Order theOrder = MapperRegistry.order().find(orderID(key));
        return doLoad(key, rs, theOrder);
    }
    protected DomainObjectWithKey doLoad(Key key, ResultSet rs,
        Order order) throws SQLException
    {
        LineItem result;
        int amount = rs.getInt("amount");
        String product = rs.getString("product");
        result = new LineItem(key, amount, product);
        order.addLineItem(result);//links to the order
        return result;
```

```
    }
    //overrides the default case
    protected Key createKey(ResultSet rs) throws SQLException {
        Key key = new Key(new Long(rs.getLong("orderID")), new
            Long(rs.getLong("seq")));
        return key;
    }
```

また、LineItem は、Order に対して LineItem を読み込むときに使用される独立した load メソッドを持っている。

```
class LineItemMapper...

    public void loadAllLineItemsFor(Order arg) {
        PreparedStatement stmt = null;
        ResultSet rs = null;
        try {
            stmt = DB.prepare(findForOrderString);
            stmt.setLong(1, arg.getKey().longValue());
            rs = stmt.executeQuery();
            while (rs.next())
                load(rs, arg);
        } catch (SQLException e) {
            throw new ApplicationException(e);
        } finally {
                DB.cleanUp(stmt, rs);
        }
    }
    private final static String findForOrderString =
            "SELECT orderID, seq, amount, product " +
            "FROM line_items " +
            "WHERE orderID = ?";
    protected DomainObjectWithKey load(ResultSet rs, Order order)
        throws SQLException {
        Key key = createKey(rs);
        if (loadedMap.containsKey(key)) return
            (DomainObjectWithKey) loadedMap.get(key);
        DomainObjectWithKey result = doLoad(key, rs, order);
        loadedMap.put(key, result);
        return result;
    }
```

Order オブジェクトは、作成後まで Order の**一意マッピング**に挿入されないので、あらかじめ空のオブジェクトを作成し**一意フィールド**に挿入しておく。

12.1.6.3 ■ 挿入

読み込みの場合と同様、挿入もシンプルな整数型キーに対してのデフォルトのアクションと、より複雑なキーにアクションをオーバーライドするフックメソッドを持つ。マッパースーパータイプでは、挿入操作を行うテンプレートメソッドとインタフェースの機能を提供する。

```
class AbstractMapper...

   public Key insert(DomainObjectWithKey subject) {
      try {
         return performInsert(subject, findNextDatabaseKeyObject());
      } catch (SQLException e) {
         throw new ApplicationException(e);
      }
   }
   protected Key performInsert(DomainObjectWithKey subject, Key
      key) throws SQLException {
      subject.setKey(key);
      PreparedStatement stmt = DB.prepare(insertStatementString());
      insertKey(subject, stmt);
      insertData(subject, stmt);
      stmt.execute();
      loadedMap.put(subject.getKey(), subject);
      return subject.getKey();
   }
   abstract protected String insertStatementString();

class OrderMapper...

   protected String insertStatementString() {
      return "INSERT INTO orders VALUES(?,?)";
   }
```

このオブジェクトからのデータは、オブジェクトの基本データとキーのデータとを分離する2つのメソッドを介し insert 文へ送られる。私がこの方法を使用する理由は、Order などデフォルトのシンプルな整数型キーを使うクラスに対して機能するキーに、デフォルトの実装を提供できるからである。

```
class AbstractMapper...

    protected void insertKey(DomainObjectWithKey subject,
          PreparedStatement stmt) throws SQLException
    {
       stmt.setLong(1, subject.getKey().longValue());
    }
```

insert 文の残りデータは特定のサブクラスにあるため、振る舞いはスーパークラスにおいては抽象的である。

```
class AbstractMapper...

    abstract protected void insertData(DomainObjectWithKey
          subject, PreparedStatement stmt) throws SQLException;

class OrderMapper...

    protected void insertData(DomainObjectWithKey
          abstractSubject, PreparedStatement stmt) {
       try {
          Order subject = (Order) abstractSubject;
          stmt.setString(2, subject.getCustomer());
       } catch (SQLException e) {
          throw new ApplicationException(e);
       }
    }
```

LineItem は、メソッドの両方をオーバーライドしてキーに 2 つの値を取り出す。

```
class LineItemMapper...

    protected String insertStatementString() {
       return "INSERT INTO line_items VALUES (?, ?, ?, ?)";
    }
    protected void insertKey(DomainObjectWithKey subject,
       PreparedStatement stmt) throws SQLException
    {
       stmt.setLong(1, orderID(subject.getKey()));
       stmt.setLong(2, sequenceNumber(subject.getKey()));
    }
```

また、残りのデータに対しては、insert 文独自の実装を提供する。

```
class LineItemMapper...

   protected void insertData(DomainObjectWithKey subject,
         PreparedStatement stmt) throws SQLException
   {
      LineItem item = (LineItem) subject;
      stmt.setInt(3, item.getAmount());
      stmt.setString(4, item.getProduct());
   }
```

このように、insert 文にデータ読み込みを配置することが有効なのは、ほとんどのクラスがキーとして同一フィールドを使用している場合だけである。キー処理により多くのバリエーションがある場合、情報の挿入を 1 つのコマンドとした方が容易な場合もある。

また、次のデータベースキーを作成して、デフォルトのケースとオーバーライドされるケースとに分離することもできる。デフォルトのケースには、すでに解説したキーテーブルスキームを使用できる。しかし、Line Items に対して使用する場合には問題が生じる。それは Line Items のキーは、複合キーの一部として Order のキーを使用するが、LineItem クラスから Order クラスへの参照がないため、正しい Order の提供なしで、LineItem 自身をデータベースに挿入できないためである。そのため、スーパークラスのメソッドをオーバーライドして、UnSupportedOprationException を起こすという手法を取らざるを得なくなる。

```
class LineItemMapper...

   public Key insert(DomainObjectWithKey subject) {
      throw new UnsupportedOperationException
            ("Must supply an order when inserting a line item");
   }
   public Key insert(LineItem item, Order order) {
      try {
         Key key = new Key(order.getKey().value(),
            getNextSequenceNumber(order));
         return performInsert(item, key);
      } catch (SQLException e) {
         throw new ApplicationException(e);
      }
   }
```

もちろん、LineItem から Order へのバックリンクを用意し、効率的に双方向を関連付けて事態は回避できる。しかし、私がこの方法を選択しないのは、リンクがない場合にどう行うべきかを説明するためである。

Order を提供することで、Order 部分のキーの取得が容易になる。次は、LineItem に対するシーケンス番号の作成である。シーケンス番号を検索するには、Order に対して次にどのシーケンス番号が使用可能であるかを確認する必要がある。そのため、SQL の max クエリーを発行するか、あるいはメモリ上の Order に含まれる LineItem を参照する必要がある。この例では、後者の方法を取り上げる。

```
class LineItemMapper...

   private Long getNextSequenceNumber(Order order) {
      loadAllLineItemsFor(order);
      Iterator it = order.getItems().iterator();
      LineItem candidate = (LineItem) it.next();
      while (it.hasNext()) {
         LineItem thisItem = (LineItem) it.next();
         if (thisItem.getKey() == null) continue;
         if (sequenceNumber(thisItem) >
            sequenceNumber(candidate)) candidate = thisItem;
      }
      return new Long(sequenceNumber(candidate) + 1);
   }
   private static long sequenceNumber(LineItem li) {
      return sequenceNumber(li.getKey());
   }
   //comparator doesn't work well here due to unsaved null keys
   protected String keyTableRow() {
      throw new UnsupportedOperationException();
   }
```

Collections.maxメソッドを使用すると、アルゴリズムは簡略化されるが、少なくとも 1 つの null（設定なし）キーが存在するので、うまくいかない。

12.1.6.4 ■ 更新と削除

今まで解説した内容を応用すれば、更新と削除の実装は容易である。繰り返しになるが、予測される一般的ケースに対しては抽象メソッドを使用し、特定のケースにおいてはオーバーライドメソッドを使用している。

更新の動作は以下のとおりである。

```
class AbstractMapper...

   public void update(DomainObjectWithKey subject) {
      PreparedStatement stmt = null;
      try {
         stmt = DB.prepare(updateStatementString());
         loadUpdateStatement(subject, stmt);
         stmt.execute();
      } catch (SQLException e) {
         throw new ApplicationException(e);
      } finally {
         DB.cleanUp(stmt);
      }
   }
   abstract protected String updateStatementString();
   abstract protected void loadUpdateStatement
         (DomainObjectWithKey subject, PreparedStatement stmt)
         throws SQLException;

class OrderMapper...

   protected void loadUpdateStatement(DomainObjectWithKey
      subject, PreparedStatement stmt) throws SQLException
   {
      Order order = (Order) subject;
      stmt.setString(1, order.getCustomer());
      stmt.setLong(2, order.getKey().longValue());
   }
   protected String updateStatementString() {
      return "UPDATE orders SET customer = ? WHERE id = ?";
   }

class LineItemMapper...

   protected String updateStatementString() {
      return
         "UPDATE line_items " +
         " SET amount = ?, product = ? " +
         " WHERE orderId = ? AND seq = ?";
   }
   protected void loadUpdateStatement(DomainObjectWithKey
      subject, PreparedStatement stmt) throws SQLException
   {
      stmt.setLong(3, orderID(subject.getKey()));
```

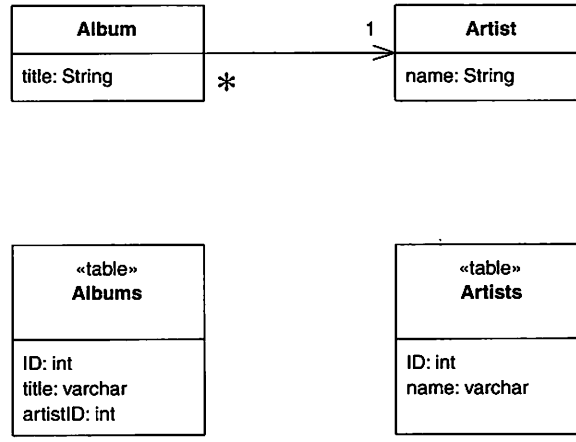
```
        stmt.setLong(4, sequenceNumber(subject.getKey()));
        LineItem li = (LineItem) subject;
        stmt.setInt(1, li.getAmount());
        stmt.setString(2, li.getProduct());
    }
```

削除の動作は以下のとおりである。

```
class AbstractMapper...

    public void delete(DomainObjectWithKey subject) {
        PreparedStatement stmt = null;
        try {
            stmt = DB.prepare(deleteStatementString());
            loadDeleteStatement(subject, stmt);
            stmt.execute();
        } catch (SQLException e) {
            throw new ApplicationException(e);
        } finally {
            DB.cleanUp(stmt);
        }
    }
    abstract protected String deleteStatementString();
    protected void loadDeleteStatement(DomainObjectWithKey
        subject, PreparedStatement stmt) throws SQLException
    {
        stmt.setLong(1, subject.getKey().longValue());
    }
class OrderMapper...

    protected String deleteStatementString() {
        return "DELETE FROM orders WHERE id = ?";
    }

class LineItemMapper...

    protected String deleteStatementString() {
        return "DELETE FROM line_items WHERE orderid = ? AND seq = ?";
    }
    protected void loadDeleteStatement(DomainObjectWithKey
        subject, PreparedStatement stmt) throws SQLException
    {
        stmt.setLong(1, orderID(subject.getKey()));
        stmt.setLong(2, sequenceNumber(subject.getKey()));
    }
```

12.2 | 外部キーマッピング

オブジェクト間の関係を、テーブル間の外部キー参照にマッピングする。



オブジェクトは、オブジェクト参照によって直接相互に参照し合う。シンプルなオブジェクト指向システムも、さまざまな方法で相互に関連付けられた一連のオブジェクトを含んでいる。これらのオブジェクトをデータベースに保存するには、参照を保存することが不可欠である。しかし、参照に含まれるデータは、実行中のプログラムの特定のインスタンスに固有であるため、そのままのデータ値を保存することはできない。その上、オブジェクトは他のオブジェクトへの参照のコレクションを容易に保持できるのである。このような構造は、リレーショナルデータベースの標準形式に反している。

外部キーマッピングは、オブジェクト参照をデータベースの外部キーにマッピングしている。

12.2.1 | 動作方法

上述の問題に明確な鍵となるのが、**一意フィールド**である。それぞれのオブジェクトには、データベーステーブルからのデータベースキーが含まれる。2つのオブジェクトの関連性によってリンクしている場合、データベースの外部キーによって置き換えることができる。つまり、図12.1に示すように、データベースにAlbumを保存する場合、Albumがリンクされるの IDは、Albumレコードに保存される。

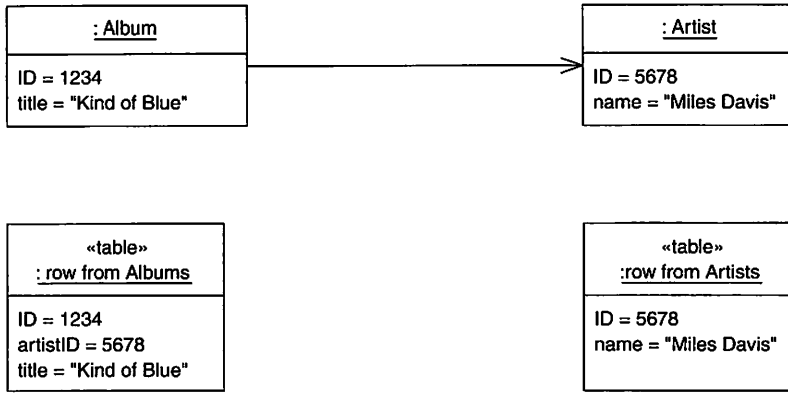


図 12.1 ——外部キーへのコレクションのマッピング

これはシンプルなケースである。オブジェクトのコレクションを持つと、さらに複雑なケースが発生する。データベースにコレクションは保存できないため、参照項目を変える必要がある。そのため、図 12.2 および 12.3 に示すように、Album の Track というコレクションで Album の外部キーを Track レコードに配置する。複雑さが発生するのは、更新を行う時点である。更新とは、Album 内のコレクションに対して Track の追加や削除ができるのである。その際、どのような方法で、データベースに含むべき変更部分を指示できるだろうか。基本的には、(1) 削除と挿入、(2) バックポインタの追加、(3) コレクションの差分表示という 3 つのオプションから選択できる。

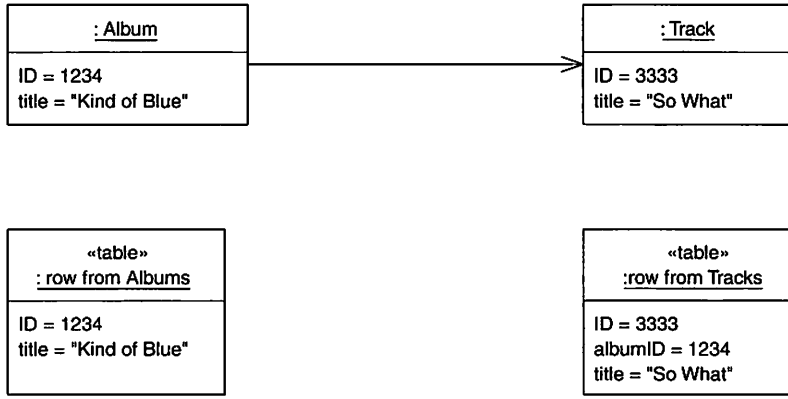


図 12.2 ——外部キーへのコレクションのマッピング

削除と挿入の場合、Album にリンクされているデータベース内のすべての Track を削除した後、Album 上に現在あるすべての Track を挿入する。一見すると、この方法はとても

大胆で危険なもののように思えるかもしれない。特にTrackをまったく変更していない場合などはそう思うだろう。しかし、ロジックは実装が容易であり、他の方法と比較してもとても有効に機能する。短所は、この方法を使用できるのはTrackが**依存マッピング**である場合だけ、つまりTrackがAlbumに所有されていて外部への参照はできないということである。

バックポインタの追加は、TrackからAlbumへのリンクを配置することであり、効果的に関係に双方向性を持たせられる。これはオブジェクトモデルを変更するが、もう一方で、単一値フィールドに対してもシンプルに更新を行うことができる。

いずれのオプションも効果的と思われない場合は差分表示を実行するが、ここでは次の2つの方法を検討する。データベースの現在の状況による差分表示と、最初に読み取った内容による差分表示である。データベースの差分表示では、データベースからのコレクションの再読み込み、およびその後読み込んだコレクションとAlbum中のコレクションとの比較を行う。データベース上のAlbumにないすべての要素は完全に削除される。一方、Album内のディスク上にないすべての要素は、確実に追加する項目である。その後アプリケーションのロジックに移りそれぞれに項目の処理を決定する。

最初に読み込んだ内容を差分表示するためには、読み込んだ内容を保持するが、この方法には、新たなデータベースの読み込みを回避できるというメリットがある。また、**軽オフラインロック**を使用しているデータベースには、差分表示を行わなければならない場合がある。

通常、コレクションに追加されるすべての要素は、新しいオブジェクトであるかどうか最初の時点でチェックする必要がある。チェックを行うには、それがキーを持っているかを確認し、持っていない場合はデータベースに追加する。このステップは**ユニットオブワーク**によって大幅に簡素化できるのである。理由は、この方法では新しいオブジェクトが、自動的に最初に挿入されるからである。

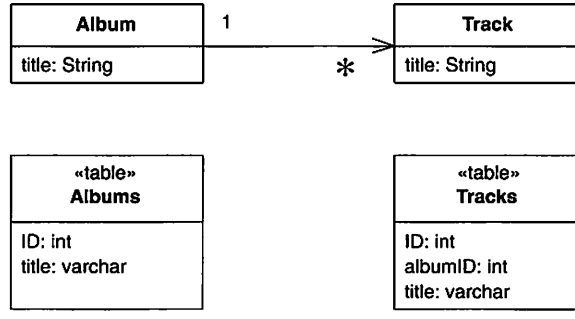


図12.3——複数値参照のクラスおよびテーブル

いずれの場合も、次にデータベース中のリンクされた行を検出し、外部キーが現行のAlbumを指し示すように更新を行う。

削除する場合、Track が他の Album に移動されたのか、Track の Album があるか、また Track が一緒に削除されたのかどうかを把握しておく必要がある。他の Album に移動されている場合、移動先の Album の更新と同時に更新されなければならない。Album がない場合、外部キーを null（設定なし）にする。また Track が削除されている場合、何らかの削除を実行する時点で外部キーも削除すべきである。バックリンクが強制的に行われる場合、削除の処理は極めて容易であり、ここに示すように、すべての Track が Album 上にあるようになる。これにより、コレクションから何が削除された項目かを考える必要がなくなる。つまり、追加先の Album を処理する時点で更新されるからである。

リンクが不変で Album の Track が変更できない場合、追加は必ず挿入を、消去は必ず削除を意味する。すべてがさらにシンプルになる。

ここで注意すべきは、リンクの循環である。たとえば、Order を読み込む必要があるとする。Order は、（読み込みの対象である）Customer（顧客）へのリンクを持っている。Customer は、1 セットの（読み込みの対象である）Payment（支払い）データを持ち、各 Payment データは、支払い対象となる Order データを持っている。この場合、読み込もうとしているオリジナルの Order データが含まれていることもある。したがって、再び Order を読み込むことになる（こうして段落の先頭へと再び戻ることになる）。

このような循環を避けるために、オブジェクトの作成方法を要約する 2 つのオプションを使用する。勧めるのは、完全に形成されたオブジェクトを提供するデータを作成メソッドに含める方法である。この場合、循環を断ち切るポイントに**レイジーロード**を配置する。ポイントを間違った場合、スタックオーバーフローが発生するが、十分なテストが行われている場合にはこの負荷を管理することもできる。

もう 1 つの選択肢は、空のオブジェクトを作成し即座に**一意マッピング**に挿入する方法である。この場合、再び循環元に戻った時点で、オブジェクトはすでに読み込まれているため、その循環は終了する。作成するオブジェクトはこの時点では完全に形成されてはいないが、ロードの手順が完了するまでには完成されるのである。正しい読み込みを行うためだけに**レイジーロード**を使用するかどうかという決定を行う必要がなくなる。

12.2.2 | 使用するタイミング

外部キーマッピングは、クラス間のすべての関係に対して使用できる。使用できないケースで最も一般的なのは、多対多の関係である。外部キーは単一値であり、標準形式では単一フィールドに複数の外部キーを格納できないのである。その代わり**関連テーブルマッピング**を使用する。

バックポインタのないコレクションフィールドを持つ場合、多くの側面から**依存マッピング**を使用するかどうかを考慮すべきである。この場合、コレクションの処理を簡略化できる。

関連オブジェクトがバリューオブジェクトの場合は、**組込みバリュー**を使用する。

12.2.3 ｜例：単一値参照（Java）

これは最もシンプルなケースであり、Album は Artist に対する単一参照を持つ。

```
class Artist...

   private String name;
   public Artist(Long ID, String name) {
      super(ID);
      this.name = name;
   }
   public String getName() {
      return name;
   }
   public void setName(String name) {
      this.name = name;
   }

class Album...

   private String title;
   private Artist artist;
   public Album(Long ID, String title, Artist artist) {
      super(ID);
      this.title = title;
      this.artist = artist;
   }
   public String getTitle() {
      return title;
   }
   public void setTitle(String title) {
      this.title = title;
   }
   public Artist getArtist() {
      return artist;
   }
   public void setArtist(Artist artist) {
      this.artist = artist;
   }
```

図 12.4 に、Album の読み込み方法を示す。AlbumMapper は、特定の Album を読み込

む指示が出されると、データベースにクエリーを発行し、結果群を引き出す。次に結果群の各外部キーフィールドに対するクエリーを実行し、オブジェクトを検索する。これで検索された適切なオブジェクトを持つ Album を完成できるようになる。Artist オブジェクトがすでにメモリ上にある場合、キャッシュからフェッチされる。メモリ上にない場合は、同様の方法でデータベースから読み込まれる。

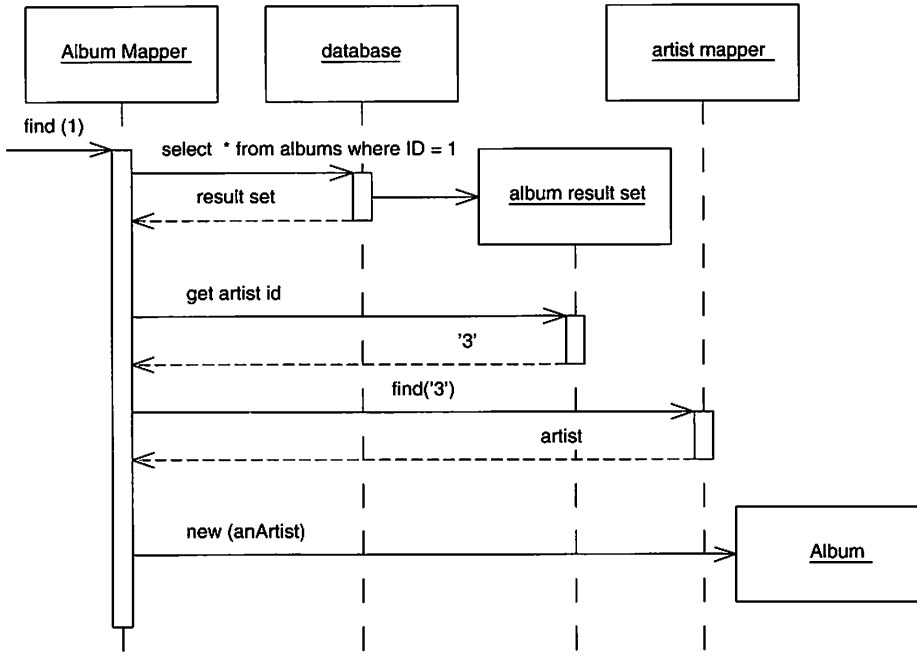


図 12.4 ——単一値フィールドの読み込みシーケンス

この検索操作は、抽象的な振る舞いを使用して**一意マッピング**を操作する。

```
class AlbumMapper...

   public Album find(Long id) {
      return (Album) abstractFind(id);
   }
   protected String findStatement() {
      return "SELECT ID, title, artistID FROM albums WHERE ID = ?";
   }

class AbstractMapper...
```

```
    abstract protected String findStatement();
    protected DomainObject abstractFind(Long id) {
        DomainObject result = (DomainObject) loadedMap.get(id);
        if (result != null) return result;
        PreparedStatement stmt = null;
        ResultSet rs = null;
        try {
            stmt = DB.prepare(findStatement());
            stmt.setLong(1, id.longValue());
            rs = stmt.executeQuery();
            rs.next();
            result = load(rs);
            return result;
        } catch (SQLException e) {
            throw new ApplicationException(e);
        } finally {cleanUp(stmt, rs); }
    }
    private Map loadedMap = new HashMap();
```

この検索操作は読み込み操作を呼び出し、Albumにデータを読み込む。

```
class AbstractMapper...

    protected DomainObject load(ResultSet rs) throws SQLException {
        Long id = new Long(rs.getLong(1));
        if (loadedMap.containsKey(id)) return (DomainObject)
            loadedMap.get(id);
        DomainObject result = doLoad(id, rs);
        doRegister(id, result);
        return result;
    }
    protected void doRegister(Long id, DomainObject result) {
        Assert.isFalse(loadedMap.containsKey(id));
        loadedMap.put(id, result);
    }
    abstract protected DomainObject doLoad(Long id, ResultSet rs)
        throws SQLException;

class AlbumMapper...

    protected DomainObject doLoad(Long id, ResultSet rs) throws SQLException {
        String title = rs.getString(2);
        long artistID = rs.getLong(3);
```

```
        Artist artist = MapperRegistry.artist().find(artistID);
        Album result = new Album(id, title, artist);
        return result;
    }
```

Album を更新する場合、外部キーの値はリンクされている Artist オブジェクトから抽出される。

```
class AbstractMapper...

    abstract public void update(DomainObject arg);

class AlbumMapper...

    public void update(DomainObject arg) {
        PreparedStatement statement = null;
        try {
            statement = DB.prepare(
                "UPDATE albums SET title = ?, artistID = ? WHERE id = ?");
            statement.setLong(3, arg.getID().longValue());
            Album album = (Album) arg;
            statement.setString(1, album.getTitle());
            statement.setLong(2, album.getArtist().getID().longValue());
            statement.execute();
        } catch (SQLException e) {
            throw new ApplicationException(e);
        } finally {
            cleanUp(statement);
        }
    }
```

12.2.4 ｜例：複数テーブルの検索（Java）

1 つのテーブルに対して 1 つのクエリーを発行することは概念的にわかりやすいが、SQL はリモートコールから構成されていて、その上リモートコールの速度が遅いため効率的ではない。したがって、1 つのクエリーによって複数のテーブルからデータを収集する方法は、検討する価値が十分ある。上記の例を修正することで、1 つのクエリーを発行し、1 つの SQL 呼び出しによって Album 情報と Artist 情報の両方を取得することができるようになる。最初の変更箇所は、SQL の find 文の部分である。

```
class AlbumMapper...

   public Album find(Long id) {
      return (Album) abstractFind(id);
   }
   protected String findStatement() {
      return "SELECT a.ID, a.title, a.artistID, r.name " +
            " FROM albums a, artists r " +
            " WHERE ID = ? AND a.artistID = r.ID";
   }
```

次に、Album 情報と Artist 情報の両方を読み込む別の load メソッドを使用する。

```
class AlbumMapper...

   protected DomainObject doLoad(Long id, ResultSet rs) throws SQLException {
      String title = rs.getString(2);
      long artistID = rs.getLong(3);
      ArtistMapper artistMapper = MapperRegistry.artist();
      Artist artist;
      if (artistMapper.isLoaded(artistID))
         artist = artistMapper.find(artistID);
      else
         artist = loadArtist(artistID, rs);
      Album result = new Album(id, title, artist);
      return result;
   }
   private Artist loadArtist(long id, ResultSet rs) throws SQLException {
      String name = rs.getString(4);
      Artist result = new Artist(new Long(id), name);
      MapperRegistry.artist().register(result.getID(), result);
      return result;
   }
```

SQL の結果を Artist オブジェクトにマッピングするメソッドを配置することができる。一方で、Artist を読み込むクラスであるため、Artist のマッパーに配置する方法も有効な手段である。さらに load メソッドは SQL と緊密に結合しているため SQL クエリーとともに配置される。ここでは後者を採用した。

12.2.5 | 例：参照のコレクション（C#）

参照のコレクションのケースは、コレクションを構成するフィールドがある場合に発生する。Team と Player を例に挙げて、Player を**依存マッピング**することはできない場合を仮定する（図 12.5）。

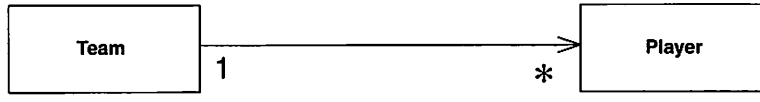


図 12.5 —— 複数の Player がいる Team

```
class Team...

   public String Name;
   public IList Players {
      get {return ArrayList.ReadOnly(playersData);}
      set {playersData = new ArrayList(value);}
   }
   public void AddPlayer(Player arg) {
      playersData.Add(arg);
   }
   private IList playersData = new ArrayList();
```

データベースにおいて、上記の例は Team に対する外部キーを持つ Player レコードによって処理される（図 12.6）。

図 12.6 —— 複数の Player がいる Team のデータベース構造

```
class TeamMapper...
      public Team Find(long id) {
         return (Team) AbstractFind(id);
      }

class AbstractMapper...
```

```
        protected DomainObject AbstractFind(long id) {
            Assert.True (id != DomainObject.PLACEHOLDER_ID);
            DataRow row = FindRow(id);
            return (row == null) ? null : Load(row);
        }
        protected DataRow FindRow(long id) {
            String filter = String.Format("id = {0}", id);
            DataRow[] results = table.Select(filter);
            return (results.Length == 0) ? null : results[0];
        }
        protected DataTable table {
            get {return dsh.Data.Tables[TableName];}
        }
        public DataSetHolder dsh;
        abstract protected String TableName {get;}

class TeamMapper...

        protected override String TableName {
            get {return "Teams";}
        }
```

データセットホルダーは使用中のデータセットを保持するクラスであり、データベースに更新するアダプターも持っている。

```
class DataSetHolder...

        public DataSet Data = new DataSet();
        private Hashtable DataAdapters = new Hashtable();
```

この例では、データセットホルダーがいくつかのクエリーによって配置されていると仮定する。find メソッドは、load メソッドを呼び出し新しいオブジェクトにデータを読み込む。

```
class AbstractMapper...
        protected DomainObject Load (DataRow row) {
            long id = (int) row ["id"];
            if (identityMap[id] != null) return (DomainObject)
                identityMap[id];
            else {
                DomainObject result = CreateDomainObject();
                result.Id = id;
```

```
            identityMap.Add(result.Id, result);
            doLoad(result,row);
            return result;
        }
    }
    abstract protected DomainObject CreateDomainObject();
    private IDictionary identityMap = new Hashtable();
    abstract protected void doLoad (DomainObject obj, DataRow row);

class TeamMapper...

    protected override void doLoad (DomainObject obj, DataRow row) {
        Team team = (Team) obj;
        team.Name = (String) row["name"];
        team.Players = MapperRegistry.Player.FindForTeam(team.Id);
    }
```

Player を取得するためには、Player マッパーで特定の find コードを実行する必要がある。

```
class PlayerMapper...

    public IList FindForTeam(long id) {
        String filter = String.Format("teamID = {0}", id);
        DataRow[] rows = table.Select(filter);
        IList result = new ArrayList();
        foreach (DataRow row in rows) {
            result.Add(Load (row));
        }
        return result;
    }
```

更新する場合、Team は自分自身のデータを保存し、Player テーブルへのデータ保存を Player マッパーに委譲する。

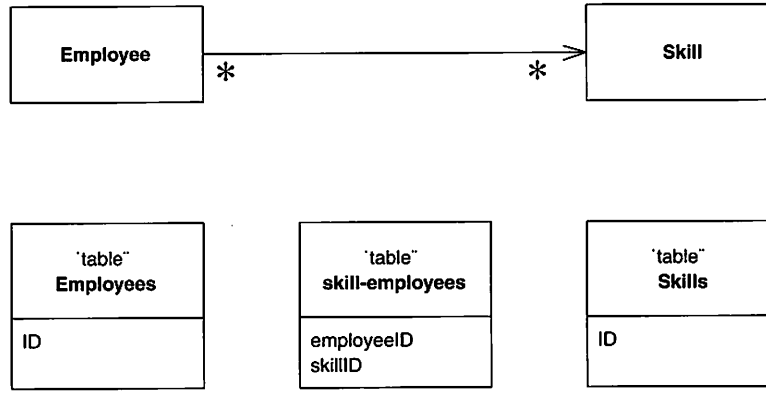
```
class AbstractMapper...

    public virtual void Update (DomainObject arg) {
        Save (arg, FindRow(arg.Id));
    }
    abstract protected void Save (DomainObject arg, DataRow row);

class TeamMapper...
```

```
        protected override void Save (DomainObject obj, DataRow row){
            Team team = (Team) obj;
            row["name"] = team.Name;
            savePlayers(team);
        }
        private void savePlayers(Team team){
            foreach (Player p in team.Players) {
                MapperRegistry.Player.LinkTeam(p, team.Id);
            }
        }

class PlayerMapper...

        public void LinkTeam (Player player, long teamID) {
            DataRow row = FindRow(player.Id);
            row["teamID"] = teamID;
        }
```

Player から Team への関係が必須となっているため、更新のためのコードはよりシンプルになっている。Team 間で Player を移動する場合、両方の Team を更新するだけでよく、複雑な差分を表示して Player を整理する必要はない。この例を、読み込みコードを自習する際の教材にしてほしい。そのため、解答は示さないことにする。

12.3 関連テーブルマッピング

リンクされたテーブルへの外部キーを持つテーブルとして、関連を保存する。



オブジェクトは、コレクションをフィールド値として使用することで、複数値フィールド

を容易に扱うことができるようになる。リレーショナルデータベースはこのような機能を持っていないため、単一値フィールドに制限されている。1対多の関係をマッピングする場合、**外部キーマッピング**を使用してこの処理を実現できる。つまり基本的に関連の単一値端に対して外部キーを使用するのである。しかし、外部キーを保持するための単一値端がないため、多対多の関係ではこのような処理を行うことができない。

この状況に対する解決策は、リレーショナルデータを好む人が長年使用してきた従来の方法、つまり、関連を記録するための特別なテーブルを作成することである。次に、**関連テーブルマッピング**を使って、複数値フィールドをリンクテーブルに追加する。

12.3.1 | 動作方法

関連テーブルマッピングの基本的な考えは、関連の格納にリンクテーブルを使用することである。このテーブルは、互いにリンクされる2つのテーブルの外部キーIDだけを持ち、それぞれペアとなっている関連オブジェクトごとに1行を持つ。

リンクテーブルには、メモリ上に対応するオブジェクトはないため、リンクテーブルがIDを持つことはなく、またプライマリキーは関連付けられるテーブルの2つのプライマリキーを組み合わせたものである。

一言で言うと、リンクテーブルからデータを読み出すには、2つのクエリーを発行する。ここではEmployee（従業員）のSkill（スキル）を読み込む場合を考えてみよう。この場合、概念上は少なくとも2段階でクエリーを発行する。最初に`SkillsEmployees`テーブルに対するクエリーを発行し、求めるEmployeeとリンクしているすべての行を検索する。次にリンクテーブルの各行の関連IDに対応するSkillオブジェクトを見つけ出す。

情報がメモリ上にある場合、スキームは問題なく機能する。情報がメモリにない場合は、リンクテーブル内にある各Skillに1つのクエリーを発行するためクエリーにコストがかかる。それを回避するには、Skillテーブルとリンクテーブルとのジョインを行う。結果、マッピングは多少複雑にはなるものの、1つのクエリーによってすべてのデータを取得することができる。

リンクデータの更新には、多値フィールドの更新と同じような操作が必要だが、幸い**依存マッピング**でリンクテーブルを扱うことができるため、それほど面倒ではない。リンクテーブルを参照するテーブルは他にはないため、必要に応じて自由にリンクの作成および削除を行うことができる。

12.3.2 | 使用するタイミング

関連テーブルマッピングの標準的なケースは多対多の関連である。その理由は、他の現実

的な代替案がないためである。

関連テーブルマッピングは、その他の関連の形式に対しても使用できる。しかし、**外部キーマッピング**よりも複雑でジョインが追加されるため、あまり勧められない。たとえいくつかの状況で**関連テーブルマッピング**が関連を単純にするのに適していても、スキーマを制御できないデータベースが関連してくる場合がある。2つの既存テーブルのリンクが必要なときに、テーブルに対して列を追加することができない場合もある。その場合は、新しいテーブルを作成して**関連テーブルマッピング**を使用する。必要でないときであっても、既存のスキーマが関連テーブルを使用する場合もある。その場合は、データベーススキーマを簡素化するより**関連テーブルマッピング**を使用した方が簡単な場合が多い。

リレーショナルデータベースの設計において、関係についての情報を含む関連テーブルを持つことがある。たとえば、ある人物の会社における雇用に関する情報を格納する人/会社（Person/Company）関連テーブルなどがその例である。この場合、人/会社(Person/Company）関係テーブルは、ドメインオブジェクトに対応する。

12.3.3 ｜例：Employee と Skill（C#）

これはスケッチのモデルを使用したシンプルな例である。ここでは、Skill のコレクションを持った Employee クラスを挙げて、コレクションに含まれる各 Skill を複数の Employee に対応させている。

```
class Employee...

      public IList Skills {
         get {return ArrayList.ReadOnly(skillsData);}
         set {skillsData = new ArrayList(value);}
      }
      public void AddSkill (Skill arg) {
         skillsData.Add(arg);
      }
      public void RemoveSkill (Skill arg) {
         skillsData.Remove(arg);
      }
      private IList skillsData = new ArrayList();
```

データベースから Employee を読み込むには、EmployeeMapper を使って Skill を抽出する。それぞれの EmployeeMapper クラスは、Employee オブジェクトを作成する Find メソッドを持っている。すべてのマッパーは、共通サービスを抽出する AbstractMapper クラスのサブクラスとなる。

```
class EmployeeMapper...

      public Employee Find(long id) {
         return (Employee) AbstractFind(id);
      }

class AbstractMapper...

      protected DomainObject AbstractFind(long id) {
         Assert.True (id != DomainObject.PLACEHOLDER_ID);
         DataRow row = FindRow(id);
         return (row == null) ? null : Load(row);
      }
      protected DataRow FindRow(long id) {
         String filter = String.Format("id = {0}", id);
         DataRow[] results = table.Select(filter);
         return (results.Length == 0) ? null : results[0];
      }
      protected DataTable table {
         get {return dsh.Data.Tables[TableName];}
      }
      public DataSetHolder dsh;
      abstract protected String TableName {get;}

class EmployeeMapper...

      protected override String TableName {
         get {return "Employees";}
      }
```

データセットホルダーは、ADO.NET データセットとデータベースに保存するための関連アダプターを含むシンプルなオブジェクトである。

```
class DataSetHolder...

      public DataSet Data = new DataSet();
      private Hashtable DataAdapters = new Hashtable();
```

この例をわかりやすくするため、ここではデータセットは必要なデータとともにすでに読み込まれていることを前提としている。

Find メソッドは Load メソッドを呼び出して、Employee のデータを読み込む。

```
class AbstractMapper...

      protected DomainObject Load (DataRow row) {
         long id = (int) row ["id"];
         if (identityMap[id] != null) return (DomainObject) identityMap[id];
         else {
            DomainObject result = CreateDomainObject();
            result.Id = id;
            identityMap.Add(result.Id, result);
            doLoad(result,row);
            return result;
         }
      }
      abstract protected DomainObject CreateDomainObject();
      private IDictionary identityMap = new Hashtable();
      abstract protected void doLoad (DomainObject obj, DataRow row);

class EmployeeMapper...

      protected override void doLoad (DomainObject obj, DataRow row) {
         Employee emp = (Employee) obj;
         emp.Name = (String) row["name"];
         loadSkills(emp);
      }
```

Skill（スキル）の読み込みは分離したメソッドで実行するためとてもぎこちない。

```
class EmployeeMapper...

      private IList loadSkills (Employee emp) {
         DataRow[] rows = skillLinkRows(emp);
         IList result = new ArrayList();
         foreach (DataRow row in rows) {
            long skillID = (int)row["skillID"];
            emp.AddSkill(MapperRegistry.Skill.Find(skillID));
         }
         return result;
      }
         private DataRow[] skillLinkRows(Employee emp) {
            String filter = String.Format("employeeID = {0}", emp.Id);
            return skillLinkTable.Select(filter);
      }
```

```
    private DataTable skillLinkTable {
        get {return dsh.Data.Tables["skillEmployees"];}
    }
```

Skill 情報の変更は、AbstractMapper 上で Update メソッドを用いて行う必要がある。

```
class AbstractMapper...

    public virtual void Update (DomainObject arg) {
        Save (arg, FindRow(arg.Id));
    }
    abstract protected void Save (DomainObject arg, DataRow row);
```

Update メソッドは、サブクラスで Save メソッドを呼び出す。

```
class EmployeeMapper...

    protected override void Save (DomainObject obj, DataRow row) {
        Employee emp = (Employee) obj;
        row["name"] = emp.Name;
        saveSkills(emp);
    }
```

ここでも私は、Skill の保存用に、分離したメソッドを作成している。

```
class EmployeeMapper...

    private void saveSkills(Employee emp) {
        deleteSkills(emp);
        foreach (Skill s in emp.Skills) {
            DataRow row = skillLinkTable.NewRow();
            row["employeeID"] = emp.Id;
            row["skillID"] = s.Id;
            skillLinkTable.Rows.Add(row);
        }
    }
    private void deleteSkills(Employee emp) {
        DataRow[] skillRows = skillLinkRows(emp);
        foreach (DataRow r in skillRows) r.Delete();
    }
```

ここに示すロジックは、既存のリンクテーブル行をすべて削除し新しい行を作成するというシンプルなものである。これによって、どの行が追加されどの行が削除されたかを確認する手間を省略できる。

12.3.4 ｜例：ダイレクトSQLの使用（Java）

私がADO.NETが素晴らしいと考える点の1つは、クエリーの最小化についての詳細内容に触れることなくオブジェクトリレーショナルマッピングの基本についてわかりやすく解説できるという点である。他のリレーショナルマッピングスキームの場合、SQLに類似しているためクエリーの最小化も考えなければならない。

データベースに直接アクセスする場合、クエリーの最小化は重要である。最初のバージョンでは2つのクエリーを発行して、EmployeeとSkillを抽出する。この方法はわかりやすいが最適ではないので、もう少し加えたい。

以下にテーブルのDDLを示す。

```
create table employees (ID int primary key, firstname varchar,
   lastname varchar)
create table skills (ID int primary key, name varchar)
create table employeeSkills (employeeID int, skillID int,
   primary key (employeeID, skillID))
```

1人のEmployeeを読み込む場合も、私は前述のような手法を使用する。EmployeeMapperは、**レイヤスーパータイプ**の抽象的なfindメソッドに対しシンプルなラッパーを定義する。

```
class EmployeeMapper...

   public Employee find(long key) {
      return find (new Long (key));
   }
   public Employee find (Long key) {
      return (Employee) abstractFind(key);
   }
   protected String findStatement() {
      return
         "SELECT " + COLUMN_LIST +
         " FROM employees" +
         " WHERE ID = ?";
      }
      public static final String COLUMN_LIST = " ID, lastname, firstname ";
```

```
class AbstractMapper...

   protected DomainObject abstractFind(Long id) {
      DomainObject result = (DomainObject) loadedMap.get(id);
      if (result != null) return result;
      PreparedStatement stmt = null;
      ResultSet rs = null;
      try {
         stmt = DB.prepare(findStatement());
         stmt.setLong(1, id.longValue());
         rs = stmt.executeQuery();
         rs.next();
         result = load(rs);
         return result;
      } catch (SQLException e) {
         throw new ApplicationException(e);
      } finally {DB.cleanUp(stmt, rs);
      }
   }
   abstract protected String findStatement();
   protected Map loadedMap = new HashMap();
```

次に、find メソッドは load メソッドを呼び出す。Employee のデータが Employee Mapper に読み込まれる一方、抽象的な load メソッドは ID 読み込みを処理する。

```
class AbstractMapper...

   protected DomainObject load(ResultSet rs) throws SQLException {
      Long id = new Long(rs.getLong(1));
      return load(id, rs);
   }
   public DomainObject load(Long id, ResultSet rs) throws SQLException {
      if (hasLoaded(id)) return (DomainObject) loadedMap.get(id);
      DomainObject result = doLoad(id, rs);
      loadedMap.put(id, result);
      return result;
   }
   abstract protected DomainObject doLoad(Long id, ResultSet rs)
      throws SQLException;

class EmployeeMapper...

   protected DomainObject doLoad(Long id, ResultSet rs) throws SQLException {
```

```
        Employee result = new Employee(id);
        result.setFirstName(rs.getString("firstname"));
        result.setLastName(rs.getString("lastname"));
        result.setSkills(loadSkills(id));
        return result;
    }
```

Employeeは、Skillを読み込むために別のクエリーを発行する必要があるが、ここでは1つのクエリーですべてのSkillを簡単に読み込むことができる。SkillMapperを呼び出し特定のSkillのデータを読み込む。

```
class EmployeeMapper...

    protected List loadSkills(Long employeeID) {
          PreparedStatement stmt = null;
          ResultSet rs = null;
          try {
             List result = new ArrayList();
             stmt = DB.prepare(findSkillsStatement);
             stmt.setObject(1, employeeID);
             rs = stmt.executeQuery();
             while (rs.next()) {
                Long skillId = new Long (rs.getLong(1));
                result.add((Skill) MapperRegistry.skill().loadRow
                   (skillId, rs));
             }
             return result;
          } catch (SQLException e) {
             throw new ApplicationException(e);
          } finally {DB.cleanUp(stmt, rs);
          }
    }
    private static final String findSkillsStatement =
       "SELECT skill.ID, " + SkillMapper.COLUMN_LIST +
       " FROM skills skill, employeeSkills es " +
       " WHERE es.employeeID = ? AND skill.ID = es.skillID";

class SkillMapper...

    public static final String COLUMN_LIST = " skill.name skillName ";

class AbstractMapper...
```

```
    protected DomainObject loadRow (Long id, ResultSet rs) throws SQLException {
        return load (id, rs);
    }

class SkillMapper...

    protected DomainObject doLoad(Long id, ResultSet rs) throws SQLException {
        Skill result = new Skill (id);
        result.setName(rs.getString("skillName"));
        return result;
    }
```

AbstractMapper もまた、Employee の検索に役立つ。

```
class EmployeeMapper...

    public List findAll() {
        return findAll(findAllStatement);
    }
    private static final String findAllStatement =
            "SELECT " + COLUMN_LIST +
            " FROM employees employee" +
            " ORDER BY employee.lastname";

class AbstractMapper...

    protected List findAll(String sql) {
        PreparedStatement stmt = null;
        ResultSet rs = null;
        try {
            List result = new ArrayList();
            stmt = DB.prepare(sql);
            rs = stmt.executeQuery();
            while (rs.next())
                result.add(load(rs));
            return result;
        } catch (SQLException e) {
            throw new ApplicationException(e);
        } finally {DB.cleanUp(stmt, rs);
        }
    }
```

これらの機能は、とてもシンプルでわかりやすい。しかし、クエリーの数についての問題がある。各 Employee には読み込みのために、2 つのクエリーが必要になるという問題である。1 つのクエリーで多数の Employee に対する基本的な Employee データを読み込むことはできるが、加えて Skill の読み込みには Employee1 人あたり 1 つのクエリーが必要になる。そのため、100 人分の Employee を読み込むためには 101 個のクエリーが必要となるのである。

12.3.5 ｜例：複数の Employee に対する 1 つのクエリーの使用（Java）

1 つのクエリーで、多数の Employee をその Skill とともに呼び戻すことができる。この例は、より複雑な複数テーブルクエリーの最適化の優れた例である。毎回ではなく必要な場合にだけ、この方法を使うことも優れている理由である。重要性の低いクエリーよりも、速度の遅いクエリーの高速化に多くの労力を注ぐことを勧める。

最初にあげるケースは、基本データを保持する同一のクエリーで 1 人の Employee の Skill を抽出するというシンプルなものである。3 つのテーブルにまたがってジョインを行う SQL 文を使用する。

```
class EmployeeMapper...

   protected String findStatement() {
      return
         "SELECT " + COLUMN_LIST +
         " FROM employees employee, skills skill, employeeSkills es" +
         " WHERE employee.ID = es.employeeID AND skill.ID =
            es.skillID AND employee.ID = ?";
   }
   public static final String COLUMN_LIST =
         " employee.ID, employee.lastname, employee.firstname, " +
         " es.skillID, es.employeeID, skill.ID skillID, " +
            SkillMapper.COLUMN_LIST;
```

スーパークラスの abstractFind メソッドと load メソッドは、前述した例と同じであるためここでは省略する。EmployeeMapper は、複数のデータ行を使用するため別にデータを読み込む。

```
class EmployeeMapper...

   protected DomainObject doLoad(Long id, ResultSet rs) throws SQLException {
      Employee result = (Employee) loadRow(id, rs);
```

```
        loadSkillData(result, rs);
        while (rs.next()){
            Assert.isTrue(rowIsForSameEmployee(id, rs));
            loadSkillData(result, rs);
        }
        return result;
    }
    protected DomainObject loadRow(Long id, ResultSet rs) throws SQLException {
        Employee result = new Employee(id);
        result.setFirstName(rs.getString("firstname"));
        result.setLastName(rs.getString("lastname"));
        return result;
    }
    private boolean rowIsForSameEmployee(Long id, ResultSet rs)
        throws SQLException {
        return id.equals(new Long(rs.getLong(1)));
    }
    private void loadSkillData(Employee person, ResultSet rs)
        throws SQLException {
        Long skillID = new Long(rs.getLong("skillID"));
        person.addSkill ((Skill)MapperRegistry.skill().loadRow(skillID, rs));
    }
```

この場合、EmployeeMapper の load メソッドが、残りの結果群を調べてデータを読み込む。

1 人の Employee のデータを読み込む場合は、シンプルである。しかし、この複数テーブルクエリーの有効性は、多数の Employee を読み込むときに発揮される。特にその結果群を Employee でグループ化したくない場合には、読み込み権の取得が難しくなる。ここでは、関連テーブル自体に焦点をあてた結果群を通して、同時に Employee と Skill を読み込むヘルパークラスを導入している。

SQL と特定の Loader クラスへの呼び出しから始める。

```
class EmployeeMapper...

    public List findAll() {
        return findAll(findAllStatement);
    }
    private static final String findAllStatement =
            "SELECT " + COLUMN_LIST +
            " FROM employees employee, skills skill, employeeSkills es" +
            " WHERE employee.ID = es.employeeID AND skill.ID = es.skillID" +
```

```
            " ORDER BY employee.lastname";
    protected List findAll(String sql) {
        AssociationTableLoader loader = new AssociationTableLoader
            (this, new SkillAdder());
        return loader.run(findAllStatement);
    }

class AssociationTableLoader...

    private AbstractMapper sourceMapper;
    private Adder targetAdder;
    public AssociationTableLoader(AbstractMapper primaryMapper,
        Adder targetAdder) {
        this.sourceMapper = primaryMapper;
        this.targetAdder = targetAdder;
    }
```

skillAdder について、ここでは気にする必要はない。後の解説で少しずつ明らかになってくる。現時点では、マッパーへの参照を持つ Loader クラスを構築し、次にクエリーによって読み込みを実行するように指示している点に少しだけ注意してほしい。これがメソッドオブジェクトの標準的な構造である。**メソッドオブジェクト**[Beck Patterns]とは、複雑なメソッドを独自のオブジェクトに変換する方法である。この方法の最大のメリットは、パラメータを介して値を受け渡す代わりに、値をフィールドに渡せる点である。メソッドオブジェクトを使用する一般的な方法では、まずメソッドオブジェクトを作成し、実行して、役割が終わった後は削除する。

読み込みの振る舞いは、3段階で行われる。

```
class AssociationTableLoader...

    protected List run(String sql) {
        loadData(sql);
        addAllNewObjectsToIdentityMap();
        return formResult();
    }
```

loadData メソッドは、SQL 呼び出しを形成し、実行して、結果群をループさせる。これはメソッドオブジェクトであり、結果群はフィールドに配置されているため受け渡す必要はない。

```
class AssociationTableLoader...

   private ResultSet rs = null;
   private void loadData(String sql) {
      PreparedStatement stmt = null;
      try {
         stmt = DB.prepare(sql);
         rs = stmt.executeQuery();
         while (rs.next())
            loadRow();
      } catch (SQLException e) {
         throw new ApplicationException(e);
      } finally {DB.cleanUp(stmt, rs);
      }
   }
```

loadRow メソッドは、結果群の 1 つの行からデータを読み込む。この部分は少し複雑である。

```
class AssociationTableLoader...

   private List resultIds = new ArrayList();
   private Map inProgress = new HashMap();
   private void loadRow() throws SQLException {
      Long ID = new Long(rs.getLong(1));
      if (!resultIds.contains(ID)) resultIds.add(ID);
      if (!sourceMapper.hasLoaded(ID)) {
         if (!inProgress.keySet().contains(ID))
         inProgress.put(ID, sourceMapper.loadRow(ID, rs));
         targetAdder.add((DomainObject) inProgress.get(ID), rs);
      }
   }

class AbstractMapper...

   boolean hasLoaded(Long id) {
      return loadedMap.containsKey(id);
   }
```

Loader クラスは、結果群のすべての順序を保つため、Employee の出力リストは最初の表示と同じ順序となる。そのため、私は ID リストを表示されたとおりの順番にしている。ID を取得した後、私はそれが完全にマッパーに読み込まれているかどうかを確認する（通

常は以前のクエリーを使用する)。読み込まれていない場合は、持っているデータを読み込み、それを進行中のリストとして保持する。Employee からすべてのデータを集めるためにいくつかの行が結合されるが、すべての行を連続的にヒットすることはできないため、このようなリストが必要となる。

このコードの最も複雑な部分は、読み込む Skill を Employee の Skill リストに追加できるようにしつつ、Employee と Skill に依存しないように Loader クラスの汎用性を保証することである。これを実現するため、私は仕掛けを検討して、内部インタフェース (Adder) を発見した。

```
class AssociationTableLoader...

   public static interface Adder {
      void add(DomainObject host, ResultSet rs) throws SQLException ;
   }
```

オリジナルの呼び出し側は、Employee および Skill の特定のニーズに、Loader クラスをバインドするインタフェースを実装する必要がある。

```
class EmployeeMapper...
   private static class SkillAdder implements AssociationTableLoader.Adder {
      public void add(DomainObject host, ResultSet rs) throws SQLException {
         Employee emp = (Employee) host;
         Long skillId = new Long (rs.getLong("skillId"));
         emp.addSkill((Skill) MapperRegistry.skill().loadRow(skillId, rs));
      }
   }
```

インタフェースには、関数ポインタまたはクロージャを持った言語が最初から備わっているので、クラスおよびインタフェースでは、少なくともこのジョブを実行させることはできる (この場合、内部的である必要はないが、狭いスコープを提供することに役立つ)。

読者の予想どおり、私はスーパークラス上で load メソッドと loadRow メソッドを定義し、load メソッドを呼び出すため loadRow を実装している。この方法を使用した理由は、読み込み動作によって結果群が転送されないようにしたいと考えたからである。load メソッドは、オブジェクトを読み込むために必要なことはすべて行うが、loadRow メソッドは、カーソルの位置を変更することなく行からデータを読み込むことができる。多くの場合これら 2 つのメソッドは同じ動作をするが、Employee Mapper の場合、2 つのメソッドは異なった動作を行う。

現時点では、すべてのデータは結果群の形式となっている。私の手元には 2 つのコレクションがある。1 つは、結果群の中にあった Employee ID が、最初に表示されたときのと

おりの順序で並んでいるリストであり、もう1つはEmployee Mapperの**一意マッピング**でまだ表示されていない新しいオブジェクトのリストである。

次のステップは、新規オブジェクトを**一意マッピング**に配置する方法である。

```
class AssociationTableLoader...

   private void addAllNewObjectsToIdentityMap() {
      for (Iterator it = inProgress.values().iterator();
         it.hasNext();) sourceMapper.putAsLoaded
         ((DomainObject)it.next());
   }

class AbstractMapper...

   void putAsLoaded (DomainObject obj) {
      loadedMap.put (obj.getID(), obj);
   }
```

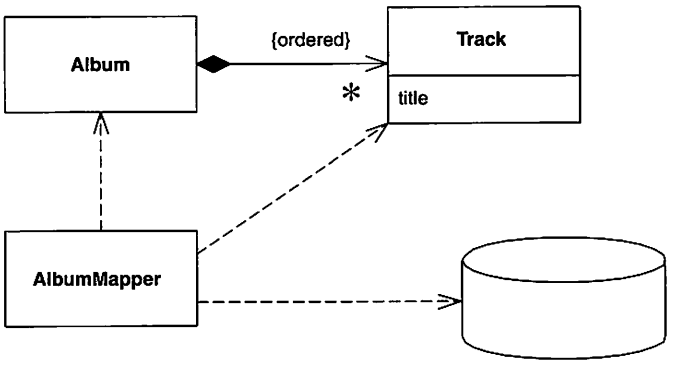
最後のステップは、マッパーからIDを参照し、結果リストを構築することである。

```
class AssociationTableLoader...

   private List formResult() {
      List result = new ArrayList();
      for (Iterator it = resultIds.iterator(); it.hasNext();) {
         Long id = (Long)it.next();
         result.add(sourceMapper.lookUp(id));
      }
      return result;
   }

class AbstractMapper...

   protected DomainObject lookUp (Long id) {
      return (DomainObject) loadedMap.get(id);
   }
```

このようなコードは、平均的な読み込みコードよりも複雑だがクエリーの数を削減することに役立つ。複雑なのでデータベースのインタラクションが少ない場合には、使用を控えるべきだ。しかし、**データマッパー**が、ドメインレイヤに複雑さを意識させることなくクエリーを提供するときは適している。

12.4 | 依存マッピング

任意のクラスに対して、子クラスへのデータマッピングを実行させる。



他のオブジェクトとのつながりで表示されるオブジェクトもある。Album（アルバム）のTrack（曲目）は、ベースとなるオブジェクトが読み込まれたり保存されるときには、必ず読み込まれたり保存されたりする。データベースの他のテーブルから参照されない場合、Album MapperにTrackへのマッピングを実行させることで（このマッピングを**依存マッピング**として扱う）、マッピングの手順を簡素化できる。

12.4.1 | 動作方法

依存マッピングの基本的な考え方は、任意のクラス**Dependent**（依存）がデータベースの永続性のために他のクラス**Owner**（所有者）に依存するということである。各Dependentは、Ownerを1つだけ必ず持たなければならない。

これは明らかにマッピングを行うクラスである。**アクティブレコード**と**行データゲートウェイ**の場合、Dependentクラスには一切のデータベースマッピングコードは含まれず、マッピングコードはOwnerに配置されている。**データマッパー**の場合、Dependentに対するマッパーはなく、マッピングコードはOwnerのマッパーに配置されている。**テーブルデータゲートウェイ**では、通常、Dependentはまったく存在せず、Dependentの処理のすべてはOwnerで行われる。

多くの場合、Ownerを読み込むたびにDependentも読み込むことになる。Dependentの読み込み負荷が大きくしかも使用頻度が少ない場合、**レイジーロード**を使用することで、必要になるまでDependentの読み込みを回避できる。

Dependentの重要な特性は、**一意フィールド**を持たないため**一意マッピング**に格納されないという点である。そのためIDを参照するfindメソッドによる読み込みはできない。あ

らゆる検索は、Owner によって行われるため Dependent の find メソッドはない。

Dependent 自身が他の Dependent の Owner となることもある。その場合、最初の Dependent の Owner は、2 番目の Dependent の永続性の責任も負うことになる。Dependent のすべての階層構造は、1 つのプライマリ Owner によって制御できる。

データベースのプライマリキーを、Owner のプライマリキーを含む複合キーにすることは難しいことではない。オブジェクトが同じ Owner を持たない限り、Dependent のテーブルへの外部キーを持つテーブルはないはずである。その結果、Owner または Dependent 以外のメモリ上のオブジェクトが、Dependent を参照することはない。厳密に言うと、参照はデータベースに対して永続的ではないというルールを緩和できるが、永続的ではない参照を持つこと自体が混乱を招く原因でもある。

UML モデルでは、Owner と Dependent 間のつながりを示すには、コンポジションが適切である。

Dependent の書き込みおよび保存は Owner に任されていて、外部参照がないため、削除および挿入することによって Dependent の更新を処理する。そのため、Dependent のコレクションを更新する場合、Owner にリンクしている行を削除してから、Dependent を再び挿入する。これで Owner のコレクションに対し追加または削除されたオブジェクトの分析を省略できる。

Dependent は、多くの点で**バリューオブジェクト**に似ているが、何かを**バリューオブジェクト**（等号記号のオーバーライドなど）にするときに使用する方法を必要としないことが多い。純粋にメモリ上の観点からその違いを考えると、Dependent と**バリューオブジェクト**との間には際立った違いは見あたらない。オブジェクトの依存性は、データベースマッピングの振る舞いだけを表している。

依存マッピングを使用すると、Owner に行った変更の記録が複雑化してしまう。Dependent（依存）に変更を加えた場合、変更の時点で Owner にマーキングする必要が生じ、結果 Owner はその変更をデータベースに書き込むことになる。このような動作をシンプルにするためには、Dependent を不変にすることであり Dependent に行う一切の変更は、消去または新しい Dependent の追加で行うことである。メモリ上のモデルは動作が難しくなるが、データベースマッピングはシンプルになる。理論上、**データマッパー**を使用する場合、メモリ上のマッピングとデータベースマッピングは独立しているはずだが、これに関しては柔軟性を持った対応も必要な場合がある。

12.4.2 | 使用するタイミング

依存マッピングは、他の 1 つのオブジェクトだけから参照されるオブジェクトがあるときに使用する。このような状況は、1 つのオブジェクトが Dependent のコレクションを持つ

ときに起こる。**依存マッピング**は、Owner が Dependent への参照のコレクションを持つが、バックポインタがないなどの状況に適切に対応できる。独自の ID がないオブジェクトが数多くある場合、**依存マッピング**を使用して永続性の管理が容易になる。

依存マッピングを有効に使用するには、以下の前提条件を満たす必要がある。

- Dependent は、必ず 1 つの Owner を持たなければならない。
- Dependent の Owner 以外は、いかなるオブジェクトからも参照されていてはいけない。

Entity（エンティティ）オブジェクトと Dependent オブジェクトを使って**ドメインモデル**を設計するオブジェクト指向設計を教える講義がある。私は、**依存マッピング**とは根本的なオブジェクト指向の手段というよりむしろ、データベースマッピングをシンプルにするための 1 つの技法と考える。特に、大規模な Dependent グラフは避けるようにしている。問題は、グラフの外部から Dependent を参照することができなくなり、ルート Owner の周辺をベースとした複雑な照合スキームに陥りやすい点にある。

ユニットオブワークを使う場合、**依存マッピング**の使用はお勧めしない。**ユニットオブワーク**を使用して記録する場合、削除と再挿入がまったく役立たなくなってしまう。また、Dependent を制御しないため、さまざまな問題が生じる可能性もある。Mike Retting が私に語ったのは、**ユニットオブワーク**がテストのために挿入された行を記録し、テストが済んだ時点でその行を削除するというアプリケーションについてである。この場合は Dependent を記録しなかったため、オーファン（孤立）行が発生しテストでエラーの原因となったのである。

12.4.3 ｜例： Album と Track（Java）

このドメインモデル（図 12.7）では、Album は Track のコレクションを保持している。アプリケーションは実用的ではなくシンプルではあるが、Track を参照するためにはこのアプリケーションだけで十分であり、**依存マッピング**の例としては有用である（パターンのために用意された例として考える人もいるだろう）。

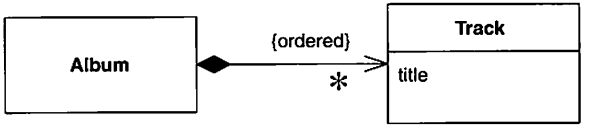


図 12.7——依存マッピングを使用して処理する Track を持つ Album

Track は、Title だけを持つ。私の場合、Track クラスを不変なクラスとして定義している。

```
class Track...

   private final String title;
   public Track(String title) {
      this.title = title;
   }
   public String getTitle() {
      return title;
   }
```

Track は、Album クラスに保持される。

```
class Album...

   private List tracks = new ArrayList();
   public void addTrack(Track arg) {
      tracks.add(arg);
   }
   public void removeTrack(Track arg) {
      tracks.remove(arg);
   };
   public void removeTrack(int i) {
      tracks.remove(i);
   }
   public Track[] getTracks() {
      return (Track[]) tracks.toArray(new Track[tracks.size()]);
   }
```

AlbumMapper クラスは、Track の SQL を処理するため、Track テーブルにアクセスする SQL 文を定義している。

```
class AlbumMapper...
   protected String findStatement() {
      return
         "SELECT ID, a.title, t.title as trackTitle" +
         " FROM albums a, tracks t" +
         " WHERE a.ID = ? AND t.albumID = a.ID" +
         " ORDER BY t.seq";
   }
```

Album（アルバム）が読み込まれるときには、常にTrack（曲目）がAlbum（アルバム）に読み込まれる。

```
class AlbumMapper...

   protected DomainObject doLoad(Long id, ResultSet rs) throws SQLException {
      String title = rs.getString(2);
      Album result = new Album(id, title);
      loadTracks(result, rs);
      return result;
   }
   public void loadTracks(Album arg, ResultSet rs) throws SQLException {
      arg.addTrack(newTrack(rs));
      while (rs.next()) {
         arg.addTrack(newTrack(rs));
      }
   }
   private Track newTrack(ResultSet rs) throws SQLException {
      String title = rs.getString(3);
      Track newTrack = new Track (title);
      return newTrack;
   }
```

わかりやすくするため、この例では分離したクエリーでTrackの読み込みを行っている。パフォーマンスを考えると、262ページの例のように同一のクエリーで各行を読み込むという方法を使用した方がよい場合もある。

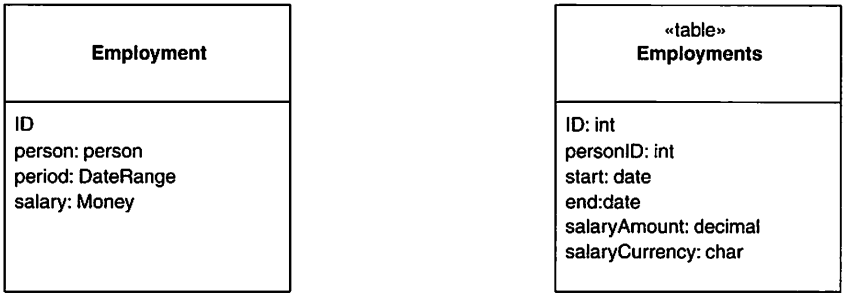
Albumが更新されると、Trackが削除または再挿入される。

```
class AlbumMapper...

   public void update(DomainObject arg) {
      PreparedStatement updateStatement = null;
      try {
         updateStatement = DB.prepare("UPDATE albums SET title = ?
            WHERE id = ?");
         updateStatement.setLong(2, arg.getID().longValue());
         Album album = (Album) arg;
         updateStatement.setString(1, album.getTitle());
         updateStatement.execute();
         updateTracks(album);
      } catch (SQLException e) {
         throw new ApplicationException(e);
```

```
    } finally {DB.cleanUp(updateStatement);
    }
}
public void updateTracks(Album arg) throws SQLException {
    PreparedStatement deleteTracksStatement = null;
    try {
        deleteTracksStatement = DB.prepare("DELETE FROM tracks
            WHERE albumID = ?");
        deleteTracksStatement.setLong(1, arg.getID().longValue());
        deleteTracksStatement.execute();
        for (int i = 0; i < arg.getTracks().length; i++) {
            Track track = arg.getTracks()[i];
            insertTrack(track, i + 1, arg);
        }
    } finally {DB.cleanUp(deleteTracksStatement);
    }
}
public void insertTrack(Track track, int seq, Album album)
    throws SQLException {
    PreparedStatement insertTracksStatement = null;
    try {
        insertTracksStatement = DB.prepare("INSERT INTO tracks
            (seq, albumID, title) VALUES (?, ?, ?)");
        insertTracksStatement.setInt(1, seq);
        insertTracksStatement.setLong(2, album.getID().longValue());
        insertTracksStatement.setString(3, track.getTitle());
        insertTracksStatement.execute();
    } finally {DB.cleanUp(insertTracksStatement);
    }
}
```

12.5 組込バリュー

オブジェクトを他のオブジェクトテーブルの複数フィールドにマッピングする。



データベースのテーブルとしては意味を持たない多くの小さなオブジェクトが、オブジェクト指向システムでは意味を持つ。例では、並行性を重視するMoney（貨幣）オブジェクトとDateRange（日付範囲）オブジェクトを含んでいる。最初に考えるべきことはオブジェクトをテーブルとして保存する方法だが、データがないPerson（人）には、Money値のテーブルは必要ない。

組込バリューは、オブジェクトの値をオブジェクトのOwnerのレコードフィールドにマッピングする。このスケッチでは、Date RangeオブジェクトとMoneyオブジェクトへのリンクを持ったEmploymentオブジェクトを使用している。結果テーブルでは、オブジェクトのフィールドは新しいレコードとして作成されるのではなく、Employmentテーブルのフィールドにマッピングされる。

12.5.1 動作方法

この例の動作はとてもシンプルである。所有しているオブジェクト（Employment）が読み込まれるか保存されると、依存しているオブジェクト（Date RangeやMoney）も同時に読み込まれ保存される。永続性はOwnerによって確保されるため、Dependentが独自の永続性のためのメソッドを持つことは必要ない。**組込バリュー**は特定のケースでの**依存マッピング**と考えられる。この場合値は、1つのDependentオブジェクトになる。

12.5.2 使用するタイミング

動作はとてもシンプルだが、使用するタイミングを判断することが難しいパターンの1つである。

組込バリューを使う最もシンプルなケースは、Money や Date Range のようにシンプルな**バリューオブジェクト**の場合である。**バリューオブジェクト**は、ID を持たないため同期を維持するための**一意マッピング**のようなことを考えることなく、いつでも自由に作成および削除できる。**バリューオブジェクト**のテーブルが必要ないため、**バリューオブジェクト**は**組込バリュー**として維持されるのである。

判断が難しいのは、Order や Shipping オブジェクトなどの参照オブジェクトを、**組込バリュー**で保存する価値があるかどうかである。はじめに検討すべき点は、Shipping データが、Order とは別のコンテキストで関連性があるかどうかである。問題は読み込みと保存である。Order の読み込み時にメモリに Shipping データの読み込みだけを行う場合、同一テーブルに双方を保存することは一考を要する点である。もう 1 つの検討すべき点は、SQL を介して個別に Shipping データにアクセスしたいかどうかである。SQL を介してレポートを行い、他にレポートする分離したデータベースがない場合にこの点が重要となる。

既存のスキーマにマッピングする場合、メモリ上の複数のオブジェクトに分割するデータが、テーブルに含まれているのであれば、**組込バリュー**を使って行う。このような状況が発生する原因は、オブジェクトモデルの振る舞いを抜き出すために、分離したオブジェクトが必要なのだが、データベースでメモリ上にある複数のオブジェクトすべてを 1 つのエンティティとして扱ってしまう場合である。この場合、Dependent の変更により、Owner が不確定なオブジェクトとなることに注意する必要がある（Owner 内で置き換えられる**バリューオブジェクト**にとっては問題ではない）。

多くの場合、オブジェクト間の関連が双方とも単一値（1 対 1 の関連）である時には、参照オブジェクト上で**組込バリュー**を使用すればよいのである。また、複数の Dependent の候補があり、その個数が少なくかつ固定されていれば使用する場合もある。このとき、各値に番号付けされたフィールドを持つ。このような方法はテーブル設計の点でも、SQL によるクエリー実行の点でも複雑だが、パフォーマンス上のメリットがある。しかし、この場合には**シリアライズ LOB** の使用を勧める。

いつ**組込バリュー**を使用するかを決定するロジックの多くは、**シリアライズ LOB** のロジックと同じであるため、双方のどちらを選択するかを検討する。**組込バリュー**の最大のメリットは、Dependent オブジェクトの値に対し SQL クエリーを作成できるという点である。XML を基盤とした SQL への追加クエリーと組み合わせ、直列化機能で XML を使用する方法は将来的には変わるかもしれないが、現時点ではクエリーで Dependent 値を使用する場合には**組込バリュー**が必要となる。**組込バリュー**という点は、データベース上の分離したレポートメカニズムでも有用である。

組込バリューは、シンプルな Dependent にだけ使用できる。独立した Dependent、あるいは 2 ～ 3 の分離した Dependent の場合に正しく機能する。**シリアライズ LOB** は、大規模なオブジェクトサブグラフを含む、より複雑な構造に有効である。

12.5.3 | 参考文献

今まで**組込バリュー**はさまざまな名称で呼ばれてきて、たとえばTOPLinkでは**集合体マッピング**と呼ばれ、Visual Ageでは**コンポーザ**と呼ばれている。

12.5.4 | 例：バリューオブジェクトの例（Java）

組込バリューにマッピングされたバリューオブジェクトの標準的な例である。例では、以下のフィールドを持つシンプルなProductOffering（製品販売）クラスから始める。

```
class ProductOffering...

   private Product product;
   private Money baseCost;
   private Integer ID;
```

フィールド内で、IDは**一意フィールド**であり、Productは標準的なレコードマッピングである。**組込バリュー**を使って、baseCost（基盤となるコスト）をマッピングすることにする。シンプルにするため、**アクティブレコード**によってマッピング全体を行う。

アクティブレコードを使用するため、saveルーチンとloadルーチンが必要となる。これらシンプルなルーチンは、Ownerである ProductOfferingクラスの中に配置されている。Moneyクラスは、永続的な振る舞いを一切持っていない。以下にloadメソッドを示す。

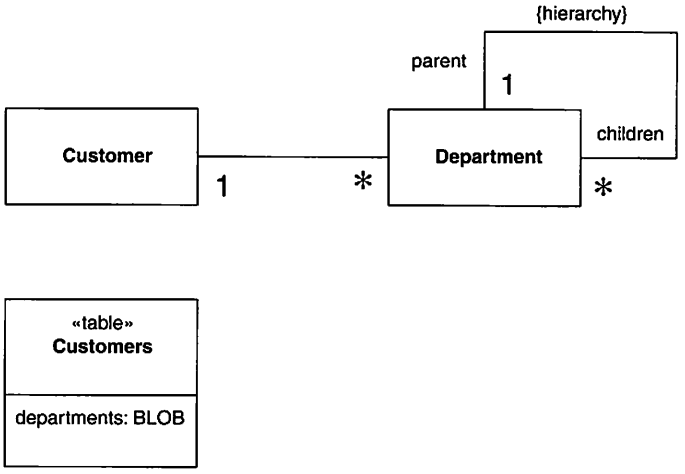
```
class ProductOffering...

   public static ProductOffering load(ResultSet rs) {
      try {
         Integer id = (Integer) rs.getObject("ID");
         BigDecimal baseCostAmount = rs.getBigDecimal("base_cost_amount");
         Currency baseCostCurrency = Registry.getCurrency
            (rs.getString("base_cost_currency"));
         Money baseCost = new Money(baseCostAmount, baseCostCurrency);
         Integer productID = (Integer) rs.getObject("product");
         Product product = Product.find((Integer) rs.getObject("product"));
         return new ProductOffering(id, product, baseCost);
      } catch (SQLException e) {
         throw new ApplicationException(e);
      }
   }
```

以下に更新の振る舞いを示す。更新のシンプルなバリエーションである。

```
class ProductOffering...

   public void update() {
      PreparedStatement stmt = null;
      try {
         stmt = DB.prepare(updateStatementString);
         stmt.setBigDecimal(1, baseCost.amount());
         stmt.setString(2, baseCost.currency().code());
         stmt.setInt(3, ID.intValue());
         stmt.execute();
      } catch (Exception e) {
         throw new ApplicationException(e);
      } finally {DB.cleanUp(stmt); }
   }
   private String updateStatementString =
         "UPDATE product_offerings" +
         " SET base_cost_amount = ?, base_cost_currency = ? " +
         " WHERE id = ?";
```

12.6 | シリアライズ LOB

オブジェクトのグラフを直列化し、1 つの大規模オブジェクト（LOB）としてデータベースフィールドに格納する。



オブジェクトモデルには、小規模オブジェクトの複雑なグラフが含まれることが多い。構造に含まれる情報の多くはオブジェクトの中ではなく、オブジェクト間のリンクの中にある。Customer の組織図を保存する場合を考えてみよう。オブジェクトモデルは、組織図を表示するためのコンポジットパターンを自然に表現できる。また、先祖、兄弟、子孫、およびその他のつながりを取得するためのメソッドも容易に追加できる。

ただし、これらすべてをリレーショナルスキーマに配置することは容易ではない。基本スキーマ、親の外部キーを持った Organization（組織）テーブルはシンプルだが、スキーマの操作にはさまざまな作業が必要となり困難で手間がかかる。

オブジェクトは、相互に関連したテーブル行として永続的である必要はない。永続性を確保する別の形式が直列化である。オブジェクトのグラフ全体が、1 つの大規模オブジェクト（LOB）としてテーブルに書き出された場合、**シリアライズ LOB** はメメント[Gang of Four]の形式になる。

12.6.1 | 動作方法

直列化には、バイナリ形式（**BLOB**）とテキスト形式（**CLOB**）の 2 種類の方法がある。さまざまなプラットフォームは、オブジェクトグラフを自動的に直列化する機能を含んでいるため、バイナリ形式（BLOB）を作成することが最も容易である場合が多い。グラフの保

存とは、バッファで直列化しバッファを関連するフィールドに保存するというシンプルな処理である。

BLOBのメリットとしては、プログラミングが容易であり（プラットフォームがサポートしている場合）、使用するスペースを最小化できるという点が挙げられる。一方短所は、使用するデータベースがバイナリデータ型をサポートしていなければならないという点である。また、そのオブジェクトがないとグラフの再構成ができず、フィールドはまったく理解できないものになってしまう。しかし、最も考えなくてはならない問題はバージョニングである。たとえば、Department（部署）クラスを変更する場合、データはデータベース内に長期間存在することができるため、以前の直列化をすべて読み込むことができない場合がある。これは重要な問題である。

この代替案がCLOBである。この場合、Departmentグラフは、必要な情報を含むテキスト文字列に直列化される。テキスト文字列は、人間が簡単に読むことができるため、データベースの何気ない参照に役立つ。ただしテキスト手法は、より多くのスペースを必要としテキスト形式のために単独のパーサーを作成しなければならない場合もある。さらに、バイナリ直列化よりも速度が遅くなりがちである。

CLOBのデメリットの多くは、XMLによって解決できる。XMLパーサーは一般的なため記述する必要はない。さらに、XMLは幅広くサポートされている標準であり操作に対してもさまざまなツールを使用できる。一方、XMLでも解決できない短所は、スペースの問題である。この形式は冗長であるためスペースの問題は極めて深刻である。1つの対処法としては、ZIP化されたXML形式をBLOBとして使用するという方法がある。人が見て読めなくなるが、スペースが問題となる場合には有効な選択肢である。

シリアライズLOBを使用する場合、IDの問題にも注意する必要がある。たとえば、あるOrderでのCustomerの詳細に**シリアライズLOB**を使用するとしよう。この場合、Orderテーブルに Customer LOBを配置してはいけない。Customerデータが Orderにコピーされ、更新時に問題が発生してしまうからである（しかし、発注時点のCustomerデータのスナップショットをそのまま格納しておきたい場合にはよいことである。一時的な関係を回避できる）。従来のリレーショナルから考えると、各Orderに合わせてCustomerデータを更新したい場合には、LOBをcustomerテーブルに配置して、複数のOrderがLOBにリンクできるようにする必要がある。ID、およびデータのための1つのLOBフィールドだけを持つテーブルに関しては何の問題もない。

一般的にこのパターンを使用する場合には、データの重複に注意する必要がある。**シリアライズLOB**全体が複製されず、その一部分が他のLOBとオーバーラップしてしまう場合も多い。**シリアライズLOB**に格納されるデータには十分な注意を払い、**シリアライズLOB**のOwnerとしての役割を果たす1つのオブジェクト以外からは、確実にデータにアクセスできないようにしなければならない。

12.6.2 | 使用するタイミング

シリアライズLOBは、意外にも検討される機会が少ない。実装が容易なテキスト形式の手法であるXMLは、シリアライズLOBをより使い勝手のよいものに変身させることができる。しかし短所は、SQLを使用する構造にクエリーを発行できない点である。SQL拡張はXMLデータをフィールドのように見せるが、それは同じ（あるいは移植可能）ではない。

このパターンは、オブジェクトモデルの一部を使ってLOBを表現する場合にも有効に機能する。LOBは、アプリケーション外部からSQLクエリーを発行する可能性の低いオブジェクトの束を取得するための方法である。次いでこのグラフはSQLスキーマにフックされる。

LOB参照オブジェクトに外部オブジェクトが埋め込まれている場合、**シリアライズLOB**はうまく機能しない。この状況に対応するには、LOB内部のオブジェクトへの参照をサポートする参照スキームの形式をいくつか考えなくてはいけないが、とても面倒であるため現実に使用されることはほとんどない。XML、あるいはXPathはこのような煩雑さを多少軽減することはできる。

分離したデータベースをレポートのために使用していて、他のすべてのSQLがそのデータベースに対して発行されている場合、LOBを適切なテーブル構造に変換することができる。レポート用のデータベースは非正規化されるため、**シリアライズLOB**に適した構造は、分離したレポート用のデータベースにも適している場合が多い。

12.6.3 | 例：XMLによるDepartment階層構造の直列化（Java）

ここでは、スケッチのCustomerとDepartmentという概念を例に挙げて、DepartmentをXML CLOBに直列化する方法を解説する。JavaのXML処理はややプリミティブで不安定であるため、最終的に生成されるコードには若干の差異が見られることがある（私は初期バージョンのJDOMを使用している）。

このスケッチのオブジェクトモデルは、以下のクラス構造を構築する。

```
class Customer...

   private String name;
   private List departments = new ArrayList();

class Department...
   private String name;
   private List subsidiaries = new ArrayList();
```

この場合のデータベースが持っているテーブルは、1つだけである。

```
create table customers (ID int primary key, name varchar, departments varchar)
```

Customer（顧客）をアクティブレコードとして扱い、挿入の振る舞いによるデータの書き込みを説明する。

```
class Customer...
   public Long insert() {
      PreparedStatement insertStatement = null;
      try {
         insertStatement = DB.prepare(insertStatementString);
         setID(findNextDatabaseId());
         insertStatement.setInt(1, getID().intValue());
         insertStatement.setString(2, name);
         insertStatement.setString(3,
            XmlStringer.write(departmentsToXmlElement()));
         insertStatement.execute();
         Registry.addCustomer(this);
         return getID();
      } catch (SQLException e) {
         throw new ApplicationException(e);
      } finally {DB.cleanUp(insertStatement);
      }
   }
   public Element departmentsToXmlElement() {
      Element root = new Element("departmentList");
      Iterator i = departments.iterator();
      while (i.hasNext()) {
         Department dep = (Department) i.next();
         root.addContent(dep.toXmlElement());
      }
      return root;
   }

class Department...

   Element toXmlElement() {
      Element root = new Element("department");
      root.setAttribute("name", name);
      Iterator i = subsidiaries.iterator();
      while (i.hasNext()) {
```

```
            Department dep = (Department) i.next();
            root.addContent(dep.toXmlElement());
        }
        return root;
    }
```

Customer クラスは、Department フィールドを 1 つの XML DOM に直列化するためのメソッドを持つ。各 Department クラスもまた、自ら（およびその再帰的な子孫）を DOM に直列化するメソッドを持つ。次に insert メソッドは、Department の DOM を取得し文字列に変換して（ユーティリティクラスを介して）、データベースに配置する。文字列の構造について、ここでは特に気にする必要はない。読むことはできるが定期的に参照することはないだろう。

```
<?xml version="1.0" encoding="UTF-8"?>
<departmentList>
    <department name="US">
        <department name="New England">
            <department name="Boston" />
            <department name="Vermont" />
        </department>
        <department name="California" />
        <department name="Mid-West" />
    </department>
    <department name="Europe" />
</departmentList>
```

読み込みは、このようなプロセスを反対にしたものに過ぎない。

```
class Customer...

    public static Customer load(ResultSet rs) throws SQLException {
            Long id = new Long(rs.getLong("id"));
            Customer result = (Customer) Registry.getCustomer(id);
            if (result != null) return result;
            String name = rs.getString("name");
            String departmentLob = rs.getString("departments");
            result = new Customer(name);
            result.readDepartments(XmlStringer.read(departmentLob));
            return result;
    }
    void readDepartments(Element source) {
```

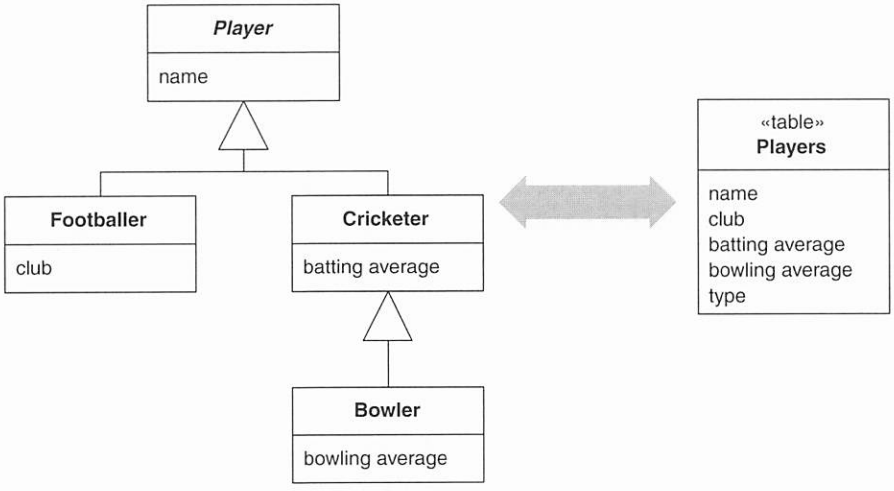
```
        List result = new ArrayList();
        Iterator it = source.getChildren("department").iterator();
        while (it.hasNext())
            addDepartment(Department.readXml((Element) it.next()));
    }

class Department...

    static Department readXml(Element source) {
        String name = source.getAttributeValue("name");
        Department result = new Department(name);
        Iterator it = source.getChildren("department").iterator();
        while (it.hasNext())
            result.addSubsidiary(readXml((Element) it.next()));
        return result;
    }
```

読み込みコードは、明らかに挿入コードのミラーイメージである。Department クラスは、XML 要素から自ら（およびその子孫）を作成する方法を知っていて、Customer クラスは XML 要素を取得して、そこから Department リストを作成する方法を知っている。load メソッドは、ユーティリティクラスを使用して、データベースから取得した文字列をユーティリティ要素に変換する。

ここで注意しなくてはならないのは、誰かがデータベース内の XML を手動で編集しようとして XML を壊し、この load ルーチンによる読み込みができなくなってしまう場合である。妥当性確認のために DTD または XML スキーマをフィールドに追加する機能をサポートできるツールがあれば、このような状況を回避するためにとても役立つ。

12.7 | シングルテーブル継承

クラスの継承階層構造を、さまざまなクラスのフィールドに対する列を持つ、1つのテーブルとして作成する。



リレーショナルデータベースは、継承をサポートしていないため、オブジェクトからデータベースにマッピングする時には、リレーショナルテーブル内で優れた継承構造を作成する方法について考えなければならない。リレーショナルデータベースへのマッピングを行う場合、私たちは最小限のジョインで処理しようとする。ジョインは、複数テーブルの継承構造を処理すると、急速に増えてしまう。**シングルテーブル継承**でも、継承構造のクラスのフィールドを1つのテーブルにマッピングするのである。

12.7.1 | 動作方法

継承マッピングスキームでは、継承階層構造のクラスすべてのデータを含む1つのテーブルを持つことになる。各クラスには、1つのテーブル行に継承に関連するデータが格納される。データベース内の関連のない列は空のままとなる。マッピングの基本的な振る舞いは、**継承マッパー**の汎用スキームに従っている。

オブジェクトをメモリに読み込む場合、どのクラスをインスタンス化するのかを知っておく必要がある。そのため、どのクラスを使用するかを指示するフィールドをテーブルに持たせる。このフィールドは、クラス名かコードフィールドのいずれかである。フィールドを関連するクラスにマッピングするためには、コードフィールドは何らかのコードによって解釈されるので、クラスが階層構造に追加されるときには、コードを拡張する必要がある。クラ

ス名をテーブルに組み込む場合、クラス名を直接使うだけでインスタンスを作成することができる。しかしクラス名は、多くのスペースを占有するので、データベースのテーブル構造をクラス名で直接処理するのは困難な場合がある。また、データベーススキーマにクラス構造をより密接に結合してしまう場合もある。

データの読み込みでは、先ずコードを読み込み、どのサブクラスをインスタンス化するかを知る必要がある。データの保存時には、コードは階層構造のスーパークラスによって書き出される。

12.7.2 | 使用するタイミング

シングルテーブル継承は、継承階層構造のフィールドをリレーショナルデータベースにマッピングするための選択肢の1つである。代替案としては、**クラステーブル継承**と**具象テーブル継承**がある。

シングルテーブル継承のメリットは、以下のとおりである。

- テーブルがデータベースに1つしかない。
- データを抽出するためのジョインはない。
- 階層構造に対するフィールドをリファクタリングしても、データベースを変更する必要はない。

一方、**シングルテーブル継承**の短所は以下のとおりである。

- フィールドは、関連を持つ場合と持たない場合があり、テーブルを直接使う人の混乱を招く原因ともなる。
- 一部のサブクラスだけが使う列によって、データベースのスペースが浪費される。スペースがどの程度問題となるかは、データの性質と、データベースによる空きフィールドの圧縮率によって左右される。たとえばOracleは、特にデータベーステーブルの右側に追加列を確保している場合、浪費スペースのトリミング効率に優れている。それぞれのデータベースは、このような問題に対処するための特別な仕掛けの構造を持っている。
- シングルテーブルが巨大化し、インデックスも多くロックも頻繁に行われる結果、パフォーマンスに悪影響が及ぶことがある。これを回避するには、特定のプロパティを持つ行のキーをリストアップするか、インデックスに関連のあるフィールドのサブセットをコピーする分離したインデックステーブルを持つ。

■ フィールドに対して1つの名前スペースだけしか持っていないため、異なるフィールドに対して同じ名前を使用しないように注意しなければならない。この場合、クラス名にプレフィックスまたはサフィックスを付けた複合名が役立つ。

階層構造全体に、継承マッピングの1つの形式を使う必要はないことを忘れないでほしい。大量の特定のデータを持つクラスに対して、**具象テーブル継承**を使う場合は、1つのテーブル内にある類似した6つのクラスを完全にマッピングできる。

12.7.3 ｜例： Player のためのシングルテーブル（C#）

他の継承例と同様に、ここでは**継承マッパー**を基に、図12.8に示すクラスを使用している。各マッパーは、ADO.NET データセットのデータテーブルにリンクしている。リンクは、マッパースーパークラスで汎用的に作成できる。ゲートウェイのデータプロパティは、クエリーによって読み込みできるデータセットである。

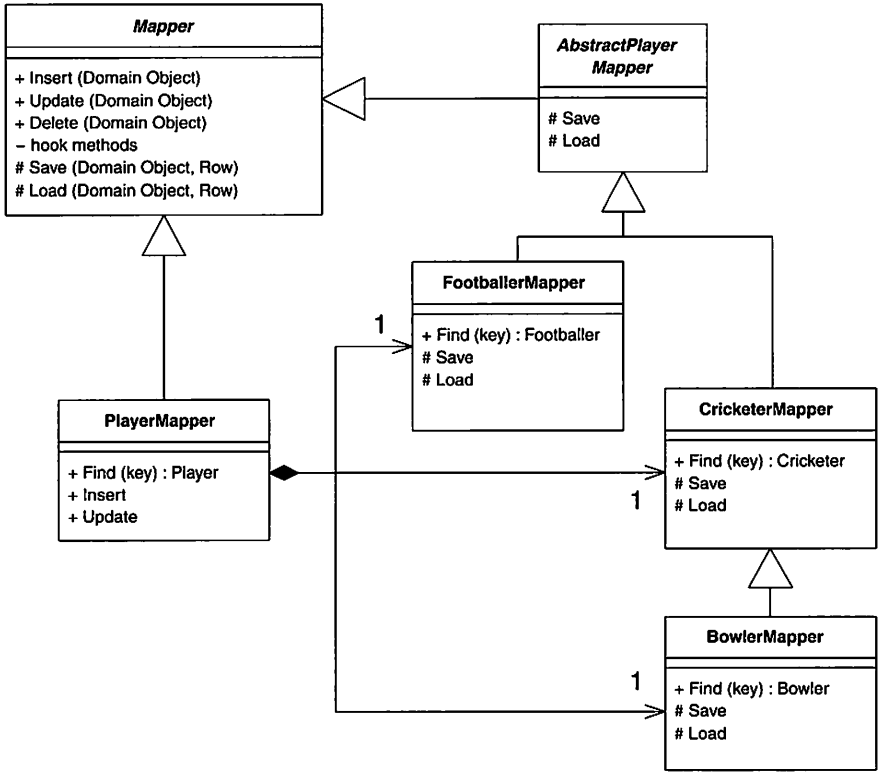


図 12.8 —— 継承マッパーの汎用クラス図

```
class Mapper...
      protected DataTable table {
         get {return Gateway.Data.Tables[TableName];}
      }
      protected Gateway Gateway;
      abstract protected String TableName {get;}
```

この例にはテーブルが1つしかないため、AbstractPlayerMapperクラスによって定義する。

```
class AbstractPlayerMapper...

      protected override String TableName {
         get {return "Players";}
      }
```

それぞれのクラスには、処理対象となるPlayerの種類をマッパーコードが把握できるようにするために、タイプコードが必要になる。タイプコードは、スーパークラスで定義されサブクラスに実装される。

```
class AbstractPlayerMapper...

      abstract public String TypeCode {get;}

class CricketerMapper...

      public const String TYPE_CODE = "C";
      public override String TypeCode {
         get {return TYPE_CODE;}
      }
```

PlayerMapperクラスは、3つの具象Mapperクラスに対するフィールドを持っている。

```
class PlayerMapper...

      private BowlerMapper bmapper;
      private CricketerMapper cmapper;
      private FootballerMapper fmapper;
      public PlayerMapper (Gateway gateway) : base (gateway) {
         bmapper = new BowlerMapper(Gateway);
         cmapper = new CricketerMapper(Gateway);
         fmapper = new FootballerMapper(Gateway);
      }
```

12.7.4 | データベースからのオブジェクトの読み込み

各々の具象 Mapper クラスは、データからオブジェクトを取得する Find メソッドを持っている。

```
class CricketerMapper...

      public Cricketer Find(long id) {
         return (Cricketer) AbstractFind(id);
      }
```

以下の例は、オブジェクトを検索する汎用の振る舞いを呼び出す。

```
class Mapper...

      protected DomainObject AbstractFind(long id) {
         DataRow row = FindRow(id);
         return (row == null) ? null : Find(row);
      }
      protected DataRow FindRow(long id) {
         String filter = String.Format("id = {0}", id);
         DataRow[] results = table.Select(filter);
         return (results.Length == 0) ? null : results[0];
      }
      public DomainObject Find (DataRow row) {
         DomainObject result = CreateDomainObject();
         Load(result, row);
         return result;
      }
      abstract protected DomainObject CreateDomainObject();

class CricketerMapper...

      protected override DomainObject CreateDomainObject() {
         return new Cricketer();
      }
```

私の場合は、一連の Load メソッドによってデータを階層構造中の各クラスの新規オブジェクトに読み込む。

```
class CricketerMapper...
```

```
    protected override void Load(DomainObject obj, DataRow row) {
        base.Load(obj,row);
        Cricketer cricketer = (Cricketer) obj;
        cricketer.battingAverage = (double)row["battingAverage"];
    }

class AbstractPlayerMapper...

    protected override void Load(DomainObject obj, DataRow row) {
        base.Load(obj, row);
        Player player = (Player) obj;
        player.name = (String)row["name"];
    }

class Mapper...

    protected virtual void Load(DomainObject obj, DataRow row) {
        obj.Id = (int) row ["id"];
    }
```

また、PlayerMapper クラスを介して Player を読み込むこともできる。これを実現するにはデータを読み込み、どの具象マッパーを使うかを判断するためにタイプコードを使う。

```
class PlayerMapper...

    public Player Find (long key) {
        DataRow row = FindRow(key);
        if (row == null) return null;
        else {
            String typecode = (String) row["type"];
            switch (typecode){
                case BowlerMapper.TYPE_CODE:
                    return (Player) bmapper.Find(row);
                case CricketerMapper.TYPE_CODE:
                    return (Player) cmapper.Find(row);
                case FootballerMapper.TYPE_CODE:
                    return (Player) fmapper.Find(row);
                default:
                    throw new Exception("unknown type");
            }
        }
    }
```

12.7.4.1 ■ オブジェクトの更新

更新のための基本操作は、すべてのオブジェクトで共通であるためマッパースーパークラスに定義できる。

```
class Mapper...

      public virtual void Update (DomainObject arg) {
         Save (arg, FindRow(arg.Id));
      }
```

SaveメソッドはLoadメソッドと似ていて、各クラスはSaveメソッドに含まれるデータを保存するように定義される。

```
class CricketerMapper...

      protected override void Save(DomainObject obj, DataRow row) {
         base.Save(obj, row);
         Cricketer cricketer = (Cricketer) obj;
         row["battingAverage"] = cricketer.battingAverage;
      }

class AbstractPlayerMapper...

      protected override void Save(DomainObject obj, DataRow row) {
         Player player = (Player) obj;
         row["name"] = player.name;
         row["type"] = TypeCode;
      }
```

PlayerMapperクラスは適切な具象マッパーを返す。

```
class PlayerMapper...

      public override void Update (DomainObject obj) {
         MapperFor(obj).Update(obj);
      }
      private Mapper MapperFor(DomainObject obj) {
         if (obj is Footballer) return fmapper;
         if (obj is Bowler) return bmapper;
         if (obj is Cricketer) return cmapper;
         throw new Exception("No mapper available");
      }
```

12.7.4.2 ■ オブジェクトの挿入

挿入は更新と似ているが、唯一の違いは新しい行をテーブル内に作成した後、保存する必要がある点である。

```
class Mapper...

      public virtual long Insert (DomainObject arg) {
         DataRow row = table.NewRow();
         arg.Id = GetNextID();
         row["id"] = arg.Id;
         Save (arg, row);
         table.Rows.Add(row);
         return arg.Id;
      }

class PlayerMapper...

      public override long Insert (DomainObject obj) {
         return MapperFor(obj).Insert(obj);
      }
```

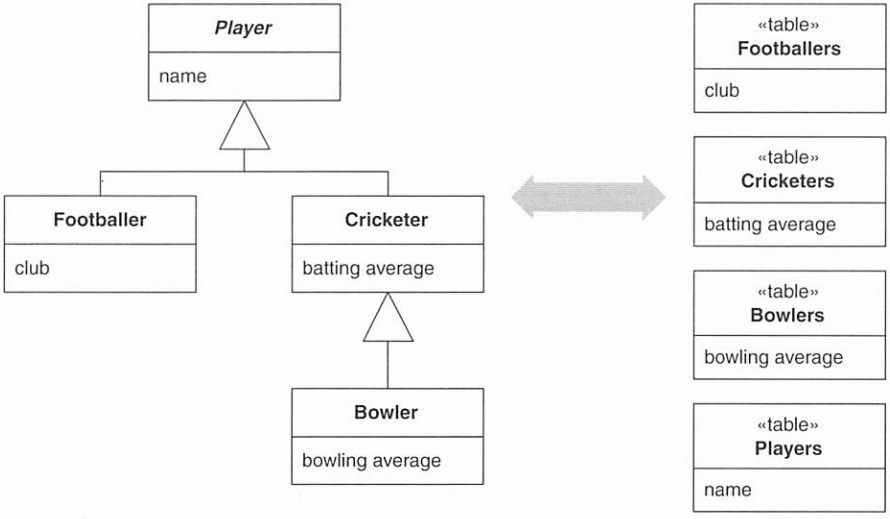
12.7.4.3 ■ オブジェクトの削除

削除はとてもシンプルである。削除は抽象マッパーレベル、またはPlayerMapperに定義される。

```
class Mapper...

      public virtual void Delete(DomainObject obj) {
         DataRow row = FindRow(obj.Id);
         row.Delete();
      }

class PlayerMapper...

      public override void Delete (DomainObject obj) {
         MapperFor(obj).Delete(obj);
      }
```

12.8 | クラステーブル継承

クラスごとに1つのテーブルを使って、クラスの継承階層構造を作成する。



オブジェクトとリレーショナルのミスマッチが顕著に現れるのは、リレーショナルデータベースが継承をサポートしていない点である。求められるデータベース構造は、オブジェクトに対して明確にマッピングされ、階層構造のどこへでもリンクできる構造である。**クラステーブル継承**は、継承構造のクラスごとに1つのデータベーステーブルを使うことで、このような要件をサポートしている。

12.8.1 | 動作方法

ドメインモデルのクラスごとに1つのテーブルを持つ**クラステーブル継承**は、とてもわかりやすいパターンである。ドメインクラスのフィールドは、対応するテーブルのフィールドへ直接マッピングされる。他の継承マッピングと同様、ここでも**継承マッパー**の基本的な手法が適用されている。問題となるのは、データベーステーブルの対応する行へのリンクの方法である。考えられる解決策としては、共通のプライマリキーの値を使う方法が挙げられる。たとえば、Footballers（フットボールプレイヤー）テーブルのキー101の行と、Playersテーブルのキー101の行は、同じドメインオブジェクトに対応する。スーパークラステーブルは、他のテーブルの各行にそれぞれ対応する行を持っているため、この方法を使用する場合、プライマリキーはすべてのテーブル間で一意となる。代替案として、各テーブルに単独のプライマリキーを持たせ、スーパークラステーブルに対する外部キーを使って行を結び付

ける方法もある。

クラステーブル継承による実装に関する最大の問題は、複数テーブルからのデータの抽出をいかに効率的に実行するかである。当然、データベースに対する複数の呼び出しがすでにあるため、各テーブルに呼び出しを行うことはあまり効率的な方法とは言えない。これを回避するには、さまざまなコンポーネントテーブルにまたがったジョインを行う必要があるが、3つまたは4つ以上のテーブルのジョインは、データベースが最適化を行う方法によっては速度低下につながる傾向がある。

さらに大きな問題は、所定のあらゆるクエリー内で、ジョイン対象であるテーブルを正確に把握できなくなってしまうことが多いという点である。たとえば、Footballer（フットボールプレイヤー）を検索する場合、Footballer テーブルを使うことはわかっている。しかし、任意の Group に属している Player を探す場合、どのテーブルを使用するかは容易に判断できない。一部のテーブルにデータがない場合に効率的にジョインを行うために、外部ジョインを行う必要があるが、これは標準規格ではなく、速度が低下する場合も多い。代替案として、最初にルートテーブルを読み込んだ後、コードを使って次に読み込むテーブルを判断するという方法も考えられるが、これには複数のクエリーが必要となる。

12.8.2 | 使用するタイミング

継承マッピングを考える場合、**クラステーブル継承**、**シングルテーブル継承**および、**具象テーブル継承**の3つの選択肢がある。

クラステーブル継承のメリットは以下のとおりである。

- すべての列が各行に関連を持つため、テーブルの関係がわかりやすくスペースも浪費されない。
- ドメインモデルとデータベース間の関連が、シンプルでわかりやすい。

一方、**クラステーブル継承**の短所は以下のとおりである。

- オブジェクトを読み込むために、複数のテーブルに作用する必要がある。これはジョインまたはメモリ内での複数のクエリーとソーイングが必要なことを意味する。
- 階層構造に対するフィールドのリファクタリングが、データベースの変化を引き起こす。
- スーパータイプテーブルは頻繁にアクセスされるため、ボトルネックとなる可能性がある。

■ 高度な正規化によって、個別クエリーの把握が困難になる。

1つのクラス階層構造に対して継承マッピングパターンを1つに限定する必要はない。階層構造の最上部のクラスに**クラステーブル継承**を使用し、そのクラスより下位のクラスに一連の**具象テーブル継承**を使うこともできる。

12.8.3 | 参考文献

IBMのテキストでは、このパターンは**ルートリーフマッピング**[Brown et al.]と呼ばれている。

12.8.4 | 例：Playerとその Kin（仲間）（C#）

スケッチの実装は以下のとおりである。ここでは再び、図12.9の**継承マッパー**を使って、Playerとその仲間というわかりやすいテーマに沿って進める。

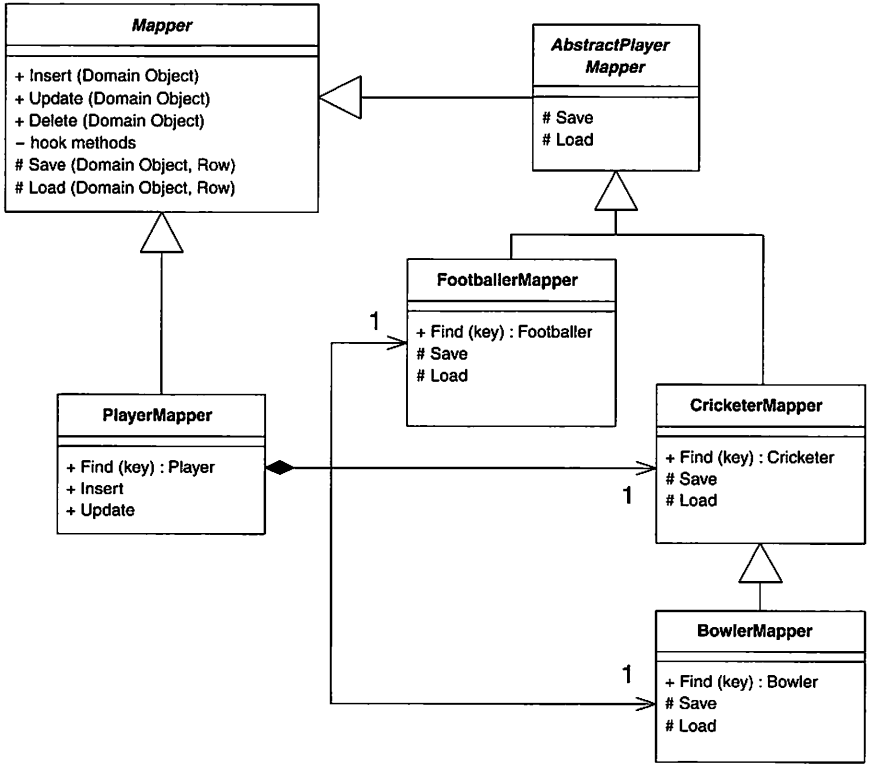


図12.9——継承マッパーの汎用クラス図

各クラスは、データとタイプコードを保持するテーブルを定義する必要がある。

```
class AbstractPlayerMapper...

      abstract public String TypeCode {get;}
      protected static String TABLENAME = "Players";

class FootballerMapper...

      public override String TypeCode {
         get {return "F";}
      }
      protected new static String TABLENAME = "Footballers";
```

他の継承例とは異なり、この例にはオーバーライドされるテーブル名はない。これはインスタンスがサブクラスのインスタンスである場合であっても、クラスに対するテーブル名を持たなければならないためである。

12.8.4.1 ■ オブジェクトの読み込み

すでに他のマッピングを読み込んでいる場合、最初のステップは具象マッパーのFindメソッドである。

```
class FootballerMapper...

      public Footballer Find(long id) {
         return (Footballer) AbstractFind (id, TABLENAME);
      }
```

AbstractFindメソッドは、キーに一致する行を探し、検索に成功するとドメインオブジェクトを作成し、そのLoadメソッドを呼び出す。

```
class Mapper...

      public DomainObject AbstractFind(long id, String tablename) {
         DataRow row = FindRow (id, tableFor(tablename));
         if (row == null) return null;
         else {
            DomainObject result = CreateDomainObject();
            result.Id = id;
            Load(result);
```

```
            return result;
        }
    }
    protected DataTable tableFor(String name) {
        return Gateway.Data.Tables[name];
    }
    protected DataRow FindRow(long id, DataTable table) {
        String filter = String.Format("id = {0}", id);
        DataRow[] results = table.Select(filter);
        return (results.Length == 0) ? null : results[0];
    }
    protected DataRow FindRow (long id, String tablename) {
        return FindRow(id, tableFor(tablename));
    }
    protected abstract DomainObject CreateDomainObject();

class FootballerMapper...

    protected override DomainObject CreateDomainObject(){
        return new Footballer();
    }
```

各クラスには、クラスが定義するデータを読み込むLoadメソッドが1つある。

```
class FootballerMapper...

    protected override void Load(DomainObject obj) {
        base.Load(obj);
        DataRow row = FindRow (obj.Id, tableFor(TABLENAME));
        Footballer footballer = (Footballer) obj;
        footballer.club = (String)row["club"];
    }

class AbstractPlayerMapper...

    protected override void Load(DomainObject obj) {
        DataRow row = FindRow (obj.Id, tableFor(TABLENAME));
        Player player = (Player) obj;
        player.name = (String)row["name"];
    }
```

他の例のコードと同様（ただしこの例の方がより顕著だが）、ここでも私は、ADO.NETデータセットはデータベースからデータを抽出し、それをメモリにキャッシュすることに頼っている。これによりテーブルベースのデータ構造への複数のアクセスが可能となり、パ

フォーマンスコストが大幅に低下することはない。データベースに直接アクセスする場合は、このような負荷を削減する必要がある。この例では、テーブルにまたがるジョインを作成し操作することで負荷を削減してある。

PlayerMapper クラスは、検索する必要のある Player の種類を決定し、正しい具象マッパーへ委譲する。

```
class PlayerMapper...

      public Player Find (long key) {
         DataRow row = FindRow(key, tableFor(TABLENAME));
         if (row == null) return null;
         else {
            String typecode = (String) row["type"];
            if (typecode == bmapper.TypeCode)
               return bmapper.Find(key);
            if (typecode == cmapper.TypeCode)
               return cmapper.Find(key);
            if (typecode == fmapper.TypeCode)
               return fmapper.Find(key);
            throw new Exception("unknown type");
         }
      }
      protected static String TABLENAME = "Players";
```

12.8.4.2 ■ オブジェクトの更新

Update メソッドは、マッパースーパークラスに配置される。

```
class Mapper...
      public virtual void Update (DomainObject arg) {
         Save (arg);
      }
```

（階層構造の各クラスが 1 つずつ持っている）一連の Save メソッドを介して実装される。

```
class FootballerMapper...

      protected override void Save(DomainObject obj) {
         base.Save(obj);
         DataRow row = FindRow (obj.Id, tableFor(TABLENAME));
         Footballer footballer = (Footballer) obj;
```

```
            row["club"] = footballer.club;
        }
class AbstractPlayerMapper...

        protected override void Save(DomainObject obj) {
            DataRow row = FindRow (obj.Id, tableFor(TABLENAME));
            Player player = (Player) obj;
            row["name"] = player.name;
            row["type"] = TypeCode;
}
```

PlayerMapper（プレイヤーマッパー）クラスのUpdateメソッドは、正しい具象マッパーに委譲するための汎用メソッドをオーバーライドする。

```
class PlayerMapper...

        public override void Update (DomainObject obj) {
            MapperFor(obj).Update(obj);
        }
        private Mapper MapperFor(DomainObject obj) {
            if (obj is Footballer)
                return fmapper;
            if (obj is Bowler)
                return bmapper;
            if (obj is Cricketer)
                return cmapper;
            throw new Exception("No mapper available");
        }
```

12.8.4.3 ■ オブジェクトの挿入

オブジェクトを挿入するためのメソッドは、マッパースーパークラスで宣言される。これには次の2つの段階がある。まずは、新しいデータベース行を作成し、次にSaveメソッドを使い空白行を必要なデータで更新する。

```
class Mapper...

        public virtual void Update (DomainObject arg) {
            Save (arg);
        }
```

各クラスは、テーブルに1つの行を挿入する。

```
class FootballerMapper...

      protected override void AddRow (DomainObject obj) {
         base.AddRow(obj);
         InsertRow (obj, tableFor(TABLENAME));
      }

class AbstractPlayerMapper...

      protected override void AddRow (DomainObject obj) {
         InsertRow (obj, tableFor(TABLENAME));
      }

class Mapper...

      abstract protected void AddRow (DomainObject obj);
      protected virtual void InsertRow (DomainObject arg, DataTable table) {
         DataRow row = table.NewRow();
         row["id"] = arg.Id;
         table.Rows.Add(row);
      }
```

PlayerMapper クラスは、具象マッパーへの委譲を行う。

```
class PlayerMapper...

      public override long Insert (DomainObject obj) {
         return MapperFor(obj).Insert(obj);
      }
```

12.8.4.4 ■ オブジェクトの削除

オブジェクトを削除する場合、各クラスはデータベースの対応するテーブルから行を削除する。

```
class FootballerMapper...

      public override void Delete(DomainObject obj) {
         base.Delete(obj);
         DataRow row = FindRow(obj.Id, TABLENAME);
         row.Delete();
      }

class AbstractPlayerMapper...
```

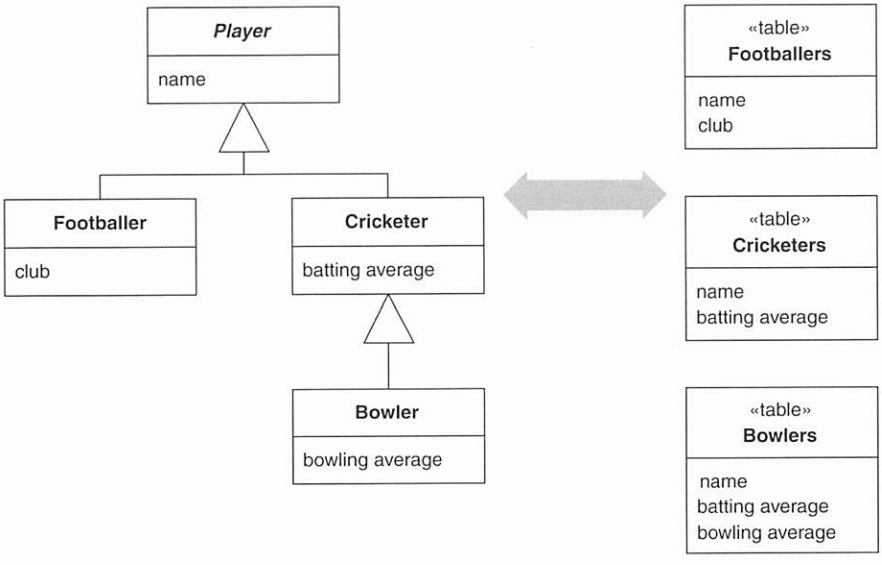
```
        public override void Delete(DomainObject obj) {
            DataRow row = FindRow(obj.Id, tableFor(TABLENAME));
            row.Delete();
        }

class Mapper...

        public abstract void Delete(DomainObject obj);
```

PlayerMapper クラスは、ここでも作業を軽減するため具象マッパーへと委譲を行う。

```
class PlayerMapper...

        override public void Delete(DomainObject obj) {
            MapperFor(obj).Delete(obj);
        }
```

12.9 | 具象テーブル継承

階層構造の具象クラスごとに 1 つのテーブルを使って、クラスの継承階層構造を作成する。



純粋なオブジェクト指向主義者が主張するように、リレーショナルデータベースは継承をサポートしていない。これがオブジェクトリレーショナルマッピングを複雑にしている理由

である。オブジェクトインスタンスの観点からテーブルを考えた場合、最も有効な手段は、メモリ上の各オブジェクトを取得し、1つのデータベース行にマッピングすることである。この方法は、継承階層構造の各具象クラスに1つのテーブルがある**具象テーブル継承**を暗に示唆している。

このパターンに対して難しい名前を付けてしまったのは私である。多くの人は、階層構造のリーフクラスごとに1つのテーブルを持つことから、これをリーフ指向と呼んでいる。このロジックに従うと、このパターンをリーフテーブル継承と呼ばざるを得ず、「リーフ」という用語をパターンで使用することが多い。しかし、厳密にはリーフではない具象クラスもテーブルを持つことがあるため、あまり直感的な用語ではないと思うが、より正確な表現を使うことにしたのである。

12.9.1 ｜動作方法

具象テーブル継承は、階層構造の各具象クラスに1つのデータベーステーブルを使用する。各テーブルには、具象クラスと列が含まれるため、スーパークラスのフィールドはサブクラスのテーブルにまたがってコピーされる。他の継承スキームと同様に基本的な振る舞いは、**継承マッパー**を使用する。

このパターンにおいては、キーに注意する点がある。重要なポイントは、キーは1つのテーブルに対してだけでなく、必ず階層構造のすべてのテーブルに対して一意になるようにすることである。これが必要なケースとは、たとえばPlayerのコレクションを持ち、テーブルワイドなキーを持った**一意フィールド**を使用するような場合である。具象クラスをマッピングするテーブル間でキーの複製が可能な場合、特定のキーの値に対して複数の行を取得することになる。そのため、テーブル全体のキーの使用方法を記録するキー割り当てシステムが必要となり、データベースのプライマリキーの一意性メカニズムは使用できなくなる。

そのため、他のシステムのデータベースに接続する場合、状況は複雑になる。多くの場合、テーブル間でのキーの一意性は保証されないので、スーパークラスフィールドの使用を回避するかテーブル識別子を含む複合キーを使用することになる。

この状況の一部は、スーパークラスに型付けされていないフィールドを持つことで回避できるが、明らかにオブジェクトモデルの妥協である。代替案としては、スーパータイプではインタフェースとしてアクセッサーを持ち、各具象タイプでは実装としてプライベートフィールドを持つ方法がある。インタフェースではプライベートフィールドから値を取得して返す。単一値の場合は、nullではないプライベート値を返し、コレクション値の場合は、プライベートフィールドの値を組み合わせて返す。

複合キーの場合、特別なキーオブジェクトを**一意フィールド**のIDフィールドとして使用

する。キーはテーブルのプライマリキーとテーブル名の両方を使って一意性を判断する。

この点に関する問題は、データベースでの参照整合性である。ここでは図12.10に示すオブジェクトモデルを取り上げてみよう。参照整合性を実装するには、Charity Function（チャリティイベント）とPlayerの外部キー列を含むリンクテーブルが必要となる。問題は、Playerのテーブルがない点であり、Footballer、Cricketer（クリケットプレイヤー）のいずれかを取得する外部キーの参照整合性の制約をできないことである。残された選択肢は、参照整合性を無視するか、複数のリンクテーブル（データベースの実際のテーブルに対して1つずつ持つ）を使うかである。最大の問題は、キーの一意性を保証できるかである。

select文によってPlayerを検索する場合、適切な値がどのテーブルに含まれているかを判断するには、すべてのテーブルをチェックしなければならない。これは、複数のクエリーを使うか、外部ジョインを使うかになるが、どちらもパフォーマンスに悪影響を与える。必要なクラスがわかっていれば、パフォーマンスへの影響は避けられるが、パフォーマンスを改善するためには、具象クラスを使う必要がある。

このパターンは、**リーフテーブル継承**とも呼ばれている。具象クラスごとに1つのテーブルを持たせるのではなくて、リーフクラスごとに1つのテーブルを持たせるバリエーションを好む人もいる。階層構造に具象スーパークラスがない場合にも同じ結果となる。具象スーパークラスがある場合でもその差異は極めて小さい。

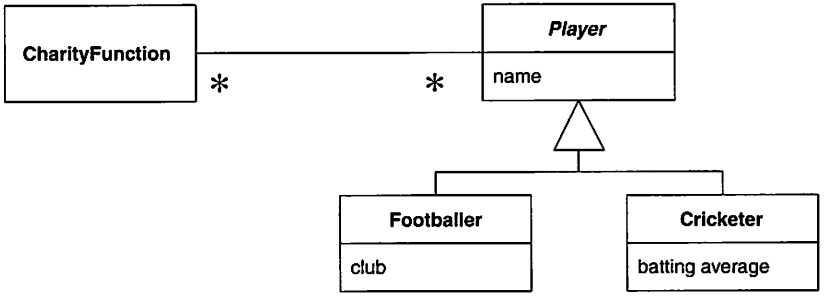


図12.10——具象テーブル参照に参照整合性の問題を引き起こすモデル

12.9.2 | 使用するタイミング

継承のマッピング方法を決める代替案としては、**具象テーブル継承**、**クラステーブル継承**、および**シングテーブル継承**がある。

具象テーブル継承のメリットは、以下のとおりである。

■ 各テーブルは自己完結していて、関連性のあるフィールドを持たない。結果と

して、オブジェクトを使用しない他のアプリケーションから使われる際に役立つ。

- 具象マッパーからデータを読み込む場合、ジョインの必要がない。
- 各テーブルはクラスがアクセスされるときにだけアクセスされるため、アクセス負荷が分散される。

具象テーブル継承の短所は、以下のとおりである。

- プライマリキーが扱いにくい。
- 抽象クラスに対してデータベースの関連付けを実行できない。
- ドメインクラスのフィールドが階層構造で上下に移動された場合、テーブル定義を変更しなければならない。**クラステーブル継承**ほどの変更は必要ないが、**シングルテーブル継承**のときのように無視することはできない。
- スーパークラスフィールドが変化する場合、スーパークラスフィールドはテーブル間で複製されるため、このフィールドを持つ各テーブルを修正する必要がある。
- スーパークラスの検索によって、すべてのテーブルのチェックが強制され、複数のデータベースアクセス（あるいは正常とは言えないジョイン）へとつながっていく。

この3つの継承パターンは1つの階層構造に混在できるということを覚えておいてほしい。そのため、1つまたは2つのサブクラスに対して**具象テーブル継承**を使い、残りに**シングルテーブル継承**を使うこともある。

12.9.3 ｜例：具象 Player（C#）

次に、スケッチの実装を示すことにしよう。本章の他の継承例と同様、ここでも図12.11で示した**継承マッパー**におけるクラスの基本設計を使っている。

各マッパーは、データのソースであるデータベーステーブルにリンクされている。ADO.NETでは、データセットはデータテーブルを保持する。

```
class Mapper...

      public Gateway Gateway;
      private IDictionary identityMap = new Hashtable();
```

```
public Mapper (Gateway gateway) {
   this.Gateway = gateway;
}
private DataTable table {
   get {return Gateway.Data.Tables[TableName];}
}
abstract public String TableName {get;}
```

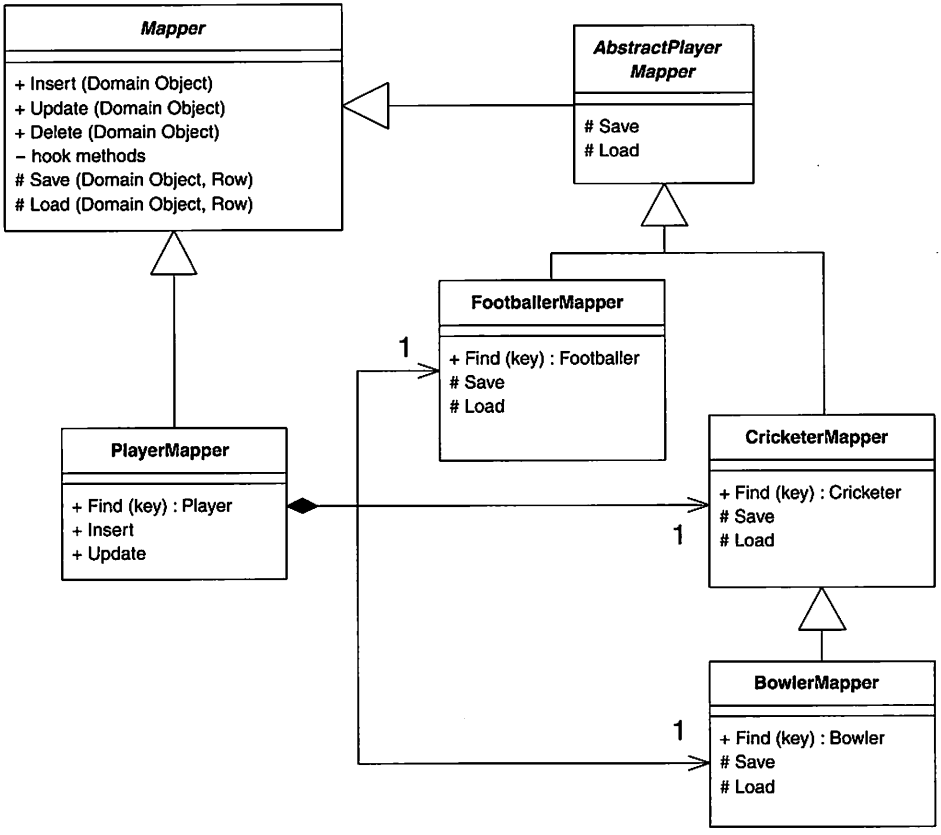


図 12.11 ——継承マッパーの汎用クラス図

Gateway クラスは、データプロパティ内にデータを保持する。データは適切なクエリーを発行することによって読み込むことができる。

```
class Gateway...

   public DataSet Data = new DataSet();
```

各具象マッパーは、データを保持するテーブルの名前を定義する必要がある。

```
class CricketerMapper...

      public override String TableName {
         get {return "Cricketers";}
      }
```

PlayerMapper は、各具象マッパーのフィールドを持つ。

```
class PlayerMapper...

      private BowlerMapper bmapper;
      private CricketerMapper cmapper;
      private FootballerMapper fmapper;
      public PlayerMapper (Gateway gateway) : base (gateway) {
         bmapper = new BowlerMapper(Gateway);
         cmapper = new CricketerMapper(Gateway);
         fmapper = new FootballerMapper(Gateway);
      }
```

12.9.3.1 ■ データベースからのオブジェクトの読み込み

各具象 Mapper クラスは、与えられたキーの任意のオブジェクトを返す Find メソッドを持っている。

```
class CricketerMapper...

      public Cricketer Find(long id) {
         return (Cricketer) AbstractFind(id);
      }
```

スーパークラスの抽象的な振る舞いは、ID に対する正しいデータベース行を見つけ出し、正しい型の新規ドメインオブジェクトを作成する。そして、それを読み込むために Load メソッドを使う。

```
class Mapper...

      public DomainObject AbstractFind(long id) {
         DataRow row = FindRow(id);
         if (row == null) return null;
```

```
            else {
                DomainObject result = CreateDomainObject();
                Load(result, row);
                return result;
            }
        }
        private DataRow FindRow(long id) {
            String filter = String.Format("id = {0}", id);
            DataRow[] results = table.Select(filter);
            if (results.Length == 0) return null;
            else return results[0];
        }
        protected abstract DomainObject CreateDomainObject();

class CricketerMapper...

        protected override DomainObject CreateDomainObject(){
            return new Cricketer();
        }
```

データベースからのデータ読み込みは、Load メソッドによって行われる（または、Mapper クラス、およびスーパークラスごとに持っている複数の Load メソッドによって行われる）。

```
class CricketerMapper...

        protected override void Load(DomainObject obj, DataRow row) {
            base.Load(obj,row);
            Cricketer cricketer = (Cricketer) obj;
            cricketer.battingAverage = (double)row["battingAverage"];
        }

class AbstractPlayerMapper...

        protected override void Load(DomainObject obj, DataRow row) {
            base.Load(obj, row);
            Player player = (Player) obj;
            player.name = (String)row["name"];

class Mapper...

        protected virtual void Load(DomainObject obj, DataRow row) {
            obj.Id = (int) row ["id"];
        }
```

これは、具象クラスに対するマッパーを使ってオブジェクトを検索するためのロジックである。あるいはスーパークラスに対するマッパーである PlayerMapper を使うこともできる。PlayerMapper は、オブジェクトがどのテーブルにあるかを見つけ出すためには不可欠である。データはデータセットとしてメモリ上にあるため、以下のような操作を行う。

```
class PlayerMapper...

      public Player Find (long key) {
         Player result;
         result = fmapper.Find(key);
         if (result != null) return result;
         result = bmapper.Find(key);
         if (result != null) return result;
         result = cmapper.Find(key);
         if (result != null) return result;
         return null;
      }
```

もちろん、このような方法が有効なのは、データがメモリ上にあるのが条件である。データベースに3回もアクセスしなければならないとすると（サブクラスが増えると回数も増える）、アクセスする速度は低下する。これは、具象テーブル間にジョインを作成する場合に役立ち、1回のデータベース呼び出しでデータにアクセスできるようになる。しかし、大規模なジョインはそれ自体に時間がかかるため、アプリケーションに対してはいくつかのベンチマークを実行して、何が機能し何が機能しないかを把握する必要がある。また、これは外部ジョインとなるため、構文の速度が低下するだけでなく移植性が失われ、暗号文のようになってしまうことも多い。

12.9.3.2 ■ オブジェクトの更新

Update メソッドは、マッパースーパークラスで定義できる。

```
class Mapper...

      public virtual void Update (DomainObject arg) {
         Save (arg, FindRow(arg.Id));
      }
```

読み込みと同様、それぞれの Mapper クラスの Save メソッドを順番に使用する。

```
class CricketerMapper...

      protected override void Save(DomainObject obj, DataRow row) {
         base.Save(obj, row);
         Cricketer cricketer = (Cricketer) obj;
         row["battingAverage"] = cricketer.battingAverage;
      }

class AbstractPlayerMapper...

      protected override void Save(DomainObject obj, DataRow row) {
         Player player = (Player) obj;
         row["name"] = player.name;
      }
```

PlayerMapper は正しい具象マッパーを検索し、更新呼び出しを委譲する必要がある。

```
class PlayerMapper...

      public override void Update (DomainObject obj) {
         MapperFor(obj).Update(obj);
      }
      private Mapper MapperFor(DomainObject obj) {
         if (obj is Footballer) return fmapper;
         if (obj is Bowler) return bmapper;
         if (obj is Cricketer) return cmapper;
         throw new Exception("No mapper available");
      }
```

12.9.3.3 ■ オブジェクトの挿入

挿入は、更新のバリエーションである。追加した振る舞いは、新しい行を作成することである。この動作はスーパークラスで実行できる。

```
class Mapper...

      public virtual long Insert (DomainObject arg) {
         DataRow row = table.NewRow();
         arg.Id = GetNextID();
         row["id"] = arg.Id;
         Save (arg, row);
         table.Rows.Add(row);
         return arg.Id;
      }
```

ここで再び、Player クラスは、適切なマッパーに委譲を行う。

```
class PlayerMapper...

      public override long Insert (DomainObject obj) {
         return MapperFor(obj).Insert(obj);
      }
```

12.9.3.4 ■ オブジェクトの削除

削除は、とてもシンプルである。前述の例と同様、メソッドはスーパークラス上に定義されている。

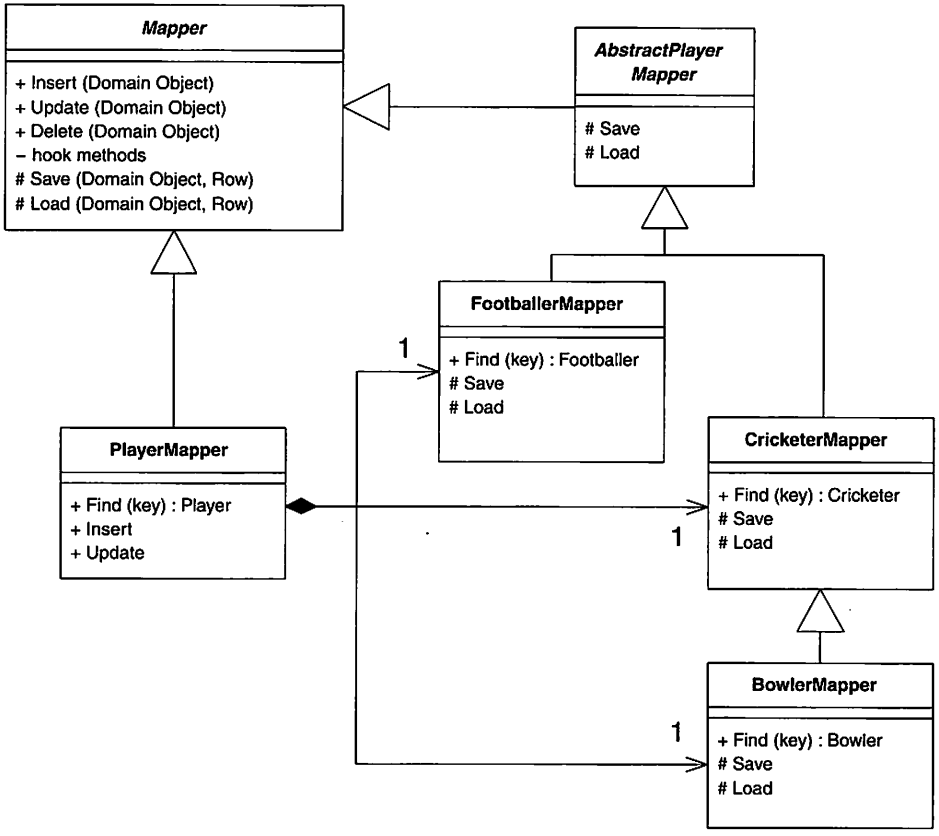
```
class Mapper...

      public virtual void Delete(DomainObject obj) {
         DataRow row = FindRow(obj.Id);
         row.Delete();
      }
```

また、委譲メソッドは、PlayerMapper 上に定義されている。

```
class PlayerMapper...

      public override void Delete (DomainObject obj) {
         MapperFor(obj).Delete(obj);
      }
```

12.10 継承マッパー

継承階層構造を処理するデータベースマッパーを組織化するための構造。



メモリ上のオブジェクト指向の継承階層構造からリレーショナルデータベースへマッピングする場合、データをデータベースに保存し、読み込むために必要なコード量は最小限に抑える必要がある。また同時に、抽象マッピングの振る舞いと具象マッピングの振る舞いを作成し、スーパークラスまたはサブクラスの保存または読み込みを可能にしたい。

このような振る舞いの詳細は、どのような継承マッピングスキーム（**シングルテーブル継承**、**クラステーブル継承**、および**具象テーブル継承**）を使用するかによって異なるが、汎用構造はいずれのスキームにおいても同じように機能する。

12.10.1 | 動作方法

階層構造によってマッパーを組織化することで、各ドメインクラスに対してドメインクラスのデータの保存や読み込みを行うマッパーを持たせることができるようになる。これでマッピングの変更が可能なポイントが確保される。この手法は、階層構造の具象オブジェクトをマッピングする具象マッパーには特に有効である。しかし、抽象クラスのマッパーが必要となる場合もある。それらは、実際には基本階層構造の外部にあるが、適切な具象マッパーへの委譲を行うマッパーで実装することができる。

一連の動作をわかりやすく説明するため、まずは具象マッパーから始めることにしよう。スケッチでの具象マッパーは、Footballer、Cricketer（クリケットプレイヤー）、およびBowler（ボウリングプレイヤー）の各Playerのマッパーである。基本的な振る舞いには、find、insert、update、およびdelete操作が含まれる。

findメソッドが具象クラスを返すため、findメソッドは具象サブクラス上で宣言される。これによって、BowlerMapper（ボウリングプレイヤーマッパー）クラスのfindメソッドは、抽象クラスではなく、Bowlerを返す。標準的なオブジェクト指向言語は、メソッドの宣言された戻り値タイプの変更を許可しないため、find操作を継承し、特定の戻り値型を宣言することはできない。もちろん抽象型を返すことは可能だが、クラスのダウンキャストをユーザに強制してしまうため絶対に回避すべきである（動的な型付けを行う言語では、このような問題は起こらない）。

findメソッドの基本的な振る舞いは、データベースの行を検索し、正しいタイプのオブジェクトをインスタンス化し（その判断はサブクラスによって行われる）、次にデータベースからデータとともにオブジェクトを読み込むことである。loadメソッドは、階層構造の各マッパーによって実装され、対応するドメインオブジェクトに対する振る舞いを読み込む。つまり、BowlerMapperのloadメソッドは、Bowlerクラスに固有のデータを読み込み、Cricketerクラス固有のデータを読み込むためにスーパークラスメソッドを呼び出す。そして、さらにこのメソッドがスーパークラスメソッドを呼び出すという順序で処理は進んでいく。

insertメソッドとupdateメソッドは、saveメソッドを使用する場合と同様に動作する。インタフェースは、スーパークラス上、**レイヤスーパータイプ**上に定義できる。insertメソッドは、新しい行を作成した後、saveフックメソッドを使ってドメインオブジェクトのデータを保存する。updateメソッドも、同様にsaveフックメソッドを使うが、データの保存だけを行う。これらのメソッドはloadフックメソッドと同じように動作し、各クラスは固有データを保存しスーパークラスのsaveメソッドを呼び出す。

このようなスキームによって、階層構造の特定の部分にとって必要な情報を保存する適切なマッパーの記述が容易になる。次のステップは、抽象クラスの読み込みと保存であり、例ではPlayerが対象である。最初は、スーパークラスマッパーに適切なメソッドを配置する方法だが、現実にはかなり複雑である。具象Mapperクラスは、抽象マッパーのinsertメ

ソッドと update メソッドを使うが、PlayerMapper の insert メソッドと update メソッドが具象マッパーを呼び出すにはこれらをオーバーライドしなければならない。その結果が汎用化と組織化の組み合わせの 1 つである。

私は、マッパーを 2 つのクラスに分離する方法を好んで用いる。抽象的な PlayerMapper は、データベースに対する特定の Player の読み込みおよび保存を行う。これは抽象クラスであり、振る舞いは具象マッパーオブジェクトでだけ使われる。分離した PlayerMapper クラスは、Player レベルの操作のためのインタフェースに使われる。PlayerMapper は、find メソッドを提供し、insert メソッドや update メソッドをオーバーライドする。これらが果たす役割とは、どの具象マッパーがタスクを処理するかを判断し委譲を行うことである。

このような広範な仕組みは継承マッピングのすべてのタイプに有効だが、詳細部分は異なっている。そのため、ケースに対する例のコードを提示することはできない。各継承マッピングパターン（**シングルテーブル継承**、**クラステーブル継承**、および**具象テーブル継承**）の中で例を探すことである。

12.10.2 | 使用するタイミング

汎用スキームは、継承ベースのデータベースマッピングに有効であるが、代替案として、具象マッパー間におけるスーパークラスマッピングコードのコピーや、Player のインタフェースを抽象的な Player Mapper クラスに含めるなどの操作が考えられる。前者は問題外であり、後者はありえなくはないが煩雑でわかりにくい PlayerMapper クラスになる。全体的に見て、このパターンの優れた代替案を考え出すことは困難だろう。

第13章

オブジェクトリレーショナルメタデータマッピングパターン

13.1 | メタデータマッピング

メタデータでのオブジェクトリレーショナルマッピングの詳細を保持する。

オブジェクトリレーショナルマッピングを処理するコードは、データベース内のフィールドとメモリ上のオブジェクト内のフィールドとを対応させる方法を記述している。コードの作成は、単調で繰り返しの多い作業になりがちである。**メタデータマッピング**を使うと、開発者はシンプルなテーブル形式でマッピングを定義した後、表書式を汎用コードで処理し、データの読み込み、挿入、および更新を実行できる。

13.1.1 | 動作方法

メタデータマッピングを使う際の重要な決定は、実行時のコードにどのようにメタデータの情報を組み込むかという点である。それには、主としてコード生成とリフレクティブプログラミングという2つの方法がある。

コード生成によってプログラムを書き込む場合、入力はメタデータになり、出力はマッピングを実行するクラスのソースコードとなる。クラスは手動で書き込まれたように見えるが、すべてビルドプロセスで生成されていて、コンパイルの直前に生成される。その結果のMapperクラスはサーバコードとともに配置される。

コード生成を使うときは、コード生成がビルドプロセスに完全に統合されていることを確

認すべきで、どのビルドスクリプトを使っても構わない。生成されたクラスは、手動で編集ができないためソースコードとして管理しない。

リフレクティブプログラムでは、オブジェクトに setName メソッドを呼び出した後、setName メソッド上で適切な引数を渡して invoke メソッドを実行する。複数のメソッド（とフィールド）をデータとして処理することによって、リフレクティブプログラムでは、フィールド名とメソッド名をメタデータファイルから読み込むことができ、これらのデータを使用してマッピングを実行できる。通常はリフレクションを勧めない。それは、リフレクションの速度が遅いことや、リフレクションを用いたコードはデバッグが困難であることが多いからである。たとえそうであったとしても、リフレクションはデータベースマッピングには適している。1 つのファイルから複数のフィールド名とメソッド名の読み込みにリフレクションの柔軟性を最大限に活用できる。

コード生成はリフレクションほど動的な手法ではない。それは、マッピングを変更するには、少なくともソフトウェアの一部で再コンパイルと再配置を常に行う必要があるからである。リフレクティブ手法の場合、マッピングデータファイルを変更するだけで、既存のクラスで新しいメタデータを使えるようになる。これは、特定の割り込みを実行して、メタデータの再読み込みを行うことで、実行時にも行うことができる。つまり、データベースやコードに変更を加える場合があるため、マッピングの変更は最小限に抑えなければならない。しかし最近の環境では、アプリケーションを部分的に再配置することが容易にできる。

リフレクティブプログラミングでは、速度の面で不都合が生じることが多い。ただし、この問題は使用環境によって大きく異なり、ある環境ではリフレクティブ呼び出しの速度が大幅に遅くなる場合もある。ただし、SQL 呼び出しのコンテキストでリフレクションが実行中であり、リモート呼び出しの速度が遅いことを考えると、速度の低下は必ずしも大きな問題ではない場合がある。パフォーマンスの問題と同様に、環境内を測定して、速度がどの程度重要であるかを確認する必要がある。

どちらの手法でもデバッグがやや困難である。これらの手法を比較するには、コード生成とリフレクティブコードに開発者がどの程度まで習熟しているかによって大きく左右される。コード生成の方が明示的であり、デバッガで実行されている内容を確認できる。このため、私はリフレクションよりコード生成の方を好んで使う。また一般的に、コード生成は（私が思うに）経験の浅い開発者でも容易に使うことができる。

メタデータは、別のファイル形式で保存される場合がとても多い。最近は XML が頻繁に使用されるようになったが、それは XML が階層構造になっていて、ユーザが単独のパーサやその他解析ツールを使う必要がないからである。読み込みステップで、このメタデータを取得しプログラミング言語構造に変換する。そのプログラミング言語によってコード生成出力かリフレクティブマッピングのいずれかが行われる。

シンプルなケースでは、外部ファイル形式を省略し、ソースコード内に直接メタデータを

表現する。解析は不要になるが、メタデータの編集は困難になる。

他にもデータベース自体でマッピング情報を保持する方法がある。この方法ではデータベースと内部データが結合される。データベーススキーマが変更されてもマッピング情報は正しく表示される。

メタデータ情報を保持する方法を選択する場合、アクセスと解析のパフォーマンスを考える必要はない。コード生成を使う場合、アクセスと解析は設計時に行われ、実行時には行われない。リフレクティブプログラミングを使う場合、実行時にアクセスと解析が行われるが、システム起動時に一度だけ行われる。それから、ユーザはメモリ上の表現を保持することができる。

メタデータをどこまで複雑にするかが最大の決定事項の1つである。一般的なリレーショナルマッピングの問題に直面している場合、メタデータに保持する多くの異なる要因が存在する。しかし多くのプロジェクトでは、完全に包括的なスキームより少ないスキームで管理することができるため、メタデータはよりシンプルなものになる。つまり、メタデータ駆動型のソフトウェアでは新しい機能を簡単に追加できるので、この方法は需要に応じた設計の拡張に有効である。

メタデータの問題の1つとして挙げられるのは、通常シンプルなメタデータスキームは存続期間中90%は問題なく動作するが、特定のケースではとても扱いにくい場合が多いことである。このような例外的なケースに対処するため、通常はメタデータを複雑にしなければならない場合が多い。この他に役立つ方法は、サブクラスで汎用コードをオーバーライドすることである。このサブクラスでは特定のコードが手動で作成される。このような特定のケースのサブクラスは、生成されたコードかリフレクティブルーチンのいずれかのサブクラスである。これは特別なケースであり、オーバーライドをサポートするためにどのような処理を行うかを一般的な用語で説明するのは容易ではない。臨機応変に対処してほしい。オーバーライドが必要な場合、特別なケースでオーバーライドを行う必要があり、実際にオーバーライドした1つのメソッドを分離するために、生成されたコードあるいはリフレクティブコードを変更する必要がある。

13.1.2 | 使用するタイミング

メタデータマッピングを使用すると、データベースマッピング処理に必要な作業負荷を大幅に削減できる。ただし、**メタデータマッピング**のフレームワークを使用するにはいくつかの設定が必要になる。また、**メタデータマッピング**を使用すると多くの場合は処理が容易になるが、例外的に、メタデータが実際には複雑になってしまうこともある。

当然のことながら、商用のオブジェクトリレーショナルマッピングツールでは**メタデータマッピング**を採用している。すなわち、洗練された**メタデータマッピング**を生成する製品を

販売するためには、それなりの労力を常に費やさなければならないということだ。

独自のシステムを構築する場合、トレードオフについて考慮する必要がある。新しいマッピングを追加するため、手動で作成したコードを使用する場合と**メタデータマッピング**を使用する場合を比較する。リフレクションを使用する場合、リフレクションがパフォーマンスに与える影響を検討する必要がある。リフレクションによって、パフォーマンスが低下することがあれば、向上する場合もあるからである。また測定方法自体に問題がある場合もある。

共通的な振る舞いの処理すべてに最適な**レイヤスーパータイプ**を作成することで、手動コーディングの追加作業を大幅に削減できるようになる。**レイヤスーパータイプ**を作成するためには、各マッピングにいくつかのフックルーチンを追加する必要がある。これによって、通常は**メタデータマッピング**の数が大幅に減少する。

メタデータマッピングは、特に自動ツールを使用している場合にリファクタリングを妨げるときがある。プライベートフィールドの名前を変更すると、あるアプリケーションが突然壊れることがある。自動リファクタリングツールでも、マッピングのXMLデータファイル内に隠ぺいされたフィールド名を検出することはできない。

検索メカニズムによって自動ツールを使用できるため、コードを生成する手間が若干省ける。それでもコードを再生成すると、必ず自動更新される。ツールが問題を警告することもあるが、メタデータを変更するかどうかはユーザが決定することである。リフレクションを使用すると、警告が表示されることすらない。

しかし、**メタデータマッピング**を使用することで容易にデータベースリファクタリングを行える理由は、メタデータがデータベーススキーマのインタフェースのステートメントとして機能するからである。したがって、**メタデータマッピング**を変更することで、データベースへの変更を行うことができる。

13.1.3 ｜例：メタデータとリフレクションの使用（Java）

本書の例の多くでは、理解しやすくするため明示的なコードを使用している。ただし、これによってとても単調なプログラミングとなり、また単調なプログラミングということは、どこかに異常があることを示している。メタデータを使用することによって、多くの単調なプログラミングを取り除くことができる。

13.1.3.1 ■ メタデータの保持

メタデータに関して最初に決定することは、メタデータをどのように保持するかということである。ここでは、2つのクラスでメタデータを保持している。データマッピングとは、1つのクラスを1つのテーブルにマッピングすることである。これはシンプルなマッピングであるが、説明する場合には役立つ。

```
class DataMap...

   private Class domainClass;
   private String tableName;
   private List columnMaps = new ArrayList();
```

データマッピングは、テーブル内の列をフィールドにマッピングする列マッピングのコレクションを含んでいる。

```
class ColumnMap...

   private String columnName;
   private String fieldName;
   private Field field;
   private DataMap dataMap;
```

これはあまり洗練されたマッピングとは言えない。私はデフォルトのJava型マッピングだけを使用しているが、これはフィールドと列の間に型の変換がないことを意味している。また、テーブルとクラスの関係も1対1に設定している。

上記の構造でマッピングを保持する。次の決定は、どのようにしてメタデータを配置するかである。この例では、私はJavaコードを使用して特定のMapperクラスにメタデータを配置する。少し奇妙に思うかもしれないが、Javaコードを使用することで、たとえば、コードの繰り返しを回避できるなど、メタデータのメリットを最大限に活用できる。

```
class PersonMapper...

   protected void loadDataMap(){
      dataMap = new DataMap (Person.class, "people");
      dataMap.addColumn ("lastname", "varchar", "lastName");
      dataMap.addColumn ("firstname", "varchar", "firstName");
      dataMap.addColumn ("number_of_dependents", "int",
         "numberOfDependents");
   }
```

列のマッパーを構築するときに、私はフィールドへのリンクを構築する。厳密にいうと、これは最適化であり、フィールドを計算する必要がない場合がある。しかし、最適化することにより、小型のノートパソコンでのその後のアクセスが大幅に削減される。

```
class ColumnMap...

   public ColumnMap(String columnName, String fieldName, DataMap dataMap) {
      this.columnName = columnName;
      this.fieldName = fieldName;
      this.dataMap = dataMap;
      initField();
   }
   private void initField() {
      try {
         field = dataMap.getDomainClass().getDeclaredField(getFieldName());
         field.setAccessible(true);
      } catch (Exception e) {
         throw new ApplicationException ("unable to set up
            field: " + fieldName, e);
      }
   }
```

XML ファイルから、またはメタデータデータベースからマッピングを読み込ませるためのルーチンを記述する方法を理解することはそれほど難しくはない。難しくはないが敢えてここでは省略するので、各自で習得してほしい。

現時点では複数のマッピングが定義されていて、これらのマッピングを使用することができる。メタデータ手法のメリットは、実際に操作するコードすべてがスーパークラスの中にある点である。したがって、明示的に記述した場合は、マッピングコードを書く必要はない。

13.1.3.2 ■ ID による検索

ID メソッドによる検索メソッドから始める。

```
class Mapper...

   public Object findObject (Long key) {
      if (uow.isLoaded(key)) return uow.getObject(key);
      String sql = "SELECT" + dataMap.columnList() + " FROM " +
         dataMap.getTableName() + " WHERE ID = ?";
      PreparedStatement stmt = null;
      ResultSet rs = null;
      DomainObject result = null;
      try {
         stmt = DB.prepare(sql);
         stmt.setLong(1, key.longValue());
```

```
            rs = stmt.executeQuery();
            rs.next();
            result = load(rs);
        } catch (Exception e) {throw new ApplicationException (e);
        } finally {DB.cleanUp(stmt, rs);
        }
    return result;
    }
    private UnitOfWork uow;
    protected DataMap dataMap;

class DataMap...

    public String columnList() {
        StringBuffer result = new StringBuffer(" ID");
        for (Iterator it = columnMaps.iterator(); it.hasNext();) {
            result.append(",");
            ColumnMap columnMap = (ColumnMap)it.next();
            result.append(columnMap.getColumnName());
        }
        return result.toString();
    }
    public String getTableName() {
        return tableName;
    }
```

ここでの select は他の例よりも動的に構築されているが、このようにデータベースセッションが select を適切にキャッシュ内に格納できるようにしておくことは有効である。必要な場合には、データマッピングの存続期間中に列を更新する呼び出しがないため、構築中に列リストを計算してキャッシュに格納することができる。この例の場合、データベースセッションを処理するために**ユニットオブワーク**を使用している。

本書の例に共通することであるが、私は読み出しと検索を分離している。したがって、同じ load メソッドを使用して別の find メソッドから検索できる。

```
class Mapper...

    public DomainObject load(ResultSet rs)
            throwsInstantiationException, IllegalAccessException, SQLException
    {
        Long key = new Long(rs.getLong("ID"));
        if (uow.isLoaded(key)) return uow.getObject(key);
        DomainObject result = (DomainObject)
```

```
            dataMap.getDomainClass().newInstance();
        result.setID(key);
        uow.registerClean(result);
        loadFields(rs, result);
        return result;
    }
    private void loadFields(ResultSet rs, DomainObject result)
        throws SQLException {
        for (Iterator it = dataMap.getColumns(); it.hasNext();) {
            ColumnMap columnMap = (ColumnMap)it.next();
            Object columnValue = rs.getObject(columnMap.getColumnName());
            columnMap.setField(result, columnValue);
        }
    }

class ColumnMap...

    public void setField(Object result, Object columnValue) {
        try {
            field.set(result, columnValue);
        } catch (Exception e) { throw new ApplicationException
            ("Error in setting " + fieldName, e);
        }
    }
```

これは典型的なリフレクティブプログラムである。個々の列マッピングを介し、その列マッピングを使用して、ドメインオブジェクト内のフィールドを読み込む。より複雑な場合にどのように拡張するかを示すために、loadFields メソッドを分離する。シンプルなメタデータでは表現できないクラスとテーブルがあった場合でも、サブクラスマッパー内の loadFields をオーバーライドするだけで、任意に複雑なコード内に置くことができる。これはメタデータを使用した一般的な技法である。変則的なケースではフックを使用してオーバーライドする。通常、サブクラスを使用して変則的なケースをオーバーライドする方が、少数の稀なケースに適用される洗練されたメタデータを構築するより容易である。

もちろん、サブクラスがある場合は、サブクラスを使用してダウンキャストを回避したほうが賢明である。

```
class PersonMapper...

    public Person find(Long key) {
        return (Person) findObject(key);
    }
```

13.1.3.3 ■ データベースへの書き込み

更新には、1つの更新ルーチンを使用する。

```
class Mapper...

   public void update (DomainObject obj) {
      String sql = "UPDATE " + dataMap.getTableName() +
         dataMap.updateList() + " WHERE ID = ?";
      PreparedStatement stmt = null;
      try {
         stmt = DB.prepare(sql);
         int argCount = 1;
         for (Iterator it = dataMap.getColumns(); it.hasNext();) {
            ColumnMap col = (ColumnMap) it.next();
            stmt.setObject(argCount++, col.getValue(obj));
         }
         stmt.setLong(argCount, obj.getID().longValue());
         stmt.executeUpdate();
      } catch (SQLException e) {throw new ApplicationException (e);
      } finally {DB.cleanUp(stmt);
      }
   }

class DataMap...

   public String updateList() {
      StringBuffer result = new StringBuffer(" SET ");
      for (Iterator it = columnMaps.iterator(); it.hasNext();) {
         ColumnMap columnMap = (ColumnMap)it.next();
         result.append(columnMap.getColumnName());
         result.append("=?,");
      }
      result.setLength(result.length() - 1);
      return result.toString();
   }
   public Iterator getColumns() {
      return Collections.unmodifiableCollection(columnMaps).iterator();
   }

class ColumnMap...

   public Object getValue (Object subject) {
      try {
```

```
            return field.get(subject);
        } catch (Exception e) {
            throw new ApplicationException (e);
        }
    }
```

挿入には、同様のスキームを使用する。

```
class Mapper...

    public Long insert (DomainObject obj) {
        String sql = "INSERT INTO " + dataMap.getTableName() + "
            VALUES (?" + dataMap.insertList() + ")";
        PreparedStatement stmt = null;
        try {
            stmt = DB.prepare(sql);
            stmt.setObject(1, obj.getID());
            int argCount = 2;
            for (Iterator it = dataMap.getColumns(); it.hasNext();) {
                ColumnMap col = (ColumnMap) it.next();
                stmt.setObject(argCount++, col.getValue(obj));
            }
            stmt.executeUpdate();
        } catch (SQLException e) {throw new ApplicationException (e);
        } finally {DB.cleanUp(stmt);
        }
        return obj.getID();
    }

class DataMap...

    public String insertList() {
        StringBuffer result = new StringBuffer();
        for (int i = 0; i < columnMaps.size(); i++) {
            result.append(",");
            result.append("?");
        }
        return result.toString();
    }
```

13.1.3.4 ■ 複数のオブジェクトの検索

1つのクエリーを発行して複数のオブジェクトを取得するための手法がいくつかある。汎

用マッパー上で汎用クエリー機能が必要な場合、SQL の where 句を引数とするクエリーを発行することができる。

```
class Mapper...

   public Set findObjectsWhere (String whereClause) {
      String sql = "SELECT" + dataMap.columnList() + " FROM " +
         dataMap.getTableName() + " WHERE " + whereClause;
      PreparedStatement stmt = null;
      ResultSet rs = null;
      Set result = new HashSet();
      try {
         stmt = DB.prepare(sql);
         rs = stmt.executeQuery();
         result = loadAll(rs);
      } catch (Exception e) {
         throw new ApplicationException (e);
      } finally {DB.cleanUp(stmt, rs);
      }
   return result;
   }
   public Set loadAll(ResultSet rs) throws SQLException,
      InstantiationException, IllegalAccessException {
      Set result = new HashSet();
      while (rs.next()) {
         DomainObject newObj = (DomainObject)
            dataMap.getDomainClass().newInstance();
         newObj = load (rs);
         result.add(newObj);
      }
      return result;
   }
```

別の方法として、マッパーサブタイプ上で特定のケースの find メソッドを使用することもできる。

```
class PersonMapper...

   public Set findLastNamesLike (String pattern) {
      String sql =
         "SELECT" + dataMap.columnList() +
         " FROM " + dataMap.getTableName() +
```

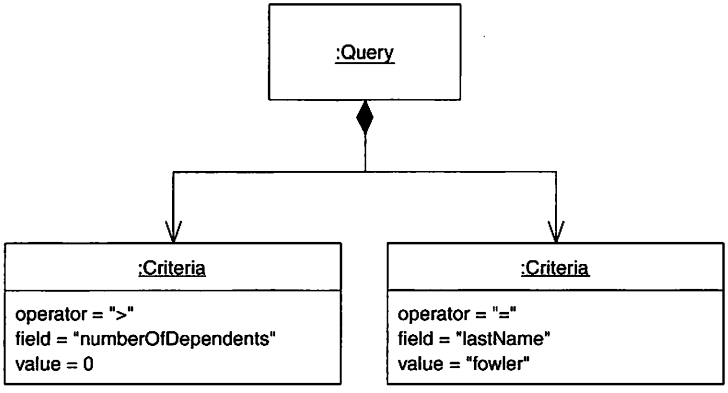
```
            " WHERE UPPER(lastName) LIKE UPPER(?)";
        PreparedStatement stmt = null;
        ResultSet rs = null;
        try {
            stmt = DB.prepare(sql);
            stmt.setString(1, pattern);
            rs = stmt.executeQuery();
            return loadAll(rs);
        } catch (Exception e) {throw new ApplicationException (e);
        } finally {DB.cleanUp(stmt, rs);
        }
    }
```

さらに一般的な選択肢として、**クエリーオブジェクト**がある。

つまり、メタデータ手法の大きなメリットは、loadMap メソッドといくつかの特化された find メソッドを追加することで、データマッピングに新しいテーブルとクラスを追加できることである。

13.2 | クエリーオブジェクト

データベースクエリーとして機能するオブジェクト。



SQL は複雑な言語になることがあるため、SQL に習熟している開発者は多くない。さらに、クエリーを形成するためにはデータベーススキーマの構造を知っている必要がある。パラメータ化されたメソッド内部に SQL を隠ぺいする特化された find メソッドを作成することによって、SQL を使用しないこともできるが、その場合には、よりアドホックなクエリーを作成することは難しい。また、データベーススキーマを変更する場合、SQL 文内で

重複してしまうこともある。

クエリーオブジェクトはインタプリタプログラム[Gang of Four]である。つまり、**クエリーオブジェクト**自体がSQLクエリーとして機能することのできるオブジェクト構造になっている。テーブルと列ではなく、クラスとフィールドを参照することによってSQLクエリーを作成することができる。上記のとおり、クエリーの作成者はデータベーススキーマに関わる必要がなく、スキーマへの変更を1箇所に格納することができる。

13.2.1 | 動作方法

クエリーオブジェクトとはインタプリタパターンのアプリケーションであり、SQLクエリーとして機能する。**クエリーオブジェクト**の第一の役割は、クライアントがさまざまな種類のクエリーを作成し、そのオブジェクト構造を適切なSQL文字列に変換できるようにすることである。

クエリーとして機能するためには、柔軟性のある**クエリーオブジェクト**が必要である。しかし、多くの場合、アプリケーションはSQLのすべての機能よりもはるかに少ない機能を使用して動作している。この場合には、**クエリーオブジェクト**はよりシンプルになる。シンプルになった**クエリーオブジェクト**がすべての機能を実行できるわけではないが、特定のニーズを満たすことはできる。さらに、通常、最初から完全な機能を持つ**クエリーオブジェクト**を作成するより、追加機能が必要になったときに拡張する方が作業負荷が少ない。結果として、現在必要な最低限の機能を持つ**クエリーオブジェクト**を作成し、必要に応じて拡張するべきである。

クエリーオブジェクトの一般的な機能は、データベーススキーマではなくメモリ上のオブジェクトの言語で、クエリーを表現できることである。つまり、テーブル名や列名を使用せずに、オブジェクト名とフィールド名を使用できる。オブジェクトとデータベースが同じ構造かどうかは重要ではないが、この2つがさまざまな方法で組み合わされているときにはとても役に立つ場合がある。ビューをこのように変更するためには、**クエリーオブジェクト**は、どのようにしてデータベース構造をオブジェクト構造にマッピングするか、すなわち実際に**メタデータマッピング**を必要としている機能を知る必要がある。

複数のデータベースの場合、クエリーが動作しているデータベースに応じて、異なったSQLを生成するように**クエリーオブジェクト**を設計できる。最もシンプルなレベルでも、**クエリーオブジェクト**はSQL構文の面倒な相違を考慮できる。大規模なレベルでは、**クエリーオブジェクト**は多様なマッピングを使用して、異なるデータベーススキームに格納された同じクラスを処理することができる。

クエリーオブジェクトの特に洗練された使用法は、データベースに対する重複したクエリーを取り除くことである。セッションの初期に同じクエリーを発行したことがわかる場合、

そのクエリーを使用して**一意マッピング**からオブジェクトを選択してデータベースへの移行を回避できる。洗練された手法では、あるクエリーが以前のクエリーの特定のケースかどうかを検出できる。以前のクエリーとは、たとえば以前と同一だが、AND と連結した句が追加されたようなクエリーである。

この洗練された機能を実現する正確な方法は本書の内容の範囲を超えているが、O/R マッピングツールがこの種類の機能を提供している。

クエリーオブジェクトは、ドメインオブジェクトの例によってクエリーを特定することができる。したがって、Person オブジェクトの姓は Fowler と設定され、その他のすべての属性は null（設定なし）に設定されている場合もある。インタプリタ型**クエリーオブジェクト**のような処理を行ったサンプルによって、Person オブジェクトをクエリーとして動作させることができるようになる。これによって、データベース内で姓が Fowler というすべての Person が返される。この方法はとてもシンプルで使用しやすいが、複雑なクエリーは使用できない。

13.2.2 ｜使用するタイミング

クエリーオブジェクトは組み立てが必要な洗練されたパターンであるため、手動で作成したデータソースレイヤがある場合、ほとんどのプロジェクトでは使用しない。**ドメインモデル**や**データマッパー**を使っている場合にだけ**クエリーオブジェクト**が必要になり、さらに**クエリーオブジェクト**を正しく使用するためには**メタデータマッピング**も必要になる。

その場合でも、開発者が SQL に慣れていれば**クエリーオブジェクト**は必ずしも必要ではない。特化された find メソッドの裏側にデータベーススキーマの詳細を隠ぺいできる。

クエリーオブジェクトのメリットによって、カプセル化されたデータベーススキーマの保持、複数のデータベースのサポート、複数のスキーマのサポート、および重複したクエリーを回避するための最適化などの、より洗練されたニーズに対応できる。特に洗練されたデータソースチームとのプロジェクトでは、独自にこれらの機能を作成することを望むこともある。しかし、商用のツールによって**クエリーオブジェクト**を使用する場合が多い。私は、ツールの購入にはそれなりの価値が常にあると思っている。

ここまで読み進んだ読者は、限定された機能しかない**クエリーオブジェクト**でも十分にニーズを満たすことができ、機能満載のバージョンを必要としないプロジェクトなら難なく構築できることに気付くだろう。実際に使用しない機能を外すことが秘訣である。

13.2.3 ｜参考文献

インタプリタの解説としては、[Alpert et al.]に記載されている**クエリーオブジェクト**の

サンプルを挙げることができる。またクエリーオブジェクトは、[Evans and Fowler]や[Evans]の仕様パターンと密接に関連している。

13.2.4 ｜例：シンプルなクエリーオブジェクト（Java）

クエリーオブジェクトのシンプルな例である。実用的なレベルとは言えないが、クエリーオブジェクトの概要を理解するには十分である。「AND'ed」（やや専門的な言語であり、初歩的な述部の連結処理を実行する）で連結された一連の判断基準に従って、1つのテーブルにクエリーを発行できる。

クエリーオブジェクトは、テーブル構造の言語ではなく、ドメインオブジェクトの言語を使って作成される。したがって、クエリーは、そのクエリーを発行するクラス、および複数のwhere句に対応する判断基準のコレクションを把握している。

```
class QueryObject...

   private Class klass;
   private List criteria = new ArrayList();
```

シンプルな判断基準はフィールド、値、およびSQL演算子を取得し、これらを比較する。

```
class Criteria...

   private String sqlOperator;
   protected String field;
   protected Object value;
```

的確な判断基準を容易に作成できるように、適切な作成メソッドを提供しよう。

```
class Criteria...

   public static Criteria greaterThan(String fieldName, int value) {
      return Criteria.greaterThan(fieldName, new Integer(value));
   }
   public static Criteria greaterThan(String fieldName, Object value) {
      return new Criteria(" > ", fieldName, value);
   }
   private Criteria(String sql, String field, Object value) {
      this.sqlOperator = sql;
      this.field = field;
```

```
        this.value = value;
    }
```

以下のようなクエリーを作成し、扶養家族を持つすべてのPersonを検索できる。

```
class Criteria...

        QueryObject query = new QueryObject(Person.class);
        query.addCriteria(Criteria.greaterThan("numberOfDependents", 0));
```

ここで、私が使用するPersonオブジェクトは、以下のとおりである。

```
class Person...

    private String lastName;
    private String firstName;
    private int numberOfDependents;
```

扶養家族を持つPersonを検索するには、Personを検索するクエリーを作成して、ある判断基準を加える。

```
        QueryObject query = new QueryObject(Person.class);
        query.addCriteria(Criteria.greaterThan("numberOfDependents", 0));
```

クエリーを記述するにはこれで十分である。次にクエリーをSQLのselect文に変換して実行する必要がある。この場合、Mapperクラスが、where句の文字列に基づいてオブジェクトを検索するメソッドをサポートすると仮定する。

```
class QueryObject...

    public Set execute(UnitOfWork uow) {
        this.uow = uow;
        return uow.getMapper(klass).findObjectsWhere(generateWhereClause());
    }

class Mapper...

    public Set findObjectsWhere (String whereClause) {
        String sql = "SELECT" + dataMap.columnList() + " FROM " +
            dataMap.getTableName() + " WHERE " + whereClause;
```

```
        PreparedStatement stmt = null;
        ResultSet rs = null;
        Set result = new HashSet();
        try {
            stmt = DB.prepare(sql);
            rs = stmt.executeQuery();
            result = loadAll(rs);
        } catch (Exception e) {
            throw new ApplicationException (e);
        } finally {DB.cleanUp(stmt, rs);
        }
    return result;
    }
```

クラスによってインデックス付けされたマッパーを保持する**ユニットオブワーク**と、**メタデータマッピング**を使うマッパーを使っている。コードは、本項で同じコードの繰り返しを省略するための**メタデータマッピング**の例と同じものである。

where 句を生成するには、クエリーが判断基準全体を繰り返して各クエリーを出力した後、AND と結合して使う。

```
class QueryObject...

    private String generateWhereClause() {
        StringBuffer result = new StringBuffer();
        for (Iterator it = criteria.iterator(); it.hasNext();) {
            Criteria c = (Criteria)it.next();
            if (result.length() != 0)
                result.append(" AND ");
            result.append(c.generateSql(uow.getMapper(klass).getDataMap()));
        }
        return result.toString();
    }

class Criteria...

    public String generateSql(DataMap dataMap) {
        return dataMap.getColumnForField(field) + sqlOperator + value;
    }

class DataMap...

    public String getColumnForField (String fieldName) {
```

```
        for (Iterator it = getColumns(); it.hasNext();) {
            ColumnMap columnMap = (ColumnMap)it.next();
            if (columnMap.getFieldName().equals(fieldName)) return
                columnMap.getColumnName();
        }
        throw new ApplicationException ("Unable to find column for
            " + fieldName);
    }
```

シンプルなSQL演算子を持つ判断基準と同様に、多くのことを実行する複雑な判断基準クラスを作成することができる。たとえば、姓がFで始まるPersonを検索するような大文字と小文字を区別しないパターンマッチクエリーがあるとする。この場合、Fで始まる姓を持ち、扶養家族がいるPersonを対象としたクエリーオブジェクトを作成することができる。

```
        QueryObject query = new QueryObject(Person.class);
        query.addCriteria(Criteria.greaterThan("numberOfDependents", 0));
        query.addCriteria(Criteria.matches("lastName", "f%"));
```

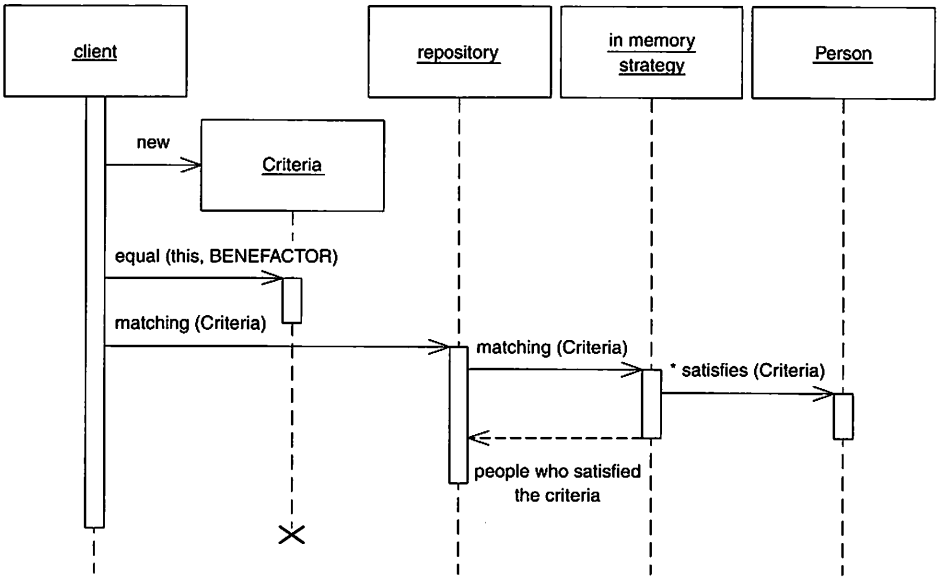
where文でより複雑な句を作成するCriteriaクラスを使う。

```
class Criteria...

    public static Criteria matches(String fieldName, String pattern){
        return new MatchCriteria(fieldName, pattern);
    }

class MatchCriteria extends Criteria...

    public String generateSql(DataMap dataMap) {
        return "UPPER(" + dataMap.getColumnForField(field) + ")
            LIKE UPPER(' " + value + " ')";
    }
```

13.3 | リポジトリ

(by Edward Hieatt, Rob Mee)

ドメインオブジェクトにアクセスするためのコレクション型インタフェースを使って、ドメインとデータマッピングレイヤとを仲介する。



複雑なドメインモデルを持つシステムは、**データマッパー**のレイヤなどから複数のメリットを得る場合が多い。この場合、**データマッパー**は、ドメインオブジェクトをデータベースアクセスコードから分離する。このようなシステムで、クエリーを構成するコードが集中したマッピングレイヤの上に、抽象化した他のレイヤを構築することは価値がある。この構築が重要なのは、多くのドメインクラスや複雑なクエリーがある場合である。特に抽象レイヤの追加時には、同じクエリーロジックの繰り返しを最小限に抑えることに役立つ。

リポジトリは、ドメインとデータマッピングレイヤの仲介をし、メモリ上のドメインオブジェクトコレクションの役割を果たす。クライアントオブジェクトは、宣言型でクエリーの仕様を構築し、仕様を満たすため**リポジトリ**に送る。オブジェクトは、シンプルなオブジェクトコレクションから削除できるのと同様に、**リポジトリ**に追加することも削除することもできる。また、**リポジトリ**によってカプセル化されたマッピングコードは裏側で操作を行う。理論上**リポジトリ**は、データストアにある一連のオブジェクトおよびオブジェクト上で実行された操作をカプセル化し、永続的なレイヤよりオブジェクト指向の強いビューを提供する。

リポジトリは、明確な目的の分離も、ドメインとデータマッピングレイヤ間での一方向の依存性もサポートしている。

13.3.1 | 動作方法

リポジトリは、本書に記述されている他のパターンを使うことのできる洗練されたパターンの1つである。リポジトリは、オブジェクト指向データベースの一部のように思われていて、その意味ではクエリーオブジェクトに類似している。開発チームは、オブジェクトリレーショナルマッピングツールでクエリーオブジェクトを見つける方が、独自に作成するよりも多い。ただし、すでにチームがクエリーオブジェクトを作成済みである場合、リポジトリ機能を追加するのは容易である。クエリーオブジェクトと連結して使うと、リポジトリによってあまり労力を使うことなく、オブジェクトリレーショナルマッピングレイヤが大幅に使いやすくなる。

裏側での処理だが、リポジトリはシンプルなインタフェースとして機能し、クライアントは、クエリーから返したいオブジェクトの特性を指定するCriteria（判断基準）オブジェクトを作成する。たとえば、Personオブジェクトを名前で検索するには、まずCriteriaオブジェクトを作成して、個別の判断基準を設定する。`criteria.equals(Person.LAST_NAME, "Fowler")`、および`criteria.like(Person.FIRST_NAME, "M")`、次に`repository.matching(criteria)`を呼び出して、姓がFowlerで名がMで始まるPersonを示すドメインオブジェクトのリストを返す。`matching (criteria)`に類似した便利なメソッド類を抽象リポジトリで定義できる。たとえば、一致するものが1つだけ予想される場合、`soleMatch(criteria)`は、コレクションではなく検索条件に適合したオブジェクトを返すことがある。他の共通のメソッドには、`soleMatch`を使って簡単に実装できる`byObjectId(id)`が含まれている。

リポジトリを使うコードは、シンプルなドメインオブジェクトのメモリ上のコレクションで表示される。リポジトリ内に直接格納されていないドメインオブジェクトは、クライアントコードからは見ることができない。もちろん、リポジトリを使うコードを認識しておく必要があるのは、可視的なオブジェクトコレクションが、数十万ものレコードを持つProductテーブルにマッピングされることが十分考えられるからである。カタログシステムの`ProductRepository`上で`all()`を呼び出すことはあまり勧められない。

リポジトリは、データマッパークラス上の特化されたfindメソッドを、オブジェクトセレクション[Evans and Fowler]の仕様ベースの手法に置き換える。クエリーオブジェクトを直接使う方法と比較すると、クエリーオブジェクトではクライアントコードがCriteriaオブジェクト（仕様パターンの簡単な例）を作成し、クエリーオブジェクトに直接`add()`を実行しクエリーを発行する。リポジトリでは、クライアントコードが判断基準を作成してリポジトリに渡し、一致するオブジェクトの判断基準を選択するように要求する。クライアント

コードの観点から見ると、クエリーの「発行」という考えはない。むしろ、クエリーの仕様を「満たす」ことでオブジェクトの選択を行う。この違いが、**リポジトリ**とオブジェクトとの相互作用を明示しているのである。理論上では、この処理が**リポジトリ**のほぼすべての機能なのである。

隠れた所で**リポジトリ**は、**メタデータマッピング**と**クエリーオブジェクト**を組み合わせて、自動的にその判断基準でSQLコードを生成する。判断基準がクエリーへの追加方法を知っているかどうか、**クエリーオブジェクト**がCriteriaオブジェクトを組み込む方法を知っているかどうか、または**メタデータマッピング**自体が相互作用を制御するかどうかということが、実装時の詳細項目となる。

リポジトリのオブジェクトソースは、決してリレーショナルデータベースではない。これは、特化したストラテジーオブジェクトを介したデータマッピングコンポーネントの再配置にとても役立つ。したがって、**リポジトリ**は、複数のデータベーススキーマやドメインオブジェクトのソースを持つシステムで特に有効であり、メモリ上のオブジェクトを使う速度の最適化のテスト時においても有効である。

リポジトリは、広範囲にクエリーを発行するコードであり、読みやすく分かりやすい優れたメカニズムである。たとえば、多くのクエリーページを表示する機能があるブラウザーベースのシステムで、`HttpRequest`オブジェクトを処理してクエリー結果を表示するには、明確なメカニズムが必要になる。リクエストに対するハンドラコードは、自動ではなくても`HttpRequest`をCriteriaオブジェクトに容易に変換できる。判断基準を**リポジトリ**に送るために必要なのは、1、2行の追加コードだけである。

13.3.2 | 使用するタイミング

多くのドメインオブジェクトタイプとクエリーを持つ大規模システムにおいて、**リポジトリ**は、すべてのクエリーを扱うのに必要なコードの量を減らす。リポジトリは、（本書の例にあるCriteriaオブジェクトの形式において）仕様パターンを促進する。これは、純粋なオブジェクト指向の方法で、実行するクエリーをカプセル化する。状況ごとに、**クエリーオブジェクト**を設定するコードはすべて削除できる。クライアントは、SQLを考える必要は一切なく純粋にオブジェクト指向で記述する。

複数のデータソースの環境では、**リポジトリ**を使うことになる。たとえばパフォーマンス向上のためにメモリ上で一連のユニットテストを実行したいために、シンプルなメモリ上のデータ格納を使いたくなる場合がある。データベースアクセスがない場合、長時間にわたる一連のテストの速度は格段に速くなる。ドメインオブジェクトを作成してそれをコレクションの中に挿入する方法の方が、テスト前にドメインオブジェクトをデータベースに保存し、テスト後に削除するよりもユニットテストを容易に作成できる。

また、アプリケーションが正常に稼動しているとき、常に任意のドメインオブジェクトがメモリ内にある点も考えるべきである。このような例は、（ユーザが変更できない）不変なドメインオブジェクトである。これは、ドメインオブジェクトが一度メモリ内に格納されると、そこに留まり、再度クエリーは実行されない。本章で後述するが、**リポジトリ**パターンへのシンプルな拡張によって、状況に応じて異なるクエリーストラテジーを取り入れることもある。

また、**リポジトリ**が有用な例として、データフィードがドメインオブジェクトのソースとして使われている場合が挙げられる。たとえば、SOAP を使ってインターネットを介したXML ストリームをソースとして取り入れる場合や、フィードから読み込み、XML からドメインオブジェクトを作成する `XMLFeedRepositoryStrategy` を実装する時である。

13.3.3 ｜参考文献

仕様パターンについては、まだ推奨できる参考書はない。ここまで記述した中で、出版物として最良のものが[Evans and Fowler]である。また、[Evans]の中に優れた解説を見つけることができる。

13.3.4 ｜例：Person が持つ扶養家族の検索（Java）

クライアントオブジェクトの観点からすると、**リポジトリ**は簡単に使うことができる。データベースから扶養家族を検索するには、Person オブジェクトに一致させたい検索基準を表す Criteria オブジェクトを作成し、Criteria オブジェクトを**リポジトリ**に送る。

```
public class Person {
   public List dependents() {
      Repository repository = Registry.personRepository();
      Criteria criteria = new Criteria();
      criteria.equal(Person.BENEFACTOR, this);
      return repository.matching(criteria);
   }
}
```

共通のクエリーを、**リポジトリ**に特化したサブクラスに格納する。前の例では、**リポジトリ**のサブクラス `PersonRepository` を作成し、**リポジトリ**自体に検索基準の作成を移行した。

```
public class PersonRepository extends Repository {
   public List list dependentsOf(Person aPerson) {
```

```
        Criteria criteria = new Criteria();
        criteria.equal(Person.BENEFACTOR, aPerson);
        return matching(criteria);
    }
}
```

次にPersonオブジェクトは、dependents()メソッドを直接**リポジトリ**上に呼び出す。

```
public class Person {
    public List dependents() {
        return Registry.personRepository().dependentsOf(this);
    }
}
```

13.3.5 ｜例：リポジトリストラテジーのスワッピング（Java）

リポジトリのインタフェースは、ドメインレイヤをデータソースから隠ぺいするため、クライアントからの呼び出しを変更せずに**リポジトリ**内部のクエリーコードの実装をリファクタリングできる。ドメインコードは、ドメインオブジェクトのソースや送信先に関係しない。メモリ上に格納する場合、matching()メソッドを変更してドメインオブジェクトコレクションの中から判断基準を満たすものを選択する。ただし、使うデータ格納を随時変更するのではなく、データ格納間のスワッピングが自在にできることに魅力がある。したがって、クエリーを発行するストラテジーオブジェクトに委譲するmatching()メソッドの実装を変更する必要がある。もちろんメリットは、複数のストラテジーを持ち必要に応じてストラテジーを決定できるという点にある。この場合、次の2つのストラテジーを持つことをすすめる。RelationalStrategyは、データベースにクエリーを発行し、InMemoryStrategyは、ドメインオブジェクトのメモリ上のコレクションにクエリーを発行する。いずれのストラテジーもRepositoryStrategyインタフェースを実装し、matching()メソッドを公開する。したがって、リポジトリクラスの実装は次のようになる。

```
abstract class Repository {
    private RepositoryStrategy strategy;
    protected List matching(Criteria aCriteria) {
        return strategy.matching(aCriteria);
    }
}
```

RelationalStrategyは、判断基準から**クエリーオブジェクト**を作成し、データベースに

クエリーを発行することによってmatching()メソッドを実装する。**クエリーオブジェクト**が判断基準からクエリーを生成する方法を知っていると仮定すると、判断基準によって定義される適切なフィールドと値を使ってそれを準備できる。

```
public class RelationalStrategy implements RepositoryStrategy {
   protected List matching(Criteria criteria) {
      Query query = new Query(myDomainObjectClass())
      query.addCriteria(criteria);
      return query.execute(unitOfWork());
   }
}
```

InMemoryStrategyは、ドメインオブジェクトコレクションを繰り返し、判断基準を満たしている場合、それぞれのドメインオブジェクトで判断基準を要求することによって、matching()メソッドを実装する。判断基準は、ドメインオブジェクトに特定のフィールドの値を問い合わせ、リフレクションを使って要求を満たすコードを実装する。以下に、この選択を行うためのコードを示す。

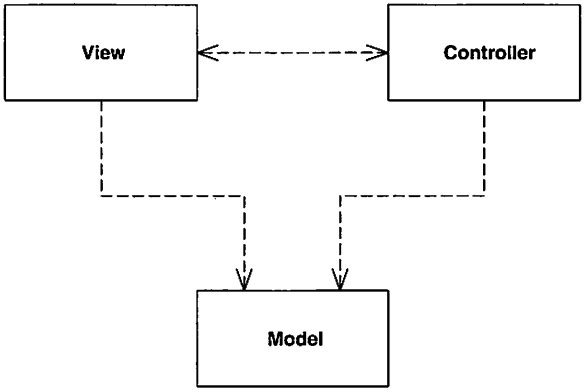
```
public class InMemoryStrategy implements RepositoryStrategy {
   private Set domainObjects;
   protected List matching(Criteria criteria) {
      List results = new ArrayList();
      Iterator it = domainObjects.iterator();
      while (it.hasNext()) {
         DomainObject each = (DomainObject) it.next();
         if (criteria.isSatisfiedBy(each))
            results.add(each);
      }
      return results;
   }
}
```

第14章

Webプレゼンテーションパターン

14.1 | モデルビューコントローラ

ユーザインタフェースの相互作用を 3 つの明確な役割へ分割する。



モデルビューコントローラ（MVC: Model View Controller）は、頻繁に引用される（それと同時に誤った引用も行われる）パターンである。1970 年代後半、Trygve Reenskaug によって Smalltalk プラットフォーム向けに開発されたフレームワークとしてスタートした。その後 UI フレームワークおよび UI 設計思想に重要な役割を果たしてきた。

14.1.1 | 動作方法

MVC には、それぞれの役割を持つ 3 つの要素がある。その要素とはモデル、ビュー、そしてコントローラである。モデルは、ドメインについて何らかの情報を表すオブジェクトである。これはデータや振る舞いを収納する非ビジュアル的なオブジェクトで、UI には使われない。**ドメインモデル**の中のオブジェクトは、このモデルの最もオブジェクト指向的な形

態である。また、もしUIの機構を含まなければ、**トランザクションスクリプト**をこのモデルと見なすこともできる。このような定義はモデルの概念を拡げてしまうが、MVCの内容を表すには適している。

ビューはUIの中でモデルを表現する。ユーザオブジェクトのモデルであれば、そのビューはUIウィジェットでいっぱいのフレームか、モデルからの情報を載せたHTMLページとなる。ビューは情報の表示にだけ関わる。情報の変更を扱うのは、MVCの第3の要素であるコントローラである。コントローラはユーザ入力を受け取り、モデルを操作し、ビューを適切に更新する。このように、UIはビューとコントローラのコンビネーションによって成り立つのである。

MVCを考えるとき、そこには根本的に異なる2種類の分離がある。モデルからのプレゼンテーションの分離と、ビューからのコントローラの分離である。

まず、**モデルからのプレゼンテーションの分離**であるが、優れたソフトウェア設計における基本的な経験則の1つである。この分離が重要なのは、次のとおりである。

- 基本的にプレゼンテーションおよびモデルは、利害関係が異なっている。ビューの開発を行う場合、UIの機構と優れたインタフェースのレイアウト方法を考えるが、一方、モデルを操作する場合、ビジネスポリシーについて、特にデータベースとの相互作用について考えることになる。どちらか一方を操作する場合には、まったく異なるライブラリを使う。2つの領域がある場合、設計者はどちらか一方を選択し専属的に扱うことが多い。
- 状況によっては、同一の基本的なモデル情報を別の角度から見る場合もある。プレゼンテーションとビューを分離することで、複数のプレゼンテーション（実際にまったく別物のインタフェース）の開発が可能となる一方、引き続き同じモデルコードを使うこともできる。顕著なメリットある構築として、リッチクライアント、Webブラウザー、リモートAPI、コマンドラインインタフェースを備えた同一モデルを提供することができる。単独のWebインタフェース内部でも、アプリケーションのさまざまなポイントに異なる顧客ページを持つこともできる。
- 非ビジュアルなオブジェクトは、通常ビジュアルなオブジェクトよりもテストが容易である。プレゼンテーションとモデルを分離することで、面倒なGUIスクリプティングツールの手段に頼ることなくドメインロジックを簡単にテストできる。

分離におけるキーポイントは、依存の方向である。プレゼンテーションはモデルに依存す

るが、モデルはプレゼンテーションに依存することはない。モデルでプログラミングを行う場合、どんなプレゼンテーションが使われているかについては、まったく意識しなくてすむ。このためタスクが簡単になると同時に、後の新しいプレゼンテーションの追加が容易になる。モデルを変更することなく自由にプレゼンテーションを変更できるのである。

こうした原則によって、ある共通の問題が発生する。複数ウィンドウを持ったリッチクライアントインタフェースの場合、あるモデルの複数のプレゼンテーションが同時に画面上に表示される可能性が生じる。したがってユーザが1つのプレゼンテーションからモデルに変更を加えた場合、他のプレゼンテーションも同様に変更しなければならない。依存性を作らずにこうした作業を実行するには、イベント伝播またはリスナーといった「オブザーバー」パターンの実装 [Gang of Four] が必要となる。プレゼンテーションは、モデルのオブザーバーとして動作し、モデルが変化するごとにイベントが送信され情報を更新する。

もう1つのエリアである**ビューとコントローラの分離**だが、こちらの重要度はやや低めである。皮肉なことだが、Smalltalk のほとんどのバージョンでビューとコントローラは、分離されていない。両者を分離したいと考える最も一般的な例は、編集可能な振る舞いと編集不可能な振る舞いのサポートである。この2つのケースに対して、1つのビューと2つのコントローラで対応できる（ここではコントローラが、ビューに対するストラテジーとして機能する [Gang of Four]）。しかし、ほとんどのシステムで、ビューごとに1つのコントローラしか用意されず、その分離も通常は行われていない。この方法が再び脚光を浴びるようになったのは、コントローラとビューの分離が有効な Web インタフェースの登場によってである。

GUI フレームワークは、ビューとコントローラを組み合わせているために、MVC は誤って引用されることが多かった。モデルおよびビューは明白だが、コントローラはどこにあるのか。一般的に、モデルとビューの間にある（**アプリケーションコントローラ**）と考えられているが、それではなぜ「コントローラ」という言葉が2つの場所で使われているのか。**アプリケーションコントローラ**のメリットがどのようなものでも、MVC コントローラとはまったく異質のものである。

このパターンセットの目的を理解するために、原則は必ず知っておくべきである。MVC を深く掘り下げて知りたい場合に最も役立つ参考書は、[POSA] である。

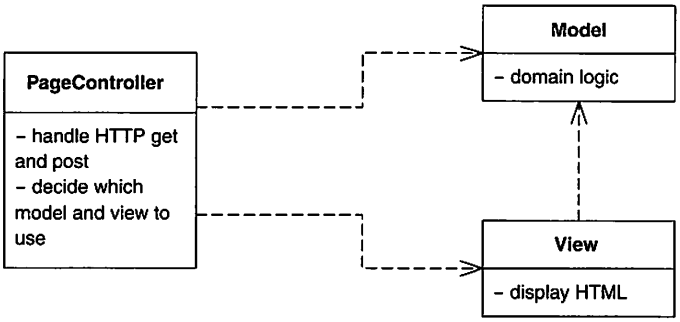
14.1.2 ｜使用するタイミング

すでに述べたように、MVC の特長は2種類の分離にある。プレゼンテーションとモデルの分離は、ソフトウェアの設計における最も重要な原則の一つである。唯一の例外は、モデルが振る舞いを持たないようなとてもシンプルなシステムの場合である。非ビジュアルなロジックなら、すぐに分離すべきである。残念なことに、多くの UI フレームワークでは、その実行は困難で、そうでないフレームワークは分離しないように教えられることが多い。

ビューとコントローラの分離は、比較的重要ではなく、必要なときにだけ分離を実践することを推奨する。リッチクライアントシステムの場合、分離はほとんどないが、コントローラが分離されているWebフロントエンドにおいて分離は珍しいことではない。ここに示すWeb設計のパターンは、こうした原則をベースとしている。

14.2 | ページコントローラ

特定のページに対するリクエストやWebサイト上のアクションを扱うオブジェクト。



多くの人にとって基本的なWeb環境とは、静的なHTMLページである。静的HTMLをリクエストする場合、Webサーバに対して、リクエスト先に保存されているHTML文書の名前とパスを渡す。つまりWebサイトの各ページは、サーバ上では独立した文書であるという考え方である。動的ページの場合、パス名と対応するファイルの間には、複雑な関連があるため、より興味深い。しかし、リクエストを扱う1つのファイルにアクセスするために1つのパスを使うという手法は、シンプルなモデルなのでわかり易い。

その結果、**ページコントローラ**は、Webサイトの論理ページごとに1つの入力コントローラを持つ。コントローラは、サーバページ環境に置かれることが多いために、ページ自体であるだけでなく、ページに対応する独立したオブジェクトの場合もある。

14.2.1 | 動作方法

ページコントローラの背後にある基本的な考え方とは、Webサーバ上の1つのモジュールを、Webサイト上の各ページのコントローラとして機能させることである。正確にページごとに1つのモジュールというわけにはいかない。リンクをクリックすると、動的な情報に応じてさまざまなページが表示されることがあるからである。厳密には、コントローラは、リンクやボタンをクリックするそれぞれの**アクション**と結び付いている。

ページコントローラは、スクリプト（CGIスクリプト、サーブレットなど）として構築す

ることも、サーバページ（ASP、PHP、JSP など）として構築することも可能である。サーバページを利用するとページコントローラとテンプレートビューが同一ファイル内で結合する。ページの表示方法がシンプルな場合は問題ないが、適切なモデルの構築には手間がかかるため、テンプレートビューにとっては良いが、ページコントローラにとってはそれほど有用ではない。しかし、リクエストからのデータの抽出や表示するビューの決定に関するロジックが含まれる場合は、サーバページにおいて難しいスクリプトレットコードとなってしまう可能性がある。

スクリプトレットコードを処理する 1 つの方法としては、ヘルパーオブジェクトの利用が挙げられる。この場合、サーバページが最初に行うのは、ヘルパーページを呼び出してロジックを処理することである。ヘルパーは、オリジナルのサーバページに制御を返すことも、異なるサーバページへ転送してビューとして機能することも可能である。その場合サーバページはリクエストハンドラとなるが、コントローラロジックの大部分はヘルパーにある。

別の手法としては、スクリプトをハンドラおよびコントローラにする方法が挙げられる。Web サーバがスクリプトに制御を渡すと、スクリプトはコントローラの機能を実行し、最終的に適切なビューへ結果を転送して表示させる。

ページコントローラの基本的な機能は、以下のとおりである。

- URL をデコードし、フォームデータを抽出してアクションに必要なデータを算出する。
- データを処理するモデルオブジェクトを作成し実行する。そして、HTML リクエストの関連データをモデルへ渡す。その結果モデルオブジェクトは、HTML リクエストへの関係を持つ必要はなくなる。
- どのビューが結果を表示するかを判断し、モデル情報を転送する。

ページコントローラは単独のクラスである必要はなく、ヘルパーオブジェクトを実行することができる。複数のハンドラが同じタスクを行わなければならない場合は、特に有効である。ヘルパークラスは、重複する可能性のあるコードの置き場所としても利用できる。

サーバページやスクリプトで何らかの URL を処理できないという理由はない。コントローラロジックをまったく持たないか、もしくはほとんど持たない URL は、わかりやすく修正しやすいシンプルなメカニズムのため、サーバページにとっては簡単な処理対象である。より複雑なロジックを持った URL は、スクリプトにまかせるのがよい。私は、すべて同一の方法（サーバページまたはスクリプト）で処理する方針を持ったチームと交流を持った経験がある。アプリケーションにおける一貫性というメリットも、スクリプトレットが満載されたサーバページやスクリプトを介した大量のシンプルなパスによる弊害により、相殺されてしまう。

14.2.2 | 使用するタイミング

意思決定のポイントは、**ページコントローラ**と**フロントコントローラ**のどちらを利用するかである。**ページコントローラ**は一般の作業に適し、特定の動作が特定のサーバページまたはスクリプトクラスによって処理される構築メカニズムにつながる。したがって、トレードオフは、**フロントコントローラ**のメリットとその複雑性をどう評価するかである。トレードオフの大部分は、Web サイトの案内操作の複雑性の違いとなって現れる。

ページコントローラは、コントローラロジックの大半がとてもシンプルなサイトでは、特に有用である。この場合、ほとんどの URL はサーバページによって処理でき、より複雑なものもヘルパーによって処理できる。コントローラロジックがシンプルな場合、**フロントコントローラ**は多くのオーバーヘッドを追加する。

一部のリクエストを**ページコントローラ**によって処理し、その他は**フロントコントローラ**によって処理するようなサイトは珍しくない（特に、開発チームがそれらをリファクタリングする場合）。これら 2 つのパターンは、特に問題なく混在できる。

14.2.3 | 例：サーブレットコントローラと JSP ビューによるシンプルな表示（Java）

ページコントローラのシンプルな例とは、ある対象に対する情報を表示するというものである。ここでは、レコーディング Artist（アーティスト）に関する情報を表示する。URL は、`http://www.thingy.com/recordingApp/artist?name=danielaMercury` という行に含まれている。

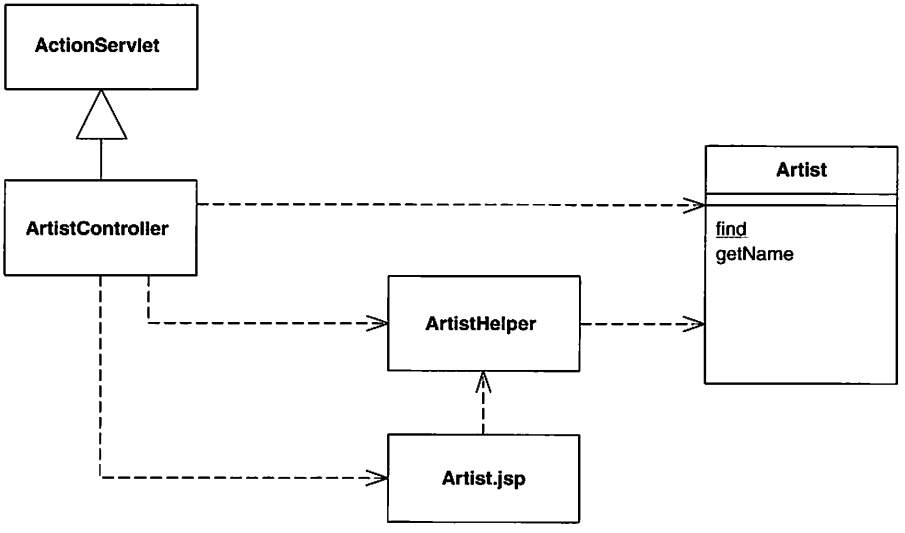


図 14.1 ——ページコントローラサーブレットおよび JSP ビューによるシンプルな表示方法に関連したクラス

Web サーバは、/artist を ArtistController の呼び出しと認識するように設定しておく。Tomcat では、web.xml ファイルに次のようなコードを記述する。

```
<servlet>
   <servlet-name>artist</servlet-name>
   <servlet-class>actionController.ArtistController</servlet-class>
</servlet>
<servlet-mapping>
   <servlet-name>artist</servlet-name>
   <url-pattern>/artist</url-pattern>
</servlet-mapping>
```

ArtistController は、リクエストを処理するメソッドを実装する。

```
class ArtistController...

   public void doGet(HttpServletRequest request, HttpServletResponse response)
         throws IOException, ServletException {
      Artist artist = Artist.findNamed(request.getParameter("name"));
      if (artist == null)
         forward("/MissingArtistError.jsp", request, response);
      else {
         request.setAttribute("helper", new ArtistHelper(artist));
         forward("/artist.jsp", request, response);
      }
   }
```

とてもシンプルな例だが、重要なポイントが含まれている。まず、コントローラは、処理を行う（ここでは表示すべき正しいモデルオブジェクトを見つけ出す）ために必要なモデルオブジェクトを作成しなければならない。次に、JSP が、オブジェクトを正しく表示できるように、HTTP リクエストに正しい情報を格納する。例では、ヘルパーを作成しリクエストに格納している。最後に、**テンプレートビュー**へと転送して、表示の処理を行わせる。転送は、標準的な振る舞いであるため、当然ながら**ページコントローラ**のスーパークラスに置かれている。

```
class ActionServlet...

   protected void forward(String target,
                          HttpServletRequest request,
                          HttpServletResponse response)
```

```
        throws IOException, ServletException
    {
        RequestDispatcher dispatcher =
            getServletContext().getRequestDispatcher(target);
        dispatcher.forward(request, response);
    }
```

テンプレートビューとページコントローラを結合するときの重要なポイントは、JSPが必要とするオブジェクトを渡すリクエストの中のパラメータ名である。

ここに示すコントローラロジックはとてもシンプルだが、より複雑になった場合は、サーブレットをコントローラとして引き続き利用する。Album（アルバム）にも同様の振る舞いができる。ただし、ClassicalAlbum（クラシックアルバム）は、異なるモデルオブジェクトを持ち、他の異なるJSPによって加工される必要があるとする。この振る舞いを実現するには、ここでもコントローラクラスを使用する。

```
class AlbumController...

    public void doGet(HttpServletRequest request,
        HttpServletResponse response) throws IOException,
        ServletException
    {
        Album album = Album.find(request.getParameter("id"));
        if (album == null) {
            forward("/missingAlbumError.jsp", request, response);
            return;
        }
        request.setAttribute("helper", album);
        if (album instanceof ClassicalAlbum)
            forward("/classicalAlbum.jsp", request, response);
        else
            forward("/album.jsp", request, response);
    }
```

例では、独立したヘルパークラスを作成する代わりに、モデルオブジェクトをヘルパーとして利用している点に注目してほしい。ヘルパークラスがモデルクラスに対し、転送する意味がないようなコードである場合、こうした方法は効果的である。ただしその場合、モデルクラスに一切のサーブレット依存コードが含まれないようにする必要がある。サーブレット依存コードは、必ず独立したヘルパークラスに置かなければならない。

14.2.4 ｜例： JSP をハンドラとして使用する（Java）

サーブレットをコントローラとして利用することは1つの手段ではあるが、最も一般的な手法は、サーバページ自体をコントローラにすることである。この手法の問題点は、データの収集時におけるサーバページの先頭のスクリプトレットコードにある。同意していただけると思うが、優れた設計のソフトウェアに対するスクリプレットコードの位置付けは、純粋なスポーツに対するプロレスの位置付けに等しいというのが私の考えだ。

それにもかかわらず、コントローラの機能を実際に実行するヘルパーに制御を委譲する一方で、サーバページをリクエストハンドラにすることは可能である。そのため、サーバページによってURLを示すというシンプルなプロパティは維持される。私は、http://localhost:8080/isa/album.jsp?id=zero 形式の URL を利用して、この方法を Album 表示のために行っている。Album は、アルバム JSP によって直接表示されるが、クラシックレコードの場合、別の表示方法であるクラシックアルバム JSP を必要とする。

コントローラの振る舞いは、JSP のヘルプクラスにおいて現れる。ヘルパーは、アルバム JSP 自身に設定されている。

```
album.jsp...

<jsp:useBean id="helper" class="actionController.AlbumConHelper"/>
<%helper.init(request, response);%>
```

init の呼び出しは、コントローラの振る舞いを実行するようにヘルパーを設定する。

```
class AlbumConHelper extends HelperController...

   public void init(HttpServletRequest request, HttpServletResponse response) {
      super.init(request, response);
      if (getAlbum() == null) forward("missingAlbumError.jsp",
         request, response);
      if (getAlbum() instanceof ClassicalAlbum) {
         request.setAttribute("helper", getAlbum());
         forward("/classicalAlbum.jsp", request, response);
      }
   }
```

標準的なヘルパーの振る舞いは、当然ヘルパースーパークラスに置かれている。

```
class HelperController...

   public void init(HttpServletRequest request,
         HttpServletResponse response) {
      this.request = request;
      this.response = response;
   }
   protected void forward(String target,
                     HttpServletRequest request,
                     HttpServletResponse response)
   {
      try {
         RequestDispatcher dispatcher = request.getRequestDispatcher(target);
         if (dispatcher == null) response.sendError(response.SC_NO_CONTENT);
         else dispatcher.forward(request, response);
      } catch (IOException e) {
         throw new ApplicationException(e);
      } catch (ServletException e) {
         throw new ApplicationException(e);
      }
   }
```

ここに示すコントローラの振る舞いと、サーブレットを使う場合のコントローラの振る舞いとの大きな違いは、ハンドラJSPもデフォルトビューであり、コントローラが別のJSPに転送しない限り、制御はオリジナルのハンドラへと戻る点である。大抵の場合、JSPがビューとして直接機能するため、転送がほとんど行われないようなページにおいてはこれが長所となる。ヘルパーの初期化はあらゆるモデルの振る舞いを開始し、その後のビューの設定を行う役割を担う。Webページは、ビューとして機能するサーバページと関連付けられるためとてもわかりやすいモデルであり、当然ながらWebサーバのシステム構成とも適合することが多い。

ハンドラを初期化するための呼び出しは、あまりうまくいかない場合がある。JSP環境では、こうした点もカスタムタグによる処理によって大幅に改善される。タグは自動的に適切なオブジェクトを作成し、リクエストに収めて初期化する。そのために必要なのは、JSPページのシンプルなタグだけである。

```
<helper:init name = "actionController.AlbumConHelper"/>
```

以下のカスタムタグの実装は、その作業を行うためである。

```
class HelperInitTag extends HelperTag...

   private String helperClassName;
   public void setName(String helperClassName) {
      this.helperClassName = helperClassName;
   }
   public int doStartTag() throws JspException {
      HelperController helper = null;
      try {
         helper = (HelperController) Class.forName(helperClassName).newInstance();
      } catch (Exception e) {
         throw new ApplicationException("Unable to instantiate " +
            helperClassName, e);
      }
      initHelper(helper);
      pageContext.setAttribute(HELPER, helper);
      return SKIP_BODY;
   }
   private void initHelper(HelperController helper) {
      HttpServletRequest request = (HttpServletRequest)
         pageContext.getRequest();
      HttpServletResponse response = (HttpServletResponse)
         pageContext.getResponse();
      helper.init(request, response);
   }

class HelperTag...

   public static final String HELPER = "helper";
```

私がカスタムタグを使う場合、プロパティアクセス用に作成することもある。

```
class HelperGetTag extends HelperTag...

   private String propertyName;
   public void setProperty(String propertyName) {
      this.propertyName = propertyName;
   }
   public int doStartTag() throws JspException {
      try {
         pageContext.getOut().print(getProperty(propertyName));
      } catch (IOException e) {
         throw new JspException("unable to print to writer");
      }
```

```
        return SKIP_BODY;
    }

class HelperTag...
    protected Object getProperty(String property) throws JspException {
        Object helper = getHelper();
        try {
            final Method getter =
                helper.getClass().getMethod(gettingMethod(property), null);
            return getter.invoke(helper, null);
        } catch (Exception e) {
            throw new JspException ("Unable to invoke " +
                gettingMethod(property) + " - " + e.getMessage());
        }
    }
    private Object getHelper() throws JspException {
        Object helper = pageContext.getAttribute(HELPER);
        if (helper == null) throw new JspException("Helper not found.");
        return helper;
    }
    private String gettingMethod(String property) {
        String methodName = "get" + property.substring(0, 1).toUpperCase()
            + property.substring(1);
        return methodName;
    }
```

（リフレクションを利用してgetメソッドを呼び出すよりも、Java Beanを使う方がよいと考える人も中にはいるだろうが、それは正しい。そういう人は、おそらくメソッドをどう変更すればよいかも心得ているにちがいない）。

取得用のタグを定義して、ヘルパーの情報を抽出するのに使うこともできる。タグのほうが短くて済み、「helper」という綴りを間違うことも減る。

```
<B><helper:get property = "title"/></B>
```

14.2.5 ｜例：コードビハインドによるページハンドラ（C#）

.NETにおけるWebシステムは、**ページコントローラとテンプレートビュー**のパターンとともに動作するように設計されている。ただし、Webイベントを別の手法によって処理することも可能である。以下の例では、私は.NETの推奨スタイルを基に、**テーブルモジュール**を使ってドメイン上部にプレゼンテーション層を構築し、レイヤ間の情報のメインキャリアとしてデータセットを使っている。

ここでは、クリケットの試合の1イニングにおける得点数および得点率を表示するページを作成する。競技が多くの読者にとって馴染みのないものであることは十分承知しているので、次のように定義させてもらうことにする。まず得点数とは打者の得点であり、得点率とは、得点を投球数で割ったものとする。得点数と投球数はデータベースにあり、得点率は(小さいが、説明する上で有用なドメインロジックの構成要素である)アプリケーションが計算によって求めるのである。

この設計におけるハンドラは、ASP.NET Webページであり、.aspxファイルに収められている。他のサーバページの構築と同様、ファイルによって、プログラミングロジックをスクリプトレットとして直接ページに埋め込むことができる。スクリプトレットを書くぐらいなら気の抜けたビールを飲んでいる方がましだと考えている私は、もちろんそんなことはしない。ASP.NETの**コードビハインド**メカニズムを使えば、ファイルおよびクラスをaspxページと関連付け、aspxページのヘッダーに示すことができるからだ。

```
<%@ Page language="c#" Codebehind="bat.aspx.cs" AutoEventWireup="false"
   trace="False" Inherits="batsmen.BattingPage" %>
```

ページは、コードビハインドクラスのサブクラスとして設定されているため、保護されているプロパティおよびメソッドの利用が可能となる。ページオブジェクトは、リクエストのハンドラであり、コードビハインドは、page_Loadメソッドを定義することによって、処理方法を定義できる。ページが共通のフローに従うなら、テンプレートメソッド[Gang of Four]を持った**レイヤスーパークラス**を定義することができる。

```
class CricketPage...

      protected void Page_Load(object sender, System.EventArgs e) {
         db = new OleDbConnection(DB.ConnectionString);
         if (hasMissingParameters())
            errorTransfer (missingParameterMessage);
         DataSet ds = getData();
         if (hasNoData (ds))
            errorTransfer ("No data matches your request");
         applyDomainLogic (ds);
         DataBind();
         prepareUI(ds);
      }
```

テンプレートメソッドは、リクエスト処理をいくつかの共通の手順へと分割する。こうすることで、Webリクエストを処理する単一の共通フローを定義できる一方、各**ページコン**

トローラに特定の手順の実装を行わせることもできる。そのためページコントローラを記述してしまえば、テンプレートメソッド用にどの共通フローを使用すべきかがわかる。まったく別の動作を必要とするページがある場合、いつでもそのページ読み込みメソッドを無効にできる。

最初の作業は、ページに送られてくるパラメータの妥当性の確認である。より現実に近い例では、さまざまな形式の値の初期チェックが必要になるが、例では http://localhost/batsmen/bat.aspx?team=England&innings=2&match=905 形式の URL を解読している。例における唯一の妥当性確認は、データベースクエリーに必要なパラメータがあるかどうかである。これに関しても私は、誰かが妥当性確認の優れたパターンセットを作成してくれるまで、シンプルなエラー処理に徹してきた。そのため、ここで示す特定のページには、一連の必須パラメータが定義されていて、**レイヤスーパータイプ**はそれをチェックするロジックを持っている。

```
class CricketPage...

   abstract protected String[] mandatoryParameters();
   private Boolean hasMissingParameters() {
      foreach (String param in mandatoryParameters())
         if (Request.Params[param] == null) return true;
      return false;
   }
   private String missingParameterMessage {
      get {
         String result = "<P>This page is missing mandatory parameters:</P>";
         result += "<UL>";
         foreach (String param in mandatoryParameters())
            if (Request.Params[param] == null)
               result += String.Format("<LI>{0}</LI>", param);
         result += "</UL>";
         return result;
      }
   }
   protected void errorTransfer (String message) {
      Context.Items.Add("errorMessage", message);
      Context.Server.Transfer("Error.aspx");
   }

class BattingPage...

   override protected String[] mandatoryParameters() {
```

```
        String[] result = {"team", "innings", "match"};
        return result;
    }
```

次の段階では、データベースからデータを抽出し、ADO.NET 非接続データセットオブジェクトに収める。これは batting テーブルに対する単独のクエリーである。

```
class CricketPage...

        abstract protected DataSet getData();
        protected Boolean hasNoData(DataSet ds) {
            foreach (DataTable table in ds.Tables)
                if (table.Rows.Count != 0) return false;
                return true;
        }

class BattingPage...

        override protected DataSet getData() {
            OleDbCommand command = new OleDbCommand(SQL, db);
            command.Parameters.Add(new OleDbParameter("team", team));
            command.Parameters.Add(new OleDbParameter("innings", innings));
            command.Parameters.Add(new OleDbParameter("match", match));
            OleDbDataAdapter da = new OleDbDataAdapter(command);
            DataSet result = new DataSet();
            da.Fill(result, Batting.TABLE_NAME);
            return result;
        }
        private const String SQL =
            @"SELECT * FROM batting
                WHERE team = ? AND innings = ? AND matchID = ?
                ORDER BY battingOrder";
```

ここで、**テーブルモジュール**にまとめられているドメインロジックの出番となる。コントローラは、抽出されたデータセットを**テーブルモジュール**に渡して処理を委ねる。

```
class CricketPage...

        protected virtual void applyDomainLogic (DataSet ds) {}

class BattingPage...
```

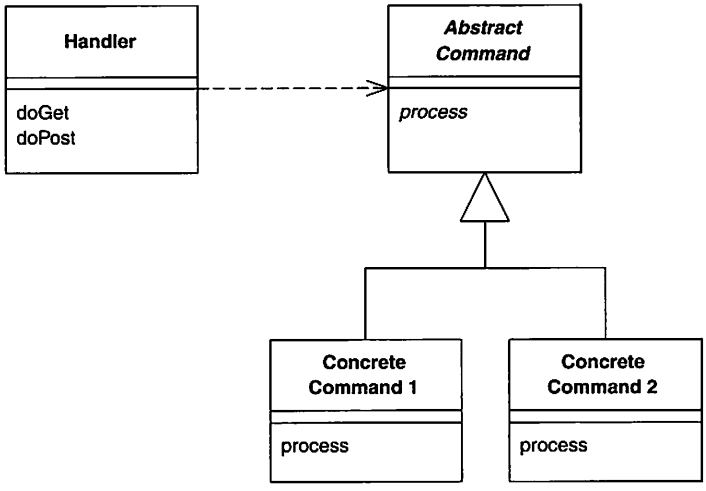
```
override protected void applyDomainLogic (DataSet dataSet) {
    batting = new Batting(dataSet);
    batting.CalculateRates();
}
```

この時点で、ページハンドラのコントローラの部分が実行される。つまり、従来の**モデルビューコントローラ**の言い方に従うと、コントローラは、表示を行うためにここでビューに引き渡すのである。設計において、BattingPage はコントローラおよびビューとして機能し、prepareUI に対する最後の呼び出しはビューの振る舞いの一部である。

以上でパターンの例は終了である。ドラマティックなエンディングに欠けていると思われるかもしれないが、そうした例は後ほど（373 ページ参照）紹介することにしよう。

14.3 フロントコントローラ

Web サイトへのあらゆるリクエストを扱うコントローラ。



複雑な Web サイトでは、リクエストの処理時に、同じようなことをいろいろ行わなければならない。その中には、セキュリティ、国際化、特定のユーザに対する特定のビューの提供などが含まれる。入力コントローラの振る舞いが複数のオブジェクトに分散している場合、振る舞いの多くは重複してしまうことが多く、実行時に振る舞いを変更することも難しくなる。

フロントコントローラは、単独のハンドラオブジェクトを介してリクエストをチャンネル化することで、リクエスト処理を統合する。オブジェクトは共通の振る舞いを実行し装飾子によって実行時に修正することもできる。その後ハンドラは、リクエストに固有の振る舞い

を行うコマンドオブジェクトを送信する。

14.3.1 | 動作方法

フロントコントローラは、Web サイトのすべての呼び出しを扱い、Web ハンドラとコマンド階層構造の 2 つの部分から構成される。Web ハンドラは、サーバからの情報を受け取ったり、リクエストを取得したりする。URL とリクエストから必要な情報を抽出し、どのようなアクションを実行するかを判断し、アクションを実行するコマンドに委譲を行う(図 14.2 参照)。

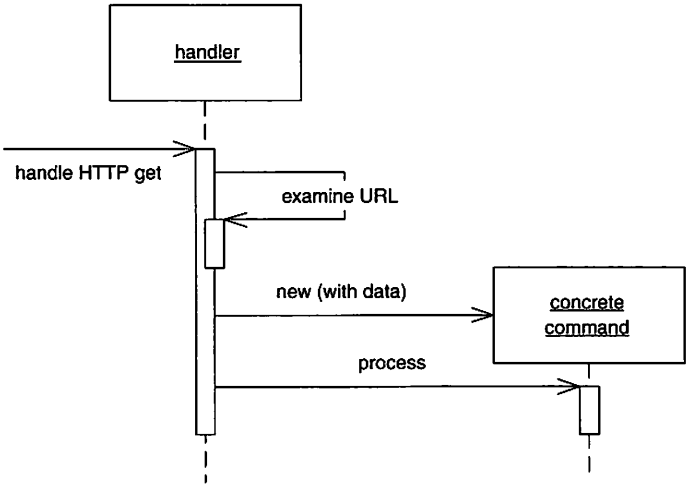


図 14.2 ——フロントコントローラの動作

Web ハンドラはまったくレスポンスを返さないため、サーバページとしてではなくクラスとして実装されることがほとんどである。コマンドもサーバページというよりはクラスであり、頻繁に HTTP の情報を渡されるが、Web 環境についてまったく感知する必要がない。Web ハンドラ自体は極めてシンプルなプログラムであり、実行するコマンドを判断する以外は何も行わない。

Web ハンドラは、実行するコマンドの判断を静的に行うことも、動的に行うこともできる。静的バージョンには URL の解析と条件分岐ロジックが含まれ、動的バージョンには URL の標準部分の抽出と、動的インスタンス化を使ったコマンドクラスの作成が含まれる。

静的なケースには、明示的なロジック、ディスパッチのコンパイルタイムエラーチェック、および URL の柔軟性というメリットがある。動的なケースでは、Web ハンドラを変更することなく新しいコマンドを追加することができる。

動的な呼び出しの場合、コマンドクラスの名前をURLに収めることもでき、URLをコマンドクラス名と結び付けるプロパティファイルを使うこともできる。プロパティファイルを使うと編集しなければならないファイルが1つ増えるが、大量のWebページ検索をしなくても簡単にクラス名を変更することができる。

フロントコントローラとの組み合わせが特に有効なパターンとしては、**インターセプティングフィルタ**が挙げられる（[Alur et al]参照）。これは基本的にはフロントコントローラのハンドラをラッピングする装飾子であり、認証、ログの記録、ロケールの識別といった問題を処理する**フィルタチェーン**（フィルタのパイプライン）の構築ができる。フィルタを利用することで、システム構成時に使うフィルタの動的な設定ができる。

Rob Meeは、**フロントコントローラ**の素晴らしいバリエーションを私に見せてくれた。そこでは、Webハンドラとディスパッチャとに分離した2段階のWebハンドラが使われていた。Webハンドラは、httpパラメータから基本データを抽出し、ディスパッチャがWebサーバフレームワークから完全に独立するような方法でディスパッチャに渡す。これによりテストが簡素化される。テストコードは、Webサーバ上で実行しなくてもディスパッチャを直接動かすことができるのである。

ハンドラもコマンドも、コントローラの一部であることを覚えておいてほしい。結果として、コマンドはレスポンスのためにどのビューを使用するかを選ぶことができる（あるいは選ばなければならない）。ハンドラの唯一の役割は、実行するコマンドの選択である。選択が終わればリクエストに対するハンドラの役割はもはやない。

14.3.2 | 使用するタイミング

フロントコントローラは、対応する**ページコントローラ**より複雑な設計となっている。そのため利用する以上は、いくつかのメリットがなければならない。

Webサーバには、**フロントコントローラ**が1つだけ構成され、Webハンドラは残りの送信を行う。これによってWebサーバのシステム構成が簡素化され、Webサーバの設定が面倒な場合にはメリットとなる。動的コマンドの場合、一切の変更をせずに新しいコマンドを追加できる。ハンドラはWebサーバに一定の方法で登録するだけでよいので、移植も容易となる。

リクエストごとに新しいコマンドオブジェクトを作成するため、コマンドクラスを安全なスレッドにすることに気を使わなくてよい。マルチスレッド化プログラミングの複雑さを回避できるが、モデルオブジェクトなどほかのオブジェクを共有しないように注意する必要がある。

一般に言われている**フロントコントローラ**のメリットとは、**ページコントローラ**において重複する可能性のあるコードを外に出せる点にある。しかし厳密に言えば、こうした操作の

大部分は、スーパークラスである**ページコントローラ**によって実現できる。

存在するコントローラはただ1つなので、装飾子によって実行時にその振る舞いを簡単に拡張できる [Gang of Four]。認証、文字のエンコーディング、国際化のための装飾子とシステム構成ファイルを使って、サーバの実行中であっても装飾子を追加することができる([Alur et al.]では、**インターセプティングフィルタ**の名でこうした手法について詳しく解説されている)。

14.3.3 | 参考文献

[Alur et al.]は、Java による**フロントコントローラ**の実装方法について詳しく解説している。また、**フロントコントローラ**と相性の良い**インターセプティングフィルタ**についても解説されている。

Java の Web フレームワークの多くがこのパターンを使用している。中でも優れた例が [Struts] に紹介されている。

14.3.4 | 例　シンプルな表示 (Java)

これは、レコーディングアーティストについての情報を表示するという、オリジナルかつ斬新なタスクのために**フロントコントローラ**を使用した簡単な例である。

ここでは、`http://localhost:8080/isa/music?name=barelyWorks&command=Artist` 形式の URL とともに動的コマンドを使用している。コマンドパラメータによって、Web ハンドラにどのコマンドを使うかを指示している。

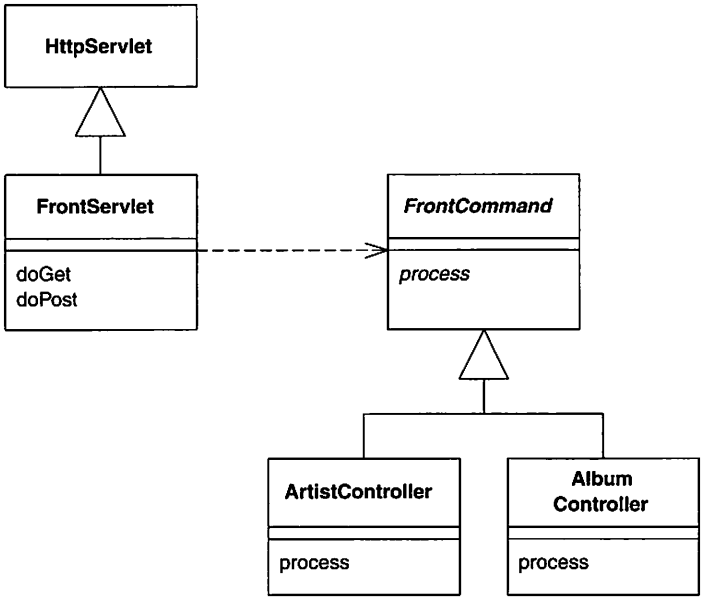


図14.3——フロントコントローラを実装するクラス

まずハンドラから始めよう。私はこれをサーブレットとして実装している。

```
class FrontServlet...

   public void doGet(HttpServletRequest request,
         HttpServletResponse response) throws IOException,
         ServletException {
      FrontCommand command = getCommand(request);
      command.init(getServletContext(), request, response);
      command.process();
   }
   private FrontCommand getCommand(HttpServletRequest request) {
      try {
         return (FrontCommand) getCommandClass(request).newInstance();
      } catch (Exception e) {
         throw new ApplicationException(e);
      }
   }
   private Class getCommandClass(HttpServletRequest request) {
      Class result;
      final String commandClassName = "frontController." +
         (String) request.getParameter("command") + "Command";
```

```
        try {
            result = Class.forName(commandClassName);
        } catch (ClassNotFoundException e) {
            result = UnknownCommand.class;
        }
        return result;
    }
```

ロジックはシンプルである。ハンドラは、コマンド名と「Command」の組み合わせを名前として持つクラスをインスタンス化しようとする。新しいコマンドが用意できたら、HTTP サーバから得た必要な情報によってそれを初期化する。私は、このシンプルな例のために必要なものだけを渡しているが、HTTP セッションなどが必要になることもある。コマンドが見つからない場合、私は**スペシャルケース**パターンを使って、定義されていないコマンドを返すようにしている。よくあることだが**スペシャルケース**で多くのエラーチェックの追加を回避できる。

コマンドは、かなりのデータと振る舞いを共有する。それらはすべて Web サーバからの情報による初期化が必要である。

```
class FrontCommand...

    protected ServletContext context;
    protected HttpServletRequest request;
    protected HttpServletResponse response;
    public void init(ServletContext context,
                HttpServletRequest request,
                HttpServletResponse response)
    {
        this.context = context;
        this.request = request;
        this.response = response;
    }
```

また、forward メソッドなど共通の振る舞いを提供することができ、コマンドをオーバーライドする抽象プロセスコマンドを定義することもできる。

```
class FrontCommand...

    abstract public void process()throws ServletException, IOException ;
    protected void forward(String target) throws ServletException, IOException
    {
```

```
        RequestDispatcher dispatcher = context.getRequestDispatcher(target);
        dispatcher.forward(request, response);
    }
```

コマンドオブジェクトは、少なくともこの例ではとてもシンプルである。process メソッドを実装するだけだが、そこにはモデルオブジェクトの適切な振る舞いの実行、ビューが必要とする情報リクエストの追加、および**テンプレートビュー**への転送などが含まれている。

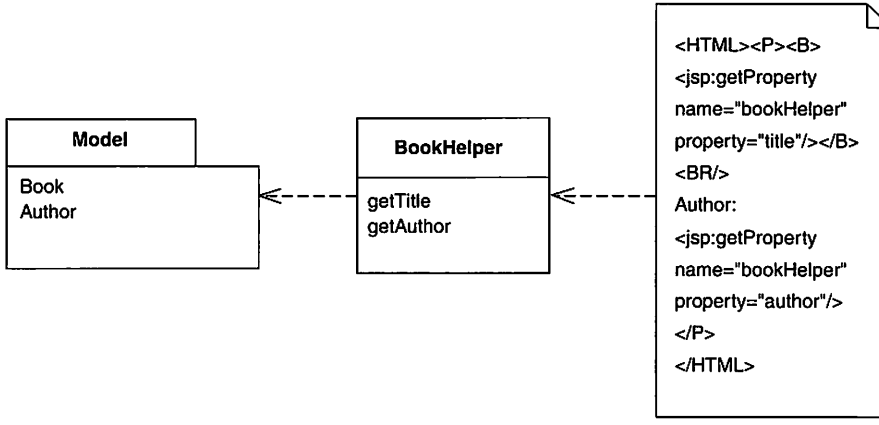
```
class ArtistCommand...

    public void process() throws ServletException, IOException {
        Artist artist = Artist.findNamed(request.getParameter("name"));
        request.setAttribute("helper", new ArtistHelper(artist));
        forward("/artist.jsp");
    }
```

定義されていないコマンドは、単調なエラーページを表示するだけである。

```
class UnknownCommand...

    public void process() throws ServletException, IOException {
        forward("/unknown.jsp");
    }
```

14.4 | テンプレートビュー

HTML ページにマークを埋め込むことで、HTML へ情報を加工する。



HTML を吐き出すプログラムの記述は、意外に難しいことが多い。プログラミング言語は、以前にくらべてテキストの作成を得意としてはいるが（Fortran や標準的な Pascal でのキャラクタ処理を覚えている読者は何人いるだろう）、文字列の作成や連結には今でも苦労する。そうした作業が少なければいいのだが、しかし HTML ページ全体はテキスト処理の集合である。

静的 HTML ページ（リクエストによって変化することがない）の場合、適当な WYSIWYG エディタが利用できる。通常のテキストエディタを好む人であっても、プログラム言語で文字列連結を行うよりは、テキストやタグを入力する方がはるかに簡単であると感じている。

もちろん、問題となるのは動的 Web ページの場合である。つまり、データベースクエリーなどの結果を取り込み、それを HTML へと組み込むようなページである。ページは結果ごとに外観が異なるため、HTML エディタは役立たない。

最も適切な方法は、動的ページと静的ページを同様に構成し、動的な情報を収集するための呼び出しだと解釈されるマークを埋め込むことである。ページの静的な部分は、特定のレスポンスに対するテンプレートとして機能するので、私はこれを**テンプレートビュー**と呼んでいる。

14.4.1 | 動作方法

テンプレートビューの基本的な考え方は、記述時に静的ＨＴＭＬにマークを埋め込むことである。ページがリクエストへの情報提供に使用される場合、マークは、データベースクエ

リーなどの計算処理によって置き換えられる。こうしたページは、プログラマではない人でも、WYSIWYGエディタなどを利用することによりレイアウトすることができる。マークは実際のプログラムとやり取りを行い、結果を組み込む。

テンプレートビューを利用しているツールは数多い。そのため、ここで紹介するパターンは自分でページを構築するためのものではなく、ページの効果的な使い方やその他のオプションに関するものである。

14.4.1.1 ■ マークの埋め込み

HTMLにマークを配置する方法は数多くある。1つはHTML風のタグを使う方法である。これはWYSIWYGエディタとの相性が良い。というのも、角括弧（< >）で囲まれているものはすべて特別であるとみなされ、無視されるか、別な扱い方が行われる。タグがきちんと整形されたXMLのルールに従っていれば、結果として生成される文書にXMLツールを使うこともできる（HTMLの場合はXHMTL）。

もう1つの方法は、本文中に特別なテキストマークを使う方法である。WYSIWYGエディタは、これをテキストとして扱い、無視はするがスペルチェックのような処理を行うことがある。この方法のメリットは、シンタックスがHTML/XMLのシンタクスよりも簡単になるという点である。

多くの環境において、一連のタグが用意されているが、それぞれのニーズに合ったページを設計するため、独自のタグやマークを定義する機能を持ったプラットフォームが増えつつある。

最も一般的な**テンプレートビュー**の形式は、ASP、JSP、PHPなどの**サーバページ**である。これらは基本的な**テンプレートビュー**形式よりはるかに進化していて、自由度の高いプログラミングロジック（**スクリプトレット**と呼ばれる）をページに組み込めるようになっている。しかし、私の考えでは、こうした機能は数多くの問題を含んでいて、サーバページ技術を利用する場合、基本的な**テンプレートビュー**の振る舞いに限定することを推奨する。

ページにスクリプトレットを配置することの明らかなデメリットは、プログラマ以外のユーザがページを編集できなくなる点が挙げられる。ページ設計にグラフィックデザイナーを起用する場合などには、重要な問題である。しかし、ページへのスクリプトレット埋め込みによる最大の問題は、ページがプログラムの貧弱なモジュールとなってしまう点である。オブジェクト指向言語を使っても、そうしたページ構成方法では、オブジェクト指向スタイルやプロシージャルスタイルによるモジュラー設計の構造的機能の大部分が失われてしまう。

さらに悪いことには、ページにスクリプトレットを詰め込み過ぎると、エンタープライズアプリケーションの異なるレイヤが紛れ込みやすくなる。サーバページ上にドメインロジックが出現し始めると、異なるサーバページが重複しやすくなり、ドメインロジックをうまく構築することが難しくなる。過去数年間に私が目にした最悪のコードは、サーバページコー

ドである。

14.4.1.2 ■ ヘルパーオブジェクト

スクリプトレットを避けるために重要なのは、各ページに対するヘルパーとしての標準オブジェクトを提供することである。ヘルパーは、あらゆる現実のプログラミングロジックを持つ。ヘルパーに対する呼び出しだけを持つことで、ページは簡略化され、テンプレートビューは保たれる。簡潔であるためプログラマ以外のユーザによるページの編集ができ、プログラマはヘルパーだけに専念できるようになる。使うツールによっては、ページ上のテンプレートをHTML/XMLタグへと変換することができ、その結果、ページの一貫性は保たれツールサポートを受けやすくなる。

これは簡単で優れた原則のように思えるが、状況を複雑にする問題点もいくつかある。最もシンプルなマークとは、システムのある場所から情報を取り出してページの正しい位置に収めるものである。マークはヘルパーの呼び出しに変換され、結局はテキストとなり（またはテキストへ変換される何かになり）、エンジンは、テキストをページ上に配置することになる。

14.4.1.3 ■ 条件付き表示

より複雑な問題が、条件付きページの振る舞いである。最も簡単な例としては、たとえば条件が真のときだけ、あるものが表示されるような状況が挙げられる。たとえば、`<IF condition = "$pricedrop > 0.1"> ...show some stuff </IF>` のような行に見られる条件付きタグである。ここでの問題は、このような条件付タグを使い始めると、やがてテンプレートからプログラミング言語への道をたどり始めてしまう点である。その結果、ページにスクリプトレットを埋め込む場合と同様の問題に直面することになる。完全なプログラミング言語が必要だと言うならスクリプトレットを使う手もあるが、スクリプトレットについての私の考えはすでに述べたとおりである。

つまり、私は条件付きタグにも一抹の不安を感じているので、あまり勧められない。条件付きタグが必要な状況もあるかもしれないが、できる限りふつうの`<IF>`タグよりも目的のはっきりした方法を見つけるべきである。

あるテキストを条件的に表示する場合の1つのオプションとしては、条件式をヘルパーに移すという方法が挙げられる。その場合、ページは必ず呼び出しの結果をヘルパーへと挿入する。条件式が真でなければ、ヘルパーは空の文字列を送り返すが、ロジックはヘルパーが持つ。こうした手法は、返されるテキストへのマークアップがない場合、またはブラウザーによって無視される空のマークアップを返せる場合に適している。

しかし、一番頻度の高いアイテムの名前を太字に設定し強調したいときには、こうした方法は適していない。つまり、名前は常に表示する必要があり、特別なマークアップを施した

いと考える場合もある。そうした処理を実現するには、ヘルパーにマークアップを生成させるという方法が挙げられる。こうすることで、ロジックはページの外部に置かれ、強調のための選択はページ設計者の手を離れプログラムコードにまかされる。

HTML の選択をページ設計者の手元に置くには、何らかの条件付きタグが必要となる。これは <IF> の先に目を向けることである。目指すべきは狙いのはっきりしたタグ（focused tag）であり、次のようなタグではなく、

```
<IF expression = "isHighSelling()"><B></IF>
<property name = "price"/>
<IF expression = "isHighSelling()"></B></IF>
```

次のようなタグにすべきである。

```
<highlight condition = "isHighSelling" style = "bold">
   <property name = "price"/>
</highlight>
```

いずれの場合も、ヘルパーの単独のブール型プロパティをベースに条件が満たされることが重要である。より複雑な式をページに挿入するのは、事実上ページ自体にロジックを組み込むことに等しい。

もう 1 つの例は、システムが実行されているロケールに依存する情報をページに配置する方法である。ロケールが、アメリカまたはカナダの場合だけテキストを表示する場合、次のようなタグにはならない。

```
<IF expression = "locale = 'US' || 'CA'"> ...special text </IF>
```

次のようなタグになる。

```
<locale includes = "US, CA"> ...special text </locale>
```

14.4.1.4 ■ イテレーション

コレクションに対する繰り返しは、同じような問題を起こす。各行がある注文の品目名に対応する表が必要な場合、各行ごとの情報の表示を簡単にできる構造が必要である。ここでは、コレクションタグに対する一般的な繰り返しを回避することは難しいが、ほとんど問題なく動作する。

もちろん、作業の対象となるタグは、自分が置かれている環境によって制限されることが多い。決まったテンプレートセットが提供される環境もあり、その場合は、前述のガイドライン

に従う以上に、より多くの制限を受けるだろう。しかし、他の環境では使うタグに関しては、より多くの選択肢が利用でき、さらには独自のタグライブラリを定義できるものもある。

14.4.1.5 ■ 処理の時期

テンプレートビューという名前から、パターンの基本機能が**モデルビューコントローラ**におけるビューの表示の役割であることは明らかである。さまざまなシステムにおいて、**テンプレートビュー**はビューの表示だけを行う。モデルプロセスをできる限り分離したいが、よりシンプルなシステムにおいては、コントローラとして利用することも、モデルとすることもできる。**テンプレートビュー**がビューの裏の役割を担う状況では、そうした役割をページではなくヘルパーによって処理させることが重要である。コントローラとモデルの役割は、プログラムロジックに関連し、プログラムロジックはヘルパーに置かれる必要がある。

あらゆるテンプレートシステムは、Web サーバによる追加処理を必要とする。これは、ページ作成後のコンパイル時に行うことも、また最初のリクエストにおけるページのコンパイル時に行うことも、あるいは、各リクスエトにおけるページの解釈時に行うこともできる。ただし、解釈に時間がかかる場合、3 番目の選択肢は勧められない。

テンプレートビューでは例外に注意すべきである。例外が Web コンテナに及ぶ場合、リダイレクトの代わりに、呼び出し側のブラウザーへ異例な出力を提供するという処理途中のページを目にすることがある。自分の Web サーバが、例外をどのように処理するか注視する必要がある。何か変な処理を行っているようであれば、例外をすべてヘルパークラスに処理させるのである（スクリプトレットを軽視するという理由もある）。

14.4.1.6 ■ スクリプトの利用

サーバページは**テンプレートビュー**の最も一般的な形式だが、**テンプレートビュー**スタイルのスクリプトを記述することもできる。私は、こうした方法で書かれた Perl を目にしたことがある。Perl の CGI で目立つのは、レスポンスに適切なタグを出力する機能呼び出しを持つことで、文字列の連結を回避する仕掛けである。こうすることで、自分のプログラミング言語を使ってスクリプトを記述し、プログラミングロジックによる print 文字列の散乱を回避できるのである。

14.4.2 | 使用するタイミング

モデルビューコントローラにおいてビューを実装する場合の主な選択肢には、**テンプレートビュー**と**トランスフォームビュー**がある。**テンプレートビュー**の長所は、ページ構造に注目してページのコンテンツを作成できる点である。これは多くの人にとって実践しやすく学習も容易である。とりわけプログラマはヘルパーを担当し、ページのレイアウトはグラ

フィックデザイナーが担当するという方法が最適である。

一方、**テンプレートビュー**には、2つの大きな弱点もある。1つ目は、標準的な実装では、ページに複雑なロジックが入り込みやすく、プログラマ以外のユーザにとっては、維持管理が困難になる点である。ページはできる限りシンプルかつ表示専用にして、ロジックの部分はヘルパーにまかせるといったルールに従う必要がある。2つ目の弱点は、**トランスフォームビュー**に比べて、**テンプレートビュー**はテストが困難だという点である。**テンプレートビュー**の実装のほとんどは、Webサーバとの利用を前提として設計されているため、それ以外のテストはとても困難もしくは不可能である。**トランスフォームビュー**の実装は、テスト用のハーネスへの設置が容易で、Webサーバなしでもテストができる。

ビューについて考える場合、**ツーステップビュー**についても考慮する必要がある。テンプレートスキームによっては、特定のタグによってこうしたパターンの実装が可能となることもある。しかし、**トランスフォームビュー**をベースにして実装する方法のほうが簡単かもしれない。**ツーステップビュー**が必要になったら、この点について考慮する必要がある。

14.4.3 ｜例：独立コントローラを持ったビューとしてのJSPの利用（Java）

JSPをビューとしてだけ利用する場合、サーブレットコンテナからではなく必ずコントローラから実行する。そのため、何を表示すべきかを判断するために必要なすべての情報をJSPに渡すことが重要である。最良の手段は、コントローラにヘルパーオブジェクトを作成させ、HTTPリクエストを使ってJSPに渡すことである。ここでは、**ページコントローラ**のところで示した簡単な表示例を使って、その方法を紹介する。サーブレットのWeb操作メソッドは以下のようになる。

```
class ArtistController...

   public void doGet(HttpServletRequest request, HttpServletResponse response)
         throws IOException, ServletException {
      Artist artist = Artist.findNamed(request.getParameter("name"));
      if (artist == null)
         forward("/MissingArtistError.jsp", request, response);
      else {
         request.setAttribute("helper", new ArtistHelper(artist));
         forward("/artist.jsp", request, response);
      }
   }
```

テンプレートビューに関する限り、重要な振る舞いとは、ヘルパーの作成とリクエストへの配置である。サーバページは、useBeanタグによってヘルパーへと到達できる。

```
<jsp:useBean id="helper" type="actionController.ArtistHelper" scope="request"/>
```

ヘルパーの準備ができると、それを使って表示に必要な情報へのアクセスができる。ヘルパーが必要とするモデル情報は、その作成時にすでに渡されている。

```
class ArtistHelper...

   private Artist artist;
   public ArtistHelper(Artist artist) {
      this.artist = artist;
   }
```

ヘルパーを利用することで、モデルから適切な情報を入手する。最もシンプルなケースでは、メソッドによって、シンプルなデータ（Artist 名など）を取得する。

```
class ArtistHelper...

   public String getName() {
      return artist.getName();
   }
```

その後、Java の式によって情報にアクセスする。

```
<B> <%=helper.getName()%></B>
```

あるいはプロパティを利用する。

```
<B><jsp:getProperty name="helper" property="name"/></B>
```

プロパティを使うか式を使うかは、誰が JSP を編集するかによって決まってくる。プログラマにとっては式の方がコンパクトで読みやすいが、HTML エディタはそうした式を扱えないこともある。プログラマ以外のユーザは、タグを好むだろう。タグは HTML の標準形式に適合し、エラーが発生する余地が少ない。

また、ヘルパーを利用することで、難しいスクリプトレットコードも回避できる。ある Artist のアルバムリストを表示する場合、ループの実行が必要になるが、そうした処理はサーバページのスクリプトレットによって実現できる。

```
<UL>
<%
   for (Iterator it = helper.getAlbums().iterator(); it.hasNext();) {
      Album album = (Album) it.next();%>
   <LI><%=album.getTitle()%></LI>
<% } %>
</UL>
```

率直に言って、このようなJavaとHTMLの混在はとても読みにくい。その他のオプションとしては、次のようにforループをヘルパーへ移動するという方法がある。

```
class ArtistHelper...

   public String getAlbumList() {
      StringBuffer result = new StringBuffer();
      result.append("<UL>");
      for (Iterator it = getAlbums().iterator(); it.hasNext();) {
         Album album = (Album) it.next();
         result.append("<LI>");
         result.append(album.getTitle());
         result.append("</LI>");
      }
      result.append("</UL>");
      return result.toString();
   }
   public List getAlbums() {
      return artist.getAlbums();
   }
```

HTMLの分量が少ないので、私としてもこれはわかりやすいと思う。また、プロパティを利用したリストの取得も可能となっている。ヘルパーにHTMLコードを配置する方法を好まない人は多い。私も好きではないが、スクリプトレットとどちらを選ぶかと問われれば、ヘルパーにHTMLを置く方法を選ぶ。

最良の手段は、繰り返しのための特殊なタグである。

```
<UL><tag:forEach host = "helper" collection = "albums" id = "each">
   <LI><jsp:getProperty name="each" property="title"/></LI>
</tag:forEach></UL>
```

これはヘルパーのJSPやHTMLにスクリプトレットを使わなくてすむという点で、より優れたオプションである。

14.4.4 | 例：ASP.NET サーバページ (C#)

ここでも、**ページコントローラ**のところで示した例を引き続き使う（362 ページ）。クリケットの試合の 1 イニングにおいて打者が獲得した得点を示すものである。クリケットに興味のない読者もいるだろうから、ここではこの偉大なるスポーツの真の魅力を語ることは敢えて避け、基本的な以下の 3 種類の情報だけを表示することにする。

- 試合を参照するための ID 番号
- どのチームの得点でどのイニングの得点かが表示されている
- 各打者の名前、得点、得点率（その打者への総ボール数を総得点で割った値）

こうした統計データの意味がわからなくても、心配する必要はない。クリケットには統計データが豊富にある。クリケットの最大の功績は、物好きな新聞雑誌に対してふんだんに統計データを提供していることだろう。

ページコントローラに関する論点は、Web リクエストをどう扱うかという点であるが、一言で言えば、コントローラおよびビューの両方として機能するオブジェクトは、aspx ASP .NET ページである。スクリプトレットを使わずにこのコントローラコードを作成するためには、クラスの裏に独立したコードを定義することである。

```
<%@ Page language="c#" Codebehind="bat.aspx.cs" AutoEventWireup="false"
   trace="False" Inherits="batsmen.BattingPage" %>
```

このページは、クラスの裏にあるコードのメソッドとプロパティに直接アクセスできる。さらに、裏のコードはリクエストを扱う page_Load メソッドを定義できる。ここで私は、テンプレートメソッド [Gang of Four] として**レイヤスーパータイプ**に page_Load を定義している。

```
class CricketPage...

      protected void Page_Load(object sender, System.EventArgs e) {
         db = new OleDbConnection(DB.ConnectionString);
         if (hasMissingParameters())
            errorTransfer (missingParameterMessage);
         DataSet ds = getData();
         if (hasNoData (ds))
            errorTransfer ("No data matches your request");
         applyDomainLogic (ds);
         DataBind();
         prepareUI(ds);
      }
```

テンプレートビューの目的のために、私はpage_Loadメソッドの最後の数行以外はすべて無視している。DataBindへの呼び出しによって、さまざまなページ変数は、ベースとなるデータソースと適切に結び付けられている。これはシンプルなケースのためのものだが、より複雑なケースでは、最後の行は特定ページの裏のコードにあるメソッドを呼び出すことで使うオブジェクトを用意する。

対応するID番号、チーム、イニングは、ページにとって単一の値である。それらはHTTPリクエストのパラメータとしてページに加わったものである。私は、クラスの裏にあるコードのプロパティを利用することで、これらの値を提供している。

```
class BattingPage...

    protected String team {
       get {return Request.Params["team"];}
    }
    protected String match {
       get {return Request.Params["match"];}
    }
    protected String innings {
       get {return Request.Params["innings"];}
    }
    protected String ordinalInnings{
       get {return (innings == "1") ? "1st" : "2nd";}
    }
```

プロパティを定義することで、ページのテキスト中での利用もできる。

```
<P>
   Match id:
   <asp:label id="matchLabel" Text="<%# match %>"
      runat="server" font-bold="True">
   </asp:label> 
</P>
<P>
   <asp:label id=teamLabel Text="<%# team %>"
      runat="server" font-bold="True">
   </asp:label> 
   <asp:Label id=inningsLabel Text="<%# ordinalInnings %>"
      runat="server">
   </asp:Label> innings</P>
<P>
```

この表はやや複雑だが、Visual Studioのグラフィック機能のおかげで簡単に動作する。

Visual Studio は、データセットの単独の表に結び付けることができるデータグリッドコントロールを提供する。私は、page_Load メソッドで呼び出される prepareUI メソッドで、このバインドを行っている。

```
class BattingPage...

      override protected void prepareUI(DataSet ds) {
         DataGrid1.DataSource = ds;
         DataGrid1.DataBind();
      }
```

Batting クラスは、データベースの batting テーブルにドメインロジックを提供する**テーブルモジュール**である。データプロパティは、**テーブルモジュール**のドメインロジックによって補強される表のデータである。ここで補強されるものとは得点率であり、データベースに保存されているものではなく計算によって求められる。

ASP.NET データグリッドによって、表の外観に関する情報とともに、Web ページに表示したい列の選択ができる。例では、名前（name）列、得点（runs）列、および得点率（rate）列を選択している。

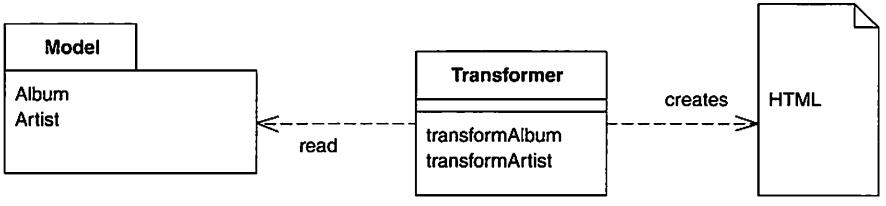
```
<asp:DataGrid id="DataGrid1" runat="server" Width="480px" Height="171px"
      BorderColor="#336666" BorderStyle="Double" BorderWidth="3px"
      BackColor="White" CellPadding="4" GridLines="Horizontal"
      AutoGenerateColumns="False">
   <SelectedItemStyle Font-Bold="True" ForeColor="White"
      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="name" HeaderText="Batsman">
         <HeaderStyle Width="70px"></HeaderStyle>
      </asp:BoundColumn>
      <asp:BoundColumn DataField="runs" HeaderText="Runs">
         <HeaderStyle Width="30px"></HeaderStyle>
      </asp:BoundColumn>
      <asp:BoundColumn DataField="rateString" HeaderText="Rate">
         <HeaderStyle Width="30px"></HeaderStyle>
      </asp:BoundColumn>
   </Columns>
   <PagerStyle HorizontalAlign="Center" ForeColor="White"
      BackColor="#336666" Mode="NumericPages"></PagerStyle>
</asp:DataGrid></P>
```

このデータグリッド用のHTMLの組み込みは一見大変そうだが、Visual Studioではそれらを直接操作するわけではなく、ページのその他の部分と同様に、開発環境でプロパティシートを介して操作するのである。

データセットおよびデータテーブルのADO.NETの抽象的概念を理解し、Webフォームコントロールを Webページに配置できるという能力は、このスキームの強みでもあり限界でもある。強みとしては、Visual Studioが提供するツールのおかげで、データセットを介して情報を転送できる点が挙げられる。一方、限界としては、**テーブルモジュール**のようなパターンを使うときだけ、シームレスな動作ができる点が挙げられる。とても複雑なドメインロジックを持っている場合**ドメインモデル**が役立つが、ツールを利用するためには**ドメインモデル**は独自のデータセットを作成しなければならない。

14.5 | トランスフォームビュー

要素ごとにドメインデータ要素を処理しHTMLへと変換するビュー。



ドメインおよびデータソースレイヤに対してデータリクエストを発行するとき、要件を満たすために必要なデータが返されるが、適切なWebページを作成するために必要なフォーマッティングは行われていない。**モデルビューコントローラ**におけるビューの役割は、データをWebページへと加工することである。**トランスフォームビュー**は、こうした処理を、モデルのデータを入力、HTMLを出力とする一種の変換と考える。

14.5.1 | 動作方法

トランスフォームビューの基本的な考え方は、ドメイン指向のデータに注目し、HTMLへと変換するプログラムを記述することである。プログラムはドメインデータの構造をチェックし、ドメインデータの各形式を確認しながら、HTMLの特定の要素へと書き出す。これを絶対的なものとするには、customerオブジェクトを受け取り、HTMLへと加工する`renderCustomer`というメソッドを持つ必要がある。顧客が多くの注文をしている場合、メソッドは、`renderOrder`を呼び出すことでその注文にループ処理を行うのである。

トランスフォームビューと**テンプレートビュー**の重要な違いは、ビューのまとめ方である。**テンプレートビュー**は、出力を中心に形成される。一方、**トランスフォームビュー**は、それぞれの入力要素に対する独立した変換を中心として形成される。変換は、各入力要素に注目し、要素に合った変換方法を判断し、そのための変換メソッドを呼び出すシンプルなループ処理で制御される。標準的な**トランスフォームビュー**のルールは、結果出力に影響することなく、いかなる順番にも並べ替えることができるのである。

トランスフォームビューはどんな言語でも記述できるが、現時点ではXSLTが優勢である。XSLTの興味深い点は、LispやHaskellなどの言語と同様に、決して国際規格の主流になることがなかった機能プログラミング言語だということである。そのため異なる種類の構造を持っている。たとえば、XSLTは明示的にルーチンを呼び出す代わりに、ドメインデータの要素を確認した後、適切な加工変換を実行する。

XSLT変換を実行するには、まずXMLデータから始める必要がある。変換が起こる一番シンプルな状況は、ドメインロジックの戻り型が、XMLまたは自動的にXMLに変換可能なもの（たとえば.NET）の場合である。それを行わない場合、自らXMLに直列化できる**データ変換オブジェクト**を投入することで、独自にXMLを生成する必要がある。このように、データは便利なAPIを使ってアセンブルできる。シンプルなケースでは、**トランザクションスクリプト**によってXMLを直接返すこともできる。

通信回線を混線させるために文字列形式が必要だというのでない限り、変換へ送られるXMLは文字列である必要はない。DOMを作成しそれを変換へと渡す方が迅速かつ簡単である。

XMLを用意できたら、それをXSLTエンジンに渡す（XSLTエンジンも最近では商用になってきた）。変換のためのロジックはXSLTスタイルシートに取り込まれ、また変換コードへと渡される。その後変換コードは、スタイルシートを入力であるXMLに適用して、出力であるHTMLを生成するが、これはHTTPレスポンスに直接書き出すことができる。

14.5.2 | 使用するタイミング

トランスフォームビュー、**テンプレートビュー**の選択は、ビューソフトウェアに携わるチームがどの環境を好むかによって決まることがほとんどであるため、ツールが重要な要因となる。**テンプレートビュー**の記述に利用できるHTMLエディタは、数多くある。一方、XSLT用のツールは、現時点では十分に成熟していない。またXSLT自体、機能プログラミングスタイルと複雑なXMLシンタクスとが組み合わされた結果、習得するのがとても困難な言語となっている。

一方、XSLTの長所は、他のWebプラットフォームへの移植性の高さである。J2EEまたは.NETから作成されたXMLの変換にも同じXSLTが利用できるが、これは異なる

ソースからのデータに共通のHTMLビューを配置するときに役立つ。

XMLドキュメントにビューを構築する場合も、XSLTは容易であることが多い。他の環境では、ドキュメントをオブジェクトに変換するよう求められたり、複雑なXML DOMの解析に多くの時間を費やされたりすることが多い。その点、XSLTはXMLの世界に自然と適合する。

トランスフォームビューは、**テンプレートビュー**が抱える2つの重大な問題を回避できる。変換の目的をHTMLへの加工に絞り込むことが容易であり、他のロジックがビューへと大量に注ぎこまれることを回避できる。また、**トランスフォームビュー**の実行は容易であるため、テスト用の出力の取得も簡単である。そのため、ビューのテストも簡単で、テストを行う場合もWebサーバを必要としない。

トランスフォームビューは、ドメイン指向のXMLをHTMLへと直接変換する。たとえば、Webサイトの外観全体を変更したいときは、複数の変換プログラムへの変更が必要である。XSLTが持つような標準的な変換コードは、こうした問題の解決に役立つ。**テンプレートビュー**を使うよりも、**トランスフォームビュー**を使って共通の変換コードを呼び出す方が簡単である。全体的な変更を加えたり、同一データに対する複数の表示方法をサポートしたりする必要がある場合は、2段階のプロセスを使用する**ツーステップビュー**の利用も考えるべきである。

14.5.3 | 例：シンプルな変換（Java）

シンプルな変換のための設定には、レスポンスを形成する正しいスタイルシートを呼び出すJavaコードの準備作業も含まれる。またレスポンスに書式を設定するスタイルシートの準備も含まれる。ページに対するほとんどのレスポンスは標準的であり、**フロントコントローラ**を使うことで十分である。ここではコマンドについてだけ解説するが、コマンドオブジェクトがその他のリクエストに対するレスポンス操作にいかに適合するかについては、**フロントコントローラ**の項を参照してほしい。

コマンドオブジェクトが行う作業は、モデルのメソッドを呼び出してXML入力ドキュメントを取得し、XMLプロセッサを介してXMLドキュメントを渡すことだけである。

```
class AlbumCommand...

   public void process() {
      try {
         Album album = Album.findNamed(request.getParameter("name"));
         Assert.notNull(album);
         PrintWriter out = response.getWriter();
         XsltProcessor processor = new SingleStepXsltProcessor("album.xsl");
```

```
            out.print(processor.getTransformation(album.toXmlDocument()));
        } catch (Exception e) {
            throw new ApplicationException(e);
        }
    }
```

XML ドキュメントは以下のようになる。

```
    <album>
        <title>Stormcock</title>
        <artist>Roy Harper</artist>
        <trackList>
            <track><title>Hors d' Oeuvres</title><time>8:37</time></track>
            <track><title>The Same Old Rock</title><time>12:24</time></track>
            <track><title>One Man Rock and Roll
                Band</title><time>7:23</time></track>
            <track><title>Me and My Woman</title><time>13:01</time></track>
        </trackList>
    </album>
```

XML ドキュメントの変換は、XSLT プログラムによって行われる。各テンプレートは XML の特定の部分と一致し、適切な HTML 出力をページのために生成する。このケースでは、私は基本要素だけを示すため、出力書式は基本的なものにとどめている。以下のテンプレート句は、XML ファイルの基本要素と一致している。

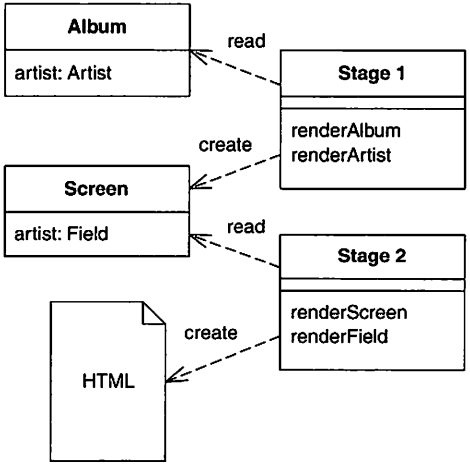
```
<xsl:template match="album">
   <HTML><BODY bgcolor="white">
   <xsl:apply-templates/>
   </BODY></HTML>
</xsl:template>
<xsl:template match="album/title">
   <h1><xsl:apply-templates/></h1>
</xsl:template>
<xsl:template match="artist">
      <P><B>Artist: </B><xsl:apply-templates/></P>
</xsl:template>
```

こうしたテンプレートの一致によって表は処理され、異なる色によって行を強調するような変更が加えられている。これはまさにスタイルシートだけでは不可能であり、XML を利用する必然性を示す良い例である。

```
<xsl:template match="trackList">
   <table><xsl:apply-templates/></table>
</xsl:template>
<xsl:template match="track">
   <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="track/title">
   <td><xsl:apply-templates/></td>
</xsl:template>
<xsl:template match="track/time">
   <td><xsl:apply-templates/></td>
</xsl:template>
```

14.6 | ツーステップビュー

ドメインデータを 2 つの手順で HTML に変換する。まずは論理ページを形成し、次に論理ページを HTML へと加工する。



Web アプリケーションが複数ページから構成される場合、外観および構成には一貫性が求められることが多い。各ページの外観が異なっていると、ユーザの混乱を招きやすいサイ

トとなってしまう。また、サイト全体に対して、容易に全面的な変更を加えたいこともある。しかし、**テンプレートビュー**や**トランスフォームビュー**による一般的な手法では、そうした変更は難しい。プレゼンテーションの決定が、複数のページまたはトランスフォームモジュールにまたがって重複することが多いからである。そうした大規模な変更を行うと複数のファイルの変更が必要になる。

ツーステップビューは、変換プロセスを2つの段階に分割することで、こうした問題に対処する。最初の段階では、特定の書式設定をせずに、モデルデータを論理的なプレゼンテーションへと変換する。2番目の段階では、論理的なプレゼンテーションを本来求められる出力書式に変換する。このようにして、全体的な変更が必要なら、2番目の段階を変更すればよい。あるいは、2番目の段階を複数用意することで、複数の外観を備えた出力もサポートできる。

14.6.1 | 動作方法

このパターンのポイントは、HTMLへの変換を2段階の処理として行う点である。最初の段階では、論理的な画面構造で情報を組み立てていて、HTMLがまだ含まれていない表示要素を構成する。第2段階では、そのプレゼンテーション指向の構造をHTMLへと加工する。

この中間フォームは、一種の論理画面である。要素には、フィールド、ヘッダー、フッター、表、オプションが含まれる。したがって、これはプレゼンテーション指向であるため、画面は確実に明確なスタイルに従うようになる。プレゼンテーション指向モデルは、所有可能なさまざまな仕掛けや含まれるデータを定義はするが、HTMLの外観は指定しないものと考えることができる。

こうしたプレゼンテーション指向の構造は、それぞれの画面用に記述された特定のコードによって組み立てられる。最初の段階の役割とは、ドメイン指向モデルにアクセスすることであり、データベース、実際のドメインモデル、あるいはドメイン指向の**データ変換オブジェクト**にアクセスし、画面に関連する情報を抽出して情報をプレゼンテーション指向の構造へと収めることである。

第2段階では、プレゼンテーション指向の構造をHTMLへと変換する。この段階では、プレゼンテーション指向の構造に含まれる各要素を把握し、HTMLとして表示する方法を把握している。そのため、多くの画面を持つシステムでも、第2段階だけですべて加工し、HTML書式の決定も一度で済む。ただし、その結果である画面は、プレゼンテーション指向の構造から派生する必要があるという制限は依然としてある。

第1段階のスタイルシートは、ドメイン指向のXMLをプレゼンテーション指向のXMLへと変換し、第2段階のスタイルシートは、XMLをHTMLへと変換する。

その場合、プレゼンテーション指向の構造を、テーブルクラス、行クラスなど一連のクラスとして定義する。第1段階では、ドメイン情報を取得し、論理画面をモデル化する構造へクラスをインスタンス化する。第2段階では、クラスをHTMLへ加工する。そのためには、プレゼンテーション指向の各クラスを取得し、独自にHTMLを生成するか、あるいはHTMLへの加工を行う独立したクラスに作業を行わせる。

いずれの手法も**トランスフォームビュー**をベースとしている。**テンプレートビュー**ベースの手法を利用することもできるが、その場合は、論理画面のアイデアをベースとしたテンプレートを選択する。

```
<field label = "Name" value = "getName" />
```

その後、テンプレートシステムは、論理タグをHTMLへと変換する。こうした仕組みにおいては、ページ定義にはHTMLは含まれず、論理画面タグだけが含まれる。結果としてそれはXML文書になり、WYSIWYG HTMLエディタを使う能力は失われることになる。

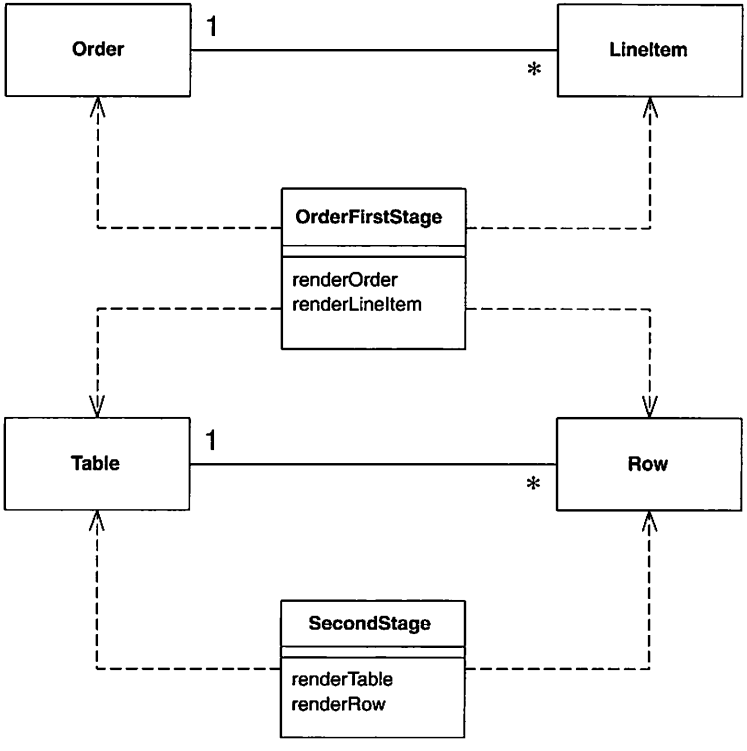


図14.4 ―― 2段階の加工のための例クラス

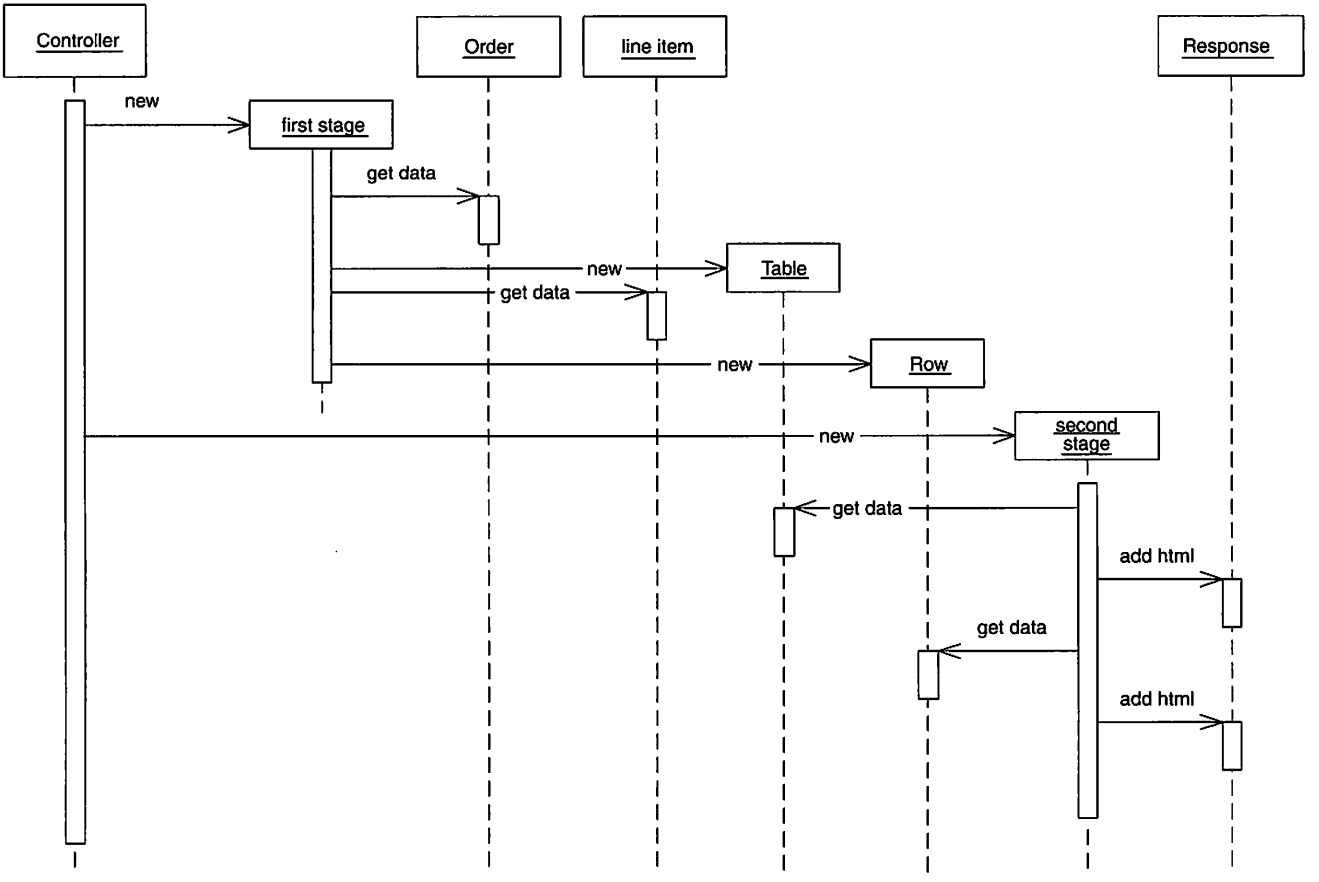


図 14.5 —— 2 段階の加工のためのシーケンス図

14.6.2 | 使用するタイミング

ツーステップビューの最も重要な点は、第1段階と第2段階とが独立しているために、全体的な変更が簡単に行えるという点にある。これは、複数の外観を持つWebアプリケーションと、単独の外観を持つWebアプリケーションの2つの状況を想定する手助けとなる。複数の外観を持つアプリケーションは、現在はかなり稀であるが、成長しつつある。そうしたアプリケーションでは、複数の組織に提供される基本機能はほとんど同じでも、外観は組織ごとに異なる。具体的な例としては、航空旅行者向けサイトが挙げられる。ページごとのレイアウトおよび設計は明らかに違っていても、すべては基本となるサイトが変化したものである。多くの航空会社は、明らかに異なる外観を除いて、ほとんど同じ機能を必要としているようである。

単独の外観を持つアプリケーションはとても一般的である。代表を務める組織は1つだけであり、彼らはサイト全体に渡って一貫した外観を求めている。まずはこの簡単な例から先に取り上げることにしよう。

シングルステージビュー（**テンプレートビュー**または**トランスフォームビュー**）の場合、Webページごとに1つのビューモジュールを構築する（図14.6参照）。**ツーステップビュー**には2つの段階がある。ページごとに1つの第1段階のモジュールと、アプリケーション全体に対して1つの第2段階のモジュールである（図14.7参照）。**ツーステップビュー**を使うメリットは、第2段階におけるサイトの外観の変更が容易であることだ。これは第2段階の1回の変更がサイト全体に影響するためである。

複数の外観を持つアプリケーションの場合は、画面と外観の組み合わせごとに第1段階のビューを持つため、こうしたメリットは大きくなる（図14.8）。たとえば、10個の画面と3個の外観は30個の第1段階ビューモジュールを必要とする。しかし、**ツーステップビュー**を使えば（図14.9参照）、10の第1段階と3つの第2段階で十分となる。画面および外観の数が増えるほど、その節約の効果は高まる。

ただし、成否はプレゼンテーション指向の構造をいかに外観のニーズに応えられるようにできるかにかかっている。ページごとに外観が異なるような設計重視のサイトでは、**ツーステップビュー**は効果を発揮できない。それは、シンプルなプレゼンテーション指向の構造が得られるだけの共通性を画面間に見出すことが難しいからである。基本的にサイト設計は、プレゼンテーション指向の構造によって制限されるが、そうした制限が多すぎるサイトは少なくない。

ツーステップビューのもう1つの弱点は、利用するためのツールが必要となることだ。プログラミング能力を持たないデザイナーが**テンプレートビュー**を使ってHTMLページのレイアウトを行うためのツールはたくさんあるが、**ツーステップビュー**では、プログラマはレンダリングオブジェクトとコントローラオブジェクトの記述を要求される。そのため、プログラマはあらゆる設計上の変更に関係を持つ必要が出てくるのである。

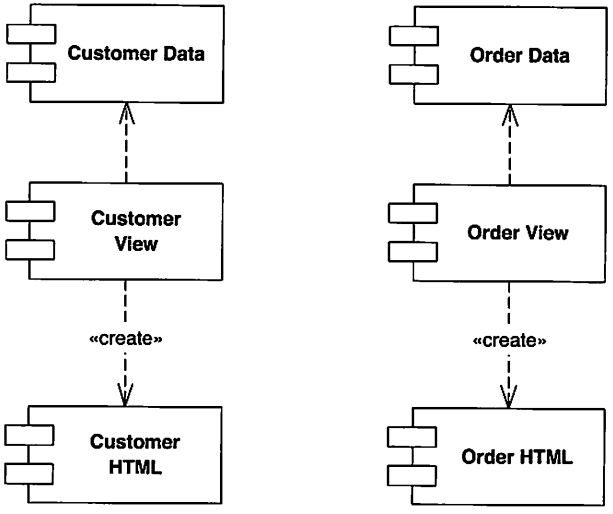


図 14.6 ——1 つの外観を持ったシングルステージビュー

複数レイヤを持つ**ツーステップビュー**が、習得困難なプログラミングモデルであることは事実である。しかし、慣れてしまえばそれほど難しくはなく、決まりきった反復的なコードを減らすこともできる。

複数の外観という点では、異なるデバイスに対する異なる第 2 段階の提供というバリエーションも考えられる。具体的には、ブラウザーと PDA ごとに異なる第 2 段階を用意することもできるのである。ここでの制限事項としては、いずれの外観も同じ論理画面に従わなければならないことである。そのため、まったく異なるデバイスの場合には行うのが難しい。

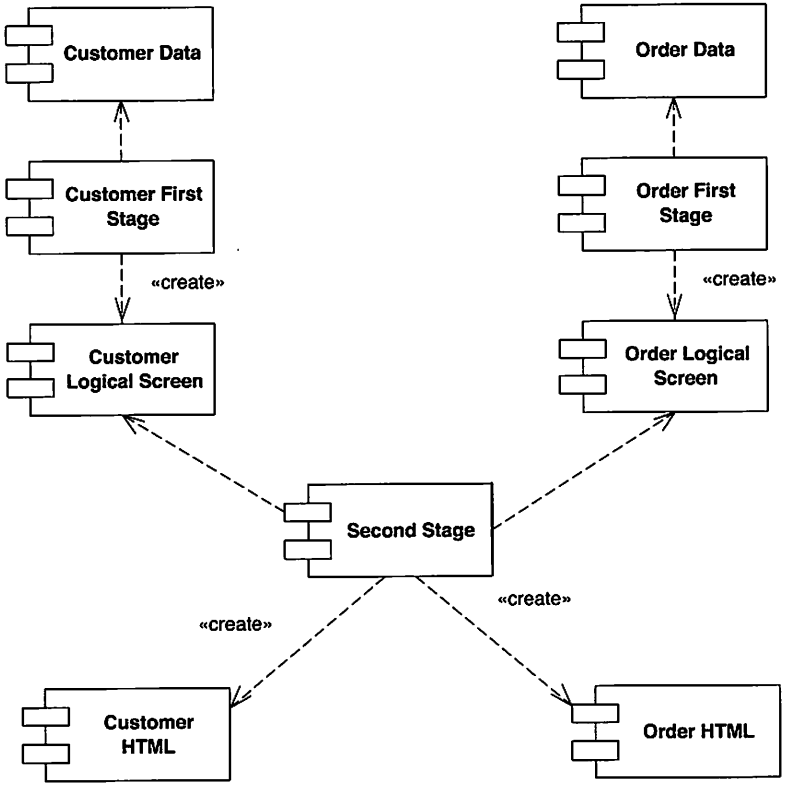


図 14.7 —— 1 つの外観を持ったツーステップビュー

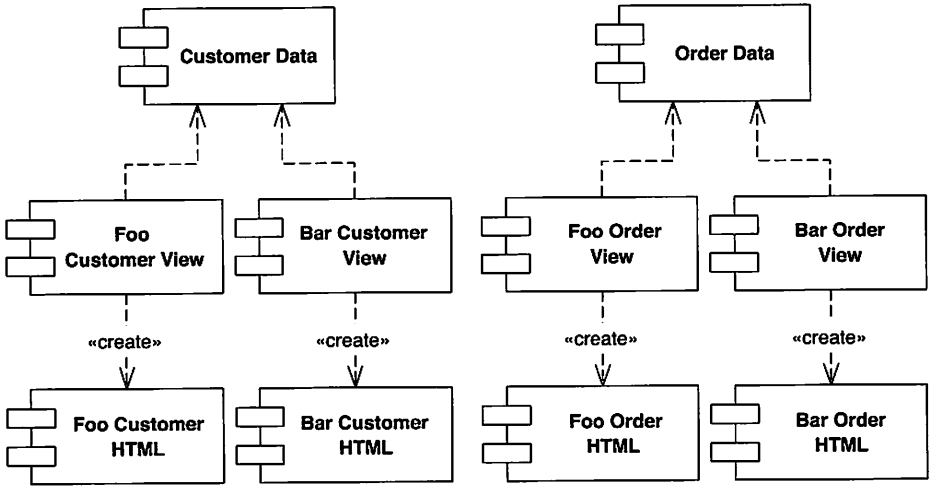


図 14.8 —— 2 つの外観を持ったシングルステージビュー

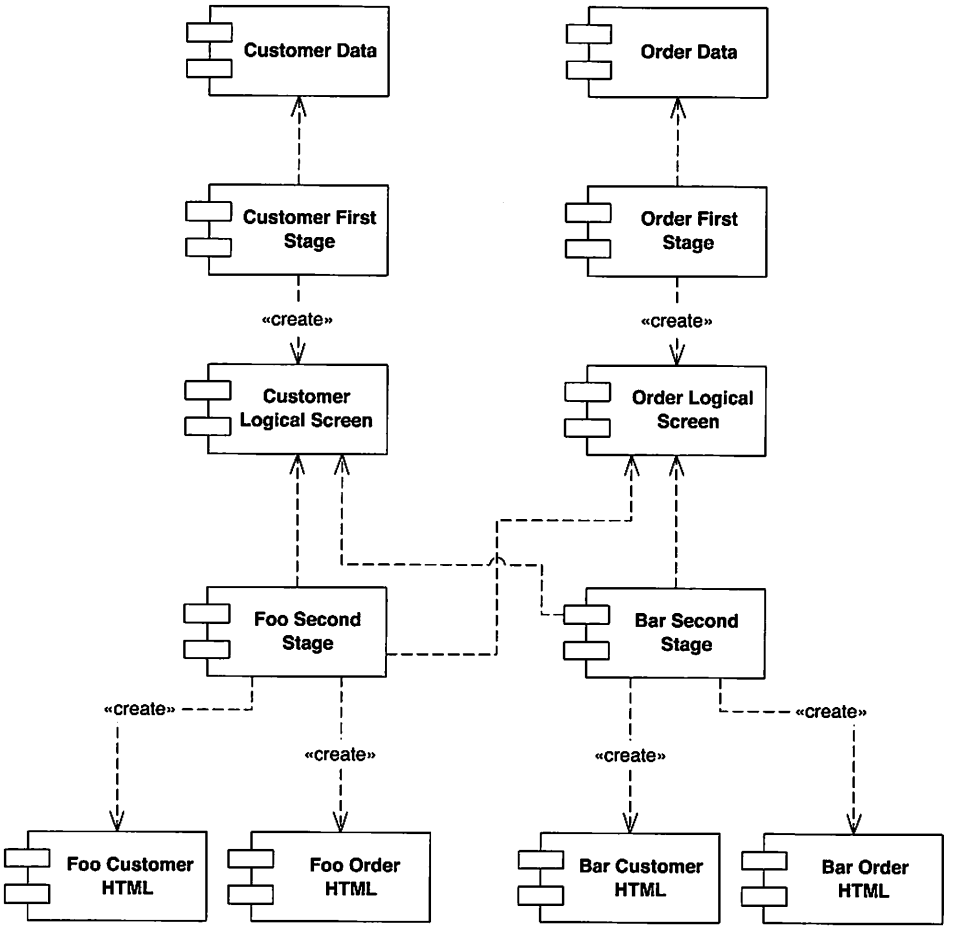


図 14.9 —— 2 つの外観を持ったツーステージビュー

14.6.3 | 例：ツーステージ XSLT（XSLT）

ツーステップビューのこの手法は、2 段階の XLST 変換を利用する。第 1 段階ではドメイン固有の XML を論理画面 XML へ変換し、第 2 段階では論理画面 XML を HTML へと変換する。

初期のドメイン指向 XML は、以下のようになる。

```
<album>
   <title>Zero Hour</title>
   <artist>Astor Piazzola</artist>
   <trackList>
      <track><title>Tanguedia III</title><time>4:39</time></track>
```

```
            <track><title>Milonga del Angel</title><time>6:30</time></track>
            <track><title>Concierto Para Quinteto</title><time>9:00</time></track>
            <track><title>Milonga Loca</title><time>3:05</time></track>
            <track><title>Michelangelo '70</title><time>2:50</time></track>
            <track><title>Contrabajisimo</title><time>10:18</time></track>
            <track><title>Mumuki</title><time>9:32</time></track>
        </trackList>
    </album>
```

第1段階のXSLTプロセッサは、以下のような画面指向のXMLへと変換する。これを行うためには、以下のXSLTプログラムが必要となる。

```
<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>
```

画面指向XMLはとても簡素である。これをHTMLに変換するには、第2段階のXSLTプログラムを利用する。

```
<xsl:template match="screen">
   <HTML><BODY bgcolor="white">
   <xsl:apply-templates/>
   </BODY></HTML>
</xsl:template>
```

```
<xsl:template match="title">
   <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>
```

2つの段階の組み立てにおいて、作業を行うコードを分離するための助けとして**フロントコントローラ**を利用している。

```
class AlbumCommand...

   public void process() {
      try {
         Album album = Album.findNamed(request.getParameter("name"));
         album = Album.findNamed("1234");
         Assert.notNull(album);
         PrintWriter out = response.getWriter();
         XsltProcessor processor = new
            TwoStepXsltProcessor("album2.xsl", "second.xsl");
         out.print(processor.getTransformation(album.toXmlDocument()));
      } catch (Exception e) {
         throw new ApplicationException(e);
      }
   }
```

これと、**トランスフォームビュー**における1段階手法とを比較することは有効である。たとえば、行の色を交互に変更したい場合、**トランスフォームビュー**ではすべてのXSLTプ

ログラムの変更が必要となるが、**ツーステップビュー**では、変更は第2段階のXSLTプログラム1つで済む。呼び出し可能なテンプレートによって同様の操作を行うことはできるかもしれないが、それにはかなりのXSLT能力が求められる。**ツーステップビュー**のデメリットとしては、最終的なHTMLが画面指向XMLの制限を強く受けてしまうという点が挙げられる。

14.6.4 ｜例：JSPおよびカスタムタグ（Java）

理論的には、XSTL方式は**ツーステップビュー**の実装を考える上で最も簡単な方法だが、その他の方法も数多くある。例では、私はJSPとカスタムタグを使っている。それらはXSLTに比べると面倒で非力だが、パターンがいかにさまざまな方法で自らを宣言できるかを示す例となっている。私はこうした方法が実際に行われているのを見たことがないため、この例の紹介については若干のためらいがある。しかし理論的な例によって、可能性を提示できると期待している。

ツーステップビューの基本ルールとは、表示対象の選択と表示するHTMLの選択とを完全に分離するというものである。この例では、第1段階はJSPページとヘルパーによって処理し、第2段階は一連のカスタムタグによって処理している。第1段階のポイントはJSPページである。

```
<%@ taglib uri="2step.tld" prefix = "2step" %>
<%@ page session="false"%>
<jsp:useBean id="helper" class="actionController.AlbumConHelper"/>
<%helper.init(request, response);%>
<2step:screen>
<2step:title><jsp:getProperty name = "helper" property =
    "title"/></2step:title>
<2step:field label = "Artist"><jsp:getProperty name = "helper" property
    = "artist"/></2step:field>
<2step:table host = "helper" collection = "trackList" columns = "title, time"/>
</2step:screen>
```

私はこのJSPページに対して、ヘルパーオブジェクトとともに、**ページコントローラ**を利用している（詳しくは**ページコントローラ**の項を参照）。ここでのポイントは、2stepネームスペースの一部であるタグに注目している点である。第2段階を呼び出すためにこれらを利用している。JSPページ上にはHTMLはなく、存在するタグは第2段階タグ、もしくはヘルパーから値を取得するbean操作タグである。

各第2段階タグは、論理画面要素に必要なHTMLを送り出すための実装を持っている。

最もシンプルな例はタイトルである。

```
class TitleTag...

   public int doStartTag() throws JspException {
      try {
         pageContext.getOut().print("<H1>");
      } catch (IOException e) {
         throw new JspException("unable to print start");
      }
      return EVAL_BODY_INCLUDE;
   }
   public int doEndTag() throws JspException {
      try {
         pageContext.getOut().print("</H1>");
      } catch (IOException e) {
         throw new JspException("unable to print end");
      }
      return EVAL_PAGE;
   }
```

カスタムタグは、タグ化されたテキストの最初と最後に呼び出されるフックメソッドを実装することで機能する。タグは、シンプルに <H1> タグで本体部分を囲んでいる。フィールドのように、より複雑なタグは属性を持つことができる。属性は、設定メソッドによってタグのクラスと結び付けられている。

```
class FieldTag...

   private String label; public void setLabel(String label) {
      this.label = label;
   }
```

値がセットされた後は、それを出力に利用できる。

```
class FieldTag...

   public int doStartTag() throws JspException {
      try {
         pageContext.getOut().print("<P>" + label + ": <B>");
      } catch (IOException e) {
         throw new JspException("unable to print start");
      }
```

```
        return EVAL_BODY_INCLUDE;
    }
    public int doEndTag() throws JspException {
        try {
            pageContext.getOut().print("</B></P>");
        } catch (IOException e) {
            throw new JspException("how are checked exceptions helping me here?");
        }
        return EVAL_PAGE;
    }
```

Table タグは最も高度なタグである。JSP の記述者が表に収める列を選択できるようにするだけでなく、行を交互に強調する。タグの実装は第 2 段階として機能するため、強調はそこで行われ、その結果、システム全体のグローバルな変更ができる。

Table タグは、コレクションプロパティの名前、コレクションがあるオブジェクト、そしてカンマ区切りの列名リストに対する属性を持つ。

```
class TableTag...

    private String collectionName;
    private String hostName;
    private String columns;
    public void setCollection(String collectionName) {
        this.collectionName = collectionName;
    }
    public void setHost(String hostName) {
        this.hostName = hostName;
    }
    public void setColumns(String columns) {
        this.columns = columns;
    }
```

私は、オブジェクトからプロパティを取得するためのヘルパーメソッドを作成した。「何かを取得する」メソッドを呼び出す代わりに、Java Bean をサポートするクラスを利用する方法については十分な根拠がある。しかし、この例にとっては前者が有効である。

```
class TableTag...

    private Object getProperty(Object obj, String property) throws JspException {
        try {
            String methodName = "get" + property.substring(0, 1).toUpperCase()
```

```
            + property.substring(1);
        Object result = obj.getClass().getMethod(methodName,
            null).invoke(obj, null);
        return result;
    } catch (Exception e) {
        throw new JspException("Unable to get property " + property +
            " from " + obj);
    }
}
```

このタグは本文を持たない。呼び出されると、リクエストプロパティから名前の付けられたコレクションを抽出し、コレクションに繰り返し処理を行って表の行を生成する。

```
class TableTag...

    public int doStartTag() throws JspException {
        try {
            JspWriter out = pageContext.getOut();
            out.print("<table>");
            Collection coll = (Collection) getPropertyFromAttribute(hostName,
                collectionName);
            Iterator rows = coll.iterator();
            int rowNumber = 0;
            while (rows.hasNext()) {
                out.print("<tr");
                if ((rowNumber++ % 2) == 0) out.print(" bgcolor = "
                    + HIGHLIGHT_COLOR);
                out.print(">");
                printCells(rows.next());
                out.print("</tr>");
            }
            out.print("</table>");
        } catch (IOException e) {
            throw new JspException("unable to print out");
        }
        return SKIP_BODY;
    }
    private Object getPropertyFromAttribute(String attribute,
        String property) throws JspException
    {
        Object hostObject = pageContext.findAttribute(attribute);
        if (hostObject == null) throw new JspException("Attribute
            " + attribute + " not found.");
        return getProperty(hostObject, property);
```

```
    }
    public static final String HIGHLIGHT_COLOR = "'linen'";
```

繰り返しの間、タグは一行おきにその背景色を設定することで行を強調する。

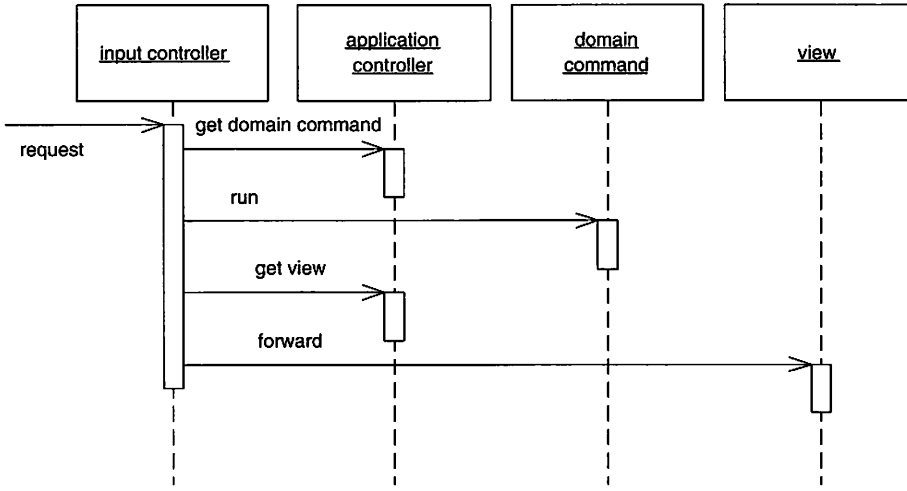
各行のセルを印刷する場合、私はコレクションのオブジェクトのプロパティ値を列名として使う。

```
class TableTag...

   private void printCells(Object obj) throws IOException, JspException {
      JspWriter out = pageContext.getOut();
      for (int i = 0; i < getColumnList().length; i++) {
         out.print("<td>");
         out.print(getProperty(obj, getColumnList()[i]));
         out.print("</td>");
      }
   }
   private String[] getColumnList() {
      StringTokenizer tk = new StringTokenizer(columns, ", ");
      String[] result = new String[tk.countTokens()];
      for (int i = 0; tk.hasMoreTokens(); i++) result[i] =
         tk.nextToken();
      return result;
   }
```

XSLT の実装と比べて、このソリューションは、サイトレイアウトの統一性に対する制限が少ない。ページごとに独立したHTMLを仕込みたいと考えるページ作者は、それが容易であることに気付くはずだ。もちろん、設計重視のページができる一方で、動作に詳しくない人が不適切に利用することもある。制限もときにはミスを防ぐ手助けとなる。これもまた開発チームが自ら判断すべきトレードオフなのである。

14.7 | アプリケーションコントローラ

アプリケーションの画面ナビゲーションとフローを扱うための集中管理ポイント。



アプリケーションによっては、異なるポイントごとに使われる画面に関する膨大な量のロジックが含まれ、その中には、アプリケーションにおける特定のタイミングでの特定の画面の呼び出しも含まれる。具体的には、一連の画面が一定の順番でユーザの前に提示される、いわゆるウイザード方式のやり取りが挙げられる。あるいは、一定の条件下においてだけ特定の画面が表示されたり、前の入力の内容によって表示される画面が変わったりすることもある。

モデルビューコントローラ入力コントローラは、このような決定を行うことができるが、アプリケーションがより複雑になると、異なる画面に対する複数のコントローラが特定の状況において行うべき操作を把握しなければならないため、コードの重複につながる。

こうした重複を回避するには、すべてのフローロジックを**アプリケーションコントローラ**に配置する。そして、入力コントローラは**アプリケーションコントローラ**に対して、モデルへの適切なコマンドの実行と、アプリケーションの状況に応じた正しいビューを要求する。

14.7.1 | 動作方法

アプリケーションコントローラは2つの役割を持つ。実行するドメインロジックの判断と、レスポンスを表示するビューの判断である。それを行うため、クラス参照の2種類の構造化されたコレクションを保持する（1つはドメインレイヤにおいて実行されるドメインコマンド用、もう1つはビュー用）（図14.10）。

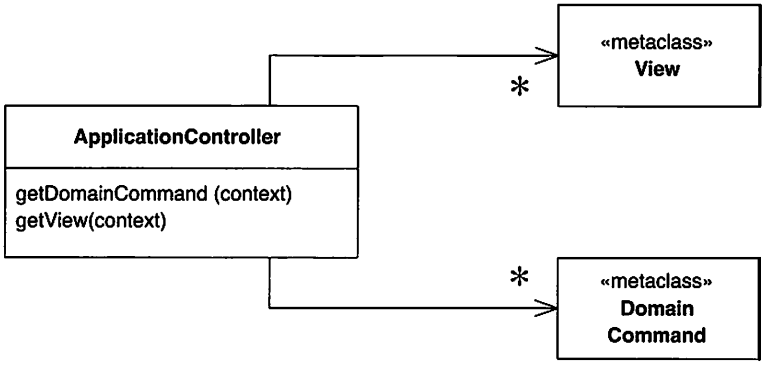


図14.10——アプリケーションコントローラは、ドメインロジック用とビュー用というクラスへの参照の2つのコレクションを持つ。

ドメインコマンドとビューの両方に対して、**アプリケーションコントローラ**は、呼び出しできる対象を保存する手段を必要とする。コマンド[Gang of Four]は、コードのブロックを容易に入手、実行できるので、良い選択肢である。関数を扱える言語も、それらに対する参照を保持することができる。その他の選択肢としては、リフレクションによるメソッドの呼び出しに使う文字列を持つという方法も挙げられる。

ドメインコマンドは、**アプリケーションコントローラ**レイヤの一部であるコマンドオブジェクトでも、**トランザクションスクリプト**への参照またはドメインレイヤのドメインオブジェクトメソッドでもかまわない。

サーバページをビューとして利用する場合、そのサーバページ名を使用できる。クラスを使う場合は、反復的な呼び出しのためのコマンドや文字列を使う。また、**アプリケーションコントローラ**が、参照としての文字列を保持できるXSLT変換も使える。

ここで下すべき判断は、**アプリケーションコントローラ**をプレゼンテーションの他の部分からどれくらい分離するかである。最初のレベルでは、**アプリケーションコントローラ**はそのUI機構に対して依存性を持つかどうかを判断することになる。おそらくそれはHTTPセッションデータに直接アクセスし、サーバページへと転送するか、あるいはリッチクライアントクラス上のメソッドを呼び出すか、という判断になる。

私はダイレクト**アプリケーションコントローラ**を目にしたことがあるが、個人的には**アプリケーションコントローラ**がUI機構へのリンクを持つ方が好みである。まず、UIとは切り離した形での**アプリケーションコントローラ**のテストができるが、これにはかなり大きなメリットがある。複数のプレゼンテーションに対して同じ**アプリケーションコントローラ**を使う場合にも、こうした方法は有効である。そのため**アプリケーションコントローラ**を、プレゼンテーションとドメインの中間レイヤと考える人も少なくない。

アプリケーションは、複数の**アプリケーションコントローラ**に個々の異なる部分を操作させることができる。これによって、複雑なロジックも複数のクラスへと分割できる。こうしたケースでは、作業はさまざまなUIの領域へと分割され、領域ごとに独立した**アプリケーションコントローラ**が構築される。シンプルなアプリケーションであれば、単独の**アプリケーションコントローラ**で十分であろう。

Webのフロントエンド、リッチクライアント、PDAなど複数のプレゼンテーションがある場合、各プレゼンテーションに同じ**アプリケーションコントローラ**を使うことは可能だが、あまり固執すべきではない。実用的なユーザインタフェースを実現するためには、異なるUIが異なる画面フローを必要とすることも多い。しかし、単独の**アプリケーションコントローラ**を再利用すれば、開発の手間が節約でき、その労力をより面倒なUIへと回すこともできる。

UIについての最も一般的な考え方は、アプリケーションの特定のキーオブジェクトの状態に応じて、特定のオブジェクトが異なるレスポンスのトリガーとなる状態マシンである。その場合、**アプリケーションコントローラ**は、メタデータを使って状態マシンの制御フローを表現することに適応しやすい。メタデータは、プログラミング言語の呼び出しによって設定することもでき（これが最も簡単な方法である）、独立したシステム構成ファイルに保存することもできる。

アプリケーションコントローラに置かれる1つのリクエストにだけ固有のドメインロジックを目にすることもあるかもしれない。私はこのような方法には大反対である。しかし、ドメインロジックとアプリケーションロジックの境界は極めて不透明である。たとえば、保険のアプリケーションにおいて、申込者が喫煙者の場合にだけ、別の質問のための画面を表示したいとしよう。これはアプリケーションロジックだろうか、それともドメインロジックだろうか。そうした場面が少なければロジックを**アプリケーションコントローラ**に収めることは可能だが、頻繁な場合は、それを誘導するような方法で**ドメインモデル**を設計する必要がある。

14.7.2 | 使用するタイミング

アプリケーションのフローおよびナビゲーションがあまりにシンプルで、どのような順番でも目的の画面にたどり着ける場合、**アプリケーションコントローラ**を利用する価値はほとんどない。**アプリケーションコントローラ**の長所は、表示されるページの順番に対する明確なルールと、オブジェクトの状態に応じてビューが異なる点にある。

アプリケーションのフローが変更されたときに、多くの異なる箇所に同じような変更を加える必要があると判断したなら、**アプリケーションコントローラ**を使うべきである。

14.7.3 | 参考文献

このパターンの記述のベースになっている概念は、ほとんど[Knight and Day]に由来する。そうした概念は必ずしも新しいものではないが、説明がわかりやすく説得力に富んでいる。

14.7.4 | 例：状態モデルアプリケーションコントローラ（Java）

状態モデルは、ユーザインタフェースを考える一般的な方法である。あるオブジェクトの状態に応じて、イベントに対する反応をさまざまに変える必要がある場合には特に有効である。この例では、私はある Asset（資産）に対する 2、3 のコマンドに対してシンプルな状態モデルを用意している（図 14.11）。ThoughtWork 社のリースの専門家がこのモデルを見たら、その非現実的なシンプルさに驚くかもしれないが、状態ベースのアプリケーションコントローラの例としては十分である。

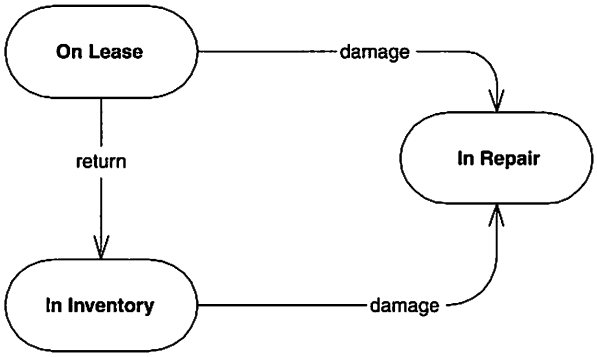


図 14.11 —— Asset（資産）のためのシンプルなステートチャート図

コードに関しては、以下のようなルールが設定されている。

- return コマンドを受け取ったとき、On Lease 状態にある場合、Asset の収益に関する情報を取得するページを表示する。
- In Inventory 状態にあるときのリターンイベントがエラーになる場合、不正アクションページを表示する。
- damage コマンドを受け取ったときには、Asset が In Inventory 状態にあるか、On Lease 状態にあるかに応じて、異なるページを表示する。

入力コントローラはフロントコントローラである。以下のようなリクエストに対してサービスを提供する。

```
class FrontServlet...

    public void service(HttpServletRequest request, HttpServletResponse
            response) throws IOException, ServletException
    {
        ApplicationController appController = getApplicationController(request);
        String commandString = (String) request.getParameter("command");
        DomainCommand comm =
            appController.getDomainCommand(commandString, getParameterMap(request));
        comm.run(getParameterMap(request));
        String viewPage = "/" + appController.getView(commandString,
                getParameterMap(request)) + ".jsp";
        forward(viewPage, request, response);
    }
```

service メソッドのフローは極めてシンプルである。まず任意のリクエストに対する正しいアプリケーションコントローラを判断して、アプリケーションコントローラにドメインコマンドを要求し、次にこのドメインコマンドを実行しアプリケーションコントローラにビューを要求して、最後にビューへと転送する。

スキームについては、私はすべてが同じインタフェースを実装する数多くの**アプリケーションコントローラ**を想定している。

```
interface ApplicationController...

    DomainCommand getDomainCommand (String commandString, Map params);
    String getView (String commandString, Map params);
```

コマンドに対して適切な**アプリケーションコントローラ**は、Asset アプリケーションコントローラである。Response クラスを使って、ドメインコマンドおよびビュー参照を保持する。私は、ドメインコマンドに対してはクラスへの参照を使い、ビューに対してはフロントコントローラが JSP 用の URL へと変換する文字列を使っている。

```
class Response...

    private Class domainCommand;
    private String viewUrl;
    public Response(Class domainCommand, String viewUrl) {
        this.domainCommand = domainCommand;
        this.viewUrl = viewUrl;
    }
```

```
    public DomainCommand getDomainCommand() {
        try {
            return (DomainCommand) domainCommand.newInstance();
        } catch (Exception e) {throw new ApplicationException (e);
        }
    }
    public String getViewUrl() {
        return viewUrl;
    }
```

アプリケーションコントローラは、コマンド文字列とAsset状態によってインデックス付けされたマッピングの1つを使って、レスポンスを維持している（図14.12）。

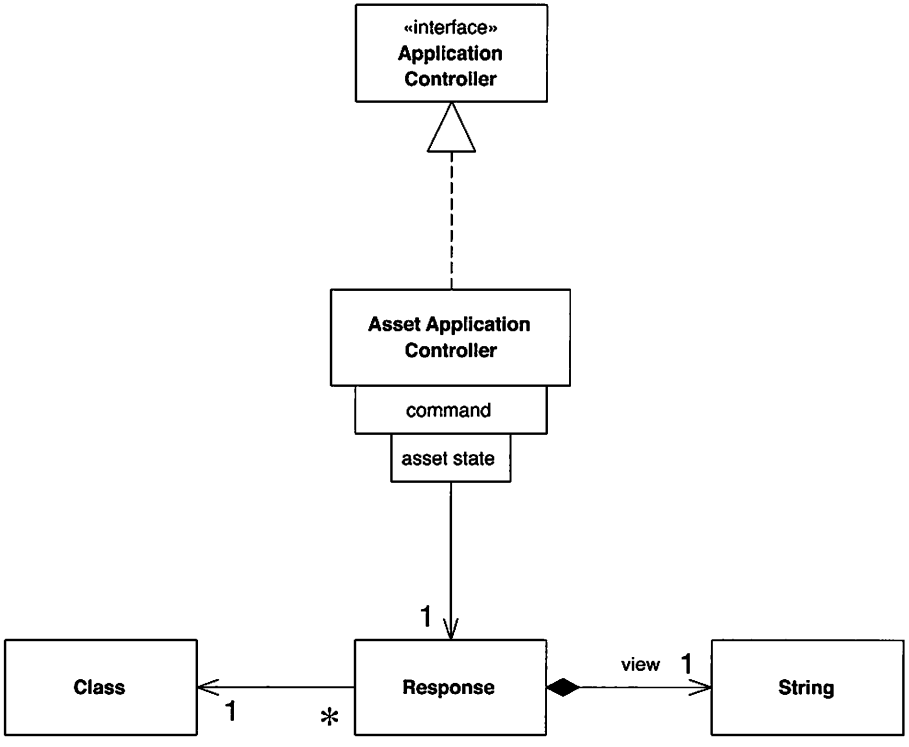


図14.12 —— Assetアプリケーションコントローラによるドメインコマンドとビューへの参照の保存方法

```
class AssetApplicationController...

    private Response getResponse(String commandString, AssetStatus state) {
        return (Response) getResponseMap(commandString).get(state);
    }
```

```
    private Map getResponseMap (String key) {
        return (Map) events.get(key);
    }
    private Map events = new HashMap();
```

ドメインコマンドを要求されると、コントローラはまずリクエストからAsset IDを調べ、次にドメインでAssetの状態を判断し、ドメインコマンドクラスを参照してクラスをインスタンス化し、新しいオブジェクトを返す。

```
class AssetApplicationController...

    public DomainCommand getDomainCommand (String commandString, Map params) {
        Response reponse = getResponse(commandString, getAssetStatus(params));
        return reponse.getDomainCommand();
    }
    private AssetStatus getAssetStatus(Map params) {
        String id = getParam("assetID", params);
        Asset asset = Asset.find(id);
        return asset.getStatus();
    }
    private String getParam(String key, Map params) {
        return ((String[]) params.get(key))[0];
    }
```

ドメインコマンドは、フロントコントローラによるコマンド実行できるシンプルなインタフェースに従う。

```
interface DomainCommand...

    abstract public void run(Map params);
```

ドメインコマンドが必要な操作を行った後、ビューを要求されると、**アプリケーションコントローラ**は再び活動を開始する。

```
class AssetApplicationController...

    public String getView (String commandString, Map params) {
        return getResponse(commandString, getAssetStatus(params)).getViewUrl();
    }
```

このケースでは、**アプリケーションコントローラ**は完全なURLをJSPに返さない。ある文字列が返されるが、フロントコントローラがURLへ変換しているものである。私がこうした処理を行っているのは、レスポンスにおけるURLパスの重複を防ぐためである。また、後でインダイレクションが必要となった場合、追加も容易になる。

アプリケーションコントローラは、コードによる利用の目的で読み込むことができる。

```
class AssetApplicationController...

   public void addResponse(String event, Object state, Class
         domainCommand, String view) {
      Response newResponse = new Response (domainCommand, view);
      if ( ! events.containsKey(event))
         events.put(event, new HashMap());
      getResponseMap(event).put(state, newResponse);
   }
   private static void
         loadApplicationController(AssetApplicationController appController) {
      appController = AssetApplicationController.getDefault();
      appController.addResponse("return", AssetStatus.ON_LEASE,
                        GatherReturnDetailsCommand.class, "return");
      appController.addResponse("return", AssetStatus.IN_INVENTORY,
                        NullAssetCommand.class, "illegalAction");
      appController.addResponse("damage", AssetStatus.ON_LEASE,
                        InventoryDamageCommand.class, "leaseDamage");
      appController.addResponse("damage", AssetStatus.IN_INVENTORY,
                        LeaseDamageCommand.class, "inventoryDamage");
   }
```

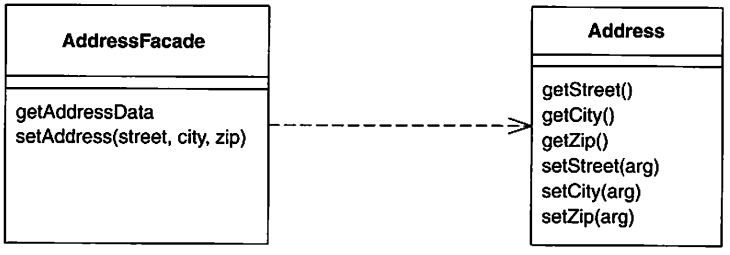
これをファイルから行うこともそれほど難しくはない。

第15章

分散パターン

15.1 リモートファサード

ネットワークの効率性を向上させるため、細かい粒度のオブジェクトに対して粗い粒度のファサードを提供する。



オブジェクト指向モデルでは、メソッドが少なく規模の小さなオブジェクトを使うのが最善である。その理由は、振る舞いの制御や変更を行ったり、適切な命名規則を設定したりすることで、理解しやすいアプリケーションを構築することが容易になるからである。このような細かい粒度の振る舞いとした結果、オブジェクト間の多大な相互作用が生じ、大量のメソッド呼び出しが必要になる。

1つのアドレススペース内では、細かい粒度の相互作用は有効に動作するが、プロセス間で呼び出しを行う場合には困難が生じる。リモートコールはとてもコストがかかる。データをマーシャリングしたり、セキュリティをチェックしたり、パケットを複数のスイッチを介してルーティングしたりするなどの操作が必要になるからである。2つのプロセスが互いに地球の反対側のマシン上で動作している場合、光LANの速さですらコスト要因になることがある。しかし相互のプロセス呼び出しは、両方のプロセスが同じマシン上にある場合でも、プロセス内の呼び出しと比べると桁ちがいにコストがかかる。それが効率性に及ぼす影響は、楽観的に考えても無視することができない。

その結果、リモートオブジェクトとして使う予定のあらゆるオブジェクトには、呼び出しの数を最小限にするインタフェースが必要になる。これは、メソッド呼び出しだけではなくオブジェクトにも影響する。Order と個々の Order Line を要求するのではなく、1 回の呼び出しで、Order と Order Line にアクセスし、更新する必要がある。これはオブジェクト構造全体に影響する。そのため、規模の小さいオブジェクトとメソッドで実行できた明確な目的と細かい粒度の制御をあきらめることになる。プログラミングは一層困難なものとなり、生産性も低下する。

リモートファサードとは、複雑化した細かい粒度のオブジェクトに対するファサード[Gang of Four]である。どの細かい粒度のオブジェクトにもリモートインタフェースはなく、**リモートファサード**はドメインロジックを一切含まない。

リモートファサードの機能は、メソッドを基本的な細かい粒度のオブジェクトに変換することである。

15.1.1 | 動作方法

リモートファサードは、分散の問題、つまり個別の責任を異なるオブジェクトに分けるという標準的なオブジェクト指向手法を使った解決策であり、標準的なパターンとなっている。細かい粒度のオブジェクトが複雑なロジックの適切な対処法であり、どのような複雑なロジックも、1 つのプロセス内でコラボレートするように設計された細かい粒度のオブジェクトに配置されるべきであると私は確信している。オブジェクトへのリモートアクセスの効率性を向上させるため、リモートインタフェースとして機能する個々のファサードオブジェクトを作成する。名前のとおり、ファサードは、単にインタフェースから細かい粒度のインタフェースへ変換するという薄い皮にしかすぎない。

アドレスオブジェクトのようなシンプルな例の場合、**リモートファサード**は、元のアドレスオブジェクトの get メソッドと set メソッドを、通常**バルクアクセッサー**と呼ばれる、1 つの get メソッドと 1 つの set メソッドに置き換える。クライアントがバルクセッターを呼び出すと、アドレスファサードは、set メソッドからデータを読み込み、本来のアドレスオブジェクトの個々のアクセッサーを呼び出し（図 15.1 参照）、処理は終了する。このように、アドレスオブジェクトは正確に組み込まれ、別の細かい粒度のオブジェクトが使えるようになり、アドレスオブジェクト上に妥当性確認と計算が構築される。

複雑な場合には、1 つの**リモートファサード**が多数の細かい粒度のオブジェクトのリモートゲートウェイとしての役割を果たす場合もある。たとえば、Order ファサードを使って、Order の情報、つまりすべての Order Line のほか、場合によっては何らかの顧客データに関する情報の取得と更新を行う。

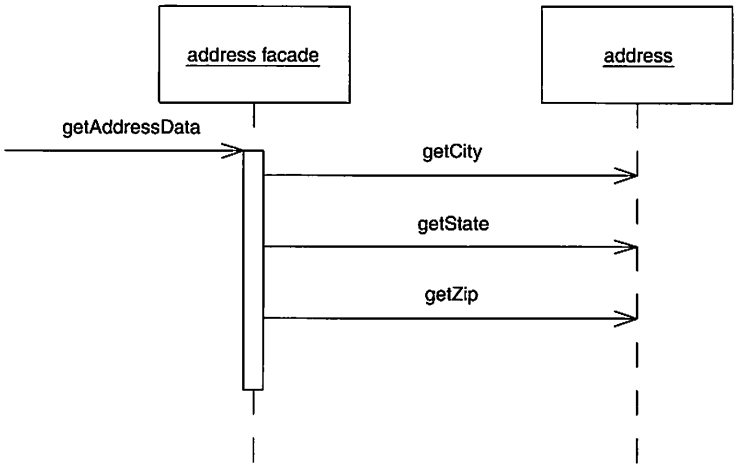


図 15.1 ——ファサードへの 1 つの呼び出しが、ファサードからドメインオブジェクトへの複数の呼び出しを行う

このように情報を一括して転送する際、回線上で容易に伝送できる形式にする必要がある。細かい粒度のクラスが接続の両端にあり、直接接続ができる場合、コピーして直接転送する。このような場合、getAddressData メソッドがオリジナルのアドレスオブジェクトのコピーを作成する。setAddressData は、アドレスオブジェクトを受け取り、それを使用してアドレスオブジェクトのデータを更新する（オリジナルのアドレスオブジェクトは一意性を保持する必要があり、容易には新しいアドレスに変更できないものとしている）。

しかし、普通上記の方法は使えない。その理由は、ドメインクラスを複数のプロセスに複製することを望まなかったり、複雑な関連構造であったりするために、ドメインモデルのセグメントを直列化することが困難な場合があるからである。クライアントは、モデル全体ではなく、単純化されたサブセットを必要とすることがある。このような場合、転送には基本的に**データ変換オブジェクト**を使うことが有効である。

スケッチでは、1 つのドメインオブジェクトに対応する**リモートファサード**を示している。これは目新しいものではなく、理解しやすいものであるが、必ずしも通常のケースとは限らない。1 つのファサードには、複数のオブジェクトから情報を伝えるように設計されたいくつかのメソッドがある。そのため、getAddressData と setAddressData は、getPurchasingHistory や updateCreditData で指定するメソッドも持つ、CustomerService のようなクラスで定義されるメソッドとなる。

粒度は、**リモートファサード**に関する最も困難な問題の 1 つである。ユースケースごとに 1 つなど、とても小さな**リモートファサード**を作成することを好む人もいる。私は、**リモートファサード**の少ない、とても限定した粗い粒度の構造を好む。普通サイズのアプリケーションの場合は 1 つだけ、大規模なアプリケーションでも 6 つ程度である。このことは、リ

モートファサードには多数のメソッドがあることを意味しているが、メソッドが小型であるため問題にはならないと思う。

ユーザインタフェースを介して情報の表示と更新を行うというユーザの特定のニーズに基づいて**リモートファサード**を設計するとする。このような場合、一連の画面用に1つの**リモートファサード**があり、各画面に対して、1つのバルクアクセッサーメソッドがデータのロードとセーブを行う。画面上のボタンを押すことで、Order状況の変更を指示すると、ファサードのコマンドメソッドが呼び出される。ほとんどの場合**リモートファサード**上には、基盤となるオブジェクトと同じ機能のある複数の異なるメソッドがある。これはよくあることで合理的である。ファサードは、システム内部ではなく、外部のユーザがわかりやすいように設計されている。そのため、クライアントプロセスが異なるコマンドと見なす場合、プロセスがまったく同じ内部コマンドに達している場合でも、異なるコマンドとなる。

リモートファサードをステートフルまたはステートレスにすることができる。ステートレス**リモートファサード**はプールできるので、特にB2Cにおいてリソースの使用頻度と効率性を向上させることができる。しかし、相互作用でセッションを超えた状態が呼び出される場合、セッションは**クライアントセッションステート**、**データベースセッションステート**、および**サーバセッションステート**の実装を使って、どこかにセッション状態を格納する必要がある。ステートフルの場合、**リモートファサード**はそれ自体の状態を持続できるので、**サーバセッションステート**の実装は容易になるが、何千ものユーザが同時に使用する場合は、パフォーマンスの問題が生じる可能性がある。

粗い粒度のインタフェースを提供するだけでなく、ほかのいくつかの役割を**リモートファサード**に追加することができる。たとえば、メソッドにセキュリティを適用するのは自然なことである。アクセス制御リストには、どのユーザがどのメソッドで呼び出しを行えるかを表示できる。**リモートファサード**メソッドは、さらにトランザクション制御を適用する適切な場所でもある。**リモートファサード**メソッドは、トランザクションの開始、内部動作のすべての実行、そしてトランザクションのコミットを行うことができる。各呼び出しは、クライアントに返すときにトランザクションをオープンにしたくないのでかなり長いものとなる。トランザクションがこのような長時間継続するケースでの効率性を考慮に入れて構築されていないからである。

リモートファサードに関して、私の知る限り最も大きな間違いの1つは、ドメインロジックを**リモートファサード**に配置することである。「**リモートファサード**にはドメインロジックを配置しない」と、私の後について3回繰り返して言ってほしい。いずれのファサードも、最低限の役割だけを持つ薄い皮であるべきだ。ワークフローや調整にドメインロジックが必要な場合、それを細かい粒度のオブジェクトに配置するか、あるいはそれを含めるために別個のリモート処理不可能な**トランザクションスクリプト**を作成する必要がある。**リモートファサード**を使うことなく、またはいずれかのコードの複製を要することなく、アプリケー

ション全体をローカルで動作させるべきだ。

15.1.1.1 ■ リモートファサードとセッションファサード

ここ2、3年、セッションファサード[Alur et al.]パターンがJ2EEコミュニティに登場してきた。私は、以前の草稿で、**リモートファサード**をセッションファサードと同じパターンと見なし、セッションファサードという名前を使っていた。しかし両者には決定的な違いがある。**リモートファサード**は、薄いリモートの皮を持つものにすぎない。そのため、**リモートファサード**にドメインロジックを配置することに対して私は警鐘を鳴らすのである。反対に、セッションファサードについてのほとんどの解説には、ロジックを、セッションファサード、ワークフロー分野のものに配置することが含まれている。大部分は、エンティティBeanをラップするためにJ2EEセッションBeanを使う一般的な手法によるものである。エンティティBeanのあらゆる連携は、エンティティBeanが再入可能ではないので、他のオブジェクトによって行う必要がある。

その結果、私は、セッションファサードをいくつかの**トランザクションスクリプト**をリモートインタフェースに配置したものと理解している。これは妥当な手法であるが、**リモートファサード**とは異なる。セッションファサードには、ドメインロジックが含まれるので、それは到底ファサードとは呼べないと主張したい。

15.1.1.2 ■ サービスレイヤ

ファサードに類似した概念が**サービスレイヤ**である。主な相違点は、サービスレイヤはリモートである必要はなく、そのため、細かい粒度のメソッド以外にも使うことができることである。シンプルにした**ドメインモデル**では通常はメソッドで終了するが、これはわかりやすさのためであり、ネットワークの効率性を考慮したものではない。さらに、サービスレイヤで**データ変換オブジェクト**を使う必要はない。サービスレイヤは、何の問題もなく、ドメインオブジェクトをクライアントに返すのである。

ドメインモデルがプロセス内とリモートの両方で使われる予定の場合、**サービスレイヤ**を持ち、その一番上に個別の**リモートファサード**をレイヤ化する。プロセスが、リモートだけで使われる場合、**サービスレイヤ**がアプリケーションロジックをまったく持たないという条件で、**サービスレイヤ**を**リモートファサード**に組み込む方が容易だと思われる。サービスレイヤに何らかのアプリケーションロジックがある場合、私なら**リモートファサード**を個別のオブジェクトにする。

15.1.2 | 使用するタイミング

細かい粒度のオブジェクトモデルへのリモートアクセスが必要な場合、いつでも**リモート**

ファサードを使うべきである。インタフェースのメリットを享受しながら、他方で細かい粒度のオブジェクトのメリットも保持されるなど、両者の最も良いところを活用する。

このパターンが最も普及しているのは、プレゼンテーションと**ドメインモデル**の間で、両者が異なるプロセスで動作している場合である。Swing UI とサーバドメインモデル間、またはアプリケーションと Web サーバが異なるプロセスの場合にはサーブレットとサーバオブジェクトモデルでこれを使うことができる。

ほとんどの場合、これは異なるマシン上の異なるプロセスで実行される。しかし、同じボックス上の相互のプロセス呼び出しコストは大きいので、プロセスがある場所に関係なく、相互のプロセス通信用の粗いインタフェースが必要である。

アクセスが 1 つのプロセス内である場合、変換は必要ないため、私はパターンを 1 つの Web サーバで動作するクライアント**ドメインモデル**とプレゼンテーション間または CGI スクリプトと**ドメインモデル**間の通信には使うつもりはない。**トランザクションスクリプト**は、**リモートファサード**を意識することはない。

リモートファサードは、分散の同期（つまりリモート手続コールの）形式である。非同期のメッセージベースのリモート通信を使うことで、アプリケーションのレスポンスを改善できる場合もあり、事実、非同期の手法には多くのメリットがある。残念ながら非同期パターンの解説は、本書の範囲を逸脱するものである。

15.1.3 ｜例：リモートファサードとしての Java セッション Bean の使用（Java）

エンタープライズ Java プラットフォームで作業している場合、分散ファサードの適切な選択はセッション Bean である。理由は、リモートオブジェクトでステートフルあるいはステートレスであるからだ。例では、EJB コンテナ内部で一群の POJO（Plain Old Java Object）を実行し、**リモートファサード**として設計されたセッション Bean を介しリモートでアクセスする。セッション Bean は、特に複雑なものではないので、以前に取り扱ったことがなくても理解できるはずである。

2 つの補足説明が必要である。まず私にとって驚きなのだが、Java では EJB コンテナ内部でプレーンオブジェクトを実行できないと信じている人が多いように思える。「ドメインオブジェクトはエンティティ Bean ですか」という質問を耳にする。答えは、そうではあるが必要はないということである。シンプルな Java オブジェクトは、例のようにとても有効に機能する。

もう 1 つの補足説明は、これがセッション Bean を使う唯一の方法ではないということである。セッション Bean は、**トランザクションスクリプト**のホスト機能としても使うことができる。

例では、音楽アルバムについての情報にアクセスするためのリモートインタフェースを考

える。**ドメインモデル**は、Artist、Album、Track を表すオブジェクトから構成されている。周辺にアプリケーションにデータソースを提供する他のいくつかのパッケージがある（図 15.2 参照）。

図 15.2 の dto パッケージには、回線を介してデータをクライアントに伝送する**データ変換オブジェクト**が含まれている。オブジェクトは、シンプルなアクセッサーとして振る舞い、さらにバイナリまたは XML テキストフォーマットで直列化する機能もある。リモートパッケージには、ドメインオブジェクトと**データ変換オブジェクト**間でデータを伝送するアセンブラオブジェクトがある。この機能の動作の詳細は、**データ変換オブジェクト**の項を参照してほしい。

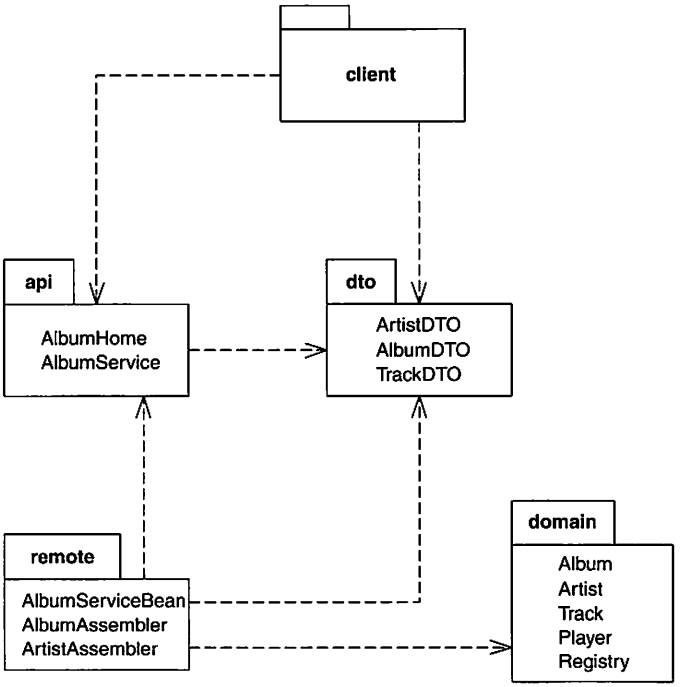


図 15.2 ——リモートインタフェースのパッケージ

ファサードの解説のため、データを**データ変換オブジェクト**に入出力して、リモートインタフェースを使えることとする。1 つの論理的な Java セッション Bean には、3 つの実体クラスがある。その中の 2 つはリモート API（実際は Java インタフェースでもある）を構成し、もう 1 つは API を実装するクラスである。2 つのインタフェースは、`AlbumService` とホームオブジェクトの `AlbumHome` である。ホームオブジェクトは、分散ファサードにアクセスするための命名サービスに使われるが、ここでは EJB の詳細は省略する。ここでの焦点

は、**リモートファサード**つまりAlbumServiceである。インタフェースは、クライアントが使えるようAPIパッケージで宣言されていて、単なるメソッドのリストである。

```
class AlbumService...

    String play(String id) throws RemoteException;
    String getAlbumXml(String id) throws RemoteException;
    AlbumDTO getAlbum(String id) throws RemoteException;
    void createAlbum(String id, String xml) throws RemoteException;
    void createAlbum(String id, AlbumDTO dto) throws RemoteException;
    void updateAlbum(String id, String xml) throws RemoteException;
    void updateAlbum(String id, AlbumDTO dto) throws RemoteException;
    void addArtistNamed(String id, String name) throws RemoteException;
    void addArtist(String id, String xml) throws RemoteException;
    void addArtist(String id, ArtistDTO dto) throws RemoteException;
    ArtistDTO getArtist(String id) throws RemoteException;
```

この短い例でさえ、**ドメインモデル**の中にArtistとAlbumという2つの異なるクラスのメソッドを確認できる。さらに、同じメソッドだが少しだけ異なるバリエーションも確認できる。データをリモートサービスに伝送するため、メソッドには**データ変換オブジェクト**またはXML文字列を使うバリエーションがある。クライアントは接続の環境に応じて使う形式を選択できる。小規模のアプリケーションの場合でもAlbumService上には多くのメソッドがある。

幸い、メソッド自体はとてもシンプルなものである。Albumを処理するメソッドは以下のとおりである。

```
class AlbumServiceBean...

    public AlbumDTO getAlbum(String id) throws RemoteException {
            return new AlbumAssembler().writeDTO(Registry.findAlbum(id));
    }
    public String getAlbumXml(String id) throws RemoteException {
            AlbumDTO dto = new AlbumAssembler().writeDTO(Registry.findAlbum(id));
            return dto.toXmlString();
    }
    public void createAlbum(String id, AlbumDTO dto) throws RemoteException {
            new AlbumAssembler().createAlbum(id, dto);
    }
    public void createAlbum(String id, String xml) throws RemoteException {
            AlbumDTO dto = AlbumDTO.readXmlString(xml);
            new AlbumAssembler().createAlbum(id, dto);
```

```
}
public void updateAlbum(String id, AlbumDTO dto) throws RemoteException {
        new AlbumAssembler().updateAlbum(id, dto);
}
public void updateAlbum(String id, String xml) throws RemoteException {
        AlbumDTO dto = AlbumDTO.readXmlString(xml);
        new AlbumAssembler().updateAlbum(id, dto);
}
```

メソッドは単に他のオブジェクトに委譲するだけなので、1行または2行に過ぎない。この断片は、分散ファサードがどうあるべきかを端的に表している。つまり、ロジックをほとんど含まないとても短いメソッドの長いリストである。ファサードは、単にメカニズムをパッケージ化したものである。

テストに関する数行でこの例を完了させよう。1つのプロセスで可能な限りのテストができるのはとても役立つ。セッションBeanの実装用のテストを直接記述できる。つまりEJBコンテナに配置することなく実行することができるのである。

```
class XmlTester...

    private AlbumDTO kob; private AlbumDTO newkob;
    private AlbumServiceBean facade = new AlbumServiceBean();
    protected void setUp() throws Exception {
        facade.initializeForTesting();
        kob = facade.getAlbum("kob");
        Writer buffer = new StringWriter();
        kob.toXmlString(buffer);
        newkob = AlbumDTO.readXmlString(new StringReader(buffer.toString()));
    }
    public void testArtist() {
        assertEquals(kob.getArtist(), newkob.getArtist());
    }
```

これはメモリで実行するJunitテストの1つであった。コンテナ外部で私がどのようにセッションBeanのインスタンスを作成し、テストを実行したかを示したもので、所要時間も短いテストである。

15.1.4 ｜例：Webサービス（C#）

私は本書についてAddison-Wesley社の編集者であるMike Hendricksonと相談した。常に最新の専門用語に注意するようにということで、彼は本書にWebサービスについての

用語があるかどうかと聞いてきた。私は何でも流行に走ることは嫌いだが、結局、本の出版の遅々としたペースを考えると、私が記述した「最新の手法」も読者が読むころには古風なものになっているかもしれない。最新のさまざまな技術的転換でも核心となるパターンは、その価値を保つことができるが、そのことを示すよい例になるかもしれない。

根本的にWebサービスは、（速度の遅い文字列解析ステップが付いてくるが）単にリモート用途のインタフェースである。したがってリモートファサードに対する基本的なアドバイスがそのまま当てはまる。細かな方法で機能を構築し、その上にWebサービスを処理するためのリモートファサードのレイヤを配置すればよい。

以前に解説した基本的な事項を使うが、ここでは1つのAlbum情報のリクエストだけを解説する。図15.3は、関与するさまざまなクラスを示している。これらのクラスは次のグループに分類される。Album Service（リモートファサード）、2つのデータ変換オブジェクト、ドメインモデルの3つのオブジェクト、およびドメインモデルからデータを引き出しデータ変換オブジェクトに挿入するアセンブラである。

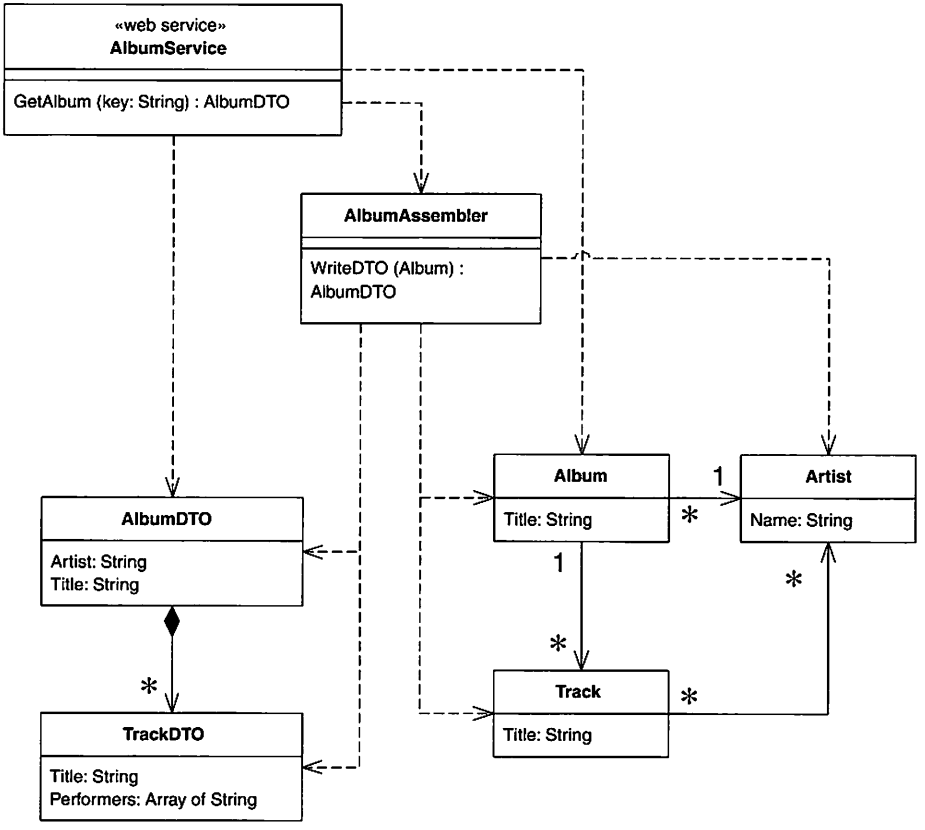


図15.3 —— Album（アルバム）Webサービスのクラス

ドメインモデルは、他に例をみないほどシンプルである。**テーブルデータゲートウェイ**を使って**データ変換オブジェクト**を直接作成した方が有効であるが、そうすると逆に、ドメインモデル上にレイヤ化した**リモートファサード**に悪影響を及ぼす。

```
class Album...

      public String Title;
      public Artist Artist; public IList Tracks {
         get {return ArrayList.ReadOnly(tracksData);}
      }
      public void AddTrack (Track arg) {
         tracksData.Add(arg);
      }
      public void RemoveTrack (Track arg) {
         tracksData.Remove(arg);
      }
      private IList tracksData = new ArrayList();

class Artist...

      public String Name;

class Track...

      public String Title;
      public IList Performers {
         get {return ArrayList.ReadOnly(performersData);}
      }
      public void AddPerformer (Artist arg) {
         performersData.Add(arg);
      }
      public void RemovePerformer (Artist arg) {
         performersData.Remove(arg);
      }
      private IList performersData = new ArrayList();
```

私は、回線を介してデータを渡すのに**データ変換オブジェクト**を使用する。これらは単にWebサービスのために構造を平坦化するデータホルダーである。

```
class AlbumDTO...

      public String Title;
```

```
        public String Artist;
        public TrackDTO[] Tracks;

class TrackDTO...

        public String Title;
        public String[] Performers;
```

これは.NETなので、直列化とXMLへの復元を行うためコードを記述する必要はまったくない。.NETフレームワークには、ジョブを実行するシリアライザクラスが付属している。

Webサービスなので、**データ変換オブジェクト**の構造をWSDLで宣言する必要もある。Visual StudioツールはWSDLを生成する。私は怠慢な方なので、このツールを使うつもりでいる。**データ変換オブジェクト**のXMLスキーマの定義は次のとおりである。

```
<s:complexType name="AlbumDTO">
   <s:sequence>
      <s:element minOccurs="1" maxOccurs="1" name="Title"
            nillable="true" type="s:string" />
      <s:element minOccurs="1" maxOccurs="1" name="Artist"
            nillable="true" type="s:string" />
      <s:element minOccurs="1" maxOccurs="1" name="Tracks"
            nillable="true" type="s0:ArrayOfTrackDTO" />
   </s:sequence>
</s:complexType>
<s:complexType name="ArrayOfTrackDTO">
   <s:sequence>
      <s:element minOccurs="0" maxOccurs="unbounded" name="TrackDTO"
            nillable="true" type="s0:TrackDTO" />
   </s:sequence>
</s:complexType>
<s:complexType name="TrackDTO">
   <s:sequence>
      <s:element minOccurs="1" maxOccurs="1" name="Title"
            nillable="true" type="s:string" />
      <s:element minOccurs="1" maxOccurs="1" name="Performers"
            nillable="true" type="s0:ArrayOfString" />
   </s:sequence>
</s:complexType>
<s:complexType name="ArrayOfString">
   <s:sequence>
      <s:element minOccurs="0" maxOccurs="unbounded" name="string"
            nillable="true" type="s:string" />
```

```
    </s:sequence>
  </s:complexType>
```

XML の詳細なデータ構造の定義であり、これでジョブが実行される。

ドメインモデルからデータ変換オブジェクトへデータを渡すためアセンブラが必要である。

```
class AlbumAssembler...

    public AlbumDTO WriteDTO (Album subject) {
        AlbumDTO result = new AlbumDTO();
        result.Artist = subject.Artist.Name;
        result.Title = subject.Title;
        ArrayList trackList = new ArrayList();
        foreach (Track t in subject.Tracks) trackList.Add (WriteTrack(t));
        result.Tracks = (TrackDTO[]) trackList.ToArray(typeof(TrackDTO));
        return result;
    }
    public TrackDTO WriteTrack (Track subject) {
        TrackDTO result = new TrackDTO();
        result.Title = subject.Title;
        result.Performers = new String[subject.Performers.Count];
        ArrayList performerList = new ArrayList();
        foreach (Artist a in subject.Performers) performerList.Add (a.Name);
        result.Performers = (String[]) performerList.ToArray(typeof (String));
        return result;
    }
```

最後に必要なのは、サービス定義である。最初に C# クラスから作成される。

```
class AlbumService...

    [ WebMethod ]
    public AlbumDTO GetAlbum(String key) {
        Album result = new AlbumFinder()[key];
        if (result == null)
            throw new SoapException ("unable to find album with key: " +
                        key, SoapException.ClientFaultCode);
        else return new AlbumAssembler().WriteDTO(result);
    }
```

もちろんこれは、WSDL ファイルから得られる本来のインタフェース定義ではない。関連する部分は、以下のとおりである。

```
<portType name="AlbumServiceSoap">
   <operation name="GetAlbum">
      <input message="s0:GetAlbumSoapIn" />
      <output message="s0:GetAlbumSoapOut" />
   </operation>
</portType>
<message name="GetAlbumSoapIn">
   <part name="parameters" element="s0:GetAlbum" />
</message>
<message name="GetAlbumSoapOut">
   <part name="parameters" element="s0:GetAlbumResponse" />
</message>
<s:element name="GetAlbum">
   <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="GetAlbumResponse">
   <s:complexType>
      <s:sequence>
         <s:element minOccurs="1" maxOccurs="1" name="GetAlbumResult"
               nillable="true" type="s0:AlbumDTO" />
      </s:sequence>
   </s:complexType>
</s:element>
```

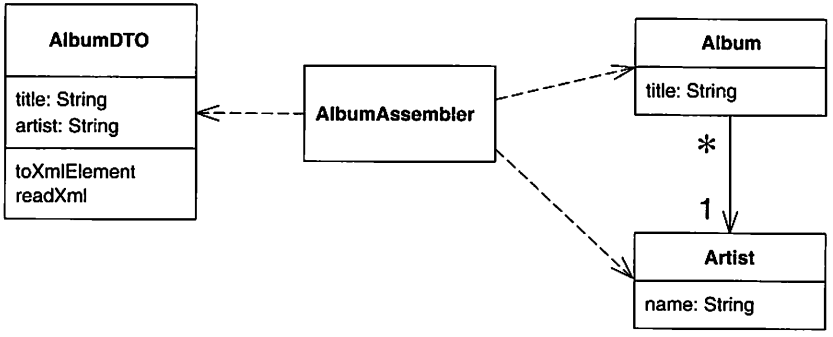
予想どおりWSDLは、冗長であるがジョブを実行する。これで私は、SOAPメッセージを送信してサービスを呼び出すことができる。

```
<?xml version="1.0" encoding="utf-8"?>
<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>
      <GetAlbum xmlns="http://martinfowler.com">
         <key>aKeyString</key>
      </GetAlbum>
   </soap:Body>
</soap:Envelope>
```

この例で忘れないでほしい重要な点は、これらは、SOAP と.NET を使った方法ではなく基本的なレイヤ化手法である。分散しないでアプリケーションを設計し、その一番上にリモートファサードとデータ変換オブジェクトを置いて分散機能をレイヤ化するのである。

15.2 | データ変換オブジェクト

メッセージ呼び出しの数を削減するため、プロセス間のデータを伝送するオブジェクト。



リモートファサードなどリモートインタフェースで処理をする場合、呼び出しには多大なコストがかかる。その結果、呼び出しの数を減らさなければならないが、これはそれぞれの呼び出しでより多くのデータを転送する必要があるということである。これを実行する 1 つの方法は、多くのパラメータを使うことだが、プログラムを複雑にする上、単一の値だけを返す Java などの言語では、プログラムできない。

この解決策は、呼び出しの全データを保持できるデータ変換オブジェクトを作成することである。それには、接続全体を直列化する必要がある。サーバ側では、データを DTO とドメインオブジェクト間で転送するのにアセンブラを使う。

Sun コミュニティの多くの人は、このパターンについて「バリューオブジェクト」という用語を使っている。私はこの用語を少し異なる意味で使っているので、508 ページの解説を参照することを勧める。

15.2.1 | 動作方法

広義の意味でデータ変換オブジェクトは、暗黙の了解オブジェクトとでも呼べるものである。これは単に一群のフィールドと get メソッドと set メソッドに過ぎないが、1 つの呼び出しで情報をネットワークを介し伝送できるので、分散システムでは不可欠である。

リモートオブジェクトが、何らかのデータを必要とする場合はいつでも、データ変換オブ

ジェクトを要求する。**データ変換オブジェクト**は、リモートオブジェクトがリクエストする以上のデータを保持できるが、今のところはリモートオブジェクトが必要とするデータを保持するだけでよい。リモートコールの待ち時間コストを考えても、複数の呼び出しをしてデータを送り過ぎる失敗の方がまだましである。

データ変換オブジェクトには、1つのサーバオブジェクトだけではなく、それ以上のデータが含まれている。これには、リモートオブジェクトがデータを要求しそうな、すべてのサーバオブジェクトからのデータが集積されている。したがって、リモートオブジェクトがOrderオブジェクトのデータをリクエストする場合、**データ変換オブジェクト**には、Order、Customer、Line Items、注文明細のProducts、Delivery Information（配送情報）など、あらゆる種類の要素のデータが含まれている。

通常は**ドメインモデル**内のオブジェクトを転送することはできない。オブジェクトは、複雑に絡み合って接続されているので、不可能ではないにしても直列化が困難というのがその理由である。さらに、たいていは、サーバ上のクライアントのドメインオブジェクトクラスを必要とはしない。これは**ドメインモデル**全体をコピーすることと同じになるからである。その代わりドメインオブジェクトをシンプルにした形式のデータを転送する必要がある。

データ変換オブジェクトのフィールドは、とてもシンプルでプリミティブであり、文字列や日付または他の**データ変換オブジェクト**のようなシンプルなクラスである。**データ変換オブジェクト**間の構造は、シンプルなグラフ構造（階層構造）で、**ドメインモデル**で解説する複雑なグラフ構造とは対照的である。**データ変換オブジェクト**は直列化可能であり、かつ回線の両端で理解される必要があるので、このようなシンプルな属性を保持しなければならない。結果として**データ変換オブジェクト**のクラスおよびそれらが参照するクラスは、両側に必要である。

特定のクライアントのニーズを考えても、**データ変換オブジェクト**を設計することは合理的である。そのような理由から、Webページ やGUI画面に対応した**データ変換オブジェクト**を見かけるのである。さらに、特定の画面に応じた複数の**データ変換オブジェクト**も見かけることがある。プレゼンテーションが異なっても同様のデータが必要な場合に、1つの**データ変換オブジェクト**を使って、処理する方が役立つ。

関連した問題で、相互作用全体に1つの**データ変換オブジェクト**を使うか、それともそれぞれのリクエストに対して異なる**データ変換オブジェクト**を使うかということがある。異なる**データ変換オブジェクト**を使うと、呼び出しでどのようなデータが転送されているかを把握しやすくなるが、**データ変換オブジェクト**の数が増加する。1つだけの場合は記述する手間は省けるが、呼び出しがどのように情報を転送するかを把握するのは困難になる。私は、データ全体に多くの共通点がある場合、1つだけを使うことにしている。しかし特定のリクエストでそれが提案されている場合には、異なる**データ変換オブジェクト**を使うことに躊躇しない。総括的な規則をつくることはできないからである。

その場合、ほとんどの相互作用においては同じデータ変換オブジェクトを使用し、特定のリクエストとレスポンスに対してだけ他のデータ変換オブジェクトを使用することになるだろう。

他の同様な問いとしては、リクエストとレスポンスの両方に1つのデータ変換オブジェクトを使うか、それともそれぞれに個別のデータ変換オブジェクトを使うかというものである。繰り返しになるが、総括的な規則はないので、データがほとんど同じであれば1つを使い、大きく異なれば2つ使うのである。

中には、データ変換オブジェクトを不変にしたいと考える人もいる。このスキーマでは、同じクラスの場合でさえクライアントから1つのデータ変換オブジェクトを受け取り、異なるデータ変換オブジェクトを作成し返信する。リクエストであるデータ変換オブジェクトを変更する人もいる。私には、どちらの方法がいいかについての明確な見解はないが、全般的には、レスポンス用に新しいオブジェクトを作成する場合でも、データを徐々に配置する方が容易なので、可変のデータ変換オブジェクトを好む。不変のデータ変換オブジェクトを支持する議論では、バリューオブジェクトとの命名の混乱を避ける必要がある。

データ変換オブジェクトの一般的な形式は、レコードセットの形式である。テーブルレコードのセットでSQLクエリーによって返されるものである。レコードセットは、SQLデータベースのデータ変換オブジェクトであり、アーキテクチャでは設計全体でそのレコードセットを使う。ドメインモデルは、クライアントに転送してデータのレコードセットを生成し、セットをSQLから直接受け取ったかのように処理する。レコードセット構造にバインドするツールがクライアントにあれば役立つ。レコードセットの全体をドメインロジックで作成することはできるが、それよりもプレゼンテーションに渡す前にSQLクエリーで生成し、ドメインロジックで変更する。この形式は、テーブルモジュールに適している。

データ変換オブジェクトの別の形式は、包括的なコレクションデータ構造である。私はこれに使われる配列を見たことがあるが、配列のインデックスがコードを不明瞭にしていて、失望した。最善のコレクションとは、キーとしてわかりやすい文字列を使うことができるディクショナリである。問題は、明示的なインタフェースと強制的な型定義のメリットを失うことである。手元に生成プログラムがない特定の状況では、ディクショナリを使う方が、手作業で明示的なオブジェクトを記述するより処理しやすくなる。しかし私は、生成プログラムで明示的なインタフェースを作成した方が賢明だと考える。特に、異なるコンポーネント間の通信プロトコルとして使うことを考える場合は尚更である。

15.2.1.1 ■ データ変換オブジェクトの直列化

シンプルなgetメソッドとsetメソッド以外に、データ変換オブジェクトはそれ自体を直列化して、回線上を移動して何らかのフォーマットにする機能もある。どのフォーマットかは、接続のいずれかの側に接続されているもの、接続自体を介して伝送ができるもの、直列化の簡素化の方法などに左右される。いくつかのプラットフォームでは、シンプルなオブ

ジェクトにビルトインの直列化機能を提供している。たとえばJavaには、ビルトインのバイナリ直列化機能があり、.NETには、ビルトインのバイナリとXMLの直列化機能がある。ビルトインの直列化を使う場合、**データ変換オブジェクト**は、ドメインモデルのオブジェクトにあるような複雑な処理をしないシンプルな構造なので、細かい粒度の設定をしなくてもすぐに動作する。そのため、私はできるだけ自動メカニズムを使う。

自動メカニズムがない場合、自分自身でメカニズムを作成することができる。シンプルなレコード記述を取り込み、データを保持する適切なクラスの生成、アクセッサーの提供、およびデータの直列化の読み書きを行ういくつかのコード生成プログラムを見たことがある。重要なことは、生成プログラムは必要な分だけにとどめ、自分だけが必要と考える機能を組み込まないようにすることである。最初のクラスは手作業で記述し、それを使って生成プログラムの記述に役立てることはよい考え方である。

さらに、リフレクティブプログラミングを直列化の処理に使うこともできる。この場合必要なのは、直列化ルーチンと非直列化ルーチンを一度記述し、スーパークラスに組み込むことである。これはパフォーマンスを低下させることがあるので、その場合は調査測定する必要がある。

その場合、接続の両端が協調するメカニズムを選択する。両端を制御している場合は、容易な方を選ぶ。そうでない場合は、所有していない方の終端にコネクターを供給できる。そして接続の両側でシンプルな**データ変換オブジェクト**を使い、外部のコードに適合させるためコネクターを使う。

データ変換オブジェクトで直面する一般的な問題の1つは、直列化形式をテキストまたはバイナリのどちらにするかということである。テキストの直列化の場合、読み込みが容易なので何が通信されているかもすぐにわかる。XMLは、簡単にXML文書の作成および解析を行うツールを入手できるので人気がある。テキストを使う場合の大きな欠点は、同じデータを送信するのに広いバンド幅が必要なため（特にXMLではそのようなことがある）、パフォーマンスが犠牲になりかなり深刻になる場合があることだ。

直列化で重要なのは、両側の**データ変換オブジェクト**の同期化である。理論的には、サーバが**データ変換オブジェクト**の定義を変更すると直ちに、クライアントも同様の更新をするはずだが、実際のところこれがうまくいかない場合がある。旧式のクライアントでサーバにアクセスすると必ず問題が発生するが、直列化のメカニズムが問題を多かれ少なかれ困難なものにすることがある。**データ変換オブジェクト**の純粋なバイナリ直列化の場合、構造に対するいかなる変更も通常非直列化でエラーの原因となるため、通信は完全に失われてしまう。オプションのフィールドを追加するなどのたいしたことのない変更でさえ、この影響を被ることがある。結果的に、直接的なバイナリ直列化で、通信回線の脆弱性が高くなるということである。

別の直列化スキーマを使うと、この状況を回避できる。その1つはXMLの直列化で、ク

ラスの変更に耐性を強化した方法で記述することができる。もう1つは、ディクショナリを使ったデータの直列化などの、耐性を強化したバイナリ手法である。私は**データ変換オブジェクト**としてのディクショナリを好まないが、一定の耐性を同期化に組み込むので、データをバイナリ直列化する場合の有効な方法と思われる。

15.2.1.2 ■ ドメインオブジェクトからのデータ変換オブジェクトの組み立て

データ変換オブジェクトは、ドメインオブジェクトとのつながりがわからない。それは、接続の両側に**データ変換オブジェクト**を配置するからである。このため、**データ変換オブジェクト**をドメインオブジェクトに依存させたくない。さらに、インタフェースフォーマットの変更時に**データ変換オブジェクト**の構造は変わるので、ドメインオブジェクトはこれに依存させたくない。原則的にドメインモデルは外部インタフェースから独立させている。

そのため、ドメインモデルからの**データ変換オブジェクト**の作成およびドメインモデルの更新の機能を果たす個別のアセンブラオブジェクトをつくる（図15.4）。アセンブラは、**データ変換オブジェクト**とドメインオブジェクト間のマッピングを行う**マッパー**の一例である。同じ**データ変換オブジェクト**を共有する複数のアセンブラが存在する場合もある。このよくある例は、同じデータを使う、異なるシナリオの更新セマンティクスである。アセンブラを独立させるもう1つの理由は、**データ変換オブジェクト**はシンプルなデータ記述から容易に自動的に生成できるためである。アセンブラの生成は困難で、普通は不可能である。

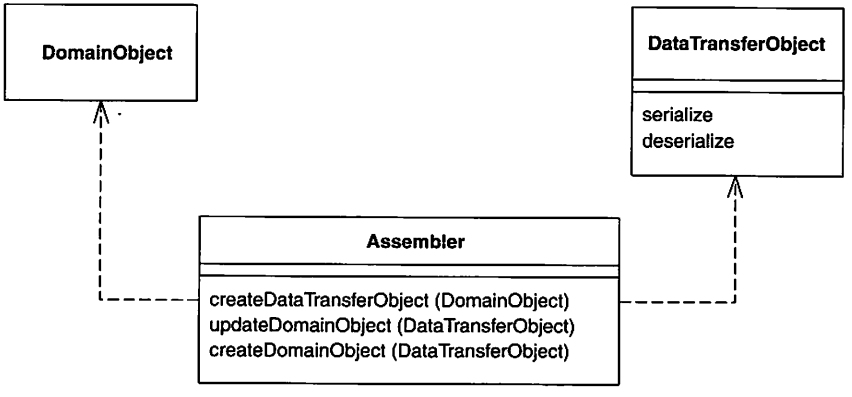


図15.4——アセンブラオブジェクトは、ドメインモデルとデータ変換オブジェクトを互いに独立させる。

15.2.2 | 使用するタイミング

2つのプロセス間の1つのメソッド呼び出しで、複数のデータ項目を転送する場合は、いつでも**データ変換オブジェクト**を使ってかまわない。

私が推奨するわけではないが、**データ変換オブジェクト**には、いくつかの代替方法がある。その1つは、オブジェクトをまったく使わないで、単に多くの引数を持つ set メソッドまたは受け渡し参照引数を持つ get メソッドを使う方法である。この方法の問題点は、Java などの多くの言語では戻り値として使えるオブジェクトが1つだけなので、更新には使えるが、適切なコールバックなしでは情報検索に使えないことである。

もう1つの代替方法は、インタフェースとして機能するオブジェクトを使わずに、何らかの形式の文字列表現を直接使う方法である。この方法の問題は、他も文字列表現と連結されることである。明示的なインタフェースの裏側の表現を隠ぺいするには良い方法で、文字列を変更したり、文字列をバイナリ構造に置き換えたりしたい場合、他に何も変更する必要はない。

特に、XML でコンポーネント間の通信をしたい場合、**データ変換オブジェクト**を作成する価値がある。XML DOM は、処理上の悩みの種であるが、**データ変換オブジェクト**はとても簡単に生成できるので、XML DOM のカプセル化にとても役立つ。

データ変換オブジェクトが使われる他の目的は、異なるレイヤのさまざまなコンポーネントの共通データソースとしての機能である。コンポーネントは、**データ変換オブジェクト**に何らかの変更を加え次のレイヤに渡す。COM や.NET における**レコードセット**はこの好例である。この場合、各レイヤはデータをベースとしたレコードセットの処理方法を認識していて、それが SQL データベースから直接取得したものか、または他のレイヤで変更されたものかどうかを認識できる。.NET は、レコードセットを XML に直列化するビルトインメカニズムを提供して機能を拡張している。

本書は同期システムに焦点を合わせているが、非同期システムに**データ変換オブジェクト**を使うのにも興味深いものがある。インタフェースを同期と非同期の両方で使いたい場合を取り上げる。同期の場合は、**データ変換オブジェクト**を返し、非同期の場合は、**データ変換オブジェクト**の**レイジーロード**を作成し、それを返す。非同期呼び出しが必要とされる箇所に**レイジーロード**を接続する。**データ変換オブジェクト**を使用する側は、非同期呼び出しの結果にアクセスした場合にだけブロックされる。

15.2.3 | 参考文献

[Alur et al.]は、**バリューオブジェクト**の名でパターンを検討しているが、これは前述した**データ変換オブジェクト**と同様である。私の**バリューオブジェクト**はまったく異なるパターンだ。同じ名前が異なる意味に使われているわけだが、ほとんどの人は私が使う意味でも**バリューオブジェクト**を使っている。言えることとしては、私が**データ変換オブジェクト**を使っているのは、J2EE コミュニティ内においてだけであり、普通は一般的な用法に従ってきたということである。

バリューオブジェクトアセンブラ[Alur et al.]は、アセンブラの検討事項である。私はマッパーベースの名前ではなくアセンブラ関連の名前を使うが、個別のパターンにすることはしない。

[Marinescu]はデータ変換オブジェクトの実装バリエーションについて検討し、[Riehle et al.]は直列化の異なる形式間の切り換えも含め、直列化の柔軟な方法について検討している。

15.2.4 | 例：Album についての情報の転送（Java）

例では、図 15.5 のドメインモデルを使う。転送したいデータは、リンクされたオブジェクトについてのデータで、データ変換オブジェクトの構造は図 15.6 に示している。

データ変換オブジェクトは、この構造をとてもシンプルにする。Artist クラスからの関連データは Album DTO に凝縮され、Track のパフォーマーは文字列で表現される。これは、データ変換オブジェクトの標準的な構造の凝縮である。

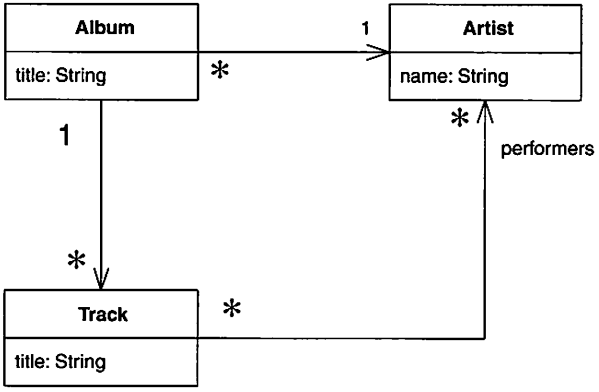


図 15.5 —— Artist と Album のクラス図

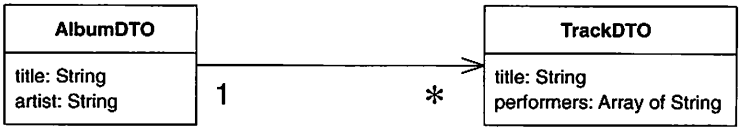


図 15.6 ——データ変換オブジェクトのクラス図

2 つのデータ変換オブジェクトがあり、1 つは Album 用で、もう 1 つは各 Track 用である。この場合、データは他の 2 つの内の 1 つに存在するので、Artist 用のデータ変換オブジェクトは必要ない。Album にはいくつかの Track があり、それぞれに 1 つ以上のデータ

項目が含まれている可能性があるので、変換オブジェクトとしてTrackがあれば十分である。

ドメインモデルからデータ変換オブジェクトを記述するためのコードは、以下のとおりである。アセンブラは、リモートファサードなど、リモートインタフェースを処理するいずれかのオブジェクトから呼び出される。

```
class AlbumAssembler...

   public AlbumDTO writeDTO(Album subject) {
      AlbumDTO result = new AlbumDTO();
      result.setTitle(subject.getTitle());
      result.setArtist(subject.getArtist().getName());
      writeTracks(result, subject);
      return result;
   }
   private void writeTracks(AlbumDTO result, Album subject) {
      List newTracks = new ArrayList();
      Iterator it = subject.getTracks().iterator();
      while (it.hasNext()) {
         TrackDTO newDTO = new TrackDTO();
         Track thisTrack = (Track) it.next();
         newDTO.setTitle(thisTrack.getTitle());
         writePerformers(newDTO, thisTrack);
         newTracks.add(newDTO);
      }
      result.setTracks((TrackDTO[]) newTracks.toArray(new TrackDTO[0]));
   }
   private void writePerformers(TrackDTO dto, Track subject) {
      List result = new ArrayList();
      Iterator it = subject.getPerformers().iterator();
      while (it.hasNext()) {
         Artist each = (Artist) it.next();
         result.add(each.getName());
      }
      dto.setPerformers((String[]) result.toArray(new String[0]));
   }
```

データ変換オブジェクトからモデルを更新するためには、普通はさらにコードが必要である。例では、新しいAlbumの作成と既存のAlbumの更新には違いがある。作成のためのコードは以下のとおりである。

```
class AlbumAssembler...

    public void createAlbum(String id, AlbumDTO source) {
        Artist artist = Registry.findArtistNamed(source.getArtist());
        if (artist == null)
            throw new RuntimeException("No artist named " + source.getArtist());
        Album album = new Album(source.getTitle(), artist);
        createTracks(source.getTracks(), album);
        Registry.addAlbum(id, album);
    }
    private void createTracks(TrackDTO[] tracks, Album album) {
        for (int i = 0; i < tracks.length; i++) {
            Track newTrack = new Track(tracks[i].getTitle());
            album.addTrack(newTrack);
            createPerformers(newTrack, tracks[i].getPerformers());
        }
    }
    private void createPerformers(Track newTrack, String[] performerArray) {
        for (int i = 0; i < performerArray.length; i++) {
            Artist performer = Registry.findArtistNamed(performerArray[i]);
            if (performer == null)
                throw new RuntimeException("No artist named " + performerArray[i]);
            newTrack.addPerformer(performer);
        }
    }
```

DTO の読み込みには、多くの決定事項が必要になる。注目は Artist の名前が入ってくる時の、その名前の処理方法である。私の要件としては、Album の作成時に、すでに Artist がレジストリにあることで、Artist が見つからない場合エラーとなる。異なる生成メソッドでは、**データ変換オブジェクト**で名前が言い渡されている時に、Artist を作成することにするものもある。

例では、既存の Album の更新には異なるメソッドを使っている。

```
class AlbumAssembler...

    public void updateAlbum(String id, AlbumDTO source) {
        Album current = Registry.findAlbum(id);
        if (current == null) throw new RuntimeException("Album does
            not exist: " + source.getTitle());
        if (source.getTitle() != current.getTitle())
            current.setTitle(source.getTitle());
        if (source.getArtist() != current.getArtist().getName()) {
```

```
            Artist artist = Registry.findArtistNamed(source.getArtist());
            if (artist == null) throw new RuntimeException("No artist
                named " + source.getArtist()); current.setArtist(artist);
        }
        updateTracks(source, current);
    }
    private void updateTracks(AlbumDTO source, Album current) {
        for (int i = 0; i < source.getTracks().length; i++) {
            current.getTrack(i).setTitle(source.getTrackDTO(i).getTitle());
            current.getTrack(i).clearPerformers();
            createPerformers(current.getTrack(i),
                source.getTrackDTO(i).getPerformers());
        }
    }
```

更新する際には、既存のドメインオブジェクトを更新するか、既存のものを破棄し、新しいものに置き換えるかを決定する。ここで生じる疑問は、更新したいオブジェクトを参照する他のオブジェクトがあるかどうかである。このコードの場合、Album と Track を参照する他のオブジェクトがあるので Album を更新するが、Track のタイトルとパフォーマーについては、単にそこにあるオブジェクトに置き換えている。

別の疑問は、Artist の変更についてである。既存の Artist の名前の変更だろうか、または Album のリンク先の Artist の変更だろうか。ケースバイケースで解決する必要があり、私はリンク先を新しい Artist にすることで処理している。

例では、ネイティブバイナリの直列化を使っているので、回線両側の**データ変換オブジェクト**の各クラスが同期を保つよう注意する必要がある。サーバの**データ変換オブジェクト**のデータ構造を変更し、クライアントは変更しないままの場合、転送でエラーが発生することもある。直列化にマッピングすることで、転送の耐性を強化することができる。

```
class TrackDTO...

    public Map writeMap() {
        Map result = new HashMap();
        result.put("title", title);
        result.put("performers", performers);
        return result;
    }
    public static TrackDTO readMap(Map arg) {
        TrackDTO result = new TrackDTO();
        result.title = (String) arg.get("title");
        result.performers = (String[]) arg.get("performers");
```

```
        return result;
    }
```

次に、フィールドをサーバに追加し、以前のクライアントを使う場合、新しいフィールドはクライアントが受け取らないがデータは正常に転送される。

このような直列化と非直列化のルーチンを記述することは、単調な作業である。私は**レイヤスーパータイプ**上などにある自己反映的ルーチンを使うことで、単調な作業の大半を回避している。

```
class DataTransferObject...

    public Map writeMapReflect() {
        Map result = null;
        try {
            Field[] fields = this.getClass().getDeclaredFields();
            result = new HashMap();
            for (int i = 0; i < fields.length; i++)
                result.put(fields[i].getName(), fields[i].get(this));
        } catch (Exception e) {throw new ApplicationException (e);
        }
        return result;
    }
    public static TrackDTO readMapReflect(Map arg) {
        TrackDTO result = new TrackDTO();
        try {
            Field[] fields = result.getClass().getDeclaredFields();
            for (int i = 0; i < fields.length; i++)
                fields[i].set(result, arg.get(fields[i].getName()));
        } catch (Exception e) {throw new ApplicationException (e);
        }
        return result;
    }
```

ルーチンで、ほとんどのケースが処理される（プリミティブを処理するにはコードを余分に追加する必要がある）。

15.2.5 ｜例：XML を使用する直列化（Java）

解説してきたとおり、Java による XML 処理はとても流動的で、不安定な API も全般的に良くなりつつある。読者がここまで読み進む頃には、本項は時代遅れの古い情報になって

いるかもしれないが、XMLへの変換の基本概念はほとんど同じである。

最初に、**データ変換オブジェクト**からデータ構造を入手し、直列化を決定する必要がある。Javaでは、マーカーインタフェースを使うことで、自由にバイナリ直列化を行える。**データ変換オブジェクト**の場合、作業は完全自動化されていて、私も最初に選択するのはこれである。しかし、テキストベースの直列化も必要とされる。例ではXMLを使っている。

この例の場合、XMLの処理がW3C規格のインタフェースより容易なので、JDOMを使っている。**データ変換オブジェクト**クラスであるクラスを表現するための、XML要素の読み書きをするメソッドを記述する。

```
class AlbumDTO...

    Element toXmlElement() {
        Element root = new Element("album");
        root.setAttribute("title", title);
        root.setAttribute("artist", artist);
        for (int i = 0; i < tracks.length; i++)
            root.addContent(tracks[i].toXmlElement());
        return root;
    }
    static AlbumDTO readXml(Element source) {
        AlbumDTO result = new AlbumDTO();
        result.setTitle(source.getAttributeValue("title"));
        result.setArtist(source.getAttributeValue("artist"));
        List trackList = new ArrayList();
        Iterator it = source.getChildren("track").iterator();
        while (it.hasNext())
            trackList.add(TrackDTO.readXml((Element) it.next()));
        result.setTracks((TrackDTO[]) trackList.toArray(new TrackDTO[0]));
        return result;
    }

class TrackDTO...
    Element toXmlElement() {
        Element result = new Element("track");
        result.setAttribute("title", title);
        for (int i = 0; i < performers.length; i++) {
            Element performerElement = new Element("performer");
            performerElement.setAttribute("name", performers[i]);
            result.addContent(performerElement);
        }
        return result;
    }
```

```
static TrackDTO readXml(Element arg) {
    TrackDTO result = new TrackDTO();
    result.setTitle(arg.getAttributeValue("title"));
    Iterator it = arg.getChildren("performer").iterator();
    List buffer = new ArrayList();
    while (it.hasNext()) {
        Element eachElement = (Element) it.next();
        buffer.add(eachElement.getAttributeValue("name"));
    }
    result.setPerformers((String[]) buffer.toArray(new String[0]));
    return result;
}
```

もちろんこれらのメソッドは、XML DOM の要素を作成するだけである。直列化を行うには、テキストを読み書きする必要がある。Track は、Album のコンテキストに合わせて転送されるので、必要なのはこの Album のコードを記述するだけである。

```
class AlbumDTO...

    public void toXmlString(Writer output) {
        Element root = toXmlElement();
        Document doc = new Document(root);
        XMLOutputter writer = new XMLOutputter();
        try {
            writer.output(doc, output);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    public static AlbumDTO readXmlString(Reader input) {
        try {
            SAXBuilder builder = new SAXBuilder();
            Document doc = builder.build(input);
            Element root = doc.getRootElement();
            AlbumDTO result = readXml(root);
            return result;
        } catch (Exception e) {
            e.printStackTrace();
            throw new RuntimeException();
        }
    }
```

難しいことではないと思うが、JAXBによってこの種の要素が不必要なものになることを期待している。

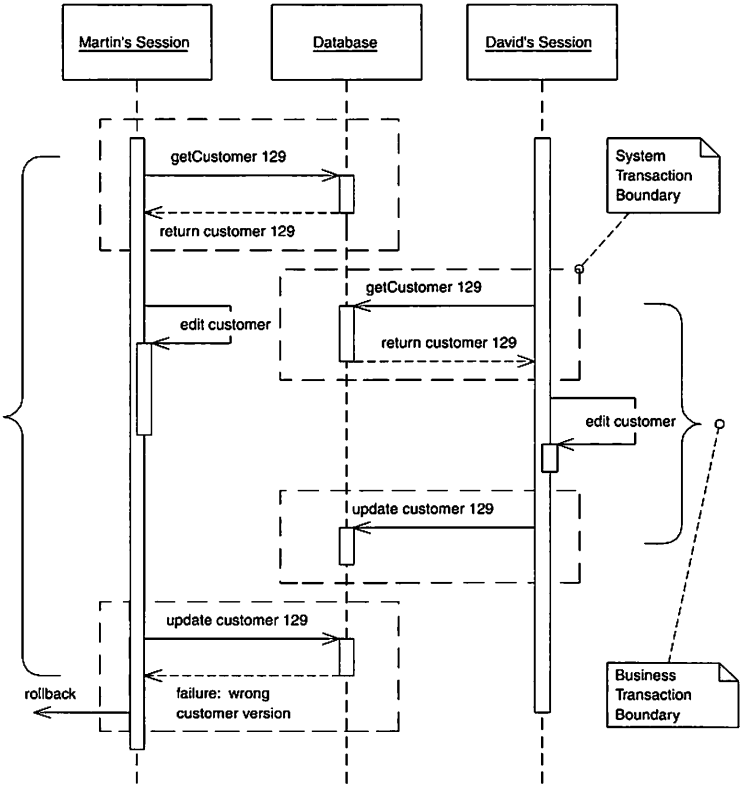
第16章

オフライン並行性パターン

16.1 軽オフラインロック

(by David Rice)

コンフリクトの検出とトランザクションのロールバックによって、同時に発生するビジネストランザクション間のコンフリクトを防止する。



ビジネストランザクションは、一連のシステムトランザクションにまたがって実行される場合が多い。一旦、1つのシステムトランザクションの領域外に出た場合、データベースマネージャーだけに頼っていたのでは、ビジネストランザクションが確実に一貫した状態でレコードデータを保持し続けることはできない。2つのセッションが同じレコードで処理を開始すると、データ整合性は危険な状態になり、更新が無効になる可能性が十分ある。さらに、あるセッションが読み込み中のデータを別のセッションが編集すると、同様に一貫性のない読み込みも発生する可能性がある。

軽オフラインロックは、この問題の解決策として、1つのセッションでコミット予定の変更が別のセッションの変更とコンフリクトしないよう妥当性確認を行う。事前コミット妥当性確認が正常に行われるということはつまり、レコードデータへの変更開始の承認を示すロックを取得できるということである。妥当性確認と更新が1つのシステムトランザクション内で発生する限り、ビジネストランザクションは一貫していると言える。

重オフラインロックは、セッションコンフリクトの可能性が高いので、システムの並行性を制限することを前提としているのに対し、**軽オフラインロック**はコンフリクトの可能性が低いことを前提としている。セッションコンフリクトが発生する可能性は少ないという予想の下で、複数のユーザが同時に同じデータを処理することができる。

16.1.1 | 動作方法

軽オフラインロックは、1つのセッションがレコードをロードし、別のセッションがレコードを修正していない時点で妥当性確認を行うことによって取得できる。**軽オフラインロック**はいつでも獲得できるが、取得したシステムトランザクションの間だけ有効である。したがって、ビジネストランザクションはレコードデータとコンフリクトしないように、データベースに変更を行うシステムトランザクション間の変更セットの各メンバに対して、**軽オフラインロック**を獲得する必要がある。

最も一般的な実装は、システムのバージョン番号と各レコードを関連付けることである。レコードがロードされると、番号は他のすべてのセッションステートとともにセッションによって保持される。**軽オフラインロック**を取得することは、セッションデータに格納されているバージョンとレコードデータの現行バージョンを比較することである。妥当性確認が正常に行われると、バージョンの増分を含むすべての変更をコミットできるようになる。バージョンの増分により、旧バージョンのセッションがロックを獲得できないので、一貫性のないレコードデータを防止することができる。

RDBMS データストアの場合、妥当性確認は、バージョン番号をレコードの更新または削除に使われる SQL 文の基準に追加することである。1つの SQL 文で、ロックの獲得と、レコードデータの更新の両方を行うことができる。最終ステップは、ビジネストランザクショ

ンがSQLの実行によって返される行カウントを検査することである。行カウントの1は正常終了を示し、0はレコードがすでに変更または削除されていることを示す。行カウントが0の場合、ビジネストランザクションはシステムトランザクションをロールバックし、レコードデータに何らかの変更を行うことを防止する。この時点で、ビジネストランザクションは、コンフリクトの解決と再試行の中止または試行のいずれかを行う。

各レコードのバージョン番号の他に、並行性コンフリクトを管理する場合、だれが最後にレコードを修正したか、いつ修正するのが有効かについての情報を格納する。ユーザに並行性違反による更新失敗を通知する場合、適切なアプリケーションを使用することによって、いつ、だれがレコードを修正したかを伝える。軽チェックに、バージョンカウントではなく修正したタイムスタンプを使用することは勧められない。その理由は、システムクロックはまったく信頼性がないためである。特に、複数のサーバ全体を調整する場合には信頼性がなくなる。

代替実装の場合、更新のwhere句には行のすべてのフィールドが含まれる。メリットは、何らかの形式のVersion（バージョン）フィールドを使うことなくwhere句を使えることである。これは、データベーステーブルを修正してVersionフィールドを追加できない場合に役立つ。問題点は、これによってwhere句が長大になり、UPDATE文を複雑化してしまうことである。さらにデータベースが適切にプライマリキーインデックスを使えるかどうかによっては、パフォーマンスに影響を与える可能性もある。

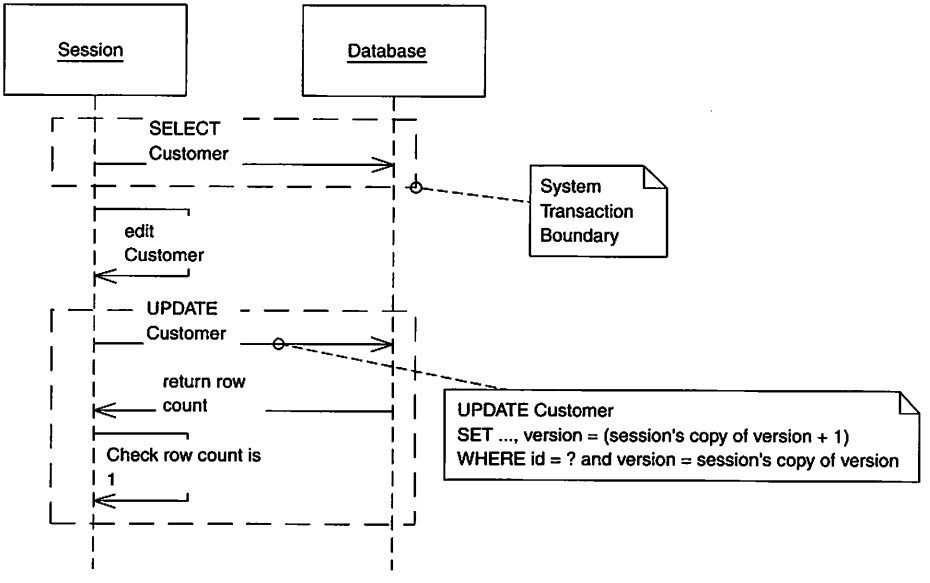


図 16.1 —— UPDATE文の軽チェック

軽オフラインロックを実装すると、UPDATE 文と DELETE 文にバージョンを含んでいないので、一貫性のない読み込みの問題を解決することができなくなる。料金設定と適切な売上税の計算を行う請求システムを考えてみる。1 つのセッションが料金を設定し、それに加算する税金を計算するため Customer の Address を調べる。しかし料金生成セッション中に、別の Customer 保守セッションが、Customer の Address を編集する場合がある。税率は所在地に左右されるので、料金生成セッションによって計算された税率は無効になるはずである。しかし、料金生成セッションは、コンフリクトが検出されない場合には Address を一切変更しない。

軽オフラインロックを、一貫性のない読み込みの検出に使えないという理由はない。上記の例の場合、料金生成セッションは正確性が Customer の Address の値に左右されてしまうことを認識しなければならない。したがって、料金生成セッションは Address に対してもバージョンチェックを行う必要がある。これは、Address を変更セットに追加すること、またはバージョン確認された項目の個別のリストを保持することによって実行できる場合がある。後者の場合、設定に若干作業負荷が増えるが、より明確なコードによって一貫性のある読み込みを実現できる。人為的更新ではなく、単にバージョンの再読み込みで一貫性のある読み込みをチェックする場合、特にシステムトランザクションの分離レベルを認識してほしい。バージョンの再読み込みは、繰り返しできる読み込みまたはより明確な分離を行う場合にだけ有効である。分離レベルが低い場合は、必ずバージョンの増分が必要となる。

一貫性のない読み込みの任意の問題に対しては、バージョンチェックはやり過ぎである。トランザクションはレコード、またはフィールドの 1 つの値にだけ依存していることがある。その場合、同時更新がビジネストランザクションのコンフリクトを発生させることがほとんどないため、バージョンではなく状態をチェックすることでシステムの稼働率を向上できることがある。並行性問題の理解が進むにつれ、さらにその問題をコードで管理することができるようになる。

緩ロックは、オブジェクトのグループを 1 つのロック可能な項目として扱うことによって、一貫性のない読み込みの特定の難問に対処するのに役立つ。別の選択肢は、単に長時間継続するトランザクション内に問題になりそうなビジネストランザクションのステップを実行することである。実装が簡単であるため、散在する数個のロングトランザクションを使うときには、リソースのヒットに有効な場合がある。

トランザクションが特定のレコードの読み込みではなく、動的なクエリーの結果に依存している場合、一貫性のない読み込みの検出は少し難しくなる。**軽オフラインロック**の取得方法として、最初の結果を保存し、コミット時の同じクエリーの結果と比較することができる。

ロックメカニズムと同様に、ビジネスアプリケーションの複雑な並行性や一時的な問題によっては、**軽オフラインロック**それ自体では適切な解決策を提供することができないことがある。ビジネスアプリケーションの並行性管理が、技術的問題であるのと同様にドメインの

問題でもあることを強調しておく。上記のCustomer Addressのシナリオは、本当にコンフリクトだろうか。Customerの旧バージョンで売上税の計算することでも問題ないかもしれないが、どのバージョンを使うべきだろうか。これはビジネス上の問題である。あるいは、コレクションを考えてほしい。2つのセッションが同時に項目を1つのコレクションに追加すると、どうなるだろうか。一般的な**軽オフラインロック**スキームでは、確実にビジネスルールに違反している場合でもこれを防止できない。

習得すべき**軽オフラインロック**を使った1つのシステムが、ソースコード管理（SCM）である。SCMシステムは、プログラマ間にコンフリクトを発見した場合、適切なマージで解決し、コミットを再試行することができる。上位のマージメカニズムによって、システムの並行性がとても優れたものになるからだけでなく、ユーザが何らかの作業を再度行う必要がほとんどなくなるため、**軽オフラインロック**がとても強力なものとなる。もちろん、SCMシステムとエンタープライズビジネスアプリケーションの大きな相違点は、SCMが1種類のマージを実装するのに対し、ビジネスシステムは何百ものマージを実装する点である。中にはとても複雑なため、コーディングのコストをかけるだけの価値がないものもある。ビジネスにおいてとても価値があるため、マージを何とかしてコーディングすべきものもある。ほとんど行われることはないが、ビジネスオブジェクトのマージも可能である。ビジネスデータのマージは、それ自体でパターンである。テーマを台無しにしないためそれには触れないでおく。しかし、マージで**軽オフラインロック**を追加する機能を理解してほしい。

ビジネストランザクションがコミットする場合、**軽オフラインロック**だけが最後のシステムトランザクション中に通知してくる。しかし、コンフリクトが発生するとき、早期に通知されることが役立つ場合がある。この場合、他のだれかがデータを更新しているかどうかをチェックするcheckCurrentメソッドを使うことができる。これによってコンフリクトにならないとは保証できないが、事前にコミットしないと通知できる場合、複雑なプロセスを中止できることが有効な場合がある。失敗の早期チェックが役立つことがある場合には常に、checkCurrentを使ってほしい。しかしコミット時に失敗しないという保証には決してならないことを忘れないでほしい。

16.1.2 | 使用するタイミング

軽い並行性管理は、いずれか2つのビジネストランザクション間のコンフリクトの可能性が低い場合に適切である。コンフリクトが起こりそうな場合、ユーザが作業を終了しコミットの用意ができているときだけ通知するようでは、ユーザフレンドリーとは言えない。結局、ユーザはビジネストランザクションが失敗したと思い、システムを中止する。コンフリクトの可能性が高い場合や、コンフリクトによる犠牲を受け入れられない場合、**重オフラインロック**の方が適している。

ただし、軽ロックは実装しやすく、**重オフラインロック**で起こるような欠点や実行時エラーが発生しにくいため、構築するシステムにおけるビジネストランザクションコンフリクト管理のデフォルトの手法として使うことを考えてほしい。重いバージョンは、軽いバージョンを補完することもある。コンフリクトを防止するために、重い手法をいつ使うべきかではなく、軽い手法だけでは不十分なのはどんな時かと自問していただきたい。

並行性管理の適切な手法は、データに対する同時アクセスを最大限にする一方で、コンフリクトを最小限にするものである。

16.1.3 | 例：データマッパーによるドメインレイヤ（Java）

軽オフラインロックの最も短い例には、Version 列を含むデータベーステーブルと、更新基準の一部としてバージョンの UPDATE 文と DELETE 文だけが必要なはずである。もちろん、読者はより洗練されたアプリケーションを構築するだろう。そこで**ドメインモデル**と**データマッパー**を使う実装を示す。これで**軽オフラインロック**を実装するときに生じる問題がさらに明確になる。

行うべき最初の作業の 1 つは、ドメインの**レイヤスーパータイプ**が、**軽オフラインロック**の実装に必要なあらゆる情報、つまり修正とバージョンのデータを格納可能かどうかを確認することである。

```
class DomainObject...

   private Timestamp modified;
   private String modifiedBy;
   private int version;
```

データはリレーショナルデータベースに格納されているので、テーブルにはバージョンと修正のデータも格納されていなければならない。customer テーブルのほか、**軽オフラインロック**のサポートに必要とされる標準的な CRUD SQL のスキーマは、以下のとおりである。

```
table customer...

   create table customer(id bigint primary key, name varchar,
      createdby varchar, created datetime, modifiedby varchar,
      modified datetime, version int)

SQL customer CRUD...
   INSERT INTO customer VALUES (?, ?, ?, ?, ?, ?, ?)
```

```
SELECT * FROM customer WHERE id = ?
UPDATE customer SET name = ?, modifiedBy = ?, modified = ?,
    version = ? WHERE id = ? AND version = ?
DELETE FROM customer WHERE id = ? AND version = ?
```

いったん数個以上のテーブルやドメインオブジェクトを持つようになると、O/R マッピングの単調で繰り返しの多いセグメントを処理するデータマッパーのために、**レイヤスーパータイプ**を導入したくなる。これは、**データマッパー**を記述する際の多くの作業を省くだけではない。**暗黙ロック**を使うことによって、開発者がロック手順の一部のコーディングを忘れ、ロックの組み込みを忘れてしまうことを防止することができる。

抽象マッパーに入力する最初の部分は、SQL の構築である。マッパーにテーブルに関するわずかなメタデータを提供する必要がある。マッパーが実行時に SQL を構築するようにするための代替方法は、SQL のコードを生成することである。しかし、これは読者の自習に任せ、SQL 文の構築には手をつけないことにする。以下の抽象マッパーでは、修正データの列名と位置についていくつかの仮説を立てた。これによってレガシーデータを使う場合の実行可能性が低下する。抽象マッパーは、それぞれの具象マッパーから供給される若干の列メタデータを必要とする可能性がある。

抽象マッパーは SQL 文を持つと CRUD 動作を管理することができる。find メソッドを実行する方法は、以下のとおりである。

```
class AbstractMapper...

    public AbstractMapper(String table, String[] columns) {
        this.table = table;
        this.columns = columns;
        buildStatements();
    }
    public DomainObject find(Long id) {
        DomainObject obj = AppSessionManager.getSession().getIdentityMap().get(id);
        if (obj == null) {
            Connection conn = null;
            PreparedStatement stmt = null;
            ResultSet rs = null;
            try {
                conn = ConnectionManager.INSTANCE.getConnection();
                stmt = conn.prepareStatement(loadSQL);
                stmt.setLong(1, id.longValue());
                rs = stmt.executeQuery();
                if (rs.next()) {
                    obj = load(id, rs);
```

```
                String modifiedBy = rs.getString(columns.length + 2);
                Timestamp modified = rs.getTimestamp(columns.length + 3);
                int version = rs.getInt(columns.length + 4);
                AppSessionManager.getSession().getIdentityMap().put(obj);
            } else {
                throw new SystemException(table + " " + id + " does not exist");
            }
        } catch (SQLException sqlEx) {
            throw new SystemException("unexpected error finding " + table + " " + id);
        } finally {
            cleanupDBResources(rs, conn, stmt);
        }
    }
    return obj;
}
protected abstract DomainObject load(Long id, ResultSet rs) throws SQLException;
```

少し注釈を加えよう。最初にマッパーは、**一意マッピング**をチェックし、オブジェクトがロード済みではないことを確かめる。**一意マッピング**を使わないと、ビジネストランザクション中で異なる時間にロードされた、異なるバージョンのオブジェクトとなることがある。これはアプリケーションに定義されていない振る舞いを行う結果となり、またすべてのバージョンチェックが混乱することにもなる。結果群が取得されると、マッパーは抽象 load メソッドに従い、具象マッパーは抽象 load メソッドを実装して、フィールドの抽出とアクティブなオブジェクトを返す。マッパーは setSystemFields() を呼び出し、抽象ドメインオブジェクトにバージョンと修正データを設定する。このデータを渡す方法としてはコンストラクタの方が適切に思えるかもしれないが、コンストラクタを使うと、バージョンを格納する責任の一部をそれぞれの具象マッパーやドメインオブジェクトが担うことになるため、**暗黙ロック**が弱体化することになる。

具象 load() メソッドは、以下のとおりである。

```
class CustomerMapper extends AbstractMapper...

    protected DomainObject load(Long id, ResultSet rs) throws SQLException {
        String name = rs.getString(2);
        return Customer.activate(id, name, addresses);
    }
```

抽象マッパーも同様に、更新および削除動作の実行を管理する。このジョブは、データベースの動作が 1 の行カウントを返すかどうかをチェックすることである。行がまったく更

新されない場合、軽ロックを取得できず、マッパーは並行性例外を起こす必要がある。削除動作は、以下のとおりである。

```
class class AbstractMapper...

    public void delete(DomainObject object) {
        AppSessionManager.getSession().getIdentityMap().remove(object.getId());
        Connection conn = null;
        PreparedStatement stmt = null;
        try {
            conn = ConnectionManager.INSTANCE.getConnection();
            stmt = conn.prepareStatement(deleteSQL);
            stmt.setLong(1, object.getId().longValue());
            int rowCount = stmt.executeUpdate();
            if (rowCount == 0) {
                throwConcurrencyException(object);
            }
        } catch (SQLException e) {
            throw new SystemException("unexpected error deleting");
        } finally { cleanupDBResources(conn, stmt);
        }
    }
    protected void throwConcurrencyException(DomainObject object) throws SQLException {
        Connection conn = null;
        PreparedStatement stmt = null;
        ResultSet rs = null;
        try {
            conn = ConnectionManager.INSTANCE.getConnection();
            stmt = conn.prepareStatement(checkVersionSQL);
            stmt.setInt(1, (int) object.getId().longValue());
            rs = stmt.executeQuery();
            if (rs.next()) {
                int version = rs.getInt(1);
                String modifiedBy = rs.getString(2);
                Timestamp modified = rs.getTimestamp(3);
                if (version > object.getVersion()) {
                    String when = DateFormat.getDateTimeInstance().format(modified);
                    throw new ConcurrencyException(table + " " + object.getId() +
                            " modified by " + modifiedBy + " at " + when);
                } else {
                    throw new SystemException("unexpected error checking timestamp");
                }
            } else {
```

```
                throw new ConcurrencyException(table + "
                        " + object.getId() + " has been deleted");
            }
        } finally {
            cleanupDBResources(rs, conn, stmt);
        }
    }
```

並行性例外のバージョンチェックに使われるSQLも、抽象マッパーが知る必要がある。マッパーはCRUD SQLを構築するときには、SQLも構築する必要がある。それは、以下のとおりである。

```
checkVersionSQL...

    SELECT version, modifiedBy, modified FROM customer WHERE id = ?
```

このコードを見ても、さまざまな部分が1つのビジネストランザクション内で複数のシステムトランザクションにまたがって実行されているとは思わないかもしれない。重要なことは、**軽オフラインロック**の獲得はレコードデータの一貫性を維持するため、変更のコミットを保持している同じシステムトランザクション内で実行されなければならない点である。UPDATE文とDELETE文にバンドルされるチェックの場合、このことは問題にはならない。

緩ロックの例のコードでVersionオブジェクトを使う場合を見てみよう。**緩ロック**を使うと一貫性のない読み込みのいくつかの問題を解決することができる。しかし、シンプルな非共有Versionオブジェクトは、increment()またはcheckVersionIsLatest()などの軽チェックの振る舞いを追加する場所でもあるため、一貫性のない読み込みを検出するのに役立てられる。追加する**ユニットオブワーク**は以下のとおりであり、分離レベルがわからないため、Versionの差分を根本的に測定することよって、一貫性のある読み込みチェックをコミットプロセスに追加する。

```
class UnitOfWork...

    private List reads = new ArrayList();
    public void registerRead(DomainObject object) {
        reads.add(object);
    }
    public void commit() {
        try {
```

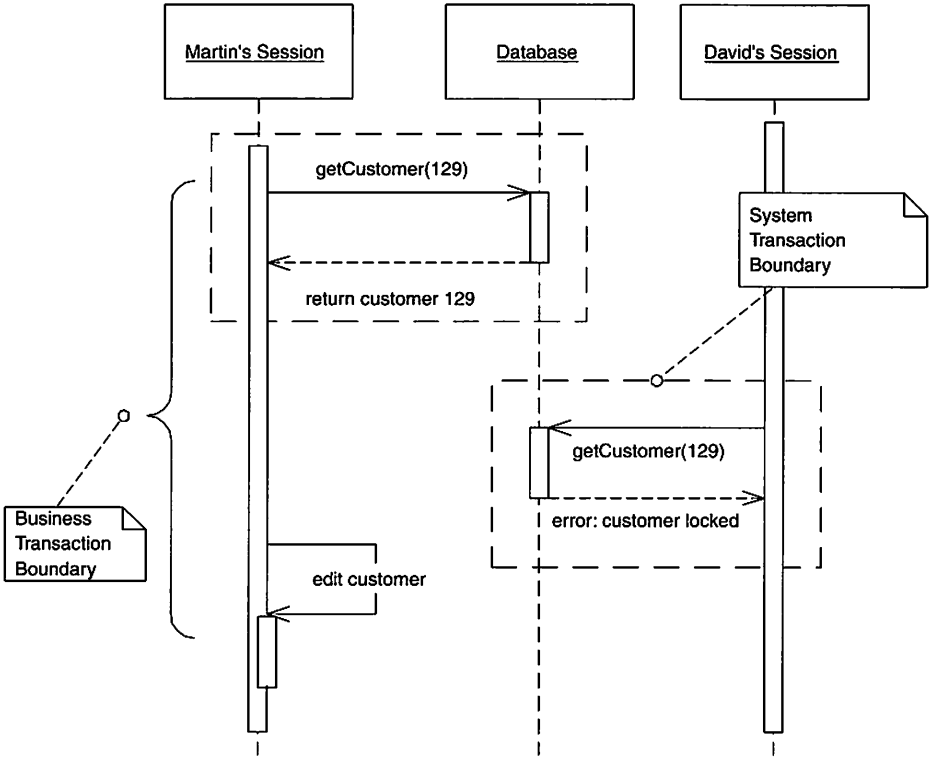
```
            checkConsistentReads();
            insertNew();
            deleteRemoved();
            updateDirty();
        } catch (ConcurrencyException e) {
            rollbackSystemTransaction();
            throw e;
        }
    }
    public void checkConsistentReads() {
        for (Iterator iterator = reads.iterator(); iterator.hasNext();) {
            DomainObject dependent = (DomainObject) iterator.next();
            dependent.getVersion().increment();
        }
    }
```

ユニットオブワークは並行性違反を検出すると、システムトランザクションをロールバックする。ほとんどの場合、コミット中に何らかの例外が発生した場合には、ロールバックを行うことになる。このステップを忘れないでほしい。Version オブジェクトの代替として、バージョンチェックをマッパーインタフェースに追加することもできる。

16.2 重オフラインロック

(by David Rice)

データへのアクセスを一度に1つのビジネストランザクションに限定することで、同時ビジネストランザクション間のコンフリクトを防止する。



オフライン並行性では、複数のリクエストにまたがってビジネストランザクションのデータを処理する場合がある。その場合の最もシンプルな手法は、システムトランザクションを、ビジネストランザクション全体に対してオープンにすることである。しかし残念なことに、トランザクションシステムはロングトランザクションで機能するようになっていないため、この手法が必ずしも有効とは限らない。このため複数のシステムトランザクションを使う必要があり、データへの同時アクセスの管理をしているデバイスに任せる。

最初に取り組むべき手法は**軽オフラインロック**である。ただし、パターンには問題がある。数人がビジネストランザクション内の同じデータにアクセスすると、そのうちの1人は容易にコミットされるが、その他の人はコンフリクトし失敗することになる。コンフリクトはビジネストランザクションの最後でだけ検出されるため、その犠牲者はトランザクションすべ

てを実行し終わってはじめて、すべてが失敗に終わり、費やした時間が無駄になったことに気付く。長時間継続するビジネストランザクションでこのようなコンフリクトが頻繁に発生するなら、システムの評判はたちまち失墜してしまうだろう。

重オフラインロックを使うことによって、上記の状態をすべて回避し、コンフリクトを完全に防止できる。**重オフラインロック**は使用開始前に、ビジネストランザクションがデータの一部分にロックを獲得することを強制する。多くの場合、ビジネストランザクションを開始しても並行性制御によって拒否されることなく、トランザクションを確実に完了できる。

16.2.1 | 動作方法

重オフラインロックは次の3つのフェーズで実装することができる。つまり、必要なロックの種類の決定、ロックマネージャーの構築、およびビジネストランザクションがロックを使う手続の定義である。さらに、**軽オフラインロック**の補完として**重オフラインロック**を使う場合、ロックするレコードの種類を決定する必要がある。

ロックの種類における最初の選択肢は**書き込み専用ロック**である。これは、ビジネストランザクションにセッションデータを編集するロックだけを獲得することを求める。また、2つのビジネストランザクションが同じレコードを同時に変更することを許可しないため、コンフリクトを防止する。データの読み込みをこのロックメカニズムは無視するので、ビューセッションが最新データを保持することが必須ではない場合には、この考え方で十分である。

ビジネストランザクションが常に最新データを保持していなければならない場合には、編集するかどうかに関わらず**読み込み専用ロック**を使ってほしい。これは、ビジネストランザクションに単にレコードをロードするロックを獲得することを求める。この考え方では明らかに、システムの並行性が厳しく制限される可能性がある。エンタープライズシステムでは、書き込み専用ロックの方が、読み込み専用ロックより多くのレコードに同時アクセスできる。

3つ目の考え方は2種類のロックを組み合わせたものであり、読み込み専用ロックの制限的なロックに加え、書き込み専用ロックの向上した並行性も提供する。**読み書きロック**と呼ばれ、最初の2つよりも少し複雑になっている。読み込みロックと書き込みロックの関係は以下のとおりであり、両方の機能のメリットを活かすことが重要である。

- 読み込みロックと書き込みロックは相互に排他的である。他のいずれかのビジネストランザクションがレコードに読み込みロックを持っている場合、そのレコードに書き込みロックを実行することは不可能である。また、他のいずれかのビジネストランザクションが書き込みロックを持っている場合、レコードに読み込みロックを実行することも不可能である。
- 同時の読み込みロックは許可されている。1つの読み込みロックが存在すると、

ビジネストランザクションがレコードを編集できなくなるため、1つのセッションでの読み込みが許可されるとき、いくつものセッションでの読み込みが可能になることには問題はない。

複数の読み込みロックが許可されると、システムの並行性が向上する。この仕組みの弱点は実装がとても難しいことであり、システムをモデリングする際にドメインの専門家は頭を抱えながら、弱点を克服するためにより一層奮闘するのである。

適切なロックの種類を選ぶ場合、システムの並行性の最大化、ビジネスニーズへの適合性、およびコードの複雑性の最小化を考えてほしい。さらにドメインモデラーとアナリストは、ロックメカニズムを理解しなければならないことにも留意してほしい。ロックは単なる技術的な問題ではない。その理由は、不適切なロックの種類、つまり単にすべてのレコードをロックしたり、不適切な種類のレコードをロックしたりするだけなら、効果のない**重オフラインロック**メカニズムになる可能性があるからである。効果のない**重オフラインロック**メカニズムでは、ビジネストランザクションの開始時点でコンフリクトを防止できず、シングルユーザシステムのように見えるほどに、マルチユーザシステムの並行性が劣化してしまう。不適切なロックメカニズムは、適切な技術的実装で埋め合わせることはできない。ドメインモデルに**重オフラインロック**を含めることは、悪い考えではない。

ロックの種類を決定したら、ロックマネージャーを定義してほしい。ロックマネージャーのジョブは、ビジネストランザクションがロックを獲得または解除するリクエストに対し、許可または拒否を行うことである。ジョブを実行するために、ロックマネージャーはビジネストランザクションにおいてロックされているもの以外に、ロックを実行した所有者も認識している必要がある。ビジネストランザクションの概念が、一意に識別できるようなものでない可能性は十分ある。その場合、ビジネストランザクションをロックマネージャーに渡すことが少し困難になる。この場合、自身の判断でセッションオブジェクトを持つ可能性を多くするようにセッションの概念を考えること。「セッション」と「ビジネストランザクション」という用語は、まったく区別せずに使うことができるものである。ビジネストランザクションが、1つのセッション内で直列的に実行される限り、セッションは**重オフラインロック**の所有者として適切である。コードの例で、この考え方をいくらか明確にさせなければいけないだろう。

ロックマネージャーを構成するのは、ロックを所有者にマッピングするテーブルである。シンプルなロックマネージャーがメモリ内のハッシュテーブルをラップする場合、あるいはロックマネージャーがデータベーステーブルである可能性がある。どのような場合であっても、1つ、つまりロックテーブルを1つだけ持つ必要がある。それがメモリ内にある場合、必ずシングルトン[Gang of Four]を使ってほしい。アプリケーションサーバがクラスタ化している場合、メモリ内のロックテーブルは、1つのサーバインスタンスに対応していない限

り機能しない。クラスタ化されたアプリケーションサーバ環境である場合、データベースをベースとするロックマネージャーの方が適切である可能性が高い。

ロックは、データベーステーブルにオブジェクトまたはSQLとして実装されているかどうかに関わらず、ロックマネージャー専用である必要がある。ビジネストランザクションは、ロックマネージャーとだけ相互作用を行う必要があり、ロックオブジェクトとは決して相互作用を行わない。

次に、ビジネストランザクションがロックマネージャーを使うために従うべきプロトコルを定義する。プロトコルには、何をいつロックするか、いつロックを解除するか、そしてロックが獲得できなかったときはどのように動作するかを含める必要がある。

何をロックするかは、いつロックするかに左右される。そのため、いつロックするかを最初に考慮する。一般的に、ビジネストランザクションはデータをロードする前にロックを獲得する必要がある。ロックされた項目の最新バージョンを持てるという保証がない場合は、ロックを獲得するメリットは少ない。しかし、システムトランザクション内でロックを獲得しているため、ロックとロードの順序を問題にしない環境もある。ロックの種類によっては、直列化可能または繰り返し可能な読み込みトランザクションを使っている場合、オブジェクトのロードとロックの獲得を行う順序は問題にはならないはずである。**重オフラインロック**を獲得した後に、軽チェックを項目に対して実行するという選択肢もある。オブジェクトをロックしてしまえば、オブジェクトが最新バージョンであることはまちがいない。このことは、オブジェクトはデータをロードする前にロックを獲得する、と言うのと同じなのである。

次に、何をロックするか。オブジェクトでもレコードでも、つまりどのようなものでもロックされているようだが、通常それらのオブジェクトを検索するために使うIDまたはプライマリキーをロックしている。これによって、オブジェクトをロードする前にロックを取得できる。ロックを獲得した後もオブジェクトが現時点と同じものであるというルールに違反しない限りは、オブジェクトのロックは有効である。

ロックを解除する最もシンプルなルールは、ビジネストランザクションが完了したときに解除することである。ロックの種類とオブジェクトを再度トランザクション内で使う場合、完了前のロック解除も許可されるはずである。さらに、とても扱いにくいシステム稼働率の問題など、解除を早めるべき極めて特殊な問題がない限りは、ビジネストランザクションの完了時に解除することを厳守してほしい。

ロックを獲得できないビジネストランザクションに対する最も簡単な対処は、中止することである。**重オフラインロック**では、トランザクションのやや早い時点で失敗するはずなので、ユーザは受け入れることができるはずである。このような状況を改善するには、開発者と設計者が、失敗する可能性の特に高いロックをトランザクションの後半まで待たずに獲得することである。可能な限りユーザが作業を開始する前に、すべてのロックを獲得することを勧める。

ロックしたい特定の項目がある場合、ロックテーブルへは直列化されたアクセスを行う。メモリ内のロックテーブルの場合、最も簡単な方法は、使っているプログラミング言語が提供する構造体を用いて、ロックマネージャー全体へのアクセスを直列化することである。この方法で提供されるよりも高度な並行性を必要としている場合、複雑な領域に足を踏み入れることを覚悟してほしい。

ロックテーブルがデータベースに格納されている場合、もちろん、最初のルールは1つのシステムトランザクション内でそのテーブルと相互作用することである。データベースが提供する直列化機能のメリットを十分活用しよう。読み込み専用ロックと書き込み専用ロックを使う場合に直列化を行うには、ロック可能な項目のIDを格納する列に一意性制約を適用するだけでよい。しかし、データベースに読み込み／書き込みロックを格納する場合、ロジックがロックテーブルの挿入だけではなく読み込みも必要とするため、一貫性のない読み込みの防止が必須になり、事態は少し難しくなる。直列化可能な分離レベルのシステムトランザクションは、一貫性のない読み込みを行わないことを保証するため、最高の安全性を提供できる。システム全体で直列化可能トランザクションを使用すると、パフォーマンス障害が発生する場合があるが、ロックを獲得するために個別の直列化可能システムトランザクションを用意し、他の動作に対しては分離レベルを低くすることで、この問題を緩和できる場合がある。別の選択肢は、ストアドプロシージャがロック管理に役立つかどうかを調べることである。並行性の管理は難しくなる場合があるので、重要な時には躊躇せずにデータベースに管理を任せるようにする。

直列的なロック管理によって、パフォーマンスのボトルネックが増大する。ロックが少ないと、発生するボトルネックも少なくなるため、ロックの細分度を十分検討する。**緩ロック**によって、ロックテーブルの競合に対処できる。

「SELECT FOR UPDATE...」またはエンティティEJBなどの、システムトランザクションの重ロックメカニズムを使う場合、メカニズムはロックが使えるまで待機するため、明らかにデッドロックの問題が発生する可能性がある。デッドロックを以下のように考えてほしい。2人のユーザがリソースAとBを必要としているとする。一方がAのロックを取得し、もう一方がBのロックを取得した場合、両方のトランザクションは停止し、他のロックをいつまでも待機し続ける。1つのビジネストランザクションに20分もの時間が必要となることもあるので、範囲が複数のシステムトランザクションに及ぶ場合は、ロックを待機してもあまり意味がない。だれもそんなロックを待ちたくはないだろう。また、待機のコーディングにはタイムアウトが必要になり、すぐにコードが複雑になるので、待機しないほうがよい。ロックが使えなくなった場合はすぐに、ロックマネージャーで例外を起こす。これによって、デッドロックに対処する負荷を削減できる。

最後の要件は、失われたセッションのロックタイムアウトの管理である。トランザクションの途中でクライアントマシンに障害が発生した場合、失われたトランザクションが所有し

ていたロックすべての完了および解除が不可能になる。定期的にユーザから破棄されるセッションを含むWebアプリケーションの場合、これは大問題である。アプリケーションでタイムアウトを処理するよりも、アプリケーションサーバが管理するタイムアウトメカニズムを使う方が望ましい。Webアプリケーションサーバは、このためにHTTPセッションを提供している。タイムアウトは、HTTPセッションが無効になったときにロックを解除するユーティリティオブジェクトを登録して実装できる。別の選択肢は、タイムスタンプとロックを関連付け、一定時間を経過したロックは無効と見なすようにすることである。

16.2.2 ｜使用するタイミング

重オフラインロックは、同時セッション間のコンフリクトの可能性が高い場合に有効である。ユーザは、決して作業をあきらめる必要はない。さらに、ロックはコンフリクト発生の可能性に関係なく、コンフリクトのコストがあまりにも高い場合にも適切である。システムのすべてのエンティティをロックすると、ほぼ確実に膨大なデータの競合の問題が発生する。そのため、**重オフラインロック**は**軽オフラインロック**を補完するものにすぎず、本当に必要な部分にだけ使用すべきものであるということを忘れないでほしい。

重オフラインロックを使う必要がある場合、ロングトランザクションのことも考える必要がある。ロングトランザクションは決して推奨できるものではないが、状況によっては、**重オフラインロック**ほど損害を与えることもなく、はるかにプログラムしやすい場合がある。いくつか負荷テストを行ってみてから、この方法を選択するとよい。

ビジネストランザクションが1つのシステムトランザクション内に収まっている場合は、これらの技法は使わないでほしい。多くのシステムトランザクションの重ロック技法が、すでに使っているアプリケーションやデータベースサーバに同梱されていて、それらの中には、データベースロック用の「SELECT FOR UPDATE」SQL文とアプリケーションサーバロック用のエンティティEJBが含まれている。必要がないのに、どうしてタイムアウトやロックの可視性などを気にかけることがあるだろうか。これらのロックの種類について理解していれば、確かに**重オフラインロック**の実装時に多くの付加価値を加えることができる。しかし、その逆は真実ではない。ここでの解説は、データベースマネージャーやトランザクションモニタを記述するためのものではない。本書で紹介しているオフラインロック技法のすべては、システムが独自のトランザクションモニタを持っていることを前提としている。

16.2.3 ｜例：シンプルなロックマネージャー（Java）

例では、最初に読み込み専用ロックのためのロックマネージャーを構築する（ロックはオブジェクトの読み込みや編集に必要なものである）。次に、ビジネストランザクションが複

数のシステムトランザクションにまたがる場合のロックマネージャーの使い方を解説する。

最初のステップは、ロックマネージャーインタフェースを定義することである。

```
interface ExclusiveReadLockManager...

    public static final ExclusiveReadLockManager INSTANCE =
            (ExclusiveReadLockManager)
    Plugins.getPlugin(ExclusiveReadLockManager.class);
    public void acquireLock(Long lockable, String owner) throws ConcurrencyException;
    public void releaseLock(Long lockable, String owner);
    public void relaseAllLocks(String owner);
```

lockable を長整数型、owner を文字列型で定義していることに注意してほしい。Lockable は長整数型である。理由は、データベースのテーブルがシステム全体で一意である長いプライマリキーを使っていて、正確なロック可能な ID として機能するためである（ロック可能な ID はロックテーブルで処理されるすべての種類のテーブルで一意である必要がある）。owner ID は文字列型である。理由は、例は Web アプリケーションであり内部で HTTP セッション ID が適切なロックの所有者を作成するからである。

ロックオブジェクトではなく、データベースのロックテーブルと直接相互作用を行うロックマネージャーを記述する。これはデータベース内部のロックメカニズムではなく、他のアプリケーションテーブルと同様の lock と呼ばれる独自のテーブルであることに留意してほしい。ロックを獲得するということは、行をロックテーブルに正しく挿入するということである。ロックを解除するということは、行を削除するということである。ロックテーブルとロックマネージャーの一部の実装スキーマは、以下のとおりである。

```
table lock...

    create table lock(lockableid bigint primary key, ownerid bigint)

class ExclusiveReadLockManagerDBImpl implements ExclusiveLockManager...

    private static final String INSERT_SQL =
            "INSERT INTO lock VALUES(?, ?)";
    private static final String DELETE_SINGLE_SQL =
            "DELETE FROM lock WHERE lockableid = ? AND ownerid = ?";
    private static final String DELETE_ALL_SQL =
            "DELETE FROM lock WHERE ownerid = ?";
    private static final String CHECK_SQL =
            "SELECT lockableid FROM lock WHERE lockableid = ? AND ownerid = ?";
    public void acquireLock(Long lockable, String owner) throws ConcurrencyException {
```

```
        if (!hasLock(lockable, owner)) {
            Connection conn = null;
            PreparedStatement pstmt = null;
            try {
                conn = ConnectionManager.INSTANCE.getConnection();
                pstmt = conn.prepareStatement(INSERT_SQL);
                pstmt.setLong(1, lockable.longValue());
                pstmt.setString(2, owner);
                pstmt.executeUpdate();
            } catch (SQLException sqlEx) {
                throw new ConcurrencyException("unable to lock " + lockable);
            } finally {
                closeDBResources(conn, pstmt);
            }
        }
    }
    public void releaseLock(Long lockable, String owner) {
        Connection conn = null;
        PreparedStatement pstmt = null;
        try {
            conn = ConnectionManager.INSTANCE.getConnection();
            pstmt = conn.prepareStatement(DELETE_SINGLE_SQL);
            pstmt.setLong(1, lockable.longValue());
            pstmt.setString(2, owner);
            pstmt.executeUpdate();
        } catch (SQLException sqlEx) {
            throw new SystemException("unexpected error releasing lock on
                    " + lockable);
        } finally {
            closeDBResources(conn, pstmt);
        }
    }
```

ロックマネージャーに示されていないのは、パブリックな`releaseAllLocks()`メソッドとプライベートな`hasLock()`メソッドである。`releaseAllLocks()`は、名前のとおりにOwnerのロックを解除する。`hasLock()`は、Ownerがすでにロックを所有しているかどうかをデータベースに確認する。セッションコードがすでに所有しているロックを獲得しようとすることは珍しくない。そのため、`acquireLock()`はロック行を挿入しようとする前に、まず、Ownerがすでにロックを所有していないかどうかをチェックする必要がある。ロックテーブルはリソースが競合するポイントであるため、読み込みを繰り返すと、アプリケーションパフォーマンスが低下する場合がある。所有者チェックのため、セッションレベルで所有しているロックをキャッシュすることが必要な場合がある。これを慎重に行ってほしい。

次にCustomerレコードを管理するため、シンプルなWebアプリケーションを構築しよう。最初に、ビジネストランザクションの処理に役立つインフラの一部を設定する。ユーザセッションの概念によっては、Webティアの下のレイヤで必要とされる場合がある。そのため、HTTPセッションだけに依存することはできない。新しいセッションを、HTTPセッションとは別のアプリケーションセッションと見なすことにする。アプリケーションセッションには、ID、ユーザ名、および**一意マッピング**が格納され、ビジネストランザクションの間にロードまたは作成されたオブジェクトをキャッシュする。それらのオブジェクトは、見つけやすくするために現在実行中のスレッドと関連付けられている。

```
class AppSession...

   private String user;
   private String id;
   private IdentityMap imap;
   public AppSession(String user, String id, IdentityMap imap) {
      this.user = user;
      this.imap = imap;
      this.id = id;
   }

class AppSessionManager...

   private static ThreadLocal current = new ThreadLocal();
   public static AppSession getSession() {
      return (AppSession) current.get();
   }
   public static void setSession(AppSession session) {
      current.set(session);
   }
```

フロントコントローラを使ってリクエストを処理するためには、コマンドの定義が必要になる。それぞれのコマンドが最初に行うべきことは、新しいビジネストランザクションを開始するか、または既存のビジネストランザクションを継続するかのいずれであるかを明確にすることである。この決定によって、新しいアプリケーションセッションを設定するか、または現行のアプリケーションセッションを発見するかが決まる。ビジネストランザクションのコンテキストを設定するために使いやすいメソッドを提供する抽象コマンドは、以下のとおりである。

```
interface Command...

   public void init(HttpServletRequest req, HttpServletResponse rsp);
```

```
    public void process() throws Exception;

abstract class BusinessTransactionCommand implements Command...

    public void init(HttpServletRequest req, HttpServletResponse rsp) {
        this.req = req;
        this.rsp = rsp;
    }
    protected void startNewBusinessTransaction() {
        HttpSession httpSession = getReq().getSession(true);
        AppSession appSession = (AppSession) httpSession.getAttribute(APP_SESSION);
        if (appSession != null) {
            ExclusiveReadLockManager.INSTANCE.relaseAllLocks(appSession.getId());
        }
        appSession = new AppSession(getReq().getRemoteUser(),
                httpSession.getId(), new IdentityMap());
        AppSessionManager.setSession(appSession);
        httpSession.setAttribute(APP_SESSION, appSession);
        httpSession.setAttribute(LOCK_REMOVER,
                new LockRemover(appSession.getId()));
    }
    protected void continueBusinessTransaction() {
        HttpSession httpSession = getReq().getSession();
        AppSession appSession = (AppSession) httpSession.getAttribute(APP_SESSION);
        AppSessionManager.setSession(appSession);
    }
    protected HttpServletRequest getReq() {
        return req;
    }
    protected HttpServletResponse getRsp() {
        return rsp;
    }
```

新しいアプリケーションセッションを設定する場合、既存のアプリケーションセッションが持つロックを削除する必要がある。さらに、リスナーをHTTPセッションバインドイベントに追加する。そして、イベントは、対応するHTTPセッションが期限切れになったとき、アプリケーションセッションが持つあらゆるロックを削除する。

```
class LockRemover implements HttpSessionBindingListener...

   private String sessionId;
   public LockRemover(String sessionId) {
      this.sessionId = sessionId;
```

```
    }
    public void valueUnbound(HttpSessionBindingEvent event) {
        try {
            beginSystemTransaction();
            ExclusiveReadLockManager.INSTANCE.relaseAllLocks(this.sessionId);
            commitSystemTransaction();
        } catch (Exception e) {
            handleSeriousError(e);
        }
    }
```

コマンドには標準ビジネスロジックとロック管理の両方が含まれ、1つのシステムトランザクションの範囲内で実行する必要がある。確実に行うため、トランザクションのコマンドオブジェクトで装飾する[Gang of Four]ことができる。1つのリクエストに対するロックおよび標準ドメインビジネスが、1つのシステムトランザクション内で発生することを確認してほしい。システムトランザクションの境界を定義するメソッドは、配置コンテキストに依存している。並行性例外、およびこのケースでの他の例外のいずれかが検出される場合、システムトランザクションをロールバックすることが必須である。これでコンフリクトが発生しても、何らかの変更が永続レコードデータに挿入されるのを防止できる。

```
class TransactionalComamnd implements Command...

    public TransactionalCommand(Command impl) {
        this.impl = impl;
    }
    public void process() throws Exception {
        beginSystemTransaction();
        try {
            impl.process();
            commitSystemTransaction();
        } catch (Exception e) {
            rollbackSystemTransaction();
            throw e;
        }
    }
```

次は、コントローラサーブレットと具象コマンドの書き込みについてである。コントローラサーブレットには、コマンドをトランザクションコントローラでラップする責任がある。具象コマンドは、ビジネストランザクションのコンテキストの設定、ドメインロジックの実行、適切なロックの獲得と解除を行うために必要である。

```
class ControllerServlet extends HttpServlet...

    protected void doGet(HttpServletRequest req, HttpServletResponse rsp)
                    throws ServletException, IOException {
        try {
            String cmdName = req.getParameter("command");
            Command cmd = getCommand(cmdName);
            cmd.init(req, rsp);
            cmd.process();
        } catch (Exception e) {
            writeException(e, rsp.getWriter());
        }
    }
    private Command getCommand(String name) {
        try {
            String className = (String) commands.get(name);
            Command cmd = (Command) Class.forName(className).newInstance();
            return new TransactionalCommand(cmd);
        } catch (Exception e) {
            e.printStackTrace();
            throw new SystemException("unable to create command object for " + name);
        }
    }

class EditCustomerCommand implements Command...

    public void process() throws Exception {
        startNewBusinessTransaction();
        Long customerId = new Long(getReq().getParameter("customer_id"));
        ExclusiveReadLockManager.INSTANCE.acquireLock(
                customerId, AppSessionManager.getSession().getId());
        Mapper customerMapper = MapperRegistry.INSTANCE.getMapper(Customer.class);
            Customer customer = (Customer) customerMapper.find(customerId);
        getReq().getSession().setAttribute("customer", customer);
        forward("/editCustomer.jsp");
    }

class SaveCustomerCommand implements Command...

    public void process() throws Exception {
        continueBusinessTransaction();
        Customer customer = (Customer) getReq().getSession().getAttribute("customer");
        String name = getReq().getParameter("customerName");
        customer.setName(name);
        Mapper customerMapper = MapperRegistry.INSTANCE.getMapper(Customer.class);
```

```
        customerMapper.update(customer);
        ExclusiveReadLockManager.INSTANCE.releaseLock(customer.getId(),
                                AppSessionManager.getSession().getId());
        forward("/customerSaved.jsp");
    }
```

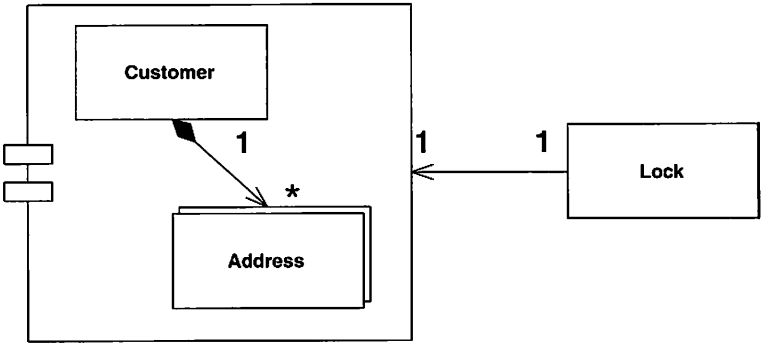
上記で表示しているコマンドは、任意の2つのセッションが同時にCustomerの処理を行うことを防止する。アプリケーションのCustomerオブジェクトを処理する他のコマンドも、確実にロックを獲得するか、または同じビジネストランザクションの以前のコマンドによってロックされたCustomerだけを処理するかのいずれかである必要がある。ロックマネージャーにhasLock()チェックがあれば、コマンドでロックを容易に獲得することもできる。このことはパフォーマンスにとっては不利であるが、ロックを設定する場合には確かな保証となるはずである。**暗黙ロック**では、ロックメカニズムに関する他の極めて簡単な手法を解説する。

フレームワークコードの量は、ドメインコードの量に比べ若干バランスが取れていないと感じられるかもしれない。**重オフラインロック**には、最小限の組み合わせでも、アプリケーションセッション、ビジネストランザクション、ロックマネージャー、およびシステムトランザクションが必要であり、これは疑いもなく難問である。この例は、多くの部分で強固さが不足しているため、アーキテクチャテンプレートとしてよりも、アイデアの源泉として役立つだろう。

16.3 緩ロック

(by David Rice, Matt Foemmel)

1つのロックで関連オブジェクトセットをロックする。



オブジェクトはグループとして編集できる場合が多い。おそらく、CustomerとAddressのセットを使っているはずである。アプリケーションを使う際にこれらの中のいずれか1つをロックしたい場合、項目すべてをロックすることは有効である。個々のオブジェクトを個別にロックすることには、いくつもの課題がある。まず、誰であれそのようなオブジェクトを処理する人は、ロックするためにオブジェクトを発見するコードを記述する必要がある。CustomerとそのAddressについては容易だが、さらにロックグループを取得するためのコードは複雑で難しいものになる。グループが複雑になったときはどう対処すればいいのだろうか。フレームワークがロックの獲得を管理する場合、振る舞いはどこに配置すればいいのだろうか。**軽オフラインロック**を使う場合のように、ロックするためにオブジェクトをロードする必要があるロックメカニズムの場合、大規模なグループをロックするとパフォーマンスに影響を及ぼす。また、**重オフラインロック**では、大規模なロックセットは管理が厄介となりロックテーブルの競合を増大させる。

緩ロックは、多くのオブジェクトを対象とする1つのロックである。これによって、ロックアクション自体が簡素化されるだけではなく、ロックするためにグループのメンバすべてをロードする必要もなくなる。

16.3.1 | 動作方法

緩ロックを実装する最初のステップは、オブジェクトのグループをロックするための、単一の競合ポイントを作成することである。これで、セット全体をロックするのに必要なロックは1つだけになる。これによって、ロック取得プロセスにおいて、メモリへロードされ識別されなければならないグループメンバを最小化するためのシングルロックポイントを見つける最短パスを供給できるようになる。

軽オフラインロックの場合、1つのグループの各項目でバージョンを共有すると（図16.2参照）、競合のシングルポイントが作成される。これは、**同じ**バージョンを共有することであって、**等しい**バージョンを共有することではない。このバージョンを増分すると、グループ全体が**共有ロック**でロックされる。グループのすべてのメンバが共有バージョンを指すようにモデルを設定する必要がある。これによって、確実に競合のポイントへのパスが最短化される。

共有**重オフラインロック**では、グループのメンバがいくつかの種類のロック可能なトークンを共有し、このロックを獲得する必要がある。**重オフラインロック**は、**軽オフラインロック**を補完するものとして使われるため、共有バージョンオブジェクトはロック可能なトークンの役割にふさわしい優れた対象になる（図16.3）。

Eric EvansとDavid Siegel [Evans]は、データ変更のユニットとして扱われる関連オブジェクトのクラスタとして**集合体**を定義している。それぞれの集合体には、セットのメンバ

への唯一のアクセスポイントを提供するルートと、セットに含まれているものを定義する境界がある。集合体のメンバのいずれかを処理する場合には、メンバすべてをロックする必要があるため、集合体の特性上緩ロックが求められる。集合体のロックは、ルートロックと呼ばれる共有ロックの代替方法となる（図16.4参照）。定義では、ルートをロックすると集合体のすべてのメンバがロックされる。ルートロックにより、競合のシングルポイントが提供される。

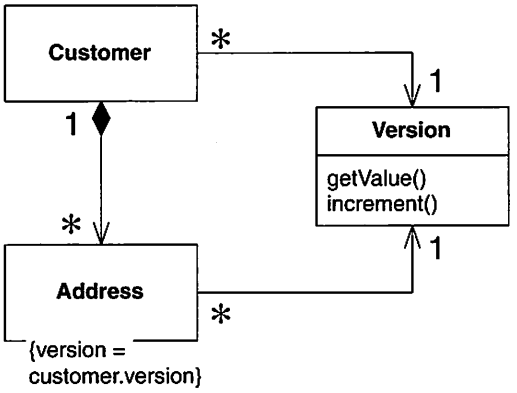


図16.2 ——バージョンの共有

緩ロックとしてルートロックを使うと、オブジェクトグラフにルートへの遷移を実装する必要がある。これで集合体のすべてのオブジェクトにロックが要求される場合、ロックメカニズムはルートに遷移してそのままルートをロックすることができる。遷移は2つの方法で実現することができる。集合体のそれぞれのオブジェクトはルートへの直接遷移の管理、または一連の中間関係を使うことができる。たとえば、階層構造では、ルートは明白にトップレベルの親であり下層に直接リンクできる。あるいは、ノードをその直上の親とリンクさせ、遷移を行いルートに到達することもできる。大きなグラフの場合、後者の考え方ではそれぞれの親が自分には親があるかどうかを特定するためにロードする必要があるので、パフォーマンスの問題が発生する可能性がある。ルートへのパスを形成するオブジェクトをロードする場合、必ずレイジーロードを使う。これによって、オブジェクトが必要とされる前にロードされるのを防止できるだけでなく、双方向の関係をマッピングする場合には、無限のマッピングループも防止できる。1つの集合体のレイジーロードが、複数のシステムトランザクション全体にまたがる可能性があることに十分注意してほしい。これで一貫性のない部分で構築された集合体になる可能性がある。もちろんこれは推奨されることではない。

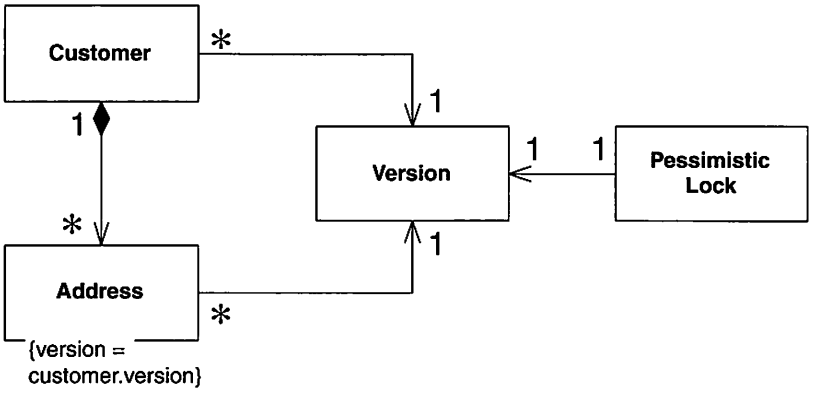


図 16.3 ——共有バージョンのロック

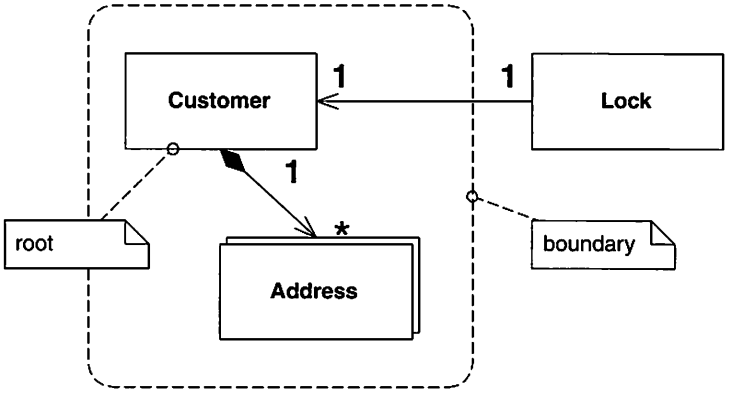


図 16.4 ——ルートのロック

集合体のオブジェクトがロックされると同時にルートもロックされるため、共有ロックも集合体のロックとして有効であることに留意してほしい。

緩ロックの共有ロックとルートロックの両方の実装には、それぞれトレードオフがある。リレーショナルデータベースを使う場合、共有ロックでは、選択したものをバージョンテーブルに関連付けしなければならないという困難な作業が必要になる。しかし、オブジェクトのロードとルートへの遷移を同時に行うと、さらにパフォーマンスへの打撃となる可能性もある。おそらくルートロックと**重オフラインロック**は奇妙な組み合わせになるだろう。ルートへの遷移が行われロックされるときまでに、数個のオブジェクトを再ロードして、オブジェクトを最新に保つ必要がある場合がある。レガシーデータに対してシステムを構築すると、実装を選択するときに多くの制約が生じる。ロック実装の選択肢は多数あり、これに類似した選択肢はもっと多くなる。必ずニーズに合った実装を実行しなければならない。

16.3.2 | 使用するタイミング

緩ロックを使う明確な理由は、ビジネス要件を満たすことである。その実例が、集合体をロックする場合である。Asset（資産）コレクションを所有するLease（賃貸借契約）オブジェクトを考えてみる。おそらく、1人のユーザがLeaseを編集し、もう1人のユーザが同時にAssetを編集するということは、ビジネスではありえないだろう。LeaseまたはAssetのいずれかをロックすると、LeaseとAssetがロックされるはずである。

緩ロックを使うメリットは、ロックの獲得と解除のコストを節約できることである。これは確かに**緩ロック**を使うための合理的な動機である。共有ロックは、集合体[Evans]の概念の枠を越えて使えるが、パフォーマンスなどの非機能的な要件に基づいて動作させる場合には慎重さが求められる。**緩ロック**を十分に活用するためには、不自然なオブジェクト関係を作成しないように注意してほしい。

16.3.3 | 例：共有軽オフラインロック（Java）

例では、**レイヤスーパータイプ**、永続ストアとしてのリレーショナルデータベース、および**データマッパー**を備えるドメインモデルを紹介する。最初の手順は、Versionクラスとテーブルの作成である。簡素化するためにはやや適用範囲の広いバージョンを作成し、値を格納するだけでなく静的なfindメソッドも持つようにする。

留意してほしいのは、一意マッピングを使ってセッションのバージョンをキャッシュすることである。オブジェクトがバージョンを共有する場合、オブジェクトがバージョンの同じインスタンスを正確に指すことは重要なことである。Versionクラスはドメインモデルの一部であるため、データベースコードを配置する形式にはなっていない可能性がある。そのため、ここでは個別のバージョンのデータベースコードをマッパーレイヤに置いたままにしておき、それ以上のことは読者の学習に委ねたい。

```
table version...

    create table version(id bigint primary key, value bigint,
        modifiedBy varchar, modified datetime)

class Version...

    private Long id;
    private long value;
    private String modifiedBy;
    private Timestamp modified;
    private boolean locked;
    private boolean isNew;
```

```
private static final String UPDATE_SQL =
        "UPDATE version SET value = ?, modifiedBy = ?, modified = ? " +
        "WHERE id = ? AND value = ?";
private static final String DELETE_SQL =
        "DELETE FROM version WHERE id = ? AND value = ?";
private static final String INSERT_SQL =
        "INSERT INTO version VALUES (?, ?, ?, ?)";
private static final String LOAD_SQL =
        "SELECT id, value, modifiedBy, modified FROM version WHERE id = ?";
public static Version find(Long id) {
    Version version = AppSessionManager.getSession().
            getIdentityMap().getVersion(id);
    if (version == null) {
        version = load(id);
    }
    return version;
}
private static Version load(Long id) {
    ResultSet rs = null;
    Connection conn = null;
    PreparedStatement pstmt = null;
    Version version = null;
    try {
        conn = ConnectionManager.INSTANCE.getConnection();
        pstmt = conn.prepareStatement(LOAD_SQL);
        pstmt.setLong(1, id.longValue());
        rs = pstmt.executeQuery();
        if (rs.next()) {
            long value = rs.getLong(2);
            String modifiedBy = rs.getString(3);
            Timestamp modified = rs.getTimestamp(4);
            version = new Version(id, value, modifiedBy, modified);
            AppSessionManager.getSession().getIdentityMap().putVersion(version);
        } else {
            throw new ConcurrencyException("version " + id + " not found.");
        }
    } catch (SQLException sqlEx) {
        throw new SystemException("unexpected sql error loading version", sqlEx);
    } finally {
        cleanupDBResources(rs, conn, pstmt);
    }
    return version;
}
```

バージョンは、バージョン自身の作成方法も知っている。データベースの挿入と作成を分離させることで、1つのOwnerをデータベースに挿入するまでは、挿入を禁止する。ドメインのデータマッパーは、対応するドメインオブジェクトが挿入されている場合、バージョンでの挿入を安全に呼び出すことができる。バージョンは、ドメインオブジェクトが一度だけ挿入されることを確かめるために、新規かどうかを追跡する。

```
class Version...

    public static Version create() {
        Version version = new Version(IdGenerator.INSTANCE.nextId(), 0,
                AppSessionManager.getSession().getUser(), now());
        version.isNew = true;
        return version;
    }
    public void insert() {
        if (isNew()) {
            Connection conn = null;
            PreparedStatement pstmt = null;
            try {
                conn = ConnectionManager.INSTANCE.getConnection();
                pstmt = conn.prepareStatement(INSERT_SQL);
                pstmt.setLong(1, this.getId().longValue());
                pstmt.setLong(2, this.getValue());
                pstmt.setString(3, this.getModifiedBy());
                pstmt.setTimestamp(4, this.getModified());
                pstmt.executeUpdate();
                AppSessionManager.getSession().getIdentityMap().putVersion(this);
                isNew = false;
            } catch (SQLException sqlEx) {
                throw new SystemException("unexpected sql error inserting
                        version", sqlEx);
            } finally {
                cleanupDBResources(conn, pstmt);
            }
        }
    }
```

次は、対応するデータベース行のバージョンの値を増分させるincrement()メソッドである。変更セットの複数のオブジェクトは同じバージョンを共有する可能性が高いため、バージョンは増分される前にまずロックされていないことを確認する。データベースを呼び出した後、increment()メソッドは、バージョン行が本当に更新されたかどうかをチェックする

必要がある。行カウント0が返される場合、並行性違反を検出し、例外を投げる。

```
class Version...

    public void increment() throws ConcurrencyException {
        if (!isLocked()) {
            Connection conn = null;
            PreparedStatement pstmt = null;
            try {
                conn = ConnectionManager.INSTANCE.getConnection();
                pstmt = conn.prepareStatement(UPDATE_SQL);
                pstmt.setLong(1, value + 1);
                pstmt.setString(2, getModifiedBy());
                pstmt.setTimestamp(3, getModified());
                pstmt.setLong(4, id.longValue());
                pstmt.setLong(5, value);
                int rowCount = pstmt.executeUpdate();
                if (rowCount == 0) {
                    throwConcurrencyException();
                }
                value++;
                locked = true;
            } catch (SQLException sqlEx) {
                throw new SystemException("unexpected sql error
                        incrementing version", sqlEx);
            } finally {
                cleanupDBResources(conn, pstmt);
            }
        }
    }
    private void throwConcurrencyException() {
        Version currentVersion = load(this.getId());
        throw new ConcurrencyException(
                "version modified by " + currentVersion.modifiedBy + " at " +
                DateFormat.getDateTimeInstance().format
                (currentVersion.getModified()));
    }
```

このコードを使って、コミットしたビジネストランザクションのシステムトランザクション内だけでincrementを呼び出していることを確認してほしい。コミットトランザクションの間で、トランザクションの初期での増分が、誤ったロックを獲得となるために

isLocked フラグを使う。軽ロックでは、コミットするときにだけロックするので問題にはならない。

このパターンを使うときには、以前のシステムトランザクションのデータベースと比較して、データが現在のものかどうかを確認したい場合がある。checkCurrent メソッドを Version クラスに追加することでそれを確認できる。メソッドを使うことで、**軽オフラインロック**が更新しなくても使えるかどうかを容易にチェックできる。

データベースから Version を削除するための SQL を実行する delete メソッドは、ここでは示していない。返される行カウントが 0 の場合、並行性例外が発生する。このバージョンで使う最後のオブジェクトが削除されるとき、**軽オフラインロック**が取得されていなかったためであると思われる。これは決して起こってはならないことである。実際の動作は、いつ共有バージョンの削除が許可されるかを認識している。集合体全体でバージョンを共有している場合、集合体のルートを削除した後、単にバージョンを削除するだけである。しかし、他のシナリオの場合はかなり難しい。1 つは、Version オブジェクトがその Owner の参照カウントを保持し、そのカウントが 0 になったときに自らを削除することである。このために、やや洗練された Version オブジェクトになる可能性があることを警告しておきたい。バージョンが複雑になると、Version オブジェクトを本格的なドメインオブジェクトにすることを考える場合がある。もちろんこれは十分に有効なことだが、Version のない特殊なドメインオブジェクトとなる。

次は、共有バージョンの使い方について解説する。ドメイン**レイヤスーパータイプ**には、シンプルなカウントではなく Version オブジェクトが含まれている。**データマッパー**によって、ドメインオブジェクトのロード時に Version を設定する。

```
class DomainObject...

    private Long id;;
    private Timestamp modified;
    private String modifiedBy;
    private Version version;
    public void setSystemFields(Version version, Timestamp modified,
              String modifiedBy) {
        this.version = version;
        this.modified = modified;
        this.modifiedBy = modifiedBy;
    }
```

作成に関して、Customer のルートと Address から構成される集合体を見てみよう。Customer の create メソッドで、共有バージョンが作成される。Customer は、Address

を作成する addAddress() メソッドを持ち、Customer のバージョンを渡す。抽象データベースマッパーは、バージョンを挿入してから対応するドメインオブジェクトを挿入する。バージョンによって一度だけ挿入されることを忘れないようにする。

```
class Customer extends DomainObject...

   public static Customer create(String name) {
      return new Customer(IdGenerator.INSTANCE.nextId(),
            Version.create(), name, new ArrayList());
   }

class Customer extends DomainObject...

   public Address addAddress(String line1, String city, String state) {
      Address address = Address.create(this, getVersion(), line1, city, state);
      addresses.add(address);
      return address;
   }

class Address extends DomainObject...

   public static Address create(Customer customer, Version version,
      String line1, String city, String state) {
      return new Address(IdGenerator.INSTANCE.nextId(), version,
                  customer, line1, city, state);
   }

class AbstractMapper...

   public void insert(DomainObject object) {
      object.getVersion().insert();
```

増分は、**データマッパー**がオブジェクトの更新または削除を行う前に、**データマッパー**から Version に要求される必要がある。

```
class AbstractMapper...

   public void update(DomainObject object) {
      object.getVersion().increment();

class AbstractMapper...
```

```
    public void delete(DomainObject object) {
        object.getVersion().increment();
```

これは集合体なので、Customer を削除するときに Address も削除する。その直後に Version を削除できる。

```
class CustomerMapper extends AbstractMapper...

    public void delete(DomainObject object) {
        Customer cust = (Customer) object;
        for (Iterator iterator = cust.getAddresses().iterator();
                iterator.hasNext();) {
            Address add = (Address) iterator.next();
            MapperRegistry.getMapper(Address.class).delete(add);
        }
        super.delete(object);
        cust.getVersion().delete();
    }
```

16.3.4 ｜例：共有重オフラインロック（Java）

関連セットのオブジェクトの関連付けができるいくつかの種類のロック可能なトークンが必要になる。前述のように、**重オフラインロック**を**軽オフラインロック**を補完するものとして使うので、ロック可能なトークンとして共有バージョンを使う。

共有バージョンに達するために、同じコードを使う。唯一の問題は、バージョンを取得するため一部のデータをロードする必要があることである。データをロードしてから**重オフラインロック**を獲得する場合、データが現在のものであることをどうやって知ることができるだろうか。簡単な方法としては、**重オフラインロック**を取得したシステムトランザクション内のバージョンを増分することである。システムトランザクションがコミットされると、重ロックが有効になり、システムトランザクション内のどこでロードしたとしても、そのバージョンを共有するデータの最新のコピーを持っていることがわかる。

```
class LoadCustomerCommand...

        try {
            Customer customer = (Customer)
                    MapperRegistry.getMapper(Customer.class).find(id);
            ExclusiveReadLockManager.INSTANCE.acquireLock
                    (customer.getId(), AppSessionManager.getSession().getId());
```

```
            customer.getVersion().increment();
            TransactionManager.INSTANCE.commit();
        } catch (Exception e) {
            TransactionManager.INSTANCE.rollback();
            throw e;
        }
```

バージョンの増分は、ロックマネージャーに構築したいものであることがわかる。少なくとも、バージョンを増分するコードでロックマネージャーを装飾したい[Gang of Four]と思うはずである。もちろん、本稼動するコードには、示した例より強固な例外処理やトランザクション制御が必要になる。

16.3.5 | 例：ルート軽オフラインロック（Java）

この例は、これまでの**レイヤスーパータイプ**や**データマッパー**などの例と同じことを前提としている。Version オブジェクトもあるが、ここでは省くことにする。ここでは、単に**データマッパー**の外側で**軽オフラインロック**を獲得しやすくするための使いやすいincrement()メソッドを提供する。また、変更セットを追跡する**ユニットオブワーク**も使っている。

集合体には親－子関係が構築されているので、ルートを見つけるためには、子－親遷移を使う。ドメインモデルとデータモデルもこれに対応させる。

```
class DomainObject...

    private Long id;
    private DomainObject parent;
    public DomainObject(Long id, DomainObject parent) {
        this.id = id;
        this.parent = parent;
    }
```

Owner を持つと、**ユニットオブワーク**をコミットする前にルートロックを取得できる。

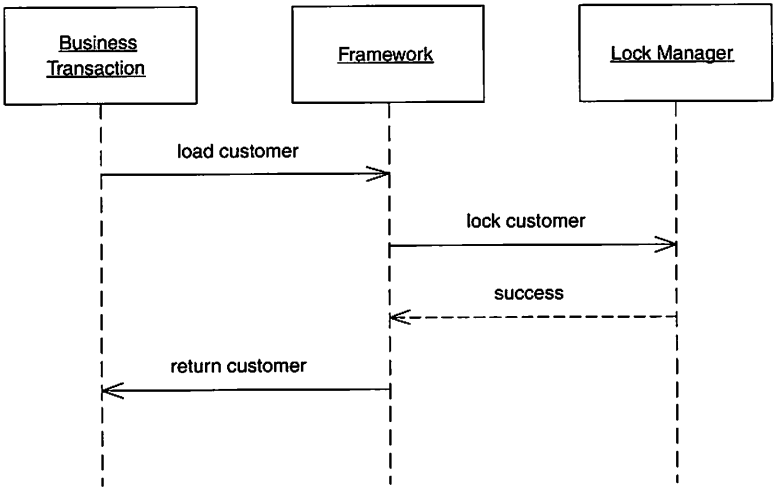
```
class UnitOfWork...

    public void commit() throws SQLException {
        for (Iterator iterator = _modifiedObjects.iterator(); iterator.hasNext();) {
            DomainObject object = (DomainObject) iterator.next();
            for (DomainObject owner = object; owner != null; owner =
```

```
                owner.getParent()) {
            owner.getVersion().increment();
        }
    }
    for (Iterator iterator = _modifiedObjects.iterator(); iterator.hasNext();) {
        DomainObject object = (DomainObject) iterator.next();
        Mapper mapper = MapperRegistry.getMapper(object.getClass());
        mapper.update(object);
    }
}
```

16.4 | 暗黙ロック

(by David Rice)

フレームワークまたはレイヤスーパータイプコードがオフラインロックを獲得できる。



ロックメカニズムで重要な点は、すべての場所で使うことである。ロックを獲得するコードの記述を1行忘れたために、オフラインロックメカニズム全体が使い物にならない場合もある。他のトランザクションが読み込みロックを使っているところで書き込みロックの検索をしないと、更新セッションデータを取得できないこともあり、またバージョンカウントを使わないと、他人が変更を上書きしても気付かないこともある。一般的には、ある項目が**任意の場所で**ロックされた場合、**すべての場所でも**ロックされている必要がある。アプリケーションのロックメカニズムを無視すると、ビジネストランザクションは一貫性のないデータとなってしまう。ロックを解除しないと、レコードデータは破損することはないが、作業が

できなくなってしまう。オフライン並行性管理のテストは困難なので、どのようなテストスイートでもエラーを検出できない場合がある。

その1つの解決策は、開発者に誤りを許さないことである。一切の見落としが許されないロックタスクは、開発者により明示的に処理されるのではなく、アプリケーションにより暗黙的に処理される必要がある。ほとんどのエンタープライズアプリケーションでは、フレームワークの組み合わせである**レイヤスーパータイプ**を使っているので、コード生成が**暗黙ロック**を活用する十分な機会を提供する。

16.4.1 | 動作方法

暗黙ロックを実装するとは、**絶対に省略できない**ロックメカニズムをアプリケーションフレームワークで実行できるようにコードを構成することである。適切な表現が思い浮かばないので「フレームワーク」という用語を使うが、これは**レイヤスーパータイプ**、Framework クラス、および他の任意の「プラミング」コードの組み合わせをいうこととする。コード生成ツールもまた、適切なロックを実現するための手段である。これは決して驚くようなことではない。これを使う可能性があるのは、アプリケーション全体で数回同じロックメカニズムのコーディングを行う場合である。正しくコーディングされないことが多いので、注意して見る必要がある。

最初のステップは、ロックメカニズム内で処理するビジネストランザクションに必要なタスクのリストをまとめることである。**軽オフラインロック**の場合、リストには、更新 SQL 基準のバージョンを含む各レコードのバージョンカウントの格納、およびレコード変更時に増分されたバージョンなどの項目が含まれている。**重オフラインロック**リストは、1つのデータをロードするために必要なロック（通常、読み込み専用ロック、または読み込み／書き込みロックの読み込みロック部分）を獲得するビジネストランザクション、またはセッションの完了時にすべてのロックを解除する明細項目を含んでいる。

重オフラインロックのリストに含まれないものとして、1つのデータを編集するために必要なロック（書き込み専用ロックや読み込み／書き込みロックの書き込みロック部分）の獲得があることに留意する。ビジネストランザクションがデータの編集を行う予定の場合、ロックは必須である。しかしロックが使えない場合、暗黙的にそれらを獲得すると2つの困難な問題が発生する。第1に、**ユニットオブワーク**内の不確定なオブジェクトを登録するような、書き込みロックを暗黙的に獲得する際の唯一のポイントは、ロックが使えない場合があることだ。この場合、ユーザの作業開始直後にトランザクションが中止されるという保証はない。アプリケーションは、これらのロックを獲得する時点を発見できないのだ。早い時期に失敗しないトランザクションは、**重オフラインロック**の「ユーザに同じ作業を2度させてはならない」という目的とコンフリクトしてしまう。

第2に、同じくらい重要な点だが、ロックの種類は極限までシステムの並行性を制限してしまうことである。**暗黙ロック**を使わないということは、その問題を技術的な領域から切り離して、ビジネスドメインの領域で考え、並行性にどのような影響が及ぼされるかについて検討するのに役立つ。それでも、書き込みに必要なロックは、変更がコミットされる前に獲得する必要がある。フレームワークにできることは、何らかの変更がコミットされる前に書き込みロックが取得されるようにすることである。コミット時までにロックが獲得されない場合、それはプログラマの間違いであり、コードは少なくともアサーションの失敗を発生させる。しかし私は、アサーションを省略して並行性例外を起こすことを勧める。理由は、アサーションが機能を停止するときに発生する、本稼動システム上のいかなるエラーも望ましくないからである。

暗黙ロックを使うことについての注意を一言述べたい。**暗黙ロック**によって、開発者はロックメカニズムの大部分を無視することできるが、結果を無視することはできない。たとえば、開発者がロックを待機する重ロックメカニズムで**暗黙ロック**を使う場合、デッドロックについても考える必要がある。**暗黙ロック**の軽視とは、いったん開発者がロックについて考えていないと、ビジネストランザクション全体で予想外の形で失敗する可能性があるということである。

ロックを機能させることは、フレームワークを取得する最善の方法を決定することであり、暗黙的にロックシステムを実行する。そのロックの種類の暗黙的な処理の例については、**軽オフラインロック**を参照してほしい。上位レベルの**暗黙ロック**の実装は、とても多くの種類があるため、本書ですべてを紹介することはできない。

16.4.2 | 使用するタイミング

暗黙ロックを使う必要があるのは、フレームワークの概念をもたない最もシンプルなアプリケーション以外の場合である。ロックを1つ忘れた場合、そのリスクの大きさは計りしれない。

16.4.3 | 例：暗黙重オフラインロック（Java）

読み込み専用ロックのシステムを考えてみよう。このアーキテクチャには**ドメインモデル**が含まれ、ドメインオブジェクトとリレーショナルデータベース間を仲介するものとして**データマッパー**を使っている。フレームワークは、読み込み専用ロックでドメインオブジェクトのロックを獲得する。その後ビジネストランザクションは、ドメインオブジェクトでいかなる処理でも行うことができる。

ビジネストランザクションで使われる任意のドメインオブジェクトは、マッパーの

find()メソッドで配置される。これは、ビジネストランザクションがfind()を呼び出すことで直接的に、またオブジェクトグラフを遷移して間接的に、マッパーを使ういずれの場合にも適用できる。次に、マッパーにロック機能を装飾する[Gang of Four]ことができる。ロックを獲得するロックマッパーを記述してから、オブジェクトの検索に取りかかろう。

```
interface Mapper...

    public DomainObject find(Long id);
    public void insert(DomainObject obj);
    public void update(DomainObject obj);
    public void delete(DomainObject obj);

class LockingMapper implements Mapper...

    private Mapper impl;
    public LockingMapper(Mapper impl) {
        this.impl = impl;
    }
    public DomainObject find(Long id) {
        ExclusiveReadLockManager.INSTANCE.acquireLock(
            id, AppSessionManager.getSession().getId());
        return impl.find(id);
    }
    public void insert(DomainObject obj) {
        impl.insert(obj);
    }
    public void update(DomainObject obj) {
        impl.update(obj);
    }
    public void delete(DomainObject obj) {
        impl.delete(obj);
    }
```

セッションのオブジェクトを1回以上検索することはよくあるので、上記コードを動作させるために、ロックマネージャーがセッションをロックする前に、セッションがすでにロックを持っていないかをまずチェックする必要がある。読み込み専用ロックではなく書き込み専用ロックを使う場合は、ロックを獲得するときではなく更新や削除を行うときに、以前のロック獲得をチェックするマッパー装飾子を記述する。

装飾子のメリットの1つは、ラップされているオブジェクトが機能を拡張していることがわからないことである。以下のようにレジストリでマッパーをラップできる。

```
LockingMapperRegistry implements MappingRegistry...

   private Map mappers = new HashMap();
   public void registerMapper(Class cls, Mapper mapper) {
      mappers.put(cls, new LockingMapper(mapper));
   }
   public Mapper getMapper(Class cls) {
      return (Mapper) mappers.get(cls);
   }
```

ビジネストランザクションが、マッパーにアクセスして標準的な更新メソッドを呼び出すときに実際に起こっていることを図16.5に示す。

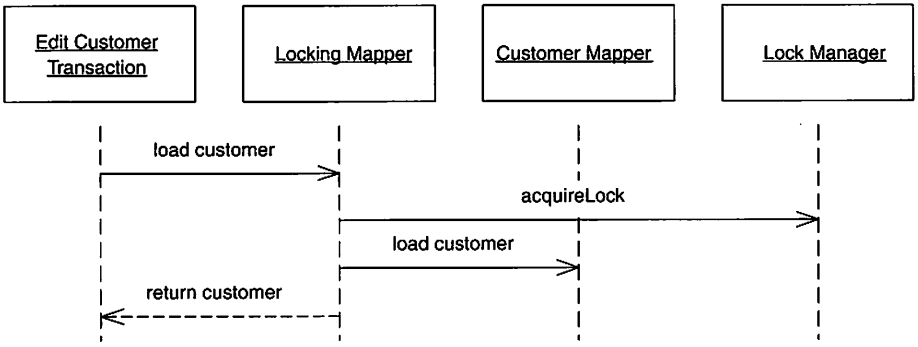


図16.5——ロックマッパー

第17章

セッションステートパターン

17.1 | クライアントセッションステート

セッションステートをクライアントに格納する。

17.1.1 | 動作方法

サーバ指向の設計でセッション識別子を保持するだけでも、少なくとも**クライアントセッションステート**が必要になる。アプリケーションによっては、クライアントのすべてのセッションデータを配置することもある。そのような場合クライアントは、各リクエストでセッションデータのセットを送り、サーバは各レスポンスでセッションステートを返信する。これでサーバは完全にステートレスになる。

データの転送処理には、**データ変換オブジェクト**を使いたいと思う。回線上で自らを直列化することができるので、複雑なデータでさえ送信することができるからだ。

さらに、クライアントはデータの格納を求める。リッチクライアントアプリケーションの場合、インタフェース内のフィールドのように、構造の中でこれを行えるが、私としてはそんな作業を行うならバドワイザーでも飲んでいたいものだ。**データ変換オブジェクト**やドメインモデルなどの非ビジュアルオブジェクトを使う方がよい方法だ。いずれの場合も大きな問題はない。

HTML インタフェースを使う場合、やや複雑になる。クライアントセッションステートを実行するのによく使われているのは、URL パラメータ、隠しフィールド、クッキーという 3 つの方法である。

URL パラメータは、少量のデータに対しても容易に機能する。基本的にいずれのレスポンスページであっても、URL がパラメータとしてセッションステートを取得する。この機能の限界は、URL のサイズが限定されるためであるが、データ項目が 2 つだけの場合は有効である。そのためセッション ID などによく選択される。プラットフォームによっては、

自動的にURLを書き換えてセッションIDを追加する。URLの変更はブックマークで問題になる場合があるので、一般的なWebサイトにはURLパラメータを使うことに反対する意見がある。

隠しフィールドは、ブラウザーに送ってもWebページに表示されないフィールドである。<INPUT type = "hidden"> 形式のタグを使って取得する。隠しフィールドを機能させるには、リクエストに対してレスポンスおよび再読み込みを行う場合、フィールドへのセッションステートを直列化し、データを隠し、フィールドに配置するフォーマットが必要になる。標準的にはXMLが選択されるが、やや冗長であるといえる。何らかのテキストベースのエンコーディングスキーマで、データをエンコードする方法もあるが、隠しフィールドは表示されるページから隠されているだけで、ページソースを閲覧すれば誰でもデータが見られることを覚えていてほしい。

以前のWebページや固定されたWebページを含む混合サイトには注意が必要である。セッションデータへの遷移を行うと、すべてのセッションデータを失う可能性がある。

最後に、賛否両論をもたらす選択はクッキーである。クッキーでは返信と送信が自動的に行われ、隠しフィールドとまったく同様に、その中に入れるセッションステートを直列化して使う。クッキーのサイズには限界があり、多くの人はクッキーを好まず、使わない設定にしている。クッキーを使わない設定の場合は、サイトは機能しなかったが、現在多くのサイトでクッキーに依存する傾向が強くなってきているため、サイトが機能しないということは減少している。また、純粋な企業内システムの場合、問題にならないことは確かである。

クッキーでは確実な安全性は保てないため、あらゆる種類の不正な閲覧が起こり得ると考えてほしい。さらに1つのドメイン内でだけ機能するため、サイトがさまざまなドメイン名で分割されている場合、クッキーはその間を移動できない。

プラットフォームによってはクッキーが有効かどうかを判断できるが、その機能がない場合、URLの再読み込み機能を使う。データ量が極めて少ない場合、このURLの再読み込み機能によってクライアントセッションステートを簡素化できる。

17.1.2 | 使用するタイミング

クライアントセッションステートには多くのメリットがある。クラスタリングとフェイルオーバーは、ステートレスサーバオブジェクトをサポートする場合、特に有効に機能する。もちろん、クライアントに障害が発生した場合にはすべてを失うが、そのようになってもユーザは覚悟して使っている場合が多い。

クライアントセッションステートのとらえ方は、データの量で幾何級数的に違ってくる。2～3のフィールドの場合、すべては有効に機能する。データが大量の場合、データの格納場所、およびリクエストで要求されるものを転送する時間コストの問題がとても深刻になる。

実行中のスタートポロジーにhttpクライアントが含まれている場合は特にこの傾向が強い。さらに、セキュリティの問題もある。クライアントに送信されるデータは、閲覧されたり、修正されたりすることがとても多い。これを防止する唯一の方法は暗号化であるが、それぞれのリクエストで暗号化および複号化を行うことは、パフォーマンスに対して負荷がかかる。暗号化しない場合、不正な閲覧から隠ぺいした方がよいと思われるものを確実に送信しないようにする必要がある。fingerサービスも不正に閲覧される場合があるため、送信したものと、返信されたものが同じであると考えるべきではなく、返信されるデータの妥当性確認を再度徹底的に行わなければならない。

セッションの識別のためには、**クライアントセッションステート**を常時使う必要がある。幸い、この**クライアントセッションステート**は1つだけなので、上記のどの仕組みでも負荷はかからない。さらに、セッションスティーリングについても注意が必要である。セッションスティーリングは、悪意を持つユーザが、他人のセッションを妨害できるかどうかを確認するため自らのセッションIDを変更するときに発生する。プラットフォームではこのリスクを軽減するため、ランダムなセッションIDを採用しているが、そうでない場合にはハッシュを使ってシンプルなセッションIDを実行する。

17.2 | サーバセッションステート

サーバシステムのセッションステートを直列化形式で保持する。

17.2.1 | 動作方法

このパターンの最もシンプルな形式は、セッションオブジェクトをアプリケーションサーバのメモリ上に保持することだ。こうしてセッションIDがキーとなるセッションオブジェクトを保持するメモリ上に、数種類のマッピングを持つことができる。つまり、クライアントが行うべき唯一の動作は、セッションIDを提供し、セッションオブジェクトをマッピングから検索しリクエストを処理することである。

この基本的なシナリオは、アプリケーションサーバがタスクを実行するのに必要なメモリを十分に搭載していることを前提としている。さらに、アプリケーションサーバは1つである。つまりクラスタリングが行われていないことを前提としているので、アプリケーションサーバに障害が発生した場合、当然セッションが中止され、それまで行われたすべての処理が消失する。

ほとんどのアプリケーションでこのような状況が問題になることはないが、一部のアプリケーションによっては問題となる場合もある。この状況を回避するための対処方法は複数ある。それにより基本的にシンプルなパターンにいくらか複雑さが加わり、共通のバリエー

ションが作られる。

最初は、セッションオブジェクトによって保持されるメモリリソースの処理の問題である。これは**サーバセッションステート**によく見られる難点である。もちろん、その解決方法はメモリ上にリソースを保持するのではなく、すべてのセッションステートを永続的なストレージのメメント[Gang of Four]に直列化することである。この方法を使う場合には、次の2つの点を決定する必要がある。どのような形式で**サーバセッションステート**を永続化するかという点と、どこで永続化するかという2点である。

サーバセッションステートの特長はプログラミングのシンプルさであるため、使用する形式はできる限りシンプルなものでなければならない。いくつかのプラットフォームでは、極めて簡単にオブジェクトのグラフを直列化できるシンプルなバイナリ直列化メカニズムを提供している。別の方法は、テキストや流行のXMLファイルなど別の形式で直列化することである。

一般的には、プログラミングがほとんど不要なバイナリ形式の方が容易である。一方、テキスト形式の場合、少しのコードが必要になる。バイナリ直列化ではディスク容量も節約できる。総ディスク容量が問題になることは稀だが、直列化した大型のグラフの場合、メモリへのアクティブ化には、そのサイズに応じた時間がかかる。

バイナリ直列化には、一般的に次の2つの問題がある。1つは、直列化形式は可視ではない。もっとも、読みたいと思わなければ問題はない。2つ目は、バージョニングの問題が発生する可能性があることである。たとえば、直列化した後にフィールドを追加してクラスを修正した場合、それを再読み込みできない場合がある。もちろん、複数のマシンでクラスタ化されている365日24時間稼動のノンストップサーバを使って、アップグレードされているマシンとされていないマシンが存在する環境でない限り、サーバソフトウェアのアップグレードの影響を受けるセッションは少ない。

このため、**サーバセッションステート**をどこに格納するか決定することである。明解に言えることは、アプリケーションサーバ自体のファイルシステムかローカルデータベースのいずれかに格納すべきだということである。シンプルな方法であるがクラスタリングやフェイルオーバーを十分にサポートしない場合がある。サポートするには、パッシブ化された**サーバセッションステート**で一般的にアクセスできる場所が必要である。サーバの起動に長い時間がかかるというコストはあるが、クラスタリングとフェイルオーバーがサポートできる。もちろんキャッシュ機能によってコストの大部分は削減することができる。

この解説は皮肉にも、セッションIDのインデックス付きのセッションテーブルを使って、直列化された**サーバセッションステート**をデータベースに格納するということを、推し進めているかもしれない。直列化された**サーバセッションステート**を保持するために、テーブルには**シリアライズLOB**が必要になる。大型のオブジェクトを処理する場合、データベースのパフォーマンスは変動するため、この方法ではデータベースに左右されることが極めて多い。

現時点では、私は**サーバセッションステート**と**データベースセッションステート**との境界線の狭間に立っている。境界はまったく任意のもので、私は**サーバセッションステート**のデータをテーブルの形式に変換するポイントに対してその一線を画した。

サーバセッションステートをデータベースに格納する場合、特に一般用のアプリケーションでのセッション削除の処理について考える必要がある。1つの方法は、古いセッションを検索して削除するデーモンを持つことであるが、この方法ではセッションテーブル上で多くの類似が発生することがある。Kai Yu は、成功した手法について次のように話してくれた。セッションテーブルを 12 個のデータベースセグメントに分割し、2 時間ごとにセグメントを循環し、最も古いセグメント中のデータを削除し、そこにすべてのデータを挿入するよう指定する。この方法では、24 時間アクティブなセッションが突然落ちることもあるが、これはめったに起こらないため問題になることはないとのことである。

このようなバリエーションを実行するには、多大な労力が必要となるが、幸いなことに最近のアプリケーションサーバは、徐々にこれらの機能を自動的にサポートするようになってきている。アプリケーションサーバのベンダーにとっては助けとなる作業であると思われる。

Java の実装

サーバセッションステートの最も普及している 2 つの技法は、http セッションとステートフルセッション Bean を使う技法である。http セッションはシンプルな方法であり、これを使って Web サーバはセッションデータを格納する。ただしこの方法によって、多くはサーバアフィニティによりフェイルオーバーに対処できなくなる。ベンダーによっては、アプリケーションサーバで使うデータベースに、http セッションデータを格納する共有 http セッション機能を実装している(もちろん手作業でもできることではある)。

ステートフルセッション Bean を使うというもう 1 つの方法では、EJB サーバが必要になる。EJB コンテナは永続性とパッシブ化を処理するが、とてもプログラムしやすい。ただし主な短所は、サーバアフィニティの回避をアプリケーションサーバに要求する仕様になっていないことである。しかし、この機能を装備しているアプリケーションサーバもある。それがアプリケーションサーバの 1 つである IBM の WebSphere であり、ステートフルセッション Bean を DB2 の BLOB に直列化して、複数のアプリケーションサーバが状態を取得できる。

ステートレスセッション Bean の方がパフォーマンスがよいため、ステートフル Bean の代わりに使うべきだと言う人は多い。率直に言って、これはまったくの誤りである。まずステートフルとステートレス間の速度の違いが、使っているアプリケーションに影響を与えるかどうかを負荷テストで調べてみるとよい。ThoughtWorks 社には、数百人もの同時ユーザでテストを行った負荷テスト済みの

アプリケーションがある。そのアプリケーションでは、ユーザ負荷サイズでステートフル Bean を使っているが、何のパフォーマンスの問題も発生しない。現在の負荷ではパフォーマンスメリットがあまりなく、ステートフル Bean の方が容易な場合は、こちらを使うべきである。ステートフル Bean で注意すべき別の理由は、フェイルオーバーがベンダー依存になるという問題である。パフォーマンスの違いは、高負荷の場合にしか発生しない。

別の代替方法としては、エンティティ Bean がある。全体的に、私はエンティティ Bean にとても否定的だが、セッションデータの**シリアライズ LOB** を格納するために使うことはできる。とてもシンプルで、あまり問題は発生しない。

.NET の実装

サーバセッションステートは、ビルトインのセッションステート機能を使うと実装しやすい。デフォルトの場合、.NET はセッションデータをサーバプロセス自体に格納する。さらに、ステートサービスを使ってストレージを調整することができる。ステートサービスは、ローカルマシンまたはネットワーク上のマシンのいずれかに常駐できる。個別のステートサービスを使うと、Web サーバをリセットし、セッションステートを保持できる。プロセス内の状態と構成ファイル内のステートサービスの間で変更を行うと、アプリケーション全体の変更は必要ない。

17.2.2 | 使用するタイミング

サーバセッションステートの一番の魅力はシンプルさである。大半は、これを動作させるプログラミングがまったく必要ない。プログラミングしなくてもよいかどうかは、メモリ内の実装をどうするかで決まる。実装が必要な場合は、アプリケーションサーバプラットフォームの機能が大きな助けになる。

機能がなくても実装はわずかな労力で済む。BLOB をデータベーステーブルに直列化すると、サーバオブジェクトをテーブルの形式に変換するよりかなりの労力を削減できる。

プログラミングの成果が試されるのは、セッションの保守である。クラスタリングやフェイルオーバーをするためサポートを行う場合、特にその成果は大きい。ただし、処理データ量があまり多くない場合やセッションデータが容易にテーブル形式に変換できる場合、他の選択肢よりも多くの問題を生じることもある。

17.3 | データベースセッションステート

コミットされたデータのセッションデータをデータベースに格納する。

17.3.1 | 動作方法

呼び出しがクライアントからサーバへ送られる場合、サーバオブジェクトはまずデータベースからリクエストに必要なデータを抜き出す。次にサーバオブジェクトは処理を行い、データをデータベースに返して保存する。

データベースから情報を抜き出すため、サーバオブジェクトはセッションについての情報を必要とする。ここで、クライアントに格納されているセッションIDの番号は必須である。通常この情報は、データベース内でデータを見つけるために必要なキー群でしかない。

含まれているデータは、ローカルで現在相互作用しているデータとすべての相互作用でコミットされたデータの混合である。

ここで考えるべき問題の1つは、セッションデータはセッションにローカルなものであり、セッション全体がコミットされるまでは、システムの別の部分に影響を与えてはならないことである。そのため、セッションのOrderで作業して、中間状態をデータベースに保存したい場合、セッションの最後で確認されたOrderとは異なった方法で処理する必要がある。理由は、書籍の在庫や日々の収益などに関するデータベースに対し実行するクエリー内に、保留中のOrderを頻繁に表示しないためである。

では、どのようにセッションデータを分離するのか。セッションデータを持つデータベース行に、フィールドを追加することが1つの方法である。このシンプルな形式で必要なのは、論理`isPending`フィールドだけである。しかし、より良い方法は、セッションIDをPending（保留）フィールドとして格納することである。これで特定のセッションデータがとても見つけやすくなる。レコードデータだけを求めるクエリーでは、`sessionID is not NULL`句を使って修正したり、あるいはレコードデータをフィルタするビューが必要である。

セッションIDフィールドを使うのは、侵略的な解決方法である。それは、レコードデータベースにアクセスするすべてのアプリケーションがフィールドを認識し、セッションデータを取得しないようにするためである。ビューが有効に機能し、他への影響を回避する場合もある。しかし、ビュー自身のコストが増大することも多い。

2つ目の代替方法は、Pending（保留）テーブルの個別セットである。データベースにすでにOrdersおよびOrder Linesテーブルがある場合、Pending Orders（保留注文）およびPending Order Lines（保留注文品目）のテーブルを追加する。そして、Pendingセッションのデータは Pendingテーブルに保存する。そして、それがレコードデータとなったときに実際のテーブルに保存する。これで多くの影響を回避できるが、テーブル選択ロジッ

クをデータベースマッピングコードに追加するため、ある程度複雑にならざるを得ない。

レコードデータには、保留データに適用されない整合性のルールがある。この場合、Pending テーブルを使うことで、必要でない場合はルールを無視し、必要な場合はルールを実施することができる。同様に妥当性確認ルールも保留データとして保存する場合は適用されない。セッション内の場所によっては、異なる妥当性確認ルールが適用される場合もあるが、これはサーバのオブジェクトロジックに示される。

Pending テーブルを使う場合、テーブルの正確なクローンとして動作する必要がある。マッピングロジックもできる限り同じものにする。2 つのテーブル間では同じフィールド名を使う必要があるが、Pending テーブルにセッション ID フィールドを追加して、セッションのデータを見つけやすくする。

セッションが中止または破棄される場合、セッションデータを空にする必要がある。セッション ID を使って、ID のついたデータを見つけて削除する。ユーザが通知せずにセッションを破棄する場合、タイムアウトメカニズムが必要になる。数分ごとに実行されるデーモンは、古いセッションデータを探すことができる。この場合、セッションとの最新の相互作用が実行された時間を保存するテーブルが必要になる。

更新によってロールバックはとても複雑なものになってしまう。セッション全体のロールバックができるセッション内の既存の Order を更新する場合、どのような方法でロールバックを行うのだろうか。1 つの選択肢は、セッションを中止できないようにすることである。既存のレコードデータの更新も、リクエストの最後でレコードデータの一部になる。この方法はシンプルで、ユーザが直感的にわかりやすい。代替方法は、Pending フィールドまたは Pending テーブルのいずれかを使うことである。修正しようとするデータを Pending テーブルにコピーして修正し、セッションの最後でコミットし、そのデータをレコードテーブルに返すことは容易である。セッション ID がキーの一部になる場合にだけ、Pending フィールドでこの方法が行える。新旧の ID を同時に同じテーブルで保存することができるが、これはとても複雑なものになる。

セッションを処理するオブジェクトだけから読み込みができる個別の Pending テーブルを使う場合、データをテーブル形式化するポイントがほとんどない場合は、**シリアライズ LOB** を使うことで改善できる。ここからは**サーバセッションステート**の領域である。

保留データをまったく持たなければ、保留データのすべての問題を回避できる。つまり、データがすべてレコードデータと見なされるようなシステムを設計することである。もちろんこれが常に有効であるとは限らないし、できてもとても面倒な場合があるため、設計者は明示的な保留データについては十分に考えなくてはいけない。しかし、この選択肢が選べるなら**データベースセッションステート**がとても処理しやすくなるといえる。

17.3.2 | 使用するタイミング

データベースセッションステートはセッションステートを処理する1つの代替方法であり、**サーバセッションステート**や**クライアントセッションステート**と比較してみる必要がある。

このパターンについて考えるべき第1の点は、パフォーマンスである。サーバ上でステートレスオブジェクトを使うことによってパフォーマンスが向上するため、プールや簡単なクラスタリングを行える。しかしその分、それぞれのリクエストでデータベースへの挿入と抽出に時間がかかる。サーバオブジェクトをキャッシュすれば、このコストを削減することができる。キャッシュがヒットする場合は、データベースからデータを読み込む必要はなくなるが、それでも書き込みコストは必要になる。

第2は、プログラミングに要する労力であり、ほとんどはセッションステートの処理に関する作業であるが、セッションステートがなく、リクエストでデータをレコードデータとして保存する場合は、明らかにこのパターンを選択する。理由は、労力とパフォーマンスのいずれでも無駄がないからである（サーバオブジェクトをキャッシュする場合）。

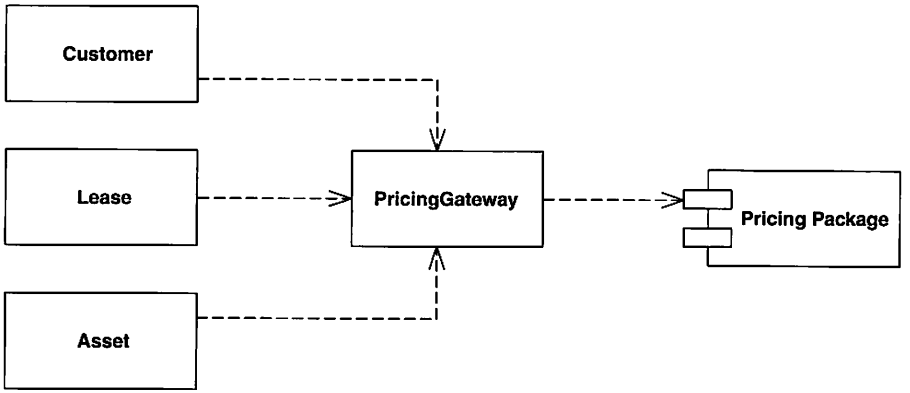
データベースセッションステートと**サーバセッションステート**のどちらを選択するかの最も大きな問題は、使っている特定のアプリケーションサーバにおいて、**サーバセッションステート**でクラスタリングとフェイルオーバーをサポートするのがどのくらい容易かということにある。**データベースセッションステート**のクラスタリングとフェイルオーバーの方がよりわかりやすい方法である。

第18章

ベースパターン

18.1 ゲートウェイ

外部システムまたはリソースへのアクセスをカプセル化するオブジェクト。



ソフトウェアが孤立して動作することは稀である。オブジェクト指向システムでも、リレーショナルデータベーステーブル、CICS トランザクション、XML データ構造など、オブジェクト以外のものとつなぎ合わさっている。

このような外部リソースにアクセスする場合、リソースから API を取得する。この API は、リソースをアカウントに取り入れるためやや複雑になっている。リソースを理解しようとする人は、リソースの API も理解すべきである。たとえそれがリレーショナルデータベース用の JDBC や SQL、あるいは XML 用の W3C や JDOM のいずれであっても理解すべきである。API の知識が不足していると、ソフトウェア全体の理解もなく、将来のある時点で、何らかのデータをリレーショナルデータベースから XML メッセージにシフトする必要があるときなどに変更がとても難しくなる。

対策は一般的なものなので、特に取り上げて述べる必要もないだろう。特定のAPIコードをインタフェースが、オブジェクトのように見えるクラスにラップすればよい。他のオブジェクトは、**ゲートウェイ**を介してリソースにアクセスし、シンプルなメソッド呼び出しを特定のAPIに変換する。

18.1.1 | 動作方法

これは極めてシンプルなラッパーパターンである。まず外部リソースを取得する。アプリケーションはどのように処理する必要があるだろうか。使用方法に応じたシンプルなAPIを作成し、**ゲートウェイ**を使って外部リソースを変換する。

ゲートウェイの主な使用法の1つは、**サービススタブ**を適用するための優れたポイントとして使うことである。**ゲートウェイ**の設計を変更して、**サービススタブ**を適用しやすくする場合が多い。この方法を採用することを躊躇してはいけない。この方法で**サービススタブ**を配置することによって、システムのテストと書き込みが容易になる。

ゲートウェイは、できる限りシンプルにするべきである。外部サービスの適合とスタブ機能に優れたポイントを提供する基本的な役割に集中すればよい。**ゲートウェイ**をできる限り最小化してタスクを処理する必要がある。より複雑化したロジックは、**ゲートウェイ**のクライアントに配置する。

ゲートウェイを作成するために、コード生成を使うと有効である場合が多い。外部リソースの構造を定義することにより、外部リソースをラップするGatewayクラスを生成することができる。また、リレーショナルメタデータを使ってリレーショナルテーブル用のWrapperクラスを、あるいはXMLスキーマまたはDTDを使ってXML用の**ゲートウェイ**のコードを生成することができる。これで**ゲートウェイ**は単純な仕掛けとして動作するようになる。他のオブジェクトは多少複雑な各操作を実行する。

1つ以上のオブジェクトという観点から**ゲートウェイ**を構築することが優れたストラテジーである。明白な形式は、バックエンドとフロントエンドの2つのオブジェクトを使うことである。バックエンドは、外部リソースに対して最小限のオーバーレイを果たすが、リソースのAPIはシンプルにはならない。次のフロントエンドは、複雑なAPIをアプリケーションが使えるように、より使いやすいものに変換する必要がある。外部サービスのラッピングとニーズへの適合が極めて複雑な場合には、この手法は有効である。それぞれが1つのクラスで処理されるからである。逆に、外部サービスのラッピングがシンプルな場合は、1つのクラスでラッピングなどすべてを処理することができる。

18.1.2 | 使用するタイミング

外部へのインタフェースが複雑な場合は常にゲートウェイの使用を考えるべきである。複雑さをシステム全体に拡大せずに、それらを格納するゲートウェイを使うのである。ゲートウェイを作成することで、システムの他の場所にあるコードを読み込むことがとても容易になる。

ゲートウェイは、サービススタブの明確なポイントで、システムをテストしやすくする。外部システムのインタフェースが有効でも、サービススタブの適用は、第1段階として役立つ。

さらにゲートウェイのメリットは、1種類のリソースから別のリソースにスワップアウトしやすくすることである。Gateway クラスを変更するだけでリソースに何らかの変更を行うことができ、変更はシステムの他の部分には影響しない。ゲートウェイは、保護されたバリエーションでしかもシンプルで強力な形式と言える。ゲートウェイを使うことへの議論の焦点は、こうした柔軟性に関するものになる。しかし、リソースの変更を考えていない場合でも、ゲートウェイがもたらすシンプルさとテストの容易さは大変有効である。

上述したような一対のサブシステムがある場合、サブシステムを分離する別の選択はマッパーである。ただし、マッパーはゲートウェイより複雑であるため、私は外部リソースへのアクセスへはゲートウェイを使っている。

このパターンをファサードやアダプター[Gang of Four]などの既存のこのパターンとは異なった、新しいパターンにすべきかどうかについてかなり悩んだことがある。結局、このパターンを他のパターンから分離することにしたが、それはこの区別が有効だと判断したからである。

- ファサードは複雑化したAPIをシンプルにする一般向けのサービスを開発するときに使う。ゲートウェイは、特定の使用目的を持つクライアントによって記述される。ファサードは常に隠ぺいされているインタフェースと考えられているが、ゲートウェイはラップされたファサード全体をコピーして置換またはテストのために使われる。
- アダプターは実装するインタフェースを変更し、別のインタフェースに適合させる。実装をゲートウェイインタフェースにマッピングするためにアダプターを使用することはあるが、ゲートウェイを使う場合は既存のインタフェースが存在しないので、アダプターはゲートウェイの実装の一部になる。
- メディエータは複数のオブジェクトを分離する。複数のオブジェクトは互いに認識しないが、メディエータは認識できる。ゲートウェイでは2つのオブジェクトのみ存在し、ラップされたリソースはゲートウェイを認識していない。

18.1.3 ｜例：固有のメッセージングサービスへのゲートウェイ（Java）

同僚のMike Rettigと私はこのパターンについて話し合った。彼は、EAI（Enterprise Application Integration）ソフトウェアとのインタフェースを処理するためにこのパターンをどのように使ったかを話してくれた。私たちはこのパターンは**ゲートウェイ**の例として読者に素晴らしいアイデアをもたらすものになるだろうと判断した。

事象を一般的なシンプルなレベルに保つため、メッセージサービスを使ってメッセージを送信するだけのインタフェースへの**ゲートウェイ**を構築する。インタフェースは1つのメソッドだけで構築されている。

```
int send(String messageType, Object[] args);
```

最初の引数はメッセージの型を示す文字列であり、2番目はメッセージの引数である。メッセージングシステムを使って任意の種類のメッセージを送信できるため、このような汎用インタフェースが必要になる。メッセージシステムを構築するときには、システムが送信するメッセージの型と対応する引数の数や型を指定する。これで文字列「CNFRM」で確認メッセージを構成し、文字列、整数、ティッカーコードの文字列として、ID番号の引数を持つことができる。メッセージングシステムは引数の型を確認して、誤ったメッセージ、または誤った引数を持つメッセージを送信した場合にエラーを生成する。

これは必要不可欠な優れた柔軟性を提供する方法ではあるが、汎用インタフェースは明示的ではないため使いにくい。インタフェースを見ただけでは、規定のメッセージの型や特定のメッセージの種類に必要な引数はわからない。代わりに必要になるのは、以下のようなメソッドを持つインタフェースである。

```
public void sendConfirmation(String orderID, int amount, String symbol);
```

この場合、メッセージを送信するドメインオブジェクトが必要なら、以下のとおりに処理する。

```
class Order...

    public void confirm() {
        if (isValid()) Environment.getMessageGateway().sendConfirmation
                (id, amount, symbol);
    }
```

メソッドの名前を見ればどのようなメッセージを送信しているかがわかり、引数に型と名

前が与えられる。汎用メソッドよりも、とても呼び出しやすいメソッドになっている。役立つインタフェースを作成するのが、**ゲートウェイ**の役割である。しかし、これはメッセージングシステムでメッセージ型の追加または変更を行うときに常にGatewayクラスを変更する必要があり、またメッセージ型を変更しない場合でも呼び出しコードを変更する必要があることを意味する。この方法で、少なくともコンパイラでクライアントを見つけ、エラーをチェックすることはできる。

さらに別の問題もある。インタフェースのエラーを受け取る場合は、リターンエラーコードが提供され、エラーが通知される。0は正常終了を示し、他のあらゆる数字は失敗を意味し、それぞれの番号が異なったエラーを意味している。Cプログラマが行う一般的な方法ではあるが、Javaで実行する方法とは言えない。Javaでは例外を起こしてエラーを表示する。そのため、**ゲートウェイ**のメソッドはリターンエラーコードよりも例外を発生させる。

起こりそうなエラーはすべて、私たちが無視しがちなものである。次の2つのエラーだけに話を絞ろう。それは不明なメッセージ型によるメッセージの送信と、引数の1つがnull値であるメッセージの送信である。リターンコードは、メッセージングシステムのインタフェースで定義される。

```
public static final int NULL_PARAMETER = -1;
public static final int UNKNOWN_MESSAGE_TYPE = -2;
public static final int SUCCESS = 0;
```

これら2つのエラーには大きな違いがある。不明なメッセージ型のエラーはGatewayクラスのエラーを示す。クライアントは完全に明示的なメソッドだけしか呼び出さないため、クライアントはこのエラーを生成しない。しかし、クライアントはnull値を渡すため、null値パラメータエラーは表示される。エラーはプログラマのエラーを示し、チェック済みの例外ではない（特定のハンドラーを記述するようなエラーではない）。**ゲートウェイ**は自らnull値のチェックができるはずである。しかし、メッセージングシステムが同じエラーを発生させるような場合は、チェックが有効ではない可能性がある。

このため**ゲートウェイ**は、明示的なインタフェースを汎用インタフェースに変換し、リターンコードを例外に変換する処理をする。

```
class MessageGateway...

    protected static final String CONFIRM = "CNFRM";
    private MessageSender sender;
    public void sendConfirmation(String orderID, int amount, String symbol) {
        Object[] args = new Object[]{orderID, new Integer(amount), symbol};
        send(CONFIRM, args);
```

```
    }
    private void send(String msg, Object[] args) {
        int returnCode = doSend(msg, args);
        if (returnCode == MessageSender.NULL_PARAMETER) throw new
            NullPointerException("Null Parameter passed for msg type: " + msg);
        if (returnCode != MessageSender.SUCCESS) throw new IllegalStateException(
            "Unexpected error from messaging system #:" + returnCode);
    }
    protected int doSend(String msg, Object[] args) {
        Assert.notNull(sender);
        return sender.send(msg, args);
    }
```

このままでは、doSend メソッドの重要な部分を見つけにくい。そこでゲートウェイのもう1つの重要な役割、つまりテスト機能を採用する。メッセージ送信サービスがない場合もゲートウェイを使うオブジェクトをテストできる。それにはサービススタブを作成する必要がある。この場合、ゲートウェイスタブはゲートウェイのサブクラスであり、doSend を上書きする。

```
class MessageGatewayStub...

    protected int doSend(String messageType, Object[] args) {
        int returnCode = isMessageValid(messageType, args);
        if (returnCode == MessageSender.SUCCESS) {
            messagesSent++;
        }
        return returnCode;
    }
    private int isMessageValid(String messageType, Object[] args) {
        if (shouldFailAllMessages) return -999;
        if (!legalMessageTypes().contains(messageType))
            return MessageSender.UNKNOWN_MESSAGE_TYPE;
        for (int i = 0; i < args.length; i++) {
            Object arg = args[i];
            if (arg == null) {
                return MessageSender.NULL_PARAMETER;
            }
        }
        return MessageSender.SUCCESS;
    }
    public static List legalMessageTypes() {
        List result = new ArrayList();
        result.add(CONFIRM);
```

```
        return result;
    }
    private boolean shouldFailAllMessages = false;
    public void failAllMessages() {
        shouldFailAllMessages = true;
    }
    public int getNumberOfMessagesSent() {
        return messagesSent;
    }
```

送信されたメッセージ数を取得することは、ゲートウェイが正常に動作しているかどうかをテストするのに役立つシンプルな方法である。

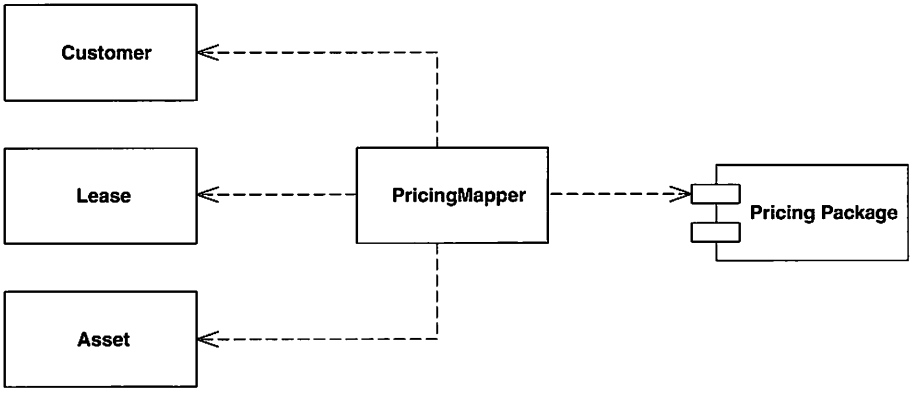
```
class GatewayTester...

    public void testSendNullArg() {
        try {
            gate().sendConfirmation(null, 5, "US");
            fail("Didn't detect null argument");
        } catch (NullPointerException expected) {
        }
        assertEquals(0, gate().getNumberOfMessagesSent());
    }
    private MessageGatewayStub gate() {
        return (MessageGatewayStub) Environment.getMessageGateway();
    }
    protected void setUp() throws Exception {
        Environment.testInit();
    }
```

クラスが既知の場所から発見できるように**ゲートウェイ**を設定して、静的な環境インタフェースを作っている。**プラグイン**を使って、システム構成時にサービスとスタブの間を切り換えたり、テストを設定するルーチンで**サービススタブ**を使う環境を初期化したりする。

この場合、ゲートウェイのサブクラスを使ってメッセージングサービスのスタブを作成するが、サービス自体のサブクラス化（または再実装）を行うという別の方法もある。テストのため、送信する**サービススタブ**にゲートウェイを接続する。サービスの再実装があまり困難でない場合には有効である。サービスのスタブを作成するか、またはゲートウェイのスタブを作成するかを選択する。場合によっては、ゲートウェイのクライアントをテストするスタブ化されたゲートウェイと、ゲートウェイ自体をテストするスタブ化されたサービスの両方を使うことで、両方のスタブの作成が役立つ場合もある。

18.2 | マッパー

独立した2つのオブジェクト間の通信を設定するオブジェクト。



互いに環境設定が不明な状態で、2つのサブシステム間の通信を設定しなければいけない場合がある。設定が必要な理由は、2つのサブシステムを修正できない、あるいは修正はできるが2つのサブシステムの間、または2つのサブシステムと他の分離している要素との間に依存性を作成したくないからである。

18.2.1 | 動作方法

マッパーはサブシステム間の分離レイヤである。レイヤは、どちらのサブシステムにも互いを認識させることなく、両者間の通信の詳細を制御する。

多くの場合、マッパーは、1つのレイヤから別のレイヤにデータをシャッフルする。シャッフル動作をアクティブにすると、極めて容易に動作方法を確認できる。マッパーを使うときに難しいのは、マッパーを呼び出す方法の決定である。それは、マッピングしているサブシステムのいずれからも、マッパーを直接呼び出せないためである。第3のサブシステムがマッピングを開始して、マッパーを呼び出す場合もある。代替方法としては、1つまたは他のサブシステムのオブザーバー[Gang of Four]をマッパーとして機能させる方法がある。この方法では、サブシステムの1つのイベントをリッスンすることによって呼び出せる。

マッパーの動作方法は、マッピングレイヤの種類に左右される。マッピングレイヤで普及している例では、**データマッパー**を使う。**マッパー**の使用方法についての詳細は、**データマッパー**を参照してほしい。

18.2.2 | 使用するタイミング

基本的に、マッパーはシステムの異なる部分を分離させる。分離させたい場合は、マッパーとゲートウェイのどちらを使うかを選択する。コードの記述とその後の両面で、マッパーよりもゲートウェイの方がはるかにシンプルであるため、ゲートウェイを選択することが多い。

どちらのサブシステムも相互作用に依存しないようにする場合はマッパーだけを使う。マッパーが重要になるのは、サブシステム間の相互作用が特に複雑で両サブシステムの目的から逸脱する場合である。このように、エンタープライズアプリケーションでは、マッパーはデータマッパーなどのようにデータベースとの相互作用に使うことがほとんどである。

マッパーは異なる要素に対して使われるため、メディエータ[Gang of Four]にも似ている。ただし、メディエータを使うオブジェクトは、お互いのオブジェクトは認識していなくてもメディエータを認識している。これに対して、マッパーが分離するオブジェクトは、マッパーさえも認識していない。

18.3 | レイヤスーパータイプ

レイヤのすべてのタイプに対して、スーパータイプとしての役割を果たすタイプ。

1つのレイヤのオブジェクトが、システム全体では複製されたくないメソッドを持つ場合が多い。この振る舞いを共通のレイヤスーパータイプに移動させる。

18.3.1 | 動作方法

レイヤスーパータイプはシンプルな考えに基づくとても短いパターンである。必要なのは、1つのレイヤ内にあるオブジェクトのスーパークラスである。たとえば、ドメインモジュールのドメインオブジェクトであるDomain Object（ドメインオブジェクト）スーパークラスがある。一意フィールドの格納や処理などの共通の機能をレイヤスーパータイプに移動できる。同様に、マッピングレイヤのすべてのデータマッパーが、共通のスーパークラスを持つすべてのドメインオブジェクトに依存することもできる。

レイヤに1種類以上のオブジェクトがある場合、複数のレイヤスーパータイプを持つのは有用である。

18.3.2 | 使用するタイミング

レイヤのオブジェクトから共通の機能を取得する場合に、**レイヤスーパータイプ**を使うとよい。私は多数の共通機能を利用するため、これを自動的に行っている場合が多い。

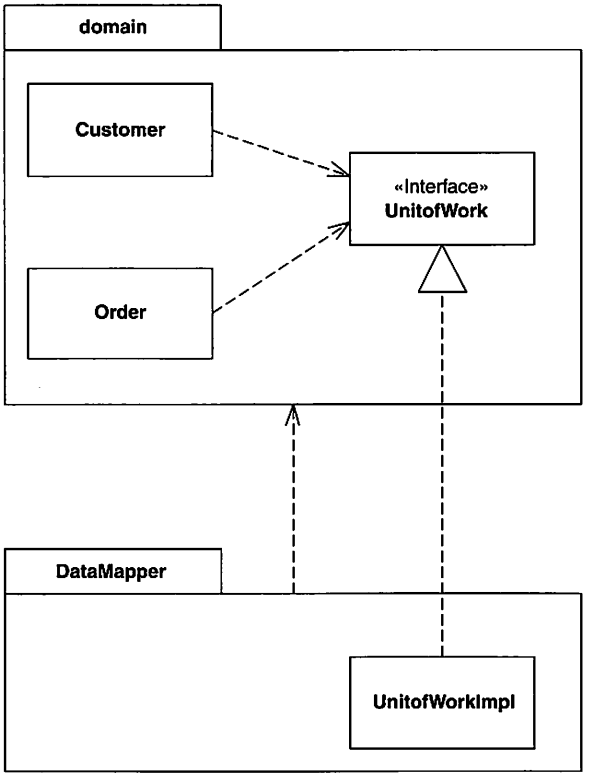
18.3.3 | 例：ドメインオブジェクト（Java）

ドメインオブジェクトでは、ID 処理のための共通のスーパークラスを持つ。

```
class DomainObject...

   private Long ID;
   public Long getID() {
      return ID;
   }
   public void setID(Long ID) {
      Assert.notNull("Cannot set a null ID", ID);
      this.ID = ID;
   }
   public DomainObject(Long ID) {
      this.ID = ID;
   }
   public DomainObject() {
   }
```

18.4 | セパレートインタフェース

実装から分離したパッケージでインタフェースを定義する。



システム開発を行うときには、システムの各部分間の結合を最小限にすることによって、設計の質を向上させることができる。これを実現するための優れた方法は、クラスをパッケージにまとめてパッケージ間の依存性を制御することである。1つのパッケージのクラスが、どのように別のパッケージのクラスを呼び出すかというルールに従うことでこれを行う。このルールはたとえば、ドメインレイヤのクラスは、presentation（プレゼンテーション）パッケージのクラスを呼び出してはならないというようなものである。

ただし、一般的な依存構造にあてはまらないメソッドを呼び出すこともある。この場合1つのパッケージで**セパレートインタフェース**を使ってインタフェースを定義し、別のパッケージでインタフェースを実装する。この方法によって、インタフェースに依存する必要があるクライアントが、実装を一切心配する必要がなくなる。**セパレートインタフェース**で**ゲートウェイ**との適切なプラグインポイントが提供される。

18.4.1 | 動作方法

パターンはとても簡単に採用することができる。基本的に実装はインタフェースに依存しているが、その逆の依存はないためである。これは、インタフェースと実装を別々のパッケージに配置でき、implementation（実装）パッケージはinterface（インタフェース）パッケージに依存するということである。他のパッケージはimplementationパッケージに依存することなくinterfaceパッケージに依存する。

当然、ソフトウェアを実行するにはいくつかのインタフェースを実装する必要がある。コンパイル時に2つのパッケージを結び付ける別のパッケージを使うか、またはシステム構成時に**プラグイン**を使うかのいずれかの方法でインタフェースを実装する。

（冒頭のスケッチにあるように）インタフェースはclient（クライアント）パッケージに配置するか、または（図18.1のように）3つ目のパッケージに配置することができる。実装するクライアントが1つだけ、またはクライアントが同じパッケージにある場合にも、同様にインタフェースをクライアントに配置できる。インタフェースのクライアントへの配置については、clientパッケージ開発者がインタフェースの定義の責任を担うのが適切である。基本的にclientパッケージは、clientパッケージが定義するインタフェースを実装する他の任意のパッケージとともに動作するパッケージである。clientパッケージが複数ある場合は、3つ目のinterfaceパッケージを使う方が有効である。クライアントパッケージ開発者がインタフェースの定義の責任を担っていないことを明確にしたい場合にも、この方法が有効である。実装の開発者が責任を担っている場合にも同様である。

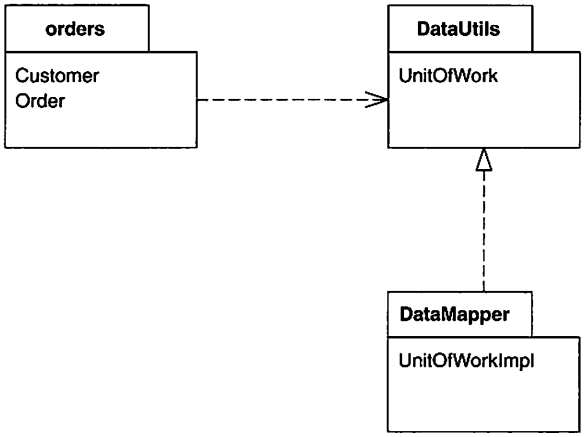


図18.1 ——セパレートインタフェースを3つ目のパッケージに配置

インタフェースに使う言語の特徴を考える必要がある。JavaやC#などのインタフェース構造体を持つ言語の場合、インタフェースキーワードは当然の選択である。しかし、最良

の選択ではないかもしれない。抽象クラスは、クラス内に共通ではあるが、選択できる実装の振る舞いを持てるので、優れたインタフェースを作成できる。

分離したインタフェースについて考えるべき点の1つは、実装のインスタンス化方法である。それにはImplementationクラスの知識が必要になる場合がほとんどである。分離したFactory（ファクトリー）オブジェクトを使うのが一般的な手法である。この場合も**セパレートインタフェース**がある。さらに実装をファクトリーにバインドするので、**プラグイン**を使って処理を行う。つまり、依存性が生じないばかりか、システム構成時までImplementationクラスについて決定する必要がない。

プラグインを使うほどではない場合のシンプルな代替案は、インタフェースと実装の両方を認識する別のパッケージを使って、アプリケーション起動時にオブジェクトのインスタンスを作成するという方法である。**セパレートインタフェース**を使うオブジェクトは、自らをインスタンス化できるか、または起動時にインスタンス化されるファクトリーを持っているかのいずれかである。

18.4.2 | 使用するタイミング

システムの2つの部分間の依存性を排除する必要がある場合、**セパレートインタフェース**を使う。いくつかの例を以下に示す。

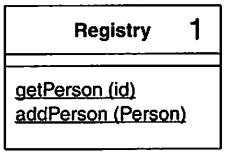
- frameworkパッケージが、特定のアプリケーションコードを呼び出す必要があるときに、共通のケースに対応する抽象コードを構築してframeworkパッケージに挿入する場合。
- 1つのレイヤが、認識すべきでない別のレイヤのコードを呼び出す必要があるときに、レイヤ内にある**データマッパー**を呼び出すドメインコードなどのいくつかのコードを持つ場合。
- 別の開発グループによって開発された関数を呼び出す必要があるが、APIに依存することを望んでいない場合。

私が出会った数多くの開発者は、記述するクラスごとに別のインタフェースを使っている。しかし、アプリケーション開発では、この方法は負荷がかかりすぎると私は考える。個別のインタフェースと実装の保持は、特に（インタフェースと実装を持つ）Factoryクラスも必要になってくるため、無駄な作業も入ってくる場合が多い。アプリケーションのためには、依存性を排除したい場合または複数の独立した実装を持つ場合にだけ、分離したインタフェースを使うことをお勧めする。この方法は、インタフェースと実装をともに配置した後で分離するとき、分離処理を必要な時まで遅延できるシンプルなリファクタリングである。

この方法で依存性を決定して管理することは、度合いによっては不要な作業になる場合がある。依存性だけを考えてオブジェクトを生成し、その後そのインタフェースを使っても問題はない。問題は、構築時に依存性チェックの実行などの依存性ルールを実施しようとする場合に発生する。発生した場合は、すべての依存性を削除しなければならない。小規模システムでは依存性ルールを実施しても問題は大きくはならないが、大規模システムではルールを遵守させることが極めて重要である。

18.5 | レジストリ

他のオブジェクトが一般的なオブジェクトとサービスを検索するために使う既知のオブジェクト。



オブジェクトを検索したい場合、関連する別のオブジェクトから開始し、関連を使って検索したいオブジェクトへ遷移する。Customer の Order すべてを検索したい場合、Customer オブジェクトから開始し、オブジェクト上のメソッドを使って Order を取得する。ただし、開始するためのオブジェクトがない場合もある。たとえば、Customer の ID 番号は認識しているが、参照できないときがある。このような場合、いくつかの種類の lookup（照合）メソッド、つまり find メソッドが必要である。ただし、find メソッドを取得する方法には、どちらにしろ問題も残る。

基本的に**レジストリ**は、たとえ一見するとグローバルには見えない場合でも、グローバルオブジェクトまたは少なくともグローバルオブジェクトに似たオブジェクトである。

18.5.1 | 動作方法

すべてのオブジェクトに言えることだが、インタフェースと実装の観点から**レジストリ**の設計について考える必要がある。インタフェースと実装を同じように考える人が多いが、それは誤りであり、多くのオブジェクトと同様これら 2 つは極めて異なる。

最初に考えるべきなのはインタフェースである。**レジストリ**の場合、私は静的なメソッドのインタフェースを好む。クラス上の静的なメソッドは、アプリケーション内のどこに配置されていても見つけやすい。さらに、好みの任意のロジックを静的なメソッド内でカプセル化

できる。ロジックには、(静的あるいはインスタンスのいずれかの) 他のメソッドへの委譲が含まれている。

ただし、メソッドが静的であるからといってデータも静的なフィールドになければいけないということはない。定数でない限り、私はほとんど静的なフィールドを使わない。

データの保持方法を決定する前に、データのスコープについて考えたい。**レジストリ**のデータは、実行方法によって異なる。プロセス全体でグローバル、スレッドの中でグローバル、セッションの中でグローバルのようにさまざまである。スコープが異なると、異なる実装が必要となるがインタフェースは必要ない。アプリケーションプログラマは、静的なメソッド呼び出しがプロセススコープまたはスレッドスコープのいずれであるかを知る必要はない。異なるスコープに異なる**レジストリ**を持つことも、異なるメソッド内に異なるスコープの1つの**レジストリ**を持つこともできる。

データがプロセス全体で共通の場合、静的なフィールドが1つの選択肢となる。ただし、置換ができないため、私が静的な可変フィールドを使うことは稀である。特定の目的、特にテストのために**レジストリ**の置換が可能であれば、とても役立つ (**プラグイン**はテストのための優れた方法である)。

プロセススコープ**レジストリ**の場合には、選択肢はシングルトン[Gang of Four]である。**レジストリ**クラスには、**レジストリ**インスタンスを保持する静的な1つのフィールドがある。シングルトンを使うと、呼び出し元が内在するデータに明示的にアクセスできる場合が多いが (`Registry.soleInstance.getFoo()`)、私は、Singleton (シングルトン) オブジェクトが隠ぺいされる静的なメソッドの方を好む (`Registry.getFoo()`)。C ベースの言語では、静的なメソッドがプライベートインスタンスデータにアクセスできるメソッドは特に有効である。

シングルトンはシングルスレッドのアプリケーションで広く使われているが、マルチスレッドのアプリケーションでは問題になる可能性がある。理由は、予測不可能な方法で、複数のスレッドが同じオブジェクトの処理を簡単に行ってしまうからである。この問題は同期化することで解決できるが、同期化コードの記述は難しく、バグをすべて取り除く頃には神経を病むことになりかねない。この理由で、私はマルチスレッド環境での可変データに対するシングルトンは勧めない。変更できないデータは何であれスレッドクラッシュの問題に陥ることはない。シングルトンは、不変データに対しては有効である。そのため米国全州のリストのようなデータは、プロセススコープ**レジストリ**のよい例になる。データは、プロセスが開始されて変更の必要がないときにロードする。あるいは稀な場合だが、いくつかの種類のプロセス割り込みによって更新される場合がある。

一般的な**レジストリ**データはスレッドスコープである。例としてはデータベース接続が適している。多くの環境では、Java スレッドローカルなどの形式のスレッド専用記憶域を提供している。代替技術は、スレッドがキーとなるディクショナリであり、スレッドの値が

データオブジェクトである。接続のリクエストがあると、現在のスレッドがディクショナリを照合する。

スレッドスコープデータについて忘れてはならない重要なことは、それがプロセススコープデータと同じように見えるということである。私は、プロセススコープデータにアクセスするときと同じ形式の、`Registry.getDbConnection()`などのメソッドを使う。

ディクショナリ照合も、セッションスコープデータに使える技術である。必要なのはセッションIDであるが、リクエストの開始時にスレッドスコープ**レジストリ**に配置する。後続するセッションデータにアクセスする場合、マッピングのデータを照合する。マッピングはセッションIDを使うセッションがキーとなっていて、セッションIDはスレッド専用のストレージに保持されている。

静的なメソッドを持つスレッドスコープ**レジストリ**を使う場合、複数のスレッドでメソッドを使うとパフォーマンスに問題が起きる場合がある。この場合、スレッドインスタンスに直接アクセスすることでボトルネックを回避できる。

1つの**レジストリ**を持つアプリケーションもあれば、いくつかの**レジストリ**を持つアプリケーションもある。普通**レジストリ**は、システムレイヤまたは実行コンテキストで分割される。私が好む方法は、実装ではなく**レジストリ**を使用方法ごとに分割する方法である。

18.5.2 | 使用するタイミング

レジストリはメソッドのカプセル化にも関わらずグローバルデータであるため、私は使うことを躊躇してしまう。アプリケーションにいくつかの形式の**レジストリ**を見つける場合がとても多いが、**レジストリ**を使うよりは、相互のオブジェクト参照を介してオブジェクトへのアクセスを常に試みることにしている。基本的に、**レジストリ**は最後の手段として使うべきである。

レジストリを使うための代替方法がいくつかある。その1つは、広範囲で必要なデータをパラメータとして渡すことである。問題は、パラメータはメソッド呼び出しに追加されるが、パラメータを必要とするのは呼び出されたメソッドではなく、呼び出しツリーのさらに下方にあるレイヤの別のメソッドであるという点である。データが処理時にほとんど必要ない場合、パラメータに渡すよりも私なら**レジストリ**を使うことにする。

レジストリについてのもう1つの代替方法は、オブジェクト作成時に共通データの参照をオブジェクトに追加することである。コンストラクタに不必要なパラメータを追加することになるが、少なくともそれはコンストラクタでは使われる。この場合、メリットより問題の方が多いが、データがサブクラスで使われるのであれば、この技法で制限することができる。

レジストリの問題の1つは、新しいデータを追加するごとに修正しなければならないことである。このため、グローバルデータのホルダーとしてマッピングを好む人もいるが、私は

明示的なクラスを好む。理由は、明示的なクラスはメソッドを明示的に保ち、検索に使うキーとの混乱が生じないからである。明示的なクラスを使うと、ソースコードや生成された文書を見るだけで、使うデータを調べることができる。マッピングでは、どのようなキーを使っているかを見つけるために、システムの中でマップにデータの読み書きを行っている部分を探したり、すぐに不要になるドキュメントに頼らなければならない。明示的なクラスでは、静的な型付き言語で型の安全性を維持でき、システム拡張に合わせてリファクタリングするためにレジストリの構造をカプセル化する。そのままのマッピングはカプセル化されていないため、実装を隠ぺいすることが難しい。データの実行スコープの変更が必要な場合、特に問題になる。

したがって、レジストリを使う方が適している場合が多い。しかし、どのようなグローバルデータであっても完全に正常であることが実証されるまでは問題が潜んでいることを忘れてはいけない。

18.5.3 ｜例：シングルトンレジストリ（Java）

データベースからデータを読み込んだ後、データを変更して情報に変えるアプリケーションを考えてみよう。データにアクセスするために、行データゲートウェイを使うとてもシンプルなシステムを想定する。システムは、データベースクエリーをカプセル化する Find オブジェクトを持っている。find メソッドは、インスタンスとして作成されるのが最も有効である。理由は、テストのためのサービススタブを作成するために、インスタンスを置き換えることができるからである。インスタンスを配置する場所を選択する必要があるが、レジストリが自明の選択である。

シングルトンレジストリは、シングルトンパターン [Gang of Four] のシンプルな例である。1 つのインスタンスに対し 1 つの静的な変数を持つ。

```
class Registry...

   private static Registry getInstance() {
      return soleInstance;
   }
   private static Registry soleInstance = new Registry();
```

レジストリに格納されているデータはインスタンスに格納される。

```
class Registry...

   protected PersonFinder personFinder = new PersonFinder();
```

ただしアクセスを簡単にするため、静的なパブリックメソッドを使う。

```
class Registry...

   public static PersonFinder personFinder() {
      return getInstance().personFinder;
   }
```

1つの新しいインスタンスを作成するだけで、**レジストリ**を再初期化できる。

```
class Registry...

   public static void initialize() {
      soleInstance = new Registry();
   }
```

テストのために**サービススタブ**を使いたい場合、代わりにサブクラスを使う。

```
class RegistryStub extends Registry...

   public RegistryStub() {
      personFinder = new PersonFinderStub();
   }
```

find メソッド**サービススタブ**は、Person **行データゲートウェイ**の直接書かれたインスタンスを返すだけである。

```
class PersonFinderStub...

   public Person find(long id) {
      if (id == 1) {
         return new Person("Fowler", "Martin", 10);
      }
      throw new IllegalArgumentException("Can't find id: " + String.valueOf(id));
   }
```

メソッドを**レジストリ**に配置して**レジストリ**をスタブモードで初期化し、スタブの振る舞いをサブクラスに保持することによって、テストに必要なコードを分離することができる。

```
class Registry...

   public static void initializeStub() {
      soleInstance = new RegistryStub();
   }
```

18.5.4 | 例：スレッドセーフレジストリ（Java）

(by Matt Foemmel, Martin Fowler)

上記のシンプルな例は、異なるスレッドが独自の**レジストリ**を必要とするマルチスレッドアプリケーションでは動作しない。Java は、スレッドにローカルな Thread Specific Storage（スレッド専用のストレージ）変数[Schmidt] を提供し、スレッドローカル変数と呼ばれている。変数を使って**レジストリ**を作成し、**レジストリ**は 1 つのスレッドに対して一意とする。

```
class ThreadLocalRegistry...

   private static ThreadLocal instances = new ThreadLocal();
   public static ThreadLocalRegistry getInstance() {
      return (ThreadLocalRegistry) instances.get();
   }
```

レジストリは、獲得と解放のためのメソッドを設定する必要がある。一般的には、トランザクションまたはセッション呼び出しの境界で行う。

```
class ThreadLocalRegistry...

   public static void begin() {
      Assert.isTrue(instances.get() == null);
      instances.set(new ThreadLocalRegistry());
   }
   public static void end() {
      Assert.notNull(getInstance());
      instances.set(null);
   }
```

次に、personFind メソッドを格納する。

```
class ThreadLocalRegistry...

   private PersonFinder personFinder = new PersonFinder();;
   public static PersonFinder personFinder() {
      return getInstance().personFinder;
   }
```

begin メソッドと end メソッドで、外部からの呼び出しがレジストリをラップする。

```
      try {
         ThreadLocalRegistry.begin();
         PersonFinder f1 = ThreadLocalRegistry.personFinder();
         Person martin = Registry.personFinder().find(1);
         assertEquals("Fowler", martin.getLastName());
      } finally {ThreadLocalRegistry.end();
      }
```

18.6 | バリューオブジェクト

ID に基づいた等価性を確保していない、Money や Date Range などのシンプルな小型オブジェクト。

さまざまな種類のオブジェクトシステムを使うとき、参照オブジェクトとバリューオブジェクトの相違が役立つことに気づいた。これら 2 つのオブジェクトではバリューオブジェクトの方が小型であり、純粋にオブジェクト指向ではない多くの言語にあるプリミティブな型に類似している。

18.6.1 | 動作方法

参照オブジェクトとバリューオブジェクトの相違の定義は難しい場合がある。広い意味では、バリューオブジェクトは Money（貨幣）オブジェクトや Date（日付）オブジェクトなどの小型のオブジェクトであり、一方参照オブジェクトは Order オブジェクトや Customer オブジェクトなどの大型のオブジェクトである。この定義は便利だが、どちらかというと不適切である。

参照オブジェクトとバリューオブジェクトの主な相違点は、等価性の処理の方法である。参照オブジェクトは ID を等価性の基準として使う。ID は、オブジェクト指向プログラミング言語のビルトイン ID など、プログラミングシステム内の ID であったり、リレーショナルデータベースの主キーなど何らかの ID 番号であったりする。バリューオブジェクトの等

価性の概念はクラス内のフィールド値に基づいている。2つのDateオブジェクトで日、月、年の値が同じ場合は、等価ということになる。

相違点は、2つのオブジェクトを処理する方法で明白になる。**バリューオブジェクト**は小型で容易に生成されるため、参照ではなく値によって渡される場合が多い。March 18, 2001（2001年3月18日）オブジェクトがシステム内にいくつ存在するかを意識する必要はあまりない。さらに、2つのオブジェクトが同じ物理データオブジェクトを共有しているのか、それとも異なっているが同じデータのコピーであるかどうかなどを意識する必要もない。

ほとんどの言語は**バリューオブジェクト**の特定の機能を持っていない。**バリューオブジェクト**が機能するためには、**バリューオブジェクト**を不変、つまり作成時のフィールドがそのまま変更できないようにすることが有効である。理由は、別名割り当てのバグを防止するためである。別名割り当てのバグは、2つのオブジェクトが同じ**バリューオブジェクト**を共有し、一方のOwnerがオブジェクトの値を変更するときに発生する。したがって、Martinの雇用日付が3月18日で、Cindyも同じ日に雇用されたことを知っている場合、Cindyの雇用日付は、Martinの雇用日付と同じに設定する場合がある。Martinが自分の雇用日付の月を5月に変更する場合、Cindyの雇用日付も変更される。正しいかどうかに関係なく変更される。このように値が少ない場合、既存のDate（日付）オブジェクトを新しいオブジェクトに置き換えて、雇用日付を変更する。**バリューオブジェクト**を不変にするだけで完成する。

バリューオブジェクト は完全なレコードとして永続化するべきではない。**バリューオブジェクト**ではなく、**組込バリュー**または**シリアライズLOB**を使うとよい。**バリューオブジェクト**は小型であるため、**組込バリュー**が最良の選択肢である。**組込バリュー**は**バリューオブジェクト**のデータを使うSQLクエリーも発行できるからである。

特に、**バリューオブジェクト**を処理しないJavaなどの言語でバイナリ直列化を多く行う場合、**バリューオブジェクト**の直列化を最適化してパフォーマンスを向上できる。

バリューオブジェクトの例については、**マネー**を参照してほしい。

.NETの実装

.NETは**バリューオブジェクト**を処理するという優れた機能を持っている。C#の場合、オブジェクトはクラスではなく構造体として宣言することによって、**バリューオブジェクト**としてマーク付けされ、値のセマンティクスで**バリューオブジェクト**を処理する。

18.6.2 | 使用するタイミング

等価性を確保するためにID以外を基準にしたい場合、基準を**バリューオブジェクト**で処理するとよい。作成が簡単な小型の任意のオブジェクトでこの方法は有効である。

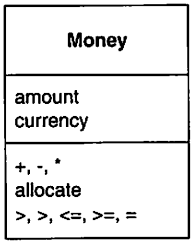
18.6.2.1 ■ 名前の衝突

かなり以前から、**バリューオブジェクト**という名称がこのパターンに使われていることを知っている。残念なことに、最近J2EEコミュニティ[Alur et al.]では、**バリューオブジェクト**という名称は**データ変換オブジェクト**の意味で使っているため、パターンコミュニティでは争いの原因になっている。この論争は、業界で常に起こる名前に関する争いの1つに過ぎない。最近[Alur et al.]は、上記の代わりに**変換オブジェクト**という名称を使うことに決めた。

私は、引き続き**バリューオブジェクト**を本文で使うことにする。少なくとも私の執筆内容が、私が以前執筆した内容と同じであれば何の問題もないはずだ。

18.7 | マネー

貨幣の値として機能する。



コンピュータはMoney（貨幣）を処理しているが、困ったことに主流のプログラミング言語はどれもMoneyを最重要データ型として扱っていない。通貨を扱う環境にとって、型がないことが問題の原因であることは明らかである。すべての計算が1つの通貨だけで行われる場合、このことはあまり問題にならないが、複数の通貨を扱う場合は、通貨の違いを無視してドルを円に足したりしないようにするはずである。最も難解なのは、通貨の丸めに関する問題である。金銭の計算では最小通貨単位までに数値が丸められる場合が多い。このため丸めエラーにより小さな金額が失われてしまいやすい。

オブジェクト指向プログラミングのメリットは、処理するMoneyクラスを作成して問題を修正できることである。意外なのは、主流のベースクラスライブラリにこれを処理する機能がまったくないことである。

18.7.1 | 動作方法

基本的な考え方は、数量と通貨のフィールドを持つ Money クラスを持つことである。数量を整数型または固定小数点型のいずれかで格納する。小数型が処理しやすい場合もあれば、整数型が処理しやすい場合もある。必ずどの種類であっても浮動小数点型は避ける必要がある。それを使うと Money で回避しようとしている丸めに関する問題を持ち込んでしまう。多くの場合、ドルのセントのように、最小の端数単位まで丸め処理された金銭の値が必要とされる。しかし、場合によっては小数単位も必要である。処理している Money の種類を明確にすることが重要である。特に両方の種類の金額を使用する場合は重要である。演算は2つのケースで異なる振る舞いを示すため、2つのケースに対しそれぞれの型を持つことが有効である。

Money は**バリューオブジェクト**のため、貨幣単位と数量に基づいて、等価性とハッシュコードの動作がオーバーライドされなければならない。

Money の場合、Money オブジェクトを数字のように簡単に使えるような、算術演算が必要である。しかし、Money の算術演算と数字の Money 演算との間には、重要な相違点がいくつかある。最も明確なのは、異なる貨幣単位の金銭を合算しようとする場合、加算や減算では常に貨幣単位を認識する必要があることである。シンプルでよく使われる対処方法は、異なる貨幣単位の合算をエラーとして処理することである。洗練された状況では、財布に関する Ward Cunningham の考え方を使うこともできる。複数の貨幣単位の Money をともに1つのオブジェクトに含めるというオブジェクトである。オブジェクトは、Money オブジェクトのように計算に加わり通貨の貨幣単位での評価もできる。

乗算と除算は、丸めの問題でさらに複雑になってしまう。スカラー量で、Money の乗算を行う。請求書に5%の税金を加算したい場合 0.05 を掛ける。このように、通常の数値型によって乗算の結果を得る。

特に、異なる場所に Money を配分する際、丸めのためにさらに複雑になってしまう。ここでは、Matt Foemmel のシンプルな問題を紹介する。Money の合計を2つの口座に配分するというビジネスルールがあるとする。つまり、1箇所には 70 %、そしてもう1箇所には 30 %である。ここで、5セントを配分するとする。計算すると、3.5 セントと 1.5 セントになる。どちらの方法で丸めを処理しても問題が起こる。通常の四捨五入で処理すると、1.5 は2に、3.5 は4になる。つまり1セント増加する。端数を切り捨てるなら4セントになり、切り上げるなら6セントになるのである。1セントの増加または減少を防止するため、この分配に適用できる丸めの汎用スキーマはないのである。

この問題には、いろいろな解決策がある。

- 私が最も行うのは、無視することである。たった1セントの違いじゃないか。もちろん、経理担当者の逆鱗には触れることになる。

- 分配するときは常に、分配済みの分から差し引いて最新の分配を行う。こうすることでセントの損失を防止できるが、最新の分配の際にセントが累積されていく可能性がある。
- Money クラスのユーザがメソッドを呼び出す時、丸めスキーマを宣言する。これによって、プログラマは 70 %の側を切り上げて、30 %の側は切り下げると宣言できる。これが 2 口座ではなく、10 口座すべてに分配する場合は複雑になる可能性がある。丸めの処理を忘れてはいけない。丸め処理を忘れないよう、Money クラスが丸めパラメータに強制的に乗算させているのを私は見たことがある。プログラマは必要な丸め処理について考えるだけではなく、書き込みのテストを行うことも忘れないようにする。しかし、すべて同じような方法で丸めを処理する税金計算がたくさんある場合、処理が難しくなる。
- 私の好む解決策は、金銭についてのアロケータ関数を持つことである。アロケータのパラメータは、分配すべき比率を表す数字のリストである (`aMoney.allocate([7,3])` など)。アロケータは Money リストを返し、外部からは擬似乱数に見える方法で、分配された Money にセントを分散させることによって、確実にセント単位で切り捨て分がないようにする。アロケータには、使い忘れてはいけないという欠点がある。忘れてしまうと、セントの割り当て場所についての正確なルールを実行することが難しくなってしまう。

ここでの基本的な問題は、(税金など) 比例して課金されるものを乗算によって決めることと、乗算によって複数の場所に金額を配分することの間にある。乗算は、前者に対しては正しく機能するが、後者に対してはアロケータの方が正しく機能する。重要なことは、金銭の値に対して乗算／除算を行う目的を考えることである。

`aMoney.convertTo(Currency.DOLLARS)` などのメソッドを使って、1 つの貨幣単位から別の貨幣単位へ変換することができる。変換のための明解な方法は、交換レートを掛けることである。たいていの場合はこの方法で十分だが、丸めのためにうまく機能しない場合もある。ユーロ圏の固定された通貨間の変換規則には独特の丸め処理があったが、これは単純な乗算では不十分だった。それゆえアルゴリズムをカプセル化する Convertor オブジェクトを持つほうが賢明である。

比較動作によって、金銭を分類できる。加算演算と同様に、比較するには貨幣単位を認識する必要がある。異なる貨幣単位を比較する場合、例外を発生させるか、変換を行うかのいずれかを選択する。

マネーは印刷の振る舞いをカプセル化できる。カプセル化により、ユーザインタフェースとレポートに良好な表示を行うことがとても容易になる。さらに、Money クラスは文字列を解析して貨幣単位認識入力メカニズムを提供するが、これはユーザインタフェースにとっ

てとても有用である。Money クラスは、プラットフォームライブラリが支援可能な場所である。特定の国に対して特定の数のフォーマッターを使うことによって、ますます多くのプラットフォームがグローバル化をサポートしている。

データベースもまた金銭の重要性を理解していないようなので（ベンダーは理解しているのだが）、データベースにマネーを格納すると、いつでも問題が発生する。取るべき方法は、**組込バリュー**を使って金銭ごとに貨幣単位を格納することである。たとえば、1つの口座に入っているのがすべてポンドの場合には、やりすぎになる場合がある。この場合、口座に貨幣を格納し、データベースのマッピングを修正し、口座の貨幣が引き出せるようにすればよい。

18.7.2 | 使用するタイミング

私はオブジェクト指向環境での大多数の数値計算に対してマネーを使う。理由は、丸めの振る舞いの処理をカプセル化するためである。これによって、丸めエラーの問題を減少させることができる。マネーを使う別の理由は、複数の貨幣単位での動作がとても簡単になるためである。マネーに対する反対意見はパフォーマンスの問題である。しかし私は、何らかの著しい違いが発生したという話をほとんど聞いたことがない。それどころかカプセル化チューニングが簡単になる。

18.7.3 | 例：Money クラス（Java）

(by Matt Foemmel, Martin Fowler)

最初の決定事項は、数量に使うデータ型である。浮動小数点数を推奨しないことを納得させる必要がある場合、以下のコードを実行しよう。

```
double val = 0.00;
for (int i = 0; i < 10; i++) val += 0.10;
System.out.println(val == 1.00);
```

浮動小数点では安全ではないので、固定小数点小数か整数のいずれかの選択になる。Java では、BigDecimal、BigInteger、long になる。整数値を使うと、内部の演算が容易になる。そして、long を使うと、基本形を使うことができ、それによって読み取り可能な計算式を持つことができる。

```
class Money...

   private long amount;
   private Currency currency;
```

私は整数量、つまり最少の基本単位を使っている。コード内ではcentと呼んでいるが、いい名前だと思う。longの場合、数値があまりにも大きいとオーバーフローエラーが発生する。92,233,720,368,547,758.09ドルを受け取る場合、BigIntegerを使うバージョンを記述する。種々の数値型からコンストラクタを提供できることは便利である。

```
public Money(double amount, Currency currency) {
    this.currency = currency;
    this.amount = Math.round(amount * centFactor());
}
public Money(long amount, Currency currency) {
    this.currency = currency;
    this.amount = amount * centFactor();
}
private static final int[] cents = new int[] { 1, 10, 100, 1000 };
private int centFactor() {
    return cents[currency.getDefaultFractionDigits()];
}
```

貨幣単位が異なると、小数量も異なる。Java1.4のCurrencyクラスは、クラスの小数の桁数を通知する。大きい方の単位の中に、10の累乗で、どのくらいの数の小さい方の単位を入れるかを決定できるが、Javaではとても面倒なので配列の方が簡単である（そのほうがおそらく処理速度が速い）。これは、4つの小数桁を使った際にコードが壊れることにうまく対処する準備をしているのである。

多くの場合、金銭処理を直接使いたいと思うだろうだが、基盤となるデータにアクセスすることが必要な場合もある。

```
class Money...

    public BigDecimal amount() {
        return BigDecimal.valueOf(amount, currency.getDefaultFractionDigits());
    }
    public Currency currency() {
        return currency;
    }
```

いかなる場合でも、アクセスを使うかどうかを考える必要がある。たいていはカプセル化を破壊しない方法を選ぶが、無視できない例として**組込バリュー**などのデータベースのマッピングが挙げられる。

リテラルな量に1つの貨幣単位を頻繁に使う場合、ヘルパーコンストラクタが役立つ。

```
class Money...

   public static Money dollars(double amount) {
      return new Money(amount, Currency.USD);
   }
```

Money はバリューオブジェクトなので、同等を定義する必要がある。

```
class Money...

   public boolean equals(Object other) {
      return (other instanceof Money) && equals((Money)other);
   }
   public boolean equals(Money other) {
      return currency.equals(other.currency) && (amount == other.amount);
   }
```

さらに、同等がある箇所は、必ずハッシュが必要である。

```
class Money...

   public int hashCode() {
      return (int) (amount ^ (amount >>> 32));
   }
```

加算と減算のある演算を行ってみよう。

```
class Money...

   public Money add(Money other) {
      assertSameCurrencyAs(other);
      return newMoney(amount + other.amount);
   }
   private void assertSameCurrencyAs(Money arg) {
      Assert.equals("money math mismatch", currency, arg.currency);
   }
   private Money newMoney(long amount) {
      Money money = new Money();
      money.currency = this.currency;
      money.amount = amount;
      return money;
   }
```

上記の private なファクトリーメソッドの使い方に注意してほしい。セントベースの量への通常の変換を行わない。これをマネーコードの内部で何度か使っている。

定義を追加することで、減算は簡単に行える。

```
class Money...

   public Money subtract(Money other) {
      assertSameCurrencyAs(other);
      return newMoney(amount - other.amount);
   }
```

比較のためのベースメソッドは、compareTo である。

```
class Money...

   public int compareTo(Object other) {
      return compareTo((Money)other);
   }
   public int compareTo(Money other) {
      assertSameCurrencyAs(other);
      if (amount < other.amount) return -1;
      else if (amount == other.amount) return 0;
      else return 1;
   }
```

現在、Java クラスで取得できるのはこれですべてであるが、次のような他の比較メソッドを使うと、コードがさらに読み取りやすくなることが分かる。

```
class Money...

   public boolean greaterThan(Money other) {
      return (compareTo(other) > 0);
   }
```

次に、乗算に着目することにする。ここではデフォルトの丸めモードを提供しているが、読者も自分自身で同様に設定することができる。

```
class Money...

   public Money multiply(double amount) {
      return multiply(new BigDecimal(amount));
   }
```

```
    public Money multiply(BigDecimal amount) {
        return multiply(amount, BigDecimal.ROUND_HALF_EVEN);
    }
    public Money multiply(BigDecimal amount, int roundingMode) {
        return new Money(amount().multiply(amount), currency, roundingMode);
    }
```

金額を多くのターゲットに分配し、しかもセントを損失したくない場合、allocation メソッドが必要である。最もシンプルなのは、多くのターゲットに対して（ほどんど）同じ額を分配することである。

```
class Money...

    public Money[] allocate(int n) {
        Money lowResult = newMoney(amount / n);
        Money highResult = newMoney(lowResult.amount + 1);
        Money[] results = new Money[n];
        int remainder = (int) amount % n;
        for (int i = 0; i < remainder; i++) results[i] = highResult;
        for (int i = remainder; i < n; i++) results[i] = lowResult;
        return results;
    }
```

より高度な分配アルゴリズムは、任意の比率で処理することができる。

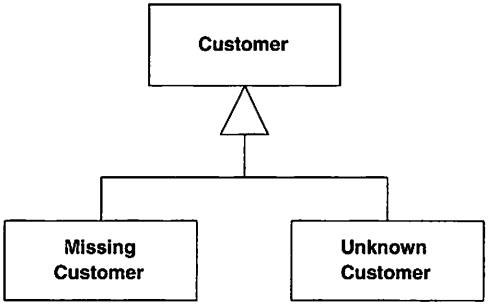
```
class Money...

    public Money[] allocate(long[] ratios) {
        long total = 0;
        for (int i = 0; i < ratios.length; i++) total += ratios[i];
        long remainder = amount;
        Money[] results = new Money[ratios.length];
        for (int i = 0; i < results.length; i++) {
            results[i] = newMoney(amount * ratios[i] / total);
            remainder -= results[i].amount;
        }
        for (int i = 0; i < remainder; i++) {
            results[i].amount++;
        }
        return results;
    }
```

これを使うと、Foemmel の謎を解決できる。

```
class Money...

   public void testAllocate2() {
      long[] allocation = {3,7};
      Money[] result = Money.dollars(0.05).allocate(allocation);
      assertEquals(Money.dollars(0.02), result[0]);
      assertEquals(Money.dollars(0.03), result[1]);
   }
```

18.8 | スペシャルケース

特定のケースで特別な振る舞いを提供するサブクラス。



オブジェクト指向プログラムでは、null を扱うのは難しい。ポリモフィズムを無効にするからだ。通常は、項目が正確な型かサブクラスのいずれかについて心配することなく、所定の型の変数参照で自由に foo を呼び出すことができる。強力な型付き言語を使うと、コンパイラに呼び出しが正しいことの確認もできる。しかし、変数には null が含まれるので、null のメッセージを呼び出すと実行時エラーになる場合があり、正確でフレンドリーなスタックトレース情報を取得しなければならない。

変数が null である場合、null テストコードで周りを取り囲む必要がある。null テストコードで周りを取り囲んでおくと、null があるときに適切に対処できる。しかし、多くのコンテキストで同じ場合が多いので、多くの場所に同様のコードを記述するということになり、コードの重複になる。

null は、上記のような問題の一般的な例であり、他の問題も頻繁に発生する。数値体系では、無限大を処理する必要がある。数値体系には、実数の通常の不変量を破壊する加算に対して特殊な規則がある。ビジネスソフトウェアでの最初の経験の 1 つは、あまり知られていない「オキュパント（occupant）」と呼ばれるユーティリティ Customer に関するもので

あった。タイプの普通の振る舞いを修正しなければならない。

null やその他の値を返すのではなく、呼び出し元が同じインタフェースを持つ**スペシャルケース**を返さなければならない。

18.8.1 | 動作方法

基本的な考え方は、**スペシャルケース**を処理するサブクラスを作成することである。つまり、Customer オブジェクトがあり、null チェックを回避したい場合、nullCustomer オブジェクトを作成する。顧客のすべてのメソッドを、無害な振る舞いを提供するスペシャルケースのメソッドでオーバーライドする。そして、null があるときは必ず、nullCustomer のインスタンスを使う。

nullCustomer のインスタンス間の違いを区別する理由はないので、フライウェイト[Gang of Four]で**スペシャルケース**を実装することができる。ただし、いつでもフライウェイトで実装できるとは限らない。ユーティリティの場合、あまり課金できないときもオキュパント Customer に対する料金を累積することができるので、オキュパントを別々に保持することは重要である。

null は違った意味を持つ場合もある。nullCustomer は Customer がいないこと、または Customer（顧客）があるがだれかは知らないことを意味することもある。この場合、単に nullCustomer を使うのではなく、Missing Customer（行方不明の顧客）や Unknown Customer（身元不明の顧客）として別々に**スペシャルケース**を持つことも考えなければならない。

スペシャルケースがメソッドをオーバーライドする一般的な方法は、別の**スペシャルケース**を返すことである。Unknown Customer に最新の請求書を要求すると、不明な請求書となる場合がある。

IEEE 754 浮動小数点演算は、正の無限大、負の無限大、および NaN(not-a-number)を備える**スペシャルケース**の優れた例である。0 で除算をする場合、処理が必要な例外を取得する代わりに、システムは NaN を返す。そして、NaN は他のいずれかの浮動小数点数と同じように扱われる。

18.8.2 | 使用するタイミング

特定のクラスのインスタンスに対する条件チェックの後か、null チェックの後に同じ振る舞いを持つ複数の箇所がシステム内にある場合は、いつでも**スペシャルケース**を使うべきである。

18.8.3 | 参考文献

パターンとして解説されたスペシャルケースを見たことはまだないが、null オブジェクトは[Woolf]で解説されている。駄洒落を許してもらえるなら、null オブジェクトはスペシャルケース(スペシャルケース)の特別な場合と言える。

18.8.4 | 例：シンプルな Null オブジェクト（C#）

ここでは、null オブジェクトとして使われるスペシャルケースのシンプルな例を紹介しよう。

まず、正規の従業員がいるとする。

```
class Employee...

      public virtual String Name {
         get {return _name;}
         set {_name = value;}
      }
      private String _name;
      public virtual Decimal GrossToDate {
         get {return calculateGrossFromPeriod(0);}
      }
      public virtual Contract Contract {
         get {return _contract;}
      }
      private Contract _contract;
```

クラスの機能は、nullEmployee でオーバーライドできる。

```
class NullEmployee : Employee, INull...

      public override String Name {
         get {return "Null Employee";}
         set {}
      }
      public override Decimal GrossToDate {
         get {return 0m;}
      }
      public override Contract Contract {
         get {return Contract.NULL;}
      }
```

nullEmployee に契約を要求すると、null 契約が返されることに注意してほしい。

デフォルト値が同じ null 値になる場合、多くの null テストを回避することになる。繰り

返される null 値は、デフォルトで null オブジェクトによって処理される。null でないことを明示的にテストするには、Customer に isNull メソッドを提供するか、またはマーカーインタフェースのタイプテストを使うことで行える。

18.9 | プラグイン

(by David Rice, Matt Foemmel)

コンパイル時ではなくシステム構成の際にクラスをリンクする。

セパレートインタフェースは、アプリケーションコードが複数の実行時環境で実行される場合に使われ、それぞれの環境では特定の振る舞いを異なる方法で実装することが必要である。ほとんどの開発者は、ファクトリーメソッドを記述することで、適切な実装を提供している。ユニットテストにはシンプルなメモリ内カウンターを使うが、本番時の稼動には、データベース管理のシーケンスを使うように、**セパレートインタフェース**を備える主キー生成プログラムを定義する。ファクトリーメソッドにはある条件文が含まれている。ローカル環境変数を調べてシステムがテストモードかどうかを判定し、適切なキー生成プログラムを返す条件文である。いったん数個のファクトリーを持つと管理に混乱が発生する。新しい配備システム構成、つまり、「トランザクション制御機能なしのメモリ内データベースに対してユニットテストを実行する」または「トランザクション全制御機能のある DB2 データベースに対して本番稼動モードで実行する」では、システム構成を設定するのにいくつかのファクトリーでの条件文の編集、再構築、および再配備が必要とされる。システム構成のアプリケーション全体に対する分散、再構築や再配備は必要ではない。**プラグイン**は、一元管理された実行時構成を提供して両方の問題を解決する。

18.9.1 | 動作方法

最初に行うことは、実行時環境に基づいた異なる実装の振る舞いを持つセパレートインタフェースを定義することである。さらに、特別な要件を数個だけ加えた基本的なファクトリーパターンを使用する。**プラグイン**ファクトリーでは、システム構成を容易に管理できるようにするために、1 つの外部ポイントで示されるリンク命令が必要である。さらに、実装へのリンクは、コンパイル時ではなく実行時に動的に行われなければならないので、再構築

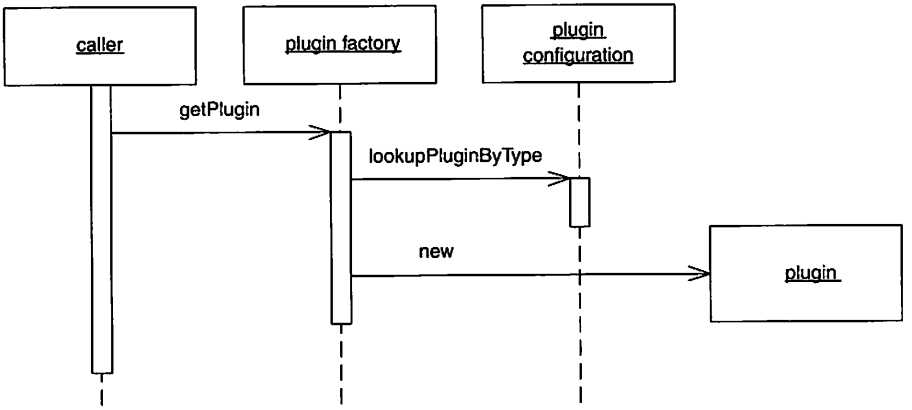


図 18.2——呼び出し元は、分離されたインタフェースのプラグイン実装を取得する。

テキストファイルは、リンク規則の表明手段としてとても良好に機能する。**プラグインファクトリー**は、単にテキストファイルを読み取り、要求されたインタフェースの実装を指定して返す。

ファクトリーはコンパイル時に実装へ依存しないで構築できるので、**プラグイン**はリフレクションをサポートする言語において見事に機能する。リフレクションを使う際、システム構成ファイルには実装クラス名へのインタフェース名のマッピングが含まれている。ファクトリーは、フレームワークパッケージから独立して配置され、新しい実装をシステム構成の選択肢に追加する際にも変更する必要がない。

リフレクションをサポートしない言語を使う場合でも、システム構成で中央ポイントを設定するだけの価値がある。リンク規則を設定する場合にもテキストファイルを使うが、その違いは、ファクトリーがインタフェースを希望する実装にマッピングするために条件分岐ロジックを使うことだけである。それぞれの実装の型は、ファクトリーで説明される必要があるが、実際にはそれほどの量ではない。新しい実装をコードベースに追加する場合はいつでも、同時にファクトリーメソッド内に別のオプションを追加する。ビルド時チェックでレイヤとパッケージの依存性を持たせるために、ファクトリーを自らのパッケージに配置し、ビルドプロセスの中断を避けなければならない。

18.9.2 | 使用するタイミング

実行時環境に基づいた実装が必要な振る舞いを持つ場合は、常に**プラグイン**を使う。

18.9.3 | 例：ID 生成プログラム（Java）

上述のとおり、キーまたは ID の生成では、その実装が配備環境によって変化する（図 18.3）。

最初に、IDGenerator セパレートインタフェースや必要な実装を記述する。

```
interface IDGenerator...

   public Long nextId();

class OracleIDGenerator implements IDGenerator...

   public OracleIDGenerator(){
      this.sequence = Environment.getProperty("id.sequence");
      this.datasource = Environment.getProperty("id.source");
   }
```

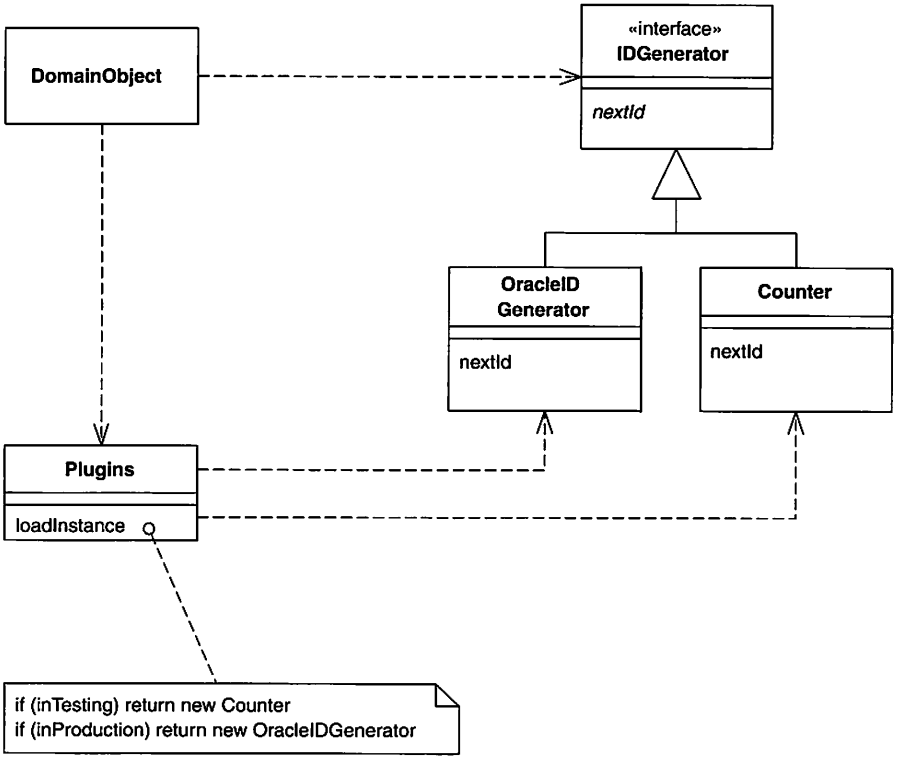


図 18.3 ——複数の ID 生成プログラム

OracleIDGeneratorの場合nextId()は、次に利用可能な番号を定義済みデータソースの定義済みのシーケンスから選択する。

```
class Counter implements IDGenerator...

   private long count = 0;
   public synchronized Long nextId() {
      return new Long(count++);
   }
```

構築すべきものがある場合、現在のインタフェースから実装へのマッピングを実現するプラグインファクトリーを記述してみよう。

```
class PluginFactory...

   private static Properties props = new Properties();

   static {
      try {
         String propsFile = System.getProperty("plugins");
         props.load(new FileInputStream(propsFile));
      } catch (Exception ex) {
         throw new ExceptionInInitializerError(ex);
      }
   }
   public static Object getPlugin(Class iface) {

      String implName = props.getProperty(iface.getName());
      if (implName == null) {
         throw new RuntimeException("implementation not specified for
               " + iface.getName() + " in PluginFactory propeties.");
      }
      try {
         return Class.forName(implName).newInstance();
      } catch (Exception ex) {
         throw new RuntimeException("factory unable to construct
               instance of " + iface.getName());
      }
   }
```

リンク命令を含むファイルが配置され、プラグインという名前を持つシステムプロパティを検索することによって、システム構成をロードできることに注意してほしい。リンク命令の定義や格納には多くの選択肢があるが、シンプルなプロパティファイルが最も簡単である。

クラスパスを見るのではなく、ファイルを見つけるシステムプロパティを使うことによって、マシン上のどこに配置しても新しいシステム構成の指定が簡単になる。開発、テスト、本番時の稼動の各環境間でビルドを変更する場合、上記のことはとても便利である。次のように、1つがテスト用で、もう1つが本番稼動用の2つの異なるシステム構成ファイルがどのようなものかを紹介する。

```
config file test.properties...

   # test configuration
   IDGenerator=TestIDGenerator

config file prod.properties...

   # production configuration
   IDGenerator=OracleIDGenerator
```

IDGeneratorインタフェースに戻り、**プラグイン**ファクトリーへの呼び出しで設定される静的なINSTANCEメンバを追加する。INSTANCEメンバは、**プラグイン**とシングルトンパターンを組み合わせて、IDを取得するための簡単で読み取りできる呼び出しを提供する。

```
interface IDGenerator...

   public static final IDGenerator INSTANCE =
      (IDGenerator) PluginFactory.getPlugin(IDGenerator.class);
```

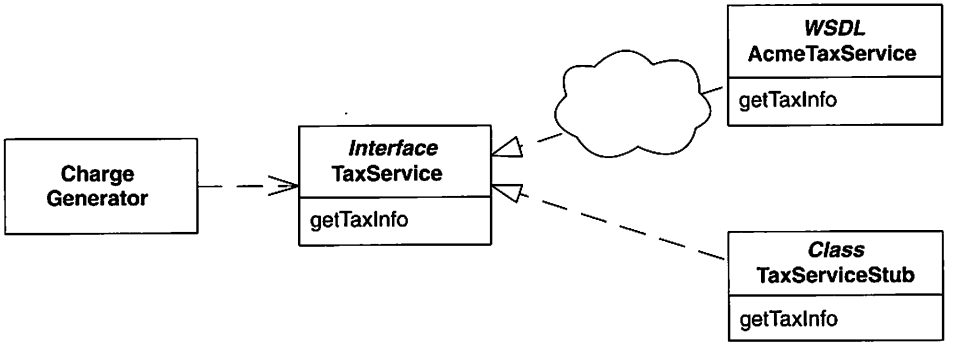
これで、適切な環境に対して適切なIDを取得するということを認識している呼び出しを作成できる。

```
class Customer extends DomainObject...

   private Customer(String name, Long id) {
      super(id);
      this.name = name;
   }
   public Customer create(String name) {
      Long newObjId = IDGenerator.INSTANCE.nextId();
      Customer obj = new Customer(name, newObjId);
      obj.markNew();
      return obj;
   }
```

18.10 | サービススタブ

(by David Rice)

テスト中に問題となるサービスへの依存性を除去する。



エンタープライズシステムは、貸方の記入、税率の調査、および価格決定などサードパーティのサービスコンポーネントへのアクセスに依存している。上記のようなシステムを構築している開発者はすべて、自分が完全に制御できないリソースに依存していることから来るフラストレーションを口にする。機能の配信は予測できず、機能がリモート信頼性の場合もあり、パフォーマンスも悪化する可能性がある。

少なくとも、このような問題があると開発プロセスは遅れてしまう。開発者は、サービスがオンラインで戻ってくるのをぼんやりと座って待っているか、または、おそらく間に合わせのプログラムをコードに挿入し、まだ配信されない機能の代わりとするかもしれない。さらに発生する可能性が大きい深刻な問題は、そうした依存性により何回もテストが実行不可能になることである。テストが実行不可能になると開発プロセスは中断してしまうからである。

テストの際、サービスをローカルで高速に実行してメモリで実行する**サービススタブ**に置き換えると、開発作業を改善することができる。

18.10.1 | 動作方法

最初のステップは、**ゲートウェイ**を使ってサービスへのアクセスを定義することである。**ゲートウェイ**は、クラスではなく**セパレートインタフェース**なので、サービスを呼び出す1つの実装と少なくとも1つの単なる**サービススタブ**を持つ。**ゲートウェイ**の望ましい実装は、**プラグイン**を使ってロードされる必要がある。**サービススタブ**を記述する際に重要な点は、できるだけシンプルにすることである。複雑にすると目的を果たすことができない。

ここでは、売上税サービスをスタブする流れを行ってみることにする。このサービスは、

住所、製品タイプ、および売上金額が与えられている州の売上税額と税率を提供する。**サービススタブ**を提供する最もシンプルな方法は、リクエストに適合する一定の税率を使った数行のコードを記述することである。

もちろん、税金の法律はシンプルではない。製品によっては、特定の州で非課税な場合がある。そのため、製品と州をどのように組み合わせると非課税になるかについては実際の税金サービスに委ねている。しかし、アプリケーションの多くの機能は税金が課されるかどうかに依存しているので、**サービススタブ**で非課税機能に対応する必要がある。この振る舞いをスタブに追加する最もシンプルな方法は、住所と製品の特定の組み合わせを除外する条件文を通して、該当するいずれかのテストケースで同じデータを使うことである。スタブのコードの行数は片手でカウントできる。

より動的な**サービススタブ**では、非課税製品と州の組み合わせリストを管理し、テストケースリストを追加することができる。この場合でさえ、10 行程度のコードである。開発プロセスの時間の短縮が目的なので、コードはシンプルである。

動的な**サービススタブ**によって、**サービススタブ**とテストケース間の依存性に関して興味深い疑問が提起される。**サービススタブ**は、本来の税金サービス**ゲートウェイ**インタフェースにはない非課税の対象を追加する setup メソッドに依存している。**サービススタブ**のロードに**プラグイン**を利用するために、この setup メソッドが**ゲートウェイ**に追加される。setup メソッドは、あまり余計なものをコードに追加することなく、テストの名前で行われるので差し支えない。サービスを呼び出す**ゲートウェイ**の実装が、いずれかのテストメソッドの中でアサーションの失敗を確実に起こすようにしなければならない。

18.10.2 | 使用するタイミング

特定のサービスへの依存性が開発とテストを妨げていることがわかった場合は、**サービススタブ**を使うべきである。

エクストリームプログラミング（Extreme Programming）に従事する多くのユーザは、**サービススタブ**の代わりに**モックオブジェクト**という用語を使っているが、**サービススタブ**の方が長い間使われているので、ここでは**サービススタブ**を使うことにする。

18.10.3 | 例：売上税サービス（Java）

アプリケーションは、Web サービスとして配備された税金サービスを使う。考慮したい最初の項目は、ドメインコードが Web サービスの未知な点を処理することがないように**ゲートウェイ**を定義することである。**ゲートウェイ**は、記述した任意の**サービススタブ**のロードに役立つインタフェースとして定義される。税金サービスの実装のロードには**プラグ**

インを使う。

```
interface TaxService...

    public static final TaxService INSTANCE =
        (TaxService) PluginFactory.getPlugin(TaxService.class);
    public TaxInfo getSalesTaxInfo(String productCode, Address addr,
            Money saleAmount);
```

シンプルな一定税率のサービススタブは、次のようになる。

```
class FlatRateTaxService implements TaxService...

    private static final BigDecimal FLAT_RATE =
              new BigDecimal("0.0500");
    public TaxInfo getSalesTaxInfo(String productCode, Address addr,
              Money saleAmount) {
        return new TaxInfo(FLAT_RATE, saleAmount.multiply(FLAT_RATE));
    }
```

特定の住所と製品の組み合わせを非課税扱いにするサービススタブは、次のとおりである。

```
class ExemptProductTaxService implements TaxService...

    private static final BigDecimal EXEMPT_RATE = new BigDecimal("0.0000");
    private static final BigDecimal FLAT_RATE = new BigDecimal("0.0500");
    private static final String EXEMPT_STATE = "IL";
    private static final String EXEMPT_PRODUCT = "12300";
    public TaxInfo getSalesTaxInfo(String productCode, Address addr,
            Money saleAmount) {
        if (productCode.equals(EXEMPT_PRODUCT) &&
                addr.getStateCode().equals(EXEMPT_STATE)) {
            return new TaxInfo(EXEMPT_RATE, saleAmount.multiply(EXEMPT_RATE));
        } else {
            return new TaxInfo(FLAT_RATE, saleAmount.multiply(FLAT_RATE));
        }
    }
```

次に、ここでは、テストケースを追加して非課税の組み合わせをリセットできるメソッドを備えた動的なサービススタブを紹介する。追加のテストメソッドが必要な場合は、逆戻りして、よりシンプルなサービススタブや実際の税金 Web サービスを呼び出す実装にもメ

ソッドを追加する必要がある。不要なテストメソッドは、アサーションの失敗を起こさなければならない。

```
class TestTaxService implements TaxService...

    private static Set exemptions = new HashSet();
    public TaxInfo getSalesTaxInfo(String productCode, Address addr,
              Money saleAmount) {
        BigDecimal rate = getRate(productCode, addr);
        return new TaxInfo(rate, saleAmount.multiply(rate));
    }
    public static void addExemption(String productCode, String stateCode) {
        exemptions.add(getExemptionKey(productCode, stateCode));
    }
    public static void reset() {
        exemptions.clear();
    }
    private static BigDecimal getRate(String productCode, Address addr) {
        if (exemptions.contains(getExemptionKey(productCode,
                  addr.getStateCode()))) {
            return EXEMPT_RATE;
        } else {
            return FLAT_RATE;
        }
    }
```

税金のデータを提供するWebサービスを呼び出す実装は示されず、本番稼動の**プラグインシステム構成**により税金サービスインタフェースにリンクする。テスト**プラグインシステム構成**は、上記の**サービススタブ**にリンクする。

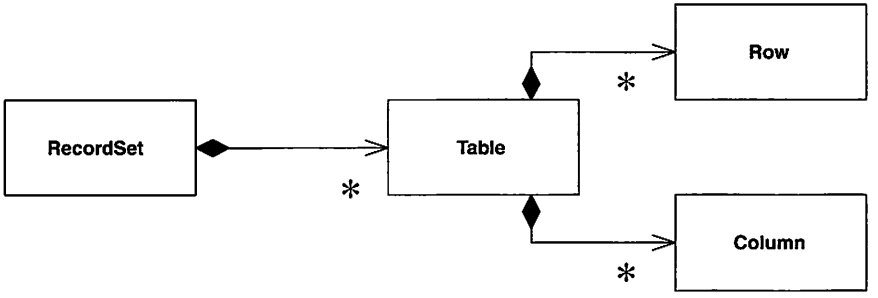
最後に、税金サービスのあらゆる呼び出し元は、**ゲートウェイ**を介してサービスにアクセスする。以下の料金生成プログラムは、標準料金を生成し、対応する税金をすべて生成する税金サービスを呼び出す。

```
class ChargeGenerator...

    public Charge[] calculateCharges(BillingSchedule schedule) {
        List charges = new ArrayList();
        Charge baseCharge = new Charge(schedule.getBillingAmount(), false);
        charges.add(baseCharge);
        TaxInfo info = TaxService.INSTANCE.getSalesTaxInfo(
              schedule.getProduct(), schedule.getAddress(),
```

```
            schedule.getBillingAmount());
        if (info.getStateRate().compareTo(new BigDecimal(0)) > 0) {
            Charge taxCharge = new Charge(info.getStateAmount(), true);
            charges.add(taxCharge);
        }
        return (Charge[]) charges.toArray(new Charge[charges.size()]);
    }
```

18.11 レコードセット

テーブルデータのメモリ上の表現。



ここ20年間、データベースのデータを表現する方法は、テーブルのリレーショナル形式が主流である。大小のデータベース企業と標準的なクエリー言語の後押しを受けて、私が知っている多くの新しい開発にはリレーショナルデータが使われている。

これに加えて、UIを短時間で構築するための豊富なツールがある。ツールのデータ認識UIフレームワークは、基盤となるデータがリレーショナルであることに依存し、データをプログラミングすることなく、データの表示や処理をしやすくするさまざまな種類のUIウィジェットを提供している。

上記の環境の欠点は、表示やシンプルな更新は極めて容易だが、ビジネスロジックを配置する実際の機能がないことである。「有効な日付」を過ぎた妥当性確認も、どんなビジネスルールや計算も行き場所がない。ビジネスロジックを配置する機能は、ストアドプロシージャとしてデータベースに押し込まれるか、UIコードに混入されるかのいずれかである。

レコードセットの考え方は、SQLクエリーの結果と厳密に類似しているが、システムの他の部分から生成および処理を行うことができるメモリ内構造を提供するというものである。

18.11.1 | 動作方法

レコードセットは、作業中のソフトウェアプラットフォームのベンダーから提供され、自ら構築することのない種類のものである。一例として、ADO.NET のデータセットとJDBC 2.0 の行セットが挙げられる。

レコードセットの最初の基本的要素は、厳密にデータベースクエリーの結果と同じに見えることである。つまり、クエリーを発行しデータを直接データ認識 UI に投げる 2 ティア手法を、2 ティアツールを使うことで簡単に利用できる。2 番目の基本的な要素は、容易に自ら**レコードセット**を構築すること、およびデータベースクエリーの結果のレコードセットを取得し、ドメインロジックコードで容易に処理できることである。

プラットフォームに**レコードセット**が用意されているが、自分で作成することもできる。問題は、データ認識 UI ツールがない場合、自分で作成しなければならなくなるという点にある。いずれにしても**レコードセット**構造をマッピングのリストとして構築することは、動的型付きスクリプト言語では一般的であり、このパターンの優れた例であると言える。

データソースへのリンクから**レコードセット**を切り離す機能は価値がある。これによってデータベースへの接続を意識することなく、ネットワーク一帯に**レコードセット**を渡すことができる。さらに、簡単に**レコードセット**を直列化することができる場合、アプリケーションに対して**データ変換オブジェクト**としての機能を果たすこともできる。

接続を解除すると、**レコードセット**を更新した場合に何が発生するかという問題が起きる。ますます多くのプラットフォームで**レコードセット**を**ユニットオブワーク**の形式にすることができるので、**レコードセット**を修正して、データソースに返し、コミットすることができる。通常、データソースでは**軽オフラインロック**を使って競合の有無を確認し、競合がない場合、変更をデータベースに書き込む。

18.11.1.1 ■ 明示的なインタフェース

レコードセットの実装では、**暗黙的なインタフェース**が使われる。つまり、**レコードセット**から情報を取得するためには、必要なフィールドを示す引数を持つ汎用メソッドを呼び出す。たとえば、航空会社の予約の乗客を取得するには `aReservation["passenger"]` などの式を使う。明示的なインタフェースでは、メソッドとプロパティが定義済みの Reservation クラスが必要となる。明示的な予約では、乗客を求める式は `aReservation.passenger` となる。

暗黙的なインタフェースは、任意の種類のデータに対して汎用の**レコードセット**を使うので柔軟性がある。新しい種類の**レコードセット**を定義するたびに新しいクラスを記述する必要がなくなる。しかし、概して私は、暗黙的なインタフェースが悪いものであることに気づいた。予約についてのプログラミングをする場合、どのようにして客を取得する方法がわかるのだろうか。文字列は、「passenger」、「guest」、「flyer」のいずれであるのか。唯一の方

法は、コードベースを探し回り、予約が作成されて使われている場所を見つけるように試みることである。明示的なインタフェースがある場合、予約の定義を見て必要なプロパティを確認することができる。

この問題は、静的型付き言語ではさらに悪化する。乗客の名字を知りたい場合、((Person) aReservation["passenger"]).lastName などの複雑な式を使わざるを得ないが、コンパイラはすべての型情報を失うので、手作業で入力し、必要な情報を取得する必要がある。明示的なインタフェースでは、型情報を保持することができるので、aReservation.passenger.lastName を使うことができる。

このため、私は暗黙的なインタフェースには不満である（ディクショナリで時間を余計に費やすインタフェースに対しても同様である）。さらに**レコードセット**を伴う暗黙的なインタフェースもあまり評価していない。しかし、ここでの唯一の救いは、通常、**レコードセット**が中にある規定の列についての情報を保持することである。さらに、列の名前は**レコードセット**を作成する SQL で定義されているので、必要な場合にプロパティを見つけることはそれほど難しくない。

しかし、もう一言述べると、明示的なインタフェースを持つ方が優れていると言える。ADO.NET では、**レコードセット**が強力な型付きデータセット、**レコードセット**用の明示的で完全な型付きインタフェースを提供する生成されたクラスとともに提供されている。ADO.NET データセットには、多くのテーブルとテーブル間の関係を含めるので、強力な型付きデータセットもその関係情報を使うことができるプロパティを提供している。クラスは、XSD のデータセットの定義から生成される。

暗黙的なインタフェースの方がよく使われているので、本書の例では型なしデータセットを使っている。しかし、ADO.NET の本番稼動のコードでは、型付きのデータセットを使うことを提案する。ADO.NET 以外の環境の場合は、自らの明示的な**レコードセット**のコード生成を提案する。

18.11.2 | 使用するタイミング

私の考えでは、**レコードセット**の価値は、データ処理の共通の方法として**レコードセット**に依存する環境を得ることにある。**レコードセット**は多くの UI ツールで使われている。UI ツールを使う場合、**テーブルモジュール**を使用してドメインロジックを構築すべきである。データベースから**レコードセット**を取得し、引き出した情報を計算するために**テーブルモジュール**に渡し、結果を UI に渡して表示と編集を行う。そして、**テーブルモジュール**に戻して妥当性確認を行い、データベースに更新をコミットする。

リレーショナルデータベースと SQL はこれまでにも存在していたが、何らかの代替構造とクエリー言語がなかったために、いろいろな意味で**レコードセット**をとても価値あるもの

にするツールが登場した。現在は、もちろん広く標準化された構造とXPathでのクエリー言語を持つXMLがある。しかし私は、現行のツールが現在レコードセットを使っているのと同じ方法で階層的な構造ツールが登場するようになるのではないかと考えている。おそらく、これは一般的なパターンの特別なケース、つまり、汎用データ構造のようなものである。しかし、そのパターンについてはその時期が来てから考えることにする。

参考文献

[Alexander et al.]
Alexander, et al. A Pattern Language . Oxford, 1977.（『パタン・ランゲージ―環境設計の手引』、平田翰那訳、鹿島出版会、1984）
パターンに携わる人々のためのアイディア集。私はそれほど熱中しなかったが、いろいろな手法を理解する上で一見の価値がある。

[Alpert et al.]
Alpert, Brown and Woolf. Design Patterns Smalltalk Companion . Addison- Wesley, 1998.
Smalltalkコミュニティ以外にはあまり知られていないが、従来のパターンの多くを詳細に説明している。

[Alur et al.]
Alur, Crupi, and Malks. Core J2EE Patterns : Best Practices and Design Strategies. Prentice Hall, 2001.（『J2EEパターン―明暗を分ける設計の戦略』、中野明彦ほか訳、ウルシステムズ株式会社監訳、ピアソン・エデュケーション、2002）
形式に新しい生命を吹き込むパターン書籍のニューウェイブの一つ。パターンはJ2EEプラットフォーム向けに表現されているが、他のプラットフォームにも活用できる。

[Ambler]
http://www.ambysoft.com/mappingObjects.html
オブジェクトリレーショナルマッピングに関する考えに役立つソース。

[Beck XP 2000]
Beck, Extreme Programming Explained . Addison-Wesley, 2000.（『XPエクストリーム・プログラミング入門―ソフトウェア開発の究極の手法』、永田渉・飯塚麻理香訳、長瀬嘉秀監訳、ピアソン・エデュケーション、2000）
XP（エクストリームプログラミング）のマニフェスト。ソフトウェアプロセスに関心がある人には必読の書。

[Beck Patterns]
Beck. Smalltalk Best Practice Patterns . Prentice Hall, 1997.（『ケント・ベックのSmalltalkベストプラクティス・パターン―シンプル・デザインへの宝石集』、梅澤真史ほか訳、ピアソン・エデュケーション、2003）
Smalltalkベースのため残念ながらあまり読まれていないが，OO言語向けに書かれたたいていの書籍より優れたアドバイスが収められている。

[Beck TDD]
Beck. Test-Driven Development: By Example. Addison-Wesley, 2003.（『テスト駆動開発入門』、テクノロジックアート訳、長瀬嘉秀監訳、ピアソン・エデュケーション、2003）
本書と同時期に出版された。設計を発展させることができるテストとリファクタリングの緊密なサイクルへのBeck氏の指針である。

[Bernstein and Newcomer]
Bernstein and Newcomer. Principles of Transaction Processing . Morgan Kaufmann, 1997.（『トランザクション処理システム入門』、大磯和広ほか訳、日経BP社、1998）
頭を痛めるトランザクションの世界のすばらしい入門書。

[Brown et al.]
Brown et al. Enterprise Java Programming with IBM Websphere . Addison-Wesley, 2001.（『IBM WebSphereエンタープライズJavaプログラミング』、落合修監訳、出版：エスアイビー・アクセス、発売：星雲社、2002）
本書の2/3はソフトウェアのマニュアルであるが、その他の1/3には、このテーマに特化したほとんどすべての本より、優れた設計上のアドバイスが凝縮されている。

[Brown and Whitenack]
http://members.aol.com/kgb1001001/Chasms.htm
オブジェクトリレーショナルマッピングに関する最初期の、かつ最も優れたレポートの1つ。

[Cockburn UC]
Cockburn. Writing Effective Use Cases. Addison-Wesley, 2001.（『ユースケース実践ガイド―効果的なユースケースの書き方』、山岸耕二ほか訳、ウルシステムズ株式会社監訳、翔泳社、2001）
ユースケースに関する間違いなく最高の参考文献。

[Cockburn PloP]
Cockburn, "Prioritizing Forces in Software Design," in [PLoPD 2].
アプリケーションの境界についての解説。

[Coleman et al.]
Coleman, Arnold, and Bodoff. Object-Oriented Development: The Fusion Method, Second Edition. Prentice Hall, 2001.
UML以前の部分に対しては歴史的な関心しかそそられないが、インタフェースモデルについての解説は、サービスレイヤの設計者にとって非常に役立つものである。

[Evans and Fowler]
http://martinfowler.com/apsupp/spec.pdf
Specificationパターンの解説。

[Evans]
Evans.Domain Driven Design. Addison Wesley, 2003.
ドメインモデルの開発に関する書籍。エンタープライズアプリケーション開発の重要で困難な側面についての興味深い解説。

[Fowler Temporal Patterns]
http://martinfowler.com/ap2/timeNarrative.html
時間とともに変わるオブジェクトの履歴を処理するパターン。

[Fowler AP]
Fowler.Analysis Patterns. Addison-Wesley, 1997.（『アナリシスパターン―再利用可能なオブジェクトモデル』、児玉公信・友野晶夫・大脇文雄訳、堀内一監訳、アジソン・ウェズレイ・パブリッシャーズ・ジャパン、1998）
ドメインモデルのパターン。

[Fowler Refactoring]
Fowler,Refactoring. Addison-Wesley, 1999.（『リファクタリング―プログラミングの体質改善テクニック』、児玉公信ほか訳、ピアソン・エデュケーション、2000）
既存のコードベースの設計を改善するための技術。

[Fowler CI]
http://martinfowler.com/articles/continuousIntegration.html
1日に数回ソフトウェアを自動的にビルドする方法を説明するエッセイ。

[Gang of Four]
Gamma, Helm, Johnson, and Vlissides.Design Patterns.Addison-Wesley, 1995.（『オブジェクト指向における再利用のためのデザインパターン』、本位田真一・吉田和樹監訳、ソフトバンクパブリッシング、1999）
パターンに大きな影響を与えた書籍。

[Hay]
Hay.Data Model Patterns. Dorset House, 1995.
リレーショナルの観点からの概念モデルのパターン。

[Jacobson et al.]
Jacobson et al.Object-Oriented Software Engineering. Addison-Wesley, 1992.（『オブジェクト指向ソフトウェア工学OOSE―Use-caseによるアプローチ』、西岡利博ほか監訳、トッパン、1995）
OO設計に関する初期の書籍で、ユースケースと設計に対するインタフェース－コントローラーエンティティ手法を紹介している。

[Keller and Coldewey]
http://www.objectarchitects.de/ObjectArchitects/orpatterns/index.htm
オブジェクトリレーショナルマッピングについてのすばらしいソース。

[Kirtland]
Kirtland.Designing Component-Based Applications. Microsoft Press, 1998.
DNAアーキテクチャの説明。

[Knight and Dai]
Knight and Dai. "Objects and the Web."IEEE Software, March/April 2002.
Webアプリケーションにおけるモデルビューコントローラとその発展および使用に関するすばらしいレポート。

[Larman]
Larman.Applying UML and Patterns, Second Edition.Prentice Hall, 2001.（『実践UML―パターンによる統一プロセスガイド』、依田光江訳、今野睦・依田智夫監訳、ピアソン・エデュケーション、2003）
私が選ぶOO設計の入門書の筆頭。

[Lea]
Lea.Concurrent Programming in Java, Second Edition.Addison-Wesley, 2000.（「Javaスレッドプログラミング―並列オブジェクト指向プログラミングの設計原理」、松野良蔵監訳、翔泳社、2000）
複数のスレッドでプログラムしたい場合、まず本書を最初に理解する必要がある。

[Marinescu]
Marinescu.EJB Design Patterns. New York: John Wiley, 2002.（「EJBデザインパターン」、トップスタジオ訳、出版：日経BP社、発売：日経BP出版センター、2003）
JavaのEJBについての最近のパターンに関する書籍。

[Martin and Odell]
Martin and Odell.Object Oriented Methods: A Foundation (UML Edition).Prentice Hall, 1998.
概念的観点からのオブジェクトモデリングのほか、モデリングとは何かに関する基礎研究。

[Nilsson]
Nilsson. .NET Enterprise Design with Visual Basic .NET and SQL Server 2000.Sams, 2002.
Microsoftプラットフォームのアーキテクチャについての内容が充実した書籍。

[Peckish]
200万（79ページを参照）

[PLoPD 2]
Vlissides, Coplien, and Kerth (eds.).Pattern Languages of Program Design 2.Addison-Wesley, 1996.
さまざまなパターンレポートの選集。

[PLoPD 3]
Martin, Buschmann, and Rielhe (eds.).Pattern Languages of Program Design 3.Addison-Wesley, 1998.
さまざまなパターンレポートの選集。

[POSA]
Buschmann et al.Pattern-Oriented Software Architecture. Wiley, 2000.（「ソフトウェアアーキテクチャ―ソフトウェア開発のためのパターン体系」、金澤典子ほか訳、近代科学社、2000）
広範囲のアーキテクチャパターンに関する最良の書籍。

[Riehle et al.]
Riehle, Siberski, Baumer, Megert, and Zullighoven. "Serializer," in [PLoPD 3].
オブジェクト構造の直列化について掘り下げて説明しているもので、異なる形式への直列化を要する場合には必要な書。

[Schmidt]
Schmidt, Stal, Rohnert, and Buschmann.Pattern-Oriented Software Architecture,Volume 2. New York: John Wiley, 2000.
並列・分散システムのついてのパターン。アプリケーションサーバーを使用する人より、アプリケーションサーバーを設計する人向け。ただし、設計されたものを使用する場合にも、その設計思想についてある程度の知識を持っていると有益である。

[Snodgrass]
nodgrass.Developing Time-Oriented Database Applications in SQL. Morgan-Kaufmann, 1999.
リレーショナルデータベースで履歴情報を追跡する方法。

[Struts]
http://jakarta.apache.org/struts/
ますます人気が高まっているJavaのWebプレゼンテーションフレームワーク。

[Waldo et al.]
Waldo, Wyant, Wollrath, and Kendall.A Note on Distributed Computing. SMLITR-94-29,http://research.sun.com/technical-reports/1994/smli_tr-94-29.pdf,SunMicrosystems, 1994.
なぜ「透過的な分散オブジェクト」に危険な矛盾が含まれているのかに関する古典的なレポート。

[wiki]
http://c2.com/cgi/wiki
Ward Cunninghamによって開発された当初のWiki Web。まとまりには欠けるが魅力的でオープンなWebサイトで、あらゆる人々があらゆる種類の考え方を共有している。

[Woolf]
Woolf. "Null Object," in [PLoPD 3].
ヌルオブジェクトパターンの説明。

[Yoder]
http://www.joeyoder.com/Research/objectmappings
オブジェクトリレーショナルパターンの優れたソース。

記号・数字

A

B

C

D

E

F

G

X

あ

い

う

え

お

か

き

く

け

こ

さ

し

と

に

は

め

も

ゆ

よ

り

る

れ

ろ

わ

■ 著者紹介

Martin Fowler（マーチン・ファウラー）

エンタープライズアプリケーションの開発元であるThoughtWorks社のチーフサイエンティストをつとめる。10年以上にわたりオブジェクト指向テクノロジーをエンタープライズ系のソフトウェア開発に適用してきた。パターン、UML、リファクタリング、およびアジャイル手法の分野の第一人者である。マサチューセッツ州メルローズ市在住。ホームページはhttp://www.martinfowler.com

【著書】

Analysis Patterns: Reusable Object Modeling, 1997
『アナリシスパターン』（ピアソン・エデュケーション、2002）
Refactoring: Improving the Design of Existing Code, 1999
『リファクタリング』（ピアソン・エデュケーション、2000）
UML Distilled, 2nd Edition, 2000
『UMLモデリングのエッセンス第2版』（翔泳社、2000）

■ 監訳者紹介

長瀬 嘉秀（ながせ よしひで）

1986年東京理科大学理学部応用数学科卒業。朝日新聞を経て、1989年テクノロジックアートを設立。OSF（Open Software Foundation）のテクニカルコンサルタントとしてDCE関連のオープンシステムの推進を行う。OSF日本ベンダ協議会DCE技術検討委員会の主査をつとめる。現在、株式会社テクノロジックアート代表取締役。
UMLによるオブジェクト指向セミナーの講師、UML関連のコンサルティングを行っている。UML Profile for EDOCの共同提案者、ISO/IECJTC1 SC32/WG2委員、情報処理相互運用技術協会（INTAP）オープン分散処理委員、電子商取引推進協議会（ECOM）XML/EDI標準化調査委員。明星大学情報学部講師。

【著書】

『分散コンピューティング環境DCE』（共著、共立出版、1997）
『DCE Today』（共著、米国Prentice-Hall、1998）
『ソフトウェアパターン再考』（共著、日科技連出版社、2000）
『コンポーネントモデリングガイド』（監修、ピアソン・エデュケーション、2001）
『eXtreme Programmingテスト技法』（監修、翔泳社、2001）
『オブジェクト脳のつくり方』（監修、翔泳社、2003）
『UMLシステム設計実践技大全』（監修、ナツメ社、2003）
『独習デザインパターン』（監修、翔泳社、2003）
『UMLトレーニングブック』（監修、ソーテック社、2004）
『UML 2 ハンドブック』（監修、翔泳社、2004）

【訳書】

『パターンハッチング』（共訳、ピアソン・エデュケーション、1999）

『独習UML』（監訳、翔泳社、2000）
『XPエクストリーム・プログラミング入門』（監訳、ピアソン・エデュケーション、2000）
『XPエクストリーム・プログラミングアドベンチャー』（監訳、ピアソン・エデュケーション、2002）
『アジャイルソフトウェア開発』（監訳、ピアソン・エデュケーション、2002）
『テスト駆動開発入門』（監訳、ピアソン・エデュケーション、2003）
『プログラミングJakarta Struts』（監訳、オライリー・ジャパン、2003）
『Executable UML』（監訳、翔泳社、2003）
『MDA（モデル駆動型アーキテクチャ）導入ガイド』（監訳、インプレス、2003）
『実践eXtremeプログラミング』（監訳、九天社、2004）
『アジャイルソフトウェア開発ソフトウェアチーム』（監訳、九天社、2004）

■ 訳者紹介

永田 渉（ながた わたる）

1991年東京大学文学部社会心理学専修課程卒業。トランス・コスモス株式会社、マイルストーン株式会社を経て、1994年株式会社テクノロジックアート入社。現在、分析、実装、テスト、執筆、教育などシステム開発のあらゆるフェーズに携わっている。方法論やオブジェクト指向に関する書籍および雑誌記事の執筆多数。

【訳書】

『パターンハッチング』（共訳、ピアソン・エデュケーション, 1999）
『XPエクストリーム・プログラミング入門』（共訳、ピアソン・エデュケーション, 2000）『アジャイルソフトウェア開発』（共訳、ピアソン・エデュケーション, 2002）
『アジャイルプロジェクト管理』（共訳、ピアソン・エデュケーション, 2002）
『初級デザインパターン』（共訳、ピアソン・エデュケーション, 2002）
『プログラマのためのJava設計ベストプラクティス』（共訳、ピアソン・エデュケーション, 2002）
『入門Carbon』（共訳、オライリー・ジャパン, 2001）
『入門Cocoa』（共訳、オライリー・ジャパン, 2002）
『Webサービスエッセンシャルズ』（共訳、オライリー・ジャパン, 2002）
『Java Webサービス』（共訳、オライリー・ジャパン, 2002）
『プログラミングJakarta Struts』（オライリー・ジャパン, 2003）
『Jakarta Struts デスクトップリファレンス』（オライリー・ジャパン, 2003）

坂本 武志（さかもと たけし）

1994年千葉大学大学院工学研究科修了。同年産業用電気機器メーカに入社。人間共存型ロボットの研究・開発に従事。2000年医療系ソフトウェア開発会社に入社。電子カルテシステムを中心とした医療情報システムの開発に従事。2004年株式会社テクノロジックアート入社。現在、UML Profile for EDOCやRM-ODPを用いた医療分野のモデリング、組込分野における開発方法論の開発に従事。

【著書】

『独習UML 第3版』（共著、翔泳社、2005）

【訳書】

『マインドストーム・プログラム入門－LEGOでメカトロニクス/ロボティクスを学習するCOMPUTER TECHBOLOGY』（CQ出版）

藤川 幸一（ふじかわ こういち）

2001年早稲田大学理工学部卒業。同年ヤフー株式会社入社。2004年株式会社テクノロジックアート入社。現在、UMLやMDAなどの執筆やセミナー活動、ebXML（UMM）を用いたEDIのモデリングに従事。

【著書】

『JSPプログラミングステップアップラーニング［応用編］』（技術評論社）

『独習UML 第3版』（共著、翔泳社、2005）

武田 知子（たけだ ともこ）

1993年川村短期大学英文科卒業。生命保険会社を経て、2002年株式会社テクノロジックアート入社。現在、UML関連書籍の翻訳、執筆等に従事。

【著書】

『基礎UML』（共著、インプレス、2003）

『UMLモデリング教科書 UMLモデリングL1（T1・T2対応）』（共著、翔泳社、2004）

【訳書】

『ワークブック形式で学ぶUMLオブジェクトモデリング－「ユースケース駆動」でソフトウェアを開発する』（共訳、ソフトバンクパブリッシング、2002）

『Developing Applications with JAVA and UML－プログラマーのための統一モデリング』（共訳、ピーエヌエヌ新社、2003）

『JavaによるExtreme Programmingクックブック』（共訳、オライリー・ジャパン、2003）

『MDA（モデル駆動型アーキテクチャ）導入ガイド』（共訳、インプレス、2003）

『リファクタリングワークブック－設計の改善テクニックを学ぶ』（共訳、アスキー、2004）

装丁・本文デザイン　河原田智（ポルターハウス）
DTP・翻訳協力　株式会社コスモユノー

エンタープライズアプリケーション アーキテクチャパターン

2005年 4月20日　初版第1刷発行
2012年 4月 5日　初版第5刷発行

著　者　Martin Fowler（マーチン・ファウラー）
監　訳　長瀬 嘉秀（ながせ よしひで）
翻　訳　株式会社テクノロジックアート
発行人　佐々木 幹夫
発行所　株式会社 翔泳社（http://www.shoeisha.co.jp/）
印刷・製本　大日本印刷株式会社

＊本書は著作権法上の保護を受けています。本書の一部または全部について（ソフトウェアおよびプログラムを含む）、株式会社 翔泳社から文書による許諾を得ずに、いかなる方法においても無断で複写、複製することは禁じられています。

＊本書へのお問い合わせについては、iiページに記載の内容をお読みください。

＊落丁・乱丁はお取り替えいたします。03-5362-3705までご連絡ください。

ISBN4-7981-0553-8　Printed in Japan

目的別パターン一覧

ドメインロジックを構造化するには

シンプルなロジックの場合 → トランザクションスクリプト（p115）
複雑なロジックの場合 → ドメインモデル（p123）
ロジックの複雑さは普通で、レコードセットに関する優れたツールがある場合
→ テーブルモジュール（p133）

ドメインロジックに明確な API を提供するには

→ サービスレイヤ（p142）

Web プレゼンテーションを構造化するには

→ モデルビューコントローラ（p351）

HTTP リクエストの処理を組織化するには

ほとんどが直接ページ URL を用いたシンプルなアプリケーションフローの場合
→ ページコントローラ（p354）
複雑なフローの場合 → フロントコントローラ（p366）
国際化または柔軟性のあるセキュリティポリシーが必要な場合 → フロントコントローラ（p366）

Web ページの書式を制御するには

ページを編集し、動的なデータ用にフックを挿入したい場合 → テンプレートビュー（p373）
ページを、ドメインデータ（おそらく XML）を変換したものと見なす場合
→ トランスフォームビュー（p384）
サイトの外観に全面的な変更が必要な場合 → ツーステップビュー（p388）
論理的には同じ画面フォーマットに、複数の外観が必要な場合 → ツーステップビュー（p388）

複雑なアプリケーションフローを制御するには

→ アプリケーションコントローラ（p403）

データベースと相互作用を行うには

トランザクションスクリプトを使用する場合 → 行データゲートウェイ（p162）
トランザクションスクリプトを使用し、プラットフォームがレコードセットを十分にサポートする場合
→ テーブルデータゲートウェイ（p153）
データベーステーブルに厳密に対応するドメインモデルがある場合 → アクティブレコード（p170）
十分なドメインモデルがある場合 → データマッパー（p175）

メモリ上の異なる場所で、同じデータベースデータを更新しないようにするには

→ 一意マッピング（p209）

データベースレコードにドメインオブジェクトをリンクさせておくには

→ 一意フィールド（p231）

データベースにドメインデータをマッピングするためのコードを最小限にするには

→ メタデータマッピング（p327）

ドメインモデルでデータベースにクエリーを行うには

→ クエリーオブジェクト（p338）

オブジェクトの関連をデータベースに格納するには

オブジェクトへの参照が1つの場合 → 外部キーマッピング（p254）

オブジェクトのコレクションがある場合 → 外部キーマッピング（p254）

多対多の関係がある場合 → 関連テーブルマッピング（p266）

他のオブジェクトと共有しないオブジェクトのコレクションがある場合 → 依存マッピング（p282）

バリューオブジェクトであるフィールドがある場合 → 組込バリュー（p288）

データベースの他の部分が使用しない複雑に絡み合っているオブジェクトがある場合

→ シリアライズLOB（p292）

データベース全体をメモリに読み込むのを避けるには

→ レイジーロード（p213）

継承構造をリレーショナルデータベースに格納するには

→ シングルテーブル継承（p298）

1つのテーブルがデータベースアクセスのボトルネックとして作用する場合

→ クラステーブル継承（p306）

1つのテーブルがデータベースのスペースを無駄にする場合 → クラステーブル継承（p306）

結合が多すぎるが、1つのテーブルとはしたくない場合 → 具象テーブル継承（p314）

読み込んだり変更したオブジェクトを追跡するには

→ ユニットオブワーク（p197）

1つのクライアントリクエスト内で、データベースへの変更をコミットできるようにするには

→ 軽オフラインロック（p439）

ユーザの作業の損失を防止する場合 → 重オフラインロック（p450）

1つのロックで関連オブジェクトをすべてロックするには

→ 緩ロック（p462）

ロック戦略において妥協を避けるには

→ 暗黙ロック（p474）

リモートから細かいオブジェクトにアクセスするには

→ リモートファサード（p411）

1つのリモートコールで多くのオブジェクトからデータを渡すには

→ データ変換オブジェクト（p425）

ステートフルなビジネストランザクションの途中の状態を格納するには

ステートがあまりない場合 → クライアントセッションステート（p479）

多くのステートがある場合 → サーバセッションステート（p481）

データベースへ処理中の作業をコミットできるようにする場合

→ データベースセッションステート（p485）

パターン一覧

アクティブレコード (p170)：データベーステーブルまたはビューの行をラップし、データベースアクセスをカプセル化してデータにドメインロジックを追加するオブジェクト。

アプリケーションコントローラ (p403)：アプリケーションの画面ナビゲーションとフローを扱うための集中管理ポイント。

暗黙ロック (p474)：フレームワークまたはレイヤスーパータイプコードがオフラインロックを獲得できる。

依存マッピング (p282)：任意のクラスに対して、子クラスへのデータマッピングを実行させる。

一意フィールド (p231)：データベースＩＤフィールドをオブジェクトに保存することで、メモリ上のオブジェクトとデータベース行との一意性を維持する。

一意マッピング (p209)：読み込まれたすべてのオブジェクトを１つのマッピングに保存することで、各オブジェクトが確実に一度だけ読み込まれるようにする。オブジェクトを参照する場合には、マッピングを使って参照する。

外部キーマッピング (p254)：オブジェクト間の関係を、テーブル間の外部キー参照にマッピングする。

関連テーブルマッピング (p266)：リンクされたテーブルへの外部キーを持つテーブルとして、関連を保存する。

緩ロック (p462)：１つのロックで関連オブジェクトセットをロックする。

行データゲートウェイ (p162)：データソース内の１つのレコードに対してゲートウェイの役割を果たすオブジェクト。行ごとに１つのインスタンスが存在する。

クエリーオブジェクト (p338)：データベースクエリーとして機能するオブジェクト。

具象テーブル継承 (p314)：階層構造の具象クラスごとに１つのテーブルを使って、クラスの継承階層構造を作成する。

組込バリュー (p288)：オブジェクトを他のオブジェクトテーブルの複数フィールドにマッピングする。

クライアントセッションステート (p479)：セッションステートをクライアントに格納する。

クラステーブル継承 (p306)：クラスごとに１つのテーブルを使って、クラスの継承階層構造を作成する。

軽オフラインロック (p439)：コンフリクトの検出とトランザクションのロールバックによって、同時に発生するビジネストランザクション間のコンフリクトを防止する。

継承マッパー (p324)：継承階層構造を処理するデータベースマッパーを組織化するための構造。

ゲートウェイ (p489)：外部システムまたはリソースへのアクセスをカプセル化するオブジェクト。

サーバセッションステート (p481)：サーバシステムのセッションステートを直列化形式で保持する。

サービススタブ (p526)：テスト中に問題となるサービスへの依存性を除去する。

サービスレイヤ (p142)：サービスのレイヤとアプリケーションの境界を定義する。サービスは利用できる操作セットを設定し、各操作に対するアプリケーションのレスポンスを調整するものである。

重オフラインロック (p450)：データへのアクセスを一度に１つのビジネストランザクションに限定することで、同時ビジネストランザクション間のコンフリクトを防止する。

シリアライズ LOB (p292)：オブジェクトのグラフを直列化し、１つの大規模オブジェクト（LOB）としてデータベースフィールドに格納する。

シングルテーブル継承 (p298)：クラスの継承階層構造を、さまざまなクラスのフィールドに対する列を持つ、１つのテーブルとして作成する。

スペシャルケース (p518)：特定のケースで特別な振る舞いを提供するサブクラス。

セパレートインタフェース (p499)：実装から分離したパッケージでインタフェースを定義する。
ツーステップビュー (p388)：ドメインデータを2つの手順でHTMLに変換する。まずは論理ページを形成し、次に論理ページをHTMLへと加工する。
データベースセッションステート (p485)：コミットされたデータのセッションデータをデータベースに格納する。
データ変換オブジェクト (p425)：メッセージ呼び出しの数を削減するため、プロセス間のデータを伝送するオブジェクト。
データマッパー (p175)：オブジェクト、データベース、およびマッパー自体の独立性を保ちつつ、オブジェクトとデータベース間でデータを移動するマッパーのレイヤ。
テーブルデータゲートウェイ (p153)：データベーステーブルに対してゲートウェイの役割を果たすオブジェクト。1つのインスタンスがテーブル内のすべての行を処理する。
テーブルモジュール (p133)：データベーステーブルかビューのすべての行に関するビジネスロジックを扱うシングルインスタンス。
テンプレートビュー (p373)：HTMLページにマークを埋め込むことで、HTMLへ情報を加工する。
ドメインモデル (p123)：振る舞いとデータの両方を一体化させたドメインのオブジェクトモデル。
トランザクションスクリプト (p115)：ビジネスロジックを一連の手続きで構築して、その各手順でプレゼンテーションからの1つの要求を処理する。
トランスフォームビュー (p384)：要素ごとにドメインデータ要素を処理しHTMLへと変換するビュー。
バリューオブジェクト (p508)：IDに基づいた等価性を確保していない、MoneyやDate Rangeなどのシンプルな小型オブジェクト。
プラグイン (p521)：コンパイル時ではなくシステム構成の際にクラスをリンクする。
フロントコントローラ (p366)：Webサイトへのあらゆるリクエストを扱うコントローラ。
ページコントローラ (p354)：特定のページに対するリクエストやWebサイト上のアクションを扱うオブジェクト。
マッパー (p496)：独立した2つのオブジェクト間の通信を設定するオブジェクト。
マネー (p510)：貨幣の値として機能する。
メタデータマッピング (p327)：メタデータでのオブジェクトリレーショナルマッピングの詳細を保持する。
モデルビューコントローラ (p351)：ユーザインタフェースの相互作用を3つの明確な役割へ分割する。
ユニットオブワーク (p197)：ビジネストランザクションの影響を受けるオブジェクトのリストを保持しつつ、変更点の書き込みと並行性の問題の解決を調整する。
リポジトリ (p345)：ドメインオブジェクトにアクセスするためのコレクション型インタフェースを使って、ドメインとデータマッピングレイヤとを仲介する。
リモートファサード (p411)：ネットワークの効率性を向上させるため、細かい粒度のオブジェクトに対して粗い粒度のファサードを提供する。
レイジーロード (p213)：必要なすべてのデータは含まないが、その取得方法を知っているオブジェクト。
レイヤスーパータイプ (p497)：レイヤのすべてのタイプに対して、スーパータイプとしての役割を果たすタイプ。
レコードセット (p530)：テーブルデータのメモリ上の表現。
レジストリ (p502)：他のオブジェクトが一般的なオブジェクトとサービスを検索するために使う既知のオブジェクト。