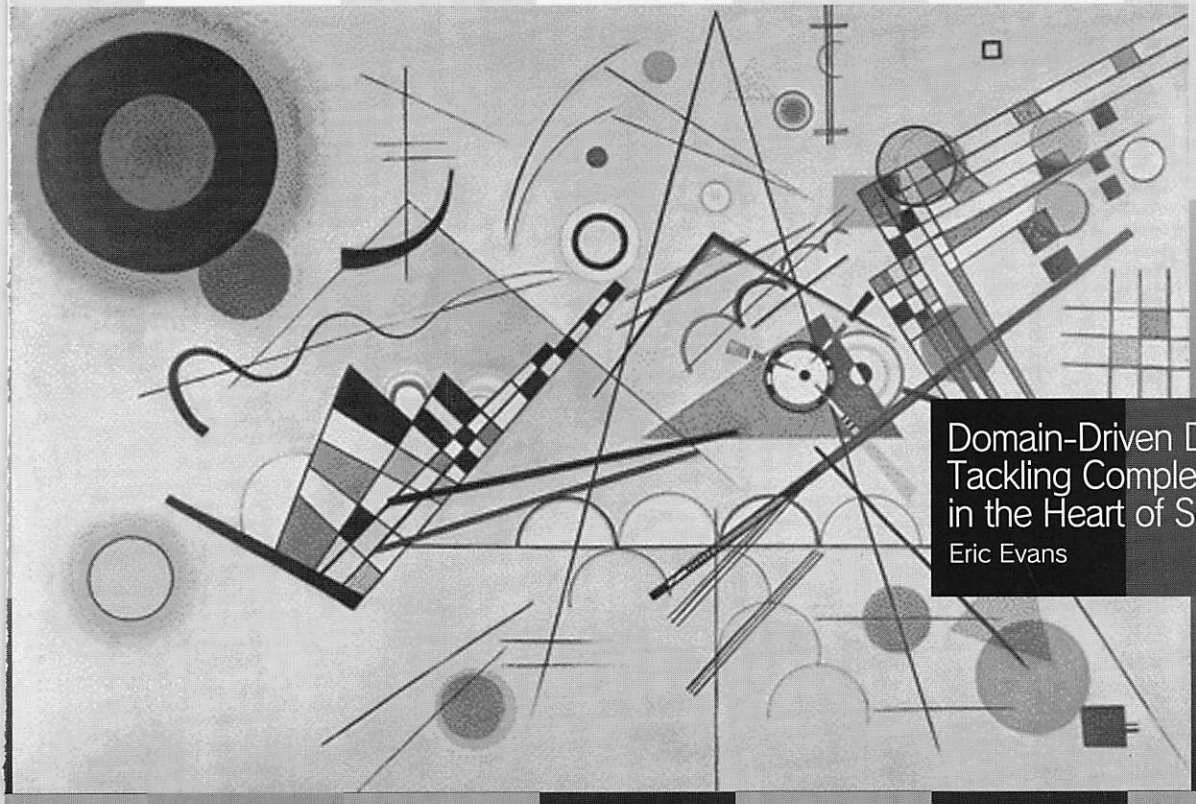
ソフトウェア開発の実践

エリック・エヴァンスのドメイン駆動設計

ソフトウェアの核心にある複雑さに立ち向かう

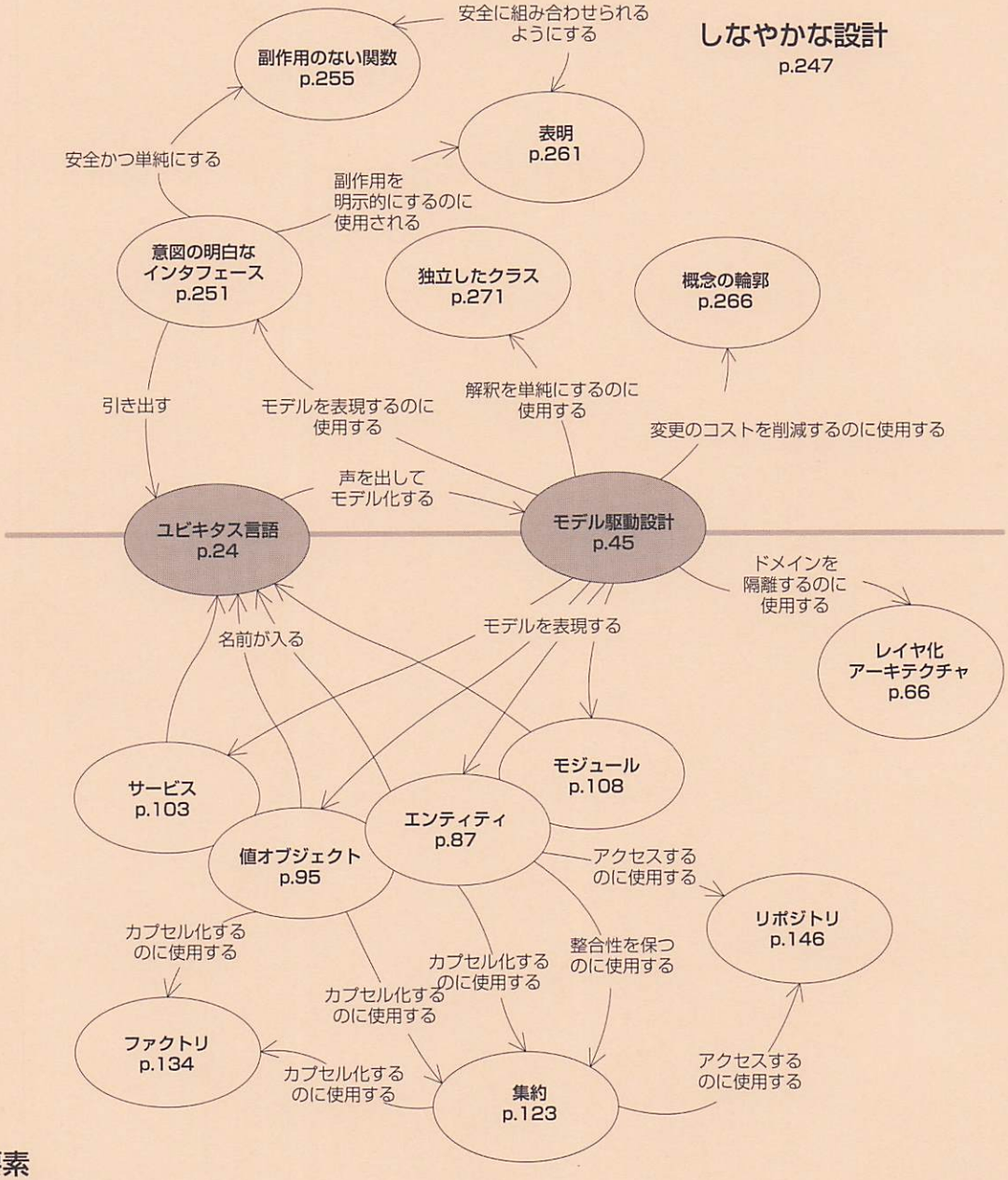
エリック・エヴァンス著　今関 剛 監訳　和智 右桂、牧野 祐子 訳



Domain-Driven Design:
Tackling Complexity
in the Heart of Software
Eric Evans

達成を誓うすべてのITプロフェッショナルに奉げる世界的名著

IT Architects' Archive
SE SHOEISHA


しなやかな設計
p.247
副作用のない関数
p.255
安全に組み合わせられる
ようにする
表明
p.261
安全かつ単純にする
副作用を
明示的にするのに
使用される
意図の明白な
インタフェース
p.251
独立したクラス
p.271
概念の輪郭
p.266
引き出す
モデルを表現するのに
使用する
解釈を単純にするのに
使用する
変更のコストを削減するのに使用する
声を出して
モデル化する
ユビキタス言語
p.24
モデル駆動設計
p.45
ドメインを
隔離するのに
使用する
レイヤ化
アーキテクチャ
p.66
名前が入る
モデルを表現する
サービス
p.103
モジュール
p.108
エンティティ
p.87
値オブジェクト
p.95
アクセスする
のに使用する
リポジトリ
p.146
カプセル化する
のに使用する
カプセル化する
のに使用する
整合性を保つ
のに使用する
カプセル化する
のに使用する
ファクトリ
p.134
カプセル化する
のに使用する
集約
p.123
アクセスする
のに使用する
構成要素
p.61

本書に寄せられた賛辞

「この本は、思慮深いソフトウェア開発者全員の必携書である。」

—Kent Beck

「ソフトウェアの設計を、今取り組んでいる問題ドメインのメンタルモデルに適合させるにはどうすればよいか、ということについて、Eric Evansは素晴しい本を書いた。」

「この本はXPと相性がいい。この本が対象としているのは、ドメインの図を描くことではない。述べられているのは、ドメインについてどう考えるか、ドメインについて語る際に用いる言語がどのようなものか、さらに、精緻になった理解を反映するためにどうソフトウェアをまとめ上げるか、といったことである。問題ドメインの習得は、プロジェクトの開始時と同様に終わる頃にも起こり得るとEricは考えているので、リファクタリングが彼のテクニックにおける重要な要素となっている。」

「この本は読んでいて実に楽しい。Ericは興味深い話を豊富に持っていて、言葉づかいも巧みだ。この本はソフトウェア開発者の必読書だと思う。いずれは古典と呼ばれるだろう。」

—Ralph Johnson『Design Patterns』の著者

「オブジェクト指向プログラミングで、投資に見合った効果が得られないと思っているなら、本書を読むことだ。何をし忘れたかを教えてくれるだろう。」

—Ward Cunningham

「Ericが見事にとらえたのは、熟練のオブジェクト設計者が常々用いてきた設計プロセスの一部でありながら、グループとして見ると、この業界の他の人々へうまく伝えられずにいたものだ。これまで我々は、この知識を断片的には提供してきた。しかし、ドメインロジックを構築するための原理をまとめ上げ、体系化したことはなかった。本書は重要である。」

—Kyle Brown『Enterprise Java Programming with IBM WebSphere』の著者

「Eric Evansは、開発の焦点としてのドメインモデリングの重要性について説得力のある理論を唱え、それを実現するための確固たるフレームワークとテクニック一式を提供している。これは時間を超越した知恵であり、流行の方法論が時代遅れになった後も、ずっと生き続けることだろう。」

—Dave Collins『Designing Object-Oriented User Interfaces』の著者

「Ericは、ビジネスアプリケーションをモデリングし構築するという現実世界の体験を、実用的で役立つ本にした。信頼できる実践者の視点から書かれているので、ユビキタス言語、モデルをユーザと共有することの利点、オブジェクトのライフサイクル管理、論理的および物理的アプリケーション構造、そして、深いリファクタリングのプロセスとその結果など、これらに関する彼の記述は、我々の分野に大きく貢献している。」

—Luke Hohmann『Beyond Software Architecture』の著者

ソフトウェア開発の実践

エリック・エヴァンスのドメイン駆動設計

ソフトウェアの核心にある複雑さに立ち向かう

エリック・エヴァンス著　今関 剛 監訳　和智 右桂、牧野 祐子 訳

Domain-Driven Design:
Tackling Complexity
in the Heart of Software

Eric Evans

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.shoeisha.co.jp/book/errata/

●ご質問方法
弊社Webサイトの「刊行物Q&A」をご利用ください。
刊行物Q&A　http://www.shoeisha.co.jp/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“翔泳社 愛読者サービスセンター”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、あらかじめご了承ください。

●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 愛読者サービスセンター

※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、著者や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スクリーンイメージなどは、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。

Domain-Driven Design: Tackling complexity in the heart of software/ Eric Evans.
Copyright © 2004 by Eric Evans

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher. Printed in the United States of America. Published simultaneously in Canada.

Japanese translation rights arranged with Pearson Education, Inc., Publishing as Addison-Wesley Professional through Japan UNI Agency, Inc., Tokyo.

日本語版への序文

Domain-Driven Design（DDD）が日本語に翻訳されて、大変うれしく思っています。この本の原書が出版されたのは、2003年のことでした。技術書としてみれば、かなりの時間が経ったことになります。それでもドメイン駆動設計は、控えめに見ても、今日でも重要性を失っていません。技術の変化に対応して、DDDのコミュニティは新しいフレームワークやプログラミング言語にDDDを適用させる方法を見出してきましたが、その一方で、コアとなる原則は同じままに保たれてきたのです。

出版から今日までの7年間で、DDDの基礎となる原則が本当は何であったのかが、より一層明確になりました。根本的には、DDDを駆動している原則は次の3つだけです。

- コアドメインに集中すること。
- ドメインの実践者とソフトウェアの実践者による創造的な共同作業を通じて、モデルを探究すること。
- 明示的な境界づけられたコンテキストの内部で、ユビキタス言語を語ること。

この原則はきわめて簡潔です。しかしもちろん、これで十分ではありません。まず、この原則はいくつかの用語の定義に依存しています。そして、この定義は本書の最も重要な内容の一部なのです。また、これらの原則を明確に理解していたとしても、成し遂げるのは容易ではありません。

それを実現することが、本書で取り上げる多くのパターンの目的となっています。こうした原則を適用しようとすると、なんらかの障害に直面することになるのが常ですが、それらの障害に対する優れた解決策が、本書のパターンによって説明されているのです。こうしたパターンが一体となってDDDが定義され、同時に、どうやってDDDを機能させるかが示されます。

時間が経つにつれて、私がより一層の価値を置くようになった要素が、実はもう1つあります。ドメイン駆動設計が語っているのは、私の物語です。原則やパターンを紹介する上で、無味乾燥な技術的方法をとってはいません。その代わりに、2000年代初頭に、私がこの本を書いていた時に持っていた展望が書かれています。これは一人の人間の物語です。

しかしこの物語は、厳しい現実の中でソフトウェア設計を習得しようと奮闘してきた、他の多くの人々の物語にも似ています。こうした物語は、抽象的な原則を実際の経験の中につなぎ止める錨となるのであり、不完全な状況の中でその原則を適用する際の助けになるのです。私が気に入っている例の1つが、第8章「ブレイクスルー」です。深刻で頑固な問題を解決する優れたドメインモデルを発見することは、常に望ましいわけではありません。そして、その理由を伝えられるのは、真実の物語だけです。偉大なソフトウェアを作り上げるためには、知性と忍耐が必要です。同様に勇気が必要であり、リスクもある程度は受け入れなければならないのです。

エリック・エヴァンス（Eric Evans）

日本語版推薦文

IT業界の七不思議。奇想天外、驚天動地、日進月歩なこの業界、数え上げれば不可思議なことの7つなんてすぐに枚挙できてしまうのだが、中でも最上級の不思議がこれ。*Domain-Driven Design*（DDD）の邦訳が待てど暮らせど出版されないことであった。

Eric EvansがDDDのアイデアを著書のドラフトというかたちでWeb上で公開し始めた2002年以来、この知恵の書は多くの識者の間で話題の中心となり、「徹頭徹尾有用な書籍（most thoroughly useful book）」とまで評された。記述されている内容の一言一句すべてにおいて役に立つことしか書かれていない、と賞賛された書籍が、この長きにわたって日本語で紹介されていなかったことは、まったく謎なのである。かく言う私もDDDで多くのことを学んだ一人であり、誰もやらないなら自分が翻訳すると意気込み、出版社に翻訳オプションの問い合わせを行ったこともある。が、返答は「既に翻訳権が取得されている」というもの。何度も言うが、今まで出版されなかったことは私にとってはバミューダトライアングルよりもネス湖の恐竜ネッシーよりも深い深い謎だった。

しかし、このたびめでたく邦訳が出版され、広く日本のプラクティショナーに本書が紹介されるに至ったことは非常に喜ばしいことだ。DDDを一言で言うと、ドメイン知識とIT開発をスムーズにつなぐ実践的知識の宝庫、である。EricはIT技術者に向けて、ITの語法だけではない技術書を作り上げた。第1章「知識をかみ砕く」を、第2章「コミュニケーションと言語の使い方」を読んでみてほしい。書籍の冒頭からドメイン知識を得ることの有用性と、抽象的な議論に陥らずにドメインエキスパートとIT技術者が有益な協力を行うことの重要性に言及している。この書籍の中で新しく提唱された言葉である「ユビキタス言語」という考え方はまさにそうした根本概念を端的に表すものだと言ってもいいだろう。とかくIT技術者は「ビジネスサイド」の人間との協業を自ら制限しがちである。曰く、このモデルは彼らには抽象的すぎる、システム設計の話をしても理解してもらえないだろう、云々。しかし、本書を読み進めるならばドメイン知識を得て、正しいものを実装していくことへの勇気が得られるはずだ。もう、自分自身を縛り付ける必要はない。

本書を貫くもう1つのテーマはリファクタリングだ。Ericはリファクタリングの概念をオブジェクト設計のみならず、ドメイン知識の組み立て部分にまで拡大して適用を図ってい

る。リファクタリングの目指すところは、シンプル化と改善だ。ドメイン駆動の設計においては、シンプルなドメイン知識を膨らませて、大規模な構造にも太刀打ちできる術を提示してくれる。無駄を排除して着実に設計を進めるミニマリズムという点では、アジャイル手法とも相通じるものがある。ともに現代的な開発において無視できない考え方である。原著に推薦の言葉を書いているIBM WebSphere開発プロジェクトのアーキテクトの一人である、Kyle Brownも、以前私と話している時に、DDDによってWebSphereチームに変化が起きたと言っていた。彼らの開発はよりシンプルに、より的確なリリースを行うことができるようになったそうだ。

本書の内容は様々なレベルの技術者、様々な立場や役割を担う技術者に読んでもらえるようになっている。詳しくはEricの序章を見ていただければお分かりになると思うが、本書はあなただけではなく、あなたの仲間にも、あなたの上司や後輩にも薦められるものだ。きっと、この邦訳の出版をきっかけに日本全国で勉強会が自主的に組織されるだろう。あなたもその和の中に加わる大事な一人となっていただきたい。

2011年3月　ソウルにて

榊原　彰

日本アイ・ビー・エム株式会社　グローバル・ビジネス・サービス事業CTO

＊　＊　＊

世に名高いEric EvansのDDD（Domain Driven Design）本は翻訳されるべき本の筆頭であったけれど、原著の出版から7年を経てようやく世に出ることになったのは長かっただけにその価値は大きい。DDDのコンセプトの重要性はみんなが理解しながら、それを実践できるだけの土壌・カルチャーができあがっていなかったということなのだと思う。それは、欧米でも同じで、OOPSLA 2004でのEric自身によるワークショップに私も参加したが、当時それほど盛り上がっている雰囲気はなかった。その後、「DDD難民」などという言葉も囁かれた。それがクラウドコンピューティングという大きな環境の激変の中で、現場に入って行って現場の声と一体化したシステム作りのできないソフトウェア屋さんは存在価値がないという時代の訪れとともに、DDDの重要性の再認識どころかその実践の必要性に迫られている。クラウド時代には、ビジネスはもちろん行政、医療、農業や福祉といった生活の現場でいかにITを利活用するかが問題となる。まさに生活世界（ドメイン）からのシステムの立ち上げが求められ、DDDがようやく開花する土壌が整ったといえる。

DDDは、まさに現場の声と開発チームとソフトウェアとをつなぐ共通の言語をじっくりと作りながら開発を進めていこうという当たり前のことを、ドメインモデルを語彙データベースとしてうまく保守管理しながら、アジャイルにPDCAを回しながら行っていく方法をパターンランゲージという複数のステークホルダーにもわかりやすい記述フォーマット

で提供している。過去のオブジェクト指向方法論が目指して挫折していた現実世界をモデル化してソフトウェア世界をスムーズに繋ぐという夢は、DDDにおいてはドメインを記述する単位をオブジェクトからより自然な「ユビキタス言語」というDSLの次元に持ち上げることで、より的確に表現されている。これでまた夢に一歩近づいたといえる。

DDDがAlexanderの思想を受け継ぎ、パターンランゲージの形式を取っている事の意味は大きい。デザインパターンではうまく果たせなかったパターン同士の有機的なネットワーク化と、ユビキタス言語というランゲージを使って現場エキスパートという生活世界の住民と市や行政といった他のステークホルダを繋ぎ、ワークショップ形式で現場検証しながら現場の声をアーキテクチャに反映していくプロセスを駆動する「アーキテクトビルダー（住民窓口兼アーキテクト兼現場監督兼大工さん）」の存在の重要性もDDDではうまく謳われているからだ。こうしたドメイン価値駆動、モデル管理、アーキテクチャ中心、アジャイルプロセスといった複数のプラクティスが有機的に結びついて初めてDDDはうまく機能する。だからそれだけ実践は困難だと考えるには及ばない。それぞれのプラクティスはパターンとして言語化されており、誰しもが日々の生活の中で楽しみながら言葉をいつとはなしに自然に話し始めているように、DDDもまずは勇気を持って話してみることが大事だと思う。

2011年3月7日

羽生田栄一

株式会社豆蔵取締役CTO/プロフェッショナル・フェロー

＊　＊　＊

「ドメイン駆動設計」（DDD: Domain Driven Design）は、エリック・エヴァンスとコミュニティによって書かれた、ソフトウェア開発、デザイン活動を包含するパターン言語です。この長く邦訳が待たれた本書は、私にとっては、オブジェクト指向分析設計活動の終着到達点、と言ってしまってもいいくらい、大きな意味を持った内容です。

コーンウェイの法則によれば、ソフトウェアのアーキテクチャはチームの構造と相似形だといいます。すなわち、ソフトウェアのデザインは、チームのデザインでもあるのです。本書はその両方の視点からデザイン活動を記述しています。プロセスとしての開発手法に依らず、「よいソフトウェアをデザインしたい」と考えている方、「よいソフトウェア開発チームを作りたい」と考えている方には、ぜひ読んでいただきたい。

私自身がDDDの中で特に気に入っているのは「ユビキタス言語」。ケント・ベックがエクストリームプログラミングにおいて「メタファ」（Metaphor）というプラクティスで表現しようとしたのは、顧客から開発者までが会話できる共通ボキャブラリの提供です。これこそが、DDDにおいてユビキタス言語と呼ばれているプラクティスであり、ソフトウェア

デザイン活動のコアであろう、と私は考えています。

日本でも、本書邦訳出版の機会に、DDD普及活動が再燃することを期待しています。

2011年3月6日

平鍋健児

株式会社チェンジビジョン代表取締役

序　文

ソフトウェア開発を複雑にする要因は数多く存在する。しかし、この複雑さの核心にあるのは、問題ドメイン自体が本質的に複雑であるという事実だ。人間の行う入り組んだ仕事を何らかのかたちで自動化しようとすれば、ソフトウェアはそこにある複雑さを避けて通ることができない。できるのは、コントロールすることだけだ。

複雑さをコントロールする秘訣は、優れたドメインモデルにある。このモデルは、ドメインに関する表層的なビジョンを超えるために、ドメインの根底に存在する構造を導入する。このモデルによって、ソフトウェア開発者は必要な力を与えられる。優れたドメインモデルは、信じられないほどの価値をもたらし得るが、容易に作成できるものでもない。的確に作成できる人がほとんどいない上、教えるのも非常に難しいのである。

Eric Evansは、ドメインモデルを巧みに作成できる数少ない一人だ。彼と一緒に仕事をしていて、私はそのことに気がついた。それは自分よりも高いスキルを持つクライアントに出会うという、素晴しい経験の1つであった。私たちが共同作業をした期間は短かったが、実に楽しかった。その後も私たちは連絡を取り合い、本書がゆっくりと練り上げられていくのを、私はずっと見守ってきたのである。

待った甲斐は十二分にある。

本書は偉大な野望を実現するものへと進化した。野望とはすなわち、ドメインモデリングの技法そのものに関する語彙を記述し、構築すること。そして、基準となる枠組みを提供することにより、ドメインモデリングという活動を説明し、同時に、この習得困難なスキルを教えられるようにすること。この本は、形が整うにつれて、私に新しい発想を数多く与えてくれた。概念モデリングのベテランであっても、本書を読んで新しい着想を得ないということはまずないだろう。

さらにEricは、私たちが何年もかけて学んできた多くの事柄を確かなものにしている。まず、ドメインモデリングにおいては、概念を実装から切り離してはならない。優秀なドメインモデラは、経理担当者と一緒にホワイトボードを使うこともできれば、プログラマと一緒にJavaを書くこともできる。概念が実装から切り離せない理由の一端には、実装上の問題を考慮せずには役に立つ概念モデルを構築することができない、ということもある。

しかし、概念と実装を一緒に考える一番の理由は、ドメインモデルの最大の価値が、**ユビキタス言語**を提供し、ドメインエキスパートと技術者を結びつけることにある、ということだ。

本書から得られるもう1つの教訓は、ドメインモデルというものが、最初にモデル化され、その後で実装されるようなものではない、ということだ。「まず設計し、その後に構築する」という段階を区切った考え方に対しては、多くの人々と同様に私も拒絶するようになってきている。だが、Ericの経験から導き出される教訓は、真に強力なドメインモデルは時間と共に進化するということであり、偉大なベテランモデラであっても、最高のアイデアを得られるのはシステムが初期リリースされた後のことだ、ということである。

本書はきわめて影響力の大きな本になると思うし、そうなることを願ってもいる。つかみ所のない非常に不安定な分野に構造と凝集性を与える一方で、価値あるツールの使い方を多くの人々に教える本なのだ。ドメインモデルは、ソフトウェア開発をコントロールする上で大きな影響を与えることができる。実装に使われる言語や環境が何であっても、このことは変わらない。

最後に、重要な考えを1つ。本書に関して、私が最も尊敬していることの1つに、うまくいかなかった時のことをEricが堂々と述べているという点がある。たいていの著者は、公平無私で全能だという雰囲気を保ちたがるものだ。だが、Ericが明らかにしているのは、彼が私たちと同じように、成功も失敗も味わってきたということだ。重要なのは、Ericが成功と失敗の両方から学べるということである。そして、私たちにとってはさらに重要なことだが、Ericは自分の得た教訓を私たちに伝えることもできるのだ。

2003年4月

マーチン・ファウラー（Martin Fowler）

まえがき

先進的なソフトウェア設計者は、少なくともここ20年の間、ドメインモデリングと設計が重大なテーマであると認識してきた。しかし、何をする必要があって、それをどのようにすべきかについては、驚くほどわずかしか書かれていない。だが、明確に体系化されてはいないものの、オブジェクトコミュニティの底流として、ある哲学が出現してきている。私はその哲学を、**ドメイン駆動設計**と呼んでいる。

私はこの10年間、いくつかのビジネスドメインや技術ドメインにおいて、複雑なシステムを開発してきた。仕事上では、オブジェクト指向開発の先駆者たちが編み出してくる、設計や開発プロセスのベストプラクティスを、いち早く試してきた。私が携わったプロジェクトの中には、大成功したものもいくつかあるが、失敗したものも多少ある。成功したプロジェクトに共通する特徴は、豊かなドメインモデルがあったということだ。このドメインモデルは設計のイテレーションを通じて進化しながら、プロジェクトの基礎の一部となった。

本書が提供するのは、設計上の意思決定を行うためのフレームワークと、ドメイン設計について議論するための技術的な語彙である。これは、広く受け入れられているベストプラクティスを、私自身の洞察や経験と統合したものだ。複雑なドメインに直面しているソフトウェア開発チームは、このフレームワークを使用して、ドメイン駆動設計に体系的に取り組めるだろう。

対照的な3つのプロジェクト

ドメイン設計のプラクティスが、開発結果に対してどれほど劇的に影響を及ぼすかを考えてみると、覚えている中では、3つのプロジェクトが生きた事例として際立っている。3つのプロジェクトはいずれも役に立つソフトウェアを完成させたが、野心的な目標を達成して、企業の継続的な要求を満たすべく進化し続ける複雑なソフトウェアを作り出すことができたプロジェクトは、1つだけだった。

私が見たプロジェクトの1つは、素早く最初の一歩を踏み出し、便利でシンプルなWeb

ベースの取引システムを完成させた。開発者たちは勘を頼りに作業を進めていたが、それが妨げになることはなかった。単純なソフトウェアは、設計にほとんど注意を払わなくても作成できるからだ。初期リリースが成功した結果、将来の開発にかけられる期待はとてつもなく高まった。私の元に第2バージョンに関する依頼が来たのは、その時である。つぶさに調べてわかったのだが、そのプロジェクトはドメインモデルを構築しておらず、共通の言語さえもなく、構造化されていない設計を抱え込んでいた。私のこの評価に対してプロジェクトリーダたちが同意しなかったので、私はその仕事を断ることにした。1年後、そのチームは泥沼にはまり込み、第2バージョンを仕上げられずにいた。技術の使い方も褒められたものではなかったが、それ以上に問題だったのは、ビジネスロジックである。第1リリースは早々に風化し、手のかかるレガシーとなってしまっていた。

より一層の複雑さに耐えられるようにするには、ドメインロジックの設計に対する、より真剣なアプローチが必要である。幸いなことに、私は以前、ドメイン設計を重視していたプロジェクトに関わったことがある。このプロジェクトは、少なくとも最初の例と同じくらい複雑なドメインを扱っており、やはり初めはそれなりに成功を収め、機関投資家向けのシンプルなアプリケーションを完成させた。しかし、こちらのプロジェクトでは、最初の成果が出た後も、開発が継続的に加速した。イテレーションのたびに、前のリリースの機能を統合して練り上げるための、刺激的で新しい選択肢が開かれていった。チームは、投資家の要求に対し、柔軟性と拡張能力をもって応えることができた。こうした上向きの軌道が生まれた理由は、鋭いドメインモデルに直接帰することができる。このドメインモデルは繰り返し改良され、コードにおいて表現されていたのだ。チームがドメインについての新たな洞察を得るにつれ、モデルは深みを増していった。コミュニケーションの質は、開発者同士の間だけでなく、開発者とドメインエキスパートの間でも向上した。また、設計のせいで、保守の負担が重くなることは一切なく、むしろ、修正や拡張が容易にできるようになった。

残念ながら、モデルを重要視するだけでは、プロジェクトがこうした好循環に到達することはない。私が過去に携わったもう1つのプロジェクトは、ドメインモデルに基づいたグローバルなエンタープライズシステムを構築するという高尚な志を持って始まったが、数年間失望し続けた結果、目標を低くしてありふれたかたちに落ち着いた。チームは適切なツールを持っており、ビジネスのことを十分に理解した上で、モデリングにも慎重に注意を払っていた。しかし、開発者の役割を誤って分割したために、モデリングが実装から切り離されてしまい、進行中の深い分析が設計に反映されなかった。いずれにしても、詳細なビジネスオブジェクトの設計は、そのオブジェクトを組み合わせて手の込んだアプリケーションにまとめられるほどには厳密ではなかったのだ。イテレーションを繰り返してもコードが改善されることはなかったが、これは、開発者間のスキルレベルが均一でな

かったことが原因である。モデルベースのオブジェクトを作成し、それを実用的な動作するソフトウェアとしても機能させるためには、スタイルやテクニックに関する非公式の体系が必要なのだが、そのことを彼らは全く意識していなかった。数か月が流れ、開発作業は複雑さの泥沼にはまり込み、チームはシステムに関する凝集したビジョンを失っていた。何年も努力した結果、このプロジェクトはそれ相応に役立つソフトウェアを作り出しはした。しかし、チームは、モデルに集中しなくなるのと同時に、初期の大志も失ってしまっていたのだ。

複雑さという課題

多くの事柄が原因で、プロジェクトは正しい道から逸れてしまう。官僚主義や不明確な目的、リソースの欠如などがその一例だ。しかし、ソフトウェアがどこまで複雑になれるかを決定する主な要因は、設計に対するアプローチである。手に負えないほど複雑になると、開発者はもはやソフトウェアを十分には理解できなくなり、変更や拡張を容易かつ安全に行えなくなる。だが、優れた設計であれば、そうした複雑な特徴を活用する機会を作り出せる。

設計に影響を与える要因には、技術的なものもある。ネットワークやデータベースをはじめとして、ソフトウェアの持つその他の技術的側面には、多大な労力が注ぎ込まれてきた。こうした問題の解決方法についての書籍も多数ある。大勢の開発者が自分のスキルを磨き、各技術の進歩に追随してきた。

しかし、多くのアプリケーションにおいて、最も重要な複雑さは、技術的なものではない。複雑なのはドメインそのもの、すなわち、ユーザの活動やビジネスなのである。ドメインの持つこの複雑さが設計で扱われないのであれば、基盤となる技術が適切に考えられていたとしても意味がない。設計を成功させるためには、ソフトウェアにおけるこの中心的な側面を、体系的に扱わなければならない。

本書が前提とするのは、次の2つである。

1. ほとんどのソフトウェアプロジェクトにおいて、一番の焦点は、ドメインとドメインロジックに合わせなければならない。
2. 複雑なドメインの設計は、モデルに基づかなければならない。

ドメイン駆動設計は、1つの考え方であると同時に、ひとまとまりの優先順位でもあり、入り組んだドメインを扱わなければならないソフトウェアプロジェクトを促進することを目指している。この目標を達成するため、本書では、設計のプラクティスやテクニックおよび原則を幅広く紹介する。

設計対開発プロセス

設計を扱う書籍もあれば、プロセスを扱う書籍もある。だが、互いにもう一方を参照することはめったにない。どちらの題材もそれ自体が複雑なのだ。本書では設計を扱うが、私は、設計とプロセスは切り離せないと思っている。設計上の概念はうまく実装に落とさなければならず、そうでなければ、無味乾燥な学術的議論になってしまう。

設計のテクニックを学ぶ時には、それがもたらす可能性に興奮するものだ。それから、実際のプロジェクトのひどい現実が降りかかってくる。設計についての新しい考え方を、使用すべき技術に適合させることができなくなるのだ。あるいは、時間の都合で設計の一側面をあきらめるべき時と、自分の主張を貫いて非の打ち所のない解決策を探し出すべき時を、見分けることができないのである。開発者たちは、設計についての原則を適用することについて、お互いに抽象的に話すことができるし、実際に話してもいるのだが、現実の物事をどうやるかについて話す方がはるかに自然である。そういうわけで、これは設計を扱う本ではあるが、必要に応じて人工的な境界を越え、プロセスにも言及するつもりだ。そうすることで、設計の原則がコンテキストの中に位置づけられるのだ。

本書は特定の方法論と結びついてはいないが、「アジャイル開発プロセス」という新しいグループを指向している。具体的には、プロジェクトに2つのプラクティスが定着していることを想定している。その2つのプラクティスは、本書のアプローチを適用する上での前提条件となっている。

1. **開発がイテレーティブである。**イテレーション開発は、ここ2、30年の間、支持され、実践され続けていて、アジャイル開発手法の礎にもなっている。アジャイル開発とエクストリームプログラミング（XP）の文献には優れた議論が多く、『Surviving Object-Oriented Projects』（Cockburn 1998）や、『Extreme Programming Explained』（Beck 1999）などが挙げられる。
2. **開発者とドメインエキスパートが密接に関わっている。**ドメイン駆動設計は、膨大な知識をかみ砕き、ドメインについての深い洞察と、集中すべき主要な概念を反映したモデルを構築する。これは、ドメインを理解している人と、ソフトウェアの構築方法を知っている人で行う共同作業だ。開発がイテレーティブであるため、この共同作業はプロジェクトの期間中、ずっと継続しなければならない。

エクストリームプログラミングは、Kent BeckやWard Cunninghamらによって考案されたもので（『Extreme Programming Explained』[Beck 2000]参照）、アジャイルプロセスの中で最も有名であり、私が一番多く手がけたものでもある。本書では、設計とプロセスの相互作用についての説明を具体的にするため、議論の基礎としてXPを使用する。だが、本書で紹介する原則は、XP以外のアジャイルプロセスにも容易に適合させることができる。

役に立たない固定化したドキュメントと強迫的なアップフロントの計画や設計によってプロジェクトに負担をかけるような手の込んだ開発方法論があるが、近年、そうした方法論に対する反乱が続いている。それに対して、XPなどのアジャイルプロセスは、変化と不確実さに対処する能力を重視するのだ。

エクストリームプログラミングも、設計上の意思決定が重要であることは認識しているが、アップフロントの設計には強く抵抗する。その代わりに、かなりの労力がつぎ込まれるのが、コミュニケーションであり、また、プロジェクトの対応能力を改善して、進路を素早く変更できるようにすることなのだ。その対応能力のおかげで、プロジェクトのどの段階でも、開発者は「機能し得る最もシンプルなもの（simplest thing that could work)」を使用できるし、それ以降は継続的にリファクタリングを行い、設計上の細かい改善を数多く重ねながら、最終的には顧客の真の要求に適した設計に到達することができる。

このミニマリズムは、過剰な設計の熱狂的な信者に対する対抗手段として、切望されてきたものである。多くのプロジェクトが、ほとんど価値のない厄介なドキュメントによって行き詰まってきた。彼らが思っていたのは、「分析麻痺（analysis paralysis)」である。設計が不完全であることをチームメンバが恐れるあまり、全く仕事が進まなくなってしまうのだ。何かが変わらなければならなかった。

残念なことに、このプロセスの考え方の中には誤解されかねないものもある。「最もシンプル（simplest)」の定義は人によってさまざまだ。継続的なリファクタリングは細かい再設計の積み重ねだが、開発者が確固とした設計原則を持っていないと、理解するのも変更するのも難しいコードベースを作り出してしまう。これでは、俊敏さ（アジャイルであること）とは正反対だ。思いがけない要求を恐れるあまり、必要以上の機能を持たせるオーバーエンジニアリングになってしまうことがしばしばあるが、オーバーエンジニアリングを回避しようとすれば、また別の恐怖が生じる。それは、設計について少しでも深く考えることに対する恐怖である。

実際、XPは、設計に関する鋭いセンスを持った開発者に最も適している。XPプロセスが想定しているのは、リファクタリングによって設計を改善できるということであり、それが頻繁かつ迅速に行われるということである。しかし、過去に行われた設計上の選択によっては、リファクタリングそのものが容易になることもあれば、困難になることもある。XPプロセスはチームのコミュニケーションを促進しようとするのだが、モデルや設計における選択によって、コミュニケーションは明確になることもあれば、混乱することもある。

本書は、設計と開発のプラクティスを絡み合わせ、ドメイン駆動設計とアジャイル開発が、どのように相互に補強し合うかについて説明する。ドメインモデリングに対する洗練されたアプローチが、アジャイル開発プロセスにおいて行われれば開発は加速する。プロセスとドメイン開発との相互関係があることによって、このアプローチは、「純粋」な設計

を単独で扱う、他のどのアプローチよりも実用的になっている。

本書の構成

この本は、次に示す4部に分かれている。

「第1部：ドメインモデルを機能させる」では、ドメイン駆動開発の基本的な目標を紹介する。この目標が、後で紹介されるプラクティスを実践する動機となる。ソフトウェア開発には非常に多くのアプローチがあるので、第1部では本書で使われる用語を定義し、コミュニケーションと設計を推進するためにドメインモデルを使用するということがどういう意味を持つのかを概観する。

「第2部：モデル駆動設計の構成要素」では、オブジェクト指向ドメインモデリングにおけるベストプラクティスのコアを要約して、基本的な構成要素の集まりを作りあげる。ここでは、モデルと動作する実用的なソフトウェアとの間にある溝を埋めることに重点を置く。ここで述べる標準的なパターンを共有することによって、設計に秩序がもたらされる。また、チームメンバは互いの仕事をより容易に理解するようになる。標準的なパターンを使用することで、共通の言語に用語法が提供され、この共通の言語を使って、チームメンバ全員が、モデルと設計上の意思決定について議論できる。

しかし、第2部で一番重要なのは、モデルと実装を互いに整合させ、片方がもう一方の有効性を補強する状態を維持するような意思決定に集中することである。こうした整合性を維持するためには、個々の要素の詳細に注意する必要がある。こうした小規模の要素をていねいに作り上げれば、第3部と第4部で示すモデリングアプローチを適用するにあたって、開発者は、その基となる安定した土台を与えられるのだ。

「第3部：より深い洞察へ向かうリファクタリング」では、個別の構成要素を超えて、利益をもたらす実用的なモデルを組み立てるという課題へ進む。深遠な設計原則に直接飛び込むのではなく、ここでは発見のプロセスを重視する。価値あるモデルはすぐに姿を現さない。ドメインに関する深い理解を必要とするのだ。その理解を得るためには、没頭して、おそらくは素朴なモデルに基づく初期の設計を実装し、後でそれを繰り返し変化させなければならない。チームが洞察を得るたび、モデルはより豊かになった知識を明らかにするように変形され、コードはリファクタリングされて深められたモデルを反映するようになり、その潜在能力がアプリケーションで使用できるようになる。こうしていると、時折、この地道な作業がさらに深いモデルへとブレイクスルーする機会につながり、その後には、奥深い設計の変化があふれ出すのだ。

本来、探究に終わりはないが、手当たり次第に行う必要はない。第3部では、探究の途上で選択の指針となり得るモデリングの原則と、探究を方向づけるのに役立つテ

クニックを掘り下げる。

「第4部：戦略的設計」では、複雑なシステムや大企業、あるいは外部システムおよびレガシーシステムとの相互作用において発生する状況に対処する。ここで検討するのは、全体としてのシステムに対して適用される3つ組の原理、すなわち、コンテキスト、蒸留、および大規模な構造である。戦略的設計において意思決定を行うのはチームであり、さらに、こうした意思決定は複数のチーム間で行われることもある。戦略的設計によって、第1部で示した目標がより大きな規模で実現できるようになる。ここで大規模と言っているのは、無秩序に広がるエンタープライズ規模のネットワークに組み込まれる、巨大なシステムやアプリケーションのことである。

本書を通じて議論に用いるのは、過度に単純化した「おもちゃのような」問題ではなく、実際のプロジェクトからとった現実の事例である。

本書の大半は「パターン」の集まりとして書かれている。読者はこのパターンという道具については気にすることなく内容を理解できるはずだが、そのスタイルと形式に興味があるなら、付録を参照するとよいだろう。

補足資料は、http://domaindrivendesign.orgにある。ここには、追加のサンプルコードやコミュニティでの議論も含まれている。

本書が対象とする読者

本書は、主としてオブジェクト指向ソフトウェアの開発者向けに書かれている。ソフトウェアプロジェクトチームのほとんどのメンバは、本書のどこかからは恩恵を受けられるだろう。本書の意義を最も強く感じるのは、現在プロジェクトに関わっていて、今まさにこうしたことの一部を試そうとしている人と、そうしたプロジェクトに関する深い経験をすでに備えている人であろう。

本書から何かを得るには、オブジェクト指向モデリングに関する若干の知識が必要である。例には、UML図とJavaコードが含まれているので、これらの言語に対する基本レベルの読解力は重要だが、どちらも詳細まで習得する必要はない。エクストリームプログラミングを知っていれば、開発プロセスに関する議論を全体的に見通せるようになるが、背景となる知識がなくても、本書の内容は理解できるはずである。

すでにオブジェクト指向設計について何らかの知識を持っていて、ソフトウェア設計の文献を1、2冊は読んだことがある中級のソフトウェア開発者にとって、本書は、オブジェクトモデリングがソフトウェアプロジェクトにおける現実生活にどう当てはまるかについて、理論と実践の隙間を埋め、全体を見渡す視点を提供するものとなる。この本は、中級

の開発者が、洗練されたモデリングと設計スキルを実際の問題に適用することを学ぶ上で助けになるだろう。

上級のソフトウェア開発者や、熟練したソフトウェア開発者は、本書の提示する、ドメインを扱うための包括的なフレームワークに関心を持つだろう。設計に対するこの体系的アプローチにより、技術リーダは自分のチームを導いて、本書の示す道を進ませることができるようになるだろう。また、本書を通じて使用されている一貫した用語法により、上級開発者は同僚とコミュニケーションできるようになるだろう。

本書は物語風になっていて、最初から始めて最後まで読んでもよいし、あるいは任意の章の始めから読んでもよい。読者にはそれぞれ別々の背景があるから、別々のやり方で読み進めたいと思うかもしれない。しかし、私は、全員が第1部の導入と第1章から読み始めることを推奨する。その先で中核となる章は、2、3、9、14章であろう。テーマについてすでにある程度把握していて、飛ばし読みをしたい人は、冒頭と太字になった本文を読めば、要点を拾い上げられるはずである。さらに進んだ読者なら、第1部と第2部はざっと目を通す程度でよく、第3部と第4部に強く興味を引かれるだろう。

このようなコアとなる読者層に加え、アナリストや比較的技術寄りのプロジェクトマネージャも、本書を読むことで得るものがあるだろう。アナリストは、モデルと設計とのつながりを活用することで、アジャイルプロジェクトのコンテキストにおいてより効果的に貢献できる。また、戦略的設計の原則を使うことで、これまで以上に自分たちの仕事に集中して、それをまとめ上げられるかもしれない。

本書が強調しているのは、チームをより効果的にし、ビジネスエキスパートとユーザにとって意味のあるソフトウェアの設計に集中させるということだが、プロジェクトマネージャなら、間違いなくこの点に興味を持つだろう。しかも、戦略的設計における意思決定は、チーム編成や作業スタイルと関係するから、そうした設計上の意思決定は、必然的にプロジェクトのリーダシップに関係し、プロジェクトの軌道に大きな影響を与えるのである。

ドメイン駆動チーム

ドメイン駆動設計を理解している個々の開発者は、価値のある設計テクニックと全体を見る視点を本書から得るだろう。しかし、最大の収穫が得られるのは、チームが一致団結してドメイン駆動設計アプローチを適用し、ドメインモデルをプロジェクトで交わされる会話の中心に持ってきた時である。そうすることで、チームメンバは言語を共有し、コミュニケーションを豊かにして、コミュニケーションがソフトウェアとつながった状態を保てるようになる。チームはモデルと一致した明快な実装を作り出し、アプリケーション

開発に力を与える。また、このチームは、別々のチームの設計作業が互いにどう関係するかを示す地図を共有し、組織にとって最も特徴的で価値のある機能に対して、体系的に注意を向けるようになる。

ドメイン駆動設計は困難な技術的課題だが、たいていのソフトウェアプロジェクトがまさにレガシーへと風化し始める時に、チャンスを大きく広げることができるものなのだ。

謝　辞

本書には、何らかのかたちで4年以上を費やしたが、その間、多くの方々に助けていただき、また支えていただいた。

原稿を読み、感想を述べてくれた方々に感謝する。そうしたフィードバックがなければ、本書は到底完成しなかっただろう。惜しみない注意を払ってレビューしてくれた方も数人いる。Russ Rufer と Tracy Bialek が率いる The Silicon Valley Patterns Group は、本書の最初の全草稿を精査するのに7週間も費やしてくれた。Ralph Johnson が率いる The University of Illinois reading group も、数週間かけて、その後の草稿をレビューしてくれている。このグループ内で交わされる、長期にわたる活発な議論に耳を傾けるのは、実に有効だった。Kyle Brown と Martin Fowler は、詳細なフィードバックや、貴重な洞察、さらに、かけがえのない精神的サポートを（魚の番をしながら）与えてくれた。Ward Cunningham のコメントによって、私は重大な欠点を補強することができた。Alistair Cockburn は、早い時期から私を励まし、出版に至るまでの過程を手助けしてくれた。これは Hilary Evans も同じである。David Siegel と Eugene Wallingford のおかげで、より専門的な部分で、私はまごつかずに済んだ。Vibhu Mohindra と Vladimir Gitlevich は、すべてのサンプルコードを丹念にチェックしてくれた。

Rob Mee は、本書にあるような内容に関して私が初期にまとめた調査資料を読み、この設計スタイルを伝えるための何らかの方法を私が模索していた時にも、一緒にアイデアのブレインストーミングをしてくれた。それからずっと後の草稿も、私と一緒に熟読してくれている。

Josh Kerievsky は、本書の執筆における、大きな転機の1つをもたらした。彼からは「Alexander 流」のパターンフォーマットを試すことを勧められたのだが、これは本書の構成の中心となった。彼はまた、現在では第2部にある内容の一部を、初めて一貫した形式にまとめる手助けもしてくれた。これは 1999 年の PLoP 会議に先立つ、徹底的な「シェパーディング」過程でのことである。それが本書の残りのほとんどを形成する源となった。

また、Awad Faddoul にも感謝している。何百時間もの間、彼の素晴らしいカフェに居座って、私は執筆を行った。ウィンドサーフィンも大いに楽しんだが、そうやってこもっ

ていたおかげで、完成までこぎつけることができたのだ。

そして、Martine Jousset、Richard Paselk と Ross Venables にも非常に感謝している。彼らは、主要な概念のいくつかを説明するための美しい写真を作成してくれた（写真のクレジットを参照）。

本書を思いつく以前に、私はソフトウェア開発に関する、自分の見方と理解を形成しなければならなかった。こうした形成は、優れた方々の惜しみない助けに負うところが大きい。私にとって彼らは、非公式のメンターであると同時に、友人でもある。David Siegel、Eric Gold、Iseult White は、それぞれ別々のかたちで、ソフトウェア設計に関する私の考えを発展させるのを助けてくれた。また、Bruce Gordon、Richard Freyberg、そして Judith Segal 博士にも、全く別の方法で、うまくいっているプロジェクトで作業するにあたって、どうすればよいかを模索している時に手助けをしてもらった。

私自身の考えは、当時の風潮の中に存在した考え方の体系の中から自然に生じてきたものだ。それに寄与したものの一部は本文で明らかにし、可能な場合には参照する。言及していないものは、あまりにも基本的なので、影響を受けたということすら自分では認識していない。

修士論文の指導教官だった Bala Subramanium 博士は、私を数学的モデリングへと向かわせ、それを私たちは化学反応速度論に適用した。これもモデリングには違いなく、この研究は本書へと通じる道筋の一部になった。

さらにさかのぼると、私の考え方を形作ったのは両親の Carol Evans と Gary Evans である。また、数人の特別な先生が、私の興味を呼び起こし、基礎を築くのを手伝ってくれた。中でも Dale Currier（高校の数学の先生）、Mary Brown（高校の英作文の先生）、そして Josephine McGlamery（6年生の時の科学の先生）の名前を挙げたい。

最後に、友人と家族、それから Fernando De Leon に、ずっと私を励まし続けてくれたことを感謝したい。

翻訳者謝辞

本書を手に取って下さり、どうもありがとうございます。この本は、ソフトウェアの設計を扱ったものです。それと同時に、開発プロセスにおいて言葉の果たす役割や、人間がものごとをどう考えているのかということについての、深い洞察に裏打ちされたものでもあります。DDDを読んで、私のソフトウェア開発に対する考え方は大きく変わりました。深みがあることを知り、より多くを学びたいと思うようになりました。この素晴しい本を書いてくれた、著者のEric Evans氏に感謝します。

翻訳プロジェクトを進めて下さり、途中参加した私を受け入れて下さった、監訳者の今関剛さん、翻訳者の牧野祐子さん、そして翔泳社の古田島義和さんに感謝します。この方々がいなければ、DDDの日本語化が実現することはなかったでしょう。また、この日本語版を美しく組版して下さり、相当な量の校正にていねいに対応して下さったエスアイビー・アクセスの富澤昇さんに感謝します。

私が原書と深く向き合うことができたのは、2008年から2009年にかけて開催されたDDD読書会（JavaEE勉強会）でのことでした。あの場で行われた濃密な議論は、私は今でも生き生きと覚えていて、今回の翻訳作業をする上でも助けになりました。JavaEE勉強会の主催者である角田直行さんと、DDD読書会に参加していたすべての方々に感謝します。

この本をできることならば翻訳したいと、もう何年も思っていました。ここに、長年の夢が叶ったことになります。私が翻訳プロジェクトに関わるきっかけを作って下さり、常に見守って下さった和田卓人さんに感謝します。また、いつも支えて下さった、翔泳社の岩切晃子さんに感謝します。

翻訳作業にあたっては、多くの方々に助けて頂きました。プロジェクトに参加した当初から、私の原稿を査読し、数多くの鋭い指摘を下さった渡邉健太郎さん、佐藤匡剛さんに感謝します。お2人に背中を任せることで、私は安心して翻訳作業を進めることができました。また、佐藤さんが執筆された「Domain-Driven Designのエッセンス（http://www.ogis-ri.co.jp/otc/hiroba/technical/DDDEssence/chap1.html）」は、パターン名を翻訳する際に大いに参考にさせて頂きました。限られた時間の中で素晴しい精読をして下さった山野裕司さんに感謝します。また、きわめて短い時間の中で、すべての章を読んで下さった、高江洲睦

さん、瀬賀直樹さんに感謝します。そして、DDDの実践的な知識に基づいて査読をして下さった、じゅんいち☆かとうさん、都元ダイスケさんに感謝します。

最後に、妻に。彼女はクリスマスも正月もなく翻訳作業に明け暮れる私を、文句1つ言わずに支えてくれました。さらに、ソフトウェアとは全く関係のない仕事をしているにも関わらず、全文をていねいに読み、的確な指摘で原稿を真っ赤に染めてくれました。本当にありがとう。こうした方々のおかげで、この翻訳はずいぶんと読みやすくなりました。もちろん、誤訳や意味のわかりにくいところが残っていれば、それは私の責任です。

さらに、ここにお名前を挙げることのできなかった多くの方々にも、精神的に支えて頂きました。深く感謝します。

読者の方々にとって、本書が、いつも手元に置いておきたい宝物になりますように。

2011年3月

和智 右桂

目　次

日本語版への序文 iv
日本語版推薦文 vi
序　文 x
まえがき xii
謝　辞 xxi
翻訳者謝辞 xxiii

第1部　ドメインモデルを機能させる 1

ドメイン駆動設計におけるモデルの有用性 3
ソフトウェアの核心 4

第1章　知識をかみ砕く 7

効果的なモデリングの要素 13
知識のかみ砕き 14
継続的学習 16
知識豊富な設計 17
例1.1——隠された概念を引き出す 18
深いモデル 20

第2章　コミュニケーションと言語の使い方 23

ユビキタス言語（UBIQUITOUS LANGUAGE） 24
例2.1——貨物輸送プログラムを完成させる 27
声に出してモデリングする 30
1つのチームに1つの言語 32
ドキュメントと図 34
書かれた設計ドキュメント 36

実行可能な基盤 39
説明のためのモデル 39
例 2.2——輸送業務と経路 40

第 3 章 モデルと実装を結びつける 43
モデル駆動設計（MODEL-DRIVEN DESIGN） 45
モデリングパラダイムとツールによるサポート 48
例 3.1——手続き型からモデル駆動へ 50
骨格を見せる：なぜモデルがユーザにとって重要なのか？ 55
実践的モデラ（HANDS ON MODELERS） 57

第 2 部 モデル駆動設計の構成要素 61

第 4 章 ドメインを隔離する 65
レイヤ化アーキテクチャ（LAYERED ARCHITECTURE） 66
例 4.1——オンラインバンキングの機能をレイヤに分割する 69
レイヤを関係づける 70
アーキテクチャフレームワーク 72
ドメイン層はモデルが息づく場所 73
利口な UI「アンチパターン」（SMART UI "ANTI-PATTERN"） 74
その他の隔離 77

第 5 章 ソフトウェアで表現されたモデル 79
関 連 80
例 5.1——証券取引口座における関連 83
エンティティ（ENTITIES）（別名 参照オブジェクト（REFERENCE OBJECTS）） 87
エンティティをモデル化する 91
同一性のための操作を設計する 92
値オブジェクト（VALUE OBJECTS） 95
値オブジェクトを設計する 98
例 5.2——値オブジェクトを使ってデータベースをチューニングする 101
値オブジェクトを含む関連を設計する 101
サービス（SERVICES） 103
サービスと隔離されたドメイン層 105
粒 度 106
サービスへのアクセス 107

モジュール（MODULES）（別名　パッケージ（PACKAGES））……108
アジャイルモジュール……110
例5.3——Javaにおけるパッケージのコーディング規約……110
インフラストラクチャ駆動パッケージングの落とし穴……111
モデリングパラダイム……114
なぜオブジェクトパラダイムが主流なのか？……114
オブジェクトの世界におけるオブジェクトではないもの……117
パラダイムを混在させる際にはモデル駆動設計に忠実であること……118

第6章　ドメインオブジェクトのライフサイクル　121

集約（AGGREGATES）……123
例6.1——購入注文の整合性……128
ファクトリ（FACTORIES）……134
ファクトリとその場所を選択する……137
コンストラクタがあればよい場合……139
インタフェースを設計する……141
不変条件のロジックはどこへ置くべきか？……142
エンティティファクトリ対値オブジェクトファクトリ……142
格納したオブジェクトを再構成する……143
リポジトリ（REPOSITORIES）……146
リポジトリに対して問い合わせる……151
クライアントのコードはリポジトリの実装を無視するが、開発者はそうではない……153
リポジトリを実装する……154
フレームワークの範囲内で作業する……156
ファクトリとの関係……157
関係データベースに合わせてオブジェクトを設計する……159

第7章　言語を使用する：応用例　163

貨物輸送システムを導入する……163
ドメインを隔離する：アプリケーションの導入……166
エンティティと値オブジェクトを区別する……166
役割とその他の属性……168
輸送ドメインの関連を設計する……168
集約の境界……170
リポジトリを選択する……171
シナリオをウォークスルーする……173
サンプルアプリケーションの機能：貨物の荷出し地を変更する……173

サンプルアプリケーションの機能：リピータへの対応 173
オブジェクトの生成 174
貨物用のファクトリとコンストラクタ 174
荷役イベントを追加する 175
リファクタリングのために立ち止まる：貨物集約についてのもう1つの設計 177
輸送モデルにおけるモジュール 179
新機能を導入する：配分チェック 182
2つのシステムを接続する 183
モデルを強化する：ビジネスのセグメント化 183
パフォーマンスチューニング 186
最後に 186

第3部 より深い洞察へ向かうリファクタリング 189

リファクタリングのレベル 190
深いモデル 191
深いモデル／しなやかな設計 193
発見のプロセス 193

第8章 ブレイクスルー 195

ブレイクスルーの話 196
悪くないモデルなのだが… 197
ブレイクスルー 198
さらに深いモデル 201
冷静な意思決定 203
結 末 204
好 機 204
基本への集中 204
エピローグ：新しい洞察の連鎖 205

第9章 暗黙的な概念を明示的にする 207

概念を掘り出す 207
言葉に耳を傾ける 208
例9.1——輸送モデルに欠けている概念を聞き分ける 209
ぎこちなさを精査する 212
例9.2——利息を得る 難しい方法 213
矛盾について熟考する 218

文献を読む 218
例9.3——利息を得る　文献を用いた場合 219
何度でも挑戦すること 221
それほど明白でない概念をモデル化する方法 221
明示的な制約 221
例9.4——再考：オーバーブッキングポリシー 224
ドメインオブジェクトとしてのプロセス 224
仕様（SPECIFICATION） 226
仕様の適用と実装 229
例9.5——化学製品倉庫での格納 238
例9.6——倉庫内格納サービスの、実際に動作するプロトタイプ 242

第10章　しなやかな設計　247

意図の明白なインタフェース（INTENTION-REVEALING INTERFACES） 251
例10.1——リファクタリング：塗料混合アプリケーション 252
副作用のない関数（SIDE-EFFECT-FREE-FUNCTIONS） 255
例10.2——リファクタリング：塗料混合アプリケーション再考 257
表明（ASSERTIONS） 261
例10.3——塗料の混合に戻る 262
概念の輪郭（CONCEPTUAL CONTOURS） 266
例10.4——発生の輪郭 268
独立したクラス（STANDALONE CLASSES） 271
閉じた操作（CLOSURE OF OPERATIONS） 273
例10.5——コレクションから選択する 274
宣言的な設計 275
ドメイン特化言語 277
設計の宣言的スタイル 278
宣言的スタイルで仕様を拡張する 278
例10.6——コンポジット仕様を実装する他の方法 283
攻める角度 287
サブドメインを切り取る 287
可能な場合には、確立された形式主義を活用する 288
例10.7——パターンを統合する：シェア算 288

第11章　アナリシスパターンを適用する　299

例11.1——利息を得る　勘定を用いた場合 301
例11.1（続き）——夜間バッチについての洞察 307

アナリシスパターンは活用すべき知識である 312

第12章 デザインパターンをモデルに関係づける 315

ストラテジー（STRATEGY）（別名 ポリシー（POLICY）） 317

例 12.1——経路検索ポリシー 318

コンポジット（COMPOSITE） 321

例 12.2——経路で構成された輸送経路 323

なぜ、フライウェイトではないのか？ 328

第13章 より深い洞察へ向かうリファクタリング 329

開 始 329

探究チーム 330

先達の技 331

開発者のための設計 332

タイミング 332

好機となる危機 333

第4部 戦略的設計 335

第14章 モデルの整合性を維持する 339

境界づけられたコンテキスト（BOUNDED CONTEXT） 343

例 14.1——予約コンテキスト 345

境界づけられたコンテキスト内での分派を認識する 347

継続的な統合（CONTINUOUS INTEGRATION） 349

コンテキストマップ（CONTEXT MAP） 352

例 14.2——輸送アプリケーションにおける2つのコンテキスト 355

コンテキストの境界で行うテスト 360

コンテキストマップを構成してドキュメント化する 360

境界づけられたコンテキスト間の関係 361

共有カーネル（SHARED KERNEL） 363

顧客／供給者の開発チーム（CUSTOMER/SUPPLIER DEVELOPMENT TEAMS） 365

例 14.3——収益分析と予約 368

順応者（CONFORMIST） 370

腐敗防止層（ANTICORRUPTION LAYER） 373

腐敗防止層のインタフェースを設計する 375

腐敗防止層を実装する 375

例 14.4——レガシー予約アプリケーション 378

訓　話 378
別々の道（SEPARATE WAYS） 380
例 14.5——保険プロジェクトの縮小化 381
公開ホストサービス（OPEN HOST SERVICE） 383
公表された言語（PUBLISHED LANGUAGE） 384
例 14.6——化学のための公表された言語 386
象のモデルを統一する 387
モデルコンテキスト戦略を選択する 391
チームでの意思決定と、より上層での意思決定 391
コンテキストに自らの身を置く 391
境界を変換する 392
変更できないものを受け入れる：外部システムの輪郭を描く 392
外部システムとの関係 393
設計中のシステム 394
別のモデルで特殊な要求を満たす 394
デプロイ 396
トレードオフ 396
すでにプロジェクトが進行中の場合 397
変　換 398
コンテキストをマージする：別々の道 → 共有カーネル 398
コンテキストをマージする：共有カーネル → 継続的な統合 400
レガシーシステムを段階的に廃止する 401
公開ホストサービス → 公表された言語 402

第 15 章　蒸　留　405

コアドメイン（CORE DOMAIN） 408
コアを選択する 410
誰がこの作業をやるのか？ 411
蒸留の拡大 412
汎用サブドメイン（GENERIC SUBDOMAINS） 414
例 15.1——2 つのタイムゾーンの物語 418
汎用とは再利用可能という意味ではない 420
プロジェクトのリスク管理 420
ドメインビジョン声明文（DOMAIN VISION STATEMENT） 422
強調されたコア（HIGHLIGHTED CORE） 424
蒸留ドキュメント 425

コアにフラグを立てる 425
プロセスツールとしての蒸留ドキュメント 427
凝集されたメカニズム（COHESIVE MECHANISMS） 429
例 15.2——組織図におけるメカニズム 430
汎用サブドメイン対凝集されたメカニズム 431
メカニズムがコアドメインの一部である場合 431
例 15.3——一巡：組織図がメカニズムを再び吸収する 432
蒸留して宣言的スタイルにする 432
隔離されたコア（SEGREGATED CORE） 434
隔離されたコアを作成するコスト 435
チームの意思決定を進化させる 435
例 15.4——貨物輸送モデルのコアを隔離する 436
抽象化されたコア（ABSTRACT CORE） 441
深いモデルの蒸留 443
リファクタリングの対象を選ぶ 443

第 16 章 大規模な構造 445

進化する秩序（EVOLVING ORDER） 450
システムのメタファ（SYSTEM METAPHOR） 453
「素朴なメタファ」とそれを必要としない理由 454
責務のレイヤ（RESPONSIBILITY LAYERS） 456
例 16.1——深く掘り下げる：輸送システムをレイヤ化する 458
適切なレイヤを選択する 468
知識レベル（KNOWLEDGE LEVEL） 472
例 16.2——従業員の給料と年金（1） 473
例 16.3——従業員の給料と年金（2）知識レベル 479
着脱可能コンポーネントのフレームワーク（PLUGGABLE COMPONENT FRAMEWORK） 482
例 16.4——SEMATECH CIM フレームワーク 483
構造による制約をどの程度厳しくするべきか？ 486
ふさわしい構造へのリファクタリング 488
ミニマリズム 488
コミュニケーションと自己規律 488
再構成によってしなやかな設計がもたらされる 489
蒸留によって負荷が軽減される 489

第17章　戦略をまとめ上げる　491

大規模な構造と境界づけられたコンテキストを組み合わせる 492
大規模な構造と蒸留を組み合わせる 496
まず評価する 498
誰が戦略を策定するのか？ 498
アプリケーション開発から現れる構造 499
顧客に焦点を合わせたアーキテクチャチーム 499
戦略的設計上の意思決定を行うために欠かせない6つのこと 500
同じことが技術的なフレームワークにも当てはまる 503
マスタプランに注意すること 504

結　論　505

エピローグ 506
展　望 511

付録：本書におけるパターンの使い方　513

用語解説　517

参考文献　521

掲載写真クレジット　523

索　引　525

監訳者あとがき　537

第1部　ドメインモデルを機能させる

第1章　知識をかみ砕く 7
第2章　コミュニケーションと言語の使い方 23
第3章　モデルと実装を結びつける 43



この18世紀の中国の地図は、世界全体を表している。中央に配置され、スペースの大半を占めているのが中国で、その周りを囲んでいるのが、おざなりに表現された他の国々である。世界についてのこのモデルは、国内に意識が向いていた当時の中国社会にふさわしいものだった。この地図が示す世界観は、外国人に対応するには役に立たなかったに違いない。もちろん、現代の中国には全く無意味だろう。地図はモデルである。そしてどんなモデルも、現実の何らかの側面や興味の対象となる概念を表している。モデルとは簡素化である。つまり、当面の問題を解決する上で関連する側面を抽象化し、それ以外の詳細を無視することによって行われた、現実に対する1つの解釈なのだ。

すべてのソフトウェアプログラムは、それを使用するユーザの何らかの活動や関心と関係がある。ユーザがプログラムを適用するこの対象領域が、ソフトウェアのドメインである。ドメインの中には、物理的な世界を含んでいるものもある。例えば、航空会社の予約プログラムのドメインは、実際の航空機に搭乗する現実の人々を含んでいる。一方、実体を含まないドメインもある。会計プログラムのドメインは金銭と財務である。ソフトウェアのドメインは、コンピュータとほとんど関係ないのが普通だが、例外もある。ソースコード管理システムのドメインは、ソフトウェア開発そのものである。

ユーザの活動において有効利用されるソフトウェアを作成するために、開発チームはその活動に関係する体系化された知識を身につけていかなければならない。ここで要求され

る知識の幅広さに手こずるかもしれないし、情報の量と複雑さに圧倒されるかもしれない。モデルはこの重荷と格闘するためのツールなのだ。モデルとは、選び抜かれてシンプルにされ、意図的に組み立てられた知識の表現形式である。適切なモデルは情報の持つ意味を明らかにし、その情報を問題に集中させる。

ドメインモデルとは特定の図ではなく、図が伝えようとしている考え方である。これはドメインエキスパートの頭の中にある単なる知識ではなく、**その知識が厳密に構成され、選び抜かれて抽象化されたもの**なのだ。図を用いることによって、モデルを表現し、伝えることができる。これは、慎重に作成されたコードでもできることだし、自然言語で書かれた文章でもできることだ。

ドメインモデリングとは、モデルをできるだけ「写実的に」作成することではない。目に見える実世界の事柄に関するドメインにおいてさえ、モデルは人為的な成果物にすぎない。また、ドメインモデリングは、必要な結果を出力するソフトウェアの仕組みを構築するだけでもない。むしろ映画制作に近く、ある目的に従って、現実の概要を表現しているのだ。ドキュメンタリ映画であっても、未編集の実生活を見せたりはしない。映画の制作者は、体験したことの側面をいくつか選び出し、それらを独特な手法で提示することで、ストーリーを伝えたり意見を主張したりする。それと同じようにドメインモデラも、実用のために特定のモデルを選択するのである。

ドメイン駆動設計におけるモデルの有用性

ドメイン駆動設計では、次に示すモデルの3つの基本的用法によって、どのモデルを選択するかが決定される。

1. **モデルと設計の核心が相互に形成し合う。**モデルと実装が密接に結びつくことによって、モデルはドメインと深く関連したものになり、モデルに対する分析が、最終成果物である実際に動くプログラムに適用されることが保証される。このモデルと実装の結びつきは、保守や継続的開発においても有効だが、それは、モデルの理解に基づいてコードを解釈できるからである（第3章を参照）。
2. **モデルは、チームメンバ全員が使用する言語の基盤である。**モデルと実装が結びついているので、開発者はプログラムについて話す際にこの言語を使える。通訳せずにドメインエキスパートとコミュニケーションができるのだ。しかも、言語はモデルに基づいているから、生まれながらの言語能力を使ってモデルそのものを改良できる（第2章を参照）。
3. **モデルとは、蒸留された知識である。**モデルは、ドメインの知識を構成して最も関心のある要素を区別するための、チーム内で取り決めた方法である。我々が用語を選択

し、概念を分解して関連づける際に、ドメインについてどう考えることにしたかということが、1つのモデルによってとらえられる。共有された言語があれば、開発者とドメインエキスパートが、情報をモデルという形式に苦労して変換する際に、効果的に共同作業ができるようになる。モデルと実装が結びついていることで、ソフトウェアの初期バージョンで得た経験が、フィードバックとしてモデリングプロセスに適用できるようになる（第1章を参照）。

この後の3つの章では、モデルによるこれらの貢献の意味と価値、そしてこれらがどう絡み合っているかについて、順に見ていく。こうしたやり方でモデルを使うことにより、豊富な機能を持ったソフトウェアの開発が支えられる。このようなソフトウェアは、その場しのぎの開発をしようものなら、莫大な投資が必要となってしまうのだ。

ソフトウェアの核心

ソフトウェアの核心は、ドメインに関係した問題をユーザのために解決する能力である。それ以外の特徴はいずれも、どれほど重要だとしても、この基本的な目的を支えるにすぎない。ドメインが複雑だと、ドメインに関係した問題を解決するのは難しい作業になり、熟練した有能な人々が集中して取り組まなければならなくなる。開発者は、ビジネスの知識を積み重ねるべく、ドメインに没頭しなければならない。モデリングスキルを研ぎ澄まし、ドメイン設計に精通することが要求される。

しかし、たいていのソフトウェアプロジェクトでは、ドメインに関係した問題の解決が重要視されない。有能な開発者のほとんどは、自分の手がける特定のドメインについて勉強する気があまりなく、ドメインモデリングのスキルを真剣に伸ばすつもりはさらにない。技術指向の開発者は、自分の専門スキルを発揮できる、定量化可能な問題を楽しむものだ。ドメインに関する作業は面倒な上に、入り組んだ新しい知識を大量に必要とするが、そうした知識は開発者にとって、コンピュータ科学者としての能力を向上させるものではないように見えるのだ。

その代わり、技術的な才能のある開発者は複雑なフレームワークに取り組み、技術によってドメインの問題を解決しようとする。ドメインを学んだりモデリングしたりするのは他人に任せてしまうのだ。しかし、ソフトウェアの核心にある複雑さには、真正面から立ち向かわなければならない。そうしなければ、的外れなソフトウェアを作ってしまう危険があるのだ。

テレビ番組のインタビューで、コメディアンのJohn Cleeseが、『モンティ・パイソン・アンド・ホーリー・グレイル』の撮影中に起こった出来事について話をしていた。あるシー

ンを繰り返し撮影していたが、どうもおもしろくない。とうとう休憩を取って、仲間のコメディアンであるMichael Palin（そのシーンに登場するもう1人の俳優）に相談し、多少変更することにした。もう1回撮り直したら、おもしろいものになったので、その日はそれで終わりにした。

翌朝、Cleeseは、映像編集者が前日分をまとめたラフカットを見ていた。例の苦労したシーンが来たところで、全然おもしろくないことに気がついた。1つ前に撮影した方が使用されていたからだ。

なぜ指示通りに最後の撮影分を使用しなかったのか、Cleeseは編集者に尋ねた。すると、「使えませんよ。誰かがスクリーンに入って来たので」との答えが返ってきた。Cleeseはもう一度シーンをじっくり見て、さらに見直した。それでも何がまずいのかわからなかった。結局、編集者はフィルムを止め、コートの袖がスクリーンの端に一瞬見えることを指摘した。

この映像編集者は、自分の専門的な仕事を正確に遂行することしか考えていなかった。彼は、他の編集者がこの映画を見たら、技術が完璧かどうかで自分の仕事を評価するだろうと気にしていたのである。その過程で、このシーンの核心は失われてしまった（CBSで2001年9月に放送された『The Late Late Show with Craig Kilborn』）。

幸い、コメディのなんたるかを理解している監督によっておもしろいシーンは復活した。これと全く同じように、チームのリーダがドメインの中心に何があるのかを理解していれば、深い理解を反映したモデルの開発が混乱して、進むべき道を見失った場合でも、ソフトウェアプロジェクトを元の進路に戻すことができる。

本書では、ドメインの開発には非常に高度な設計スキルを養う機会があふれていることを示していく。ほとんどのソフトウェアドメインにおけるこの混沌とした状況は、実は、興味深い技術的な課題なのである。実際、多くの科学分野では、「複雑性」が最も刺激的な最新テーマの1つであり、研究者たちは現実世界の混沌に取り組もうとしている。ソフトウェア開発者も、まだ形式化されたことのない入り組んだドメインに直面した際には、同じ立場に立つことになる。その複雑さを一刀両断にする明快なモデルを作り出すのは、刺激的な作業だ。

開発者が、洞察を追求して役立つモデルを生み出すために採用できる、体系的な考え方がある。また、無秩序なソフトウェアアプリケーションに秩序をもたらすことのできる設計技術もある。これらのスキルを養うことで、最初はあまりなじみのないドメインにおいてさえも、開発者ははるかに価値のある存在となるのである。

第1章

知識をかみ砕く

数年前、私はプリント基板（PCB: Printed-Circuit Board）設計用の専門ソフトウェアツールを設計することになった。しかし、1つ問題があった。私は電子機器について何も知らなかったのだ。もちろん、PCB設計者から話は聞けたが、彼らの話にはものの3分で頭がクラクラしてくるのが常だった。このソフトウェアを作成するのに十分なくらい電子機器について理解するには、どうすればよいだろう？ 納期が来る前に電子技術者になることなど、できるはずもないのだ！

ソフトウェアがすべきことを、PCB設計者に正確に説明してもらおうとしたが、これがまずかった。彼らは素晴らしい回路設計者なのだが、彼らの考えているソフトウェアはたいてい、ASCIIファイルを読み込み、それをソートした上で注釈をつけ、そして帳票を作成する、というものだった。これでは、彼らの求める生産性の飛躍的向上につながらないことは明白だった。

最初の数回の打ち合わせには失望させられたが、彼らが必要としていた帳票にかすかな希望を託すことができた。帳票にはいつでも「ネット」と、それに関するさまざまな詳細を出力することになっていた。このドメインにおけるネットとは、要するに、PCB上で任意の数のコンポーネントを接続することができる導体のことで、接続されたものすべてに電気信号を伝えるのだ。我々はドメインモデルの最初の要素を手に入れたのである。

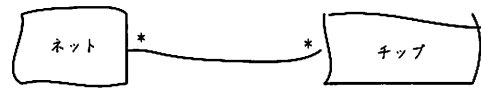


図1.1

ソフトウェアに対する彼らの要求について一緒に話し合いながら、私は図を描き始めた。シナリオのウォークスルーには、オブジェクト相互作用図のようなものを使用した。

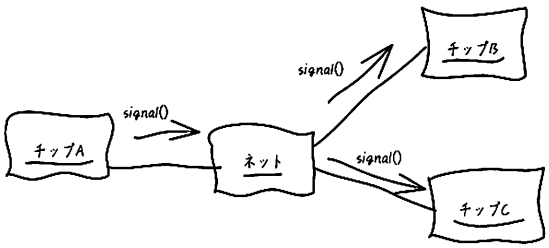


図1.2

PCBエキスパート1：コンポーネントはチップでなくてもよいのです。
開発者（私）：では、単に「コンポーネント」と呼んだ方がよいですか？
エキスパート1：我々は「コンポーネントインスタンス」と呼んでいます。同じコンポーネントが多数ある場合もありますから。
エキスパート2：「ネット」を表しているこの四角が、コンポーネントインスタンスとそっくりですね。
エキスパート1：我々の表記法を使っているわけではありませんからね。何でも四角で表すのではないですか？
開発者：残念ながら、そうなのです。この表記法については、もう少し説明した方がよさそうですね。

彼らはしきりに私の言うことを訂正し、それによって私にも知識が身につき始めた。用語法にあった不一致やあいまいさ、あるいは、技術的な見解の相違を一緒になって解決し、その過程で彼らも学んだ。その結果、彼らは物事をより厳密に矛盾なく説明してくれるようになったのだ。こうして、我々は一緒になってモデルの開発を始めたのである。

エキスパート1：信号（signal）がレフデスに届くと言うだけでは不十分で、ピンもわからないといけません。
開発者：レフデスとは何ですか？
エキスパート2：コンポーネントインスタンスと同じものです。レフデスというのは、我々が使っている特別なツールでの呼び名です。
エキスパート1：とにかく、ネットは、あるインスタンスの特定のピンを別のインスタンスの特定のピンに接続します。

開発者：つまり、ピンは1つのコンポーネントインスタンスだけに属し、1つのネットにしか接続されないということですか？

エキスパート1：ええ、その通りです。

エキスパート2：また、すべてのネットにはトポロジーがあります。トポロジーとは、ネットの要素がどう接続されるかを決定する配置の仕方です。

開発者：なるほど、これはどうでしょう？

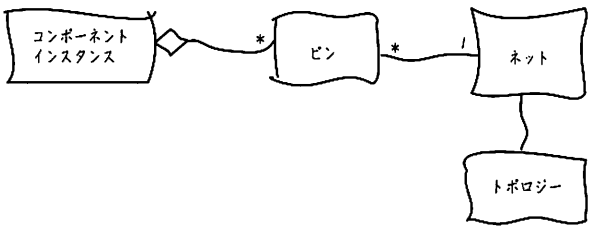


図1.3

調査の焦点を絞るため、当面、ある1つの機能に限って調べることにした。「プローブシミュレーション」とは、信号の伝搬を追跡し、設計において、ある種の問題がありそうな場所を検出する機能である。

開発者：ネットによって、そこに接続されているすべての**ピン**へ信号が送られる仕組みはわかりましたが、その先はどうなるのですか？ **トポロジー**が関係するのでしょうか？

エキスパート2：いいえ。コンポーネントが信号をプッシュ（push）して通過させます。

開発者：チップ内部のふるまいをモデル化するのは無理ですね。あまりにも複雑すぎます。

エキスパート2：モデル化する必要はありませんよ。単純化できます。ある**ピン**から他のピンへコンポーネントを通過するプッシュの一覧だけあればよいのです。

開発者：こういう感じですか？

［相当な試行錯誤をしながら、我々は一緒にシナリオの概要を描いた。］

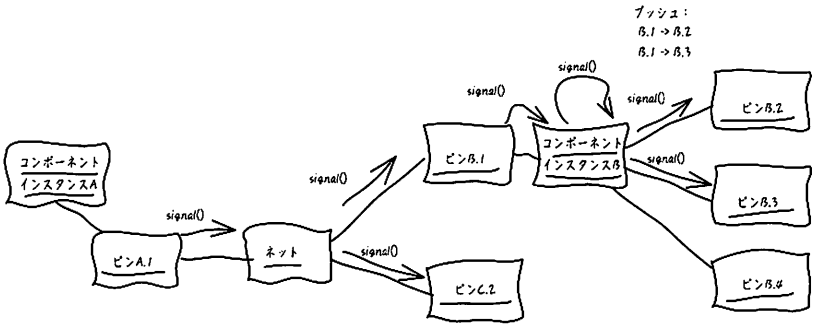


図1.4

開発者：でも、この処理をすることによって、いったい何を知りたいのですか？

エキスパート1：見つけたいのは、信号の長時間の遅延、例えば、2～3回以上ホップしている信号の経路です。大まかな目安ですけどね。経路が長すぎると、信号はクロックサイクル内に届かない可能性があります。

開発者：3回以上のホップですか…。ということは、経路の長さを計算する必要がありますね。何を1回のホップと見なすのですか？

エキスパート2：信号が**ネット**を通過することです。それが1回のホップとなります。

開発者：では、ホップの回数を渡して、**ネット**がホップ回数を加算すればよいですね、こんなふうに。

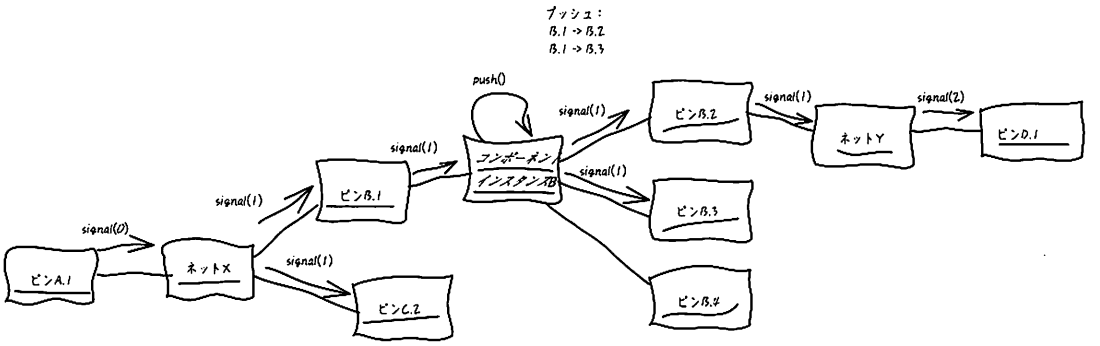


図1.5

開発者：1つわからないのは、「プッシュ」がどこから生じるのかです。すべての**コンポーネントインスタンス**に対して、プッシュのデータを格納するのですか？

エキスパート2：プッシュは、あるコンポーネントのすべてのインスタンスで同じになります。

開発者：それなら、コンポーネントタイプによってプッシュが決定されますね。どのインスタンスでも同じですか？

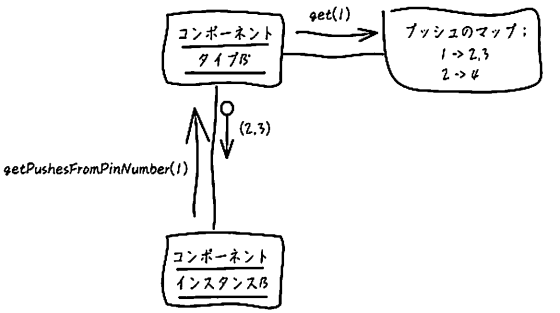


図1.6

エキスパート2：意味がよくわからないところもありますが、各コンポーネントでプッシュを格納する際には、そういう感じになると思います。

開発者：すみません、詳細に立ち入りすぎました。深く考えすぎていたもので…。それでは、**トポロジー**はどこで登場するのですか？

エキスパート1：プローブシミュレーションには使用しません。

開発者：では、ここは省いていいですね？ その機能を取り上げる時に再考しましょう。

こうして議論は進んだ（実際には、ここに示したよりはるかに多くつまずいたが）。ブレインストーミングを行ってはモデルを改良し、質問してはそれに対して説明してもらった。ドメインに関する私の理解が増し、問題解決にあたってモデルがどのような役割を果たすのかを彼らが理解するにつれて、モデルは成長した。この初期モデルを表すクラス図を、次に示す。

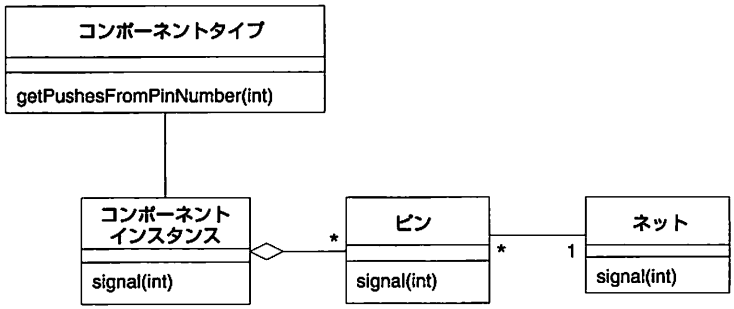


図1.7

こうしたことを、さらに2日ほど行ったところで、試しにコードを書いてみる程度には理解できた気がした。そこで、自動化テストフレームワークを使って、非常にシンプルなプロトタイプを作成した。インフラストラクチャ部分はすべて省いた。したがって、永続化もなければ、ユーザインタフェース（UI）もない。これでふるまいだけに集中することができるようになった。その後、ほんの数日のうちに、簡単なプローブシミュレーションを実演することができた。ダミーデータを使って、加工していないテキストをコンソールに表示しただけだったが、Javaオブジェクトを使用して経路の長さを実際に計算したことには変わりない。それらのJavaオブジェクトには、ドメインエキスパートと私が共有するモデルが反映されていた。

このプロトタイプが具体的であったことによって、モデルが何を意味し、実際に機能するソフトウェアにどう関係しているかを、ドメインエキスパートがより明確に理解できるようになった。それ以降、モデルに関する我々の議論は活発になった。それは、新しく獲得した知識を、私がどのようにモデルに組み込み、さらにはソフトウェアに組み込んでいるかということを、彼らが理解したからである。しかも、彼らは自身の考えを評価するための具体的なフィードバックを、プロトタイプから得ていたのだ。

当然、モデルはここに示したよりもずっと複雑になったが、そのモデルに組み込まれていたのは、解決しようとしていた問題と関連の深い、PCBのドメインに関する知識だった。また、説明に使われた多くの同義語や、ちょっとした揺れが統一された。コンポーネントの実際のディジタル特性など、技術者が理解しているものでも、ドメインと直接関係がないものはいくつも除外した。私のようなソフトウェアの専門家がこの図を見れば、数分でソフトウェアの内容を理解できるようになった。新しい情報を整理して素早く習得したり、重要なものとそうでないものをより正確に推測したり、PCB技術者とより円滑にコミュニケーションしたりするためのフレームワークができていたのである。

技術者が、自分たちの必要とする新しい機能を説明する際には、オブジェクト間の相互

作用に関するシナリオを、私に対して机上で示してもらった。モデルオブジェクトを用いて、重要なシナリオを最後まで実現できなかった時には、新しいモデルオブジェクトでブレインストーミングしたり、以前のオブジェクトモデルを変更したりして知識をかみ砕いたのだ。モデルを改良すると、コードも一緒に進化した。数か月後、PCB技術者は自分たちの期待を上回る、機能の豊富なツールを手にしていた。

効果的なモデリングの要素

我々の行った次のような事柄が、前述の成功につながった。

1. **モデルと実装を結びつける。**あの荒削りなプロトタイプによって、本質的な結びつきが早い時期に作り出され、それが以後のイテレーションを通してずっと維持された。
2. **モデルに基づいて言語を洗練させる。**当初、エンジニアはPCBの基本的な問題を私に説明しなければならず、私もクラス図が意味するものを説明しなければならなかった。しかし、プロジェクトが進むにつれ、我々全員がモデルから用語をそのまま取り出し、モデルの構造と矛盾しない文章にまとめて、通訳しなくても正確に理解し合えるようになった。
3. **知識豊富なモデルを開発する。**オブジェクトには、ふるまいと、守るべきルールがある。モデルは単なるデータスキーマではなく、複雑な問題を解決するのに欠かせないものだった。モデルはさまざまな種類の知識をとらえていたのだ。
4. **モデルを蒸留する。**モデルが完成していくにつれて、重要な概念が追加されていった。だが、それと同様に重要なのは、役に立たない、あるいは核心でないとわかった概念が、取り除かれていったことである。不要な概念が必要な概念と結びついている時には、本質的な概念を切り分け、その他の不要なものを削除できるように新たなモデルを探し出した。
5. **ブレインストーミングと実験を行う。**言語をスケッチやブレインストーミングと組み合わせることで、我々の議論の場はモデルの実験室に変わった。そこではモデルの実験用バリエーションを数多く使用し、試し、使えるかどうかを判断することができた。チームがシナリオを詳細に検討する際には、会話で使われる表現自体が、提案されたモデルを実現できるかどうかについて調べる簡易的なテストになった。表現が明確でわかりやすいか、それともぎこちないかを、聞いてすぐに察知することができたからである。

ブレインストーミングと大量の実験が持つ創造性は、モデルに基づいた言語を通じて活用され、実装を通じたフィードバックループによって鍛えられたものだ。この創造性によって、知識豊富なモデルを見つけ出し、そのモデルを蒸留できるようになる。こうして**知識をかみ砕くこと**で、チームの持っている知識が価値のあるモデルへと変わるのだ。

知識のかみ砕き

金融アナリストは数字をかみ砕く。多くの詳細な数字をふるいにかけ、根底にある意味を求めて何度も組み合わせ直し、本当に重要なものを明らかにしてくれるシンプルな説明を探す。こうした理解こそが、金融上の意思決定を行う際に、基礎となり得るのである。

有能なドメインモデラは知識をかみ砕く。彼らは情報の奔流の中に重要な一滴を探る。次から次へと考えをまとめてみて、大部分の意味を理解できるシンプルな見方を見つけようとする。多くのモデルが試されて、却下されたり変更されたりする。すべての詳細を解明する抽象概念が、一式現れてくれば成功だ。こうした蒸留により、ビジネスと最も関連の深いことが判明した特定の知識が、厳格に表現される。

知識のかみ砕きは独りで行う作業ではない。開発者とドメインエキスパートからなるチームが共同して行うもので、たいていは開発者が率いる。一緒になって、彼らは情報を引き出し、それをかみ砕いて役に立つかたちにする。その材料は、ドメインエキスパートの頭の中や既存システムのユーザ、関連するレガシーシステムや、同一ドメインの別プロジェクトにおける技術チームの過去の経験などから得られる。それは、プロジェクト用に作成されたドキュメントや、ビジネスで使用されるドキュメント、さらにおびただしい量の話し合いというかたちでもたらされる。初期のバージョンやプロトタイプでの経験がチームにフィードバックされて、解釈が変わっていく。

旧来のウォーターフォール手法では、ビジネスエキスパートがアナリストに話をして、アナリストがかみ砕き、抽象化して、その結果をソフトウェア作成者であるプログラマに伝える。このアプローチがうまくいかないのは、フィードバックが完全に欠けているからだ。アナリストはモデルを作成する全責任を負っているが、基になるのはビジネスエキスパートから得た情報だけだ。プログラマから学んだり、ソフトウェアの初期バージョンを体験したりする機会は全くない。知識は一方向に少しずつ流れるだけで、蓄積されないのである。

イテレーティブなプロセスを採用しているのに、抽象化を行っていないせいで、知識の積み重ねに失敗しているプロジェクトもある。開発者は、必要とする機能をエキスパートに説明してもらった上で、それを構築する。その結果をエキスパートに示し、次に何をす

べきかを尋ねる。プログラマが習慣的にリファクタリングを実行すれば、ソフトウェアを、継続的に拡張できる程度にはクリーンな状態に保てる。だが、プログラマがドメインに興味を持っていなければ、プログラマはアプリケーションが何をすべきかだけを調べ、その背後にある原理には注意を払わない。そうしたやり方でも、使えるソフトウェアを構築することはできるが、そういうプロジェクトは、既存機能の必然的な帰結として強力な新機能が現れるという段階には、決して到達することがない。

優秀なプログラマは、当然、抽象化することから始め、より多くの処理を行うことのできるモデルを開発する。ただし、その背景にあるのが技術だけで、ドメインエキスパートとの共同作業がない場合、その概念は素朴なものにしかならない。そのように知識が浅薄でも、基本的な処理を実行するソフトウェアを作り出すことはできるが、そのように作られたソフトウェアが、ドメインエキスパートの考え方と深く結びつくことはない。

チームメンバ間のやりとりは、メンバ全員がモデルを一緒にかみ砕くことで変化する。ドメインモデルを絶えず改良すると、開発者は、自分が力を貸しているビジネスにおける重要な原理を習得するように強いられ、機械的に機能を作り出すことがなくなる。一方、ドメインエキスパートは、自分の知っていることを蒸留して本質を抽出するように強いられることによって、しばしば自分の理解を精緻にし、ソフトウェアプロジェクトが必要とする概念の正確さを理解するようになる。

こうしたことすべてによって、チームメンバはよりうまく知識をかみ砕けるようになる。チームメンバは、無関係なものを選り分け、より一層役立つかたちにモデルを作り直す。アナリストとプログラマによって情報が与えられているので、そのモデルはきれいに構成され、また抽象化されており、それによって実装は大いに助けられる。ドメインエキスパートもモデルに対して情報を提供しているから、モデルにはビジネスに関する深い知識が反映されている。その抽象がビジネスの真の原理なのだ。

改善されるにつれて、モデルは、プロジェクトの中を流れ続ける情報を体系化するためのツールとなっていく。モデルは、要件を分析することに焦点を合わせているが、プログラミングや設計とも深く相互作用する。好循環が起きれば、こうしたモデルの性質によってチームメンバのドメインに対する洞察が深まり、状況がより明確に理解できるようになって、モデルがさらに精緻化される。このようなモデルは、決して完成することがない。代わりに、進化していくのだ。ドメインを解明するにあたって、モデルは実用的で役に立つものでなければならない。また、アプリケーションがシンプルに実装され、理解しやすくなるように、モデルは厳密でなければならない。

継続的学習

ソフトウェアを書き始める時、我々は対象を十分に理解しているわけではない。プロジェクトに関する知識は断片的で、多くの人々やドキュメントの間に散在しており、他の情報とも混ざっているから、どの知識が本当に必要なのかということさえもわからない。技術的にそれほど難しくなさそうなドメインには、だまされるかもしれない。つまり我々は、自分たちがどれほど理解していないかを認識していないのだ。こうした無知のせいで、誤った想定をしてしまうのである。

一方、すべてのプロジェクトから知識は流出している。何かを習得した人は他の所へ移動する。再編成によってチームはばらばらとなり、知識が再び断片化してしまう。重大なサブシステムがアウトソーシングされると、コードは納品されても、知識はついてこない。典型的な設計アプローチの場合、コードとドキュメントは、苦労して手に入れた知識を使える形式で表現しないから、口頭での伝承が何らかの理由で途切れたら、知識は失われてしまう。

高度に生産的なチームは、自分たちの知識を意識的に育てて、**継続的学習**（Kerievsky 2003）を実践する。開発者にとって、これが意味するのは、（本書にあるような）一般的なドメインモデリングのスキルと合わせて、技術的な知識を向上させるということだ。ただし、その時に取り組んでいる具体的なドメインについて真剣に学習することも、ここに含まれる。

こうして自ら学んだチームメンバは、最も重要な領域を含む開発作業に集中するにあたって、メンバの中で信頼できるコアとなる（これに関する詳細は、第15章を参照）。このコアチームの頭の中に知識が蓄積されることで、彼らは知識をよりうまくかみ砕けるようになる。

ここで立ち止まって、自問して欲しい。PCB設計プロセスについて、何か学ぶところがあっただろうか？ この例はドメインを表面的にしか扱わなかったが、ドメインモデルについての議論からは、何かしら学んだはずである。私自身は大いに勉強になった。PCB技術者になる方法を学んだのではない。目標はそこにはなかった。私が学習したのは、PCBエキスパートと話をして、アプリケーションに関係する重要な概念を理解し、構築している内容が正しいかどうかのチェックを行うことである。

実のところ、我々のチームは、最終的にプローブシミュレーションの開発の優先順位が低いことを発見し、この機能は結局開発しないことになった。同時に、モデルの中から、信号をプッシュしてコンポーネントを通過させることと、ホップをカウントすることについての理解をとらえていた部分も除去された。アプリケーションのコアは別にあることがわかったので、その側面を舞台の中央へ持ってくるようにモデルが変更されたのである。

ドメインエキスパートがより多くを学んだことで、アプリケーションの目標が明確になったのだ（第15章で、この問題を掘り下げて説明する）。

機能の開発が取り止めになったとはいえ、初期の作業は不可欠だった。主要なモデル要素がそのまま残ったということもあるが、それよりも重要なのは、初期の作業が知識のかみ砕きというプロセスを始動させ、それによって以降の全作業が効果的に行われたということだ。すなわち、チームメンバ、開発者、ドメインエキスパートが同じように知識を得て、共有された言語が芽を出し、実装を通してフィードバックループが完結した。発見の旅は、どこかから始めなければならないのだ。

知識豊富な設計

PCBの例で示したように、モデルによってとらえられる知識は、「名詞を見つける」ことに留まらない。ビジネスの活動やルールも、ドメインに含まれるエンティティと同じように、ドメインにとって中心的なのだ。このように、どんなドメインにもさまざまな種類の概念がある。知識をかみ砕くことによって、この種の洞察を反映したモデルが生み出される。モデルの変化と平行して、そのモデルを表現するべく、開発者は実装をリファクタリングする。それによって、新しい知識をアプリケーションで使用できるようになる。

エンティティや値を超えてその先に行こうとする、このような動きに伴った時こそ、知識のかみ砕きは力を発揮できる。複数のビジネスルールの間に、実は矛盾があるかもしれないからである。通常、ドメインエキスパートは、自分の頭の中で起きているプロセスがいかに複雑かを意識することなく、仕事をする中でこれらのルールをすべて調べて矛盾を調整し、常識で考えて隔たりを埋めている。だが、こういうことはソフトウェアにはできない。ソフトウェアエキスパートと密接に協力する中で、知識をかみ砕くことによって初めて、ルールが明確となり、具体化されて、折り合いがつけられるか、あるいはスコープの対象外とされるのである。

例 1.1 ——隠された概念を引き出す

非常に単純なドメインモデルから始めよう。これは、ある船の一回の航海に対して、貨物を運んでもらうように予約するアプリケーションの土台になり得るものだ。

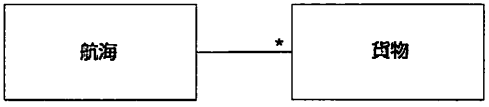


図1.8

予約アプリケーションの責務は、各貨物（Cargo）を航海（Voyage）と関連づけ、その関係を記録し、追跡することだと言える。ここまでは順調だ。アプリケーションコードのどこかに、次のようなメソッドが入るだろう。

```
public int makeBooking(Cargo cargo, Voyage voyage){
   int confirmation = orderConfirmationSequence.next();
   voyage.addCargo(cargo, confirmation);
   return confirmation;
}
```

土壇場でのキャンセルはよくあることだから、輸送業界の標準的な慣習では、1回の航海でその船舶が運搬できる量より多く貨物を受けつける。これを「オーバーブッキング」という。収容能力の110%を予約するなど、積載量に対して単純なパーセンテージを使用することもあるし、主要顧客や特定の貨物を優先して、より複雑なルールを適用する場合もある。

これは、輸送業界で働く人なら誰もが知っている輸送ドメインの基本的戦略だが、ソフトウェアチームの技術担当者全員が把握しているとは限らない。

要求定義書の中には、次の一文が書かれている。

> 10%のオーバーブッキングを認める。

クラス図とコードは次のようになる。

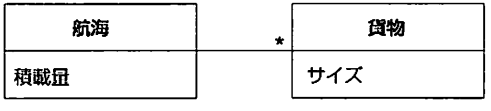


図1.9

```
public int makeBooking(Cargo cargo, Voyage voyage){
   // 最大予約＝航海の積載量×1.1
   double maxBooking = voyage.capacity() * 1.1;
   if ((voyage.bookedCargoSize() + cargo.size()) > maxBooking)
      return -1;
```

```
    int confirmation = orderConfirmationSequence.next();
    voyage.addCargo(cargo,confirmation);
    return confirmation;
}
```

ここでは、重要なビジネスルールが、ガード節としてアプリケーションメソッドの中に隠されている。第4章で、オーバーブッキングのルールをドメインオブジェクトへ移すよう導くレイヤ化アーキテクチャの原理を取り上げるが、ここでは、この知識をより明示的にし、プロジェクトに携わる人全員が使えるようにする方法に専念しよう。そのためには、似たような解決策がある。

1. このままでは、たとえ開発者が手助けしても、ビジネスエキスパートがコードを読んで、ルールが正しいかどうかを検証できそうもない。
2. ビジネスについて知らない技術者が、要求定義書の記述をコードと結びつけることは困難である。

ルールがより複雑になれば、問題はさらに大きくなる。

この知識をより適切にとらえるよう、設計を変更することができる。オーバーブッキングのルールは、ポリシーである。**ポリシー**とは、ストラテジーとして知られている設計パターンの別名だ（Gamma他 1995）。通常、このパターンが使われるのは、さまざまなルールを置き換える必要のある場合だが、現時点でわかっている限り、その必要性はない。しかし、ここでとらえようとしている概念は、ポリシーの意味と適合している。このこともまた、ドメイン駆動設計では重要な動機となる（第12章「デザインパターンをモデルに関係づける」を参照）。

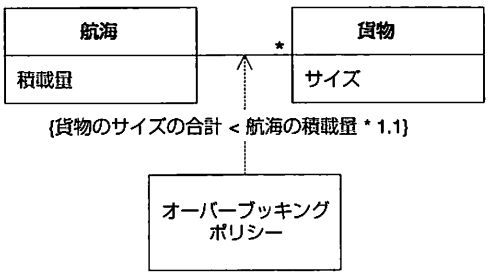


図1.10

コードは次のようになる。

```
public int makeBooking(Cargo cargo,Voyage voyage){
    // オーバーブッキングポリシーに照らして、貨物が航海で許可されるか
    if (!overbookingPolicy.isAllowed(cargo, voyage)) return -1;
    int confirmation = orderConfirmationSequence.next();
    voyage.addCargo(cargo,confirmation);
    return confirmation;
}
```

●例 1.1 ——隠された概念を引き出す

新しいオーバーブッキングポリシークラスには、次のメソッドがある。

```
public boolean isAllowed(Cargo cargo, Voyage voyage){
   return (cargo.size() + voyage.bookedCargoSize()) <=
      (voyage.capacity() * 1.1);
}
```

オーバーブッキングというものがはっきり識別できるポリシーであることは、誰にとっても明白であり、そのルールの実装は明示的で独立している。

ところで、**私は、こうした手の込んだ設計をドメインのあらゆる詳細に適用することを推奨しているわけではない。**第15章「蒸留」では、重要なことに集中して、それ以外のものをすべて最小限にするか分離するための方法を掘り下げる。この輸送ドメインの例は、ドメインモデルとそれに対応する設計が、知識を確固たるものにして共有するのに使えることを示すために用いた。より明示的な設計には、次のような利点がある。

1. この段階まで設計を進められる頃には、プログラマやその他の関係者の誰もが、オーバーブッキングの本質を、単なるわかりにくい計算ではなく、明確で重要なビジネスルールだと理解するようになっているだろう。
2. プログラマは、ビジネスエキスパートに技術的な成果物を見せることができる。コードさえも見せられるのだ。こうした成果物は、ドメインエキスパートに（開発者の手助けがあれば）理解できるはずで、それによってフィードバックループが完結する。

深いモデル

役立つモデルがすぐに見つかる表層にあることはめったにない。ドメインとアプリケーションがやるべきことを我々が理解するようになるにつれて、たいていの場合、表層にあり最初は重要と思われたモデル要素を廃棄するか、それらのモデル要素を違った視点で見ることになる。すると、最初は考えつかなかった、それでいて問題の核心を突くような、うまい抽象が現れてくるのだ。

前述した輸送ドメインの例がおおよそ基にしたのは、コンテナ輸送システムである。これは、本書を通じていくつもの例で利用するプロジェクトの1つである。本書で示す例は、輸送業のエキスパートでなくても理解できるようにしてある。しかし、実際のプロジェクトでは、継続的学習によってチームメンバもドメインの複雑さに対応できるようになるのであり、有用で明確なモデルを作るにあたって、ドメインとモデリングテクニックの両面においてさらなる洗練が求められることも多い。

例に挙げたプロジェクトの場合、貨物の予約行為によって輸送が始まるため、貨物や輸送日程などを記述できるようにするモデルを開発した。これは必要であったし、役に立ち

もしたが、ドメインエキスパートは満足していなかった。我々が見落していた、彼らなりのビジネスの見方があったのである。

結局、何か月も知識をかみ砕いた後で我々が気づいたのは、貨物の取り扱い、すなわち物理的な荷積みと荷下ろしや、ある場所から別の場所への移動などは、ほとんど下請け会社や社内の業務担当者によって行われているということだった。輸送業エキスパートから見れば、関係者の間で連続して責任が移動するということだったのだ。そうした荷主から地元の輸送会社へ、さらに輸送手段を複数経由して最終的な荷受人までという法的かつ実務的な責任の移動を、1つのプロセスが管理していた。貨物が倉庫に置かれたまま、重要な手続きが行われることもよくあった。あるいは、貨物の物理的移動を伴うような複雑な手続きが行われても、それが輸送会社のビジネス上の意思決定とは関係ないこともあった。重要視されるようになったのは、輸送日程の手配よりも、船荷証券などの法律書類や支払い免除につながるプロセスだったのだ。

輸送業に対する見方がこうして深まったことにより、輸送日程オブジェクトが削除されることこそなかったが、モデルは抜本的に変化した。輸送に関する我々の考え方は、コンテナをある場所から別の場所へ動かすことから、貨物に対する責任を、ある当事者から別の当事者へ移動することに変わった。この責任の移動を扱う機能は、荷積み業務へとぎこちなく加えられることはなくなった。代わりにこの機能は、そうした業務と責任の間にある重要な関係性を理解することで得られたモデルによって支えられるようになった。

知識のかみ砕きは探究だから、最終的にどこへ行き着くかはわからないのだ。

第2章

コミュニケーションと言語の使い方

ドメインモデルは、ソフトウェアプロジェクトにおける共通言語のコアとなることができる。モデルとは、プロジェクトに携わる人々の頭の中で構築された概念の集まりであり、ドメインについての洞察を反映した、用語と概念間の関係性からできている。こうした用語と相互関係は、そのドメイン用にあつらえた言語のセマンティクスを提供するのであり、同時に、技術的な開発ができるくらい正確でもある。言語は、開発という活動にモデルを織り込んでモデルとコードと結び合わせるための、重要な糸なのだ。

モデルに基づいたこのようなコミュニケーションは、統一モデリング言語（UML: Unified Modeling Language）で描かれた図に限られない。モデルを最大限に有効活用するには、コミュニケーションのためのあらゆる媒体にモデルを浸透させる必要がある。そうすれば、テキストで書かれたドキュメントだけでなく、略式の図や、くだけた会話も、より役に立つようになる。こうした図や会話は、アジャイルプロセスで再び強調されるようになったものだ。また、コード自体とそのコードに対するテストを通じて、コミュニケーションが改善される。

プロジェクトにおける言語の使い方は、とらえにくいが、実に重要なのだ…。

ユビキタス言語（UBIQUITOUS LANGUAGE）

まず一文を書いて、
次にそれを細かく切り刻みなさい。
そして、刻んだ欠片を混ぜ合わせ、
こぼれ落ちるに任せて並べてみなさい。
フレーズの順序が変わっても、
結局違いはないのだから。

—Lewis Carroll『Poeta Fit, Non Nascitur』
（詩人は作られるものにして、生まれるものにあらず）

しなやかで知識豊富な設計を行うには、用途の幅広い、共有されたチームの言語と、その言語を使った活発な実験が必要である。しかし、そういったことは、ソフトウェアプロジェクトではめったに見られない。

＊ ＊ ＊

ドメインエキスパートは、ソフトウェア開発における技術的な専門用語のことは部分的にしか理解せず、代わりに自分の得意分野の専門用語を使用する。それも、さまざまなニュアンスで使うだろう。一方で、開発者はシステムを理解して議論する際に、機能に関する説明的な用語を用いるかもしれないが、こうした用語はドメインエキスパートの言語が持っていた意味を欠いてしまっている。あるいは、開発者による抽象化は、自分たちの設計を支えてはいても、ドメインエキスパートには理解されないかもしれない。ある問題に対して、それぞれ別の部分に取り組んでいる開発者たちは、自分たち独自の設計概念とドメインを記述する方法を、それぞれで考え出してしまう。

言語の間にはこうした断絶があるので、ドメインエキスパートは自分たちが欲しいものをあいまいにしか記述しない。開発者は、よく知らないドメインを理解しようと苦労するが、その理解もあいまいだ。チームメンバのうち数人は、何とか両者の言語を理解できるようになるものの、情報の流れにおいてボトルネックになってしまうし、その通訳も正確ではない。

共通言語のないプロジェクトの場合、開発者は、ドメインエキスパートのために通訳しなければならない。ドメインエキスパートは、開発者とさらに別のドメインエキスパートとの間で通訳する。開発者同士でも通訳することがある。通訳することでモデルの概念を混乱させてしてしまい、それがコードの破壊的なリファクタリングにつながる。間接的にしかコミュニケーションをしていないと、分裂が起きつつあることが見えなくなる。つまり、さまざまなチームメンバが用語を別々の意味で使いながら、そのことを認識していな

いのである。ここから作り出されるのは、整合性の取れていない、信頼できないソフトウェアだ（第14章参照）。通訳にかかる労力のせいで、知識と考え方に関する交流が妨げられてしまうが、この交流によってこそ、モデルについての深い洞察がもたらされるのだ。

言語に亀裂があると、プロジェクトは深刻な問題に直面する。ドメインエキスパートが自分たちの専門用語を使用するのに対して、技術チームのメンバも独自の言語を持っている。この言語は、設計の観点からドメインを議論するために調整されたものだ。

日々の議論で使う用語法が、コードに埋め込まれる用語法から切り離されている（だが、コードはソフトウェアプロジェクトにおいて最終的に最も重要な成果物である）。同一人物でさえ話し言葉と書き言葉で違いがあるので、ドメインについての最も鋭い表現は、たいてい一時的なかたちでしか現れず、コードや文章ではとらえられることがない。

通訳はコミュニケーションを鈍らせ、知識のかみ砕きを沈滞させる。

しかも、これらの方言はどれも共通語になれない。どの方言もすべての要求には応えられないからだ。

すべての通訳にかかる余分なコストに、誤解のリスクまで加わるのでは高くつきすぎる。プロジェクトには、複数の方言についての最大公約数に留まらない、より強固な共通言語が必要なのだ。チームが意識的に努力すれば、ドメインモデルはその共通言語の基盤となれるし、その一方で、チームのコミュニケーションをソフトウェアの実装と結びつけることもできる。この言語はチームで行う作業のいたるところに（ユビキタスに）存在できるのだ。

ユビキタス言語の語彙には、クラスや主要な操作の名前が含まれている。また、モデルの中で明示されたルールについて議論するための用語も含まれている。この言語は、モデルが従うべき高次の構成原理（コンテキストマップや大規模な構造など、第14章と第16章で説明するもの）に由来する用語によって補完される。そして最後は、ドメインモデルに対して一般に適用されるパターンの名前によって、この言語は強化される。

モデルの中にある概念間の関係性は、あらゆる言語に存在する結合規則となる。また、単語やフレーズの意味には、モデルのセマンティクスが反映されている。

モデルに基づいたこの言語は、開発者の間で使用されなければならない。それも、システムにおける成果物を記述するためだけでなく、タスクや機能を記述するためにも使われなければならないのだ。開発者が使うのと同じモデルによって提供される言語は、開発者とドメインエキスパートが互いに意思疎通をする際にも用いられなければならないし、ドメインエキスパート同士が要件や開発計画、あるいは機能を伝え合う際にも用いられなければならない。言語が広く使用されるほど、理解もよりスムーズに進むようになるのだ。

最低でも、ここまではたどり着かなければならない。しかし最初は、そういう役割を果たせるほど、モデルは優れていないかもしれない。対象となる分野に特化した専門用語の

持つ意味の豊かさが欠けているかもしれないのだ。しかし、こうした専門用語も、あいまいで矛盾があるものだから、そのまま使用することはできない。また、開発者がコードの中に作り込んだ、実際に使われているちょっとした機能が、モデルで表現されていないかもしれない。これは、開発者がこうした機能をモデルの一部と見なしていないか、コーディングスタイルが手続き的で、ドメインの概念を暗黙的にしか伝えていないことによる。

モデルと言語のこうした循環関係は、どちらが先かわかりにくいかもしれない。しかし、知識をかみ砕くプロセスが、より役立つモデルを作り出せるかどうかは、モデルに基づく言語に対してチームが献身するかどうかにかかっている。粘り強くユビキタス言語を使用すれば、モデルの弱点は明らかにならざるを得ない。チームは実験を行って、用語や、用語の組み合わせがぎこちない場合に、その代わりとなる表現を探す。言語に食い違いが見つかると、新しい単語が議論で使われるようになる。**言語に対するこうした変更は、ドメインモデルに対する変更として認識され**、用語の意味が変わった場合には、チームがクラス図を更新して、コード中のクラスやメソッドの名前を変えたり、ふるまいを変更したりすることにまでつながる。

この言語を実装のコンテキストで使用することに全力を傾けると、開発者は不正確なところや矛盾を指摘し、ドメインエキスパートを巻き込んで、有効な代わりの表現を探すことになる。

もちろん、ドメインエキスパートはユビキタス言語のスコープの外で話すこともあるが、それはより広いコンテキストを説明して提示するためだ。しかし、モデルが対象とするスコープの中ではユビキタス言語を使用すべきで、それがぎこちなかったり、不完全だと思われたりした場合、さらには誤っていると思われた場合には、懸念を表明しなければならない。モデルに基づく言語を広く使用し、妥協せずにそれがよどみなく流れるようにすることによって、我々は完全で包括的なモデルに近づいていくのである。こうしたモデルはシンプルな要素から成り立っており、この要素が結合して複雑な概念を表現するのだ。

それゆえ：

モデルを言語の骨格として使用すること。チーム内のすべてのコミュニケーションとコードにおいて、その言語を厳格に用いることを、チームに約束させること。図やドキュメント、そして何より会話の中では同一の言語を使用すること。

言語を使う上で問題があれば、代わりの表現を用いて実験することで、問題を取り除くこと。そうした表現は代わりとなるモデルを反映している。そこで、新しいモデルに合わせてコードをリファクタリングし、クラス、メソッド、モジュールの名前を変更すること。会話の中で用語が混同されていたら、普通の単語の意味について認識を合わせるのと同じやり方で解決すること。

ユビキタス言語における変更は、モデルに対する変更であると認識すること。

ドメインエキスパートは、ドメインについての理解を伝えるには使いにくかったり不適切だったりする用語や構造に異議を唱えるべきであり、開発者は、設計を妨害することになるあいまいさや不整合に目を光らせるべきである。

ユビキタス言語を用いることによって、モデルは単なる設計成果物ではなくなり、開発者とドメインエキスパートが協同で行うことすべてにおいて、欠かせないものとなる。また、ユビキタス言語は知識を生き生きとした形式で伝える。そうしたユビキタス言語を使って議論することによって、図とコードの背後にある意味に命が吹き込まれるのだ。

＊　＊　＊

前述したユビキタス言語の説明では、モデルが1つだけ使われることを想定している。第14章「モデルの整合性を維持する」では、別々のモデル（およびユビキタス言語）の共存と、モデルを分派させないための方法を扱う。

ユビキタス言語は、設計にあってコードに現れない側面を伝達するための主要な手段である。このような側面としては、システム全体を構成する大規模な構造（第16章参照）や、別々のシステムや別々のモデル間の関係性を定義する境界づけられたコンテキスト（第14章参照）のほか、モデルと設計に適用されるその他のパターンがある。

例 2.1 ——貨物輸送プログラムを完成させる

次に続く2つの対話には、わずかながら重要な違いがある。それぞれのシナリオにおける会話の内容について、このソフトウェアがビジネスにとって何を意味しているのかと、このソフトウェアが技術的にどう機能するのかを比較した時に、それぞれについて、どのくらい多く話しているかに注目して欲しい。ユーザと開発者は、同じ言語で話しているだろうか？ その言語は、アプリケーションがすべきことについて議論する上で、十分に豊かだろうか？

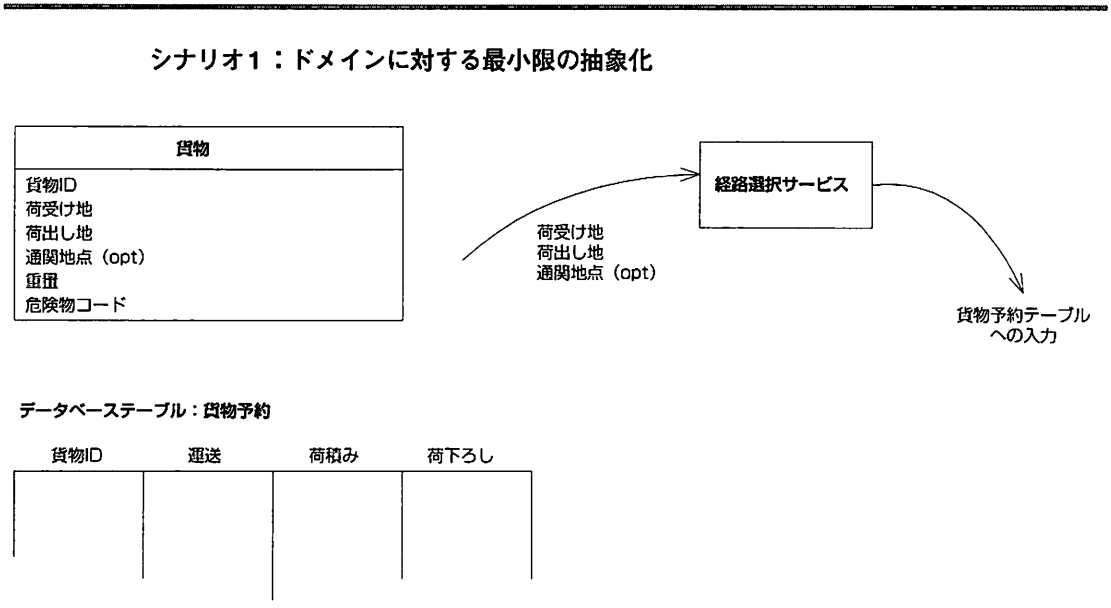


図2.1

ユーザ：したがって、通関地点を変更すると、経路計画を全部作り直さないといけません。

開発者：わかりました。該当する貨物IDを持ったすべての行を輸送テーブルから削除してから、荷受け地と荷出し地、新しい通関地点を**経路選択サービス**（Service）に渡せば、テーブルにデータが再登録されます。**貨物**（Cargo）にブール値を持たせて、貨物テーブルにデータがあることがわかるようにしましょう。

ユーザ：行を削除する？ まあ、いいでしょう。とにかく、それまでに通関地点を一切入力していなくても、通関地点が変更されたら経路計画は作り直す必要があります。

開発者：もちろんです。荷受け地や荷出し地、通関地点を変更（または初めて入力）したら、貨物データがあるかどうかを必ずチェックして、そのデータを削除してから、**経路選択サービス**に再生成させることになります。

ユーザ：もちろん、過去に入力した通関地点が偶然正しかった場合には、そんなことをしたくありませんね。

開発者：ええ、問題ありません。**経路選択サービス**で、毎回荷積みと荷下ろしの計画を作り直すだけの方が簡単ですから。

ユーザ：なるほど。しかし、輸送日程が新しくなった時に、補助的な計画を全部作成するのは、余計な仕事になってしまいます。ですから、必要にならない限り、経路変更はしたくありません。

開発者：おっと。そうなると、最初に通関地点を入力する場合、テーブルに問い合わせて以前の通関地点を探した後に、新しいものと比較しなければなりません。それで、やり直し

が必要かどうかわかります。

ユーザ：荷受け地や荷出し地の変更に関しては、計画のこうした再利用を気にする必要はありません。その時には、いつも輸送日程が変わるでしょうから。

開発者：わかりました。

シナリオ2：議論を支えられるように豊かになったドメインモデル

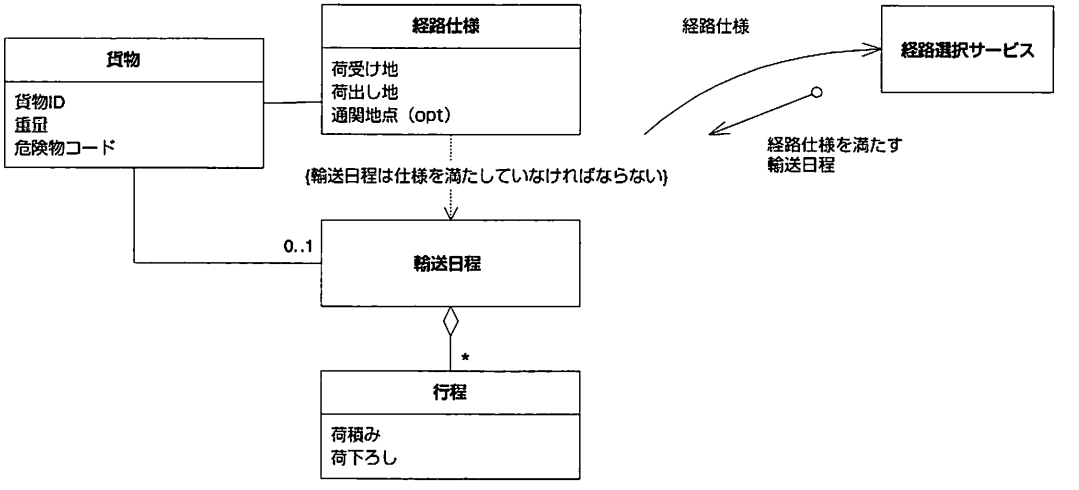


図2.2

ユーザ：したがって、通関地点を変更すると、経路計画を全部作り直さないといけません。

開発者：わかりました。**経路仕様**（Route Specification）にある属性をどれか1つでも変更したら、古い**輸送日程**（Itinerary）を削除した上で、**経路選択サービス**に対して、新しい**経路仕様**に基づいた新しい輸送日程を生成するよう依頼します。

ユーザ：それまでに通関地点の仕様を一切定めていなかった場合は、その定義も同時に行わなければなりません。

開発者：大丈夫です。**経路仕様**で何かを変更した時は、必ず、**輸送日程**を生成し直します。それには、初めて何かを入力した場合も含まれます。

ユーザ：もちろん、過去に入力した通関地点が偶然正しかった場合には、そんなことをしたくありませんね。

開発者：ええ、問題ありません。**経路選択サービス**に**輸送日程**を毎回作り直させる方が簡単ですから。

ユーザ：なるほど。しかし、**輸送日程**が新しくなった時に、補助的な計画を全部作成するのは、余計な仕事になってしまいます。ですから、変更によって必要にならない限り、経路

変更はしたくありません。

開発者：ああ。それなら、**経路仕様**に少し機能を追加する必要がありますね。そうして、**仕様**で何か変更されたら常に、**輸送日程**がその**仕様**を引き続き満たしているかどうかを調べます。満たしていなければ、また**経路選択サービス**に**輸送日程**を生成させましょう。

ユーザ：荷受け地や荷出し地に関しては、計画のこうした再利用を気にする必要はありません。その時には、いつも**輸送日程**が変わるでしょうから。

開発者：いいでしょう。しかし、毎回比較する方がシンプルですよ。**輸送日程**は**経路仕様**が満たされなくなった時にしか生成されませんから。

2番目の対話の方が、ドメインエキスパートの意図をより多く伝えている。ユーザはどちらの対話でも「輸送日程」という言葉を使用しているが、2番目の方では、正確かつ具体的に2人が議論できるオブジェクトになっていた。2人は「経路の仕様」について明示的に論じていて、属性や手続きという観点からその都度説明するようなことはない。

この2つの対話は、わざとよく似せて構成してある。しかし、もし現実であれば、前者はよりくどくなり、アプリケーションの持つ機能の説明と会話の行き違いによって膨れ上がったはずだ。後者の設計では、ドメインモデルに基づいた用語法が使われているので、2番目の対話はより簡潔になっている。

声に出してモデリングする

会話での表現が、他のコミュニケーション形式と切り離されてしまうと、とりわけ大きな損失となる。これは、我々人間が話し言葉を操る才能を持っているからである。しかし残念ながら、人は話す時に、通常はドメインモデルの言語を使わない。

こう言っても、最初のうちは本当とは思えないであろうし、実際に例外もある。しかし、次に要求や設計の議論に参加する際には、じっくりと耳を傾けて欲しい。機能を説明するにあたって、ビジネスの専門用語や、門外漢によるその意訳が使われているのがわかるだろう。技術的な成果物と具体的な機能に関する話も耳に入る。もちろん、ドメインモデルに由来する用語も聞こえてくるだろう。ビジネスの専門用語に由来する共通言語に含まれる明らかな名詞は、たいていオブジェクトとしてコーディングされるから、そうした用語は言及されがちなのだ。しかし、これがフレーズとなると、現在のドメインモデルにおける関係性と相互作用を表す用語を使って、遠回しにでも説明されているのを聞いたことがあるだろうか？

モデルを改良する最適な方法の1つは、話してみることだ。考えられるモデルのバリエーションから生じるさまざまな概念を、声に出して構成してみる。粗削りな表現は聞け

ばすぐわかる。

「**経路選択サービス**に荷受け地と荷出し地および到着時間を与えた場合、それが貨物に必要な停泊箇所を調べて、ええと、それらをデータベースに入れておけばいい。」(**あいまいで技術的**)

「荷受け地や荷出し地などがあって、これらを全部**経路選択サービス**に与えると、必要なものすべてが揃った**輸送日程**が戻ってくる。」(**より完全だが冗長**)

「**経路選択サービス**は、**経路仕様**を満たしている**輸送日程**を見つける。」(**簡潔**)

単語やフレーズを使っていろいろ試しながら、モデリング作業に言語能力を活用することがきわめて重要だ。これは、図を描くことによって視覚的、空間的な思考を駆使することがきわめて重要なのと変わらない。我々が自分たちの分析能力を用いる際に、方法論的な分析や設計と合わせて、コードの持つあの不思議な「感じ」を使うのと全く同じなのだ。このような思考方法は互いに補完し合うものなので、役に立つモデルと設計を見つけるにはそれらすべてが必要になる。その中で、言語による実験は見過ごされることが一番多い(本書の第3部では、この発見プロセスを掘り下げ、いくつかの対話の中で言語能力と視覚的な思考が相互に影響しあう様子を示そう)。

実際のところ、我々の脳は、話し言葉の複雑さに対応することにやや特化していると思われる(私のような素人向けの優れた参考書は、Steven Pinkerの『言語を生みだす本能』(Pinker 1994)だ)。例えば、異なる言語で育った人々が貿易のために集まった時、共通言語がないと、**ピジン語**(通商用混合語)と呼ばれる言語を考案する。ピジン語は、話者が本来使用している言語ほど包括的ではないが、さしあたりの作業には適している。会話をしていると、単語の解釈や意味が違っていることが自然に見つかり、その違いが自然に解消される。言語の粗削りな部分を見つけて、それを滑らかにするわけだ。

昔、私は大学でスペイン語の集中講座を受講したことがある。その時の教室内でのルールは、英語をひとことも話してはならないというものだった。最初はそれにいらいらした。とても不自然に感じられる上、強い自制と鍛錬が求められたからだ。しかし、最終的には、級友も私も、机上の練習では決して到達できなかったレベルまで流暢にスペイン語を話せるようになった。

ドメインモデルのユビキタス言語を議論で使用すると、全員がその言語でよどみなく話せるようになり、ニュアンスを互いに教え合う。このことは、開発者とドメインエキスパートがシナリオと要求を徹底的に吟味する議論では、特にそうだ。図とドキュメントでは起こり得ない方法で、会話で使う言語を自然と共有するようになるのである。

ユビキタス言語をソフトウェアプロジェクトで作り出すのは言うほど簡単ではなく、これを成し遂げるには、持って生まれた才能を十分に活用しなければならない。人間は視覚的、空間的能力を持っているので、グラフィカルに要約することで、情報を素早く伝達し

処理できる。それと同じように、文法と意味を持つ言語に対して我々が生来持っている才能を活かして、モデルの開発を推し進めることができる。

それゆえ、ユビキタス言語パターンへの補足として：

システムについて語る際には、モデルをいろいろと試してみること。モデルの要素と相互作用を使い、モデルに許された方法で概念を結びつけながら、シナリオを声に出して描写すること。表現すべきことをより簡単に言う方法を見つけ、その新しい考え方を図とコードに再び反映させること。

1つのチームに1つの言語

技術系の人々は、しばしば、ビジネスエキスパートをドメインモデルから「遮断する」必要を感じる。そして、こんなふうに言うのだ。

「彼らには抽象的すぎる。」
「オブジェクトがわかっていない。」
「彼らの用語法に従って要求を集めなければならない。」

これらは、チームに2つの言語があることへの言い訳として私が耳にした、ほんの数例である。こうした言い訳は忘れること。

もちろん、ドメインエキスパートとは関係のない設計上の技術的なコンポーネントはあるが、モデルのコアは、彼らの関心を引くべきである。抽象的すぎる？ それなら、どうしてその抽象化が適切だとわかるのか？ ドメインエキスパートと同じくらい深くドメインを理解できているというのか？ 特定の要求がもっと現場に近いユーザから集められ、そうした人のためにより具体的な用語法のサブセットが必要となる場合もあるが、ドメインエキスパートには自分の専門分野をある程度深く考える責任がある。**豊富な知識を持つドメインエキスパートがモデルを理解できないとしたら、モデルに何か問題があるのだ。**

では、最初はどうか。システムが将来持つことになるが、まだモデル化されていない機能について、ユーザが議論する時には、使用できるモデルは存在しない。しかし、彼らが開発者と一緒に新しいアイデアを検討し始めるとすぐに、共有されるモデルへ向かう手探りのプロセスが開始される。初めはぎこちなくて不完全かもしれないが、次第に改良されていく。新しい言語が進化するにつれて、ドメインエキスパートは、その言語を身につけて、依然として重要な古いドキュメントを改めて修正するといった余分な努力もしなければならない。

ドメインエキスパートが、開発者との議論や、自分たちの中での議論にユビキタス言語を使用すると、モデルが自分たちの要求を満たすのに不十分であったり、誤っていると思われたりする領域を素早く発見する。また、モデルに基づく言語が厳密であるため、ドメ

インエキスパートは（開発者の手を借りて）、自分たちの思考の矛盾やあいまいさが明らかにされる領域があることにも気づく。

開発者とドメインエキスパートは、モデルオブジェクトを段階的に使用しながらシナリオのウォークスルーを行うことにより、モデルを非公式にテストすることができる。たいていの議論は、開発者とエキスパートのユーザが共にモデルをいろいろと試してみる機会である。それが進むにつれて、相互理解が深まり、概念に磨きがかかっていく。

ドメインエキスパートは、モデルの言語を使ってユースケースを書けるし、また受入テストを定義することで、より直接的にモデルに取り組める。

要求を集めるのにモデルの言語を使用するという考えには、異議が唱えられることもある。やはり、要求はそれを満たす設計から独立していなければならないのではないか？ だが、この主張は、どの言語も何らかのモデルに基づいているという現実を見落としている。単語の意味というものは、つかみどころがない。ドメインモデルはドメインエキスパート自身の専門用語から引き出されるのが通例だが、「整理」されて、限定されたより鋭い定義を持つようになる。無論、こうした定義が該当する分野で認められている意味から外れている場合には、ドメインエキスパートは異議を唱えるべきである。アジャイルプロセスでは、要求がプロジェクトの進行に合わせて進化する。アプリケーションを適切に定義する知識は、最初から存在することがほとんどないからだ。この進化の一部として、改良されたユビキタス言語を用いて要求を再構成しなければならない。

言語の多様性が必要なこともよくあるが、言語の仕切りをドメインエキスパートと開発者の間に置いてはならない（第14章「モデルの整合性を維持する」で、同一プロジェクトにおける複数モデルの共存を取り上げる）。

もちろん、開発者がドメインエキスパートにはわからない技術用語を使用することは確かにある。開発者には、システムの技術的側面を議論するのに必要な専門用語が多数あるからだ。アプリケーションの限られたスコープや開発者の理解をはるかに超えた専門用語は、ほぼ間違いなくユーザにもあるだろう。しかし、そうした専門用語は言語が**拡張**されたものでしかない。こうした方言は、同じドメインに関して、異なるモデルを反映した別の語彙を含んでいてはならないのだ。

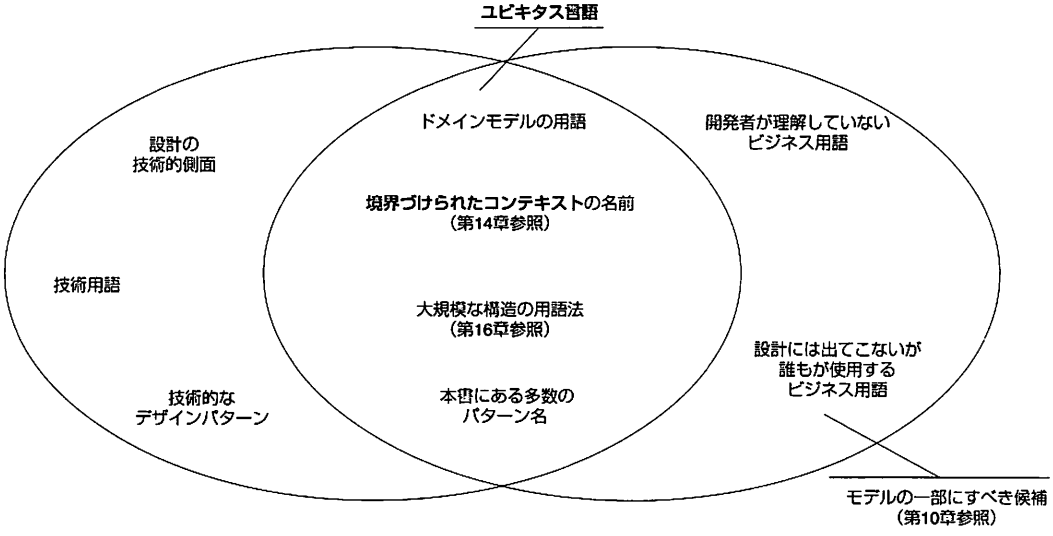


図2.3 ユビキタス言語は専門用語が交差するところで高められる

ユビキタス言語を用いることにより、開発者間の会話やドメインエキスパート間の議論、そして、コードそのものでの表現がすべて、共有されたドメインモデルから得られた同一の言語に基づくようになる。

ドキュメントと図

私は、ソフトウェア設計について議論する会議に出席する時はいつでも、ホワイトボードやスケッチブックに図を描かないと、ほとんど頭を働かすことができない。大部分はUML図で、たいていはクラス図かオブジェクト相互作用図である。

生まれつき視覚的な理解に長けた人々がいることもあって、図を描くことにより、ある種の情報は把握しやすくなる。UML図は、オブジェクト間の関係性を伝える上ではきわめて有効で、相互作用を示すのにもある程度使える。しかし、UML図では、そうしたオブジェクトの概念的な定義は伝わらない。会議中であれば、その意味は、私が図を描きながら会話の中で具体的にするか、あるいは、他の出席者との対話によって現れてくることもある。

シンプルな略式のUML図によって、議論に確固とした基盤が与えられる。目の前にある問題の中心となるオブジェクト3～5個を図に描けば、誰もが集中したままでいられる。全員がオブジェクト間の関係性についての見方を共有するようになるし、さらに意義深いことに、オブジェクトの名前も共有するようになるからだ。こうしたことに助けられて、口

頭での議論がますます効果的になる。別の思考実験を試す際には図を変更することができ、スケッチであれば、話される言葉が変わっていっても対応できる。そして、この移り変わりこそが議論の真髄でもある。結局、UMLは統一モデリング言語の略なのだ。

問題が発生するのは、モデルや設計の全体を伝えるのに、UMLを使うことを強いられていると人々が感じた時である。オブジェクトモデル図の多くは完全すぎると同時に、多くを省きすぎてもいる。完全すぎるというのは、コーディングしようとしているオブジェクトを、すべてモデリングツールに描かなければならないと人々が思っているからである。だが、そこまで詳細にしたら、木を見て森を見ずになってしまう。

それだけ詳細にしたところで、属性と関係性は、オブジェクトモデルの全体からすれば半分でしかない。オブジェクトのふるまいとオブジェクトに対する制約は、それほど簡単に図示できない。オブジェクト相互作用図は、設計において注意すべき危険な箇所を図示することはできても、相互作用のほとんどは示すことができない。図を作成するにも、それを読むにも、とにかく手間がかかりすぎるのだ。それでも相互作用図は、モデルの背景にある目的については示唆することしかできない。制約と表明を含めようとすると、UMLはテキストに頼らざるを得ず、波括弧でくくって図に挿入することになる。

オブジェクトのふるまいに関する責務は、操作名でそれとなく知らせ、オブジェクト相互作用（またはシーケンス）図で暗黙的に示すことができるが、はっきりと**述べる**ことはできない。そのため、その役目は補足的なテキストや会話が担うことになる。つまり、UML図はモデルの最も重要な側面のうちの2つ、すなわち、モデルが表している概念の意味と、オブジェクトが何を行うことになっているかを伝えられないのである。しかし、このことは問題にならない。自然言語（英語でも、スペイン語でも何でも）を慎重に使用すれば、その役割を十分にうまくこなせるからだ。

また、UMLはプログラミング言語としても、それほど満足できるものではない。モデリングツールのコード生成機能を利用する試みは、私が見てきた限り、すべて非生産的だった。UMLの能力に縛られると、モデルの最も重要な部分であるルールのいくつかを、四角と線で描かれる図に合わないからという理由で除外しなければならなくなることがある。しかも、当然ながら、コードジェネレータはテキストによる注釈を利用できない。UML風の図を使う言語で実行可能なプログラムを書けるようにする技術を使用すると、今度はUML図がプログラム自体に対する、別の見方でしかなくなり、「モデル」の意味そのものが失われてしまう。実装の言語としてUMLを使用しても、整然としたモデルを伝える別の手段はやはり必要なのだ。

図はコミュニケーションと説明のための手段であり、ブレインストーミングを容易にする。図がこれらの目的に最も合致するのは、余計なものがない場合である。オブジェクトモデル全体の包括的な図では、コミュニケーションや説明ができない。読者を詳細で圧倒

してしまうし、意味にも乏しいからだ。そのため我々は、すべてを網羅したオブジェクトモデル図や、UMLで描かれた網羅的なデータベースリポジトリを避ける。そこで向かうのが、オブジェクトモデルにおいて概念的に重要な部分を表す、簡略化した図である。このような図は設計を理解するのに欠かせない。本書中の図は、私がプロジェクトで使用しているものの典型である。これらの図はモデルを簡略化し、説明するものであり、要点を明確にする時には、標準ではない表記法も多少取り入れている。設計上の制約を示してはいるが、あらゆる詳細についての設計仕様書ではない。図が表現しているのは、考え方の骨格なのだ。

設計に関する本質的な詳細は、コードにおいてとらえられる。適切に作成された実装は透過的で、根底にあるモデルを明らかにしなければならない（これを確実にすることが、次章および本書の大部分におけるテーマである）。補足的な図とドキュメントを用いることで、人々の注意を要点へと導くことができる。自然言語で議論することで、意味のニュアンスを補うことができる。典型的なUML図が意味のニュアンスを扱うのとは全く逆のやり方を私が好むのは、こうした理由からである。図にテキストで注釈を加えるのではなく、テキストを書いた上で、選び抜かれたシンプルな図を用いて説明するのだ。

常に覚えておいて欲しいのは、**モデルは図ではない**ということである。図の目的は、モデルについてコミュニケーションし、説明するのを助けることにある。コードも、設計の詳細を格納したリポジトリとして機能できる。適切に書かれたJavaには、UMLと同じくらい豊かな独自の表現力があるのだ。図を注意深く選び抜いて構築すれば、意識を集中して話を進めるのに役立つ。ただし、モデルや設計を完全に表現しなければならないという脅迫観念のせいで、わかりにくくならなければ、の話だ。

書かれた設計ドキュメント

口頭でのコミュニケーションによって、厳密で詳細なコードに対して意味が与えられる。確かに、すべての人をモデルに結びつける上で、話すことはきわめて重要だ。だが、そうはいっても、書かれたドキュメントが安定し共有可能であることは、関係者の人数に関わらずおそらく必要なことだろう。しかし、チームが優れたソフトウェアを作り出す上で、実際に役立つドキュメントを書くのは難題である。

ひとたび永続的な形式になってしまうと、ドキュメントはプロジェクトの進捗とのつながりを失ってしまうことが多い。コードの進化や、プロジェクトで用いられる言語の進化から置いていかれてしまうのだ。

この問題に対処するには、多くのアプローチが使える。具体的なドキュメントは本書の最後の方にある第4部で示すが、これは特定の要求に対応するものであり、プロジェクトで使用すべきドキュメント一式を私が指図するつもりはない。代わりに、ドキュメントを

評価するための一般的な指針を2つ提案しよう。

ドキュメントはコードや会話での表現を補わなければならない

各アジャイルプロセスは、ドキュメントについて独自の哲学を持っている。エクストリームプログラミングは、余分な設計ドキュメントを一切使用しないで、コード自体に語らせることを主張する。動作しているコードが嘘をつくことはないが、他のドキュメントだとその可能性がある。動作しているコードのふるまいには、あいまいなところがないのだ。

エクストリームプログラミングは、プログラム中の実際に動作する要素と実行可能なテストに対してのみ集中する。コードにつけられたコメントでさえも、プログラムのふるまいには影響を与えないので、動作するコードやそれを駆動するモデルと同期しなくなるのが常だ。外部のドキュメントや図もプログラムのふるまいに影響しないから、やはり同期しなくなる。その一方で、口頭でのコミュニケーションやホワイトボード上のすぐに消される短命な図は、混乱を招くほどには長く残らない。このように、コミュニケーションの媒体としての機能をコードに頼れば、開発者はコードをクリーンで透過的に保とうという気になる。

しかし、設計ドキュメントとしてのコードにも限界はある。詳細によって読者を圧倒しかねないのだ。ふるまいにあいまいなところがなくても、それは理解しやすいということではない。しかも、ふるまいの背景にある意味を伝えるのが困難なこともある。言い換えると、コードだけでドキュメントを作成することには、包括的なUML図を使うのと同様の、基本的な問題がいくつかあるのだ。もちろん、充実した口頭でのコミュニケーションをチーム内で行うことにより、コードにコンテキストと方針をもたらすことはできるが、そうしたコミュニケーションは短命で局所的である。それに、開発者だけがモデルを理解できればよいというわけでもない。

すでにコードがうまくやっていることを、ドキュメントでもやろうとするべきではない。詳細については、すでにコードが提供している。コードというものは、プログラムのふるまいに関する正確な仕様なのだ。

意味をはっきりと説明し、大規模な構造への洞察を与え、コアとなる要素に意識を集中させる必要のあるドキュメントも存在する。プログラミング言語が概念を直接的に実装できない時には、ドキュメントによって、設計の意図を明確にすることができる。書かれたドキュメントは、コードと会話を補足すべきである。

ドキュメントは活動の役に立たなければならず、最新の状態を保たなければならない

モデルをドキュメントに書き出す場合、私は、そのモデルの厳選した小さなサブセットを図に描き、その周りにテキストを書く。クラスとその責務を言葉で定義し、意味を支えるコンテキストの中に囲い込む。こうしたことは、自然言語でしかできないからだ。しかし、図が示すのは、概念を形式化し、余分なものを削り落としてオブジェクトモデルにす

る際に行われた、いくつかの選択である。これらの図は形式張っておらず、手書きのことさえある。手書きの図は、労力を節約できるし、形式張らないその場限りのものであるという**印象**を与えるという利点も持っている。そういう図はコミュニケーションに向いている。形式張らないその場限りのものというこの性質は、我々のモデルに対する考え方にも当てはまるからだ。

設計ドキュメントの持つ最大の価値は、モデルの概念を説明すること、コードの詳細を書き進める上で役に立つこと、そしておそらく、モデルで想定されている使用方法について多少の洞察を与えることにある。チームの考え方次第で、設計ドキュメント全体が、スケッチ一式を壁に貼り出せば済むくらいまで簡潔になることもあれば、膨大な量になることもある。

ドキュメントはプロジェクトの活動に取り込まれていなければならない。そうなっているかどうかを判断する最も簡単な方法は、ユビキタス言語とドキュメントとの相互作用を観察することである。ドキュメントは、プロジェクトで（現在）人々が話している言語で書かれているか？ コードに埋め込まれている言語で書かれているか？

ユビキタス言語に耳を傾け、どのように変化していくかを聴こう。設計ドキュメントで説明されている用語が会話とコードに出てきそうもなければ、そのドキュメントは目的を果たしていない。たぶん、長すぎるか複雑すぎるのだろう。重要なテーマに十分に焦点が合っていないのかもしれない。読まれていないか、読むべきだと思われていないのだ。ユビキタス言語に全く影響を与えていないのなら、何か問題がある。

逆に、ユビキタス言語が自然に変化している様子は聴き取れるのに、ドキュメントが取り残されている場合もある。どうやら、そのドキュメントは当面関係ないか、更新するほど重要でないらしい。履歴として保管しておくなら安全かもしれないが、使われ続けているのであれば、混乱を招いてプロジェクトに支障をきたすかもしれない。それに、ドキュメントが重要な役割を果たしていないなら、純然たる意志と規律をもってドキュメントを最新の状態に保つことは、無駄な努力である。

ユビキタス言語によって、要件定義書などの他のドキュメントがより簡潔で明確になる。ドメインモデルがビジネスと最も関連の深い知識を反映するようになると、アプリケーションの要求はそのモデルにおけるシナリオとなり、ユビキタス言語を使うことで、そうしたシナリオはモデル駆動設計（第3章参照）に直結する用語で記述することができる。その結果、仕様をよりシンプルに書けるようになるが、それは、モデルの背景にあるビジネスの知識を伝えなくてもよいからである。

ドキュメントを最小限に留め、その焦点をコードと会話の**補足**に絞ることで、ドキュメントがプロジェクトに結びつけられた状態を保つことができる。ユビキタス言語とその進化を指針として、プロジェクトの活動に織り込まれる生きたドキュメントを選び出すこと。

実行可能な基盤

今度は、XPやその他のコミュニティによる選択肢を検証しよう。これは、実行可能なコードとそのテストにほぼ全面的に依存する。本書の大部分は、モデル駆動設計を通じてコードに意味を伝えさせる方法を論じている（第3章参照）。適切に作成されたコードは表現力に富むが、コードによって伝えられるメッセージが正確である保証はない。一片のコードが引き起こす**ふるまい**こそが現実なのであって、そこからは逃れられないのだ。それでもなお、メソッドの名前は、あいまいだったり、誤解を招いたり、メソッドの内容に比べて古くなっていることがある。テストの中の表明は厳密でも、変数名とコードの構成が伝える内容はそうではない。優れたプログラミングの作法によって、このつながりはできる限り直接的に保たれるが、それは自分を律した結果でしかない。正しいことを**する**だけでなく、正しいことを**言う**コードを書くためには、潔癖でなければならない。

ふるまいと表現の間のこうした食い違いを取り除くことが、宣言的設計（第10章で説明）のようなアプローチの主なセールスポイントである。宣言的設計においては、プログラム要素の目的をあらわす一文が、プログラム中の実際のふるまいを決定する。UMLからプログラムを生成しようという傾向は、これによって動機づけられている部分もある。しかし、一般的に言って、これまでうまくいった試しはない。

だが、誤解を招くことがあるとはいえ、他のドキュメントと比べれば、コードは根本に近い。現在の標準的な技術を使って、コードのふるまい、意図、そしてメッセージを同調させるには、規律を守らなければならないし、設計について特定の考え方をしなければならない（第3部で詳述する）。効果的にコミュニケーションを行うためには、コードは、要件を書くのに使用されたのと同じ言語に基づいていなければならない。これは、開発者がお互い会話したり、ドメインエキスパートと会話したりするのと同じ言語でもある。

説明のためのモデル

本書の要点は、実装、設計、そしてチーム内のコミュニケーションの根底にあるモデルは1つだけであるべきだ、ということである。こうした別々の目的のために別々のモデルを持つのは危険である。

モデルは、ドメインについての教材としても価値を持つことがある。設計を推進するモデルというのは、ドメインに対する見方の1つである。しかし、ドメインについて学ぶ上では、ドメインに関する一般的な知識についてコミュニケーションする際に教材としてのみ使用されるような、別の見方も役に立つ。この目的のためには、画像や言葉の中で、ソフトウェア設計とは関係しない、別の種類のモデルを伝えるものが使用できる。

他のモデルが必要とされる特別な理由の1つはスコープだ。ソフトウェア開発プロセス

を推進する技術的なモデルは、厳密に刈り込み、機能を満たす上で必要な最小限のものにしなければならない。それに対して、説明のためのモデルであれば、ドメインの持つ、スコープがより限定されたモデルをわかりやすく示すコンテキストを提供するような側面を含むことができる。

説明のためのモデルのおかげで、特定の話題に合わせてはるかに多くを伝えるスタイルを作り出す余地が与えられる。ある分野のドメインエキスパートが用いる視覚的なメタファによって、説明はより明確になり、開発者を教育して、エキスパート同士の意見を一致させることができるようになる。また、説明のためのモデルは全く別の方法でもドメインを表現するが、多様な説明は人々の理解を助けるものだ。

説明のためのモデルがオブジェクトモデルである必要は全くなく、むしろ、通常はそうでない方がよい。実際、このモデルではUMLを避け、ソフトウェア設計と一致するという誤った印象を与えないようにした方がよい。説明のためのモデルと設計を推進するモデルは、確かに一致することも多いが、似ていても正確に同じになることはめったにない。混乱を避けるには、すべての人がこの違いを認識していなければならない。

例2.2——輸送業務と経路

輸送会社のために貨物を追跡するアプリケーションを考えよう。このモデルには、港湾業務と輸送機器の運行（vessel voyage）が貨物の業務計画（「経路」）にどうやって組み込まれるかを示す詳細な見解が含まれている。しかし、十分な知識のない人にとって、クラス図はあまりわかりやすくないかもしれない。

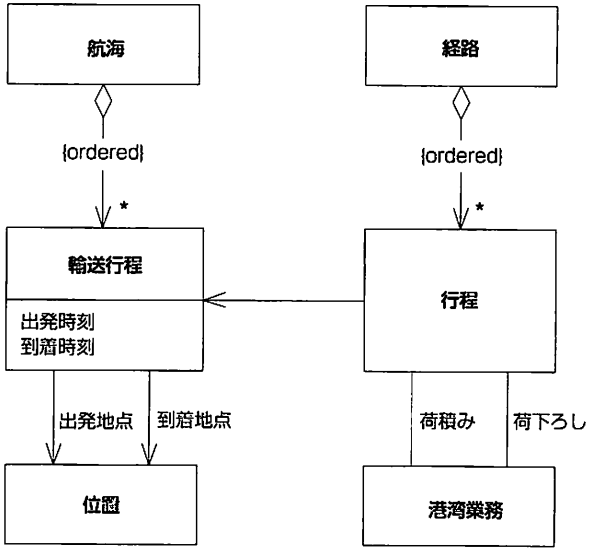


図2.4 輸送経路を表すクラス図

そういう場合、説明のためのモデルを使うことで、クラス図が実際に意味している内容をチームメンバが理解できるようになる。次に、同じ概念の別の見方を示そう。

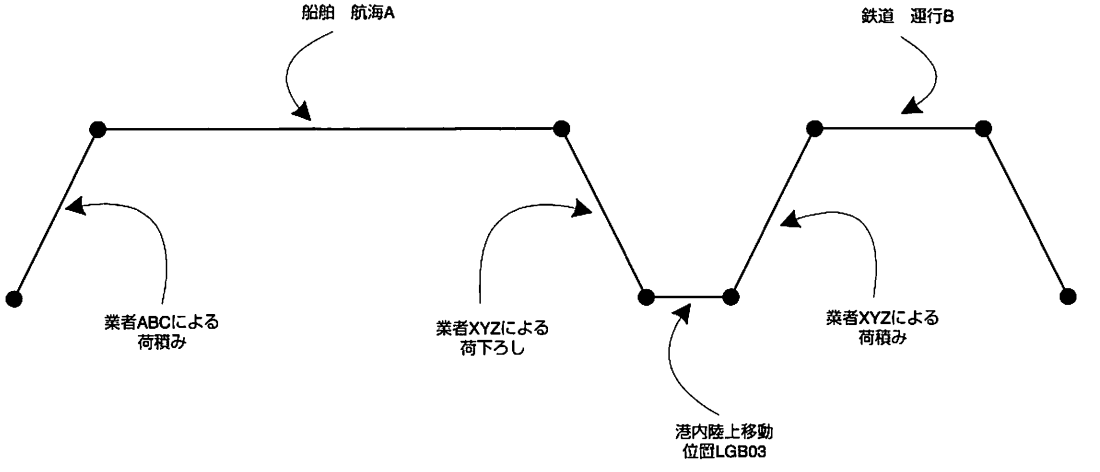


図2.5　輸送経路を表す説明のためのモデル

図2.5にある線はそれぞれ、港湾業務（貨物の積み下ろし）、陸上の倉庫にある貨物、途中の船上にある貨物のいずれかを表している。詳細においてはクラス図と一致していないが、ドメインの要点を補強している。

この種の図は、その図が表すモデルについての自然言語を用いた説明と合わせることで、開発者もドメインエキスパートも、より厳密なソフトウェアのモデル図を理解できるようになる。両方の図を一緒に見た方が、それぞれを単独に見るよりも理解しやすくなるのである。

第3章

モデルと実装を結びつける

部屋に入った時にまず目に入ったのは、すべてが書き込まれたクラス図だった。このクラス図は大量の紙に印刷されて、巨大な壁を覆っていた。これは私にとってプロジェクト参加初日のことだったが、このプロジェクトではすでに頭の切れる人々が何か月もの時間を費やして、ドメインについての詳細なモデルを慎重に調査し、開発していた。そのモデルに含まれる典型的なオブジェクトは、他のオブジェクト3、4個と複雑に関連していて、この関連の網の目には自然な境界がほとんどなかった。その点では、アナリストはドメインの性質に忠実であったと言える。

壁一面の図は圧巻だったが、それなりにモデルは知識をとらえていた。軽く調べただけで私は非常に多くのことを学んだ（ただ、どう学んでいけばよいかはわかりにくく、Webを適当に眺めるのに近かった）。それより困ったのは、自分の調査からは、アプリケーションのコードと設計に対する洞察が全く得られないとわかったことだ。

アプリケーションの実装を始めてすぐに、開発者たちはあることに気づいた。関連がもつれているせいで、人間のアナリストなら辿ることができても、登録や検索のできる単位に変換してトランザクションの整合性を保ちつつ操作することはできなかったのだ。念のために言うが、このプロジェクトはオブジェクトデータベースを使用していたので、開発者たちはオブジェクトを関係テーブルにマッピングさせるという難題と向き合う必要すらなかった。根本的なレベルで、このモデルは実装のための指針を提供しなかったのである。

このモデルは、技術チームのアナリストとビジネスエキスパートが広範囲にわたって協力した結果生まれた「正しい」ものだったので、開発者たちが至った結論は、概念に基づいたオブジェクトを自分たちの設計の基礎に据えることはできないというものだった。そこで、彼らはその場しのぎの設計に取りかかったのである。データストレージでは、モデルにあるのと同じクラス名や属性がいくつか使用されてはいたが、その設計は既存のモデ

ルに基づいてはいなかったし、そもそも何らかのモデルに基づくものではなかった。

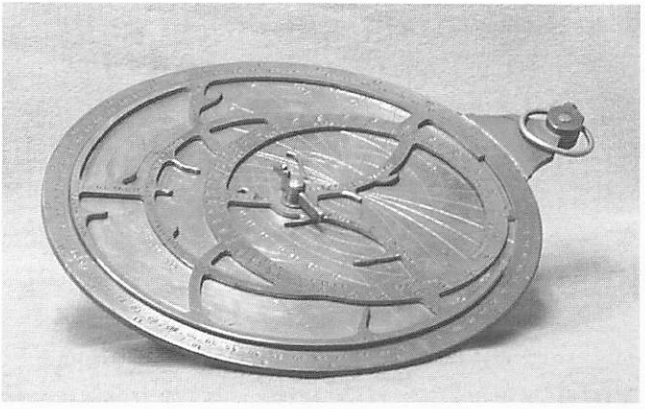
そのプロジェクトには、ドメインモデルがあった。しかし、動作するソフトウェアの開発を直接支援しないなら、紙に書かれたモデルが何の役に立つというのだろう？

数年後、私は全く異なるプロセスから同じ結末に至る様子を目にした。そのプロジェクトは、既存のC++アプリケーションを、Javaで実装される新しい設計に置き換えるというものだった。旧アプリケーションは間に合わせで作成されていて、オブジェクトモデリングは一切考慮されていなかった。旧アプリケーションの設計は、仮に設計と呼べるものがあったとしても、次から次へと機能を融合させて既存のコードの上に積み重ねてきたものだったため、注目に値する一般化も抽象化もなかった。

不気味なことに、これら2つのプロセスの最終成果物は非常によく似ていたのだ！ どちらも機能は備えていたが肥大化していて、実に理解しにくく、結局は保守ができなかったのである。実装上、ある意味わかりやすく書かれているところも場所によってはあったが、コードを読んでも、システムの目的についての洞察はあまり得られなかった。いずれのプロセスも、彼らの開発環境で利用できたはずのオブジェクトパラダイムを全く活かしておらず、ただ、データ構造だけが凝ったものになっていた。

モデルというものは、ソフトウェア開発プロジェクトのコンテキストに限定して述べても多種多様であり、多くの役割を果たしている。ドメイン駆動設計が必要とするモデルは、初期の分析を支援するだけでなく、設計においてまさに土台となる。このアプローチはコードに対して重要な意味合いを持つ。さらに、コードに対するアプローチほどはっきりしていないが、モデリングに対しても、ドメイン駆動設計では別のアプローチが求められるのだ…。

モデル駆動設計（MODEL-DRIVEN DESIGN）



アストロラーベは、天体モデルの機械による実装であり、
星の位置を計算するのに使用された。

コラム――中世の天体観測機
古代ギリシアの天文学者がアストロラーベを考案し、中世イスラムの科学者が完成させた。回転するクモの巣状のもの（リートという）が、天球上の恒星の位置を表していた。ある地域の天球座標系が刻まれたプレートは交換できて、さまざまな緯度からの見方を表していた。リートをプレート上で回転させることによって、1年のうち、どんな日時の天体位置でも計算することができた。反対に、恒星や太陽の位置を与えれば、時刻を計算することができた。アストロラーベは、天体を表すオブジェクト指向モデルを機械によって実装したものだったのである。

コードをその元になっているモデルへ緊密に関係づけることにより、コードに意味が与えられ、モデルがドメインと関連の深いものになる。

＊　＊　＊

ドメインモデルを一切持たず、次から次へと機能を満たすコードを作成するだけのプロジェクトは、これまでの2章で述べた、知識のかみ砕きとコミュニケーションによる恩恵をほとんど受けない。そうしたプロジェクトは、複雑なドメインが現れた時に手も足も出せないだろう。

その一方で、多くの複雑なプロジェクトは、何らかのドメインモデルを試してはいるものの、モデルとコードとの間の緊密な結びつきは維持していない。こうしたプロジェクトで開発されるモデルは、最初は調査のための道具として役立つと思われるかもしれないが、徐々に的外れになっていき、誤解を招くようにさえなる。モデルにどれだけ細心の注意を払っても、設計が正しいという安心はほとんど得られないのだが、それはモデルとコード

が別物だからだ。

この結びつきはさまざまなやり方で断ち切られるかもしれないが、その分離は意識的に選択されていることが多い。設計の方法論の多くは**分析モデル**を作ることを主張する。これは設計とはっきり区別され、通常は別の人々によって開発される。分析モデルと呼ばれるのは、これがビジネスドメインを分析した結果できていて、ソフトウェアシステムで果たす役割を全く考慮せずに概念をまとめ上げるからだ。分析モデルはビジネスドメインを理解するためだけに使うツールとされ、実装の問題を混ぜてしまうとわかりにくくなると考えられている。後の工程で設計が行われるが、分析モデルとの対応はゆるやかなものでしかないかもしれない。分析モデルは設計の問題に留意しないまま作成されるので、設計の際に求められることに対して全く合わない可能性が高いのだ。

こうした分析の際にも多少は知識のかみ砕きが行われるが、コーディングが始まり、開発者が設計のために新しい抽象を考え出さざるを得なくなると、そのほとんどが失われる。そうなれば、アナリストによって得られてモデルに組み込まれた洞察が、維持され、再発見される保証は全くない。これを見る限り、設計と、それにあいまいにしか対応していないモデルとの間の紐づけを維持することは、費用対効果が低いのだ。

純粋な分析モデルでは、ドメインを理解するという第一の目標にさえも到達できない。重大な発見はいつでも、設計や実装をするために努力する際に現れるからだ。非常に特殊で予期せぬ問題というものが常に発生する。あらかじめ作られるモデルは、的外れな対象を掘り下げる一方で、重要な問題を見過ごしてしまうのである。また、アプリケーションの役には立たないかたちで問題が表現されることもある。その結果、純粋な分析モデルはコーディングが始まるとまもなく捨て去られ、問題の大半は再び取り上げざるを得なくなる。しかし、2回目のイテレーションで、開発者が分析は別のプロセスだと思ってしまうと、モデリングは規律正しいやり方で行われなくなる。マネージャが分析は別のプロセスだと思ってしまうと、開発チームがドメインエキスパートと十分に接する機会が与えられなくなってしまうかもしれない。

原因が何であれ、設計の根本において概念が欠如しているソフトウェアは、せいぜい、役に立つことは行えても、その行為を説明することができない機械仕掛けにすぎない。

設計が、あるいは設計の中心となる部分が、ドメインモデルに紐づいていないならば、そのモデルにほとんど価値はなく、ソフトウェアが正確であるかどうかも疑わしい。同じように、モデルと設計された機能との紐づけが複雑だと理解するのが難しく、実際には、設計が変更された時に紐づけを維持できなくなる。分析と設計の間に致命的な亀裂が生じるため、それぞれの作業で得られる洞察は互いに活かされることがないのだ。

分析は、ドメインから得られる基本的な概念をわかりやすく表現力豊かな方法でとらえなければならない。設計は、コンポーネントの集合を定義しなければならない。そうした

コンポーネントは、プロジェクトで使用されるプログラミングツールを使って構築できて、デプロイ先の環境で効率的に実行され、アプリケーションに課せられた問題を正しく解決できなければならない。

モデル駆動設計は、分析モデルと設計という二分法を捨て去り、両方の目的に使える単一のモデルを探し出す。純粋に技術的問題はさておき、設計にあるオブジェクトはそれぞれモデルで記述されている概念的な役割を果たすことになる。そのため、我々は選択したモデルに対する要求をもっと厳しくしなければならない。そのモデルは全く別々の2つの目標を満たす必要があるからだ。

ドメインを抽象化する方法は常に数多く存在し、アプリケーションの問題を解決できる設計も常に数多く存在する。だから、モデルと設計を結びつけることは現実的に可能なのだ。この結びつきの代償として、分析が、技術的な配慮から致命的に妥協した貧弱なものになってはならない。同様に、ドメインに対する考え方を反映してはいるが、ソフトウェア設計の原則を避けた出来の悪い設計も、受け入れることはできない。モデル駆動設計のアプローチは、分析としても設計としてもうまく機能するモデルを要求する。モデルが実装にとって実用的でないと思われる場合も、モデルがドメインの主要な概念を忠実に表現していない場合も、新しいモデルを探さなければならない。こうして、モデリングと設計のプロセスは、反復されるただ1つのループになるのだ。

ドメインモデルを設計と密接に関連づけよという命題により、考えられるすべてのモデルの中から、より使えるものを選ぶための基準が1つ追加される。精力的な思考が求められ、通常は多数のイテレーションと大量のリファクタリングが必要になるが、それによってモデルは**適切な**ものになるのである。

それゆえ：

ソフトウェアシステムの一部を設計する際には、紐づけが明らかになるように、ドメインモデルを文字通りの意味で忠実に反映させること。モデルについて再検討し、より自然にソフトウェアに実装されるように修正すること。これは、ドメインに対するより深い洞察を反映させようとする時にも言える。強固な*ユビキタス言語*を支えることに加えて、ドメインと実装両方の目的に使える単一のモデルを要求すること。

設計で使用する用語法と責務の基礎的な割り当てをモデルから引き出すこと。コードはモデルの表現となるから、コードに対する変更はモデルに対する変更になるかもしれない。その影響は、プロジェクトの他の活動全体へと適宜伝わっていかなければならない。

実装を一分の狂いもなくモデルに結びつけるには、通常、オブジェクト指向プログラミングのようなモデリングパラダイムをサポートする、ソフトウェア開発のためのツールと言語が必要である。

別々のサブシステムに別々のモデルが存在することもあるが（第14章参照）、システム

の特定の部分に対しては、開発作業のあらゆる局面、すなわち、コードから要求の分析に至るまで一貫して、適用されるモデルは1つだけであるべきだ。

モデルが1つであることで、間違いが発生する可能性が低くなる。今や設計は、慎重に検討されたモデルから直接的に生み出されるからである。設計、さらにコード自体までもが、モデルを伝達するという性質を有することになる。

＊ ＊ ＊

モデルを1つだけ開発して、それによって問題をとらえつつ、実用的な設計を提供するということは、言葉で言うほど簡単ではない。任意のモデルを単純に取り上げただけでは、それを有効な設計に変えることはできないのだ。モデルは注意深く作り上げて、実用的な実装に役立つようにしなければならない。コードがより効果的にモデルを表現できるような、設計と実装技術を採用しなければならない（第2部参照）。知識をかみ砕く者はモデルの選択肢を調べつくし、それを実用的なソフトウェア要素に磨き上げる。開発は、モデルと設計およびコードを単一の活動として改良し続けるイテレーティブなプロセスとなる（第3部参照）。

モデリングパラダイムとツールによるサポート

モデル駆動設計を成功させるには、モデルと設計は文字通りに対応し、人が綴り間違いを犯さない限り正確でなければならない。モデルと設計をそのように厳密に一致させる上でほぼ必須となるのは、作業に用いるモデリングパラダイムが、モデルに含まれる概念と直接的に類似したものを生成できるソフトウェアツールによってサポートされるということだ。

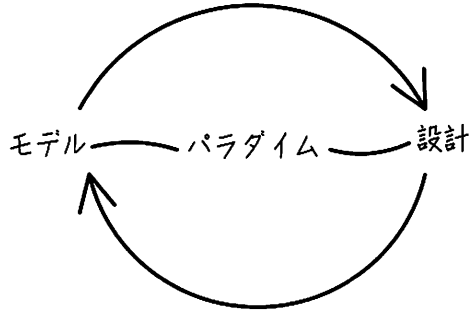


図3.1

オブジェクト指向プログラミングが強力なのは、あるモデリングパラダイムに基づいていて、モデルを構成する概念に実装を提供するからだ。プログラマから見る限り、オブジェクトは実際にメモリ上に存在していて、他のオブジェクトと関連を持ち、クラスとして構成されている。また、メッセージングによって呼び出せるふるまいも提供する。多くの開発者が、オブジェクトの持つ、プログラムコードをまとめるという技術面の性質だけを適用することで恩恵を受けている。しかし、オブジェクト設計の真のブレイクスルーが訪れるのは、コードがモデルにある概念を表現した時なのだ。Javaや他の多くのツールを使うことで、オブジェクトやそれらの関係性を作成する際に、概念的なオブジェクトモデルと直接的に類似したものにできる。

オブジェクト指向言語ほど広く使用されるには至らなかったものの、Prolog言語もモデル駆動設計と自然に適合する。この場合、パラダイムにあたるものは論理で、モデルは論理学上のルール群とそれらが作用するファクトの集合となる。

モデル駆動設計も、Cなどの言語を使った場合には、限定的にしか適用できないが、これは純粋な**手続き型**言語と対応するモデリングパラダイムがないからだ。こうした言語が手続き型とされるのは、プログラマがコンピュータに対して、従うべき一連のステップを指示することによる。プログラマがドメインの概念について考えていたとしても、プログラム自体は、データに対して行われる一連の技術的な操作でしかないのだ。操作による結果は役に立つかもしれないが、プログラムは意味のほとんどをとらえ損ねている。手続き型言語は複雑なデータ型をサポートすることもあり、そうしたデータ型には、ドメインにおけるより自然な概念との一致が見られ始めている。しかし、この複雑な型はデータをまとめたものでしかなく、ドメインの活動的な側面をとらえることはない。結果として、手続き型言語で書かれたソフトウェアにある入り組んだ関数は、想定される実行パスに基づいてつなぎ合わされるだけで、ドメインモデルにおける概念的なつながりを反映することがないのだ。

オブジェクト指向プログラミングという言葉をまだ聞いたこともなかったころ、私は数学のモデルを解決するのにFORTRANのプログラムを作成した。この数学のモデルは、まさにFORTRANが秀でている類のドメインだった。このようなモデルにおける主要な概念的コンポーネントは数学的関数であり、FORTRANですっきりと表現できる。ただ、そうであっても、関数の先にある高次の意味をとらえるすべはない。数学的でないドメインのほとんどは、手続き型言語で書かれると、モデル駆動設計が実現できない。そうしたドメインは、数学的関数や、手続きにおけるステップとして概念化されないからだ。

オブジェクト指向設計は、現在、大掛かりなプロジェクトの多くで採用されているパラダイムであり、本書でも主にこのアプローチを使用する。

例3.1 ——手続き型からモデル駆動へ

第1章で述べたように、プリント基板（PCB）は、さまざまなコンポーネントのピンをつなぐ導体（ネットと呼ばれる）の集まりと見なせる。何万ものネットが存在することもしばしばある。PCBレイアウトツールと呼ばれる特殊なソフトウェアは、ネットが相互に交差や干渉をしないようにしつつ、すべてのネットを物理的に配置する方法を見つけ出す。これは、経路を最適化しつつ、人の設計した、レイアウトを限定する大量の制約を満たすことで実現される。PCBレイアウトツールは非常に洗練されているが、それでもいくつか欠点がある。

問題の1つは、この何千というネットそれぞれに独自のレイアウトルールがあることだ。PCBエンジニアは、多くのネットを、同じルールを共有すべき自然なグループに帰属するものととらえる。例えば、ネットにはバスを形成するものがある。

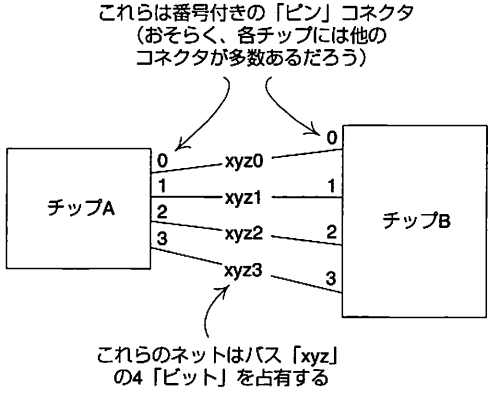


図3.2 バスとネットの説明図

ネットをまとめてバスにする際、一度に、おそらく8か16、または256の単位で行うことで、技術者たちは仕事をもっと扱いやすい量に切り詰め、生産性を向上させてエラーを減らす。問題は、レイアウトツールにバスのような概念がないことである。そのため、何万ものネットに対してルールを指定しなければならない。それも1回に1つずつだ。

機械的設計

物量に絶望した技術者たちは、スクリプトを作成することで、レイアウトツールの制約に対処した。これは、レイアウトツールのデータファイルを解析し、そのファイルにルールを直接書き込んで、バス全体に対して一度に適用するというものだった。

このレイアウトツールは、それぞれの回路の接続をネット一覧ファイルに格納している。これは次のようなものだ。

```
ネット名      コンポーネント.ピン
------      --------------
Xyz0        A.0, B.0
Xyz1        A.1, B.1
Xyz2        A.2, B.2
...
```

レイアウトルールは、次に示すファイル形式で格納されている。

```
ネット名      ルールタイプ          パラメータ
------      -------------     ---------
Xyz1        min_linewidth     5
Xyz1        max_delay         15
Xyz2        min_linewidth     5
Xyz2        max_delay         15
...
```

技術者たちは、ネットの命名規約を注意深く使用することにより、データファイルのアルファベット順ソートによってバスにまとめられた各ネットが、ソート後のファイルへ一緒に入るようにしている。そうすれば、スクリプトがファイルを解析し、バスに基づいて各ネットを修正できるのだ。ファイルの解析と操作および書き込みを行う実際のコードは、ここで例に挙げるにはあまりにも冗長で不明瞭だから、手続きのステップを一覧化するだけにしておく。

1．ネット名によりネット一覧ファイルをソートする．
2．ファイルの各行を読み込み、バス名のパターンで始まる最初の行を探す．
3．一致する名前を持つ各行に対し、行を解析してネット名を取得する．
4．ネット名にルールを示すテキストをつけて、ルールファイルへ追記する．
5．残っている行がバス名と一致しなくなるまで、3以降を反復する．

したがって、以下のようなバスルールが入力された場合、

```
バス名        ルールタイプ          パラメータ
-----       ---------         -------
Xyz         max_vias          3
```

次のようなネットルールをファイルに追記することになる。

```
ネット名      ルールタイプ          パラメータ
------      --------          -------
...
Xyz0        max_vias          3
Xyz1        max_vias          3
Xyz2        max_vias          3
...
```

想像するに、こういうスクリプトを最初に作成した人は、このようなシンプルなものしか必要としなかったのだろうし、これが唯一の要求であったなら、こういうスクリプトも十分に意味をなしただろう。しかし、実際には、今や何十ものスクリプトが存在するのだ。もちろん、

ソートや文字列の突き合わせを行う関数を共有するようにリファクタリングすることはできたし、詳細をカプセル化する関数呼び出しを言語がサポートしていれば、スクリプトは、これまでで要約したステップのようなものとして読み始められただろう。だが、それでも、それらのスクリプトは単なるファイル操作にすぎない。ファイル形式が異なれば（しかも複数ある）、パスのグループ分けとそれに対するルールの適用という考え方が同じであっても、一から始めなければならない。より豊かな機能や双方向性が欲しければ、隅から隅まで自分でまかなう必要がある。

スクリプトの作成者が試みたのは、ツールの持つドメインモデルに「パス」の概念を補うということだった。実装上、ソートと文字列の突き合わせを通じて、パスの存在が暗示されてはいるが、その概念が明示的に扱われてはいない。

モデル駆動設計

これまでの議論で、すでに、ドメインエキスパートが問題について考える際に使用する概念を説明してきた。今度は、その概念をソフトウェアの土台にできるモデルへと、明示的にまとめ上げなければならない。

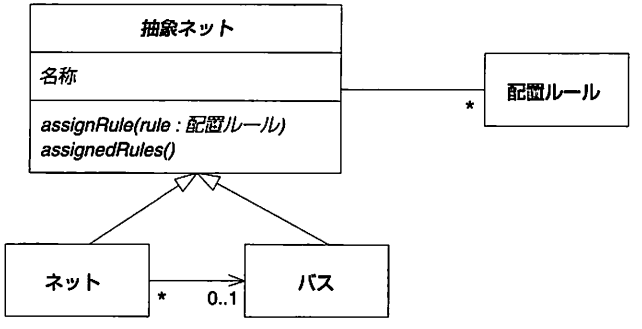


図3.3 レイアウトルールの効率的な割り当てを目指したクラス図

これらのオブジェクトをオブジェクト指向言語で実装すると、コアとなる機能は実に平凡になる。

`assignRule()`メソッドは、**抽象ネット**に実装できる。ネットの`assignedRules()`メソッドは、それ自体のルールと、パスのルールを取得する。

```
/**
 * 抽象ネット.
 */
abstract class AbstractNet {
// 配置ルール
   private Set rules;
```

```
    /**
     * ルールを割り当てる.
     * @param rule 配置ルール
     */
    void assignRule(LayoutRule rule) {
        rules.add(rule);
    }

    /**
     *割り当てられたルールを取得する.
     * @return 配置ルール
     */
    Set assignedRules() {
        return rules;
    }
}

/**
 * ネット.
 */
class Net extends AbstractNet {
    // バス
    private Bus bus;

    /**
     * 割り当てられたルールを取得する.
     * @return 配置ルール
     */
    Set assignedRules() {
        Set result = new HashSet();
        result.addAll(super.assignedRules());
        result.addAll(bus.assignedRules());
        return result;
    }
}
```

もちろん、他に補助的なコードは大量にあるだろうが、これでスクリプトの基本的な機能は網羅している。

このアプリケーションはインポートとエクスポートのロジックを必要とするが、それは2つのシンプルなサービスにカプセル化することにする。

サービス	責務
ネット一覧インポート	ネット一覧ファイルを読み込み、各エントリに対してネットのインスタンスを作成する
ネットルールエクスポート	ネットの集合が与えられると、添付されたルールをすべてルールファイルに書き込む

また、ユーティリティもいくつか必要だ。

クラス	責務
ネットリポジトリ	名前を指定して、ネットにアクセスできるようにする
推測結果バスファクトリ	ネットの集合が与えられると、命名規約を使用してバスを推測し、インスタンスを作成する
バスリポジトリ	名前を指定して、バスにアクセスできるようにする

さて、アプリケーションを開始するには、インポートされたデータを使ってリポジトリを初期化すればよい。

```
Collection nets = NetListImportService.read(aFile);
NetRepository.addAll(nets);
Collection buses = InferredBusFactory.groupIntoBuses(nets);
BusRepository.addAll(buses);
```

サービスとリポジトリは、それぞれユニットテストが可能である。それよりも重要なのは、中核となるドメインロジックがテスト可能だということだ。次に示すのは、最も中心的なふるまいに対するユニットテストである（JUnitテストフレームワークを使用）。

```
public void testBusRuleAssignment() {
   Net a0 = new Net("a0");
   Net a1 = new Net("a1");
   Bus a = new Bus("a"); // 概念上、バスは名前を基準とした識別に
   a.addNet(a0);         // 依存していないから、
   a.addNet(a1);         // テスト時にも依存してはならない

   NetRule minWidth4 = NetRule.create(MIN_WIDTH, 4);
   a.assignRule(minWidth4);

   assertTrue(a0.assignedRules().contains(minWidth4));
   assertEquals(minWidth4, a0.getRule(MIN_WIDTH));
   assertEquals(minWidth4, a1.getRule(MIN_WIDTH));
}
```

対話式のユーザインタフェースであれば、バスの一覧を見せて、それぞれに対してユーザがルールを割り当てられるようにすることも、後方互換性のためにルールを記したファイルから読み込むようにすることもできる。ファサードを置くことで、いずれのインタフェースからもシンプルにアクセスできるようになる。ファサードの実装は前述のテストと似たものになる。

```
public void assignBusRule(String busName, String ruleType,
      double parameter){
   Bus bus = BusRepository.getByName(busName);
   bus.assignRule(NetRule.create(ruleType, parameter));
}
```

最後は次の通りである。

```
NetRuleExport.write(aFileName, NetRepository.allNets());
```

（このサービスは各ネットに対してassignedRules()を呼び出し、結果を完全に展開して書き込む。）

もちろん、操作が（この例のように）1つしかなければ、スクリプトを基にしたアプローチでも十分実用的だろう。しかし、現実には20以上あったのだ。モデル駆動設計は規模を容易に拡大させることができ、ルールを結合する上での制約や、その他の機能追加を含めることができる。

モデル駆動設計ではテストもやりやすくなる。この設計のコンポーネントには適切に定義されたインタフェースが存在し、そのインタフェースに対してユニットテストを行える。スクリプトをテストするには、全体を流してファイルの入力と出力を比較するよりほかにない。

忘れないで欲しいのだが、こうしたモデルに基づく設計は一度に現れるものではない。ドメインの重要な概念を蒸留して、シンプルで鋭いモデルにするには、リファクタリングと知識のかみ砕きを何度か反復しなければならないのだ。

骨格を見せる：なぜモデルがユーザにとって重要なのか？

理論上は、システムの根底に何があっても、ユーザに対しては任意のかたちで見せることができるだろう。だが、実際には食い違いがあり、その食い違いは、よくても混乱を、最悪の場合にはバグを引き起こす。非常にシンプルな例を使い、モデルが押しつけられたせいで、ユーザがどう誤解することになるかを考えよう。Microsoft社インターネットエクスプローラの現リリースにおける、Webサイト用ブックマークのモデルを題材とする[1]。

インターネットエクスプローラのユーザは、「お気に入り」のことを、Webサイトの名前の一覧で、ブラウザの利用をまたいで存続していると考える。しかし、実装上はお気に入りを、URLを保持したファイルとして扱っており、そのファイル名がお気に入り一覧に入れられているのだ。WebページがWindowsのファイル名で禁じられている文字を含んでいる場合、これは問題となる。ユーザがお気に入りを保存しようとして、「怠惰:幸せになるための秘訣」という名前を打ち込んだとしよう。すると、「ファイル名に以下の文字を含めることはできません：\ / : * ?」というエラーメッセージが表示される。どのファイル名だと言うのだ？ 一方、Webページタイトルに禁止文字がすでに含まれている場合、インターネットエクスプローラはエラーを出さずにその文字を取り除く。データが失われても、ここでは支障がないかもしれないが、ユーザが想定しているふるまいではない。何も通知せずにデータを変更することは、ほとんどのアプリケーションにおいて断じて許されない。

1 この例はBrian Marickが話してくれたものである。

モデル駆動設計は、(どんな単一のコンテキストにおいても) ただ1つのモデルを用いて作業することを求める (第14章で議論する)。本書にある助言と具体例のほとんどは、分析モデルと設計モデルが別々に作られることで生じる問題に行き着く。だが、ここで示しているのは、それとは別のモデルの対、すなわち、ユーザモデルと設計・実装モデルから生じる問題だ。

もちろん、ドメインモデルを飾らずに見せても、ほとんどの場合は、ユーザにとって都合がよいものにならない。しかし、ユーザインタフェースの中にドメインモデル以外のモデルの幻想を作り出そうとすれば、その幻想が完璧でない限り、混乱が生じるものだ。Webのお気に入りが、実際は単なるショートカットファイルの集まりにすぎないなら、その事実をユーザに明かし、混乱を招くような代わりのモデルを排除しなければならない。機能がわかりやすくなるだけでなく、ユーザはファイルシステムに関する自分の知識を利用して、Webのお気に入りを扱えるようにもなる。例えば、お気に入りを構成し直すのに、Webブラウザに組み込まれた扱いにくいツールを使うのではなく、ファイルエクスプローラを用いることができる。ドメインモデルの情報を与えられることで、ユーザは、ファイルシステムのどの場所にもWebショートカットを保存できるという柔軟性を、さらに活用できるようになる。誤解を招く余分なモデルを削除するだけで、アプリケーションの持つ力が増大し、より明白にもなる。プログラマが前のモデルで十分だと思っているのに、どうしてユーザに新しいモデルを習得させる必要があるのだろうか?

別の方法として、お気に入りを違うかたち、つまりデータファイルにも保存し、独自のルールの対象とすることもできる。そのルールは、おそらく、Webページに適用される命名規約になるだろう。その場合も、単一のモデルが提供されることになる。このモデルがユーザに伝えるのは、Webサイトの命名について知っていることが、すべてお気に入りにも適用されるということだ。

設計が基づくモデルが、ユーザとドメインエキスパートの基本的な関心事を反映していれば、他の設計アプローチを用いた場合に比べて、その設計の骨格をよりはっきりとユーザにさらけ出すことができる。モデルが明らかになれば、ユーザはソフトウェアの潜在能力にもっと触れられるようになり、ふるまいは一貫した予測可能なものになる。

実践的モデラ（HANDS ON MODELERS）

製造業はソフトウェア開発を表すのによく使われるたとえだ。このメタファから推測できるのは、次のようなことである。すなわち、高度なスキルを持つ技術者が設計し、それほどスキルのない労働者が製品を組み立てる。このメタファは数多くのプロジェクトを台無しにしてきたが、理由は1つ、単純なことである。すなわち、ソフトウェア開発は、すべてが設計なのだ。どんなチームでも、メンバに対して専門の役割が与えられるものだが、分析、モデリング、設計、プログラミングのように責任を過剰に分離することは、モデル駆動設計の妨げになる。

あるプロジェクトにおいて、私は、いろいろなアプリケーションチームを調整して、設計を推進するドメインモデルの開発を助けるという仕事を任された。しかし、経営陣の考えでは、モデラはモデリングを行うべきで、コーディングはそうしたスキルの無駄遣いとされたため、私がプログラミングしたり、プログラマと詳細を検討したりすることは、事実上許されなかった。

しばらくの間、事態は順調に見えた。ドメインエキスパートやいろいろなチームの開発リーダと協力して、我々は知識をかみ砕き、コアとなる優れたモデルに磨きをかけたのである。だが、そのモデルが利用されることはなかった。理由は2つある。

第一に、モデルの持つ意図の一部が、引き継ぎの際に失われてしまった。モデルがもたらす全体的な効果は、詳細の影響を非常に受けやすいこともあり（第2部および第3部で詳述する）、その詳細はUML図や通常の議論で必ずしも出てくるとは限らない。私が腕まくりをして他の開発者と直接協力することができ、手本とすべきコードを提供して、密接に支援をしていたら、チームはモデルの抽象を理解し、うまくつきあっていくことができただろう。

もう1つの問題は、モデルが実装や技術と相互作用したことで生じるフィードバックを、直接得られなかったことだ。例えば、モデルのある側面が、その時使用していた技術プラットフォームではひどく非効率だとわかっても、それによって引き起こされるすべての結果は、私の元まで何か月も伝わってこなかった。比較的小さい変更をすれば問題を解決できたのだが、私が知る頃には、そこは重要ではなくなっていた。開発者たちは確実に動くソフトウェアをどんどん書き進めていた。だが、その際にモデルは用いられておらず、何らか使われる場所があっても、ただのデータ構造となってしまっていた。開発者は不要なものを除こうとして、大事なモデルまで失ってしまったのだ。しかし、彼らに他に何ができただろうか？ 彼らはもはや、象牙の塔のアーキテクトの命令を背負い込む危険を冒すことはできなかったのである。

このプロジェクトの初期の状況は、手を動かさないモデラにとっては、いつものように好ましいものだった。私はそのプロジェクトで使用されている技術の大半について、すで

に実践経験をかなり積んでいた。担当が変更になる前は、同じプロジェクトで少人数の開発チームを率いたことさえあったから、その開発プロセスとプログラミング環境も熟知していた。そうした要因があっても、私が効果的に仕事をするのに十分ではなかった。モデラが実装から引き離されてしまっていたからだ。

コードを作成する人々がモデルに責任を感じていない場合や、アプリケーションのためにモデルを機能させる方法を理解していない場合、そのモデルはソフトウェアと無関係になってしまう。コードを変更するとモデルも変わることを、開発者が認識していない場合は、開発者によるリファクタリングによって、モデルは力を増すことなく、むしろ力を失うことになる。一方、モデラが実装プロセスから切り離されている場合、実装に伴う制約に対する感覚を、モデラは習得することがないか、できてもすぐに失ってしまう。モデル駆動設計においては、モデルが、効果的な実装を支えながら、ドメインについての主要な知識を抽象化する。だがこのような場合には、その基本的な制約の半分は失われ、結果としてできるモデルは実用的ではなくなる。最後に、仕事が分割されることで、モデル駆動設計のコーディングに伴う機微を伝えるような共同作業が妨げられてしまうと、熟練の設計者が持つ知識とスキルが、他の開発者に伝達されることがなくなってしまう。

実践的モデラが必要とされるからといって、チームメンバが専門の役割を持てないという意味ではない。エクストリームプログラミングを含むすべてのアジャイルプロセスは、チームメンバの役割を定義しているし、それ以外の非公式な専門も、自然に生まれる傾向がある。問題が起きるのは、モデル駆動設計において結びつけて考えられている2つの作業、すなわち、モデリングと実装を分離してしまった時なのだ。

全体的な設計が効果的かどうかは、適切な粒度の設計や実装上の決定が持つ品質と一貫性に、非常に影響されやすい。モデル駆動設計を用いた場合、コードの一部はモデルの表現となるから、そのコードを変更すると、モデルを変更することになる。好むと好まざるとに関わらず、プログラマはモデラでもある。したがって、プロジェクトの体制を整えて、プログラマがモデリングの作業を適切に行えるようにした方がよい。

それゆえ：

モデルに貢献する技術的な人はだれでも、一定の時間をコードに触れることに費やさなければならない。プロジェクトで主に果たしている役割が何であれ、そうしなければならないのだ。コードの変更に対して責任を負う人はだれでも、コードを通してモデルを表現することを習得しなければならない。すべての開発者は、モデルに関する議論にいずれかの段階で参加して、ドメインエキスパートと話をしなければならない。その他の方法で寄与する人々は、ユビキタス言語を通じてモデルに対する考え方をダイナミックに交換する際に、コードに触れる人々を意識して巻き込まなければならない。

＊ ＊ ＊

モデリングとプログラミングを厳密に分離するとうまくいかない。しかし、巨大なプロジェクトには、高次の設計とモデリングを調整して、最も困難であったり重要であったりする意思決定を助けるようなプロジェクトリーダが依然として必要である。第4部「戦略的設計」ではそういう意思決定を扱っているので、レベルの高い技術者の役割と責任を定義するための、より生産的な方法に関する考えが刺激されるだろう。

ドメイン駆動設計は、モデルがアプリケーションの問題を解決できるようにする。知識をかみ砕くことで、チームは、混沌とした情報のとめどない流れを実用的なモデルへと蒸留する。モデル駆動設計は、モデルと実装を密接に結びつける。ユビキタス言語はそれらすべての情報のための水路であり、開発者、ドメインエキスパート、ソフトウェアの間で情報がよどみなく流れるようにしてくれる。

その結果できるのは、コアドメインに対する基本的な理解に基づいた、豊富な機能を提供するソフトウェアである。

これまで述べてきたように、モデル駆動設計が成功するかどうかは、詳細な設計上の意思決定に影響されやすい。こうした意思決定が、この後に続く数章の主題である。

第2部　モデル駆動設計の構成要素

第4章　ドメインを隔離する 65
第5章　ソフトウェアで表現されたモデル 79
第6章　ドメインオブジェクトのライフサイクル 121
第7章　言語を使用する：応用例 163

ソフトウェアの実装を単純明快なものにしつつ、モデルと歩調の合った状態に保つためには、たとえ現実がひどいものでも、モデリングと設計のベストプラクティスを適用しなければならない。本書は、オブジェクト指向設計の解説書でもなければ、急進的な設計原理を提案することもない。ドメイン駆動設計は、伝統的ないくつかの考え方を基に、強調する点をずらしているのだ。

ある種の意思決定をすることによって、モデルと実装の協調が保たれ、互いがもう一方の効果を高めるようになる。このように協調するためには、個々の要素の詳細に注目する必要がある。こういう規模の小さい要素を慎重に作り上げることで、開発者は安定したプラットフォームを手に入れることができる。このプラットフォームに基づいて、第3部および第4部で解説するモデリングアプローチを適用することになるのだ。

本書の設計スタイルは、「責務駆動設計（responsibility-driven design）」の原則に従うところが大きい。この原則は、1990年にWirfs-Brockらによって提起され、2003年に改訂された。また、『オブジェクト指向入門』（Meyer 1988）で説明されている「契約による設計（design by contract）」の考え方からも、（特に第3部で）多くを引き継いでいる。また、オブジェクト指向設計のベストプラクティスとして広く受け入れられ、『実践UML』（Larman 1998）といった書籍において説明されている、その他の一般的な背景とも、本書は矛盾しない。

プロジェクトが障害に突き当たると、その障害の大小に関係なく、開発者は、こうした設計の原則が当てはまらないように見える状況に自分たちが陥ったと思うかもしれない。ドメイン駆動設計のプロセスを回復力のあるものにするためには、開発者は広く知られた基本原則が、モデル駆動設計を**どのように**支えているかを理解する必要がある。それによって道を外れることなく、折衷案を考え出すことができるのだ。

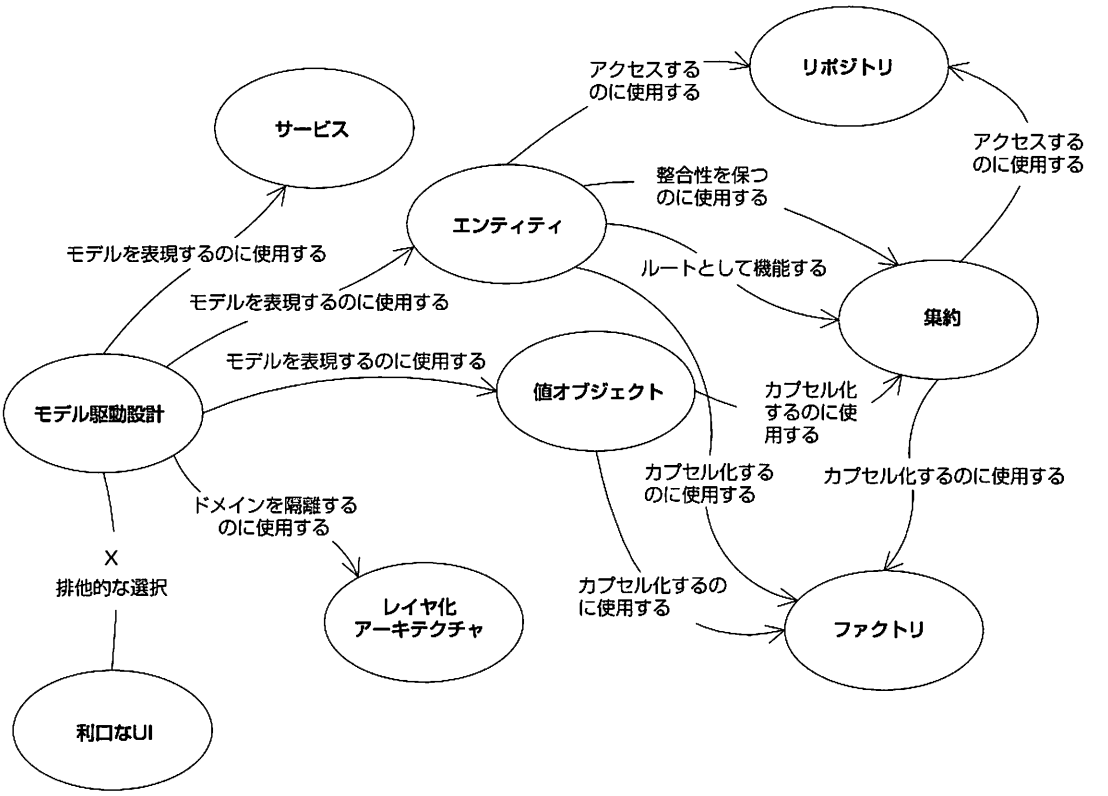
これからの3つの章の内容は、「パターンランゲージ」（付録参照）として構成されており、そこでは、モデルのわずかな違いと設計における意思決定が、ドメイン駆動設計のプロセスにどれほど影響を与えるかが示される。

次ページの図は**ナビゲーションマップ**である。ここで登場するパターンと、パターンの相互の関係をいくつか示している。

これらの標準パターンを共有すれば、設計に秩序がもたらされ、チームメンバが互いの仕事を理解しやすくなる。標準パターンを使うことで、ユビキタス言語も成長する。この言語は、チームメンバ全員が、モデルと設計上の意思決定について議論するのに使用できる。

優れたドメインモデルを開発することは芸術である。しかし、モデルの個々の要素を実際に設計し、実装することは、比較的体系立てて行える。ドメインの設計を、ソフトウェアシステムにおけるその他の大量の関心事から分離することで、設計とモデルとのつながりが非常に明確になる。特定の区別に従ってモデル要素を定義することで、モデル要素の

意味が鮮明になる。個々の要素に対して実績のあるパターンに従うことで、実際に実装できるモデルを作れるようになる。



モデル駆動設計を構成する言語のナビゲーションマップ

手の込んだモデルは複雑さを断ち切ることができるが、ただし、それは基本に注意が払われた時だけだ。その結果として得られる詳細な要素を、チームは自信を持って組み合わせることができるのである。

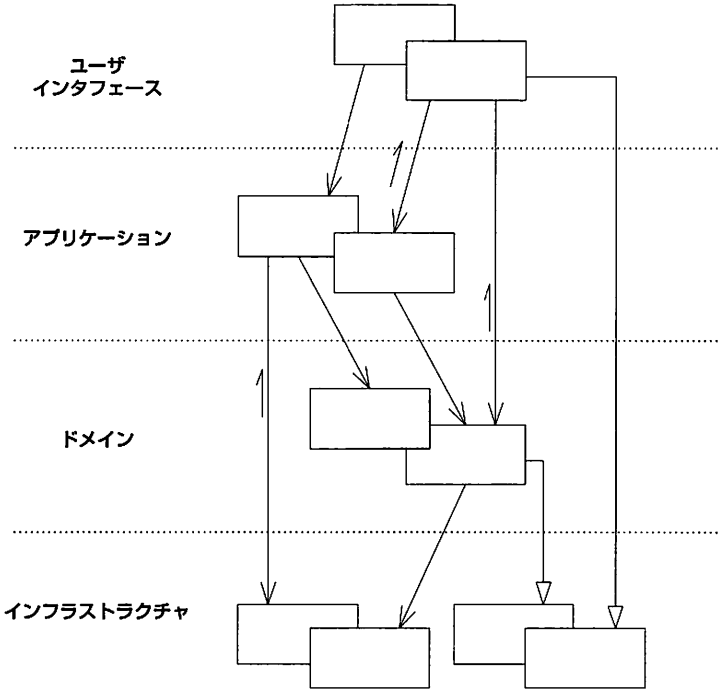
第4章

ドメインを隔離する

ソフトウェアの中でドメインから生じる問題を解決するのに特化した部分は、通常、ソフトウェアシステム全体の中ではごく一部を構成するにすぎない。ただし、小さいから重要ではないということではない。思考を最適に働かせるには、モデル要素を見た時に1つの体系として理解できる必要がある。より多くのオブジェクトが入り混じった中から、モデル要素を拾い上げるように強制されることはあってはならない。これは夜空に星座を見つけ出そうとするようなものだ。ドメインオブジェクトはシステムの他の機能から切り離す必要がある。そうすることで、ドメインの概念を、ソフトウェアの技術にしか関係しない概念と混同したり、システム全体の中に紛れ込んでドメインを見失ってしまったりすることを避けられるようになる。

この分離を行うために、洗練された技術が登場した。よく用いられるものではあるが、ドメインモデリングの原則を適用する上では決定的に重要なので、ドメイン駆動の視点から簡単に見直しておくべきである…。

レイヤ化アーキテクチャ（LAYERED ARCHITECTURE）



輸送アプリケーションで、貨物の荷出し地を都市の一覧から選択するという、ユーザの簡単な操作をサポートしたい。そのためには、次の処理を実行するプログラムコードがなければならない。①スクリーンにウィジェットを描き、②選択できる都市をすべてデータベースに問い合わせて、③ユーザの入力を解釈して検証し、④選択された都市を貨物と結びつけ、⑤データベースへの変更をコミットする。このコードすべてが同じプログラムの一部になるが、輸送業というビジネスに関係するのは、そのごく一部にすぎない。

ソフトウェアプログラムには、さまざまな種類のタスクを実行する設計とコードがある。ユーザ入力を受けつけ、ビジネスロジックを実行し、データベースにアクセスして、ネットワークを通じて通信し、ユーザに情報を表示する、といった具合である。したがって、各プログラム機能に含まれているコードは、かなりの量になる可能性がある。

オブジェクト指向プログラムでは、ユーザインタフェース（UI）、データベースおよびその他の補助的なコードが、ビジネスオブジェクトに直接書かれることはしばしばある。また、ビジネスロジックが新たに追加される時には、UIウィジェットやデータベーススクリ

プトのふるまいに組み込まれてしまう。こういうことが起きるのは、短期的に見ると、動くようにするには最も簡単な方法だからだ。

ドメイン関連のコードがそうした膨大な他のコードの中に拡散してしまうと、コードを見て意味を理解するのがきわめて困難になる。ユーザインタフェースに対する表面的な変更が、実はビジネスロジックを変更することもある。ビジネスルールを変更するために、ユーザインタフェースコードやデータベースコード、その他のプログラム要素を丹念に追跡しなければならないかもしれない。これでは、凝集度の高いモデル駆動のオブジェクトを実装することが現実的ではなくなる。また、自動化されたテストはぎこちないものになってしまう。すべての技術とロジックがそれぞれの活動に関係しているので、プログラムを非常にシンプルに保たなければ、理解できなくなってしまう。

非常に複雑な作業を処理できるプログラムを作成するには、関心事を分離し、分離されたさまざまな設計の要素に集中できるようにしなければならない。同時に、システム内での入り組んだ相互作用は、分離していても引き続き行われなければならない。

ソフトウェアシステムを分割する方法にはあらゆる種類のものがあるが、経験と慣習を通じて業界が収束してきている共通見解はレイヤ化アーキテクチャであり、それも具体的には数個のかなり標準的なレイヤである。レイヤ化というメタファはあまりにも広範に使用されているから、ほとんどの開発者には直観的に感じられる。レイヤ化についての優れた議論の多くは文献として入手可能で、パターンのかたちになっているものもある（Buschmann 他 1996, p.31-51 など）。本質的な原則は、レイヤ内のどの要素も、同じレイヤの他の要素か、その「下にある」レイヤの要素にしか依存しない、ということである。上方へのコミュニケーションは何らかの間接的な仕組みを介さなければならないが、それについては後ほど説明する。

レイヤの価値は、それぞれがコンピュータプログラムにおける特定の側面を専門的に扱うことにある。この専門化によって、各側面の設計はより凝集度が高まり、はるかに解釈しやすくなる。もちろん、決定的に重要なのは、設計における最も重要で凝集度の高い側面を隔離するレイヤを選び出すことだ。ここでも、経験と慣習による何らかの収束が見られる。多くのバリエーションはあるが、うまくいっているアーキテクチャのほとんどは、以下に示す4つの概念的レイヤを基にした、何らかのバージョンを使用している。

ユーザインタフェース（またはプレゼンテーション層）	ユーザに情報を表示して、ユーザのコマンドを解釈する責務を負う。外部アクタは人間のユーザではなく、別のコンピュータシステムのこともある。
アプリケーション層	ソフトウェアが行うことになっている仕事を定義し、表現力豊かなドメインオブジェクトが問題を解決するように導く。このレイヤが責務を負う作業は、ビジネスにとって意味があるものか、あるいは他システムのアプリケーション層と相互作用するのに必要なものである。 このレイヤは薄く保たれる。ビジネスルールや知識を含まず、やるべき作業を調整するだけで、実際の処理は、ドメインオブジェクトによって直下のレイヤで実行される共同作業に委譲する。ビジネスの状況を反映する状態は持たないが、ユーザやプログラムが行う作業の進捗を反映する状態を持つことはできる。
ドメイン層（またはモデル層）	ビジネスの概念と、ビジネスが置かれた状況に関する情報、およびビジネスルールを表す責務を負う。ビジネスの状況を反映する状態はここで制御され使用されるが、それを格納するという技術的な詳細は、インフラストラクチャに委譲される。**この層がビジネスソフトウェアの核心である。**
インフラストラクチャ層	上位のレイヤを支える一般的な技術的機能を提供する。これには、アプリケーションのためのメッセージ送信、ドメインのための永続化、ユーザインタフェースのためのウィジェット描画などがある。インフラストラクチャ層は、ここで示す4層間における相互作用のパターンも、アーキテクチャフレームワークを通じてサポートすることがある。

プロジェクトによっては、ユーザインタフェース層とアプリケーション層を厳密に区別しないこともある。また、複数のインフラストラクチャ層を持つこともある。しかし、ドメイン層を分離してはじめて、モデル駆動開発が可能になるのだ。

それゆえ：

複雑なプログラムはレイヤに分割すること。各レイヤで設計を進め、凝集度を高めて下位層だけに依存するようにすること。標準的なアーキテクチャパターンに従って、上位のレイヤに対しては疎結合にすること。ドメインモデルに関係するコード全部を1つの層に集中させ、ユーザインタフェース、アプリケーション、インフラストラクチャのコードから分離すること。表示や格納、アプリケーションタスク管理などの責務から解放されることで、ドメインオブジェクトはドメインモデルを表現するという責務に専念できる。これによって、モデルは十分豊かで明確になるように進化し、本質的なビジネスの知識をとらえて、それを機能させることができるようになる。

ドメイン層をインフラストラクチャ層とユーザインタフェース層から分離することにより、各レイヤをはるかに明確に設計できるようになる。レイヤを隔離すると、保守にかかるコストが格段に低くなるが、それは、各レイヤが別々の速度で進化して、別々の要求に

対応する傾向があるからである。このように分離することで、分散システムにも配置しやすくなる。別々のレイヤを、それぞれ別々のサーバやクライアントに柔軟に配置できるようにすることで、通信のオーバーヘッドを最小化して、性能を向上させることができるのだ（Fowler 1996）。

例 4.1 ——オンラインバンキングの機能をレイヤに分割する

あるアプリケーションが、銀行預金口座を維持管理するためのさまざまな機能を提供している。その1つが資金振替の機能であり、そこでユーザは、2つの口座番号と金額を入力するか選択した後に、送金を開始する。

この例を扱いやすくするために、セキュリティに代表される主要な技術的機能を省略している。また、ドメインの設計も過度に単純化している（現実に即して複雑にしても、レイヤ化アーキテクチャの必要性が強まるだけだっただろう）。さらに、ここで暗示される特定のインフラストラクチャは、例を明確にするために単純でわかりやすいものにしている。したがって、こう設計することを提案しているわけではない。残る機能の責務は、図4.1に示すようにレイヤ化される。

注意して欲しいのは、基本的なビジネスルールに関する責務を負っているのはドメイン層であって、アプリケーション層ではないということだ。この場合、そのルールとは「すべての振り込みには、それに対応する引き落としがある」である。

このアプリケーションは、振替要求の出所について何の前提も設けない。このプログラムに含まれるユーザインタフェースには、おそらく口座番号と金額の入力フィールドと、指示用のボタンがある。しかし、そのユーザインタフェースは、アプリケーション層やその他のどんな下位のレイヤにも影響を及ぼさずに、XMLで書かれた出金依頼に置き換えることもできるだろう。ただ、こうした分離が重要なのは、プロジェクトで頻繁にユーザインタフェースを出金依頼に置き換える必要があるからではなく、関心事を明確に分離することで、各レイヤの設計を理解し維持管理するのが容易になるからである。

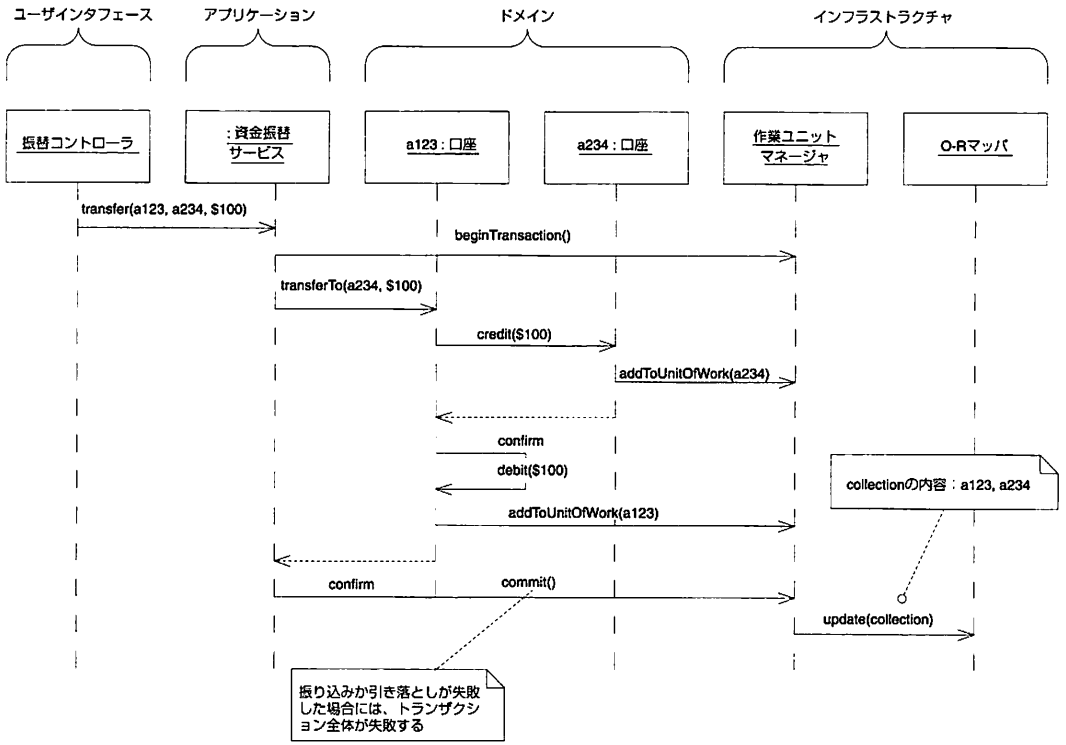


図4.1 オブジェクトは自分のレイヤと一致した責務を実行し、同じレイヤにある他のオブジェクトと強く結びつく

実際、図4.1はそれ自体が、ドメインを分離しないことによって生じる問題をある程度示してくれている。リクエストからトランザクション制御まで、あらゆるものを含めなければならなかったため、全体の相互作用を追いかけられるくらい単純にしておくためには、ドメイン層を簡単なものに書き直さなければならなかった。分離されたドメイン層の設計に集中していたのであれば、ページにも、頭の中にも、ドメインのルールをよりうまく表現するモデルのための余地ができていたはずである。おそらくそこには、元帳オブジェクトや、振り込みおよび引き落としオブジェクト、金銭取引オブジェクトなどが含まれていただろう。

レイヤを関係づける

ここまでの議論で焦点を合わせてきたのは、レイヤの分離と、そのように区切ることでプログラムの各側面の設計、特にドメイン層の設計が改善される様子だった。しかし、当然、レイヤ同士はつながっていなければならない。分離によってもたらされる恩恵を失わ

ずに、レイヤ同士をつなげたいという要求が、数多くのパターンを生む動機になっている。

レイヤ同士は疎結合であるべきで、設計の依存関係は1方向にだけ向けられる。上位のレイヤは、下位のレイヤにある要素を直接使用したり操作したりできる。これは、公開インタフェースを呼び出したり、要素の参照を（少なくとも一時的に）保持したりするか、一般的には相互作用のための従来の手段を使用することによって行われる。しかし、下位のレイヤにあるオブジェクトが、（直接的な呼び出しへ応答することを超えて）上方と通信をしなければならない場合には、別の仕組みが必要となる。そのために活用されるのが、コールバックやオブザーバ（OBSERVER）（Gamma他 1995）といった、レイヤ同士を関係づけるためのアーキテクチャパターンである。

ユーザインタフェースをアプリケーション層とドメイン層に結びつけるパターンの父祖に当たるのは、モデル・ビュー・コントローラ（MVC：MODEL-VIEW-CONTROLLER）である。これは遡ること1970年代にSmalltalkの世界で開拓され、後続の多くのUIアーキテクチャにインスピレーションを与えてきた。Fowler（2002）は、このパターンと、こうしたテーマに関する便利なバリエーションを論じている。また、Larman（1998）は、こうした関心についてモデル／ビュー分離パターン（MODEL-VIEW SEPARATION PATTERN）において調査していて、その中にあるアプリケーションコーディネータ（APPLICATION COORDINATOR）は、アプリケーション層を結びつけるアプローチの1つである。

ユーザインタフェースとアプリケーションを結びつける方法は他にもある。ただ、ここでの目的を達成する上では、ドメイン層を隔離した状態に保つことさえできれば、どのアプローチでも構わない。ドメインオブジェクトを設計するにあたって、それと同時に、ドメインオブジェクトと相互作用する可能性のあるユーザインタフェースのことを考慮しないでいられればよいのだ。

インフラストラクチャ層は、通常、ドメイン層におけるアクションを開始させることはない。ドメイン層より「下位」にあるので、サービスを提供する対象のドメインについて特定の知識を持っていてはならないからだ。事実、そういう技術的な機能は、ほとんどの場合サービスとして提供される。例えば、アプリケーションが電子メールを送信しなければならない場合、何らかのメッセージ送信インタフェースがインフラストラクチャ層に配置されることで、アプリケーション層の要素がメッセージの伝送を要求できる。こうして分離することで、機能の用途はさらに広がる。このメッセージ送信インタフェースは、電子メール送信プログラムやファックス送信プログラムなど、使用できるものなら何にでも接続できるかもしれない。しかし、主要な利点は、アプリケーション層をシンプルにして、本来の仕事だけに集中した状態に保つことにある。すなわち、メッセージを**いつ**送信するかは知っているが、それを**どのように**行うかまでは背負い込まないようにするのだ。

アプリケーション層とドメイン層は、インフラストラクチャ層が提供するサービスを呼

び出す。サービスのスコープが適切に選択され、インタフェースがうまく設計されていれば、呼び出し側は疎結合を保つことができ、サービスのインタフェースによってカプセル化されている手の込んだふるまいによって混乱させられることもない。

しかし、すべてのインフラストラクチャが、上位レイヤから呼び出せるサービスのかたちになっているとは限らない。技術的コンポーネントによっては、他のレイヤの基本的な機能を直接支援するように設計されたものもあれば（すべてのドメインオブジェクトに抽象基底クラスを提供するなど）、他のレイヤを関係づけるための仕組み（MVCの実装など）を提供するものもある。そういう「アーキテクチャフレームワーク」は、プログラムにある他の部分の設計に、はるかに大きな影響を与える。

アーキテクチャフレームワーク

インフラストラクチャが、インタフェース経由で呼び出されるサービスのかたちで提供される場合、レイヤ化の作用とレイヤ間を疎結合に保つ方法は、かなり直観的に理解できる。しかし、何らかの技術的問題によって、より立ち入った形式のインフラストラクチャが必要となることもある。多くのインフラストラクチャの要求を統合するフレームワークを使うと、他のレイヤをきわめて特殊な仕方で実装しなければならなくなることも多い。これは例えば、フレームワークが定義したクラスのサブクラスとして実装したり、またはメソッドシグネチャを規定したりすることが挙げられる†（サブクラスが親クラスより上位のレイヤにあるのは直観に反しているように見えるかもしれない。しかし、他方の知識をより反映しているのはどちらのクラスかを忘れないで欲しい）。最もよくできたアーキテクチャフレームワークは、複雑な技術的問題を解決する一方で、ドメイン開発者がモデルを表現することに集中できるようにする。しかし、フレームワークは、ドメインについての設計の選択肢を制限する前提を多く設けすぎたり、開発の速度を低下させるほど実装を重苦しくしてしまったりすることで、容易に開発の妨げとなり得るのだ。

通常は、何らかのかたちのアーキテクチャフレームワークが必要である（あまり役に立たないフレームワークを、チームが選ぶこともあるが）。フレームワークを適用する場合、チームはその目標に集中しなければならない。それはすなわち、ドメインモデルを表現し、重要な問題を解決するためにそのモデルを使用するような実装を構築するということである。チームは、この目標を達成するためにフレームワークを採用する道を模索しなければならない。それがフレームワークのすべての機能を使用しないことを意味しても構わない。

---

† 訳注：原書が出版された2003年当時、Javaのバージョンは1.4であったため、ほとんどのフレームワークが処理系コンポーネントを提供する際、基底クラスとするか、リフレクションのためのネーミング規約を定める必要があった。しかし、Java 5.0より導入された註釈（アノテーション）により、現在ではフレームワークにほとんど依存しないで実装できるようになっている。

例えば、初期のJ2EEアプリケーションは、すべてのドメインオブジェクトを「エンティティBean」として実装することが多かった。だが、このアプローチは、性能も開発の速度も落としてしまう。代わりに、最近のベストプラクティスでは、J2EEフレームワークを使うのは粒度の大きいオブジェクトだけにして、ほとんどのビジネスロジックを普通のJavaオブジェクトで実装する。フレームワークの持つ欠点の多くを避けるためには、万能の解決策を探してはならない。難しい問題を解決するために、必要なものだけを選んで適用すればよいのだ。フレームワークの持つ機能の中で、最も価値のあるものだけを賢明に適用することで、実装とフレームワークの結合度を下げ、後の設計上の意思決定を、より柔軟に行えるようになる。さらに重要なことだが、現在の多くのフレームワークを使うのがいかに複雑かを考えれば、このミニマリズムによって、ビジネスオブジェクトは読みやすく、表現力豊かな状態に保たれるのだ。

アーキテクチャフレームワークとその他のツールは進化を続けるだろう。新しいフレームワークによって、アプリケーションの技術的側面は、ますます自動化されたり、組み立てるだけになったりするだろう。こうしたことが正しく行われれば、アプリケーション開発者は、コアビジネスの問題をモデリングすることにますます多くの時間を割けるようになり、生産性と品質は大きく向上するだろう。しかし、この方向に進む場合は、技術的な解決策に熱中しないよう警戒しなければならない。これは例えば、フレームワークが定義したクラスのサブクラスとして実装したり、またはメソッドシグネチャを規定したりすることが挙げられる。

ドメイン層はモデルが息づく場所

レイヤ化アーキテクチャは、今日、さまざまなレイヤ化の構造をとりながら、ほとんどのシステムで使用されている。開発スタイルの多くは、レイヤ化の恩恵を受けられる。だが、ドメイン駆動設計が求めるのは、たった1つの特別なレイヤが存在することだけだ。

ドメインモデルは概念の集合である。「ドメイン層」は、モデルと、モデルに直接関係する設計上のすべての要素が現れる場所である。ビジネスロジックの設計と実装によって、ドメイン層が構成されるのだ。そして、モデル駆動設計においては、モデルの概念を映し出すドメイン層によって、ソフトウェアが構成される。

ドメインロジックがプログラムの他の関心事と混ざり合っていたら、設計と実装をこのように一致させることは現実的ではない。ドメインの実装を隔離することは、ドメイン駆動設計の必要条件なのだ。

利口なUI「アンチパターン」(SMART UI "ANTI-PATTERN")

…ここまでで論じたことが、オブジェクトアプリケーションのための、広く認められているレイヤ化アーキテクチャパターンについての要約である。しかし、ユーザインタフェース、アプリケーション、ドメインをこのように分離することはよく試みられるのだが、ほとんど達成されることはないので、正反対のパターンにもそれ自体で議論する価値がある。

多くのソフトウェアプロジェクトが実際に採用し、今後も採用し続けるであろう、洗練とは程遠い設計アプローチのことを、私は利口なUIと呼んでいる。だが、この利口なUIは相互に排他的な二者択一の分岐点で、ドメイン駆動設計のアプローチとは両立しない。その道を選んだら、本書の内容はほとんど適用できなくなる。私が興味を持っているのは、利口なUIが適用されない状況である。だからこそ、皮肉を込めて「アンチパターン」と呼んでいるのだ。利口なUIについてここで議論することにより、ドメイン駆動設計との違いが示され、本書の残りの部分で選ばれるさらに困難な道を進むことが、どのような状況なら正当化されるかが明確になる。

* * *

あるプロジェクトで単純な機能を実現しようとしている。この機能はデータの入力と表示が大部分を占め、ビジネスルールはほとんどない。

洗練されていないチームが、単純なプロジェクトを担当するにあたって、レイヤ化アーキテクチャを伴うモデル駆動設計に挑戦しようとすると、習得の難しさに苦労することになる。チームメンバは複雑な新しい技術を習得し、オブジェクトモデリングを学ぶ過程を、つまずきながら通らなければならない(それは、本書の助けがあっても骨の折れることだ!)。インフラストラクチャと各レイヤを管理するオーバーヘッドにより、非常に単純な作業にも、より長い時間がかかる。単純なプロジェクトはたいていスケジュールが短く、期待も控えめである。チームが割り当てられた作業を完了させるよりもはるか前、ましてやこのアプローチの持つ刺激的な可能性を実演するよりも前に、プロジェクトはキャンセルされることになるだろう。

チームにもっと時間が与えられたとしても、チームメンバは、専門家の手を借りなければ、この技術を習得できそうもない。最終的に、これらの難題を本当に克服したとしても、作られるのは単純なシステムである。豊富な機能が望まれることはない。

もっと熟練したチームであれば、このようなトレードオフには直面しないはずだ。熟練した開発者であれば、これから学ぶべきことも少なく、レイヤを管理するのに必要な時間を縮められる。ドメイン駆動設計が最も効果があるのは、大掛かりなプロジェクトに対してであり、実現するには強力なスキルも必要になる。だが、すべてのプロジェクトが大掛かりとは限らないし、すべてのプロジェクトチームがそういうスキルを習得できるわけで

もない。

それゆえ、状況が許せば：

すべてのビジネスロジックをユーザインタフェースに入れること。アプリケーションを小さな機能に切り刻み、それを独立したユーザインタフェースとして実装し、ビジネスルールもその中に埋め込むこと。関係データベースを、データの共有リポジトリとして使用すること。ユーザインタフェースの構築と視覚的なプログラミングのためのツールについては、手に入る中で最も自動化が進んでいるものを使用すること。

異義あり！ 教義によれば(本書の他の箇所も含め、いたるところで主張されている通り)、ドメインとユーザインタフェースは分離しなければならない。実際、そうした分離を行わなければ、本書後半で議論する方法はどれも適用することが困難である。したがって、利口なUIは、ドメイン駆動設計の文脈では「アンチパターン」と見なせるのだ。それでも、他の文脈においては正当なパターンとなることもある。実際は、利口なUIにも利点はあり、このパターンが最も有効に機能する状況もある。利口なUIがこれほど広く使われているのには、こうした理由もあるのだ。このような利点についてここで考えることで、なぜアプリケーション処理をドメイン層から分離しなければならないのか、さらに重要なこととして、どういう時に分離したくないのかが理解できるようになる。

利　点

- 単純なアプリケーションの場合、生産性が高く、すぐに作れる。
- それほど有能でない開発者でも、この方法なら、ほとんど訓練しないで仕事ができる。
- 要求分析が不足していても、プロトタイプをユーザに公開し、その要望を満たすように製品を変更することで、問題を克服できる。
- アプリケーションが互いに分離しているので、小さなモジュールの納品スケジュールは比較的正確に計画できる。単純なふるまいをつけ加えるようなシステムの拡張であれば、容易に対応できるだろう。
- 関係データベースはうまく機能し、データレベルでの統合が実現される。
- 4GLツールが実にうまく機能する。
- アプリケーションが引き継がれた場合、保守プログラマは自分が理解できない部分を素早く作り替えられる。変更による影響が、それぞれ特定のユーザインタフェースに限定されるからだ。

欠　点

- アプリケーションの統合は困難で、データベースを経由させるしかない。
- ふるまいが再利用されることも、ビジネスの問題が抽象化されることもない。ビジネスルールは、適用先の操作それぞれで複製されることになる。

- 迅速なプロトタイピングやイテレーションを行おうとしても、自然と限界に行き当たる。抽象化が欠けているために、リファクタリングの選択肢が制限されるからだ。
- 複雑さによってすぐに覆い尽くされてしまうので、成長しようとしても、単純なアプリケーションを追加することしかできない。より豊かなふるまいが実現できるようになるといった、優雅な道は存在しない。

このパターンを意識して適用すれば、チームは他のアプローチによって必要とされる大量のオーバーヘッドを負わずに済む。最後までやり遂げる気のない高度な設計アプローチに手を出してしまうというのは、チームが犯しがちな間違いだ。同じようによくある、高くつく間違いとしては、プロジェクトで必要もない複雑なインフラストラクチャを構築して高性能なツールを使用するというものもある。

こういうアプリケーションにとって、ほとんどの柔軟な言語（Javaなど）は過剰であり、ひどく高くつくことになる。選ぶべきなのは、4GL風のツールである。

忘れないで欲しいのだが、このパターンの結末の1つが、別の設計アプローチに移行するにあたって、アプリケーションをすべて置き換えるほかなくなるというものである。Javaのような汎用言語を使用したからといって、それだけでは、必ずしも後になって利口なUIを断念することになるわけではない。したがって、利口なUIへ向かうなら、それに適合した開発ツールを選択するべきだ。あえて両天秤にかけてはならない。柔軟な言語を使用したからといって、柔軟なシステムができるわけではないが、できあがるシステムはおそらく高価にはなるだろう。

同じ理由から、モデル駆動設計に取り組むチームは、最初からそれにふさわしいやり方で設計する必要がある。もちろん、大志を抱いた熟練のプロジェクトチームでも、シンプルな機能から始め、継続的なイテレーションを通じて、より豊かな機能を構築していかなければならない。しかし、最初の一歩を試しに踏み出す先が、ドメインレイヤを隔離したモデル駆動でなければ、プロジェクトは利口なUIに行き詰まることになるだろう。

＊ ＊ ＊

利口なUIについて議論したのは、レイヤ化アーキテクチャのようなパターンが、ドメイン層を隔離するために必要になる理由と、それがどういう時なのかを明確にするためだけだ。

利口なUIとレイヤ化アーキテクチャとの中間には、他にも解決策がある。例えば、Fowler（2002）が説明しているトランザクションスクリプト（TRANSACTION SCRIPT）は、ユーザインタフェースをアプリケーションから分離はするが、オブジェクトモデルは提供しない。要はこういうことである。**アーキテクチャがドメインに関連するコードを隔離して、凝集度が高いドメインの設計が、システムの他の部分と疎結合できるようにしている**

なら、そのアーキテクチャはおそらくドメイン駆動設計を支えられるだろう。

他の開発スタイルにもそれなりの存在意義があるが、複雑さと柔軟性において、さまざまな限界があることを認めなければならない。ドメインの設計を分離し損ねると、状況によっては実に悲惨なことになりかねない。アプリケーションが複雑で、モデル駆動設計に取り組むつもりなら、歯を食いしばり、必要な専門家をそろえた上で、利口なUIを避けるべきである。

その他の隔離

あいにく、インフラストラクチャとユーザインタフェース以外にも、繊細なドメインモデルを台無しにしかねない影響がある。他のドメインのコンポーネントで、自分のモデルには完全に統合しきれていないものを扱わなければならないこともあるだろうし、同じドメインに関する別のモデルを使用する他の開発チームにも対応する必要があるだろう。これらをはじめとする要因がモデルをぼやけさせ、その有用性を奪ってしまう。第14章「モデルの整合性を維持する」では、この話題を取り上げ、境界づけられたコンテキストや腐敗防止層などのパターンを紹介する。本当に複雑なドメインモデルは、それ自体が扱いにくくなるかもしれない。第15章「蒸留」では、ドメインレイヤ内で差別化をはかり、ドメインについての本質的な概念を、周辺のそれほど重要ではない詳細から解放できるようにする方法について議論する。

ただし、その議論をするのはまだ先だ。次は、効果的なドメインモデルと表現力豊かな実装を共に進化させるための土台を見ていく。結局、ドメインを隔離することの最も優れた点は、それ以外のものをすべて取り払って、ドメインの設計に本当に集中できるようにすることにあるのだ。

第5章

ソフトウェアで表現されたモデル

実装にあたって妥協をしつつ、モデル駆動設計の持つ力を失わないようにするには、基本を見直す必要がある。モデルと実装は詳細のレベルで結びつけなければならない。本章は、モデルを構成する詳細な個々の要素に焦点を合わせ、後続の章での取り組みを支えられるよう具体化する。

この議論は、関連を設計して無駄をなくすという問題から始める。オブジェクト間の関連は、想像するのも描くのも簡単だが、実装するとなると泥沼になるかもしれない。関連によって示されるのは、詳細な実装上の意思決定がモデル駆動設計を実現する上でどれほど重大か、ということだ。

オブジェクト自体に目を向けつつも、詳細なモデル上の選択と実装上の関心事との間にある関係性を絶えず精査することで、モデルを表現する3パターンの要素の区別に専念することにしよう。その要素とは、エンティティ、値オブジェクト、サービスである。

ドメインの概念をとらえるオブジェクトを定義することは、表面上はきわめて直観的なことに見える。しかし、概念の持つ意味の陰に、深刻な難題が潜んでいるのだ。モデル要素の持つ意味を明確にできるような、ある区別が登場してきている。さらに、その区別が設計の原則と結びつくことで、特殊なオブジェクトが切り出されるのだ。

あるオブジェクトは、状態が異なったり、さらに別々の実装をまたいだりしたとしても追跡されるような、連続性と一意性を持ったものを表現しているのか？ それとも、他の何かの状態を記述する属性なのか？ これがエンティティと値オブジェクトとの基本的な区別である。何らかのパターンに明確に従うオブジェクトを定義することで、そのオブジェクトはあいまいではなくなり、強固な設計を行うための具体的な選択に向かう道筋が整えられる。

次に、ドメインの側面によっては、オブジェクトとしてよりも、アクションや操作とし

て表現した方が明確になるものもある。オブジェクト指向モデリングの伝統からはやや外れるが、これらについてはサービスとして表現して、操作を行う責務をエンティティや値オブジェクトに押しつけない方が適切であることも多い。サービスとは、要求に応じてクライアントのために行われる何かである。ソフトウェアの技術的なレイヤには多くのサービスがある。また、サービスはドメインにも登場する。その際にモデル化されるのは、ソフトウェアが実行すべきことに対応し、状態には対応しないような活動だ。

オブジェクトモデルを純粋なかたちで保つことができず、妥協しなければならなくなる状況も必ず訪れる。例えば、関係データベースに格納する場合などだ。本章では、こうした厄介な現実への対処を迫られた際、正しい道に留まるための指針を提示する。

最後に、モジュールについて議論することによって、設計上のあらゆる意思決定は、ドメインについての何らかの洞察によって動機づけられなければならないという点を強調する。高凝集と低結合という考え方は、しばしば技術的な指標と考えられているが、概念そのものにも適用できる。モデル駆動設計においては、モジュールはモデルの一部であり、ドメインにある概念を反映していなければならない。

本章では、これらの構成要素をすべてまとめる。これらの構成要素によって、モデルはソフトウェアというかたちに具体化されるのだ。こういう考え方は従来からあるものだし、その後に続く、モデリングと設計を重視する傾向もこれまで論じてきたものだ。しかし、これらの構成要素をドメイン駆動設計の文脈でとらえることによって、開発者は詳細なコンポーネントを作成しやすくなる。こうしたコンポーネントは、さらに大きなモデルや設計の問題に取り組む際に、ドメイン駆動設計における重要事項に対して寄与してくれるのだ。また、避けられない妥協を行う時でも、基本原則に対する感覚に助けられることによって、開発者は道を踏み外さずにいられる。

関 連

モデリングと実装との間の相互作用は、オブジェクト間の関連を見た時に、特に油断ならないものとなる。

モデルにおいて辿ることのできる関連それぞれに対応して、ソフトウェアにおいても同じ特性を持つ仕組みが存在する。

顧客とセールスマンの関連を示すモデルは、2つの事柄に対応している。第一に、これは現実に存在する2人の人物の間にある関係で、開発者がドメインに関連していると見なしたものを抽象化している。その一方で、2つのJavaオブジェクトの間のオブジェクトポインタ、データベース検索のカプセル化、もしくはそれと同じような実装とも対応している。

例えば、一対多の関連は、インスタンス変数のコレクションとして実装されるかもしれ

ない。しかし、そこまで直接的に設計する必要は、必ずしもない。コレクションは存在せず、アクセッサメソッドがデータベースに問い合わせて適切なレコードを見つけ、それに基づいてオブジェクトをインスタンス化するかもしれない。これらの設計はどちらも同じモデルを反映しているだろう。関連を辿る特定の仕組みを設計で定義し、モデル内にある関連とふるまいを一致させなければならない。

現実の世界では、多対多の関連がたくさんあり、その多くはもともと双方向である。同じことは、ブレインストーミングを行ってドメインを調べている時に現れるような、モデルの初期のかたちにも、たいてい当てはまる。だが、これらの一般的な関連のせいで、実装と保守が複雑になってしまう。しかも、こうした一般的な関連は、そこにある関係の性質について、ほとんど何も伝えないのだ。

関連をもっと扱いやすくするには、少なくとも3つの方法がある。

1. 関連を辿る方向を強制する。
2. 限定子を付加して、多重度を効果的に減らす。
3. 本質的ではない関連を除去する。

関係性をできる限り制限することが重要だ。双方向の関連があるということは、両方のオブジェクトが一緒になった時にしか理解できないということを意味している。アプリケーションの要件が双方向に辿ることを求めないなら、一方向にだけ辿るようにすることで、相互依存関係が減り、設計がシンプルになる。ドメインを理解することで、本来はどちらの方向性が重要であるのかが、明らかになる場合がある。

他の多くの国々と同じく、米国にも多くの大統領がいる。これは双方向の一対多関係である。だが、「ジョージ・ワシントン」の名前を出してから、「彼はどの国の大統領だったか？」と聞くことはめったにない。現実に即せば、関連を一方向に減らし、国から大統領の方向へのみ辿るようにできる。このように改良することで、設計がより実用的になると同時に、ドメインに対する洞察が実際に反映される。こうした方向性への配慮によってとらえられるのは、関連のある方向が、もう一方よりはるかに有意義で重要だという理解である。これによって「人」クラスは、「大統領」というはるかに基本的ではない概念から、独立した状態に保たれる。

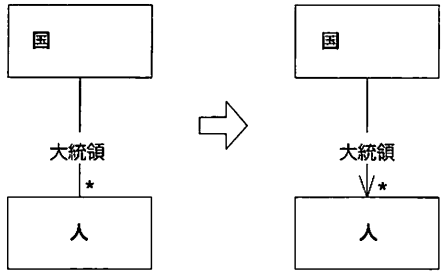


図5.1 関連を辿る方向は、ドメインにおいて自然に重要視されるものを反映していることがある

より深く理解することによって、「限定された」関係に至ることは非常に多い。大統領についてさらに深く調べてみると、1つの国にはある時点に1人しか大統領がいない（たぶん、内戦中を除いて）ことに気づく。この限定子は、多重度を一対一に減らし、重要なルールを明示的にモデルへ組み込む。1790年の米国大統領は誰か？ ジョージ・ワシントンである。

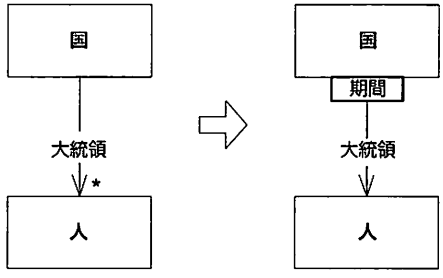


図5.2 限定された関連はより多くの知識を伝える、より実用的な設計である

多対多の関連を辿る方向を制限することで、実装は一対多へと効果的に減らされる。これは**はるかに**容易な設計である。

ドメインにおける重要度を反映させるように、関連を一貫して限定することによって、その限定された関連は、伝達力がより豊かになり、実装もシンプルになる。だが、それだけでなく、残された双方向の関連にも意味が与えられる。関係が双方向であることがドメインにおけるセマンティックな特徴であり、アプリケーションの機能にとって必要とされているならば、両方向へ辿れるようにしておくことで、そのことを伝えられる。

もちろん、究極の単純化は、関連をすべて消すことだ。ただし、消してよいのは目下の仕事やモデルオブジェクトの本質的意味にとって必須でないものだけだ。

例 5.1 ——証券取引口座における関連

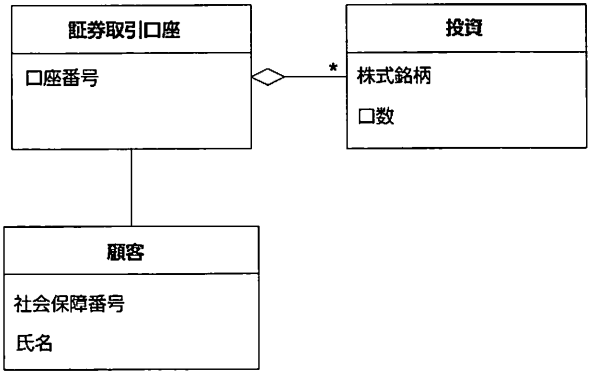


図5.3

このモデルにおける**証券取引口座**（**Brokerage Account**）のJavaによる実装の1つは、次のようになるだろう。

```
/**
 * 証券取引口座.
 */
public class BrokerageAccount {
  // 口座番号
  String accountNumber;
  // 顧客
  Customer customer;
  // 投資
  Set investments;

  // コンストラクタ他は省略

  public Customer getCustomer() {
    return customer;
  }
  public Set getInvestments() {
    return investments;
  }
}
```

しかし、関係データベースからデータを取り出す必要がある場合、同じくモデルと矛盾しない別の実装として、次のようなものもある。

テーブル：BROKERAGE_ACCOUNT

ACCOUNT_NUMBER	CUSTOMER_SS_NUMBER

テーブル：CUSTOMER

SS_NUMBER	NAME

テーブル：INVESTMENT

ACCOUNT_NUMBER	STOCK_SYMBOL	AMOUNT

```
/**
 * 証券取引口座.
 */
public class BrokerageAccount {
  // 口座番号
  String accountNumber;
  // 顧客社会保障番号
  String customerSocialSecurityNumber;

  // コンストラクタ他は省略する

  public Customer getCustomer() {
    String sqlQuery =
      "SELECT * FROM CUSTOMER WHERE" +
      "SS_NUMBER='" + customerSocialSecurityNumber + "'";
    return QueryService.findSingleCustomerFor(sqlQuery);
  }
  public Set getInvestments() {
    String sqlQuery =
      "SELECT * FROM INVESTMENT WHERE" +
      "ACCOUNT_NUMBER='" + accountNumber + "'";
    return QueryService.findInvestmentsFor(sqlQuery);
  }
}
```

（注：ここに示しているQueryServiceは、データベースから行を取り出してオブジェクトを作成するユーティリティである。これは例を説明するために単純化しており、実際のプロジェクトにとっては、必ずしもよい設計とは限らない。）

証券取引口座と投資（Investment）との間にある関連を限定し、多重度を減らすことでモデルを改良しよう。ここでは、1つの株式（stock）に対して1つの投資しかあり得ないとしている。

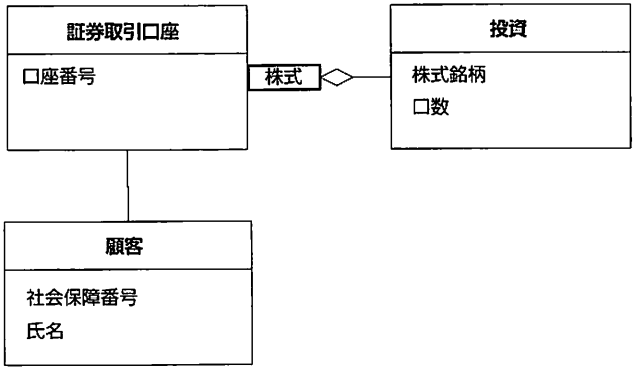


図5.4

これがすべてのビジネスの状況に当てはまるとは限らない（例えば、ロットの追跡が必要な場合など）。しかし、具体的なルールが何であれ、関連に対する制約が見つかったら、それをモデルと実装に含めるべきである。そうすることで、モデルはより正確になり、実装は保守しやすくする。

Javaの実装は、次のようになるだろう。

```
public class BrokerageAccount {
  String accountNumber;
  Customer customer;
  Map investments;

  // コンストラクタ他は省略

  public Customer getCustomer() {
    return customer;
  }

  /**
   * 投資を取得する.
   * @param stockSymbol 株式銘柄
   * @return 投資
   */
```

●例5.1 ——証券取引口座における関連

```
  public Investment getInvestment(String stockSymbol) {
    return (Investment) investments.get(stockSymbol);
  }
}
```

そして、SQLを用いた実装は以下のようになる。

```
public class BrokerageAccount {
  String accountNumber;
  String customerSocialSecurityNumber;

  // コンストラクタ他は省略

  public Customer getCustomer() {
    String sqlQuery = "SELECT * FROM CUSTOMER WHERE SS_NUMBER='"
      + customerSocialSecurityNumber + "'";
    return QueryService.findSingleCustomerFor(sqlQuery);
  }
  public Investment getInvestment(String stockSymbol) {
    String sqlQuery = "SELECT * FROM INVESTMENT "
      + "WHERE ACCOUNT_NUMBER='" + accountNumber + "'"
      + "AND STOCK_SYMBOL='" + stockSymbol + "'";
    return QueryService.findInvestmentFor(sqlQuery);

  }
}
```

モデルの関連を慎重に蒸留し制限することは、モデル駆動設計を行う上で大いに役立つ。それではここで、オブジェクトそのものに目を向けよう。ある区別をすることによって、モデルが明確になり、より現実に即して実装できるようになる…。

エンティティ（ENTITIES）（別名　参照オブジェクト（REFERENCE OBJECTS））



多くのオブジェクトは、本質的に、その属性によってではなく、連続性と同一性（identity）によって定義される。

＊　＊　＊

ある家主の女性が、私が建物に大きな損害を与えたと主張して、私を訴えた。送られてきた書類によると、アパートの部屋の壁には穴が開き、カーペットにはしみがあって、流し台の毒々しい液体からは腐食性のガスが発生し、台所の壁紙がはがれてしまったという。裁判文書は、損害の責任を負う賃借人として私を指定し、名前と当時の現住所で私のことを識別していた。だが、これには面食らった。そんな荒れ果てた場所に、私は行ったことすらなかったからである。

しばらくして、これは人違いに違いないと思った。私は原告に電話してそう話したが、彼女は信じてくれなかった。前の賃借人が、何か月もの間、彼女を巧みに避けていたのだ。彼女にこれほどの金額を負担させた人物は私ではないと、どうすれば証明できるだろう？電話帳に載っているEric Evansは私しかいなかったのだ。

だが結局は、その電話帳が私を救済してくれることになったのである。私は、2年間同じアパートに住んでいたから、家主に前年の電話帳がまだあるかを尋ねた。彼女は電話帳を探し出し、私の名前が（私と同名の人の記載のすぐ隣で）同じように記載されていることを確かめると、訴えたい相手が私ではないとわかって謝罪し、訴訟を取り下げることを約

束してくれた。

コンピュータはそこまで臨機応変ではない。ソフトウェアシステムで同一性を取り違えたら、データの破損やプログラムエラーにつながる。

ここには特殊な技術的課題があり、それについては後述するが、まずは、本質的な問題から見ていこう。多くの物事が定義されるのは同一性によってであり、属性によってではない。典型的な考え方をすると、人には（引き続き技術的ではない例だが）誕生から死亡まで、さらにその後にも及ぶ同一性がある。その人の身体的な属性は変化するし、ついには消える。名前も変わるかもしれない。金銭上の関係は現れては消える。人に関して変化し得ない属性は1つもないが、それでも同一性は存続する。私は5歳の時と同じ人間だろうか？ この種の形而上学的問いは、効果的なドメインモデルを探し求める上で重要だ。少々言い換えてみよう。アプリケーションのユーザは、私が5歳の時と同一人物かどうかを**気にする**だろうか？

未収金を追跡するソフトウェアシステムでは、その地味な「顧客」オブジェクトにも、より変化に富んだ一面があるかもしれない。迅速な支払いによって地位を積み上げたり、支払い不履行で集金代行会社に引き継がれたりする。販売部が顧客データを抽出して連絡先管理システムに流し込めば、顧客オブジェクトは、そちらのシステムと合わせて二重生活を送るかもしれない。いずれにしても、データベーステーブルに格納するために、オブジェクトは、あっさりと押しつぶされて平らにされる。その情報源が新規のビジネスを生まなくなったら、顧客オブジェクトは引退してアーカイブになるだろう。これは、昔の面影だ。

これらの顧客のかたちはそれぞれ、別々のプログラミング言語と技術に基づいた、別々の実装である。しかし、注文の電話がかかってきた時に、次のことを知ることは重要だ。これは延滞金がある顧客か？ ジャック（特定のセールスマン）が何週間も働きかけている顧客か？ 全く新規の顧客か？

概念上の同一性は、オブジェクトの複数の実装や、その格納形式、さらに電話をかけてきた人といった実世界の登場人物との間で一致しなければならない。だが、属性は一致しないかもしれない。セールスマンが連絡先ソフトウェアに住所の更新を入力し、それがちょうど未収金ソフトウェアに伝わったのかもしれないのだ。あるいは、2つの別々の顧客連絡先は同名かもしれない。分散ソフトウェアの場合、複数ユーザが別々の情報源からデータを入力する可能性がある。そのため、更新トランザクションは複数のシステムに伝えなければならず、したがって、別々のデータベース間で非同期に調整しなければならなくなる。

オブジェクトモデリングを行うと、我々はオブジェクトの属性に集中しがちだが、エンティティの根本的な概念は抽象的な連続性である。この連続性はエンティティのライフサ

イクルを通じて続き、エンティティが多様なかたちをとっても変わることがない。

オブジェクトの中には、主要な定義が属性によってなされないものもある。そういうオブジェクトは同一性のつながりを表現するのであり、その同一性は、時間が経っても、異なるかたちで表現されても変わらない。そういうオブジェクトは属性が異なっていても、他のオブジェクトと一致しなければならないことがある。また、あるオブジェクトは、同じ属性を持っていたとしても、他のオブジェクトと区別しなければならない。同一性を取り違えるとデータの破損につながりかねない。

主として同一性によって定義されるオブジェクトはエンティティと呼ばれる[1]。エンティティはモデリングと設計に際して、特別な考慮が必要だ。ライフサイクルにおいて、エンティティの形態と内容は根本的に変わることがあるが、連続性のつながりは維持されなければならない。同一性は、効果的に追跡できるように定義しなければならない。また、エンティティのクラス定義、責務、属性、および関連は、エンティティの持つ特定の属性ではなく、それが何者であるかを中心にすえるべきだ。それほど本質的には変化しないエンティティや、そこまでライフサイクルが入り組んでいないエンティティであっても、セマンティック上はこのカテゴリの中に位置づけることによって、モデルはよりわかりやすく、実装はより強固になる。

言うまでもなく、ソフトウェアシステムにおけるほとんどの「エンティティ」は、人間ではないし、通常は実体という意味で使われるエンティティでもない。ライフサイクルを通じた連続性を持ち、アプリケーションのユーザにとって重要な区別が属性から独立してなされるものは、すべてエンティティなのだ。考えられるものとしては、人や都市、自動車、宝くじ、銀行取引などがある。

他方、モデル中のすべてのオブジェクトが、意味のある同一性を持ったエンティティであるとは限らない。この問題を複雑にしているのは、オブジェクト指向言語が、あらゆるオブジェクトに「同一性」の演算を組み込んでいるという事実である（Javaの「==」演算子など）。こうした演算は、2つの参照が同一のオブジェクトを指しているかどうかを判別するもので、これはメモリ中の位置の比較やその他の仕組みによって実現される。その意味で、すべてのオブジェクトインスタンスは同一性を持っている。例えば、Javaの実行環境を作成するドメインや、リモートオブジェクトをローカルでキャッシュする技術的なフレームワークのドメインであれば、すべてのオブジェクトインスタンスがまさにエンティティになるだろう。しかし、この同一性の仕組みは他のアプリケーションドメインではほとんど意味をなさない。同一性は、エンティティの持つ巧妙で意味のある属性であり、プ

1　モデルのエンティティはJavaの「エンティティBean」と同じものではない。エンティティBeanは、多かれ少なかれエンティティを実装するためのフレームワークとなることを意図したものだが、そのようには機能しなかった。ほとんどのエンティティは普通のオブジェクトとして実装される。どう実装されるかに関係なく、エンティティはドメインモデルにおいて、本質的に区別されるのだ。

ログラミング言語の持つ自動化された機能には引き継げないのである。

銀行業アプリケーションにおける取引を考えよう。同日に同一口座に対する同額の預け入れが2つあっても、これは別々の取引である。したがって、取引は同一性を持つエンティティである。一方、その2つの取引における金額を表す属性は、おそらく金銭オブジェクトか何かのインスタンスである。これらの値は同一性を持たない。両者を区別しても何の役にも立たないからだ。実は、2つのオブジェクトを同一であると見なす上では、同じ属性を持たなくてもよく、必ずしも同一クラスですらなくてもよいのだ。銀行の顧客が明細書の取引を小切手台帳の取引と照合する際に、具体的に行わなければならないのは、同一の取引同士を突き合わせることである。これは、別々の日付で別々の人に記録されていてもよい（銀行の手形交換日付は小切手の日付より後になる）。小切手番号の目的は、このための一意の識別子として機能することにある。コンピュータプログラムと人手のどちらで処理されても変わりはない。預け入れと現金引き出しは識別番号を持たないので、より注意が必要だが、同じ原則が適用できる。すなわち、各取引はエンティティであり、少なくとも2つの形式で現れるのだ。

同一性が特定のソフトウェアシステムの外でも重要になることはよくある。銀行取引やアパート賃借人の場合がそうだ。しかし、同一性が重要なのはシステムのコンテキスト内だけ、ということもある。例えば、コンピュータプロセスの同一性がそうだ。

それゆえ：

あるオブジェクトが属性ではなく同一性によって識別されるのであれば、モデルでこのオブジェクトを定義する際には、その同一性を第一とすること。クラスの定義をシンプルに保ち、ライフサイクルの連続性と同一性に集中すること。形式や履歴に関係なく、各オブジェクトを識別する手段を定義すること。オブジェクト同士を突き合わせる際に、属性を用いるよう求めてくる要件には注意すること。各オブジェクトに対して結果が一意となることが保証される操作を定義すること。これは、一意であることが保証された記号を添えることで、おそらく実現できる。この識別手段は外部に由来する場合もあれば、システムによってシステムのために作成される任意の識別子の場合もあるが、モデルにおける同一性の区別とは一致しなければならない。モデルは、同じものであるということが何を意味するかを定義しなければならない。

同一性は、世の中にある物に本来備わっているわけではない。これはある「意味」であり、役に立つので、その物に重ね合わされているのだ。現実には同じ物であっても、あるドメインモデルにおいては、エンティティとして表現されるかもしれないし、そうでないかもしれない。

スタジアムの座席予約アプリケーションであれば、座席と参加者をエンティティとして扱うだろう。指定席の場合、各チケットに座席番号が振ってあるだろうから、その座席は

エンティティである。ここでの識別子は座席番号で、これはスタジアム内で一意である。座席には他にも多くの属性があるかもしれない。場所、眺めが遮られるかどうか、それに価格などだ。しかし、座席番号、つまり一意な列と位置だけが、座席を識別して他と区別するのに使用される。

一方、イベントが「自由席」で、チケットを持っていれば、空いている席のどこにでも座れる場合、個々の座席を区別する必要はない。総座席数のみが重要である。この場合にも、座席番号は物理的な座席に刻まれているが、ソフトウェアがそれを追跡する必要はない。実際、モデルが特定の座席番号をチケットと関連づけていたら、それは誤りである。自由席のイベントにそのような制約はないからだ。このような場合、座席は**エンティティではなく**、識別子も必要とされない。

* * *

エンティティをモデル化する

オブジェクトをモデル化する時に、属性について考えるのは自然なことであり、そのふるまいについて考えることもきわめて重要だ。しかし、エンティティにとって最も基本的な責務は、ふるまいが明確で予測可能になるよう、連続性を確立することである。これが一番うまくいくのは、余計なものがない状態が保たれている時だ。属性やふるまいに集中するよりは、エンティティオブジェクトの定義を、最も本質的な特徴にまで削ぎ落とすこと。特に、エンティティを識別するものや、それを検索し突き合わせるのに通常使用されるものが、そういう本質的な特徴だ。つけ加えるのはその概念にとって本質的なふるまいと、そのふるまいが必要とする属性だけにすること。その上で、ふるまいと属性を、コアとなるエンティティに関連した他のオブジェクトへ移動できないか調べること。別のエンティティになるものもあるだろうし、値オブジェクトという本章で次に取り上げるパターンになるものもあるだろう。同一性の問題を克服すると、エンティティは、所有するオブジェクトの操作を調整することによって責務を果たすようになる。

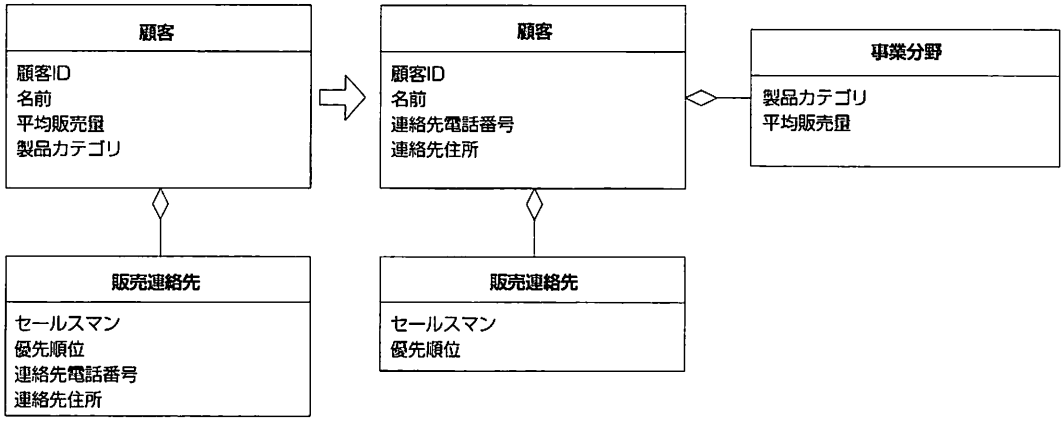


図5.5 同一性と関連した属性は、その**エンティティ**と一緒に留まる

図5.5において、顧客IDは**顧客**エンティティの唯一の識別子だが、電話番号と住所も、**顧客**の検索や突き合わせにしばしば使用される。名前はその人の同一性を**定義**しないが、それを決める手段の一部として使用されることが多い。この例では、電話と住所の属性が**顧客**へ移動したが、実際のプロジェクトにおいてどうするかは、ドメインにおいて顧客が、典型的にどう突き合わされたり識別されたりするか次第である。例えば、**顧客**が別々の目的のために多くの連絡先電話番号を持っていれば、電話番号は同一性と関係しないから、**販売連絡先**の中に残すべきだ。

同一性のための操作を設計する

各エンティティには、同一性を確立して別のオブジェクトと比較するための、操作手段がなければならない。この操作は、記述的な属性が同一である別のオブジェクトからも、区別できるものでなければならない。識別するための属性は、システム内で一意であることが保証されていなければならない。これはシステムがどう定義されているかに関係なく、分散していても、オブジェクトが圧縮されていても一意でなければならない。

先に述べたように、オブジェクト指向言語には「識別」のための操作があり、メモリ内でのオブジェクトの位置を比較することによって、2つの参照が同一のオブジェクトを指しているかどうかを判定する。この種の同一性の追跡は、我々の目的にとって脆弱すぎる。オブジェクトを永続的に記憶するほとんどの技術では、オブジェクトがデータベースから検索されるたびに新しいインスタンスが作成されるので、最初の同一性が失われてしまう。オブジェクトがネットワークを通じて伝送される時も、送り先で新しいインスタンスが生成されるため、やはり同一性は失われる。問題をさらに悪化させる可能性があるのは、シ

ステム内に同じオブジェクトの複数のバージョンが存在する場合だ。分散データベースに更新が伝わる時がその一例だ。

これらの技術的問題を単純化するフレームワークを使うとしても、根本的な問題は残る。どうすれば、2つのオブジェクトが同じ概念的なエンティティを表していることがわかるのか？ 同一性の定義はモデルから現れる。同一性を定義するには、ドメインを理解しなければならない。

あるデータの属性や属性の組み合わせが、システム内で一意であると保証されたり、単にそう制限されたりすることもあり得る。このアプローチにより、エンティティに一意のキーが与えられる。例えば、日刊の新聞は、名前と都市および発行日付によって識別できるだろう（ただし、号外と名前の変更に注意！）。

オブジェクトの属性によって構成される真の一意キーがない場合、一般的な解決策としては、各インスタンスにクラス内で一意の記号（番号や文字列など）をつける、というものもある。いったんこのID記号がエンティティの属性として作成され格納されたら、それは不変なもの（immutable）として指定される。開発システムがこのルールを直接強制することができないとしても、このIDを決して変更してはならない。例えば、オブジェクトを平坦にしてデータベースに格納し、その後再構築する際も、ID属性は維持される。技術的なフレームワークがこのプロセスの一助となることもあるが、そうでないなら、エンジニアリングの技術が必要になるだけだ。

たいてい、IDはシステムによって自動生成される。生成アルゴリズムはシステム内で一意になることを保証しなければならないが、並行処理と分散システムにおいて、これは課題となる。そういうIDを生成するのに必要な技術は、本書のスコープを越えているだろう。ここでの目標は、考慮すべき事項がいつ発生するかを指摘して、開発者が解決すべき問題を抱えていることを自覚し、決定的に重要な領域へと関心を絞り込む方法を理解できるようにすることだ。重要なのは、同一性に関する関心が、モデルの特定の側面を軸にしているということを認識することである。識別の手段のために、ドメインに対する注意深い調査が必要になることも多い。

IDが自動生成される場合、ユーザはそれを目にする必要がないかもしれない。IDが必要とされるのは内部だけかもしれない。例えば、連絡先管理アプリケーションにおいて、ユーザが人名でレコードを検索する時がそうだ。このプログラムは、全く同名の2つの連絡先を、単純ながらもはっきりした方法で区別する必要がある。一意の内部的なIDがあれば、まさにそれができる。別々の2つの項目を検索した後で、システムはユーザに連絡先2件を分けて表示するが、IDは表示されないかもしれない。ユーザは会社や場所などを基準に、両者を区別することだろう。

最後に、生成されるIDがユーザにとって関心の対象となる場合がある。小包配達サービ

ス経由で荷物を発送すると、配送会社のソフトウェアが生成した追跡用の番号が与えられるが、これを使うと自分の荷物を識別し、追跡できる。航空券やホテルを予約した場合も、その取引に対する一意の識別子である予約番号が与えられる。

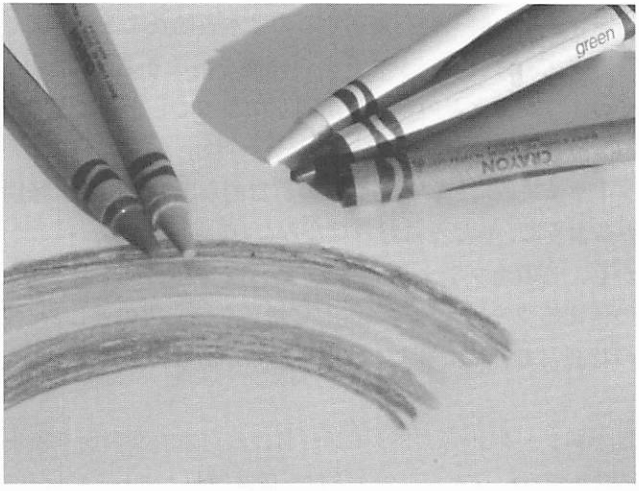
場合によっては、コンピュータシステムの境界を越えて、IDの一意性を適用しなければならないこともある。例えば、医療記録が別々のコンピュータシステムを持つ2つの病院間で交換された場合、理想的なのは、各システムが同じ患者IDを使用することだが、それぞれ独自のシンボルを生成していたらそれは難しい。こういうシステムは、他の機関が発行する識別子を使用することが多い。こうした機関の典型が、政府機関である。米国では、社会保障番号が個人の識別子として病院で使用されることが多い。こうした方法は完全に信頼できるわけではない。誰でも社会保障番号を持っているとは限らず（特に、子供と非米国居住者）、多くの人がプライバシーを理由として使用することに反対している。

そこまで正式ではない場合（レンタルビデオなど）には、電話番号が識別子として使用される。しかし、電話は共有できるし、番号が変わるかもしれない。古い番号が別の人へ再度割り当てられることさえある。

これらの理由から、特別に割り当てられる識別子（マイレージサービス番号など）が使用されることが多く、電話番号や社会保障番号といった他の属性は突き合わせと確認に使用される。いずれの場合も、アプリケーションが外部のIDを必要とした時には、システムのユーザが一意のIDを提供する責任を負うことになり、システムは、発生する例外を十分に処理できる手段を提供しなければならない。

技術的問題がこれほどあると、その根底にある概念的な問題は、たやすく見失われることになる。2つのオブジェクトが同じものである、とはどういう意味なのだろうか？ 各オブジェクトにIDを刻印したり、2つのインスタンスを比較する操作を作成したりすることはいとも簡単だが、そのIDや操作がドメインにおいて意味のある区別と一致しなければ、事態をいっそう混乱させるだけである。だからこそ、同一性を割り当てる操作は人の手による入力を伴うことが多いのだ。例えば、小切手帳の照合をするソフトウェアは、候補になりそうなものを提供しはしても、最終判断はユーザが行うことになっている。

値オブジェクト（VALUE OBJECTS）



多くのオブジェクトには概念的な同一性がない。そういうオブジェクトは、物事の特徴を記述する。

＊　＊　＊

ある子供が絵を描いている時、選ぶマジックの色にはこだわるし、ペン先の細さも気にするかもしれない。しかし、同じ色で同じ太さのマジックが2本あったら、どちらを使うかはおそらく気にしないだろう。マジックがなくなって、新しいセットの同じ色のマジックと交換しても、交換したことを気にしないで、また絵を描き始められる。

冷蔵庫に描かれたいろいろな絵について質問をすれば、子供は自分が描いた絵と妹が描いた絵を、すぐに見分けるだろう。その子と妹には有用な同一性があり、彼らが完成させた絵にも同一性がある。だが、絵の中で、どの線がそれぞれどのマジックで描かれたかを追跡しなければならないとしたら、どんなにややこしいかを想像して欲しい。お絵かきがもう子供の遊びではなくなってしまうだろう。

モデルで最も目立つオブジェクトが、通常はエンティティであり、また、各エンティティの同一性を追跡することが非常に重要であることから、あらゆるドメインオブジェクトに同一性を割り当てようと考えるのは自然なことである。実際、フレームワークの中には、あらゆるオブジェクトに一意のIDを割り当てるものもある。

システムはそうした追跡すべてに対処しなければならず、行えたかもしれない性能の最適化の多くは不可能になる。意味のある同一性を定義して、分散システムやデータベース

ストレージを通じてオブジェクトを追跡する、間違いのない方法を考え出すには、分析の手間がかかる。同様に重要なのだが、人工的な同一性を使うと、誤解を招きかねない。オブジェクトをすべて同じ枠組みへ押し込んで、モデルを台無しにしてしまうのだ。

エンティティの同一性を追跡するのは本質的なことだが、それ以外のオブジェクトに同一性を与えてしまうと、システムの性能を損なうことになり、分析作業が増え、さらに、すべてのオブジェクトの見た目が同じになってしまうことでモデルが台無しになりかねない。

ソフトウェア設計は、複雑さとの恒常的な戦いである。特別な処理が必要な場所だけで行われるように、区別しなければならない。

しかし、このオブジェクトのカテゴリを、単に同一性のないものと見なしてしまうと、我々の使えるツールや語彙は大して増えない。実のところ、これらのオブジェクトには、独自の特徴とモデルに対する独自の意味がある。これは物事を記述するオブジェクトなのだ。

あるオブジェクトが、ドメインにおける記述的な側面を表現し、概念的な同一性を持たない場合、そういうオブジェクトは、値オブジェクトと呼ばれる。値オブジェクトがインスタンス化される際に表現しようとするのは、**何**であるかだけが問題となり、誰であるか、あるいは**どれ**であるかは問われないような設計の要素である。

色は値オブジェクトの一例で、最近の多くの開発システムでベースライブラリの一部として提供されている。文字列や数もそうだ（例えば「4」や「Q」を書く時に、それがどの「4」や「Q」であるかは気にしないだろう）。これらの基本的な例は単純だが、値オブジェクトは必ずしも単純ではない。例えば、色を混ぜるプログラムには、より高機能になった色オブジェクトを組み合わせて別の色を作り出すような豊かなモデルが備わっていることだろう。そういう色は、協力して新しい値オブジェクトを導き出す複雑なアルゴリズムを持っているかもしれない。

値オブジェクトは、他のオブジェクトが組み合わされてできていることもある。家の見取り図を設計するソフトウェアでは、窓の様式それぞれについて、1つのオブジェクトが作成されるかもしれない。この「窓様式」は、高さと幅、およびこれらの属性がどう変更され組み合わされるかを規定するルールと合わせて、「窓」オブジェクトに組み込める。この窓は他の値オブジェクトで構成される入り組んだ値オブジェクトである。次に、この窓は、見取り図におけるより大きな要素である「壁」オブジェクトなどに組み込まれることになる。

値オブジェクトはエンティティを参照することもできる。例えば、オンライン地図サービスで、サンフランシスコからロサンゼルスまでの眺めのいいドライブルートを検索したとすると、パシフィック・コースト・ハイウェイ経由でロサンゼルスとサンフランシスコを

つなぐ経路オブジェクトが得られるかもしれない。参照する3つのオブジェクト（2都市と高速道路）がすべてエンティティであっても、その経路オブジェクトは値オブジェクトになるだろう。

コラム――「住所」は値オブジェクト？　その質問をしているのは誰か？
通販会社のソフトウェアでは、クレジットカードを確認し、小包の宛先とするのに住所が必要である。しかし、同居人も同じ会社に注文したとしても、両者が同じ場所に住んでいると気づくことは重要でない。この場合、住所は**値オブジェクト**である。
郵便サービスのソフトウェアは、配達経路を体系化するために、国は地方、都市、郵便区、街区からなる階層形式となり、これは個々の住所にまで至る。これらの住所オブジェクトは、階層における親から郵便番号を導き出すのであり、郵便サービスが郵便区を割り当てなおすことにしたら、その中のすべての住所が一緒に移動する。ここでは、住所は**エンティティ**だ。
電力会社のソフトウェアにおいては、住所は、電線と電力サービスの目的地に相当する。同居人それぞれが電力サービスを依頼する電話をかけた場合、会社は同じ場所であることに気がつく必要がある。ここでも、住所は**エンティティ**である。もしくは、モデルにおいて、電力サービスを、「住居」のような住所の属性を持った**エンティティ**と関連づけることもできる。それであれば、住所は**値オブジェクト**になるだろう。

値オブジェクトは、しばしば、オブジェクト間のメッセージでパラメータとして渡される。一過性のことも多く、操作のために生成されては破棄される。また、エンティティ（および他の値オブジェクト）の属性としても使用される。人は同一性のあるエンティティとしてモデル化されるかもしれないが、その人の名前は値オブジェクトである。

あるモデル要素について、その属性しか関心の対象とならないのであれば、その要素を値オブジェクトとして分類すること。値オブジェクトに、自分が伝える属性の意味を表現させ、関係した機能を与えること。値オブジェクトを不変なものとして扱うこと。同一性を与えず、エンティティを維持するために必要となる複雑な設計を避けること。

値オブジェクトを構成する属性は、概念的な統一体を形成すべきである[2]。例えば、街区、都市、郵便番号は人オブジェクトの別々な属性であってはならない。それらはある住所全体の一部であり、それによって人オブジェクトはよりシンプルになり、より凝集度の高い値オブジェクトができる。

2　Ward Cunningham による完結した値（WHOLE VALUE）パターン。

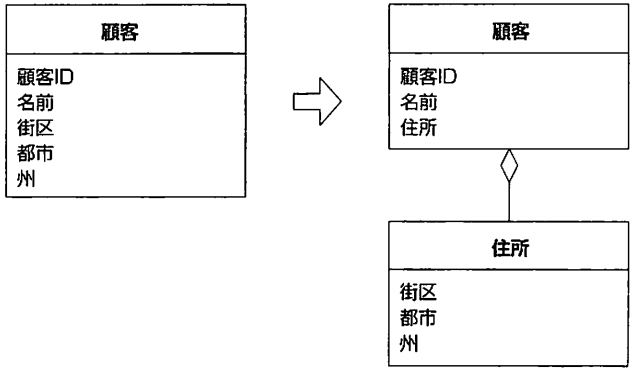


図5.6 **値オブジェクト**は、**エンティティ**に関する情報を与えることができる。それは概念的な統一体でなければならない

* * *

値オブジェクトを設計する

値オブジェクトについては、それがどのインスタンスであるかは気にならない。このように制約がないことで、設計が自由になり、設計を簡素化したり、性能を最適化したりできるようになる。ここには、コピーと共有および不変性に関する選択も含まれる。

2人の人物が同名だったとしても、それで2人が同一人物になるわけではないし、お互いに入れ換われるようになるわけでもない。しかし、名前を表すオブジェクトは相互に交換できる。重要なのは、名前の綴りだけだからだ。名前オブジェクトは、1つめの人オブジェクトから2つめに**コピーする**ことができる。

実際、2つの人オブジェクトは、独自の名前インスタンスを必要としないかもしれない。同じ名前オブジェクトは、2つの人オブジェクト間で**共有する**（それぞれが同一の名前インスタンスへのポインタを持つ）こともでき、それによってふるまいや同一性が変わることはない。つまり、片方の名前が変更されるまで、両者のふるまいは正しいはずなのだ。片方の名前が変更された場合、もう一方の人の名前も変わってしまうかもしれない！ これを防いで、オブジェクトを安全に共有するためには、オブジェクトは**不変**でなければならない。これはつまり、完全に置き換える以外、変更することはできないということだ。

あるオブジェクトが、属性の1つを引数や戻り値として別オブジェクトに渡す場合にも、同じ問題が発生する。所有者の手を離れている間、さすらうオブジェクトにはどんなことでも起こり得る。所有者の不変条件に違反するといった、所有者を破壊するようなかたちで値が変更されるかもしれないのだ。この問題は、渡すオブジェクトを不変にするか、コピーを渡すことで回避される。

パフォーマンスチューニングのために、余分に選択肢を作っておくことは重要になるかもしれない。値オブジェクトは数が多くなりがちだからだ。家の設計を行うソフトウェアの例がこのことを示唆している。電気のコンセントそれぞれが別個の値オブジェクトだったら、家1件の見取り図の1バージョンに対して、そのオブジェクトが100個できるかもしれない。しかし、すべてのコンセントが相互に交換できると見なせば、コンセントのインスタンス1つだけを共有し、それを100回参照すればよい（フライウェイト（FLYWEIGHT）（Gamma他 1995）の一例）。巨大なシステムでは、この種の影響が何千倍になることもあり、こうした最適化が、速くて使い勝手のよいシステムと、何百万もの冗長なオブジェクトがぎっしり詰まった遅いシステムとの差を生み出す。これは、エンティティでは使用できない最適化の技の一例である。

コピーと共有を比較した時にどちらが高くつくかは、実装環境に依存する。コピーをすると、何百ものオブジェクトができてシステムの動きを鈍くするかもしれないが、共有も分散システムの速度を遅くするかもしれない。2台のマシン間で渡されるのがコピーであれば、メッセージが送信されるのは一度であり、コピーは受信側のマシンで独立して生き続ける。しかし、単一のインスタンスが共有されていたら、参照のみが渡され、相互作用のたびにオブジェクトへメッセージを戻さなければならなくなる。

インスタンスを共有するのは、次のような最も有益で面倒でない場合に限定するのが最適である。

- データベース内で、スペースやオブジェクト数を節約することがきわめて重要である場合
- 通信オーバーヘッドが低い場合（中央集中型サーバなど）
- 共有オブジェクトが完全に不変である場合

属性やオブジェクトの不変性は、言語と環境によっては宣言することができるが、できないこともある。こうした機能は、設計上の決定についてコミュニケーションするのに役立つが、本質的ではない。モデルで行う区別の多くは、現在のツールとプログラミング言語を使ったとしても、実装において明示的に宣言することはできない。例えば、エンティティであることを宣言して、同一性の演算を自動的に強制することはできないのだ。しかし、概念的な区別を言語が直接サポートしないからといって、その区別が役に立たないわけではない。実装において暗示されるにすぎないルールを守るためには、さらに多くの規律が必要だということである。これは、命名規約と精選された文章、そして多くの議論によって補強できる。

値オブジェクトが不変である限り、変更の管理は単純だ。完全に置き換える以外に変更はないからだ。不変オブジェクトは、電気コンセントの例にあったとおり、自由に共有できる。ガベージコレクションが信頼できるなら、削除するにはオブジェクトへの参照をす

べて取り除くだけでよい。値オブジェクトが設計において不変であると指定されていれば、開発者は純粋に技術的な基準によって、コピーと共有といった問題について自由に意思決定できるし、そうしたオブジェクトの特定のインスタンスにアプリケーションが依存しないことがわかっているから、安心できるのだ。

値オブジェクトを定義し、それを不変と指定するのは、次の一般的なルールに従っている事例の1つだ。すなわち、モデルにおける不必要な制約を避けることで、開発者は純粋に技術的なパフォーマンスチューニングを自由に行えるようになる。本質的な制約を明示的に定義することで、開発者は設計を微調整できるようになり、一方で意味のあるふるまいを誤って変更してしまうことがなくなる。そうした設計の微調整は、特定のプロジェクトで使用される技術にきわめて特化していることが多い。

コラム——特殊な場合：いつ可変性を認めるべきか？

不変性によって、実装は大幅に単純化され、安全に共有や参照渡しができるようになる。また、値という言葉の持つ意味とも矛盾しない。属性の値が変化したら、既存のオブジェクトを修正するのではなく、別の**値オブジェクト**を使用する。それでも、性能を考慮して、**値オブジェクト**が可変であることを許可する方がよい場合もある。以下の要因があれば、可変な実装を選ぶようになるかもしれない。

- **値**が頻繁に変化する場合
- オブジェクトの生成や削除が高くつく場合
- 置き換えによって（修正ではなく）クラスタリングが妨げられる場合（前述の例で説明）
- **値**を共有することがあまりない場合、またはクラスタリングを改良するためや他の技術的理由からそういう共有を見合わせる場合

繰り返しになるが、**値**の実装を可変にするなら、それを共有してはならない。共有してもしなくても、可能ならば**値オブジェクト**を不変なものとして設計すること。

例 5.2——値オブジェクトを使ってデータベースをチューニングする

データベースは、最も物理的なレベルで見ると、ディスク上の物理的な位置にデータを設定しなければならず、機械部品が移動してそのデータを読み取るには時間がかかる。洗練されたデータベースは、これらの物理アドレスをクラスタ化して、1回の物理操作によって関係するデータをディスクから取り出せるようにしている。

あるオブジェクトが他の多くのオブジェクトから参照されている場合、参照している側のオブジェクトの中には、参照されているオブジェクトの近く（同一ページ上）になくて、データを得るのに追加の物理操作が必要となるものもある。同一インスタンスへの参照を共有するのではなく、コピーを生成することによって、多くのエンティティの一属性として機能する値オブジェクトを、それを使用する各エンティティと同じページに格納することができる。同一データに対して複数のコピーを格納するこの技術は**非正規化**と呼ばれ、ストレージの容量や保守の単純化よりも、アクセス時間の方が決定的に重要な場合に、しばしば用いられる。

関係データベースでは、特定の値を独立したテーブルに入れて関連を生成するよりも、その値オブジェクトを所有するエンティティのテーブルに入れたいことがあるかもしれない。分散システムでは、値オブジェクトへの参照を別サーバに持つと、メッセージに対する応答が遅くなるかもしれないから、代わりにオブジェクト全体のコピーを相手のサーバに渡すべきだ。扱うのが値オブジェクトだから、こうしたコピーは自由に生成できる。

値オブジェクトを含む関連を設計する

関連についての前述の議論のほとんどは、エンティティと値オブジェクトにも同じように当てはまる。モデルの中の関連は、数が少なく単純であればあるほどよい。

しかし、両者の間には違いもある。エンティティの間に双方向の関連があると、保守するのが難しくなるかもしれない。これに対して、2つの値オブジェクトの間に双方向の関連があっても意味をなさない。つまり、同一性がなければ、あるオブジェクトが参照し返しているのは、自身を参照しているのと同じ値オブジェクトだと言っても無意味なのだ。せいぜい言えるのは、自身を参照しているオブジェクトと**等しい**ものを参照するということだが、その不変条件はどこかで強制しなければならないだろう。また、不変条件を強制できて、双方向の参照を設定することができたとしても、そういう配置が役立つ例はなかなか思いつかない。したがって、値オブジェクト同士の双方向の関連は完全に取り除くように試みること。もし、最終的にそういう関連がモデルで必要だと思われたら、そもそも、そのオブジェクトを値オブジェクトとして宣言するという決定自体を考え直すべきである。おそらく、そのオブジェクトには、まだ明示的には認識されていない同一性があるはずである。

エンティティと値オブジェクトは、従来のオブジェクトモデルにおける主要な要素だが、実践的な設計者たちは、これとは別のある要素を使用するようになってきた。それがサービスである…。

サービス（SERVICES）



時には、単純に「物」とはできないこともある。

場合によっては、設計をできる限り明確にしつつ、実践的に進めていくと、概念的にどのオブジェクトにも属さないような操作が含まれることがある。強引に決着をつけるのではなく、問題領域に引かれる自然な輪郭に従って、モデルの中に明確にサービスを含めればよい。

＊　＊　＊

ドメインの重要な操作でありながら、エンティティにも値オブジェクトにも自然な落ち着き場所を見つけることのできないものがある。その中には、本質的に活動や行動であって、物事でないものもあるが、我々のモデリングパラダイムはオブジェクトなので、とにかくオブジェクトに当てはめてみよう。

ここで犯しがちな間違いは、ふるまいを適切なオブジェクトへ当てはめることをあまりに早くあきらめ、徐々に手続き型プログラミングへと滑り落ちてしまうことだ。しかし、定義に合わない操作をオブジェクトに押し込めば、そのオブジェクトは概念的に明確でなくなってしまって、理解するのもリファクタリングするのも難しくなる。複雑な操作は容易にシンプルなオブジェクトを浸食し、その役割をあいまいにしかねない。さらに、そういう操作は、多くのドメインオブジェクトを一緒に調整したり動かしたりすることが多いので、つけ加えられた責務によってそれらのオブジェクト全部に依存関係が生まれ、独立して理解できたはずの概念を、もつれさせてしまう。

時には、サービスがモデルオブジェクトになりすまし、何らかの操作を行う以上の意味を持たないオブジェクトのように見えることがある。こういう「実行者」は結局のところ、

「マネージャ」などで終わる名前を持つことになる。これらは独自の状態を持たず、実行する操作以外にはドメインにおいて意味を持たない。それでも、サービスという解決策により、この独特なふるまいは少なくとも落ち着き先を得て、実際のモデルオブジェクトを混乱させることがなくなる。

ドメインから生まれる概念の中には、オブジェクトとしてモデル化すると不自然なものもある。こうしたドメインで必要な機能をエンティティや値オブジェクトの責務として押しつけると、モデルに基づくオブジェクトの定義を歪めるか、意味のない不自然なオブジェクトを追加することになる。

サービスとは、モデルにおいて独立したインタフェースとして提供される操作で、エンティティと値オブジェクトのようには状態をカプセル化しない。サービスは技術的なフレームワークでは一般的なパターンだが、ドメイン層にも適用できる。

サービスという名前は、他のオブジェクトとの関係性を強調している。エンティティや値オブジェクトとは異なり、純粋にクライアントに対して何が実行できるかという観点から定義されるのだ。サービスは、実体よりも活動、つまり名詞よりも動詞にちなんで命名される傾向がある。サービスもやはり抽象的で意図的な定義を持つことはできるのであり、オブジェクトの定義とは少し趣が異なるだけだ。また、定義された責務を負う必要もあり、その責務と、責務を満たすためのインタフェースはドメインモデルの一部として定義すべきである。操作名はユビキタス言語に由来しなければならず、そうでなければユビキタス言語に取り入れられなければならない。引数と結果はドメインオブジェクトであるべきだ。

サービスは節度を持って使用すべきで、エンティティと値オブジェクトからすべてのふるまいを奪ってはならない。しかし、実際に、操作が重要なドメインの概念なのであれば、サービスはモデル駆動設計の自然な一部を形成する。実際には何も表していない偽のオブジェクトとして宣言するよりも、モデルの中でサービスとして宣言されている方が、独立した操作によって誰かが誤解することはない。

優れたサービスには3つの特徴がある。

1. 操作がドメインの概念に関係しており、その概念がエンティティや値オブジェクトの自然な一部ではない。
2. ドメインモデルの他の要素の観点からインタフェースが定義されている。
3. 操作に状態がない。

ここで状態がないというのは、どのクライアントでも、特定のサービスの任意のインスタンスを使うにあたって、インスタンスの持つ個々の履歴を気にする必要がない、という意味である。サービスを実行すると、グローバルにアクセス可能な情報を使用し、そのグローバルな情報を変更することもあるかもしれない（つまり、副作用がある場合もあるということだ）。しかし、ほとんどのドメインオブジェクトが持っているような、オブジェク

トの内部にあって、自身のふるまいに影響を与える状態は、サービスには存在しない。

ドメインにおける重要なプロセスや変換処理が、エンティティや値オブジェクトの自然な責務ではない場合、その操作は、サービスとして宣言される独立したインタフェースとしてモデルに追加すること。モデルの言語を用いてインタフェースを定義し、操作名が必ずユビキタス言語の一部になるようにすること。サービスには状態を持たせないこと。

＊　＊　＊

サービスと隔離されたドメイン層

このパターンは、ドメインにおいてそれ自体が重要な意味を持つサービスに焦点を絞っているが、もちろん、サービスはドメイン層だけで使用されるのではない。ドメイン層に属するサービスを他のレイヤのサービスから区別することと、その区別をはっきり保つように責務を分解することには、注意が必要である。

文献で論じられているサービスのほとんどは純粋に技術的なものであり、こうしたサービスはインフラストラクチャ層に属する。ドメイン層とアプリケーション層のサービスは、これらインフラストラクチャ層のサービスと協力して動作する。例えば、ある銀行のアプリケーションでは、口座残高が一定限度額を下回ると、顧客に電子メールを送信するかもしれない。この電子メールシステムはインタフェースによってカプセル化され、場合によっては、代わりの通知手段もそこに含まれるかもしれない。このインタフェースが、インフラストラクチャ層のサービスである。

アプリケーションサービスをドメインサービスから区別するのは、さらに難しいかもしれない。アプリケーション層は通知を指示する責務を負う。ドメイン層は、限度額に達したかどうかを判別する責務を負うが、この仕事はサービスを必要としないだろう。これは「口座」オブジェクトの責務にふさわしいからだ。その銀行アプリケーションは資金振替の責務も負っているかもしれない。あるサービスが、資金振替のために適切な引き落としと預け入れを行うように作られていれば、その機能はドメイン層に属するだろう。資金振替は銀行ドメインの言語において意味を持ち、本質的なビジネスロジックを含んでいるからだ。技術的なサービスにはビジネスに関する意味が一切ないはずである。

多くのドメインサービスやアプリケーションサービスは、エンティティと値オブジェクトで構成される集合体の上に構築され、ドメインに本来備わっている能力をまとめ上げて、実際に何らかの処理を行うスクリプトのようにふるまう。エンティティと値オブジェクトは、ドメイン層の持つ能力への便利なアクセスを提供するには、粒度が細かすぎることが多い。ここで、ドメイン層とアプリケーション層の間に引かれている微妙な境界線に行き当たる。例えば、銀行業アプリケーションが、我々が分析できるよう取引を変換してスプ

レッドシートファイルにエクスポートできるとすると、そのエクスポートはアプリケーションサービスである。銀行業ドメインにおいて「ファイル形式」には意味がなく、ビジネスルールも関係してはいない。

その一方、資金をある口座から別の口座に振り替える機能はドメインサービスである。重要なビジネスルール（例えば、適切な口座に振り込みと引き落としを行うこと）を組み込んでいるし、「資金振替」は意味のある銀行用語だからだ。この場合、サービスはそれ自体ではたいしたことを行わず、2つの口座オブジェクトに処理のほとんどを依頼する。しかし、口座オブジェクトに「振替」の操作を入れては扱いにくくなる。この操作は2つの口座と何らかのグローバルなルールを伴っているからだ。

資金振替オブジェクトを作成する際に、振り込みと引き落としというこの2つの記帳に加えて、振替に関するルールと履歴を表現させたいと思うかもしれない。しかし、まだ銀行間ネットワークのサービスに対する呼び出しが残っている。その上、たいていの開発システムの場合、ドメインオブジェクトと外部リソースの間に直接インタフェースを設けるとぎこちなくなる。そういう外部サービスには、モデルの用語で入力を受け取り、おそらく資金振替オブジェクトを結果として戻すファサード（FACADE）を被せればよい。しかし、仲介が何であれ、それが我々のものでないとしても、それらのサービスは資金振替というドメインの責務を果たしている。

サービスをレイヤに分割する

アプリケーション	**資金振替アプリケーションサービス** ◉入力（XMLリクエストなど）を理解する。 ◉処理を実行するよう、ドメインサービスにメッセージを送信する。 ◉確認を待つ。 ◉インフラストラクチャサービスを使用して、通知を送信することを決定する。
ドメイン	**資金振替ドメインサービス** ◉必要な口座オブジェクトおよび元帳オブジェクトとやり取りし、適切な引き落としと振り込みを行う。 ◉結果（振替の可否など）の証跡を出力する。
インフラストラクチャ	**通知送信サービス** ◉アプリケーションの指示に従って、電子メールや手紙、その他のメッセージを送信する。

粒 度

このパターンの議論では、概念をサービスとしてモデル化することで得られる表現力を強調しているが、このパターンは、クライアントをエンティティと値オブジェクトから分離する手段としても、ドメイン層のインタフェースの粒度を制御する手段としても価値があ

る。

中粒度で状態を持たないサービスは、巨大なシステムで再利用しやすい。これは、重要な機能をシンプルなインタフェースの後ろにカプセル化しているためだ。また、細粒度のオブジェクトは、分散システムにおいては、非効率的なメッセージングにつながるかもしれない。

前述した通り、細粒度のドメインオブジェクトを用いると、ドメイン層からアプリケーション層へ知識が流出するかもしれない。アプリケーション層は、ドメインオブジェクトのふるまいが組み合わされる場所だからだ。きわめて詳細なレベルでの相互作用が複雑であるため、結局アプリケーション層で処理されることになり、ドメインの知識がアプリケーションやユーザインタフェースコードに這い出るのを許して、その知識はドメイン層から失われてしまう。ドメインサービスを慎重に導入すれば、複数のレイヤ間で境界を鮮明に維持できる。

このパターンでは、クライアントの操作や用途の広さよりも、インタフェースの単純さが優先される。また、このパターンにより、巨大なシステムや分散システムでコンポーネントをパッケージングするのに、非常に便利な中粒度の機能が提供される。そして、サービスが、ドメインの概念を表現するのに最も自然な方法であることもある。

サービスへのアクセス

J2EEやCORBAのような分散システムアーキテクチャでは、サービスを公開するための特別な仕組みが提供されている。こうしたサービスは使い方に慣例があり、分散とアクセスの機能が追加されている。しかし、そういうフレームワークはプロジェクトで常に使用されるとは限らないし、使用されるとしても、ただ関心事を論理的に分離することだけが目的ならば、やりすぎに見える。

サービスへのアクセスを提供する手段は、特定の責務を切り出すという設計上の意思決定ほどには重要ではない。サービスのインタフェースを実装するのに、「実行者」オブジェクトで十分な場合もある。単純なシングルトン（SINGLETON）（Gamma他 1995）は、アクセスを提供するために簡単に作成できる。コーディング規約があれば、こうしたオブジェクトはサービスのインタフェースを単に配布する仕組みであって、意味のあるドメインオブジェクトではないということを明確にすることができる。手の込んだアーキテクチャを使用すべきなのは、システムを分散するか、そうでなければ、フレームワークの機能をどうしても利用する必要がある場合だけだ。

モジュール（MODULES）（別名 パッケージ（PACKAGES））

モジュールは、長い歴史を持つ確立された設計要素である。技術的な考慮も一因ではあるが、認知のための負荷を下げることが、モジュール化にあたっての主要な動機になっている。モジュールによって、モデルについて2つの見方ができるようになる。すなわち、全体によって圧倒されることなく、モジュール内部の詳細を見ることができるし、その一方で、内部の詳細を無視した上で、モジュール間の関係性を見ることもできるのだ。

ドメイン層のモジュールは、モデルにおいて意味のある一部として現れ、より大きな尺度でドメインのことを語らなければならない。

＊ ＊ ＊

誰もがモジュールを使用するが、モデルのれっきとした一部として扱う人はほとんどいない。コードは、技術的アーキテクチャの側面から開発者の仕事の分担に至るまで、あらゆるカテゴリに分類される。大量のリファクタリングを行う開発者でも、プロジェクトの初期に考え出したモジュールで満足する傾向がある。

モジュール間では低結合、モジュール内では高凝集が必要なことは自明の理だ。結合と凝集について説明すると、技術的な指標のように聞こえがちで、関連と相互作用の分布に基づいて機械的に判断されやすい。しかし、モジュールに分割されるのはコードだけでない。概念も分割されるのだ。人が一度に考えられる物事の数には限りがあるし（ゆえに低結合）、首尾一貫していない思考の断片は、思考がばらばらに溶け合ったスープのようなもので、理解するのが難しい（ゆえに高凝集）。

低結合と高凝集は、一般的な設計の原則であり、モジュールに対するのと同じように、個々のオブジェクトに対しても適用される。しかし、より粒度の大きいこのようなモデリングと設計においては、とりわけ重要なのだ。これらの用語は長いこと使用されてきており、パターン形式での説明は『実践UML』（Larman 1998）に見られる。

2つのモデル要素が別々のモジュールに分離される時はいつでも、両者の関係は分離前ほど直接的ではなくなり、そのため、設計におけるそれぞれの位置づけを理解するためのオーバーヘッドが増える。モジュール間の結合度を低くすることによって、このコストが最小限に抑えられ、1つのモジュールの内容を分析する際に、相互作用する他のモジュールへの参照を最小ですませることができる。

同時に、優れたモデル要素には相乗効果があり、モジュールを適切に選択すれば、概念上の関係性が特に豊かなモデル要素がまとめられる。関係する責務を負ったオブジェクトの凝集度が高いことで、モデリングと設計は1つのモジュール内に集中できる。これは人間の頭で容易に扱える程度の複雑さだ。

モジュールとそれより小さい要素は共に進化すべきだが、普通はそうならない。モジュールは、初期段階の複数のオブジェクトをまとめ上げるために選択される。その後、そうしたオブジェクトは既存のモジュールの定義による境界内に留まるかたちで変化する傾向がある。モジュールをリファクタリングするのは、クラスをリファクタリングするよりも手間がかかり、混乱を招きもするので、それほど頻繁に行うことはできないだろう。しかし、モデルオブジェクトが素朴で具体的なところから始まり、次第により深い洞察を明らかにするように変わっていく傾向があるのと同じく、モジュールも巧妙で抽象的になることがある。ドメインに対する理解が変化していくのに合わせて、それをモジュールに反映させることで、モジュールの中のオブジェクトがより自由に進化できるようになる。

ドメイン駆動設計にある他のパターンと同じく、モジュールも**コミュニケーションの仕組み**である。分割されるオブジェクトの**意味**が、モジュールの選択を駆動させなければならない。いくつかのクラスを1つのモジュールに一緒に入れるということは、その設計を見る次の開発者に、それらをひとまとめに考えるよう伝えていることになる。モデルが物語だとすれば、モジュールは章に相当する。モジュールの名前はその意味を伝える。そういう名前がユビキタス言語に組み込まれる。「では、『顧客』モジュールについて話しましょう」とビジネスエキスパートに言えば、会話をするためのコンテキストが整うのだ。

それゆえ：

モジュールを選択する際には、システムに関する物語を伝え、概念の凝集した集合を含んでいるものを選ぶこと。こうすることで、モジュール間は低結合になることが多い。だが、そうでない場合は、概念のもつれをほぐすようにモデルを変更する方法を探すか、意味のある方法で要素をまとめられるような、モジュールの基礎となる概念を見逃していないか模索すること。相互に独立して理解でき、理論づけることのできる概念という意味での低結合を実現するよう試みること。モデルがドメインの高次の概念に従って分割され、対応するコードも同じように分離されるようになるまで、モデルを改良すること。

モジュールには、ユビキタス言語の一部になる名前をつけること。モジュールとその名前はドメインに対する洞察を反映していなければならない。

概念上の関係性について考えることは、技術的な評価の代わりではない。同じ問題について別のレベルで見ているだけで、どちらも達成されなければならないのだ。しかし、モデル重視の思考は、偶発的ではない、より深い解決策を生み出す。トレードオフが避けられない場合は、概念的な明確さの方を選んだ方がよい。それによってモジュール間の参照が増えたり、モジュールに対して変更が加えられた時に起こる連鎖反応が増えたりしても、やはりそうすべきだ。そうした問題は、モデルが伝える物語を開発者が理解していれば、処理することができるのだ。

＊　＊　＊

アジャイルモジュール

モジュールは、モデルの残りの部分と共に進化する必要がある。これは、モジュールをリファクタリングする時には、モデルやコードと一緒に行うことを意味する。しかし、こういうリファクタリングは行われないことが多い。モジュールを変更すると、広範囲に及ぶコードの更新が必要になりがちだ。そういう変更はチームのコミュニケーションに混乱を生じさせるかもしれず、ソースコード管理システムなどの開発ツールを混乱させる可能性すらある。その結果、モジュールの構造と名前は、クラスに比べて、モデルのはるか初期のかたちを反映することが多い。

モジュールの選択において初期に間違いが犯されることは避けられないが、そのことが高結合につながり、それによってリファクタリングが困難になる。リファクタリングを行わなければ、惰性に流されるばかりだ。こうしたことを克服するためには、問題が起こりそうな場所に関する経験を基に、歯を食いしばってモジュールを再構成するしかない。

開発ツールやプログラミングシステムによっては、問題がさらに深刻になる。実装が基づく開発技術が何であれ、モジュールのリファクタリング作業を最小限とし、他の開発者とのコミュニケーションにおける混乱を最小にする方法を探す必要がある。

例5.3 —— Javaにおけるパッケージのコーディング規約

Javaではインポート（依存関係）を個々のクラスで宣言しなければならない。パッケージが他のパッケージに依存すると考えるモデラもおそらくいるが、それをJavaで宣言することはできないのだ。一般的なコーディング規約では、具体的なクラスのインポートを推奨しており、その結果、コードは次のようになる。

```
ClassA1
import packageB.ClassB1;
import packageB.ClassB2;
import packageB.ClassB3;
import packageC.ClassC1;
import packageC.ClassC2;
import packageC.ClassC3;
...
```

あいにくJavaの場合は、個々のクラスへインポートするのを避けられないが、少なくともパッケージをまとめて一度にインポートすることはできる。そうすることで、パッケージがきわめて凝集度の高い単位だという意図を反映すると同時に、パッケージ名をいちいち書き換える労力を削減できる。

```
ClassA1
import packageB.*;
import packageC.*;
...
```

このテクニックを使うことは、確かに2つの尺度（クラスはパッケージに依存している）を混ぜ合わせることを意味するが、先に挙げたおびただしい数のクラスの一覧よりは、多くを伝えている。つまり、特定のモジュールに対する依存関係を作成するという意図を伝えているのである。

個別のクラスが他のパッケージに置かれている特定のクラスに強く依存しているにも関わらず、それを含むモジュールには他のモジュールへの概念的依存関係がなさそうな場合は、クラスを移動するか、モジュール自体を検討しなおす必要があるだろう。

インフラストラクチャ駆動パッケージングの落とし穴

パッケージングの決定に際して強い原動力となるのは、技術的なフレームワークである。その中には有益なものもあれば、阻止しなければならないものもある。

非常に有効なフレームワーク標準の例としては、レイヤ化アーキテクチャを強制することで、インフラストラクチャとユーザインタフェースコードを別々のグループのパッケージに入れ、ドメイン層を物理的に分離してそれ自身のパッケージに残すというものがある。

他方、ティア化アーキテクチャ（tiered architecture）では、モデルオブジェクトの実装が断片化されるかもしれない。フレームワークの中には、1つのドメインオブジェクトの責務を複数のオブジェクトに広げ、それらのオブジェクトを別個のパッケージに入れることによってティアを作り出すものもある。例えば、J2EEでの一般的なプラクティスでは、データとデータアクセスを「エンティティBean」に置く一方、関連するビジネスロジックは「セッションBean」に置く。各コンポーネントの実装がより複雑になることに加え、こうして分離することでオブジェクトモデルから凝集度が奪われてしまう。オブジェクトの最も基本的な構想の1つは、データを操作するロジックによって、そのデータをカプセル化することである。この種のティア型実装は致命的ではない。両方のコンポーネントが一緒になって、1つのモデル要素の実装を構成すると見なすことができるからだ。しかし、さらに悪いことに、エンティティBeanとセッションBeanはたいてい別々のパッケージに分離されるのだ。ここまで来ると、多様なオブジェクトを見てから、それらを頭の中で組み合わせて単一の概念的なエンティティにするのは、労力がかかりすぎる。モデルと設計のつながりが失われるのだ。ベストプラクティスとしては、エンティティオブジェクトより大きい粒度でEJBを使用し、ティアを分離することによるマイナス面を小さくするというものがある。しかし、細粒度のオブジェクトも複数のティアに分けられていることが多い。

例えば、私がこれらの問題に遭遇したのは、比較的理にかなった運営をしているプロジェクトにおいてだったが、概念的なオブジェクトが、実際には4つのティアに分割されていた。分割にはそれぞれもっともな理由があった。最初のティアはデータ永続化層で、関

係データベースへのマッピングとアクセスを扱っていた。次に来るティアは、特定のコンテキストに依存しない、オブジェクトに本来備わっているふるまいを扱っていた。その次は、アプリケーション特有の機能を、オブジェクトのふるまいに重ね合わせるためのティアであった。4番目のティアは公開インタフェースとなることが意図され、下位のあらゆる実装から分離されていた。この構成は少々複雑すぎたが、レイヤは適切に定義され、関心事の分離についても整然としていた。1つの概念的なオブジェクトを構成するのに、すべての物理的なオブジェクトを頭の中でつなげることで我慢することもできただろう。複数の側面を分離することが役立つことも時にはあった。とりわけ、永続化のためのコードが取り除かれたことで、ずいぶん整理されたのだ。

しかし、何にも増して、このフレームワークでは各階層を別個のパッケージに入れなければならず、そのパッケージ名も、規約に従ってティアを識別する名前をつけなければならなかった。このために、分割について考える余地が奪われてしまった。結果として、ドメインの開発者は、モジュール（それぞれが4倍になる）をあまり多く作りすぎないようにして、変更もほとんど行わなくなった。モジュールをリファクタリングする労力が桁外れだったからである。さらにひどかったのは、単一の概念的なクラスを定義するすべてのデータとふるまいを突き止めることが（階層化による間接性と相まって）非常に難しかったため、開発者にはモデルについて考える精神的な余裕がほとんど残されなかったことだ。アプリケーションは完成したが、ドメインモデルは貧血性になっていて、アプリケーションの求める基本的なデータベースアクセスの要求を満たす程度であり、ふるまいを提供しているのは数少ないサービスだった。モデル駆動設計によって得られたはずの力が限られてしまっていたのは、コードがモデルを透過的に明らかにしておらず、開発者がモデルを使えるようにもなっていなかったからである。

この種のフレームワーク設計は、2つの合理的な問題に取り組もうとしている。1つは、関心事の論理的な分割だ。すなわち、あるオブジェクトはデータベースのアクセスに関する責務を負い、別のオブジェクトはビジネスロジックの責務を負う。そういう分割によって、各階層の働きを（技術的なレベルで）理解しやすくなり、レイヤ間の切り替えが容易になる。問題は、アプリケーション開発にかかるコストが意識されていないことだ。本書はフレームワーク設計のためのものではないから、そういった問題に対する解決策の代わりになるものは説明しないが、解決策があることは確かである。また、他に選択肢がない場合でも、こういう利点よりは凝集度の高いドメイン層を重視した方が賢明である。

こうしたパッケージングの方式に対するもう1つの動機が、ティアの分散である。これは、コードが実際に別々のサーバに配置される場合には非常に説得力を持つが、普通はそうはならない。柔軟性が求められるのは必要となった時だけだ。モデル駆動設計によってもたらされる力を期待するプロジェクトでは、差し迫った問題を解決するのでない限り、

この犠牲はあまりにも大きすぎる。

技術駆動の手の込んだパッケージング構成を行うと、代償を2つ背負うことになる。

- フレームワークの持つ分割の規約によって、概念的なオブジェクトを実装する要素がばらばらに分けられてしまうとしたら、もはやコードがモデルを明らかにすることはなくなる。
- 頭の中で元通りに縫い合わせることができる分割には限りがあり、フレームワークがそれを使い切ってしまえば、ドメインの開発者はモデルを意味のある塊に分けることができなくなる。

物事はシンプルなままにしておくのが一番である。技術的な分割のルールは、最小限のものを選択すべきだ。これは、技術的な環境にとって本質的であるか、実際に開発の助けになるものだけにする、ということだ。例えば、複雑なデータ永続化コードをオブジェクトの持つふるまいの側面から切り離すことで、リファクタリングは容易になるかもしれない。

別々のサーバにコードを分散させようという意図が実際にない限り、単一の概念オブジェクトを実装するコードはすべて、同一のオブジェクトにならなくても、同一のモジュールにまとめること。

昔から存在する基準である「高凝集／低結合」を利用しても、同じ結論に至っただろう。ビジネスロジックを実装する「オブジェクト」と、データベースアクセスの責務を持つオブジェクトとのつながりはきわめて強いので、結合度は非常に高い。

他にも落とし穴はある。フレームワークの設計や、企業やプロジェクトが作った規約程度でも、ドメインオブジェクトの自然な凝集を不明瞭にして、モデル駆動設計を弱めてしまうかもしれない。これも、要点は同じことだ。制約があるせいで、もしくはパッケージが大量に必要であるというだけでも、ドメインモデルの必要性に合わせた別のパッケージ構成を使うことが認められなくなってしまうのだ。

ドメイン層を他のコードから分離するためにパッケージングを使用すること。そうでなければ、ドメインの開発者にできる限り選択の余地を残し、モデルと設計上の選択をサポートするように、ドメインオブジェクトをパッケージングできるようにすること。

1つ例外が発生するのは、コードが宣言的設計（第10章で説明）に基づいて生成される時である。その場合、開発者はコードを読む必要がないので、邪魔にならないように別個のパッケージに入れて、開発者が実際に取り組まなければならない設計要素を混乱させないようにした方がよい。

モジュール性は、設計が大きく複雑になるにつれて、ますます重要になる。本節では、基本的な考慮事項を紹介している。第4部「戦略的設計」のほとんどは、大きなモデルや設計を分割してパッケージングするためのアプローチと、理解を助けるために焦点を与え

る方法を説明している。

ドメインモデルに由来する概念は、それぞれ、実装上の一要素に反映されなければならない。エンティティと値オブジェクトおよびその関連は、数少ないドメインサービスやそれらを構成するモジュールと合わせて、実装とモデルが直接的に一致する場所である。実装におけるオブジェクトとポインタおよび検索の仕組みは、当然、モデル要素へ素直にマッピングしなければならない。そうでない場合は、コードを見直すか、モデルに立ち戻って変更するか、あるいはその両方を行うこと。

ドメインオブジェクトに対して、それが表現している概念と密接に関係しないものを追加したくなる誘惑には、追加したいものが何であれ抵抗すること。ドメインオブジェクトには、設計要素としてすべきことがある。それは、モデルを表現するということだ。他にも、ドメインに関する実行すべき責務や、システムを機能させるために管理すべきデータもあるが、それらはこうしたオブジェクトには属さない。第6章では、ドメイン層に課せられる技術的な責務を遂行する補助的なオブジェクトについて議論する。こうしたオブジェクトには、データベース検索を定義するものや、複雑なオブジェクトの生成をカプセル化するものがある。

本章で解説した4つのパターンにより、オブジェクトモデルの基本要素が提供される。しかし、モデル駆動設計を行うということは、何もかもを画一的にオブジェクトとして扱うということを必ずしも意味するわけではない。ルールエンジンのように、ツールによってサポートされる他のモデルパラダイムもある。プロジェクトは、そうしたパラダイムの間で現実的なトレードオフを行う必要がある。これら他のツールやテクニックは、モデル駆動設計という目標に向かうための手段であって、モデル駆動設計の代わりにはならない。

モデリングパラダイム

モデル駆動設計を実践するためには、適用される特定のモデリングパラダイムに合わせた実装技術が必要である。そういうパラダイムはこれまで多くが試されてきたが、実際に広く使用されてきたものはわずかしかない。現在主流となっているパラダイムはオブジェクト指向設計であり、今日の複雑なプロジェクトのほとんどはオブジェクトを使い始めている。この流行にはさまざまな理由があり、オブジェクトが本来持っている性質によるものもあれば、環境的なものもある。さらに広く使われていること自体に由来する強みが要因となっているものもある。

なぜオブジェクトパラダイムが主流なのか？

チームがオブジェクトパラダイムを選ぶ理由の多くは、技術的なものではないし、オブ

ジェクトに本質的なものですらない。しかし、オブジェクトモデリングは、それが世に出た時から、シンプルでありながら洗練されているという絶妙なバランスを確かに保っている。

モデリングパラダイムがあまりにも難解な場合には、習得できる開発者の数が不足するために、間違った使い方をされてしまうだろう。チームにいる技術寄りでないメンバが、少なくともパラダイムの初歩を把握することができなければ、モデルを理解することもできず、ユビキタス言語が失われることになる。オブジェクト指向設計の基本は、たいていの人にとって自然なものに見えるようだ。開発者の中には、モデリングの機微を理解できない人もいるが、技術者でなくてもオブジェクトモデルの図を理解することはできる。

オブジェクトモデリングの概念はシンプルであっても、ドメインについての重要な知識をとらえられるくらい豊かなことは実証されている。さらに、ソフトウェアでモデルを表現できるようにした開発ツールによるサポートも、最初からあった。

今日、オブジェクトパラダイムには、成熟し広範囲で使われていることに起因する重要な利点もある。成熟したインフラストラクチャとツールのサポートがなければ、プロジェクトは脱線して技術的な研究開発となってしまい、スケジュールは遅れ、リソースはアプリケーション開発とは別の場所に割り当てられ、技術的なリスクを持ち込むことになりかねない。技術の中には、他の技術とうまく折り合わないものもあり、業界標準の解決法と統合することができず、チームが一般的なユーティリティを再発明しなければならなくなるかもしれない。しかし、年月と共に、これらの問題の多くが、オブジェクトに関しては解決されるか、広く採用されていることで問題とされなくなってきた（今や、主流となっているオブジェクトの技術と統合することは、その他のアプローチの方で行う必要が出てきている）。新しい技術のほとんどは、人気のあるオブジェクト指向プラットフォームと統合する手段を提供している。それによって、統合が容易になり、他のモデリングパラダイムに基づくサブシステムを混在させるという、選択の余地もできる（これについては、本章で後述する）。

同様に重要なのは、**開発者コミュニティと設計文化そのものの成熟である。**斬新なパラダイムを採用するプロジェクトでは、その技術を熟知している開発者や、選択したパラダイムで効果的なモデルを作成した経験のある開発者を見つけられないかもしれない。妥当な時間内で開発者を教育することも現実的ではないだろう。そのパラダイムと技術を最大限活用するためのパターンがまだ固まっていないからだ。おそらく、その分野の開拓者を迎えれば有効だが、その洞察は、まだ手に入るかたちで公表されてはいない。

オブジェクトは、何千もの開発者やプロジェクトマネージャ、その他プロジェクトの作業に携わる専門家によって形成されたコミュニティで、すでに十分理解されている。

ほんの10年前にあったオブジェクト指向プロジェクトの話は、成熟していないパラダイ

ムで仕事をすることの危険性を示している。1990年代初め、このプロジェクトはいくつかの最先端技術に自らを捧げることにした。この技術には、オブジェクト指向データベースの大規模な使用も含まれていた。これは刺激的なプロジェクトだった。これまでにこの技術がサポートした中で、最大のデータベースを配備する予定だと、チームメンバは誇らしげに訪問者に話したものである。私がプロジェクトに加わった時には、別々のチームが独自のオブジェクト指向設計を次々に考え出しては、オブジェクトを苦もなくデータベースに格納していた。しかし、徐々に現実がわかってきたのだが、我々はデータベースの容量の相当部分を使い果たしつつあるのだった。しかもテストデータで！ 実際のデータベースは数十倍も大きく、実際のトランザクションは数十倍になる想定だった。このアプリケーションでこの技術を使うのは不可能だったのか？ 使い方が適切でなかったのか？ 我々は深みにはまっていった。

幸運にも、この問題から我々を救い出すスキルを持った、世界でも一握りしかいない人の1人をチームに迎えることができた。先方の言い値で、彼を雇ったのである。問題には原因が3つあった。第一に、データベースと一緒に提供される既製のインフラストラクチャが、我々の要求に合わせて拡大できるものではなかった。第二に、細粒度のオブジェクトを格納するには、思っていたよりもはるかに多くのコストがかかることがわかった。第三に、オブジェクトモデルの一部にあった相互依存関係が非常にもつれていため、比較的少数の並列トランザクションでも競合が問題になった。

この雇われエキスパートの助けを借りて、我々はインフラストラクチャを強化した。その時には、チームは細粒度のオブジェクトがもたらす影響を意識して、この技術により適したモデルを探し始めた。全員がモデルにおける関係性のネットワークを制限する重要性についての考えを深め、その新しい理解に基づいて、緊密に相関した集約同士を、より低結合にするような、より優れたモデルを作り始めたのである。

失敗へ向かっていたその前の数か月に加え、この復旧にも数か月がかかった。しかも、このチームにとって、選択した技術が成熟していないことや、関連する技術の経験不足によって学習コストが発生したことに起因する手戻りは、これが最初ではなかった。残念ながら、このプロジェクトは結局縮小され、きわめて保守的になってしまった。今でも彼らは風変わりな技術を使用してはいるが、慎重にスコープを定めたアプリケーションに限っている。また、そのアプリケーションも、使用された技術の恩恵を受けることは、実際にはなさそうだ。

それから10年が経ち、オブジェクト指向の技術はかなり成熟した。一般的なインフラストラクチャのたいていの要求は、この分野で使用されてきた既製の解決策で実現することができる。必要不可欠なツールも主要なベンダから出ており、複数のベンダから出ていることもよくある。あるいは、信頼できるオープンソースプロジェクトからも入手すること

ができる。こうしたインフラストラクチャを構成するものの多くは、それ自体が十分に幅広く使用されているので、すでに理解している人々の基盤があり、解説書などもそろっている。こういう確立された技術の限界は、かなりよく理解されているから、博識なチームであれば背伸びしすぎることはないだろう。

興味深いモデリングパラダイムは他にもあるが、ここまで成熟してはいない。なかには習得が難しすぎて、数少ない専門以外では使用されることがないだろうと思えるものもある。一方、潜在的な可能性はあっても、技術的なインフラストラクチャが依然として寄せ集めであったり不安定であったりして、優れたモデルを巧妙に作成するすべを理解している人がほとんどいないものもある。こういうパラダイムもいずれ発達はするだろうが、ほとんどのプロジェクトにとっては、まだ使用する準備が整っていない。

そういうわけで、現在のところ、モデル駆動設計を試みるほとんどのプロジェクトは、賢明にも、オブジェクト指向技術をシステムのコアとして使用する。こういうプロジェクトがオブジェクトだけのシステムに閉じ込められることはないだろう。オブジェクトは業界の主流になっていて、統合ツールを入手すれば、最近使用されている技術の、ほとんどすべてと連結できるからだ。

それでも、永久にオブジェクトのパラダイムだけに限定しなければならないということではない。大勢で旅行すればいくぶん安全ではあるが、いつもそうすべきとは限らないということだ。オブジェクトモデルは、多数の実際に使われるソフトウェアに存在する問題に対応するが、ドメインによっては、ふるまいをカプセル化し、それを特定の単位に分割してモデル化することが不自然なものもある。例えば、高度に数学的なドメインや、システム全体に関わる論理的推論に支配されるドメインは、オブジェクト指向パラダイムにうまくなじまないのである。

オブジェクトの世界におけるオブジェクトではないもの

ドメインモデルはオブジェクトモデルでなくてもよい。例えば、Prologで実装され、そのモデルが論理ルールとファクトから構成されているモデル駆動設計もある。モデルパラダイムは、人がドメインについて考える際に好んで用いる、特定の方法に取り組むものだと考えられてきた。すると、そうしたドメインのモデルは、そのパラダイムによって形作られる。その結果できあがるモデルは、使用されたモデリングスタイルをサポートするツールを使って効率的に実装できるように、そのパラダイムに準拠したものとなる。

プロジェクトにおいて支配的なモデルパラダイムが何であれ、ドメインの中には、他のパラダイムならばはるかに容易に表現できる場所が必ずあるものだ。あるドメインに、ほんのわずか変則的な要素があるが、その他の点では、特定のパラダイムで問題なく機能するという場合、開発者は首尾一貫したモデルの中に若干扱いにくい対象があっても我慢す

ることができる（またはその正反対で、問題となるドメインの大部分が他の特定のパラダイムで自然に表現されるなら、パラダイムをまるごと切り替えて、別の実装プラットフォームを選択した方が理にかなっているかもしれない）。しかし、ドメインの主要部分が、さまざまな異なるパラダイムに属していると思われる場合、各部分を適したパラダイムでモデル化することは、知的好奇心を刺激する。この場合には、実装をサポートするツール類を混ぜて使用することになる。相互依存関係が少なければ、他のパラダイムで作られたサブシステムはカプセル化することができる。複雑な数学的計算であっても、単にオブジェクトから呼び出せばよいというようなものだ。あるいは、さまざまな側面がより絡み合っていることもある。例えば、オブジェクトの相互作用が、何らかの数学的関係で決まる場合が挙げられる。

こうしたことが、ビジネスルールエンジンやワークフローエンジンといった、オブジェクトでないコンポーネントを、オブジェクトシステムに統合する動機となる。パラダイムを混在させることで、開発者は特定の概念をモデル化するにあたり、最もよく合ったスタイルを選べるようになる。さらに、ほとんどのシステムは、オブジェクト的ではない技術的インフラストラクチャを使用せざるを得ない。その最も一般的な例が、関係データベースである。しかし、複数のパラダイムにまたがる一貫したモデルを作成することは困難で、サポートツールを共存させるのも単純にできることではない。ソフトウェアに埋め込まれる一貫したモデルを、開発者が明確に理解することができなければ、モデル駆動設計が完全に失われてしまうことになりかねない。このようにパラダイムが混在していることで、モデル駆動設計の必要性が増すにも関わらず、である。

パラダイムを混在させる際にはモデル駆動設計に忠実であること

ルールエンジンは、オブジェクト指向のアプリケーション開発プロジェクトに加えられる可能性のある技術の一例になる。知識豊富なドメインモデルは、明示的なルールを含んでいるだろうが、オブジェクトパラダイムには、ルールとその相互作用を表明するための特別なセマンティクスがない。ルールをオブジェクトとしてモデル化することができ、それが成功することもしばしばあるが、オブジェクトのカプセル化のせいで、システム全体に及ぶグローバルなルールは適用しにくくなる。ルールエンジンの技術が魅力的なのは、ルールを定義するためのより自然な宣言的方法を提供し、ルールパラダイムを効果的にオブジェクトパラダイムと混在できるようにするからだ。論理パラダイムは十分発達していて強力であり、オブジェクトの長所と短所をうまく補完すると考えられる。

しかし、期待するものが、常にルールエンジンから得られるとは限らない。製品の中には、とにかくうまく動かないものもある。2つの実装環境で実行されるモデル概念の関連性を、シームレスに表示させることができないものもある。その結果として、アプリケー

ションが2つに割れてしまうことがよくある。すなわち、オブジェクトを使用する静的データストレージシステムと、場当たり的なルール処理アプリケーションができるのだ。そして、ルール処理アプリケーションの方は、オブジェクトモデルとのつながりをほぼ完全に失ってしまう。

ルールに取り組みながらも、モデルの観点から考え続けるのは重要である。チームはどちらの実装パラダイムでも使える単一のモデルを見つけなければならない。これは容易でないが、ルールエンジンが表現力豊かに実装できるなら可能なはずだ。そうでなければ、データとルールの間につながりがなくなってしまう。さらに、エンジン内にあるルールは、ドメインモデルにおける概念的なルールというより、小さなプログラムに近くなってしまう。ルールとオブジェクトとの間の緊密で明確な関係があれば、ルールとオブジェクト両方の意味が保たれるのだ。

ルールとオブジェクトをシームレスにつなぐ環境がなければ、明確で本質的な概念で構成されるモデルを蒸留して、設計全体を1つにまとめられるかどうかは、開発者次第となる。

各場所を1つにまとめる上で最も効果的なツールは、さまざまなパラダイムによって構成されるモデル全体の根底に依存する、強固なユビキタス言語である。これはある。2つの環境で一貫した名前を適用し、その名前をユビキタス言語で使い続けることで、両者の溝を埋めるのに役立つ。

この話題は、これだけで一冊の本が書ける。本節が目標としているのは、モデル駆動設計を断念する必要はないことと、それを保つ苦労はする価値があるということを示すことだけである。

モデル駆動設計がオブジェクト指向である必要はないが、オブジェクトであれ、ルールであれ、ワークフローであれ、モデルの構成要素が表現力豊かに実装されることにかかっているのは確かだ。手に入るツールを使ってもその表現力が豊かにならないなら、ツールの選択を考え直した方がいい。表現力に乏しい実装は、つけ加えられたパラダイムの利点を無効にしてしまう。

オブジェクト指向が主流となっているシステムに、非オブジェクトの要素を混ぜ合わせるための経験則を4つ挙げよう。

- **実装パラダイムと対立しないこと。**ドメインに関する別の考え方は常にある。パラダイムに合うモデルの概念を見つけること。
- **ユビキタス言語に頼ること。**ツール間に厳格なつながりがない場合でも、言語を一貫して使用すれば、設計の各部分が分かれていってしまうことはない。
- **UMLにこだわらないこと。**UML図のようなツールに固執すると、容易に描けるもの

に合わせてモデルがゆがめられることがある。例えば、UMLには制約を表現する機能が確かにあるが、常にそれで十分とは限らない。別の作図スタイル（他のパラダイムでは一般的かもしれない）を選んだり、単純な自然言語で記述したりした方が、オブジェクトの特定の見方を示すことを意図された作図スタイルを、無理に適合させるよりはよい。

- **懐疑的であること。**ツールは本当に相応の働きをしているか？ ルールがあるからといって、ルールエンジンを使うオーバーヘッドが必要とは限らない。ルールは、多少わかりにくくはなるにしても、オブジェクトとして表現できるが、パラダイムを複数持つことで、事態はきわめて複雑になるのだ。

パラダイムを混在させるという負担を引き受ける前に、支配的なパラダイムの中にある選択肢を徹底的に検討すべきである。ドメインの概念の中に明らかなオブジェクトとして現れないものがあっても、そのパラダイムでモデル化できることがしばしばある。第9章では、オブジェクトの技術を使用した、型にはまらない概念のモデリングについて議論する。

関係データベースのパラダイムは、パラダイムの混在の中でも特殊な場合である。最も一般的な非オブジェクト技術である関係データベースは、他のコンポーネントと比べて、オブジェクトモデルと密接な関係にある。これはオブジェクトそのものを構成するデータの、永続ストレージとして機能するからである。オブジェクトデータを関係データベースへ格納することについては、オブジェクトのライフサイクルにおける他の多くの問題と一緒に、第6章で議論する。

第6章

ドメインオブジェクトのライフサイクル

すべてのオブジェクトにはライフサイクルがある。オブジェクトは誕生した後、おそらくさまざまな状態を経て、最終的には死ぬ。死ぬと言っても、アーカイブされるか削除されるということだ。もちろん、こうしたオブジェクトの多くは単純な一過性のもので、コンストラクタを気軽に呼び出すことで生成され、何らかの処理に使用された後で、ガベージコレクタに引き渡される。そういうオブジェクトを複雑にする必要はない。しかし、そ

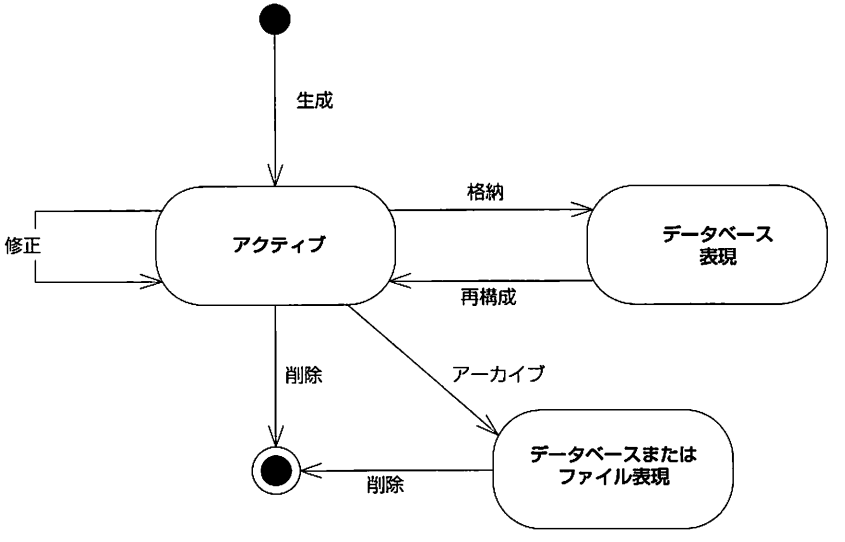


図6.1　ドメインオブジェクトのライフサイクル

れより寿命の長いオブジェクトもあり、メモリ上に存在するだけとは限らない。そうしたオブジェクトには、他のオブジェクトとの間に複雑な相互依存関係がある。不変条件が適用されるような、状態の変化も経験する。こうしたオブジェクトを管理することに伴う課題は、モデル駆動設計に向けた挑戦を容易に脱線させかねない。

この課題は2つのカテゴリに分けられる。

1. ライフサイクルを通じて整合性を維持すること。
2. ライフサイクルを管理するのが複雑でも、モデルが浸食されないようにすること。

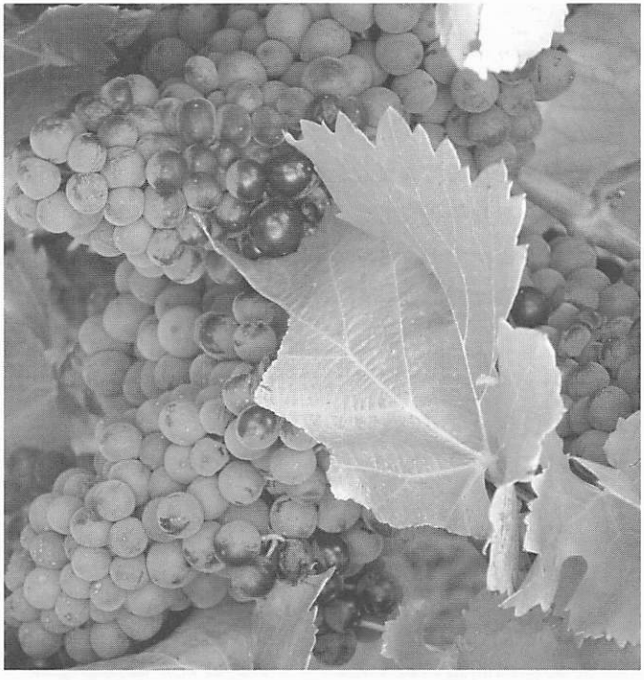
本章では、3つのパターンを通じて、これらの問題に対応する。まず集約が、明確な所有権と境界を定義することで、モデルそれ自体を引き締める。こうすることで、オブジェクトのネットワークが混沌としてもつれたものになるのを避けられる。このパターンは、ライフサイクルのすべてのフェーズにおいて整合性を保つ上で、決定的に重要である。

次に、焦点をライフサイクルの始まりに合わせ、ファクトリを使用して、複雑なオブジェクトや集約を生成したり再構成したりする。そうすることで、内部構造はカプセル化されたままとなる。最後に、リポジトリがライフサイクルの中期と終わりに対応して、永続化されたオブジェクトにアクセスする手段を提供しつつ、アクセスに伴って必要となる巨大なインフラストラクチャをカプセル化する。

リポジトリとファクトリは、それ自体はドメインに由来しないが、ドメイン設計においては意味のある役割を持っている。これらの構成概念は、モデルオブジェクトを操作できるようにすることで、モデル駆動設計を完成させるものなのだ。

集約をモデリングし、ファクトリとリポジトリを設計へ追加することで、モデルオブジェクトの操作を体系的に、それもライフサイクルを通じて意味のある単位で行えるようになる。集約が区切るスコープによって、不変条件が維持されなければならない範囲が示される。これは、ライフサイクルのあらゆる段階に当てはまる。ファクトリとリポジトリは集約を対象としている。ライフサイクルにおける特定の変遷は複雑さを伴うが、ファクトリとリポジトリによってその複雑さがカプセル化される。

集約（AGGREGATES）



関連を最小限に抑えるように設計することにより、関連を辿る処理は単純化され、関係性の爆発的増加もある程度は制限される。しかし、ほとんどのビジネスドメインは非常に強く相互に結びついているので、結局はオブジェクトの参照を通じて、長くて深い経路を辿ることになる。ある意味で、こうしたもつれはこの世界の現実を反映している。現実には、はっきりした境界が引いてもらえることはめったにないのだ。これはソフトウェアの設計における問題である。

データベースから人オブジェクトを削除しているとしよう。その人と一緒に名前や生年月日、職務経歴書はなくなる。しかし、住所はどうだろうか？ 同じ住所に別の人がいるかもしれない。住所を削除すると、そうした別の人オブジェクトは、削除されたオブジェクトへの参照を持つことになる。逆に、放置すれば、データベースに不要な住所をためこんでしまう。自動ガベージコレクションを使えば、不要な住所を除去できるだろうが、そうした技術的解決は、データベースシステムを用いて実現できたとしても、基本的なモデリングの問題を無視している。

隔離されたトランザクションについて検討する時でさえ、典型的なオブジェクトモデル

が関係性のネットワークを持っているせいで、変更による潜在的な影響の範囲は明確にはわからない。依存関係があった場合に備えて、システム内のオブジェクトをすべてリフレッシュするのは現実的ではない。

この問題が深刻化するのは、複数クライアントによる同一オブジェクトへの同時アクセスがあるシステムにおいてである。多くのユーザが、システム内でいろいろなオブジェクトを参照して更新すると、相互依存するオブジェクトへの同時変更を防がなければならない。スコープの決め方を誤ると、深刻な結果につながる。

複雑な関連を伴うモデルでは、オブジェクトに対する変更の一貫性を保証するのは難しい。維持すべき不変条件には、個々のオブジェクトに適用されるものだけでなく、密接に関連するオブジェクトのグループに適用されるものもある。だが、慎重にロックしすぎると、今度は複数のユーザが指針もなく相互に干渉し合い、システムが使いものにならなくなる。

別の言い方をすると、他の複数のオブジェクトから構成されているオブジェクトが、いつ作られ、いつなくなるのかは、どうすればわかるのか？ データの永続ストレージを持つシステムには、データを変更するトランザクションのためのスコープと、データの一貫性を維持する（つまり、不変条件を維持する）方法がなければならない。データベースでは多様なロック機構が使用でき、テストをプログラミングすることもできる。しかし、こうしたその場しのぎの解決策は、注意をモデルからそらすことになり、間に合わせのプログラムを書いては祈るという日々へと、すぐに逆戻りしてしまう。

実際、この種の問題に対するバランスの取れた解決策を見つけるには、ドメインに対するより深い理解が必要である。しかもこの場合は、その範囲が、クラスのインスタンス間に見られる変更頻度の違いといった要素にまで広がる。競合が集中する場所に対するロックは緩やかにし、厳格な不変条件についてはより厳しく保証するモデルを探す必要がある。

この問題は、データベーストランザクションにおける技術的な難問であるように見えるが、その根源はモデルにある。つまり、境界が定義されていないことが問題なのだ。解決策をモデルから導き出すことによって、モデルは理解しやすくなり、設計は伝達しやすくなる。モデルが改訂されると、その改訂によって実装を変更するよう導かれるのだ。

モデル内に存在する関係性について、所有権を定義するための構想が考えられてきた。次に述べるのは、単純だが厳格なシステムである。このシステムは、これまで説明した概念から蒸留されたもので、そこに含まれるルールの集合は、これらのオブジェクトとその所有者を変更するトランザクションを実装するためのものである[1]。

1 1990年代にDavid Siegelがこのシステムをプロジェクトで考案して使用したが、正式に発表してはいない。

まず、モデル内にある参照をカプセル化するための抽象化が必要である。集約とは、関連するオブジェクトの集まりであり、データを変更するための単位として扱われる。各集約にはルートと境界がある。境界は集約の内部に何があるかを定義するものだ。ルートは集約に含まれている特定の1エンティティである。集約のメンバの中で、外部のオブジェクトが参照を保持してよいのはルートだけである。ただし、境界内のオブジェクトは互いに参照を保持し合ってもよい。ルート以外のエンティティは局所的な同一性を持っているが、その同一性は集約内部でのみ識別できればよい。外部のオブジェクトは、ルートエンティティのコンテキストを外れると、それらのエンティティを見ることさえできないからだ。

自動車修理店向けのソフトウェアでは、自動車のモデルが使われることがあるだろう。自動車はグローバルな同一性を持つエンティティである。その自動車は、酷似しているものも含め、世界中にある他のどんな自動車とも区別したいからだ。それには車両登録番号を使用すればよい。この番号は、新車それぞれに割り当てられる一意の識別子だ。4つの車輪に対して、タイヤのローテーションの記録を追跡したくなるかもしれないし、各タイヤの走行距離と接地面の磨耗を知りたくなるかもしれない。どのタイヤがどれかを知るには、タイヤも識別されたエンティティでなければならない。しかし、その特定の自動車というコンテキストの外部で、それらのタイヤの同一性が気にされるとは思えない。タイヤを交換して古いタイヤをリサイクル工場へ送れば、ソフトウェアはそのタイヤの追跡を一切止めるか、積み上げられたタイヤの山を構成する無名のメンバとなるかのいずれかである。ローテーションの記録など誰も気にしない。さらに重要なことだが、自動車に装着されている間も、特定のタイヤを見つけるためにシステムに問い合わせてから、それがどの自動車に装着されているかを調べる人はいない。自動車を見つけるためにデータベースに問い合わせ、その上でタイヤへの一時的な参照を要求する。したがって、自動車が集約のルートエンティティとなり、集約の境界がタイヤも囲い込むことになる。一方、エンジンブロックにはシリアル番号が刻印されていて、自動車とは独立して追跡されることがある。アプリケーションによっては、エンジンが集約を持ち、同時にそのルートになるだろう。

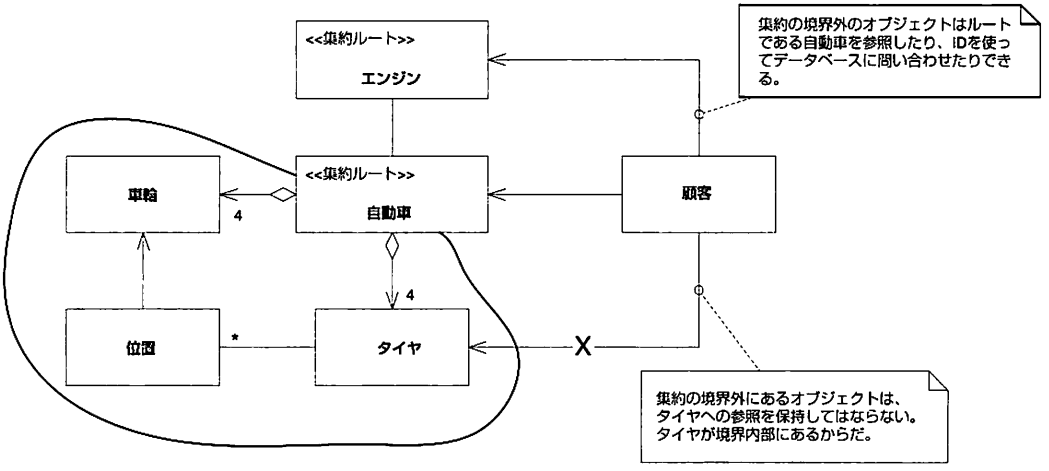


図6.2 ローカルの同一性対グローバルの同一性とオブジェクト参照

不変条件とは、データが変更される時は常に維持されなければならない一貫性のルールで、集約のメンバ間の関係も含んでいる。複数の集約にまたがるルールはどれも、常に最新の状態にあるということが期待できない。イベント処理やバッチ処理、その他の更新の

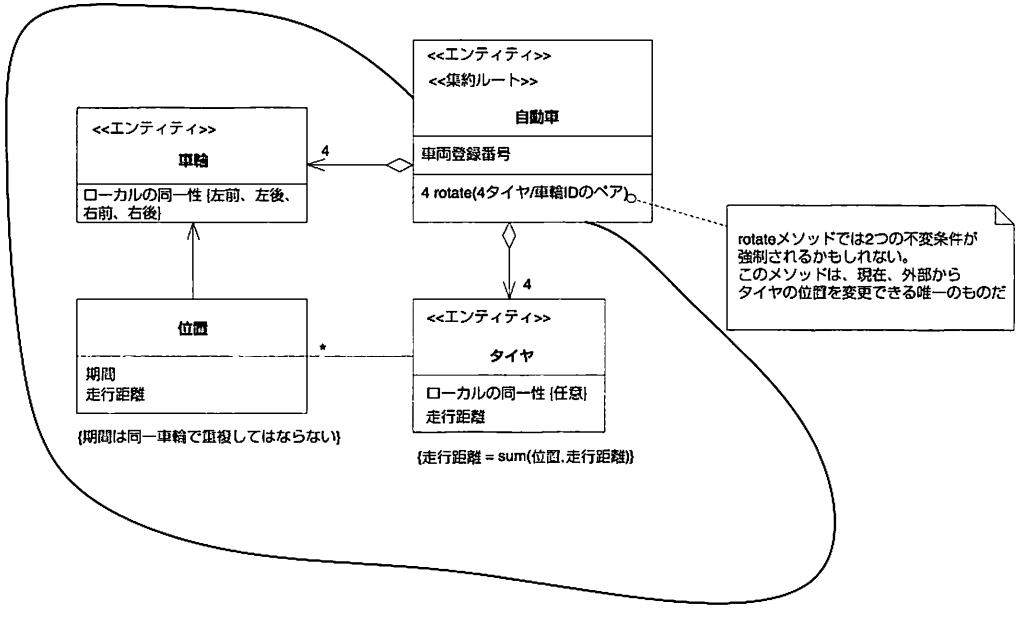


図6.3 集約の不変条件

仕組みを通じて、他の依存関係は一定の時間内に解消できる。しかし、1つの集約内で適用される不変条件は、各トランザクションの完了によって強制される。

さて、この概念的な集約を実装するためには、すべてのトランザクションに適用できるルールが必要になる。

- 集約のルートエンティティは、グローバルな同一性を持ち、不変条件をチェックする最終的な責務を負う。
- ルートエンティティはグローバルな同一性を持つ。境界内部のエンティティは、集約内でのみ一意となるローカルな同一性を持つ。
- 集約の境界外にあるオブジェクトは、ルートエンティティを除き、境界内部への参照を保持することができない。ルートエンティティは内部のエンティティへの参照を他のオブジェクトに渡せるが、受け取ったオブジェクトは参照を一時的に使用することができるだけで、その参照を保持してはならない。ルートは値オブジェクトのコピーを別のオブジェクトに渡すこともあるが、その場合はそのコピーがどうなってもよい。渡されるのは単なる値で、集約とは何の関連もなくなるからである。
- 前述のルールの必然的結果として、データベースに問い合わせて直接取得できるのは、集約ルートだけとなる。それ以外のオブジェクトはすべて、関連を辿ることで取得しなければならない。
- 集約内部のオブジェクトは、他の集約ルートへの参照を保持することができる。
- 削除の操作は、集約境界の内部に存在するあらゆるものを一度に削除しなければならない（ガベージコレクションがあれば簡単である。ルート以外に対しては、外部から参照されていないので、ルートを削除した際に、他もすべてガベージコレクションすればよい）。
- 集約境界の内部に存在するオブジェクトに対する変更がコミットされる時には、集約全体の不変条件がすべて満たされていなければならない。

エンティティと値オブジェクトを集約の中にまとめ、各集約の周囲に境界を定義すること。各集約に対してルートとなるエンティティを1つ選び、境界の内部に存在するオブジェクトへのアクセスは、そのルートを経由して制御すること。外部のオブジェクトが参照を保持できるのは、ルートのみとすること。内部のメンバに対する一時的な参照を渡してよいのは、単一の操作で使用する時だけだ。ルートがアクセスを制御するので、内部が知らないうちに変更されることはなくなる。この取り決めにより、どんな状態変化においても、集約内にあるオブジェクトと集約全体に対して、不変条件をすべて強制することが現実的になる。

技術的なフレームワークを用いて集約が宣言でき、さらにロック機構などが自動的に実行されると、非常に便利だ。そういう助けがなければ、集約に関して合意し、それと矛盾しないようにコーディングする自己規律がチームに必要となる。

例6.1——購入注文の整合性

単純化された購入注文システムで起こり得る複雑な事態について考察しよう。

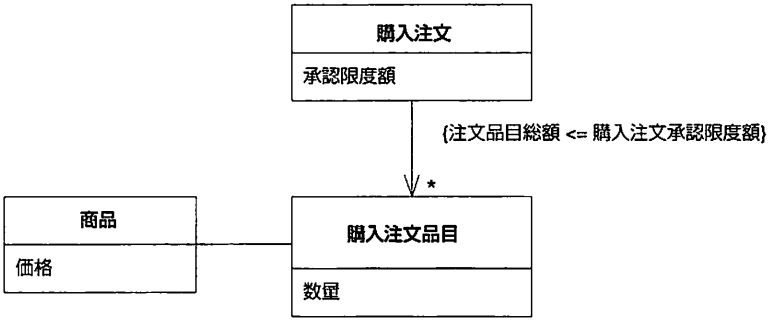


図6.4 購入注文システムのモデル

この図は、品目名ごとに分けた購入注文のきわめて慣例的な見方を示していて、品目の合計額が購入注文承認限度額を超えてはならないという不変条件のルールがついている。既存の実装には、相互に関連した問題が3つある。

1. **不変条件の強制。**新しい品目が追加されると、購入注文は合計額をチェックし、限度額を超えた場合は無効の印を付ける。この後見ていくように、これでは十分に保護できていない。
2. **変更管理。**購入注文が削除されたりアーカイブされたりすると、品目も同じように扱われるが、どこまで関係をたどるべきかについて、モデルは何の指示も出していない。別のタイミングで商品の価格を変更した際の影響についても混乱がある。
3. **データベースの共有。**複数ユーザが使用すると、データベースで競合の問題が起こる。

複数のユーザはさまざまな購入注文を並行して入力して更新するので、互いの作業を台無しにするのを防がなければならない。まずはきわめて単純な戦略からはじめよう。これは、あるユーザが編集し始めたら、トランザクションがコミットされるまで、あらゆるオブジェクトをロックするというものだ。そうすると、ジョージが品目001を編集している間、アマンダはそれにアクセスすることができない。ただし、アマンダは他のあらゆる購入注文で、どの品目でも（ジョージが作業中の購入注文にある別の品目も含めて）編集できる。

購入注文 #0012946 承認限度額： $1,000.00

品目#	数量	要素	価格	総額
001	3	ギター	@ 100.00	300.00
002	2	トロンボーン	@ 200.00	400.00
			合計：	700.00

図6.5 データベースに格納されている購入注文の初期状態

オブジェクトはデータベースから読み込まれ、ユーザごとに割り当てられたメモリ空間にインスタンス化される。そのメモリ上で表示と編集が行われるが、データベースのロックは編集開始時にのみ要求される。したがって、お互いの品目に干渉しない限り、ジョージもアマンダも並行して作業できる。すべてはうまくいく…ただし、ジョージとアマンダが2人とも同じ購入注文にある別々の品目について作業するまでの話だ。

ジョージが自分のビューでギターを追加する

購入注文 #0012946 承認限度額：$1,000.00

品目#	数量	要素	価格	総額
001	**5**	ギター	@ 100.00	**500.00**
002	2	トロンボーン	@ 200.00	400.00
			合計：	900.00

アマンダが自分のビューでトロンボーンを追加する

購入注文 #0012946 承認限度額：$1,000.00

品目#	数量	要素	価格	総額
001	3	ギター	@ 100.00	300.00
002	**3**	トロンボーン	@ 200.00	**600.00**
			合計：	900.00

図6.6 別個のトランザクションにおける同時編集

2人のユーザとソフトウェアにとってすべてが順調に見えるが、これはトランザクション中に発生する、データベースの他の箇所への変更を無視しているからだ。ロックされた品目はどちらも、もう片方のユーザによる変更には関わっていないのだ。

購入注文 #0012946 承認限度額：**$1,000.00**

品目#	数量	要素	価格	総額
001	5	ギター	@ 100.00	500.00
002	3	トロンボーン	@ 200.00	600.00
			合計：	**1,100.00**

図6.7 結果としてできる購入注文は承認限度額に違反している（不変条件違反）

両方のユーザが変更を保存した後で、ドメインモデルの不変条件に違反している購入注文がデータベースに格納される。重要なビジネスルールが破られ、しかも、誰も気づいてさえいないのだ。

明らかに、単一の品目をロックするだけでは保護手段として十分ではない。代わりに、購入注文全体を一度にロックしていたら、この問題は防げただろう。

ジョージが自分のビューを編集する

購入注文 #0012946 承認限度額：$1,000.00

品目#	数量	要素	価格	総額
001	5	ギター	@ 100.00	**500.00**
002	2	トロンボーン	@ 200.00	400.00
			合計：	900.00

アマンダは購入注文 #0012946から締め出される

ジョージの変更がコミットされた

アマンダがアクセスすると、ジョージの変更した内容が表示される

購入注文 #0012946 承認限度額：$1,000.00

品目#	数量	要素	価格	総額
001	5	ギター	@ 100.00	500.00
002	3	トロンボーン	@ 200.00	**600.00**
		限度額超過→	合計：	1,100.00

図6.8 購入注文全体をロックすると、不変条件を強制できるようになる

アマンダが限度額を引き上げたり、ギターを削除したりしてこの問題を解決するまで、プログラムはこのトランザクションが保存されることを認めない。この仕組みを使えば競合の問題は防げるし、作業が多数の購入注文にわたって広く分散していることが多いのであれば、よい解決策かもしれない。しかし、多くの人々が、大きな購入注文にあるいろいろな品目に対して同時に作業することが普通である場合、このようなロックでは面倒なことになる。

小さな購入注文が多数あると想定しても、この表明に違反する方法は他にもある。「商品」で考えよう。アマンダが注文を追加している間に、誰かがトロンボーンの価格を変更したら、それも不変条件に違反することになるのではないだろうか？

購入注文全体に加え、商品もロックしてみよう。ジョージとアマンダとサムが別々の購入注文で作業するとどうなるかを以下に示す。

ジョージが購入注文を編集
ギターとトロンボーンがロックされる

購入注文 #0012946 承認限度額：$1,000.00

品目#	数量	要素	価格	総額
001	2	ギター	@ 100.00	**200.00**
002	2	トロンボーン	@ 200.00	400.00
			合計：	600.00

アマンダがトロンボーンを追加するが、ジョージの作業が終わるのを待たなければならない
バイオリンがロックされる

購入注文 #0012932 承認限度額：$1,850.00

品目#	数量	要素	価格	総額
001	3	バイオリン	@ 400.00	1,200.00
002	**2**	**トロンボーン**	**@ 200.00**	**400.00**
			合計：	1,600.00

サムがトロンボーンを追加するが、ジョージの作業が終わるのを待たなければならない

購入注文 #0013003 承認限度額：$15,000.00

品目#	数量	要素	価格	総額
001	1	ピアノ	@1,000.00	1,000.00
002	**2**	**トロンボーン**	**@ 200.00**	**400.00**
			合計：	1,400.00

図6.9 慎重すぎるロックは作業を妨げる

さらに不便になっているが、それは楽器（「商品」）の競合が多くあるからである。そして、次のようになる。

ジョージがバイオリンを追加、アマンダを待たなければならない（！）

購入注文 #0012946 承認限度額：$1,000.00				
品目#	数量	要素	価格	総額
001	2	ギター	@ 100.00	200.00
002	2	トロンボーン	@ 200.00	400.00
003	1	バイオリン	@ 400.00	400.00
			合計：	1,000.00

図6.10 デッドロック

3人ともしばらく待ち続けることになる。

この時点で、以下に挙げるビジネスの知識を組み込むことにより、モデルを改良し始めることができる。

1. 商品は多くの購入注文で使用される（高競合）。
2. 商品に対する変更は購入注文に対する変更より少ない。
3. 商品の価格に対する変更は、必ずしも既存の購入注文に伝える必要はない。これは購入注文の状態に対して、価格がいつ変更されたかによる。

ポイント3は、すでに納品済みでアーカイブされた購入注文を考えると、特に明らかである。表示するべきなのは、もちろん現在の価格ではなく、入力された時の価格である。

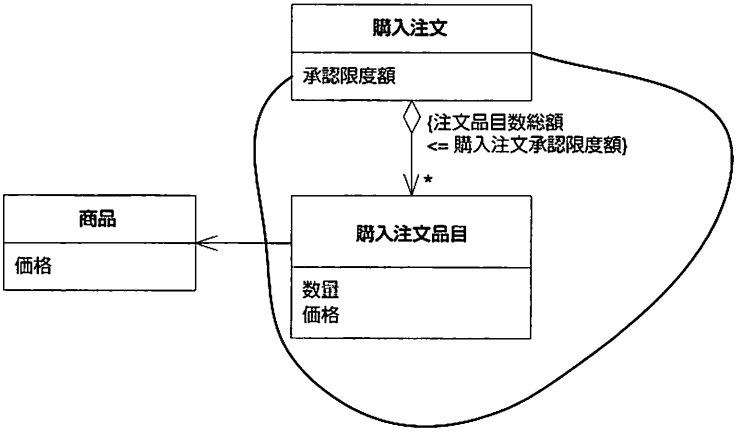


図6.11 価格が品目へコピーされる。これで**集約**の不変条件を強制できる

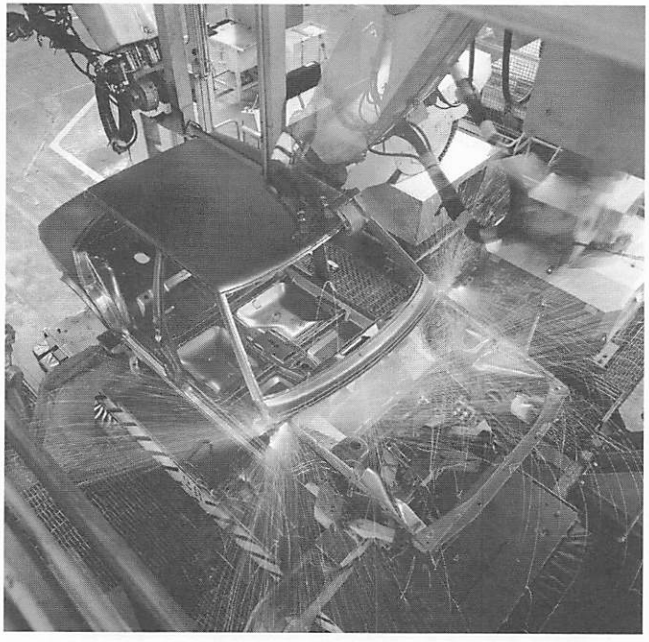
このモデルと一致した実装をすれば、購入注文とその品目に関連する不変条件は保証されるだろう。一方で、商品の価格変更は、それを参照する品目にすぐに反映する必要はない。より広範な一貫性のルールには、別の方法で対応できるだろう。例えば、価格の古くなった品目の一覧をシステムが毎日ユーザに示すことで、ユーザはそれらを更新するか、削除できるようになるだろう。ただし、これは常に強制する必要のある不変条件ではない。商品に対する品目の依存関係を緩めることで、我々は競合を回避し、ビジネスの現実をより的確に反映するのだ。同時に、購入注文と品目の関係を緊密にして、重要なビジネスルールが守られることを保証する。

集約により、購入注文とその品目に対して、ビジネスプラクティスと一致した所有権が与えられる。購入注文と品目の生成と削除が自然と結びつけられるのに対し、商品の生成と削除は独立しているのだ。

＊　＊　＊

集約によって区切られるスコープによって、不変条件が維持されるべき範囲が示される。次に取り上げるパターンのファクトリとリポジトリは集約を対象としている。ライフサイクルにおける特定の変遷は複雑さを伴うが、ファクトリとリポジトリによってその複雑さがカプセル化される…。

ファクトリ（FACTORIES）



オブジェクトや集約全体を生成するのが複雑になったり、内部構造をさらけ出しすぎていたりする場合、ファクトリによってカプセル化が実現される。

＊　＊　＊

オブジェクトが持つ能力の多くは、オブジェクトの内部と他のオブジェクトとの関連の複雑な構成の中に眠っている。オブジェクトは蒸留して、その意味に関係しないものや相互作用において果たす役割をサポートしないものが一切残らないようにしなければならない。こうしたライフサイクル中期における責務は大量にある。複雑なオブジェクトに、それ自体を生成する責務を負わせると、問題が生じる。

自動車エンジンは複雑な機械で、その責務を実行するために数十の部品が協調している。エンジンの責務は、シャフトを回転させることだ。エンジンブロックを設計することを想像してみよう。エンジンブロックはピストン一式をつかんでシリンダに挿入し、点火プラグを発火させる。このプラグは専用のソケットにねじ込まれている。しかし、ここまで入り組んだ機械を自分で組み立てたとしたら、典型的なエンジン同様に信頼できたり、効率的だったりするとは思えない。代わりに、何か他のものに部品を組み立ててもらうことに

する。おそらく、人間の機械工か産業用ロボットだろう。ロボットも人間も、実際は組み立てるエンジンより複雑である。さらに、部品を組み立てる仕事は、シャフトを回転させる仕事とは全く関係ない。組み立てを行うのは、自動車を生産している間だけである。つまり、自動車を運転している時には、ロボットや機械工は必要ない。自動車の組み立てと運転が同時に行われることはないから、これらの機能を同じ仕組みの中に組み合わせることに価値はない。同じように、複雑な複合オブジェクトの組み立ても、そのオブジェクトが完成した時に行うことになる仕事からは、分離しておくのが一番である。

しかし、責務を別の当事者であるアプリケーションのクライアントオブジェクトへ移すと、問題はさらに悪化する。クライアントが知っているのは、どのような仕事が実行されなければならないかであり、必要な処理を実行するのはドメインオブジェクトに任せる。必要なドメインオブジェクトを組み立てることをクライアントに期待したら、そのオブジェクトの内部構造について、クライアントがある程度知らなければならなくなる。ドメインオブジェクトにおける各部分の関係性について、そこに適用される不変条件をすべて強制するためには、クライアントは生成されるオブジェクトのルールについても知らなければならない。コンストラクタの呼び出しでさえ、構築しているオブジェクトの具象クラスにクライアントを結びつけてしまう。クライアントを変更しないと、ドメインオブジェクトの実装を変更することができず、リファクタリングは困難になる。

オブジェクトの生成を担うクライアントは不必要に複雑化して、本来の責務が不明瞭になる。生成されるドメインオブジェクトと集約のカプセル化にも違反している。さらに悪いことに、クライアントがアプリケーション層の一部だとすると、責務がドメイン層から漏れ出してしまったことになる。このようにアプリケーションが実装の詳細に対して緊密に結びついていると、ドメイン層における抽象化によって得られる利点がほとんど奪い去られ、継続的な変更が一層高くつくことになる。

オブジェクトの生成は、それ自体が主要な操作になり得るが、複雑な組み立て操作は、生成されるオブジェクトの責務には合わない。そういう責務を組み合わせてしまうと、理解しにくく不格好な設計が作り出されるかもしれない。そうかといって、クライアントに直接構築させると、クライアントの設計を混乱させ、組み立てられるオブジェクトや集約のカプセル化にも違反する上に、生成されるオブジェクトの実装とクライアントが過度に結合することになる。

複雑なオブジェクトの生成はドメイン層の責務であるが、その仕事はモデルを表現するオブジェクトのものではない。オブジェクトの生成と組み立てが、「銀行口座開設」のように、ドメインにおける重要なマイルストーンと一致する場合もある。しかし、通常は、オブジェクトの生成と組み立ては、ドメインにおいて意味を持たない。実装のために必要なのである。この問題を解決するため、エンティティ、値オブジェクト、サービスのどれとも

違う構成要素をドメインの設計に追加しなければならない。これは前章の議論とは離れてしまうので、要点を明確にすることが重要だ。すなわち、これから設計に追加する要素は、モデル中のどれにも対応していないが、それでもドメイン層の責務の一部なのである。

オブジェクト指向言語はすべて、オブジェクトを生成する仕組みを提供しているが（例えば、JavaとC++のコンストラクタやSmalltalkのインスタンス生成クラスメソッドなど）、他のオブジェクトから分離した、より抽象的なオブジェクト構築の仕組みが必要なのだ。他のオブジェクトの生成を責務とするプログラム要素は、ファクトリと呼ばれる。

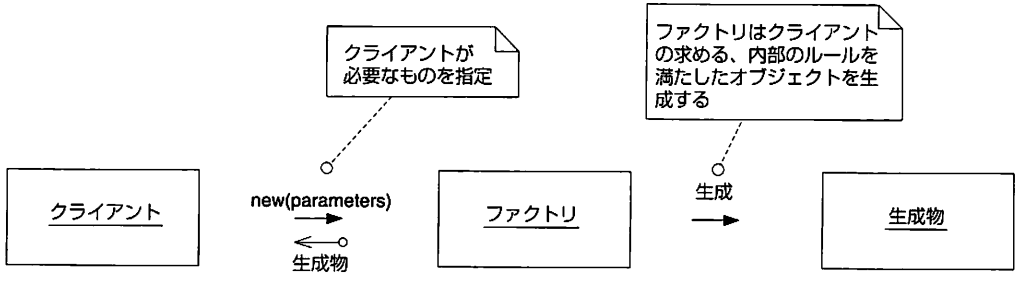


図6.12 **ファクトリ**との基本的な相互作用

オブジェクトのインタフェースが実装をカプセル化し、それによってクライアントが、どう動くのかを知らずにオブジェクトのふるまいを使えるようになるのと同じく、ファクトリも、複雑なオブジェクトや集約を生成するのに必要な知識をカプセル化する。ファクトリが提供するインタフェースは、クライアントにとっての目的や、生成されるオブジェクトに対する抽象的な見方を反映したものになる。

それゆえ：

複雑なオブジェクトと集約のインスタンスを生成する責務を、別のオブジェクトに移すこと。その別のオブジェクトは、それ自体ではドメインモデルにおいて何の責務も負っていないかもしれないが、それでもドメイン設計の一部であることに変わりはない。複雑な組み立てをすべてカプセル化するインタフェースを提供すること。その際に、インスタンス化されるオブジェクトの具象クラスを、クライアントが参照しなくてもよいようにすること。集約全体をひとまとまりとして生成し、その不変条件を強制すること。

＊ ＊ ＊

ファクトリの設計方法は数多くある。ファクトリメソッド（FACTORY METHOD）、抽象ファクトリ（ABSTRACT FACTORY）、ビルダ（BUILDER）といった特別な目的を持つ生成パターンは、『デザインパターン』（Gamma他 1995）において詳述されている。『デザイ

ンパターン』は主に、オブジェクト構築にある問題のうちで、最も難解なもののためのパターンを探究している。だが、ここで重要なのは、ファクトリの設計を掘り下げることではなく、ドメイン設計における重要なコンポーネントとして、ファクトリの居場所を示すことである。ファクトリを適切に使用すれば、モデル駆動設計を順調に進められる。

優れたファクトリには、共通する基本的な要件が2つある。

1. 生成メソッドはそれぞれアトミックであり、生成するオブジェクトや集約の不変条件をすべて強制する。ファクトリが生成してよいのは、一貫した状態のオブジェクトだけだ。エンティティであれば、それは不変条件をすべて満たして集約全体を生成することを意味するが、必須ではない要素が追加される可能性はある。不変の値オブジェクトであれば、すべての属性が正しい最終状態に初期化されることを意味する。もしインタフェースが、正しく生成できないオブジェクトを要求できるものであるなら、例外を発生させるか、別の仕組みを起動して不適切な戻り値があり得ないことを保証する必要がある。
2. ファクトリは、生成される具象クラスではなく、要求される型に応じて抽象化しなければならない。これには、『デザインパターン』（Gamma他 1995）で紹介されている、洗練されたファクトリパターンが役に立つ。

ファクトリとその場所を選択する

一般に、ファクトリを生成するのは、詳細を隠したいと思うものを構築するためであり、配置するのは制御を行わせたい場所である。こうした決定は、通常、集約を中心として行われる。

例えば、既存の集約に要素を追加する必要があれば、その集約のルートにファクトリメソッドを生成することもできる。これにより、集約内部の実装が外部のクライアントから隠蔽される一方で、要素が追加された際に集約の整合性を保証する責務がルートに与えられる。これについて、図6.13に示す。

別の例として、他のオブジェクトの生成に密接に関わるオブジェクトにファクトリメソッドを置くというものもある。ただ、一旦生成した後は、そのオブジェクトが生成物を所有するわけではない。あるオブジェクトの持つデータや、場合によってはルールが、別のオブジェクトを生成する上で非常に支配的である場合、そのオブジェクトにファクトリメソッドを置くことによって、別のオブジェクトを生成するために情報をそのオブジェクトから引き出して任意の場所で使用できるようにする必要がなくなる。また、こうすることで、生成するものと生成されるものの間に特別な関係があることも伝わるようになる。

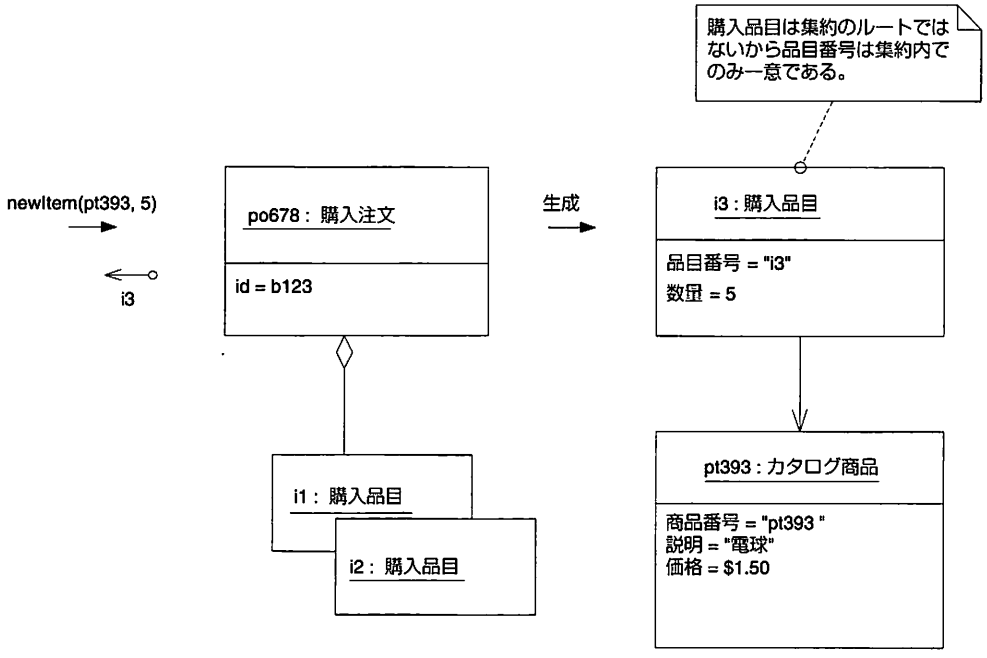


図6.13 **ファクトリメソッド**は**集約**の拡張をカプセル化する

図6.14で、**取引注文**（TradeOrder）は**証券取引口座**（BrokerageAccount）と同じ集約の一部ではない。第一に、**取引注文**は取引実行アプリケーションとやり取りすることになるが、そこでは**証券取引口座**は邪魔にしかならないからである。それでも、**証券取引口座**に**取引注文**の生成を制御させるのは、自然なことのように見える。**証券取引口座**には、（独自の同一性を持って生成される）**取引注文**に組み込まれることになる情報もあり、その情報の中にはどんな取引が認められるかを定めるルールも含まれているからだ。**取引注文**の実装を隠蔽することによる恩恵も受けられるだろう。例えば、実装が**買い注文**と**売り注文**という別個のサブクラスを持つ階層にリファクタリングされるかもしれない。ファクトリによって、クライアントは具象クラスと結合しないでいられる。

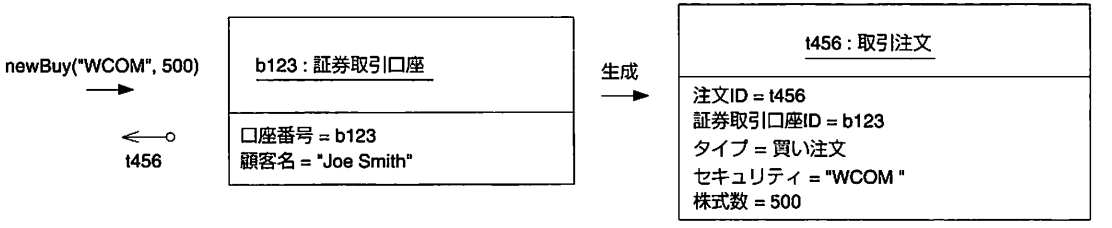


図6.14 **ファクトリメソッド**により、同じ**集約**の一部ではない**エンティティ**が生成される

ファクトリは生成物と非常に緊密に結合するので、ファクトリを所属させるのは、その生成物に対して密接で自然な関係を持つオブジェクトだけにするべきである。具体的な実装であれ、構築に関する純粋な複雑さであれ、何か隠蔽したいものがあるのに、自然な置き場所がなさそうな場合、専用のファクトリオブジェクトかサービスを作成しなければならない。独立したファクトリは、通常、集約全体を生成してルートへの参照を渡し、生成された集約の不変条件が必ず強制されることを保証する。集約の内部にあるオブジェクトがファクトリを必要とし、集約ルートがそのファクトリの妥当な置き場所でない場合は、ためらわずに独立したファクトリを作成すること。ただし、アクセスを集約の内部に制限するルールを尊重し、集約の外部からは、生成物を一時的にしか参照しないようにすること。

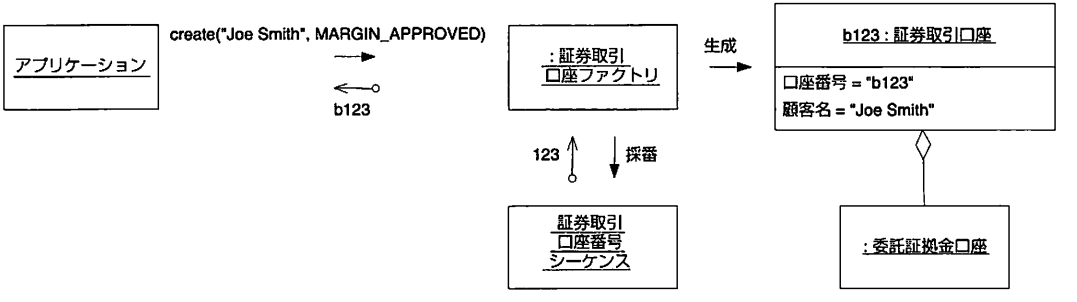


図6.15 独立した**ファクトリ**が**集約**を構築する

コンストラクタがあればよい場合

これまで私が飽きるほど目にしてきたコードは、**すべての**インスタンスを生成するにあたって、クラスコンストラクタやそのプログラミング言語にとってプリミティブなレベルのインスタンス生成手段を直接呼び出していた。ファクトリを導入することには大きな利点があるが、一般には十分に活用されていない。ただ、コンストラクタは、直接的である

という理由により、最善の選択肢になることがある。ファクトリを使うと、実は、多態性を使用しない単純なオブジェクトが不明瞭になりかねない。

次のような状況におけるトレードオフは、むき出しの公開コンストラクタに有利に働く。

- クラスが型（type）である。つまり、興味深い階層の一部ではなく、インタフェースを実装することで多態的に使用されることもない。
- クライアントが実装に関心がある。おそらくは、ストラテジーを選択する方法として考えているのだろう。
- オブジェクトの持つ属性をすべてクライアントが取得できるため、クライアントが目にするコンストラクタの内部に、さらに別のオブジェクトの生成がネストされることがない。
- 構築が複雑でない。
- 公開コンストラクタは、ファクトリと同じルールに従わなければならない。すなわち、生成されるオブジェクトの不変条件をすべて満たすアトミックな操作でなければならない。

コンストラクタ内で他のクラスのコンストラクタを呼び出すのは避けること。コンストラクタはいたって単純でなければならない。複雑な組み立てに関して、特に集約を組み立てる場合にはファクトリが必要となる。ちょっとしたファクトリメソッドなら、使う際の敷居は高くない。

Javaのクラスライブラリが興味深い例を提供している。コレクションはすべて、具体的な実装からクライアントを分離するインタフェースを実装している。それにも関わらず、コレクションはすべてコンストラクタを直接呼び出すことで生成されるのである。ファクトリがあれば、コレクションの階層をカプセル化することもできただろう。ファクトリのメソッドを作成する際に、クライアントが必要な機能を要求できるようにして、インスタンス化する適当なクラスをファクトリが選択するようにすることもできたはずだ。コレクションを生成するコードはもっと表現力豊かになったはずであり、新しいコレクションクラスはJavaプログラムを何ひとつ破壊することなくインストールできたはずである。

しかし、具体的なコンストラクタが有利な場合もある。そもそも、実装の選択によっては、多くのアプリケーションにおいて性能に影響を与える可能性があるので、アプリケーションで制御したいと思うかもしれないのだ（そうだとしても、本当にうまく作られたファクトリなら、そういう要因にも対応できる）。いずれにしても、コレクションクラスはあまり多くないので、選択するのはそれほど複雑ではない。

抽象的なコレクションの型は、ファクトリがなくても価値を失うことはない。これは次のような利用パターンによるものだ。コレクションは、ほとんどの場合1か所で生成され、別の場所で使用される。これが意味しているのは、最終的にコレクションを使用するクラ

イアントが、要素の追加、削除、検索を行うにあたってインタフェースとやり取りし、実装からは分離していられるということだ。コレクションクラスの選択は、コレクションを所有するオブジェクトか、そのオブジェクトのファクトリの役目であるのが普通だ。

インタフェースを設計する

ファクトリのメソッドシグネチャを設計する際には、独立したファクトリであれファクトリメソッドであれ、以下の2点を念頭に置いておくこと。

- **各操作はアトミックでなければならない。**完全な生成物を生成するのに必要なものは、ファクトリとのやりとり一回ですべて渡さなければならない。また、不変条件のどれかが満たされずに生成が失敗した場合どうなるかも決めておく必要がある。例外をスローするかもしれないし、NULLを戻すだけかもしれない。一貫性を保つため、ファクトリ内で起きたエラーに関するコーディングの標準を採用することを検討すべきである。
- **ファクトリはその引数と結合する。**入力パラメータを慎重に選択しないと、ネズミの巣のように入り組んだ依存関係を作り出しかねない。結合の度合いは、引数を使って何をしたいかによる。生成物に単純に設定するだけなら、依存関係はそれほど強くならない。引数から一部を取り出して構築に使用する場合は、結合がより緊密になる。

最も安全なパラメータは、下位の設計層に由来するものである。1つのレイヤの内部でも、基本的なオブジェクトがより上位のオブジェクトに使用されるというかたちで、自然な階層ができる傾向がある（そういうレイヤ化については、第10章「しなやかな設計」と第16章「大規模な構造」でも、それぞれ別のかたちで議論する）。

パラメータとするのが適切なもう1つのものとして、モデルにおいて生成物と密接に関連するオブジェクトがある。その場合、新しい依存関係は追加されない。前述した**購入品目**の例で、ファクトリメソッドは引数として**カタログ商品**を取っているが、これは品目に欠かせない関連である。これによって**購入注文**クラスと**商品**との間に直接的な依存関係がつけ加えられている。しかし、これら3つのオブジェクトは密接な概念的グループを形成している。どちらにしても、**購入注文**の集約はすでに**商品**を参照しているのだ。したがって、集約ルートが内部を制御できるようにして、集約の内部構造をカプセル化するのは、うまいトレードオフである。

ファクトリでは引数として渡されたものの具象クラスではなく、その抽象型を使用すること。ファクトリは生成物の具象クラスとは結合しているが、具体的なパラメータと結合する必要はないのである。

不変条件のロジックはどこへ置くべきか？

ファクトリは、生成するオブジェクトや集約に関する不変条件が、すべて満たされていることを保証する責務を負っている。それでも、あるオブジェクトに適用されるルールを、そのオブジェクトの外部に移動させるなら、その前によく考えなければならない。ファクトリは、不変条件の検証を生成物に委譲することができ、それが最善であることも多いのである。

しかし、ファクトリは生成物と特別な関係にある。ファクトリはすでに生成物の内部構造を知っているし、ファクトリの全存在理由には生成物の実装も含まれている。状況によっては、不変条件のロジックをファクトリに入れ、生成物をすっきりさせる方がよい場合もある。この方法は、集約のルール（これは多くのオブジェクトに及ぶ）に対して用いると特に魅力的だ。逆に、ファクトリメソッドが他のドメインオブジェクトに付加されている場合には、**全く魅力を感じない。**

不変条件は各操作の終わりに適用されるのが原則だが、オブジェクトで起きることが認められる変遷のせいで、不変条件が利用されないこともしばしばある。エンティティにおいて、同一性を表す属性を割り当てる際に適用されるルールもあるかもしれない。しかし、生成された後には、その同一性は不変になる。値オブジェクトは完全に不変である。アクティブなライフタイムの中では決して適用されることのないロジックであれば、オブジェクトが持っている必要はない。このような場合、不変条件を入れるのに理にかなった場所はファクトリであり、そうすることで生成物はよりシンプルなままに保たれる。

エンティティファクトリ対値オブジェクトファクトリ

エンティティファクトリは、値オブジェクトファクトリとは2つの点で異なっている。値オブジェクトは不変であるため、その生成物は、最終形として完成した状態でできあがってくる。したがって、値オブジェクトファクトリを操作する際には、生成物を完全に記述することを考えなければならない。一方のエンティティファクトリは、有効な集約を生成するのに必要な、本質的な属性だけを受け取る傾向がある。不変条件によって求められないのならば、詳細は後から追加すればよい。

ここで、あるエンティティに同一性を割り当てる際に起きる問題もある。これは値オブジェクトには関係ない。第5章で指摘したように、識別子は、プログラムで自動的に割り当てることもできるし、ユーザから指定された場合のように、外部から提供することもできる。顧客の同一性が電話番号によって追跡されるなら、その電話番号は当然、ファクトリに引数として渡されなければならない。プログラムが識別子を割り当てる場合は、ファクトリで制御するのがよい。確かに、追跡用の一意のIDを実際に生成する処理は、一般的にデータベースの「シーケンス」や他のインフラストラクチャの仕組みによって行われる。

だが、何を要求して、それをどこに入れるかを知っているのはファクトリなのだ。

格納したオブジェクトを再構成する

ここまでのところ、ファクトリはオブジェクトのライフサイクルのまさに始まりにおいて、その役割を果たしてきた。ほとんどのオブジェクトは、どこかの時点でデータベースに格納されるか、ネットワーク経由で転送されるが、現在のデータベース技術の中には、オブジェクトの内容が持つ特徴を維持するものはほとんどない。ほとんどの変換方法は、オブジェクトをより制限された表現に平坦化する。したがって、検索には部品を組み立て直して生きたオブジェクトにするという、潜在的に複雑なプロセスが必要になる。

再構成に使用されるファクトリは、生成に使用されるものとよく似ているが、2つの大きな違いがある。

1. **再構成に使用されるエンティティファクトリでは、新しい追跡IDを割り当てることがない。**割り当ててしまうと、再構成前のオブジェクトとの連続性が失われてしまう。そのため、識別用の属性は、格納されているオブジェクトを再構成するファクトリに対する入力パラメータの一部でなければならない。
2. **オブジェクトを再構成するファクトリは、不変条件の違反を違うかたちで制御する。**新規オブジェクトを生成する場合、不変条件が満たされなければ、ファクトリは処理を中止するだけでよいが、再構成の場合には、より柔軟な対応が必要になることがある。すでにオブジェクトがシステムのどこか（データベース中など）に存在しているならば、その事実を無視することはできない。そうかといって、ルール違反を無視することもできない。そういう矛盾を埋め合わせる何らかの戦略が必要で、それによって再構成は、オブジェクトの新規生成よりも難しくなるかもしれない。

図6.16と6.17は、2種類の再構成を示している。オブジェクトマッピング技術を使えば、データベースから再構成する場合に、前述したようなことを行う便利なサービスの一部または全部が提供されるかもしれない。あるオブジェクトを他の媒体から再構成する際に、その複雑さが無防備にさらされている場合は、ファクトリが適切な選択肢となる。

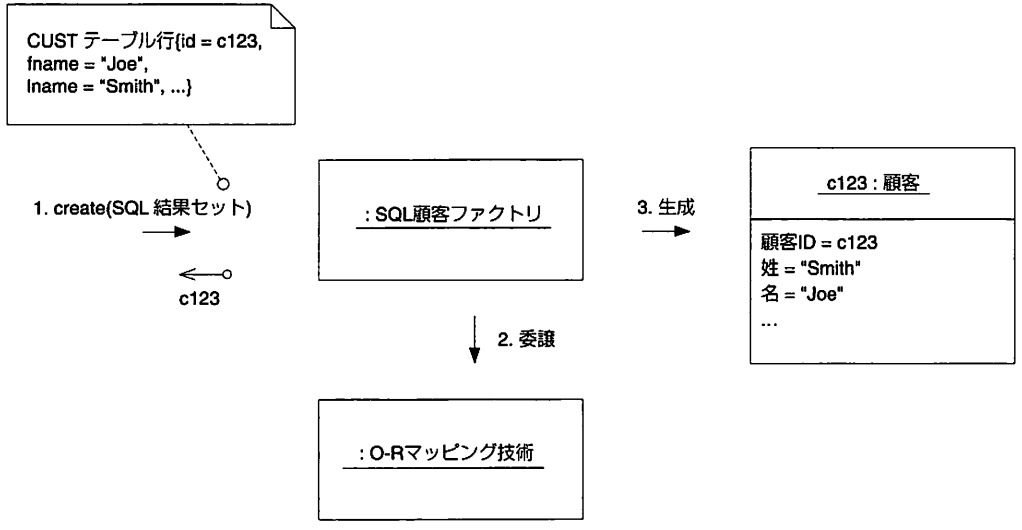


図6.16 関係データベースから取り出された**エンティティ**の再構成

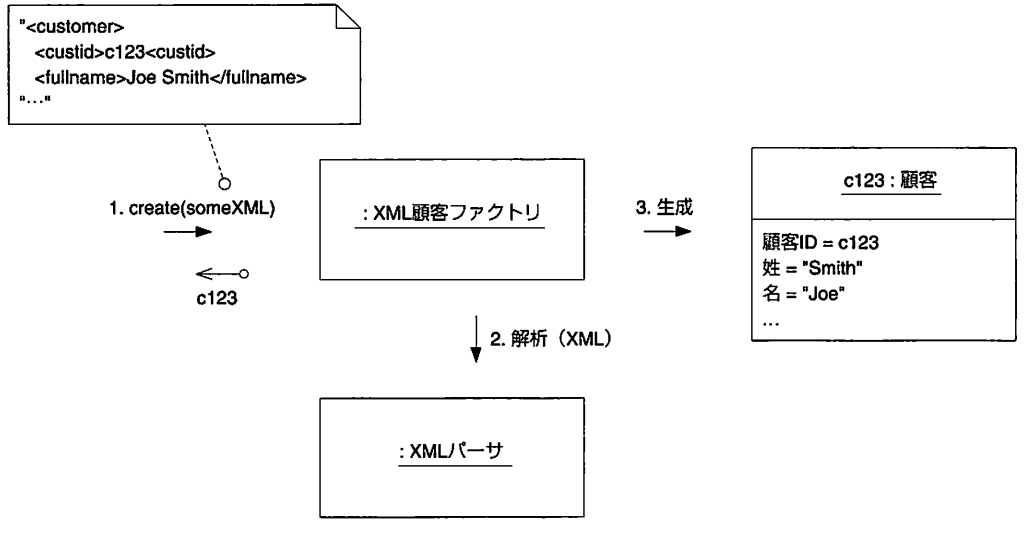
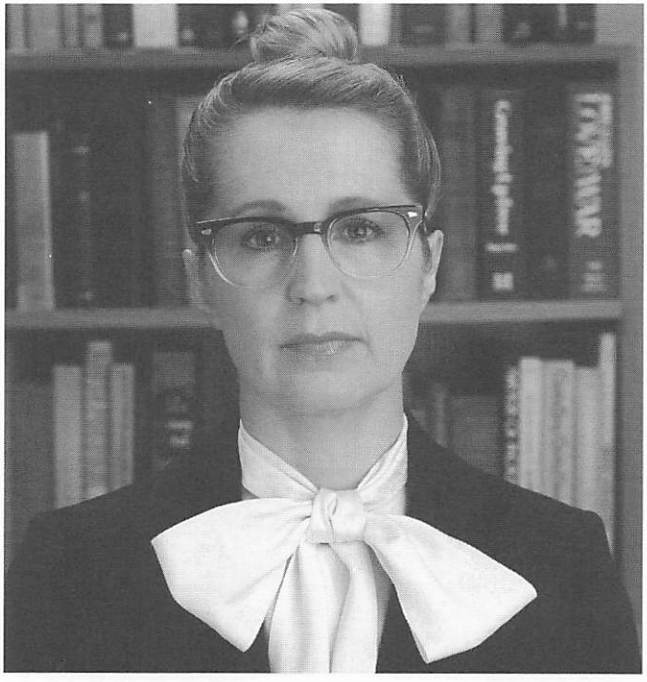


図6.17 XMLとして転送された**エンティティ**の再構成

要約すると、インスタンス生成のためのアクセスポイントは特定する必要があり、そのスコープは明示的に定義しなければならない。そうしたアクセスポイントは、単なるコンストラクタのこともあるが、より抽象的で手の込んだインスタンス生成のための仕組みが必要になることもしばしばある。この要求によって、設計に新しい構成要素、すなわち、ファクトリが導入される。通常、ファクトリはモデルのどこかの部分を表現するものではないが、ドメイン設計の一部であり、モデルを表現するオブジェクトを鮮明にしておく上で役に立つ。

ファクトリは生成と再構成というライフサイクルの変遷をカプセル化する。技術的な複雑さを露呈させて、ドメイン設計を浸食しかねない変遷の中には、ストレージへの移動とストレージからの移動もある。この変遷について責務を負うのは、ドメイン設計のもう1つの構成要素であるリポジトリである。

リポジトリ（REPOSITORIES）



関連があることによって、あるオブジェクトを見つけるにあたり、他のオブジェクトへの関係性に基づいて探すことができるようになる。しかし、ライフサイクルの中期にあるエンティティや値オブジェクトに対しては、関連を辿り始める場所がなければならない。

＊　＊　＊

オブジェクトを用いて何かを行うには、そのオブジェクトへの参照を持っていなければならない。では、その参照をどのようにして手に入れるのか？ 1つめの方法はオブジェクトを生成することだ。生成する操作によって、新しいオブジェクトへの参照が戻される。2つめの方法は、関連を辿ることだ。すでに知っているオブジェクトから始め、関連しているオブジェクトを要求する。どんなオブジェクト指向プログラムでも、こうしたことは頻繁に行うことになるし、そうしたリンクこそが、オブジェクトモデルに表現力の多くを与えている。しかし、その最初のオブジェクトは取得しなければならない。

私がかつて実際に遭遇したプロジェクトでは、チームが熱狂的にモデル駆動設計を採用していて、**あらゆる**オブジェクトへのアクセスを、生成するか関連を辿ることによって行

おうとしていた！ これらのオブジェクトはオブジェクトデータベースにあり、既存の概念的な関係性によって、必要な関連はすべて提供されると考えられていた。そのプロジェクトでは、概念的な関係性を十分に分析して、ドメインモデル全体の凝集度を高めればよかった。だが、こうした制約を自らに課すことによって、彼らは、終わりのない関係性のもつれを作り出すことになってしまったのだ。それは、まさにこれまでの数章で、エンティティを注意深く実装し、集約を適用させることによって避けようとしてきたものだった。チームメンバはこの戦略に長くは固執しなかったが、一貫性のある別のアプローチに替えることもしなかった。その場しのぎの解決策をつぎはぎして、意欲も失っていたのである。

このようなアプローチについて考えてみようという人はごくわずかで、興味をそそられる人などいないに等しかっただろう。一般的には、ほとんどのオブジェクトを関係データベースに格納していたからだ。だが、このストレージ技術を用いれば、参照を得るのに第三の方法を使用するのが自然となる。その方法とは、クエリを実行して、属性に基づいてデータベース内でオブジェクトを見つけるか、もしくは、オブジェクトの構成要素を見つけて、それを再構成するというものだ。

データベース検索はグローバルにアクセスすることができて、どんなオブジェクトにも直接到達できる。すべてのオブジェクトが相互に結びついている必要はないから、オブジェクトのネットワークは管理しやすく保てるようになる。関連を辿れるようにするか、それとも検索に頼るかは設計上の決定であり、検索の疎結合と関連の高凝集との間でトレードオフを図ることになる。顧客オブジェクトは、出した注文すべてのコレクションを保持するべきなのか？ あるいは、注文はデータベースに入れて、顧客ID列で検索するようにすべきなのか？ 検索と関連を正しく組み合わせることで、設計はわかりやすくなる。

残念ながら、開発者は通常、そういう設計の機微についてあまり多くを考えない。オブジェクトを格納して再び取り出し、最後にはストレージから削除するという芸当をうまくやるのに必要な仕組みは、海のように広大であり、開発者はその中を泳いでいるようなものだからだ。

技術的な観点から見ると、格納したオブジェクトを取り出すことは、実は生成することのサブセットである。データベースから取り出されたデータは、新しいオブジェクトを組み立てるのに使用されるからだ。確かに、通常書かなければならないコードを思えば、この現実は忘れ難い。しかし、概念上は、これはエンティティのライフサイクルにおける**中期**なのである。ある顧客オブジェクトは、いったんデータベースに格納して取り出したからといって、新規の顧客を表すわけではない。この区別を忘れないようにするために、格納されたデータからインスタンスを生成することを**再構成**と呼んでいるのだ。

ドメイン駆動設計の目標は、技術ではなくドメインについてのモデルに焦点を合わせる

ことによって、よりよいソフトウェアを作ることである。開発者がSQLクエリを構築し、それをインフラストラクチャ層のクエリサービスに渡して、テーブル行の結果セットを取得し、必要な情報を引き出すなどということをしていたのでは、その情報をコンストラクタかファクトリへ渡す頃には、モデルに合わせていた焦点が逸れてしまう。オブジェクトのことを、問い合わせによって取得されるデータのコンテナであると自然に考えるようになり、設計全体がデータ処理のスタイルへと移行する。技術の詳細はさまざまであっても、クライアントが扱っているのが技術であって、モデルの概念ではないという問題は残る。メタデータマッピング層（METADATA MAPPING LAYERS）（Fowler 2002）のようなインフラストラクチャは、非常に役立つもので、クエリの結果をオブジェクトへ容易に変換できるようにしてくれるが、それでもまだ開発者が考えているのは、ドメインではなく技術的な仕組みなのである。さらにまずいことに、クライアントコードがデータベースを直接使用していると、開発者は、集約のようなモデルの機能や、オブジェクトのカプセル化さえも迂回し、その代わりに必要なデータを直接取り出して操作したくなってしまう。ますます多くのドメインのルールがクエリのコードに埋め込まれるか、単に失われてしまう。オブジェクトデータベースを用いれば変換に伴う問題が取り除かれるのは確かだが、検索の仕組みはやはり機械的でモデルを意識しないため、開発者は相変わらず、欲しいオブジェクトを何でも直接つかんでしまおうという気になる。

クライアントが必要とするのは、すでに存在するドメインオブジェクトへの参照を手に入れる、実用的な手段である。インフラストラクチャによってそれが簡単にできるようになるとすると、クライアント開発者は辿ることのできる関連をさらに追加して、モデルを混乱させてしまうかもしれない。一方で、クエリを使用してデータベースから必要なデータだけを引き出したり、集約ルートから進む代わりに数個の特定のオブジェクトを取り出したりすることもあるかもしれない。ドメインロジックがクエリとクライアントコードに移され、エンティティと値オブジェクトはただのデータコンテナとなる。データベースアクセスインフラストラクチャを適用するのは、たいていの場合、純粋に技術的に見ても複雑である。そうした複雑さによって、たちまちクライアントコードが浸食されてしまい、結果として、開発者はドメイン層を縮小して、モデルが的外れになってしまう。

これまで議論した設計の原則を活用することで、我々は、オブジェクトへのアクセスに関する問題のスコープを若干狭めることができる。ただしそれは、設計の原則を採用できるくらいに、モデルに対して鋭く集中し続けられるようなアクセス手法を見つけたと想定した上でのことだ。まず、一時的なオブジェクトは気にする必要がない。一時的なオブジェクト（典型的なのは値オブジェクト）はライフサイクルが短く、それを使うクライアントの操作の中で生成され、その後破棄される。さらに、永続化されたオブジェクトのうちで、関連を辿って見つける方が便利なものに対しても、クエリによるアクセスは必要な

い。例えば、ある人の住所は、その人を表す人オブジェクトから要求できるだろう。そして、何より重要なことに、**集約内部にあるどのオブジェクトも、ルートから辿る以外の方法でアクセスすることは禁止されている。**

永続化された値オブジェクトを見つけるには、それをカプセル化する集約のルートとして機能するエンティティから関連を辿るのが普通である。実際に、値に対して、グローバルな検索によってアクセスするのは無意味なことが多い。プロパティを用いて値オブジェクトを検索するのは、そのプロパティを持つ新しいインスタンスを生成するのと同じだからだ。しかし、いくつか例外もある。例えば、オンラインで旅行の計画を立てる場合、見込みのある旅程をいくつか保存しておき、後から戻って、予約する旅程を選ぶことがある。この旅程は値オブジェクトだが（同じ飛行便からなる2つの旅程があったとして、どちらがどちらであるかは気にならない）、これは私のユーザ名と関連づけられていて、元のままの状態で取り出される。別の例としては「列挙（enumeration）」があるだろう。これは型が厳密に制限されていて、あらかじめ取り得る値が決まっている場合だ。そうは言っても、値オブジェクトへのグローバルアクセスは、エンティティに対して行われるよりずっとまれだ。もし、すでにある値オブジェクトをデータベースで検索する必要が生じたのであれば、まだ同一性を認識していないだけで、実はエンティティを取得しようとしているのではないかと考えてみる価値がある。

これまでの議論により、ほとんどのオブジェクトに対しては、グローバルな検索でアクセスすべきでないということが明らかになった。そうすべきものについては、設計で伝える方がよいだろう。

これで、問題をより正確に言い直すことができる。

永続化されたオブジェクトのサブセットの中には、オブジェクトの属性に基づいた検索を通じて、グローバルにアクセスできなければならないものもある。そういうアクセスを必要とするのは、集約のルートのうち、関連を辿って到達しようとすると都合の悪いものだ。こうした集約ルートは通常エンティティだが、複雑な内部構造を持つ値オブジェクトのこともあれば、列挙された値のこともある。集約ルート以外のオブジェクトにグローバルなアクセスを提供してしまうと、重要な区別が台無しになる。データベースクエリを自由に行うと、実はドメインオブジェクトと集約のカプセル化に違反しかねないのだ。技術的なインフラストラクチャやデータベースアクセスの仕組みを露呈させてしまうと、クライアントが複雑になり、モデル駆動設計が不明瞭になってしまう。

データベースアクセスに起因する技術的な課題に対処するテクニックは多数ある。例えば、SQLをクエリオブジェクト（QUERY OBJECT）にカプセル化することや、メタデータマッピング層（Fowler 2002）でオブジェクトとテーブル間の変換を行うことなどが挙げられる。ファクトリは格納されたオブジェクトを再構成するのに役立つ（本章で後述する）。

この他にも多くのテクニックにより、複雑さを隠しておける。

しかし、それでも、失われてしまったものには注意しなければならない。我々が考えているのは、もはや、ドメインモデルに存在する概念についてではないのだ。コードはビジネスについて伝えるのではなく、データ検索技術を扱うようになる。リポジトリパターンは、そういう解決策をカプセル化してモデルに焦点を合わせ直す、シンプルな概念的フレームワークなのだ。

リポジトリは、特定の型のオブジェクトを、すべて概念上の集合（通常は、それを模したもの）として表現する。これはコレクションのように動作するが、より手の込んだクエリ機能を持っている。適切な型のオブジェクトについて追加や削除が実行されると、リポジトリの背後にある仕組みが、データベースに対して挿入や削除を行う。この定義によって集められる、凝集した責務の集合によって、ライフサイクルの始まりから終わりまでを通じて、集約のルートに対するアクセスが提供されるのだ。

クライアントがリポジトリに対してオブジェクトを要求する際はクエリメソッドを使用する。このクエリメソッドは、ある属性の値といった、クライアントが指定した条件に従ってオブジェクトを選択する。リポジトリは要求されたオブジェクトを取り出すが、そこではデータベースクエリやメタデータマッピングの仕組みがカプセル化されている。リポジトリが実装できるクエリは多岐にわたっており、クライアントがどんな条件を要求しても、それに基づいてオブジェクトを選択できる。条件を満たすインスタンスの数といった集計結果を戻すこともできる。さらに、条件に一致したオブジェクトが持つ何らかの数値属性をすべて合計したもののような、集計演算の結果を戻すことさえできるのだ。

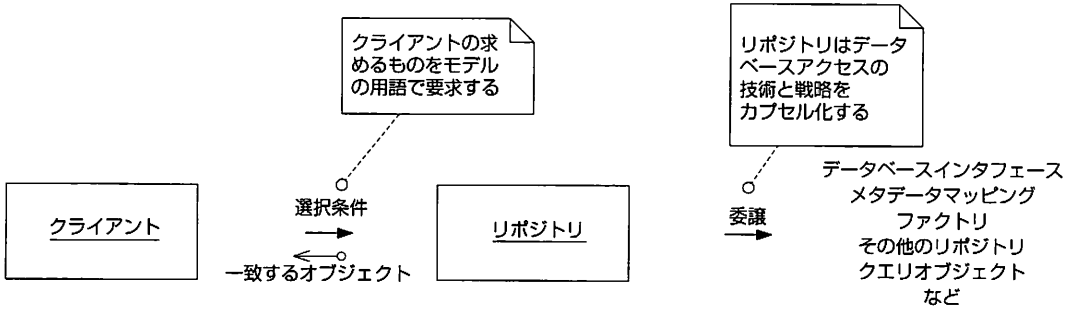


図6.18 クライアントのために検索を実行するリポジトリ

リポジトリのおかげで、クライアントは相当な負担から解放される。それによってクライアントは、シンプルで意図の明白なインタフェースとやり取りして、モデルの言葉で必要なものを要求できる。こうした要求をすべてサポートするには、複雑な技術的インフラ

ストラクチャが必要だが、リポジトリのインタフェースはシンプルで、ドメインモデルと概念的に結びついている。

それゆえ：

グローバルアクセスを必要とするオブジェクトの各型に対して、あるオブジェクトを生成し、その型のすべてのオブジェクトで構成されるコレクションが、メモリ上にあると錯覚させることができるようにすること。よく知られているグローバルインタフェースを経由してアクセスできるようにすること。オブジェクトの追加と削除を行うメソッドを提供し、データストアにおける実際のデータの挿入や削除をカプセル化すること。また、ある条件に基づいてオブジェクトを選択し、属性値が条件に一致するような、完全にインスタンス化されたオブジェクトかオブジェクトのコレクションを戻すメソッドを提供すること。それによって、実際のストレージや問い合わせの技術をカプセル化すること。実際に直接的なアクセスを必要とする集約ルートに対してのみ、リポジトリを提供すること。クライアントをモデルに集中させ、あらゆるオブジェクトの格納とアクセスをリポジトリに委譲すること。

＊　＊　＊

リポジトリには、次に挙げるものを含め、数多くの利点がある。

- クライアントに対して、永続化されたオブジェクトを取得し、そのライフサイクルを管理するためのシンプルなモデルを与える。
- アプリケーションとドメインの設計を、永続化技術や複数のデータベース戦略、さらには複数のデータソースからも分離する。
- オブジェクトアクセスに関する設計上の決定を伝える。
- テストで使用するために、ダミーの実装で置き換えるのが容易になる（インメモリコレクションを使用するのが一般的）。

リポジトリに対して問い合わせる

あらゆるリポジトリは、クライアントが何らかの条件に一致するオブジェクトを要求できるようにするためのメソッドを提供する。だが、このインタフェースの設計方法にはさまざまな選択肢がある。

構築するのが最も簡単なリポジトリは、ハードコードされたクエリを持ち、特定のパラメータを受け取るものだ。クエリには、さまざまなものがあり得る。識別子を用いてエンティティを取り出したり（ほぼすべてのリポジトリによって提供される）、特定の属性値やパラメータの複雑な組み合わせを用いてオブジェクトのコレクションを要求したり、値の範囲（日付範囲など）に基づいてオブジェクトを選択したり、さらには、リポジトリに与えられる一般的な責務の範囲内で、何らかの計算を実行したりするものまである（特に、

下層にあるデータベースがサポートする操作を利用するもの)。

ほとんどのクエリはオブジェクトかオブジェクトのコレクションを戻すが、何らかのかたちの集計演算を戻すという概念にも適している。これには、オブジェクトの数やモデルが計算することを想定している数値属性の合計などがある。

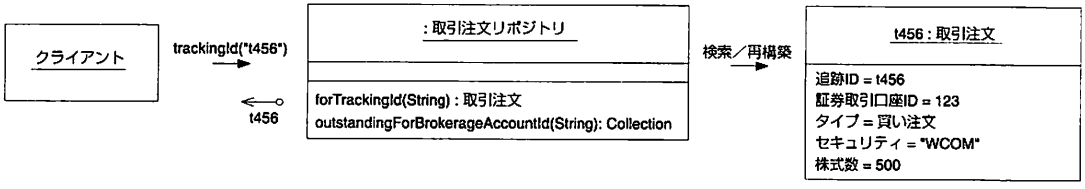


図6.19 単純な**リポジトリ**内でハードコードされたクエリ

ハードコーディングされたクエリであれば、どんなインフラストラクチャの上にも構築できるし、多額の投資も必要ない。いずれにせよ、リポジトリがなければクライアントがしなければならなかったことを、実行するだけだからだ。

大量の問い合わせが必要なプロジェクトの場合、リポジトリフレームワークは、より柔軟なクエリが実行できるように構築されるだろう。これには必要な技術に精通したスタッフが求められるし、そうした機能をサポートするインフラストラクチャがあれば、大いに役に立つ。

フレームワークを用いてリポジトリを一般化する上で、とりわけ適したアプローチの1つが、仕様パターンに基づいたクエリを使用することである。仕様を使うことで、クライアントは取得方法を気にすることなく、求めるものを記述（つまり、仕様を定めること）できるようになる。その過程で、実際に選択処理を実行できるオブジェクトが生成される。このパターンについては、第9章で詳しく議論する。

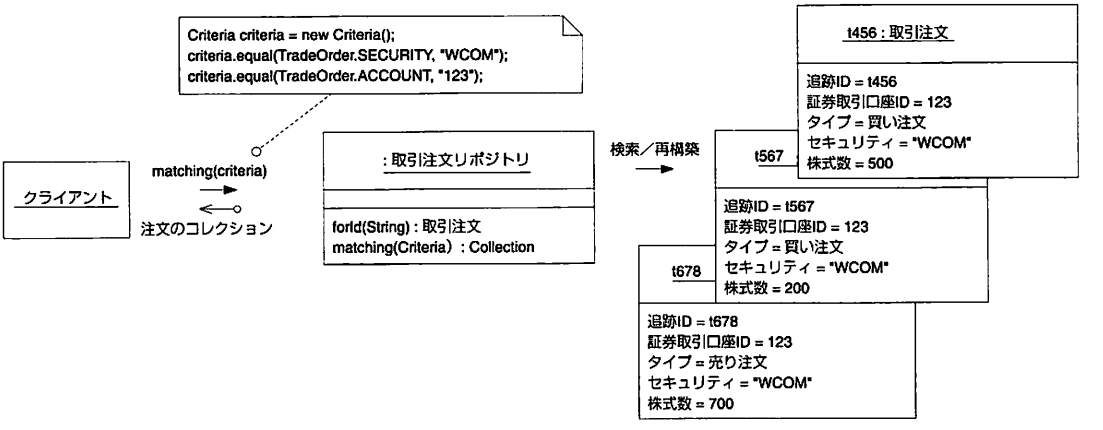


図6.20 洗練された**リポジトリ**における、柔軟で宣言的な検索条件の**仕様**

仕様に基づいたクエリはエレガントで柔軟性がある。利用可能なインフラストラクチャ次第で、そう複雑でないフレームワークになることもあれば、恐ろしく難解になることもある。『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』（Fowler 2002）において、Rob MeeとEdward Hieattが、そういうリポジトリの設計にまつわる技術的問題について議論している。

柔軟にクエリを構築できるリポジトリを設計する際にも、特殊なクエリをハードコードして追加できるようにしなければならない。追加されるのは、よく使われるクエリや、オブジェクトそのものを戻すわけではないクエリをカプセル化した便利なメソッドかもしれない。ここで言う、クエリから戻されるオブジェクト以外のものには、選択されたオブジェクトの数学的集計結果などがある。フレームワークで、そうした不測の事態を考慮しておかないと、ドメインの設計をゆがめたり、開発者に避けられたりすることになるのだ。

クライアントのコードはリポジトリの実装を無視するが、開発者はそうではない

永続化技術をカプセル化することで、クライアントは非常にシンプルになり、リポジトリの実装から完全に切り離される。しかし、カプセル化ではよくあるように、開発者は中で具体的に何が起こっているかを理解していなければならない。リポジトリが異なるやり方で使われたり、異なるかたちで動作したりすると、性能に極端な影響を与える可能性がある。

Kyle Brownが、WebSphereが本格展開され始めた頃に、この製品を用いた製造業アプリケーションの応援に呼ばれた時のことを話してくれた。そのシステムは、不思議なことに使用開始から数時間でメモリ不足になった。Kyleはコードにざっと目を通し、その原因を

発見した。このアプリケーションは、ある時点で工場内の全品目に関する情報を集計していたのである。開発者はそのために「全オブジェクト」と呼ばれるクエリを使用していたが、これはすべてのオブジェクトをそれぞれインスタンス化してから、必要なごく一部を選択していた。このコードには、データベースの情報をまるごと一度にメモリ上へ持ち込むという作用があったのだ！ 問題がテストで発覚しなかったのは、テストデータの量が少なかったからである。

これは明らかに、やってはならないことだが、より些細な見落としがあった場合でも、同様に深刻な問題になりかねない。開発者は、カプセル化されたふるまいを使用した際の影響について、理解する必要があるのだ。それは実装を熟知しなければならないということではない。うまく設計されたコンポーネントには特徴がある（これが第10章「しなやかな設計」の要点の1つである）。

第5章で議論したように、下層にある技術によって、モデリングの選択肢が制限されることがある。例えば、関係データベースを使うと、深い複合オブジェクトの構造に事実上の制限が設けられるかもしれない。全く同様に、リポジトリの使用とそのクエリの実装との間で、双方向のフィードバックが開発者に対してなされなければならない。

リポジトリを実装する

永続化のために使用される技術と手持ちのインフラストラクチャ次第で、実装は大きく異なってくる。理想は、クライアントから（クライアントの開発者からではないが）内部の働きをすべて隠蔽し、それによってクライアントコードが、データの格納場所に関わらず同じになることだ。そうすれば、データの格納先がオブジェクトデータベースでも、関係データベースでも、あるいはただメモリ上に保持されているだけでも関係なくなる。リポジトリは処理の実行を適切なインフラストラクチャサービスに委譲する。格納と取り出し、および問い合わせの仕組みをカプセル化することが、リポジトリの実装における最も基本的な機能である。

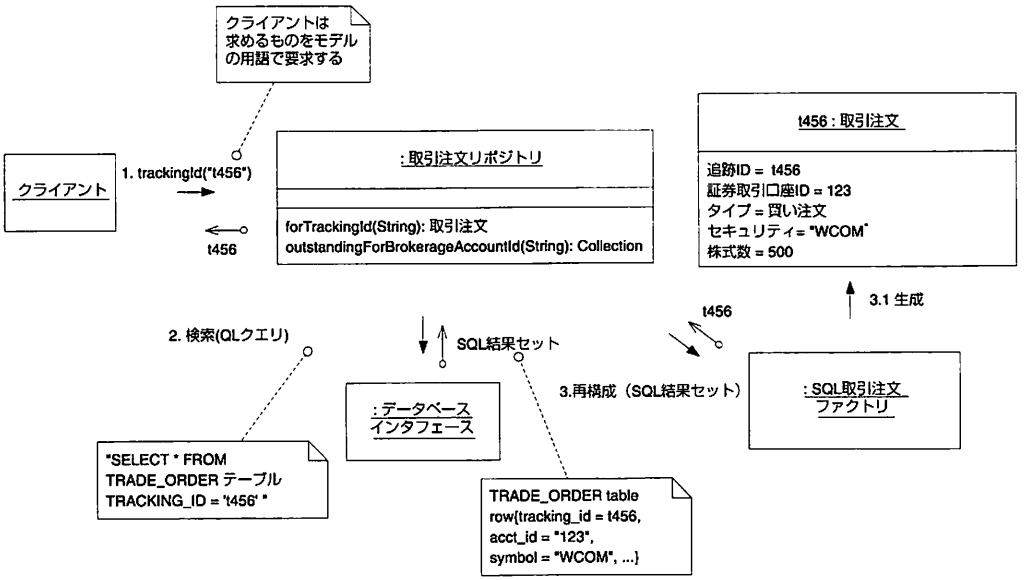


図6.21 **リポジトリ**は基底にあるデータストアをカプセル化する

リポジトリの概念は多くの状況に適応できる。実装のやり方はきわめて多岐にわたるので、ここでは覚えておくべき懸案をいくつか挙げるだけにしておく。

- **型を抽象化すること。**リポジトリは特定の型のインスタンスをすべて「含んでいる」が、それは各クラスにリポジトリが1つ必要だということではない。この型は階層における抽象基底クラスでも構わない（例えば、**取引注文**は**買い注文**にも**売り注文**にもなり得る）。その型はインタフェースで、実装とは階層上の関係がないかもしれないし、特定の具象クラスかもしれない。覚えておいて欲しいのだが、データベースにはこうした多態性が欠けているので、そのことで課されるさまざま制約に直面するかもしれない。
- **クライアントから切り離す利点を活かすこと。**クライアントが実装の仕組みを直接呼び出していた場合に比べると、リポジトリの実装はより自由に変更できるようになる。このことは、性能を最適化する上で生かすことができる。クエリのテクニックを変更したり、オブジェクトをメモリへキャッシュしたりと、永続化の戦略をいつでも切り替えられるのだ。クライアントコードとドメインオブジェクトのテストは、簡単に操作できるダミーのインメモリ戦略を使うことで楽になる。
- **トランザクション制御をクライアントに委ねること。**リポジトリはデータベースに対する挿入と削除を行うが、通常は何もコミットしない。例えば、保存した後にはコ

ミットしたくなるが、おそらくクライアントには、作業ユニット（unit of work）を正しく開始し、コミットするためのコンテキストがある。トランザクション管理は、リポジトリが手を出さないでいる方が単純になる。

リポジトリの実装をサポートするために、チームがインフラストラクチャ層にフレームワークを追加するのは、よくあることだ。下位レベルにあるインフラストラクチャコンポーネントと協調することに加え、リポジトリの基底クラスは基本的なクエリを実装することもある。柔軟にクエリを組み立てるように実装される場合は特にそうだ。残念ながら、Javaなどにあるような型システムでは、こうしたアプローチを採用すると、戻されるオブジェクトの型を「Object」にしなければならず、それをリポジトリが持つ型にキャストするのはクライアントにゆだねられる。だが当然ながら、こうしたことは、Javaでコレクションを戻すクエリにおいては、いずれにしてもやらなければならないことだ†。

リポジトリと、クエリオブジェクトのようなそれをサポートする技術的なパターンの実装に関しては、さらなる指針が『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』（Fowler 2002）に書かれている。

フレームワークの範囲内で作業する

リポジトリのようなものを実装する前には、これから扱うことになるインフラストラクチャ、中でもアーキテクチャフレームワークについて慎重に検討する必要がある。こうしたフレームワークは、リポジトリを簡単に作成するのに使用できるサービスを提供することもあれば、どこまでもこちらの意に反することもある。アーキテクチャフレームワークによって、永続化されたオブジェクトを取得するためのリポジトリと同等のパターンが、すでに定義されているかもしれない。逆に、定義されているパターンは、リポジトリとは似ても似つかないかもしれない。

例えば、あなたの今いるプロジェクトがJ2EEに傾注しているとする。このフレームワークとモデル駆動設計のパターンの間で概念的に似ているものを探すと、（エンティティBeanがエンティティと同じものでないことを忘れずにいたとしても）集約ルートに対応するものとしてエンティティBeanを使うことにするかもしれない。J2EEのアーキテクチャフレームワークにおいて、それらのオブジェクトに対するアクセスを提供する責務を負う構成要素は「EJBホーム」である。EJBホームの体裁を整えて、リポジトリのように見せようとすると、また別の問題につながりかねない。

---

† 訳注：第4章のフレームワークと同じく、これもJavaのバージョンが1.4であったことによる問題である。Java 5.0で導入された総称型（ジェネリクス）によって、コレクションに型情報を持たせられるようになったため、現在ではクライアントコードでのキャストは不要になっている。

一般的に言えることだが、使っているフレームワークとは争わないこと。フレームワークと対立してしまった時には、ドメイン駆動設計の基本を保ちながら、詳細は捨て去る方法を模索すること。ドメイン駆動設計の概念とフレームワークの概念に似通った部分がないか探すこと。ここでは、そのフレームワークを使用する以外に、選択肢がないことを想定している。多くのJ2EEプロジェクトでは、エンティティBeanを一切使用していない。自由に裁量できるなら、使用したい設計スタイルと調和するフレームワークか、その一部を選択すべきである。

ファクトリとの関係

ファクトリは、オブジェクトのライフサイクルにおける始まりを処理し、リポジトリは中期から終わりを管理するのを助ける。オブジェクトがメモリ上に保持されているか、オブジェクトデータベースに格納されている場合には、こうした役割は直接的でわかりやすい。しかし通常は、オブジェクトの格納場所として、関係データベースやファイルなど、少なくとも何らかのオブジェクト指向ではないシステムがあるものだ。このような場合、取り出したデータはオブジェクトのかたちに再構成しなければならない。

こうした場合、リポジトリはデータに基づいてオブジェクトを生成するので、多くの人はリポジトリのことをファクトリ**である**と考える。確かに、技術的観点からはそうだ。しかし、モデルは常に前面に押し出しておいた方がよいし、すでに述べたように、格納されたオブジェクトを再構成することは、新しい概念オブジェクトを生成することとは違う。このように、設計をドメイン駆動の観点から見てみると、ファクトリとリポジトリには別々の責務があるのだ。ファクトリは新しいオブジェクトを生成し、リポジトリは古いオブジェクトを見つけ出す。リポジトリのクライアントには、オブジェクトがメモリにあると錯覚させなければならない。オブジェクトは再構成されるかもしれないが（そう、新しいインスタンスが生成されるのだ）、それは以前あったものと同じ概念オブジェクトであり、まだライフサイクルの中期にいるのである。

これら2つの見方を一致させるためには、リポジトリがオブジェクトの生成をファクトリへ**委譲**すればよく、そのファクトリは（理論上の話で、実際はめったにないが）、オブジェクトを一から生成するのにも使用できる。

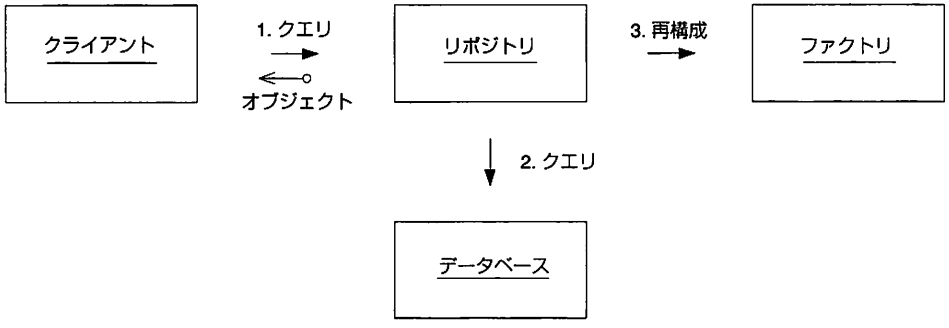


図6.22 **リポジトリ**は既存のオブジェクトを再構成するにあたって**ファクトリ**を使用する

このように、明確に分離することで、ファクトリから永続化に関するすべての責務を取り除くこともできる。ファクトリの仕事は、潜在的に複雑なオブジェクトを、データを基にインスタンス化することである。その生成物が新しいオブジェクトであれば、クライアントにはそのことがわかるし、それをリポジトリに追加することができる。このリポジトリによって、データベースへのオブジェクトの格納はカプセル化されている。

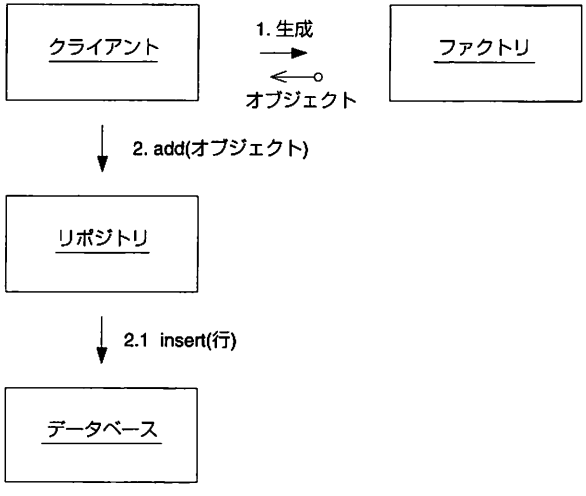


図6.23 クライアントは、新しいオブジェクトを格納するのに**リポジトリ**を使用する

ファクトリとリポジトリを組み合わせたいと思わせるものの1つに、「探して、なければ生成する」という機能に対する欲求がある。この機能を使うと、クライアントは求めるオブジェクトを記述することができて、そういうオブジェクトが見つからない場合には、新しく生成されたものが与えられる。しかし、こういう機能は避けなければならない。せいぜい、少し便利になるにすぎないのだ。エンティティと値オブジェクトを区別すれば、こうした機能が便利に見える状況はほとんどなくなる。値オブジェクトが欲しいクライアントは、ファクトリに対して新しいオブジェクトを直接要求すればよい。通常、新しいオブジェクトと既存のオブジェクトを区別することは、ドメインにおいて重要であり、この両者を透過的に組み合わせるフレームワークは、実際には状況を混乱させてしまう。

関係データベースに合わせてオブジェクトを設計する

主要なパラダイムがオブジェクト指向であるソフトウェアシステムの中にあって、オブジェクトではない構成要素のうち、最も一般的なものが関係データベースである。この現実によって、パラダイムの混在につきまとう問題がもたらされる（第5章参照）。しかし、データベースは、他のほとんどの構成要素よりもオブジェクトモデルと密接に関係している。データベースは、オブジェクトと相互作用するだけではなく、オブジェクト自体を構成しているデータを永続化のための形式で格納することもできるのだ。オブジェクトを関係テーブルへマッピングさせ、効果的に格納し取り出すことに関する技術的な課題については、これまでも数多論じられてきた。最近の議論は『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』（Fowler 2002）に見られる。オブジェクトと関係テーブルの間にマッピングを作成し管理する上では、それなりに洗練されたツールもある。こうした技術的な関心事を別にしても、両者の間にある不整合は、オブジェクトモデルに対して重大な影響を及ぼす可能性がある。

一般的なケースは3つある。

1. データベースの主要な目的が、オブジェクトを格納することである。
2. データベースが、別のシステムのために設計されている。
3. データベースは該当するシステムのために設計されているが、オブジェクトを格納する以外の役割を果たしている。

データベーススキーマがオブジェクトを格納するために特別に作成されている場合、マッピングをごく単純に保つためにモデルの制約を受け入れることには価値がある。スキーマの設計に対する要求が他になければ、更新する際の集約の整合性がより安全で効率的になるように、データベースを構築できる。技術的には、関係テーブルの設計がドメイ

ンモデルを反映している必要はない。マッピングツールが非常に洗練されているので、顕著に相違があっても橋渡しできる。問題は、複数の重なり合ったモデルが入り組みすぎている場合である。分析モデルと設計モデルが分離するのを避けるという、モデル駆動設計について行ってきた議論の多くが、ここでの不整合にも当てはまる。ここでは確かに、オブジェクトモデルの豊かさを多少犠牲にする必要があり、時にはデータベース設計の方で歩み寄らなければならない（選択的な非正規化など）。そうしなければ、モデルと実装の緊密な結びつきが失われてしまう危険があるのだ。このアプローチでも、1オブジェクトに対して1テーブルを単純にマッピングさせる必要はない。マッピングツールの能力次第で、何らかの集約やオブジェクトの組み立てが可能になることもある。しかし、ここで欠かせないのが、マッピングが透過的であるということと、コードを調べたり、マッピングツールの説明を読んだりすることによって簡単に理解できるということだ。

- データベースをオブジェクトの格納先として見る場合には、マッピングツールの能力に関わらず、データモデルとオブジェクトモデルをかけ離れたものにしてはならない。関係モデルと近づけておくため、オブジェクトの関係性が持つ豊かさを若干犠牲にすること。オブジェクトとのマッピングを単純化するのに役立つなら、正規化のような、関係モデルが持つ正式な標準に関して、ある程度は妥協すること。
- オブジェクトシステムの外部にあるプロセスからは、そうしたオブジェクトの格納先にアクセスしてはならない。オブジェクトが強制する不変条件に違反する可能性があるからだ。また、そうしたアクセスがあるとデータモデルが固定化されてしまい、オブジェクトをリファクタリングする際に、変更するのが難しくなる。

一方で、データが、オブジェクトの格納先として全く想定されていなかった、レガシーや外部のシステムから来ているという場合も多い。こういう状況では、実は、同一のシステムに2つのドメインモデルが共存しているのだ。第14章「モデルの整合性を維持する」で、この問題を掘り下げる。他システムで暗に存在しているモデルに順応するのが理にかなっていることもあれば、モデルを完全に別のものにする方がよい場合もある。

例外を作り出す別の理由としては性能もある。実行速度の問題を解決するために、設計に奇抜な変更を導入しなければならないこともあるのだ。

しかし、関係データベースが、オブジェクト指向ドメインの永続化された形式として機能するというのは、重要で一般的な状況だ。ただ、この場合であれば、単純で直接的なかたちにするのが最適である。1つのテーブル行には1つのオブジェクトが含まれるべきであり、もしかすると集約の支配下にあるものと一緒になっているかもしれない。テーブル中の外部キーは、別のエンティティオブジェクトに対する参照に変換しなければならない。時には、この単純で直接的な実装から逸脱する必要があったとしても、それによってマッピングは単純にするという原則を、すっかり放棄することになってはならない。

ユビキタス言語は、オブジェクトモデルと関係モデルのコンポーネントを結びつけて、単一のモデルにするのに役立つ。オブジェクト内の要素が持つ名前と関連は、関係テーブルのものと紐づけるにあたって、細心の注意を払うべきである。マッピングツールの中には強力なものもあって、こうした紐づけが不要であるかのように見せるかもしれないが、関係性が微妙に異なっていると大変な混乱が引き起こされる。

オブジェクトの世界では徐々に定着してきたリファクタリングの慣習も、関係データベースの設計には、実はあまり影響を与えていない。その上、データ移行という重大な問題が頻繁な変更に水を差す。これによって、オブジェクトモデルのリファクタリングが妨げられるかもしれないが、オブジェクトモデルとデータモデルが分岐しだしたら、透過性が急速に失われかねない。

最後に、オブジェクトモデルとは全く異なるスキーマと共存しなければならない理由もいくつかある。これは、自分のシステム専用にデータベースが作成される場合でも起こり得る。そのデータベースは、オブジェクトをインスタンス化しない他のソフトウェアからも使用されるかもしれない。オブジェクトのふるまいが急速に変化したり、進化したりしている間も、データベースにはほとんど変更が必要ないかもしれない。両者の結びつきを緩やかに切り離しておきたいというのは、魅力的な方針である。これは、チームが、データベースをモデルと同期した状態に保つのに失敗した場合に、意図せず採用されていることも多い。この分離が意識的に選択された場合には、すっきりしたデータベーススキーマができるかもしれない。すでに古くなったオブジェクトモデルに合わせようと妥協を重ねて、不格好になったりせずに済むからだ。

第7章

言語を使用する：応用例

直前の3つの章では、モデルの詳細に磨きをかけつつ、堅固なモデル駆動設計を維持するためのパターンランゲージを紹介した。これまで挙げてきた例では、パターンはだいたい一度に1つ適用されていたが、実際のプロジェクトではこれらのパターンを組み合わせなければならない。本章では、入り組んだ例を1つ取り上げる（もちろん、それでも現実のプロジェクトよりはるかに単純ではあるが）。この例では、モデルと設計が改良されていく様子を、段階を追いながら見ていく。その際には、仮想的なチームを想定し、要求と実装の問題に対処しながらモデル駆動設計を開発していくものとする。そして、その過程で生じるさまざまな問題を示し、また、第2部のパターンによってそれがどう解決されるのかについて見ていく。

貨物輸送システムを導入する

我々は貨物輸送会社のために新しいソフトウェアを開発中である。最初の要求は、次の3つの基本的な機能だ。

1. 顧客貨物に対する主要な荷役を追跡する。
2. あらかじめ貨物を予約する。
3. 貨物が荷役の過程で所定の場所に到達した際に、自動的に請求書を顧客に送付する。

実際のプロジェクトでは、このモデルを明確にするまで多少の時間とイテレーションが必要になるだろう。本書の第3部では、その発見の過程を詳しく説明する。しかし、ここでは必要な概念がそれなりのかたちになっているモデルから始めることにして、設計を支えるために詳細に対して加えられる微調整に集中する。

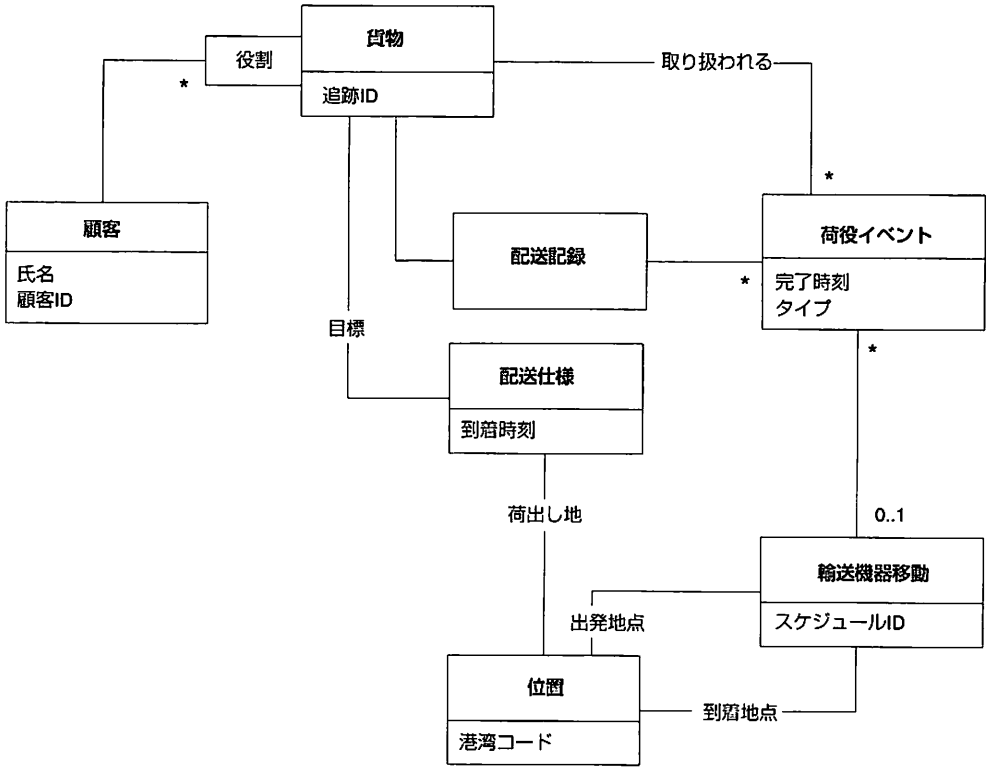


図7.1 輸送ドメインのモデルを表しているクラス図

このモデルはドメインに関する知識を体系化しており、チームに対して言語を提供する。これによって、次のように言えるようになる。

「1つの**貨物**には複数の**顧客**が関係していて、それぞれが異なる**役割**を果たしている。」
「**貨物**配送の**目標**が**定義される**。」
「**仕様**を満たした一連の**輸送機器移動**によって、配送の**目標**が実現される。」

モデルの各オブジェクトには明確な意味がある。

荷役イベントは貨物に関して行われる個別のアクションであり、**貨物**の船への積み込みや通関手続きなどがある。このクラスはおそらく、荷積みや荷下ろし、荷受人による請求など、さまざまな出来事からなる階層に作り上げられるだろう。

配送仕様は配送の目標を定義するものであり、少なくとも、荷出し地と到着日時を含んでいるであろうが、それより複雑でも構わない。このクラスは仕様パターンに従っている(第9章参照)。

この責務は**貨物**オブジェクトが引き受けることもできたが、**配送仕様**として抽象化する

ことには、少なくとも次の3つの利点がある。

1. **配送仕様**がないと、配送目標を指定する属性と関連すべての詳細な意味に関して、**貨物**オブジェクトが責任を持つことになる。これは**貨物**オブジェクトを混乱させ、理解したり変更したりするのを困難にする。
2. この抽象化を行うことで、モデル全体を説明する際に、詳細を容易かつ安全に隠せるようになる。例えば、**配送仕様**に別の条件もカプセル化されているかもしれないが、このレベルまで詳細な図であれば、条件の違いを見えるようにする必要はない。この図が読者に伝えているのは、配送の仕様があるということと、その詳細について考えるのは重要でないということである（実際に、後であっさりと変更される可能性もある）。
3. このモデルの方が、表現力が豊かになっている。**配送仕様**を追加することによって、**貨物**を配送する具体的な手段が決まっていなくても、**配送仕様**で設定された目標を達成しなければならないということが、明示的に述べられるのだ。

役割は、輸送において**顧客**が果たすさまざまな要素を区別する。「荷送人」がいて、「荷受人」がいて、「支払人」がいるという具合である。特定の**貨物**に対して所定の役割を果たせる**顧客**は1人だけなので、関連は多対多ではなく、限定つきの多対一になる。**役割**は、わかりやすく文字列として実装することもあるし、他のふるまいが必要ならば、クラスにすることもできる。

輸送機器移動は、特定の**輸送機器**（トラックや船など）によって実現される、ある**位置**から他の**位置**への特定の移動を表す。**貨物**は、**輸送機器**に荷積みされることにより、**輸送機器移動**が1回か、もしくは複数回行われる間、ある場所から別の場所へ移動できる。

配送記録は、実際に**貨物**に起こったことを反映しており、目標を記述する**配送仕様**とは対照的だ。**配送記録**オブジェクトは、**貨物**の現在の**位置**を割り出せるが、これは最後に行われた荷積みや荷下ろしと、それに対応する**輸送機器移動**の荷出し地を解析することで実現される。成功した配送は、**配送仕様**の目標を達成した**配送記録**で終わる。

ここまでいくつか要件を記述してきたが、これらの要求を実現するために必要なすべての概念は、このモデルで示されている。ただし、オブジェクトの永続化や関連するオブジェクトの検索などについては、適切な仕組みがあるものと想定している。そういう実装の問題はこのモデルでは対処されていないが、設計には含まれていなければならない。

しっかりとした実装を組み立てるには、このモデルをもう少し明確かつ厳密にする必要がある。

普通は、モデルの改良と設計および実装が、イテレーティブな開発プロセスにおいて、連携して進められなければならないということを思い出して欲しい。しかし、この章では、説明を明確にするために、比較的成熟したモデルから始める。そして、モデルを変更する

きっかけとなるのは、構成要素パターンを採用した実践的な実装に対して、モデルを結びつける必要が生じた場合に限定する。

また、普通、モデルが設計をよりよく支えるように改良される際には、ドメインへの新しい洞察も反映するように改良されなければならない。しかし本章では、やはり説明を明確にするため、変更のきっかけは、モデルと設計を結びつけることに限定する。

ドメインを隔離する：アプリケーションの導入

ドメインの責務が、システムの他の部分が持つ責務と混ざらないようにするため、レイヤ化アーキテクチャを適用してドメイン層を区切ることにしよう。

深く分析しなくても、ユーザレベルのアプリケーション機能を3つ識別して、それを3つのアプリケーション層クラスに割り当てることができる。

1. 特定の貨物に対して行われた過去と現在の荷役にアクセスできる**追跡問い合わせ**
2. 新規の**貨物**を登録できるようにして、それに対してシステムの準備を整える**予約アプリケーション**
3. **貨物**に対して行われた各荷役を記録できる**イベント記録アプリケーション**（**追跡問い合わせ**で検索される情報を提供する）

これらのアプリケーションクラスは調停役（coordinator）である。調停役は自分で出した質問に対する答えを考え出してはならない。それはドメイン層の仕事である。

エンティティと値オブジェクトを区別する

各オブジェクトを順に検討しながら、追跡すべき同一性や、表現されている基本的な値を探す。まず、明快なケースを見た後に、よりあいまいなケースについて検討する。

顧客（Customer）

まずは、簡単なものから始めよう。**顧客**オブジェクトは個人や企業を表している。これは実体を持つという、実際に使われる意味でのエンティティだ。**顧客**オブジェクトには、ユーザにとって重要な同一性が明らかに存在するから、モデルにおいてもエンティティとして扱える。これをどう追跡するか？ 税金IDが適切となる場合もあるが、国際企業では使用できないだろう。この問いについては、ドメインエキスパートとの協議が必要だ。そこで、輸送会社のビジネスマンと議論したところ、この会社にはすでに顧客データベースがあり、そこでは最初の営業の時に、**顧客**それぞれにID番号が割り当てられることがわかった。このIDはすでに全社的に使用されているので、これから開発するソフトウェアで

もこのIDを使用することにより、すでに存在するシステムとの間で同一性を連続させることができる。なおこのIDは、最初は手入力することになる。

貨物（Cargo）

そっくりの木箱が2つあっても、それぞれ区別できなくてはならないから、**貨物**オブジェクトはエンティティである。実際、すべての輸送会社では貨物1つ1つに追跡IDを割り当てている。このIDは自動的に生成され、ユーザも見ることができ、この場合にはおそらく予約時に顧客に伝えられているだろう。

荷役イベント（Handling Event）と輸送機器移動（Carrier Movement）

こういう個々のイベントに留意するのは、そうすることで何が起きているかを追跡できるようになるからである。こうしたイベントは現実世界での出来事を反映していて、通常は置き換えることができないから、これらはエンティティである。各**輸送機器移動**は、輸送スケジュールから得られるコードによって識別される。

ドメインエキスパートと交わした他の議論によって明らかになったのは、**貨物**ID、完了時刻、タイプの組み合わせによって、**荷役イベント**が一意に特定できるということである。例えば、同じ**貨物**に対して、荷積みと荷下ろし両方が同時に行われることはない。

位置（Location）

2つの場所は、名前が同じであっても同一のものではない。緯度と経度から一意なキーが得られるが、あまり実用的ではないだろう。そういう計測値は、このシステムのほとんどの目的にとって関心を引くものではないし、導入したら、かなり複雑になると思われるからだ。それよりも、**位置**は何らかの地理的なモデルの一部となるだろう。その地理的なモデルは、輸送経路やその他ドメイン特有の関心事に従って、場所同士を結びつける。したがって、位置を識別するためには、内部で使用される任意の自動生成識別子を用いれば十分であろう。

配送記録（Delivery History）

これには注意が必要だ。**配送記録**は交換することができないから、エンティティである。しかし、**配送記録**には**貨物**と一対一の関係があるので、実は独自の同一性を持っていない。同一性は、所有者である**貨物**から借用するのである。このことは、集約をモデル化すると一層明確になる。

配送仕様（Delivery Specification）

これは**貨物**の目標を表しているが、この抽象は**貨物**に依存しない。実際に表現しているのは、ある配送記録の状態に対する仮説である。ある**貨物**と結びついた**配送記録**は、最終的にはその**貨物**と結びついた**配送仕様**を満たすことが望ましい。同じ場所に向かっている2つの**貨物**がある場合、同一の**配送仕様**を共有することはできるが、**配送記録**を共有するこ

とはできない。2つの記録が同じように開始されていても（記録が空）、やはり共有することはできないのだ。**配送仕様**は値オブジェクトである。

役割とその他の属性

役割は、それによって限定される関連について何らかの説明をしているが、履歴や連続性は持たない。これは値オブジェクトであり、いろいろな**貨物**や**顧客**の関連で共有できる。

タイムスタンプや名前などといった他の属性は値オブジェクトである。

輸送ドメインの関連を設計する

オリジナルの図に描かれている関連は、どれも辿れる方向を指定していないが、双方向の関連は、設計上問題になる。また、関連を辿れる方向は、ドメインに対する洞察をとらえていることが多く、モデル自体を深めもする。

発送した**貨物**すべてに対して**顧客**が直接参照を持っていると、**顧客**が長年のリピータであった場合に扱いにくくなってしまう。しかも、**顧客**の概念は**貨物**に固有のものではない。巨大なシステムにおいては、**顧客**には多くのオブジェクトに対して果たす役割があるだろう。そういう特定の責務からは解放しておくのが一番である。**顧客**から**貨物**を検索する機能が必要ならば、データベースクエリによって実現できる。この問題は、本章後半のリポジトリに関する節で再考する。

このアプリケーションが船の目録を追跡するなら、**輸送機器移動**から**荷役イベント**へ辿ることが重要になるだろう。しかし、我々のビジネスが追跡しなければならないのは**貨物**だけである。**荷役イベント**から**輸送機器移動**への関連のみ辿れるようにすることで、我々のビジネスに対する理解をとらえることができる。こうすることで、実装は1つのオブジェクトへの参照へと単純化されるが、それは複数の方向を認めなかったおかげである。

残りの決定の論理的根拠は、図7.2で説明している。

このモデルには循環参照が1つある。**貨物**は**配送記録**について知っており、**配送記録**は一連の**荷役イベント**を保持している。この**荷役イベント**が、今度は**貨物**を参照するのである。循環参照は論理的に多くのドメインに存在し、設計上必要なこともあるが、維持管理しにくいものだ。実装上の選択としては、同期させなければならない同じ情報を2か所で持たれないようにすることで、うまくいくかもしれない。だが、ここでつくる最初のプロトタイプでは、単純だが脆弱な実装をすることにしよう。すなわち、**配送記録**に**荷役イベント**を含んだ**リスト**オブジェクトを与えることにする。しかし、どこかでコレクションをやめ、**貨物**をキーとしてデータベース検索を行うことにするだろう。この議論は、リポジトリの選択において再度取り上げる。さて、記録を参照するような問い合わせが比較的ま

れであれば、データベースを検索する実装によって性能は向上し、保守は単純になって、**荷役イベント**を追加する際のオーバーヘッドも減るに違いない。だが、この問い合わせが頻繁な場合には、直接参照を管理する方が適している。こうした設計におけるトレードオフによって、実装の単純さと性能との間でバランスが取られるのだ。どちらの場合も、モデルは変わらず、循環と双方向の関連を含んでいる。

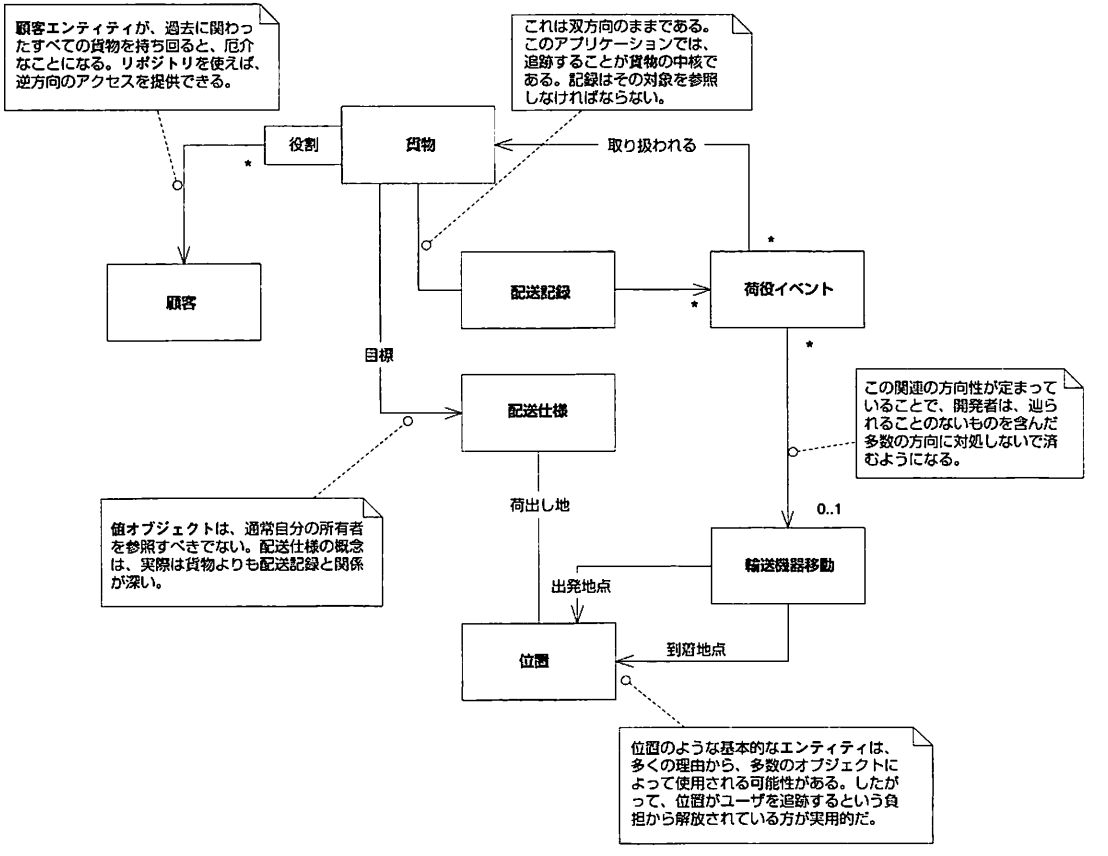


図7.2 辿る方向が制限されている関連もある

集約の境界

顧客、**位置**、**輸送機器移動**には、独自の同一性があり、多くの**貨物**に共有されているので、それぞれ独自に集約を持ち、そのルートにならなければならない。こうした集約に含まれているのは、これらエンティティの属性と、おそらくここで議論しているよりも詳細なレベルのオブジェクトであろう。**貨物**も明らかに集約ルートだが、どこに境界を引くかを決めるには検討が必要だ。

貨物の集約は、特定の**貨物**がなければ存在しないであろうものを、すべて集められるだろうから、**配送記録**、**配送仕様**、**荷役イベント**を含むことになるだろう。これは**配送記録**によく当てはまる。**貨物**を必要とせずに、**配送記録**を直接探す人はいないと思えるからだ。直接的なグローバルアクセスが必要なく、同一性が実際には**貨物**から派生するだけなので、**配送記録**は**貨物**の境界内にうまく収まり、ルートである必要がない。**配送仕様**は値オブジェクトなので、**貨物**の集約に含めても問題はない。

だが、**荷役イベント**に関しては話が別である。以前に、**荷役イベント**を検索する2つのデータベースクエリの可能性について考察した。1つは、コレクションの代わりとして**配送記録**に関する**荷役イベント**を見つけるというもので、**貨物**集約内で閉じることになるだろう。もう1つが特定の**輸送機器移動**に向けて荷積みと準備を行うすべての業務を見つけるのに使用されるであろうものだ。後者において、**貨物**を**取り扱う**アクティビティは、**貨物**自体から切り離して考えたとしても意味を持つだろう。したがって、**荷役イベント**は独自の集約のルートであるべきだ。

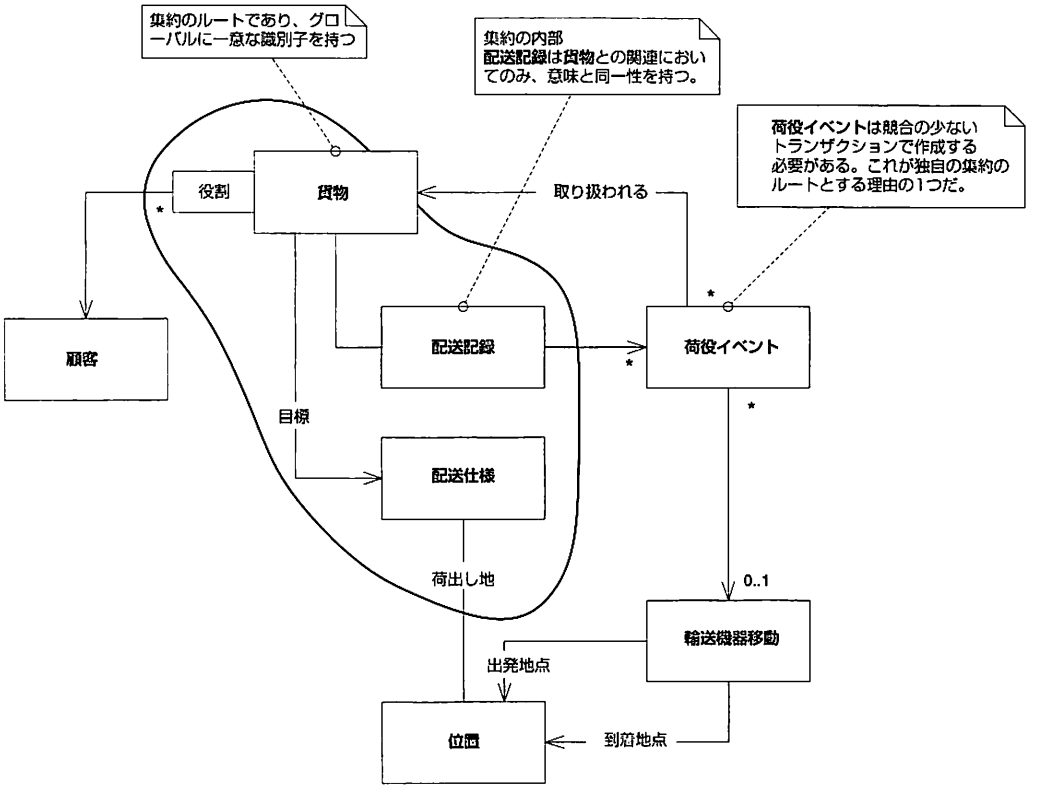


図7.3 モデルに対して引かれた**集約境界**（注：引かれた境界の外部にある**エンティティ**は、独自の**集約**のルートであることを暗に示している）

リポジトリを選択する

この設計にある、集約のルートとなるエンティティは5つなので、リポジトリを検討する際には、これらに限定できる。他のオブジェクトはどれも、リポジトリを持つことが認められないからだ。

これらの候補のうち、実際にリポジトリを持つべきものを決めるためには、アプリケーションの要求に立ち返らなければならない。**予約アプリケーション**によって予約を取るために、ユーザは、さまざまな役割を果たす**顧客**（荷送人、荷受人など）を選択する必要がある。そこで、**顧客リポジトリ**が必要だ。また、**貨物**の荷出し地として指定するための**位置**も見つけなければならないから、**位置リポジトリ**を作成する。

アクティビティ記録アプリケーションは、ある**貨物**が積まれた**輸送機器移動**をユーザが

検索できるようにしなければならないため、**輸送機器移動リポジトリ**が必要である。また、このユーザはどの**貨物**が荷積みされたかについてもシステムに伝えなければならないから、**貨物リポジトリ**も必要になる。

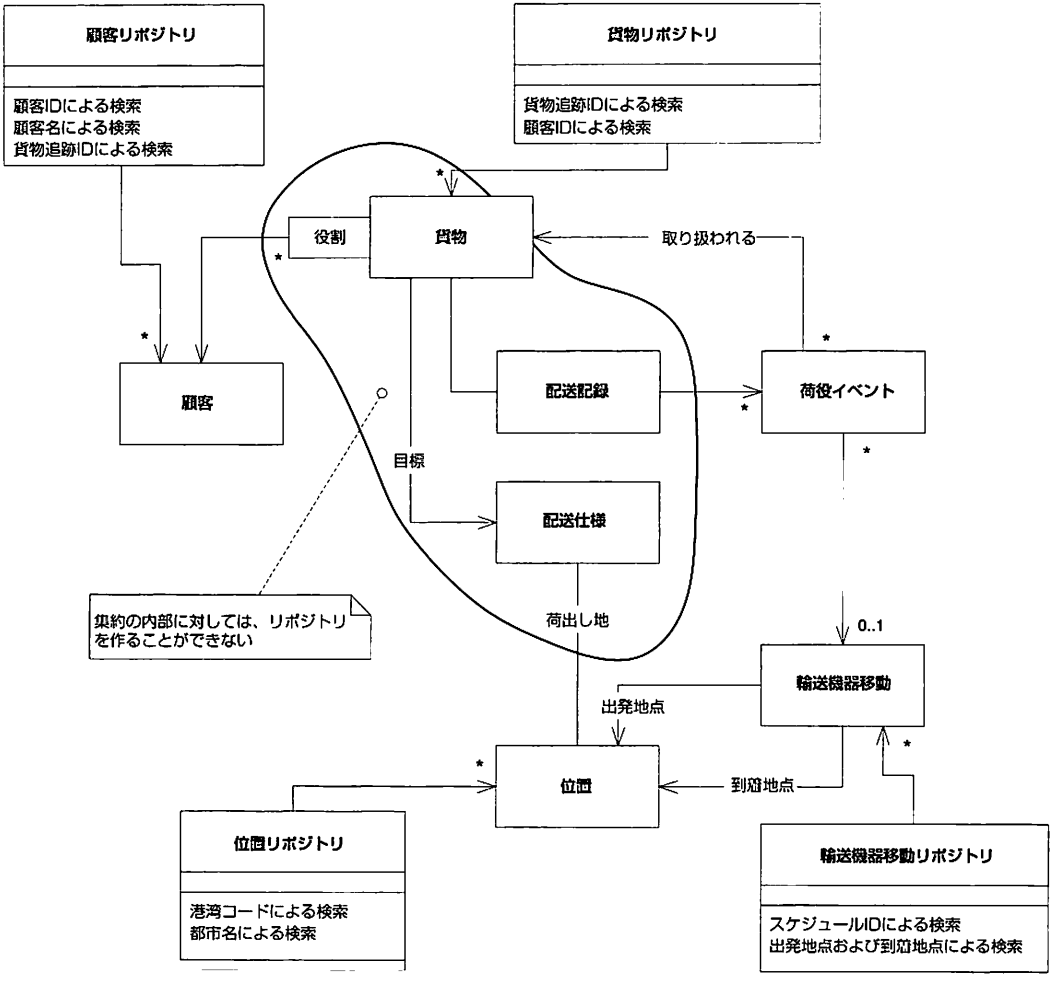


図7.4 **リポジトリ**により、選ばれた**集約**ルートへのアクセスが与えられる

これまでで**荷役イベントリポジトリ**がないのは、最初のイテレーションでは**配送記録**との関連をコレクションとして実装することにしたからであるし、**輸送機器移動**に何が荷積みされているかを調べるというアプリケーションの要求がないからでもある。これらの理由はどちらも変わる可能性があり、そうなった場合にはリポジトリを追加することになるだろう。

シナリオをウォークスルーする

これまでの決定をすべてクロスチェックするためには、常に段階を追ってシナリオを見ながら、アプリケーションの問題を効果的に解決できることを確かめなければならない。

サンプルアプリケーションの機能：貨物の荷出し地を変更する

時折、「間違えました！ 貨物はハッケンサックへ送るように頼みましたが、本当はホーボーケンで必要なのです†」と、**顧客**が電話してくることがある。こうした要求に応えるために我々はここにいるのだから、システムはこの変更に備えなければならない。

配送仕様は値オブジェクトなので、最も単純に実現するなら、今あるものを廃棄して新しいオブジェクトを取得し、**貨物**のセッタメソッドを使用して、新しい**配送仕様**で古いものを置き換えればよい。

サンプルアプリケーションの機能：リピータへの対応

ユーザとしては、同じ**顧客**によって繰り返される予約は似ている傾向があるので、新規の**貨物**のプロトタイプとして古い**貨物**を使用したい。そのため、アプリケーションでは、ユーザがリポジトリから**貨物**を検索した後で、選んだ**貨物**を基に新規の**貨物**を生成するコマンドを選択できるようにする。これは、プロトタイプパターン（PROTOTYPE）（Gamma 他 1995）を使用して設計することにしよう。

貨物はエンティティであり、集約のルートでもある。したがって、コピーは慎重にしなければならない。つまり、集約の境界に囲まれたオブジェクトや属性それぞれに、何が起こるべきかを考える必要があるのだ。1つずつ見ていこう。

- **配送記録**：古い記録は適用できないから、新しい空の記録を生成すべきだ。集約の境界内にあるエンティティに関しては、こうするのが普通である。
- **顧客の役割**：役割をキーとした**顧客**を参照する**マップ**（または別のコレクション）は、キーも含めてコピーしなければならない。新規の輸送でも、顧客は以前と同じ役割を果たす可能性が高いからだ。しかし、**顧客**オブジェクト自体をコピーしないように注意しなければならない。最終的に、旧**貨物**オブジェクトが参照していたのと同じ**顧客**オブジェクトを参照しなければならない。**顧客**オブジェクトは集約の境界外にある工ンティティだからだ。
- **追跡ID**：新規の**貨物**を一から生成するのと同じところから、新しい**追跡ID**を提供しなければならない。

† 訳注：ハッケンサック（Hackensack）と、ホーボーケン（Hoboken）は、いずれもアメリカ、ニュージャージー州にある地名。

貨物集約の境界内にあるものはすべてコピーし、そのコピーを一部修正したが、**集約の境界外のものに対しては、全く影響を及ぼしていない**ことに注意して欲しい。

オブジェクトの生成

貨物用のファクトリとコンストラクタ

貨物用の手の込んだファクトリがあったり、「リピータ」のシナリオのように別の**貨物**をファクトリとして使用したりする場合でも、プリミティブなコンストラクタは必要である。そのコンストラクタを用いて生成したいのは、不変条件を満たしているオブジェクトか、あるいはそれがエンティティの場合には、少なくとも同一性が侵されていないオブジェクトである。

これらの決定を踏まえると、次のようなファクトリメソッドを**貨物**上に作成することになるだろう。

```
public Cargo copyPrototype(String newTrackingID)
```

あるいは、次のような独立したファクトリにメソッドを作成してもよい。

```
public Cargo newCargo(Cargo prototype, String newTrackingID)
```

独立したファクトリは、新規の**貨物**のために新しい（自動生成される）IDを取得するプロセスをカプセル化することもできる。その場合、必要な引数は1つだけとなる。

```
public Cargo newCargo(Cargo prototype)
```

これらのファクトリが戻す結果はどれも同じ、**配送記録**が空で**配送仕様**がNULLの**貨物**である。

貨物と**配送記録**との間にある、この双方向の関連が意味するのは、**貨物**も**配送記録**も相手を参照していなければ完全ではないということなので、両者は一緒に生成しなければならない。**貨物**が、**配送記録**を含んだ集約のルートであることを忘れないで欲しい。だからこそ、**貨物**のコンストラクタやファクトリに**配送記録**を生成させることができる。**配送記録**のコンストラクタは引数として**貨物**を取る。その結果は、次のようになる。

```
public Cargo(String id) {
   trackingID = id;
   deliveryHistory = new DeliveryHistory(this);
   customerRoles = new HashMap();
}
```

このコンストラクタによって生成されるのは、自らを参照し返す新しい**配送記録**を持った新規の**貨物**である。**配送記録**のコンストラクタは、それを含む集約のルート、すなわち**貨物**によってのみ使用され、それにより**貨物**の構造がカプセル化される。

荷役イベントを追加する

現実の世界で**貨物**が取り扱われるたびに何人かのユーザが、**イベント記録アプリケーション**を使って**荷役イベント**を入力する。

どのクラスにも、プリミティブなコンストラクタがなければならない。**荷役イベント**はエンティティなので、その同一性を定義する属性はすべて、コンストラクタに渡さなければならない。前述した通り、**荷役イベント**を一意に識別するのは、**貨物**のIDと完了時刻およびイベントタイプの組み合わせである。**荷役イベント**にあるそれ以外の属性は**輸送機器移動**への関連だけだが、**荷役イベント**のタイプによっては、それを持たないものすらある。有効な**荷役イベント**を生成する基本的なコンストラクタは、次のようになる。

```
public HandlingEvent(Cargo c, String eventType, Date timeStamp) {
   handled = c;
   type = eventType;
   completionTime = timeStamp;
}
```

エンティティの識別に関係ない属性は、通常、後から追加することもできる。だがこの場合は、**荷役イベント**のすべての属性が最初のトランザクションで設定され、その後は（おそらくデータ入力のエラーを訂正する場合を除いて）変更されることがない。このため、より便利にしてクライアントコードの表現力をさらに豊かにするには、単純なファクトリメソッドをイベントタイプ毎に**荷役イベント**に追加し、必要な引数をすべて取るようにすればよい。例えば、「荷積みイベント」には**輸送機器移動**が含まれる。

```
/**
 * 新規荷積み.
 * @param c 貨物
 * @param loadedOnto 荷積み対象（輸送機器移動）
 * @param timeStamp タイムスタンプ
 * @return 荷役イベント
 */
public static HandlingEvent newLoading(
   Cargo c, CarrierMovement loadedOnto, Date timeStamp) {
      HandlingEvent result =
         new HandlingEvent(c, LOADING_EVENT, timeStamp);
      result.setCarrierMovement(loadedOnto);
      return result;
}
```

このモデルにおける**荷役イベント**は抽象であり、さまざまなかたちで特化した荷役イベントクラスをカプセル化している可能性がある。こうした**荷役イベント**クラスは、荷積みと荷下ろしにはじまり、梱包や保管、さらに、輸送機器とは無関係なその他の活動にまで及ぶ。これらは複数のサブクラスとして実装されるかもしれないし、複雑な初期化を伴うかもしれない。あるいは、その両方のこともある。基底クラス（**荷役イベント**）に各イベントタイプのファクトリメソッドを追加することにより、インスタンス生成が抽象化され、クライアントは実装を知らずに済むようになる。ファクトリの責務は、どのクラスがインスタンス化されるべきかと、それをどうやって初期化するかを理解していることである。

残念ながら、話はそれほど単純ではない。**貨物**から**配送記録**に向かい、そこから**荷役イベント**を経て、**貨物**に戻るという参照の循環によって、インスタンス生成は複雑になる。**配送記録**は**貨物**に関係する**荷役イベント**のコレクションを保持しており、新規のオブジェクトは、トランザクションの一部としてこのコレクションに追加されなければならない。この逆の参照が生成されないと、オブジェクトの間に矛盾が生じる。

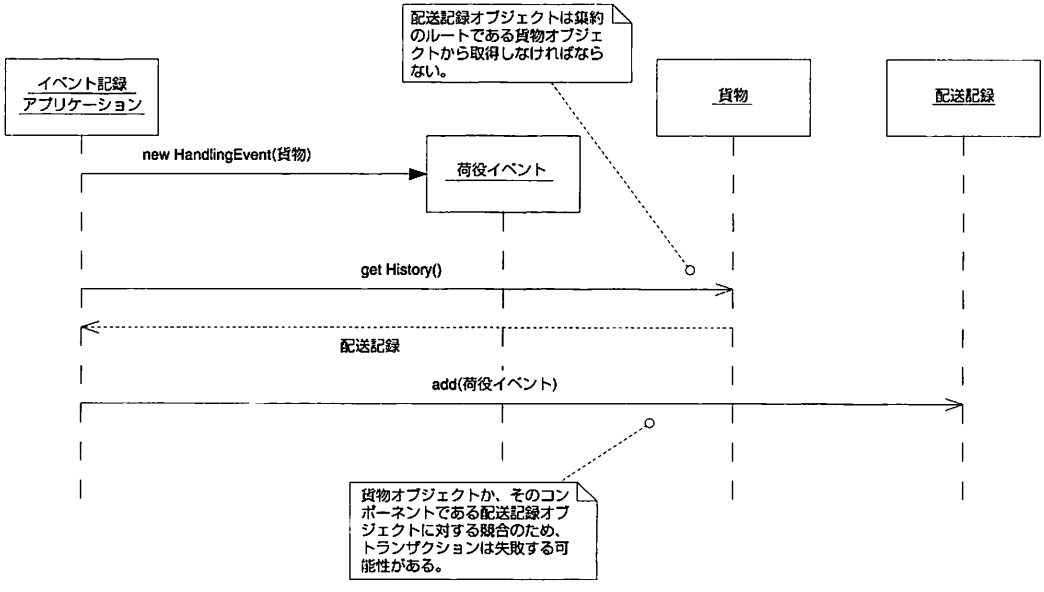


図7.5 荷役イベントを追加するには配送記録へ挿入する必要がある

逆の参照の生成は、ファクトリでカプセル化（および、それが属しているドメイン層で保持）できるが、ここではそうしたぎこちない相互作用をすべて取り除く、もう1つの設計を見ていく。

リファクタリングのために立ち止まる：貨物集約についてのもう１つの設計

モデリングと設計は、絶えず前進するだけのプロセスではない。頻繁にリファクタリングを行い、モデルと設計を改善する新しい洞察を活用しなければ、立ち往生してしまう。

これまでのところ、この設計は間違いなく機能し、モデルを反映してもいるのだが、扱いにくい側面が2つ3つある。設計を始めた時には重要と思えなかった問題が、困ったものになり始めているのだ。そのうちの1つに立ち戻り、後知恵を活かして、こちらに有利になるように設計のカードを積み込んでみよう。

荷役イベントを追加する時に**配送記録**を更新しなければならないことから、**貨物**集約がそのトランザクションに巻き込まれる。他のユーザが同時に**貨物**を修正していたら、**荷役イベント**トランザクションは、失敗するか遅れる可能性がある。**荷役イベント**の入力は、迅速かつ単純でなければならない操作だから、アプリケーションの要求で重要とされるのは、**競合することなく荷役イベント**を入力できるということである。このために、別の設計を検討せざるを得なくなる。

配送記録が持つ**荷役イベント**のコレクションをクエリで置き換えることにより、**荷役イベント**をルートとした集約の外部に整合性の問題を引き起こすことなく、**荷役イベント**自体を追加できるようになる。この変更により、前述したトランザクションは競合することなく完了できるようになる。**荷役イベント**が、入力されることが多く、問い合わせが比較的少なければ、この設計はより効果的となる。実のところ、下層にある技術が関係データベースであれば、コレクションを模すために、いずれにしても見えないところでクエリが使用されただろう。コレクションではなくクエリを使用することで、**貨物**と**荷役イベント**との間にある循環参照において一貫性を保つのも容易になる。

こうした問い合わせの責務を負わせるため、**荷役イベント**用にリポジトリを追加しよう。この**荷役イベント**リポジトリは、特定の**貨物**に関係した**イベント**に対するクエリをサポートする。さらに、このリポジトリは特定の質問に効率よく答えるように最適化されたクエリを提供することもできる。例えば、頻繁にアクセスされるのが**配送記録**に対して最後に行われた荷積みか荷下ろしを検索するクエリで、それが**貨物**の現在の状態を推察するためであれば、関係する**荷役イベント**だけを戻すようにクエリを工夫することもできる。特定の**輸送機器移動**に荷積みされた**貨物**をすべて見つけるクエリが必要であれば、簡単に追加することもできる。

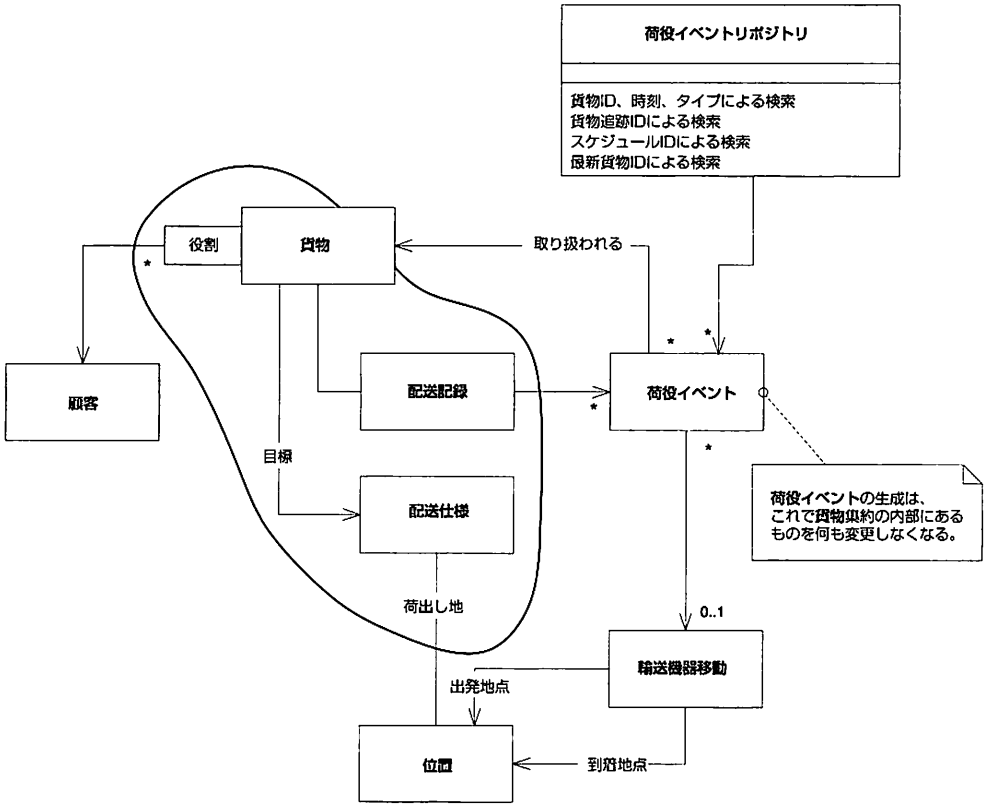


図7.6 配送記録が持つ荷役イベントのコレクションをクエリとして実装することで、荷役イベントの挿入が単純になり、貨物集約との競合からも解放される

これで、**配送記録**には永続化された状態がなくなる。この時点では、**配送記録**を保持しておく必要が全くない。何らかの質問に答えるために必要になれば、いつでも**配送記録**そのものを導き出せるだろう。これは、エンティティが繰り返し生成されても、同じ**貨物**オブジェクトとの間に関連があるため、何度も生成されるインスタンスをつなぐ連続性が維持されているからである。

循環参照は、生成するのも維持するのも、もはや困難ではない。**貨物ファクトリ**は単純化され、もう新しいインスタンスに空の**配送記録**をつけ加えることはない。データベースの領域は少し削減することができ、永続化されたオブジェクトの実際の数は、大幅に減らせるかもしれない。オブジェクトの数は、オブジェクトデータベースによっては限られたリソースである。このシステムの一般的な利用パターンで、**貨物**が到着するまでにユーザがその状態を問い合わせることがめったにないのなら、多くの不要な作業も一緒にやめる

ことができる。

一方、オブジェクトデータベースを使用しているなら、関連や明示的なコレクションを辿る方が、リポジトリによる問い合わせよりも、おそらくはるかに高速だろう。アクセスのパターンとして、記録をすべて一覧化することがよくあり、逆に最新の履歴に絞った問い合わせがあまりないのであれば、性能のトレードオフでは、明示的なコレクションの方が好ましくなるかもしれない。ただ、忘れてはならないことだが、この追加機能（「この**輸送機器移動**は何を載せているか？」）はまだ要求されておらず、今後も要求されないかもしれないので、その選択肢のためにあまり多くのことをしたくはない。

この種の代替案や設計のトレードオフはあらゆるところにあり、この単純化した小さなシステムだけでも多くの例を考え出せるだろう。だが、重要なのは、こうした自由が許されるのが、同一モデルの中だけに限られるということだ。値オブジェクト、エンティティ、そしてそれらの集約を、このようにモデル化することによって、こうした設計上の変更が与える影響を抑えてきた。例えば、ここに挙げた例でも、すべての変更は**貨物**の集約内にカプセル化されている。また、**荷役イベントリポジトリ**を追加する必要もあったが、それによって**荷役イベント**自体の再設計が必要になることはなかった（ただし、リポジトリフレームワークの詳細次第では、実装の変更を伴うかもしれない）。

輸送モデルにおけるモジュール

これまではあまり多くのオブジェクトを取り上げなかったため、モジュール性は問題にならなかった。今度は輸送モデルにおける、もう少し大きい部分を取り上げ（もちろん、依然として単純化してあるが）、それをモジュールへ構成することで、モデルが影響を受けるところを見ていこう。

図7.7が示しているのは、本書を熱心に読んだ架空の読者によって、きれいに分割されたモデルである。この図は、第5章で提起した、インフラストラクチャ駆動のパッケージングが持つ問題を示すバリエーションの1つであるといえる。ここでオブジェクトは、それぞれが従っているパターンによってグループ化されている。結果として、概念的にはほとんど関係ない（低凝集）オブジェクトが一緒に詰め込まれ、すべてのモジュール間に関連が無秩序に走っている（高結合）。このパッケージは、あるストーリーを伝えてはいるが、それは輸送に関するものではなく、開発者がその時に読み取ったストーリーなのである。

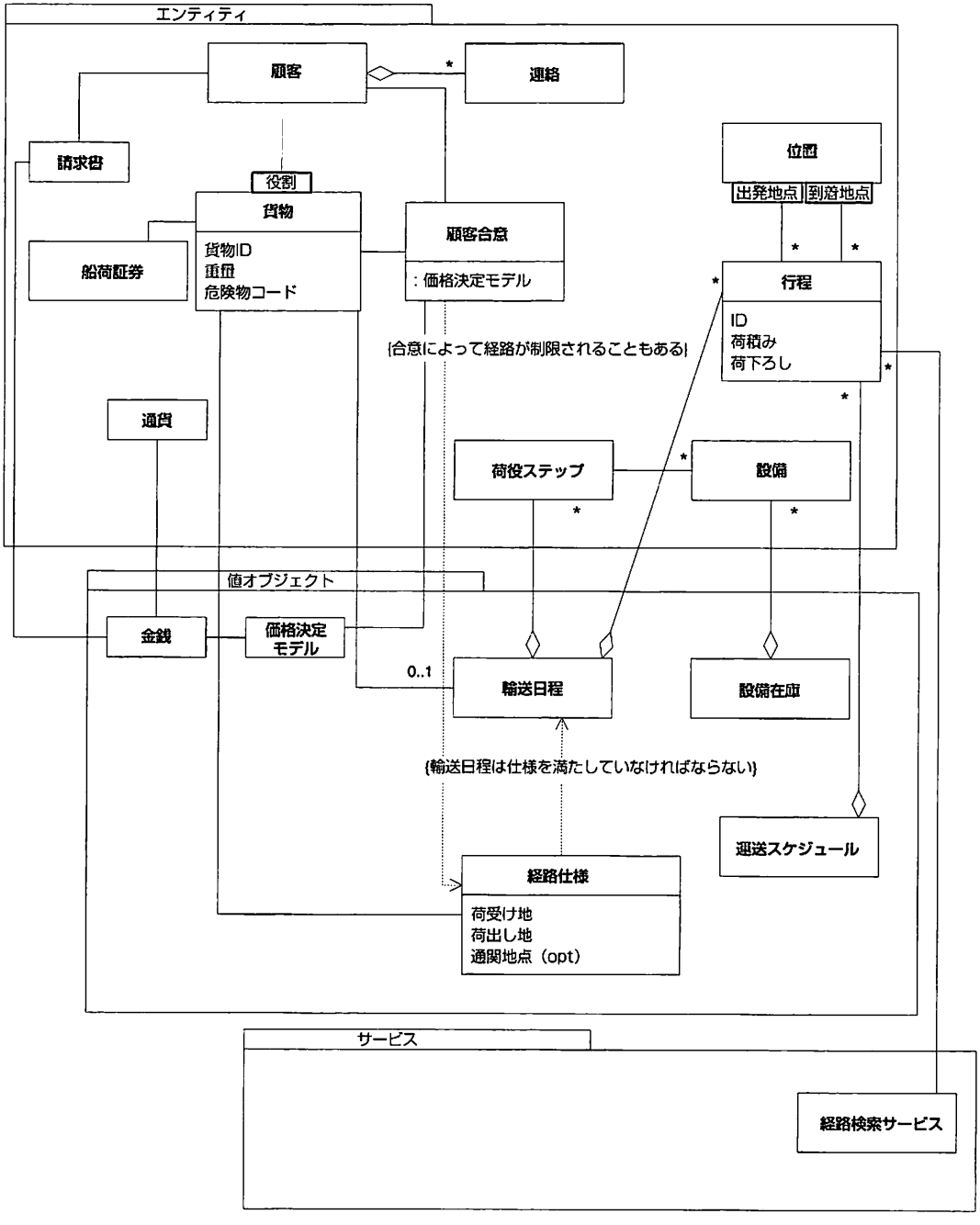


図7.7 これらの**モジュール**は、ドメインについての知識を伝えていない

パターンに従って分割するのは明らかな間違いに見えるかもしれない。だが、永続化されたオブジェクトを一時的なオブジェクトと分離することを始めとする、オブジェクトの意味に基づかないその他の整然としたスキームと比べた時に、パターンに従った分割の方が非常識だということもない。

我々がすべきことは、凝集度の高い概念を探し、プロジェクトにおいて他の人々に伝えたいことに集中することなのだ。より小規模なモデリングの決定におけるのと同じく、その方法は多くある。素直に分割したものを図7.8に示す。

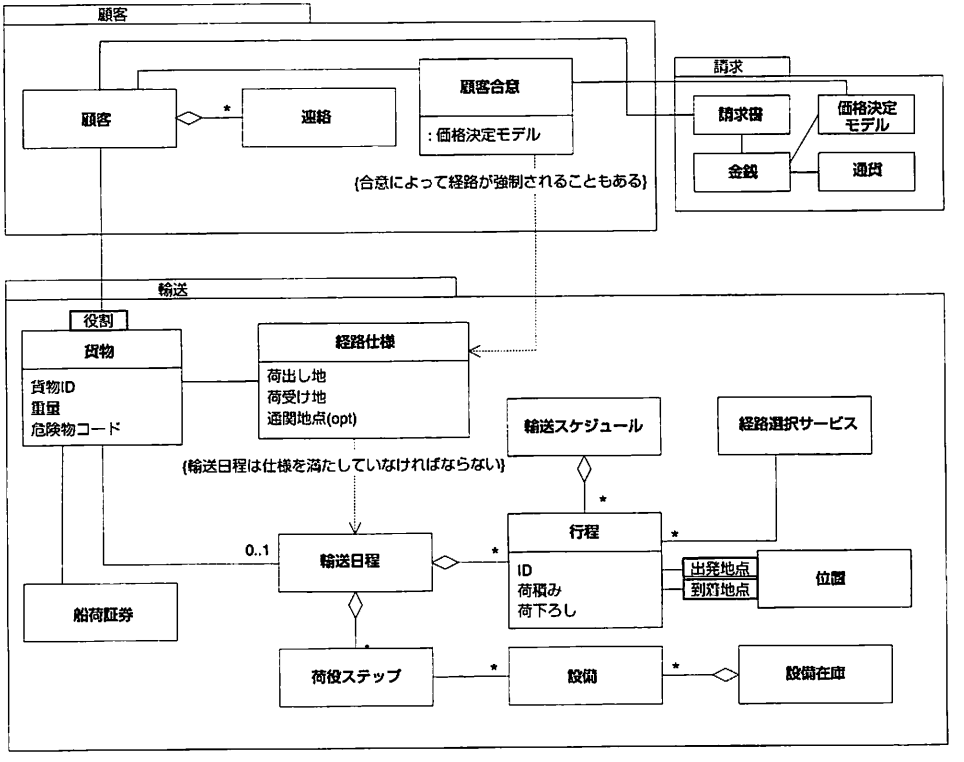


図7.8 大まかなドメイン概念に基づいたモジュール

図7.8にあるモジュールの名称は、チームの言語に貢献している。わが社は確かに**顧客**への**輸送**を行っており、したがって、顧客に**請求**できる。営業販売の担当社員が**顧客**に対応して、合意を取りつける。業務部門が**輸送**を行い、貨物を指定された荷出し地へ届ける。事務部門は**請求**処理を引き受け、**顧客**と合意した価格に従って請求書を送付する。これらのモジュール一式を使って、こうしたストーリーを語ることもできるのだ。

この直観的な分析は、もちろん後に続くイテレーションで改良されるかもしれないし、

すべて置き換えられるかもしれない。それでも、現段階ではモデル駆動設計を助け、ユビキタス言語に貢献している。

新機能を導入する：配分チェック

ここまでで、初期の要求とモデルを片づけてきた。今度は初めて大掛かりな新機能を追加することにする。

架空の輸送会社の販売部門は、顧客との関係や販売計画などを管理するのに別のソフトウェアを使用している。その機能の1つは、特定のタイプの貨物をどれくらい予約するべきかについて配分できるようにすることで、イールドマネジメントをサポートしている。この配分は、商品のタイプや荷受け地と荷出し地、およびカテゴリ名として入力できるようなその他の要素に基づいて行われる。これによって、貨物の各タイプをどのくらい売らなければならないかという目標が設定される。ビジネス上でより収益の高いタイプの貨物が、それほどでもない貨物で一杯になったために積み込めないということが起きないようにしながら、同時に過少予約（輸送能力をすべて活用していないこと）や、過度のオーバーブッキング（貨物の取り消しが頻発して、顧客との関係を損なうことにつながる）を回避する。

さて、販売部門は、この機能を予約システムに統合したいと考えている。予約が来たら、そうした配分に照らして受けつけるかどうかを確認したいのだ。

必要な情報は2か所にあるが、どちらも**予約アプリケーション**で問い合わせて、要求された予約を受けつけることも、拒否することも、できるようにしなければならない。全般的な情報の流れの概要は次のようになる。

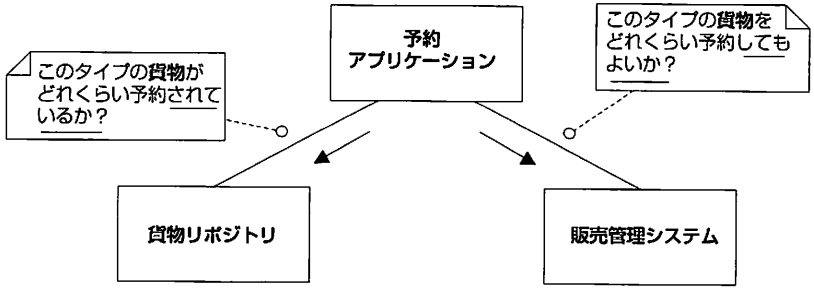


図7.9 予約アプリケーションは、販売管理システムと、こちらのドメインの**リポジトリ**から取得した情報を使用しなければならない

2つのシステムを接続する

販売管理システムは、ここで取り組んでいるものと同じモデルを念頭に書かれていない。2つのシステムが直接相互作用するなら、**予約アプリケーション**は販売管理システムの設計を受け入れなければならないが、それでは明確なモデル駆動設計を維持することが困難になり、ユビキタス言語に混乱をきたしてしまう。代わりに、もう1つクラスを作成して、我々のモデルと**販売管理システム**の言語との間の変換をさせよう。これは汎用的な変換機構にはならない。このクラスは、我々のアプリケーションが必要とする機能だけをこちらに見せ、それらの機能をこちらのドメインモデルの観点から再び抽象化する。このクラスは、腐敗防止層（第14章で説明）として機能するのだ。

これは**販売管理システム**に対するインタフェースだから、最初は「**販売管理インタフェース**」などと呼ぼうと考えるかもしれない。だがそれでは、言語を活用してより役立つ輪郭に沿って問題をとらえ直す機会を逃してしまう。代わりに、他のシステムから取得する必要のある配分機能について、それぞれサービスを定義しよう。そのサービスを実装するクラスの名前は、こちらのシステムにおける責務を反映させて、「**配分チェックサービス**」としよう。

これ以外にも統合が必要となった際には（例えば、こちらの**顧客**リポジトリではなく、**販売管理システム**の顧客データベースを使用するなど）、その責務を果たすサービスと共に、別の変換サービスが作成されることになるだろう。**販売管理システムインタフェース**のような下位レベルのクラスを作成し、他のプログラムとの通信機構を処理させるのは、やはり便利かもしれないが、そのクラスは変換の責務は負わないだろう。また、**配分チェックサービス**の背後に隠されるので、ドメイン設計に出てくることもない。

モデルを強化する：ビジネスのセグメント化

2システムの相互作用について概略を述べたのだが、それでは「このタイプの**貨物**をどれくらい予約してもよいか？」という質問に答えるためには、どのようなインタフェースを提供すればよいだろうか？ 厄介なのは、**貨物**の「タイプ」とは何かを定義することだ。我々のドメインモデルでは、まだ**貨物**をカテゴリ分けしていないからだ。**販売管理システム**では、**貨物**タイプはカテゴリに関するキーワードの集まりにすぎないので、こちらで用いているタイプを、そのリストに順応させてもよい。あるいは、文字列のコレクションを引数として渡すこともできるだろう。だがそれでは、せっかくの機会をやはり逃してしまう。今回は、他のシステムのドメインを、こちら側の観点で抽象化し直すことができるのだ。ドメインモデルを豊かにして、貨物にカテゴリが存在するという知識に適応できるようにする必要がある。ドメインエキスパートとブレインストーミングして、この新しい概念を考え出さなければならない。

時には（第11章で議論するように）、分析パターンからモデリングに関する解決策が得られることもある。『アナリシスパターン』（Fowler 1996）には、この種の問題に取り組むパターンが記述されている。それが、エンタープライズセグメント（ENTERPRISE SEGMENT）だ。エンタープライズセグメントとは、ビジネスの分割方法を定義した次元軸の集まりである。それらの次元軸には、過去1か月といった時間的なものに加えて、輸送ビジネスに関してこれまで述べたものをすべて含めることができる。この概念を配分のモデルで使用することで、モデルはより表現力豊かになり、インタフェースはシンプルになる。「**エンタープライズセグメント**」と呼ばれるクラスが、これまで考えてきたドメインモデルと設計に値オブジェクトとして追加され、各**貨物**に対して設定される。

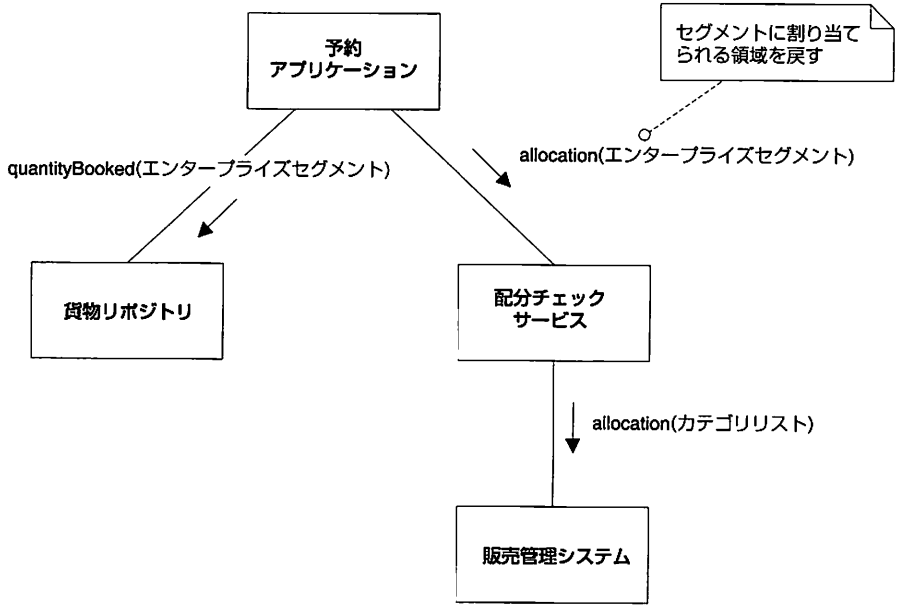


図7.10 配分チェックサービスは**腐敗防止層**として機能し、販売管理システムへのインタフェースの中から必要なものを選んで、こちらのドメインモデルの用語に基づくインタフェースを提供する

配分チェックサービスは、**エンタープライズセグメント**と外部システムのカテゴリ名との間で変換を行うことになる。**貨物リポジトリ**も、**エンタープライズセグメント**に基づいたクエリを提供しなければならない。どちらの場合も、**エンタープライズセグメント**オブジェクトと協調することによって、セグメントのカプセル化に違反したり、実装を複雑にしたりすることなく、操作を実行できるようになる（注意すべきは、**貨物リポジトリ**がク

エリに回答する際に、インスタンスのコレクションではなく、インスタンスの数を返していることである)。

この設計には、依然として次のような問題がある。

1. **予約アプリケーション**に対して、ルールの適用を任せている。このルールとはすなわち、「貨物を受けつけるのは、貨物の**エンタープライズセグメント**に割り当てた空間が、すでに予約済みの空間と、この新しい**貨物**の大きさを合わせたよりも大きい場合である」というものだ。しかし、ビジネスルールを遂行するのはドメインの責務だから、アプリケーション層で実行すべきではない。
2. **予約アプリケーション**が**エンタープライズセグメント**をどう取得するかが明確でない。

これらの責務はいずれも、**配分チェックサービス**に属するように見える。インタフェースを変更することで、これら2つのサービスを分離し、相互作用を明確で、明示的なものにできる。

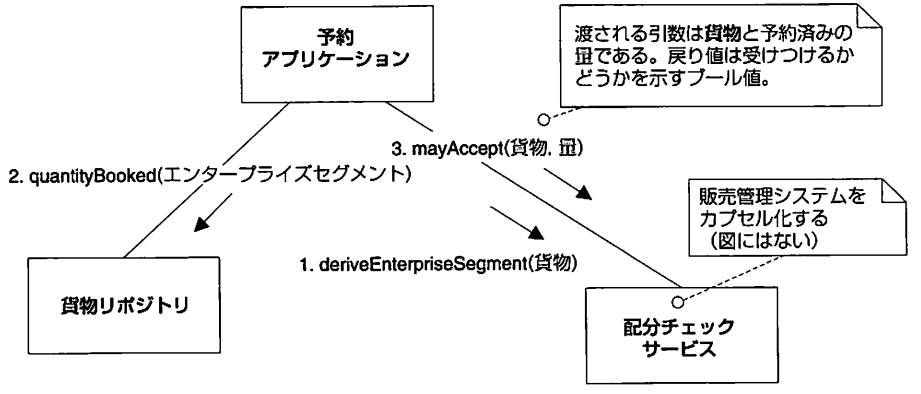


図7.11 予約アプリケーションから配分チェックサービスに移ったドメイン責務

この統合によって課される唯一かつ重大な制約は、**配分チェックサービス**がエンタープライズセグメントに変換できない次元軸を、**販売管理システム**は使用してはならないということである（エンタープライズセグメントパターンを適用しなくても、これと同じ制約によって、**販売管理システム**は**貨物リポジトリ**に対するクエリで使用できる次元軸しか使うことができない。このアプローチは実現可能だが、**販売管理システム**がドメインの他の部分にあふれ出してしまう。エンタープライズセグメントを用いた設計であれば、**貨物リポジトリ**は**エンタープライズセグメント**を処理するように設計するだけでよく、**販売管理システム**に加えられた変更が影響するのは、**配分チェックサービス**までである。しかも、この**配分チェックサービス**は当初ファサードと考えられていた)。

パフォーマンスチューニング

配分チェックサービスのインタフェースは、ドメイン設計の残りの部分に関わる唯一の要素だが、その内部の実装は、性能問題が発生した時に、それを解決できる場となることがある。例えば、**販売管理システム**が別のサーバで稼動しており、地理的にも別の場所にあるかもしれないと仮定する。そうすると、通信のオーバーヘッドが著しくなる可能性がある上、配分チェックのたびにメッセージ交換が2回発生してしまう。2回目に送られるメッセージは、**販売管理システム**を呼び出して、特定の貨物を受け入れるべきかどうかという基本的質問に答えるものであり、このメッセージを別のものに置き換えることはできない。しかし、最初のメッセージは貨物の**エンタープライズセグメント**を取得するものであり、配分の決定そのものと比べると、比較的静的なデータとふるまいに基づいている。設計における選択肢の1つに、この情報をキャッシュして、**配分チェックサービス**のあるサーバへ配置しなおせるようにし、メッセージングのオーバーヘッドを半分に減らす、というものがある。ただ、これによって得られる柔軟性には代償を払う必要がある。設計はより複雑になるし、重複したデータは何らかのかたちで最新の状態に保たなければならないのだ。だが、分散システムで性能が決定的に重要である場合、柔軟な配置も設計における重要な目標となり得る。

最後に

この話は以上である。この統合のせいで、概念的に一貫した単純な設計がもつれて混乱した可能性もあったが、現在では腐敗防止層やサービス、エンタープライズセグメントを使用することによって、**販売管理システム**の機能を予約システムへきれいに統合できたし、ドメインも豊かになった。

次が設計上の最後の問題である。**エンタープライズセグメント**を取得する責務を、なぜ**貨物**に持たせないのか？ 一見したところ、取得するのに必要なデータがすべて**貨物**の中にあれば、それを**貨物**の導出属性とするのは的確なように見える。しかし、残念ながらそう単純ではないのだ。**エンタープライズセグメント**は、ビジネス戦略にとって有用な輪郭に沿って分割するように、任意に定義される。同じエンティティでも、目的が変われば異なるセグメントになる可能性があるのだ。ここでは**予約を配分する**目的で、特定の**貨物**のセグメントを取得しようとしているが、税金会計のためであれば、全く別の**エンタープライズセグメント**になるかもしれない。この配分用**エンタープライズセグメント**さえ、新しい販売戦略のために**販売管理システム**が再構成されれば変わるかもしれない。したがって、**貨物**が**配分チェックサービス**について知る必要が生じるが、それは**貨物**の概念上の責務か

ら大きく外れ、しかも、特定の種類の**エンタープライズセグメント**を取得するためのメソッドを、大量に背負い込むことになる。したがって、この値を取得する責務は、セグメントのルールを理解しているオブジェクトに課すのが適切なのであり、そうしたルールが適用されることになるデータを持っているオブジェクトに課すべきではない。そうしたルールは別の「**戦略**」オブジェクトに切り出して、**貨物**に渡し、**エンタープライズセグメント**を取得できるようにしてもよい。この解決策は、ここで取り上げる要求を超えていると思われるが、後の設計における選択肢の1つになり得るし、それほど破壊的な変更にはならないはずである。

第3部　より深い洞察へ向かうリファクタリング

第8章　ブレイクスルー 195
第9章　暗黙的な概念を明示的にする 207
第10章　しなやかな設計 247
第11章　アナリシスパターンを適用する 299
第12章　デザインパターンをモデルに関係づける 315
第13章　より深い洞察へ向かうリファクタリング 329

本書の第2部では、モデルと実装との一致を保つための基礎を構築した。実証済みの基本構成要素を矛盾のない言語と一緒に使用することで、開発作業はいくらか健全になる。

もちろん、本当の課題は、的確なモデルを実際に**見つける**ことにある。こうしたモデルは、ドメインエキスパートの微妙な関心事をとらえて、実用的な設計を推進することができる。最終的に開発したいのは、ドメインについての深い理解をとらえたモデルである。そうすることで、ソフトウェアはドメインエキスパートの考え方とさらに一致して、ユーザの要望に対してより機敏にうまく対応できるようになるはずだ。第3部では、そうした目標を明確にした上で、目標が達成されるまでのプロセスを記述する。また、設計は、開発者自身の要求と同じく、アプリケーションの要求に対しても対応しなければならないのだが、ここでは、そのために適用される設計の原則とパターンをいくつか説明する。

役に立つモデルの開発が成功しているのを見ると、次の3点が導き出せる。

1. 洗練されたドメインモデルを作ることは可能であり、苦労して実現する価値がある。
2. そうしたモデルを開発するのにほぼ欠かせないのが、リファクタリングを行うイテレーティブなプロセスである。このプロセスにおいて、ドメインエキスパートと、ドメインについて学ぶことに興味のある開発者が、密接に協力する。
3. そうしたモデルは、実装して効果的に使用するにあたって、高度な設計スキルが必要かもしれない。

リファクタリングのレベル

リファクタリングとは、機能を変更しないようにしつつ行われる、ソフトウェアの再設計である。手の込んだ設計上の決定をアップフロントに行うのではなく、開発者はコードに対して、個別の小規模な設計変更を連続的に加える。それぞれの変更においては、既存の機能はそのまま残しつつ、設計をより柔軟にしたり理解しやすくしたりする。自動化されたユニットテストが一式揃っていることで、コードに対する実験を比較的安全に行えるようになる。このプロセスにより、開発者はずっと先まで見通さなくても済むようになる。

しかし、リファクタリングの方法に関する文献はほぼすべてが、コードを読みやすくするか、非常に詳細なレベルで機能を追加するといった機械的な変更に焦点を合わせている。『パターン指向リファクタリング』[1]のアプローチを用いることで、確立されたデザインパターンを適用できる機会に開発者が気づいた時には、よりレベルの高い目標がリファクタ

1 リファクタリングの目標としてのパターンという考え方は、『デザインパターン』(Gamma他 1995) で簡単に言及されている。Joshua Kerievskyは、パターンへのリファクタリングをより成熟した有用な形式に発展させた (Kerievsky 2003)。

リングのプロセスに与えられる。ただし、この場合でも、設計の品質について、主に技術的な観点から見ている。

だが、リファクタリングの中でシステムの活力に最も大きく影響するのは、ドメインについての新たな洞察によって動機づけられたものや、コードを通してモデルの表現を明確にするものなのだ。この種のリファクタリングが、デザインパターンに向けたリファクタリングやマイクロリファクタリングなど、継続的に進めるべきものに取って代わることは決してない。それとは別のレベルを重ね合わせるものなのだ。それが、より深いモデルへのリファクタリングである。ドメインについての洞察に基づいてリファクタリングを実施する際には、たいてい一連のマイクロリファクタリングを伴うが、それを行う動機となっているのはコードの状態の良し悪しだけではない。むしろ、マイクロリファクタリングを行うことによって、より一層の洞察に満ちたモデルへ向かう際に便利な、変更の単位が作り出されるのだ。目標は、コードが何を行っているかを開発者が理解できるようになることだけではなく、**なぜ**それを行うかを理解した上で、ドメインエキスパートとの継続的なコミュニケーションに、コードが行っている処理とその動機を関係づけられるようになることである。

『リファクタリング』(Fowler 1999) に挙げられているカタログは、定期的に話題に上るマイクロリファクタリングのほとんどを網羅している。それらのリファクタリングを行う動機となっているのは、コード自体の中に見つけることのできる問題である。それとは対照的に、新たな洞察が現れた時には、ドメインモデルはきわめて広い範囲にわたって変化するので、包括的なカタログにまとめることができない。

モデリングは他の探究と同じように、本質的に構造化されていないのだ。より深い洞察へ向かうリファクタリングは、学習と深い思考が導く所であれば、どこへでも従っていかねばならない。成功したモデルについての公開されているコレクションは、第11章で述べるように、役には立つが、ドメインモデリングをクックブックやツールキットに還元しようとして道を外れてはならない。モデリングと設計には創造性が必要なのだ。次に続く6つの章では、ドメインモデルの改良について考える際の特別なアプローチを、それを実現する設計と合わせて提案する。

深いモデル

オブジェクト分析について説明する従来の方法には、要件定義書にある名詞と動詞を識別して、まずはそれらをオブジェクトとメソッドとして使用するというものがある。この説明は、過度に単純化しているが初心者にオブジェクトモデリングを教えるには有効かもしれない、と考えられている。だが実際には、最初に作られるモデルは、素朴かつ表面的

で、浅い知識にしか基づいていないのが普通なのだ。

例えば、私はかつて輸送アプリケーションに携わったことがあるが、その時最初に考えたオブジェクトモデルには船舶とコンテナがあった。船舶は場所から場所へ移動する。コンテナは荷積みと荷下ろしの業務を通じて、船舶に関連づけられたり、切り離されたりする。これは物理的な輸送という活動を正確に記述してはいる。しかし、輸送ビジネスのためのソフトウェアにとっては、あまり有益なモデルでないことがわかった。

最終的には、多くのイテレーションを通じ、輸送のエキスパートと何か月も作業した後で、全く異なるモデルが導き出された。これは素人にはわかりにくいが、エキスパートにとっては、ビジネスとの関連がはるかに深いものだった。焦点が貨物を配送するというビジネスに移ったのだ。

船舶オブジェクトは依然として存在したが、「輸送機器の運行（vessel voyage）」というかたちで抽象化された。これは、船舶、鉄道、その他の輸送機器のためにスケジュールされた特定の移動である。船舶自体は二次的なものであり、整備やスケジュールのずれのせいでぎりぎりで置き換えられるかもしれないが、輸送機器の運行は計画通りに進む。輸送コンテナはモデルから消えたも同然である。荷役アプリケーションの方に、これとは別の非常に複雑なかたちで登場しはしたが、そのもともとのアプリケーションのコンテキストでは操作上の詳細にすぎなくなった。貨物の物理的移動は、その貨物に対する法的責任の移動に主役の座を譲った。「船荷証券（bill of lading）」のような、一見すると明らかではないオブジェクトが前面に来たのだ。

新しいオブジェクトモデラがプロジェクトに参加すると、決まって最初に提案するものがあった。それが、あるはずのクラス、船舶とコンテナである。彼らが聡明でなかったわけではなく、ただ、深いモデルを発見するためのプロセスを経験していなかっただけなのだ。

深いモデルは、ドメインエキスパートの主要な関心事と、それに最も深く関連した知識に関する明快な表現を提供するが、一方で、ドメインの表面的な側面は捨て去るのだ。この定義は抽象化には言及していない。深いモデルには通常、抽象的な要素があるが、問題の核心に迫る所には具体的な要素もあるだろう。

用途の多さ、簡潔性、説明能力は、ドメインと真に調和したモデルから生まれる。そういうモデルには、ほぼ常に、ある特徴が備わっている。それは、ビジネスエキスパートが好んで使用する、シンプルで、もしかしたら抽象的かもしれない言語である。

深いモデル／しなやかな設計

恒常的にリファクタリングを行うプロセスでは、設計そのものによって変更が支えられる必要がある。第10章では、設計を変更する際にも、システムの他の部分と統合する際にも、利用しやすくする方法を取り上げる。

設計にいくつかの特徴を持たせることで、変更するのも使用するのも、より容易になる。この特徴は複雑ではないが、手強いものではある。「しなやかな設計」とそれに取り組むための方法が、第10章のテーマである。

恒常的なリファクタリングの成果は他にもある。モデルとコードを繰り返し変形させるというその行為自体によって、それぞれの変更が新しい理解を反映していれば、最もよく変更される場所が柔軟になり、共通したことを行うのも簡単になるのだ。使い込んだグローブは、指の曲がる場所はしなやかになるが、他の部分は堅く、手を保護してくれる。モデリングと設計に関するこのアプローチは多くの試行錯誤を伴うが、それによって実際に変更が行いやすくなり、繰り返すうちに柔軟性のあるしなやかな設計へと近づいていく。

しなやかな設計は、変更を容易にすることに加え、モデル自体の改良にも貢献する。モデル駆動設計は、2本の足で立っている。深いモデルにより、表現力豊かな設計ができるようになる。同時に設計が、開発者が実験できるくらい柔軟で、何が起きているかを開発者に示せるくらい明確であれば、モデルを発見するプロセスに対して実際に洞察を与えられる。フィードバックループにおいて、設計がプロセスに与える洞察は欠かせない。我々が求めているモデルは、単に優れた考え方を集めたものではなく、システムの基礎になるものだからである。

発見のプロセス

抱えている問題に本当に合った設計を作り出すためには、まず、ドメインと最も関連の深い、中心となる概念をとらえるモデルがなくてはならない。そういう概念を積極的に探し、設計に取り込むことが、第9章「暗黙的な概念を明示的にする」におけるテーマである。

モデルと設計との間には密接な関係があるため、コードのリファクタリングが難しいと、モデリングプロセスは止まってしまう。第10章「しなやかな設計」では、自分だけではなくソフトウェア開発者のためにソフトウェアを作成する方法を議論する。それによって、ソフトウェアを拡張する際にも変更する際にも、生産性が向上する。この作業は、モデルをさらに改良する作業と歩調を合わせて進んでいく。この作業をするにあたっては、より高度な設計テクニックが必要となることも多く、また、モデルの定義をより厳密にしなけ

ればならないことも多い。

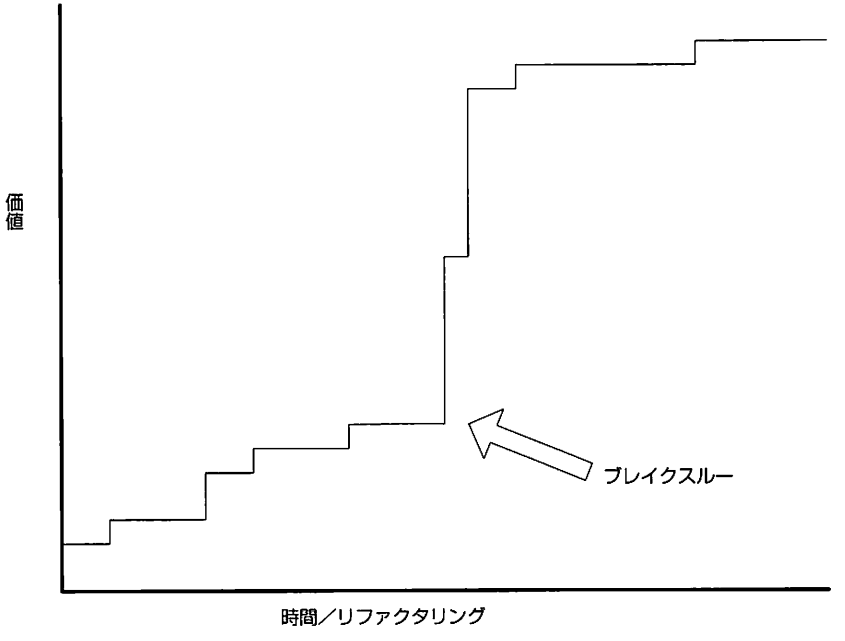
見出した概念をモデル化する上で適切な方法を見つけるには、通常は創造性と試行錯誤が必要だが、使えるパターンを誰かが準備していることもある。第11章、第12章では、「アナリシスパターン」と「デザインパターン」の適用について議論する。こうしたパターンは既製の解決策というわけではないが、知識をかみ砕くプロセスに寄与して、探究すべき範囲を狭めてくれる。

このように、語るべきことは多いが、この第3部はドメイン駆動設計で起きる最も刺激的な出来事の話から始めよう。モデル駆動設計と明示的な概念によって舞台が整っていると、ブレイクスルーが起きることがある。ソフトウェアを、予想以上に表現力豊かで用途が多いものへと変える機会が開けるのだ。これは、新しい機能を意味する場合もあれば、柔軟性のないコードの大きな塊を、より深いモデルの持つシンプルで柔軟な表現に置き換えることを意味する場合もある。そういうブレイクスルーは毎日あるものではないが、非常に貴重なので、ブレイクスルーが訪れた時には、そこに生まれる好機に気づいて逃さないようにしなければならない。

第8章で紹介するのは、より深い洞察へ向かうリファクタリングのプロセスがブレイクスルーにつながった、実際のプロジェクトの話である。こうした経験は、前もって計画しておくことができないものだ。それでも、ドメインのリファクタリングについて考えるのにふさわしいコンテキストを提供してくれるのだ。

第8章

ブレイクスルー



リファクタリングをすることで得られる収穫は、労力に正比例するものではない。通常は小さい努力に対するわずかな収穫があり、小さな改良が積み重なっていく。こうしたリファクタリングはエントロピーに対する戦いであり、時代遅れのレガシーとなってしまうことを防ぐ防衛前線である。しかし、最も重要な洞察の中には、突如として現れて、プロジェクト全体に衝撃を与えるものもあるのだ。

ゆっくりだが確実に、チームは知識を吸収し、それをかみ砕いてモデルにする。深いモデルは、一度に1オブジェクトという小規模なリファクタリングが連続して行われることで、徐々に姿を現すことができるのだ。この小規模なリファクタリングは、こちらで関連を微調整し、あちらで責務を移動する、といったかたちで行われる。

だが、継続的なリファクタリングをすることによって、より不規則なものが生まれる土壌も整えられる。コードとモデルを改良するたびに、開発者の視界は明確になってくる。このように視界が明確になったことにより、洞察のブレイクスルーをもたらす可能性が作り出される。立て続けに変更が行われた結果として現れるモデルは、実際に起きていることやユーザが重要視するものと、より深いレベルで一致する。複雑さが消滅するまさにその時に、用途の広さや説明する力が急激に増大するのである。

この種のブレイクスルーは、テクニックではなく、出来事である。課題は、何が起こっているかを認識して、どう対処すべきかを決めることである。この経験の雰囲気を伝えるために、数年前に私が取り組んだプロジェクトで実際に起きたことを例に、どのようにして非常に有益な深いモデルに到達したかを話すことにしよう。

ブレイクスルーの話

ニューヨークでの長い冬をリファクタリングしながら過ごしたことで、我々は、ドメインの主要な知識をある程度とらえているモデルと、アプリケーションの実際の処理をいくつか行う設計にまで、たどり着いていた。当時開発していたのは、投資銀行のシンジケートローンを管理する巨大なアプリケーションのコアとなる部分であった。

例えばIntel社が、10億ドルの工場を建てたいと思ったとすると、そこで必要となる融資額は1つの融資会社で引き受けるにはあまりに巨額である。そこで、複数の融資会社がシンジケートを組織し、**ファシリティ**（コラム参照）を支える資金を共同で出資するのである。通常は、投資銀行がシンジケートリーダの役割を務め、取引やその他のサービスを調整する。このプロジェクトは、そうしたプロセス全体を追跡してサポートするソフトウェアを作成するものだった。

コラム――「ファシリティ」とは何か？

この文脈でいう「ファシリティ」は施設ではない。たいていのプロジェクトと同様、ドメインエキスパートの専門的な用語法が開発側の語彙に取り込まれて**ユビキタス言語**の一部となったものだ。商業銀行ドメインにおいて、ファシリティとは、「融資する側の企業による確約」である。クレジットカードはファシリティであり、顧客の請求に応じて、あらかじめ決められた金利で事前に取り決めた限度額まで借りる権利を与える。カードを使用すると未払いのローンが発生し、さらに追加手数料がそれぞれファシリティから「引き出されて」、ローンの総額が増える。最終的には、ローンの元金を返済することになる。また、年会費を払うこともある。年会費とは、カード（ファシリティ）を持つ権利に対する料金であり、ローンには依存しない。

悪くないモデルなのだが…

我々は実によい気分だった。4か月前は、引き継いだコードベースが全く機能しないことにひどく悩まされていて、それ以来ずっと、一貫したモデル駆動設計にするべく奮闘してきたのである。

図8.1に示すモデルは、一般的なケースをきわめて単純化したものだ。**ローン出資**（**Loan Investment**）は、**ローン**（**Loan**）における特定の出資者の分担額を表すオブジェクトで、これは**ファシリティ**（**Facility**）における出資者の分担率に比例する。

だが、混乱の兆候もあった。予想外の要求につまずいて、設計が入り組んだものになってしまうことが何度もあったのだ。代表的な例としては、**ファシリティ**の分担率は、特定のローンにおける引き出しに参加する上での**指針**にすぎないということが少しずつわかってきたことが挙げられる。借り手が融資を求めた場合、シンジケートリーダがメンバ全員に分担金を出資するよう依頼する。

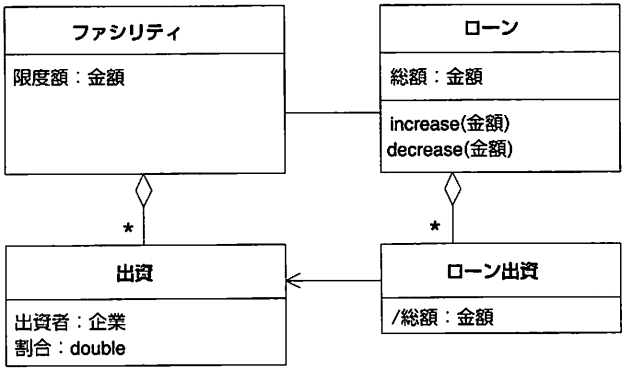


図8.1　貸し手の分担率が固定であると想定しているモデル

依頼されると、出資者は自分の分担を支払うことになるのが普通だが、シンジケートの他のメンバと交渉して、出資を減らす（増やす）こともしばしばある。これには、モデルへ**ローン調整**（**Loan Adjustment**）を追加することで対応した。

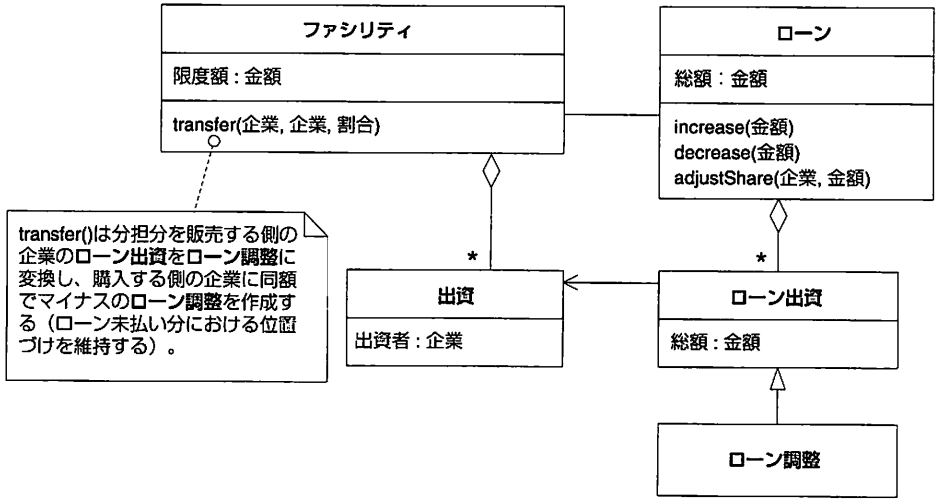


図8.2 問題を解決するために少しずつ変更されたモデル。ローン調整は、貸し手が当初合意していたファシリティ内での分担率からの変更を追跡する

こうした改良により、さまざまな取引のルールが明らかになっても、我々はついて行くことができた。しかし、複雑さは増していっており、本当に信頼できる機能を素早く作り上げることができているとは思えなくなってきた。

さらに厄介だったのは、丸めによって起こる微妙な不整合である。この問題は、どんどん複雑になっていくアルゴリズムでは押さえ込むことができなくなっていた。実際には、1億ドルの取引で余分な数セントがどうなったかなど誰も気にかけないが、銀行家は、そういう小額を正確に計算できないソフトウェアを信用しない。これほど苦労するのは基本的な設計に問題がある兆候ではないかと、我々は疑い始めた。

ブレイクスルー

ある週になって突然、何が悪かったのか理解できた。我々のモデルは、このビジネスにはふさわしくない方法で、ファシリティの分担率とローンの分担率を結びつけていたのである。この驚くべき新事実は、広範囲に影響を与えた。ビジネスエキスパートはうなずきながら、熱心に支援してくれた。ただ、あえて言えば、なぜ我々がこんなに時間がかかったのか疑問に思ってもいただろう。ドメインエキスパートの助けを受けながら、我々はホワイトボード上で新しいモデルについて徹底的に議論した。詳細はまだ固まっていなかったものの、新しいモデルが持つ決定的に重要な特徴は理解できた。すなわち、ローンの分担率とファシリティの分担率は互いに独立して変更できたのだ。その洞察を基に、新モデルを視覚化した図を使用して、無数のシナリオに対するウォークスルーを実施した。これ

を図8.3に示す。

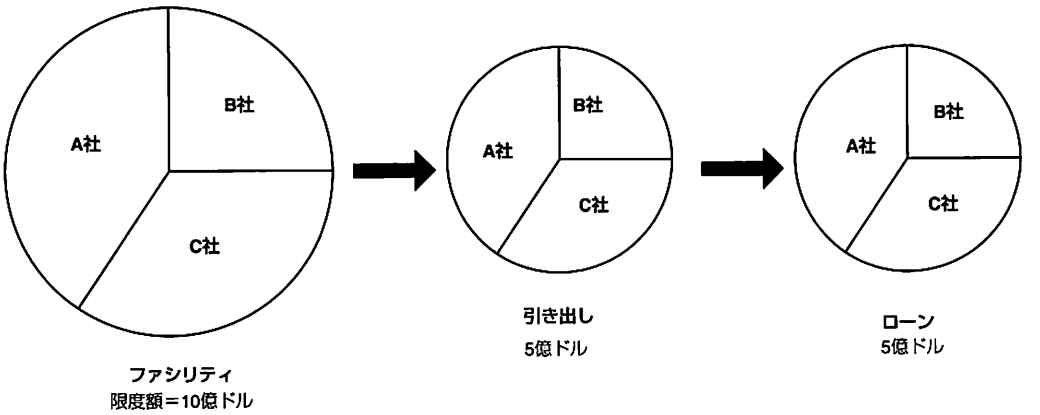


図8.3　ファシリティの分担率に基づいて配分される引き出し

この図8.3が表しているのは、借り手がまず、ファシリティで約束された10億ドルの中から5億ドルを引き出すことにしたということだ。貸し手の3社は、ファシリティの分担率に対して正確に資金を出し合うから、3社間で5億ドルを分けることになる。

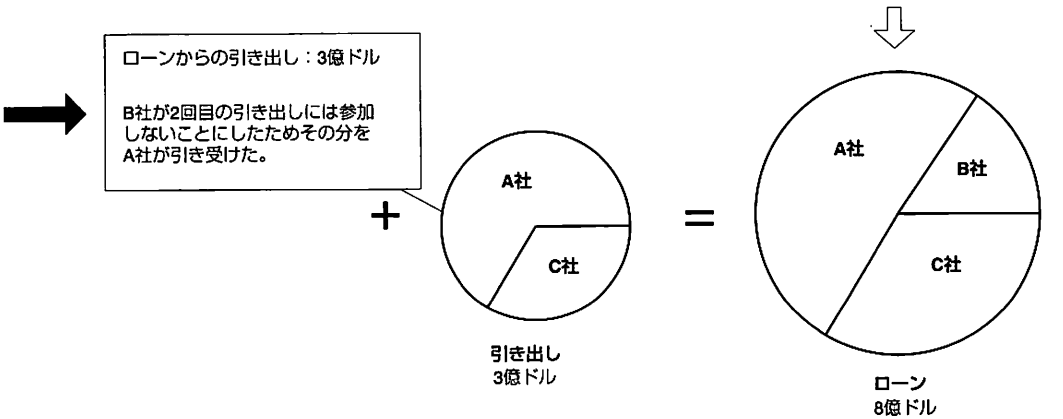


図8.4　貸し手側のB社が2回目の資金引き出しに加わらないことにした

次に、図8.4では、借り手がさらに3億ドルを引き出して、未払いのローンが8億ドルになるが、それでもファシリティの限度額10億ドルは超えない。今回はB社が参加しないことにしたため、その分はA社が引き受ける。引き出しに対する分担率は、こうした出資に関する意思決定を反映したものとなる。この引き出し額がローンに加算されると、ローン

の分担率は**ファシリティ**の分担率に比例しなくなる。これはよくあることなのだ。

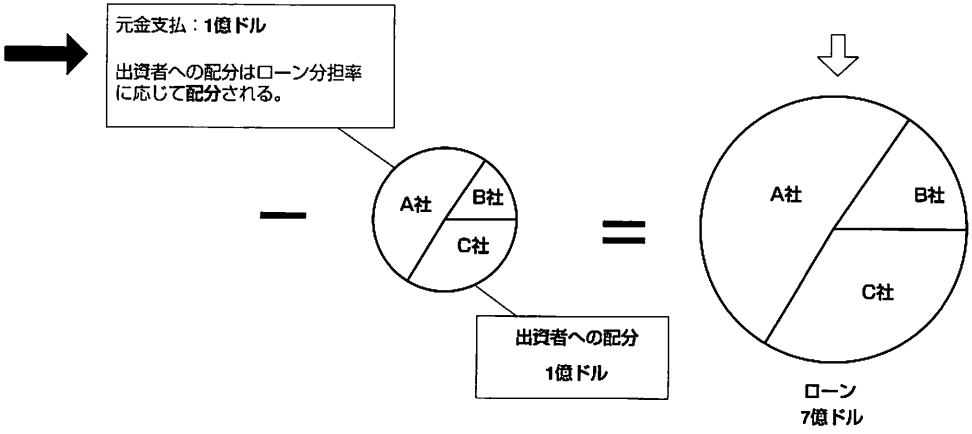


図8.5 元金支払いは、常に未払いローンに対する分担率に比例して配分される

借り手が**ローン**を支払った場合、その金額は**ファシリティ**ではなく**ローン**の分担率に従って、貸し手の間で分割される。同様に、金利の支払も**ローン**の分担率に従って分けられる。

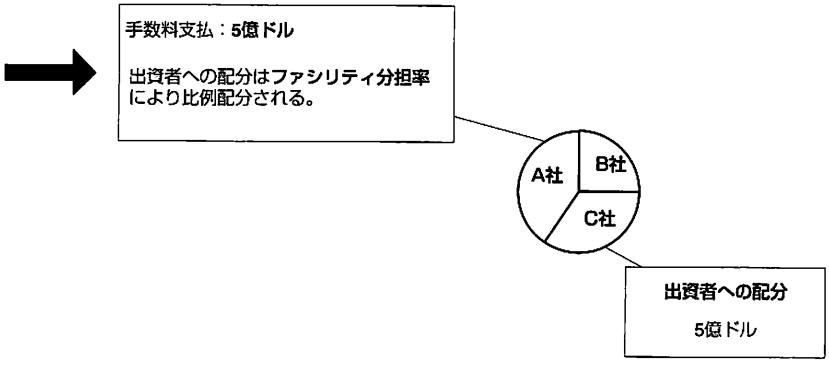


図8.6 手数料の支払は、常にファシリティの分担率に比例して配分される

一方、借り手が**ファシリティ**を使用する権利に対して手数料を支払った場合、その金額は**ファシリティ**の分担率に従って分けられ、実際に誰が貸したかは関係しない。**ローン**は手数料の支払いによって減ることはない。さらに、貸し手同士が、手数料の分担率を金利の分担率から切り離して交換するといったシナリオまである。

さらに深いモデル

我々は深い洞察を2つ得ていた。そのうち1つは、ここで言う「出資」と「ローン出資」が一般的かつ本質的な概念の、2つの特殊なケースにすぎないと認識したことである。それが、分担率、すなわちシェアという概念だ。ファシリティのシェア、ローンのシェア、**支払い配分**のシェアなど、いたるところにシェアがある。分割できる値なら何にでもシェアがあるのだ。

慌ただしい日々を何日か過ごした後で、私はシェアのモデルをスケッチしてみた。これ以降の説明で使われている言語は、エキスパートとの議論と、共同で考察してきたシナリオで使用したものだ。

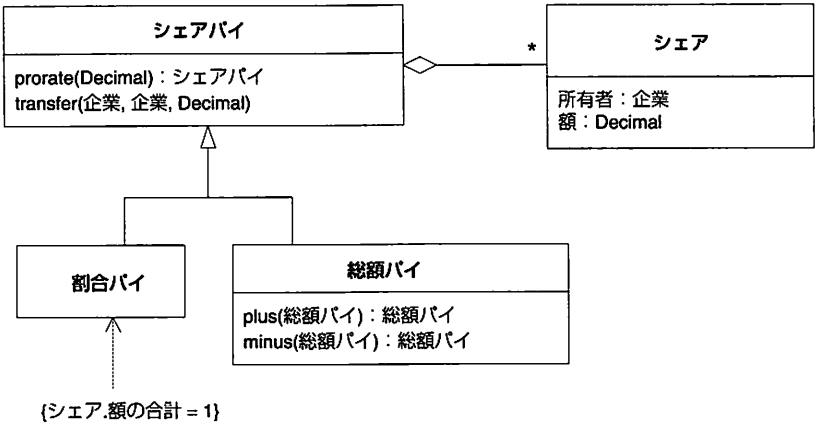


図8.7　シェアの抽象モデル

それに合わせて、新しいローンモデルもスケッチした。

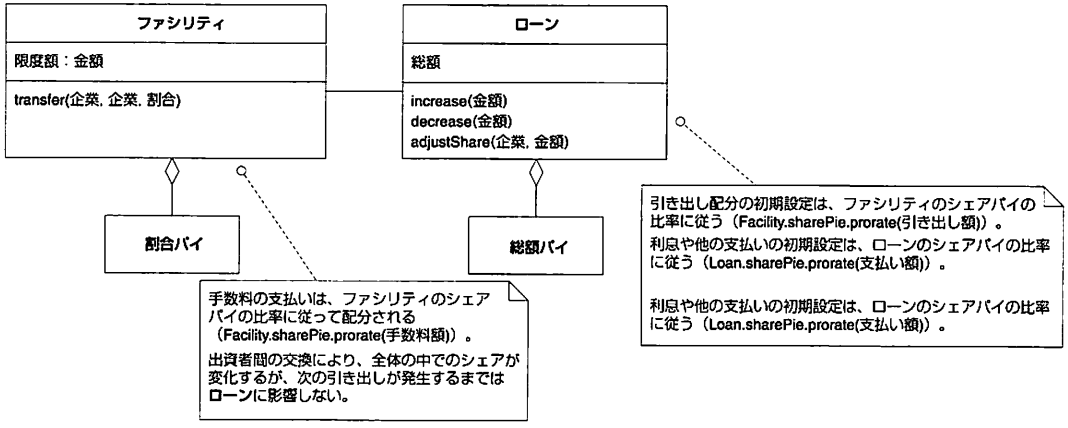


図8.8 シェアパイを使用するローンモデル

ファシリティや**ローン**のシェアに関して、特別なオブジェクトはもうなくなった。どちらも、より直観的にわかる「**シェアパイ（Share Pie）**」に括り出されたのだ。この一般化によって「シェア算」を導入できるようになり、それによってどんな取引においてもシェアの計算が大幅に簡略化された。同時にその計算も表現力豊かで簡潔になり、容易に組み合わせられるようになった。

しかし、何より大きかったのは、新しいモデルが不適切な制約を除去したことにより、問題がいくつも消え去ったことだ。このモデルによって、**ローンのシェア**は**ファシリティのシェア**から切り離され、その一方で、合計や手数料配分などに関する正しい制約は保たれた。**ローンのシェアパイ**を直接調整できるようになったため、**ローン調整**はもはや不要になり、特殊なケースのためのロジックが大量に削除された。

ローン出資も姿を消し、その時点になって初めて、我々も、「ローン出資」が銀行用語ではないことに気づいた。実は、ビジネスエキスパートは、かねてより、この用語が理解できないと何度も言っていたのである。彼らは、我々の持っているソフトウェアの知識を尊重し、ローン出資の概念が技術的な設計では役に立つと思っていたのだ。だが、実際に我々がそれを設計した時に基づいていたのは、ドメインに関する不完全な理解だった。

このドメインの新しい見方を基にしたところ、突然、すでにあったシナリオを読むのが、どれも比較的楽に、また、かつてないほどシンプルになった。**しかも、今回のモデル図は、ビジネスエキスパートが完全に理解できた。彼らは前の図に対しては、自分たちには「専門的すぎる」と言っていたのである。**ホワイトボードにスケッチしただけでも、最も困難だった丸めの問題が根こそぎ取り除かれていて、複雑な丸めのコードを一部捨てられそうだということが見てとれた。

この新しいモデルはうまく機能した。本当に、本当にうまくいったのだ。

そして、我々は全員気が重くなった！

冷静な意思決定

あなたは当然、我々の士気がこの時点で上がったと思うことだろう。しかし、そうではなかったのだ。厳しい納期が近づく中、プロジェクトのスケジュールは、すでに大幅に遅れていたのである。我々を支配していた感情は恐怖だった。

リファクタリングの教義は、常に小さいステップで進みつつ、必ずすべてが動くよう保つというものだ。しかし、当時のコードをリファクタリングして、新しいモデルに合わせるには、周りを取り巻くコードの多くを変更しなければならなかった。変更の過程で安定して立ち止まれる場所も、全くないとは言わないにしても、ごくわずかしかなかった。いくつか小さい改善ができそうなところもあったが、新しい概念へ近づけるようなものではなかった。小さいステップを何度か繰り返せば、目的地にたどり着けそうではあったが、その間、アプリケーションの一部が使えなくなってしまう。しかもこの話は、こうしたプロジェクトでテストの自動化が幅広く使用されるようになる以前のことだ。実際、自動化されたテストなどなく、予期しない破綻が起きることは目に見えていた。

努力が必要なこともわかっていたが、何か月も無理をしてきたせいで、すでに我々は疲れきっていた。

この時点で、我々はプロジェクトマネージャとミーティングを行った。私はこのミーティングを忘れることはないだろう。このマネージャは知的で大胆な人だった。彼はいくつか質問をしてきた。

Q：新しい設計で現在の機能を回復するには、どのくらいの時間が必要ですか？
A：3週間ぐらいです。
Q：新しい設計でなくても、問題を解決できますか？
A：おそらく。でも、確証はないですね。
Q：ここで解決しなかったとして、次のリリースで先に進むことはできると思いますか？
A：変更しなければ、先に進むのは遅くなるでしょう。それに、一度導入してしまうと、変更はずっと難しくなるでしょう。
Q：我々は、変更するのが正しいと考えるべきですか？
A：政治的に考えれば状況が不安定なことはわかっていましたので、やるべきことには対処してきました。そして、私たちは疲れています。でも、やるべきです。これはよりシンプルな解決策で、ビジネスにもずっとうまく整合します。長い目で見れば、リスクは低いです。

彼は我々にゴーサインを出し、非難には自分が対処すると言ってくれた。その決断を下した彼の勇気と信頼には、今でも非常に感嘆している。

我々は必死に取り組んで、3週間で仕上げた。大変な仕事ではあったが、驚くほどスムーズに進んだ。

結 末

我々を当惑させるような、予期せぬ要求変更は出なくなった。丸めのロジックは、決して単純にはならなかったものの、安定し、わかりやすくはなった。我々はバージョン1を納品し、バージョン2への道も明確だった。私は辛うじてノイローゼにならずに済んだのである。

バージョン2が進化するにつれ、このシェアパイはアプリケーション全体の統一テーマとなった。技術担当者とビジネスエキスパートは、システムについて議論する際にこの言葉を使用した。**販売部門の担当者はこの言葉を使って、見込み客に機能を説明した。**こうした見込み客や顧客は、すぐに言葉の意味を理解し、機能について議論するのに使用した。この言葉は本当の意味でユビキタス言語の一部となったのだ。それはこの言葉が、ローンシンジケートとはどのようなものであるかということの核心をついていたからである。

好 機

より深いモデルへのブレイクスルーが訪れそうな時は、恐怖を感じることが多い。こうした変化は、たいていのリファクタリングよりチャンスも多いが、同時にリスクも大きいのだ。タイミングがよくないこともある。

そうあって欲しいとどれほど思っていても、進捗は順調に進まない。本当に深いモデルへの移行は、発想の転換であり、設計の大幅な変更を必要とする。多くのプロジェクトにおいて、モデルと設計の最も重要な進歩は、こうしたブレイクスルーのかたちで現れる。

基本への集中

ブレイクスルーを引き起こそうとして麻痺してはならない。ブレイクスルーが起こり得るのは、適度なリファクタリングを何度も行った後のことである。ほとんどの時間は少しずつ改良することに費やされるが、こうした連続的な改良の中から、モデルについての洞察が徐々に姿を現してくるのだ。

ブレイクスルーの舞台を整えるためには、知識をかみ砕き、強固なユビキタス言語を育成するのに集中すること。重要なドメインの概念を探究して、それをモデルで明示すること（第9章で議論する）。設計をよりしなやかになるように改良すること（第10章参照）。

モデルを蒸留すること（第15章参照）。こうした予測可能な方策を押し進めることで、モデルや設計がより明確になる。こうした明確さは、通常、ブレイクスルーの前兆となるのだ。

わずかな改良をためらってはならない。たとえ、いままでの汎用的な概念フレームワークの枠から外に出なかったとしても、こうした改良によって徐々にモデルは深まるのだ。あまりにも遠い先を見て、麻痺してはならない。ただ、機会を見逃さないように注意すること。

エピローグ：新しい洞察の連鎖

このブレイクスルーによって、我々は困難を切り抜けられたが、それで話が終わったわけではない。この深くなったモデルによって、アプリケーションを豊かにして設計をよりわかりやすくする予想外の好機が開かれたのである。

シェアパイバージョンのソフトウェアをリリースしてから、わずか数週間後、我々は、モデルにまた別のぎこちない側面があることと、それが設計を複雑にしていることに気づいた。ある重要なエンティティが欠落していたので、その分の余計な責務を他の複数のオブジェクトが引き受けていたのだ。具体的には、ローンからの引き出しや手数料の支払いなどを支配している重要なルールがあったのだが、そのロジックがすべて、**ファシリティ**と**ローン**のさまざまなメソッドに詰め込まれていたのである。こういう設計上の問題は、シェアパイのブレイクスルーが起こる前はほとんど気づかれることがなかったが、我々の視界が明確になったことで明るみに出たのだ。ここで我々が気づいたのは、モデルのどこにもない複数の用語が議論をする際に飛び出しているということである。こうした用語の中には「トランザクション」（金融取引の意）などがあり、それがファシリティやローンに詰め込まれたメソッドに暗に含まれていることがわかってきたのだ。

これまで説明してきたものと同じようなプロセスを経たことで（ただし、ありがたいことに時間的プレッシャーはずっと少なかったが）、さらなる洞察と一層深いモデルに至った。この新しいモデルによって、**トランザクション**のような暗黙的な概念が明示的になると同時に、**ポジション**（**Position**）（**ファシリティ**と**ローン**を含む抽象概念）も単純化されたのである。さまざまなトランザクションを定義することが容易になり、しかも、ルールや交渉手続き、承認プロセスを含めて、すべてを意図が比較的わかりやすいコードに記述することができた。

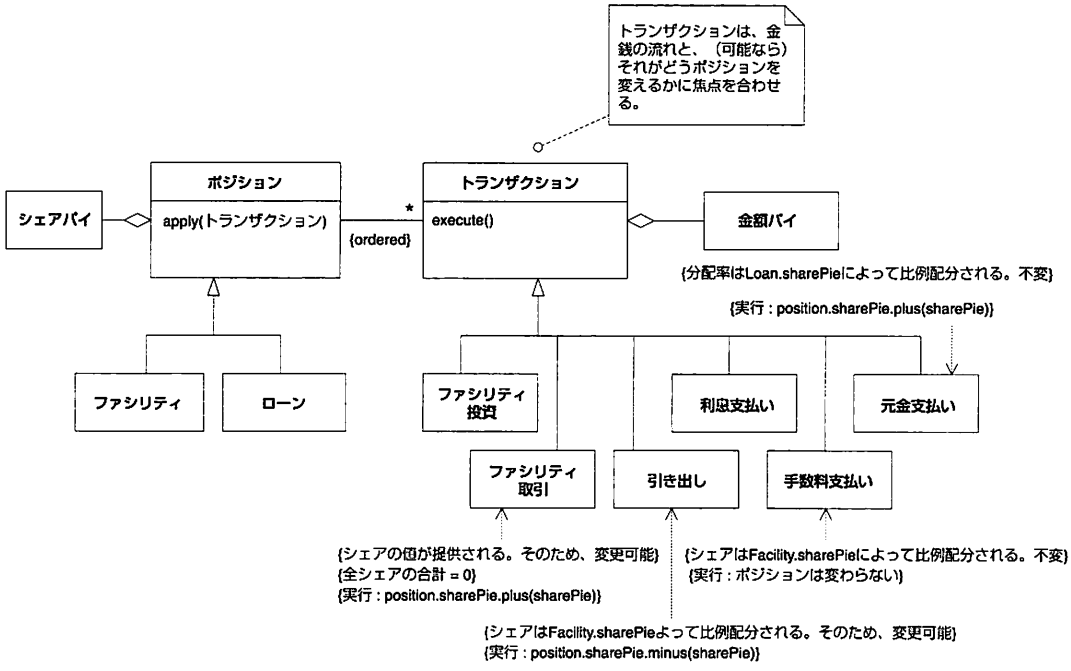


図8.9 数週間後に起きた別のモデルのブレイクスルー。トランザクションに対する制約を正確に表現するのは容易である

深いモデルへ向かう本物のブレイクスルーの後にはよくあることだが、この新しい設計が明確かつシンプルだったことで、新しいユビキタス言語に基づくコミュニケーションが改善したこととも相まって、さらなるモデリングのブレイクスルーにつながった。

我々の開発ペースは、ある段階に至って加速したのだが、たいていのプロジェクトは、この段階になると、すでに構築したものの量と複雑さに足をとられて、身動きが取れなくなり始めるものなのだ。

第9章

暗黙的な概念を明示的にする

深いモデリングといえば聞こえはいいが、実際にはどうすればよいのだろう？ 深いモデルに力があるのは、ドメインの中心となる概念や抽象を含んでいるからだ。こうした概念や抽象によって、ユーザの活動や問題、その解決策に関する本質的な知識を、簡潔かつ柔軟に表現できる。最初のステップは、そのドメインにおける本質的な概念を、なんとかしてモデルで表現することである。改良はその後の話で、知識のかみ砕きとリファクタリングを何度も繰り返した後にやってくる。しかし、このプロセスが本当に軌道に乗るのは、重要な概念が認識され、モデルと設計の中で明示的にされた時だ。

ドメインモデルとそれに対応したコードが大きく変化するのは、議論で示唆されたり、設計の中に暗に存在したりする概念に開発者が気づき、それを1つあるいは複数のオブジェクトや関係性を使って、モデルの中で明示的に表現した時である。

それまで暗黙的だった概念を明示的にするこうした変化が、時には、深いモデルへとつながるブレイクスルーになることもある。だが多くの場合、ブレイクスルーは、後になってやってくる。数々の重要な概念がモデルの中で明示的にされた後であり、連続したリファクタリングによって、こうした概念の責務が繰り返し微調整され、他のオブジェクトとの関係も変化し、その概念につけられる名前さえも数回変更されてからのことなのだ。最終的にはすべてをはっきりととらえることができるようになる。しかし、このプロセスは、暗黙的な概念を、荒削りであっても何らかのかたちで認識することから始まる。

概念を掘り出す

開発者は、暗黙的に潜んでいる概念を明らかにする手がかりに対して敏感にならなければならず、時には積極的に探さなければならない。そういう発見をもたらしてくれるのは、

たいてい、チームの言語に耳を傾け、設計のぎこちない点やエキスパートの発言で矛盾すると思われる点を詳細に調べ、ドメインに関する文献を読み、大量の実験をするといったことだ。

言葉に耳を傾ける

こういう経験をしたことはないだろうか。ユーザがいつも、帳票のある項目について話している。その項目は、さまざまなオブジェクトの属性かもしれないし、もしかしたらデータベースに直接問い合わせた値を編集したものかもしれない。同一のデータセットがアプリケーションの別の場所で組み立てられ、何かを表示したり、帳票に出力したり、取り出したりするのに使われる。しかし、あなたにはそのオブジェクトが必要な理由がわからない。おそらく、特定の用語を用いることでユーザが何を意味しているかを実際には理解できておらず、その用語が重要だとは認識していないのだろう。

そのような時に突然、頭の中の霧が晴れる。帳票に記載されたその項目の名前は、重要なドメインの概念を指し示しているのだ。あなたは興奮して、その新しい洞察についてドメインエキスパートと話をする。あなたがついに理解したのを見て、彼らはほっとした様子を見せるだろう。あるいは、何を今さらと退屈そうな顔をするかもしれない。いずれにしても、あなたがボードに描き始めるモデル図が、いつもしてきたような大げさな身振りの代わりになる。新しいモデルがドメインとどう結びついているのかについて、詳細に関してはユーザから訂正されるが、あなたは議論の質が変わったことを実感できる。あなたとユーザはより正確に互いを理解するようになって、特定のシナリオを解決するためのモデルにおける相互作用の説明もより自然になる。ドメインモデルの言語がより強力になったのである。あなたはコードをリファクタリングして、この新しいモデルを反映させ、設計がより明確になったことに気づくのだ。

ドメインエキスパートの使う言葉に耳を傾けること。何か複雑なものを簡潔に述べている用語がないだろうか？ ドメインエキスパートに言葉の選び方を（たぶん、角が立たないように）正されていないか？ あなたが特定のフレーズを使った時に、ドメインエキスパートたちの困惑した表情が消えることはないか？ これらは、モデルにとって有益になり得る概念を示す手がかりである。

こうした考え方は、旧来の「名詞がオブジェクトになる」というものではない。新しい単語を耳にすることで手がかりができる。それを受けてさらに会話と知識のかみ砕きを続け、明確で有用な概念を切り出すことを目指すのだ。ユーザやドメインエキスパートが設計のどこにもない語彙を使用したら、それは警告のサインである。開発者とドメインエキスパートのどちらもが設計にない用語を使用したら、それは2倍の警告である。

あるいは、それは好機ととらえた方がよいかもしれない。ユビキタス言語を構成する語

彙は、話し言葉やドキュメント、モデル図、さらにはコードにまで行きわたっている。ある用語が設計にないなら、その用語を取り込むことでモデルと設計を改善する、よい機会だということだ。

例 9.1 ——輸送モデルに欠けている概念を聞き分ける

このチームは、動作するアプリケーションをすでに開発しており、これで貨物を予約することができる。今から構築する「業務支援」アプリケーションは、荷受け地と荷出し地、および船舶間の移動で発生する貨物の荷積みと荷下ろしについて、その作業順序の調整を支援するものだ。

予約アプリケーションは経路選択エンジンを使用して、貨物の移動を計画した。各行程はデータベースの一行に格納され、その貨物を運ぶ予定の輸送機器の運行（特定の船舶による特定の航海）のIDと、荷積みされる位置および荷下ろしされる位置を示していた。

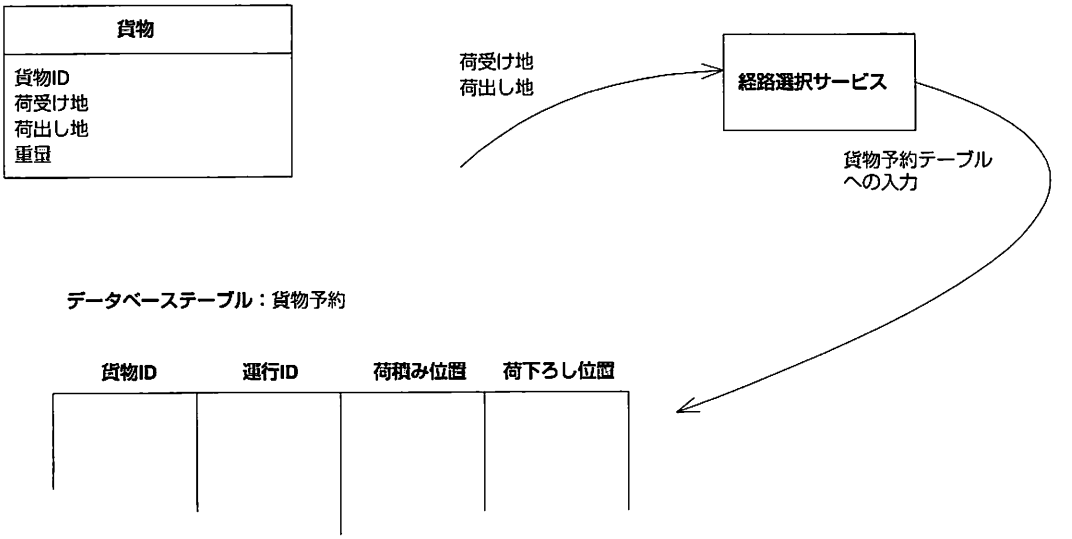


図9.1

開発者と輸送業エキスパートとの会話（大幅に省略してあるが）に聞き耳を立ててみよう。

開発者：「貨物予約」テーブルに、輸送業務アプリケーションで必要なデータがすべて入っているかを確認したいです。

エキスパート：貨物の全輸送日程が必要になりますが、今のところ何が入っていますか？

開発者：各行程についての、貨物ID、輸送機器の運行、それに荷積み港と荷下ろし港です。

エキスパート：日付は入っていないのですか？ 輸送業務では、予想時間に基づいて荷役作業

●例 9.1 ——輸送モデルで欠けている概念を聞き分ける

を請け負う必要があります。

開発者：えーと、それは輸送機器の運行スケジュールから導き出せますね。テーブルは正規化されていますので。

エキスパート：確かに、日付が必要になるのは正規のことですね。輸送業務担当者はそういった輸送日程を使用して、次にやるべき荷役作業の計画を立てますので。

開発者：あ、正規化というのは…。まあ、大丈夫です。日付には間違いなくアクセスできるでしょう。輸送業務管理アプリケーションでは、一連の荷積みと荷下ろしをすべて、それぞれの貨物取り扱い業務が行われた日付と一緒に提供できるようになります。あなたの言う「輸送日程」ですね。

エキスパート：いいでしょう。その輸送日程が、まさに担当者の必要とするものなのです。実際には、ご存知の通り、予約アプリケーションに、輸送日程を出力したり、それを顧客へ電子メールで送信したりするメニュー項目がありますよね。それをどうにか利用できませんか？

開発者：それはたぶん、単なる帳票にすぎないと思います。それに基づいて輸送業務アプリケーションを作ることはできないでしょう。

[**開発者**は考え込んでから、ひらめいたようだった。]

開発者：それでは、この輸送日程が、本当は予約と輸送業務のリンクになっているんですね。

エキスパート：そうです。それに顧客との関係ともリンクしていますね。

開発者：［ホワイトボードに図をスケッチしながら］では、こんな感じだということですか？

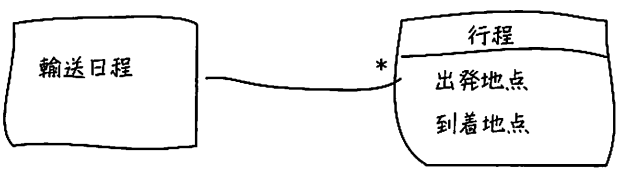


図9.2

エキスパート：はい、基本的にはそれで合っていそうですね。各行程に関して知りたいのは、輸送機器の運行、荷積みと荷下ろしの位置、それと時間ですね。

開発者：それでは、いったん**行程**オブジェクトを生成すれば、そのオブジェクトを使って輸送機器の運行スケジュールから時間を導き出せます。**輸送日程**オブジェクトを輸送業務アプリケーションとの主な接点にすることができます。そして、これを使用するように輸送日程帳票を書き直せば、ドメインロジックをドメイン層へ取り戻すことができます。

エキスパート：今の話、全部はわかりませんでしたが、**輸送日程**には主な用途が2つあって、それが予約アプリケーションの帳票と輸送業務アプリケーションだというのは正しいです。

開発者：あ！ **経路選択サービス**のインタフェースが、データをデータベーステーブルへ入れる代わりに、輸送日程オブジェクトを戻すようにできますね。そうすれば、経路選択エンジンはテーブルについて知る必要がなくなります。

エキスパート：というと？

開発者：つまり、経路選択エンジンが**輸送日程**だけを戻すようにするということです。そうすると、予約の残りが保存される時に、予約アプリケーションが輸送日程をデータベースに保存するようにできます。

エキスパート：今はそうなっていないということですか？

ここで、この開発者は、経路選択プロセスに携わっている他の開発者たちと話し合いに行った。彼らはモデルに対する変更と、それが設計に与える意味合いを徹底的に議論し、必要な時には輸送業エキスパートに助言を求めた。そして考え出されたのが、図9.3の図である。

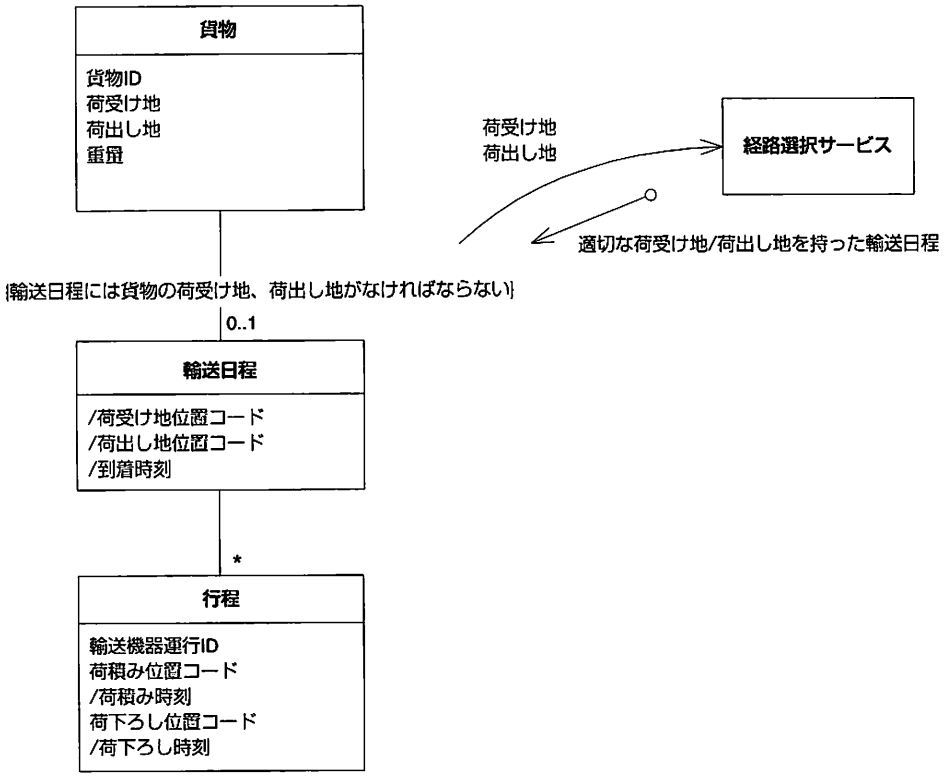


図9.3

●例 9.1 ——輸送モデルで欠けている概念を聞き分ける

次に、開発者たちは新しいモデルを反映するようにコードをリファクタリングした。これは2、3回のリファクタリングを連続して行うものだったが、1週間で素早く行われた。ただ、予約アプリケーションの輸送日程帳票を簡略化させる作業だけは、翌週前半に持ち越された。

開発者は、輸送業エキスパートの言うことにじっくりと耳を傾けてきたから、「輸送日程」の概念がエキスパートにとってどれだけ重要かということに気づくことができた。確かに、データはすべて集まっていて、ふるまいは輸送日程帳票の中に暗黙的に含まれていたのだが、モデルの一部としての**輸送日程**を明示的にすることで、好機が到来したのだ。

輸送日程オブジェクトが明示的になるようにリファクタリングする利点：

1. **経路選択サービス**のインタフェースを、より表現力豊かに定義する。
2. **経路選択サービス**を予約データベーステーブルから分離する。
3. 予約アプリケーションと輸送業務支援アプリケーションの関係（**輸送日程**オブジェクトの共有）を明確にする。
4. 重複を減らす。これは、**輸送日程**が予約帳票と輸送業務支援アプリケーションの両方に対して、荷積み／荷下ろしの時刻を導き出すことによる。
5. ドメインロジックを予約帳票から削除し、隔離されたドメイン層へ入れる。
6. ユビキタス言語を拡張し、それによって開発者とドメインエキスパート間、および開発者同士で、モデルと設計についてより正確に議論できるようにする。

ぎこちなさを精査する

必要とされる概念が、いつも表面に浮かんでいたり、会話やドキュメントに現れたりしているとは限らない。掘り下げて考え出さなければならないこともある。掘り下げるべきところは、設計の中でも最もぎこちない部分だ。手続きによって説明しにくい複雑な処理が行われているところや、新しい要求が出てくるたびに複雑さが増しているようなところである。

欠けている概念があることにさえ、なかなか気づかないこともある。すべての処理はオブジェクトで行われているが、その責務の中にはぎこちないものが見つかるかもしれない。あるいは、何か足りないとわかっていても、モデルの解決策がなかなか浮かばないかもしれない。

そういう時には、積極的にドメインエキスパートを調査に参加させるべきである。運がよければ、彼らはさまざまに考えを巡らしたり、モデルで実験したりするのを楽しむかもしれない。そこまで運がよくなければ、あなたは開発者仲間と何かを考え出し、ドメインエキスパートに検証してもらいつつ、顔色をうかがいながら、モデルに不満がありそうか、認められたのかを判断することになるだろう。

例9.2——利息を得る　難しい方法

次の話は、商業融資をはじめとした利息を生む資産に投資する、架空の金融会社が舞台である。投資と、そこから得られる収益を追跡するアプリケーションがあり、1機能ずつ徐々に進化してきている。毎晩、あるコンポーネントがバッチスクリプトとして実行され、その日の利息（interest）と手数料（fee）をすべて計算してから、社内の会計ソフトウェアで適切に記録することになっていた。

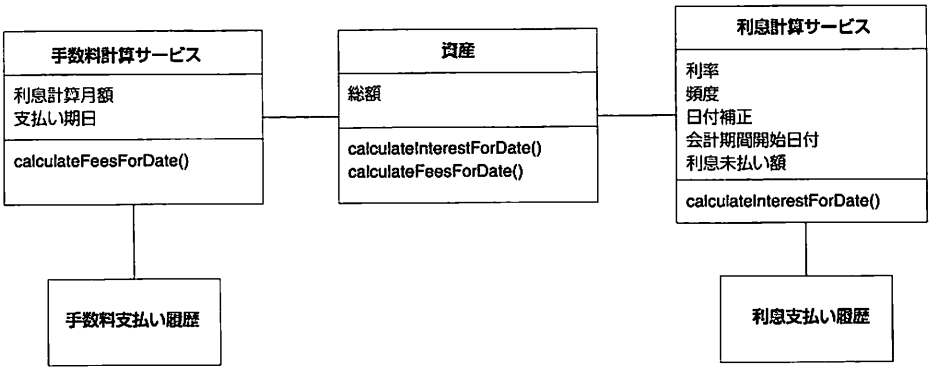


図9.4　ぎこちないモデル

夜間バッチスクリプトは資産ごとに繰り返され、各資産に対して「その日の利息を計算する」（`calculateInterestForDate()`）よう指示していた。そしてこのスクリプトが戻り値（収益総額）を受け取り、その金額に具体的な元帳（ledger）の名前を添えて、会計プログラムの公開インタフェースを提供するサービスに渡していた。すると、このソフトウェアが指定された元帳に総額を記入した。また、スクリプトは同様のプロセスを通じて各資産からその日の手数料を取得し、それを別の元帳に記入していた。

ある開発者は、次第に利息計算が複雑になっていくことに悩んでいた。彼女はこの状況について、より利息計算に適したモデルを見つける好機なのではないかと思い始めた。そこで、仲のよいドメインエキスパートに、この問題領域を掘り下げるのを手伝って欲しいと頼んだのである。

開発者：利息計算サービスが手に負えなくなってきました。

エキスパート：そこは複雑な部分です。まだお知らせしていないケースもありますし。

開発者：わかっています。新しい利息タイプは、別の**利息計算サービス**と置き換えることで追加できます。でも、現在最も悩んでいるのは、スケジュール通りに利息が支払われない特殊ケースなのです。

エキスパート：そうしたことは実際には特殊なケースではありませんよ。利息を支払うタイミングは、非常に柔軟に決められるのです。

開発者：以前、**資産**から**利息計算サービス**を取り出した時は、それで非常に助かりました。さらに分割する必要があるのかもしれません。

エキスパート：わかりました。

開発者：この利息計算についてうまく説明する方法があるのではないかと思うのですが。

エキスパート：どういうことですか？

開発者：えーと、例えば、このアプリケーションは、満期なのに会計期間内に支払われていない利息を追跡しています。この利息に名前はありますか？

エキスパート：うーん、実はそういうやり方はしていません。得られた利息と支払いは全く別の記帳になりますから。

開発者：それでは、この数字は必要ないということですか？

エキスパート：まあ、調べることはあるかもしれませんが、私たちのビジネスのやり方とは違いますね。

開発者：わかりました。では、支払いと利息が区別されているなら、そのようにモデル化するべきでしょう。これでいかがですか？［ホワイトボードにスケッチする］

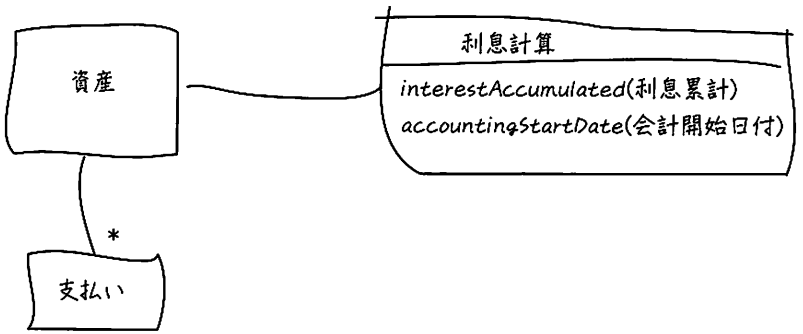


図9.5

エキスパート：たぶん、筋は通っていると思います。しかし、場所を単に移動しただけですよね。

開発者：ただ、こうすると、**利息計算サービス**は得られた利息を追跡するのみで、その数字は**支払い**が別個に保持するようになります。まだあまり単純化していませんが、ビジネスの慣習は、より適切に反映されているでしょう？

エキスパート：ああ、なるほど。利息履歴を持つこともできますか？ **支払い履歴**と同じように。

開発者：はい、それは新機能として要望がありました。けれども、元の設計に追加される予

定でした。

エキスパート：なるほど。いや、利息と**支払い記録**がそのように分離されたのを見て、あなたが利息を分解したのは**支払い記録**のような構成にするためかと思ったのです。発生主義会計については何かご存知ですか？

開発者：いいえ、教えて下さい。

エキスパート：毎日、またはスケジュールで必要な時にはいつでも、利息は発生し、元帳に記帳されます。支払いはそれとは別の方法で記帳されます。ここの集計はちょっとぎこちないですね。

開発者：つまり、「発生（accrual）」の一覧を保持していれば、集計あるいは「記帳」が、必要に応じてできるということですか。

エキスパート：おそらく、発生日付で記帳されるでしょうが、はい、いつでも集計できます。手数料も同様で、もちろん別の元帳に記帳されます。

開発者：実は、利息計算は1日か、またはある期間についてのみ行う方がシンプルになるのです。もう少し、ここについて教えて下さい。これはどうですか？

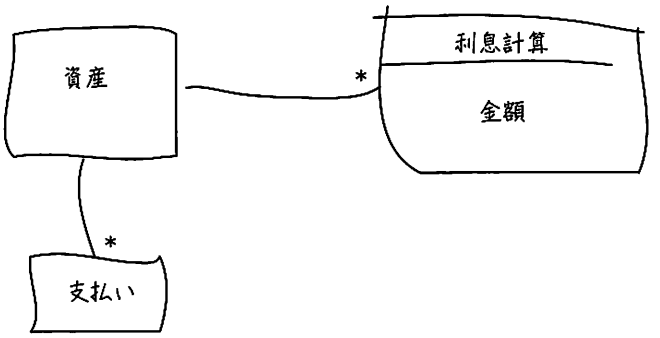


図9.6

エキスパート：ええ、よさそうですね。どうしてこちらの方が簡単なのかはわかりませんが。でも、基本的にどんな資産も、それを価値あるものにしているのは、利息や手数料などのかたちで発生させることができる何かなのです。

開発者：手数料も同じ仕組みとおっしゃいましたね？ それは、えーと、別の元帳に記帳するのですか？

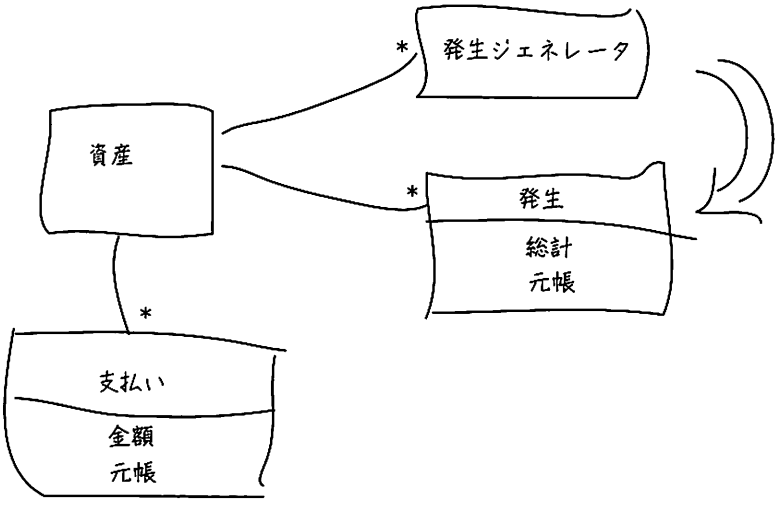


図9.7

開発者：このモデルを使用すると、**利息計算サービス**の中にあった、利息計算というか、発生計算ロジックを、追跡処理から分離できます。今まで気づきませんでしたが、**手数料計算サービス**に重複した部分がずいぶんありましたね。それに、これでいろいろな種類の手数料も簡単に追加できるようになりました。

エキスパート：そうですね。計算は前から正しかったのですが、これで私にも全部わかるようになりました。

各**計算サービス**クラスが設計の他の場所と直接結合していなかったため、このリファクタリングは容易だった。開発者は数時間で、発生という新しい言葉を使用するようにユニットテストを書き直すことができ、翌日遅くには新しい設計を動かすことができた。最終的には次のようになった。

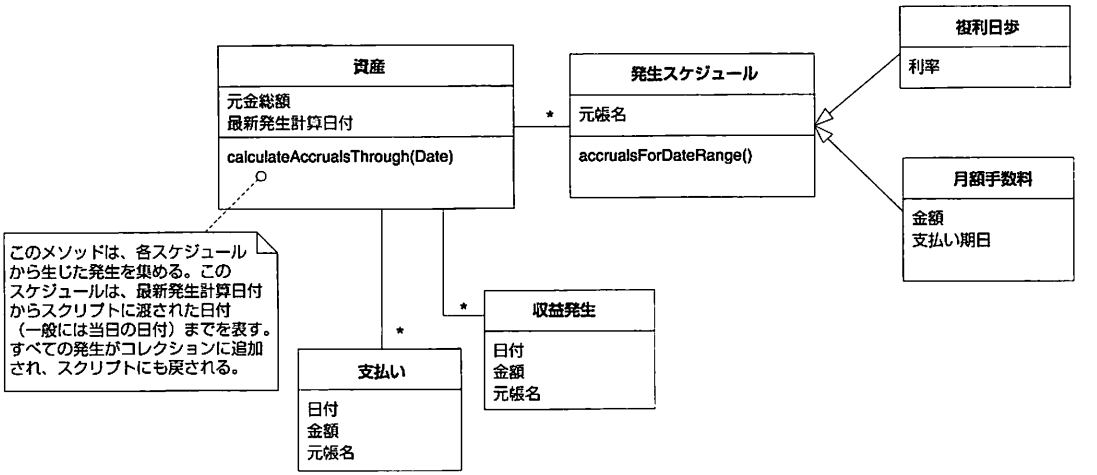


図9.8 リファクタリング後のさらに深いモデル

リファクタリングしたアプリケーションでは、夜間バッチスクリプトが、各**資産**に期間内の発生を計算する（`calculateAccrualsThrough(Date)`）よう指示する。戻り値は**発生**のコレクションで、**発生**の金額はそれぞれ指示された元帳に記入される。

この新しいモデルには長所がいくつかある。変更によって、次のことが起きた。

1. ユビキタス言語が「発生」という用語で豊かになった。
2. 発生が支払いから切り離された。
3. ドメインに関する知識（どの元帳に記入するかなど）が、スクリプトからドメイン層に移された。
4. 手数料と利息がビジネスに適合したかたちで1つになり、コードにおける重複がなくなった。
5. 手数料と利息の新しいバリエーションを追加する際には、わかりやすく、**発生スケジュール**とすればよくなった。

今回は、必要な新しい概念を開発者が掘り出さなければならなかった。利息計算におけるぎこちなさがわかっていたので、より深い答えを見つけるために努力をしたのだ。

幸運だったのは、銀行業のエキスパートという、聡明で意欲的な相棒がいたことである。専門知識を持った人がもっと受け身だったら、もっと間違った場所から議論を始めてしまい、ブレインストーミングの相手として、他の開発者たちにもっと頼ることになっただろう。そうであれば、進展は遅くなっただろうが、それでも進んでいくことはできる。

矛盾について熟考する

ドメインエキスパートはそれぞれ、自分の経験と要求に基づき、物事に対して異なった見方をする。同一人物からもたらされた情報であっても、注意して分析すれば論理的に整合しないこともある。そういう厄介な矛盾は、プログラムの要件を掘り下げる際に常に遭遇するものだが、より深いモデルへの大きな手がかりになり得る。用語法のバリエーションにすぎないものもあれば、誤解に基づいているものもある。しかし、ドメインエキスパートが事実を2つのかたちで記述して、それが矛盾していると思われるのであれば、何か見落としがある。

かつて、天文学者ガリレオがパラドックスを提起した。感覚に基づく証拠は、明らかに地球が静止していることを示している。すなわち、人間が吹き飛ばされたり、落ちていったりすることはない。だが、コペルニクスが説得力のある主張を行い、地球が太陽の周りをかなりの速度で回っているとした。これを両立させれば、自然の仕組みに関する何か深遠なものが明らかになるかもしれなかった。

ガリレオはある思考実験を考案した。騎手が疾走する馬からボールを落とした場合、それはどこに落ちるだろうか？ もちろん、ボールは馬と一緒に移動して、その脚元付近の地面に落ちる。これはまるで馬がじっとしていたかのような動きだ。ここからガリレオが推論したのが、慣性座標系という考え方の初期の形式である。この逆説を解決することが、運動の物理学に関するはるかに有益なモデルへとつながったのだ。

さて、我々が直面する矛盾は、通常はそれほど興味深くはなく、暗に示されているものも、それほど奥深くはない。それでも、これと同じ思考パターンによって、問題ドメインの表層から深い洞察へと突き抜けられることがしばしばある。

すべての矛盾に折り合いをつけることは現実的ではなく、望ましくないことさえある（第14章では、意思決定をどのように行い、結果をどううまく処理するかについて、徹底的に検討する）。しかし、矛盾をそのまま残す場合でも、異なる2つの記述が、両方とも外の同じ現実にどう適用できるのかということは、熟考することで明らかになるかもしれない。

文献を読む

モデルの概念を探す時には、自明のことを見過ごしてはならない。多くの分野において、基本的な概念や社会通念について解説している文献を見つけられる。その場合でも、引き続きドメインエキスパートと協業して問題に関係する場所を蒸留し、それをかみ砕いてオブジェクト指向ソフトウェアに適したものにしなければならない。それでも、本を読むことによって、首尾一貫し深く検討された見方から始められるかもしれない。

例9.3——利息を得る　文献を用いた場合

前述の例にあった投資追跡アプリケーションについて、別のシナリオを考えてみよう。この話も前と同様、特に利息計算サービスの設計が手に負えなくなってきたと開発者が気づくところから始まる。ただし、今度のシナリオでは、ドメインエキスパートは、主要な責任がどこか他のところにあるため、このソフトウェア開発プロジェクトを手伝うことにあまり関心を持っていない。このシナリオの場合、開発者は、エキスパートを頼ってブレインストーミングを行い、欠けている概念を探ることができない。彼女はこの欠けている概念が、表層の下に潜んでいるのではないかと思っているのだ。

代わりに、この開発者は書店へ出かけた。少し見て回って、気に入った会計入門書を見つけ、それに目を通してみた。そして、明確に定義された概念の体系を見つけたのである。なかでも特に彼女の思考をかき立てたものを抜粋する。

> **発生主義会計**　この手法は、たとえ支払われていなくても、収入が生じた時に収入と認める。すべての支出も、それが支払い済みであれ、後日支払う請求であれ、発生した時に示す。税金を含めて、債務は何でも支出として示す。
>
> —Suzanne Caplan『財務会計：MBAや会計士、あるいは博士がいなくても帳簿をつけて財務を管理する方法（Finance and Accounting: How to Keep Your Books and Manage Your Finances Without an MBA, a CPA or a Ph.D)』(Adams Media, 2000)

この開発者は、もう経理の考え方を再発明する必要はなかった。別の開発者とブレインストーミングをした後で、次のようなモデルを考え出した。

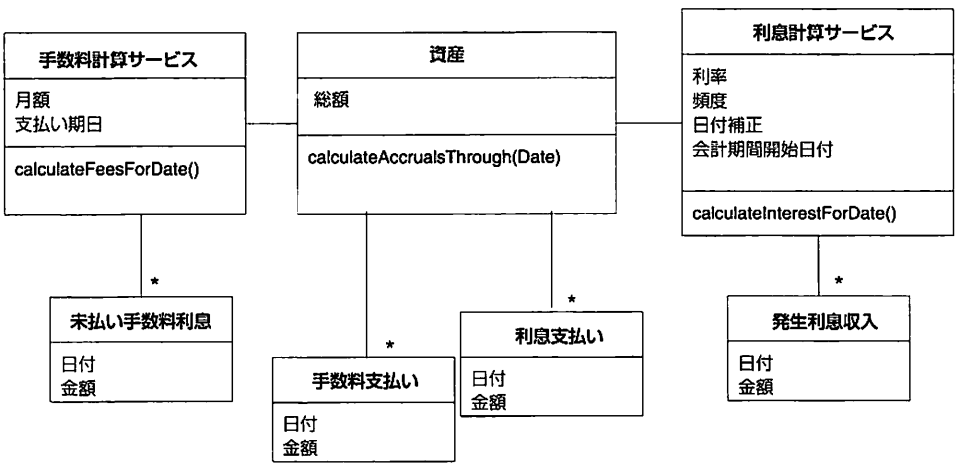


図9.9　文献を読んで学習したことに基づく、やや深まったモデル

彼女には資産が収入ジェネレータだという洞察はないから、まだ計算サービスは複数ある。元帳に関する知識もアプリケーション層に残っていて、それが属しているはずのドメイン層にはない。しかし、彼女は、支払いという最も問題の多い領域を、確かに収入の発生から分離して、モデルとユビキタス言語に「発生」という単語を導入した。以後のイテレーションで、さらに改良されていくだろう。

この開発者がやっとドメインエキスパートと話す機会を得た時、エキスパートは非常に驚いた。自分の手がける仕事にプログラマがわずかでも興味を示したのは、これが初めてだったからである。エキスパートの主要な責務がここになかったせいで、前の例のシナリオとは異なり、エキスパートが彼女と協力して、仕事の手を休め詳しくモデルを調べることはなかった。それでも、この開発者は調べた知識によって的確な質問をすることができたので、それがわかってからは、エキスパートも注意深く彼女の話を聞き、彼女の質問にはすぐに回答できるよう、特別に努力したのだ。

当然のことながら、どちらか一方しかできないわけではない。ドメインエキスパートからの支援が十分にあっても、該当分野の理論を理解するために文献を調べることで得るものはある。ほとんどのビジネスには、会計や財務のレベルまで緻密なモデルはないものの、多くの分野には思想家がいて、そのビジネスに関する一般的な慣習を組織立て、抽象化しているものだ。

また、その開発者が選ぶことのできた別の選択肢としては、該当するドメインの開発経験があるソフトウェアの専門家が書いた文献を読むという方法がある。例えば、『アナリシスパターン』(Fowler 1997) の第6章を読んでいたら、いいか悪いかは別として、全く異なる方向へ進んだはずだ。そうした読書では既製の解決策は得られなかっただろう。ただ、開発者自身の実験に新しい出発点をいくつか準備できたかもしれず、そこに、過去に同じ分野を探究した人々の蒸留された経験も加わったはずだ。車輪の再発明を避けられたに違いない。第11章「アナリシスパターンの適用」では、この選択肢についてさらに掘り下げる。

何度でも挑戦すること

これまで取り上げてきた例は、どれだけの試行錯誤が行われたかを伝えていない。モデルを作ってみることができるくらい、わかりやすくて有用なものを見つけるためには、会話に出てきた手がかりを、片手では数えきれないくらい辿らなければならなかっただろう。それも、後で最低1回は取り替えることになる。その後の経験と知識のかみ砕きによって、より優れた考え方が生み出されるからである。モデラや設計者は自身の考えに固執してはいられないのだ。

このように方向性を変えることは、単なる無駄な努力ではない。変更するたびに、より深い洞察がモデルに埋め込まれる。リファクタリングをするたびに、設計はよりしなやかになり、次回の変更がより容易になる。曲げる必要があると判明したところで曲げられるようになるのだ。

いずれにしても他に方法はないのだ。実験することで、使えるものと使えないものがわかる。設計における誤りを避けようとすれば、結果としてできるものの品質は低くなる。基になる経験が少ないからだ。しかも、簡単な実験を連続して行うよりも時間がかかりかねない。

それほど明白でない概念をモデル化する方法

オブジェクト指向パラダイムによって、我々はある種の概念を探し、考え出すようになった。「発生」のように非常に抽象的なものも含めて、物（thing）が、ほとんどのオブジェクトモデルにとっての骨子となるのであり、そこに物によって行われるふるまいがつけ加わる。これらが、オブジェクト指向の入門書の言う「名詞と動詞」だ。しかし、それとは別の重要な概念のカテゴリも、モデルで明示することができる。

オブジェクトを学び始めた頃の私には理解しにくかった、3つのカテゴリについて議論することにしよう。これらを1つ習得するたびに、私の設計は鋭くなっていった。

明示的な制約

制約は、モデルの概念において、とりわけ重要なカテゴリである。こうした制約は暗黙のうちに現れることが多いので、明示的に表現することによって、設計が大幅に向上するかもしれない。

時には、制約の自然な落ち着き場所がオブジェクトやメソッドのこともある。「バケツ」（Bucket）オブジェクトは、最大容量以上は入らないという不変条件を保証しなければならない。

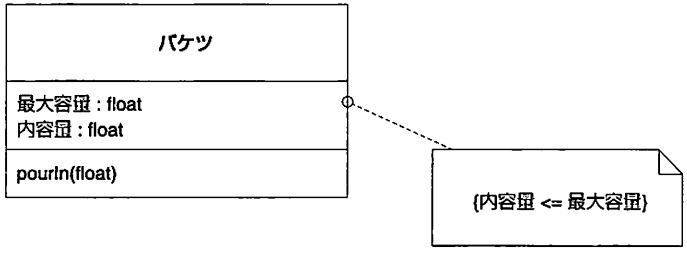


図9.10

こうした単純な不変条件を強制するには、内容量を変更できる各操作で、条件つきロジックを使用すればよい。

```
/**
 * バケツクラス.
 */
class Bucket {
   // 最大容量
   private float capacity;
   // 内容量
   private float contents;

   /**
    * 注ぐ.
    * @param addedVolume 加えられた量
    */
   public void pourIn(float addedVolume) {
      // 内容量と加えられた量の合計が最大容量を超えていた場合
      if (contents + addedVolume > capacity) {
         contents = capacity;
      } else {
         contents = contents + addedVolume;
      }
   }
}
```

このロジックは非常にわかりやすいから、ルールも明白である。しかし、より複雑なクラスでは、この制約が見えなくなってしまうことが容易に想像できる。これを別個のメソッドに括り出し、制約の意義を明確かつ明示的に表現する名前をつけることにしよう。

```
class Bucket {
   private float capacity;
   private float contents;
   public void pourIn(float addedVolume){
      float volumePresent = contents + addedVolume;
```

```
        contents = constrainedToCapacity(volumePresent);
    }

    /**
     * 最大容量以下に制限する.
     * @param 入れられた量
     * @return 入れることのできた量
     */
    private float constrainedToCapacity(float volumePlacedIn) {
        if (volumePlacedIn > capacity) return capacity;
        return volumePlacedIn;
    }
}
```

どちらのコードでも制約は強制されるが、後者の方が、モデルとの間により明白な関係を持っている（モデル駆動設計の基本的な要件）。このように非常に単純なルールなら元のかたちでも理解できるが、強制すべきルールが複雑になれば、あらゆる暗黙的な概念と同様に、ルールが適用されるオブジェクトや操作を圧倒し始める。制約を独立したメソッドに分解することで、そのメソッドに意図の明白な名前をつけて、設計上でも制約を明示できるようになる。こうすることで制約に名前が与えられ、制約について議論できるようになる。このアプローチによって、制約に変化する余地も与えられる。例に挙げたものよりも複雑なルールでは、メソッドは呼び出し側（この場合は`pourIn()`メソッド）よりもすぐに長くなってしまうだろう。こうして制約を独立させておけば、呼び出し側をシンプルで自分のやるべきことに専念できる状態に保ちながら、必要に応じて制約の側を複雑にすることができる。

この独立したメソッドによって制約を拡大させる余地が与えられるが、制約を1つのメソッドにすんなりと収めきれなくなることも多々ある。あるいは、メソッドが単純なままでも、オブジェクトの主要な責務には必要ない情報を要求することもある。そのルールが落ち着くべき適当な場所が、既存のオブジェクトにはないかもしれないのだ。

次に挙げるのは、制約を実装するオブジェクトの設計が、制約によってゆがめられていることを示す警告である。

1. 制約を評価するために、本来であればオブジェクトの定義に合わないデータが必要になる。
2. 関連するルールが複数のオブジェクトに出てきて、コードを重複させなければならなくなったり、本来であれば同じ系統でないオブジェクトを継承しなければならなくなったりする。
3. 設計や要求に関する多くの会話が制約をめぐって行われるが、実装では制約が手続き型のコードの中に隠されてしまっている。

制約によってオブジェクトの基本的責務があいまいになっている場合、あるいは、制約がドメインではっきりと見てとれるのにモデルではそうでない場合、制約を括り出して明示的に1オブジェクトにするか、オブジェクトの集まりとそれらの関係性としてモデル化してもよい（このテーマを詳細かつ正式に近いかたちで扱っている例は『The Object Constraint Language: Precise Modeling with UML』（Warmer and Kleppe 1999）にある）。

例9.4——再考：オーバーブッキングポリシー

第1章では、一般的な輸送業の慣習を用いて考察した。その慣習とは、運送できる量より10%多い貨物を予約するというものだ（輸送会社が経験から学んだのは、このオーバーブッキングによって土壇場のキャンセルを埋め合わせられるので、船舶がほぼ満載で運行できるということである）。

運行と**貨物**との関連についての制約を、図とコードの両方において明示的にするため、この制約を表す新しいクラスが追加された。

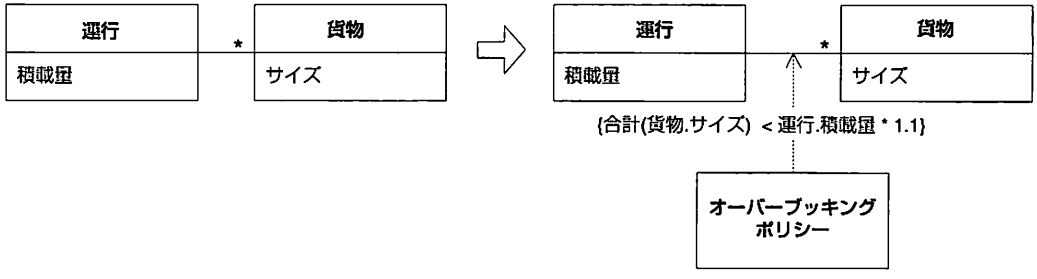


図9.11 ポリシーを明示的にするためリファクタリングされたモデル

この例について、コードと根拠をすべて復習するには、例1.1を参照。

ドメインオブジェクトとしてのプロセス

まず、我々はプロシージャをモデルの主要な側面に**したくない**のだ、ということで意見を一致させよう。オブジェクトは、手続きをカプセル化し、その代わりに目標や意図について我々に考えさせるために存在する。

ここで述べるのは、ドメインに存在するプロセスについてである。これはモデルの中に表現されなければならない。このようなプロセスが姿を現すと、オブジェクト設計はぎこちなくなりがちだ。

本章の最初の例では、貨物の経路を選択する輸送システムについて説明した。この経路選択プロセスは、ビジネス上、意味のあるものだった。サービスは、そういうプロセスを

明示的に表現しつつ、極端に複雑なアルゴリズムを引き続きカプセル化する方法の1つである。

プロセスの実行方法が複数ある場合、アルゴリズムそのものやアルゴリズムの主要な部分を、独立したオブジェクトにするというアプローチもある。プロセス間での選択が、これらオブジェクトの間での選択となり、各オブジェクトが、それぞれが別々のストラテジーを表現する（第12章で、ドメインにおけるストラテジーの使用について詳細に見ていく）。

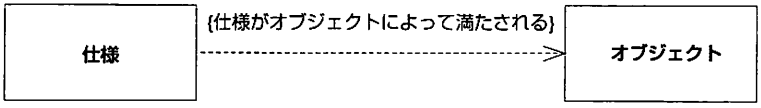
明示すべきプロセスと隠蔽すべきプロセスを区別する鍵は単純だ。ドメインエキスパートが話題に上げるものか、コンピュータプログラムにおける仕組みの一部にすぎないか、である。

制約とプロセスは、オブジェクト指向言語でプログラミングしている時にすぐには思い浮かばない、2大モデル概念である。しかし、いったんこれらをモデル要素として考えるようになると、設計は実に鋭くなる。

概念のカテゴリで役に立つものの中には、より限定的なものもある。より具体的でありながらきわめて一般的なもので本章を締めくくることにしよう。仕様を用いることで、ある種のルールを表現し、それらのルールを条件つきロジックから解放し、モデルの中で明示することが、簡潔にできるようになる。

私はこの仕様をMartin Fowlerと共同で開発した（Evans and Fowler 1997）。概念がシンプルであるのに反して、その使い方と実装は巧妙であるため、本節では詳細について多く書いてある。第10章では、このパターンを拡張してさらに議論を進める。次に続くパターンの最初の説明を読めば、実際にパターンを適用してみようと思うまで、「仕様の適用と実装」（229ページ）には目を通すだけでも構わない。

仕様（SPECIFICATION）



あらゆる種類のアプリケーションにおいて、ブール値による評価を行うメソッドは、実際には小さいルールの一部であるように見える。ルールがシンプルである限り、我々はanIterator.hasNext()や、anInvoice.isOverdue()（請求書の期限が超過しているか）といった評価メソッドで処理する。**請求書**（**Invoice**）クラスの場合、isOverdue()のコードはルールを評価するアルゴリズムである。例えば、次のようなものだろう。

```
public boolean isOverdue() {
   Date currentDate = new Date();
   return currentDate.after(dueDate);
}
```

しかし、すべてのルールが、これほど単純とは限らない。同じ**請求書**クラスで、別のルールであるanInvoice.isDelinquent()（請求書が延滞しているか）は、おそらく**請求書**の期限が超過しているかどうかの評価で始まるが、その後にも処理は続くだろう。支払い猶予期間に関する方針は、顧客の口座の状態に応じて変わるかもしれない。延滞請求書には、2回目の通知を送る準備をするものもあれば、回収機関へ送る準備をするものもある。顧客の支払い記録や、いろいろな取り扱い品目に対する企業の方針などのせいで、**請求書**の持つ支払いの要求という明確な性質は、ルールを評価するコードの塊の中ですぐに見失われてしまう。さらにこの**請求書**は、支払いの要求という基本的な意味をサポートしていないドメインクラスとサブシステムに対しても、あらゆる種類の依存関係を持つようになっていく。

この時点で、**請求書**クラスを守るため、開発者はルール評価コードをリファクタリングして、アプリケーション層に入れることが多い（この場合は料金回収アプリケーション）。こうしてルールがすべてドメイン層から分離され、魂の抜けたデータオブジェクトが後に残る。このオブジェクトは、ビジネスモデルに本来備わっていたルールを表現することがない。こうしたルールはドメイン層に留めておく必要があるが、評価されるオブジェクト（この場合は**請求書**）には馴染まない。そればかりでなく、評価メソッドは条件文で肥大しているから、ルールを読み取ることが困難になる。

論理プログラミングパラダイムに基づく開発者なら、この状況を別の方法で処理するか

もしれない。こういうルールは**述語**（predicate）として表現される。述語とは、「真」か「偽」を評価する関数のことで、「AND」と「OR」などの演算子を使用して結合することで、より複雑なルールを表現できる。述語を使うことによって、ルールを明示的に宣言し、**請求書**と一緒に使用できる。ただし、論理パラダイムを採用していればの話であるが。

こうした状況を受けて、オブジェクトのかたちで論理的ルールを実装しようという試みが数多くなされてきた。そうした試みの中には、きわめて洗練されたものもあれば、未熟なものもあった。大掛かりなものも控えめなものもあった。有意義だと判明したものがある一方で、失敗した実験として脇に押しやられたものもある。プロジェクトを頓挫させてしまったものもわずかながら存在する。1つはっきりしていることがある。これは発想としてはきわめて魅力的であるのだが、論理をオブジェクトのかたちで完全に実装するのも、やはり大きな仕事だということである（結局、論理プログラミングはそれ自体がまるごとモデリングと設計のパラダイムなのである）。

ビジネスルールは、明らかな*エンティティ*や*値オブジェクト*の責務のどれとも合致しないことがしばしばあり、その多様性と組み合わせによって、ドメインオブジェクトの基本的な意味を圧倒しかねない。しかし、ルールをドメイン層から取り出してしまうのはさらに悪い。ドメインコードがモデルを表現しなくなってしまうからである。

論理プログラミングは、「述語」と呼ばれる、独立した結合可能なルールオブジェクトの概念を提供するが、この概念をオブジェクトで完全に実装するのは面倒である。また、あまりにも一般的なため、より特殊な設計と同様に、多くの意図を伝えることもない。

幸運なことに、論理プログラミングから大きな恩恵を受けるにあたって、それを本当にすべて実装する必要はない。我々が用いるルールのほとんどは、いくつかの特殊なケースに収まる。だから、述語という概念を借用して、評価の結果ブール値を返す特別なオブジェクトを作成すればよい。手に負えなくなった評価メソッドは、うまく拡張して独立オブジェクトにする。それらは、独立した値オブジェクトに括り出せる、小さい真偽評価メソッドなのだ。この新しいオブジェクトは、他のオブジェクトを評価して、述語がそのオブジェクトに対して「真」かどうかを調べることができる。

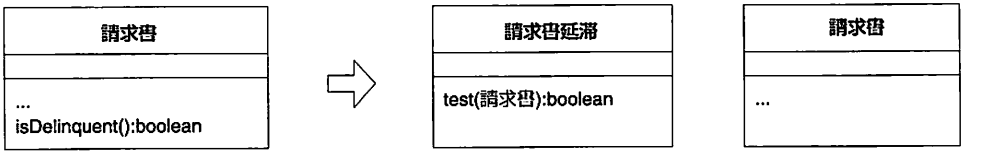


図9.12

別の言い方をすれば、この新しいオブジェクトが**仕様**なのだ。仕様は、他のオブジェクトの状態に関する制約を述べるものだ。この時、他のオブジェクトは、存在していてもいなくてもよい。仕様の用途は多数あるが、最も基本的な概念を伝えているのは、どんなオブジェクトでも評価して、定義された基準を満たしているかどうかを調べるという使い方である。

それゆえ：

特殊な目的を持った述語的な値オブジェクトを明示的に作成すること。仕様とは、あるオブジェクトが何らかの基準を満たしているかどうかを判定する述語である。

延滞請求書の例で見たように、多くの仕様は特殊な目的を持った単純な評価メソッドである。ルールが複雑な場合には、この概念を拡張して、単純な仕様を結合することができる。これは、述語が論理演算子で結合されるのと同じである（このテクニックについては次章で議論する）。基本的なパターンは変わらないが、単純なモデルからより複雑なモデルへの道が開かれる。

延滞請求書の例をモデル化するのに用いる仕様は、延滞とはどういう意味であるかを記述しており、それを使ってあらゆる**請求書**を評価して判定することができる。

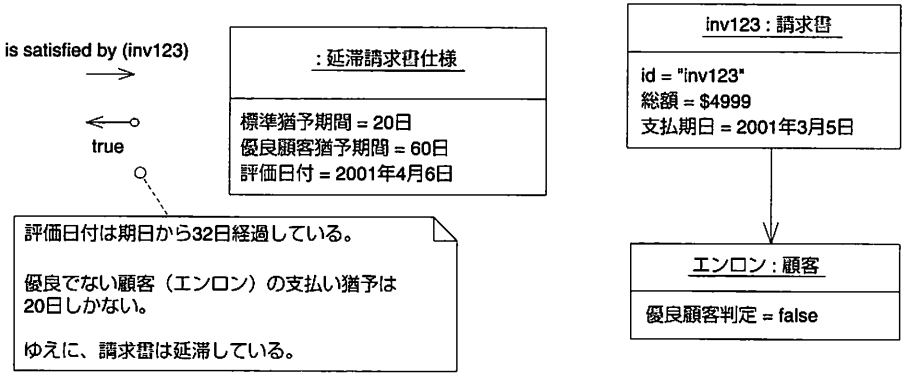


図9.13 **仕様**として分解されたより精巧な延滞ルール

仕様により、ルールはドメイン層に維持される。ルールはれっきとしたオブジェクトなのだから、設計はモデルをより明示的に反映させることができる。ファクトリは、仕様を構成する上で、顧客の口座や企業方針データベースといった、他の情報源からの情報を使用できる。これらの情報源に**請求書**から直接アクセスできるようにしてしまうと、支払いの要求（**請求書**の基本的な責務）とは無関係な方法で、オブジェクト同士を結合させることになる。この場合では、**延滞請求書仕様**が生成され、**請求書**を評価するのに使用されて

から削除されるので、具体的な評価日付が直に組み込まれている。これはうまい簡略化だ。処理に必要となる情報は、仕様に対して単純明快な方法で与えられるのである。

＊　＊　＊

仕様の基本概念は非常にわかりやすく、ドメインモデリングの問題について考える上で役に立つ。しかし、モデル駆動設計には、その概念を表現することもできる効果的な実装が必要である。うまく実装するためには、パターンをどう適用するかについて、もう少し掘り下げなければならない。ドメインパターンは、UML図のための気の利いた考え方に留まるものではない。モデル駆動設計を保ちつつプログラミングの問題を解決する方法なのである。

あるパターンをうまく適用すれば、ドメインモデリングにおけるある種の問題にどう取り組むべきか、ということについての思想の体系全体を活用できるようになり、効果的な実装を見つけるために費やされた長年の経験から恩恵を受けることができる。これから行う仕様に関する議論には、機能に関する複数の選択肢や、実装へのアプローチに関する詳細が多く含まれている。パターンは手順書ではない。解決策を組み上げるにあたり、経験に基づいて始められるようにし、さらに、自分が何をしているかを述べるための言語を提供するのである。

続く節は、初めて読む際には、主要な概念に目を通す程度でかまわない。後日、具体的な状況に遭遇した時に、ここに立ち戻って、詳細な議論の中に表された経験を利用すればよい。そうすれば、自分の問題に対する解決策を考え出せるだろう。

仕様の適用と実装

仕様の価値の多くは、全く異なるように見えるアプリケーションの機能を統一することにある。以下に挙げる3つの目的のうち、1つでも当てはまれば、オブジェクトの状態を定義する必要があるだろう。

1. オブジェクトを検証して、何らかの要求を満たしているか、何らかの目的のための用意ができているかを調べる。
2. コレクションからオブジェクトを選択する（期限が超過した請求書を問い合わせる場合など）。
3. 何かの要求に適合する新しいオブジェクトの生成を定義する。

検証、選択、要求に応じた構築という、これら3つの用途は、概念レベルでは同じものである。仕様のようなパターンがなければ、同一のルールがさまざまな外見で、場合によっては矛盾した形式で出てきてしまうかもしれない。それにより概念の統一性が失われかねない。仕様パターンを適用することで、実装が異なってしまう場合でも、一貫性のあ

るモデルを使用できるようになる。

検 証

仕様の最も単純な用途は検証であり、仕様という概念を最も直接的に示す利用法である。

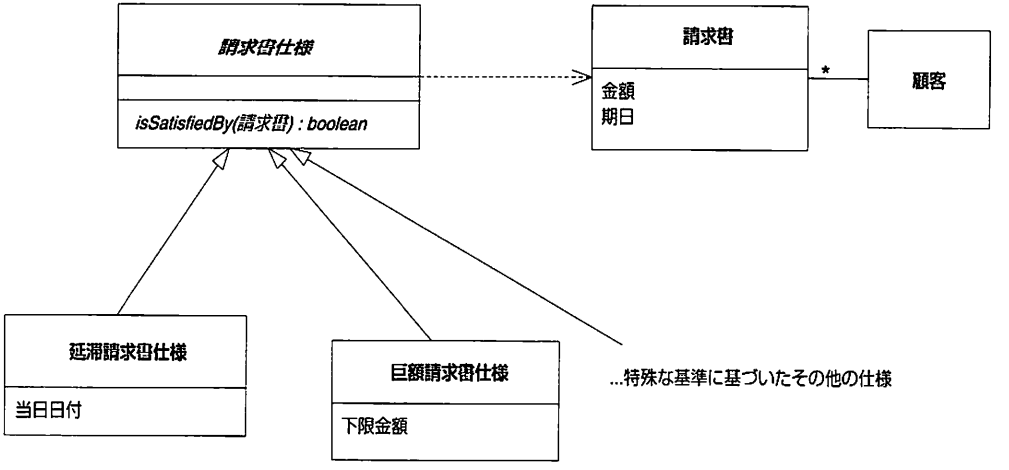


図9.14 検証のために**仕様**を適用するモデル

```
/**
 * 延滞請求書仕様.
 */
class DelinquentInvoiceSpecification extends InvoiceSpecification {
   // 当日日付
   private Date currentDate;
   // あるインスタンスは1つの日付に対して使用され、その後削除される

   public DelinquentInvoiceSpecification(Date currentDate) {
      this.currentDate = currentDate;
   }

   /**
    * 仕様を満たしているかを判定する.
    * @param candidate 判定対象の請求書
    * @return 対象が仕様を満たしていれば true
    */
   public boolean isSatisfiedBy(Invoice candidate) {
      // 猶予期間
      int gracePeriod =
         candidate.customer().getPaymentGracePeriod();
      // 最終期限
      Date firmDeadline =
         DateUtility.addDaysToDate(candidate.dueDate(),
```

```
                gracePeriod);
        return currentDate.after(firmDeadline);
    }

}
```

ここで、セールスマンが支払いを延滞している顧客を表示するたびに、赤い目印を表示する必要があるとしよう。そのためには、クライアントクラスに次のようなメソッドを書くだけでよい。

```
/**
 * 口座に延滞請求書があるかを判定する.
 * @param customer 顧客
 * @return 延滞した請求書があれば true
 */
public boolean accountIsDelinquent(Customer customer) {
    Date today = new Date();
    Specification delinquentSpec =
        new DelinquentInvoiceSpecification(today);
    Iterator it = customer.getInvoices().iterator();
    while (it.hasNext()) {
        Invoice candidate = (Invoice) it.next();
        // 延滞仕様が候補によって満たされるか
        if (delinquentSpec.isSatisfiedBy(candidate)) return true;
    }
    return false;
}
```

選択（もしくは問い合わせ）

検証は個々のオブジェクトを評価して、何らかの基準を満たしているかどうかを調べるものだが、おそらくそれは、クライアントが評価結果に応じてふるまえるようにするためである。他にも一般的な要求としては、何らかの基準に基づいて、オブジェクトのコレクションから一部を選び出すというものもある。同じ仕様の概念がここにも適用できるが、実装のポイントは異なる。

延滞**請求書**を持つ顧客をすべて一覧化したいという、アプリケーションの要件があるとする。理論的には、先に定義した**延滞請求書仕様**がやはり使えるが、実際には実装を変更する必要があるだろう。**概念**は同じであることを実証するため、まずは、**請求書**の数が少なくて、すでにメモリに入っているものと想定しよう。この場合であれば、検証のために開発した直接的な実装を使うことがまだできる。**請求書リポジトリ**には、仕様に基づいて**請求書**を選択する汎用的なメソッドがあるだろう。

```
/**
 * 仕様を満たす請求書を選択する.
 * @param spec 請求書仕様
 * @return 仕様を満たす請求書のセット
```

```
 */
public Set selectSatisfying(InvoiceSpecification spec) {

   Set results = new HashSet();
   Iterator it = invoices.iterator();
   while (it.hasNext()) {
      Invoice candidate = (Invoice) it.next();
      if (spec.isSatisfiedBy(candidate)) results.add(candidate);
   }

   return results;
}
```

したがって、クライアントはコードを1行書くだけで、延滞**請求書**のコレクションを得られる。

```
Set delinquentInvoices = invoiceRepository.selectSatisfying(
   new DelinquentInvoiceSpecification(currentDate));
```

この1行のコードは、操作の背景にある概念を規定している。もちろん、**請求書**オブジェクトはメモリの中にはないだろう。**請求書**は何千とあるかもしれない。典型的なビジネスシステムの場合、おそらく、データは関係データベースに入っている。そして、これまでの章で指摘した通り、モデルの焦点は、こうした他の技術と交差するところで失われがちなのである。

関係データベースには強力な検索能力がある。この問題を効率的に解決するために関係データベースの持つ能力を利用しながら、一方で、仕様のモデルを維持するにはどうすればよいだろうか？ モデル駆動設計においては、モデルは実装と歩調を合わせなければならない。だが、モデルの意味を忠実にとらえていれば、どんな実装でも自由に選択してよい。幸運なことに、SQLは仕様を作成する上で、きわめて自然な方法だ。

次に挙げる単純な例では、クエリが検証ルールと同じクラスにカプセル化されている。あるメソッドが1つ、**請求書仕様**に追加され、**延滞請求書仕様**サブクラスでそのメソッドが実装されている。

```
public String asSQL() {
   return
      "SELECT * FROM INVOICE, CUSTOMER" +
      "  WHERE INVOICE.CUST_ID = CUSTOMER.ID" +
      "  AND INVOICE.DUE_DATE + CUSTOMER.GRACE_PERIOD" +
      "      < " + SQLUtility.dateAsSQL(currentDate);
}
```

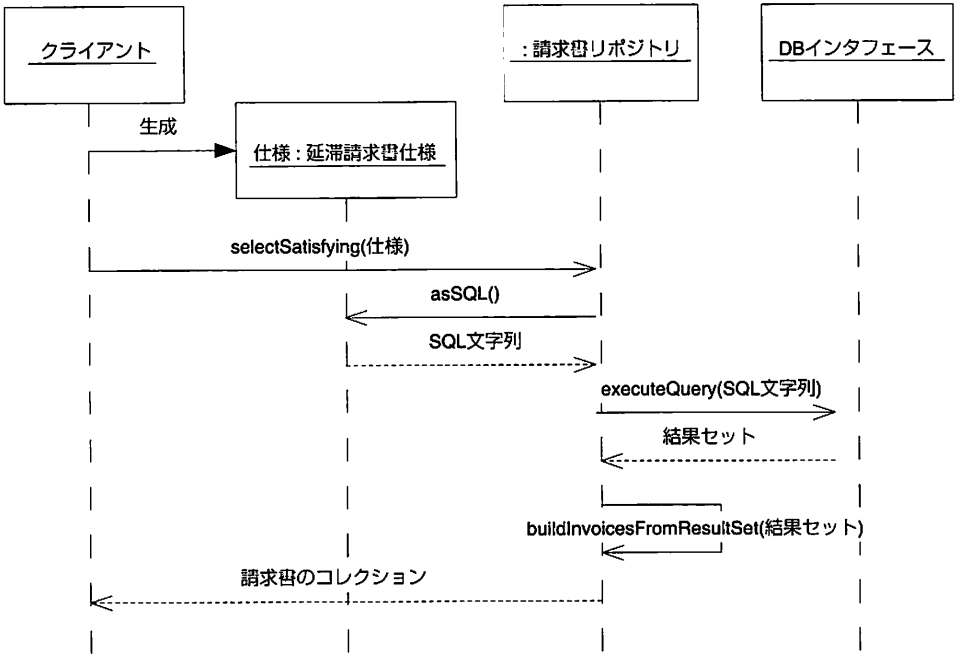


図9.15　リポジトリと仕様の間の相互作用

仕様はリポジトリとスムーズに整合する。リポジトリとは、ドメインモデルの構成要素となる仕組みであり、ドメインオブジェクトに対するクエリアクセスを提供し、データベースへのインタフェースをカプセル化するものである（図9.15参照）。

さて、この設計にはいくつか問題がある。何より重要なのは、テーブル構造の詳細がドメイン層に流出していることだ。本来であれば、こうした詳細は、ドメインオブジェクトを関係テーブルに結びつけるマッピング層に隔離されていなければならない。その情報をここで暗黙のうちに重複させてしまうと、**請求書**と**顧客**オブジェクトを修正したり、保守したりするのが困難になりかねない。マッピングに対するあらゆる変更を、複数の場所で追跡しなければならなくなるからだ。ただし、この例はルールを1か所だけで保持する方法を簡単に示したものである。O/Rマッピングフレームワークの中には、こういったクエリをモデルオブジェクトと属性というかたちで表現する手段を提供して、インフラストラクチャ層で実際のSQLを生成するものもある。こうしたフレームワークを使えば、ルールを1か所だけで保持しつつ、マッピングの知識をドメイン層に流出させずに済むだろう。

インフラストラクチャの助けが望めない場合は、表現力豊かなドメインオブジェクトからSQLを取り除くために、リファクタリングして、特殊なクエリメソッドを**請求書リポジトリ**に追加すればよい。ルールがリポジトリに埋め込まれるのを避けるためには、クエリ

をより汎用的なかたちで表現しなければならない。つまり、ルールをとらえはしないものの、ルールを実現するために、コンテキストに応じて結合したり、配置したりできる方法が求められるのだ（この例ではダブルディスパッチを使用する）。

```
/**
 * 請求書リポジトリ.
 */
public class InvoiceRepository {

    /**
     * 支払い猶予期間がすぎたものを選択する.
     * @param aDate 評価日付
     */
    public Set selectWhereGracePeriodPast(Date aDate) {
        // これはルールではなく、特殊なクエリにすぎない
        String sql = whereGracePeriodPast_SQL(aDate);
        ResultSet queryResultSet =
            SQLDatabaseInterface.instance().executeQuery(sql);
        return buildInvoicesFromResultSet(queryResultSet);
    }

    /**
     * 支払い猶予期間超過Where句を持つSQL.
     * @param aDate 評価日付
     * @return SQL文
     */
    public String whereGracePeriodPast_SQL(Date aDate) {
        return
            "SELECT * FROM INVOICE, CUSTOMER" +
            "  WHERE INVOICE.CUST_ID = CUSTOMER.ID" +
            "  AND INVOICE.DUE_DATE + CUSTOMER.GRACE_PERIOD" +
            "      < " + SQLUtility.dateAsSQL(aDate);
    }

    /**
     * 仕様を満たす請求書を選択する.
     * @param spec 請求書仕様
     * @return 仕様を満たす請求書のセット
     */
    public Set selectSatisfying(InvoiceSpecification spec) {
        return spec.satisfyingElementsFrom(this);
    }
}
```

請求書仕様にあったasSql()メソッドはsatisfyingElementsFrom(Invoice-Repository)で置き換えられ、これは**延滞請求書仕様**で次のように実装される。

```
/**
```

```
 * 延滞請求書仕様.
 */
public class DelinquentInvoiceSpecification {
    // 延滞請求書仕様の基本的なコードがここに入る

    /**
     * 条件を満たす要素を取得する.
     * @param repository 請求書リポジトリ
     * @return 条件を満たした請求書のセット
     */
    public Set satisfyingElementsFrom(
            InvoiceRepository repository) {
        // 延滞ルールは次のように定義される：
        //    "現日付を基点として経過した支払い猶予期間"
        return repository.selectWhereGracePeriodPast(currentDate);
    }
}
```

こうすることでSQLはリポジトリに置かれ、その一方で、どのクエリを使うべきかが仕様によって制御される。ルールは、それほどうまく仕様に集められてはいないが、延滞を構成するもの（すなわち、超過した支払い猶予期間）に関する本質的な宣言は揃っている。

このリポジトリには、ほぼこの場合でしか使用されないだろうと思われる、非常に特殊なクエリができた。それでもかまわないのだが、延滞請求書の数に対する、期限を超過した**請求書**の数の比率によっては、中間的な解決策が考えられるかもしれない。つまり、リポジトリのメソッドをより汎用的に保った方が、優れた性能を出しつつ、仕様をより意図の表現されたものに保てるかもしれないのだ。

```
/**
 * 請求書リポジトリ.
 */
public class InvoiceRepository {

    /**
     * 期限日を超過した請求書を選択する.
     * @param aDate 評価日付
     * @return 期限日を超過した請求書のセット
     */
    public Set selectWhereDueDateIsBefore(Date aDate) {
        String sql = whereDueDateIsBefore_SQL(aDate);
        ResultSet queryResultSet =
            SQLDatabaseInterface.instance().executeQuery(sql);
        return buildInvoicesFromResultSet(queryResultSet);
    }

    /**
     * 期限日超過 Where 句を持つ SQL.
     * @param aDate 評価日付
```

```
     * @return SQL文
     */
    public String whereDueDateIsBefore_SQL(Date aDate) {
        return
            "SELECT * FROM INVOICE" +
            "  WHERE INVOICE.DUE_DATE" +
            "        < " + SQLUtility.dateAsSQL(aDate);
        }

    /**
     * 仕様を満たす請求書を選択する.
     * @param spec 請求書仕様
     * @return 仕様を満たす請求書のセット
     */
    public Set selectSatisfying(InvoiceSpecification spec) {
        return spec.satisfyingElementsFrom(this);
    }
}

/**
 * 延滞請求書仕様.
 */
public class DelinquentInvoiceSpecification {
    // 延滞請求書仕様の基本的なコードがここに入る

    /**
     * 条件を満たす要素を取得する.
     * @param repository 請求書リポジトリ
     * @return 条件を満たした請求書のセット
     */
    public Set satisfyingElementsFrom(
            InvoiceRepository repository){
        Collection pastDueInvoices =
            repository.selectWhereDueDateIsBefore(currentDate);

        Set delinquentInvoices = new HashSet();
        Iterator it = pastDueInvoices.iterator();
        while (it.hasNext()) {
            Invoice anInvoice = (Invoice) it.next();
            // ある請求書がこの仕様を満たしているか
            if (this.isSatisfiedBy(anInvoice))
                delinquentInvoices.add(anInvoice);
        }
        return delinquentInvoices;
    }
}
```

このコードでは、性能問題が起きるかもしれない。より多くの**請求書**を取得して、メモリ上でその中から選択しなければならないからである。責務をよりうまく括り出すことと比較して、これが割に合うかどうかは、完全に状況次第だ。仕様とリポジトリの間の相互作用を実装して、開発プラットフォームを活用しながら、基本的な責務をしかるべき場所に置いたままにしておく方法は数多くある。

時には性能改善のため、あるいは、多くの場合セキュリティ強化のために、クエリをストアドプロシージャとしてサーバに実装することもある。その場合、仕様はストアドプロシージャで許可されたパラメータしか渡すことができない。これらを踏まえても、さまざまな実装の間にモデルの違いはない。実装は、モデルによって特別に制約されない限り、自由に選択できるのだ。その代償は、クエリの作成や保守がより面倒になるというかたちで現れる。

ここでの議論は、仕様をデータベースと組み合わせるという課題の表層をなぞっているにすぎないので、起こり得る考慮事項をすべて網羅しようとはしていない。私は、どういった選択をしなければならないかを少し紹介したかっただけなのだ。Mee と Hieatt が、リポジトリを仕様と合わせて設計することに伴う技術的な問題のいくつかを『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』（Fowler 2002）で議論している。

要求に応じた構築（生成）

米国国防省で新しい戦闘機が必要になると、職員が仕様書を作成する。この仕様書が定めているのはおそらく、戦闘機の最高速度がマッハ2に到達すること、航続距離が約2900キロメートルあること、コストが5000万ドル以下であることなどであろう。しかし、どれほど詳細に書かれていても、仕様書は機体の設計ではなく、ましてや飛行機そのものではない。航空宇宙産業の一企業が仕様書を受け取り、それに基づいて1つか複数の設計を行う。競合他社も別の設計を作り出すだろうが、そのどれもが、おそらく基となる仕様を満たしている。

多くのコンピュータプログラムは何かを生成するが、そういうものには定義が必要である。ワープロドキュメントに絵を挿入すると、テキストはその周囲を回り込む。その際にあなたが定義するのは、絵の挿入場所と、おそらくテキストの回り込みのスタイルだ。ページ上での文章の正確な配置は、仕様に合ったかたちでワープロが行う。

初めはわかりにくいかもしれないが、これも、検証と選択に適用したのと同じ仕様の概念である。まだ存在していないオブジェクトの条件を定義しているのだ。ただし、実装は全く異なるだろう。この仕様は、問い合わせのように、すでに存在するオブジェクトに対するフィルタではないし、検証のように、既存のオブジェクトに対する評価でもない。今回は、全く新しいオブジェクトやオブジェクトの集合が、仕様を満たすように作成されるか、再構成されるのだ。

仕様を用いなくても、ジェネレータを書いて、そこに必要なオブジェクトを生成する手続きか命令一式を与えることができる。その場合、コードは、ジェネレータのふるまいを暗黙的に定義することになる。

その代わりに、ジェネレータのインタフェースを記述的な仕様のかたちで定義すれば、ジェネレータの生成物を明示的に制約できる。このアプローチにはいくつかの利点がある。

- ジェネレータの実装がインタフェースから分離される。仕様は出力に対する要求を宣言するが、どうやってその結果を出力するかについては定義しない。
- このインタフェースは明示的にルールを伝えるので、開発者は操作の詳細をすべて理解していなくても、ジェネレータに何を期待すべきかがわかる。手続き的に定義されたジェネレータだと、そのふるまいを予想するためには、複数の例を実行するか、コードの中身をすべて理解するしかない。
- インタフェースはより柔軟になるか、さらなる柔軟性によって機能が強化される。要求を提示するのはクライアントの役目であり、ジェネレータは仕様の文面を満たしさえすればよいからである。
- 最後になってしまったが、この種のインタフェースの方がテストしやすい。モデルには、ジェネレータへの入力を明示的に行う方法が含まれており、**これが同時に出力に対する検証にもなるからだ。**つまり、生成プロセスを制約するためにジェネレータのインタフェースに渡されるのと同じ仕様が、(実装がサポートしていれば) 生成されたオブジェクトが正しいことを確認するという検証の役割にも使用できるのだ (これは第10章で取り上げる表明の一例である)。

要求に応じた構築は、一からオブジェクトを生成するということも意味しているが、同じように仕様を満たすよう既存のオブジェクトを構成することでもある。

例 9.5 ——化学製品倉庫での格納

ある倉庫には、さまざまな化学製品が大型コンテナに詰められて保管されている。ここにはワンボックスカーのようなコンテナが積み重ねられている。化学製品の中には、不活性であるため、どこに保管してもかまわないものがある。また、揮発性であるため、特別な通気設備がついたコンテナに保管しなければならないものもある。爆発性であるために、特殊な強化コンテナに保管しなければならないものもある。また、1つのコンテナに入れてよい組み合わせに関するルールもある。

ここでの目的は、化学製品をコンテナへ入れる上で、効率的で安全な方法を見つけるソフトウェアを作成することだ。

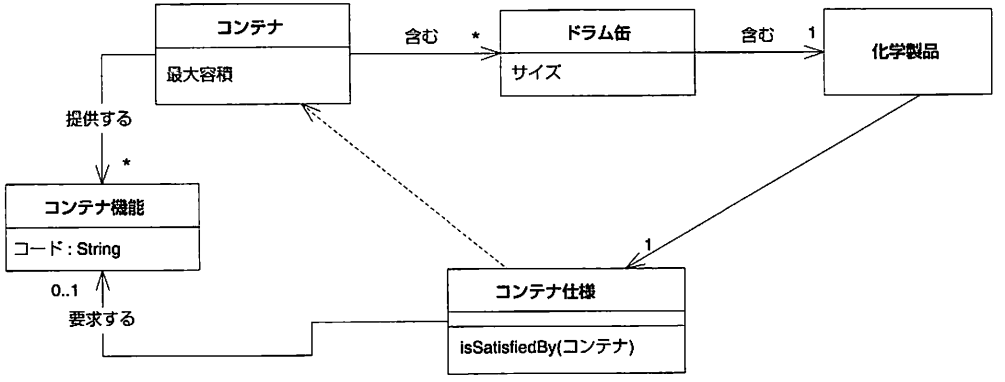


図9.16　倉庫保管のモデル

化学製品を取り出してコンテナに入れる手続きから書き始めることもできるが、まずは、検証の問題から見ていこう。そうすることで、ルールを明示的にせざるを得なくなり、最終的な実装をテストするための道も開ける。

化学製品にはそれぞれコンテナ仕様がある。

化学製品	コンテナ仕様
TNT	強化コンテナ
砂	
生物標本	爆発物と同じコンテナに入れてはならない
アンモニア	通気設備つきコンテナ

さて、これらを**コンテナ仕様**として記述すれば、化学製品の詰まったコンテナの中身を取得して、これらの制約を満たしているかどうかを判定できるはずである。

コンテナ機能	内容		仕様を満たしているか？
強化コンテナ	TNT	9キロ	○
	砂	230キロ	
	生物標本	23キロ	○
	アンモニア		×

コンテナ仕様にあるメソッド`isSatisfiedBy()`は、必要な**コンテナ機能**をチェックするように実装しなければならない。例えば、爆発性化学製品に添えられる仕様であれば、「強化」機能を求めるだろう。

```
/**
 * コンテナ仕様.
 */
public class ContainerSpecification {
   // コンテナ機能
   private ContainerFeature requiredFeature;

   public ContainerSpecification(ContainerFeature required) {
      requiredFeature = required;
   }

   /**
    * 受け取ったコンテナが仕様を満たすかどうかを判定する.
    * @param 判定対象となるコンテナ
    * @return 仕様を満たせば true
    */
   boolean isSatisfiedBy(Container aContainer){
      // 受け取ったコンテナの機能には必要な機能が含まれているか
      return aContainer.getFeatures().contains(requiredFeature);
   }
}
```

爆発性化学製品に仕様を設定するクライアントコードのサンプルを次に示す。

```
// TNTに強化 (ARMORED) コンテナ仕様を設定する
tnt.setContainerSpecification( new ContainerSpecification(ARMORED));
```

コンテナオブジェクトのメソッドisSafelyPacked()は、コンテナが、中にある化学製品の指定する機能を、すべて備えていることを確かめる。

```
/**
 * 安全に格納されているか.
 * @return 安全に格納されていれば true
 */
boolean isSafelyPacked() {
   Iterator it = contents.iterator();
   while (it.hasNext()){
      Drum drum = (Drum) it.next();
      // ドラムのコンテナ仕様はこのコンテナによって満たされていないか
      if (!drum.containerSpecification().isSatisfiedBy(this))
         return false;
   }
   return true;
}
```

この時点で、在庫データベースを調べて、危険な状況があれば報告する監視アプリケーションを作成することもできた。

```
Iterator it = containers.iterator();
while (it.hasNext()) {
   Container container = (Container) it.next();
```

```
        // コンテナは安全に格納されているか
        if (!container.isSafelyPacked())
            unsafeContainers.add(container);
    }
```

ただし、これは我々が作成するように依頼されたソフトウェアではない。ビジネス部門の人々にこういう機会を知らせるのはよいことだが、我々がやるべきなのは、格納サービスの設計である。我々が書いたのは、格納サービス用の判定処理である。このドメインに対する理解と仕様に基づいたモデルにより、**ドラム缶**と**コンテナ**のコレクションを取り、それをルールに従って格納するサービスのために、明確でシンプルなインタフェースを定義できるようになった。

```
/**
 * 倉庫内格納サービス.
 */
public interface WarehousePacker {

    /**
     * ドラム缶をコンテナに格納する.
     * @param containersToFill 格納先コンテナのコレクション
     * @param drumsToPack 格納されるドラム缶のコレクション
     */
    public void pack(Collection containersToFill,
        Collection drumsToPack) throws NoAnswerFoundException;

        /* 表明：pack()の処理が終わると、各ドラム缶のコンテナ仕様は、
        それを格納するコンテナによって満たされるものとする.
        完全な解決策が見つからない場合は例外がスローされる. */

}
```

これで、**格納サービス**の責務を果たすための最適化された制約解決サービスを設計する作業は、アプリケーションの他の部分から分離された。したがって、そのメカニズムにより、設計でモデルを表現している部分が混乱させられることはないだろう（第10章の節「設計の宣言的スタイル」（278ページ）と第15章の凝集されたメカニズム（429ページ）を参照）。この場合でも、格納を規定するルールはドメインオブジェクトから引き離されてはいない。

●例9.5——化学製品倉庫での格納

例9.6——倉庫内格納サービスの、実際に動作するプロトタイプ

倉庫内格納ソフトウェアを機能させる最適化ロジックを作成するのは大仕事である。開発者とビジネスエキスパートで構成される少人数のチームが、プロジェクトチームから分かれてこれに取りかかったが、まだコードを書き始めてさえいない。その間に、別の少人数チームもアプリケーションを開発している。このアプリケーションを使うことで、ユーザはデータベースから在庫を取り出して**格納サービス**に入力し、その結果を解釈できるようになる。アプリケーション開発チームは**格納サービス**がどんなものかを想定して設計しようとしている。しかし、できることといえば、外枠だけのユーザインタフェースを作って、データベースと統合するコードに取り組むことくらいである。意味のあるふるまいをするインタフェースをユーザに提示して、適切なフィードバックを得ることができないのである。同じ理由で、**格納サービス**チームも孤立して仕事をしている。

ドメインオブジェクトとサービスインタフェースが倉庫内格納サービスの例に作りこまれたことにより、アプリケーションチームは、自分たちが**格納サービス**の非常に単純な実装を作成できることがわかった。それによって開発プロセスを前に進めて、作業を並行できるようにしながら、フィードバックループを完結させることができたのだ。ただし、これによる最大の効果を得るためには、完全に動くシステムが必要になる。

```
/**
 * コンテナ.
 */
public class Container {
   // 最大容量
   private double capacity;
   // 内容物
   private Set contents; //ドラム缶

   /**
    * ドラム缶を入れる空間があるかどうかを判定する.
    * @param aDrum ドラム缶
    * @return 空間があれば true
    */
   public boolean hasSpaceFor(Drum aDrum) {
      // 残りの空間がドラム缶のサイズよりも大きいか
      return remainingSpace() >= aDrum.getSize();
   }

   /**
    * 残りの空間を取得する.
    * return 残りの空間の大きさ
    */
   public double remainingSpace() {
      double totalContentSize = 0.0;
      Iterator it = contents.iterator();
      while (it.hasNext()){
         Drum aDrum = (Drum) it.next();
```

```
            // 現在のサイズの総計にドラム缶のサイズを追加
            totalContentSize =totalContentSize + aDrum.getSize();
        }
        return capacity - totalContentSize;
    }

    /**
     * あるドラム缶が収容可能かどうかを判定する.
     * @param aDrum ドラム缶
     * @return 収容可能であれば true
     */
    public boolean canAccommodate(Drum aDrum) {
        // 空間があり、かつ、ドラム缶のコンテナ仕様がこのコンテナによって満たされるか
        return hasSpaceFor(aDrum) &&
            aDrum.getContainerSpecification().isSatisfiedBy(this);
    }

}

/**
 * プロトタイプ格納サービス.
 */
public class PrototypePacker implements WarehousePacker {

    /**
     * 格納する.
     * @param containers コンテナのコレクション
     * @param drums ドラム缶のコレクション
     */
    public void pack(Collection containers, Collection drums)
                throws NoAnswerFoundException {

        /* このメソッドは書かれた通りに表明を満たす.しかし、
        例外が投げられた場合は、コンテナの内容が変更されている可能性がある.
        ロールバックは、より高次のレベルで処理されなければならない. */

        Iterator it = drums.iterator();
        while (it.hasNext()) {
            Drum drum = (Drum) it.next();
            // ドラム缶に合うコンテナを探す
            Container container = findContainerFor(containers, drum);
            container.add(drum);
        }
    }

    /**
     * ドラム缶に合うコンテナを見つける.
     * @param containers コンテナのコレクション
     * @param drum ドラム缶
     * @return ドラム缶に合うコンテナ
     */
    public Container findContainerFor(
              Collection containers, Drum drum)
          throws NoAnswerFoundException {
```

●例 9.6——倉庫内格納サービスの、実際に動作するプロトタイプ

```
        Iterator it = containers.iterator();
        while (it.hasNext()) {
            Container container = (Container) it.next();
            // コンテナがドラム缶を収容できるか
            if (container.canAccommodate(drum))
                return container;
        }
        throw new NoAnswerFoundException();
    }

}
```

確かに、このコードには不十分な点が数多く残されている。砂を特殊なコンテナに入れてしまったために、危険な化学製品を詰めないうちに空きがなくなってしまうかもしれない。収入を最適化するものでないことは間違いない。しかし、最適化問題の多くは、いずれにしても完全には解決されないものである。この実装でも、前述してきたルールには確実に従っている。

このプロトタイプがあることによって、アプリケーション開発者は、外部システムとのあらゆる統合を含め、全速力で進んで行けるようになった。**格納サービス**開発チームも、ドメインエキスパートがプロトタイプを使って考えをまとめることで、フィードバックを得ることができる。このことにより、要求と優先順序が明確になる。**格納サービス**開発チームはプロトタイプを引き継いで、ドメインエキスパートの考えをテストするために、微調整を行うことにした。

さらに**格納サービス**開発チームは、インタフェースを最新の設計に合わせた状態に保ち、アプリケーションと一部のドメインオブジェクトのリファクタリングを強制した。そうすることで、統合の問題にも早期に着手したのである。

洗練された**格納サービス**が使えるようになると、統合は容易に進むようになった。これは、**格納サービス**が、わかりやすく特徴づけられたインタフェースになるように書かれているからである。インタフェースと表明は、アプリケーションがプロトタイプと相互作用して作られていた時に使用されていたものと同じである。

最適化アルゴリズムを正しく動かすために、専門家が数か月を費やした。彼らにとって有益だったのは、プロトタイプを使ってみたユーザからのフィードバックである。そうしている間も、システムの他のどの部分にも、開発中に相互作用できる対象がある。

ここにあるのは、「機能し得る最もシンプルなもの」の一例だが、モデルがより洗練されたことによって、実際に機能するようになった。非常に複雑なコンポーネントの機能を示すプロトタイプも、容易に理解できる20～30行のコードで作れる。モデル駆動的でないアプローチは理解しにくく、アップグレードが困難なので（**格納サービス**が設計の他の部分とより強く結合してしまうだろうから）、今回の場合では、プロトタイプの作成にもっと長い時間がかかっただろう。

コラム——動くプロトタイプを使って開発の停滞を解消する

あるチームは先へ進むために、他チームの実際に動くコードを待っていなければならない。どちらのチームも、コンポーネントを動かしたり、ユーザからのフィードバックを得たりするためには全体の統合を待たなければならない。

こういう停滞は、しばしば、主要なコンポーネントの**モデル駆動**プロトタイプによって解消される。こうしたプロトタイプは、要求をすべて満たしていなくてもよい。実装をインタフェースから分離すると、動く実装さえあれば、それがどんなものでも、プロジェクトの作業を並行して進められる柔軟性が生まれる。時機がきたら、プロトタイプをより効果的な実装で置き換えればよい。そうしている間も、システムの他のどの部分にも、開発中に相互作用できる対象がある。

第10章

しなやかな設計



ソフトウェアの最終的な目的は、ユーザの役に立つことである。しかし、それ以前に、そのソフトウェアは開発者の役に立たなければならない。このことは、リファクタリングを重視するプロセスに特に当てはまる。プログラムが進化するにつれ、開発者はあらゆる部分を再配置し、書き直す。ドメインオブジェクトをアプリケーションに組み込むと同時に、新しいドメインオブジェクトと統合する。作られてから何年経過しようが、保守プログラマはコードの変更と拡張を行う。開発者はこうしたアプリケーションと仕事をしなければならない。だが、そうしたいと思うだろうか？

複雑なふるまいをするソフトウェアの設計が優れたものでないと、リファクタリングしたり、複数の要素を結びつけたりするのは困難になる。開発者が、処理の中に暗黙的に含

まれているものを完全に予測する自信がなくなると、すぐに重複が起き始める。設計要素が一枚岩のようになっていると、重複が避けられない。各部分を再結合させることができないからだ。クラスとメソッドはより優れた再利用のために分解できるが、これらの細かい要素のすべてについて、何をしているのかを把握し続けるのは難しい。ソフトウェアの設計が明確でないと、開発者は現在の混乱した状態に目を向けることさえも恐れ、ましてや変更を加えるなどもってのほかだと考えるようになる。予期せぬ依存関係のせいで、もつれをさらに悪化させたり、何かを壊したりするかもしれないからだ。最小のシステムでない限り、どんなシステムであっても、このように脆弱なせいで、構築できるふるまいの豊かさに限界ができてしまう。リファクタリングとイテレーションによる改良が止まってしまうのである。

過去に作成したレガシーによって押しつぶされないようにし、開発の進行に合わせてプロジェクトを加速させるためには、楽しく仕事ができて変更を呼び込むような設計が欠かせない。それがしなやかな設計である。

しなやかな設計は、深いモデリングを補完するものだ。いったん暗黙的な概念を掘り出して明示的にした際、そこに現れるのはまだ加工されていない原料だ。イテレーションのサイクルを通じて、その原料を叩いて役に立つかたちに変え、主要な概念を簡潔かつ明確にとらえたモデルを育てる。こうして形作られた設計によって、クライアント開発者は、そのモデルを実際に使えるようになるのだ。さらに、設計とコードの開発によって、モデルの概念を改良する洞察が生み出される。こうしたことが繰り返され、イテレーションのサイクルと、より深い洞察へ向かうリファクタリングに戻ってくる。しかし、こうすることで、どんな設計に到達しようとしているのだろうか？ その過程でどういう実験を行うべきだろうか？ それが本章の内容である。

オーバーエンジニアリングの多くが、柔軟性の名の下に正当化されてきた。しかし、抽象化や間接化のためのレイヤを作りすぎてしまうと、邪魔になることの方が多い。扱う人に対して本当に力を与えるソフトウェアがどのような設計になっているか、調べてみるといい。通常はシンプルなものであることがわかるだろう。だが、シンプルに作るのは容易でない。手の込んだシステムへと組み立てることができて、それでも理解しやすい要素を作成するためには、モデル駆動設計に専念する際に、適度に厳密な設計スタイルが求められる。もちろん、こうした設計スタイルは、作成するにも**使用するにも**、比較的洗練された設計スキルが必要となるだろう。

開発者には2つの役割があり、設計はそのどちらに対しても寄与しなければならない。同じ人物が両方の役割を果たすことも十分あるだろうし、数分で役割が切り替わるかもしれない。それでも、役割が異なればコードとの関係は変わる。この役割の1つがクライアント開発者である。クライアント開発者は設計の持つ能力を活用しながら、ドメインオブ

ジェクトをアプリケーションコードや他のドメイン層のコードに織り込む。しなやかな設計によって明らかにされるのは、根底にある深いモデルであり、このモデルによって設計の持つ可能性が明確になる。クライアント開発者は、疎結合した概念を最低限集めたものを柔軟に使用して、ドメインにおける幅広いシナリオを表現できる。設計の各要素は自然に組み合わさり、組み合わせによってもたらされる結果は、予測可能で明確に特徴づけられた強固なものとなる。

同じく重要なこととして、設計は、それを変更しようとする開発者にとっても役に立たなければならない。変更を受け入れやすくするには、設計は容易に理解できなければならず、根底にある、クライアント開発者が利用しているのと**同じ**モデルを明らかにしなければならない。ドメインについての深いモデルの持つ輪郭に従わなければならないので、ほとんどの変更は柔軟な箇所で設計を曲げることになる。コードがもたらす効果は透けて見えるくらい明白でなければならないから、変更がもたらす結果は容易に予想できる。

設計の初期バージョンはたいてい固いものだ。納期や予算のせいで、プロジェクトの多くはしなやかになることが決してない。巨大なプログラムが徹底してしなやかに作られているのを、私は見たことがない。しかし、複雑さによって進捗を阻まれているなら、最も重大で難解な部分を磨いてしなやかな設計にすれば、レガシーの保守に飲み込まれることなく、複雑さの限界を打ち破れるようになる。

ソフトウェアをこのように設計するための公式はないが、私はここにパターンを一式集めてみた。経験上、これはうまく適合すれば、設計にしなやかさを与える傾向があるものだ。これらのパターンと例によって、しなやかな設計がどのようなもので、どういう考え方でそこに至るのかということについての感覚が得られるはずだ。

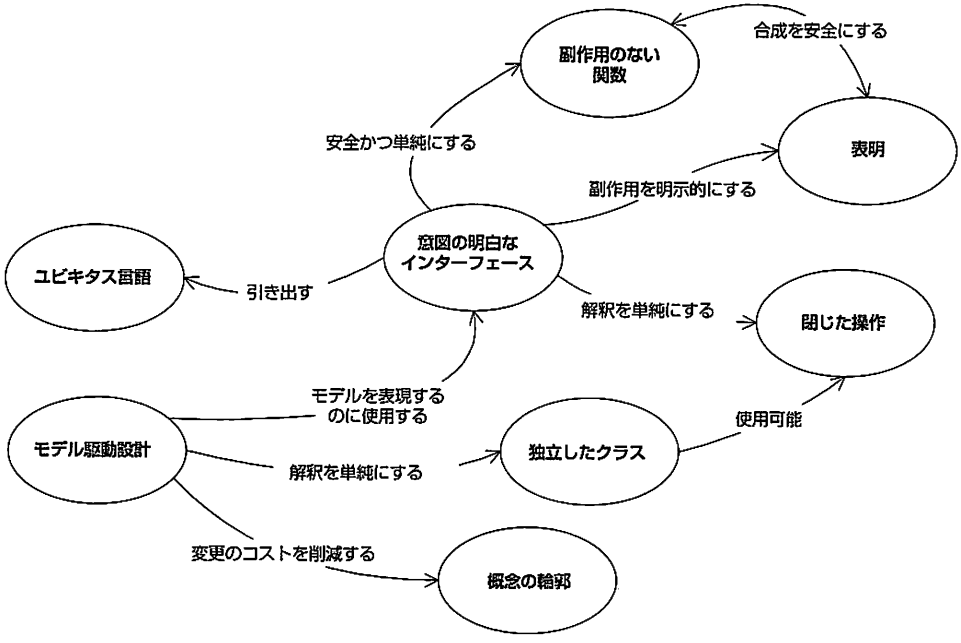


図10.1 しなやかな設計に貢献するパターン

意図の明白なインタフェース（INTENTION-REVEALING INTERFACES）

ドメイン駆動設計を行う際には、意味のあるドメインロジックについて考えたい。あるルールによる効果を生み出しているのに、そのルールを明白に宣言していないコードは、1ステップずつ進むソフトウェアの手続きを思い起こさせる。同じことは、ある演算が、何らかのコードを実行したことから生じる結果でありながら、明示的ではない場合にも当てはまる。モデルと明確に結びついていないと、コードを実行することによる効果を理解したり、変更による影響を予想したりすることは難しい。前章では、ルールや演算を明示的にモデル化することについて深く考察した。そういうオブジェクトを実装するためには、その演算の生々しい詳細やルールの細々とした部分について、多くの理解が必要である。オブジェクトのよいところは、そういうものすべてをカプセル化できる点にある。それによって、クライアントコードはシンプルになり、より高いレベルの概念から解釈できるようになるのだ。

しかし、オブジェクトのインタフェースを見ても、クライアント開発者がオブジェクトを有効に使用する上で何を知っているべきかがわからなければ、詳細をなんとか理解するためにコードの内部を掘り下げなければならなくなる。クライアントコードを読む人も同じことをしなければならないだろう。それでは、カプセル化の価値のほとんどが失われてしまう。我々は常に、認識の過負荷と戦っているのだ。もしクライアント開発者の頭が、コンポーネントがどう動いているのかという詳細でいっぱいになってしまったら、考えが明確にまとまらず、クライアント設計の複雑さを解決できなくなる。このことは、同一人物がコードの開発とその利用という両方の役割を果たしている場合にも当てはまる。詳細について学ぶ必要がないとはいえ、一度に考えることのできる要素の数には限りがあるからだ。

ある開発者があるコンポーネントを使用するために、その実装についてじっくり考えなければならないのであれば、カプセル化の価値は失われる。もともとそのコンポーネントを開発した人とは別の人がオブジェクトや操作の目的を推測する上で、実装を確認しなければならないとしたら、その新しい開発者は、操作やクラスが偶然満たしているだけのものを目的だと思ってしまうかもしれない。そうして推測された目的が本来の意図と異なっていたら、コードは、さしあたりは動くかもしれないが、設計の概念的な基盤が崩壊し、2人の開発者は互いに矛盾した目的に向けて仕事をすることになるだろう。

クラスやメソッドという形式で概念を明示的にモデル化するのは価値のあることだ。だが、その価値を獲得するためには、それらの概念を反映した名前をプログラム要素に与えなければならない。クラスやメソッドの名前は、開発者間のコミュニケーションを高め、

システムの抽象化のされ方を改善させる絶好の機会なのである。

Kent Beckは、メソッド名によって目的を伝えることについて、意図の明白なセレクタ（INTENTION-REVEALING SELECTOR）を用いて説明している（Beck 1997）。設計におけるパブリックな要素がすべて集まってインタフェースを構成しており、こうした要素の名前によって、設計の意図を明らかにする機会が与えられる。型名、メソッド名、引数名が、すべて組み合わされて、意図の明白なインタフェースを形成する。

それゆえ：

クラスと操作には、その効果と目的を記述する名前をつけ、約束したことを実行する手段には言及しないこと。そうすることで、クライアント開発者はインタフェースの内部を理解しなくてよくなる。こうした名前はユビキタス言語に従っていなければならない。そうすることで、チームメンバがすぐ意味を推測できるようになるからだ。ふるまいを作成する前にそのテストを書いて、自分がクライアント開発者の視点で考えられるようにすること。

手の込んだ仕組みはすべて抽象的なインタフェースの背後にカプセル化し、またそうしたインタフェースには、手段ではなく目的の観点から語らせなければならない。

ドメインの持つ公開インタフェースでは、関係性やルールは宣言されるが、それをどのように強制するかは語られない。またイベントやアクションは記述されるが、それをどのように実行するかについては触れられない。方程式は立てられても、それを解く数値メソッドは示されない。質問を投げかけても、答えを見つける手段を示してはならないのだ。

例10.1 ——リファクタリング：塗料混合アプリケーション

塗料店向けに作られたあるプログラムは、標準的な塗料を混ぜ合わせた結果を顧客に見せることができる。次に示すのはその初期の設計で、ドメインクラスが1つある。

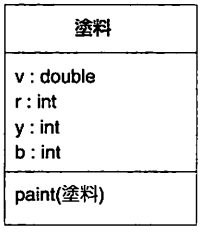


図10.2

`paint(塗料)`メソッドが何をするかについては、見当をつけるだけでも、コードを読むしかない。

```
public void paint(Paint paint) {
    v = v + paint.getV(); // 混ぜ合わせた後に容量を合計する
    // 色を混ぜ合わせるための複雑なロジックは多くの行にわたっているが、省略する.
    // 最後にインスタンス変数r（赤）、b（青）、y（黄色）に新しい値を代入する.
}
```

わかった。このメソッドは2つの塗料（Paint）を混ぜ合わせるようだから、結果としては容量が増えて色も混ざる。

視点を変えて、このメソッドのためにテストを書こう（このコードはJUnitテストフレームワークに基づいている）。

```
public void testPaint() {
    // 単色で量が100の黄色い塗料を作成する
    Paint yellow = new Paint(100.0, 0, 50, 0);
    // 単色で量が100の青い塗料を作成する
    Paint blue =new Paint(100.0, 0, 0, 50);

    // 青を黄色に混ぜ合わせる
    yellow.paint(blue);

    // 結果として量が200.0の緑色の塗料ができるはず
    assertEquals(200.0, yellow.getV(), 0.01);
    assertEquals(25, yellow.getB());
    assertEquals(25, yellow.getY());
    assertEquals(0, yellow.getR());
}
```

テストが通ることは出発点にすぎない。この時点で満足することはできないのは、テストのコードが何をしているかを我々に伝えていないからである。テストを書き直して、我々がクライアントアプリケーションを作成しているとしたら、塗料オブジェクトをどう使いたいかを反映させることにしよう。最初、このテストは失敗する。実際には、コンパイルすらできない。我々がこのテストコードを書いているのは、塗料オブジェクトのインタフェース設計をクライアント開発者の視点から検討するためだ。

```
public void testPaint() {
    // 単色で量が100の黄色い塗料から始める
    Paint ourPaint = new Paint(100.0, 0, 50, 0);
    // 単色で量が100の青い塗料を取得する
    Paint blue =new Paint(100.0, 0, 0, 50);

    // 青を黄色に混ぜる
    ourPaint.mixIn(blue);

    // 量200.0の緑色の塗料ができる
    assertEquals(200.0, ourPaint.getVolume(), 0.01);
    assertEquals(25, ourPaint.getBlue());
    assertEquals(25, ourPaint.getYellow());
    assertEquals(0, ourPaint.getRed());
}
```

●例10.1——リファクタリング：塗料混合アプリケーション

これらのオブジェクトをどう扱いたいかをテストに反映させるためには、時間をかける必要がある。その後で、塗料クラスをリファクタリングして、テストが通るようにする。

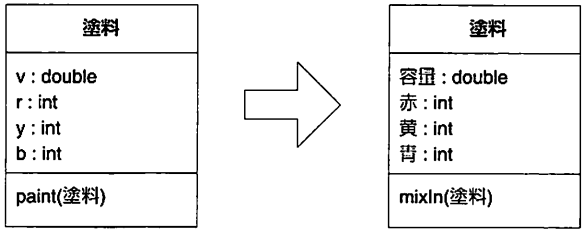


図10.3

新しいメソッド名も、別の塗料を「混ぜ合わせる」効果を読む人にすべて伝えきれないかもしれない（だから、数ページ後に出てくる表明が必要になる）。それでも、クラスを使用し始めるには十分な手がかりを、しかもテストによって示される例と合わせて、読む人に与えている。さらに、クライアントコードを読む人は、クライアントの意図を解釈できるようになる。本章でこれから取り上げるいくつかの例では、このクラスをさらにリファクタリングして、より明確にしていく。

* * *

サブドメインはすべて別々のモジュールへと切り分けられ、意図の明白なインタフェースの背後にカプセル化される。そのようにかたちを整えることによって、プロジェクトに集中し巨大なシステムの複雑さを管理できるようになる。これについては、第15章「蒸留」の中の凝集されたメカニズム（429ページ）や汎用サブドメイン（414ページ）を使って説明する。

だが、次の2つのパターンでは、メソッドを使用した結果を、非常に予測しやすいものにすることを目指そう。複雑なロジックは、副作用のない関数によって安全に実行できる。システムの状態を変更するメソッドは、表明を使って特徴づけることができる。

副作用のない関数（SIDE-EFFECT-FREE-FUNCTIONS）

操作は大きく2つのカテゴリに分けられる。それが、コマンド（command）とクエリ（query）だ。クエリとはシステムから情報を取得するもので、単に変数の持つデータにアクセスするだけのこともあれば、そのデータに基づいて演算を行うこともある。コマンド（モディファイア（modifier）ともいう）とは、システムに何らかの変更を与える操作である（簡単な例としては、変数に対する値の設定がある）。普通の言葉で**副作用**（side effect）と言えば、意図しない結果を意味するが、コンピュータ科学では、システムの状態に対して与えられる、あらゆる影響を意味する。ここでは説明をわかりやすくするために、その意味を狭めて、システムの状態に対するあらゆる変化のうちで将来の操作に影響するものだとしよう。

操作によって影響を受ける、きわめて意図的な変更を表すのに、なぜ**副作用**という言葉が選ばれ、使われたのだろうか？ それは複雑なシステムでの経験に基づいてのことではないかと私は思う。ほとんどの操作は他の操作を要求し、呼び出された操作がさらに別の操作を呼び出す。この恣意的な深いネストがあると、ある操作を呼び出したことの結果をすべて予想することが非常に難しくなる。クライアント開発者は、ネストの2層目、3層目の操作による影響を意図していなかったかもしれない。そうした影響が、語のあらゆる意味において副作用になったのである。複雑な設計にある複数の要素は、これとは別のかたちで相互作用するが、同じように予測不可能になりそうだ。**副作用**という言葉を使用することで、その相互作用を避けられないことが強調される。

複数のルール間に存在する相互作用や、演算の合成は、予測するのが極端に困難になる。操作を呼び出す開発者は、結果を予想するために、呼び出し先の実装とそれが委譲する実装をすべて理解しなければならない。開発者がインタフェースによって隠されたものを見る必要があるのだとしたら、そのインタフェースによる抽象化の効用は限定されてしまう。安全に予測できるように抽象化されていないと、開発者は組み合わせの爆発を制限しなければならなくなり、現実に構築できるふるまいの豊かさを低く抑えることになる。

副作用を起こさずに結果を戻す操作は、**関数**（function）と呼ばれる。関数は何度も呼び出すことができ、毎回同じ値を戻すことができる。ネストの深さを気にすることなく、ある関数から他の関数を呼び出すこともできる。副作用のある操作よりも、関数をテストする方がはるかに容易である。これらの理由から、関数によってリスクを減らせる。

ほとんどのソフトウェアシステムがコマンドを使わずにいられないのは明らかだが、この問題を抑える方法は2つある。第一に、コマンドとクエリを別々の操作に厳密に分離しておくことができる。変更を引き起こすメソッドに関して、ドメインデータを戻さないようにすることと、可能な限りシンプルに保つことを保証すること。クエリと演算はすべて、目に見える副作用を起こさないメソッドで実行すること（Meyer 1988）。

第二に、今ある設計に代わる、既存のオブジェクトを一切変更しないようなモデルや設計が、たいていの場合あるはずだ。オブジェクトを変更する代わりに、処理の結果を表す新しい値オブジェクトが生成されて戻される。これは一般的なテクニックで、この後に続く例でも説明する。値オブジェクトはクエリに対する回答として生成され、受け渡されて、忘れられる。値オブジェクトは、ライフサイクルが慎重に規定されるエンティティとは異なるのだ。

値オブジェクトは不変である。このことは、生成時にのみ呼び出されるイニシャライザを別にすると、**すべての**操作が関数であることを示唆している。関数と同じように、値オブジェクトも安全に使用することができ、テストも容易である。ロジックや演算が状態の変更と混在している操作は、リファクタリングして2つの別々の操作にすべきである(Fowler 1999, p.279)。しかし、当然のことだが、このように副作用を分離して単純なコマンドメソッドにする方法は、エンティティにしか適用されない。クエリから変更を分離するリファクタリングを完了した**後で**、複雑な演算を行う責務を値オブジェクトへと移す、2回目のリファクタリングを検討すること。既存の状態を変更する代わりに値オブジェクトを取得するか、すべての責務を値オブジェクトに移すことによって、副作用は完全に取り除ける。

それゆえ：

プログラムロジックのうち、できる限り多くの部分を関数に置くこと。関数とは、目に見える副作用なしに結果を戻す操作である。コマンド（目に見える状態を変更することになるメソッド）は厳密に分離して、ドメインについての情報を戻さない、非常に単純な操作にすること。ある概念が、値オブジェクトの担う責務に合致する場合には、複雑なロジックをその値オブジェクトに移すことによって、副作用をさらに制御すること。

副作用のない関数、中でも不変の値オブジェクトに置かれたものは、操作を安全に組み合わせることができる。関数が意図の明白なインタフェースによって示されていれば、開発者は実装の詳細を理解していなくても、その関数を使うことができる。

例 10.2——リファクタリング：塗料混合アプリケーション再考

塗料店向けに作られたあるプログラムは、標準的な塗料を混ぜ合わせた結果を顧客に見せることができる。先ほどの例が終わったところから説明を再開すると、1つあったドメインクラスは次のようなものだった。

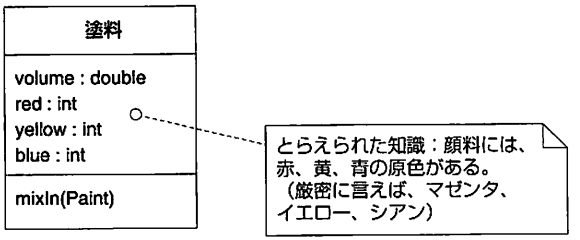


図 10.4

```
public void mixIn(Paint other) {
    volume = volume.plus(other.getVolume());
    // 色を混合させる複雑なロジックが多くの行にわたる.
    // 最後に red、blue、yellow に新しい値を代入する
}
```

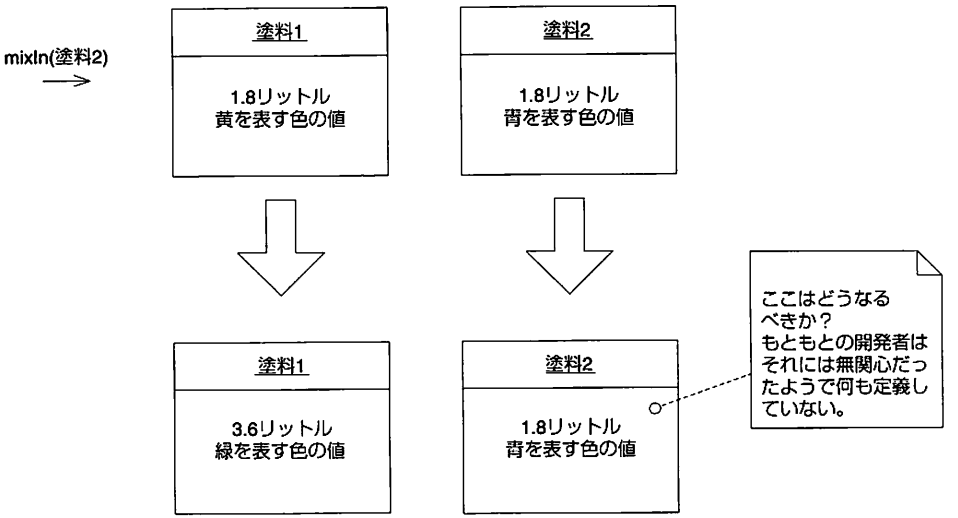


図 10.5　mixIn()メソッドの副作用

mixIn()メソッドの中では多くの処理が行われるが、この設計は確かに、問い合わせと変更を分離するルールを守っている。ただ1つ問題があって、これは後で取り上げるが、mixIn()メソッドの引数である塗料2オブジェクトの容量がどうなるかが宙に浮いているのだ。塗料2の容量は操作によって変更されないが、このことは、この概念モデルのコンテキストに置くとあまり論理的でないように見える。このことがもともとの開発者にとって問題にならなかったのは、想像する限り、操作後の塗料2に全く興味がなかったからだろう。しかし、副作用がもたらす結果を予想したり、あるいは副作用がないことを予想したりするのは困難である。この問題については表明を議論する際に再度取り上げる。今のところは、色について見ていこう。

色は、このドメインにおける重要な概念である。これを明示的にオブジェクトにしてみよう。どう呼ぶべきだろうか？ まず思い浮かぶのは「色（color）」だが、知識をかみ砕いたことで、すでに重要な洞察が得られている。すなわち、色の混ぜ合わせ方は、塗料の場合、よく知られているRGB表示とは異なるのだ。名前にはこれを反映させる必要がある。

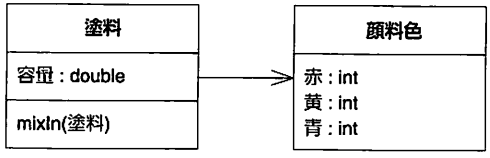


図10.6

顔料色（Pigment Color）を分解することで、初期バージョンよりも確実に多くの内容が伝わるようになるが、処理は同じで、まだmixIn()メソッドの中にある。色のデータを移動した際に、それと一緒に関連するふるまいを取り出すべきだった。その前に注意しておくべきなのは、**顔料色**が値オブジェクトだということである。したがって、不変なものとして扱わなければならない。塗料を混ぜ合わせた際には、**塗料**オブジェクト自体が変化した。これはライフサイクルの途中にあるエンティティだったのだ。これとは対照的に、特定の色合いの黄色を表す**顔料色**は、常に変わらない。その代わり、混ぜ合わせることで、結果として新しい色を表す新しい**顔料色**オブジェクトができる。

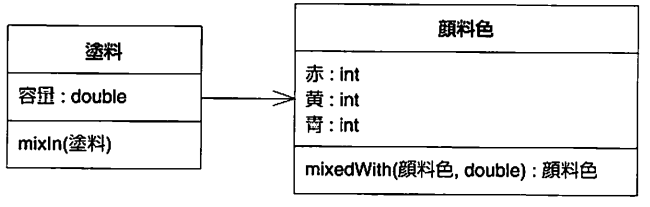


図10.7

```
/**
 * 顔料色.
 */
public class PigmentColor {

   /**
    * 他の顔料色と混ぜ合わせる.
    * @param other もう1つの顔料
    * @param ratio 混ぜ合わせる割合
    * @return 顔料色
    */
   public PigmentColor mixedWith(PigmentColor other, double ratio){
      // 色の混合の複雑なロジックのライン多数
      // 最後に適切な新しいred、blue、yellowの値を持った
      // 新しい顔料色オブジェクトを作成する.
   }
}

/**
 * 塗料.
 */
public class Paint {

   public void mixIn(Paint other){
      volume =volume + other.getVolume();
      // 割合＝他の顔料色の量 / この顔料色の量
      double ratio =other.getVolume() / volume;
      // 今の顔料色を他の顔料色と指定の割合で混ぜる.
      pigmentColor =
         pigmentColor.mixedWith(other.pigmentColor(), ratio);
   }
}
```

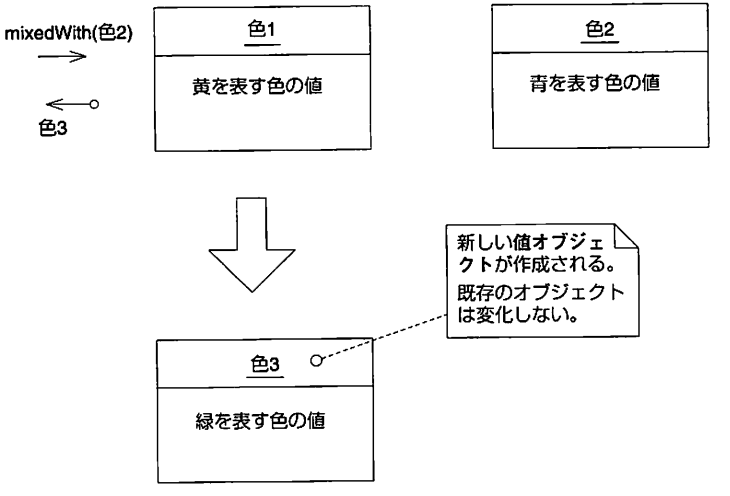


図 10.8

これで、塗料に置かれた変更を行うコードは可能な限り単純になった。新しい顔料色クラスはドメインの知識をとらえてそれを明示的に伝え、副作用のない関数を提供する。その結果は理解しやすく、テストをするのも容易で、使うのも、他の操作と組み合わせるのも、安全に行える。このように安全であることにより、色の混合という複雑なロジックが本当にカプセル化されている。このクラスを使用する開発者は、実装を理解しなくてもよいのだ。

* * *

表明（ASSERTIONS）

複雑な演算を副作用のない関数に分離することによって、問題は適当な大きさに切り詰められる。だが、エンティティ上にはまだ副作用を作り出すコマンドが残っており、そのコマンドを使用する人は誰でも、結果を理解していなければならない。表明があることで、副作用が明示的になり、対処しやすくなる。

＊　＊　＊

確かに、複雑な処理を含んでいないコマンドは、調査によって解釈するのがかなり簡単かもしれない。しかし、大きい部品が小さい部品によって構築される設計では、あるコマンドが別のコマンドを呼び出しているかもしれない。高次のコマンドを使用する開発者は、その奥にあるコマンドそれぞれの結果を理解していなければならない。カプセル化はここまでだ。また、オブジェクトのインタフェースは副作用を禁止しないため、同じインタフェースを実装する2つのサブクラスには、別々の副作用があるかもしれない。このサブクラスを使用する開発者は、結果を予測するために、どちらがどちらなのかを知りたいと思うはずだ。抽象化と多態性はここまでだ。

操作の副作用が実装によって暗黙的にしか定義されていない場合、委譲が多く行われている設計では、原因と結果がもつれてしまう。あるプログラムを理解するには、分岐したパスをたどって実行を追跡するしかない。カプセル化の価値は失われる。具体的な実行を追跡する必要があるせいで、抽象化も打ち負かされる。

我々に必要なのは、内部を徹底的に調べなくても、設計要素の意味と操作の実行結果を理解できるようにする方法である。意図の明白なインタフェースによって一部は達成できるが、意図を非定型的に示唆しても、常にそれで十分とは限らない。「契約による設計」学派は次の段階に進み、クラスとメソッドについて、開発者によって保証される「表明」を作成する。このスタイルは『オブジェクト指向入門』（Meyer 1988）において詳述されている。要約すると、「事後条件」（post-conditions）が操作の副作用、つまり、メソッドを呼び出すことで保証される結果を記述する。「事前条件」（preconditions）とは契約にあるただし書きのようなもので、事後条件が成り立つことを保証するために満たさなければならない条件のことである。クラスの不変条件は、あらゆる操作が終わった時のオブジェクトの状態に関する表明となる。不変条件は、集約全体に対しても宣言することができ、整合性に関するルールを厳密に定義する。

こうした表明が記述するのは、すべて状態であって手続きではないため、容易に分析できる。クラスの不変条件によって、クラスの意味が特徴づけられ、オブジェクトをより予測しやすくすることで、クライアント開発者の仕事が単純化される。事後条件による保証を信用するのであれば、メソッドがどう動くかを気にする必要はない。委譲による影響は、

表明の中に組み込まれているはずである。

それゆえ：

操作の事後条件と、クラスおよび集約の不変条件を宣言すること。プログラミング言語で表明を直接コーディングできない場合は、その代わり、自動化されたユニットテストを書くこと。プロジェクトの開発プロセスのスタイルに合う場合には、表明をドキュメンテーションや図の中に記述すること。

凝集した概念が集まったモデルを探し求めること。それも、意図された表明を開発者に推測させ、学習効率を上げながら、矛盾したコードが作られるリスクを小さくするものでなければならない。

多くのオブジェクト指向言語において今のところは直接サポートされていないとはいえ、やはり、表明は設計について考える上で強力な方法である。自動化されたユニットテストによって、言語でサポートされていないことでもある程度は補える。表明はどれも手続きではなく状態に関するものだから、テストを書くのが容易になるのだ。テストを準備する際に事前条件を整え、実行後に事後条件が成り立っているかどうかを判定する。

不変条件と事前条件、事後条件を明確に述べることによって、開発者は操作やオブジェクトを使用した結果を理解できるようになる。理論上、矛盾のない表明の集合はすべて機能する。しかし、人は、頭の中に述語を溜め込むだけではない。モデルの概念を推測したり、補ったりするので、アプリケーションの要求を満たすだけでなく、人にとって意味のあるモデルを見つけ出すことが重要だ。

例 10.3 ——塗料の混合に戻る

前述の例では、塗料クラスの持つ`mixIn(塗料)`操作の引数がどうなるかという点があいまいであったが、それについて私が懸念していたのを思い出して欲しい。

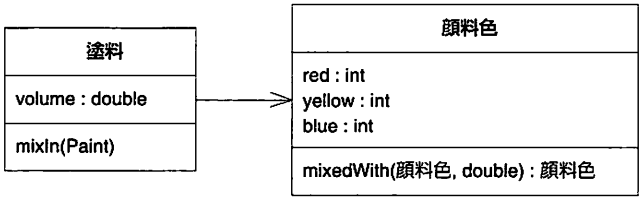


図10.9

受け取る側の容量は、引数の容量分だけ増える。物理的な塗料に関する一般的な理解を活用すると、この混合のプロセスは、もう一方の塗料を同じ量だけ使い果たさなければならない。そのためには、もう一方から抜き取って容量をゼロにするか、完全に削除するかである。現在

の実装では引数として与えられた塗料を変更しないが、いずれにしても、引数を変更することは、副作用の中でもとりわけリスクが高い。

確実なところから始めるため、mixIn()メソッドの事後条件について現状を述べておこう。

p1.mixIn(p2)の後：
  p1.volumeはp2.volume分だけ増加する。
  p2.volumeは変わらない。

問題は、開発者が間違いを犯しそうだということである。こうした性質は、我々が開発者を誘導してきた概念と整合しないからだ。素直な解決策は、片方の塗料の容量をゼロに変更することだろう。引数を変更してしまうのは悪い習慣だが、簡単であると同時に直観的でもあるようだ。不変条件は次のように宣言できるだろう。

塗料の総容量は、混合によって変化しない。

だが、待ってほしい！ 開発者はこの選択肢について深く考えている間に、ある発見をしたのだ。もともとの開発者がこのように作ったのには、説得力のある理由があった。最終的に、**このプログラムは追加された塗料の混ぜ合わせる前の一覧を帳票として出力している。**結局、このアプリケーションの最終的な目的は、どの塗料を混ぜ合わせたらよいかを、ユーザがわかるようにすることなのだ。

したがって、容量モデルを論理的に一貫したものにすると、アプリケーションの要求には適さなくなってしまう。ここにはジレンマがあるようだ。我々は奇妙な事後条件をドキュメント化するのに行き詰まり、優れたコミュニケーションで埋め合わせようとしているのだろうか？この世のすべてが直観的とは限らず、それが最適な答えになることもある。しかし今回は、このようにぎこちないことにより、概念の欠如が示されているように思われる。新しいモデルを探そう。

これで明確にわかるようになった

よりよいモデルを探している時には、我々はもともとの開発者と比べてかなり有利な立場にいるのだが、それは知識のかみ砕きとより深い洞察に向けたリファクタリングがその間に起きるためである。例えば、色を処理するのに、値オブジェクトに副作用のない関数を使用する。これが意味しているのは、必要な時にいつでも計算を繰り返せるということだ。この有利な立場は利用すべきである。

どうも塗料には、2つの異なる基本的な責務が与えられているようだ。これを分けてみよう。

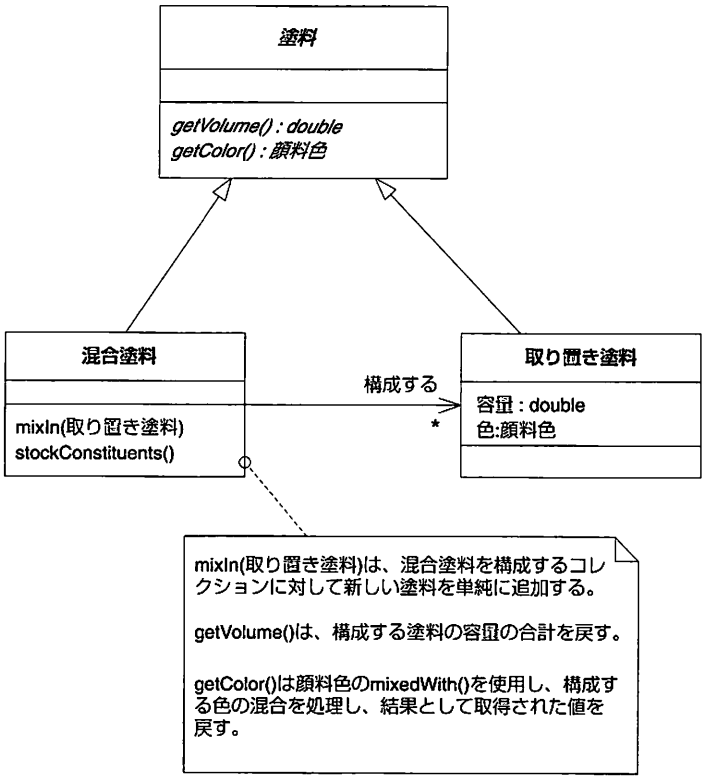


図10.10

ここまで来ると、コマンドは`mixIn()`だけである。このメソッドはオブジェクトをコレクションに加えるものであり、その効果はモデルに対する理解に照らしても直観的である。これ以外の操作はすべて、副作用のない関数だ。

図10.10に挙げた表明の1つを確認するテストメソッドは、次のようになる（JUnitテストフレームワークを使用）。

```
public void testMixingVolume {
    // 顔料色：黄
    PigmentColor yellow = new PigmentColor(0, 50, 0);
    // 顔料色：青
    PigmentColor blue = new PigmentColor(0, 0, 50);

    // 取り置き塗料1：黄
    StockPaint paint1 = new StockPaint(1.0, yellow);
    // 取り置き塗料2：青
    StockPaint paint2 = new StockPaint(1.5, blue);
```

```
        // 混合塗料を生成
        MixedPaint mix = new MixedPaint();
        // 塗料1を混合
        mix.mixIn(paint1);
        // 塗料2を混合
        mix.mixIn(paint2);

        // 混合塗料の容量は2.5
        assertEquals(2.5, mix.getVolume(), 0.01);
    }
```

このモデルは、ドメインについての知識をより多くとらえ、伝えている。不変条件と事後条件は常識的であり、そのため保守するのも使用するのも容易になっている。

＊　＊　＊

意図の明白なインタフェースの持つ豊かな伝達力が、副作用のない関数および表明によってもたらされる予測可能性と結びついて、カプセル化と抽象化が安全になるはずだ。

組み替え可能なさまざまな要素の中で、次に登場するのは、効果的な分解である…。

●例 10.3 ——塗料の混合に戻る

概念の輪郭（CONCEPTUAL CONTOURS）

機能を細かく切り刻んで、柔軟に組み合わせられるようにすることがある。機能を大きくまとめて、複雑さをカプセル化することもある。粒度を一定にすることを追求して、すべてのクラスと操作を似たような規模にすることもある。こうしたことは過度の単純化であり、一般的なルールとしてはうまく機能しない。だが、動機となっているのは基本的な問題群である。

モデルや設計の要素が一枚岩の構造に埋め込まれていると、機能は重複する。外部インタフェースは、クライアントが重要だと思うことをすべて伝えるとは限らない。そのような要素の意味を理解するのは、さまざまな概念が混在しているため、難しい。

その一方で、クラスやメソッドを分割すると、クライアントを無意味に複雑にしてしまうかもしれない。細々とした要素の組み合わせ方を理解することを、クライアントオブジェクトに強制するからだ。さらに悪いことに、概念が完全に失われかねない。ウランの原子を半分に分けたら、それはウランではない。もちろん、重要なのは単なる粒度ではなく、その粒がどこで活動するかである。

クックブックのようなルールは機能しない。しかし、たいていのドメインの深いところには、論理的な一貫性があるもので、そうでなければ、ドメインは自らが対象とする領域の中に存在できない。これは、ドメインに全く矛盾がないということではない。確かに、ドメインについて人々の語る内容は一貫性を欠いているだろう。だが、どこかで語呂や論理がつながっているはずであって、そうでなければ、モデリングは無意味になってしまう。このように、根底には一貫性があるので、ドメインの一部と共鳴するモデルが見つかったら、そのモデルは後から発見される部分と矛盾しない可能性は高い。新しい発見にモデルを適合させるのが難しいこともあるが、その場合にはより深い洞察に向けたリファクタリングを行って、次の発見とは適合することを期待する。

リファクタリングを繰り返すことで、最終的にはしなやかになる理由の1つがこれだ。新しく理解された概念や要求にコードが適合するにつれ、概念の輪郭が姿を現す。

高凝集と低結合という一対の基本は、あらゆる規模の設計でその役割を果たす。これは、個々のメソッドからクラスとモジュールを経て、大規模な構造に至る（第16章参照）。この2つの原理は、コードに適用されるのと同様に、概念にも適用される。これを機械的に見てしまわないよう、ドメインについての自分の直観に頻繁に立ち返ることで、技術的思考を抑えること。意思決定を行う際には、次のように自問しなければならない。「これは、現在のモデルやコードに存在する特定の関係性の集まりに基づいて行われる応急処置にすぎないのか、それとも根底にあるドメインの輪郭となにかしら共鳴しているのか？」

概念的に意味のある機能の単位を見つけ出すこと。そうすれば、その結果できる設計は柔軟で理解しやすくなる。例えば、2つのオブジェクトの「加算」が、そのドメインにおい

て凝集した意味を持つのであれば、そのレベルでメソッドを実装すること。add()メソッドを2つのステップに分けてはならない。同じ操作の中で次のステップに進んでもいけない。もう少し広い見方をすると、各オブジェクトは単一の完結した概念、つまり「完結した値[1]」でなければならない。

同様に、どんなドメインにも、ソフトウェアを利用する側の人々が、詳細に対しては興味を持たない領域がある。この疑似的な塗料混合アプリケーションのユーザが、赤や青の顔料を加えることはない。3つの顔料がすべて入った完成した塗料を混ぜ合わせるのだ。切り離したり再配置したりする必要のないものをまとめれば、雑音を避けられるし、組み合わせることが本当に意図された要素を理解しやすくなる。ユーザが物理的な設備を持っており、そこで個々の顔料を加えることができるのであれば、ドメインは変更され、個々の顔料が扱われることになるかもしれない。塗料を扱う化学者であれば、さらに細かい制御を必要とするだろうし、それには全く異なる分析が行われ、塗料を混ぜ合わせるための抽象化された顔料より、はるかに詳細な塗料を生成するモデルが作り出されるだろう。しかし、この塗料混合アプリケーションプロジェクトでは、そこに携わる誰にとっても、こうしたモデルは単に的外れなだけだ。

それゆえ：

設計要素（操作、インタフェース、クラス、集約）を凝集した単位に分解すること。その際、ドメインにおける重要な分割に対する自分の直観を考慮に入れること。連続的なリファクタリングを通して変化と安定の軸を観察し、設計の根底にある概念の輪郭の中で、こうした分割のパターンを説明しているものを探すこと。ドメインの持つ一貫したいくつかの側面の中には、そもそもモデルを知識の息づく領域にしているものがある。モデルを、そうした側面と揃えること。

目標はシンプルなインタフェースの集まりを作ることだ。こうしたインタフェースは、論理的に結びつけられて、ユビキタス言語において意味のある文を作るのであり、的外れの選択肢によって気が散らされたり、保守の負荷が増えたりすることはない。これは通常、リファクタリングの結果であるから、アップフロントに作り出すのは難しい。しかし、技術指向のリファクタリングからは、現れることがまずないだろう。これが姿を現すのは、より深い洞察に向けたリファクタリングからなのだ。

設計が概念の輪郭に従っている場合でも、修正とリファクタリングは必要である。連続的なリファクタリングが局所的になりがちで、モデルの広い範囲に及んで複数の概念に影響を与えることがないのであれば、それはモデルが適合していることを示している。ある要件に直面したことにより、オブジェクトとメソッドの分割方法を広範囲にわたって変更しなければならないのであれば、それは警告である。すなわち、我々はドメインに対する

1 Ward Cunninghamの完結した値パターン

理解を改良しなければならないのだ。これは、モデルを深めて、設計をよりしなやかにするための好機となる。

例 10.4——発生の輪郭

第9章では、会計の概念に関する深い洞察に基づいてローン追跡システムをリファクタリングした。

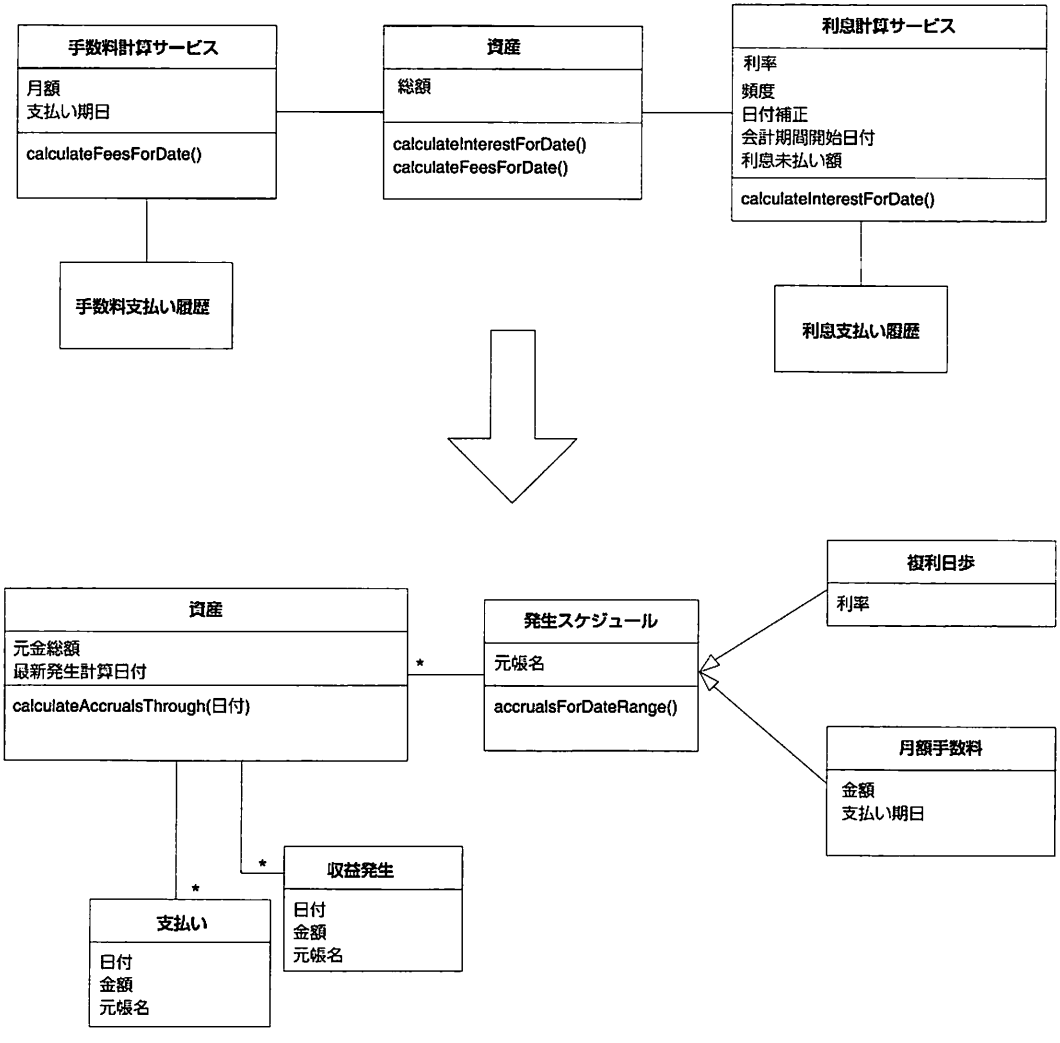


図 10.11

新しいモデルは古いものと比べてオブジェクトが1つ増えただけだが、責務の区分が大きく変わっている。

スケジュールは**計算サービス**クラスにあったcase文のロジックとして機能していたものだが、これが手数料と利息という異なる型に対応して、別々のクラスになった。一方、手数料と利息の支払いは、もともと分けられていたものだが、1つにまとめられた。

新しく明示的になった概念と**発生スケジュール**の階層にある凝集性が相互に共鳴しているので、開発者はこのモデルの方がドメインの概念の輪郭にうまく従っていると信じたのだ。

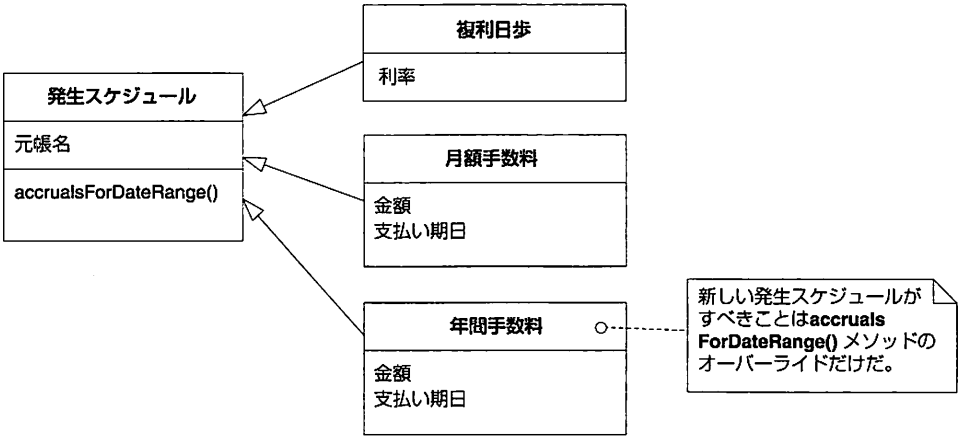


図10.12　このモデルは新しい種類の発生スケジュールの追加にも対応している

開発者が自信を持って予測することのできた変更が1つあった。それは新しい**発生スケジュール**の追加である。この要件があることは、すでにわかっていた。そこで、既存の機能をよりシンプルで明確にするだけでなく、この開発者は新しいスケジュールを簡単に導入できるモデルを選択したのだ。しかし、この開発者が見つけたのは、アプリケーションとビジネスの進化に合わせて、ドメインの設計が変化し成長するのを助けるような概念の輪郭だったのだろうか？ 予期しない変更を設計がどう処理するかについては、何も保証することができないのだが、この開発者は、勝率を上げることはできたと考えた。

予想外の変更

プロジェクトが進むにつれ、早期支払い（early payment）と遅延支払い（late payment）を処理する詳細なルールに関する要件が出てきた。この問題について調べたところ、利息の支払いと手数料の支払いにも事実上同じルールが適用できることがわかり、この開発者は満足した。つまり、新しいモデル要素は、単一の**支払い**クラスと自然につながるのだ。

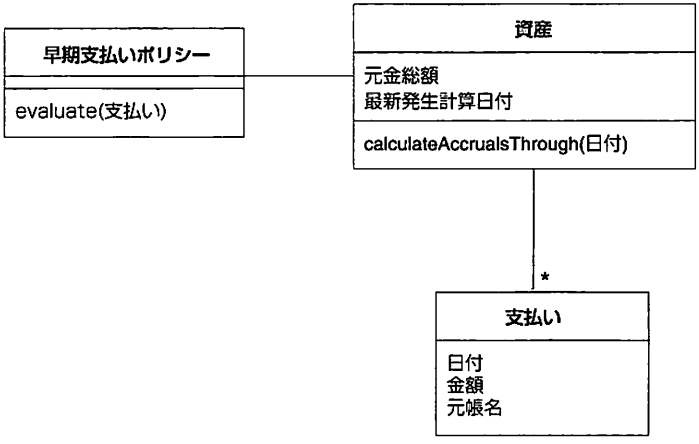


図10.13

古い設計であれば、2つの**支払い履歴**クラスの間で重複が起こってしまっただろう（この問題が契機となって**支払い**クラスを共有するという洞察が生まれ、前述のものとは別の経路によって同じようなモデルができたかもしれない）。このように簡単に拡張できたのは、開発者が変更を予期したからではない。また、設計の用途を非常に広くして、考えられるどんな変更にも対応できるようにしたからでもない。以前のリファクタリングによって、設計がドメインの根底にある概念と揃えられたからである。

＊　＊　＊

意図の明白なインタフェースにより、クライアントはオブジェクトのことをただの仕組みとしてではなく、意味の単位として示せるようになる。副作用のない関数と表明によって、そういう単位を使用し、複雑に組み合わせることが安全にできるようになる。概念の輪郭が出現したことで、モデルを構成する各部分を安定させて、意味の単位をより直観的に使用したり、組み合わせたりできるようになる。

それでも、相互依存関係のせいで一度にあまりに多くのことを考えなければならなくなると、やはり概念的な過負荷に悩まされることになる…。

独立したクラス（STANDALONE CLASSES）

相互依存関係があると、モデルと設計は理解するのが難しくなる。また、テストや保守も困難になる。しかも、相互依存関係は簡単に積み重なってしまうのだ。

もちろん、すべての関連は依存関係であり、あるクラスを理解するためには、それが何に結びつけられているかを理解しなければならない。こうして結びつけられたものはさらに多くのものと結びつけられることになり、それらも理解しなければならない。あらゆるメソッドのあらゆる引数の型も依存関係であり、あらゆる戻り値もそうだ。

依存関係が1つあると、同時に2つのクラスとそれらの関係の性質について考えなければならない。依存関係が2つになれば、考えなければならないことは、3つのクラスそれぞれと、各クラスに対して持つ関係の性質、さらに、クラス間にあるあらゆる関係となる。それぞれのクラスがまた何かに依存していたら、それにも注意しなければならない。さらに、依存関係が3つなら…と、雪だるま式に大きくなるのだ。

モジュールも集約も、相互依存関係の網の目を制限するためのものだ。凝集度の高いサブドメインを切り分けてモジュールにすれば、あるオブジェクトの集合がシステムの残りの部分から分離され、相互に関連する概念の数は限られる。しかし、1つのモジュールでさえ、内部の依存関係を制御することに対して、熱狂的とも言えるくらいに取り組まなければ、考えるべきことが多くなってしまう。

1つのモジュールの中であっても、設計を解釈することは、依存関係が加わるにつれて無秩序に難しくなる。これによって精神的な負荷が大きくなり、開発者が処理できる設計の複雑さが制限される。暗黙的な概念の方が、明示的な参照よりも負荷を大きくする。

精緻なモデルは、そこに含まれる概念間のつながりが、それらの概念の意味にとって根本的なものを表現するもの以外は残らなくなるまで蒸留される。重要なサブセットにおいては、依存関係の数をゼロにすることができる。すると、その結果作成されるクラスは、数個のプリミティブと基本的なライブラリで表現された概念と合わせれば、そのクラス単独で完全に理解できる。

あらゆるプログラミング環境において基本的要素のいくつかは非常に広く普及しているため、常に開発者の頭の中に入っている。例えば、Javaの開発では、プリミティブと数個の標準ライブラリによって数や文字列、コレクションといった基本要素が提供される。事実上、「整数」は精神的な負荷の一因にならない。だがそれ以外に、あるオブジェクトを理解するために付加的な概念を考えていなければならないのであれば、そうした付加的な概念は、どれも精神的過負荷を引き起こすのだ。

暗黙的な概念も、それが認識されているかどうかに関わらず、明示的な参照と同様に重要である。整数や文字列といったプリミティブな値への依存関係は、通常、無視しても構わないのだが、それが**表しているもの**を無視することはできない。例えば、最初の塗料混

合の例において、**塗料**オブジェクトは赤、黄、青の色合いを表す3つの公開された整数を持っていた。**顔料色**オブジェクトを作成しても、関係する概念の数や依存関係が増えることはなかった。すでにあったものを、より明示的に理解しやすくしただけだからだ。一方、**コレクション**の`size()`操作が戻す`int`は、単なる数であり、これは整数の基本的意味であるから、暗黙的に示された新しい概念は存在しない。

あらゆる依存関係は、オブジェクトの背後にある概念にとって本質的であると証明されるまでは、疑われるべきだ。この精査は、モデルの概念自体を分解することから始まる。それから個々の関連と操作に注意しなければならない。モデルと設計上の選択によって、依存関係を少しずつ削り落としていくことができるのであり、その結果、依存関係がなくなってしまうことも少なくない。

低結合はオブジェクト設計の基本である。できる時には徹底的にやること。オブジェクトのイメージの中から他の概念をすべて取り除くこと。そうすれば、クラスは完全に自己完結し、単独で調べて理解できるようになる。そういう自己完結型のクラスは、どれもモジュールを理解する際の負担を軽減してくれる。

同じモジュール内にある他クラスに対する依存関係は、外部のクラスに対する依存関係ほどには害がない。同様に、2つのオブジェクトが緊密に結合しているのが自然であるなら、同じペアを扱う複数の操作によって、関係の性質を実際に明確にすることができる。目標はすべての依存関係を取り除くことではない。本質的でない依存関係をすべて取り除くことなのだ。依存関係をすべて解消できなくても、1つ取り除かれるごとに開発者は自由になって、残る概念の依存関係に集中できる。

最も入り組んだ処理を独立したクラスへ分解するよう試みること。おそらくこれは、他とつながりの多いクラスで保持される値オブジェクトをモデル化することで実現できる。

塗料の概念は、本質的に色の概念と関係している。しかし、色というものは、顔料色であっても塗料に関係なく考察できる。これら2つの概念を明示的にし、両者の間の関係性を蒸留することで、残された一方向の関連は何か重要なことを意味するようになる。そして、複雑な演算のほとんどが置かれている**顔料色**クラスは、単独で調査し、**テストする**ことができるようになるのだ。

* * *

低結合は、概念的な過負荷を低減させるための基本的な方法である。そして、独立したクラスは低結合を極限まで進めたものだ。

依存関係を取り除くといっても、あらゆるものを恣意的にプリミティブにすることで、モデルの表現力を落とすという意味ではない。本章最後のパターンである閉じた操作は、依存関係を減らしつつ豊かなインタフェースを維持するテクニックの一例である…。

閉じた操作（CLOSURE OF OPERATIONS）

> 2つの実数を選び、それを掛け合わせると、また別の実数ができる。[実数は、あらゆる有理数とあらゆる無理数で構成される。] これは常に真であるから、実数は「乗算の下で閉じている」という。つまり、集合を逃れる方法はない。集合の中から任意の2つの要素を組み合わせると、結果もその集合に含まれる。
>
> —Drexel大学、数学フォーラム

当然、何らかの依存関係はあるはずだし、その依存関係が概念にとって本質的であるならば、あって悪いものでもない。インタフェースをそぎ落として、プリミティブしか扱わないようにしたら、そのインタフェースは貧弱になってしまうかもしれない。しかし、大量の不要な依存関係と、場合によっては概念全部さえもが、インタフェースに導入されてしまうのだ。

たいていの興味深いオブジェクトは、最終的に、プリミティブだけでは特徴づけることのできない処理を実行することになる。

精緻な設計に見られるもう1つの一般的なプラクティスに、私が「閉じた操作」と呼んでいるものがある。この名前は、最も精緻な概念の体系である、数学から取った。1 + 1 = 2である。加算は、実数の集合の下で閉じている。数学者は、異質な概念を導入することに対して狂信的なまでに反対するのであり、閉じているという性質を用いることで、彼らは他の概念を巻き込まずに操作を定義できるようになる。我々は数学の精緻さに慣れてしまっているため、このちょっとしたコツがどれほど強力かを把握するのは難しいかもしれない。しかし、これはソフトウェア設計でも広範に使用されている。XSLTの基本的な使い方は、あるXMLドキュメントを別のXMLドキュメントに変換することである。この種のXSLT操作は、XMLドキュメントの集合の下で閉じている。閉じているという性質によって、操作を解釈するのが大幅に単純化されるので、閉じた操作をつなぎ合わせ、組み合わせることについて考えるのが容易になる。

それゆえ：

戻り値の型が引数の型と同じにできる場合は、そのように操作を定義すること。実装クラスが処理で使われる状態を持っていれば、そのクラスは事実上操作の引数となるから、引数と戻り値は実装クラスと同じ型でなければならない。そういう操作は、その型のインスタンス集合の下で閉じている。ある閉じた操作によって、他の概念への依存関係を導入することなく、高度なインタフェースが提供される。

このパターンが最も頻繁に適用されるのは、値オブジェクトの操作である。エンティティのライフサイクルはドメインにおいて重要であるため、問いに答えるために新しいエンティティを魔法のように素早く生成することはできない。エンティティの型の下で閉じ

ている操作もある。**従業員**オブジェクトに上司を問い合わせ、別の**従業員**を得ることもできるだろう。だが、一般に、エンティティは処理の結果となるような類の概念ではない。したがって、たいていは値オブジェクトの中を探すと好機がある。

ある操作が抽象的な型の下で閉じていることもある。その場合、具体的な引数は別の具象クラスでもよい。結局、加算は実数の下で閉じていて、実数は有理数と無理数のどちらでもよいのだ。

実験しながら相互依存関係を減らして凝集度を高める方法を探している間に、このパターンへと至る道の途中まで来ることも多い。引数が実装側と一致するのに戻り値の型が異なっていたり、戻り値の型がレシーバと一致するのに引数が異なっていたりするということだ。こういう操作は閉じていないものの、閉じた操作の強みを多少は持っている。その型以外の余分な型がプリミティブや基本ライブラリクラスだけなら、閉じた操作とほぼ同程度まで精神的な負荷を下げてくれる。

先ほどの例で言えば、**顔料色**の`mixedWith()`操作は**顔料色**の下で閉じていたし、他にも複数の例が本書の中に散在している。次に挙げる例は、真の閉じた操作には至らない場合であっても、この考えがどれほど役に立ち得るかを示している。

例 10.5 ——コレクションから選択する

Javaでは、**コレクション**から要素のサブセットを選択したい場合に、**イテレータ**を要求する。次に、要素のすべてに対して繰り返し、それぞれを評価して一致するものを新しい**コレクション**に蓄積していく。

```
Set employees = (従業員オブジェクトのSet);
// 低賃金の従業員
Set lowPaidEmployees = new HashSet();
Iterator it = employees.iterator();
while (it.hasNext()) {
   Employee anEmployee = it.next();
   // 給料が40000以下かどうか
   if (anEmployee.salary() < 40000)
      lowPaidEmployees.add(anEmployee);
}
```

概念的には、ある集合のサブセットを選択したことになる。この余分な概念である**イテレータ**と、機械的な複雑さをどうすべきか？ Smalltalkでは、**コレクション**にある「選択 (select)」操作を呼び出し、引数として評価ロジックを渡す。戻り値となるのは、評価を通過した要素だけを含んだ新しい**コレクション**である。

```
employees := (従業員オブジェクトのSet).
lowPaidEmployees := employees select:
          [:anEmployee | anEmployee salary < 40000 ].
```

このSmalltalkのコレクションは、複数の具象クラスで構成されるコレクションを導き出して戻すような関数を、他にも提供している。この操作は閉じていないが、それは引数として「ブロック」を取っているからだ。しかし、ブロックはSmalltalkにおいて基本ライブラリ型だから、開発者にとって精神的負荷にはならない。戻り値が実装側と一致するため、フィルタを連続させてつなぎ合わせることもできる。読むのも書くのも簡単で、サブセットを選択するという問題に無関係な、異質な概念を持ち込むこともない。

＊　＊　＊

本章で取り上げるパターンが示しているのは、設計の一般的なスタイルと設計についての考え方である。ソフトウェアを明白で、予測可能で、伝達力のあるものにすると、抽象化とカプセル化による効力が発揮される。モデルを分解することで、オブジェクトを使うのも理解するのも単純になるが、それでも、高次の豊かなインタフェースを持ち続けることができる。

これらのテクニックを適用するには、かなり高度な設計スキルが必要になるし、そうしたスキルは時に、クライアントを作成する上でも必要となる。モデル駆動設計の有用性は、詳細な設計と実装上の意思決定の品質に影響されやすく、混乱した開発者がたった数人いるだけで、プロジェクトは目標から逸脱してしまう。

それでも、モデリングと設計のスキルを養う意欲のあるチームにとって、これらのパターンとそれを反映している考え方によって生み出されるソフトウェアは、開発者が作成と作り直しを繰り返しながら複雑にしていけるものなのだ。

宣言的な設計

表明を用いることで、テスト方法が比較的略式であっても、はるかに優れた設計を生み出すことができる。しかし、手書きのソフトウェアでは本当の保証が存在し得ない。表明に違反する方法の一例としては、コード内に、明確に除外されなかった余計な副作用がある場合が挙げられる。設計がどれほどモデル駆動であっても、結局は概念の相互作用という結果を生み出す手続きを書くことになる。さらに我々がほとんどの時間を費やして書く定型のコードは、実際に意味やふるまいを何もつけ加えない。こうしたコードは退屈でエラーに満ちあふれている上、そのほとんどがモデルの意味をわかりにくくする（言語によって良し悪しはあるが、どれを使っても単調な作業を大量に行わなければならない）。意図の明白なインタフェースや、その他の本章で紹介したパターンは、役には立つものの、従来のオブジェクト指向プログラムに正式な厳密さを与えることは決してできない。

宣言的な設計（declarative design）の背後にある動機のうち、いくつかはここに挙げたよ

●例 10.5——コレクションから選択する

うなものだ。この用語の意味は人によってさまざまだが、通常は、プログラムやその一部をある種の実行可能な仕様として作成する方法を指している。性質を厳格に記述することで、実際にソフトウェアを制御するのだ。取り得る形式はさまざまで、これはリフレクションの仕組みやコンパイル時のコード生成（宣言に基づいて、従来通りのコードを自動的に作り出すこと）を用いても実現できるかもしれない。このアプローチを用いれば、別の開発者が宣言を文字通りの意味に解釈できる。これは絶対的な保証だ。

モデルの特徴を宣言することで動作するプログラムを生成するということは、モデル駆動設計にとって究極の目標だが、実際には落とし穴もある。例えば、私が一度ならず遭遇した独特の問題を2つだけ以下に挙げる。

- 宣言を行う言語は、必要なことをすべて行えるほど表現力豊かではない。しかし、フレームワークがあると、ソフトウェアを自動化された部分より先へと拡張するのが非常に困難になる
- コード生成のテクニックは、イテレーションのサイクルを損なってしまう。これは、生成されたコードを手書きのコードに組み込む際に、再生成をするとコードが破壊されてしまうようなやり方を取っているからだ

宣言的な設計をしようとした多くの試みが招いた、意図していなかった帰結として、モデルやアプリケーションが簡単なものに書き直されてしまうというものがある。これは、開発者がフレームワークの制約にとらわれて、**何か**を完成させるために設計のトリアージを実施することによる。

インタフェースエンジンと、ベースとなるルールを持ったルールベースプログラミングも、宣言的な設計に対する有望なアプローチである。だが残念ながら、その意図は微妙な問題によって掘り崩されかねない。

ルールベースプログラムは原則として宣言的だが、ほとんどのシステムにはパフォーマンスチューニングを可能にするために追加された「制御述語」(control predicate) がある。この制御コードが副作用を持ち込むので、ふるまいはもはや、宣言されたルールによって完全には決定づけられなくなる。ルールの追加や削除あるいは再編成によって、予期しない誤った結果が引き起こされるかもしれない。だから、ロジックプログラマもオブジェクトプログラマ同様、コードの影響が明白であり続けるように注意しなければならない。

多くの宣言的なアプローチは、開発者に迂回されると、それが故意かどうかに関わらず、破綻する可能性がある。こうしたことは、システムを使うのが難しいか、過剰に制限がある場合に起こる。宣言的プログラムの恩恵を受けるためには、誰もがフレームワークのルールに従わなければならない。

これまでに私が見てきた中で、最も価値があったのは、スコープの狭いフレームワークが、特に退屈でエラーを生みやすい設計の側面を自動化する場合である。こうした側面に

は、永続化やO/Rマッピングなどがある。こうした中でも最高のものは、開発者を骨の折れる単純作業から解放する一方、設計する際の完全な自由を残している。

ドメイン特化言語

宣言的なかたちをとることもある興味深いアプローチに、**ドメイン特化言語**（domain-specific language）がある。このスタイルでは、クライアントコードを書く際に、特定のドメインの特定のモデル用にあつらえたプログラミング言語が用いられる。例えば、輸送システム用の言語には、**貨物**や**経路**といった用語が、それらを関連づけるシンタクスと一緒に含まれているかもしれない。このプログラムはその後コンパイルされ、たいていは従来のオブジェクト指向言語になるが、そこではクラスのライブラリによって、専用の言語で書かれた用語に実装が提供される。

こういう言語で書かれると、プログラムはきわめて表現力豊かになって、ユビキタス言語と最も強く結びつくことができる。これは刺激的な考え方だが、私の見てきたオブジェクト指向技術に基づくアプローチでは、ドメイン特化言語にも欠点がある。

モデルを改良するためには、開発者は言語を修正できなければならない。この時の修正対象には、根底にあるクラスライブラリだけなく、文法の宣言や、その他の言語解釈機能も含まれるかもしれない。高度な技術や設計概念を習得することに対して私は全面的に賛成だが、特定のチームのスキルに加え、将来の保守チームに予想されるスキルも冷静に評価しなければならない。また、価値があるのは、アプリケーションとモデルが同じ言語でシームレスに実装されている点だ。欠点としては、改良されたモデルとそれに関連するドメイン特化言語に従うよう、クライアントコードをリファクタリングするのが困難になるかもしれないということがある。もちろん、リファクタリングに伴うこの問題に対する技術的な解決策を、誰かが考えつくかもしれない。

このテクニックは非常に成熟したモデルにおいて最も有効かもしれない。そこでは、おそらくクライアントコードが別のチームによって作成されているだろう。そういう人員配置は、高度に技術的なフレームワーク作成者と、技術的スキルのないアプリケーション作成者、という有害な差別につながることがよくあるが、必ずそうなるわけでもない。

Schemeプログラミング言語では、ドメイン特化言語と非常に似たものが標準プログラミングスタイルの一部になっているから、ドメイン特化言語の持つ表現力はシステムを2つに分岐させなくても作り出せる。

コラム——基本から考える
別のパラダイムの方が、オブジェクトよりもドメイン特化言語をうまく処理するかもしれない。「関数型プログラミング」ファミリーの代表であるSchemeプログラミング言語では、ドメイン特化言語と非常に似たものが標準プログラミングスタイルの一部になっているから、ドメイン特化言語の持つ表現力はシステムを2つに分岐させなくても作り出せる。

設計の宣言的スタイル

設計に意図の明白なインタフェース、副作用のない関数、表明が揃ったら、宣言的な考え方の領域に少しずつ近づいている。宣言的な設計が持つ利点の多くを得るためには、複数の要素を組み合わせることができて、その要素が意味を伝えると同時に、影響を与える場合にはそれが特徴づけられているか明らかにされており、そうでなければ目に見える影響が全くないという状態であればよい。

しなやかな設計により、クライアントコードは設計の宣言的**スタイル**を使えるようになる。これについて説明するため、次節では本章にあるパターンをいくつか一緒に使い、仕様をよりしなやかで宣言的なものにする。

宣言的スタイルで仕様を拡張する

第9章で説明したのは、仕様の基本概念と、それがプログラム内で果たす役割、実装に含まれるものの意味であった。今度は、このオプション機能を少し見ていこう。これは複雑なルールの存在する状況で、非常に役立つかもしれない。

仕様は、確立された形式である、述語を改造したものだ。述語には他にも役立つ性質があり、我々は選んで利用できる。

論理演算子を使用して仕様を結合する

仕様を使っていると、それを組み合わせたいと思う状況にすぐに行き当たる。前述した通り、仕様は述語の一例であり、述語は「AND」、「OR」、「NOT」といった演算子を使って組み合わせ、修正することができる。これらの論理演算は述語の下で閉じているから、仕様の組み合わせは閉じた操作となる。

重要で一般的な機能が仕様に組み込まれることにより、あらゆる種類の仕様に使える、抽象クラスやインタフェースを作成することが非常に有益になる。これは、引数を高次の抽象クラスとして型づけすることを意味している。

```
public interface Specification {
   boolean isSatisfiedBy(Object candidate);
}
```

この抽象化によって、メソッドの始めにガード節が必要になるが†、それを除けば、機能には影響しない。例えば、**コンテナ仕様**（例9.5、238ページ）は、次のように変更される。

```
public class ContainerSpecification implements Specification {
   private ContainerFeature requiredFeature;

   public ContainerSpecification(ContainerFeature required) {
      requiredFeature = required;
   }

   boolean isSatisfiedBy(Object candidate) {
      // 候補がContainerのインスタンスでなければ、falseを返す
      if (!candidate instanceof Container) return false;

      return (Container)
         candidate.getFeatures().contains(requiredFeature);
   }
}
```

では、新しい処理を3つ追加することで、**仕様**インタフェースを拡張しよう。

```
public interface Specification {
   boolean isSatisfiedBy(Object candidate);

   Specification and(Specification other);
   Specification or(Specification other);
   Specification not();
}
```

コンテナ仕様の構成によっては、通気設備つき**コンテナ**を必要とするものもあれば、強化**コンテナ**を必要とするものもあったことを思い出して欲しい。揮発性で**かつ**爆発性でもある化学製品には、**両方**の仕様が必要になるだろう。新しいメソッドを使えば、これは簡単だ。

```
// 通気設備つき
Specification ventilated = new ContainerSpecification(VENTILATED);
// 強化型
Specification armored = new ContainerSpecification(ARMORED);

// 通気設備つき、かつ、強化型
Specification both = ventilated.and(armored);
```

† 訳注：Java 5.0で導入された総称型（ジェネリクス）を用いることで、型をコンパイル時にチェックできる。その場合、ガード節は不要になる。

この宣言は、期待された性質を持つ新しい**仕様**オブジェクトを定義している。こうした組み合わせを元のやり方で実現するには、さらに複雑な**コンテナ仕様**が必要となり、それでも相変わらず目的は特化されていただろう。

通気設備つき**コンテナ**が数種類あるとしよう。製品によっては、どの種類の**コンテナ**に詰めても構わないかもしれない。その場合には、どれにも入れられる。

```
// 通気設備つきタイプ1
Specification ventilatedType1 =
   new ContainerSpecification(VENTILATED_TYPE_1);
// 通気設備つきタイプ2
Specification ventilatedType2 =
   new ContainerSpecification(VENTILATED_TYPE_2);

// 通気設備つきタイプ1、または、通気設備つきタイプ2
Specification either = ventilatedType1.or(ventilatedType2);
```

特殊なコンテナに砂を入れるのが無駄だとすれば、特殊な機能のない「安価な」コンテナを**定義する**ことによって、それを防止できる。

```
// 安価なコンテナ＝通気設備つきではない、かつ、強化型ではない
Specification cheap = (ventilated.not()).and(armored.not());
```

この制約があれば、第9章で説明したプロトタイプの倉庫内格納クラスにあった、最善とは言いがたいふるまいを一部防げる。

単純な要素から複雑な仕様を組み立てる能力があると、コードの表現力が高まる。この組み合わせは、宣言的スタイルで書かれる。

仕様の実装方法次第で、これらの演算子を提供するのは、容易なこともあれば困難なこともある。次に挙げるのはきわめてシンプルな実装で、状況によって非効率的なこともあれば、きわめて実用的なこともある。これは**説明のための一例**である。他のパターンと同様に、実装する方法は数多くあるのだ。

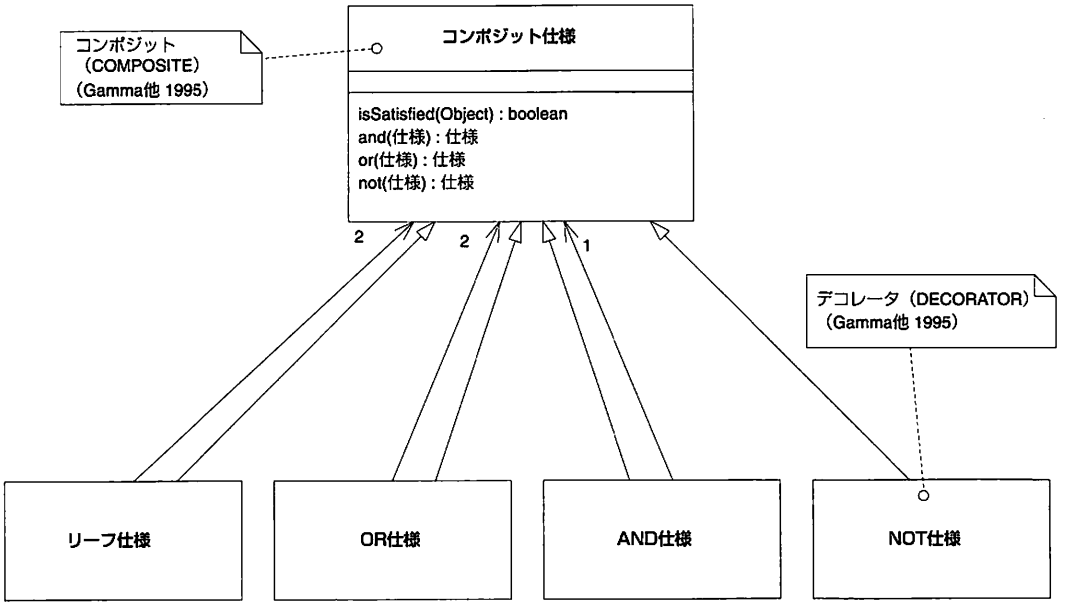


図10.14 **仕様**の合成の設計

```
/**
 * 抽象仕様.
 */
public abstract class AbstractSpecification implements
      Specification {
   public Specification and(Specification other) {
      return new AndSpecification(this, other);
   }
   public Specification or(Specification other) {
      return new OrSpecification(this, other);
   }
   public Specification not() {
      return new NotSpecification(this);
   }
}

/**
 * And仕様.
 */
public class AndSpecification extends AbstractSpecification {
   Specification one;
   Specification other;
   public AndSpecification(Specification x, Specification y) {
      one = x;
      other = y;
```

```
    }
    public boolean isSatisfiedBy(Object candidate) {
        return one.isSatisfiedBy(candidate) &&
            other.isSatisfiedBy(candidate);
    }
}

/**
 * Or仕様.
 */
public class OrSpecification extends AbstractSpecification {
    Specification one;
    Specification other;
    public OrSpecification(Specification x, Specification y) {
        one = x;
        other = y;
    }
    public boolean isSatisfiedBy(Object candidate) {
        return one.isSatisfiedBy(candidate) ||
            other.isSatisfiedBy(candidate);
    }
}

/**
 * Not仕様.
 */
public class NotSpecification extends AbstractSpecification {
    Specification wrapped;

    public NotSpecification(Specification x) {
        wrapped = x;
    }
    public boolean isSatisfiedBy(Object candidate) {
        return !wrapped.isSatisfiedBy(candidate);
    }
}
```

このコードは、本になっても読みやすいように書いてある。前述の通り、これでは効率が悪いという状況もあるだろう。だが、オブジェクト数を最小化したり、速度を上げたりする実装方法もあるだろうし、プロジェクトで使われている独自の技術と互換性を持たせることもできるだろう。重要なのは、ドメインの主要概念をとらえているモデルと、そのモデルに忠実な実装だ。性能問題を解決する余地は多く残されている。

また、多くの場合、これほど完全に一般化する必要はない。とりわけ、ANDは他の演算子より多用されやすく、実装もそれほど複雑にはならない傾向がある。必要なのがそれだけであれば、ANDのみ実装することをためらわなくてもよい。

第2章のシナリオ2（29ページ）に示した対話例では、開発者は仕様の「satisfied by」の

ふるまいを明らかに実装していなかった。その時点まで、仕様は注文のために構築する目的でしか使用されていなかった。それでも抽象は元のままだったし、機能の追加も比較的容易だったのである。パターンを使用するということは、必要もない機能を構築するということではない。概念が混乱しない限り、機能は後からでも追加できる。

例 10.6——コンポジット仕様を実装する他の方法

実装環境によっては、非常に粒度の細かいオブジェクトにあまりうまく対応できないものもある。かつて私が手がけたプロジェクトのオブジェクトデータベースでは、すべてのオブジェクトにオブジェクトIDを与えて、それを追跡することにこだわっていた。各オブジェクトによって、メモリ空間と性能に大きなオーバーヘッドがもたらされ、アドレス空間の総量に気をつけなければならなかった。私はドメインの設計における重要なポイントに仕様を採用した。それ自体は自分でも適切な決定だったと思う。だが、私が使ったのは、本章で述べたものよりもう少し手の込んだ実装であり、それは明らかに間違いだった。その結果、粒度の細かいオブジェクトが大量にできて、システムを停止させることになったのだ。

ここで紹介するのは、仕様を他の方法で実装する例である。コンポジット仕様をエンコードして、論理式をエンコードした文字列や配列に変換し、実行時に解釈するようにしている。

（この実装方法がわからなくても気にする必要はない。大切なのは、論理演算子を持つ仕様の実装方法が多数あるということを認識することだ。したがって、単純な方法が状況に合わなければ、他の選択肢がある。）

「安価コンテナ」向け**仕様**スタックの内容

トップ	And仕様演算子（フライウェイト）
	Not仕様演算子（フライウェイト）
	強化型
	Not仕様演算子
	通気設備つき

ある候補を評価したければ、この構造を解釈しなければならない。その解釈をするためには、各要素をポップしてから評価するか、演算子の要求に従って次をポップすればよい。すると、次のようなものができる。

```
and(not(armored), not(ventilated))
```

この設計には、長所（+）と短所（-）がある。

+ オブジェクト数が少ない
+ メモリが効率的に使用される
- より洗練された開発者が必要となる

自分の置かれた状況に適したトレードオフをしている実装を見つけなければならない。同じパターンとモデルが、全く異なる実装の基盤になり得るのだ。

包含（Subsumption）

この最後の機能は、常に必要とされるわけではなく、実装するのも難しいかもしれないが、時には本当に難しい問題を解決することがある。また、仕様の意味を明らかにする。

第9章の例9.5（238ページ）の化学製品倉庫内格納クラスの例を再考しよう。各**化学製品**には**コンテナ仕様**があり、**格納**サービスによって**ドラム缶**が**コンテナ**に割り当てられた時に、これらの仕様がすべて満たされることを保証していた。万事良好だ…。ただし、誰かが規則を変更するまでの話だ。

数か月ごとに新しいルールが一式発効されるので、ユーザは要求が厳しくなった化学製品の型の一覧を作成できるようにしたいと思っている。

無論、これに部分的に回答することはできる（そしておそらく、ユーザも部分的な回答を望んでいる）。新しい仕様を使って、在庫の各**ドラム缶**に対して検証を行い、仕様に合わなくなったものをすべて見つけ出せばよい。こうすることで、ユーザは既存の在庫の中で、どの**ドラム缶**を移動するべきかがわかる。

しかし、ユーザが**求めている**のは、取り扱いがより厳重になった化学製品の一覧である。現在は該当するものがないかもしれないし、必要以上に厳重なコンテナに偶然入れられているのかもしれない。いずれにしても、前述した帳票には挙げられない。

2つの仕様を直接比較するために、新しい操作を取り入れることにしよう。

```
boolean subsumes(Specification other);
```

より厳重な仕様はそれほど厳しくない方の仕様を含んでいる。それまでの要求を無視することなく、置き換えることができるのだ。

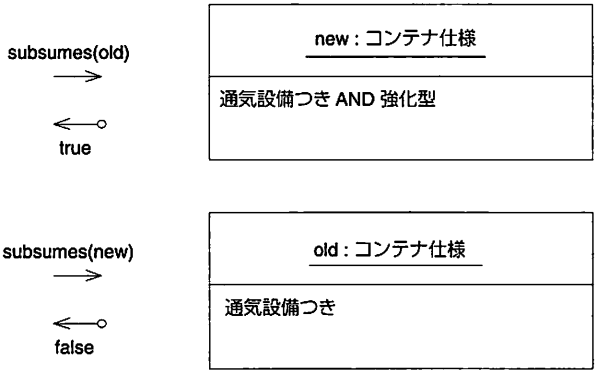


図10.15 ガソリンコンテナの**仕様**が厳しくなった

仕様の言語を用いると、新しい仕様が古い仕様を**包含する**（subsume）という言い方をするかもしれない。新しい仕様を満たす候補はどれも、古い仕様を満たしているからだ。

これらの仕様をそれぞれ述語として見ると、包含は論理的含意（logical implication）に相当する。従来の表記法を使用すると、$A \rightarrow B$は文Aが文Bを含むことを意味するから、Aが真ならばBも真である。

このロジックを、コンテナを突き合わせるという要求に適用しよう。仕様が変更された場合、提案された新しい仕様が古い仕様の条件をすべて満たすかどうかが知りたいのだ。

新しい仕様 → 古い仕様

すなわち、新しい仕様が真ならば、古い仕様も真である。論理的含意を一般的な方法で証明するのは非常に難しいが、特殊なケースだけでよければ簡単かもしれない。例えば、特定のパラメータ化された仕様は、独自の包含ルールを定義できる。

```
/**
 * 最小年齢仕様.
 */
public class MinimumAgeSpecification {
    // 境界値
    int threshold;

    public boolean isSatisfiedBy(Person candidate) {
        return candidate.getAge() >= threshold;
    }

    public boolean subsumes(MinimumAgeSpecification other) {
        return threshold >= other.getThreshold();
    }
}
```

JUnitテストには、次のコードが書かれているかもしれない。

```
// 運転できる年齢
drivingAge = new MinimumAgeSpecification(16);

// 投票できる年齢
votingAge = new MinimumAgeSpecification(18);

// 投票できる年齢は運転できる年齢を包含する
assertTrue(votingAge.subsumes(drivingAge));
```

コンテナ仕様の問題を扱うのに適した、実用的な特殊ケースをもう1つ挙げよう。これは、論理演算子のうちANDだけと、包含を組み合わせた仕様インタフェースである。

```
public interface Specification {
   boolean isSatisfiedBy(Object candidate);
   Specification and(Specification other);
   boolean subsumes(Specification other);
}
```

AND演算子だけを用いた包含の証明は単純である。

$$A \text{ AND } B \rightarrow A$$

もっと入り組んだ場合では、次のようになる。

$$A \text{ AND } B \text{ AND } C \rightarrow A \text{ AND } B$$

そこで、**コンポジット仕様**が「AND」で結びつけられたリーフ仕様をすべて集めることができれば、検証すべきことは、包含している側の仕様が包含されている側のリーフをすべて含んでいるということと、余分なものもあるかもしれないということだけだ。包含する側のリーフは、もう一方の仕様の持つリーフの集合にとって、上位の集合となる。

```
public boolean subsumes(Specification other) {
   if (other instanceof CompositeSpecification) {
      Collection otherLeaves = (CompositeSpecification)
         other.leafSpecifications();
      Iterator it = otherLeaves.iterator();
      while (it.hasNext()) {
         if (!leafSpecifications().contains(it.next()))
            return false;
      }
   } else {
      if (!leafSpecifications().contains(other))
         return false;
   }
   return true;
}
```

この相互作用を拡張すれば、慎重に選ばれ、パラメータ化されたリーフ仕様と、他の複雑な仕様とを比較するようにもできる。残念ながら、ORとNOTが含まれる場合は、その証明は格段に複雑になる。たいていの状況では、演算子のいくつかをあきらめるか、包含をあきらめることで、そういう複雑さは回避した方がよい。どちらも必要であるならば、それによる困難を正当化できるほど利点が大きいかどうかを慎重に検討すること。

仕様で表現されたアリストテレス

すべての人間は死すべき存在である。	`Specification manSpec = new ManSpecification();`
	`Specification mortalSpec = new MortalSpecification();`
	`assert manSpec.subsumes(mortalSpec);`

アリストテレスは人間である。	`Man aristotle = new Man();` `assert manSpec.isSatisfiedBy(aristotle);`
したがって、アリストテレスは死すべき存在である。	`assert mortalSpec.isSatisfiedBy(aristotle);`

攻める角度

本章では多くのテクニックについて紹介してきたが、これらはコードの意図を明確にし、コードを使った結果を透過的にし、さらにモデル要素を分離するためのものである。そうは言っても、この種の設計は難しい。巨大なシステムを見て、単に「これをしなやかにしよう」というわけにはいかない。対象を選ばなければならないのだ。ここでは、広く適用できるアプローチを2つ挙げた上で応用例を示し、より大きな設計を引き受けるのにあたって、どのようにパターンを適合させ、利用するかを説明する。

サブドメインを切り取る

設計全体に対して、一度に取り組むことはできない。一部を切り出すこと。システムの側面によっては、とるべきアプローチを示唆するものもあり、そうした側面は、括り出してうまく解決できる。モデルの一部が特殊な数学と見なせることがわかったら、それを切り離すこと。状態の変更を制限する複雑なルールが、アプリケーションによって強制されているのであれば、それを抜き出して、分離したモデルか単純なフレームワークにし、そのルールを宣言できるようにすること。そういう手順を重ねるごとに、新しいモジュールが整理されるだけでなく、残った部分も小さく整理されたことになる。残りは、宣言的スタイル、特殊な数学や検証フレームワークの用語を用いた宣言など、サブドメインのとる何らかの形式で記述される。

ある領域にだけ大きな衝撃を与え、設計の一部を本当にしなやかにする方が、広く薄く

努力するよりも有効である。第15章では、どのようにサブドメインを選択し、管理するかについて議論する。

可能な場合には、確立された形式主義を活用する

緻密な概念フレームワークを一から作成するのは、毎日できることではない。プロジェクトが進む中では、そうしたフレームワークのうちの1つが発見されて改良されることはある。だが、対象とするドメインやその他のドメインで確立されて久しい概念体系を使って、それをプロジェクトに順応させることも、しばしばあるのだ。これらの中には何世紀にもわたって改良され蒸留されてきたものもある。例えば、多くのビジネスアプリケーションは会計を含んでいる。会計は、十分に発達したエンティティとルールの集合を定義しており、そのために深いモデルとしなやかな設計に容易に適用できる。

このような形式化された概念フレームワークは数多くあるが、私が個人的に好きなのは数学である。基本的な計算に多少工夫を加えたものを引き出す上で、数学は驚くほど有用なのだ。多くのドメインがどこかに数学を含んでいる。それを探して、掘り起こすこと。特殊な数学には不要なものがなく、明確なルールによって結合することができて、理解するのも簡単だと感じさせることができる。私が過去に経験した一例は「シェア算」(Shares Math) だが、その説明で本章を締めくくることにする。

例 10.7 ——パターンを統合する：シェア算

第8章では、シンジケートローンシステムを構築するプロジェクトで起きた、モデルのブレイクスルーについて語った。今度はこの例の詳細に踏み込み、設計の中の1つの機能だけに焦点を絞って、プロジェクトの機能と比較する。

このアプリケーションの要件の1つに、借り手が元金を支払う際、初期設定では、ローンにおける貸し手のシェアに従って金額が比例配分されるというものがあった。

支払い配分の初期設計

リファクタリングするにつれ、このコードは理解しやすくなっていくから、このバージョンで行き詰まらないこと。

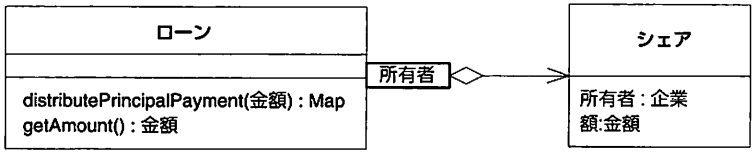


図10.16

```
/**
 * ローン.
 */
public class Loan {
   // シェア
   private Map shares;

   // アクセッサ、コンストラクタ、非常に単純なメソッドは除外してある

   /**
    * 元金を配分する.
    * @param paymentAmount 支払い総額
    * @return 貸し手をキー、シェアを値としたマップ
    */
   public Map distributePrincipalPayment(double paymentAmount) {
      Map paymentShares = new HashMap();
      Map loanShares = getShares();
      double total = getAmount();
      Iterator it = loanShares.keySet().iterator();
      while(it.hasNext()) {
         Object owner = it.next();
         // 初期のローンシェア額
         double initialLoanShareAmount = getShareAmount(owner);
         // 支払いシェア額 = 初期のローンシェア額 / 総額 * 支払い額
         double paymentShareAmount =
            initialLoanShareAmount / total * paymentAmount;
         Share paymentShare = new Share(owner, paymentShareAmount);
         paymentShares.put(owner, paymentShare);

      // 変更後のローンシェア額 = 初期ローンシェア額 - 支払いシェア額
      double newLoanShareAmount =
         initialLoanShareAmount - paymentShareAmount;
         Share newLoanShare = new Share(owner, newLoanShareAmount);
         loanShares.put(owner, newLoanShare);
      }
      return paymentShares;
   }

   /**
    * 総額を取得する.
    * @return ローンの総額
    */
   public double getAmount(){
      Map loanShares = getShares();
      double total = 0.0;
      Iterator it = loanShares.keySet().iterator();
      while(it.hasNext()) {
         Share loanShare = (Share) loanShares.get(it.next());
         total = total + loanShare.getAmount();
      }
      return total;
   }
}
```

●例 10.7——パターンを統合する：シェア算

コマンドと副作用のない関数を分離する

この設計には、すでに意図の明白なインタフェースがある。しかし、distributePrincipalPayment()メソッドは危険なことを行っている。すなわち、配分のシェアを計算した上に、ローンも変更しているのである。リファクタリングして、コマンドからクエリを切り離そう。

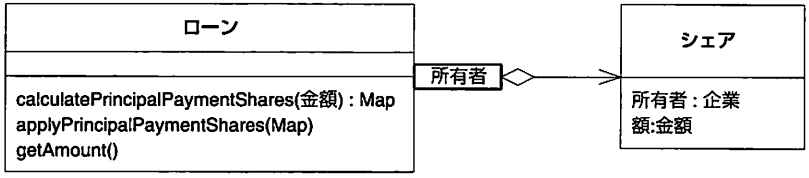


図10.17

```
/**
 * 元金支払いのシェアを適用する.
 * @param paymentShares 支払いシェア
 */
public void applyPrincipalPaymentShares(Map paymentShares) {
   Map loanShares = getShares();
   Iterator it = paymentShares.keySet().iterator();
   while(it.hasNext()) {
      Object lender = it.next();
      Share paymentShare = (Share) paymentShares.get(lender);
      Share loanShare = (Share) loanShares.get(lender);
      double newLoanShareAmount =
         loanShare.getAmount() - paymentShare.getAmount();
      Share newLoanShare = new Share(lender, newLoanShareAmount);
      loanShares.put(lender, newLoanShare);
   }
}

/**
 * 元金支払いシェアを計算する.
 * @param paymentAmount 支払い額
 * @return 貸し手をキー、シェアを値としたマップ
 */
public Map calculatePrincipalPaymentShares(double paymentAmount) {
   Map paymentShares = new HashMap();
   Map loanShares = getShares();
   double total = getAmount();
   Iterator it = loanShares.keySet().iterator();
   while(it.hasNext()) {
      Object lender = it.next();
      Share loanShare = (Share) loanShares.get(lender);
      double paymentShareAmount =
         loanShare.getAmount() / total * paymentAmount;
      Share paymentShare = new Share(lender, paymentShareAmount);
```

```
            paymentShares.put(lender, paymentShare);
        }
        return paymentShares;
    }
```

クライアントコードは次のようになる。

```
// ローンの元金シェアを計算する
Map distribution =
    aLoan.calculatePrincipalPaymentShares(paymentAmount);
// ローンに元金シェアを適用する
aLoan.applyPrincipalPaymentShares(distribution);
```

悪くない。この関数は意図の明白なインタフェースの陰に大量の複雑さをカプセル化している。しかし、applyDrawdown()、calculateFeePaymentShares()などを追加すると、確実にコード量は増え始める。拡張するたびにコードは入り組んでいき、重さで押しつぶされてしまう。粒度が粗すぎるのかもしれない。従来のアプローチに従えば、計算するメソッドを分解してサブルーチンにすることになる。それでも途中まではうまくいくだろうが、最終的にやりたいのは、根底にある概念の境界を見えるようにして、モデルを深めることだ。そうした概念的な輪郭となる粒度を用いた設計の要素であれば、組み合わせることによって必要なバリエーションを作り出せるだろう。

暗黙的な概念を明示的にする

今や、新しいモデルの探究を始めるのに十分な情報が揃った。この実装においては、シェアオブジェクトは受動的であり、複雑で低次の方法で操作されている。これは、シェアに関するルールと計算のほとんどが、1つのシェアではなく、シェアのグループに適用されることによる。ここには、ある概念が欠けているのだ。実はシェアは、全体を構成する部分として相互に関係しているのである。この概念を明示的にすることで、ルールと計算をより簡潔に表現できるようになる。

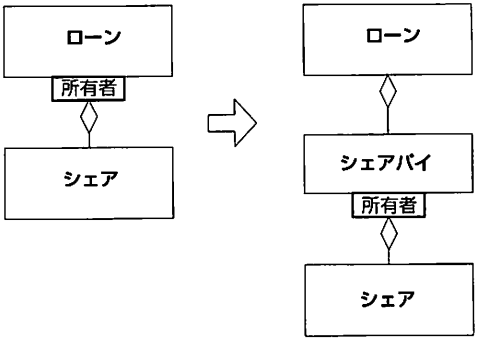


図10.18

シェアパイは、ある特定のローンの配分全体を表す。これは、ローンの集約内において局所的な同一性を持つエンティティである。実際の配分計算はシェアパイに委譲できる。

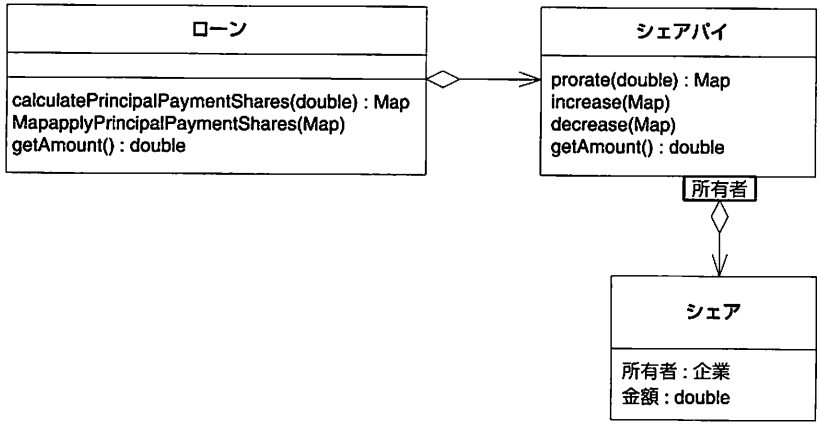


図10.19

```
/**
 * ローン.
 */
public class Loan {
    // シェアパイ
    private SharePie shares;

    // アクセッサ、コンストラクタ、その他、当たり前のメソッドは省略してある

    /**
     * 元金支払い配分を計算する.
     * @param paymentAmount 支払い額
     * @return 支払いシェア
     */
    public Map calculatePrincipalPaymentDistribution(
            double paymentAmount) {
        // 比例配分する
        return getShares().prorated(paymentAmount);
    }

    /**
     * 元金支払いを適用する.
     * @param paymentShares 支払いシェア
     */
    public void applyPrincipalPayment(Map paymentShares){
        // 支払いシェアの分、減少させる
        shares.decrease(paymentShares);
    }
}
```

これでローンは単純になり、シェアの計算はその責務に集中する値オブジェクトに集められた。それでもまだこの計算は、本当の意味で、多くの用途で使われるようになったり、使いやすくなったりしたわけではない。

シェアパイが値オブジェクトになる：洞察の連鎖

新しいモデルを実装するという実践的な経験は、しばしばモデル自体に対する新しい洞察を生み出すきっかけとなる。この場合では、ローンとシェアパイが緊密に結合しているせいで、シェアパイとシェアの関係がわかりにくくなっているようだ。シェアパイを値オブジェクトにしたら、どうなるだろう？

これが意味するのは、`increase(Map)`と`decrease(Map)`が認められなくなるということだ。シェアパイを不変にしなければならないからである。シェアパイの値を変更するには、パイ全体を置き換えなければならない。そこで、`addShares(Map)`のような操作を作り、全く新しい、より大きくなったシェアパイを戻させればよいだろう。

閉じた操作まで一気に進んでしまおう。シェアパイを「増やし」たり、それにシェアをつけ加えたりせずに、2つのシェアパイを足すだけでよいのだ。その結果、新しい、より大きなシェアパイができる。

シェアパイに対する`prorate()`（比例配分する）操作を部分的に閉じるには、戻り値の型を変更するだけでよい。名前を`prorated()`（比例配分された）に変えれば、副作用のないことが強調される。「シェア算」のかたちが見え始めた。まずは4つの操作からである。

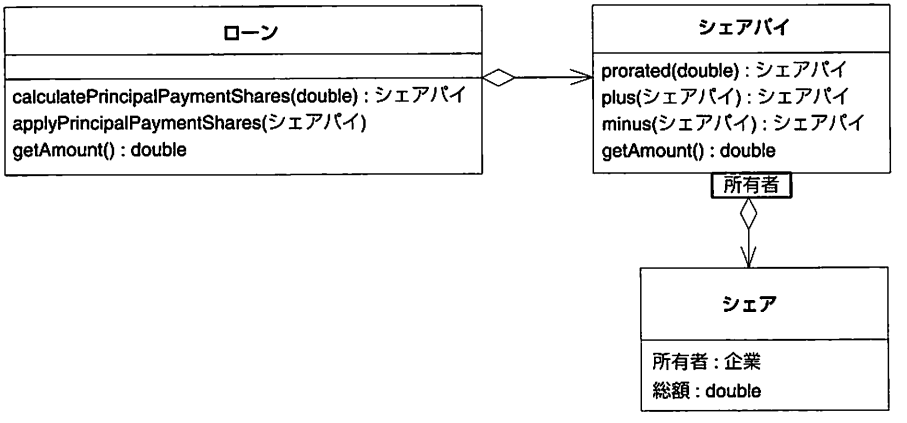


図10.20

新しい値オブジェクトであるシェアパイに対しては、明確に定義された表明を作成できる。それぞれのメソッドには何らかの意味があるのだ。

```
public class SharePie {
   private Map shares = new HashMap();

   // アクセッサと、その他の、当たり前のメソッドは省略してある

   public double getAmount() {
      double total = 0.0;
      Iterator it = shares.keySet().iterator();
      while(it.hasNext()) {
         Share loanShare = getShare(it.next());
         total = total + loanShare.getAmount();
      }
      return total;
   }

   public SharePie minus(SharePie otherShares) {
      SharePie result = new SharePie();
      Set owners = new HashSet();
      owners.addAll(getOwners());
      owners.addAll(otherShares.getOwners());
      Iterator it = owners.iterator();
      while(it.hasNext()) {
         Object owner = it.next();
         double resultShareAmount = getShareAmount(owner) -
            otherShares.getShareAmount(owner);
         result.add(owner,resultShareAmount);
      }
      return result;
   }

   public SharePie plus(SharePie otherShares) {
      // minus()の実装に類似している
   }

   public SharePie prorated(double amountToProrate) {
      SharePie proration = new SharePie();
      double basis = getAmount();
      Iterator it = shares.keySet().iterator();
      while(it.hasNext()) {
         Object owner = it.next();
         Share share = getShare(owner);
         double proratedShareAmount =
            share.getAmount() / basis * amountToProrate;
         proration.add(owner, proratedShareAmount);
      }
      return proration;
   }
}
```

全体は部分の合計に等しい。

2つのパイの差はそれぞれの所有者のシェアの差である。

2つのパイの結合はそれぞれ所有者のシェアを結合したものである。

総額はすべてのシェアの所有者の間で比例配分することができる。

新しい設計のしなやかさ

この時点で、最も重要なローンクラスにあるメソッドは、次に示すくらいにまで単純化されている。

```
/**
 * ローン.
 */
public class Loan {
   // シェアパイ
   private SharePie shares;

   // アクセッサ、コンストラクタ、その他、当たり前のメソッドは省略してある

   /**
    * 元金支払い配分を計算する.
    * @param paymentAmount 支払い額
    * @return シェアパイ
    */
   public SharePie calculatePrincipalPaymentDistribution(
            double paymentAmount) {
      return shares.prorated(paymentAmount);
   }

   /**
    * 元金支払いを適用する.
    * @param paymentShares 支払いシェア
    */
   public void applyPrincipalPayment(SharePie paymentShares) {
      setShares(shares.minus(paymentShares));
   }
}
```

これらの短いメソッドは、それぞれ意味をはっきりと述べている。元金支払いの適用が意味しているのは、シェアごとに支払い分をローンから引くということだ。元金支払いの配分は、シェアの所有者の間で、一定の割合に従って総額を分けることにより行われる。シェアパイの設計によって、ローンのコードで宣言的スタイルが使えるようになった。これによって作り出されるコードは、計算としてではなく、ビジネスにおける取引の概念的な定義のように読める。

他の取引タイプ（以前は複雑すぎて挙げられなかった）も、今では簡単に宣言できる。例えば、ローンの引き出しは、ファシリティのシェアに基づいて貸し手の間で分割される。新しい引き出しは未払いのローンに追加される。これは、新しいドメイン言語では、次のように書ける。

```
public class Facility {
   private SharePie shares;
   ...
   public SharePie calculateDrawdownDefaultDistribution(
         double drawdownAmount) {
      return shares.prorated(drawdownAmount);
   }
}

public class Loan {
   ...
```

●例 10.7 ——パターンを統合する：シェア算

```
    public void applyDrawdown(SharePie drawdownShares) {
        setShares(shares.plus(drawdownShares));
    }
}
```

それぞれの貸し手が合意していた出資からどの程度乖離しているかを知るには、未払いローン総額の理論上の配分を取得して、それをローンの実際のシェアから引けばよい。

```
SharePie originalAgreement =
    aFacility.getShares().prorated(aLoan.getAmount());
SharePie actual = aLoan.getShares();
SharePie deviation = actual.minus(originalAgreement);
```

シェアパイの設計にあるいくつかの特徴のおかげで、このように容易に組み替え、コードによって意図を伝えることができる。

- **複雑なロジックが、副作用のない関数を持つ、特化した値オブジェクトにカプセル化されている。**複雑なロジックのほとんどが、これらの不変オブジェクトにカプセル化された。シェアパイは値オブジェクトなので、数学的な操作によって新しいインスタンスを生成できるし、新しいインスタンスを自由に使って、古くなったインスタンスを置き換えることができる。

  シェアパイにあるこれらのメソッドで、既存のオブジェクトを変更するものはない。そのため、plus()、minus()、prorated()は、中間的な計算で自由に使うことができ、組み合わせることもできる。その名前が示す処理だけを実行し、それ以外のことは一切行わないと考えることもできるのだ。また、これらと同じメソッドに基づいて、分析機能を構築することもできる（以前、これらのメソッドは、実際の配分が行われる時にしか呼び出せなかった。それぞれを呼び出した後にデータが変更されたかもしれなかったからだ）。
- **状態を変更する操作が単純で、表明により特徴づけられている。**シェア算が高度に抽象化されているため、取引の不変条件を宣言的スタイルで簡潔に書ける。例えば乖離は、実際のパイから、ファシリティのシェアパイに基づいて比例配分されたローン総額を引いたものである。
- **モデルの概念が分離されているので、操作に巻き込まれる他の型は最小限になる。**シェアパイ上にあるいくつかのメソッドには、閉じた操作となっているもある（加算または減算のメソッドはシェアパイの下で閉じている）。他に、引数や戻り値として単純な金額を取るものもある。それらは閉じてはいないが、概念的な負荷をほとんどかけない。シェアパイが密接に相互作用する他のクラスはただ1つ、シェアだけである。結果として、シェアパイは自己完結的で、理解しやすく、テストも簡単で、宣言的トランザクションを形成するために簡単に組み合わせることができる。こうした性質は、数学の形式主義から継承したものだ。
- **よく知られた形式主義によって、プロトコルが把握しやすくなる。**シェアを操作する、完

全に独自のプロトコルを、金融用語に基づいて考え出すこともできた。原理上は、しなやかにすることもできただろう。しかし、そうしていたとすれば、欠点が2つあったはずだ。第一に、そのプロトコルを考え出さなければならなかっただろう。この考え出すという作業は、難しく不確かである。第二に、そのプロトコルを扱う人全員が習得しなければならなかっただろう。シェア算を見れば、すでに自分の知っている数学の体系だと認識できる。また、設計は計算のルールと矛盾しないように注意深く保たれてきたから、そのプロトコルを扱う人が誤解することもない。

問題から数学の形式主義に対応する部分を抜き出すことで、シェアというしなやかな設計に到達し、その設計により、コアとなるローンとファシリティのメソッドがさらに蒸留されるのである（コアドメインの説明は、第15章を参照）。

しなやかな設計は、ソフトウェアが変化と複雑さに対処できるようになる上で、底知れない効果がある。本章の例で示したように、こうしたことは、きわめて詳細なモデリングと設計上の意思決定にかかっていることが多い。その影響力が及ぶ範囲は、特定のモデリングや設計の問題を超えるかもしれない。第15章では、しなやかな設計の持つ戦略的な価値について議論する。そこでは、しなやかな設計は、ドメインモデルを蒸留して、巨大で複雑なシステムを扱いやすいものにするためのツールの1つとして扱われる。

第11章

アナリシスパターンを適用する

深いモデルとしなやかな設計は、容易に手に入るものではない。こうしたモデルや設計を発展させるためには、ドメインについて大いに学び、多くの話し合いと試行錯誤を重ねる必要がある。もっとも、時にはよいスタートが切れることもある。

経験のある開発者がドメインの問題を検討していて、よく知っている責務や関係性の網の目を見つけたならば、その問題が以前にどう解決されたかという記憶を活用できる。どういうモデルが試され、どれが効果的だったのか？ 実装時にどのような難題が発生し、それをどう解決したのか？ そういう過去の経験における試行錯誤が、突然、新しい状況と深く関連づけられるのだ。こうした経験に基づくパターンのうちいくつかは、ドキュメント化され共有されているので、我々もその蓄積された経験を活用できる。

第2部で紹介した基本的な構成要素のパターンや、第10章のしなやかな設計とは対照的に、そういうパターンはより高いレベルにあり、より特化していて、何らかの概念を表すのにオブジェクトを複数使用する。これらを使うことで、高くつく試行錯誤を手早くすませられる。この時に出発点となるモデルは、すでに表現力豊かで、実装可能であり、習得するには高くつきそうな機微に対応しているのだ。そこから、リファクタリングと実験を行う。これは、そのまま使える既製の解決策といったものではない。

『アナリシスパターン』においてMartin Fowlerは、自身のパターンを次のように定義している。

> アナリシスパターンとは、ビジネスモデリングにおける共通の構造を表す、概念のグループである。ただ1つのドメインにしか適さないものもあれば、多くのドメインに及ぶものもある。(Fowler 1997, p.8)

Fowlerが紹介しているアナリシスパターンは、当該する分野での経験を基にしているた

め、適切な状況で用いれば実践的である。こういうパターンは、骨の折れるドメインに直面している人に対して、イテレーティブな開発プロセスを始めるにあたっての、非常に有益な出発点を提供する。このアナリシス（分析）という名前が、その概念的な性質を強調している。アナリシスパターンは技術的な解決策ではなく、特定のドメインでモデルを考え出すのを支援する手引きなのである。

このアナリシスパターンという名前からは残念ながら**伝わらない**のが、これらのパターンには実装に関する意義深い議論があり、ある程度コードも含まれているということである。Fowlerは実践的な設計を考慮しない分析の落とし穴を理解している。次に挙げる興味深い例では、Fowlerはリリースの先まで見越し、特定のモデルの選択が、その分野におけるシステムの長期的な保守に対して与える影響を考えている。

> 新しい［会計］のプラクティスを構築する際、我々が作成するのは、記帳ルールの新しいインスタンスで構成されるネットワークである。これには再コンパイルもシステムの再構築も必要なく、システムが稼動している最中に実行できる。記帳ルールの新しいサブタイプが必要になることもあるだろうが、それはまれである。[p.151]

成熟したプロジェクトでは、モデルについて選択する際に、そのアプリケーションに関する経験から情報を得られることがしばしばある。それまでに、多様なコンポーネントについて複数の実装が試されているのだろう。そのうちのいくつかは製品化され、保守フェーズに入ったものさえあるはずだ。そういう経験から得られるものがあるなら、多くの問題が回避できる。アナリシスパターンは、うまく行けば他のプロジェクトからそういった種類の経験を持ってきて、モデルについての洞察を、設計の方向性と実装の結果に関する応用的な議論と結びつけることができる。このコンテキストから外れてモデルの考え方を議論すると、適用は困難となり、分析と設計との間に致命的な溝が生じる危険がある。これでは、モデル駆動設計と正反対だ。

アナリシスパターンの原則とその適用は、抽象的に記述するよりも、例を示した方がうまく説明できる。本章では例を2つ示す。ここでは開発者が、『アナリシスパターン』の「在庫管理と会計（Inventory and Accounting）」の章で示されている、代表的な小さいモデルを利用しようとする。アナリシスパターンは、これらの例を紹介する上で必要な程度にまで要約されている。当然、この種のパターンのカタログを作ったり、サンプルのパターンをすべて説明したりするつもりはない。重要なのは、アナリシスパターンがドメイン駆動設計プロセスにどう統合されるかを示すことである。

例 11.1 ——利息を得る　勘定を用いた場合

第10章で示したのは、ある専門的な会計アプリケーションのために、深いモデルを探索する上で取り得るさまざまな方法だった。ここで、さらに別のシナリオを取り上げよう。今回は、開発者が役に立つ考え方を求めて、Fowlerの『アナリシスパターン』を調べる。

復習しておくと、ローンやその他の利息が発生する資産を追跡するあるアプリケーションでは、発生した利息と手数料を計算し、債務者からの支払を追跡する。夜間バッチプロセスは、それらの数字を受け取って、レガシーの会計システムに渡す。その際、それぞれの総額を記帳すべき元帳が指定される。この設計は機能するが、使い方はぎこちなく、変更するにも注意が必要で、意図をうまく伝えていない。

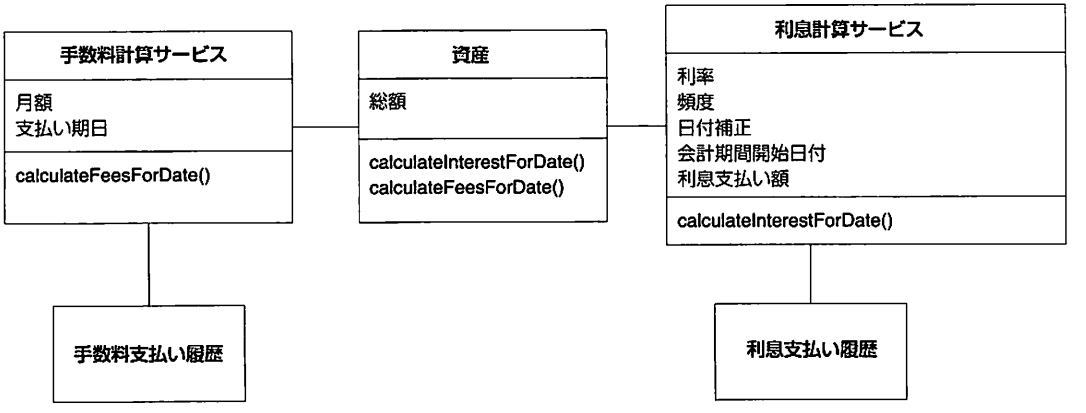


図 11.1　初期のクラス図

今回の開発者は、『アナリシスパターン』の第6章「在庫管理と会計」を読むことにした。彼女がこのアプリケーションと最も関連が深いと思った部分を要約すると、次のようになる。

『アナリシスパターン』における会計モデル

あらゆるビジネスアプリケーションは勘定科目を追跡するが、そこでは価値のあるもの、すなわち、一般的には資金が保持されている。多くのアプリケーションでは、勘定科目の総額を追跡するだけでは不十分である。本質的なのは、総額に対して行われる変更をそれぞれ明らかにし、管理することだ。このことが、最も基本的な会計モデルにとっての動機となっている。

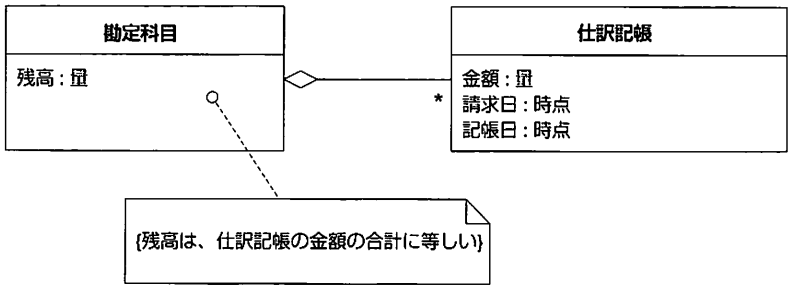


図11.2 基本的な会計モデル

値は**仕訳記帳**（Entry）を挿入することで増やせるし、負の**仕訳記帳**を挿入すれば減らせる。**仕訳記帳**は決して削除されないので、すべての記録が保持される。残高はすべての**仕訳記帳**を合計した結果である。この残高は逐次計算しても、キャッシュしてもよく、その実装に関する決定は、**勘定科目**（Account）インタフェースによってカプセル化される。

会計の基本原則は**保存**（conservation）である。資金は何もないところからは現れないし、跡形もなく消えるわけでもない。ただ、ある**勘定科目**から別の**勘定科目**へ移されるだけだ。

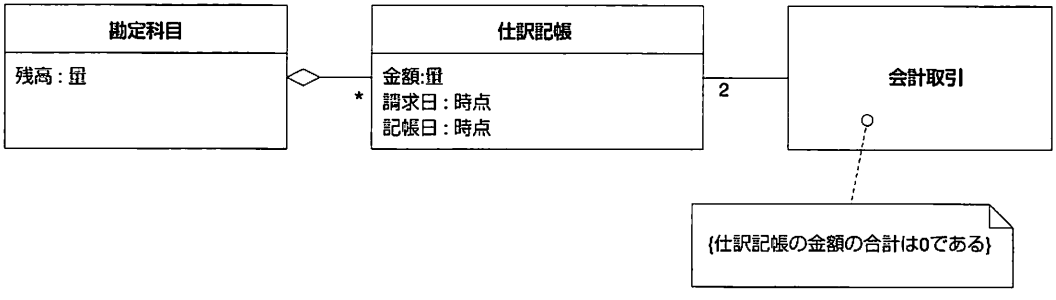


図11.3 会計取引モデル

これは**複式簿記**という確立された概念である。各貸方には対応する借方がある。もちろん、他の保存原則と同じように、これは閉じたシステム、つまりすべての出所と行き先が含まれたシステムだけに適用される。多くの単純なアプリケーションは、これほど厳密である必要はない。

『アナリシスパターン』においてFowlerは、これらのモデルの手の込んだ形態を扱い、トレードオフについて注目に値する議論を行っている。

この本を読むことで、開発者（**開発者1**）はいくつかの新しい考え方を学んだ。彼女はその章を同僚（**開発者2**）に見せた。この同僚は、利息計算ロジックをいくつか彼女と一緒に作成した人物であり、夜間バッチプログラムを作成した人物でもある。2人は共同で、自分たちのモデルに対する変更を大まかに描いた。その際に、本にあったモデル要素をいくつか組み込んでいる。

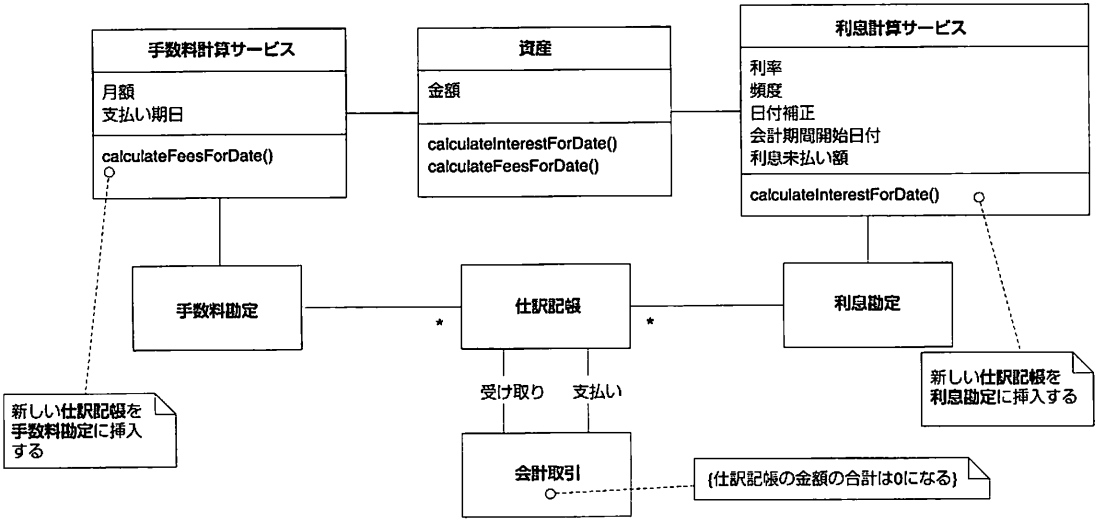


図11.4　新しいモデルの提案

さらに彼らは、ドメインエキスパート（**エキスパート**）を、新しいモデルの考え方についての議論に招き入れた。

開発者1：この新しいモデルでは、得た利息に対して、**利息勘定**に**仕訳記帳**を挿入することにします。ただ利息受け取り金額を調整するだけではありません。その後、支払い用の別の**仕訳記帳**でこれを相殺します。

エキスパート：では、支払い履歴のほかに、利息発生履歴もすべて見られるようになるのですね？ そういう機能が欲しかったのです。

開発者2：私は、自分たちが「**会計取引**（Transaction）」の概念を正しく使えているか心配です。定義によれば、ある**勘定科目**から別の**勘定科目**への資金の移動となっていて、2つの仕訳記帳を同一の**勘定科目**で相殺することとはなっていないのです。

開発者1：よい指摘です。その本では、会計取引がすべて同時に作成されると主張しているように見えますが、それは私も気になっていました。利息の支払いは数日遅れる可能性があります。

エキスパート：そういった支払いは、必ずしも遅れたととらえる必要はありません。支払うタイミングはかなり融通が利くのです。

開発者1：そうだとすると、行き詰まったかもしれませんね。何か暗黙的だった概念を突き止めたかと思っていました。**利息計算サービス**に**仕訳記帳**オブジェクトを生成させる方が、うまく意図が伝わるように思えました。それに、**会計取引**によって、計算した利息と支払いがうまく結びつけられるように思えたのです。

エキスパート：どうして発生を支払いと結びつけなければならないのですか？ 会計システムでは別々に記帳されますよ。**勘定科目**での収支がここでの主題です。個々の**仕訳記帳**と合わせれば、必要なものはもう揃っています。

開発者2：利息支払いが行われたかどうかは追跡しないということですか？

エキスパート：いえ、もちろん追跡します。ただ、あなたがたが言うような、1つの発生に対して1つの支払いがあるという構想ほど、単純ではないのです。

開発者2：その結びつきで悩まなくなれば、実際にはかなり単純化できるでしょう。

開発者1：わかりました。では、こういうのはどうでしょう？ ［元のクラス図をコピーして、修正内容をスケッチし始める］ところで、**発生**（accrual）という言葉を何度か使っていましたね。どういう意味か、明確に教えていただいてもいいですか？

エキスパート：もちろんです。発生というのは、支出や収入について、それが生まれた瞬間に計上するもので、金銭が実際にいつ受け渡されるかは気にしません。したがって、利息は毎日発生しますが、その支払いは（例えば）月末に行われるわけです。

開発者1：なるほど、まさにそういう言葉が必要でした。では、これならどうですか？

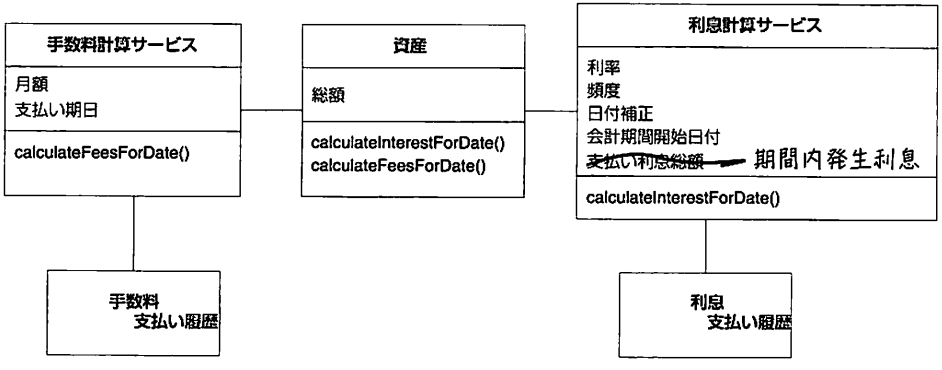


図11.5 元のクラス図で、発生を支払いから切り離したもの

開発者1：これで、計算サービスにあった入り組んだ処理を、すべて関連する支払いから取り除くことができました。また、発生という用語を取り入れたことで、意図がよく伝わるようになります。

エキスパート：それでは、**勘定科目**オブジェクトはなくなるのですか？ **勘定科目**オブジェクトで、発生や支払い、収支などがすべて一緒に見られると思っていたのですが。

開発者1：本当ですか？ そういうことなら、こうすればうまくいくのではないでしょうか。

［別の図をコピーしてスケッチする］

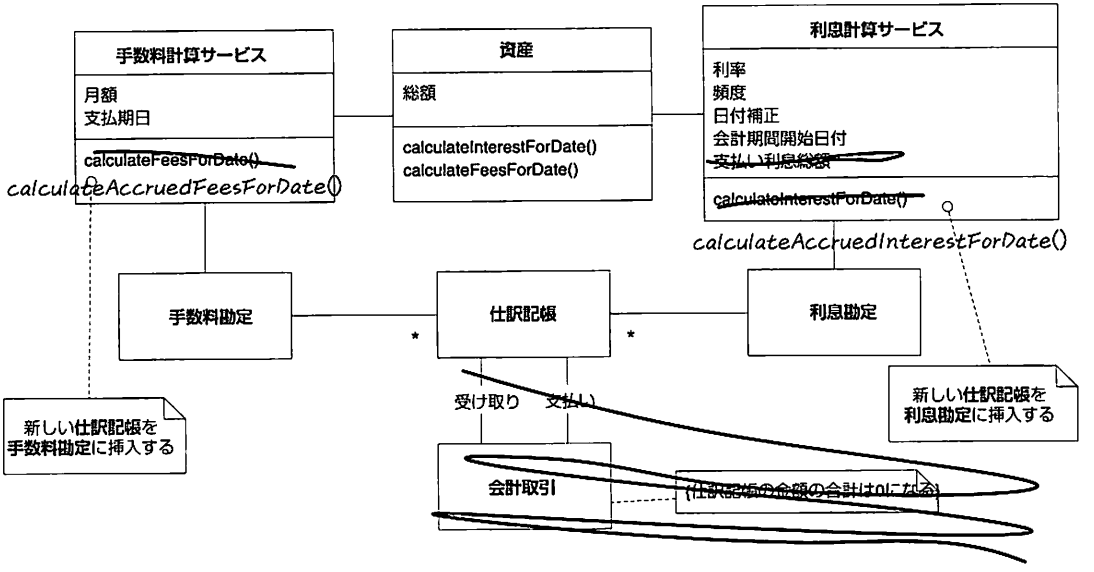


図11.6　勘定科目の考え方に基づく、会計取引を取り除いた図

エキスパート：本当に、とてもよくなりましたね！

開発者2：これらのオブジェクトを使用するよう、バッチスクリプトを変更するのは簡単です。

開発者1：新しい**利息計算サービス**を動くようにするには、数日かかります。変更しなければならないテストがかなりありますので。でも、これが終われば、テストは読みやすくなるでしょう。

開発者2人は作業に戻り、新しいモデルに基づいて、リファクタリングに着手した。コードに手を入れ、設計を厳密にしていく過程で、2人はモデルを改良するための洞察を得た。

仕訳記帳は、**支払い**と**発生**というサブクラスに分けられた。これは、綿密に調べた結果、アプリケーションにおける両者の責務が若干異なっていることが判明したからであり、また、支払いも発生もドメインの重要な概念だからでもある。その一方で、**仕訳記帳**が手数料と利息のどちらに由来しても、それぞれの**仕訳記帳**は概念上もふるまい上も区別されていなかった。適切な**勘定科目**に挿入されるというだけなのだ。

しかし、残念なことに、この最後の抽象化は、実装の都合であきらめなければならなかった。データは関係テーブルに格納されていて、プロジェクトの標準によれば、そうしたテーブルはプログラムを実行しなくても解釈できるようにしなければならなかったのだ。これは、手数料仕訳記帳と利息仕訳記帳を、別々のテーブルに保存することを意味した。2人がこの分割を実現する方法は、特定のO/Rマッピングフレームワークを使っていたこともあって、具体的なサブクラスを作成する以外になかった（**手数料支払い**、**利息支払い**など）。違うインフラストラクチャを使っていれば、この不恰好な肥大化は避けられたかもしれない。

大部分はフィクションであるこの話に、このような展開を導入したのは、我々が常に遭遇することになる現実の障害を表すためである。我々はトレードオフを計算した上で妥協し、モデル駆動設計を投げ出すことなく先へ進まなければならないのだ。

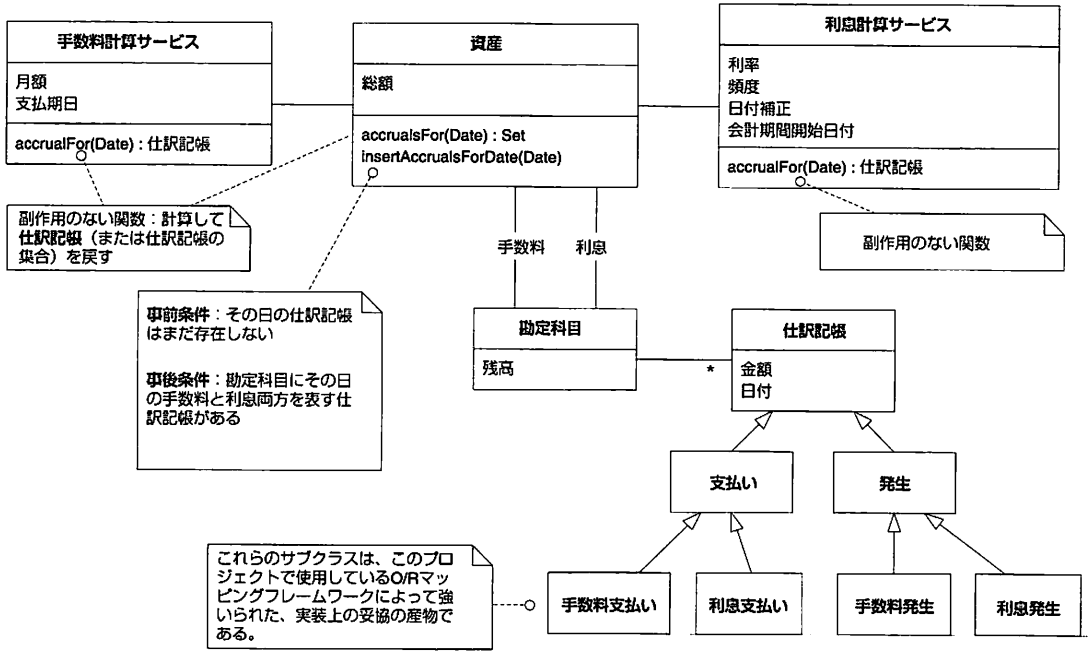


図11.7 実装後のクラス図

この新しい設計は、分析するのもテストするのもはるかに容易になったが、それは、最も複雑な機能が副作用のない関数に実装されているからである。残るコマンドも（さまざまな関数を呼び出しているため）コードは単純で、表明によって特徴づけられている。

プログラムの中には、我々が、ドメインモデルから恩恵を受けられるのではないかと思うことすらしない部分が存在することがある。そういう部分は、始まりが非常に単純で、進化も機械的かもしれない。ドメインロジックというよりも、入り組んだアプリケーションコードのように見える。アナリシスパターンは我々に、このような盲点を示してくれる。

次に続く例では、開発者が夜間バッチのブラックボックスについて新しい洞察を得ているのだが、この夜間バッチは、これまでドメイン指向と見なされていなかったものである。

例 11.1（続き）——夜間バッチについての洞察

数週間後、**勘定科目**の概念に基づく改善されたモデルが定着し始めた。よくあることだが、新しい設計が明確であることで、他の問題が前よりも見えるようになった。夜間バッチを新しい設計と相互作用するように適合させていた開発者（**開発者 2**）は、バッチのふるまいと『アナリシスパターン』の概念の一部につながりがあると気づき始めた。最も関係が深いと彼が思った概念を要約すると、次のようになる。

記帳ルール

会計システムは、基本的な財務情報に関して、1つのものに対する複数の見方を提供することがある。ある勘定科目が収入を追跡するかもしれず、その一方で、別の勘定科目はその収入に対して見積もられる税金を追跡するかもしれない。このシステムが、見積もられる税金の勘定科目を自動的に更新するようになっていれば、これら2つの勘定科目の実装は非常に絡み合ったものになる。勘定科目の仕訳記帳の大多数がそういうルールに起因しているシステムがある。そうしたシステムでは、依存関係のロジックが混乱したものになっていく。そこまで極端ではないシステムでも、こうした複式の記帳（cross-posting）は油断ならないものになり得る。依存関係のもつれを制御するための最初のステップは、新しいオブジェクトを導入することによって、こうしたルールを明示的にすることである。

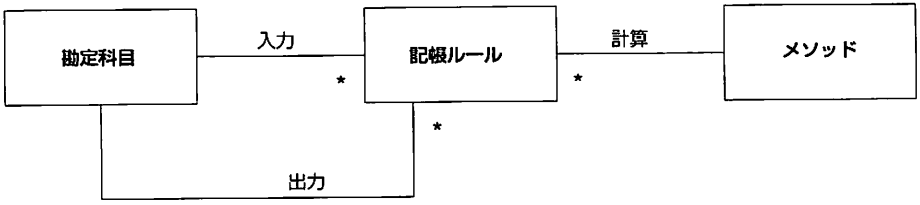


図 11.8　基本的な記帳ルールのクラス図

ある記帳ルールでは、「入力」勘定に挿入された新しい**仕訳記帳**がきっかけになる。その後、新しい**仕訳記帳**を（独自の計算**メソッド**に基づいて）派生させ、その新しい**仕訳記帳**を「出力」勘定に挿入する。給与システムであれば、**給料勘定**の**仕訳記帳**がトリガ

となって**記帳ルール**（**Posting Rule**）が起動し、30%の見積もり所得税を計算して、それを源泉徴収**勘定**に**仕訳記帳**として挿入することになる。

記帳ルールを実行する

この**記帳ルール**は**勘定科目**間の概念的な依存関係を規定しているが、パターンがそこで止まってしまうと、従うのは困難になるだろう。依存関係の設計において、最も注意を要する部分の1つは、更新のタイミングと制御である。Fowlerは、次の3つの選択肢について議論している。

1. 「都度の実行（eager firing）」が最も理解しやすいが、たいていは最も実用的でない。**仕訳記帳**が**勘定科目**に挿入されるたびに、即座に**記帳ルール**を起動し、すべての更新を直ちに行う。
2. 「**勘定科目**ベースの実行」によって、処理は延期できるようになる。ある時点で、メッセージが**勘定科目**に送信され、それによって**記帳ルール**は、最後に実行された後に挿入された**仕訳記帳**を、すべて処理する。
3. 最後に、「**記帳ルールベース**の実行」は、**記帳ルール**に実行を指示する外部のエージェントによって開始される。**記帳ルール**は、最後に実行された後に入力**勘定**に対して作成された**仕訳記帳**をすべて見つける責務を負う。

実行モードをシステム内で混在させることもできるが、特定のルールの集合は、それぞれ明確に定義された開始ポイントを1つと入力**勘定仕訳記帳**を識別する責務を持つ必要がある。3つの実行モードをユビキタス言語に追加することは、このパターンを成功させる上で、モデルオブジェクトの定義それ自体と同じくらい重要だ。それによってあいまいさがなくなり、明確に定義された選択肢の中から意思決定できるようになる。これらのモードがあることで、そうでなければ容易に見過ごされてしまう課題が識別され、明確な議論を支える語彙がもたらされる。

開発者2は、自分の新しい考えを議論するために話し相手が必要だった。そこで彼は同僚（**開発者1**）と会うことにした。彼女は発生のモデリングを主に担当していたのである。

開発者2：どこかの時点で、夜間バッチは問題を隠す場所になりだしたようです。スクリプトの処理の中に暗黙的なドメインロジックがあって、それがどんどん複雑化しています。私はずっと前から、このバッチにモデル駆動設計を適用して、ドメイン層を分離して取り出し、スクリプト自体はドメイン層に被せられるシンプルなレイヤにしたいと思っていました。でも、ドメインモデルがどういう風になるか、見当がつきませんでした。オブジェクトとしては意味を持たない、ただのプロシージャでしかないように思えていたのです。「アナリシスパターン」の**記帳ルール**に関する節を読んでいて、ある考えが浮かびました。こういうことを考えたのです。[スケッチを手渡す]

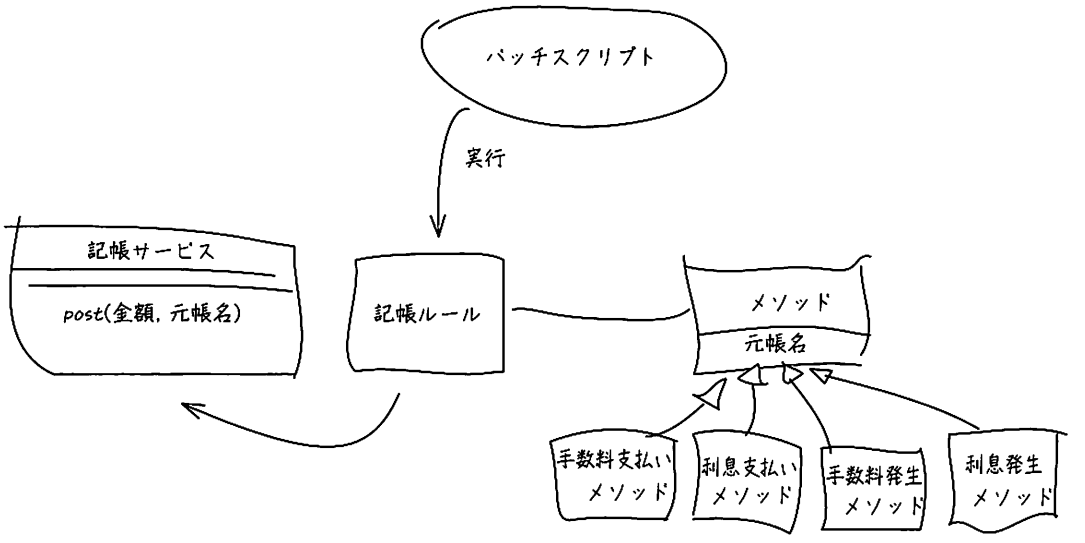


図11.9　バッチで記帳ルールを使用する試み

開発者1：この「**記帳サービス**」とは何ですか？

開発者2：それはファサードで、会計アプリケーションのAPIを使えるようにし、それをサービスとして提供するものです。実は、少し前に、バッチコードを単純化するために私が作成したものですが、レガシーシステムへ記帳するための意図の明白なインタフェースも、それによってできました。

開発者1：おもしろいですね。それで、どの実行スタイルをこの**記帳ルール**に使うつもりですか？

開発者2：実はまだ、そこまでは考えていません。

開発者1：**発生**には都度の実行が使えるでしょう。実際、バッチが**資産**に**発生**の挿入を指示するのですから。でも、日中に入力される**支払い**には使えませんね。

開発者2：計算メソッドは、そこまで緊密にバッチと結合させるべきではないと思います。利息計算を別の時に起動させたら、混乱してしまうでしょう。それに、概念的にも正しくない気がします。

開発者1：それでは、**記帳ルール**ベースの実行のようですね。バッチが各**記帳ルール**に実行を指示して、ルールが適切な新規**仕訳記帳**を探し、その内容を処理します。今書いた図とほぼ同じですね。

開発者2：それなら、バッチの設計で大量の依存関係を作成しないで済みますし、バッチの方で制御し続けることもできます。よさそうですね。

開発者1：ただ、これらのオブジェクトと、**勘定科目**や**仕訳記帳**との相互作用がまだはっきり

しません。

開発者2：同感です。あの本の中の例では、**勘定科目**と**記帳ルール**との間に直接リンクを作成しています。論理的ではありますが、ここでうまく機能するとは思えません。これらのオブジェクトは、毎回データからインスタンス化しなければならないので、関連づけるにはどのルールが適用されるかを理解する必要があるでしょう。その一方で、**資産**オブジェクトは各**勘定科目**の内容を知っており、したがってどのルールを適用するべきかも知っています。さて、他の場所はどうですか？

開発者1：あら探しをするつもりはありませんが、「**メソッド**」を正しく使えていないと思います。概念としては、**メソッド**が記帳する金額を計算するということだと思います。例えば、収入に対して20%の源泉徴収を記帳するということです。でも、私たちのケースでは、これは単純で、記帳されるのは常に総額になっています。**記帳ルール**自体がどの**勘定科目**に記帳すべきかということ、つまり我々にとって「元帳名」に対応するものを知っているべきではないでしょうか。

開発者2：なるほど。**記帳ルール**が正しい元帳名を知っているという責務を負うなら、たぶん**メソッド**は一切必要なくなるでしょう。実を言うと、正しい元帳名を選択するというこの業務全体が、ますます複雑になってきています。すでに、収入のタイプ（手数料か利息）と「資産クラス」（このビジネスが各**資産**に適用するカテゴリ）との組み合わせになってしまっています。ここは、この新しいモデルで改善できたらよいとは思っている部分です。

開発者1：わかりました。そこに集中しましょう。**記帳ルール**には、**勘定科目**の属性に基づいて元帳を選択する責務があります。今のところは、資産クラスと、利息と手数料との間の区別を処理するのに、素直な方法を取れます。将来的にはオブジェクトモデルができて、より複雑なケースを処理できるように強化されるでしょう。

開発者2：もう少し考える必要がありますね。じっくり考えて、パターンを読み返してから、もう一度試してみます。この件で、明日の午後にもう一度話し合えますか？

それから数日をかけて、2人の開発者はあるモデルを考え出し、コードをリファクタリングした。それによって、バッチは各**資産**に対してイテレーションを行うだけとなった。その際、それぞれの資産に意図の表現されたメッセージを送り、その後でデータベースのトランザクションをコミットする。複雑さは、ドメイン層へと移り、オブジェクトモデルのおかげで、より明示的で抽象的になった。

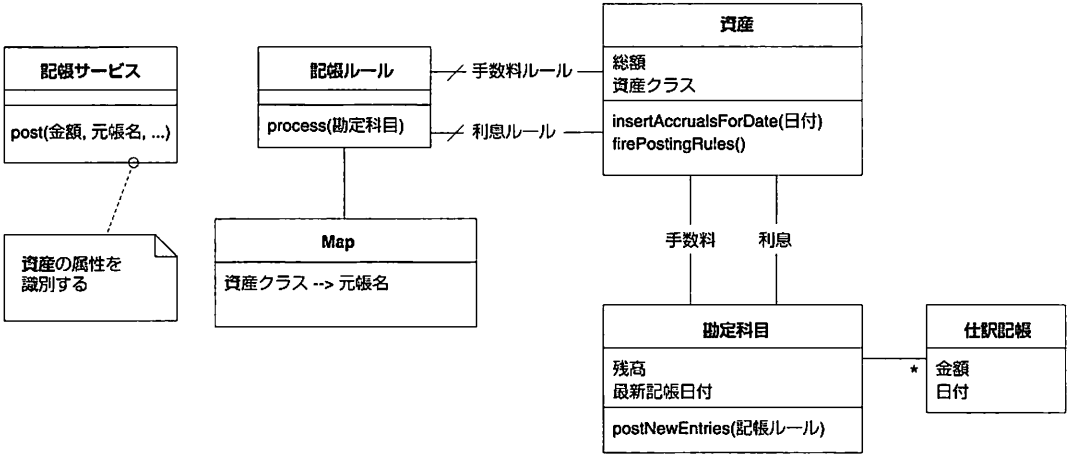


図11.10　記帳ルールのあるクラス図

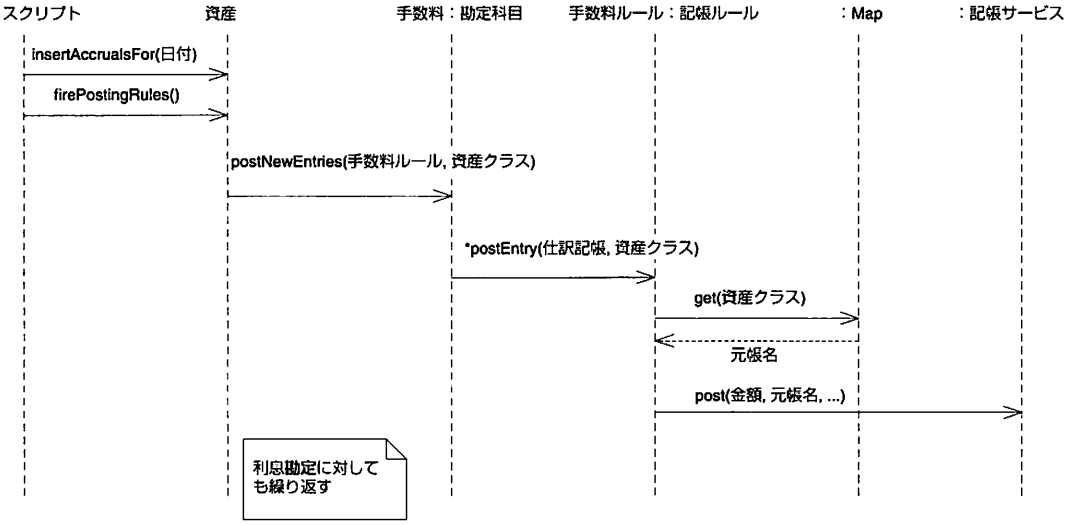


図11.11　ルールの実行を示すシーケンス図

この2人の開発者は、『アナリシスパターン』で説明されているモデルの詳細からはかなり離れたが、それでも概念の本質は維持していると感じた。2人が少し気に入らなかったのは、**記帳ルール**を選択する際に**資産**を巻き込んでいることであった。そのようにした理由は、**資産**が各**勘定科目**（手数料または利息）の性質に関する知識を持っていただけでなく、スクリプトにとって自然なアクセスポイントでもあったからだ。もし、ルールオブジェクトを**勘定科目**と直接関連させていたら、オブジェクトをインスタンス化するたびに（毎回バッチが実行されるたびに）、**資産**オブジェクトと協調しなければならなかったはずだ。その代わりに、**資産**オブジェクトがシングルトン（SINGLETON）アクセスを通じて2つの関連するルールを探し、それを適切な**勘定科目**に渡せるようにした。そうすることで、コードがはるかに直接的になると思われたので、2人は実践的に決定したのである。

2人とも、**記帳ルール**を**勘定科目**とだけ関連づけて、**資産**の方は**発生**を生み出す作業に専念させ続ける方が、概念的にはよいと思っていた。2人は、今後のリファクタリングとより深い洞察によって自分たちがこの問題に立ち返り、そこでコードを明瞭にしたまま明確に分割する方法が示されることを期待したのだ。

アナリシスパターンは活用すべき知識である

運よくアナリシスパターンがあっても、それが具体的な要求に応えてくれることはほとんどない。だが、アナリシスパターンは、調査に対して有益な手がかりを与え、明確に抽象化された語彙を提供してくれる。また、実装結果に関する指針も示すはずだから、後になって痛い思いをしないで済む。

こうしたことすべては、知識のかみ砕きと深い洞察へ向かうリファクタリングに活力を与え、開発を活性化させる。その結果は、アナリシスパターンでドキュメント化された形式に似ていることも多いが、それぞれの状況に適合している。時には結果が、アナリシスパターン自体とは明白に関係しないこともあるが、それでも、パターンから得た洞察に刺激されたものではあるのだ。

避けるべき変更が1種類ある。よく知られたアナリシスパターンの用語を使用する場合には、そのパターンによって規定される基本的な概念を、注意深く元のまま保つこと。表面的な形式がどれほど変わっても、そうしなければならないのだ。その理由は2つある。第一に、そのパターンは、将来問題を避ける上で助けになるような理解を組み込んでいるかもしれない。第二に、こちらの方が重要なのだが、ユビキタス言語に含まれている用語が広く理解されているか、少なくともうまく説明されているのであれば、そのユビキタス言語は強化されるのである。モデルの自然な進化を通じてモデルの定義が変化するなら、その名前も変更する手間を惜しまないこと。

かなりの数のオブジェクトモデルが文献で取り上げられており、1つの業界の1種類のア

プリケーションに特化したものもあれば、非常に一般的なものもある。そうしたモデルのほとんどが考え方の素は提供しているが、その選択とそれに続く結果の背景にある理論をとらえているものはわずかしかない。だがこうした理論こそが、アナリシスパターンにおいて最も役に立つ部分なのだ。こうした精緻なアナリシスパターンがもっとあれば有益だろう。車輪の再発明を繰り返さなくて済むからだ。包括的なカタログを目にすることがあったら私は驚くだろうが、業界に特化したカタログなら出てくるかもしれない。多くのアプリケーションを横断するいくつかのドメインのパターンなら、広く共有できるだろう。

体系化された知識のこの種の再適用は、フレームワークやコンポーネントを通じてコードを再利用しようとする試みとは全く異なる。だが、どちらも明白ではない考え方の素を提供できる。モデルは、それが汎用的なフレームワークの場合でも、動作する完結した全体であるのに対し、分析は、モデルの断片を一式取り揃えたものである。アナリシスパターンは、最も重要で困難な意思決定に焦点を合わせ、代替案や選択肢を浮かび上がらせる。アナリシスパターンが予想してくれる、後に続く結果は、自力で発見しなければならないとしたら高くつくものなのだ。

第12章

デザインパターンをモデルに関係づける

これまで本書で探究してきたパターンは、モデル駆動設計のコンテキストにおいて、ドメインモデルの問題を解決することを意図していた。だが、実際のところ、今までに公開されているパターンのほとんどは、より技術的な側面に焦点を合わせている。デザインパターンとドメインパターンの違いは何だろうか？ 第一に、よく知られている書籍、『デザインパターン』の著者たちは次のように述べている。

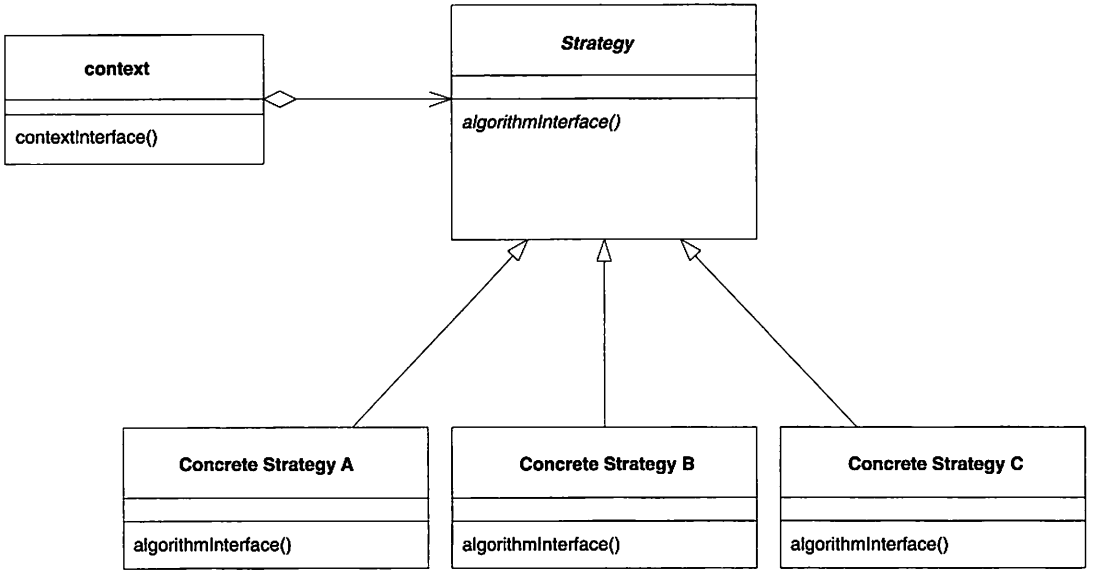
> 何がパターンで、何がそうでないかという解釈は、視点の影響を受ける。ある人にとってはパターンであるものも、別の人にとってはプリミティブな構成要素かもしれない。この本では、我々はあるレベルまで抽象化されたパターンに専念した。デザインパターンは、連結リストやハッシュテーブルのように、クラスとしてコード化され、そのままで再利用できる設計を扱うものではない。また、アプリケーションやサブシステム全体に関する、複雑なドメインに特化した設計でもない。本書でいうデザインパターンは、オブジェクトやクラスとコミュニケーションすることに関する記述であり、一般的な設計の問題を特定のコンテキストで解決するためにカスタマイズされているのだ。(Gamma他 1995, p.3)

『デザインパターン』に書かれたパターンは、全部ではないにしても、一部はドメインパターンとして使用できる。ただ、そのためには強調する点を変えなければならない。『デザインパターン』が提供している設計要素のカタログは、さまざまなコンテキストで頻繁に直面する問題を解決してきたものだ。それらのパターンの動機やパターンそのものは、純粋に技術的な用語で紹介されている。しかし、こうした要素のサブセットは、ドメインモデリングや設計といった、より広範なコンテキストでも適用できる。こうした要素は、多くのドメインにおいて出現する一般的概念と一致するからである。

『デザインパターン』で紹介されているパターンに加え、他にも長年にわたって紹介されてきた、多くの技術的なデザインパターンがある。それらの中には、ドメインに現れる深い概念と一致するものもある。これを活用できれば素晴しい。ドメイン駆動設計でそういうパターンを利用するためには、パターンを同時に2つのレベルで考えなければならない。すなわち、一方に、コードで用いられる技術的なデザインパターンがあり、もう一方に、モデルで用いられる概念的なパターンがあるのだ。

『デザインパターン』に登場する具体的なパターンのサンプルを見ることで、デザインパターンであると考えられるパターンが、ドメインモデルに対してどう適用できるかがわかり、技術的なデザインパターンとドメインパターンの区別が明確になる。コンポジットとストラテジーを見ることで、古典的なデザインパターンのいくつかが、考え方を変えることによってどのようにドメインの問題に適用できるかが示される…。

ストラテジー（STRATEGY）（別名　ポリシー（POLICY））



> アルゴリズムのファミリーを定義し、それぞれをカプセル化して、相互に交換可能にする。ストラテジーによって、アルゴリズムは、それを使用するクライアントから独立して変化できるようになる。（Gamma 他 1995）

ドメインモデルに含まれているプロセスの中には、技術的に動機づけられたものだけでなく、問題となっているドメインで、実際に意味を持つものもある。あるプロセスに対して、その代替となるものを提供しなければならないとすると、適切なプロセスを選択することに伴う複雑さは、複数のプロセス自体にある複雑さと相まって、手に負えなくなる。

プロセスをモデル化していると、そのための正当なやり方が複数あることに気づくことも多い。この選択肢を記述し始めると、そのプロセスの定義は不格好で入り組んだものとなる。実際のふるまいの選択肢で、複数ある中から我々がどれかを選ぶようなものは、他のふるまいと混ぜ合わされるとあいまいになってしまう。

こうしたふるまいのバリエーションを、プロセスにとっての主要な概念から分離したい。そうすれば、主要なプロセスと選択肢の両方を、より明確に見ることができるようになるだろう。ストラテジーパターンは、すでにソフトウェア設計のコミュニティで確立されており、まさにこの問題に取り組むものだが、その焦点は技術面にある。だが、ここでは、

ストラテジーパターンがモデル内の概念として適用され、モデルを実装したコードに反映される。同じように、プロセスの中にある非常に変わりやすい部分を、より安定した部分から切り離すという要求もある。

それゆえ：

プロセスの中で変化する部分を、モデルの中で独立した「ストラテジー」オブジェクトに括り出すこと。ルールと、ルールが支配するふるまいは、切り離して括り出すこと。そのようなルールや置き換え可能なプロセスは、ストラテジーデザインパターンに従って実装すること。ストラテジーオブジェクトの複数のバージョンは、プロセスを実行する上でのさまざまな方法を表している。

デザインパターンとしてのストラテジーに対する従来の見方は、さまざまなアルゴリズムを置き換える能力に焦点を合わせているが、ドメインパターンとして使う場合には、プロセスや、ポリシーのルールに代表されるような概念を表現する能力に焦点を合わせている。

例 12.1 ——経路検索ポリシー

ある**経路仕様**は**経路選択サービス**に渡され、そこで仕様を満たす詳細な**輸送日程**が構築される。このサービスは一種の最適化エンジンで、最速の経路か最安の経路のいずれかを検索するように調整できる。

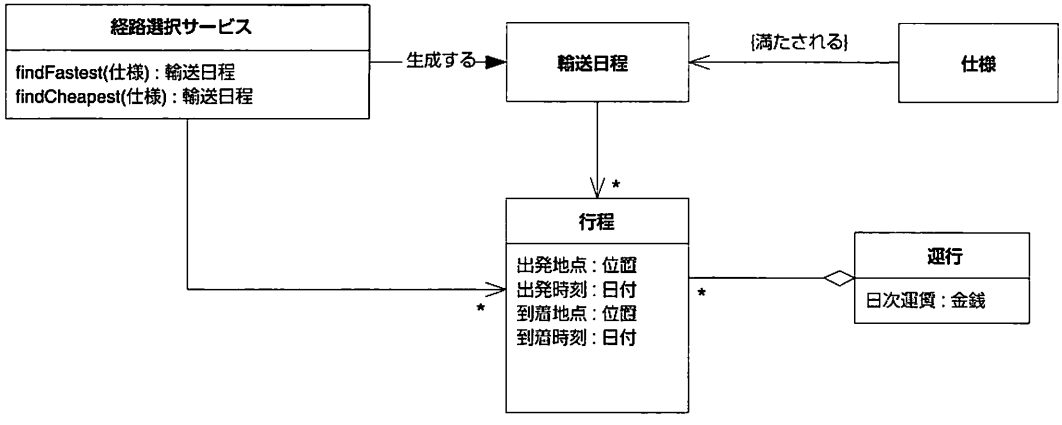


図12.1 選択肢のある**サービス**インタフェースには、条件つきロジックが必要になる。

この設定はよさそうに見えるが、経路選択を行うコードを詳細に調べると、すべての処理に条件文があり、最速か最安かの決定をいたるところで行っている。経路をよりきめ細かく選択

するための新しい基準が追加されれば、さらに問題が生じることになる。

これを解決するアプローチとして、選択基準を調整するためのパラメータをストラテジーに分離する、というものがある。そうすることで、基準が明示的に表現され、パラメータとして**経路選択サービス**に渡すこともできるようになる。

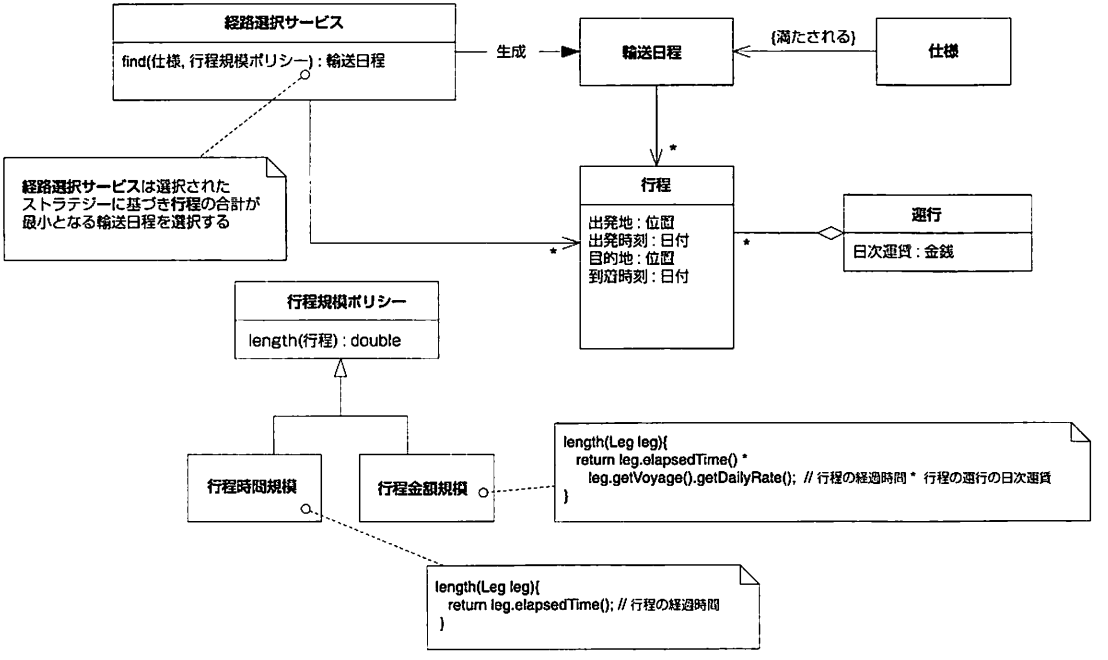


図12.2　引数として渡される**ストラテジー（ポリシー）**の選択によって定められる選択肢

この**経路選択サービス**は、すべての要求を同一の条件句を使わない方法で処理し、**行程**の連なりの中から、**行程規模ポリシー**によって計算される合計が最も少ないものを探す。

この設計が持つ利点こそが、『デザインパターン』におけるストラテジーパターンの動機となっている。アプリケーションの用途の広さと柔軟性というレベルで見れば、**経路選択サービス**のふるまいは、今や、適切な**行程規模ポリシー**を導入することで制御し、拡張できるようになっている。図12.2に示したストラテジー（最速または最安）は最もわかりやすいものにすぎない。実際には、スピードとコストのバランスを取った組み合わせになるだろう。他の要因として、貨物を予約する際に自社での輸送の方を好み、他の輸送業者へ下請けには出さないようにする傾向なども、一緒になっているかもしれない。こういう修正は、ストラテジーに頼らなくても可能だっただろうが、その場合、ロジックは**経路選択サービス**の内部を曲がりくねって進み、インタフェースを肥大させただろう。疎結合にすることで、インタフェースは明確になり、また、容易にテストできるようになるのである。

●例 12.1 ——経路検索ポリシー

ドメインにおける本質的に重要なルールは、**輸送日程**を構築する上で、ある**行程**を選択する際の基準であるが、これは今や、明示されて他とは区別されている。このルールは、行程それぞれが持つ特定の属性（潜在的に導き出されたもの）を、ある数字にまで煮詰めたものが経路選択の基準だという知識を伝えている。こうすることで、そのドメインの言語で書かれたシンプルな文により、**経路選択サービス**のふるまいが定義できるようになる。すなわち、この**経路選択サービス**は、選択されたストラテジーに基づいて、**行程**の規模の合計が最小となる**輸送日程**を選択するのだ。

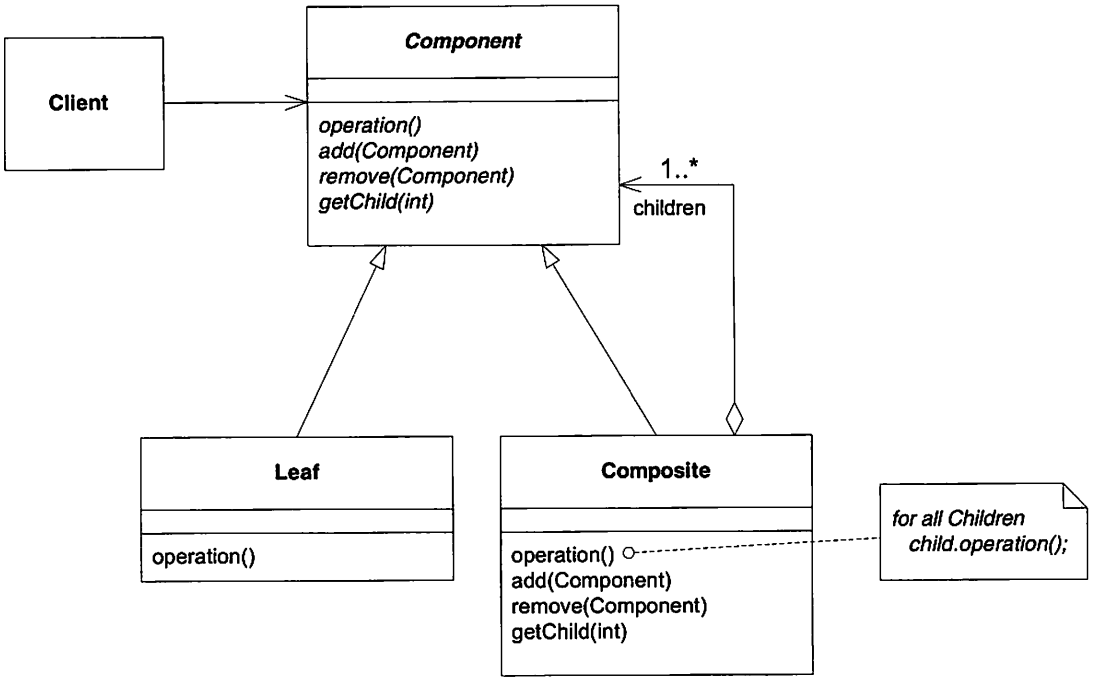
注：この議論では、**経路選択サービス**が**輸送日程**を検索する際に、実際には**行程**を評価していることが示唆されている。このアプローチは、概念的にはわかりやすく、それなりのプロトタイプ実装なら作れるかもしれないが、おそらく容認できないくらい非効率的だろう。このアプリケーションについては、第14章「モデルの整合性を維持する」で再度取り上げるが、そこでは同じインタフェースが、全く異なって実装された**経路選択サービス**によって使用される。

* * *

ドメイン層で技術的なデザインパターンを使用する場合には、別の動機を追加しなければならない。つまり、意味のレイヤをもう1つ追加しなければならないのだ。ストラテジーが現実のビジネス戦略や方針と一致すれば、このパターンは便利な実装テクニック（それだけでも十分価値はあるが）以上のものになる。

このデザインパターンがもたらす**帰結**は、ここにも完全に当てはまる。例えば、『デザインパターン』でGammaらが指摘しているのは、クライアントがさまざまなストラテジーを意識していなければならないということだ。だが、これもまた、モデリングの懸念なのだ。純粋な実装上の懸念は、ストラテジーを使うとアプリケーションにおけるオブジェクトの数が増えるかもしれない、ということである。これが問題になるのであれば、コンテキスト内で共有できるステートレスなオブジェクトとしてストラテジーを実装することにより、オーバーヘッドを減らすことができる。また、『デザインパターン』で紹介されている、実装アプローチに関する広範な議論は、すべてここにも適用される。その理由は、我々がやはりストラテジーを使用しているということにある。我々の動機は一部異なっていて、それがパターンの選択に際して影響することもあるが、デザインパターンに組み込まれている経験は、我々の意図に合わせて使用できるのだ。

コンポジット（COMPOSITE）



> オブジェクトを組み立てて、部分と全体の階層を表すツリー構造を作ること。コンポジットを使うことで、クライアントは、個々のオブジェクトとオブジェクトの複合体を一様に扱えるようになる。(Gamma他 1995)

複雑なドメインをモデリングしていると、次のような構造に遭遇することがある。すなわち、ある重要なオブジェクトが複数の部品から構成され、その部品も複数の部品から構成され、さらにその部品も複数の部品から構成されているというもので、こうしたネストは任意の深さまで続くこともある。ドメインによっては、これらのレベルがそれぞれ概念的に区別されていることもあるが、各部品が、集まって全体を構成した場合と、種類が同じで、ただ小さくなっているだけと感じられることもある。

ネストされたコンテナの関係性がモデルに反映されていないと、共通するふるまいを階層の各レベルで繰り返さなければならず、さらにネストに柔軟性がなくなる（例えば、コンテナは通常、自分のいるレベルにある他のコンテナを含むことができず、また、レベルの数は固定されている）。クライアントは、階層のさまざまなレベルを扱うにあたって別々

のインタフェースを使わなければならない。こうした使い分けは、概念的な違いをクライアントが意識していなくても必要である。階層を再帰しながら集約された情報を作り出すのは、非常に入り組んだ処理である。

どんなデザインパターンでも、ドメインに適用する際にまず気にすべきことは、パターンの考え方が本当にドメインの概念に合うかどうかである。いくつかのオブジェクトの関連を辿る際に、再帰的に行った方が便利かもしれないが、そこにあるのは、本当に、全体と部分の階層だろうか？ すべての部分が、概念的に間違いなく同じタイプになるような抽象化が見つかっただろうか？ 見つかったのであれば、コンポジットを使うことで、モデルのそうした側面をより明確にしながら、デザインパターンにおいて慎重に考え抜かれた、設計と実装の考慮事項を活用できるようになる。

それゆえ：

コンポジットを構成するすべてのメンバを包括的に含む抽象型を定義すること。情報を戻すメソッドはコンテナに実装され、内容についての集約された情報を戻す。「リーフ(Leaf)」ノードは、それぞれ独自の値に基づいて、それらのメソッドを実装する。クライアントは抽象型を扱えばよく、リーフとコンテナを区別する必要はない。

コンポジットは構造のレベルに関する比較的わかりやすいパターンだが、設計者が自らを駆り立てて、パターンにおける操作のレベルを具体化することはあまりない。コンポジットでは、構造の各レベルで同じふるまいが提供され、意味のある質問は、小さい部品に対しても大きい部品に対しても行える。それらの部品は自分を組成しているものを透過的に反映しているからだ。この厳密な対称性が、コンポジットの持つ力の鍵である。

例 12.2——経路で構成された輸送経路

完全な貨物輸送経路は、入り組んだものである。まず、コンテナを終着駅までトラック輸送してから港へ運び、次に船で別の港か別の船まで輸送して、最終的には目的地へ陸路で輸送しなければならない。

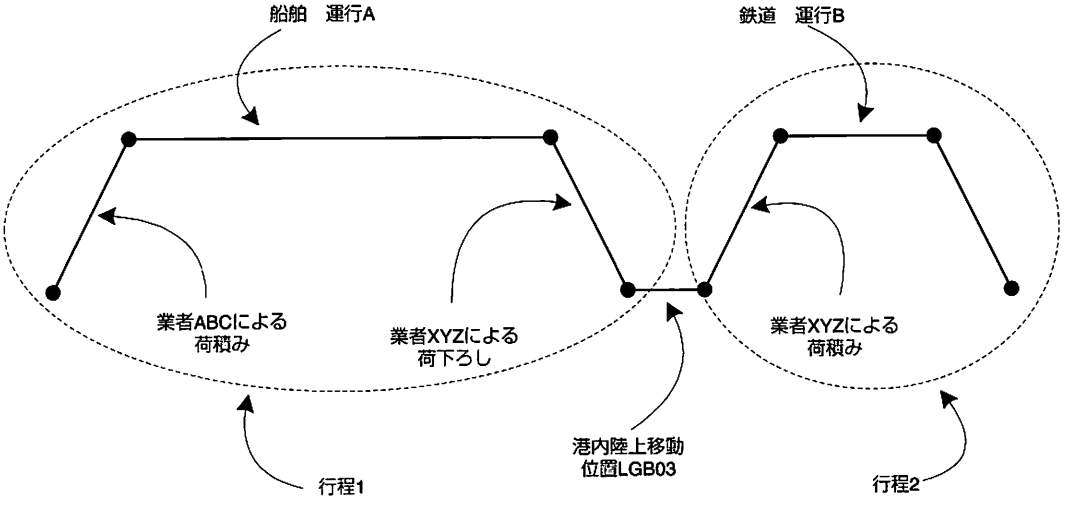


図 12.3　「行程」で構成された「経路」の概略図

アプリケーション開発チームは、経路を構成する、任意の長さで連続した行程を表現するためのオブジェクトモデルを作成した。

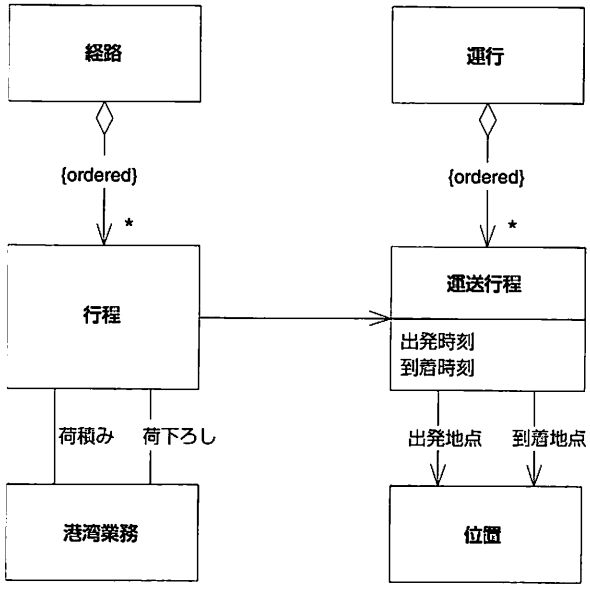


図12.4 行程で構成された経路のクラス図

このモデルを使用すると、開発者は予約要求に基づいて経路オブジェクトを生成できる。行程を加工して、順次行われる荷役のための業務計画を作れる。そこで彼らは、あることを発見した。

開発者はいつも、経路のことを、行程が任意の長さで一様に連続したものと見なしていたのだった。

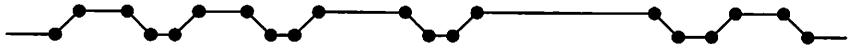


図12.5 経路に関する開発者の概念

ここで明らかになったのは、ドメインエキスパートがこの経路のことを、論理的なセグメントが5個連続したものと見なしているということだ。

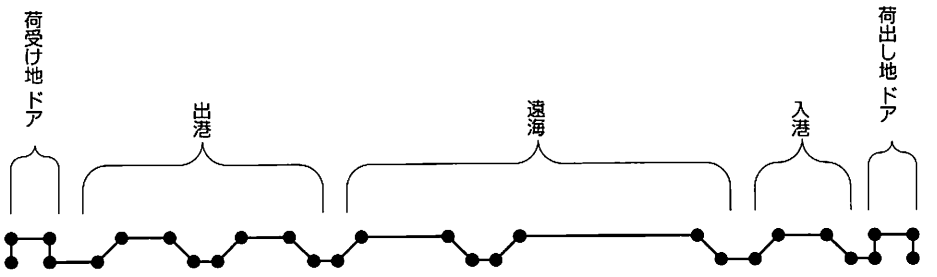


図12.6　ビジネスエキスパートが持っていた経路の概念

何よりも、これらのサブ経路は、別々のタイミングで別々の人によって計画されるかもしれないので、それぞれ別個のものと見なすべきだ。さらに、より細かく見ると、「ドア行程」（door leg）は他の行程と全く異なり、現地で借りたトラックや顧客による運搬さえも伴っていて、念入りにスケジュールされた、鉄道と船舶による運送とは対照的である。

こうした区別をすべて反映したオブジェクトモデルは、入り組んだものになり始める。

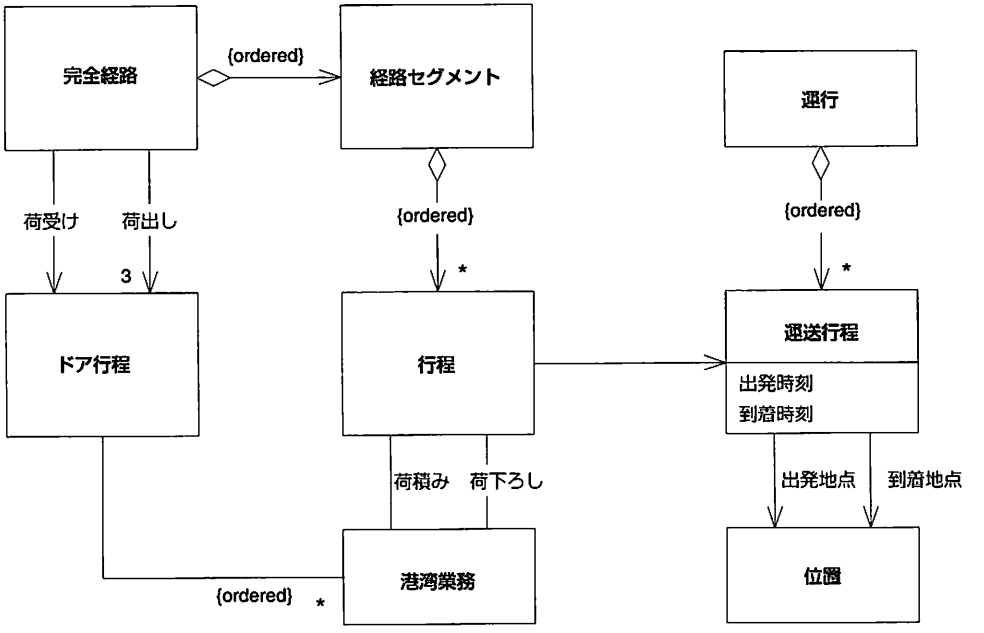


図12.7　経路を詳述したクラス図

構造的にはこのモデルもそれほど悪くないが、業務計画の実施に統一性がなくなるため、コードや、さらにはふるまいの記述も、はるかに入り組んだものになってしまう。それ以外に

も、入り組んだ問題が表面化し始めている。どんな経路を辿る場合にも、さまざまな型のオブジェクトで構成される、複数のコレクションを伴うことになるのだ。

ここで、コンポジットの世界に入ろう。クライアントによっては、経路の構成に見られるさまざまなレベルを一様に扱うことを、望ましいと思うかもしれない。それはある経路が、複数の他の経路で構成されているからである。概念的には、この考え方は適切である。経路の各レベルは、コンテナをある場所から他の場所へ移動させることを表していて、分解していけば個々の行程に行き着く（図12.8参照）。

さて、静的なクラス図は、ドア行程と他のセグメントがどのように合致するのかについて、前掲の図ほどには語らない。しかし、モデルは静的なクラス図だけで表現できるものではない。組み合わせに関する情報を伝えるには、別の図（図12.9参照）と（はるかにシンプルになった）コードを用いる。このモデルは、多様な「経路」すべてに関する深い関係性をとらえている。業務計画を作成するのは再び単純になった。他の経路を辿る操作も同様である。

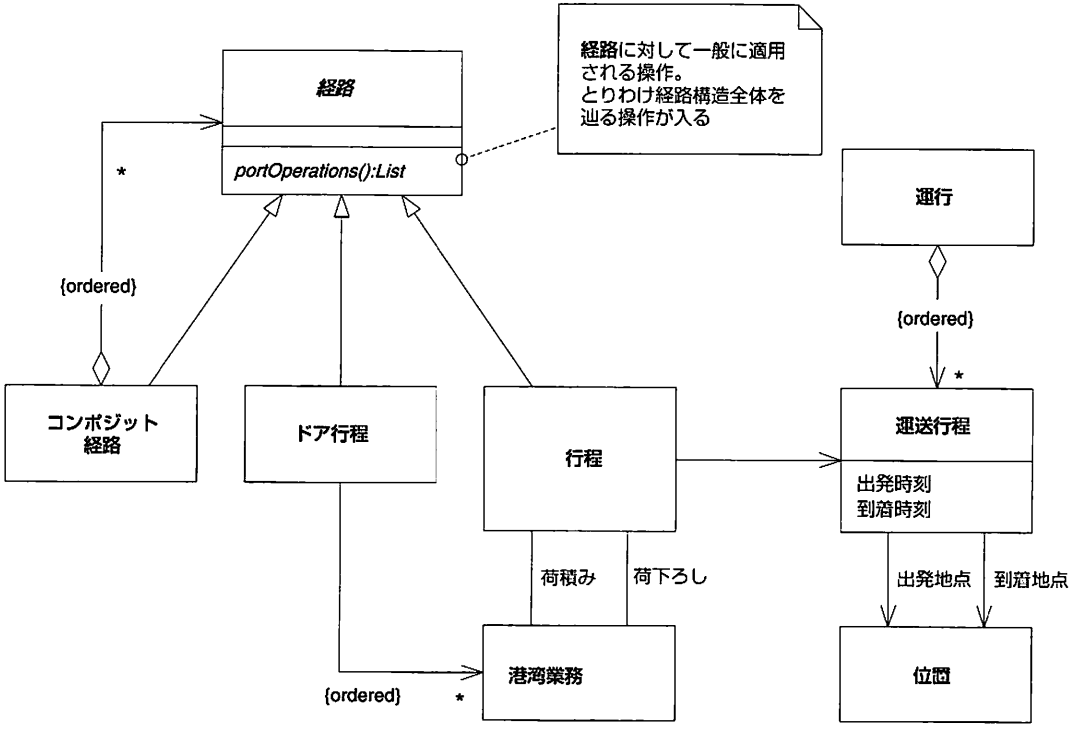


図12.8 **コンポジット**を使用したクラス図

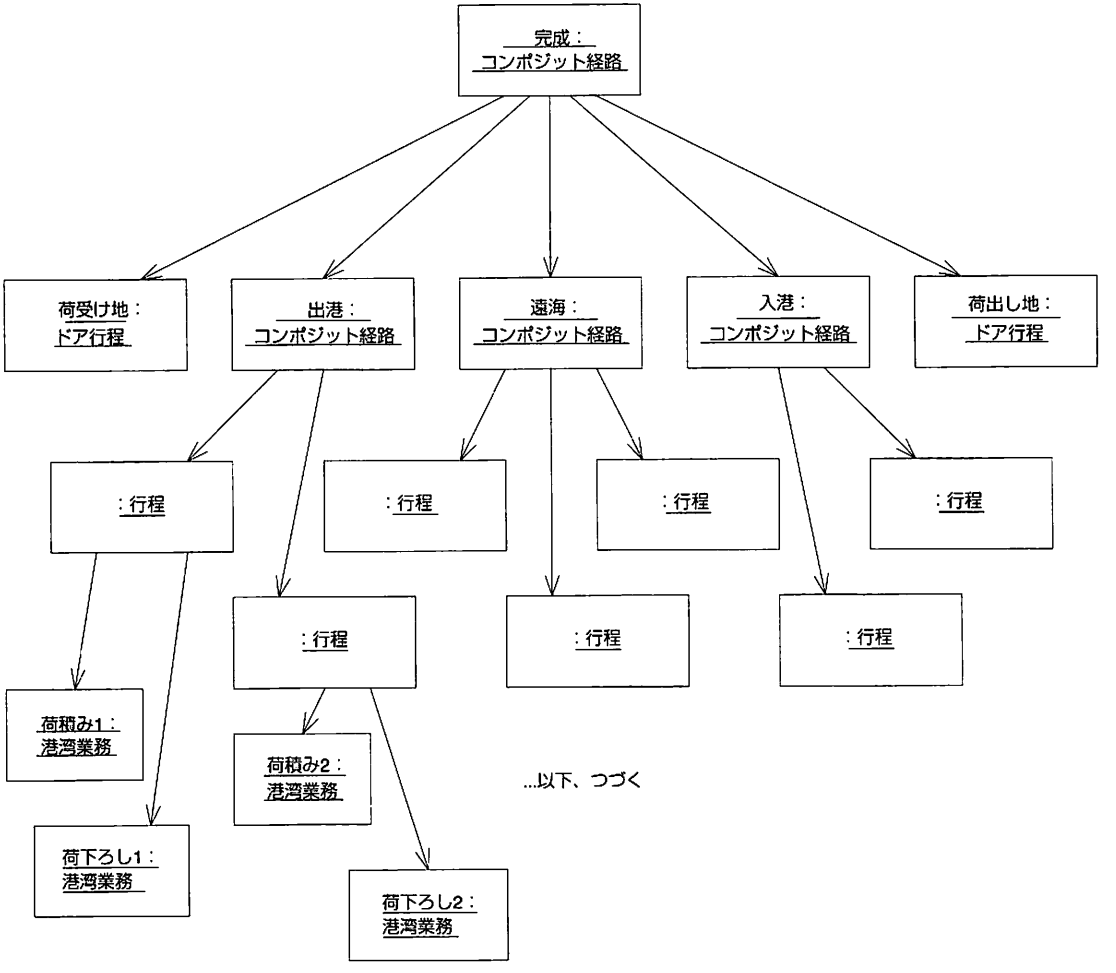


図12.9　完成した経路を表すインスタンス

ある経路を構成するにあたって複数の他の経路を使い、ある場所から他の場所に行くために経路の端同士をつなぐことで、詳細が変化する経路を実装できる。ある経路の終端を切り落として、新しい終端につなぎ合わせることができるし、詳細を任意の深さでネストすることもできる。その上、役に立つと思われるあらゆる選択肢を考え出せる。

もちろん、そういう選択肢はまだ必要ない。それに、経路セグメントやドア行程を区別する必要が生じる前は、コンポジットがなくても支障はなかった。デザインパターンは、必要な場合に限って適用すべきである。

＊　＊　＊

●例 12.2——経路で構成された輸送経路

なぜ、フライウェイトではないのか？

すでに（第5章で）フライウェイトパターンに言及したので、あなたは、それがドメインモデルに適用すべきパターンの例だと考えたかもしれない。実は、フライウェイトは、ドメインモデルにおいて全く**一致するものがない**デザインパターンの好例なのである。

値オブジェクトの限られた集合が何度も使用されている場合には（家の見取り図における電気コンセントの例に見られたのと同様）、それをフライウェイトとして実装することには意味があるかもしれない。だがこれは、値オブジェクトで使用できる実装の選択肢で、エンティティでは使用できない。これをコンポジットと比べて欲しい。コンポジットでは、概念オブジェクトが他の概念オブジェクトから構成されている。その場合、パターンはモデルと実装の両方に適用されているのであり、これこそがドメインパターンの本質的な特徴なのだ。

私は、ドメインパターンとして使用できるデザインパターンの一覧を作成しようとしているわけではない。インタプリタ（interpreter）をドメインパターンとして使用する例は考えられなかったが、これが適合する概念がどのドメインにもないとは、まだ言い切れない。唯一の要件は、パターンが概念的なドメインについて何かを述べるべきであり、技術的な問題に対する、技術的な解決策に留まってはならないということだ。

第13章

より深い洞察へ向かうリファクタリング

より深い洞察へ向かうリファクタリングは、多くの側面を持つプロセスである。いったん立ち止まって、主要なポイントをまとめておくと役に立つだろう。重点的に取り組むべきことが3つある。

1. ドメインに馴染む。
2. 常に、物事に対して違う見方をする。
3. ドメインエキスパートとの対話を途切れさせない。

ドメインについての洞察を得ようと模索することで、リファクタリングのプロセスのための広範なコンテキストが作られる。

古典的なリファクタリングのシナリオは、開発者1人か2人がキーボードの前に座って、あるコードを改善できることに気づき、その場で（当然、結果を検証するユニットテストと共に）変更するというものだ。このプラクティスは常に行うべきだが、これがすべてではない。

直前の5つの章では、リファクタリングの応用的な見方を、従来のマイクロリファクタリングのアプローチに重ねて示してきた。

開　始

より深い洞察へ向かうリファクタリングは、さまざまな方法で始められる。始まりは、コードの持つ問題、つまり複雑さやぎこちなさへの対応かもしれない。コードに対して標準的な変換方法を用いる代わりに、開発者は問題の根源がドメインモデルにあると感じるのだ。ある概念が欠けているのかもしれないし、何らかの関係性が間違っているのかもし

れない。

リファクタリングに対する従来の見方から離れると、コードが整然として見えたとしても、モデルの言語がドメインエキスパートと切り離されたように見えたり、新しい要求が自然には適合しなかったりする場合には、やはりドメインモデルが間違っているという認識が得られることがある。リファクタリングは学習の結果生まれることもある。より深くドメインを理解するようになった開発者は、モデルをもっとわかりやすくしたり、役立つようにしたりするための好機を見出すのだ。

問題のある場所を発見するのは、最も困難で不確実であることも多い。その後で初めて、開発者は新しいモデルの要素を体系的に探し出せるようになる。同僚やドメインエキスパートとブレインストーミングすることもできる。アナリシスパターンやデザインパターンに見られるような、本に書かれた体系的な知識を利用することもできる。

探究チーム

モデルに対する不満の原因が何であれ、次の段階は、意図を明確かつ自然に伝達するようモデルを改良する方法を探し出すことである。そのためには、すぐに気づけて、数時間で終わるような、わずかな変更をするだけでよいかもしれない。その場合、変更は従来のリファクタリングに似たものとなる。しかし、新しいモデルを模索するには、より長い時間がかかり、より多くの人々の参加が必要となるだろう。

変更を先導する人が、開発者数人を選び出す。その種の問題を考え抜くことを得意とする人、ドメインの中の該当する領域に精通している人、あるいは強力なモデリングスキルを持つ人などだ。問題の中に繊細な箇所がある時には、必ずドメインエキスパートを参加させる。この4、5人のグループで会議室やコーヒーショップへ行き、30分から1時間半のブレインストーミングを行う。UML図をスケッチしたり、オブジェクトを使用してシナリオのウォークスルーを試したりする。主題に関するエキスパートが、モデルを理解して役に立つと思っていることを確認する。満足のいくものを見つけたら、戻ってコーディングする。あるいは、数日かけてじっくり考えることにして、戻って何か別の作業をする。数日後、そのグループは再び集まって、またウォークスルーを行う。今回は、それまでに考えたことについて一晩二晩寝かせているから、前より自信もあり、何らかの結論に至る。そこで、コンピュータの前に戻り、新しい設計をコーディングする。

このプロセスを生産的なものに保つ上で、いくつかポイントがある。

- **自主的な意思決定。**少人数のチームであれば、設計の問題を探究するためにその場で編成することができるし、数日間作業した後で解散することもできる。長期的な手の込んだ組織構造は必要ない。

- **スコープと休憩。**数日の間隔を空けて、短いミーティングを2、3回すれば、試す価値のある設計が作り出せるはずだ。長引かせても意味はない。行き詰まるようなら、一度に多くをやりすぎているのかもしれない。設計の中でより小さい側面を選び出し、それに集中すること。
- **ユビキタス言語の駆使。**他のチームメンバ、特に主題に関係するエキスパートをブレインストーミングに参加させると、ユビキタス言語を実践し改良する機会が生まれる。この作業による最終的な成果は、その言語が改良されることであり、もともとの開発者はそれを引き取って、コードの中において形式化する。

本書のこれまでの章で、私は、よりよいモデルを探している開発者とドメインエキスパートの対話を紹介した。本格的なブレインストーミングは、ダイナミックでかたちが定まらず、信じられないほど生産的なものである。

先達の技

車輪の再発明が常に必要とは限らない。欠けている概念や、より優れたモデルを求めるブレインストーミングのプロセスは、きわめて懐が深く、どんな情報源からの考え方でも吸収できる。さらにこのプロセスは、それを現場の知識と組み合わせてかみ砕き続けることで、現状に対する答えを探すのだ。

ドメイン自体に関する文献やその他の情報源からも、着想を得ることができる。該当する分野の人々は、ソフトウェアを動かすのに適したモデルは作っていないかもしれないが、概念を体系化して役立つ抽象を見つけてはいるだろう。こうしたことを知識をかみ砕くプロセスの糧にすれば、より豊かな成果をより早く上げることにつながり、その成果はドメインエキスパートにとっても、より馴染みのあるものになるはずだ。

他者の経験を、アナリシスパターンというかたちで利用できることもある。この種の情報からはドメインに関する本を読むのと同じ効果も得られるが、この場合には特にソフトウェア開発に向けたものになっている。しかも、該当するドメインにおいてソフトウェアを実装するという経験に、直接基づいているはずなのだ。アナリシスパターンによって、とらえにくいモデルの概念が提供され、多くの間違いを回避できるようになる。しかし、これは手順書ではない。知識をかみ砕くプロセスの糧となるものなのだ。

ばらばらだった断片が組み合わされていくにつれ、モデルの関心事と設計の関心事を並行して処理しなければならなくなる。この場合でも、常にすべてを一から発明しなければならないわけではない。デザインパターンがドメイン層で採用されることも多いが、それは実装上の必要性とモデルの概念の両方に適合した時なのだ。

同様に、数学や述語論理のような一般的形式がドメインの一部に合致するのであれば、

その部分を括り出して、その形式に関する体系のルールを適合させることができる。こうすることで非常に堅固で、理解しやすいモデルが提供される。

開発者のための設計

ソフトウェアはユーザだけのためのものではなく、開発者のためのものでもある。開発者は、コードをシステムの他の部分と統合しなければならない。イテレーティブなプロセスで、開発者は何度も繰り返しコードを変更する。より深い洞察へ向かうリファクタリングは、しなやかな設計につながると同時に、しなやかな設計の恩恵も受ける。

また、しなやかな設計は意図を伝える。そういう設計では、コードの実行結果が予想しやすくなる。したがって、コードを変更した結果も容易に予想できる。しなやかな設計は、精神的な過負荷を制限することにも役立つ。これは主に、依存関係と副作用を抑制することによって実現される。しなやかな設計が基づくのは、ドメインについての深いモデルであり、こうしたモデルが適切な粒度となるのは、ユーザにとって最も重大なところだけだ。しなやかな設計によって、よく変更される場所が柔軟になり、あらゆる場所がシンプルになる。

タイミング

変更を完全に正当化できるまで待つのは、待ちすぎというものである。プロジェクトはすでに重いコストを負っていて、先延ばしにすると変更するのがより困難になる。対象となるコードには今よりも手が入り、さらに多くの他のコードに組み込まれることになるからだ。

継続的なリファクタリングは「ベストプラクティス」と考えられるようになってきたが、ほとんどのプロジェクトチームは依然として慎重すぎる。コードを変更するリスクと、変更にかかる開発者の時間のコストを見込んでいるためだ。しかし、それほど簡単に見抜けないのが、設計をぎこちないままにしておくリスクと、その設計を何とかするためのコストである。リファクタリングを行いたい開発者は、その決定が正当なものであると証明するよう求められることが多い。こうした要求は理にかなっているように見えるが、もともと難しいものを不可能なほど難しくして、リファクタリングを抑圧する（または見えない所で行わせる）傾向がある。ソフトウェア開発は、変更することで得られる利益や、変更しないことで生じるコストを正確に割り出せるような、予測可能なプロセスではない。

より深い洞察に向かうリファクタリングは、ドメインの主題に関する継続的な探究、開発者の教育、さらに開発者とドメインエキスパートの間での合意などの一部にならなけれ

ばならない。それゆえ、次の場合にはリファクタリングすること。

- 設計が、ドメインに関するチームの現在の理解を表現していない。
- 重要な概念が設計で暗黙的になっている（かつ、それを明示的にする方法がわかっている）。
- 設計において重要な部分を、よりしなやかにする好機がある。

この積極的な姿勢は、好きな時に好きな変更をすることを正当化するものではない。リリース前日にはリファクタリングをしないこと。技術的な名人芸を披露するだけでドメインの中核に切り込んでいない、括弧つきの「しなやかな設計」を導入してはならない。ドメインエキスパートにその使用を納得させることができない括弧つきの「より深いモデル」は、どれほどエレガントに見えようと導入してはならない。現状を絶対視せず、安全でいられる場所を超えて、リファクタリングを支持する方向に向かうこと。

好機となる危機

Charles Darwinが発表してから1世紀以上の間、進化についての標準的なモデルは、種は時間をかけて、徐々に、安定しつつ変化するというものだった。1970年代になって、このモデルは突然、「断続平衡説」（punctuated equilibrium）モデルに取って代わられた。進化に対するこの拡張された考え方によれば、長期間の緩やかな変化や安定は、比較的短い期間におこる急激な変化の爆発によって中断される。それから、また新しい平衡に落ち着く。ソフトウェア開発は、背後で意図的に方向づけられており、これは進化論にはないが（プロジェクトによっては、明白に方向づけられていないものもあるかもしれないが）、それでもこの種類のリズムに従っている。

リファクタリングも、古典的記述では非常に安定したものに見える。だが、より深い洞察へ向かうリファクタリングは、それとは異なるのが普通だ。モデルを安定して改良しているところで、突然ある洞察へと導かれ、それによって、あらゆるものが刷新される。こういうブレイクスルーは毎日起こるものではないが、深いモデルとしなやかな設計につながる変更の大部分は、そこから現れる。

そうした状況は好機には見えないことが多く、むしろ危機のようにとらえられる。突然、モデルの中に、明らかに不適切な部分があるとわかるのだ。モデルが表現できる内容に大きな穴が開いていたり、きわめて重要な領域であるのにモデルが明瞭でなかったりする。おそらく、モデルが単に間違った表現をしているのだろう。

このことが意味しているのは、チームが理解の新しいレベルに到達したということだ。今や高められた視点から見れば、古いモデルは貧弱に見える。この視点であれば、はるか

に優れたモデルを思いつくことができる。

より深い洞察へ向かうリファクタリングは、継続的なプロセスである。暗黙的な概念が認識されて明示的になる。設計の一部は、よりしなやかになる。もしかしたら、宣言的スタイルになるかもしれない。開発をしていると突然、ブレイクスルーが訪れ、深いモデルへと突き進む。そしてまた、安定した改良が始まるのだ。

第4部　戦略的設計

第14章　モデルの整合性を維持する 339
第15章　蒸　留 405
第16章　大規模な構造 445
第17章　戦略をまとめ上げる 491

システムが複雑になりすぎて、個々のオブジェクトのレベルで完全に理解することができなくなると、巨大なモデルを扱い、あるいは把握するためのテクニックが必要になる。本書第4部では、モデリングプロセスをスケールアップさせて、非常に入り組んだドメインに対応させるための原則を説明する。こうした意思決定のほとんどは、チームレベルか、複数のチーム間で交渉して行わなければならない。こうした意思決定は、設計と政治が交差することの多い場所で行われるのだ。

大掛かりなエンタープライズシステムがたいてい掲げる目標は、ビジネス全体にまたがる緊密に統合されたシステムを作るというものである。だが、そうした組織のほぼすべてにおいて、ビジネスモデル全体はあまりにも巨大で複雑になるため、単一のユニットとして管理することはできないし、理解することさえもできない。システムは、概念も実装も、より小さい部分に分解しなければならない。ここでの課題は、**統合による恩恵を失わずに**、このモジュール性を実現することである。システムにおける別々の部分の相互運用を可能にすることで、ビジネスにおける多様な業務の調整を支援できるようにするのだ。あらゆるものを含む一枚岩のドメインモデルは大きすぎて扱いにくく、微妙な重複と矛盾が詰め込まれる。逆に、別々の小さなサブシステムの集合がその場しのぎのインタフェースでつなぎ合わされていたら、エンタープライズ全体の問題を解決する力は失われ、統合のあらゆるポイントにおいて、一貫性に関わる問題が発生する。この両極にある落とし穴は、いずれも、進化する体系的な設計戦略によって回避できる。

このような規模であっても、ドメイン駆動設計が、実装と結びつかないモデルを作り出すことはない。あらゆる意思決定はシステム開発に直接影響するはずであり、そうでないなら的外れである。戦略的設計の原則は、設計上の意思決定を導いて、各部分の相互依存関係を減らし、設計をより明確にしながら、決定的に重要な相互運用性と相乗効果を失わないようにするものでなければならない。システムにおいて概念的にコアとなる場所、つまり、システムの「ビジョン」をとらえるモデルに集中しなければならないのだ。**しかも、これらすべては、プロジェクトが行き詰まることのないように行わなければならない。**その目標を達成するため、第4部では広い範囲にわたる3つのテーマを考察する。それが、コンテキスト、蒸留、大規模な構造である。

コンテキストはこれらの原則の中で最もわかりにくいものだが、実は最も根本的なものである。成功するモデルは、その大小を問わず、最初から最後まで論理的に一貫しているべきで、矛盾したり定義が重複したりしていてはならない。エンタープライズシステムが統合するサブシステムの中には、出自が多様なものがある。また、サブシステムにあるアプリケーションが、あまりに異なっているために、同一の視点で見ることのできるところがドメインの中にほとんどないものもある。異質なもので構成される部分の中で、モデルが暗に示されているだけであれば、そのモデルを統合するというのは、無理な注文かもし

れない。明示的に境界づけられたコンテキストを定義した上で、モデルを適用するのはその内部に限定し、さらに、必要に応じてコンテキスト同士の関係も定義することによって、モデラは、モデルの質が低下するのを避けられる。

蒸留によって雑音が減り、注意の対象が適切に絞られる。ドメインにおいて周辺的な重要でない問題に、かなりの努力が費やされるのはよくあることだ。すべてを包括するドメインモデルは、システムにおいて最も付加価値のある特別な側面を目立たせて、その部分にできる限り多くの力を与えるような構造にする必要がある。補助的なコンポーネントにも決定的に重要なものはあるが、それぞれ適切な視点でとらえなければならない。このように焦点を合わせることで、システムのきわめて重大な部分に努力を向けやすくなるだけでなく、システムのビジョンを失わずにいられる。戦略的蒸留によって、巨大なモデルは明確になる。そして、明確に見えるようになることで、コアドメインの設計はより役立つものになり得る。

大規模な構造によって、全体像が完成する。非常に複雑なモデルにおいては、「木を見て森を見ず」になってはならない。蒸留によって、注意の対象をコアとなる場所に絞り、他の要素を脇役として示せるようになる。だが、包括的なテーマがなければ、それら要素の関係は依然としてあまりに紛らわしいままかもしれない。こうした包括的なテーマは、システム全体にわたって、設計要素やパターンを適用するものなのだ。大規模な構造に対するいくつかのアプローチを概観してから、そのパターンの1つである責務のレイヤについて掘り下げ、そういった構造を使用する意義を探ることにする。ここで議論する特定の構造は一例にすぎず、包括的なカタログではない。進化する秩序のプロセスを通じ、必要に応じて新しいものを考案するか、既存のものを修正すべきだ。こうした構造の中には、設計に統一性をもたらし、開発を加速して統合を改善できるものもある。

これらの3つの原則は、別々に使っても役に立つが、一緒に使うと特に強力になり、優れた設計を作り出す上で助けになる。適用するシステムが無秩序に広がっていて、完全に理解している人がいなくても、優れた設計を作り出せるのだ。大規模な構造は、全く異なるものから構成される部分に一貫性をもたらして、それらの部分を調和させる上での助けになる。大規模な構造と蒸留によって、各部分の間にある複雑な関係を理解できるようになる一方で、全体像を見失わずにいられる。境界づけられたコンテキストによって、別々の部分で作業を進めても、モデルを壊してしまったり、意図せず断片化してしまったりすることがなくなる。さらに、これらの概念をチームのユビキタス言語に追加することで、開発者は独自の解決策を考え出せるようになるのだ。

第14章

モデルの整合性を維持する

私がかつて関わったあるプロジェクトでは、複数のチームが並行して主要な新システムに取り組んでいた。ある日、顧客に請求書を送るモジュールを担当していたチームで**料金**（**Charge**）というオブジェクトを実装する準備が整ったのだが、その時に別のチームですでに構築したものがあることがわかった。彼らは、既存のオブジェクトをまじめに再利用することにした。このオブジェクトに「支出コード」がないことがわかったので、自分たちで追加した。必要としていた「記帳金額」属性はすでにあった。この属性のことは「未払い金額」と呼ぶつもりでいたが、名前は問題ではないと考えて「記帳金額」とすることにした。さらにメソッドと関連をいくつか追加すると、欲しかったオブジェクトに似たものが得られ、もともとあったものも妨害せずに済んだ。自分たちには必要ない関連の多くを無視しなければならなかったが、作成したアプリケーションモジュールを実行させることはできたのである。

数日後、もともと**料金**オブジェクトを使っていた支払アプリケーションのモジュールで、不思議な問題が浮上した。誰も入力した覚えのない、全く意味のない奇妙な**料金**が現れたのである。一部の機能、特に過去1か月間の税金帳票が使用されると、プログラムがクラッシュし始めた。調査によって明らかになったのは、当月の全支払に対して控除可能額を合計する機能が使用された時にクラッシュするということだった。この不思議なレコードは「控除可能パーセント」フィールドに値がなかったが、データ入力アプリケーションの検証ロジックはそれを必要とし、初期値を入れてさえいたのである。

問題は、この2つのグループが**異なるモデル**を持ちながら、そのことを認識しておらず、さらに、それを検出するプロセスもないということだった。どちらのグループも自分たちのコンテキスト（顧客への請求とベンダへの支払）に都合のいいように、料金の性質について推測を立てていた。こうした矛盾を解消しないでコードを結合した結果、信頼できな

いソフトウェアが出来上がったのである。

この現実について、彼らがもっと自覚してさえいれば、それをどう扱うかを自覚的に決定できたはずだ。協力して共通のモデルを作り上げ、将来の思いがけない事態を防止するため、自動化されたテストを一式書くことができたかもしれない。あるいは、単純に別々のモデルを開発し、お互いのコードには手を出さないことで合意することもできた。いずれにしても、それぞれのモデルが適用される境界について、明示的に合意するところから始まるのだ。

いったん問題がわかった後で、彼らはどうしただろうか？ 彼らは**顧客料金**クラスと**供給者料金**クラスを別々に作成し、対応するチームの要求に従ってそれぞれを定義した。差し迫った問題が解決すると、彼らは戻っていって、今までと変わらないやり方で仕事をしたのだ。まあ、仕方ない。

明示的に考える機会はあまりないのだが、モデルに対する最も根本的な要件は、内部で一貫していることだ。これは、用語の意味が常に同じであることであり、矛盾したルールが含まれていないことでもある。各用語の意味が明瞭で矛盾したルールがないという、モデルの内部におけるこうした一貫性は、**統一性**（unification）と呼ばれる。あるモデルは、論理的に一貫していなければ意味がない。理想的な世界なら、エンタープライズの全ドメインに広がる1つだけのモデルが手に入るかもしれない。このモデルは統一されていて、用語の定義に矛盾も重複もなく、ドメインに関するあらゆる論理的な声明が一貫していることだろう。

しかし、巨大なシステム開発の世界は理想とは違う。エンタープライズシステム全体で、そのレベルの統一を維持するには苦労が多いが、その割に得られるものは少ない。システムのいろいろな部分で、複数のモデルを開発できるようにする必要があるが、システムのどの部分なら枝分かれしてもよく、それら相互の関係がどうなるかについては慎重に選択する必要がある。モデルにおけるきわめて重要な部分を、緊密に統一された状態に保つ方法が必要なのだ。こういうことは、どれも自然には起こらないし、心構えがよければ生まれるというわけでもない。ただ、設計上の意識的な決定と、独自のプロセスを確立することによってのみ生まれるのだ。**巨大なシステム向けのドメインモデルを完全に統一することは、現実的ではないし、コストにも見合わない。**

だが、時には、この事実に抗ってしまうことがある。複数のモデルがあると、統合が制限されてコミュニケーションが煩雑になるという代償を支払わなければならず、ほとんどの人はそのことを理解している。その上、モデルが複数あるのは、どこかエレガントではないようにも見える。複数のモデルを持つことに対するこうした抵抗によって、時として、非常に野心的な試みが生まれ、巨大なプロジェクトのソフトウェアをすべて単一のモデルの下に統一しようとするのだ。この種の行きすぎに関して、私に責任があることはわかっ

ている。しかし、次のようなリスクを考えて欲しい。

1. 一度にあまりにも多くのレガシーシステムを置き換えようとしてしまうかもしれない。
2. 調整にかかるオーバーヘッドが能力の限界を超えてしまうために、巨大なプロジェクトは動きが取れなくなってしまうかもしれない。
3. 特殊な要件を伴うアプリケーションは、その要求を完全には満たしていないモデルを使わなければならないかもしれず、そのせいで、ふるまいをどこか別の場所に入れなければならなくなる。
4. 逆に、単一のモデルを使って万人を満足させようとすれば、選択肢が複雑になり、モデルを使うのが難しくなる。

さらに言えば、モデルの分岐は、技術的な関心事に由来するだけでなく、政治的な断片化や、経営上の別々の優先順位からも生じるかもしれない。そして、さまざまなモデルが現れるのは、チーム編成や開発プロセスの結果かもしれない。したがって、完全な統合を妨げる技術的要因が一切なくても、やはり、プロジェクトに複数のモデルが登場してしまうかもしれないのだ。

エンタープライズ全体のための統一モデルを維持するのが現実的ではないとしても、我々にできることがないわけではない。何を統一すべきかについての前向きな意思決定と、何を統一しないでおくかについての現実に即した認識を組み合わせることで、置かれている状況に関する明確な全体像を共有できる。この全体像があれば、確実に統一したい部分を統一する方向に向けつつ、統一しない部分のせいで混乱や破損が生じないようにすることに着手できる。

さまざまなモデル間の境界と関係性をはっきりさせる方法が必要である。戦略を意識的に選択し、その後は、一貫してその戦略に従わなければならない。

本章で提示するのは、モデルの境界や他のモデルとの関係性を認識し、伝達し、選択するためのテクニックである。すべてはプロジェクトの現在の領域をマッピングするところから始まる。境界づけられたコンテキストによって、各モデルを適用できる範囲が定義され、コンテキストマップによって、プロジェクトのコンテキストと、それらコンテキスト間の関係性を全体的に概観できるようになる。このようにあいまいさを減らすこと自体によっても、そのプロジェクトで物事がどう起こるかが変わるが、それだけでは十分でない。いったん境界づけられたコンテキストができると、継続的な統合のプロセスによってモデルの統一が保たれる。

そして、この安定した状況から始めることで、コンテキストを境界づけ、コンテキスト同士を関係づけるためのより効果的な戦略へと移行できるようになる。こうした戦略は、共有カーネルを用いて緊密に提携しているコンテキストから、別々の道を行く疎結合のモデルまで幅広く存在する。

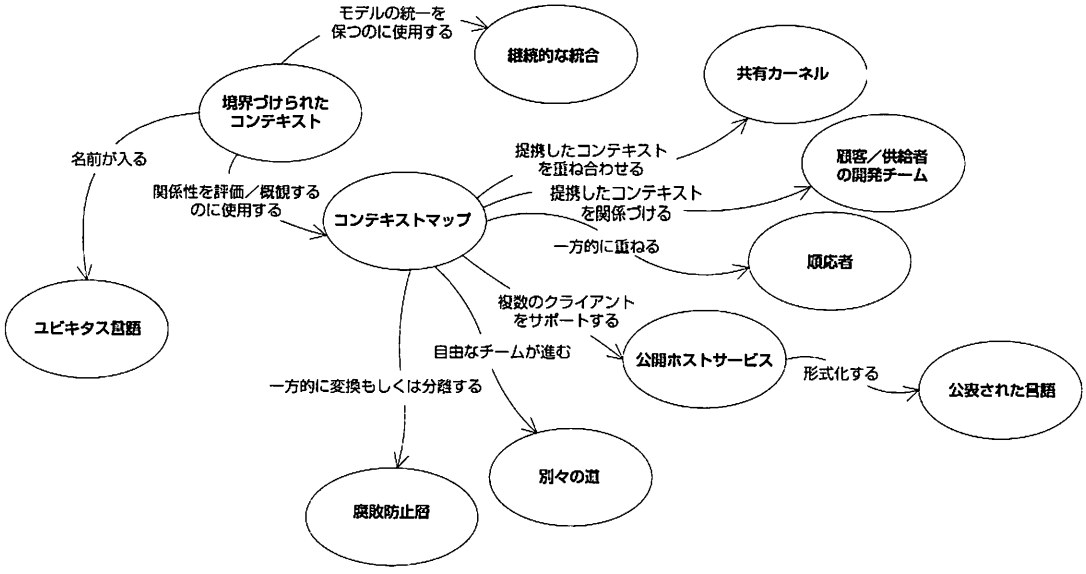
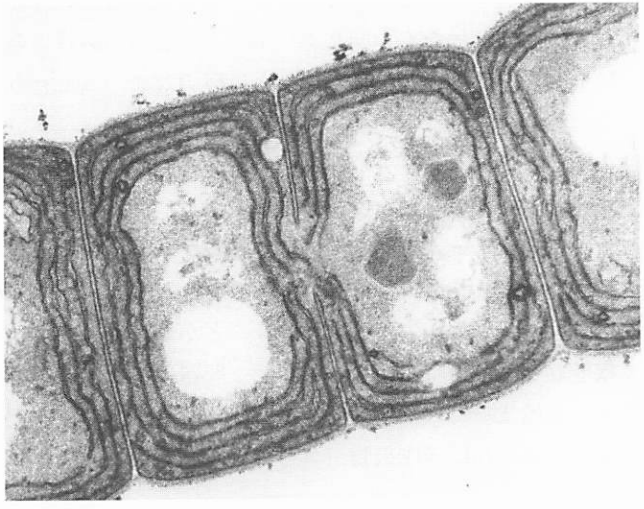


図14.1 モデルの整合性に関するパターンを表すナビゲーションマップ

境界づけられたコンテキスト（BOUNDED CONTEXT）



細胞が存在できるのは、細胞膜が内部にあるものと外部にあるものとを定義し、何が膜を通過できるかを決定するからである。

複数のモデルが大きなプロジェクトで共存し、それがうまく機能することも多々ある。さまざまなモデルがさまざまなコンテキストに適用される。例えば、新しいソフトウェアを、自分のチームではコントロールできない外部システムと統合しなければならないかもしれない。こういう状況であれば、統合先のコンテキストが全く別であって、開発中のモデルを適用できないということは誰にとっても明確だろうが、もっとあいまいで紛らわしい状況もある。本章の冒頭で取り上げた話の中では、2つのチームが、同じ新システムの中で別々の機能に取り組んでいた。それでは、彼らは同じモデルに取り組んでいたのだろうか？ 彼らの意図は自分たちがやったことの少なくとも一部を共有することだったが、何を共有し、何を共有しないのかを教えてくれる境界は存在しなかった。また、共有されたモデルを保持し、分岐が起きた時に素早く検出するプロセスも整っていなかった。システムのふるまいが突然予測できなくなって初めて、分岐してしまっていたことに気がついたのである。

1つのチームの中でさえも、複数のモデルができてしまうことがある。コミュニケーションが途切れてしまうと、そのモデルの解釈が微妙に衝突することになるかもしれない。古いコードには、モデルに以前にあった概念が反映されていることが多く、そのモデルは、最新のモデルとは微妙に異なっている。

他システムのデータ形式は異なっていて、データの変換が必要だということには誰もが気づいているが、これは問題の中の機械的な一面にすぎない。より本質的な問題は、2つのシステムで暗に示されているモデルが異なっているということなのだ。食い違いが外部システムに対してではなく、同じコードベースの中にある場合、それが認識される可能性はさらに低い。それでも、こうしたことは巨大なチームで行うプロジェクトにおいて常に発生するのである。

複数のモデルはどんな巨大なプロジェクトにも存在する。だが、別々のモデルに基づくコードが組み合わされると、ソフトウェアは、バグの温床となり、信頼できなくなり、理解しにくくなる。チームメンバ間のコミュニケーションは混乱し始める。あるモデルをどのコンテキストで適用すべきでないのかについては、ほとんどの場合不明瞭である。

物事を正しく保つのに失敗したということが最終的に判明するのは、稼働中のコードが正しく機能しなくなる時である。しかし、問題は、チームが編成され、人々が交流するそのやり方から始まっているのだ。したがって、モデルのコンテキストを明確にするためには、プロジェクトと最終成果物（コードやデータベーススキーマなど）の両方を見なければならない。

あるモデルが適用されるのは、あるコンテキストの中においてである。そのコンテキストというのは、コードのある部分かもしれないし、ある特定のチームによる作業のことかもしれない。ブレインストーミングで考案されたモデルであれば、そのコンテキストは具体的な会話に限定されるかもしれない。本書の例で使用しているモデルのコンテキストは、その具体例を説明している場所と、その具体例に関するその後の議論すべてである。モデルのコンテキストとは、モデル内の用語が特定の意味を持つと言えるようにするために適用しなければならない、あらゆる条件の集合である。

複数のモデルに起因する問題を解決するにあたっては、ある特定のモデルのスコープを明示的に定義しなければならない。このスコープはあるソフトウェアシステムにおける境界づけられた部分として定義される。そこには単一のモデルが適用され、できる限り統一性が保たれる。この定義はチーム編成と一致させなければならない。

それゆえ：

モデルが適用されるコンテキストを明示的に定義すること。明示的な境界は、チーム編成、そのアプリケーションに特有の部分が持つ用途、コードベースやデータベーススキーマなどの物理的な表現などの観点から設定すること。その境界内では、モデルを厳密に一貫性のあるものに保つこと。ただし、境界の外部の問題によって注意を逸らされたり、混乱させられたりするのを避けること。

境界づけられたコンテキストは、特定のモデルが適用できる範囲を制限する。そうすることで、チームメンバは、何が一貫性を持つべきで、それを他のコンテキストとどう関係

づけるかということについて、明確な理解を共有できるようになる。そのコンテキスト内では、モデルを論理的に統一された状態に保つこと。ただし、その境界の外に対して適用できるかどうかは気にする必要がない。他のコンテキストでは他のモデルが適用されるが、そのモデルとは用語法や概念とルール、ユビキタス言語の方言が異なっている。

境界を明示的に引くことにより、モデルを純粋に保つことができる。そして、純粋であることによって、モデルを適用できるところでは、有効なものとなる。同時に、注意を他のコンテキストに向けた際の混乱も避けられる。境界を横断して統合するには必然的に変換が行われることになるが、それは明示的に分析することができる。

コラム――境界づけられたコンテキストはモジュールではない

この問題は混同されることもあるが、動機が異なる別々のパターンなのだ。確かに、2つのオブジェクトの集合が別々のモデルを構成していると認識される場合、そうしたオブジェクトはほとんど常に別個の**モジュール**に置かれることになる。そうすることで、別々の名前空間（異なる**コンテキスト**には不可欠）と何らかの境界が与えられる。

しかし、**モジュール**は1つのモデルの中で複数の要素を構成するものでもあり、別の**コンテキスト**に対して、必ずしも意図を伝えるものではない。**境界づけられたコンテキスト**の中で**モジュール**が個別の名前空間を作成することにより、実は、偶発的に発生するモデルの断片化が見つけにくくなっているのだ。

例 14.1 ――予約コンテキスト

ある輸送会社には、貨物の予約用に新しいアプリケーションを開発する社内プロジェクトがある。このアプリケーションはオブジェクトモデルによって駆動されることになっている。このモデルが適用される境界づけられたコンテキストは何だろうか？ この質問に答えるには、このプロジェクトで何が起こっているかを見なければならない。忘れないで欲しいのだが、ここで見るのはありのままのプロジェクトであって、理想としてのあるべき姿ではない。

あるプロジェクトチームが予約アプリケーション本体に取り組んでいる。モデルオブジェクトは修正しないことになっているが、構築中のアプリケーションはそれらのオブジェクトを表示したり、操作したりしなければならない。このチームは、このモデルの消費者なのだ。モデルは、このアプリケーション（モデルの主要な消費者）内で通用するので、予約アプリケーションは境界内にあることになる。

完了した予約は、レガシーである貨物追跡システムに渡さなければならない。新しいモデルがレガシーのモデルとは違ったものになるという意思決定は、前もってなされているので、レガシーである貨物追跡システムは境界外になる。新しいモデルとレガシーとの間で必要となる変換は、レガシーシステムの保守チームの責任となる。この変換の仕組みはモデル駆動ではない。これは境界づけられたコンテキスト内にはないのだ（境界自体の一部にはなっているのだが、これについてはコンテキストマップで議論する）。この変換がコンテキストの外にある（モデルに基づいていない）のはよいことだ。レガシーシステムチームに対して、そのモデルを実際に利用してくれというのは、非現実的だろう。彼らの主な仕事はコンテキストの外部にあるからだ。

モデルに対して責任を負うチームは、各オブジェクトについて、永続化も含めた全ライフサイクルを扱う。このチームはデータベーススキーマも管理しているので、オブジェクトと関係データベースのマッピングがわかりにくくならないよう、慎重に保ってきた。つまり、スキーマはモデルによって駆動されているので、境界内である。

さらに別のチームが、貨物船の運行スケジュールを立てるためのモデルとアプリケーションに取り組んでいる。スケジュール担当チームと予約担当チームは一緒に作業を始め、両チームとも単一の統一されたシステムを作り出すつもりでいた。この両チームは形式張らずに調整し合い、たまにオブジェクトを共有することもあるが、その方法を体系化してはいない。この2チームは同じ境界づけられたコンテキスト内で仕事をしているわけではないのだ。これが危険なのは、自分たちが別々のモデルに取り組んでいることに、彼らが気づいていないからである。この状況を管理するプロセスを整えない限り、彼らによって統合されるあらゆる場所で問題が起きるだろう（本章で後述する共有カーネルが適した選択になるだろう）。だが、最初の一歩は、状況をあるがまま認識することである。同じコンテキスト内にないのだから、何かが変わるまでは、コードを共有しようとするのをやめるべきである。

この境界づけられたコンテキストは、この特定のモデルによって駆動されるシステムの持つ、あらゆる側面によって構成される。すなわち、モデルオブジェクトや、そのモデルオブジェクトを永続化するデータベーススキーマ、そして予約アプリケーションといったものだ。2つのチームが、主にこのコンテキスト内で作業している。それが、モデリングチームとアプリケーションチームだ。レガシーの貨物追跡システムと情報を交換しなければならず、レガシーシステムチームはモデリングチームの協力を得て、この境界で行われる変換に責任を負っている。予約モデルと運行スケジュールモデルの関係は明確に定義されていないので、その関係の定義は、いずれかのチームで、まずやらなければならないことだ。それまでの間は、コードやデータを共有する際に、細心の注意を払わなければならない。

それでは、この境界づけられたコンテキストを定義することで何が得られただろうか？ コンテキストの内側で作業するチームにとって、得られるものは明確さである。モデリングチームとアプリケーションチームは、1つのモデルを用いて、互いに一致した状態を保つべきであることを知っている。それを念頭に置いた上で、設計上の意思決定を行い、分裂を警戒するのだ。コンテキストの外側にいるチームが得るのは自由だ。同じモデルを使わないのに、なんとなく使わなければならないような気がするというグレーゾーンを歩かずに済むようになる。しかし、この例における最も具体的な収穫は、予約モデルチームと運行スケジュールチームの間で、非公式にモデルを共有することのリスクが認識されたということだろう。問題が起きないようにするためには、共有に関する費用対効果のトレードオフについて意思決定を行い、それを実現させるプロセスを整えることが本当に必要とされるのだ。モデルが含まれるコンテキストの境界がどこにあるかを全員が理解していなければ、これは実現しない。

＊ ＊ ＊

もちろん、境界は特別な場所である。境界づけられたコンテキストとそれに隣接するものとの関係に対しては、注意する必要がある。コンテキストマップは領域を図示し、コンテキストとそれらのつながりについて全体像がわかるようにする。その一方、コンテキスト間のさまざまな関係の性質を定義するパターンがいくつかある。そして、継続的な統合のプロセスによって、ある境界づけられたコンテキスト内におけるモデルの統一性が維持される。

しかし、それらに話を進める前に、モデルの統一性が崩壊したらどうなるのだろうか？どうすれば概念的な分派を認識できるのだろうか？

境界づけられたコンテキスト内での分派を認識する

モデルの違いが認識されていないことを示す兆候は、多くあるだろう。最も明白なものの1つは、コーディングされたインタフェースが組み合わせられないということである。より微妙なものとしては、予期していないふるまいも、兆候の1つかもしれない。自動化されたテストを用いた継続的な統合のプロセスにより、この種の問題はとらえやすくなる。しかし、最初の警告は、言語の混乱として現れるのが普通である。

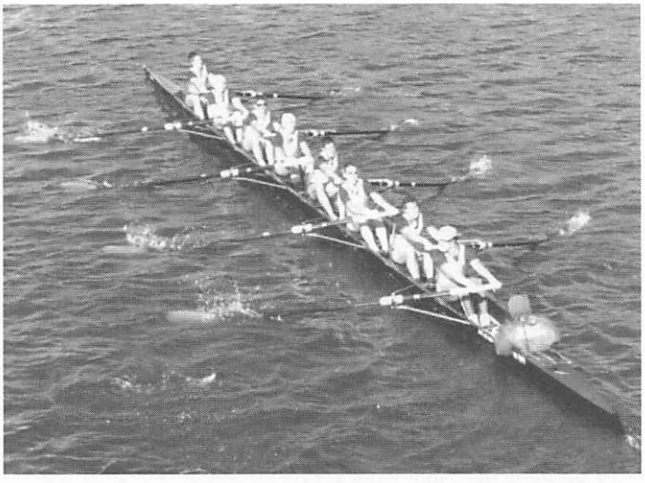
別々のモデルの要素を組み合わせると、2種類の問題が生じる。それが、**重複した概念**と**偽同族語**（false cognate）だ。概念の重複とは、実際は同じ概念を表しているモデル要素（およびそれに付随する実装）が2つあるということだ。その情報が変更されるたびに、2か所で変換を伴う変更が必要になる。新しい知識によって、オブジェクトの1つが変更されるたびに、もう一方も分析をやり直して変更しなければならない。ただし、実際には分析がやり直されることはないので、結果として同じ概念の2つのバージョンが作られ、別々のルールに従い、データまで異なることもある。その上、チームメンバは、同じことを実行する方法を、1つだけでなく2つ学ばなければならず、しかもそれをずっと同期させておかなければならないのだ。

偽同族語はそれほど一般的でないかもしれないが、知らないうちにより多くの害を与える。これは、2人の人が同じ用語（または実装されたオブジェクト）を使って、同じことを話しているつもりでいるのに、実はそうでないというものである。本章の初めに取り上げた例（2つの異なるビジネスの活動がどちらも**料金**と呼ばれていた）が典型的だ。しかし、2つの定義がドメインにおける同一の側面と実際に関連しているのに、若干異なる方法で概念化されていると、こうした衝突は一層見つけにくくなる。偽同族語があると、開発チームは互いのコードにひどい仕打ちをし合い、データベースに奇妙な矛盾が生じて、チーム内のコミュニケーションが混乱することになる。**偽同族語**という用語は、本来、自然言語に適用されるものだ。例えば、英語を話す人がスペイン語を勉強すると、よくembarazadaという単語を間違って使う。この言葉の意味は「当惑した（embarrassed）」では

なく、「妊娠している」なのだ。いやはや。

これらの問題に気づいたら、チームは決断を下さなければならない。モデルを元のように一緒にして、分裂を防ぐようにプロセスを改良してもよいし、その分裂を起こしているのが、正当な理由からモデルを異なる方向に動かしたい人々なのであれば、彼らが独立して開発できるようにしてもよいだろう。こうした問題に対処することが、本章でこれから取り上げる各パターンのテーマである。

継続的な統合（CONTINUOUS INTEGRATION）



境界づけられたコンテキストを定義したら、それを安定させなければならない。

＊　＊　＊

多くの人々が同一の境界づけられたコンテキストで作業していると、モデルが分裂する傾向は強くなる。チームが大きければ大きいほど、問題も大きくなるが、3、4人ほどの少人数でも深刻な問題に直面することがある。しかし、システムをさらに小さいコンテキストに分割してしまうと、結局は有益なレベルで統合し、一貫させることができなくなる。

誰か別の人がモデル化したオブジェクトや相互作用の意図を開発者が完全に理解せず、もともとの目的には使えなくなるような方法で変更してしまうことがある。自分の取り組んでいる概念が、すでにモデルの別の部分に埋め込まれていることに気づかないで、その概念とふるまいを（不正確に）重複させることもある。そうした他の表現があることに気づいていながら、既存の機能を壊すことを恐れて手を加えられず、概念と機能を重複させてしまうこともある。

どんな規模であっても、統一されたシステムを開発するのに必要なコミュニケーションのレベルを維持することは非常に困難である。コミュニケーションを推進して、複雑さを減らす方法が必要だ。また、開発者が既存のコードを壊すことが怖いからといって、機能を重複させてしまうような、慎重すぎる行動を防止するセーフティネットも求められる。

こういう環境でこそ、エクストリームプログラミング（XP: Extreme Programming）は本領を発揮する。XPの多くのプラクティスは、常に多くの人々によって変更される設計におい

て一貫性を維持するという、この具体的な問題をねらいとしている。最も純粋なかたちのXPは、単一の境界づけられたコンテキスト内でモデルの整合性を維持するのに適している。しかし、XPを採用するかどうかに関わらず、継続的な統合のプロセスを持つことは不可欠だ。

継続的な統合とは、そのコンテキスト内で行われるすべての作業について、頻繁にマージして一貫性を保つことにより、分派が見られた時にすぐに発見して修正できるようにすることを意味する。ドメイン駆動設計にある他のパターンと同様、継続的な統合も2つのレベルで作用する。それが、①モデル概念の統合と②実装の統合である。

概念の統合は、チームメンバ間の絶え間ないコミュニケーションによって実現される。チームは、常に変化するモデルに関して共有された理解を培わなければならない。多くのプラクティスが有効だが、最も基本的なのはユビキタス言語を継続的に練り上げることだ。一方、実装成果物の統合は、システム的なマージ／ビルド／テストのプロセスによって実現され、モデルの分派を早い段階で明らかにする。統合に向けて多くのプロセスが用いられるが、効果的なもののほとんどは、次のような特徴を共通して持っている。

- 段階的に行われ、再生可能なマージ／ビルドテクニック
- 自動化されたテストスイート
- 変更が統合されない期間に対して、妥当な短さの上限を設定するルール

効果的なプロセスににおいて実装と対(つい)になるのが、**概念の**統合である。ただ、これが正式に含められることはめったにない。

- モデルとアプリケーションに関する議論における、ユビキタス言語の絶え間ない鍛錬

ほとんどのアジャイルプロジェクトでは、少なくとも1日に一度、各開発者によるコードの変更をマージしている。この頻度は変更のペースに合わせて調節してよく、他のチームメンバが互換性のない変更を大量に行ってしまう前に、未統合の変更がマージできればよい。

モデル駆動設計においては、概念の統合によって実装の統合がスムーズになり、実装の統合によって、モデルの正当性と一貫性が立証され、分派が明らかにされる。

それゆえ：

すべてのコードと他の実装成果物を頻繁にマージさせるプロセスを策定すること。その際、分裂に対して素早く警告する自動化されたテストも一緒に用いること。ユビキタス言語の執拗な鍛錬を絶え間なく行い、モデルに対する共有された見方を練り上げること。概念は、別々の人の頭の中で進化していくからだ。

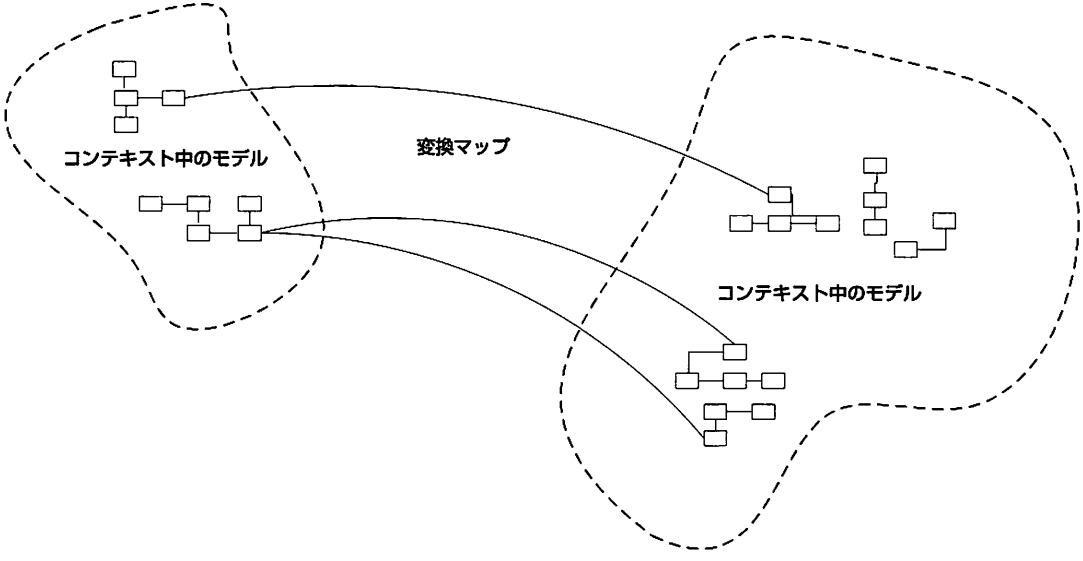
最後に、仕事を必要以上に大きくしないこと。継続的な統合が不可欠になるのは、境界づけられたコンテキスト内でだけだ。隣接したコンテキストを巻き込んだ設計に関する問

題は、変換も含め、同じペースで取り扱わなくてもよい。

＊　＊　＊

継続的な統合は、境界づけられたコンテキストが2人以上で作業する程度に大きいものであれば、どれに対しても適用されるだろう。そうすることで、その単一のモデルの統一性が維持される。境界づけられたコンテキストが複数共存している場合には、それらの関係を定め、必要なインタフェースをすべて設計しなければならない…。

コンテキストマップ（CONTEXT MAP）



個々の境界づけられたコンテキストでは、まだ全体的な視点は得られない。他のモデルのコンテキストは、依然、漠然としていて流動的だ。

＊ ＊ ＊

他のチームに所属する人々は、コンテキストの境界をあまり意識せず、境目をぼかしたり相互の接続を複雑にしたりする変更を、そうとは知らずに加えるかもしれない。別々のコンテキスト同士を接続しなければならない場合、コンテキストは相互ににじみ出す傾向がある。

境界づけられたコンテキスト間でのコードの再利用は、危険なので避けるべきである。機能とデータの統合は、変換を経て行わなければならない。混乱を減らすためには、別々のコンテキスト同士の関係を定義し、プロジェクトにあるモデルのコンテキストすべてに関する、全体的な視点を作成しなければならない。

コンテキストマップは、プロジェクト管理とソフトウェア設計が重なり合う場所にある。境界がチーム編成の輪郭に従うのは、自然な成り行きというものだ。近くで仕事をする人々は自然にモデルのコンテキストを共有するようになる。別々のチームにいる人々や、同じチームにいても会話をしない人々は、別々のコンテキストに分断されることになる。物理的なオフィス空間も影響を与えるかもしれない。ビルの両端に分かれているチームメ

ンバは、統合するために特別な努力をしなければ、おそらく分岐してしまうからだ。まして、別々の都市にいたらどうなるかは、言うまでもない。ほとんどのプロジェクトマネージャは、こうした要因を直観的に認識しており、大まかに言えばサブシステムを単位としてチームを組織する。しかし、ソフトウェアのモデルや設計とチーム編成の間にある相互関係は、まだ十分に知られていない。マネージャもチームメンバも、作成中のソフトウェアモデルや設計における概念の再分割に対して、明確な視点を持つ必要がある。

それゆえ：

プロジェクトで機能しているモデルをそれぞれ識別して、その境界づけられたコンテキストを定義すること。これには、非オブジェクト指向のサブシステムにある暗黙的なモデルも含まれる。境界づけられたコンテキストそれぞれに名前をつけ、その名前をユビキタス言語の一部にすること。

モデル同士の接点を記述して、あらゆるコミュニケーションで必要となる明示的な変換について概略を述べ、共有するものがあれば強調すること。

まず、「現在存在する」領域の地図を書くこと。変換にはその後で取りかかること。

それぞれの境界づけられたコンテキスト内では、一貫性のあるユビキタス言語の方言ができる。境界づけられたコンテキストの名前そのものがユビキタス言語に加わるので、コンテキストを明確にすれば、設計のどの部分のモデルに関しても、あいまいにならずに話すことができる。

このマップは、特定の形式でドキュメント化しなくてもよい。私にとっては、本章にあるような図が、マップを視覚化して伝達するのに役立っている。より文章の多い記述や、図を使った別の表現の方を好む人もいるだろう。状況によっては、チームの仲間で議論するだけで十分かもしれない。求められる詳細度は異なることがあるのだ。形式がどんなものであれ、マップはプロジェクトに関わる人全員に共有され、理解されなければならない。コンテキストマップは、それぞれの境界づけられたコンテキストに対して明確な名前を与えた上で、その接点と性質を明確にしなければならない。

＊　＊　＊

境界づけられたコンテキスト間の関係は、設計の問題とプロジェクト編成の問題の両方に応じて、多くの形態をとる。本章後半で、コンテキスト間の関係について多様なパターンを紹介する。これらのパターンが効果を発揮するのはそれぞれ別々の状況においてであり、あなたが自分のマップで見出される関係を記述するための用語を提供できる。覚えておいて欲しいのは、コンテキストマップは、常に**ありのままの状況**を表しているということだ。あなたが見出す関係は、最初はこうしたパターンに合わないかもしれない。近いものがある場合には、パターン名をそのまま使いたいと思うかもしれないが、無理に使ってはならない。見出した関係をそのまま記述すること。より標準的な関係性を目指した移行は、その後で開始すればよい。

さて、完全にからみあったモデルが不整合を含んでいるといったモデルの分派を発見したら、あなたはどうするだろうか？ 危険を表すために、マップにドラゴンの絵でも描き加え、すべてを記述しようするのをやめること。その後で、正確な全体的な視点をもって、混乱している場所に取り組むこと。ささいな分派は修復可能で、立て直すためのプロセスを導入することもできる。関係があいまいであれば、最も近いパターンを選び、それを目指して進むことができる。最優先にすべきことは、明確なコンテキストマップにたどり着くことである。そのためには、すでにわかっている実際の問題を修正する必要があるかもしれない。しかし、やらなければならないこうした修正を、大々的な再構成に発展させてはならない。明快なコンテキストマップができて、すべての作業が境界づけられたコンテキストのいずれかに納まり、結びつけられたモデル間の関係がすべて明示的になるまでは、明らかな矛盾だけを変更すべきである。

いったん一貫性のあるコンテキストマップができれば、変更したいものが見えてくる。チーム編成や設計に対して考え抜いた変更を加えることもできるだろう。ただし、実際に変更が**完了するまで**は、マップを変更してはならない。

例 14.2——輸送アプリケーションにおける 2 つのコンテキスト

再び、輸送アプリケーションに戻る。このアプリケーションの持つ主要な機能の1つは、予約時に貨物の経路を自動選択することだった。このモデルは次のようなものだった。

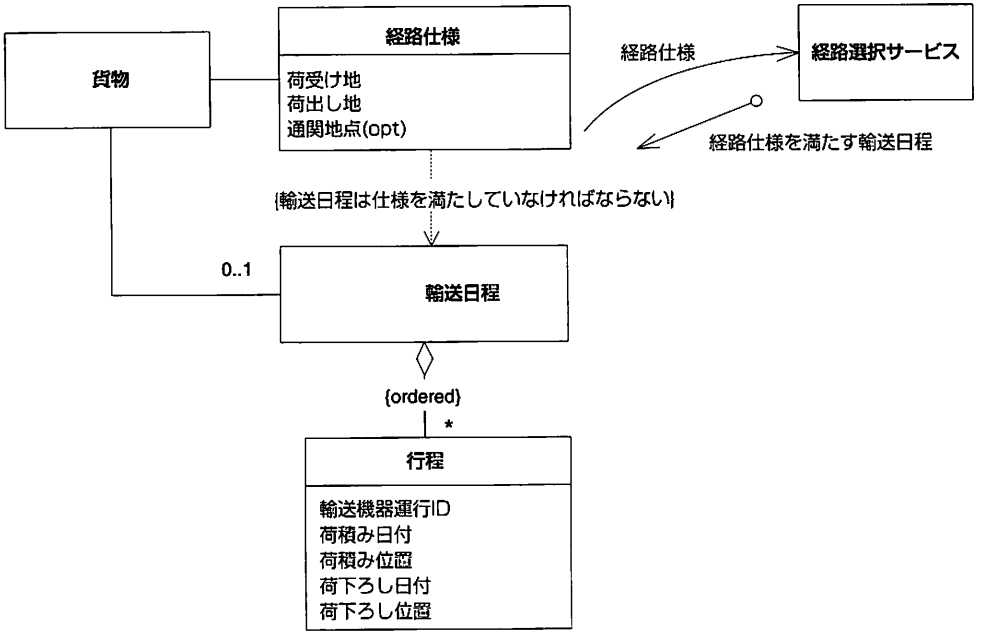


図 14.2

経路選択サービスはサービスであり、副作用のない関数で構成された意図の明白なインタフェースの背後に、ある仕組みをカプセル化している。これらの関数の結果は、表明によって性質が記述される。

1. インタフェースは、**経路仕様**が渡された場合に、**輸送日程**が戻されることを宣言している。
2. 表明は、戻される**輸送日程**が渡された**経路仕様**を満たすとはっきり述べている。

この実に難しいタスクがどのように実行されるかについては、何も述べられていない。そこで、裏に回って仕組みを見ることにしよう。

この例の元になっているプロジェクトにいた時、**経路選択サービス**の内部について、当初、私は自分勝手に想定しすぎていた。実際の経路選択業務を行うのに、輸送機器の運行を表現することになる拡張されたドメインモデルを用いようとして、さらに、それを直接、**輸送日程**を構成する**行程**に結びつけようとしたのだ。しかし、経路選択の問題を担当するチームから指摘されたのは、経路選択をうまく実行し、すでに確立されたアルゴリズムを活用するためには、

この解決策を最適化ネットワークとして実装し、運行の各行程をマトリクス内の要素として表現する必要があるということだった。彼らは、その目的のために輸送業務に関して、他とは異なるモデルを主張したのである。

当時設計されていた経路選択プロセスに対する処理上の要求に関して、彼らは明らかに正しかったので、他にもっとよい案もなかった私は譲歩した。結局、我々は別々の境界づけられたコンテキストを2つ作成したのだが、それぞれには輸送業務に関する独自の概念的構造があった（図14.3参照）。

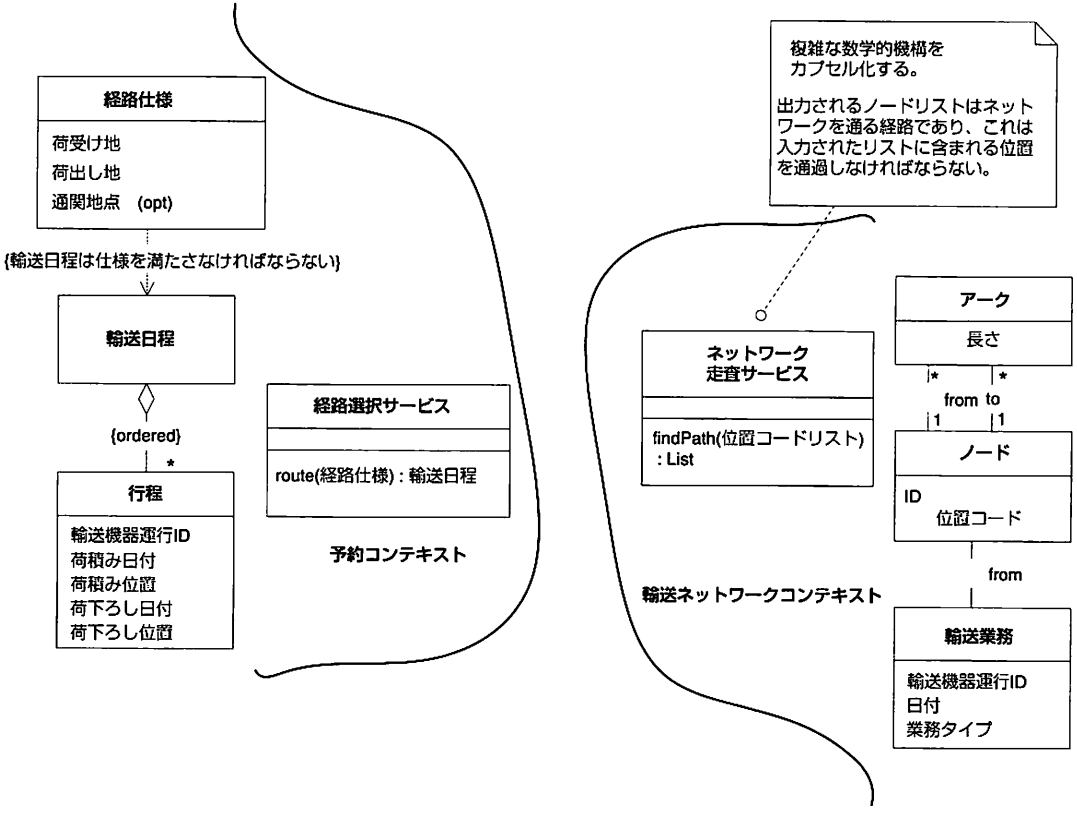


図14.3 効率的な経路選択アルゴリズムを適用できるかたちにした、2つの**境界づけられたコンテキスト**

我々の要件は、**経路選択サービス**からのリクエストを受け取り、それを**ネットワーク走査サービス**（Network Traversal Service）が理解できる用語に変換してから、結果を受け取り、それを**経路選択サービス**が提供することになっている形式に変換することであった。

これが意味するのは、2つのモデルにあるすべてを紐づける必要はなく、以下の2つの具体的な変換が実行できればよいということである。

経路仕様 → 位置コードの**リスト**
ノードIDのリスト → **輸送日程**

そのためには、一方のモデルの要素が持つ意味を調べ、それをもう一方の言葉でどう表現すればよいかを考え出さなければならない。

まず、最初の変換（**経路仕様** → 位置コードの**リスト**）では、リストに含まれる位置の順番が持つ意味について考えなければならない。リストの先頭がパスの始まりである。その後、各位置を順番に通過することを強制され、リストの最後尾にある位置に到達する。したがって、荷受け地と荷出し地はリストの先頭と最後尾であり、その中間に通関の位置が（もしあれば）が入っている。

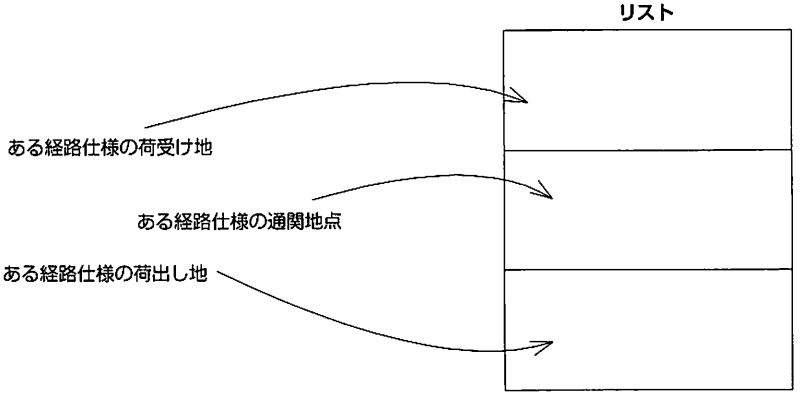


図14.4 ネットワーク走査サービスに対するクエリの変換

（ありがたいことに、2つのチームは同じ位置コードを使用したので、このレベルの変換は扱わなくてよい。）

逆の変換はあいまいになるかもしれないことに注意して欲しい。このネットワーク走査への入力には、通関地点として特別に指定される1つの位置以外にも中間地点が任意の数含まれていてよいからである。幸い、逆方向に変換する必要がないので、ここでは問題にならないが、変換できないものが存在する理由の一端を示してはいる。

次に、結果を変換しよう（**ノードIDのリスト→輸送日程**）。受け取るノードIDに基づいて**ノード**と**輸送業務**を検索するのに、リポジトリが使用できると想定する。それでは、取得した**ノード**をどう**行程**に紐づければよいだろうか？ 業務タイプコードに基づき、ノードのリストを出発／到着の対に分ければよい。そうすれば、それぞれの対が1つの行程に結びつく。

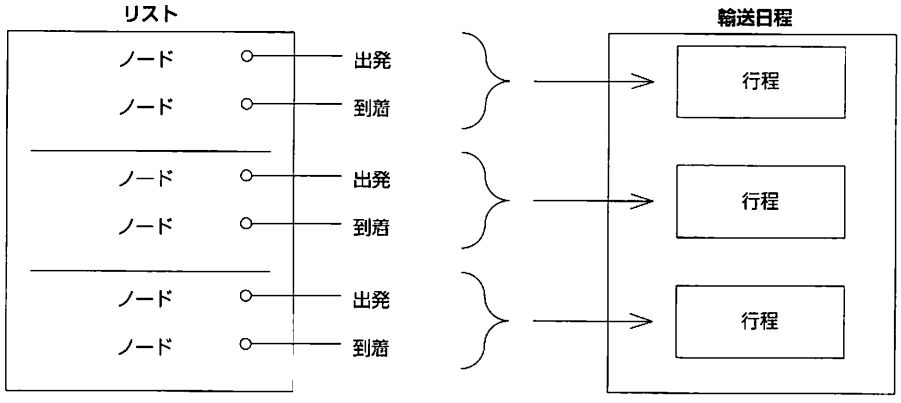


図14.5 ネットワーク走査サービスによって見つけられた経路の変換

ノードの対それぞれが持つ属性は、次のように紐づけられるだろう。

出発ノード.輸送業務.輸送機器運行ID → 行程.輸送機器運行ID
出発ノード.輸送業務.日付 → 行程.荷積み日付
出発ノード.位置コード → 行程.荷積み位置コード
到着ノード.輸送業務.日付 → 行程.荷下ろし日付
到着ノード.位置コード → 行程.荷下ろし位置コード

これは2つのモデル間にある概念的な変換マップである。ここで、変換を行うことのできる何かを実装しなければならない。このように単純なものであれば、通常私は目的に合ったオブジェクトを作成し、その上で、サブシステムの残りの部分にサービスを提供する別のオブジェクトを探すか、自分で作成する。

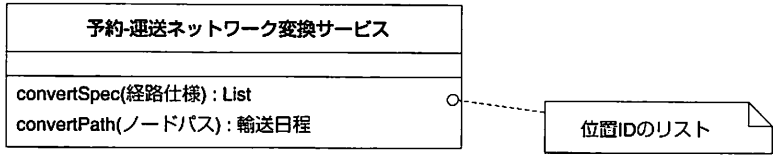


図14.6 2方向変換サービス

両チームが保守をする上で、協力して作業しなければならないオブジェクトはこれだけである。この設計によってユニットテストは非常に容易になるはずであり、2つのチームがテストスイート上で協働するというのは、実によい考えであろう。これ以外の場所では、2つのチームは別々の道を進むことができる。

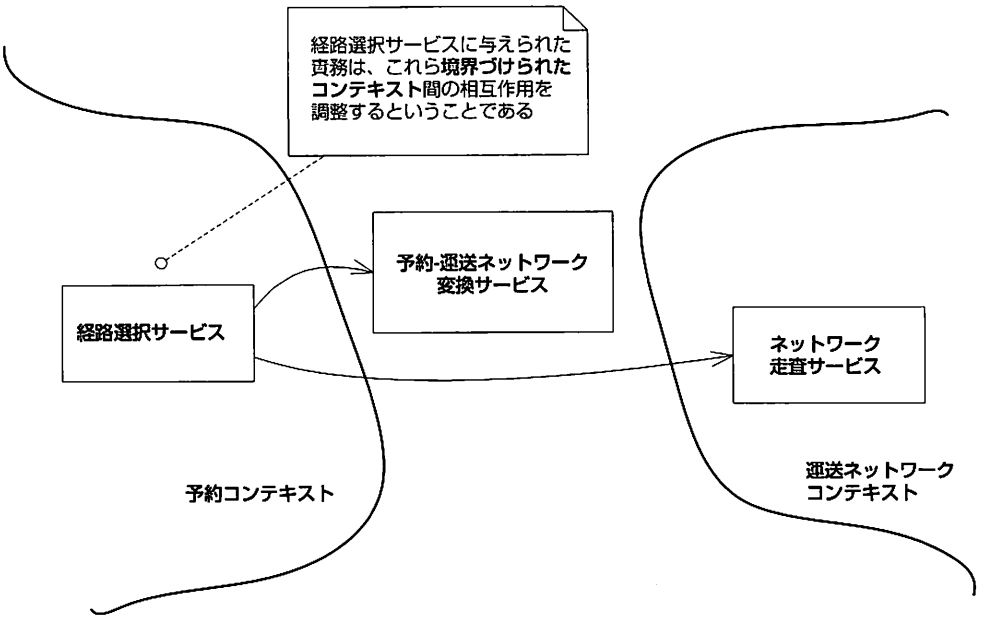


図14.7

これで、**経路選択サービス**の実装は変換サービスとネットワーク走査サービスに委譲するだけとなる。唯一の操作は、次のようなものになるだろう。

```
public Itinerary route(RouteSpecification spec){

    // 予約-逆送ネットワーク変換サービス
    Booking_TransportNetwork_Translator translator =
        new Booking_TransportNetwork_Translator();

    List constraintLocations = translator.convertConstraints(spec);

    // NetworkTraversalServiceへのアクセスを取得する
    List pathNodes = traversalService.findPath(constraintLocations);

    Itinerary result = translator.convert(pathNodes);
    return result;
}
```

悪くない。境界づけられたコンテキストによって、それぞれのモデルは比較的整理された状態に保たれ、チームがほとんど独立して作業できるようになる。初期の想定が正しければ、十分に使えたはずである（これについては、本章の後半で立ち戻る）。

この2つのコンテキスト間にあるインタフェースはかなり小さい。**経路選択サービス**のイン

●例14.2——輸送アプリケーションにおける2つのコンテキスト

タフェースにより、予約コンテキストにあるそれ以外の設計は、経路検索の世界で起きる出来事から隔離されている。このインタフェースは副作用のない関数でできているので、テストしやすい。他のコンテキストと快適に共存するための秘訣の1つは、インタフェースに対して効果的なテストを一式作ることである。「信頼するが、検証もする」とは、軍縮についてロシアと交渉する際に、レーガン大統領が言った言葉である[1]。

経路仕様を**経路選択サービス**に入力して、戻される**輸送日程**をチェックするような、自動化されたテストを一式考え出すのは容易なはずである。

モデルにとってのコンテキストは常に存在するが、意識して注意を払わなければ、重なり合ったり、変化したりするかもしれない。境界づけられたコンテキストとコンテキストマップを明示的に定義することにより、チームは、複数のモデルを統一しつつ、別々のモデル同士を結びつけるプロセスに着手できる。

コンテキストの境界で行うテスト

他の境界づけられたコンテキストとの接点をテストするのは、とりわけ重要である。テストをすることにより、変換に伴う機微や、典型的に境界に存在する下位レベルのコミュニケーションが補われる。テストは有益な早期警告システムとして機能し、特に、依存しているのが自分では制御できないモデルの詳細である場合に、安心させてくれる。

コンテキストマップを構成してドキュメント化する

ここでの重要なポイントは2つしかない。

1. 境界づけられたコンテキストには、それについて話ができるように、名前がなければならない。その名前は、チームのユビキタス言語に加えなければならない。
2. 全員がどこに境界があるかを知っていなければならず、コードのどの一部をとっても、あるいはどんな状況にあっても、そのコンテキストを認識できなければならない。

2つめの要件は、チームの文化に応じて、多くの方法で実現される。いったん境界づけられたコンテキストが定義されると、コンテキストの異なるコードは、自然と別のモジュールに分離されるようになる。しかし、これでは、どのモジュールがどのコンテキストに属するかを、どうやって追跡し続けるかという問題が残る。どこに属するかを示すのには、命名規約が使用されたり、もしくは簡単で混乱を避けられるようなその他の仕組みが使用されたりするかもしれない。

1 レーガンが翻訳したロシアの古い格言は、両陣営の問題の核心を要約したものだが、コンテキストを橋渡しすることのメタファにもなっている。

同じく重要なのは、概念的な境界について、チームの誰もが同じように理解できる方法でコミュニケーションするということである。このコミュニケーションのために私が好んで使うのは、例で使用しているような略式の図である。より厳密な図やテキストの一覧を作成し、接点や、接続と変換の責務を負う仕組みと合わせて、各コンテキストに含まれるパッケージをすべて示してもよい。このように厳密にした方がやりやすいチームもあれば、口頭での合意と多くの議論によって問題なくやっていけるチームもある。

いずれの場合であっても、コンテキストなどの名前をユビキタス言語に含めるのであれば、コンテキストマップを議論に取り入れることが欠かせない。「ジョージのチームが担当しているものが変更されているから、それと通信するこちら側の部分も変更しなければならない」などと言ってはいけない。代わりに、「**運送ネットワークモデル**が変更されているから、**予約コンテキスト**向けの変換サービスを変更する必要がある」と言うこと。

境界づけられたコンテキスト間の関係

この後に議論するパターンによってカバーされるさまざまな戦略は、2つのモデルを関係づけるためのものだ。そのモデルを組み合わせることで、エンタープライズ全体を包括することができる。これらのパターンには目的が2つある。それは、開発作業をうまく構成できるようにするための目標を与えることと、既存の構成を記述するための語彙を提供することだ。

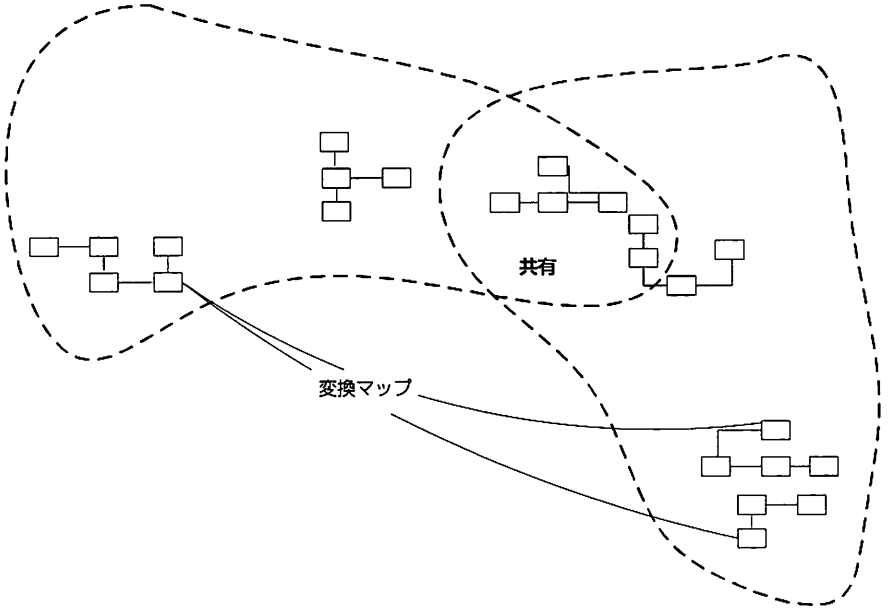
既存の関係性は、偶然であれ設計されたものであれ、これらのパターンのうちのどれかに近いものになるかもしれない。その場合には、異なっている点に十分注意しつつ、その用語を使用して記述できる。それ以降は、設計に小さな変更があるたびに、その関係性を、選択したパターンへ近づけていけばよい。

一方で、既存の関係が混乱していたり、あまりに入り組んだりしていることに気づくかもしれない。明快なコンテキストマップを作れるようにするためだけに、何らかの再構成が必要になるかもしれない。そういう状況をはじめとして、再構成を検討している場合にはどんな状況に対しても、これらのパターンはさまざまな環境に応じた広範な選択肢を提供する。よくある可変要素としては、他のモデルに対して及ぼし得るコントロールのレベルや、チーム間での協力のレベルと形態、それに機能とデータの統合の程度などがある。

これから述べるパターン一式は、最も一般的で重要なケースのいくつかをカバーしている。そのため、それ以外のケースに対しても、どのように対処すべきかを示すヒントになるはずだ。一流のチームが、緊密に統合された製品に念入りに取り組めば、巨大な統一モデルであっても配備することができる。別々のユーザコミュニティで使う必要があったり、チームの調整能力に限界があったりする場合には、共有カーネルや顧客／供給者関係にな

るかもしれない。要求を真剣に検討しつくすと、統合することが必須ではなく、各システムが別々の道を進んだ方がよいことが明らかになることもある。そして、当然ながら、ほとんどのプロジェクトはレガシーや外部システムとある程度統合する必要があり、その場合には公開ホストサービスや腐敗防止層になるかもしれない。

共有カーネル（SHARED KERNEL）



機能面での統合が制限されると、継続的な統合にかかるオーバーヘッドが大きくなりすぎると思われるかもしれない。これが特に該当するのは、継続的な統合を維持するためのスキルや政治的組織がチームにないか、1つのチームとしてはとにかく大きすぎて扱いにくい場合だ。そこで、別の境界づけられたコンテキストが定義され、複数のチームが編成されることもある。

＊　＊　＊

まとまりのない複数のチームが、密接に関連したアプリケーションに取り組んでいると、しばらくの間は作業を急いで進めることができても、それぞれが作り出すものはうまく適合しないだろう。結果的に、初めから継続的な統合を行った場合よりも多くの時間を、変換層と修復作業に費やすことになり、この二度手間のせいで共通のユビキタス言語が持つ利点を失うことになる。

多くのプロジェクトにおいて、大部分は独立して作業している複数のチーム間で、インフラストラクチャ層が共有されているのを私は見てきた。これと似たようなことが、ドメインの中でもうまく機能することがある。モデルとコードベース全体を完全に同期させるとオーバーヘッドが大きくなりすぎるかもしれないが、慎重に選択したサブセットを同期

するのであれば、少ないコストで利益のほとんどを得られる。

それゆえ：

2つのチームが共有することに合意したドメインモデルのサブセットを指定すること。もちろん、これには、モデルのサブセットに加え、コードのサブセットや、モデルの該当箇所に関連するデータベース設計も含まれる。この明示的に共有されたものには特別な地位が与えられているので、もう一方のチームに相談せずに変更してはならない。

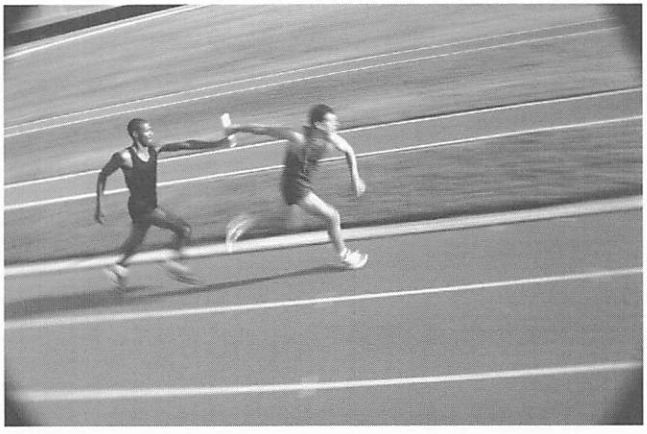
稼働しているシステムは頻繁に統合すること。ただし、チームの中で行われる継続的な統合よりは、やや頻度を下げること。この統合の際には、両チームのテストを実行すること。

このバランスは、慎重に取らなければならない。共有カーネルは、設計の他の要素ほどは自由に変更できない。変更の決定には、他チームとの相談が必要になる。自動化されたテストスイートも統合しなければならない。変更した場合には、両方のチームのテストがすべて通らなければならないからだ。通常、チームが共有カーネルに変更を加える時には、共有カーネルを別にコピーして行い、時間をおいて他のチームと統合する（例えば、1日に1回以上の頻度で継続的に統合するチームであれば、カーネルのマージは1週間に一度程度だろう）。コードの統合がいつに予定されているかに関わらず、両方のチームが変更について話し合うのは早ければ早いほどよい。

* * *

共有カーネルは、コアドメインであることもあるし、複数の汎用サブドメインの集合であることもある。あるいは、その両方であることもある（第15章参照）。両チームが必要とするものであれば、モデルのどの部分であってもよいのだ。目標は、重複を減らして（ただし、境界づけられたコンテキストが1つしかなかった場合のように、重複を取り除くのではない）、2つのサブシステム間の統合を比較的容易にすることである。

顧客／供給者の開発チーム（CUSTOMER/SUPPLIER DEVELOPMENT TEAMS）



一方のサブシステムが、本質的にもう一方のサブシステムに対して入力を与えるものであることがある。その「下流」のコンポーネントが実行するのは、分析や、「上流」のコンポーネントにほとんどフィードバックを与えないその他の関数であり、すべての依存関係は一方向になっている。この2つのサブシステムがサービスを提供するのは、全く別のユーザコミュニティに対してであるのが普通である。これらのユーザはそれぞれ違う仕事をしていて、使えるモデルも異なるかもしれない。ツールセットも異なるだろうから、プログラムコードを共有することはできない。

＊　＊　＊

上流のサブシステムと、下流のサブシステムは、2つの境界づけられたコンテキストに自然に分かれる。2つのコンポーネントが別々のスキルを必要としたり、別々のツールセットを実装に採用したりする場合に、このことは特に当てはまる。変換を一方向に対してのみ行えばよいなら、その方が簡単だ。しかし、2つのチームの間にある政治的な関係次第では、問題が起きるかもしれない。

上流チームが自由に開発の舵を取れなくなるのは、下流のチームが変更に対する拒否権を持っている場合や、変更を要求するための手続きがあまりに面倒な場合である。下流のシステムを壊してしまうのではないかと心配するあまり、上流のチームが抑制されることさえある。その一方で、下流のチームは上流チームがつける優先順位の言いなりになって、自分たちではどうすることもできない。

下流は上流から来るものを必要とするが、上流は下流が作り出すものに対して責任を負わない。何が他のチームに影響を及ぼすかを予想するには、多くの余計な努力が必要になる。だが、人間の本性とはかくあるもので、時間的なプレッシャーは知っての通りのものだから…。続きは言わなくてもわかるだろう。したがって、チーム間の関係を形式化すれば誰もが楽になる。このプロセスは、2つのユーザコミュニティの間で要求のバランスを取り、下流で必要な機能に関する作業をスケジューリングするために構成される。

エクストリームプログラミングを行っているプロジェクトでは、まさにそのための仕組みが準備されている。それが、イテレーションプランニングプロセスだ。やるべきなのは、プランニングプロセスの観点から、2チームの関係を定義することだけだ。下流チームの代表はユーザの代表と同じような役割を果たせるので、彼らをプランニングプロセスに参加させ、必要な作業に関するトレードオフについて、この「顧客」と直接議論するとよい。その結果、供給者チームのイテレーション計画は、下流チームが最も必要とする作業を含むか、合意によって作業が延期される。したがって、納品されるかどうかを期待して待つことはない。

XP以外のプロセスを用いるのであれば、さまざまなユーザの関心事のバランスを取るのに役立つ、何らかの似たような手法を拡張して、下流のアプリケーションの要求を含むようにすればよい。

それゆえ：

2つのチームの間に明確な顧客／供給者という関係を確立すること。計画時には、下流のチームが上流のチームに対して、顧客としての役割を果たすようにすること。下流の要件に必要となる作業について交渉し、予算を立てることで、提供の約束とスケジュールを全員が理解できるようにすること。

期待されるインタフェースを検証する、自動化された受入テストを共同で開発すること。そのテストを上流チームのテストスイートに追加し、継続的な統合の一部として実行されるようにすること。このテストを実施することで、下流への副作用を心配せずに、上流チームは自由に変更を行えるようになる。

イテレーションの間、下流チームのメンバは、普通の顧客同様、上流の開発者の質問に答えて問題解決を支援できるようになっている必要がある。

受入テストの自動化は、顧客とのこうした関係において非常に重要な部分である。協力関係がきわめてうまくいっているプロジェクトで、顧客が依存関係を識別して伝えることができ、また、供給者も変更について熱心に伝えようとしたとしても、テストがなければ予想外の事態は起きるものだ。そうなれば、下流チームの作業は中断され、上流チームに対して、想定外の緊急修正を強いることになる。そうならないように、顧客チームには供給者チームと共同で自動化された受入テストを開発させ、期待するインタフェースを検証

させるようにすべきである。上流チームは、そのテストを通常のテストスイートの一部として実行することになる。このテストに変更を加えるには、他のチームとのコミュニケーションが必要になるが、それは、テストの変更がインタフェースの変更を意味するからだ。

顧客／供給者の関係は、別々の企業で進められるプロジェクトの間であっても、ある顧客が供給者のビジネスにとって非常に重要な場合には発生する。この場合は、本末転倒になりかねない。つまり、影響力のある顧客は、上流のプロジェクトを成功させる上で重要な要求を出すこともあるが、同様に、その要求によって上流のプロジェクトの開発が阻害されるかもしれないのだ。要件に対応するプロセスを形式化することで、両者が恩恵を受ける。費用対効果のトレードオフは会社間の関係で見るよりも、内部的なITの状況として見た方がはるかにわかりやすいからだ。

このパターンには、きわめて重要な要素が2つある。

1. この関係は顧客と供給者の関係でなければならない。これが暗に示しているのは、顧客の要求が最優先であるということだ。下流チームだけが唯一の顧客というわけではないため、さまざまな顧客の要求は交渉の中でバランスを取らなければならない。それでも、優先されるべきであることに変わりはない。この状況は、時折現れる貧者と富者の関係とは対照的である。貧者と富者の関係においては、下流チームが要求を実現してもらうために、上流チームに懇願しなければならない。
2. 上流チームが、下流を壊してしまわないかと恐れることなくコードを変更できるようにし、下流チームが上流チームのことを常に気にすることなく自分の仕事に集中できるようにするには、自動化されたテストスイートがなければならない。

リレー競走では、バトンを受ける走者は、常に後ろを見て確認することはできない。バトンを持つ走者が正確に手渡してくれることを信頼できないと、チームはどうしようもなくスピードを落としてしまう。

例14.3——収益分析と予約

頼りになる輸送システムの例に戻ろう。きわめて専門的なチームが編成され、会社が受けつけた予約をすべて分析して、どう収益を最大化するかを調べることになった。チームメンバは、船内に空きスペースがあるのを見つけて、オーバーブッキングを増やすことを勧めるかもしれない。あるいは、船舶がばら積み貨物で早々に満杯になってしまい、より利益率の良い特殊な貨物を断らないといけなくなっていることを発見するかもしれない。その場合は、利益率の高い貨物用にスペースをあらかじめ空けておくか、ばら積み貨物に対する価格を引き上げることを勧めるだろう。

この分析を行うために独自の複雑なモデルが使用され、実装にあたっては、分析モデル構築用ツールの付属したデータウェアハウスが使用される。予約アプリケーションから大量の情報を取得する必要もある。

最初から、境界づけられたコンテキストが2つあることは明確である。別々の実装ツールが使われ、何よりも、別々のドメインモデルが使われているからだ。これらの関係はどういうものになるべきだろうか？

共有カーネルが理にかなっていると思われるかもしれない。収益分析が関心の対象としているのが予約モデルのサブセットであり、収益分析の独自のモデルには貨物や価格などの重複する概念があるからである。しかし、異なる実装技術が使用されている場合、共有カーネルを用いるのは難しい。その上、収益分析チームが扱うモデリングニーズはきわめて特殊であるため、彼らは絶えずモデルをいろいろと試しながら、代わりとなるモデルにも当たってみる。予約コンテキストから必要なものを変換して自分たちのコンテキストに取り込む方がよいだろう。(他方で、共有カーネルを使うことができれば、変換の負荷ははるかに軽くなる。モデルを自分たちのために実装し、データを新しい実装向けに変換することは相変わらず必要だが、モデルが同じなら変換は単純にできるはずだ。)

予約アプリケーションには収益分析に対する依存関係がない。自動的にポリシーを調整するつもりはないからだ。人間の専門家が意思決定を行い、それを必要な人々とシステムに伝達する。したがって、ここにあるのは、上流／下流の関係だ。下流が必要とするものを、次に挙げる。

1. 予約業務では必要としないデータ
2. データベーススキーマが安定していること（または、少なくとも変更があった時には、確実に通知されること）、もしくはエクスポート用ユーティリティ

幸いなことに、予約アプリケーション開発チームのプロジェクトマネージャは収益分析チームを支援するのに乗り気である。そうでなければ問題になる可能性はあった。実際に日々の予約を処理する業務部門の報告先が、実際に収益分析を行う担当者ではなく、副社長だからである。しかし、上層の経営陣が収益管理に深い関心を持っており、過去に2部門間の協力に問題があったことを知っていたので、ソフトウェア開発プロジェクトの体制を作った際に、両チームのプロジェクトマネージャが報告する相手を同じにしたのだった。

したがって、すべての要件が顧客／供給者の開発チームを適用するように整えられている。

私はこのシナリオが多くの場面で進化するのを目にしてきたが、そこでは分析ソフトウェア開発チームと業務ソフトウェア開発チームが顧客／供給者の関係になっていた。上流チームのメンバが、自分たちの役割は顧客に仕えることだと考えると、非常にうまくいった。こうした関係は、ほぼ常に非公式に組織され、どのケースでも、2人のプロジェクトマネージャが個人的に仲がよいほど、うまくいったのである。

あるXPプロジェクトでは、各イテレーションにあたって、下流チームの代表が「計画ゲーム」で顧客の役割を演じるというかたちで形式化されており、（アプリケーションの機能を使う）本来の顧客代表と集まって、どの作業をイテレーション計画に入れるかを交渉していた。このプロジェクトは小さい企業で行われていたので、共通の直近の上司が、組織でそれほど離れた上層にいなかった。これは実にうまくいった。

＊　＊　＊

顧客／供給者チームが成功するのは、2つのチームが同じ経営陣の下にいて、最終的な目標を共有しているか、実際に顧客と供給者の関係である別々の企業にいる場合であるようだ。上流チームを動機づけるものが何もない場合は非常に難しい…。

順応者（CONFORMIST）



上流／下流関係にある2つのチームを、同じところから効果的に引き出せない場合、顧客／供給者チームのような協力関係に基づくパターンはうまくいかない。素朴に適用しようとすれば、下流チームが苦しむことになる。こうしたことが起きるのは、企業の規模が大きく、2つのチームが経営階層上で遠く離れていたり、共通の上司が2チームの関係に無頓着であったりする場合である。別々の会社のチームでも、顧客のビジネスがそれ単独で供給者にとって重要でない場合にはこうしたことが起こる。供給者は小さな顧客を数多く抱えているかもしれないし、市場の方向性を変えて昔からの顧客に価値を見出さなくなるかもしれない。単に供給者の経営がうまくいっていないこともある。ビジネスから撤退してしまったのかもしれない。理由が何であれ、現実には下流が自分でどうにかしなければならないのだ。

2つの開発チームに上流／下流関係があるにも関わらず、上流に下流チームの要求に応える動機がない場合、下流チームはどうすることもできない。人の役に立ちたいという思いから上流開発者は約束するかもしれないが、それが守られるとは思えない。そういう善意を信じると、下流チームは決して手に入れられない機能に基づいて計画を立ててしまうことになる。下流のプロジェクトは遅れ、チームは最終的に、与えられたもので我慢する

ことを学ぶことになってしまう。下流チームの要求に合わせたインタフェースは実現する運命にないのだ。

この状況で、考えられる選択肢が3つある。1つは上流の機能を使用するのを一切断念することである。この選択肢は、上流が下流の要求に対応することを一切想定しないとした上で、現実的に評価されるべきである。上流の機能の持つ価値を過大評価したり、コストを過小評価したりすることがあるのだ。下流チームが上流に頼らないと決めた場合には、別々の道を進むことになる（本章で後述するパターンの説明を参照）。

上流のソフトウェアを使用することの価値が非常に高く、依存関係を維持しなければならない（または、チームが変更することのできない政治的判断がなされる）こともある。この場合にも、2つの道が開かれていて、どちらを選ぶかは上流の設計の品質とスタイルによる。カプセル化の欠如や、ぎこちない抽象、下流チームでは使用できないパラダイムを用いたモデリングなどにより、その設計を扱うのが難しすぎる場合、下流チームはやはり独自のモデルを開発する必要がある。複雑化しそうな変換層には、全面的に責任を持たなければならない（本章後半にある腐敗防止層を参照）。

それに対して、設計の品質がそれほど悪くなく、スタイルもそれなりに互換性がある場合は、独立したモデルを完全に断念するのがよいだろう。このような場合で、順応者が求められるのだ。

それゆえ：

上流チームのモデルに隷従することで生じる、境界づけられたコンテキスト間での複雑な変換を取り除くこと。確かに下流の設計者がとれるスタイルは制限され、そのアプリケーションにとって理想的なモデルが作れなくなるかもしれないが、順応することを選ぶことで統合は大幅に簡略化されるのだ。また、ユビキタス言語を供給者チームと共有することにもなる。供給者がハンドルを握っているのだから、供給者がコミュニケーションしやすくした方がよい。人の役に立ちたいという思いから、彼らは情報を共有してくれるだろう。

この決定により、上流への依存関係は深くなり、アプリケーションの可能性は上流モデルに今ある能力と純粋に付加的に行われる機能追加によって制限される。心情的には全く魅力的でないので、実際には選択すべき時でも、用いられないことが多い。

このトレードオフは受け入れられないが、上流への依存関係は欠かせないというのであれば、2番目の選択肢が残っている。すなわち、腐敗防止層を作成することで、できる限り自分自身を守ること。これは変換マップを実装する際に取り得る積極的なアプローチであり、これについては後述する。

コラム——追従することが常に悪いとは限らない

巨大なインタフェースを持つ既製のコンポーネントを使用する場合には、そのコンポーネントに潜在するモデルに**順応**しなければならないのが通例である。コンポーネントとアプリケーションは、明確に異なる**境界づけられたコンテキスト**内にあり、そのコンテキストはチームの編成と管理に基づいているので、多少形式を変更するためにアダプタが必要になるかもしれないが、モデルは等しくなければならない。そうでないなら、そのコンポーネントを持つことの価値を疑うべきだ。十分に価値が得られるほど優れているなら、知識がかみ砕かれて設計に反映されているだろう。その狭い範囲内では、利用者の理解よりもはるかに進んでいるかもしれない。利用者のモデルは、このコンポーネントのスコープを超えて広がっているものだろうから、独自の概念はコンポーネント以外の部分に対して進化することになる。しかし、接続部分においては、利用者のモデルは**順応者**で、コンポーネントのモデルの導きに従う。実際、それでよりよい設計につながることもあり得るのだ。

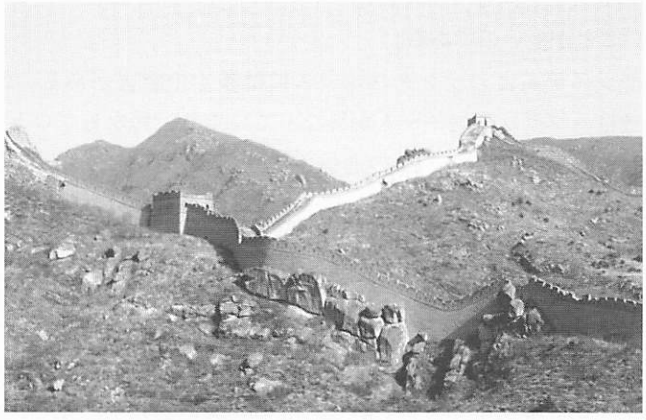
コンポーネントとのインタフェースが小さければ、統一されたモデルを共有することはそれほど必須ではなく、変換も選択肢の1つにすぎない。しかし、インタフェースが巨大で、統合がより重大な場合、先導者の後について行くのは道理にかなっている。

＊ ＊ ＊

順応者は共有カーネルと似ていると言える。どちらにも同じモデルを持つ重なり合う領域と、つけ加えることで自分のモデルが拡張される領域、それに、他のモデルによる影響を受けない領域があるのだ。この2つのパターンの違いは、意思決定と開発プロセスにある。共有カーネルが、密接に協力し合う2つのチームによる共同作業であるとすれば、順応者は共同作業に興味がないチームとの統合を扱うものだ。

ここまで、境界づけられたコンテキスト同士の統合における、さまざまな共同作業のやり方を見てきた。非常に協力的な共有カーネルや顧客／供給者開発チームから、一方的な順応者までさまざまなものがある。次の最終ステップでは、関係性に対する、より悲観的な見方へと進む。ここでは、向こう側からは協力も得られなければ、役に立つ設計も得られないと想定される…。

腐敗防止層（ANTICORRUPTION LAYER）



新しいシステムというものは、ほぼいつでも、独自のモデルを持ったレガシーシステムや他のシステムと統合しなければならない。変換層はシンプルで、エレガントなことさえあるが、それはうまく設計された境界づけられたコンテキスト同士を協力的なチームで橋渡しする場合である。しかし、境界の向こう側が流れ込んでくると、変換層はより防御的な色合いを帯びるようになるだろう。

＊　＊　＊

新しいシステムを構築していて、それが他のシステムと接する巨大なインタフェースを持たなければならないという場合、2つのモデルを関係づけることが難しいと、最終的にはその新しいモデルの意図は完全に圧倒されてしまい、結果として、もう1つのシステムのモデルに似せようとその場しのぎで修正せざるを得なくなるかもしれない。レガシーシステムのモデルは貧弱であるのが普通だが、例外的にうまく開発されていても、現在のプロジェクトの要求には合わないかもしれない。それでも、統合することには多くの価値が見出されるかもしれないし、それが絶対的な要件になっていることもある。

これに対する答えは、他システムとの統合をすべて回避することではない。私がかつて関わったプロジェクトでは、すべてのレガシーを置き換えようと熱狂的な取り組みが行われていたこともあったが、それは一度にこなせるものではない。しかも、既存のシステムとの統合は、価値のある再利用の一形態である。巨大プロジェクトの場合、1つのサブシステムが、それとは独立して開発された複数のサブシステムとのインタフェースを持たなければならないこともある。これらのサブシステムには、問題ドメインが異なったかたち

で反映されることになるだろう。さまざまなモデルに基づくシステムが組み合わされると、新しいシステムを他システムのセマンティクスに適合させる必要があることから、新しいシステムのモデル自体が崩壊しかねない。他システムがうまく設計されていたとしても、クライアントと**同じ**モデルに基づいてはいない。それに、他システムはうまく設計されていないことが多いのである。

外部システムとのインタフェースには多くの障害がある。例えば、インフラストラクチャ層は、プラットフォームが異なっていたり、異なるプロトコルを使用していたりするかもしれない他システムとの通信手段を提供しなければならないのだ。他のシステムのデータ型は、新しいシステムのデータ型に変換しなければならない。見過ごされがちなのは、他のシステムが同じ概念ドメインモデルを使用していないという可能性である。

あるシステムからデータを取り出し、それを別のシステムで間違って解釈してしまうと、確実にエラーが起きる。データベースが破壊されてしまうことさえあるかもしれない。しかし、それでもこの問題が忍び寄って来てしまうのは、システム間でやり取りしているものがプリミティブなデータであり、その意味にあいまいなところがなく、いずれの側でも同じに違いないと我々が考えているからである。この想定はたいてい誤っている。微妙ながら重要な意味の違いは、そのデータが各システムにどう関連づけられるかということから生じる。そして、プリミティブなデータ要素が厳密に同じ意味であっても、他システムへのインタフェースをこのような低レベルで動作させることは、一般的に間違っている。低レベルのインタフェースを使うと、他システムのモデルの持っている、データを説明し、その値と関係性を制約する力が奪われてしまう。また一方で新しいシステムに対しては、自分のモデルの言葉にないプリミティブなデータを解釈するという負担を課すことになる。

必要なのは、別のモデルと接している部品の間で変換できるようにすることにより、異質なモデルの要素を消化しきれずにモデルが崩壊しないようにすることである。

それゆえ：

隔離するためのレイヤを作成することによって、クライアントに対して、独自のドメインモデルの用語で表現された機能を提供すること。この層は、既存のインタフェースを通して他のシステムと通信するので、他のシステムを修正する必要はほとんどないか、全くないこともある。内部的には、このレイヤが必要に応じて、2つのモデル間での変換を両方向に対して行う。

＊ ＊ ＊

2つのシステムを連結する仕組みに関するここまでの説明は、プログラム間やサーバ間でデータをやり取りすることに伴う問題を想起させるかもしれない。技術的な通信の仕組みを組み込むことについては、これから簡潔に述べる。しかし、そういう詳細が腐敗防止層であると混同してはならない。腐敗防止層とは他システムへメッセージを送信する仕組み

ではない。むしろ、あるモデルやプロトコルの持つ概念オブジェクトやアクションを、別のものに変換する仕組みなのだ。

腐敗防止層は、それ自体で複雑なソフトウェアになり得る。次は、腐敗防止層を作る上で、設計上、考慮すべき事項について概観しよう。

腐敗防止層のインタフェースを設計する

腐敗防止層の持つ公開インタフェースは、通常、サービスの集合として現れるが、まれにエンティティのかたちをとることもある。全く新しいレイヤを構築して、そこに2つのシステムのセマンティクスを変換する責務を負わせることにより、他システムのふるまいを再び抽象化し、自システムには、こちらのモデルと一貫性のあるかたちで、サービスと情報を提供できるようになる。我々のモデルにとって、その外部システムを単一コンポーネントとして表現することに意味がないことさえある。我々のモデルから見て一貫した責務を負った、複数のサービス（またはエンティティのこともある）を使用した方がよいかもしれないのだ。

腐敗防止層を実装する

腐敗防止層の設計を構成する方法の1つに、ファサード（FACADE）とアダプタ（ADAPTER）（どちらもGamma他 1995より）および変換サービスを組み合わせ、システム間の通信に通常必要となるコミュニケーションと転送の仕組みと合わせて使う、というものがある。

我々が統合しなければならないシステムが、巨大で複雑なひどいインタフェースを持っていることも多い。これは実装の問題であって、腐敗防止層を使用する動機となった概念モデルの相違の問題ではないが、腐敗防止層を作成しようとすると直面する問題ではある。あるモデルから別のモデルへの変換は（特に一方があいまいな場合）、通信しにくいサブシステムのインタフェースを同時に扱わないとしても、それだけで十分に難しい仕事だ。幸い、まさにそのためにファサードがある。

ファサードとは、サブシステムのための代替インタフェースで、クライアントのためにアクセスを簡略化して、サブシステムを使いやすくする。使用したい他のシステムの機能が正確にわかっているので、その機能へのアクセスを容易で効率的にし、残りを隠蔽するファサードを作成できる。ファサードは基盤となるシステムのモデルを変更しない。他システムのモデルに厳密に従って作成しなければならないのだ。そうしないと、うまくいっても、変換の責務が複数のオブジェクトに拡散して、ファサードに負担をかけることになるし、最悪の場合、他システムにも、こちらの境界づけられたコンテキストにも属さない、別のモデルをもう1つ作成することになる。ファサードは他システムの境界づけられたコン

テキストに属する。こちらの要求に特化した、より親しみやすい外観を提供してくれるだけなのだ。

アダプタとは、ふるまいを実装している側が理解しているものとは別のプロトコルを、クライアントが使えるようにするためのラッパである。クライアントがアダプタにメッセージを送ると、アダプタはそれをセマンティクス上で同等なメッセージに変換して、「アダプトされる側（adaptee）」へ送る。応答も変換されて送信者に返される。ここでの**アダプタ**という用語の使い方はあまり厳密ではない。『デザインパターン』（Gamma他 1995）で強調されていたのが、ラップされたオブジェクトをクライアントが期待する標準インタフェースに従わせるということだったのに対して、ここでは適合（アダプト）したインタフェースを選ぶことができる上、適合される側はおそらくオブジェクトでさえないからである。我々が強調しているのは、2つのモデル間で行われる変換だが、私はこれもアダプタの意図と一致していると考えている。

定義するサービスそれぞれに対して、そのインタフェースをサポートし、他のシステムやそのファサードに対して同等のリクエストをどうやって作成するのかを知っているアダプタが必要である。

残る要素は変換サービスだ。アダプタの仕事はリクエストをどうやって作成するかを知ることだったが、概念オブジェクトやデータを実際に変換するのは、それとは別の複雑な作業で、独自のオブジェクトによって行われる。そうすることで、両者がはるかに理解しやすくなる。変換サービスは、必要な時にインスタンス化される軽量オブジェクトでもよい。状態を持たせる必要も分散させる必要もないが、それは変換サービスがアダプタに属するからである。

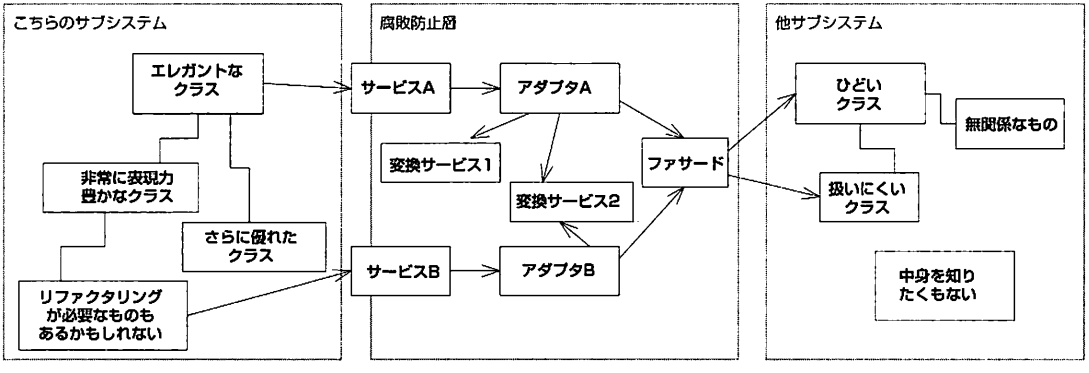


図14.8 **腐敗防止層**の構造

これらは、私が腐敗防止層を作成するのに使用する基本要素である。他にも考慮すべきことがいくつかある。

- 一般的には、図14.8が示唆するように、設計中のシステム（こちらのサブシステム）がアクションを開始する。しかし、他のサブシステムがこちらのサブシステムに何かを要求したり、何かイベントを通知したりしなければならないこともある。腐敗防止層は双方向に向けることができ、両方のインタフェースにそれぞれ独自のアダプタを用いてサービスを定義する。その際に、対称的な変換を行う同一の変換サービスを使用してもよい。腐敗防止層を実装する上で、他のサブシステムを変更する必要はないのが普通だが、他のシステムに腐敗防止層のサービスを呼び出させるために変更が必要となることもある。
- 2つのサブシステムを接続するには、通常は通信の仕組みが必要だが、そのサブシステムは別々のサーバにあるかもしれない。この場合、通信リンクをどこに置くかを決めなければならない。他のサブシステムに対してアクセスできなければ、ファサードと他のサブシステムとの間にリンクを置かなければならないだろう。しかし、ファサードを他のサブシステムと直接統合できるなら、アダプタとファサードの間に通信リンクを置くのが適切な選択肢となる。ファサードのプロトコルは、通信リンクが対応しているプロトコルよりも単純だと思われるからである。腐敗防止層全体を他のサブシステムと同居させることができる場合もあるが、それなら通信リンクや分散機構は、こちらのシステムと腐敗防止層のインタフェースを構成するサービスとの間に配置すればよい。こうしたことは実装やデプロイに関する意思決定であり、現実に即して行うべきである。腐敗防止層が果たす概念上の役割とは何ら関係がない。
- 相手のサブシステムを見ることができた場合、向こう側を少しリファクタリングすることで、こちらの仕事が楽になることに気づくことがある。特に、使おうとしている機能に対する、より明示的なインタフェースの作成を試みて、もし可能であれば自動化されたテストから始めるとよい。
- 統合の要件が広範囲に及んでいると、変換のコストが格段に上昇する。設計中のシステムのモデルを、外部システムに近づけておくという選択をする必要があるだろう。それで変換が容易になる。ただし、きわめて慎重に行い、モデルの整合性で妥協してはならない。変換が難しくて手に負えなくなった時にのみ、選択的に実行すべきだ。その問題の重要な部分のほとんどにおいて、このアプローチが最も自然な解決策と思われるならば、こちらのサブシステムを順応者パターンにして、変換を取り除くことを検討すること。
- 他のサブシステムがシンプルであるか、インタフェースが整理されている場合、ファサードは必要ないかもしれない。

- **2つのサブシステムの関係に特有な機能**であれば、それを腐敗防止層に追加してもよい。外部システムの使用に対する監査証跡や、他のインタフェースへの呼び出しをデバッグする追跡ロジックの2つが、有用な機能としてまず思い浮かぶ。

腐敗防止層が2つの境界づけられたコンテキストを連結する手段であるということを覚えておくこと。普通は誰か他の人が作成したシステムが想定される。そのシステムに関する理解は不完全であり、コントロールすることはほとんどできない。しかし、サブシステムの間に少し詰め物をする必要が生じる状況は、これだけではない。自分で設計した2つのサブシステムであっても、別々のモデルに基づいている場合には、それらを腐敗防止層でつなぐことに意味があるのだ。そういう場合であれば、おそらくはどちら側も完全にコントロールすることができ、たいていは単純な変換層を使うことができるだろう。しかし、2つの境界づけられたコンテキストが別々の道に進みながら、やはり機能的な統合が多少必要になる場合は、腐敗防止層によって両者の間の摩擦を減らすことができる。

例 14.4——レガシー予約アプリケーション

最初のリリースを小規模に素早く行うために、我々が書くことになるのは、輸送を準備し、予約と支援業務用の変換層を通じて、それをレガシーシステムに渡すという、最小限のアプリケーションだ。我々がこの変換層を構築したのは、特に、レガシーの設計の影響から開発するモデルを保護するためだったので、この変換は腐敗防止層である。

最初のリリースでは、腐敗防止層は輸送を表すオブジェクトを受け取り、それを変換してレガシーシステムに渡し、予約を要求して、その後で確認要求を受け取り、新しい設計における確認オブジェクトに変換する。この分離によって、新しいアプリケーションを旧アプリケーションからほぼ独立して開発できるようになるが、変換に相当な投資をしなければならない。

その後に続くリリースのたびに、新しいシステムは、レガシーシステムにあった機能をさらに引き継ぐことも、あるいは既存の機能を置き換えずに新しい価値を追加するだけに留めることもできる。これは後々の意思決定次第である。この柔軟性と、組み合わされたシステムを継続的に運用しながら徐々に移行できることを考えれば、腐敗防止層を構築するための出費に見合う価値があるはずだ。

訓 話

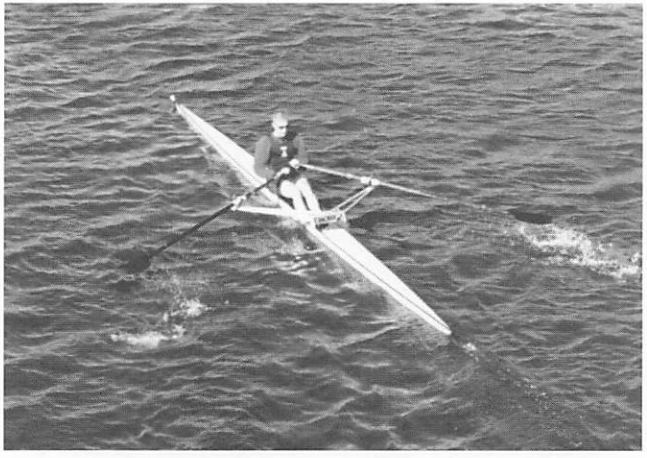
古代中国人は、近隣の遊牧騎馬民族による襲撃から国境を守るために万里の長城を築いた。だが、誰も通れない防壁だったわけではなく、隣接地域との規制された交易は認めつつ、一方で侵略やその他の望ましくない影響に対する障害物となっていたのだ。2千年の間、万里の長城は国境を定義し、中国の農耕文明が外部の混沌にそれほど妨害されずに定

着する上で役に立った。

万里の長城がなければ、中国の文化はこれほど独特なものにはならなかったかもしれないが、壁の建設には膨大な費用がかかり、少なくとも1つの王朝を破綻させ、おそらく没落の一因にもなった。孤立させるという戦略が持つ利点は、そのコストとつり合わなければならない。現実的になってモデルを慎重に修正し、異質なモデルとよりスムーズに適合できるようにすべき時があるのだ。

どんな統合にも、オーバーヘッドはつきものである。単一の境界づけられたコンテキスト内で徹底して継続的な統合を行う場合から、共有カーネルや顧客／供給者開発者チームといった、それほど深く関わり合わないものを経て、一方通行の順応者や腐敗防止層という防御的な態度に至るまで、オーバーヘッドは存在する。統合は非常に有益かもしれないが、必ず高くつくのだ。それが本当に必要かは、確かめなければならない…。

別々の道（SEPARATE WAYS）



要件は容赦なく調べ尽くさなければならない。2つの機能の集合が互いにとって不可欠でないのなら、切り離すことができる。

＊　＊　＊

統合は常に高くつくが、それによる利益は小さいこともある。

チーム間を調整するという通常のコストに加え、統合するには妥協が避けられない。特定の要求を満たすためのシンプルで特化したモデルは、あらゆる状況に対処できる、より抽象的なモデルに道を譲らなければならない。全く異なる技術を使えば、特定の機能は非常に簡単に提供できるかもしれないが、統合するのは難しい。他のチームが共同作業をしようとやってきた際に、あるチームでは、うまくやっていくのが非常に困難で、何も機能しないかもしれない。

多くの状況では、統合することで特別な利益が得られることはない。機能を実現する2つの部品が、互いの機能を呼び出したり、両方が扱うオブジェクトとの相互作用を必要としたり、あるいは操作が行われている間にデータを共有したりしないのであれば、変換層経由であっても統合は必要ないかもしれない。ユースケース上で機能が関連しているという理由だけでは、統合が必須であるということにはならないのだ。

それゆえ：

境界づけられたコンテキストを他とは一切つながりを持たないものと宣言し、開発者がその小さいスコープ内で、シンプルで特化した解決策を見つけられるようにすること。

この場合も、それぞれの機能をミドルウェアやユーザインタフェース層においてまとめ

ることはできる。しかし、ロジックを共有することはないし、変換層を通じてのデータ転送は最低限に抑えるか、できればない方が望ましい。

例 14.5 ——保険プロジェクトの縮小化

あるプロジェクトチームが、保険金請求のための新しいソフトウェアの開発に乗り出した。このソフトウェアは、顧客サービスの代理人や保険査定人が必要とするものをすべて1つのシステムに統合しようとしていた。1年間努力した後で、チームメンバは行き詰まってしまった。分析麻痺（analysis paralysis）とインフラストラクチャに対する大規模な先行投資が相まって、徐々にいらだってきた経営陣に何も見せられなくなったのである。さらに深刻なことに、自分たちのやろうとしていることのスコープに、彼らは圧倒されていた。

新しいプロジェクトマネージャは、1週間の間、彼らを部屋に閉じ込めて、新たな計画を練った。まず、要件の一覧を作成し、その難易度を見積もって、重要度を割り当てようとした。難しくて重要でないものは、容赦なく切り捨てていった。次に、残りの一覧に順番をつけ始めた。その週、その部屋では多くの賢明な決断が下されたが、最終的に重要だと判明したのはたった1つだけだった。どこかの時点で認識されていたのは、**統合してもほとんど付加価値を生まない機能があるということだった。**例えば、保険査定人は既存のデータベースにいくつかアクセスする必要があり、当時のアクセスは非常に不便だった。**しかし、ユーザがこのデータを必要としたにもかかわらず、提案されたソフトウェアシステムにある他のどの機能もそのデータを使用することはなかった。**

チームメンバはアクセスを簡単にするさまざまな方法を提案した。あるケースでは、主要な帳票をHTMLとしてエクスポートし、イントラネットに掲載できるようにした。他のケースでは、標準ソフトウェアパッケージを使って作成した特別な問い合わせを保険査定人に提供できるようにした。これらの機能は、イントラネットページでリンクを構成するか、ユーザのデスクトップにボタンを配置することによって、すべて統合できた。

チームは、小さいプロジェクト一式を立ち上げたが、これは同一メニューから起動するということ以上には、統合を試みないというものだった。いくつかの価値ある機能がほぼ一晩のうちに仕上がった。そういう本質とは関係ない機能の負担をなくしたことで、蒸留された要求が一式残り、しばらくの間は、その要求を満たすメインアプリケーションが完成するのではないかと思われた。

そうなる可能性もあったのだが、残念なことに、チームは元の習慣に戻ってしまった。再び麻痺してしまったのである。結局、彼らが残したものは、別々の道に進んだ複数の小さいアプリケーションだけということになった。

* * *

別々の道に進むと、いくつかの選択肢が閉ざされる。継続的なリファクタリングによって、最終的にはどんな決定も取り消すことはできるが、完全に分離して開発した複数のモデルをマージさせるのは困難である。結局統合が必要であると判明した場合には、変換層が必要となり、複雑になるかもしれない。もちろん、これらにはいずれにしても直面することになる。

さて、より協力的な関係に戻って、今度は統合を拡大する方法を見ていこう…。

公開ホストサービス（OPEN HOST SERVICE）

一般的には、各境界づけられたコンテキストにおいて、コンテキストの外にある統合すべき各コンポーネントに向けた変換層を定義することになる。統合が1回限りであれば、各外部システム用に変換層を挿入するというアプローチをとることで、最低限のコストでモデルの崩壊が回避できる。しかし、サブシステムに対する需要が高いとわかったら、より柔軟なアプローチが必要になるだろう。

＊　＊　＊

あるサブシステムをその他多くのサブシステムと統合しなければならない場合、それぞれに対して特別に変換サービスを作成すると、チームが行き詰まるかもしれない。保守すべきものが次第に大きくなっていき、変更を行う際に注意すべきことも次々と増える。

チームは同じことを何度も行っているかもしれない。そのサブシステムへの要求に一貫性があれば、その部分を、他のサブシステムからの共通する要求を網羅するサービスの集合として記述することもできるかもしれない。

複数のチームが理解して使用できるくらいに整理されたプロトコルを設計するのはかなり困難なので、その苦労が報われるのは、サブシステムのリソースをサービスの凝集した集合として記述できる場合と、相当な数の統合がある場合だけだ。そうした状況であれば、保守と継続的な開発を区別できる。

それゆえ：

こちらのサブシステムにアクセスできるようにするプロトコルを、サービスの集合として定義すること。そのプロトコルを公開し、こちらのサブシステムと統合する必要のある人が全員使用できるようにすること。新しい統合の要件に対応する際には、プロトコルに機能を追加し、拡張すること。ただし、あるチームだけに特有の要求は別だ。そのような特殊なケースには、1回限りの変換サービスを使用してプロトコルを拡張し、共有プロトコルは単純で一貫性のある状態に保つこと。

＊　＊　＊

通信をこのように形式化することは、モデル語彙を共有するということも暗に示している。この語彙はサービスインタフェースの基礎となるものだ。結果として、他のサブシステムは公開ホストのモデルと結合することになり、他のチームはホストチームが使用する特定の方言を習得せざるを得なくなる。状況によっては、知名度の高い公表された言語を交換可能なモデルとして用いることで、結合度が低くなり、理解が容易になることもある…。

公表された言語（PUBLISHED LANGUAGE）

2つの境界づけられたコンテキスト内にあるモデル同士で変換するには、共通の言語が必要である。

＊ ＊ ＊

2つのドメインモデルを共存させ、その間で情報をやり取りすることが必須であれば、変換プロセス自体が複雑化し、ドキュメント化して理解するのが困難になるかもしれない。新しいシステムを構築している時には、自分たちの新しいモデルが、手に入る中で最高であると思うものだから、直接そのモデルに変換することを考える。しかし、いくつかの古いシステムに機能追加を行い、それらを統合しようとすることもあるだろう。ひどいモデルを2つ渡されて、どちらか1つを選ぶというのは、2つの悪のうちから、ましな方を選ぶようなものかもしれない。

別の状況を考えよう。複数のビジネスが相互に情報交換したい場合、どのようにしているのだろうか？ 一方がもう一方のドメインモデルを採用することを期待するのは、非現実的であるだけでなく、双方にとって望ましくないだろう。ドメインモデルは、ユーザのために問題を解決することを目的として開発される。そういうモデルは他システムとの通信を不必要に複雑化する機能を含んでいることがある。同様に、複数のアプリケーションのうち、どれか1つの基礎になっているモデルをコミュニケーションの媒体として使用すると、新しい要求を満たすための自由な変更ができなくなる。むしろ、継続している伝達する役割をサポートするため、きわめて安定していなければならなくなるのだ。

既存のドメインモデルへの直接的な変換や、既存のドメインモデルからの直接的な変換は、適切な解決策でないことがある。既存のモデルは過度に複雑であったり、うまく分割されていなかったりするかもしれない。おそらくは、ドキュメント化されてもいない。そのモデルをデータ交換用の言語として使ってしまうと、本質的に動きが取れなくなり、新しい開発の要求に対応できなくなる。

公開ホストサービスは、複数の関係者が携わる統合に対して、標準化されたプロトコルを使用する。そのプロトコルが採用するのはシステム間での交換というドメインのモデルだが、そのモデルは統合されるシステムの内部では使われてすらいないかもしれない。ここでさらに踏み込んで、その言語を公表するか、すでに公表されている言語を探す。公表という言葉で私が意味しているのは、その言語を使用することに関心があるコミュニティが、いつでもその言語を入手できる状態にあり、それぞれの解釈が互換性を持つように、十分にドキュメント化されているということである。

最近、電子商取引の世界が新しい技術に対して非常に活気づいている。拡張マークアップ言語（XML: Extensible Markup Language）は、データの交換をはるかに容易にすると期待されている。XMLが持つ非常に有益な特徴に、ドキュメント型定義（DTD: Document Type Definition）やXMLスキーマを使うことで、特殊なドメイン言語を正式に定義できるようにし、データをその形式に変換できるようにする、というものがある。業界における唯一の標準DTDを定義して、化学式情報や遺伝子コード情報を関係者間で伝達できるようにするために、業界グループが形成され始めている。本質的に、これらのグループが作成しているのは、言語定義というかたちをとった、共有されるドメインモデルなのだ。

それゆえ：

必要なドメインの情報をコミュニケーションにおける共通の媒体として表現できる、明確にドキュメント化された共有言語を使用し、必要に応じてその言語への変換と、その言語からの変換を行うこと。

こうした言語は一から作成しなくてもよい。何年も前に私が契約を結んだ企業には、Smalltalkで書かれ、データを格納するのにDB2を使用しているソフトウェアがあった。その企業はDB2ライセンスがないユーザにもソフトウェアを配布できるようにする柔軟性を求めており、Btrieveに対するインタフェースを構築するよう私に依頼したのである。Btrieveは軽量データベースエンジンで、無料のランタイム配布ライセンスがついていた。Btrieveは完全な関係データベースではないが、この顧客はDB2が持つ機能のごく一部を使用しているだけで、それは2つのデータベースの最大公約数に収まっていた。この企業の開発者は、DB2の上にオブジェクトの格納に関する抽象を構築していた。私は、Btrieveコンポーネントのインタフェースとして、これを使用することにした。

このアプローチは確かにうまくいき、ソフトウェアは顧客のシステムとスムーズに統合できた。ただし、顧客の設計にあった永続オブジェクトの抽象に関して、正式な仕様もドキュメントもなかったため、新しいコンポーネントに対する要件を把握するのに多くの作業が必要となった。また、DB2からBtrieveへ他のアプリケーションを移行するにあたり、そのコンポーネントを再利用する機会もあまりなかった。新しいソフトウェアは企業の永続化モデルにより深く入り込んでいたので、永続オブジェクトのモデルをリファクタリングしようと思うと、より難しかったことだろう。

よりよい方法は、企業が使用しているDB2インタフェースのサブセットを識別し、それをサポートすることだったかもしれない。DB2のインタフェースは、SQLと数多くの独自プロトコルで構成されている。非常に複雑ではあるが、このインタフェースは厳密に定義され、徹底的にドキュメント化されている。ただ、使われていたのはインタフェースの小さいサブセットだけだったので、複雑さは緩和されていただろう。DB2インタフェースのうち、必要なサブセットをエミュレートするコンポーネントが開発されていたら、ただサ

ブセットを識別するだけで、開発者向けにきわめて効果的にドキュメント化できただろう。統合対象のアプリケーションはDB2とやり取りする方法を知っていたから、追加の作業はほとんど必要なかった。ただ、将来、永続化層を再設計する時には、機能拡充の前と同様、DB2のサブセットだけを使うように制限されることになるだろう。

DB2インタフェースは公表された言語の一例である。この場合、2つのモデルはビジネスドメインの中にないが、それでもすべての原則は同じように当てはまる。協調するモデルの一方がすでに公表された言語であるため、第3の言語を導入する必要もない。

例14.6——化学のための公表された言語

業界や学会では、化学式のカタログ、分析、および操作のために、無数のプログラムが使用されている。データの交換は常に難しかったが、それは、ほぼすべてのプログラムが化学構造を表現するのに別々のモデルを使用しているからである。そしてもちろん、そのほとんどはFORTRANのような言語で書かれていて、いずれにしてもドメインモデルを完全には表現しない。データを共有したいと思えば、他システムの持つデータベースの詳細をひも解いて、何らかの変換スキームを考案しなければならないのが常だった。

そこに現れたのが化学マークアップ言語（CML: Chemical Markup Language）というXMLの方言である。CMLはこのドメインの共通交換言語となるよう意図されたもので、学会と業界を代表するグループによって開発され、管理されている（Murray-Rust他 1995）。

化学情報は非常に複雑かつ多様で、新発見があると常に変化する。そこで開発されたのが、有機分子と無機分子の化学式や、タンパク質のアミノ酸配列、スペクトル、物理量といった基礎を記述できる言語だった。

今や、この言語は公表されているから、1つのデータベースにしか使えなかった頃であれば、作成する手間に見合わなかったであろうツールも開発できるようになった。例えば、JUMBO Browserと呼ばれるJavaアプリケーションが開発された。これはCMLで格納された化学構造のグラフィカルなビューを作成するアプリケーションだ。したがって、データをCMLフォーマットで入力しておけば、そのような視覚化ツールが使えるようになる。

実のところ、XMLという一種の「公表されたメタ言語」を使用することで、CMLは2倍の優位性を得た。CMLの学習曲線は、XMLがよく知られるようになったことで平坦化した。実装は、パーサなどの多様な既製ツールによって容易になり、XMLを扱うあらゆる側面について書かれた多数の文献により、ドキュメント化も楽になっているのである。

次に示すのは、CMLのちょっとしたサンプルである。専門家ではない私にはかすかにわかる程度だが、原則は明確である。

```
<CML.ARR ID="array3" EL.TYPE=FLOAT NAME="ATOMIC ORBITAL ELECTRON POPULATIONS"
      SIZE=30 GLO.ENT=CML.THE.AOEPOPS>
   1.17947 0.95091 0.97175 1.00000 1.17947 0.95090 0.97174 1.00000
   1.17946 0.98215 0.94049 1.00000 1.17946 0.95091 0.97174 1.00000
   1.17946 0.95091 0.97174 1.00000 1.17946 0.98215 0.94049 1.00000
   0.89789 0.89790 0.89789 0.89789 0.89790 0.89788
</CML.ARR>
```

* * *

象のモデルを統一する

ヒンドゥスタンの6人の男が
象のことを知りたいと
見に出かけた
(全員目が見えなかったが)
触ってみれば
満足できるだろう

最初に象に近づいた者が
うっかり倒れてしまった先にあったのは
大きくがっしりした横腹だったので
すぐに大声でわめきだした
「しまった！ でも、象というものは
まるで壁のようだ」

…

3人目がその動物に近づき、
偶然手にしたのは
もぞもぞ動く鼻だった
そこではっきりと言った
「わかった。象というものは蛇そっくりだ」

4人目がしきりに手を伸ばすと、
ひざに触れて、こう思った。
「このなんとも不思議な獣は
大平原みたいだ」
そこでこう言った、「間違いようがない、
象は木そのものだ！」

…

6人目が獣のいるあたりを
手探りし始めてすぐに、
つかんだのは揺れ動く尻尾で
手の届く範囲にあったものだった
そこでこう言った。「わかった。象は
縄によく似ている！」

こうしてヒンドゥスタンの男たちは
大声で長いこと議論した
それぞれ自分の意見に
頑固にこだわっていたが
それぞれ一部正しいだけで
全員が間違っていた！

——ヒンドゥー教テキストのUdana所収の話に基づく
John Godfrey Saxe（1816-1887）の『盲人と象』より

* * *

●例 14.6——化学のための公表された冗語

象と触れ合うことの目標が何であるかによっては、盲目の男たちはまだ先に進めただろう。象の本質に関して、完全に意見が一致することがなかったとしても、である。統合する必要がなければ、モデルが統一されなくても問題にはならない。何らかの統合が必要であれば、象が何であるかに関して実際に合意しなくてもよいかもしれず、ただ意見が一致していないということを認識するだけで多くの価値が得られる。そうすれば、少なくとも知らずに話が食い違うことはなくなるのだ。

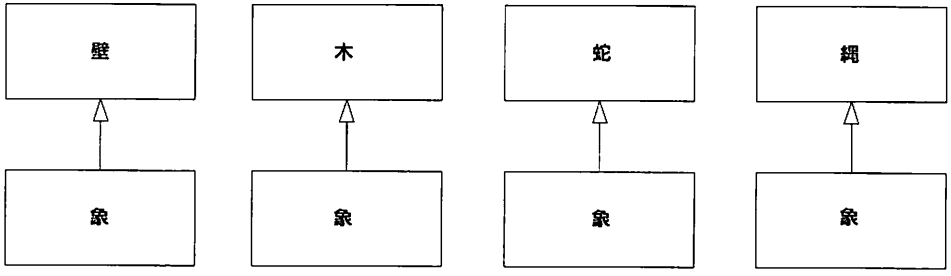
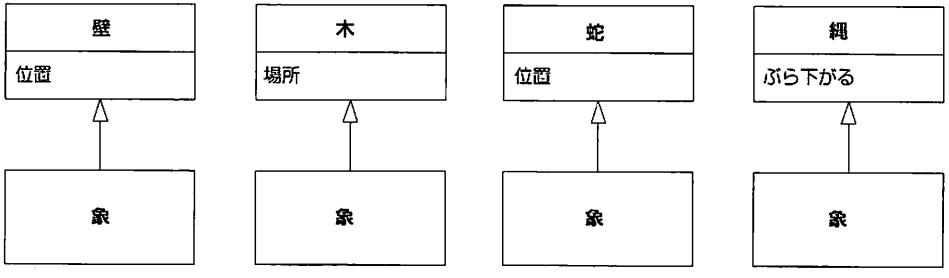


図14.9 4つのコンテキスト：統合なし

図14.9にあるのは、盲目の男たちが象について形成したモデルをUMLで表現したものである。独立した境界づけられたコンテキストを確立すれば、状況が十分明確になり、彼らが共通して気にかけている側面について、互いにコミュニケーションする方法を考え出せる。その側面とは、おそらく象に触れた位置だろう。



変換：{壁.位置 <-> 木.場所 <-> 蛇.位置 <-> 縄.ぶら下がる}

図14.10 4つのコンテキスト：最小限の統合

盲目の男たちが象に関する情報をもっと共有したいと思うにつれ、単一の境界づけられたコンテキストを共有する重要性は増す。しかし、完全に異なるモデルを統一するのは難

題である。誰も自分のモデルを断念して他の人のモデルを採用しそうにない。結局、尻尾に触れた者は象が木に似ていないと**知っている**から、そのモデルは、彼にとっては無意味で役に立たないのだ。複数のモデルを統一することは、ほぼ常に新しいモデルを作成することを意味する。

いくらかの想像力をもって議論（おそらく激論）を継続することで、盲目の男たちは、大きな全体のさまざまな部分を記述してモデル化していたことに、最終的には気づけるだろう。多くの目的にとって、部分と全体の統一はそれほど追加作業を必要としないかもしれない。少なくとも、統合の第1段階で必要なのは、各部分がどう関連しているかを把握することだけだ。求められているものによっては、象を壁と見なし、それが木の幹に支えられていて、一方に縄、もう一方に蛇がついていると考えれば十分かもしれない。

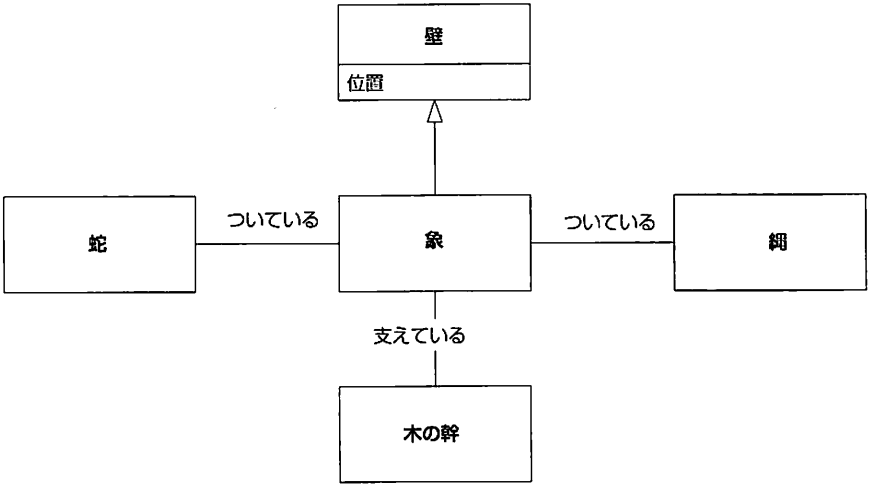


図14.11　1つのコンテキスト：荒削りな統合

さまざまな象のモデルを統一するのは、たいていのこうしたマージよりも容易である。残念ながら、2つのモデルが全体の中で別々の部分を純粋に記述しているというのは、違いの1側面ではあるが、例外的なものだ。2つのモデルが同じ部分に対して別々の見方をしていると、問題はさらに難しくなる。2人の男が鼻に触れ、1人が蛇と言い、もう1人が消防ホースと言っていたとしたら、さらに厄介になっていたはずである。どちらも相手のモデルを受け入れられないが、それは自身の経験と矛盾するからである。実際、2人に必要なのは新しい抽象だ。それは蛇の「生きているという性質」と消防ホースの放水機能を包含させながら、最初のモデルにある不適切な含意を取り除いたものである。取り除くべき不適切な含意には、毒牙かもしれないという予想や本体から取り外して消防車のタンクに巻

き上げて収納できるといったことがある。

部分を全体に結合させたとしても、結果としてできるモデルは荒削りなものだ。一貫性がなく、基盤となるドメインの輪郭に従うという感覚が一切ない。継続的な改良のプロセスの中で、新しい洞察が、より深いモデルにつながるかもしれない。新しいアプリケーションの要件も、より深いモデルへの移行を強制する。象が動き出せば、「木」説は通用しなくなるから、盲目のモデラはブレイクスルーによって、「脚」という概念にたどり着くかもしれない。

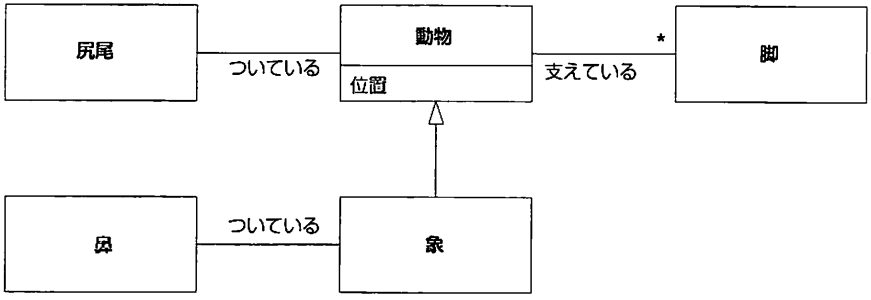


図14.12 1つのコンテキスト：より深いモデル

モデルを統合するにあたり、この2回目の試みでは、個々のモデルが持つ偶発的な側面や不正確な側面から抜け出し、複数の新しい概念を作成する。この場合では、「鼻」、「脚」、「胴体」、「尻尾」を持つ「動物」になる。この概念はそれぞれが、独自の特徴を持っており、他の部分に明確に関連づけられている。モデルの統一が成功するかどうかは、大体において、ミニマリズムが条件になる。象の鼻にあって蛇にないものも、蛇にあって象の鼻にないものも存在するが、おそらくは蛇にないものの方が重要だ。水を噴出する特徴がない方が、毒牙という不適切な特徴があるよりましなのである。

目標が単に象を知ることであれば、各モデルの持つ位置表現の間で変換するだけでよい。さらなる統合が必要であれば、統一されたモデルは最初のバージョンで完全に成熟していなくてもよい。求められているものによっては、象を壁と見なし、それが木の幹に支えられていて、一方に縄、もう一方に蛇がついていると考えれば十分かもしれない。その後、新しい要件や、理解とコミュニケーションの改善によって推進されて、モデルは深められ、改良される。

複数のドメインモデルがぶつかり合っていることを認識することは、実は現実と向き合っているにすぎない。各モデルが適用されるコンテキストを明示的に定義することにより、それぞれのモデルの整合性を維持して、2つのモデル間に作成したいと思うどんな具体的なインタフェースについても、そこに含意されたものを明確に理解できる。盲目の男た

ちが象全体を見ることはできないが、自分たちの認識が不完全であることに気づきさえすれば、問題は処理できるようになるのだ。

モデルコンテキスト戦略を選択する

どんな時でも現状を反映させたコンテキストマップを描くことは、常に重要である。だが、いったん描けたら、その現実を変更したくなるだろう。その時から、コンテキストの境界間の関係を意識的に選択し始めることができるようになる。そのための指針をいくつか挙げよう。

チームでの意思決定と、より上層での意思決定

チームが最初に決定しなければならないのは、境界づけられたコンテキストをどこに定義し、コンテキストの間にどんな種類の関係を持たせるかということだ。そうした意思決定はチームで行うか、少なくとも、チーム全体に伝えて全員に理解させなければならない。実際には、そうした意思決定にはチームより上層の合意が必要となることも多い。検討に際して、境界づけられたコンテキストを拡大するか分割するかを決定する際に基準とすべき費用対効果のトレードオフは、チームが独立して活動することの価値と、直接的で豊かな統合の価値との間で考えるべきである。だが現実には、チーム間の政治的な関係によってシステムをどう統合するかが決まることも多い。技術的には好都合な統一が、報告系統のせいで不可能になるかもしれない。経営陣がやりにくいマージを命令することもある。欲しいものがいつでも手に入るとは限らないが、少なくとも、かかるコストの一部を見積もって伝え、それを軽減する方策を講じることはできるだろう。現状を示すコンテキストマップから始め、変換を選択する上では、現実的になること。

コンテキストに自らの身を置く

ソフトウェアプロジェクトに従事していると、まず興味を持つのは、システムの中で自分のチームが変更している部分（「設計中のシステム」）であり、その次が通信することになるシステムである。典型的なケースでは、設計中のシステムが1、2個の境界づけられたコンテキストとして切り出されて、それに対して主要な開発チームが取り組み、別のコンテキスト1、2個がそのサポート役になる。そこに加わるのが、これらコンテキスト間の関係と、外部システムとの関係である。これは、我々が直面するであろう状況を大まかに予想した、単純で典型的な見方である。

我々は、実は取り組んでいる主要なコンテキストの一部なのであり、それはコンテキストマップに間違いなく反映しなければならない。我々が先入観というものを認識していて、

コンテキストマップが適用できる範囲から外へ出た時のことを忘れなければ、これは問題ではない。

境界を変換する

境界づけられたコンテキストの境界線を描くにあたっては、数えきれない状況と選択肢がある。しかし、たいてい苦労することになるのは、次に挙げるフォースのサブセットの間でバランスを取ることである。

より大きな境界づけられたコンテキストを好む

- 統一されたモデルで処理される量が増えると、ユーザタスク間のフローがスムーズになる。
- 2つの個別のモデルと、そのマッピングよりも、一貫性のあるモデルが1つある方が理解しやすい。
- 2つのモデルの間で行われる変換は難しいかもしれない（不可能な場合もある）。
- 共有された言語により、明確なチームのコミュニケーションが促進される。

より小さな境界づけられたコンテキストを好む

- 開発者間のコミュニケーションオーバーヘッドが減少する。
- 継続的な統合はチームとコードベースが小さい方が容易である。
- 大きなコンテキストは、より用途の広い抽象モデルを必要とするが、そのためのスキルを持っている人は少ない。
- 特殊な要求を満たしたり、特化したユーザグループの専門用語を包括したりするには、別のモデルを作って、ユビキタス言語の特化した方言と合わせて使った方がよい。

別々の境界づけられたコンテキスト間で、機能を深く統合するのは実用的でない。統合は、あるモデルの中でも、他のモデルの用語で厳密に述べることのできる部分に限定されるし、このレベルの統合でも多大な努力を要するかもしれない。ただ、2つのシステム間にあるインタフェースが小さくなるなら、機能を深く統合することにも意味がある。

変更できないものを受け入れる：外部システムの輪郭を描く

最も簡単な意思決定から始めるのがよい。サブシステムの中には、開発中のシステムの境界づけられたコンテキストには明らかに入っていないものもある。その例としては、すぐには置き換える予定のないレガシーシステムや、必要となるサービスを提供する外部システムが挙げられるだろう。これらはすぐに識別できるので、設計から分離する準備ができ

る。

ここでは、自分たちの想定に注意しておかなければならない。そうしたシステムが、それぞれ独自の境界づけられたコンテキストを構成していると考えると都合がよいが、ほとんどの外部システムは、その定義にはわずかしか合致しない。まず、境界づけられたコンテキストは、一定の境界内でモデルを統一するという意図によって定義される。レガシーシステムの保守をコントロールできることがあるかもしれない。その場合は統一するという意図を宣言できるし、あるいはレガシーシステムを担当するチームが協力して、略式の継続的な統合を遂行するかもしれない。しかし、それを当然と思ってはならない。他システムを調査して、開発が十分統合されていない場合には、特に注意すること。そういうシステムのさまざまな場所にセマンティクスの矛盾が見つかることは珍しくないのだ。

外部システムとの関係

ここで適用できるパターンは3つある。1つめは、別々の道を考えることだ。そう、統合が必要なければ、外部システムを含めることはない。ただし、本当にそうなのかを確かめること。ユーザが両方のシステムに簡単にアクセスできるようにすれば十分なのか？ 統合は高くつくし、注意すべき場所が増えるものだから、できる限りプロジェクトの負担を軽くすること。

統合が実際に不可欠なら、順応者と腐敗防止層という両極端から選択できる。順応者になるのは、楽しいものではない。創造性と新しい機能に関する選択肢が限定されてしまう。主要な新システムを構築する際には、レガシーシステムや外部システムのモデルに固執するのは現実的でなさそうだ（そもそも、なぜ新システムを構築しているのか？）。だが、今後も支配的であり続ける巨大なシステムを周辺的に拡張するケースでは、レガシーのモデルにこだわる方が適切なこともある。そうした選択の例としては、Excelやその他の単純なツールで書かれることの多い、軽量な意思決定支援ツールがある。作成するアプリケーションが、実は既存システムの拡張で、そのシステムとのインタフェースが巨大化するのであれば、コンテキスト間の変換は大きな仕事になって、アプリケーションの機能そのものを容易に超えてしまうかもしれない。他システムの境界づけられたコンテキスト内に身を置いたとしても、優れた設計を機能させる余地はまだある。他システムの背後に識別可能なドメインモデルがあれば、厳密に旧モデルに追従する限り、そのモデルを旧システムにあった時よりも明示的にすることで、自分の担当する実装を改善できる。順応者として設計することに決めたのであれば、心の底からそうしなければならない。拡張だけに専念して、既存モデルを修正してはならない。

設計中のシステムの機能が既存システムの拡張以上のものになりつつあり、他システムへのインタフェースが小さいか、設計が非常に粗末である場合、自分自身の境界づけられ

たコンテキストが何としても欲しくなる。これが意味しているのは、変換層か、場合によっては腐敗防止層を構築するということだ。

設計中のシステム

プロジェクトチームが実際に構築しているソフトウェアが、ここで言う**設計中のシステム**である。この領域内に複数の境界づけられたコンテキストを宣言して、それぞれの統一性を維持するために、継続的な統合を適用できる。しかし、そうしたコンテキストはいくつ必要だろう？ お互いの関係はどんなものにすべきか？ その答えは外部システムの場合ほど型にはまったものにならない。より自由で、コントロールできる範囲も広いからだ。

きわめて単純にすることもできる。設計中のシステム全体を、1つの境界づけられたコンテキストにするのだ。例えば、10人未満のチームで相互の関連の非常に大きい機能に取り組んでいるとすると、これはありがちな選択肢である。

チームが大きくなるにつれ、継続的な統合は困難になるかもしれない（ただ、ある程度大人数のチームでも維持されるのを見たことはある）。共有カーネルを探して、比較的独立した機能のまとまりを別々の境界づけられたコンテキストに切り離し、それぞれを10人未満で担当すればよいだろう。境界づけられたコンテキストの間にあるすべての依存関係が一方向であれば、顧客／供給者の開発チームを準備すればよい。

2つのグループの考え方があまりに異なっているせいで、モデリング作業が常に衝突していることに気づくことがある。両者がモデルに対して求めているものが、実は全く異なっているのかもしれないし、知識の背景が単に異なっているだけのこともあれば、プロジェクトが組み込まれている管理系統の結果であることもある。衝突の原因が変更できないものであったり、あるいは変更したくないものであったりした場合には、モデルを別々の道に進ませてもよい。統合が必要なところでは、2チームで協力して変換層の開発と保守を行い、そこを継続的な統合が起こる唯一の場所とすればよい。これとは対照的なのが外部システムとの統合で、その場合には、通常腐敗防止層を用いて、他システムのそのままの状態に、相手側のサポートを受けることなく対応しなければならない。

一般的に言って、境界づけられたコンテキスト1つに対して、1つのチームが対応する。1つのチームで複数の境界づけられたコンテキストを保守することもできるが、複数のチームが1つの境界づけられたコンテキストに一緒に取り組むのは（不可能とは言わないまでも）難しい。

別のモデルで特殊な要求を満たす

同じビジネスに関わるさまざまなグループが、独自の専門的な用語法を発達させることもあるし、それは互いに異なっているかもしれない。それらの局所的な専門用語は、非常

に正確であり、使う人の要求に合わせてあつらえられているだろう。そうした用語を変更する（例えば、標準化された全社的な用語法を押しつける）には、そこにある違いを解消するために大々的なトレーニングと分析が必要になる。そうしたとしても、新しい用語法は、すでに存在する細かく調整されたものほどには役に立たないかもしれない。

そういう特殊な要求には別の境界づけられたコンテキストで対応することにして、変換層の継続的な統合を除き、モデルに別々の道を進ませてもよい。ユビキタス言語のさまざまな方言が、それらのモデルと、基になっている特化した専門用語を取り巻いて進化する。2つの方言で重なり合うところが多ければ、共有カーネルを用いることで、求められる特殊性を提供しつつ、変換のコストを最小限にすることができる。

統合が必要なかったり、比較的限定的であったりしたら、別のモデルにすることで今までの用語法が引き続き使えるようになり、モデルが崩壊するのを避けられる。だが、これにはコストとリスクも伴う。

- 共有された言語がなくなることで、コミュニケーションが減る。
- 統合に余分なオーバーヘッドがかかる。
- 同一のビジネスの活動とエンティティに対する別々のモデルが進化するにつれ、作業の重複が発生する。

しかし、おそらく最大のリスクは、この考え方に従うと、変更することに反対して突飛で偏狭なモデルを正当化することになりかねないということである。特殊な要求を満たすために、システムの個別の部分をどれくらい調整しなければならないだろうか？ 何より重要なことだが、**そのユーザグループ特有の専門用語にどれほどの価値があるのか？** チームが独立して活動できることの価値と変換のリスクを秤にかけて、価値のない用語法のバリエーションを正当化しないように目を光らせなければならない。

時には深いモデルが出現して、こうした別々の言語を統一して、どちらのグループも満足させられることもある。ただ注意すべきなのは、深いモデルが出現するのはライフサイクルの後半で、出現するとしても多くの開発と知識のかみ砕きを経た後だということだ。深いモデルをあらかじめ計画することはできない。ただ、機会が訪れた時に受け入れて、戦略を変更してリファクタリングするしかない。

覚えておいて欲しいのだが、統合の要求が広範囲にわたると、変換のコストは急上昇する。複雑な変換を行うオブジェクトを1つピンポイントで修正する場合から、共有カーネルに至るまで、チーム間の調整を行うことによって、完全な統一を必要としなくても、容易に変換できるようになる。

デプロイ

複雑なシステムのパッケージングとデプロイを調整することは、ほぼ常に、見た目よりはるかに辛い退屈な作業の1つである。境界づけられたコンテキスト戦略の中でどれを選択するかは、デプロイに影響を及ぼす。例えば、顧客／供給者チームが新しいバージョンをデプロイする際には、一緒にテストされたバージョンをリリースするために互いに調整しなければならない。コードとデータの移行はまとめて行わなければならない。そうすることによって、分散システムにおいては、コンテキスト間の変換層を、単一のプロセス内に一緒にしておき、複数のバージョンを共存させる必要がなくなる。

単一の境界づけられたコンテキスト内にあるコンポーネントをデプロイするのであっても、データ移行に時間がかかったり、分散システムを瞬時に更新できなかったりするせいで、結果的にコードとデータの2つのバージョンが共存する場合には、困難かもしれない。

デプロイ先の環境と技術によっては、技術的な考慮すべき事項が多く生じてくる。しかし、境界づけられたコンテキストの関係を見れば、危険な場所がわかる。変換インタフェースは境界で仕切られているのだ。

デプロイ計画の現実性は、コンテキスト境界の引き方にフィードバックすべきである。2つのコンテキストが変換層によって橋渡しされている場合、一方のコンテキストを更新したら、すぐに新しい変換層がもう一方のコンテキストに同じインタフェースを提供できる。共有カーネルがあると、開発だけでなくデプロイについても、調整に課される負担ははるかに大きくなる。別々の道を進むことで、作業は大幅に単純化できる。

トレードオフ

これまでの指針を要約すると、モデルを統一したり統合したりする上では幅広い戦略がある。一般的な言い方をすると、機能のシームレスな統合による利益と、調整やコミュニケーションという追加作業との間でトレードオフを考えることになるのだ。より独立した活動をとるか、よりスムーズなコミュニケーションをとるかである。より大掛かりな統一を行うには、関連するサブシステムの設計をコントロールしなければならない。

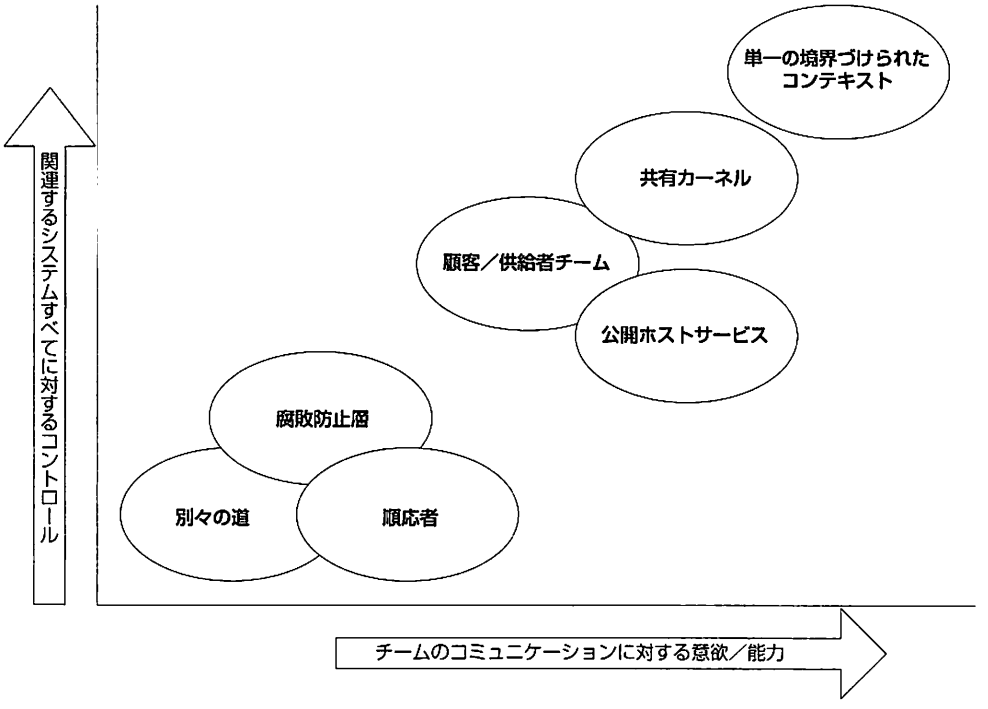


図14.13 **コンテキスト**間の関係についてのパターンの相対的要望

すでにプロジェクトが進行中の場合

プロジェクトを開始するのではなく、すでに進行中のプロジェクトを改善しようとする立場に立つことの方が多いだろう。その場合、最初にやるべきことは、**現在の状況に従って**境界づけられたコンテキストを定義することである。これは非常に重要だ。効果的に行うためにコンテキストマップに反映するべきなのは、チームが実際に行っていることであり、前述した指針に従って決定される理想的な構成**ではない**。

現在の境界づけられたコンテキストについて本当の輪郭を描き、現在の関係性を記述したら、次にやるべきなのは、その**現在の編成**を中心として、チームが実際に行っていることを統制することである。コンテキスト内で行われる継続的な統合を改善すること。どこかに迷い込んだ変換コードがあれば、腐敗防止層へリファクタリングすること。既存の境界づけられたコンテキストには名前をつけ、プロジェクトのユビキタス言語に確実に入っているようにすること。

これで、境界とその関係性に対する変更を検討する準備ができた。この変更は、当然、新規プロジェクトについてすでに述べてきたものと同じ原則によって推進される。しかし、

細かく砕いた上で、努力と混乱を最小限にして最大の価値が得られるよう、現実に即して選ばなければならない。

次節では、いったん決定した境界づけられたコンテキストに対する実際の変更をどう行うかについて議論する。

変 換

モデリングや設計が持つ他の側面と同様、境界づけられたコンテキストに関する意思決定は、取り消せないものではない。実際、境界と境界づけられたコンテキスト間の関係について、初期の決定を変更しなければならないケースは多々ある。一般的に言って、コンテキストを分割するのは実に容易だが、マージしたり、関係を変更したりするのは難しい。困難ではあるが重要な変更のうち、代表的なものをいくつか述べることにする。こういう変換は大きすぎて、1回のリファクタリングや、プロジェクトにおける1回のイテレーションでは、処理しきれないのが普通だ。そこで、変換を一連の管理可能なステップにするための作戦について概観する。これらの指針は当然、あなたが具体的な状況と出来事に合わせて用いていかなければならない。

コンテキストをマージする：別々の道 → 共有カーネル

変換のオーバーヘッドがあまりに大きい。重複があまりにもはっきりしている。このように、境界づけられたコンテキストをマージする動機は数多くある。しかし、実行するのは難しい。手遅れということはないが、根気は必要だ。

最終的な目標が、継続的な統合を伴う単一のコンテキストに完全にマージすることであっても、共有カーネルへ移るところから始めること。

1. 初期の状況を評価すること。2つのコンテキストを相互に統一し始める前に、それぞれが実際に内部で統一されていることを確認すること。
2. プロセスを準備すること。コードがどのように共有され、モジュールの命名規則がどうなるかを決定する必要がある。少なくとも、1週間に一度は共有カーネルのコードを統合しなければならない。また、テストスイートも欠かせない。共有されるコードを開発する前に、テストを準備しておくこと（テストスイートは空だから、テストは容易に通るはずだ！）。
3. 着手するにあたって、小さいサブドメインを選択すること。両方のコンテキストで重複するが、コアドメインの一部**ではない**ものが望ましい。この最初のマージによってプロセスが確立されるので、シンプルで比較的汎用的なものか、重要でないものを使うのが最適である。すでに存在する統合と変換をよく調べること。すでに変換されて

いるものを選択すると、実績のある変換を用いて開始できるという利点があるのに加え、変換層を薄くすることにもなる。

この時点で、同一のサブドメインに対して2つのモデルが存在することになる。基本的に、マージには3つのアプローチがある。モデルを1つ選び、もう一方はそれと矛盾しないようにリファクタリングすることができる。この決定は大規模にわたるかもしれない。あるコンテキストのモデルを組織的に置き換え、1つの単位として開発されたモデルの一貫性を維持しようとするのだ。あるいは、一度に一部分ずつ選んでマージしていき、最終的にはどちらのモデルも最高のものにすることもできる（ただし、結局ごちゃ混ぜにならないように注意すること）。

3番目の選択肢は新しいモデルを見つけることだ。このモデルは、もともとあるモデルのどちらよりも深くて、両方の責務を引き受ける能力がある。

4. サブドメインの共有モデルを考案するため、両チームから集めた2～4人の開発者からなるグループを作ること。モデルの由来に関わらず、詳細にわたって精査しなければならない。これは、同義語を特定して、まだ変換されていない用語をマッピングするという重労働を伴う。この合同チームは、モデルのための基本的なテスト一式の概観を示す。
5. どのチームに属していた開発者も、このモデルを実装する（または既存のコードを適合して、共有できるようにする）作業を担い、詳細を考え出して、それを機能させる。この開発者たちがモデルに関する問題に行き当たったら、ステップ3からチームを再召集し、概念の必要な改訂すべてに関与させる。
6. 各チームの開発者は新しい共有カーネルによる統合の作業を引き受ける。
7. もはや必要なくなった変換を削除すること。

ここまで来れば、非常に小さい共有カーネルができていて、それを保守するプロセスが整う。プロジェクトにおける以降のイテレーションでは、ステップ3～7を繰り返して共有する内容を増やしていく。プロセスが安定し、チームが自信をつけるにつれ、より複雑なサブドメインや、同時に複数のドメイン、またはコアドメイン内にあるサブドメインを引き受けられるようになる。

注：モデルにおいて、ドメインにより特化した部分を引き受けるようになると、別々のユーザコミュニティの持つ特殊な専門用語に対して、2つのモデルが順応しているケースに遭遇することがある。これらをマージして共有カーネルにすることは後に回した方が賢明だ。やがて、深いモデルへ向かうブレイクスルーが起こり、それらの特化した言語の両方に取って代わることのできる言語が提供されるだろう。共有カーネルが有利な点は、継続的な統合による利点が得られる一方で、別々の道による利点も失わないことにある。

これらが、マージして共有カーネルにする際の指針である。先へ進む前に、この変換によって対応される要求のうち、いくつかを満たすことのできる代替案を1つ検討しよう。2つあるモデルの一方が確実に望ましい場合、統合しないでそちらに移行することを検討すること。共通のサブドメインを共有する代わりに、それらサブドメインに対するすべての責務を一方の境界づけられたコンテキストからもう一方へと体系的に移行するのだ。これは、望ましい方のコンテキストのモデルを呼び出すようにアプリケーションをリファクタリングして、モデルに必要となる機能をすべて追加することで実現される。こうすれば継続的な統合に伴うオーバーヘッドなしに、冗長なものを取り除いたことになる。将来的には（必ずではないが）、望ましい方の境界づけられたコンテキストが完全に責務を引き継げることになるかもしれず、そうなればマージと同じ効果が作り出されることになる。移行している間は（きわめて長くなるか際限がなくなるかもしれないが）、別々の道を進んだ際に通常得られる利点と欠点があるので、共有カーネルのプラス面とマイナス面を比較して秤にかけなければならない。

コンテキストをマージする：共有カーネル → 継続的な統合

共有カーネルが大きくなってくると、2つの境界づけられたコンテキストを完全に統一することによってもたらされる利点に心を動かされるかもしれない。これは単にモデルの相違を解消するという問題ではない。チーム編成を変えることから始まり、究極的には人が話す言語を変更することになる。

人々とチームとを準備するところから始めること。

1. 継続的な統合に必要なプロセス（コードの所有権の共有や頻繁な統合など）がすべて、**チームごと**にそれぞれ確実に準備されているようにすること。2チームで統合の手続きを一致させ、全員が同じ方法で物事を実行するようにすること。
2. チーム間でメンバのローテーションを開始すること。こうすることで両方のモデルを理解している人をプールでき、2チームのメンバにつながりができ始める。
3. 各モデルの蒸留を個別に明確化すること（第15章参照）。
4. この時点で、コアドメインの共有カーネルへのマージに着手する上での、十分な自信がついているはずだ。この作業には数回のイテレーションが必要かもしれず、新しく共有された部分とまだ共有されていない部分の間に一時的な変換層が必要になることもある。いったんコアドメインのマージに着手したら、迅速に進めた方がよい。これはオーバーヘッドの高いフェーズで、間違いがよく起きるので、できる限り期間を短くし、ほとんどの新規開発よりも優先させるべきである。しかし、扱いきれないほど引き受けてはならない。

コアドメインのモデルをマージするには、いくつかの選択肢がある。一方のモデルに固執して他方をそれと互換性を持つように修正することもできるし、サブドメインの新しいモデルを作成して、それを使用するように両方のコンテキストを適合させることもできる。2つのモデルが別々のユーザの要求に対応するようあつらえられている場合には注意すること。両方のオリジナルのモデルが持つ特別な能力が必要かもしれない。そうなると、両方のオリジナルのモデルに取って代われる、より深いモデルを開発しなければならなくなる。より深い統一されたモデルを開発するのは非常に難しいが、2つのコンテキストを完全にマージすると固く決意しているなら、もはや方言を複数持つという選択肢はない。結果としてできるモデルとコードが明確に統合されているという点で、この苦労は報われるだろう。これによってユーザの特殊な要求に対処する能力を犠牲にすることがないように注意すること。

5. 共有カーネルが大きくなるにつれ、統合の頻度を1日に一度に上げ、最終的には継続的な統合を行うこと。
6. 共有カーネルが、もともとあった2つの境界づけられたコンテキストをすべて包括するようになる頃には、1つの大きいチームができているか、2つの小さいチームがコードベースを共有して継続的な統合を行い、メンバを頻繁に入れ替えているかの、いずれかになっている。

レガシーシステムを段階的に廃止する

よいものにはすべて終わりがあるもので、レガシーのコンピュータソフトウェアも例外ではない。ただし、ひとりでにそうなるわけではない。これらの古いシステムは、ビジネスや他システムにしっかりと織り込まれているかもしれないので、解きほぐすには何年もかかるかもしれない。幸い、一度にすべてを行わなくてもよいのである。

考えられる状況は多岐にわたりすぎて、ここでは表面を軽く眺める程度しかできない。だが、一般的なケースについて議論しよう。ビジネスで日々使用されている古いシステムが、最近になっていくつかの最新のシステムによって補完された。この新しいシステムは、腐敗防止層を経由してレガシーシステムと通信する。

最初にやるべきことの1つは、テスト戦略を決定することであるに違いない。自動化されたユニットテストは新システムの新しい機能に対して書くべきだが、レガシーを段階的に廃止するには特殊なテストが必要になる。組織の中には、ある一定期間、新旧システムを並行して稼動させるところもある。

どんなイテレーションにおいても、次のことが必要になる：

1. 1回のイテレーションで望ましいシステムのどれかに追加できるような、レガシーシス

テムの特定の機能を識別すること。

2. 腐敗防止層で必要となる追加機能を識別すること。
3. 実装すること。
4. デプロイすること。

レガシーから段階的に廃止できる、あるユニットに相当する機能を作成するために、複数回のイテレーションが必要になることもある。しかし、新しい機能を計画する際にはイテレーションに見合った量の小さなユニットで行い、複数回のイテレーションが行われている間はデプロイするのを待つこと。

デプロイについても、あまりに多くのバリエーションが存在するために、すべての基礎を網羅できない。小さい変更を少しずつ行った結果として製品ができるなら、開発にとって都合がよいのだが、通常はより大きなリリースを構成する必要がある。ユーザは新しいソフトウェアを使用するために教育を受けなければならない。並行期間は滞りなく完了しなければならない。事業計画に関する問題も解決しなければならない。

最終的に実地で稼動してからは、次のことが必要になる：

5. 腐敗防止層にある不要な部分を識別して除去する。
6. 現実的でないと判明するかもしれないが、レガシーシステムで現在使用されていないモジュールを稼働させることを考えること。皮肉なことに、レガシーシステムが優れた設計であるほど、段階的な廃止が容易なのだ。しかし、粗末な設計のソフトウェアは、一度に少しずつ取り除くことも難しい。使用されていない部分は、残りが段階的に廃止されて、全体が切り替えられるようになるまで、ただ無視しておくこともできる。

これを何度も繰り返すこと。レガシーシステムは、前ほどにはビジネスに関わらなくなり、最終的に長いトンネルの先に光が見えてきて、旧システムを切り離せるようになる。その間、多様な組み合わせによって、システム間の相互依存関係が増加したり減少したりするのに合わせて、腐敗防止層は収縮と膨張を交互に繰り返す。もちろん、他の条件がすべて同じならば、より小さい腐敗防止層につながる機能を最初に移行すべきである。だが、他の要因が支配的になることもあり、移行中は骨の折れる変換で我慢せざるを得なくなるかもしれない。

公開ホストサービス → 公表された言語

他システムとの統合を一連のその場しのぎのプロトコルで行ってきたのだが、より多くのシステムがアクセスを求め、あるいは、相互作用を理解するのが難しくなり始めるにつれて、保守の負担が増してきている。システム間の関係を公表された言語を用いて形式化

する必要がある。

1. 業界標準の言語が入手できるならば、それを評価し、使えるようなら使用すること。
2. 標準の言語や、すでに公表された言語が入手できなければ、ホストとして機能するシステムのコアドメインを明確にすることから始めること。(第15章参照)
3. コアドメインを交換用言語の土台として使用すること。その際に、できればXMLのような標準的な交換パラダイムを用いること。
4. 新しい言語を(少なくとも)共同作業に携わる人全員に公表すること。
5. 新しいシステムアーキテクチャが関係する場合は、それも公表すること。
6. 協調するシステムそれぞれに変換層を構築すること。
7. 切り替えること。

ここまで来れば、新しい協力者が後から参加するにあたって、それを妨げるものは最小限になっている。

覚えておいて欲しいのだが、公表された言語は安定していなければならないものの、容赦のないリファクタリングを続けるには、ホストのモデルを変更する自由も必要なのだ。それゆえ、**交換用言語とホストのモデルを同一視しないこと。**両者を近づけておけば、変換のオーバーヘッドが減り、ホストを順応者とする選択もできる。しかし、変換層を強化する権利は残しておき、費用対効果のトレードオフに見合う場合には分岐させること。

プロジェクトリーダは、機能的統合の要件と開発チームの関係とに基づいて境界づけられたコンテキストを定義しなければならない。いったん境界づけられたコンテキストとコンテキストマップが明示的に定義され尊重されるようになったら、論理的一貫性は保護されるはずである。関連するコミュニケーションの問題は、少なくとも露呈するようになるので、対処することができる。

しかし、モデルのコンテキストは、境界が意識的に決められたものであれ、自然に発生したものであれ、システム内で起きる論理的不整合以外の問題を解決するために、誤って適用されることがある。巨大なコンテキストのモデルが複雑すぎて、全体として理解したり、完全に分析したりできないように見えることに、チームが気づくかもしれない。選択の結果であっても、偶然であっても、結果的にコンテキストを管理しやすい小片に分割することになる場合も多い。この断片化によって機会が失われる。そういう時こそ、広範なコンテキストで巨大なモデルを規定するという意思決定について、真剣に考える価値があるのだ。一緒にしておくことが、組織的もしくは政治的に不可能であったり、現実に断片化していたりするのであれば、マップを描き直し、維持できる境界を定義すること。しかし、巨大な境界づけられたコンテキストによって、やむを得ない統合の要求に対応でき、しかもモデルそのものの複雑さを別にすれば、1つのコンテキストを作ることが現実的であ

るなら、コンテキストを分割するのは最善の策ではないかもしれない。

巨大なモデルを扱いやすくする手段で、こうした犠牲を払う前に考慮すべきものが他にある。次の2つの章では、より広範な2つの原則を適用することで、巨大なモデルにある複雑さを管理することに焦点を合わせる。それが、蒸留と大規模な構造である。

第15章

蒸　留

$$\nabla \cdot \mathbf{D} = \rho$$
$$\nabla \cdot \mathbf{B} = 0$$
$$\nabla \times \mathbf{E} = -\frac{\partial \mathbf{B}}{\partial t}$$
$$\nabla \times \mathbf{H} = \mathbf{J} + \frac{\partial \mathbf{D}}{\partial t}$$

—James Clerk Maxwell『A Treatise on Electricity and Magnetism』1873

これら4つの方程式は、その項の定義および根拠とする数学本体に加えて、
古典的な19世紀電磁気学の全容を表現している。

中心となる問題に集中し、枝葉末節の問題の海で溺れずにいるためにはどうすればよいだろうか？ レイヤ化アーキテクチャを用いることによって、コンピュータシステムを動かすための技術的なロジックからドメインの概念が分離されるが、巨大なシステムでは、隔離されたドメインさえも手に負えないほど複雑かもしれない。

蒸留（distillation）とは、混ざり合ったコンポーネントを分離するプロセスであり、価値があって役立つ形式で本質を抽出するためのものだ。モデルとは、知識が蒸留されたものである。より深いモデルへ向けてリファクタリングするたびに、ドメインの知識と優先順位に関する重要な側面が抽象化されるのだ。ここで戦略的な見方に戻り、本章では、広範囲にわたるモデルを識別して、全体としてのドメインモデルを蒸留する方法を見ていく。

多くの化学的な蒸留と同様に、蒸留プロセスによって分離された副産物は、それ自体が（汎用サブドメインや凝集されたメカニズムのように）はるかに価値のあるものになる。し

かし、蒸留するというこの努力の動機になっているのは、特別に価値がある部分をたった1つ抽出したいという欲求である。それが我々のソフトウェアを特徴づけ、構築する価値のあるものにする部分、すなわち「コアドメイン」である。

ドメインモデルに対する戦略的蒸留は、次に挙げることをすべて行う。

1. システムの全体的な設計と、設計同士がどう関係するのかを、チームメンバ全員が把握できるようにする。
2. ユビキタス言語に入れやすいサイズのコアモデルを識別することで、コミュニケーションを促進する。
3. リファクタリングの指針となる。
4. モデルで最も価値がある領域に作業を集中させる。
5. アウトソーシングと既製のコンポーネントの利用、その割り当てについて決定する際の指針となる。

本章は、コアドメインの戦略的蒸留に対する体系的なアプローチを提示する。また、チーム内でその見方を効果的に共有する方法を説明して、何を行っているかについて話すための言語を提供する。

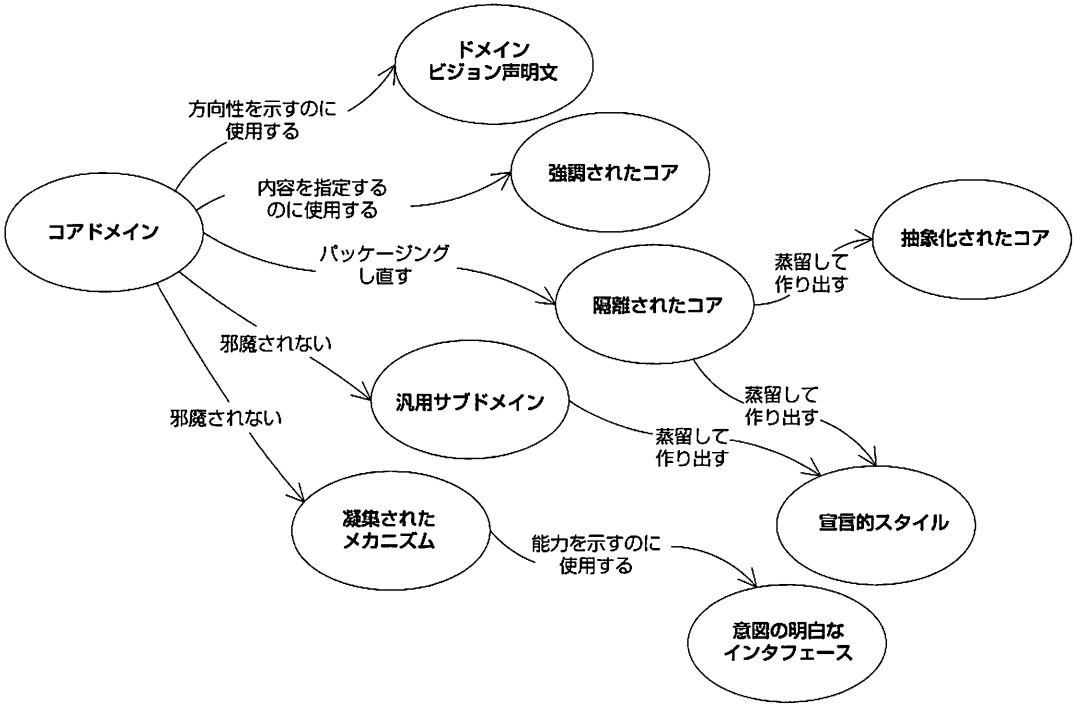


図15.1 戦略的蒸留のためのナビゲーションマップ

木を剪定して主要な枝の成長を妨げないようにする庭師のように、テクニックを一式適用することで、モデルから気を散らすものを切り落として、最も重要な部分に集中できるようにする…。

コアドメイン（CORE DOMAIN）



巨大なシステムを設計する際には、寄与するコンポーネントの数が非常に多い。しかも、どれも複雑な上に成功にはすべてが絶対に必要である。そのため、ドメインモデルの本質である実際のビジネスの資産が不明瞭になり、おろそかにされてしまうかもしれない。

理解するのが難しいシステムは、変更するのも難しい。変更による影響が予測しにくいのだ。開発者は自分が熟知する領域の外に出ると、道に迷ってしまう（このことは、チームに新しい人材を迎える時に特に該当するが、チームのベテランであっても、コードが表現力豊かに正しく構成されていなければ苦労するだろう）。このために、人は専門的にならざるを得ない。開発者が作業を特定のモジュールに限定すると、知識の伝達がさらに損なわれる。作業を区分化することで、システムのスムーズな統合が損なわれ、作業を割り当てる際の柔軟性が失われる。あるふるまいがすでにどこかに存在していることを開発者が認識していないと重複が生じ、それによってシステムはさらに複雑になる。

こうしたことは、設計が理解しにくい時には常に起こる結末だ。しかし、これと同じように深刻なリスクは、ドメインの全体像を見失うことからも生じる。

現実は厳しいもので、設計のすべての部分が等しく改良されるわけではない。優先順位を設定しなければならないのだ。ドメインモデルを資産化するには、モデルのきわめて重要なコアを洗練し、アプリケーションの機能を作り出す上で最大限に活用しなければならない。反面、高いスキルを持った開発者はそう多くいないが、そういう開発者が魅かれがちなのは、技術的なインフラストラクチャや、特別なドメインの知識がなくても理解可能な、うまく限定できるドメインに関する問題なのである。

システムのそういう部分は、コンピュータ科学者の関心を引くらしく、他でも使える専門スキルを構築して、履歴書を書く時のいい材料になると考えられている。特化したコアは、アプリケーションを実際に差別化してビジネスの資産とするようなモデルの部分であるのに、それほどスキルのない開発者によってまとめられることになるのが通例である。そういう開発者はデータベース管理者と協力してデータスキーマを作成し、モデルの持つ概念の力を一切利用することなく1機能ずつコードを書く。

ソフトウェアにおけるこの重要な部分の設計や実装をおろそかにすると、どれほど技術的なインフラストラクチャがうまく機能し、どれほどサポート機能が優れていても、作られるアプリケーションがユーザに訴求することはない。ひっそりと進行するこの問題が根を張るのは、プロジェクトに設計全体の明確なイメージが欠け、さまざまな部分について相対的な重要度が考えられていない場合である。

私が参加したことがある中で、最も成功したプロジェクトの1つは、当初、この症状に悩まされていた。このプロジェクトの目標は、非常に複雑なシンジケートローンシステムを開発することであった。優れた才能を持つ人々のほとんどが喜々としてデータベースマッピング層とメッセージングインタフェースに取り組む一方で、ビジネスモデルはオブジェクト技術に不慣れな開発者に委ねられていた。

唯一例外があって、経験豊富な1人のオブジェクト開発者がドメインの問題に取り組んでおり、寿命の長いドメインオブジェクトすべてにコメントをつける方法を考え出した。このコメントは、トレーダが過去の決定について自分や他の人が記録した論理的根拠を見られるようにまとめることができた。この開発者はさらに、的確なユーザインタフェースを構築して、コメントモデルの柔軟な機能に直観的にアクセスできるようにした。

これらの機能は便利で、うまく設計されており、製品にもなった。

ただ、残念ながら、この機能はドメインにおいて周辺的なものだった。この才能ある開発者は、コメントをつけるための興味深く、汎用的な方法をモデル化し、それを整理して実装して、ユーザの手に渡した。その間、適任ではない開発者がミッションクリティカルである「ローン」モジュールを理解できないほどもつれさせ、このプロジェクトはほぼ回復不可能な状態になってしまったのだ。

計画プロセスの時に、モデルと設計における最も重要なポイントにリソースを向けるべきだ。そのためには、計画と開発の間、全員がこうしたリソース配置について理解していなければならない。

開発対象のアプリケーションの目的にとって、代表的で中心的なモデルの部分が、コアドメインを構成する。コアドメインとは、システム内で最大の価値を付加すべき場所である。

それゆえ：

モデルを煮詰めること。コアドメインを見つけて、それをサポートする大量のモデルやコードから容易に区別する手段を提供すること。最も価値のある特化した概念を浮き彫りにすること。コアは小さくすること。

最も才能がある人をコアドメインに割り当て、そうした区分に応じて採用すること。コアに対して力を尽くし、深いモデルを見つけ出し、システムのビジョンを実現するのに十分な、しなやかな設計を開発すること。それ以外の部分への投資は、蒸留したコアをどうサポートするかによって正当化すること。

コアドメインを蒸留するのは簡単なことではないが、それによっていくらか意思決定が容易になる。コアの差別化を図りつつ、設計の残りの部分は、実用的である限りにおいて汎用性を保つために、読者は労力を注ぎ込むことになるだろう。競合時に優位に立つために、設計の一部の側面を秘密にしておく必要があるなら、それがコアドメインだ。残りを隠すのに無駄な努力を払う必要はない。望ましいリファクタリングが2つあって、(時間の制約により) どちらかを選択しなければならない時には、コアドメインに与える影響が大きい方を先に選ぶべきである。

* * *

本章の各パターンにより、コアドメインを見たり、使用したり、変更したりするのが容易になる。

コアを選択する

ここで見ていくのは、モデルの中でも特に、ビジネスドメインを表現して、ビジネスの問題を解決する部分である。

選択するコアドメインは視点によって決まる。例えば、多くのアプリケーションでは、多様な通貨と為替レートおよび両替を表現できる、汎用的な金銭のモデルが必要になるだろう。一方、為替取引をサポートするアプリケーションは、より手の込んだ金銭モデルを必要とするかもしれず、それはコアの一部と見なされるだろう。そういう場合にも、金銭モデルの一部がきわめて汎用的であることもある。ドメインに対する洞察が経験によって深まっても、蒸留プロセスは継続できる。その際には、汎用的な金銭の概念を分離し、モデルの特化した側面だけをコアドメインに残しておく。

輸送アプリケーションの場合、複数の貨物を輸送用にまとめる方法や、コンテナの所有者が変わった場合の責任の受け渡し方法、特定のコンテナが荷出し地に到達するまでの多様な運送経路の選択方法といったモデルが、コアとなるだろう。投資銀行であれば、コアには、代理人と関係者の間で資産をシンジケート化するためのモデルが含まれているだろう。

あるアプリケーションのコアドメインは、別のアプリケーションにとっては補助的な汎用コンポーネントになる。それでも、1つのプロジェクトを通じて、また、通常は1つの企業全体を通して、一貫したコアを定義することはできる。設計にあるその他すべての部分と同様、コアドメインの識別はイテレーションを通じて進化しなければならない。関係の特定の集まりが持つ重要性は、一見しただけではわからないかもしれない。最初はどう見ても中心的であると思われたオブジェクトが、実は補助的な役割しか持っていないことが明らかになるかもしれない。

次に続く節における議論、特に汎用サブドメインにおける議論では、こうした決定に関するさらなる指針を提供する。

誰がこの作業をやるのか？

プロジェクトチームにおいて技術的に最も卓越したメンバが、ドメインについて多くの知識を持っていることはめったにない。そのために、彼らが貢献できる度合いが制限され、補助的なコンポーネントに割り当てられる傾向が強まり、知識が不足しているせいで、ドメインの知識を形成するはずの作業から遠ざけられるという悪循環が重ねられる。

この循環を断ち切ることが不可欠である。そのために、長期にわたって参画していてドメインの知識のリポジトリとなることに関心がある強力な開発者の一団と、ビジネスを熟知するドメインエキスパート1人以上がうまく協力しあえるようなチームを召集する。ドメイン設計は、真剣に取り組めば興味深く、技術的にやりがいのある仕事であり、そういう考えを持つ開発者も見つかるものだ。

通常、コアドメインを作成するための基礎に対して、外部から専門家を短期間雇うのは実践的でない。チームはドメインについての知識を蓄積する必要があり、一時的なメンバは知識が流出する原因になるからである。その一方で、教育や指導を行う専門家であれば、非常に価値があるかもしれない。チームがドメイン設計のスキルを形成するのを支援して、チームメンバがおそらくまだ修得していない洗練された原則を使うよう促してくれるからだ。

同様の理由から、コアドメインを購入することもできそうにない。業界に特化したモデルフレームワークを構築する努力は重ねられている。著名な例には、半導体業界のコンソーシアムであるSEMATECHが作成した、半導体の製造を自動化するCIMフレームワークや、広範なビジネスに対応するIBM社の「San Francisco」フレームワークがある。これは非常に魅力的な考えだが、これまでの成果で訴求力があるのは、データ交換を容易にする公表された言語（第14章参照）を除けば他にない。『Domain-Specific Application Frameworks』（FayadとJohnson 2000）を読めば、この最先端の技術が概観できる。この分野が進歩するにつれ、もっと多くの実用的なフレームワークが入手可能になるだろう。

その上で、注意する必要のある、より根本的な理由がある。つまり、カスタマイズされたソフトウェアの持つ最大の価値は、コアドメインを完全にコントロールできることにあるのだ。うまく設計されたフレームワークであれば、用途に合わせて特化させることのできる、高度な抽象を提供できるかもしれない。そうすることで、より汎用的な部分を開発する手間が省け、自由にコアに集中できるようになるだろう。しかし、それ以上に制約があるとすれば、考えられる可能性は3つある。

1. 本質的なソフトウェア資産を失いかけている。コアドメインでは、制約のあるフレームワークの利用を控えること。
2. フレームワークの扱う領域が思っていたほど中枢となるものでなかった。モデル内で本当に特徴的な部分を示すようにコアドメインの境界を引き直すこと。
3. コアドメインについての特別な要求がない。使用中のアプリケーションと統合できるソフトウェアを購入するなど、リスクの低い解決策を検討すること。

いずれにしても、他とは違うソフトウェアを作ろうと思えば、専門知識を蓄積して、それをかみ砕いて豊かなモデルにする、安定したチームに立ち返ることになる。近道も魔法の銃弾もない。

蒸留の拡大

本章の残りを構成するさまざまな蒸留テクニックは、ほぼどんな順番でも適用できるが、設計をどこまで徹底的に変更するかという点には幅がある。

シンプルなドメインビジョン声明文は、最小の投資で基本的な概念とその価値を伝える。強調されたコアは、コミュニケーションを改善して、意思決定をしやすくする。ただし、それでも設計はほとんど変更しなくてよいか、あるいは全く変更する必要がない。

より積極的なリファクタリングと再度のパッケージングによって、汎用サブドメインが明示的に分離され、それぞれ個別に扱えるようになる。凝集されたメカニズムは、用途が広く、伝達力があるしなやかな設計でカプセル化できる。こうして気を散らすものを取り除くことで、コアのもつれが解消される。

隔離されたコアをパッケージングし直すことで、コードの中であってもコアが直接見えるようになり、コアモデルに対する将来の作業が容易になる。

最も野心的なものが抽象化されたコアで、これは最も根本的な概念と関係性を純粋な形式で表現する（そして、広範囲にわたるモデルの再構成とリファクタリングが必要となる）。

こうしたテクニックは順次深く関わらなければならないものになっていくが、刃が研ぎすまされるほどナイフの切れ味は鋭くなるものだ。ドメインモデルを繰り返し蒸留するこ

とで作り出される資産は、プロジェクトにスピードと俊敏さ、そして実行の正確さをもたらす。

まず、モデルから最も特徴的でない部分を除去すればよい。汎用サブドメインはコアドメインの好対照となっており、それぞれの意義を明確にする…。

汎用サブドメイン（GENERIC SUBDOMAINS）

モデルの中には、専門的な知識をとらえることも伝えることもなく、複雑さをつけ加えるだけの部分もある。本質と無関係なものがあると、それが何であれ、コアドメインを見分け、理解するのが困難になる。モデルを停滞させるのは、誰もが知っている一般原理や、専門領域に属するのに主要な焦点ではなく、補佐役を果たすような詳細である。だが、いかに汎用的でも、こういう本質とは別の要素は、システムを機能させ、モデルを完全に表現する上で欠かせない。

モデルの中には、当然と見なしたい部分もある。紛れもなくドメインモデルの一部なのだが、相当数のビジネスで必要になると思われる概念を抽象化している。例えば、輸送業、銀行業、製造業などビジネスが多様であっても、何らかのかたちで企業の組織図は必要になる。別の例では、多くのアプリケーションが追跡する売掛金や経費元帳、その他財務に関するものは、汎用的な会計モデルを使うことですべて処理できる。

多大な労力がドメインにおける周辺的な問題に費やされることも多い。個人的に目にしたところでは、2つの別々のプロジェクトで、最高の開発者を数週間雇い、タイムゾーンを含む日付と時刻の設計をやり直していた。そういうコンポーネントは、機能しなければならないが、システムにおける概念的なコアではない。

そうした汎用的なモデル要素が重要と考えられる場合でも、ドメインモデル全体としては、システムにおいて最も付加価値の高い、特別な側面を目立たせなければならず、できる限りその部分に力を与える構造になっている必要がある。コアが相互に関連する要素すべてと混ざり合っていると、これは難しい。

それゆえ：

設計中のプロジェクトにとって動機となっていない、高凝集のサブドメインを識別すること。そうしたサブドメインから汎用的なモデルを括り出して、別のモジュールに入れること。その中に特化した要素を一切残してはならない。

いったん分離したら、その部分に対して継続される開発の優先順位をコアドメインより下げ、その作業にコアとなる開発者を割り当てるのを避けること（そこから得られるドメインについての知識はほとんどないからだ）。このような汎用サブドメインに対しては、既製品による解決策や公表されているモデルの採用も検討すること。

＊ ＊ ＊

こういったパッケージを開発する際には、他にもいくつか選択肢があるかもしれない。

選択肢1：既製品による解決策

実装されたものを購入したり、オープンソースコードを使用したりできることがある。

利点

- 開発するコードが少なくなる。
- 保守の負担が社外に出される。
- コードは、おそらく成熟度が高く、多くの場所で使用されているから、自家製のコードよりも安定していて完成している。

欠点

- それでも使う前には、時間をかけて評価し、理解しなければならない。
- 品質管理について、この業界はこの有様だから、正確で安定しているだろうと当てにすることはできない。
- 読者の目的を実現するものとしては過剰に作り込まれすぎているかもしれず、統合する際には、ミニマリズムで作られた自家製の実装よりも作業が多くなりかねない。
- 通常、異質な要素はスムーズに統合できない。別の境界づけられたコンテキストがあるかもしれないのだ。そうでなくても、エンティティを他のパッケージからスムーズに参照するのは難しいだろう。
- プラットフォームへの依存やコンパイラバージョンへの依存などが入り込むかもしれない。

既製のサブドメインによる解決は、投資する価値がある。だが、苦労する価値はないのが普通だ。非常に手の込んだワークフローを要求されたアプリケーションが成功した話を聞いたことがあるのだが、これはAPIフックのついた市販されている外部のワークフローシステムを使用していた。また、アプリケーションに深く統合されたエラーロギングパッケージが成功しているのも見たことがある。汎用サブドメインを用いた解決策は、フレームワークというかたちでパッケージングされていることもあり、それは設計中のアプリケーションと統合したり、そのアプリケーションに特化したりできる、非常に抽象的なモデルを実装している。サブコンポーネントが汎用的であるほど、また自身のモデルが蒸留されているほど、それが有益である見込みは高くなる。

選択肢2：公表されている設計やモデル

利点

- 自家製のモデルより成熟していて、多くの人々の洞察を反映している。
- 手間がかからず、品質の高いドキュメント。

欠点

- 要求に完全には合致していないかもしれないし、要求に対して作り込みすぎているかもしれない。

Tom Lehrer（1950年代から1960年代の風刺歌手）は、数学で成功する秘訣についてこう言った。「剽窃せよ！ 剽窃せよ。誰の研究も見逃してはならない…。ただ、いつも調査と呼ぶことをお忘れなく」。これは、ドメインモデリングに対しても優れたアドバイスであるし、汎用サブドメインに取り組む場合には特に当てはまる。

これが最もうまくいくのは、『アナリシスパターン』（Fowler 1996）で書かれているような、広く流通しているモデルがある場合だ（第11章参照）。

その分野において、高度に形式化された厳密なモデルがすでにある場合には、それを使用すること。まず思い浮かぶ2つの例は会計と物理学である。これらは非常に強固で合理化されているだけでなく、広く人々に理解されているので、現在や将来行うべき教育の負担が軽減される（確立された形式主義の使用に関しては第10章を参照）。

公表されているモデルの持つ側面をすべて実装しなければならないとは思わないこと。首尾一貫していて要求を満たす、簡略化されたサブセットが特定できればよいのだ。しかし、使い込まれて、適切にドキュメント化され、さらに欲を言えば形式化されたモデルが手に入るなら、車輪を再発明することに意味はない。

選択肢3：実装のアウトソーシング

利点

- コアとなるチームが自由にコアドメインに取り組めるようになる。このコアドメインこそ、ほとんどの知識が必要とされる場所であり、その知識が蓄積される場所でもある。
- 恒久的にチームを拡大させることなく、またコアドメインの知識を消散もさせずに、より多くの開発ができるようになる。
- インタフェース指向の設計を強制して、サブドメインを汎用的に保ちやすくする。これは、仕様が外部に渡されるためだ。

欠点

- やはりコアとなるチームは、時間を割く必要がある。インタフェースやコーディング標準、その他の重要な側面を伝えなければならないからだ。
- 所有権を元の内部へ戻すのに、著しいオーバーヘッドが生じる（それでも、特化したサブドメインよりはオーバーヘッドが少ない。汎用的なモデルを理解するのに特別な背景は必要ないからだろう）。
- コードの品質は、さまざまであるかもしれない。2チーム間の相対的な力量次第で、こ

れはいいことも悪いこともある。

自動化されたテストは、アウトソーシングの際に重要な役割を果たせる。実装側には、作成するコードのユニットテストを提供するように要求すべきである。実際に強力なアプローチとしては、アウトソースされるコンポーネントに対して自動化された受入テストを定義したり、作成したりするというものがある。このアプローチは、品質の程度を保証するのに役立ち、仕様を明確にして再統合をスムーズにする。また、「実装のアウトソーシング」は、「公表されている設計やモデル」と組み合わせると、実にうまくいくかもしれない。

選択肢4：自社での実装

利点

- 統合が容易である。
- まさに欲しいものだけが手に入り、余分なものがない。
- 一時的な社員を割り当てることができる。

欠点

- 保守と教育の負担が継続する。
- そういうパッケージを開発するのにかかる時間とコストは、少なく見積もられてしまうことが多い。

もちろん、これも「公表されている設計やモデル」とうまく組み合わせられる。

汎用サブドメインは、外部の設計にある専門知識を適用してみるのに適した場所である。特別なコアドメインを深く理解する必要がないからであり、そのドメインについて学ぶ格好の機会を提供することもないからである。機密性は大きな関心事にはならない。独占すべき情報やビジネスで行われていることが、そういうモジュールにはほとんど含まれないからだ。汎用サブドメインにより、対象となるドメインについての深い知識にあまり関わらない人々を教育する負担が軽減される。

時とともに、コアドメインを構成するのは何かということについての考え方は狭まっていき、ますます多くの汎用的なモデルが、実装フレームワークや、少なくとも公表されたモデルやアナリシスパターンとして手に入るようになると思われる。今のところは、汎用サブドメインのほとんどを自分たちで開発しなければならないが、それをコアドメインモデルから切り分けることには大きな価値がある。

例15.1——2つのタイムゾーンの物語

プロジェクトで最も優秀な開発者たちが数週間を費やし、タイムゾーンを用いた時間の格納と変換を行う問題を解決しようとするのを、私はこれまで2回見たことがある。私はそういう活動には常に懐疑的だが、実際には必要なこともある。この2つのプロジェクトは、ほぼ完全に対照的であった。

1つめは、貨物輸送のスケジュールを立てるソフトウェアを開発するのに向けた努力だった。国際運送のスケジュールを立てるには、正確に時間を計算できることがきわめて重要である。そうしたスケジュールがすべてローカルタイムで追跡されるので、運送を調整するには変換が欠かせないのだ。

この機能に対する要求を明確に設定した上で、このチームはコアドメインの開発を進め、利用可能な時間クラスとダミーデータを使用して、アプリケーションに対する最初のイテレーションを行った。このアプリケーションが成熟し始めるにつれて明確になったのは、既存の時間クラスでは不十分であるということと、多くの国でバリエーションが見られ、日付変更線が複雑であるせいで、問題が非常に錯綜しているということであった。この時点で要件はさらに明確になっていたため、彼らは既製の解決策を探したが、何も見つからなかった。自分たちで構築するよりほかなかったのである。

この作業には調査と正確な技術が必要だったので、チームリーダたちは最も優秀なプログラマの1人を割り当てた。しかし、輸送に関する特別な知識は必要なく、そういう知識を育むものでもなかったから、プロジェクトに一時的な契約で参加していたプログラマが選ばれた。

このプログラマは一から始めることはしなかった。タイムゾーンに関する既存の実装をいくつか調査したが、そのほとんどは要件に合わなかったので、BSD Unixが持つパブリックドメインの解決策を改造することにした。これには手の込んだデータベースがあり、C言語で実装されていた。ロジックのリバースエンジニアリングを行って、データベース用の取り込み処理を書いたのである。

問題は予想よりも難しいことが判明した（例えば、特殊なケースをデータベースにインポートすることもあった）が、コードは書き上げられてコアと統合され、製品は納品された。

もう一方のプロジェクトでは、全く別の方向に進んだ。ある保険会社が新しい支払請求処理システムを開発しており、さまざまなイベントの時間（車の衝突時間、霰を伴う嵐の時間など）を把握しようと計画していた。このデータはローカルタイムで記録することになっていたので、タイムゾーンの機能が必要とされた。

私が参加した時、その作業を担当していたのは、若いながらも非常に頭の切れる開発者だったが、アプリケーションの正確な要件はまだ確定しておらず、最初のイテレーションすら試みられていなかった。職務に忠実だった彼は、前もって推測を立ててタイムゾーンモデルの構築に取り掛かった。

何が必要なのかわからなかったので、どんなものでも扱えるくらいに柔軟でなければならないと想定された。この作業を割り当てられたプログラマは、そういう難解な問題に対処するに

は助けを必要としたので、上級の開発者も担当することになった。彼らは複雑なコードを書いたのだが、具体的なアプリケーションでそれを使用することはなかったから、コードが正しく動いたかどうかは定かではない。

このプロジェクトはさまざまな理由で暗礁に乗り上げ、タイムゾーンコードが使用されることはなかった。しかし、使われていたとしても、そのローカルタイムをタイムゾーンでタグづけして格納すれば十分で、変換する必要はなかったかもしれない。これは主として参照用のデータであり、計算の基準にはならなかったからだ。変換が必要だとわかったとしても、データはすべて北米から収集されるので、タイムゾーンの変換は比較的単純なのである。

タイムゾーンにこれだけ注意を向けたことによる最大の代償が、コアドメインモデルを無視してしまったことだった。同じ量のエネルギーをそちらに注いでいれば、実際に動くアプリケーションのプロトタイプと機能するドメインモデルの初版ができていたことだろう。さらに、担当した開発者たちがこのプロジェクトに長期間打ち込むことになっていれば、保険ドメインに浸かり、チームの中で重要な知識が蓄積されたはずである。

どちらのプロジェクトでも正しく行われていたのは、汎用的なタイムゾーンモデルをコアドメインから明確に分離することだった。タイムゾーンに関して輸送や保険に特化したモデルにしていたら、コアモデルがこの汎用的な補助モデルと結びついて、コアは理解しにくくなっただろう（そのモデルはタイムゾーンについて、本来のコアとは無関係な詳細を含んでいただろうから）。また、タイムゾーンモジュールの保守も困難になっていただろう（保守担当者がコアとそのタイムゾーンとの相互関係を理解しなければならなかっただろうから）。

輸送プロジェクトの戦略	保険プロジェクトの戦略
利点	**利点**
◎汎用的なモデルがコアから分離されている。	◎汎用的なモデルがコアから分離されている。
◎コアモデルが成熟しているので、リソースを他に向けても支障がなかった。	**欠点**
◎何が必要かを正確に把握している。	◎コアモデルが開発されておらず、他の問題へ注意が向いたことで、このことが放置され続けた。
◎国際的なスケジュールを立てる上で決定的に重要な支援機能がある。	◎要件が未知のため、完全に汎用的に作ってみることになったが、北米に限った単純な変換で十分だった可能性がある。
◎短期契約のプログラマを汎用的なタスクに割り当てている。	◎ドメインについての知識のリポジトリになり得る長期のプログラマに担当させた。
欠点	
◎トッププログラマをコアから転用した。	

我々技術系の人間は、タイムゾーン変換のような限定可能な問題を楽しむ傾向があり、そういう問題に時間を費やすことをたやすく正当化してしまう。しかし、規律に従って優先順位を見れば、重要視すべきはコアドメインである。

●例 15.1 ── 2つのタイムゾーンの物語

汎用とは再利用可能という意味ではない

これまでサブドメインの汎用性を強調しながらも、コードの再利用性には言及してこなかったことに注意して欲しい。既製の解決策は具体的な状況に対して、効果があることもないこともある。しかし、コードを実装するのが、自分か自社かアウトソーシングであると仮定すると、そのコードの再利用性は特に気にすべきではない。そんなことをすれば、蒸留の基本的な動機に反する。努力はできる限り多くコアドメインに対して行い、補助的な汎用サブドメインには必要な場合にだけ投資すべきなのだ。

再利用は確かに行われるが、いつもコードの再利用とは限らない。公表された設計やモデルを使用する時のように、モデルの再利用の方が再利用としては優れていることが多い。そして、独自のモデルを作成しなければならないのであれば、後の関連するプロジェクトで役に立つこともあるだろう。しかし、そういうモデルの概念が多くの状況に適用可能であるとはいえ、モデルに完全な汎用性を持たせるように開発しなくてもよいのである。ビジネスに必要な部分だけをモデル化し、実装すればよい。

再利用性のために設計する必要はめったにないとはいえ、汎用的な概念の範囲内に収めることに関しては厳格でなければならない。業界に特化したモデル要素を導入するのにかかるコストは2つある。第一に、将来の開発を妨げることになる。現在はサブドメインモデルのごく一部しか必要なかったとしても、要求は拡大するものである。概念の一部ではないものを設計に取り入れてしまうと、システムを拡張する際には、古い部分を完全に再構築するか、その異質な部分を使用しているモジュールを再設計しない限り、クリーンに行うことがより難しくなる。

第二のより重要な理由は、そうした業界に特化した概念はコアドメインに属するか、独自の特殊なサブドメインに属するかのいずれかであり、こうした特別なモデルは汎用的なモデルよりも価値があるということだ。

プロジェクトのリスク管理

アジャイルプロセスで通常求められるリスク管理は、最もリスクが高いタスクに早期に対応するというものだ。特に、XPはエンドツーエンドのシステムを即座に立ち上げて稼動させることを要求する。この初期システムによって、技術的なアーキテクチャの妥当性が証明されることも多く、補助的な汎用サブドメインを扱う周辺的なシステムを構築したいという欲求もありがちだ。こうした周辺的なシステムは、たいてい分析しやすいからである。しかし、注意して欲しい。これでは、リスク管理という目的が達成できないのだ。

プロジェクトが直面するリスクは、技術とドメインモデリングの両方から来る。つまり、技術的なリスクが大きいプロジェクトもあれば、ドメインモデリングのリスクが大きいプロジェクトもある。エンドツーエンドのシステムがリスクを軽減できるのは、実際のシス

テムにおいて課題となる部分に対する、萌芽期のバージョンであるという点に限られる。ドメインモデリングのリスクは容易に過小評価されてしまう。このリスクは、予期せぬ複雑さ、ビジネスエキスパートに十分にアクセスできない状況、開発者の主要なスキルの不足といったかたちで現れる。

それゆえ、チームのスキルが実証されていて、ドメインがよく知られている場合を除き、初版のシステムは、単純であっても、コアドメインの一部に基づいて作らなければならない。

同じ原則は、リスクの高いタスクを推進しようとするプロセスであれば、どんなものにも当てはまる。コアドメインのリスクが高いのは、しばしば予想に反して難解な場合がありながら、それでもコアドメインなしにプロジェクトが成功することはあり得ないからである。

本章にある蒸留パターンのほとんどは、コアドメインを蒸留するためにモデルとコードをどのように変更すればよいかを示している。これに対して、次に挙げる2つのパターン、すなわちドメインビジョン声明文と強調されたコアが示すのは、補足的なドキュメントをどう使えば、非常にわずかな投資でコミュニケーションとコアに対する意識を改善し、開発の努力を集中させられるか、ということである…。

ドメインビジョン声明文（DOMAIN VISION STATEMENT）

プロジェクトが始まった時には、モデルは存在さえしていないのが普通だが、開発に集中する必要性はすでに生じている。開発の後期になると、モデルを詳細に調べなくてもわかるかたちで、システムの価値を説明する必要がある。また、ドメインモデルの重要な側面は、複数の境界づけられたコンテキストにまたがることもあるが、これら別々のモデルはその定義上、共通の焦点を示すような構造にはできない。

多くのプロジェクトチームは、管理のために「ビジョン声明文」（vision statement）を作成する。このドキュメントの最もよいところは、アプリケーションが組織にもたらす具体的な価値を並べて示している点だ。ドメインモデルを作成することについて、これを戦略的資産として挙げるものもある。通常、ビジョン声明文を記したドキュメントはプロジェクトに資金の都合がつけば捨てられ、実際の開発プロセスで使用されることも、技術系の担当者が読むことさえもない。

ドメインビジョン声明文はそういうドキュメントにならってモデル化されるものだが、焦点はドメインモデルの本質とエンタープライズにどのように役に立つかにある。ドメインビジョン声明文は、開発のあらゆるフェーズで経営陣と技術系の担当者が直接使うことができて、リソースの割り当てやモデリングの選択における指針となり、チームメンバの教育にも使われる。ドメインモデルが仕えるべき対象が多くても、このドキュメントがあれば、それぞれの関心がどうバランスを取られているかがわかる。

それゆえ：

コアドメインとそれがもたらす価値に関する簡潔な記述を作成すること（約1ページ）。これが、「価値の提議」（value proposition）である。ドメインモデルを他と差別化するものでない側面は無視すること。ドメインモデルがどのように役立ち、多様な関心に対してどうバランスを取るかを示すこと。スコープを狭い範囲に留めること。この声明文は早期に作成し、新しい洞察を得たら改訂すること。

ドメインビジョン声明文は道標として用いられるもので、開発チームを共通の方向に向かわせ、モデルとコード自体の蒸留を続けさせる。技術系ではないチームメンバや経営陣、さらには顧客とも共有できるのだ（もちろん、独占すべき情報を含む部分を除く）。

これは**ドメインビジョン声明文**の一部である	これは重要であるが、**ドメインビジョン声明文**の一部ではない
航空会社予約システム このモデルは乗客の優先順位と航空会社予約戦略を表し、柔軟なポリシーに基づいてそれらのバランスを取ることができる。乗客のモデルには、航空会社がリピータとの間に構築しようと努めている「関係性」を反映するべきだ。それゆえ、このモデルは乗客の履歴を役立つかたちで凝縮して表現しなければならない。こうした履歴としては、特別プログラムへの参加や、戦略的な企業顧客との提携などがある。 いろいろなユーザの持つさまざまな役割（乗客、職員、経営者）も表現されて、関係性を表現したモデルを豊かにし、セキュリティフレームワークに必要な情報を与える。 モデルは、ルートや座席の効率的な検索をサポートしなければならず、すでにある他の航空会社予約システムとも統合しなければならない。	**航空会社予約システム** ユーザインタフェースは、慣れたユーザには効率的でありながら、初めて使うユーザにとっても、使いやすくなければならない。 アクセスはWeb経由や、他システムへのデータ転送、さらに他のユーザインタフェースを経由しても提供されるかもしれないので、インタフェースはXMLを中心として設計し、Webページを提供したり、他システムへ変換を行ったりする変換層を付加する。 カラフルでアニメーションするバージョンのロゴはクライアント端末にキャッシュし、次回に訪問した時にすぐに起動するようにする。 顧客が予約を送信したら、5秒以内にコンファメーションを表示すること。 セキュリティフレームワークはユーザが本人であることを認証し、あらかじめ定義されたユーザの役割に割り当てられる特権に基づいて、特定の機能へのアクセスを制限する。

これは**ドメインビジョン声明文**の一部である	これは重要であるが、**ドメインビジョン声明文**の一部ではない。
半導体工場の自動化 このドメインモデルはウエハ製造工場にある原料と設備の状態を表し、それによって、必要な監査証跡を提供し、自動化された製品の製造計画をサポートするものとする。 モデルには、プロセスで必要とされるヒューマンリソースは含まれないが、ダウンロードした製法に従って、選択的にプロセスを自動化できなければならない。 工場の状態を表現する際には、管理者にとって理解できるものであるべきで、それによって深い洞察を与え、意思決定をサポートするものでなければならない。	**半導体工場の自動化** このソフトウェアはサーブレットによってWeb化するが、代替となるインタフェースも許容するように構成する。 業界標準の技術はできる限り使用して、社内での開発や保守コストを下げつつ、外部の専門知識を最大限活用する。オープンソースソリューションが望ましい（例えば、Apache Webサーバなど）。 Webサーバは専用のサーバで稼働させ、アプリケーションも、単体の専用サーバで稼働させる。

* * *

ドメインビジョン声明文は、チームに共通の方向性を与える。高次の声明文とコードやモデルに関する完全な詳細との間では、通常、橋渡しが必要になる…。

強調されたコア（HIGHLIGHTED CORE）

ドメインビジョン声明文は、広い観点から見た場合のコアドメインを識別するものだ。しかし、具体的なコアモデルの要素の識別は、予測できない個人の解釈に委ねられる。並外れて高いレベルのコミュニケーションがチームで行われていない限り、ドメインビジョン声明文だけではほとんど効果がない。

＊ ＊ ＊

チームメンバがコアドメイン**の構成を大まかに知っているとしても、別の人々が全く同じ要素を拾い上げることはなく、同じ人であってさえも日によって変わるものだ。絶えずモデルをフィルタリングして主要な部分を識別するという知的労働は、本来であれば設計について考えるのに費やすべき集中力を奪う上、モデルに関する広範な知識を必要とする。**コアドメイン**は、もっと見やすくしなければならない。**

コードの構造を著しく変更することは、コアドメイン**を識別するための理想的な方法だが、短期的に見ると、必ずしも実用的と限らない。実は、そういう大きなコードの変更を容易に行うには、チームでまさに欠けている見方が必要なのだ。**

モデルの構成に対する構造的な変更としては、汎用サブドメインの切り分けや本章で後述するものなどがある。こうした変更によって、モジュールに語らせることはできるようになる。しかし、コアドメインについて伝達する唯一の手段として、これをさっそく使おうというのは大胆すぎる。

こうした積極的なテクニックを補足する、軽量な解決策が必要だろう。コアを物理的に分離するのを妨げる制約があるかもしれない。あるいは、コアを十分に差別化していない既存のコードから開始したものの、より優れた蒸留へ向けて効果的にリファクタリングをするのに、コアをよく見きわめて、その見方を共有する必要が生じるかもしれない。また、さらに進んだ段階でも、実は慎重に選んだ図やドキュメントが数点あれば、チームは精神的なよりどころやエントリポイントを得られるのだ。

こういう問題は、精巧なUMLモデルを使用するプロジェクトでも、（XPプロジェクトのように）ドキュメントをほとんど外部に持たず、コードをモデルの主要なリポジトリとして使用するプロジェクトでも、同様に発生する。エクストリームプログラミングチームの方が、ミニマリズムが強く、こうした補足的なドキュメントを形式張らない一過性のもの（手描きの図を壁に貼って全員に見えるようしたものなど）にしているかもしれないが、こうしたテクニックはうまくプロセスに混ぜ込める。

モデルにおける特権的な部分に、それを具体化する実装と合わせて印をつけることは、モデルに対する考察ではあるが、必ずしもモデルそのものの一部ではない。コアドメインを全員にわかりやすくするテクニックなら何でもよいのだ。この種の解決策を提示する特

殊なテクニックが2つある。

蒸留ドキュメント

コアドメインを記述して説明するために、私は独立したドキュメントを作成することが多い。最も本質的な概念オブジェクトの一覧程度の単純なものでも構わない。そういうオブジェクトに焦点を絞り、最も重要な関係を示した図一式でもよい。基本的な相互作用のウォークスルーは、抽象レベルで行うことも、例を用いて行うこともできる。使うのは、UMLのクラス図やシーケンス図、ドメインに固有の標準的ではない図、ていねいに言葉を選んだテキストによる説明のほか、これらを組み合わせたものでもよい。**蒸留ドキュメントはすべてが揃った設計ドキュメントではない。**これは、コアに境界を引いて説明するのに必要な最低限のエントリポイントであり、特定の箇所をより詳細に精査すべき理由を提案するものでもある。これを読めば、各部分がどのように整合しているかについての広範な見方が得られ、さらに詳細を知るためにはコードの適切な部分を参照できるようになる。

それゆえ（強調されたコアの一形式として）：

簡潔なドキュメント（3～7ページ程度）を書き、コアドメインとコアを構成する要素間の主要な相互作用を記述すること。

ドキュメントを独立させることに伴う通常のリスクは、すべてここにも該当する。

1. ドキュメントが維持されないかもしれない。
2. ドキュメントが読まれないかもしれない。
3. 情報源が増えることで、複雑さを切り開くというドキュメント本来の目的が失われるかもしれない。

これらのリスクを制限する最善の方法は、ミニマリズムに徹することである。ありふれた詳細からは距離を取り、中心的な抽象とその相互作用に集中すれば、ドキュメントが古くなるのを遅らせることができる。そのレベルのモデルは安定しているのが普通だからだ。

チームの非技術系のメンバが理解できるようにドキュメントを書くこと。そのドキュメントは、誰もが知っているべきことの輪郭を描く共通の見方として、また、チームメンバ全員がモデルとコードの探究を始める際の指針として使用すること。

コアにフラグを立てる

有名な保険会社のプロジェクト初日、私が受け取った「ドメインモデル」の資料は200ページもあるドキュメントで、その業界のコンソーシアムから高額で購入したものだった。数日かけ、保険契約の詳細な構成から人間関係のきわめて抽象的なモデルまで、一切を網羅しているクラス図の寄せ集めを何とか読み終えた。これらのモデルを括り出した技術の

質は、高校生レベルから悪くないものまで多岐にわたっている（少なくともテキストによる註釈をつけて、ビジネスルールまで記述しているものもいくつかはあった）。しかし、どこから始めればよいのだろうか？ 何といっても200ページである。

このプロジェクトの文化では、抽象的なフレームワークを構築することが非常に好まれていて、私の前任者たちが焦点を合わせていたのは、人と人、人と物事、人と活動や合意といった関係性に関する非常に抽象的なモデルだった。実際、こうした関係性の分析としては優れていて、モデルを用いた実験は学術研究プロジェクト並みの品質だった。しかし、保険アプリケーションからはかけ離れていたのだ。

私の最初の直観では、切り捨てることから始めて、頼りになる小さなコアドメインを見つけてから、それをリファクタリングし、進みながら他の複雑さを再び取り込むのがよいと思われた。だが、経営陣はこの姿勢を不安がった。このドキュメントは、大変な権威をもって投資されたものである。作成には業界をあげて専門家が参加したほか、とにかくコンソーシアムに支払った金額は私に対するよりもはるかに高額だったので、私の勧める徹底的な変更が重んじられるとは思えなかった。それでも私にわかっていたのは、コアドメインに対する共通のイメージを得て、それに全員の努力を集中させなければならないということだった。

リファクタリングをする代わりに、私はドキュメントを通読し、保険業界一般と特にこれから構築するアプリケーションの具体的な要件を熟知しているビジネスアナリストの助けを借りて、わずか数節の重要な場所を特定した。これは、作業に必要な、本質的で他との差別化の要因となっている概念を示すものである。私が提供したのは、コア自体と、補助機能との関係を明確に示すモデルの道案内だったのだ。

この視点から新しいプロトタイピングの作業が開始され、必要とされる機能の一部を実演できるシンプルなアプリケーションがすぐにできあがった。

約1キロの紙資源が、わずかの付箋と黄色の蛍光ペンによってビジネスの資産に変わったのだ。

このテクニックは紙に描かれたオブジェクト図に限ったものではない。UML図を使用するチームなら、コアとなる要素を識別するのに「ステレオタイプ」を広く使用できる。コードをモデルの唯一のリポジトリとして使用するチームであれば、おそらくJavadocとして構造化されたコメントを使用したり、開発環境のツールを使用したりできる。コアドメインに入るものと入らないものを楽に見分けられさえすれば、具体的なテクニックは問題ではない。

それゆえ（強調されたコアのもう1つの形式として）：

モデルの主要なリポジトリ内においてコアドメインの各要素にフラグを立てること。ただし、その役割をことさら明らかにしようとしなくてよい。コアに入るものと入らないも

のを開発者が楽にわかるようにすること。

これで、コアドメインはモデルを使って作業する人々にとって明確となり、かなり少ない労力とわずかの保守で、少なくとも各部分がどう貢献しているかを十分区別できるくらいには、モデルを細かく括り出せるようになる。

プロセスツールとしての蒸留ドキュメント

理論上、XPプロジェクトでは、どのペア（一緒に仕事をする2人のプログラマ）も、システムのあらゆるコードを変更できる。だが現実には、変更によっては大きな影響を与える可能性があって、協議と調整が必要となることもある。インフラストラクチャ層で作業しているならば、変更が与える影響は明確だろうが、典型的な構成のドメイン層ではそこまで明白ではないかもしれない。

コアドメインの概念を使えば、この影響は明確化できる。コアドメインにあるモデルを変更すれば大きな影響があるはずだ。広範に使用されている汎用的な要素を変更すると、大量のコードを更新しなければならないかもしれないが、コアを変更する時のような概念上の転換は生じない。

蒸留ドキュメントを指針として使用すること。コードやモデルの変更と同期させるため、蒸留ドキュメントそのものを変更しなければならないと開発者が気づいた場合には、協議が必要になる。考えられるのは、コアドメインの要素か関係性を根本的に変更しているか、もしくは何か異質なものを含めるか除外しているせいでコアの境界を変更しているかのいずれかである。モデル変更をチーム全体に広めることが必要であり、やり方は蒸留ドキュメントの新バージョンの配布を含め、チームが使用するどんなコミュニケーションチャネルを用いても構わない。

蒸留ドキュメントがコアドメインの本質を概説しているなら、それはモデルを変更する意義を示す、実用的な指標としての役割を果たす。モデルやコードの変更が蒸留ドキュメントに影響を与える場合には、他のチームメンバとの協議が必要である。変更したらチームメンバ全員に速やかに通知し、蒸留ドキュメントの新バージョンを配布すべきだ。コアの外に対して行われる変更や、蒸留ドキュメントに含まれていない詳細に対する変更であれば、協議も通知もなしで統合することができ、他のメンバは作業の過程で気づくことになる。そうなれば、開発者はXPが提唱する完全な自律を手に入れることになる。

＊　＊　＊

ドメインビジョン声明文と強調されたコアは、情報を伝えて道案内をするが、モデルやコードそのものを実際に変更することはない。汎用サブドメインを切り分けると、気を散らす要素をいくつか物理的に除去できる。次に挙げるパターンでは、モデルと設計自体を構造的に変更して、コアドメインをより見えやすく扱いやすいものにする方法を見ていく…。

凝集されたメカニズム（COHESIVE MECHANISMS）

カプセル化の仕組みはオブジェクト指向設計にとって標準的なプラクティスである。複雑なアルゴリズムを、意図の明白な名前がついたメソッドに隠蔽することで、「何が」（what）が「どのように」（how）から分離される。このテクニックによって、設計は理解するのも使用するのもシンプルになる。それでも、自然と限界には行き着くものだ。

処理が設計を肥大させ始めるくらい複雑なレベルに到達することもある。概念的な「何が」が、機械的な「どのように」によって浸食されてしまうのだ。問題を解決するためのアルゴリズムを提供するメソッドが大量に存在してしまうと、問題を表現するメソッドがわかりにくくなる。

手続きがこうして増殖するのは、モデルに問題があることの兆候である。より深い洞察へ向けたリファクタリングが生み出すモデルと設計の持つ要素の方が、そうした問題を解決するには適しているかもしれない。最初に探すべき解決策は、処理のメカニズムを単純にするモデルである。だが時には、処理のメカニズムの一部が、それ自体で概念的に凝集しているのだという洞察が現れることがある。この概念的な処理は、実際に必要となる厄介な処理をすべて含んでいるとは限らない。我々は、あらゆることに対応できる「計算機」の類の話をしているのではない。だが、凝集した部分を抜き出せば、残りのメカニズムが理解しやすくなるはずである。

それゆえ：

概念的に凝集されたメカニズムを切り分けて別個の軽量なフレームワークにすること。特に、形式主義や十分にドキュメント化された種類のアルゴリズムに気を配ること。意図の明白なインタフェースを用いてフレームワークの機能を他から使えるようにすること。これで、ドメインの他の要素は問題を表現すること（「何が」）に集中でき、複雑な解決（「どのように」）はフレームワークに委譲できる。

このような分離されたメカニズムは補助的な役割に置かれ、後に残される小さく表現力豊かになったコアドメインは、以前よりも宣言的なスタイルになったインタフェース経由でそのメカニズムを使用する。

標準的なアルゴリズムや形式主義を識別することにより、設計にあった複雑さの一部が取り除かれ、すでに研究された概念の集まりになる。そうした指針があれば、多少試行錯誤をするだけで、自信を持って解決策を実装できる。他の開発者についても、その指針を知っているか、少なくとも調べることはできると考えられる。これは公表された汎用サブドメインモデルの利点と似ているが、ドキュメント化されたアルゴリズムや形式化された処理の方がより頻繁に目にされる。このレベルのコンピュータ科学の研究の方が進んでいるからだ。それでも、少なからず何か新しいものを作成しなければならない。処理だけに焦点を絞り、表現力豊かなドメインモデルを混ぜないようにすること。ここに、責務の分

離が存在する。すなわち、コアドメインや汎用サブドメインのモデルはファクトやルール、あるいは問題を公式化する。それに対して、凝集されたメカニズムは、モデルの定義に従い、ルールを決定したり、処理を完了させたりするのだ。

例 15.2——組織図におけるメカニズム

私がこのプロセスを経験したのは、組織図に関するかなり精巧なモデルを必要とするプロジェクトにおいてであった。このモデルが表していた事実は、ある1人が別の人のために働いているということと、それが組織のどの部門であるかということで、関連する問いによって答えが得られるインタフェースを提供していた。ほとんどの質問が「この指揮系統で、これを承認する権限を持っているのは誰か？」や、「この部門で、このような問題を処理できるのは誰か？」といったものだったため、チームは、複雑さのほとんどが、組織系統図にある特定の枝を辿り、特定の人や関係を検索することにある、ということに気づいた。これはまさしく、よく発達した形式主義であるグラフ（graph）によって解決される種類の問題である。グラフとは円弧（エッジと呼ばれる）でつながれたノードの集合であり、グラフを辿るのに必要なルールとアルゴリズムを含んでいる。

ある請負業者がグラフトラバースフレームワークを凝集されたメカニズムとして実装した。このフレームワークが使用したのは、グラフに関する標準的な用語法とアルゴリズムだった。この用語法とアルゴリズムは、ほとんどのコンピュータ科学者がよく知っていて、教科書でもふんだんにドキュメント化してあるものだ。彼が完全に一般的なグラフを実装したわけでは決してない。実装したのは、組織モデルに必要な機能を網羅した、概念的フレームワークのサブセットだった。さらに意図の明白なインタフェースもあったので、回答を得るための手段はたいして問題にならなかった。

こうして、組織モデルは標準のグラフ用語を使用して、それぞれの人がノードであることと、人と人の関係がそれらのノードをつなぐエッジ（円弧）であることをシンプルに述べられるようになった。これ以降は、グラフフレームワーク内のメカニズムによって、任意の2人の関係を検索できるはずだ。

このメカニズムがドメインモデルに組み込まれていたら、2通りのかたちでコストが生じていただろう。モデルは問題を解決する特定のメソッドと結合して、将来の選択肢を制限してしまう。さらに重要なのは、組織を表すモデルが大幅に複雑化して混乱の元となっただろうということだ。メカニズムとモデルを分離しておくことで、はるかに明確に組織を記述する宣言的スタイルが使用できるようになった。そしてグラフを操作するための複雑なコードは、実証済みのアルゴリズムに基づく、メカニズムを純粋に実装するフレームワークに切り離され、独立して保守しユニットテストを実行できる。

凝集されたメカニズムのもう1つの例としては、仕様オブジェクトを構築し、そのオブジェクトに期待される基本的な比較と結合の操作をサポートするフレームワークが挙げられる。こうしたフレームワークを採用することで、コアドメインと汎用サブドメインは、仕様を宣言するのに、仕様パターンで記述された明確なわかりやすい言語を使える（第10章参照）。比較と結合を実行するのに必要な複雑な操作はフレームワークに任せることができる。

* * *

汎用サブドメイン対凝集されたメカニズム

汎用サブドメインと凝集されたメカニズムはどちらも、コアドメインの負担を軽減したいという同じ願望に基づいている。異なっているのは、引き受ける責務の性質である。汎用サブドメインが基づいているのは、表現力の豊かなモデルであり、これはチームがドメインをどう見ているかについての、いくつかの側面を表している。その点ではコアドメインと違いはなく、ただ、やや中心からはずれ、重要度も低く、それほど特別ではないのだ。凝集されたメカニズムはドメインを表現しない。表現力豊かなモデルにより提起された、面倒な処理の問題を解決するのだ。

モデルが提案（propose）し、凝集されたメカニズムが処理（dispose）する。

実際には、形式化され公表された処理に気づくまでは、この区別は純粋ではないのが普通で、少なくとも最初のうちはそうだ。後に続くリファクタリングによって、より純粋なメカニズムに蒸留されるか、それまでは認識されていなかったモデル概念を持ち、メカニズムを単純にするような汎用サブドメインに変換される。

メカニズムがコアドメインの一部である場合

コアドメインからメカニズムを除去したいという欲求は常にあるものだ。唯一の例外は、メカニズムそのものが独占的なもので、ソフトウェアの価値を構成する主要な部分になっている場合である。これはきわめて特殊なアルゴリズムなどで該当することがある。例えば、輸送手配アプリケーションの持つ特徴的な機能の1つが、スケジュールを計画するための特別に効率的なアルゴリズムだとすると、そのメカニズムは概念的コアの一部と見なせる。私は以前、投資銀行のプロジェクトに従事したことがあるが、そこにあった、リスクを評価するための極度に独占されたアルゴリズムは、間違いなくコアドメインにあった（実際、それらのアルゴリズムは非常に厳重に保管されていて、コアの開発者さえもほとんどが見ることを許されなかった）。もちろん、こういうアルゴリズムは、実際にリスクを予想するルールの集合を特別に実装したものだろう。より深く分析すれば、そうしたルール

を明示的にでき、カプセル化された解決メカニズムを持つ深いモデルが生み出されるかもしれない。

しかし、それは設計に対する段階的な改良の1つで、日を改めて行うものだろう。その次の段階に進むかどうかの決定は、費用対効果の分析に基づいて行われる。その新しい設計を考え出すのはどれくらい難しいか？ 現在の設計を理解して変更するのはどれくらい難しいか？ より進んだ設計だと、実際に作業すると思われるタイプの人を想定した時に、どれくらい作業が楽になるか？ そして当然ながら、新しい設計がどんなかたちになるか見当がつく人がいるのか？

例 15.3——一巡：組織図がメカニズムを再び吸収する

実際には、前述の例にあった組織モデルを完成させた1年後、他の開発者たちはそれを設計し直して、グラフフレームワークを分離するのをやめた。増加したオブジェクト数と、メカニズムを独立したパッケージに分離する複雑さが許し難いと感じたのである。代わりに、組織エンティティの親クラスにノードとしてのふるまいを追加した。ただし、組織モデルにあった宣言的な公開インタフェースは残した。メカニズムがカプセル化された状態は、組織エンティティの中で保ったのである。

このように一巡することはよくあるが、出発点に戻っているのではない。最終的な結果は、通常、より深いモデルとなって、ファクトや目標、そしてメカニズムを明確に他のものと区別できるのである。実践的なリファクタリングによって、中間段階にある重要な価値は維持しながら、不要な複雑さを捨て去るのだ。

蒸留して宣言的スタイルにする

宣言的設計と「宣言的スタイル」は第10章の主題だったが、その設計スタイルは、戦略的蒸留を扱う本章でも、特別に言及するに値する。蒸留することの価値は、何をしているかを見えるようにすることにある。つまり、本質と関係ない詳細に気をそらされることなく、本質に切り込むことにあるのだ。コアドメインの重要な部分が宣言的スタイルに従うことができるのは、補助的な設計によってコアにある概念とルールを表現するための効率的な言語が提供され、一方で概念を処理し、ルールを強制する手段がカプセル化されている場合である。

凝集されたメカニズムが最も役立つものとなるのは、意図の明白なインタフェースによるアクセスが提供され、それが概念的に一貫性のある表明と副作用のない関数を持っている場合である。メカニズムとしなやかな設計により、コアドメインは、はっきりしない関数を呼び出すのではなく、意味のある文を作成できるようになる。しかし、例外的な結末を

迎えることもある。コアドメイン自体の一部が、ブレイクスルーの結果、深いモデルに至り、アプリケーションにおける最も重要なシナリオを柔軟かつ簡潔に表現できる言語として機能し始めることもあるのだ。

深いモデルには、それに対応するしなやかな設計がついてくることも多い。しなやかな設計が成熟すると、理解しやすい要素の集合が提供され、それをわかりやすく結合することで、複雑なタスクを達成したり、複雑な情報を表現したりできるようになる。ちょうど単語をつなげて文章にするようなものだ。その時点で、クライアントのコードは宣言的スタイルを身につけ、さらなる蒸留が可能になる。

汎用サブドメインを括り出すことで雑音が減り、凝集されたメカニズムによって複雑な操作がカプセル化される。後に残るのはより焦点の絞られたモデルであり、ユーザの活動に何ら価値を加えることのない、注意を散漫にする要因は減る。しかし、コアでないもの**すべて**について、ドメインモデルの中に落ち着く場所が見つかるわけでもなさそうだ。次に取り上げる隔離されたコアは、コアドメインを構造的に区切る直接的なアプローチをとる…。

隔離されたコア（SEGREGATED CORE）

モデル内にある複数の要素は、一部がコアドメインに仕え、一部が補助的な役割を果たすかもしれない。コアに仕える要素が、汎用的な要素と密接に結合していることもある。コアの概念上の凝集は、強くもなく可視的でもないかもしれない。こうした雑音ともつれによって、コアの発達が妨げられる。設計者は最も重要な関係性を明確に理解することができず、貧弱な設計が作られることになる。

汎用サブドメインを括り出すことで、ドメインからはっきりしない詳細の一部が取り払われ、コアをより見やすくできる。しかし、そうしたサブドメインをすべて識別し明確化するのは困難な作業であり、中にはそこまでする価値がないものもある。他方、最も重要なコアドメインは残留物と絡まったまま放置される。

それゆえ：

モデルをリファクタリングして、コアの概念を補助的な役割を果たすもの（はっきりと定義されていないものも含む）から分離すること。そうすることにより、コアの凝集度を高めつつ、他のコードへの結合を低くすること。汎用的な要素や補助的な要素はすべて別のオブジェクトに括り出し、それを別のパッケージに入れること。モデルをリファクタリングした結果、密接に結合している要素を分離することになるとしても、やらなければならないのだ。

これは、汎用サブドメインに適用したのと基本的には同じ原則だが、方向性が異なる。アプリケーションの中心である凝集されたサブドメインは、識別して一貫性のあるパッケージに分割できる。後に残るどこにも入らない塊をどうするかは、どうでもよいというわけではないが、そこまで重要でもない。おおよそ元の場所に残してもよいし、主要なクラスに基づいてどこかのパッケージに入れてもよい。最終的に、汎用サブドメインに括り出せる残留物が増えていくが、短期的にはどんな簡単な解決策でも、隔離されたコアへの集中が保たれればよい。

＊ ＊ ＊

リファクタリングして隔離されたコアを作るのに必要なステップは、典型的には次のようなものである。

1. コアサブドメインを識別すること（蒸留ドキュメントから引き出すこともできる）。
2. 関連するクラスを、それを関係づけている概念に由来する新しいモジュールへ移動すること。
3. コードをリファクタリングして、その概念を直接表現していないデータと機能を切り離すこと。取り除いた側面は他のパッケージにある（たぶん新しい）クラスに入れること。概念的に関連するタスクと一緒にすべきだが、完璧を期すことにあまり時間を

浪費してはならない。コアサブドメインから不純物を除去してきれいにし、他のパッケージへの参照を、明示的で意図の表現されたものにするよう専念すること。

4. 新しい隔離されたコアのモジュールをリファクタリングして、関係と相互作用をより単純で伝達力のあるものにし、他のモジュールとの関係を最小限におさえて明確化すること（これは継続するリファクタリングの目標になる）。
5. 隔離されたコアが完成するまで、別のコアサブドメインに対して前述のステップを繰り返すこと。

隔離されたコアを作成するコスト

コアを隔離すると、コアと密接に結合した、コアではないクラス群との関係が、さらにあいまいで複雑になることがあるが、そのコストよりコアドメインを明確化して作業しやすくする利点の方が勝っている。

隔離されたコアにより、そのコアドメイン内での凝集度を高められるようになる。モデルを分割する有意義な方法は数多くあり、隔離されたコアを作成したために、うまく凝集されていたモジュールが分割されることもある。その凝集を犠牲にする代わりにコアドメインの凝集度が高められるのだ。これは純粋な利益である。エンタープライズソフトウェアにおける最大の付加価値は、モデルにあるエンタープライズに特化した側面から得られるからだ。

もう1つのコストは、言うまでもなく、コアを隔離するのは大変な作業だということだ。隔離されたコアへ向かうと決定することで、システム全体に加えられる変更に開発者たちを取られてしまう可能性があることは認識しておくべきだ。

隔離されたコアを切り出すべきタイミングは、システムにとって重要な、巨大な境界づけられたコンテキストがあり、モデルの本質的な部分が大量の補助的な機能のせいでわかりにくくなってきた時である。

チームの意思決定を進化させる

戦略的設計における多くの意思決定と同様に、ここでもチーム全体が隔離されたコアに向かって一丸となって進まなければならない。この段階には、チームにおける意思決定のプロセスと、決められたことを実行できるだけの規律と協調性があるチームが求められる。ここでの課題は、コアに対して同じ定義を用いることを全員に強制しながら、その決定を凍結させないようにすることである。コアドメインは、設計のその他の側面と同じように進化するものだから、隔離されたコアで作業する経験は、何が本質的で何が補助的な要素かについての新しい洞察につながる。そういう洞察は、コアドメインや隔離されたコアの

モジュールに対する定義を改良するためにフィードバックしなければならない。

これが意味しているのは、新しい洞察は、チームで継続的に共有する必要があるものの、個人（またはプログラミングペア）がそれに基づいて一方的に行動することはできない、ということだ。共同での意思決定がどのようなプロセスでなされたものであっても、つまりそれが合意による決定であれ、チームリーダによる命令であれ、進むべき道を繰り返し修正できるくらいに俊敏でなければならない。コアに対しては全員が同じ見方をするくらい、コミュニケーションは効果的でなければならないのだ。

例 15.4 ——貨物輸送モデルのコアを隔離する

貨物輸送を調整するソフトウェアの土台として、図15.2に示すモデル図から始める。

現実のアプリケーションで必要になると思われるものと比べると、大幅に簡略化してあることに注意して欲しい。現実的なモデルは、例とするにはあまりにも厄介なものになるだろう。そこで、この例は隔離されたコアに駆り立てるほどには入り組んでいないかもしれないが、想像力を飛躍させて、このモデルが非常に複雑で容易に解釈したり全体として処理したりできないものとして扱って欲しい。

さて、輸送モデルの本質とは何だろう？ 通常、まず見るべきは「最終的な収益」である。そうだとすると、価格決定と請求書に集中することになる。しかし、本当に調べるべきはドメインビジョン声明文である。その一部を抜粋しよう。

> …業務の可視化をさらに進め、顧客の要件を、より迅速かつ確実に満たすツールを提供する…。

このアプリケーションは営業部門向けに設計されたものではない。使用するのは、現場にいるその会社の担当者である。そこで金銭に関連する問題はすべて、（明らかに重要ではある）補佐役に格下げしよう。すでに、誰かがこれらの問題の一部を独立したパッケージ（**請求(Billing)**）に入れている。それはそのままにしておけばよく、さらには、補助的な役割を果たしていることもわかるようになっている。

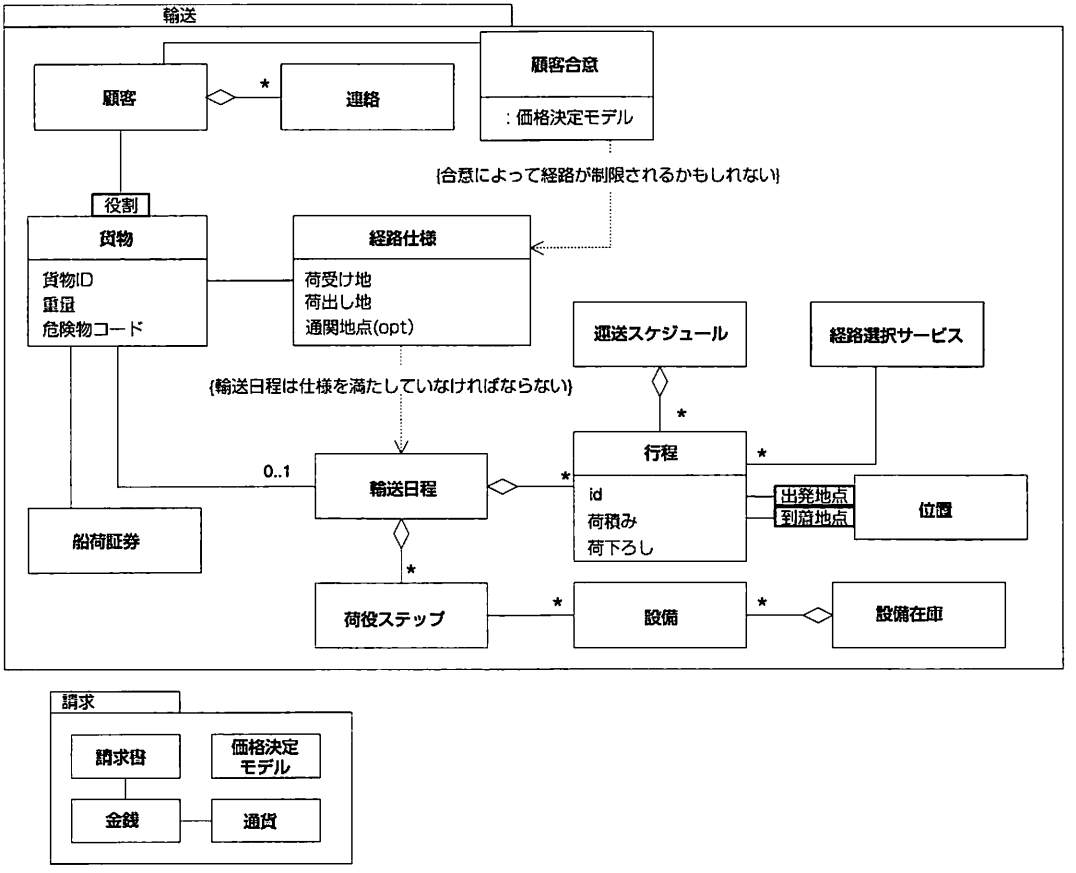


図15.2

焦点を合わせるべきなのは、貨物の取り扱い、すなわち、顧客の要件に従って貨物を配送することだ。これらの活動と最も直接的に関係するクラスを引き出すと、図15.3に示すような、**配送（Delivery）**という新しいパッケージに、隔離されたコアができる。

大部分については、クラスが新規のパッケージへ移されただけだが、モデルそのものにもいくつか変更がある。

まず、**顧客合意**が**荷役ステップ**を制約するようになった。これは、チームがコアを隔離するにしたがって発生する傾向がある洞察の典型である。効率的で正確な配送に注意が集まるにつれ、**顧客合意**の中にある配送に対する制約が本質的であり、モデルで明示的にしなければならないことが明らかになってくる。

●例15.4——貨物輸送モデルのコアを隔離する

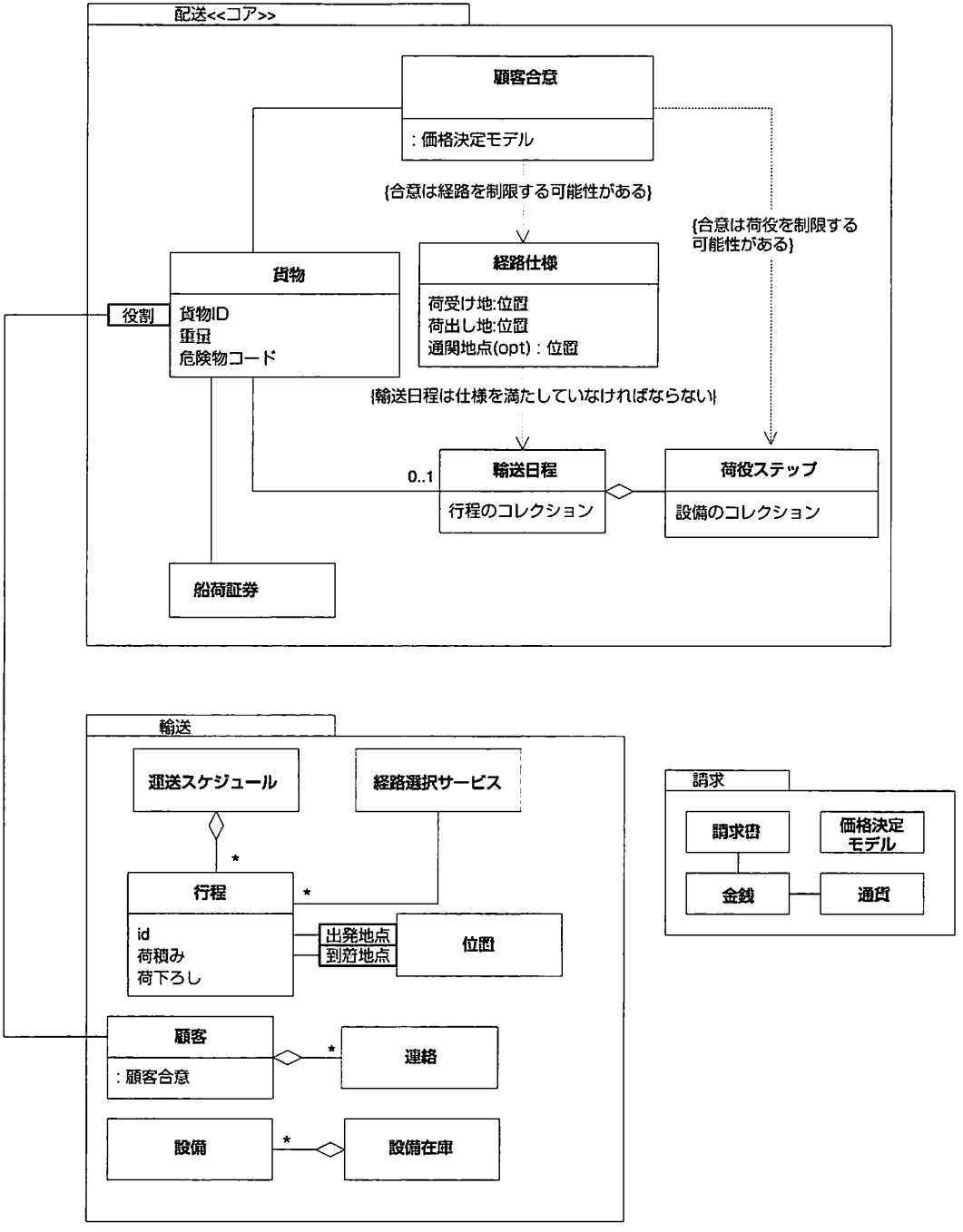


図15.3 顧客要件に忠実な、確実な配送が、このプロジェクトのコアとなる目標である

もう1つの変更はより実践的である。リファクタリング後のモデルで、**顧客合意**は直接貨物に結びつけられていて、**顧客**を経由する必要がない（**顧客**と同様、**貨物**の予約時に結びつけなければならない）。実際の配送時には、業務にとっては**顧客**よりも合意そのものの方が関係深いのだ。別のモデルでは、**輸送**において果たす役割に従って正しい**顧客**を検索してから、その**顧客合意**に対して問い合わせなければならなかった。この相互作用があると、モデルについて説明しようと準備していたことがすべて滞っていただろう。新しい関連のおかげで、最も重要なシナリオが、できる限り単純で直接的なものになっている。こうして、**顧客**をコアから完全に抜き出すことが容易になった。

とにかく**顧客**を抜き出してみるというのはどうなのだろうか？ 焦点は**顧客**の要件を満たすことにあるので、最初は**顧客**がコアに属するように見える。だが、配送中に行われる相互作用では、通常、**顧客**クラスを含める必要がなく、すでに**顧客合意**が直接使用できるようになっている。しかも、**顧客**の基本的なモデルはきわめて汎用的である。

行程をコアに残すかどうかに関しては激しい議論が行われるだろう。私はコアに関してはミニマリストになりがちであるし、また**行程**と高い凝集度で結びつく**運送スケジュール**や**経路選択サービス**、**位置**は、どれもコアにない。しかし、図15.3のこのモデルについて説明したいと思う内容の多くに**行程**が含まれるのであれば、**配送**パッケージへ移動して、関係する他のクラスから分離することによるぎこちなさは、甘んじて受け入れるだろう。

この例ではすべてのクラス定義が以前と同じだが、蒸留することにより、クラス自体をリファクタリングして、汎用的な責務とドメインに特化した責務を分離し、隔離できるようにすることが求められることもある。

隔離されたコアができたところで、リファクタリングは完了である。しかし、**輸送**パッケージには「コアを抜き出した後に残されるすべて」が残ることになる。さらにリファクタリングを行うことで、図15.4に示すような、より多くを伝えるパッケージングが得られる。

そこまで到達するにはさらに何度かリファクタリングが必要になるかもしれないが、すべてを一度に済ませなくてもよい。ここで最終的にできたのは、隔離されたコアパッケージが1個、汎用サブドメインが1個、補助的役割のドメインに特化したパッケージが2個である。より深い洞察が得られれば、最終的に顧客の汎用サブドメインができるか、**顧客**がさらに輸送に特化したものになるだろう。

●例 15.4 ——貨物輸送モデルのコアを隔離する

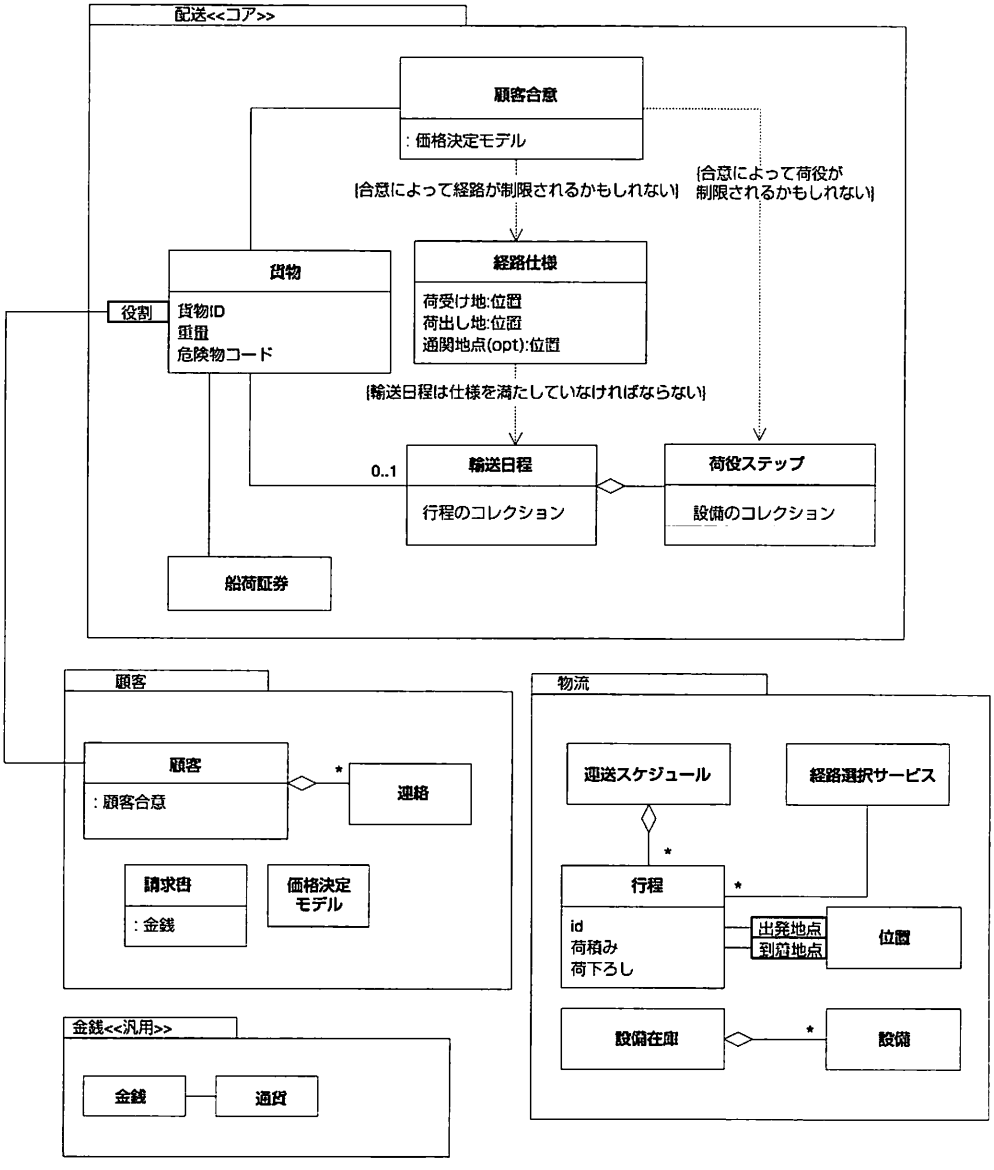
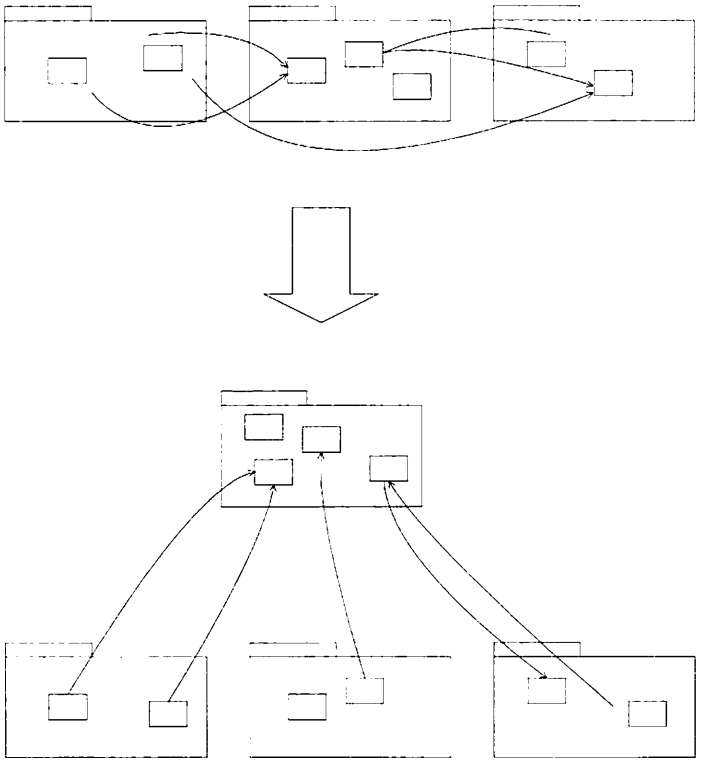


図 15.4 **隔離されたコア**の完成後に非**コア**サブドメインの有意義な**モジュール**化が続く

役に立つ有意義なモジュールを認識することが、モデリングという活動である（第5章で述べた通り）。開発者とドメインエキスパートは、知識をかみ砕くプロセスの一環として、協力して戦略的蒸留を行う。

抽象化されたコア（ABSTRACT CORE）



コアドメインのモデルにさえ、通常は非常に多くの詳細があり、全体像を伝えることは難しいかもしれない。

＊　＊　＊

通常、巨大なモデルを扱う時には、把握できるくらい小さい限定的なサブドメインに分割して、それぞれ別のモジュールに入れる。こうした減らしていくスタイルのパッケージングは、しばしば、複雑なモデルを扱いやすくするのに役立つ。しかし、別々のモジュールを作成することで、サブドメイン間の相互作用があいまいになったり、かえって複雑になったりすることもある。

別々のモジュールに置かれたサブドメインの間で大量の相互作用があると、モジュール間で多数の参照を生成する必要が生じて、分割したことの価値をほとんどなくしてしまうか、相互作用を間接的に行わなければならなくなって、モデルがあいまいになるかのいず

れかとなる。

分割する線を垂直にではなく、水平に引くことを検討すること。多態性によって、抽象型のインスタンス間にある詳細なバリエーションの多くを無視できるようになる。モジュール間の相互作用のほとんどが、多態的なインタフェースのレベルで表現できれば、それらの型をリファクタリングして、特別なコアモジュールに入れるのは意味のあることかもしれない。

小細工をするためのテクニックを見つけようというのではない。このテクニックに価値が出るのは、多態的なインタフェースがドメインにおける根本的な概念と一致する時だけだ。その場合には、この抽象を分離することによって、モジュールが疎結合となる一方で、より小さく凝集度の高いコアドメインが蒸留される。

それゆえ：

モデルにおける最も根本的な概念を識別し、それを別のクラス、抽象クラスまたはインタフェースに括り出すこと。この抽象的なモデルは、重要なコンポーネント間の相互作用をほとんど表現するように設計すること。この抽象的なモデル全体を独自のモジュールに入れること。ただし、特殊で詳細な実装クラスは、サブドメインによって定義されたモジュールに残すこと。

これで特殊なクラスのほとんどが、他の特殊なモジュールではなく、抽象化されたコアモジュールを参照するようになる。抽象化されたコアにより、主要な概念とその相互作用について簡潔な見方ができるようになる。

抽象化されたコアを括り出すプロセスは、機械的なものではない。例えば、モジュールをまたいで頻繁に参照されるクラスを、すべて別のモジュールへ機械的に移動したら、結果として予想されるのは意味のないひどい塊だろう。抽象化されたコアをモデリングするには、主要な概念とシステムの重要な相互作用において、その概念が果たす役割についての深い理解が必要となる。つまりこれは、より深い洞察に向けたリファクタリングの一例なのだ。しかも、通常は相当な量の再設計が必要となる。

抽象化されたコアは、最終的に蒸留ドキュメントに非常に似たものになるはずだ（両者が同じプロジェクトで使用され、洞察が深まるにつれて、アプリケーションと一緒に蒸留ドキュメントが進化した場合）。もちろん、抽象化されたコアはコードに書かれることになるので、より厳密で完全なものになる。

＊ ＊ ＊

深いモデルの蒸留

蒸留は、コアからドメインの一部を切り離すという全体的なレベルだけで行われるのではない。サブドメイン、とりわけコアドメインを改良することも意味している。それは深い洞察に向かう継続的なリファクタリングによって行われ、深いモデルやしなやかな設計へと駆り立てる。目標は、モデルを明らかにする設計であり、ドメインをシンプルに表現するモデルである。深いモデルは、ドメインにおける最も本質的な側面を蒸留してシンプルな要素にするが、この要素は組み合わされて、アプリケーションの重要な問題を解決できる。

深いモデルを生み出すブレイクスルーはどこで発生しても価値を生むが、プロジェクト全体の軌道を変えられるのは、それがコアドメインにおいて発生した時だけだ。

リファクタリングの対象を選ぶ

うまく分解されていない巨大なシステムに遭遇したら、どこから手をつければよいだろう？XPコミュニティにおいて、その答えは以下のどちらかになる傾向がある。

1. すべてリファクタリングが必要なのだから、どこから着手しても構わない。
2. どこであれ困っているところから始める。自分の作業をかたづけるのに必要なものをリファクタリングする。

だが、私はどちらも支持しない。前者は、トッププログラマだけで構成されたごくわずかなプロジェクトを除いて、現実的ではない。後者は、重箱の隅をつつくことになりやすく、対症療法はしても根本原因を無視し、最もひどくもつれた場所を敬遠してしまう。結局、コードはどんどんリファクタリングしにくくなる。

では、すべてには手が回らず、かといって困った時だけ対応することもできない場合、どうすればよいだろうか？

1. 困った時のリファクタリングをする際に、原因がコアドメインかコアと補助的要素との関係を含むものかどうかを調べる。そうであれば、歯を食いしばって、まずそれを修正すること。
2. 自由にリファクタリングする余裕がある場合、最初に集中すべきなのは、コアドメインのより適切な括り出し、コアの隔離の改善、補助的なサブドメインから不純物を取り除くことによる汎用サブドメイン化である。

これがリファクタリングという出費によって、最大の効果を上げる方法なのだ。

第16章

大規模な構造



何千という人々が独立して作業し、AIDSメモリアルキルトを作り上げた。

シリコンバレーの小さい設計会社が衛星通信システム用シミュレータの作成を請け負った。仕事は順調に進んでいる。モデル駆動設計を用いて開発されていたのは、広範囲にわたってネットワークの状況と障害を示してシミュレートできるソフトウェアだった。

しかし、プロジェクトの開発リーダは不安だった。この問題はもともと複雑である。モ

デルにある複雑な関係性を明確化する必要に駆られ、設計を扱いやすい大きさの凝集したモジュールに分解した。それによって、モジュールが大量にできた。機能の特定の側面を見つけるために、開発者はどのパッケージを調べればよいのか？ 新しいクラスはどこに入れるべきか？ これらの小さいパッケージのいくつかは、実際は何を意味しているのか？ 全体として、どうまとまるのか？ その上、まだ構築しなければならないものもある。

開発者は互いに十分コミュニケーションをとっていて、日々何をすべきかを見つけ出すことはまだできたが、プロジェクトリーダは、わかりやすくしたことによる強みが活かされていないことが不満だった。複雑さのレベルが上がるにつれ、設計を組織化して、理解し操作できるようにする方法が求められた。

ブレインストーミングを行ったところ、さまざまな可能性が提示された。今あるものに代わるパッケージ構成がいくつも提案された。何らかのドキュメントがあればシステムの概観を示すことができるかもしれないし、モデリングツールにあるクラス図の新しいビューを使えば、開発者を導いて正しいモジュールが作れるかもしれない。しかし、このプロジェクトリーダは、こうした仕掛けで満足できなかった。

シミュレーションについてシンプルに説明することはできた。内容としては、インフラストラクチャを通じてデータを整理する方法や、遠隔通信技術のレイヤによって保証されているデータの整合性とルーティングなどである。だが、この話に出てくる詳細はすべてモデルの中にあるにも関わらず、ストーリーの太い輪郭が見えてこないのだった。

実は、ドメインに由来する本質的な概念で、足りないものがあった。ただし、オブジェクトモデルにクラスが1、2個欠けていたということではない。欠けていたのは、全体としてのモデルに対する構造だったのだ。

開発者たちがこの問題について1、2週間かけて検討した結果、考えが具体化し始めた。設計が従うべき構造を規定することにしたのだ。シミュレータ全体は、通信システムという側面に関係する一連のレイヤと見なされる。最下層が表現するのは、物理的インフラストラクチャ、つまりノード間でビットを伝送する基本能力である。次に来るのがパケット経路選択層で、特定のデータストリームが向かう先をどう決めるかという関心事をまとめる。その他のレイヤは問題の中にある他の概念レベルをそれぞれ識別する。**これらのレイヤは、システムについて彼らが語る話の概要となっていた。**

開発者たちはコードのリファクタリングに着手し、新しい構造に順応させようとした。モジュールはレイヤをまたがないように定義し直さなければならない。場合によっては、オブジェクトの責務もリファクタリングされ、各オブジェクトが明確に1つの層へ属するようになった。逆に、このプロセスを通じて、概念的レイヤの定義自体もそれを適用する実際の経験に基づいて改良された。レイヤ、モジュール、オブジェクトが共に進化し、ついには設計全体がこのレイヤ化構造の輪郭に沿うようになった。

こうしたレイヤはモジュールでもなければ、コードに見られるその他の成果物でもない。包括的なルールの集合であり、設計全体を通じて、特定のモジュールやオブジェクトの境界と関係性を制約するのだ。このレイヤによる制約は、他システムとのインタフェースにさえも及ぶ。

この秩序を課したことで、設計は再び、十分に理解できるようになった。特定の機能を探すべき場所も、大体はわかるようになった。独立して作業する各個人が、おおよそ相互に矛盾しないで設計上の意思決定を行えるようになった。複雑にできる限度が高められたのだ。

モジュール化による分割を行っても、巨大なモデルは複雑すぎて理解できないことがある。モジュールによって、設計は扱いやすい欠片になるが、その数が多いこともある。また、モジュール化によって、必ずしも設計に統一性がもたらされるとは限らない。複数のオブジェクトや複数のパッケージを通じて、それぞれ理由は説明できても、独特で脈絡のない設計上の意思決定がなされるかもしれない。

境界づけられたコンテキストによって厳格に分離されることで、崩壊と混乱は予防される。しかし、境界づけられたコンテキストがそれ自体で全体としてのシステムを見やすくするわけではない。

蒸留することによって、確かに注意はコアドメインに集中し、他のサブドメインには補助的な役が割り当てられることになる。しかし、相変わらず理解する必要があるのは、補助的な要素とコアドメインに対する関係、さらに補助的な要素同士の関係である。コアドメインが非常に明確で理解しやすく、手引きをつけ加える必要がないというのは理想的であるが、我々は常にそういう状況にあるわけではない。

プロジェクトの規模に関わらず、システムの別々の部分に対しては、何らかの独立した取り組みが行われなければならない。調整やルールがなければ、さまざまなスタイルや、同じ問題に対する別々の解決策による混乱が生じる。そうなると、各部分がどう整合しているのかが理解しにくくなり、全体像が見えなくなる。設計のある部分について何かを学んでも、それを別の部分に転用することができないから、プロジェクトはいろいろなモジュールの専門家を抱えることになるが、そうした専門家は自分の担当する狭い範囲の外では、互いに助け合うことができない。継続的な統合は崩壊し、境界づけられたコンテキストが断片化されてしまう。

巨大なシステムに包括的な原則がなく、そのせいで各要素を解釈する際に、設計全体にまたがるパターンにおいてどのような役割を果たすかという観点から考えることができなければ、開発者は「木を見て森を見ず」になってしまう。全体の詳細を徹底的に調べなくても、全体の中で個々の部分が果たす役割を理解できる必要があるのだ。

「大規模な構造」は、システムをおおよその構造から議論し、理解できるようにするための言語である。高次の概念の集合やルールの集合、もしくはその両方によって、システム全体に対する設計パターンが確立される。この構成原理は理解を助けることに加え、設計の指針ともなり得る。独立した作業を調整することにも役立つのだが、それは全体像に関する概念が共有されているからだ。すなわち、さまざまな部分の果たす役割が、どのように全体を構成するかについての概念が共有されているのだ。

ルールについてのパターンや、役割や関係性についてのパターンを考え出すこと。このパターンはシステム全体に行きわたる。また、このパターンがあることで、全体における各部分の位置づけが、その部分の責務を詳細に知らなくても理解できるようになる。

構造は1つの境界づけられたコンテキスト内に限られることもあるが、通常は複数のコンテキストにまたがっていて、プロジェクトに携わるすべてのチームとサブシステムを1つにまとめる概念的な構成を提供する。優れた構造は、モデルについての洞察を与え、蒸留を補完するのだ。

ほとんどの大規模な構造はUMLで表現することができず、またその必要もない。ほとんどの大規模な構造はモデルと設計を形作り説明するものの、その中には現れない。提供されるのは、設計に関する別のレベルのコミュニケーションなのだ。本章の例では、略式のUML図を多数挙げているが、私はそこに大規模な構造についての情報を重ね合わせている。

チームが適度に少人数でモデルが複雑すぎなければ、ふさわしい名前のモジュールに分解し、ある程度の蒸留を行って、開発者間で非公式に調整することで、モデルが組織化された状態を十分に保てる。

大規模な構造はプロジェクトを救うこともできるが、構造がうまく合わないと開発を著しく妨げることにもなる。本章では、このレベルの設計をうまく構造化するためのパターンを見ていく。

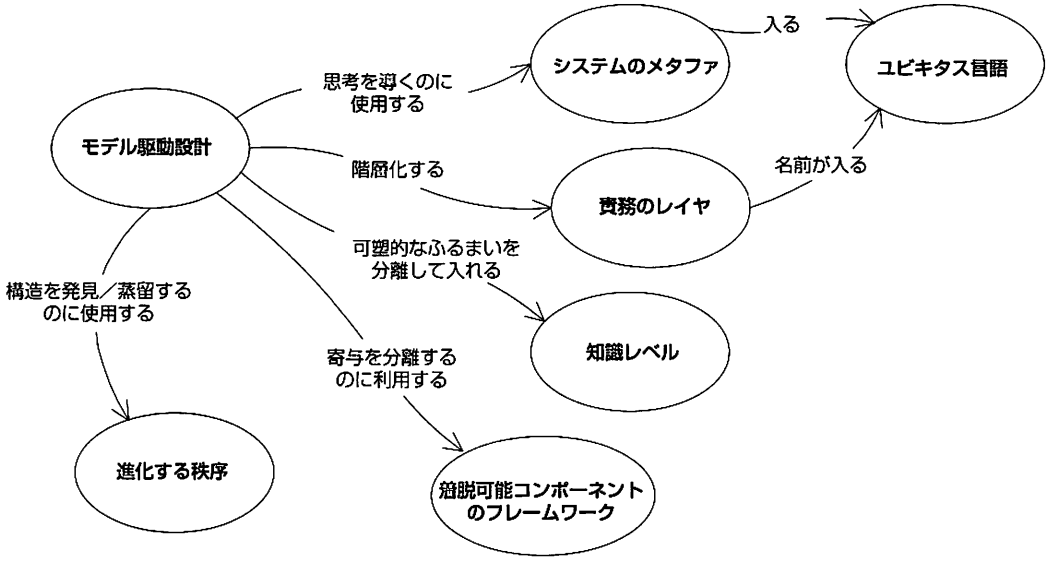


図 16.1　大規模な構造のパターンの一部

進化する秩序（EVOLVING ORDER）

多くの開発者が、設計が構造化されていないことによる損失を経験している。無秩序を避けるために、プロジェクトはアーキテクチャを課して、開発をさまざまな方法で制約する。技術的なアーキテクチャによっては、ネットワークやデータの永続化といった技術的な問題を解決するものも確かにあるが、アーキテクチャがアプリケーションやドメインモデルの領域に乗り出すと、アーキテクチャ自体によって問題が作り出されることがある。アーキテクチャのせいで、問題の細かい仕様に対してうまく機能するような設計とモデルを、開発者が作れなくなってしまうのだ。きわめて野心的なアーキテクチャは、アプリケーション開発者からプログラミング言語そのものが持つ、親しみやすさと技術的な能力を奪ってしまうかもしれない。技術指向であれドメイン指向であれ、アップフロントに行われる設計上の意思決定について、その多くを変えられなくしてしまうアーキテクチャは、要件が変化し、ドメインに対する理解が深まるにつれ、開発者を拘束しかねない。

技術的なアーキテクチャ（J2EEなど）によっては、ここ数年で有名になったものもあるが、ドメイン層における大規模な構造はまだあまり探究されていない。アプリケーションによって、要求が大幅に異なるからだ。

大規模な構造をアップフロントに課すのは高くつきそうだ。開発が進むにつれて、より適した構造が見つかるのはほぼ確実だが、規定された構造のせいで、アプリケーションを大幅に明確化し単純化するように設計できなくなるかもしれないのだ。いくつかの構造は使用できるかもしれないが、多くの機会を捨ててしまっている。次善策を試したり、アーキテクトと交渉したりするため作業は遅くなる。それでも、マネージャはアーキテクチャが完成していると考える。アーキテクチャの目的はこのアプリケーションを簡単にすることだったのだから、そんなアーキテクチャの問題に関わっていないで、アプリケーションに取り組めばいいではないか？ マネージャとアーキテクチャチームが自分とは違う意見に寛容だったとしても、変更のたびに熾烈な戦いになっていたら、疲れ果ててしまう。

設計に制約がなければ、できあがるシステムの全体像が誰にも理解できない上に、保守するのも非常に困難になる。しかし、アーキテクチャも、アップフロントな設計上の想定によってプロジェクトを束縛し、アプリケーションの特定の部分を受け持つ開発者や設計者からあまりに多くを奪ってしまうかもしれない。やがて、開発者は構造に合うようにアプリケーションを書き直すか、構造を転覆させて完全になくし、調整されていない開発による問題を呼び戻すことになる。

指針となるルールの有無が問題なのではなく、そうしたルールの厳格さと成り立ちが問題なのだ。設計を支配するルールが、本当に状況に適合していれば、開発の妨げになることなどなく、むしろ開発を役に立つ方向へ押し進めると同時に、一貫性をもたらす。

それゆえ：

概念上の大規模な構造をアプリケーションと共に進化させ、場合によっては途中で全く別の種類の構造に変更すること。詳細な知識を必要とする、設計とモデルに関する詳細な意思決定を、制約しすぎてはならない。

個別の部分をまとめ上げて表現する上では、自然な方法や役立つ方法がある。しかし、それらの方法は、全体には適用できないかもしれない。したがって、個別の部分に全体的なルールを押しつけると、理想的とは言えないものになってしまう。大規模な構造を使用するという選択は、個別の部分を最適に構造化することよりも、モデル全体としての扱いやすさを優先するということだ。したがって、構造を統一することと、最も自然な方法で個別のコンポーネントを表現する自由との間で、何らか妥協が生じることになる。こうした妥協を軽減するには、ドメインに関する実際の経験と知識に基づいて構造を選択し、過度に制約的な構造を避ければよい。構造がドメインと要件に対して本当にうまく合致すると、多くの選択肢を素早く除去できるので、実は、詳細なモデリングと設計が容易になるのだ。

構造は設計上の意思決定に近道を与えることもできる。原則として、そうした近道は個々のオブジェクトレベルに取り組むことで見つけられるものだが、現実には時間がかかりすぎるし、一貫性のない結果が生じることになるだろう。もちろん、継続的なリファクタリングは引き続き必要だが、構造があるおかげでリファクタリングが管理しやすいプロセスとなっているので、別々の人でも一貫した解決策を考え出すことができる。

大規模な構造は、一般に、境界づけられたコンテキストを横断して適用できる必要がある。実際のプロジェクトでのイテレーションを通じて、構造は、特定のモデルと緊密に結びついた特徴を失っていき、ドメインの持つ概念の輪郭に一致する特徴が進化する。これが意味するのは、モデルに関する前提がないということではなく、特定の局所的な状況に合わせた考え方を、プロジェクト全体に課すようなことはしないということだ。独自のコンテキストに取り組む開発チームに対しては、局所的な要求に対応した方法でモデルを変更できる自由を残しておかなければならない。

また、大規模な構造は開発に対する現実の制約にも対応しなければならない。例えば、設計者はシステムの一部のモデルをコントロールできないことがある。外部サブシステムやレガシーサブシステムの場合は、特にそうだ。これには、特定の外部の要素によりよく合致するよう、構造を変更することで対処できる。アプリケーションが外部と関係する方法を定義することによって処理することもできるだろう。構造を十分に緩やかにし、厄介な現実を柔軟にかわせるようにして対応してもよいだろう。

コンテキストマップと違い、大規模な構造は必須ではない。大規模な構造を課すべきなのは、費用対効果が有利に働き、うまく整合する構造が見つかった場合である。実際、システムが単純で、モジュールに分解することで理解できるのであれば必要ない。**大規模な**

構造を適用すべきなのは、モデルの開発に不自然な制約を強いることなく、システムを大幅に明確化する構造が見つけられた時だ。うまく合わない構造なら、ない方がましなのだから、包括的なものを目指すのではなく、出てきた問題を解決する最小限のものを見つけることが一番だ。「より少ないことは、より豊かなこと（Less is more）」なのだ。

大規模な構造は非常に役に立つが、いくつか例外も存在する。しかし、その例外には何らかのかたちでフラグを立てて、ただし書きがない限りは構造が守られていると、開発者が想定できるようにしなければならない。そういう例外が増えてきたら、その構造は変更するか、廃棄する必要がある。

＊　＊　＊

前述した通り、開発者に必要な自由を与えつつ、一方で混沌を回避する構造を作成するのは至難の業である。ソフトウェアシステムの技術的なアーキテクチャに関してはさまざまなことが行われてきているが、ドメイン層の構造化についてはほとんど公表されていない。アプローチの中には、ドメインをアプリケーションタスクやユースケースによって分割するといった、オブジェクト指向パラダイムを弱めてしまうものもある。この領域全体がまだ未開発なのだ。大規模な構造の一般的なパターンのうちいくつかが、多様なプロジェクトで姿を現すのを私は目にしてきた。本章ではそのうち4つを取り上げる。中には、読者の要求に合うか、プロジェクトに合った構造についてのアイデアにつながるものもあるかもしれない。

システムのメタファ（SYSTEM METAPHOR）

メタファ的思考はソフトウェア開発では広く行われており、とりわけモデルを用いた開発ではそう言える。しかし、エクストリームプログラミングにある「メタファ」のプラクティスは、システム全体の開発に秩序をもたらすための、独特のメタファの使い方を意味するようになってきた。

＊　＊　＊

防火壁によって、隣の建物で猛威をふるう火災から自分の建物を守れるのと同じように、ソフトウェアの「ファイアウォール（防火壁）」も、外部のより大規模なネットワークにある脅威からローカルネットワークを保護する。このメタファがネットワークアーキテクチャに影響して、ある製品カテゴリの全体を形作った。複数の競合するファイアウォールが、個別に開発され、何らかの交換が可能だと考えられていて、消費者が入手できるようになっている。ネットワーキングの初心者でも、この概念はすぐ把握できる。このように業界と顧客全体で理解が共有されていることは、メタファによるところが大きい。

だが、メタファは正確な喩えではなく、その力は両刃の剣である。ファイアウォールというメタファを使用すると、時としてフィルタリングが不十分で、望ましい通信を妨げる一方、壁の内側で起こる脅威からは一切保護しないソフトウェア障壁が開発されることになるのである。例えば、無線LANは攻撃を受けやすい。ファイアウォールというメタファの明確さは重宝するが、どんなメタファにも難点はあるのだ[1]。

ソフトウェアの設計は、非常に抽象的で把握しにくい傾向がある。システムを理解して、全体としてのシステムの見方を共有するための具体的な方法は、開発者もユーザも等しく必要としている。

あるレベルにおいて、メタファは我々の思考方法にあまりにも深く浸透しているため、あらゆる設計に行きわたっている。システムには「レイヤ（層）」があって、互いの「上に重なって」おり、「中心」には「カーネル（核）」がある、といった具合だ。しかし、時には、設計全体の中心的な主題を伝え、チームメンバ全員で共有される理解を提供するようなメタファが現れることもあるのだ。

そうなると、システムが実際にメタファによって形作られる。開発者はシステムのメタファと矛盾しないよう設計上の意思決定を行う。この一貫性により、他の開発者は複雑なシステムの多くの部分を同じメタファの観点から解釈できるようになる。開発者とエキスパートにとっては議論のための基準点ができるが、それはモデルそのものより具体的かもしれない。

1　システムのメタファのことを私がようやく理解できたのは、Ward Cunningham がこのファイアウォールの例をワークショップ講座で使うのを聞いた時である。

システムのメタファは厳密ではなく、理解しやすい大規模な構造で、オブジェクトパラダイムと調和がとれている。いずれにしてもシステムのメタファはドメインに対する類比にすぎないため、さまざまなモデルをある大まかな方法でマッピングすることができる。それによって複数の境界づけられたコンテキストに適用できるようになるので、コンテキスト間での作業を調整しやすくなる。

システムのメタファが著名なアプローチとなったのは、エクストリームプログラミングにおけるコアプラクティスの1つだからだ（Beck 2000）。残念ながら、本当に役立つメタファが見つかったプロジェクトはほとんどなく、この考え方をドメインに押し込もうとした結果、逆効果になったものもある。説得力のあるメタファが持ち込むリスクもある。設計に与えられる類比の側面は当面の問題にとって望ましいものでないかもしれず、また、用いられる類比が魅力的であっても適切ではないかもしれないのだ。

そうはいっても、システムのメタファは大規模な構造のよく知られたかたちの1つであり、プロジェクトによっては役に立つし、構造の一般的な概念をうまく説明している。

それゆえ：

システムを表す具体的な類比が登場し、チームメンバの想像力をとらえて、役立つ方向に思考を導きそうであれば、それを大規模な構造として採用すること。そのメタファを中心に設計を構成して、ユビキタス言語に取り入れること。システムのメタファは、システムに関するコミュニケーションを容易にすると同時に、システムの開発を導かなければならない。これによってシステムのさまざまな部分が一貫したものとなり、さらには、別の境界づけられたコンテキストを越えて広がるかもしれない。しかし、メタファはすべて不正確なものなので、拡大しすぎたり不適切であったりしないかを絶えず調べ直し、障害になるようなら取り消せるようにすること。

* * *

「素朴なメタファ」とそれを必要としない理由

役に立つメタファが現れるプロジェクトはほとんどないので、XPコミュニティの中には**素朴なメタファ**（naive metaphor）を話題に上げ始めたところもある。素朴というのは、ドメインモデルそのものという意味だ。

この用語にある問題は、成熟したドメインモデルが全く素朴でないという点にある。実際、「給与処理は組立工程のようなものだ」と言う方が、モデルよりはるかに素朴な見方に思われる。モデルはドメインエキスパートと共に繰り返し知識をかみ砕いたことの成果物であり、実際に動作するアプリケーションの実装に緊密に組み込まれることで、実証されてきたものだからだ。

素朴なメタファという用語はもう使うのをやめるべきだ。

システムのメタファはすべてのプロジェクトで効果的とは限らない。一般に大規模な構造は本質的ではない。エクストリームプログラミングにある12のプラクティスのうち、システムのメタファの役割はユビキタス言語で果たせる。うまく適合するシステムのメタファやその他の大規模な構造を見つけたら、それらを用いてユビキタス言語を強化するべきである。

責務のレイヤ（RESPONSIBILITY LAYERS）

本書を通じて、個々のオブジェクトには関連する責務の限定された集合が割り当てられている。だが、責務駆動設計は、個々のオブジェクトより大きな規模に対しても適用される。

＊　＊　＊

個々のオブジェクトがそれぞれ手作りの責務を与えられているとすると、ドメインの広い範囲をまとめて処理する上で指針も統一性も能力もないことになる。巨大なモデルに一貫性を与えるには、そうした責務を割り当てる際に何らかの構造を課すことが有効である。

ドメインについての深い理解が得られると、広範囲にわたるパターンが見えるようになる。ドメインによっては自然な階層を形成するものもある。特定の概念と活動が生まれる背後で、他の要素がさまざまなスピードとさまざまな理由で、独立して変化する。この自然な構造を活かし、構造を可視化して役立つものにするには、どうすればよいだろうか？この階層の形成が示唆しているのはレイヤ化である。レイヤ化は、最も成功を収めたアーキテクチャ設計パターンの1つである（Buschmann他 1996など）。

レイヤとはシステムにおける区画である。各区画に置かれた要素は「下位にある」レイヤのサービスのことを知っていて、使用することもできるが、「上位にある」レイヤのことは知らず、独立している。モジュールの依存関係を描く時には、あるモジュールに依存するものがあると、そのモジュールは自分に依存するものの下位に来るように配置される。この考え方に基づいてレイヤの順序が自然と整理され、下位レベルのオブジェクトで、上位のオブジェクトに概念的に依存するものがなくなることもある。

このようにその場しのぎでレイヤを形成しても、依存関係を追跡するのが容易になり、時には直観的に理解できることもあるのは確かだ。だが、こうしたレイヤがモデルに対して洞察を与えたり、モデリング上の意思決定を導いたりすることはない。そうした目的のためには、より計画的にレイヤ化を行う必要があるのだ。

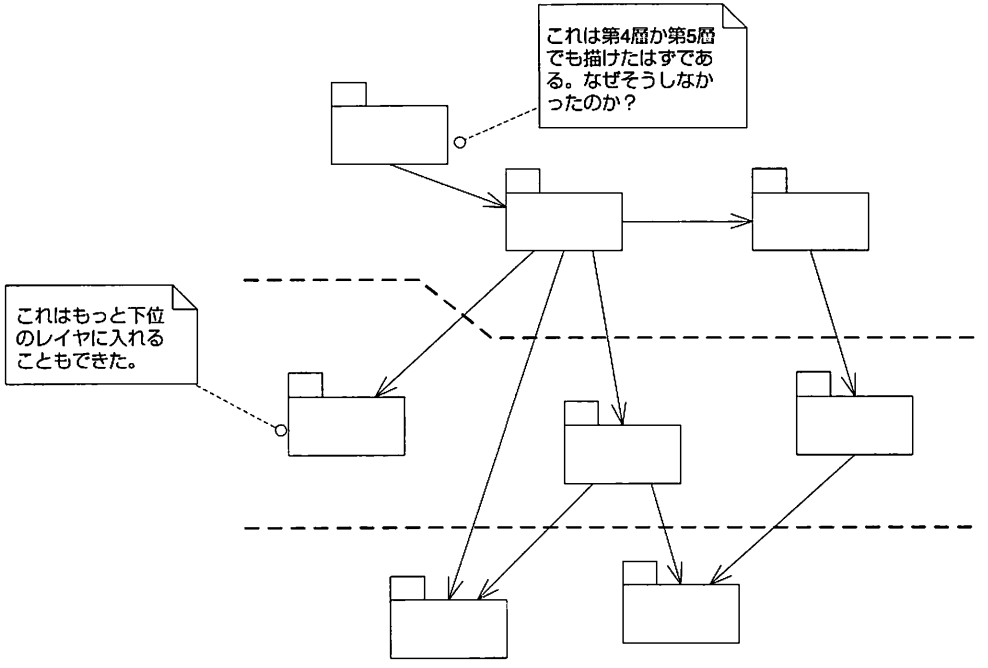


図16.2　その場しのぎのレイヤ化：これらのパッケージは何が目的なのか？

階層が自然に形成されるモデルにおいては、概念的なレイヤは主要な責務を中心として定義され、それによってレイヤ化と責務駆動設計という2つの強力な原理が統一される。

レイヤに割り当てられるこれらの責務は、後の例で簡単に示す通り、個々のオブジェクトに典型的に割り当てられる責務よりも、かなり広範なものでなければならない。個々のモジュールと集約を設計する際には、こうした広い責務のうち、どれか1つの境界内に収まるように括り出される。このように、責務をグループ化して名前をつけると、それだけでもモジュール化されたシステムがわかりやすくなる。モジュールに与えられた責務をより容易に解釈できるようになるからだ。しかし、高次の責務をレイヤ化と組み合わせることで得られるのは、システムをまとめ上げるための原則なのだ。

それゆえ：

モデルの中にある概念上の依存関係を調べること。また、ドメインのさまざまな部分の変更について、変化の割合とその原因も調べること。ドメインに自然な階層が認められたら、それに幅広い抽象的な責務を割り当てること。それらの責務は、システムの高次の目的と設計について語るものでなければならない。モデルをリファクタリングして、各ドメインオブジェクトと集約およびモジュールの責務が、1つのレイヤに与えられた責務の範囲

にうまく収まるようにすること。

これはかなり抽象的な記述だが、いくつかの例を示せば明確になるだろう。本章始めに挙げた衛星通信シミュレータでは、責務をレイヤ化していた。責務のレイヤが使われた結果、ドメインによい効果が与えられるところをこれまで私は何度も見ており、それは製造管理から財務管理まで多岐にわたっていた。

責務のレイヤに最適なレイヤ化パターンは**緩やかなレイヤ化システム**（RELAXED LAYERED SYSTEM）(Buschmann 他 1996, p.45) と呼ばれるものの変種である。これはあるレイヤのコンポーネントが直下のレイヤだけでなく、どの下位層へもアクセスできるようにするものだ。

* * *

次に挙げる例では、責務のレイヤを詳細に検討する。それによって、**あらゆる種類**に当てはまるような大規模な構造を発見する時の感触を示すと共に、構造がモデリングと設計をどのように導き、制約するかを説明する。

例 16.1 ——深く掘り下げる：輸送システムをレイヤ化する

これまでの章で例として述べてきた貨物輸送アプリケーションを取り上げて、そこに責務のレイヤを適用することが何を意味するかを見ていくことにしよう。

話を戻すと、チームはモデル駆動設計の作成とコアドメインの蒸留に関して、かなり前進してきている。しかし、設計が具体化されるにつれ、すべての部分をどう1つにまとめるかを調整するのが難しくなってきている。システムの主要テーマを明らかにして、全員の考えを統一しておける大規模な構造が求められているのだ。

このモデルの代表的な部分を次に示す。

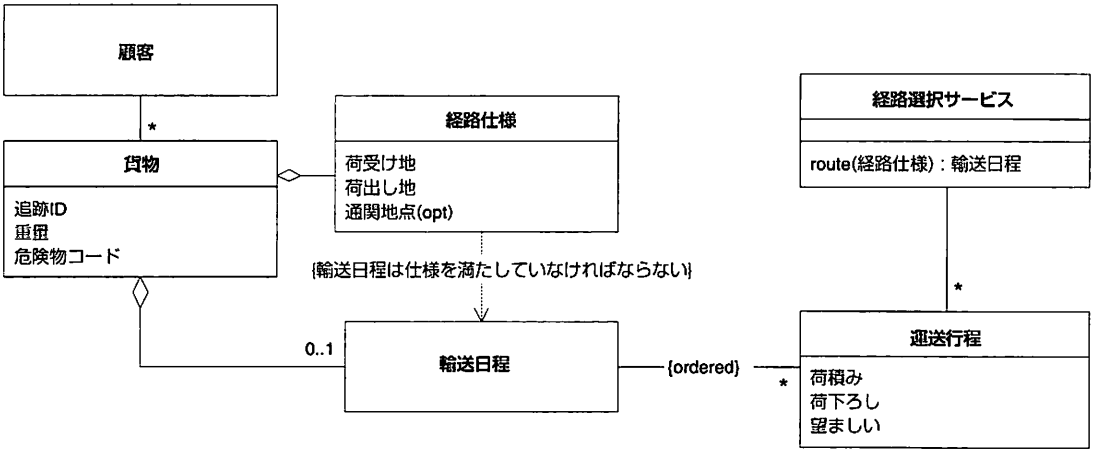


図16.3 貨物の経路選択を行う基本的な輸送ドメインモデル

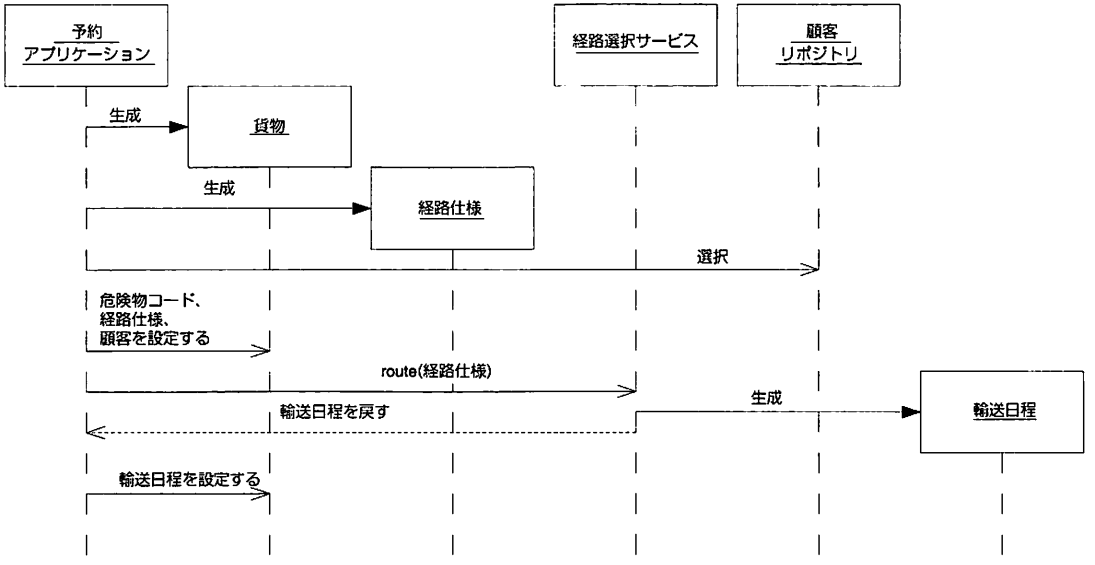


図16.4 予約時の貨物の経路選択にモデルを使用する

チームメンバは輸送のドメインに数か月間没頭してきており、概念の階層が自然に形成されていることに気づいた。運送中の貨物を参照しないで運送スケジュール（船舶と鉄道による予定された運行）について考えることは理にかなっている。逆に貨物を運ぶ運送を参照せずにその貨物を追跡することを考えるのは難しい。概念的な依存関係は非常に明確である。チームは2つのレイヤをすぐに区別できる。そこで、1つを「業務（Operation）」とし、その業務の基にあるものを「能力（Capability）」とした。

「業務」の責務

この会社の過去、現在および計画中の活動は業務層に集められる。最もわかりやすい業務オブジェクトは**貨物**だが、これは会社で日々行われるほとんどの活動の焦点である。**経路仕様**は**貨物**になくてはならない部分で、配送の要件を示す。**輸送日程**は業務上の配送計画である。これらのオブジェクトはどちらも**貨物**の集約の一部であり、そのライフサイクルは進行中の配送に与えられた期限に縛られる。

「能力」の責務

このレイヤは、業務を行うために会社が利用するリソースを反映している。**運送行程**が典型的な例だ。船舶は運行をスケジュールされ、貨物を運ぶための一定の積載量があるが、この積載量は十分に利用されないこともある。

確かに、輸送船舶の運航に焦点を絞れば、**運送行程**は業務層に入るだろう。しかし、このシステムのユーザはその問題を気にしない。（会社が両方の活動に携わっていて両方を調整し

●例16.1——深く掘り下げる：輸送システムをレイヤ化する

たければ、開発チームは別のレイヤ化案を検討して、「運送業務（Transport Operation）」と「貨物業務（Cargo Operation）」のように2つの別のレイヤにしなければならなかっただろう。）

より慎重に決定するべきなのは、**顧客**をどこに入れるかである。ビジネスによっては、顧客は一過性になりがちだ。荷物が配送されている間は関心の対象となるが、その後は、次の配送時まではほとんど忘れられる。この性質を見ると、顧客は、個人顧客向け小包配達サービスにおける業務的な関心事にしかならない。しかし、例で使っている架空の輸送会社は、顧客との長期にわたる関係を築こうとする傾向があり、ほとんどの仕事はリピータから来るものである。**ビジネスユーザのそういう意図を考えると、顧客は能力層に属する。見ての通り、これは技術的な観点による意思決定ではない。**ドメインの知識をとらえ、それを伝えようという試みだったのである。

貨物と**顧客**の関連は一方向にしか辿れないため、**貨物**リポジトリには、特定の**顧客**に紐づく**貨物**をすべて検索するクエリが必要となる。そのように設計する十分な理由はそもそもあったのだが、大規模な構造を課したことで、今や要件となった。

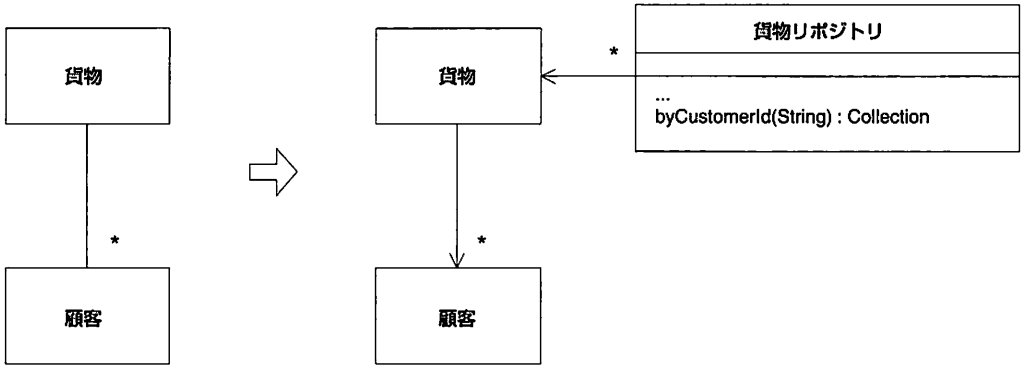


図16.5 クエリがレイヤ化に違反する双方向の関連と置き換えられる

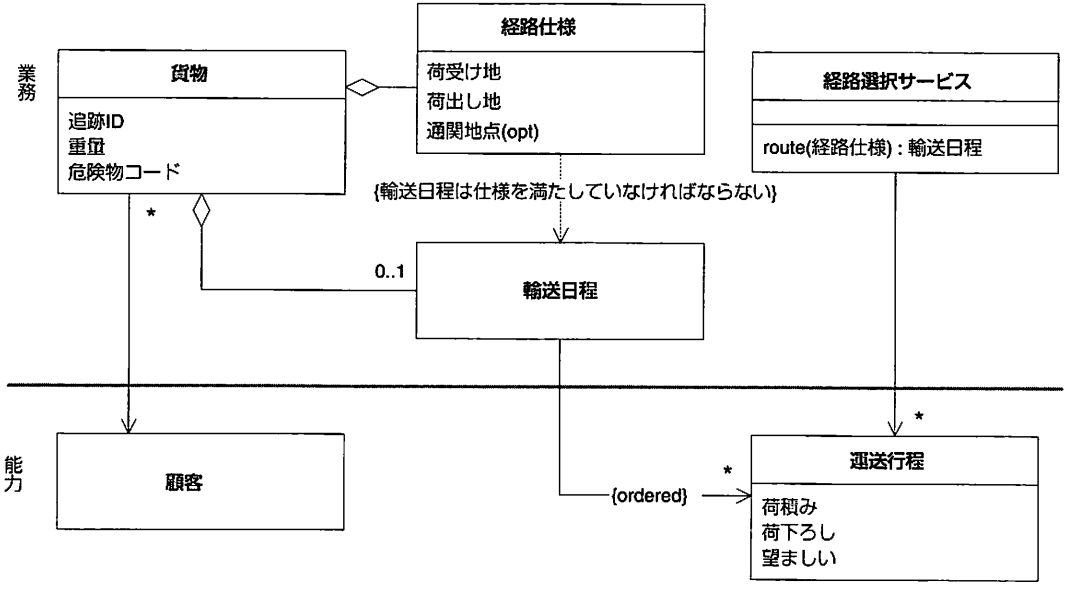


図16.6 最初に作成したレイヤ化モデル

業務と能力を区別することでイメージは明確になったが、秩序は進化し続ける。実験を数週間行った後、チームは別の区別に的を絞った。大体において、最初に作成したレイヤが集中しているのは、どちらも**現在置かれている**状況や計画である。しかし、**経路選択サービス**（および、この例からは除外されている他の要素）は、現在行われている業務や計画の一部ではない。そうした計画を変更する際に、意思決定を助けるものなのだ。チームは「意思決定支援（Decision Support）」の責務を負う新しいレイヤを定義した。

「意思決定支援」の責務

ソフトウェアにおけるこのレイヤは、計画作成と意思決定に用いるツールをユーザに提供し、潜在的には一部の意思決定を自動化できる（運送スケジュールが変更になった時、自動的に**貨物**の経路を選択し直すなど）。

経路選択サービスはサービスで、予約担当者が**貨物**を送る最適な方法を選択するのを支援する。このことから、**経路選択サービス**が意思決定支援に入ることははっきりしている。

このモデル内にある参照はすべてこの3つの層と一致しているが、唯一調和していない要素がある。**運送行程**にある「望ましい（is preferred）」という属性だ。この属性があるのは、使用する船舶を選ぶ際に、できれば自社の船舶か、有利な契約を結んでいる他社の船舶にしたいと会社が考えているからだ。「望ましい」属性は、そうした有利な運送へ経路選択サービスを偏らせるために使われる。この属性は「機能」とは全く関係ない。これは意思決定を方向づけるポリシーである。この新しい責務のレイヤを使用するには、モデルをリファクタリング

する必要がある。

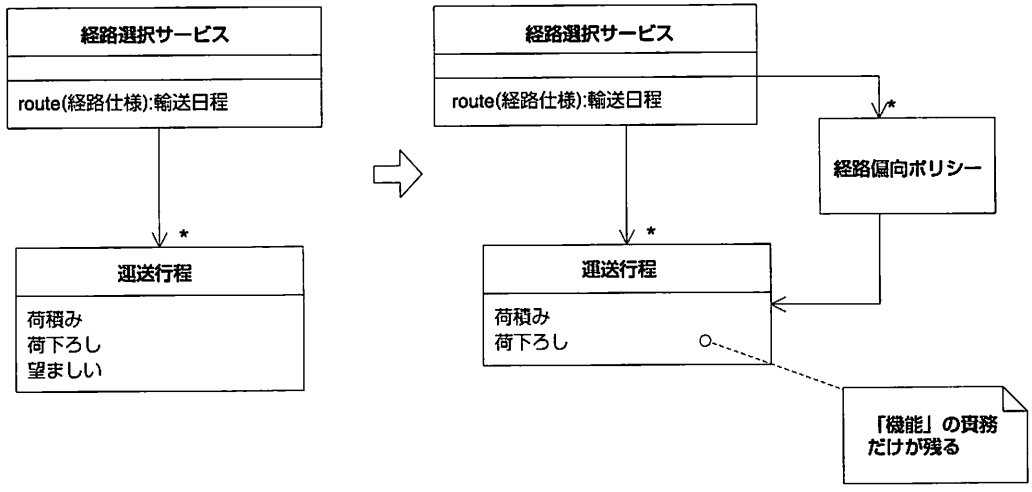


図16.7 新しいレイヤ化の構造に合わせてモデルをリファクタリングする

このように括り出すことで、**経路偏向ポリシー**（Route Bias Policy）がより明示的になり、その一方で**運送行程**を運送能力の基本となる概念にさらに集中させる。ドメインについての深い理解に基づいた大規模な構造は、しばしば、意味を明確にする方向へモデルを押し進める。

この新しいモデルは、これで大規模な構造にスムーズに収まる。

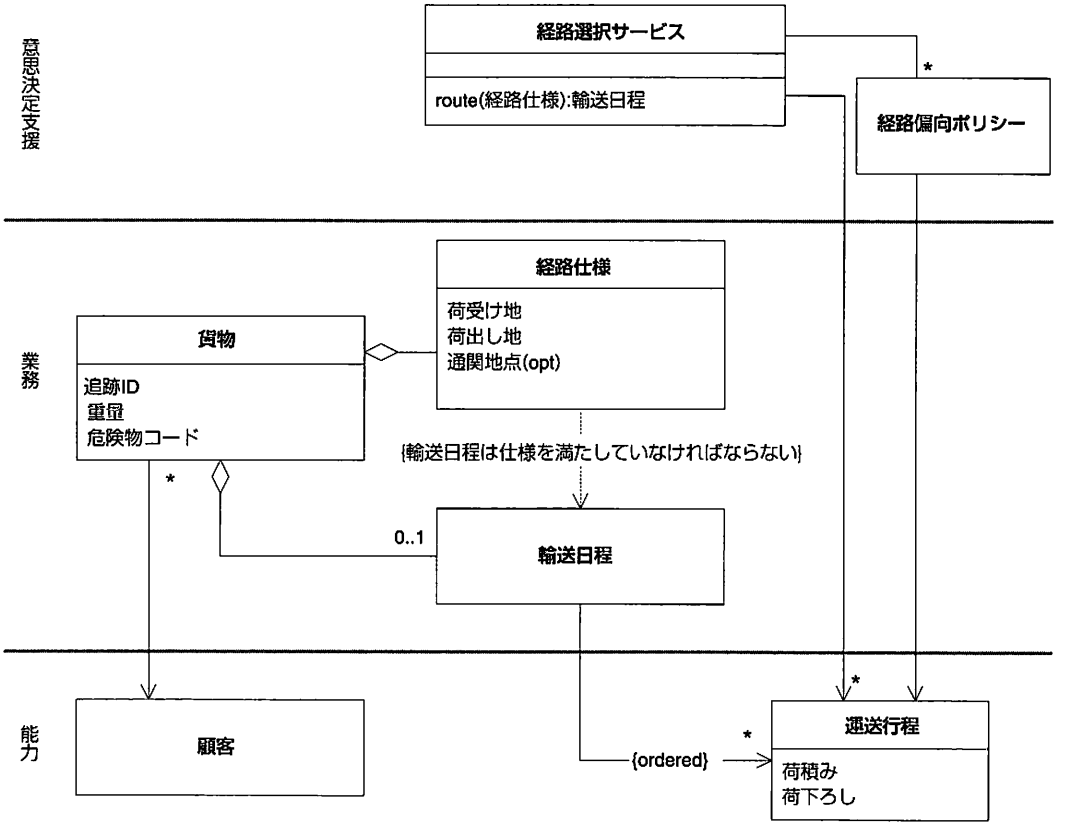


図16.8　再構造化とリファクタリングを行ったモデル

ここで選択されたレイヤに慣れている開発者であれば、各部分の役割と依存関係を即座に見分けられる。大規模な構造の価値は、複雑さが増すにつれて大きくなるのだ。

注意して欲しいのは、私はこの例を説明するのに修正版UML図を用いているが、図はレイヤ化を伝える手段にすぎないということだ。UMLにこういう表記はないから、これは読む人のために無理につけた追加情報である。コードがプロジェクトにとっての最終設計ドキュメントであるなら、レイヤ単位でクラスを表示するか、少なくともレイヤ単位でクラスを帳票出力するツールがあると便利である。

この構造は進行中の設計にどう影響するか？

一度大規模な構造を採用したら、その後に続くモデリングと設計上の意思決定では、その構造を考慮しなければならない。説明のため、すでにレイヤ化されているこの設計に、新しい機能を追加しなければならないものとしよう。ちょうどここでドメインエキスパートが話してくれたのは、特定のカテゴリの危険物に対しては、経路選択時に制限が適用されるというこ

●例16.1——深く掘り下げる：輸送システムをレイヤ化する

とだった。特定の物資は一部の運送や港で許可されないかもしれない。**経路選択サービス**をこうした規制に従うようにしなければならない。

とることのできるアプローチはいくつもある。大規模な構造がなければ、これらの経路選択ルールを調整するという責務を、**経路仕様**と危険物コードを所有するオブジェクトである**貨物**に与えるという設計も、よさそうに見える。

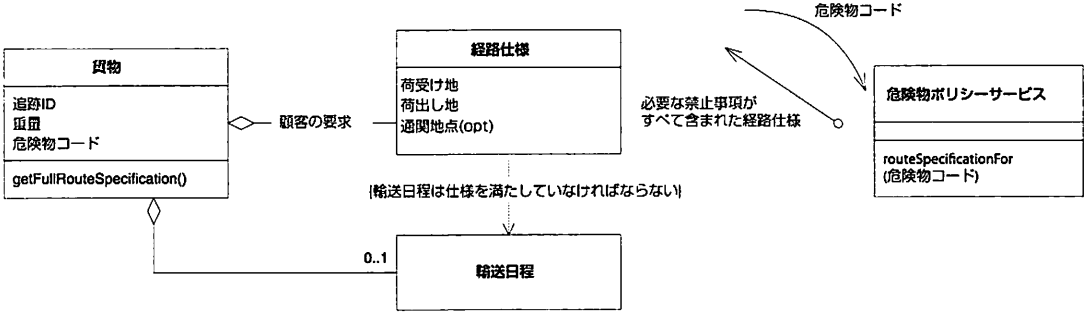


図16.9 危険な貨物の経路選択を行うための設計の可能性

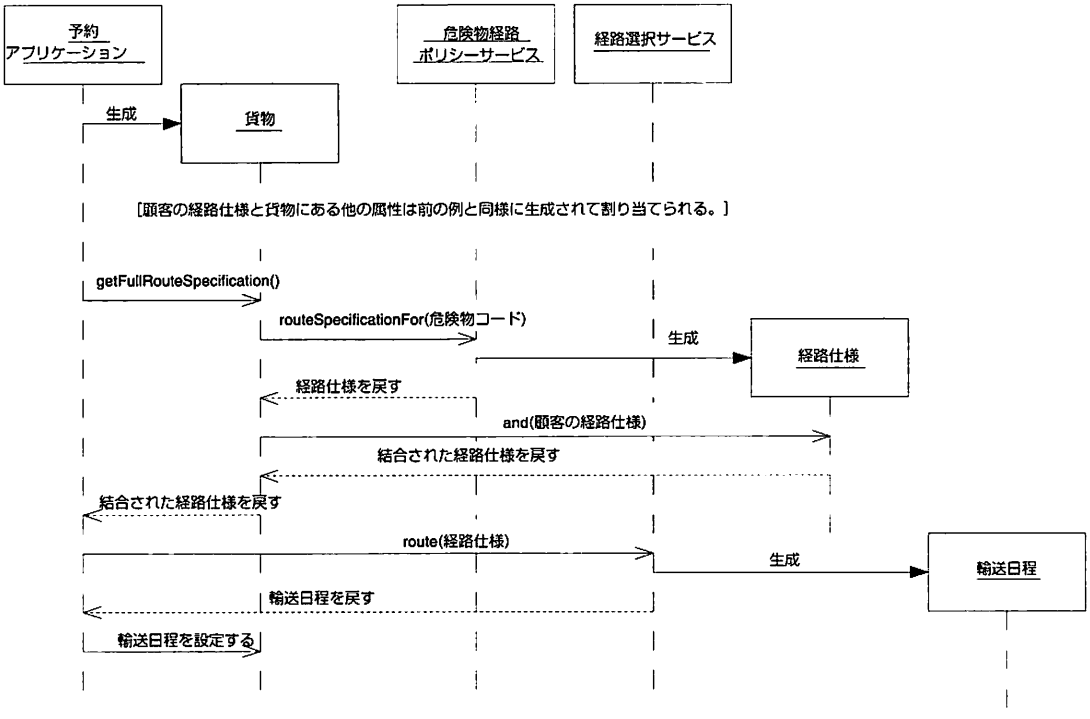


図16.10

厄介なのは、この設計が大規模な構造に合わないことである。**危険物経路ポリシーサービス**（HazMat Route Policy Service）に問題はない。意思決定支援層の責務にうまくなじむからだ。問題は、**貨物**（業務オブジェクト）が**危険物経路ポリシーサービス**（意思決定支援オブジェクト）に依存していることにある。プロジェクトがこれらのレイヤを堅持する限り、このモデルを認めることはできない。認めれば、構造には従うものと思っている開発者を混乱させてしまうからだ。

設計の可能性はいつでも数多くあるから、別の大規模な構造のルールに従う設計を選びさえすればよい。**危険物経路ポリシーサービス**に問題はないが、ポリシーを使用する責務は移動する必要がある。試しに、経路選択サービスに対して、経路を検索する前に適切なポリシーを収集する責務を与えてみよう。これが意味するのは、**経路選択サービス**のインタフェースを変更して、ポリシーが依存するであろうオブジェクトを含めるということだ。考えられる設計は次のようになる。

●例 16.1 ——深く掘り下げる：輸送システムをレイヤ化する

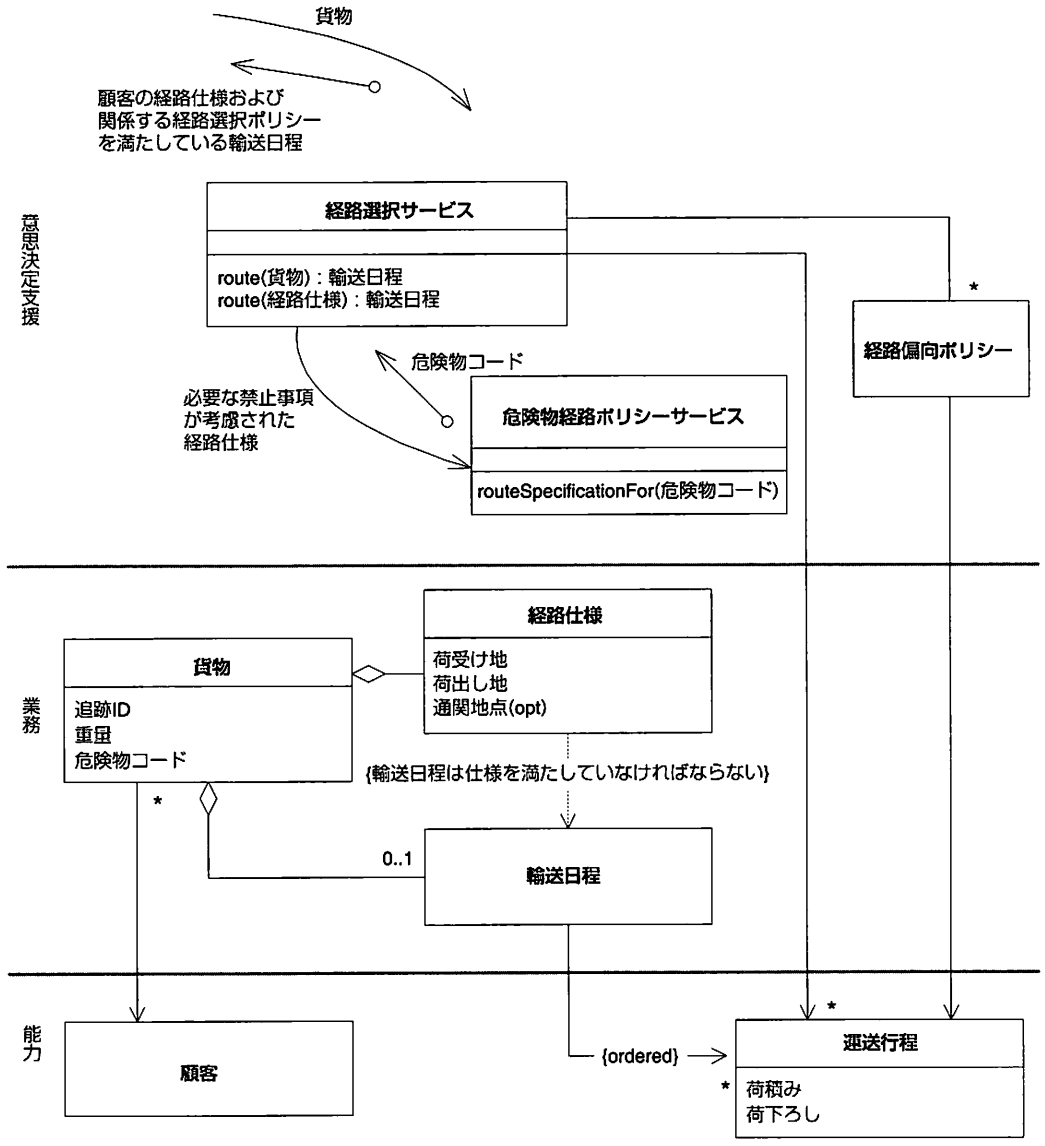


図16.11 レイヤ化と矛盾しない設計

典型的な相互作用を図16.12に示す。

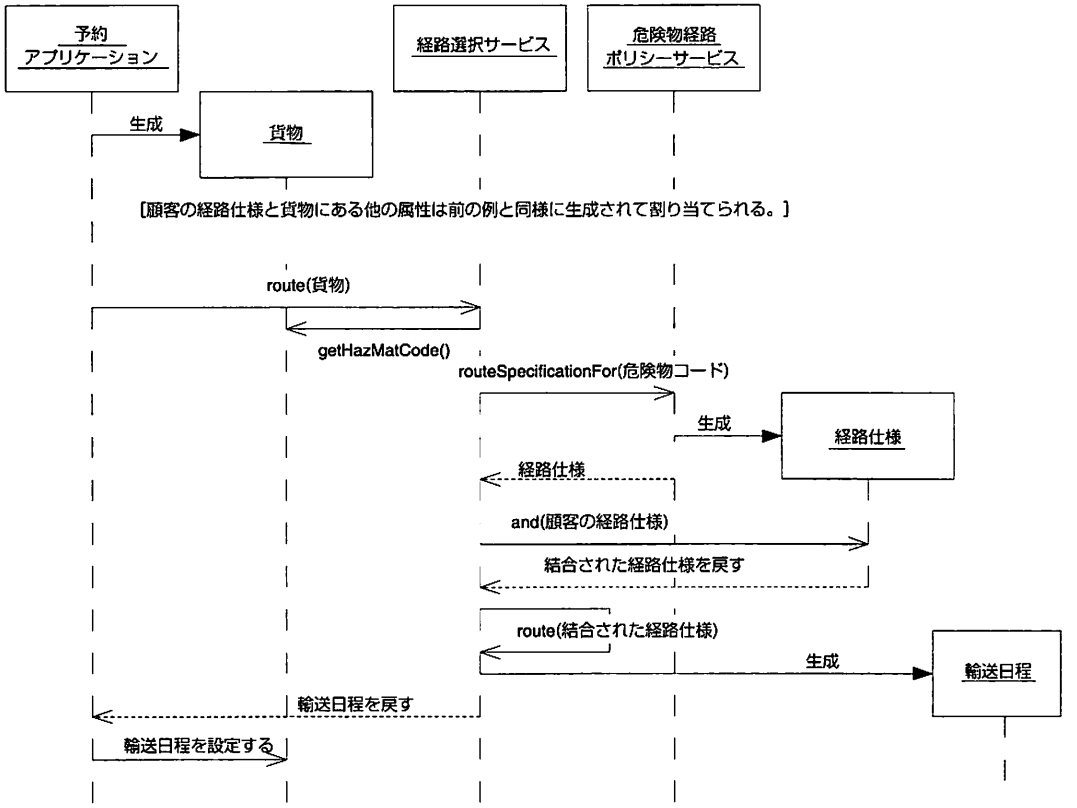


図16.12

さて、これは必ずしも、先ほどより優れた設計ではない。どちらにも長所と短所がある。しかし、プロジェクト関係者全員が一貫したやり方で意思決定をすれば、全体としての設計ははるかに理解しやすくなる。それを思えば、詳細な設計上の選択を行うにあたって、適度なトレードオフを行う価値はある。

構造のせいで多数のぎこちない設計を選択するよう強いられるのであれば、進化する秩序に合わせてその構造を評価し、場合によっては修正したり、廃棄したりすることも必要になるだろう。

適切なレイヤを選択する

ドメインにとってふさわしい責務のレイヤや、何かしらの大規模な構造を見つけるということは、問題となっているドメインを理解し、それについて実験するということである。進化する秩序を容認するのであれば、最初の出発点は決定的に重要というわけでもないが、選択を誤れば作業が増えることは確かだ。構造が進化して、何か認識できないものになるかもしれない。そのため、ここで提案する指針は、構造の変換を検討する際にも、一から選択する場合と同じく適用するべきである。

レイヤは切り替えられ、マージされ、分離され、再定義されるものである。その際に、見つけ出して保持しておくべき、役に立つ特徴を次に挙げよう。

- **ストーリーテリング。**レイヤはドメインの基本的な現実や優先事項を伝える必要がある。大規模な構造の選択は、技術的な意思決定というよりも、ビジネスモデリング上の意思決定である。レイヤはビジネスの優先事項を明らかにしなければならない。
- **概念的な依存関係。**「上位」層にある概念は「下位」層を背景とした意味を持つべきであるのに対し、下位層の概念は単独で意味を持たなければならない。
- **概念の輪郭。**別々のレイヤにあるオブジェクトを見た時に、変化のスピードや原因が別々であれば、レイヤは両者を切り分ける場所となる。

新しいモデルそれぞれに対してレイヤを定義する際は、必ずしも一から始める必要はない。関連するドメインのファミリー全体に出てくるレイヤもある。

例えば、工場や貨物船のような大型固定資産の活用に基づくビジネスでは、物流ソフトウェアを「潜在能力（Potential）」層（例に挙げた「能力」層の別名）と「業務」層とに分けることもできる。

- **潜在能力。**何ができるか？ 何を実行しようとしているかは気にしなくてよい。何ならできたか？ 人間を含む組織のリソースと、そのリソースの構成方法が、潜在能力層のコアである。ベンダとの契約も潜在能力を定義する。このレイヤは、ほぼどんなビジネスドメインでも見つかる。しかし、このレイヤが話題の中心となるようなビジネスには、運送や製造といった、ビジネスを可能にする比較的大口の固定資産投資があるものだ。潜在能力には一時的な資産も含まれているが、主に一時的な資産によって推進されるビジネスであれば、後述するように、そのことを強調するレイヤを選択するだろう（例ではこのレイヤを「能力」と呼んでいた）。
- **業務。**何がなされたか？ 潜在能力から何を作り出せたのか？ 潜在能力層と同様に、このレイヤが反映すべきなのも現実の状況であって、そうあって欲しいと望む状況ではない。このレイヤで見ようとしているのは、我々自身の取り組みと活動である。すなわち、実際に販売しているものを見るのであって、販売を可能にしているものを見

るのではない。業務オブジェクトが潜在能力オブジェクトを参照したり、潜在能力オブジェクトで構成されたりするのは、珍しいことではない。しかし、潜在能力オブジェクトが業務層を参照してはならない。

この種のドメインにおける既存システムの多く、あるいはそのほとんどでは、これら2つのレイヤによって、すべてが網羅されている（ただし、全く別の、より啓発的な分割もあり得る）。この2つの層は現在の状況と実行中の業務計画を追跡して、それに関するレポートやドキュメントを出力する。しかし、追跡するだけで十分とは限らない。プロジェクトが目指すものがユーザの手引きや補佐、あるいは意思決定の自動化だとすると、付加的な責務の集合が存在する。これは業務の上に位置づけられる、もう1つのレイヤとして構成できる。

- **意思決定支援。**どのように活動し、どのようなポリシーを設定すべきか？ このレイヤは分析と意思決定のために存在する。分析の根拠となるのは、潜在能力や業務といった下位層からの情報である。意思決定支援ソフトウェアは、現在や将来における業務の機会を積極的に探すために、履歴情報を使用するかもしれない。

意思決定支援システムは、業務や潜在能力といった他のレイヤに概念的に依存する。意思決定は何もないところではできないからだ。多くのプロジェクトでは、データウェアハウス技術を使用して意思決定支援を実装する。このレイヤは、はっきりとした境界づけられたコンテキストになって、業務ソフトウェアとは顧客／供給者関係になる。他のプロジェクトでは前述の応用例のようにさらに深く統合される。レイヤに内在する利点の1つに、下位層は上位層がなくても存在できることがある。これによって、段階的に導入したり、旧来の業務システム上に高度な追加機能を構築したりするのが容易になる。

もう1つのケースとして、入り組んだビジネスルールや法的要件を強制するソフトウェアがあるが、これも責務のレイヤを構成できる。

- **ポリシー。**ルールや目標は何か？ ルールや目標はたいてい受動的だが、他のレイヤにおけるふるまいを制約する。こういう相互作用の設計は扱いが難しい。ポリシーが引数として下位レベルのメソッドへ渡されることもある。ストラテジーパターンが適用されることもある。ポリシーは意思決定支援層と合わせて使うと効果的である。意思決定支援層によってポリシーが設定した目標を探し出す手段が提供され、また、意思決定支援層はポリシーが設定するルールに制約される。

ポリシー層は他のレイヤと同じ言語で書くことができるが、ルールエンジンを使用して実装されることもある。ルールエンジンは必ずしも別の境界づけられたコンテキストに入れる必要はない。実際、そういう別々の実装技術を調整する難しさは、両方で同一のモデルを使用するように、細心の注意を払うことによって軽減できる。適用されるオブジェク

トとは異なるモデルに基づいてルールが作成された場合は、複雑さが格段に増すか、扱いやすくするためにオブジェクトが書き直されることになる。

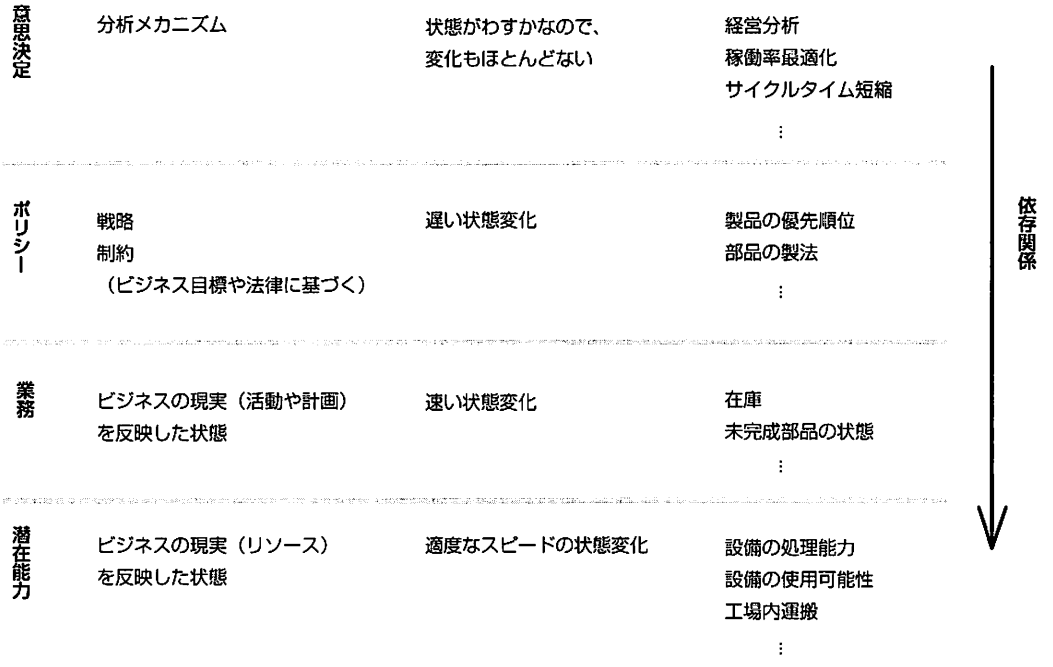


図16.13 工場自動化システムにおける概念的依存関係と切り分けポイント

多くのビジネスにおいて、その能力は工場や設備に基づくものではない。金融サービスや保険の2つを例に挙げると、潜在能力はほとんどが現在の業務によって決まる。新規契約を結ぶことによる新しいリスクを保険会社が負担する能力は、現在のビジネスの多様性に基づいている。この場合、潜在能力層は業務にマージされるか、または、別のレイヤ化が進化するかもしれない。

こういう状況で前面に出て来ることの多い領域が、顧客に対してなされる確約（commitment）である。

- **確約。**何を約束したか？ このレイヤは、将来の業務を方向づける目標を述べている点で、ポリシーとしての性質を持つ。同様に、進行中のビジネスの活動の一部として確約が現れ、変化するという点で、業務の性質も持っている。

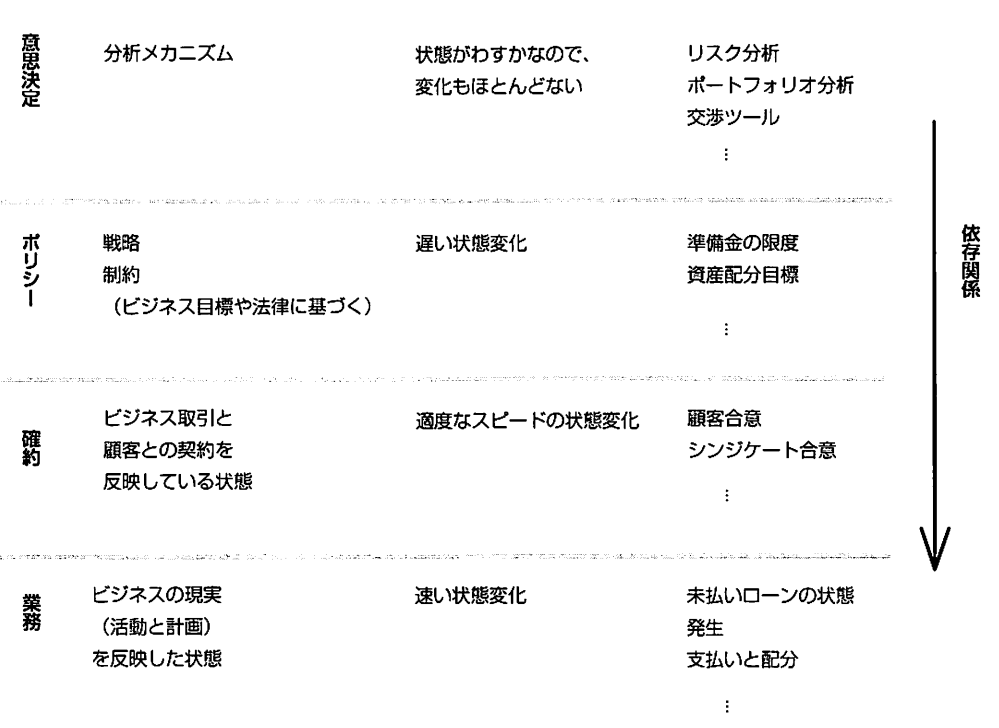
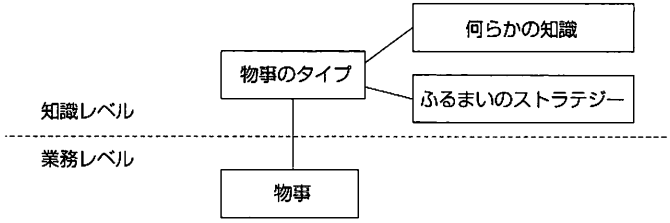


図16.14 投資銀行システムにおける概念的依存関係と切り分けポイント

潜在能力と確約のレイヤは相互に排他的ではない。カスタマイズされた輸送サービスを伴う運送会社のように、両方のレイヤが目立つドメインでは、どちらも使用されるだろう。そういうドメインにより特化した別のレイヤも役立つかもしれない。物事を変えて実験すること。しかし、レイヤ化システムは単純にしておくのが一番だ。4層あるいはせいぜい5層を超えると扱いにくくなる。レイヤが多すぎると、ストーリーを語るには効果的でない上、大規模な構造で解決するはずだった複雑さの問題がかたちを変えて戻ってきてしまう。大規模な構造は徹底的に蒸留しなければならないのだ。

ここに挙げた5つのレイヤは広範なエンタープライズシステムに適用できるが、すべてのドメインにある顕著な責務をとらえているわけではない。場合によっては、設計をこのかたちに押し込もうとすることが逆効果になり、むしろ、責務のレイヤの自然な集合を用いた方がうまくいくこともあるだろう。これまで議論してきたものと一切関係ないドメインの場合、責務のレイヤは完全に独自のものでなければならないかもしれない。最終的には、自らの直観を活かして、どこかから着手し、秩序を進化させなければならないのである。

知識レベル（KNOWLEDGE LEVEL）



［知識レベルとは］他のオブジェクトグループがどうふるまうべきかを記述するオブジェクトグループである。［Martin Fowler, "Accountability," www.martinfowler.com］

モデル自体の一部をユーザの手で自由に変えられるようにしながらも、より広範なルールの集合で制約しなければならない場合、知識レベルによって物事のもつれが解きほぐされる。知識レベルはソフトウェアの要件に対し、設定可能なふるまいで対応する。そこでは、エンティティの役割と関係が、インストール時や実行時にさえも変更できなければならない。

『アナリシスパターン』（Fowler 1996, p.24-27）において、このパターンは組織内の説明責任（accountability）をモデル化する議論から現れ、その後、会計の記帳ルールに適用されている。このパターンは同書で複数の章に出てくるものの、独立した章にはなっていない。これは、その他のほとんどのパターンとは異なっているためだ。他のアナリシスパターンのようにドメインをモデル化するのではなく、知識レベルは、モデルを構造化するのである。

この問題を具体的に理解するため、「説明責任」のモデルを考えよう。組織とは、人々とさらに小さい組織によって構成され、それらの果たす役割と相互の関係性を定義するものだ。そうした役割と関係性を規定するルールは、組織によって大きく異なる。ある企業では、「部門」を統率するのは「取締役」で、その報告相手は「副社長」かもしれない。別の企業では、「課」は「課長」によって統率され、報告は「部長」に対して行う。さらに「マトリクス」型の組織もあり、そこでは各人が別々の目的に応じて別々の管理者に報告する。

アプリケーションは、通常、何らかの前提を設けるものだ。その前提とうまく合わないと、ユーザは意図されたのとは異なる方法でデータ入力フィールドを使い始める。だが、セマンティクスがユーザによって変更されてしまっているため、アプリケーションの持つふるまいはすべて失敗するだろう。ユーザはそのふるまいの次善策を考え出すか、アプリケーションの高度な機能を使わなくなってしまう。あるいは、自分が作業したことと、ソフトウェアの動きとの間を、複雑に紐づけることを習得せざるを得なくなるかもしれない。

だが、そうした紐づけがうまくいくことはないだろう。

システムの変更やリプレイスが必要になったときに、開発者は機能の意味が自分たちの考えたものと違うことを（遅かれ早かれ）悟るだろう。ユーザコミュニティや状況が異なれば、機能は全く別のものを意味するかもしれないのだ。このように重ね合わされた用途を壊さずに何かを変更するのは骨が折れるだろう。特定の目的に、より適合したシステムにデータを移行するには、そうした独特のふるまいをすべて理解してコーディングする必要がある。

例 16.2——従業員の給料と年金（1）

ある中規模の企業の人事部に、給料と年金の拠出金を計算する簡単なプログラムがある。

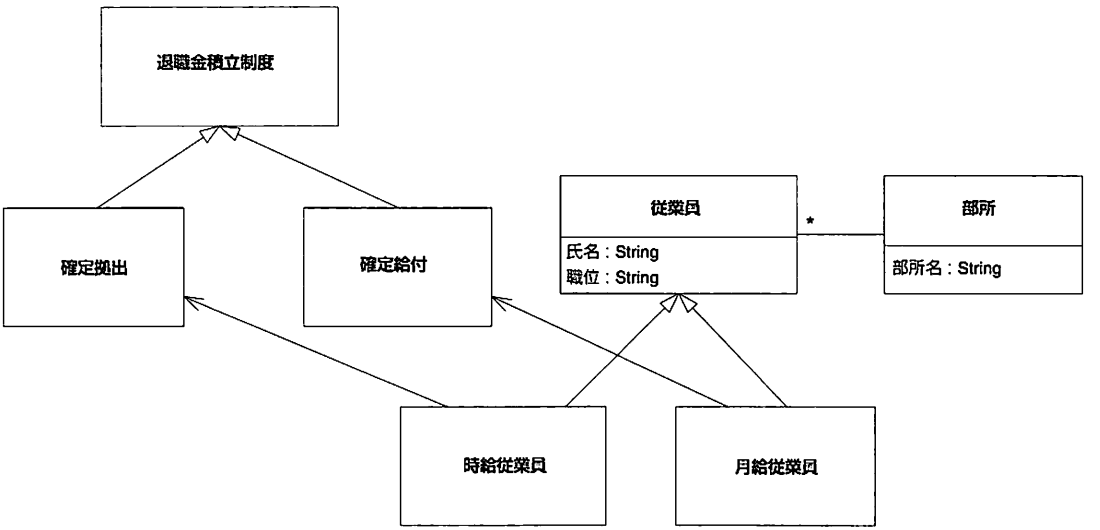


図 16.15　新しい要件に対して制約が多すぎる旧モデル

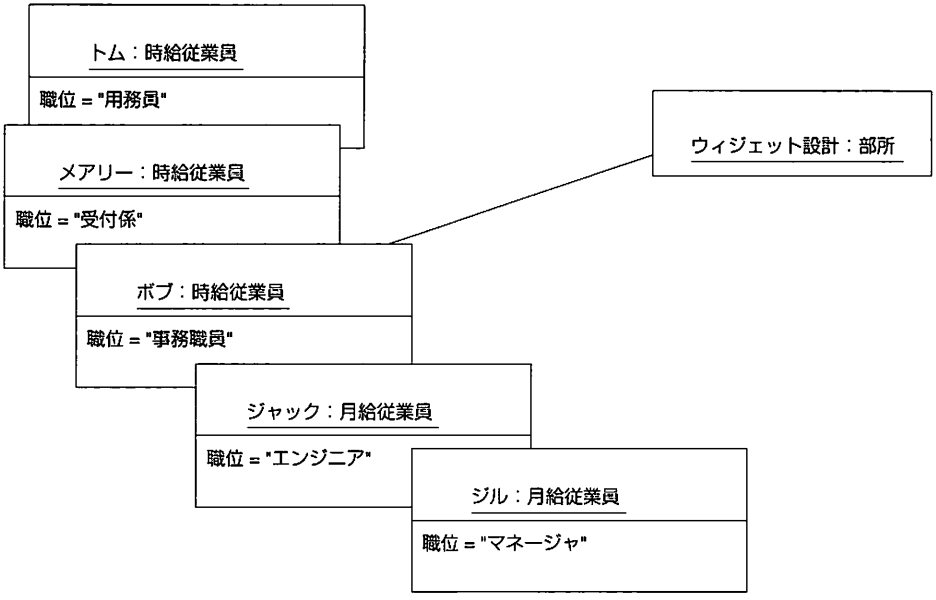


図16.16 旧モデルを使用して表現した一部の従業員

だがここで、経営陣が事務職員を「確定給付」の退職金積立制度へ入れるべきだと決定した。問題は、事務職員が時給制であるのに、このモデルが混在を認めていないことである。モデルを変更しなければならない。

次のモデルの提案はきわめて単純で、ただ制約を取り除いただけである。

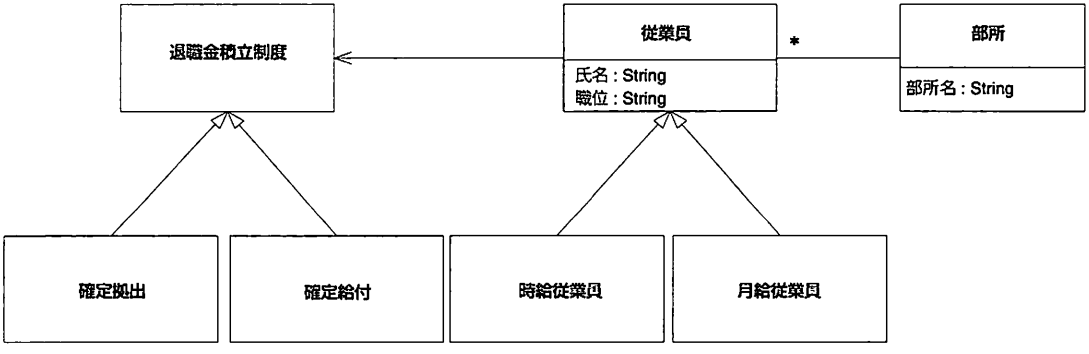


図16.17 今度は制約が足りない提案されたモデル

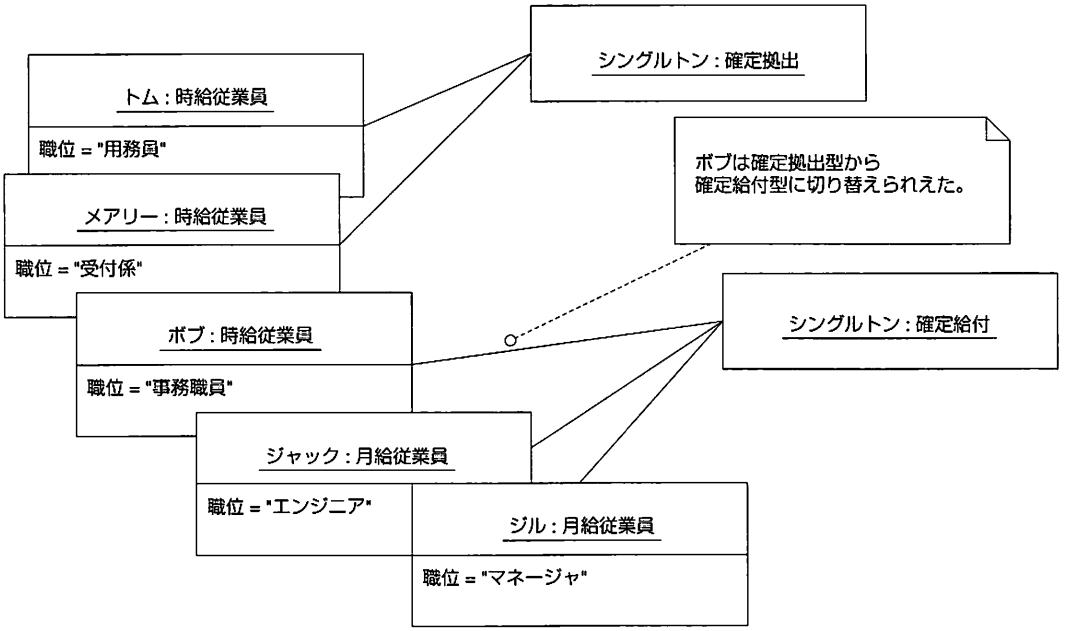


図16.18　従業員を誤った制度と結びつけることができる

このモデルは従業員それぞれをどちらの退職金制度にも関連づけられるようにしているから、各事務職員について切り替えができる。だが、このモデルは経営陣から却下された。会社の方針を反映していないからだ。事務職員でも、切り替えられる人とそうでない人がいる。あるいは、用務員も切り替えられる。経営陣が求めたのは、次の方針を徹底するモデルである。

事務職員とは確定給付の退職金制度を持つ時間給従業員である。

この方針が示唆しているのは、「職位」フィールドが今や重要なドメインの概念を表すようになったということである。開発者はリファクタリングを行うことで、この概念を「**従業員タイプ（Employee Type）**」として明示できた。

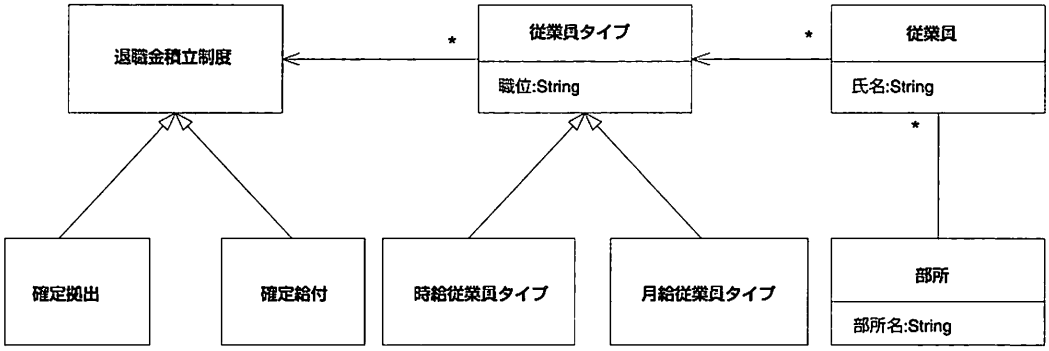


図16.19 従業員タイプオブジェクトによって要件を満たせるようになる

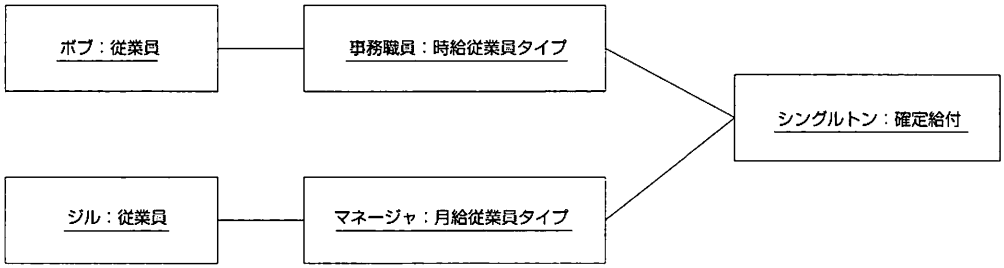


図16.20 各従業員タイプに退職金制度が割り当てられる

この要件はユビキタス言語で次のように述べられる。

従業員タイプは、いずれかの**退職金制度**またはいずれかの給料に割り当てられる。

従業員は**従業員タイプ**によって制約される。

従業員タイプオブジェクトを編集するためにプログラムにアクセスするのは、「スーパーユーザ」に限定される。スーパーユーザが変更を行うのは、会社の方針に変更があった時だけだ。人事部に属する普通のユーザにできるのは、**従業員**オブジェクトを変更することと、別の**従業員タイプ**を指すようにすることだ。

このモデルは要件を満たしている。開発者は暗黙的な概念が1つ、2つあることを感じてはいるが、それは一時的に頭から離れなくなっている感覚にすぎない。追求すべき確固たる案もないので、彼らはこれで切り上げることにした。

静的なモデルは問題を引き起こすかもしれない。しかし、あり得る関係をすべて示せる完全に柔軟なシステムでも、問題は同じくらいひどいものになりかねない。そういうシステムは使いにくい上、組織独自のルールを強制できるようにはならないのである。

組織ごとにソフトウェアを完全にカスタマイズするのは現実的でない。各組織が特注のソフトウェアの費用を支払えるとしても、組織構造が頻繁に変わるかもしれないからだ。

したがって、そういうソフトウェアはオプションを提供し、組織の現在の構造を反映するために**ユーザ**が自分で設定できるようにしなければならない。問題は、そうしたオプションを付加すると、モデルオブジェクトが扱いにくくなることである。柔軟性をつけ加えればつけ加えるほど、ますます複雑化してしまうのだ。

エンティティ間にある役割と関係が状況によって変わるアプリケーションでは、複雑さが爆発的に増えかねない。完全に汎用的なモデルも、高度にカスタマイズされたモデルも、ユーザの要求には役立たない。最終的にオブジェクトに残るのは、多様な状況を網羅するための他の型への参照か、状況によって使い道が変わる属性である。同一のデータとふるまいを持つクラスが、別々の組み立てルールに適応するためだけに数が増えてしまうかもしれない。

我々のモデルに埋め込まれるのは、我々のモデルについてのもう1つのモデルである。知識レベルはそうしたモデルの自己定義的な側面を分離して、それによる制約を明示的にする。

知識レベルはリフレクション（REFLECTION）パターンをドメイン層に適用したものだ。リフレクションは多くのソフトウェアアーキテクチャや技術的なインフラストラクチャに使用されており、『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』（Buschmann他 1996）で詳述されている。リフレクションは変化する要求に対応するために、ソフトウェアを「自覚的（self-aware）」にして、その構造とふるまいの中から選ばれた特定の側面に対して、適合や変更を行うためにアクセスできるようにする。これはソフトウェアを、アプリケーションの業務に関する責務を持つ「ベースレベル」と、ソフトウェアの構造とふるまいに関する知識を表す「メタレベル」に分けることによって実現される。

重要なことだが、このパターンは知識「レイヤ」とは呼ばれない。レイヤ化と非常に似てはいるが、リフレクションでは両方向に向かう相互依存関係が必要となるからだ。

Javaには、クラスに対してメソッドなどを直接問い合わせるプロトコルのかたちで、最小限のリフレクションが組み込まれている。こうした仕組みにより、プログラムがそれ自身の設計について問い合わせができるようになる。CORBAにもよく似たリフレクションプロトコルがあるが、もう少し拡張性がある。永続化技術の中には、豊かな自己記述の能力を拡張して、データベーステーブルとオブジェクト間で部分的に自動化されたマッピングをサポートするものもある。技術的な例は他にもあるが、このパターンはドメイン層内部にも適用できる。

知識レベルと**リフレクション**の用語比較

Fowlerの用語法	POSAの用語法[2]
知識レベル	メタレベル
業務レベル	ベースレベル

ここではっきりさせておくと、プログラミング言語のリフレクションツールは、ドメインモデルの知識レベルの実装に使うためのものではない。リフレクションツールのメタオブジェクトは、言語の構成体そのものの構造とふるまいを記述するものである。それに対して知識レベルは、普通のオブジェクトで構築しなければならない。

知識レベルにより、役に立つ対比が2つ提供される。第一に、知識レベルはリフレクションのよく知られた使い方とは対照的に、アプリケーションドメインに焦点を合わせている。第二に、知識レベルは完全な汎用性を目指してはいない。仕様が汎用的な述語よりも役に立つのと同様に、オブジェクトとその関係性の集合に対する非常に特殊な制約が一式あれば、汎用化されたフレームワークよりも役立つことがある。知識レベルの方が単純で、設計者の具体的な意図を伝えられる。

それゆえ：

基礎となるモデルの構造とふるまいを記述し制約するのに使用できる、独特なオブジェクトの集合を作成すること。これらの関心事は2つの「レベル」として別々に保ち、一方を非常に具体的にし、もう一方にはユーザやスーパーユーザがカスタマイズできるルールと知識を反映させること。

あらゆる強力なアイデアと同様、リフレクションと知識レベルには中毒性がある。このパターンは慎重に使用しなければならない。業務オブジェクトを器用貧乏にしておかなくてもよくなることで、複雑さを解きほぐすことはできる。しかし、このパターンを用いるとどの方向にも向かえるようになるため、再びあいまいな部分が生じてしまうことにもなる。知識レベルが複雑になると、システムのふるまいは開発者にとってもユーザにとっても等しく理解しにくくなる。設定を行うユーザ（またはスーパーユーザ）には、最終的にプログラマのスキルが必要になる。それもメタレベルのプログラマのスキルだ。このユーザが間違いを犯せば、アプリケーションは間違ったふるまいをするようになるのである。

また、データ移行に伴う基本的な問題は、完全にはなくならない。知識レベルの構造が変更されれば、既存の業務レベルのオブジェクトもそれに対応しなければならない。新旧を共存させることができても、何らかのかたちで慎重な分析が必要なのだ。

2 POSAは、Buschmann他 1996の『Pattern-Oriented Software Architecture』（邦訳：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）の頭文字を取ったものである。

こうした論点がすべて知識レベルの設計者にとって大きな負担となる。この設計は、開発で示されるシナリオを扱うだけでなく、将来、ユーザがソフトウェアの設定を行う際に基にするであろう、あらゆるシナリオも扱えるくらい強固でなければならない。カスタマイズが決定的に重要で、そうしなければ設計をゆがめることになる場所に対して慎重に適用すれば、知識レベルは、他のどんな方法でも対処するのが非常に難しい問題を解決することができる。

例 16.3——従業員の給料と年金（2）知識レベル

チームメンバは一晩眠り、リフレッシュして戻ってきたところで、そのうちの1人が設計に存在するぎこちない点の1つに肉薄してきた。特定のオブジェクトが保護されている一方で、自由に編集できるオブジェクトもあるのはなぜなのか？ 編集を制限されたオブジェクトの集まりを見て、この開発者は知識レベルパターンのことを思い出し、モデルに対する1つの見方として使ってみることにした。既存のモデルのことを、次に示すように見ることができるということに気づいたのである。

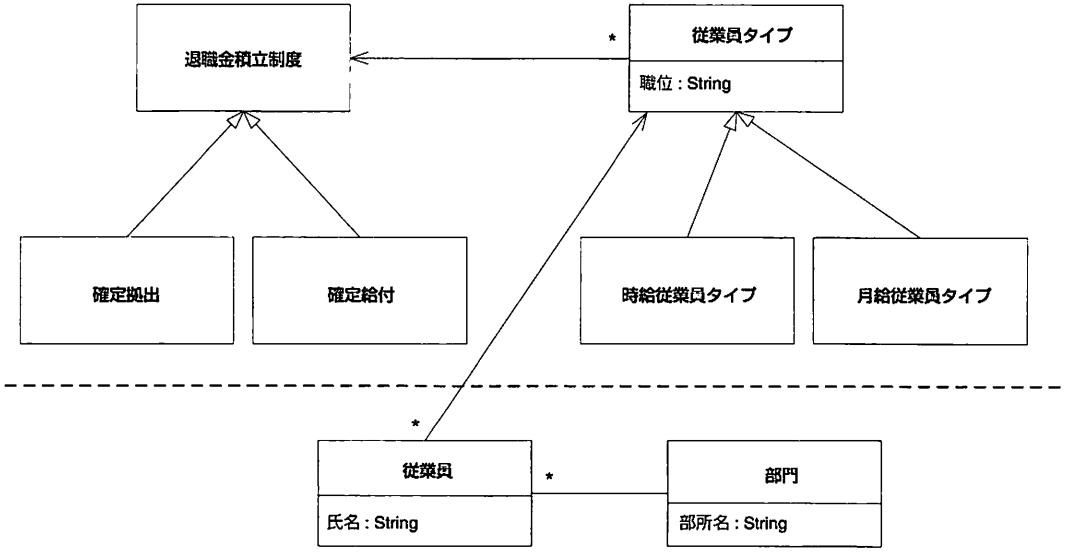


図 16.21 既存のモデルに**知識レベル**が暗黙的に含まれていることに気づく

制限された編集は知識レベルで行われるのに対し、日々の編集は業務レベルで行われた。これならば、知識レベルパターンにうまく適合している。線よりも上にあるオブジェクトはすべて、タイプか長期間存続するポリシーを記述している。**従業員タイプ**は**従業員**に対して効果的にふるまいを与えている。

この開発者が自分の洞察を同僚たちに話したところ、もう1人の開発者がそれとは別の洞察を得た。知識レベルによって構成されたモデルが明確に見えるようになったことで、彼女は前日に自分が何に悩んでいたかを突き止めたのである。実は、2つの別々の概念が同一のオブジェクト内で一体となっている。それを前日に使用された言語で耳にしてはいたものの、はっきりと指摘することができなかったのだ。

従業員タイプは、いずれかの**退職金制度**またはいずれかの給料に割り当てられる。

しかし、これは実際にはユビキタス言語による発言**ではない**。モデルには「給料（payroll）」がないのだ。**使いたい**言語で話していたのであって、持っている言語で話しているわけではなかった。給料の概念はモデルの中に暗黙的に存在し、**従業員タイプ**と一緒にまとめられていた。このことは、知識レベルが分離される前は、それほど明らかではなかった。そのため、前述した重要なフレーズに含まれる複数の要素が、すべて同じレベルで一緒に現れていたのだ…ただ1つを除いて。

この洞察に基づき、彼女は再度リファクタリングして、上記の発言を間違いなくサポートするモデルにした。

オブジェクトを関連づけるルールをユーザがコントロールする必要があったので、チームは暗黙的な知識レベルがあるモデルを作るように駆り立てられた。

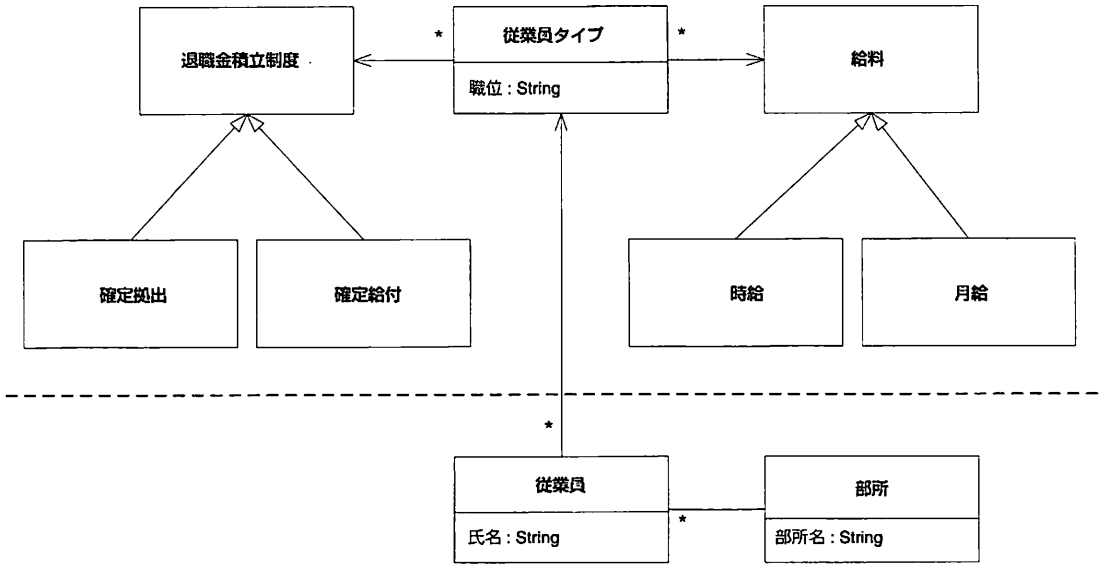


図16.22 給料は今や明示的になり、従業員タイプとは区別されている

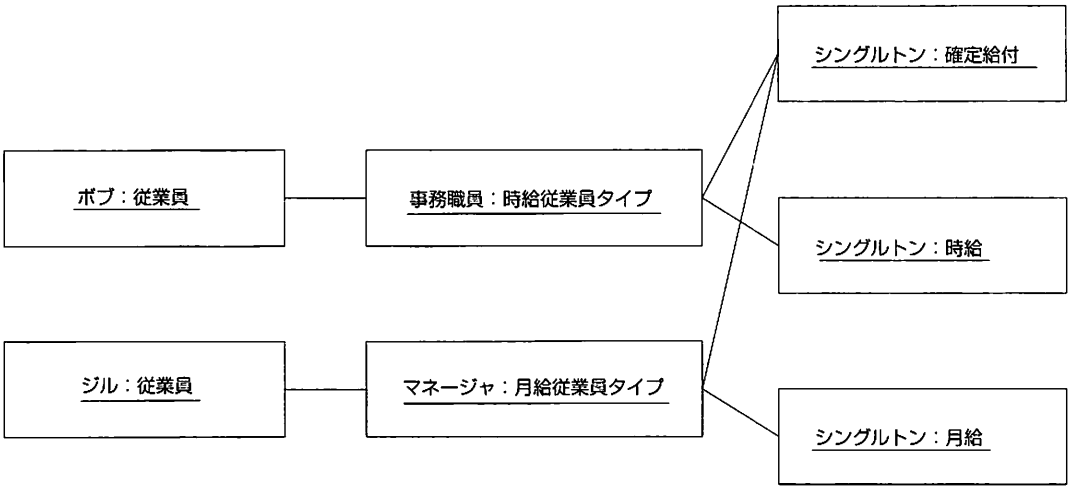


図16.23　今では各従業員タイプに退職金制度と給料がある

知識レベルのヒントとなったのは、特徴的なアクセス制限と「物と物」型の関係性である。いったん整理されると、知識レベルによって明確になったことで、また別の洞察が生み出され、給料を括り出すことによって、2つの重要なドメインの概念のもつれがほぐされた。

知識レベルは、他の大規模な構造と同様、厳密に必要なものではない。これがなくてもオブジェクトは機能するし、**従業員タイプ**と**給料**を分離した洞察を見つけ出して使用することもやはりできただろう。この構造が負担を軽減していないように見える時が来るかもしれず、捨て去られるかもしれない。しかし、今のところはシステムについて役立つストーリーを語り、開発者がモデルと取り組む助けになっているようである。

＊　＊　＊

一見した限りでは、知識レベルは責務のレイヤ、特に「ポリシー」層の特殊ケースのように見えるが、そうではない。一例を挙げると、レベルの場合、依存関係は双方向に向かうが、レイヤの場合には下位層が上位層から独立している。

実は、知識レベルは他の大規模な構造のほとんどと共存することができ、組織に関する付加的な一面を提供するのである。

着脱可能コンポーネントのフレームワーク (PLUGGABLE COMPONENT FRAMEWORK)

深められて蒸留された非常に成熟したモデルにも、ブレイクスルーの機会は訪れる。着脱可能コンポーネントのフレームワークが作用し始めるのは、通常、同一のドメインでいくつかのアプリケーションが実装された後のことなのだ。

* * *

多様なアプリケーションの相互運用が必要で、すべてが同一の抽象に基づきながら独立して設計されている場合、複数の境界づけられたコンテキスト間で変換しなければならないために、統合が制限されてしまう。共有カーネルは緊密に連携していないチームにおいては現実的ではない。重複と断片化のせいで開発とインストールのコストが増大し、相互運用が非常に困難になる。

成功しているプロジェクトの中には、設計をコンポーネントに分割し、それぞれに対して特定のカテゴリの機能に関する責務を負わせているものもある。通常、すべてのコンポーネントは中央のハブに接続され、そのハブがあらゆる必要なプロトコルをサポートし、コンポーネントが提供するインタフェースとの通信方法を理解している。他のコンポーネント接続パターンを用いてもよい。コンポーネントのインタフェースと、それらを接続するハブを設計するには調整が必要だが、こうすることで、内部の設計をより独立して行えるようになる。

広く使用されているいくつかの技術的なフレームワークがこのパターンをサポートしているが、それは二次的な問題である。技術的なフレームワークが必要になるのは、分散や別々のアプリケーション間でのコンポーネント共有といった本質的な技術的問題を解決する場合だけだ。基本的なパターンは、責務を概念的にまとめ上げたものである。これは単一のJavaプログラムの中でも容易に適用できる。

それゆえ：

インタフェースや相互作用にある抽象化されたコアを蒸留し、そのインタフェースの多様な実装を自由に置換できるようにするフレームワークを作成すること。同様に、抽象化されたコアのインタフェースを経由して操作される限り、あらゆるアプリケーションがこのコンポーネントを使用できるようにすること。

高次の抽象はシステムの広い範囲で識別および共有されるのに対し、専門化はモジュール内で起きる。アプリケーションの中心的なハブとなるのは、共有カーネル内の抽象化されたコアである。しかし、カプセル化されたコンポーネントインタフェースの背後に、複数の境界づけられたコンテキストが存在することもあり得る。この構造がとりわけ便利なのは、多くのコンポーネントがいろいろな出自を持つ場合や、コンポーネントが統合のた

めに以前からあるソフトウェアをカプセル化している場合である。

これは、コンポーネントに複数の多様なモデルがなければならないということではない。チームが、継続的な統合を行うか、密接に関係するコンポーネントの集合のための共有カーネルを新たに定義できるのであれば、複数のコンポーネントを単一のコンテキスト内で開発することができる。これらの戦略がすべて着脱可能コンポーネントという大規模な構造の中で容易に共存できるのだ。他の選択肢として、場合によっては、ハブのプラグインインタフェースとして、公表された言語を使用してもよい。

着脱可能コンポーネントのフレームワークにはいくつかマイナス面もある。1つは、適用するのが非常に難しいパターンだということである。インタフェースは正確に設計する必要があり、抽象化されたコアで必要なふるまいをとらえられるだけの深いモデルがなければならない。それ以外の主要なマイナス面としては、アプリケーションの選択肢が限られているということもある。アプリケーションがコアドメインに対する全く別のアプローチを必要とする場合、この構造が妨げとなるのだ。開発者はモデルを専門化することはできるが、抽象化されたコアを変更するには、多様なコンポーネントのプロトコルをすべて変更しなければならない。結果として、コアを継続的に改良するプロセス、つまりより深い洞察へ向けてリファクタリングするプロセスは、多かれ少なかれその場で膠着してしまう。

Fayad と Johnson（2000）は、SEMATECH CIM に関する議論を含むいくつかのドメインにおいて、着脱可能コンポーネントのフレームワークに向けた意欲的な取り組みを詳細に観察している。そういうフレームワークを成功させるには、さまざまな要素が必要である。おそらく、最大の障害は、役に立つフレームワークを設計するのに必要な理解を成熟させることだろう。着脱可能コンポーネントのフレームワークは、プロジェクトで大規模な構造を適用するにあたり、最初に取り組んではならない。最も成功した例では、複数の専用アプリケーションが完全に開発された後に、このパターンが適用されていた。

例 16.4 —— SEMATECH CIM フレームワーク

コンピュータチップを製造する工場では、シリコンウエハのグループ（ロットと呼ばれる）が、機械の間を移動しながら、数百の加工ステップを経て、最終的には極小の電子回路のプリントと焼きつけが完了する。工場が必要としているのは、各ロットを追跡し、ロットに対して行われた加工を正確に記録してから、そのロットを工場従業員がオートメーション設備のいずれかに送り、次の適切な機械で適切な加工が行われるようにするソフトウェアだ。こういうソフトウェアは製造実行システム（MES: Manufacturing Execution System）と呼ばれる。

数十のベンダが製造した数百台のさまざまな機械が使用され、各加工ステップには慎重に仕立てられた製法がある。そういう複雑な混在に対処できるMESソフトウェアの開発は、手ごわいだけでなくひどく高くついた。これに応えて、業界のコンソーシアムであるSEMATECHがCIMフレームワークを開発したのだ。

CIMフレームワークは巨大で入り組んでおり、多くの側面があるが、ここでの問題に深く関わっている点は2つある。まず、このフレームワークは、半導体MESドメインの基本的な概念のための抽象インタフェースを定義する。ここで定義されるのは、つまり、抽象化されたコアのかたちをしたコアドメインである。このインタフェース定義には、ふるまいとセマンティクスの両方が含まれている。

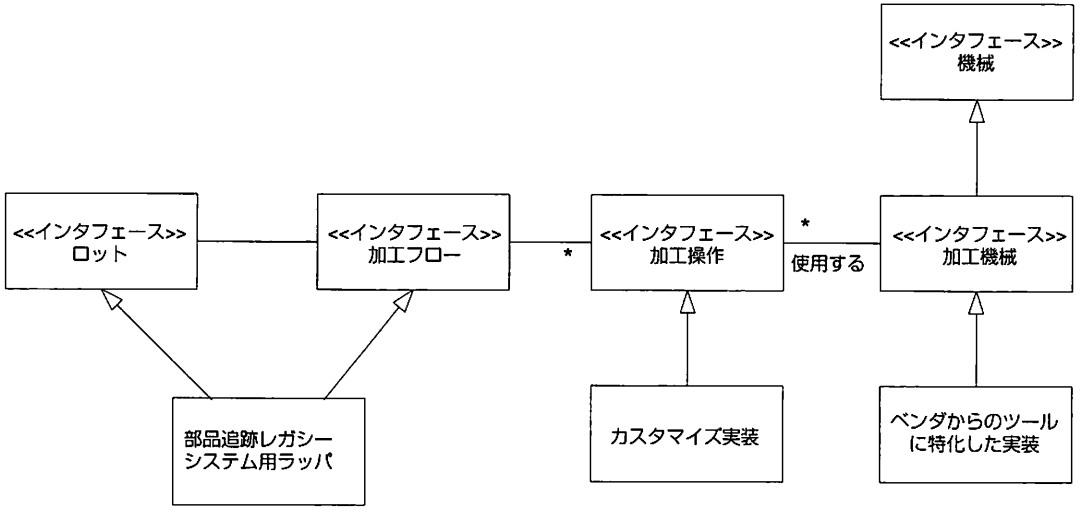


図16.24 CIMインタフェースの非常に単純化されたサブセット、サンプル実装つき

ベンダは新しい機械を製造すると、**加工機械**インタフェースの特殊な実装を開発しなければならない。このインタフェースに忠実なら、機械制御コンポーネントはCIMフレームワークに基づくどんなアプリケーションにもプラグインできるはずである。

これらのインタフェースを定義した後で、SEMATECHはそれらがアプリケーション内で相互作用できるようにするルールを定義した。CIMフレームワークに基づくアプリケーションはどれも、これらインタフェースのサブセットを実装したオブジェクトを取りまとめるプロトコルを実装しなければならない。このプロトコルが実装され、アプリケーションが抽象インタフェースを厳密に守るならば、アプリケーションは、実装に関係なくインタフェースによって約束されたサービスを信頼することができる。インタフェースとそれらを使用するためのプロトコルの組み合わせによって、制約の厳しい大規模な構造が構成される。

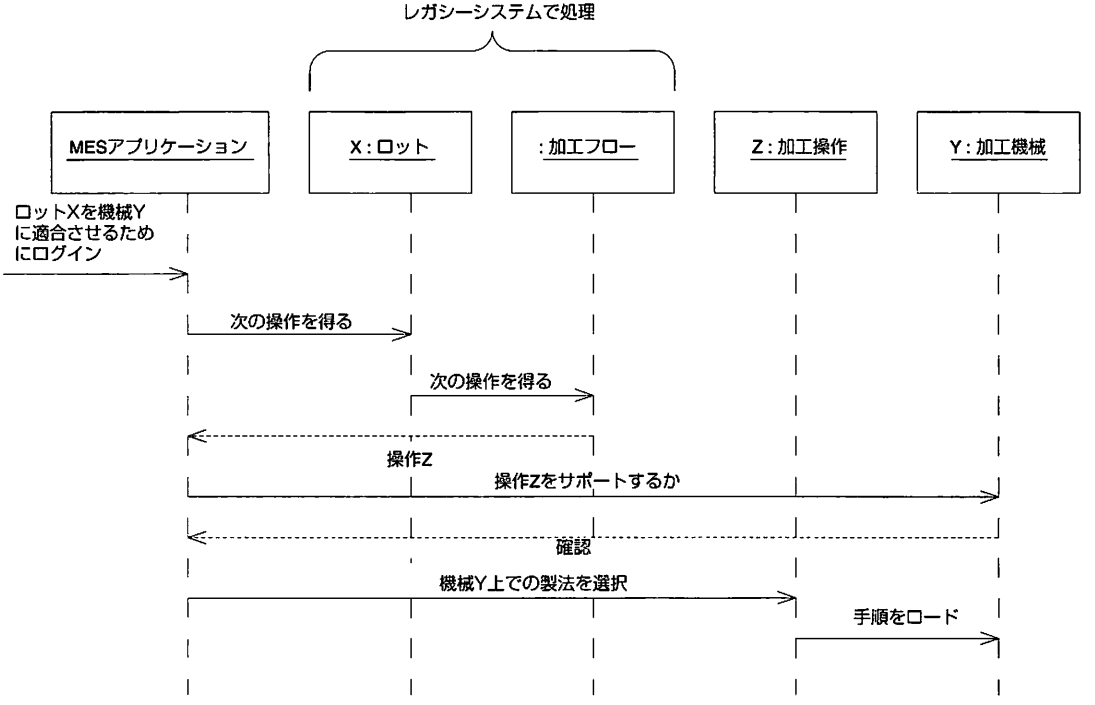


図16.25 ユーザはロットを次の機械に置き、その移動をコンピュータに記録する

このフレームワークには、インフラストラクチャに対する非常にはっきりした要件がある。永続化やトランザクション、イベント、その他の技術的サービスを提供するべくCORBAと緊密に結合しているのだ。しかし、興味深いのは着脱可能コンポーネントのフレームワークの定義だ。これによってソフトウェアは独立して開発され、巨大なシステムへとスムーズに統合できるようになる。こういうシステムの詳細をすべて知っている人は1人もいないが、誰もが概要は理解している。

＊ ＊ ＊

どうすれば、何千という人々が独立して作業して、40,000パネル以上のキルトを作成することができるのか？

シンプルなルールがいくつかあることで、AIDSメモリアルキルトに対して大規模な構造がもたらされ、詳細は各参加者に任されるようになっている。注目すべきは、そのルールがいかに全体の使命（AIDSによって亡くなった人々を追悼すること）に集中しているかと、統合を現実的にするコンポーネントの機能、および、キルトをより広い場所で処理する

（折りたたむなどの）能力である。

キルト用のパネルの作成方法は以下の通り

[AIDS Memorial Quilt Project Webサイト（www.aidsquilt.org）より]

パネルを設計する

追悼する人の名前を含めて下さい。誕生日や亡くなった日付、生まれ故郷などの追加情報を自由に入れて構いません…1パネルにつき1人に限定して下さい…。

材料を選択する

キルトは何度も折りたたんだり広げたりするものなので、耐久性が重要です。接着剤は時が経つと劣化するので、パネルに縫いつけるのが一番です。木綿のズックやポプリンのように、ほどほどの重さで伸縮性のない生地が最も適しています。

デザインは縦でも横でも構いませんが、完成した縁かがり済みのパネルはぴったり3フィート×6フィート（90cm×180cm）として下さい！ 生地を裁断する際は、両側に2～3インチ余分に縁を残して下さい。縁かがりがご自身でできない場合は、こちらで行います。パネルに詰め物は必要ありませんが、補強用の裏地はつけた方が良いでしょう。裏地があれば、地面に広げた時にパネルが汚れませんし、生地が形崩れしないようにするのにも役立ちます。

パネルを作成する

パネルを作成する際には、次のようなテクニックが使えます。

- ○アップリケ：背景となる生地に、布や文字、ちょっとした思い出の品を縫いつけます。長持ちしないので、接着剤には頼らないで下さい！
- ◎塗料：繊維用塗料や退色しない染料を塗るか、消えないインクペンを使用します。「膨れる」塗料は粘り気がありすぎるので使用しないで下さい。
- ○ステンシル：鉛筆でデザインをなぞって生地に転写し、ステンシルをのせた後、ブラシで繊維用塗料を塗るか消えないマーカを使って下さい。
- ○コラージュ：パネルにつける素材は、生地を破くようなもの（ガラスやスパンコールなど）は避け、あまりかさばるものは使わないようにして下さい。
- ◎写真：写真や手紙を含める最適な方法は、アイロン転写シートへ複写して、それを綿100%の生地にアイロンで定着させ、その生地をパネルに縫いつけることです。写真を透明なビニール袋に入れ、それをパネルに縫いつけても構いません（折り目に当たらないように、中心からずらして下さい）。

構造による制約をどの程度厳しくするべきか？

本章で説明している大規模な構造のパターンは、非常に緩やかなシステムのメタファから、制約の厳しい着脱可能コンポーネントのフレームワークまで、幅広い。もちろん、それ以外の構造も可能であり、一般的な構造パターンの中にさえ、ルールを作る際にどこまで制約を厳しくするか、ということについての選択肢が数多くある。

例えば、責務のレイヤはモデルの概念と依存関係を何らか括り出すことを指示する。だ

が、レイヤ間のコミュニケーションパターンを定義するルールを付加することもできる。

ある製造工場のことを考えてみよう。そこではソフトウェアが各部品を機械に送り、その部品は何らかの製法に従って加工される。適切な加工がポリシー層から指図され、業務層で実行される。しかし、実際に現場で間違いが起きることは避けられない。実際の状況はソフトウェアのルールと一致しないのだ。ここで、業務層は**世界をありのままに反映しなければならない**。つまり、部品が偶然間違った機械にセットされた場合、その情報は無条件に受け入れなければならないことになる。何とかして、この例外的な状況を上位層に伝える必要がある。そうすれば、意思決定層が他のポリシーを使用して状況を正すことができる。おそらく、その部品を修理プロセスへコース変更したり、廃棄したりするのだろう。しかし、業務層は上位層のことを何も知らない。コミュニケーションを行う際には、下位層から上位層へという方向も加わった双方向の依存関係を作り出してはならない。

典型的には、この通知は何らかのイベント処理の仕組みを通じて行われる。業務オブジェクトは、状態が変化した時は、常にイベントを生成する。ポリシー層のオブジェクトは、関心の対象となるイベントが下位層から送られるのを待つ。ルールに違反するイベントが発生したら、そのルールは、あるアクション（ルールの定義の一部）を実行して適切に応答するか、さらに上位にあるレイヤのためにイベントを生成する。

銀行業の例では、資産価値が変化する（業務）と、ポートフォリオの各区分の価値も変化する。これらの価値がポートフォリオの配分制限（ポリシー）を超えると、おそらくトレーダに警告が出されるので、収支を回復するために資産を売買することができる。

こうした構造は臨機応変に考え出すこともできるし、特定層のオブジェクトの相互作用において、全員が従う一貫したパターンを決めておいてもよい。制限的な構造ほど統一性は増し、設計は解釈しやすくなる。構造がうまく適合すると、ルールのおかげで開発者は優れた設計ができるようになる。異質な断片も、よりうまく組み合わせられるようだ。

他方、制約によって、開発者が必要とする柔軟性が奪われるかもしれない。非常に独特なコミュニケーションの経路は、境界づけられたコンテキストを越えて適用するのは非現実的である。混成システムで実装技術が異なる場合には、特にそうだ。

したがって、フレームワークを構築して大規模な構造の実装を厳格に管理したいという誘惑には、抗わなければならない。大規模な構造による最も重要な貢献は概念の一貫性であり、ドメインへの洞察をもたらすことである。**構造上の各ルールは開発を容易にするものでなければならない。**

ふさわしい構造へのリファクタリング

今の時代、業界は過剰なアップフロントの設計から脱しつつあるが、大規模な構造をウォーターフォールアーキテクチャという古き悪しき時代への逆行だと見なす人もいる。しかし、実を言えば、役に立つ構造を見つけることのできる唯一の方法は、ドメインと問題に対する非常に深い理解から来るのであり、その理解を得る実践的な方法は、イテレーション型開発プロセスなのである。

進化する秩序に全力で取り組むチームは、プロジェクトライフサイクルを通じて、大規模な構造について、恐れることなく再考しなければならない。そういうチームでは、初期に思いついた構造に乗って行ってはならない。そうした初期には、誰も、ドメインや要件を十分に理解していないからだ。

残念ながら、進化という単語が意味しているのは、最終的な構造が最初は手に入らないということであり、進化の過程で、リファクタリングしながらそれを課していかなければならないということである。これは高くついて、難しいかもしれないが、必要なことなのだ。コストをコントロールして最大の収穫を得るための、一般的方法がいくつかある。

ミニマリズム

コストを抑えるポイントの1つは、構造をシンプルかつ軽量に保つことである。あらゆるものを包括しようとしてはならない。最も深刻な問題にだけ取り組んで、残りは臨機応変に処理すること。

最初のうちは、システムのメタファや2層の責務のレイヤといった緩やかな構造を選択するのがよいだろう。最小限の緩やかな構造といえども、混沌を防ぐ助けとなる軽量の指針を提供することはできるのだ。

コミュニケーションと自己規律

新しい開発とリファクタリングでは、チーム全体が構造に従わなければならない。それには、構造をチーム全体で理解している必要がある。用語法と関係性はユビキタス言語に加えなければならない。

大規模な構造によって提供される語彙は、プロジェクトが幅広くシステムに対処し、さまざまな人が独立して行う意思決定を調和の取れたものにするために使用できる。しかし、ほとんどの大規模な構造は、緩やかな概念的指針であるため、チームは自己を律しなければならない。

関係する多くの人々に忠実に守られないと、構造は減退する傾向がある。構造がモデルや実装の詳細部分に対して持つ関係は、通常、コードでは明示的に表現されず、また機能

テストは構造に依存しない。そのことに加えて、構造は抽象的である傾向があるから、アプリケーションの一貫性を大きな（または複数の）チーム全体で維持するのは難しいかもしれない。

たいていのチームで交わされる類の会話だけでは、一貫性のある大規模な構造をシステム内で維持するのに十分でない。大規模な構造をプロジェクトのユビキタス言語に組み入れて、その言語を全員で執拗に鍛えることが不可欠なのだ。

再構成によってしなやかな設計がもたらされる

次に、構造に対するどんな変更も、大量のリファクタリングにつながるかもしれない。システムの複雑さが増して理解が深まるにつれ、構造は進化していく。構造が変化するたびに、**新しい秩序を忠実に守るためにシステム全体を変更しなければならない。**当然、これは大変な作業だ。

しかし、これを聞いて思うほどにはひどくない。私がこれまで見た限り、通常は、大規模な構造を持つ設計の方が、持たないものより、はるかにかたちを変えやすいのだ。それは、例えば、メタファからレイヤへというように、構造の種類を変更する場合にさえ当てはまるようだ。これについて、すべてを説明することはできない。ただ、一因としては、現在の配置を理解できている方が、再配置するのも容易であり、そして既存の構造があれば、現在の配置を理解しやすくなるということが挙げられる。またそれは、初期の構造を維持するのに必要だった規律が、システムの全側面に浸透しているためでもある。しかし、それ以上のものがあると私は思う。過去に構造が**2つ**あったシステムの方が、変更するのが**さらに容易**だからである。

新しい革ジャケットはこわばっていて着心地が悪いが、一日着れば肘が数回曲げられて、それによって曲げやすくなってくる。さらに数回着れば、肩回りがゆるくなってジャケットは着やすくなる。数か月着用すれば、皮はしなやかになり、着心地がよく動きやすくなる。同じことが、適切な変換によって繰り返しかたちを変えられるモデルにも言えるようだ。増え続ける知識が埋め込まれ、**変更時の主要な軸が識別されて柔軟になる**一方で、安定している側面が簡略化される。根底にあるドメインのより広い概念の輪郭が、モデルの構造に現れてくる。

蒸留によって負荷が軽減される

モデルに適用すべきもう1つの重要なフォースに、継続的蒸留がある。これは、構造を変更することの難しさをさまざまな方法で軽減させる。まず、コアドメインからメカニズムや汎用サブドメイン、その他の補助的構造を削除することにより、再構成すべきものが単純に減るかもしれない。

可能であれば、それらの補助的要素は大規模な構造にシンプルに収まるよう定義すべきである。例えば、責務のレイヤのシステムでは、汎用サブドメインが1つのレイヤにうまく収まるように定義できる。着脱可能コンポーネントを用いると、汎用サブドメイン全体を単一のコンポーネントに所有させることもできるし、関連するコンポーネントの集合にとっての、共有カーネルにすることもできる。こうした補助的要素は、構造内で適切な場所を見つけるためにリファクタリングしなければならないこともある。しかし、こうした要素はコアドメインから独立して移動して、より狭い範囲に集中する傾向があるので、わかりやすいものになる。最終的には、それほど重要ではなくなるので、その部分を改良しなくてもよくなる。

蒸留の原則と、より深い洞察に向けたリファクタリングの原則は、大規模な構造それ自体にも適用される。例えばレイヤは、最初のうちはドメインについての表面的な理解に基づいて選択されるかもしれない。しかし、システムの基本的な責務を表現するより深い抽象によって、こうしたレイヤは次第に置き換えられる。深い抽象がこのように明確であることにより、設計について深く考えられるようになるのであり、それこそが目標なのだ。同時に、明確さは手段の一部でもある。明確であることによって、大規模なシステムの操作が、より容易で安全になるからだ。

第17章

戦略をまとめ上げる

前の3つの章では、ドメイン駆動の戦略的設計のための多くの原則とテクニックを紹介した。巨大で複雑なシステムでは、こうした原則やテクニックを、1つの設計に対していくつかまとめて適用する必要があるかもしれない。大規模な構造をコンテキストマップとどう共存させるか？ 構成要素はどこになじむか？ 最初に何をすべきか？ 次は？ その次は？ 戦略を考え出すにはどうすればよいのか？

大規模な構造と境界づけられたコンテキストを組み合わせる

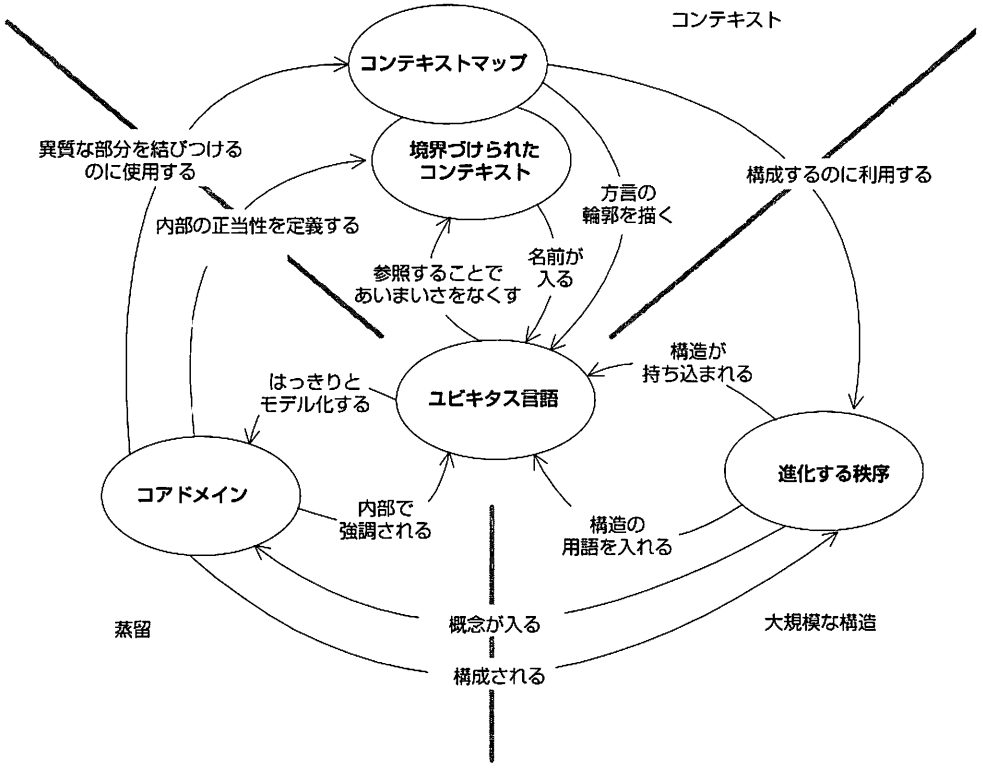


図 17.1

戦略的設計の3つの基本原則（コンテキスト、蒸留、大規模な構造）は互いに置き換わるものではない。いろいろな方法で補完し合って相互に作用するのだ。例えば、大規模な構造は、1つの境界づけられたコンテキスト内に存在することもあれば、多くのコンテキストを横断してコンテキストマップを構成することもある。

前述した責務のレイヤの例は、1つの境界づけられたコンテキストに限定されていた。これは、責務のレイヤの考え方を説明する最も簡単な方法であり、このパターンの一般的な使い方でもある。こういうシンプルなシナリオでは、それぞれのレイヤにつけられた名前の持つ意味は、該当するコンテキスト内に限られる。これは、そのコンテキスト内に存在するモデル要素やサブシステムのインタフェースの名前と同様である。

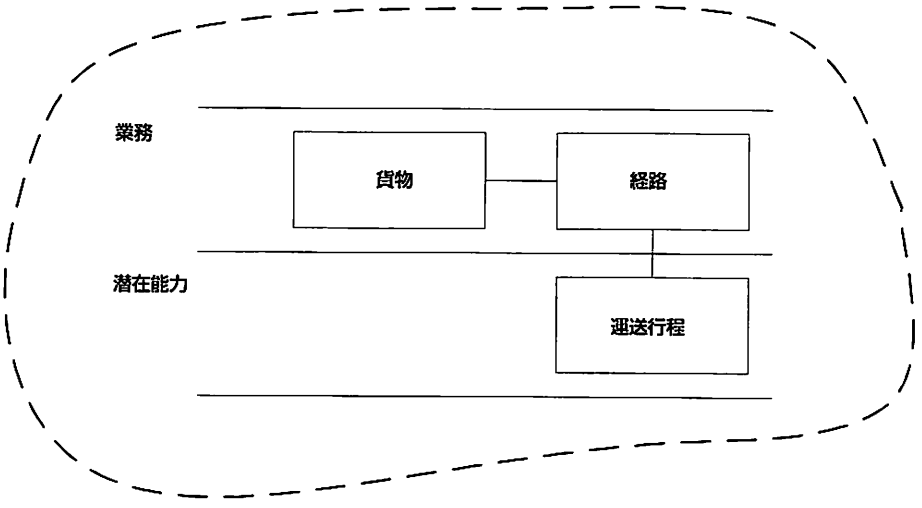


図17.2 単一の**境界づけられたコンテキスト**内でモデルを構築する

こういう局所的な構造は、きわめて入り組んでいながらも統一されたモデルにおいて役に立つ。こうした構造により、単一の境界づけられたコンテキスト内で維持できる複雑さの限界が押し上げられる。

しかし、多くのプロジェクトに見られる、より大きな課題は、完全に別々の部分がどう組み合わされるかを理解することである。そういう部分は、別々のコンテキストに区分けされることもある。しかし、それぞれは統合されたシステム全体でどのような役割を果たし、相互にどう関係し合っているのだろうか？ そうした問いに答えるためには、大規模な構造を使って、コンテキストマップを構成すればよい。この場合、構造に関する用語法はプロジェクト全体（または、少なくとも明確に境界が定められている部分）に適用される。

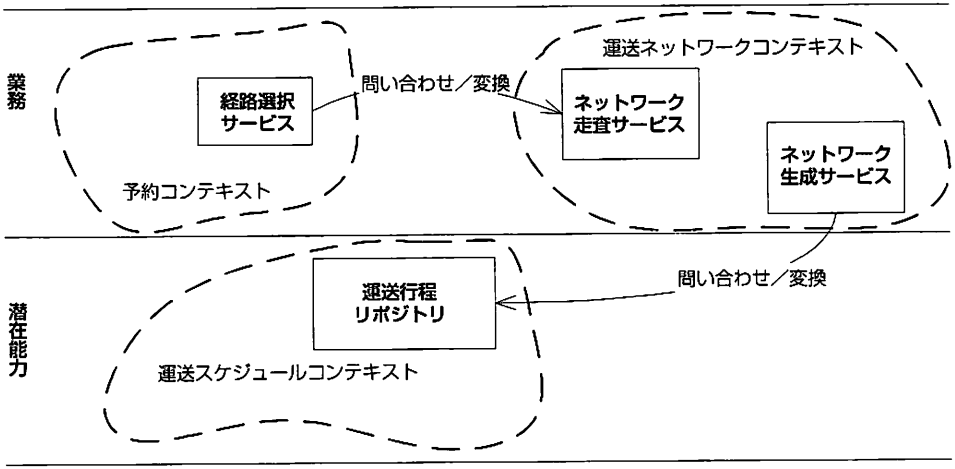


図17.3 別々の**境界づけられたコンテキスト**にあるコンポーネントの関係に課せられた構造

責務のレイヤを採用したいのに、レガシーシステムがあって、その構成が望ましい大規模な構造と一致しないと仮定しよう。レイヤをあきらめなければならないだろうか？ そんなことはないが、構造内でレガシーシステムが実際にどこかに置かれることを認める必要はある。実はそうすることで、レガシーシステムを特徴づけることができるかもしれない。レガシーシステムの提供するサービスが、実はいくつかのレイヤだけに限られているかもしれない。レガシーシステムが特定の責務のレイヤに収まると言えるということは、レガシーシステムのスコープと役割に関する主要な側面を簡潔に記述したことになるのだ。

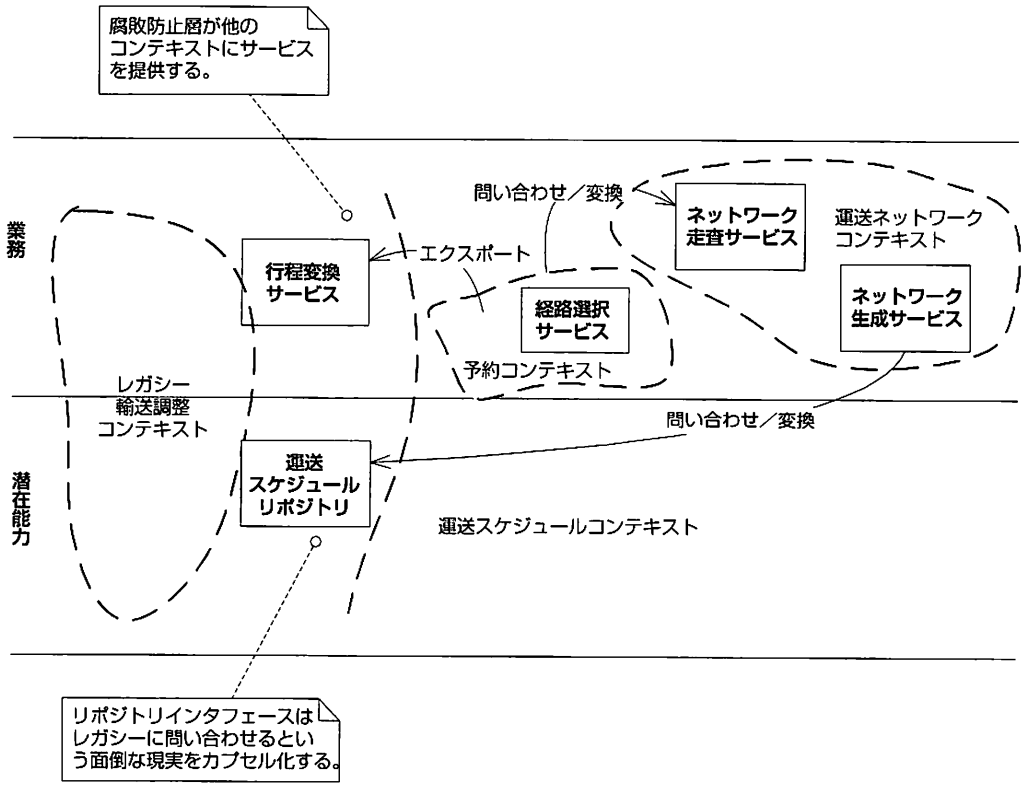


図17.4 いくつかのコンポーネントがレイヤをまたがることを容認する構造

レガシーサブシステムが持つ能力にファサード経由でアクセスしているのであれば、そのファサードが提供する各サービスを、1つのレイヤに収まるように設計できるかもしれない。

輸送調整アプリケーションはこの例ではレガシーシステムとなっているので、内部は一様な塊として示される。しかし、コンテキストマップをまたがった大規模な構造が、適切に構築されたプロジェクトであれば、チームは自分たちのコンテキスト内で、よく知っている同じレイヤに従って、モデルを秩序立てることを選択することもできる。

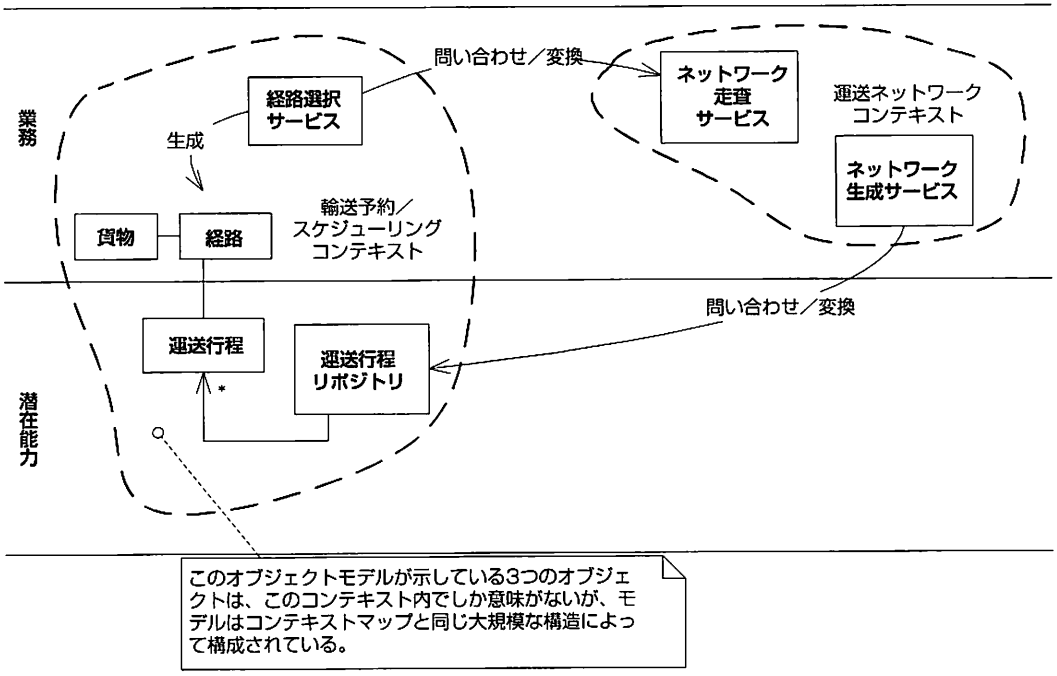


図17.5 同じ構造が1つの**コンテキスト**内と**コンテキストマップ**全体に適用された

もちろん、それぞれの境界づけられたコンテキストは独自の名前空間であるため、1つの構造を使用して、1つのコンテキスト内でモデルを構成することができるし、隣接するコンテキストにはまた別の構造を使用し、さらに別の構造でコンテキストマップを構成することもできる。しかし、この方向に進みすぎると、プロジェクトのための概念を統一した集合という、大規模な構造の持つ価値を蝕みかねない。

大規模な構造と蒸留を組み合わせる

大規模な構造と蒸留の概念も相互に補完し合う。大規模な構造によって、コアドメイン内の関係性と汎用サブドメイン間の関係が説明しやすくなる。

同時に、**大規模な構造自体がコアドメインの重要部分になっていることもある。**例えば、潜在能力、業務、ポリシーおよび意思決定支援からなるレイヤを区別することで、ソフトウェアが取り組むビジネスの問題にとって基本的な洞察が蒸留される。この洞察がとりわけ役立つのは、プロジェクトが多くの境界づけられたコンテキストに切り分けられていて、コアドメインにあるモデルオブジェクトがプロジェクトの大半にとって意味を持たない場合である。

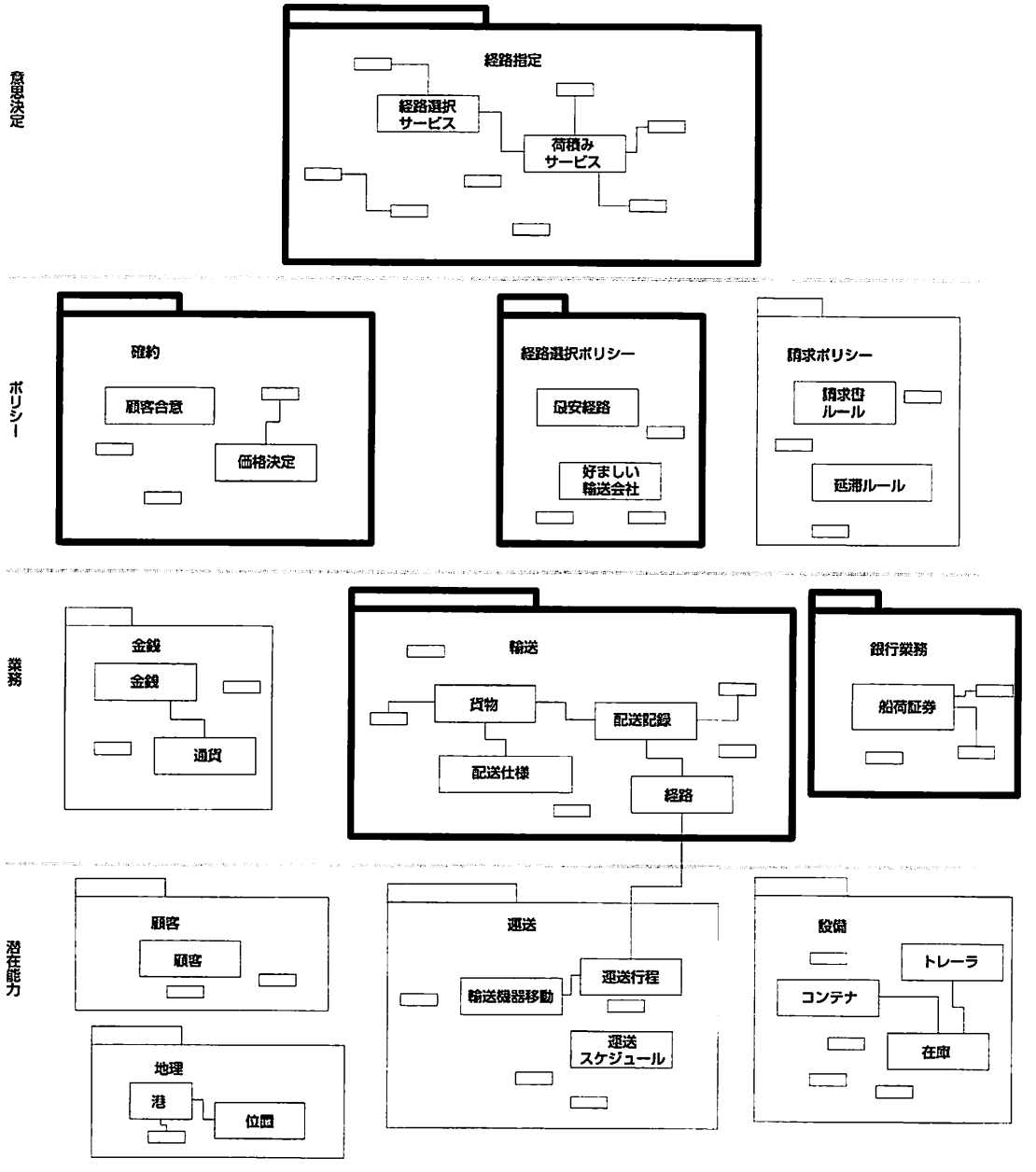


図17.6 **コアドメイン**（枠が太線）と**汎用サブドメイン**のモジュールがレイヤによって明確になっている

まず評価する

プロジェクトにおいて戦略的設計に取り組む時には、現在の状況を明確に評価することから始める必要がある。

1. コンテキストマップを描くこと。一貫したものが描けるか、それともあいまいな場所があるか？
2. プロジェクトでの言語の使われ方に注意を払うこと。ユビキタス言語はあるか？ それは開発を支援できるくらい豊かであるか？
3. 何が重要であるかを理解すること。コアドメインは識別されているか？ ドメインビジョン声明文はあるか？ それを書くことはできるか？
4. プロジェクトで使われている技術はモデル駆動設計に向いているか？
5. チームの開発者は必要な技術的スキルを持っているか？
6. 開発者はドメインについてよく知っているか？ ドメインに**関心**を持っているか？

完璧な答えが見つからないのは当然である。プロジェクトに関して現在知っていることは、将来知るであろうものより少ない。しかし、前述の質問によって確かな出発点が得られる。これらの質問に対する具体的な初期回答が得られる頃には、最も緊急に行う必要があるのは何かを示す洞察が得られ始めているだろう。時が経つにつれ、その回答、特にコンテキストマップやドメインビジョン声明文、その他作成した成果物すべてを改良して、変化した状況と新しい洞察を反映することができる。

誰が戦略を策定するのか？

伝統的には、アーキテクチャは人から人へ伝えられるものであり、アプリケーション開発が始まる前に、組織においてアプリケーション開発チームよりも権力を持つチームによって作成される。しかし、そうでなければならないわけではない。この方法はたいていうまくいかないのだ。

戦略的設計は、その名が示すとおり、プロジェクト全体に適用しなければならない。プロジェクトを編成する方法は多く存在するのであり、私は規範的になりすぎるのは好まない。それでも、すべての意思決定プロセスを効果的にするには、何らかの原理が必要になる。

まず、実践にあたって何らかの価値を提供していた、2つのスタイルを簡単に見てみよう(こうして、権力のあるチームが決めたアーキテクチャが降りて来るという、時代遅れの「天からの知恵」スタイルには触れない)。

アプリケーション開発から現れる構造

非常に優れたコミュニケーション能力を持つメンバで構成される自己規律型のチームは、中央の権威がなくても機能し、進化する秩序に従って、共有された原則の集合を生み出せる。したがって、その秩序は命令によらず有機的に成長するのだ。

これはエクストリームプログラミングチームの典型的なモデルである。理論上は、こうした構造はどのプログラミングペアの洞察からでも、全く自然発生的に現れる可能性がある。だが、たいていの場合、大規模な構造を監督する責任を、個人やチーム内の小さなグループに負わせると、構造の統一を維持しやすくなる。このアプローチが特にうまくいくのは、そういう非公式なリーダが実践的な開発者である場合だ。こうしたリーダは、調停役や伝達役であって、アイデアの唯一の源泉でないのがよい。私の見てきたエクストリームプログラミングチームでは、こうした戦略的設計のリーダがしばしば技術指導者というかたちで自然発生的に現れてきたようだ。その自然なリーダが誰であれ、その人は依然として開発チームの1メンバである。当然、開発チームには、プロジェクト全体に影響することになる設計上の意思決定を行う才能のある人が、少なくとも数人はいなければならない。

大規模な構造が複数チームにまたがる場合、密接に協力し合っているチーム同士が非公式に協調し始めるかもしれない。そういう状況では、アプリケーションチームそれぞれで大規模な構造のアイデアにつながる発見をすることがまだあるものの、その後で、多様なチームの代表で構成された非公式な委員会によって、特定の選択肢が議論される。設計が与える影響を評価した後で、参加者たちはその設計を採用するか、修正するか、または見送るかを決定すればよい。複数のチームがこのように緩やかな協力関係で一緒に仕事を進めようとする。こうした取り決めが機能するのは、チームの数が比較的少なく、どのチームも互いに協調することに積極的で、似たような設計能力を持っており、単一の大規模な構造で満たせるくらい構造への要求が類似している場合である。

顧客に焦点を合わせたアーキテクチャチーム

戦略が複数のチームで共有される場合、意思決定を何らかのかたちで中央集権化するのは確かに魅力的に見える。中央集権化を行ったからと言って、象牙の塔のアーキテクトが作ったモデルが失敗する、という典型に陥るとは限らない。アーキテクチャチームは、さまざまなアプリケーションチームに対して同僚として行動することができ、境界づけられたコンテキストやチーム間の技術的な問題に加え、大規模な構造も調整して一致させやすくする。そこで貢献するためには、アプリケーション開発を重要視するという心構えを持っていなければならない。

組織図を見ると、このチームは従来のアーキテクチャチームと同じように見えるだろうが、実際にはすべての活動において異なっている。チームメンバは開発に対する真の協力

者で、開発者と一緒にパターンを見つけ、さまざまなチームと実験して蒸留に到達し、自ら手を動かすのである。

このシナリオは2回見たことがあるが、プロジェクトの終わりには、次に挙げることをほとんどすべて実践するアーキテクトリーダが誕生していた。

戦略的設計上の意思決定を行うために欠かせない6つのこと

意思決定はチーム全体に伝えなければならない。

当然、誰もが戦略を理解してそれに従うのでなければ意味がない。この要件から、公式の「権威」を持つ中央集権型アーキテクチャチームを中心にチームが構成されることになる。この権威によって、同一のルールがどこででも適用されることになるのだ。皮肉なことに、象牙の塔のアーキテクトは無視されたり抜け道を作られたりすることも多い。アーキテクトが自分たちのルールを実際のプロジェクトに適用するのに、実際に手を動かしてフィードバックを得ることがなく、そのせいで非実用的な案ができていれば、開発者はそれを無視したり、抜け道を探したりするほかないのだ。

コミュニケーションの非常に盛んなプロジェクトでは、アプリケーションチームから現れる戦略的設計が、実際に全員に対して、より効果的に伝わるかもしれない。そういう戦略は当面の問題と関連が深く、その権威は聡明なグループによる意思決定に帰属する。

どんなシステムであっても、経営者が与える権威より、開発者が戦略に対して持つ実際の関係の方を重視すること。

意思決定プロセスはフィードバックを吸収しなければならない。

構成原理や大規模な構造を作成したり、それらにある機微を蒸留したりするには、プロジェクトの要求とドメインの概念に対する本当に深い理解が求められる。そこまで深い知識を唯一持っているのが、アプリケーション開発チームのメンバである。だからこそ、多くのアーキテクトに疑いようのない才能があるにも関わらず、アーキテクチャチームが作成したアプリケーションアーキテクチャは、めったに役に立たないのだ。

技術的なインフラストラクチャやアーキテクチャとは異なり、戦略的設計自体はコードを書くものではないが、開発全体に影響を与えることになる。間違いなく必要になるのは、アプリケーション開発チームの参加である。熟練のアーキテクトはさまざまなチームから寄せられる考えに耳を傾けて、汎用的な解決策の開発をスムーズに行うことができるだろう。

私が一緒に仕事をしたことのある技術的なアーキテクチャチームでは、メンバがそのフレームワークを使おうとしているさまざまなチームの間を回って意見を聞いていた。こうすることで、アーキテクチャチームは開発者が直面する課題の実地経験を吸い上げると同時に、フレームワークを適用する際の機微についても、知識の引き継ぎを行っていたので

ある。戦略的設計にも、これと同じ密接なフィードバックループが必要である。

計画は進化を許容しなければならない。

効果的なソフトウェア開発は、非常にダイナミックなプロセスである。最高レベルの意思決定を、変更できないものとして固定してしまうと、チームが変化に対応しなければならなくなった時の選択肢が狭くなってしまう。進化する秩序によって、この落とし穴は回避できる。継続的な変更を重視することで、洞察が深まるのに応じて大規模な構造が作られるのだ。

あまりにも多くの設計上の意思決定が予定されていると、課せられた問題を解決するための柔軟性がないせいで、開発チームの進行を妨げかねない。バランスを取るための原理には価値があるだろうが、それは開発プロジェクトの進行に合わせて成長し、変化しなければならないものである。ただでさえ困難な仕事をしているアプリケーション開発者から、あまり多くを奪ってはならないのだ。

強力なフィードバックがあれば、アプリケーション構築において障害に遭遇したり、予期せぬ機会を発見したりするのに合わせて、新しい構想が姿を現す。

アーキテクチャチームが、最も優秀な人材をすべて吸い上げてはならない。

このレベルの設計には洗練された技術が必要であり、そうした技術を持つ人はおそらく多くない。マネージャは、技術的に最も才能のある開発者をアーキテクチャチームとインフラストラクチャチームへ移動させがちだが、それはこうした上級設計者のスキルを活用したいからである。開発者としても、広範囲にわたる影響力を持ったり、「もっとおもしろい」問題に取り組んだりする機会に魅力を感じる。しかも、エリートチームのメンバであることには名声もついてくる。

このような経緯でありがちなのが、技術的には最も未熟な開発者だけを実際のアプリケーション構築に残すというものだ。しかし、優れたアプリケーションを構築するには設計スキルが必要だから、これでは失敗を準備しているようなものだ。戦略チームが素晴しい戦略的設計を作り出しても、それに従う上で必要な設計上の高度な知識がアプリケーションチームにないのだ。

逆に、アーキテクチャチームに加わることがほぼないのが、設計スキルは不足していても該当するドメインでの経験が最も豊富な開発者である。戦略的設計は純粋に技術的な作業ではないから、ドメインについての深い知識を持つ開発者を除外してしまうと、アーキテクトの作業をさらに妨げることになる。また、ドメインエキスパートも必要である。

すべてのアプリケーションチームに強力な設計者を入れることが不可欠だ。戦略的設計を試みるどのチームにも、ドメインについての知識が欠かせない。高度なスキルを持つ設計者を、単にもっと多く雇う必要があるかもしれない。アーキテクチャチームを非常勤にしておくことが一助になるかもしれない。効果的な方法は多数あるはずだが、どんなに有

能な戦略チームであっても、有能なアプリケーションチームをパートナとして持たなければならない。

戦略的設計にはミニマリズムと謙虚さが必要である。

蒸留とミニマリズムはどんな優れた設計にも欠かせないものだが、戦略的設計にとっては、ミニマリズムはより一層重要である。ごくわずかなずれでも妨げになってしまう可能性があるのだ。離れた場所にいるアーキテクチャチームは特に慎重にならなければならない。アプリケーションチームが進む道に、自分たちが障害物を置いているかもしれないという意識があまりないからだ。同時に、主要な責任に対して意気込んで、アーキテクトが浮かれてしまうかもしれない。この現象を私は何度も見たことがあるし、私自身がやってしまったこともある。1つの名案が次へとつながり、最終的に生産性の低い、作り込みすぎたアーキテクチャができ上がってしまう。

そうではなく、自制しながら、設計を特に明確にするものだけを含むように削ぎ落とされた構成原理とコアモデルを作り出さなければならない。実際、ほぼすべてのものが何かの妨げになるのだから、各要素には相応の価値が求められる。自分にとっては最高の考えでも、誰かの妨げになるかもしれないと認識するためには、謙虚でなければならない。

オブジェクトはスペシャリストだが、開発者はジェネラリストである。

優れたオブジェクト設計の本質は、各オブジェクトに明白で限定された責務を与え、相互依存関係を最小限に減らすことにある。だが時には、チームでの交流を、ソフトウェアにおいてあるべき姿と同じくらい整理しようとしてしまうことがある。だが、うまくいっているプロジェクトには、他人のことに首を突っ込む人々が多い。開発者がフレームワークを試し、アーキテクトがアプリケーションコードを書く。全員が全員と話をする。これは効果的な混沌だ。オブジェクトをスペシャリストに仕立てて、開発者はジェネラリストにすること。

私は**戦略的設計**とその他の設計を区別したが、これはそこに伴う作業を明白にするためであった。指摘しておかなければならないのだが、2種類の設計作業があるからといって、人々を2種類に分けることはない。深いモデルに基づいてしなやかな設計を作成することは高度な設計作業だが、詳細がきわめて重要なので、こうした設計作業はコードを使って仕事をする人が行わなければならない。戦略的設計はアプリケーション設計から生まれてくるものだが、活動に関する全体的な見方も必要とする。こうした活動は、もしかしたら複数チームにまたがっているかもしれない。好んで求められるのは、作業を細切れにし、設計のエキスパートがビジネスを知る必要がないようにすると同時に、ドメインエキスパートが技術を理解しなくても済むようにする方法だ。個人が学ぶことのできる量には限界があるが、過度の専門化はドメイン駆動設計から推進力を奪ってしまう。

同じことが技術的なフレームワークにも当てはまる

技術的なフレームワークがあると、アプリケーション開発はドメイン層も含めて大いに加速する。これはインフラストラクチャ層を提供してアプリケーションを基本的なサービスの実装から解放し、ドメインをその他の関心事から隔離しやすくすることによるものだ。しかし、アーキテクチャが**ドメインモデルの表現力豊かな実装と容易な変更を妨げかねない**、というリスクが存在する。このことはフレームワーク設計者にドメイン層やアプリケーション層へ足を踏み入れるつもりが全くない場合にも発生し得る。

戦略的設計のマイナス面を制限するのと同じ方向づけが、技術的アーキテクチャでも有効なことがある。進化とミニマリズム、およびアプリケーション開発チームの参加によって、サービスとルールの集合が継続的に改良されるが、これは妨げになることがなく、アプリケーション開発に間違いなく役立つのだ。この道を辿らないアーキテクチャは、アプリケーション開発の創造性を抑圧してしまうか、アーキテクチャそのものを問題から迂回させて、アプリケーション開発を事実上アーキテクチャのないものにしてしまうかのいずれかである。

確実にフレームワークを台無しにする独特の態度が1つある。

素人にもわかるフレームワークを作成してはならない。

設計ができるほどには賢くない人も開発者の中にいると想定したチーム分割は、失敗する可能性が高い。これはアプリケーション開発の難しさを過小評価しているからである。設計ができるほどには賢くないなら、ソフトウェア開発を担当させてはならない。逆に十分に賢いのであれば、大事に面倒を見ようとすると、かえって担当者が必要とするツールとの間に障壁を作り出すだけである。

この態度はチーム間の関係も害する。私の過去の経験でも、最終的にこうした傲慢なチームができ上がり、関わっているのが恥ずかしくて、会話のたびに開発者に謝っていたことがある（あいにく私には、そういうチームを変えることができた経験がない）。

さて、**ビジネスと無関係な技術的詳細をカプセル化すること**は、私が非難しているプリパッケージの類とは全く別物である。フレームワークがあることによって、開発者は強力な抽象とツールを手にし、面倒な単純作業から解放される。一般化できるやり方で違いを説明するのは難しいが、フレームワーク設計者に、そのツール／フレームワーク／コンポーネントをどんな人に使用して欲しいかを質問すれば、違いはわかる。設計者がフレームワークのユーザに高い敬意を払っているようなら、おそらく正しい方向を向いているだろう。

マスタプランに注意すること

Christopher Alexanderが率いるアーキテクト（実際の建築物を設計する建築家）のグループは、建築と都市計画の分野で段階的成長を提唱した。彼らはマスタプランが失敗する理由を実にうまく説明している。

> 何らかの計画プロセスがなければ、ケンブリッジ大学の根底にあるのと同じくらい深遠で調和のとれた秩序に近いものを、オレゴン大学が持てるようになる見込みは全くない。
>
> マスタプランは、この難題に取り組む伝統的な方法である。マスタプランが目指すのは、十分な指針を定めることによって、全体としての環境に一貫性を提供しながら、個々の建築物やオープンスペースには局所的な要求に適合する自由を残すことである。
>
> …そして、この将来の大学にある多様な部分すべてが、統一された全体を形作る。各部分は設計のスロットに差し込まれただけだからである。
>
> …だが実際には、マスタプランは失敗する。マスタプランによって作り出されるものが全体主義的な秩序であって、有機的な秩序ではないからだ。あまりにも厳格すぎるため、共同体の生活で当然発生する、自然で予想できない変化に対し、容易には適合できないのだ。こういう変化が起こるにつれて、マスタプランは時代遅れとなり、守られなくなる。マスタプランが守られる範囲においても…建築物やヒューマンスケール、バランスの取れた機能性などのつながりに関する定義が不十分で、各場所で建築および設計活動それぞれを、全体としての環境とうまく結びつけるのに役立つことはない。
>
> …そういう方向に進もうとするのは、子供の塗り絵に色を塗るようなものである。…そんなプロセスから生まれる秩序は、せいぜいありふれたものだ。
>
> …したがって、有機的秩序の源としては、マスタプランは正確でありすぎると同時に、正確さが足りない。全体性は正確すぎるが、詳細は十分に正確ではないのだ。
>
> …マスタプランの存在はユーザを遠ざける。[これは、定義上] 共同体のメンバが自らの共同体の将来の姿にほとんど影響を与えることができないからであり、それは重要な決定のほとんどがすでになされているからだ。
>
> 『オレゴン大学の実験（The Oregon Experiment）』p.16-28（Alexander他 1975）より

Alexanderと彼の仲間が代わりに提唱したのは、段階的成長に関するあらゆる活動に対して、共同体のメンバ全員が適用できる**原理の集合**である。それによって、状況にうまく適合した「有機的秩序（organic order）」が姿を現すようにしたのだ。

結　論

エピローグ

最先端のプロジェクトに従事して、興味深いアイデアとツールを使って実験するのは非常にやりがいのあることだが、そのソフトウェアに生産的な使い道がなければ、そうした経験も私にとっては空しいものにすぎない。実際、成功したかどうかの真の評価は、ソフトウェアが長期間にわたってどれだけ役立つかで決まる。かつて携わったプロジェクトのその後の様子を、私は数年にわたって追跡調査することができた。

ここではそのうちの5つを取り上げるが、どれも真剣にドメイン駆動設計に取り組んだプロジェクトである。ただし、当然のことながら、体系的に行ったわけではないし、ドメイン駆動設計という名で呼ばれてもいなかった。これらのプロジェクトは、どれもソフトウェアを完成させた。中には、なんとか持ちこたえてモデル駆動設計を作り出したものもあるが、そこから外れてしまったものも1つある。何年もの間、成長と変化を続けたアプリケーションがある一方で、成長が止まってしまったものが1つ、若くして死んでしまったものがもう1つある。

第1章で説明したPCB設計ソフトウェアは、その分野のベータ版ユーザの間で大当たりした。ただ、残念なことに、プロジェクトを始めた新興企業がマーケティングで完全に失敗し、最終的にはお蔵入りとなった。現在、このソフトウェアを使っているのは、ベータ版プログラムのコピーを持っていた一握りのPCBエンジニアだけである。後ろ盾のなくなった他のソフトウェアと同様、統合しているプログラムのどれかに致命的な変更が行われるまで、このソフトウェアは使われ続けるだろう。

第9章で説明したローンソフトウェアは成功して、例のブレイクスルーの後、3年間にわたって、ほぼ同じように進化を続けた。3年経った時点で、プロジェクトは独立した企業と



新しく植樹されたオリーブ畑

してスピンオフした。この組織再編の混乱の中で、プロジェクトを最初から率いてきたプロジェクトマネージャが解雇され、コアとなっていた開発者も数人、プロジェクトマネージャと共に去った。新しくなったチームの設計哲学は若干異なっていて、オブジェクトモデリングにそれほど傾倒していなかった。それでも、複雑なふるまいのあるドメイン層は他から区別された状態に保ち、開発チームの持っていたドメインについての知識を引き続き尊重した。スピンオフから7年経ったが、このソフトウェアには新機能が追加され続けている。また、この分野においてはトップクラスのアプリケーションになっていて、利用する機関も増え、企業に最大の収益をもたらしている。

ドメイン駆動のアプローチがさらに普及するまでは、多くのプロジェクトで、興味深いソフトウェアが、非常に高い生産性で短期間に構築されるだろう。だが、結局、こうしたプロジェクトは変質して、従来のプロジェクトと同じようなものになる。それまでに蒸留された、深いモデルの持つ力を十分に利用することはできないだろうし、ましてや拡充することは不可能だろう。もっと多くを望むこともできるであろうが、実はこういうプロジェクトも、長年にわたってユーザに安定した価値を届けているという意味では、成功なのである。

あるプロジェクトで、私はもう1人の開発者とペアになり、顧客が主力製品を作るのに必要としていたユーティリティを作成した。機能はきわめて入り組んでおり、複雑に組み合わされていた。私はプロジェクトの仕事を楽しみ、我々は抽象化されたコアのある、しなやかな設計を作り出した。だが、引き継ぎを境に、最初にこのソフトウェアを開発していた全員が、開発に関わらないことになってしまった。あまりにも突然な変化だったので、複数の要素を結合できるようにしている設計の特徴は理解されず、より一般的なケース文



7年後

によって置き換えられるのだろうと思った。だが、最初はそうはならなかった。引き継いだ時点で、パッケージには徹底的なテストスイートと蒸留ドキュメントが含まれていた。新しいチームメンバは、そのドキュメントを指針として調査を行い、あれこれ詳しく見ていくうちに、設計の示す可能性に胸を躍らせ始めた。1年経って彼らに感想を聞いてわかったのは、ユビキタス言語が別のチームにまで広がり、進化し続けながら生きているということだった。

それから、さらに1年後、私は、また違った話を耳にした。開発チームは新しい要件に直面したのだが、開発者たちは、引き継いだ設計でそれをどう実現したらよいか、全くわからなかったというのである。そこで、ほとんど見る影もなくなるほど、設計を変更せざるを得なかったという。私がさらに詳しく調べてみると、我々のモデルにあったいくつかの側面が、問題を解決するにはぎこちないものであったことがわかった。まさにそういう瞬間にこそ、より深いモデルへ向かうブレイクスルーが可能になるものなのだ。特にこの場合のように、ドメインについての深い知識と経験を開発者が蓄積している場合はそうである。実際、彼らは次から次へと新しい洞察を得て、最終的には、そうした洞察に基づいて、モデルと設計を一新したのだ。

この話をした時、彼らは角が立たないように、私に気を遣っていた。自分の仕事をほとんど破棄されて、私ががっかりするのではないかと考えたのだろう。だが私は、自分の設計に関してそこまで感情的にはならない。設計が成功しているかどうかは、必ずしも均衡を保っているかどうかで評価されるわけではない。人々が頼りにしているシステムを1つ選んで、それをわかりにくくするだけでも、触れてはならないレガシーとして永遠に生き続けることになるだろう。深いモデルによって、明確なビジョンがもたらされ、そのビジョンが新しい洞察を生み出す一方で、しなやかな設計によって継続的な変更が容易になる。彼らの考案したモデルは、より深いものであり、ユーザの実際の問題によりよく合致していた。彼らの設計が現実の問題を解決したのだ。変化することはソフトウェアの本質であり、このプログラムは、所有するチームの手によって進化し続けてきたのである。

本書のあちこちで取り上げた輸送の例は、大手の国際コンテナ輸送会社のプロジェクトをおおよそ基にしている。当初、プロジェクトのリーダたちはドメイン駆動のアプローチに傾倒していたが、それを十分にサポートできる開発文化を作り出すことはなかった。設計スキルとオブジェクト経験のレベルに大きな差のある複数のチームがモジュールの作成に着手し、チームリーダ間の非公式な協力と顧客を重視したアーキテクチャチームによって、ゆるやかな調整が行われた。我々はコアドメインに対してそれなりに深いモデルを開発したし、生き生きとしたユビキタス言語も存在していた。

しかし、この企業文化はイテレーション開発に激しく抵抗していたので、実際に動作するソフトウェアを社内向けにリリースするまでに、我々はあまりにも長い間待たされた。

そのために、問題が露呈したのは開発の後半で、その頃には修正するリスクもコストも大きくなっていた。ある時点で、モデルの特定の側面がデータベースの性能問題を引き起こしていることが判明した。モデル駆動設計の本質は、実装上の問題からフィードバックを受けてモデルを変更することにあるのだが、その頃には、開発があまりにも進んでしまって、根本的なモデルは変更できないという認識があった。代わりに、より効率的にするような変更がコードに対して行われ、コードとモデルとのつながりが弱くなってしまった。初期リリースは、技術的なインフラストラクチャをスケールさせる際の限界も露呈していて、経営陣をぎくりとさせた。インフラストラクチャにあったこの問題を解決するために専門家が投入され、プロジェクトは持ち直したのだが、実装とドメインモデリング間のループが完結することはなかった。

いくつかのチームは、複雑な機能と表現力豊かなモデルを持った、優れたソフトウェアを完成させた。それ以外のチームが作ったのは、ユビキタス言語の痕跡は残っていても、モデルがデータ構造になってしまった、融通の利かないソフトウェアだった。コンテキストマップがあれば、何よりも役に立っただろう。さまざまなチームによる成果物の関係に、計画性がなかったからである。それでも、例でも取り上げたような、ユビキタス言語に取り込まれたコアモデルのおかげで、最終的には各チームの成果物をつなぎ合わせてシステムにすることはできた。

スコープが狭まったとはいえ、このプロジェクトによって、レガシーシステムがいくつか置き換えられた。共有された概念の集まりによって全体がまとまってはいるものの、設計のほとんどはあまりしなやかでなかった。数年経った現在では、構築したシステム自体がほとんどレガシーと化しているが、それでも国際的なビジネスで、1日24時間使われている。より大きな成功を収めたチームの影響は次第に広がったが、時間は結局なくなるもので、それは、資金のきわめて豊富な企業でも変わらないのだ。このプロジェクトの文化が、モデル駆動設計を本当の意味で自分のものにすることはなかった。現在行われている新しい開発は、プラットフォームが異なっており、我々の仕事の影響を間接的にしか受けていない。新しい開発者が、我々の作ったレガシーに順応しているからである。

場所によっては、この輸送会社が最初に設定したような意欲的な目標は、疑われるようになっている。それよりも、作り方のわかっている、小型のアプリケーションを作成する方がましらしい。設計の最大公約数にこだわって、単純な処理を行った方がよいということか。この保守的なアプローチにもそれなりの意義はあって、ほどよいスコープで、レスポンスの速いプロジェクトを可能にする。しかし、統合されたモデル駆動のシステムが約束する価値は、そのようなつぎはぎでは実現できない。そのために、第3の方法があるのだ。ドメイン駆動設計に従えば、豊富な機能を持つ大きなシステムを、段階的に成長させることができるようになる。これは、深いモデルとしなやかな設計を構築することで実現

される。

プロジェクトのリストは、Evant社で締めくくろう。これは在庫管理ソフトウェアを開発している企業だ。ここで私は二次的サポートの役割を果たし、すでに強力だった設計文化にいくらか貢献した。他の人たちはこのプロジェクトについて、エクストリームプログラミングの申し子と書いているが、指摘されることがほとんどないのは、このプロジェクトが極度にドメイン駆動だったということである。このプロジェクトでは、かつてないほど深いモデルが蒸留され、この上なくしなやかな設計で表現されていた。このプロジェクトは、2001年に「ドットコム・クラッシュ」が起きるまでは、うまくいっていたのだ。ドットコム・クラッシュ以降、投資資金が不足した同社は規模を縮小し、ソフトウェア開発はほとんど休止状態になって、終わりも近いように思われたのだった。ところが、2002年の夏、Evant社は、世界でも上位10社に入る小売業者からビジネスを持ちかけられた。この潜在顧客は製品を気に入っていたものの、設計を変更して膨大な在庫計画業務に対応できるように、アプリケーションをスケールアップする必要があった。それはEvant社にとって、最後のチャンスだったのである。

開発者は4人に減っていたが、このチームには資産があった。彼らにはスキルと、ドメインについての知識があり、メンバの1人は、スケールアップに伴う問題についての専門知識も持っていた。また、非常に効果的な開発文化もあった。しかも、彼らの持っていたコードベースは、しなやかに設計され、変更を容易にするものだった。その夏、4人の開発者は英雄的な努力で開発を行い、膨大な数の計画要素と、何百というユーザを処理する能力を作り出した。この能力に支えられて、Evant社は巨大なクライアントを獲得し、その直後には、別の企業に買収された。買収した企業は、Evant社のソフトウェアと、折り紙つきの能力を活用して、新たな需要に対応したいと考えたのだ。

ドメイン駆動設計の文化は（エクストリームプログラミングの文化と共に）過渡期を乗り越えて、息を吹き返した。モデルと設計は進化を続け、私が参加してから2年が経った現在、はるかに豊かでしなやかになっている。しかも、買収した企業に吸収されることなく、Evantチームのメンバは既存のプロジェクトチームに影響を与え、自分たちの後に従わせているようだ。この話は、まだ終わっていないのである。

本書にあるテクニックをすべて採用するプロジェクトなどないだろう。それでも、ドメイン駆動設計を真剣に行っているプロジェクトはすべて、何らかのかたちでそれとわかるようになる。決定的な特徴は、対象となるドメインを理解し、その理解をソフトウェアにおいて具現化することを、重視するところにある。すべては、その前提から流れ出すのだ。チームメンバはプロジェクトでの言語の使い方を意識し、精緻にするよう努める。チームメンバがドメインモデルの品質に満足することはなかなかないかもしれないが、それはドメインについて学び続けているからだ。彼らにとって、改良し続けることこそがチャンス

なのであり、整合していないモデルはリスクなのだ。設計スキルがきわめて重要だと考えられるのは、製品になり得る品質を持ち、ドメインモデルを明確に反映しているソフトウェアを開発するのが容易ではないからだ。彼らも障害物につまずくことはある。しかし、自分たちの原則を守りながら、立ち上がって前へと進んでいくのだ。

展　望

気象や生態系および生物学は、これまで混沌としたものと考えられており、物理学や化学とは対照的な、「ソフト」な分野とされていた。しかし、最近になって認識されだしたのは、この「混沌」とした外観が、実は、非常に複雑な現象の中にある秩序を発見して理解するという、深遠な技術的難題を提起しているということだ。「複雑性」と呼ばれるこの分野は、多くの科学の先駆けになっている。純粋に技術的な課題は、通常、才能のあるソフトウェアエンジニアにとって最も興味深くやりがいがあるように見えるものだが、ドメイン駆動設計によって開かれる新しい挑戦の領域は、少なくともそれに匹敵する。ビジネスソフトウェアを作るのに、混沌としたものをボルトでつなぎ合わせるようにする必要はない。複雑なドメインと格闘して、わかりやすいソフトウェア設計にすることは、優秀な技術者にとって刺激的な挑戦なのだ。

実際に動く複雑なソフトウェアを、素人が作成できる時代にはなっていない。初歩的なスキルを持ったプログラマを大勢集めても、ある種のソフトウェアを作り出すことはできるが、そうやって作れるのは、土壇場にある企業を救い出せるような類のものではない。ツールの作成者に求められるのは、優秀なソフトウェア開発者の能力と生産性を高めるような仕事に注意を向けることである。必要とされているのは、ドメインモデルを探究し、それを実際に動くソフトウェアで表現するための、より先鋭的な方法なのだ。この目的で考案される新しいツールと技術を用いて実験するのを、私は楽しみにしている。

しかし、改良されたツールが役立つものであっても、そちらに気を取られて、優れたソフトウェアを作成することが学び考える活動であるという、中核的な事実を見失ってはならない。モデリングには想像力と自己規律が必要である。思考を助けたり、注意がそれるのを防いだりするツールが望ましい。思考することで生み出さなければならないものを自動化しようと骨を折るのは、考えが甘く、逆効果になる。

すでにあるツールと技術を用いれば、今日あるプロジェクトのほとんどが作り出しているものよりも、はるかに価値のあるシステムを構築できる。我々は、使用するのも作業するのも楽しいソフトウェアを書けるのであり、さらに、成長するにつれて我々の進路をふさぐことなく、新しい機会を生み出し、所有者にとっての価値をつけ加え続けるソフトウェアを書けるのだ。

付録

本書におけるパターンの使い方

私にとって最初の「愛車」は、大学を卒業してすぐにもらった、8年目のプジョーだった。「フランスのメルセデス」と呼ばれることもあるこの車は、ていねいに造られていて、運転するのも快適であり、非常に信頼できた。しかし、私が手にした時には、調子が悪くなり始めていて、さらなるメンテナンスが必要な時期にさしかかっていた。

プジョー社は歴史のある企業で、何十年にもわたって独自の進化を遂げてきた。機械に関する独自の用語法を持ち、そのデザインは特異で、機能を部品に分割する方法さえも標準的でないことがある。結果としてできる車は、プジョーの専門家でしか対応することができず、大学院生の収入からすれば、潜在的な問題だった。

典型的だったのは、私が地元の整備士に燃料漏れを調べてもらいに行った時のことである。彼は車台を調べてこう言った。オイルは「後方3分の2くらいの所にある小さいボックスから漏れている。そのボックスは、おそらく前と後ろに、ブレーキの力を振り分けるのに関係していると思う」。そして彼は、車に触れることを拒み、私に80キロも離れた正規のディーラへ行くように言った。フォードやホンダなら誰でも修理することができる。だからこそ、そうした車の方が便利で、所有するには安上がりなのだ。だが、機械的に複雑であることには変わりない。

確かにプジョーは大好きだったが、二度と奇抜な車は持つまいと思う。ついにある日、とりわけ費用のかかる問題があると診断され、プジョーはもうたくさんという気になった。地元のチャリティへ持って行き、寄付として引き取ってもらった。そして、使い古されたホンダシビックを買ったのだが、その値段は、プジョーの修理にかかったであろう値段とほぼ同額だった。

ドメインの開発に標準的な設計要素が欠けているので、ドメインモデルや対応する実装は、どれも奇抜で理解しにくい。しかも、どのチームも車輪（もしくは、ギアやワイパー）

を再発明しなければならないのだ。オブジェクト指向設計の世界では、すべてがオブジェクトや参照、あるいはメッセージであり、これらは言うまでもなく便利な抽象である。しかし、ドメイン設計における選択肢の範囲を十分に制約することもなければ、ドメインモデルに関する効率的な議論を支えることもない。

「すべてはオブジェクトである」と言って止めてしまうのは、大工や建築家が家のことを要約して、「すべては部屋である」と言うようなものだ。高圧コンセントと流しのある大きな部屋があって、そこでは料理をするだろう。2階には小部屋があって、そこで眠るかもしれない。普通の家を記述するにも数ページが必要になる。家を建てたり利用したりする人々は、部屋がパターンに従っていて、そのパターンには「台所」などの特別な名前があることに気がつく。この言語によって、家の設計に関する効率的な議論ができるようになるのだ。

さらに、機能の組み合わせが、すべて実用的になるとは限らない。入浴と睡眠を同じ部屋でしたらどうか？ 便利ではないだろうか。しかし、長年の経験が慣習となって、「寝室」と「浴室」は別になっている。結局、入浴設備は寝室よりも多くの人々に共有される傾向があり、しかも、寝室を共有する他の人からさえも切り離された、最大限のプライバシーが求められる。さらに浴室には、特殊で高価なインフラストラクチャ要件がある。浴槽とトイレは同じ部屋にされるのが一般的だが、それはどちらも同じインフラストラクチャ（水と排水設備）を必要とし、どちらもプライベートで使用されるからだ。

インフラストラクチャに対する特殊な要件がある部屋の別の例としては、「台所」という食事を用意する部屋がある。浴室とは対照的に、台所には特別なプライバシー要件がない。費用がかかるため、比較的大きな家にも1つしかないのが普通である。1つであることにより、一緒に調理して食事をする習慣も促進される。

「寝室3部屋と浴室2つ、さらに間取りが自由な台所のある家が欲しい」と言えば、膨大な量の情報を簡潔な文に詰め込んだ上に、冷蔵庫の隣にトイレを配置するといった、多くのばかげた間違いを回避できることになる。

家、車、ボート、ソフトウェアといった、あらゆる領域の設計において、我々は、機能することがすでにわかっているパターンを基にして、確立されたテーマの範囲でアドリブを加えながら構築するのだ。時には、全く新しいものを考え出さなければならないこともある。しかし、標準的な要素をパターンを基に作ることで、既知の解決策が存在する問題に対して労力を浪費してしまうのを避け、特殊な要求に集中できるようになる。また、伝統的なパターンを基に構築すれば、設計があまりにも特異でコミュニケーションしにくいものになってしまうこともなくなる。

ソフトウェアのドメイン設計は、他の設計分野ほど成熟していないし、どのみち、多様化しすぎていて、自動車部品や部屋に用いるほど具体的なパターンには適応できない。だ

が、それでも「すべてがオブジェクトである」で終わらせずに、少なくともネジとバネとを区別する程度には、先に進む必要がある。

設計における洞察を共有し、標準化する形式は、Christopher Alexander率いる建築家グループによって1970年代に紹介された（Alexander他 1977）。彼らの「パターンランゲージ」は、よくある問題に対する、実証済みの設計上の解決策をまとめ上げた（これは、私が用いた「台所」の例よりはるかに巧妙なものだった。Alexanderの本の読者は、台所の例にうんざりしただろう）。そこで意図されていたのは、建築家とユーザがこの言語でコミュニケーションをとることであり、パターンの導きに従い利用者にとって機能的で快適な素晴しい建物を作り出すことである。

建築家がこの考え方についてどう思っていたかはともかく、パターンランゲージはソフトウェア設計に大きな影響を与えた。1990年代、ソフトウェアパターンはいろいろな方法で適用され、いくらかの成功を収めたのだが、そのなかで注目すべきなのは、詳細な設計（Gamma他 1995）と技術的アーキテクチャ（Buschmann他 1996）である。最近では、基本的なオブジェクト指向設計テクニック（Larman 1998）とエンタープライズアーキテクチャ（Fowler 2002、Alur他 2001）をドキュメント化するのにパターンが使用されている。パターンの言語は、今やソフトウェア設計に関する考えをまとめる上で、主流のテクニックとなっている。

パターン名は、チームで用いられる言語において、用語となることが意図されていて、本書でもそのように使用した。議論にパターン名が出てくる時は、注意を喚起するために(細）ゴシック体にしてある。

本書でのパターンのフォーマットは次の通りである。この基本方針に若干バリエーションがあるのは、厳格な構造よりも個別の明確さと読みやすさを、私が優先したからだ…。

パターン名

[概念の説明。視覚的なメタファーや
示唆に富むテキストのこともある。]

[コンテキスト。その概念が他のパターンとどう関係するかについての簡単な説明。場合によっては、パターンの概要。

しかし、本書において、コンテキストに関する議論の多くは、パターン内部よりも章の導入部や説明を行っている場所にある。

＊　＊　＊　]

[問題の議論]

問題の要約

問題の解明に関する議論から、必然的に解決策に焦点が当てられる。

それゆえ：

解決策の要約

結果。実装上の考慮事項。具体例。

＊　＊　＊

結果として生じるコンテキスト：このパターンが後のパターンにどうつながるかについての簡潔な説明。

[実装上の課題に関する議論。Alexanderによるオリジナルフォーマットの場合、この議論は問題を解明する節に入れられているので、本書でもAlexanderの構成に従うことがしばしばある。しかし、パターンによっては、実装に関する非常に長い説明が必要なものもある。コアとなるパターンの議論を簡潔に保つため、そういった実装上の長い議論はパターンの後へ移した。

また、長い例、とりわけ複数のパターンを組み合わせる例も、パターンの外に出していることが多い。]

用語解説

次に示すのは、本書で使用している中から精選した用語、パターン名、その他の概念の簡単な定義である。

値オブジェクト（VALUE OBJECTS）　ある特徴や属性を記述するが、同一性の概念を持たないオブジェクト。

アナリシスパターン（analysis pattern）　ビジネスモデリングにおける、共通の構造を表す概念グループ。ただ1つのドメインに関係することもあれば、多くのドメインにまたがることもある（Fowler 1997, p.8）。

暗黙的な概念（implicit concept）　モデルや設計の意味を理解する上で必要だが、明示的に言及されていない概念。

イテレーション（iteration）　プログラムが小さいステップで繰り返し改善されるプロセス。または、そのステップの1つ。

意図の明白なインタフェース（INTENTION-REVEALING INTERFACES）　クラス、メソッド、およびその他の要素の名称が、もともとの開発者が作成した目的と、クライアント開発者に対して持つ価値の両方を伝えている設計。

エンティティ（ENTITY）　本質的に、属性によってではなく、連続性と同一性の連なりによって定義されるオブジェクト。

概念の輪郭（CONCEPTUAL CONTOURS）　ドメイン自体の根底にある一貫性。モデルに反映されれば、設計がさらに自然に変更に対応できるようになる。

完結した値（WHOLE VALUE）　単一の完結した概念をモデル化したオブジェクト。

関数（function）　目に見える副作用なしに、処理を行って結果を戻す操作。

境界づけられたコンテキスト（BOUNDED CONTEXT）　特定モデルを適用できる限定された範囲。コンテキストの境界を定めることで、チームメンバは何を一致させるべきで、何を独立して開発できるのかについての理解を明確化し、共有できる。

凝集（cohesion）　論理的な合意と依存関係。

クライアント（client）　設計中の要素を呼び出し、その能力を使用するプログラム要素。

コアドメイン（CORE DOMAIN） モデルの特徴を示す部分。ユーザの目標の中心となり、アプリケーションを差別化して価値あるものにする。

コマンド（またはモディファイア）（command / modifier） システムに何らかの変化をもたらす操作（例えば変数の設定など）。意図的に副作用を作り出す操作。

コンテキスト（context） 単語や文が出てくる環境で、その単語や文の意味を決定するもの。境界づけられたコンテキストを参照。

コンテキストマップ（CONTEXT MAP） プロジェクトに含まれる複数の境界づけられたコンテキストや、コンテキストとモデルとの実際の関係を表現したもの。

サービス（SERVICES） インタフェースとして提供される操作。このインタフェースはモデル内で独立していて、カプセル化された状態を持たない。

しなやかな設計（supple design） 深いモデルに本来備わっている力をクライアント開発者に渡し、明確で柔軟な表現をすることで、期待される結果を確固たるものにする設計。また、しなやかな設計は、同じ深いモデルを利用することで、新しい洞察に設計を適合させるため、実装者が設計自体を作り直して新しいかたちにすることを容易にする。

集約（AGGREGATES） 関連したオブジェクトの集合で、データを変更する目的で1つの単位として扱われるもの。外部から参照できるのは、集約で**ルート**に指定されたメンバ1つに限られる。一貫性に関するルールの集合が、集約の境界内に適用される。

状態を持たない（ステートレス）（stateless） 設計要素が持つ性質で、その要素にある任意の操作を、要素の履歴を気にすることなく、クライアントが使用できるようにするもの。状態を持たない要素は、グローバルにアクセスできる情報を使用するかもしれず、そのグローバルな情報を変更する可能性さえある（つまり、副作用があるかもしれない）が、自分のふるまいに影響を与えるプライベートな状態は持たない。

蒸留（distillation） 混ざり合ったコンポーネントを分離し、より価値があって役に立つかたちで本質を抽出するためのプロセス。ソフトウェア設計では、モデルの主要な側面を抽象化すること、または巨大なシステムを分割してコアドメインを前面に持ってくることを意味する。

責務（responsibility） タスクを実行したり、情報について知っておいたりする義務（Wirfs-Brock他 2003, p.3）。

宣言的設計（declarative design） プログラミングの1形式で、属性を正確に記述することで、実際にソフトウェアを制御するもの。実行可能な仕様。

戦略的設計（strategic design） システムの大部分に適用される、モデリングや設計上の意思決定。こうした意思決定はプロジェクト全体に影響するので、チームレベルで決定しなければならない。

大規模な構造（large-scale structure） システム全体に関する設計のパターンを規定する、

高次の概念やルールの集合、もしくはその両方。システムを大まかな流れで議論し、理解できるようにするための言語。

デザインパターン（design pattern） 一般的な設計上の問題を特定のコンテキストにおいて解決するためにカスタマイズされた、オブジェクトとクラスのコミュニケーションに関する記述。（Gamma他 1995, p.3）

統一（unification） モデルの内部的な一貫性。各用語が明快で矛盾するルールがないようにするもの。

独立したクラス（STANDALONE CLASSES） システムプリミティブと基本ライブラリを除き、他のクラスを参照することなく理解し、テストできるクラス。

ドメイン（domain） 知識、影響、または活動の領域。

ドメインエキスパート（domain expert） ソフトウェアプロジェクトのメンバで、ソフトウェア開発ではなく、アプリケーションのドメインの担当者。ソフトウェアの単なるユーザと違い、ドメインエキスパートはその対象に関する深い知識を持っている。

ドメイン層（domain layer） レイヤ化アーキテクチャにおいて、ドメインロジックの責務を負う、設計および実装の一部分。ドメイン層は、ドメインモデルのソフトウェアにおける表現が息づく場所である。

表明（ASSERTIONS） ある時点におけるプログラムの正しい状態を、どのようにそうなるかとは独立して表した文。表明は、操作の結果や設計要素の不変条件を定義するのが通例である。

モデリングパラダイム（modeling paradigm） ドメインにおける諸概念を切り取る特定のスタイル。ツールと組み合わされて、それらの概念に類似したソフトウェアを作成する（例えば、オブジェクト指向プログラミングや論理プログラミング）。

モデル（model） ドメインにおける選択された側面を記述し、そのドメインに関連した問題を解決するのに使用できる抽象の体系。

モデル駆動設計（MODEL-DRIVEN DESIGN） ソフトウェア要素のサブセットがモデル要素と密接に対応している設計。また、相互に一致した状態を保ちながら、モデルと実装を共に開発するプロセス。

ファクトリ（FACTORY） クライアントのために複雑な生成ロジックをカプセル化し、生成されたオブジェクトの型を抽象化するための仕組み。

深いモデル（deep model） ドメインエキスパートの主要な関心事と、そこに最も深く関連した知識を鋭く表現したもの。深いモデルは、ドメインの表層的な側面と素朴な解釈から脱却するものである。

副作用（side effect） 操作に起因する目に見える状態の変化。故意かどうかに関わらず、意図的な更新であってもこう呼ばれる。

副作用のない関数（SIDE-EFFECT-FREE FUNCTIONS） **関数**を参照。

不変（immutable） 生成された後、目に見える状態が変化しないという性質。

不変条件（invariant） 何らかの設計要素についての表明で、常に真でなければならないもの。ただし、メソッドの実行中やまだコミットされていないデータベーストランザクション中といった特殊な過渡的状況を除く。

ライフサイクル（life cycle） オブジェクトが生成されてから削除されるまでの間に生じる一連の状態。通常は、状態の遷移に伴う整合性を保証する制約がある。システム間や、別々の境界づけられたコンテキスト間における、エンティティの移動を伴うこともある。

リポジトリ（REPOSITORIES） 格納、取り出し、および検索のふるまいをカプセル化し、オブジェクトのコレクションをエミュレートする仕組み。

レイヤ化アーキテクチャ（LAYERED ARCHITECTURE） ソフトウェアシステムの関心事を切り離し、ドメイン層を他から分離するテクニック。

ユビキタス言語（UBIQUITOUS LANGUAGE） ドメインモデルを取り巻いて構築され、チームのあらゆる活動をソフトウェアと結びつけるために、チームメンバ全員によって使用される言語。

参考文献

Alexander, C., M. Silverstein, S. Angel, S. Ishikawa, and D. Abrams. 1975. *The Oregon Experiment.* Oxford University Press. [邦訳：宮本雅明訳『オレゴン大学の実験』鹿島出版会]

Alexander, C., S. Ishikawa, and M. Silverstein. 1977. *A Pattern Language: Towns, Buildings, Construction.* Oxford University Press. [邦訳：平田翰那訳『パタン・ランゲージ：環境設計の手引』鹿島出版会]

Alur, D., J. Crupi, and D. Malks. 2001. *Core J2EE Patterns.* Sun Microsystems Press. [邦訳：近棟稔・吉田悦万・小森美智子監訳、トップスタジオ訳『J2EEパターン　第2版』日経BP社]

Beck, K. 1997. *Smalltalk Best Practice Patterns.* Prentice Hall PTR. [邦訳：梅沢真史・皆川　誠・小黒直樹・森島みどり『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』ピアソンエデュケーション（現、ピアソン桐原）]

—. 2000. *Extreme Programming Explained: Embrace Change.* Addison-Wesley. [邦訳：長瀬嘉秀監訳、（株）テクノロジックアート訳『XPエクストリーム・プログラミング入門—変化を受け入れる』ピアソンエデュケーション（現、ピアソン桐原）]

—. 2003. *Test-Driven Development: By Example.* Addison-Wesley. [邦訳：長瀬嘉秀監訳、（株）テクノロジックアート訳『テスト駆動開発入門』ピアソンエデュケーション（現、ピアソン桐原）]

Buschmann, F., R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. 1996. *Pattern-Oriented Software Architecture: A System of Patterns.*Wiley. [邦訳：金澤典子・水野貴之・桜井麻理・関富登志・千葉寛之訳『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』近代科学社]

Cockburn, A. 1998. *Surviving Object-Oriented Projects: A Manager's Guide.* Addison-Wesley. [邦訳：長瀬嘉秀・今野　睦監訳、（株）テクノロジックアート訳『アジャイルプロジェクト管理』ピアソンエデュケーション（現、ピアソン桐原）]

Evans, E., and M. Fowler. 1997. "Specifications." Proceedings of PLoP 97 Conference.

Fayad, M., and R. Johnson. 2000. *Domain-Specific Application Frameworks.* Wiley.

Fowler, M. 1997. Analysis Patterns: *Reusable Object Models.* Addison-Wesley. [邦訳：堀内　一監訳、児玉公信・友野晶夫訳『アナリシスパターン—再利用可能なオブジェクトモデル』ピアソンエデュケーション（現、ピアソン桐原）]

—. 1999. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley. [邦訳：児玉公信・友野晶夫・平澤　章・梅澤真史共訳『リファクタリング—プログラムの体質改善テクニック』ピアソ

ンエデュケーション（現、ピアソン桐原）]

—. 2002. *Patterns of Enterprise Application Architecture.* Addison-Wesley. [邦訳：長瀬嘉秀監訳、(株) テクノロジックアート訳『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』翔泳社]

Gamma, E., R. Helm, R. Johnson, and J. Vlissides. 1995. *Design Patterns.* Addison-Wesley. [邦訳：本位田真一・吉田和樹訳『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　改訂版』ソフトバンククリエイティブ]

Kerievsky, J. 2003. "Continuous Learning," in *Extreme Programming Perspectives,* Michele Marchesi et al. Addison-Wesley.

—. 2003. Web site: http://www.industriallogic.com/xp/refactoring.

Larman, C. 1998. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design.* Prentice Hall PTR. [邦訳：今野　睦・依田智夫監訳、依田光江訳『実践UML　第3版—オブジェクト指向分析設計と反復型開発入門』ピアソンエデュケーション（現、ピアソン桐原）]

Merriam-Webster. 1993. *Merriam-Webster's Collegiate Dictionary.* Tenth edition. Merriam-Webster.

Meyer, B. 1988. *Object-oriented Software Construction.* Prentice Hall PTR. [邦訳：酒匂　寛訳『オブジェクト指向入門第2版　原則・コンセプト、方法論・実践』翔泳社]

Murray-Rust, P., H. Rzepa, and C. Leach. 1995. *Abstract 40.* Presented as a poster at the 210th ACS Meeting in Chicago on August 21, 1995. http://www.ch.ic.ac.uk/cml/

Pinker, S. 1994. *The Language Instinct: How the Mind Creates Language.* HarperCollins. [邦訳：椋田直子訳『言語を生みだす本能　上巻・下巻』日本放送出版協会]

Succi, G. J., D. Wells, M. Marchesi, and L. Williams. 2002. *Extreme Programming Perspectives.* Pearson Education.

Warmer, J., and A. Kleppe. 1999. *The Object Constraint Language: Precise Modeling with UML.* Addison-Wesley.

Wirfs-Brock, R., B. Wilkerson, and L. Wiener. 1990. *Designing Object-Oriented Software.* Prentice Hall PTR.

Wirfs-Brock, R., and A. McKean. 2003. *Object Design: Roles, Responsibilities, and Collaborations.* Addison-Wesley. [邦訳：(株) オージス総研　藤井　拓監訳、辻　博靖・井藤晶子・山口雅之・林直樹共訳『オブジェクトデザイン』翔泳社]

掲載写真クレジット

本書の写真はすべて、許可を得て掲載している。

Richard A. Paselk, Humboldt State University
Astrolabe（第3章、45ページ）

© Royalty-Free/Corbis
Fingerprint（第5章、85ページ）、Service Station（第5章、105ページ）、Auto Factory（第6章、134ページ）、Librarian（第6章、146ページ）

Martine Jousset
Grapes（第6章、123ページ）、Olive Trees（young and old）（結論、506、507ページ）

Biophoto Associates/Photo Researchers, Inc.
Electron micrograph of Oscillatoria（第14章、343ページ）

Ross J. Venables
Rowers（group and single）（第14章、349、380ページ）

Photodisc Green/Getty Images
Runners（第14章、365ページ）、Child（第14章、370ページ）

U.S. National Oceanic and Atmospheric Administration
Great Wall of China（第14章、373ページ）

© 2003 NAMES Project Foundation, Atlanta, Georgia.Photographer Paul Margolies.
www.aidsquilt.org
AIDS Quilt（第16章、445ページ）

索　引

英　字

AIDSメモリアルキルトプロジェクト 483
Cleese, John 4
Evant社 510
Javaエンティティ Bean対エンティティ 89
MVC（モデル・ビュー・コントローラ） 70
PCB設計の話 7, 506
UML（統一モデリング言語） 35

あ　行

アーキテクチャ主導のモジュールにかかるコスト 112
アーキテクチャフレームワーク .68, 72, 156, 276, 503
アウトソーシング 416
アジャイルな設計
　——依存関係を減らす 271, 441, 470
　——しなやかな設計 247, 266
　——蒸留 489
　——モジュール 110
アストロラーベ 45
値オブジェクト→エンティティ；サービスも参照
　——エンティティを参照する 96
　——オブジェクトの組み合わせ 96
　——概要 95
　——関連 101
　——共有する 98
　——クラスタリング→集約を参照
　——設計する 97
　——双方向の 関連 101
　——データベースをチューニングする、例 101
　——パラメタとして渡す 97
　——不変性 98
　——変更管理 99
　——例 166
アダプタ 376
アナリシスパターン→デザインパターンも参照
　——概念的整合性 312
　——概要 300
　——定義 299
　——ユビキタス言語 312
　——例 301
アプリケーション層 68, 74
暗黙的な概念
　——カテゴリ 221
　——認識する 207
一時的なオブジェクト 148
イテレーティブな設計プロセス 14, 190, 451
意図の明白なインタフェース 251, 429
インターネットエクスプローラのお気に入りの話 55
インフラストラクチャ駆動パッケージング 111
インフラストラクチャ層 68
ウェブサイトのお気に入りの話 55
ウォーターフォール設計手法 14
映画編集の話 5
永続化オブジェクト 149
エンティティ→関連；サービス；値オブジェクトも参照
　——IDの一意性 93
　——値オブジェクトを用いて参照する 96
　——クラスタリング→集約を参照
　——自動生成されるID 93
　——属性を識別する 92
　——対Javaエンティティ Bean 89

——同一性を確立する 88
——同一性を追跡する 92
——モデル化 91
——例 166
オーバーブッキング、例 19, 224
お気に入りの話 55
オブザーバ 70
オブジェクト→エンティティ；値オブジェクトも参照
——永続化 149
——関係データベースのために設計する 159
——関連 80
——生成する 237
→コンストラクタ；ファクトリも参照
——定義する 79
——複数のオブジェクトでできた→集約；コンポジットを参照
オブジェクトではないもののモデル 117
オブジェクトの参照→リポジトリを参照
オブジェクトを検証する 229, 230
オブジェクトを選択する 231, 274
オンラインバンキング、例 69

か 行

概念的レイヤ→レイヤ化アーキテクチャ；責務のレイヤを参照
概念の分析→アナリシスパターン、例：概念の分析も参照
——ぎこちなさ 212
概念の輪郭 266
開発チーム→チームを参照
外部システム 392
→統合も参照
化学製品倉庫内格納サービス、例 238
化学の例 386
隠された概念を引き出す 18
→暗黙的な概念も参照
隔離されたコア 433
隔離されたドメイン層 105
欠けている概念 209
カプセル化→ファクトリも参照
——意図の明白なインタフェース 251
——凝集されたメカニズム 429
——リポジトリ 154
貨物輸送の例→例：貨物輸送を参照
関数、副作用のない 255, 290
関連
——値オブジェクト 101
——実践的な設計のための 80
——双方向の 101
——例 168
ぎこちなさ、概念の分析 212
既製品による解決 415
偽同族語 347
境界づけられたコンテキスト→コンテキストマップも参照
——概要 343
——関係性 361
——偽同族語 347
——境界をテストする 360
——継続的な統合 349
——コードの再利用 352
——対モジュール 343
——大規模な構造 491
——重複した概念 347
——定義する 391
——分派 347
——変換層 383
→腐敗防止層、公表された言語も参照
——例 345
境界をテストする 360
境界を変換する 392
凝集、モジュール 102, 108
凝集されたメカニズム
——概要 429
——と宣言的スタイル 432
——対 汎用サブドメイン 431
——例 432
強調されたコア 424
共通言語→公表された言語、ユビキタス言語を参照
共有カーネル
——概要 363
——継続的な統合とのマージ 400
——別々の道とのマージ 398

——例 368
キルトプロジェクト 486
クライアントとの疎結合 155
継続的学習 16
継続的な統合 349, 400
→統合も参照
コアドメイン
——概要 408
——主要な要素にフラグを立てる 425
——ドメインビジョン声明文 422
——メカニズム 431
語彙→公表された言語；ユビキタス言語を参照
公開ホストサービス、公表された言語に変換する 402
公表された言語
——概要 384
——公開ホストサービスとマージする 402
——象と盲人 387
——例 386
コードの再利用
——境界づけられたコンテキスト 352
——先達の技を再利用する 331
——汎用サブドメイン 420
コールバック 70
顧客／供給者 365
顧客に焦点を合わせたチーム 499
コミュニケーション、書き言葉
→ドキュメント、UML、ユビキタス言語を参照
コミュニケーション、話し言葉→ユビキタス言語を参照
コンストラクタ 139, 174
→ファクトリも参照
コンテキストの原則 337
→境界づけられたコンテキスト；コンテキストマップも参照
コンテキストマップ→境界づけられたコンテキストも参照
——概要 352
——構成してドキュメント化する 360
——対大規模な構造 451, 494
——例 355
コンテキストマップ、戦略を選択する
——外部システム 392
——境界づけられたコンテキストを定義する .391
——境界を変換する 392
——共有カーネル 363
——共有カーネルと継続的な統合をマージする 400
——公開ホストサービスと公表された言語をマージする 402
——顧客／供給者の開発チーム 365
——順応者 393
——進行中のプロジェクトのために 397
——設計中のシステム 394
——チームのコンテキスト 391
——デプロイ 396
——統合 393
——特殊な用語法 394
——トレードオフ 396
——パッケージング 396
——腐敗防止層 393
——別々の道 393
——別々の道と共有カーネルをマージする 398
——変換 398
——レガシーシステムを段階的に廃止する 401
コンポジット仕様 281
コンポジットパターン 321

さ 行

サービス→エンティティ；値オブジェクトも参照
——アクセス 107
——概要 103
——と隔離されたドメイン層 105
——特徴 104
——命名 104
——粒度 107
——レイヤに分割する 106
再構成 143, 147
在庫管理、例 510
参照オブジェクト→エンティティを参照
自社での解決、汎用サブドメイン 417
システムメタファ 453
しなやかな設計
——アプローチ 287
——意図の明白なインタフェース 251
——概念の輪郭 266

——概要 247
——コンポジット仕様 281
——設計の宣言的スタイル 278
——宣言的設計 275
——相互依存関係 271
——大規模な構造 484
——独立したクラス 271
——ドメイン特化言語 277
——表明 261
——副作用のない関数 255, 290
——命名規約 252
——例 252
——閉じた操作 273
シナリオ、例 173
柔軟性→しなやかな設計を参照
集約
——概要 123
——所有関係 124
——定義 124
——不変条件 126
——例 128, 170, 177
——ローカルな一意性とグローバルな一意性 ..126
順応者 370, 393
仕様→アナリシスパターン；デザインパターンを参照
——オブジェクトを検証する 229, 230
——オブジェクトを生成する 237
——オブジェクトを選択する 231
——概要 226
——組み合わせる→コンポジット仕様を参照
——構成する 228
——コンポジット 281
——実装する 229
——定義 228
——適用する 229
——ビジネスルール 227
——目的 229
——例 29, 239, 283
蒸留→例：蒸留も参照
——PCB設計の話 13
——意図の明白なインタフェース 429
——概要 405
——隔離されたコア 433
——カプセル化 429
——強調されたコア 424
——コアの概念を分離する 432
——設計における役割 337
——大規模な構造 489, 496
——多態性 442
——抽象化されたコア 441
——ドメインビジョン声明文 422
——深いモデル 443
——リファクタリングの対象 443
蒸留、凝集されたメカニズムと宣言的スタイル ..432
——概要 429
——対 汎用サブドメイン 431
蒸留、コアドメインドメインビジョン声明文 422
——概要 408
——主要な要素にフラグを立てる 425
——メカニズム 431
蒸留、汎用サブドメイン、公表されている設計を適用する 416
——アウトソーシング 416
——概要 414
——既製品による解決 415
——再利用性 420
——自社での実装 417
——対 凝集されたメカニズム 431
——リスク管理 420
蒸留ドキュメント 425, 427
進化する秩序 450, 499
図→ドキュメント；UMLを参照
ストラテジーパターン 19, 317
整合性→モデルの整合性を参照
責務のレイヤ
——概要 456
——役に立つ特徴 468
——例 458
——レイヤを選択する 468
設計中のシステム 394
設計の宣言的スタイル 278, 432
設計の変更→リファクタリングを参照
設計をシンプルにする→蒸留；大規模な構造；レイヤ化アーキテクチャを参照

設計を変更する→リファクタリングを参照
説明のためのモデル 40
宣言的設計 275
専門用語→公表された言語；ユビキタス言語を参照
戦略的設計→大規模な構造も参照
——オブジェクトの役割 502
——開発者の役割 502
——欠かせない要件 500
——技術的なフレームワーク 503
——現状を評価する 498
——顧客に焦点を合わせたアーキテクチャチーム 499
——進化 501
——進化する秩序 499
——戦略を策定する 498
——チーム作り 501
——チームのコミュニケーション 500
——フィードバックプロセス 500
——複数の開発チーム 499
——ミニマリズム 502
象と盲人 387
双方向の関連 101
素朴なメタファ 454

た 行

大規模な構造→蒸留；例：大規模な構造；レイヤ化アーキテクチャ；戦略的設計も参照
——開発の制約 451
——概要 445
——コンテキストマップ 451
——システムメタファ 453
——しなやかな設計 489
——柔軟性 486
——進化する秩序 450
——設計における役割 337
——素朴なメタファ 454
——チームのコミュニケーション 488
——知識レベル 472
——着脱可能コンポーネントのフレームワーク 482
——定義 448
——ミニマリズム 488
——リファクタリング 488
大規模な構造、責務のレイヤ
——概要 456
——役に立つ特徴 468
——レイヤを選択する 468
多態性 442
チーム
——開発者コミュニティの成熟度 115
——境界づけられたコンテキストを定義する ..391
——探究 330
——顧客に焦点を合わせた 499
——コミュニケーション、大規模な構造 488
——戦略を選択する 391
チーム、と戦略的設計
——開発者の役割 501
——顧客に焦点を合わせた 499
——コミュニケーション 500
——チーム作り 501
——複数のチーム 499
チーム、複数の
——共有カーネル 363, 368
——顧客／供給者 365
——順応者 370
——戦略的設計 499
——腐敗防止層 373
——例 367
チームのコンテキスト 391
知識のかみ砕き 14
知識のかみ砕き、例 13
知識レベル 472
着脱可能コンポーネントのフレームワーク 482
抽象化されたコア 441
重複した概念 347
データベースをチューニングする、例 101
デザインパターン→アナリシスパターンも参照
——概要 315
——コンポジット 321
——ストラテジー 317
——対 ドメインパターン 315
——フライウェイト 328
手続き型言語、とモデル駆動設計 49
デプロイ 396
→モジュールも参照

統一性 340
→継続的な統合も参照
同一性
——確立する 88
——追跡する 92
——ローカル対グローバル 126
統一モデリング言語（UML） 35
統合
——外部システム 393
——継続的な統合 349, 362
——公開ホストサービス 383
——象と盲人 387
——費用対効果の分析 380
——腐敗防止層 373
——別々の道 380
——変換層 383
→公表された言語も参照
——例 381
投資銀行、例 196, 213, 507
ドキュメント
——最新の状態を保つ 37
——蒸留ドキュメント 425, 427
——説明のためのモデル 39
——妥当性 37
——ドキュメントとしてのコード 39
——ドメインビジョン声明文 422
——プロジェクトの活動に取り込まれた 38
——目的 36
——ユビキタス言語 38
ドキュメントとしてのコード 39
独立したクラス 271
閉じた操作 273
ドメインエキスパート、要件を集める→概念の分析；知識のかみ砕きも参照
——ドメインエキスパートの言語 208
→ユビキタス言語も参照
ドメインエキスパートの言語 208
ドメインオブジェクト、ライフサイクル 121
→集約；ファクトリ；リポジトリも参照
ドメインオブジェクトとしてのプロセス 224
ドメインオブジェクトのライフサイクル 121
→集約；ファクトリ；リポジトリも参照
ドメイン層 68, 73
ドメイン特化言語 277
ドメインパターン対デザインパターン 315
ドメインビジョン声明文 422
ドメインを隔離する→腐敗防止層；蒸留；レイヤ化アーキテクチャを参照
トランザクションスクリプト 76
トランザクション制御 155
塗料混合アプリケーション、例 252, 257, 262

は 行

パターン 513
→アナリシスパターン；デザインパターン；大規模な構造も参照
パッケージング→デプロイ；モジュールを参照
発見 193
話し言葉、共通言語→ユビキタス言語を参照
話し言葉、モデリングする 30
パフォーマンスチューニング、例 186
汎用サブドメイン
——アウトソーシング 416
——概要 414
——既製品による解決 415
——公表されている設計を適用する 416
——再利用性 420
——自社での解決 417
——対凝集されたメカニズム 431
——リスク管理 420
——例 417
ビジネスルール 17, 277
ビジネスロジック、ユーザインタフェース層にある 75
ビジョン声明文→ドメインビジョン声明文を参照
人違いの話 87
表明 261
ファクトリ 143
ファクトリ、位置づける 137
ファクトリ、インタフェースを設計する 141
ファクトリ、エンティティ対値オブジェクト 142
ファクトリ、オブジェクトを生成する 135
ファクトリ、概要 134
ファクトリ、仕様を構成する 228

ファクトリ、生成する 137
ファクトリとリポジトリ 157
ファクトリ、不変条件ロジック 141
ファクトリ、要件 137
ファクトリ、例 174
ファクトリメソッド 136
ファサード 375
ファシリティ 196
深いモデル
——概要 20
——蒸留 443
——リファクタリング 191
複雑さ、減らす→蒸留；大規模な構造；レイヤ化アーキテクチャ；しなやかな設計を参照
副作用 255
→表明も参照
副作用のない関数 255, 290
複数のモデル 341, 343
腐敗防止層
——アダプタ 376
——インタフェース設計 375
——外部システムとの関係 393
——概要 373
——考慮事項 377
——ファサード 375
——例 378
不変条件ロジック 126, 141
不変性、値オブジェクトの 98
フライウェイトパターン 328
ブレイクスルー 195, 205
ブレインストーミング 13, 208, 219
プレゼンテーション層→ユーザインタフェース層を参照
プロトタイプ 242
分割する
——サービスを複数のレイヤに 106
——複雑なプログラム→大規模な構造；レイヤ化アーキテクチャを参照
分析モデル 45
別々の道 393, 398
変換する 398
変換層 383
ホーリー・グレイルの話 4
保険プロジェクト、例 381
ポリシーパターン→ストラテジーパターンを参照

ま 行

マージ
——共有カーネルから継続的な統合へ 400
——公開ホストサービスと公表された言語 402
——別々の道から共有カーネルへ 398
矛盾 218
——欠けている概念 209
——試行錯誤 221
——仕様 225
——存在するリソースを調査する 218
——ドメインエキスパートの言語 208
——ドメインオブジェクトとしてのプロセス 224
——明示的な制約 221
矛盾、概念の分析 218
明示的な制約, 概念の分析 221
命名
——意図の明白なインタフェース 252
——境界づけられたコンテキスト 353
——サービス 104
——しなやかな設計のための規約 252
——モジュール 109
メタデータマッピング層 148
盲人と象 387
モジュール
——アジャイル 110
——意味を決定する 109
——インフラストラクチャ駆動パッケージング 111
——オブジェクトではないもののモデル 117
——オブジェクトパラダイム 114
——概要 108
——凝集 108, 112
——結合 108
——コスト 112
——対 境界づけられたコンテキスト 343
——ドメインオブジェクトをパッケージングする 113
——パラダイムを混在させる 117
——命名 109

——モジュール性 113
——モデリングパラダイム 114
——リファクタリング 109, 110
——例 110, 179
モジュール性 113
モジュールの結合度 108
モデリング
——エンティティ 91
——オブジェクトではないもの 117
——関連 80
——実践的モデラ 57
——プログラミングによる統合 57
モデル
——実装と結びつける→モデル駆動設計を参照
——とユーザの理解 55
モデル・ビュー・コントローラ（MVC） 70
モデル駆動設計
——概要 47
——設計との一致 48
——ツールによるサポート 48
——手続き的言語 49
——モデリングパラダイム 48
——モデルの妥当性 47
モデル層→ドメイン層を参照
モデルに基づいた言語→ユビキタス言語を参照
モデルの整合性→境界づけられたコンテキスト；コンテキストマップ；複数のモデルも参照
——概要 339
——関係性を認識する 341
——境界を設定する 341
——統一性 340
→継続的な統合も参照
——複数のモデル 341
モンティ・パイソンの話 4

や 行

ユーザインタフェース層
——アプリケーション層とドメイン層から分離する 74
——定義 68
——ビジネスロジック 75
輸送、例→例：貨物輸送を参照
ユビキタス言語→公表された言語も参照
——アナリシスパターン 312
——一貫して使う 32
——概要 24
——貨物経路選択サービス、例 27
——関係データベースのためにオブジェクトを設計する 161
——専門的な用語法 395
——ドメインエキスパートの言語 208
——ドメイン特化言語 277
——話し言葉の役割 31
——モデルを改良する 30
——要件分析 25
要件、集める→概念の分析；知識のかみ砕き；ユビキタス言語を参照
用語法→境界づけられたコンテキスト；公表された言語；ユビキタス言語を参照

ら 行

利口なUI 74
リスク管理 420
利息計算サービス、例 213, 219, 301
リファクタリング
——開始 329
——開発者のために設計する 332
——危機の間 333
——探究チーム 330
——しなやかな設計 193
——蒸留 443
——先達の技を再利用する 331
——大規模な構造 488
——タイミング 332
——定義 190
——パターン指向 190
——発見 193
——深いモデル 191
——ブレイクスルー 196
——モジュール 109, 110
——例 177, 182, 196, 252
——レベル 190
リファクタリング、対象 443
リポジトリ

——アーキテクチャフレームワーク 156
——一時的なオブジェクト 148
——永続化されたオブジェクト 149
——オブジェクトへのアクセス 148
——概要 146
——型の抽象化 155
——カプセル化 154
——関係データベースのためにオブジェクトを設計する 159
——クライアントとの疎結合 155
——グローバルな検索 149
——実装する 154
——すでに存在するオブジェクトを参照する ..148
——問い合わせる 151
——とファクトリ 157
——トランザクション制御 155
——メタデータマッピング層 148
——利点 151
——例 171
粒度 106
例
——Evant社 510
——Javaにおけるパッケージのコード 110
——SEMATECH CIMフレームワーク 483
——値オブジェクト 101
——新しい機能を導入する 182
——アナリシスパターン 301
——意図の明白なインタフェース 430
——オンラインバンキング 69
——概念の輪郭 266
——化学、公表された言語 386
——化学製品倉庫内格納サービス 238
——隠された概念を引き出す 18
——給料と年金 473
——凝集されたメカニズム 432
——購入注文の整合性 128
——公表された言語 386
——コレクションから選択する 274
——コンストラクタ 174
——コンポジット仕様 283
——在庫管理 510
——しなやかな設計 252
——集 128
——仕様 239
——責務のレイヤ 458
——説明のためのモデル 40
——組織図 430
——タイムゾーン 418
——他システムとの統合 381
——知識レベル 473
——着脱可能コンポーネントのフレームワーク 482
——データベースをチューニングする 101
——手続き的言語 50
——投資銀行 213
——閉じた操作 274
——塗料混合アプリケーション 252, 257, 262
——表明 262
——副作用のない関数 257, 290
——複数のチーム 367
——ブレイクスルー 205
——プロトタイプ 242
——保険プロジェクト 381
——モジュール 110
——モデル駆動設計 50
——リファクタリング 252
——レイヤ化アーキテクチャ 69
レイヤ化アーキテクチャ→蒸留；例：レイヤ化アーキテクチャ；大規模な構造も参照
——MVC（モデル・ビュー・コントローラ）...70
——アプリケーション層 68, 74
——依存関係の設計 70
——インフラストラクチャ層 68
——概念的レイヤ 68
——隔離されたドメイン層 105
——価値 67
——コールバック 70
——サービス 71
——図 66
——ドメイン層 68, 73
——トランザクションスクリプト 76
——複雑なプログラムを分割する 68
——フレームワーク 72
——ユーザインタフェース、アプリケーション、ドメインを分離する 74

——ユーザインタフェース層 68, 74
——利口なUI 74
——レイヤを関係づける 70
レイヤ化アーキテクチャ、責務のレイヤ
——概要 456
——役に立つ特徴 468
——レイヤを選択する 468
レイヤ化アーキテクチャ、腐敗防止層
——アダプタ 376
——インタフェース設計 375
——外部システムとの関係 393
——概要 373
——考慮事項 377
——ファサード 375
レガシーシステム,段階的に廃止する 401
ローカルな同一性 対 グローバルな同一性 126
ローン管理、例→例：ローン管理を参照

例
例：概念の分析
——暗黙的な概念 291
——隠れた概念を引き出す 18
——欠けている概念を識別する 209
——ぎこちなさを解消する 212
——存在するリソースを調査する 219
例：貨物輸送
——値オブジェクト 166
——エンティティ 167
——概念の分析 224
——隔離されたコア 436
——隠れた概念を引き出す 18
——貨物の経路を選択する 27
——貨物を追跡する 40
——関連 168
——経路検索 318
——経路を自動選択する 355
——結論 508
——コンストラクタ 174
——コンテキストマップ 355
——コンポジット経路 323
——コンポジットパターン 323
——システムの概要 163
——シナリオ 173
——集約 170, 177
——ストラテジー 318
——責務のレイヤ 458
——大規模な構造 458
——ドメインを隔離する 166
——配分チェック 182
——パフォーマンスチューニング 186
——ファクトリ 174
——複数の開発チーム 368
——腐敗防止層 378
——モジュール 179
——輸送業務と経路 40
——ユビキタス言語 27
——予約
——オーバーブッキング 18, 224
——隠れた概念を引き出す 18
——境界づけられたコンテキスト 345
——対 収益分析 368
——レガシーアプリケーション 378
——リファクタリング 177
——リポジトリ 171
——欠けている概念を識別する 209
例：蒸留
——隔離されたコア 433
——凝集されたメカニズム 430, 432
——組織図 430, 432
——タイムゾーン 418
——汎用サブドメイン 417
例：大規模な構造 482
——責務のレイヤ 468
——知識レベル 473
例：統合
——象のモデルを統一する 387
——腐敗防止層 378
——変換サービス 355
例：レイヤ化アーキテクチャ
——アプリケーションを分割する 69
——責務のレイヤ 458
例：ローン管理
——アナリシスパターン 301
——概念の分析 213, 219

——概念の輪郭 268
——結論 507
——投資銀行 196
——ブレイクスルー 196
——利息計算サービス 213, 219, 301
——リファクタリング 196, 288

監訳者あとがき

この度、Domain-Driven Designの翻訳書籍をここに出版でき、関係者の皆様に感謝いたします。

そもそも、本書を私が最初に手にしたのは、プロダクトライン・エンジニアリングにおいて、ドメインに関する技術を獲得および補完するため、いくつかの海外文献を参考にしていたときでした。中規模から大規模レベルのシステム開発の現場において、機能追加や改修に伴うバージョンアップを繰り返していくと、システムは硬直し、いずれ過去の遺物となり、果てはリプレースの対象となってしまいます。IT予算が潤沢にある組織では問題にならないかもしれませんが、社運をかけたシステムが惨憺たる結果となり、社員の業務や顧客にぎこちない機能しか提供できない状況を多く見受けます。このような結果を生み出さないよう、中長期的にシステムを維持していくため、ドメイン分析をはじめとする上流の開発技術にフォーカスしてきたのです。

同時に、過去いくつものシステム開発を開発していく中で、ドメインに精通していくと、システムの構造を意識的に進化させることが可能であることを経験しました。アーキテクチャパターンやデザインパターンを適用し、システムの柔軟性と免震性の両立を目指すだけでなく、顧客への逆提案や経営の仕組みにもインパクトをもたせる視点がとても重要であるとの認識に至ったのでした。

的確な再利用、変更容易性、テスト容易性を実践していくためには、経営の方針、組織の特徴や文化、既存の資産、開発の戦略、そして顧客の現場で話されている言葉に敏感でなくてはなりません。また、既存の技術基盤や方法論の見極めと、新規の技術開発が不可欠です。これらをシステムに盛り込むため、レイヤ化アーキテクチャを採用し、経営視点からのシステム構成要素および技術視点からのシステム構成要素のプラグイン構造をフレームワークとして実現する必要があるでしょう。

本書には、開発者がドメインに精通していく上で実践すべきノウハウが詰まっています。有用なパターンを一つだけ紹介しろと言われたら、選択に迷ってしまいます。しかし、あえて、一つ重要なパターンを挙げるとしたら、「腐敗防止層」を私は選択します。昨今M&Aによるシステムの統合、業務の見直しなどが直近の課題として重要性を増しています。

もちろん、既存システムと新規システムの統合などは古くから数限りなくあるでしょう。本書では、ファサードやアダプタなどのデザインパターンを取り上げて平易に解説していますが、同時に限界も示しています。この部分が大変重要であり、書籍全体を通じて言えることとして、失敗経験や限界についての議論を提示することで、設計判断の余地をしっかりと残してくれているのです。設計判断は、複数のシステムおよびシステム系列を通じて、次の事態を想定しつつアーキテクトが責任を持って積み上げていくことになります。システム化の効果を得るために、遠回りをすることもあるかもしれませんが、それも一つの判断になると思います。システムが成功したところで、顧客のビジネスが成功しなければ元も子もないのです。

最後になりますが、本書は、翻訳書籍となる以前に、有志による多くの勉強会で有用な題材として取り上げられてきました。なかなか社外の勉強会などに参加できない方も多いと思われます。この機に、社内外の勉強会等で取り上げていただき、解決すべき問題について多くの議論を重ねて頂きたいと思います。特に社外の勉強会では、ドメイン用語や技術者が持つ語彙の違いに気づくと同時に、システム開発におけるより広い観点と示唆を得られます。その場で培われた知識や発見は、きっとシステム開発に有効に働くでしょう。本書が、システムとビジネスの成功に少しでも寄与するようお祈り申し上げます。

今関　剛

株式会社イマテック　代表取締役

著者紹介

Eric Evans（エリック・エヴァンス）

Eric Evansは、企業のビジネスと深く結び付いた進化するソフトウェア構築の支援を専門とするコンサルティンググループDomain Languageの創設者である。1980年代から、複数の複雑なビジネスドメインと技術的ドメインの大型オブジェクト指向システムで、彼は設計者兼プログラマとして仕事をしている。また、エクストリームプログラミングの開発チームの教育指導者でもある。

監訳者紹介

今関　剛（いまぜき たけし）

1991年より、大手電機メーカ向けのCAEシステムの構築、導入、組込みソフトウェア開発に従事。製造業における設計開発業務および製品品質の改善に貢献する。2000年より、製造業ドメインにて培った知見とソフトウェア開発技術をベースに、技術コンサルティングおよび組織内プロセス改善に取り組む。現在は、製造業、金融および物流システムの再利用型開発による効率化を目指して技術と管理の両面から改善に取り組んでいる。アーキテクチャ改善では、DSM（Dependency Structure Matrix）による診断経験が豊富。プロダクトラインには2000年より着手し、セミナーおよび記事執筆、翻訳書などの普及活動を行っている。IEEE、SEA、EEBOF、SESSAME各会員、Embedded Software Guild幹事。

訳者紹介

和智右桂（わち ゆうけい）

思想系プログラマ。大学院人文社会系研究科博士課程在籍中にソフトハウスに就職し、現在はSIerに勤務。2008年よりJavaEE勉強会に参加。本書の原典である『Domain Driven Design』との出会いを契機に、概念をとらえるパラダイムとしてのオブジェクト指向に傾倒するようになる。プログラミングパラダイムやソフトウェアアーキテクチャ、アジャイル系開発プロセスについて学びつつ、海外ブログの翻訳や書き下ろし記事を時々ブログで公開している。認定スクラムマスター、認定プロダクトオーナー。

Blog: Digital Romanticism（https://digitalsoul.hatenadiary.org/）、Twitter: @digitalsoul0124

牧野祐子（まきの さちこ）

1986年　国際基督教大学教養学部語学学科卒業

現　在　翻訳家

主な訳書（共訳含む）：『ソフトウェア開発の持つべき文化』、『プロブレムフレーム』、『実践ユーザビリティテスティング』、『Javaスタイルブック』、『システムアーキテクチャ構築の原理』（以上翔泳社刊）、『人月の神話（改装版）』、『ウィンストンのJava』、『Gnu Emacs 19マニュアル』、『標準HTML 4』、『C++実践オブジェクト指向プログラミング』、『大規模C++ソフトウェアデザイン』、『STL 2』（以上ピアソン桐原刊）、『Mac OS X Hacks』、『実用CVS』（オライリー・ジャパン刊）、『[新版] Macintoshではじめる Cプログラミング』、『コンピュータ科学者がめったに語らないこと』（以上SIBアクセス）ほか。

● DTP／編集　株式会社エスアイビー・アクセス
● 装丁　アピア:2

エリック・エヴァンスのドメイン駆動設計

2011年 4月 8日　初版第 1刷発行
2022年 9月10日　初版第14刷発行

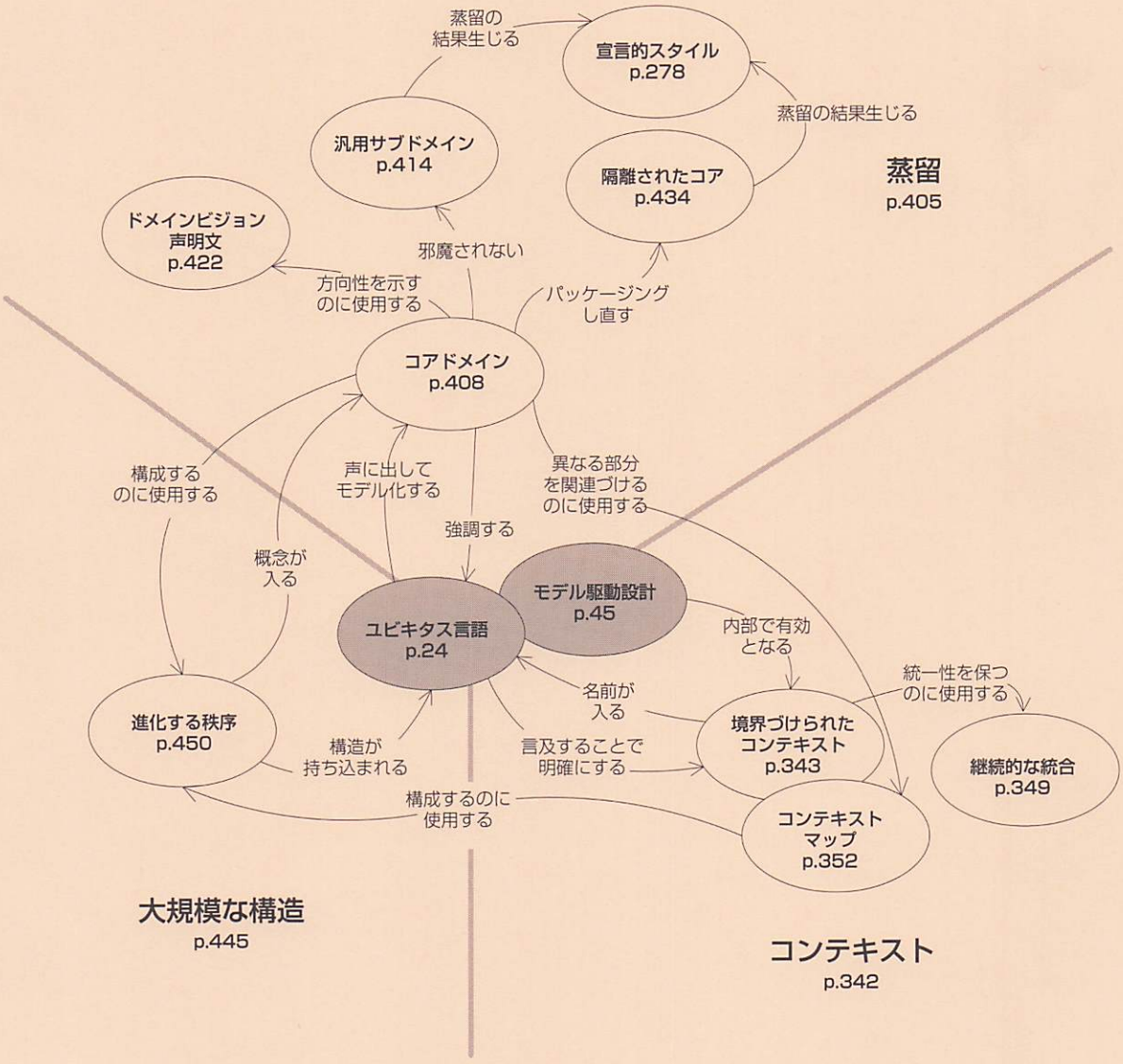
著　者　Eric Evans（エリック・エヴァンス）
監訳者　今関 剛（いまぜき たけし）
訳　者　和智右桂（わち ゆうけい）
　　牧野祐子（まきの さちこ）
発行人　佐々木 幹夫
発行所　株式会社 翔泳社（https://www.shoeisha.co.jp）
印刷・製本　凸版印刷株式会社

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。

ISBN978-4-7981-2196-3　Printed in Japan


蒸留の
結果生じる
宣言的スタイル
p.278
蒸留の結果生じる
汎用サブドメイン
p.414
隔離されたコア
p.434
蒸留
p.405
ドメインビジョン
声明文
p.422
邪魔されない
方向性を示す
のに使用する
パッケージング
し直す
コアドメイン
p.408
構成する
のに使用する
声に出して
モデル化する
異なる部分
を関連づける
のに使用する
強調する
概念が
入る
モデル駆動設計
p.45
ユビキタス言語
p.24
内部で有効
となる
統一性を保つ
のに使用する
名前が
入る
進化する秩序
p.450
構造が
持ち込まれる
言及することで
明確にする
境界づけられた
コンテキスト
p.343
継続的な統合
p.349
構成するのに
使用する
コンテキスト
マップ
p.352
大規模な構造
p.445
コンテキスト
p.342

「この本は、思慮深いソフトウェア開発者全員の書棚にふさわしい。」—Kent Beck

「Eric が見事にとらえたのは、熟練のオブジェクト設計者が常々用いてきた設計プロセスの一部でありながら、グループとして見ると、この業界の他の人々へうまく伝えられずにいたものだ。これまで、我々は、この知識を断片的には提供してきた。しかし、ドメインロジックを構築するための原理をまとめ上げ、体系化したことはなかった。本書は重要である。」
— Kyle Blown『Enterprise Java Programming with IBM WebSphere』の著者

ドメイン駆動設計

ソフトウェア開発コミュニティでは、ドメインモデリングがソフトウェア設計の中心であることが広く認められてきている。ドメインモデルを通して、ソフトウェア開発者は豊富な機能を表現し、それをユーザの要求に本当の意味で応えるソフトウェアの実装に移すことができる。しかし、明らかに重要であるにもかかわらず、効果的なドメインモデリングをどのようにソフトウェア開発プロセスに組み入れるかを説明する、実用的なリソースはほとんど存在しない。ドメイン駆動設計はこの要求に応えるものだ。これは具体的な技術についての本ではない。読者にドメイン駆動設計への体系的なアプローチを提示するものなのだ。設計のベストプラクティスの応用的なセット、経験に基づくテクニック、さらに、複雑なドメインに直面するソフトウェアプロジェクトにおける開発を容易にする基本原則を紹介する。
本書は、設計と開発のプラクティスを縒り合わせながら、実際のプロジェクトに基づく数々の例に立脚して、ドメイン駆動設計を現実のソフトウェア開発に適用する方法を説明している。読者は、ドメインモデルを用いて、複雑な開発作業をより集中してダイナミックに行うにはどうすればよいかを学ぶことになる。ベストプラクティスと標準パターンというコアによって、開発チームに共通の言語が提供される。コードだけではなく、コードの根底にあるモデルをリファクタリングするという力点の移動が、アジェイル開発における頻繁なイテレーションと相まって、ドメインに対するより深い洞察を生み出し、ドメインエキスパートとプログラマのコミュニケーションを高める。ドメイン駆動設計は、さらにこれを土台として、複雑なシステムと巨大な組織のためのモデリングと設計に取り組む。

具体的な題材には以下のようなものがある。

- チームメンバー全員に同じ言語で会話をさせる
- モデルと実装をより深く結び付ける
- ポイントとなる特徴を、モデルにおいてはっきりさせる
- ドメインオブジェクトのライフサイクルを管理する
- 安全に結合できるドメインコードを入念に作成する
- 複雑なコードを明白で予測できるものにする
- ドメインビジョン声明文を記述する
- 複雑なドメインのコアを蒸留する
- モデルに必要な、暗黙的な概念を掘り出す
- アナリシスパターンを適用する
- デザインパターンをモデルに関係づける
- 巨大なシステムでモデルの整合性を維持する
- 同じプロジェクトで複数のモデルが共存できるようにする
- 大規模な構造によりシステムを構成する
- モデリングのブレイクスルーを認識して、それに対応する

本書を手にすることにより、オブジェクト指向開発者、システムアナリスト、および設計者は、自分の仕事を計画して、そこに集中することや、豊かで役に立つドメインモデルを作成すること、さらにそれらのモデルを高品質で長く使われるソフトウェアの実装に利用することの指針を得るであろう。