

DOMAIN-DRIVEN DESIGN

TACKLING COMPLEXITY  
IN THE HEART OF SOFTWARE

Pearson

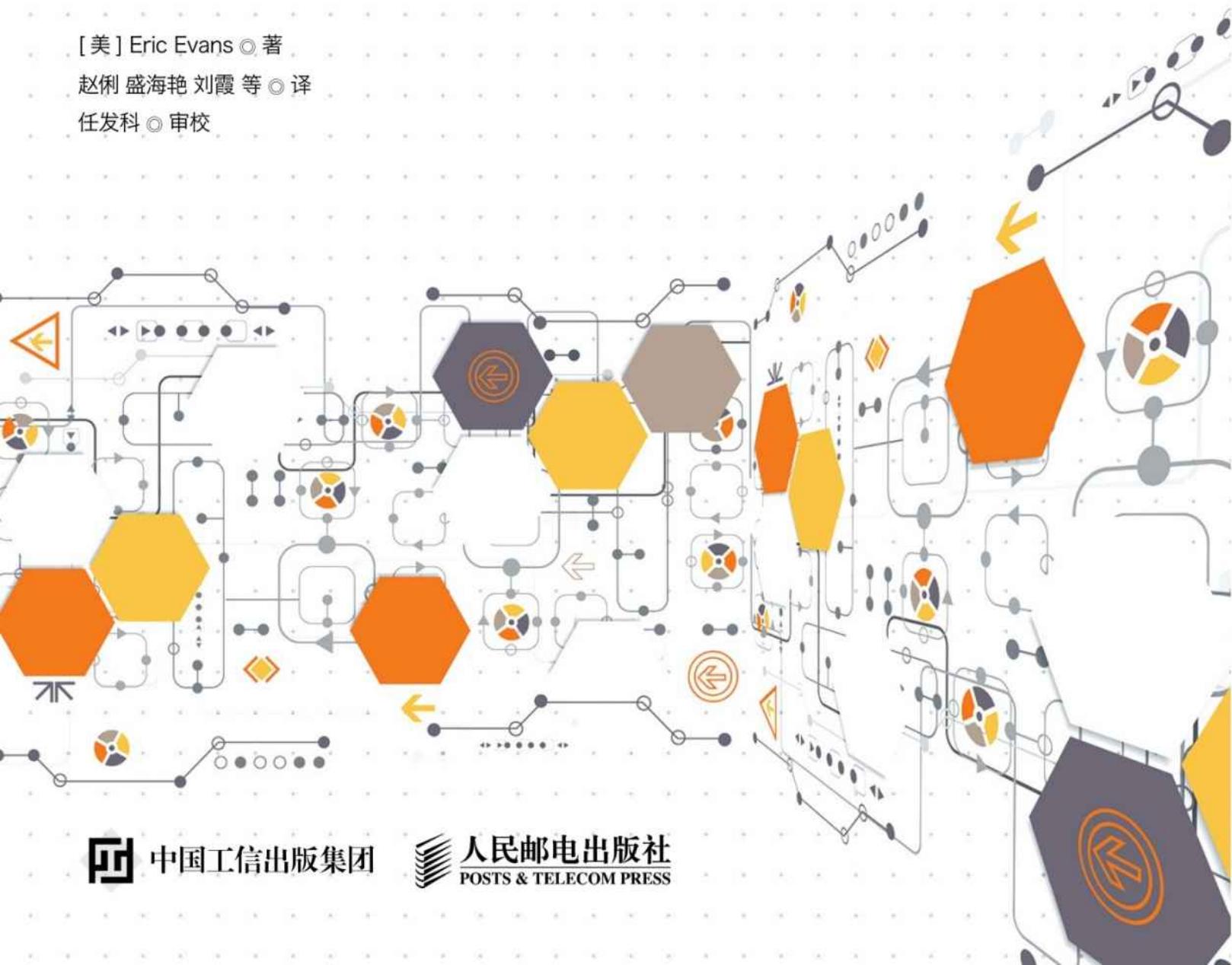
# 领域驱动设计

## 软件核心复杂性应对之道

[美] Eric Evans ◎ 著

赵俐 盛海艳 刘霞 等 ◎ 译

任发科 ◎ 审校



中国工信出版集团



人民邮电出版社  
POSTS & TELECOM PRESS

# 目 錄

---

[封面](#)

[其他](#)

[扉頁](#)

[版權](#)

[版權聲明](#)

[譯者序](#)

[序](#)

[前言](#)

[致謝](#)

[第一部分 運用領域模型](#)

[第1章 消化知識](#)

[1.1 有效建模的要素](#)

[1.2 知識消化](#)

[1.3 持續學習](#)

[1.4 知識豐富的設計](#)

[1.5 深層模型](#)

[第2章 交流與語言的使用](#)

[2.1 模式：UBIQUITOUS LANGUAGE](#)

[2.2 「大聲地」建模](#)

[2.3 一個團隊，一種語言](#)

[2.4 文檔和圖](#)

[2.4.1 書面設計文檔](#)

[2.4.2 完全依賴可執行代碼的情況](#)

[2.5 解釋性模型](#)

[第3章 綁定模型和實現](#)

[3.1 模式：MODEL-DRIVEN DESIGN](#)

### 3.2 建模範式和工具支持

### 3.3 揭示主旨：為什麼模型對用戶至關重要

### 3.4 模式：HANDS-ON MODELER

## 第二部分 模型驅動設計的構造塊

### 第4章 分離領域

#### 4.1 模式：LAYERED ARCHITECTURE

##### 4.1.1 將各層關聯起來

##### 4.1.2 架構框架

#### 4.2 領域層是模型的精髓

#### 4.3 模式：THE SMART UI「反模式」

#### 4.4 其他分離方式

### 第5章 軟件中所表示的模型

#### 5.1 關聯

#### 5.2 模式：ENTITY ( 又稱為REFERENCE OBJECT )

##### 5.2.1 ENTITY建模

##### 5.2.2 設計標識操作

#### 5.3 模式：VALUE OBJECT

##### 5.3.1 設計VALUE OBJECT

##### 5.3.2 設計包含VALUE OBJECT的關聯

#### 5.4 模式：SERVICE

##### 5.4.1 SERVICE與孤立的領域層

##### 5.4.2 粒度

##### 5.4.3 對SERVICE的訪問

#### 5.5 模式：MODULE ( 也稱為PACKAGE )

##### 5.5.1 敏捷的MODULE

##### 5.5.2 通過基礎設施打包時存在的隱患

#### 5.6 建模範式

##### 5.6.1 對像範式流行的原因

## 5.6.2 對像世界中的非對像

## 5.6.3 在混合範式中堅持使用 MODEL-DRIVEN DESIGN

# 第6章 領域對象的生命週期

## 6.1 模式：AGGREGATE

## 6.2 模式：FACTORY

### 6.2.1 選擇FACTORY及其應用位置

### 6.2.2 有些情況下只需使用構造函數

### 6.2.3 接口的設計

### 6.2.4 固定規則的相關邏輯應放置在哪裡

### 6.2.5 ENTITY FACTORY與VALUE OBJECT FACTORY

### 6.2.6 重建已存儲的對象

## 6.3 模式：REPOSITORY

### 6.3.1 REPOSITORY的查詢

### 6.3.2 客戶代碼可以忽略REPOSITORY的實現，但開發人員不能忽略

### 6.3.3 REPOSITORY的實現

### 6.3.4 在框架內工作

### 6.3.5 REPOSITORY與FACTORY的關係

## 6.4 為關係數據庫設計對像

# 第7章 使用語言：一個擴展的示例

## 7.1 貨物運輸系統簡介

## 7.2 隔離領域：引入應用層

## 7.3 將ENTITY和VALUE OBJECT區別開

## 7.4 設計運輸領域中的關聯

## 7.5 AGGREGATE邊界

## 7.6 選擇REPOSITORY

## 7.7 場景走查

7.7.1 應用程序特性舉例：更改Cargo的目的地

7.7.2 應用程序特性舉例：重複業務

7.8 對象的創建

7.8.1 Cargo的FACTORY和構造函數

7.8.2 添加Handling Event

7.9 停一下，重構：Cargo AGGREGATE的另一種設計

7.10 運輸模型中的MODULE

7.11 引入新特性：配額檢查

7.11.1 連接兩個系統

7.11.2 進一步完善模型：劃分業務

7.11.3 性能優化

7.12 小結

第三部分 通過重構來加深理解

第8章 突破

8.1 一個關於突破的故事

8.1.1 華而不實的模型

8.1.2 突破

8.1.3 更深層模型

8.1.4 冷靜決策

8.1.5 成果

8.2 機遇

8.3 關注根本

8.4 後記：越來越多的新理解

第9章 將隱式概念轉變為顯式概念

9.1 概念挖掘

9.1.1 傾聽語言

9.1.2 檢查不足之處

9.1.3 思考矛盾之處

9.1.4 查閱書籍

9.1.5 嘗試 · 再嘗試

9.2 如何為那些不太明顯的概念建模

9.2.1 顯式的約束

9.2.2 將過程建模為領域對像

9.2.3 模式 : SPECIFICATION

9.2.4 SPECIFICATION的應用和實現

第10章 柔性設計

10.1 模式 : INTENTION-REVEALING INTERFACES

10.2 模式 : SIDE-EFFECT-FREE FUNCTION

10.3 模式 : ASSERTION

10.4 模式 : CONCEPTUAL CONTOUR

10.5 模式 : STANDALONE CLASS

10.6 模式 : CLOSURE OF OPERATION

10.7 聲明式設計

10.8 聲明式設計風格

10.9 切入問題的角度

10.9.1 分割子領域

10.9.2 盡可能利用已有的形式

第11章 應用分析模式

第12章 將設計模式應用於模型

12.1 模式 : STRATEGY ( 也稱為POLICY )

12.2 模式 : COMPOSITE

12.3 為什麼沒有介紹FLYWEIGHT

第13章 通過重構得到更深層的理解

13.1 開始重構

13.2 探索團隊

13.3 借鑒先前的經驗

[13.4 針對開發人員的設計](#)

[13.5 重構的時機](#)

[13.6 危機就是機遇](#)

## [第四部分 戰略設計](#)

### [第14章 保持模型的完整性](#)

[14.1 模式 : BOUNDED CONTEXT](#)

[14.2 模式 : CONTINUOUS INTEGRATION](#)

[14.3 模式 : CONTEXT MAP](#)

[14.3.1 測試CONTEXT的邊界](#)

[14.3.2 CONTEXT MAP的組織和文檔化](#)

[14.4 BOUNDED CONTEXT之間的關係](#)

[14.5 模式 : SHARED KERNEL](#)

[14.6 模式 : CUSTOMER/SUPPLIER DEVELOPMENT TEAM](#)

[14.7 模式 : CONFORMIST](#)

[14.8 模式 : ANTICORRUPTION LAYER](#)

[14.8.1 設計ANTICORRUPTION LAYER的接口](#)

[14.8.2 實現ANTICORRUPTION LAYER](#)

[14.8.3 一個關於防禦的故事](#)

[14.9 模式 : SEPARATE WAY](#)

[14.10 模式 : OPEN HOST SERVICE](#)

[14.11 模式 : PUBLISHED LANGUAGE](#)

[14.12 「大象」的統一](#)

[14.13 選擇你的模型上下文策略](#)

[14.13.1 團隊決策或更高層決策](#)

[14.13.2 路身上下文中](#)

[14.13.3 轉換邊界](#)

#### 14.13.4 接受那些我們無法更改的事物：描述外部系統

#### 14.13.5 與外部系統的關係

#### 14.13.6 設計中的系統

#### 14.13.7 用不同模型滿足特殊需要

#### 14.13.8 部署

#### 14.13.9 權衡

#### 14.13.10 當項目正在進行時

### 14.14 轉換

#### 14.14.1 合併 CONTEXT : SEPARATE WAY →SHARED KERNEL

#### 14.14.2 合併 CONTEXT : SHARED KERNEL→CONTINUOUS INTEGRATION

#### 14.14.3 逐步淘汰遺留系統

#### 14.14.4 OPEN HOST SERVICE→PUBLISHED LANGUAGE

## 第15章 精煉

### 15.1 模式：CORE DOMAIN

#### 15.1.1 選擇核心

#### 15.1.2 工作的分配

### 15.2 精煉的逐步提升

### 15.3 模式：GENERIC SUBDOMAIN

#### 15.3.1 通用不等於可重用

#### 15.3.2 項目風險管理

### 15.4 模式：DOMAIN VISION STATEMENT

### 15.5 模式：HIGHLIGHTED CORE

#### 15.5.1 精煉文檔

#### 15.5.2 標明CORE

[15.5.3 把精煉文檔作為過程工具](#)

[15.6 模式 : COHESIVE MECHANISM](#)

[15.6.1 GENERIC SUBDOMAIN 與 COHESIVE MECHANISM的比較](#)

[15.6.2 MECHANISM是CORE DOMAIN一部分](#)

[15.7 通過精煉得到聲明式風格](#)

[15.8 模式 : SEGREGATED CORE](#)

[15.8.1 創建SEGREGATED CORE的代價](#)

[15.8.2 不斷發展演變的團隊決策](#)

[15.9 模式 : ABSTRACT CORE](#)

[15.10 深層模型精煉](#)

[15.11 選擇重構目標](#)

[第16章 大型結構](#)

[16.1 模式 : EVOLVING ORDER](#)

[16.2 模式 : SYSTEM METAPHOR](#)

[16.3 模式 : RESPONSIBILITY LAYER](#)

[16.4 模式 : KNOWLEDGE LEVEL](#)

[16.5 模式 : PLUGGABLE COMPONENT FRAMEWORK](#)

[16.6 結構應該有一種什麼樣的約束](#)

[16.7 通過重構得到更適當的結構](#)

[16.7.1 最小化](#)

[16.7.2 溝通和自律](#)

[16.7.3 通過重構得到柔性設計](#)

[16.7.4 通過精煉可以減輕負擔](#)

[第17章 領域驅動設計的綜合運用](#)

[17.1 把大型結構與BOUNDED CONTEXT結合起來使用](#)

[17.2 將大型結構與精煉結合起來使用](#)

[17.3 首先評估](#)

## 17.4 由誰制定策略

17.4.1 從應用程序開發自動得出的結構

17.4.2 以客戶為中心的架構團隊

## 17.5 制定戰略設計決策的6個要點

17.5.1 技術框架同樣如此

17.5.2 注意總體規劃

結束語

附錄

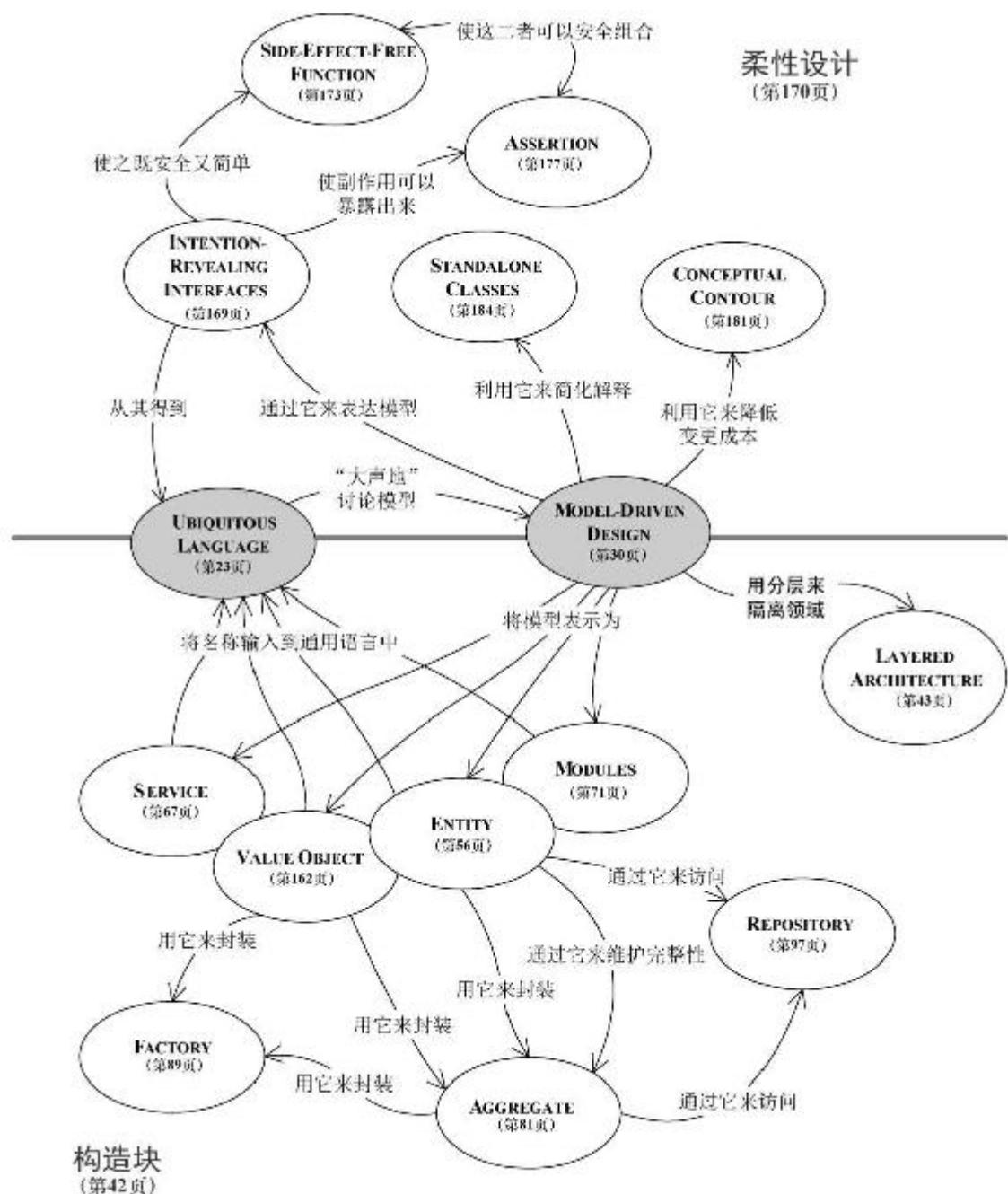
術語表

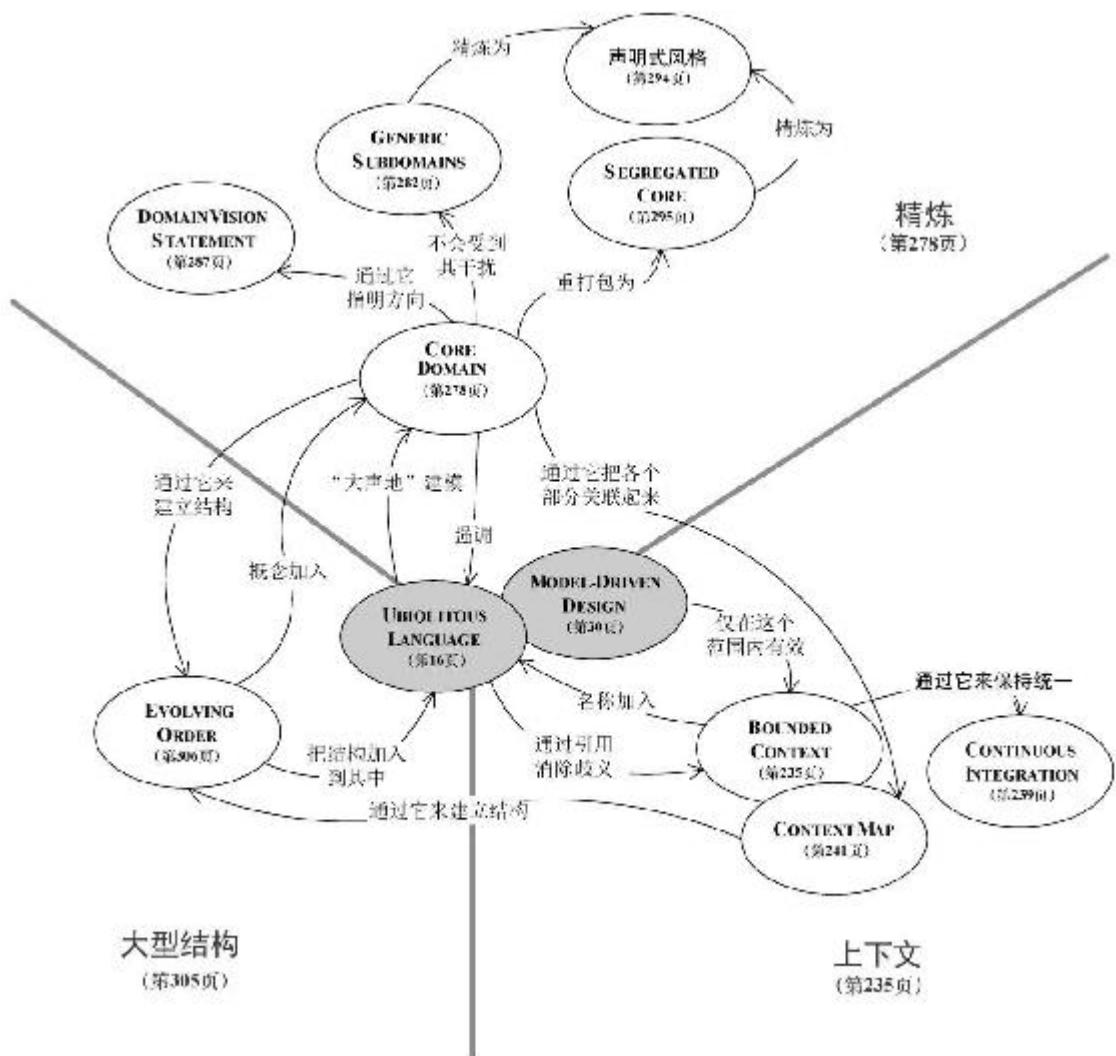
參考文獻

圖片說明

索引

其他





DOMAIN-DRIVEN DESIGN TACKLING COMPLEXITY IN THE HEART  
OF SOFTWARE

領域驅動設計 軟件核心複雜性應對之道

[美]Eric Evans◎著

趙俐 盛海艷 劉霞 等◎譯

任發科◎審校

人民郵電出版社

北京

## 圖書在版編目 (CIP) 數據

領域驅動設計：軟件核心複雜性應對之道 / (美) 埃文斯 (Evans, E.) 著；趙俐等譯。--2版 (修訂本)。--北京：人民郵電出版社，2016.6

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

ISBN 978-7-115-37675-6

I. 1 領... II. 1 埃... 2 趙... III. 1 軟件設計 IV. 1 TP311.5

中國版本圖書館CIP數據核字 (2016) 第069725號

### 內容提要

本書是領域驅動設計方面的經典之作，修訂版更是對之前出版的中文版進行了全面的修訂和完善。

全書圍繞著設計和開發實踐，結合若干真實的項目案例，向讀者闡述如何在真實的軟件開發中應用領域驅動設計。書中給出了領域驅動設計的系統化方法，並將人們普遍接受的一些最佳實踐綜合到一起，融入了作者的見解和經驗，展現了一些可擴展的設計最佳實踐、已驗證過的技術以及便於應對複雜領域的軟件項目開發的基本原則。

本書適合各層次的面向對像軟件開發人員、系統分析員閱讀。

◆著 [美]Eric Evans

譯 趙俐 盛海艷 劉霞 等

審校 任發科

責任編輯 楊海玲

責任印製 焦志輝

◆人民郵電出版社出版發行 北京市豐臺區成壽寺路11號

郵編 100164 電子郵件 315@ptpress.com.cn

網址 <http://www.ptpress.com.cn>

三河市海波印務有限公司印刷

◆開本：800×1000 1/16

印張：24.25

字數：515千字 2016年6月第2版

印數：1-3000冊 2016年6月河北第1次印刷

著作權合同登記號 圖字：01-2010-0695號

定價：69.00

讀者服務熱線：**(010)81055410** 印裝質量熱線：

**(010)81055316**

反盜版熱線：**(010)81055315**

## 版權聲明

Authorized translation from the English language edition,entitled Domain-Driven Design:Tackling Complexity in the Heart of Software,9780321125217 by Eric Evans,published by Pearson Education,Inc.,publishing as Addison-Wesley,Copyright c 2004 by Eric Evans.

All rights reserved.No part of this book may be reproduced or transmitted in any form or by any means,electronic or mechanical,including photocopying,recording or by any information storage retrieval system,without permission from Pearson Education,Inc.

CHINESE SIMPLIFIED language edition published by PEARSON EDUCATION ASIA LTD.and POSTS & TELECOM PRESS Copyright c 2016.

本書中文簡體字版由 Pearson Education Asia Ltd. 授權人民郵電出版社獨家出版。未經出版者書面許可，不得以任何方式複製或抄襲本書內容。

本書封面貼有 Pearson Education ( 培生教育出版集團 ) 激光防偽標籤，無標籤者不得銷售。

版權所有，侵權必究。

## 譯者序

我最早聽說Eric Evans的《領域驅動設計》是在2007年，那時我所在的項目組出於知識儲備的考慮購進了一批軟件設計書和相關資料。其中一篇英文的短篇技術文檔與我們當時的項目非常相關，於是我們就仔細研讀了一番。這篇僅有幾萬字的文檔多次提到了Eric Evans的《領域驅動設計》，並引用了他的很多精闢觀點。由於當時領域驅動設計遠遠沒有現在這樣普及，因此這些觀點使我耳目一新，也給我留下了深刻的印象。隨後我又經常在一些文獻中看到Eric Evans的名字，更多地瞭解了他的領域驅動設計思想，沒想到時隔幾年後竟然有機會把這位大師的作品翻譯出來奉獻給各位讀者，也算是機緣巧合了。

相信大家對這本書都不陌生，它已經成為軟件設計書中的經典。在網上搜索一下，讀者對它好評如潮，我再多說一句讚美的話都是多餘的。而我能想到的也唯有「經典」二字，它堪稱經典中的經典。

我們對「領域」這個概念都很熟悉，但有多少人真正重視過它呢？軟件開發人員幾乎總是專注於技術，把技術作為自己能力的展示和成功的度量。而直到Eric Evans出版了他的這部鉅著之後，人們才真正開始關注領域，關注核心領域，關注領域驅動的設計，關注模型驅動的開發。相信在讀完本書後，你會對軟件設計有全新的認識。

我曾經和一些好友探討過以下一些問題。項目怎樣開發才能確保成功？什麼樣的軟件才能為用戶提供真正的價值？什麼樣的團隊才算

是優秀的團隊？現在，在仔細研讀完本書後，這些問題都找到了答案。

本書廣泛適用於各種領域的軟件開發項目。在每個項目的生命週期中，都會有一些重大關頭或轉折點。如何制定決策，如何把握項目的方向，如何處理和面對各種機會和挑戰，將對項目產生決定性的影響。讓我們一起跟隨大師的腳步，分享他通過大量項目獲得的真知灼見和開發心得吧。

最後，衷心感謝人民郵電出版社各位編輯在翻譯工作中給予的幫助和寶貴意見，感謝熱心讀者魏海楓，他在百忙之中抽出時間對本書譯稿做了修訂工作，發現並修正了很多問題。由於譯者水平有限，在翻譯過程中難免還會留有一些錯誤，懇請讀者批評指正。

# 序

有很多因素會使軟件開發複雜化，但最根本的原因是問題領域本身錯綜複雜。如果你要為一家人員複雜的企業提高自動化程度，那麼你開發的軟件將無法迴避這種複雜性，你所能做的只有控制這種複雜性。

控制複雜性的關鍵是有一個好的領域模型，這個模型不應該僅僅停留在領域的表面，而是要透過表象抓住領域的實質結構，從而為軟件開發人員提供他們所需的支持。好的領域模型價值連城，但要想開發出好的模型也並非易事。精通此道的人並不多，而且這方面的知識也很難傳授。

Eric Evans就是為數不多的能夠創建出優秀領域模型的人。我是在與他合作時發現他的這種才能的——發現一個客戶竟然比我技術更精湛，這種感覺有些奇妙。我們的合作雖然短暫，但卻充滿樂趣。從那之後我們一直保持聯繫，我也有幸見證了本書整個「孕育」過程。

本書絕對值得期待。

本書終於實現了一個宏偉抱負，即描述並建立了領域建模藝術的詞彙庫。它提供了一個參考框架，人們可以用它來解釋相關活動，並用它來傳授這門難學的技藝。本書在寫作過程中，也帶給我很多新想法，如果哪位概念建模方面的老手沒有從閱讀本書中獲得大量的新思想，那我反而該驚訝莫名了。

Eric還對我們多年以來學過的知識進行了歸納總結。首先，在領域建模過程中不應將概念與實現割裂開來。高效的領域建模人員不僅應該能夠在白板上與會計師進行討論，而且還應該能與程序員一道編寫Java代碼。之所以要具備這些能力，一部分原因是如果不考慮實現問題就無法構建出有用的概念模型。但概念與實現密不可分的最主要原因在於，領域模型的最大價值是它提供了一種通用語言，這種語言是將領域專家和技術人員聯繫在一起的紐帶。

我們將從本書中學到的另一個經驗是領域模型並不是按照「先建模，後實現」這個次序來工作的。像很多人一樣，我也反對「先設計，再構建」這種固定的思維模式。Eric的經驗告訴我們，真正強大的領域模型是隨著時間演進的，即使是最有經驗的建模人員也往往發現他們是在系統的初始版本完成之後才有了最好的想法。

我衷心希望本書成為一本有影響力的著作，並希望本書能夠將如何利用領域模型這一寶貴工具的知識傳授給更多的人，從而為這個高深莫測的領域梳理出一個結構，並使它更有內聚力。領域模型對軟件開發的控制有著巨大影響，不管軟件開發是用什麼語言或環境實現的。

最後，也是很重要的一點，我最敬佩Eric的一點是他敢於在本書中談論自己的一些失敗經歷。很多作者都喜歡擺出一副無所不能的架勢，有時著實讓人不屑。但Eric清楚地表明他像我們大多數人一樣，既品嚐過成功的美酒，也體驗過失敗的沮喪。重要的是他能夠從成功和失敗中學習，而對我們來說更重要的是他能夠將所有經驗傳授給我們。

Martin Fowler

2003年4月

# 前言

至少20年前，一些頂尖的軟件設計人員就已經認識到領域建模和設計的重要性，但令人驚訝的是，這麼長時間以來幾乎沒有人寫出點兒什麼，告訴大家應該做哪些工作或如何去做。儘管這些工作還沒有被清楚地表述出來，但一種新的思潮已經形成，它像一股暗流一樣在對像社區中湧動，我把這種思潮稱為領域驅動設計（domain-driven design）。

過去10年中，我在幾個業務和技術領域開發了一些複雜的系統。我在設計和開發過程中嘗試了一些最佳實踐，它們都是面向對像開發高手用過的領先技術。有些項目非常成功，但有幾個項目卻失敗了。成功的項目有一個共同的特徵，那就是都有一個豐富的領域模型，這個模型在迭代設計的過程中不斷演變，而且成為項目不可分割的一部分。

本書為作出設計決策提供了一個框架，並且為討論領域設計提供了一個技術詞彙庫。本書將人們普遍接受的一些最佳實踐綜合到一起，並融入了我自己的見解和經驗。面對複雜領域的軟件開發團隊可以利用這個框架來系統性地應用領域驅動設計。

## **三個項目的對比**

談到領域設計實踐對開發結果的巨大影響時，我的記憶中立即就會跳出三個項目，它們就是鮮活的例子。雖然這三個項目都交付了有

用的軟件，但只有一個項目實現了宏偉的目標——交付了能夠滿足組織後續需求、可以不斷演進的複雜軟件。

我要說的第一個項目完成得很迅速，它提供了一個簡單實用的Web交易系統。開發人員主要憑直覺開發，但這並沒有妨礙他們，因為簡單軟件的編寫並不需要過多地注意設計。由於最初的這次成功，人們對未來開發的期望值變得極高。我就是在這個時候被邀請開發它的第二個版本的。當我仔細研究這個項目時，發現他們沒有使用領域模型，甚至在項目中沒有一種公共語言，而且項目完全沒有一種結構化的設計。項目領導者對我的評價並不贊同，於是我就拒絕了這項工作。一年後，這個項目團隊陷入困境，無法交付第二個版本。儘管他們在技術的使用方面也值得商榷，但真正挫敗他們的是業務邏輯。他們的第一個版本過早地變得僵化，成為一個維護代價十分高昂的遺留系統。

要想克服這種複雜性，需要非常嚴格地使用領域邏輯設計方法。在我職業生涯的早期，我幸運地完成了一個非常重視領域設計的項目，這就是我要說的第二個項目。這個項目的領域複雜性與上面提到的那個項目相仿，它最初也小獲成功，為貿易機構提供了一個簡單的應用程序。但在最初交付之後緊跟著又進行了連續的加速開發。每次迭代都為上一個版本在功能的集成和完善上增加了非常好的新選項。開發團隊能夠按照貿易商的要求提供靈活性和擴展性。這種良性發展直接歸功於深刻的領域模型，它得到了反覆精化，並在代碼中得以體現。當團隊對該領域有了新的理解後，領域模型也隨之深化。開發人員之間、開發人員與領域專家之間的溝通質量都得到改善，而且設計不但沒有加重維護負擔，反而變得易於修改和擴展。

遺憾的是，僅靠重視模型並不會使項目達到這樣的良性循環。我要說的第三個項目就是這種情況，它開始制訂的目標很高，打算基於

一個領域模型建立一個全球企業系統，但在經過了幾年的屢戰屢敗之後，不得不降格以求，最終「泯然眾人矣」。團隊擁有很多好的工具，對業務也有較好的理解，也非常認真地進行了建模。但團隊卻錯誤地將開發人員的角色獨立出來，導致建模與實現脫節，因此設計無法反映不斷深化的分析。總之，詳細的業務對像設計不能保證它們能夠嚴絲合縫地被整合到複雜的應用程序中。反覆的迭代並沒有使代碼得以改進，因為開發人員的技術水平參差不齊，他們沒有認識到他們使用了非正式的風格和技術體系來創建基於模型的對象（這些對象也充當了實用的、可運行的軟件）。幾個月過去了，開發工作由於巨大的複雜性而陷入困境，而團隊對項目也失去了一致的認識。經過幾年的努力，項目確實創建了一個適當的、有用的軟件，但團隊已經放棄了當初的宏偉抱負，也不再重視模型。

### 複雜性的挑戰

很多因素可能會導致項目偏離軌道，如官僚主義、目標不清、資源缺乏等。但真正決定軟件複雜性的是設計方法。當複雜性失去控制時，開發人員就無法很好地理解軟件，因此無法輕易、安全地更改和擴展它。而好的設計則可以為開發複雜特性創造更多機會。

一些設計因素是技術上的。軟件的網絡、數據庫和其他技術方面的設計耗費了人們大量的精力。很多書籍都介紹過如何解決這些問題。大批開發人員很注意培養自己的技能，並緊跟每一次技術進步。

然而很多應用程序最主要的複雜性並不在技術上，而是來自領域本身、用戶的活動或業務。當這種領域複雜性在設計中沒有得到解決時，基礎技術的構思再好也無濟於事。成功的設計必須系統地考慮軟件的這個核心方面。

本書有兩個前提：

(1) 在大多數軟件項目中，主要的焦點應該是領域和領域邏輯；

(2) 複雜的領域設計應該基於模型。

領域驅動設計是一種思維方式，也是一組優先任務，它旨在加速那些必須處理複雜領域的軟件項目的開發。為了實現這個目標，本書給出了一整套完整的設計實踐、技術和原則。

## 設計過程與開發過程

設計書就是講設計，過程書只是講過程。它們之間很少互相參考。設計和過程本身就是兩個足夠複雜的主題。本書是一本設計書，但我相信設計與過程這二者是密不可分的。設計思想必須被成功實現，否則它們就只是紙上談兵。

當人們學習設計技術時，各種可能性令他們興奮不已，然而真實項目的錯綜複雜又會為他們潑上一盆冷水。他們無法用所使用的技術來貫徹新的設計思想，或者不知道何時應該為了節省時間而放棄某個設計方面，何時又應該堅持不懈直至找到一個乾淨利落的解決方案。開發人員可以抽像地討論設計原則的應用，而且他們也確實在進行著這樣的討論，但更自然的做法應該是討論如何完成實際工作。因此，雖然本書是一本有關設計的書，但我會在必要的時候穿越這條人為設置的邊界，進入過程的領域。這有助於將設計原則放到一個適當的語境下進行討論。

雖然本書並不侷限於某一種特定的方法，但主要還是面向「敏捷開發過程」這一新體系。特別地，本書假定項目必須遵循兩個開發實踐，要想應用書中所講的方法，必須先瞭解這兩個實踐。

(1) 迭代開發。人們倡導和實踐迭代開發已經有幾十年時間了，而且它是敏捷開發方法的基礎。在敏捷開發和極限編程 (XP) 的文獻中有很多關於迭代開發的精彩討論，其中包括 *Surviving Object-*

Oriented Projects[Cockburn 1998][\[1\]](#) 和 Extreme Programming Explained[Beck 1999]。

(2) 開發人員與領域專家俱有密切的關係。領域驅動設計的實質就是消化吸收大量知識，最後產生一個反映深層次領域知識並聚焦於關鍵概念的模型。這是領域專家與開發人員的協作過程，領域專家精通領域知識，而開發人員知道如何構建軟件。由於開發過程是迭代式的，因此這種協作必須貫穿整個項目的生命週期。

極限編程的概念是由Kent Beck、Ward Cunningham和其他人共同提出的[Beck 2000]，它是敏捷過程最重要的部分，也是我使用得最多的一種編程方法。為了使討論更加具體，整本書都將使用XP作為基礎討論設計和過程的交互。本書論述的原則很容易應用於其他敏捷過程。

近年來，反對「精細開發方法學」(elaborate development methodology)的呼聲漸起，人們認為無用的靜態文檔以及死板的預先規劃和設計加重了項目的負擔。相反，敏捷過程(如XP)強調的是應對變更和不確定性的能力。

極限編程承認設計決策的重要性，但強烈反對預先設計。相反，它將相當大的精力投入到促進溝通和提高項目快速變更能力的工作中。具有這種反應能力之後，開發人員就可以在項目的任何階段只利用「最簡單而管用的方案」，然後不斷進行重構，一步一步做出小的設計改進，最終得到滿足客戶真正需要的設計。

這種極端的簡約主義是解救那些過度追求設計的執迷者的良方。那些幾乎沒有價值的繁瑣文檔只會為項目帶來麻煩。項目受到「分析癱瘓症」的困擾，團隊成員十分擔心會出現不完美的設計，這導致他們根本沒法取得進展。這種狀況必須得到改變。

遺憾的是，這些有關過程的思想可能會被誤解。每個人對「最簡單」都有不同的定義。持續重構其實是一系列小規模的重新設計，沒有嚴格設計原則的開發人員將會創建出難以理解或修改的代碼，這恰好與敏捷的精神相悖。而且，雖然對意外需求的擔心常常導致過度設計，但試圖避免過度設計又可能走向另一個極端——不敢做任何深入的設計思考。

實際上，XP最適合那些對設計的感覺很敏銳的開發人員。XP過程假定人們可以通過重構來改進設計，而且可以經常、快速地完成重構。但重構本身的難易程度取決於先前的設計選擇。XP過程試圖改善團隊溝通，但模型和設計的選擇有可能使溝通更明確，也有可能會使溝通不暢。

本書將設計和開發實踐結合起來討論，並闡述領域驅動設計與敏捷開發過程是如何互相增強的。在敏捷開發過程中使用成熟的領域建模方法可以加速開發。過程與領域開發之間的相互關係使得這種方法比任何「純粹」真空式的設計更加實用。

## 本書的結構

本書分為4個部分。

第一部分「運用領域模型」提出領域驅動開發的基本目標，這些目標是後面幾部分中所討論的實踐的驅動因素。由於軟件開發方法有很多，因此第一部分還定義了一些術語，並給出了用領域模型來驅動溝通和設計的總體含義。

第二部分「模型驅動設計的構造塊」將面向對像領域建模中的一些核心的最佳實踐提煉為一組基本的構造塊。這一部分主要是消除模型與實際運行的軟件之間的鴻溝。團隊一致使用這些標準模式就可以使設計井然有序，並且使團隊成員更容易理解彼此的工作。使用標準

模式還可以為公共語言貢獻術語，使得所有團隊成員可以使用這些術語來討論模型和設計決策。

但這一部分的主旨是討論一些能夠保持模型和實現之間互相協調並提高效率的設計決策。要想達到這種協調，需要密切注意個別元素的一些細節。這種小規模的仔細設計為開發人員提供了一個穩固的基礎，在此基礎上就可以應用第三部分和第四部分討論的建模方法了。

第三部分「通過重構來加深理解」討論如何將構造塊裝配為實用的模型，從而實現其價值。這一部分沒有直接討論深奧的設計原則，而是著重強調一個發現過程。有價值的模型不是立即就會出現的，它們需要對領域的深入理解。這種理解是一步一步得到的，首先需要深入研究模型，然後基於最初的（可能是不成熟的）模型實現一個初始設計，再反覆改進這個設計。每次團隊對領域有了新的理解之後，都需要對模型進行改進，使模型反映出更豐富的知識，而且必須對代碼進行重構，以便反映出更深刻的模型，並使應用程序可以充分利用模型的潛力。這種一層一層「剝洋蔥」的方法有時會創造一種突破的機會，使我們得到更深刻的模型，同時快速進行一些更深入的設計修改。

探索本身是永無止境的，但這並不意味著它是隨機的。第三部分深入闡述一些指引我們保持正確方向的建模原則，並提供了一些指導我們進行探索的方法。

第四部分「戰略設計」討論在複雜系統、大型組織以及與外部系統和遺留系統的交互中出現的複雜情況。這一部分探討了作為一個整體應用於系統的3條原則：上下文、提煉和大型結構。戰略設計決策通常由團隊制定，或者由多個團隊共同制定。戰略設計可以保證在大型系統或應用程序（它們應用於不斷延伸的企業級網絡）上以較大規模去實現第一部分提出的目標。

本書通篇討論使用的例子並不是一些過於簡單的「玩具式」問題，而是全部選自實際項目。

本書的大部分內容實際上是作為一系列的「模式」編寫的。但讀者無需顧忌這一方法也應該能夠理解本書，對模式的風格和格式感興趣的讀者可以參考附錄。

補充材料可以參考<http://domaindrivendesign.org>，該網站提供了示例代碼和社區討論內容。

### **本書面向的讀者**

本書主要是為面向對像軟件開發人員編寫的。軟件項目團隊的大部分成員都能夠從本書的某些部分獲益。本書最適合那些正在項目上嘗試這些實踐的人員，以及那些已經在這樣的項目上積累了豐富經驗的人員。

要想從本書受益，掌握一些面向對像建模知識是非常必要的，如UML圖和Java代碼，因此一定要具備基本讀懂這些語言的能力，但不必精通細節。瞭解極限編程的知識有助於從這個角度來理解開發過程的討論，但不具備這一背景知識也能讀懂這些內容。

一些中級軟件開發人員可能已經瞭解面向對像設計的一些知識，也許讀過一兩本軟件設計的書，那麼本書將填補這些讀者的知識空缺，向他們展示如何在實際的軟件項目上應用對像建模技術。本書將幫助這些開發人員學會用高級建模和設計技巧來解決實際問題。

高級軟件開發人員或專家可能會對書中用於處理領域的綜合框架感興趣。這種系統性的設計方法將幫助技術負責人指導他們的團隊保持正確的方向。此外，本書從頭至尾所使用的明確術語將有助於高級開發人員與他們的同行溝通。

本書採用記敘體，讀者可以從頭至尾閱讀，也可以從任意一章的開頭開始閱讀。具有不同背景知識的讀者可能會有不同的閱讀方式，

但我推薦所有讀者從第一部分的引言和第1章開始閱讀。除此之外，本書的核心是第2、3、9和14章。已經掌握一定知識的讀者可以採取跳躍式閱讀的方式，通過閱讀標題和粗體字內容即可掌握要點。一些高級讀者則可以跳過前兩部分，重點閱讀第三部分和第四部分。

除了這些主要讀者以外，分析員和相關的技術項目經理也可以從閱讀本書中獲益。分析員在掌握了領域與設計之間的聯繫之後，能夠在敏捷項目中作出更卓越的貢獻，也可以利用一些戰略設計原則來更有重點地組織工作。

項目經理感興趣的重點是提高團隊的工作效率，並致力於設計出對業務專家和用戶有用的軟件。由於戰略設計決策與團隊組織和工作風格緊密相關，因此這些設計決策必然需要項目領導者的參與，而且對項目的路線有著重要的影響。

### 領域驅動團隊

儘管開發人員個人能夠從理解領域驅動設計中學到有價值的設計技術和觀點，但最大的好處卻來自團隊共同應用領域驅動設計方法，並且將領域模型作為項目溝通的核心。這樣，團隊成員就有了一種公共語言，可以用來進行更充分的溝通，並確保圍繞軟件來進行溝通。他們將創建出一個與模型步調一致的清晰的實現，從而為應用程序的開發提供幫助。所有人都瞭解不同團隊的設計工作之間的互相聯繫，而且他們會一致將注意力集中在那些對組織最有價值、最與眾不同的特性的開發上。

領域驅動設計是一項艱巨的技術挑戰，但它也會帶來豐厚的回報，當大多數軟件項目開始僵化而成為遺留系統時，它卻為你敞開了機會的大門。

---

[1].這種表述指這是本書參考文獻中提到的圖書。——編者注

# 致謝

本書的創作歷時4年多，其間經歷了諸多工作形式的變化，在這個過程中很多人為我提供了幫助和支持。

感謝那些閱讀本書書稿並提出意見的人。沒有這些人的反饋意見，本書將不可能出版。其中有幾個團隊和一些人員對本書的評閱給予了特別的關注。由Russ Rufer和Tracy Bialek領導的硅谷模式小組（Silicon Valley Patterns Group）花費了幾周時間詳細審閱了本書完整的第一稿。由Ralph Johnson領導的伊利諾伊大學的閱讀小組也花費了幾周時間審閱了本書的第二稿。這些小組長期、精彩的討論對本書產生了深遠的影響。Kyle Brown和Martin Fowler提供了細緻入微的反饋意見和寶貴的建議，也給了我無價的精神支持（在我們坐在一起釣魚的時候）。Ward Cunningham的意見幫助我彌補了一些重大的缺陷。Alistair Cockburn在早期給了我很多鼓勵，並和Hilary Evans一起幫助我完成了整個出版過程。David Siegel和Eugene Wallingford幫助我避免了很多技術上的錯誤。Vibhu Mohindra和Vladimir Gitlevich不厭其煩地檢查了所有代碼示例。

Rob Mee看了我對一些素材所做的早期研究，並在我嘗試表達這種設計風格的時候與我進行了頭腦風暴活動，幫我產生了很多新的想法。他後來又與我一起仔細探討了後面的書稿。

本書在寫作過程中經歷了一次重大轉折，這完全歸功於Josh Kerievsky。他勸說我在寫作本書時借鑒「亞歷山大」模式[\[1\]](#)，後

來本書正是按這種方式組織的。在1999年PLoP會議臨近時的忙碌時刻，Josh還幫我收集第二部分的材料，首次將它們組織為更嚴密的形式。這些材料成了一粒種子，本書大部分後續內容都是圍繞這些內容創作的。

還要感謝Awad Faddoul，我有數百個小時坐在他的咖啡廳中寫作。咖啡廳寧靜優雅，窗外的湖面上總有片片風帆，我正是這樣才堅持寫下去。

此外還要感謝Martine Jousset、Richard Paselk和Ross Venables，他們拍攝了一些非常精美的照片，用來演示一些關鍵概念（參見本書後面的圖片說明）。

在構思本書之前，我必須先要形成我自己對軟件開發的看法和理解。這個過程得到了一些傑出人員的無私幫助，他們是我的良師益友。David Siegel、Eric Gold和Iseult White各自從不同方面幫助我形成了對軟件設計的思考方式。同時，Bruce Gordon、Richard Freyberg和Judith Segal博士也從不同角度幫助我找到了項目的成功之路。

我自己的觀念就是從那時的思想體系中自然而然發展形成的。有些內容我在正文中清楚地列了出來，並且在可能的地方標明瞭出處。還有些可能是十分基礎的知識，我甚至自己都沒有意識到它們對我產生了影響。

我的碩士論文導師Bala Subramaniam博士是我在數學建模方面的引路人，當時我們用數學建模來進行化學反應動力學方面的研究。雖說建模本身沒什麼稀奇，但那時的工作是引導我創作本書的一部分原因。

在更早之前，我的母親Carol和父親Gary對我思維模式的形成產生了很大影響。還有幾位特別值得一提的教師激發了我的興趣，幫助我打下堅實的基礎，在此感謝Dale Currier（我的高中數學老師）、Mary

Brown（我的高中英文寫作老師）和Josephine McGlamery（我上6年級時的自然科學老師）。

最後，感謝我的朋友和家人，以及Fernando De Leon，感謝他們一直以來給我的鼓勵。

---

[1].克裡斯托弗 „亞歷山大 ( Christopher Alexander )，1936年10月4日出生於奧地利的維也納，是一名建築師，以其設計理論和豐富的建築設計作品而聞名於世。亞歷山大認為，建築的使用者比建築師更清楚他們需要什麼，他創造並以實踐驗證了「模式語言」，建築模式語言賦予所有人設計並建造建築的能力。亞歷山大的代表作是《建築模式語言》，該書對計算機科學領域中的「設計模式」運動產生了巨大的影響。亞歷山大創立的增量、有機和連貫的設計理念也影響了「極限編程」運動。——編者注

## 第一部分 運用領域模型



上面這張圖是18世紀中國描繪的世界地圖。圖中央最大的部分是中國，其周圍散佈著其他國家，但這些國家只是草草地表示了一下。

這是適用於當時中國社會的世界模型，它意在關注中國自身。然而，這幅地圖所呈現的世界觀對於處理外交事務並無助益。當然，它對現代中國也毫無用處。地圖就是模型，而模型被用來描繪人們所關注的現實或想法的某個方面。模型是一種簡化。它是對現實的解釋——把與解決問題密切相關的方面抽像出來，而忽略無關的細節。

每個軟件程序是為了執行用戶的某項活動，或是滿足用戶的某種需求。這些用戶應用軟件的問題區域就是軟件的領域。一些領域涉及物質世界，例如，機票預訂程序的領域中包括飛機乘客在內。有些領域則是無形的，例如，會計程序的金融領域。軟件領域一般與計算機關係不大，當然也有例外，例如，源代碼控制系統的領域就是軟件開發本身。

為了創建真正能為用戶活動所用的軟件，開發團隊必須運用一整套與這些活動有關的知識體系。所需知識的廣度可能令人望而生畏，龐大而複雜的信息也可能超乎想像。模型正是解決此類信息超載問題的工具。模型這種知識形式對知識進行了選擇性的簡化和有意的結構化。適當的模型可以使人理解信息的意義，並專注於問題。

領域模型並非某種特殊的圖，而是這種圖所要傳達的思想。它絕不單單是領域專家頭腦中的知識，而是對這類知識嚴格的組織且有選擇的抽像。圖可以表示和傳達一種模型，同樣，精心書寫的代碼或文字也能達到同樣的目的。

領域建模並不是要盡可能建立一個符合「現實」的模型。即使是对具體、真實世界中的事物進行建模，所得到的模型也不過是對事物的一種模擬。它也不單單是為了實現某種目的而構造出來的軟件機制。建模更像是製作電影——出於某種目的而概括地反映現實。即使是一部紀錄片也不會原封不動地展現真實生活。就如同電影製片人講

述故事或闡明觀點時，他們會選擇素材，並以一種特殊方式將它們呈現給觀眾，領域建模人員也會依據模型的作用來選擇具體的模型。

### **模型在領域驅動設計中的作用**

在領域驅動的設計中，3個基本用途決定了模型的選擇。

(1) 模型和設計的核心互相影響。正是模型與實現之間的緊密聯繫才使模型變得有用，並確保我們在模型中所進行的分析能夠轉化為最終產品（即一個可運行的程序）。模型與實現之間的這種緊密結合在維護和後續開發期間也會很有用，因為我們可以基於對模型的理解來解釋代碼。（參見第3章）

(2) 模型是團隊所有成員使用的通用語言的中樞。由於模型與實現之間的關聯，開發人員可以使用該語言來討論程序。他們可以在無需翻譯的情況下與領域專家進行溝通。而且，由於該語言是基於模型的，因此我們可藉助自然語言對模型本身進行精化。（參見第2章）

(3) 模型是濃縮的知識。模型是團隊一致認同的領域知識的組織方式和重要元素的區分方式。透過我們如何選擇術語、分解概念以及將概念聯繫起來，模型記錄了我們看待領域的方式。當開發人員和領域專家在將信息組織為模型時，這一共同的語言（模型）能夠促使他們高效地協作。模型與實現之間的緊密結合使來自軟件早期版本的經驗可以作為反饋應用到建模過程中。（參見第1章）

接下來的3章分別考查上述3種基本用途的意義和價值，以及它們之間的關聯方式。遵循這些原則使用模型可以很好地支持具有豐富功能的軟件的開發，否則就需要耗費大規模投資進行專門開發。

### **軟件的核心**

軟件的核心是其為用戶解決領域相關的問題的能力。所有其他特性，不管有多麼重要，都要服務於這個基本目的。當領域很複雜時，這是一項艱巨的任務，要求高水平技術人員的共同努力。開發人員必

須鑽研領域以獲取業務知識。他們必須磨礪其建模技巧，並精通領域設計。

然而，在大多數軟件項目中，這些問題並未引起足夠的重視。大部分有才能的開發人員對學習與他們的工作領域有關的知識不感興趣，更不會下力氣去擴展自己的領域建模技巧。技術人員喜歡那些能夠提高其技能的可量化問題。領域工作很繁雜，而且要求掌握很多複雜的新知識，而這些新知識看似對提高計算機科學家的能力並無裨益。

相反，技術人才更願意從事情細的框架工作，試圖用技術來解決領域問題。他們把學習領域知識和領域建模的工作留給別人去做。軟件核心的複雜性需要我們直接去面對和解決，如果不這樣做，則可能導致工作重點的偏離。

在一次電視訪談節目中，喜劇演員John Cleese講述了電影《巨蟒和聖盃》（Monty Python and the Holy Grail）在拍攝期間發生的一個小故事。有一幕他們反覆拍了很多次，但就是感覺不夠滑稽。最後，他停下來，與另一位喜劇演員Michael Palin（該幕中的另一位演員）商量了一下，他們決定稍微改變一下。隨後又拍了一次，終於令他們滿意了，於是收工。

第二天早上，Cleese觀看了剪輯人員為前一天工作所做的粗剪。到了那個令他們頗費周章的場景時，Cleese發現剪輯人員竟然使用了先前拍攝的一個鏡頭，影片到這裡又變得不滑稽了。

他問剪輯人員為什麼沒有按要求使用最後拍的那個鏡頭，剪輯人員回答說：「那個鏡頭不能用，因為有人闖入了鏡頭。」Cleese連看了兩遍，仍未發現有什麼不妥。最後，剪輯人員將影片暫停，並指出在屏幕邊緣有一隻一閃而過的大衣袖子。

影片的剪輯人員專注於準確完成自己的工作。他擔心其他看到這部電影的剪輯人員會給他挑錯。在這個過程中，鏡頭的核心作用被忽略了（「The Late Late Show with Craig Kilborn」，CBS，2001年9月）。

幸運的是，該劇的導演很懂喜劇，他最終使用了那個鏡頭。同樣，在一個團隊中，反映了對領域深層次理解的模型開發有時也會在混亂中迷失方向，此時，理解領域核心的領導者能夠將軟件項目帶回到正確的軌道上來。

本書將展示領域開發中蘊藏的巨大機會，它能夠培養精湛的設計技巧。大多數混亂的軟件領域其實是一項充滿樂趣的技術挑戰。事實上，在許多科學領域中，「複雜性」都是當前最熱門的話題之一，因為研究人員都在想辦法解決真實世界中的複雜性。軟件開發人員在面對尚未規範的複雜領域時，也會有同樣的期望。創建一個克服這些複雜性的易懂模型會帶來巨大的成就感。

開發人員可以採用一些系統性的思考方法來透徹地理解領域並開發出有效的模型。還有一些設計技巧可以使毫無頭緒的軟件應用變得井井有條。掌握這些技能可以令開發人員的價值倍增，即使是在一個最初不熟悉的領域中也是如此。

# 第1章 消化知識

幾年前，我著手設計一個用於設計印製電路板（PCB）的專用軟件工具。但有一個問題，我對電子硬件一無所知。當然，我也曾拜訪過一些PCB設計師，但用不了3分鐘，他們就令我暈頭轉向。如何才能瞭解足夠多的知識，以便開始編寫這個軟件呢？當然，我並不打算在交付期限到來之前成為電子工程師。

我們試著讓PCB設計師說明軟件具體應該做些什麼，但我們錯了。雖然他們是優秀的電路設計師，但軟件知識卻太有限了，往往只知道如何讀取一個ASCII文件、對它排序，然後添加一些註釋並將它寫回文件中，再生成一個報告。這些知識顯然無法幫助他們大幅度提高效率。

最初的幾次會面令人氣餒，但我們在他們要求的報告中也看到了一絲希望。這些報告中總是涉及net這個詞以及與其相關的各種細節。在這個領域中，net實質上是一種導線，它可以連接PCB上任意數量的元件，並向它連接的所有元件傳遞電子信號。這樣，我們就得到了領域模型的第一個元素，如圖1-1所示。



圖1-1

就這樣，我們一邊討論所需的軟件功能，一邊開始畫圖。我使用一種非正式的、稍加變化的對象交互圖來走查[1]各種場景，如圖1-2

所示。

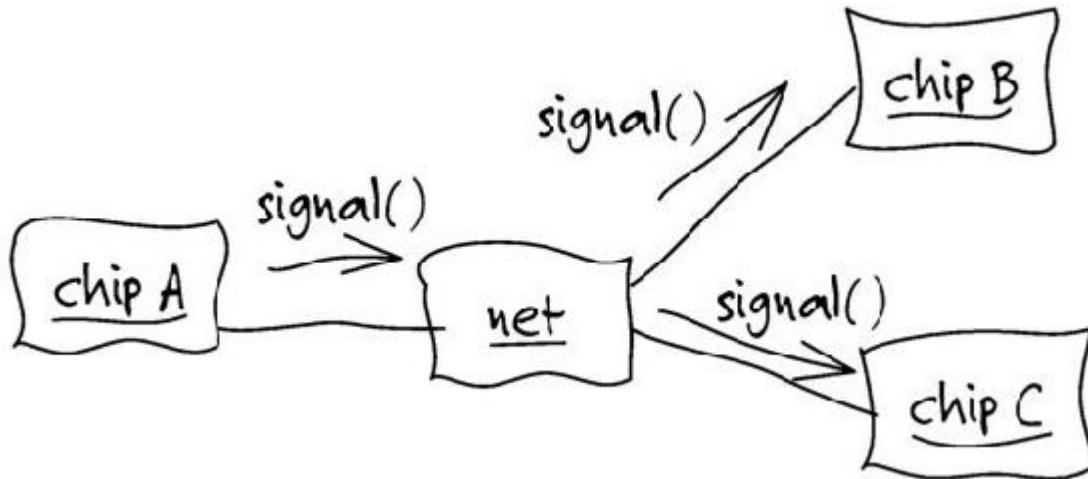


圖1-2

PCB專家1：元件不一定就是芯片（chip）。

開發人員（我）：那它們是不是隻應該叫做「元件」？

專家1：我們將它們稱作「元件實例」（component instance）。

相同的元件可能有很多。

專家2：他把「net」畫成和元件實例一樣的框了。

專家1：他沒有使用我們的符號。我猜想，他要把每一項都畫成方框。

開發人員：很抱歉，是這樣的。我想我最好對這個符號稍加解釋。

他們不斷地糾正我的錯誤，在這個過程中我開始學習他們的知識。我們共同消除了術語上的不一致和歧義，也消除了他們在技術觀點上的分歧，在這個過程中，他們也得到了學習。他們的解釋更準確和一致了，然後我們開始共同開發一個模型。

專家1：只說一個信號到達一個ref-des是不夠明確的，我們必須知道信號到達了哪個引腳。

開發人員：什麼是ref-des？

專家2：它就是一個元件實例。我們用的一個專門工具中用ref-des這個名稱。

專家1：總之，net將一個實例的某個引腳與另一個實例的某個引腳相連。

開發人員：一個引腳是不是隻屬於一個元件實例，而且只與一個net相連？

專家1：對，是這樣。

專家2：還有，每個net都有一個拓撲結構，也就是電路的佈局，它決定了net內部各元件的連接方式。

開發人員：嗯，這樣畫如何（如圖1-3所示）？

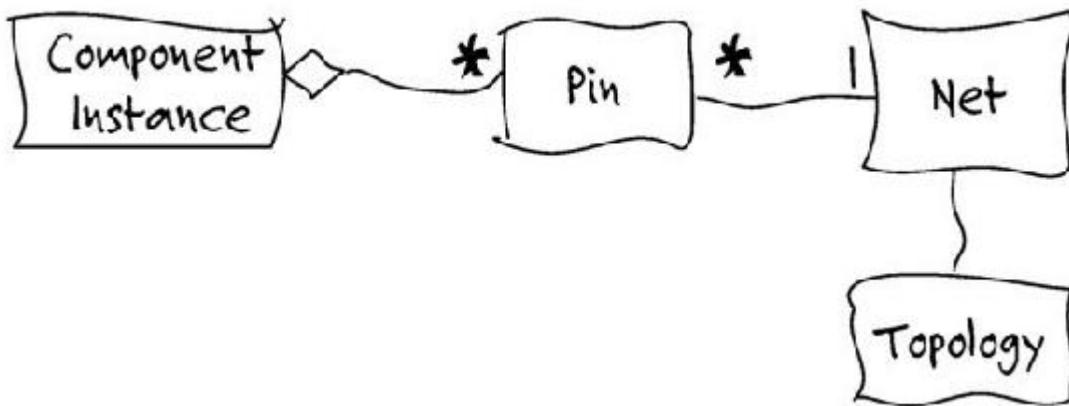


圖1-3

為了讓討論更集中，接下來的一段時間我們探討了一個特定的功能：探針仿真（probe simulation）。探針仿真跟蹤信號的傳播，以便檢測在設計中可能出現特定類型問題的位罷。

開發人員：現在我已經明白了Net是如何將信號傳播給它所連接的所有Pin的，但如何將信號傳送得更遠呢？這與拓撲結構（topology）有關係嗎？

專家2：沒有，是元件推送信號前進。

開發人員：我們肯定無法對芯片的內部行為建模，因為這太複雜了。

專家2：我們不必這樣做。可以使用一種簡化形式。只需列出通過元件可從某些Pin將信號推送到其他引腳即可。

開發人員：類似於這樣嗎？

（經過反覆的嘗試和修改，我們終於共同繪製出了一個草圖，如圖1-4所示。）

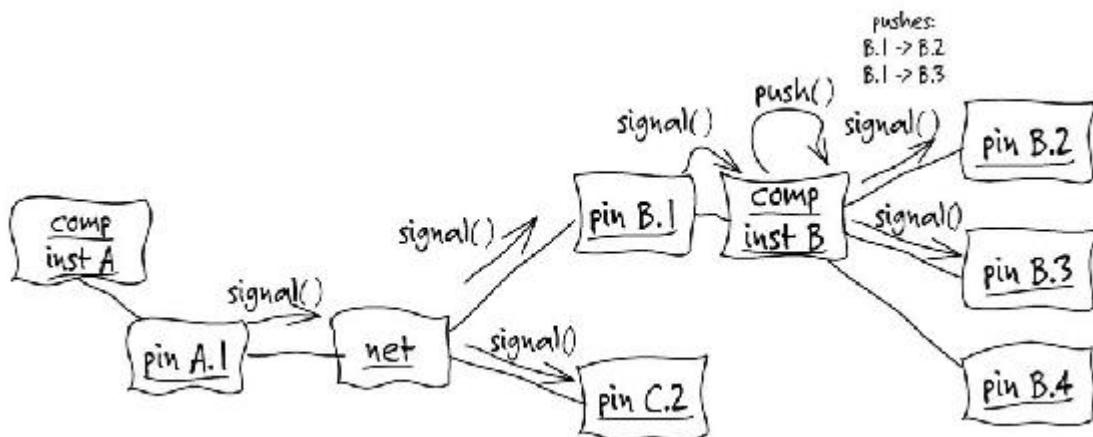


圖1-4

開發人員：但你想從這種計算中知道什麼呢？

專家2：我們要查找較長的信號延遲，也就是說，查找超過2或3跳的信號路徑。這是一條經驗法則。如果路徑太長，信號可能無法在時鐘週期內到達。

開發人員：超過3跳.....這麼說我們需要計算路徑長度。那麼怎樣算作一跳呢？

專家2：信號每通過一個Net，就稱為1跳。

開發人員：那麼我們可以沿著電路來計算跳數，每遇到一個net，跳數就加1，如圖1-5所示。

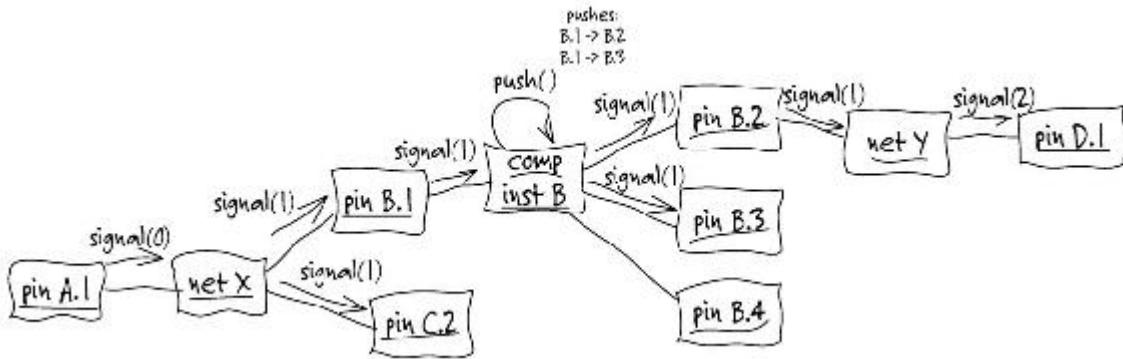


圖1-5

開發人員：現在我唯一不明白的地方是「推動」是從哪裡來的。是否每個元件實例都需要存儲該數據？

專家2：一個元件的所有實例的推動行為都是相同的。

開發人員：那麼元件的類型決定了推動行為，而每個實例的推動行為都是相同的（如圖1-6所示）？

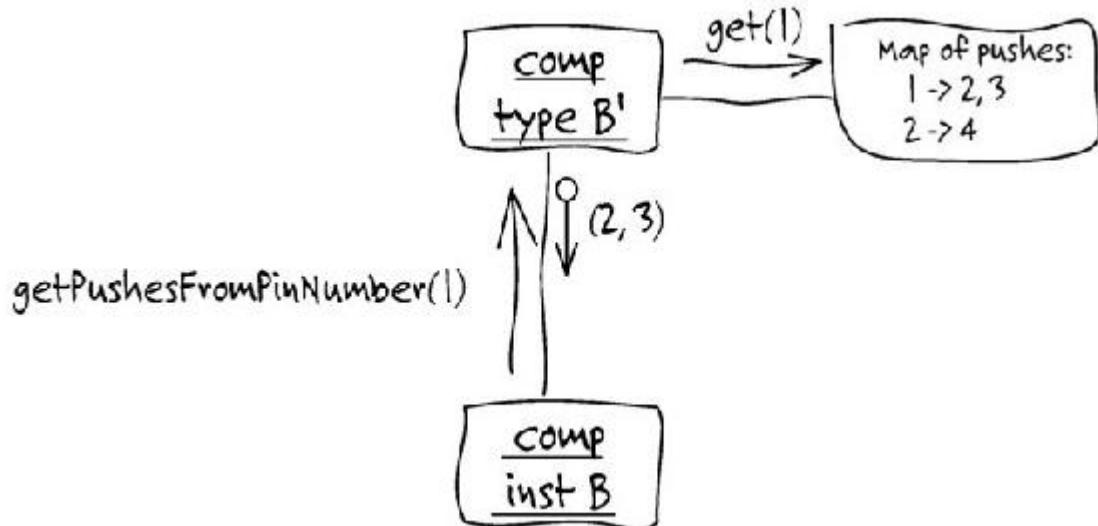


圖1-6

專家2：這個圖的意思我沒完全明白，但我猜想每個元件存儲的推動行為就差不多是這樣的吧。

開發人員：抱歉，這個地方我可能問得有點過細了。我只是想考慮得全面一些……現在，拓撲結構對它有什麼影響嗎？

專家1：拓撲結構不影響探針仿真。

開發人員：那麼可以暫不考慮它，是嗎？等用到這些特性時再回來討論它。

就這樣，我們的討論一直進行下去（其中遇到的困難比上面顯示的多得多）。我們一邊進行「頭腦風暴」式的討論，一邊對模型進行精化，邊提問邊回答。隨著我對領域理解的加深，以及他們對模型在解決方案中作用的理解的加深，模型不斷發展。圖1-7顯示了那個早期模型的類圖。

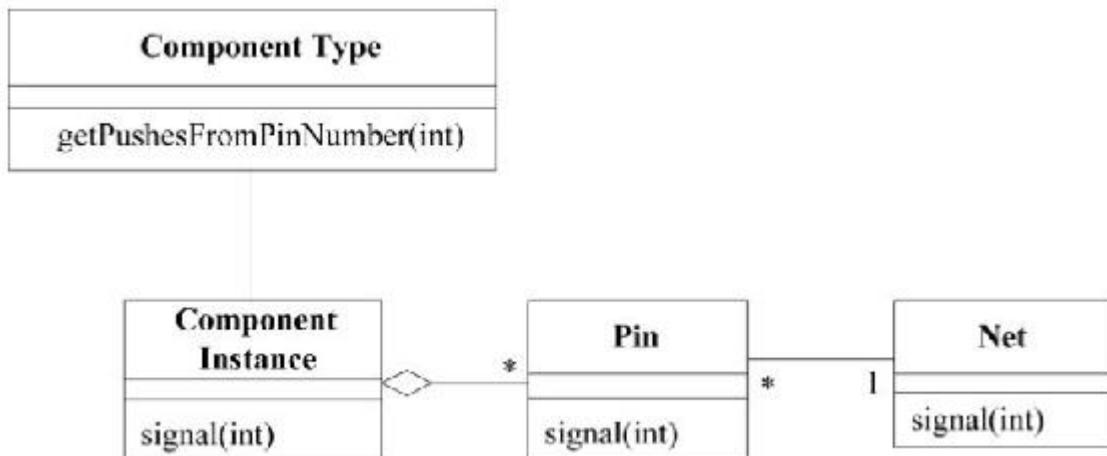


圖1-7

隨後，我們又拿出一部分工作時間進行了幾輪這樣的討論，我覺得自己已經理解了足夠多的知識，可以試著編寫一些代碼了。我寫了一個非常簡單的原型，並用一個自動測試框架來測試它。我避開了所有的基礎設施。這個原型沒有持久化機制，也沒有用戶界面（UI）。這樣我就可以專注於代碼的行為。只不過幾天我就能夠演示簡單的探針仿真了。雖然它使用的是虛擬數據，而且向控制檯輸出的是原始文本，但確實是使用Java對像對路徑長度執行實際的計算。這些Java對像所反映的模型正是我和領域專家們一起開發出來的。

這個具體的原型使得領域專家們更清楚地理解了模型的含義，以及它與最終軟件之間的聯繫。從那時起，我們的模型討論越來越具有互動性了，因為他們可以看到我如何將新學到的知識融合到模型中，然後反映到軟件上。他們也可以從原型得到具體的反饋，從而印證自己的想法。

模型中包含與我們要解決的問題有關的PCB領域知識，這些知識遠遠比我們在這裡演示的複雜。模型將很多同義詞和語言描寫中的微小差別做了統一，並排除了數百條與問題沒有直接關係的事實（雖然工程師們都理解這些事實），如元件的實際數字特性。像我這樣的軟件專業人員看到這張圖後，幾分鐘內就能明白軟件是做什麼的。這個模型就相當於一個框架，開發人員可以藉助它來組織新的信息並更快地學習，從而更準確地判斷哪些部分重要，哪些部分不重要，並更好地與PCB工程師進行溝通。

當PCB工程師提出新的功能需求時，我就讓他們帶我走查對像交互的場景。當模型對像無法清楚地表達某個重要場景時，我們就通過頭腦風暴活動創建新的模型對像或者修改原有的模型對象，並消化理解這些模型對像中的知識。在我們精化模型的過程中，代碼也隨之一步步演進。幾個月後，PCB工程師們得到了一個遠遠超乎他們期望的功能豐富的工具。

## 1.1 有效建模的要素

以下幾方面因素促使上述案例得以成功。

(1) 模型和實現的綁定。最初的原型雖然簡陋，但它在模型與實現之間建立了早期鏈接，而且在所有後續的迭代中我們一直在維護該鏈接。

(2) 建立了一種基於模型的語言。最初，工程師們不得不向我解釋基本的PCB問題，而我也必須向他們解釋類圖的含義。但隨著項目的進展，雙方都能夠直接使用模型中的術語，並將它們組織為符合模型結構的語句，而且無需翻譯即可理解互相要表達的意思。

(3) 開發一個蘊含豐富知識的模型。對像具有行為和強制性規則。模型並不僅僅是一種數據模式，它還是解決複雜問題不可或缺的部分。模型包含各種類型的知識。

(4) 提煉模型。在模型日趨完整的過程中，重要的概念不斷被添加到模型中，但同樣重要的是，不再使用的或不重要的概念則從模型中被移除。當一個不需要的概念與一個需要的概念有關聯時，則把重要的概念提取到一個新模型中，其他那些不要的概念就可以丟棄了。

(5) 頭腦風暴和實驗。語言和草圖，再加上頭腦風暴活動，將我們的討論變成「模型實驗室」，在這些討論中可以演示、嘗試和判斷上百種變化。當團隊走查場景時，口頭表達本身就可以作為所提議的模型的可行性測試，因為人們聽到口頭表達後，就能立即分辨出它是表達得清楚、簡捷，還是表達得很笨拙。

正是頭腦風暴和大量實驗的創造力才使我們找到了一個富含知識的模型並對它進行提煉，在這個過程中，基於模型的語言提供了很大幫助，而且貫穿整個實現過程中的反饋閉環也對模型起到了「訓練」作用。這種知識消化將團隊的知識轉化為有價值的模型。

## 1.2 知識消化

金融分析師要消化理解的內容是數字。他們篩選大量的詳細數字，對其進行組合和重組以便尋求潛在的意義，查找可以產生重要影響的簡單表示方式——一種可用作金融決策基礎的理解。

高效的領域建模人員是知識的消化者。他們在大量信息中探尋有用的部分。他們不斷嘗試各種信息組織方式，努力尋找對大量信息有意義的簡單視圖。很多模型在嘗試後被放棄或改造。只有找到一組適用於所有細節的抽象概念後，工作才算成功。這一精華嚴謹地表示了所發現的最為相關的知識。

知識消化並非一項孤立的活動，它一般是在開發人員的領導下，由開發人員與領域專家組成的團隊來共同協作。他們共同收集信息，並通過消化而將它組織為有用的形式。信息的原始資料來自領域專家頭腦中的知識、現有系統的用戶，以及技術團隊以前在相關遺留系統或同領域的其他項目中積累的經驗。信息的形式也多種多樣，有可能是為項目編寫的文檔，有可能是業務中使用的文件，也有可能來自大量的討論。早期版本或原型將經驗反饋給團隊，然後團隊對一些解釋做出修改。

在傳統的瀑布方法中，業務專家與分析員進行討論，分析員消化理解這些知識後，對其進行抽象並將結果傳遞給程序員，再由程序員編寫軟件代碼。由於這種方法完全沒有反饋，因此總是失敗。分析員全權負責創建模型，但他們創建的模型只是基於業務專家的意見。他們既沒有向程序員學習的機會，也得不到早期軟件版本的經驗。知識只是朝一個方向流動，而且不會累積。

有些項目使用了迭代過程，但由於沒有對知識進行抽象而無法建立起知識體系。開發人員聽專家們描述某項所需的特性，然後開始構建它。他們將結果展示給專家，並詢問接下來做什麼。如果程序員願意進行重構，則能夠保持軟件足夠整潔，以便繼續擴展它；但如果程序員對領域不感興趣，則他們只會瞭解程序應該執行的功能，而不去瞭解它背後的原理。雖然這樣也能開發出可用的軟件，但項目永遠也不會從原有特性中自然地擴展出強大的新特性。

好的程序員會自然而然地抽像並開發出一個可以完成更多工作的模型。但如果在建模時只是技術人員唱獨角戲，而沒有領域專家的協作，那麼得到的概念將是很幼稚的。使用這些膚淺知識開發出來的軟件只能做基本工作，而無法充分反映出領域專家的思考方式。

在團隊所有成員一起消化理解模型的過程中，他們之間的交互也會發生變化。領域模型的不斷精化迫使開發人員學習重要的業務原理，而不是機械地進行功能開發。領域專家被迫提煉自己已知道的重要知識的過程往往也是完善其自身理解的過程，而且他們會漸漸理解軟件項目所必需的概念嚴謹性。

所有這些因素都促使團隊成員成為更合格的知識消化者。他們對知識去粗取精。他們將模型重塑為更有用的形式。由於分析員和程序員將自己的知識輸入到了模型中，因此模型的組織更嚴密，抽像也更為整潔，從而為實現提供了更大支持。同時，由於領域專家也將他們的知識輸入到了模型中，因此模型反映了業務的深層次知識，而且真正的業務原則得以抽像。

模型在不斷改進的同時，也成為組織項目信息流的工具。模型聚焦於需求分析。它與編程和設計緊密交互。它通過良性循環加深團隊成員對領域的理解，使他們更透徹地理解模型，並對其進一步精化。模型永遠都不會是完美的，因為它是一個不斷演化完善的過程。模型對理解領域必須是切實可用的。它們必須非常精確，以便使應用程序易於實現和理解。

## **1.3 持續學習**

當開始編寫軟件時，其實我們所知甚少。項目知識零散地分散在很多人和文檔中，其中夾雜著其他一些無關信息，因此我們甚至不知

道哪些知識是真正需要的知識。看起來沒什麼技術難度的領域很可能是一種錯覺——我們並沒意識到不知道的東西究竟有多少。這種無知往往會導致我們做出錯誤的假設。

同時，所有項目都會丟失知識。已經學到了一些知識的人可能幹別的事去了。團隊可能由於重組而被拆散，這導致知識又重新分散開。被外包出去的關鍵子系統可能只交回了代碼，而不會將知識傳遞回來。而且當使用典型的設計方法時，代碼和文檔不會以一種有用的形式表示出這些來之不易的知識，因此一旦由於某種原因人們沒有口頭傳遞知識，那麼知識就丟失了。

高效率的團隊需要有意識地積累知識，並持續學習[Kerievsky 2003]。對於開發人員來說，這意味著既要完善技術知識，也要培養一般的領域建模技巧（如本書中所講的那些技巧）。但這也包括認真學習他們正在從事的特定領域的知識。

那些善於自學的團隊成員會成為團隊的中堅力量，涉及最關鍵領域的開發任務要靠他們來攻克（有關這方面的更多內容，參見第15章）。這個核心團隊頭腦中積累的知識使他們成為更高效的知識消化者。

讀到這裡，請先停一下來問自己一個問題。你是否學到了一些PCB設計知識？雖然這個示例只對該領域作了些表面處理，但當討論領域模型時，仍會學到一些知識。我學習了大量知識，但並沒有學習如何成為一名PCB工程師，因為這不是我的目的。我的目的是學會與PCB專家溝通，理解與應用有關的主要概念，並學會檢查所構建的內容是否合理。

事實上，我們的團隊最終發現探針仿真並不是一項重要的開發任務，因此最後徹底放棄了這個功能。連同它一起刪除的還有模型中的一些部分，這些部分只是幫助我們理解如何通過元件推動信號以及如

何計算跳數。這樣，應用程序的核心就轉移到了別處，而且模型也隨之改變，將新的重點作為核心。在這個過程中，領域專家們也學到了很多東西，而且更加清楚地理解了應用程序的目標（第15章會更深入地討論這些問題）。

儘管如此，那些早期工作還是非常重要的。關鍵的模型元素被保留下來，而更重要的是，早期工作啟動了知識消化的過程，這使得所有後續工作更加高效：團隊成員、開發人員和領域專家等都學到了知識，他們開始使用一種公共的語言，而且形成了貫穿整個實現過程的反饋閉環。這樣，一個發現之旅悄然開始了。

## 1.4 知識豐富的設計

通過像PCB示例這樣的模型獲得的知識遠遠不只是「發現名詞」。業務活動和規則如同所涉及的實體一樣，都是領域的核心，任何領域都有各種類別的概念。知識消化所產生的模型能夠反映出對知識的深層理解。在模型發生改變的同時，開發人員對實現進行重構，以便反映出模型的變化，這樣，新知識就被合併到應用程序中了。

當我們的建模不再侷限於尋找實體和值對像時，我們才能充分吸取知識，因為業務規則之間可能會存在不一致。領域專家在反覆研究所有規則、解決規則之間的矛盾以及以常識來彌補規則的不足等一系列工作中，往往不會意識到他們的思考過程有多麼複雜。軟件是無法完成這一工作的。正是通過與軟件專家緊密協作來消化知識的過程才使得規則得以澄清和充實，並消除規則之間的矛盾以及刪除一些無用規則。

### 示例 提取一個隱藏的概念

我們從一個非常簡單的領域模型開始學習，基於此模型的應用程序用來預訂一艘船在一次航程中要運載的貨物，如圖1-8所示。

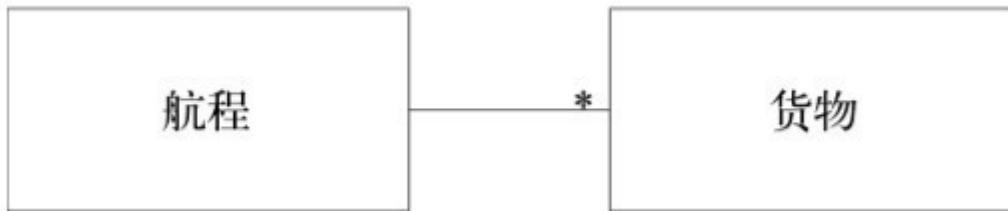


圖1-8

我們規定這個應用程序的任務是將每件貨物（Cargo）與一次航程（Voyage）關聯起來，記錄並跟蹤這種關係。現在看來一切都還算簡單。應用程序代碼中可能會有一個像下面這樣的方法：

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

    return confirmation;
}
```

由於總會有人在最後一刻取消訂單，因此航運業的一般做法是接受比其運載能力多一些的貨物。這稱為「超訂」。有時使用一個簡單的容量百分比來表示，如預訂110%的載貨量。有時則採用複雜的規則——主要客戶或特定種類的貨物優先。

這是航運領域的一個基本策略，從事航運業的業務人員都知道它，但在軟件團隊中可能不是所有技術人員都知道這條規則。

需求文檔中包含下面這句話：

允許10%的超訂。

現在，類圖就應該像圖1-9這樣，代碼如下：

```

public int makeBooking(Cargo cargo, Voyage voyage) {
    double maxBooking = voyage.capacity() * 1.1;
    if ((voyage.bookedCargoSize() + cargo.size()) > maxBooking)
        return -1;
    int confirmation = orderConfirmationSequence.next();
    voyage.addCargo(cargo, confirmation);
    return confirmation;
}

```

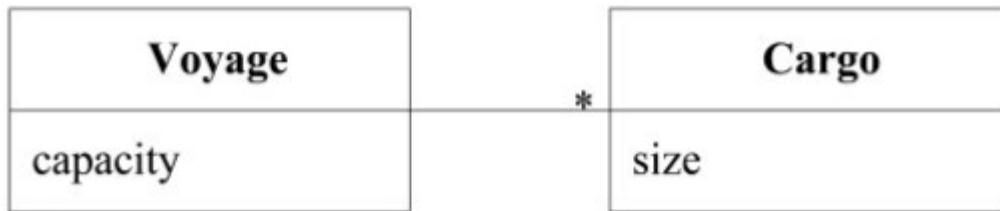


圖1-9

現在，一條重要的業務規則被隱藏在上面這段方法代碼的一個衛語句[2]中。第4章將介紹LAYERED ARCHITECTURE，它會幫助我們將超訂規則轉移到領域對像中，但現在我們主要考慮如何把這條規則更清楚地表達出來，並讓項目中的每個人都能瞭解到它。這將使我們得到一個類似的解決方案。

(1) 如果業務規則如上述代碼所寫，不可能有業務專家會通過閱讀這段代碼來檢驗規則，即使在開發人員的幫助下也無法完成。

(2) 非業務的技術人員很難將需求文本與代碼聯繫起來。

如果規則更複雜，情況將更糟。

我們可以改變一下設計來更好地捕獲這個知識。超訂規則是一個策略，如圖1-10所示。策略（policy）其實是STRATEGY模式[3] [Gamma et al.1995]的別名。我們知道，使用STRATEGY的動機一般是為了替換不同的規則，雖然在這裡並不需要這麼做。但我們要獲取的概念的確符合策略的含義，這在領域驅動設計中是同等重要的動機（參見第12章）。

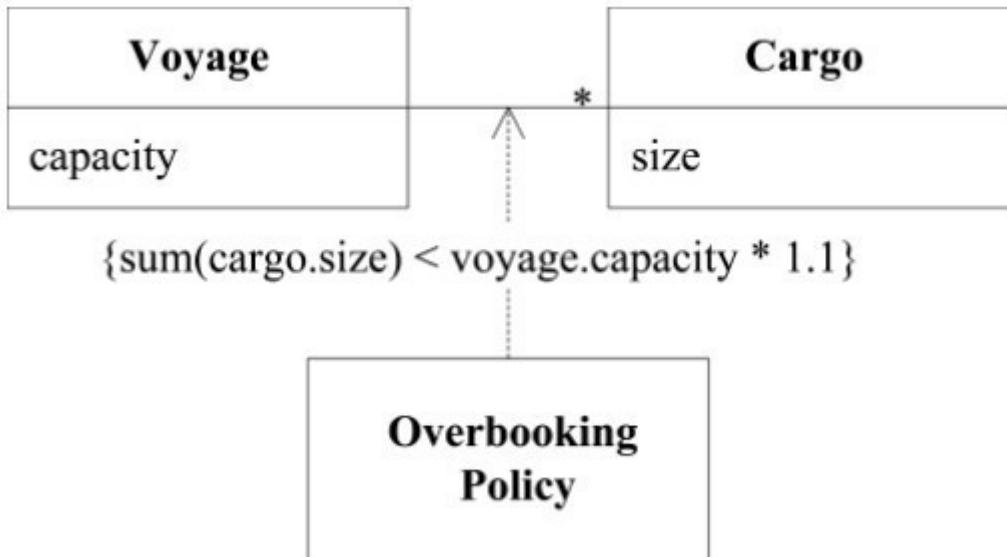


圖1-10

修改後的代碼如下：

```

public int makeBooking(Cargo cargo, Voyage voyage) {
    if (!overbookingpolicy.isallowed(cargo, voyage)) return -1;
    int confirmation = orderConfirmationSequence.next();
    voyage.addCargo(cargo, confirmation);
    return confirmation;
}
  
```

新的Overbooking Policy類包含以下方法：

```

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

現在所有人都清楚超訂是一個獨特的策略，而且超訂規則的實現即明確又獨立。

現在，我並不建議將這樣的精細設計應用到領域的每個細節中。第15章將深入闡述如何關注重點以及如何隔離其他問題或使這些問題

最小化。這個例子的目的是說明領域模型和相應的設計可用來保護和共享知識。更明確的設計具有以下優點：

(1) 為了實現更明確的設計，程序員和其他各位相關人員都必須理解超訂的本質，明白它是一個明確且重要的業務規則，而不只是一個不起眼的計算。

(2) 程序員可以向業務專家展示技術工件，甚至是代碼，但應該是領域專家（在程序員指導下）可以理解的，以便形成反饋閉環。

## 1.5 深層模型

有用的模型很少停留在表面。隨著對領域和應用程序需求的理解逐步加深，我們往往會丟棄那些最初看起來很重要的表面元素，或者切換它們的角度。這時，一些開始時不可能發現的巧妙抽象就會漸漸浮出水面，而它們恰恰切中問題的要害。

前面的例子大體上是基於一個集裝箱航運項目，這是本書列舉的幾個項目之一，本書還有幾個示例會引用這個項目。本書所舉的示例都很簡單，即使不是航運專家也能理解它們。但在一個需要團隊成員持續學習的真實項目中，要想建立實用且清晰的模型則要求團隊成員既精通領域知識，也要精通建模技術。

在這個項目中，由於航運從預訂貨運開始，因此我們開發了一個能夠描述貨物和運貨航線等事物的模型。這是必要且有用的，但領域專家卻不買賬。他們有自己的考慮業務的方式，這種方式是我們沒有考慮到的。

最後，在經過幾個月的知識消化後，我們知道貨物的處理主要是由轉包商或公司中的操作人員完成的，這包括實際的裝貨、卸貨和運貨。航運專家的觀點是，各部分之間存在一系列的責任傳遞。法律責

任和執行責任的傳遞由一個過程控制—從託運人傳遞到某個本地運輸商，再從這家運輸商傳遞到另一家運輸商，最後到達收貨人。通常，在一些重要的步驟中，貨物停放在倉庫裡。在其他時間裡，貨物則是通過複雜的物理步驟來運輸，而這些與航運公司的業務決策無關。在處理航線的物流之前，必須先確定諸如提單等法律文件以及支付流程。

對航運業務有了更深刻的認識後，我們並沒有刪除Itinerary（航線）對象，但模型發生了巨大改變。我們對航運業務的認識從「集裝箱在各個地點之間的運輸」轉變為「運貨責任在各個實體之間的傳遞」。處理這些責任傳遞的特性不再是一些附屬於裝貨作業的次要特性，而是由一個獨立的模型來提供支持，這個模型正是在理解了作業與責任之間的重要關係之後開發出來的。

知識消化是一種探索，它永無止境。

## 第2章 交流與語言的使用

領域模型可成為軟件項目通用語言的核心。該模型是一組得自於項目人員頭腦中的概念，以及反映了領域深層含義的術語和關係。這些術語和相互關係提供了模型語言的語義，雖然語言是為領域量身定製的，但就技術開發而言，其依然足夠精確。正是這條至關重要的紐帶，將模型與開發活動結合在一起，並使模型與代碼緊密綁定。

這種基於模型的交流並不侷限於UML（統一建模語言）圖。為了最有效地使用模型，需要充分利用各種交流手段。基於模型的交流提高了書面文檔的效用，也提高了敏捷過程中再度強調的非正式圖表和交談的效用。它還通過代碼本身及對應的測試促進了交流。

在項目中，語言的使用很微妙，但卻至關重要.....

### 2.1 模式：UBIQUITOUS LANGUAGE

首先寫下一個句子，  
然後將它分成小段，  
再將它們打亂並重新排序。  
彷彿是巧合一樣，  
短語的順序對意思完全沒有影響。

——Lewis Carroll, 「Poeta Fit, Non Nascitur」

要想創建一種靈活的、蘊含豐富知識的設計，需要一種通用的、共享的團隊語言，以及對語言不斷的試驗——然而，軟件項目上很少出現這樣的試驗。

雖然領域專家對軟件開發的技術術語所知有限，但他們能熟練使用自己領域的術語——可能還具有各種不同的風格。另一方面，開發人員可能會用一些描述性的、功能性的術語來理解和討論系統，而這些術語並不具備領域專家的語言所要傳達的意思。或者，開發人員可能會創建一些用於支持設計的抽象，但領域專家無法理解這些抽象。負責處理問題不同部分的開發人員可能會開發出各自不同的設計概念以及描述領域的方式。

由於語言上存在鴻溝，領域專家們只能模糊地描述他們想要的東西。開發人員雖然努力去理解一個自己不熟悉的領域，但也只能形成模糊的認識。雖然少數團隊成員會設法掌握這兩種語言，但他們會變成信息流的瓶頸，並且他們的翻譯也不準確。

在一個沒有公共語言的項目上，開發人員不得不為領域專家做翻譯。而領域專家需要充當開發人員與其他領域專家之間的翻譯。甚至開發人員之間還需要互相翻譯。這些翻譯使模型概念變得混淆，而這會導致有害的代碼重構。這種間接的溝通掩蓋了分裂的形成——不同的團隊成員使用不同的術語而尚不自知。由於軟件的各個部分不能夠渾然一體，因此這就導致無法開發出可靠的軟件（參見第14章）。翻譯工作導致各類促進深入理解模型的知識和想法無法結合到一起。

如果語言支離破碎，項目必將遭遇嚴重問題。領域專家使用他們自己的術語，而技術團隊所使用的語言則經過調整，以便從設計角度討論領域。

日常討論所使用的術語與代碼（軟件項目的最重要產品）中使用的術語不一致。甚至同一個人在講話和寫東西時使用的語言也不一致，這導致的後果是，對領域的深刻表述常常稍縱即逝，根本無法記錄到代碼或文檔中。

翻譯使得溝通不暢，並削弱了知識消化。

然而任何一方的語言都不能成為公共語言，因為它們無法滿足所有的需求。

所有翻譯的開銷，連帶著誤解的風險，成本實在太高了。項目需要一種公共語言，這種語言要比所有語言的最小公分母健壯得多。通過團隊的一致努力，領域模型可以成為這種公共語言的核心，同時將團隊溝通與軟件實現緊密聯繫到一起。該語言將存在於團隊工作中的方方面面。

**UBIQUITOUS LANGUAGE**（通用語言）的詞彙包括類和主要操作的名稱。語言中的術語，有些用來討論模型中已經明確的規則，還有一些則來自施加於模型上的高級組織原則（如第14章和第16章要討論的**CONTEXT MAP**和大型結構）。最後，團隊常常應用於領域模型的模式名稱也使這種語言更為豐富。

模型之間的關係成為所有語言都具有的組合規則。詞和短語的意義反映了模型的語義。

開發人員應該使用基於模型的語言來描述系統中的工件、任務和功能。這個模型應該為開發人員和領域專家提供一種用於相互交流的語言，而且領域專家還應該使用這種語言來討論需求、開發計劃和特性。語言使用得越普遍，理解進行得就越順暢。

至少，我們應該將它作為目標。但最初，模型可能不太好，因此無法很好地履行這些職責。它可能不會像領域的專業術語那樣具有豐富的語義。但我們又不能直接使用那些術語，因為它們有歧義和矛盾。模型可能缺乏開發人員在代碼中所創建的更為微妙和靈活的特性，這要麼是因為開發人員認為模型不必具備這些特性，要麼是因為編碼風格是過程式的，只能隱含地表達領域概念。

儘管模型和基於模型的語言之間的次序像是循環論證，但是，能夠產生更有用模型的知識消化過程依賴於團隊投身於基於模型的語

言。持續使用 **UBIQUITOUS LANGUAGE** 可以暴露模型中存在的缺點，這樣團隊就可以嘗試並替換不恰當的術語或組合。當在語言中發現缺失時，新的詞語將被引入到討論中。這些語言上的更改也會在領域模型中引起相應的更改，並促使團隊更新類圖並重命名代碼中的類和方法，當術語的意義改變時，甚至會導致行為也發生改變。

通過在實現的過程中使用這種語言，開發人員能夠指出不準確和矛盾之處，並和領域專家一起找到有效的替代方案。

當然，為瞭解釋和給出更廣泛的上下文，領域專家的語言會超出 **UBIQUITOUS LANGUAGE** 的範圍。但在模型應對的範圍內，他們應該使用 **UBIQUITOUS LANGUAGE**，並在發現不合適、不完整或錯誤之處後要引起注意。通過大量使用基於模型的語言，並且不達流暢絕不罷休，我們可以逐步得到一個完整的、易於理解的模型，它由簡單元素組成，並通過組合這些簡單元素表達複雜的概念。

因此：

將模型作為語言的支柱。確保團隊在內部的所有交流中以及代碼中堅持使用這種語言。在畫圖、寫東西，特別是講話時也要使用這種語言。

通過嘗試不同的表示方法（它們反映了備選模型）來消除難點。然後重構代碼，重新命名類、方法和模塊，以便與新模型保持一致。解決交談中的術語混淆問題，就像我們對普通詞彙形成一致的理解一樣。

要認識到，**UBIQUITOUS LANGUAGE** 的更改就是對模型的更改。

領域專家應該抵制不合適或無法充分表達領域理解的術語或結構，開發人員應該密切關注那些將會妨礙設計的有歧義和不一致的地方。

有了UBIQUITOUS LANGUAGE，模型就不僅僅是一個設計工件了。它成為開發人員和領域專家共同完成的每項工作中不可或缺的部分。語言以動態形式傳遞知識。使用這種語言進行討論能夠呈現圖和代碼背後的真實含義。

我們在這裡討論的UBIQUITOUS LANGUAGE假設只有一個模型在起作用。第14章將討論不同模型（和語言）的共存，以及如何防止模型分裂。

UBIQUITOUS LANGUAGE是那些以非代碼形式呈現的設計的主要載體，這些包括把整個系統組織在一起的大尺度結構（參見第16章）、定義了不同系統和模型之間關係的限界上下文（參見第14章），以及在模型和設計中使用的其他模式。

### 示例 制定貨運路線

下面這兩段對話有著微妙但重要的差別。在每個對話場景中，注意觀察講話者有多少內容是談論軟件的業務功能，有多少內容是從技術上談論軟件的工作機理的。用戶和開發人員用的是同一種語言嗎？它的表達是否豐富，足以應對應用程序功能的討論？

### 場景1：最小化的領域抽像

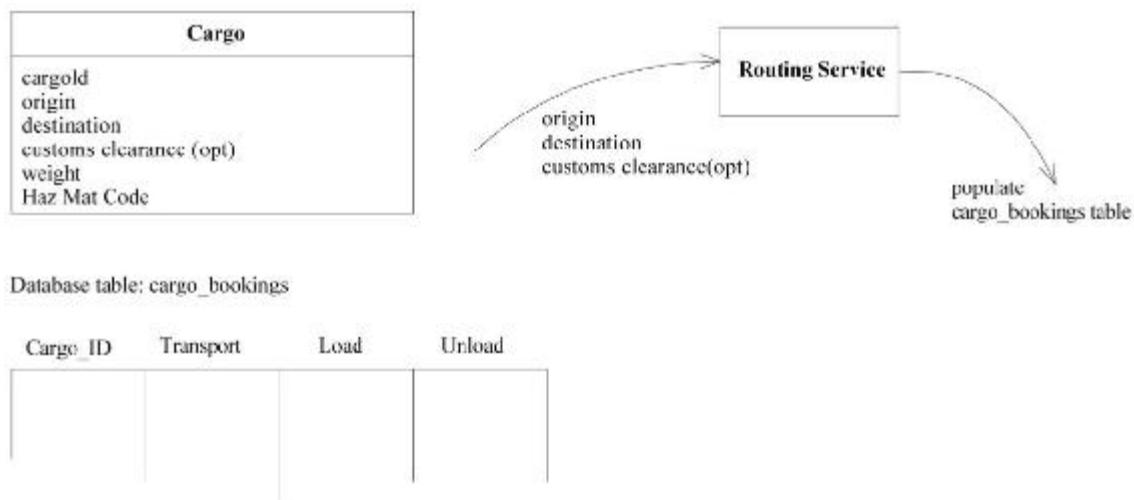


圖2-1

用戶：那麼，當更改清關 ( customs clearance ) [4] 地點時，需要重新制定整個路線計劃囉。

開發人員：是的。我們將從貨運表 ( shipment table ) 中刪除所有與該貨物 id 相關聯的行，然後將出發地、目的地和新的清關地點傳遞給 Routing Service，它會重新填充貨運表。Cargo 中必須設立一個布爾值，用於指示貨運表中是否有數據。

用戶：刪除行？好，就按你說的做。但是，如果先前根本沒有指定清關地點，也需要這麼做嗎？

開發人員：是的，無論何時更改了出發地、目的地或清關地點（或是第一次輸入），都將檢查是否已經有貨運數據，如果有，則刪除它們，然後由 Routing Service 重新生成數據。

用戶：當然，如果原有的清關數據碰巧是正確的，我們就不需要這樣做了。

開發人員：哦，沒問題。但讓 Routing Service 每次重新加載或卸載數據會更容易些。

用戶：是的，但為新航線制定所有支持計劃的工作量很大，因此，除非非改不可，我們一般不想更改航線。

開發人員：哦，好的，當第一次輸入清關地點時，我們需要查詢表格，找到以前的清關地點，然後與新的清關地點進行比較，從而判斷是否需要重做。

用戶：這個處理不必考慮出發地和目的地，因為航線在此總要變更。

開發人員：好的，我明白了。

**場景2：用領域模型進行討論**

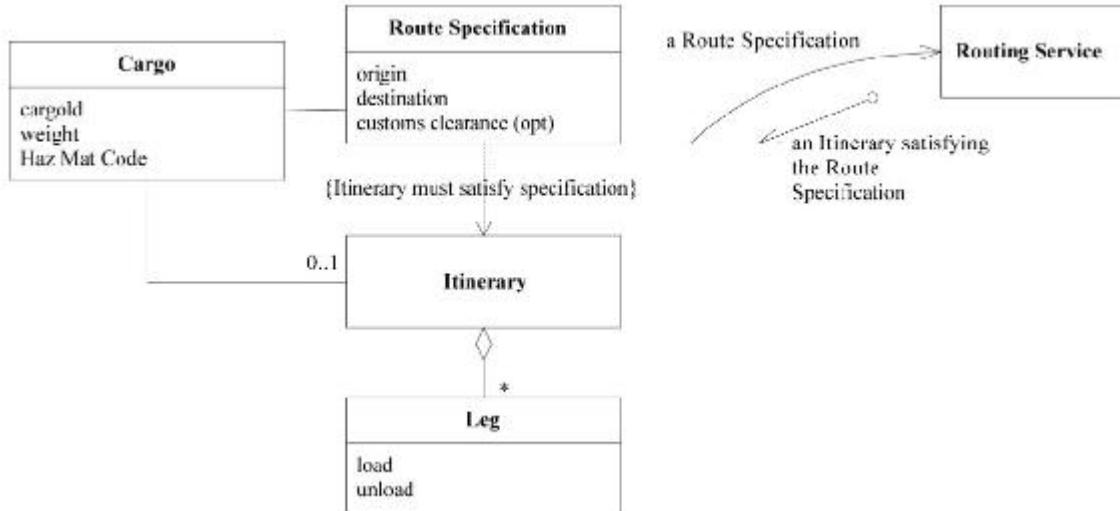


圖2-2

用戶：那麼，當更改清關地點時，需要重新制定整個路線計劃囉。

開發人員：是的。當更改**Route Specification**（路線說明）的任意屬性時，都將刪除原有的**Itinerary**（航線），並要求**Routing Service**（路線服務）基於新的**Route Specification**生成一個新的**Itinerary**。

用戶：如果先前根本沒有指定清關地點，也需要這麼做嗎？

開發人員：是的，無論何時更改了**Route Spec**的任何屬性，都將重新生成**Itinerary**。這也包括第一次輸入某些屬性。

用戶：當然，如果原有的清關數據碰巧是正確的，我們就不需要這樣做了

開發人員：哦，沒問題。但讓**Routing Service**每次重新生成一個**Itinerary**會更容易些。

用戶：是的，但為新航線制定所有支持計劃的工作量很大，因此，除非非改不可，我們一般不想更改路線。

開發人員：哦。那麼需要在**Route Specification**添加一些功能。這樣，當更改**Route Specification**中的屬性時，查看**Itinerary**是否仍滿足

Specification。如果不滿足，則需要由 Routing Service 重新生成 Itinerary。

用戶：這一點不必考慮出發地和目的地，因為 Itinerary 在此總是需要變更的。

開發人員：好的，但每次只做比較就簡單多了。只有當不滿足 Route Specification 時，才重新生成 Itinerary。

第二段對話表達了領域專家的更多意圖。在這兩段對話中，用戶都使用了「*itinerary*」這個詞，但在第二段中它是一個對象，這使得雙方可以更準確、具體地進行討論。他們明確討論了「*route specification*」，而不是每次都通過屬性和過程來描述它。

這兩段對話有意使用了相似的結構。實際上，第一段對話顯得更囉嗦，對話雙方需要不斷對應用程序的特性和表達不清的地方進行解釋。第二段對話使用了基於領域模型的術語，因此討論更簡潔。

## 2.2 「大聲地」建模

假如將交談從溝通方式中除去的話，那會是巨大的損失，因為人類本身頗具談話的天賦。遺憾的是，當人們交談時，通常並不使用領域模型的語言。

可能開始時你並不認為上述論斷是正確的，而且的確有例外情況。但下次你參加需求或設計討論時，不妨認真聽一下。你將聽到人們用業務術語或者各種業餘術語來描述功能。還會聽到人們討論技術工件和具體的功能。當然，你還會聽到來自領域模型的術語；在人們共同使用的那部分業務術語中，那些顯而易見的名詞在編碼時通常被用作對像名稱，因此這些術語經常被人們提及。但你是否也聽到一些使用當前領域模型中的關係和交互來描述的措辭呢？

改善模型的最佳方式之一就是通過對話來研究，試著大聲說出可能的模型變化中的各種結構。這樣不完善的地方很容易被聽出來。

「如果我們向Routing Service提供出發地、目的地和到達時間，就可以查詢貨物的停靠地點，嗯……將它們存到數據庫中。」（含糊且偏重於技術）

「出發地、目的地……把它們都輸入到Routing Service中，而後我們得到一個Itinerary，它包含我們所需的全部信息。」（更具體，但過於囉嗦）

「Routing Service查找滿足Route Specification的Itinerary。」（簡潔）

使用單詞和短語是極為重要的——其將我們的語言能力用於建模工作，這就如同素描對於表現視覺和空間推理十分重要一樣。我們即要利用系統性分析和設計方面的分析能力，也要利用對代碼的神秘「感覺」。這些思考方式互為補充，要充分利用它們來找到有用的模型和設計。在所有這些方式中，語言上的試驗常常是最容易被忽視的（本書第三部分將深入探討這種發現過程，並通過幾段對話來顯示它們之間的相互影響）。

事實上，我們的大腦似乎很擅長處理口語的複雜性（對於像我這樣的門外漢，有本好書是Steven Pinker所著的The Language Instinct[Pinker 1994]）。例如，當具有不同語言背景的人湊在一起做生意時，如果沒有公共語言，他們就會創造一種稱為「混雜語」（pidgin）的公共語言。混雜語雖然不像講話者的母語那樣詳盡，但它適合當前任務。當人們交談時，自然會發現詞語解釋和意義上的差別，而後會自然而然地解決這些差別。他們會發現這種語言中的簡陋晦澀之處並把它們搞順暢。

上大學時，我曾經修過西班牙語速成課。課堂上規定不準講英語。起初，令人相當沮喪。這不僅感覺很彆扭，而且需要很強的自制力。但最終我和同學們都達到了通過書面練習永遠不可能達到的流利程度。

當我們在討論中使用領域模型的UBIQUITOUS LANGUAGE時，特別是在開發人員和領域專家一起推敲場景和需求時，通用語言的使用會越來越流利，而且我們還可以互相指點一些細微的差別。我們自然而然地共享了我們所說的語言，而這種方式是圖和文檔無法做到的。

想要在軟件項目上產生一種UBIQUITOUS LANGUAGE，說起來容易，做起來卻難，我們必須充分利用自然賦予我們的才能來實現這個目標。正如人類的視覺和空間思維能力使我們能夠快速傳達和處理圖形概述中的信息一樣，我們也可以利用自己在基於語法的、有意義的語言方面的天賦來推動模型的開發。

因此，下面這段話可作為UBIQUITOUS LANGUAGE模式的補充：  
討論系統時要結合模型。使用模型元素及其交互來大聲描述場景，並且按照模型允許的方式將各種概念結合到一起。找到更簡單的表達方式來講出你要講的話，然後將這些新的想法應用到圖和代碼中。

## 2.3 一個團隊，一種語言

技術人員通常認為業務專家最好不要接觸領域模型，他們認為：  
「領域模型對他們來說太抽象了。」  
「他們不理解對象。」  
「這樣我們就不得不使用他們的術語來收集需求。」

上面只列舉了我從一個使用兩種語言的團隊中聽到的少數幾個原因。忘掉它們吧。

當然，設計中有一些技術組件與領域專家無關，但模型的核心最好讓他們參與。過於抽象？那你怎麼知道抽象是否合理？你是否像他們一樣深入理解領域？有時，某些特定需求是從底層用戶那裡收集的，他們在描述這些需求時可能會用到一小部分更具體的術語，但領域專家應該能夠更深入地思考他們所從事的領域。如果連經驗豐富的領域專家都不能理解模型，那麼模型一定出了什麼問題。

最初，當用戶討論系統尚未建模的未來功能時，他們沒有模型可供使用。但當他們開始與開發人員一起仔細討論這些新想法時，探索共享模型的過程就開始了。最初的模型可能很笨拙且不完整，但會逐漸精化。隨著新語言的演進，領域專家必須付出更多努力來適應它，並更新那些仍然很重要的舊文檔。

當領域專家使用這種語言互相討論，或者與開發人員進行討論時，很快就會發現模型中哪些地方不符合他們的需要，甚至是錯誤的。另一方面，模型語言的精確性也會促使領域專家（在開發人員的幫助下）發現他們想法中的矛盾和含糊之處。

開發人員和領域專家可以通過一步一步地使用模型對像來走查場景，從而對模型進行非正式的測試。每次討論都是開發人員和專家一起使用模型的機會，在這個過程中，他們可以加深彼此的理解，並對概念進行精化。

領域專家可以使用模型語言來編寫用例，甚至可以直接利用模型來具體說明驗收測試。

有時，有人會反對使用模型語言來收集需求。畢竟，難道需求不應該獨立於實現它們的設計嗎？這種觀點忽視了所有語言都要基於某種模型這一事實。詞的意義是不明確。領域模型通常是從領域專家自

己的術語中推導出來的，但已經經過了「清理」，以便具有更明確、更嚴密的定義。當然，如果這些定義與領域公認的意義有較大差別，領域專家應該反對。在敏捷過程中，需求是隨著項目的前進而演變的，因為幾乎不存在現成的知識可以充分說明一個應用程序。用精化後的UBIQUITOUS LANGUAGE來重新組織需求應該是這種演變過程的一部分。

語言的多樣性通常是必要的，但領域專家與開發人員之間不應該有語言上的分歧（第12章將討論多個模型在同一個項目上共存的情況）。

當然，開發人員的確會使用領域專家無法理解的技術術語。開發人員有其所需的大量術語來討論系統技術。幾乎可以肯定的是，用戶也會用開發人員無法理解的、超出應用程序範疇的專用術語。這些都是對語言的擴展。但在這些語言擴展中，同一領域的相同詞彙不應該反映不同的模型。

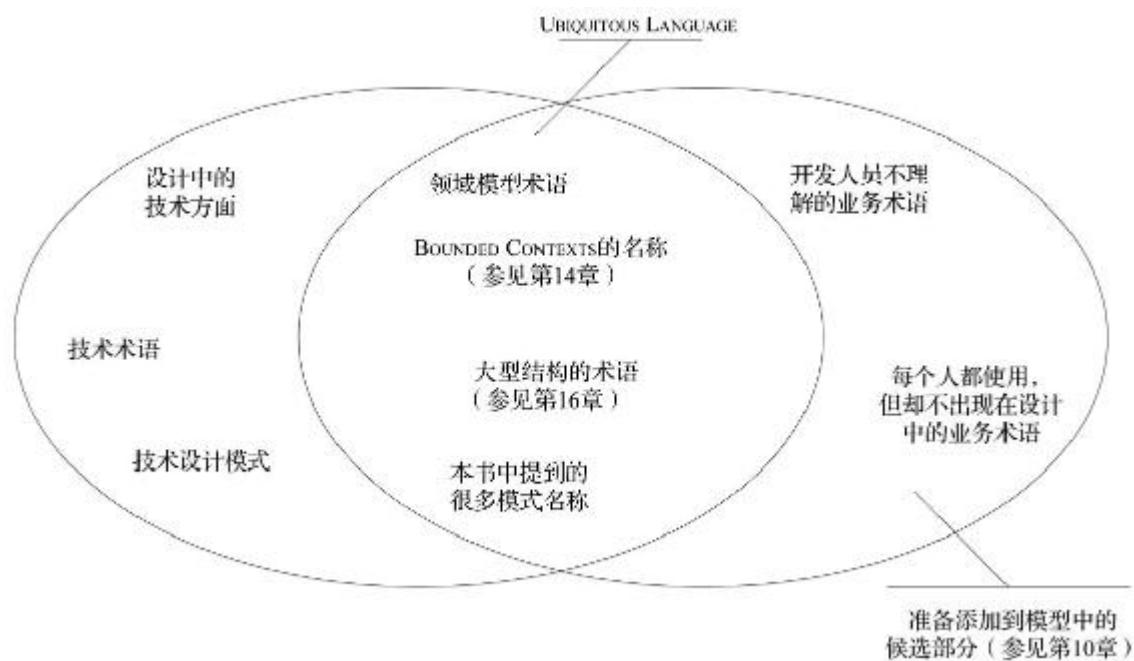


圖2-3 術語的交集產生了UBIQUITOUS LANGUAGE

有了UBIQUITOUS LANGUAGE之後，開發人員之間的對話、領域專家之間的討論以及代碼本身所表達的內容都基於同一種語言，都來自於一個共享的領域模型。

## 2.4 文檔和圖

每當我參加討論軟件設計的會議時，如果不在白板或畫板上畫圖，我就很難討論下去。我畫的大部分是UML圖，主要以類圖和對像交互圖為主。

有些人天生是視覺動物，圖可以幫助人們掌握某些類型的信息。UML圖在傳達對像之間的關係上真是遊刃有餘，而且也很擅長表現交互。但它們卻無法給出這些對象的概念定義。在會議中，我會一邊畫圖一邊用語言來豐富它們的意義，或者在與其他參與者討論時進行解釋。

簡單、非正式的UML圖能夠維繫整個討論。繪製一幅包含當前問題最關鍵的3~5個對象的圖，這樣每個人都可以集中注意力。所有人就對像關係會達成一致的認識，更重要的是，他們將使用相同的對象名稱。如此，口頭討論會更加高效。當人們嘗試不同的想法時，圖也隨之改變，草圖在某種程度上可以反映討論的變化，這是討論中真正重要的部分。畢竟，UML就是統一建模語言。

當人們必須通過UML圖表示整個模型或設計時，麻煩也隨之而來。很多對像模型圖在某些方面過於細緻，同時在某些方面又有很多遺漏。說它們過於細緻是因為人們認為必須將所有要編碼的對象都放到建模工具中。而細節過多的結果是「只見樹木，不見森林」。

儘管存在所有這些細節，但屬性和關係只是對像模型的一部分。這些對象的行為以及這些對像上的約束就不那麼容易表示了。對像交

互圖可以闡明設計中的一些複雜之處，但卻無法用這種方式來展示大量的交互，就是工作量太大了，既要製作圖，還要學習這些圖。而且交互圖也只能暗示出模型的目的。要想把約束和斷言包括進來，需要在UML圖中使用文本，這些文本用括號括起來，插入到圖中。

操作名稱可能會暗示出對象的行為職責，對像交互圖（或序列圖）中也會隱含地展示出這些職責，但無法直接表述。因此，這項任務就要靠補充文本或對話來完成。換言之，UML圖無法傳達模型的兩個最重要的方面，一個方面是模型所表示的概念的意義，另一方面是對像應該做哪些事情。但是，這並不是大問題，因為通過仔細地使用語言（英語、西班牙語或其他任何一種語言）就可以很好地完成這項任務。

UML也不是一種十分令人滿意的編程語言。我從未見過有人使用建模工具的代碼生成功能達到了預期目的。如果UML的能力無法滿足需要，通常人們就不得不忽略模型最關鍵的部分，因為有些規則並不適合用線框圖來表示。當然，代碼生成器也無法使用上面所說的那些文本註釋。如果確實能使用UML這樣的繪圖語言來編寫可執行程序，那麼UML圖就會退化為程序本身的另一種視圖，這樣，「模型」的真正含義就丢失了。如果使用UML作為實現語言，則仍然需要利用其他手段來表達模型的確切含義。

圖是一種溝通和解釋手段，它們可以促進頭腦風暴。簡潔的小圖能夠很好地實現這些目標，而涵蓋整個對象模型的綜合性大圖反而失去了溝通或解釋能力，因為它們將讀者淹沒在大量細節之中，加之這些圖也缺乏目的性。鑒於此，我們應避免使用包羅萬象的對象模型圖，甚至不能使用包含所有細節的UML數據存儲庫。相反，應使用簡化的圖，圖中只包含對像模型的重要概念——這些部分對於理解設計至關重要。本書中的圖都是我在項目中使用過比較典型的圖。它們很

簡單，而且具有很強的解釋能力，在澄清一些要點時，還使用了一些非標準的符號。它們顯示了設計約束，但它們不是面面俱到的設計規範。它們只體現了思想綱要。

設計的重要細節應該在代碼中體現出來。良好的實現應該是透明的，清楚地展示其背後的模型（下一章及本書其他許多章節的主題就是闡述如何做到這一點）。互為補充的圖和文檔能夠引導人們將注意力放在核心要點上。自然語言的討論可以填補含義上的細微差別。這就是為什麼我喜歡把典型的UML使用方法顛倒過來的原因。通常的用法是以圖為主，輔以文本註釋；而我更願意以文本為主，用精心挑選的簡化圖作為說明。

務必要記住模型不是圖。圖的目的是幫助表達和解釋模型。代碼可以充當設計細節的存儲庫。書寫良好的Java代碼與UML具有同樣的表達能力。經過仔細選擇和構造的圖可以幫助人們集中注意力，並起到指導作用，當然前提條件是不能強制用圖來表示全部模型或設計，因為這樣會削弱圖的清晰表達的能力。

### **2.4.1 書面設計文檔**

口頭交流可以解釋代碼的含義，因此可作為代碼精確性和細節的補充。雖然交談對於將人們與模型聯繫起來是至關重要的，但書面文檔也是必不可少的，任何規模的團隊都需要它來提供穩定和共享的交流。但要想編寫出能夠幫助團隊開發出好軟件的書面文檔卻是一個不小的挑戰。

一旦文檔的形式變得一成不變，往往會從項目進展流程中脫離出來。它會跟不上代碼或項目語言的演變。

書面文檔有很多編寫方法。本書第四部分將介紹幾種滿足特定需要的具體文檔，但不會列出項目需要使用的所有文檔，而是給出兩條用於評估文檔的總體原則。

## 文檔應作為代碼和口頭交流的補充

每種敏捷過程在編寫文檔方面都有自己的理念。極限編程主張完全不使用（多餘的）設計文檔，而讓代碼解釋自己。實際運行的代碼不會說謊，而其他文檔則不然。運行代碼所產生的行為是明確的。

極限編程只關注對程序及可執行測試起作用的因素。由於為代碼添加的註釋並不影響程序的行為，因此它們往往無法與當前代碼及其模型保持同步。外部文檔和圖也不會影響程序的行為，因此它們也無法保持同步。另一方面，口頭交流和臨時在白板上畫的圖不會長久保留而產生混淆。依賴代碼作為交流媒介可以促使開發人員保持代碼的整潔和透明。

然而，將代碼作為設計文檔也有侷限性。它可能會把讀代碼的人淹沒在細節中。儘管代碼的行為是非常明確的，但這並不意味著其行為是顯而易見的。而且行為背後的意義可能難以表達。換言之，只用代碼做文檔與使用大而全的UML圖面臨著差不多相同的基本問題。當然，團隊進行大量的口頭交流能夠為代碼提供上下文和指導，但是，口頭交流很短暫，而且範圍很小。此外，開發人員並不是唯一需要理解模型的人。

文檔不應再重複表示代碼已經明確表達出的內容。代碼已經含有各個細節，它本身就是一種精確的程序行為說明。

其他文檔應該著重說明含義，以便使人們能夠深入理解大尺度結構，並將注意力集中在核心元素上。當編程語言無法直接明瞭地實現概念時，文檔可以澄清設計意圖。我們應該把書面文檔作為代碼和口頭討論的補充。

## 文檔應當鮮活並保持最新

我在為模型編寫書面文檔時，會仔細選擇一個小的模型子集來畫圖，然後讓文字放鎔在這些圖周圍。我用文字定義類及其職責，並且

像自然語言那樣把它們限定在一個語義上下文中。而圖顯示了在將概念形式化和簡化為對像模型的過程中所做的一些選擇。這些圖可以隨意一些，甚至是手繪的。手繪圖除了節省工作量，也讓人們一看就知道它們是不正式、臨時的。這些優點都非常有利於交流，因為它們適用於我們的模型思想。

設計文檔的最大價值在於解釋模型的概念，幫助在代碼的細節中指引方向，或許還可以幫助人們深入瞭解模型預期的使用風格。根據不同的團隊理念，整個設計文檔可能會十分簡單，如只是貼在牆上的一組草圖，也可能會非常詳盡。

文檔必須深入到各種項目活動中去。判斷是否做到這一點的最簡單方法，是觀察文檔與**UBIQUITOUS LANGUAGE**之間的交互。文檔是用人們（當前）在項目上講的語言編寫的嗎？它是用嵌入到代碼中的語言編寫的嗎？

注意聽**UBIQUITOUS LANGUAGE**，觀察它是如何變化的。如果設計文檔中使用的術語不再出現在討論和代碼中，那麼文檔就沒有起到它的作用。或許是文檔太大或太複雜了，或許是它沒有關注足夠重要的主題。人們要麼不閱讀文檔，要麼覺得它索然無味。如果文檔對**UBIQUITOUS LANGUAGE**沒有影響，那麼一定是出問題了。

相反，我們會注意到**UBIQUITOUS LANGUAGE**隨著文檔漸漸過時而自然地改變。顯然，要麼人們不再關心文檔，要麼認為它不重要而不再去更新它。這時可以將它作為歷史文件安全地歸檔，如果繼續使用這樣的文檔可能會產生混淆並損害項目。如果文檔不再擔負重要的作用，那麼純粹靠意志和紀律保持其更新就是浪費精力。

**UBIQUITOUS LANGUAGE**可以使其他文檔（如需求規格說明）更簡潔和明確。當領域模型反映了與業務最相關的知識時，應用程序的需求成為該模型內部的場景，而**UBIQUITOUS LANGUAGE**可直接用

MODEL-DRIVEN DESIGN（模型驅動設計）的方式描述此類場景（參見第3章）。結果就是規格說明的編寫更簡單，因為它們不必傳達模型背後隱含的業務知識。

通過將文檔減至最少，並且主要用它來補充代碼和口頭交流，就可以避免文檔與項目脫節。根據UBIQUITOUS LANGUAGE及其演變來選擇那些需要保持更新並與項目活動緊密交互的文檔。

#### 2.4.2 完全依賴可執行代碼的情況

現在，我們來考查一下XP社區和其他一些人為何選擇幾乎完全依賴可執行代碼及其測試。本書主要討論瞭如何通過MODEL-DRIVEN DESIGN使代碼表達出設計的含義（參見第3章）。良好的代碼具有很強的表達能力，但它所傳遞的信息不能確保是準確的。一段代碼所產生的實際行為是不會改變的。但是，方法名稱可能會有歧義、會產生誤導或者因為已經過時而無法表示方法的本質含義。測試中的斷言是嚴格的，但變量和代碼組織方式所表達出來的意思未必嚴格。好的編程風格會盡力使這種聯繫直接化，但其仍然主要靠開發人員的自律。編碼時需要一絲不苟的態度，只有這樣才能編寫出「言行全部正確」的代碼。

消除這些差異是諸如聲明式設計（參見第10章）這樣的方法的最大優點，在這類方法中，程序元素用途的陳述決定了它在程序中的實際行為。從UML生成程序的部分動機就來源於此，雖然目前看來這通常不會得到好的結果。

儘管代碼可能會產生誤導，但它仍然比其他文檔更基礎。要想利用當前的標準技術使代碼所傳達的消息與它的行為和意圖保持一致，需要紀律和思考設計的特定方式（第三部分將詳細討論這些問題）。要有效地交流，代碼必須基於在編寫需求時所使用的同一種語言，也

就是開發人員之間、開發人員與領域專家之間進行討論時所使用的語言。

## 2.5 解釋性模型

本書的核心思想是在實現、設計和團隊交流中使用同一個模型作為基礎。如果各有各的模型，將會造成危害。

模型在幫助領域學習方面也具有很大價值。對設計起到推動作用的模型是領域的一個視圖，但為了學習領域，還可以引入其他視圖，這些視圖只用作傳遞一般領域知識的教學工具。出於此目的，人們可以使用與軟件設計無關的其他種類模型的圖片或文字。

使用其他模型的一個特殊原因是範圍。驅動軟件開發過程的技術模型必須經過嚴格的精簡，以便用最小化的模型來實現其功能。而解釋性模型則可以包含那些提供上下文的領域方面——這些上下文用於澄清範圍更窄的模型。

解釋性模型提供了一定的自由度，可以專門為某個特殊主題定製一些表達力更強的風格。領域專家在一個領域中所使用的視覺隱喻通常呈現了更清晰的解釋，這可以教給開發人員領域知識，同時使領域專家們的意見更一致。解釋性模型還可以以一種不同的方式來呈現領域，並且各種不同角度的解釋有助於人們更好地學習。

解釋性模型不必是對像模型，而且最好不是。實際上在這些模型中不使用UML是有好處的，這樣可以避免人們錯誤地認為這些模型與軟件設計是一致的。儘管解釋性模型與驅動設計的模型往往有對應關係，但它們並不完全類似。為了避免混淆，每個人都必須知道它們之間的區別。

### 示例 航運操作和路線

考慮一個用來追蹤航運公司貨物的應用程序。模型包含一個詳細的視圖，它顯示瞭如何將港口裝卸和貨輪航次組合為一次貨運的操作計劃（「路線」）。如圖2-4所示。但對外行而言，類圖可能起不到多大的說明作用。

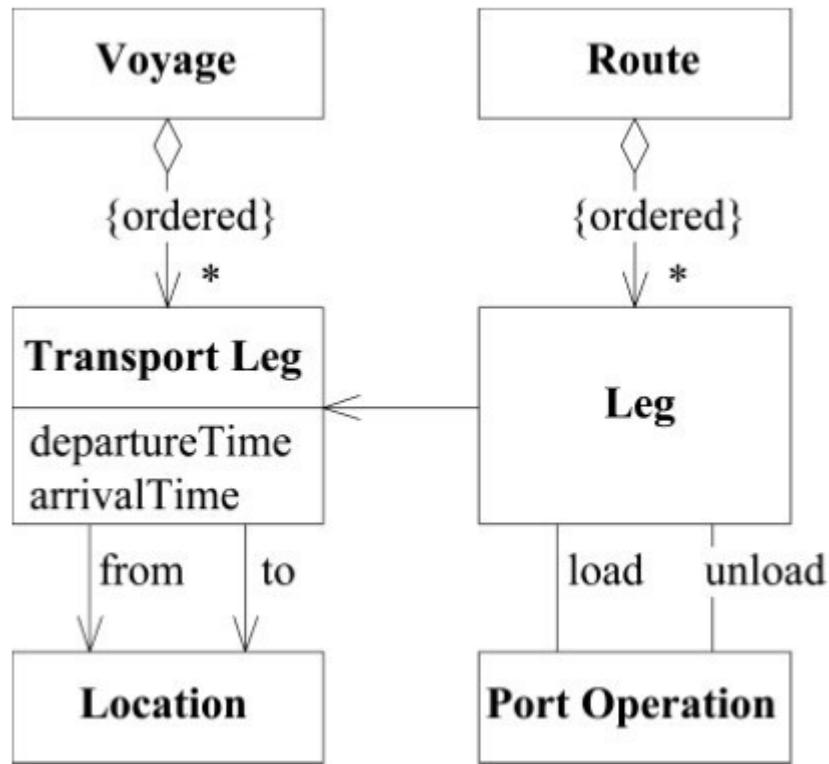


圖2-4 航運路線的類圖

在這種情況下，解釋性模型可以幫助團隊成員理解類圖的實際含義。圖2-5是表示相同概念的另一種方式。

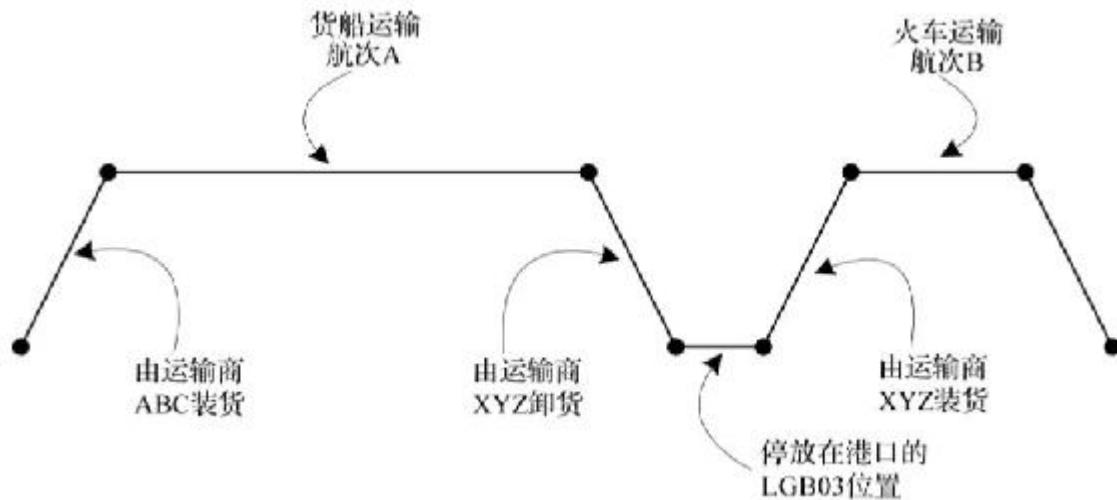


圖2-5 航運路線的解釋性模型

圖中的每根線段都表示貨物的一種狀態——或者正在港口裝卸（裝貨或卸貨），或者停放在倉庫裡，或者正在運輸途中。這個圖並沒有與類圖中的細節一一對應，但強調了領域的要點。

這種圖連同對它所表示的模型的自然語言解釋，能夠幫助開發人員和領域專家理解更嚴格的軟件模型圖。綜合使用這兩種圖要比單獨使用一種圖更容易理解。

## 第3章 綁定模型和實現

當我走進辦公室，首先映入眼簾的是打印在數張大紙上的完整類圖，它鋪滿了一整面牆。這是我進入某個項目的第一天，在此之前，聰明的項目組成員花費了幾個月的時間進行仔細的研究並且開發出了上面這幅詳盡的領域模型。該模型中的對象一般都與3~4個其他對像有著複雜的關聯，而這張關聯網幾乎沒有邊界。在這方面，分析人員忠實地反映了領域自身的性質。

儘管這張牆面大小的圖讓人吃不消，但是它所表現的模型確實捕獲了一些知識。經過一段時間的研究，我確實從中學到了不少知識（但是這種學習很難找到頭緒，更像是在隨意地瀏覽網頁）。然而對類圖的研究並不能讓我深入地瞭解該應用程序的代碼和設計，這讓我備感困擾。

當開發人員開始實現應用程序時，他們很快就發現，儘管分析人員說得頭頭是道，他們依然無法將這種錯綜複雜的關係轉換成可存儲、可檢索的且具有事務完整性的單元。請注意，該項目使用的是對像數據庫，也就是說開發人員根本不用考慮對像一關係表映射這種難題。從根本上說，該模型無法為應用程序的實現提供幫助。

由於模型是「正確的」，這是經過技術分析人員和業務專家大量協作才得到的結果，因此開發人員得出這樣的結論：無法把基於概念的對象作為設計的基礎。於是他們開始進行專門針對程序開發的設計。他們的設計確實用了一些原有模型中類和屬性的名稱進行數據存儲，但這種設計並不是建立在任何已有模型的基礎上的。

這個項目雖然建立了領域模型，但是如果模型不能直接幫助開發可運行的軟件，那麼這種紙上談兵的模型又有什麼意義呢？

幾年後，我在一個完全不同的項目中又看到了完全相同的結果。該項目要用Java實現新設計，並用新設計替換現存的C++應用程序。老版本的程序根本沒有進行對像建模，僅僅是把功能堆積在一起。老版本的設計（如果有的話）就是在已有代碼的基礎上一個一個地堆積新功能，完全沒有任何泛化或抽像的跡象。

奇怪的是，這兩種開發流程所完成的最終產品卻非常相似！它們都充斥了大量功能，難於理解，難以維護。儘管有些程序實現是比較直觀的，但是僅通過閱讀代碼依然無法深入瞭解該系統的目的所在。除了精心設計的數據結構之外，這兩種開發流程都沒有利用其開發環境中的面向對象的設計範式。

模型種類繁多，作用各有不同，即使是那些僅用於軟件開發項目的模型也是如此。領域驅動設計要求模型不僅能夠指導早期的分析工作，還應該成為設計的基礎。這種設計方法對於代碼的編寫有著重要的意義。不太明顯的一點就是：領域驅動設計要求一種不同的建模方法.....

### **3.1 模式：MODEL-DRIVEN DESIGN**



過去用來計算星體位置的星盤[5]是天空模型的機械實現

### 中世紀的星象電腦

星盤是由古希臘的天文學家發明的；在中世紀，伊斯蘭科學家又對它進行了改進。星盤上可旋轉的銅環（又稱「網環」）代表各恆星在天球上的位置。刻有當地地平坐標系的盤面是可換的，它代表的是不同緯度的星空景象。在星盤盤面上旋轉網環，可以計算出全年任何時刻的天體位置。反過來，如果知道太陽或某個恆星的位置，也可以用星盤算出時間。星盤以機械化的方式實現了代表星空的面向對像模型。

嚴格按照基礎模型來編寫代碼，能夠使代碼更好地表達設計含義，並且使模型與實際的系統相契合。

那些壓根兒就沒有領域模型的項目，僅僅通過編寫代碼來實現一個又一個的功能，它們無法利用前兩章所討論的知識消化和溝通所帶來的好處。如果涉及複雜的領域就會使項目舉步維艱。

另一方面，許多複雜項目確實在嘗試使用某種形式的領域模型，但是並沒有把代碼的編寫與模型緊密聯繫起來。這些項目所設計的模型，在項目初期還可能用來做一些探索工作，但是隨著項目的進展，這些模型與項目漸行漸遠，甚至還會起誤導作用。所有在模型上花費的精力都無法保證程序設計的正確性，因為模型和設計是不同的。

模型和程序設計之間的聯繫可能在很多情況下被破壞，但是二者的這種分離往往是有意而為之的。很多設計方法都提倡使用完全脫離於程序設計的分析模型，並且通常這二者是由不同的人員開發的。之所以稱其為分析模型，是因為它是對業務領域進行分析的結果，它在組織業務領域中的概念時，完全不去考慮自己在軟件系統中將會起到的作用。分析模型僅僅是理解工具，人們認為把它與程序實現聯繫在一起無異於攬渾一池清水。隨後的程序設計與分析模型之間可能僅僅保持一種鬆散的對應關係。在創建分析模型時並沒有考慮程序設計的問題，因此分析模型很有可能無法滿足程序設計的需求。

這種分析中會有一些知識消化的過程，但是在編碼開始後，如果開發人員不得不重新對設計進行抽象，那麼大部分的領域知識就會被丟棄。如此一來，就不能保證在新的程序設計中還能保留或者重現分析人員所獲得的並且嵌入在模型中的領域知識。到了這一步，要維護程序設計和鬆散連接的模型之間的對應關係就很不合算了。

純粹的分析模型甚至在實現理解領域這一主要目的方面也捉襟見肘，因為在程序設計和實現過程中總是會發現一些關鍵的知識點，而細節問題則會出人意料地層出不窮。前期模型可能會深入研究一些不相關的問題，反而忽略了一些重要的方面。而且它對於其他問題的描

述也可能對應用程序沒有任何幫助。最後的結果就是：編碼工作一開始，純粹的分析模型就被拋到一邊，大部分的模型都需要重新設計。即便是重新設計，如果開發人員認為分析與程序開發毫不相關，那麼建模過程就不會那麼規範。而如果項目經理也這麼認為，那麼開發團隊可能沒有足夠的機會與領域專家進行交流。

無論是什麼原因，軟件的設計如果缺乏概念，那麼軟件充其量不過是一種機械化的產品——只實現有用的功能卻無法解釋操作的原因。

如果整個程序設計或者其核心部分沒有與領域模型相對應，那麼這個模型就是沒有價值的，軟件的正確性也值得懷疑。同時，模型和設計功能之間過於複雜的對應關係也是難於理解的，在實際項目中，當設計改變時也無法維護這種關係。若分析與和設計之間產生嚴重分歧，那麼在分析和設計活動中所獲得的知識就無法彼此共享。

分析工作一定要抓住領域內的基礎概念，並且用易於理解和易於表達的方式描述出來。設計工作則需要指定一套可以由項目中使用的編程工具創建的組件，使項目可以在目標部署環境中高效運行，並且能夠正確解決應用程序所遇到的問題。

**MODEL-DRIVEN DESIGN** (模型驅動設計) 不再將分析模型和程序設計分離開，而是尋求一種能夠滿足這兩方面需求的單一模型。不考慮純粹的技術問題，程序設計中的每個對象都反映了模型中所描述的相應概念。這就要求我們以更高的標準來選擇模型，因為它必須同時滿足兩種完全不同的目標。

有很多方法可以對領域進行抽象，也有很多種設計可以解決應用程序的問題。因此，綁定模型和程序設計是切實可行的。但是這種綁定不能夠因為技術考慮而削弱分析的功能，我們也不能接受那些只反映了領域概念卻捨棄了軟件設計原則的拙劣設計。模型和設計的綁定

需要的是在分析和程序設計階段都能發揮良好作用的模型。如果模型對於程序的實現來說顯得不太實用時，我們必須重新設計它。而如果模型無法忠實地描述領域的關鍵概念，也必須重新設計它。這樣，建模和程序設計就結合為一個統一的迭代開發過程。

將領域模型和程序設計緊密聯繫在一起絕對是必要的，這也使得在眾多可選模型中選擇最適用的模型時，又多了一條選擇標準。它要求我們認真思考，並且通常會經過多次反覆修改和重新構建的過程，但是通過這樣的過程可以得到與設計關聯的模型。

因此：

軟件系統各個部分的設計應該忠實地反映領域模型，以便體現出這二者之間的明確對應關係。我們應該反覆檢查並修改模型，以便軟件可以更加自然地實現模型，即使想讓模型反映出更深層次的領域概念時也應如此。我們需要的模型不但應該滿足這兩種需求，還應該能夠支持健壯的**UBIQUITOUS LANGUAGE**（通用語言）。

從模型中獲取用於程序設計和基本職責分配的術語。讓程序代碼成為模型的表達，代碼的改變可能會是模型的改變。而其影響勢必要波及接下來相應的項目活動。

完全依賴模型的實現通常需要支持建模範式的軟件開發工具和語言，比如面向對象的編程。

有時，不同的子系統會有不同的模型（參見第14章），但是從需求分析到代碼編寫的整個開發過程中，軟件系統的每一部分只能對應一個模型。

單一模型能夠減少出錯的概率，因為程序設計直接來源於經過仔細考慮而創建的模型。程序設計，甚至是代碼本身，都與模型密不可分。

要想創建出能夠抓住主要問題並且幫助程序設計的單一模型並沒有說的那麼容易。我們不可能隨手抓個模型就把它轉化成可使用的設計。只有經過精心設計的模型才能促成功切實可行的實現。想要使代碼有效地描述模型就需要用到程序設計和實現的技巧（參見第二部分）。知識消化人員需要研究模型的各個選項，並將它們細化為實用的軟件元素。軟件開發於是就成了一個不斷精化模型、設計和代碼的統一的迭代過程（參見第三部分）。

## 3.2 建模範式和工具支持

為了使MODEL-DRIVEN DESIGN發揮作用，一定要在可控範圍內嚴格保證模型與設計之間的一致性。要實現這種嚴格的一致性，必須要運用由軟件工具支持的建模範式，它可以在程序中直接創建模型中的對應概念。

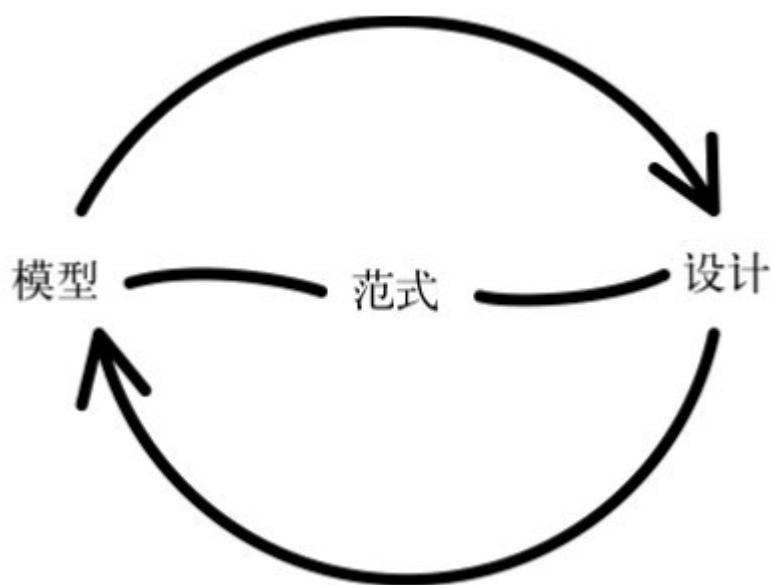


圖3-1

面向對像編程之所以功能強大，是因為它基於建模範式，並且為模型構造提供了實現方式。如圖3-1所示。從程序員的角度來看，對像真實存在於內存中，它們與其他對像相互聯繫，它們被組織成類，並且通過消息傳遞來完成相應的行為。許多開發人員只是得益於對象的技術能力——用其組織程序代碼，只有用代碼表達模型概念時，對像設計的真正突破之處才彰顯出來。**Java**和許多其他工具都允許創建直接反映概念對像模型的對象和關係。

**Prolog**語言並不像面向對像語言那樣被廣泛使用，但是它卻非常適合**MODEL-DRIVEN DESIGN**。在**MODEL-DRIVEN DESIGN**中，建模範式是邏輯的，而模型則是一組邏輯規則以及這些規則所操作的事實。

像C這樣的語言並不適用於**MODEL-DRIVEN DESIGN**，因為沒有適用於純粹過程語言的建模範式。對過程語言而言，程序員要告訴電腦一系列要執行的操作步驟。儘管程序員也會考慮到領域中的概念，但是程序本身僅僅是一組對數據進行的技術操作。最終程序可能是實用的，但是它並沒有包含太多的意義。過程語言通常支持複雜的數據類型，這些數據類型一開始就能很自然地對應到領域中的概念上，但是它們也只是被組織在一起的數據，並不能描述領域的活躍方面。因此，用過程語言編寫的軟件具有複雜的函數，這些函數基於預先制定的執行路徑連接在一起，而不是通過領域模型中的概念聯繫進行連接的。

在我還沒聽說面向對像編程的時候，我是通過**Fortran**程序來實現數學模型的，這也正是**Fortran**所擅長的領域。數學函數是這種模型中主要的概念組件，也是**Fortran**語言能夠清晰描述的。即便如此，對於超越函數的更高層次的意義，**Fortran**就毫無辦法了。大部分非數學領

域都不適合用過程語言來進行**MODEL-DRIVEN DESIGN**，因為這些領域無法被抽象成數學函數或者過程中的操作步驟。

面向對像設計是目前大多數項目所使用的建模範式，也是本書中使用的主要方法。

### 示例 從過程設計到**MODEL-DRIVEN DESIGN**

第1章講過，我們可以把PCB看作是連接各種電路元件引腳的導體（稱為**net**）集合。電路板上通常都會有成千上萬個**net**。有一種叫做PCB布線工具的專用軟件，能夠為所有**net**安排物理布線，而不會使它們相互交叉或幹擾。它的實現方法就是根據設計者規定的大量限制條件來限制布線方式以及優化路徑選擇。儘管PCB布線工具已經非常先進了，但是它仍然有一些缺陷。

其中一個問題是這些數以千計的**net**都擁有各自的布線規則，而PCB工程師會根據**net**自身的性質將其分組，同組的**net**共用相同的規則。比如，有些**net**構成了總線。如圖3-2所示。

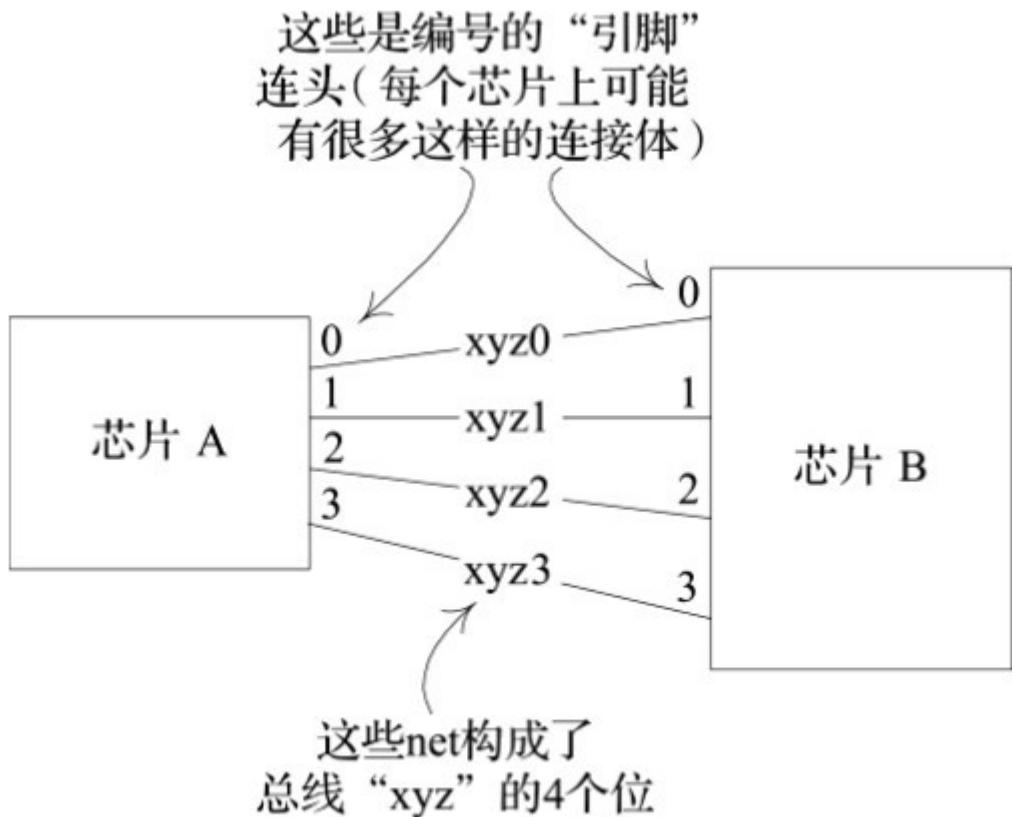


圖3-2 net構成總線的示意圖

工程師每次用8個、16個或者256個net組合成總線，這樣布線工作就更易於管理了，不但提高了效率也減少了錯誤。問題是布線工具中沒有類似於總線這樣的概念。布線規則不得不應用於成千上萬個net，一次處理一個net。

### 呆板的設計

走投無路的工程師只能用變通方法繞過布線工具的限制，編寫腳本來分析布線工具的數據文件，然後將規則直接插入到文件中，從而一次性將規則應用於整個總線。

布線工具在net列表文件中存儲每個電路連接，如下所示：

Net Name	Component.Pin
Xyz0	A.0, B.0
Xyz1	A.1, B.1
Xyz2	A.2, B.2
...	

而布線規則被存儲在類似於下面格式的文件中：

Net Name	Rule Type	Parameters
Xyz1	min_linewidth	5
Xyz1	max_delay	15
Xyz2	min_linewidth	5
Xyz2	max_delay	15
...		

工程師為**net**制定嚴格的命名約定，這樣將數據文件的內容按照字母排序，就可以使構成同一條總線的所有**net**都排列在一起。然後他們編寫的腳本就可以解析該文件並且基於總線來修改每個**net**。用來解析、處理和寫入文件的實際代碼太過冗長晦澀，對解釋這個例子沒有什麼幫助，所以我在下面只列出了這個處理過程中要執行的步驟。

- (1) 按照**net**名稱將**net**列表文件排序。
- (2) 逐行讀取文件，尋找以總線名稱開頭的第一行數據。
- (3) 解析名稱匹配的每一行，獲取每行中**net**的名稱。
- (4) 將**net**名稱和規則文本附加到規則文件的末尾。
- (5) 從第(3)步起重複執行，直到沒有匹配該總線名稱的行。

總線規則的輸入文件採用如下的格式：

Bus Name	Rule Type	Parameters
-----	-----	-----
Xyz	max_vias	3

經過處理後，輸出的是添加了net規則的文件，如下所示：

Net Name	Rule Type	Parameters
-----	-----	-----
• • •		
Xyz0	max_vias	3
Xyz1	max_vias	3
Xyz2	max_vias	3
• • •		

我猜想第一個編寫這個腳本的人只有這種簡單的需求，如果情況確實如此，那麼使用這樣的腳本是完全合理的。但實際情況是，已經存在成堆的腳本了。當然，可以通過重構來共用排序及字符串匹配之類的函數，而且如果腳本語言支持通過函數調用來封裝細節，那麼這些腳本看上去會和上面給出的步驟差不多。但是，它們依然只是對文件進行操作。文件格式一有不同（確實有幾種）就需要重新編寫一套程序，即便是總線分組以及為其分配規則的概念是相同的。如果你想要實現更多的功能和交互，就得下血本。

腳本的編寫者試圖在布線工具的領域模型中補充「總線」這個概念，他們的腳本通過排序和字符串匹配來判斷總線的存在，卻沒有明確地定義總線概念。

## MODEL-DRIVEN DESIGN

前面我們已經描述了領域專家思考問題時所使用的概念。現在需要將這些概念組織成模型，作為軟件開發的基礎。

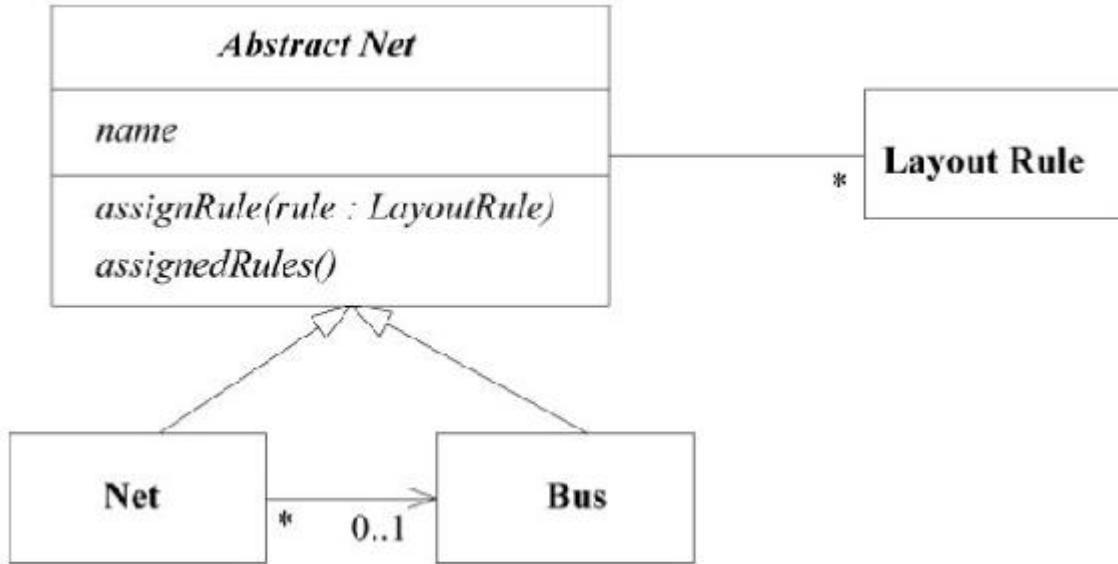


圖3-3 用來高效指定布線規則的類圖

用面向對象的語言實現上圖的這些對像後，核心功能的實現會變得輕而易舉。

方法*assignRule*可以在抽象類*AbstractNet*中實現。而類*Net*中的方法*assignedRules*則分配了其自身的規則以及類*Bus*的規則。

```
abstract class AbstractNet {  
    private Set rules;  
  
    void assignRule(LayoutManager rule) {  
        rules.add(rule);  
    }  
  
    Set assignedRules() {  
        return rules;  
    }  
}  
  
class Net extends AbstractNet {  
    private Bus bus;  
  
    Set assignedRules() {  
        Set result = new HashSet();  
        result.addAll(super.assignedRules());  
        result.addAll(bus.assignedRules());  
        return result;  
    }  
}
```

當然，程序還需要大量的支持代碼，但上面的代碼片段已經呈現了腳本的基本功能。

這個程序還需要導入/導出邏輯，我們可以將其封裝成一些簡單的服務。

服 务	职 责
Net List import	读取Net列表文件，为每一行数据创建Net实例
Net Rule export	已知Net集合，将所有附加规则写入规则文件

我們還需要幾個工具類：

类	职 责
Net Repository	提供通过名称访问Net的接口
Inferred Bus Factory	已知Net集合，利用命名约定来推断总线，并且创建总线实例
Bus Repository	提供通过名称访问Bus（总线）的接口

現在，啟動應用程序，用導入數據來初始化Net和Bus倉庫。

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

上面提到的服務和倉庫都可以進行單元測試。更重要的是還可以測試核心領域邏輯。下面是對最核心的行為進行的單元測試（採用了JUnit測試框架）：

```
public void testBusRuleAssignment() {
    Net a0 = new Net("a0");
    Net a1 = new Net("a1");
    Bus a = new Bus("a"); //Bus is not conceptually dependent
    a.addNet(a0);           //on name based recognition, and so
    a.addNet(a1);           //its tests should not be either.

    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));
}
```

程序應該有一個交互式的用戶界面，可以列出所有總線，讓用戶逐個指定規則；或者可以向後兼容，從規則文件中讀取規則。採用外

觀 ( facade ) 模式可以更容易地訪問這些接口，如下代碼是對應於上面測試的實現代碼：

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

最後一行代碼：

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

（這項服務調用每個Net的assignedRules()方法，然後將所有規則完全寫入規則文件。）

當然，如果只有一種操作（就像這個例子一樣），那麼基於腳本的處理方式可能也同樣實用。但實際上，通常會需要20個甚至更多的操作。**MODEL-DRIVEN DESIGN**易於擴展，能夠為規則的組合設置限制條件，還能提供其他的一些增強功能。

**MODEL-DRIVEN DESIGN**也為測試提供了方便。它的組件都具有定義完善的接口，可以進行單元測試。而測試腳本程序的唯一方法就是基於文件進行端到端的比較。

記住，這樣的設計不是一蹴而就的。我們需要反覆研究領域知識，不斷重構模型，才能將領域中重要的概念提煉成簡單而清晰的模型。

### **3.3 揭示主旨：為什麼模型對用戶至關重要**

從理論上講，也許你可以向用戶展示任何一種系統視圖，而不管底層如何實現。但實際上，系統上下層結構的不匹配輕則導致誤解，

重則產生bug。讓我們看一個非常簡單的例子——微軟IE瀏覽器的早期版中，網站書籤功能對應的模型是如何誤導用戶的[6]。

IE瀏覽器用戶會認為「收藏夾」是存儲網站名稱的列表，網站名稱在不同的會話中是保持不變的。但是系統實現卻將收藏夾中的書籤當作一個包含URL的文件，並將文件名稱存儲在收藏夾列表中。這樣做的問題是，如果網頁標題含有Windows系統文件名不能接受的非法字符，就會出現錯誤。假如用戶想要收藏某頁面並將其命名為：「*Laziness: The Secret to Happiness*」（懶惰：幸福的秘密），就會彈出一個錯誤信息：「文件名不能包含下列任何字符：\ / : \* ? " < > |」。用戶會奇怪文件名是指什麼。另一方面，如果網頁標題已經包含非法字符，IE瀏覽器則會悄悄地把字符刪除。這種數據丟失在這種情況下也許危害不大，但絕不是用戶所期望的。在大多數應用中，程序悄悄地修改數據是不能接受的。

**MODEL-DRIVEN DESIGN**要求只使用一個模型（在任何一個上下文中都是如此，第14章會討論這一點）。大部分的設計建議和例子都只針對將分析模型和設計模型分離的問題，但是這裡的問題涉及了另外一對不同的模型：用戶模型和設計/實現模型。

當然，大多數情況下，沒有經過處理的領域模型視圖肯定不便於用戶使用。但是在用戶界面中出現與領域模型不同的「影像」將會使用戶產生迷惑，除非這個「影像」完美無缺。如果網站收藏夾實際上只是快捷方式文件的集合，那麼應該將這一事實告訴用戶，還應該刪除之前那個起誤導作用的模型。這樣不但能使收藏夾的功能更加清晰，用戶還可以利用自己所知道的文件系統的知識來對網站收藏夾進行操作。比如，用戶可以用資源管理器來重新組織已收藏的文件，而不是用瀏覽器內置的拙劣工具。而電腦高手還能夠靈活地在文件系統的任何位臵存儲網頁快捷方式。僅僅通過刪除起誤導作用的多餘模型

就可以讓應用程序的功能更加強大且清晰。如果程序員認為原有模型足夠好，那麼為什麼還要讓用戶學習新模型呢？

此外，如果以不同的方式來存儲收藏夾，比如將其存儲在一個數據文件中，這樣收藏文件就可以有自己的規則了。這些規則很可能是應用於網頁的命名規則。這又是一個單一模型。這個模型告訴用戶所有關於網站的命名規則都適用於網站收藏夾。

如果程序設計基於一個能夠反映出用戶和領域專家所關心的基本問題的模型，那麼與其他設計方式相比，這種設計可以將其主旨更明確地展示給用戶。讓用戶瞭解模型，將使他們有更多機會挖掘軟件的潛能，也能使軟件的行為合乎情理、前後一致。

### **3.4 模式：HANDS-ON MODELER**

人們總是把軟件開發比喻成製造業。這個比喻的一個推論是：經驗豐富的工程師做設計工作，而技能水平較低的勞動力負責組裝產品。這種做法使許多項目陷入困境，原因很簡單——軟件開發就是設計。雖然開發團隊中的每個成員都有自己的職責，但是將分析、建模、設計和編程工作過度分離會對**MODEL-DRIVEN DESIGN**產生不良影響。

我曾經在一個項目中負責協調不同的應用程序開發團隊，幫助開發可以驅動程序設計的領域模型。但是管理層認為建模人員就應該只負責建模工作，編寫代碼就是在浪費這種技能，於是他們不准我編寫代碼或者與程序員討論細節問題。

開始項目進展的還算順利。我和領域專家以及各團隊的開發負責人共同工作，消化領域知識並提煉出了一個不錯的核心模型。但是該模型卻從來沒有派上用場，原因有兩個。

其一，模型的一些意圖在其傳遞過程中丢失了。模型的整體效果受細節的影響很大（這將在第二部分和第三部分討論），這些細節問題並不是總能在UML圖或者一般討論中遇到的。如果我能擼起袖子，直接與開發人員共同工作，提供一些參考代碼和近距離的技術支持，那麼他們也許能夠理解模型中的抽象概念並據此進行開發。

第二個原因是模型與程序實現及技術互相影響，而我無法直接獲得這種反饋。例如，程序實現過程中發現模型的某部分在我們的技術平臺上的工作效率極低，但是經過幾個月的時間，我才一點一點獲得了關於這個問題的全部信息。其實只需較少的改動就能解決這個問題，但是幾個月過去了，改不改已經不重要了。因為開發人員已經自行編寫出了可以運行的軟件——完全脫離了模型的設計，在那些還在使用模型的地方，也僅僅是把它當作純粹的數據結構。開發人員不分好壞地把模型全盤否定，但是他們又有什麼辦法呢？他們再也不願意冒險任由呆在象牙塔裡的架構師擺佈了。

與其他項目一樣，這個項目的初始環境傾向於不讓建模人員參與太多的程序實現。對於該項目所使用的大部分技術，我都有著大量的實踐經驗。在做建模工作之前，我甚至曾經在同類項目中領導過一個小的開發團隊，所以我對項目開發過程和編程環境非常熟悉。但是如果不能讓建模人員參與程序實現，我就是有這些經歷也無法有效地工作。

如果編寫代碼的人員認為自己沒必要對模型負責，或者不知道如何讓模型為應用程序服務，那麼這個模型就和程序沒有任何關聯。如果開發人員沒有意識到改變代碼就意味著改變模型，那麼他們對程序的重構不但不會增強模型的作用，反而還會削弱它的效果。同樣，如果建模人員不參與到程序實現的過程中，那麼對程序實現的約束就沒有切身的感受，即使有，也會很快忘記。**MODEL-DRIVEN DESIGN**

的兩個基本要素（即模型要支持有效的實現並抽像出關鍵的領域知識）已經失去了一個，最終模型將變得不再實用。最後一點，如果分工阻斷了設計人員與開發人員之間的協作，使他們無法轉達實現**MODEL-DRIVEN DESIGN**的種種細節，那麼經驗豐富的設計人員則不能將自己的知識和技術傳遞給開發人員。

HANDS-ON MODELER（親身實踐的建模者）並不意味著團隊成員不能有自己的專業角色。包括極限編程在內的每一種敏捷過程都會給團隊成員分配角色，其他非正式的專業角色也會自然而然地產生。但是如果把**MODEL-DRIVEN DESIGN**中密切相關的建模和實現這兩個過程分離開，則會產生問題。

整體設計的有效性有幾個非常敏感的影響因素——那就是細粒度的設計和實現決策的質量和一致性。在**MODEL-DRIVEN DESIGN**中，代碼是模型的表達，改變某段代碼就改變了相應的模型。程序員就是建模人員，無論他們是否喜歡。所以在開始項目時，應該讓程序員完成出色的建模工作。

因此：

任何參與建模的技術人員，不管在項目中的主要職責是什麼，都必須花時間瞭解代碼。任何負責修改代碼的人員則必須學會用代碼來表達模型。每一個開發人員都必須不同程度地參與模型討論並且與領域專家保持聯繫。參與不同工作的人都必須有意識地通過**UBIQUITOUS LANGUAGE**與接觸代碼的人及時交換關於模型的想法。

將建模和編程過程完全分離是行不通的，然而大型項目依然需要技術負責人來協調高層次的設計和建模，並幫助做出最困難或最關鍵的決策。本書的第四部分描述的就是這種決策，通過學習該部分內容

可以激發靈感，找到更高效的方法來定義高級技術人員的角色和職責。

MODEL-DRIVEN DESIGN利用模型來為應用程序解決問題。項目組通過知識消化將大量雜亂無章的信息提煉成實用的模型。而MODEL-DRIVEN DESIGN將模型和程序實現過程緊密結合。UBIQUITOUS LANGUAGE則成為開發人員、領域專家和軟件產品之間傳遞信息的渠道。

最終的軟件產品能夠在完全理解核心領域的基礎上提供豐富的功能。

如上所述，MODEL-DRIVEN DESIGN的成功離不開詳盡的設計決策。在下面幾章中我們將會講述這方面的內容。

---

[1].走查，walk through，原來是指一種非正式的代碼評審活動，現在也廣泛用於其他方面，一般是指一步步檢查或分步討論。——譯者注

[2].衛語句，guard clause，指起保護作用的語句。——譯者注

[3].STRATEGY一般是指定義一組算法，將每個算法都封裝起來，並且使它們之間可以互換。策略模式的優點是軟件可以由許多可替換的部分組成，各個部分之間是弱連接的關係，這樣軟件具有更強的可擴展性、可維護性和可重用性。作者在這裡提到策略模式，是指將每個規則當成一個算法。——譯者注

[4].清關即結關，習慣上又稱通關，是指進口貨物、出口貨物和轉運貨物進出一國海關或國境時必須向海關申報，辦理海關規定的各項手續，履行各項法規規定的義務。——譯者注

[5].一種中世紀的儀器，曾用來測量太陽或其他天體的位鎔。——譯者注

[\[6\].Brian Marick曾向我提及這個示例。](#)

## 第二部分 模型驅動設計的構造塊

為了保證軟件實現得簡潔並且與模型保持一致，不管實際情況如何複雜，必須運用建模和設計的最佳實踐。本書既不是一本介紹面向對像設計的書，也不是為了提出一些基本的設計原理。領域驅動設計改變了某些傳統觀唸的側重點。

某些設計決策能夠使模型和程序緊密結合在一起，互相促進對方的效用。這種結合要求我們注意每個元素的細節。對細節問題的精雕細琢能夠打造出一個穩定的平臺，開發人員可以在這個平臺上運用第三部分和第四部分中要講到的建模方法。

本書中的軟件設計風格主要遵循「職責驅動設計」的原則，這個原則是在[Wirfs-Brock et al.1990]中提出的，並在[Wirfs-Brock 2003]中進行了更新。同時本書也大量利用了[Meyer 1988]中所提出的「契約式設計」思想。它與其他被廣泛採用的面向對像設計最佳實踐有著基本相同的背景，這些最佳實踐在[Larman 1998]等書中給出了描述。

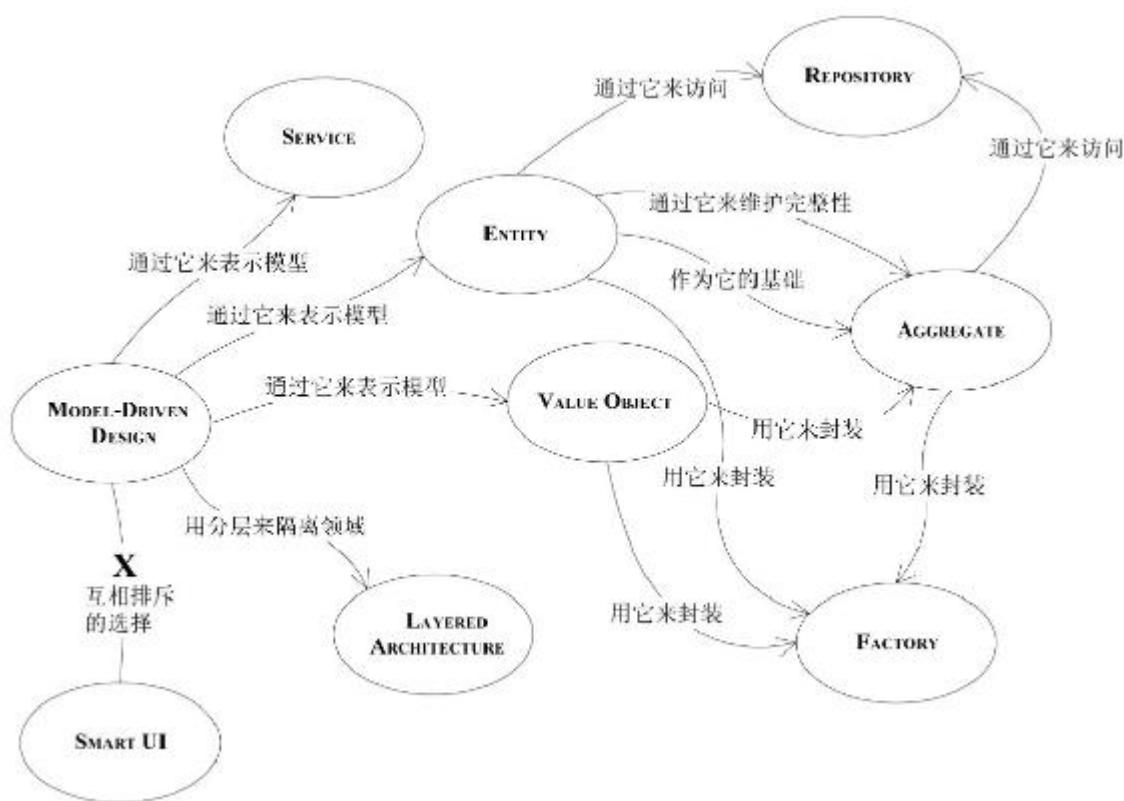
當項目遇到或大或小的困難時，開發人員可能會發現這些原則都無法適用於項目當前的狀況。為了使領域驅動設計過程更靈活，開發人員需要理解上面這些眾所周知的基本原理是如何支持**MODEL-DRIVEN DESIGN**的，這樣才能在設計過程中做出一些折中選擇，而又不脫離正確的軌道。

下面3章的內容是按照「模式語言」（參見附錄A）組織的，主要說明瞭細微的模型差別和設計決策是如何影響領域驅動設計過程的。

下面的簡圖是一張導航圖，它描述的是本部分所要講解的模式以及這些模式彼此關聯的方式。

共用這些標準模式可以使設計有序進行，也使項目組成員能夠更方便地瞭解彼此的工作內容。同時，使用標準模式也使 **UNIVERSAL LANGUAGE** 更加豐富，所有的項目組成員都可以使用 **UNIVERSAL LANGUAGE** 來討論模型和設計決策。

開發一個好的領域模型是一門藝術。而模型中各個元素的實際設計和實現則相對系統化。將領域設計與軟件系統中的其他關注點分離會使設計與模型之間的關係非常清晰。根據不同的特徵來定義模型元素則會使元素的意義更加鮮明。對每個元素使用已驗證的模式有助於創建出更易於實現的模型。



## MODEL-DRIVEN DESIGN語言的導航圖

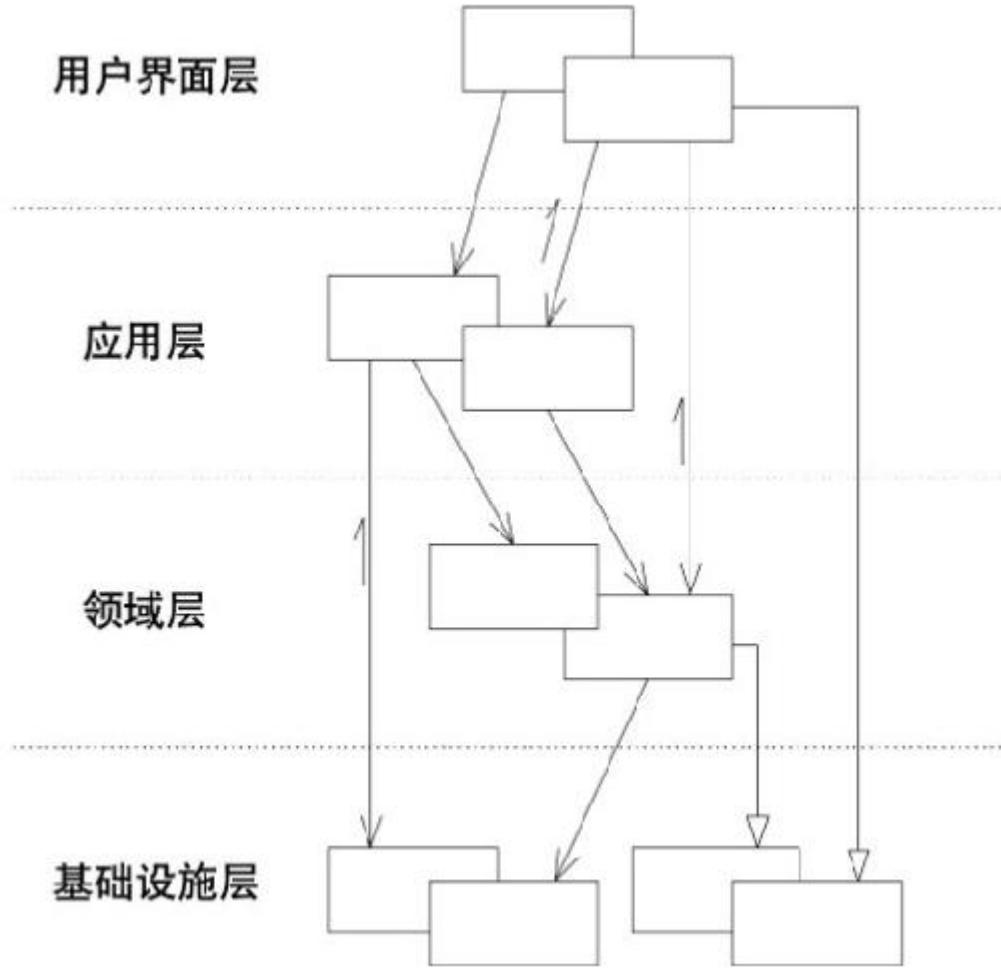
只有在充分考慮這些基本原理之後，精心設計的模型才能化繁為簡，創建出項目組成員可以放心地進行組合使用的詳細元素。

## 第4章 分離領域

在軟件中，雖然專門用於解決領域問題的那部分通常只佔整個軟件系統的很小一部分，但其卻出乎意料的重要。要想實現本書的想法，我們需要著眼於模型中的元素並且將它們視為一個系統。絕不能像在夜空中辨認星座一樣，被迫從一大堆混雜的對象中將領域對像挑選出來。我們需要將領域對象與系統中的其他功能分離，這樣就能夠避免將領域概念和其他只與軟件技術相關的概念搞混了，也不會在紛繁蕪雜的系統中完全迷失了領域。

分離領域的複雜技術早已出現，而且都是我們耳熟能詳的，但是它對於能否成功運用領域建模原則起著非常關鍵的作用，所以我們要從領域驅動的視角對它進行簡要的回顧。

### 4.1 模式：LAYERED ARCHITECTURE



在一個運輸應用程序中，要想支持從城市列表中選擇運送貨物目的地這樣的簡單用戶行為，程序代碼必須包括：(1) 在屏幕上繪製一個屏幕組件 (widget)；(2) 查詢數據庫，調出所有可能的城市；(3) 解析並驗證用戶輸入；(4) 將所選城市與貨物關聯；(5) 向數據庫提交此次數據修改。上面所有的代碼都在同一個程序中，但是隻有一小部分代碼與運輸業務相關。

軟件程序需要通過設計和編碼來執行許多不同類型的任務。它們接收用戶輸入，執行業務邏輯，訪問數據庫，進行網絡通信，向用戶顯示信息，等等。因此程序中的每個功能都可能需要大量的代碼來實現。

在面向對象的程序中，常常會在業務對像中直接寫入用戶界面、數據庫訪問等支持代碼。而一些業務邏輯則會被嵌入到用戶界面組件和數據庫腳本中。這麼做是為了以最簡單的方式在短期內完成開發工作。

如果與領域有關的代碼分散在大量的其他代碼之中，那麼查看和分析領域代碼就會變得異常困難。對用戶界面的簡單修改實際上很可能會改變業務邏輯，而要想調整業務規則很可能需要對用戶界面代碼、數據庫操作代碼或者其他的程序元素進行仔細的篩查。這樣就不太可能實現一致的、模型驅動的對象了，同時也會給自動化測試帶來困難。考慮到程序中各個活動所涉及的大量邏輯和技術，程序本身必須簡單明瞭，否則就會讓人無法理解。

要想創建出能夠處理複雜任務的程序，需要做到關注點分離——使設計中的每個部分都得到單獨的關注。在分離的同時，也需要維持系統內部複雜的交互關係。

軟件系統有各種各樣的劃分方式，但是根據軟件行業的經驗和慣例，普遍採用**LAYERED ARCHITECTURE**（分層架構），特別是有幾個層基本上已成了標準層。分層這種隱喻被廣泛採用，大多數開發人員都對其有著直觀的認識。許多文獻對**LAYERED ARCHITECTURE**也進行了充分的討論，有些是以模式的形式給出的[Buschmann et al.1996, pp.31-51]。**LAYERED ARCHITECTURE**的基本原則是層中的任何元素都僅依賴於本層的其他元素或其下層的元素。向上的通信必須通過間接的方式進行，這些將在後面討論。

分層的價值在於每一層都只代表程序中的某一特定方面。這種限制使每個方面的設計都更具內聚性，更容易解釋。當然，要分離出內聚設計中最重要的方面，選擇恰當的分層方式是至關重要的。在這裡，經驗和慣例又一次為我們指明瞭方向。儘管**LAYERED**

ARCHITECTURE的種類繁多，但是大多數成功的架構使用的都是下面這4個概念層的某種變體。

用戶界面層（或表示層）	負責向用戶顯示信息和解釋用戶指令。這裡指的用戶可以是另一個計算機系統，不一定是使用用戶界面的人。
應用層	定義軟件要完成的任務，並且指揮表達領域概念的對象來解決問題。這一層所負責的工作對業務來說意義重大，也是與其他系統的應用層進行交互的必要渠道。
領域層（或模型層）	應用層要盡量簡單，不包含業務規則或者知識，而只為下一層中的領域對象協調任務、分配工作，使它們互相協作。它沒有反映業務情況的狀態，但是却可以具有另外一種狀態，為用戶或程序顯示某個任務的進度。
基礎設施層	負責保存業務狀態的技術細節是由基礎設施層實現的，但是反映業務情況的狀態是由本層控制並主使用的。領域層是業務軟件的核心。

有些項目沒有明顯劃分出用戶界面層和應用層，而有些項目則有多個基礎設施層。但是將領域層分離出來才是實現MODEL-DRIVEN DESIGN的關鍵。

因此：

給複雜的應用程序劃分層次。在每一層內分別進行設計，使其具有內聚性並且只依賴於它的下層。採用標準的架構模式，只與上層進行鬆散的耦合。將所有與領域模型相關的代碼放在一個層中，並把它與用戶界面層、應用層以及基礎設施層的代碼分開。領域對像應該將重點放在如何表達領域模型上，而不需要考慮自己的顯示和存儲問題，也無需管理應用任務等內容。這使得模型的含義足夠豐富，結構足夠清晰，可以捕捉到基本的業務知識，並有效地使用這些知識。

將領域層與基礎設施層以及用戶界面層分離，可以使每層的設計更加清晰。彼此獨立的層更容易維護，因為它們往往以不同的速度發展並且滿足不同的需求。層與層的分離也有助於在分佈式系統中部署程序，不同的層可以靈活地放在不同服務器或者客戶端中，這樣可以減少通信開銷，並優化程序性能[Fowler 1996]。

示例 為網上銀行功能分層

該應用程序能提供維護銀行賬戶的各種功能。其中一個功能就是轉賬，用戶可以輸入或者選擇兩個賬戶號碼，填寫要轉的金額，然後開始轉賬。

為了讓這個例子更容易實現，這裡省略了一些主要的技術特性，特別是安全性方面的一些特性。領域設計也盡量簡化。（在現實生活中，銀行業務的複雜性只會增加對**LAYERED ARCHITECTURE**的需求。）此外，這個例子中的基礎設施只是為了使程序更簡單和清楚一些而已——我並不建議你使用這個設計。簡化後的功能所要完成的任務將會按照圖4-1來分層。

注意，負責處理基本業務規則的是領域層，而不是應用層——在這個例子中，業務規則就是「每筆貸款必須有與其數目相同的借款」。

這個應用程序沒有設定轉賬請求的發起方。程序中假定包含了用戶輸入界面，界面中有賬戶號碼和轉賬金額的輸入字段以及一些命令按鈕。但是也可以用基於**XML**的電匯請求來替換，這並不會影響應用層及其下面的各層。這種解耦至關重要，這並不是因為在項目中經常需要用電匯請求來代替用戶界面，而是因為關注點的清晰分離可以使每一層的設計更易理解和維護。

事實上，圖4-1本身也略微說明瞭不分離領域層會出現的問題。這張圖需要包含從請求到事務控制的所有方面，所以不得不簡化領域層來保證整個交互過程簡單易懂。如果我們專注於研究獨立領域層的設計，就可以構思並繪製出更好地表達領域規則的模型，也許模型中會包含分類賬、貸款和借款對象，或者是現金交易對象。

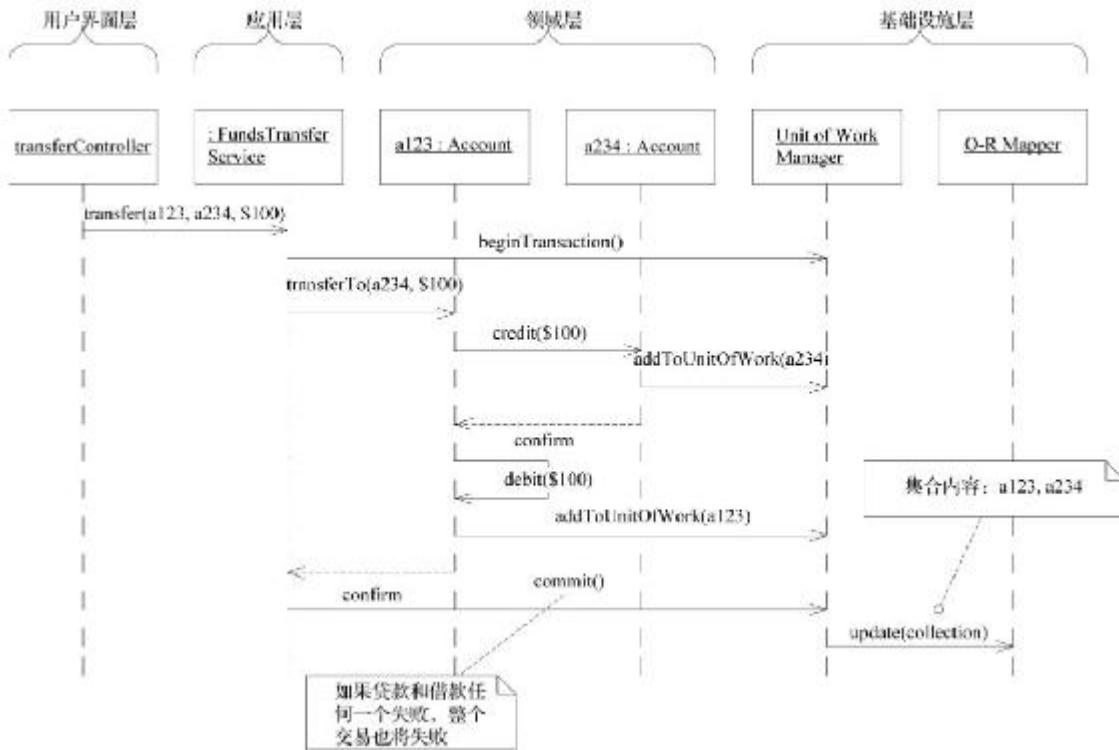


圖4-1 對像所執行的任務與其所在層一致，並且與同層其他對象的聯繫更為緊密

#### 4.1.1 將各層關聯起來

到目前為止，我們的討論主要集中在層次劃分以及如何分層才能改進程序各個方面的設計上，特別是集中在領域層上。但是顯然，各層之間也需要互相連接。在連接各層的同時不影響分離帶來的好處，這是很多模式的目的所在。

各層之間是鬆散連接的，層與層的依賴關係只能是單向的。上層可以直接使用或操作下層元素，方法是通過調用下層元素的公共接口，保持對下層元素的引用（至少是暫時的），以及採用常規的交互手段。而如果下層元素需要與上層元素進行通信（不只是回應直接查詢），則需要採用另一種通信機制，使用架構模式來連接上下層，如回調模式或OBSERVERS模式[Gamma et al.1995]。

最早將用戶界面層與應用層和領域層相連的模式是MODEL-VIEW-CONTROLLER（MVC，模型—視圖—控制器）框架。它是為Smalltalk

語言發明的一種設計模式，創建於20世紀70年代。隨後出現的許多用戶界面架構都是受到它的啟發而產生的。Fowler在[Fowler 2002]中討論了這種模式以及幾個實用的變體。Larman也在**MODEL-VIEW SEPARATION**模式中探討了這些問題，他提出的**APPLICATION COORDINATOR**（應用協調器）是連接應用層的一種方法[Larman 1998]。

還有許多其他連接用戶界面層和應用層的方式。對我們而言，只要連接方式能夠維持領域層的獨立性，保證在設計領域對像時不需要同時考慮可能與其交互的用戶界面，那麼這些連接方式就都是可用的。

通常，基礎設施層不會發起領域層中的操作，它處於領域層「之下」，不包含其所服務的領域中的知識。事實上這種技術能力最常以**SERVICE**的形式提供。例如，如果一個應用程序需要發送電子郵件，那麼一些消息發送的接口可以放在基礎設施層中，這樣，應用層中的元素就可以請求發送消息了。這種解耦使程序的功能更加豐富。消息發送接口可以連接到電子郵件發送服務、傳真發送服務或任何其他可用的服務。但是這種方式最主要的好處是簡化了應用層，使其只專注於自己所負責的工作：知道何時該發送消息，而不用操心怎麼發送。

應用層和領域層可以調用基礎設施層所提供的**SERVICE**。如果**SERVICE**的範圍選擇合理，接口設計完善，那麼通過把詳細行為封裝到服務接口中，調用程序就可以保持與**SERVICE**的鬆散連接，並且自身也會很簡單。

然而，並不是所有的基礎設施都是以可供上層調用的**SERVICE**的形式出現的。有些技術組件被設計成直接支持其他層的基本功能（如為所有的領域對像提供抽象基類），並且提供關聯機制（如**MVC**及類

似框架的實現）。這種「架構框架」對於程序其他部分的設計有著更大的影響。

### **4.1.2 架構框架**

如果基礎設施通過接口調用SERVICE的形式來實現，那麼如何分層以及如何保持層與層之間的鬆散連接就是相當顯而易見的。但是有些技術問題要求更具侵入性的基礎設施。整合了大量基礎設施需求的框架通常會要求其他層以某種特定的方式實現，如以框架類的子類形式或者帶有結構化的方法簽名。（子類在父類的上層似乎是違反常理的，但是要記住哪個類反映了另一個類的更多知識。）最好的架構框架既能解決複雜技術問題，也能讓領域開發人員集中精力去表達模型，而不考慮其他問題。然而使用框架很容易為項目製造障礙：要麼是設定了太多的假設，減小了領域設計的可選範圍；要麼是需要實現太多的東西，影響開發進度。

項目中一般都需要某種形式的架構框架（儘管有時項目團隊選擇了不太合適的框架）。當使用框架時，項目團隊應該明確其使用目的：建立一種可以表達領域模型的實現並且用它來解決重要問題。項目團隊必須想方設法讓框架滿足這些需求，即使這意味著拋棄框架中的一些功能。例如，早期的J2EE應用程序通常都會將所有的領域對像實現為「實體bean」。這種實現方式不但影響程序性能，還會減慢開發速度。現在，取而代之的最佳實踐是利用J2EE框架來實現大粒度對象，而用普通Java對像來實現大部分的業務邏輯。不妄求萬全之策，只要有選擇性地運用框架來解決難點問題，就可以避開框架的很多不足之處。明智而審慎地選擇框架中最具價值的功能能夠減少程序實現和框架之間的耦合，使隨後的設計決策更加靈活。更重要的是，現在許多框架的用法都極其複雜，這種簡化方式有助於保持業務對象的可讀性，使其更富有表達力。

架構框架和其他工具都在不斷的發展。新框架將越來越多的應用技術問題變得自動化，或者為其提供了預先設定好的解決方案。如果框架使用得當，那麼程序開發人員將可以更加專注於核心業務問題的建模工作，這會大大提高開發效率和程序質量。但與此同時，我們必須要保持克制，不要總是想著要尋找框架，因為精細的框架也可能會束縛住程序開發人員。

## 4.2 領域層是模型的精髓

現在，大部分軟件系統都採用了LAYERED ARCHITECTURE，只是採用的分層方案存在不同而已。許多類型的開發工作都能從分層中受益。然而，領域驅動設計只需要一個特定的層存在即可。

領域模型是一系列概念的集合。「領域層」則是領域模型以及所有與其直接相關的設計元素的表現，它由業務邏輯的設計和實現組成。在MODEL-DRIVEN DESIGN中，領域層的軟件構造反映出了模型概念。

如果領域邏輯與程序中的其他關注點混在一起，就不可能實現這種一致性。將領域實現獨立出來是領域驅動設計的前提。

## 4.3 模式：THE SMART UI「反模式」

上面總結了面向對像程序中廣泛採用的LAYERED ARCHITECTURE模式。在項目中，人們經常會嘗試分離用戶界面、應用和領域，但是成功分離的卻不多見，因此，分層模式的反面就很值得一談。

許多軟件項目都採用並且應該會繼續採用一種不那麼複雜的設計方法，我稱其為SMART UI（智能用戶界面）。但是SMART UI是另一種設計方法，與領域驅動設計方法迥然不同且互不兼容。如果你選擇

了SMART UI，那麼本書中所講的大部分內容都不適合你。我感興趣的是那些不應該使用SMART UI的情況，這也是我半開玩笑地稱其為「反模式」的原因。本節討論SMART UI是為了提供一種有益的對比，其將幫助我們認清在本書後面章節中的哪些情況下需要選擇相對而言更難於實現的領域驅動設計模式。

假設一個項目只需要提供簡單的功能，以數據輸入和顯示為主，涉及業務規則很少。項目團隊也沒有高級對像建模師。

如果一個經驗並不豐富的項目團隊要完成一個簡單的項目，卻決定使用 **MODEL-DRIVEN DESIGN** 以及 **LAYERED ARCHITECTURE**，那麼這個項目組將會經歷一個艱難的學習過程。團隊成員不得不去掌握複雜的新技術，艱難地學習對像建模。（即使有這本書的幫助，這也依然是一個具有挑戰性的任務！）對基礎設施和各層的管理工作使得原本簡單的任務卻要花費很長的時間來完成。簡單項目的開發週期較短，期望值也不是很高。所以，早在項目團隊完成任務之前，該項目就會被取消，更談不上去論證有關這種方法的許多種令人激動的可行性了。

即使項目有更充裕的時間，如果沒有專家的幫助，團隊成員也不太可能掌握這些技術。最後，假如他們確實能夠克服這些困難，恐怕也只會開發出一套簡單的系統。因為這個項目本來就不需要豐富的功能。

經驗豐富的團隊則不會做出這樣的選擇。身經百戰的開發人員能夠更容易學習，進而減少管理各層所需要的時間。領域驅動設計只有應用在大型項目上才能產生最大的收益，而這也確實需要高超的技巧。不是所有的項目都是大型項目；也不是所有的項目團隊都能掌握那些技巧。

因此，當情況需要時：

在用戶界面中實現所有的業務邏輯。將應用程序分成小的功能模塊，分別將它們實現成用戶界面，並在其中嵌入業務規則。用關係數據庫作為共享的數據存儲庫。使用自動化程度最高的用戶界面創建工具和可用的可視化編程工具。

這真是異端邪說啊！福音（所有地方，包括本書其他地方，都在倡導的原則）說應該是領域和UI彼此獨立。事實上，不將領域和用戶界面分離，則很難運用本書後面所要討論的方法，因此在領域驅動設計中，可以將SMART UI看作是「反模式」。然而在其他情況下，它也是完全可行的。其實，SMART UI也有其自身的優勢，在某些情況下它能發揮最佳的作用——這也是它如此普及的原因之一。在這裡介紹SMART UI能夠幫助我們理解為什麼需要將應用程序與領域分離，而且更重要的是，還能讓我們知道什麼時候不需要這樣做。

#### 優點

效率高，能在短時間內實現簡單的應用程序。

能力較差的開發人員可以幾乎不經過培訓就採用它。

甚至可以克服需求分析上的不足，只要把原型發佈給用戶，然後根據用戶反饋快速修改軟件產品即可。

程序之間彼此獨立，這樣，可以相對準確地安排小模塊交付的日期。額外擴展簡單的功能也很容易。

可以很順利地使用關係數據庫，能夠提供數據級的整合。

可以使用第四代語言工具。

移交應用程序後，維護程序員可以迅速重寫他們不明白的代碼段，因為修改代碼只會影響到代碼所在的用戶界面。

#### 缺點

不通過數據庫很難集成應用模塊。

沒有對行為的重用，也沒有對業務問題的抽象。每當操作用到業務規則時，都必須重複這些規則。

快速的原型建立和迭代很快會達到其極限，因為抽象的缺乏限制了重構的選擇。

複雜的功能很快會讓你無所適從，所以程序的擴展只能是增加簡單的應用模塊，沒有很好的辦法來實現更豐富的功能。

如果項目團隊有意識地應用這個模式，那麼就可以避免其他方法所需要的大量開銷。項目團隊常犯的錯誤是採用了一種複雜的設計方法，卻無法保證項目從頭到尾始終使用它。另一種常見的也是代價高昂的錯誤則是為項目構建一種複雜的基礎設施以及使用工業級的工具，而這樣的項目根本不需要它們。

大部分靈活的編程語言（如Java）對於小型應用程序來說是大材小用了，並且使用它們的開銷很大。第四代語言風格的工具就足以滿足這種需要了。

記住，在項目中使用智能用戶界面後，除非重寫全部的應用模塊，否則不能改用其他的設計方法。使用諸如Java這類的通用語言並不能讓你在隨後的開發過程中放棄使用SMART UI，因此，如果你選擇了這條路線，就應該採用與之匹配的開發工具。不要浪費時間去同時採用多種選擇。只使用靈活的編程語言並不一定會創建出靈活的軟件系統，反而有可能會開發出一個維護代價十分高昂的系統。

同樣道理，採用MODEL-DRIVEN DESIGN的項目團隊從項目初始就應該採用模型驅動的設計。當然，即使是經驗豐富的項目團隊在開發大型軟件系統時，也不得不從簡單的功能著手，然後在整個開發過程中使用連續的迭代開發。但是最初試探性的工作也應該是由模型驅動的，而且要分離出獨立的領域層，否則很有可能項目進行到最後就變成智能用戶界面模式了。

這裡討論SMART UI只是為了讓你認清為什麼以及何時需要採用諸如LAYERED ARCHITECTURE這樣的模式來分離出領域層。

除SMART UI和LAYERED ARCHITECTURE之外，還有一些其他的設計方案。例如，Fowler在 [Fowler 2002] 中描述了TRANSACTION SCRIPT ( 事務腳本 )，它將用戶界面從應用中分離出來，但卻並不提供對像模型。總而言之：如果一個架構能夠把那些與領域相關的代碼隔離出來，得到一個內聚的領域設計，同時又使領域與系統其他部分保持鬆散耦合，那麼這種架構也許可以支持領域驅動設計。

其他的開發風格也有各自的用武之地，但是必須要考慮到各種對於複雜度和靈活性的限制。在某些條件下，將領域設計與其他部分混在一起會產生災難性的後果。如果你要開發複雜應用軟件並且決定使用MODEL-DRIVEN DESIGN，那麼做好準備，咬緊牙關，僱用必不可少的專家，並且不要使用SMART UI。

## 4.4 其他分離方式

遺憾的是，除了基礎設施和用戶界面之外，還有一些其他的因素也會破壞你精心設計的領域模型。你必須要考慮那些沒有完全集成到模型中的領域元素。你不得不與同一領域中使用不同模型的其他開發團隊合作。還有其他的因素會讓你的模型結構不再清晰，並且影響模型的使用效率。在第14章中，會討論這方面的問題，同時會介紹其他模式，如BOUNDED CONTEXT和ANTICORRUPTION LAYER。非常複雜的領域模型本身是難以使用的，所以，第15章將會說明如何在領域層內進行進一步區分，以便從次要細節中突顯出領域的核心概念。

但是，這些都是後話。接下來，我們將會討論一些具體細節，即如何讓一個有效的領域模型和一個富有表達力的實現同時演進。畢

竟，把領域隔離出來的最大好處就是可以真正專注於領域設計，而不用考慮其他的方面。

## 第5章 軟件中所表示的模型

要想在不削弱模型驅動設計能力的前提下對實現做出一些折中，需要重新組織基本元素。我們需要將模型與實現的各個細節一一聯繫起來。本章主要討論這些基本模型元素並理解它們，以便為後面章節的討論打好基礎。

本章的討論從如何設計和簡化關聯開始。對像之間的關聯很容易想出來，也很容易畫出來，但實現它們卻存在很多潛在的麻煩。關聯也表明了具體的實現決策在MODEL-DRIVEN DESIGN中的重要性。

本章的討論將側重於模型本身，但仍繼續仔細考查具體模型選擇與實現問題之間的關係，我們將著重區分用於表示模型的3種模型元素模式：**ENTITY**、**VALUE OBJECT**和**SERVICE**。

從表面上看，定義那些用來捕獲領域概念的對象很容易，但要想反映其含義卻很困難。這要求我們明確區分各種模型元素的含義，並與一系列設計實踐結合起來，從而開發出特定類型的對象。

一個對象是用來表示某種具有連續性和標識的事物的呢（可以跟蹤它所經歷的不同狀態，甚至可以跨不同的實現跟蹤它），還是用於描述某種狀態的屬性呢？這是**ENTITY**與**VALUE OBJECT**之間的根本區別。明確地選擇這兩種模式中的一個來定義對象，有利於減少歧義，並幫助我們做出特定的選擇，這樣才能得到健壯的設計。

領域中還有一些方面適合用動作或操作來表示，這比用對像表示更加清楚。這些方面最好用**SERVICE**來表示，而不應把操作的責任強加到**ENTITY**或**VALUE OBJECT**上，儘管這樣做稍微違背了面向對象的

建模傳統。**SERVICE**是應客戶端請求來完成某事。在軟件的技術層中有很多**SERVICE**。在領域中也可以使用**SERVICE**，當對軟件要做的某項無狀態的活動進行建模時，就可以將該活動作為一項**SERVICE**。

在有些情況下（例如，為了將對像存儲在關係數據庫中）我們不得不對對像模型做一些折中改變，雖然這會影響對像模型的純度。本章將給出一些指導原則，以便在被迫處理這種複雜局面時保持正確的方向。

最後，**MODULE**的討論將有助於理解這樣一個要點——每個設計決策都應該是在深入理解領域中的某些深層知識之後做出的。高內聚、低耦合這種思想（通常被認為是一種技術指標）可應用於概念本身。在**MODEL-DRIVEN DESIGN**中，**MODULE**是模型的一部分，它們應該反映領域中的概念。

本章將所有這些體現軟件模型的構造塊組織到一起。這些都是一些傳統思想，而且一些書籍中已經介紹過從中產生的建模和設計思想。但將這些思想組織到模型驅動開發的上下文中，可以幫助開發人員創建符合領域驅動設計主要原則的具體組件，從而有助於解決更大的模型和設計問題。此外，掌握這些基本原則可以幫助開發人員在被迫做出折中設計時把握好正確的方向。

## 5.1 關聯

對像之間的關聯使得建模與實現之間的交互更為複雜。

模型中每個可遍歷的關聯，軟件中都要有同樣屬性的機制。

一個顯示了顧客與銷售代表之間關聯的模型有兩個含義。一方面，它把開發人員所認為的兩個真實的人之間的關係抽象出來。另一

方面，它相當於兩個Java對像之間的對象指針，或者相當於數據庫查詢（或類似實現）的一種封裝。

例如，一對多關聯可以用一個集合類型的實例變量來實現。但設計無需如此直接。可能沒有集合，這時可以使用一個訪問方法（accessor method）來查詢數據庫，找到相應的記錄，並用這些記錄來實例化對象。這兩種設計方法反映了同一個模型。設計必須指定一種具體的遍歷機制，這種遍歷的行為應該與模型中的關聯一致。

現實生活中有大量，多對多'關聯，其中有很多關聯天生就是雙向的。我們在模型開發的早期進行頭腦風暴活動並探索領域時，也會得到很多這樣的關聯。但這些普遍的關聯會使實現和維護變得很複雜。此外，它們也很少能表示出關係的本質。

至少有3種方法可以使得關聯更易於控制。

- (1) 規定一個遍歷方向。
- (2) 添加一個限定符，以便有效地減少多重關聯。
- (3) 消除不必要的關聯。

盡可能地對關係進行約束是非常重要的。雙向關聯意味著只有將這兩個對像放在一起考慮才能理解它們。當應用程序不要求雙向遍歷時，可以指定一個遍歷方向，以便減少相互依賴，並簡化設計。理解了領域之後就可以自然地確定一個方向。

像很多國家一樣，美國有過很多位總統。這是一種雙向的、一對多的關係。然而，在提到，喬治“華盛頓'這個名字時，我們很少會問，他是哪個國家的總統？'。從實用的角度講，我們可以將這種關係簡化為從國家到總統的單向關聯。如圖5-1所示。這種精化實際上反映了對領域的深入理解，而且也是一個更實用的設計。它表明一個方向的關聯比另一個方向的關聯更有意義且更重要。也使得Person類不受非基本概念President的束縛。

通常，通過更深入的理解可以得到一個，限定的'關係。進一步研究總統的例子就可以知道，一個國家在一段時期內只能有一位總統（內戰期間或許有例外）。這個限定條件把多重關係簡化為一對一關係，並且在模型中植入了一條明確的規則。如圖5-2所示。1790年誰是美國總統？喬治 // 華盛頓。

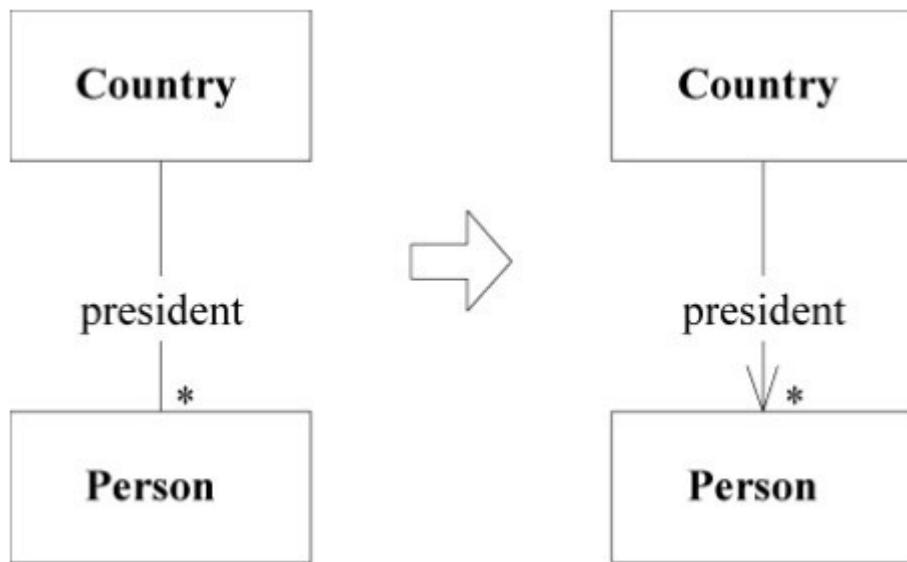


圖5-1 反映了領域自然傾向的一些遍歷方向

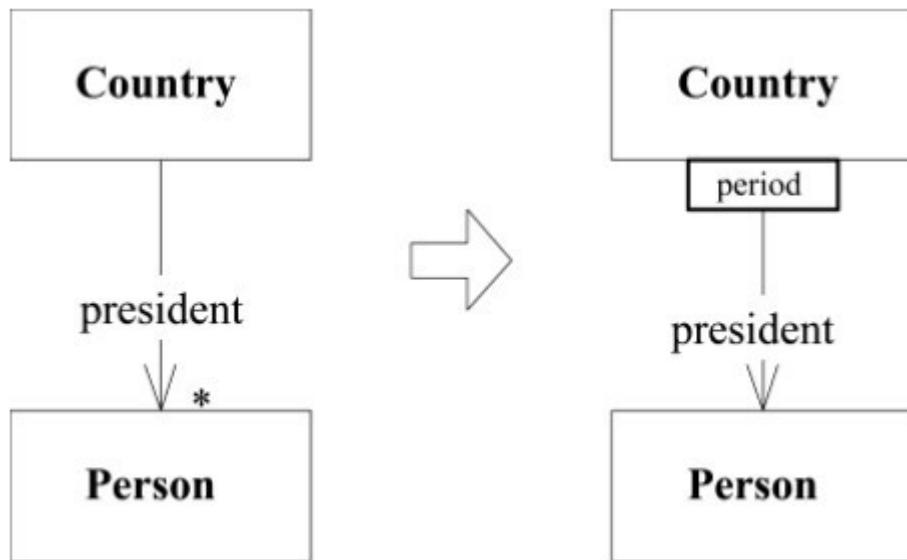


圖5-2 被約束的關聯可以傳達更多知識，而且是更實用的設計

限定多對多關聯的遍歷方向可以有效地將其實現簡化為一對多關聯，從而得到一個簡單得多的設計。

堅持將關聯限定為領域所傾向的方向，不僅可以提高這些關聯的表達力並簡化其實現，而且還可以突出剩下的雙向關聯的重要性。當雙向關聯是領域的一個語義特徵時，或者當應用程序的功能要求雙向關聯時，就需要保留它，以便表達出這些需求。

當然，最終的簡化是清除那些對當前工作或模型對象的基本含義來說不重要的關聯。

### 示例 **Brokerage Account** ( 經紀賬戶 ) 中的關聯

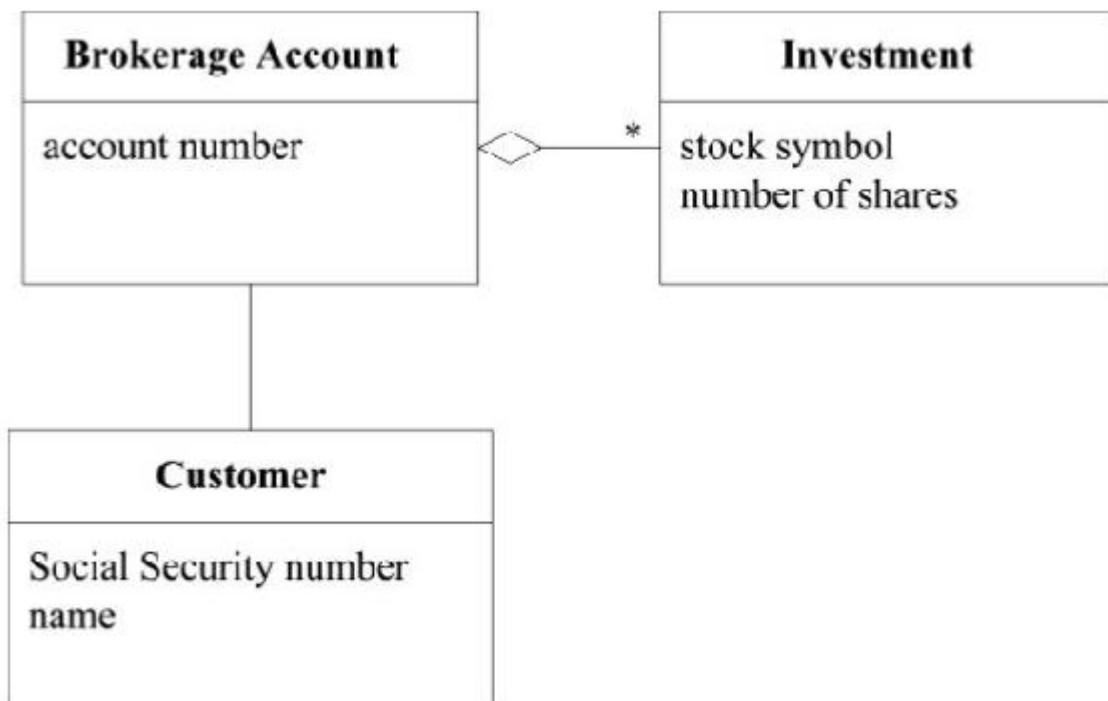


圖5-3

此模型中的Brokerage Account的一個Java實現如下：

```

public class BrokerageAccount {
    String accountNumber;
    Customer customer;
    Set investments;
    // Constructors, etc. omitted

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

```

但是，如果需要從關係數據庫取回數據，那麼就可以使用另一種實現（它同樣也符合模型）：

**Table: BROKERAGE\_ACCOUNT**

ACCOUNT_NUMBER	CUSTOMER_SS_NUMBER

**Table: CUSTOMER**

SS_NUMBER	NAME

**Table: INVESTMENT**

ACCOUNT NUMBER	STOCK_SYMBOL	AMOUNT

```

public class BrokerageAccount {
    String accountNumber;
    String customerSocialSecurityNumber;

    // Omit constructors, etc.

    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" +
            " BROKERAGE_ACCOUNT='"+accountNumber+"'";
        return QueryService.findInvestmentsFor(sqlQuery);
    }
}

```

（注意：QueryService是一個實用類，它從數據庫中取回數據行（row）並創建對象，這裡使用它是為了讓示例簡單，但這在實際項目中可不一定是個好的設計。）

下面，我們通過限定 Brokerage Account（經紀賬戶）與 Investment（投資）之間的關聯來簡化其多重性，從而對模型進行精化。具體的限定是：每支股票只能對應於一筆投資，如圖5-4所示。

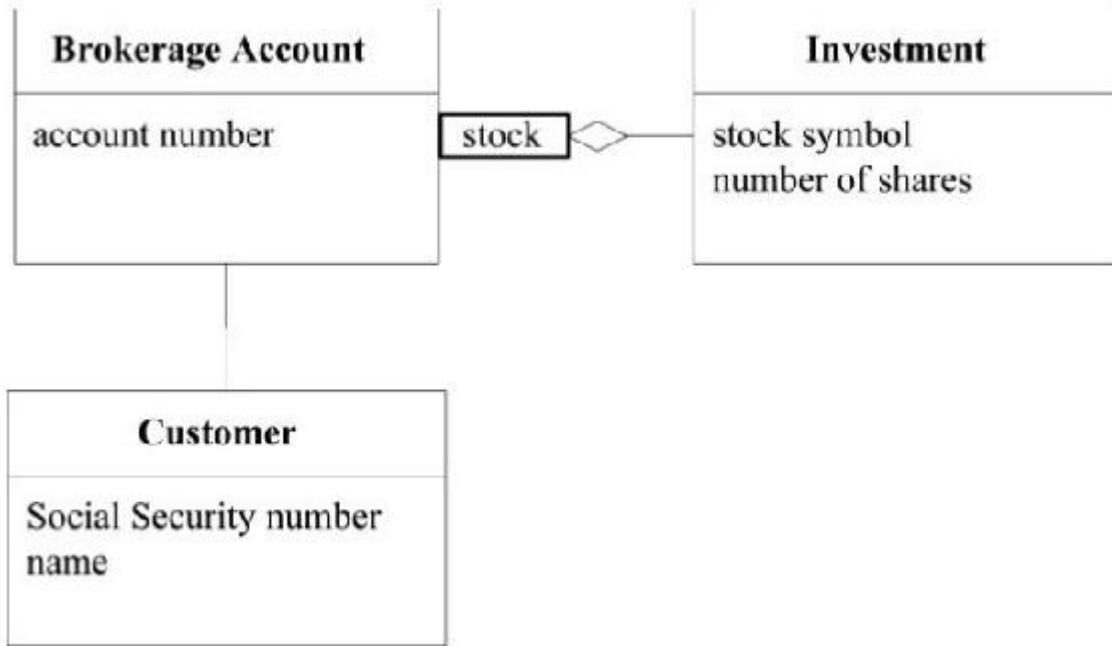


圖5-4

這種簡化並不適合所有的業務情形（例如，當所有投資都要可追蹤時），但不管是什麼特殊規則，只要發現了關聯的約束，就應該將這些約束添加到模型和實現中。它們可以使模型更精確，使實現更易於維護。

現在，Java實現變成下面這樣：

```
public class BrokerageAccount {  
    String accountNumber;  
    Customer customer;  
    Map investments;  
  
    // Omitting constructors, etc.  
  
    public Customer getCustomer() {  
        return customer;  
    }  
    public Investment getInvestment(String stockSymbol) {  
        return (Investment)investments.get(stockSymbol);  
    }  
}
```

基於SQL的實現如下：

```
public class BrokerageAccount {  
    String accountNumber;  
    String customerSocialSecurityNumber;  
  
    //Omitting constructors, etc.  
  
    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 BROKERAGE_ACCOUNT=' " + accountNumber + " '"  
            + "AND STOCK_SYMBOL=' " + stockSymbol + "'";  
        return QueryService.findInvestmentFor(sqlQuery);  
    }  
}
```

從仔細地簡化和約束模型的關聯到MODEL-DRIVEN DESIGN，還有一段漫長的探索過程。現在我們轉向對像本身。仔細區分對象可以使得模型更加清晰，並得到更實用的實現。

## **5.2 模式：ENTITY ( 又稱為REFERENCE OBJECT )**



很多對像不是通過它們的屬性定義的，而是通過連續性和標識定義的。

一位女房東起訴了我，要求我賠償她房屋的大部分損失。訴狀上是這樣寫的：房間的牆上有很多小洞，地毯上滿是汙漬，水池裡的髒物散發出的腐蝕性氣體導致廚房牆皮脫落。法庭文件認定我作為承租人應該為這些損失負責，依據就是我的名字和我當時的地址。這把我完全搞糊塗了，因為我從未去過那個被損壞的房子。

過了一會兒，我意識到這一定是認錯人了。我給原告打電話，告訴她這一點，但她並不相信我。幾個月以來，上一位租客一直在躲避她。如何才能證明我不是那個破壞她房屋的人呢？現在電話簿裡只有一個Eric Evans名字，那就是我。

還是電話簿成了我的救星。由於我在這所公寓裡已經住了兩年，於是我在電話簿裡找到了去年的電話簿。她找到了電話簿，發現有與我同名的人（我就在那個人下面），她意識到我不是她要起訴的那個人，於是向我道歉，並答應撤銷起訴。

計算機可不會這麼，足智多謀<sup>1</sup>。軟件系統中的錯誤標識將導致數據破壞和程序錯誤。

這裡存在一些特殊的技術挑戰，我們稍後將會稍加說明，這裡先來看一下基本問題。很多事物是由它們的標識定義的，而不是由任何屬性定義的。我們一般會認為，一個人（繼續使用非技術示例）有一個標識，這個標識會陪伴他走完一生（甚至死後）。這個人的物理屬性會發生變化，最後消失。他的名字可能改變，財務關係也會發生變化，沒有哪個屬性是一生不變的，但標識卻是永久的。我跟我5歲時是同一個人嗎？這種聽上去像是純哲學的問題在探索有效的領域模型時非常重要。稍微變換一下問題的角度：應用程序的用戶是否關心現在的我和5歲時的我是不是同一個人？

在一個跟蹤到期應收賬款的軟件系統中，即便最普通的，客戶<sup>1</sup>對象也可能具有豐富多彩的一面。如果按時付款的話客戶信用就會提高，

如果未能付款則將其移交給賬單清繳機構。當銷售人員將客戶數據提取出來，並放到聯繫人管理軟件中時，'客戶'對像在這個系統中就開始了另一種生活。無論是哪種情況，它都會被扁平化以存儲在數據庫表中。當業務最終停擺的時候，客戶對象就'退休'了，變成歸檔狀態，成為先前自己的一個影子。

客戶對象的這些形式都是基於不同編程語言和技術的不同實現。但當接到訂單電話時，知道以下事情是很重要的：這個客戶是不是那個拖欠了賬務的客戶？這個客戶是不是那個已經與Jack（一位銷售代表）保持聯絡達好幾個星期的客戶？還是說他完全是一個新客戶？

在對象的多個實現、存儲形式和真實世界的參與者（如打電話的人）之間，概念性標識必須是匹配的。屬性可以不匹配，例如，銷售代表可能已經在聯繫軟件中更新了地址，而這個更新正在傳送給到期應收賬款軟件。兩個客戶可能同名。在分佈式軟件中，多個用戶可能從不同地點輸入數據，這需要在不同的數據庫中異步地協調這些更新事務，使它們傳播到整個系統。

對像建模有可能把我們的注意力引到對象的屬性上，但實體的基本概念是一種貫穿整個生命週期（甚至會經歷多種形式）的抽象的連續性。

一些對像主要不是由它們的屬性定義的。它們實際上表示了一條「標識線」（**A Thread of Identity**），這條線跨越時間，而且常常經歷多種不同的表示。有時，這樣的對象必須與另一個具有不同屬性的對象相匹配。而有時一個對像必須與具有相同屬性的另一個對像區分開。錯誤的標識可能會破壞數據。

主要由標識定義的對象被稱作**ENTITY**<sup>[1]</sup>。ENTITY（實體）有特殊的建模和設計思路。它們具有生命週期，這期間它們的形式和內容可能發生根本改變，但必須保持一種內在的連續性。為了有效地跟蹤

這些對象，必須定義它們的標識。它們的類定義、職責、屬性和關聯必須由其標識來決定，而不依賴於其所具有的屬性。即使對於那些不發生根本變化或者生命週期不太複雜的**ENTITY**，也應該在語義上把它們作為**ENTITY**來對待，這樣可以得到更清晰的模型和更健壯的實現。

當然，軟件系統中的大多數**ENTITY**並不是人，也不是其通常意義上所指的，實體'或，存在'。**ENTITY**可以是任何事物，只要滿足兩個條件即可，一是它在整個生命週期中具有連續性，二是它的區別並不是由那些對用戶非常重要的屬性決定的。**ENTITY**可以是一個人、一座城市、一輛汽車、一張彩票或一次銀行交易。

另一方面，在一個模型中，並不是所有對象都是具有有意義標識的**ENTITY**。但是，由於面向對像語言在每個對象中都構建了一些與，標識'有關的操作（如Java中的，`==`操作符），這個問題變得有點讓人困惑。這些操作通過比較兩個引用在內存中的位臘（或通過其他機制）來確定這兩個引用是否指向同一個對象。從這個角度講，每個對象實例都有標識。比方說，當創建一個用於將遠程對像緩存到本地的Java運行時環境或技術框架時，這個領域中的每個對象可能確實都是一個**ENTITY**。但這種標識機制在其他應用領域中卻沒什麼意義。標識是**ENTITY**的一個微妙的、有意義的屬性，我們是不能把它交給語言的自動特性來處理的。

讓我們考慮一下銀行應用程序中的交易。同一天、同一個賬戶的兩筆數額相同的存款實際上是兩次不同的交易，因此它們是具有各自標識的**ENTITY**。另一方面，這兩筆交易的金額屬性可能是某個貨幣對象的實例。這些值沒有標識，因為沒有必要區分它們。事實上，兩個對象可能有相同的標識，但屬性可能不同，在需要的情況下甚至可能不屬於同一個類。當銀行客戶拿銀行結算單與支票記錄簿進行交易對賬時，這項任務就是匹配具有相同標識的交易，儘管它們是由不同的

人在不同的日期記錄的（銀行清算日期比支票上的日期晚）。支票號碼就是用於對賬的唯一標識符，無論這個問題是由計算機程序處理還是手工處理。存款和取款沒有標識號碼，因此可能更複雜，但同樣的原則也是適用的——每筆交易都是一個ENTITY，至少出現在兩張業務表格中。

標識的重要性並不僅僅體現在特定的軟件系統中，在軟件系統之外它通常也是非常重要的，銀行交易和公寓租客的例子中就是如此。但有時標識只有在系統上下文中才重要，如一個計算機進程的標識。

因此：

當一個對像由其標識（而不是屬性）區分時，那麼在模型中應該主要通過標識來確定該對象的定義。使類定義變得簡單，並集中關注生命週期的連續性和標識。定義一種區分每個對象的方式，這種方式應該與其形式和歷史無關。要格外注意那些需要通過屬性來匹配對象的需求。在定義標識操作時，要確保這種操作為每個對象生成唯一的結果，這可以通過附加一個保證唯一性的符號來實現。這種定義標識的方法可能來自外部，也可能是由系統創建的任意標識符，但它在模型中必須是唯一的標識。模型必須定義出「符合什麼條件才算是相同的事物」。

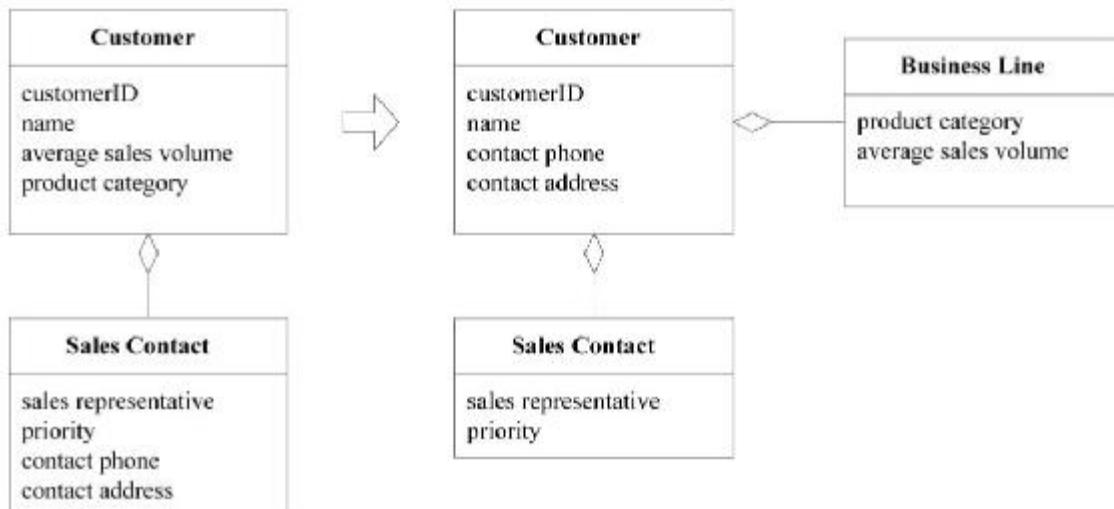
在現實世界中，並不是每一個事物都必須有一個標識，標識重不重要，完全取決於它是否有用。實際上，現實世界中的同一個事物在領域模型中可能需要表示為ENTITY，也可能不需要表示為ENTITY。

體育場座位預訂程序可能會將座位和觀眾當作ENTITY來處理。在分配座位時，每張票都有一個座位號，座位是ENTITY。其標識符就是座位號，它在體育場中是唯一的。座位可能還有很多其他屬性，如位鎔、視野是否開闊、價格等，但只有座位號（或者說某一排的一個位鎔）才用於識別和區分座位。

另一方面，如果活動採用入場卷的方式，那麼觀眾可以尋找任意的空座位來坐，這樣就不需要對座位加以區分。在這種情況下，只有座位總數才是重要的。儘管座位上仍然印有座位號，但軟件已經不需要跟蹤它們。事實上，這時如果模型仍然將座位號與門票關聯起來，那麼它就是錯誤的，因為採用入場卷的活動並沒有這樣的約束。在這種情況下，座位不是ENTITY，因此不需要標識符。

### 5.2.1 ENTITY建模

當對一個對像進行建模時，我們自然而然會考慮它的屬性，而且考慮它的行為也顯得非常重要。但ENTITY最基本的職責是確保連續性，以便使其行為更清楚且可預測。保持實體的簡練是實現這一責任的關鍵。不要將注意力集中在屬性或行為上，應該擺脫這些細枝末節，抓住ENTITY對像定義的最基本特徵，尤其是那些用於識別、查找或匹配對象的特徵。只添加那些對概念至關重要的行為和這些行為所必需的屬性。此外，應該將行為和屬性轉移到與核心實體關聯的其他對像中。這些對像中，有些可能是ENTITY，有些可能是VALUE OBJECT（這是本章接下來要討論的模式）。除了標識問題之外，實體往往通過協調其關聯對象的操作來完成自己的職責。



### 圖5-5 與標識有關的屬性留在ENTITY內

在圖5-5中，customerID是Customer ENTITY的一個（也是唯一的）標識符，但phone number（電話號碼）和address（地址）都經常用來查找或匹配一個Customer（客戶）。name（姓名）沒有定義一個人的標識，但它通常是確定人的方式之一。在這個示例中，phone和address屬性被移到Customer中，但在實際的項目上，這種選擇取決於領域中的Customer一般是如何匹配或區分的。例如，如果一個Customer有很多用於不同目的的phone number，那麼phone number就與標識無關，因此應該放在Sales Contact（銷售聯繫人）中。

## 5.2.2 設計標識操作

每個ENTITY都必須有一種建立標識的操作方式，以便與其他對像區分開，即使這些對象與它具有相同的描述屬性。不管系統是如何定義的，都必須確保標識屬性在系統中是唯一的，即使是在分佈式系統中，或者對像已被歸檔，也必須確保標識的唯一性。

如前所述，面向對像語言有一些‘標識’操作，它們通過比較對像在內存中的位鎔來確定兩個引用是否指向同一個對象。這種標識跟蹤機制過於簡單，無法滿足我們的目的。在大多數對像持久存儲技術中，每次從數據庫檢索出一個對像時，都會創建一個新實例，這樣原來的標識就丟失了。每次在網絡上傳輸對像時，在目的地也會創建一個新實例，這也會導致標識的丟失。當系統中存在同一對象的多個版本時（例如，通過分佈式數據庫來傳播更新的時候），問題將會更複雜。

儘管有一些用於簡化這些技術問題的框架，但基本問題仍然存在。如何才能判定兩個對象是否表示同一個概念ENTITY？標識是在模型中定義的。定義標識要求理解領域。

有時，某些數據屬性或屬性組合可以確保它們在系統中具有唯一性，或者在這些屬性上加一些簡單約束可以使其具有唯一性。這種方法為**ENTITY**提供了唯一鍵。例如，日報可以通過名稱、城市和出版日期來識別。（但要注意臨時增刊和名稱變更！）

當對像屬性沒辦法形成真正唯一鍵時，另一種經常用到的解決方案是為每個實例附加一個在類中唯一的符號（如一個數字或字符串）。一旦這個**ID**符號被創建並存儲為**ENTITY**的一個屬性，必須將它指定為不可變的。它必須永遠不變，即使開發系統無法直接強制這條規則。例如，當對像被扁平化到數據庫中或從數據庫中重新創建時，**ID**屬性應該保持不變。有時可以利用技術框架來實現此目的，但如果沒有這樣的框架，就需要通過工程紀律來約束。

**ID**通常是由系統自動生成的。生成算法必須確保**ID**在系統中是唯一的。在並行處理系統和分佈式系統中，這可能是一個難題。生成這種**ID**的技術超出了本書的範圍。這裡的目的是指出何時需要考慮這些問題，以便使開發人員能夠意識到有一個問題等待他們去解決，並知道如何將注意力集中到關鍵問題上。關鍵是要認識到標識問題取決於模型的特定方面。通常，要想找到解決標識問題的方法，必須對領域進行仔細的研究。

當自動生成**ID**時，用戶可能永遠不需要看到它。**ID**可能只是在內部需要，例如，在一個可以按人名查找記錄的聯繫人管理應用程序中。這個程序需要用一種簡單、明確的方式來區分兩個同名聯繫人，這就可以通過唯一的內部**ID**來實現。在檢索出兩個不同的條目後，系統將顯示這兩個不同的聯繫人，但可能不會顯示**ID**。用戶可以通過這兩個人的公司、地點等屬性來區分他們。

最後，在有些情況下用戶會對生成的**ID**感興趣。當我委託一個包裹運送服務寄包裹時，我會得到一個跟蹤號，它是由運送公司的軟件

生成的，我可以使用這個號碼來識別和跟蹤我的包裹。當我預訂機票或酒店時，會得到一個確認號碼，它是預訂交易的唯一標識符。

在某些情況下，需要確保ID在多個計算機系統之間具有唯一性。例如，如果需要在兩傢俱有不同計算機系統的醫院之間交換醫療記錄，那麼理想情況下每個系統對同一個病人應該使用同一個ID，但如果這兩個系統各自生成自己的ID，這就很難實現。這樣的系統通常使用由另外一家機構（一般是政府機構）發放的標識符。在美國，醫院通常使用社會保險號碼作為病人的標識符。但這樣的方法也不是萬無一失的，因為並不是每個人都有社會保險號碼（特別是兒童和非美國居民），而且很多人會出於個人隱私原因而反對這種做法。

在一些非正式的場合（比方說，音像出租），可以使用電話號碼作為標識符。但電話可能是共用的，號碼也可能會更改，甚至一個舊的電話號碼可能會重新分配給一個不同的人。

由於這些原因，我們一般使用特別指定的標識符（如常飛乘客[\[2\]](#)編號），並使用其他屬性（如電話號碼和社會保險號碼[\[3\]](#)）進行匹配和驗證。在任何情況下，當應用程序需要一個外部ID時，都由系統的用戶負責提供唯一的ID，而系統必須為用戶提供適當的工具來處理異常情況。

在這些技術問題的幹擾下，人們很容易忽略基本的概念問題：兩個對象是同一事物時意味著什麼？我們很容易為每個對象分配一個ID，或是編寫一個用於比較兩個實例的操作，但如果這些ID或操作沒有對應領域中有意義的區別，那隻會使問題更混亂。這就是分配標識的操作通常需要人工輸入的原因。例如，支票簿對賬軟件可以提供一些有可能匹配的賬目，但它們是否真的匹配則要由用戶最終決定。

## **5.3 模式：VALUE OBJECT**



很多對像沒有概念上的標識，它們描述了一個事務的某種特徵。

當一個小孩畫畫的時候，他注意的是畫筆的顏色和筆尖的粗細。但如果有兩隻顏色和粗細相同的畫筆，他可能不會在意使用哪一支。如果有一支筆弄丟了，他可以從一套新筆中拿出一支同樣顏色的筆來繼續畫，根本不會在意已經換了一支筆。

問問孩子冰箱上的畫都是誰畫的，他會很快辨認出哪些是他畫的，哪些是他姐姐畫的。姐弟倆有一些實用的標識來區分自己，與此類似，他們完成的作品也有。但設想一下，如果孩子必須記住哪些線條是用哪支筆畫的，情況該有多麼複雜？如果這樣的話，畫畫將不再是小孩子遊戲了。

由於模型中最引人注意的對象往往是**ENTITY**，而且跟蹤每個**ENTITY**的標識是極為重要的，因此我們很自然地會想到為每個領域對

象都分配一個標識。實際上，一些框架確實為每個對象分配了一個唯一的ID。

這樣一來，系統就必須處理所有這些ID的跟蹤問題，從而導致許多本來可能的性能優化不得不被放棄。此外，人們還需要付出大量的分析工作來定義有意義的標識，還需要開發出一些可靠的跟蹤方式，以便在分佈式系統或在數據庫存儲中跟蹤對象。同樣重要的是，盲目添加無實際意義的標識可能會產生誤導。它會使模型變得混亂，並使所有對象看起來千篇一律。

跟蹤**ENTITY**的標識是非常重要的，但為其他對象也加上標識會影響系統性能並增加分析工作，而且會使模型變得混亂，因為所有對象看起來都是相同的。

軟件設計要時刻與複雜性做鬥爭。我們必須區別對待問題，僅在真正需要的地方進行特殊處理。

然而，如果僅僅把這類對像當作沒有標識的對象，那麼就忽略了它們的工具價值或術語價值。事實上，這些對像有其自己的特徵，對模型也有著自己的重要意義。這些是用來描述事物的對象。

用於描述領域的某個方面而本身沒有概念標識的對象稱為**VALUE OBJECT**（值對像）。**VALUE OBJECT**被實例化之後用來表示一些設計元素，對於這些設計元素，我們只關心它們是什麼，而不關心它們是誰。

「地址」是**VALUE OBJECT**嗎？誰會問這個問題？

在一個郵購公司的軟件中，需要用地址來核實信用卡並投遞包裹。但如果一個人的室友也從同一家公司訂購了貨物，那麼是否意識到他們住在同一個地方並不重要。因此地址是一個**VALUE OBJECT**。

在一個用於安排投遞路線的郵政服務軟件中，國家可能被組織為一個由地區、城市、郵政區、街區以及最終的個人地址組成的層次結構。這些地址對象可以從它們在層次結構中的父對像獲取郵政編碼，而且，如果郵政服務決定重新劃分郵政區，那麼所有地址都將隨之改變。在這裡，地址是一個**ENTITY**。

在電力運營公司的軟件中，一個地址對應於公司線路和服務的一個目的地。如果幾個室友各自打電話申請電力服務，公司需要知道他們其實是住在同一個地方。在這種情況下，地址是一個**ENTITY**。換種方式，模型可以將電力服務與「住處」關聯起來，那麼住處就是一個帶有地址屬性的**ENTITY**了，這時，地址就是一個**VALUE OBJECT**。

顏色是很多現代開發系統的基礎庫所提供的**VALUE OBJECT**的一個例子，字符串和數字也是這樣的**VALUE OBJECT**（我們不會關心所使用的是哪一個,4'或哪一個,Q'）。這些基本的例子非常簡單，但**VALUE OBJECT**並不都這樣簡單。例如，調色程序可能有一個功能豐富的模型，在這個模型中，可以把功能更強的顏色對像組合起來產生其他顏色。這些顏色可能具有很複雜的算法，通過這些算法的共同計算得到新的**VALUE OBJECT**。

**VALUE OBJECT**可以是其他對象的集合。在房屋設計軟件中，可以為每種窗戶樣式創建一個對象。我們可以將，窗戶樣式'連同它的高度、寬度以及修改和組合這些屬性的規則一起放到，窗戶'對像中。這些窗戶就是由其他**VALUE OBJECT**組成的複雜**VALUE OBJECT**。它們進而又被合併到更大的設計元素中，如，牆'對象。

**VALUE OBJECT**甚至可以引用**ENTITY**。例如，如果我請在線地圖服務為我提供一個從舊金山到洛杉磯的駕車風景遊路線，它可能會得出一個，路線'對象，此對像通過太平洋海岸公路連接舊金山和洛杉磯。

這個'路線'對象是一個**VALUE**，儘管它所引用的3個對象（兩座城市和一條公路）都是**ENTITY**。

**VALUE OBJECT**經常作為參數在對像之間傳遞消息。它們常常是臨時對象，在一次操作中被創建，然後丟棄。**VALUE OBJECT**可以用作**ENTITY**（以及其他**VALUE**）的屬性。我們可以把一個人建模為一個具有標識的**ENTITY**，但這個人的名字是一個**VALUE**。

當我們只關心一個模型元素的屬性時，應把它歸類為**VALUE OBJECT**。我們應該使這個模型元素能夠表示出其屬性的意義，並為它提供相關功能。**VALUE OBJECT**應該是不可變的。不要為它分配任何標識，而且不要把它設計成像**ENTITY**那麼複雜。

**VALUE OBJECT**所包含的屬性應該形成一個概念整體[4]。例如，**street**（街道）、**city**（城市）和**postal code**（郵政編碼）不應是**Person**（人）對象的單獨的屬性。它們是整個地址的一部分，這樣可以使得**Person**對像更簡單，並使地址成為一個更一致的**VALUE OBJECT**，如圖5-6所示。

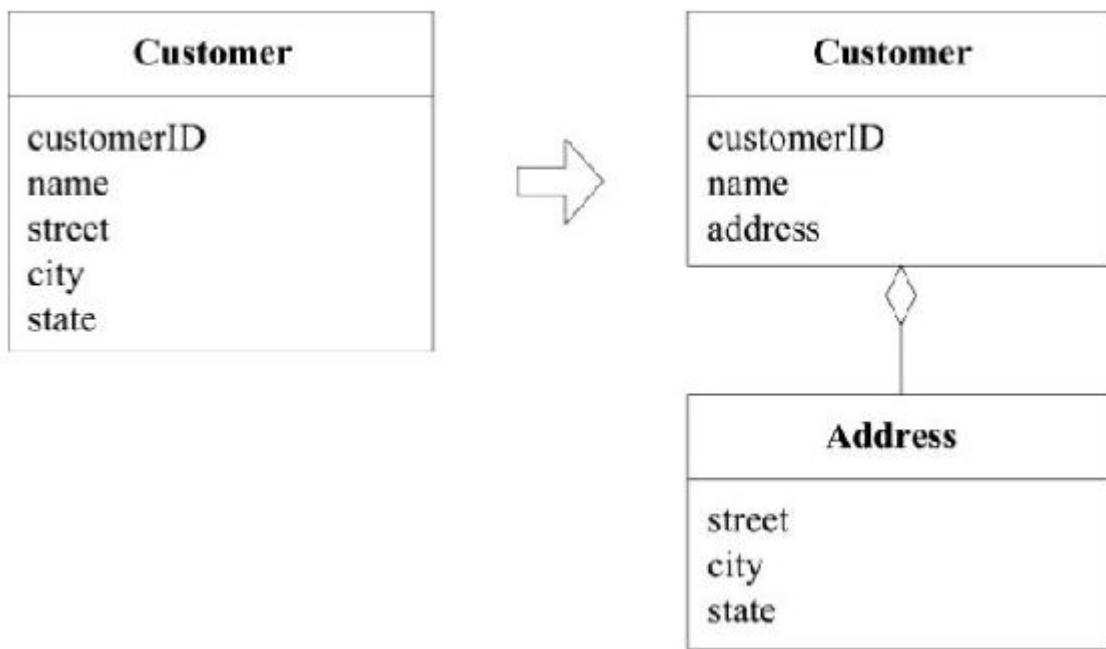


圖5-6 VALUE OBJECT可以提供一個ENTITY的相關信息，它在概念上應該是一個整體

### 5.3.1 設計VALUE OBJECT

我們並不關心使用的是VALUE OBJECT的哪個實例。由於不受這方面的約束，設計可以獲得更大的自由，因此可以簡化設計或優化性能。在設計VALUE OBJECT時有多種選擇，包括複製、共享或保持VALUE OBJECT不變。

兩個人同名並不意味著他們是同一個人，也不意味著他們是可互換的。但表示名字的對象是可以互換的，因為它們只涉及名字的拼寫。一個Name對象可以從第一個Person對像複製給第二個Person對象。

事實上，這兩個Person對象可能不需要自己的名字實例，它們可以共享同一個Name對像（其中每個Person對象都有一個指向同一個名字實例的指針），而無需改變它們的行為或標識。如此一來，當修改其中一個人名字時就會產生問題，這時另一個人的名字也將改變！為了防止這種錯誤發生，以便安全地共享一個對象，必須確保Name對象是不變的——它不能改變，除非將其整個替換掉。

當一個對像將它的一個屬性作為參數或返回值傳遞給另一個對像時，也會發生同樣的問題。一個脫離了其所有者控制的，流浪'對象可能會發生任何事情。VALUE的改變可能會破壞所有者的約束條件。這個問題可以通過傳遞一個不變對像或傳遞一個副本來解決。

VALUE OBJECT為性能優化提供了更多選擇，這一點可能很重要，因為VALUE OBJECT往往為數眾多。房屋設計軟件的示例就說明瞭這一點。如果每個電源插座都是一個單獨的VALUE OBJECT，那麼在一所房屋的一個設計版本中可能就會有上百個這種VALUE OBJECT。但如果把電源插座看成是可互換的，就只需共享一個電源插

座實例，並讓所有電源插座都指向這個實例（FLYWEIGHT，[Gamma et al.1995]中的一個示例）。在大型系統中，這種效果可能會被放大數千倍，而且這樣的優化可能決定一個系統是可用的，還是由於數百萬個多餘對像而變得異常緩慢。這只是無法應用於ENTITY的優化技巧中的一個。

複製和共享哪個更划算取決於實現環境。雖然複製有可能導致系統被大量的對象阻塞，但共享可能會減慢分佈式系統的速度。當在兩個機器之間傳遞一個副本時，只需發送一條消息，而且副本到達接收端後是獨立存在的。但如果共享一個實例，那麼只會傳遞一個引用，這要求每次交互都要向發送方返回一條消息。

以下幾種情況最好使用共享，這樣可以發揮共享的最大價值並最大限度地減少麻煩：

  節省數據庫空間或減少對像數量是一個關鍵要求時；

  通信開銷很低時（如在中央服務器中）；

  共享的對象被嚴格限定為不可變時。

在有些語言和環境中，可以將屬性或對像聲明為不可變的，但有些卻不具備這種能力。這種聲明能夠體現出設計決策，但它們並不是十分重要。我們在模型中所做的很多區別都無法用當前工具和編程語言在實現中顯式地聲明出來。例如，我們無法聲明ENTITY並自動確保其具有一個標識操作。但是，編程語言沒有直接支持這些概念上的區別並不說明這些區別沒有用處。這只是說明我們需要更多的約束機制來確保滿足一些重要的規則（這些規則只有在實現中才是隱式的）。命名規則、精心準備的文檔和大量討論都可以強化這些需求。

只要VALUE OBJECT是不可變的，變更管理就會很簡單，因為除了整體替換之外沒有其他的更改。不變的對象可以自由地共享，像在電源插座的例子中一樣。如果垃圾回收是可靠的，那麼刪除操作就只

是將所有指向對象的引用刪除。當在設計中將一個**VALUE OBJECT**指定為不可變時，開發人員就可以完全根據技術需求來決定是使用複製，還是使用共享，因為他們沒有後顧之憂——應用程序不依賴於對象的特殊實例。

### 特殊情況：何時允許可變性

保持**VALUE OBJECT**不變可以極大地簡化實現，並確保共享和引用傳遞的安全性。而且這樣做也符合值的意義。如果屬性的值發生改變，我們應該使用一個不同的**VALUE OBJECT**，而不是修改現有的**VALUE OBJECT**。儘管如此，在有些情況下出於性能考慮，仍需要讓**VALUE OBJECT**是可變的。這包括以下因素：

如果**VALUE**頻繁改變；

如果創建或刪除對象的開銷很大；

如果替換（而不是修改）將打亂集群（像前面示例中討論的那樣）；

如果**VALUE**的共享不多，或者共享不會提高集群性能，或其他某種技術原因。

再次強調：如果一個**VALUE**的實現是可變的，那麼就不能共享它。無論是否共享**VALUE OBJECT**，在可能的情況下都要將它們設計為不可變的。

定義**VALUE OBJECT**並將其指定為不可變的是一條一般規則，這樣做是為了避免在模型中產生不必要的約束，從而讓開發人員可以單純地從技術上優化性能。如果開發人員能夠顯式地定義重要約束，那麼他們就可以在對設計做出必要調整時，確保不會無意更改重要的行為。這樣的設計調整往往特定於具體項目所使用的技術。

### 示例 通過**VALUE OBJECT**來優化數據庫

數據庫——在其最底層——是將數據存儲到物理磁盤的一個具體位鎔上，或者花時間移動物理部件將數據讀取出來。高級數據庫則嘗試將這些物理地址聚集到一起，以便可以在一次物理操作中從磁盤讀取相互關聯的數據。

如果一個對像被許多對像引用，其中有些對像將不會在它附近（不在同一分頁上），這就需要通過額外的物理操作來獲取數據。通過複製（而不是共享對同一個實例的引用），可以將這種作為很多**ENTITY**屬性的**VALUE OBJECT**存儲在**ENTITY**所在的同一分頁上。這種存儲相同數據的多個副本的技術稱為非規範化（denormalization），當訪問時間比存儲空間或維護的簡單性更重要時，通常使用這種技術。

在關係數據庫中，我們可能想把一個具體的值放到擁有此值的**ENTITY**的表中，而不是將其關聯到另一個單獨的表。在分佈式系統中，對一個位於另一臺服務器上的**VALUE OBJECT**的引用可能導致對消息的響應十分緩慢，在這種情況下，應該將整個對象的副本傳遞到另一臺服務器上。我們可以隨意地使用副本，因為處理的是**VALUE OBJECT**。

### 5.3.2 設計包含**VALUE OBJECT**的關聯

前面討論的與關聯有關的大部分內容也適用於**ENTITY**和**VALUE OBJECT**。模型中的關聯越少越好，越簡單越好。

但是，如果說**ENTITY**之間的雙向關聯很難維護，那麼兩個**VALUE OBJECT**之間的雙向關聯則完全沒有意義。當一個**VALUE OBJECT**指向另一個**VALUE OBJECT**時，由於沒有標識，說一個對像指向的對象正是那個指向它的對象並沒有任何意義的。我們充其量只能說，一個對像指向的對象與那個指向它的對象是等同的，但這可能要求我們必須在某個地方實施這個固定規則。而且，儘管我們可以這樣做，並設鎔

雙向指針，但很難想出這種安排有什麼用處。因此，我們應盡量完全清除**VALUE OBJECT**之間的雙向關聯。如果在你的模型中看起來確實需要這種關聯，那麼首先應重新考慮一下將對像聲明為**VALUE OBJECT**這個決定是否正確。或許它擁有一個標識，而你還沒有注意到它。

**ENTITY**和**VALUE OBJECT**是傳統對像模型的主要元素，但一些注重實效的設計人員正逐漸開始使用一種新的元素——**SERVICE**。

## 5.4 模式：**SERVICE**



有時，對像不是一個事物。

在某些情況下，最清楚、最實用的設計會包含一些特殊的操作，這些操作從概念上講不屬於任何對象。與其把它們強制地歸於哪一

類，不如順其自然地在模型中引入一種新的元素，這就是**SERVICE**（服務）。

有些重要的領域操作無法放到**ENTITY**或**VALUE OBJECT**中。這當中有些操作從本質上講是一些活動或動作，而不是事物，但由於我們的建模範式是對象，因此要想辦法將它們劃歸到對像這個範疇裡。

現在，一個比較常見的錯誤是沒有努力為這類行為找到一個適當的對象，而是逐漸轉為過程化的編程。但是，當我們勉強將一個操作放到不符合對像定義的對象中時，這個對象就會產生概念上的混淆，而且會變得很難理解或重構。複雜的操作很容易把一個簡單對像搞亂，使對象的角色變得模糊。此外，由於這些操作常常會牽扯到很多領域對像——需要協調這些對像以便使它們工作，而這會產生對所有這些對象的依賴，將那些本來可以單獨理解的概念纏雜在一起。

有時，一些**SERVICE**看上去就像是模型對象，它們以對象的形式出現，但除了執行一些操作之外並沒有其他意義。這些，實幹家' ( Doer ) 的名字通常以,Manager' 之類的名字結尾。它們沒有自己的狀態，而且除了所承載的操作之外在領域中也沒有其他意義。儘管如此，該方法至少為這些特立獨行的行為找到了一個容身之所，避免它們擾亂真正的模型對象。

一些領域概念不適合被建模為對象。如果勉強把這些重要的領域功能歸為**ENTITY**或**VALUE OBJECT**的職責，那麼不是歪曲了基於模型的對象的定義，就是人為地增加了一些無意義的對象。

**SERVICE**是作為接口提供的一種操作，它在模型中是獨立的，它不像**ENTITY**和**VALUE OBJECT**那樣具有封裝的狀態。**SERVICE**是技術框架中的一種常見模式，但它們也可以在領域層中使用。

所謂**SERVICE**，它強調的是與其他對象的關係。與**ENTITY**和**VALUE OBJECT**不同，它只是定義了能夠為客戶做什麼。**SERVICE**往往

是以一個活動來命名，而不是以一個**ENTITY**來命名，也就是說，它是動詞而不是名詞。**SERVICE**也可以有抽象而有意義的定義，只是它使用了一種與對像不同的定義風格。**SERVICE**也應該有定義的職責，而且這種職責以及履行它的接口也應該作為領域模型的一部分來加以定義。操作名稱應來自於**UBIQUITOUS LANGUAGE**，如果**UBIQUITOUS LANGUAGE**中沒有這個名稱，則應該將其引入到**UBIQUITOUS LANGUAGE**中。參數和結果應該是領域對象。

使用**SERVICE**時應謹慎，它們不應該替代**ENTITY**和**VALUE OBJECT**的所有行為。但是，當一個操作實際上是一個重要的領域概念時，**SERVICE**很自然就會成為**MODEL-DRIVEN DESIGN**中的一部分。將模型中的獨立操作聲明為一個**SERVICE**，而不是聲明為一個不代表任何事情的虛擬對象，可以避免對任何人產生誤導。

好的**SERVICE**有以下3個特徵。

(1) 與領域概念相關的操作不是**ENTITY**或**VALUE OBJECT**的一個自然組成部分。

(2) 接口是根據領域模型的其他元素定義的。

(3) 操作是無狀態的。

這裡所說的無狀態是指任何客戶都可以使用某個**SERVICE**的任何實例，而不必關心該實例的歷史狀態。**SERVICE**執行時將使用可全局訪問的信息，甚至會更改這些全局信息（也就是說，它可能具有副作用）。但**SERVICE**不保持影響其自身行為的狀態，這一點與大多數領域對象不同。

當領域中的某個重要的過程或轉換操作不是**ENTITY**或**VALUE OBJECT**的自然職責時，應該在模型中添加一個作為獨立接口的操作，並將其聲明為**SERVICE**。定義接口時要使用模型語言，並確保操

作名稱是 **UBIQUITOUS LANGUAGE** 中的術語。此外，應該使 **SERVICE** 成為無狀態的。

### **5.4.1 SERVICE與孤立的領域層**

這種模式只重視那些在領域中具有重要意義的 **SERVICE**，但 **SERVICE** 並不只是在領域層中使用。我們需要注意區分屬於領域層的 **SERVICE** 和那些屬於其他層的 **SERVICE**，並劃分責任，以便將它們明確地區分開。

文獻中所討論的大多數 **SERVICE** 是純技術的 **SERVICE**，它們都屬於基礎設施層。領域層和應用層的 **SERVICE** 與這些基礎設施層 **SERVICE** 進行協作。例如，銀行可能有一個用於向客戶發送電子郵件的應用程序，當客戶的賬戶餘額小於一個特定的臨界值時，這個程序就向客戶發送一封電子郵件。封裝了電子郵件系統的接口（也可能是其他的通知方式）就是基礎設施層中的 **SERVICE**。

應用層 **SERVICE** 和領域層 **SERVICE** 可能很難區分。應用層負責通知的設置，而領域層負責確定是否滿足臨界值，儘管這項任務可能並不需要使用 **SERVICE**，因為它可以作為 *account*（賬戶）對象的職責中。這個銀行應用程序可能還負責資金轉賬。如果設計一個 **SERVICE** 來處理資金轉賬相應的借方和貸方，那麼這項功能將屬於領域層。資金轉賬在銀行領域語言中是一項有意義的操作，而且它涉及基本的業務邏輯。而純技術的 **SERVICE** 應該沒有任何業務意義。

很多領域或應用層 **SERVICE** 是在 **ENTITY** 和 **VALUE OBJECT** 的基礎上建立起來的，它們的行為類似於將領域的一些潛在功能組織起來以執行某種任務的腳本。**ENTITY** 和 **VALUE OBJECT** 往往由於粒度過細而無法提供對領域層功能的便捷訪問。我們在這裡會遇到領域層與應用層之間很微妙的分界線。例如，如果銀行應用程序可以把我們的交易進行轉換並導出到一個電子錶格文件中，以便進行分析，那麼這個導

出操作就是應用層SERVICE。,文件格式'在銀行領域中是沒有意義的，它也不涉及業務規則。

另一方面，賬戶之間的轉賬功能屬於領域層SERVICE，因為它包含重要的業務規則（如處理相應的借方賬戶和貸方賬戶），而且，資金轉賬'是一個有意義的銀行術語。在這種情況下，SERVICE自己並不會做太多的事情，而只是要求兩個Account對像完成大部分工作。但如果將，轉賬'操作強加在Account對像上會很彆扭，因為這個操作涉及兩個賬戶和一些全局規則。

我們可能喜歡創建一個Funds Transfer（資金轉賬）對像來表示兩個賬戶，外加一些與轉賬有關的規則和歷史記錄。但在銀行間的網絡中進行轉賬時，仍然需要使用SERVICE。此外，在大多數開發系統中，在一個領域對像和外部資源之間直接建立一個接口是很彆扭的。我們可以利用一個FAÇADE（外觀）[\[5\]](#)將這樣的外部SERVICE包裝起來，這個外觀可能以模型作為輸入，並返回一個，Funds Transfer'對像（作為它的結果）。但無論中間涉及什麼SERVICE，甚至那些超出我們掌控範圍的SERVICE，這些SERVICE都是在履行資金轉賬的領域職責。

### 將SERVICE劃分到各個層中

应用层	资金转账应用服务 <input type="checkbox"/> 获取输入（如一个XML请求） <input type="checkbox"/> 发送消息给领域层服务，要求其执行 <input type="checkbox"/> 监听确认消息 <input type="checkbox"/> 决定使用基础设施SERVICE来发送通知
领域层	资金转账领域服务 <input type="checkbox"/> 与必要的Account（账户）和Ledger（总账）对像进行交互，执行相应的借入和贷出操作 <input type="checkbox"/> 提供结果的确认（允许转帐或拒绝转帐等）
基础设施层	发送通知服务 <input type="checkbox"/> 按照应用程序的指示发送电子邮件、信件和其他信息

### 5.4.2 粒度

上述對SERVICE的討論強調的是將一個概念建模為SERVICE的表現力，但SERVICE還有其他有用的功能，它可以控制領域層中的接口的粒度，並且避免客戶端與ENTITY和VALUE OBJECT耦合。

在大型系統中，中等粒度的、無狀態的SERVICE更容易被復用，因為它們在簡單的接口背後封裝了重要的功能。此外，細粒度的對象可能導致分佈式系統的消息傳遞的效率低下。

如前所述，由於應用層負責對領域對象的行為進行協調，因此細粒度的領域對象可能會把領域層的知識洩漏到應用層中。這產生的結果是應用層不得不處理複雜的、細緻的交互，從而使得領域知識蔓延到應用層或用戶界面代碼當中，而領域層會丟失這些知識。明智地引入領域層服務有助於在應用層和領域層之間保持一條明確的界限。

這種模式有利於保持接口的簡單性，便於客戶端控制並提供了多樣化的功能。它提供了一種在大型或分佈式系統中便於對組件進行打包的中等粒度的功能。而且，有時SERVICE是表示領域概念的最自然的方式。

### 5.4.3 對SERVICE的訪問

像J2EE和CORBA這樣的分佈式系統架構提供了特殊的SERVICE發佈機制，這些發佈機制具有一些使用上的慣例，並且增加了發佈和訪問功能。但是，並非所有項目都會使用這樣的框架，即使在使用了它們的時候，如果只是為了在邏輯上實現關注點的分離，那麼它們也是大材小用了。

與分離特定職責的設計決策相比，提供對SERVICE的訪問機制的意義並不是十分重大。一個'操作'對象可能足以作為SERVICE接口的實現。我們很容易編寫一個簡單的SINGLETON對像[Gamma et al.1995]來實現對SERVICE的訪問。從編碼慣例可以明顯看出，這些對像只是SERVICE接口的提供機制，而不是有意義的領域對象。只有當真正需

要實現分佈式系統或充分利用框架功能的情況下才應該使用複雜的架構。

## 5.5 模式：MODULE ( 也稱為PACKAGE )

MODULE是一個傳統的、較成熟的設計元素。雖然使用模塊有一些技術上的原因，但主要原因卻是，認知超載<sup>[6]</sup>。MODULE為人們提供了兩種觀察模型的方式，一是可以在MODULE中查看細節，而不會被整個模型淹沒，二是觀察MODULE之間的關係，而不考慮其內部細節。

領域層中的MODULE應該成為模型中有意義的部分，MODULE從更大的角度描述了領域。

每個人都會使用MODULE，但卻很少有人把它們當做模型中的一個成熟的組成部分。代碼按照各種各樣的類別進行分解，有時是按照技術架構來分割的，有時是按照開發人員的任務分工來分割的。甚至那些從事大量重構工作的開發人員也傾向於使用項目早期形成的一些MODULE。

眾所周知，MODULE之間應該是低耦合的，而在MODULE的內部則是高內聚的。耦合和內聚的解釋使得MODULE聽上去像是一種技術指標，彷彿是根據關聯和交互的分佈情況來機械地判斷它們。然而，MODULE並不僅僅是代碼的劃分，而且也是概念的劃分。一個人一次考慮的事情是有限的（因此才要低耦合）。不連貫的思想和「一鍋粥」似的思想同樣難於理解（因此才要高內聚）。

低耦合高內聚作為通用的設計原則既適用於各種對象，也適用於MODULE，但MODULE作為一種更粗粒度的建模和設計元素，採用低

耦合高內聚原則顯得更為重要。這些術語由來已久，早在[Larman 1998]中就從模式角度對其進行了解釋。

只要兩個模型元素被劃分到不同的MODULE中，它們的關係就不如原來那樣直接，這會使我們更難理解它們在設計中的作用。MODULE之間的低耦合可以將這種負面作用減至最小，而且在分析一個MODULE的內容時，只需很少地參考那些與之交互的其他MODULE。

同時，在一個好的模型中，元素之間是要協同工作的，而仔細選擇的MODULE可以將那些具有緊密概念關係的模型元素集中到一起。將這些具有相關職責的對象元素聚合到一起，可以把建模和設計工作集中到單一MODULE中，這會極大地降低建模和設計的複雜性，使人們可以從容應對這些工作。

MODULE和較小的元素應該共同演變，但實際上它們並不是這樣。MODULE被用來組織早期對象。在這之後，對像在變化時不脫離現有模塊定義的邊界。重構MODULE需要比重構類做更多工作，也具有更大的破壞性，並且可能不會特別頻繁。但就像模型對像從簡單具體逐漸轉變為反映更深層次的本質一樣，MODULE也會變得微妙和抽象。讓MODULE反映出對領域理解的不斷變化，可以使MODULE中的對象能夠更自由地演變。

像領域驅動設計中的其他元素一樣，MODULE是一種表達機制。MODULE的選擇應該取決於被劃分到模塊中的對象的意義。當你將一些類放到MODULE中時，相當於告訴下一位看到你的設計的開發人員要把這些類放在一起考慮。如果說模型講述了一個故事，那麼MODULE就是這個故事的各個章節。模塊的名稱表達了其意義。這些名稱應該被添加到UBIQUITOUS LANGUAGE中。你可能會向一位業務

專家說,現在讓我們討論一下『客戶』模塊'，這就為你們接下來的對話設定了上下文。

因此：

選擇能夠描述系統的**MODULE**，並使之包含一個內聚的概念集合。這通常會實現**MODULE**之間的低耦合，但如果效果不理想，則應尋找一種更改模型的方式來消除概念之間的耦合，或者找到一個可作為**MODULE**基礎的概念（這個概念先前可能被忽視了），基於這個概念組織的**MODULE**可以以一種有意義的方式將元素集中到一起。找到一種低耦合的概念組織方式，從而可以相互獨立地理解和分析這些概念。對模型進行精化，直到可以根據高層領域概念對模型進行劃分，同時相應的代碼也不會產生耦合。

**MODULE**的名稱應該是**UBIQUITOUS LANGUAGE**中的術語。  
**MODULE**及其名稱應反映出領域的深層知識。

僅僅研究概念關係是不夠的，它並不能替代技術措施。這二者是相同問題的不同層次，都是必須要完成的。但是，只有以模型為中心進行思考，才能得到更深層次的解決方案，而不是隨便找一個解決方案應付了事。當必須做出一個折中選擇時，務必保證概念清晰，即使這意味著**MODULE**之間會產生更多引用，或者更改**MODULE**偶爾會產生,漣漪效應'。開發人員只要理解了模型所描述的內容，就可以應付這些問題。

### 5.5.1 敏捷的**MODULE**

**MODULE**需要與模型的其他部分一同演變。這意味著**MODULE**的重構必須與模型和代碼一起進行。但這種重構通常不會發生。更改**MODULE**可能需要大範圍地更新代碼。這些更改可能會對團隊溝通起到破壞作用，甚至會妨礙開發工具（如源代碼控制系統）的使用。因

此，**MODULE**結構和名稱往往反映了模型的較早形式，而類則不是這樣。

在**MODULE**選擇的早期，有些錯誤是不可避免的，這些錯誤導致了高耦合，從而使**MODULE**很難進行重構。而缺乏重構又會導致問題變得更加嚴重。克服這一問題的唯一方法是接受挑戰，仔細地分析問題的要害所在，並據此重新組織**MODULE**。

一些開發工具和編程系統會使問題變得更加嚴重。無論在實現中採用哪種開發技術，我們要想盡一切辦法來減少重構**MODULE**的工作量，並最大限度地減少與其他開發人員溝通時出現的混亂情況。

### 示例 **Java**中的包編碼慣例

在**Java**中，類使用**import**語句來聲明依賴。建模人員可能認為有些包會依賴其他的包，但在**Java**中無法說明這一點。常見的編碼慣例鼓勵導入具體的類，如以下代碼所示：

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

```

遺憾的是，在**Java**中，我們不可避免地需要在類中使用**import**聲明依賴，但至少可以一次導入一個完整的包，這既反映出包是一種高內聚的單元，同時又減少了更改包名稱的工作量。

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

```

的確，這種技術意味著把類和包混在一起（類依賴於包），但它除了表達前面一長串類的列表之外，還表達了在具體MODULE上建立一種依賴性的意圖。

如果一個類確實依賴於另一個包中的某個類，而且本地MODULE對該MODULE並沒有概念上的依賴關係，那麼或許應該移動一個類，或者考慮重新組織MODULE。

### 5.5.2 通過基礎設施打包時存在的隱患

技術框架對打包決策有著極大的影響，有些技術框架是有幫助的，有些則要堅決抵制。

一個非常有用的框架標準是LAYERED ARCHITECTURE，它將基礎設施和用戶界面代碼放到兩組不同的包中，並且從物理上把領域層隔離到它自己的一組包中。

但從另一個方面看，分層架構可能導致模型對像實現的分裂。一些框架的分層方法是把一個領域對象的職責分散到多個對象當中，然後把這些對像放到不同的包中。例如，當使用J2EE早期版本時，一種常見的做法是把數據和數據訪問放到實體bean中，而把相關的業務邏輯放到會話bean中。這樣做除了導致每個組件的實現變得更複雜以外，還破壞了對象模型的內聚性。對象的一個最基本的概念是將數據和操作這些數據的邏輯封裝在一起。由於我們可以把這兩個組件看作是一起組成一個單一模型元素的實現，因此這種分層實現還不算是致命的。但實體bean和會話bean通常被隔離到不同的包中，從而使情

況變得更糟。在這種情況下，通過查看若干對象並把它們腦補成單一的概念**ENTITY**是非常困難的。我們失去了模型與設計之間的聯繫。最好的做法是在比**ENTITY**對像更大的粒度上應用**EJB**，從而減少分層的副作用。但細粒度的對象通常也會被分層。

例如，我就曾經在一個籌劃得相當不錯的項目上遇到過這些問題，這個項目的每個概念模型實際上被分為**4層**。每個層的劃分都有很好的理由。第一層是數據持久層，負責處理映射和訪問關係數據庫。第二層負責處理對像在所有情況下的固有行為。第三層放置特定於應用程序的功能。第四層是一個公共接口，它隱藏了第一、二、三層的所有實現細節。這種分層方案有些複雜，但每層都有很好的定義，而且清楚地實現了關注點的分離。我們可以在大腦中將所有物理對像連接到一起，組成一個概念對象。有時，方面的分離也是有幫助的。具體來講，把持久化代碼移出來可以減少很多混亂。

但最重要的是，這個項目的框架要求將每個層放到單獨的一組包中，並根據層的標識慣例來命名。這一下子就把我們所有的注意力都吸引到分層上來。結果，領域開發人員盡量避免創建太多的**MODULE**（每個模塊都要乘以**4**），而且幾乎不能更改模塊，因為重構**MODULE**的工作量不允許這樣做。更糟的是，由於很難跟蹤定義了一個概念類的所有數據和行為（而且還要考慮分層產生的間接關係），因此開發人員沒有多少精力思考模型了。這個應用最終交付使用了，但它使用了貧血領域模型，只是基本滿足了應用程序的數據庫訪問需求，此外通過很少的幾個**SERVICE**提供了一些行為。這個項目從**MODEL-DRIVEN DESIGN**獲得的益處十分有限，因為代碼並沒有清晰地揭示模型，因此開發人員也無法充分地利用模型。

這種框架設計是在嘗試解決兩個合理的問題。一個問題是關注點的邏輯劃分：一個對像負責數據庫訪問，另外一個對像負責處理業務

邏輯，等等。這種劃分方法使人們更容易（在技術層面上）理解每個層的功能，而且更容易切換各個層。這種設計的問題在於沒有顧及應用程序的開發成本。本書不是討論框架設計的書，因此不會給出此問題的替代解決方案，但它們確實存在。而且，即使別無選擇，也值得犧牲一些分層的好處來換取更內聚的領域層。

這些打包方案的另一個動機是層的分佈。如果代碼實際上被部署到不同的服務器上，那麼這會成為這種分層的有力論據。但通常並不是這樣。應該在需要時才尋求靈活性。在一個希望充分利用**MODEL-DRIVEN DESIGN**的項目上，這種分層設計的犧牲太大了，除非它是為瞭解決一個緊迫的問題。

精巧的技術打包方案會產生如下兩個代價。

如果框架的分層慣例把實現概念對象的元素分得很零散，那麼代碼將無法再清楚地表示模型。

人的大腦把劃分後的東西還原成原樣的能力是有限的，如果框架把人的這種能力都耗盡了，那麼領域開發人員就無法再把模型還原成有意義的部分了。

最好把事情變簡單。要極度簡化技術分層規則，要麼這些規則對技術環境特別重要，要麼這些規則真正有助於開發。例如，將複雜的數據持久化代碼從對象的行為方面提取出來可以使重構變得更簡單。

**除非真正有必要將代碼分佈到不同的服務器上，否則就把實現單一概念對象的所有代碼放在同一個模塊中（如果不能放在同一個對像中的話）。**

從傳統的，高內聚、低耦合'標準也可以得出相同的結論。實現業務邏輯的對象與負責數據庫訪問的對象之間的聯繫非常廣泛，因此它們之間的耦合度很高。

在框架設計中，或者在公司或項目的工作慣例方面，可能還有其他一些隱患，這些隱患可能會妨礙領域模型的自然內聚性，從而破壞模型驅動的設計，但所有隱患的基本問題都是相同的。種種限制（或者只是由於所需的包太多了）使我們無法使用專門根據領域模型需要量身定做的其他打包方案。

**利用打包把領域層從其他代碼中分離出來。否則，就盡可能讓領域開發人員自由地決定領域對象的打包方式，以便支持他們的模型和設計選擇。**

如果代碼是基於聲明式設計（第10章有這方面的討論）生成的，則是一種例外情況。在這種情況下，開發人員無需閱讀代碼，因此為了不礙事最好將代碼放到一個單獨的包中，這樣就不會搞亂開發人員實際要處理的設計元素。

隨著設計規模和複雜度的增加，模塊化變得更加重要。本節只是介紹了一些基本的注意事項。本書第四部分主要介紹打包方法以及分解大型模型和設計的方法，並介紹如何抓住重點以幫助理解問題。

領域模型中的每個概念都應該在實現元素中反映出來。**ENTITY**、**VALUE OBJECT**、它們之間的關聯、領域**SERVICE**以及用於組織元素的**MODULE**都是實現與模型直接對應的地方。實現中的對象、指針和檢索機制必須直接、清楚地映射到模型元素。如果沒有做到這一點，就要重寫代碼，或者回頭修改模型，或者同時修改代碼和模型。

不要在領域對像中添加任何與領域對像所表示的概念沒有緊密關係的元素。領域對象的職責是表示模型。當然，其他一些與領域有關的職責也是必須要實現的，而且為了使系統工作，也必須管理其他數據，但它們不屬於領域對象。第6章將討論一些支持對象，這些對像履行領域層的技術職責，如定義數據庫搜索和封裝複雜的對象創建。

本章介紹的4種模式為對像模型提供了構造塊。但 MODEL-DRIVEN DESIGN並不是說必須將每個元素都建模為對象。一些工具還支持其他的模型範式，如規則引擎。項目需要在它們之間做出契合實際的折中選擇。這些其他的工具和技術是MODEL-DRIVEN DESIGN的補充，而不是要取而代之。

## 5.6 建模範式

MODEL-DRIVEN DESIGN要求使用一種與建模範式協調的實現技術。人們曾經嘗試了大量的建模範式，但在實踐中只有少數幾種得到了廣泛應用。目前，主流的範式是面向對像設計，而且現在的大部分複雜項目都開始使用對象。這種範式的流行有許多原因，包括對像本身的固有因素、一些環境因素，以及廣泛使用所帶來的一些優勢。

### 5.6.1 對像範式流行的原因

一些團隊選擇對像範式並不是出於技術上的原因，甚至也不是出於對像本身的原因，而是從一開始，對像建模就在簡單性和複雜性之間實現了一個很好的平衡。

如果一個建模範式過於深奧，那麼大多數開發人員可能無法掌握它，因此也無法正確地運用它。如果團隊中的非技術人員無法掌握範式的基本知識，那麼他們將無法理解模型，以至於無法建立 UBIQUITOUS LANGUAGE。大部分人都比較容易理解面向對像設計的基本知識。儘管一些開發人員還沒有完全領悟建模的奧妙，但即使是非專業人員也可以理解對像模型圖。

然而，雖然對像建模的概念很簡單，但它的豐富功能足以捕獲重要的領域知識。而且它從一開始就獲得了開發工具的支持，使得模型可以在軟件中表達出來。

現在，對像範式已經發展很成熟並得到了廣泛採用，這使得它具有明顯的優勢。項目如果沒有成熟的基礎設施和工具支持，可能就要在這些方面進行研發工作，這不僅會耽誤應用程序的開發，分散應用程序的開發資源，還會帶來技術風險。有些技術不能與其他技術很好地協同工作，而且它們可能也無法與行業標準解決方案集成，這使團隊不得不重新開發一些常用的輔助工具。但近年來，很多這樣的問題已經在對像領域得以解決，而且有些問題也隨著對像範式的廣泛採用而變得無關緊要（現在，對像技術已經成為主流，因此集成的任務已經落到其他方法的肩上）。大多數新技術都提供了與主流的面向對像平臺進行集成的方式。這使得集成更容易，甚至允許將基於其他建模範式的子系統混合在一起（本章稍後將討論）。

開發者社區和設計文化的成熟也同樣重要。採用新範式的項目可能很難找到精通它的開發人員，也很難找到能夠使用新範式創建有效模型的人員。要想在短時間內培訓開發人員使用新範式往往是行不通的，因為能夠最大限度地利用新範式和技術的模式尚未形成。或許新領域的一些開拓者已經可以有效地使用新範式，但他們尚未發佈可供人們學習的知識。

而對像範式則不同，大多數開發人員、項目經理和從事項目工作的其他專家都已經很瞭解它。

下面我講一個10年前在一個面向對像項目中發生的小故事，它說明瞭在工作中使用不成熟範式所產生的風險。這個項目是在20世紀90年代早期開始的，它採用了幾種當時最前沿的技術，包括大規模使用面向對像數據庫。當時這讓人很興奮。團隊成員驕傲地告訴訪客他們正在部署迄今為止最大的面向對像數據庫。當我加盟這個項目時，各個團隊正在研究一些面向對象的設計，並且可以毫不費力地將對像存儲在數據庫中。但我們漸漸意識到，大部分數據庫容量已經被耗盡

了，而這僅僅只輸入了測試數據而已！實際所需的數據庫還要大幾十倍。實際的事務量也要大上幾十倍。是不是這個應用程序根本不適合使用面向對像數據庫？是我們使用不當嗎？我們已經力不從心了。

幸運的是，我們找到了一位精通對像數據庫技術的專家來幫助我們擺脫困境。我們談妥服務價格後，他指出了3個問題根源。首先，與數據庫一起提供的基礎設施沒有擴展到我們所需的規模。其次，細粒度對象的存儲比我們預計的代價要大得多。最後，對像模型的有些部分其內部依賴過於複雜，以至於很少的並發事務就會產生競爭問題。

在這位專家的幫助下，我們對基礎設施進行了強化。現在，項目團隊意識到細粒度對象的影響，並開始尋找更適合對像數據庫的模型。所有人員都深刻認識到對模型中的關係進行限制的重要性，我們利用這種新的理解開始設計更好的模型——將原來那些緊密聯繫在一起的對象解耦。

除了前幾個月浪費在錯誤路線上以外，項目的修復又損失了好幾個月的時間。而且這並不是團隊由於選擇了不成熟的技術和沒有相關經驗而遭遇的第一個挫折。遺憾的是，這個項目最終被削減了，而且變得十分保守。直到今天，他們雖然仍會使用一些外來技術，但在應用範圍上變得謹小慎微，這導致他們可能無法真正從這些技術中獲益。

十年過去了，面向對像技術已經相對成熟。業內已經提供了很多現成的解決方案，它們可以滿足大部分常見的基礎設施需要。多數大型供應商，或者穩定的開源項目都提供了關鍵工具。這些基礎設施本身就已經被廣泛使用，因此瞭解它們的人很多，相關書籍也很多，等等。人們已經相當瞭解這些成熟技術的侷限性，因此內行團隊也不會過度使用它們。

其他一些令人感興趣的建模範式並沒有這麼成熟。有些建模範式太難掌握了，以至於只能在很小的專業領域內使用。有些建模範式雖然有潛力，但技術基礎設施仍然不夠完整、可靠，而且很少有人理解為這些範式創建良好模型的訣竅。這些範式可能已經出現很長一段時間了，但仍然不適合用於大多數項目。

這就是目前大部分採用**MODEL-DRIVEN DESIGN**的項目很明智地使用面向對像技術作為系統核心的原因。它們不會被束縛在只有對象的系統裡，因為對像已經成為內業的主流技術，人們目前使用的幾乎所有的技術都有與之對應的集成工具。

然而，這並不意味著人們就應該永遠只侷限於對像技術。隨大流具有一定的安全性，但這並非總是應該走的道路。對像模型可以解決很多實際的軟件問題，但也有一些領域不適合用封裝了行為的各種對像來建模。例如，涉及大量數學問題的領域或者受全局邏輯推理控制的領域就不適合使用面向對象的範式。

### **5.6.2 對像世界中的非對像**

領域模型不一定是對像模型。例如，使用**Prolog**語言實現的**MODEL-DRIVEN DESIGN**，它的模型是由邏輯規則和事實構成的。模型範式為人們提供了思考領域的方式。這些領域的模型由範式塑造成型。結果就得到了遵守範式的模型，這樣的模型可以用支持對應建模風格的工具來有效地實現。

不管在項目中使用哪種主要的模型範式，領域中都會有一些部分更容易用某種其他範式來表達。當領域中只有個別元素適合用其他範式時，開發人員可以接受一些蹩腳的對象，以使整個模型保持一致（或者，在另一種極端的情況下，如果大部分問題領域都更適合用其他範式來表達，那麼可以整個改為使用那種範式，並選擇一個不同的實現平臺）。但是，當領域的主要部分明顯屬於不同的範式時，明智

的做法是用適合各個部分的範式對其建模，並使用混合工具集來進行實現。當領域的各個部分之間的互相依賴性較小時，可以把用另一種範式建立的子系統封裝起來，例如，只有一個對像需要調用的複雜數學計算。其他時候，不同方面之間的關係更為複雜，例如，對象的交互依賴於某些數學關係的時候。

這就是將業務規則引擎或工作流引擎這樣的非對像組件集成到對像系統中的動機。混合使用不同的範式使得開發人員能夠用最適當的風格對特殊概念進行建模。此外，大部分系統都必須使用一些非對象的技術基礎設施，最常見的就是關係數據庫。但是在使用不同的範式後，要想得到一個內聚的模型就比較難了，而且讓不同的支持工具共存也較為複雜。當開發人員在軟件中無法清楚地辨認出一個內聚的模型時，**MODEL-DRIVEN DESIGN**就會被拋諸腦後，儘管這種混合設計更需要它。

### **5.6.3 在混合範式中堅持使用**MODEL-DRIVEN DESIGN****

在面向對象的應用程序開發項目中，有時會混合使用一些其他的技術，規則引擎就是一個常見的例子。一個包含豐富知識的領域模型可能會含有一些顯式的規則，然而對像範式卻缺少用於表達規則和規則交互的具體語義。儘管可以將規則建模為對像（而且常常可以成功地做到），但對像封裝卻使得那些針對整個系統的全局規則很難應用。規則引擎技術非常有吸引力，因為它提供了一種更自然、聲明式的規則定義方式，能夠有效地將規則範式融合到對像範式中。邏輯範式已經得到了很好的發展並且功能強大，它是對像範式的很好補充，使其可以揚長避短。

但人們並不總是能夠從規則引擎的使用中得到預期結果。有些產品並不能很好地工作。有些則缺少一種能夠顯示出銜接兩種實現環境的模型概念相關性的無縫視圖。一個常見的結果是應用程序被割裂成

兩部分：一個是使用了對象的靜態數據存儲系統，另一個是幾乎完全與對像模型失去聯繫的某種規則處理應用程序。

重要的是在使用規則的同時要繼續考慮模型。團隊必須找到能夠同時適用於兩種實現範式的單一模型。雖然這並非易事，但還是可以辦到的，條件是規則引擎支持富有表達力的實現方式。如果不這樣，數據和規則就會失去聯繫。與領域模型中的概念規則相比，引擎中的規則更像是一些較小的程序。只有保持規則與對像之間緊密、清晰的關係，才能確保顯示出這二者所表達的含義。

如果沒有無縫的環境，就要完全靠開發人員提煉出一個由清晰的基本概念組成的模型，以便完全支撐整個設計。

將各個部分緊密結合在一起的最有效工具就是健壯的 **UBIQUITOUS LANGUAGE**，它是構成整個異構模型的基礎。堅持在兩種環境中使用一致的名稱，堅持用 **UBIQUITOUS LANGUAGE** 討論這些名稱，將有助於消除兩種環境之間的鴻溝。

這個話題本身就值得寫一本書了。本節的目的只是想說明（在使用其他範式時）沒有必要放棄 **MODEL-DRIVEN DESIGN**，而且堅持使用它是值得的。

雖然 **MODEL-DRIVEN DESIGN** 不一定是面向對象的，但它確實需要一種富有表達力的模型結構實現，無論是對像、規則還是工作流，都是如此。如果可用工具無法提高表達力，就要重新考慮選擇工具。缺乏表達力的實現將削弱各種範式的優勢。

當將非對像元素混合到以面向對像為主的系統中時，需要遵循以下4條經驗規則。

不要和實現範式對抗。我們總是可以用別的方式來考慮領域。找到適合於範式的模型概念。

把通用語言作為依靠的基礎。即使工具之間沒有嚴格聯繫時，語言使用上的高度一致性也能防止各個設計部分分裂。

不要一味依賴UML。有時固定使用某種工具（如UML繪圖工具）將導致人們通過歪曲模型來使它更容易畫出來。例如，UML確實有一些特性很適合表達約束，但它並不是在所有情況下都適用。有時使用其他風格的圖形（可能適用於其他範式）或者簡單的語言描述比牽強附會地適應某種對像視圖更好。

保持懷疑態度。工具是否真正有用武之地？不能因為存在一些規則，就必須使用規則引擎。規則也可以表示為對象，雖然可能不是特別優雅。多個範式會使問題變得非常複雜。

在決定使用混合範式之前，一定要確信主要範式中的各種可能性都已經嘗試過了。儘管有些領域概念不是以明顯的對象形式表現出來的，但它們通常可以用對像範式來建模。第9章將討論如何使用對像技術對一些非常規類型的概念進行建模。

關係範式是範式混合的一個特例。作為一種最常用的非對像技術，關係數據庫與對像模型的關係比其他技術與對像模型的關係更緊密，因為它作為一種數據持久存儲機制，存儲的就是對象。第6章將討論用關係數據庫來存儲對像數據，並介紹在對像生命週期中將會遇到的諸多挑戰。

## 第6章 領域對象的生命週期

每個對象都有生命週期，如圖6-1所示。對像自創建後，可能會經歷各種不同的狀態，直至最終消亡——要麼存檔，要麼刪除。當然，很多對象是簡單的臨時對象，僅通過調用構造函數來創建，用來做一些計算，而後由垃圾收集器回收。這類對像沒必要搞得那麼複雜。但有些對像具有更長的生命週期，其中一部分時間不是在活動內存中度過的。它們與其他對像具有複雜的相互依賴性。它們會經歷一些狀態變化，在變化時要遵守一些固定規則。管理這些對像時面臨諸多挑戰，稍有不慎就會偏離MODEL-DRIVEN DESIGN的軌道。

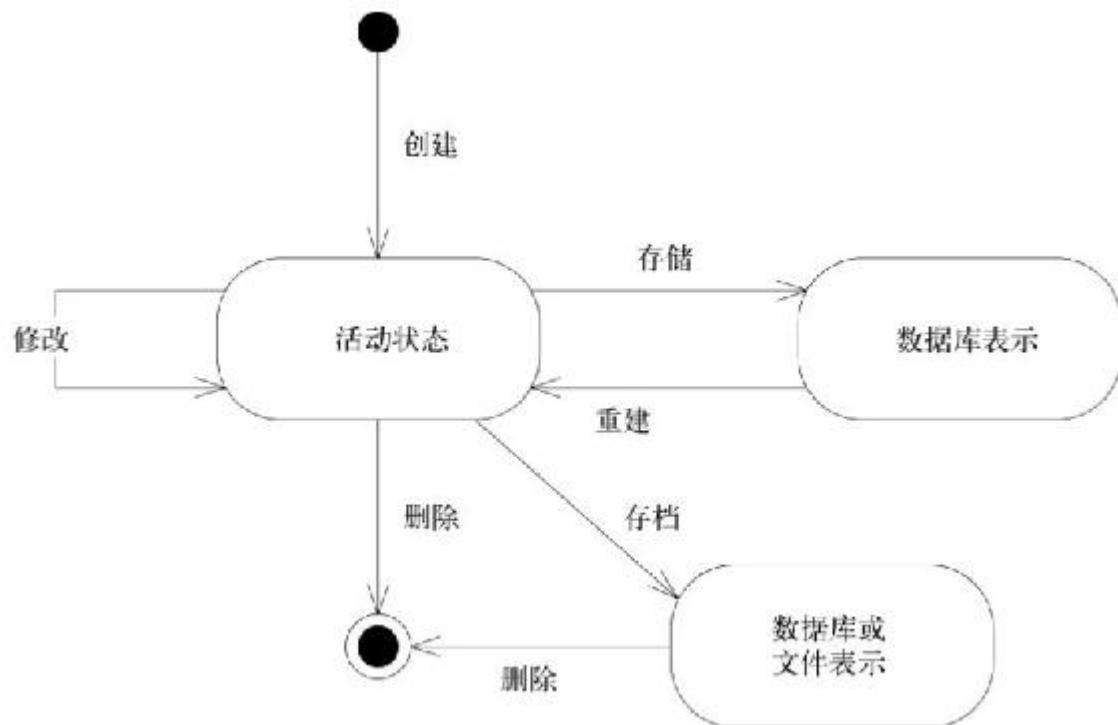


圖6-1 領域對象的生命週期

主要的挑戰有以下兩類。

(1) 在整個生命週期中維護完整性。

(2) 防止模型陷入管理生命週期複雜性造成的困境當中。

本章將通過3種模式解決這些問題。首先是**AGGREGATE**（聚合），它通過定義清晰的所屬關係和邊界，並避免混亂、錯綜複雜的對象關係網來實現模型的內聚。聚合模式對於維護生命週期各個階段的完整性具有至關重要的作用。

接下來，我們將注意力轉移到生命週期的開始階段，使用**FACTORY**（工廠）來創建和重建複雜對像和**AGGREGATE**（聚合），從而封裝它們的內部結構。最後，在生命週期的中間和末尾使用**REPOSITORY**（存儲庫）來提供查找和檢索持久化對象並封裝龐大基礎設施的手段。

儘管**REPOSITORY**和**FACTORY**本身並不是來源於領域，但它們在領域設計中扮演著重要的角色。這些結構提供了易於掌握的模型對像處理方式，使**MODEL-DRIVEN DESIGN**更完備。

使用**AGGREGATE**進行建模，並且在設計中結合使用**FACTORY**和**REPOSITORY**，這樣我們就能夠在模型對象的整個生命週期中，以有意義的單元、系統地操縱它們。**AGGREGATE**可以劃分出一個範圍，這個範圍內的模型元素在生命週期各個階段都應該維護其固定規則。**FACTORY**和**REPOSITORY**在**AGGREGATE**基礎上進行操作，將特定生命週期轉換的複雜性封裝起來。

## **6.1 模式：**AGGREGATE****



減少設計中的關聯有助於簡化對像之間的遍歷，並在某種程度上限制關係的急劇增多。但大多數業務領域中的對象都具有十分複雜的聯繫，以至於最終會形成很長、很深的對象引用路徑，我們不得不在這個路徑上追蹤對象。在某種程度上，這種混亂狀態反映了現實世界，因為現實世界中就很少有清晰的邊界。但這卻是軟件設計中的一個重要問題。

假設我們從數據庫中刪除一個**Person**對象。這個人的姓名、出生日期和工作描述要一起被刪除，但要如何處理地址呢？可能還有其他人住在同一地址。如果刪除了地址，那些**Person**對像將會引用一個被刪除的對象。如果保留地址，那麼垃圾地址在數據庫中會累積起來。雖然自動垃圾收集機制可以清除垃圾地址，但這也只是一種技術上的修復；就算數據庫系統存在這種處理機制，一個基本的建模問題依然被忽略了。

即便是在考慮孤立的事務時，典型對像模型中的關係網也使我們難以斷定一個修改會產生哪些潛在的影響。僅僅因為存在依賴就更新系統中的每個對象，這樣做是不現實的。

在多個客戶對相同對像進行並發訪問的系統中，這個問題更加突出。當很多用戶對系統中的對象進行查詢和更新時，必須防止他們同時修改互相依賴的對象。範圍錯誤將導致嚴重的後果。

在具有複雜關聯的模型中，要想保證對像更改的一致性是很困難的。不僅互不關聯的對象需要遵守一些固定規則，而且緊密關聯的各組對象也要遵守一些固定規則。然而，過於謹慎的鎖定機制又會導致多個用戶之間毫無意義地互相干擾，從而使系統不可用。

換句話說，我們如何知道一個由其他對像組成的對象從哪裡開始，又到何處結束呢？在任何具有持久化數據存儲的系統中，對數據進行修改的事務必須要有範圍，而且要有保持數據一致性的方式（也就是說，保持數據遵守固定規則）。數據庫支持各種鎖機制，而且可以編寫一些測試來驗證。但這些特殊的解決方案分散了人們對模型的注意力，很快人們就會回到「走一步，看一步」的老路上來。

實際上，要想找到一種兼顧各種問題的解決方案，要求對領域有深刻的理解，例如，要瞭解特定類實例之間的更改頻率這樣的深層次

因素。我們需要找到一個使對像間衝突較少而固定規則聯繫更緊密的模型。

儘管從表面上看這個問題是數據庫事務方面的一個技術難題，但它的根源卻在模型，歸根結底是由於模型中缺乏明確定義的邊界。從模型得到的解決方案將使模型更易於理解，並且使設計更易於溝通。當模型被修改時，它將引導我們對實現做出修改。

人們已經開發出很多模式（scheme）來定義模型中的所屬關係。下面這個簡單但嚴格的系統就提煉自這些概念，其包括一組用於實現事務（這些事務用來修改對象及其所有者）的規則[7]。

首先，我們需要用一個抽像來封裝模型中的引用。AGGREGATE就是一組相關對象的集合，我們把它作為數據修改的單元。每個AGGREGATE都有一個根（root）和一個邊界（boundary）。邊界定義了AGGREGATE的內部都有什麼。根則是AGGREGATE所包含的一個特定ENTITY。對AGGREGATE而言，外部對像只可以引用根，而邊界內部的對象之間則可以互相引用。除根以外的其他ENTITY都有本地標識，但這些標識只在AGGREGATE內部才需要加以區別，因為外部對像除了根ENTITY之外看不到其他對象。

汽車修配廠的軟件可能會使用汽車模型。如圖6-2所示。汽車是一個具有全局標識的ENTITY：我們需要將這部汽車與世界上所有其他汽車區分開（即使是一些非常相似的汽車）。我們可以使用車輛識別號來進行區分，車輛識別號是為每輛新汽車分配的唯一標識符。我們可能想通過4個輪子的位移跟蹤輪胎的轉動歷史。我們可能想知道每個輪胎的里程數和磨損度。要想知道哪個輪胎在哪兒，必須將輪胎標識為ENTITY。當脫離這輛車的上下文後，我們很可能就不再關心這些輪胎的標識了。如果更換了輪胎並將舊輪胎送到回收廠，那麼軟件將不再需要跟蹤它們，它們會成為一堆廢舊輪胎中的一部分。沒有人會關

心它們的轉動歷史。更重要的是，即使輪胎被安在汽車上，也不會有人通過系統查詢特定的輪胎，然後看看這個輪胎在哪輛汽車上。人們只會在數據庫中查找汽車，然後臨時查看一下這部汽車的輪胎情況。因此，汽車是AGGREGATE的根ENTITY，而輪胎處於這個AGGREGATE的邊界之內。另一方面，發動機組上面都刻有序列號，而且有時是獨立於汽車被跟蹤的。在一些應用程序中，發動機可以是自己的AGGREGATE的根。

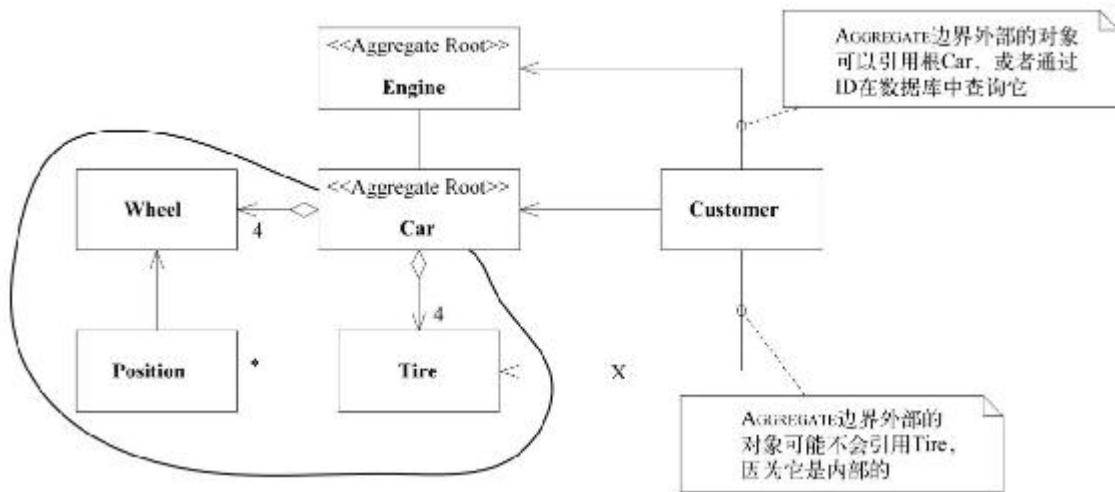


圖6-2 本地標識與全局標識及對像引用

固定規則 ( invariant ) 是指在數據變化時必須保持的一致性規則，其涉及 AGGREGATE 成員之間的內部關係。而任何跨越 AGGREGATE 的規則將不要求每時每刻都保持最新狀態。通過事件處理、批處理或其他更新機制，這些依賴會在一定的時間內得以解決。但在每個事務完成時，AGGREGATE 內部所應用的固定規則必須得到滿足，如圖6-3所示。

現在，為了實現這個概念上的AGGREGATE，需要對所有事務應用一組規則。

根ENTITY具有全局標識，它最終負責檢查固定規則。

根ENTITY具有全局標識。邊界內的ENTITY具有本地標識，這些標識只在AGGREGATE內部才是唯一的。

AGGREGATE外部的對象不能引用除根ENTITY之外的任何內部對象。根ENTITY可以把對內部ENTITY的引用傳遞給它們，但這些對像只能臨時使用這些引用，而不能保持引用。根可以把一個VALUE OBJECT的副本傳遞給另一個對象，而不必關心它發生什麼變化，因為它只是一個VALUE，不再與AGGREGATE有任何關聯。

作為上一條規則的推論，只有AGGREGATE的根才能直接通過數據庫查詢獲取。所有其他對像必須通過遍歷關聯來發現。

AGGREGATE內部的對象可以保持對其他AGGREGATE根的引用。

刪除操作必須一次刪除AGGREGATE邊界之內的所有對象。（利用垃圾收集機制，這很容易做到。由於除根以外的其他對象都沒有外部引用，因此刪除了根以後，其他對像均會被回收。）

當提交對AGGREGATE邊界內部的任何對象的修改時，整個AGGREGATE的所有固定規則都必須被滿足。

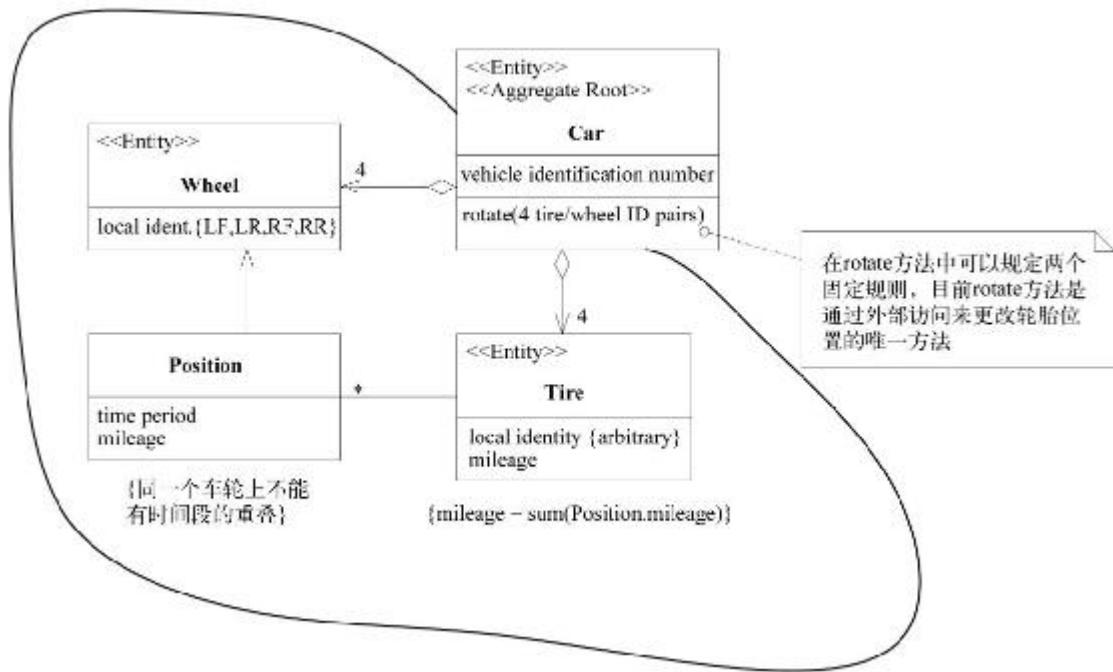


圖6-3 AGGREGATE的固定規則

我們應該將**ENTITY**和**VALUE OBJECT**分門別類地聚集到**AGGREGATE**中，並定義每個**AGGREGATE**的邊界。在每個**AGGREGATE**中，選擇一個**ENTITY**作為根，並通過根來控制對邊界內其他對象的所有訪問。只允許外部對像保持對根的引用。對內部成員的臨時引用可以被傳遞出去，但僅在一次操作中有效。由於根控制訪問，因此不能繞過它來修改內部對象。這種設計有利於確保**AGGREGATE**中的對象滿足所有固定規則，也可以確保在任何狀態變化時**AGGREGATE**作為一個整體滿足固定規則。

有一個能夠聲明**AGGREGATE**的技術框架是很有幫助的，這樣就可以自動實施鎖機制和其他一些功能。如果沒有這樣的技術框架，團隊就必須靠自我約束來使用事先商定的**AGGREGATE**，並按照這些**AGGREGATE**來編寫代碼。

#### 示例 採購訂單的完整性

考慮一個簡化的採購訂單系統（見圖6-4）可能具有的複雜性。

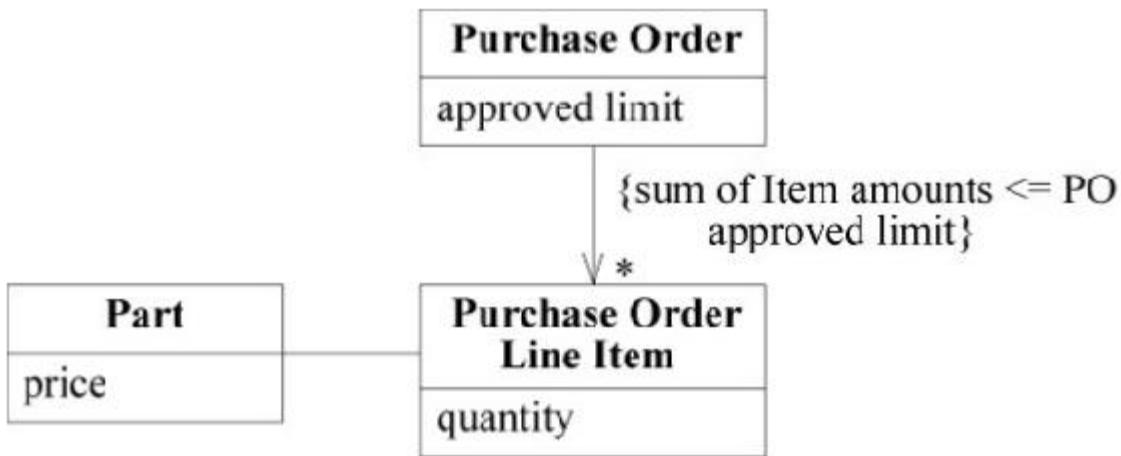


圖6-4 一個採購訂單系統的模型

圖6-4展示了三個典型的採購訂單（Purchase Order，PO）視圖，它被分解為採購項（Line Item），一條固定規則是採購項的總量不能

超過PO總額的限制。當前實現存在以下3個互相關聯的問題。

(1) 固定規則的實施。當添加新採購項時，PO檢查總額，如果新增的採購項使總額超出限制，則將PO標記為無效。正如我們將要看到的那樣，這種保護機制並不充分。

(2) 變更管理。當PO被刪除或存檔時，各個採購項也將被一塊處理，但模型並沒有給出關係應該在何處停止。在不同時間更改部件（Part）價格所產生的影響也不明確。

(3) 數據庫共享。數據庫會出現由於多個用戶競爭使用而帶來的問題。

多個用戶將並發地輸入和更新各個PO，因此必須防止他們互相干擾。讓我們從一個非常簡單的策略開始，當一個用戶開始編輯任何一個對像時，鎖定該對象，直到用戶提交事務。這樣，當George編輯採購項001時，Amanda就無法訪問該項。Amanda可以編輯其他PO上的任何採購項（包括George正在編輯的PO上的其他採購項），如圖6-5所示。

PO #0012946 Approved Limit: \$1,000.00				
Item #	Quantity	Part	Price	Amount
001	3	Guitars	@ 100.00	300.00
002	2	Trombones	@ 200.00	400.00
Total:				700.00

圖6-5 數據庫中存儲的PO的初始情形

每個用戶都將從數據庫讀取對象，並在自己的內存空間中實例化對象，而後在那裡查看和編輯對象。只有當開始編輯時，才會請求進行數據庫鎖定。因此，George和Amanda可以同時工作，只要他們不同時編輯相同的採購項即可。一切正常，直到George和Amanda開始編輯同一個PO上的不同採購項，如圖6-6所示。

George在他的視圖中添加了兩把吉他				
PO #0012946 Approved Limit: \$1000.00				
Item #	Quantity	Part	Price	Amount
001	5	Guitars	@ 100.00	<b>500.00</b>
002	2	Trombones	@ 200.00	400.00
Total:				900.00

Amanda在她的視圖中添加了一把長號				
PO #0012946 Approved Limit: \$1,000.00				
Item #	Quantity	Part	Price	Amount
001	3	Guitars	@ 100.00	300.00
002	3	Trombones	@ 200.00	<b>600.00</b>
Total:				900.00

圖6-6 在不同事務中同時進行的編輯

從這兩個用戶和他們各自軟件的角度來看，他們的操作都沒有問題，因為他們忽略了事務期間數據庫其他部分所發生的變化，而且每個用戶都沒有修改被對方鎖定的採購項。

當這兩個用戶保存了修改之後，數據庫中就存儲了一個違反領域模型固定規則的PO。一條重要的業務規則被破壞了，但並沒有人知道，如圖6-7所示。

PO #0012946 Approved Limit: \$1,000.00				
Item #	Quantity	Part	Price	Amount
001	5	Guitars	@100.00	500.00
002	3	Trombones	@200.00	600.00
				<b>Total: 1,100.00</b>

圖6-7 最後的PO超過了批准限額 ( 破壞了固定規則 )

顯然，鎖定單個行並不是一種充分的保護機制。如果一次鎖定一個PO，可以防止這樣的問題發生，如圖6-8所示。

George编辑他的视图				
PO #0012946 Approved Limit: \$1,000.00				
Item #	Quantity	Part	Price	Amount
001	5	Guitars	@ 100.00	500.00
002	2	Trombones	@ 200.00	400.00
				<b>Total: 900.00</b>

George已经提交更改

Amanda被锁定在PO #0012946之外

Amanda获得访问权，George所做的更改被显示出来				
PO #0012946 Approved Limit: \$1,000.00				
Item #	Quantity	Part	Price	Amount
001	5	Guitars	@ 100.00	500.00
002	3	Trombones	@ 200.00	600.00
				<b>Limit exceeded → Total: 1,100.00</b>

圖6-8 鎖定整個PO可以確保滿足固定規則

直到Amanda解決這個問題之前，程序將不允許保存這個事務，Amanda可以通過提高限額或減少一把吉他來解決此問題。這種機制防止了問題，如果大部分工作分佈在多個PO上，那麼這可能是個不錯的解決方案。但如果是很多人同時對一個大PO的不同項進行操作時，這種鎖定機制就顯得很笨拙了。

即便是很多小PO，也存在其他方法破壞這條固定規則。讓我們看看「Part」。如果在Amanda將長號加入訂單時，有人更改了長號的價格，這不也會破壞固定規則嗎？

那麼，我們試著除了鎖定整個PO之外，也鎖定Part。圖6-9展示了當George、Amanda和Sam在不同PO上工作時將會發生的情況。

PO #0012946 Approved Limit: \$1,000.00					PO #0012932 Approved Limit: \$1,850.00																																																																											
<table border="1"> <thead> <tr> <th>Item #</th><th>Quantity</th><th>Part</th><th>Price</th><th>Amount</th></tr> </thead> <tbody> <tr> <td>001</td><td>2</td><td>Guitars</td><td>@ 100.00</td><td>200.00</td></tr> <tr> <td>002</td><td>2</td><td>Trombones</td><td>@ 200.00</td><td>400.00</td></tr> <tr> <td></td><td></td><td></td><td></td><td></td></tr> <tr> <td></td><td></td><td></td><td></td><td></td></tr> <tr> <td></td><td></td><td></td><td></td><td></td></tr> <tr> <td colspan="5">Total: 600.00</td><td colspan="5">Total: 1,600.00</td></tr> </tbody></table>					Item #	Quantity	Part	Price	Amount	001	2	Guitars	@ 100.00	200.00	002	2	Trombones	@ 200.00	400.00																Total: 600.00					Total: 1,600.00					<table border="1"> <thead> <tr> <th>Item #</th><th>Quantity</th><th>Part</th><th>Price</th><th>Amount</th></tr> </thead> <tbody> <tr> <td>001</td><td>3</td><td>Violins</td><td>@ 400.00</td><td>1,200.00</td></tr> <tr> <td>002</td><td>2</td><td>Trombones</td><td>@ 200.00</td><td>400.00</td></tr> <tr> <td></td><td></td><td></td><td></td><td></td></tr> <tr> <td></td><td></td><td></td><td></td><td></td></tr> <tr> <td></td><td></td><td></td><td></td><td></td></tr> </tbody> </table>						Item #	Quantity	Part	Price	Amount	001	3	Violins	@ 400.00	1,200.00	002	2	Trombones	@ 200.00	400.00															
Item #	Quantity	Part	Price	Amount																																																																												
001	2	Guitars	@ 100.00	200.00																																																																												
002	2	Trombones	@ 200.00	400.00																																																																												
Total: 600.00					Total: 1,600.00																																																																											
Item #	Quantity	Part	Price	Amount																																																																												
001	3	Violins	@ 400.00	1,200.00																																																																												
002	2	Trombones	@ 200.00	400.00																																																																												

圖6-9 過於謹慎的鎖定會妨礙人們的工作

工作變得越來越麻煩，因為在**Part**上出現了很多爭用的情況。這樣就會發生圖6-10中的結果：3個人都需要等待。

現在我們可以開始改進模型，在模型中加入以下業務知識。

- (1) Part在很多PO中使用（會產生高競爭）。
- (2) 對Part的修改少於對PO的修改。
- (3) 對Price（價格）的修改不一定要傳播到現有PO，它取決於修改價格時PO處於什麼狀態。

George添加小提琴，必須等待Amanda完成工作(!)

PO #0012946 Approved Limit: \$1,000.00				
Item #	Quantity	Part	Price	Amount
001	<b>2</b>	Guitars	@ 100.00	<b>200.00</b>
002	2	Trombones	@ 200.00	400.00
<b>003</b>	<b>1</b>	<b>Violins</b>	<b>@ 400.00</b>	<b>400.00</b>
Total: 1,000.00				

圖6-10 死鎖

當考慮已經交貨並存檔的PO時，第三點尤為明顯。它們顯示的當然是填寫時的價格，而不是當前價格。

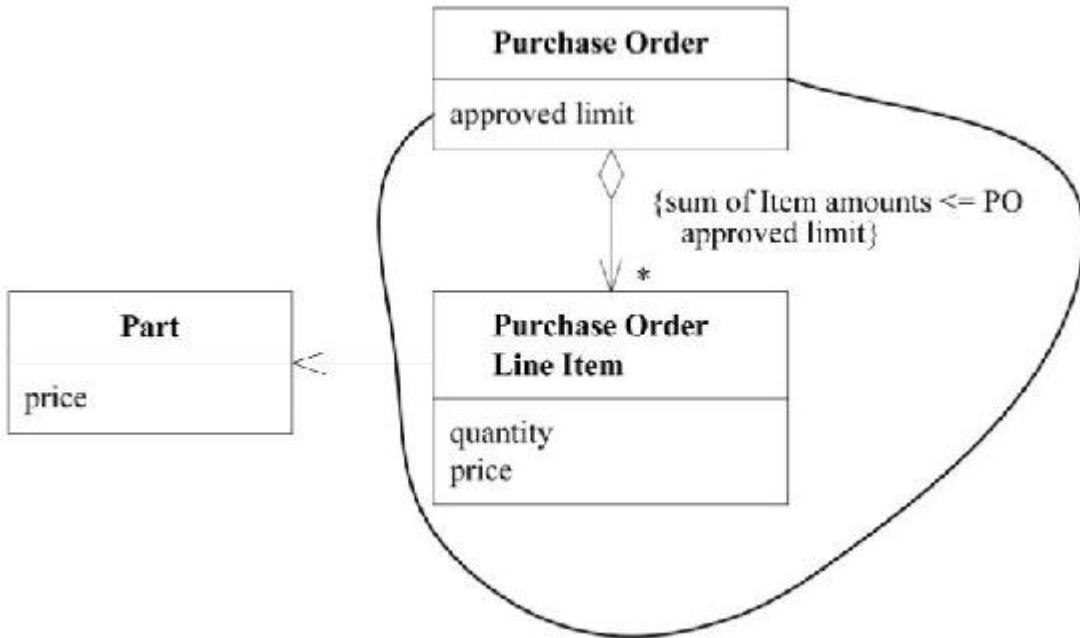


圖6-11 price被複製到Line Item中，現在可以確保滿足聚合的固定規則了

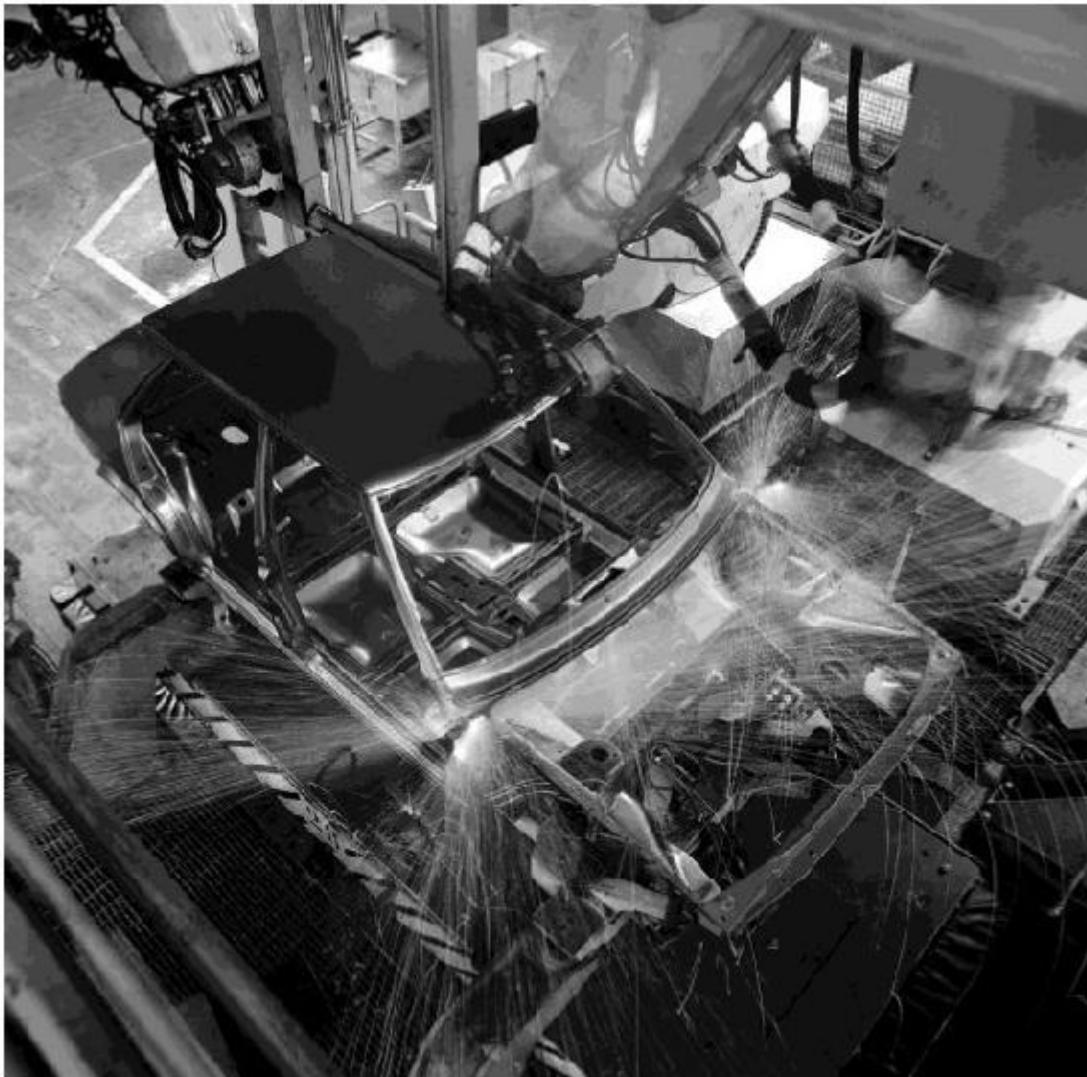
按照圖6-11，這個模型得到的實現可以確保滿足PO和採購項相關的固定規則，同時，修改部件的價格將不會立即影響引用部件的採購項。涉及面更廣的規則可以通過其他方式來滿足。例如，系統可以每天為用戶列出價格過期的採購項，這樣用戶就可以決定是更新還是去掉採購項。但這並不是必須一直保持的固定規則。通過減少採購項對Part的依賴，可以避免爭用，並且能夠更好地反映出業務的現實情況。同時，加強PO與採購項之間的關係可以確保遵守這條重要的業務規則。

AGGREGATE強制了PO與採購項之間符合業務實際的所屬關係。PO和採購項的創建及刪除很自然地被聯繫在一起，而Part的創建和刪除卻是獨立的。

AGGREGATE劃分出一個範圍，在這個範圍內，生命週期的每個階段都必須滿足一些固定規則。接下來要討論的兩種模式FACTORY和

REPOSITORY都是在AGGREGATE上執行操作，它們將特定生命週期轉換的複雜性封裝起來.....

## 6.2 模式：FACTORY



當創建一個對像或創建整個AGGREGATE時，如果創建工作很複雜，或者暴露了過多的內部結構，則可以使用FACTORY進行封裝。

對象的功能主要體現在其複雜的內部配鎔以及關聯方面。我們應該一直對對像進行提煉，直到所有與其意義或在交互中的角色無關的

內容被完全剔除為止。一個對像在它的生命週期中要承擔大量職責。如果再讓複雜對像負責自身的創建，那麼職責過載將會導致問題。

汽車發動機是一種複雜的機械裝置，它由數十個零件共同協作來履行發動機的職責——使軸轉動。我們可以試著設計一種發動機組，讓它自己抓取一組活塞並塞到汽缸中，火花塞也可以自己找到插孔並把自己擰進去。但這樣組裝的複雜機器可能沒有我們常見的發動機那樣可靠或高效。相反，我們用其他東西來裝配發動機。或許是機械師，或者是工業機器人。無論是機器人還是人，實際上都比二者要裝配的發動機複雜。裝配零件的工作與使軸旋轉的工作完全無關。只是在生產汽車時才需要裝配工，我們駕駛時並不需要機器人或機械師。由於汽車的裝配和駕駛永遠不會同時發生，因此將這兩種功能合併到同一個機制中是毫無價值的。同理，裝配複雜的複合對象的工作也最好與對像要執行的工作分開。

但將職責轉交給另一個相關方——應用程序中的客戶（client）對像——會產生更嚴重的問題。客戶知道需要完成什麼工作，並依靠領域對像來執行必要的計算。如果指望客戶來裝配它需要的領域對象，那麼它必須要瞭解一些對象的內部結構。為了確保所有應用於領域對像各部分關係的固定規則得到滿足，客戶必須知道對象的一些規則。甚至調用構造函數也會使客戶與所要構建的對象的具體類產生耦合。結果是，對領域對像實現所做的任何修改都要求客戶做出相應修改，這使得重構變得更加困難。

當客戶負責創建對像時，它會牽涉不必要的複雜性，並將其職責搞得模糊不清。這違背了領域對象及所創建的**AGGREGATE**的封裝要求。更嚴重的是，如果客戶是應用層的一部分，那麼職責就會從領域層洩漏到應用層中。應用層與實現細節之間的這種耦合使得領域層抽象的大部分優勢蕩然無存，而且導致後續更改的代價變得更加高昂。

對象的創建本身可以是一個主要操作，但被創建的對象並不適合承擔複雜的裝配操作。將這些職責混在一起可能產生難以理解的拙劣設計。讓客戶直接負責創建對像又會使客戶的設計陷入混亂，並且破壞被裝配對像或**AGGREGATE**的封裝，而且導致客戶與被創建對象的實現之間產生過於緊密的耦合。

複雜的對象創建是領域層的職責，然而這項任務並不屬於那些用於表示模型的對象。在有些情況下，對象的創建和裝配對應於領域中的重要事件，如「開立銀行賬戶」。但一般情況下，對象的創建和裝配在領域中並沒有什麼意義，它們只不過是實現的一種需要。為瞭解決這一問題，我們必須在領域設計中增加一種新的構造，它不是**ENTITY**、**VALUE OBJECT**，也不是**SERVICE**。這與前一章的論述相違背，因此把它解釋清楚很重要。我們正在向設計中添加一些新元素，但它們不對應於模型中的任何事物，而確實又承擔領域層的部分職責。

每種面向對象的語言都提供了一種創建對象的機制（例如，Java和C++中的構造函數，Smalltalk中創建實例的類方法），但我們仍然需要一種更加抽象且不與其他對像發生耦合的構造機制。這就是**FACTORY**，它是一種負責創建其他對象的程序元素。如圖6-12所示。

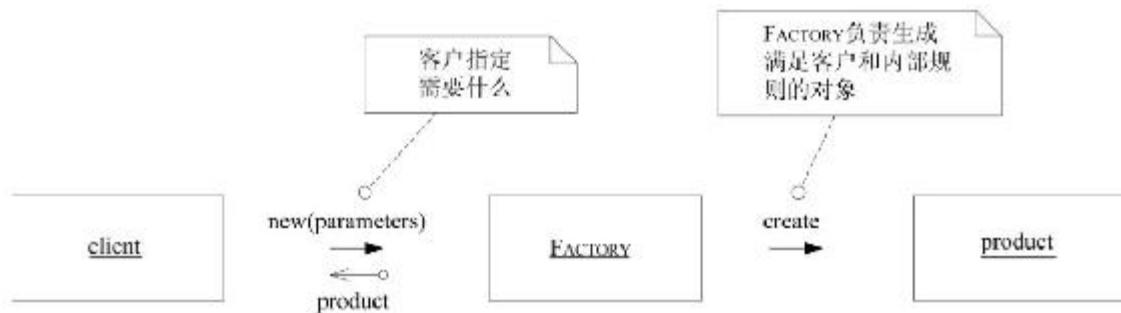


圖6-12 與FACTORY的基本交互

正如對象的接口應該封裝對象的實現一樣（從而使客戶無需知道對象的工作機理就可以使用對象的功能），**FACTORY**封裝了創建複雜

對像或**AGGREGATE**所需的知識。它提供了反映客戶目標的接口，以及被創建對象的抽象視圖。

因此：

應該將創建複雜對象的實例和**AGGREGATE**的職責轉移給單獨的對象，這個對象本身可能沒有承擔領域模型中的職責，但它仍是領域設計的一部分。提供一個封裝所有複雜裝配操作的接口，而且這個接口不需要客戶引用要被實例化的對象的具體類。在創建**AGGREGATE**時要把它作為一個整體，並確保它滿足固定規則。

**FACTORY**有很多種設計方式。**[Gamma et al.1995]**中詳盡論述了幾種特定目的的創建模式，包括**FACTORY METHOD**（工廠方法）、**ABSTRACT FACTORY**（抽象工廠）和**BUILDER**（構建器）。該書主要研究了適用於最複雜的對象構造問題的模式。本書的重點並不是深入討論**FACTORY**的設計問題，而是要表明**FACTORY**的重要地位——它是領域設計的重要組件。正確使用**FACTORY**有助於保證**MODEL-DRIVEN DESIGN**沿正確的軌道前進。

任何好的工廠都需滿足以下兩個基本需求。

(1) 每個創建方法都是原子的，而且要保證被創建對像或**AGGREGATE**的所有固定規則。**FACTORY**生成的對象要處於一致的狀態。在生成**ENTITY**時，這意味著創建滿足所有固定規則的整個**AGGREGATE**，但在創建完成後可以向聚合添加可選元素。在創建不變的**VALUE OBJECT**時，這意味著所有屬性必須被初始化為正確的最終狀態。如果**FACTORY**通過其接口收到了一個創建對象的請求，而它又無法正確地創建出這個對象，那麼它應該拋出一個異常，或者採用其他機制，以確保不會返回錯誤的值。

(2) **FACTORY**應該被抽象為所需的類型，而不是所要創建的具體類。**[Gamma et al.1995]**中的高級**FACTORY**模式介紹了這一話題。

### 6.2.1 選擇FACTORY及其應用位置

一般來說，FACTORY的作用是隱藏創建對象的細節，而且我們把FACTORY用在那些需要隱藏細節的地方。這些決定通常與AGGREGATE有關。

例如，如果需要向一個已存在的AGGREGATE添加元素，可以在AGGREGATE的根上創建一個FACTORY METHOD。這樣就可以把AGGREGATE的內部實現細節隱藏起來，使任何外部客戶看不到這些細節，同時使根負責確保AGGREGATE在添加元素時的完整性，如圖6-13所示。

另一個示例是在一個對像上使用FACTORY METHOD，這個對象與生成另一個對像密切相關，但它並不擁有所生成的對象。當一個對象的創建主要使用另一個對象的數據（或許還有規則）時，則可以在後者的對象上創建一個FACTORY METHOD，這樣就不必將後者的信息提取到其他地方來創建前者。這樣做還有利於表達前者與後者之間的關係。

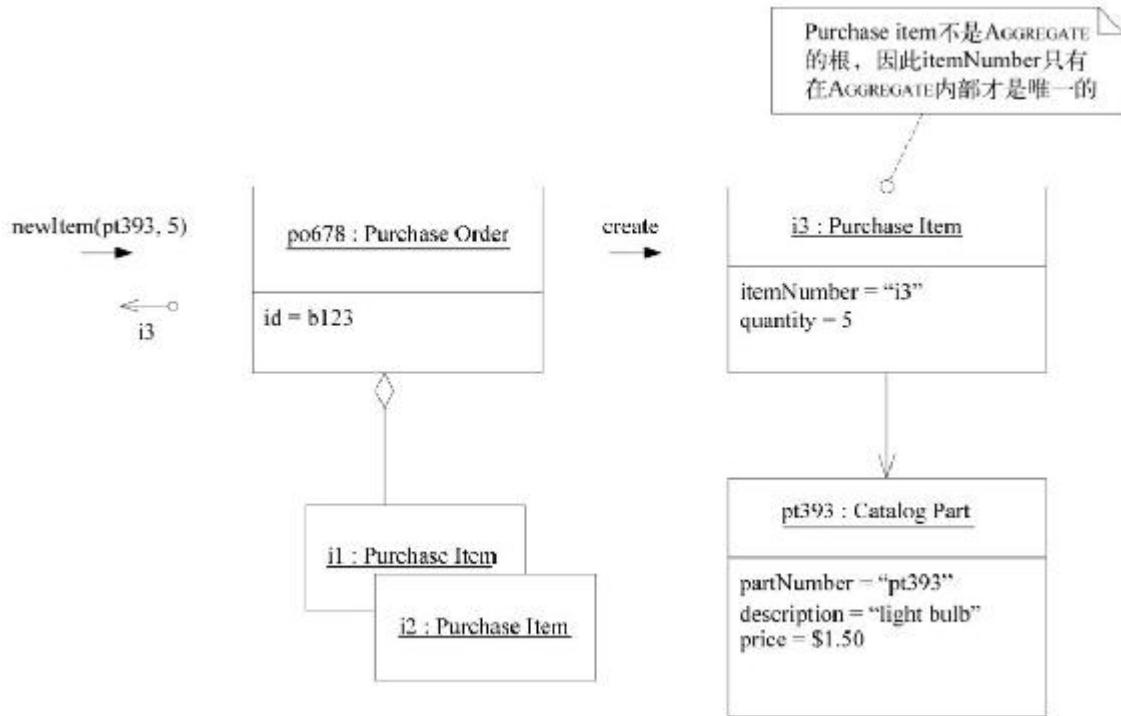


圖6-13 一個FACTORY METHOD封裝了AGGREGATE的擴展

在圖 6-14 中，Trade Order 不屬於 Brokerage Account 所在的 AGGREGATE，因為它從一開始就與交易執行應用程序進行交互，所以把它放在 Brokerage Account 中只會礙事。儘管如此，讓 Brokerage Account 負責控制 Trade Order 的創建卻是很自然的事情。Brokerage Account 含有會被嵌入到 Trade Order 中的信息（從自己的標識開始），而且它還包含與交易相關的規則——這些規則控制了哪些交易是允許的。隱藏 Trade Order 的實現細節還會帶來一些其他好處。例如，我們可以將它重構為一個層次結構，分別為 Buy Order 和 Sell Order 創建一些子類。FACTORY 可以避免客戶與具體類之間產生耦合。

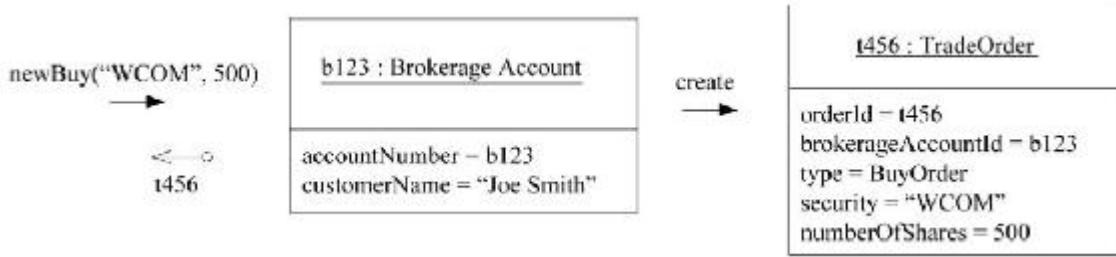


圖6-14 FACTORY METHOD生成一個ENTITY，但這個ENTITY並不屬於FACTORY所在的AGGREGATE

FACTORY與被構建對像之間是緊密耦合的，因此FACTORY應該只被關聯到與被構建對像有著密切聯繫的對象上。當有些細節需要隱藏（無論要隱藏的是具體實現還是構造的複雜性）而又找不到合適的地方來隱藏它們時，必須創建一個專用的FACTORY對像或SERVICE。整個AGGREGATE通常由一個獨立的FACTORY來創建，FACTORY負責把對根的引用傳遞出去，並確保創建出的AGGREGATE滿足固定規則。如果AGGREGATE內部的某個對象需要一個FACTORY，而這個FACTORY又不適合在AGGREGATE根上創建，那麼應該構建一個獨立的FACTORY。但仍應遵守規則——把訪問限制在AGGREGATE內部，並確保從AGGREGATE外部只能對被構建對像進行臨時引用，如圖6-15所示。

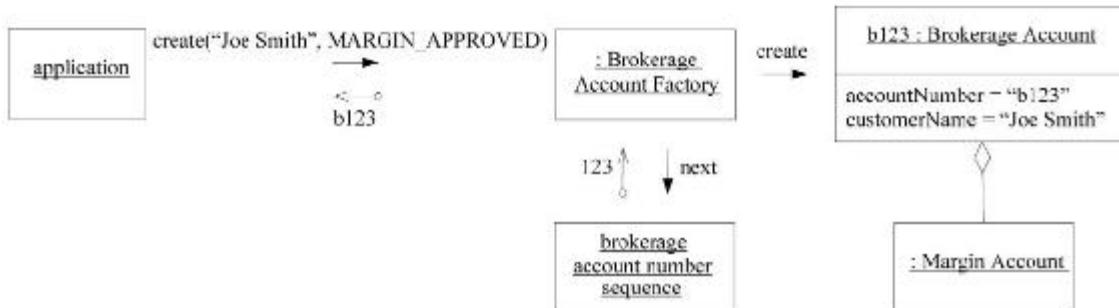


圖6-15 由一個獨立的FACTORY來構建AGGREGATE

## 6.2.2 有些情況下只需使用構造函數

我曾經在很多代碼中看到所有實例都是通過直接調用類構造函數來創建的，或者是使用編程語言的最基本的實例創建方式。**FACTORY**的引入提供了巨大的優勢，而這種優勢往往並未得到充分利用。但是，在有些情況下直接使用構造函數確實是最佳選擇。**FACTORY**實際上會使那些不具有多態性的簡單對像複雜化。

在以下情況下最好使用簡單的、公共的構造函數。

類 ( **class** ) 是一種類型 ( **type** ) 。它不是任何相關層次結構的一部分，而且也沒有通過接口實現多態性。

客戶關心的是實現，可能是將其作為選擇**STRATEGY**的一種方式。

客戶可以訪問對象的所有屬性，因此向客戶公開的構造函數中沒有嵌套的對象創建。

構造並不複雜。

公共構造函數必須遵守與**FACTORY**相同的規則：它必須是原子操作，而且要滿足被創建對象的所有固定規則。

不要在構造函數中調用其他類的構造函數。構造函數應該保持絕對簡單。複雜的裝配，特別是**AGGREGATE**，需要使用**FACTORY**。使用**FACTORY METHOD**的門檻並不高。

**Java**類庫提供了一些有趣的例子。所有集合都實現了接口，接口使得客戶與具體實現之間不產生耦合。然而，它們都是通過直接調用構造函數創建的。但是，集合類本來是可以使用**FACTORY**來封裝集合的層次結構的。而且，客戶也可以使用**FACTORY**的方法來請求所需的特性，然後由**FACTORY**來選擇適當的類來實例化。這樣一來，創建集合的代碼就會有更強的表達力，而且新增集合類時不會破壞現有的**Java**程序。

但在某些場合 下使用具體的構造函數更為合適。首先，在很多應用程序中，實現方式的選擇對性能的影響是非常敏感的，因此應用程序需要控制選擇哪種實現（儘管如此，真正智能的FACTORY仍然可以滿足這些因素的要求）。不管怎樣，集合類的數量並不多，因此選擇並不複雜。

雖然沒有使用FACTORY，但抽象集合類型仍然具有一定價值，原因就在於它們的使用模式。集合通常都是在一個地方創建，而在其他地方使用。這意味著最終使用集合（添加、刪除和檢索其內容）的客戶仍可以與接口進行對話，從而不與實現發生耦合。集合類的選擇通常由擁有該集合的對象來決定，或是由該對象的FACTORY來決定。

### **6.2.3 接口的設計**

當設計FACTORY的方法簽名時，無論是獨立的FACTORY還是FACTORY METHOD，都要記住以下兩點。

每個操作都必須是原子的。我們必須在與FACTORY的一次交互中把創建對像所需的所有信息傳遞給FACTORY。同時必須確定當創建失敗時將執行什麼操作，比如某些固定規則沒有被滿足。可以拋出一個異常或僅僅返回null。為了保持一致，可以考慮採用編碼標準來處理所有FACTORY的失敗。

Factory將與其參數發生耦合。如果在選擇輸入參數時不小心，可能會產生錯綜複雜的依賴關係。耦合程度取決於對參數（argument）的處理。如果只是簡單地將參數插入到要構建的對象中，則依賴度是適中的。如果從參數中選出一部分在構造對像時使用，耦合將更緊密。

最安全的參數是那些來自較低設計層的參數。即使在同一層中，也有一種自然的分層傾向，其中更基本的對象被更高層的對象使用

（第10章將從不同方面討論這樣的分層，第16章也會論述這個問題）。

另一個好的參數選擇是模型中與被構建對像密切相關的對象，這樣不會增加新的依賴。在前面的 **Purchase Order Item** 示例中，**FACTORY METHOD** 將 **Catalog Part** 作為一個參數，它是 **Item** 的一個重要的關聯。這在 **Purchase Order** 類和 **Part** 之間增加了直接依賴。但這 3 個對象組成了一個關係密切的概念小組。不管怎樣，**Purchase Order** 的 **AGGREGATE** 已經引用了 **Part**。因此將控制權交給 **AGGREGATE** 根，並封裝 **AGGREGATE** 的內部結構是一個不錯的折中選擇。

使用抽象類型的參數，而不是它們的具體類。**FACTORY** 與被構建對象的具體類發生耦合，而無需與具體的參數發生耦合。

#### 6.2.4 固定規則的相關邏輯應放置在哪裡

**FACTORY** 負責確保它所創建的對象或 **AGGREGATE** 滿足所有固定規則，然而在把應用於一個對象的規則移到該對像外部之前應三思。**FACTORY** 可以將固定規則的檢查工作委派給被創建對象，而且這通常是最佳選擇。

但 **FACTORY** 與被創建對像之間存在一種特殊關係。**FACTORY** 已經知道被創建對象的內部結構，而且創建 **FACTORY** 的目的與被創建對象的實現有著密切的聯繫。在某些情況下，把固定規則的相關邏輯放到 **FACTORY** 中是有好處的，這樣可以讓被創建對象的職責更明晰。對於 **AGGREGATE** 規則來說尤其如此（這些規則會約束很多對像）。但固定規則的相關邏輯卻特別不適合放到那些與其他領域對像關聯的 **FACTORY METHOD** 中。

雖然原則上在每個操作結束時都應該應用固定規則，但通常對像所允許的轉換可能永遠也不會用到這些規則。可能 **ENTITY** 標識屬性的賦值需要滿足一條固定規則。但該標識在創建後可能一直保持不變。

VALUE OBJECT 則是完全不變的。如果邏輯在對象的有效生命週期內永遠也不被用到，那麼對象就沒有必要攜帶這個邏輯。在這種情況下，FACTORY 是放鎔固定規則的合適地方，這樣可以使FACTORY創建出的對象更簡單。

### **6.2.5 ENTITY FACTORY與VALUE OBJECT FACTORY**

ENTITY FACTORY 與 VALUE OBJECT FACTORY 有兩個方面的不同。由於VALUE OBJECT 是不可變的，因此，FACTORY 所生成的對象就是最終形式。因此FACTORY操作必須得到被創建對象的完整描述。而ENTITY FACTORY 則只需具有構造有效AGGREGATE 所需的那些屬性。對於固定規則不關心的細節，可以之後再添加。

我們來看一下為 ENTITY 分配標識時將涉及的問題（ VALUE OBJECT 不會涉及這些問題）。正如第5章所指出的那樣，既可以由程序自動分配一個標識符，也可以通過外部（通常是用戶）提供一個標識符。如果客戶的標識是通過電話號碼跟蹤的，那麼該電話號碼必須作為參數被顯式地傳遞給FACTORY。當由程序分配標識符時，FACTORY 是控制它的理想場所。儘管唯一跟蹤ID 實際上是由數據庫「序列」或其他基礎設施機制生成的，但FACTORY 知道需要什麼樣的標識，以及將標識放到何處。

### **6.2.6 重建已存儲的對象**

到目前為止，FACTORY 只是發揮了它在對像生命週期開始時的作用。到了某一時刻，大部分對象都要存儲在數據庫中或通過網絡傳輸，而在當前的數據庫技術中，幾乎沒有哪種技術能夠保持對象的內容特徵。大多數傳輸方法都要將對像轉換為平面數據才能傳輸，這使得對像只能以非常有限的形式出現。因此，檢索操作潛在地需要一個複雜的過程將各個部分重新裝配成一個可用的對象。

用於重建對象的FACTORY與用於創建對象的FACTORY很類似，主要有以下兩點不同。

(1)用於重建對象的ENTITY FACTORY不分配新的跟蹤ID。如果重新分配ID，將丟失與先前對象的連續性。因此，在重建對象的FACTORY中，標識屬性必須是輸入參數的一部分。

(2)當固定規則未被滿足時，重建對象的FACTORY採用不同的方式進行處理。當創建新對像時，如果未滿足固定規則，FACTORY應該簡單地拒絕創建對象，但在重建對像時則需要更靈活的響應。如果對像已經在系統的某個地方存在（如在數據庫中），那麼不能忽略這個事實。但是，同樣也不能任憑規則被破壞。必須通過某種策略來修復這種不一致的情況，這使得重建對像比創建新對像更困難。

圖6-16和圖6-17顯示了兩種重建。當從數據庫中重建對像時，對像映射技術就可以提供部分或全部所需服務，這是非常便利的。當從其他介質重建對像時，如果出現複雜情況，FACTORY是個很好的選擇。

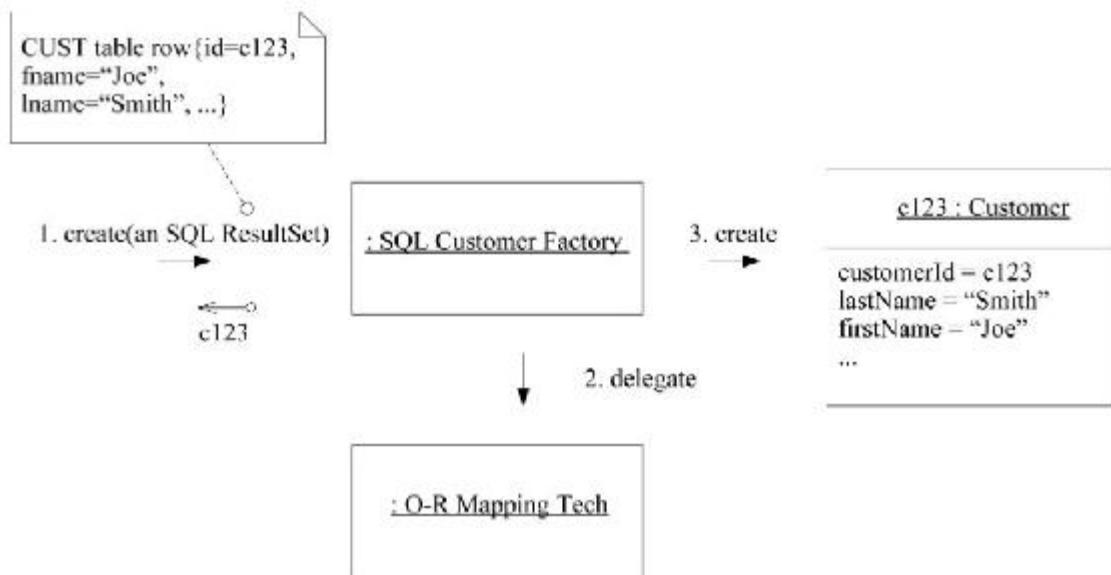


圖6-16 從關係數據庫中檢索一個ENTITY並重建它

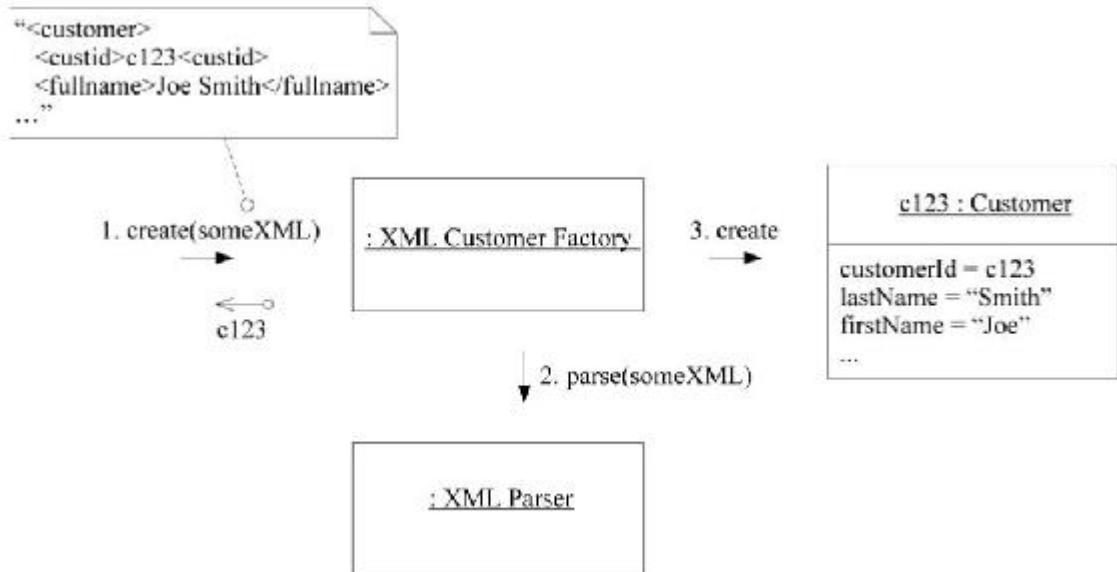


圖6-17 重建以XML形式傳輸的ENTITY

總之，必須把創建實例的訪問點標識出來，並顯式地定義它們的範圍。它們可能只是構造函數，但通常需要有一種更抽象或更複雜的實例創建機制。為了滿足這種需求，需要在設計中引入新的構造—FACTORY。FACTORY通常不表示模型的任何部分，但它們是領域設計的一部分，能夠使對像更明確地表示出模型。

FACTORY封裝了對象創建和重建時的生命週期轉換。還有一種轉換大大增加了領域設計的技術複雜性，這是對象與存儲之間的互相轉換。這種轉換由另一種領域設計構造來處理，它就是REPOSITORY。

## 6.3 模式：REPOSITORY



我們可以通過對像之間的關聯來找到對象。但當它處於生命週期的中間時，必須要有一個起點，以便從這個起點遍歷到一個**ENTITY**或**VALUE**。

無論要用對像執行什麼操作，都需要保持一個對它的引用。那麼如何獲得這個引用呢？一種方法是創建對象，因為創建操作將返回對新對象的引用。第二種方法是遍歷關聯。我們以一個已知對像作為起點，並向它請求一個關聯的對象。這樣的操作在任何面向對象的程序

中都會大量用到，而且對像之間的這些鏈接使對像模型具有更強的表達能力。但我們必須首先獲得作為起點的那個對象。

實際上，我曾經遇到過一個項目，團隊成員對**MODEL-DRIVEN DESIGN**懷有極大的熱情，因而試圖通過創建對像或遍歷對象的方法來訪問所有對象。他們的對象存儲在對像數據庫中，而且他們推斷出已有的概念關係將提供所有必要的關聯。他們只需完成充分的分析工作，以便使整個領域滿足內聚的要求。這種自己強加的限制導致他們創建出的模型錯綜複雜，而前幾章我們一直試圖通過仔細地實現**ENTITY**和應用**AGGREGATE**來避免這種複雜性。這種策略並沒有堅持多長時間，但團隊成員也一直沒有用一種更有條理的方法來取代它。他們臨時拼湊了一些解決方案，並放棄了最初的宏偉抱負。

想到這種方法的人並不多，嘗試它的人就更少了，因為人們將大部分對像存儲在關係數據庫中。這種存儲技術使人們自然而然地使用第三種獲取引用的方式——基於對象的屬性，執行查詢來找到對像；或者是找到對象的組成部分，然後重建它。

數據庫搜索是全局可訪問的，它使我們可以直接受訪問任何對象。由此，所有對象不需要相互聯接起來，整個對象關係網就能夠保持在可控的範圍內。是提供遍歷還是依靠搜索，這成為一個設計決策，需要在搜索的解耦與關聯的內聚之間做出權衡。**Customer**對像應該保持該客戶所有已訂的**Order**嗎？應該通過**Customer ID**字段在數據庫中查找**Order**嗎？恰當地結合搜索與關聯將會得到易於理解的設計。

遺憾的是，開發人員一般不會過多地考慮這種精細的設計，因為他們滿腦子都是需要用到的機制，以便很有技巧地利用它們來實現對象的存儲、收回和最終刪除。

現在，從技術的觀點來看，檢索已存儲對像實際上屬於創建對象的範疇，因為從數據庫中檢索出來的數據要被用來組裝新的對象。實

際上，由於需要經常編寫這樣的代碼，我們對此形成了根深蒂固的觀念。但從概念上講，對像檢索發生在**ENTITY**生命週期的中間。不能只是因為我們將Customer對像保存在數據庫中，而後把它檢索出來，這個Customer就代表了一個新客戶。為了記住這個區別，我把使用已存儲的數據創建實例的過程稱為重建。

領域驅動設計的目標是通過關注領域模型（而不是技術）來創建更好的軟件。假設開發人員構造了一個SQL查詢，並將它傳遞給基礎設施層中的某個查詢服務，然後再根據得到的錶行數據的結果集提取出所需信息，最後將這些信息傳遞給構造函數或**FACTORY**。開發人員執行這一連串操作的時候，早已不再把模型當作重點了。我們很自然地會把對像看作容器來放罷查詢出來的數據，這樣整個設計就轉向了數據處理風格。雖然具體的技術細節有所不同，但問題仍然存在——客戶處理的是技術，而不是模型概念。諸如**METADATA MAPPING LAYER**[Fowler 2002]這樣的基礎設施可以提供很大幫助，利用它很容易將查詢結果轉換為對象，但開發人員考慮的仍然是技術機制，而不是領域。更糟的是，當客戶代碼直接使用數據庫時，開發人員會試圖繞過模型的功能（如**AGGREGATE**，甚至是對像封裝），而直接獲取和操作他們所需的數據。這將導致越來越多的領域規則被嵌入到查詢代碼中，或者乾脆丟失了。雖然對像數據庫消除了轉換問題，但搜索機制還是很機械的，開發人員仍傾向於要什麼就去拿什麼。

客戶需要一種有效的方式來獲取對已存在的領域對象的引用。如果基礎設施提供了這方面的便利，那麼開發人員可能會增加很多可遍歷的關聯，這會使模型變得非常混亂。另一方面，開發人員可能使用查詢從數據庫中提取他們所需的數據，或是直接提取具體的對象，而不是通過**AGGREGATE**的根來得到這些對象。這樣就導致領域邏輯進入查詢和客戶代碼中，而**ENTITY**和**VALUE OBJECT**則變成單純的數

據容器。採用大多數處理數據庫訪問的技術複雜性很快就會使客戶代碼變得混亂，這將導致開發人員簡化領域層，最終使模型變得無關緊要。

根據到目前為止所討論的設計原則，如果我們找到一種訪問方法，它能夠明確地將模型作為焦點，從而應用這些原則，那麼我們就可以在某種程度上縮小對像訪問問題的範圍。。。初學者可以不必關心臨時對象。臨時對像（通常是**VALUE OBJECT**）只存在很短的時間，在客戶操作中用到它們時才創建它們，用完就刪除了。我們也不需要對那些很容易通過遍歷來找到的持久對像進行查詢訪問。例如，地址可以通過**Person**對像獲取。而且最重要的是，除了通過根來遍歷查找對象這種方法以外，禁止用其他方法對**AGGREGATE**內部的任何對像進行訪問。

持久化的**VALUE OBJECT**一般可以通過遍歷某個**ENTITY**來找到，在這裡**ENTITY**就是把對像封裝在一起的**AGGREGATE**的根。事實上，對**VALUE**的全局搜索訪問常常是沒有意義的，因為通過屬性找到**VALUE OBJECT**相當於用這些屬性創建一個新實例。但也有例外情況。例如，當我在線規劃旅行線路時，有時會先保存幾個中意的行程，過後再回頭從中選擇一個來預訂。這些行程就是**VALUE**（如果兩個行程由相同的航班構成，那麼我不會關心哪個是哪個），但它們已經與我的用戶名關聯到一起了，而且可以原封不動地將它們檢索出來。另一個例子是「枚舉」，在枚舉中一個類型有一組嚴格限定的、預定義的可能值。但是，對**VALUE OBJECT**的全局訪問比對**ENTITY**的全局訪問更少見，如果確實需要在數據庫中搜索一個已存在的**VALUE**，那麼值得考慮一下，搜索結果可能實際上是一個**ENTITY**，只是尚未識別它的標識。

從上面的討論顯然可以看出，大多數對象都不應該通過全局搜索來訪問。如果很容易就能從設計中看出那些確實需要全局搜索訪問的對象，那該有多好！

現在可以更精確地將問題重新表述如下：

在所有持久化對像中，有一小部分必須通過基於對像屬性的搜索來全局訪問。當很難通過遍歷方式來訪問某些**AGGREGATE**根的時候，就需要使用這種訪問方式。它們通常是**ENTITY**，有時是具有複雜內部結構的**VALUE OBJECT**，還可能是枚舉**VALUE**。而其他對像則不宜使用這種訪問方式，因為這會混淆它們之間的重要區別。隨意的數據庫查詢會破壞領域對象的封裝和**AGGREGATE**。技術基礎設施和數據庫訪問機制的暴露會增加客戶的複雜度，並妨礙模型驅動的設計。

有大量的技術可以用來解決數據庫訪問的技術難題，例如，將SQL封裝到**QUERY OBJECT**中，或利用**METADATA MAPPING LAYER**進行對像和表之間的轉換[Fowler 2002]。**FACTORY**可以幫助重建那些已存儲的對象（本章後面將會討論）。這些技術和很多其他技術有助於控制數據庫訪問的複雜度。

有得必有失，我們應該注意失去了什麼。我們已經不再考慮領域模型中的概念。代碼也不再表達業務，而是對數據庫檢索技術進行操縱。**REPOSITORY**是一個簡單的概念框架，它可用來封裝這些解決方案，並將我們的注意力重新拉回到模型上。

**REPOSITORY**將某種類型的所有對象表示為一個概念集合（通常是模擬的）。它的行為類似於集合（*collection*），只是具有更複雜的查詢功能。在添加或刪除相應類型的對象時，**REPOSITORY**的後臺機制負責將對像添加到數據庫中，或從數據庫中刪除對象。這個定義將

一組緊密相關的職責集中在一起，這些職責提供了對AGGREGATE根的整個生命週期的全程訪問。

客戶使用查詢方法向REPOSITORY請求對象，這些查詢方法根據客戶所指定的條件（通常是特定屬性的值）來挑選對象。REPOSITORY檢索被請求的對象，並封裝數據庫查詢和元數據映射機制。REPOSITORY可以根據客戶所要求的各種條件來挑選對象。它們也可以返回匯總信息，如有多少個實例滿足查詢條件。REPOSITORY甚至能返回匯總計算，如所有匹配對象的某個數值屬性的總和，如圖6-18所示。

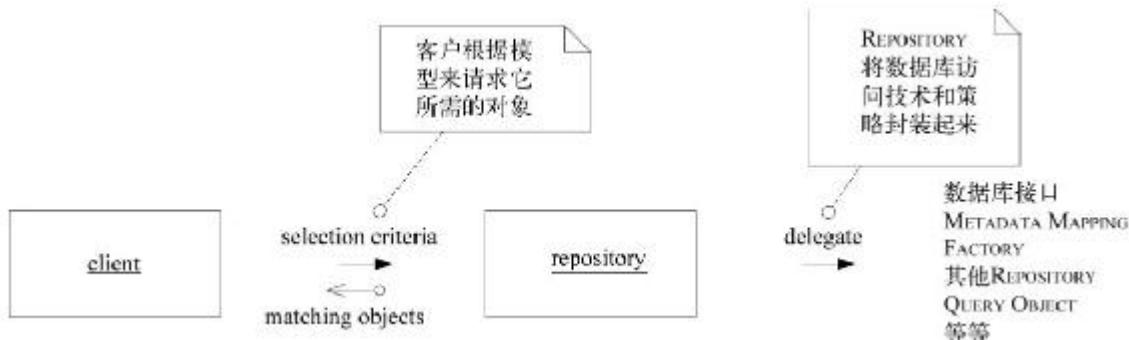


圖6-18 REPOSITORY為客戶執行一個搜索

REPOSITORY解除了客戶的巨大負擔，使客戶只需與一個簡單的、易於理解的接口進行對話，並根據模型向這個接口提出它的請求。要實現所有這些功能需要大量複雜的技術基礎設施，但接口很簡單，而且在概念層次上與領域模型緊密聯繫在一起。

因此：

為每種需要全局訪問的對象類型創建一個對象，這個對象相當於該類型的所有對象在內存中的一個集合的「替身」。通過一個眾所周知的全局接口來提供訪問。提供添加和刪除對象的方法，用這些方法來封裝在數據存儲中實際插入或刪除數據的操作。提供根據具體條件來挑選對象的方法，並返回屬性值滿足查詢條件的對象或對像集合

( 所返回的對象是完全實例化的 ) ，從而將實際的存儲和查詢技術封裝起來。只為那些確實需要直接訪問的 **AGGREGATE** 根提供 **REPOSITORY**。讓客戶始終聚焦於模型，而將所有對象的存儲和訪問操作交給 **REPOSITORY** 來完成。

**REPOSITORY** 有很多優點，包括：

它們為客戶提供了一個簡單的模型，可用來獲取持久化對象並管理它們的生命週期；

它們使應用程序和領域設計與持久化技術（多種數據庫策略甚至是多個數據源）解耦；

它們體現了有關對像訪問的設計決策；

可以很容易將它們替換為「啞實現」（ *dummy implementation* ），以便在測試中使用（通常使用內存中的集合）。

### **6.3.1 REPOSITORY的查詢**

所有 **REPOSITORY** 都為客戶提供了根據某種條件來查詢對象的方法，但如何設計這個接口卻有很多選擇。

最容易構建的 **REPOSITORY** 用硬編碼的方式來實現一些具有特定參數的查詢。這些查詢可以形式各異，例如，通過標識來檢索 **ENTITY**（幾乎所有 **REPOSITORY** 都提供了這種查詢）、通過某個特定屬性值或複雜的參數組合來請求一個對像集合、根據值域（如日期範圍）來選擇對象，甚至可以執行某些屬於 **REPOSITORY** 一般職責範圍內的計算（特別是利用那些底層數據庫所支持的操作）。如圖6-19所示。

儘管大多數查詢都返回一個對像或對像集合，但返回某些類型的匯總計算也符合 **REPOSITORY** 的概念，如對像數目，或模型需要對某個數值屬性進行求和統計。



圖6-19 在簡單REPOSITORY中進行的硬編碼查詢

在任何基礎設施上，都可以構建硬編碼式的查詢，也不需要很大的投入，因為即使它們不做這些事，有些客戶也必須要做。

在一些需要執行大量查詢的項目上，可以構建一個支持更靈活查詢的REPOSITORY框架。如圖6-20所示。這要求開發人員熟悉必要的技術，而且一個支持性的基礎設施會提供巨大的幫助。

基於SPECIFICATION（規格）的查詢是將REPOSITORY通用化的好辦法。客戶可以使用規格來描述（也就是指定）它需要什麼，而不必關心如何獲得結果。在這個過程中，可以創建一個對像來實際執行篩選操作。第9章將深入討論這種模式。

基於SPECIFICATION的查詢是一種優雅且靈活的查詢方法。根據所用的基礎設施的不同，它可能易於實現，也可能極為複雜。Rob Mee 和 Edward Hieatt 在 [Fowler 2002] 一書中探討了設計這樣的REPOSITORY時所涉及的更多技術問題。

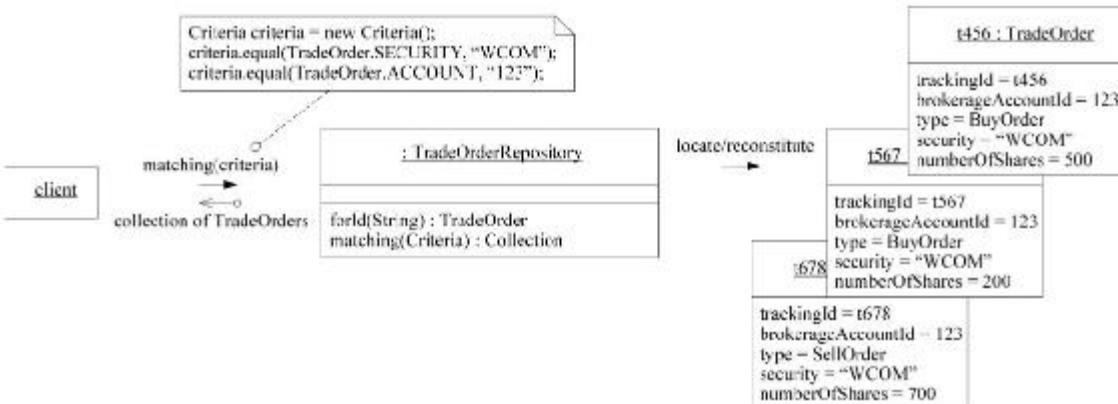


圖6-20 在一個複雜的REPOSITORY中，用一種靈活的、聲明式的SPECIFICATION來表述一個搜索條件

即使一個REPOSITORY的設計採取了靈活的查詢方式，也應該允許添加專門的硬編碼查詢。這些查詢作為便捷的方法，可以封裝常用查詢或不返回對像（如返回的是選中對象的匯總計算）的查詢。不支持這些特殊查詢方式的框架有可能會扭曲領域設計，或是乾脆被開發人員棄之不用。

### **6.3.2 客戶代碼可以忽略REPOSITORY的實現，但開發人員不能忽略**

持久化技術的封裝可以使得客戶變得十分簡單，並且使客戶與REPOSITORY的實現之間完全解耦。但像一般的封裝一樣，開發人員必須知道在封裝背後都發生了什麼事情。在使用REPOSITORY時，不同的使用方式或工作方式可能會對性能產生極大的影響。

Kyle Brown曾告訴過我他的一段經歷，有一次他被請去解決一個基於WebSphere的製造業應用程序的問題，當時這個程序正向生產環境部署。系統在運行幾小時後會莫名其妙地耗盡內存。Kyle在檢查代碼後發現了原因：在某一時刻，系統需要將工廠中每件產品的信息匯總到一起。開發人員使用了一個名為all objects（所有對象）的查詢來進行匯總，這個操作對每個對象進行實例化，然後選擇他們所需的數據。這段代碼的結果是一次性將整個數據庫裝入內存中！這個問題在測試中並未發現，原因是測試數據較少。

這是一個明顯的禁忌，而一些更不容易注意到的疏忽可能會產生同樣嚴重的問題。開發人員需要理解使用封裝行為的隱含問題，但這並不意味著要熟悉實現的每個細節。設計良好的組件是有顯著特徵的（這是第10章的重點之一）。

正如第5章所討論的那樣，底層技術可能會限制我們的建模選擇。例如，關係數據庫可能對複合對像結構的深度有實際的限制。同

樣，開發人員要獲得REPOSITORY的使用及其查詢實現之間的雙向反饋。

### 6.3.3 REPOSITORY的實現

根據所使用的持久化技術和基礎設施不同，REPOSITORY的實現也將有很大的變化。理想的實現是向客戶隱藏所有內部工作細節（儘管不向客戶的開發人員隱藏這些細節），這樣不管數據是存儲在對像數據庫中，還是存儲在關係數據庫中，或是簡單地保持在內存中，客戶代碼都相同。REPOSITORY將會委託相應的基礎設施服務來完成工作。將存儲、檢索和查詢機制封裝起來是REPOSITORY實現的最基本的特性，如圖6-21所示。

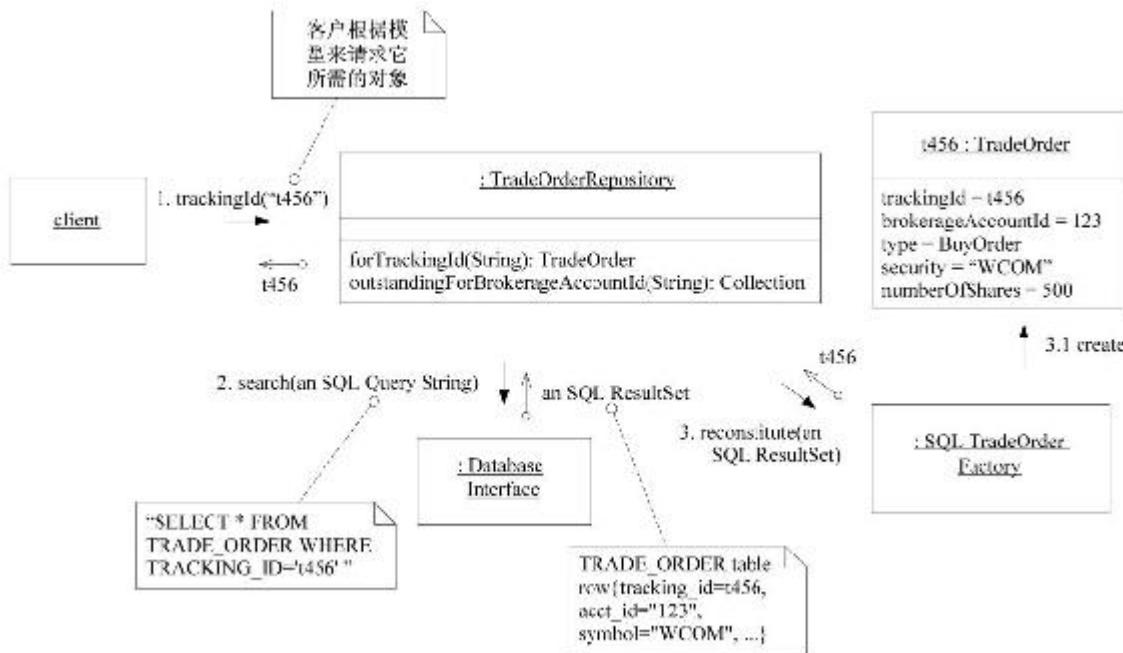


圖6-21 REPOSITORY將底層數據存儲封裝起來

REPOSITORY概念在很多情況下都適用。可能的實現方法有很多，這裡只能列出如下一些需要謹記的注意事項。

對類型進行抽象。REPOSITORY「含有」特定類型的所有實例，但這並不意味著每個類都需要有一個REPOSITORY。類型可以是一個

層次結構中的抽象超類（例如，`TradeOrder`可以是`BuyOrder`或`SellOrder`）。類型可以是一個接口——接口的實現者並沒有層次結構上的關聯，也可以是一個具體類。記住，由於數據庫技術缺乏這樣的多態性質，因此我們將面臨很多約束。

充分利用與客戶解耦的優點。我們可以很容易地更改`REPOSITORY`的實現，但如果客戶直接調用底層機制，我們就很難修改其實現。也可以利用解耦來優化性能，因為這樣就可以使用不同的查詢技術，或在內存中緩存對象，可以隨時自由地切換持久化策略。通過提供一個易於操縱的、內存中的（*in-memory*）啞實現，還能夠方便客戶代碼和領域對象的測試。

將事務的控制權留給客戶。儘管`REPOSITORY`會執行數據庫的插入和刪除操作，但它通常不會提交事務。例如，保存數據後緊接著就提交似乎是很自然的事情，但想必只有客戶才有上下文，從而能夠正確地初始化和提交工作單元。如果`REPOSITORY`不插手事務控制，那麼事務管理就會簡單得多。

通常，項目團隊會在基礎設施層中添加框架，用來支持`REPOSITORY`的實現。`REPOSITORY`超類除了與較低層的基礎設施組件進行協作以外，還可以實現一些基本查詢，特別是要實現的靈活查詢時。遺憾的是，對於類似Java這樣的類型系統，這種方法會使返回的對象只能是`Object`類型，而讓客戶將它們轉換為`REPOSITORY`含有的類型。當然，如果在Java中查詢所返回的對象是集合時，客戶不管怎樣都要執行這樣的轉換。

有關實現`REPOSITORY`的更多指導和一些支持性技術模式（如`QUERY OBJECT`）可以在〔Fowler 2002〕一書中找到。

#### 6.3.4 在框架內工作

在實現**REPOSITORY**這樣的構造之前，需要認真思考所使用的基礎設施，特別是架構框架。這些框架可能提供了一些可用來輕鬆創建**REPOSITORY**的服務，但也可能會妨礙創建**REPOSITORY**的工作。我們可能會發現架構框架已經定義了一種用來獲取持久化對象的等效模式，也有可能定義了一種與**REPOSITORY**完全不同的模式。

例如，你的項目可能會使用**J2EE**。看看這個框架與**MODEL-DRIVEN DESIGN**的模式之間有哪些概念上近似的地方（記住，實體**bean**與**ENTITY**不是一回事），你可能會把實體**bean**和**AGGREGATE**根當作一對類似的概念。在**J2EE**框架中，負責對這些對像進行訪問的構造是**EJB Home**。但如果把**EJB Home**裝飾成**REPOSITORY**的樣子可能會導致其他問題。

一般來講，在使用框架時要順其自然。當框架無法切合時，要想辦法在大方向上保持領域驅動設計的基本原理，而一些不符的細節則不必過分苛求。尋求領域驅動設計的概念與框架中的概念之間的相似性。這裡的假設是除了使用指定框架之外沒有別的選擇。很多**J2EE**項目根本不使用實體**bean**。如果可以自由選擇，那麼應該選擇與你所使用的設計風格相協調的框架或框架中的一些部分。

### **6.3.5 REPOSITORY與FACTORY的關係**

**FACTORY**負責處理對像生命週期的開始，而**REPOSITORY**幫助管理生命週期的中間和結束。當對像駐留在內存中或存儲在對像數據庫中時，這是很好理解的。但通常至少有一部分對像存儲在關係數據庫、文件或其他非面向對象的系統中。在這些情況下，檢索出來的數據必須被重建為對像形式。

由於在這種情況下**REPOSITORY**基於數據來創建對象，因此很多人認為**REPOSITORY**就是**FACTORY**，而從技術角度來看的確如此。但我們最好還是從模型的角度來看待這一問題，前面講過，重建一個已

存儲的對象並不是創建一個新的概念對象。從領域驅動設計的角度來看，FACTORY和REPOSITORY具有完全不同的職責。FACTORY負責製造新對象，而REPOSITORY負責查找已有對象。REPOSITORY應該讓客戶感覺到那些對象就好像駐留在內存中一樣。對象可能必須被重建（的確，可能會創建一個新實例），但它是同一個概念對象，仍舊處於生命週期的中間。

REPOSITORY也可以委託FACTORY來創建一個對象，這種方法（雖然實際很少這樣做，但在理論上是可行的）可用於從頭開始創建對象，此時就沒有必要區分這兩種看問題的角度了，如圖6-22所示。

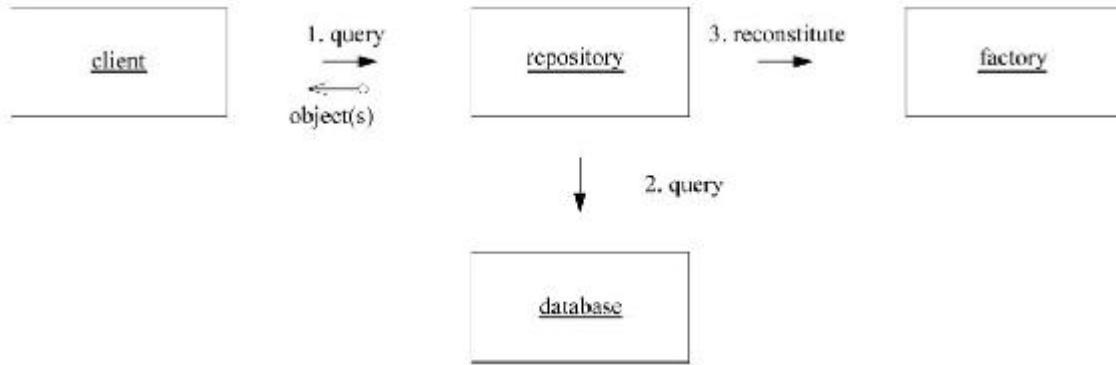


圖6-22 REPOSITORY使用FACTORY來重建一個已有對象

這種職責上的明確區分還有助於FACTORY擺脫所有持久化職責。FACTORY的工作是用數據來實例化一個可能很複雜的對象。如果產品是一個新對象，那麼客戶將知道在創建完成之後應該把它添加到REPOSITORY中，由REPOSITORY來封裝對像在數據庫中的存儲，如圖6-23所示。

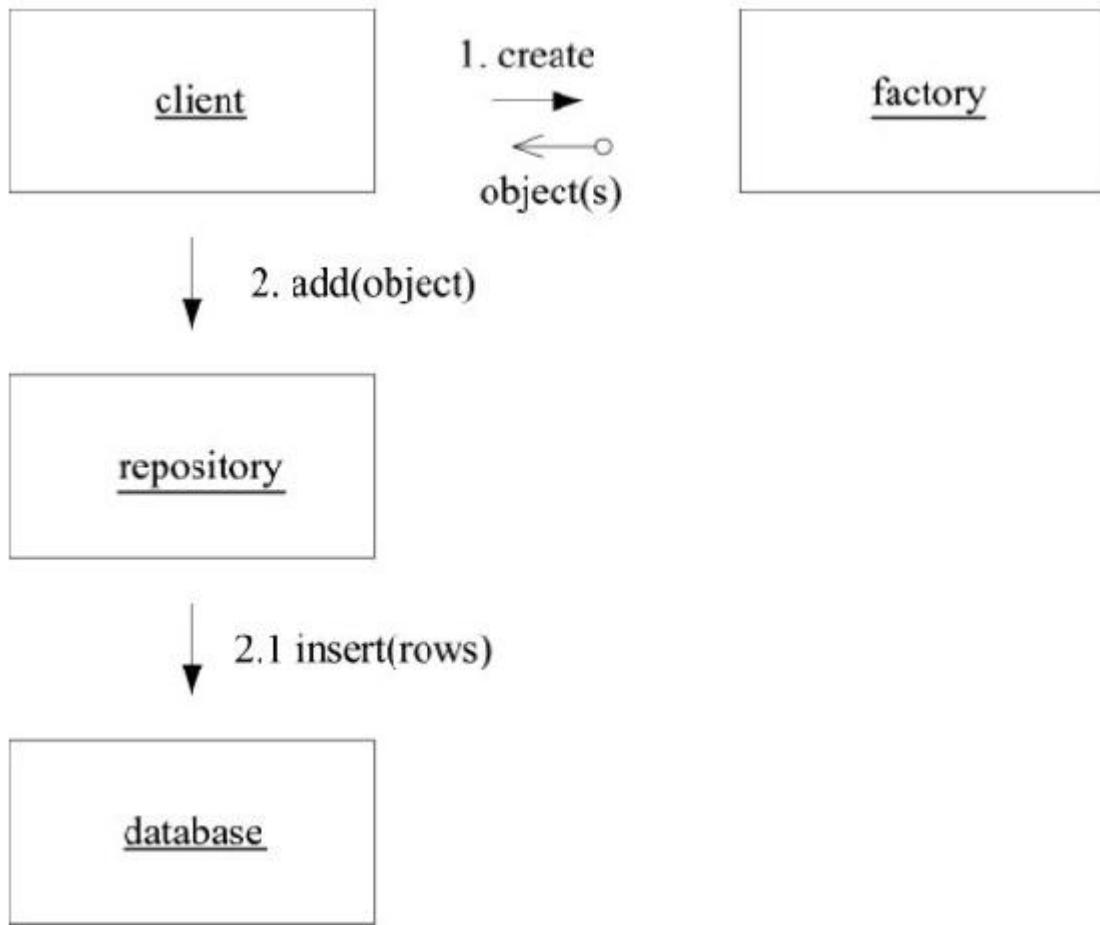


圖6-23 客戶使用REPOSITORY來存儲新對像

另一種情況促使人們將FACTORY和REPOSITORY結合起來使用，這就是想要實現一種「查找或創建」功能，即客戶描述它所需的對象，如果找不到這樣的對象，則為客戶新創建一個。我們最好不要追求這種功能，它不會帶來多少方便。當將ENTITY和VALUE OBJECT區分開時，很多看上去有用的功能就不復存在了。需要VALUE OBJECT的客戶可以直接請求FACTORY來創建一個。通常，在領域中將新對像和原有對象區分開是很重要的，而將它們組合在一起的框架實際上只會使局面變得混亂。

## 6.4 為關係數據庫設計對像

在以面向對像技術為主的軟件系統中，最常用的非對像組件就是關係數據庫。這種現狀產生了混合使用範式的常見問題（參見第5章）。但與大部分其他組件相比，數據庫與對像模型的關係要緊密得多。數據庫不僅僅與對像進行交互，而且它還把構成對象的數據存儲為持久化形式。已經有大量的文獻對於如何將對像映射到關係表以及如何有效存儲和檢索它們這樣的技術挑戰進行了討論。最近的一篇討論可參見[Fowler 2002]一書。有一些相當完善的工具可用來創建和管理它們之間的映射。除了技術上的難點以外，這種不匹配可能對對像模型產生很大的影響。

有3種常見情況：

- (1) 數據庫是對象的主要存儲庫；
- (2) 數據庫是為另一個系統設計的；
- (3) 數據庫是為這個系統設計的，但它的任務不是用於存儲對象。

如果數據庫模式（*database schema*）是專門為對像存儲而設計的，那麼接受模型的一些限制是值得的，這樣可以讓映射變得簡單一點。如果在數據庫模式設計上沒有其他的要求，那麼可以精心設計數據庫結構，以便使得在更新數據時能更安全地保證聚合的完整性，並使數據更新變得更加高效。從技術上來看，關係表的設計不必反映出領域模型。映射工具已經非常完善了，足以消除二者之間的巨大差別。問題在於多個重疊的模型過於複雜了。**MODEL-DRIVEN DESIGN**的很多關於避免將分析和設計模型分開的觀點，也同樣適用於這種不匹配問題。這確實會犧牲一些對像模型的豐富性，而且有時必須在數據庫設計中做出一些折中（如有些地方不能規範化）。但如果不做這些犧牲就會冒另一種風險，那就是模型與實現之間失去了緊密的耦合。這種方法並不要必須使用一種簡單的、一個對像/一個表的映射。

依靠映射工具的功能，可以實現一些聚合或對象的組合。但至關重要的是：映射要保持透明，並易於理解——能夠通過審查代碼或閱讀映射工具中的條目就搞明白。

當數據庫被視作對像存儲時，數據模型與對像模型的差別不應太大（不管映射工具有多麼強大的功能）。可以犧牲一些對像關係的豐富性，以保證它與關係模型的緊密關聯。如果有助於簡化對像映射的話，不妨犧牲某些正式的關係標準（如規範化）。

對像系統外部的過程不應該訪問這樣的對象存儲。它們可能會破壞對像必須滿足的固定規則。此外，它們的訪問將會鎖定數據模型，這樣使得在重構對像時很難修改模型。

另一方面，很多情況下數據是來自遺留系統或外部系統的，而這些系統從來沒打算被用作對象的存儲。在這種情況下，同一個系統中就會有兩個領域模型共存。第14章將深入討論這個問題。或許與另一個系統中隱含的模型保持一致有一定的道理，也可能更好的方法是使這兩個模型完全不同。

允許例外情況的另一個原因是性能。為瞭解決執行速度的問題，有時可能需要對設計做出一些非常規的修改。

但大多數情況下關係數據庫是面向對像領域中的持久化存儲形式，因此簡單的對應關係才是最好的。表中的一行應該包含一個對象，也可能還包含**AGGREGATE**中的一些附屬項。表中的外鍵應該轉換為對另一個**ENTITY**對象的引用。有時我們不得不違背這種簡單的對應關係，但不應該由此就全盤放棄簡單映射的原則。

**UBIQUITOUS LANGUAGE**可能有助於將對像和關係組件聯繫起來，使之成為單一的模型。對像中的元素的名稱和關聯應該嚴格地對應於關係表中相應的項。儘管有些功能強大的映射工具使這看上去有些多此一舉，但關係中的微小差別可能引發很多混亂。

對像世界中越來越盛行的重構實際上並沒有對關係數據庫設計造成多大的影響。此外，一些嚴重的數據遷移問題也使人們不願意對數據庫進行頻繁的修改。這可能會阻礙對像模型的重構，但如果對像模型和數據庫模型開始背離，那麼很快就會失去透明性。

最後，有些原因使我們不得不使用與對像模型完全不同的數據庫模式，即使數據庫是專門為我們的系統創建的。數據庫也有可能被其他一些不對對像進行實例化的軟件使用。即使當對象的行為快速變化或演變的時候，數據庫可能並不需要修改。讓模型與數據庫之間保持鬆散的關聯是很有吸引力的。但這種結果往往是無意為之，原因是團隊沒有保持數據庫與模型之間的同步。如果有意將兩個模型分開，那麼它可能會產生更整潔的數據庫模式，而不是一個為了與早前的對象模型保持一致而到處都是折中處理的拙劣的數據庫模式。

# 第7章 使用語言：一個擴展的示例

前面三章介紹了一種模式語言，它可以對模型的細節進行精化，並可以嚴格遵守**MODEL-DRIVEN DESIGN**。前面的示例基本上一次只應用一種模式，但在實際的項目中，必須將它們結合起來使用。本章介紹一個比較全面的示例（當然還是遠遠比實際項目簡單）。這個示例將通過一個假想團隊處理需求和實現問題，並開發出一個**MODEL-DRIVEN DESIGN**，來一步步介紹模型和設計的精化過程，其間會展示所遇到的阻力，以及如何運用第二部分討論的模式來解決它們。

## 7.1 貨物運輸系統簡介

假設我們正在為一家貨運公司開發新軟件。最初的需求包括3項基本功能：

- (1) 跟蹤客戶貨物的主要處理；
- (2) 事先預約貨物；
- (3) 當貨物到達其處理過程中的某個位鎔時，自動向客戶寄送發票。

在實際的項目中，需要花費一些時間，並經過多次迭代才能得到清晰的模型。本書的第三部分將深入討論這個發現過程。這裡，我們先從一個已包含所需概念並且形式合理的模型開始，我們將通過調整模型的細節來支持設計。

這個模型將領域知識組織起來，並為團隊提供了一種語言。我們可以做出像下面這樣的陳述。

「一個Cargo（貨物）涉及多個Customer（客戶），每個Customer承擔不同的角色。」

「Cargo的運送目標已指定。」

「由一系列滿足Specification（規格）的Carrier Movement（運輸動作）來完成運送目標。」

圖7-1顯示的模型中，每個對象都有明確的意義：

Handling Event（處理事件）是對Cargo採取的不同操作，如將它裝上船或清關。這個類可以被細化為一個由不同種類的事件（如裝貨、卸貨或由收貨人提貨）構成的層次結構。

Delivery Specification（運送規格）定義了運送目標，這至少包括目的地和到達日期，但也可能更為複雜。這個類遵循規格模式（參見第9章）。

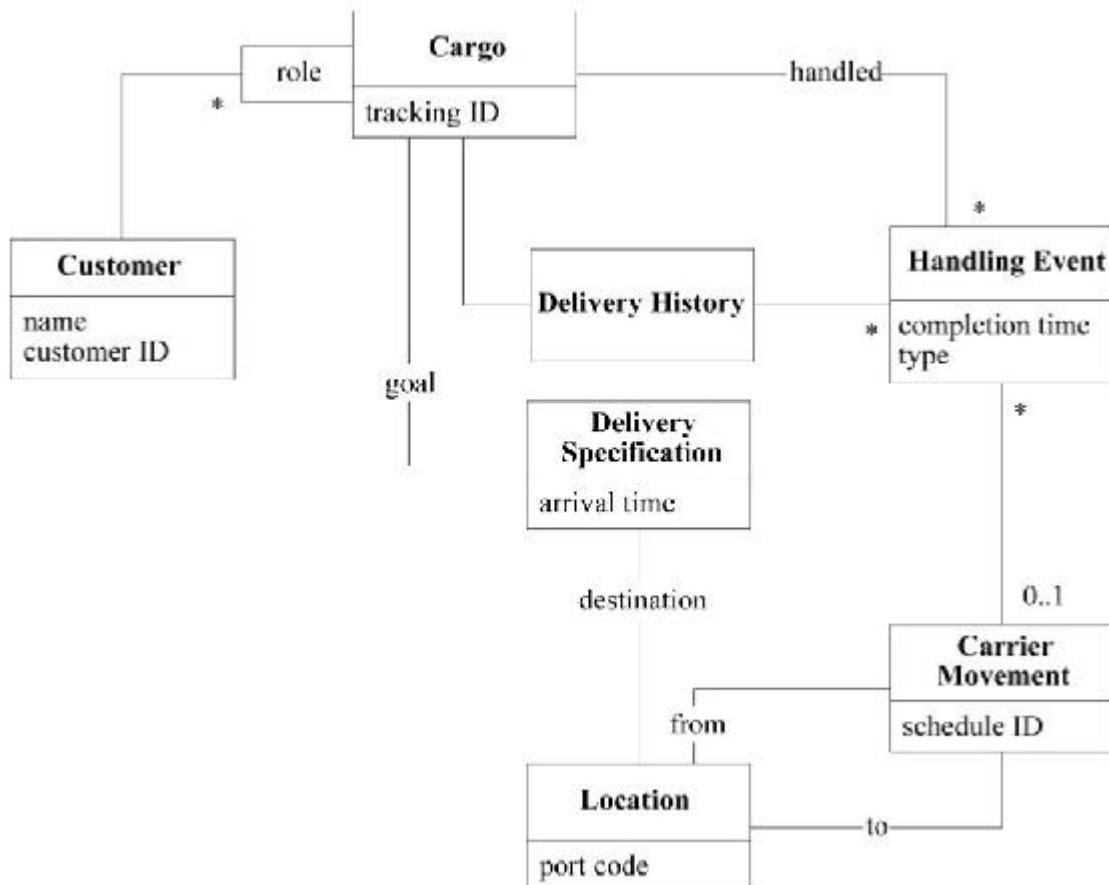


圖7-1 表示貨運領域模型的類圖

Delivery Specification的職責本來可以由Cargo對像承擔，但將Delivery Specification抽象出來至少有以下3個優點。

(1) 如果沒有Delivery Specification，Cargo對象就需要負責提供用於指定運送目標的所有屬性和關聯。這會把Cargo對像搞亂，使它難以理解或修改。

(2) 當將模型作為一個整體來解釋時，這個抽象使我們能夠輕鬆且安全地省略掉細節。例如，Delivery Specification中可能還封裝了其他標準，但就圖7-1所要展示的細節而言，可以不必將其顯示出來。這個圖告訴讀者存在運送規格，但其細節並非思考的重點（事實上，過後修改細節也很容易）。

(3) 這個模型具有更強的表達力。Delivery Specification清楚地表明：運送Cargo的具體方式沒有明確規定，但它必須完成Delivery Specification中規定的目標。

Customer在運輸中所承擔的部分是按照角色（role）來區分的，如shipper（託運人）、receiver（收貨人）、payer（付款人）等。由於一個Cargo只能由一個Customer來承擔某個給定的角色，因此它們之間的關聯是限定的多對一關係，而不是多對多。角色可以被簡單地實現為字符串，當需要其他行為的時候，也可以將它實現為類。

Carrier Movement表示由某個Carrier（如一輛卡車或一艘船）執行的從一個Location（地點）到另一個Location的旅程。Cargo被裝上Carrier後，通過Carrier的一個或多個Carrier Movement，就可以在不同地點之間轉移。

Delivery History（運送歷史）反映了Cargo實際上發生了什麼事情，它與Delivery Specification正好相對，後者描述了目標。Delivery History對象可以通過分析最後一次裝貨和卸貨以及對應的Carrier

**Movement**的目的地來計算貨物的當前位臘。成功的運送將會得到一個滿足**Delivery Specification**目標的 **Delivery History**。

用於實現上述需求的所有概念都已包含在這個模型中，並假定已經有適當的機制來保存對像、查找相關對像等。這些實現問題不在模型中處理，但它們必須在設計中加以考慮。

為了建立一個健壯的實現，這個模型需要更清晰和嚴密一些。

記住，一般情況下，模型的精化、設計和實現應該在迭代開發過程中同步進行。但在本章中，為了使解釋更加清楚，我們從一個相對成熟的模型開始，並嚴格限定修改的唯一動機是保證模型與具體實現相關聯，在實現時採用構造塊模式。

一般來說，當為了更好地支持設計而對模型進行精化時，也應該讓模型反映出對領域的新理解。但在本章中，仍然是為了使解釋更加清楚，嚴格限定修改的動機在於保證模型與具體實現相關聯，在實現時採用構造塊模式。

## 7.2 隔離領域：引入應用層

為了防止領域的職責與系統的其他部分混雜在一起，我們應用 **LAYERED ARCHITECTURE** 把領域層劃分出來。

無需深入分析，就可以識別出三個用戶級別的應用程序功能，我們可以將這三個功能分配給三個應用層類。

(1) 第一個類是 **Tracking Query** ( 跟蹤查詢 )，它可以訪問某個 **Cargo** 過去和現在的處理情況。

(2) 第二個類是 **Booking Application** ( 預訂應用 )，它允許註冊一個新的 **Cargo**，並使系統準備好處理它。

(3) 第三個類是Incident Logging Application（事件日誌應用），它記錄對Cargo的每次處理（提供通過Tracking Query查找的信息）。

這些應用層類是協調者，它們只是負責提問，而不負責回答，回答是領域層的工作。

## **7.3 將ENTITY和VALUE OBJECT區別開**

依次考慮每個對象，看看這個對象是必須被跟蹤的實體還是僅表示一個基本值。首先，我們來看一些比較明顯的情況，然後考慮更含糊的情況。

### **Customer**

我們從一個簡單的對象開始。Customer對像表示一個人或一家公司，從一般意義上來講它是一個實體。Customer對像顯然有對用戶來說很重要的標識，因此它在模型中是一個ENTITY。那麼如何跟蹤它呢？在某些情況下可以使用Tax ID（納稅號），但如果是跨國公司就無法使用了。這個問題需要諮詢領域專家。我們與運輸公司的業務人員討論這個問題，發現公司已經建立了客戶數據庫，其中每個Customer在第一次聯繫銷售時被分配了一個ID號。這種ID已經在整個公司中使用，因此在我們的軟件中使用這種ID號就可以與那些系統保持標識的連貫性。ID號最初是手工錄入的。

### **Cargo**

兩個完全相同的貨箱必須要區分開，因此Cargo對象是ENTITY。在實際情況中，所有運輸公司會為每件貨物分配一個跟蹤ID。這個ID是自動生成的、對用戶可見，而且在本例中，在預訂時可能還要發送給客戶。

### **Handling Event和Carrier Movement**

我們關心這些獨立事件是因為通過它們可以跟蹤正在發生的事情。它們反映了真實世界的事件，而這些事件一般是不能互換的，因此它們是ENTITY。每個Carrier Movement都將通過一個代碼來識別，這個代碼是從運輸調度表得到的。

在與領域專家的另一次討論中，我們發現Handling Event有一種唯一的識別方法，那就是使用Cargo ID、完成時間和類型的組合。例如，同一個Cargo不會在同一時間既裝貨又卸貨。

### **Location**

名稱相同的兩個地點並不是同一個位臘。經緯度可以作為唯一鍵，但這並不是一個非常可行的方案，因為系統的大部分功能並不關心經緯度是多少，而且經緯度的使用相當複雜。Location更可能是某種地理模型的一部分，這個模型根據運輸航線和其他特定於領域的關注點將地點關聯起來。因此，使用自動生成的內部任意標識符就足夠了。

### **Delivery History**

這是一個比較複雜的對象。Delivery History是不可互換的，因此它是ENTITY。但Delivery History與Cargo是一對一關係，因此它實際上並沒有自己的標識。它的標識來自於擁有它的Cargo。當對AGGREGATE進行建模時這個問題會變得更清楚。

### **Delivery Specification**

儘管它表示了Cargo的目標，但這種抽象並不依賴於Cargo。它實際上表示某些Delivery History的假定狀態。運送貨物實際上就是讓Cargo的Delivery History最後滿足該Cargo的Delivery Specification。如果有兩個Cargo去往同一地點，那麼它們可以用同一個Delivery Specification，但它們不會共用同一個Delivery History，儘管運送歷史

都是從同一個狀態（空）開始。因此，Delivery Specification是VALUE OBJECT。

#### Role和其他屬性

Role表示了有關它所限定的關聯的一些信息，但它沒有歷史或連續性。因此它是一個VALUE OBJECT，可以在不同的Cargo/Customer 關聯中共享它。

其他屬性（如時間戳或名稱）都是VALUE OBJECT。

## 7.4 設計運輸領域中的關聯

圖7-1中的所有關聯都沒有指定遍歷方向，但雙向關聯在設計中容易產生問題。此外，遍歷方向還常常反映出對領域的洞悉，使模型得以深化。

如果Customer對它所運送的每個Cargo都有直接引用，那麼這對長期、頻繁託運貨物的客戶將會非常不便。此外，Customer這一概念並非只與Cargo相關。在大型系統中，Customer可能具有多種角色，以便與許多對像交互，因此最好不要將它限定為這種具體的職責。如果需要按照Customer來查找Cargo，那麼可以通過數據庫查詢來完成。本章後面討論REPOSITORY時還會回頭討論這個問題。

如果我們的應用程序要對一系列貨船進行跟蹤，那麼從Carrier Movement遍歷到Handling Event將是很重要的。但我們的業務只需跟蹤Cargo，因此只需從Handling Event遍歷到Carrier Movement就能滿足我們的業務需求。由於捨棄了具有多重性的遍歷方向，實現簡化為簡單的對象引用。

圖7-2解釋了其他設計決策背後的原因。

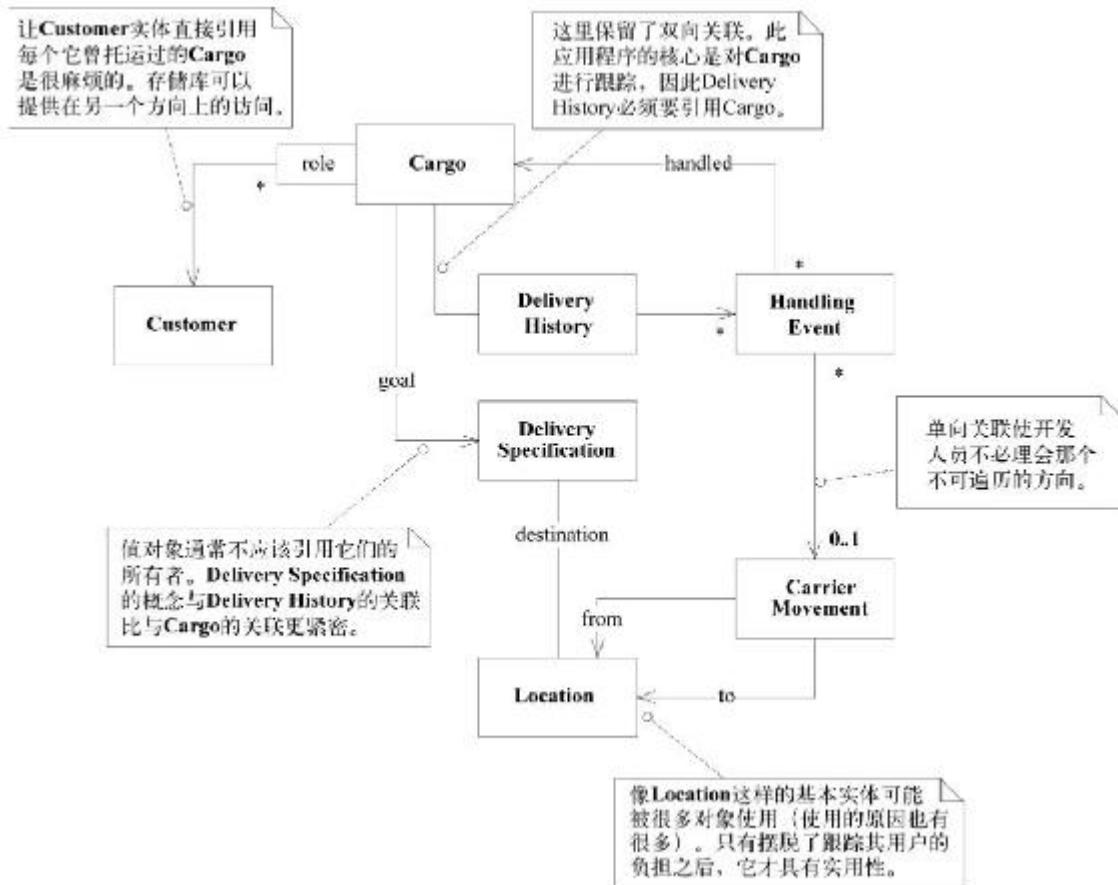


圖7-2 在一些關聯上對遍歷方向進行了約束

模型中存在一個循環引用：Cargo知道它的Delivery History，Delivery History中保存了一系列的Handling Event，而Handling Event又反過來指向Cargo。很多領域在邏輯上都存在循環引用，而且循環引用在設計中有時是必要的，但它們維護起來很複雜。在選擇實現時，應該避免把必須同步的信息保存在兩個不同的地方，這樣對我們的工作很有幫助。對於這個例子，我們可以在初期原型中使用一個簡單但不太健壯的實現（用Java語言）——在Delivery History中提供一個List對象，並把Handling Event都放到這個List對像中。但在某些時候，我們可能不想使用集合，以便能夠用Cargo作為鍵來執行數據庫查詢。在選擇存儲庫時，我們還會討論到這一點。如果查詢歷史的操作相對來說不是很多，那麼這種方法可以提供很好的性能、簡化維護

並減少添加Handling Event的開銷。如果這種查詢很頻繁，那麼最好還是直接引用。這種設計上的折中其實就是在實現的簡單性和性能之間達成一個平衡。模型還是同一個模型，它包含了循環關聯和雙向關聯。

## 7.5 AGGREGATE邊界

Customer、Location和Carrier Movement都有自己的標識，而且被許多Cargo共享，因此，它們在各自的AGGREGATE中必須是根，這些聚合除了包含各自的屬性之外，可能還包含其他比這裡討論的細節級別更低層的對象。Cargo也是一個明顯的AGGREGATE根，但把它的邊界畫在哪裡還需要仔細思考一下。

如圖7-3所示，Cargo AGGREGATE可以把一切因Cargo而存在的事物包含進來，這當中包括Delivery History、Delivery Specification和Handling Event。這很適合Delivery History，因為沒人會在不知道Cargo的情況下直接去查詢Delivery History。因為Delivery History不需要直接的全局訪問，而且它的標識實際上只是由Cargo 派生出的，因此很適合將Delivery History放在Cargo的邊界之內，並且它也無需是一個AGGREGATE根。Delivery Specification是一個VALUE OBJECT，因此將它包含在Cargo AGGREGATE中也不複雜。

Handling Event就是另外一回事了。前面已經考慮了兩種與其有關的數據庫查詢，一種是當不想使用集合時，用查找某個Delivery History的Handling Event作為一種可行的替代方法，這種查詢是位於Cargo AGGREGATE內部的本地查詢；另一種查詢是查找裝貨和準備某次Carrier Movement時所進行的所有操作。在第二種情況中，處理

Cargo的活動看起來是有意義的（即使與Cargo本身分開來考慮時也是如此），因此Handling Event應該是它自己的AGGREGATE的根。

## 7.6 選擇REPOSITORY

在我們的設計中，有5個ENTITY是AGGREGATE的根，因此在選擇存儲庫時只需考慮這5個實體，因為其他對象都不能有REPOSITORY。

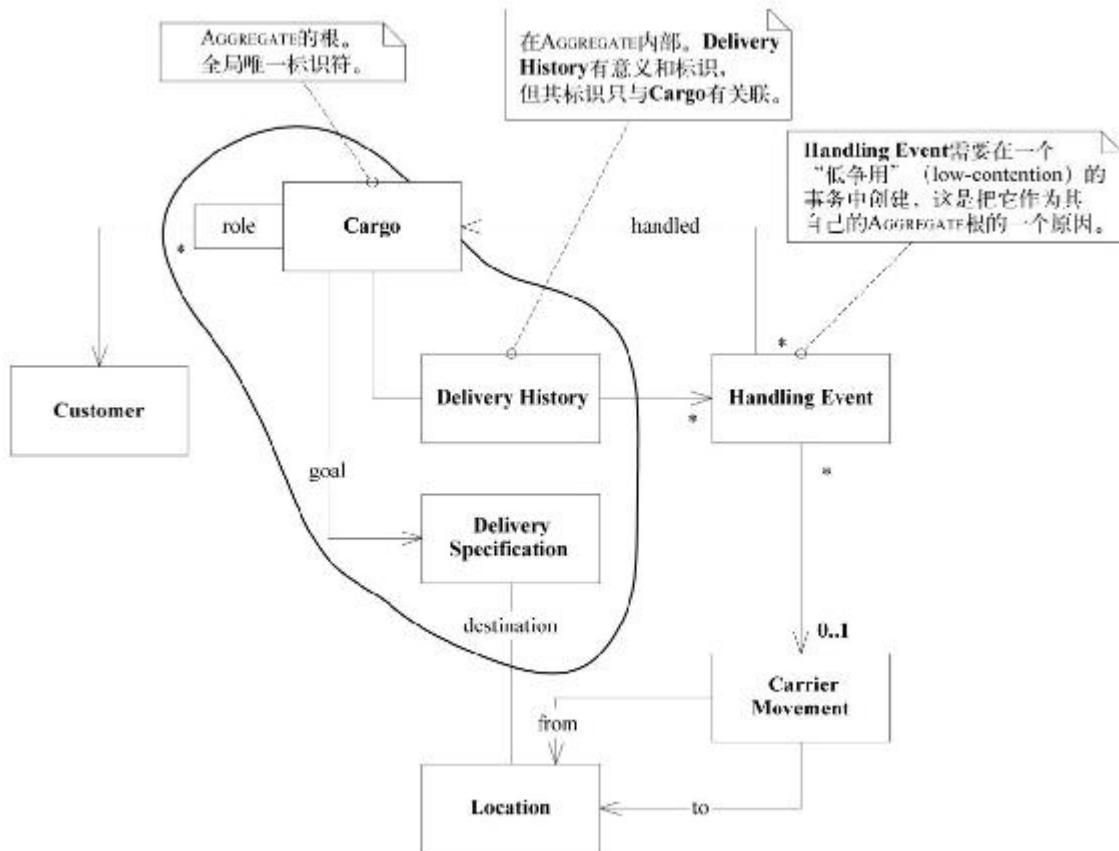


圖7-3 模型中的AGGREGATE邊界（注意：邊界之外的ENTITY是其自己的AGGREGATE的根）

為了確定這5個實體當中哪些確實需要REPOSITORY，必須回頭看一下應用程序的需求。要想通過Booking Application進行預訂，用戶需要選擇承擔不同角色（託運人、收貨人等）的Customer。因此需要

一個 Customer Repository。在指定貨物的目的地時還需要一個 Location，因此還需要創建一個 Location Repository。

用戶需要通過 Activity Logging Application 來查找裝貨的 Carrier Movement，因此需要一個 Carrier Movement Repository。用戶還必須告訴系統哪個 Cargo 已經完成了裝貨，因此還需要一個 Cargo Repository，如圖 7-4 所示。

我們沒有創建 Handling Event Repository，因為我們決定在第一次迭代中將它與 Delivery History 的關聯實現為一個集合，而且應用程序並不需要查找在一次 Carrier Movement 中都裝載了什麼貨物。這兩個原因都有可能發生變化，如果確實改變了，可以增加一個 REPOSITORY。

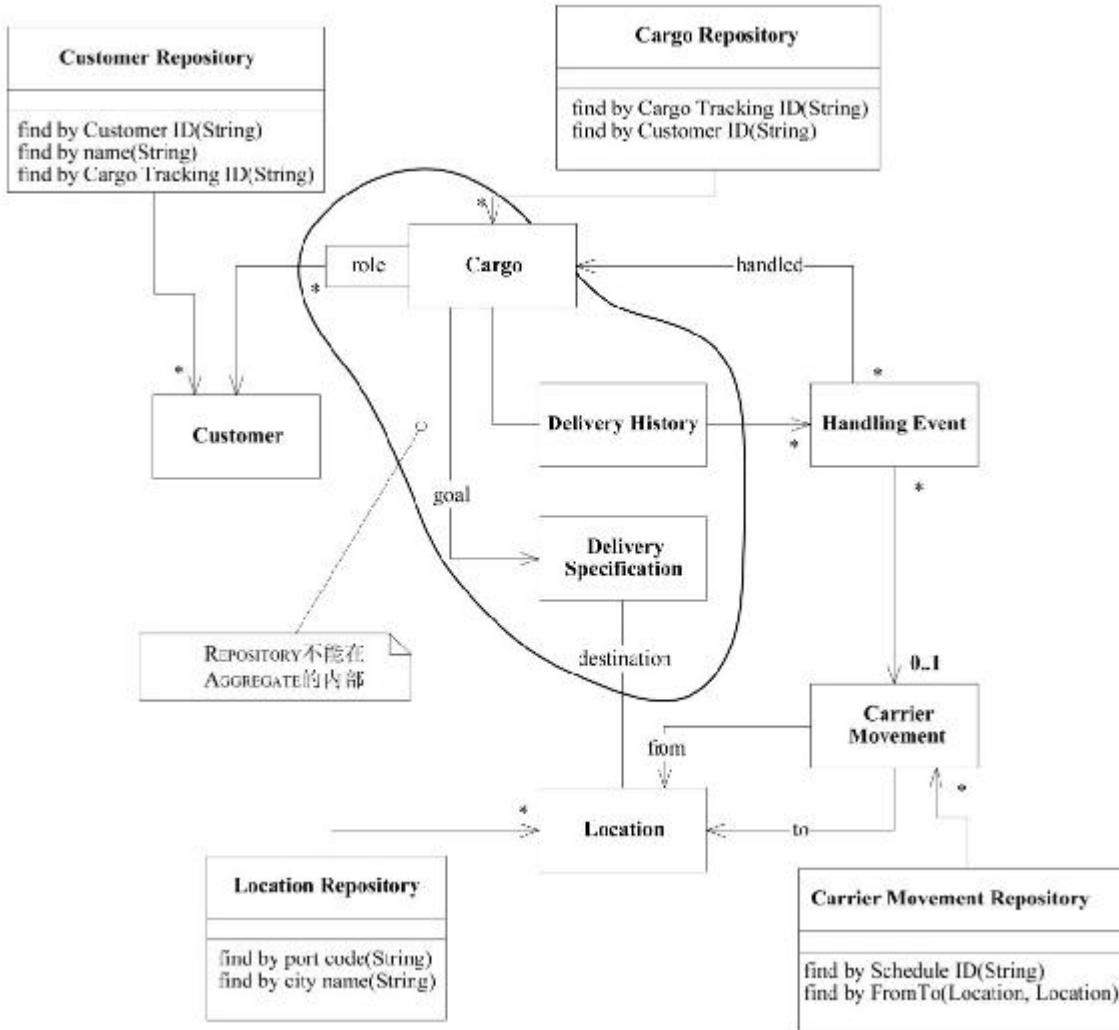


圖7-4 REPOSITORY提供了對所選AGGREGATE根的訪問

## 7.7 場景走查

為了覆核這些決策，我們需要經常走查場景，以確保能夠有效地解決應用問題。

### 7.7.1 應用程序特性舉例：更改Cargo的目的地

有時Customer會打電話說：「糟了！我們原來說把貨物運到Hackensack，但實際上應該運往Hoboken。」既然我們提供運輸服務，就一定要讓系統能夠進行這樣的修改。

Delivery Specification是一個VALUE OBJECT，因此最簡單的方法是拋棄它，再創建一個新的，然後使用Cargo上的setter方法把舊值替換成新值。

### 7.7.2 應用程序特性舉例：重複業務

用戶指出，相同Customer的重複預訂往往是類似的，因此他們想要將舊Cargo作為新Cargo的原型。應用程序應該允許用戶在存儲庫中查找一個Cargo，然後選擇一條命令來基於選中的Cargo創建一個新Cargo。我們將利用PROTOTYPE模式[Gamma et al.1995]來設計這一功能。

Cargo是一個ENTITY，而且是AGGREGATE的根。因此在複製它時要非常小心，其AGGREGATE邊界內的每個對象或屬性的處理都需要仔細考慮，下面逐個來看一下。

**Delivery History**：應創建一個新的、空的Delivery History，因為原有Delivery History的歷史並不適用。這是AGGREGATE內部的實體的常見情況。

**Customer Roles**：應該複製存有Customer引用的Map（或其他集合）——這些引用通過鍵來標識，鍵也要一起複製，這些Customer在新的運輸業務中可能擔負相同的角色。但必須注意不要複製Customer對像本身。在複製之後，應該保證和原來的Cargo引用相同的Customer對象，因為它們是AGGREGATE邊界之外的ENTITY。

**Tracking ID**：我們必須提供一個新的Tracking ID，它應該來自創建新Cargo時的同一個來源。

注意，我們複製了Cargo AGGREGATE邊界內部的所有對象，並對副本進行了一些修改，但這並沒有對AGGREGATE邊界之外的對象產生任何影響。

## 7.8 對象的創建

### 7.8.1 Cargo的FACTORY和構造函數

即使為Cargo創建了複雜而精緻的FACTORY，或像「重複業務」一節那樣使用另一個Cargo作為FACTORY，我們仍然需要有一個基本的構造函數。我們希望用構造函數來生成一個滿足固定規則的對象，或者，就ENTITY而言，至少保持其標識不變。

考慮到這些因素，我們可以在Cargo上創建一個FACTORY方法，如下所示：

```
public Cargo copyPrototype(String newTrackingID)
```

或者可以為一個獨立的FACTORY添加以下方法：

```
public Cargo newCargo(Cargo prototype, String newTrackingID)
```

獨立FACTORY還可以把為新Cargo獲取新（自動生成的）ID的過程封裝起來，這樣它就只需要一個參數：

```
public Cargo newCargo(Cargo prototype)
```

這些FACTORY返回的結果是完全相同的，都是一個Cargo，其Delivery History為空，且Delivery Specification為null。

Cargo與Delivery History之間的雙向關聯意味著它們必須要互相指向對方才算是完整的，因此它們必須被一起創建出來。記住，Cargo是AGGREGATE的根，而這個AGGREGATE包含Delivery History。因此，我們可以用Cargo的構造函數或FACTORY來創建Delivery History。Delivery History的構造函數將Cargo作為參數。這樣就可以編寫以下代碼：

```
public Cargo(String id) {  
    trackingID = id;  
    deliveryHistory = new DeliveryHistory(this);  
    customerRoles = new HashMap();  
}
```

結果得到一個新的Cargo，它帶有一個指向它自己的新的Delivery History。由於Delivery History的構造函數只供其AGGREGATE根（即Cargo）使用，這樣Cargo的組成就被封裝起來了。

### 7.8.2 添加Handling Event

貨物在真實世界中的每次處理，都會有人使用Incident Logging Application來輸入一條Handling Event記錄。

每個類都必須有一個基本的構造函數。由於Handling Event是一個ENTITY，所以必須把定義了其標識的所有屬性傳遞給構造函數。如前所述，Handling Event是通過Cargo的ID、完成時間和事件類型的組合來唯一標識的。Handling Event唯一剩下的屬性是與Carrier Movement的關聯，而有些類型的Handling Event甚至沒有這個屬性。綜上，創建一個有效的Handling Event的基本構造函數是：

```
public HandlingEvent(Cargo c, String eventType, Date timeStamp) {  
    handled = c;  
    type = eventType;  
    completionTime = timeStamp;  
}
```

就ENTITY而言，那些非標識作用的屬性通常可以過後再添加。在本例中，Handling Event的所有屬性都是在初始事務中設置的，而且過後不再改變（糾正數據錄入錯誤除外），因此針對每種事件類型，為Handling Event添加一個簡單的FACTORY METHOD（並帶有所有必

要的參數 ) 是很方便的做法 , 這還使得客戶代碼具有更強的表達力 。例如 , loading event ( 裝貨事件 ) 確實涉及一個 Carrier Movement 。

```
public static HandlingEvent newLoading(  
    Cargo c, CarrierMovement loadedOnto, Date timeStamp) {  
    HandlingEvent result =  
        new HandlingEvent(c, LOADING_EVENT, timeStamp);  
    result.setCarrierMovement(loadedOnto);  
    return result;  
}
```

模型中的 Handling Event 是一個抽象 , 它可以把各種具體的 Handling Event 類封裝起來 , 包括裝貨、卸貨、密封、存放以及其他與 Carrier 無關的活動 。它們可以被實現為多個子類 , 或者通過複雜的初始化過程來實現 , 也可以將這兩種方法結合起來使用 。通過在基類 ( Handling Event ) 中為每個類型添加 FACTORY METHOD , 可以將實例創建的工作抽象出來 , 這樣客戶就不必瞭解實現的知識 。FACTORY 會知道哪個類需要被實例化 , 以及應該如何對它初始化 。

遺憾的是 , 事情並不是這麼簡單 。 Cargo → Delivery History → History Event → Cargo 這個引用循環使實例創建變得很複雜 。 Delivery History 保存了與其 Cargo 有關的 Handling Event 集合 , 而且新對像必須作為事務的一部分來添加到這個集合中 ( 見圖 7-5 ) 。如果沒有創建這個反向指針 , 那麼對像間將發生不一致 。

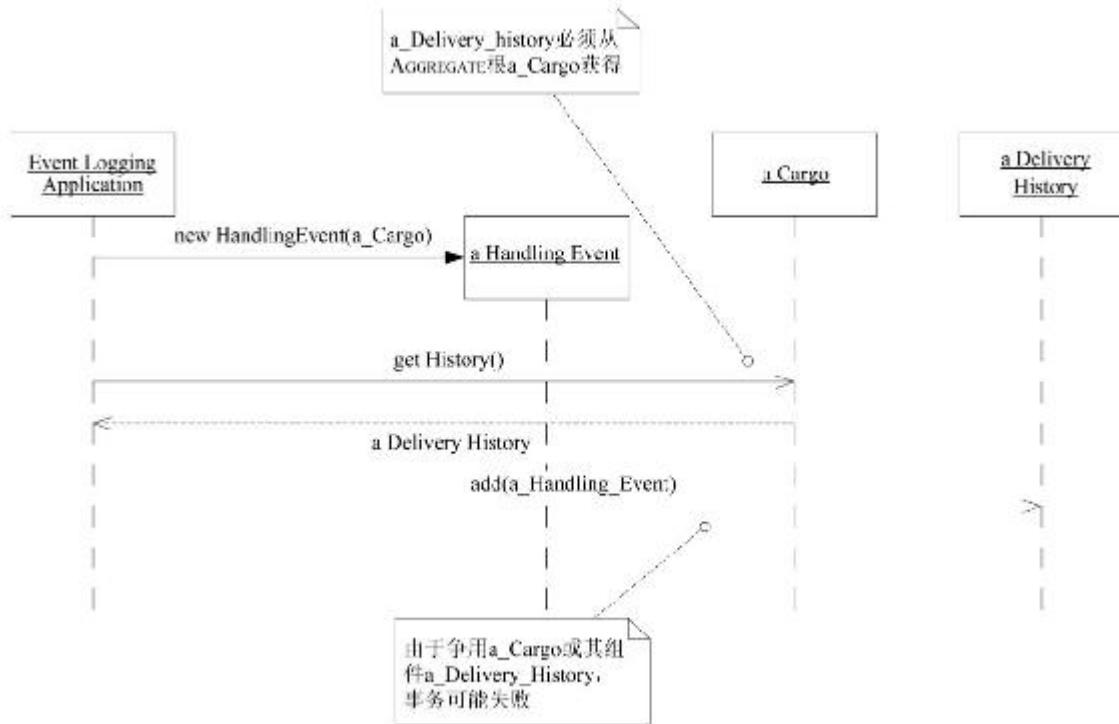


圖7-5 添加Handling Event需要將它插入到Delivery History中

我們可以把反向指針的創建封裝到FACTORY中（並將其放在領域層中——它屬於領域層），但現在我們來看另一種設計，它完全消除了這種彆扭的交互。

## 7.9 停一下，重構：Cargo AGGREGATE的另一種設計

建模和設計並不總是一個不斷向前的過程，如果不經常進行重構，以便利用新的知識來改進模型和設計，那麼建模和設計將會停滯不前。

到目前為止，我們的設計中有幾個蹩腳的地方，雖然這並不影響設計發揮作用，而且設計也確實反映了模型。但設計之初看上去不太重要的問題正漸漸變得棘手。讓我們藉助事後的認識來解決其中一個問題，以便為以後的設計做好鋪墊。

由於添加Handling Event時需要更新Delivery History，而更新Delivery History會在事務中牽涉Cargo AGGREGATE。因此，如果同一時間其他用戶正在修改Cargo，那麼Handling Event事務將會失敗或延遲。輸入Handling Event是需要迅速完成的簡單操作，因此能夠在不發生爭用的情況下輸入Handling Event是一項重要的應用程序需求。這促使我們考慮另一種不同的設計。

我們在Delivery History中可以不使用Handling Event的集合，而是用一個查詢來代替它，這樣在添加Handling Event時就不會在其自己的AGGREGATE之外引起任何完整性問題。如此修改之後，這些事務就不再受到幹擾。如果有很多Handling Event同時被錄入，而相對只有很少的查詢，那麼這種設計更加高效。實際上，如果使用關係數據庫作為底層技術，那麼我們可以設法在底層使用查詢來模擬集合。使用查詢來代替集合還可以減小維護Cargo和Handling Event之間循環引用一致性的難度。

為了使用查詢，我們為Handling Event增加一個REPOSITORY。Handling Event Repository將用來查詢與特定Cargo有關的Event。此外，REPOSITORY還可以提供優化的查詢，以便更高效地回答特定的問題。例如，為了推斷Cargo的當前狀態，常常需要在Delivery History中查找最後一次報告的裝貨或卸貨操作，如果這個查找操作被頻繁地使用，那麼就可以設計一個查詢直接返回相關的Handling Event。而且，如果需要通過查詢找到某次Carrier Movement裝載的所有Cargo，那麼很容易就可以增加這個查詢。

這樣一來，Delivery History就不再有持久狀態了，因此實際上無需再保留它。無論何時需要用Delivery History回答某個問題時，都可以將其生成出來。我們之所以可以生成這個對象——儘管在不斷地重

建這個 Entity，是因為這些對像關聯了相同的 Cargo 對象，而這個 Cargo 對像在 Delivery History 的各個化身間維護了連續性。

循環引用的創建和維護也不再是問題。Cargo Factory 將被簡化，不再需要為新的 Cargo 實例創建一個空的 Delivery History。數據庫空間會略微減少，而且持久化對象的實際數量可能減少很多（在某些對像數據庫中，能容納的持久化對象的數量是有限的）。如果常見的使用模式是：用戶在貨物到達之前很少查詢它的狀態，那麼這種設計可以避免很多不必要的工作。

另一方面，如果我們正在使用對像數據庫，則通過遍歷關聯或顯式的集合來查找對象可能會比通過 REPOSITORY 查詢快得多。如果用戶在使用系統時需要頻繁地列出貨物處理的全部歷史，而不是偶爾查詢最後一次處理，那麼出於性能上的考慮，使用顯式的集合比較有利。此外要記住，現在並不需要查詢「這次 Carrier Movement 上都裝載了什麼」，而且這個要求可能永遠也不會被提出來，因此暫時不必過多地注意該選項。

這些類型的修改和設計折中隨處可見，僅僅在這個簡化的小系統中，我就可以舉出許多示例。但重要的一點是，這些修改和折中僅限於同一個模型內部。通過對 VALUE、ENTITY 以及它們的 AGGREGATE 進行建模（正如我們已經做的那樣），已經大大減小了這些設計修改的影響。例如，在這個示例中，所有的修改都被封裝在 Cargo 的 AGGREGATE 邊界之內。它還需要增加一個 Handling Event Repository，但並不需要重新設計 Handling Event 本身（雖然根據不同的 REPOSITORY 框架細節，可能需要對實現進行一些修改）。

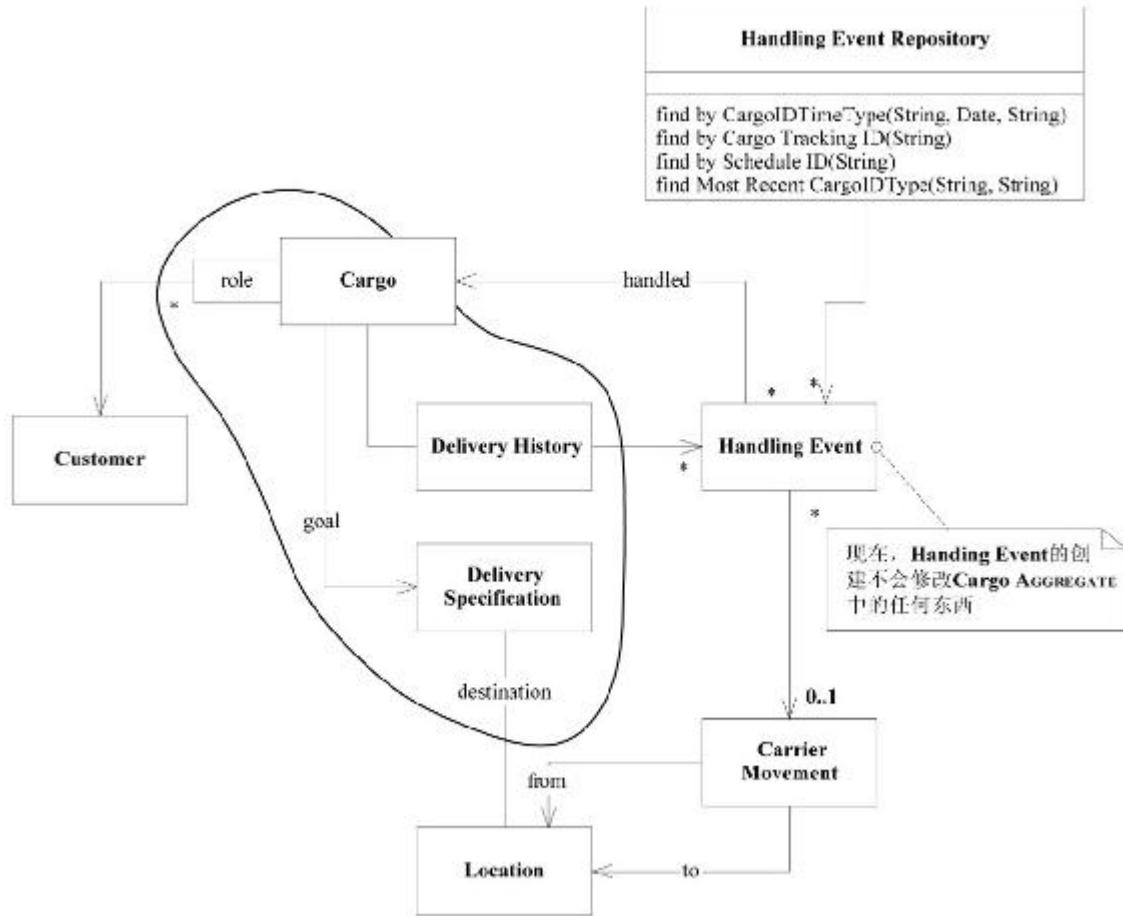


圖7-6 將Delivery History中的Handling Event集合實現為一個查詢，這樣可以使Handling Event的插入變得簡單，而且不會與Cargo AGGREGATE發生爭用

## 7.10 運輸模型中的MODULE

到目前為止，我們只看到了很少的幾個對象，因此MODULE化還不是問題。現在，我們來看看大一點的運輸模型（當然，這仍然是簡化的），從而瞭解一下MODULE的組織怎樣影響模型。

圖7-7展示了劃分整齊的模型，這裡假設該模型是由本書的一位熱心讀者劃分的。這個圖是第5章中所提及的由基礎設施驅動的打包問題的一個變形。在本例中，對象是根據其所遵循的模式來分組的。結果那些在概念上幾乎沒有關係（低內聚）的對象被分到了一

起，而且所有MODULE之間的關聯錯綜複雜（高耦合）。這種打包方式也描述了一件事情，但描述的不是運輸，而是開發人員在那個時候對模型的認識。

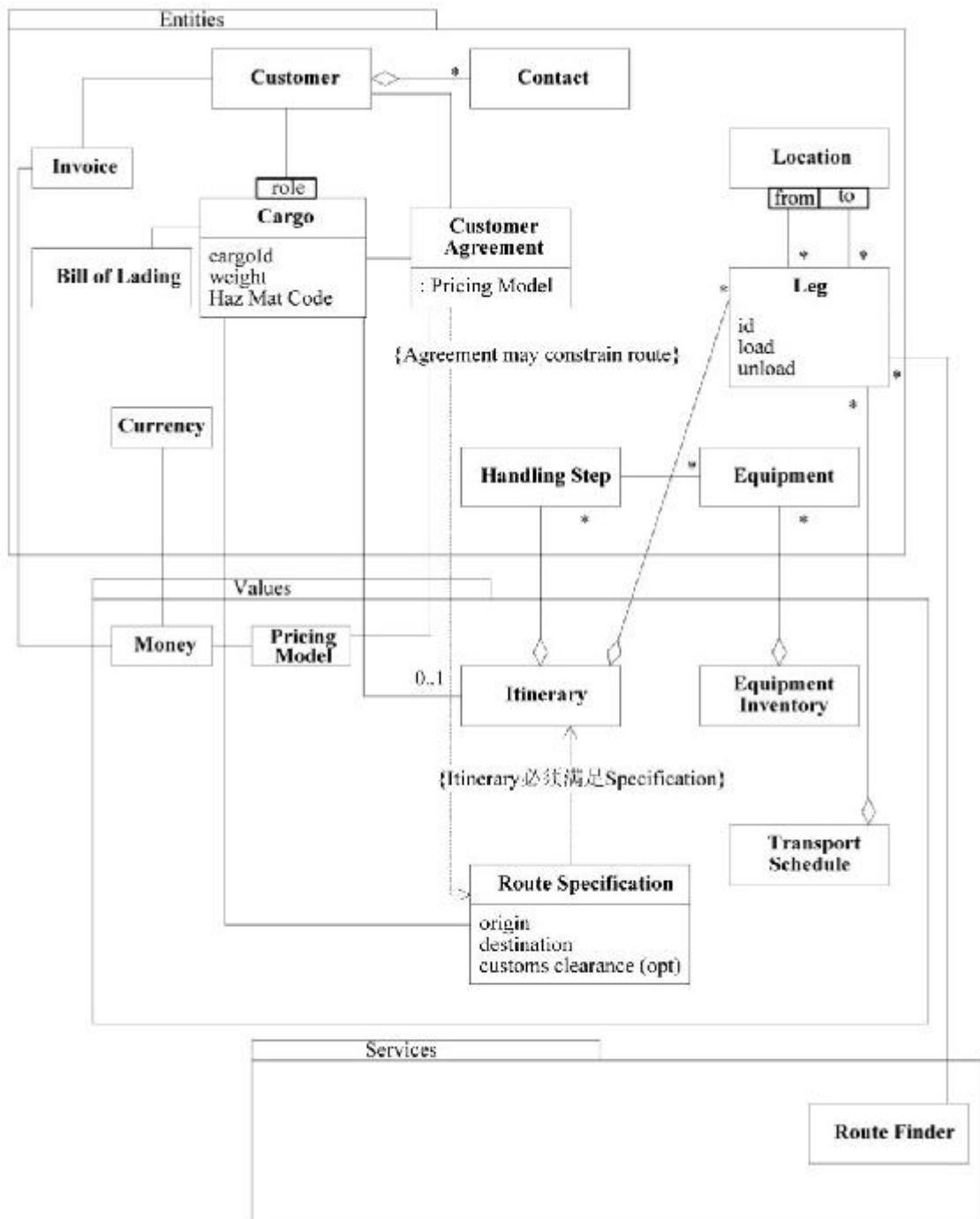


圖7-7 這些MODULE並沒有傳達領域知識

按模式劃分看起來是一個明顯的錯誤，但按照對象是持久對像還是臨時對像來劃分，或者使用任何其他劃分方法，而不是根據對象的意義來劃分，也同樣不靠譜。

相反，我們應該尋找緊密關聯的概念，並弄清楚我們打算向項目中的其他人員傳遞什麼信息。如同應對規模較小的建模決策時，總是會有多種方法可以達成目的。圖7-8顯示了一種直觀的劃分方法。

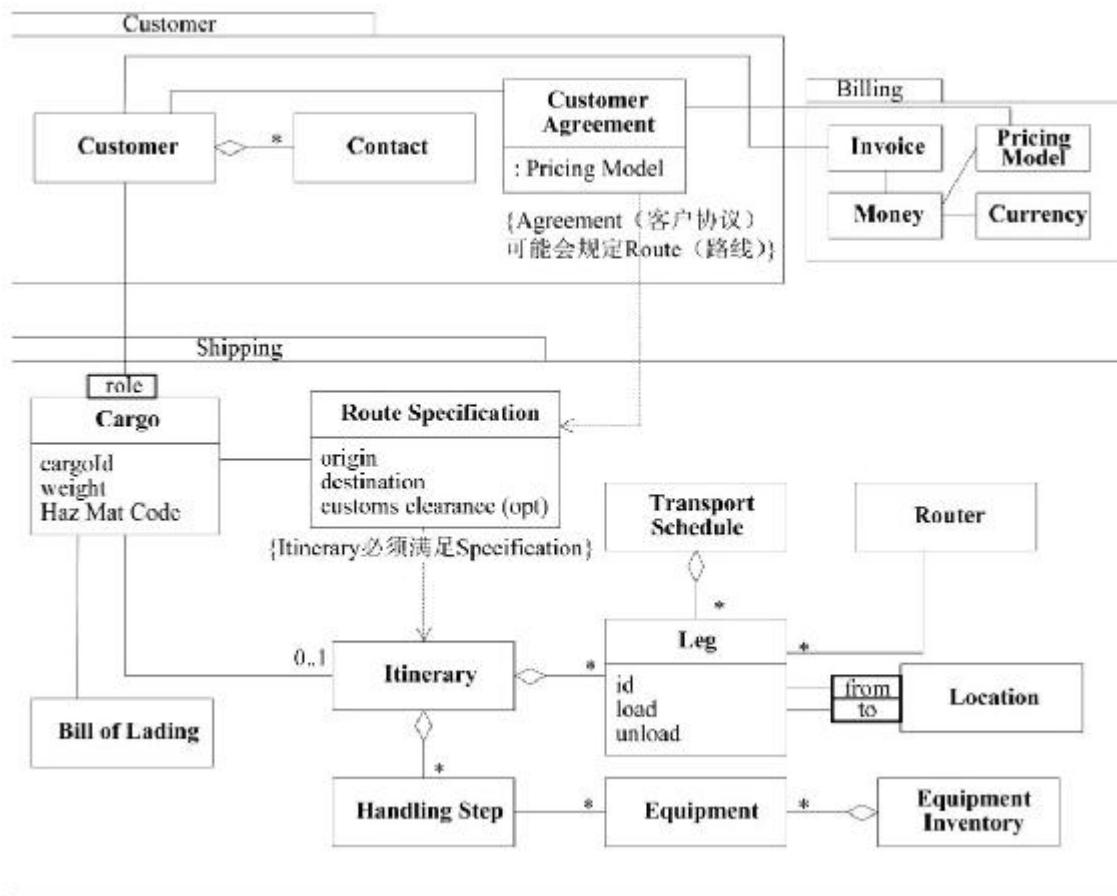


圖7-8 基於寬泛的領域概念的模塊

圖7-8中的MODULE名稱成為團隊語言的一部分。我們的公司為客戶（Customer）運輸貨物（Shipping），因此向他們收取費用（Bill），公司的銷售和營銷人員與Customer磋商並簽署協議，操作人員負責將貨物Shipping到指定目的地，後勤辦公人員負責Billing（處理

帳單），並根據Customer協議開具發票。這就是可以通過這組MODULE描述的業務。

當然，這種直觀的分解可以通過後續迭代來完善，甚至可以被完全取代，但它現在對MODEL-DRIVEN DESIGN大有幫助，並且使UBIQUITOUS LANGUAGE更加豐富。

## 7.11 引入新特性：配額檢查

到目前為止，我們已經實現了最初的需求和模型。現在要添加第一批重要的新功能。

在這個假想的運輸公司中，銷售部門使用其他軟件來管理客戶關係、銷售計劃等。其中有一項功能是效益管理（yield management），利用此功能，公司可以根據貨物類型、出發地和目的地或者任何可作為分類名輸入的其他因素來制定不同類型貨物的運輸配額。這些配額構成了各類貨物的運輸量目標，這樣利潤較低的貨物就不會佔滿貨艙而導致無法運輸利潤較高的貨物，同時避免預訂量不足（沒有充分利用運輸能力）或過量預訂（導致頻繁地發生貨物碰撞，最終損害客戶關係）。

現在，他們希望把這個功能集成到預訂系統中。這樣，當客戶進行預訂時，可以根據這些配額來檢查是否應該接受預訂。

配額檢查所需的信息保存在兩個地方，Booking Application必須通過查詢這些信息才能確定接受或拒絕預訂。圖7-9給出了一個大體的信息流草圖。

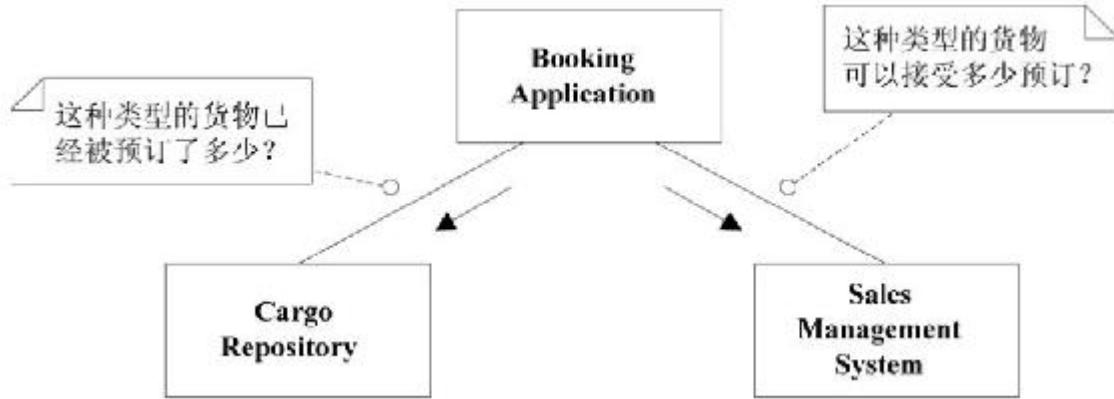


圖7-9 Booking Application所使用的信息一方面來自Sales Management System ( 銷售管理系統 ) , 一方面來自我們自己的領域REPOSITORY

### 7.11.1 連接兩個系統

銷售管理系統 ( Sales Management System ) 並不是根據這裡所使用的模型編寫的。如果Booking Application與它直接交互，那麼我們的應用程序就必須適應另一個系統的設計，這將很難保持一個清晰的MODEL-DRIVEN DESIGN，而且會混淆UBIQUITOUS LANGUAGE。相反，我們創建一個類，讓它充當我們的模型和銷售管理系統的語言之間的翻譯。它並不是一種通用的翻譯機制，而只是對我們的應用程序所需的特性進行翻譯，並根據我們的領域模型重新對這些特性進行抽象。這個類將作為一個ANTICORRUPTION LAYER ( 將在第14章討論 ) 。

這是連接銷售管理系統的一個接口，因此首先就會想到將它叫做 **Sales Management Interface** ( 銷售管理接口 ) 。但這樣一來就失去了用對我們更有價值的語言來重新描述問題的機會。相反，讓我們為每個需要從其他系統獲得的配額功能定義一個SERVICE。我們用一個名為 **Allocation Checker** ( 配額檢查器 ) 的類來實現這些SERVICE，這個類名反映了它在系統中的職責。

如果還需要進行其他集成 ( 例如，使用銷售管理系統的客戶數據庫，而不是我們自己的Customer REPOSITORY ) ，則可以創建另一個

翻譯類來實現用於履行該職責的SERVICE。用一個更低層的類（如 Sales Management System Interface）作為與其他程序進行對話的機制仍然是一種很有用的方法，但它並不負責翻譯。此外，它將隱藏在 Allocation Checker後面，因此領域設計中並不展示出來。

### **7.11.2 進一步完善模型：劃分業務**

我們已經大致描述了兩個系統的交互，那麼提供什麼樣的接口才能回答「這種類型的貨物可以接受多少預訂」這個問題呢？問題的複雜之處在於定義Cargo的「類型」是什麼，因為我們的領域模型尚未對Cargo進行分類。在銷售管理系統中，Cargo類型只是一組類別關鍵詞，我們的類型只需與該列表一致即可。我們可以把一個字符串集合作為參數傳入，但這樣會錯過另一個機會——重新抽像那個系統的領域。我們需要在領域模型中增加貨物類別的知識，以便使模型更豐富；而且需要與領域專家一起進行頭腦風暴活動，以便抽像出新的概念。

有時，分析模式可以為建模方案提供思路（第11章將會討論到）。《分析模式》[Fowler 1996]一書介紹了一種用於解決這類問題的模式：ENTERPRISE SEGMENT（企業部門單元）。ENTERPRISE SEGMENT是一組維度，它們定義了一種對業務進行劃分的方式。這些維度可能包括我們在運輸業務中已經提到的所有劃分方法，也包括時間維度，如月初至今（month to date）。在我們的配額模型中使用這個概念，可以增強模型的表達力，並簡化接口。這樣，我們的領域模型和設計中就增加了一個名為Enterprise Segment的類，它是一個VALUE OBJECT，每個Cargo都必須獲得一個Enterprise Segment類。

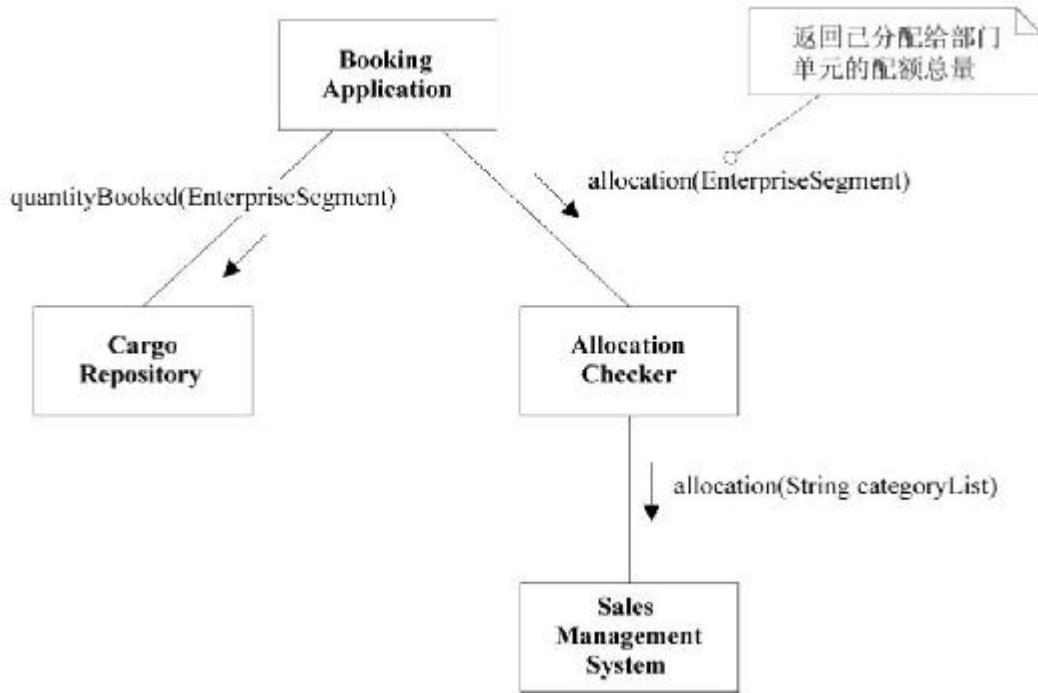


圖7-10 Allocation Checker充當了一個ANTICORRUPTION LAYER，它在我們的領域模型中展現了一個到銷售管理系統的可選接口

Allocation Checker將充當Enterprise Segment與外部系統的類別名稱之間的翻譯。Cargo Repository還必須提供一種基於Enterprise Segment的查詢。在這兩種情況下，我們可以利用與Enterprise Segment對像之間的協作來執行操作，而不會破壞Segment的封裝，也不會導致它們自身實現的複雜化（注意，Cargo Repository的查詢結果是一個數字，而不是實例的集合）。

但這種設計還存在幾個問題：

(1) 我們給Booking Application分配了一個不該由它來執行的工作，那就是對如下規則的應用：「如果Enterprise Segment的配額大於已預訂的數量與新Cargo數量的和，則接受該Cargo。」執行業務規則是領域層的職責，而不應在應用層中執行。

(2) 沒有清楚地表明Booking Application是如何得出Enterprise Segment的。

這兩個職責看起來都屬於Allocation Checker。通過修改接口就可以將這兩個服務分離出來，這樣交互就更整潔和明顯了。

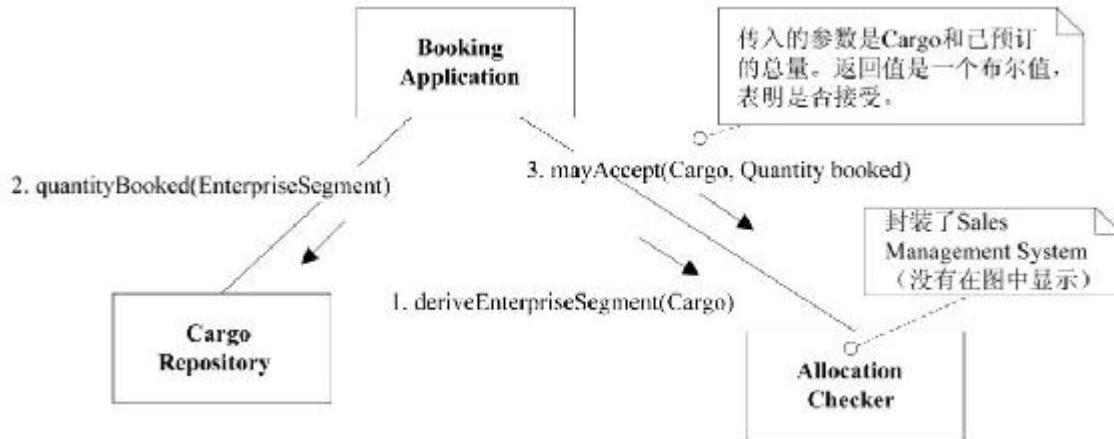


圖7-11 領域職責從Booking Application轉移到Allocation Checker

這種集成只有一條嚴格的約束，那就是有些維度是不能被Sales Management System使用的，具體來說就是那些無法用Allocation Checker轉換為Enterprise Segment的維度（在不使用ENTERPRISE SEGMENT的情況下，這條約束的作用是使銷售系統只能使用那些可以在Cargo Repository查詢中使用的維度。雖然這種方法也行得通，但銷售系統將會溢出而進入領域的其他部分中。在我們這個設計中，Cargo Repository只需處理Enterprise Segment，而且銷售系統中的更改隻影響到Allocation Checker，而Allocation Checker可以被看作是一個FAÇADE）。

### 7.11.3 性能優化

雖然與領域設計的其他方面有利害關係的只是Allocation Checker的接口，但當出現性能問題時，Allocation Checker的內部實現可能為解決這些問題提供機會。例如，如果Sales Management System運行在另一臺服務器上（或許在另一個位罷上），那麼通信開銷可能會很大，而且每個配額檢查都需要進行兩次消息交換。第二條消息需要調

用Sales Management System來回答是否應該接受貨物，因此並沒有其他的替代方法可用來處理這條消息。但第一條消息是得出貨物的Enterprise Segment，這條消息所基於的數據和行為與配額決策本身相比是靜態的。這樣，一種設計選擇就是緩存這些信息，以便Allocation Checker在需要的時候能夠在自己的服務器上找到它們，從而將消息傳遞的開銷降低一半。但這種靈活性也是有代價的。設計會更複雜，而且被緩存的數據必須保持最新。但如果性能在分佈式系統中是至關重要的因素的話，這種靈活部署可能成為一個重要的設計目標。

## 7.12 小結

情況就是這樣了。這種集成原本可能把我們這個簡單且在概念上一致的設計弄得亂七八糟，但現在，在使用了ANTICORRUPTION LAYER、SERVICE和ENTERPRISE SEGMENT之後，我們已經乾淨利落地把Sales Management System的功能集成到我們的預訂系統中了，從而使領域更加豐富。

還有最後一個設計問題：為什麼不把獲取Enterprise Segment的職責交給Cargo呢？如果Enterprise Segment的所有數據都是從Cargo中獲取的，那麼乍看上去把它變成Cargo的一個派生屬性是一種不錯的選擇。遺憾的是，事情並不是這麼簡單。為了用有利於業務策略的維度進行劃分，我們需要任意定義Enterprise Segment。出於不同的目的，可能需要對相同的ENTITY進行不同的劃分。出於預訂配額的目的，我們需要根據特定的Cargo進行劃分；但如果是出於稅務會計的目的時，可能會採取一種完全不同的Enterprise Segment劃分方式。甚至當執行新的銷售策略而對Sales Management System進行重新配

铬時，配額的Enterprise Segment劃分也可能會發生變化。如此，Cargo就必須瞭解Allocation Checker，而這完全不在其概念職責範圍之內。而且得出特定類型Enterprise Segment所需使用的方法會加重Cargo的負擔。因此，正確的做法是讓那些知道劃分規則的對象來承擔獲取這個值的職責，而不是把這個職責施加給包含具體數據（那些規則就作用於這些數據上）的對象。另一方面，這些規則可以被分離到一個獨立的Strategy對像中，然後將這個對象傳遞給Cargo，以便它能夠得出一個Enterprise Segment。這種解決方案似乎超出了這裡的需求，但它可能是之後設計的一個選擇，而且應該不會對設計造成很大的破壞。

---

[1].模型ENTITY與Java的'實體bean'並不是一回事。實體bean本打算成為一種用於實現ENTITY的框架，但它實際上並沒有做到。大多數ENTITY都被實現為普通對象。不管它們是如何實現的，ENTITY都是領域模型中的一個根本特徵。

[2].常飛乘客，frequent flier，是指經常乘飛機的人。——譯者注

[3].社會保險號碼，Social Security Number，由美國政府對其合法公民和居民頒發。主要用於報稅、申請駕照、申請賬戶等功能，是一種身份證明。——編者注

[4].WHOLE VALUE模式是由Ward Cunningham提出的。

[5].FAÇADE（外觀）是一種設計模式，它將一系列複雜的類包裝成一個簡單的封閉接口。——譯者注

[6].認知超載，cognitive overload，認知負荷理論（cognitive load theory）中的一個術語。問題解決和學習過程中的各種認知加工活動均需消耗認知資源，若所有活動所需的資源總量超過個體擁有的資源

總量，就會引起資源的分配不足，從而影響個體學習或問題解決的效率，這種情況被稱為認知超載。——譯者注

[7].David Siegel於20世紀90年代發明瞭這個系統，並在一些項目上使用了它，但並沒有公開發表。

## 第三部分 通過重構來加深理解

本書的第二部分為維護模型和實現之間的對應關係打下了基礎。在開發過程中使用一系列成熟的基本構造塊並運用一致的語言，能夠使開發工作更加清晰而有條理。

當然，我們面臨的真正挑戰是找到深層次的模型，這個模型不但能夠捕捉到領域專家的微妙的關注點，還可驅動切實可行的設計。我們的最終目的是開發出能夠捕捉到領域深層含義的模型。以這種方式設計出來的軟件不但更加貼近領域專家的思維方式，而且能更好地滿足用戶的需求。本部分將會對這個目標加以說明並詳細描述其實現過程，同時也會解釋某些設計原則和模式。我們應用這些原則和模式來得到滿足應用程序以及開發人員自身需求的設計。

要想成功地開發出實用的模型，需要注意以下3點。

(1) 複雜巧妙的領域模型是可以實現的，也是值得我們去花費力氣實現的。

(2) 這樣的模型離開不斷的重構是很難開發出來的，重構需要領域專家和熱愛學習領域知識的開發人員密切參與進來。

(3) 要實現並有效地運用模型，需要精通設計技巧。

### **重構的層次**

重構就是在不改變軟件功能的前提下重新設計它。開發人員無需在著手開發之前做出詳細的設計決策，只需要在開發過程中不斷小幅調整設計即可，這不但能夠保證軟件原有的功能不變，還可使整個設

計更加靈活易懂。自動化的單元測試套件能夠保證對代碼進行相對安全的試驗。這個過程解放了開發人員，使他們不再需要提前考慮將來的事情。

然而，幾乎所有關於重構的文獻都專注於如何機械地修改代碼，以使其更具可讀性或在非常細節的層次上有所改進。如果開發人員能夠看準時機，利用成熟的設計模式進行開發，那麼「通過重構得到模式」[\[1\]](#) (refactoring to patterns) 這種方式就可以讓重構過程更上一層樓。不過，這依然是從技術視角來評估設計的質量。

有些重構能夠極大地提高系統的可用性，它們要麼源於對領域的新認知，要麼能夠通過代碼清晰地表達出模型的含義。這些重構不能取代設計模式重構和代碼細節重構，這兩種重構應該持續進行。但前者添加了另一種重構層次：為實現更深層模型而進行的重構。在深入理解領域的基礎上進行重構，通常需要實現一系列的代碼細節重構，但這麼做絕不僅僅是為了改進代碼狀態。相反，代碼細節重構是一組操作方便的修改單元，通過這些重構可以得到更深層次的模型。其目標在於：開發人員通過重構不僅能夠瞭解代碼實現的功能，還能明白箇中原因，並把它們與領域專家的交流聯繫起來。

《重構》[\[Fowler 1999\]](#)一書中所列出的重構分類涵蓋了大部分常用的代碼細節重構。這些重構主要是為瞭解決一些可以從代碼中觀察到的問題。相比之下，領域模型會隨著新認識的出現而不斷變化，由於其變化如此多樣，以至於根本無法整理出一個完整的目錄。

與所有的探索活動一樣，建模本質上是非結構化的。要跟隨學習與深入思考所指引的一切道路，然後據此重構，才能得到更深層的理解。儘管已發佈的成功模型會對我們大有幫助（第11章將會提及），但是不能因此將領域建模簡化為照本宣科的行為，當其是秘籍類的書籍或者工具包，依樣畫葫蘆。建模和設計都需要你發揮創造力。接下

來的6章將會給出一些改進領域模型的具體思考方式以及可實現這些領域模型的設計方法。

## 深層模型

對像分析的傳統方法是先在需求文檔中確定名詞和動詞，並將其作為系統的初始對像和方法。這種方式太過簡單，只適用於教導初學者如何進行對像建模。事實上，初始模型通常都是基於對領域的淺顯認知而構建的，既不夠成熟也不夠深入。

例如，我曾參與過一個運輸應用系統的開發，我的初始想法是構建一個包括貨輪（*ship*）和集裝箱的對象模型。貨輪將貨物從一個地點運送到另一個地點。集裝箱則通過裝卸操作與貨輪建立關聯或解除關聯。這確實能夠準確描述一部分實際運輸活動。但事實證明，它對於運輸業務的軟件實現並沒有太多幫助。

最終，在與運輸專家一起工作了幾個月並進行了多次迭代後，我們得到了一個完全不同的模型。在外行人看來，它也許沒那麼淺顯易懂，但卻能貼切地反映出專家的想法。這個模型的關注點再次回到了運送貨物的業務。

我們依然保留了*ship*，但是將其抽像為「船隻航次」（*vessel voyage*），即貨輪、火車或其他運輸工具的某一調度好的航程。貨輪本身不再重要，如遇維修或計劃變動可臨時改用其他方式，只要保證原定航次按計劃執行即可。運輸集裝箱則完全從模型中移除了。它現在以一種完全不同的複雜形式出現在貨物裝卸應用程序中，而在原來的應用程序中，集裝箱變成了操作細節。貨物實際的位移變化已不重要，重要的是其法律責任的轉移。原來一些諸如「提貨單」之類不被關注的對象也出現在模型中。

每當有新的對象建模人員加入這個項目時，他們首先提出的建議是什麼？就是添加「貨輪」和「集裝箱」這兩個缺少的類。他們都很

聰明，只不過還沒有仔細揣摩運輸領域的知識罷了。

深層模型能夠穿過領域表象，清楚地表達出領域專家們的主要關注點以及最相關的知識。以上定義並沒有涉及抽象。事實上，深層模型通常含有抽象元素，但在切中問題核心的關鍵位罷也同樣會出現具體元素。

恰當反映領域的模型通常都具有功能多樣、簡單易用和解釋力強的特性。這種模型的共同之處在於：它們提供了一種業務專家青睞的簡單語言，儘管這種語言可能也是抽象的。

### **深層模型/柔性設計**

在不斷重構的過程中，設計本身也需要支持重構所帶來的變化。第10章將會探討如何使設計更易於使用，不但方便修改還能夠簡單地將其與系統其他部分集成。

設計自身的某些特性就可以使其更易於修改和使用。這些特性並不複雜，卻很有挑戰性。第10章將主要討論「柔性設計」（*supple design*）及其實現方法。

幸運的是，如果每次對模型和代碼所進行的修改都能反映出對領域的新理解，那麼通過不斷的重構就能給系統最需要修改的地方增添靈活性，並找到簡單快捷的方式來實現普通的功能。戴久了的手套在手指關節處會變得柔軟；而其他部分則依然硬實，可起到保護的作用。同樣道理，用這種方式來進行建模和設計時，雖然需要反覆嘗試、不斷改正錯誤，但是對模型和設計的修改卻因此而更容易實現，同時反覆的修改也能讓我們越來越接近柔性設計。

柔性設計除了便於修改，還有助於改進模型本身。**MODEL-DRIVEN DESIGN**需要以下兩個方面的支持：深層模型使設計更具表現力；同時，當設計的靈活性可以讓開發人員進行試驗，而設計又能清晰地表達出領域含義時，那麼這個設計實際上就能夠將開發人員的深

層理解反饋到整個模型發現的過程中。這段反饋迴路是很重要的，因為我們所尋求的模型並不僅僅只是一套好想法：它還應該是構建系統的基礎。

## 發現過程

要想創建出確實能夠解決當前問題的設計，首先必須擁有可捕捉到領域核心概念的模型。第9章將會介紹如何主動搜尋這些概念，並將它們融入到設計中。

由於模型和設計之間具有緊密的關係，因此如果代碼難於重構，建模過程也會停滯不前。第10章將會探討如何為軟件開發者（尤其是為你自己）編寫軟件，以使開發人員能夠高效地擴展和修改代碼。這一設計過程與模型的進一步精化是密不可分的。它通常需要更高級的設計技巧以及更嚴格的模型定義。

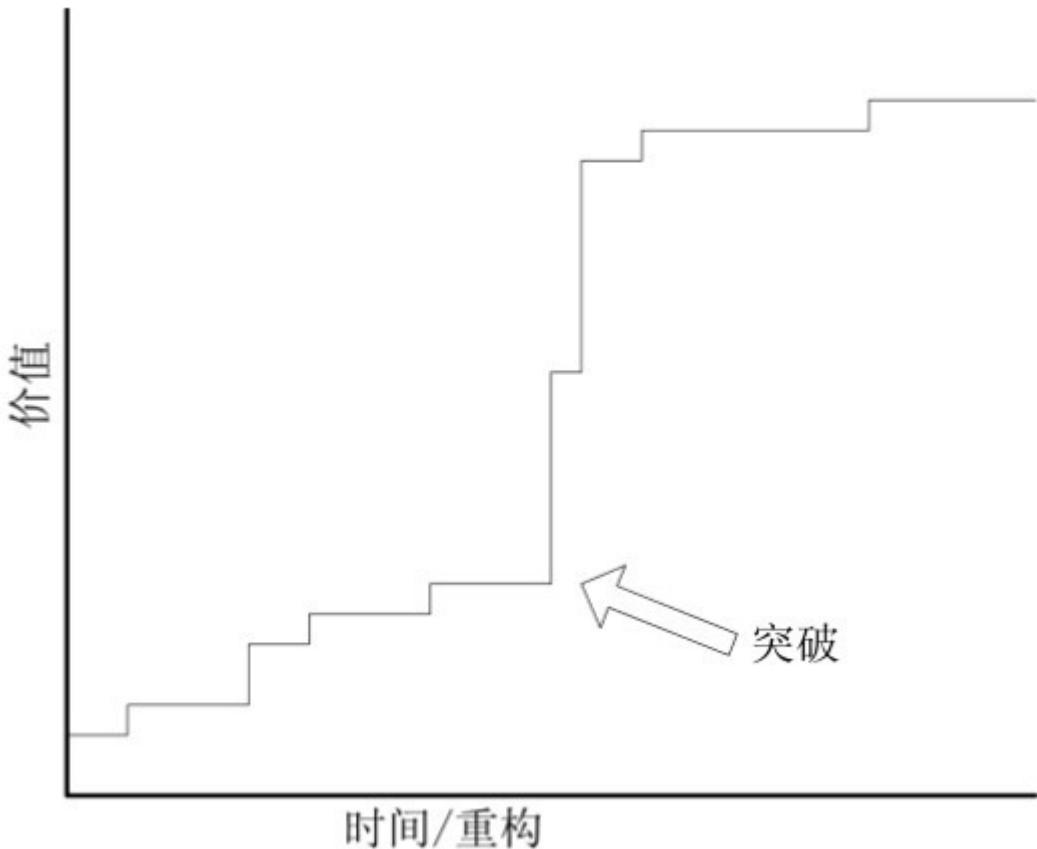
你需要富有創造力，不斷地嘗試，不斷地發現問題才能找到合適的方法為你所發現的領域概念建模，但有時你也可以借用別人已建好的模式。第11章和第12章將會討論「分析模式」和「設計模式」的應用。這些模式並不是現成的解決方案，但是它們可以幫助我們消化領域知識並縮小研究範圍。

但是，讓我們以領域驅動設計中最令人興奮的事件來開始第三部分吧，那就是突破。有時，當我們擁有了MODEL-DRIVEN DESIGN和顯式概念，就能夠產生突破。我們有機會使軟件更富表達力、更加多樣化，甚至會使它變得超乎我們的想像。這可以為軟件帶來新特性，或者意味著我們可以用簡單靈活的方式來表達更深層次的模型，從而替換掉大段死板的代碼。儘管這種突破不會時常出現，但它們非常有價值，當我們有機會進行突破時，一定要懂得識別並抓住機會。

第8章講述了一個真實的項目，這個項目通過重構過程得到了更深層的理解，最終實現了突破。這種經歷是可遇而不可求的。儘管如

此，它卻為我們提供了一個很好的學習背景，幫助我們思考領域重構。

## 第8章 突破



重構的投入與回報並非呈線性關係。通常，小的調整會帶來小的回報，小的改進也會積少成多。小改進可防止系統退化，成為避免模型變得陳腐的第一道防線。但是，有些最重要的理解也會突然出現，給整個項目帶來巨大的衝擊。

可以確定的是，項目團隊會積累、消化知識，並將其轉化成模型。微小的重構可能每次只涉及一個對象，在這裡加上一個關聯，在

那裡轉移一項職責。然而，一系列微小的重構會逐漸匯聚成深層模型。

一般來說，持續重構讓事物逐步變得有序。代碼和模型的每一次精化都讓開發人員有了更加清晰的認識。這使得理解上的突破成為可能。之後，一系列快速的改變得到了更符合用戶需要並更加切合實際的模型。其功能性及說明性急速增強，而複雜性卻隨之消失。

這種突破不是某種技巧，而是一個事件。它的困難之處在於你需要判斷發生了什麼，然後再決定如何處理。為了說明這是種什麼樣的經歷，我將會講述一個我幾年前參與過的真實項目，以及我們是如何獲得一個寶貴的深層模型的。

## 8.1 一個關於突破的故事

這個故事發生在紐約，經過一個冬天的漫長重構之後，我們最終得到了能夠捕捉到一些領域關鍵知識的模型和一個確實能在應用程序中發揮作用的設計。當時我們正在給一家投資銀行開發一個大型應用程序的核心部分，該程序用於管理銀團貸款。

假如Intel想要建造一座價值10億美元的工廠，就需要申請貸款，但貸款的額度太大，以至於任何一家借貸公司 (lending company) 都無法獨立承擔，於是這些公司就組成銀團[2]，集中它們的資源，以此來支持這種巨額信貸。投資銀行通常在銀團裡擔當領導者的角色，負責協調各種交易和其他服務。我們的項目就是要開發這樣一個用於跟蹤和支持以上整個過程的軟件。

### 8.1.1 華而不實的模型

當時，我們對於自己的成果感覺相當不錯。4個月前，我們還身陷困境，因為之前留下的代碼完全不可行，從那時開始我們就竭盡所

能將其遷移到一個一致的**MODEL-DRIVEN DESIGN**中。

圖 8-1 中展示的模型對常見業務進行了大幅簡化。Loan Investment ( 貸款投資 ) 是一個派生對象，用來表示某一投資者在 Loan ( 貸款 ) 中所承擔的股份，它與投資者在 Facility ( 信貸 ) 中所持有的股份成正比。

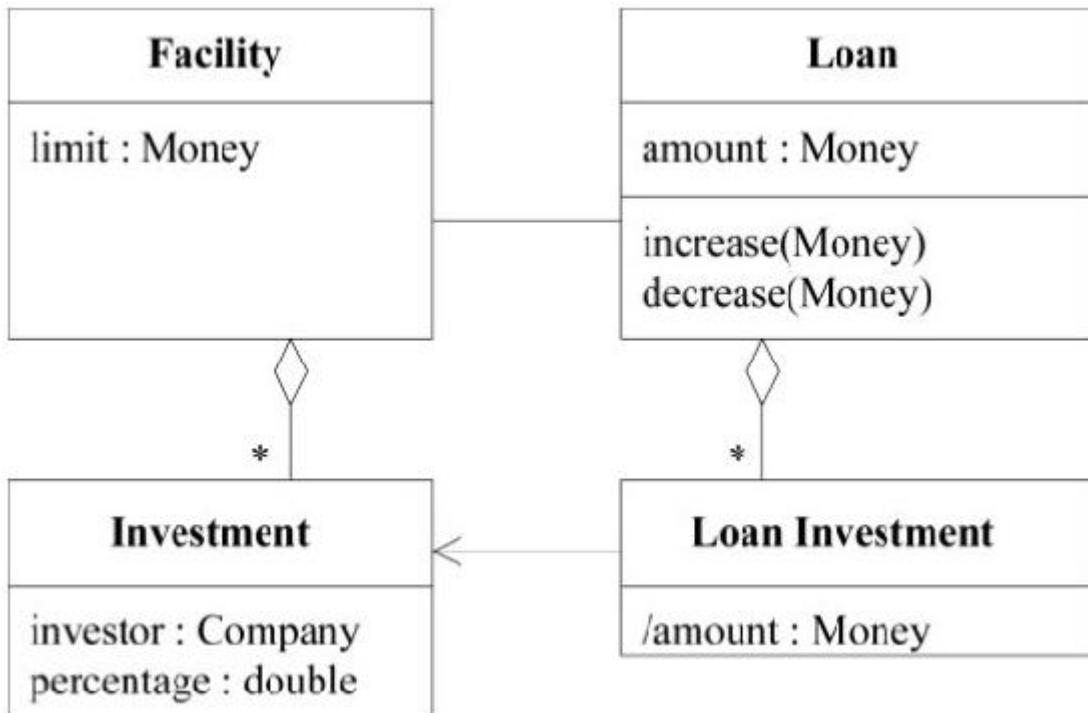


圖8-1 假定放貸方股份固定的模型

### Facility是什麼？

Facility ( 信貸 ) 在這裡並不是建築物的意思。在大部分項目中，領域專家提供的專用術語都會變成我們自己的詞彙，並成為 **UBIQUITOUS LANGUAGE** 的一部分。在商業銀行領域中，信貸是公司為借款而作出的承諾。信用卡就是一種信貸，卡片持有者有權在需要時借出不超過預設限制的金額，並且以預定利息還款。

當你使用信用卡時，就會產生一筆未償貸款，每筆支出都會降低你的信貸額度，並增加你的貸款金額。最後，你需要償還貸款本金。也許還需要繳納年費。年費是持有信用卡（信貸）所需繳納的費用，與你的貸款無關。

但是這個模型已顯露出了一些令人擔憂的跡象。各種意料之外的需求一直困擾著我們，也使設計更加複雜。最突出的例子就是對 Facility 股份的理解不斷深入，在提取（Drawdown）貸款時，信貸股份僅僅是放貸方投入金額的指導原則。當借款者（borrower）要求提取貸款時，銀團領導者會通知所有成員按各自的股份進行支付。

收到通知後，投資者通常會按自己的股份來支付，但是有時他們也會與銀團其他成員協商，以求少投入（或多投入）一些。於是我們在模型中添加了 Loan Adjustment（貸款調整）以反映這一事實。

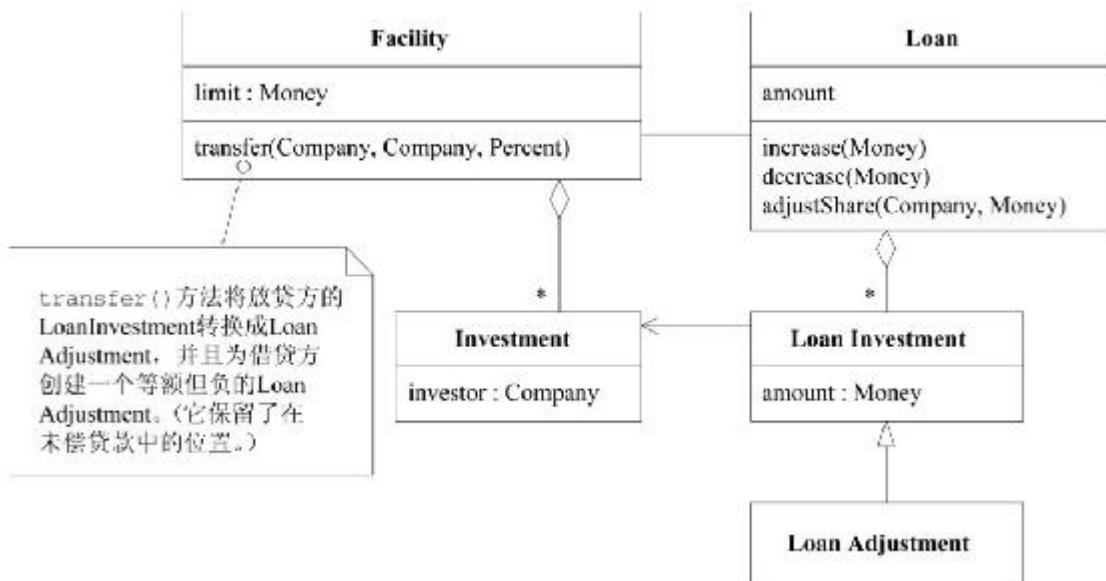


圖8-2 為解決問題而逐步修改的模型。Loan Adjustment用來跟蹤放貸方最初同意放貸的股份與實際放貸額之差

這種類型的精化使我們能夠越來越清楚地理解各種交易規則。但同時，模型的複雜度也在不斷增加，並且看起來我們無法很快從模型

中提煉出真正健壯的功能。

更麻煩的是儘管算法越來越複雜，我們卻無法解決捨入運算所帶來的細微差別。在1億美元的交易中，確實沒有人會在意幾美分的去向，但是銀行家是不會信任無法精確計算到美分的軟件的。我們開始懷疑我們所遇到的難題可能是因為基本設計存在問題。

### 8.1.2 突破

在項目進行過程中，我們突然領悟到了問題的所在。我們在模型中把**Facility**的股份和**Loan**的股份綁定到一起，而這種方式並不適用於實際的業務。這一發現得到了廣泛的認可。業務專家點頭稱是，並開始熱情地給予幫助（我敢說他們一定還在奇怪我們怎麼花了這麼長時間才明白這一點），我們迅速在白板上創建了一個新模型。雖然細節問題尚未確定，但是我們已經知道新模型的關鍵特徵了：**Loan**的股份和**Facility**的股份可以在互不影響的前提下獨立發生改變。有了這層認知，我們利用與下圖類似的模型圖走查了許多場景：

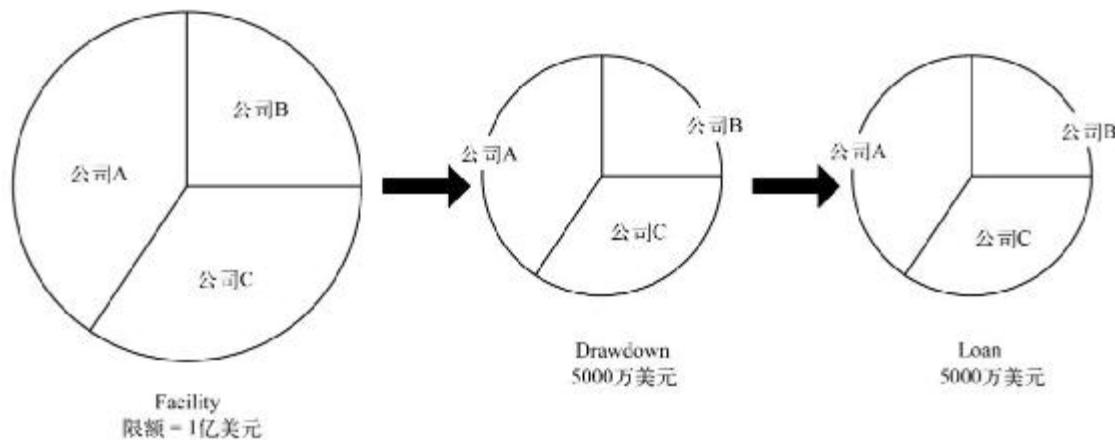


圖8-3 基於Facility的股份來進行分配的提取

這張圖表明**Facility**的總額是1億美元，而借款者選擇從中提取的第一筆**Loan**金額是5000萬美元。3個放貸方按照各自原先承諾的**Facility**的股份來支付，這樣5000萬的**Loan**就被分配到了這3個放貸方頭上。

隨後，借款者又提取了另一筆3000萬美元的貨款（如圖8-4所示），這樣他的未償Loan就達到了8000萬美元，依然在Facility的1億美元限額之內。這次，公司B決定不參與Loan，而由公司A來承擔這部分額外的股份。各個放貸方在借款者提取貸款的過程中所支付的股份反映了它們的投資選擇。當Loan的提取額不斷增加時，Loan的股份份額就不再與Facility的股份成比例了。這種現象很普遍。

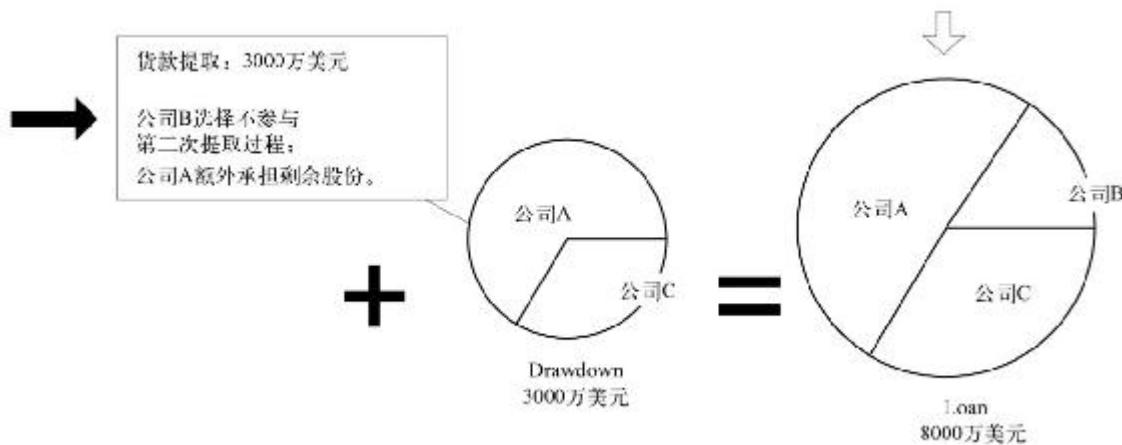


圖8-4 貸方B選擇不參與第二次提款

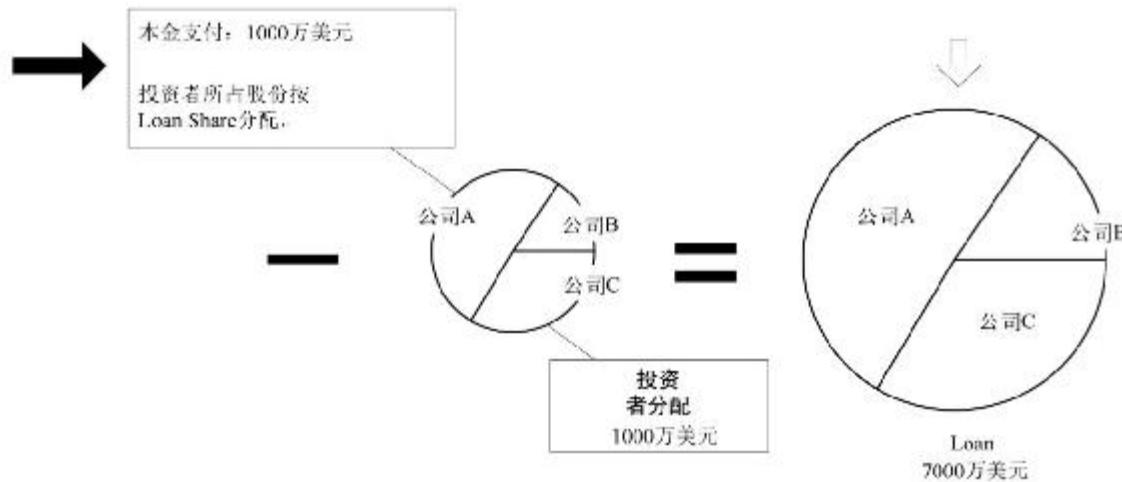


圖8-5 本金支付始終按照未償Loan的股份比例來進行分配

當借款者償還Loan時，所償還的金額會根據Loan的股份分配給各放貸方，而不是根據Facility的股份來劃分。同樣，利息支付也會按照

Loan的股份進行分配。

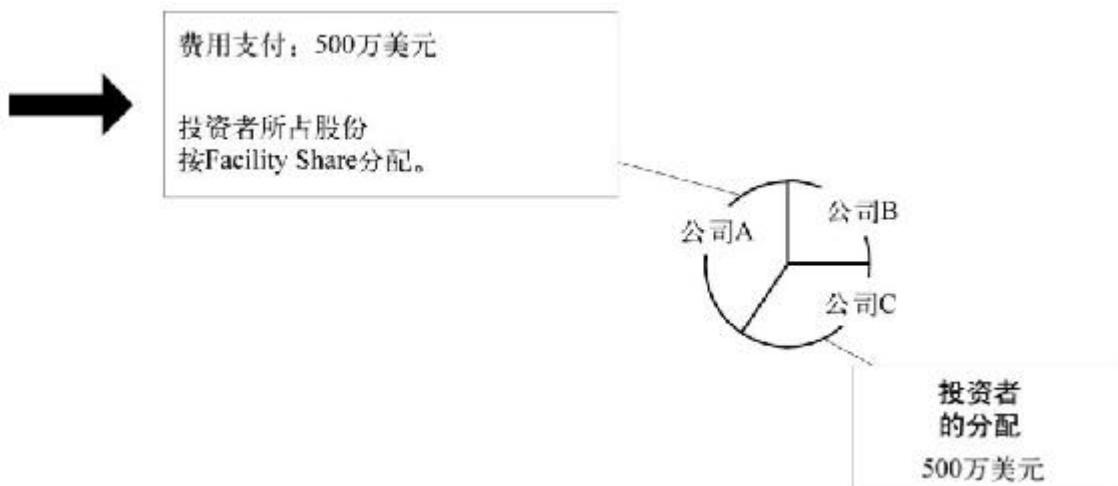


圖8-6 費用支付始終按照Facility的股份比例來進行分配

另一方面，當借款者為享有Facility權而支付費用時，這筆錢是按照Facility的股份劃分的，而不考慮放貸方是否借出了錢。Loan不會因費用支付而發生變化。甚至還有這種情況，放貸方單獨交易費用股份，與利息股份等無關。

### 8.1.3 更深層模型

我們得到了兩個深層理解。其一是意識到「投資」和「Loan投資」是「股份」這個常規基礎概念的兩種特例。信貸股份、貨款股份、支付比例股份，這些都是股份，股份無處不在。任何可分配的價值都是股份。

經過幾天忙碌的工作，我根據與專家討論時所使用的語言以及我們一起研究的場景，初步搭建起了一個股份模型，如圖8-7所示。

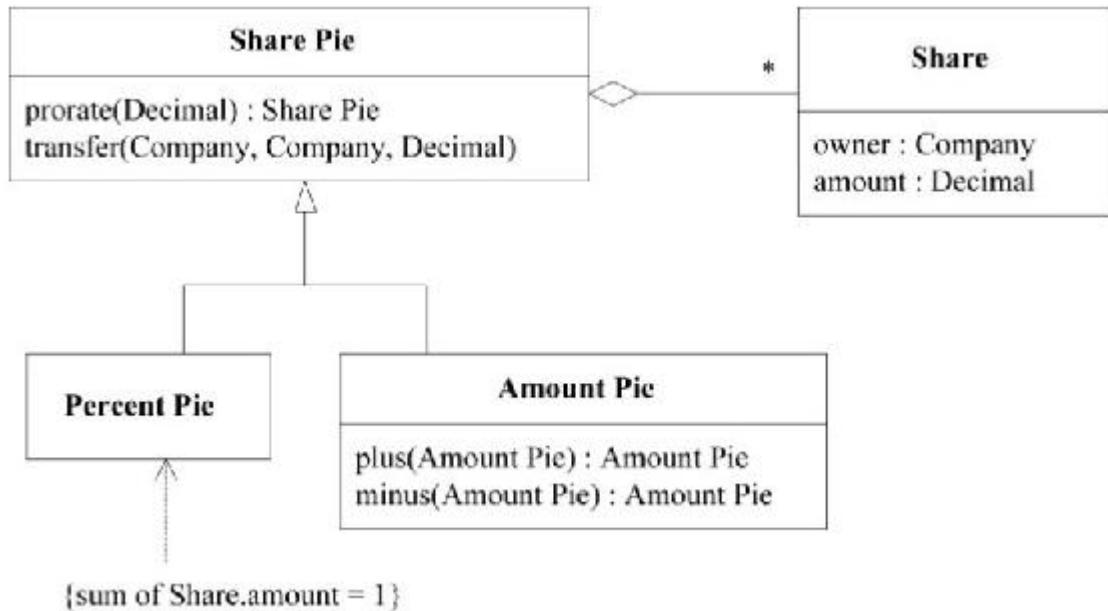


圖8-7 股份的抽象模型

我同時也草繪了一個與股份模型搭配的新貸款模型。

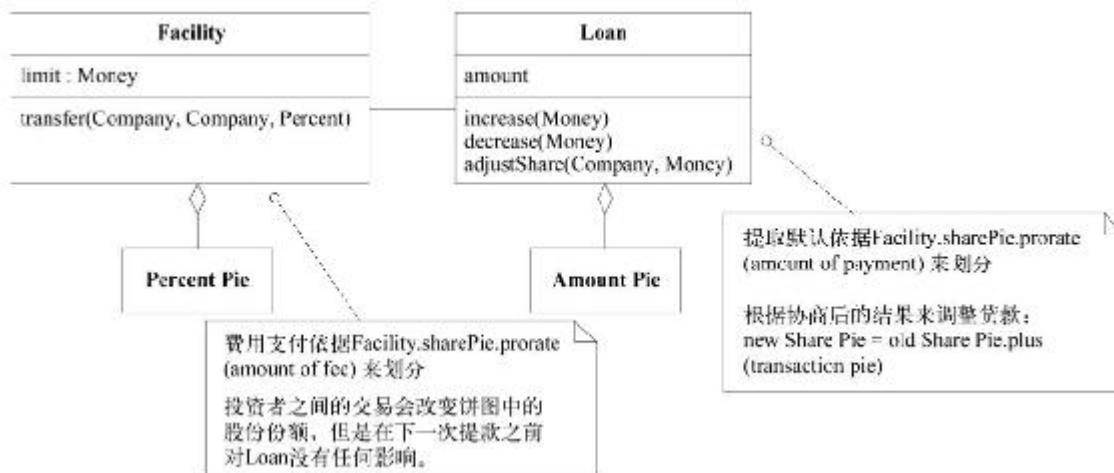


圖8-8 使用Share Pie的Loan模型

現在**Facility**股份和**Loan**股份不再由專用對像來表示了。它們都被分解成了更直觀的**Share Pie**。這種泛化引入了「股份數學」的概念，極大地簡化了所有交易中的股份計算，同時也使這些計算更富有表達力、更簡潔且更易於組合。

但最重要的是，新模型刪除了不恰當的約束，這使得我們的問題迎刃而解。Loan的Share因而無需與Facility 的Share成比例，同時新模型仍然保留著對總額、費用分配等的有效約束。我們可以直接調整Loan的Share Pie，因此新模型也不再需要Loan Adjustment和大量處理特殊情況的邏輯了。

新模型中不再包含Loan Investment對象，我們此時才意識到「貸款投資」並不是一個銀行業術語。事實上，業務專家早已多次告訴我們，他們不明白「貸款投資」是什麼意思。但是他們還是尊重我們在軟件方面的知識，並假定它對技術方面的設計是有所幫助的。而事實上，我們之所以創建它是因為沒有完全理解領域。

這種看待領域的新方法使我們立即就可以毫不費力地處理之前遇到的所有場景，處理過程要比以往簡單許多。業務專家認為我們的模型圖非常合理，他們曾經指出我們之前的模型圖對他們來說「技術性太強」了。即使只是在白板上畫出草圖，我們也能看出新模型能夠徹底解決長期困擾我們的捨入計算問題，我們可以不再使用複雜的捨入代碼了。

新模型的效果很好。非常非常好。

而我們都已疲憊不堪了！

#### **8.1.4 冷靜決策**

你也許會認為我們在那時一定會洋洋自得。但我們沒有。項目的期限很緊，而我們的進度已嚴重落後了。所以，那時我們最強烈的感受就是擔憂。

重構的原則是始終小步前進，始終保持系統正常運轉。但是要按照這個新模型來重構則需要修改大量的支持代碼，在重構的過程中，系統幾乎無法正常運轉。我們能夠看出一些力所能及的微小改進，但這些改進無法讓我們實現新的領域概念。我們也知道通過一系列小改

動可以實現新模型，但是在這個過程中必然會導致程序的一部分功能無法正常工作。而且在當時，自動化測試還沒有廣泛應用於這種項目。我們一無所有，所以肯定會出現一些讓我們始料不及的破壞。

此外，重構是需要花費精力去實現的。但幾個月以來，我們一直壓力重重，早已精疲力盡了。

這時，我們與項目經理開了一次會，這次會議令我終生難忘。我們的項目經理是個睿智而勇敢的人。他問了我們許多問題：

Q1：如果採用新設計，需要多久才能重新實現已有功能？

A1：大約3周。

Q2：不用新設計可以解決問題嗎？

A2：有可能。但我們無法保證。

Q3：如果現在不採用新設計，可以繼續進行下一個版本的開發嗎？

A3：如果不做修改，開發進度會非常緩慢。而且我們的系統一旦有了客戶群，再做修改就會變得更加困難。

Q4：這是正確的行動？

A4：我們知道目前的局面很不穩定，如果不是非做不可，我們也可以將就。而且我們都很疲憊了。但是，是的，這是個更加簡單的解決方案，也更符合業務需求。從長遠角度來看，它會降低風險。

他給我們開了綠燈，並告訴我們他會控制好局面。做出這種決定需要無比的勇氣和信心，這使我一直對他欽佩不已。

我們全力以赴，在3個星期內完成了任務。這是個巨大的工程，但是卻進展得異常順利。

### **8.1.5 成果**

項目需求不再有意外的、難以捉摸的改變了。捨入邏輯的實現雖然並不簡單，卻很穩定並且合理。我們交付了軟件的第一個版本，第

二個版本的開發思路也很清晰了。我的神經衰弱也多少有了好轉。

在進行第二個版本的開發時，Share Pie成了整個程序的統一主題。技術人員和業務專家利用它來對系統進行討論。市場人員使用它來向預期客戶解釋系統特性。這些預期客戶和其他的客戶都能立刻理解它，並且可以馬上用它來討論特性。由於它抓住了銀團貸款的核心問題，所以它真正成為了**UBIQUITOUS LANGUAGE**的一部分。

## 8.2 機遇

當突破帶來更深層的模型時，通常會令人感到不安。與大部分重構相比，這種變化的回報更多，風險也更高。而且突破出現的時機可能很不合時宜。

儘管我們希望進展順利，但往往事與願違。過渡到真正的深層模型需要從根本上調整思路，並且對設計做大幅修改。在很多項目中，建模和設計工作最重要的進展都來自於突破。

## 8.3 關注根本

不要試圖去製造突破，那隻會使項目陷入困境。通常，只有在實現了許多適度的重構後才有可能出現突破。在大部分時間裡，我們都在進行微小的改進，而在這種連續的改進中模型深層含義也會逐漸顯現。

要為突破做好準備，應專注於知識消化過程，同時也要逐漸建立健壯的**UBIQUITOUS LANGUAGE**。尋找那些重要的領域概念，並在模型中清晰地表達出來（參見第9章）。精化模型，使其更具柔性（參見第10章）。提煉模型（參見第15章）。利用這些更容易掌握的手段使模型變得更清晰，這通常會帶來突破。

不要猶豫著不去做小的改進，這些改進即使脫離不開常規的概念框架，也可以逐漸加深我們對模型理解。不要因為好高騖遠而使項目陷入困境。只要隨時注意可能出現的機會就夠了。

## 8.4 後記：越來越多的新理解

突破使我們走出了困境，但故事並沒有就此結束。更深層次的模型為我們帶來了意想不到的機會，它使應用程序的功能更加豐富，設計也更加清晰。

在Share Pie版本的程序發佈幾周之後，我們注意到在模型中還存在一個使設計變得複雜的地方。我們漏掉了一個重要的ENTITY，結果是本來應該由它承擔的職責不得不由其他對像來完成。具體來說就是提取貨款、繳納費用等業務是由一些重要的規則控制的，而所有這些邏輯都分散在Facility和Loan中的各種方法裡了。這些設計問題在Share Pie突破出現之前幾乎沒有引起我們的注意，但是隨著我們對領域的理解日漸清晰，它們也變得明顯起來。現在我們開始注意到，那些經常在討論中出現的術語（如「交易」，代表一次金融交易）並沒有體現在模型中，反而隱含在了那些複雜的方法裡。

經過與之前類似的過程（但所幸的是，我們有了更加充足的時間），我們對領域的理解又向前邁進了一步，並獲得了一個更深層次的模型。這個新模型不但使所有隱含的概念顯現了出來，如Transaction，以及一個簡化的Position（包括Facility和Loan的抽象類）。於是定義各種交易、交易規則、協商程序和審批流程就變得輕而易舉了，而且實現這些概念的代碼也相對來說變得更好理解了。

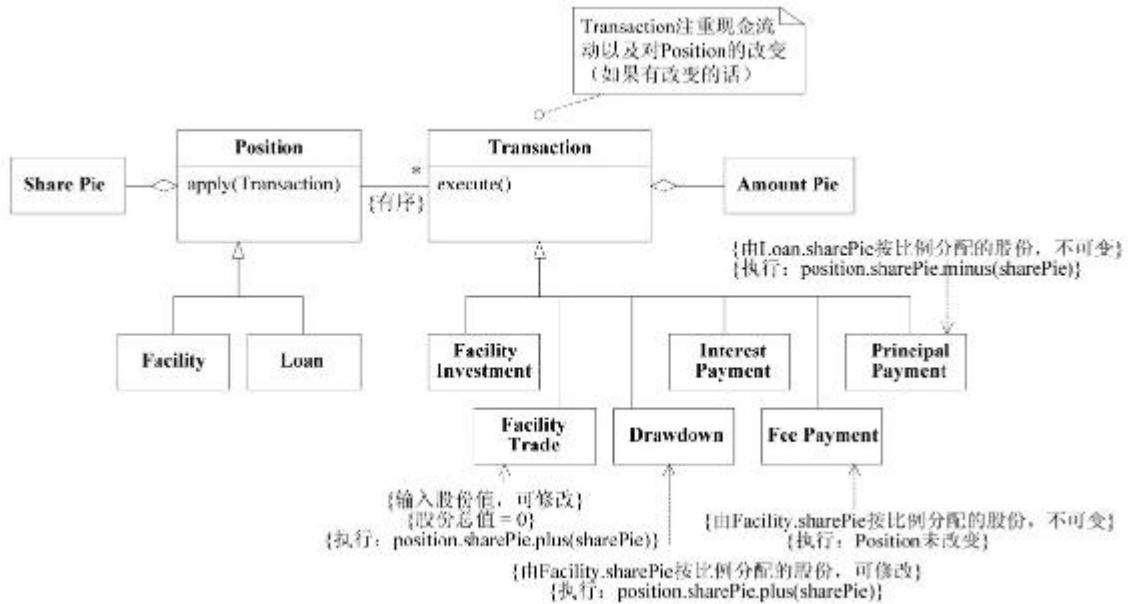


圖8-9 幾周之後的又一次模型突破。Transaction的約束可以被簡單精確地表達出來

通常，在經過一次真正的突破並獲得了深層模型之後，所獲得的新設計變得更加清晰簡單，新的**UBIQUITOUS LANGUAGE**也會增進溝通，於是又促成了下一次建模突破。

當大部分項目由於規模和複雜度的累積而舉步維艱時，我們的項目卻正在加速前進。

# 第9章 將隱式概念轉變為顯式概念

深層建模聽起來很不錯，但是我們要如何實現它呢？深層模型之所以強大是因為它包含了領域的核心概念和抽象，能夠以簡單靈活的方式表達出基本的用戶活動、問題以及解決方案。深層建模的第一步就是要設法在模型中表達出領域的基本概念。隨後，在不斷消化知識和重構的過程中，實現模型的精化。但是實際上這個過程是從我們識別出某個重要概念並且在模型和設計中把它顯式地表達出來的那個時刻開始的。

若開發人員識別出設計中隱含的某個概念或是在討論中受到啟發而發現一個概念時，就會對領域模型和相應的代碼進行許多轉換，在模型中加入一個或多個對象或關係，從而將此概念顯式地表達出來。

有時，這種從隱式概念到顯式概念的轉換可能是一次突破，使我們得到一個深層模型。但更多的時候，突破不會馬上到來，而需要我們在模型中顯式表達出許多重要概念，並通過一系列重構不斷地調整對象職責、改變它們與其他對象的關係、甚至多次修改對象名稱，在這之後，突破才會姍姍而來。最後，所有事情都變得清晰了。但是要實現上述過程，必須首先識別出以某種形式存在的隱含概念，無論這些概念有多麼原始。

## 9.1 概念挖掘

開發人員必須能夠敏銳地捕捉到隱含概念的蛛絲馬跡，但有時他們必須主動尋找線索。要挖掘出大部分的隱含概念，需要開發人員去

傾聽團隊語言、仔細檢查設計中的不足之處以及與專家觀點相矛盾的地方、研究領域相關文獻並且進行大量的實驗。

### **9.1.1 傾聽語言**

你可能會想起這樣的經歷：用戶總是不停地談論報告中的某一项。該項可能來自各種對象的參數彙編，甚至還可能來自一次直接的數據庫查詢。同時，應用程序的另一部分也需要這個數據集來進行顯示、報告或其他操作。但是，你卻一直認為沒有必要為此創建一個對象。也許你一直沒有真正理解用戶想通過某個特定術語傳達的東西，也沒有意識到它的重要性。

然後，你突然靈機一動。原來，報告中該項名稱給出了一個重要的領域概念。你高興地與專家談起了這個新發現。他們可能會鬆一口氣，因為你終於明白了。也可能會覺得很平常，因為他們一直認為這是理所當然的。不管專家們如何反應，你開始在白板上畫模型圖了（之前你也一直這麼做）。用戶會幫助你修正新模型連接方面的細節，但你明顯感到討論的質量有所提高。你和用戶可以更加準確地理解對方，並且可以更加自然地用模型交互來演示特定場景。領域模型的語言也變得更加強大。然後，你可以重構代碼來反映新模型，同時也會發現你的設計變得更加清晰了。

**傾聽領域專家使用的語言。有沒有一些術語能夠簡潔地表達出複雜的概念？他們有沒有糾正過你的用詞（也許是很委婉的提醒）？當你使用某個特定詞語時，他們臉上是否已經不再流露出迷惑的表情？這些都暗示了某個概念也許可以改進模型。**

這不同於原來的「名詞即對像」概念。聽到新單詞只是個開頭，然後我們還要進行對話、消化知識，這樣才能挖掘出清晰實用的概念。如果用戶或領域專家使用了設計中沒有的詞彙，這就是個警告信

號。而當開發人員和領域專家都在使用設計中沒有的詞彙時，那就是一個倍加嚴重的警告信號了。

或者，應該把這種警告看成一次機會。**UBIQUITOUS LANGUAGE**是由遍佈於對話、文檔、模型圖甚至代碼中的詞彙構成的。如果出現了設計中沒有的術語，就可以把它添加到通用語言中，這樣也就有機會改進模型和設計了。

### 示例 聽出運輸模型中缺失的一個概念

團隊已經開發出了可用來預訂貨物的有效應用程序。現在他們開始開發「作業支持」應用程序，此程序可幫助工作人員管理工作單，這些工作單用於安排起始地和目的地的貨物裝卸以及在不同貨輪之間轉運時需要的貨物裝卸。

預定應用程序使用一個路線引擎來安排貨物行程。運輸過程的每段行程都作為一行數據存儲在數據庫表中，其中指定了裝載該貨物的船名航次（某一貨輪的某一航次）ID、裝貨地點以及卸貨地點，如圖9-1所示。

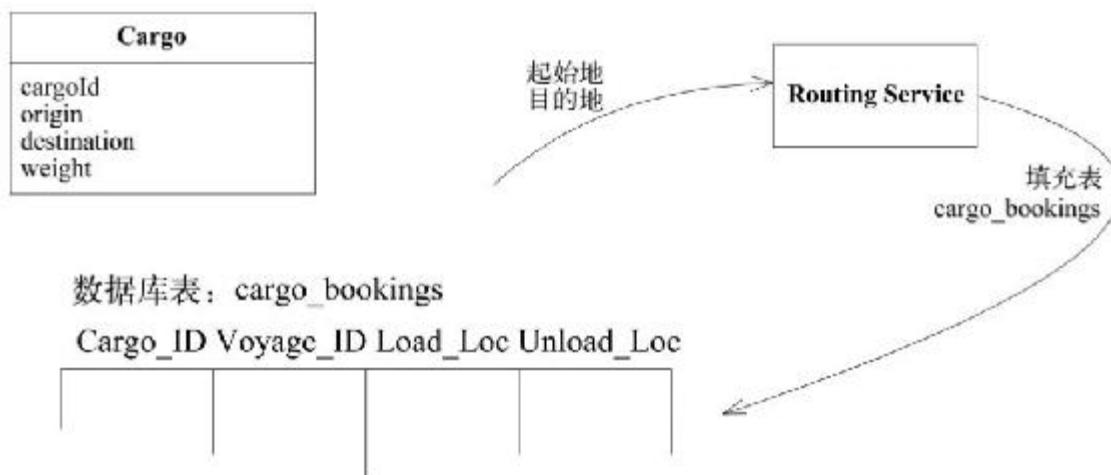


圖9-1

讓我們來聽聽開發人員和運輸專家之間的對話吧（對話已被高度簡化）。

開發人員：我想要確認一下cargo bookings（貨物預訂）表中是否已包含了作業應用程序所需的全部數據。

專家：他們需要Cargo的全部航海日程（Itinerary）。現在表中有哪些信息？

開發人員：貨物ID、船名航次以及每個航段的裝貨港口和卸貨港口。

專家：那麼日期呢？需要按照預計的時間來進行裝卸工作。

開發人員：嗯，日期可以從船名航次安排中獲得。該表的數據已經得到了規範化處理。

專家：是的，日期通常都是必需的數據。作業人員會用這類航海日程來安排後面的裝卸工作。

開發人員：嗯……好的。他們肯定可以得到日期數據。作業管理應用程序可以提供全部裝貨和卸貨信息以及每次裝卸作業的日期。我猜這也就是你所說的「航海日程」。

專家：很好。航海日程是他們需要的主要數據。事實上，你知道，預訂應用程序包含了一個菜單項，可以打印出航海日程或將航海日程通過電子郵件發送給顧客。你能想辦法利用這個功能嗎？

開發人員：我想那只是個報表。我們無法據此來開發作業應用程序。

[開發人員陷入了沉思，然後開始興奮起來。]

開發人員：那麼，航海日程實際上把預訂程序和作業程序連接起來了。

專家：是的。它同時還連接了一些客戶關係。

開發人員：[在白板上畫出了一個草圖。]那麼，你覺得是這樣的嗎？

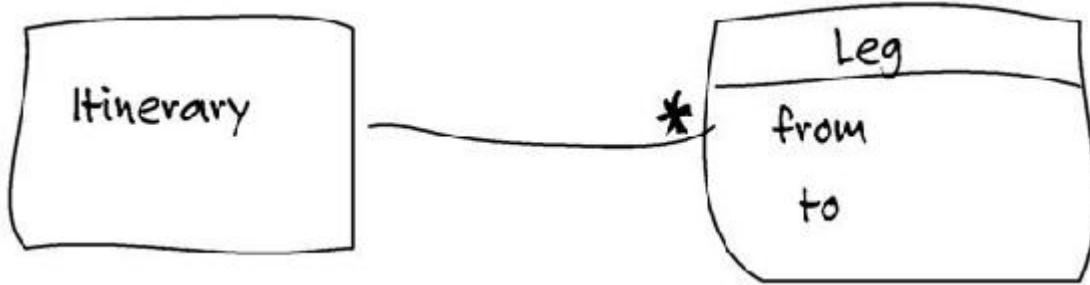


圖9-2

專家：是的，基本上是這樣。在每段行程中，我們都希望看到船名航次、裝貨和卸貨地點以及時間。

開發人員：所以，我們一旦創建了Leg（般段）對象，就能夠從航次安排中獲取時間信息。我們可以將Itinerary（航海日程）對像作為與作業應用程序聯繫的主要連接點。同時，還可以用這種方式重新編寫航海日程報表，這樣領域邏輯就重新回到領域層中了。

專家：有些地方我不太明白，但是你說對了Itinerary的兩個主要用途，一是用在預訂應用程序報表功能中，二是用在作業應用程序中。

開發人員：嘿！我們可以讓Routing Service（路線服務）接口返回航程對象，而不用將數據寫入數據庫表。這樣一來，路線引擎就不需要知道數據庫表了。

專家：嗯？

開發人員：我是說，我可以讓路線引擎只返回一個Itinerary。然後，預訂應用程序在保存剩下的信息時把它一起存儲到數據庫中。

專家：你是說現在的程序並沒有這麼做嗎？！

這位開發人員回去與負責路線處理的人員進行討論。他們仔細研究了這會給模型和設計帶來什麼影響和變化，在必要的時候也去請教了運輸專家。最後，他們得到了圖9-3所示的模型。

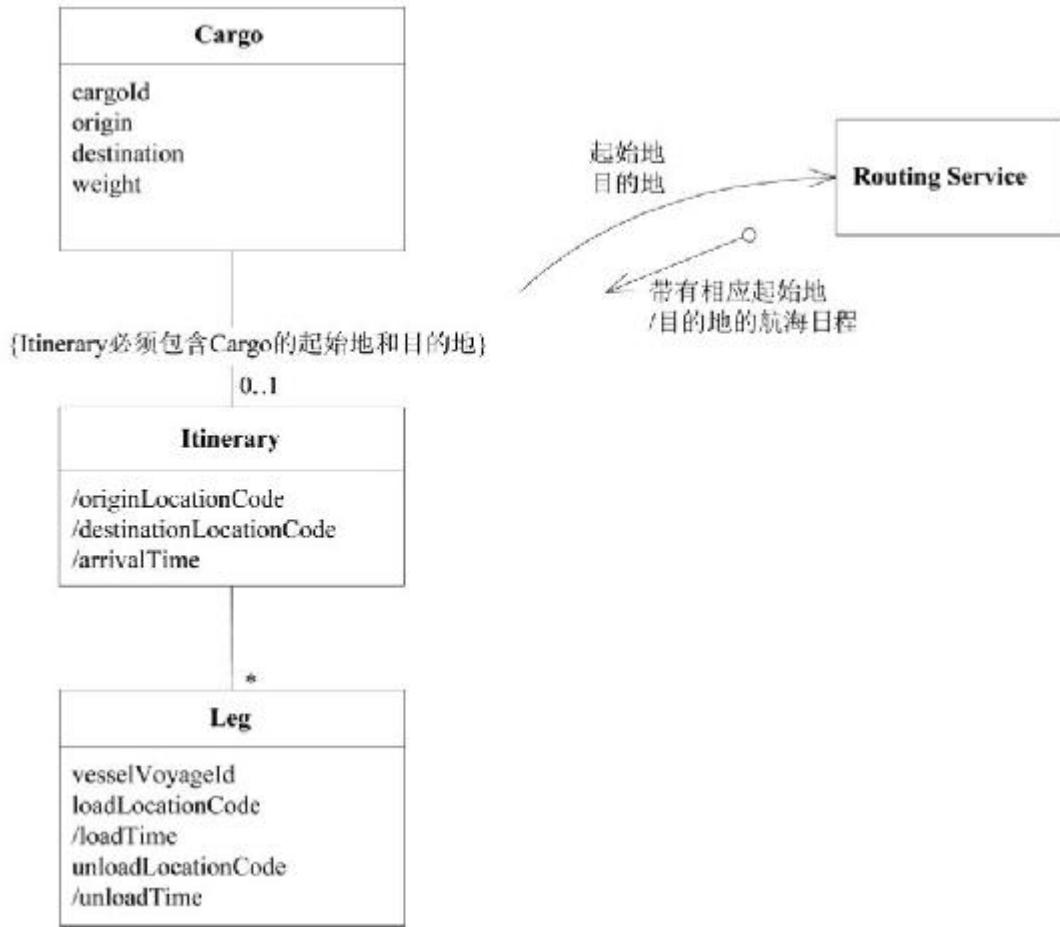


圖9-3

接下來，開發人員對代碼進行了重構，以使它能反映出新的模型。在一週內，他們很快對代碼作出了一系列的修改，每次修改都進行兩到三次重構。但是他們還沒有對預訂應用程序中的航海日程報告進行簡化，而簡化工作將會在接下來的一週開始進行。

這位開發人員一直都在仔細傾聽運輸專家的見解，並注意到「航海日程」概念的重要性。事實上，所有的數據都已收集，在航海日程報告中也已隱含了操作行為，但是，把顯式的 **Itinerary** 對像作為模型的一部分給他們帶來了新的機會。

通過重構得到顯式的 **Itinerary** 對象的益處是：

(1) 更明確地定義 **Routing Service** 接口；

- (2) 將Routing Service與預訂數據庫表解耦——Routing Service無需關心存儲邏輯；
- (3) 明確了預訂應用程序和作業支持應用程序之間的關係（即共享Itinerary對像）；
- (4) 減少重複，因為Itinerary可同時為預訂報表和作業支持應用程序提供裝貨/卸貨時間；
- (5) 從預訂報表中刪除領域邏輯，並將其移至獨立的領域層；
- (6) 擴充了UBIQUITOUS LANGUAGE，使得開發人員和領域專家之間或者開發人員內部能夠更準確地討論模型和設計。

### **9.1.2 檢查不足之處**

你所需要的概念並不總是浮在表面上，也絕不僅僅是通過對話和文檔就能讓它顯現出來。有些概念可能需要你自己去挖掘和創造。要挖掘的地方就是設計中最不足的地方，也就是操作複雜且難於解釋的地方。每當有新的需求時，似乎都會讓這個地方變得更加複雜。

有時，你很難意識到模型中丟失了什麼概念。也許你的對象能夠實現所有的功能，但是有些職責的實現卻很笨拙。而有時，你雖然能夠意識到模型中丟失了某些東西，但是卻無法找到解決方案。

這個時候，你必須積極地讓領域專家參與到討論中來。如果你足夠幸運，這些專家可能會願意一起思考各種想法，並通過模型來進行驗證。如果你沒那麼幸運，你和你的同事就不得不自己思索出不同的想法，讓領域專家對這些想法進行判斷，並注意觀察專家的表情是認同還是反對。

#### **示例 摸索利息計算模型**

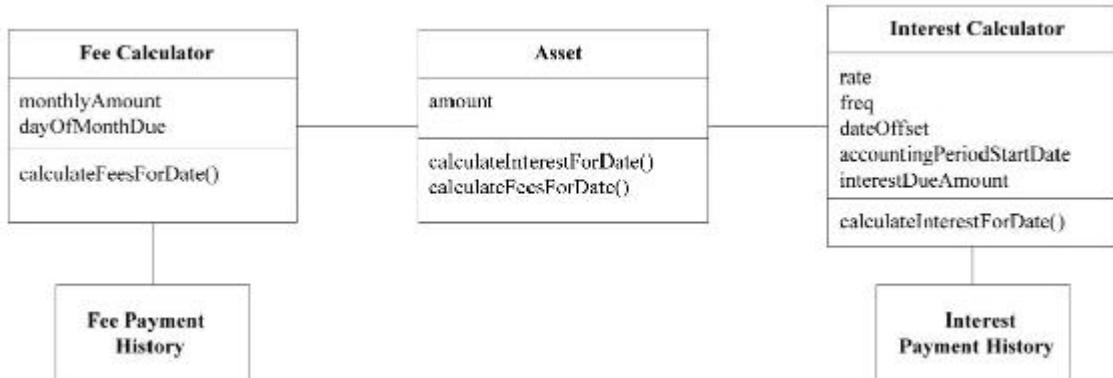


圖9-4 笨拙的模型

下面的故事以一家假想的金融公司為背景，該公司經營商業貸款和其他一些生息資產。公司開發了一個用於跟蹤這些投資及收益的應用程序，通過一項一項地添加功能來使它不斷地發展。每天晚上，公司都會運行一個批處理腳本，用於計算當天所生成的利息和費用，並把它們相應地記錄到公司的財務軟件中。

晚間批處理腳本會遍歷每筆 **Asset** (資產)，並讓其執行 `calculateInterestForDate()`，按照當天的日期來計算利息。然後，該腳本會接收返回值 (收益金額)，並將它和指定分類賬的名稱一起發送給一個 **SERVICE** (這個 **SERVICE** 提供了記賬程序的公共接口)。再由記賬軟件將收入金額過賬到指定的分類賬中。這個腳本還會對每筆 **Asset** 當日的手續費作類似的處理，並記錄到另一個不同的分類賬中。

負責這個程序的一位開發人員一直在費力地應對日益複雜的利息計算。她開始懷疑應該能找到一個更適合完成此項任務的模型。於是，她向她熟識的領域專家尋求幫助，希望專家可以協助她深入研究這個問題。

開發人員：我們的 **Interest Calculator** (利息計算器) 太複雜了。

專家：這一部分確實很複雜。還有很多情況我們都推遲考慮了。

開發人員：我知道。我們可以使用另一個不同的 **Interest Calculator** 來添加新的利息類型。但現在最大的麻煩是，如果沒有按時

支付利息，該如何去處理由此引發的各種特殊情況。

專家：其實這些不算是特殊情況。人們支付利息的方式可以非常靈活。

開發人員：記得之前我們重構Asset，將Interest Calculator從中分離出來，這對開發工作大有幫助。我們可能還需要進一步分解Asset。

專家：沒問題。

開發人員：我在想你們在討論這種利息計算時可能有另外的方式。

專家：你指的是什麼？

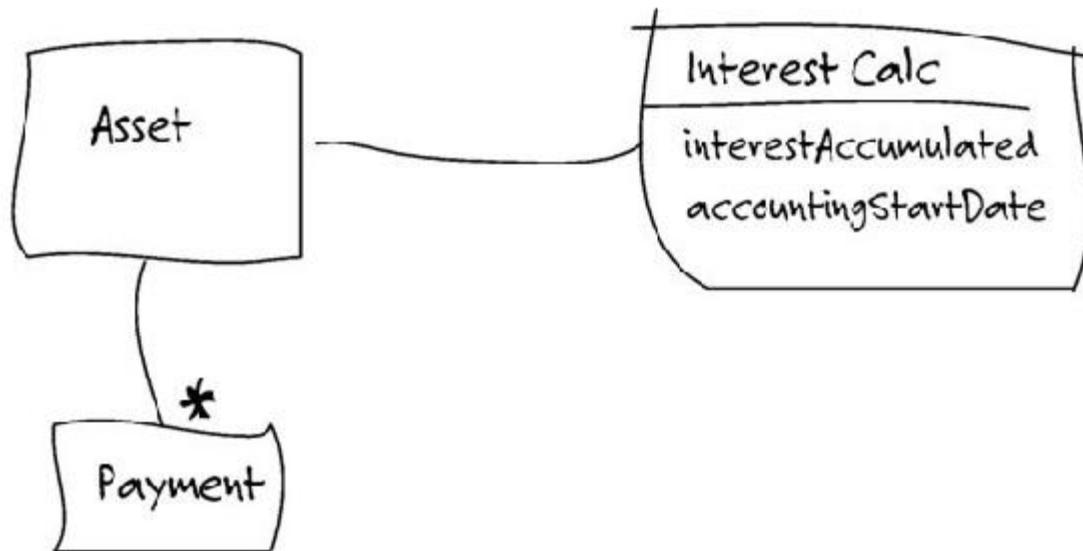
開發人員：舉個例子，假設我們正在跟蹤會計期（accounting period）內到期的未付利息。這種利息有名字嗎？

專家：哦，實際上我們並不會這麼做。利息收入和付款是完全獨立的過帳。

開發人員：所以你們不需要這個數字（到期的未付利息）？

專家：有時我們也許會看看，但這不是我們處理業務的方式。

開發人員：好吧。如果付款和利息是彼此獨立的，也許我們應該這樣建模。這看起來怎麼樣？[在白板上畫出草圖。]



## 圖9-5

專家：我想這是合乎情理的。但你只是把它從一個地方移到了另一個地方。

開發人員：不過現在Interest Calculator只負責追蹤利息收入了，付款的數目則是由Payment單獨管理。這個模型並沒有簡化什麼，但它是不是能夠更好地反映出業務慣例呢？

專家：啊。我懂了。我們能夠同時保留利息的歷史記錄嗎？就像之前模型中的 Payment History（付款歷史）一樣。

開發人員：可以。這已經被作為一項新功能提出來了。但是，它本來應該在初始設計中就加進來。

專家：哦。是這樣，我看到你以這種方式分離利息和Payment History，還以為你們要把利息分解並組織成類似於Payment History的結構。你對應計制會計（accrual basis accounting）有所瞭解嗎？

開發人員：請解釋一下。

專家：我們每天（或根據計劃安排）都會把應計利息過帳到收支總賬中。而支付的過帳方法則完全不同。你在這裡把應計利息累加起來有點不大合適。

開發人員：你是說，如果我們保留「應計利息」列表，那麼這些利息就可以根據需要來進計算總計或者……「過帳」。

專家：應該是在應計日期過帳，但是可以在任意時間內累加。費用的處理與此相同，當然，是要提交到另一個分類賬中。

開發人員：事實上，如果只計算一天或一段時間的利息，問題就會簡單得多。然後，我們就能夠解決所有這些問題了。這看起來怎麼樣？

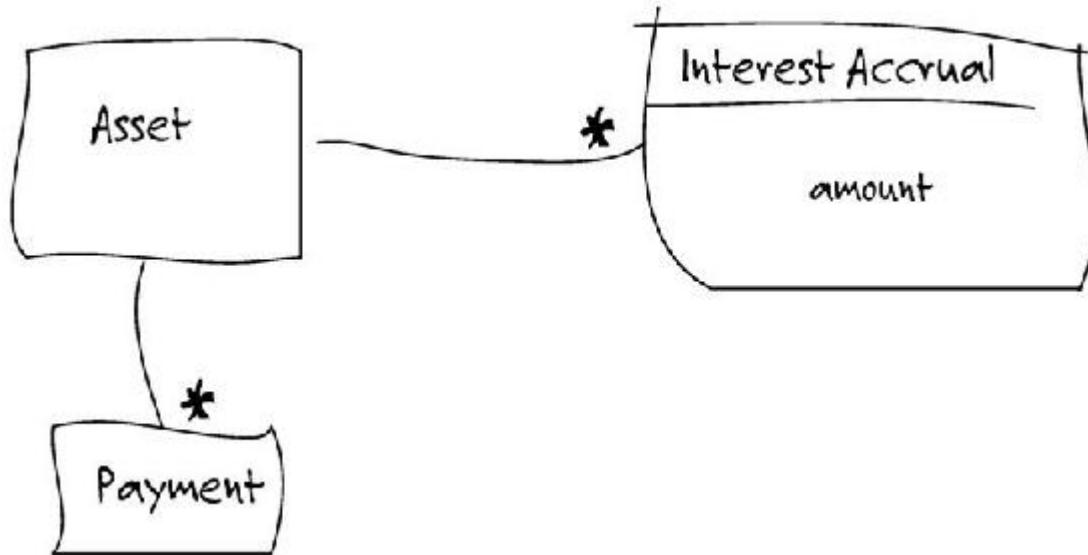


圖9-6

專家：不錯。這看起來很好。我不明白為什麼這對你來說會簡單得多。但基本上，資產之所以有價值，就是因為通過它可以累積利息、費用等。

開發人員：你是說手續費也是一樣的嗎？它們……怎麼說來著？……要過帳到不同的分類帳中？

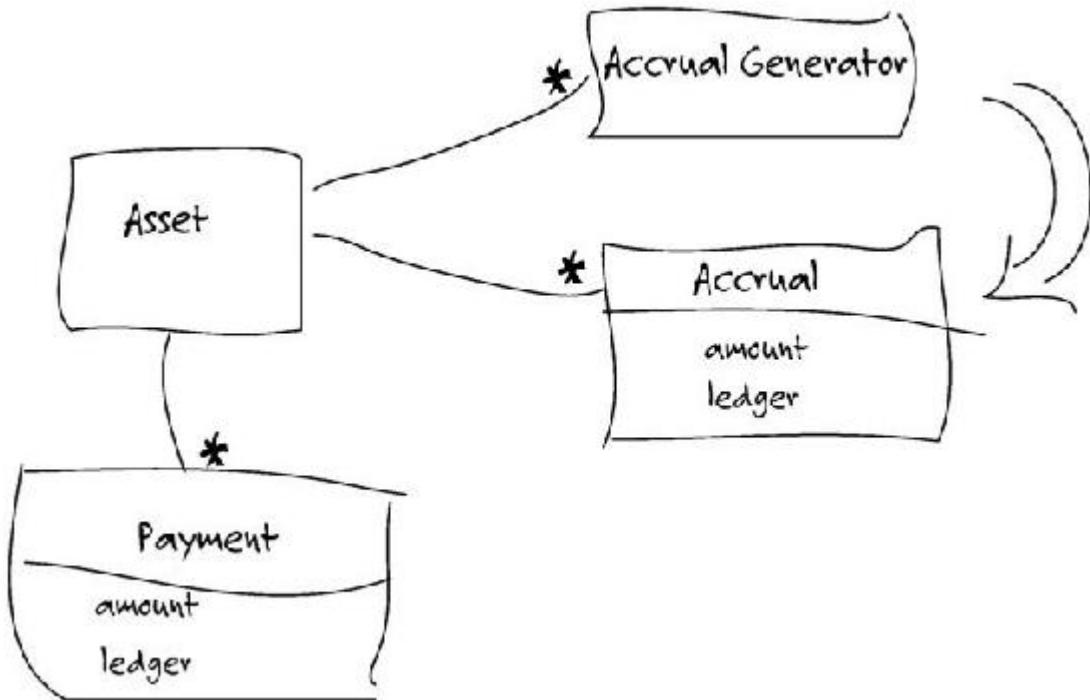


圖9-7

開發人員：在這個模型中，我們將Interest Calculator中的利息計算（或者說是應計費用的計算邏輯）與跟蹤利息的功能分開了。直到現在我才注意到Fee Calculator與Interest Calculator有很多重複的地方。此外，現在不同類型的費用也可以輕鬆地添加進來了。

專家：是的。之前的計算也是正確的，但現在變得一目瞭然了。

由於Calculator類並沒有直接與設計中的其他部分相關聯，所以這其實是一個非常簡單的重構。這位開發人員只需花幾個小時就能夠通過重寫單元測試來驗證新的語言，第二天新的設計就可以用了。最終，她得到了下面的模型。

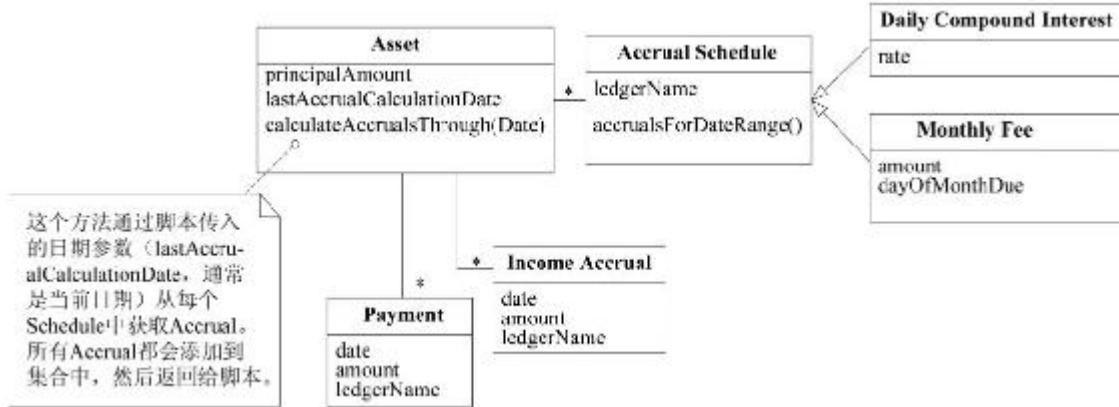


圖9-8 重構後的深層模型

在重構後的應用程序中，夜間批處理腳本會通知每個Asset執行 `calculateAccrualsThroughDate()`。其返回值是Accrual的集合，而其中的每筆金額都會過帳到指定的分類帳中。

新模型具有幾個優點，包括：

- (1) 術語「應計費用 ( accrual ) 」使UBIQUITOUS LANGUAGE更豐富；
- (2) 將應計費用從付款中分離出來；
- (3) 將領域知識（如過帳到哪個分類帳）從腳本中移出來，並放到領域層中；
- (4) 將費用與利息統一，既能夠符合業務邏輯，又可消除重複代碼；
- (5) 新形式的費用和利息可以通過Accrual Schedule直接添加到模型中。

這一次，開發人員不得不自己挖掘所需的新概念。她能夠看出利息計算的不足之處，並堅持不懈地尋找更深層次的解決方案。

她很幸運地找到一位聰明且熱忱的銀行專家作為合作夥伴。如果合作的專家不那麼主動的話，她可能會在初期犯更多的錯誤，而後則

需要更多地依賴與其他開發人員進行頭腦風暴來解決問題。這樣，程序開發的進度會放慢，但還是有可能獲得進展。

### **9.1.3 思考矛盾之處**

由於經驗和需求的不同，不同的領域專家對同樣的事情會有不同的看法。即使是同一個人提供的信息，仔細分析後也會發現邏輯上不一致的地方。在挖掘程序需求的時候，我們會不斷遇到這種令人煩惱的矛盾，但它們也為深層模型的實現提供了重要線索。有些矛盾只是術語說法上的不一致，有些則是由於誤解而產生的。但還有一種情況是專家們會給出相互矛盾的兩種說法。

天文學家伽利略曾提出過一個悖論。我們的感覺清楚地表明地球是靜止的：人們既不會被吹走也不會被拋出去。然而哥白尼提出了一個很有說服力的觀點，即地球是圍繞著太陽飛速轉動的。將這一矛盾統一起來可能會揭示出大自然運轉的某種深奧的規律。

於是，伽利略設計了一個假想實驗。如果一個騎手在奔跑的馬背上丟下一個球，這個球會掉到哪裡？顯然，這個球會隨著馬一起向前移動，直到它落在馬蹄旁邊的地面上，就像馬一直站著沒動時一樣。根據這個實驗，伽利略推導出了慣性參考系思想的早期雛形，它可以解決前面提到的悖論並可引出更為實用的物理運動模型。

我們遇到的矛盾通常不會這麼有趣，也不會具有如此深刻的意義。儘管如此，採用同樣的思考模式通常可以幫助我們透過問題領域的表面獲得更深層的理解。

要解決所有矛盾是不太現實的，甚至是不需要的。（第14章將會深入探討如何取捨以及如何處理結果。）然而，即使不去解決矛盾，我們也應該仔細思考對立的兩種看法是如何同時應用於同一個外部現實的，這會給我們帶來啟示。

### **9.1.4 查閱書籍**

在尋找模型概念時，不要忽略一些顯而易見的資源。在很多領域中，你都可以找到解釋基本概念和傳統思想的書籍。你依然需要與領域專家合作，提煉與你的問題相關的那部分知識，然後將其轉化為適用於面向對像軟件的概念。但是，查閱書籍也許能夠使你一開始就形成一致且深層的認識。

### 示例 藉助參考書來設計利息計算模型

讓我們設想一下前面討論的投資跟蹤應用程序的另一個場景。與前面一樣，這個故事的開頭也是開發人員意識到設計變得越來越笨拙，特別是Interest Calculator。但是在這個場景中，領域專家主要負責其他工作，他對幫助軟件開發項目並不十分感興趣。在這裡，開發人員不能指望專家與其一起進行頭腦風暴，幫助她探尋隱藏於表象之下的遺漏概念。

於是，她去了書店。隨意翻閱了幾本書之後，她找到了一本自己比較喜歡的會計學入門書籍，並把它粗略瀏覽了一遍。她發現書中有一整套明確定義的概念體系。其中一段文字給了她特別大的啟發：

應計制會計。這種方法把所有已經產生的收入均計到收入中（即使尚未支付），所有支出也均在產生時顯示出來（無論是已經支付還是以後才支付）。所有到期債務，包括稅金，都列入費用。

——Suzanne Caplan的Finance and Accounting:How to Keep Your Books and Manage Your Finances Without an MBA,a CPA or a Ph.D[Adams Media,2000]

開發人員再也不用自己去重新編造一個會計學出來了。在與其他開發人員進行了一些討論之後，她設計出了一個模型，如圖9-9所示。

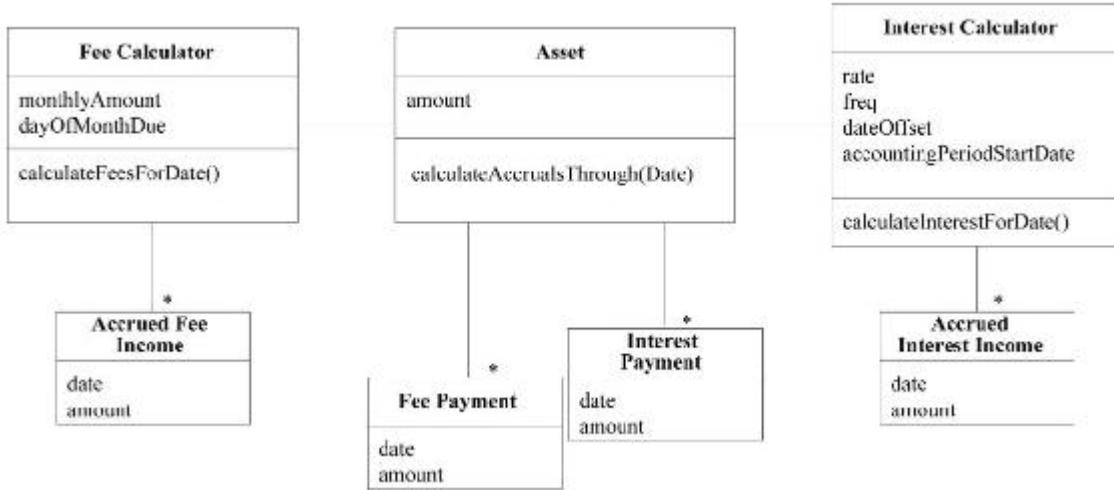


圖9-9 通過閱讀書籍而得來的略為深層的模型

她還沒有認識到收入是由Asset產生的，所以模型中依然含有Calculator。分類賬的概念還是包含在應用程序中，而不是在它本應歸屬的領域層中。但是，她確實將付款從應計收入中分離出來了（這曾是最大的問題）並且她將「應計費用」這個詞引入到模型和UBIQUITOUS LANGUAGE中。在之後的迭代過程中，模型還會得到進一步的精化。

當這位開發人員終於有機會與領域專家討論時，專家大吃一驚。她是專家遇到的第一個對其工作表現出些許興趣的程序人員。由於職責分配的原因，專家從未像之前例子那樣密切配合過——坐下來與她共同商討模型問題。但是，這位開發人員從書中獲取了知識，這使她能夠提出很好的問題，所以自此以後，專家開始認真傾聽她的見解，並盡力及時地回答她的問題。

當然，看書與諮詢領域專家並不衝突。即便能夠從領域專家那裡得到充分的支持，花點時間從文獻資料中大致瞭解領域理論也是值得的。雖然許多業務並不會像會計學或金融行業那樣具有極其細緻的模型，但大多數領域中都有一些擅於思考的人，他們已組織並抽像出了業務的一些通用的慣例。

開發人員還有另一個選擇，就是閱讀在此領域中有過開發經驗的軟件專業人員編寫的資料。例如，《分析模式》[Fowler 1997]一書的第6章可能會為她提供一個完全不同的思考方向——無論這個方向會讓開發變得更好還是更糟。閱讀書籍並不能提供現成的解決方案，但可以為她提供一些全新的實驗起點，以及在這個領域中探索過的人總結出來的經驗。這樣可以避免開發人員重複設計已有的概念。第11章將更深入地探討這一主題。

### **9.1.5 嘗試，再嘗試**

上面的例子並沒有顯示出不斷嘗試和出錯的次數。在討論過程中，我可能嘗試六七種不同的思路，然後找到一個看起來足夠清晰且實用的概念，並在模型中嘗試它。後面，隨著經驗的積累和知識的消化，我們會有更好的想法，最終，這個概念至少會被替換一次。因此，建模人員/設計人員絕對不能固執己見。

並不是所有這些方向性的改變都毫無用處。每次改變都會把開發人員更深刻的理解添加到模型中。每次重構都使設計變得更靈活並且為那些可能需要修改的地方做好準備。

我們其實別無選擇。只有不斷嘗試才能瞭解什麼有效什麼無效。企圖避免設計上的失誤將會導致開發出來的產品質量低劣，因為沒有更多的經驗可用來借鑒，同時也會比進行一系列快速實驗更加費時。

## **9.2 如何為那些不太明顯的概念建模**

面向對像範式會引導我們去尋找和創造特定類型的概念。所有事物（即使是像「應計費用」這種非常抽象的概念）及其操作行為是大部分對像模型的主要部分。它們就是面向對像設計入門書籍所講到的

「名詞和動詞」。但是，其他重要類別的概念也可以在模型中顯式地表現出來。

下面我將會描述3個這樣的類別，我在開始接觸對像時，對它們的認識並不夠清晰。我每學會一個這樣的類別，就會讓設計變得更加清晰深刻。

### **9.2.1 顯式的約束**

約束是模型概念中非常重要的類別。它們通常是隱含的，將它們顯式地表現出來可以極大地提高設計質量。

有時，約束很自然地存在於對像或方法中。Bucket ( 桶 ) 對像必須滿足一個固定規則——內容 ( contents ) 不能超出它的容量 ( capacity )，如圖9-10所示。

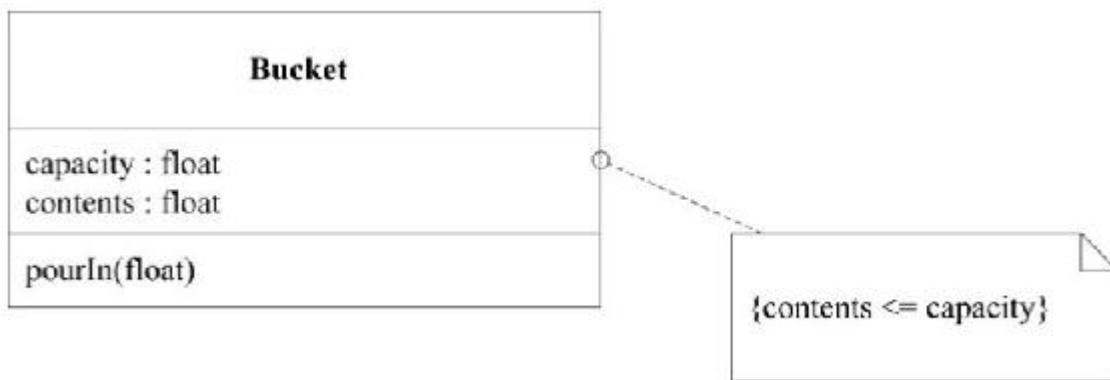


圖9-10

這樣一個簡單的固定規則可以在每次可改變內容的操作中使用一個邏輯判斷來保證。

```
class Bucket {  
    private float capacity;  
    private float contents;  
  
    public void pourIn(float addedVolume) {  
        if (contents + addedVolume > capacity) {  
            contents = capacity;  
        } else {  
            contents = contents + addedVolume;  
        }  
    }  
}
```

這裡的邏輯非常簡單，規則也很明顯。但是不難想像，在更複雜的類中這個約束可能會丟失。讓我們把這個約束提取到一個單獨的方法中，並用清晰直觀的名稱來表達它的意義。

```
class Bucket {  
    private float capacity;  
    private float contents;  
  
    public void pourIn(float addedVolume) {  
  
        float volumePresent = contents + addedVolume;  
        contents = constrainedToCapacity(volumePresent);  
    }  
  
    private float constrainedToCapacity(float volumePlacedIn) {  
        if (volumePlacedIn > capacity) return capacity;  
        return volumePlacedIn;  
    }  
}
```

這兩個版本的代碼都實施了約束，但是第二個版本與模型的關係更為明顯（這也是MODEL-DRIVEN DESIGN的基本需求）。這個規則十分簡單，使用最初形式的代碼也很容易理解，但如果要是執行的規則比較複雜的話，它們就會像所有隱式概念一樣淹沒掉被約束的對象或操作。將約束條件提取到其自己的方法中，這樣就可以通過方法名來表達約束的含義，從而在設計中顯式地表現出這條約束。現在這個約束條件就是一個「有名有姓」的概念了，我們可以用它的名字來討論它。這種方式也為約束的擴展提供了空間。比這更複雜的規則很容易就會產生比其調用者（在這裡就是pourIn()方法）更長的方法。這樣，調用者就可以簡單一些，並且只專注於處理自己的任務，而約束條件則可以根據需要進行擴展。

這種獨立方法為約束預留了一定的增加空間，但是在很多時候，約束條件是無法用單獨的方法來輕鬆表達的。或者，即使方法自身能夠保持其簡單性，但它可能會調用一些信息，但對於對象的主要職責而言，這些信息毫無用處。這種規則可能就不適合放到現有對象中。

下面是一些警告信號，表明約束的存在正在擾亂其「宿主對像」（Host Object）的設計。

- (1) 計算約束所需的數據從定義上看並不屬於這個對象。
- (2) 相關規則在多個對象中出現，造成了代碼重複或導致不屬於同一族的對象之間產生了繼承關係。
- (3) 很多設計和需求討論是圍繞這些約束進行的，而在代碼實現中，它們卻隱藏在過程代碼中。

如果約束的存在掩蓋了對象的基本職責，或者如果約束在領域中非常突出但在模型中卻不明顯，那麼就可以將其提取到一個顯式的對象中，甚至可以把它建模為一個對像和關係的集合。（The Object

Constraint Language: Precise Modeling with UML [Warmer and Kleppe 1999]一書中提供了關於這個問題的半正式的深入解決方案。)

### 示例 覆核：超訂策略

在第1章中，我們討論了一個常見的運輸業務慣例：預訂超出運輸能力10%的貨物。（貨運公司的經驗表明，這種程度的超定可以抵消因客戶臨時取消訂單而空出來的艙位，這樣貨輪基本能夠滿載起航。）

通過加入一個新類來反映Voyage和Cargo關聯中的約束，該約束不管是在圖表中還是在代碼中都能顯式地體現出來，如圖9-11所示。

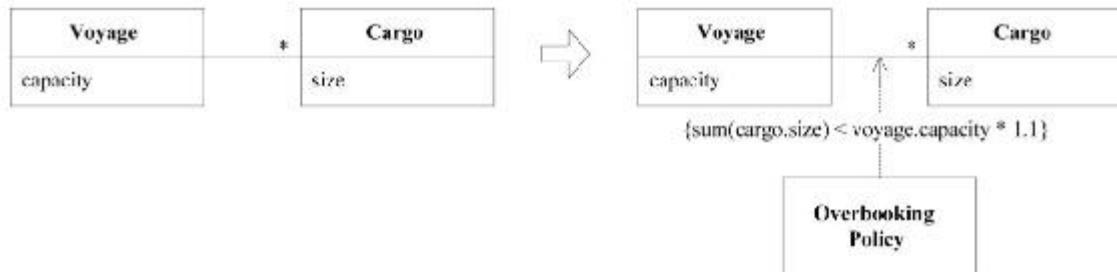


圖9-11 為顯式表達超訂策略而重構的模型

要查看完整例子的代碼和思路，請參閱1.4節示例。

## 9.2.2 將過程建模為領域對像

首先要說明的是，我們都不希望過程變成模型的主要部分。對象是用來封裝過程的，這樣我們只需考慮對象的業務目的或意圖就可以了。

在這裡，我們討論的是存在於領域中的過程，我們必須在模型中把這些過程表示出來。否則當這些過程顯露出來時，往往會使對像設計變得笨拙。

本章的第一個例子描述了用來安排貨運路線的運輸系統。安排路線的過程具有業務意義。SERVICE是顯式表達這種過程的一種方式，同時它還會將異常複雜的算法封裝起來。

如果過程的執行有多種方式，那麼我們也可以用另一種方法來處理它，那就是將算法本身或其中的關鍵部分放到一個單獨的對象中。這樣，選擇不同的過程就變成了選擇不同的對象，每個對象都表示一種不同的**STRATEGY**。（第12章將會更詳細地討論如何在領域中使用**STRATEGY**。）

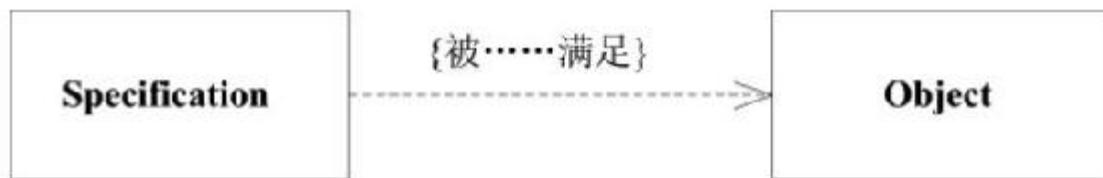
過程是應該被顯式表達出來，還是應該被隱藏起來呢？區分的方法很簡單：它是經常被領域專家提起呢，還是僅僅被當作計算機程序機制的一部分？

約束和過程是兩大類模型概念，當我們用面向對像語言編程時，不會立即想到它們，然而它們一旦被我們視為模型元素，就真的可以讓我們的設計更為清晰。

有些類別的概念很實用，但它們可應用的範圍要窄很多。為了使本章的討論更全面，我會探討一個更特殊但也非常常用的概念——規格（*specification*）。「規格」提供了用於表達特定類型的規則的精確方式，它把這些規則從條件邏輯中提取出來，並在模型中把它們顯式地表示出來。

**SPECIFICATION**是我與Martin Fowler[Evans and Fowler 1997]協作開發出來的。這個概念看起來很簡單，但是應用和實現中起來卻很微妙，因此在本節中會有大量的細節描述。在第10章中還會繼續討論**SPECIFICATION**，並對這種模式進行擴展。在閱讀完接下來對該模式的初步解釋後，你可以跳過9.2.4節，等到你真正想要應用這種模式時再回來閱讀也不遲。

### 9.2.3 模式：**SPECIFICATION**



在所有類型的應用程序中，都會有布爾值測試方法，實際上它們只是些小規則。只要它們很簡單，就可以用測試方法（如 `anIterator.hasNext()` 或 `anInvoice.isOverdue()` ）來處理它們。在 `Invoice` 類中，`isOverdue()` 的代碼是計算一條規則的算法。例如：

```
public boolean isOverdue() {  
    Date currentDate = new Date();  
    return currentDate.after(dueDate);  
}
```

但是並非所有規則都如此簡單。在同一個 `Invoice`（發票）類中，還有另外一個規則 `anInvoice.isDelinquent()`，它一開始也是用來檢查 `Invoice` 是否過期的，但僅僅是開始部分。根據客戶賬戶狀態的不同，可能會有寬限期政策。一些拖欠票據正準備再一次發出催款通知，而另一些則準備發給收賬公司。此外，還要考慮客戶的付款歷史紀錄、公司在不同產品線上的政策等。`Invoice` 作為付款請求是明白無誤的，但它很快就會消失在大量雜亂的規則計算代碼中。`Invoice` 還會發展出對領域類和子系統的各種依賴關係，而這些領域類和子系統與 `Invoice` 的基本含義無關。

到了這一步，為了簡化 `Invoice` 類，開發人員通常會將規則計算代碼重構到應用層中（在這裡就是賬單收集應用程序）。現在規則已經從領域層中分離出來，留下了一個純粹的數據對象，它將不再表達本來應該在業務模型中表示的規則。這些規則需要保留在領域層中，但是把它們放到被其約束的對象（在這裡是 `Invoice`）裡又不合適。此外，計算規則的方法中到處都是條件代碼，這也使得規則變得複雜難懂。

那些使用邏輯編程範式的開發人員會用一種不同的方式來處理這種情況。這種規則被稱為謂詞。謂詞是指計算結果為「真」或「假」的函數，並且可以使用操作符（如AND和OR）把它們連接起來以表達更複雜的規則。通過謂詞，我們可以顯式地聲明規則並在Invoice中使用這些規則。但前提是必須使用邏輯範式。

認識到這一點後，人們已經開始嘗試以對象的形式來實現邏輯規則。在這些嘗試中，有些很成熟，有些則很幼稚。有些很激進，有些則很謹慎。有些被證明很有價值，有些則被當作失敗的試驗丟到一邊。雖然項目允許進行幾次這樣的嘗試，但是，有一件事情是很清楚的：無論這個想法多麼吸引人，完全用對像來實現邏輯可是個大工程。（畢竟，邏輯編程本身就是一套建模和設計範式。）

業務規則通常不適合作為**ENTITY**或**VALUE OBJECT**的職責，而且規則的變化和組合也會掩蓋領域對象的基本含義。但是將規則移出領域層的結果會更糟糕，因為這樣一來，領域代碼就不再表達模型了。

邏輯編程提供了一種概念，即「謂詞」這種可分離、可組合的規則對象，但是要把這種概念用對像完全實現是很麻煩的。同時，這種概念過於通用，在表達設計意圖方面，它的針對性不如專門的設計那麼好。

幸運的是，我們並不真正需要完全實現邏輯編程即可從中受益。大部分規則可以歸類為幾種特定的情況。我們可以借用謂詞概念來創建可計算出布爾值的特殊對象。那些難於控制的測試方法可以巧妙地擴展出自己的對象。它們都是些小的真值測試，可以提取到單獨的**VALUE OBJECT**中。而這個新對像則可以用來計算另一個對象，看看謂詞對那個對象的計算是否為「真」。



圖9-12

換言之，這個新對象就是一個規格。SPECIFICATION（規格）中聲明的是限制另一個對像狀態的約束，被約束對象可以存在，也可以不存在。SPECIFICATION有多種用途，其中一種體現了最基本的概念，這種用途是：SPECIFICATION可以測試任何對像以檢驗它們是否滿足指定的標準。

因此：

**為特殊目的創建謂詞形式的顯式 VALUE OBJECT**。  
**SPECIFICATION**就是一個謂詞，可用來確定對象是否滿足某些標準。

許多SPECIFICATION都是具有特殊用途的簡單測試，就像在拖欠票據示例中的規格一樣。當規則很複雜時，可以擴展這種概念，對簡單的規格進行組合，就像用邏輯運算符把多個謂詞組合起來一樣。（這種技術將在下一章中討論。）基本模式保持不變，並且提供了一種從簡單模型過渡到複雜模型的途徑。

拖欠票據的例子可以使用SPECIFICATION來建模，如圖9-13所示。在規格中聲明拖欠的含義，對任意的Invoice對像進行計算並做出判斷。

SPECIFICATION將規則保留在領域層。由於規則是一個完備的對象，所以這種設計能夠更加清晰地反映模型。利用工廠，可以用來自其他資源（如客戶賬戶或者企業政策數據庫）的信息對規格進行配鎔。之所以使用FACTORY，是為了避免Invoice直接訪問這些資源，因為這樣會使得Invoice與這些資源發生不正確的關聯（Invoice的基本職

責是請求付款，而這些資源與這一職責無關）。在這個例子中，我們將創建Delinquent Invoice Specification（拖欠發票規格）來對一些發票進行評估，這個SPECIFICATION用過之後就被丟掉，因此可以將評估日期直接放在SPECIFICATION中，這真是一次不錯的簡化。我們可以用簡單直接的方式為SPECIFICATION提供完成其職責所需的信息。

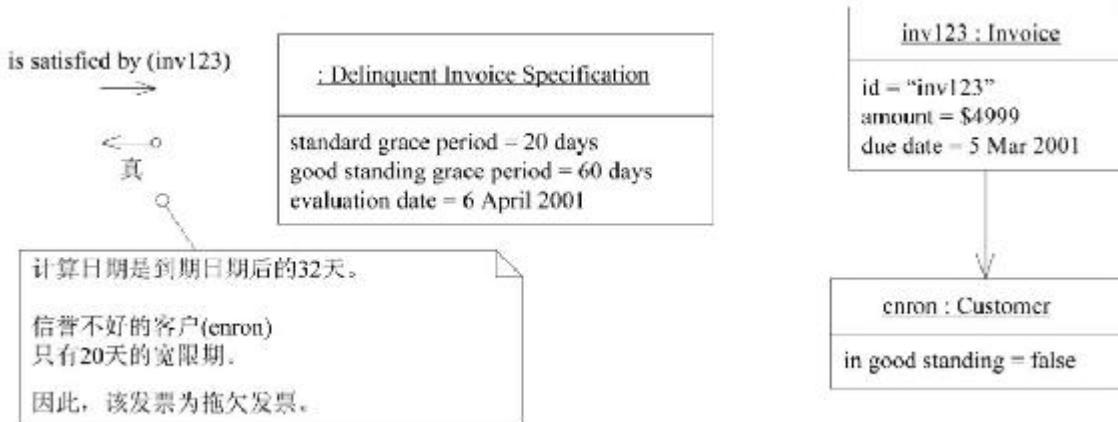


圖9-13 作為SPECIFICATION被提取出來的更為詳細的拖欠規則

SPECIFICATION的基本概念非常簡單，這能幫助我們思考領域建模問題。但是MODEL-DRIVEN DRISIGN要求我們開發出一個能夠把概念表達出來的有效實現。要實現這個目標，必須要更深入地挖掘應用這個模式的方法。領域模式不僅僅是UML圖中好的想法，也應該可以為MODEL-DRIVEN DRISIGN中的編程問題提供解決方案。

只要恰當地應用模式，就可以得出一整套如何解決領域建模問題的思路，同時也可以從這種長時間搜尋有效實現的經驗中受益。下面的SPECIFICATION討論詳細介紹了功能和實現方法的多種選擇。模式並不像菜譜那麼死板。它可以讓你以模式的經驗為起點來開發自己的解決方案，並為你討論手頭工作提供了語言。

在第一次閱讀時，你可以快速瀏覽關鍵概念。以後碰到具體情況時，可以再回過頭來閱讀並從細節討論中獲取經驗。然後就可以開始設計你自己的解決方案了。

## 9.2.4 SPECIFICATION的應用和實現

SPECIFICATION最有價值的地方在於它可以將看起來完全不同的應用功能統一起來。出於以下3個目的中的一個或多個，我們可能需要指定對象的狀態。

- (1) 驗證對象，檢查它是否能滿足某些需求或者是否已經為實現某個目標做好了準備。
- (2) 從集合中選擇一個對像（如上述例子中的查詢過期發票）。
- (3) 指定在創建新對像時必須滿足某種需求。

這3種用法（驗證、選擇和根據要求來創建）從概念層面上來講是相同的。如果沒有諸如SPECIFICATION這樣的模式，相同的規則可能會表現為不同的形式，甚至有可能是相互矛盾的形式。這樣就會喪失概念上的統一性。通過應用SPECIFICATION模式，我們可以使用一致的模型，儘管在實現時可能需要分開處理。

### 驗證

規格的最簡單用法是驗證，這種用法也最能直觀地展示出它的概念，如圖9-14所示。

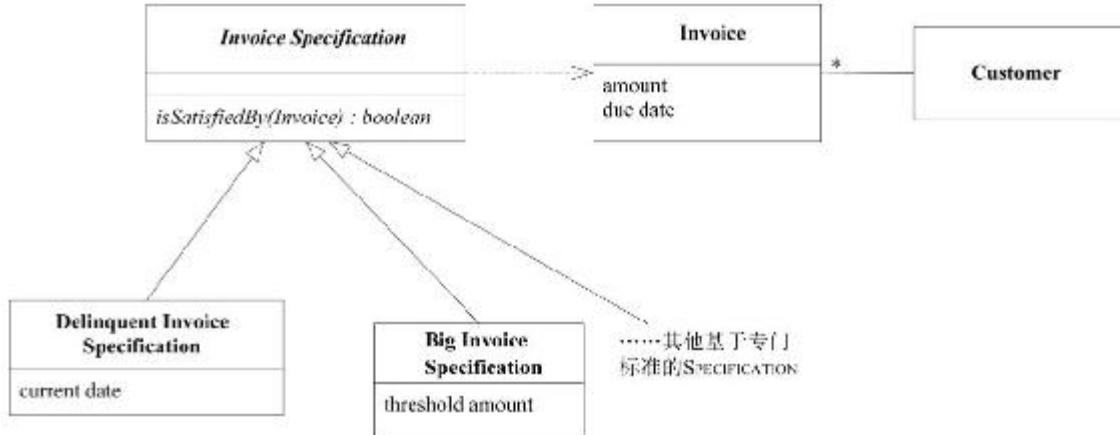


圖9-14 應用SPECIFICATION進行驗證的模型

```

class DelinquentInvoiceSpecification extends
    InvoiceSpecification {
    private Date currentDate;
    // An instance is used and discarded on a single date

    public DelinquentInvoiceSpecification(Date currentDate) {
        this.currentDate = currentDate;
    }

    public boolean isSatisfiedBy(Invoice candidate) {
        int gracePeriod =
            candidate.customer().getPaymentGracePeriod();
        Date firmDeadline =
            DateUtility.addDaysToDate(candidate.dueDate(),
                gracePeriod);
        return currentDate.after(firmDeadline);
    }
}

```

現在，假設當銷售人員看到一個欠賬客戶的信息時，系統需要顯示一個紅旗標識。我們只需要在客戶類中編寫一個方法即可，類似於下面這段代碼：

```

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();
        if (delinquentSpec.isSatisfiedBy(candidate)) return true;
    }
    return false;
}

```

## 選擇 (或查詢)

驗證是對一個獨立的對象進行測試，檢查它是否滿足某些標準，然後客戶可能根據驗證的結論來採取行動。另一種常見需求是根據某些標準從對像集合中選擇一個子集。**SPECIFICATION**概念同樣可以在此應用，但是實現問題會有所不同。

假設應用程序的需求是列出所有拖欠發票的客戶。那麼從理論上來說，我們依然可以使用之前定義的 **Delinquent Invoice Specification**，但實際上我們可能不得不去修改它的實現。為了證明二者的概念是相同的，讓我們首先假設發票的數量很少，可能已經全部裝入內存了。在這種情況下，驗證功能的最直接實現方式依然可用。**Invoice Repository**可以用一個一般化的方法來基於**SPECIFICATION**選擇**Invoice**：

```
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;  
}
```

這樣，用一行代碼即可獲得所有拖欠發票的集合：

```
Set delinquentInvoices = invoiceRepository.selectSatisfying(  
    new DelinquentInvoiceSpecification(currentDate));
```

上面這行代碼建立了操作背後的概念。當然，**Invoice**對象可能並不在內存中。也有可能會有成千上萬個**Invoice**對象。在典型的業務系統中，數據很可能會存儲在關係數據庫中。我們在前面的章節中曾經指出，在與其他技術交互使用時，很容易分散我們對模型的注意力。

關係數據庫具有強大的查詢能力。我們如何才能充分利用這種能力來有效解決這一問題，同時又能保留SPECIFICATION模型呢？**MODEL-DRIVEN DESIGN**要求模型與實現保持同步，但它同時也讓我們可以自由選擇能夠準確捕捉模型意義的實現方式。幸運的是，SQL是用於編寫SPECIFICATION的一種很自然的方式。

下面是個簡單的例子，其中查詢被封裝在驗證規則所在的類中。我們在Invoice Specification中添加了一個方法，該方法在Delinquent Invoice Specification子類中得以實現：

```
public String asSQL() {  
    return  
        "SELECT * FROM INVOICE, CUSTOMER" +  
        " WHERE INVOICE.CUST_ID = CUSTOMER.ID" +  
        " AND INVOICE.DUE_DATE + CUSTOMER.GRACE_PERIOD" +  
        " < " + SQLUtility.dateAsSQL(currentDate);  
}
```

SPECIFICATION與REPOSITORY的搭配非常合適，REPOSITORY作為一種構造塊機制，提供了對領域對象的查詢訪問，並且把數據庫接口封裝起來（參見圖9-15）。

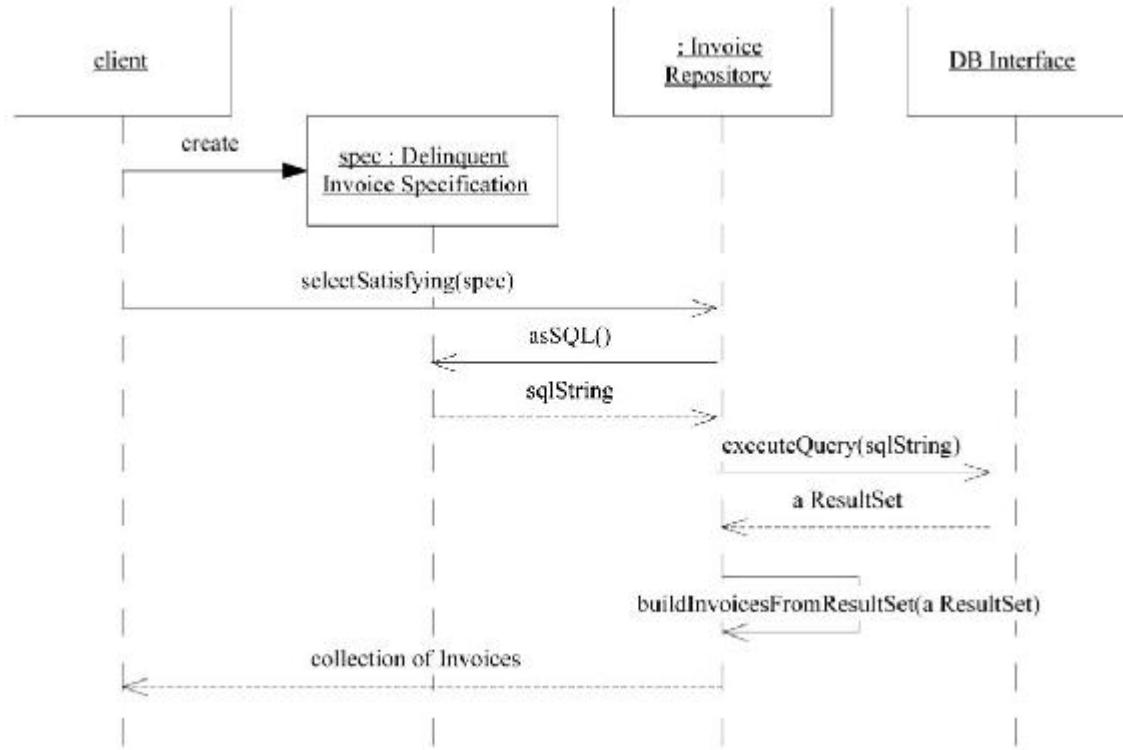


圖9-15 REPOSITORY和SPECIFICATION之間的交互

現在的設計有一些問題。最重要的問題是，表結構的細節本應該被隔離到一個映射層中（這個映射層把領域對像關聯到關係表），現在卻洩漏到了DOMAIN LAYER中。這樣一來，這些表結構信息發生了隱性的重複，因此導致對Invoice和Customer對象的修改和維護變得很麻煩，因為現在必須在多個地方跟蹤它們的映射變化。但是，這個例子只是一個簡單的例證，用來說明如何將規則放在一個地方。一些對像關係映射框架提供了用模型對像和屬性來表達這種查詢的方式，並在基礎設施層中創建實際的SQL語句。這樣就可以兩全其美了。

如果無法把SQL語句創建到基礎設施中，還可以重寫一個專用的查詢方法並把它添加到Invoice Repository中，這樣就把SQL語句從領域對像中分離出來了。為了避免在REPOSITORY中嵌入規則，必須採用更為通用的方式來表達查詢，這種方式不捕捉規則但是可以通過組

合或放錯在上下文中來表達規則（在這個例子中，使用的是雙分派模式）。

```
public class InvoiceRepository {  
  
    public Set selectWhereGracePeriodPast(Date aDate){  
        //This is not a rule, just a specialized query  
        String sql = whereGracePeriodPast_SQL(aDate);  
        ResultSet queryResultSet =  
            SQLDatabaseInterface.instance().executeQuery(sql);  
        return buildInvoicesFromResultSet(queryResultSet);  
    }  
  
    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);  
    }  
  
    public Set selectSatisfying(InvoiceSpecification spec) {  
        return spec.satisfyingElementsFrom(this);  
    }  
}
```

Invoice Specification 中的 asSql() 方法被替換為 satisfyingElementsFrom(Invoice-Repository) ，並在 Delinquent Invoice Specification 中以如下的方式實現：

```
public class DelinquentInvoiceSpecification {  
    // Basic DelinquentInvoiceSpecification code here  
  
    public Set<Invoice> satisfyingElementsFrom(  
        InvoiceRepository repository) {  
        //Delinquency rule is defined as:  
        //  "grace period past as of current date"  
        return repository.selectWhereGracePeriodPast(currentDate);  
    }  
}
```

這段代碼將SQL語句定義在**REPOSITORY**中，而應該使用哪個查詢則由**SPECIFICATION**來控制。**SPECIFICATION**中並沒有定義完整的規則，但規則的核心已位於其中——指明瞭什麼條件構成了拖欠（即超過寬限期）。

現在，**REPOSITORY**中包含的查詢非常具有針對性，可能只適用於這種情況。雖然這是可以接受的，但是根據拖欠發票在過期發票中所佔數量的不同，我們可以選擇一種更通用的**REPOSITORY**解決方案，使得性能仍然很好，同時又使**SPECIFICATION**的使用更易理解。

```

public class InvoiceRepository {

    public Set selectWhereDueDateIsBefore(Date aDate) {
        String sql = whereDueDateIsBefore_SQL(aDate);
        ResultSet queryResultSet =
            SQLDatabaseInterface.instance().executeQuery(sql);
        return buildInvcicesFromResultSet(queryResultSet);
    }

    public String whereDueDateIsBefore_SQL(Date aDate) {
        return
            "SELECT * FROM INVOICE"
            + " WHERE INVOICE.DUE_DATE"
            + " < " + SQLUtility.dateAsSQL(aDate);
    }

    public Set selectSatisfying(InvoiceSpecification spec) {
        return spec.satisfyingElementsFrom(this);
    }
}

public class DelinquentInvoiceSpecification {
    //Basic DelinquentInvoiceSpecification code here

    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();
            if (this.isSatisfiedBy(anInvcice))
                delinquentInvoices.add(anInvoice);
        }
        return delinquentInvoices;
    }
}

```

因為我們取出了更多Invoice並在內存中對其進行篩選，上面的代碼會有性能方面的影響。這種以降低性能來實現更好的職責分離的代價是否可以接受完全取決於環境因素。**SPECIFICATION** 和

**REPOSITORY**之間的交互有很多種實現方式，不但能夠利用開發平臺的優勢，還可以保證基本職責的實施。

有時，為了改善性能（或者更有可能是為了加強安全性），我們可能把查詢實現為服務器上的存儲過程。在這種情況下，**SPECIFICATION**可能只帶有存儲過程允許的參數。除此之外，這些不同實現之間的模型並沒有什麼不同。我們可以自由選擇實現方式，除非模型中有特別的約束條件。這麼做的代價是更加難於編寫和維護查詢。

上面的討論基本上沒有涉及將**SPECIFICATION**與數據庫結合時所面臨的挑戰，我並不想在這裡說明所有可能需要考慮的問題，而只是想簡單介紹一下必須要做出的選擇。**Mee**和**Hieatt**在[Fowler 2002]中討論了用規格設計**REPOSITORY**時遇到的一些技術問題。

### 根據要求來創建（生成）

如果五角大樓需要一架新式的噴氣式戰鬥機，政府官員們會先編寫規格。在規格中可能會要求這架噴氣機的速度達到2馬赫，航程1800英里，並且成本不高於5000萬美元，等等。無論規格有多詳細，它都不是飛機的設計，更不是飛機本身。航空航天工程公司將接受這份規格並且據此創建出一個或多個設計。各個競爭公司可能會提出不同的設計，所有這些方案都需要滿足原始規格。

很多計算機程序都能夠生成一些工件，這些工件是需要被指定的。當你在字處理軟件文檔中插入圖片時，文字會環繞在圖片周圍。你已指定了圖片的位臘，可能也指定了文字環繞的樣式。這樣，字處理軟件就可以按照你指定的規格來將頁面上的文字擺放到正確的位臘。

儘管乍看起來並不明顯，但是這種**SPECIFICATION**概念與應用於驗證和選擇的規格並無二致。都是在為尚未創建的對象指定標準。但

是，SPECIFICATION的實現則會大不相同。這種SPECIFICATION與查詢不同，它不用來過濾已存在對像；也與驗證不同，並不用來測試已有對象。在這裡，我們要創建或重新配鎔滿足SPECIFICATION的全新對像或對像集合。

如果不使用SPECIFICATION，可以編寫一個生成器，其中包含可創建所需對象的過程或指令集。這種代碼隱式地定義了生成器的行為。

反過來，我們也可以使用描述性的SPECIFICATION來定義生成器的接口，這個接口就顯式地約束了生成器產生的結果。這種方法具有以下幾個優點。

生成器的實現與接口分離。SPECIFICATION聲明瞭輸出的需求，但沒有定義如何得到輸出結果。

接口把規則顯式地表示出來，因此開發人員無需理解所有操作細節即可知曉生成器會產生什麼結果。而如果生成器是採用過程化的方式定義的，那麼要想預測它的行為，唯一的途徑就是在不同的情況下運行或去研究每行代碼。

接口更為靈活，或者說我們可以增強其靈活性，因為需求由客戶給出，生成器唯一的職責就是實現SPECIFICATION中的要求。

最後一點也很重要。這種接口更加便於測試，因為接口顯式地定義了生成器的輸入，而這同時也可用來驗證輸出。也就是說，傳入生成器接口的用於約束創建過程的同一個SPECIFICATION也可發揮其驗證的作用（如果實現方式能夠支持這一點的話），以保證被創建的對象是正確的。（這是ASSERTION的例子，將會在第10章中討論。）

根據要求來創建可以是從頭創建全新對象，也可以是配鎔已有對象來滿足SPECIFICATION。

## 示例 化學品倉庫打包程序

假設有一個倉庫，裡面用類似於貨車車廂的大型容器存放各種化學品。有些化學品是惰性的，可以隨意擺放。有些則是易揮發的，必須放於特製的通風容器中。還有一些是易爆品，必須保存於特製的防爆容器中。還有一些規則是關於如何在容器中混裝化學品的。

我們的目標是編寫出一個軟件，用於尋找一種安全而高效地在容器中放臵化學品的方式，如圖9-16所示。

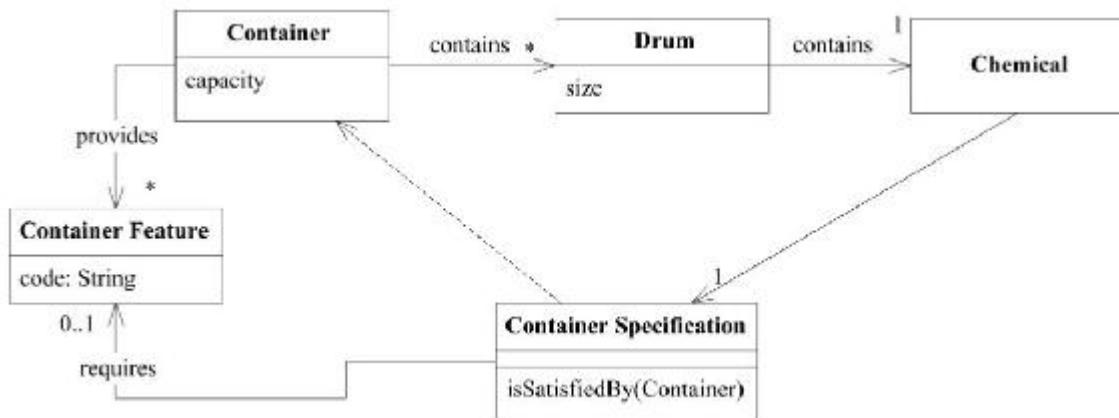


圖9-16 倉庫存儲模型

我們可以首先從編寫一個過程——取出一個化學品並將其放臵在一個容器中——開始，但是讓我們從驗證問題開始著手吧。這種方式讓我們必須顯式描述規則，同時也提供了一種測試最終實現的方式。

每種化學品都有一個容器SPECIFICATION。

化 學 品	容器SPECIFICATION
TNT	防爆容器
砂	
生物样本	不能与易燃品混装
氯水	通风容器

現在，如果將這些規格編寫成Container Specification，就可以提出一種把化學品混裝在容器中的配臵方法，並測試它是否滿足這些約束條件。

容器特性	物 品	是否满足规格？
防 爆	20 磅 TNT	✓
	500 磅砂	✓
	50 磅生物样本	✓
	氯水	✗

Container Specification的isSatisfied()方法用來檢查是否存在所需的ContainerFeature。例如，易爆化學品的規格會尋找「防爆」特性：

```
public class ContainerSpecification {
    private ContainerFeature requiredFeature;

    public ContainerSpecification(ContainerFeature required) {
        requiredFeature = required;
    }

    boolean isSatisfiedBy(Container aContainer) {
        return aContainer.getFeatures().contains(requiredFeature);
    }
}
```

下面是設置易爆化學品的客戶端示例代碼：

```
tnt.setContainerSpecification(
    new ContainerSpecification(ARMORED));
```

Container 對象的isSafelyPacked()方法用來保證 Container 具有 Chemical 要求的所有特性。

```
boolean isSafelyPacked() {
    Iterator it = contents.iterator();
    while (it.hasNext()) {
        Drum drum = (Drum) it.next();
        if (!drum.containerSpecification().isSatisfiedBy(this))
            return false;
    }
    return true;
}
```

到了這一步，我們就可以編寫一個監控程序，用來監視庫存數據庫並報告不安全狀況。

```
Iterator it = containers.iterator();
while (it.hasNext()) {
    Container container = (Container) it.next();
    if (!container.isSafelyPacked())
        unsafeContainers.add(container);
```

客戶並沒有要求我們編寫這樣一個軟件。讓業務人員知道這個程序當然很好，但客戶的要求是設計一個打包程序。而現在我們得到的是打包的測試程序。這些對領域的理解和基於**SPECIFICATION**的模型使我們有能力為服務定義一個清晰而簡單的接口，這個服務可接受**Drum**和**Container**集合併將它們按照規則進行打包。

```
public interface WarehousePacker {
    public void pack(Collection containersToFill,
                     Collection drumsToPack) throws NoAnswerFoundException;

    /* ASSERTION: At end of pack(), the ContainerSpecification
     * of each Drum shall be satisfied by its Container.
     * If no complete solution can be found, an exception shall
     * be thrown. */
}
```

現在，為履行**Packer**服務的職責，我們的任務就是設計一個優化的約束求解方案。這一任務已經與程序中的其他部分分離開來，因此其他部分的實現機制不會對這個部分的設計產生影響。（詳見第10章和第15章。）然而，控制打包的規則並沒有從領域對像中提取出來。

### 示例 倉庫打包程序的可工作的原型

為了讓倉庫打包軟件有效工作而編寫優化邏輯，這是一項艱巨的工作。一個小組的開發人員和業務專家已經分頭開始工作了，但是編碼工作尚未進行。同時，另一個小組正在開發一個應用程序，該程序允許用戶從數據庫中獲取庫存並提交給Packer處理，最後分析打包結果。這個小組是面向預期的Packer進行設計的。但是他們能做的只是模擬一個用戶界面，編寫一些數據庫集成代碼。他們無法為用戶顯示一個具有實際行為的界面，因此無法獲得良好的反饋。同樣，Packer小組也在閉門造車。

通過倉庫打包程序示例中創建的領域對像和SERVICE接口，開發應用程序的小組認識到他們可以構建一個非常簡單的Packer實現代碼，這有助於開發工作獲得進展，同時可以與其他小組協同工作並建立起反饋循環，但這只有在端到端的系統中才可以完全發揮作用。

```
public class Container {  
    private double capacity;  
    private Set contents; //Drums  
  
    public boolean hasSpaceFor(Drum aDrum) {  
        return remainingSpace() >= aDrum.getSize();  
    }  
}
```

```

    }

    public double remainingSpace() {
        double totalContentSize = 0.0;
        Iterator it = contents.iterator();
        while (it.hasNext()) {
            Drum aDrum = (Drum) it.next();
            totalContentSize = totalContentSize + aDrum.getSize();
        }
        return capacity - totalContentSize;
    }

    public boolean canAccommodate(Drum aDrum) {
        return hasSpaceFor(aDrum) &&
            aDrum.getContainerSpecification().isSatisfiedBy(this);
    }

}

public class PrototypePacker implements warehousePacker {
    public void pack(Collection containers, Collection drums)
        throws NoAnswerFoundException {
        /* This method fulfills the ASSERTION as written. However,
         * when an exception is thrown, Containers' contents may
         * have changed. Rollback must be handled at a higher
         * level. */
        Iterator it = drums.iterator();
        while (it.hasNext()) {
            Drum drum = (Drum) it.next();
            Container container =
                findContainerFor(containers, drum);
            container.add(drum);
        }
    }
    public Container findContainerFor(
        Collection containers, Drum drum)
        throws NoAnswerFoundException {
        Iterator it = containers.iterator();
        while (it.hasNext()) {
            Container container = (Container) it.next();
            if (container.canAccommodate(drum))

```

```
        return container;
    }

    throw new NoAnswerFoundException();
}

}
```

當然，上述代碼有很多不足之處。它可能會將砂打包到特製容器中，這就導致在打包危險化學品時，特製容器已經沒有多餘的空間了。顯然，它沒有對空間的利用進行優化。但是很多優化方面的問題無論怎樣都無法得到完美的解決。而這段實現代碼確實遵循了到目前為止已聲明過的所有規則。

### 通過可工作的原型來擺脫開發僵局

有的團隊必須要等待另一個團隊編寫出代碼後才可以繼續工作。而這兩個團隊都要等到代碼完全整合後才可以測試組件或從用戶那裡獲取反饋。這種僵局通常可以通過關鍵組件的模型驅動原型來緩解，即使原型並不滿足所有需求也可以。當實現與接口分離時，只要有可以工作的實現，項目工作就可以並行地開展下去。時機成熟的時候，可以用更為高效的實現來替代原型。同時，系統中的其他部分也能在開發期間與原型進行交互。

有了這個原型，應用程序的開發人員就可以全速開展工作了，包括進行所有與外部系統的集成。在領域專家對原型進行研究並確認自己的想法後，Packer開發小組也能夠得到專家的反饋，從而幫助他們

自己理清需求和優先級。Packer小組決定接管這個原型並對其進行調整，以便測試他們的想法。

同時，他們還使接口與最新設計保持同步，以推動應用程序和一些領域對象的重構，從而盡早解決集成問題。

一旦完成複雜的Packer程序，集成就是輕而易舉的事情了，因為它有一個描述得很清楚的接口，應用程序在與原型交互的時候也是根據相同的接口和**ASSERTION**編寫的。

專家們花費了幾個月的時間才得到了正確的優化算法。用戶與原型交互時的反饋使他們受益匪淺。同時，系統中的其他部分在開發期間也能夠與原型進行交互。

這裡的例子演示瞭如何通過更巧妙的模型使「最簡單卻可能非常最有效的事物」成為可能。我們可以用幾十行簡單易懂的代碼編寫出複雜組件的功能原型。如果不用**MODEL-DRIVEN DESIGN**，系統會更難理解和升級（因為Packer與設計的其他部分更緊密地耦合在一起），在這種情況下，開發原型可能會更加耗時。

## 第10章 柔性設計



軟件的最終目的是為用戶服務。但首先它必須為開發人員服務。在強調重構的軟件開發過程中尤其如此。隨著程序的演變，開發人員將重新安排並重寫每個部分。他們會把原有的領域對像集成到應用程序中，也會讓它們與新的領域對像進行集成。甚至幾年以後，負責維護的程序員還將修改和擴充代碼。人們必須要做這些工作，但他們是否願意呢？

當具有複雜行為的軟件缺乏良好的設計時，重構或元素的組合會變得很困難。一旦開發人員不能十分肯定地預知計算的全部含意，就會出現重複。當設計元素都是整塊的而無法重新組合的時候，重複就是一種必然的結果。我們可以對類和方法進行分解，這樣可以更好地重用它們，但這些小部分的行為又變得很難跟蹤。如果軟件沒有一個條理分明的設計，那麼開發人員不僅不願意仔細地分析代碼，他們更不願意修改代碼，因為修改代碼會產生問題——要麼加重了代碼的混亂狀態，要麼由於某種未預料到的依賴而破壞了某些東西。在任何一種系統中（除非是一些非常小的系統），這種不穩定性使我們很難開發出豐富的功能，而且限制了重構和迭代式的精化。

為了使項目能夠隨著開發工作的進行加速前進，而不會由於它自己的老化停滯不前，設計必須要讓人們樂於使用，而且易於做出修改。這就是柔性設計（*supple design*）。

柔性設計是對深層建模的補充。一旦我們挖掘出隱式概念，並把它們顯示地表達出來之後，就有了原料。通過迭代循環，我們可以把這些原料打造成有用的形式：建立的模型能夠簡單而清晰地捕獲主要關注點；其設計可以讓客戶開發人員真正使用這個模型。在設計和代碼的開發過程中，我們將獲得新的理解，並通過這些理解改善模型概念。我們一次又一次回到迭代循環中，通過重構得到更深刻的理解。但我們究竟要獲得什麼樣的設計呢？在這個過程中應該進行哪些實驗？這正是本章要討論的內容。

很多過度設計（*overengineering*）藉著靈活性的名義而得到合理的外衣。但是，過多的抽象層和間接設計常常成為項目的絆腳石。看一下真正為用戶帶來強大功能的軟件設計，你常常會發現一些簡單的東西。簡單並不容易做到。為了把創建的元素裝配到複雜系統中，而且在裝配之後仍然能夠理解它們，必須堅持模型驅動的設計方法，與

此同時還要堅持適當嚴格的設計風格。要創建或使用這樣的設計，可能需要我們掌握相對熟練的設計技巧。

開發人員扮演著兩個角色，而設計必須要為這兩個角色服務。同一個人可能會同時承擔這兩種角色，甚至在幾分鐘之內來回變換角色，但角色與代碼之間的關係是不同的。一個角色是客戶開發人員，負責將領域對像組織成應用程序代碼或其他領域層代碼，以便發揮設計的功能。柔性設計能夠揭示深層次的底層模型，並把它潛在的部分明確地展現出來。客戶開發人員可以靈活地使用一個最小化的、鬆散耦合的概念集合，並用這些概念來表示領域中的眾多場景。設計元素非常自然地組合到一起，其結果也是健壯的，可以被清晰地刻畫出來，而且也是可以預知的。

同樣重要的是，設計也必須為那些修改代碼的開發人員服務。為了便於修改，設計必須易於理解，必須把客戶開發人員正在使用的同一個底層模型表示出來。我們必須按照領域深層模型的輪廓進行設計，以便大部分修改都可以靈活地完成。代碼的結果必須是完全清晰明瞭的，這樣才容易預見到修改的影響。

早期的設計版本通常達不到柔性設計的要求。由於項目的時間期限和預算的緣故，很多設計一直就是僵化的。我也從未見過有哪個大型程序自始至終都是柔性的。但是，當複雜性阻礙了項目的前進時，就需要仔細修改最關鍵、最複雜的地方，使之變成一個柔性設計，這樣才能突破複雜性帶給我們的限制，而不會陷入遺留代碼維護的麻煩中。

設計這樣的軟件並沒有公式，但我精選了一組模式，從我自己的經驗來看，這些模式如果運用得當的話，就有可能獲得柔性設計。這些模式和示例展示了一個柔性設計應該是什麼樣的，以及在設計中所採取的思考方式。

## 10.1 模式：INTENTION-REVEALING INTERFACES

在領域驅動的設計中，我們希望看到有意義的領域邏輯。如果代碼只是在執行規則後得到結果，而沒有把規則顯式地表達出來，那麼我們就不得不一步一步地去思考軟件的執行步驟。那些只是運行代碼然後給出結果的計算——沒有顯式地把計算邏輯表達出來，也有同樣的問題。如果不把代碼與模型清晰地聯繫起來，我們很難理解代碼的執行效果，也很難預測修改代碼的影響。前一章深入探討了對規則和計算進行顯式的建模。實現這樣的對象要求我們深入理解計算或規則的大量細節。對象的強大功能是它能夠把所有這些細節封裝起來，如此一來，客戶代碼就能夠很簡單，而且可以用高層概念來解釋。

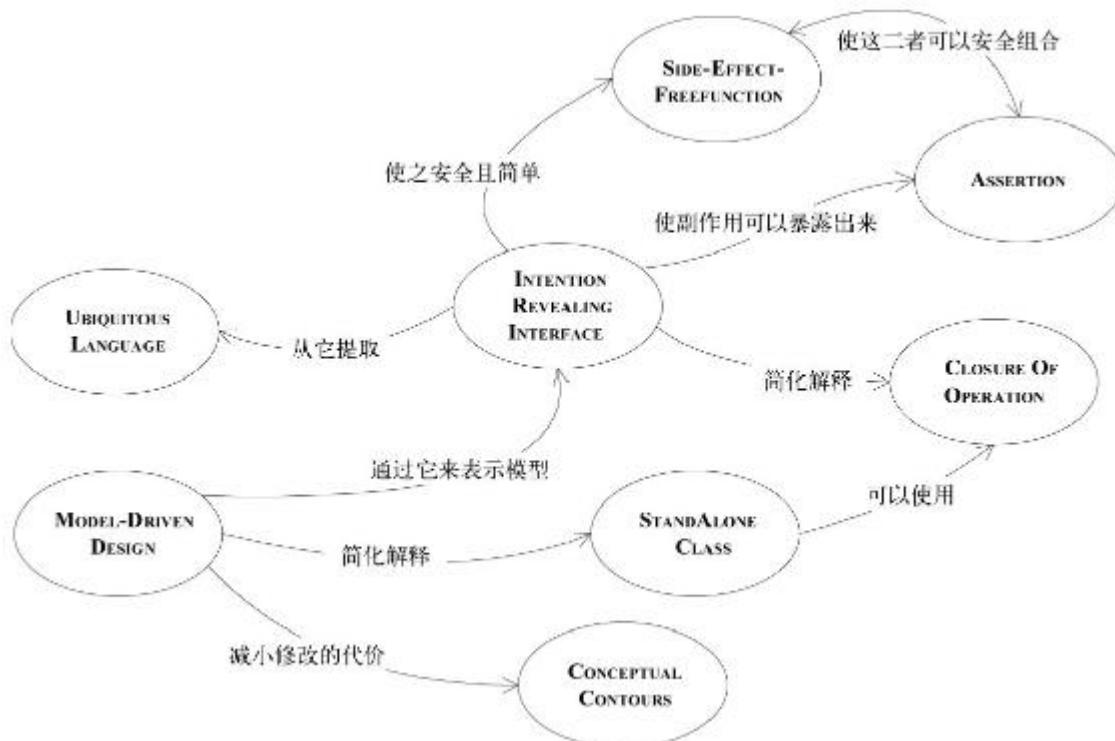


圖10-1 一些有助於獲得柔性設計的模式

但是，客戶開發人員要想有效地使用對象，必須知道對象的一些信息，如果接口沒有告訴開發人員這些信息，那麼他就必須深入研究對象的內部機制，以便理解細節。閱讀客戶代碼的人也需要做同樣的事情。這樣就失去了封裝的大部分價值。我們需要避免出現「認識過載」的問題。如果客戶開發人員必須總是思考組件工作方式的大量細節，那麼就無暇理清思路來解決客戶設計的複雜性。即便一個人同時扮演兩種角色（既開發代碼，也使用他自己的代碼）的時候也是如此，因為他即使不必去瞭解那些細節，也不可能一次就把所有的因素都考慮全面。

如果開發人員為了使用一個組件而必須要去研究它的實現，那麼就失去了封裝的價值。當某個人開發的對象或操作被別人使用時，如果使用這個組件的新的開發者不得不根據其實現來推測其用途，那麼他推測出來的可能並不是那個操作或類的主要用途。如果這不是那個組件的用途，雖然代碼暫時可以工作，但設計的概念基礎已經被誤用了，兩位開發人員的意圖也是背道而馳。

當我們把概念顯式地建模為類或方法時，為了真正從中獲取價值，必須為這些程序元素賦予一個能夠反映出其概念的名字。類和方法的名稱為開發人員之間的溝通創造了很好的機會，也能夠改善系統的抽象。

Kent Beck曾經提出通過INTENTION-REVEALING SELECTOR（釋意命名選擇器）來選擇方法的名稱，使名稱表達出其目的[Beck 1997]。設計中的所有公共元素共同構成了接口，每個元素的名稱都提供了揭示設計意圖的機會。類型名稱、方法名稱和參數名稱組合在一起，共同形成了一個INTENTION-REVEALING INTERFACE（釋意接口）。

因此：

在命名類和操作時要描述它們的效果和目的，而不要表露它們是通過何種方式達到目的的。這樣可以使客戶開發人員不必去理解內部細節。這些名稱應該與**UBIQUITOUS LANGUAGE**保持一致，以便團隊成員可以迅速推斷出它們的意義。在創建一個行為之前先為它編寫一個測試，這樣可以促使你站在客戶開發人員的角度上來思考它。

所有複雜的機制都應該封裝到抽象接口的後面，接口只表明意圖，而不表明方式。

在領域的公共接口中，可以把關係和規則表述出來，但不要說明規則是如何實施的；可以把事件和動作描述出來，但不要描述它們是如何執行的；可以給出方程式，但不要給出解方程式的數學方法。可以提出問題，但不要給出獲取答案的方法。

### 示例 重構：調漆應用程序

一家油漆商店的程序能夠為客戶顯示出標準調漆的結果。下面是初始的設計，它有一個簡單的領域類。

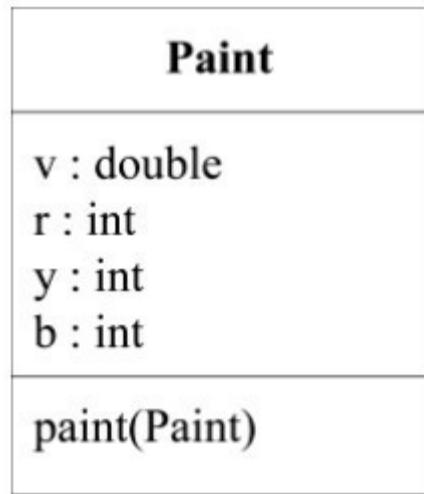


圖10-2

`paint(paint)`方法的行為根本猜不出，想知道它的唯一方法就是閱讀代碼。

```
public void paint(Paint paint) {  
    v = v + paint.getV(); //After mixing, volume is summed  
    // Omitted many lines of complicated color mixing logic  
    // ending with the assignment of new r, b, and y values.  
}
```

從代碼上看，這個方法是把兩種油漆（Paint）混合到一起，結果是油漆的體積增加了，並變為混合顏色。

為了換個角度來看問題，我們為這個方法編寫一個測試（這段代碼基於JUnit測試框架）。

```
public void testPaint() {  
    // Create a pure yellow paint with volume=100  
    Paint yellow = new Paint(100.0, 0, 50, 0);  
    // Create a pure blue paint with volume=100  
    Paint blue = new Paint(100.0, 0, 0, 50);  
  
    // Mix the blue into the yellow  
    yellow.paint(blue);  
  
    // Result should be volume of 200.0 of green paint  
    assertEquals(200.0, yellow.getV(), 0.01);  
    assertEquals(25, yellow.getB());  
    assertEquals(25, yellow.getY());  
    assertEquals(0, yellow.getR());  
}
```

通過這個測試只是一個起點，這無法令我們滿意，因為這段測試代碼並沒有告訴我們這個方法都做了什麼。讓我們來重新編寫這個測試，看一下如果我們正在編寫一個客戶應用程序的話，將以何種方式來使用Paint對象。最初，這個測試會失敗。實際上，它甚至不能編譯。我們編寫它的目的是從客戶開發人員的角度來研究一下Paint對象的接口設計。

```
public void testPaint() {
    // Start with a pure yellow paint with volume=100
    Paint ourPaint = new Paint(100.0, 0, 50, 0);
    // Take a pure blue paint with volume=100
    Paint blue = new Paint(100.0, 0, 0, 50);

    // Mix the blue into the yellow
    curPaint.mixIn(blue);

    // Result should be volume of 200.0 of green paint
    assertEquals(200.0, ourPaint.getVolume(), 0.01);
    assertEquals(25, ourPaint.getBlue());
    assertEquals(25, ourPaint.getYellow());
    assertEquals(0, ourPaint.getRed());
}
```

花時間編寫這樣的測試是非常必要的，因為它可以反映出我們希望以哪種方式與這些對像進行交互。在這之後，我們重構Paint類，使它通過測試，如圖10-3所示。

新的方法名稱可能不會告訴讀者有關混合另一種油漆（Paint）的效果的所有信息（要達到這個目的需要使用斷言，接下來我們就會討論它）。但這個名稱為讀者提供了足夠多的線索，使讀者可以開始使用這個類，特別是從測試提供的示例開始。而且它還使客戶代碼的閱讀者能夠理解客戶的意圖。在本章接下來的幾個示例中，我們將再次重構這個類，使它更清晰。

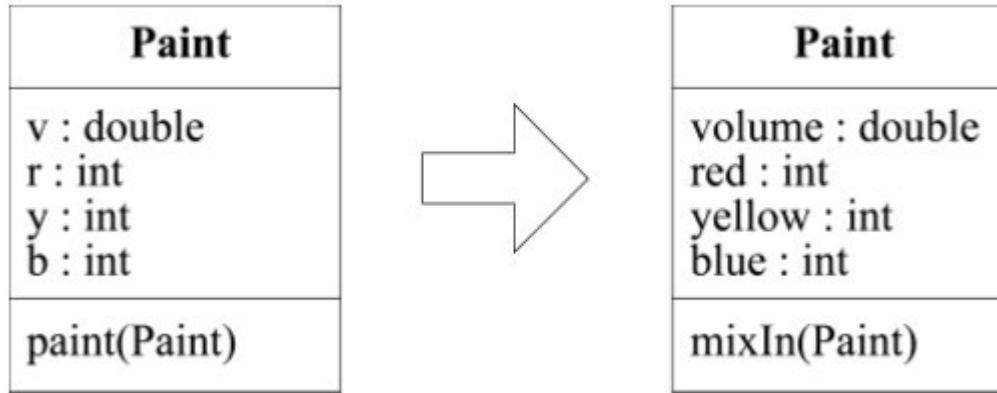


圖10-3

整個子領域可以被劃分到獨立的模塊中，並用一個表達了其用途的接口把它們封裝起來。這種方法可以使我們把注意力集中在項目上，並控制大型系統的複雜性，這些內容將在第15章中的COHESIVE MECHANISM和GENERIC SUBDOMAIN部分進行更多的討論。

在接下來的兩個模式中，我們將介紹如何令一個方法的執行結果變得易於預測。複雜的邏輯可以在SIDE-EFFECT-FREE FUNCTION中安全地執行，而改變系統狀態的方法可以用ASSERTION來刻畫。

## **10.2 模式：SIDE-EFFECT-FREE FUNCTION**

我們可以寬泛地把操作分為兩個大的類別：命令和查詢。查詢是從系統獲取信息，查詢的方式可能只是簡單地訪問變量中的數據，也可能是用這些數據執行計算。命令（也稱為修改器）是修改系統的操作（舉一個簡單的例子，設置變量）。在標準英語中，「副作用」這個詞暗示著「意外的結果」，但在計算機科學中，任何對系統狀態產生的影響都叫副作用。這裡為了便於討論，我們把它的含義縮小一下，任何對未來操作產生影響的系統狀態改變都可以稱為副作用。

為什麼人們會採用「副作用」這個詞來形容那些顯然是有意影響系統狀態的操作呢？我推測這大概是來自於複雜系統的經驗。大多數

操作都會調用其他的操作，而後者又會調用另外一些操作。一旦形成這種任意深度的嵌套，就很難預測調用一個操作將要產生的所有後果。第二層和第三層操作的影響可能並不是客戶開發人員有意為之的，於是它們就變成了完全意義上的副作用。在一個複雜的設計中，元素之間的交互同樣也會產生無法預料的結果。副作用這個詞強調了這種交互的不可避免性。

多個規則的相互作用或計算的組合所產生的結果是很難預測的。開發人員在調用一個操作時，為了預測操作的結果，必須理解它的實現以及它所調用的其他方法的實現。如果開發人員不得不「揭開接口的面紗」，那麼接口的抽象作用就受到了限制。如果沒有了可以安全地預見到結果的抽象，開發人員就必須限制「組合爆炸」[\[3\]](#)，這就限制了系統行為的豐富性。

返回結果而不產生副作用的操作稱為函數。一個函數可以被多次調用，每次調用都返回相同的值。一個函數可以調用其他函數，而不必擔心這種嵌套的深度。函數比那些有副作用的操作更易於測試。由於這些原因，使用函數可以降低風險。

顯然，在大多數軟件系統中，命令的使用都是不可避免的，但有兩種方法可以減少命令產生的問題。首先，可以把命令和查詢嚴格地放在不同的操作中。確保導致狀態改變的方法不返回領域數據，並盡可能保持簡單。在不引起任何可觀測到的副作用的方法中執行所有查詢和計算[\[Meyer 1988\]](#)。

第二，總是有一些替代的模型和設計，它們不要求對現有對象做任何修改。相反，它們創建並返回一個**VALUE OBJECT**，用於表示計算結果。這是一種很常見的技術，在接下來的示例中我們就會演示它的使用。**VALUE OBJECT**可以在一次查詢的響應中被創建和傳遞，然後被丟棄——不像**ENTITY**，實體的生命週期是受到嚴格管理的。

**VALUE OBJECT**是不可變的，這意味著除了在創建期間調用的初始化程序之外，它們的所有操作都是函數。像函數一樣，**VALUE OBJECT**使用起來很安全，測試也很簡單。如果一個操作把邏輯或計算與狀態改變混合在一起，那麼我們就應該把這個操作重構為兩個獨立的操作〔Fowler 1999,p.279〕。但從定義上來看，這種把副作用隔離到簡單的命令方法中的做法僅適用於**ENTITY**。在完成了修改和查詢的分離之後，可以考慮再進行一次重構，把複雜計算的職責轉移到**VALUE OBJECT**中。通過派生出一個**VALUE OBJECT**（而不是改變現有狀態），或者通過把職責完全轉移到一個**VALUE OBJECT**中，往往可以完全消除副作用。

因此：

盡可能把程序的邏輯放到函數中，因為函數是隻返回結果而不產生明顯副作用的操作。嚴格地把命令（引起明顯的狀態改變的方法）隔離到不返回領域信息的、非常簡單的操作中。當發現了一個非常適合承擔複雜邏輯職責的概念時，就可以把這個複雜邏輯移到**VALUE OBJECT**中，這樣可以進一步控制副作用。

**SIDE-EFFECT-FREE FUNCTION**，特別是在不變的**VALUE OBJECT**中，允許我們安全地對多個操作進行組合。當通過**INTENTION-REVEALING INTERFACE**把一個**FUNCTION**呈現出來的時候，開發人員就可以在無需理解其實現細節的情況下使用它。

### 示例 再次重構調漆應用程序

一家油漆商店的程序能夠為客戶顯示出標準調漆的結果。我們繼續前面的例子，下面是上次重構後得到的領域類：

```

public void mixIn(Paint other) {
    volume = volume.plus(other.getVolume());
    // Many lines of complicated color-mixing logic
    // ending with the assignment of new red, blue,
    // and yellow values.
}

```

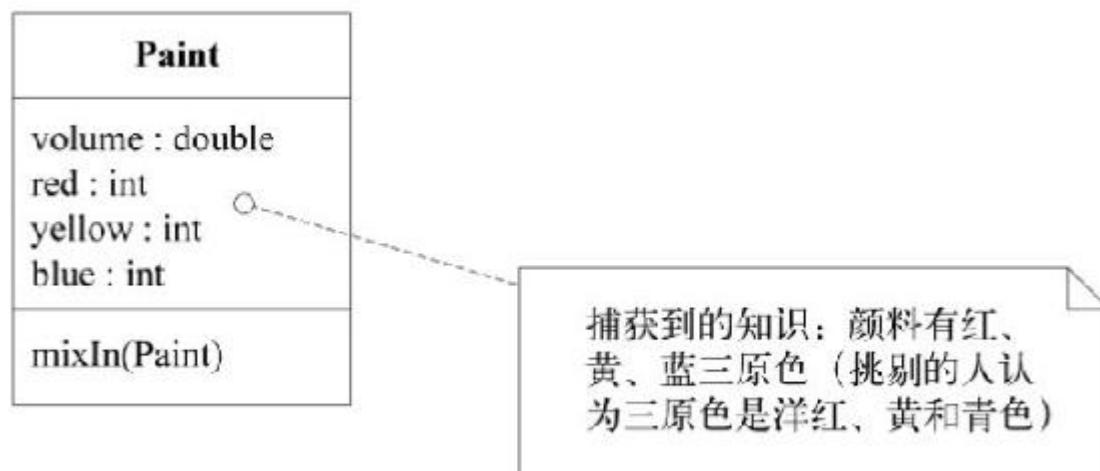


圖10-4

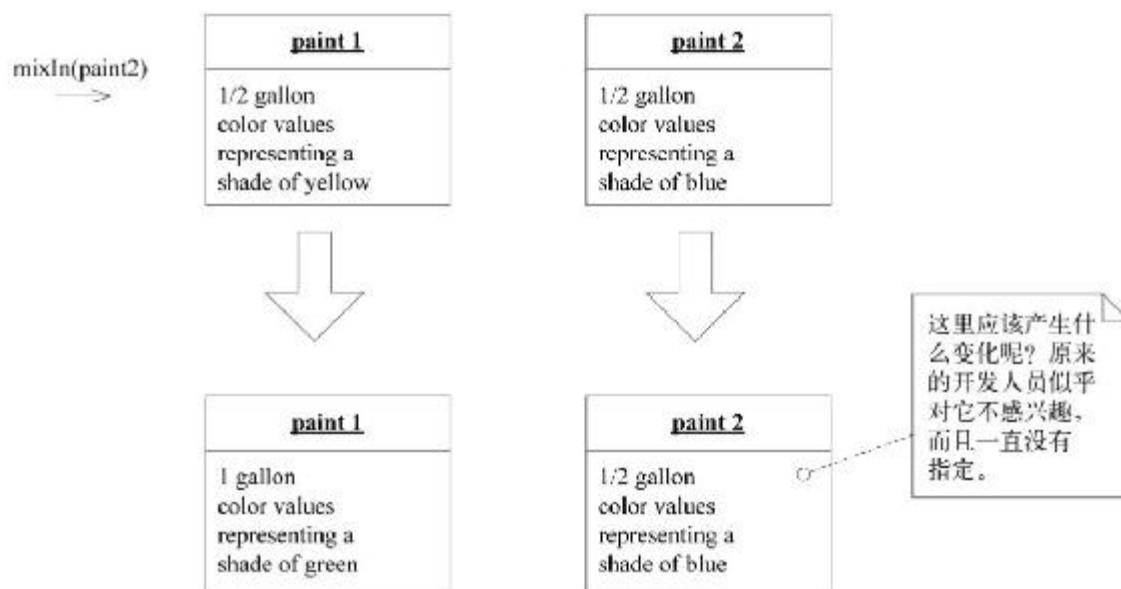


圖10-5 mixIn()方法的副作用

`mixIn()`方法中發生了很多事情，但這個設計確實遵循了「修改和查詢分離」這條原則。有一點需要注意（下面會具體討論），這裡並沒有對`paint` 2對像（`mixIn()`方法的一個參數）的體積做過多的考慮。操作不改變`Paint` 2的體積，在這個概念模型的上下文中，這看起來並不是十分合乎邏輯。就我們所知，這在原來的開發人員看來並不是問題，因為他們對操作之後的`paint` 2對像不感興趣，但我們很難預測副作用會產生什麼後果。在接下來要討論的**ASSERTION**中我們很快會回頭再討論這個問題。現在，我們先來看一下顏色。

在這個領域中，顏色是一個重要的概念。讓我們試著把它變成一個顯式的對象。它應該叫什麼名字呢？首先想到的就是`Color`（顏色），但我們通過先前的知識消化已經認識到了一個重要的知識，即油漆的調色與我們所熟悉的RGB調色是不同的。名稱必須反映出這一點。

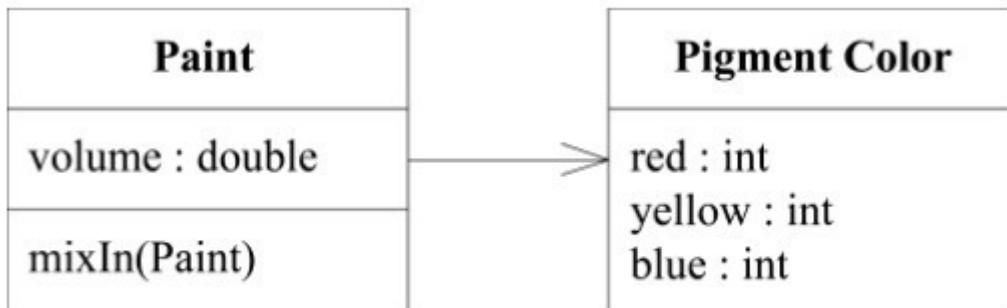


圖10-6

把`Pigment Color`（顏料顏色）分離出來之後，確實比先前表達了更多信息，但計算還是相同的，仍然是在`mixIn()`方法中進行計算。當把顏色數據移出來後，與這些數據有關的行為也應該一起移出來。但是在做這件事之前，要注意`Pigment Color`是一個**VALUE OBJECT**。因此，它應該是不可變的。當我們調漆時，`Paint`對像本身被改變了，它是一個具有生命週期的實體。相反，表示某個色調（如黃色）的

Pigment Color則一直表示那種顏色。調漆的結果是產生一個新的 Pigment Color對象，用於表示新的顏色。

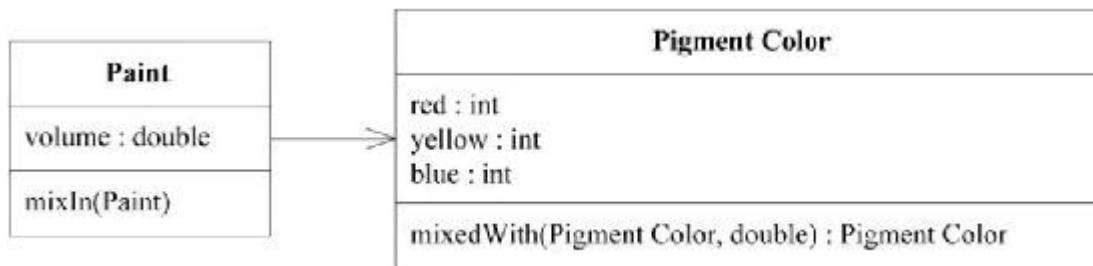


圖10-7

```
public class PigmentColor {  
  
    public PigmentColor mixedWith(PigmentColor other,  
                                  double ratio) {  
        // Many lines of complicated color-mixing logic  
        // ending with the creation of a new PigmentColor object  
        // with appropriate new red, blue, and yellow values.  
    }  
}  
  
public class Paint {  
  
    public void mixIn(Paint other) {  
        volume = volume + other.getVolume();  
        double ratio = other.getVolume() / volume;  
        pigmentColor =  
            pigmentColor.mixedWith(other.pigmentColor(), ratio);  
    }  
}
```

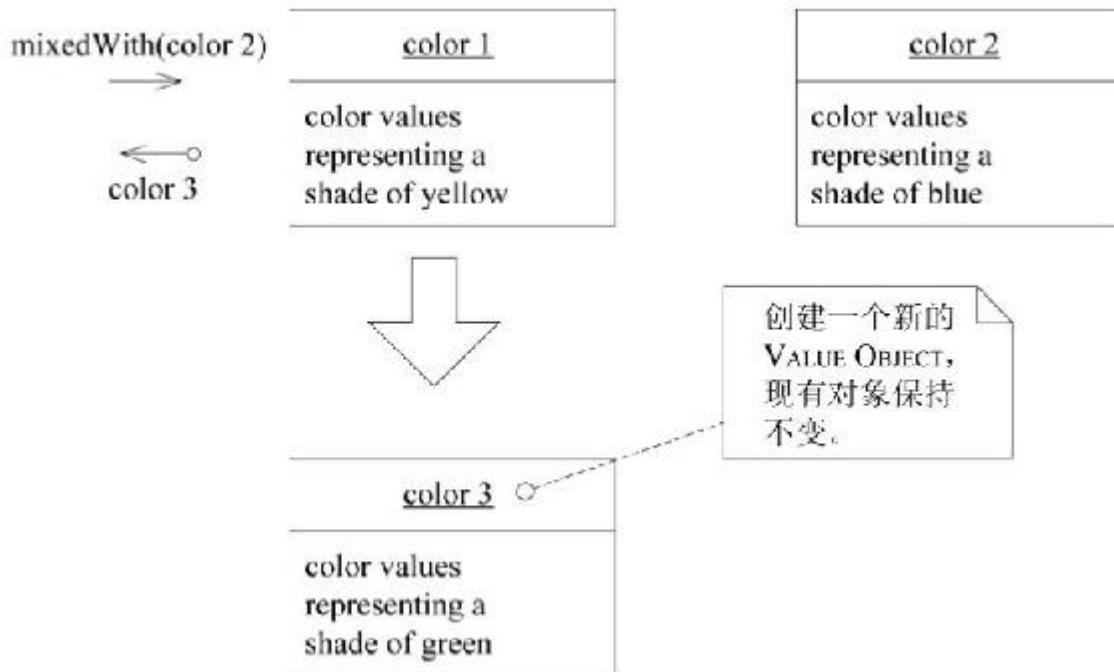


圖10-8

現在，Paint中的代碼已經盡可能簡單了。新的Pigment Color類捕獲了知識，並顯式地把這些知識表達出來，而且它還提供了一個SIDE-EFFECT-FREE FUNCTION，這個函數的計算結果很容易理解，也很容易測試，因此可以安全地使用或與其他操作進行組合。由於它的安全性很高，因此複雜的調色邏輯真正被封裝起來了。使用這個類的開發人員不必理解其實現。

### 10.3 模式：ASSERTION

把複雜的計算封裝到SIDE-EFFECT-FREE FUNCTION中可以簡化問題，但實體仍然會留有一些有副作用的命令，使用這些ENTITY的人必須瞭解使用這些命令的後果。在這種情況下，使用ASSERTION（斷言）可以把副作用明確地表示出來，使它們更易於處理。

確實，一條不包含複雜計算的命令只需查看一下就能夠理解。但是，在一個軟件設計中，如果較大的部分是由較小部分構成的，那麼一個命令可能會調用其他命令。開發人員在使用高層命令時，必須瞭解每個底層命令所產生的後果，這時封裝也就沒有什麼價值了。而且，由於對像接口並不會限制副作用，因此實現相同接口的兩個子類可能會產生不同的副作用。使用它們的開發人員需要知道哪個副作用是由哪個子類產生的，以便預測後果。這樣，抽像和多態也就失去了意義。

如果操作的副作用僅僅是由它們的實現隱式定義的，那麼在一個具有大量相互調用關係的系統中，起因和結果會變得一團糟。理解程序的唯一方式就是沿著分支路徑來跟蹤程序的執行。封裝完全失去了價值。跟蹤具體的執行也使抽像失去了意義。

我們需要在不深入研究內部機制的情況下理解設計元素的意義和執行操作的後果。**INTENTION-REVEALING INTERFACE**可以起到一部分作用，但這樣的接口只能非正式地給出操作的用途，這常常是不夠的。「契約式設計」(**design by contract**)向前推進了一步，通過給出類和方法的「斷言」使開發人員知道肯定會發生的結果。**[Meyer 1988]**中詳細討論了這種設計風格。簡言之，「後鉛條件」描述了一個操作的副作用，也就是調用一個方法之後必然會發生的結果。「前鉛條件」就像是合同條款，即為了滿足後鉛條件而必須要滿足的前鉛條件。類的固定規則規定了在操作結束時對象的狀態。也可以把**AGGREGATE**作為一個整體來為它聲明固定規則，這些都是嚴格定義的完整性規則。

所有這些斷言都描述了狀態，而不是過程，因此它們更易於分析。類的固定規則在描述類的意義方面起到幫助作用，並且使客戶開發人員能夠更準確地預測對象的行為，從而簡化他們的工作。如果你

確信後鉻條件的保證，那麼就不必考慮方法是如何工作的。斷言應該已經把調用其他操作的效果考慮在內了。

因此：

把操作的後置條件和類及**AGGREGATE**的固定規則表述清楚。如果在你的編程語言中不能直接編寫**ASSERTION**，那麼就把它們編寫成自動的單元測試。還可以把它們寫到文檔或圖中（如果符合項目開發風格的話）。

尋找在概念上內聚的模型，以便使開發人員更容易推斷出預期的**ASSERTION**，從而加快學習過程並避免代碼矛盾。

儘管很多面向對象的語言目前都不支持直接使用**ASSERTION**，但**ASSERTION**仍然不失為一種功能強大的設計方法。自動單元測試在一定程度上彌補了缺乏語言支持帶來的不足。由於**ASSERTION**只聲明狀態，而不聲明過程，因此很容易編寫測試。測試首先設置前鉻條件，在執行之後，再檢查後鉻條件是否被滿足。

把固定規則、前鉻條件和後鉻條件清楚地表述出來，這樣開發人員就能夠理解使用一個操作或對象的後果。從理論上講，如果一組斷言之間互不矛盾，那麼就可以發揮作用。但人的大腦並不會一絲不苟地把這些斷言編譯到一起。人們會推斷和補充模型的概念，因此找到一個既易於理解又滿足應用程序需求的模型是至關重要的。

### 示例 回到調漆應用程序

在前面的示例中，我們曾注意到：在**Paint**類中**mixIn(paint)**操作的參數到底會發生什麼變化，這還存在著一些不明之處。

接受者（即被混合的油漆）的所增加的體積就是參數的體積。根據我們對油漆的瞭解，這個混合過程應該使另一種油漆減少同樣的體積，把它的體積減為零或完全刪除。目前的實現並沒有修改這個參數，而修改參數無疑是有產生副作用的風險的。

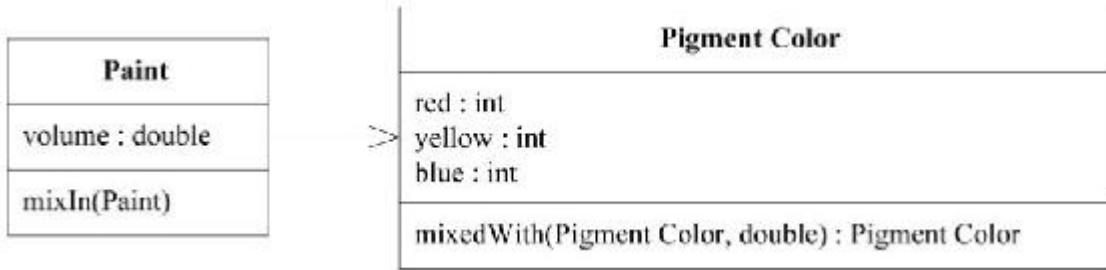


圖10-9

第一步，我們先把mixIn()方法的後鉻條件聲明如下：

在p1.mixIn(p2)之後：

p1.volume增加p2.volume的量

p2.volume不變

問題在於開發人員將會犯錯，因為這些屬性與實際概念不符。簡單的修改方法是讓另一種油漆的體積變為零。雖然修改參數不是一種好的行為，但這裡的修改簡單而直觀。我們可以聲明一個固定規則：

混合之後油漆的總體積保持不變。

但先等一下！當開發人員考慮這種選擇時，他們有了一個新發現。最初的設計人員這樣設計原來是有充分理由的。程序在最後會報告被混合之前的油漆清單。畢竟，這個程序的最終目的是幫助用戶弄清楚把哪幾種油漆混合到一起。

因此，如果要使體積模型的邏輯保持一致，那麼它就無法滿足這個應用程序的需求了。這看上去是一種進退兩難的境況。我們是否仍使用這個不合常理的後鉻條件，並為了彌補這個不足而清楚地說明這樣做的理由呢？世界上並不是一切事物都是直觀的，有時那就是最好的答案。但在這個例子中，這種尷尬局面似乎是由於丟失概念而造成的。讓我們去尋找一個新的模型。

**尋找更清晰的模型**

我們在尋找更好的模型的時候，會比原來的設計人員更有優勢，因為我們在研究的過程中消化了更多知識，而且通過重構得到了更深層的理解。例如，我們用一個**VALUE OBJECT**上的**SIDE-EFFECT-FREE FUNCTION**來計算顏色。這意味著可以在任何需要的時候重複進行這個計算。我們應該利用這種優勢。

我們似乎為**Paint**分配了兩種不同的基本職責。讓我們試著把它們分開。

現在只有一個命令，即**mixIn()**。從對模型的直觀理解可以看出，它只是把一個對像加入到一個集合中。所有其他操作都是**SIDE-EFFECT-FREE FUNCTION**。

下面的測試方法（使用了**JUnit**測試框架）用來確認圖10-10中列出的一個**ASSERTION**是否滿足：

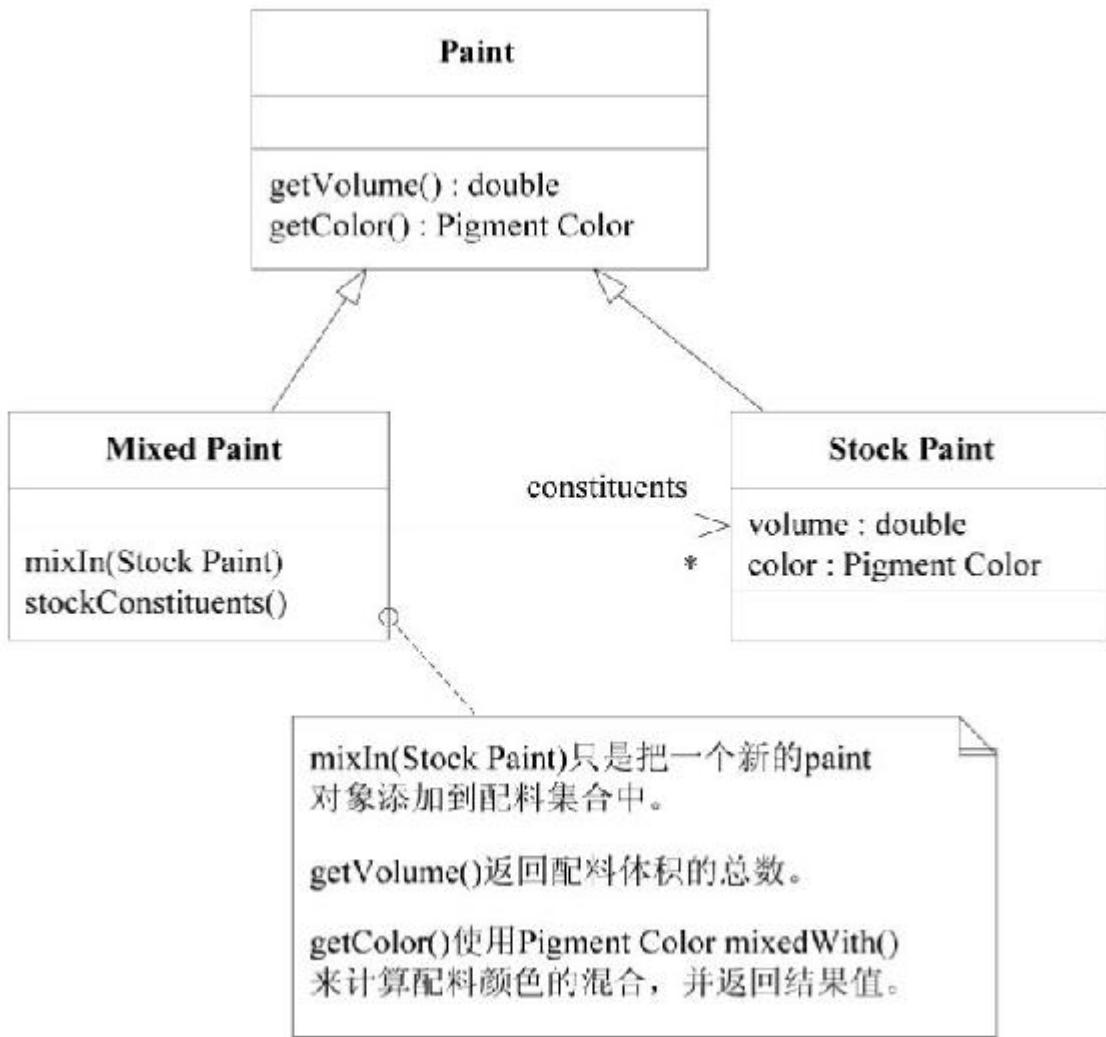


圖10-10

```
public void testMixingVolume {  
    PigmentColor yellow = new PigmentColor(0, 50, 0);  
    PigmentColor blue = new PigmentColor(0, 0, 50);  
  
    StockPaint paint1 = new StockPaint(1.0, yellow);  
    StockPaint paint2 = new StockPaint(1.5, blue);  
    MixedPaint mix = new MixedPaint();  
  
    mix.mixIn(paint1);  
    mix.mixIn(paint2);  
    assertEquals(2.5, mix.getVolume(), 0.01);  
}
```

這個模型捕捉並傳遞了更多領域知識。固定規則和後路條件符合常識，這使得它們更易於維護和使用。

**INTENTION-REVEALING INTERFACE**清楚地表明瞭用途，**SIDE-EFFECT-FREE FUNCTION**和**ASSERTION**使我們能夠更準確地預測結果，因此封裝和抽象更加安全。

可重組元素的下一個因素是有效的分解.....

## **10.4 模式：CONCEPTUAL CONTOUR**

有時，人們會對功能進行更細的分解，以便靈活地組合它們，有時卻要把功能合成大塊，以便封裝複雜性。有時，人們為了使所有類和操作都具有相似的規模而尋找一種一致的粒度。這些方法都過於簡單了，並不能作為通用的規則。但使用這些方法的動機都來自於一系列基本的問題。

**如果把模型或設計的所有元素都放在一個整體的大結構中，那麼它們的功能就會發生重複。外部接口無法給出客戶可能關心的全部信**

息。由於不同的概念被混合在一起，它們的意義變得很難理解。

而另一方面，把類和方法分解開也可能是毫無意義的，這會使客戶更複雜，迫使客戶對像去理解各個細微部分是如何組合在一起的。更糟的是，有的概念可能會完全丟失。鈾原子的一半並不是鈾。而且，粒度的大小並不是唯一要考慮的問題，我們還要考慮粒度是在哪種場合下使用的。

菜譜式的規則是沒有用的。但大部分領域都深深隱含著某種邏輯一致性，否則它們就形不成領域了。這並不是說領域就是絕對一致的，而且人們討論領域的方式肯定也不一樣。但是領域中一定存在著某種十分複雜的原理，否則建模也就失去了意義。由於這種隱藏在底層的一致性，當我們找到一個模型，它與領域的某個部分特別吻合時，這個模型很可能也會與我們後續發現的這個領域的其他部分一致。有時，新的發現可能與模型不符，在這種情況下，就需要對模型進行重構，以便獲取更深層的理解，並希望下一次新發現能與模型一致。

通過反覆重構最終會實現柔性設計，以上就是其中的一個原因。隨著代碼不斷適應新理解的概念或需求，CONCEPTUAL CONTOUR（概念輪廓）也就逐漸形成了。

從單個方法的設計，到類和MODULE的設計，再到大型結構的設計（參見第16章），高內聚低耦合這一對基本原則都起著重要的作用。這兩條原則既適用於代碼，也適用於概念。為了避免機械化地遵循它，我們必須經常根據我們對領域的直觀認識來調整技術思路。在做每個決定時，都要問自己：「這是根據當前模型和代碼中的特定關係做出的權宜之計呢，還是反映了底層領域的某種輪廓？」

尋找在概念上有意義的功能單元，這樣可以使得設計既靈活又易懂。例如，如果領域中對兩個對象的「相加」（addition）是一個連貫

的整體操作，那麼就把它作為整體來實現。不要把 `add()` 拆分成兩個步驟。不要在同一個操作中進行下一個步驟。從稍大的範圍來看，每個對象都應該是一個獨立的、完整的概念，也就是一個「WHOLE VALUE」（整體值）[4]。

出於同樣的原因，在任何領域中，都有一些細節是用戶不感興趣的。前面假想的那個調漆應用程序的用戶不會添加紅色顏料或藍色顏料，他們只是把已經做好的油漆拿來調，而油漆包含所有3種顏料。把那些沒必要分解或重組的元素作為一個整體，這樣可以避免混亂，並且使人們更容易看到那些真正需要重組的元素。如果用戶的物理設備允許加入顏料，那麼領域就改變了，而且我們可能需要分別對每種顏料進行控制。專門研究油漆的化學家將需要更精細的控制，這就需要進行完全不同的分析了，有可能會產生一個比我們的調漆應用程序中的顏料顏色更精細的油漆構成模型。但是這些與我們的調漆應用程序項目中的任何人都無關。

因此：

把設計元素（操作、接口、類和**AGGREGATE**）分解為內聚的單元，在這個過程中，你對領域中一切重要劃分的直觀認識也要考慮在內。在連續的重構過程中觀察發生變化和保證穩定的規律性，並尋找能夠解釋這些變化模式的底層**CONCEPTUAL CONTOUR**。使模型與領域中那些一致的方面（正是這些方面使得領域成為一個有用的知識體系）相匹配。

我們的目標是得到一組可以在邏輯上組合起來的簡單接口，使我們可以用**UBIQUITOUS LANGUAGE**進行合理的表述，並且使那些無關的選項不會分散我們的注意力，也不增加維護負擔。但這通常是通過重構才能得到的結果，很難在前期就實現。而且如果僅僅是從技術角

度進行重構，可能永遠也不會出現這種結果；只有通過重構得到更深層的理解，才能實現這樣的目標。

設計即使是按照**CONCEPTUAL CONTOUR**進行，也仍然需要修改和重構。當連續的重構往往只是做出一些局部修改（而不是對模型的概念產生大範圍的影響）時，這就是模型已經與領域相吻合的信號。如果遇到了一個需求，它要求我們必須大幅度地修改對像和方法的劃分，那麼這就在向我們傳遞這樣一條信息：我們對領域的理解還需要精化。它提供了一個深化模型並且使設計變得更具柔性的機會。

### 示例 應計項目的**CONCEPTUAL CONTOUR**

在第9章中，基於對會計概念的更深層理解，我們對一個貸款跟蹤系統進行了重構，如圖10-11所示。

新模型比原來的模型只多出一個對象，但職責的劃分卻發生了很大的變化。

**Schedule**原來是在**Calculator**類中通過邏輯判斷計算的，現在被分散到不同的類中，用於不同類型的手續費和利息計算。另一方面，手續費和利息的支付原來是分開的，現在也被合併到一起了。

由於新發現的顯式概念與領域非常吻合，而且**Accrual Schedule**的層次結構具有內聚性，因此開發人員認為這個模型更符合領域的**CONCEPTUAL CONTOUR**，如圖10-12所示。

新的**Accrual Schedule**的加入是開發人員早就預料到的，因為有一些需求早已等待它來處理了。這樣，她選擇的模型除了使現有功能更清晰、簡單之外，還很容易引入新的**Schedule**。但是，她是否找到了一個**CONCEPTUAL CONTOUR**，使得領域設計可以隨著應用程序和業務的演變而改變和發展呢？我們無法確定一個設計如何處理意料之外的改變，但她認為她的設計中一些不合適的地方已經有所改進了。

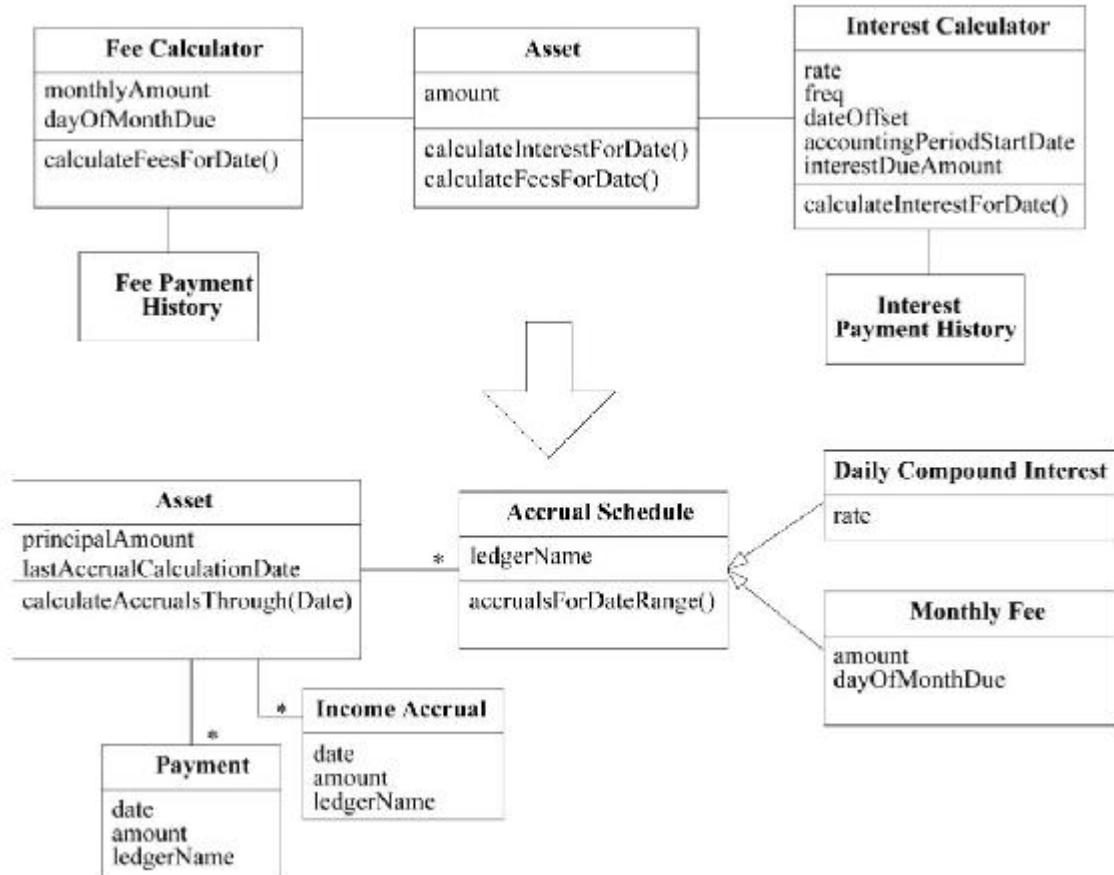


圖10-11

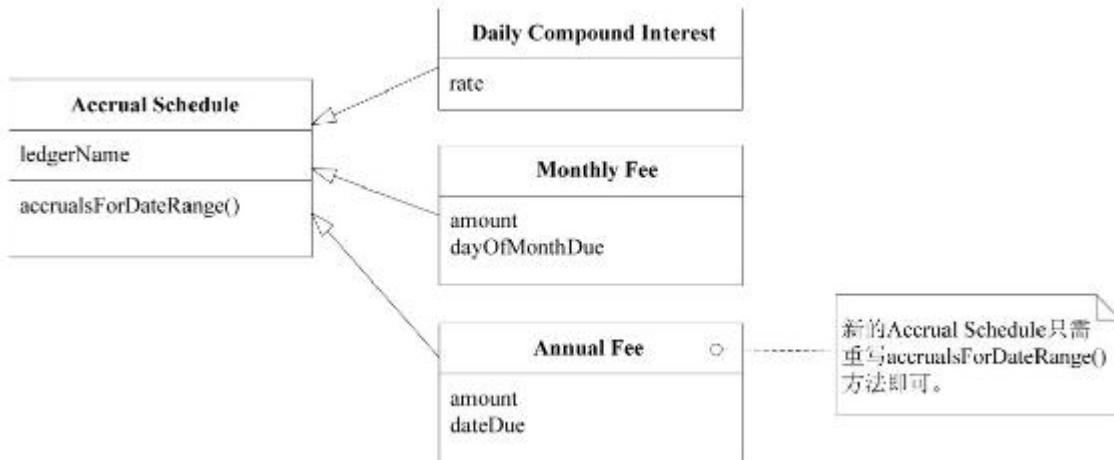


圖10-12 這個模型把新的Accrual Schedule添加進來了  
一個未預料到的改變

隨著項目向前進展，又出現了一個新的需求——需要制定一些詳細的規則來處理提早付款和延遲付款。這位開發人員在研究問題的時候，很高興地發現利息付款和手續費付款實際上使用相同的規則。這意味著新的模型元素可以很自然地使用Payment類。

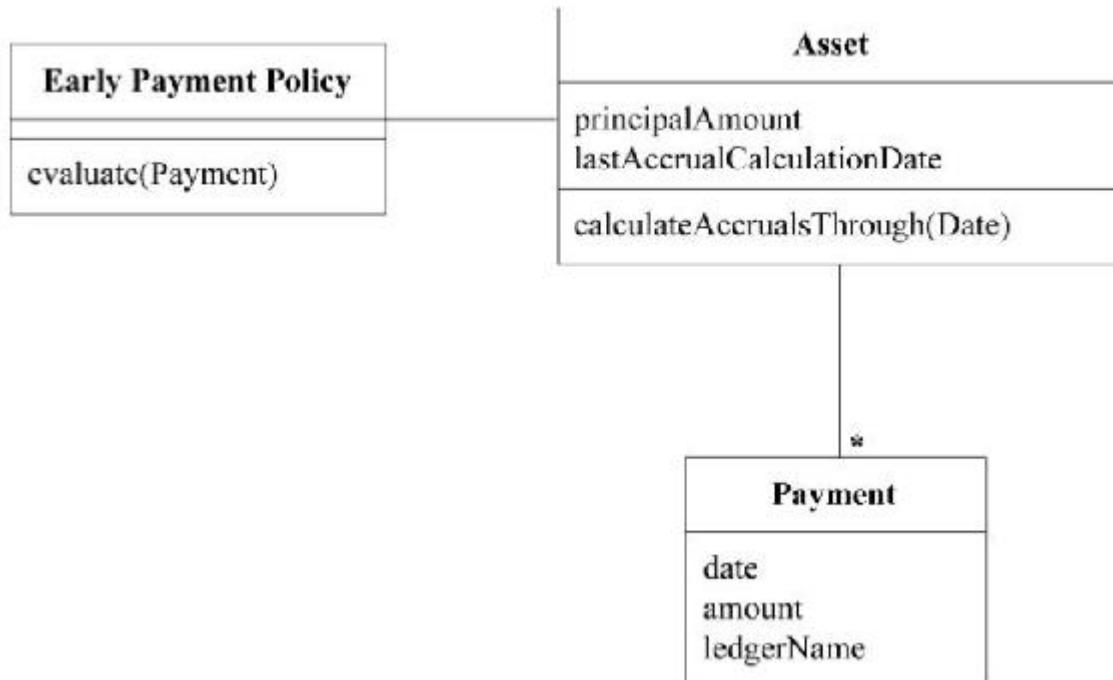


圖10-13

原有的設計導致兩個Payment History類之間必然出現重複（這個難題可能使得開發人員認識到Payment類應該被共享，這樣就會從另外一條途徑得到類似的模型）。新元素之所以很容易就添加進來了，並不是因為她預料到了這個改變，也不是因為她的設計靈活到了足以容納任何可能修改的程度。真正的原因是經過前面的重構，設計能夠很好地與領域的基本概念相契合。

**INTENTION-REVEALING INTERFACE**使客戶能夠把對像表示為有意義的單元，而不僅僅是一些機制。**SIDE-EFFECT-FREE FUNCTION**和**ASSERTION**使我們可以安全地使用這些單元，並對它們進行複雜的組

合。CONCEPTUAL CONTOUR的出現使模型的各個部分變得更穩定，也使得這些單元更直觀，更易於使用和組合。

然而，我們仍然會遇到「概念過載」（conceptual overload）的問題——當模型中的互相依賴過多時，我們就必須把大量問題放在一起考慮。

## **10.5 模式：STANDALONE CLASS**

互相依賴使模型和設計變得難以理解、測試和維護。而且，互相依賴很容易越積越多。

當然，每個關聯都是一種依賴，要想理解一個類，必須理解它與哪些對像有聯繫。與這個類有聯繫的其他對像還會與更多的對象發生聯繫，而這些聯繫也是必須要弄清楚的。每個方法的每個參數的類型也是一個依賴，每個返回值也都是一個依賴。

如果有一個依賴關係，我們必須同時考慮兩個類以及它們之間關係的本質。如果某個類依賴另外兩個類，我們就必須考慮這3個類當中的每一個、這個類與其他兩個類之間的相互關係的本質，以及這3個類可能存在的其他相互關係。如果它們之間依次存在依賴關係，那麼我們還必須考慮這些關係。如果一個類有3個依賴關係……問題就會像滾雪球一樣越來越多。

MODULE和AGGREGATE的目的都是為了限制互相依賴的關係網。當我們識別出一個高度內聚的子領域並把它提取到一個MODULE中的時候，一組對象也隨之與系統的其他部分解除了聯繫，這樣就把互相聯繫的概念的數量控制在一個有限的範圍之內。但是，即使把系統分成了各個MODULE，如果不嚴格控制MODULE內部的依賴的話，那麼MODULE也一樣會讓我們耗費很多精力去考慮依賴關係。

即使是在**MODULE**內部，設計也會隨著依賴關係的增加而變得越來越難以理解。這加重了我們的思考負擔，從而限制了開發人員能處理的設計複雜度。隱式概念比顯式引用增加的負擔更大。

我們可以將模型一直精煉下去，直到每個剩下的概念關係都表示出概念的基本含義為止。在一個重要的子集中，依賴關係的個數可以減小到零，這樣就得到一個完全獨立的類，它只有很少的幾個基本類型和基礎庫概念。

在每種編程環境中，都有一些非常基本的概念，它們經常用到，以至於已經根植於我們的大腦中。例如，在**Java**開發環境中，基本類型和一些標準類庫提供了數字、字符串和集合等基本概念。從實際來講，「整數」這個概念是不會增加思考負擔的。除此之外，為了理解一個對像而必須保留在大腦中的其他概念都會增加思考負擔。

隱式概念，無論是否已被識別出來，都與顯式引用一樣會加重思考負擔。雖然我們通常可以忽略像整數和字符串這樣的基本類型值，但無法忽略它們所表示的意義。例如，在第一個調漆應用程序的例子中，**Paint**對像包含3個公共的整數，分別表示紅、黃、藍3種顏色值。**Pigment Color**對象的創建並沒有增加所涉及的概念數量，也沒有增加依賴關係。但它確實使現有概念更明晰、更易於理解了。另一方面，**Collection**的**size()**操作返回一個整數（只是一個簡單的合計數），它只表示整數的基本含義，因此並不產生隱式的新概念。

我們應該對每個依賴關係提出質疑，直到證實它確實表示對象的基本概念為止。這個仔細檢查依賴關係的過程從提取模型概念本身開始。然後需要注意每個獨立的關聯和操作。仔細選擇模型和設計能夠大幅減少依賴關係——常常能減少到零。

**低耦合**是對像設計的一個基本要素。盡一切可能保持低耦合。把其他所有無關概念提取到對像之外。這樣類就變得完全獨立了，這就

使得我們可以單獨地研究和理解它。每個這樣的獨立類都極大地減輕了因理解**MODULE**而帶來的負擔。

當一個類與它所在的模塊中的其他類存在依賴關係時，比它與模塊外部的類有依賴關係要好得多。同樣，當兩個對像具有自然的緊密耦合關係時，這兩個對像共同涉及的多個操作實際上能夠把它們的關係本質明確地表示出來。我們的目標不是消除所有依賴，而是消除所有不重要的依賴。當無法消除所有的依賴關係時，每清除一個依賴對開發人員而言都是一種解脫，使他們能夠集中精力處理剩下的概念依賴關係。

盡力把最複雜的計算提取到**STANDALONE CLASS**（獨立的類）中，實現此目的的一種方法是從存在大量依賴的類中將**VALUE OBJECT**建模出來。

從根本上講，油漆的概念與顏色的概念緊密相關。但在考慮顏色（甚至是顏料）的時候卻與不必去考慮油漆。通過把這兩個概念變為顯式概念並精煉它們的關係，所得到的單向關聯就可以表達出重要的信息，同時我們可以對**Pigment Color**類（大部分計算複雜性都隱藏在這個類中）進行獨立的分析和測試。

低耦合是減少概念過載的最基本辦法。獨立的類是低耦合的極致。

消除依賴性並不是說要武斷地把模型中的一切都簡化為基本類型，這樣只會削弱模型的表達能力。本章要討論的最後一個模式**CLOSURE OF OPERATION**（閉合操作）就是一種在減小依賴性的同時保持豐富接口的技術。

## **10.6 模式：CLOSURE OF OPERATION**

兩個實數相乘，結果仍為實數（實數是所有有理數和所有無理數的集合）。由於這一點永遠成立，因此我們說實數的「乘法運算是閉合的」：乘法運算的結果永遠無法脫離實數這個集合。當我們對集合中的任意兩個元素組合時，結果仍在這個集合中，這就叫做閉合操作。

—The Math Forum,Drexel University

當然，依賴是必然存在的，當依賴是概念的一個基本屬性時，它就不是壞事。如果把接口精簡到只處理一些基本類型，那麼會極大地削弱接口的能力。但我們也經常為接口引入很多不必要的依賴，甚至是整個不必要的概念。

大部分引起我們興趣的對象所產生的行為僅用基本類型是無法描述的。

另一種對設計進行精化的常見方法就是我所說的**CLOSURE OF OPERATION**（閉合操作）。這個名字來源於最精煉的概念體系，即數學。 $1 + 1 = 2$ 。加法運算是實數集中的閉合運算。數學家們都極力避免去引入無關的概念，而閉合運算的性質正好為他們提供了這樣一種方式，可用來定義一種不涉及其他任何概念的運算。我們都非常熟悉數學中的精煉，因此很難注意到一些小技巧會有多麼強大。但是，這些技巧在軟件設計中也廣為應用。例如，**XSLT**的基本用法是把一個**XML**文檔轉換為另一個**XML**文檔。這種**XSLT**操作就是**XML**文檔集合中的閉合操作。閉合的性質極大地簡化了對操作的理解，而且閉合操作的鏈接或組合也很容易理解。

因此：

在適當的情況下，在定義操作時讓它的返回類型與其參數的類型相同。如果實現者（**implementer**）的狀態在計算中會被用到，那麼實現者實際上就是操作的一個參數，因此參數和返回值應該與實現者有相同的類型。這樣的操作就是在該類型的實例集合中的閉合操作。

閉合操作提供了一個高層接口，同時又不會引入對其他概念的任何依賴。

這種模式更常用於**VALUE OBJECT**的操作。由於**ENTITY**的生命週期在領域中十分重要，因此我們不能為瞭解決某一問題而草率創建一個**ENTITY**。有一些操作是**ENTITY**類型之下的閉合操作。我們可以通過查詢一個Employee（員工）對像來返回其主管，而返回的將是另一個Employee對象。但是，**ENTITY**通常不會成為計算結果。因此，大部分閉合操作都應該到**VALUE OBJECT**中去尋找。

一個操作可能是在某一抽象類型之下的閉合操作，在這種情況下，具體的參數可能是不同的具體類型。例如，加法是實數之下的閉合運算，而實數既可以是有理數，也可以是無理數。

在嘗試和尋找減少互相依賴並提高內聚的過程中，有時我們會遇到「半個閉合操作」這種情況。參數類型與實現者的類型一致，但返回類型不同；或者返回類型與接收者（*receiver*）的類型相同但參數類型不同。這些操作都不是閉合操作，但它們確實具有**CLOSURE OF OPERATION**的某些優點。當沒有形成閉合操作的那個多出來的類型是基本類型或基礎庫類時，它幾乎與**CLOSURE OF OPERATION**一樣減輕了我們的思考負擔。

在前面的示例中，**Pigment Color**的**mixedWith()**操作是**Pigment Color**之下的閉合操作，本書中還零星地穿插著幾個這樣的示例。以下示例顯示了即使在沒有達到真正**CLOSURE OF OPERATION**的時候，這種思想也發揮了強大的作用。

### 示例 從集合中選擇子集

在Java中，如果想從**Collection**（集合）中選擇一個元素子集，需要使用**Iterator**（迭代器）。用迭代器遍歷這些元素，測試每個元素，把匹配的元素收集到一個新的**Collection**中。

```
Set employees = (some Set of Employee objects);
Set lowPaidEmployees = new HashSet();
Iterator it = employees.iterator();
while (it.hasNext()) {
    Employee anEmployee = it.next();
    if (anEmployee.salary() < 40000)
        lowPaidEmployees.add(anEmployee);
}
```

從概念上講，上段代碼只是從集合中選擇了一個子集。是否真的有必要使用**Iterator**這個額外的概念以及它所帶來的所有機制上的複雜性呢？如果是使用**Smalltalk**，我將在**Collection**上調用「**select**」操作，把測試作為參數傳遞給它。返回值將是一個新的**Collection**，其中只包含通過測試的那些元素。

```
employees := (some Set of Employee objects).

lowPaidEmployees := employees select:
    [:anEmployee | anEmployee salary < 40000]
```

**Smalltalk**的**Collection**還提供了其他一些這樣的函數，它們返回新生成的**Collection**（可能是幾種不同的具體類）。這些操作並不是閉合操作，因為它們把一個**block**（塊）作為參數。但**block**在**Smalltalk**中是一個基礎庫類型，因此它們並不會增加開發人員的思考負擔。由於返回值與實現者的類型相匹配，因此它們可以像一系列過濾器一樣被串接在一起。讀寫代碼都變得很容易。它們並沒有引入與選擇子集無關的外來概念。

本章介紹的模式演示了一個通用的設計風格和一種思考設計的方式。把軟件設計得意圖明顯、容易預測且富有表達力，可以有效地發揮抽象和封裝的作用。我們可以對模型進行分解，使得對像更易於理解和使用，同時仍具有功能豐富的、高級的接口。

運用這些技術需要掌握相當高級的設計技巧，甚至有時編寫客戶端代碼也需要掌握高級技巧才能運用這些技術。**MODEL-DRIVEN DESIGN**的作用受細節設計的質量和實現決策的質量影響很大，而且只要有少數幾個開發人員沒有弄清楚它們，整個項目就會偏離目標。

儘管如此，團隊只要願意培養這些建模和設計技巧，那麼按照這些模式的思考方式就能夠開發出可以反覆重構的軟件，從而最終創建出非常複雜的軟件。

## 10.7 聲明式設計

使用**ASSERTION**可以得到更好的設計，雖然我們只是用一些相對非正式的方式來檢查這些**ASSERTION**。但實際上我們無法保證手寫軟件的正確性。舉個簡單例子，只要代碼還有其他一些沒有被**ASSERTION**專門排除在外的副作用，斷言就失去了作用。無論我們的設計多麼遵守**MODEL-DRIVEN**開發方法，最後仍要通過編寫過程代碼來實現概念交互的結果。而且我們花費了大量時間來編寫樣板代碼，但是這些代碼實際上不增加任何意義或行為。這些代碼冗長乏味而且易出錯，此外還掩蓋了模型的意義（雖然有的編程語言會相對好一些，但都需要我們做大量繁瑣的工作）。本章介紹的**INTENTION-REVEALING INTERFACE**和其他模式雖然有一定的幫助作用，但它們永遠也不會使傳統的面向對像技術達到非常嚴密的程度。

以上這些正是採用聲明式設計的部分動機。聲明式設計對於不同的人來說具有不同的意義，但通常是指一種編程方式—把程序或程序的一部分寫成一種可執行的規格（specification）。使用聲明式設計時，軟件實際上是由一些非常精確的屬性描述來控制的。聲明式設計有多種實現方式，例如，可以通過反射機制來實現，或在編譯時通過代碼生成來實現（根據聲明來自動生成傳統代碼）。這種方法使其他開發人員能夠根據字面意義來使用聲明。它是一種絕對的保證。

從模型屬性的聲明來生成可運行的程序是 **MODEL-DRIVEN DESIGN** 的理想目標，但在實踐中這種方法也有自己的缺陷。例如，下面就是我多次遇到的兩個具體問題：

聲明式語言並不足以表達一切所需的東西，它把軟件束縛在一個由自動部分構成的框架之內，使軟件很難擴展到這個框架之外。

代碼生成技術破壞了迭代循環——它把生成的代碼合併到手寫的代碼中，使得代碼重新生成具有巨大的破壞作用。

許多聲明式設計的嘗試帶來了意想不到的後果，由於開發人員受到框架侷限性的約束，為了交付工作只能先處理重要問題，而擱置其他一些問題，這導致模型和應用程序的質量嚴重下降。

基於規則的編程（帶有推理引擎和規則庫）是另一種有望實現的聲明式設計方法。但遺憾的是，一些微妙的問題會影響它的實現。

儘管基於規則的程序原則上是聲明式的，但大多數系統都有一些用於性能優化的「控制謂詞」（control predicate）。這種控制代碼引入了副作用，這樣行為就不再完全由聲明式規則來控制了。添加、刪除規則或重新排序可能導致預料不到的錯誤結果。因此，編寫邏輯的程序員必須確保代碼的效果是顯而易見的，就像對像程序員所做的那樣。

很多聲明式方法被開發人員有意或無意忽略之後會遭到破壞。當系統很難使用或限制過多時，就會發生這種情況。為了獲得聲明式程序的好處，每個人都必須遵守框架的規則。

據我所知，聲明式設計發揮的最大價值是用一個範圍非常窄的框架來自動處理設計中某個特別單調且易出錯的方面，如持久化和對像關係映射。最好的聲明式設計能夠使開發人員不必去做那些單調乏味的工作，同時又完全不限制他們的設計自由。

### 領域特定語言

領域特定語言是一種有趣的方法，它有時也是一種聲明式語言。採用這種編碼風格時，客戶代碼是用一種專門為特定領域的特定模型定製的語言編寫的。例如，運輸系統的語言可能包括cargo（貨物）和route（路線）這樣的術語，以及一些用於組合這些術語的語法。然後，程序通常會被編譯成傳統的面向對像語言，由一個類庫為這些術語提供實現。

在這樣的語言中，程序可能具有極強的表達能力，並且與UBIQUITOUS LANGUAGE之間形成最緊密的結合。領域特定語言是一個令人振奮的概念，但就我所見，在基於面向對像技術進行實現時，這種語言也存在自身的缺陷。

為了精化模型，開發人員需要修改語言。這可能涉及修改語法聲明和其他語言解釋功能，以及修改底層類庫。雖然我對學習高級技術和設計概念是完全贊同的，但我們必須冷靜地評估團隊當前的技術水平，以及將來維護團隊可能的技術水平。此外，用同一種語言實現的應用程序和模型之間是「無縫」的，這一點很有價值。另一個缺點是當模型被修改時，很難對客戶代碼進行重構，使之與修改之後的模型及與其相關的領域特定語言保持一致。當然，也許有人可以通過技術方法來解決重構問題。

## 一種完全不同的語言

有一種不同的範式能夠比對像更好地實現領域特定語言。在 Scheme 編程語言中（它是「函數式編程」家族的一個代表），有些部分非常類似於標準的編程風格，因此既具有領域特定語言的表達能力，又不會造成系統的分裂。

這種技術也許能在非常成熟的模型中發揮出最大的作用，在這種情況下，客戶代碼可能是由不同的團隊編寫的。但一般情況下，這樣的設置會產生有害的結果——團隊被分成兩部分，框架由那些技術水平較高的人來構建，而應用程序則由那些技術水平較差的人來構建了，但也並不是非得如此。

## 10.8 聲明式設計風格

一旦你的設計中有了 INTENTION-REVEALING INTERFACE、SIDE-EFFECT-FREE FUNCTION 和 ASSERTION，那麼你就具備了使用聲明式設計的條件。當我們有了可以組合在一起來表達意義的元素，並且使其作用具體化或明朗化，甚或是完全沒有明顯的副作用，我們就可以獲得聲明式設計的很多益處。

柔性設計使得客戶代碼可以使用聲明式的設計風格。為了說明這一點，下一節將會把本章介紹的一些模式結合起來使用，從而使 SPECIFICATION 更靈活，更符合聲明式設計的風格。

### 用聲明式的風格來擴展**SPECIFICATION**

第9章介紹了SPECIFICATION的基本概念、它在程序中扮演的角色，以及它在實現中的意義。現在，讓我們來看看幾個額外的、有吸

引力的技巧，它們在規則很複雜的情況下可能非常有用。

**SPECIFICATION**是由「謂詞」( predicate )這個眾所周知的形式化概念演變來的。謂詞還有其他一些有用的特性，我們可以對這些特性進行有選擇的利用。

### 使用邏輯運算對**SPECIFICATION**進行組合

當使用**SPECIFICATION**時，我們很容易就會遇到需要把它們組合起來使用的情況。正如我們剛剛提到的那樣，**SPECIFICATION**是謂詞的一個例子，而謂詞可以用「AND」、「OR」和「NOT」等運算進行組合和修改。這些邏輯運算都是謂詞這個類別之下的閉合操作，因此**SPECIFICATION**組合也是CLOSURE OF OPERATION。

隨著**SPECIFICATION**的通用性逐漸提高，創建一個可用於各種類型的**SPECIFICATION**的抽象類或接口會變得很有用。這需要把參數類型定義為某種高層的抽象類。

```
public interface Specification {  
    boolean isSatisfiedBy(Object candidate);  
}
```

這個抽象要求在方法的開始處放罷一條衛語句 ( guard clause )，但是沒有衛語句也不影響它的功能。例如，可以對 Container Specification ( 參見圖9-16以及後面的相關表格、代碼等 ) 做如下修改：

```
public class ContainerSpecification implements Specification {
    private ContainerFeature requiredFeature;

    public ContainerSpecification(ContainerFeature required) {
        requiredFeature = required;
    }

    boolean isSatisfiedBy(Object candidate) {
        if (!candidate instanceof Container) return false;

        return
(Container)candidate.getFeatures().contains(requiredFeature);
    }
}
```

現在，讓我們擴展Specification接口，加入3個新操作：

```
public interface Specification {
    boolean isSatisfiedBy(Object candidate);

    Specification and(Specification other);
    Specification or(Specification other);
    Specification not();
}
```

回憶一下，有些Container Specification需要通風性的Container（容器），而有些則需要有防爆性。如果一種化學藥品既易揮發又易爆炸，那麼它可能同時需要這兩種規格。如果使用新的方法，這就很容易實現。

```
Specification ventilated = new ContainerSpecification(VENTILATED);
Specification armored = new ContainerSpecification(ARMORED);

Specification both = ventilated.and(armored);
```

這段聲明定義了一個具有期望屬性的新的Specification對象。這種組合將需要一個用於某種特殊目的的、更複雜的Container Specification。

假設我們有多種通風容器。對於有些物品來說，把它們放進哪種容器中都沒問題。它們可以放在任何一種通風容器中。

```
Specification ventilatedType1 =  
    new ContainerSpecification(VENTILATED_TYPE_1);  
Specification ventilatedType2 =  
    new ContainerSpecification(VENTILATED_TYPE_2);  
  
Specification either = ventilatedType1.or(ventilatedType2);
```

如果我們認為把砂存放在特殊容器中是一種浪費，那麼可以通過指定一種沒有特殊性質的「便宜的」容器來禁止把砂存放在特殊容器中。

```
Specification cheap = (ventilated.not()).and(armored.not());
```

這個約束將阻止第9章中所討論的倉庫打包程序原型的某些不優化的行為。

從簡單元素構建複雜規格的能力提高了代碼的表達能力。以上組合是以聲明式的風格編寫的。

由於SPECIFICATION實現的方法存在不同，提供這些運算符的難易程度也不同。下面是一個非常簡單的實現，在有些情況下它的效率很差，而有些情況下則很實用。舉這個例子只是為了起到說明的作用。像任何模式一樣，它也有很多實現方式。

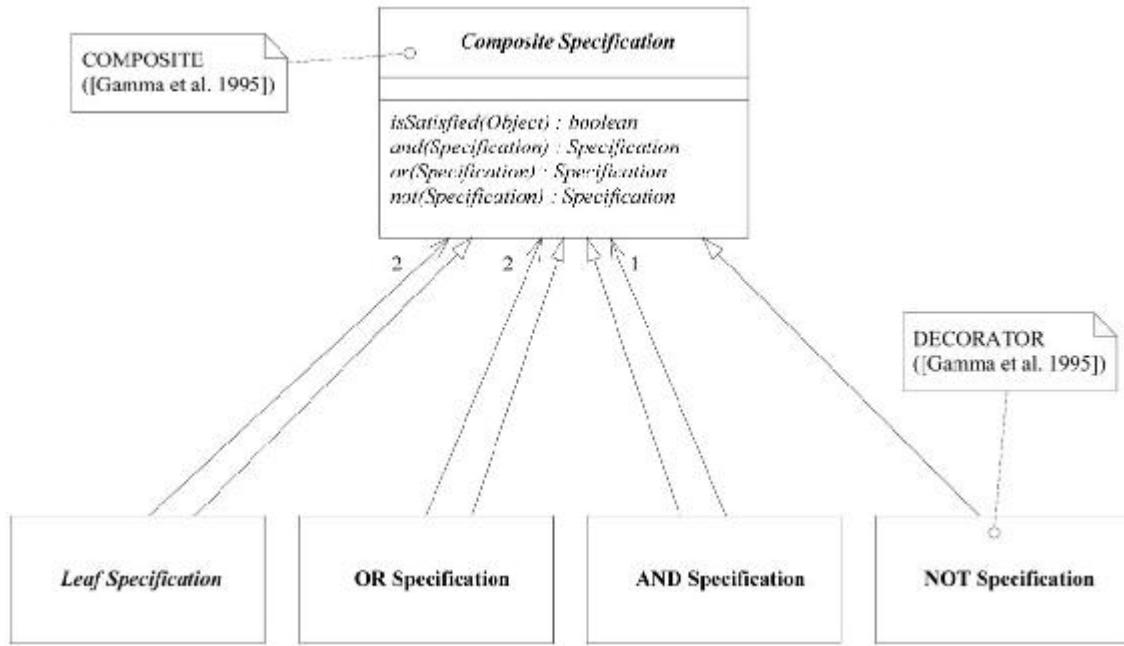


圖10-14 SPECIFICATION的COMPOSITE ( 組合 ) 設計

```

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);
    }
}

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);
    }
}

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);
    }
}

public class NotSpecification extends AbstractSpecification {
    Specification wrapped;
    public NotSpecification(Specification x) {
        wrapped = x;
    }
    public boolean isSatisfiedBy(Object candidate) {
        return !wrapped.isSatisfiedBy(candidate);
    }
}

```

為了便於閱讀，上面這段代碼寫得盡可能簡單。如前所述，它在有些情況下是低效的。可能會有一些其他的實現選擇，使得對象的數目減至最少，或極大地提高速度，或者與某個項目的特定技術兼容。

重要的是模型捕捉到領域的關鍵概念，同時有一個忠實於該模型的實現。這就為解決性能問題預留了很大的空間。

此外，這樣完全的通用性在很多情況下並不需要。特別是AND可能比其他運算用得更多，而且它的實現的複雜程度也較小。如果你只需要AND，那麼完全可以只實現它，這沒有什麼可擔心的。

我們回顧一下第2章示例中的對話，開發人員顯然沒有實現他們SPECIFICATION中的「satisfied by」行為。在他們進行那段討論的時候，SPECIFICATION只是「根據需要來構建」（building to order）。儘管如此，抽像仍然完整，而且功能添加起來也相對簡單。使用模式並不意味著構建你不需要的特性。它們可以過後再添加，只要不引起概念混淆即可。

### 示例 **COMPOSITE SPECIFICATION**的另一種實現

有些實現環境不能使用粒度很小的對象。我曾經遇到過一個項目，它有一個對像數據庫，這個數據庫為每個對象分配一個ID並跟蹤這個ID。每個對象都佔有很大的內存空間，並且產生很大的性能開銷，因此總的地址空間成為一個限制因素。我在領域設計中的一些重要地方使用了SPECIFICATION，當時我認為這是一個很好的決定。但我使用了一個過於細緻的實現（像本章中描述的那樣），這無疑是個錯誤。它產生了數百萬個粒度非常小的對象，使整個系統的速度變得非常緩慢。

下面的例子給出了一種替代實現，它把組合SPECIFICATION編碼為一個字符串或者數組（這個數組對邏輯表達式進行了編碼），然後在運行時進行解析。

（即使你沒明白它的實現也不要緊，重要的是認識到用邏輯運算符來實現SPECIFICATION的方式有很多。如果最簡單的方法不適用於你的情況，可以選擇其他的方法。）

## 「Cheap Container」的SPECIFICATION樣的內容

樣項	AndSpecificationOperator (FLY WEIGHT) NotSpecificationOperator (FLY WEIGHT) Armored NotSpecificationOperator Ventilated
----	-------------------------------------------------------------------------------------------------------------------------------------

當我們想測試一種候選方案時，必須解釋這個結構，這可以通過把每個元素彈出來並計算它（或者是根據運算符的需要彈出下一個元素）來實現。最後將得到如下結果：

```
and(not(armored), not(ventilated))
```

這種設計有一些優點（+）和缺點（-）

- + 對像個數較少
- + 內存使用效率高

需要更高級的開發人員

你必須根據自己的實際情況做出權衡，找到一種適合你的實現。基於相同的模式和模型可以創建出完全不同的實現。

包容

最後要講的這個包容特性並不是經常需要，而且實現起來也很難，但有時它確實能夠解決很困難的問題。它還能夠表達出一個SPECIFICATION的含義。

再次考慮一下前面的化學倉庫打包程序的例子。每個Chemical都有一個Container Specification，而且Packer SERVICE確保當把Drum分配到Container中時，所有這些Container Specification都被滿足，一切都沒有問題……直到有人改變了規則。

每隔幾個月都會發佈一組新的規則，我們的用戶希望能夠生成一個列表，把那些已經有了更嚴格要求的化學品列出來。

當然，通過運行一個驗證，用新實施的規格來檢查倉庫中的每個Drum，並找到所有不再滿足新SPECIFICATION的化學品，這樣可以把

一部分化學品列出來，而且這可能也是用戶需要的。這可以告訴用戶現在倉庫中有哪些Drum是需要轉移的。

但用戶要求的是把所有那些存放要求變得更嚴格的化學品都列出來。或許倉庫裡目前還沒有這樣的化學品，或者它們碰巧被裝到了一個更嚴格的容器中。無論是哪種情況，剛才的那個報告都不會列出它們。

我們引入一個用於直接比較兩種SPECIFICATION的新操作：

```
boolean subsumes (Specification other);
```

更嚴格的SPECIFICATION包容不太嚴格的SPECIFICATION。用更嚴格的SPECIFICATION來取代不嚴格的SPECIFICATION不會遺漏掉先前的任何需求，如圖10-15所示。

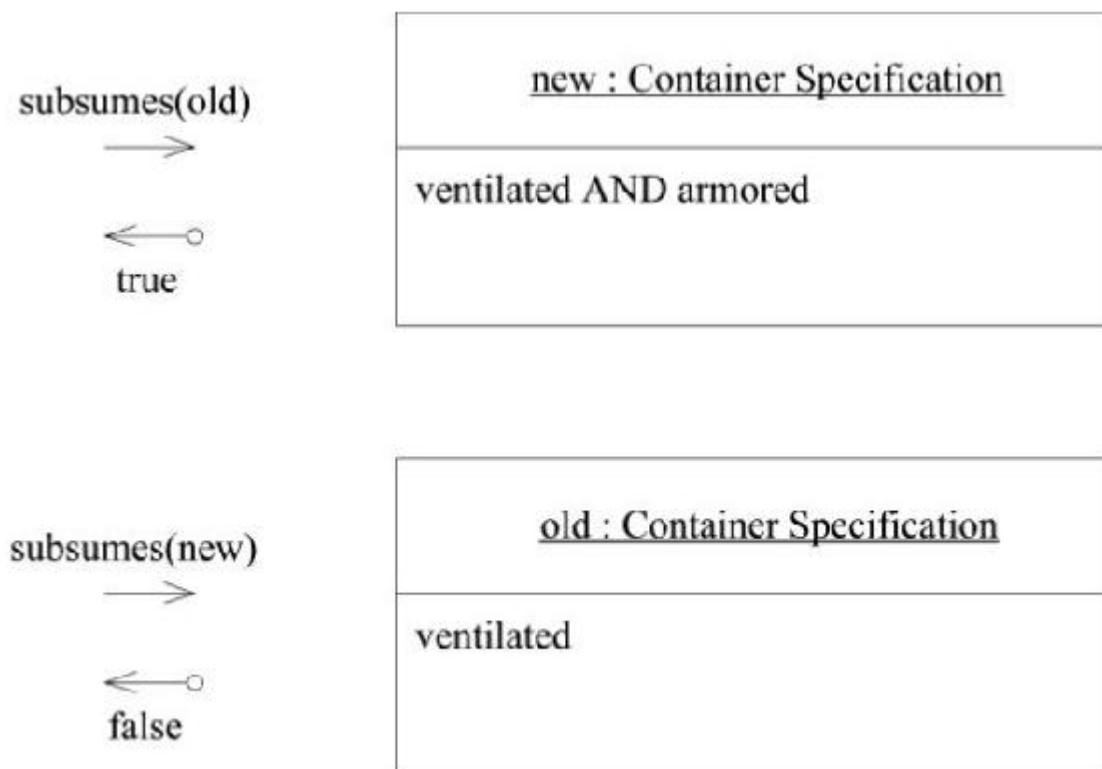


圖10-15 汽油容器的SPECIFICATION變嚴格了

在SPECIFICATION語言中，我們說新的SPECIFICATION包容舊的SPECIFICATION，因為任何滿足新SPECIFICATION的對象都將滿足舊SPECIFICATION。

如果把每個SPECIFICATION看成一個謂詞，那麼包容就等於邏輯蘊涵（logical implication）。使用傳統的符號， $A \rightarrow B$ 表示聲明A蘊涵聲明B，因此，如果A為真，則B也為真。

讓我們把這個邏輯應用於我們的容器匹配需求。當一個SPECIFICATION被修改時，我們想知道新SPECIFICATION是否滿足舊SPECIFICATION的所有條件。

### New Spec → Old Spec

也就是說，如果新規格為真，那麼舊規格一定也為真。要證明一般情況下的邏輯蘊涵是很難的，但特殊情況就很容易證明。例如，參數化的SPECIFICATION可以定義它們自己的包容規則。

```
public class MinimumAgeSpecification {
    int threshold;

    public boolean isSatisfiedBy(Person candidate) {
        return candidate.getAge() >= threshold;
    }

    public boolean subsumes(MinimumAgeSpecification other) {
        return threshold >= other.getThreshold();
    }
}
```

JUnit測試可能包含以下代碼：

```
drivingAge = new MinimumAgeSpecification(16);
votingAge = new MinimumAgeSpecification(18);
assertTrue(votingAge.subsumes(drivingAge));
```

還有一個有用的特例適用於解決Container Specification問題，它用SPECIFICATION接口把包容與邏輯操作AND結合起來。

```
public interface Specification {  
    boolean isSatisfiedBy(Object candidate);  
    Specification and(Specification other);  
    boolean subsumes(Specification other);  
}
```

證明只有一個AND操作符的涵蓋是簡單的：

AANDB→A

或者在更複雜的情況中，

AANDBANDC→AANDB

這樣，如果Composite Specification能夠把所有由「AND」連接起來的葉節點（leaf）SPECIFICATION收集到一起，那麼我們要做的事情只是檢查包容規格（subsuming SPECIFICATION）是否含有被包容規格的所有葉節點（而且它可能還包含更多的葉節點）——它的葉節點集合是另一個SPECIFICATION的葉節點集合的超集。

```

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;
}

```

我們還可以增強這種交互，對仔細選擇的參數化的葉節點 **SPECIFICATION** 進行比較或者進行其他一些複雜的比較。遺憾的是，當把 **OR** 和 **NOT** 也包括進來時，這些證明會變得更複雜。在大多數情況下，最好避免出現這樣的複雜性：要麼選擇放棄一些運算符，要麼不使用包容。如果這二者同時需要，那麼要慎重考慮這樣做的價值是否多過它所帶來的麻煩。

### 受 **SPECIFICATION** 約束的亞裡士多德

所有人都是要死的	Specification manSpec = new ManSpecification(); Specification mortalSpec = new MortalSpecification(); assert manSpec.subsumes(mortalSpec);
亞里士多德是人	man aristotle = new Man(); assert manSpec.isSatisfiedBy(aristotle);
因此，亞里士多德會死	assert mortalSpec.isSatisfiedBy(aristotle);

## 10.9 切入問題的角度

本章展示了一系列技術，它們用於澄清代碼意圖，使得使用代碼的影響變得顯而易見，並且解除模型元素的耦合。儘管有這些技術，

但要想實現這樣的設計還是很難的。我們不能只是看著一個龐大的系統說：「讓我們把它設計得靈活點吧。」我們必須選擇具體的目標。下面介紹幾種主要方法，然後給出一個擴展的示例，它展示瞭如何把這些模式結合起來使用，並用於處理更大的設計。

### **10.9.1 分割子領域**

我們無法一下子就能處理好整個設計，而需要一步一步地進行。我們從系統的某些方面可以看出適合用哪種方法處理，那麼就把它們提取出來加以處理。如果模型的某個部分可以被看作是專門的數學，那麼可以把這部分分離出來。如果應用程序實施了某些用來限制狀態改變的複雜規則，那麼可以把這部分提取到一個單獨的模型中，或者提取到一個允許聲明規則的簡單框架中。隨著這些步驟的進行，不僅新模型更整潔了，而且剩下的部分也更小、更清晰了。在剩下的模型中，有的部分是用聲明式的風格來編寫的——這些可能是根據專門數學或驗證框架編寫的聲明，或者是子領域所採用的任何形式。

重點突擊某個部分，使設計的一個部分真正變得靈活起來，這比分散精力泛泛地處理整個系統要有用得多。第15章將更深入地討論如何選擇和管理子領域。

### **10.9.2 盡可能利用已有的形式**

我們不能把從頭創建一個嚴密的概念框架當作一項日常的工作來做。在項目的生命週期中，我們有時會發現並精煉出這樣一個框架。但更常見的情況是，可以對你的領域或其他領域中那些建立已久的概念系統加以修改和利用，其中有些系統已經被精化和提煉達幾個世紀之久。例如，很多商業應用程序涉及會計學。會計學定義了一組成熟的ENTITY和規則，我們很容易對這些ENTITY和規則進行調整，得到一個深層的模型和柔性設計。

有很多這樣的正式概念框架，而我個人最喜歡的框架是數學。數學的強大功能令人驚奇，它可以用基本數學概念把一些複雜的問題提取出來。很多領域都涉及數學，我們要尋找這樣的部分，並把它挖掘出來。專門的數學很整齊，可以通過清晰的規則進行組合，並很容易理解。下面我要舉一個例子，用它來結束本章，它來自我過去的經歷——它就是「股份數學」（Shares Math）。

### 示例 把各種模式結合起來使用：股份數學

第8章講述了在銀團貸款系統項目上發生的一次模型突破的故事。現在我們將更詳細地討論這個例子，這裡我們只集中討論設計的一個特性，並與原來項目上的特性進行比較。

該應用程序的一個需求是，當借款者償付本金時，默認是根據放貸方的股份來分配這筆錢。

### 最初的付款分配設計

隨著我們對它進行重構，這段代碼會變得越來越容易理解，因此不必過度深究這個版本。

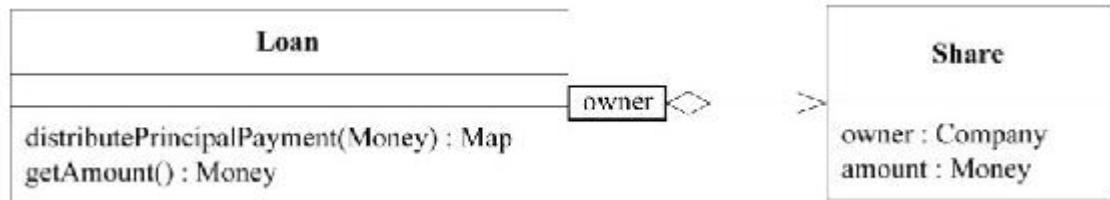


圖10-16

```

public class Loan {
    private Map shares;

    //Accessors, constructors, and very simple methods are excluded

    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();
            double initialLoanShareAmount = getShareAmount(owner);
            double paymentShareAmount =
                initialLoanShareAmount / total * paymentAmount;
            Share paymentShare =
                new Share(owner, paymentShareAmount);
            paymentShares.put(owner, paymentShare);

            double newLoanShareAmount =
                initialLoanShareAmount - paymentShareAmount;
            Share newLoanShare =
                new Share(owner, newLoanShareAmount);
            loanShares.put(owner, newLoanShare);
        }
        return paymentShares;
    }

    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;
    }
}

```

把命令和**SIDE-EFFECT-FREE FUNCTION**分開

這個設計已經有了INTENTION-REVEALING INTERFACE。但 `distributePaymentPrincipal()` 方法做了一件很危險的事情。它計算要分配的股份，並且還修改了Loan。我們通過重構把查詢從修改操作中分離出來。

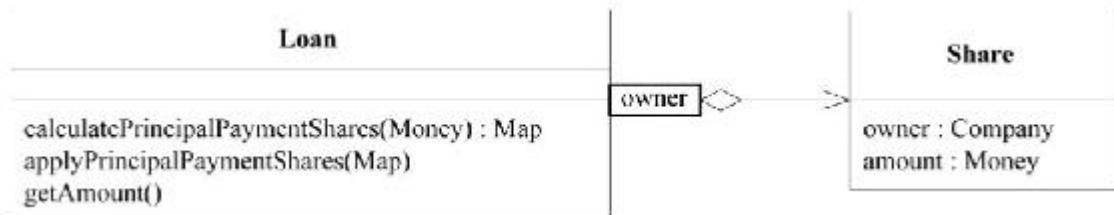


圖10-17

```

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);
    }
}

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;
}
  
```

客戶代碼現在如下：

```
Map<String, Double> distribution =  
    aLoan.calculatePrincipalPaymentShares(paymentAmount);  
    aLoan.applyPrincipalPaymentShares(distribution);
```

這段代碼不算太差。方法把大量的複雜性封裝在INTENTION-REVEALING INTERFACE背後。但當我們添加`applyDrawdown()`，`calculateFeeoaymentShares()`等一些函數之後，代碼開始大量增加。每次擴充都使代碼變得更複雜，速度也不斷減慢。這可能是由於粒度過大造成的。傳統的解決方法是把計算方法分解為子例程。這可能是一種不錯的解決辦法，但我們希望最終看到底層的概念邊界，並深化模型。當設計元素具有這種CONCEPT-CONTOURING的粒度時，就可以把這些元素進行組合，得到所需的變體。

### 把隱式概念變為顯式概念

現在我們有足夠的條件來探索新模型了。在這個實現中，`Share`對象是被動的，它們是用一些複雜、底層的方式來操縱的。這是因為大部分與股份有關的規則和計算並不適用於單獨的股份，而是用於成組的股份。有一個概念被漏掉了：各個股份在構成整體時互相之間是有關聯的。如果能把這個概念顯式地表達出來，就能更簡潔地表示這些規則和計算。

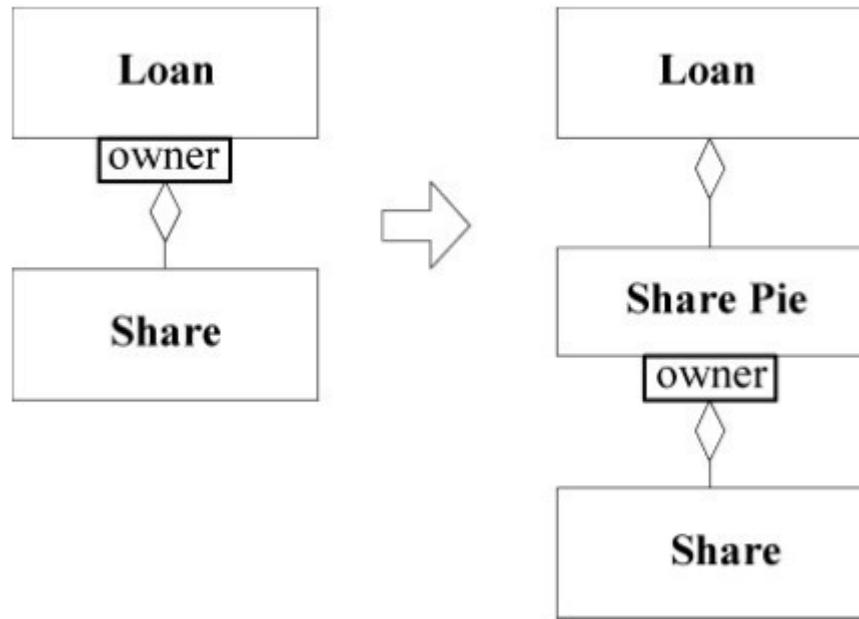


圖10-18

Share Pie 表示了一個特定的 Loan 的總體分佈。它是一個 ENTITY，其標識位於 Loan AGGREGATE 的內部。實際的分佈計算可以被委託給 Share Pie 。

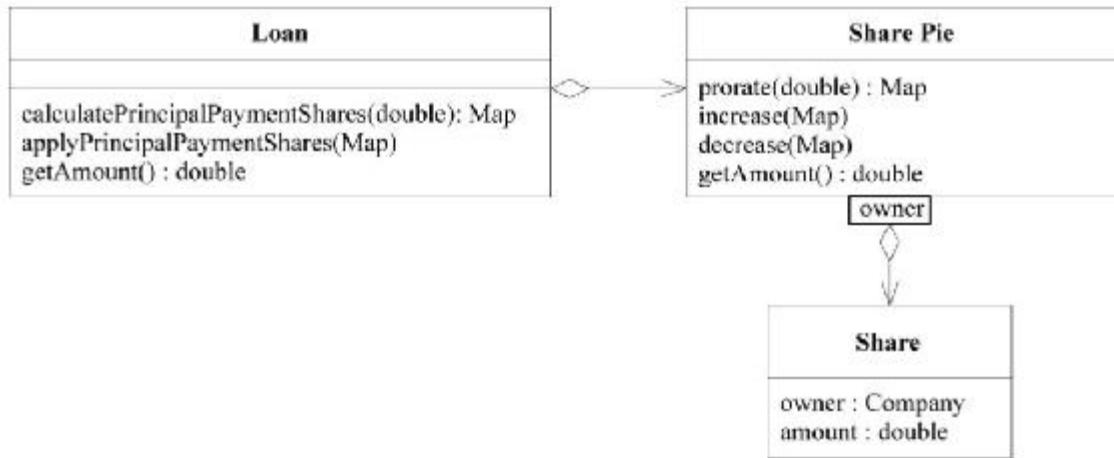


圖10-19

```
public class Loan {
    private SharePie shares;

    //Accessors, constructors, and straightforward methods
    //are omitted

    public Map calculatePrincipalPaymentDistribution(
        double paymentAmount) {
        return getShares().prorated(paymentAmount);
    }

    public void applyPrincipalPayment(Map paymentShares) {
        shares.decrease(paymentShares);
    }
}
```

這樣Loan就被簡化了，而且Share計算也被集中到了一個**VALUE OBJECT**中（這個**VALUE OBJECT**只負責這個計算）。但是，這個計算並沒有真正變得通用和易用。

### 在進一步理解之後，把Share Pie變成一個**VALUE OBJECT**

通常，在實現一個新設計的過程中，所獲得的經驗會引導我們對模型本身形成新的認識。在這個例子中，Loan和Share Pie的緊密耦合使Share Pie與Share之間的關係變得模糊不清。如果我們把Share Pie變成一個**VALUE OBJECT**，會產生什麼變化呢？

這意味著不能再使用increase(Map)和decrease(Map)了，因為Share Pie必須是不變的。要更改Share Pie的值，必須整個替換。因此需要使用addshares(Map)這樣的方法來返回一個全新的、更大的Share Pie。

讓我們再進一步把它變成**CLOSURE OF OPERATION**。我們不採用「增加」Share Pie或向它添加Share，而只是把兩個Share Pie加起來，結果是一個新的、更大的Share Pie。

我們可以先把Share Pie上的prorate()操作變成半個閉合操作，這只需要修改返回類型即可。我們把它重命名為prorated()，以便強調它

沒有副作用。「股份數學」開始成型了，最初它有4個操作。

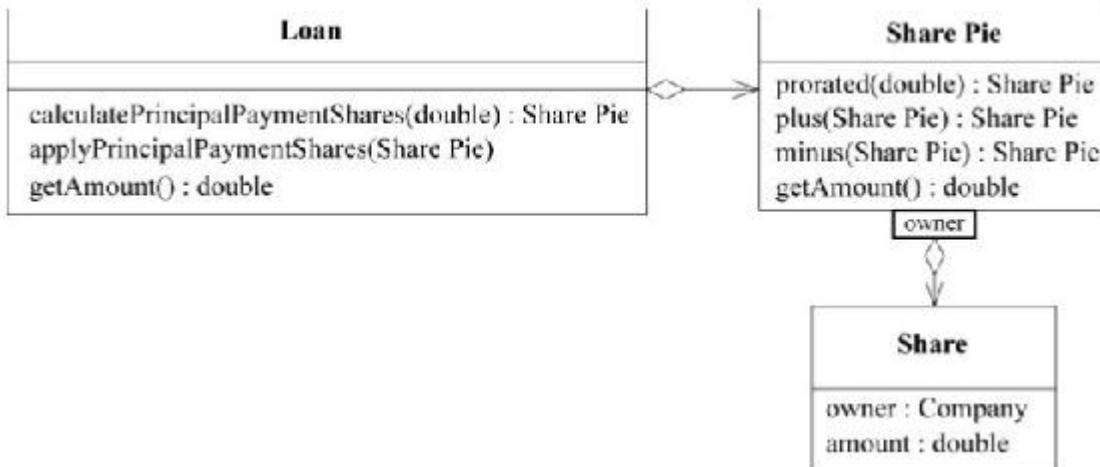


圖10-20

我們可以為新的**VALUE OBJECT** ( **Share Pie** ) 創建一些定義明確的**ASSERTION**。每個方法都有各自的意義。

```
public class SharePie {  
    private Map shares = new HashMap();  
  
    //Accessors and other straightforward methods are omitted  
  
    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;  
    }  
}
```

總股份等於各股份之和

```
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;  
}
```

兩個Pie之差等於這兩個股東所持股份之差

```
public SharePie plus(SharePie otherShares) {  
    //Similar to implementation of minus()  
}  
  
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;  
}
```

兩個Pie的組合就等於把這兩個股東所持股份加到一起  
總額可以依照所有股東所佔的股份按比例劃分

### 新設計的柔性

現在，最重要的Loan類中的方法已經很簡單了，如下：

```
public class Loan {  
    private SharePie shares;  
  
    //Accessors, constructors, and straightforward methods  
    //are omitted  
  
    public SharePie calculatePrincipalPaymentDistribution(  
        double paymentAmount) {  
        return shares.prorated(paymentAmount);  
    }  
  
    public void applyPrincipalPayment(SharePie paymentShares) {  
        setShares(shares.minus(paymentShares));  
    }  
}
```

這些簡短的方法中的每一個都表達了其自己的含義。本金償付表示從貸款中按照股份減去償付額。對已償付的本金進行分配是指在股份持有者之間按比例分配。Share Pie的設計使我們能夠在Loan代碼中使用聲明式風格，所編寫的代碼讀起來像是業務交易的概念定義，而不像是一種計算。

現在，其他交易類型（由於過於複雜沒有在前面列出）也很容易聲明瞭。例如，貸款支取是根據貸方的Facility股份來分配的。新支取的數額被加到未償貸款（Loan）中。用我們的新領域語言可以描述如下：

```

public class Facility {
    private SharePie shares;
    ...
    public SharePie calculateDrawdownDefaultDistribution(
        double drawdownAmount) {
        return shares.prorated(drawdownAmount);
    }
}

public class Loan {
    ...
    public void applyDrawdown(SharePie drawdownShares) {
        setShares(shares.plus(drawdownShares));
    }
}

```

要查看每個貸方的原定貸款額與實際貸款額之差，只需計算該貸方在未償Loan總額中的理論分配值，然後用Loan的實際股份減去這個值即可。

```

SharePie originalAgreement =
    aFacility.getShares().prorated(aLoan.getAmount());
SharePie actual = aLoan.getShares();
SharePie deviation = actual.minus(originalAgreement);

```

Share Pie設計的一些特性使這種組合變得很容易，也提高了代碼的表達能力。

複雜的邏輯通過**SIDE-EFFECT-FREE FUNCTION**被封裝到了專門的**VALUE OBJECT**中。大部分複雜邏輯都已經被封裝到這些不變的對象中。由於Share Pie是**VALUE OBJECT**，因此數學運算可以創建新實例，我們可以用這些新實例來替換舊實例。

Share Pie的所有方法都不會修改任何現有對象。這使我們在中間計算中能夠自由地使用**plus()**、**minus()**和**pro-rated()**，並通過組合它們來實現預期效果，同時又不會產生其他副作用。我們還可以根據這

些方法來創建分析功能（以前，只有在執行實際計算的時候才能調用這些方法，因為在每次調用之後數據就改變了）。

修改狀態的操作很簡單，而且是用**ASSERTION**來描述的。利用「股份數學」的高層抽象，我們可以用聲明式的風格來精確地編寫交易的固定規則。例如，差值是實際股份減去根據**Facility** 的**Share Pie**按比例分配的**Loan**額。

模型概念解除了耦合，操作只涉及最少的其他類型。**Share Pie**上的一些方法顯示出它們是**CLOSURE OF OPERATION**（加、減方法是**Share Pie**之下的閉合操作）。其他操作以簡單的總額作為參數或返回值，它們雖然不是閉合操作，但只增加了極少的概念負擔。**Share Pie**只與一個其他的類——**Share**有密切交互。這樣，**Share Pie**就非常直截了當，易於理解和測試，也很容易通過組合來產生聲明式的交易。這些特性都是從數學形式中繼承得來的。

熟悉的形式使我們更容易掌握協議（*protocol*）。最初用於操作股份的協議本來也是可以用財務術語來設計的，而且從原則上講，這樣的設計也能很靈活。但它有兩個缺點。首先，我們必須從頭開始設計它，這是一項困難且沒有把握完成的任務。其次，每個處理它的人必須先學會它。而我們現在這種設計的好處是，看到股份數學的人會發現他們對這個早已十分熟悉了，而且由於設計與算術規則保持嚴格一致，因此人們不會被誤導。

把與數學形式有關的那部分問題提取出來之後，我們得到了一個柔性的**Share**設計，這使得我們可以進一步精煉核心的**Loan**和**Facility**方法（參見第15章有關**CORE DOMAIN**的討論）。

柔性設計可以極大地提升軟件處理變更和複雜性的能力。正如本章的例子所示，柔性設計在很大程度上取決於詳細的建模和設計決策。柔性設計的影響可能遠遠超越某個特定的建模和設計問題。第15

章將討論柔性設計的戰略價值，我們將把它作為一種工具，用來精煉領域模型，以便使大型和複雜的項目更易於掌握。

## 第11章 應用分析模式

深層模型和柔性設計並非唾手可得。要想取得進展，必須學習大量領域知識並進行充分的討論，還需要經歷大量的嘗試和失敗。但有時我們也能從中獲得一些優勢。一位經驗豐富的開發人員在研究領域問題時，如果發現了他所熟悉的某種職責或某個關係網，他會想起以前這個問題是如何解決的。以前嘗試過哪些模型？哪些是有效的？在實現中有哪些難題？它們是如何解決的？先前經歷過的嘗試和失敗會突然間與新的情況聯繫起來。這些模式當中有一些已經記載到文獻中供大家分享，這樣我們就可以借鑒這些積累的經驗。

與第二部分提出的基本構造塊模式和第10章介紹的柔性設計原則相比，這些模式屬於更高級和專用的模式，其中還涉及使用少量對像來表示某種概念。利用這些模式，可以避免一些代價高昂的嘗試和失敗過程，而直接從一個已經具有良好表達力和易實現的模型開始工作，並解決了一些可能難於學習的微妙的問題。我們可以從這樣一個起點來重構和試驗。然而，它們並不是現成的解決方案。

在《分析模式》一書中，Martin Fowler這樣定義分析模式[Fowler 1997,p.8]：

分析模式是一種概念集合，用來表示業務建模中的常見結構。它可能只與一個領域有關，也可能跨越多個領域。

Fowler所提出的分析模式來自於實踐經驗，因此只要用在合適的情形下，它們會非常實用。對於那些面對著具有挑戰性領域的人們，這些模式為他們的迭代開發過程提供了一個非常有價值的起點。「分

析模式」這個名字本身就強調了其概念本質。分析模式並不是技術解決方案，他們只是些參考，用來指導人們設計特定領域中的模型。

但從這個名字中我們看不出分析模式也討論了大量實現問題，包括一些代碼。Fowler知道，在不考慮實際設計的情況下進行單純的分析是有缺陷的。下面舉一個很有趣的例子，在這個例子中，Fowler用更長遠的眼光審視了模型選擇的意義——考慮在部署之後，模型選擇對系統長期維護的影響[Fowler 1997,p.151]。

當構建一個新的[會計]實務時，我們會創建一個新的過賬規則（posting rule）的實例網。我們可以在完全不需要重新編譯或構建系統的情況下實現它，因而不影響系統的運行。有時我們將不可避免地需要過賬規則的某個新的子類型，但這種情況並不多見。

在一個成熟的項目上，模型選擇往往是根據實用經驗做出的。人們已經嘗試了各種組件的多種實現方法。其中的一些實現已經被採用，有些甚至已經到了維護階段。這些經驗可以幫助人們避免很多問題。分析模式的最大作用是借鑒其他項目的經驗，把那些項目中有關設計方向和實現結果的廣泛討論與當前模型的理解結合起來。脫離具體的上下文來討論模型思想不但難以落地，而且還會造成分析與設計嚴重脫節的風險，而這一點正是**MODEL-DRIVEN DESIGN**堅決反對的。

用實際的例子比用單純的抽象描述能夠更好地解釋分析模式的原則和應用。本章將給出兩個示例，在這兩個例子中，開發人員借鑒了[Fowler 1997]一書中提供的一個具有代表性的範例（來自「Inventory and Accounting」一章）。本章只是為了講解這兩個例子而概述分析模式。顯然，本章的目的不是對這種模式進行歸納分類，甚至對示例所使用的模式也沒有做全面的解釋。本章的重點是說明如何將它們集成到領域驅動的設計過程中。

## 示例 賬戶的利息計算

第10章展示了開發人員為某種專用會計應用程序去尋找更深層模型的各種可能途徑。本示例是另外一個場景，這裡開發人員將深入挖掘Fowler的《分析模式》一書，從中尋找有用的思想。

來複習一下。用於跟蹤貸款和其他有息資產的應用程序將計算所產生的利息和手續費，並跟蹤借方的付款情況。夜間會有一個批處理操作提取這些數字，並傳遞給原來的會計系統，並標明每個賬目應該過帳到哪個分類賬中。這種設計雖然能工作，但使用起來卻很麻煩，修改起來也很複雜，而且不易於交流溝通。

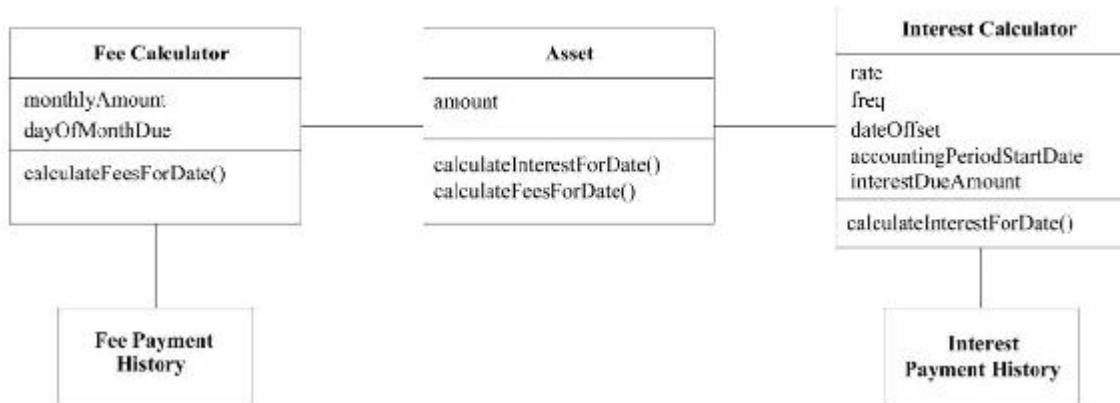


圖11-1 初始的類圖

開發人員決定讀一下《分析模式》的第6章「Inventory and Accounting」。下面摘錄了一些與之最為相關的內容。

### 《分析模式》中的賬戶模型

所有種類的業務應用程序都需要對賬戶進行跟蹤，因為賬戶中保存了與數值有關的信息（通常是錢）。在很多應用程序中，僅跟蹤賬戶總額是不夠的，記錄和控制賬戶總額的每次修改也很重要。這也是會計模型最基本的動機。

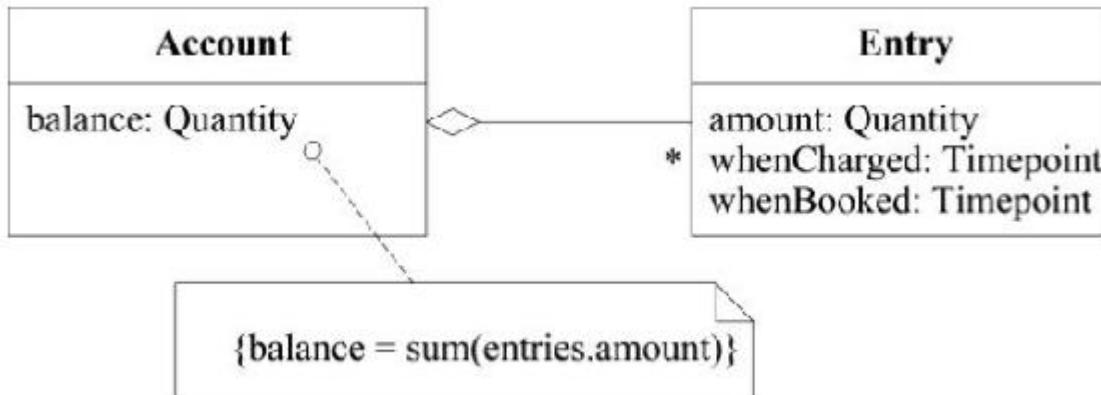


圖11-2 一個基本的賬戶模型

通過插入一個Entry (項) 可以向賬戶中增加數值，而插入一個負的Entry則可以從賬戶中減少數值。Entry永遠不會被刪除，因此整個歷史就被保留下來。餘額就是把所有Entry匯總得到的結果。這個餘額可以實時計算，也可以被緩存，這是由Account接口封裝的一個實現決策。

會計的一條基本原則就是賬目的平恆。錢即不會無中生有，也不會憑空消失。它只能從一個賬戶轉移到另一個賬戶。

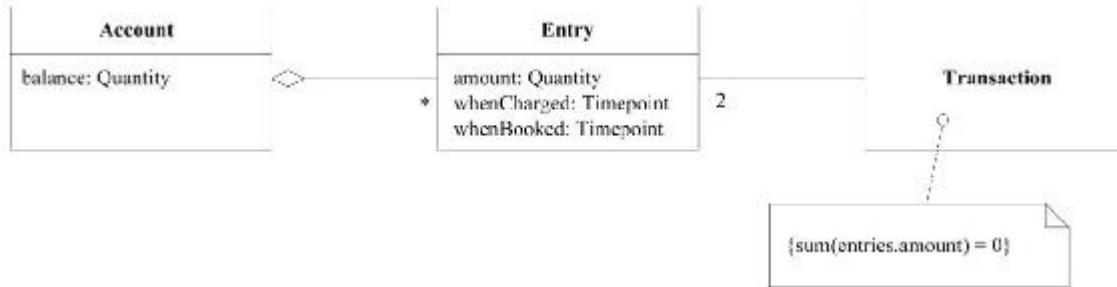


圖11-3 一個交易模型

這就是眾所周知的「複式記賬」 ( double-entry book-keeping ) 概念。每個貸方都有與之相應的借方。當然，像其他守恆定律一樣，它只適用於封閉的系統，這個系統包含了入賬和出賬的所有明細。但很多簡單的應用程序並不需要這麼嚴格。

Fowler在他的書中介紹了這些模型的較全面的形式，並對各種折中選擇做了大量討論。

開發人員（開發人員1）通過閱讀這些內容獲得了一些新的思路。她把這章內容介紹給她的同事（開發人員2）看，這位同事正在與她一起編寫利息計算邏輯，而且他還編寫了夜間批處理程序。他們一起對模型作了粗略的修改——在模型中加進了一些在閱讀該章時看到的模型元素。

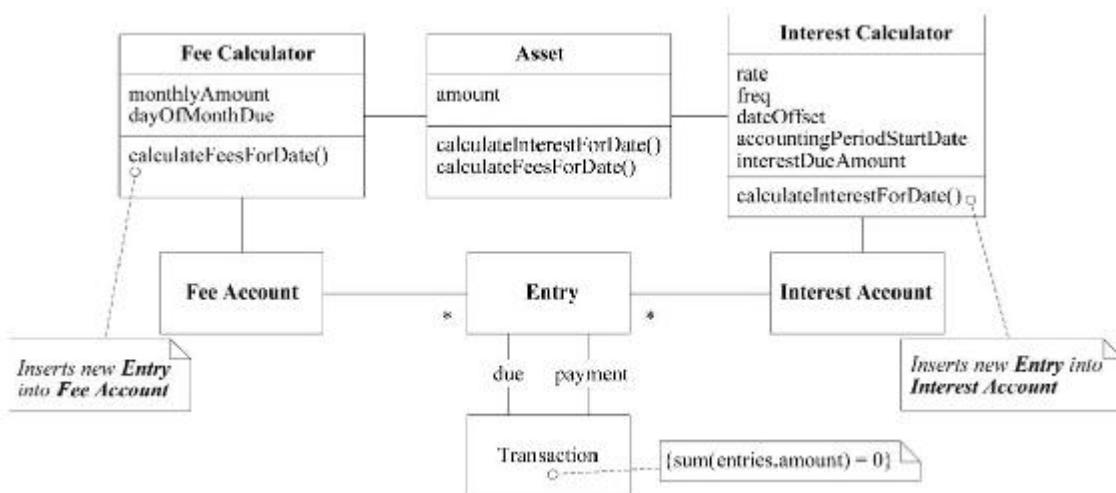


圖11-4 新提出的模型

然後他們找來領域專家（以下稱專家）一起討論新模型的思路。

開發人員1：利用這個新模型，我們可以在Interest Account中為每筆利息收入增加一個Entry，而不是隻調整interestDueAmount。然後，另一個付款的Entry會使其平賬。

專家：這樣是不是就可以看到所有的應計（accrual）利息和付款歷史了？這正是需要的功能。

開發人員2：我不確定這裡使用「Transaction」（交易）是否完全正確。定義講的是把錢從一個Account轉移到另一個Account，而不是兩個Entry在同一個Account中互相平衡。

開發人員1：這個問題很好，我也有些擔心，因為書上似乎強調交易是瞬間建立的，而利息的付款可以過幾天再進行。

專家：那些付款不一定要推遲幾天，在支付時間上可以靈活處理。

開發人員1：那麼這種擔心就沒有必要了。我想我們或許已經發現了一些隱含的概念。讓Interest Calculator來創建Entry對像似乎確實更易理解。而且Transaction似乎把計算出的利息和付款巧妙地聯繫在一起了。

專家：為什麼要把應計項目和付款聯繫在一起呢？它們在會計系統中是分開過帳的。Account的平帳才是主要的。沿著一個一個的Entry，我們就可以查出所有的賬目。

開發人員2：你的意思是說不用跟蹤利息是否已經支付這一點嗎？

專家：當然需要跟蹤。但它並不是你們所說的「一次應計項目/一筆付款」這種簡單的模式。

開發人員2：實際上，如果不用考慮那種關聯，很多事情都可以簡化。

開發人員1：好的，這樣如何？[拿來舊類圖的複印件開始把修改的地方畫出來]。順便問一下，你好幾次提到「應計項目」這個詞，能確切地講一下它的意思嗎？

專家：當然可以。應計項目（accrual）是指在一筆支出或收入發生的時候把它記錄到賬目中，而永遠不管現金實際是何時過帳的。因此，利息每天都會計算，但只有在（舉例來說）月末才會支付。

開發人員1：是的，我們確實需要這個詞。好，現在這個圖怎麼樣？

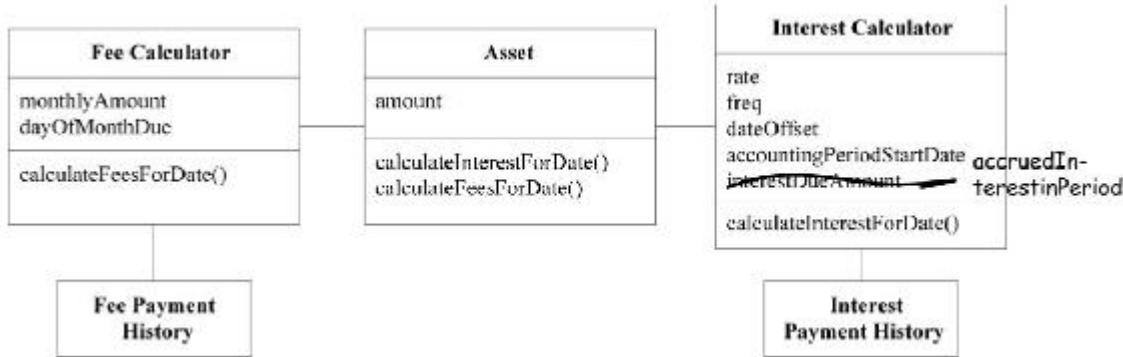


圖11-5 還是原來的類圖，只是把應計項目和付款分開了

開發人員1：現在，我們可以刪掉與付款有關的所有複雜計算了，而且我們引入了「**accrual**」這個術語，它更好地表明瞭我們的意圖。

專家：那麼我們就不會有**Account**對象了吧？我本來還希望能夠把應計項目、付款和餘額等項都放到這個對象中呢。

開發人員1：是嗎？！如果是那樣的話，或許這麼畫就可以[拿起另一張圖開始畫起來]。

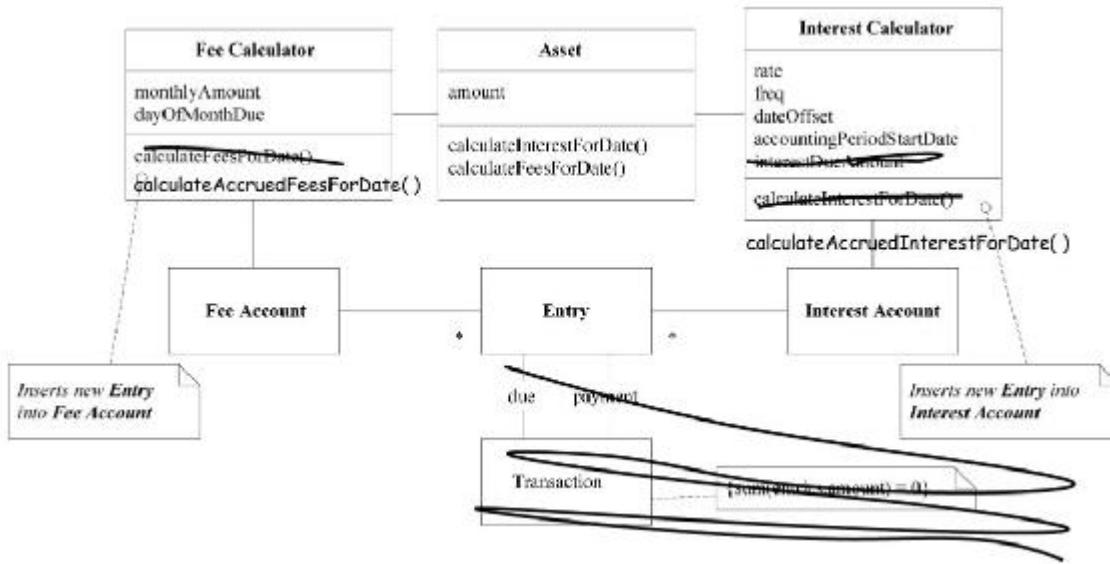


圖11-6 基於賬戶的圖，裡面沒有**Transaction**

專家：這看起來確實好極了！

開發人員2：批處理腳本也需要簡單的修改就能使用這些新對象。

開發人員1：新的Interest Calculator過幾天才能使用。有好些個測試需要修改。但修改之後測試會更清楚。

兩位開發人員開始基於新模型進行重構。他們在著手編寫代碼並加強設計時，又有了一些對模型進行精化的新想法。

通過更仔細的研究，他們決定為Entry創建兩個子類——Payment和Accrual，因為他們發現這兩個子類在應用程序中的職責稍有不同，而且都是非常重要的領域概念。但另一方面，無論Entry是因為手續費而產生的，還是因為利息產生的，其在概念和行為上都沒有任何區別。它們只是出現在適當的Account中。

但遺憾的是，開發人員們發現，出於實現方面的考慮，他們不得不放棄最後這一次抽像。數據存儲在關係表中，而且項目標準要求在不運行程序的情況下也能解釋清楚這些表。這意味著要把手續費項和利息項分開保存到不同的表中。根據他們所使用的對象—關係映射框架，將手續費項和利息項保存到不同表中的唯一方法就是創建具體的子類（Fee Payment、Interest Payment等）。如果換成別的基礎設施，他們或許可以避免使用這些笨拙的子類。

我在這個大部分是虛構的故事中講述這段小插曲的原因是想說明我們在現實中總是會遇到這類小的障礙。我們必須做出一些適當的折中選擇然後繼續前進，而不能因為這些小問題而改變MODEL-DRIVEN DESIGN的方向。

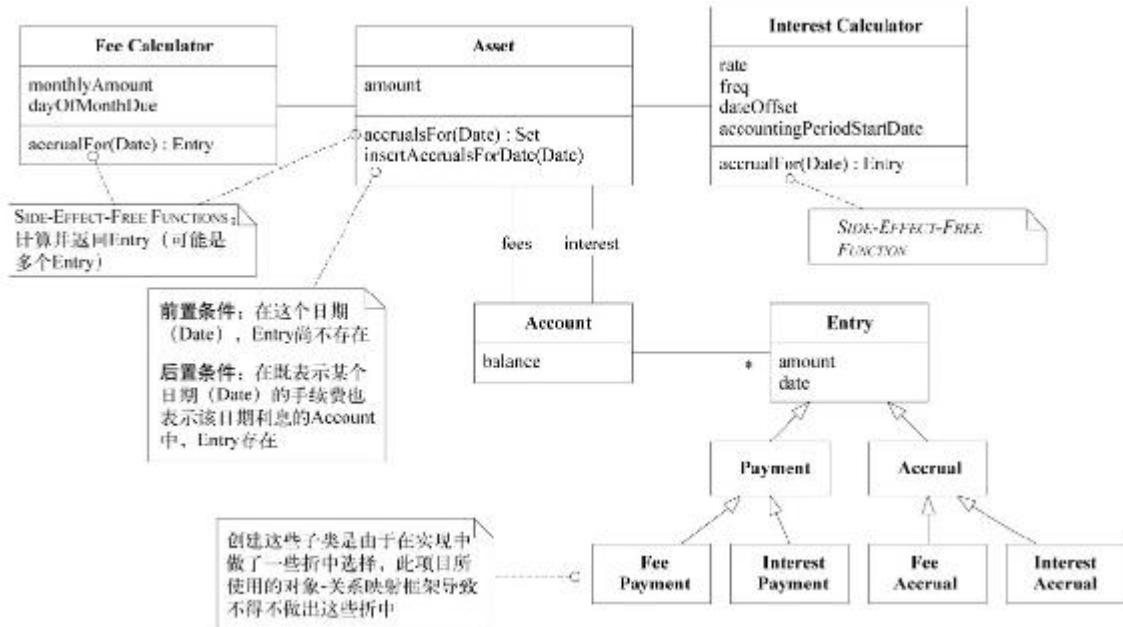


圖11-7 實現之後的類圖

新的設計更易於分析和測試，因為最複雜的功能已被封裝到了 **SIDE-EFFECT-FREE FUNCTION** 中。剩下的命令代碼很簡單（因為它只需調用各種 **FUNCTION**），並使用了 **ASSERTION**。

有時，我們甚至想像不到，程序的一些部分也能從領域模型獲益。它們可能一開始很簡單，並一步步機械地演變。它們看上去就像是複雜的應用程序代碼，而不是領域邏輯。分析模式在找到這些盲點方面特別有用。

在下一個例子中，一位開發人員對夜間批處理程序的內部機制產生了新的想法，以前他並沒有從領域的角度來考慮這一問題。

### 示例 對夜間批處理程序的深入理解

幾星期後，改進後的基於 **Account** 的模型基本完成了。如時常發生的那樣，當新設計更加清晰之後，它就暴露出其他一些問題。開發人員（開發人員2）在修改夜間批處理程序以使之與新設計交互的時候，發現批處理程序的行為與《分析模式》一書中所講的一些概念有聯繫。下面就是他發現的一些最相關的概念：

## 過帳規則

會計系統經常提供同一個基本財務信息的多種視圖。一個賬戶可能用於跟蹤收入，而另一個賬戶可能用於跟蹤該收入的估稅。如果我們希望系統自動更新估稅總額，那麼這兩個賬戶的實現將會彼此緊密關聯。在有些系統中，大部分賬目都是由這些規則產生的，在這樣的系統中，依賴邏輯會變得一團糟。即使是在規模不大的系統中，這樣的交叉過帳也會很複雜。減少這種纏雜不清的依賴的第一步是通過引入一個新對像來使這些規則明朗化。

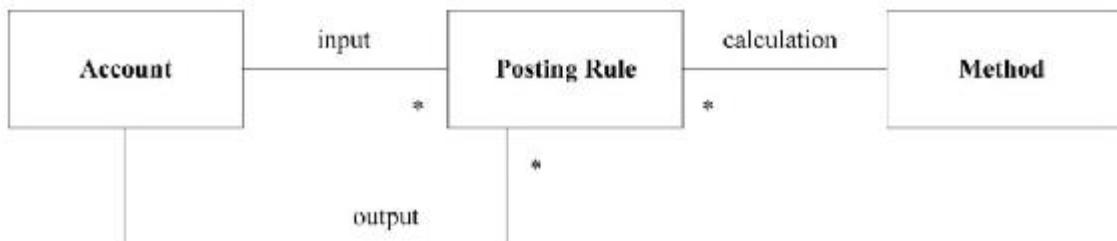


圖11-8 基本過帳規則的類圖

當過帳規則的**input**賬目收到一個新的**Entry**時，這個**Entry**就會觸發過帳規則。然後過帳規則會生成一個新的**Entry**（基於它自己的計算方法），並將這個**Entry**插入它的「**output**」賬目中。在工資系統中，工資**Account**中的**Entry**可能會觸發一個過帳規則，此規則計算30%的估計收入所得稅，並將其作為一個**Entry**插入扣稅**Account**中。

## 執行過帳規則

過帳規則建立了各個**Account**之間概念上的依賴性，但如果對這個模式的使用僅限於此，那麼它仍然很難使用。在依賴性設計中，最複雜的部分是更新的時機和控制措施。Fowler討論了3種選擇：

(1)「主動觸發」（**Eager firing**）是最直接的方式，但通常也最不實用。每當一個**Entry**被插入到**Account**中時，它立即就觸發過帳規則，並立即進行所有更新。

(2) 「基於Account的觸發」允許推遲處理。在過後的某個時刻，向Account發送一條消息，令其觸發過帳規則，來處理自從上一次觸發以來所插入的所有Entry。

(3) 最後，「基於過帳規則的觸發」由外部代理來啟動，它通知過帳規則觸發。過帳規則負責查找自從上次觸發以來插入到其輸入Account中的所有Entry。

儘管在一個系統中可以混合使用各種觸發模式，但每組特定的規則都需要有一個明確定義的「啟動點」（應該在何時啟動），還要定義由誰負責查找插入到輸入的Account中的Entry。將這三種觸發模式添加到UBIQUITOUS LANGUAGE中對於成功使用這種模式具有至關重要的意義，這與模型對像定義本身同等重要。這樣，觸發的概念就不再模糊了，而且還能直接指導決策，從而獲得一組明確定義的可選方案。這些觸發模式揭示出了一個很容易被忽略的重點，並且豐富了我們的詞彙，從而使討論更清晰。

開發人員2需要找個人來討論他的新思路。他找到了同事（開發人員1），開發人員1原來主要負責建立應計項目（accrual）的模型。

開發人員2：有的時候，夜間批處理程序成為一個隱藏問題的地方。腳本的行為中隱含了領域邏輯，而且正在變得越來越複雜。很長時間以來，我一直想用MODEL-DRIVEN DESIGN的方法來修改一下批處理，將領域層分離出來，並使腳本本身成為領域層之上的一個簡單的層。但我一直沒有想出這個領域模型應該是什麼樣的。看上去它似乎只是一些操作步驟，而把它們實現為對像沒什麼實際意義。但當我讀完《分析模式》一書中有關Posting Rule的內容後，獲得了一些思路。這個圖就是我所想到的[遞過來一張草圖]。

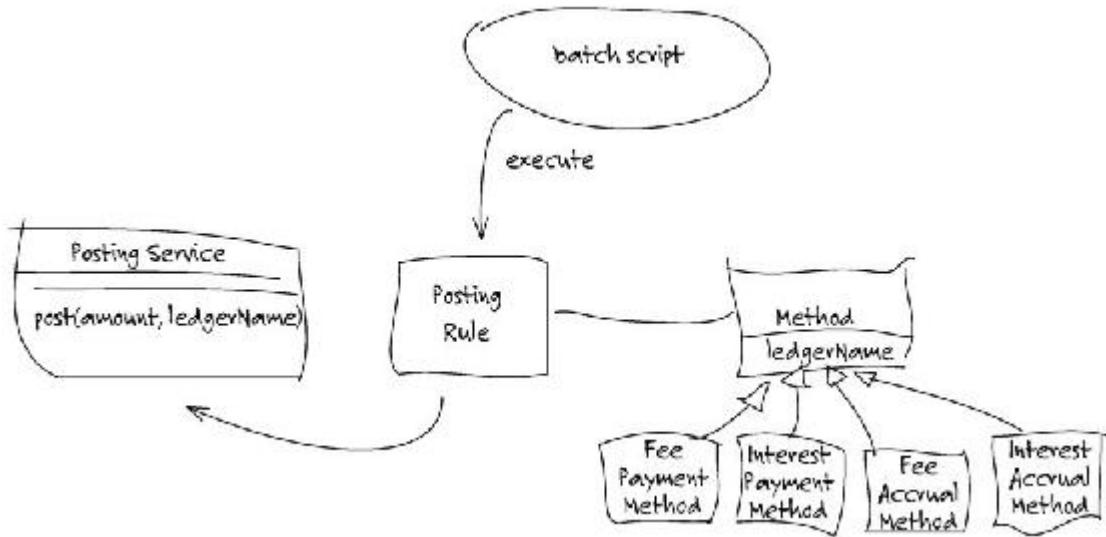


圖11-9 在批處理中使用過帳規則的一個思路

開發人員1：Posting Service是指什麼？

開發人員2：這是個FAÇADE，它提供了會計應用程序的API，並且將其呈現為一個SERVICE。實際上我使用它已經有一段時間了，主要用來簡化批處理代碼，而且它也為我提供了一個INTENTION-REVEALING INTERFACE，可用於向老系統過帳。

開發人員1：很有趣，那麼你打算為這些Posting Rule（過帳規則）使用哪種觸發模式？

開發人員2：我還沒有想那麼多。

開發人員1：「主動觸發」可能適用於Accrual，因為批處理程序實際上通知Asset插入Accrual，但「主動觸發」可能不適用於Payment，因為Payment是在白天輸入的。

開發人員2：不管怎樣，我認為我們都不希望把計算方法與批處理程序特別緊密地聯繫到一起。如果我們決定在一個不同的時間來觸發利息計算，那麼情況將會是一團糟。而且從概念上看，這也是不正確的。

開發人員1：這聽上去有點像「基於Posting Rule的觸發」。由批處理程序通知每個Posting Rule去執行，然後規則找出相應的新Entry，並完成其工作。這種思路基本上就與你畫的圖中表現出來的思路差不多吧。

開發人員2：這樣在批處理設計中就不會產生很多依賴，而且它也易於控制了，看樣子不錯。

開發人員1：我沒有完全明白這些對象是如何與Account和Entry交互的。

開發人員2：我也沒完全明白。那本書中的示例在Account和Posting Rule之間建立了直接聯繫。在某種程度上這是合乎邏輯的，但我認為它並不完全適用於我們的情況。我們每次都需要用數據來實例化這些對象，因此要使用這種方法，必須知道應用哪條規則。同時，Asset對像知道每個Account的內容，因此也知道應用哪條規則。不管怎麼說，這個模型的其他方面呢？

開發人員1：雖然我討厭過分挑剔，但我確實認為Method的使用不正確。我認為在概念上Method是用於計算要過帳的總額的，比方說，在收入上計算20%的扣稅。但我們的情形很簡單：它始終是全額過帳。我想Posting Rule本身應該是知道要過帳給哪個Account的，這個Account對應於我們的ledger name（分類賬名稱）。

開發人員2：哦，那麼如果讓Posting Rule負責查知正確的ledger name，我們可能就完全不需要Method了。

實際上，選擇正確的ledger name這件事情變得越來越複雜了。它已經是收入類型（手續費或利息）與「Asset類別」（公司對每種Asset所使用的分類）的組合了。我希望新模型能夠在解決這個複雜性上有所幫助。

開發人員1：好的，我們就把重點集中在這裡。Posting Rule負責根據Account的屬性來選擇Ledger。現在，我們可以先用一種簡單直接的方式來處理資產類型以及利息與收入之間的區分。將來，我們會有一個OBJECT MODEL，可以通過改進這個模型來處理更複雜的情形。

開發人員2：在這方面我還要多考慮一下。我會再仔細研究一番，再把模式讀一遍，然後再來嘗試解決這個問題。明天下午我能夠再和你討論這個問題嗎？

在接下來的幾天時間裡，這兩位開發人員設計出了一個模型，並對代碼進行了重構，使得批處理程序只是簡單地依次訪問各個Asset，並向每個Asset發送幾條非常淺顯易懂的消息，然後提交數據庫事務。複雜性被轉移到領域層中，領域層中的對象模型使問題變得更加明確，也更抽象。

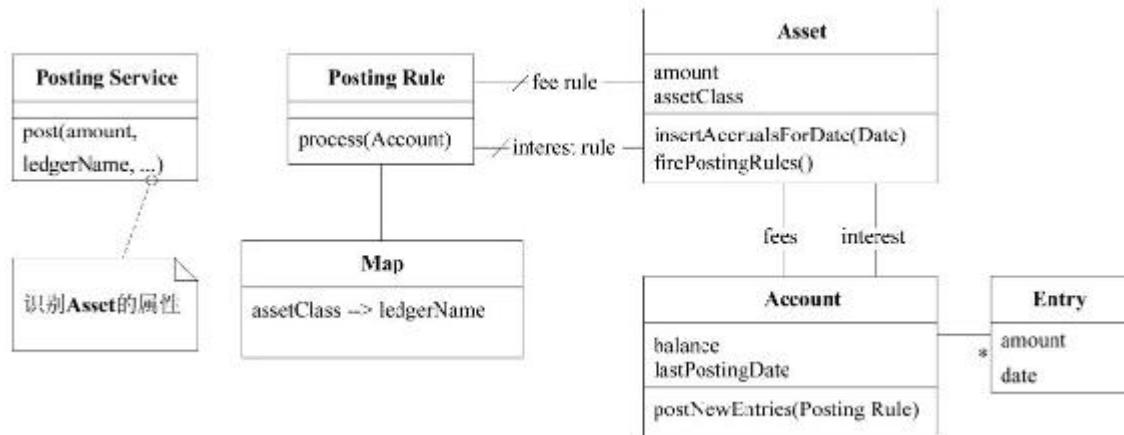


圖11-10 含有過帳規則的類圖

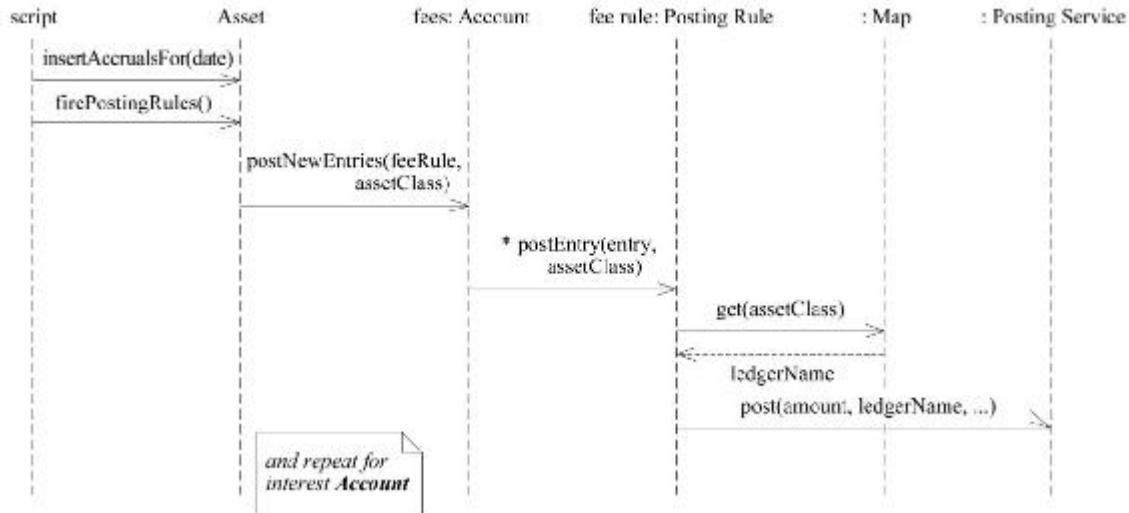


圖11-11 顯示了規則觸發的序列圖

在一些細節問題上，這兩位開發人員開發的模型與《分析模式》中給出的相差甚遠，但他們認為二者在概念本質上還是相同的。有一個問題令他們稍感不安，那就是在 **Posting Rule** 的選擇中把 **Asset** 牽扯進來了。之所以這樣做，是因為 **Asset** 知道每個 **Account** 的性質（手續費或利息），而且它也是腳本的自然訪問點。如果讓規則對像直接與 **Account** 發生關聯，這些對像在每次實例化時（每次運行批處理程序時）都需要與 **Asset** 對像進行協作。可他們沒有這樣做，而是讓 **Asset** 對像通過 **SINGLETON** 訪問來查找這兩個相關規則，並把對應的 **Account** 傳遞給它們。這樣一來代碼就變得更直接了，因此這是一個正確的決定。

從概念上看，他們都感到更好的做法是讓 **Posting Rule** 只與 **Account** 發生關聯，而令 **Asset** 只負責生成 **Accrual**。他們希望等到有了後續的重構和更深入的理解之後再回頭看這個問題，並找到一種將職責分離得更清楚而又不影響代碼明確性的方法。

### 分析模式是很有價值的知識

當你可以幸運地使用一種分析模式時，它一般並不會直接滿足你的需求。但它為你的研究提供了有價值的線索，而且提供了明確抽象

的詞彙。它還可以指導我們的實現，從而省去很多麻煩。

我們應該把所有分析模式的知識融入到知識消化和重構的過程中，從而形成更深刻的理解，並促進開發。當我們應用一種分析模式時，所得到的結果通常與該模式的文獻中記載的形式非常相像，只是因具體情況不同而略有差異。但有時完全看不出這個結果與分析模式本身有關，然而這個結果仍然是受該模式思想的啟發而得到的。

但有一個誤區是應該避免的。當使用眾所周知的分析模式中的術語時，一定要注意，不管其表面形式的變化有多大，都不要改變它所表示的基本概念。這樣做有兩個原因，一是模式中蘊含的基本概念將幫助我們避免問題，二是（也是更重要的原因）使用被廣泛理解或至少是被明確解釋的術語可以增強**UBIQUITOUS LANGUAGE**。如果在模型的自然演變過程中模型的定義也發生改變，那麼就要修改模型名稱了。

很多對像模型都有文獻資料可查，其中有些對像模型專門用於某個行業中的某種應用，而有些則是通用模型。大部分對像模型都有助於開闊思路，但只有為數不多的一些模型精闢地闡述了選擇這些模式的原理和使用的結果，而這些才是分析模式的精華所在。這些精化後的分析模式大部分都很有價值，有了它們，可以免去一次次的重複開發工作。儘管我們不大可能歸納出一個包羅萬象的分析模式類目，但針對具體行業的類目還是能夠開發出來的。而且在一些跨多個應用的領域中適用的模式可以被廣泛共享。

這種對已組織好的知識的重複利用完全不同於通過框架或組件進行的代碼重用，但是二者唯一的共同點是它們都提供了一種新思路的萌芽，而這種新思路先前可能並不十分明晰。一個模型，甚至一個通用框架，都是一個完整的整體，而分析則相當於一個工具包，它被應用於模型的一些部分。分析模式專注於一些最關鍵和最艱難的決策，

並闡明瞭各種替代和選擇方案。它們提前預測了一些後期結果，而如果單靠我們自己去發現這些結果，可能會付出高昂的代價。

## 第12章 將設計模式應用於模型

到目前為止，本書所探討的模式都是專門用來在MODEL-DRIVEN DESIGN的上下文中解決領域模型的問題。但實際上，大部分已發佈的模式都更側重於解決技術問題。設計模式與領域模式之間有什麼區別？《設計模式》這部經典著作的作者為初學者指出了以下事實 [Gamma et al.1995,p.3]：

立場不同會影響人們如何看待什麼是模式以及什麼不是模式。一個人所認為的模式在另一個人看來可能是基本構造塊。本書將在一定的抽象層次上討論模式。設計模式並不是指像鏈表和散列表那樣可以被封裝到類中並供人們直接重用的設計，也不是用於整個應用程序或子系統的複雜的、領域特定的設計。本書中的設計模式是對一些交互的對象和類的描述，我們通過定製這些對象和類來解決特定上下文中的一般設計問題。

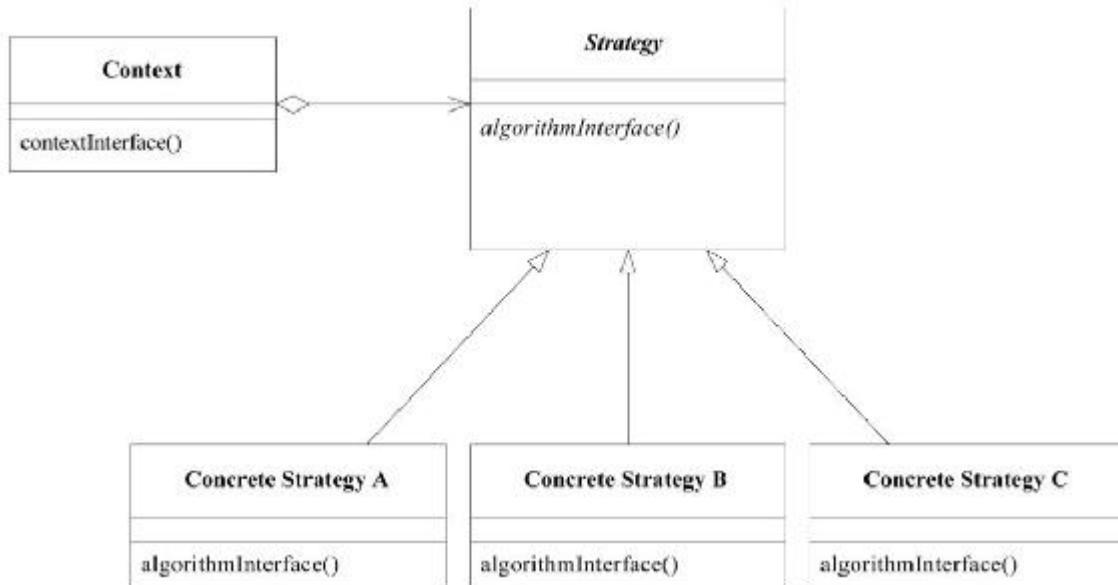
在《設計模式》中，有些（但並非所有）模式可用作領域模式，但在這樣使用的時候，需要變換一下重點。《設計模式》中的設計模式把相關設計元素歸為一類，這些元素能夠解決在各種上下文中經常遇到的問題。這些模式的動機以及模式本身都是從純技術角度描述的。但這些元素中的一部分在更廣泛的領域和設計上下文中也適用，因為這些元素所對應的基本概念在很多領域中都會出現。

除了《設計模式》中介紹的模式以外，近年來還出現了其他很多技術設計模式。有些模式反映了在一些領域中出現的深層概念。這些模式都有很大的利用價值。為了在領域驅動設計中充分利用這些模

式，我們必須同時從兩個角度看待它們：從代碼的角度來看它們是技術設計模式，從模型的角度來看它們就是概念模式。

我們將把《設計模式》所介紹的特定模式作為樣例，來說明如何將人們所認為的設計模式應用到領域模型中，而且這個例子還將澄清技術設計模式與領域模式之間的區別。本章還將通過COMPOSITE（組合）和STRATEGY（策略）這兩種模式演示如何通過改變思考方式，用一些經典的設計模式來解決領域問題。

## 12.1 模式：STRATEGY（也稱為POLICY）



定義了一組算法，將每個算法封裝起來，並使它們可以互換。STRATEGY允許算法獨立於使用它的客戶而變化。[Gamma et al.1995]

領域模型包含一些並非用於解決技術問題的過程，將它們包含進來是因為它們對處理問題領域具有實際的價值。當必須從多個過程中進行選擇時，選擇的複雜性再加上多個過程本身的複雜性會使局面失去控制。

當對過程進行建模時，我們經常會發現過程有不止一種合理的實現方式，而如果把所有的可選項都寫到過程的定義中，定義就會變得臃腫而複雜，而且可供我們選擇的實際行為也會因為混雜在其他行為中而顯得模糊不清。

我們希望把這些選擇從過程的主體概念中分離出來，這樣既能夠看清主體概念，也能更清楚地看到這些選擇。軟件設計社區中眾所周知的**STRATEGY**模式就是為瞭解決這個問題的，雖然它的側重點在於技術方面。這裡，我們把它當成模型中的一個概念來使用，並在該模型的代碼實現中把它反映出來。我們同樣也需要把過程中極易發生變化的部分與那些更穩定的部分分離開。

因此：

我們需要把過程中的易變部分提取到模型的一個單獨的「策略」對像中。將規則與它所控制的行為區分開。按照**STRATEGY**設計模式來實現規則或可替換的過程。策略對象的多個版本表示了完成過程的不同方式。

通常，作為設計模式的**STRATEGY**側重於替換不同算法的能力，而當其作為領域模式時，其側重點則是表示概念的能力，這裡的概念通常是指過程或策略規則。

### 示例 路線查找 ( **Route-Finding** ) 策略

我們把一個 **Route Specification** ( 路線規格 ) 傳遞給 **Routing Service** ( 路線服務 )，**Routing Service** 的職責是構造一個滿足 **SPECIFICATION**的詳細的 **Itinerary**。這個 **SERVICE** 是一個優化引擎，可以通過調節它來查找最快的路線或最便宜的路線。

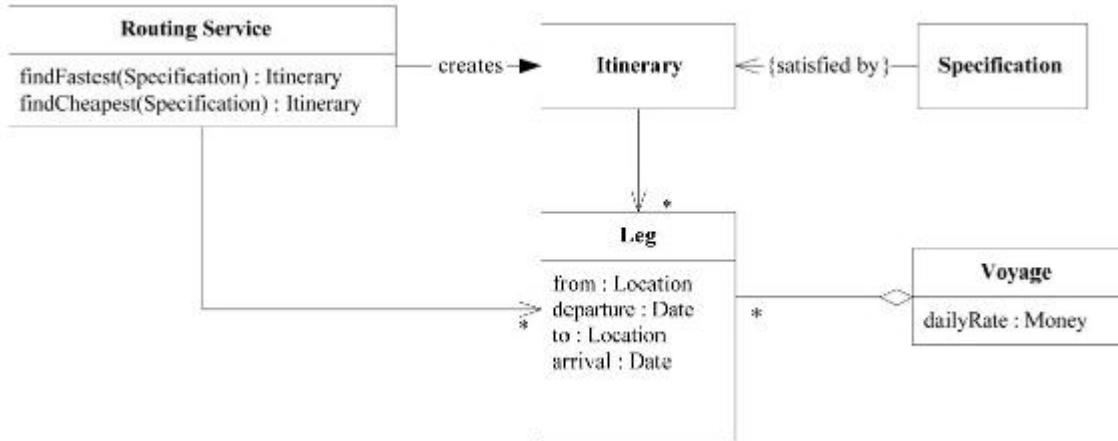


圖12-1 帶選項的SERVICE接口需要條件邏輯

這種設置看上去似乎沒問題，但仔細觀察路線代碼就會發現，每個計算中都有條件判斷，判斷最快還是最便宜的邏輯分散在程序各處。當為了做出更精細的航線選擇而把新標準添加進來時，麻煩會更多。

解決此問題的一種方法是把這些起調節作用的參數分離到 **STRATEGY** 中。這樣它們就可以被明確地表示出來，並作為參數傳遞給 **Routing Service**。

現在，**Routing Service**就可以用一種完全相同的、無需進行條件判斷的方式來處理所有請求了，它按照 **Leg Magnitude Policy** ( 航段規模策略 ) 的計算，找出一系列規模較小的 **Leg** ( 航段 )。

這種設計具有《設計模式》中所介紹的 **STRATEGY** 模式的優點。按這種思路設計的應用程序可以提供豐富的功能，同時也很靈活，現在，可以通過安裝適當的 **Leg Magnitude Policy** 來控制和擴展 **Routing Service** 的行為。圖12-2中顯示的只是最明顯的兩種 **STRATEGY** ( 最快或最便宜 )。可能還會有一些在速度和成本之間進行權衡考慮的組合策略。也可以加進其他的因素，例如，在預訂貨物時優先選擇公司自己的運輸系統，而不是外包給其他運輸公司。不使用 **STRATEGY** 模式同樣能實現這些修改，但必須將邏輯添加到 **Routing Service** 的內部。

( 這會是一個麻煩的過程 ) , 而且這些邏輯會使接口變得臃腫。解耦確實令 **Routing Service** 更清楚且易於測試。

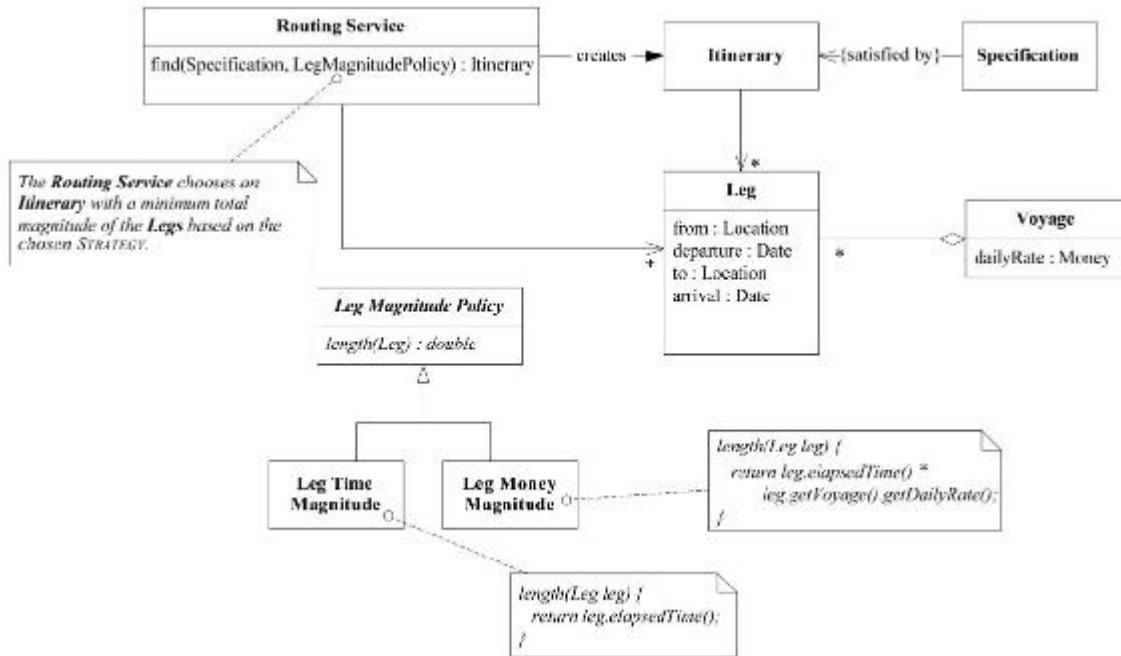


圖12-2 通過STRATEGY ( 或者POLICY ) 來確定選項 ( STRATEGY是作為參數傳入的 )

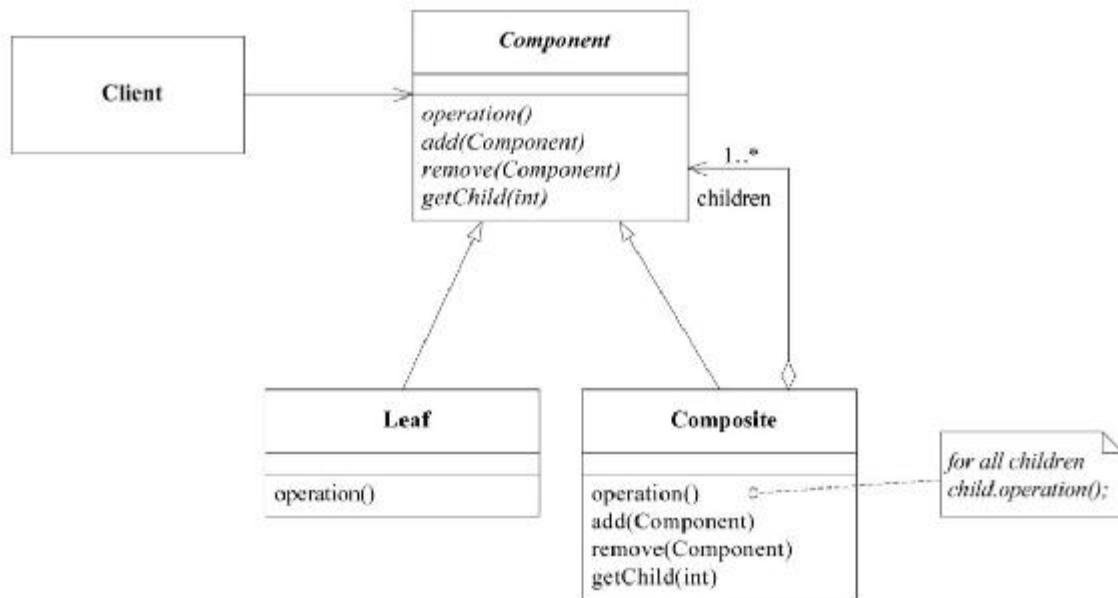
現在，領域中的一個至關重要的規則明確地顯示出來了，也就是在構建 **Itinerary** 時用於選擇 **Leg** 的基本規則。它傳達了這樣一個知識：路線選擇的基礎是航段的一個特定屬性（有可能是派生屬性），這個屬性最後可歸結為一個數字。這樣，我們就能夠通過領域語言很簡單地定義 **Routing Service** 的行為：**Routing Service** 根據所選的 **STRATEGY** 來選擇 **Leg** 總規模最小的 **Itinerary**。

說明：以上討論暗示了一件事。**Routing Service** 在查找 **Itinerary** 時實際上會計算 **Leg** 的規模。這種方法在概念上比較直接，而且可以生成一個合理的原型實現，但它的效率可能令人無法接受。第14章會再次討論這個應用程序，其將使用相同的接口，但採用完全不同的 **Routing Service** 實現。

我們在領域層中使用技術設計模式時，必須認識到這樣做的另外一種動機，也是它的另一層含義。當所使用的STRATEGY對應於某種實際的業務策略時，模式就不再僅僅是一種有用的實現技術了（但它在實現方面的價值並未改變）。

設計模式的結論也完全適用於領域層。例如，在《設計模式》一書中，Gamma等人指出客戶必須知道不同的STRATEGY，這也是建模的一個關注點。如果單純從實現上來考慮，使用策略可能會增加系統中對象的數目。如果這是個問題，可以把STRATEGY實現為無狀態對象，以便在上下文中進行共享，從而減小開銷。《設計模式》中對實現方法的全面討論在這裡也適用，這是因為我們仍然在使用STRATEGY，只是動機稍有不同，這會對我們的選擇產生一些影響，但設計模式中的經驗仍然是可以借鑒的。

## 12.2 模式：COMPOSITE



將對像組織為樹來表示部分—整體的層次結構。利用COMPOSITE，客戶可以對單獨的對象和對像組合進行同樣的處理。

## [Gamma et al.1995]

在對複雜的領域進行建模時，我們經常會遇到由多個部分組成的重要對象，這些部分本身又由其他一些部分組成，依此類推，有時甚至會出現任意深度的嵌套。在一些領域中，各層嵌套在概念上是有區別的，但在另一些領域中，各個部分與它們所組成的整體是完全相同的事物，只是規模較小一些而已。

當嵌套容器的關聯性沒有在模型中反映出來時，公共行為必然會在層次結構的每一層重複出現，而且嵌套也變得僵化（例如，容器通常不能包含同一層中的其他容器，而且嵌套的層數也是固定的）。客戶必須通過不同的接口來處理層次結構中的不同層，儘管這些層在概念上可能沒有區別。通過層次結構來遞歸地收集信息也變得非常複雜。

當在領域中應用任何一種設計模式時，首先關注的問題應該是模式的意圖是否確實適合領域概念。以遞歸的方式遍歷一些相互關聯對像確實比較方便，但它們是否真的存在整體一部分層次結構？你是否發現可以通過某種抽象方式把所有部分都歸到同一概念類型中？如果你確實發現了這種抽象方式，那麼使用**COMPOSITE**可以令模型的這些部分變得更清晰，同時使你能夠藉助設計模式所提供的那些經過深思熟慮的設計及實現的考量。

因此：

定義一個把**COMPOSITE**的所有成員都包含在內的抽象類型。在容器上實現那些查詢信息的方法時，這些方法返回由容器內容所匯總的信息。而「葉」節點則基於它們自己的值來實現這些方法。客戶只需使用抽象類型，而無需區分「葉」和容器。

相對而言，這是一種明顯的結構層面上的模式，但設計人員通常不會主動地充實它的操作方面。**COMPOSITE**模式在每個結構層上都提

供了相同的行為，而且無論是較小的部分還是較大的部分，都可以對這些部分提出一些有意義的問題，這些問題能夠透明地反映出它們的構成情況。這種嚴格的對稱是組合模式具有強大能力的關鍵所在。

### 示例 由Route構成的Shipment Route

完整的貨物運輸路線是很複雜的。首先，必須用卡車把集裝箱運輸到鐵路終點站，然後運送到港口，之後用貨輪運輸到另一個港口，中間可能還會換船，最後還要進行地面運輸才能到達目的地。

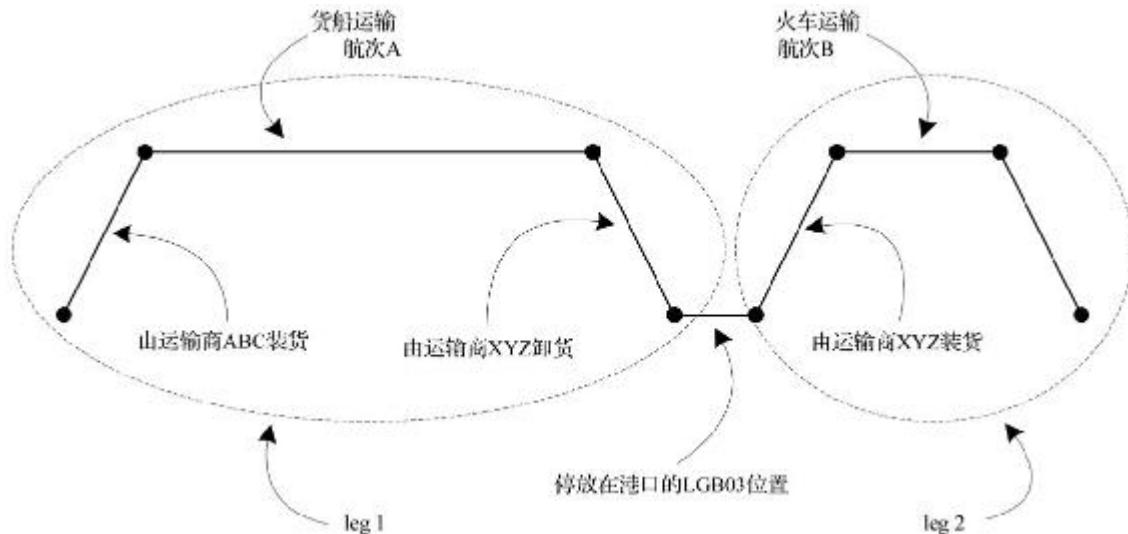


圖12-3 由「leg」（航段）構成的「route」（航線）

一個應用開發團隊創建了一個對像模型，它表示了一個航線可以由任意多個航段組成。

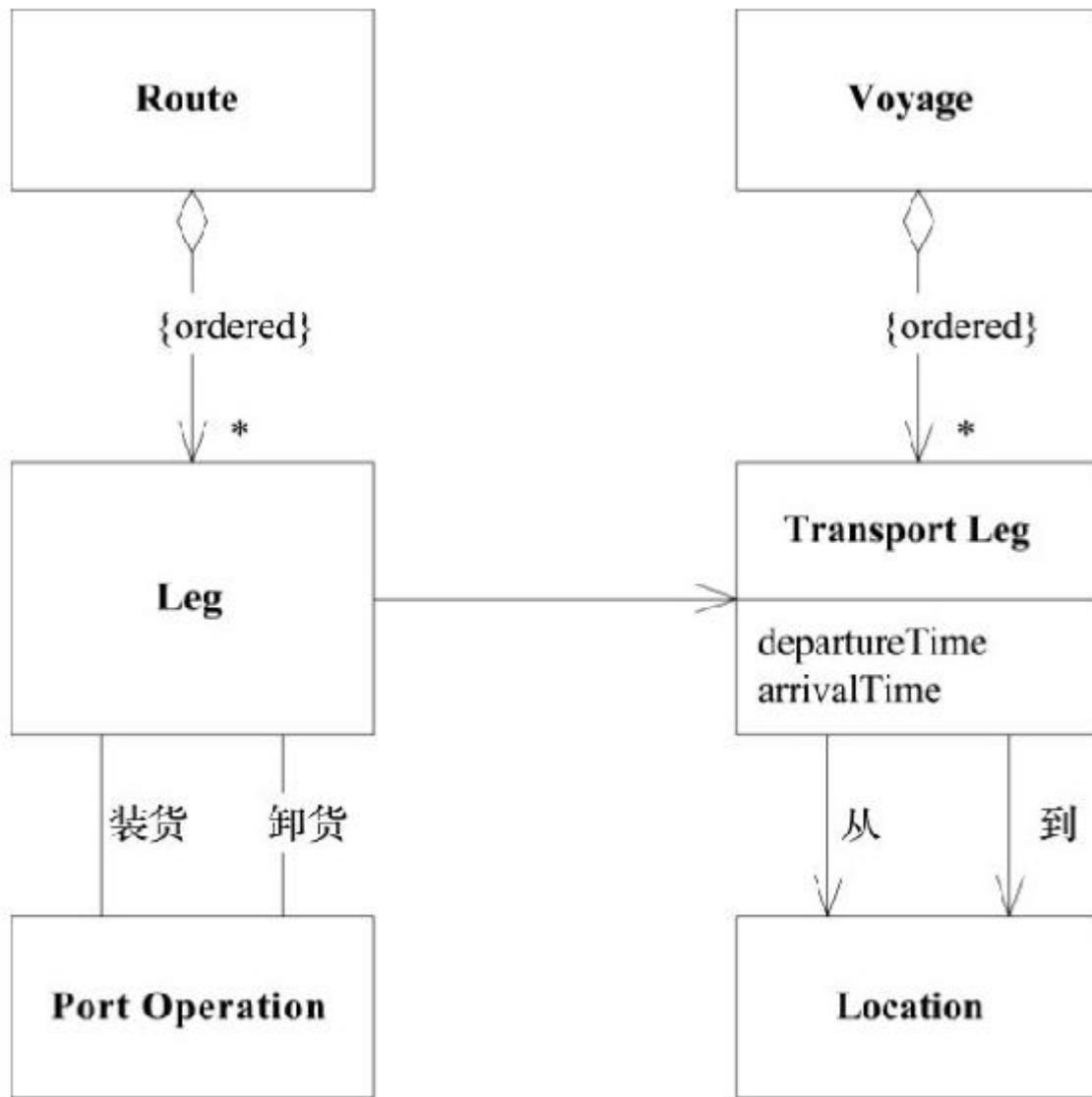


圖12-4 Route的類圖，其中Route由多個Leg組成

利用這個模型，開發人員可以根據預訂請求來創建Route對象。他們可以把這些Leg組織為一步一步運輸貨物的操作計劃。在這個過程中他們發現了一些問題。

開發人員原來一直認為航線是由任意多個航段組成的，而各個航段之間並沒有什麼區別。



圖12-5 開發人員的航線概念

而事實上領域專家把航線看成是由5個邏輯段組成的序列。

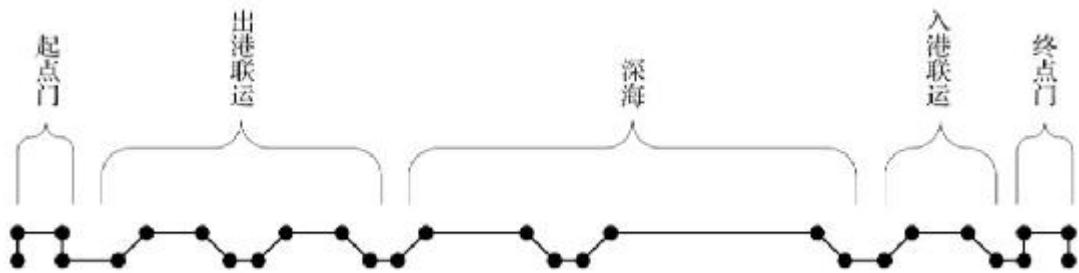


圖12-6 業務專家的航線概念

其他問題先不考慮，這些小段的航線可能是由不同的人在不同時間規劃的，因此必須區別對待。通過更仔細的研究可以發現，「門航段」（door leg）與其他航段大不相同，它涉及在當地僱用卡車甚至是客戶運輸，這與詳細計劃的鐵路和貨船運輸完全不同。

反映了所有這些區別的對象模型漸漸變得複雜起來。

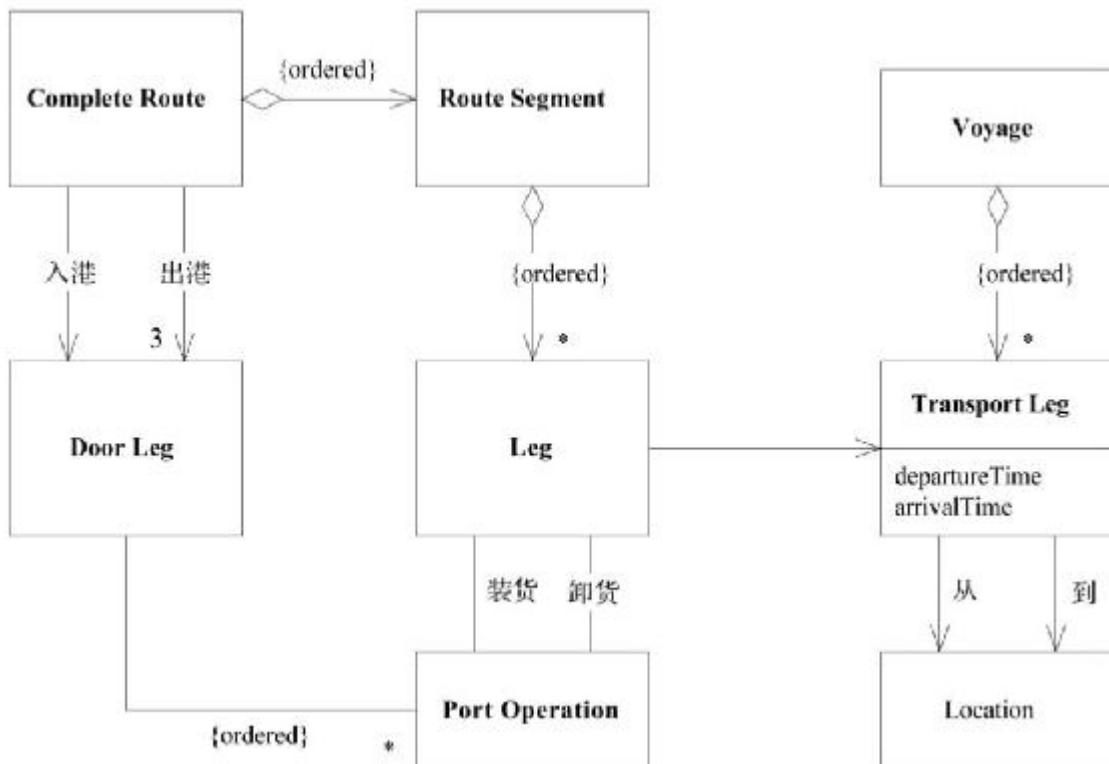


圖12-7 詳細的Route類圖

從結構上看這個模型並不是很差，但在操作計劃的處理上失去了  
一致性，因此代碼（甚至是行為的描述）變得複雜得多。其他複雜之  
處也漸漸顯現。任何一條航線的遍歷都涉及不同類型對象的多個集  
合。

運用**COMPOSITE**模式能使特定客戶在不同層上都使用這種構造進  
行統一的處理，因為大的航線是由小段的航線構成的。這種視圖在概  
念上也是合理的。每一層**Route**都是集裝箱從一個地點到另一個地點  
的移動，最後都歸結為一個獨立的航段（參見圖12-8）。

與前面那個類圖不同，從現在這個靜態類圖看不出來門航段是如  
何與其他航段組合在一起的。但模型並不只包含靜態類圖。我們將通  
過其他的圖（參見圖12-9）和代碼（現在代碼簡單多了）來表示這些  
航段的組合信息。這個模型抓住了所有這些不同類型**Route**的深層關  
聯性。生成操作計劃的工作再次變得簡單了，而且其他路線遍歷操作  
也變得簡單了。

利用這種「由航線組成航線」的方法，我們可以把各個航線的端  
點連接到一起來得到從一個地點到另一個地點的航線，從而可以實現  
各種不同的航線。我們可以把航線的一端截去，再拼接一段新的航  
線，我們可以有任何細節的嵌套，而且可以充分利用一切可能有用的  
選項。

當然，我們現在還不需要這些選擇。當不需要這些航線分段和不  
同的「門航段」時，不使用**COMPOSITE**模式也能很好地工作。設計模  
式應該僅僅在需要的時候才使用。

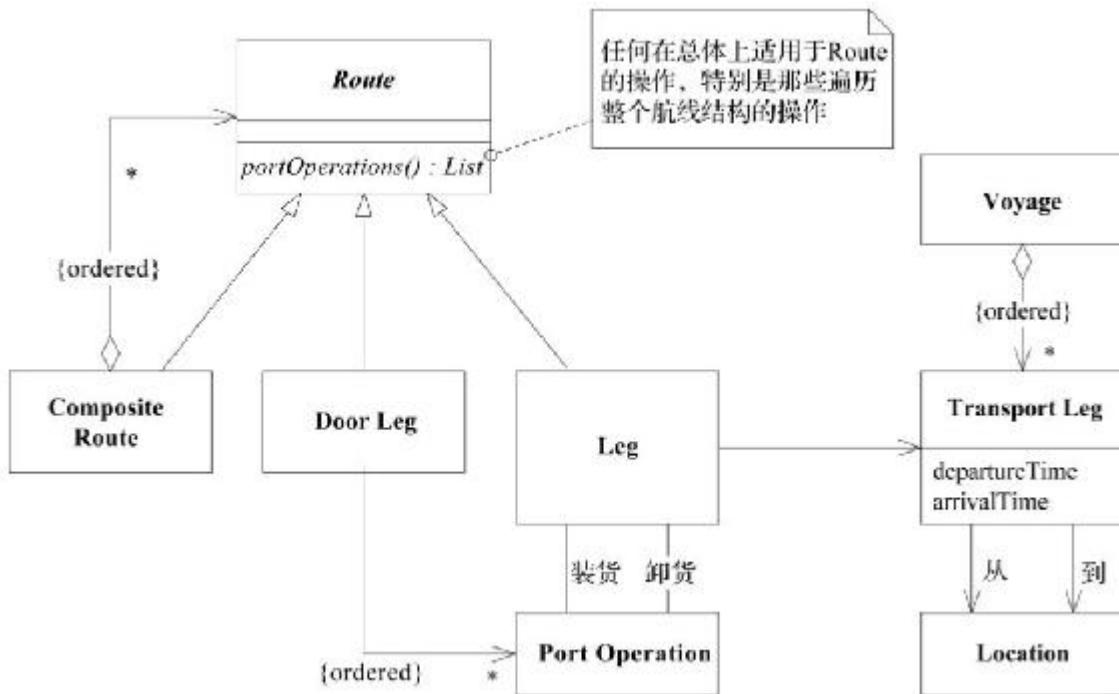


圖12-8 使用COMPOSITE之後的類圖

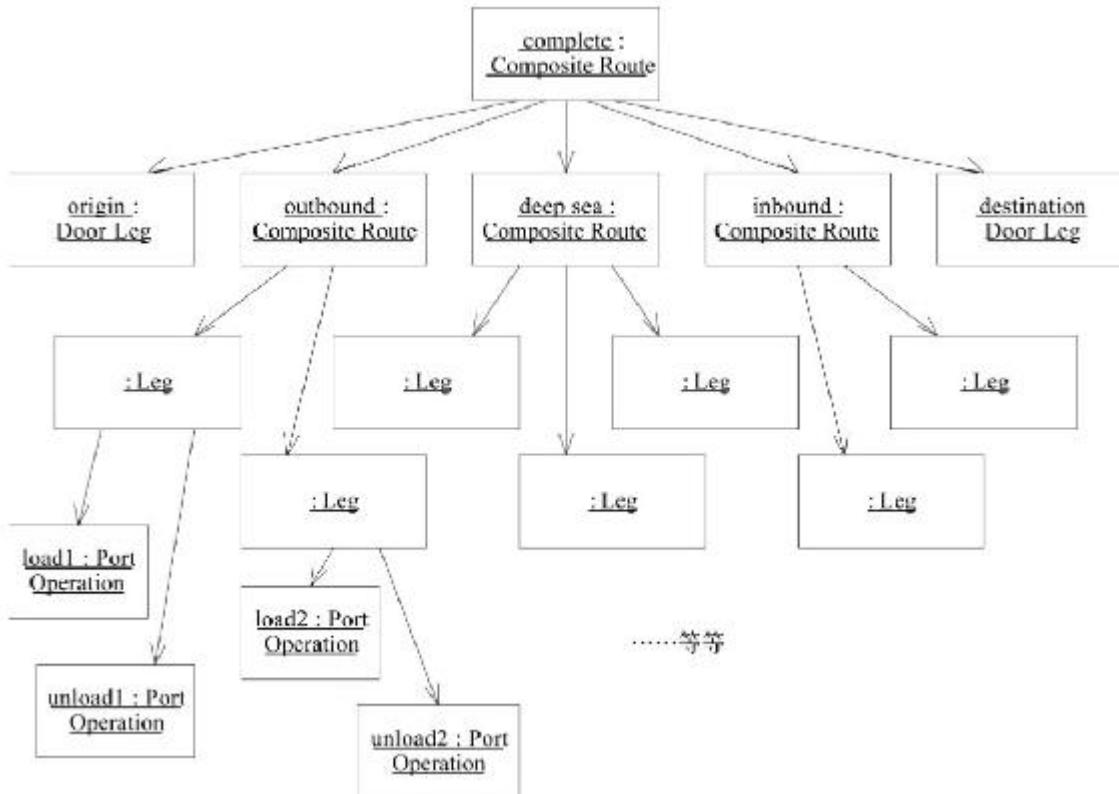


圖12-9 表示了一個完整Route的實例

## 12.3 為什麼沒有介紹FLYWEIGHT

由於第5章中提到過FLYWEIGHT模式，因此你可能認為它是一種適用於領域模型的模式。事實上，FLYWEIGHT雖然是設計模式的一個典型的例子，卻並不適用於領域模型。

當一個VALUE OBJECT集合（其中的值對像數目有限）被多次使用的時候（如房屋規劃中電源插座的例子），那麼把它們實現為FLYWEIGHT可能是有意義的。這是一個適用於VALUE OBJECT（但不適用於ENTITY）的實現選擇。COMPOSITE模式與它的不同之處在於，組合模式的概念對象是由其他概念對像組成的。這使得組合模式既適用於模型，也適用於實現，這是領域模式的一個基本特徵。

我並不打算把那些可以當作領域模式使用的設計模式完整地列出來。雖然我想不出一個把「解釋器」（interpreter）用作領域模式的例子，但我也不能斷言解釋器不適用於任何一種領域概念。把設計模式用作領域模式的唯一要求是這些模式能夠描述關於概念領域的一些事情，而不僅僅是作為解決技術問題的技術解決方案。

# 第13章 通過重構得到更深層的理解

通過重構得到更深層的理解是一個涉及很多方面的過程。我們有必要暫停一下，把一些要點歸納到一起。有三件事情是必須要關注的：

- (1) 以領域為本；
- (2) 用一種不同的方式來看待事物；
- (3) 始終堅持與領域專家對話。

在尋求理解領域的過程中，可以發現更廣泛的重構機會。

一提到傳統意義上的重構，我們頭腦中就會出現這樣一幅場景：一兩位開發人員坐在鍵盤前面，發現一些代碼可以改進，然後立即動手修改代碼（當然還要用單元測試來驗證結果）。這個過程應該一直進行下去，但它並不是重構過程的全部。

前面5章內容在傳統代碼重構方法的基礎上呈現了一幅全面的重構視圖。

## 13.1 開始重構

獲得深層理解的重構可能出現在很多方面。一開始有可能是為瞭解決代碼中的問題——一段複雜或笨拙的代碼。但開發人員並沒有使用（代碼重構所提供的）標準的代碼轉換，相反，他們認為問題的根源在於領域模型。或許是領域中缺少一個概念，或許是某個關係發生了錯誤。

與傳統重構觀點不同的是，即使在代碼看上去很整潔的時候也可能需要重構，原因是模型的語言沒有與領域專家保持一致，或者新需求不能被自然地添加到模型中。重構的原因也可能來自學習：當開發人員通過學習獲得了更深刻的理解，從而發現了一個得到更清晰或更有用的模型的機會。

如何找到問題的病灶往往是最難和最不確定的部分。在這之後，開發人員就可以系統地找出新模型的元素。他們可以與同事和領域專家一起進行頭腦風暴，也可以充分利用那些已經對知識做了系統性總結的分析模式或設計模式。

## 13.2 探索團隊

不管問題的根源是什麼，下一步都是要找到一種能夠使模型表達變得更清楚和更自然的改進方案。這可能只需要做一些簡單、明顯的修改，只需幾小時即可完成。在這種情況下，所做的修改類似於傳統重構。但尋找新模型可能需要更多時間，而且需要更多人參與。

修改的發起者會挑選幾位開發人員一起工作，這些開發人員應該擅長思考該類問題，瞭解領域，或者掌握深厚的建模技巧。如果涉及一些難以捉摸的問題，他們還要請一位領域專家加入。這個由4~5人組成的小組會到會議室或咖啡廳進行頭腦風暴，時間為半小時至一個半小時。在這個過程中，他們畫一些UML草圖，並試著用對像來走查場景。他們必須保證主題專家（subject matter expert）能夠理解模型並認為模型有用。當發現了一些令他們滿意的新思路後，他們就回去編碼，或者決定再多考慮幾天，先回去做點別的事情。幾天之後，這個小組再次碰頭，重複上面的過程。這時，他們已經對前幾天的想法

有了更深入的理解，因此更加自信了，並且得出了一些結論。他們回到計算機前，開始對新設計進行編碼。

要想保證這個過程的效率，需要注意幾個關鍵事項。

自主決定。可以隨時組成一個小的團隊來研究某個設計問題。這個團隊只工作幾天，然後就可以解散了。這種團隊沒有長期存在的必要，也不必有複雜的組織結構。

注意範圍和休息。在幾天內召開兩三次短會就應該能夠產生一個值得嘗試的設計。工作拖得太長並沒什麼好處。如果討論毫無進展，可能是一次討論的內容太多了。選一個較小的設計方面，集中討論它。

練習使用UBIQUITOUS LANGUAGE。讓其他團隊成員（特別是主題專家）參與頭腦風暴會議是練習和精化UBIQUITOUS LANGUAGE的好機會。這樣，原來的開發人員可以得到更完善的UBIQUITOUS LANGUAGE，並反映到編碼中。

本書前面幾章曾介紹過開發人員和領域專家為了設計更好的模型而進行的幾段對話。成熟的頭腦風暴是靈活機動、不拘泥於形式的，而且具有令人難以置信的高效率。

### **13.3 借鑒先前的經驗**

我們沒有必要總去做一些無謂的重複工作。用於查找缺失概念或改進模型的頭腦風暴過程具有巨大的作用，通過這個過程可以收集來自各個方面的想法，並把這些想法與已有知識結合起來。隨著知識消化的不斷開展，就能找到當前問題的答案。

我們可以從書籍和領域自身的其他知識源獲得思路。儘管相關領域的人員可能還沒有創建出適合運行軟件的模型，但他們可能已經把

概念很好地組織到了一起，並發現了一些有用的抽象。把這些知識結合到知識消化過程中，可以更快速地得到更豐富的結果，而且這個結果也更為領域專家們所熟悉。

有時我們可以從分析模式中汲取他人的經驗。這些經驗對於幫助我們讀懂領域起到了一定的作用，但分析模式是專門針對軟件開發的，因此應該直接根據我們自己在領域中實現軟件的經驗來利用這些模式。分析模式可以提供精細的模型概念，並幫助我們避免很多錯誤。但它們並不是現成的「菜譜」。它們只是為知識消化過程提供了一些供給。

隨著零散知識的歸納，必須同時處理模型關注點和設計關注點。同樣，這並不意味著總是需要從頭開發一切。當設計模式既符合實現需求，又符合模型概念時，通常就可以在領域層中應用這些模式。

同樣，當一種常見的形式體系（如算術邏輯或謂詞邏輯）與領域的某個部分非常符合時，可以把這個部分提取出來，並根據它來修改形式系統的規則。這可以產生非常簡練且易於理解的模型。

## 13.4 針對開發人員的設計

軟件不僅僅是為用戶提供的，也是為開發人員提供的。開發人員必須把他們編寫的代碼與系統的其他部分集成到一起。在迭代過程中，開發人員反覆修改代碼。開發人員應該通過重構得到更深層的理解，這樣既能夠實現柔性設計，也能夠從這樣一個設計中獲益。

柔性設計能夠清楚地表明它的意圖。這樣的設計使人們很容易看出代碼的運行效果，因此也很容易預計修改代碼的結果。柔性設計主要通過減少依賴性和副作用來減輕人們的思考負擔。這樣的設計是以深層次的領域模型為基礎的，在模型中，只有那些對用戶最重要的部

分才具有較細的粒度。在這樣的模型中，那些經常需要修改的地方能夠保持很高的靈活性，而其他地方則相對比較簡單。

## 13.5 重構的時機

如果一直等到完全證明瞭修改的合理性之後才去修改，那麼可能要等待太長時間了。項目正在承受巨大的耗支，推遲修改將使修改變得更難執行，因為要修改的代碼已經變得更加複雜，並更深地嵌入到其他代碼中。

持續重構漸漸被認為是一種「最佳實踐」，但大部分項目團隊仍然對它抱有很大的戒心。人們雖然看到了修改代碼會有風險，還要花費開發時間，但卻不容易看到維持一個拙劣設計也有風險，而且遷就這種設計也要付出代價。想要重構的開發人員往往被要求證明其重構的合理性。雖然這看似合理，但這使得一個本來就很難進行的工作變得幾乎不可能完成，而且會限制重構的進行（或者人們只能暗地裡進行）。軟件開發並不是一個可以完全預料到後果的過程，人們無法準確地計算出某個修改會帶來哪些好處，或者是不做某個修改會付出多大代價。

在探索領域的過程中、在培訓開發人員的過程中，以及在開發人員與領域專家進行思想交流的過程中，必須始終堅持把「通過重構得到更深層理解」作為這些工作的一部分。因此，當發生以下情況時，就應該進行重構了：

設計沒有表達出團隊對領域的最新理解；

重要的概念被隱藏在設計中了（而且你已經發現了把它們呈現出來的方法）；

發現了一個能令某個重要的設計部分變得更靈活的機會。

我們雖然應該有這樣一種積極的態度，但並不意味著可以隨隨便便做任何修改。在發佈的前一天，就不要進行重構了。不要引入一些只顧炫耀技術能力而沒有解決領域核心問題的「柔性設計」。無論一個「更深層的模型」看起來有多好，如果你不能說服領域專家們去使用它，那麼就不要引入它。萬事都不是絕對的，但如果某個重構對我們有利，那麼不妨在這個方向上大膽前進。

## **13.6 危機就是機遇**

在達爾文創立進化論後的一個多世紀中，人們一直認為標準的進化模型就是物種隨著時間緩慢地改變（在一定程度上這種改變是穩定的）。突然之間，這個模型在 20 世紀 70 年代被「間斷平衡」（punctuated equilibrium）模型取代了。它對原有進化論進行了擴展，認為長期的緩慢變化或穩定變化會被相對來說很短的、爆發性的快速變化所打斷。然後事物會進入一個新的平衡。軟件開發與物種進化之間的不同點是前者具有明確的方向（雖然在某些項目上可能並不明顯），儘管如此軟件開發仍遵循這種進化規律。

傳統意義上的重構聽起來是一個非常穩定的過程。但通過重構得到更深層理解往往不是這樣的。在對模型進行一段時間穩定的改進後，你可能突然有所頓悟，而這會改變模型中的一切。這些突破不會每天都發生，然而很大一部分深層模型和柔性設計都來自這些突破。

這樣的情況往往看起來不像是機遇，而更像危機。例如，你突然發現模型中有一些明顯的缺陷，在表達方面顯示出一個很大的漏洞，或存在一些沒有表達清楚的關鍵區域。或者有些描述是完全錯誤的。

這些都表明團隊對模型的理解已經達到了一個新的水平。他們現在站在更高的層次上發現了原有模型的弱點。他們可以從這種角度構

思一個更好的模型。

通過重構得到更深層理解是一個持續不斷的過程。人們發現一些隱含的概念，並把它們明確地表示出來。有些設計部分變得更具有柔性，或許還採用了聲明式的風格。開發工作一下子到了突破的邊緣，然後開發人員跨越這條界線，得到了一個更深層的模型，接下來又重新開始了穩步的改進過程。

---

[1].Grmma等人在[Grmma et al.1995]中簡要提及了應該將模式作為重構的目標，而Joshua Kerievsky則把「通過重構得到模式」發展為更加成熟實用的形式 [Kerievsky 2003]。

[2].借貸公司 ( lending company ) 對於借款者 ( borrower ) 而言是放貸方 ( lender )，對於銀團而言是投資者 ( investor )。——編者注

[3].組合爆炸 ( combinatory explosion )，源自離散數學的術語，是指隨著問題中元素的增加，所出現的可能組合數劇烈增加。——譯者注

[4].Ward Cunningham提出的WHOLE VALUE模式。

## 第四部分 戰略設計

隨著系統的增長，它會變得越來越複雜，當我們無法通過分析對像來理解系統的時候，就需要掌握一些操縱和理解大模型的技術了。本書的這一部分將介紹一些原則。遵循這些原則，就可以對非常複雜的領域進行建模。大部分這樣的決策都需要由團隊來制定，甚至需要多個團隊共同協商制定。這些決策往往是把設計和策略綜合到一起的結果。

最負雄心的企業系統意欲實現一個涵蓋所有業務、緊密集成的系統。然而在幾乎所有這種規模的組織中，整體業務模型太大也太複雜了，因此難以管理，甚至很難把它作為一個整體來理解。我們必須在概念和實現上把系統分解為較小的部分。但問題在於，如何保證實現這種模塊化的同時，不失去集成所具備的好處；從而使系統的不同部分能夠進行互操作，以便協調各種業務操作。如果設計一個把所有概念都涵蓋進來的單一領域模型，它將會非常笨拙，而且將會出現大量難以察覺的重複和矛盾。而如果用臨時拼湊的接口把一組小的、各自不同的子系統集成到一起，又不具備解決企業級問題的能力，並且在每個集成點上都有可能出現不一致。通過採用系統的、不斷演變的設計策略，就可以避免這兩種極端問題。

即使在這種規模的系統中採用領域驅動設計方法，也不要脫離實現去開發模型。每個決策都必須對系統開發產生直接的影響，否則它就是無關的決策。戰略設計原則必須指導設計決策，以便減少各個部

分之間的互相依賴，在使設計意圖更為清晰的同時而又不失去關鍵的互操作性和協同性。戰略設計原則必須把模型的重點放在捕獲系統的概念核心，也就是系統的「遠景」上。而且在完成這些目標的同時又不能為項目帶來麻煩。為了幫助實現這些目標，這一部分探索了3個大的主題：上下文、精煉和大型結構。

其中上下文是最不易引起注意的原則，但實際上它卻是最根本的。無論大小，成功的模型必須在邏輯上一致，不能有互相矛盾或重疊的定義。有時，企業系統會集成各種不同來源的子系統，或包含諸多完全不同的應用程序，以至於無法從同一個角度來看待領域。要把這些不同部分中隱含的模型統一起來可能是要求過高了。通過為每個模型顯式地定義一個**BOUNDED CONTEXT**，然後在必要的情況下定義它與其他上下文的關係，建模人員就可以避免模型變得纏雜不清。

通過精煉可以減少混亂，並且把注意力集中到正確的地方。人們通常在領域的一些次要問題上花費了太多的精力。整體領域模型需要突出系統中最有價值和最特殊的那些方面，而且在構造領域模型時應該盡可能把注意力集中在這些部分上。雖然一些支持組件也很關鍵，但絕不能把它們和領域核心一視同仁。把注意力集中到正確的地方不僅有助於把精力投入到關鍵部分上，而且還可以使系統不會偏離預期方向。戰略精煉可以使大的模型保持清晰。有了更清晰的視圖後，**CORE DOMAIN**的設計就會發揮更大的作用。

大型結構是用來描述整個系統的。在非常複雜的模型中，人們可能會「只見樹木，不見森林」。精煉確實有幫助，它使人們能夠把注意力集中到核心元素上，並把其他元素表示為支持作用，但如果不能貫徹某個主旨來應用一些系統級的設計元素和模式的話，關係仍然可能非常混亂。我將概要介紹幾種大型結構方法，然後詳細討論其中一種模式——**RESPONSIBILITY LAYER**（職責層），通過這個示例來探索使

用大型結構的含義。我們所討論的特殊結構只是一些例子，它們並不是大型結構的全部。當需要的時候，應該創造新的結構，抑或修改這些結構，但均需遵循演化順序（EVOLVING ORDER）的過程來進行。一些大型結構能夠使設計保持一致性，從而加速開發，並提高集成度。

這3種原則各有各的用處，但結合起來使用將發揮更大的力量，遵守這些原則就可以創建出好的設計，即使是對一個非常龐大的沒有人能夠完全理解的系統也是如此。大型結構能夠保持各個不同部分之間的一致性，從而有助於這些部分的集成。結構和精煉能夠幫助我們理解各個部分之間的複雜關係，同時保持整體視圖的清晰。**BOUNDED CONTEXT**使我們能夠在不同的部分中進行工作，而不會破壞模型或是無意間導致模型的分裂。把這些概念加進團隊的**UBIQUITOUS LANGUAGE**中，可以幫助開發人員找出他們自己的解決方案。

## 第14章 保持模型的完整性

我曾經參加過一個項目，在這個項目中幾個團隊同時開發一個重要的新系統。有一天，當負責「客戶發票」模塊的團隊正準備實現一個他們稱之為Charge（收費）的對象時，他們發現另一個團隊已經構建了這個對象，於是決定重複使用這個現有對象。他們發現它沒有expense code（費用代碼）屬性，因此添加了一個。對像中有一個posted amount（過帳金額）屬性是他們所需要的。他們本來計劃把這個屬性叫做amount due（到期金額），但名稱不同有什麼關係呢？於是他們把名稱改成了「posted amount」。又添加了幾個方法和關聯後，他們得到了所需的對象，而且沒有擾亂任何事情。雖然他們必須忽略掉一些不需要的關聯，但他們的模塊運行很正常。

幾天之後，「賬單支付」模塊出現了一些奇怪的問題（Charge對像最初就是為這個模塊編寫的）。系統中出現了一些奇怪的Charge，沒有人記得曾經輸入過它們，而且它們也沒有任何意義。當使用某些函數時，特別是使用當月月初至今（month-to-date）的稅務報表時，程序就會崩潰。調查發現，當用於計算所有當月付款的可扣除總額的函數被調用時，程序就會崩潰。那些來歷不明的記錄在percent deductible（可扣除百分比）字段中沒有值，儘管數據錄入應用程序的驗證需要這個值，甚至為它設置了一個默認值。

問題在於這兩個團隊使用了不同的模型，而他們並沒有認識到這一點，也沒有用於檢測這一問題的過程。每個團隊都對Charge對象的特性做了一些假設，使之能夠在自己的上下文中使用（一個是向客戶

收費，另一個是向供應商付款）。當他們的代碼被組合到一起而沒有消除這些矛盾時，結果就產生了不可靠的軟件。

如果他們一開始就意識到這一點，就能決定如何來解決它。他們可以共同開發出一個公共的模型，然後編寫自動測試套件來防止以後出現意外。也可以雙方商定開發各自的模型，而互相不幹擾對方的代碼。無論採用哪種方法，首先都要明確邊界，各模型只在各自的邊界內使用。

他們在知道了問題所在之後採取了什麼措施呢？他們創建了兩個不同的類：**Customer Charge**（客戶收費）類和**Supplier Charge**（供應商收費）類。並根據各自的需求定義了每個類。解決了眼前這個問題之後，他們又按以前的方式開始工作了。

模型最基本的要求是它應該保持內部一致，術語總具有相同的意義，並且不包含互相矛盾的規則：雖然我們很少明確地考慮這些要求。模型的內部一致性又叫做統一（*unification*），這種情況下，每個術語都不會有模稜兩可的意義，也不會有規則衝突。除非模型在邏輯上是一致的，否則它就沒有意義。在理想世界中，我們可以得到涵蓋整個企業領域的單一模型。這個模型將是統一的，沒有任何相互矛盾或相互重疊的術語定義。每個有關領域的邏輯聲明都是一致的。

但大型系統開發並非如此理想。在整個企業系統中保持這種水平的統一是一件得不償失的事情。在系統的各個不同部分中開發多個模型是很有必要的，但我們必須慎重地選擇系統的哪些部分可以分開，以及它們之間是什麼關係。我們需要用一些方法來保持模型關鍵部分的高度統一。所有這些都不會自行發生，而且光有良好的意願也是沒用的。它只有通過有意識的設計決策和建立特定過程才能實現。大型系統領域模型的完全統一即不可行，也不划算。

有時人們會反對這一點。大多數人都看到了多個模型的代價：它們限制了集成，並且使溝通變得很麻煩。更重要的是，多個模型看上去似乎不夠雅緻。有時，對多個模型的抵觸會導致「極富雄心」的嘗試——將一個大型項目中的所有軟件統一到單一模型中。我自己就很後悔曾經這麼做過了頭。但請一定要考慮下面的風險。

- (1) 一次嘗試對遺留系統做過多的替換。
- (2) 大項目可能會陷入困境，因為協調的開銷太大，超出了這些項目的能力範圍。
- (3) 具有特殊需求的應用程序可能不得不使用無法充分滿足需求的模型，而只能將這些無法滿足的行為放到其他地方。
- (4) 另一方面，試圖用一個模型來滿足所有人的需求可能會導致模型中包含過於複雜的選擇，因而很難使用。

此外，除了技術上的因素以外，權力上的劃分和管理級別的不同也可能要求把模型分開。而且不同模型的出現也可能是團隊組織和開發過程導致的結果。因此，即使完全的集成沒有來自技術方面的阻力，項目也可能會面臨多個模型。

既然無法維護一個涵蓋整個企業的統一模型，那就不要再受到這種思路的限制。通過預先決定什麼應該統一，並實際認識到什麼不能統一，我們就能夠創建一個清晰的、共同的視圖。確定了這些之後，就可以著手開始工作，以保證那些需要統一的部分保持一致，不需要統一的部分不會引起混亂或破壞模型。

我們需要用一種方式來標記出不同模型之間的邊界和關係。我們需要有意識地選擇一種策略，並一致地遵守它。

本章將介紹一些用於識別、溝通和選擇模型邊界及關係的技術。討論首先從描繪項目當前的範圍開始。**BOUNDED CONTEXT**（限界上下文）定義了每個模型的應用範圍，而**CONTEXT MAP**（上下文圖）則

給出了項目上下文以及它們之間關係的總體視圖。這些降低模糊性的技術能夠使項目更好地進行，但僅僅有它們還是不夠的。一旦確立了 CONTEXT 的邊界之後，仍需要持續集成這種過程，它能夠使模型保持統一。

其後，在這個穩定的基礎之上，我們就可以開始實施那些在界定和關聯 CONTEXT 方面更有效的策略了—從通過共享內核（ SHARED KERNEL ）來緊密關聯上下文，到那些各行其道（ SEPARATE WAYS ）地進行鬆散耦合的模型。

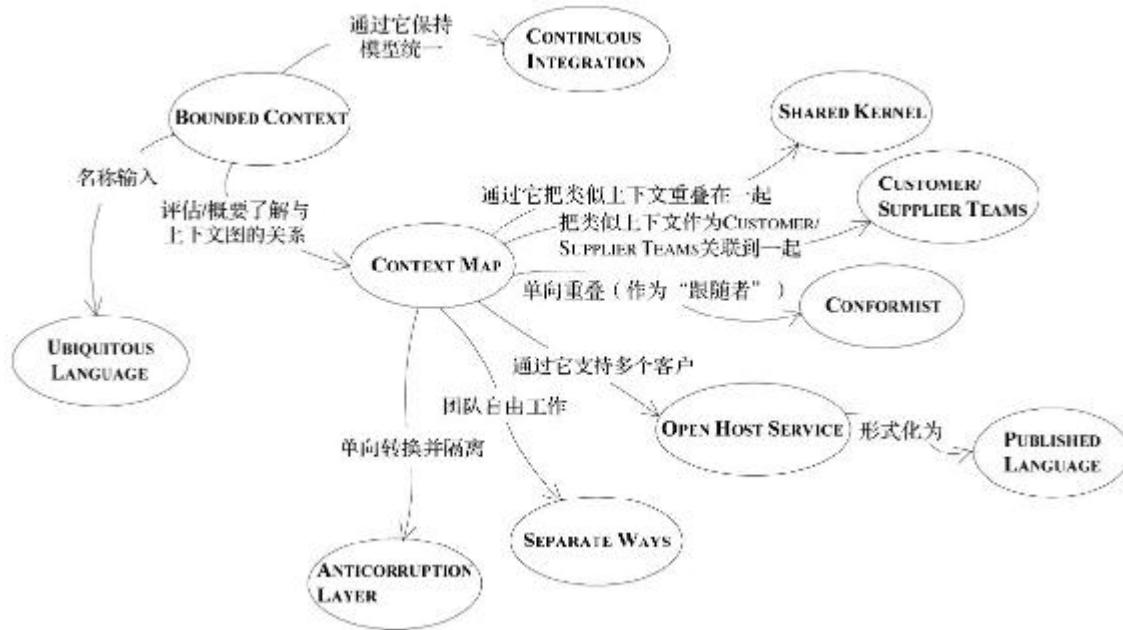
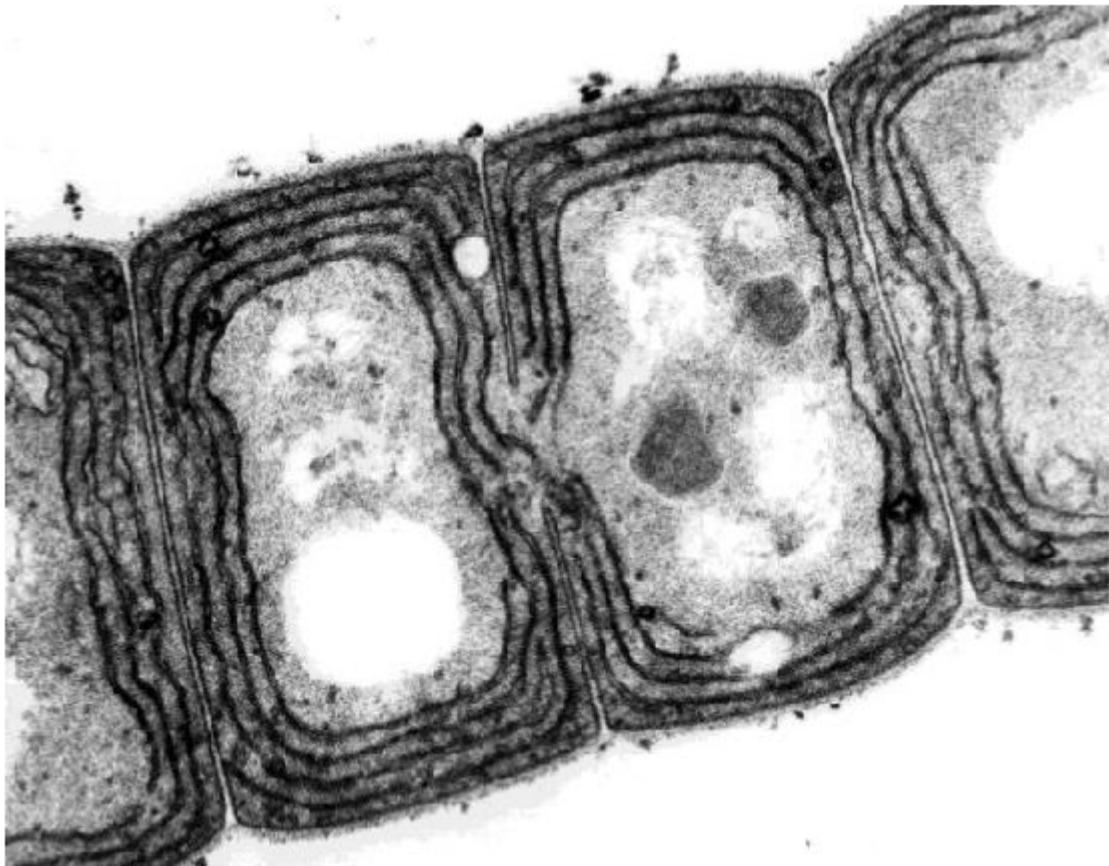


圖14-1 模型完整性模式的導航圖

## 14.1 模式：**BOUNDED CONTEXT**



細胞之所以能夠存在，是因為細胞膜限定了什麼在細胞內，什麼在細胞外，並且確定了什麼物質可以通過細胞膜

大型項目上會有多個模型共存，在很多情況下這沒什麼問題。不同的模型應用於不同的上下文中。例如，你可能必須將你的新軟件與一個外部系統集成，而你的團隊對這個外部系統沒有控制權。在這種情況下，任何人都明白這個外部系統是一種完全不同的上下文，不適用他們正在開發的模型，但還有很多情況是比較含糊和混亂的。在本章開篇所講的那個故事中，兩個團隊為同一個新系統開發不同的功能。那麼他們使用的是同一個模型嗎？他們的意圖是至少共享其所做的一部分工作，但卻沒有界限告訴他們共享了什麼、沒有共享什麼。而且他們也沒有一個過程來維持共享模型，或快速檢測模型是否有分

歧。他們只是在系統行為突然變得不可預測時才意識到他們之間產生了分歧。

即使在同一個團隊中，也可能會出現多個模型。團隊的溝通可能會不暢，導致對模型的理解產生難以捉摸的衝突。原先的代碼往往反映的是早先的模型概念，而這些概念與當前模型有著微妙的差別。

每個人都知道兩個系統的數據格式是不同的，因此需要進行數據轉換，但這只是問題的表面。問題的根本在於兩個系統所使用的模型不同。當這種差異不是來自外部系統，而是發生在同一個系統中時，它將更難發現。然而，所有大型團隊項目都會發生這種情況。

任何大型項目都會存在多個模型。而當基於不同模型的代碼被組合到一起後，軟件就會出現**bug**、變得不可靠和難以理解。團隊成員之間的溝通變得混亂。人們往往弄不清楚一個模型不應該在哪個上下文中使用。

模型混亂的問題最終會在代碼不能正常運行時暴露出來，但問題的根源卻在於團隊的組織方式和成員的交流方法。因此，為了澄清模型的上下文，我們既要注意項目，也要注意它的最終產品（代碼、數據庫模式等）。

一個模型只在一個上下文中使用。這個上下文可以是代碼的一個特定部分，也可以是某個特定團隊的工作。如果模型是在一次頭腦風暴會議中得到的，那麼這個模型的上下文可能僅限於那次討論。就拿本書中的例子來說，示例中所使用的模型的上下文就是那個示例所在的小節以及任何相關的後續討論。模型上下文是為了保證該模型中的術語具有特定意義而必須要應用的一組條件。

為瞭解決多個模型的問題，我們需要明確地定義模型的範圍——模型的範圍是軟件系統中一個有界的部分，這部分只應用一個模型，並盡可能使其保持統一。團隊組織中必須一致遵守這個定義。

因此：

明確地定義模型所應用的上下文。根據團隊的組織、軟件系統的各個部分的用法以及物理表現（代碼和數據庫模式等）來設置模型的邊界。在這些邊界中嚴格保持模型的一致性，而不要受到邊界之外問題的幹擾和混淆。

BOUNDED CONTEXT明確地限定了模型的應用範圍，以便讓團隊成員對什麼應該保持一致以及上下文之間如何關聯有一個明確和共同的理解。在CONTEXT中，要保證模型在邏輯上統一，而不用考慮它是不是適用於邊界之外的情況。在其他CONTEXT中，會使用其他模型，這些模型具有不同的術語、概念、規則和UBIQUITOUS LANGUAGE的技術行話。通過劃定明確的邊界，可以使模型保持純粹，因而在它所適用的CONTEXT中更有效。同時，也避免了將注意力切換到其他CONTEXT時引起的混淆。跨邊界的集成必然需要進行一些轉換，但我們可以清楚地分析這些轉換。

### **BOUNDED CONTEXT不是MODULE**

有時這兩個概念易引起混淆，但它們是具有不同動機的不同模式。確實，當兩組對像組成兩個不同模型時，人們幾乎總是把它們放在不同的MODULE中。這樣做的確提供了不同的命名空間（對不同的CONTEXT很重要）和一些劃分方法。

但人們也會在一個模型中用MODULE來組織元素，它們不一定要表達劃分CONTEXT的意圖。MODULE在BOUNDED CONTEXT內部創建的獨立命名空間實際上使人們很難發現意外產生的模型分裂。

### **示例 預訂系統的上下文**

一家運輸公司的內部項目——為貨物預訂開發一個新的應用程序。這個應用由一個對像模型驅動。那麼這個模型所應用的

**BOUNDED CONTEXT**是什麼呢？為了回答這個問題，我們必須看一下項目正在發生的事情。記住，這裡是觀察項目的現狀，而不是它的理想狀態。

預訂應用程序的開發工作由一個項目團隊負責。他們不能修改模型對象，但他們所構建的應用程序還必須要顯示和操作這些對象。這個團隊是模型的使用者。模型在應用程序（模型的主要使用者）中是有效的，因此預訂應用程序在**BOUNDED CONTEXT**的邊界之內。

已完成的預訂必須傳遞給用於貨物跟蹤的遺留系統來處理。項目一開始就已決定新模型將與原有系統的模型不同，因此原來的貨物跟蹤系統位於**BOUNDED CONTEXT**的邊界之外。新舊模型之間的必要轉換由原有系統的維護團隊來負責處理。轉換機制不是由新模型驅動的。因此它不在**BOUNDED CONTEXT**中（轉換其實是邊界本身的一部分，這一點將在**CONTEXT MAP**中討論）。將轉換機制置於**CONTEXT**之外（不基於模型），這一點很好。要求遺留系統的團隊使用這個模型是不切實際的，因為他們的主要工作都發生在**CONTEXT**之外。

每個對象的整個生命週期都由負責模型的團隊來處理，包括對象的持久化。由於這個團隊也控制著數據庫模式，因此他們特意把對像一關係映射設計得簡單直接。換言之，數據庫模式是由新模型驅動的，因此在**BOUNDED CONTEXT**的邊界之內。

另有一個團隊正在開發安排貨輪航次的模型和應用。從項目一開始，這個團隊與負責貨物預訂的團隊就在一起工作，他們都打算開發一個單獨的、統一的系統。這兩支團隊偶爾互相協調，也偶爾共享對象，但沒有系統性地去做。他們不在同一個**BOUNDED CONTEXT**中工作。這會帶來風險，因為他們並沒有意識到各自正在使用不同的模型。到了集成的時候，就會出現問題，除非他們採取特定的過程來管理這種情況（共享內核可能就是一個很好的選擇，本章後面會介

紹）。但是，第一步是認清現狀。他們不在同一個CONTEXT中，因此應該停止共享代碼，直到做出一些改變之後再去共享。

在這個系統中，由該具體模型驅動的所有方方面面構成了其對應的BOUNDED CONTEXT，這包括模型對像、用於模型對像持久化的數據庫模式以及預訂應用程序。在這個CONTEXT中主要有兩支團隊在工作，一個是建模團隊，另一個是應用程序團隊。這個系統需要與遺留的貨物跟蹤系統交換信息，遺留系統的維護團隊主要負責在這個邊界上的轉換，並且與建模團隊進行合作。預訂模型和航次安排模型之間沒有明確定義的關係，定義這種關係應該是這兩個團隊的首要任務之一。同時，他們應該在共享代碼或數據方面格外謹慎。

因此，通過定義這個BOUNDED CONTEXT，最終得到了什麼？對CONTEXT內的團隊而言：清晰！。這兩支團隊知道他們必須與這個模型保持一致。他們根據這一點制定設計決策，並注意防範出現不一致的情況。而CONTEXT之外的團隊獲得了：自由。他們不必行走在灰色地帶，不必使用同一個模型，雖然他們還是總覺得應該使用同一個模型。但在這個具體例子中，最實際的收穫是認識到了在預訂模型團隊和航次安排團隊之間進行信息共享存在著風險。為了避免問題產生，他們實際上需要在共享的代價和收益之間作出權衡，並制定流程來確保其發揮作用。只有每個人都理解模型上下文的邊界在哪裡，這一切才會發生。

當然，邊界只不過是一些特殊的位臘。各個BOUNDED CONTEXT之間的關係需要我們仔細地處理。CONTEXT MAP畫出了上下文的範圍，並給出了CONTEXT以及它們之間聯繫的總體視圖，而幾種模式定義了CONTEXT之間的各種關係的性質。CONTINUOUS INTEGRATION的過程可以使模型在BOUNDED CONTEXT中保持統一。

但在討論所有這些模式之前，想一想當模型的統一性被破壞時，模型會是什麼樣子呢？我們又該如何識別概念上的不一致呢？

### 識別**BOUNDED CONTEXT**中的不一致

很多徵兆都可能表明模型中出現了差異。最明顯的是已編碼的接口不匹配。對於更微妙的情況，一些意外行為也可能是一種信號。採用了自動測試的**CONTINUOUS INTEGRATION**可以幫助捕捉到這類問題。但語言上的混亂往往是一種早期的警告信號。

將不同模型的元素組合到一起可能會引發兩類問題：重複的概念和假同源。重複的概念是指兩個模型元素（以及伴隨的實現）實際上表示同一個概念。每當這個概念的信息發生變化時，都必須更新兩個地方。每次由於新知識導致一個對像被修改時，必須重新分析和修改另一個對象。如果不進行實際的重新分析，結果就會出現同一概念的兩個版本，它們遵守不同的規則，甚至有不同的數據。更嚴重的是，團隊成員必須學習做同一件事情的兩種方法，以及保持這兩種方法同步的各種方式。

假同源可能稍微少見一點，但它潛在的危害更大。它是指使用相同術語（或已實現的對象）的兩個人認為他們是在談論同一件事情，但實際上並不是這樣。本章開頭的示例就是一個典型的例子（兩個不同的業務活動都叫做Charge）。但是，當兩個定義都與同一個領域方面相關，而只是在概念上稍有區別時，這種衝突更難以發現。假同源會導致開發團隊互相干擾對方的代碼，也可能導致數據庫中含有奇怪的矛盾，還會引起團隊溝通的混淆。假同源這個術語在自然語言中也經常使用。例如，說英語的人在學習西班牙語時常常會誤用embarazada這個詞。這個詞的意思並不是embarrassed（難堪的），而是pregnant（懷孕的）。很驚訝吧！

當發現這些問題時，團隊必須要做出相應的決定。可能需要將模型重新整合為一體，並加強用來預防模型分裂的過程。分裂也有可能是由分組造成的，一些小組出於合理的原因，需要以一些不同的方式來開發模型，而且你可能也決定讓他們獨立開發。本章接下來要討論的模式的主題就是如何解決這些問題。

## **14.2 模式：CONTINUOUS INTEGRATION**



定義完一個**BOUNDED CONTEXT**後，必須讓它保持合理。

當很多人在同一個**BOUNDED CONTEXT**中工作時，模型很容易發生分裂。團隊越大，問題就越大，但即使是3、4個人的團隊也有可能會遇到嚴重的問題。然而，如果將系統分解為更小的**CONTEXT**，最終又難以保持集成度和一致性。

有時開發人員沒有完全理解其他人所創建的對象或交互的意圖，就對它進行了修改，使其失去了原來的作用。有時他們沒有意識到他們正在開發的概念已經在模型的另一個部分中實現了，從而導致了這些概念和行為（不正確的）重複。有時他們意識到了這些概念有其他的表示，但卻因為擔心破壞現有功能而不敢去改動它們，於是他們繼續重複開發這些概念和功能。

開發統一的系統（無論規模大小）需要維持很高的溝通水平，而這一點常常很難做到。我們需要運用各種方法來增進溝通並減小複雜性。還需要一些安全防護措施，以避免過於謹慎的行為（例如，開發人員由於擔心破壞現有代碼而重複開發一些功能）。

極限編程（XP）在這樣的環境中真正顯示出自己的強大威力。很多XP實踐都是針對在很多人頻繁更改設計的情況下如何維護設計的一致性這個特定問題而出現的。最純粹的XP非常適合維護單一 BOUNDED CONTEXT 中的模型完整性。但是，無論是否使用XP，都很有必要採取CONTINUOUS INTEGRATION過程。

CONTINUOUS INTEGRATION是指把一個上下文中的所有工作足夠頻繁地合併到一起，並使它們保持一致，以便當模型發生分裂時，可以迅速發現並糾正問題。像領域驅動設計中的其他方法一樣，CONTINUOUS INTEGRATION也有兩個級別的操作：(1) 模型概念的集成；(2) 實現的集成。

團隊成員之間通過經常溝通來保證概念的集成。團隊必須對不斷變化的模型形成一個共同的理解。有很多方法可以幫助做到這一點，但最基本的方法是對UBIQUITOUS LANGUAGE多加鍛煉。同時，實際工件通過系統性的合併/構建/測試過程來集成，這樣能夠盡早暴露出模型的分裂問題。用來集成的過程有很多，大部分有效的過程都具備以下這些特徵：

分步集成，採用可重現的合併/構建技術；  
自動測試套件；  
有一些規則，用來為那些尚未集成的改動設置一個相當小的生命期上限。

有效過程的另一面是概念集成，雖然它很少被正式地納入進來。在討論模型和應用程序時要堅持使用**UBIQUITOUS LANGUAGE**。

大多數敏捷項目至少每天會把每位開發人員所做的修改合併進來。這個頻率可以根據更改的步伐來調整，只要確保該間隔不會導致大量不兼容的工作產生即可。

在**MODEL-DRIVEN DESIGN**中，概念集成為實現集成鋪平了道路，而實現集成驗證了模型的有效性和一致性，並暴露出模型的分裂問題。

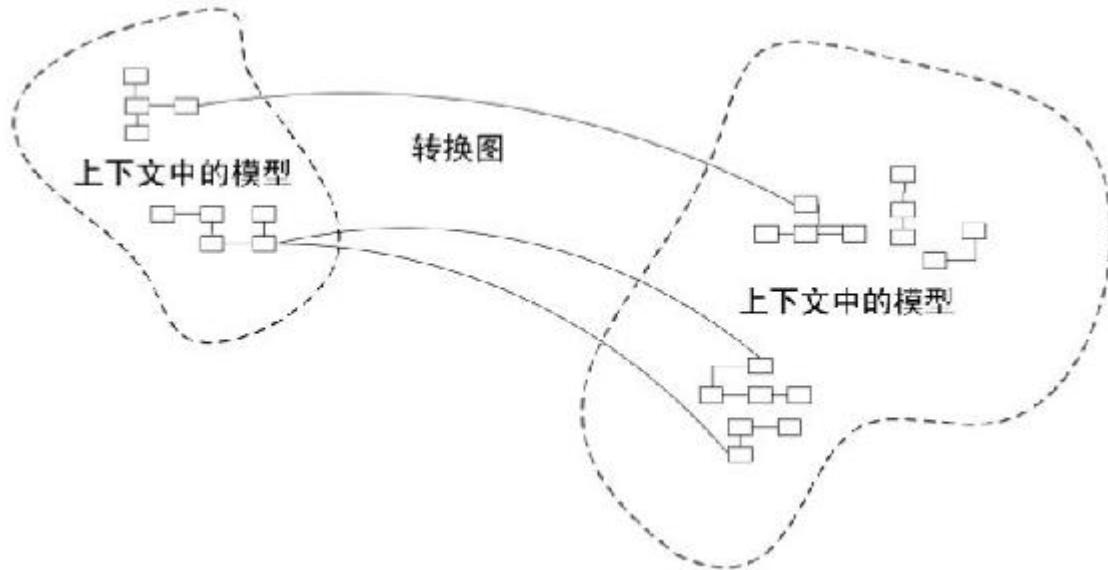
因此：

建立一個把所有代碼和其他實現工作頻繁地合併到一起的過程，並通過自動化測試來快速查明模型的分裂問題。嚴格堅持使用**UBIQUITOUS LANGUAGE**，以便在不同人的頭腦中演變出不同的概念時，使所有人對模型都能達成一個共識。

最後，不要在持續集成中做一些不必要的工作。**CONTINUOUS INTEGRATION**只有在**BOUNDED CONTEXT**中才是重要的。相鄰**CONTEXT**中的設計問題（包括轉換）不必以同一個步調來處理。

**CONTINUOUS INTEGRATION**可以在任何單獨的**BOUNDED CONTEXT**中使用，只要它的工作規模大到需要兩個以上的人去完成就可以。它可以維護單一模型的完整性。當多個**BOUNDED CONTEXT**共存時，我們必須要確定它們的關係，並設計任何必需的接口。

## 14.3 模式：CONTEXT MAP



只有一個 BOUNDED CONTEXT 並不能提供全局視圖。其他模型的上下文可能仍不清楚而且還在不斷變化。

其他團隊中的人員並不是十分清楚 **CONTEXT** 的邊界，他們會不知不覺地做出一些更改，從而使邊界變得模糊或者使互連變得複雜。當不同的上下文必須互相連接時，它們可能會互相重疊。

BOUNDED CONTEXT 之間的代碼重用是很危險的，應該避免。功能和數據的集成必須要通過轉換去實現。通過定義不同上下文之間的關係，並在項目中創建一個所有模型上下文的全局視圖，可以減少混亂。

CONTEXT MAP 位於項目管理和軟件設計的重疊部分。按照常規，人們往往按團隊組織的輪廓來劃定邊界。緊密協作的人會很自然地共享一個模型上下文。不同團隊的人員（或者在同一個團隊中但從不交流的人）將使用不同的上下文。辦公室的物理位臘也有影響，例如，分別位於大樓兩端的團隊成員（更不用說在不同城市工作的人了）如果沒有為整合做額外的工作，很有可能會使用不同的上下文。大多數項目經理會本能地意識到這些因素，並圍繞子系統大致把各個團隊組織起來。但團隊組織與軟件模型及設計之間的相互關係仍然不夠明

顯。對於軟件模型與設計的持續概念細分，項目經理和團隊成員需要一個清晰的視圖。

因此：

識別在項目中起作用的每個模型，並定義其 **BOUNDED CONTEXT**。這包括非面向對像子系統的隱含模型。為每個 **BOUNDED CONTEXT** 命名，並把名稱添加到 **UBIQUITOUS LANGUAGE** 中。

描述模型之間的聯繫點，明確所有通信需要的轉換，並突出任何共享的內容。

先將當前的情況描繪出來。以後再做改變。

在每個 **BOUNDED CONTEXT** 中，都將有一種一致的 **UBIQUITOUS LANGUAGE** 的「方言」。我們需要把 **BOUNDED CONTEXT** 的名稱添加到該方言中，這樣只要通過明確 **CONTEXT** 就可以清楚地討論任意設計部分的模型。

**CONTEXT MAP** 無需拘泥於任何特定的文檔格式。我發現類似本章的簡圖在可視化和溝通上下文圖方面很有幫助。有些人可能喜歡使用較多的文本描述或別的圖形表示。在某些情況下，團隊成員之間的討論就足夠了。需求不同，細節層次也不同。不管 **CONTEXT MAP** 採用什麼形式，它必須在所有項目人員之間共享，並被他們理解。它必須為每個 **BOUNDED CONTEXT** 提供一個明確的名稱，而且必須闡明聯繫點和它們的本質。

根據設計問題和項目組織問題的不同，**BOUNDED CONTEXT** 之間的關係有很多種形式。本章稍後將介紹 **CONTEXT** 之間的各種關係模式，這些模式分別適用於不同的情況，並且提供了一些術語，這些術語可以用來描述你自己的上下文圖中發現的關係。記住，**CONTEXT MAP** 始終表示它當前所處的情況，你所發現的關係一開始可能並不適

合這些模式。如果它們與某種模式非常接近，你可能想用這個模式名來描述它們，但不要生搬硬套。只需描述你所發現的關係即可。過後，你可以向更加標準化的關係過渡。

那麼，如果你發現模型產生了分裂——模型完全混亂且包含不一致時，你該怎麼辦呢？這時一定要十分注意，先把描述工作停下來。然後，從精確的全局角度來解決這些混亂點。小的分裂可以修復，並且可以通過實施一些過程來為修復提供支持。如果關係很模糊，可以選擇一種最接近的模式，然後向此模式靠攏。最重要的任務是畫出一個清晰的**CONTEXT MAP**，而這可能意味著修復實際發現的問題。但不要因為修復必要的問題而重組整個結構。我們只需修改那些明顯的矛盾即可，直到得出一個明確的**CONTEXT MAP**。在這個圖中，你的所有工作都被放到某個**BOUNDED CONTEXT**中，而且所有互連的模型都有明確的關係。

一旦獲得了一致的**CONTEXT MAP**，就會看到需要修改的那些地方。在經過深思熟慮後，你可以調整團隊的組織或設計。記住，在更改實際上完成以前，不要先修改**CONTEXT MAP**。

### 示例 運輸應用程序中的兩個**CONTEXT**

我們再次回到運輸系統。應用程序的主要特性之一是在客戶預訂的時候自動為貨物安排路線。模型類似於圖14-2。

**Routing Service**是一個**SERVICE**，它把服務的機制封裝在一個**INTENTION-REVEALING INTERFACE**後面，這個接口是由一些**SIDE-EFFECT-FREE FUNCTION**構成的。這些函數的結果是用**ASSERTION**刻畫的。

(1) 接口聲明瞭當傳入一個**Route Specification**時，將返回一個**Itinerary**。

(2) ASSERTION 規定返回的 **Itinerary** 將滿足所傳入的 **Route Specification**。

從上面這些並不能看出這項困難任務是如何執行的。現在，讓我們來看一下幕後的機制。

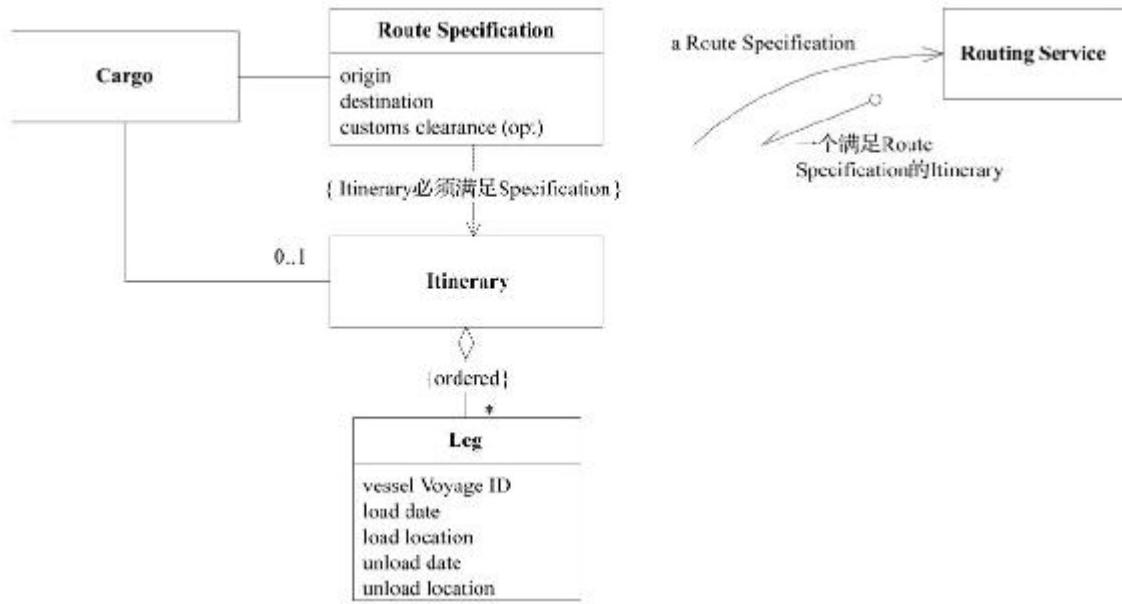


圖14-2

最初在這個示例所在的項目中，我在 **Routing Service** 的內部機制上太過教條了。我希望把領域模型擴展一下，以便把實際的路線安排操作包括進來，由模型來表示航名航次，並直接把這些航名航次與 **Itinerary** 中的 **Leg** ( 航段 ) 關聯起來。但負責處理路線問題的團隊指出，為了更好地執行路線安排，並充分利用那些成熟的算法，應該把這個解決方案實現為一個優化網絡，並把航次的每個航段表示為矩陣中的一個元素。他們堅持要用一個完全不同的運輸作業模型來實現此目的。

就當時的設計而言，他們在路線安排過程的計算要求上無疑是正確的，而且我也沒有更好的思路，因此我只好同意了。實際上，我們

創建了兩個獨立的**BOUNDED CONTEXT**，每個上下文都有各自運輸作業的概念組織（參見圖14-3）。

我們需要接受一個**Routing Service**請求，並將它轉換為**Network Traversal Service**可以理解的術語，然後獲取結果，並將其轉換為**Routing Service**所期望得到的格式。

這意味著並不需要映射這兩個模型中的所有事物，而只要能夠進行這兩個特定的轉換即可：

**Route Specification**→地點代碼的列表

**Node**標識的列表→**Itinerary**

為了進行這兩個轉換，我們必須研究元素在一個模型中的含義，並弄清楚如何在另一個模型中把它表示出來。

我們從第一個轉換開始（**Route Specification**→地點代碼的列表），我們必須考慮列表中的地點序列的含義。列表中的第一項是路線的開始，然後必須依次通過每個地點，直到到達列表中的最後一個地點。因此，起點和目的地分別是列表中的第一項和最後一項，中間（如果有的話）則是清關地點（參見圖4-14）。

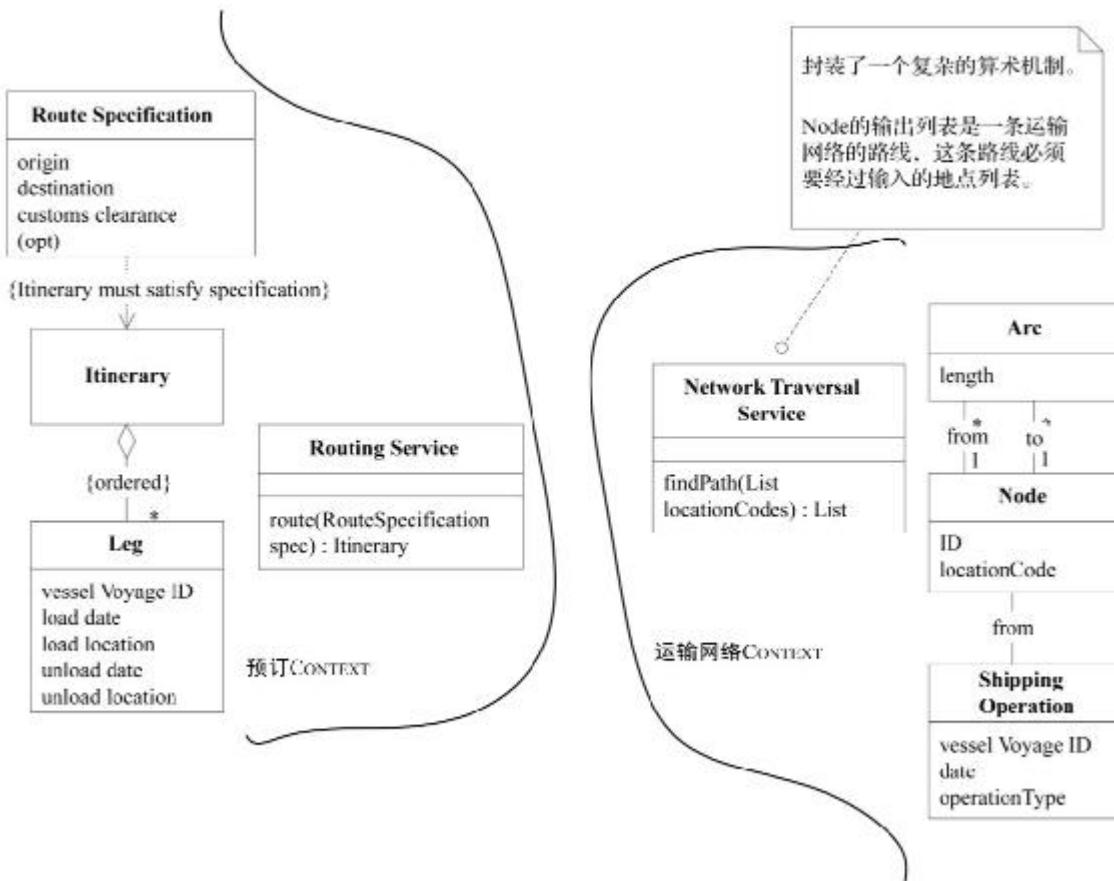


圖14-3 同時使用兩個BOUNDED CONTEXT，這樣就可以應用有效的路線安排算法

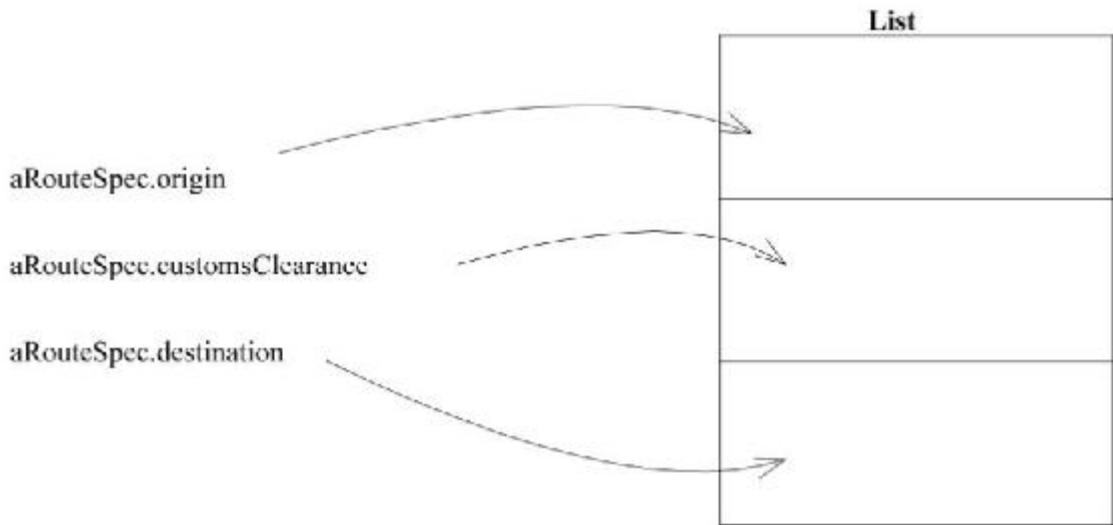


圖14-4 對Network Traversal Service的一次查詢的轉換

( 幸運的是，兩個團隊使用相同的地點代碼，因此我們不必處理地點代碼之間的轉換。 )

注意，反向轉換是不明確的，因為網絡遍歷的輸入允許任意數目的中間點，而不是隻有特別指定的清關點。幸運的是，由於我們並不需要反向轉換，因此不會產生這個問題，但由此我們也瞭解到為什麼有些轉換是不可能的。

現在，我們開始對結果進行轉換（Node標識的列表→Itinerary）。假設我們可以根據所得到的Node ID來使用Repository查詢Node和Shipping Operation對象。那麼，這些Node是如何映射到Leg上的呢？根據operationType-Code，我們可以把Node列表分解為「出發/到達」對。每一對組成一個Leg。



圖14-5 對Network Traversal Service所發現的一個路線進行轉換

每個Node對的屬性按下面這樣進行映射：

```
departureNode.shippingOperation.vesselVoyageId →  
leg.vesselVoyageId  
departureNode.shippingOperation.date → leg.loadDate  
departureNode.locationCode → leg.loadLocationCode  
arrivalNode.shippingOperation.date → leg.unloadDate  
arrivalNode.locationCode → leg.unloadLocationCode
```

這是兩個模型之間的概念轉換映射。現在，我們必須通過某種方法來實現這些轉換。在像這樣的簡單例子中，我通常先創建一個用於轉換的對象，然後找到或創建另一個對像來為子系統的其餘部分提供服務。

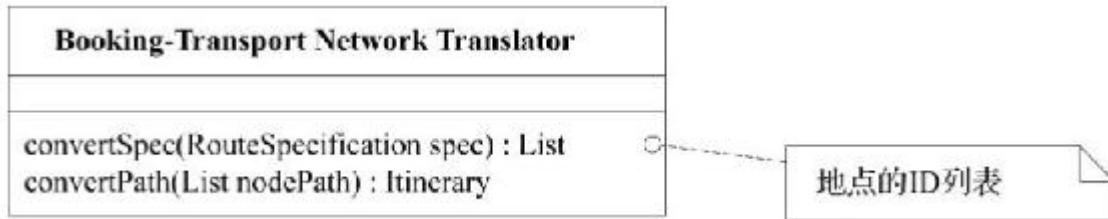


圖14-6 雙向轉換器

這是兩個團隊必須一起維護的對象。設計應該使其易於單元測試，因此最好讓兩個團隊協作開發一個測試套件。除此之外，他們可以採用不同的方式各自開發。

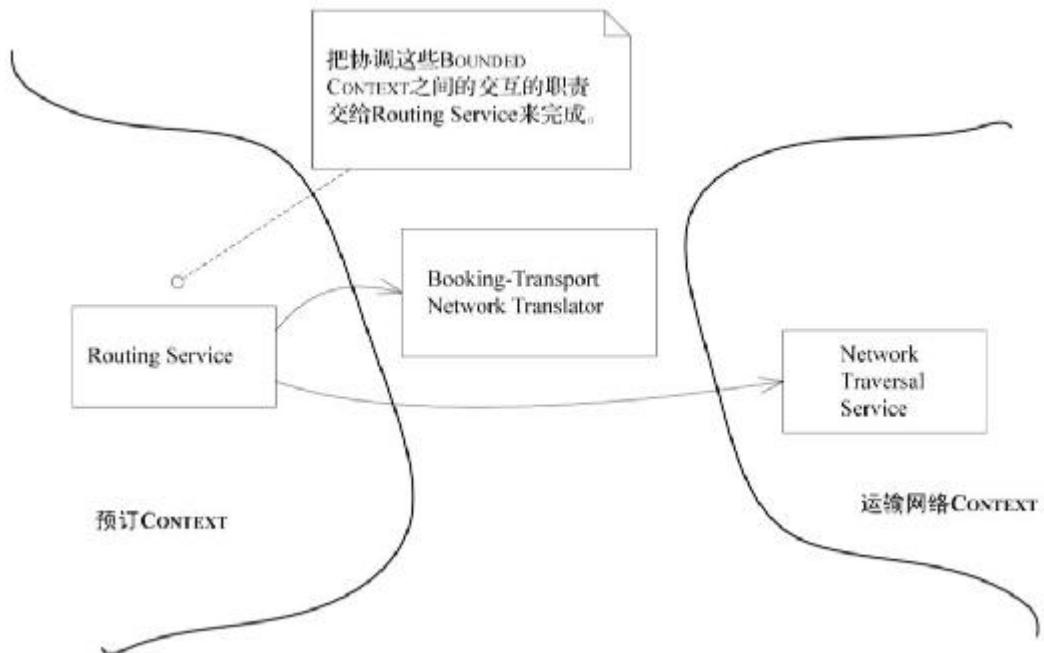


圖14-7

Routing Service的實現現在變成了把任務委託給Translator和Network Traversal Service。其唯一的操作可能如下面代碼所示：

```

public Itinerary route(RouteSpecification spec) {
    Booking_TransportNetwork_Translator translator =
        new Booking_TransportNetwork_Translator();

    List constraintLocations =
        translator.convertConstraints(spec);

    // Get access to the NetworkTraversalService
    List pathNodes =
        TraversalService.findPath(constraintLocations);

    Itinerary result = translator.convert(pathNodes);
    return result;
}

```

這種處理方法還不錯。BOUNDED CONTEXT使每個模型都保持相對整潔，使團隊很大程度上彼此獨立工作，而且，如果最初的假設是正確的，它們可能會發揮很好的作用（本章後面還會回頭討論這個問題）。

兩個上下文之間的接口非常小。Routing Service的接口把預訂上下文中的其餘部分與路線查找事件隔離開。這個接口完全由SIDE-EFFECT-FREE FUNCTION構成，因此很容易測試。與其他CONTEXT和諧共存的一個秘訣是擁有有效的接口測試集。正如裡根總統在裁減核武器談判時所說的名言「信任，但要確認」[\[1\]](#)。

我們很容易設計一組自動測試集來把Route Specification輸入到Routing Service中並檢查返回的Itinerary。

模型上下文總是存在的，但如果我們不注意的話，它們可能會發生重疊和變化。通過明確地定義BOUNDED CONTEXT和CONTEXT MAP，團隊就可以掌控模型的統一過程，並把不同的模型連接起來。

### 14.3.1 測試CONTEXT的邊界

對各個**BOUNDED CONTEXT**的聯繫點的測試特別重要。這些測試有助於解決轉換時所存在的一些細微問題以及彌補邊界溝通上存在的不足。測試充當了有用的早期報警系統，特別是在我們必須信賴那些模型細節卻又無法控制它們時，它能讓我們感到放心。

### **14.3.2 CONTEXT MAP的組織和文檔化**

這裡只有以下兩個重點。

(1) **BOUNDED CONTEXT**應該有名稱，以便可以討論它們。這些名稱應該被添加到團隊的**UBIQUITOUS LANGUAGE**中。

(2) 每個人都應該知道邊界在哪裡，而且應該能夠分辨出任何代碼段的**CONTEXT**，或任何情況的**CONTEXT**。

有很多種方式可以滿足第二項需求，這取決於團隊的文化。一旦定義了**BOUNDED CONTEXT**，那麼把不同上下文的代碼隔離到不同的**MODULE**中就再自然不過了，但這樣就產生了一個問題——如何跟蹤哪個**MODULE**屬於哪個**CONTEXT**。我們可以用命名規範來表明這一點，或者使用其他簡單且不會產生混淆的機制。

同樣重要的是以一種適當的形式來傳達概念邊界，以使團隊中的每個人都能以相同的方式來理解它們。就溝通而言，我喜歡用非正式的圖，就像示例中所顯示的那些圖一樣。也可以使用更嚴格的圖或文本列表來顯示每個**CONTEXT**中的所有包，同時顯示出聯繫點以及負責連接和轉換的機制。有些團隊更願意使用這種方法，而另一些團隊通過口頭協定和大量的討論也能很好地實現這一目的。

無論是哪種情況，將**CONTEXT MAP**融入到討論中都是至關重要的，前提是**CONTEXT**的名稱要添加到**UBIQUITOUS LANGUAGE**中。不要說「George團隊的內容改變了，因此我們也需要改變那些與其進行交互的內容」，而應該說：「Transport Network模型發生了改變，因此我們也需要修改Booking上下文的轉換器。」

## **14.4 BOUNDED CONTEXT之間的關係**

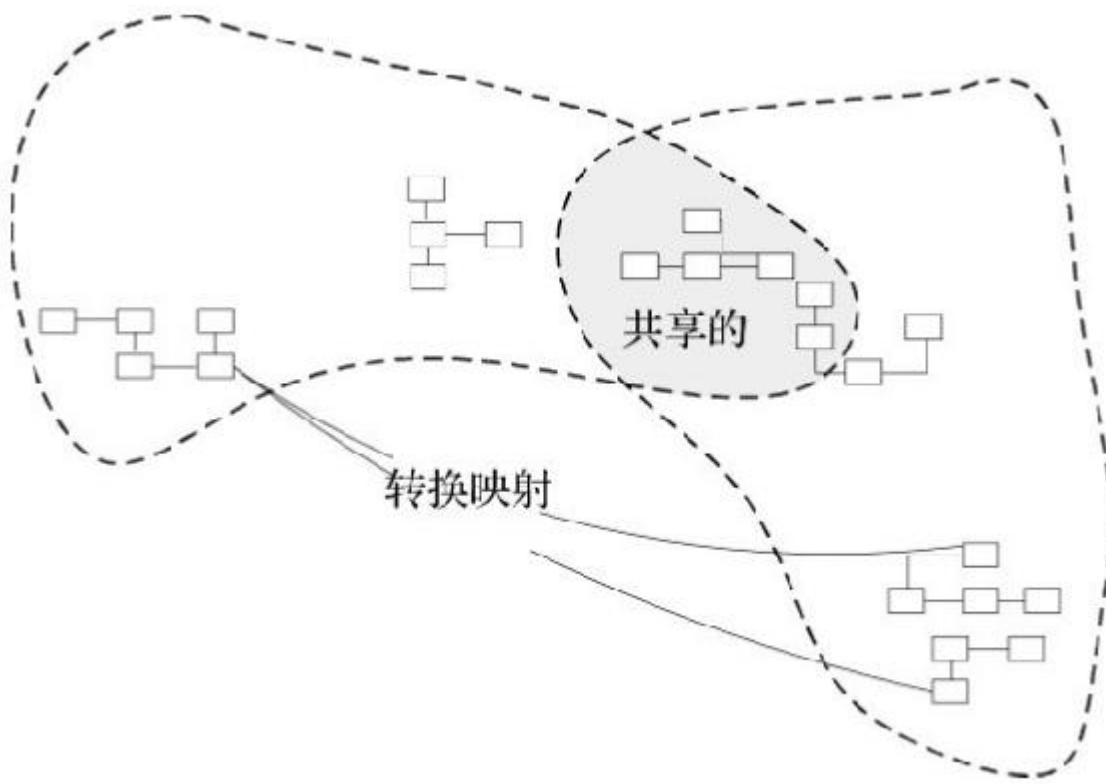
下面介紹的這些模式涵蓋了將兩個模型關聯起來的眾多策略。把模型連接到一起之後，就能夠把整個企業籠括在內。這些模式有著雙重目的，一是為成功地組織開發工作設定目標，二是為描述現有組織提供術語。

現有關係可能與這些模式中的某一種很接近——這可能是由於巧合，也可能是有意設計的——在這種情況下可以使用這個模式的術語來描述關係，但差異之處應該引起重視。然後，隨著每次小的設計修改，關係會與所選定的模式越來越接近。

另一方面，你可能會發現現有關係很混亂或過於複雜。要想得到一個明確的CONTEXT MAP，需要重新組織一些關係。在這種情況或任何需要考慮重組的情況下，這些模式提供了應對各種不同情況的選擇。這些模式的主要區別包括你對另一個模型的控制程度、兩個團隊之間合作水平和合作類型，以及特性和數據的集成程度。

下面這些模式涵蓋了一些最常見和最重要的情況，它們提供了一些很好的思路，沿著這些思路，我們就可以知道如何處理其他情況。開發一個緊密集成產品的優秀團隊可以部署一個大的、統一的模型。如果團隊需要為不同的用戶群提供服務，或者團隊的協調能力有限，可能就需要採用SHARED KERNEL（共享內核）或CUSTOMER/SUPPLIER（客戶/供應商）關係。有時仔細研究需求之後可能發現集成並不重要，而系統最好採用SEPARATE WAY（各行其道）模式。當然，大多數項目都需要與遺留系統或外部系統進行一定程度的集成，這就需要使用OPEN HOST SERVICE（開放主機服務）或ANTICORRUPTION LAYER（防護層）。

## 14.5 模式：SHARED KERNEL



當功能集成受到侷限，CONTINUOUS INTEGRATION的開銷可能會變得非常高。尤其是當團隊的技能水平或行政組織不能保持持續集成，或者只有一個龐大的、笨拙的團隊時，更容易發生這種情況。在這種情況下就要定義單獨的BOUNDED CONTEXT，並組織多個團隊。

當不同團隊開發一些緊密相關的應用程序時，如果團隊之間不進行協調，即使短時間內能夠取得快速進展，但他們開發出的產品可能無法結合到一起。最後可能不得不耗費大量精力在轉換層上，並且頻繁地進行改動，不如一開始就使用CONTINUOUS INTEGRATION那麼省心省力，同時這也造成重複工作，並且無法實現公共的UBIQUITOUS LANGUAGE所帶來的好處。

在很多項目中，我看到一些基本上獨立工作的團隊共享基礎設施層。領域工作採用類似的方法也可以得到很好的效果。保持整個模型

和代碼完全同步的開銷可能太高了，但從系統中仔細挑選出一部分並保持同步，就能以較小的代價獲得較大的收益。

因此：

從領域模型中選出兩個團隊都同意共享的一個子集。當然，除了這個模型子集以外，還包括與該模型部分相關的代碼子集，或數據庫設計的子集。這部分明確共享的內容具有特殊的地位，一個團隊在沒與另一個團隊商量的情況下不應擅自更改它。

功能系統要經常進行集成，但集成的頻率應該比團隊中**CONTINUOUS INTEGRATION**的頻率低一些。在進行這些集成的時候，兩個團隊都要運行測試。

這是一個仔細的平衡。**SHARED KERNEL**（共享內核）不能像其他設計部分那樣自由更改。在做決定時需要與另一個團隊協商。共享內核中必須集成自動測試套件，因為修改共享內核時，必須要通過兩個團隊的所有測試。通常，團隊先修改各自的共享內核副本，然後每隔一段時間與另一個團隊的修改進行集成。例如，在每天（或更短的時間週期）進行**CONTINUOUS INTEGRATION**的團隊中，可以每週進行一次內核的合併。不管代碼集成是怎樣安排的，兩個團隊越早討論修改，效果就會越好。

**SHARED KERNEL**通常是**CORE DOMAIN**，或是一組**GENERIC SUBDOMAIN**（通用子領域），也可能二者兼有（參見第15章），它可以是兩個團隊都需要的任何一部分模型。使用**SHARED KERNEL**的目的是減少重複（並不是消除重複，因為只有在一個**BOUNDED CONTEXT**中才能消除重複），並使兩個子系統之間的集成變得相對容易一些。

## **14.6 模式：CUSTOMER/SUPPLIER DEVELOPMENT TEAM**



我們常常會碰到這樣的情況：一個子系統主要服務於另一個子系統；「下游」組件執行分析或其他功能，這些功能向「上游」組件反饋的信息非常少，所有依賴都是單向的。兩個子系統通常服務於完全不同的用戶群，其執行的任務也不同，在這種情況下使用不同的模型會很有幫助。工具集可能也不相同，因此無法共享程序代碼。

上游和下遊子系統很自然地分隔到兩個**BOUNDED CONTEXT**中。如果兩個組件需要不同的技能或者不同的工具集來實現時，更需要把它們隔離到不同的上下文中。轉換很容易，因為只需要進行單向轉換。但兩個團隊的行政組織關係可能會引起問題。

如果下遊團隊對變更具有否決權，或請求變更的程序太複雜，那麼上游團隊的開發自由度就會受到限制。由於擔心破壞下遊系統，上

游團隊甚至會受到抑制。同時，由於上游團隊掌握優先權，下游團隊有時也會無能為力。

下游團隊依賴於上游團隊，但上游團隊卻不負責下游團隊的產品交付。要琢磨拿什麼來影響對方團隊，是人性呢，還是時間壓力，亦或其他諸如此類的，這需要耗費大量額外的精力。因此，正式規定團隊之間的關係會使所有人工工作起來更容易。這樣，就可以對開發過程進行組織，均衡地處理兩個用戶群的需求，並根據下游所需的特性來安排工作。

在極限編程項目中，已經有了實現此目的的機制——迭代計劃過程。我們只需根據計劃過程來定義兩個團隊之間的關係。下游團隊的代表類似於用戶代表，參加上游團隊的計劃會議，上游團隊直接與他們的「客戶」同仁討論和權衡其所需的任務。結果是供應商團隊得到一個包含下游團隊最需要的任務的迭代計劃，或是通過雙方商定推遲一些任務，這樣下游團隊也就知道這些被推遲的功能不會交付給他們。

如果使用的不是XP過程，那麼無論使用什麼類似的方法來平衡不同用戶的關注點，都可以對這種方法加以擴充，使之把下游應用程序的需求包括進來。

因此：

在兩個團隊之間建立一種明確的客戶/供應商關係。在計劃會議中，下游團隊相當於上游團隊的客戶。根據下游團隊的需求來協商需要執行的任務並為這些任務做預算，以便每個人都知道雙方的約定和進度。

兩個團隊共同開發自動化驗收測試，用來驗證預期的接口。把這些測試添加到上游團隊的測試套件中，以便作為其持續集成的一部分

來運行。這些測試使上游團隊在做出修改時不必擔心對下游團隊產生副作用。

在迭代期間，下游團隊成員應該像傳統的客戶一樣隨時回答上游團隊的提問，並幫助解決問題。

自動化驗收測試是這種客戶關係的一個重要部分。即使在合作得非常好的項目中，雖然客戶很明確他們所依賴的功能並告訴上游團隊，而且供應商也能很認真地把所做的修改傳遞給下游團隊，但如果沒有測試，仍然會發生一些很意外的事情。這些事情將破壞下游團隊的工作，並使上游團隊不得不採取計劃外的緊急修復措施。因此，客戶團隊在與供應商團隊合作的過程中，應該開發自動驗收測試來驗證所期望的接口。上游團隊將把這些測試作為標準測試套件的一部分來運行。任何一個團隊在修改這些測試時都需要與另一個團隊溝通，因為修改測試就意味著修改接口。

當某個客戶對供應商的業務至關重要時，不同公司的項目之間也會出現客戶/供應商關係。下游團隊也能制約上游團隊，一個有影響力的客戶所提出的要求對上游項目的成功非常重要，但這些要求也能破壞上游項目的開發。建立正式的需求響應過程對雙方都有利，因為與內部IT關係相比，在這種外部關係中更難做出「成本/效益」的權衡。

這種模式有兩個關鍵要素。

(1) 關係必須是客戶與供應商的關係，其中客戶的需求是至關重要的。由於下游團隊並不是唯一的客戶，因此不同客戶的要求必須通過協商來平衡，但這些要求都是非常重要的。這種關係與那種經常出現的「窮親威」關係相反，在後者的關係中，下游團隊不得不乞求上游團隊滿足其需求。

(2) 必須有自動測試套件，使上游團隊在修改代碼時不必擔心破壞下游團隊的工作，並使下游團隊能夠專注於自己的工作，而不用總

是密切關註上遊團隊的行動。

在接力賽中，前面的選手在接棒的時候不能一直回頭看，這位選手必須相信隊友能夠把接力棒準確地交到他手中，否則整個團隊的速度無疑會慢下來。

### 示例 收益分析與預訂

我們再次回到運輸示例中。公司組建了一支專門的團隊，負責分析公司收到的所有預訂，看看如何實現收益的最大化。團隊成員可能發現貨輪上還有空位駱，並建議接受更多超訂。他們可能發現貨輪過早地裝滿了散裝貨物，從而使公司不得不拒絕利潤更大的特殊貨物。在這種情況下，他們可能會建議為這類貨物預留空間，或是提高散貨的運輸價格。

為了進行這種分析，他們使用自己的複雜模型。在實現過程中，他們使用了一個帶有構建分析模型工具的數據倉庫。而且他們需要從預訂應用程序中獲取大量信息。

從一開始就知道，這顯然是兩個**BOUNDED CONTEXT**，因為它們使用不同的實現工具，而且最重要的是，它們使用不同的領域模型。那麼它們之間應該具有什麼樣的關係呢？

在這種情況下使用**SHARED KERNEL**看起來很合乎邏輯，因為收益分析只對預訂模型的一個子集感興趣，而且它們自己的模型也有一些諸如貨物、價格等的重疊概念。但是，當使用了不同的實現技術時，**SHARED KERNEL**是很難做到的。此外，收益分析團隊需要建立非常專門的模型，他們要不斷修改模型，並且嘗試其他的模型。他們最好從預訂**CONTEXT**中找到所需的東西，並把它們轉換到自己的上下文中。

（另一方面，如果他們使用**SHARED KERNEL**，他們的翻譯負擔將會輕得多。他們仍然必須重新實現模型，並把數據轉換到新的實現中，但如果模型相同的話，轉換就簡單多了。）

預訂應用程序並不依賴收益分析，因為並沒有打算做自動調整策略。調整決策將由專家來制定，並傳遞給相關的人員和系統。這樣我們就有了一個上游/下游關係。下游的需求如下：

- (1) 一些數據（任何預訂操作都不需要這些數據）；
- (2) 數據庫模式具有一定穩定性（或至少具有可靠的變更通知機制），或者一個用於導出的實用程序。

幸運的是，預訂應用程序開發團隊的項目經理非常積極主動地幫助收益分析團隊。原本以為兩個團隊的合作會是個問題，因為實際負責處理日常預訂業務的運營部門和實際執行收益分析的團隊並非向同一個副總裁報告工作。但高管層非常關心收益管理，而且過去曾看到過兩個部門之間的合作問題，因此調整了一下軟件開發項目的結構，讓兩個團隊的項目經理向同一個人匯報工作。

這樣，應用**CUSTOMER/SUPPLIER DEVELOPMENT TEAM**（客戶/供應商開發團隊）的所有要求都滿足了。

我曾經看到過這種場景出現在很多地方，其中分析軟件開發人員和操作軟件開發人員具有客戶/供應商關係。當上游團隊成員認為他們的角色是服務於客戶時，工作會進展得相當順利。這種關係幾乎總是非正式地組織起來的，因此工作順利與否有賴於兩個項目經理的私人關係。

在一個XP項目中，我曾經看到過正式的客戶/供應商關係，在每次迭代中，下游團隊的代表以客戶的身份參與到計劃過程中，他們與更為傳統的（應用程序功能的）客戶代表聚到一起，共同協商哪些任務應該被添加到迭代計劃中。這是一家小公司的項目，因此最近一級的共同主管不會處在關係鏈的很遠位罷。項目進展得非常順利。

**CUSTOMER/SUPPLIER TEAM**涉及的團隊如果能在同一個部門中工作，最後會形成共同的目標，這樣成功機會將更大一些，如果兩個

團隊分屬不同的公司，但實際上也具有這些角色，同樣也容易成功。但是，當上游團隊不願意為下游團隊提供服務時，情況就會完全不同.....

## **14.7 模式：CONFORMIST**



當兩個具有上游/下游關係的團隊不歸同一個管理者指揮時，CUSTOMER/SUPPLIER TEAM這樣的合作模式就不會奏效。勉強應用這種模式會給下游團隊帶來麻煩。大公司可能會發生這種情況，其中兩個團隊在管理層次中相隔很遠，或者兩個團隊的共同主管不關心它

們之間的關係。當兩個團隊屬於不同公司時，如果客戶的業務對供應商不是非常重要，那麼也會出現這種情況。或許供應商有很多小客戶，或者供應商正在改變市場方向，而不再重視老客戶。也可能是供應商的運營狀況較差，或者已經倒閉。不管是什麼原因，現實情況是下游團隊只能靠自己了。

當兩個開發團隊具有上/下游關係時，如果上游團隊沒有動力來滿足下游團隊的需求，那麼下游團隊將無能為力。出於利他主義的考慮，上游開發人員可能會做出承諾，但他們可能不會履行承諾。下游團隊出於良好的意願會相信這些承諾，從而根據一些永遠不會實現的特性來制定計劃。下游項目只能被擱置，直到團隊最終學會利用現有條件自力更生為止。下游團隊不會得到根據他們的需求而量身定做的接口。

在這種情況下，有3種可能的解決途徑。一種是完全放棄對上游的使用。做出這種選擇時，應進行切實地評估，絕不要假定上游會滿足下游的需求。有時我們會高估這種依賴性的價值，或是低估它的成本。如果下游團隊決定切斷這條鏈，他們將走上SEPARATE WAY（各行其道）的道路（參見本章後面介紹的模式）。

有時，使用上游軟件具有非常大的價值，因此必須保持這種依賴性（或者是行政決策規定團隊不能改變這種依賴性）。在這種情況下，還有兩種途徑可供選擇，選擇哪一種取決於上游設計的質量和風格。如果上游的設計很難使用（可能是由於缺乏封裝、使用了不恰當的抽象或者建模時使用了下游團隊無法使用的範式），那麼下游團隊仍然需要開發自己的模型。他們將擔負起開發轉換層的全部責任，這個層可能會非常複雜（參見本章後面要介紹的ANTICORRUPTION LAYER）。

## 跟隨並不總是壞事

當使用一個具有很大接口的現成組件時，一般應該遵循（CONFORM）該組件中隱含的模型。組件和你自己的應用程序顯然是不同的BOUNDED CONTEXT，因此根據團隊組織和控制的不同，可能需要使用適配器來進行一點點格式轉換，但模型一定要保持相同。否則，就應該質疑使用該組件的價值。如果它確實能夠提供價值，那說明它的設計中已經消化吸收了一些知識。在該組件的應用範圍內，它可能比你的理解要深入。你的模型大概會超出該組件的範圍，而且這些超出部分將演化出你自己的概念。但在兩者連接的地方，你的模型將是一個CONFORMIT，遵從組件模型的領導。實際上，你將被帶到一個更好的設計中。

當你與組件的接口很小時，那麼共享一個統一模型就不那麼重要了，而且轉換也是個可行的選項。但是，當接口很大而且集成更加重要時，跟隨通常是有意義的。

另一方面，如果上游設計的質量不是很差，而且風格也能兼容的話，那麼最好不要再開發一個獨立的模型。這種情況下可以使用CONFORMIST（跟隨者）模式。

因此：

通過嚴格遵從上游團隊的模型，可以消除在BOUNDED CONTEXT之間進行轉換的複雜性。儘管這會限制下游設計人員的風格，而且可能不會得到理想的應用程序模型，但選擇CONFORMITY模式可以極大地簡化集成。此外，這樣還可以與供應商團隊共享UBIQUITOUS LANGUAGE。供應商處於統治地位，因此最好使溝通變容易。他們從利他主義的角度出發，會與你分享信息。

這個決策會加深你對上游團隊的依賴，同時你的應用也受限於上游模型的功能，充其量也只能做一些簡單的增強而已。人們在主觀上不願意這樣做，因此有時本應該這樣選擇時，卻沒有這樣選擇。

如果這些折中不可接受，而上游的依賴又必不可少，那麼還可以選擇第二種方法。通過創建一個ANTICORRUPTION LAYER來盡可能把自己隔離開，這是一種實現轉換映射的積極方法，後面將會討論它。

CONFORMIST模式類似於SHARED KERNEL模式。在這兩種模式中，都有一個重疊的區域——在這個重疊區域內模型是相同的，此外還有你的模型所擴展的部分，以及另一個模型對你沒有影響的部分。這兩種模式之間的區別在於決策制定和開發過程不同。SHARED KERNEL是兩個高度協調的團隊之間的合作模式，而CONFORMIST模式則是應對與一個對合作不感興趣的團隊進行集成。

前面介紹了在兩個BOUNDED CONTEXT之間集成時可以進行的各種合作，從高度合作的SHARED KERNEL模式或CUSTOMER/SUPPLIER DEVELOPER TEAM到單方面的CONFORMIST模式。現在，我們最後來看一種更悲觀的關係，假設另一個團隊既不合作，而且其設計也無法使用時，該如何應對。

## **14.8 模式：ANTICORRUPTION LAYER**



新系統幾乎總是需要與遺留系統或其他系統進行集成，這些系統具有其自己的模型。當把參與集成的**BOUNDED CONTEXT**設計完善並且團隊相互合作時，轉換層可能很簡單，甚至很優雅。但是，當邊界那側發生滲透時，轉換層就要承擔起更多的防護職責。

當正在構建的新系統與另一個系統的接口很大時，為了克服連接兩個模型而帶來的困難，新模型所表達的意圖可能會被完全改變，最終導致它被修改得像是另一個系統的模型了（以一種特定的風格）。遺留系統的模型通常很弱。即使對於那些模型開發得很好的例外情況，它們可能也不符合當前項目的需要。然而，集成遺留系統仍然具有很大的價值，而且有時還是絕對必要的。

正確答案是不要全盤封殺與其他系統的集成。在我經歷過的一些項目中，人們非常熱衷於替換所有遺留系統，但由於工作量太大，這不可能立即完成。此外，與現有系統集成是一種有價值的重用形式。在大型項目中，一個子系統通常必須與其他獨立開發的子系統連接。

這些子系統將從不同角度反映問題領域。當基於不同模型的系統被組合到一起時，為了使新系統符合另一個系統的語義，新系統自己的模型可能會被破壞。即使另一個系統被設計得很好，它也不會與客戶基於同一個模型。而且其他系統往往並不是設計得很好。

當通過接口與外部系統連接時，存在很多障礙。例如，基礎設施層必須提供與另一個系統進行通信的方法，那個系統可能處於不同的平臺上，或是使用了不同的協議。你必須把那個系統的數據類型轉換為你自己系統的數據類型。但通常被忽視的一個事實是那個系統肯定不會使用相同的概念領域模型。

如果從一個系統中取出一些數據，然後在另一個系統中錯誤地解釋了它，那麼顯然會發生錯誤，甚至會破壞數據庫。儘管我們已經認識到這一點，這個問題仍然會「偷襲」我們，因為我們認為在系統之間轉移的是原始數據，其含義是明確的，並且認為這些數據在兩個系統中的含義肯定是相同的。這種假設常常是錯誤的。數據與每個系統的關聯方式會使數據的含義出現細微但重要的差別。而且，即使原始數據元素確實具有完全相同的含義，但在原始數據這樣低的層次上進行接口操作通常是錯誤的。這樣的底層接口使另一個系統的模型喪失瞭解釋數據以及約束其值和關係的能力，同時使新系統背負瞭解釋原始數據的負擔（而且並未使用這些數據自己的模型）。

我們需要在不同模型的關聯部分之間建立轉換機制，這樣模型就不會被未經消化的外來模型元素所破壞。

因此：

**創建一個隔離層，以便根據客戶自己的領域模型來為客戶提供相關功能。這個層通過另一個系統現有接口與其進行對話，而只需對那個系統作出很少的修改，甚至無需修改。在內部，這個層在兩個模型之間進行必要的雙向轉換。**

這種連接兩個系統的機制可能會使我們想到把數據從一個程序傳輸到另一個程序，或者從一個服務器傳輸到另一個服務器。我們很快就會討論技術通信機制的使用。但這些細節問題不應與ANTICORRUPTION LAYER混淆，因為ANTICORRUPTION LAYER並不是向另一個系統發送消息的機制。相反，它是在不同的模型和協議之間轉換概念對像和操作的機制。

ANTICORRUPTION LAYER本身就可能是一個複雜的軟件。接下來將概要描述在創建ANTICORRUPTION LAYER時需要考慮的一些事項。

#### **14.8.1 設計ANTICORRUPTION LAYER的接口**

ANTICORRUPTION LAYER的公共接口通常以一組SERVICE的形式出現，但偶爾也會採用ENTITY的形式。構建一個全新的層來負責兩個系統之間的語義轉換為我們提供了一個機會，它使我們能夠重新對另一個系統的行為進行抽象，並按照與我們的模型一致的方式把服務和信息提供給我們的系統。在我們的模型中，把外部系統表示為一個單獨的組件可能是沒有意義的。最好是使用多個SERVICE（或偶爾使用ENTITY），其中每個SERVICE都使用我們的模型來履行一致的職責。

#### **14.8.2 實現ANTICORRUPTION LAYER**

對ANTICORRUPTION LAYER設計進行組織的一種方法是把它實現為FAÇADE、ADAPTER（這兩種模式來自[Gamma et al.1995]）和轉換器的組合，外加兩個系統之間進行對話所需的通信和傳輸機制。

我們常常需要與那些具有大而複雜、混亂的接口的系統進行集成。這不是概念模型差別的問題（概念模型差別是我們使用ANTICORRUPTION LAYER的動機），而是一個實現問題。當我們嘗試創建ANTICORRUPTION LAYER時，會遇到這個實現問題。當從一個模型轉換到另一個模型的時候（特別是當一個模型很混亂時），如果不

能同時處理那些難於溝通的子系統接口，那麼將很難完成。好在 **FACADE** 可以解決這個問題。

**FACADE** 是子系統的一個可供替換的接口，它簡化了客戶訪問，並使子系統更易於使用。由於我們非常清楚要使用另一個系統的哪些功能，因此可以創建 **FACADE** 來促進和簡化對這些特性的訪問，並把其他特性隱藏起來。**FACADE** 並不改變底層系統的模型。它應該嚴格按照另一個系統的模型來編寫。否則會產生嚴重的後果：輕則導致轉換職責蔓延到多個對象中，並加重 **FACADE** 的負擔；重則創建出另一個模型，這個模型既不屬於另一個系統，也不屬於你自己的 **BOUNDED CONTEXT**。**FACADE** 應該屬於另一個系統的 **BOUNDED CONTEXT**，它只是為了滿足你的專門需要而呈現出的一個更友好的外觀。

**ADAPTER** 是一個包裝器，它允許客戶使用另外一種協議，這種協議可以是行為實現者不理解的協議。當客戶向適配器發送一條消息時，**ADAPTER** 把消息轉換為一條在語義上等同的消息，並將其發送給「被適配者」（*adaptee*）。之後 **ADAPTER** 對響應消息進行轉換，並將其發回。我在這裡使用適配器（*adapter*）這個術語略微有點兒不嚴謹，因為 [Gamma et al. 1995] 一書中強調的是使包裝後的對象符合客戶所期望的標準接口，而我們選擇的是被適配的接口，而且被適配者甚至可能不是一個對象。我們強調的是兩個模型之間的轉換，但我認為這與 **ADAPTER** 的意圖是一致的。

我們所定義的每種 **SERVICE** 都需要一個支持其接口的 **ADAPTER**，這個適配器還需要知道怎樣才能向其他系統及其 **FACADE** 發出相應的請求）。

剩下的要素就是轉換器了。**ADAPTER** 的工作是知道如何生成請求。概念對像或數據的實際轉換是一種完全不同的複雜任務，我們可以讓一個單獨的對象來承擔這項任務，這樣可以使負責轉換的對象和

**ADAPTER**都更易於理解。轉換器可以是一個輕量級的對象，它可以在需要的時候被實例化。由於它只屬於它所服務的**ADAPTER**，因此不需要有狀態，也不需要是分佈式的。

這些都是我用來創建**ANTICORRUPTION LAYER**的基本元素。此外，還有其他一些需要考慮的因素。

如圖14-8所示，一般是由正在設計的系統（你的子系統）來發起一個動作。但在有些情況下，其他子系統可能需要向你的子系統提交某種請求，或是把某個事件通知給你的子系統。**ANTICORRUPTION LAYER**可以是雙向的，它可能使用具有對稱轉換的相同轉換器來定義兩個接口上的**SERVICE**（並使用各自的**ADAPTER**）。儘管實現**ANTICORRUPTION LAYER**通常不需要對另一個子系統做任何修改，但為了使它能夠調用**ANTICORRUPTION LAYER**的**SERVICE**，有時還是有必要修改的。

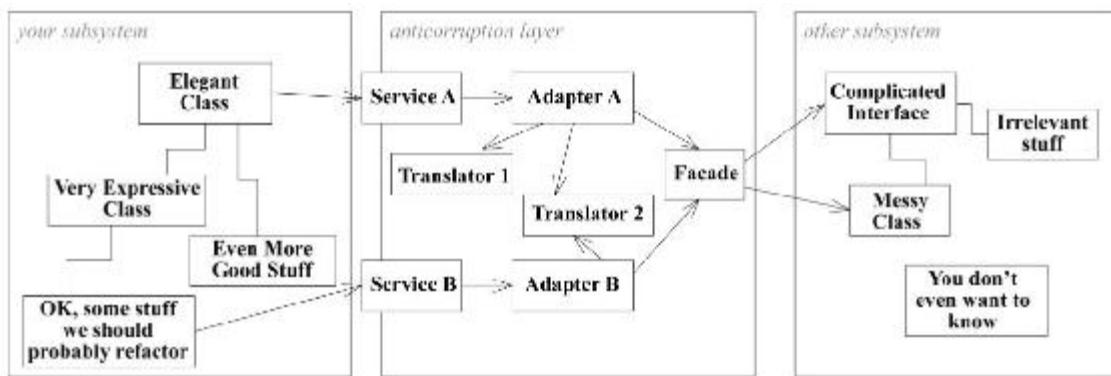


圖14-8 ANTICORRUPTION LAYER的結構

我們通常需要一些通信機制來連接兩個子系統，而且它們可能位於不同的服務器上。在這種情況下，必須決定在哪裡放鎔通信鏈接。如果無法訪問另一個子系統，那麼可能必須在**FACADE**和另一個子系統之間設鎔通信鏈接。但是，如果**FACADE**可以直接與另一個子系統集成到一起，那麼在適配器和**FACADE**之間設鎔通信鏈接也不失為一種好的選擇，這是因為**FACADE**的協議比它所封裝的內容要簡單。在有些情況

下，整個ANTICORRUPTION LAYER可以與另一個子系統放在一起，這時可以在你的系統和構成ANTICORRUPTION LAYER接口的SERVICE之間設置通信鏈接或分發機制。這些都是需要根據實際情況做出的實現和部署決策。它們與ANTICORRUPTION LAYER的概念角色無關。

如果有權訪問另一個子系統，你可能會發現對它進行少許的重構會使你的工作變得更容易。特別是應該為那些需要使用的功能編寫更顯式的接口，如果可能的話，首先從編寫自動測試開始。

當需要進行廣泛的集成時，轉換的成本會直線上升。這時需要對正在設計的系統的模型做出一些選擇，使之盡量接近外部系統，以便使轉換更加容易。做這些工作時要非常小心，不要破壞模型的完整性。這是隻有當轉換的難度無法掌控時才選擇進行的事情。如果這種方法看起來是大部分重要問題的最自然的解決方案，那麼可以考慮讓你的子系統採用CONFORMIST模式，從而消除轉換。

如果另一個子系統很簡單或有一個很整潔的接口，可能就不需要FAÇADE了。

如果一個功能是兩個系統的關係所需的，就可以把這個功能添加到ANTICORRUPTION LAYER中。此外我們還很容易想到兩個特性，一是外部系統使用情況的審計跟蹤，二是追蹤邏輯，其用於調試對另一個接口的調用。

記住，ANTICORRUPTION LAYER是連接兩個BOUNDED CONTEXT的一種方式。我們常常需要使用別人創建的系統，然而我們並未完全理解這些系統，並且也無法控制它們。但這並不是我們需要在兩個子系統之間使用防護層的唯一情況。如果你自己開發的兩個子系統基於不同的模型，那麼使用ANTICORRUPTION LAYER把它們連接起來也是有意義的。在這種情況下，你應該可以完全控制這兩個子系統，而且通常可以使用一個簡單的轉換層。但是，如果這兩個

BOUNDED CONTEXT採用了SEPARATE WAY模式，而仍然需要進行一定的功能集成，那麼可以使用ANTICORRUPTION LAYER來減少它們之間的矛盾。

### 示例 遺留預訂應用程序

為了有一個小的、可以快速開始的最初版本，我們將編寫一個最小化的應用程序，它可以建立一次裝運（shipment）並通過一個轉換層傳遞給遺留系統進行預訂和支持操作。由於我們是專門為了保護正在開發的模型不受遺留設計的影響才構建的轉換層，因此這個轉換就是一個ANTICORRUPTION LAYER。

最初，ANTICORRUPTION LAYER將接收表示裝載的對象，對它們進行轉換並傳遞給遺留系統，請求一個預訂，然後捕獲確認消息並將其轉換成新設計的確認對象。這種隔離使我們基本上能夠獨立於遺留系統來開發新的應用程序，儘管這也必須投入相當多的轉換工作。

在後續的每個版本中，根據後面的決策，新系統要麼可以接管遺留系統的更多功能，要麼可以在不替換現有功能的情況下增加一些新的功能。這種靈活性，以及能夠持續地操作合併的系統並同時進行新老系統的逐步過渡，會使我們在構建ANTICORRUPTION LAYER上投入的工作變得有價值。

### 14.8.3 一個關於防禦的故事

為了保護邊境不受周邊好戰的遊牧部落的侵犯，古代中國修建了長城。雖然它並不是一道不可逾越的屏障，但它卻使得與鄰近地區的通商變得規範有序，同時也可以抵禦侵略和其他不良影響。兩千多年來，它定義了一個邊界，保護中國的農業文明較少受到外界混亂局面的幹擾。

如果沒有長城，中國可能不會形成如此獨特的文明，但儘管如此，長城的修建耗資巨大，它至少使一個朝代「破產」，而且也可能

導致了它最終滅亡。隔離策略的益處必須平衡它產生的代價。我們應該從實際出發，對模型做出適度的修改，使之能夠更好地適應外部模型。

任何集成都是有開銷的，無論這種集成是單一 BOUNDED CONTEXT 中的完全 CONTINUOUS INTEGRATION，還是集成度較輕的 SHARED KERNEL 或 CUSTOMER/SUPPLIER DEVELOPER TEAM，或是單方面的 CONFORMIST 模式和防禦型的 ANTICORRUPTION LAYER 模式。集成可能非常有價值，但它的代價也總是十分高昂的。我們應該確保在真正需要的地方進行集成。

## **14.9 模式：SEPARATE WAY**



我們必須嚴格劃定需求的範圍。如果兩組功能之間的關係並非必不可少，那麼二者完全可以彼此獨立。

集成總是代價高昂，而有時獲益卻很小。

除了在團隊之間進行協調所需的常見開銷以外，集成還迫使我們做出一些折中。可以滿足某一特定需求的簡單專用模型要為能夠處理所有情況的更加抽象的模型讓路。或許有些完全不同的技術能夠輕而易舉地提供某些特性，但它卻難以集成。或許某個團隊很難合作，使得其他團隊在嘗試與之合作時找不到行之有效的方法。

在很多情況下，集成不會提供明顯的收益。如果兩個功能部分並不需要互相調用對方的功能，或者這兩個部分所使用的對象並不需要進行交互，或者在它們操作期間不共享數據，那麼集成可能就是沒有必要的（儘管可以通過一個轉換層進行集成）。僅僅因為特性在用例中相關，並不一定意味著它們必須集成到一起。

因此：

**聲明一個與其他上下文毫無關聯的BOUNDED CONTEXT，使開發人員能夠在這個小範圍內找到簡單、專用的解決方案。**

特性仍然可以被組織到中間件或UI層中，但它們將沒有共享的邏輯，而且應該把通過轉換層進行的數據傳輸減至最小，最好是沒有數據傳輸。

### 示例 一個保險項目的簡化

一個項目團隊著手開發一個新的保險理賠軟件，他們打算把客戶服務代理或理賠人所需的一切功能都集成到一個系統中。經過一年的工作後，團隊成員陷入僵局。分析癱瘓[2]再加上巨大的基礎設施前期投資使他們在漸漸失去耐心的管理層面前沒有任何可以展示的成果。更嚴重的是，他們嘗試完成的工作規模將他們徹底壓垮了。

新任項目經理把所有人員集中到一個房間中，讓他們一週內制定一個新的計劃。他們首先整理出需求列表，然後嘗試估計它們的難度和重要性。他們堅決地刪減掉那些困難並且不重要的需求。然後，開

始為剩下的需求列表排列順序。這個星期他們在這個房間裡制定了很多明智的決策，但最後只有一個被證明是真正重要的。某個時候他們終於認識到有些特性幾乎沒有從集成得到任何好處。例如，理賠人需要訪問一些現有數據庫，而且他們目前的訪問非常不方便。但是，儘管用戶需要得到這些數據，但軟件系統的其他特性卻沒有一個用到它們。

團隊成員提出了各種簡單的訪問方式。一個提議是，可以把關鍵報告導出為HTML並放到內部網（intranet）上。另一個提議是，可以為理賠人提供一種專用查詢，這種查詢是用一個標準軟件包編寫的。通過在內部網的頁面上放鉻鏈接，或者在用戶桌面上放鉻按鈕，就可以把所有這些功能集成進來。

團隊啟動了一組小項目，這些項目除了從同一個菜單啟動之外，不再嘗試任何集成。幾個很有價值的功能幾乎在一夜之間就完成了。卸去了這些過多特性的包袱之後，只剩下了一組精煉的需求，這使得主應用程序的交付又有了希望。

團隊本來可以這樣進行下去，但遺憾的是，他們又回到了老路，再次陷入困境。最後，只有那些採用SEPARATE WAY模式開發的小應用程序被證明是有用的。

採用SEPARATE WAY（各行其道）模式需要預先決定一些選項。儘管持續重構最後可以撤銷任何決策，但完全隔離開發的模型是很難合併的。如果最終仍然需要集成，那麼轉換層將是必要的，而且可能很複雜。當然，不管怎樣，這都是我們將要面對的問題。

現在，讓我們回到更為合作的關係上，來看一下幾種提高集成度的模式。

## **14.10 模式：OPEN HOST SERVICE**

一般來說，在**BOUNDED CONTEXT**中工作時，我們會為**CONTEXT**外部的每個需要集成的組件定義一個轉換層。當集成是一次性的，這種為每個外部系統插入轉換層的方法可以以最小的代價避免破壞模型。但當子系統要與很多系統集成時，可能就需要更靈活的方法了。

當一個子系統必須與大量其他系統進行集成時，為每個集成都定製一個轉換層可能會減慢團隊的工作速度。需要維護的東西會越來越多，而且進行修改的時候擔心的事情也會越來越多。

團隊可能正在反覆做著同樣的事情。如果一個子系統有某種內聚性，那麼或許可以把它描述為一組**SERVICE**，這組**SERVICE**滿足了其他子系統的公共需求。

要想設計出一個足夠乾淨的協議，使之能夠被多個團隊理解和使用，是一件十分困難的事情，因此只有當子系統的資源可以被描述為一組內聚的**SERVICE**並且必須進行很多集成的時候，才值得這樣做。在這些情況下，它能夠把維護模式和持續開發區別開。

因此：

定義一個協議，把你的子系統作為一組**SERVICE**供其他系統訪問。開放這個協議，以便所有需要與你的子系統集成的人都可以使用它。當有新的集成需求時，就增強並擴展這個協議，但個別團隊的特殊需求除外。滿足這種特殊需求的方法是使用一次性的轉換器來擴充協議，以便使共享協議簡單且內聚。

這種通信形式暗含一些共享的模型詞彙，它們是**SERVICE**接口的基礎。這樣，其他子系統就變成了與**OPEN HOST**（開放主機）的模型相連接，而其他團隊則必須學習**HOST**團隊所使用的專用術語。在一些情況下，使用一個眾所周知的**PUBLISHED LANGUAGE**（公開發佈的語言）作為交換模型可以減少耦合併簡化理解。

## **14.11 模式：PUBLISHED LANGUAGE**

兩個BOUNDED CONTEXT之間的模型轉換需要一種公共的語言。

當兩個領域模型必須共存而且必須交換信息時，轉換過程本身就可能很複雜，而且很難文檔化和理解。如果正在構建一個新系統，我們一般會認為新模型是最好的，因此只考慮把其他模型轉換成新模型就可以了。但有時我們的工作是增強一系列舊系統並嘗試集成它們。這時要在眾多模型中選擇一個比較不爛的模型，也就是說「兩害取其輕」。

另一種情況是，當不同業務之間需要互相交換信息時，應該如何做？想讓一個業務採用另一個業務的領域模型不僅是不現實的，而且可能也不符合雙方的需要。領域模型是為瞭解決其用戶的需求而開發的，這樣的模型所包含的一些特性可能使得與另一個系統的通信變得複雜，而實際上沒有必要這麼複雜。此外，如果把一個應用程序的模型用作通信媒介，那麼它可能就無法為滿足新需求而自由地修改了，它必須非常穩定，以便支持當前的通信職責。

與現有領域模型進行直接的轉換可能不是一種好的解決方案。這些模型可能過於複雜或設計得較差。它們可能沒有被很好地文檔化。如果把其中一個模型作為數據交換語言，它實質上就被固定住了，而無法滿足新的開發需求。

OPEN HOST SERVICE使用一個標準化的協議來支持多方集成。它使用一個領域模型來在各系統間進行交換，儘管這些系統的內部可能並不使用該模型。這裡我們可以更進一步——發佈這種語言，或找到一種已經公開發佈的語言。我這裡所說的發佈僅僅是指該語言已經可以供那些對它感興趣的群體使用，而且已經被充分文檔化，兼容一些獨立的解釋。

最近，電子商務界出現了一種激動人心的新技術：**XML**（可擴展標記語言）。這種技術有望使數據交換變得更加容易。**XML**的一個非常有價值的特性是通過**DTD**（文檔類型定義）或**XML**模式來正式定義一個專用的領域語言，從而使得數據可以被轉換為這種語言。一些行業組織已經成立，準備為各自的行業定義一種標準的**DTD**，這樣，業內多方就可以交換信息了，如交換化學公式信息或遺傳代碼信息。實際上這些組織正在以語言定義的形式創建一種共享的領域模型。

因此：

**把一個良好文檔化的、能夠表達出所需領域信息的共享語言作為公共的通信媒介，必要時在其他信息與該語言之間進行轉換。**

這種語言不必從頭創建。很多年以前，我曾經受聘於一家公司，這家公司有一個用**Smalltalk**編寫的軟件產品，它使用**DB2**存儲數據。公司希望靈活地把軟件分發給那些沒有**DB2**許可的用戶，於是請我為**Btrieve**創建一個接口。**Btrieve**是一個輕量級的數據庫引擎，它有一個免費的運行時分發許可。**Btrieve**並不完全是關係型的，但我的客戶只用到**DB2**的很小的一部分功能，而且兩個數據庫都能提供這種能力。公司的開發人員已經在**DB2**之上建立了某種存儲對象的抽像，於是決定把這些工作作為我的**Btrieve**組件的接口。

這種方法確實很有效。軟件順利地與我的客戶系統集成到一起。但是，客戶設計中缺少有關持久化對象的抽像的正式規格說明或文檔，這意味著我必須做很多工作來確定新組件的需求。此外，不太可能重用該組件把其他應用程序從**DB2**遷移到**Btrieve**。而且新軟件穩固了公司的持久化模型，使得持久化對像模型的重構變得更困難。

更好的方法可能是標識出公司所使用的那一小部分**DB2**接口，然後為其提供支持就可以了。**DB2**的接口由**SQL**和大量專有協議構成。儘管接口很複雜，但它已經被嚴格地規定並充分文檔化。由於公司只

使用接口的一個很小的子集，因此複雜性有所降低。如果已開發出一個模擬必要的DB2 接口子集的組件，那麼開發人員所需做的文檔化工作只是標識出該子集即可。與之集成的應用程序已經知道如何與DB2 對話，因此額外要做的工作很少。將來重新設計持久層的工作僅限於DB2子集的使用，就像前面做的改進一樣。

DB2接口是PUBLISHED LANGUAGE的一個例子。在這個例子中，兩個模型都不屬於業務領域，但它們所應用的原則是一致的。由於協作中的一個模型已經是一種PUBLISHED LANGUAGE，因此就不需要引入第三方語言了。

### 示例 一種化學的PUBLISHED Language

在工業界和學術界，有無數的程序用於分類、分析和處理化學公式。幾乎每個程序都使用不同的領域模型來表示化學結構，因此數據的交換總是很難。當然，大部分程序都是用一些無法充分表達領域模型的語言編寫的（如FORTRAN）。當有人想要共享數據時，他們不得不先瞭解其他系統的數據庫的細節，然後再研究出某種轉換方案。

CML（化學標記語言）正是在這種背景下誕生的，它是作為化學領域的公共交流語言被開發出來的專用XML，由一個代表學術界和工業界的組織負責開發和管理[Murray-Rust et al.1995]。

化學信息非常複雜和多樣化，而且會隨著新發現而不斷變化。因此，該組織開發了一種用於描述基礎知識的語言，如有機和無機分子的化學公式、蛋白質序列、光譜或物理量。

既然這種語言已經公開發佈，人們就可以開發相應的工具了（以前，要開發這樣的工具是不值得的，因為它們只能用於一種數據庫）。例如，人們開發一種名為JUMBO Browser的Java應用程序，它的功能是為那些以CML格式存儲的化學結構創建圖形視圖。因此，如果你的數據採用了CML格式，就可以使用這樣的可視化工具。

事實上，CML通過使用XML（一種已發佈的元語言）獲得了雙重優勢。一個優勢是人們對XML很熟悉，因此很容易學習CML，另一個優勢是由於有大量現成的工具（如解析器），因此CML的實現很容易，而且有大量書籍介紹了XML的各個方面，這對CML的文檔化有很大幫助。

下面是一個CML的小例子。雖然像我這樣的外行並不能清楚地理解它是什麼意思，但它的原則還是很清晰的。

```
<CML.ARR ID='array3' EL.TYPE=FLOAT NAME="ATOMIC ORBITAL ELECTRON POPULATIONS"
SIZE=30 GLO.ENT=CML,THE,AOEFOPS>
 1.17947  0.95031  0.97175  1.00000  1.17947  0.95090  0.97174  1.00000
 1.17946  0.96215  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  0.89730  0.89789  0.89789  0.89793  0.89783
</CML.ARR>
```

## 14.12 「大象」的統一

六個好學的古印度人，  
一起去看大象，  
(他們都是盲人)，  
都通過觸摸，  
來滿足瞭解事物的心願。  
第一個接近大象的盲人，  
恰巧了撞上了大象寬闊結實的身軀，  
馬上叫到：「上帝保佑，原來大象就像一堵牆。」  
.....  
第三個盲人，  
碰巧把扭動著的象鼻抓在手中，因此就大膽地說道：  
「依我看，大象就像一條蛇！」  
第四個盲人急切地伸出雙手，

摸到了大象的膝蓋，

「這頭奇異的怪獸最像什麼已經很明顯了」，他說，

「很明顯，大象就像一棵樹」

.....

第六個盲人一開始摸這頭大象，

就抓住了它擺動著的尾巴，

他說，「我認為大象就像一根繩子！」

這六個印度人，

大聲地爭論個不停，

他們每個人的觀點，

都過於僵化和固執，

儘管他們每人都有正確的地方，

但從整體上都是錯誤的！

.....

——摘自John Godfrey Saxe ( 1816—1887 ) 創作的《盲人與象》，來  
源於印度自說經[3]Udana中的故事

即使他們對大象的本質不能達成完全的一致，這些盲人仍然可以根據他們所觸摸到的大象身體的部位來擴展各自的認識。如果並不需要集成，那麼模型統不統一就無關緊要。如果他們需要進行一些集成，那麼實際上並不需要對大象是什麼達成一致，而只要接受各種不同意見就會獲得很多價值。這樣，他們就不會在不知不覺中各執己見。

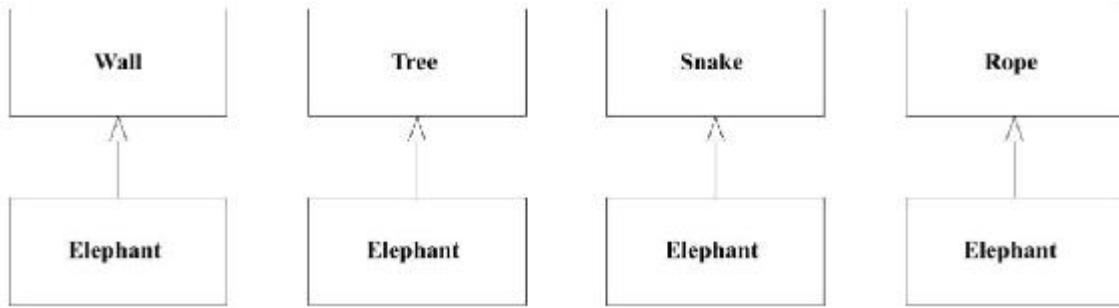
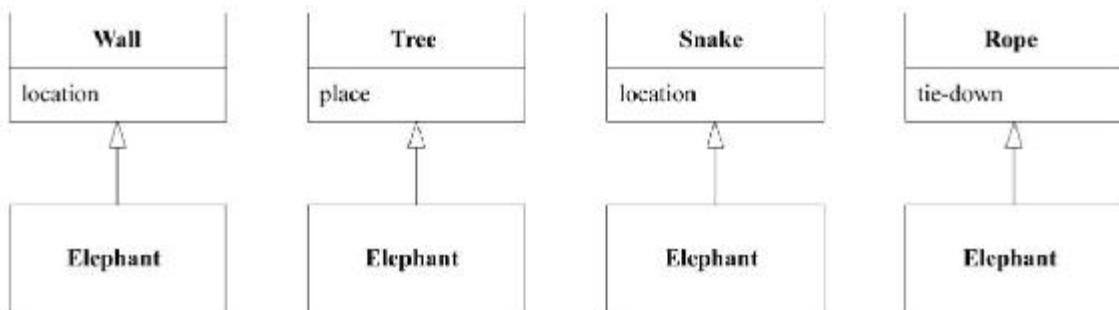


圖14-9 4個沒有集成的上下文

上圖用UML圖表示了6個盲人所認識到的大象模型。這張圖建立了4個獨立的**BOUNDED CONTEXT**，情況很明顯，他們必須找到一種方式來交流他們共同關心的少數幾個方面，或許他們共同關心的就是大象所在的位臘。

當盲人想要分享更多有關大象的信息時，他們會從共享單個**BOUNDED CONTEXT**得到更大的價值。但統一不同的模型卻很難做到。可能沒有人願意放棄自己的模型而採用別人的模型。畢竟，摸到尾巴的那個人知道大象並不像一顆樹，而且那個模型對他來說沒有意義，也沒有用處。統一多個模型幾乎總是意味著創建一個新模型。



Translations: {Wall.location ↔ Tree.place ↔ Snake.location ↔ Rope.tie-down}

圖14-10 4個只有最小集成的上下文

經過一些想像和討論（也許是激烈的討論）之後，盲人們最終可能會認識到他們正在對一個更大整體的不同部分進行描述和建模。從很多方面來講，部分—整體的統一可能不需要花費很多工作。至少集

成的第一步只需弄清楚各個部分是如何相連的就夠了。可以把大象看成一堵牆，下面通過樹幹支撐著，一頭兒是一根繩子，另一頭兒是一條蛇，這樣看就可以適當地滿足一些需求了。

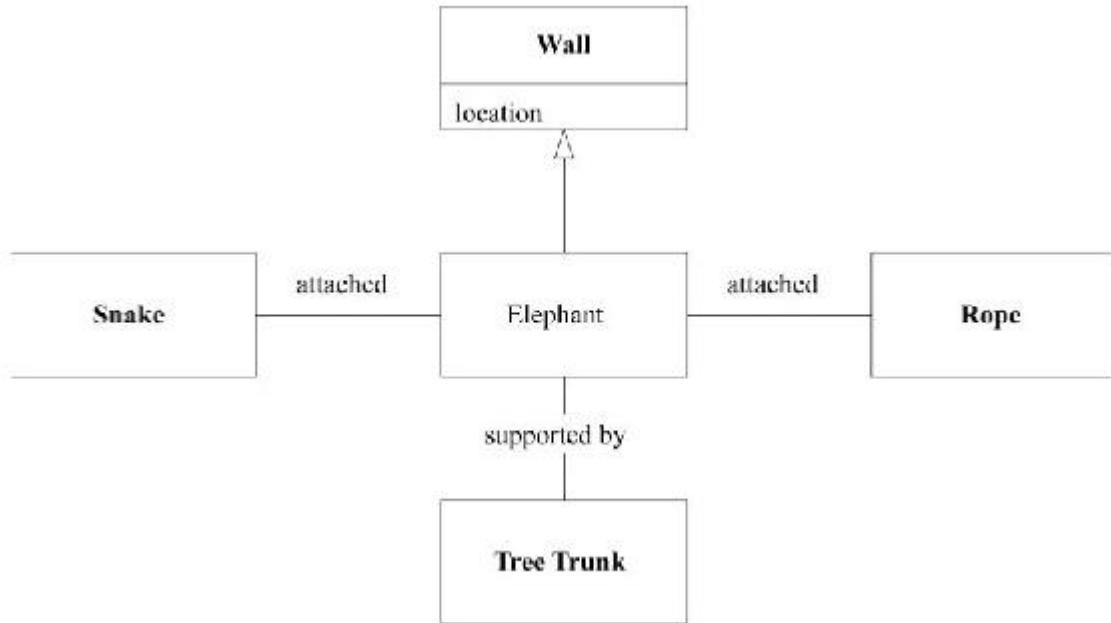


圖14-11 一個粗略集成的上下文

大象模型的統一要比大多數這樣的合併相對簡單一些。遺憾的是，大象模型的統一隻是一個特例——不同模型純粹是在描述整體的不同部分，然而，這通常是模型之間差別的一個方面而已。當兩個模型以不同方式描述同一部分時，問題會變得更加困難。如果兩個盲人都摸到了象鼻子，一個人認為它像蛇，而另一個人認為它像消防水管，那麼他們將更難集成。雙方都無法接受對方的模型，因為那不符合自己的體驗。事實上，他們需要一個新的抽象，這個抽象需要把蛇的「活著的特性」與消防水管的噴水功能合併到一起，而這個抽象還應該排除先前兩個模型中的一些不確切的含義，如人們可能會想到的毒牙，或者可以從身體上拆下並捲起來放到救火車中的這種性質。

儘管我們已經把部分合併成一個整體，但得到的模型還是很簡陋的。它缺乏內聚性，也沒有形成任何潛在領域的輪廓。在持續精化的

過程中，新的理解可能會產生更深層的模型。新的應用程序需求也可能會促成更深層的模型。如果大象開始移動了，那麼「樹」理論就站不住腳了，而盲人建模者們也可能會有所突破，形成「腿」的概念。

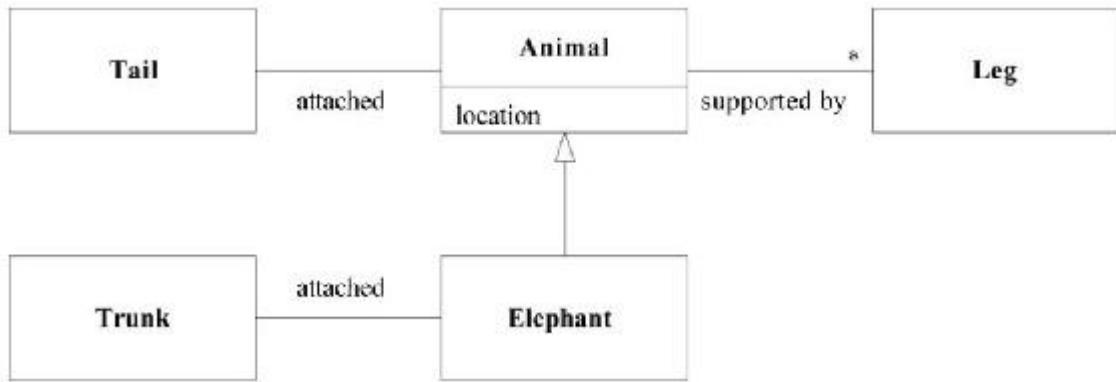


圖14-12 一個更深入集成的上下文

模型集成的第二步是去掉各個模型中那些偶然或不正確的方面，並創建新的概念，在本例中，這個概念就是一種「動物」，它長著「鼻子」、「腿」、「身體」和「尾巴」，每個部分都有其自己的屬性以及與其他部分的明確關係。在很大程度上，成功的模型應該盡可能做到精簡。象鼻與蛇相比，其特性和功能可能比蛇多，也可能比蛇少，但寧「少」勿「多」。寧可缺少噴水功能，也不要包含不正確的毒牙特性。

如果目標只是找到大象，那麼只要對每個模型中所表示的位臘進行轉換就可以了。當需要更多集成時，第一個版本的統一模型不一定達到完全的成熟。把大象看成一堵牆，下面用樹幹支撐著，一頭兒是一根繩子，另一頭兒是一條蛇，就可以適當地滿足一些需求了。緊接著，通過新需求和進一步的理解及溝通的推動，模型可以得到加深和精化。

承認多個互相衝突的領域模型實際上正是面對現實的做法。通過明確定義每個模型都適用的上下文，可以維護每個模型的完整性，並

清楚地看到要在兩個模型之間創建的任何特殊接口的含義。盲人沒辦法看到整個大象，但只要他們承認各自的理解是不完整的，他們的問題就能得到解決。

## 14.13 選擇你的模型上下文策略

在任何時候，繪製出CONTEXT MAP來反映當前狀況都是很重要的。但是，一旦繪製好CONTEXT MAP之後，你很可能想要改變現狀。現在，你可以開始有意識地選擇CONTEXT的邊界和關係。以下是一些指導原則。

### 14.13.1 團隊決策或更高層決策

首先，團隊必須決定在哪裡定義BOUNDED CONTEXT，以及它們之間有什麼樣的關係。這些決策必須由團隊做出，或者至少傳達給整個團隊，並且被團隊裡的每個人理解。事實上，這樣的決策通常需要與外部團隊達成一致。按照本身價值來說，在決定是否擴展或分割BOUNDED CONTEXT時，應該權衡團隊獨立工作的價值以及能產生直接且豐富集成的價值，以這兩種價值的成本—效益作為決策的依據。在實踐中，團隊之間的行政關係往往決定了系統的集成方式。由於匯報結構，有技術優勢的統一可能無法實現。管理層所要求的合併可能並不實用。你不會總能得到你想要的東西，但你至少可以評估出這些決策的代價，並反映給管理層，以便採取相應的措施來減小代價。從一個現實的CONTEXT MAP開始，並根據實際情況來選擇改變。

### 14.13.2 趟身上下文中

開發軟件項目時，我們首先是對自己團隊正在開發的那些部分感興趣（「設計中的系統」），其次是對那些與我們交互的系統感興趣。典型情況下，設計中的系統將被劃分為一到兩個BOUNDED

CONTEXT，開發團隊的主力將在這些上下文中工作，或許還會有另外一到兩個起支持作用的CONTEXT。除此之外，就是這些CONTEXT與外部系統之間的關係。這是一種簡單、典型的情況，能讓你對可能會遇到的情形有一些粗略的瞭解。

實際上，我們正是自己所處理的主要CONTEXT的一部分，這會在我們的CONTEXT MAP中反映出來。只要我們知道自己存在偏好，並且在超出該CONTEXT MAP的應用邊界時能夠意識到已越界，那麼就不會有什麼問題。

### 14.13.3 轉換邊界

在畫出BOUNDED CONTEXT的邊界時，有無數種情況，也有無數種選擇。但權衡時所要考慮的通常是下面所列出的某些因素。

#### 首選較大的BOUNDED CONTEXT

當用一個統一模型來處理更多任務時，用戶任務之間的流動更順暢。

一個內聚模型比兩個不同模型再加它們之間的映射更容易理解。

兩個模型之間的轉換可能會很難（有時甚至是不可能的）。

共享語言可以使團隊溝通起來更清楚。

#### 首選較小的BOUNDED CONTEXT

開發人員之間的溝通開銷減少了。

由於團隊和代碼規模較小，CONTINUOUS INTEGRATION更容易了。

較大的上下文要求更加通用的抽象模型，而掌握所需技巧的人員會出現短缺。

不同的模型可以滿足一些特殊需求，或者是能夠把一些特殊用戶群的專門術語和UBIQUITOUS LANGUAGE的專門術語包括進來。

在不同**BOUNDED CONTEXT**之間進行深度功能集成是不切實際的。在一個模型中，只有那些能夠嚴格按照另一個模型來表述的部分才能夠進行集成，而且，即便是這種級別的集成可能也需要付出相當大的工作量。當兩個系統之間有一個很小的接口時，集成是有意義的。

#### **14.13.4 接受那些我們無法更改的事物：描述外部系統**

最好從一些最簡單的決策開始。一些子系統顯然不在開發中的系統的任何**BOUNDED CONTEXT**中。一些無法立即淘汰的大型遺留系統和那些提供所需服務的外部系統就是這樣的例子。我們很容易就能識別出這些系統，並把它們與你的設計隔離開。

在做出假設時必須要保持謹慎。我們會很輕易地認為這些系統構成了其自己的**BOUNDED CONTEXT**，但大多數外部系統只是勉強滿足定義。首先，定義**BOUNDED CONTEXT**的目的是把模型統一在特定邊界之內。你可能負責遺留系統的維護，在這種情況下，可以明確地聲明這一目的，或者也可以很好地協調遺留團隊來執行非正式的**CONTINUOUS INTEGRATION**，但不要認為遺留團隊的配合是理所當然的事情。仔細檢查，如果開發工作集成得不好，一定要特別小心。在這樣的系統中，不同部分之間出現語義矛盾是很平常的事情。

#### **14.13.5 與外部系統的關係**

這裡可以應用3種模式。首先，可以考慮**SEPARATE WAY**模式。當然，如果你不需要集成，就不用把它們包括進來。但一定要真正確定不需要集成。只為用戶提供對兩個系統的簡單訪問確實夠用嗎？集成要花費很大代價而且還會分散精力，因此要盡可能為你的項目減輕負擔。

如果集成確實非常重要，可以在兩種極端的模式之中進行選擇：**CONFORMIST** 模式或**ANTICORRUPTION LAYER** 模式。作為**CONFORMIST**並不那麼有趣，你的創造力和你對新功能的選擇都會受到限制。當構建一個大型的新系統時，遵循遺留系統或外部系統的模型可能是不現實的（畢竟，為什麼要構建新系統呢？）。但是，當對一個大的系統進行外圍擴展時，而且這個系統仍然是主要系統，在這種情況下，繼續使用遺留模型可能就很合適。這種選擇的例子包括輕量級的決策支持工具，這些工具通常是用Excel或其他簡單工具編寫的。如果你的應用程序確實是現有系統的一個擴展，而且與該系統的接口很大，那麼**CONTEXT**之間轉換所需的工作量可能比應用程序功能本身需要的工作量還大。儘管你已經處於另一個系統的**BOUNDED CONTEXT**中，但你自己的一些好的設計仍然有用武之地。如果另一個系統有著可以識別的領域模型，那麼只要使這個模型比在原來的系統中更清晰，你就可以改進你的實現，唯一需要注意的是要嚴格地遵照那個老模型。如果你決定採用**CONFORMIST**設計，就必須全心全意地去做。你應該約束自己只可以去擴展現有模型，而不能去修改它。

當正在設計的系統功能並不僅僅是擴展現有系統時，而且你與另一個系統的接口很小，或者另一個系統的設計非常糟糕，那麼實際上你會希望使用自己的**BOUNDED CONTEXT**，這意味著需要構建一個轉換層，甚至是一個**ANTICORRUPTION LAYER**。

#### 14.13.6 設計中的系統

你的項目團隊正在構建的軟件就是設計中的系統。你可以在這個區域內聲明**BOUNDED CONTEXT**，並在每個**BOUNDED CONTEXT**中應用**CONTINUOUS INTEGRATION**，以便保持它們的統一。但應該有幾個上下文呢？各個上下文之間又應該是什麼關係呢？與外部系統的情

況相比，這些問題的答案會變得更加不確定，因為我們擁有更多的主動權。

情況可能非常簡單：設計中的整個系統使用一個 BOUNDED CONTEXT。例如，當一個少於 10 人的團隊正在開發高度相關的功能時，這可能就是一種很好的選擇。

隨著團隊規模的增大，CONTINUOUS INTEGRATION 可能會變得困難起來（儘管我也看到過一些較大的團隊仍能保持持續集成）。你可能希望採用 SHARED KERNEL 模式，並把幾組相對獨立的功能劃分到不同的 BOUNDED CONTEXT 中，使得在每個 BOUNDED CONTEXT 中工作的人員少於 10 人。在這些 BOUNDED CONTEXT 中，如果有兩個上下文之間的所有依賴都是單向的，就可以建成 CUSTOMER/SUPPLIER DEVELOPMENT TEAM。

你可能認識到兩個團隊的思想截然不同，以致他們的建模工作總是發生矛盾。可能他們需要從模型得到完全不同的東西，或者只是背景知識有某種不同，又或者是由於項目所採用的管理結構而引起的。如果這種矛盾的原因是你無法改變或不想改變的，那麼可以讓他們的模型採用 SEPARATE WAY 模式。在需要集成的地方，兩個團隊可以共同開發並維護一個轉換層，把它作為唯一的 CONTINUOUS INTEGRATION 點。這與同外部系統的集成正好相反，在外部集成中，一般由 ANTICORRUPTION LAYER 來起調節作用，而且從另一端得不到太多的支持。

一般來說，每個 BOUNDED CONTEXT 對應一個團隊。一個團隊也可以維護多個 BOUNDED CONTEXT，但多個團隊在一個上下文中工作卻是比較難的（雖然並非不可能）。

#### 14.13.7 用不同模型滿足特殊需要

同一業務的不同小組常常有各自的專用術語，而且可能各不相同。這些本地術語可能是非常精確的，並且是根據他們的需要定製的。要想改變它們（例如，施行標準化的企業級術語），需要大量的培訓和分析，以便解決差異問題。即使如此，新術語仍然可能沒有原來那個已經經過精心調整的術語好用。

你可能決定通過不同的**BOUNDED CONTEXT**來滿足這些特殊需要，除了轉換層的**CONTINUOUS INTEGRATION**以外，讓模型採用**SEPARATE WAY**模式。**UBIQUITOUS LANGUAGE**的不同專用術語將圍繞這些模型以及它們所基於的行話來發展。如果兩種專用術語有很多重疊之處，那麼**SHARED KERNEL**模式就可以滿足特殊化要求，同時又能把轉換成本減至最小。

當不需要集成或者集成相對有限時，就可以繼續使用已經習慣的術語，以免破壞模型。但這也有其自己的代價和風險。如下所示。

沒有共同的語言，交流將會減少。

集成開銷更高。

隨著相同業務活動和實體的不同模型的發展，工作會有一定的重複。

但是，最大的風險或許是，它會成為拒絕改變的理由，或為古怪、狹隘的模型辯護。為了滿足特殊的需要，需要對系統的這一部分進行多大的定製？最重要的是，這個用戶群的專門術語有多大的價值？你必須在團隊獨立操作的價值與轉換的風險之間做出權衡，並且留心合理地處理一些沒有價值的術語變化。

有時會出現一個深層次的模型，它把這些不同語言統一起來，並能夠滿足雙方的要求。只有經過大量開發工作和知識消化之後，深層次模型才會在生命週期的後期出現。深層次模型不是計劃出來的，我們只能在它出現的時候抓住機遇，修改自己的策略並進行重構。

記住，在需要大量集成的地方，轉換成本會大大增加。在團隊之間進行一些協調工作（從精確地修改一個具有複雜轉換的對象到採用SHARED KERNEL模式）可以使轉換變得更加容易，同時又不需要完全的統一。

### 14.13.8 部署

在複雜系統中，對打包和部署進行協調是一項繁瑣的任務，這類任務總是要比看上去難得多。BOUNDED CONTEXT策略的選擇將影響部署。例如，當CUSTOMER/SUPPLIER TEAM部署新版本時，他們必須相互協調來發佈經過共同測試的版本。在這些版本中，必須要進行代碼和數據遷移。在分佈式系統中，一種好的做法是把CONTEXT之間的所有轉換層放在同一個進程中，這樣就不會出現多個版本共存的情況。

當數據遷移可能很花時間或者分佈式系統無法同步更新時，即使是單一BOUNDED CONTEXT中的組件部署也是很困難的，這會導致代碼和數據有兩個版本共存。

由於部署環境和技術存在不同，有很多技術因素需要考慮。但BOUNDED CONTEXT關係可以為我們指出重點問題。轉換接口已經被標出。

繪製CONTEXT邊界時應該反映出部署計劃的可行性。當兩個CONTEXT通過一個轉換層連接時，要想更新其中的一個CONTEXT，新的轉換層需要為另一個CONTEXT提供相同的接口。SHARED KERNEL需要進行更多的協調工作，不僅在開發中如此，而且在部署中也同樣應該如此。SEPARATE WAY模式可以使工作簡單很多。

### 14.13.9 權衡

通過總結這些指導原則可知有很多統一或集成模型的策略。一般來說，我們需要在無縫功能集成的益處和額外的協調和溝通工作之間做出權衡。還要在更獨立的操作與更順暢的溝通之間做出權衡。更積極的統一需要對有關子系統的設計有更多控制。

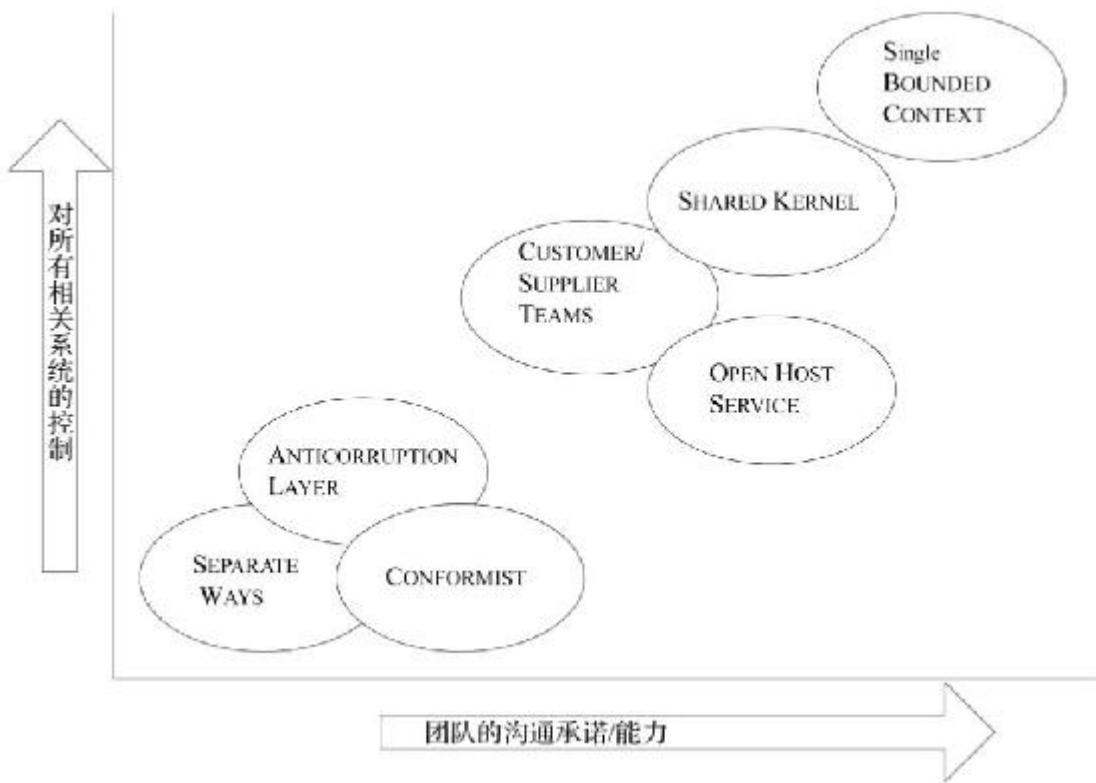


圖14-13 CONTEXT關係模式的相對要求

#### 14.13.10 當項目正在進行時

很多情況下，我們不是從頭開發一個項目，而是會改進一個正在開發的項目。在這種情況下，第一步是根據當前的狀況來定義 BOUNDED CONTEXT。這很關鍵。為了有效地定義上下文，CONTEXT MAP必須反映出團隊的實際工作，而不是反映那個通過遵守以上描述的指導原則而得出的理想組織。

描述了當前真實的 BOUNDED CONTEXT 以及它們的關係以後，下一步就是圍繞當前組織結構來加強團隊的工作。在 CONTEXT 中加強

CONTINUOUS INTEGRATION。把所有分散的轉換代碼重構到ANTICORRUPTION LAYER中。命名現有的BOUNDED CONTEXT，並確保它們處於項目的UBIQUITOUS LANGUAGE中。

現在可以開始考慮修改邊界和它們的關係了。這些修改很自然地由相同的原則來驅動——之前已經描述了在新項目上使用這些原則，但我們應該把這些修改分成較小的部分，以便根據實際情況做出選擇，從而在只花費最少的工作和對模型產生最小破壞的前提下創造最大的價值。

下一節將討論如何修改CONTEXT的邊界。

## 14.14 轉換

像建模和設計的其他方面一樣，有關BOUNDED CONTEXT的決策並非不可改變的。在很多情況下，我們必須改變最初有關邊界以及BOUNDED CONTEXT之間關係的決策，這是不可避免的。一般而言，分割CONTEXT是很容易的，但合併它們或改變它們之間的關係卻很難。下面將介紹幾種有代表性的修改，它們很難，但也很重要。這些轉換往往很大，無法在一次重構中完成，甚至無法在一次項目迭代中完成。因為這個原因，我將把這些轉換劃分為一系列簡單的步驟。當然，這些只是一些指導原則，你必須根據你的特殊情況和事件對它們進行調整。

### 14.14.1 合併CONTEXT : SEPARATE WAY → SHARED KERNEL

合併BOUNDED CONTEXT的動機很多：翻譯開銷過高、重複現象很明顯。合併很難，但什麼時候做都不晚，只是需要一些耐心。

即使你的最終目標是完全合併成一個採用CONTINUOUS INTEGRATION的CONTEXT，也應該先過渡到SHARED KERNEL。

(1) 評估初始狀況。在開始統一兩個CONTEXT之前，一定要確信它們確實需要統一。

(2) 建立合併過程。你需要決定代碼的共享方式以及模塊應該採用哪種命名約定。SHARED KERNEL的代碼至少每週要集成一次，而且它必須有一個測試套件。在開發任何共享代碼之前，先把它設置好。（測試套件將是空的，因此很容易通過！）

(3) 選擇某個小的子領域作為開始，它應該是兩個CONTEXT中重複出現的子領域，但不是CORE DOMAIN的一部分。最初的合併主要是為了建立合併過程，因此最好選擇一些簡單且相對通用或不重要的部分。檢查已存在的集成和轉換。選擇那些經過轉換的部分，其優勢在於一開始就有用於驗證的轉換機制，此外還可以簡化轉換層。

此時，我們有兩個應對相同子領域的模型。基本上有3種合併方法。我們可以選擇一個模型，並重構另一個CONTEXT，使之與第一個模型兼容。我們可以從整體上做出這個決策，把目標設置為系統性地替換一個CONTEXT的模型，並保持被開發模型的內聚性。也可以一次選擇一部分，到最後兩個模型可能會「兩全其美」（但注意最後不要弄得一團糟）。

第三種選擇是找到一個新模型，這個模型可能比最初的兩個都深刻，能夠承擔二者的職責。

(4) 從兩個團隊中共選出2~4位開發人員組成一個小組，由他們來為子領域開發一個共享的模型。不管模型是如何得出的，它的內容必須詳細。這包括一些困難的工作：識別同義詞和映射那些尚未被翻譯的術語。這個聯合團隊需要為模型開發一個基本的測試集。

(5) 來自兩個團隊的開發人員一起負責實現模型（或修改要共享的現有代碼）、確定各種細節並使模型開始工作。如果這些開發人員在模型中遇到了問題，就從第(3)步開始重新組織團隊，並進行必要的概念修訂工作。

(6) 每個團隊的開發人員都承擔與新的**SHARED KERNEL**集成的任務。

(7) 清除那些不再需要的翻譯。

這時你會得到一個非常小的**SHARED KERNEL**，並且有一個過程來維護它。在後續的項目迭代中，重複第(3)~(7)步來共享更多內容。隨著過程的不斷鞏固和團隊信心的樹立，就可以選擇更複雜的子領域了，同時處理多個子領域，或者處理**CORE DOMAIN**中的子領域。

注意：當從模型中選取更多與領域有關的部分時，可能會遇到這樣的情況，即兩個模型各自採用了不同用戶群的專用術語。聰明的做法是先不要把它們合併到**SHARED KERNEL**中，除非工作中出現了突破，得到了一個深層模型，這個模型為你提供了一種能夠替代那兩種專用術語的語言。**SHARED KERNEL**的優點是它具有**CONTINUOUS INTEGRATION**的部分優勢，同時又保留了**SEPARATE WAY**模式的一些優點。

以上這些是把模型的一些部分合併到**SHARED KERNEL**中的指導原則。在繼續討論之前，我們來看一下另外一種方法，它能夠部分解決上述轉換所面對的問題。如果兩個模型中有一個毫無疑問是符合首選條件的，那麼就考慮向它過渡，而不用進行集成。不共享公共的子領域，而只是系統性地通過重構應用程序把這些子領域的所有職責從一個**BOUNDED CONTEXT**轉移到另一個**BOUNDED CONTEXT**，從而使用那個更受青睞的**CONTEXT**的模型，並對該模型進行需要的增強。在沒有集成開銷的情況下，消除了冗餘。很有可能（但也不是必然的）那

個更受青睞的BOUNDED CONTEXT最終會完全取代另一個BOUNDED CONTEXT，這樣就實現了與合併完全一樣的效果。在轉換過程中（這個過程可能相當長或無法確定），這種方法具有SEPARATE WAY模式常見的優點和缺點，而且我們必須拿這些優缺點與SHARED KERNEL的利弊進行權衡。

#### **14.14.2 合併CONTEXT：SHARED KERNEL→CONTINUOUS INTEGRATION**

如果你的SHARED KERNEL正在擴大，你可能會被完全統一兩個BOUNDED CONTEXT的優點所吸引。但這並不只是一個解決模型差異的問題。你將改變團隊的結構，而且最終會改變人們所使用的語言。

這個過程從人員和團隊的準備開始。

(1) 確保每個團隊都已經建立了CONTINUOUS INTEGRATION所需的所有過程（共享代碼所有權、頻繁集成等）。兩個團隊協商集成步驟，以便所有人都以同一步調工作。

(2) 團隊成員在團隊之間流動。這樣可以形成一大批同時理解兩個模型的人員，並且可以把兩個團隊的人員聯繫起來。

(3) 澄清每個模型的精髓（參見第15章）。

(4) 現在，團隊應該有了足夠的信心把核心領域合併到SHARED KERNEL中。這可能需要多次迭代，有時需要在新共享的部分與尚未共享的部分之間使用臨時的轉換層。一旦進入到合併CORE DOMAIN的過程中，最好能快速完成。這是一個開銷高且易出錯的階段，因此應該盡可能縮短時間，要優先於新的開發任務。但注意量力而行，不要超過你的處理能力。

有幾種方式用於合併CORE模型。可以保持一個模型，然後修改另一個，使之與第一個兼容，或者可以為子領域創建一個新模型，並通過修改兩個上下文來使用這個模型。如果兩個模型已經被修改以滿足

不同用戶的需要，你就要注意了。你需要保留兩個初始模型中的這些專業能力。這就要求開發一個能夠替代兩個原始模型的更深層的模型。開發這樣一個更深入的統一模型是很難的，但如果你已經決定完全合併兩個CONTEXT，就沒有選擇多種專門術語的空間了。這樣做的好處是最終模型和代碼的集成變得更清晰了。注意不要影響到你滿足用戶特殊需要的能力。

(5) 隨著SHARED KERNEL的增長，把集成頻率提高到每天一次，最後實現CONTINUOUS INTEGRATION。

(6) 當SHARED KERNEL逐漸把先前兩個BOUNDED CONTEXT的所有內容都包括進來的時候，你會發現要麼形成了一個大的團隊，要麼形成了兩個較小的團隊，這兩個較小的團隊共享一個CONTINUOUS INTEGRATION的代碼庫，而且團隊成員可以經常在兩個團隊之間來迴流動。

#### 14.14.3 逐步淘汰遺留系統

好花美麗不常開，好景怡人不常在，就算遺留計算機軟件也一樣會走向終結。但這可不會自動自發地出現。這些老的系統可能與業務及其他系統緊密交織在一起，因此淘汰它們可能需要很多年。好在我們並不需要一次就把所有東西都淘汰掉。

這一話題的涉及面太廣了，這裡的討論也只能淺嘗輒止。我們將討論一種常見的情況：用一系列更現代的系統來補充業務中每天都在使用的老系統，新系統通過一個ANTICORRUPTION LAYER與老系統進行通信。

首先要執行的步驟是確定測試策略。應該為新系統中的新功能編寫自動的單元測試，但逐步淘汰遺留系統還有一些特殊的測試需求。一些組織在某段時間內會同時運行新舊兩個系統。

在任何一次迭代中：

- (1) 確定遺留系統的哪個功能可以在一個迭代中被添加到某個新系統中；
- (2) 確定需要在ANTICORRUPTION LAYER中添加的功能；
- (3) 實現；
- (4) 部署；

有時，需要進行多次迭代才能編寫一個與遺留系統的某個功能等價的功能單元，這時在計劃新的替代功能時仍以小規模的迭代為單元，最後一次性部署多次迭代。

部署涉及的變數太多，以至於我不可能涵蓋所有的基本情況。就開發而言，如果這些小規模、增量的改動能夠推到生產環境，那真是再好不過了。但通常情況，還是需要將他們組織成更大的發佈。在新軟件的使用方面，用戶培訓是必不可少的。有時在成功部署的同時還必須進行開發工作。還有很多後勤問題需要解決。

一旦最終進入運行階段後，應該遵循如下步驟。

(5) 找出ANTICORRUPTION LAYER中那些不必要的部分，並去掉它們；

(6) 考慮刪除遺留系統中目前未被使用的模塊，雖然這種做法未必實際。有趣的是，遺留系統設計得越好，它就越容易被淘汰。而設計得不好的軟件卻很難一點兒一點兒地去除。這時，我們可以暫時忽略那些未使用的部分，直到將來剩餘部分已經被淘汰，這時整個遺留系統就可以停止使用了。

不斷重複這幾個步驟。遺留系統應該越來越少地參與業務，最終，替換工作會看到希望的曙光並完全停止遺留系統。同時，隨著各種組合增加或減小系統之間的依賴，ANTICORRUPTION LAYER將相應地收縮或擴張。當然，在其他條件都相同的情況下，應該首先遷移那

些只產生較小ANTICORRUPTION LAYER的功能。但其他因素也可能會起主導作用，有時候在過渡期間可能必須經歷一些麻煩的轉換。

#### **14.14.4 OPEN HOST SERVICE→PUBLISHED LANGUAGE**

我們已經通過一系列特定的協議與其他系統進行了集成，但隨著需要訪問的系統逐漸增多，維護負擔也不斷增加，或者交互變得很難理解。我們需要通過PUBLISHED LANGUAGE來規範系統之間的關係。

- (1) 如果有一種行業標準語言可用，則盡可能評估並使用它。
- (2) 如果沒有標準語言或預先公開發佈的語言，則完善作為HOST的系統的CORE DOMAIN（參見第15章）。
- (3) 使用CORE DOMAIN作為交換語言的基礎，盡可能使用像XML這樣的標準交互範式。
- (4) (至少) 向所有參與協作的各方發佈新語言。
- (5) 如果涉及新的系統架構，那麼也要發佈它。
- (6) 為每個協作系統構建轉換層。
- (7) 切換。

現在，當加入更多協作系統時，對整個系統的破壞已經減至最小了。

記住，PUBLISHED LANGUAGE必須是穩定的，但是當繼續進行重構時，仍然需要能夠自由地更改HOST的模型。因此，不要把交換語言和HOST的模型等同起來。保持它們的密切關係可以減小轉換開銷，而你的HOST可以採用CONFORMIST模式。但是應該保留對轉換層進行補充的權力，在成本一效益的折中需要時，可以把這個權利分離出去。

項目領導者應該根據功能集成需求和開發團隊之間的關係來定義BOUNDED CONTEXT。一旦BOUNDED CONTEXT和CONTEXT MAP被

明確地定義下來並獲得認可，就應該保持它們的邏輯一致性。最起碼要把相關的通信問題提出來，以便解決它們。

但是，有時模型上下文（無論是我們有意識地劃定邊界的還是自然出現的上下文）被錯誤地用來解決系統中的一些其他問題，而不是邏輯不一致問題。團隊可能會發現一個很大的CONTEXT的模型由於過於複雜而無法作為一個整體來理解或透徹地分析。出於有意或無意的考慮，團隊往往會把CONTEXT分割為更易管理的部分。這種分割會導致失去很多機會。現在，值得花費一些功夫仔細考查在一個大的CONTEXT中建立一個大模型的決策了。如果從組織結構或行政角度來看保持一個大模型並不現實，如果實際上模型就是分裂的，那麼就重新繪製上下文圖，並定義能夠保持的邊界。但是，如果保持一個大的BOUNDED CONTEXT能夠解決迫切的集成需要，而且除了模型本身的複雜性以外，這看上去是行得通的，那麼分割CONTEXT可能就不是最佳的選擇了。

在做出這種犧牲之前，還應該考慮其他一些能夠使大模型變得易於管理的方法。下兩章將著重討論通過應用兩種更廣泛的原則（精煉和大型結構）來管理大模型的複雜性。

## 第15章 精煉

$$\nabla \cdot \mathbf{D} = \rho$$

$$\nabla \cdot \mathbf{B} = 0$$

$$\nabla \times \mathbf{E} = -\frac{\partial \mathbf{B}}{\partial t}$$

$$\nabla \times \mathbf{H} = \mathbf{J} + \frac{\partial \mathbf{D}}{\partial t}$$

——James Clerk Maxwell, A Treatise on Electricity and Magnetism, 1873

上面這4個方程式，再加上其中的術語定義，以及它們所依賴的數學體系，表達了19世紀經典電磁學的全部內涵。

如何才能專注於核心問題而不被大量的次要問題淹沒呢？**LAYERED ARCHITECTURE**可以把領域概念從技術邏輯中（技術邏輯確保了計算機系統能夠運轉）分離出來，但在大型系統中，即使領域被分離出來，它的複雜性也可能仍然難以管理。

精煉是把一堆混雜在一起的組件分開的過程，以便通過某種形式從中提取出最重要的內容，而這種形式將使它更有價值，也更有用。模型就是知識的精煉。通過每次重構所得到的更深層的理解，我們得以把關鍵的領域知識和優先級提取出來。現在，讓我們回過頭來從戰

略角度看一下精煉，本章將介紹對模型進行粗線條劃分的各種方式，並把領域模型作為一個整體進行精煉。

像很多化學蒸餾過程一樣，精煉過程所分離出來的副產品（如 **GENERIC SUBDOMAIN** 和 **COHERENT MECHANISM**）本身也很有價值，但精煉的主要動機是把最有價值的那部分提取出來，正是這個部分使我們的軟件區別於其他軟件並讓整個軟件的構建物有所值，這個部分就是**CORE DOMAIN**。

領域模型的戰略精煉包括以下部分：

- (1) 幫助所有團隊成員掌握系統的總體設計以及各部分如何協調工作；
- (2) 找到一個具有適度規模的核心模型並把它添加到通用語言中，從而促進溝通；
- (3) 指導重構；
- (4) 專注於模型中最有價值的那部分；
- (5) 指導外包、現成組件的使用以及任務委派。

本章將展示對**CORE DOMAIN**進行戰略精煉的系統性方法，解釋如何在團隊中有效地統一認識，並提供一種用於討論工作的語言。

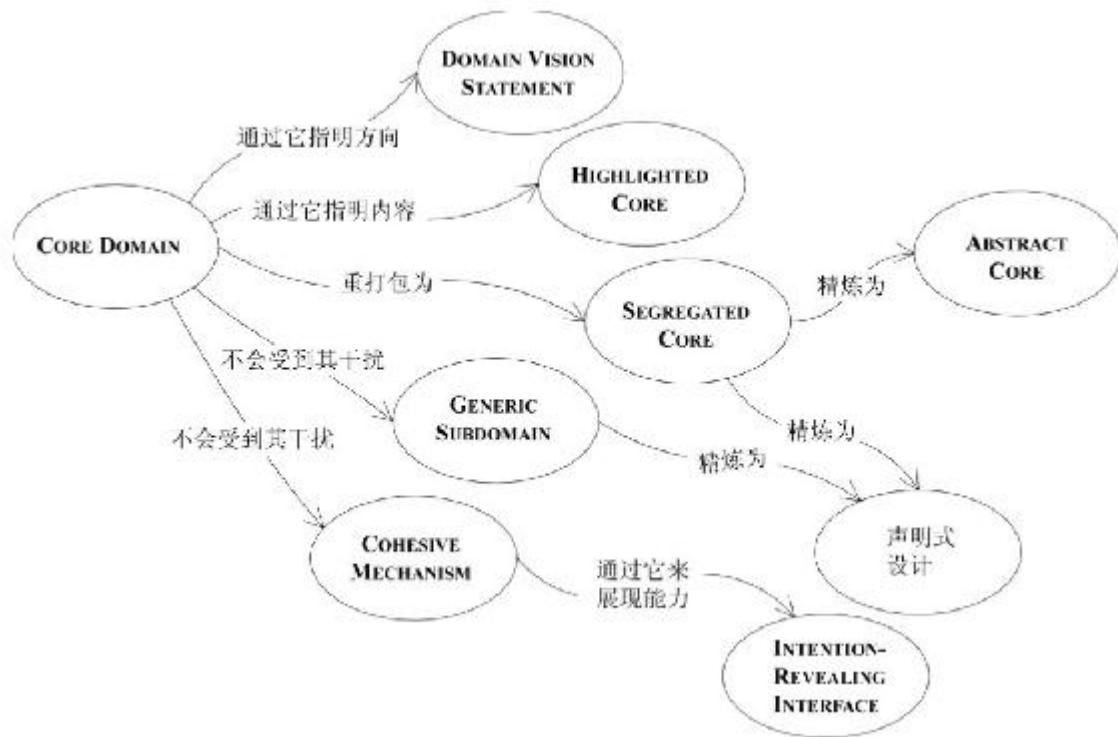


圖15-1 戰略精煉的導航圖

像那些園丁為了讓樹幹快速生長而修剪樹苗一樣，我們將使用一整套技術把模型中那些細枝末節砍掉，從而把注意力集中在最重要的部分上.....

## 15.1 模式：**CORE DOMAIN**



在設計大型系統時，有非常多的組成部分——它們都很複雜而且對開發的成功也至關重要，但這導致真正的業務資產——領域模型最為精華的部分——被掩蓋和忽略了。

難以理解的系統修改起來會很困難，而且修改的結果也難以預料。開發人員如果脫離自己熟悉的領域，也會迷失方向（當團隊中有新人加入時尤其如此，但老成員也面臨同樣的狀況，除非代碼表達得非常清楚並且組織有序）。這樣一來就必須分門別類地為人們安排任務。當開發人員把他們的工作限定到具體的模塊時，知識的傳遞就更少了。這種工作上的劃分導致系統很難平滑地集成，也無法靈活地分配工作。如果開發人員沒有瞭解到某項功能已經被實現了，那麼就會出現重複，這樣系統會變得更加複雜。

以上只是難以理解的設計所導致的一部分後果。當失去了領域的整體視圖時，還存在另一個同樣嚴重的風險。

一個嚴峻的現實是我們不可能對所有設計部分進行同等的精化，而是必須分出優先級。為了使領域模型成為有價值的資產，必須整齊地梳理出模型的真正核心，並完全根據這個核心來創建應用程序的功能。但本來就稀缺的高水平開發人員往往會把工作重點放在技術基礎設施上，或者只是去解決那些不需要專門領域知識就能理解的領域問題（這些問題都已經有了很好的定義）。

計算機科學家對系統的這些部分更感興趣，他們認為通過這些工作可以讓自己具備一些在其他地方也能派上用場的專業技能，同時也豐富了個人簡歷。而真正體現應用程序價值並且使之成為業務資產的領域核心卻通常是由那些技術水平稍差的開發人員完成的，他們與DBA一起創建數據模式，然後逐個特性編寫代碼，而根本沒有對模型的概念能力加以任何利用。

如果軟件的這個部分實現得很差，那麼無論技術基礎設施有多好，無論支持功能有多完善，應用程序永遠都不會為用戶提供真正有吸引力的功能。這個嚴重問題的根源在於項目沒有一個明確的整體設計視圖，而且也沒有認清各個部分的相對重要性。

我曾經參與過的最成功的項目中，有一個開始時就受到了這種問題的困擾。這個項目的目標是開發一個非常複雜的聯合貸款系統。技術能力最強的開發人員在數據庫映射層和消息傳遞接口這些工作上忙得不亦樂乎，而業務模型則交到了那些不熟悉對像技術的新手中。

唯一的例外是有一個領域問題是由一位經驗豐富的對象開發人員處理的，他為那些長期存在的領域對像設計了一種添加註釋的功能。通過把這些註釋組織到一起，交易商能夠看到他們或其他人過去所做的一些決策的基本思想。這位開發人員還構建了一個優秀的用戶界面，它為用戶提供了直觀的訪問，用戶可以利用這個界面來靈活地使用註釋模型的各種功能。

這些特性很有用處，而且設計得很好。它們被合併到最終產品中。

遺憾的是，它們只是一些次要特性。這位能力超群的開發人員把一種有趣的、通用的註釋方法建模出來，並乾淨利落地實現了它，最後交付到用戶手中。同時，另一位能力上不太勝任的開發人員卻把關鍵的「貸款」模塊弄得一團糟，項目差一點就因此失敗。

在制定項目規劃的時候，必須把資源分配給模型和設計中最關鍵的部分。要想達到這個目的，在規劃和開發期間每個人都必須識別和理解這些關鍵部分。

這些部分是應用程序的標誌性部分，也是目標應用程序的核心訴求，它們構成了**CORE DOMAIN**。**CORE DOMAIN**是系統中最有價值的部分。

因此：

對模型進行提煉。找到**CORE DOMAIN**並提供一種易於區分的方法把它與那些起輔助作用的模型和代碼分開。最有價值和最專業的概念要輪廓分明。盡量壓縮**CORE DOMAIN**。

讓最有才能的人來開發**CORE DOMAIN**，並據此要求進行相應的招聘。在**CORE DOMAIN**中努力開發能夠確保實現系統藍圖的深層模型和柔性設計。仔細判斷任何其他部分的投入，看它是否能夠支持這個提煉出來的**CORE**。

提煉**CORE DOMAIN**不容易，但它確實會讓一些決策變得容易。你需要投入大量的工作使你的**CORE**鮮明突出，而其他設計部分則只需依照常規做得實用即可。如果某個設計部分需要保密以便保持競爭優勢，那麼它就是你的**CORE DOMAIN**。其他的部分則沒有必要隱藏起來。當必須在兩個看起來都很有用的重構之間進行抉擇時（由於時限的緣故），應該首選對**CORE DOMAIN**影響最大的那個重構。

本章中的模式能夠使我們更容易發現、使用和修改CORE DOMAIN。

### 15.1.1 選擇核心

我們需要關注的是那些能夠表示業務領域並解決業務問題的模型部分。

對CORE DOMAIN的選擇取決於看問題的角度。例如，很多應用程序需要一個通用的貨幣模型，用來表示各種貨幣以及它們的匯率和兌換。另一方面，一個用來支持貨幣交易的應用程序可能需要更精細的貨幣模型，這個模型有可能就是CORE的一部分。即使在這種情況下，貨幣模型中可能有一部分仍是非常通用的。隨著對領域理解的不斷加深，精煉過程可以持續進行，這會把通用的貨幣概念分離出來，而只把模型中那些專有的部分保留在CORE DOMAIN中。

在運輸應用程序中，CORE可能是以下幾方面的模型：貨物是如何裝船運輸的，當集裝箱轉交時責任是如何轉接的，或者特定的集裝箱是如何經由不同的運輸路線最後到達目的地的。在投資銀行中，CORE可能包括委託人和參與者之間的合資模型。

一個應用程序的CORE DOMAIN在另一個應用程序中可能只是通用的支持組件。儘管如此，仍然可以在一個項目中（而且通常在一個公司中）定義一個一致的CORE。像其他設計部分一樣，人們對CORE DOMAIN的認識也會隨著迭代而發展。開始時，一些特定關係可能顯得不重要。而最初被認為是核心的對象可能逐漸被證明只是起支持作用。

下面幾節（特別是GENERIC SUBDOMAIN這節）將給出制定這些決策的指導。

### 15.1.2 工作的分配

在項目團隊中，技術能力最強的人員往往缺乏豐富的領域知識。這限制了他們的作用，並且更傾向於分派他們來開發一些支持組件，從而形成了一個惡性循環——知識的缺乏使他們遠離了那些能夠學到領域知識的工作。

打破這種惡性循環是很重要的，方法是建立一支由開發人員和一位或多位領域專家組成的聯合團隊，其中開發人員必須能力很強、能夠長期穩定地工作並且對學習領域知識非常感興趣，而領域專家則要掌握深厚的業務知識。如果你認真對待領域設計，那麼它就是一項有趣且充滿技術挑戰的工作。你肯定也會找到持這種觀點的開發人員。

從外界聘請一些短期的專業人員來設計**CORE DOMAIN**的關鍵環節通常是行不通的，因為團隊需要積累領域知識，而且短期人員會造成知識流失。相反，充當培訓和指導角色的專家可能非常有價值，因為他們幫助團隊建立領域設計技巧，並促進團隊成員使用尚未掌握的高級設計原則。

出於類似的原因，購買**CORE DOMAIN**也是行不通的。人們已經在建立特定於行業的模型框架方面付出了一些工作，著名的例子就是半導體行業協會**SEMATECH**創立的用於半導體製造自動化的**CIM**框架，以及**IBM**為很多業務開發的**San Francisco**框架。雖然這是一個有吸引力的想法，但除了能夠促進數據交換的**PUBLISHED LANGUAGE**（參見第 14 章）以外，其他結果並不理想。*Domain-Specific Application Frameworks*[Fayad and Johnson 2000]一書介紹了這項工作的總體狀況。隨著這個領域的進步，可能會出現一些更有用的框架。

除了上述原因之外，還有一個更重要的原因需要引起我們的注意。自主開發的軟件的最大價值來自於對**CORE DOMAIN**的完全控制。一個設計良好的框架可能會提供滿足你的專門使用需求的高水平

抽象，它可以節省開發那些更通用部分的時間，並使你能夠專注於CORE。但是，如果它對你的約束超出了這個限度，可能有以下3種原因。

(1) 你正在失去一項重要的軟件資產。此時應該讓這些限制性的框架退出你的CORE DOMAIN。

(2) 框架所處理的部分並不是你所認為的核心。此時應該重新劃定CORE DOMAIN的邊界，把你的模型中真正的標誌性部分識別出來。

(3) 你的CORE DOMAIN並沒有特殊的需求。此時應該考慮採用一種風險更低的解決方案，如購買軟件並與你的應用程序進行集成。

不管是哪種情況，創建與眾不同的軟件還是會回到原來的軌道上——需要一支穩定工作的團隊，他們不斷積累和消化專業知識，並將這些知識轉化為一個豐富的模型。沒有捷徑，也沒有魔法。

## 15.2 精煉的逐步提升

本章接下來將要介紹各種精煉技術，它們在使用順序上基本沒什麼要求，但對設計的改動卻大不相同。

一份簡單的DOMAIN VISION STATEMENT（領域願景說明）只需很少的投入，它傳達了基本概念以及它們的價值。HIGHLIGHTED CORE（突出核心）可以增進溝通，並指導決策制定，這也只需對設計進行很少的改動甚至無需改動。

更積極的精煉方法是通過重構和重新打包顯式地分離出GENERIC SUBDOMAIN，然後單獨進行處理。在使用COHESIVE MECHANISM的同時，也要保持設計的通用性、易懂性和柔性，這兩個方面可以結合起來。只有除去了這些細枝末節，才能把CORE剝離出來。

重新打包出一個SEGREGATED CORE（分離的核心），可以使這個CORE清晰可見（即使在代碼中也是如此），並且促進將來在CORE模型上的工作。

最富雄心的精煉是ABSTRACT CORE（抽象內核），它用純粹的形式表示了最基本的概念和關係（因此，需要對模型進行全面的重新組織和重構）。

每種技術都需要我們連續不斷地投入越來越多的工作，但刀磨得越薄，就會越鋒利。領域模型的連續精煉將為我們創造一項資產，使項目進行得更快、更敏捷、更精確。

首先，我們可以把模型中最普通的那些部分分離出去，它們就是GENERIC SUBDOMAIN（通用子領域）。GENERIC SUBDOMAIN與CORE DOMAIN形成鮮明的對比，使我們可以更清楚地理解它們各自的含義。

### **15.3 模式：GENERIC SUBDOMAIN**

模型中有些部分除了增加複雜性以外並沒有捕捉或傳遞任何專門的知識。任何外來因素都會使CORE DOMAIN愈發的難以分辨和理解。模型中充斥著大量眾所周知的一般原則，或者是專門的細節，這些細節並不是我們的主要關注點，而只是起到支持作用。然而，無論它們是多麼通用的元素，它們對實現系統功能和充分表達模型都是極為重要的。

模型中有你想當然的部分。不可否認，它們確實是領域模型的一部分，但它們抽象出來的概念是很多業務都需要的。比如，各個行業（如運輸業、銀行業或製造業）都需要某種形式的企業組織圖。再比

如，很多應用程序都需要跟蹤應收賬款、開支分類賬和其他財務事項，而這些都可以用一個通用的會計模型來處理。

通常，人們投注了大量精力去處理領域的周邊問題。我親眼目睹過兩個不同項目都分派了最好的開發人員來重新設計帶有時區的日期和時間功能，這些工作耗費了他們數周的時間。雖然這樣的組件必須正常工作，但它們並不是系統的概念核心。

即使這樣的通用模型元素確實非常重要，整個領域模型仍然需要把系統中最有價值和最特別的方面突出出來，而且整個模型的組織應該盡可能把重點放在這個部分上。當核心與所有相關的因素混雜在一起時，這一點會更難做到。

因此：

識別出那些與項目意圖無關的內聚子領域。把這些子領域的通用模型提取出來，並放到單獨的**MODULE**中。任何專有的東西都不應放在這些模塊中。

把它們分離出來以後，在繼續開發的過程中，它們的優先級應低於**CORE DOMAIN**的優先級，並且不要分派核心開發人員來完成這些任務（因為他們很少能夠從這些任務中獲得領域知識）。此外，還可以考慮為這些**GENERIC SUBDOMAIN**使用現成的解決方案或「公開發佈的模型」（**PUBLISHED MODEL**）。

當開發這樣的軟件包時，有以下幾種選擇。

### 選擇1：現成的解決方案

有時可以購買一個已實現好的解決方案，或使用開源代碼。

優點

可以減少代碼的開發。

維護負擔轉移到了外部。

代碼已經在很多地方使用過，可能較為成熟，因此比自己開發的代碼更可靠和完備。

#### 缺點

在使用之前，仍需要花時間來評估和理解它。

就業內目前的質量控制水平而言，無法保證它的正確性和穩定性。

它可能設計得過於細緻了（遠遠超出了你的目的），集成的工作量可能比開發一個最小化的內部實現更大。

外部元素的集成常常不順利。它可能有一個與你的項目完全不同的**BOUNDED CONTEXT**。

即使不是這樣，它也很難順利地引用你的其他軟件包中的**ENTITY**。

它可能會引入對平臺、編譯器版本的依賴等。

現成的子領域解決方案是值得我們去考慮的，但如果它們常常會帶來麻煩，那麼往往就得不償失了。我曾經看到過一些成功案例——一些應用程序需要非常精細的工作流，它們通過**API掛鉤**（API hook）成功地使用了商用的外部工作流系統。我曾經還見過錯誤日誌被深入地集成到應用程序中。有時，**GENERIC SUBDOMAIN**被打包為框架的形式，它實現了非常抽象的模型，從而可以與你的應用程序集成來滿足你的特殊需求。子組件越通用，其自己的模型的精煉程度越高，它的用處可能就越大。

## 選擇2：公開發佈的設計或模型

#### 優點

比自己開發的模型更為成熟，並且反映了很多人的深層知識。

提供了隨時可用的高質量文檔。

#### 缺點

可能不是很符合你的需要，或者設計得過於細緻了（遠遠超出了你的需要）。

Tom Lehrer ( 20世紀50和60年代的喜劇作曲家 ) 曾經講過數學上的成功秘訣是：「抄襲！抄襲。不要讓任何人的工作逃過你的眼睛……但一定要把這叫做研究。」在領域建模中，特別是在攻克 **GENERIC SUBDOMAIN** 時，這是金玉良言。

當有一個被廣泛使用的模型時，如《分析模式》[Fowler 1996]一書中所列舉的那些模型（參見第11章），這種方法最為有效。

如果領域中已經有了一種非常正式且嚴格的模型，那麼就使用它。會計和物理學是我們立即能想到的兩個例子。這些模型不僅精簡和健壯，而且被人們廣泛理解，因此可以減輕目前和將來的培訓負擔（參見10.9.2節）。

如果在一個公開發佈的模式中能夠發現一個簡化的子集，它本身是一致的而且能夠滿足你的要求，那麼就不要強迫自己完全實現一個這樣的模型。如果一個模型已經有人很好地研究過了，並且提供了完備的文檔，甚至已經得到正規化，那麼重新去設計它就沒有意義了。

### 選擇3：把實現外包出去

#### 優點

使核心團隊可以脫身去處理CORE DOMAIN，那才是最需要知識和經驗積累的部分。

開發工作的增加不會使團隊規模無限擴大下去，同時又不會導致 CORE DOMAIN 知識的分散。

強制團隊採用面向接口的設計，並且有助於保持子領域的通用性，因為規格已經被傳遞到外部。

#### 缺點

仍需要核心團隊花費一些時間，因為他們需要與外包人員商量接口、編碼標準和其他重要方面。

當把代碼移交回團隊時，團隊需要耗費大量精力來理解這些代碼。（但是這個開銷比理解專用子領域要小一些，因為通用子領域不需要理解專門的背景知識。）

代碼質量或高或低，這取決於兩個團隊能力的高低。

自動測試在外包中可能起到重要作用。應該要求外包人員為他們交付的代碼提供單元測試。真正有用的方法是為外包的組件詳細說明甚至是編寫自動驗收測試，這有助於確保質量、明確規格並且使這些組件的再集成變得順利。此外，「把實現外包出去」能夠與「公開發佈的設計或模型」完美地組合到一起。

#### 選擇4：內部實現

優點

易於集成。

只開發自己需要的，不做多餘的工作。

可以臨時把工作分包出去。

缺點

需要承受後續的維護和培訓負擔。

很容易低估開發這些軟件包所需的時間和成本。

當然，這也可以與「公開發佈的設計或模型」結合起來使用。

**GENERIC SUBDOMAIN**是你充分利用外部設計專家的地方，因為這些專家不需要深入理解你特有的**CORE DOMAIN**，而且他們也沒有太大的機會學習這個領域。機密性問題可以不用過多關注，因為這些模塊幾乎不涉及專有信息或業務實踐。**GENERIC SUBDOMAIN**可以減輕對那些不瞭解領域知識的人員進行培訓而帶來的負擔。

我相信，隨著時間的推移，CORE模型的範圍將會不斷變窄，而越來越多的通用模型將作為框架被實現出來，或者至少被實現為公開發佈的模型或分析模式。但是現在，大部分模型仍然需要我們自己開發，但把它們與CORE DOMAIN模型區分開是很有價值的。

### 示例 兩個與時區有關的故事

我曾經兩次親眼目睹項目中最好的開發人員花費好幾周的時間來解決各個時區的時間存儲和轉換問題。雖然我對這樣的工作安排總是持懷疑態度，但有時它是必要的，而且下面這兩個項目幾乎形成了鮮明的對比。

第一個項目是為貨物運輸系統設計日程安排軟件。為了安排國際運輸，準確的時間計算是非常必要的，而由於所有這些日程安排都是按照當地時間計算的，因此運輸過程的安排必然需要進行時間轉換。

既然這項功能需求已經確定了，團隊就開始了CORE DOMAIN的開發並利用現有的時間類和一些啞數據進行了一些早期的應用程序迭代。隨著應用程序不斷成熟，現有的時間類已無法滿足項目的要求，而且由於很多國家的時間是不同的，再加上國際日期變更線的複雜性，這個問題變得異常複雜。此時，他們的需求更加明確了，他們開始尋找現成的解決方案，但卻沒有找到。這樣，除了自己構建之外已經別無選擇了。

這項任務需要做一番研究並進行精確的設計，因此團隊領導打算分派一位最好的程序員來完成它。但這項任務並不需要任何運輸方面的專業知識，而且做這項任務也不會獲得這樣的知識，因此他們選擇了一位臨時在項目上工作的程序員。

這位程序員並沒有從頭開始工作。他研究了幾個現有的時區實現，但大部分並不能滿足需要，於是決定把BSD Unix的一個公共的

解決方案改造一下，它已經有了一個完善的數據庫和C語言實現。他通過逆向工程找出了其中的邏輯，並編寫了一個數據庫導入例程。

事實證明問題比他預計的要難得多（如涉及特殊情況的數據庫導入），儘管如此他仍然完成了代碼的編寫並與CORE進行了集成，最終完成了產品的交付。

在另一個項目上發生的事情就完全不同了。一家保險公司開發一個新的理賠處理系統，他們打算把各種事件發生的時間記錄下來（發生車禍的時間、下冰雹的時間等）。這些數據是按照當地時間記錄的，因此需要用到時區功能。

當我參加這個項目時，他們已經安排了一位初級（但很聰明的）開發人員來從事這項任務，儘管應用程序的準確需求仍在變化中，而且項目甚至還沒有開始嘗試第一次迭代。他已經開始盡職盡責地基於假設來構建一個時區模型。

由於不知道需要什麼樣的功能，這位開發人員假設時區組件應該足夠靈活，以便處理任何可能的情況。這個問題對他來說太難了，因此項目又分派了一位高級開發人員來幫助他。他們編寫了複雜的代碼，但由於還沒有具體的應用程序使用這些代碼，因此他們根本不知道代碼是否能正確工作。

項目由於種種原因而擱淺，時區代碼從未派上用場。但如果項目不中斷，那麼簡單地存儲標明時區的當地時間可能就足夠了，甚至不需要轉換，因為這些時間數據主要用作參考，而不是用於計算。即使需要轉換，由於所有數據都來自北美洲，時區轉換也相對很簡單。

過分關注時區帶來的主要代價是忽略了CORE DOMAIN模型。如果他們能夠把同樣的精力放在核心模型上，可能早就為自己的應用程序開發出了一個有效的原型和一個初步的、可以工作的領域模型。此

外，那些長期穩定地在項目上工作的開發人員此時本來應該對保險領域有所瞭解了，以便為團隊積累關鍵知識。

有一件事情這兩個團隊都做得很正確，那就是把通用的時區模型明確地從**CORE DOMAIN**中分離出來。如果在運輸模型或保險模型中使用各自專用的時區**MODULE**，那麼這會導致核心模型與這個通用的支持模型耦合在一起，使得**CORE**模型更難以理解（因為它將包含無關的時區細節）。而且時區模塊可能更難維護（因為維護人員必須理解核心以及它與時區的相互關係）。

運輸項目的策略	保險項目的策略
<b>优 点</b> <ul style="list-style-type: none"><li><input type="checkbox"/> <b>GENERIC</b>模型与<b>CORE</b>分离</li><li><input type="checkbox"/> <b>CORE</b>模型较成熟，因此资源的转移不会妨碍它</li><li><input type="checkbox"/> 明确知道需要什么功能</li><li><input type="checkbox"/> 为跨国的日程安排提供了关键支持功能</li><li><input type="checkbox"/> <b>GENERIC</b>模块的任务使用了短期程序员</li></ul> <b>缺 点</b> <ul style="list-style-type: none"><li><input type="checkbox"/> 最好的程序员没有从事核心工作</li></ul>	<b>优 点</b> <ul style="list-style-type: none"><li><input type="checkbox"/> <b>GENERIC</b>模型与<b>CORE</b>分离</li></ul> <b>缺 点</b> <ul style="list-style-type: none"><li><input type="checkbox"/> <b>CORE</b>模型未被开发出来，因此关注其他问题导致核心模型继续被忽略</li><li><input type="checkbox"/> 由于需求不明确，所以试图开发一个能满足所有需求的模块，而实际上只需简单地提供北美地区的时区转换功能就足够了</li><li><input type="checkbox"/> 安排长期工作的程序员来执行这项任务，他们本来应该成为领域知识的储备库</li></ul>

技術人員喜歡處理那些可定義的問題（如時區轉換），而且很容易就能證明他們花時間做這些工作是值得的。但嚴格地從優先級角度來看，他們應該先去完成**CORE DOMAIN**的工作。

### 15.3.1 通用不等於可重用

注意，雖然我一直在強調這些子領域的通用性，但我並沒有提代碼的可重用性。現成的解決方案可能適用於某種特殊情況，也可能不適用，但假設你要自己實現代碼（內部實現或外包出去），那麼不要特別關注代碼的可重用性。因為那樣做會違反精煉的基本動機——我們應該盡可能把大部分精力投入到**CORE DOMAIN**工作中，而只在必要的時候才在支持性的**GENERIC SUBDOMAIN**中投入工作。

重用確實會發生，但不一定總是代碼重用。模型重用通常是更高級的重用，例如，當使用公開發佈的設計或模型的時候就是如此。如果你必須創建自己的模型，那麼它在以後的相關項目中可能很有價值。但是，雖然這樣的模型概念可能適用於很多情況，我們也不必把它開發成「萬能的」模型。我們只要把業務所需的那部分建模出來並實現即可。

儘管我們很少需要考慮設計的可重用性，但通用子領域的設計必須嚴格地限定在通用概念的範圍之內。如果把行業專用的模型元素引入到通用子領域中，會產生兩個後果。第一，它會妨礙將來的開發。雖然現在我們只需要子領域模型的一小部分，但我們的需求會不斷增加。如果把任何不屬於子領域概念的部分引入到設計中，那麼再想靈活地擴展系統就很難了，除非完全重建原來的部分並重新設計使用該部分的其他模塊。

第二，也是更重要的，這些行業專用的概念要麼屬於CORE DOMAIN，要麼屬於它們自己的更專業的子領域，而且這些專業的模型比通用子領域更有價值。

### 15.3.2 項目風險管理

敏捷過程通常要求通過盡早解決最具風險的任務來管理風險。特別是XP過程，它要求迅速建立並運行一個端到端的系統。這種初步的系統通常用來檢驗某種技術架構，而且人們會試圖建立一個外圍系統，用來處理一些支持性的GENERIC SUBDOMAIN，因為這些子領域通常更易於分析。但是要注意，這可能會不利於風險管理。

項目面臨著兩方面的風險，有些項目的技術風險更大，有些項目則是領域建模的風險更大一些。端到端的系統是實際系統中最困難部分的「雛形」——它控制風險的能力也僅限於此。當使用這種雛形時，我們很容易低估領域建模的風險。這種風險包括未預料到存在複

雜性、與業務專家的交流不夠充分，或者開發人員的關鍵技能存在欠缺等。

因此，除非團隊擁有精湛的技術並且對領域非常熟悉，否則第一個雛形系統應該以CORE DOMAIN的某個部分作為基礎，不管它有多麼簡單。

相同的原則也適用於任何試圖把高風險的任務放到前面處理的過程。CORE DOMAIN就是高風險的，因為它的難度往往會超出我們的預料，而且如果沒有它，項目就不可能獲得成功。

本章介紹的大多數精煉模式都展示瞭如何修改模型和代碼，以便提煉出CORE DOMAIN。但是，接下來的兩個模式DOMAIN VISION STATEMENT和HIGHLIGHTED CORE將展示如何用最少的投入通過補充文檔來增進溝通、提高人們對核心的認識並使之把開發工作集中到CORE上來.....

## 15.4 模式：DOMAIN VISION STATEMENT

在項目開始時，模型通常並不存在，但是模型開發的需求是早就確定下來的重點。在後面的開發階段，我們需要解釋清楚系統的價值，但這並不需要深入地分析模型。此外，領域模型的關鍵方面可能跨越多個BOUNDED CONTEXT，而且從定義上看，無法將這些彼此不同的模型組織起來表明其共同的關注點。

很多項目團隊都會編寫「願景說明」以便管理。最好的願景說明會展示出應用程序為組織帶來的具體價值。一些願景說明會把創建領域模型當作一項戰略資產。通常，願景說明文檔在項目啟動以後就被棄之不用了，而在實際開發過程中從來不會使用它，甚至根本不會有技術人員去閱讀它。

DOMAIN VISION STATEMENT就是模仿這類文檔創建的，但它關注的重點是領域模型的本質，以及如何為企業帶來價值。在項目開發的所有階段，管理層和技術人員都可以直接用領域願景說明來指導資源分配、建模選擇和團隊成員的培訓。如果領域模型為多個群體提供服務，那麼此文檔還能夠顯示出他們的利益是如何均衡的。

因此：

寫一份**CORE DOMAIN**的簡短描述（大約一頁紙）以及它將會創造的價值，也就是「價值主張」。那些不能將你的領域模型與其他領域模型區分開的方面就不要寫了。展示出領域模型是如何實現和均衡各方利益的。這份描述要盡量精簡。盡早把它寫出來，隨著新的理解隨時修改它。

DOMAIN VISION STATEMENT可以用作一個指南，它幫助開發團隊在精煉模型和代碼的過程中保持統一的方向。團隊中的非技術成員、管理層甚至是客戶也都可以共享領域願景說明（當然，包含專有信息的情況除外）。

以下兩個表格分別包含了航班預定系統和半導體工廠自動化系統的DOMAIN VISION STATEMENT。

以下內容是DOMAIN VISION STATEMENT的一部分 以下內容雖然很重要，但它不是DOMAIN VISION STATEMENT的一部分

航班預訂系統	航班預訂系統
模型可以表示出乘客的優先級和航班預訂策略，并根據靈活的政策來平衡這些方面。乘客模型應該反映出航空公司努力發展與回头客的關係這一點。因此，它應該用簡明的形式表示出乘客的歷史記錄、參與過的特殊活動以及與戰略企業客戶的關係等。	用戶界面應該兼顧新老用戶，讓老用戶能夠快速流暢地操作，讓新用戶也能易于使用。
表示出不同用戶的不同角色（如乘客、代理商、經理），以便丰富關係模型並為安全框架提供所需的信息。	系統將提供Web訪問，可以把數據傳輸到其他系統，或通過其他的UI提供訪問，因此接口應該用XML來設計，并使用轉換層來服務Web頁面或把數據轉換到其他系統中。
模型應該支持高效的航線/座位搜索，並與其他已有的航空預訂系統集成。	彩色的動畫logo將緩存到客戶機器上，以便將來訪問時能夠快速顯示。

以下內容是DOMAIN VISION STATEMENT的一部分 以下內容雖然很重要，但它不是DOMAIN VISION STATEMENT的一部分

半導體工廠自動化	半導體工廠自動化
領域模型將表示用材料和設備在芯片廠中的狀態，以便提供必要的審計跟蹤，並支持自動化的工藝流程。	軟件應該能夠通過一個scrivter提供Web訪問，但它的結構應該允許使用不同的接口。
模型不包括工藝流程中所需的人力資源，但必須通過下級工藝配方來實現有選擇性的流程自動化。	尽可能使用行业标准的技术，以避免内部开发，减少维护成本，并最大限度地利用外部的专业资源。应该把开源解决方案作为首选（如Apache Web服务器）。
工廠狀況的描述應該使管理人員能夠理解，以便使他們有更深入的認識並制定更好的決策。	Web服務器將在專用服務器上運行。應用程序將在另一台單獨的專用服務器上運行。

DOMAIN VISION STATEMENT為團隊提供了統一的方向。但在高層次的說明和代碼或模型的完整細節之間通常還需要做一些銜接.....

## **15.5 模式：HIGHLIGHTED CORE**

DOMAIN VISION STATEMENT從寬泛的角度對CORE DOMAIN進行了說明，但它把什麼是具體核心模型元素留給人們自己去解釋和猜測。除非團隊的溝通極其充分，否則單靠VISION STATEMENT是很難產生什麼效果的。

儘管團隊成員可能大體上知道核心領域是由什麼構成的，但CORE DOMAIN中到底包含哪些元素，不同的人會有不同的理解，甚至同一個人在不同的時間也會有不同的理解。如果我們總是要不斷過濾模型以便識別出關鍵部分，那麼就會分散本應該投入到設計上的精力，而且這還需要廣泛的模型知識。因此，CORE DOMAIN必須要很容易被分辨出來。

對代碼所做的重大結構性改動是識別CORE DOMAIN的理想方式，但這些改動往往無法在短期內完成。事實上，如果團隊的認識還不夠全面，這樣的重大代碼修改是很難進行的。

通過修改模型的組織結構（如劃分GENERIC SUBDOMAIN和本章後面要介紹的一些改動），可以用MODULE表達出核心領域。但如果

把它作為表達CORE DOMAIN的唯一方法，那麼對模型的改動會很大，因此很難馬上看到結果。

我們可能需要用一種輕量級的解決方案來補充這些激進的技術手段。可能有一些約束使你無法從物理上分離出CORE，或者你可能是從已有代碼開始工作的，而這些代碼並沒有很好地區分出CORE，但你確實很需要知道什麼是CORE並建立起共識，以便有效地通過重構進行更好的精煉。即使到了高級階段，通過仔細挑選幾個圖或文檔，也能夠為團隊提供思考的定位點和切入點。

無論是使用了詳盡的UML模型的項目，還是那些只使用很少的外部文檔並且把代碼用作主要的模型存儲庫的項目（如XP項目），都會面臨這些問題。極限編程團隊可能採用更簡潔的做法，他們更少地使用這些補充解決方案，而且只是臨時使用（例如，在牆上掛一張手繪的圖，讓所有人都能看到），但這些技術可以很好地結合到開發過程中。

把模型的一個特別部分連同它的實現一起區分出來，這只是對模型的一種反映，而不必是模型自身的一部分。任何使人們易於瞭解CORE DOMAIN的技術都可以採用。這類解決方案有兩種典型的代表性技術。

### **15.5.1 精煉文檔**

我經常會創建一個單獨的文檔來描述和解釋CORE DOMAIN。這個文檔可能很簡單，只是最核心的概念對象的清單。它可能是一組描述這些對象的圖，顯示了它們最重要的關係。它可能在抽象層次上或通過示例來描述基本的交互過程。它可能會使用UML類圖或序列圖、專用於領域的非標準的圖、措辭嚴謹的文字解釋或上述這些元素的組合。精煉文檔並不是完備的設計文檔。它只是一個最簡單的切入點，描述並解釋了核心，並給出了更進一步研究這些核心部分的理由。精

煉文檔為讀者提供了一個總體視圖，指出了各個部分是如何組合到一起的，並且指導讀者到相應的代碼部分尋找更多細節。

因此（作為HIGHLIGHTED CORE（突出核心）的一種形式）：

編寫一個非常簡短的文檔（3~7頁，每頁內容不必太多），用於描述**CORE DOMAIN**以及**CORE**元素之間的主要交互過程。

獨立文檔帶來的所有常見風險也會在這裡出現：

- (1) 文檔可能得不到維護；
- (2) 文檔可能沒人閱讀；
- (3) 由於有多個信息來源，文檔可能達不到簡化複雜性的目的。

控制這些風險的最好方法是保持絕對的精簡。剔除那些不重要的細節，只關注核心抽像以及它們的交互，這樣文檔的老化速度就會減慢，因為這個層次的模型通常更穩定。

精煉文檔應該能夠被團隊中的非技術人員理解。把它當作一個共享的視圖，描述每個人都應該知道的東西，而且可以把它作為團隊所有成員研究模型和代碼的一個起點。

### **15.5.2 標明CORE**

我以前參加過一家大型保險公司的項目，在上班的第一天，有人給了我一份200頁的「領域模型」文檔的複印件，這個文檔是花高價從一家行業協會購買的。我花了幾天時間仔細研究了一大堆類圖，它們涵蓋了所有細節，從詳細的保險政策組合到人們之間極為抽像的關係模型。這些模型的質量也參差不齊，有的只有高中生的水平，有的卻相當好（有幾個甚至描述了業務規則，至少在附帶的文本中做了描述）。但我要從哪裡開始工作呢？要知道它有200頁啊。

這個項目的人員熱衷於構建抽像框架，我的前任們非常關注人與人之間、人與事物之間以及人與活動或協議之間的抽像關係模型。他們確實對關係進行了很好的分析，而且模型實驗也達到了專業研究項

目的水準，但卻並沒有使我們找到開發這個保險應用程序的任何思路。

我對它的第一反應就是大幅刪減，找到一個小的CORE DOMAIN並重構它，然後再逐步添加其他細節。但我的這個觀點使管理層感到擔心。這份文檔具有極大的權威性。它是由整個行業的專家們編寫的，而且無論如何他們付給協會的費用遠遠超過付給我的費用，因此他們不太可能慎重考慮我所提出的要進行徹底修改的建議。但我知道必須有一個共享的CORE DOMAIN視圖，並讓每個人的工作都以它為中心。

我沒有進行重構，而是走查了文檔，並且還得到了一位既懂得大量保險業一般知識又瞭解我們這個特殊應用程序的具體需求的業務分析師的幫助，把那些體現出基本的、區別於其他系統概念的部分標識出來，這些是我們真正需要處理的部分。我提供了一個模型的導航圖，它清晰地顯示了核心，以及它與支持特性的關係。

我們從這個角度開始了建立原型的新工作，很快就開發出了一個簡化的應用程序，它展示了一些必需的功能。

這沓兩磅重的再生紙變成了一項有用的業務資產，而我做的只是加了少量的頁標和一些黃色標記。

這種技術並不僅限於紙面上的對象圖。使用大量UML圖的團隊可以使用一個「原型」（Stereotype）來識別核心元素。把代碼用作唯一模型存儲庫的團隊可以使用註釋（可以採用Java Doc這樣的結構），或使用開發環境中的一些工具。使用哪種特定技術都沒關係，只要使開發人員容易分辨出什麼在核心領域內，什麼在核心領域外就可以了。

因此（作為另一種形式的HIGHLIGHTED CORE）：

把模型的主要存儲庫中的**CORE DOMAIN**標記出來，不用特意去闡明其角色。使開發人員很容易就知道什麼在核心內，什麼在核心外。

現在，我們只做了很少的處理和維護工作，負責處理模型的人員就已經清晰地看到**CORE DOMAIN**了，至少模型已經被整理得很好，使人們很容易分清各個部分的組成。

### 15.5.3 把精煉文檔作為過程工具

理論上，在XP項目上工作的任何結對成員（兩位一起工作的程序員）都可以修改系統中的任何代碼。但在實際中，一些修改會產生很大影響，因此需要更多的商量和協調。按照項目通常的組織形式，當在基礎設施層中工作時，變更的影響可能很清楚；但在領域層中，影響就不那麼明顯了。

從**CORE DOMAIN**的概念來看，這種影響會變得清楚。更改**CORE DOMAIN**模型會產生較大的影響。對廣泛使用的通用元素進行修改可能要求更新大量的代碼，但不會像**CORE DOMAIN**修改那樣產生概念上的變化。

把精煉文檔作為一個指南。如果開發人員發現精煉文檔本身需要修改以便與他們的代碼或模型修改保持同步，那麼這樣的修改需要大家一起協商。這種修改要麼是從根本上修改**CORE DOMAIN**元素或關係；要麼是修改**CORE DOMAIN**的邊界，把一些元素包含進來，或是把一些元素排除出去。不管使用什麼溝通渠道（包括新版本的精煉文檔的分發），模型的修改都必須傳達到整個團隊。

如果精煉文檔概括了**CORE DOMAIN**的核心元素，那麼它就可以作為一個指示器——用以指示模型改變的重要程度。當模型或代碼的修改影響到精煉文檔時，需要與團隊其他成員一起協商。當對精煉文檔做出修改時，需要立即通知所有團隊成員，而且要把新版本的文檔

分發給他們。**CORE**外部的修改或精煉文檔外部的細節修改則無需協商或通知，可以直接把它們集成到系統中，其他成員在後續工作過程中自然會看到這些修改。這樣開發人員就擁有了**XP**所建議的完全的自治性。

儘管**VISION STATEMENT**和**HIGHLIGHTED CORE**可以起到通知和指導的作用，但它們本身並沒有修改模型或代碼。具體地劃分**GENERIC SUBDOMAIN**可以除去一些非核心元素。接下來的幾個模式著眼於從結構上修改模型和設計本身，目的是使**CORE DOMAIN**更明顯，更易於管理。

## **15.6 模式：COHESIVE MECHANISM**

封裝機制是面向對像設計的一個基本原則。把複雜算法隱藏到方法中，再為方法起一個一看就知道其用途的名字，這樣就把「做什麼」和「如何做」分開了。這種技術使設計更易於理解和使用。然而它也有一些先天的侷限性。

計算有時會非常複雜，使設計開始變得膨脹。機械性的「如何做」大量增加，把概念性的「做什麼」完全掩蓋了。為解決問題提供算法的大量方法掩蓋了那些用於表達問題的方法。

這種方法的擴散是模型出問題的一種症狀。這時應該通過重構得到更深層的理解，從而找到更適合解決問題的模型和設計元素。首先要尋找的解決方案是找到一個能使計算機制變得簡單的模型。但有時我們會發現，有些計算機制本身在概念上就是內聚的。這種內聚的計算概念可能並不包括我們所需的全部計算。我們討論的也不是一種萬能的計算器。把內聚部分提取出來會使剩下的部分更易於理解。

因此：

把概念上的**COHESIVE MECHANISM**（內聚機制）分離到一個單獨的輕量級框架中。要特別注意公式或那些有完備文檔的算法。用一個**INTENTION-REVEALING INTERFACE**來暴露這個框架的功能。現在，領域中的其他元素就可以只專注於如何表達問題（做什麼）了，而把解決方案的複雜細節（如何做）轉移給了框架。

然後，這些被分離出來的機制承擔起支持的任務，從而留下一個更小的、表達得更清楚的**CORE DOMAIN**，這個核心以更加聲明式的方式通過接口來使用這些機制。

把標準的算法或公式識別出來以後，可以把一部分設計的複雜性轉移到一系列已經過深入研究的概念中。在這種方法的引導下，我們可以放心地實現一個解決方案，而且只需進行很少的嘗試和改錯。我們可以依靠其他一些瞭解這種算法或至少能夠查到相關資料的開發人員。這個好處類似於從公開發佈的**GENERIC SUBDOMAIN**模型獲得的好處，但找到完備的算法或公式計算的機會比利用通用子領域模型的機會更大一些，因為這種水平的計算機科學已經有了較深入的研究。但是，我們仍常常需要創建新的算法。創建的算法應該主要用於計算，避免在算法中混雜用於表達問題的領域模型。二者的職責應該分離。**CORE DOMAIN**或**GENERIC SUBDOMAIN**的模型描述的是事實、規則或問題。而**COHESIVE MECHANISM**則用來滿足規則或者用來完成模型指定的計算。

### 示例 從組織結構圖中分離出一個**COHESIVE MECHANISM**

我曾經在一個項目上經歷過這種分離過程，這個項目需要一種非常詳細的組織結構圖模型。這個模型可以表示出一個人正在為誰工作以及他屬於哪個分支部門，模型還提供了一個接口，通過這個接口可以提出和回答相關的問題。由於大部分問題都類似於「在這個指揮鏈中誰有權批准這件事」或「在這個部門中誰能夠處理這樣的問題」，

因此團隊意識到大部分複雜性都來自於遍歷組織樹中的特定分支，從中搜索特定的人員或關係。這恰好是成熟的圖系統所能夠解決的問題，圖是一個由弧連接的節點集合（弧叫做邊）以及遍歷圖所需的規則和算法組成。

負責這項工作的開發人員開發出了一個圖的遍歷框架，並把它實現為一種**COHESIVE MECHANISM**。這個框架使用了標準的圖術語和算法，大多數計算機專業人員都很熟悉這些術語和算法，而且它們在教科書中也大量出現。這位開發人員並沒有實現一個完整的概念框架，而只是實現了它的一個子集，該子集涵蓋了組織模型所需的功能。而且由於採用了**INTENTION-REVEALING INTERFACE**，因此獲取答案的方式並不是我們主要關心的問題。

現在，組織模型可以用標準的圖術語簡單地把每個人表示為一個節點，把人們之間的關係表示為連接這些節點的邊（弧）。這樣，使用這個圖框架機制就可以找到任意兩個人之間的關係了。

如果這個機制被混雜到領域模型中，那麼將會產生兩個後果。一是模型會與一個用於解決問題的特殊方法耦合在一起，這將限制將來的選擇。更重要的是，組織的模型將變得異常複雜和混亂。把該機制與模型分開的好處是可以用聲明式的風格來描述組織，使組織結構變得更清晰。而且用於圖操作的複雜代碼被分離到一個單純的、基於成熟算法的機制框架中，從而可以進行單獨的維護和單元測試。

**COHESIVE MECHANISM**的另一個例子是用一個框架來構造**SPECIFICATION**對象，並為這些對像所需的基本的比較和組合操作提供支持。利用這個框架，**CORE DOMAIN**和**GENERIC SUBDOMAIN**可以用**SPECIFICATION**模式中所描述的清晰的、易於理解的語言來聲明它們的規格（參見第10章）。這樣，比較和組合等複雜操作可以留給框架去完成。

## 15.6.1 GENERIC SUBDOMAIN與COHESIVE MECHANISM的比較

GENERIC SUBDOMAIN與COHESIVE MECHANISM的動機是相同的——都是為CORE DOMAIN減負。區別在於二者所承擔的職責的性質不同。GENERIC SUBDOMAIN是以描述性的模型作為基礎的，它用這個模型表示出團隊會如何看待領域的某個方面。在這一點上它與CORE DOMAIN沒什麼區別，只是重要性和專門程度較低而已。COHESIVE MECHANISM並不表示領域，它的目的是解決描述性模型所提出來的一些複雜的計算問題。

模型提出問題，COHESIVE MECHANISM解決問題。

在實踐中，除非你識別出一種正式的、公開發佈的算法，否則這種區別通常並不十分清楚，至少在開始時是這樣。在後續的重構中，如果發現一些先前未識別的模型概念會使這種機制變得更為簡單，那麼就可以把這種算法精煉成一種更純粹的機制，或者轉換為一個GENERIC SUBDOMAIN。

## 15.6.2 MECHANISM是CORE DOMAIN一部分

我們幾乎總是想要把MECHANISM從CORE DOMAIN中分離出去。例外的情況是MECHANISM本身就是專有的並且是軟件的一項核心價值。有時，非常專用的算法就是這種情況。例如，如果一個非常高效的算法（用於計算日程安排）是運輸物流應用程序中的標誌性特性之一，那麼該機制就可以被認為是概念核心的一部分。我以前參加過一個投資銀行的項目，在這個項目中有一個非常專業的風險評估算法，它無疑是CORE DOMAIN的一部分（事實上，這個算法是高度機密的，甚至大部分核心開發人員都看不到它們）。當然，這些算法可能是一個用於預測風險的規則集的特殊實現。通過更深入的分析可能會

得到一個更深層的模型，從而用一種封裝的解決機制把這些規則顯式地表達出來。

但那只是將來要做的進一步改進。是否做這個決定取決於成本—效益分析。實現新設計的難度有多大？當前設計有多難理解和修改？採用更高級的設計後，對從事這些工作的人來說，設計會得到多大程度的簡化？當然，有人對新模型的組成有什麼想法嗎？

### 示例 繞了一圈，MECHANISM又重新回到組織結構圖中

實際上，在我們完成了前面示例中的組織模型一年之後，其他開發人員又重新設計了它，取消了分離的圖框架。他們認為對像數量在增加，而且把這種MECHANISM分離到單獨的包中也會變得很複雜，於是覺得這二者沒必要如此。相反，他們把節點行為添加到組織ENTITY的父類中。但他們保留了組織模型的聲明式公共接口。他們甚至在組織ENTITY中保持了MECHANISM的封裝。

繞彎路之後又返回到原來的老路上是很常見的事情，但並不會退回到起點。最終結果通常是得到了一個更深層的模型，這個模型能夠更清楚地區分出事實、目標和MECHANISM。實用的重構在保留中間階段的重要價值的同時還能夠去除不必要的複雜性。

## 15.7 通過精煉得到聲明式風格

聲明式設計和「聲明式風格」是第10章的一個主題，但在本章的戰略精煉這個話題上，有必要特別提一下這種設計風格。精煉的價值在於使你能夠看到自己正在做什麼，不讓無關細節分散你的注意力，並通過不斷削減得到核心。如果領域中那些起到支持作用的部分提供了一種簡練的語言，可用於表示CORE的概念和規則，同時又能夠把計

算或實施這些概念和規則的方式封裝起來，那麼CORE DOMAIN的重要部分就可以採用聲明式設計。

COHESIVE MECHANISM 用途最大的地方是它通過一個INTENTION-REVEALING INTERFACE來提供訪問，並且具有概念上一致的 ASSERTION 和 SIDE-EFFECT-FREE FUNCTION。利用這些MECHANISM和柔性設計，CORE DOMAIN可以使用有意義的聲明，而不必調用難懂的函數。但最不同尋常的回報來自於使CORE DOMAIN的一部分產生突破，得到一個深層模型，而且這部分核心領域本身成為了一種語言，可以靈活且精確地表達出最重要的應用場景。

深層模型往往與相對應的柔性設計一起產生。柔性設計變得成熟的時候，就可以提供一組易於理解的元素，我們可以明確地把它們組合到一起來完成複雜的任務，或表達複雜的信息，就像單詞組成句子一樣。此時，客戶代碼就可以採用聲明式風格，而且更為精煉。

把GENERIC SUBDOMAIN提取出來可以減少混亂，而COHESIVE MECHANISM可以把複雜操作封裝起來。這樣可以得到一個更專注的模型，從而減少了那些對用戶活動沒什麼價值的、分散注意力的方面。但我們不太可能為領域模型中所有非CORE元素安排一個適當的去處。SEGREGATED CORE（分離的核心）採用直接的方法從結構上把CORE DOMAIN劃分出來。

## 15.8 模式：SEGREGATED CORE

模型中的元素可能有一部分屬於CORE DOMAIN，而另一部分起支持作用。核心元素可能與一般元素緊密耦合在一起。CORE的概念內聚性可能不是很強，看上去也不明顯。這種混亂性和耦合關係抑制

了**CORE**。設計人員如果無法清晰地看到最重要的關係，就會開發出脆弱的設計。

通過把**GENERIC SUBDOMAIN**提取出來，可以從領域中清除一些幹擾性的細節，使**CORE**變得更清楚。但識別和澄清所有這些子領域是很困難的工作，而且有些工作看起來並不值得去做。同時，最重要的**CORE DOMAIN**仍然與剩下的那些元素糾纏在一起。

因此：

對模型進行重構，把核心概念從支持性元素（包括定義得不清楚的那些元素）中分離出來，並增強**CORE**的內聚性，同時減少它與其他代碼的耦合。把所有通用元素或支持性元素提取到其他對像中，並把這些對像放到其他的包中——即使這會把一些緊密耦合的元素分開。

這裡基本上採用了與**GENERIC SUBDOMAIN**一樣的原則，只是從另一個方向來考慮而已。那些在應用程序中非常關鍵的內聚子領域可以被識別出來，並分離到它們自己的內聚包中。如何處理剩下那些未加區分的元素雖然也很重要，但其重要性略低。這些元素或多或少地可以保留在原先的位臘，也可以放到包含了重要類的包中。最後，越來越多的剩餘元素可以被提取到**GENERIC SUBDOMAIN**中。但就目前來看，使用哪種簡單解決方案都可以，只需把注意力集中在**SEGREGATED CORE**（分離的核心）上即可。

通過重構得到**SEGREGATED CORE**的一般步驟如下所示。

- (1) 識別出一個**CORE**子領域（可能是從精煉文檔中得到的）。
- (2) 把相關的類移到新的**MODULE**中，並根據與這些類有關的概念為模塊命名。
- (3) 對代碼進行重構，把那些不直接表示概念的數據和功能分離出來。把分離出來的元素放到其他包的類（可以是新的類）中。盡量

把它們與概念上相關的任務放在一起，但不要為了追求完美而浪費太長時間。把注意力放在提煉CORE子領域上，並且使CORE子領域對其他包的引用變得更明顯且易於理解。

(4) 對新的SEGREGATED CORE MODULE進行重構，使其中的關係和交互變得更簡單、表達得更清楚，並且最大限度地減少並澄清它與其他MODULE的關係（這將是一個持續進行的重構目標）。

(5) 對另一個CORE子領域重複這個過程，直到完成SEGREGATED CORE的工作。

### **15.8.1 創建SEGREGATED CORE的代價**

有時候，把CORE分離出來會使得它與那些緊密耦合的非CORE類的關係變得更晦澀，甚至更複雜，但CORE DOMAIN更清晰了，而且更易於處理，因此獲得的好處還是足以抵償這種代價。

SEGREGATED CORE使我們能夠提高CORE DOMAIN的內聚性。我們可以使用很多有意義的方式來分解模型，有時在創建SEGREGATED CORE時，可以把一個內聚性很好的MODULE拆分開，通過犧牲這種內聚性來換取CORE DOMAIN的內聚性。這樣做是值得的，因為企業軟件的最大價值來自於模型中企業的那些特有方面。

當然，另一個代價是分離CORE需要付出很大的工作量。我們必須認識到，在做出SEGREGATED CORE的決定時，有可能需要開發人員對整個系統做出修改。

當系統有一個很大的、非常重要的BOUNDED CONTEXT時，但模型的關鍵部分被大量支持性功能掩蓋了，那麼就需要創建SEGREGATED CORE了。

### **15.8.2 不斷發展演變的團隊決策**

就像很多戰略設計決策所要求的一樣，創建SEGREGATED CORE需要整個團隊一致行動。這一行動需要團隊的一致決策，而且團隊必須足夠自律和協調才能執行這樣的決策。困難之處在於既要約束每個人使其都使用相同的CORE定義，又不能一成不變地去執行這個決策。由於CORE DOMAIN也是不斷演變的（像任何其他設計方面一樣），在處理SEGREGATED CORE的過程中我們會不斷積累經驗，這將使我們對什麼是核心什麼是支持元素這些問題產生新的理解。我們應該把這些理解反饋到設計中，從而得到更完善的CORE DOMAIN和SEGREGATED CORE MODULE的定義。

這意味著新的理解必須持續不斷地在整個團隊中共享，但個人（或編程對）不能單方面根據這些理解擅自採取行動。無論團隊採用了什麼樣的決策過程，團隊一致通過也好，由領導者下命令決定也好，決策過程都必須具有足夠的敏捷性，可以反覆糾正。團隊必須進行有效的溝通，以便使每個人都共享同一個CORE視圖。

### 示例 把貨物運輸模型的CORE分離出來

我們從圖15-2所示的模型開始，把它作為貨物運輸調度軟件的基礎。

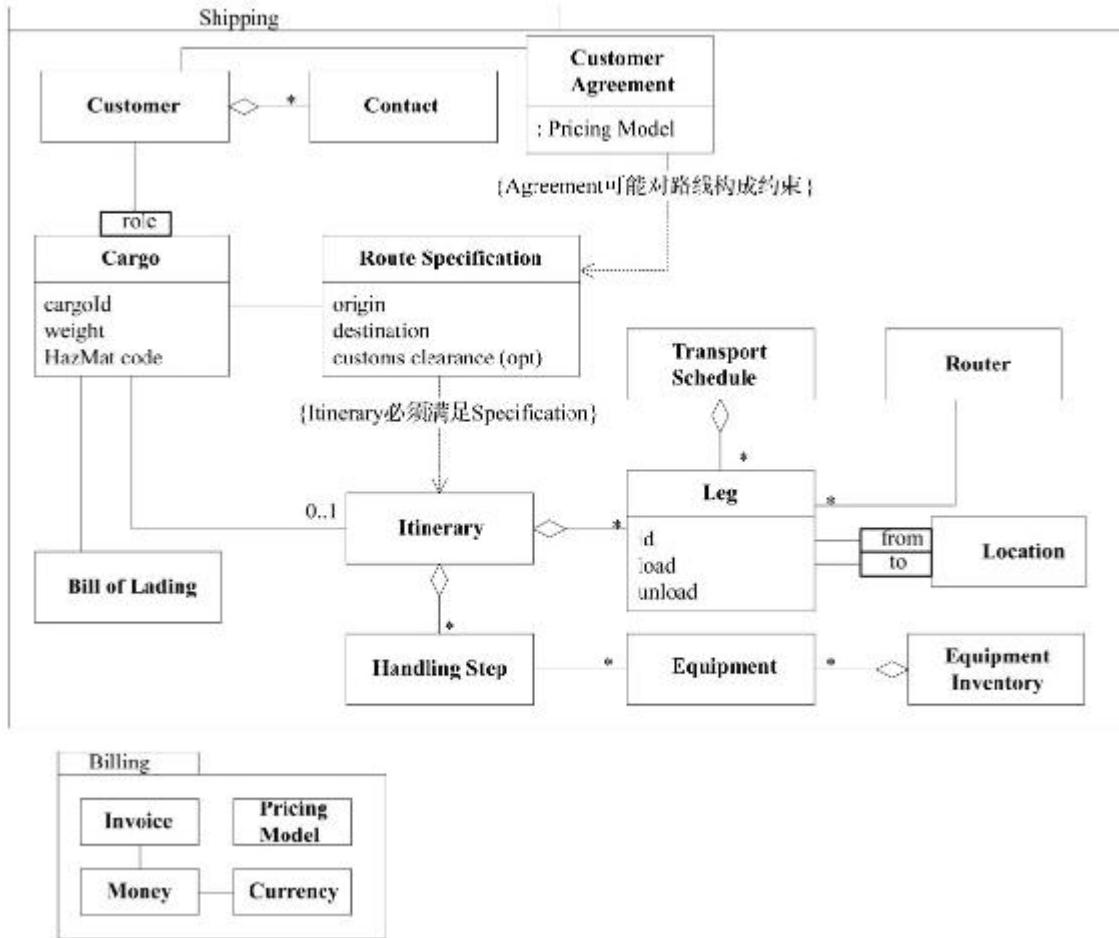


圖15-2

注意，與實際應用程序所需的模型相比，這個模型是高度簡化的。真實的模型過於複雜，不適合作為例子。因此，儘管這個示例的複雜程度可能不足以驅使我們創建SEGREGATED CORE，但可以把這個模型想像得十分複雜，很難解釋，而且無法作為一個整體來處理。

現在，運輸模型的實質是什麼？通常「梗概」是一個很好的起點。據此，我們可能會注意到Pricing（定價）和Invoice（發票）上[4]。但實際上我們需要看一下DOMAIN VISION STATEMENT。以下就是從願景說明中摘錄的：

.....提高操作的可見性，並提供更快速可靠地滿足客戶需求的工具.....

這個應用程序並不是為銷售部門設計的，而是供公司一線操作人員使用。因此，我們把所有與金錢有關的問題（當然很重要）歸結為支持性作用。已經有人把一些這樣的項放到一個單獨的包（Billing）中。我們可以保留這個包，並進一步確認它起到支持作用。

我們需要把重點放在貨物處理上：根據客戶需求來運輸貨物。我們把與這些活動直接相關的類提取出來放到一個新的包Delivery中，這樣就產生了一個SEGREGATED CORE，如圖15-3所示。

大部分操作都只是把類移動到新的包中，但模型本身也有幾處改動。

首先，Customer Agreement對Handling Step進行了約束。這是團隊在分離CORE過程中獲得的典型理解。由於團隊把注意力放在有效、正確的運輸上，顯然Customer Agreement中的運輸約束是非常重要的，而且應該在模型中顯式地表達出來。

另一項更改更有實效。在重構之後的模型中，Customer Agreement直接連接到Cargo，而不再需要通過Customer進行導航（在預訂Cargo時，Customer Agreement必須像Customer一樣連接到Cargo）。在實際運輸時，Customer與運輸作業的關係不如Agreement與作業的關係緊密。而在原來的模型中，必須根據Customer在運輸中的角色找到正確的Customer，然後再查詢其Customer Agreement。這種交互使得模型的表述不易理解。新的關聯使那些最重要的場景變得盡可能簡單和直接。現在就很容易把Customer完全從CORE中分離出去了。

那麼到底是否應該把Customer提取出來呢？我們的關注點是要滿足Customer的需求，因此最初看上去Customer應該屬於CORE。然而，由於運輸期間的交互現在可以直接訪問Customer Agreement了，

因此就不再需要Customer類。這樣Customer的基本模型就非常通用了。

Leg是否應該保留在CORE中這個問題可能會引起很大的爭議。我的意見是CORE應保持最小化，而且Leg與Transport Schedule、Routing Service和Location具有更緊密的聯繫，而這三者都不需要在CORE中。但是，如果這個模型描述的很多場景都涉及Leg，那麼我就會把它移到Delivery包中，即使把它與上面那些類分開顯得有些不協調。

在這個例子中，所有類定義都與先前相同，但精煉通常都需要對類進行重構，以便分離出通用職責和領域專有職責，然後就可以把核心分離出來了。

既然我們已經有了一個SEGREGATED CORE，重構就完成了。但剩下的Shipping包正是「把CORE提取出來後剩下的所有東西」。我們可以再進行其他的重構過程，以便得到更清晰的打包方式，如圖15-4所示。

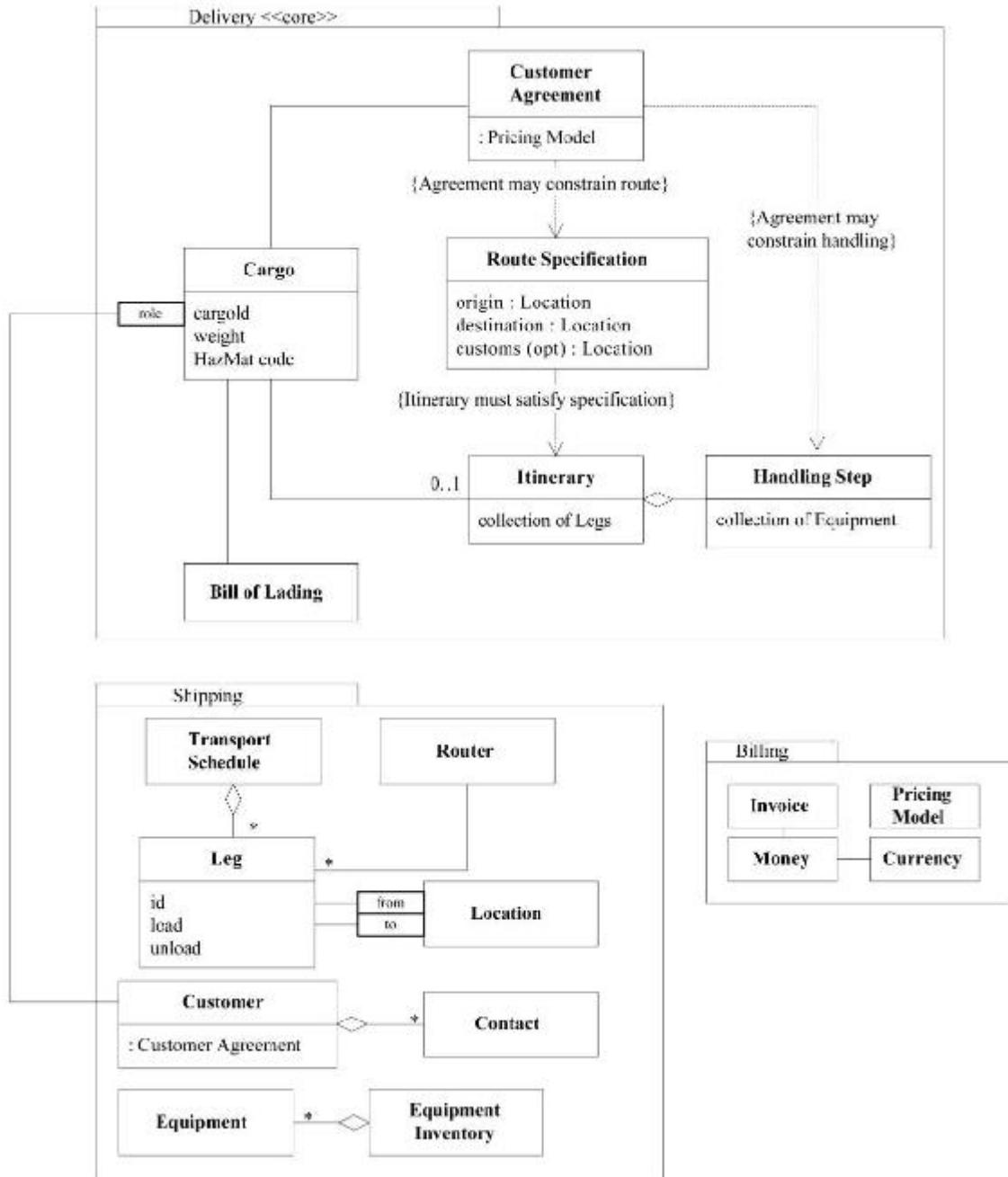


圖15-3 按照客戶需求可靠地運輸貨物是這個項目的核心目標

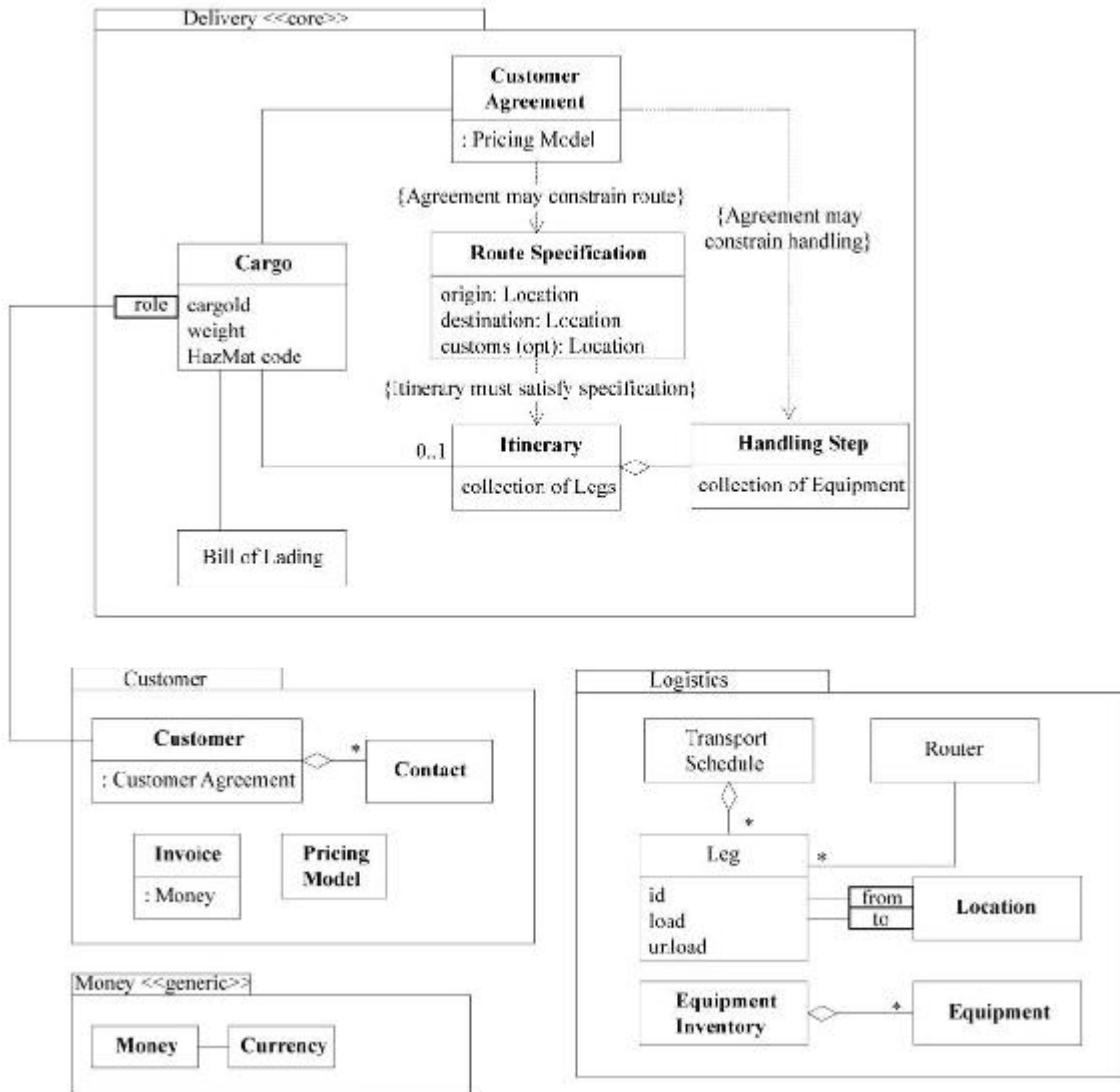
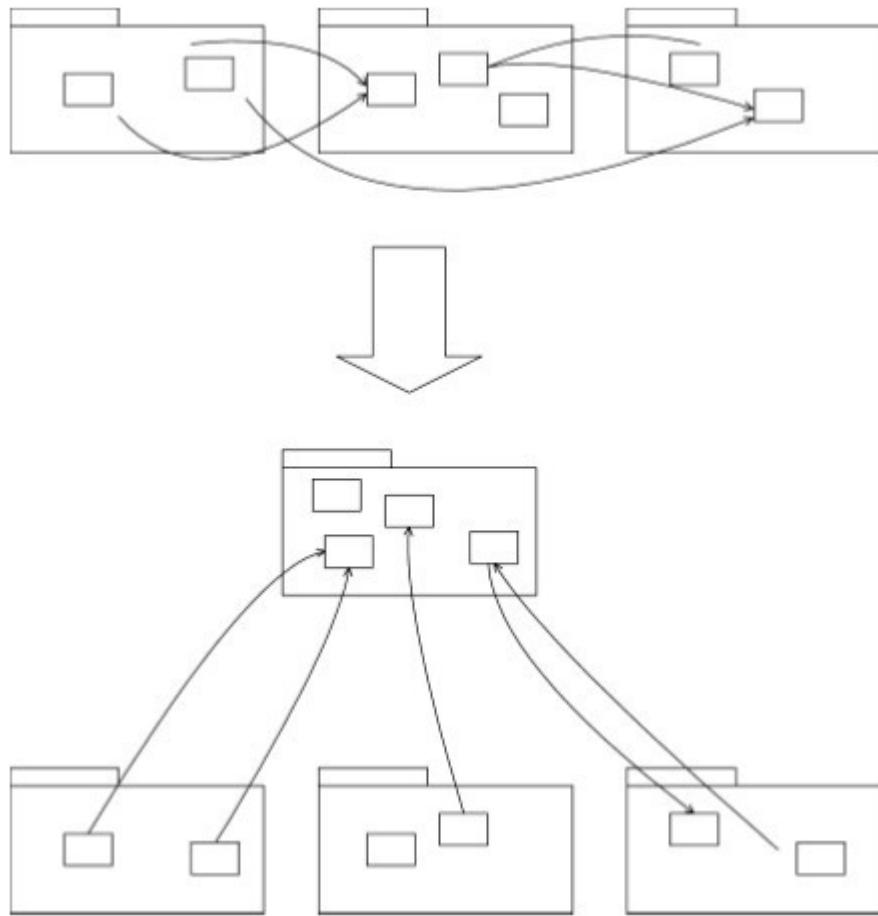


圖15-4 完成SEGREGATED CORE之後留下的有意義的非CORE子領域MODULE

這種效果不是一次就能實現的，可能需要經過多次重構。於是，我們最後得到了一個SEGREGATED CORE包、一個GENERIC SUBDOMAIN和兩個起支持作用的領域專用包。在有了更深層的理解後，可能會為Customer創建一個GENERIC SUBDOMAIN，或者將Customer專用於運輸。

識別有用的、有意義的MODULE是一項建模活動（正如第5章中所討論的那樣）。開發人員和領域專家在戰略精煉中進行協作，這種協作是知識消化過程的一部分。

## 15.9 模式：ABSTRACT CORE



通常，即便是CORE DOMAIN模型也會包含太多的細節，以至於它很難表達出整體視圖。

我們處理大模型的方法通常是把它分解為足夠小的子領域，以便能夠掌握它們並把它們放到一些獨立的MODULE中。這種簡化式的打包風格通常是行之有效的，能夠使一個複雜的模型變得易於管理。但有時創建獨立的MODULE反而會使子領域之間的交互變得晦澀難懂，甚至變得更複雜。

當不同**MODULE**的子領域之間有大量交互時，要麼需要在**MODULE**之間創建很多引用，這在很大程度上抵消了劃分模塊的價

值；要麼就必須間接地實現這些交互，而後者會使模型變得晦澀難懂。

我們不妨考慮採用橫向切割而不是縱向切割的方式。多態性 ( polymorphism ) 允許我們忽略抽象類型實例的很多細節變化。如果 **MODULE** 之間的大部分交互都可以在多態接口這個層次上表達出來，那麼就可以把這些類型重構到一個特定的 **CORE MODULE** 中。

這裡並不是尋找技術上的技巧。只有當領域中的基本概念能夠用多態接口來表達時，這才是一種有價值的技術。在這種情況下，把這些分散注意力的細節分離出來可以使 **MODULE** 解耦，同時可以精煉出一個更小、更內聚的 **CORE DOMAIN**。

因此：

把模型中最基本的概念識別出來，並分離到不同的類、抽象類或接口中。設計這個抽象模型，使之能夠表達出重要組件之間的大部分交互。把這個完整的抽象模型放到它自己的 **MODULE** 中，而專用的、詳細的實現類則留在由子領域定義的 **MODULE** 中。

現在，大部分專用的類都將引用 **ABSTRACT CORE MODULE**，而不是其他專用的 **MODULE**。**ABSTRACT CORE** ( 抽象核心 ) 提供了主要概念及其交互的簡化視圖。

提取 **ABSTRACT CORE** 並不是一個機械的過程。例如，如果把 **MODULE** 之間頻繁引用的所有類都自動移動到一個單獨的 **MODULE** 中，那麼結果可能是一團糟，而且毫無意義。對 **ABSTRACT CORE** 進行建模需要深入理解關鍵概念以及它們在系統的主要交互中扮演的角色。換言之，它是通過重構得到更深層理解的。而且它通常需要大量的重新設計。

如果項目中同時使用了 **ABSTRACT CORE** 和精煉文檔，而且精煉文檔隨著應用程序理解的加深而不斷演變，那麼抽象核心的最後結果

看起來應該與精煉文檔非常類似。當然，ABSTRACT CORE是用代碼編寫的，因此更為嚴格和完整。

## 15.10 深層模型精煉

精煉並不僅限於從整體上把領域中的一些部分從CORE中分離出來。它也意味著對子領域（特別是CORE DOMAIN）進行精煉，通過持續重構得到更深層的理解，從而向深層模型和柔性設計推進。精煉的目標是把模型設計得更明顯，使我們可以用模型簡單地把領域表示出來。深層模型把領域中最本質的方面精煉成一些簡單的元素，使我們可以把這些元素組合起來解決應用程序中的重要問題。

儘管任何帶來深層模型的突破都有價值，但只有**CORE DOMAIN**中的突破才能改變整個項目的軌道。

## 15.11 選擇重構目標

當你遇到一個雜亂無章的大型系統時，應該從哪裡入手呢？在XP社區中，答案往往は以下之一：

- (1) 可以從任何地方開始，因為所有的東西都要進行重構；
- (2) 從影響你工作的那部分開始——也就是完成具體任務所需要的那個部分。

這兩種做法我都不贊成。第一種做法並不十分可行，只有少數完全由頂尖的程序員組成的團隊才是例外。第二種做法往往只是對外圍問題進行了處理，只治其標而不治其本，迴避了最嚴重的問題。最終這會使代碼變得越來越難以重構。

因此，如果你既不能全面解決問題，又不能「哪兒痛治哪兒」，那麼該怎麼辦呢？

(1) 如果採用「哪兒痛治哪兒」這種重構策略，要觀察一下根源問題是否涉及CORE DOMAIN或CORE與支持元素的關係。如果確實涉及，那麼就要接受挑戰，首先修復核心。

(2) 當可以自由選擇重構的部分時，應首先集中精力把CORE DOMAIN更好地提取出來，完善對CORE的分離，並且把支持性的子領域提煉成通用子領域。

以上就是如何從重構中獲取最大利益的方法。

## 第16章 大型結構

硅谷一家小設計公司簽了一份為衛星通信系統創建模擬器的合同。工作進展得很順利，他們正在開發一個 MODEL-DRIVEN DESIGN，這個設計能夠表示和模擬各種網絡條件和故障。



數千人分工合作來製作「艾滋病紀念拼被」( AIDS Quilt )  
但開發團隊的領導者卻有點不安。問題本身太複雜了。為了澄清  
模型中的複雜關係，他們已經把設計分解為一些在規模上便於管理的  
內聚MODULE，於是現在便有了的很多MODULE。在這種情況下，開  
發人員要想查找某個功能，應該到哪個MODULE中去查呢？如果有了

一個新類，應該把它放在哪裡？這些小軟件包的實際意義是什麼？它們又是如何協同工作的呢？而且以後還要創建更多的MODULE。

開發人員互相之間仍然能夠進行很好的溝通，而且也知道每天都要做什麼工作，但項目領導者卻不滿足這種一知半解的狀態。他們需要某種組織設計的方式，以便在項目進入到更複雜的階段時能夠理解和掌控它。

他們進行了頭腦風暴活動，發現了很多潛在的辦法。開發人員提出了不同的打包方案。有一些文檔給出了系統的全貌，還有一些使用建模工具繪製的類圖——新視圖可以用來指引開發人員找到正確的模塊。但項目領導者對這些小花招並不滿意。

他們可以用模型把模擬器的工作流程簡單地描述出來，也可以說清楚基礎設施是如何序列化數據的，以及電信技術層怎樣保證數據的完整性和路由選擇。模型中包含了所有細節，卻沒有一條清楚的主線。

領域的一些重要概念丟失了。但這次丟失的不是對像模型中的一兩個類，而是整個模型的結構。

經過一兩周的仔細思考之後，開發人員有了思路。他們打算把設計放到一個結構中。整個模擬器將被看作由一系列層組成，這些層分別對應於通信系統的各個方面。最下面的層用來表示物理基礎設施，它具有將數據位從一個節點傳送到另一個節點的基本能力。它的上面是封包路由層，與數據流定向有關的問題都被集中到這一層中。其他的層則表示其他概念層次的問題。這些層共同描述了系統的大致情況。

他們開始按照新的結構來重構代碼。為了不讓MODULE跨越多個層，必須對它們重新定義。在一些情況下，還需要重構對像職責，以便明確地讓每個對象只屬於一個層。另一方面，藉由應用這些新思路

的實際經驗，概念層本身的定義也得到了精化。層、MODULE和對像一起演變，最後，整個設計都符合了這種分層結構的大體輪廓。

這些層並不是MODULE，也不是任何其他的代碼工件。它們是一種全局性的規則集，用於約束整個設計中的任何MODULE或對像（甚至包括與其他系統的接口）的邊界和關係。

實施了這種分層級別之後，設計重新變得易於理解了。人們基本上知道到哪裡去尋找某個特定功能。分工不同的開發人員所做的設計決策可以大體上互相保持一致。這樣就可以處理更加複雜的設計了。

即使將MODULE分解，一個大模型的複雜性也可能會使它變得很難掌握。MODULE確實把設計分解為更易管理的小部分，但MODULE的數量可能會很多。此外，模塊化並不一定能夠保證設計的一致性。對象與對像之間，包與包之間，可能應用了一堆的設計決策，每個決策看起來都合情合理，但總的來看卻非常怪異。

嚴格劃分BOUNDED CONTEXT可能會防止出現破壞和混淆，但其本身對於從整體上審視系統並無任何助益。

精煉可以幫助我們把注意力集中於CORE DOMAIN，並將子領域分離出來，讓它們承擔支持性的職責。但我們仍然需要理解這些支持性元素，以及它們與CORE DOMAIN的關係，還有它們互相之間的關係。理想的情況是，整個CORE DOMAIN非常清楚和易於理解，因此不再需要額外的指導，但我們並不總能處於這樣好的境況中。

無論項目的規模如何，人們總需要有各自的分工，來負責系統的不同部分。如果沒有任何協調機制或規則，那麼相同問題的各種不同風格和截然不同的解決方案就會混雜在一起，使人們很難理解各個部分是如何組織在一起的，也不可能看到整個系統的統一視圖。從設計的一個部分學到的東西並不適用於這個設計的其他部分，因此項目最後的結果是開發人員成為各自MODULE的專家，一旦脫離了他們自己

的小圈子就無法互相幫助。在這種情況下，CONTINUOUS INTEGRATION根本無法實現，而BOUNDED CONTEXT也使項目變得支離破碎。

在一個大的系統中，如果因為缺少一種全局性的原則而使人們無法根據元素在模式（這些模式被應用於整個設計）中的角色來解釋這些元素，那麼開發人員就會陷入「只見樹木，不見森林」的境地。

我們需要理解各個部分在整體中的角色，而不必去深究細節。

「大型結構」是一種語言，人們可以用它來從大局上討論和理解系統。它用一組高級概念或規則（或兩者兼有）來為整個系統的設計建立一種模式。這種組織原則既能指導設計，又能幫助理解設計。另外，它還能夠協調不同人員的工作，因為它提供了共享的整體視圖，讓人們知道各個部分在整體中的角色。

設計一種應用於整個系統的規則（或角色和關係）模式，使人們可以通過它在一定程度上瞭解各個部分在整體中所處的位臘（即使是在不知道各個部分的詳細職責的情況下）。

這種結構可以被限制在一個BOUNDED CONTEXT中，但通常情況下它會跨越多個BOUNDED CONTEXT，並通過提供一種概念組織把項目涉及的所有團隊和子系統緊密結合到一起。好的結構可以幫助人們深入地理解模型，還能夠對精煉起到補充作用。

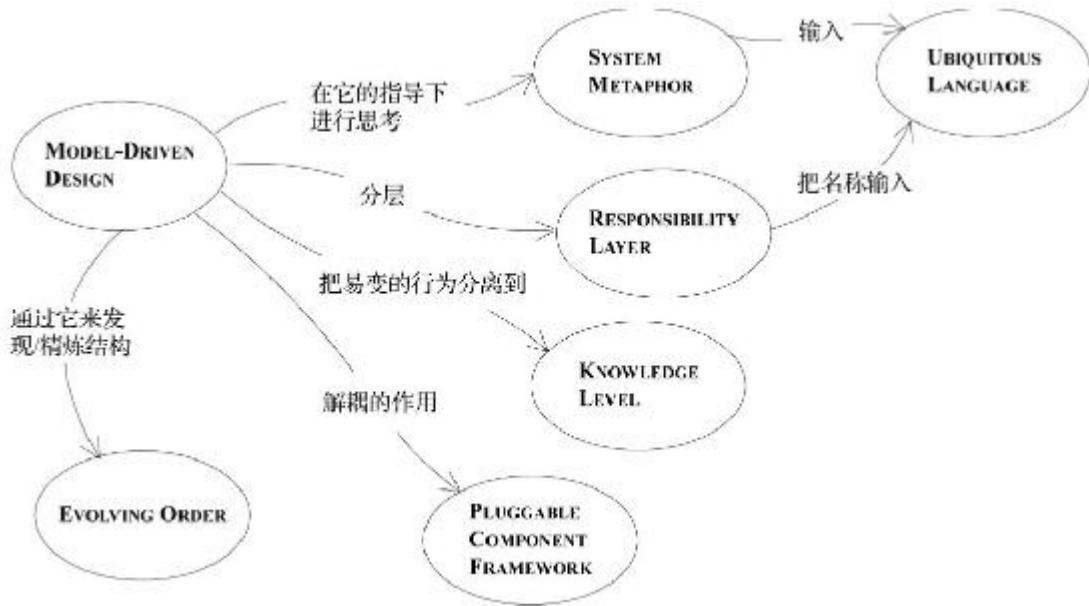


圖16-1 一些大型結構模式

大部分大型結構都無法用UML來表示，而且也不需要這樣做。這些大型結構是用來勾畫和解釋模型及設計的，但在設計中並不出現，它們只是用來表達設計的另外一種方式。在本章的示例中，你將看到許多添加了大型結構信息的非正式的UML圖。

當團隊規模較小而且模型也不太複雜時，只需將模型分解為合理命名的**MODULE**，再進行一定程度的精煉，然後在開發人員之間進行非正式的協調，以上這些就足以使模型保持良好的組織結構了。

大型結構可以節省項目的開發費用，但不適當的結構會嚴重妨礙開發的進展。本章將探討一些能成功構建這種設計結構的模式。

## **16.1 模式：EVOLVING ORDER**

很多開發人員都親身經歷過由於設計結構混亂而產生的代價。為了避免混亂，項目通過架構從各個方面對開發進行約束。一些技術架構確實能夠解決技術問題，如網絡或數據持久化問題，但當我們在應用層和領域模型中使用架構時，它們可能會產生自己的問題。它們往

往會妨礙開發人員創建適合於解決特定問題的設計和模型。一些要求過高的架構甚至會妨礙編程語言本身的使用，導致應用程序開發人員根本無法使用他們在編程語言中最熟悉的和技術能力很強的一些功能。而且，無論架構是面向技術的，還是面向領域的，如果其限定了很多前期設計決策，那麼隨著需求的變更和理解的深入，這些架構會變得束手束腳。

近年來，一些技術架構（如J2EE）已經成為主流技術，而人們對領域層中的大型結構卻沒有做多少研究，這是因為應用程序不同，其各自的需求也大為不同。

在項目前期使用大型結構可能需要很大的成本。隨著開發的進行，我們肯定會發現更適當的結構，甚至會發現先前使用的結構妨礙了我們採取一種使應用程序更清晰和簡化的路線。這種結構的一部分是有用的，但卻使你失去了其他很多機會。你的工作會慢下來，因為你要尋找解決的辦法或試著與架構師們進行協商。但經理會認為架構已經定下來了，當初選這個架構就是因為它能夠使應用程序變得簡單一些，那為什麼不去開發應用程序，卻在這些架構問題上糾纏不清呢？即使經理和架構團隊能夠接受這些問題，但如果每次修改都像是一場攻堅戰，那麼人們很快就會疲乏不堪。

一個沒有任何規則的隨意設計會產生一些無法理解整體含義且很難維護的系統。但架構中早期的設計假設又會使項目變得束手束腳，而且會極大地限制應用程序中某些特定部分的開發人員/設計人員的能力。很快，開發人員就會為適應結構而不得不在應用程序的開發上委曲求全，要麼就是完全推翻架構而又回到沒有協調的開發老路上來。

問題並不在於指導規則本身應不應該存在，而在於這些規則的嚴格性和來源。如果這些用於控制設計的規則確實符合開發環境，那麼

它們不但不會阻礙開發，而且還會推動開發在健康的方向上前進，並且保持開發的一致性。

因此：

讓這種概念上的大型結構隨著應用程序一起演變，甚至可以變成一種完全不同的結構風格。不要依此過分限制詳細的設計和模型決策，這些決策和模型決策必須在掌握了詳細知識之後才能確定。

有時個別部分具有一些很自然且有用的組織和表示方式，但這些方式並不適用於整體，因此施加全局規則會使這些部分的設計不夠理想。在選擇大型結構時，應該側重於整體模型的管理，而不是優化個別部分的結構。因此，在「結構統一」和「用最自然的方式表示個別組件」之間需要做出一些折中選擇。根據實際經驗和領域知識來選擇結構，並避免採用限制過多的結構，如此可以降低折中的難度。真正適合領域和需求的結構能夠使細節的建模和設計變得更容易，因為它快速排除了很多選項。

大型結構還能夠為我們做設計決策提供捷徑，雖然原則上也可以通過研究各個對象來做出這些決策，但實際上這會耗費太長時間，而且產生的結果可能不一致。當然，持續重構仍然是必要的，但這種結構可以幫助重構變得更易於管理，並使不同的人能夠得到一致的解決方案。

大型結構通常需要跨越**BOUNDED CONTEXT**來使用。在經歷了實際項目上的迭代之後，結構將失去與特定模型緊密聯繫的特性，也會得到符合領域的**CONCEPTUAL CONTOUR**的特性。這並不意味著它不能對模型做出任何假設，而是說它不會把專門針對局部情況而做的假設強加於整個項目。它應該為那些在不同**CONTEXT**中工作的開發團隊保留一定的自由，允許他們為了滿足局部需要而修改模型。

此外，大型結構必須適應開發工作中的實際約束。例如，設計人員可能無法控制系統的某些部分的模型，特別是外部子系統或遺留子系統。這個問題有多種解決方式，如修改結構使之更適應特定外部元素，或者指定應用程序與外部元素的關聯方式，或者使結構變得足夠鬆散，以靈活應對難以處理的現實情況。

與CONTEXT MAP不同的是，大型結構是可選的。當使用某種結構可以節省成本並帶來益處時，並且發現了一種適當的結構，就應該使用它。實際上，如果一個系統簡單到把它分解為MODULE就足以理解它，那麼就不必使用這種結構了。當發現一種大型結構可以明顯使系統變得更清晰，而又沒有對模型開發施加一些不自然的約束時，就應該採用這種結構。使用不合適的結構還不如不使用它，因此最好不要為了追求設計的完整性而勉強去使用一種結構，而應該找到盡可能精簡的方式解決所出現問題。要記住寧缺勿濫的原則。

大型結構可能非常有幫助，但也有少數不適用的情況，這些例外情況應該以某種方式被標記出來，以便讓開發人員知道在沒有特殊註明時可以遵循這種結構。如果不適用的情況開始大量出現，就要修改這種結構了，或者乾脆不用它。

如前所述，要想創建一種既為開發人員保留必要自由度同時又能保證開發工作不會陷入混亂的結構絕非易事。儘管人們已經在軟件系統的技術架構上投入了大量工作，但有關領域層的結構化研究還很少見。一些方法會破壞面向對象的範式，如那些按應用任務或按用例對領域進行分解的方法。整個領域的研究還很貧瘠。我曾經在一些項目上看到過幾個通用的大型結構模式。本章將討論4種模式，其中可能會有一種符合你的需要，或者能夠為你提供一些思路，從而找到一種適合你的項目的結構。

## **16.2 模式：SYSTEM METAPHOR**

隱喻思維在軟件開發（特別是模型）中是很普遍的。但極限編程中的「隱喻」卻具有另外一種含義，它用一種特殊的隱喻方式來使整個系統的開發井然有序。

一棟大樓的防火牆能夠在周圍發生火災時防止火勢從其他建築蔓延到它自身，同樣，軟件「防火牆」可以保護局部網絡免受來自更大的外部網的破壞。這個「防火牆」的隱喻對網絡架構產生了很大影響，並且由此而產生了一整套產品類別。有多種互相競爭的防火牆可供消費者選擇，它們都是獨立開發的，而且人們知道它們在一定程度上可以互換。即使網絡的初學者也很容易掌握這個概念。這種在整個行業和客戶中的共同理解很大一部分上得益於隱喻。

然而這個類比卻並不準確，而且防火牆從功能上來看也是把雙刃劍。防火牆的隱喻引導人們開發出了軟件屏障，但有時它並不能起到充分的防護作用，而且會阻止正當的數據交換，同時也無法防護來自網絡內部的威脅。例如，無線LAN就存在漏洞。防火牆這個形象的隱喻確實很有用，但所有隱喻也都是有弊端的[5]。

**軟件設計往往非常抽象且難於掌握。開發人員和用戶都需要一些切實可行的方式來理解系統，並共享系統的一個整體視圖。**

從某種程度上講，隱喻對人們的思考方式有著深刻地影響，它已經滲透到每個設計中。系統有很多「層」，層與層之間依次疊放起來。系統還有「內核」，位於這些層的「中心」。但有時隱喻可以傳達整個設計的中心主題，並能夠在團隊所有成員中形成共同理解。

在這種情況下，系統實際上就是由這個隱喻塑造的。開發人員所做的設計決策也將與系統隱喻保持一致。這種一致性使其他開發人員

能夠根據同一個隱喻來解釋複雜系統中的多個部分。開發人員和專家在討論時有一個比模型本身更具體的參考點。

**SYSTEM METAPHOR** ( 系統隱喻 ) 是一種鬆散的、易於理解的大型結構，它與對像範式是協調的。由於系統隱喻只是對領域的一種類比，因此不同模型可以用近似的方式來與它關聯，這使得人們能夠在多個 **BOUNDED CONTEXT** 中使用系統隱喻，從而有助於協調各個 **BOUNDED CONTEXT** 之間的工作。

**SYSTEM METAPHOR** 是極限編程的核心實踐之一，因此它已經成為一種非常流行的方法(Beck 2000)。遺憾的是，很少有項目能夠找到真正有用的 **METAPHOR**，而且人們有時還會把一些起反作用的隱喻思想灌輸到領域中。有時使用太強的隱喻反而會有風險，因為它使設計中摻雜了一些與當前問題無關的類比，或者是類比雖然很有吸引力，但它本身並不恰當。

儘管如此，**SYSTEM METAPHOR** 仍然是眾所周知的大型結構，它對一些項目非常有用，而且很好地說明瞭結構的總體概念。

因此：

當系統的一個具體類比正好符合團隊成員對系統的想像，並且能夠引導他們向著一個有用的方向進行思考時，就應該把這個類比用作一種大型結構。圍繞這個隱喻來組織設計，並把它吸收到 **UBIQUITOUS LANGUAGE** 中。**SYSTEM METAPHOR** 應該既能促進系統的交流，又能指導系統的開發。它可以增加系統不同部分之間的一致性，甚至可以跨越不同的 **BOUNDED CONTEXT**。但所有隱喻都不是完全精確的，因此應不斷檢查隱喻是否過度或不恰當，當發現它起到妨礙作用時，要隨時準備放棄它。

「幼稚隱喻」以及我們為什麼不需要它

由於在大多數項目並不會自動出現有用的隱喻，因此XP社區中的一些人開始談論「幼稚隱喻」（Naive Metaphor），他們所說的幼稚隱喻就是領域模型本身。

這個術語的一個問題在於，一個成熟的領域模型絕對不會是「幼稚的」。實際上，「工資處理就像一條裝配線」這個隱喻與模型的實際情況相比要幼稚得多，因為模型是軟件開發人員與領域專家進行了多次知識消化的迭代過程才得到的，它已經緊密結合到應用程序的實現中，並經過了實踐的檢驗。

「幼稚隱喻」這個術語應該停止使用了。

SYSTEM METAPHOR並不適用於所有項目。從總體上講，大型結構並不是必須要用的。在極限編程的12個實踐中，SYSTEM METAPHOR的角色可以由UBIQUITOUS LANGUAGE來承擔。當項目中發現一種非常合適的SYSTEM METAPHOR或其他大型結構時，應該用它來補充UBIQUITOUS LANGUAGE。

### **16.3 模式：RESPONSIBILITY LAYER**

在本書從頭至尾的討論中，單獨的對象被分配了一組相關的、範圍較窄的職責。職責驅動的設計在更大的規模上也適用。

如果每個對象的職責都是人為分配的，將沒有統一的指導原則和一致性，也無法把領域作為一個整體來處理。為了保持大模型的一致，有必要在職責分配上實施一定的結構化控制。

當對領域有了深入的理解後，大的模式會變得清晰起來。一些領域具有自然的層次結構。某些概念和活動處在其他元素形成的一個大背景下，而那些元素會因不同原因且以不同頻率獨立發生變化。如何才能充分利用這種自然結構，使它變得更清晰和有用呢？這種自然的

層次結構使我們很容易想到把領域分層，這是最成功的架構設計模式之一（[Buschmann et al.1996]等）。

所謂的層，就是對系統進行劃分，每個層的元素都知道或能夠使用在它「下面」的那些層的服務，但卻不知道它「上面」的層，而且與它上面的層保持獨立。當我們把**MODULE**的依賴性畫出來時，圖的佈局通常是具有依賴性的**MODULE**出現在它所依賴的模塊上面。按照這種方式，可以將各層的順序梳理出來，最終，低層中的對象在概念上不依賴於高層中的對象。

這種自發的分層方式雖然使跟蹤依賴性變得更容易，而且有時具有一定的直觀意義，但它對模型的理解並沒有多大的幫助，也不能指導建模決策。我們需要一種具有更明確目的的分層方式。

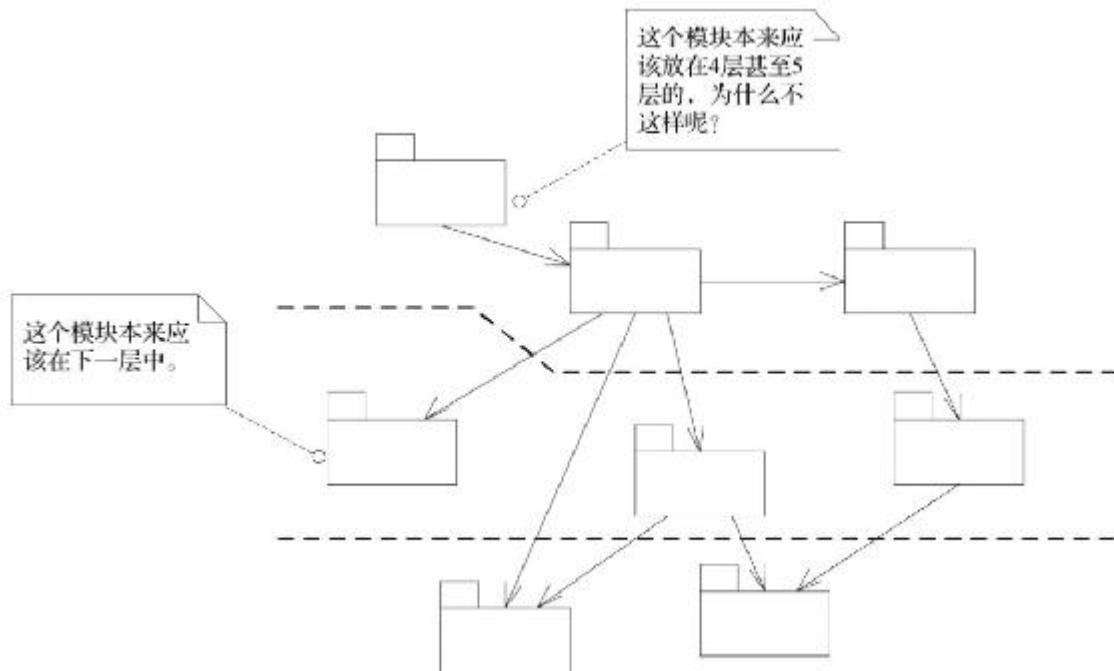


圖16-2 自發的分層，這些包描述了什麼事情

在一個具有自然層次結構的模型中，可以圍繞主要職責進行概念上的分層，這樣可以把分層和職責驅動的設計這兩個強有力的原則結合起來使用。

這些職責必須比分配給單個對象的職責廣泛得多才行，我們稍後就會舉例說明這一點。當設計單獨的**MODULE**和**AGGREGATE**時，要將其限定在其中一個主要職責上。這種明確的職責分組可以提高模塊化系統的可理解性，因為**MODULE**的職責會變得更易於解釋。而高層次的職責與分層的結合為我們提供了一種系統的組織原則。

分層模式有一種變體最適合按職責來分層，我們把這種變體稱為**RELAXED LAYERED SYSTEM**（鬆散分層系統）[Buschmann et al.1996,p.45]。如果採用這種分層模式，某一層中的組件可以訪問任何比它低的層，而不限於只能訪問直接與它相鄰的下一層。

因此：

注意觀察模型中的概念依賴性，以及領域中不同部分的變化頻率和變化的原因。如果在領域中發現了自然的層次結構，就把它們轉換為寬泛的抽象職責。這些職責應該描述系統的高層目的和設計。對模型進行重構，使得每個領域對像、**AGGREGATE**和**MODULE**的職責都清晰地位於一個職責層當中。

這是一段很抽象的描述，但通過幾個示例就可以把它說清楚了。本章開頭的衛星通信模擬器就對職責進行了分層。我曾經在各種領域（如生產控制和財務管理）中看到過使用**RESPONSIBILITY LAYER**（職責層）所產生的良好效果。

下面的示例詳細研究了**RESPONSIBILITY LAYER**，我們可以通過這個例子來體會一下如何去發現任何一種大型結構，以及它是如何指導和約束建模與設計的。、

**示例 深入研究運輸系統的分層**

讓我們看一下把RESPONSIBILITY LAYER應用於前面幾章所討論的貨運應用程序會有什麼效果。

當我們現在又回到這個應用程序時，開發團隊已經有了很大的進展，他們已經創建了一個MODEL-DRIVEN DESIGN，並且提煉出了一個CORE DOMAIN。但隨著設計變得充實，他們在如何把所有部分協調為一個整體上遇到了麻煩。他們正在尋找一種能夠顯示出整個系統主題並且讓每個人都達成一致看法的大型結構。

我們來看一下這個模型中有代表性的一個部分，如圖16-3和圖16-4所示。

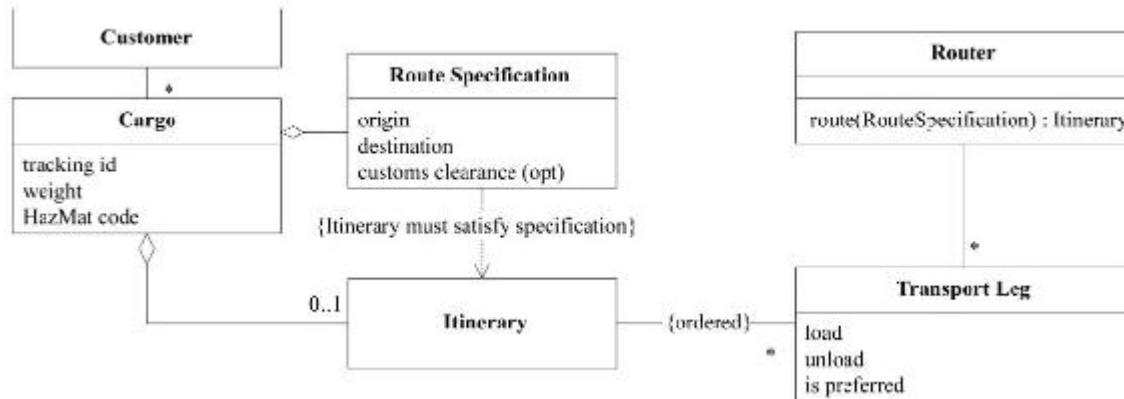


圖16-3 貨運路線的一個基本的運輸領域模型

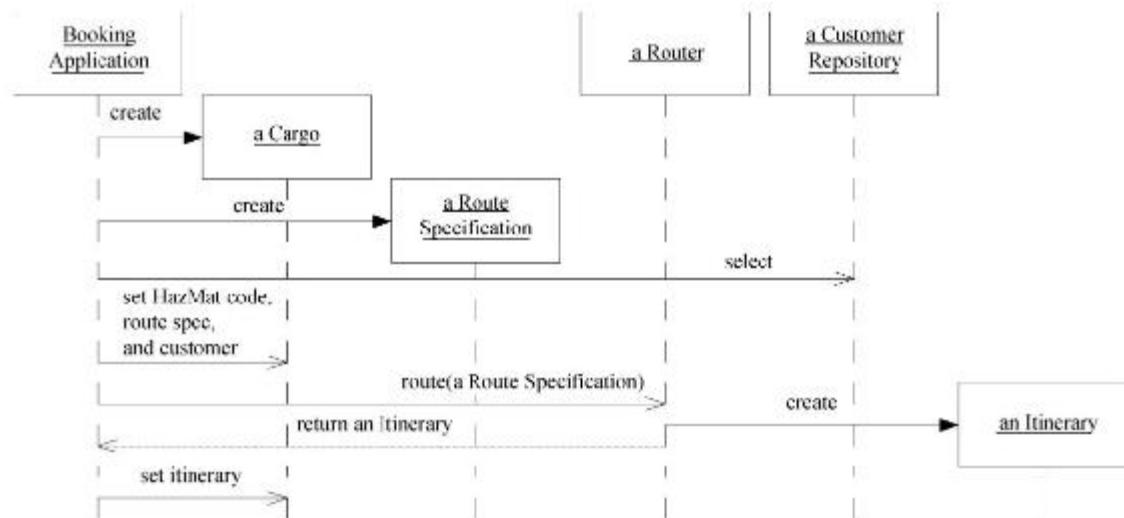


圖16-4 在預訂期間使用模型來制定一個貨運路線

團隊成員研究運輸領域已經有好幾個月了，並且已經觀察到了一些自然的概念層次結構。他們發現在討論運輸時間表（安排好的貨輪航次或火車班次）時不需要涉及所運輸的貨物。而當討論對一個貨物的跟蹤時，如果不知道它的運輸信息，那麼就很難進行跟蹤。概念依賴性是非常清楚的。團隊很容易就區分出兩個層：「作業」層和這些作業的基礎層（他們把這個層叫做「能力」層）。

### 「作業」職責

公司的活動，無論是過去、現在還是計劃的活動，都被組織到「作業」層中。最明顯的作業對象是Cargo，它是公司大部分日常活動的焦點。Route Specification是Cargo的一個不可缺少的部分，它規定了運輸需求。Itinerary是運輸計劃。這些對象都是Cargo聚合的一部分，它們的生命週期與一次進行中的運輸活動緊密地聯繫在一起。

### 「能力」職責

這個層反映了公司在執行作業時所能利用的資源。Transit Leg就是一個典型的例子。人們為貨輪制定航程時間表，貨輪具有一定的貨運能力，這個能力有可能被完全利用，也有可能未被完全利用。

當然，如果公司的主要業務是經營一個運輸船隊的話，那麼Transit Leg將是作業層中的一員。但這個系統的用戶並不需要關心這個問題（如果公司同時從事經營船隊和經營貨運這兩種業務，並且希望協調它們，那麼開發團隊可能需要考慮不同的分層方案，或許要吧作業層分成兩個不同的層，如「運輸作業」和「貨物作業」。）

一個稍微複雜一點兒的決策是把Customer放在哪裡。在一些企業中，客戶只是一些臨時對象。例如，在郵遞公司中，只有在投遞包裹的時候，才知道客戶對象，投遞完成之後，大部分客戶就被忘記了，直到出現下一次投遞。這種性質決定了在針對個人客戶的包裹投遞服務中，客戶僅僅與作業相關。但在我們假想的這家運輸公司中，

需要與客戶保持長期關係，而且大部分業務都來自回頭客。考慮到企業用戶的這些意圖，Customer應該屬於「能力」層。正如我們看到的，這並非一個技術決策，而是試圖掌握並交流領域知識。

由於Cargo與Customer之間的關聯可以限定在一個遍歷方向，因此Cargo REPOSITORY需要通過一個查詢來查找某個特定Customer的所有Cargo。不管怎樣，按照這種方式來設計都有很好的理由，但在使用了大型結構以後，現在它變成一項必須要滿足的需求了。

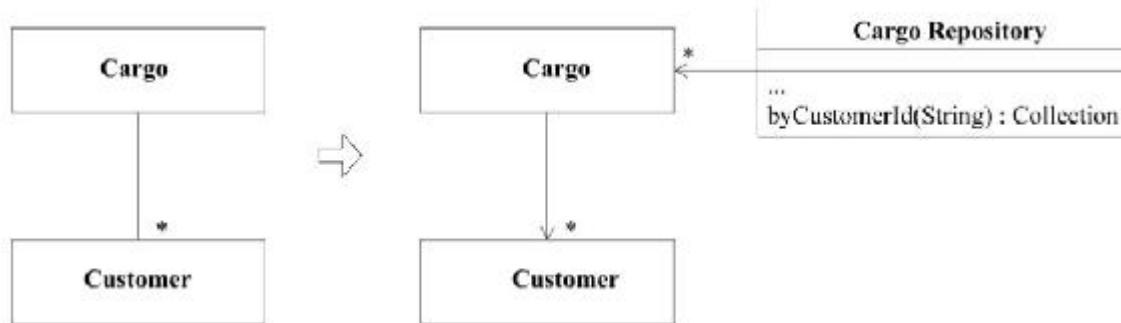


圖16-5 由於雙向關聯會破壞分層，因此用查詢來代替它

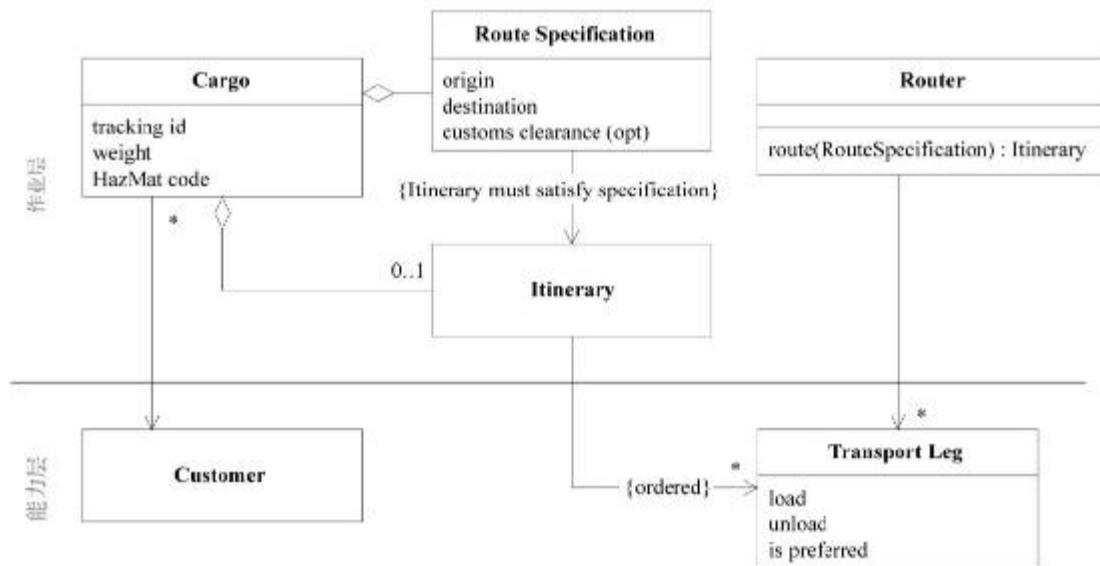


圖16-6 初次對模型進行分層

雖然作業層與能力層的區別使這張圖看上去很清楚了，但次序仍需要進一步細化。經過幾個星期的實驗之後，團隊將注意力集中在另

一個特性上。在很大程度上，最初的兩個層主要考慮的是當前的情況或計劃。但Router（以及其他很多未在圖中畫出的元素）並不是當前的作業或計劃的一部分。它是用來幫助修改這些計劃的。因此團隊定義了一個新的層，讓它來負責決策支持（Decision Support）。

### 「決策支持」職責層

軟件的這個層為用戶提供了用於制定計劃和決策的工具，它具有自動制定一些決策的潛能（例如，當運輸時間表發生變動時，自動重新制定運送Cargo的路線）。

Router是一個SERVICE，能幫助預訂代理（booking agent）選擇運送貨物的最佳路線。因此Router明顯屬於決策支持層。

現在模型中的元素基本上都按照這3個層來組織了，唯一例外的是Transport Leg的「is preferred」屬性。這個屬性存在的原因是公司希望在可能的情況下優先使用自己的貨輪，或者是那些簽訂了優惠合同的公司的貨輪。is preferred屬性用於使Router優先選擇這些首選的運輸工具。這個屬性與「能力層」毫無關係。它是一個用於指導決策制定的策略。為了使用新的RESPONSIBILITY LAYER，需要對模型進行重構。

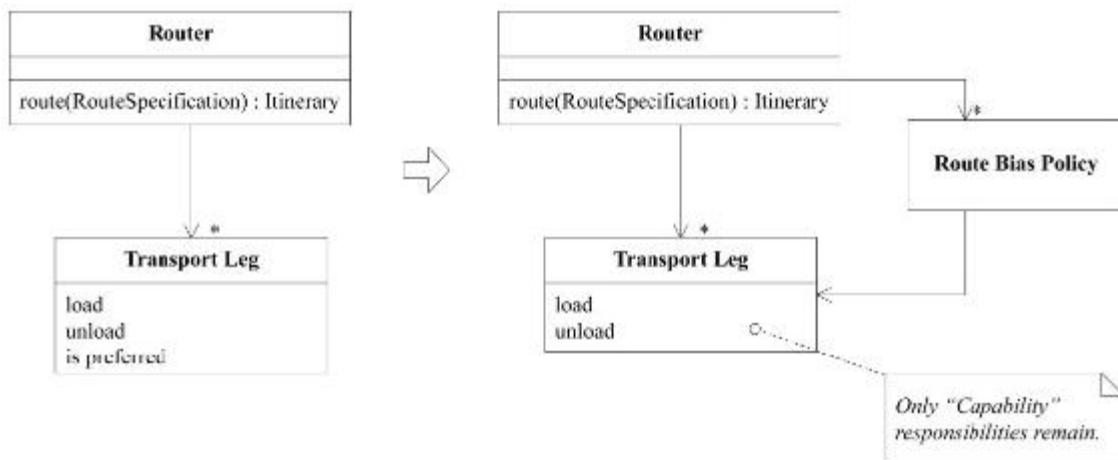


圖16-7 對模型進行重構，使之符合新的分層結構

這次重構使Route Bias Policy變得更清楚，同時使得Transport Leg更專注於運輸能力的基本概念。基於對領域的深刻理解而發現的大比例結構總是能夠使模型更清楚地表達其含義。

現在，這個新模型更加符合大比例結構了。如圖16-8所示。

開發人員在熟悉了選定的分層結構後，很容易區分出各個部分的角色和依賴關係。大比例結構的價值隨著複雜度的增加而增加。

注意，雖然我使用了一個修改後的UML圖來演示這個例子，但這只是為了表示分層而使用的一種方式。UML中並沒有這種表示法，因此這些是作為額外的信息加上去的，目的是讓讀者看得更清楚。如果在你的項目中，代碼就是最終的設計文檔，那麼最好可以使用一種可以按層查看類（或至少按照層來報告與這些類有關的信息）的工具。

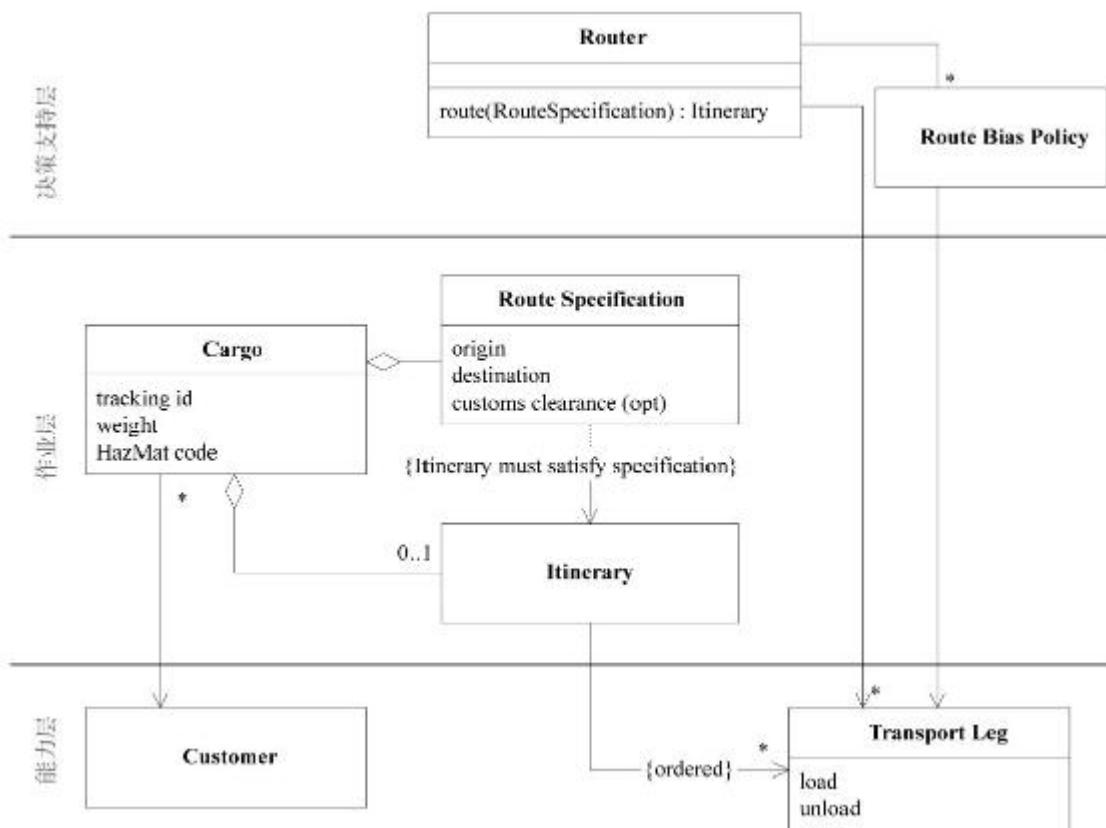


圖16-8 重構後的模型

## 大比例結構如何影響後續設計

一旦採用了一種大比例結構，後續的建模和設計決策就必須要把它考慮在內。為了說明這一點，假設我們必須在這個已分層的設計中增加一個新特性。領域專家們剛剛告訴我們一些針對特定類別危險品的航線約束。有些危險品在某些貨輪或港口上是禁止裝載的。我們必須使Router遵守這些規則。

有很多可行的方法。在未使用大比例結構時，一種吸引人的設計方法是讓擁有Route Specification和Hazardous Material ( HazMat ) 代碼的對象負責把這些航線規則加進來，這個對象就是Cargo。

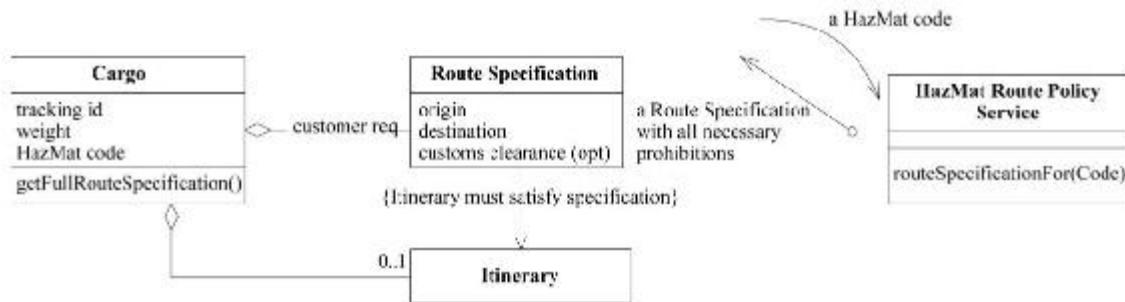


圖16-9 用於制定危險貨物運送路線的一種可能的設計

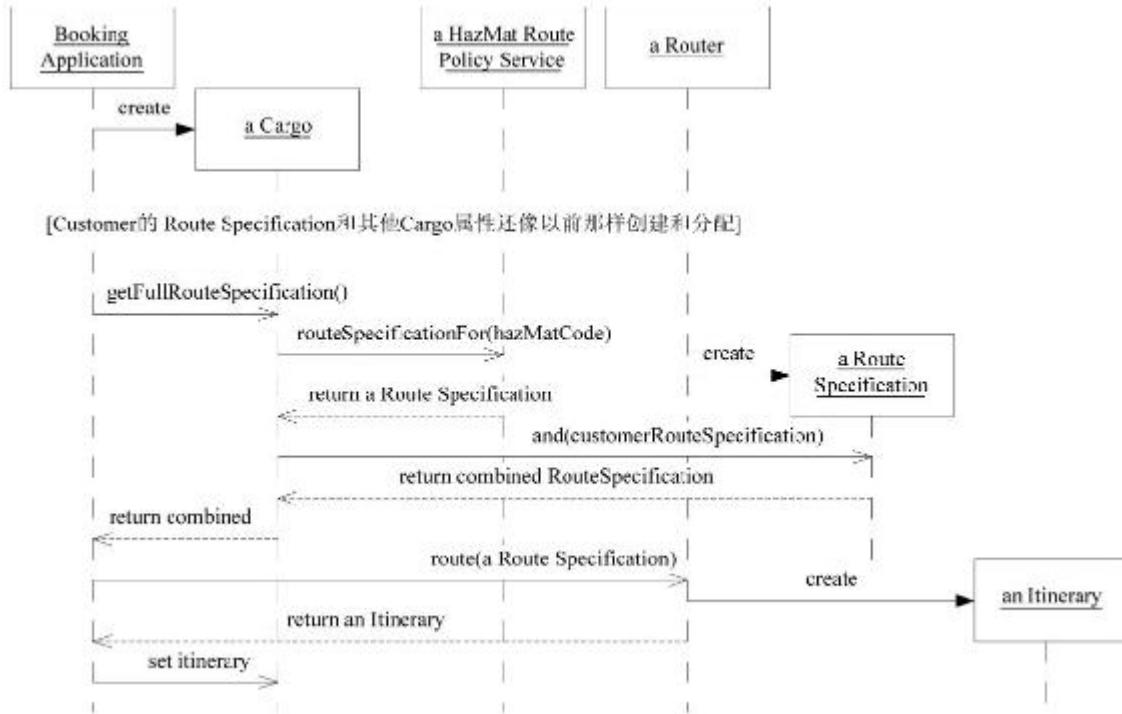


圖16-10

問題是這種設計並不適合大比例結構。HazMat Route Policy Service並沒有問題，它非常適合承擔決策支持層的職責。問題在於Cargo（一個作業層對像）對HazMat Route Policy Service（一個決策支持層對像）的依賴上。只要項目還採用目前的分層，就不能使用這個模型，因為開發人員會認為設計將遵循分層結構，而這種依賴會使開發人員感到糊塗。

可能的設計選擇總會有很多，這裡我們只選擇另外一種設計，這種設計符合大比例結構的規則。HazMat Route Policy服務本身是完全沒有問題的，但我們需要把使用它的職責轉移到別處。讓我們嘗試讓Router來承擔在搜索航線之前收集相關規則的職責。這意味著要修改Router接口，把規則可能依賴的那些對像包括進來。下面就是一種可能的設計，如圖16-11所示。

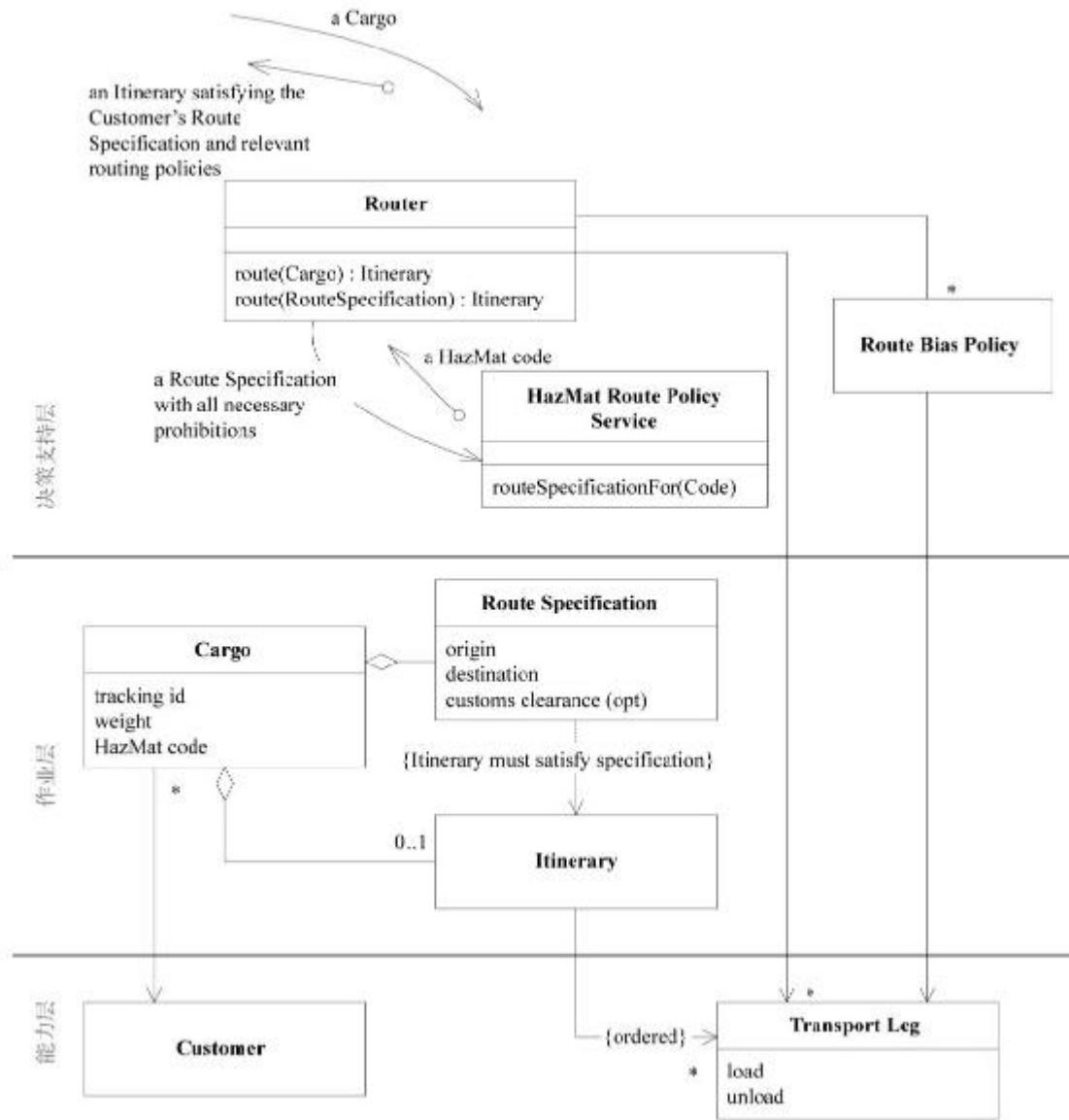


圖16-11 符合分層結構的一種設計

一種典型的交互如圖16-12所示。

現在的這個設計並不一定就比前面那個設計更好。二者都是各有利弊。但如果項目的所有人員都採用一致的方式來制定決策，那麼整體的設計就更容易理解，因此這也值得在細小的設計選擇上做出一些適度的折中。

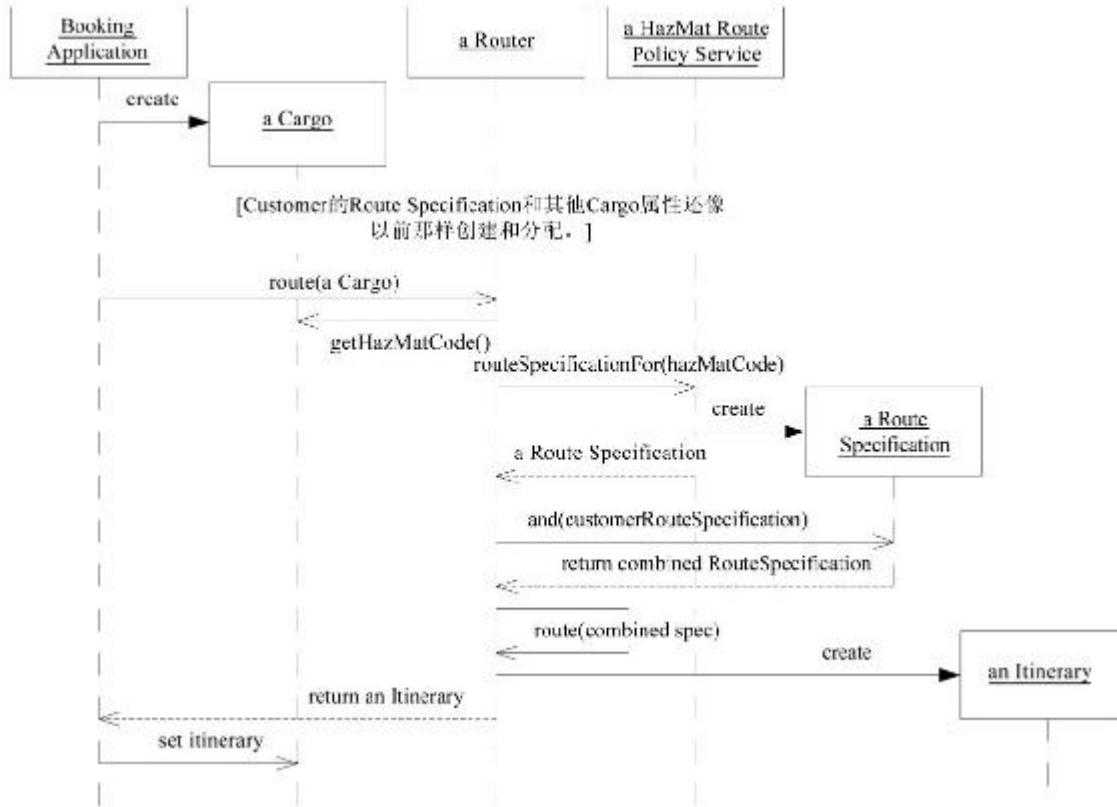


圖16-12

如果所採用的結構強制性地要求我們做出很多彎扭的設計選擇，那麼就要遵循EVOLVING ORDER（演變的順序），在項目進行過程中評估這種結構，並修改甚至放棄它。

### 選擇適當的層

要想找到一種適當的RESPONSIBILITY LAYER或大比例結構，需要理解問題領域並反覆進行實驗。如果遵循EVOLVING ORDER，那麼最初的起點並不是十分重要，儘管差勁的選擇確實會加大工作量。結構可能最後演變得面目全非。因此，下面將給出一些指導方針，無論是剛開始選擇一種結構，還是對已有結構進行轉換，這些指導方針都適用。

當對層進行刪除、合併、拆分和重新定義等操作時，應尋找並保留以下一些有用的特徵。

場景描述。層應該能夠表達出領域的基本現實或優先級。選擇一種大比例結構與其說是一種技術決策，不如說是一種業務建模決策。層應該顯示出業務的優先級。

概念依賴性。「較高」層概念的意義應該依賴「較低」層，而低層概念的意義應該獨立於較高的層。

**CONCEPTUAL CONTOUR**。如果不同層的對象必須具有不同的變化頻率或原因，那麼層應該能夠容許它們之間的變化。

在為每個新模型定義層時不一定總要從頭開始。在一系列相關領域中，有些層是固定的。

例如，在那些利用大型固定資產進行運作的企業（如工廠或貨運）中，物流軟件通常可以被組織為「潛能」層（上面例子中的「能力」層的另外一個名稱）和「作業」層。

潛能層。我們能夠做什麼？潛能層不關心我們打算做什麼，而關心能夠做什麼。企業的資源（包括人力資源）以及這些資源的組織方式是潛能層的核心。與供應商簽訂的合同也明確界定了企業的潛能。這個層幾乎存在於任何業務領域中，但在那些相對來說依靠大型固定資產來支持業務運作的企業中（如運輸和製造業）尤其突出。潛能也包括臨時性的資產，但主要依賴臨時資產來運作的企業可能會強調臨時資產的層（這個層在例子中被稱為「Capability」），這一點稍後會討論。

作業層。我們正在做什麼？我們利用這些潛能做了什麼事情？像潛能層一樣，這個層也應該反映出現實狀況，而不是我們設想的狀況。我們希望在這個層中看到自己的工作和活動：我們正在銷售什麼，而不是能夠銷售什麼。通常來說，作業層對象可以引用潛能層對象，它甚至可以由潛能層對像組成，但潛能層對像不應該引用作業層對象。

在這類領域很多（也許是大部分）現有的系統中，這兩個層可以涵蓋一切對像（儘管可能會有某種完全不同的和更清晰的分解結構）。它們可以跟蹤當前狀況和正在執行的作業計劃，以及問題報告或相關文檔。但跟蹤往往是不夠的。當項目要為用戶提供指導或幫助或者要自動制定一些決策時，就需要有另外一組職責，這些職責可以被組織到作業層之上的決策支持層中。

決策支持層。應該採取什麼行動或制定什麼策略？這個層是用來作出分析和制定決策的。它根據來自較低層（如潛能層或作業層）的信息進行分析。決策支持軟件可以利用歷史信息來主動尋找適用於當前和未來作業的機會。

決策支持系統對其他層（如作業層或潛能層）有概念上的依賴性，因為決策並不是憑空制定的。很多項目都利用數據倉庫技術來實現決策支持。在這樣的項目中，決策支持層實際上變成了一個獨特的 **BOUNDED CONTEXT**，並且與作業軟件具有一種 **CUSTOMER/SUPPLIER** 關係。在其他項目中，決策支持層被更深地集成到系統中，就像前面的擴展示例講到的那樣。分層結構的一個內在的優點是較低的層可以獨立於較高的層存在。這樣有利於在較老的作業系統上分階段引入新功能或開發高層次的增強功能。

另一種情形是軟件實施了詳細的業務規則或法律需求，這些規則或需求可以形成一個 **RESPONSIBILITY LAYER**。

策略層。規則和目標是什麼？規則和目標主要是被動的，但它們約束著其他層的行為。這些交互的設計是一個微妙的問題。有時策略會作為一個參數傳給較低層的方法。有時會使用 **STRATEGY** 模式。策略層與決策支持層能夠進行很好的協作，決策支持層提供了用於搜索策略層所設定的目標的方式，這些目標又受到策略層所設定的規則的約束。

策略層可以和其他層使用同一種語言來編寫，但它們有時是使用規則引擎來實現的。這並不是說一定要把它們放到一個單獨的 BOUNDED CONTEXT 中。實際上，通過在兩種不同的實現技術中嚴格使用同一個模型，可以減小在這兩種實現技術之間進行協調的難度。當規則與它們所應用的對象是基於不同模型編寫的時候，要麼複雜度會大大增加，要麼對像會變得十分笨拙而難以管理。如圖 16-13 所示。



圖 16-13 工廠自動化系統中的概念依賴性和切合點

很多企業並不是依靠工廠和設備能力來運營的。舉兩個例子，在金融服務或保險業中，潛能在很大程度上是由當前的運營狀況決定的。一家保險公司在考慮簽保單承擔理賠責任時，要根據當前業務的多樣性來判斷是否有能力承擔它所帶來的風險。潛能層有可能會被合併到作業層中，這樣就會演變出一種不同的分層結構。

這些情況下經常出現的一個層是對客戶所做出的承諾（見圖 16-14）。

承諾層。我們承諾了什麼？這個層具有策略層的性質，因為它表述了一些指導未來運營的目標；但它也有作業層的性質，因為承諾是作為後續業務活動的一部分而出現和變化的。

潛能層和承諾層並不是互相排斥的。在有的領域中（如一家提供很多定製運輸服務的運輸公司），這兩個層都很重要，因此可以同時使用它們。與這些領域密切相關的其他層也會用到。我們需要對分層結構進行調整和實驗，但一定要使分層系統保持簡單，如果層數超過4或5，就比較難處理了。層數過多將無法有效地描述領域，而且本來要使用大比例結構解決的複雜性問題又會以一種新的方式出現。我們必須對大比例結構進行嚴格的精簡。

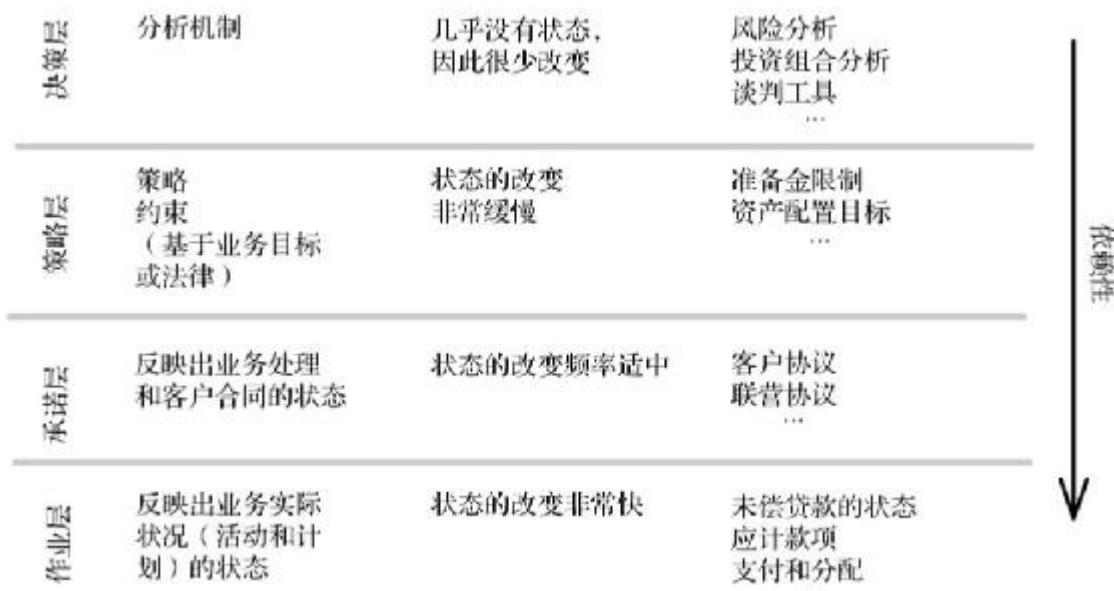
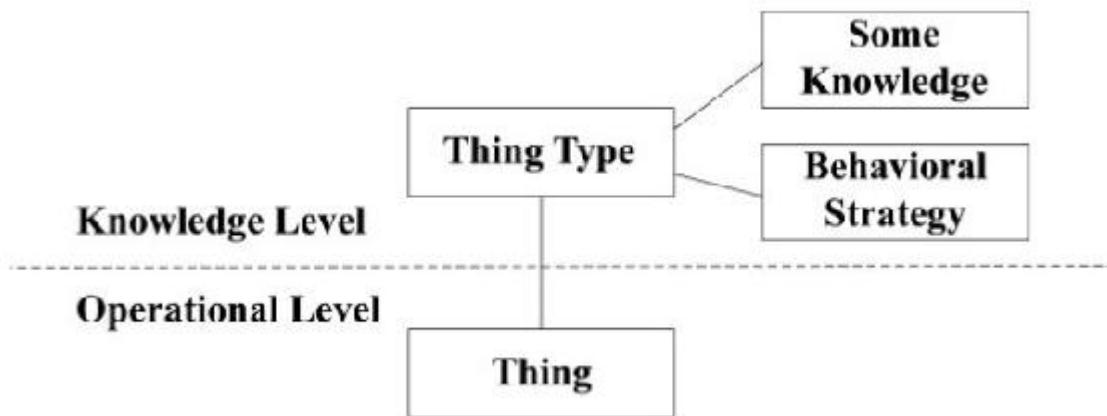


圖16-14 投資銀行系統中的概念依賴性和切合點

雖然這5個層對很多企業系統都適用，但並不是所有領域的主要概念都涵蓋在這5個層中。有些情況下，在設計中生硬地套用這種形式反而會起反作用，而使用一組更自然的**RESPONSIBILITY LAYER**會更有效。如果一個領域與上述討論毫無關係，所有的分層可能都必須

從頭開始。最後，我們必須根據直覺選擇一個起點，然後通過EVOLVING ORDER來改進它。

## 16.4 模式：KNOWLEDGE LEVEL



「KNOWLEDGE LEVEL是」一組描述了另一組對像應該有哪些行為的對象。

[Martin Fowler, 「Accountability」 · [www.martinfowler.com](http://www.martinfowler.com)]

當我們需要讓用戶對模型的一部分有所控制，而模型又必須滿足更大一組規則時，可以利用KNOWLEDGE LEVEL（知識級別）來處理這種情況。它可以使軟件具有可配鎔的行為，其中實體中的角色和關係必須在安裝時（甚至在運行時）進行修改。

在《分析模式》[Fowler 1996,pp.24–27]一書中，知識級別這種模式是討論在組織內部對責任進行建模的時候提到的，後來在會計系統的過賬規則中也用到了這種模式。雖然有幾章內容涉及此模式，但並沒有為它單獨開一章，因為它與書中所討論的大部分模式都不相同。KNOWLEDGE LEVEL並不像其他分析模式那樣對領域進行建模，而是用來構造模型的。

為了使問題更具體，我們來考慮一下「責任」( *accountability* ) 模型。組織是由人和一些更小的組織構成的，並且定義了他們所承擔的角色和互相之間的關係。不同的組織用於控制這些角色和關係的規則大不相同。有的公司分為各個「部門」，每個部門可能由一位「主管」來領導，他要向「副總裁」匯報。而有的公司則分為各個「模塊」( *module* )，每個模塊由一位「經理」來領導，他要向「高級經理」匯報。還有一些組織採用的是「矩陣」形式，其中每個人都出於不同的目的而向不同的經理匯報。

一般的應用程序都會做一些假設。當這些假設並不恰當時，用戶就會在數據錄入字段中輸入與預期不符的數據。由於語義被用戶改變，因此應用程序的任何行為都可能會失敗。用戶將會想出一些迂迴的辦法來執行這些行為，或者關閉一些高級特性。他們不得不費力地找出他們的操作與軟件行為之間的複雜對應關係。這樣他們永遠也得不到良好的服務。

當必須要對系統進行修改或替換時，開發人員（或遲或早）會發現，有一些功能的真實含義並不像它們看上去的那樣。它們在不同的用戶社區或不同情況下具有完全不同的含義。在不破壞這些互相疊加的含義的前提下修改任何東西都是非常困難的。要想把數據遷移到一個「更合適」的系統中，必須要理解這些奇怪的部分，並對其進行編碼。

### 示例 員工工資和養老金系統，第1部分

一家中等規模公司的人力資源部門有一個用於計算工資和養老金代扣的簡單程序。如圖16-15和圖16-16所示。

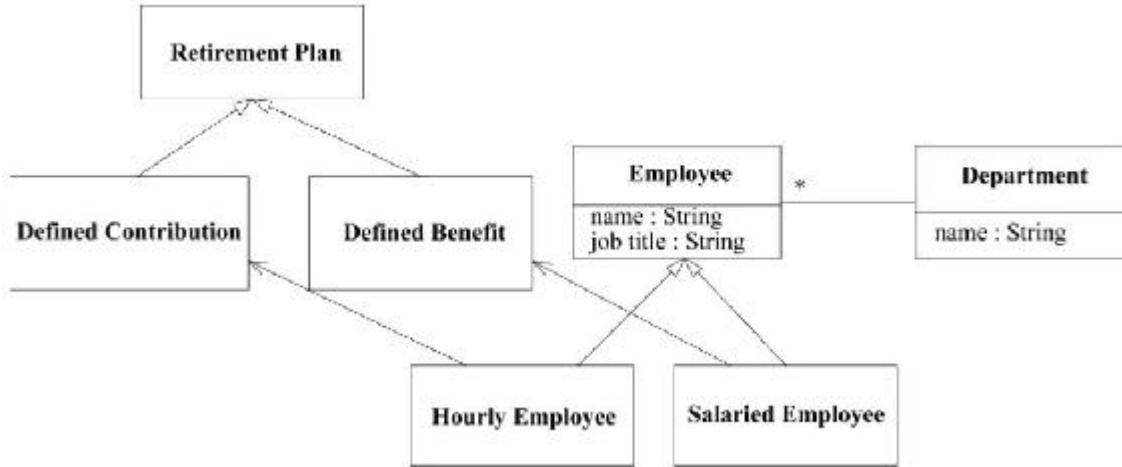


圖16-15 原來的模型，在新的需求下被過多地約束

但現在，管理層決定辦公室行政人員應該進入「固定受益」(Defined Benefit)退休計劃。問題在於辦公室行政人員是按小時付薪酬的，而這個模型不支持混合計算。因此必須修改模型。

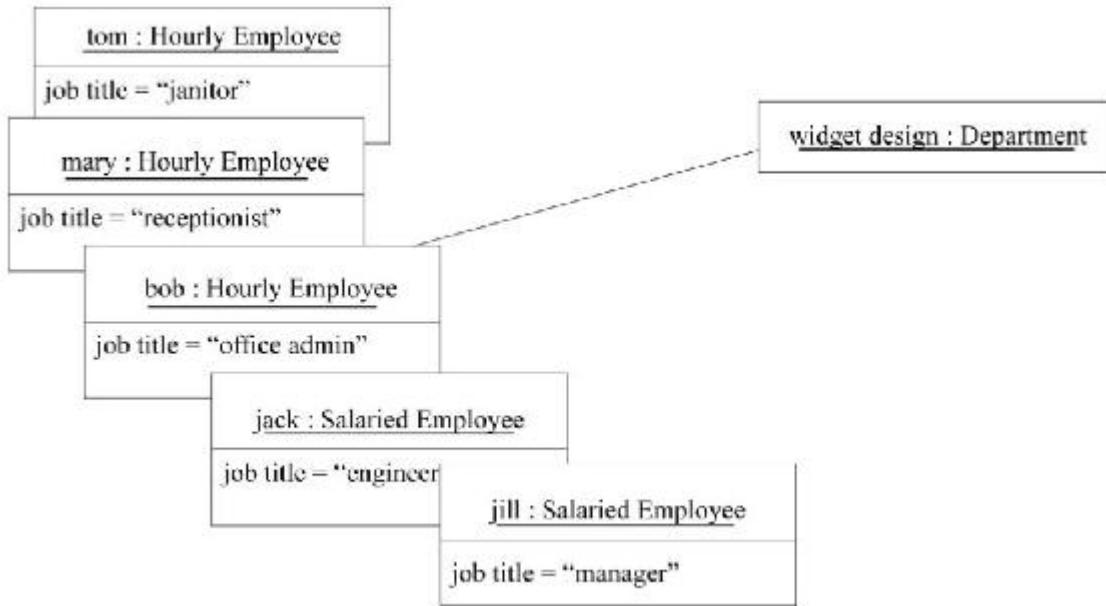


圖16-16 用原來的模型表示出來的一些員工

下面的模型提議非常簡單，只是把約束去掉了，如圖16-17所示。但也會出現一些錯誤，如圖16-18所示。

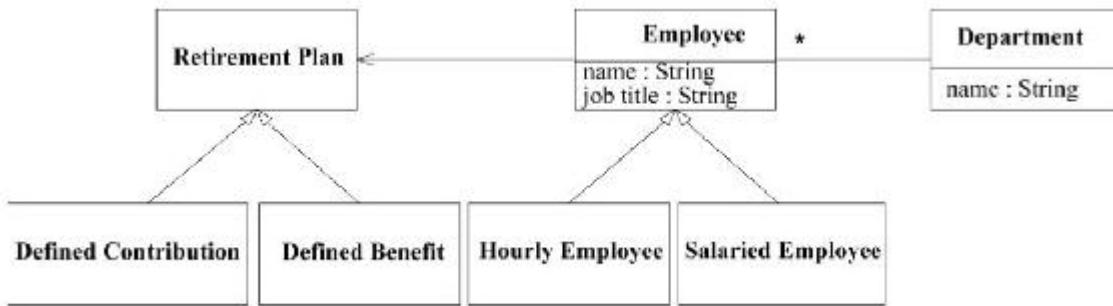


圖16-17 提議的模型，現在的情況是約束過少了

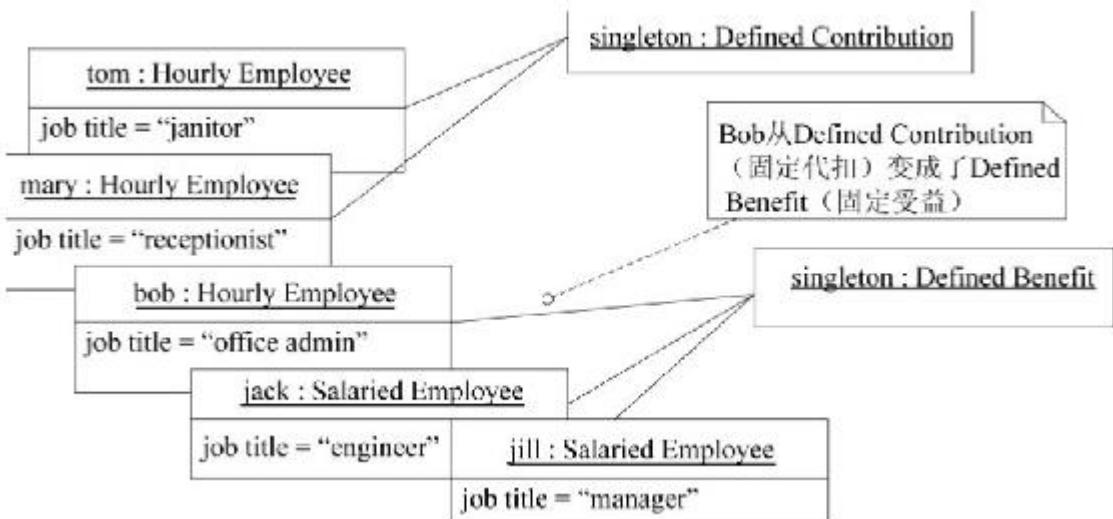


圖16-18 員工可能會與錯誤的計劃關聯起來

在這個模型中，每個員工隨便加入哪一種退休計劃都可以，因此每位辦公室行政人員都可以改變退休計劃。管理層最後放棄了這個模型，因為它沒有反映出公司的策略。一些行政人員可以選擇「固定受益」計劃，而另外一些則不能。要是使用這個模型，連門衛也可以改變退休計劃。管理層需要一個能夠實施以下策略的模型：

辦公室行政人員按小時付薪酬，且採用固定受益退休計劃。

這個策略暗示出job title (工作頭銜) 字段現在表示了一個重要的領域概念。開發人員可以重構模型，用Employee Type (員工類型) 把這個概念明確顯示出來，如圖16-19和圖16-20所示。

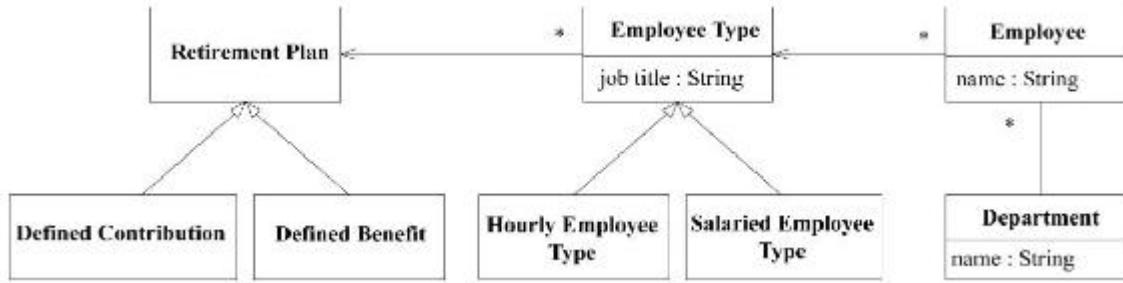


圖16-19 Type對像能夠滿足需求

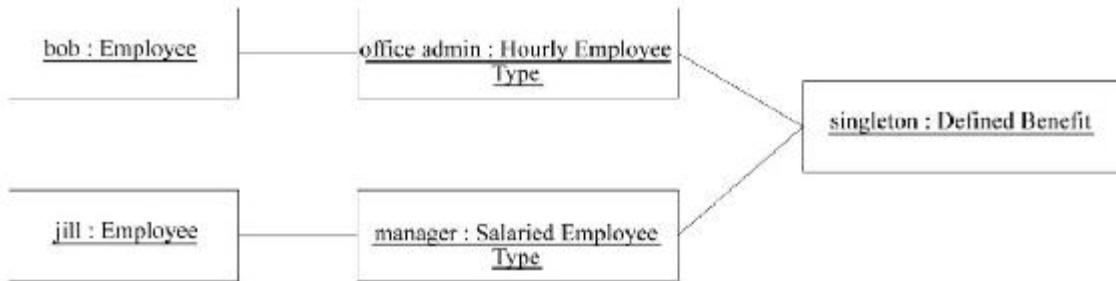


圖16-20 每個Employee Type被指定一個Retirement Plan

需求可以像下面這樣用UBIQUITOUS LANGUAGE來表述出來：

一個EMPLOYEE TYPE可以被指定兩種RETIREMENT PLAN中的任何一種，也可以被指定兩種工資中的任何一種。

EMPLOYEE受EMPLOYEE TYPE約束。

只有superuser ( 超級用戶 ) 才能編輯Employee Type對象，而且只有當公司策略變更時，他才能修改此對象。人事部門的普通用戶只能修改Employee對象，或只能將這些對像指定為另一種Employee Type。

這種模型可以滿足需求。開發人員認識到了一兩個隱含的概念，但這只是靈機一動才想到的。他們並沒有具體的思路可供追查下去，因此他們暫時結束了這一天的工作。

靜態模型可能引起問題。但在一個過於靈活的系統中，如果任何可能的關係都允許存在，問題一樣糟糕。這樣的系統使用起來會很方便，而且會導致組織無法實施自己的規則。

讓每個組織完全定製自己的軟件也是不現實的，即使組織能夠擔負得起定製軟件的費用，組織結構也可能會頻繁變化。

因此，這樣的軟件必須為用戶提供配鎔選項，以便反映出組織的當前結構。問題是，在模型對像中添加這些選項會使這些對像變得難於處理。要求的靈活性越高，模型就會變得越複雜。

如果在一個應用程序中，**ENTITY**的角色和它們之間的關係在不同的情況下有很大變化，那麼複雜性會顯著增加。在這種情況下，無論是一般的模型還是高度定製的模型，都無法滿足用戶的需求。為了兼顧各種不同的情形，對像需要引用其他的類型，或者需要具備一些在不同情況下包括不同使用方式的屬性。具有相同數據和行為的類可能會大量增加，而這些類的唯一作用只是為了滿足不同的組裝規則。

在我們的模型中嵌入了另一個模型，而它的作用只是描述我們的模型。**KNOWLEDGE LEVEL**分離了模型的這個自我定義的方面，並清楚地顯示了它的限制。

**KNOWLEDGE LEVEL**是**REFLECTION**（反射）模式在領域層中的一種應用，很多軟件架構和技術基礎設施中都使用了它，[Buschmann et al.1996] 中給出了詳盡介紹。**REFLECTION**模式能夠使軟件具有 - 自我感知 II 的特性，並使所選中的結構和行為可以接受調整和修改，從而滿足變化需要。這是通過將軟件分為兩個層來實現的，一個層是 - 基礎級別 II ( **base level** )，它承擔應用程序的操作職責；另一個是 - 元級別 II ( **meta level** )，它表示有關軟件結構和行為方面的知識。

值得注意的是，我們並沒有把這種模式叫做知識 - 層 II ( **layer** )。雖然**REFLECTION**與分層很類似，但反射卻包含雙向依賴關係。

Java有一些最基本的內鎔**REFLECTION**機制，它們採用的是協議的形式，用於查詢一個類的方法等。這樣的機制允許用戶查詢有關它

自己的一些設計信息。CORBA也有一些擴展（但類似）的REFLECTION協議。一些持久化技術增加了更豐富的自描述特性，在數據表與對像之間提供了部分自動化的映射。還有其他一些技術例子。這種模式也可以在領域層中使用。

#### KNOWLEDGE LEVEL與REFLECTION所使用的術語比較

Fowler的术语	POSA <sup>1</sup> 的术语
知识级别	元级别
操作级别	基础级别

1 POSA是Pattern-Oriented Software Architecture[Buschmann et al.1996]一書的縮寫。

要明確的一點是，編程語言的反射工具並不是用於實現領域模型的KNOWLEDGE LEVEL的。這些元對像描述的是語言構造本身的結構和行為。相反，KNOWLEDGE LEVEL必須使用普通對像來構造。

KNOWLEDGE LEVEL具有兩個很有用的特性。首先，它關注的是應用領域，這一點與人們所熟悉的REFLECTION模式的應用正好相反。其次，它並不追求完全的通用性。正如一個SPECIFICATION可能比通用的斷言更有用一樣，專門為一組對像和它們的關係定製的一個約束集可能比一個通用的框架更有用。KNOWLEDGE LEVEL顯得更簡單，而且可以傳達設計者的特別意圖。

因此：

創建一組不同的對象，用它們來描述和約束基本模型的結構和行為。把這些對像分為兩個「級別」，一個是非常具體的級別，另一個級別則提供了一些可供用戶或超級用戶定製的規則和知識。

像所有有用的思想一樣，REFLECTION和KNOWLEDGE LEVEL可能令人們感到振奮，但不應濫用這種模式。它確實能夠使對像不必為了滿足各種不同情形下的需求而變得過於複雜，但它所引入的間接性

也會使系統變得更模糊。如果**KNOWLEDGE LEVEL**太複雜，開發人員和用戶就很難理解系統的行為。負責配鎔它的用戶（或超級用戶）最終將需要具備程序員的技能，甚至需要掌握處理元數據的技能。如果他們出現了錯誤，應用程序也將會產生錯誤行為。

而且，數據遷移的基本問題並沒有完全得到解決。當**KNOWLEDGE LEVEL**中的某個結構發生變化時，必須對現有的操作級別中的對象進行相應的處理。新舊對像確實可以共存，但無論如何都需要進行仔細的分析。

所有這些問題為**KNOWLEDGE LEVEL**的設計人員增加了一個沉重的負擔。設計必須足夠健壯，因為不僅要解決開發中可能出現的各種問題，而且還要考慮到將來用戶在配鎔軟件時可能會出現的各種問題。如果得到合理的運用，**KNOWLEDGE LEVEL**能夠解決一些其他方式很難解決的問題。如果系統中某些部分的定製非常關鍵，而要是不提供定製能力就會破壞掉整個設計，這時就可以利用知識級別來解決這一問題。

### **示例 員工工資和養老金系統，第2部分：**KNOWLEDGE LEVEL****

我們的團隊成員又回來了，經過了一夜的休息，他們恢復了精神，團隊中的一個人對系統中一個難處理的問題有了點思路。為什麼有些對像要被限制起來，而其他對像則可以自由編輯呢？那些受限制的對象讓他想到了**KNOWLEDGE LEVEL**模式，他決定嘗試著從這個角度來觀察一下模型，才發現本來就可以用這種方式來觀察模型的。

從圖16-21可以看出，受限制的對象都在**KNOWLEDGE LEVEL**中，而可以自由編輯的對象都在操作級別中，區分得非常清楚。虛線上面的所有對象描述了類型或長期策略。**Employee Type**有效地把行為加在**Employee**上。

這位開發人員把他的想法告訴了大家，這使另一個人又產生了另一個想法。按照**KNOWLEDGE LEVEL**對模型進行組織後，模型變得更清晰了，這使她一下子發現了昨天困擾她的那個問題——兩個完全不同的概念被合併到同一個模型中。昨天她在團隊討論所使用的語言中就聽到了這個問題，只是沒有注意到而已：

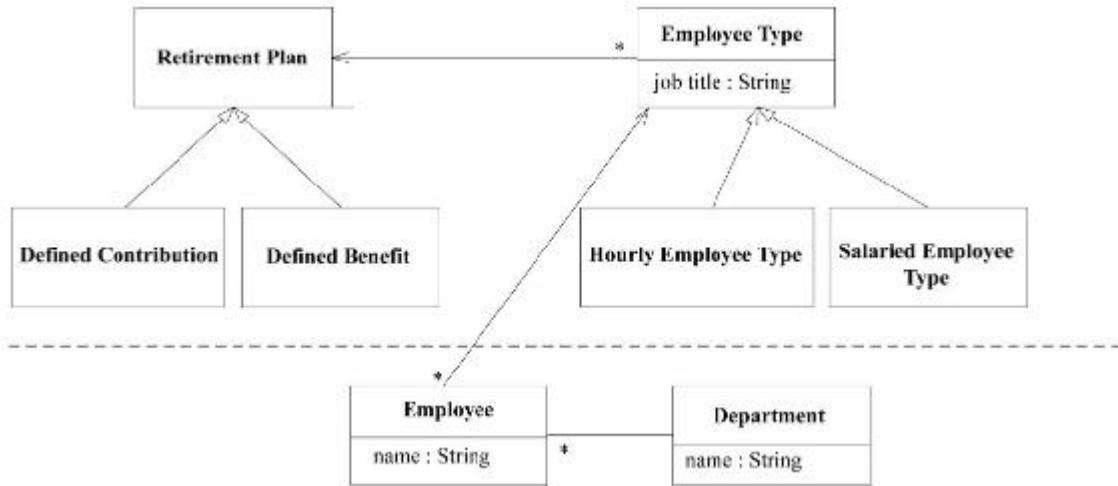


圖16-21 從現有模型中識別出隱含的**KNOWLEDGE LEVEL**

一個**Employee Type**可以被指定兩種**Retirement Plan**中的任何一種，  
也可以被指定兩種工資中的任何一種。

但這實際上並不是用**UBIQUITOUS LANGUAGE**中來表達的聲明。  
模型中並沒有「**payroll**」（工資）。他們只是根據自己的需要來講話，而沒有使用實際就有的通用語言。**payroll**的概念在模型中是隱含的，與**Employee Type**混在一起。在分離出**KNOWLEDGE LEVEL**以前，它並不明顯，而且這個聲明中的所有元素都出現在同一個級別上，只有一個元素例外。

根據這種理解，她重構了一個真正支持該聲明的模型。

為了讓用戶控制那些制約對像之間關聯的規則，開發團隊開發了一個包含隱含**KNOWLEDGE LEVEL**的模型。

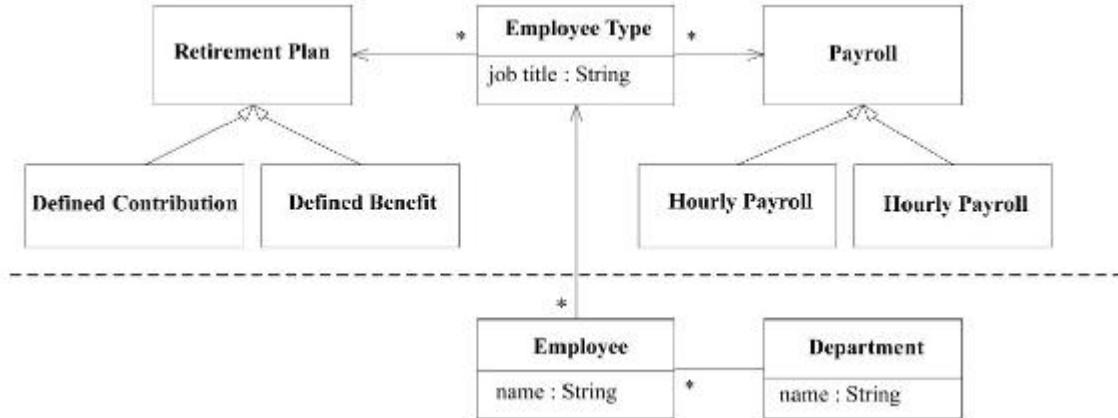


圖16-22 Payroll現在已經顯示出來了，它已與Employee Type分離

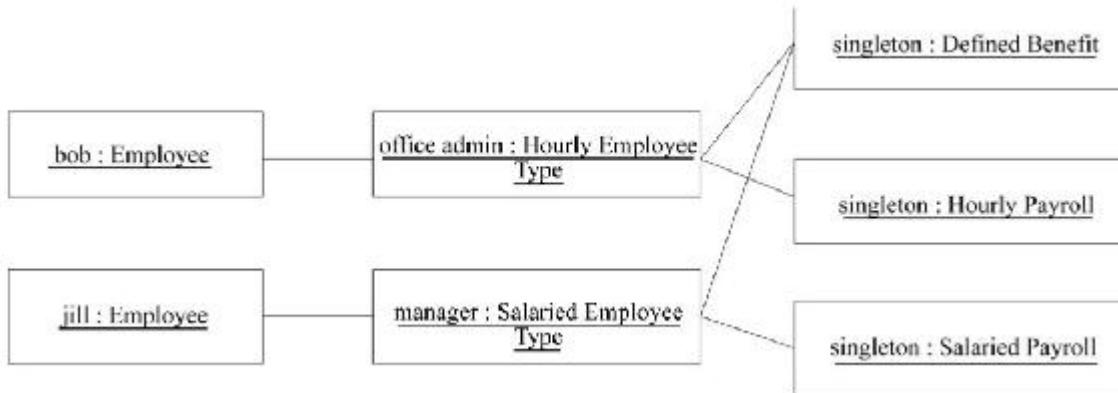


圖16-23 每個Employee Type現在都有一個Retirement Plan和一個Payroll

特有的訪問約束和一種「事物—事物」型的關係對開發團隊起到了提示的作用，使他們看出了隱含的**KNOWLEDGE LEVEL**。一旦**KNOWLEDGE LEVEL**被分離出來，它就能夠使模型變得非常清晰，從而可以通過提取出Payroll將兩個重要的領域概念分開。

像其他大比例結構一樣，**KNOWLEDGE LEVEL**也不是必須要使用的。沒有它，對像照樣能工作，而且團隊可能仍能夠認識到他們需要將Employee Type與Payroll分離。當項目進行到某個時刻，這種結構看起來已經沒什麼作用了，那麼就可以放棄它。但現在它對於描述系統很有用，並且能夠幫助開發人員理解模型。

乍看上去，KNOWLEDGE LEVEL像是RESPONSIBILITY LAYER（特別是策略層）的一個特例，但它並不是。首先，兩個級別之間的依賴性是雙向的，而在層次結構中，較低的層不依賴於較高的層。

實際上，RESPONSIBILITY LAYER可以與其他大部分的大比例結構共存，它提供了另一種用來組織模型的維度。

## **16.5 模式：PLUGGABLE COMPONENT FRAMEWORK**

在深入理解和反覆精煉基礎上得到的成熟模型中，會出現很多機會。通常只有在同一個領域中實現了多個應用程序之後，才有機會使用PLUGGABLE COMPONENT FRAMEWORK（可插入式組件框架）。

當很多應用程序需要進行互操作時，如果所有應用程序都基於相同的一些抽象，但它們是獨立設計的，那麼在多個**BOUNDED CONTEXT**之間的轉換會限制它們的集成。各個團隊之間如果不能緊密地協作，就無法形成一個**SHARED KERNEL**。重複和分裂將會增加開發和安裝的成本，而且互操作會變得很難實現。

一些成功的項目將它們的設計分解為組件，每個組件負責提供某些類別的功能。通常所有組件都插入到一個中央hub上，這個hub支持組件所需的所有協議，並且知道如何與它們所提供的接口進行對話。還有其他一些將組件連在一起的可行模式。對這些接口以及用於連接它們的hub的設計必須要協調，而組件內部的設計則可以更獨立一些。

有幾個廣泛使用的技術框架支持這種模式，但這只是次要問題。一種技術框架只有在能夠解決某類重要技術問題的時候才有必要使用，如在設計分佈式系統或在不同應用程序中共享一個組件時。可插

入式組件框架的基本模式是職責的概念組織，它很容易在單個的Java程序中使用。

因此：

從接口和交互中提煉出一個**ABSTRACT CORE**，並創建一個框架，這個框架要允許這些接口的各種不同實現被自由替換。同樣，無論是什麼應用程序，只要它嚴格地通過**ABSTRACT CORE**的接口進行操作，那麼就可以允許它使用這些組件。

高層抽象被識別出來，並在整個系統範圍內共享，而特化（specialization）發生在MODULE中。應用程序的中央hub是SHARED KERNEL內部的**ABSTRACT CORE**。但封裝的組件接口可以把多個BOUNDED CONTEXT封裝到其中，這樣，當很多組件來自多個不同地方時，或者當組件中封裝了用於集成的已有軟件時，可以很方便地使用這種結構。

這並不是說不同組件一定要使用不同的模型。只要團隊採用了CONTINUOUS INTEGRATE，或者為一組密切相關的組件定義了另一個SHARED KERNEL，那麼就可以在同一個CONTEXT中開發多個組件。在PLUGGABLE COMPONENT FRAMEWORK這種大比例結構中，所有這些策略很容易共存。在某些情況下，還有一種選擇是使用一種PUBLISHED LANGUAGE來編寫hub的插入接口。

PLUGGABLE COMPONENT FRAMEWORK也有幾個缺點。一個缺點是它是一種非常難以使用的模式。它需要高精度的接口設計和一個非常深入的模型，以便把一些必要的行為捕獲到**ABSTRACT CORE**中。另一個很大的缺點是它只為應用程序提供了有限的選擇。如果一個應用程序需要對CORE DOMAIN使用一種非常不同的方法，那麼可插入式組件框架將起到妨礙作用。開發人員可以對模型進行特殊修改，但如果不能更改所有不同組件的協議，就無法修改**ABSTRACT**

CORE。這樣一來，CORE的持續精化過程（也是通過重構得到更深層理解的過程）在某種程度上會陷入僵局。

[Fayad and Johnson 2000] 中詳細介紹了在幾個領域中使用PLUGGABLE COMPONENT FRAMEWORK的大膽嘗試，其中包括對SEMATECH CIM框架的討論。要想成功地使用這些框架，需要綜合考慮很多事情。最大的障礙可能就是人們的理解不那麼成熟，要想設計一個有用的框架，必須要有成熟的理解。PLUGGABLE COMPONENT FRAMEWORK不適合作為項目的第一個大比例結構，也不適合作為第二個。最成功的例子都是在完全開發出了多個專門應用之後才採用這種結構的。

### 示例 SEMATECH CIM 框架

在一家生產計算機芯片的工廠中，一組一組的硅片（稱為lot）從一臺機器傳送到另一臺機器，通過上百道加工工序，直到印刷上微電路並完成蝕刻。工廠需要一個軟件來跟蹤每個lot，記錄下來它上面已經完成的加工，然後指揮工人或自動設備把它送到下一臺正確的機器上，並進行下一次正確的加工。這樣的軟件稱為製造執行系統（Manufacturing Execution System，MES）。

工廠使用了數十家供應商生產的數百臺不同的機器，每道工序都仔細設計了定製的配方。為這個複雜的混合加工過程開發MES軟件是一項異常艱巨的任務，而且費用也十分高昂。為瞭解決這些問題，SEMATECH（一家行業協會）開發了CIM框架。

CIM框架龐大而複雜，它有很多方面，但只有兩個方面與我們這裡的討論相關。首先，這個框架為半導體MES領域的基本概念定義了抽象接口，換言之，以ABSTRACT CORE的形式定義了CORE DOMAIN。這些接口定義既包括行為上的，也包括語義上的。

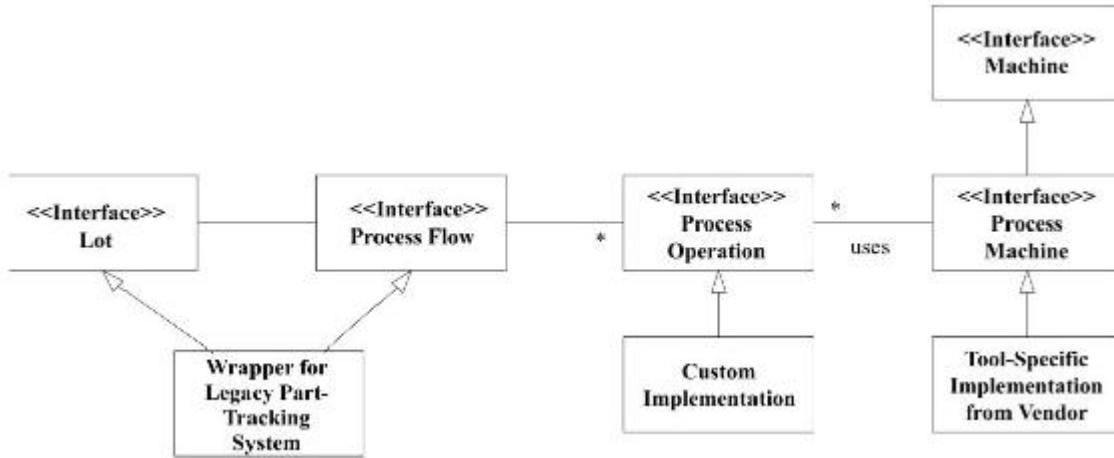


圖16-24 高度簡化的CIM接口子集，提供了一些實現樣例

如果某家供應商生產了一種新的機器，他們必須開發 **Process Machine** 接口的一個專用實現。只要他們遵守該接口，他們的機器控制組件就可以插入到任何基於CIM框架的應用程序中。

在定義了這些接口之後，SEIMATECH又定義了組件在應用程序中進行交互時需要遵守的規則。任何基於CIM框架的應用程序都必須實現一個協議，通過這個協議來為那些已經實現部分接口的對象提供服務。如果這個協議已經實現，而且應用程序嚴格遵守抽象接口，那麼這個應用程序就可以使用這些接口所提供的服務，而不用管它們是如何實現的。這些接口以及為了使用接口而實現的協議組合在一起，構成了具有嚴格限制的大比例結構。



圖16-25 用戶把一個lot放到下一臺機器上，並把這次操作記錄到計算機中

這個框架需要使用專門的基礎設施。它主要使用CORBA來提供持久化、事務、事件和其他技術服務。但它的**PLUGGABLE COMPONENT FRAMEWORK**的定義很有趣，它允許人們獨立開發軟件，並把開發出來的軟件平滑地集成到龐大的系統中。沒有人會知道這個系統中的所有細節，但每個人都理解整體視圖。

數千人是如何分工來製作一個由40 000多塊組成的「艾滋病紀念拼被」的？

幾條簡單的規則為「艾滋病拼圖被子」提供了一種大比例結構，而細節則由各個志願者來完成。注意規則重點關注的3個方面，一是整體任務（紀念那些因艾滋病而死去的人們），二是各個小塊所具有的那些使其容易拼到整體中的特性，三是處理更大的塊的能力（如把它折疊起來）。

以下就是艾滋病紀念拼被的一個拼塊的製作方法

[摘自艾滋病紀念拼被網站，[www.aidsquilt.org](http://www.aidsquilt.org)]

設計拼塊

把要紀唸的人的名字寫到拼塊上。可以自由加入其他一些信息，如出生、死亡日期和出生地等，每個拼塊僅限一人……

### 選擇你的材料

記住，被單要被折疊和打開許多次，因此材料的耐久性很重要。由於膠會隨著時間失效，因此最好把東西縫到拼塊上。最好使用重量適中、不具有拉伸性的布料，如棉帆布或毛葛。

設計可以採用橫向或縱向，但最終鑲好邊的拼塊必須是3英尺×6英尺 ( 90 cm×180 cm ) ——不能多也不能少！裁剪布料時，每個邊留出2~3英吋的鑲邊。如果你自己不能鑲邊，我們會為你代勞。無需為拼塊縫製夾層，但建議在背面縫一個襯墊，這樣當把拼塊放到地上時，可以保持乾淨，也有助於保持布料不變形。

### 製作拼塊

製作拼塊時可能會用到以下技術。

縫飾：在背景布料上縫上其他的織物、信件或小的紀念品。不要使用膠水，因為它很容易失效。

用顏料塗色：刷上紡織顏料或快速上色染料，也可以使用不褪色的墨水筆。不要使用「棉花彩」[\[6\]](#)，因為它的黏性太大了。

模繪：用鉛筆把你的設計畫到布料上，然後把得到的模板墊高，再用刷子塗上紡織顏料或不褪色的標記。

拼貼：在拼塊上使用的材料一定不要把布料劃破（因此不要使用玻璃和金屬片），還要注意不要使用體積很大的物品。

照片：加照片或信件的最好方法是把它們影印到燙印轉印紙 ( iron-on transfer ) 上，再由燙印轉印紙印到100%的純棉布料上，再把這塊布料縫到拼塊上。也可以用乙烯材料把照片塑封起來，再縫到拼塊上（不要放在中央，以避免折疊）。

## 16.6 結構應該有一種什麼樣的約束

本章所討論的大比例結構很廣泛，從非常寬鬆的 SYSTEM METAPHOR 到嚴格的 PLUGGABLE COMPONENT FRAMEWORK。當然，還有很多其他結構，而且，甚至在一個通用的結構模式中，在制定規則上也可以選擇多種不同的嚴格程度。

例如，RESPONSIBILITY LAYER 規定了一種用於劃分模型概念以及它們的依賴性的方式，但我們也可以添加一些規則，來指定各個層之間的通信模式。

假設有一家製造廠，每個零件在哪臺機器上加工（根據工藝配方）完全由軟件來指揮。正確的加工命令是從策略層發出的，並在作業層執行。但工廠的實際生產不可避免地會有錯誤。實際情況將與軟件的規則不符。現在，作業層必須要反映出工廠的實際情況，這意味著當一個零件偶然被放到一臺錯誤的機器上時，機器必須無條件地接受它。這種異常情況需要以某種方式傳遞到更高的層。然後，決策制定層可以利用其他策略來糾正這種情況，可以把該零件重新送到修理流程或直接丟棄它。但作業層不知道較高層的任何信息。通信必須是單向的，不能讓較低層產生對較高層的依賴性。

通常，這種信號傳遞是通過某種事件機制實現的。每當作業層對象的狀態發生變化時，它們就將生成事件。策略層對像將監聽來自較低層的相關事件。如果一個事件違反了某個規則，該規則將執行一個動作（規則定義的一部分）來給出適當的響應，或者生成一個事件反饋給更高的層，以便幫助更高的層做出決策。

例如，在銀行中，當投資組閣中的某些部分發生變動時，資產的價值會發生改變（作業層）。當這些值超過投資組合的配額限制時

(策略層)，交易商可能就會接到通知，然後他可以通過買入或賣出資產來恢復平衡。

我們可以為每種不同的情況設計不同的事件機制，也可以讓特殊層中的對象在交互時遵守一種一致的模式。結構越嚴格，一致性就越高，設計也越容易理解。如果結構適當的話，規則將推動開發人員得出好的設計。不同的部分之間會更協調。

另一方面，約束也會限制開發人員所需的靈活性。在異構系統中，特別是當系統使用了不同的實現技術時，可能無法跨越不同的 BOUNDED CONTEXT來使用非常特殊的通信路徑。

因此一定要剋制，不要濫用框架和死板地實現大比例結構。大比例結構的最重要的貢獻在於它具有概念上的一致性，並幫助我們更深入地理解領域。每條結構規則都應該使開發變得更容易實現。

## **16.7 通過重構得到更適當的結構**

在當今這個時代，軟件開發行業正在努力擺脫過多的預先設計，因此一些人會把大比例結構看作是倒退回了過去那段使用瀑布架構的令人痛苦的年代。但實際上，只有深入地理解領域和問題才能發現一種非常有用的結構，而獲得這種深刻的理解的有效方式就是迭代開發過程。

團隊要想堅持EVOLVING ORDER原則，必須在項目的整個生命週期中大膽地反覆思考大比例結構。團隊不應該一成不變地使用早期構思出來的那個結構，因為那時所有人對領域或需求的理解都不夠完善。

遺憾的是，這種演變意味著最終的結構不會在項目一開始就被發現，而且我們必須在開發過程中進行重構，以便得到最終的結構。這

可能很難實現，而且需要高昂的代價，但這樣做是非常必要的。有一些通用的方法可以幫助控制成本並最大化收益。

### **16.7.1 最小化**

控制成本的一個關鍵是保持一種簡單、輕量級的結構。不要試圖使結構面面俱到。只需解決最主要的問題即可，其他問題可以留到後面一個一個地解決。

開始最好選擇一種鬆散的結構，如SYSTEM METAPHOR或幾個RESPONSIBILITY LAYER。不管怎樣，一種最小化的鬆散結構可以起到輕量級的指導作用，它有助於避免混亂。

### **16.7.2 溝通和自律**

整個團隊在新的開發和重構中必須遵守結構。要做到這一點，整個團隊必須理解這種結構。必須把術語和關係納入到UBIQUITOUS LANGUAGE中。

大比例結構為項目提供了一個術語表，它概要地描述了整個系統，並且使不同人員能夠做出一致的決策。但由於大多數大比例結構只是鬆散的概念指導，因此團隊必須要自覺地遵守它。

如果很多人不遵守結構，它慢慢就會失去作用。這時，結構與模型和實現的各個部分之間的關係無法總是在代碼中明確地反映出來，而且功能測試也不再依賴結構了。此外，結構往往是抽象的，因此很難保證在一個大的團隊（或多個團隊）中一致地應用它。

在大多數團隊中，僅僅通過溝通是不足以保證在系統中採用一致的大比例結構的。至關重要的一點是要把它合併到項目的通用語言中，並讓每個人都嚴格地使用UBIQUITOUS LANGUAGE。

### **16.7.3 通過重構得到柔性設計**

其次，對結構的任何修改都可能導致大量的重構工作出現。隨著系統複雜度的增加和人們理解的加深，結構會不斷演變。每次修改結構時，必須修改整個系統，以便遵守新的秩序。顯然這需要付出大量工作。

但這並不像聽上去那麼糟糕。根據我的觀察，採用了大比例結構的設計往往比那些未採用的設計更容易轉換。即使是從一種結構更改為另一種結構（例如，從METAPHOR改為LAYER）也是如此。我無法完全解釋清楚這是什麼原因。部分原因是當完全理解了某個系統的當前佈局之後，再重新安排它就會更容易，而且先前的結構使得重新佈局變得更容易。還有部分原因是用於維護先前結構的那種自律性已經滲透到了系統的各個方面。但我覺得還有更多的原因，因為當一個系統先前已經使用了兩種結構時，它的更改甚至更加容易。

一件新皮茄克穿起來又硬又不舒服，但穿了一天之後，肘部經過若干次彎曲後就會變得更容易彎曲。再穿幾天之後，肩部也會變得寬鬆，茄克也更容易穿上了。幾個月後，皮質開始變得柔軟，穿著會更舒適，也更容易穿上。同樣，對模型反覆進行合理的轉換也有相同效果。不斷增加的知識被合併到模型中，更改的要點已經被識別出來，並且更改也變得更靈活，同時模型中一些穩定的部分也得到了簡化。這樣，底層領域的更顯著的CONCEPTUAL CONTOUR就會在模型結構中浮現出來。

#### 16.7.4 通過精煉可以減輕負擔

對模型施加的另一項關鍵工作是持續精煉。這可以從各個方面減小修改結構的難度。首先，從CORE DOMAIN中去掉一些機制、GENERIC SUBDOMAIN和其他支持結構，需要重構的內容就少多了。

如果可能的話，應該把這些支持元素簡單地定義成符合大比例結構的形式。例如，在一個RESPONSIBILITY LAYER系統中，可以把

**GENERIC SUBDOMAIN**定義成只適合放到某個特定層中。當使用了**PLUGGABLE COMPONENT FRAMEWORK**的時候，可以把**GENERIC SUBDOMAIN**定義成完全由某個組件擁有，也可以定義成一個**SHARED KERNEL**，供一組相關組件使用。這些支持元素可能需要進行重構，以便找到它們在結構中的適當位臘，但它們的移動與**CORE DOMAIN**是獨立的，而且移動也限制在很小的範圍內，因此更容易實現。最後，它們都是次要元素，因此它們的精化不會影響大局。

通過精煉和重構得到更深層理解的原理甚至也適用於大比例結構本身。例如，最初可以根據對領域的初步理解來選擇分層結構，然後逐步用更深層次的抽象（這些抽象表達了系統的基本職責）來代替它們。這種極高的清晰度使人們能夠透徹地理解領域，這也是我們的目標。它也是一種使系統的整體控制變得更容易、更安全的手段。

# 第17章 領域驅動設計的綜合運用

前面3章給出了戰略層面上應用領域驅動設計的很多原則和技術。在一個大的、複雜的系統中，可能需要在一個設計中綜合運用幾種策略。那麼，大型結構如何與CONTEXT MAP共存？應該把構造塊放到哪裡？第一步先做什麼？第二步和第三步呢？如何設計你的戰略？

## 17.1 把大型結構與BOUNDED CONTEXT結合起來使用

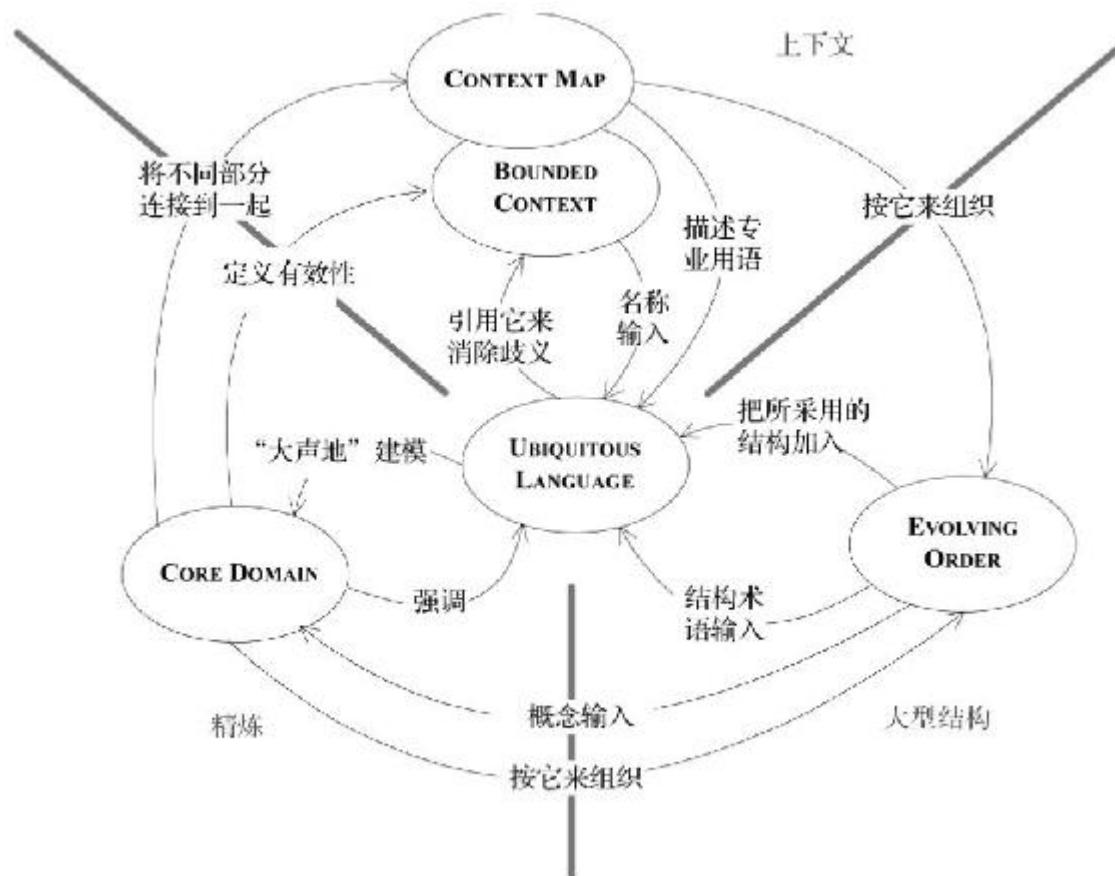


圖17-1

戰略設計的3個基本原則（上下文、精煉和大型結構）並不是可以互相代替的，而是互為補充，並且以多種方式進行互動。例如，一種大型結構可以存在於一個**BOUNDED CONTEXT**中，也可以跨越多個**BOUNDED CONTEXT**存在，並用於組織**CONTEXT MAP**。

前面的**RESPONSIBILITY LAYER**的例子被限定在一個**BOUNDED CONTEXT**中。這是解釋這一思想的最簡單的方法，也是該模式的一般用法。在這樣的簡單場景中，層名稱的含義僅用於該**CONTEXT**，該**CONTEXT**中的模型元素或子系統接口的名稱也是如此。

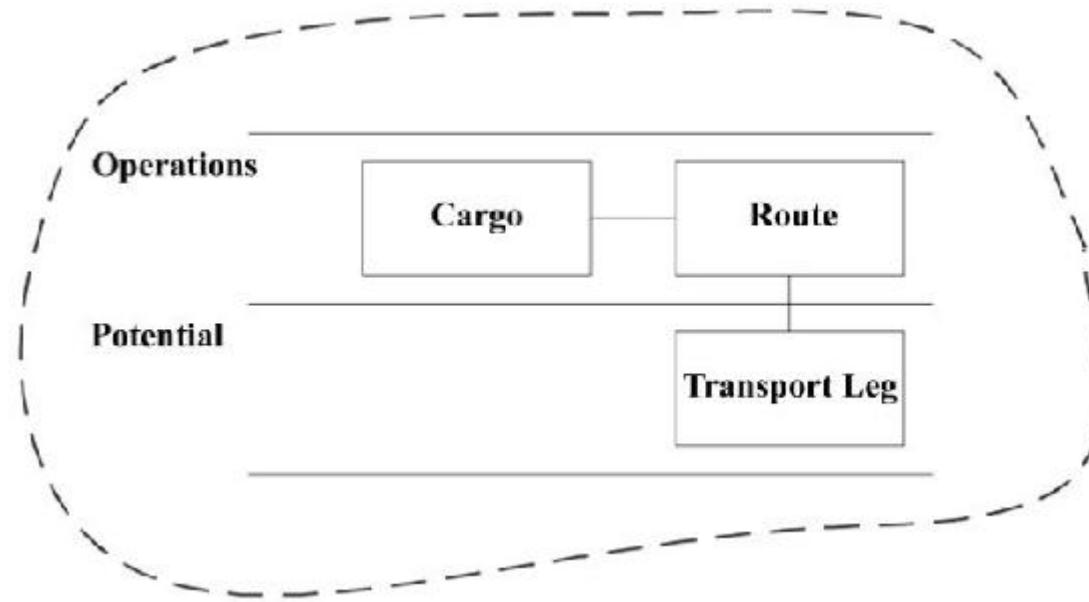


圖17-2 在單一的**BOUNDED CONTEXT**內部構造一個模型

這樣的局部結構在一個非常複雜但統一的模型中是很有用的，它使系統所能承受的複雜度上限提高了，進而使得在一個**BOUNDED CONTEXT**中可以維護更多的對象。

但是在很多項目中，更大的挑戰是理解怎樣使各個不同的部分構成一個整體，如圖 17-3 所示。這些部分可能被劃分到不同的**BOUNDED CONTEXT**中，但是各個部分在整個集成系統中的作用是什

麼，它們之間又是如何互相關聯的？理解了這些問題之後就可以用大型結構來組織CONTEXT MAP。在這種情況下，結構的術語適用於整個項目（或至少是項目中某個明確限定的部分）。

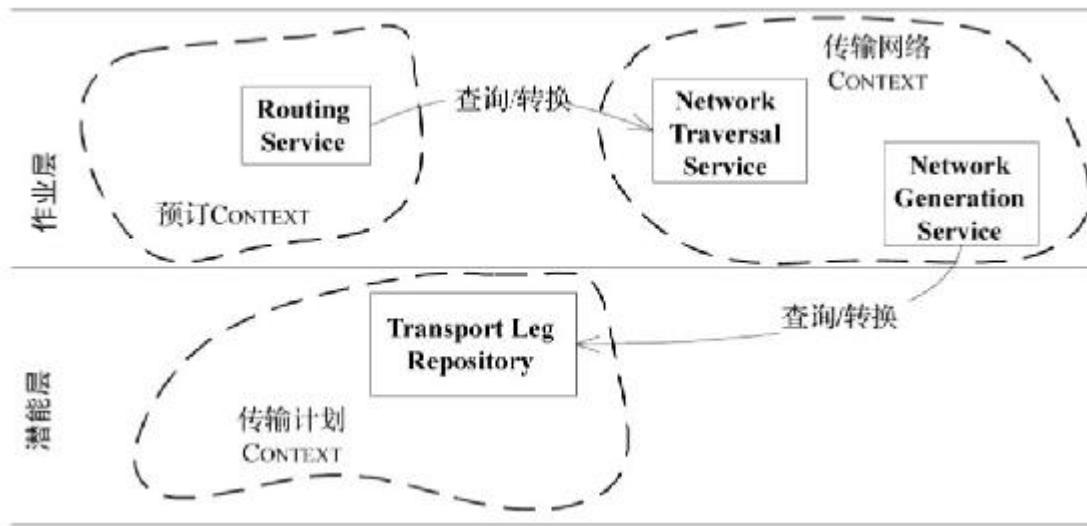


圖17-3 在不同BOUNDED CONTEXT的組件關係上所使用的結構

假設你打算採用RESPONSIBILITY LAYER模式，但你有一個遺留系統，它的組織結構與你想要採用的大型結構不一致。那麼是否必須放棄LAYERS模式？不必，但是你必須確定遺留系統在新結構中的位罷，如圖17-4所示。實際上，RESPONSIBILITY LAYER可能有助於刻畫遺留系統的特徵。遺留系統所提供的SERVICE可以被限定到幾個層中。如果我們能夠說出遺留系統與哪幾個特定的RESPONSIBILITY LAYER相符，那麼這就非常精確地描述了遺留系統的範圍和角色的關鍵方面。

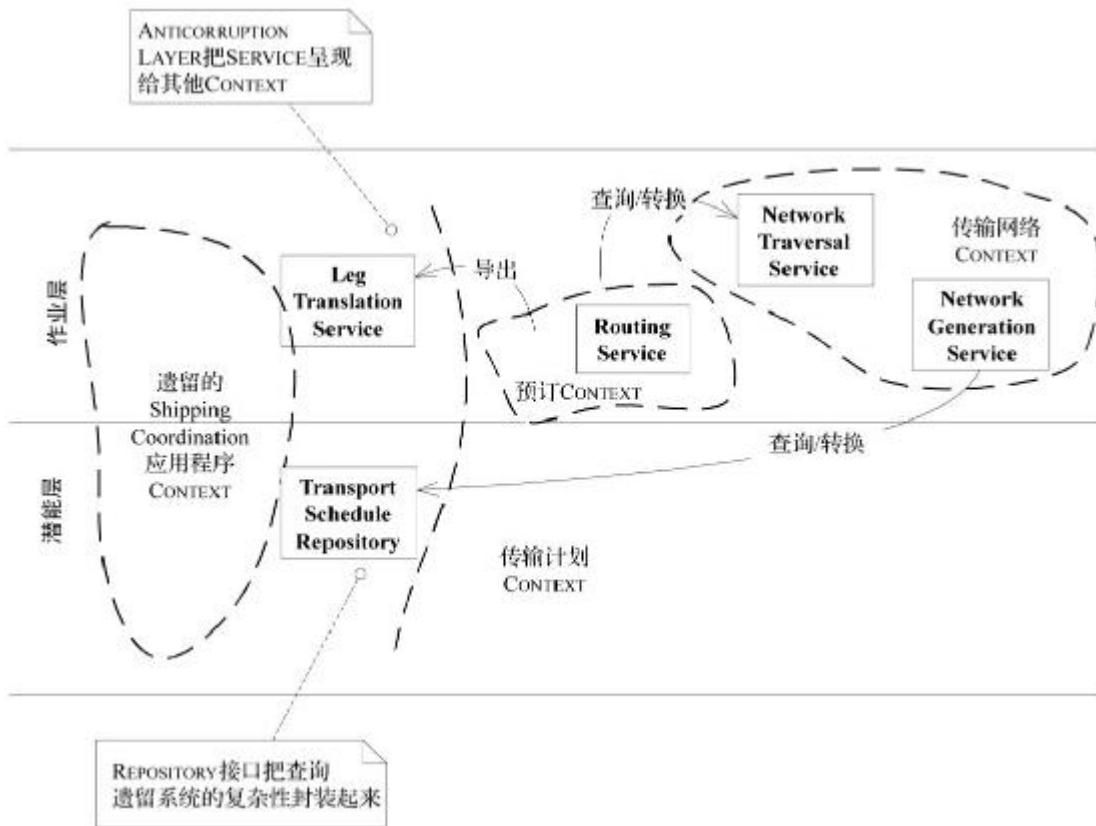


圖17-4 允許一些組件跨越多個層的結構

如果遺留子系統的功能是通過一個`FACADE`來訪問的，那麼設計時，該`FACADE`所提供的每個`SERVICE`應該只在一個層中，不跨越多個層。

在這個示例中，`Shipping Coordination`應用程序是一個遺留系統，它的內部機制是作為一個無差別的整體呈現出來的。但如果項目團隊已經很好地建立了一種跨`CONTEXT MAP`的大型結構，那麼團隊可以選擇在他們的`CONTEXT`中按照已經熟悉的層來組織模型，如圖17-5所示。

當然，由於每個`BOUNDED CONTEXT`都是其自己的命名空間，因此在一個`CONTEXT`中可以使用一種結構來組織模型，而在相鄰的`CONTEXT`中則可以使用另一種結構，然後再使用一種別的結構來組織

CONTEXT MAP。但是，使用過多的結構會損害大型結構作為項目統一概念集的價值。

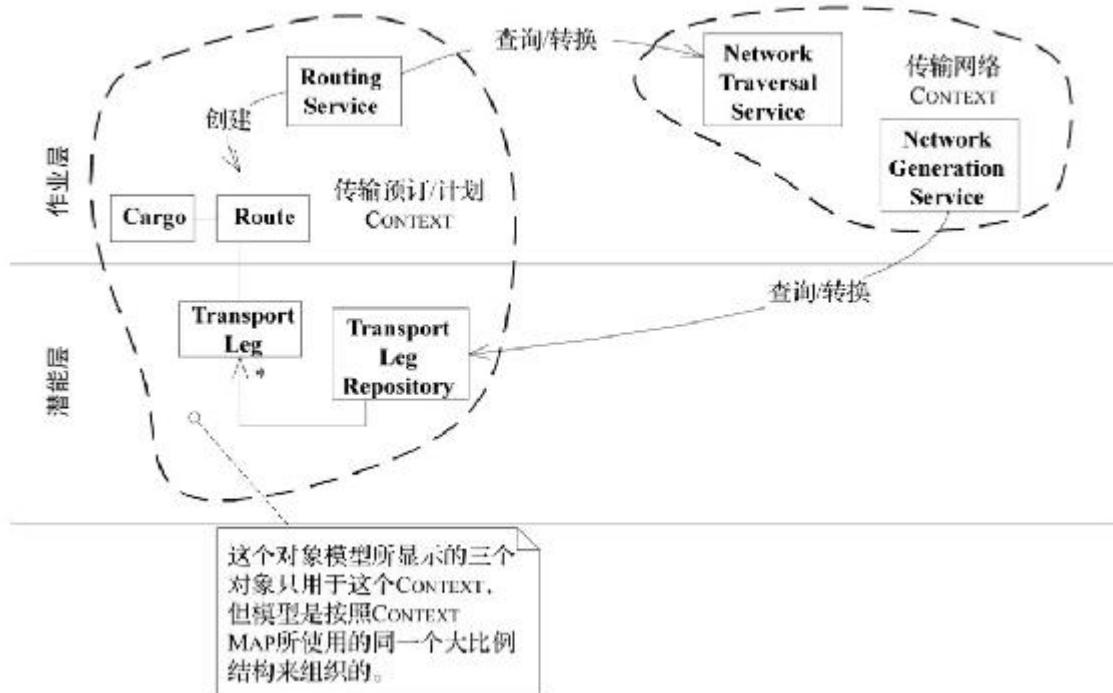


圖17-5 在一個CONTEXT中和整個CONTEXT MAP (作為一個整體) 中使用同一種結構

## 17.2 將大型結構與精煉結合起來使用

大型結構和精煉的概念也是互為補充的。大型結構可以幫助解釋 CORE DOMAIN 內部的關係以及 GENERIC SUBDOMAIN 之間的關係，如圖17-6所示。

同時，大型結構本身可能也是CORE DOMAIN的一個重要部分。例如，把潛能層、作業層、策略層和決策支持層區分開，能夠提煉出對軟件所要解決的業務問題的基本理解。當項目被劃分為多個 BOUNDED CONTEXT 時，這種理解尤其有用，這樣CORE DOMAIN的模型對象就不會具有過多的含義。

## 17.3 首先評估

當對一個項目進行戰略設計時，首先需要清晰地評估現狀。

- (1) 畫出CONTEXT MAP。你能畫出一個一致的圖嗎？有沒有一些模稜兩可的情況？
- (2) 注意項目上的語言使用。有沒有UBIQUITOUS LANGUAGE？這種語言是否足夠豐富，以便幫助開發？
- (3) 理解重點所在。CORE DOMAIN被識別出來了嗎？有沒有DOMAIN VISION STATEMENT？你能寫一個嗎？
- (4) 項目所採用的技術是遵循MODEL-DRIVEN DESIGN，還是與之相悖？
- (5) 團隊開發人員是否具備必要的技能？
- (6) 開發人員是否瞭解領域知識？他們對領域是否感興趣？

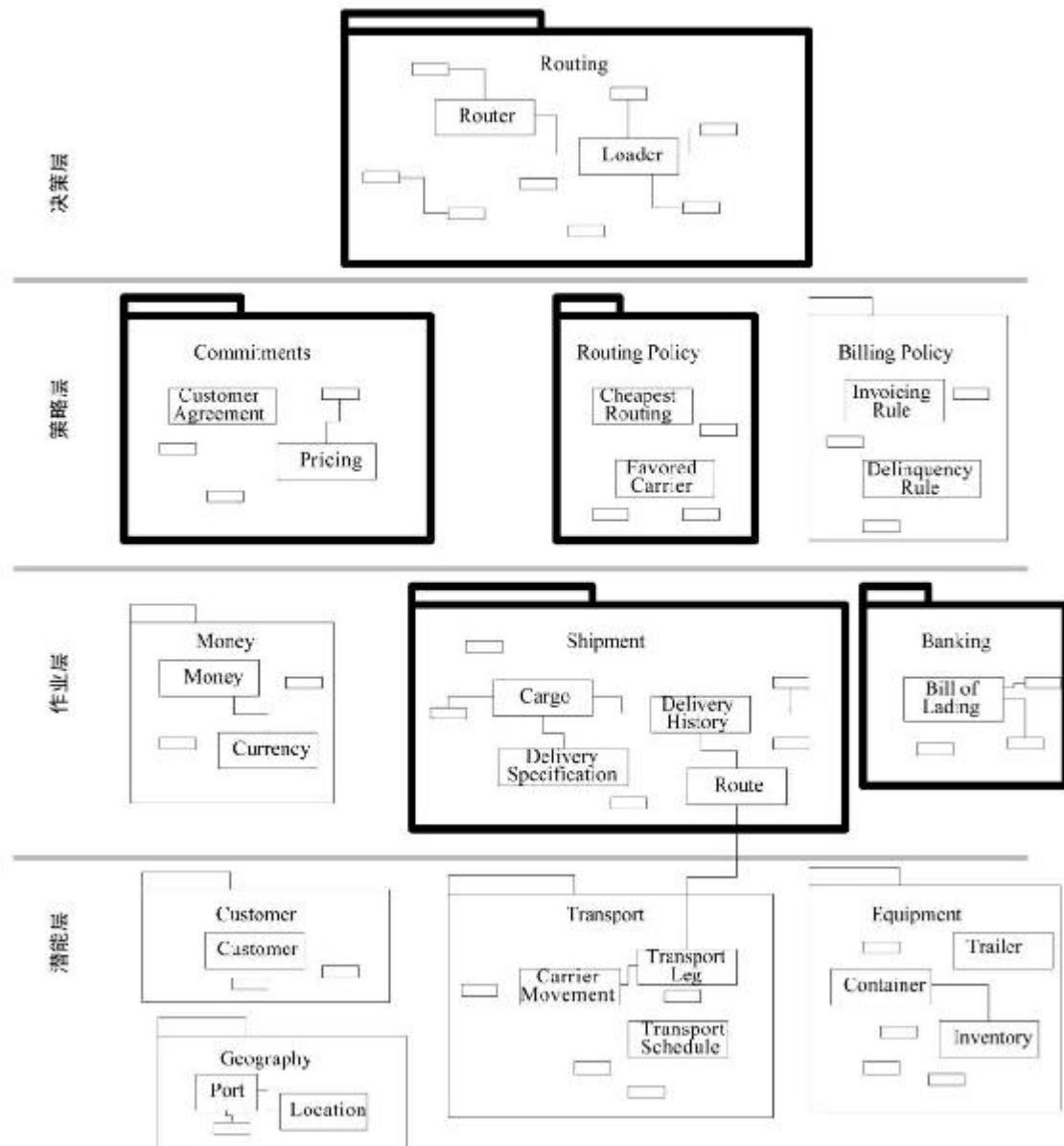


圖17-6 通過分層把CORE DOMAIN的MODULE (用粗框顯示) 和GENERIC SUBDOMAIN分得更清楚

當然，我們不會發現完美的答案。我們現在對項目的瞭解永遠不如將來的瞭解深入。但這些問題為我們提供了一個可靠的起點。當知道了這些問題的初步答案後，我們就會明白什麼是最迫切需要解決的。隨著時間的推進，我們可以得出更精煉的答案，特別是CONTEXT

MAP、DOMAIN VISION STATEMENT，以及其他創建出來的工件，這些答案都反映出了變化的情況和新的理解。

## 17.4 由誰制定策略

傳統上，架構是在應用程序開發開始之前建立的，並且在這種組織中，負責建立架構的團隊比應用開發團隊擁有更大的權力。但我們並不一定得遵循這種傳統的方式，因為它並不總是十分有效。

戰略設計必須明確地應用於整個項目。項目有很多組織方式，這一點我並不想做過多的說明。但是，要想使決策制定過程更有效，需要注意一些基本問題。

首先，我們簡單介紹一下我曾見過的兩種在實踐中具有一定價值的風格（摒棄了傳統的「由高層制定決策」的做法）。

### 17.4.1 從應用程序開發自動得出的結構

一個非常善於溝通、懂得自律的團隊在沒有核心領導的情況下照樣能夠很好地工作，他們能夠遵循EVOLVING ORDER來達成一組共同遵守的原則，這樣就能夠有機地形成一種秩序，而不用靠命令來約束。

這是極限編程團隊的典型模式。從理論上講，任何一對兒編程人員都可以根據自己的理解來完全自發地創建一種結構。通常，讓團隊中的一個人（或幾個人）來承擔大型結構的一些監管職責有利於保持結構統一。如果這位承擔監管職責的非正式的領導人也是一位負責具體工作的開發人員（仲裁者和協調員），而不是決策的唯一制定者，那麼這種方法將特別有效。在我見過的極限編程團隊中，這樣的策略設計領導者可能會自動出現，而且通常在教練中產生。不管這個自動出現的領導人是誰，他仍然是開發團隊的成員之一。由此可見，開發

團隊必須至少有幾位具有這樣才幹的人，由他們來制定一些運用到整個項目中的設計決策。

當多個團隊使用同一種大型結構時，密切相關的團隊可以開始非正式的協作。在這種情況下，對這種大型結構，每個應用程序團隊仍會產生各自的想法，而其中一些具體選擇會由一個非正式的委員會來討論，這個委員會由各個團隊的代表組成。在評估了這些選擇對設計的影響之後，委員會決定是採用它、修改它，還是放棄它。團隊在這種鬆散的合作關係下一起前進。這種安排要想發揮作用，需要保證：團隊數目相對較少，各個團隊之間能夠一致地保持彼此協調，他們的設計能力大致相同，而且他們的結構需求基本一致，可以通過同一種大型結構來滿足。

#### **17.4.2 以客戶為中心的架構團隊**

當幾個團隊共用同一種策略時，確實需要集中制定一些決策。架構師如果脫離實際開發工作，就可能會設計出失敗的模型，但這是完全可以避免的。架構團隊可以把自己放在與應用開發團隊平等的位階上，幫助他們協調大型結構、**BOUNDED CONTEXT**邊界和其他一些跨團隊的技術問題。為了在這個過程中發揮作用，架構團隊必須把思考的重點放在應用程序的開發上。

在組織結構圖中，這樣的團隊看起來與傳統的架構團隊沒什麼分別，但實際上二者在每一項活動中都存在不同。架構團隊的成員是真正的開發協作者，他們與開發人員一起發現模式，與各個團隊一起通過反覆實驗進行精煉，並親自動手參與開發工作。

這種場景我曾經見到過幾次，項目最終會由一位架構師來領導，下面列出的大部分工作都會由他來完成。

### **17.5 制定戰略設計決策的6個要點**

## 決策必須傳達到整個團隊

顯然，如果不能確保團隊中的所有人都知道策略並去遵守它，那麼策略也就失去了作用。這個要求引導人們以架構團隊（具有正式的「權威」）為中心組織到一起，以便在整個項目中應用一致的規則。然而具有諷刺意味的是，那些脫離實際開發工作的架構師往往會被人們忽略或躲開。如果架構師沒有實踐經驗，又試圖把他們自己的規則強加於實際的應用程序，那麼他們所設計出來的模式就會不切實際，這時開發人員除了忽略他們之外別無選擇。

在一個溝通良好的項目中，應用開發團隊所產生的策略設計實際上會更有效地傳播到每個人。這樣策略將會實際發揮作用，而且具有權威性，因為它是通過集體智慧制定的決策。

無論開發什麼系統，都不要用管理層所授予的權力來強制地推行戰略決策，而應該更多地關注開發人員與策略之間的實際關係。

## 決策過程必須收集反饋意見

無論是建立組織原則、大型結構還是那些微妙的精煉，都需要真正理解項目的需求和領域概念。那些唯一具有這方面深層次知識的人就是應用程序開發團隊的成員。這解釋了為什麼架構團隊所創建的應用架構很少對項目產生幫助，儘管我們必須承認很多架構師都非常有才能。

與技術基礎設施和架構不同，戰略設計雖然影響到所有的開發工作，但是它本身並不需要編寫很多代碼。戰略設計真正需要的是應用開發團隊的參與。經驗豐富的架構師可以聽取來自各個團隊的想法，並促進總體解決方案的開發。

我曾經與一個技術架構團隊合作過，這個團隊把成員輪流派到使用其架構的各個應用開發團隊中。這種流動性使架構團隊親身體驗到

了開發人員所面臨的挑戰，同時也把如何應用框架的知識傳播給了開發人員。戰略設計同樣需要這種緊密的反饋循環。

### **計劃必須允許演變**

有效的軟件開發是一個高度動態的過程。如果最高層的決策已經固定下來，那麼當團隊需要對變更做出響應時，選擇就會更少。遵循 **EVOLVING ORDER** 這一原則，可以避免出現這個問題，因為它強調的是根據理解的不斷加深來調整大型結構。

當很多設計決策過早地固定下來時，開發團隊可能會束手束腳，失去解決問題的靈活性。因此，雖然那些為了協調項目而制定的原則可能很有價值，但原則必須能夠隨著項目開發生命週期的進行而完善和變化，而且不能過分限制應用程序開發人員的能力，因為開發工作本來就已經很難了。

有了積極的反饋之後，當構建應用程序的過程中遇到障礙或是出現了意想不到的機會時，創新就自然而然地湧現出來了。

### **架構團隊不必把所有最好、最聰明的人員都吸收進來**

架構層次的設計確實需要技術精湛的人員，而這樣的人員總是供不應求。項目經理往往會把那些最有技術天分的開發人員調到架構團隊和基礎設施團隊中，因為他們想要充分利用這些高級設計人員的技能。在項目經理看來，開發人員都希望提高自己的影響力，或是攻克那些「更有趣」的問題。而且，加入精英團隊本身也會贏得威望。

這樣往往會把那些技術能力較差的人留下來構建應用程序。但要想開發出優秀的應用程序，是需要設計技巧的，因此這樣安排註定會造成項目失敗。即使戰略團隊建立了一個很好的戰略設計，應用程序開發團隊也沒有能力把它實現出來。

相反，架構團隊幾乎從來不會把那些缺乏設計技巧但精通領域知識的開發人員吸納進來。戰略設計並不是一項純粹的技術任務，把那

些精通深層次領域知識的開發人員排除在外只會使架構師的工作更難進行。而且同樣也需要領域專家的參與。

所有應用程序團隊都應該有一些技術能力很強的設計人員，而且任何從事戰略設計的團隊也都必須具有領域知識，這兩者都是非常重要的。聘用更多高級設計人員是很有必要的，而且使架構團隊偶爾從事一下開發工作也會很有幫助。我相信有很多有用的方法，但任何有效的戰略團隊必須要與一個有效的應用程序團隊通力合作。

### **戰略設計需要遵守簡約和謙遜的原則**

任何設計工作都必須精煉而簡約，而戰略設計尤為需要簡約。即使是一個非常小的設計失誤也有可能會變成可怕的隱患。把架構團隊單分出來時要格外慎重，因為他們將更少感知他們為應用程序開發團隊所設置的障礙。同時，架構師對其主要職責的過度關注會使他們迷失方向。我就曾多次看到過這種情況，甚至我自己也犯過這種錯誤。有了一個好的想法後，又會引出另一個想法，想法太多最後就會得到一個過度設計的架構，這種體系結構反而起到了負面作用。

相反，我們必須嚴格地約束自己，從而使設計出來的組織原則和核心模型精簡到只包含那些能夠顯著提高設計清晰度的內容。事實上，幾乎任何事物都會對其他某個事物構成障礙，因此每個元素都必須是確實值得存在的。我們需要有一個謙遜的態度，才能認識到我們自己認為的最佳思路可能會妨礙其它人。

### **對象的職責要專一，而開發人員應該是多面手**

良好的對象設計的關鍵是為每個對象分配一個明確且專一的職責，並且把對像之間的互相依賴減至最小。人們有時會試圖讓團隊中的交流像軟件中的交互那樣整齊。其實在一個優秀的項目中，會有很多人參與其他人的事情。開發人員有時也處理框架，而架構師有時也

會編寫應用程序代碼。所有人員都可以互相交流。這看似混亂但卻行之有效。因此，應該讓對像職責專一，而讓開發人員成為多面手。

把戰略設計與其他設計區分開，是為了幫助澄清所涉及的工作，但必須指出：這兩種設計活動並不意味著有兩種人員。雖然基於深層模型創建柔性設計是一種高級設計活動，但細節問題也至關重要，因此戰略設計工作必須由接觸編碼工作的人來完成。戰略設計源自應用設計，然而戰略設計需要一個總體的開發活動視圖，這個視圖可能跨越多個團隊。人們總喜歡想出各種辦法把工作分得很細，以使得設計專家不必瞭解業務，而領域專家也不用知道技術。確實，一個人能學的知識是有限的，但過於專業化也會削弱領域驅動設計的力量。

### **17.5.1 技術框架同樣如此**

技術框架提供了基礎設施層，從而使應用程序不必自己去實現基礎服務，而且技術框架還能幫助把領域與其他關注點隔離開，因此它能夠極大地加速應用程序（包括領域層）的開發。但技術框架也是有風險的，那就是它會影響領域模型實現的表達能力，並妨礙領域模型的自由改變。甚至當框架設計人員並沒有特意去幹涉領域層或應用層的時候，情況同樣如此。

用於克服戰略設計缺點的原則同樣適用於技術架構。遵守演變、簡約等原則並且讓應用程序開發團隊參與進來，就能夠得到一組持續精化的服務和規則，這些服務和規則能夠真正有助於應用程序的開發，而不會妨礙開發。如果架構不按照這種方式來做，那麼它們要麼會抑制應用程序開發的創造力，要麼會被人們繞過去，從而導致應用程序為了能夠把開發進行下去而根本不使用架構。

有一種態度肯定會使框架流於失敗。

**不要編寫「傻瓜式」的框架**

在劃分團隊時，如果認為一些開發人員不夠聰明，無法勝任設計工作，而讓他們去做開發工作，那麼這種態度可能會導致失敗，因為他們低估了應用程序開發的難度。如果這些人在設計方面不夠聰明，就不應該讓他們來開發軟件。如果他們足夠聰明，那麼這種隔離只會造成障礙，使他們得不到所需的工具。

這種態度還會損害團隊之間的關係。我就曾經在這樣傲慢自大的團隊中感到疲憊不堪，於是每次談話都得向開發人員道歉，我自己也因為有這樣自大的同事而感到難堪（我恐怕永遠也無法改變這樣的團隊）。

注意，把無關的技術細節封裝起來與我所反對的這種「傻瓜式」的預打包完全不同。框架可以為開發人員提供有力的抽象和工具，使他們不用去做那麼多苦差事。有用的封裝和「傻瓜式」的預打包之間的區別很難用一種通用的方式描述出來，但只要問問框架設計人員他們對將要使用工具/框架/組件的那些人有什麼期望，就可以看出區別。如果設計人員對框架的用戶非常尊重，那麼他們的工作方向可能就是正確的。

### **17.5.2 注意總體規劃**

由Christopher Alexander領導的一群建築師（設計大樓的建築師）在建築和城市規劃領域中提倡「聚少成多地成長」（piecemeal growth）。他們非常好地解釋了總體規劃失敗的原因。

如果沒有某種規劃過程，那麼俄勒岡州大學的校園永遠不會像劍橋大學校園那樣龐大、和諧而井井有條。

總體規劃是解決這種難題的傳統方法。它試圖建立足夠多的指導方針，來保持整體環境的一致性，同時仍然為每幢建築保留自由度，並為適應局部需要預留下廣闊的空間。

.....將來這所大學的所有部分將構成一致的整體，因為它們只是被「插入」到總體設計的各個位置中。

.....實際上總體規劃會失敗，因為它只是建立了一種極權主義的秩序，而不是一種有機的秩序。它們過於生硬，因此不容易根據自然變化和不可預料的社會生活變化來做出調整。當這些變化發生時.....總體規劃就過時了，而且不再被人們遵守。即使人們遵守總體規劃.....它們也沒有足夠詳細地指定建築物之間的聯繫，人口規模、功能均衡等這些用來幫助每幢建築的局部行為和設計很好地符合整體環境的方面。

.....試圖駕馭這種總體規劃過程非常類似於在小孩的填色本上填充顏色.....這個過程最多也不過是得到一種極為平常的秩序。

.....因此，通過總體規劃是無法得到一種有機的秩序的，因為這個規劃既過於精確，又不夠細緻。它在整體上過於精確了，而在細節上又不夠細緻。

.....總體規劃的存在疏遠了用戶[因為，從根本上講]大部分重要決策已經確定下來了，因此社區成員對社區未來的建設幾乎沒有什麼影響了。

——摘自Oregon Experiment，pp.16-28[Alexander et al.1975]

Alexander和他的同事倡議由社區成員共同制定一組原則，並在「聚少成多地成長」的每次行動中都應用這些原則，這樣就會形成一種「有機秩序」，並且能夠根據環境變化作出調整。

---

[1].裡根把一個俄羅斯諺語翻譯成這句名言，這句話一語道破了雙邊事務的核心——這是連接上下文的又一個隱喻。

[2].分析癱瘓，analysis paralysis，指一個項目在大量的分析工作面前陷入困境。——譯者注

[3].自說經是印度佛經的一種，為佛陀自己之體驗，佛陀自身感興語，故有「自說經」此名。——編者注

[4].原文 bottom line是一個雙關語，其可以指財務盈虧結算，也可以指概覽。因此，作者才說可能會先注意到財務上的東西。——編者注

[5].當我在一次座談會中聽到Ward Cunningham舉防火牆這個示例後，終於明白了SYSTEM METAPHOR的意思。

[6].棉花彩（puffy paint）一種繪畫顏料，可以畫在紙、石頭、木頭、金屬等上，幹後用電吹風加熱即可產生浮凸效果。——譯者注

# 結束語

## 後記

雖然開發最前沿的項目並體驗有趣的思想和工具會帶來巨大的成就感，但我認為如果軟件得不到有效的應用，那麼一切都將成為空談。事實上，檢驗軟件成功與否的最有效的方法是讓它運行一段時間。近年來，我從自己經歷過的項目中總結出了一些經驗。

這裡我們來談一下其中5個項目，每個項目都認真嘗試了領域驅動設計，但它們並沒有系統地採用這種方法，當然也沒有在這個名頭下進行開發。這5個項目都完成了軟件交付工作，其中4個項目堅持採用模型驅動的設計方法，並得到了相應的設計結果，而有一個項目卻偏離了軌道。一些應用程序多年來一直在發展和改變，但有一個程序一直沒有進步，還有一個很早就結束了。

第1章中描述的PCB設計軟件的beta 版本在業內引起了一次很大的轟動，但遺憾的是，發起該項目的公司在它的營銷方面做得非常失敗，最終公司草草收場。少數一些保留了beta版副本的PCB工程師現在仍在使用該軟件。像所有缺乏支持的軟件一樣，它會被繼續使用下去，直到其中集成的某個程序發生重大改變為止。

第9章中介紹的貸款軟件在我提到的突破之後，經歷了3年波瀾不驚的發展。在此之後，該項目脫離出來，成為一家獨立的公司。在重組的過程中，從一開始就領導這個項目的經理被解聘了，一些核心開發人員也隨他一起離開。新的團隊有一套稍微不同的設計思想，他們

不是完全遵循對像建模。但保留了具有複雜行為的獨特的領域層，而且在他們的開發團隊中依舊非常重視領域知識。在新公司獨立運轉7年後，該軟件仍在不斷增加新的功能。它在業內是該領域領先的應用程序，正在為越來越多的客戶機構服務，也是公司最大的收入來源。



一片新種植的橄欖林

在領域驅動方法廣為流行之前，很多項目的軟件將創建得更快、更高效。但項目最終仍不免按傳統的套路發展，導致先前精煉的深層模型無法被充分利用，更談不上去增強它的能力了。可能我的期望過高了，但如果做不到這一點，項目就無法在長達數年的時間內為用戶提供穩定的價值。

我曾經與另一位開發人員結對做過一個項目，我們為客戶編寫一個實用工具，客戶用這個工具來開發他們的核心產品。所需的功能及功能組合相當複雜。我很喜歡這個項目的工作，我們也開發出了一個

具有**ABSTRACT CORE**的柔性設計。這個軟件交付以後，每個人涉及的工作也就結束了。由於項目交接之後就與我們無關了，交接過程顯得有些突兀，因此我估計那些用來支持元素組合的特性可能很難被客戶理解，而且有可能被替換為更典型的條件選擇邏輯。但這種情況並沒有馬上發生。當我們交付軟件的時候，程序包含一個完整的測試套件和一個精煉文檔。新的團隊成員用這個文檔來指導他們的工作。他們對這個軟件做了一番研究之後，很高興地發現我們的設計提供了各種可能性。當我在一年之後聽到他們的評論時，我知道我們的**UBIQUITOUS LANGUAGE**已經傳遞到了新團隊，而且這種語言仍然充滿活力並繼續發展。



7年之後

又一年過去了，我聽到一個完全不同的故事。團隊遇到了新的需求，開發人員們發現用原來的設計已經無法滿足這些新需求。他們不得不修改設計，這一改幾乎使原來的設計面目全非。在瞭解了一些細

節之後，我發現我們原來的模型用來解決這些問題時顯然十分蹩腳。往往就是在這個時候有可能產生一次突破，形成一個更深層的模型，特別是在這個例子中，開發人員已經積累了大量的深層領域知識和經驗。事實上，他們確實形成了新的理解，並最終根據這些理解對模型和設計進行了轉換。

他們小心翼翼地、委婉地告訴了我這件事情，我猜他們可能是擔心我在聽到如此多的先前工作被丟棄後會感到不滿。但是我對自己的設計並沒有這種守舊情結。一個成功的設計並不一定要永遠保持不變。如果把人們賴以工作的一個系統封閉起來，那麼它將會變為一項永久的、誰也不敢碰的遺留資產。深層模型可以使人們清楚地看懂它，並據此產生新的理解，而柔性設計可以促進後續的修改。他們提出了一個更深層的模型，這個模型更符合用戶關心的需求。他們的設計解決了實際問題。變更是軟件的固有性質，這個程序在擁有它的團隊的手中得到了繼續發展。

本書很多章節中都提到過運輸的例子，這個例子大體上是基於一家大型國際集裝箱運輸公司的項目。在早期，項目的領導者們採用了領域驅動的方法，但他們一直沒有建立一種支持該方法的開發文化。幾支具有不同設計技術水平和對像經驗的團隊分頭開始創建模塊，但他們之間的工作只是由團隊領導者之間的非正式合作和一個主要負責客戶事務的架構團隊來粗略地協調。我們確實開發出了一個合理的、深層的 **CORE DOMAIN** 模型，也有一個可使用的 **UBIQUITOUS LANGUAGE**。

但公司的文化非常不利於迭代開發，因此我們過了很長時間才形成了一個可用的內部版本。因此，問題到了後期才暴露出來，而此時修復的話就要冒很大的風險並且要付出高昂的代價。我們發現模型的某些方面會引起數據庫性能問題。反饋（無論是實現問題，還是模型

修改 ) 是 **MODEL-DRIVEN DESIGN** 的一個自然的部分，但那時我們感覺到自己已經在開發這條路上走得太遠了，以至於很難再修改模型的基本部分了。相反，我們對代碼做了修改，使它更有效，但代碼與模型的聯繫卻被削弱了。最初的版本也暴露出在技術基礎設施擴展方面的侷限性，這使管理層感到擔憂。項目組聘請了專家來修復基礎設施問題，項目恢復了開發。但實現與領域建模之間卻始終沒有形成一個閉環。

有幾個團隊交付了不錯的軟件——實現了複雜的功能，模型也表達得很清楚。而有些團隊交付的軟件卻很生硬，模型退化為數據結構(儘管他們保留了 **UBIQUITOUS LANGUAGE** 的痕跡)。可能使用 **CONTEXT MAP** 會有所幫助，因為各個團隊的開發結果之間沒有什麼必然聯繫。然而，用 **UBIQUITOUS LANGUAGE** 開發出來的 **CORE** 模型確實幫助團隊把各自的工作整合為一個系統。

雖然範圍縮小了，項目還是替換了幾個遺留系統。儘管大部分設計都不夠靈活，但整體設計還是通過一個共享的概念集凝聚到了一起。經過幾年之後，系統本身已經退化為一項遺留資產，但它仍在為全球業務提供全天候的服務。雖然成功團隊的影響漸漸擴大，但整個項目最後還是走到了盡頭，即便公司有著雄厚的財力。項目文化從來沒有真正採納過 **MODEL-DRIVEN DESIGN**。現在的新開發是在不同平臺上進行的，我們的工作只是間接影響他們，因為新開發人員需要遵從 ( **CONFORM** ) 他們的遺留系統。

在一些領域中，像運輸公司最初設定的那樣宏偉的目標是不可信的。更好的做法是開發小的、確保能夠交付的應用程序，並堅持用最簡單的設計來實現簡單的功能。這種保守的方法有它自己的用武之地，可以使項目範圍保持精簡，並且使項目具有快速響應的能力。但集成的、模型驅動的系統所提供的價值是那些拼湊起來的系統無法提

供的。但我們還有一種方法，那就是使用領域驅動設計構建深層模型和柔性設計，這樣，具有豐富功能的大型系統就能夠逐步增長。

最後我們來說一下**Evant**，這是一家開發庫存管理系統的公司，我曾在這家公司做過輔助支持的工作，也為公司已經很健壯的設計文化作出了一點貢獻。有些人把這個項目看作是極限編程的典型代表，但很少有人注意到它也廣泛應用了領域驅動設計。在這個項目中，模型被不斷精煉，並且用更柔性的設計表達出來。這個項目在2001年的「dot com」泡沫破裂以前一直在快速發展。不過隨後由於投資斷流，公司一度萎縮，軟件開發也基本上陷入休眠狀態，看起來離倒閉的日子不遠了。但在2002年夏季，**Evant**被一個世界排名前十的零售商看中。這家潛在的客戶喜歡**Evant**的產品，但產品需要改變設計，以便擴展系統來支持大量庫存規劃操作。這是**Evant**的最後機會。

雖然項目人員已萎縮至4人，但團隊仍然有實力。他們都具有精湛的技術，並且掌握了大量領域知識，而且其中一位成員還精通系統的擴展問題。他們有著非常高效的開發文化，代碼庫也實現了柔性設計，因此便於修改。在那個夏天，這4位開發人員經過艱巨的努力終於使系統能夠處理數以十億計的規劃元素以及數百個用戶。藉助於這些強大功能，**Evant**贏得了這家大客戶。不久之後，它被另一家公司收購，這家公司希望利用他們的軟件以及他們所展示出的能力來應對新的需求。

領域驅動的設計文化（以及極限編程文化）在公司過渡期間倖存下來並獲得了新生。現在，模型和設計仍在不斷發展，比兩年前我工作的時候要豐富和靈活得多。而且**Evant**團隊並沒有被收購它的公司同化，相反，在**Evant**團隊成員的帶動下，公司現有項目團隊正在向**Evant**團隊的開發文化轉變。這個故事還遠未結束。

沒有哪個項目會用到本書中介紹的所有技術。儘管如此，我們很容易通過幾個方面辨認出一個項目是否採用了領域驅動設計。標誌性的特徵是把「理解目標領域並將學到的知識融合到軟件中」當作首要任務。其他工作都以它為前提。團隊成員在項目中有意識地使用通用語言，並且不斷對語言進行精化。由於他們不斷地學習越來越多的領域知識，因此他們永遠不會滿足於現有領域模型的質量。他們把持續精化視作機會，把不適當的模型視作風險。他們知道，開發出高質量的、能夠清晰反映出領域模型的軟件並非易事，因此他們一絲不苟地運用設計技巧。他們也因為遇到障礙而跌倒過，但卻始終堅持自己的原則，百折不撓，繼續前進。

## 未來展望

氣候、生態系統和生物學以前被認為是雜亂無章的，是與物理或化學恰好相反的「軟」領域。然而，近來人們認識到這種「混亂」的表象實際上提出了一個具有深遠意義的技術挑戰，這意味著要去發現和理解這些非常複雜的現象之中蘊含的規律。當下，「複雜性」領域是眾多科學的前沿。雖然有才能的軟件工程師通常都認為純粹的技術任務是最有趣、最有挑戰性的，但領域驅動設計展現了一個同樣富有挑戰性（甚至具有更大挑戰性）的新領域。業務軟件大可不必是拼湊而成的雜亂系統。與複雜的領域「搏鬥」，把它轉化為可理解的軟件設計，這對於優秀的技術人員來說是一項激動人心的挑戰。

由外行創建複雜軟件的時代還遠未到來。雖然掌握了一些初級技術的眾多編程人員可以開發出特定種類的軟件，但他們絕對無法開發出能在危急關頭拯救公司的軟件。真正需要做的是：工具構建人員必須確保他們開發出的工具能夠提高那些優秀軟件開發人員的能力和工作效率。真正需要做的是：更加透徹地研究領域模型，並在可運行的

軟件中把它們表示出來。我非常希望能夠嘗試出於這個目的而設計的新工具和技術。

然而，儘管好的工具很有價值，但我們不能把注意力都放在工具上而忽視掉一個基本事實——創建好的軟件是一項需要學習和思考的活動。建模需要想像力和自律。好的工具能夠幫助我們思考或避免分心。企圖自動實現一些只有通過思考才能完成的任務是不切實際的，如果這樣做的話，產生的效果只會適得其反。

利用已有的工具和技術，我們可以開發出比當今大多數項目更有價值的系統。我們可以編寫優秀的軟件，這樣的軟件使用起來是一種樂趣，它在擴展的時候不會對我們構成限制，反而會為我們創造新的機會，並且會不斷為其使用者提供價值。

## 附錄

我的第一部「靚車」是一部已經使用了8年的標緻 ( Peugeot )，這是我大學畢業後不久別人送給我的。有人把這款車稱為「法國的梅賽德斯」，這輛車製造精良，駕駛起來非常舒適，而且一直也沒出過什麼毛病。但到我手裡時，它已經有一些年頭了，因此到了該出毛病的時候，而且需要更多保養。

標緻是一家老牌公司，數十年來一直沿著自己的發展路線前進。它有自己的機械術語、設計和特殊風格，甚至零部件的拆卸有時也不是標準的。這導致標緻車只有標緻公司的專家才能修理，維修費用對於一個剛畢業的、沒多少收入的學生來說是一個潛在的問題。

在一次平常的養護中，我把車開到當地一家機修工那裡檢查漏油問題。他檢查了底盤，告訴我油是「從距離車尾大概2/3位鎂處的一個小箱子裡漏出來的，這個小箱子看起來與前後輪之間的制動力分配有關」。隨後他拒絕了為我修車，建議我去找50英里之外的經銷商。任何機修工都可以修理福特或本田汽車，這就是為什麼這些車開起來更方便而且維修費用也較低的緣故，儘管它們在機械製造上與標緻汽車同樣複雜。

雖然我確實喜歡這部車，但我再也不想擁有一部古怪的車了。有一天車被檢出了一個問題，而對它的維修費用相當昂貴。我實在是受不了這輛標緻了，於是就把車送給了當地一家接受汽車捐贈的慈善機

構。然後我買了一輛舊的本田思域，買這輛車的錢跟修那輛標緻的費用差不多。

領域開發缺乏標準的設計元素，因此每個領域模型和對應的實現都很奇怪且難以理解。此外，每個團隊都不得不重新發明輪子（或齒輪，或雨刷）。在面向對像設計中，所有的一切都是對像、引用或消息，這些都是有用的抽象。但這並不足以約束領域設計的選擇範圍，也無法支持對領域模型進行簡練的討論。

「一切都是對像」這個觀點就好像木匠或建築師把房屋歸納為「一切都是房間」一樣。房間有大有小，有電源插座和水池的大房間可以做飯，也有樓上用來睡覺用的小房間。描述一棟普通的房子可能需要許多頁紙的篇幅。建造和使用房屋的人意識到房屋遵循著一些模式，這些模式有具體的名稱，如「廚房」。這種語言使人們能夠對房屋設計進行簡練的討論。

此外，並非所有的功能組合都是實用的。為什麼不建一個既能供我們洗澡又能供我們睡覺的房間呢？這樣不是很方便嗎？但長期的經驗已經形成了習慣，我們把「浴室」和「臥室」分開。畢竟，洗浴設施往往可以與更多的人共用，而臥室則不然。浴室需要最大限度地保證個人隱私，甚至那些共處一個臥室的人也不能未經允許而同時使用這個浴室。而且，浴室需要裝備特殊的、昂貴的設施。浴缸和衛生間通常設在一個房間裡，因為它們需要相同的基礎設施（水和排水管道），而且二者都需要保護隱私。

另一類需要安裝特殊設施的房間是我們用來做飯的房間，也稱為「廚房」。與浴室相比，廚房沒有隱私需求。由於廚房的設計同樣很昂貴，因此通常一所房屋（即使是很大的房屋）只有一個廚房。這種單一性也促使我們形成了準備全家共用的食物和共同用餐的習慣。

當我說我需要一所有三間臥室、兩間浴室和一個開放式廚房的房屋時，我把大量的信息打包到一句很短的話裡，並且避免了很多愚蠢的錯誤，如把抽水馬桶放在冰箱旁邊。

在每個設計領域（如汽車、皮划艇或軟件）中，我們都會把設計建立在已有模式上，在已有主題範圍內即興發揮。有時我們必須發明一些全新的東西。但是，以標準的模式元素為基礎，可以避免把精力浪費在那些已經存在瞭解決方案的問題上，從而集中精力關注我們的特殊問題。此外，根據傳統的模式來建立自己的設計可以避免產生過於特殊的、很難交流的設計。

雖然軟件設計領域不像其他設計領域那麼成熟，各種情況變化多端，無法像汽車零部件或房屋那樣具體地應用模式，但不管怎樣都不能僅僅停留在「一切都是對像」這種層次上，至少要分清「螺栓」和「彈簧」。

20世紀70年代，一群由Christopher Alexander[Alexander et al.1977]領導的建築師提出了一種共享和標準化設計思想的理念。他們的「模式語言」把一些經過事實檢驗的解決方案組合在一起，用來解決一些公共的問題（這些問題比「廚房」要複雜多了，可能會使Alexander的一些讀者望而卻步）。他們的目的是讓房屋的建造者和使用者用這種語言進行交流，並且在這些模式的指導下建造出優美的建築物，為房屋的使用者提供實用的功能，並讓他們產生良好的體驗。

無論建築師們是怎樣想的，這種模式語言已經對軟件設計產生了重大的影響。在20世紀90年代，軟件模式被應用在很多方面，並且獲得了一些成功，特別是在詳細設計[Gamma et al]和技術架構[Buschmann et al.1996]方面獲得了顯著成功。近來，模式被用於描述基本的面向對像設計技巧[Larman 1998]以及企業架構[Fowler

2002,Alur et al.2001]。模式語言現在已成為組織軟件設計思想的主流技術。

模式名稱應該作為團隊語言中的術語來使用，我在本書中就是這樣使用它們的。當在討論中出現模式名時，一律採用了英文小體大寫格式，以便於區分。

以下是本書討論模式時所採用的格式。有的模式與這個基本格式略有不同，因為我喜歡具體問題具體對待，而且我認為可讀性比嚴格的結構更為重要.....

### **模式：模式名稱**

[概念的說明。有時用一種形象的比喻或引起讀者興趣的文字。]

[上下文。對概念與其他模式相關性的簡單解釋。有些情況下是一段簡單的模式概述。]

但是，本書中的大部分上下文討論都是在每章的引言以及其他敘述段落中給出的，而不是在模式中給出的。

[問題討論]

### **問題小結**

通過解決問題的討論形成一個解決方案。

因此：

### **解決方案小結。**

結果。實現考慮。示例。

結論。簡單解釋這種模式如何引出後續模式。

[實現問題的討論。在Alexander最初的形式中，這個討論應該放在一個段落內，描述問題的解決，本書一般是按照Alexander的方法來組織的。但有些模式需要較長的實現討論。為了保證核心模式討論的緊湊，我把這些較長篇幅的實現討論移到了模式討論的後面。]

此外，較長的示例，特別是涉及多個模式組合的示例，也放在模式之外進行討論。】

# 術語表

以下是本書中所選用的術語、模式名和其他概念的簡要定義。

**AGGREGATE** ( 聚合 ) —— 聚合就是一組相關對象的集合，我們把聚合作為數據修改的單元。外部對像只能引用聚合中的一個成員，我們把它稱為根。在聚合的邊界之內應用一組一致的規則。

**分析模式** ( *analysis pattern* ) —— 分析模式是用來表示業務建模中的常見構造的概念集合。它可能只與一個領域有關，也可能跨多個領域 [Fowler 1997, p.8] 。

**ASSERTION** ( 斷言 ) —— 斷言是對程序在某個時刻的正確狀態的聲明，它與如何達到這個狀態無關。通常，斷言指定了一個操作的結果或者一個設計元素的固定規則。

**BOUNDED CONTEXT** ( 限界上下文 ) —— 特定模型的限界應用。限界上下文使團隊所有成員能夠明確地知道什麼必須保持一致，什麼必須獨立開發。

**客戶** ( *client* ) —— 一個程序元素，它調用正在設計的元素，使用其功能。

**內聚** ( *cohesion* ) —— 邏輯上的協定和依賴。

**命令**，也稱為修改器命令 ( *command/modifier* ) —— 使系統發生改變的操作 ( 例如，設置變量 ) 。它是一種有意產生副作用的操作。

**CONCEPTUAL CONTOUR** ( 概念輪廓 ) —— 領域本身的基本一致性，如果它能夠在模型中反映出來的話，則有助於使設計更自然地適

應變化。

**上下文 ( context )**——一個單詞或句子出現的環境，它決定了其含義。參見 **BOUNDED CONTEXT**。

**CONTEXT MAP ( 上下文圖 )**——項目所涉及的限界上下文以及它們與模型之間的關係的一種表示。

**CORE DOMAIN ( 核心領域 )**——模型的獨特部分，是用戶的核心目標，它使得應用程序與眾不同並且有價值。

**聲明式設計 ( declarative design )**——一種編程形式，由精確的屬性描述對軟件進行實際的控制。它是一種可執行的規格。

**深層模型 ( deep model )**——領域專家們最關心的問題以及與這些問題最相關的知識的清晰表示。深層模型不停留在領域的表層和粗淺的理解上。

**設計模式 ( design pattern )**——設計模式是對一些互相交互的對象和類的描述，我們通過定製這些對像和類來解決特定上下文中的一般設計問題[Gamma et al.1995,p.3]。

**精煉 ( distillation )**——精煉是把一堆混雜在一起的組件分開的過程，從中提取出最重要的內容，使得它更有價值，也更有用。在軟件設計中，精煉就是對模型中的關鍵方面進行抽象，或者是對大系統進行劃分，從而把核心領域提取出來。

**領域 ( domain )**——知識、影響或活動的範圍。

**領域專家 ( domain expert )**——軟件項目的成員之一，精通的是軟件的應用領域而不是軟件開發。並非軟件的任何使用者都是領域專家，領域專家需要具備深厚的專業知識。

**領域層 ( domain layer )**——在分層架構中負責領域邏輯的那部分設計和實現。領域層是在軟件中用來表示領域模型的地方。

**ENTITY ( 實體 )** —— 一種對象，它不是由屬性來定義的，而是通過一連串的連續事件和標識定義的。

**FACTORY ( 工廠 )** —— 一種封裝機制，把複雜的創建邏輯封裝起來，並為客戶抽象出所創建的對象的類型。

**函數 ( function )** —— 一種只計算和返回結果而沒有副作用的操作。

**不可變的 ( immutable )** —— 在創建後永遠不發生狀態改變的一種特性。

**隱式概念 ( implicit concept )** —— 一種為了理解模型和設計的意義而必不可少的概念，但它從未被提及。

**INTENTION-REVEALING INTERFACE ( 釋意接口 )** —— 類、方法和其他元素的名稱既表達了初始開發人員創建它們的目的，也反映出了它們將會為客戶開發人員帶來的價值。

**固定規則 ( invariant )** —— 一種為某些設計元素做出的斷言，除了一些特殊的臨時情況（例如，方法執行的中間，或者尚未提交的數據庫事務的中間）以外，它必須一直保持為真。

**迭代 ( iteration )** —— 程序反覆進行小幅改進的過程。也表示這個過程中的一個步驟。

**大型結構 ( large-scale structure )** —— 一組高層的概念和/或規則，它為整個系統建立了一種設計模式。它使人們能夠從大的角度來討論和理解系統。

**LAYERED ARCHITECTURE ( 分層架構 )** —— 一種用於分離軟件系統關注點的技術，它把領域層與其他層分開。

**生命週期 ( life cycle )** —— 一個對像從創建到刪除中間所經歷的一個狀態序列，通常具有一些約束，以確保從一種狀態變為另一種狀

態時的完整性。它可能包括 ENTITY 在不同的系統和 BOUNDED CONTEXT 之間的遷移。

模型 ( model ) —— 一個抽象的系統，描述了領域的所選方面，可用於解決與該領域有關的問題。

MODEL-DRIVEN DESIGN ( 模型驅動的設計 ) —— 軟件元素的某個子集嚴格對應於模型的元素。也代表一種合作開發模型和實現以便互相保持一致的過程。

建模範式 ( modeling paradigm ) —— 一種從領域中提取概念的特殊方式，與工具結合起來使用，為這些概念創建軟件類比。(例如，面向對像編程和邏輯編程。)

REPOSITORY ( 存儲庫 ) —— 一種把存儲、檢索和搜索行為封裝起來的機制，它類似於一個對像集合。

職責 ( responsibility ) —— 執行任務或掌握信息的責任 [Wirfs-Brock et al.2003,p.3] 。

SERVICE ( 服務 ) —— 一種作為接口提供的操作，它在模型中是獨立的，沒有封裝的狀態。

副作用 ( side effect ) —— 由一個操作產生的任何可觀測到的狀態改變，不管這個操作是有意的還是無意的 ( 即使是一個有意的更新操作 ) 。

SIDE-EFFECT-FREE FUNCTION ( 無副作用的函數 ) —— 參見 [FUNCTION] 。

STANDALONE CLASS ( 孤立的類 ) —— 無需引用任何其他對像 ( 系統的基本類型和基礎庫除外 ) 就能夠理解和測試的類。

無狀態 ( stateless ) —— 設計元素的一種屬性，客戶在使用任何無狀態的操作時，都不需要關心它的歷史。無狀態的元素可以使用甚

至修改全局信息（即它可以產生副作用），但它不保存影響其行為的私有狀態。

**戰略設計 ( strategic design )** ——一種針對系統整體的建模和設計決策。這樣的決策影響整個項目，而且必須由團隊來制定。

**柔性設計 ( supple design )** ——柔性設計使客戶開發人員能夠掌握並運用深層模型所蘊含的潛力來開發出清晰、靈活且健壯的實現，並得到預期結果。同樣重要的是，利用這個深層模型，開發人員可以輕鬆地實現並調整設計，從而很容易地把他們的新知識加入到設計中。

**UBIQUITOUS LANGUAGE ( 通用語言 )** ——圍繞領域模型建立的一種語言，團隊所有成員都使用這種語言把團隊的所有活動與軟件聯繫起來。

**統一 ( unification )** ——模型的內部一致性，使得每個術語都沒有歧義且沒有規則衝突。

**VALUE OBJECT ( 值對像 )** ——一種描述了某種特徵或屬性但沒有概念標識的對象。

**WHOLE VALUE ( 完整值 )** ——對單一、完整的概念進行建模的對象。

## 參考文獻

- Alexander,C.,M.Silverstein,S.Angel,S.Ishikawa, and D.Abrams.1975.The Oregon Experiment.Oxford University Press.
- Alexander,C.,S.Ishikawa, and M.Silverstein.1977.A Pattern Language:Towns,Buildings,Construction.Oxford University Press.
- Alur,D.,J.Crupi, and D.Malks.2001.Core J2EE Patterns.Sun Microsystems Press.
- Beck,K.1997.Smalltalk Best Practice Patterns.Prentice Hall PTR.
- .2000.Extreme Programming Explained:Embrace Change.Addison-Wesley.
- .2003.Test-Driven Development:By Example.Addison-Wesley.
- Buschmann,F.,R.Meunier,H.Rohnert,P.Sommerlad, and M.Stal.1996.Pattern-Oriented Software Architecture:A System of Patterns.Wiley.
- Cockburn,A.1998.Surviving Object-Oriented Projects:A Manager's Guide.Addison-Wesley.
- 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.
- .1999.Refactoring:Improving the Design of Existing Code.Addison-Wesley.
- .2003.Patterns of Enterprise Application Architecture.Addison-Wesley.
- Gamma,E.,R.Helm,R.Johnson, and J.Vlissides.1995.Design Patterns.Addison-Wesley.
- 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.
- Merriam-Webster.1993.Merriam-Webster 』 s Collegiate Dictionary.Tenth edition.Merriam-Webster.
- Meyer,B.1988.Object-oriented Software Construction.Prentice Hall PTR.
- 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.
- 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.

## 圖片說明

本書中的所有圖片均已得到使用許可。

**Richard A.Paselk,Humboldt State University**

星盤圖（第3章，P30）

**c Royalty-Free/Corbis**

指印（第5章，P56），加油站（第5章，P67），Auto工廠（第6章，P89），圖書管理員（第6章，P97）

**Martine Jousset**

葡萄（第6章，P81），新種植的和長大後的橄欖林（結束語，P346和P347）

**Biophoto Associates/Photo Researchers,Inc.**

電子顯微鏡下的顫藻細胞（第14章，P235）

**Ross J.Venables**

劃手（一群和單個）（第14章，P239和P260）

**Photodisc Green/Getty Images**

賽跑者（第14章，P250），兒童（第14章，P253）

**U.S.National Oceanic and Atmospheric Administration**

中國長城（第14章，P255）

**c 2003 NAMES Project Foundation,Atlanta,Georgia.**

Photographer Paul Margolies. [www.aidsquilt.org](http://www.aidsquilt.org)

艾滋拼被（第16章，P303）

# 索引

索引中的頁碼為英文原書頁碼，與本書邊欄頁碼一致。

## ■A

ABSTRACT CORE,435-437  
ADAPTERS,367  
AGGREGATES  
    definition ( 定義 ) ,126-127  
    examples ( 例子 ) ,130-135,170-171,177-179  
    invariants ( 固定規則 ) ,128-129  
    local vs.global identity ( 本地標識與全局標識 ) ,127  
    overview ( 概述 ) ,125-129  
    ownership relationships ( 所屬關係 ) ,126  
    Agile design ( 敏捷設計 )  
    Distillation ( 精煉 ) ,483  
MODULES,111  
    reducing dependencies ( 減少依賴 ) ,265,435-437,463  
    supple design ( 柔性設計 ) ,243-244,260-264  
    AIDS Memorial Quilt Project ( 艾滋病紀念拼被 ) ,479  
    Analysis models ( 分析模型 ) ,47-49  
    Analysis patterns ( 分析模式 ) .參見design patterns.  
    concept integrity ( 概念完整性 ) ,306-307

definition ( 定義 ) ,293  
example ( 示例 ) ,295-306  
overview ( 概述 ) ,294  
UBIQUITOUS LANGUAGE,306-307  
ANTICORRUPTION LAYER  
ADAPTERS,367  
considerations ( 考慮因素 ) ,368-369  
example ( 示例 ) ,369-370  
FAADES,366-367  
interface design ( 接口設計 ) ,366-369  
overview ( 概述 ) ,364-366  
relationships with external systems ( 與外部系統的關係 ) ,384-385  
Application layer ( 應用層 ) ,70,76-79  
Architectural frameworks ( 架構框架 ) ,70,74,156-157,271-272,495-496  
ASSERTIONS,255-259  
Associations ( 關聯 )  
bidirectional ( 雙向的 ) ,102-103  
example ( 示例 ) ,169-170  
for practical design ( 為了獲得更實用的設計 ) ,82-88  
VALUE OBJECTS,102-103  
Astrolabe ( 星盤 ) ,47  
Awkwardness ( 不足之處 ) ,concept analysis ( 概念分析 ) ,210-216

■B

Bidirectional associations ( 雙向關聯 ) ,102-103  
Blind men and the elephant ( 盲人與象 ) ,377-381  
Bookmark anecdote ( 書籤趣聞 ) ,57-59  
BOUNDED CONTEXT. 參見CONTEXT MAP.  
code reuse ( 代碼重用 ) ,344  
CONTINUOUS INTEGRATION,341-343  
defining ( 定義 ) ,382  
duplicate concepts ( 重複的概念 ) ,339-340  
example ( 示例 ) ,337-340  
false cognates ( 假同源 ) ,339-340  
large-scale structure ( 大型結構 ) ,485-488  
overview ( 概述 ) ,335-337  
relationships ( 關係 ) ,352-353  
splinters ( 不一致 ) ,339-340  
testing boundaries ( 測試邊界 ) ,351  
translation layers ( 轉換層 ) ,374. 參見ANTICORRUPTION LAYER;  
PUBLISHED LANGUAGE.  
vs MODULES,335  
Brainstorming ( 頭腦風暴 ) ,7-13,207-216,219  
Breakthroughs ( 突破 ) ,193-200,202-203  
Business logic,in user interface layer ( 業務邏輯 · 位於用戶界面層 ) ,77  
Business rules ( 業務規則 ) ,17,225  
**■C**  
Callbacks ( 回調模式 ) ,73

Cargo shipping examples ( 貨運示例 ) . 參見 examples,cargo shipping.

Changing the design ( 改變設計 ) . 參見 refactoring.

Chemical warehouse packer example ( 化學品倉庫打包示例 ) ,235-241

Chemistry example ( 化學示例 ) ,377

Cleese,John,5

CLOSURE OF OPERATIONS,268-270

Code as documentation ( 作為文檔的代碼 ) ,40

Code reuse ( 代碼重用 )

BOUNDED CONTEXT,344

GENERIC SUBDOMAINS,412-413

reusing prior art ( 借鑒先前的經驗 ) ,323-324

Cohesion ( 內聚 ) ,MODULES,109-110,113

COHESIVE MECHANISMS

and declarative style ( 聲明式風格 ) ,426-427

example ( 示例 ) ,425-427

overview ( 概述 ) ,422-425

vs.GENERIC SUBDOMAINS,425

Common language ( 通用語言 ) . 參見 PUBLISHED LANGUAGE;UBIQUITOUS LANGUAGE.

Communication ( 溝通 ) ,speech ( 講話 ) . 參見 UBIQUITOUS LANGUAGE.

Communication ( 溝通 ) ,written ( 書面的 ) . 參見 documents;UML (Unified Modeling Language); UBIQUITOUS LANGUAGE.

Complexity ( 複雜性 ), reducing ( 減少 ). 參見 distillation; large-scale structure ( 大型結構 ) ; LAYERED ARCHITECTURE; supple design ( 柔性設計 ) .

COMPOSITE pattern,315-320

Composite SPECIFICATION,273-282

Concept analysis ( 概念分析 ) . 參見 analysis patterns; examples, concept analysis.

Awkwardness ( 不足之處 ) ,210-216

Contradictions ( 矛盾之處 ) ,216-217

explicit constraints ( 顯式約束 ) ,220-222

language of the domain experts ( 領域專家的語言 ) ,206-207

missing concepts ( 丢失的概念 ) ,207-210

processes as domain objects ( 作為領域對象的過程 ) ,222-223

researching existing resources ( 查詢現有資源 ) ,217-219

SPECIFICATION,223

trial and error ( 嘗試和出錯 ) ,219

CONCEPTUAL CONTOURS,260-264

Conceptual layers ( 概念層 ) , 參見 LAYERED ARCHITECTURE; RESPONSIBILITY LAYERS

Configuring ( 配置 ) SPECIFICATION,226-227

CONFORMIST,361-363,384-385

Constructors,141-142,174-175. 參見 FACTORIES.

CONTEXT MAP. 參見 BOUNDED CONTEXT.

Example ( 示例 ) ,346-351

organizing and documenting ( 組織和文檔化 ) ,351-352

overview ( 概述 ) ,344-346

vs.large-scale structure ( 大型結構 ),446,485-488  
CONTEXT MAP,choosing a strategy ( 選擇一種策略 )  
ANTICORRUPTION LAYER,384-385  
CONFORMIST,384-385  
CUSTOMER/SUPPLIER DEVELOPMENT TEAMS,356-360  
defining BOUNDED CONTEXT,382  
deployment ( 部署 ),387  
external systems ( 外部系統 ),383-385  
integration ( 集成 ),384-385  
merging ( 合併 ) OPEN HOST SERVICE and PUBLISHED LANGUAGE,394-396  
merging ( 合併 ) SEPARATE WAYS and SHARED KERNEL,389-391  
merging ( 合併 ) SHARED KERNEL and CONTINUOUS INTEGRATION,391-393  
packaging ( 打包 ),387  
phasing out legacy systems ( 逐步淘汰遺留系統 ),393-394 for a project in progress ( 對於一個正在開發的項目 ),388-389  
SEPARATE WAYS,384-385  
SHARED KERNEL,354-355  
specialized terminologies ( 專用術語 ),386-387  
system under design ( 正在設計的系統 ),385-386  
team context ( 團隊上下文 ),382  
trade-offs ( 折中 · 權衡 ),387  
transformations ( 轉換 ),389  
transforming boundaries ( 轉換邊界 ),382-383

Context principle ( 上下文主題 ),328-329. 參見 BOUNDED CONTEXT; CONTEXT MAP.

CONTINUOUS INTEGRATION,341-343,391-393. 參見 integration.

Continuous learning ( 持續學習 ),15-16

Contradictions ( 矛盾之處 ),concept analysis ( 概念分析 ),216-217

CORE DOMAIN

DOMAIN VISION STATEMENT,415-416

flagging key elements ( 表明核心元素 ),419-420

MECHANISMS,425

Overview ( 概述 ),400-405

Costs of architecture dictated MODULES ( 支配MODULE的架構的代價 ),114-115

Coupling ( 耦合 ) MODULES,109-110

Customer-focused teams ( 以客戶為中心的團隊 ),492

CUSTOMER/SUPPLIER,356-360

## ■D

Database tuning ( 優化數據庫 ),example ( 示例 ),102

Declarative design ( 聲明式設計 ),270-272

Declarative style of design ( 聲明式設計風格 ),273-282,426-427

Decoupling from the client ( 與客戶解耦 ),156

Deep models ( 深層模型 )

distillation ( 精煉 ),436-437

overview ( 概述 ),20-21

refactoring ( 重構 ),189-191

Deployment ( 部署 ) ,387. 參見 MODULES.

Design changes ( 設計變更 ) . 參見 refactoring.

Design patterns ( 設計模式 ) . 參見 analysis patterns.

COMPOSITE,315-320

FLYWEIGHT,320

overview ( 概述 ) ,309-310

STRATEGY,311-314

vs.domain patterns ( 領域模式 ) ,309

Development teams ( 開發團隊 ) . 參見 teams.

Diagrams ( 圖表 ) . 參見 documents; UML(Unified Modeling Language).

Discovery ( 發現 ) ,191-192

Distillation ( 精煉 ) . 參見 examples, distillation.ABSTRACT CORE,435-437

deep models ( 深層模型 ) ,436-437

DOMAIN VISION STATEMENT,415-416

Encapsulation ( 封裝 ) ,422-427

HIGHLIGHTED CORE,417-421

INTENTION-REVEALING INTERFACES,422-427

large-scale structure ( 大型結構 ) ,483,488-489

overview ( 概述 ) ,397-399

PCB design anecdote ( PCB 設計趣聞 ) ,7-13

Polymorphism ( 多態 ) ,435-437

refactoring targets ( 重構目標 ) ,437

role in design ( 設計中的角色 ) ,329

SEGREGATED CORE,428-434

separating CORE concepts ( 分離 CORE 概念 ) ,428-434  
Distillation ( 精煉 ) ,COHESIVE MECHANISMS and declarative style  
( 和聲明式風格 ) ,426-427  
    overview ( 概述 ) ,422-425  
    vs.GENERIC SUBDOMAINS,425  
    Distillation ( 精煉 ) ,CORE DOMAIN  
    DOMAIN VISION STATEMENT,415-416  
    flagging key elements ( 表明核心元素 ) ,419-420  
    MECHANISMS,425  
    overview ( 概述 ) ,400-405  
    Distillation,GENERIC SUBDOMAINS  
    adapting a published design ( 採用公開發佈的設計 ) ,408 in-  
    house solution ( 內部解決方案 ) ,409-410  
    off-the-shelf solutions ( 現成的解決方案 ) ,407  
    outsourcing ( 外包 ) ,408-409  
    overview ( 概述 ) ,406  
    reusability ( 可重用性 ) ,412-413  
    risk management ( 風險管理 ) ,413-414  
    vs.COHESIVE MECHANISMS,425  
    Distillation document ( 精煉文檔 ) ,418-419,420-421  
    Documents ( 文檔 )  
        code as documentation ( 作為文檔的代碼 ) ,40  
        distillation document ( 精煉文檔 ) ,418-419,420-421  
    DOMAIN VISION STATEMENT,415-416  
    explanatory models ( 解釋性模型 ) ,41-43  
    keeping current ( 保持最新狀態 ) ,38-40

in project activities ( 深入各種項目活動之中 ) ,39-40  
purpose of ( 目的在於 ) ,37-40 validity of ( 正當性 ) ,38-40  
UBIQUITOUS LANGUAGE,39-40  
    Domain experts ( 領域專家 )  
    gathering requirements from ( 從 „處蒐集信息 ) .參見  
    concept analysis; knowledge crunching.  
    language of ( „的知識 ) ,206-207.參見UBIQUITOUS LANGUAGE.  
    Domain layer ( 領域層 ) ,70,75-79  
    Domain objects ( 領域對像 ) ,life cycle ( 生命週期 ) ,123-124.參  
見AGGREGATES; FACTORIES; REPOSITORIES.  
    Domain patterns vs.design pattern ( 領域模式與設計模式 ) ,309  
DOMAIN VISION STATEMENT,415-416  
    Domain-specific language ( 領域特定語言 ) ,272-273  
    Duplicate concepts ( 重複的概念 ) ,339-340  
■E  
    Elephant and the blind men ( 盲人與象 ) ,377-381  
    Encapsulation ( 封裝 ) .參見 FACTORIES.COHESIVE  
MECHANISMS,422-427  
    INTENTION-REVEALING INTERFACES,246  
    REPOSITORIES,154  
    ENTITIES. 參見 associations; SERVICES; VALUE  
OBJECTS.automatic IDs ( 自動生成的ID ) ,95-96  
    clustering ( 聚集 ) .參見AGGREGATES.establishing identity ( 定  
義標識 ) ,90-93  
    example ( 示例 ) ,167-168  
    ID uniqueness ( ID唯一性 ) ,96

identifying attributes ( 標識屬性 ) ,94-96  
identity tracking ( 實體跟蹤 ) ,94-96  
modeling ( 建模 ) ,93-94  
referencing with ( 參考 ) VALUE OBJECTS,98-99  
vs.Java entity beans ( Java 實體bean ) ,91  
Event ( Event 公司 ) ,504-505  
EVOLVING ORDER,444-446,491  
Examples ( 示例 )  
AGGREGATES,130-135  
analysis patterns ( 分析模式 ) ,295-306  
ASSERTIONS,256-259  
breakthroughs ( 突破 ) ,202-203  
chemical warehouse packer ( 化學倉庫打包 ) ,235-241  
chemistry ( 化學 ) ,PUBLISHED LANGUAGE,377  
CLOSURE OF OPERATIONS,269-270  
COHESIVE MECHANISMS,425-427  
composite SPECIFICATION,278-282  
CONCEPTUAL CONTOURS,260-264  
constructors ( 構造函數 ) ,174-175  
Event ( Event 公司 ) ,504-505  
explanatory models ( 解釋性模型 ) ,41-43  
extracting hidden concepts ( 提取隱藏的概念 ) ,17-20  
insurance project ( 保險項目 ) ,372-373  
integration with other systems ( 與其他系統集成 ) ,372-373  
INTENTION-REVEALING INTERFACES,423-424  
introducing new features ( 引入新特性 ) ,181-185

inventory management ( 庫存管理 ) ,504-505  
investment banking ( 投資銀行業務 ) ,211-215  
KNOWLEDGE LEVEL,466-474  
LAYERED ARCHITECTURE,71-72  
MODEL-DRIVEN DESIGN,52-57  
MODULES,111-112  
multiple teams ( 多個團隊 ) ,358-360  
online banking ( 網上銀行 ) ,71-72  
organization chart ( 組織結構圖 ) ,423-427  
package coding in Java ( Java中的包編碼 ) ,111-112  
paint-mixing application ( 調漆應用程序 ) ,247-249,252-254,256-259  
payroll and pension ( 工資和養老金系統 ) ,466-474  
PLUGGABLE COMPONENT FRAMEWORK,475-479  
procedural languages ( 過程語言 ) ,52-57  
prototypes ( 原型 ) ,238-241  
PUBLISHED LANGUAGE,377  
purchase order integrity ( 採購訂單的完整性 ) ,130-135  
refactoring ( 重構 ) ,247-249  
RESPONSIBILITY LAYERS,452-460  
selecting from Collections ( 從集合中選擇子集 ) ,269-270  
SEMATECH CIM framework ( SEMATECH CIM框架 ) ,476-479  
SIDE-EFFECT-FREE FUNCTIONS,252-254,285-286  
SPECIFICATION,235-241  
supple design ( 柔性設計 ) ,247-249  
time zones ( 時區 ) ,410-412

tuning a database ( 優化數據庫 ),102  
VALUE OBJECTS,102  
Examples,cargo shipping ( 貨運示例 )  
AGGREGATES,170-171,177-179  
allocation checking ( 配額檢查 ),181-185  
ANTICORRUPTION LAYER,369-370  
associations ( 關聯 ),169-170  
automatic routing ( 自動安排路線 ),346-351  
booking ( 預定 )  
BOUNDED CONTEXT,337-340  
extracting hidden concepts ( 提取隱藏的概念 ),17-20  
legacy application ( 遺留應用程序 ),369-370  
overbooking ( 超訂 ),18-19,222  
vs.yield analysis ( 收益分析 ),358-360  
cargo routing ( 安排貨運路線 ),27-30  
cargo tracking ( 貨物跟蹤 ),41-43  
COMPOSITE pattern,316-320  
composite routes ( 組合的路線 ),316-320  
concept analysis ( 概念分析 ),222  
conclusion ( 結論 ),502-504  
constructors ( 構造函數 ),174-175  
CONTEXT MAP,346-351  
ENTITIES,167-168  
extracting hidden concepts ( 提取隱藏的概念 ),17-20  
FACTORIES,174-175  
identifying missing concepts ( 找出丟失的概念 ),207-210

isolating the domain ( 隔離領域 ) ,166-167  
large-scale structure ( 大型結構 ) ,452-460  
MODULES,179-181  
multiple development teams ( 多個開發團隊 ) ,358-360  
performance tuning ( 性能優化 ) ,185-186  
refactoring ( 重構 ) ,177-179  
REPOSITORIES,172-173  
RESPONSIBILITY LAYERS,452-460  
route-finding ( 路線查找 ) ,312-314  
scenarios ( 場景 ) ,173-177  
SEGREGATED CORE,430-434  
shipping operations and routes ( 航運操作和路線 ) ,41-43  
STRATEGY,312-314  
system overview ( 系統簡介 ) ,163-166  
UBIQUITOUS LANGUAGE,27-30  
VALUE OBJECTS,167-168  
Examples ( 示例 ) ,concept analysis ( 概念分析 )  
extracting hidden concepts ( 提取隱藏的概念 ) ,17-20  
identifying missing concepts ( 找出丟失的概念 ) ,207-210  
implicit concepts ( 隱式概念 ) ,286-288  
researching existing resources ( 查詢現有資源 ) ,217-219  
resolving awkwardness ( 解決不足之處 ) ,211-215  
Examples ( 示例 ) ,distillation ( 精煉 )  
COHESIVE MECHANISMS,423-424,425-427  
GENERIC SUBDOMAINS,410-412  
organization chart ( 組織結構圖 ) ,423-424,425-427

SEGREGATED CORE,428-434  
time zones ( 時區 ),410-412  
Examples ( 示例 ),integration ( 集成 )  
ANTICORRUPTION LAYER,369-370  
Translator ( 轉換者 ),346-351  
unifying an elephant ( 「大象」的統一 ),378-381  
Examples ( 示例 ),large-scale structure ( 大型結構 )  
KNOWLEDGE LEVEL,466-474  
PLUGGABLE COMPONENT FRAMEWORK,475-479  
RESPONSIBILITY LAYERS,452-460  
Examples,LAYERED ARCHITECTURE  
partitioning applications ( 為應用程序分層 ),71-72  
RESPONSIBILITY LAYERS,452-460  
Examples ( 示例 ),loan management ( 貸款管理 )  
analysis patterns ( 分析模式 ),295-306  
breakthroughs ( 突破 ),194-200  
concept analysis ( 概念分析 ),211-215,217-219  
CONCEPTUAL CONTOURS,262-264  
conclusion ( 總結 ),501-502  
interest calculator ( 利息計算器 ),211-215,217-219,295-306  
investment banking ( 投資銀行業務 ),194-200  
refactoring ( 重構 ),194-200,284-292  
Explanatory models ( 解釋性模型 ),41-43  
Explicit constraints ( 顯式的約束 ),concept analysis ( 概念分析 ),220-222  
External systems ( 外部系統 ),383-385. 參見 integration.

Extracting hidden concepts ( 提取隱藏的概念 ),17-20.參見  
implicit concepts.

■F

FACADES,366-367

Facilities ( 信貸 ),194

FACTORIES

configuring SPECIFICATION ( 配置SPECIFICATION ),226-227

creating ( 創建 ),139-141

creating objects ( 創建對像 ),137-139

designing the interface ( 接口的設計 ),143

ENTITY vs.VALUE OBJECT,144-145

example ( 示例 ),174-175

invariant logic ( 固定規則的邏輯 ),143

overview ( 概述 ),136-139

placing ( 放置 ),139-141

reconstitution ( 重建 ),145-146

and REPOSITORIES,157-159

requirements ( 要求 ),139

FACTORY METHOD,139-141

False cognates ( 假同源 ),339-340

Film editing anecdote ( 電影剪輯趣聞 ),5

Flexibility ( 靈活性 ).參見supple design.

FLYWEIGHT pattern,320

Functions ( 函數 ),SIDE-EFFECT-FREE,250-254,285-286

■G

GENERIC SUBDOMAINS

adapting a published design ( 公開發佈的設計 ),408 example ( 示例 ),410-412

in-house solution ( 內部解決方案 ),409-410

off-the-shelf solutions ( 現成的解決方案 ),407

outsourcing ( 外包 ),408-409

overview ( 概述 ),406

reusability ( 可重用性 ),412-413

risk management ( 風險管理 ),413-414

vs.COHESIVE MECHANISMS,425

Granularity ( 粒度 ),108

## ■H

Hidden concepts ( 隱藏的概念 ),extracting ( 提取 ),17-20

HIGHLIGHTED CORE,417-421

Holy Grail anecdote ( 有關《聖盃與巨蟒》電影的一些趣聞 ),5

## ■I

Identity ( 標識 )

establishing ( 創建 ),90-93

local vs.global ( 本地 vs. 全局 ),127

tracking ( 跟蹤 ),94-96

Immutability of VALUE OBJECTS ( 保持 VALUE OBJECT 不變 ),100-101

Implicit concepts ( 隱式概念 )

categories of ( „的類別 ),219-223

recognizing ( 識別 ),206-219

Infrastructure layer ( 基礎設施層 ),70

Infrastructure-driven packaging ( 基礎設施驅動的打包 ),112-116

In-house solution ( 內 部 解 決 方 案 ),GENERIC SUBDOMAINS,409-410

Insurance project example ( 保險項目示例 ),372-373

Integration ( 集成 )

ANTICORRUPTION LAYER,364-370

CONTINUOUS INTEGRATION,341-343,391-393

cost/benefit analysis ( 成本/效益分析 ),371-373

elephant and the blind men ( 盲人與象 ),377-381

example ( 示例 ),372-373

external systems ( 外部系統 ),384-385

OPEN HOST SERVICE,374

SEPARATE WAYS,371-373

translation layers ( 轉 換 層 ),374. 參 見 PUBLISHED LANGUAGE.Integrity ( 集成 ) 參見model integrity.

INTENTION-REVEALING INTERFACES,246-249,422-427

Interest calculator examples ( 利息計算器示例 ),211-215,217-219,295-306

Internet Explorer bookmark anecdote ( IE書籤趣聞 ),57-59

Invariant logic ( 固定規則的邏輯 ),128-129,143

Inventory management example ( 庫存管理示例 ),504-505

Investment banking example ( 銀行業投資示例 ),194-200,211-215,501

Isolated domain layer ( 孤立的領域層 ),106-107

Isolating the domain ( 隔離領域 ) . 參見 ANTICORRUPTION LAYER; distillation; LAYERED ARCHITECTURE.

Iterative design process ( 迭代式設計過程 ) ,14,188,445

■J

Jargon ( 術語 ) . 參見 PUBLISHED LANGUAGE; UBIQUITOUS LANGUAGE.

Java entity beans vs.ENTITIES ( Java 實體bean與ENTITY ) ,91

■K

Knowledge crunching ( 知識消化 ) ,13-15

Knowledge crunching ( 知識消化 ) ,example ( 示例 ) ,7-12

KNOWLEDGE LEVEL,465-474

■L

Language of the domain experts ( 領域專家的語言 ) ,206-207

Large-scale structure ( 大型結構 ) . 參見 distillation; examples,large-scale structure; LAYERED ARCHITECTURE; strategic design.

CONTEXT MAP,446

Definition ( 定義 ) ,442

development constraints ( 限制開發 ) ,445-446

EVOLVING ORDER,444-446

Flexibility ( 靈活性 ) ,480-481

KNOWLEDGE LEVEL,465-474

minimalism ( 最小化 ) ,481

naive metaphor ( 幼稚隱喻 ) ,448-449

overview ( 概述 ) ,439-443

PLUGGABLE COMPONENT FRAMEWORK,475-479

refactoring ( 重構 ),481  
role in design ( 設計中的作用 ),329  
supple design ( 柔性設計 ),482-483  
SYSTEM METAPHOR,447-449  
team communication ( 團隊溝通 ),482  
Large-scale structure ( 大型結構 ),RESPONSIBILITY LAYERS  
choosing layers ( 選擇層 ),460-464  
overview ( 概述 ),450-452  
useful characteristics ( 有用的特徵 ),461  
LAYERED ARCHITECTURE. 參 見 distillation; examples,LAYERED  
ARCHITECTURE; large-scale structure.  
application layer ( 應用層 ),70,76-79  
callbacks ( 回調模式 ),73  
conceptual layers ( 概念層 ),70  
connecting layers ( 連接各層 ),72-74  
design dependencies ( 設計依賴 ),72-74  
diagram ( 圖表 ),68  
domain layer ( 領域層 ),70,75-79  
frameworks ( 框架 ),74-75  
infrastructure layer ( 基礎設施層 ),70  
isolated domain layer ( 孤立的領域層 ),106-107  
MVC (MODEL-VIEW-CONTROLLER),73  
OBSERVERS,73  
partitioning complex programs ( 紿複雜的應用程序劃分層  
次 ),70

separating user interface,application,and domain ( 將用戶界面層、應用層和領域層分開 ),76-79  
SERVICES,73-74  
SMART UI,73  
TRANSACTION SCRIPT,79  
user interface layer ( 用戶界面層 ),70,76-79  
value of ( „的價值 ),69

LAYERED ARCHITECTURE,ANTICORRUPTION LAYER  
ADAPTERS,367 considerations ( 需要考慮的因素 ),368-369  
FACADES,366-367  
interface design ( 接口設計 ),366-369  
overview ( 概述 ),364-366  
relationships with external systems ( 與外部系統的關係 ),384-385

LAYERED ARCHITECTURE,RESPONSIBILITY LAYERS  
choosing layers ( 選擇層 ),460-464  
overview ( 概述 ),450-452  
useful characteristics ( 有用的特徵 ),461  
Legacy systems ( 遺留系統 ),phasing out ( 逐步淘汰 ),393-394  
Life cycle of domain objects ( 領域對象的生命週期 ),123-124. 參見AGGREGATES; FACTORIES; REPOSITORIES.  
Loan management examples ( 貸款管理示例 ). 參見examples,loan management.  
Local vs.global identity ( 本地標識與全局標識 ),127

■M  
Merging ( 合併 )

OPEN HOST SERVICE and PUBLISHED LANGUAGE,394-396  
SEPARATE WAYS to SHARED KERNEL,389-391  
SHARED KERNEL to CONTINUOUS  
INTEGRATION,391-393  
METADATA MAPPING LAYERS,149  
Missing concepts ( 丢失的概念 ) ,207-210  
Mistaken identity anecdote ( 有關錯誤實體的趣聞 ) ,89  
Model integrity ( 模型完整性 ) . 參見 BOUNDED  
CONTEXT;CONTEXT MAP; multiple models.  
establishing boundaries ( 建立邊界 ) ,333-334  
multiple models ( 多個模型 ) ,333  
overview ( 概述 ) ,331-334  
recognizing relationships ( 識別關係 ) ,333-334  
unification ( 統一 ) ,332. 參見CONTINUOUS INTEGRATION.  
Model layer ( 模型層 ) . 參見domain layer.  
Model-based language ( 基於模型的語言 ) . 參見UBIQUITOUS  
LANGUAGE.  
MODEL-DRIVEN DESIGN  
correspondence to design ( 與設計的一致性 ) ,50-51  
modeling paradigms ( 建模範式 ) ,50-52  
overview ( 概述 ) ,49  
procedural languages ( 過程語言 ) ,51-54  
relevance of model ( 模型的相關 ) ,49  
tool support ( 工具支持 ) ,50-52  
Modeling ( 建模 )  
Associations ( 關聯 ) ,82-88

ENTITIES,93-94  
HANDS-ON MODELERS,60-62  
integrating with programming ( 與編程相結合 ) ,60-62  
non-object ( 非對像 ) ,119-122  
Models ( 模型 )  
binding to implementation ( 與實現綁定 ) 參見 MODEL-DRIVEN DESIGN.  
and user understanding ( 以及讓用戶理解 ) ,57-59  
MODEL-VIEW-CONTROLLER (MVC),73  
Modularity ( 模塊化 ) ,115-116  
MODULES  
agile ( 敏捷的 ) ,111  
cohesion ( 一致性 ) ,109-110,113  
costs of ( ...的成本 ) ,114-115  
coupling ( 耦合 ) ,109-110  
determining meaning of ( 決定...的意義 ) ,110  
examples ( 示例 ) ,111-112,179-181  
infrastructure-driven packaging ( 基礎設施驅動的打包 ) ,112-116  
mixing paradigms ( 混合範式 ) ,119-122  
modeling paradigms ( 建模範式 ) ,116-119  
modularity ( 模塊化 ) ,115-116  
naming ( 命名 ) ,110  
non-object models ( 非對像模型 ) ,119-122  
object paradigm ( 對像範式 ) ,116-119  
overview ( 概述 ) ,109

packaging domain objects ( 對領域對像打包 ),115

refactoring ( 重構 ),110,111

vs.BOUNDED CONTEXT,335

Monty Python anecdote ( Monty Pyrhon的趣聞 ),5

Multiple models ( 多個模型 ),333,335-340

MVC (MODEL-VIEW-CONTROLLER),73

## ■N

Naive metaphor ( 幼稚隱喻 ),448-449

Naming ( 命名 )

BOUNDED CONTEXTS,345

conventions for supple design ( 柔性設計的慣例 ),247

INTENTION-REVEALING INTERFACES,247

MODULES,110

SERVICES,105

Non-object models ( 非對像模型 ),119-122

## ■O

Object references ( 對像引用 ).參見REPOSITORIES.

Objects. 參見 ENTITIES; VALUE OBJECTS.associations ( 關聯 ),82-88

creating ( 創建 ),234-235. 參見 constructors;

FACTORIES.defining ( 定義 ),81-82

designing for relational databases ( 為關係數據庫設計對像 ),159-161

made up of objects ( 由對像組成 ). 參見 AGGREGATES;COMPOSITE.

Persistent ( 持久化的 ),150-151

OBSERVERS,73  
Off-the-shelf solutions ( 現成的解決方案 ) ,407  
Online banking example ( 網上銀行示例 ) ,71-72  
OPEN HOST SERVICE,converting to PUBLISHED  
LANGUAGE,394-396  
Outsourcing ( 外包 ) ,408-409  
Overbooking examples ( 超訂示例 ) ,18-19,222

■P

Packaging ( 打包 ) .參見deployment; MODULES.  
Paint-mixing application ( 調漆應用程序 ) ,examples ( 示例 ) ,247-249,252-254,256-259  
Partitioning ( 劃分 )  
complex programs ( 複雜程序 ) . 參見 large-scale structure; LAYERED ARCHITECTURE.  
SERVICES into layers,107  
Patterns ( 模式 ) ,507-510. 參見analysis patterns; design patterns; large-scale structure.  
PCB design anecdote ( PCB設計趣聞 ) ,7-13,501  
Performance tuning ( 性能優化 ) ,example ( 示例 ) ,185-186  
Persistent objects ( 持久的對象 ) ,150-151  
PLUGGABLE COMPONENT FRAMEWORK,475-479  
POLICY pattern ( POLICY模式 ) . 參見STRATEGY pattern.  
Polymorphism ( 多態 ) ,435-437  
Presentation layer. 參見user interface layer.  
Procedural languages ( 過程語言 ) ,and MODEL-DRIVEN DESIGN,51-54

Processes as domain objects ( 作為領域對象的過程 ) ,222-223  
Prototypes ( 原型 ) ,238-241

#### PUBLISHED LANGUAGE

elephant and the blind men ( 盲人與象 ) ,377-381  
example ( 示例 ) ,377  
merging with OPEN HOST SERVICE ( 與OPEN HOST SERVICE合併 ) ,394-396  
overview ( 概述 ) ,375-377

#### ■Q

Quilt project ( 拼被 ) ,479

#### ■R

Reconstitution ( 重建 ) ,145-146,148  
Refactoring ( 重構 )  
Breakthroughs ( 突破 ) ,193-200  
during a crisis ( 危機之中 ) ,325-326  
deep models ( 深層模型 ) ,189-191  
definition ( 定義 ) ,188  
designing for developers ( 針對開發人員的設計 ) ,324  
discovery ( 發現 ) ,191-192  
distillation ( 精煉 ) ,437  
examples ( 示例 ) ,177-179,181-185,194-200,247-249  
exploration teams ( 探索團隊 ) ,322-323  
initiation ( 開始重構 ) ,321-322  
large-scale structure ( 大型結構 ) ,481  
levels of ( „的層次 ) ,188-189  
MODULES,110,111

to patterns ( 模式 ),188-189  
reusing prior art ( 重用先前的經驗 ),323-324  
supple design ( 柔性設計 ),191  
timing ( 時機 ),324-325  
Refactoring targets ( 重構目標 ),437  
Reference objects ( 引用對像 ).參見ENTITIES.  
**REPOSITORIES**  
advantages ( 優點 ),152  
architectural frameworks ( 架構框架 ),156-157  
decoupling from the client ( 與客戶解耦 ),156  
designing objects for relational databases ( 為關係數據庫設計對像 ),159-161  
encapsulation ( 封裝 ),154  
example ( 示例 ),172-173  
and FACTORIES,157-159  
global searches ( 全局搜索 ),150-151  
implementing ( 實現 ),155-156  
**METADATA MAPPING LAYERS**,149  
object access ( 對像訪問 ),149-151  
overview ( 概述 ),147-152  
persistent objects ( 持久對像 ),150-151  
querying ( 查詢 ),152-154  
references to preexisting domain objects ( 獲取已存在的領域對象的引用 ),149  
transaction control ( 事務的控制權 ),156  
transient objects ( 臨時對像 ),149

type abstraction ( 對類型進行的抽象 ),155-156  
Requirements gathering ( 需求收集 ). 參見 concept analysis;knowledge crunching; UBIQUITOUS LANGUAGE.

## RESPONSIBILITY LAYERS

choosing layers ( 選擇層 ),460-464  
example ( 示例 ),452-460  
overview ( 概述 ),450-452  
useful characteristics ( 有用的特徵 ),461

Reusing code ( 重新使用代碼 )

BOUNDED CONTEXT,344

GENERIC SUBDOMAINS,412-413

reusing prior art ( 重用先前的經驗 ),323-324

Risk management ( 風險管理 ),413-414

## ■S

Scenarios ( 場景 ),examples ( 示例 ),173-177

SEGREGATED CORE,428-434

Selecting objects ( 選擇對像 ),229-234,269-270

SEPARATE WAYS,384-385,389-391

SERVICES. 參見 ENTITIES; VALUE OBJECTS.

access to ( 對 „ 的訪問 ),108

characteristics of ( „ 的特徵 ),105-106

granularity ( 粒度 ),108

and the isolated domain layer ( 以及鼓勵的領域層 ),106-107

naming ( 命名 ),105

overview ( 概述 ),104-105

partitioning into layers ( 分層 ),107

## SHARED KERNEL

example (示例), 359

merging with CONTINUOUS INTEGRATION (與CONTINUOUS INTEGRATION合併), 391-393

merging with SEPARATE WAYS (與SEPARATE WAYS合併), 389-391

overview (概述), 354-355

Sharing (共享) VALUE OBJECTS, 100-101

Shipping examples (貨運示例). 參見examples, cargo shipping (貨運示例).

Side effects (副作用), 250. 參見ASSERTIONS.

SIDE-EFFECT-FREE FUNCTIONS, 250-254, 285-286

Simplifying your design (簡化你的設計). 參見distillation; large-scale structure; LAYERED ARCHITECTURE.

SMART UI, 73

SPECIFICATION. 參見analysis patterns; design patterns.

applying (應用), 227

business rules (業務規則), 225

combining (結合). 參見composite SPECIFICATION.

composite (組合), 273-281

configuring (配置), 226-227

definition (定義), 225-226

example (示例), 29, 235-241, 279-282

generating objects (生成對像), 234-235

implementing (實現), 227

overview (概述), 224-227

purpose ( 目的 ) ,227  
selecting objects ( 選擇對像 ) ,229-234  
validating objects ( 驗證對像 ) ,227,228-229  
Speech ( 講話 ) ,common language ( 公共語言 ) . 參見  
UBIQUITOUS LANGUAGE.  
Speech ( 講話 ) ,modeling through ( 通過...建模 ) ,30-32  
STANDALONE CLASSES,265-267  
Strategic design ( 策略設計 ) . 參見 large-scale  
structure.assessing the situation ( 評估現狀 ) ,490  
customer-focused architecture teams ( 以客戶為中心的架構團隊 ) ,492  
developers ( 開發者 ) ,role of ( ...的角色 ) ,494  
essential requirements ( 基本要求 ) ,492-495  
evolution ( 演變 ) ,493  
EVOLVING ORDER,491  
feedback process ( 反饋過程 ) ,493  
minimalism ( 最小化 ) ,494-495  
multiple development teams ( 多個開發團隊 ) ,491  
objects ( 對像 ) ,role of ( ...的角色 ) ,494  
setting a strategy ( 制定策略 ) ,490-492  
team communication ( 團隊溝通 ) ,492  
team makeup ( 團隊成員的組成 ) ,494  
technical frameworks ( 技術框架 ) ,495-497  
STRATEGY pattern,19,311-314  
Supple design ( 柔性設計 )  
approaches to ( ...的方法 ) ,282-292

ASSERTIONS,255-259

CLOSURE OF OPERATIONS,268-270

composite SPECIFICATION,273-282

CONCEPTUAL CONTOURS,260-264

declarative design ( 聲明式設計 ),270-272

declarative style of design ( 聲明式設計風格 ),273-282

domain-specific language ( 特定於領域的語言 ),272-273

example ( 示例 ),247-249

INTENTION-REVEALING INTERFACES,246-249

Interdependencies ( 內部依賴 ),265-267

large-scale structure ( 大型結構 ),480-483

naming conventions ( 命名規範 ),247

overview ( 概述 ),243-245

SIDE-EFFECT-FREE FUNCTIONS,250-254,285-286

STANDALONE CLASSES,265-267

SYSTEM METAPHOR,447-449

System under design ( 正在設計的系統 ),385-386

## ■T

Team context ( 團隊上下文 ),382

Teams ( 團隊 )

choosing a strategy ( 選擇一種策略 ),382

communication ( 溝通 ),large-scale structure ( 大型結構 ),482

customer-focused ( 以客戶為中心的 ),492

defining ( 定義 ) BOUNDED CONTEXT,382

developer community ( 開發者社區 ),maturity of ( ...變得成熟 ),117-119

exploration ( 探索 ),322-323  
Teams ( 團隊 ),and strategic design ( 以及策略設計 )  
communication ( 溝通 ),492  
customer-focused ( 以客戶為中心的 ),492  
developers ( 開發者 ),role of ( „的角色 ),494  
makeup of ( „的組成 ),494  
multiple teams ( 多個團隊 ),491  
Teams ( 團隊 ),multiple ( 多個 )  
ANTICORRUPTION LAYER,364-370  
CONFORMIST,361-363  
CUSTOMER/SUPPLIER,356-360  
example ( 示例 ),358-360  
SHARED KERNEL,354-355,359  
strategic design ( 策略設計 ),491  
Terminology ( 術語 ). 參見 BOUNDED CONTEXT; PUBLISHED LANGUAGE; UBIQUITOUS LANGUAGE.  
Testing boundaries ( 測試邊界 ),351  
Transaction control ( 事物的控制權 ),156  
TRANSACTION SCRIPT,79  
Transformations ( 轉換 ),389  
Transforming boundaries ( 轉換邊界 ),382-383  
Transient objects ( 臨時對像 ),149  
Translation layers ( 轉換層 ),374  
Tuning a database ( 優化數據庫 ),example ( 示例 ),102  
**■U**  
UBIQUITOUS LANGUAGE. 參見 PUBLISHED LANGUAGE.

analysis patterns ( 分析模式 ),306-307  
cargo router example ( 安排貨運路線示例 ),27-30  
consistent use of ( 持續使用 „ „ ),32-35  
designing objects for relational databases ( 為關係數據庫設計對像 ),160-161  
domain-specific language ( 領域特定語言 ),272-273  
language of the domain experts ( 領域專家的語言 ),206-207  
overview ( 概述 ),24-27  
refining the model ( 精化模型 ),30-32  
specialized terminologies ( 專用術語 ),386-387  
requirements analysis,25  
speech ( 講話 ),role of ( „ „ 的角色 ),30-32  
UML (Unified Modeling Language),35-37  
Unification ( 統一 ),332. 參見CONTINUOUS INTEGRATION.  
Unified Modeling Language (UML),35-37  
Updating the design ( 更新設計 ). 參見refactoring.  
User interface layer ( 用戶界面層 )  
business logic ( 業務邏輯 ),77  
definition ( 定義 ),70  
separating from application and domain ( 從應用層和領域層分開 ),76-79

## ■V

Validating objects ( 驗證對像 ),227,228-229

VALUE OBJECTS. 參見ENTITIES; SERVICES.

associations ( 關聯 ),102-103

bidirectional associations ( 雙向關聯 ),102-103

change management ( 變更管理 ),101  
clustering ( 聚集 ).參見AGGREGATES.  
designing ( 設計 ),99-102  
example ( 示例 ),167-168  
immutability ( 不變性 ),100-101  
object assemblages ( 對像集合 ),98-99  
overview ( 概述 ),97-99  
passing as parameters ( 作為參數傳遞 ),99  
referencing ENTITIES ( 引用ENTITY ),98-99  
sharing ( 共享 ),100-101  
tuning a database ( 優化數據庫 ),example ( 示例 ),102  
Vision statement ( 前景說明 ).參見 DOMAIN VISION STATEMENT.  
Vocabulary ( 詞彙 ).參見PUBLISHED LANGUAGE; UBIQUITOUS LANGUAGE.  
**■W**  
Waterfall design method ( 瀑布方法 ),14  
Web site bookmark anecdote ( 網站書籤趣聞 ),57-59