



## 第 II 部分

# 模型驱动设计的构建块

为了让软件实现与模型始终保持一致，不管实际处理有多么麻烦，都必须运用建模和设计的最佳实践。本书既不是一本介绍面向对象的专著，也不是为了介绍基本的设计原理。对某些传统理念来讲，领域驱动设计转移了其重心。

某些决策把模型和实现结合在一起，相互增强。这种结合要求注意各个元素的细节。这种小规模的精心制作给开发人员提供了一个稳定的平台，其中应用的建模方法将在第Ⅲ部分和第Ⅳ部分中介绍。

本书的设计风格绝大部分遵循职责驱动设计(Responsibility-Driven Design)原则，这种原则由 Wirfs-Brock 等于 1990 年提出，并且在 2003 年被修正。它着重介绍(尤其在第Ⅲ部分)了 Meyer 于 1988 年提出的“契约式设计(Design by Contract)”的观点。它与其他一些被广泛应用的面向对象设计的最佳实践有着相同的背景，这些最佳实践在 Larman 于 1998 年出的书中进行了描述。

当项目在开发过程中受到了挫折，不管是大是小，开发人员都会发现在某些情况下这些原则好像不能适用了。为了使领域驱动设计过程具有灵活性，开发人员需要了解那些众所周知的基本原则是怎样支持模型驱动设计的，这样他们可以兼顾这些原则而不会脱离领域驱动设计。

后面 3 章的资料被组织成一种“模式语言”(参见附录 A)，它将展示模型差别和设计决策对领域驱动设计过程产生的微妙影响。

下图是一个导航图，它展示了将在这部分中介绍的模式以及它们之间相互联系的一些方式。

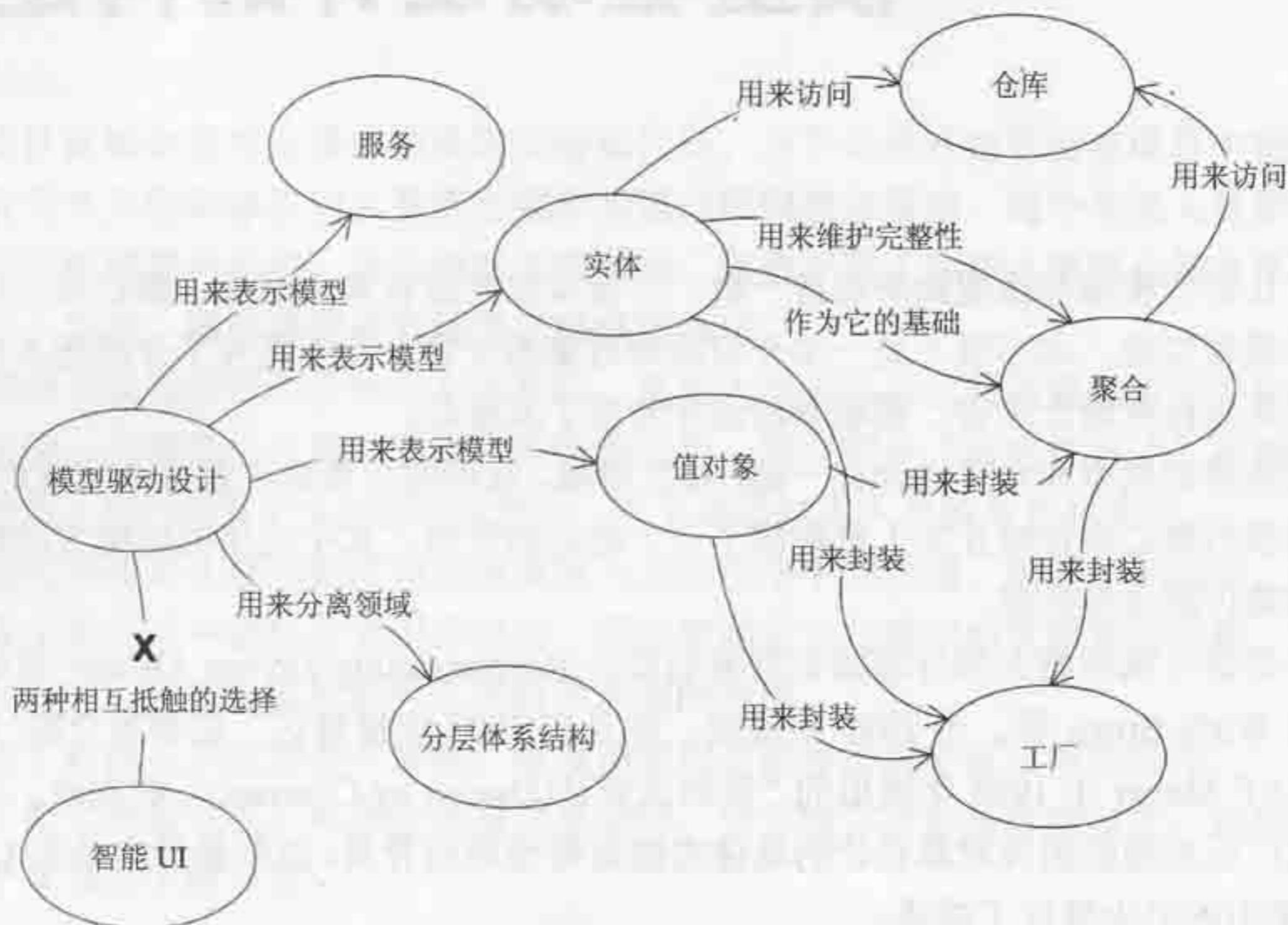
共享这些标准模式可以使设计具有条理性，并且使团队成员更容易了解彼此的工作。在通用语言中添加对标准模式的使用，整个团队可以用这种语言来讨论模型和设计决策。



开发一个优秀的领域模型是一门艺术。但是一个模型中各个元素的实际设计和实现还是比较成体系的。将领域设计从众多软件系统关注的问题中隔离出来，会使设计和模型的关系变得非常清楚。根据某些区别来定义模型元素能够突出它们的含义。对各个元素采用已被证明的模式能帮助我们建立一个实际要实现的模型。

只要谨慎地应用基本原则，精心创建模型，就可以消除复杂性，从而开发团队能够比较自信地把这些具体的元素结合起来。

本章将介绍如何通过模型驱动设计来应对复杂性，从而构建出可维护、可扩展且易于测试的系统。



模型驱动设计语言的导航图

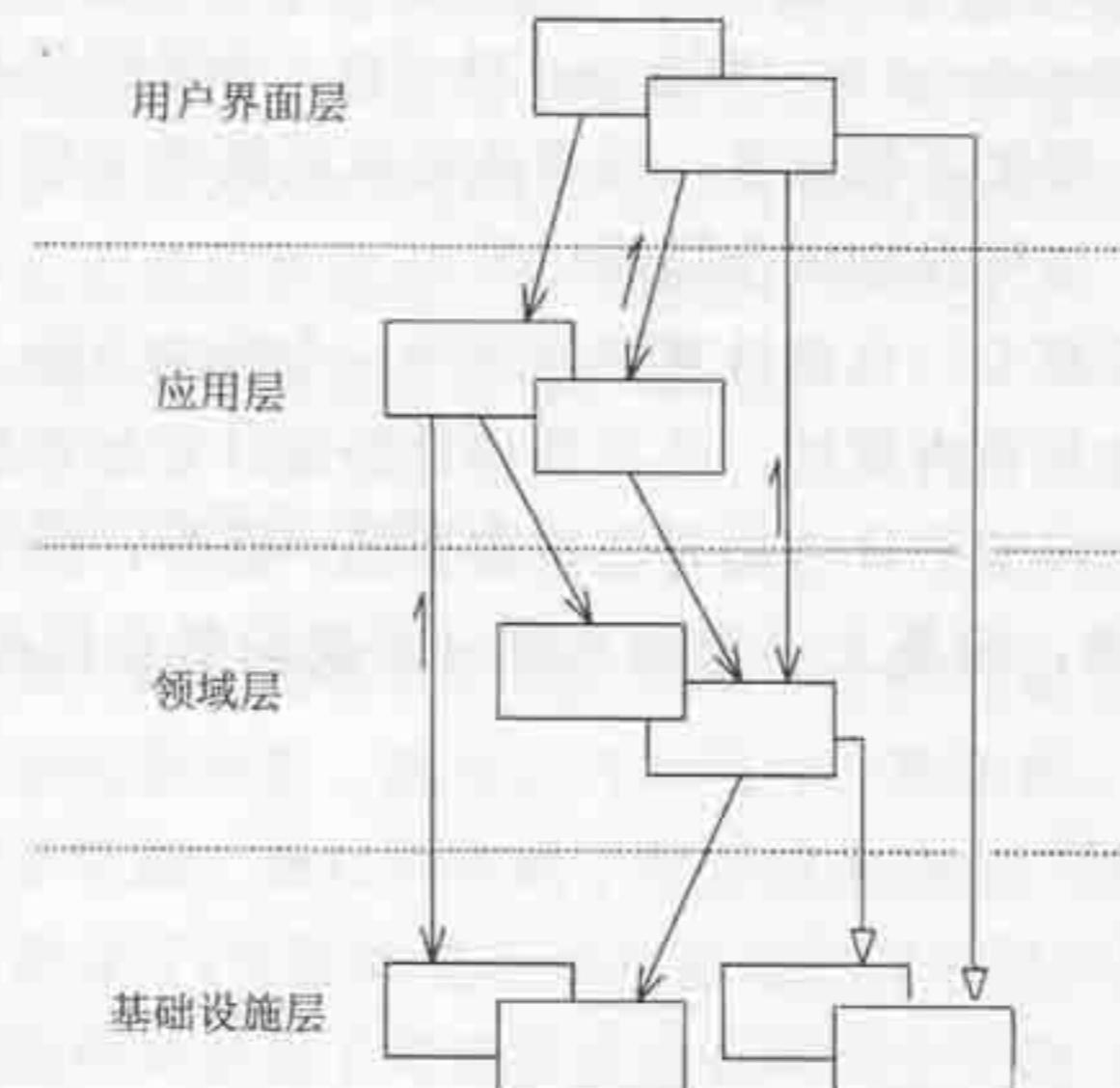
## 第4章

# 分离领域

解决来自领域方面问题的软件部分通常只占整个软件系统的一小部分，这与它的最重要性相比是不成比例的。为了应用我们最优秀的思想，我们需要考虑模型中的元素并将它们视为一个系统。这并不是要强迫我们从一大堆混合的对象中把它们挑出来，就好像要在夜空中辨别星座一样。我们需要把领域对象跟系统的其他功能分离出来，才能够避免将领域概念与其他跟软件技术相关的概念混淆或者在系统的庞大中失去了对领域的把握。

这种成熟的领域分离技术早已出现，它为我们打下了良好的基础，但对于领域建模原则的成功应用来说，分离领域是很关键的，所以必须从领域驱动的观点对其进行简单回顾。

### 4.1 分层架构





在一个航运程序中，如果要为用户提供从城市列表中选择运送货物目的地的简单操作，需要进行如下编程：1)在屏幕上绘制一个窗口；2)查询数据库中的所有可能的城市；3)解析用户输入并验证；4)把选择的城市和货物关联起来；5)向数据库提交变更。虽然所有代码都是同一个程序的一部分，但只有一小部分代码与运输业务有关。

软件程序需要进行设计和编码来执行很多不同类型的任务。它们接收用户输入、执行业务逻辑、访问数据库、进行网络通信、向用户显示信息等。所以每个程序功能涉及的代码可能都很重要。

在面向对象的程序中，用户界面(UI)、数据库和其他支持代码，经常被直接写到业务对象中去。在 UI 和数据库脚本的行为中嵌入额外的业务逻辑。出现这种情况是因为从短期的观点看，它是使系统运行起来的最容易的方式。

当与领域相关的代码和大量的其他代码混在一起时，就很难阅读并理解了。对 UI 的简单改动就会改变业务逻辑。改变业务规则可能需要小心翼翼地跟踪 UI 代码、数据库代码或者其他程序元素。实现一致的模型驱动对象变得不切实际，而且自动化测试也难以使用。如果在程序的每个行为中包括了所有的技术和逻辑，那么它必须很简单，否则会难以理解。

创建能够处理非常复杂任务的程序要求分离关注点，这样允许隔离地关注设计的不同部分。同时不管是否分离，都要维护好系统中复杂的交互。

对软件系统的分割有各种各样的方法，但是按经验和惯例，软件行业确定了分层架构，尤其是确定了一些公认的标准层。分层的隐喻得到了广泛使用，绝大多数开发人员都感到它很直观。有不少文献对分层进行了大量有见地的讨论，有时候这些讨论会以模式的形式出现(例如 Buschmann et al. 1996, pp. 31–51)。分层的基本原则是：某一层中的所有元素只能依赖于同一层的其他元素，或者依赖其直接的下层元素。向上的信息传递必须经过一些间接机制，这些我们将在稍后讨论。

分层的意义在于每层都专门负责计算机程序中一个特定方面。这种基于专责的划分，可以使各方面设计更加具有内聚性，并且使得这些设计更加容易解释。当然，如何选择层次，来隔离最重要的内聚设计方面是非常重要的。还好，经验和惯例可以再一次帮助我们。尽管有许多变体，但是大多数成功的分层架构都会使用这四种概念层的某些版本。



用户界面层 (表示层)	负责向用户显示信息，并且解析用户命令。外部的执行者有时可能会是其他的计算机系统，不一定非是人
应用层	定义软件可以完成的工作，并且指挥具有丰富含义的领域对象来解决问题。这个层所负责的任务对业务影响深远，对跟其他系统的应用层进行交互非常必要。这个层要保持简练。它不包括处理业务规则或知识，只是给下一层中相互协作的领域对象协调任务、委托工作。在这个层次中不反映业务情况的状态，但反映用户或程序的任务进度的状态
领域层 (模型层)	负责表示业务概念、业务状况的信息以及业务规则。尽管保存这些内容的技术细节由基础结构层来完成，反映业务状况的状态在该层中被控制和使用。这一层是业务软件的核心
基础结构层	为上层提供通用的技术能力：应用的消息发送、领域持久化，为用户界面绘制窗口等。通过架构框架，基础结构层还可以支持这四层之间的交互模式

一些项目没有明显区分用户界面层和应用层。而有些项目则拥有多个基础结构层。但是为了能够进行模型驱动设计，将领域层分离出来至关重要。

因此：

将一个复杂的程序进行层次划分。为每一层进行设计，每层都是内聚的而且只依赖于它的下层。采用标准的架构模式来完成与上层的松散关联。将所有与领域模型相关的代码都集中在一层，并且将它与用户界面层、应用层和基础结构层的代码分离。领域对象可以将重点放在表达领域模型上，不需要关心它们自己的显示、存储和管理应用任务等内容。这样使模型发展得足够丰富和清晰，足以抓住本质的业务知识并实现它。

将领域层从基础结构层和用户界面层中分离出来，可以使各层的设计更加清晰。相互隔离的层的维护开销小得多，因为它们往往是以不同的速度发展并且响应不同的要求。分离同样可以有助于分布式系统的部署，为了使通信开销最小、提高性能(Fowler 1996)，可以把各层灵活地放在不同的服务器或者客户端上。

### 示例：对网上银行功能进行层次划分

应用应该提供维护银行账户的各种功能。其中一个特性是资金转账，当用户输入或者选择了两个账号和资金的数目后，就可以开始执行一次转账。

为了让这个示例便于管理，我已经省略了很多技术特性，特别是在安全性方面的特性。领域设计也被大大简化了(实际的复杂性只是增加对分层架构的需要)。而且，在这里隐含使用的特定基础结构只是为了简单清楚地说明这个示例，而不是一个推荐的设计。



剩余功能的职责划分如图 4-1 所示。

注意，是领域层而不是应用层负责提供基本的规则，在这个例子中，规则是：“有借必有贷，借贷必相等。”

在这个应用程序中没有假定转账请求的发起者。该程序的用户界面大概包括输入账号、金额的输入区，以及一些命令按钮。但是这个用户界面可以被替换成一个基于 XML 的在线请求，并且不影响应用层和较低的层。这种解耦是非常重要的，这并不是因为项目经常需要用在线请求的方式更改用户界面，而是因为清晰的关注点分离可以使每个层次的设计更加容易理解和维护。

实际上，图 4-1 本身也能略微说明没有隔离领域的问题。因为必须包办交易控制请求的所有事情，领域层必须向上兼容，以保持整个交互足够简单到能够跟踪。如果我们聚焦于隔离领域层的设计，我们会更多地在纸上或者头脑中考虑模型应该如何更好地表现领域规则，这些规则可能包括总账、存款和贷款对象，或现金交易对象等。

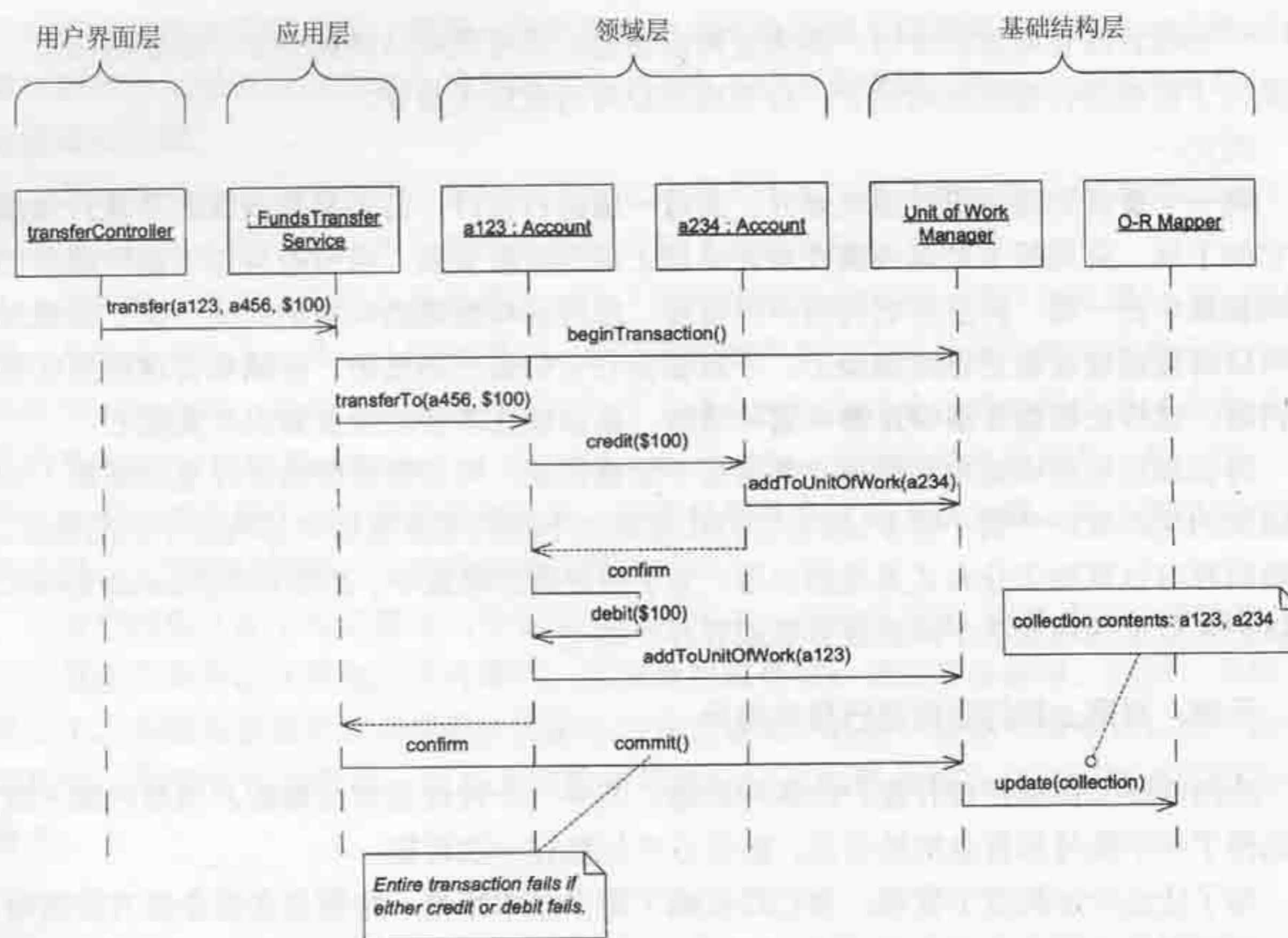


图 4-1 对象要完成的职责与其所在层一致，并与同一层的其他对象联系紧密



### 4.1.1 层间的联系

到目前为止，讨论还集中在层的划分以及可以增强程序各个方面设计的划分方法上，特别是领域层。当然，各个层之间必须要连接起来。把各个层连接起来而不影响分离给设计带来的好处，是许多模式所追求的。

层是一种松散关联结构，因为设计的依赖是单向的。上层可以直接使用或者操纵下层的元素，这可以通过直接调用下层的公共接口，维护对它们的引用(至少是临时的)或者使用常规的交互方法来完成。但是当一个下层的对象需要跟上层对象通信时(不只是应答一个简单的查询)，我们就需要另一种机制，采用架构模式把各层联系起来，例如使用回调或 Observer(观察者)模式(Gamma et al.1995)。

最早用来连接用户界面层、应用层和领域层的模式是模型-视图-控制器(Model-View-Controller, MVC)。它最早于 20 世纪 70 年代后期在 Smalltalk 环境中被提出来，并且影响了随后出现的很多 UI 架构。Fowler(2002)讨论了这种模式以及此主题的几个有用的变体。Larman(1998)在模型-视图分离模式(Model-View Separation Pattern)中研究了这些问题，并且他提出的 Application Coordinator 模式是连接应用层的一种方法。

还有其他风格来连接用户界面层和应用层。在我们看来，只要它们坚持隔离领域层的原则，让设计者在设计领域对象的同时不需要考虑与用户界面的交互，那么所有的方法都是可行的。

基础结构层通常不在领域层发起动作。因为它在领域层的下面，所以它对要服务的领域根本没有什么了解。事实上这种技术能力通常以服务的方式提供。例如，如果一个应用需要发送电子邮件，一些消息发送接口可以放在基础结构层上，应用层的元素就可以请求发送消息。这种分散化处理提供了一些附加的好处。消息发送接口可以连接到电子邮件发送器，传真发送器，或者其他可用的发送对象。但是最大的好处是简化了应用层，使应用层专注于它的工作：知道什么时候发送消息，而不用操心如何发送。

应用层和领域层调用由基础结构层提供的服务。当一个服务的范围被选定并且它的接口被设计好后，通过服务接口精心封装的行为，调用者可以保持松散关联和简单性。

但不是所有的基础结构层都作成这种能被上层调用的服务的形式。有些技术组件被设计为具有直接支持其他层的基本功能(例如为所有的领域对象提供一个抽象基类)并且为各个层提供联系机制(例如 MVC 以及其类似的实现)。这种“架构框架”对程序其他部分的设计有更大的影响。

### 4.1.2 架构框架

当基础结构被作成以通过接口进行调用的服务的形式时，那怎么分层以及怎样保持



各层之间的松散关联还是比较直观的。但是有些技术问题要求具有更加深入的形式的基础结构。集成了许多基础结构的框架，常常要求其他层按照非常特定的方式来实现，例如，作为框架类的子类或者使用构造好的方法签名(将子类而不是父类放到较高的层看起来可能会有违直觉，但是要牢记哪个类反映的内容更加丰富)。最好的架构框架在解决复杂的技术问题的同时，允许领域开发人员集中精力去表达一个模型。但是框架很容易造成障碍，要么假设的东西太多，限制了领域设计的选择；要么实现的东西太多，影响了开发的速度。

通常需要某些形式的架构框架(尽管有时候团队选择的架构框架并不是很合适)。在应用一个框架时，开发团队应该首先明确它的目标：创建一个实现来表示一个领域模型并且用这个实现来解决重要的问题。开发团队必须寻求使用框架完成这些目标的方法，尽管这可能意味着没有用到这个框架所有的特性。例如，早期的 J2EE 应用经常把所有的领域对象作为“entity beans”来实现。这种方法会影响程序的性能和开发的速度。然而，当前的最佳实践是在 J2EE 框架上实现大粒度对象，用普通的 Java 对象来实现大多数的业务逻辑。许多框架的不足可以通过有选择地应用它们解决难题来避免，不要妄想能找出一种万能的解决方案。明智地只应用框架中最有价值的部分，降低实现与框架的依赖程度，使以后的设计决策具有更多的灵活性。更重要的是，假设现在的许多框架使用起来非常复杂，这种极简主义会帮助您保持业务对象的可读和可表达。

架构框架和其他工具将会不断地发展。应用中的技术方面会越来越多地在较新的框架中自动完成或者预先制定好。如果能发展到这一步，应用的开发人员将有更多的时间和精力投入到建模核心业务问题上，极大地提高软件生产力和质量。但是当我们朝这个方向发展时，我们必须控制我们对技术解决方案的热衷程度，精细的框架也可能成为应用程序开发人员的束缚。

## 4.2 模型属于领域层

如今，在绝大多数系统中都用到了分层架构，只是各个系统的分层的方案可能不同而已。许多开发风格能够从分层中获得帮助。可是领域驱动设计要求只能有一个特定的层存在。

领域模型是一组概念。“领域层”是模型和所有与其直接相关的设计元素的显现。业务逻辑的设计和实现构成了领域层。在一个模型驱动设计中，领域层的软件结构反映出模型的概念。

当领域逻辑和程序的其他关注点混在一起时，要达到一致是很不现实的。隔离领域



实现是领域驱动设计的前提条件。

### 反模式——智能 UI

上面总结了在对象应用中被广泛采用的分层架构模式。开发人员经常尝试着把系统划分为用户界面层、应用层和领域层，但是却很少有项目能够完全做到，所以需要对这种负面行为进行公正的讨论。

许多软件项目采用而且还将继续采用一种不太成熟的设计方法，我把它叫作智能 UI(Smart UI)。但是智能 UI 走上了另一面，如同道路上互斥的岔道，与领域驱动设计的方法格格不入。如果选择了这条开发道路，那么本书的大部分内容将对您毫无帮助。我只对不采用智能 UI 的情况感兴趣，这就是为什么在这里我把它叫作智能 UI 的原因，通俗一点说就是“反模式”。在这里讨论智能 UI 是为了提供一个有用的对比，帮助我们弄清情况，证明在本书后面还会碰到更多的难题。

一个软件项目需要提供简单的功能，主要用于一些数据输入和显示，涉及的业务规则很少。而且开发团队不是由高级的对象建模人员组成。

如果一个没有经验的开发团队，决定尝试用一个使用了分层架构的模型驱动设计来开发一个简单的项目，那么该团队将面临艰难的学习曲线。团队成员必须要掌握复杂的新技术，并且经历学习对象建模的艰苦过程(这是一个挑战，即使有这本书的帮助！)。管理基础结构和层的代价会使非常简单的任务的开发周期拖得更长。简单项目往往要求在短时间内完成而且期望的要求不高。因此，在这个团队完成被分配的任务之前，在他们还没有证明采用的方法会带来令人兴奋的可能之前，这个项目早已被取消。

即使这个开发团队有充裕的时间来开发，如果没有专家的帮助，团队成员可能会无法很好地掌握这种技术。最后，如果他们确实战胜了这些挑战，他们也只不过是开发了一个简单的系统，不会有人要求丰富的功能。

一个拥有丰富开发经验的团队是不会遇到这种情况的。一个老练的开发人员可以踏平这条学习曲线，并且能够缩短管理各层的时间。要求很高的项目可以从领域驱动设计中获得很多好处，它要求有非常强的专业技术。不是所有的项目要求都很高，也不是所有的项目团队都能掌握那些技术。

因此，当条件允许时：

把所有的业务逻辑交给用户界面处理。将整个应用程序分割成小的功能函数，并且把它们作为相互独立的用户界面来实现，同时把业务规则嵌入到这些界面中。用一个关系数据库作为数据的共享仓储。使用最自动化的 UI 结构，以及可利用的可视化编程工具。

纯粹是旁门左道！(现在流行的，包括在本书中提倡的)真理是领域和 UI 应该被分离。



事实上，如果不将 UI 和领域分离，那么将很难应用本书在后面讨论的许多方法。因此，在领域驱动设计的上下文中，可以把这个智能 UI 看成是一个“反模式”，虽然它在一些上下文中是一个合理的模式。事实上，智能 UI 有它的优势，在一些情况下它工作得非常好，这就是这种方法为什么那么常见的原因。在这里我们之所以提到智能 UI，只是为了帮助我们更好地理解为什么需要将应用层与领域层分离，还有更重要的是，让我们知道什么时候不需要这么做。

### 优点

- 对于简单的应用，生产力较高，开发时间较短。
- 缺少经验的开发人员可以经过较短的培训直接上手。
- 甚至在进行需求分析时所留下的缺陷，可以通过把原型提供给用户来克服，并快速地在软件中作出修改来满足客户的要求。
- 应用可以相互分离，所以能够精确地计划递交小模块的进度表。对系统进行简单的扩展可能会很容易。
- 关系数据库工作可靠，并且提供数据级上的集成。
- 第 4 代语言(4GL)工具能很好地满足开发需要。
- 当这个应用程序被递交后，维护程序员能够很快地重新开发他们没有解决的部分，因为改变所带来的影响只局限在每个特定的 UI 中。

### 缺点

- 应用的集成比较困难，除非利用数据库。
- 这里不会考虑重用以及业务问题的抽象。业务规则必须在每个使用它的操作中复制。
- 因为缺少抽象的提炼而限制了重构的选择，所以快速原型和迭代受到了天然的限制。
- 复杂性很快会让您迷失，所以增长路线只能严格顺着在原有应用上添加简单应用而已。要想获得具有丰富行为的应用并不容易。

如果这种模式被有意识应用，开发团队能够避免承担大量其他方法所需要的开发工作。在设计时有一个通病，那就是采用了一个成熟的设计方法，但开发团队却不是严格地遵照这种方法来进行开发。另外一个通病就是，花费巨大的人力和财力去构建一个复杂的基础结构，并使用工业级的工具来实现一个根本就不需要这些基础结构的项目。

最灵活的语言(例如 Java)已经远远超过了这些应用的需求，不过花费巨大。其实一个 4GL 风格的工具就已经足以满足开发的需要了。

记住，采用这种模式的一个后果就是您不能把它移植到另外一种设计方法中，除非



替换整个应用程序。使用一种通用的语言，例如 Java，并不会真正保证您最后会放弃使用智能 UI，所以如果选择了这种开发模式后，您应该选择合适它的开发工具。不要为如何去选择而感到烦恼，就算是采用了一种灵活的语言也未必能够创建出一个灵活的系统，但很可能会增加我们开发的代价。

出于同样的原因，一个开发团队选择了模型驱动设计方法，就需要从头开始设计。当然，就算是有经验的开发团队开发大型项目时，也必须从简单的功能函数开始，兢兢业业地不断迭代。但是那些试探性的第一步，都是采用隔离领域层的模型驱动设计，要不然就是在坚持使用智能 UI。在这里讨论的智能 UI 仅仅是为了说明为什么以及在什么时候，为了隔离领域层而需要采用一个像分层架构那样的模式。

在智能 UI 和分层架构之间还有其他的解决方案。例如 Fowler(2002)提出的事务脚本(Transaction Script)模式，它从应用程序中把用户界面分离出来，但是并不提供一个对象模型。它的底线是：如果一个结构隔离了与领域相关的代码，同时，允许内聚的领域设计与系统的剩余部分保持一种松散关联性，那么这种架构就可以支持领域驱动设计。

其他的开发风格也有他们的一些优势，但是您必须在复杂性和灵活性上做出考虑。缺乏领域设计的解耦，在某种情况下可能真的是一种灾难。如果采用了模型驱动设计来开发一个复杂的应用，那就不要心痛您的钱，咬紧牙关聘请那些必要的专家，并避免采用智能 UI。

### 4.3 其他种类的隔离

遗憾的是，除了基础结构和用户界面外，还有其他因素会破坏您精致的领域模型。您必须考虑到那些没有完全集成到模型中的领域组件。您必须和在同一个领域中使用不同模型的开发团队竞争，还有其他的一些因素会破坏您的模型，并且降低它的效用。第 14 章“维护模型完整性”，会处理这个问题，并介绍限界上下文(Bounded Context)和防腐层(Anticorruption Layer)模式。一个真正复杂的领域模型自己会变得难以使用。第 15 章“精炼”，讨论了怎样才能从外围细节中精炼出领域的基本概念。

这些都将在后面的相关章节详细讨论。接下来，让我们看看共同演进一个有效的领域模型和有表达力的实现的具体细节。毕竟，隔离领域的最佳部分是排除无关的东西，这样我们可以把精力真正集中在领域设计上。

# 软件中的模型描述

在不损失模型驱动设计的能力的情况下，如果要在软件实现时做一些权衡和折衷，需要对基本元素重新组织。应该在比较具体的层面上将模型与实现相互联系起来。本章将集中讨论这些独立的模型元素，将它们明确化，为后面章节中的使用打好基础。

我们首先将从如何设计和组织关联的问题开始讨论。对象间的关联可以非常容易地得到，也很容易画出来，但是要实现它们却不是那么容易的。关联也表明，对于模型驱动设计来说，详细的实现决策是多么重要。

让我们转向对象本身，但是我们仍然将继续关注模型的细节选择与实现事宜之间的联系。我们将对用来表述模型的模型元素的3种模式进行划分。这3种模式分别为：实体(Entity)、值对象(Value Object)和服务(Service)。

从表面上来看，把捕获了领域关键概念的对象定义出来是非常容易的；但实际上，要捕捉隐藏在表象之下的概念却是一种很大的挑战。这就要求对模型元素的意义进行明确的区分，并把它应用到设计实践当中，来创建一些特定类型的对象。

一个对象所代表的事物是一个具有连续性和标识的概念(可以跟踪该事物经历的不同状态，甚至可以让该事物跨越不同的实现)，还是只是一个用来描述事物的某种状态的属性？这就是实体与值对象最基本的区别。明确地选用这两种模式中的一种来定义对象，可以使对象的意义更清晰，并可以引导我们构造出一个健壮的设计。

另外，领域中还存在很多的方面，如果用行为或操作来描述它们会比用对象来描述更加清晰。尽管与面向对象建模理念稍有抵触，但这些最好是用服务来描述，而不是将这个操作的职责强加到某些实体或值对象身上。服务用来为客户请求提供服务。在软件的技术层中就有许多服务。服务也会在领域中出现，它们用于对软件必须完成的一些



活动进行建模，但是与状态无关。

有时我们必须在对象模型中采取一些折衷的措施——这是不可避免的，例如利用关系数据库进行存储时就会出现这种情况。本章将会给出一些规则，当遇到这种复杂情况时，遵守这些规则可以使我们保持正确的方向。

最后，我们对模块(Module)的讨论可以帮助理解这样的观点：每个设计决策都应该是根据对领域的正确理解来作出。高内聚、低关联这种思想往往被看成是理想的技术标准，它们对于概念本身也是适用的。在模型驱动的设计中，模块是模型的一部分，它们应该能够反映出领域中的概念。

本章会提到所有这些用来表示软件模型的构造块。这些思想都是一些常规方法，从中得出的建模和设计思路早已有文献记载。但是，本书将把它们组织到一个上下文之中，以使帮助开发人员来创建详细的组件。这将有助于我们以领域驱动设计为中心来处理更大的建模和设计问题。此外，对基本原理的认识将使开发人员在不可避免的权衡过程中把握好方向。

## 5.1 关联

建模与实现之间的相互影响使得对象之间的关联变得特别难以处理。

对于模型中的每个可导航(traversable)的关联，在软件中都会提供一种机制来维护与之相同的特性。

一个描述客户和销售代表的关联的模型对应两件事情。一方面，它把开发人员认为的两个现实个体之间的关系抽象出来；另一方面，它又相当于两个 java 对象间的一个对象指针，或者是某个数据库查询以及某些类似实现的一种封装。

例如，一对多的关联可以在一个实例变量中用集合来实现。但是，设计也不必非要这么直接。可能没有集合，那么我们可以使用另一种做法，通过查询数据库来得到相应的记录，然后根据这些记录将对象实例化。这两种设计都可以映射到同一个模型上。在设计中必须明确指明一种特定的遍历机制，并且这种机制的行为与模型中的关联应该是致的。

在现实世界中，存在大量多对多的关联。大多数关联都存在双向关系。在我们讨论和探索领域时得到的早期形式的模型也确实存在这样趋势。但是，双向关联会给实现和维护带来很大的困难。而且，双向关联极少能将联系的本质体现出来。

至少有 3 种方法可以使得关联更易于控制。



- 指定一个导航的方向
- 通过加入限定符(qualifier)来有效地减少关联的多重性(multiplicity)
- 清除不必要的关联

尽可能地约束关联是非常重要的。一个双向关联意味着，只有这两个对象同时放在一起时才能被理解。如果应用并不要求在两个对象间进行双向交互，那么指定一个导航方向可以降低对象的相互依赖性，并且使设计得到简化。充分地理解领域可以克服一些主观偏见。

美国和其他国家一样有过许多位总统，这就是一个双向的一对多的关联。但是，我们很少会在提及乔治·华盛顿这个名字时，问“他是哪个国家的总统？”。实际上，我们可以将这种双向关联简化为一个单向关联，其导航方向为“从国家到总统”。如图 5-1 所示这种改进真实地反映了领域的内在实质，同时使设计更加实用。它反映出，领域中关联的一个方向比另一个方向更有意义、更重要，也保持了 Person 类对 president 这个基础概念的独立性。

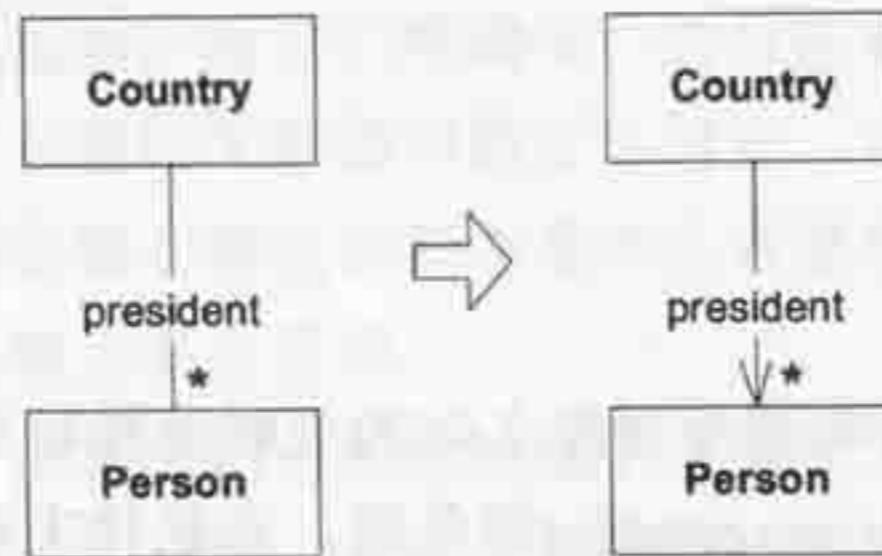


图 5-1 某些导航方向反映出领域中自然存在着的偏向

经过深入的理解后往往可以得到一个“限定的”关系。(让我们)再来深入研究总统的例子，我们会发现一个国家在一个时期(period)只会有一个总统(除非是内战时期)。这个限定条件将一对多关联简化为一对一关联，并且作为一条明确的重要规则包含在模型中，如图 5-2 所示。美国 1790 年时的总统是谁？乔治·华盛顿。

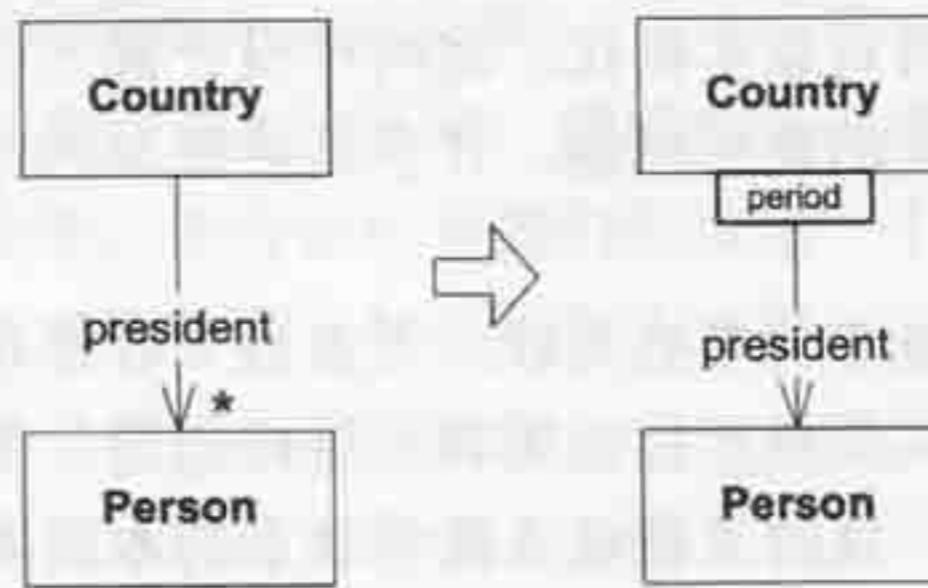


图 5-2 对关联进行约束，能够反映出更多领域知识，并使设计更加实用



对一个多对多关联的导航方向进行约束，会将它有效地简化为一对多这样更简单的设计。

坚持对关联进行严格的约束，如实反映领域中的使用偏向，不仅仅会让这些关联更容易交流、简化其实现难度，还能突出剩下的双向关联的重要性。当一个双向关系是领域中的一个语义特性时，当它是应用系统的功能必须实现的需求时，我们就要保留双向的导航来体现它。

显而易见，最极端的简化方式是：如果关联不是手头任务的本质，或者不能反映出模型对象的基本含义，那么它就应该被完全取消。

#### 示例：经纪账户 Broker Account 处理中的关联

在图 5-3 所示的模型中，Brokerage Account 对象的 Java 实现如下所示：

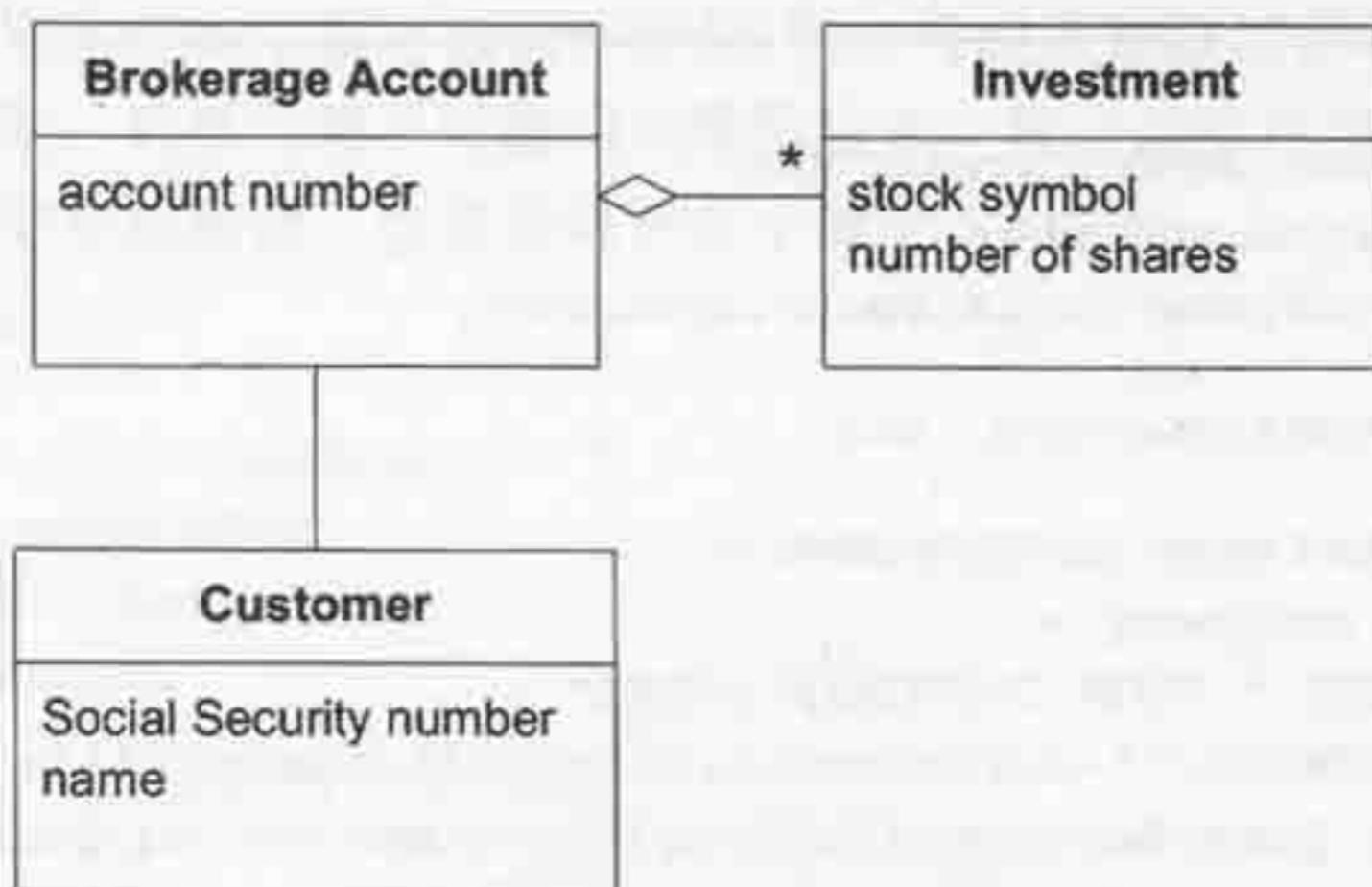


图 5-3 经纪账户示例

```

public class BrokerageAccount {
    String accountNumber;
    Customer customer;
    Set investments;
    // Constructors, etc. omitted
    public Customer getCustomer() {
        return customer;
    }
    public Set getInvestments() {
        return investments;
    }
}
  
```



## 第Ⅱ部分 模型驱动设计的构建块

但是，如果我们需要另一种遵循于该模型的实现：从关系数据库中获取数据的话，可以用如下方式：

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<Investment> getInvestments() {  
        String sqlQuery =  
            "SELECT * FROM INVESTMENT WHERE" +  
            "BROKERAGE_ACCOUNT='"+accountNumber+"'";  
        return QueryService.findInvestmentsFor(sqlQuery);  
    }  
}
```

注意，QueryService 是一个实用类，用来从数据库中获取一行一行的信息并创建对象，它只是用来说明这个例子的，在真正的项目中并不一定是个好设计。

我们可以通过限定 Brokerage Account 与 Investment(投资)的关联来对模型进行精炼，简化它们之间的多重性。也就是说一笔投资只能对应于一支股票(stock)，如图 5-4 所示。

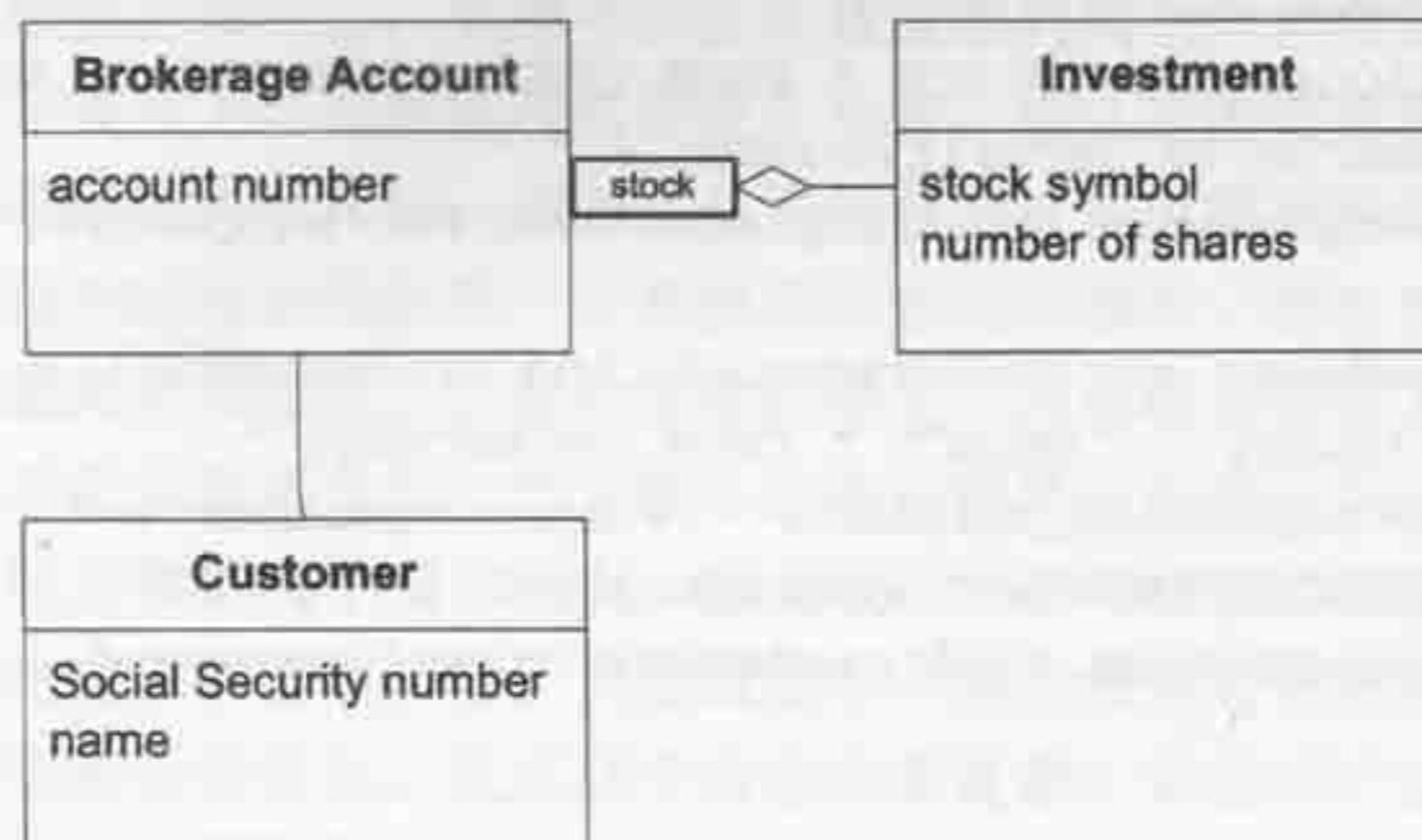


图 5-4 关联约束

不是在所有的业务活动中都可以这样简化(例如可能需要跟踪多笔投资)。但是,无论是什么特定的规则,只要发现了对象间关联的约束,就应该将它包含到模型和实现中来。这些约束使模型更加精确,也使实现更易于维护。

Java 实现变成:

```

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);
    }
}
  
```

基于 SQL 的实现为:

```

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);  
}
```

仔细地提炼并约束模型中的关联，可以引导我们沿着模型驱动设计(MDD)的方向前进。现在让我们回头研究对象本身。(对模型元素的模式进行)特定的区分可以使模型更加清晰，而且更便于实现。

## 5.2 实体(又称引用对象)



许多对象不是由它们的属性来定义，而是通过一系列的连续性(continuity)和标识(identity)来从根本上定义的。

一个女房东起诉了我，要求赔偿她巨额财产损失。我收到的起诉书上描述：公寓的



墙壁满是窟窿，地毯沾满污渍，水槽中的脏水散发出的腐臭气体导致厨房的墙纸脱落。法院的文书中认定我作为房客应该对这些破坏负责，而认定的依据就是我的名字和我现在的住址。这让我感到莫名其妙，因为我从来没有去过那所遭到破坏的房子。

不一会儿，我开始意识到这是一个典型的认错人的情况。我打电话给起诉人，并告诉她弄错了，但是她不相信我。以前的房客已经几个月都没有消息了。我怎样才能证明我和那个破坏房子的家伙不是同一个人呢？在电话簿里只有我叫 Eric Evans。

还好，还是电话簿证明了我的清白。因为我已经在现在的公寓里住了两年，我问房东是否可以找到先前那年的电话簿。当她找到后，并且发现有同名的人时，她才意识到我不是那个她想要起诉的家伙。最后这个房东向我道歉，并撤销了起诉。

计算机没有这么足智多谋，如果在软件系统中认错标识，将会导致数据的毁坏以及程序的出错。

我在这里要提到一些特殊的技术挑战，但是让我们首先来看一个普遍的现象：许多事物是通过它们的标识来定义的，而不是它们的属性。让我们继续用非技术的例子来观察：在我们的传统观念中，每个人都有他的标识，陪伴他出生到死亡，甚至是死了以后都不会改变。而每个人的物理属性会经历很多转变，最终会消失；人的名字可能发生变化；经济往来时断时续。没有哪个人或者他的哪个属性不可以改变，但是标识是永恒的。现在的我和 5 岁时的我是同一个人吗？这种哲学问题，对探究有效的领域模型是非常有效的。换句话说：应用的用户难道关心现在的我和 5 岁时的我是否是同一个人吗？

对于一个到期应收帐的跟踪软件而言，普通的“客户”对象会拥有许多鲜明的特点。当客户立即付款了或者把逾期不交的账户移交给一个票据追缴的代理处时，系统都要记录下状态。如果营销部的联络管理软件需要从系统中提取客户数据，那么(跟踪软件中的)客户对象还会完全重现到另一个系统中来。不管是哪种情况，一个客户对象数据都会对应于数据库表中的一条记录。当客户不再执行新的业务时，客户对象会被存档起来，即成为它自身在磁盘中的一个影子。

基于不同编程语言和技术，客户对象的各种形式可能有不同的实现。但是当客户打电话来下订单时，我们需要了解：这个客户有没有拖欠的账务呢？他是 Jack(某个特定的销售代表)前几个星期一直在联络的客户呢，还是完全是一个新客户？这对我们很重要。

在一个对象的多种实现形态、多种的存储形式与真实世界的参与者(如打电话的人)之间，必须具有匹配的(一致的)概念性标识，但属性可以是不匹配的。例如，销售代表可能已经在联络管理软件中更新了地址，并由联络管理软件将地址的更新传播到了账务跟踪软件。两个客户可能会出现同名。在分布式软件中，用户可以在不同的地方输入数据，这就需要在系统之间传播数据更新事务，并由不同的数据库异步地对这些事务进行



协调。

对象建模倾向于引导我们将精力集中于对象的属性上。但实体的基本概念就是一种抽象的连续性。这种连续性贯穿了对象的整个生命周期，甚至要经历多种实现形式。

有些对象并不主要是由它们的属性来定义的。它们体现了标识在时间上的延续性，经常要经历多种不同的形态。有时，一个对象与另一个对象有着不同的属性，但它们是互相匹配的；有时，一个对象与其他对象有着相同的属性，但它必须能够跟那些对象区分开来。弄错对象标识会导致数据破坏。

以标识作为其基本定义的对象称为“实体<sup>1</sup>”。在建模与设计时需要做特殊的考虑。在实体的生命周期中，其存在形式和内容可以发生很大的变化，但是必须保证其标识的连续性。必须定义实体的标识，这样就可以有效地对其进行跟踪。实体的类定义、职责、属性、关联都是围绕着它是谁来考虑的，而不考虑它有哪些特定的属性。即使实体并不会发生很大的变化，或者其生命周期并不复杂，我们仍然应该根据语义将它归类为实体，这样可以得到更清晰的模型和更健壮的实现。

当然，在软件系统中绝大多数实体都不是平常所说的的人或者实体。只要一个对象在生命周期中能够保持连续性，并且独立于它的属性(即使这些属性对系统用户非常重要)，那它就是一个实体。这样的对象可以是人、城市、汽车、彩票或银行交易。

另一方面，不是所有模型中的对象都是具有有意义的标识的实体。可能会有人对这个问题产生混淆，因为在面向对象语言中每个对象都可以创建“标识”操作(例如 Java 中的“==”操作)。这些操作判断两个引用是否指向同一个对象，可以通过比较二者指向的内存位置(或其他一些机制)来实现，从这个角度来看，每个对象实例都有惟一的标识。比如说，创建一个 Java 的运行环境或技术框架来把远程对象缓存到本地，在这种技术性领域中，每个对象实例确实是实体。但是，这种标识机制对于其他的应用领域而言基本上没什么意义。标识是实体的一个具有微妙意义的属性，这种属性是不能移交给编程语言的自动化特性来实现的。

我们来看看银行应用中的交易。同一账号中的两笔等额的存款在同一天内被存到同一个账号上，它们仍然是两笔不同的交易，因此它们都是具有标识的实体。另一方面，这两笔交易的“总金额”属性都可能是一些货币对象的实例。像“总金额”这样的值对象是没有标识的，因为在这里不需要把它们严格区分开来。事实上，两个对象可以有相同的标识而具有不同的属性；如果有必要的话，它们甚至可以不是同一个类。当银行客

<sup>1</sup> 模型中的实体与 Java 中的实体 bean 并不是一回事。本来，实体 bean 这种技术多少是想做成一种实现实体的框架，但结果它并没有达到这个目标。绝大多数实体都可以实现为普通的对象。实体是领域模型的一个基本特征，与它们是如何实现的无关。



户拿着银行报表与支票登记簿进行对账时，对账要完成的任务就是对所有具有相同标识的交易进行匹配，即使这些交易是由不同的人在不同的时间里记录的(银行的轧账时间比支票上的时间要迟一些)。支票号码可以为对账提供一个唯一的标识，无论对账是由人还是由机器来执行的。存款和现金提款这两种交易并没有标识号码，处理起来可能更复杂一些，但是它们必须遵守同样的原则：每笔交易都是一个实体，它至少有两种表现形式。

在特定的软件系统之外，譬如银行交易，公寓出租等，标识也有着同样的重要意义。但是，有些标识只有在特定的上下文中才有意义(如计算机进程的标识)。

因此：

如果一个对象是通过标识而不是属性来确定的，那么就在模型中把标识作为这个对象定义的基本要素。保持类的定义简单明了，并着重考虑其生命周期的连续性和惟一性。定义对象的方法要能够把每个不同的对象区分开来，而无需去考虑它的形态和历史。对那些需要通过属性比较来匹配对象的情况保持警惕。确保生成标识的操作能够为每个对象产生一个惟一的结果，这可以通过在标识中附加一个具有惟一性保证的符号来实现。生成的标识可能来自外界，也可能是由使用这些标识的系统随机产生的；不管是用什么方法，标识必须满足其在模型中所具有的惟一性。模型必须对“怎么才是同一个事物”的具体含义作出定义。

标识并不是现实实体所固有的，它之所以意义重大，是因为它非常有用。事实上，现实世界中的同一个事物在领域模型中有可能表示为实体，也有可能不表示为实体。

在一个体育场座位预定的应用程序里，座位和观众都可以看成是实体。在分配座位的时候，每张票都有一个座位号，因此座位是一个实体，其标识就是座位号。这个座位号在体育场里是惟一确定的。座位可能会有许多其他属性，比如位置、价格、视线是否受阻等，但是只有座位号(或者是惟一的排次号和位次号)是用来标识和区分座位的。

从另一方面来看，如果这个体育场是开放式管理(也就是说当观众买了票后，可以随意地坐到任何位子上)，那么就不需要严格地确定座位了，只有座位的总数才是关注的重点。尽管椅子上也标记有座位号，但在这种情况下，软件系统就没有必要跟踪记录它们了。事实上，如果在建模时还是将票据与特定的座位号关联起来，就很可能会出现意想不到的错误，因为在开放式管理的体育场里没有这种约束。在这种情况下，座位就不再是实体，也就无需确定的标识了。

### 5.2.1 实体建模

当对一个对象进行建模时，我们会很自然地想到它的相关属性，对其行为的考察也非常重要。但是实体最基本的职责是保证连续性，以便使之具有清晰、可预见的行为。



要有效地实现这个目的，我们必须保持实体的精简性。对于实体而言，我们关注的重点不是它们的属性或行为，而应该把繁枝茂叶从定义中剔除出去，只留下那些固有的特征，特别是那些用来唯一标识对象，以及经常用来查找和匹配对象的特征。只加入核心概念所涉及到的行为，以及行为需要的属性。此外，想办法将属性和行为转移到与核心实体相联系的其他对象中去。这些对象可能是其他实体，也可能是值对象，我们将会在下一节中讨论这个模式。除了要解决标识问题之外，实体往往通过协调它拥有的对象之间的操作来完成自己的职责。

图 5-5 中的 customerID 是 Customer 实体的标识，也是其唯一的标识。但是电话号码和地址都经常被用来查找或匹配客户。名字不能用来作为人的标识，但是我们经常把名字作为确定人的方法之一。在这个例子中，电话号码和地址属性被移到了 Customer 中，但是在实际项目中，这种选择将取决于领域中的对象是怎样匹配和区分的。例如，如果客户有多个联系电话，那么电话号码就不具备惟一性，因此应该把它放在 Sales Contact 中。

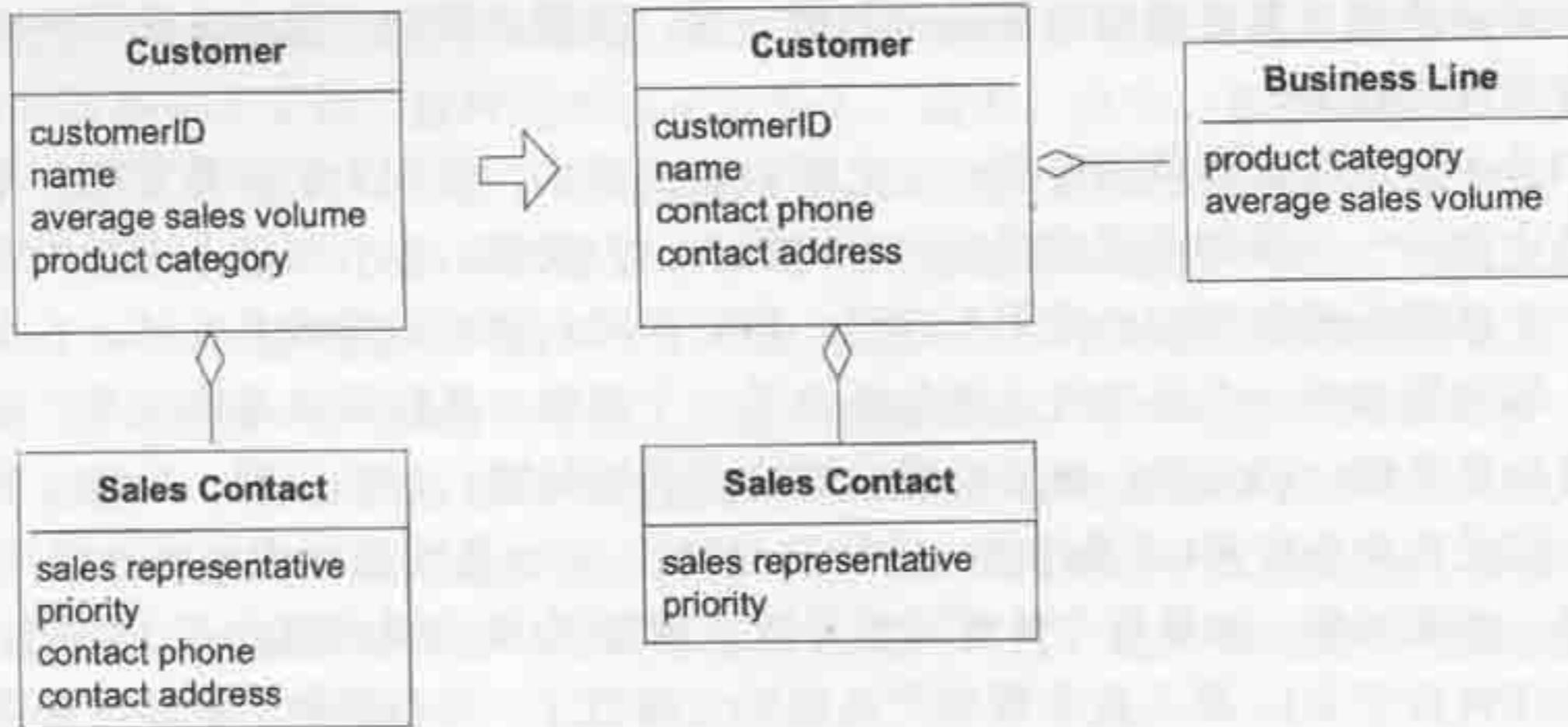


图 5-5 对象的标识属性应该保留在实体中

### 5.2.2 设计标识操作

每个实体都必须有一种可行的方法来建立标识，以便与其他对象区分开来，即使这些对象与它具有相同的描述性属性(非标识属性)。不管实体对象是处于一个分布式系统中，还是已经被保存到磁盘，它的标识属性必须能够在整个系统中保持惟一性。

如前面所提及的，面向对象语言提供了“标识”操作来判断两个引用是否同时指向同一个对象，这是通过比较对象的内存位置来实现的。对于我们来说，这种保证惟一性



的方法是很脆弱的。在大多数对象的持久存储技术中，每当把一个对象从数据库中检索出来时，就生成一个新的对象实例，这就会丢失原来的标识。每当通过网络来传输对象时，就要在目的地生成一个新的对象实例，这也会丢失标识。如果在系统中，同一个对象存在着多个不同的副本，例如在分布式数据库中进行更新传播时，情况将会更加糟糕。

尽管我们可以通过框架来简化这些技术问题，但是最根本的问题仍然存在：怎样判断两个对象在概念上是代表同一个实体呢？标识是在模型中定义的。定义标识必须对领域具有深入的理解。

有时，我们可以用某些数据属性或组合属性来保证（或简单地限制）标识在系统中的惟一性。这种方法为实体提供了一个惟一键值。例如，每天的日报就可以通过报纸的名称、城市的名称以及出版的日期来进行标识（但是要注意日报发行增刊和变更名字的情况！）。

如果对象无法通过属性来构造真正的惟一键值时，我们可以使用另一种常见的方法，那就是为每一个实例都加上一个 ID 标记（一个数字或一个字符串），这个标记在该类中具有惟一性。这种标记一旦生成并作为属性存储在实体中之后，就具有了不变性。即使开发系统不能直接强制确保这种不变性，它也永远不应被改变。例如，当把一个对象保存到数据库中或者从数据库提取重组时，它的 ID 属性要被保护起来。有时技术框架可以帮助我们完成这个过程，但是如果我没有这种框架的话，我们就必须通过工程制度来保证这一点。

ID 通常是由系统自动产生的。生成 ID 的算法必须保证它在系统中的惟一性。对于分布式系统和并发处理而言，这是非常困难的。ID 生成技术已经超过了本书所涉及的范围。在这里之所以要提出来，是为了让开发人员意识到他们需要解决的问题，同时知道怎样把精力集中到那些最为关键的问题上去。关键是要认识到标识是由模型的特定方面决定的。一般说来，只有对领域作了仔细的研究之后，才能找到确定对象标识的方法。

用户可能不需要知道自动生成的 ID。这种 ID 可能只是在内部使用，比如在联络管理系统中，用户会通过人名来查找客户。此时系统就要能够通过一种简单的、无二义的方法来区分两个名字完全相同的联系人。使用惟一的内部 ID 就可以做到这一点。在取出两个不同的联系人之后，系统会把它们都显示给用户查看，但是内部 ID 可能会被隐藏起来。用户将根据联系人的公司、位置等信息来做进一步区分。

最后，还有一些情况下，用户会对系统生成的 ID 感兴趣。当我通过包裹速递服务来运送包裹时，我会获得一个由运输公司的软件产生的跟踪号码，凭这个号码我就可以对包裹进行标识和跟踪。当我预定机票或酒店时，也会得到一个确认码，用来惟一标识这笔交易。

在某些情况下，ID 的惟一性必须能够跨越计算机系统的边界。例如，两家医院相互



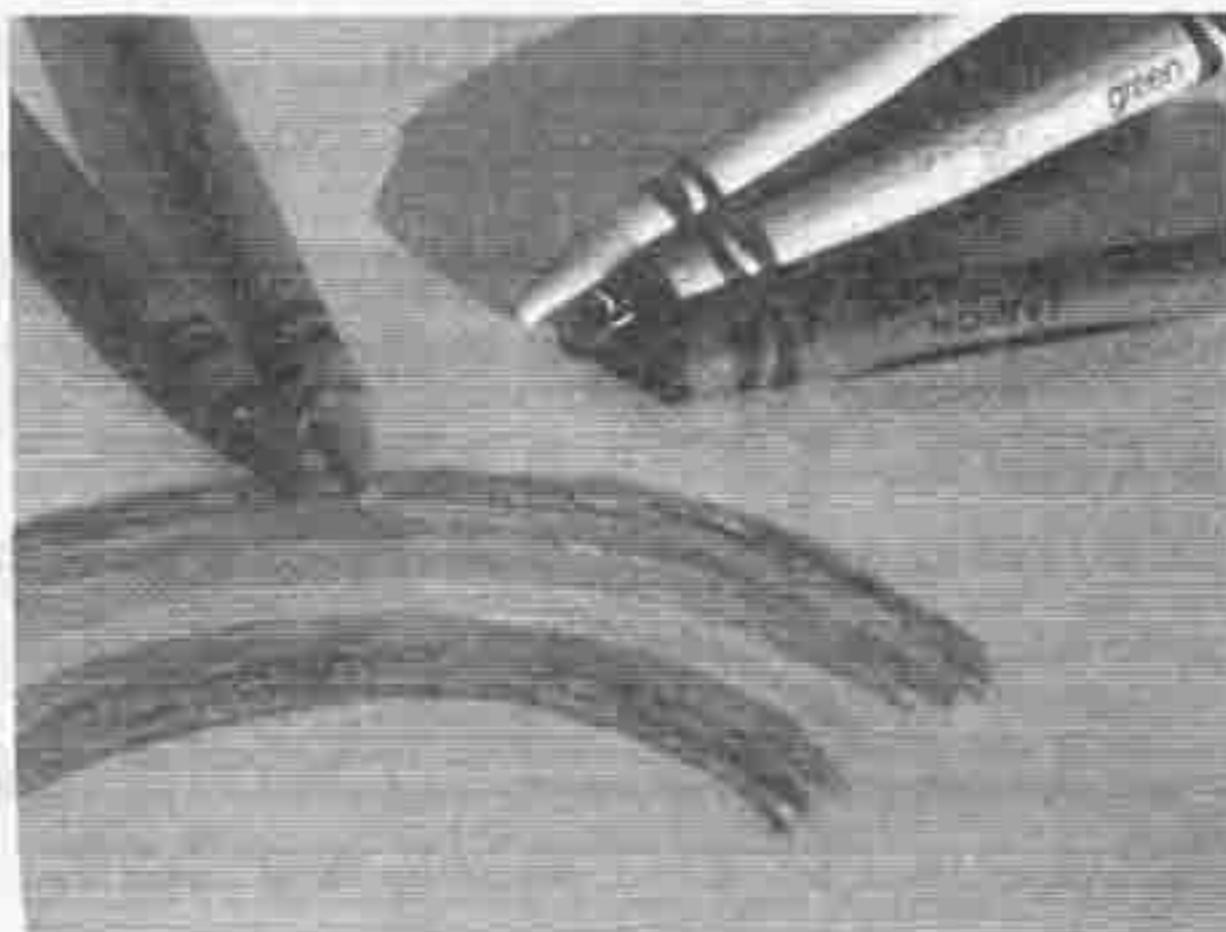
交换病历记录，这两家医院的计算机系统是相互独立的。理想情况下，每个系统都使用相同的患者 ID；但如果它们各自生成不同的标识符号，处理起来就会比较困难。这样的系统常常使用其他公共机构(一般是政府部门)发行的标识。在美国，社会保障号就经常在医院中用来标识不同的个人。这种方法并不是完美的。不是每个人都有社会保障号(特别是小孩和非居民人口)，而且还有一部分人因为隐私原因而反对使用它。

在一些非正式场合下(例如影音出租店)，电话号码可以用来作为客户的标识。但是多个人可以共用同一个电话号码；电话号码可能会发生变化，原来的号码甚至可能会过户给另一个不同的人。

基于以上原因，我们常常使用特别指定的标识(如累计飞行点数)和其他的属性(如电话号码、社会保障号等)来对对象进行匹配和检验。无论在什么情况下，如果应用需要一个外部 ID，那么系统用户就必须提供一个惟一的 ID，同时系统必须为他们提供足够的工具来处理可能发生的异常情况。

由于要面对这些技术问题，我们会很容易忽略一个根本性的概念问题：“两个对象是同一个事物”到底是什么意思？我们可以很容易让每个对象都带上 ID，或者写一个操作来比较两个实例，但是如果这些 ID 和操作并不是与领域中的区分含义相对应，它们只会使整个系统更加混乱。这就是为什么分配标识的操作经常需要由用户来输入。例如支票对账软件会提供一些可能匹配的记录，但是它们是否真的匹配需要由用户来作最终的判断。

### 5.3 值对象





许多对象都没有概念性的标识。这些对象描述了事物的某种特征。

小孩画画的时候，他会关心画笔的颜色和笔尖的粗细。但是，如果有两支具有相同颜色和形状的笔，他并不会去关心他使用的是哪一支。如果笔弄丢了，他可以拿出另一支具有相同颜色的新笔继续画他的画，而不会在意已经换了一支笔。

如果把不同人画的电冰箱拿给小孩看，他很快就可以把自己画的和他姐姐画的电冰箱区分开来。姐弟俩使用不同的标识，因此他们画的画也不同。但是，可以想象，如果他必须对每根构成电冰箱的线条进行一一辨别，那会有多么复杂。那样的话，画画将不再是小孩子游戏了。

在模型中，实体是最显而易见的对象。同时，记录每个实体的标识也相当重要。因此，我们会很自然地想到为所有领域对象分配一个标识。事实上，有些框架的确为每个对象都分配了一个唯一的 ID。

如果为所有领域对象都分配标识的话，系统就必须对所有对象进行跟踪，这会导致失去许多可能的性能优化的机会。因此，我们必须进行仔细分析，把有意义的标识定义出来，并寻求一种可靠的、能够在分布式系统或数据库存储中对对象进行跟踪的方法。同样重要的是，滥用人工标识会使我们误入歧途。它将会扰乱模型，把所有的对象都变成同一个模式。

实体标识的跟踪是非常关键的，但是将其他的对象也加上标识会影响系统的性能，增加分析的工作量。由于所有的对象看上去都是一个模式，也会使得模型变得混乱不堪。

软件设计是一场针对复杂性的持久战。我们必须具体问题具体分析，只在必要的地方作专门的处理。

然而，如果仅仅只考虑那些不具有标识的对象，我们就无法向我们的工具箱或词汇表中增加多少内容。事实上，这些对象在模型中有着它们自己的特征和它们自己的含义。这些对象都是用来描述事物的。

如果一个对象代表了领域的某种描述性特征，并且没有概念性的标识，我们就称之为值对象。值对象就是那些在设计中我们只关心它们是什么，而不关心它们谁是谁的对象。

### 地址是值对象吗

邮购公司的软件需要用地址来证实信用卡的有效性，并用它作为发货目的地。但是，如果一个人的室友在同一个邮购公司下了订单，他们是否住在同一个地方并不重要。因此，地址是一个值对象。

在邮政服务软件中，想要有计划地安排邮递线路，整个地区将组成一个层次性结构，从区、城市、邮政区域、街道，最后到个人地址。这些地址对象可以通过它在层次结构



中的上一级对象来获得它的邮政编码，如果邮政服务决定重新分配邮政区域，所有的地址将会随之改变。在这里，地址是一个实体。

在电力运营公司的软件中，一个地址对应于公司线路和服务的目的地。如果多个室友都申请了电力服务，那么这个公司需要知道这一点，因此地址是实体。我们也可以用另一种方法，在模型中将“住所”关联到运营服务，其中“住所”是一个包含地址属性的实体。此时，地址就是一个值对象。

在许多现代开发系统的基本库中，颜色(Color)是值对象的一个例子。字符串、数字也是值对象(您不用关心您使用的是哪个“4”或者哪个“Q”)这些基本示例都非常简单，但是值对象并不一定简单。例如，在调色系统中，我们可能需要一个功能非常丰富的模型来改进 Color 对象，使之能够与其他 Color 混合，产生出其他的颜色来。这些 Color 对象可能包含了非常复杂的算法，通过协作得到新的值对象(即混合之后的颜色)。

一个值对象可以是其他对象的集合。在房屋设计软件中，我们可以为每种窗户造型构造一个对象。这种“窗户造型”(包括窗户的高度和宽度属性)对象，以及控制这些属性的改变和组合的规则，可以组合到一个“窗户”对象中来。这样的“窗户”是由其他值对象组成的复杂的值对象。它们又可以组合到更大的对象中去，比如说“墙”。

值对象甚至可以引用实体。例如，假设我申请了一个在线地图服务，要求它为我提供从旧金山到洛杉矶之间的最合理的行车路线。这个服务可能会给出一个路线(Route)对象，通过沿海高速公路把旧金山和洛杉矶连接起来。这个 Route 对象可能是一个值，尽管它涉及的 3 个对象都是实体(两个城市和一条高速公路)。

值对象通常作为参数在对象之间交换信息。它们通常是临时对象，在一个操作中创建了之后马上就被丢弃了。值对象经常作为实体的属性和其他值。一个人可以被建模成一个实体，但是人的名字是一个值。

如果我们只关心模型中一个元素的属性，那么就把这个元素划分为值对象。用它来描述它所要表达的那些属性的意义，并提供相应的功能。把值对象看成是不可变的。不要给它任何标识，这样可以避免实体的维护工作，降低设计的复杂性。

如图 5-6 所示，组成值对象的属性应该形成一个总体概念<sup>2</sup>。例如，街道、城市和邮政编码不应该从 Person 对象中分离出去。但它们都是独立整体(地址)的一部分，这样可以使 Person 对象得到简化，同时也使得地址成为一个更加紧凑的值对象。

2 Whole Value 模式由 Ward Cunningham 提出。

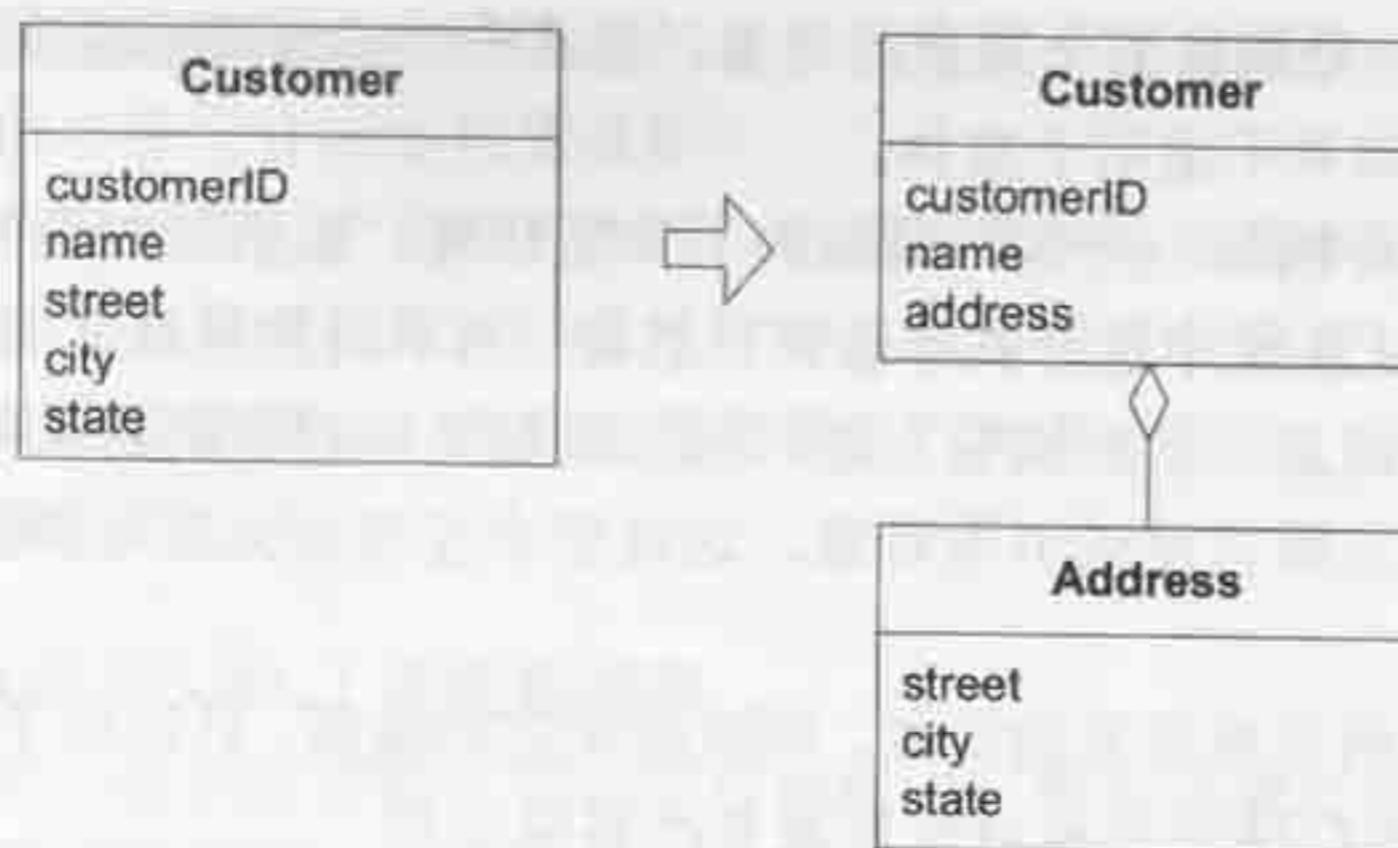


图 5-6 值对象可以为实体提供信息。它应该在概念上是个整体

### 5.3.1 设计值对象

我们并不关心使用的是值对象的哪个实例。由于没有这种约束，我们可以获得更大的自由，对设计进行简化和性能优化。设计值对象时需要对复制、共享和不变性作出选择。

如果两个人有相同的名字，这并不会使他们成为同一个人，也不会让他们俩互换角色。但是表示名字的对象是可以互换的，因为它只与名字的拼写相关。一个 Name 对象可以从一个 Person 对象复制给另一个 Person 对象。

事实上，两个 Person 对象可能不需要它们自己的名字实例。两个 Person 对象可以共享同一个名字对象(用指针指向同一个名字实例)，而无需改变它们的行为和标识。也就是说，这两个 Person 对象行为都是正确的，直到其中某个人改了名字。然后，另一个人的名字也可能会发生改变！为了防止发生这种情况，能够安全地共享对象，值对象必须具有不变性：它不能改变，除非把它整个替换掉。

当一个对象把它的属性作为参数或返回值传递给其他对象时，也会出现这种问题。对于一个脱离了其所有者控制的对象来说，任何事情都是有可能发生的。如果对值修改不当，就会破坏其所有者的不变性约束，从而破坏它的所有者对象。这个问题可以通过传递一个具有不变性的对象，或者传递一个对象副本解决。

为了优化性能，我们可以提供一些额外的选项。这一点可能很重要，因为值对象的数量会非常巨大。房屋设计软件的例子就显示出了这一点。如果每个电源插座都是一个独立的值对象，那么在一个房子的设计中会有上百个插座对象。但是如果所有的插座被看作是可以互换的，我们就可以只使用一个插座的实例，然后用指针来引用它就行了(这是一个 Flyweight(美元)的例子[Gamma et al. 1995])。在大型系统中，这种优化的效果会表现



得更为明显，它通过消除数百万个冗余的对象，使系统性能得到明显改观。这只是一个优化系统的技巧，但它却并不适用于实体。

复制和共享哪个开销低一些呢？这取决于实现环境。虽然复制会产生大量对象而阻塞系统，但是在分布式系统中进行共享会降低性能。当不同的机器之间传递对象副本时，发送方只需发出一个消息，然后接受方就可独立控制复制过来的对象实例。但是如果是共享对象实例，发送方就只发送引用对象，这就要求在每次交互时都要把消息传递回被共享的对象。

为了能够尽量利用共享带来的好处，同时避免它的缺陷，只在以下情况中使用共享：

- 当数据库中的存储空间和对象数量有严格限定时
- 当通信开销不高时(例如在一个中心服务器上)
- 当共享对象具有严格的不变性时

有的语言和环境可以声明属性或对象的不变性，但有的不能。这种特性有助于清晰地表达设计决定，但它并不是必须的。我们在模型中指定的一些特性在大多数现行工具和编程语言中都无法明确声明出来。例如，我们不能声明一个对象为实体，让系统自动对标识操作提供保证。尽管缺少在编程语言中的直接声明，但这并不意味着这么做没有用处。这恰恰说明了，在系统实现中需要更多的限制来维护那些隐性的规则。这也就对命名的约定，文档的选择以及研究的深入提出了更高的要求。

只要值对象是不可变的，在管理上就简单了，除非完全替换，否则没有任何的更改。不变对象可以被自由地共享，例如电源插座。如果垃圾收集机制是可靠的，删除没用的对象就只是释放所有对该对象的引用。如果一个值对象被设计成为不可变的，开发人员可以从纯技术的角度，自由地在共享和复制之间选择系统实现方式。可以在理论上保证，整个程序不会依赖于哪个特定的对象实例。

#### 特殊情况：当被指定为可变时

不变性大大简化了实现的难度，使得共享对象和引用对象能够安全地传递。它保证属性的值不被更改。如果要改变一个属性的值，应该使用一个不同的值对象，而不是去修改现在的这个对象。即使这样，在一些情况下考虑到系统性能，可能会允许值对象的改变。下面这些因素将倾向于在实现中允许更改。

- 如果值经常被更改
- 如果对象的生成和删除的开销很大
- 如果替换(不是修改)会打破集群(像前面讨论到的例子)
- 如果值没有太多的共享，或者共享没有提高集群，又或者一些其他技术方面的原因



重申一遍：如果值的实现是允许更改的，那么它就不能被共享。不管您是否将值共享，都应该尽量将值对象设计成不可更改的。

定义值对象并将它们指定为不可更改时，要遵循下面这条基本原则：在模型中避免不必要的约束，让开发人员可以从纯技术角度对系统性能进行调整。明确地定义必要的约束，让开发人员能够调整设计，保证改变有意义行为是安全的。这种设计调整是一种经常在特殊项目中使用的技术。

### 示例：用 VALUE OBJECT 调整数据库

数据库在最底层，必须将数据存储到物理磁盘上，数据读取时间包括转动磁盘的时间和从磁盘中读取数据的时间。成熟的数据库通常把这些物理地址聚集在一起，这样相关的数据就可以在一次读取操作中获得。

如果一个对象被很多对象引用，这些对象并不会挨得很近(在同一个页面上)，这就要求一个额外的物理操作来读取数据。通过使用复制，而不是共享同一个实例的引用，作为实体属性的值对象可以被存储到相同的页面，当每个实体要使用的时候都可以一次性读取。这种保存同一个数据的多份副本的技术被称作非规范化，如果要求的访问时间比存储空间更重要，或者要求维护简单，就会经常使用这种技术。

在一个关系数据库中，您可能想将一个特殊的值赋给实体中的表，而不是与一个独立的表创建关联。在分布式系统中，在另一个服务中保持一个值对象的引用，将会减慢消息响应速度，所以应该把整个对象的副本传给那个服务。我们可以自由地复制，因为处理的是值对象。

### 5.3.2 设计包含值对象的关联

前面有关关联的大多数讨论不仅可以应用在实体上，对值对象同样适用。模型中的关联越少、越简单就越好。

但是，实体之间的双向关联难于维护，而值对象之间的双向关联却没什么意义。由于缺少对象标识，一个对象指向一个值对象，同时该值对象又指向这个对象，是没有意义的。您最多可以说，它指向一个对象，该对象与指向它的那个对象等同，但是您可能必须在一些地方设置不变量。尽管您可以这样做，并设置指向两边的指针，但很难想象这样的安排会有什么用。只有尽力去完全消除值对象间的双向关联。如果最后模型中的这种关联看上去是必要的，那就要重新考虑一开始就将这个对象声明为值对象的决定是否正确。它可能存在还没有被明确地辨认出来的标识。

实体和值对象是传统对象模型的主要元素，但是注重实效的设计者已经开始使用另一个元素：服务。



## 5.4 服务

有时，服务不仅仅是一个事情。



在一些情况下，最清楚最务实的设计包括那些在概念上不属于任何对象的操作。不是为了强行引入这种概念，而是我们可以沿着问题空间的自然轮廓，在模型中明确地使用服务。

这儿有些重要的领域操作，不属于实体和值对象。它们是一些本质的行为和动作，不是事情，但是因为我们的建模范式是对象，所以无论如何，我们都将努力让它们适合于对象。

现在，最常犯的错误就是太轻易地放弃把这种行为配置到合适的对象中，却逐渐地转移到程序的设计上。但是当我们在一个对象里加入并不适合对象定义的操作，这个对象将失去清楚的概念，并且变得难以理解或重构。复杂的操作会轻而易举地使简单的对象陷入泥潭，混淆它的角色。因为这些行为经常会聚集许多领域对象，协调它们并让它们运作起来。增加的这种职责将创建这些对象和概念的依赖关系，使其能够单独地被理解。

有时，服务会伪装成模型对象，除了完成一些操作之外没有任何意义。这些“实干家”看上去就像是“管理者”。除了它们掌握的操作外，服务没有自己的任何状态，在领域中也没有任何意义。至少这种方式不会把这些特殊的操作和一个真正的模型对象搞混。

领域中的一些概念不能作为模型中的对象来处理。将领域需要的功能强行加给实体和值对象，不仅会破坏模型中对象的定义，而且还会人为地添加没有意义的对象。

服务作为一种接口提供操作，它独立于模型，没有像实体和值对象那样封装状态。



服务在技术框架中是一种通用模式，但是它也可以应用于领域层。

所谓服务，它强调与其他对象的联系。不像实体和值对象，服务完全是根据能够为客户做什么来定义的。服务往往代表一种行为，而不是一个实体，是一个动词而不是一个名词。服务可以有一个抽象的、有意图的定义，这与一个对象的定义有所不同。服务应该还有一个定义好的职责，它的职责和接口被定义为领域模型的一部分。操作名应该来自通用语言，如果通用语言中还没有这个操作名，则应该把它添加进去。调用的参数和返回的结果应该是领域对象。

使用服务时要小心，不能剥夺实体和值对象的所有行为。但是当一种操作确实是一个重要的领域概念时，服务就很自然地成为了模型驱动设计中的一部分。在模型中声明的服务，不是一个并不代表任何东西的对象，它的独立操作不会误导任何人。

一个优秀的服务具备3种特征。

- 与领域概念相关的操作不是实体和值对象中固有的部分
- 接口根据领域模型中的其他元素来定义
- 操作是无状态的

这里的无状态是指任何一个客户都可以使用服务的所有实例，而不管这个实例的来源。服务的执行将使用全局可访问的信息，甚至改变这些信息(这也就是说它可能有副作用)。但是服务不保留影响自己行为的状态，而绝大多数领域对象是要保留这些状态的。

当领域中的一个重要进程或转换操作不是实体和值对象本身的职责时，把操作作为一种独立的接口加入模型，并声明为服务。根据模型中使用的语言来定义接口，保证操作名是通用语言中的一部分。使这个服务变成无状态的。

#### 5.4.1 服务和分隔的领域层

这种模式只关心那些在领域中有着重要意义的服务，当然，服务不只是应用在领域层上。要仔细区分领域层和其他层上的服务，仔细分解它们的职责，将它们明确区分开来。

文献中讨论的绝大多数服务是纯技术上的，属于基本结构层。领域层和应用层的服务要与基础构架层的服务合作。例如，银行可能需要一种应用，当客户的账目余额低于规定界限时，银行会向客户发送一封电子邮件。封装这种电子邮件系统(也可以是另外一种通告方式)的接口是基础结构层上的一个服务。

要区分应用层服务和领域层服务是非常困难的。应用层负责拟订通告。领域层负责判断客户的账目余额是否达到了一个阈值，如果“账目”对象不满足这个条件，这个任务就不会调用服务。这个银行应用可以用来处理资金转账。如果设计一个服务，负责处



理资金转账的借贷关系，这种功能就属于领域层。资金转账在银行领域语言里有它的意义，它包括了基本的业务逻辑。技术层上的服务则根本没有任何的业务含义。

许多领域层和应用层的服务都是建立在大量的实体和值对象之上，有点像组织领域功能完成某种任务的脚本。实体和值对象常常因为粒度太细而不能为访问领域层的功能提供便利。这里我们遇到了在领域层和应用层间的一个非常细小的区别。例如，如果银行应用可以将我们的交易转化并输出成电子表格文件供我们分析，这种输出就是一种应用的服务。在银行领域中，“文件格式”没有任何意义，也不包括任何业务规则。

另一方面，将资金从一个账户转移到另一账户的功能是一个领域服务，因为它包含了明显的业务规则(例如，将资金从正确的账户中存入或取出)，并且因为“资金转账”是一个有意义的银行业术语。在这种情况下，服务并不对它本身进行操作，它会要求两个账户对象来完成大部分的工作。但是将“转帐”操作赋予账户对象非常难于操作，因为操作包含了两个账户和一些全局规则。

我们可能希望创建 Funds Transfer 对象来代表两个账户，并附加上一些转账规则和历史记录。但是我们还需要调用银行网络的服务。在大多数开发系统中，在领域对象和外部资源之间直接使用接口，将会使得系统难于使用。我们可以用一个外观来封装这种外部服务，根据模型接收输入，可能把一个 Funds Transfer 对象作为结果返回。但是无论我们使用什么中间件，甚至它们并不属于我们的系统，这些服务负责执行领域中的资金转账功能。

### 将服务分层

---

#### 应用层

##### 资金转账应用服务

- 读取输入(例如 XML 请求)
- 发送消息给领域服务，要求处理
- 监听确认消息
- 决定用基础结构层的服务发送通告

---

#### 领域层

##### 资金转账领域服务

- 必要的账户和分类账对象的相互作用，完成正确的提取和存入
- 确认转账结果(转账是否被允许或拒绝等)

---

#### 基础结构层

##### 发送通告服务

- 由应用选择通告方法，发送电子邮件、信件或者通过其他通信途径



#### 5.4.2 粒度

尽管这个模式着重讨论了如何表示服务的概念，但是，像将客户实体和值对象分离一样，该模式作为控制领域层接口粒度的方法也是很有用的。

中等粒度、无状态的服务在大型系统中能够很容易地被重用，因为它们把重要的功能封装在一个简单的接口里面。同样，细粒度对象会导致消息在分布式系统中低效率地传递。

正如以前讨论过的，细粒度的领域对象会导致信息从领域层转到应用层。因为在应用层领域对象的行为是协调的，所以，需要应用层来处理复杂的领域对象交互作用，从而使领域信息从领域层上丢失，而出现在应用代码或用户接口代码中。明智地引入领域服务可以有效地帮助我们分清领域层和应用层的界限。

这种模式的接口易于客户控制并且适用性强。它提供了适当的功能，对于封装大型系统或者分布式系统中的组件非常有用。有时，服务是描述领域概念最自然的方式。

#### 5.4.3 访问服务

分布式系统体系结构，例如 J2EE 和 CORBA，为服务提供了特殊的发布机制。除了传统的使用方法外，还添加了分布和访问功能。但是这样的框架并不总是会在项目中使用，就算是使用了，当实施逻辑划分时，也只是大材小用。

访问服务的方法并不重要，重要的是如何明确分离相关职责。一个“实干家”对象或许很适合作为服务的接口实现。很容易编写一个简单 Singleton(Gamma et al. 1995)提供访问。编码约定清楚地表明，这些对象只是服务接口的传递机制，是没有意义的领域对象。只有在真正需要分布系统或者利用框架功能的时候，才会使用复杂的体系结构。

### 5.5 模块(包)

模块是一种老的、确定的设计元素。这里有技术方面的考虑，但主要还是出于模块化的考虑。模块给人们提供了两种分析模型的方法：以模块为单位考虑它的实现细节，或者从全局分析模块之间的联系，而不用关心细节。

领域层的模块应该是模型中有意义的一部分，从较大的方面来描述领域。

每个人都使用模块，但是很少有人将它们看作是模型中一个完整的部分。在开发人员使用的技术框架中，代码被分门别类地进行划分。甚至那些经常重构模块的开发人员，也常常使用以前项目中设计好的模块。

的确应该实现模块间的低关联与模块内的高内聚。对于关联和内聚的解释，听上去



往往像是技术上的衡量标准，用来机械地判断模块相互联系和相互作用的分布情况。除代码外还将概念都分成了模块。一个人在某个时刻所能考虑到的事情是有限的(低关联)。不连贯的思维是很难被理解的，也很难表现出统一的想法(高内聚)。

低关联和高内聚是一种通用的设计原理，模型中的独立对象和模块都要尽可能依照这个原则进行设计，但是它们在大粒度的建模和设计中更加重要。这些术语由来已久，在 Larman 1998 中以模式的形式解释这两个原则。

只要两个模型元素被分割成两个不同的模块，它们的联系就没有以前那么直接了，这对于了解它们在设计中所处的位置来说就增加了难度。两个模块间的低关联不仅最大限度地降低了开销，而且使得可以用最少的引用来分析模块的内容。

同时，一个设计优秀的模型，它的元素能够很好地协同工作，适当地选择模块，将那些紧密联系的模型元素集中到一起。这些高度内聚的对象，具有相关的职责，可以将建模和设计的工作集中在单独一个模块中，从而把复杂度限制在一个范围内，使开发人员更容易处理。

模块和更小的元素应该共同发展，但是通常不是这样的。模块是对象早期的一种组织形式。之后，对象往往在现有模块定义的范围内进行改变。重构一个模块比重构一个类要付出更多的工作，会有更大的破坏性。因此不能经常重构模块。但是，正如模型对象刚开始从简单具体逐渐转变为能够反映模型本质的对象一样，模块则变得抽象起来。让模块反映出对领域的理解变化，将使得在领域中的对象能够更自由地发展。

和领域驱动设计里的其他所有部分一样，模块是一种通信机制。划分对象其实就是在选择模块。当您将一些类放到一个模块中时，您得让其他的开发人员明白把它们放在一个模块中的原因。如果您的模型是一本书，那么模块就是这本书的章节。模块的名称可以表达出它的意义。这些名称要加到设计时使用的通用语言中。您可能会对一个业务专家说，“现在让我们谈谈 customer 模块吧”，这样就建立了您们交流的环境。

因此：

选择的模块应该能够描述系统，包含的概念集应该是内聚的。这样通常会在模块间产生松散关联，如果不是，就想办法改变模型，重新整理概念。或是寻找一种概念，构成模块的基础，以一种有意义的方式把元素集中起来。寻找概念中的低关联，能够理解并解释它们之间的相互独立性。对模型进行精练，直到模型能够根据高级的领域概念进行划分，同时分离相应的代码。

给模块命名，并把这些模块名加到通用语言中。模块和它们的名称应该反映出对领域的理解。

考虑概念上的联系并不是一种技术手段。概念上的联系和技术上的联系是同一问题



的两个不同方面，都必须要实现。只有对模型进行仔细的思考后，才会产生一种更好的解决办法，而不是轻易就能得到的。当需要进行权衡时，一定要保证概念清晰，即使意味着模块之间会产生更多引用，或者模块的改动会引起一系列相关的影响。如果开发人员能够理解模型的这些变化，就能处理好这些问题。

### 5.5.1 敏捷的模块

模块需要与模型中其他模块共同发展。这意味着重构模块要同模型和代码的重构一同进行。但是这种重构不会经常发生。修改模块往往需要更改大量的代码。这样的变化可能会破坏开发团队之间的交流，甚至会破坏开发工具，例如源代码控制系统。结果，模块结构和模块名通常比类更早地反映模型的构成。

刚开始选择模块时，不可避免的错误导致了模块间的高度关联，使得模块难以被重构。如果不进行重构，问题就会越来越严重。要克服这种困难，就只有根据经验和问题的情况，重新组织模块。

一些开发工具和编程系统会使得这个问题更加恶化。不管实现是基于什么样的开发技术，我们都应该尽量避免重构模块，避免与其他开发人员沟通时出现混乱。

#### 示例：用 Java 打包编码约定

在 Java 中，在一些独立的类中必须声明导入(相关性)。建模者多半关心模块之间相互依赖的关系，但 Java 不能表示这种关系。一般的编程约定鼓励导入特定的类，产生如下代码：

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

在 JAVA 中，虽然需要向类中进行导入，但一次至少可以导入整个模块，这表明模块是具有高度内聚性的单元，同时减轻了修改模块名的工作。

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



这个技巧意味着将两种方式(类依赖于模块)混合在一起，但表达的不仅仅是前面的那一长串类，还表达了模块之间的依赖关系这个意图。

如果一个类确实要依赖另一个模块中的类，而且本地模块与那个模块之间看上去没有一种概念上的依赖关系，那么可能要移走一个类，要么就重新考虑模块之间的关系。

### 5.5.2 基础结构驱动打包的缺陷

我们的打包决策很大程度依赖于技术框架。有些决策是有帮助的，有些则是要坚决反对的。

一个非常有用的框架标准是分层体系结构，将基础结构和用户接口代码分别放到独立的一组模块中，而将领域层放到它自己那组模块中。

另一方面，分层体系结构可以分成几个模型对象来实现。一些框架产生的分层将一个领域对象的职责分散到多个对象，并且把这些对象放到各自的模块中。例如，J2EE 的一个基本惯例是，将数据和数据的访问放到一个“实体 bean”中，而将相关的业务逻辑放到一个“会话 bean”中。随着每个组件实现的复杂度的增加，这种分离很快就使对象模型失去了内聚力。对象的一个最基本概念就是用操作数据的逻辑来封装数据。这种分层实现并不是致命的，因为可以把这两个组件看作是共同实现了一个模型元素。但是实体 bean 和会话 bean 通常被放到不同模块中，使情况变得更糟糕。这样看来，要将各种对象在思想上恢复成一个概念实体是非常困难的。我们失去了模型与设计之间的联系。最好的方法就是使用更大粒度的 EJB 而非实体对象，减少分层带来的损失。但是小粒度的对象也经常被分层处理。

例如，我们就遇到过这样的情况，在一个项目中，所有概念对象被分成 4 层。每一层都有一个合适的理论基础。第一层是数据层，负责映射和访问关系数据库。第二层是处理对象在各种情况下的基本行为。第三层是用来增加应用的功能。第四层则是一个公共接口层，屏蔽下面三层的实现细节。这种安排可能有点太复杂了，但是各层的定义都很合适，概念清晰，划分明确。我们可以从意识上将所有具体的对象组织成一个概念对象。这种分层有时是非常有用的，特别是对代码进行处理后就会消除很多混乱。

但是最重要的是，框架要求每层都被封装到一组模块中，并且根据识别习惯来命名层。我们必须全心全意投入到分层的工作上来。所以，领域开发人员总是避免构建太多的模块(每个都要乘 4)，而且几乎不用对它们修改，因为重构模块是不允许的。更麻烦的是，搜索一个概念类的所有数据和行为很困难(还要考虑分层中的一些间接关系)，所有开发人员也就不会有太多的精力去考虑模型了。被交付用户使用的这种应用，用一个有缺陷的领域模型来完成应用对数据库的访问请求，只提供了少数几个服务。因为这些代



码不能清楚地展现模型，使开发人员不能对它进行处理，所以模型驱动设计的优势受到了限制。

这种框架设计必须要解决好两个问题。一个是合乎逻辑的划分：一个对象负责数据库访问，另一个负责业务逻辑等。这样的划分可以更容易地理解各层的功能(从技术方面)，并且能够更容易地划分层次。问题是应用程序开发的成本很难估计。本书不是一本介绍框架设计的著作，所以我不准备在这里讨论解决这个问题的方法了，但是确实有一些方法可以解决这个问题。就算是我们无法回避这个问题，最好也是为了获得一个更加内聚的领域层而去权衡这些优势。

打包方案的另一动机就是分层。如果代码在不同的机器上时，如何实现分层还存在争议。一般是不会出现这种情况的，但是如果需要的话，需要找到这种灵活性。在一个项目中，我们希望能够从模型驱动设计中获得帮助，但是代价可能会很大，除非是用来解决一个很紧急的问题。

技术打包方案会带来两种代价。

- 如果框架的划分惯例是分离那些实现概念对象的元素，那么这部分代码将不再用来解释模型。
- 划分的方法如此之多，如果框架都使用了，那么领域开发人员就不可能把模型分成有意义的小块了。

最好是坚持简单化原则。选择最少的技术划分规则，这些规则对采用的技术来说很重要，确实对开发有所帮助。例如，将复杂的数据从对象的行为中分离，使对象更容易进行重构。

除非真的是想将代码分布到不同的机器中，否则将实现一个概念对象的所有代码都包括在同一个模块中，即使这些代码实现的不是同一个对象。

我们能够得出与先前标准相同的结论：“高内聚，低关联”。一个实现业务逻辑的“对象”与另一个负责数据库访问的“对象”，它们之间的联系非常多，关联性非常高。

在框架设计或者仅仅是一个公司或项目采用的惯例中存在的缺陷，由于掩盖了领域对象的固有内聚性，可能会破坏模型驱动设计，但是底线是一样的。一些限制条件，例如需要大量模块，排除了其他满足领域模型需要的打包方案。

使用打包技术从其他代码中分离领域层。否则，尽可能地让领域开发人员根据他们的模型和设计来打包领域对象。

如果代码是基于说明性的设计(参见第10章)，则是一个例外。在这种情况下，开发人员不需要阅读代码，并且最好将它放到一个单独的模块中，不会和开发人员要实际处理的设计元素产生混淆。



随着设计越来越复杂，模块化的工作量也越来越大。本节对这个问题进行了简单的考虑。第IV部分“策略性设计”将提供一些打包和划分大型模型和设计的方法，并介绍一些帮助人们理解这些策略的方法。

领域模型中的每个概念都应该对应着一个可实现的元素。实体、值对象以及它们的联系，同时还包括一些领域的服务和模块，都是模型与实现之间直接对应的地方。显然，对象、指针和检索机制在实现中都必须直接映射模型元素。如果它们没有这样做，要么删除代码重新编写，要么修改模型，两种做法都可以。

不要在模型中给领域对象添加任何与概念关联不紧密的元素。这些设计元素都有自己的任务：描述模型。为了使系统正常运行，必须执行一些领域相关的职责，管理其他的数据，但是它们都不属于这些对象。在第6章中，我们将讨论一些辅助对象，完成领域层的技术职责，例如定义数据库的搜索和对复杂的对象进行打包。

本章介绍的4种模式提供了对象模型的构成元素。但是模型驱动设计并不一定要将所有的东西都添加到对象模型中。一些工具还支持其他的模型范式，例如规则引擎。项目必须在它们之间权衡，选择一个实用的范式。这些工具和技术意味着在设计中放弃模型驱动设计方法，而不是模型驱动设计的另外一种选择方法。

## 5.6 建模范式

模型驱动设计要求实现技术与建模范式相一致。许多范式都曾拿来做过试验，但是只有少数几个范式得到了广泛实际应用。现在主要的范式是面向对象设计的，大多数复杂的项目现在都开始使用对象了。这种优势来自几个方面：有些是对象自身的优势，有些是开发环境的原因，还有些是因为被广泛使用的原因。

### 5.6.1 对象范式的优势

开发团队选择使用对象范式的许多原因并不是出于技术上的考虑，甚至也不是出于对象本身性质的考虑。但是对象建模正在简单和复杂之间创造一种有效的平衡。

如果建模范式太深奥了，就不会有足够的开发人员能够掌握它，就不会得到正确的应用。如果团队中的非技术人员如果连范式的入门知识都不能掌握，他们就不可能理解模型，模型中的通用语言将会失去使用的机会。面向对象设计的基本原理在绝大多数人看来都很自然，容易理解。即使一些开发人员遗漏了建模的一些细节，甚至连非技术人员也能够理解对象模型的图型。

对象建模的概念不仅非常简单，而且已被证明通过这些概念足以把握重要的领域知



识。所以，在一开始便获得了不少开发工具的支持，帮助我们在软件开发中快速建立模型。

今天，对象范式由于技术成熟而且被广泛采用，具有明显的环境优势。如果没有成熟的基础结构和工具的支持，一个项目也只能是一种技术上的研究与开发(R&D)而已，不仅延缓了应用的开发，而且还会带来技术风险。一些技术不能与其他技术很好协作，因此不可能被集成到工业标准的解决方案中，从而迫使开发人员重新开发通用的应用。但这些年，许多问题由于对象的出现而被解决，或者由于被广泛使用而变得无关紧要(现在依赖其他的主流对象技术进行集成)。绝大多数新技术提供了在流行的面向对象平台上集成的方法，使集成变得更加容易，甚至可以选择将基于其他建模范式的各种子系统进行集成(本章稍后将对这些范式进行讨论)。

开发团队和设计文化自身的成熟同样重要。一个采用了新颖范式的项目可能很难找到这样一个开发人员，具备这种技术的专业知识，或者具备在这种范式中创建有效模型的经验。花费大量的时间对开发人员进行培训几乎不可行，因为使用的范式和技术的模式还没有形成。可能这个领域的先锋可以有效地应用这种新范式，但是还没有将他们的见解以一种易于理解的方式公布出来。

对象早已被大量的人所理解，他们包括开发人员、项目负责人以及项目中其他所有的专家。

我们用 10 年前的一个面向对象项目来说明工作在一个不成熟范式上，会有什么样的风险。在 90 年代早期，这个项目使用了几项前沿的技术，包括面向对象的大规模数据库。这是一个令人激动的开发项目。开发团队的成员自豪地告诉来访者，我们正在利用这个技术来开发最大的数据库。当我参与到这个项目中时，有不同的开发团队正在研究面向对象设计，能够很轻松地将他们的对象存储到数据库中。但是我们逐渐地意识到，我们的测试数据只是利用了这个数据库容量的一个部分而已，实际的数据库其实还要大上十几倍。实际的处理量也高出十几倍。难道使用这种技术就不能够实现这种应用吗？难道我们使用它的方法不正确吗？这种技术让我们感到莫测高深。

幸运的是，我们能够找到一个掌握了这门技术的人，使我们摆脱这个困境。他提出了价格，我们也接受了。导致这个问题的出现有 3 个方面。第一，这个数据库提供的基础结构不能满足我们的需要。第二，存储细粒度对象的代价比我们预计的要高得多。第三，对象模型中，有些部分的依赖关系非常混乱，较小规模的并发事物处理会出现竞争问题。

在这位专家的帮助下，我们改进了基础结构。整个团队现在都意识到了细粒度对象



产生的影响，并着手寻找更适合这个技术的模型。我们都对在模型中限制关系网的重要性加深了认识，开始用这种新的理解构造更好的模型，降低相关聚合的联系，在它们之间实现较低的关联。

模型改造加上前面不成功的开发，浪费了我们好几个月的开发时间。这已经不是因为采用不成熟的技术而第一次导致开发受到挫折，我们经验上的缺乏也使得学习这门技术的过程非常艰难。令人沮丧的是，最后项目的开发目标缩小了，也变得保守起来。直到现在他们仍然使用这种特殊的技术，但是对某些应用来说，可能不会从这项技术中获得什么好处。

10年后，面向对象技术已经相对成熟了。大多数普通的基础结构需求可以由该领域的现成解决方案来满足。关键任务工具通常会由多个主要的开发商提供，或者来自稳定的开放源代码项目。许多基础结构的组成部分被广泛地使用，早已被许多人掌握，而且还有大量的书书籍对它们进行介绍。人们对这些确定技术的局限性也相当清楚，所以具有见识的开发团队极少可能会被别人超过。

其他那些令人感兴趣的建模范式就没有这么成熟了。有一些范式太深奥，不容易掌握，除非在一些小范围的特殊项目中，否则是根本不会被采用的。这些范式都是有潜力的，但是它们的技术基础结构仍然有缺陷或者不稳定，几乎没有人知道怎样为它们创建好的模型。这些范式已经出现很长一段时间了，但是仍然没有得到很好地应用。

这就是为什么许多尝试模型驱动设计的项目，目前会很明智地使用面向对象技术作为系统的核心。它们将不会被限制在只有对象的系统里，因为对象已经成为软件行业的主流，集成工具可以将目前使用的所有技术结合在一起。

但是，这并不意味着人们应该永远将他们局限于对象。跟随主流进行开发总会提供一些保障，但是并不是只能采取这一种方式。对象模型解决了大量实际的软件问题，但是模型是由离散的模块组成，而出现的领域却与模型有点格格不入。例如，需要大量计算的领域，受到整体逻辑推理控制的领域，这些领域都不太适合面向对象范式。

### 5.6.2 对象世界中的非对象

领域模型并不一定是对象模型。在 Prolog 中实现了很多模型驱动设计，例如，由逻辑规则和行为构成的模型就是基于模型驱动设计的。模型范式已经成为人们考虑领域所钟爱的方式。这些领域模型将根据范式来制定。所以，遵从某个范式的模型能够在支持这种建模风格的工具中被有效地实现。



无论项目中采用什么样的领域模型范式，这个领域的某些部分应该更容易在其他范式中表现出来。如果领域中那些不正常的元素可以在一个范式中表现正常时，开发人员就能够忍受另外那个一致模型中难用的对象(或者是另一种极端，如果问题领域的大部分都能够在一个特殊范式中更加自然地描述，那么就要考虑交换范式，并选择不同的实现平台)。但是如果领域的主要部分看上去属于不同的范式时，用合适的范式构建每个部分非常吸引人，这可以通过使用混合的工具套件来实现。当相互依赖的关系很小时，就可以将另一个范式的子系统封装起来，就像复杂的数学计算只需要简单调用一个对象就能实现。其他时候，不同部分之间的关系纠缠得比较紧密，就像当对象的交互作用依赖于一些数学联系一样。

这就是在这种非对象组件的对象系统中，促使集成成为业务规则引擎和工作流引擎的目的。混合范式允许开发人员构造一些更加适合模型的概念。而且，大多数系统必须使用一些非对象技术框架，大多数的关系数据库也是如此。但是构建一个跨越多个范式的内聚模型是很困难的，让支持的工具共存也非常复杂。如果开发人员不能清楚地识别出嵌入在软件中的内聚模型，这时可以使用模型驱动设计，这种混合也增加了对模型驱动设计的需要。

### 5.6.3 在混合范式中使用模型驱动设计

规则引擎作为一种技术的实例，有时候将混合到面向对象应用程序开发的项目中。一个知识丰富的领域模型可能包含着明确的规则，然而对象范式缺乏声明规则和它们之间交互作用的明确语义。尽管规则能够被构建成对象，而且这样做也会经常获得成功，但是对象封装却使得在整个系统中应用全局规则变得很困难。规则引擎技术是非常有吸引力的，因为它提供了更自然、更具说明性的方式来定义规则，有效地允许规则范式应用到对象范式中去。逻辑范式不仅得到了很好的发展而且功能强大，它看上去像是对象特性的一种有利补充。

但是如果超出了规则引擎的范围，人们就不可能总能得到他们想要的东西。一些软件只是运行不太稳定。一些则缺乏一种整体视图，显示运行在两个实现环境中模型概念的相关性。一个共同的结果是，应用被分为两个部分：一个使用对象的静态数据存储系统；一个特殊的规则，处理与对象模型失去所有关系的应用。

在遵从规则进行开发时，根据模型继续考虑使用混合范式是很重要的。开发团队必须找到一个模型，能够同时应用两个实现范式。这不是一件容易的事，但是如果规则驱动允许描述性的实现，这也是可能的。否则，数据和规则就会失去联系。比起领域模型



中的规则，引擎中的规则更像小程序。规则与对象用它们之间紧密、清晰的联系保持着各自的意义。

如果没有可结合的环境，就得依靠开发人员对模型进行精练，把整个设计结合起来。

使各部分结合的最有效工具是一种健壮的通用语言，这种语言是构成整个异构模型的基础。不断地在两个环境中应用这些术语，不断地用这种通用语言来表达这些术语，就能够帮助我们消除这种差异。

就这个问题足可以写出一本专著来。本节的目标仅仅是说明没有必要放弃模型驱动设计，而且在混合范式中使用模型驱动设计是值得的。

尽管模型驱动设计不一定是面向对象的，它依赖于模型结构表现出来的实现，作为结构的对象、规则或者工作流。如果可用的工具不能支持这种表现，请考虑重新选择工具。一个没有表现的实现会否定在模型中应用多个范式的优势。

根据把非对象元素添加到一个面向对象系统中的经验，可以得出下面 4 个规则。

- 不要否定实现范式。总是可以用其他的方法来考虑领域。找出适合这个范式的模型概念。
- 依靠通用语言。尽管工具间没有严格的联系，坚持使用术语可以保持设计的各部分不会出现差异。
- 不要担心 UML 的能力。有时候，由于工具上的限制，例如 UML 图，会误导人们破坏模型来迁就这些图表。例如，UML 确实有一些用来表示约束的特性，但是它们总是不很充分。另外一些绘图风格(可能是其他范式的图表惯例)，或简单的文字描述，都比强行去适应一种绘图风格好得多。
- 要保持怀疑的态度。工具是否发挥了它真正的作用？因为您已经有了一些规则，这就意味着没有必要使用规则引擎。虽然规则能够被描述成对象，但多个范式会使问题变得非常复杂。

在处理混合范式之前，领域范式的选择应该是一直在进行。尽管一些领域概念没有将它们表达成明确的对象，但是它们经常能够在范式中被构建出来。第 9 章将讨论使用对象技术对非常规类型概念进行建模。

关系范式是范式混合的一个特例。最普通的非对象技术，关系数据库，与对象模型的联系比与其他组件的联系要密切得多。因为它持久稳固地存储组成对象本身的数据。在关系数据库中存储对象数据将在第 6 章进行讨论，同时还将讨论对象的生命周期。