

## 大比例结构



上千人分工合作共同制作“艾滋纪念被单”

硅谷的一家小设计公司签订了一个合同，为卫星通信系统制造一种模拟器。他们的工作进展顺利，正在开发一个模型驱动设计，用来描述和模拟各种网络条件和故障。

但是项目的领导却很担心。问题本身就很复杂，为了阐明模型中的复杂关系，他们已经把设计分解成一系列大小适中的内聚模块。现在的问题是，有这么多模块，开发人



员应该到哪个模块中去查找一个特定的功能？应该将一个新的类放在哪个模块？这些小模块会有什么用？它们又是如何结合在一起的呢？而且，以后还要创建更多的模块。

开发人员之间的交流非常充分，而且很清楚自己每天的任务，但是项目领导并不满足于表面上的可理解性。他们希望运用某种组织设计的方法，来保证设计在更复杂的情况下仍然能够被理解和处理。

他们进行集体讨论后提出了很多可能的办法，例如采用一种不同的打包方案，通过文档来给出系统的总体结构，或者在建模工具中使用新的类图视图把开发人员引导到相关的模块。但是项目领导并不满意这些肤浅的方法。

他们可以把模拟器的工作过程简单地描述出来。数据的编组工作是由一个基础设施来完成的，而数据的完整性和路由选择则通过电信技术层来保证。这个过程中的每一个细节都在模型中，但是我们无法从模型中看到这个过程的大致轮廓。

他们遗漏了一些领域中的核心概念。但是这一次遗漏的不是一两个对象模型的类，而是一个模型的整体结构。

开发人员花了一到两周的时间仔细考虑这个问题，逐渐形成了一个思路。他们将用一个结构把设计支撑起来。整个模拟器可以看作是一连串与通信系统各方面相关联的层。底层代表物理基础设施，具有把比特从一个节点传送到另一个节点的基本功能。然后是分组路由层，它主要关心的是如何传递数据流。而其他层将用来识别其他概念层中的内容。这些层描述了整个系统工作过程的轮廓。

他们按照新的模型结构对代码进行重构。模块必须被重新定义以便限制在一个层中，而不会跨越多个层。在一些情况下，他们需要分解对象的职责，以保证每个对象都明确属于一个层。反过来，这些概念层本身的定义又随着他们应用经验的积累得到精化。层、模块和对象一起推进，最后整个设计完全符合于这个分层结构所定下的轮廓。

这些层不是模块，也不是其他由代码组织起来的工件。它们是一种架构性的规则集，限制着整个设计中所有模块和对象(甚至包括与其他系统的接口)的边界和联系。

施加这种分层等级使得设计能够被轻松理解。人们可以知道到大概到哪里去查找一个功能。不同分工的个人都能够作出与他人基本一致的设计决策。这样就能提高设计所能达到的复杂性。

即使进行了模块分解，大的模型可能仍然会因为太复杂而难以掌握。模块分解使设计变成易于处理的小模块，但是这种小模块的数量可能会很大。而且，模块化并不能保证设计的统一性。不同的对象和模块可能会应用不同的设计决策，这些决策都行之有效但又各自为政。

限界上下文能够隔离不同的模块，防止出现破坏和混淆，但是它本身并不能让我们



更容易地从整体上理解系统。

精炼能帮助我们把精力集中于核心领域，并把其他子域提取到辅助模块中。但是，我们仍然需要理解辅助性元素和核心领域之间，以及辅助性元素和辅助性元素之间的联系。而且，尽管核心领域非常清晰易懂，不需要任何附加的指导，但是我们要处理的并不只是核心领域。

无论项目的规模怎样，人们总是要相对独立地负责处理系统的不同部分。如果没有任何协调或规则，那么各种不同的风格就会混在一起，甚至同一个问题会出现不同的解决方案，使人难以理解系统的各个部分是如何分工协作的，也不可能看清系统的全貌。研究设计的一个部分并不能让我们了解另一个部分，结果项目成员都只能处理特定的模块，一旦超出他们的范围就无能为力了。持续集成分解模型，而限界上下文又产生碎片。

在一个大的系统中，如果缺乏架构性的原则，无法根据元素在整体设计中的作用来对它进行解释，那么开发人员就会只见树木，不见森林。我们需要能够高屋建瓴地去理解每个部分在整体设计中所起的作用，而无需去深究细节。

“大比例结构(Large-Scale Structure)”是一种语言，它以一种很大的粒度来讨论和理解系统。我们可以用一组高层次的概念或规则来为整个系统的设计建立一个模式。这种组织原则能够为我们的设计和理解提供指导。同时，它还有助于分工合作，因为开发人员对系统的全貌取得了共识：它描述了部分是如何形成整体的。

设计一种规则(或角色和关系)模式，它跨越整个系统，使人们能够基本理解每个部分在整体中的所处的位置——即使他们并不了解这个部分职责的细节内容。

这种结构可以局限于一个限界上下文中，但是通常会跨越多个。它将概念组织起来，使项目涉及到的所有开发团队和子系统结合在一起。好的结构能够提供对模型的理解，并为精炼提供补充。

大多数大比例结构都不能用 UML 来描述，并且也没有必要这样做。它们体现并描述了模型，但设计并没有在其中体现出来。它们提供了另外一种传达设计思想的方法。在本章所举示例中，您会看到很多非正式的 UML 图，我在上面添加了大比例结构的信息。

如果开发团队很小，模型也不复杂，那么只需把模型分解为良好命名的模块，进行一些精炼，并在开发人员中进行一些非正式的协调，可能就足以使模型具有清晰的结构了。

大比例结构可以节省项目的开发费用，但是一个不合适的结构可能会严重地阻碍项目的发展。本章研究在此情况下成功构造设计的各种模式，如图 16-1 所示。

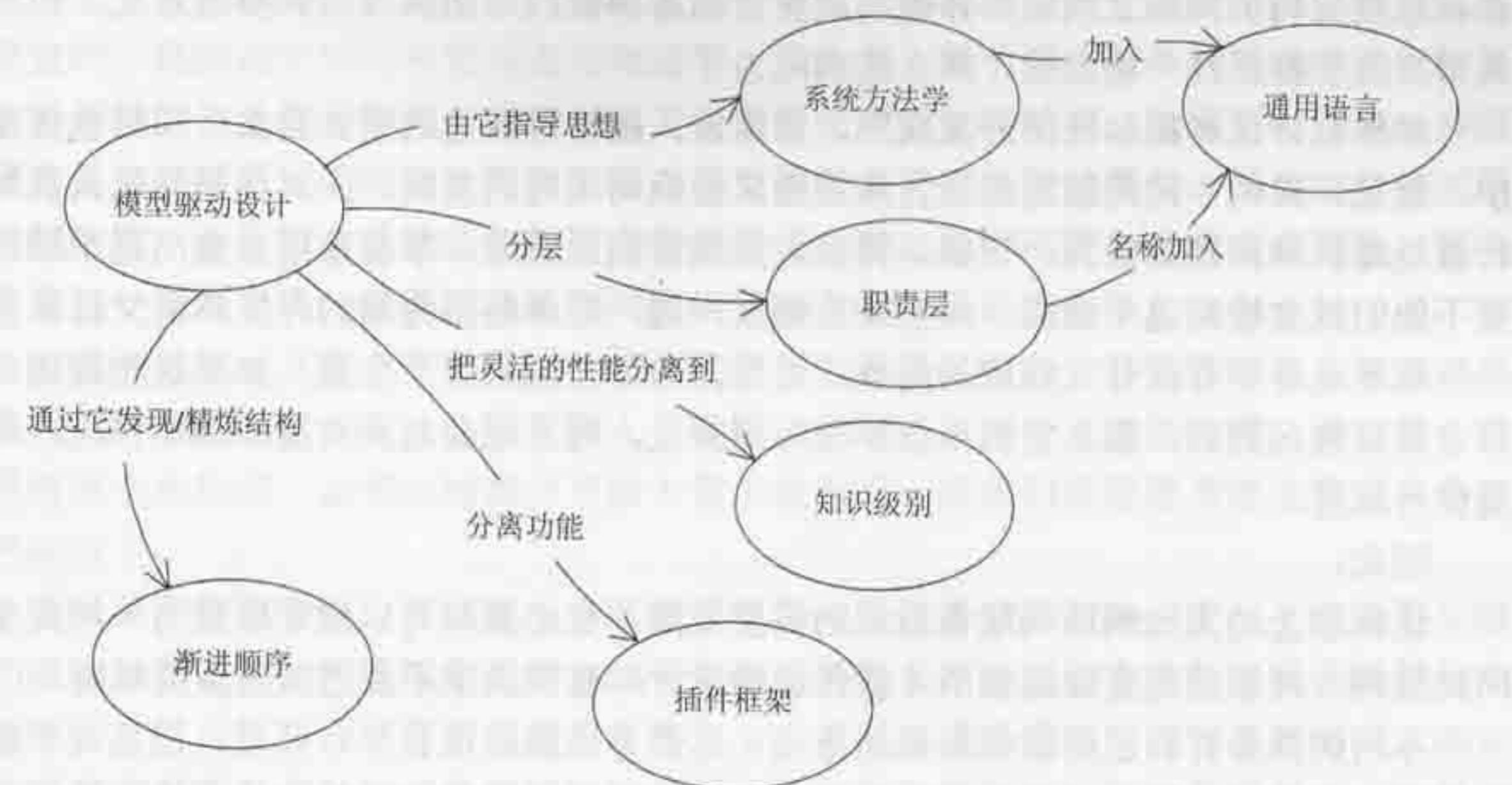


图 16-1 大比例结构的一些模式

## 16.1 渐进顺序

很多开发人员都经受过由于设计结构混乱而带来的痛苦。为了避免这种混乱，项目可以利用架构来从不同的方面对开发进行约束。有些技术性架构确实解决了一些技术问题(如联网和数据持久性)，但是，如果要用架构来解决应用模型和领域模型的问题，那么它们本身就会产生问题。它们常常阻碍了开发人员创造能够解决具体问题的设计和模型。最具挑战性的架构甚至可以不需要开发人员熟悉和掌握程序设计语言本身的技术功能。不管架构是面向技术的还是面向领域的，它们都固化了许多预先设置的设计决策，随着认识的深入和需求的变化，它们造成的束缚会越来越大。

尽管一些技术上的架构(例如 J2EE)已经得到多年的广泛应用，但是对大比例结构在领域层上的研究却不多。不同应用的需求变化范围很大。

预先采用一种大比例结构可能会花费很高的代价。随着开发的进行，您几乎总是会去寻找更加合适的结构，您甚至会发现，预先选择的那个结构竟然禁止您采用某些设计路线来大大简化应用的设计和理解。也许能使用该结构的一些部分，但是却丧失了其他机会。绕过那些约束或者与架构师商讨对策降低了开发速度，而管理者却认为架构已经完成了。架构本来应该能使应用的开发变得更加简单，那为什么不去开发应用程序，而



是在这些架构的问题上纠缠不休呢？就算管理者和架构师团队可以听得进意见，但是如果每次改变都像打一场大仗，那么就太吃力了。

如果设计没有施加任何开发规则，那就没人能够理解它的整体含义，同时也很难维护。但是，架构中预先规定的设计和假设又会阻碍项目的发展，大大限制开发人员和设计者处理具体问题的能力。很快，开发人员就会削足适履，拿着系统去套用这个结构，要不他们就会推翻这个结构，把它完全抛在一边，结果各自为政的开发问题又回来了。

这不是存不存在开发规则的问题，而是因为那些规则过于生硬。如果这些规则真的符合实际情况的话，那么它们不仅不会阻碍开发，而且还会起到有益的推动作用，同时提供一致性。

因此：

让概念上的大比例结构随着应用的需要发展，在必要时可以把它转变为一种完全不同的结构。对那些需要详细细节才能作出的设计和建模决策不要过做过多的限制。

不同的部分有自己的组织和表达方法，这些方法虽然很自然、有效，但是对于整体却并不一定适用，所以，施加某种全局的规则可能使得这些部分的设计不够完美。选择使用大比例结构的目的是为了便于把模型作为一个整体来管理，而不是为了优化不同部分的结构。因此，是采用统一的结构，还是允许以最自然的方式来表达各个组件，我们必须在这二者之间作一些折衷。我们可以根据实际经验和所掌握的领域知识来选择结构，并避免在结构中引入过多的限制，以便作出更好的折衷。如果结构与领域和需求非常匹配，那么它就能帮助我们迅速排除很多选项，使细节的建模和设计变得更加简单。

结构也可以为我们作设计决策时提供捷径。虽然我们也可以通过分析各个对象来作出这种决策，但是这很可能要花很长的时间，而且作出的决策也不一致。当然，我们仍然需要不断重构，但是结构能使这个重构过程更容易控制，还有助于让不同的人提出一致的解决方案。

在跨越限界上下文时，我们常常需要使用大比例结构。在实际项目的迭代过程中，结构会失去与某个具体模型紧密绑定的一些功能，同时又发展出一些符合领域概念轮廓(Conceptual Contour)的特征。这并不是说它对模型不作任何假设，但它不会拿着整个项目的概念去硬套一些特殊的局部情况。它必须为团队在不同上下文中的开发留下自由空间，以便他们能够根据具体的需要来改变模型。

此外，大比例结构还必须要适应开发中碰到的实际限制。例如，设计者可能不能控制系统中某些部分的模型，尤其是外部的或原来的子系统。我们可以改变结构来使之能更好地适应特定的外部情况。我们可以指定系统与外部的集成方式，或者创建足够自由的结构来适应一些棘手的实际应用的需要。



与上下文图不同，大比例结构是可选的。如果能够找到一种合适的结构，并可以从中获益时，就应该在项目中使用大比例结构。实际上，对于那些简单的系统来说，如果模块分解就能使系统易于理解的话，就没有必要使用大比例结构。如果可以找到一种能够更容易地阐明系统的结构，同时又不对模型的开发带来生硬的限制，那么就应该使用大比例结构。由于不合适的结构反而会带来弊端。因此最好不要去追求可理解性，而应该去寻找一个最小的、能够解决已有问题的解决方案。简单就是美。

大比例结构对开发非常有帮助，然而其中可能还是会存在一些例外情况，这些例外需要以某种形式标记出来。这样开发人员就可以知道，只要没有特别注明，设计就一定是遵循这个结构的。如果这种例外开始大量出现的话，那我们就需要考虑改变或者丢弃这种结构了。

正如我们前面所说，要创建一个既给开发人员提供了必要的自由度，又能够避免混乱的结构，需要的决不仅仅是技巧。尽管人们已经做了很多工作来为软件系统提供技术性的架构，但是针对领域层的结构还非常之少。有些技术会破坏面向对象的范式，例如那些根据应用任务或用例来分解领域的结构。整个这个领域还处于未开发状态。我研究了一些曾在多个项目中出现过的大比例结构的通用模式，本章将讨论4个这样的模式。这4个模式中可能会有一种符合您的需要，或者为您提供一些思路来构造符合项目需要的结构。

## 16.2 系统隐喻

在软件开发中到处渗透着隐喻的思想，建立了模型的软件尤其如此。但是极限编程的“隐喻”已经具有有了另一种含义，它表示一种使用隐喻来维护整个系统的开发秩序的特殊方式。

如果隔壁房子着火了，那么防火墙可以使我们免遭牵连。同样，软件“防火墙”可以保护本地网络免遭来自外部网络的危险。这个隐喻已经影响到网络架构，并且形成了一个完整的产品种类。市场上有很多竞争者为客户提供防火墙产品，这些产品都是独立开发的，并具有一定的可互换性。刚接触网络的新手也能很容易掌握这种概念。整个行业和客户都对防火墙具有同样的理解，这绝大部分是因为有了一个形象的隐喻。

然而，防火墙这种类比是不精确的，它所起的作用有利有弊。用防火墙来打比喻对开发形成了屏障，防火墙软件有时会作出过多的限制，妨碍必要的数据交换，同时又没



没有对来自防火墙内部的威胁提供任何保护措施。例如，无线局域网就容易受到攻击。“防火墙”这个比喻确实使我们受益匪浅，但是所有的隐喻都会有不足之处。<sup>1</sup>

软件设计的趋势是越来越抽象和难以把握了。开发人员与用户都同样需要一些切实的方法来了解系统，并获取对系统全貌的共识。

在某种程度上，隐喻非常符合我们的思维方式，因而这种思想渗透到了每一个设计之中。系统的各个“层”依次“叠放”起来。它们的“中心”具有“内核”。但是，有时候隐喻可以表达出整个设计的中心主题，并且为开发团队的所有成员提供一种共识。

在这种情况下，系统实际上是按照这个隐喻来发展的。开发人员将作出与系统隐喻(System Metaphor)相符的设计决策。这种一致性能够让其他开发人员根据相同的隐喻来解释一个复杂系统中的大多数部分。开发人员和专家们在讨论时就会有一个比模型本身更加具体的参考点。

系统隐喻是一种松散的、易于理解的大比例结构，与对象范式是一致的。由于系统隐喻只是对领域的一种类比而已，而不同的模型都可以用相似的方法来使用这个类比，因此隐喻能够应用到多个限界上下文中，并帮助协调它们之间的开发工作。

系统隐喻是极限编程的一种核心实践，因此它已经成为一种流行的开发技术(Beck 2000)。遗憾的是，几乎没有几个项目找到了真正有用的隐喻，还有的项目企图在领域中强加一些有害的隐喻。有的隐喻会给项目带来风险，因为它使得设计获得了其类比物的一些特性，而这些特性可能并不符合当前问题的需要。此外，有的类比虽然非常诱人，但是可能并不恰当。

也就是说，系统隐喻是一种众所周知的大比例结构的形式，它对于某些项目非常有用，并且能够很贴切地说明结构中的基本概念。

因此：

如果有一种具体的类比，它既描述了团队成员对于系统的想象，又可以指引他们朝着有用的方向进行思考，那么就把它作为系统的一种大比例结构。围绕这个隐喻来组织设计，并且把它吸收到通用语言中去。系统隐喻不仅能加强沟通，而且能指导系统的开发。它增加了系统不同部分的一致性，甚至还可以跨越不同的限界上下文。但是，因为所有的隐喻都是不精确的，所以要不断地重新检查隐喻是否过于引申或者不恰当，如果发现它不利于设计开发时就要准备放弃它。

<sup>1</sup> 当我听 Ward Cunningham 在一个专题讨论会的演讲上使用这个防火墙例子时，我终于明白了系统隐喻的意思。



## 自然隐喻

在大多数项目中，有用的系统隐喻并不会自动浮现出来，因此 XP 团体中已经有人开始谈及“自然隐喻(Naïve Metaphor)”这个概念了，他们指的是领域模型本身。

这个术语的一个问题是，一个成熟的领域模型决不会是自然的。实际上，模型是领域专家反复吸取知识得到的结果，并且已经融入到实际应用的系统中，得到了实践的证明。相比起来，像“工资单处理像一条装配线”这样的隐喻显得幼稚得多。

“自然隐喻”这个术语应该要停止使用了。

系统隐喻并非适用于所有的项目。大比例结构一般不是必要的。在极限编程的 12 个实践中，系统隐喻的任务可能会由一种通用语言来完成。如果找到了一种非常合适的系统隐喻，项目就应该把系统隐喻或者其他大比例结构添加到这个通用语言中。

## 16.3 职责层

纵观全书，每个对象都被分配了一组紧密相关的职责。职责驱动设计在更大的规模下同样适用。

如果每个对象的职责都要由手工来一一指定，那就看不到任何指导方针和一致性，也不能统一处理领域中的各方面问题。在分配职责时遵循某种结构将有助于维持大模型的一致性。

在深入了解一个领域以后，大的模式将会逐渐变得清晰起来。一些领域具有一种自然的层次化结构。某些概念和行为要以其他元素为基础，而那些元素又出于不同的原因，以不同的速度独立地发生变化。我们怎样才能利用这种自然结构，使它变得更加明显和有用呢？这种层次化结构使我们想到了分层，这是一种最成功的架构设计模式(Buschmann et al. 1996)。

层是一个系统的划分，每一层中的元素都知道并且能够使用“下层”提供的服务，但是不知道而且不依赖于“上层”。如果把模块之间的依赖关系画出来，那么被依赖的模块通常出现依赖它的模块的下面。这样，层本身有时候就起着一种分类的作用，所有较低层中的对象从概念上都不依赖于那些较高层中的对象。

图 16-2 所示的这种特定的分层虽然能够更容易地描绘出各层的依赖关系(有时候还能提供一些直观认识)，但并不能让我们获得对模型的深入了解，也不能指导我们作出建模决策。我们需要一些更深刻的理解。

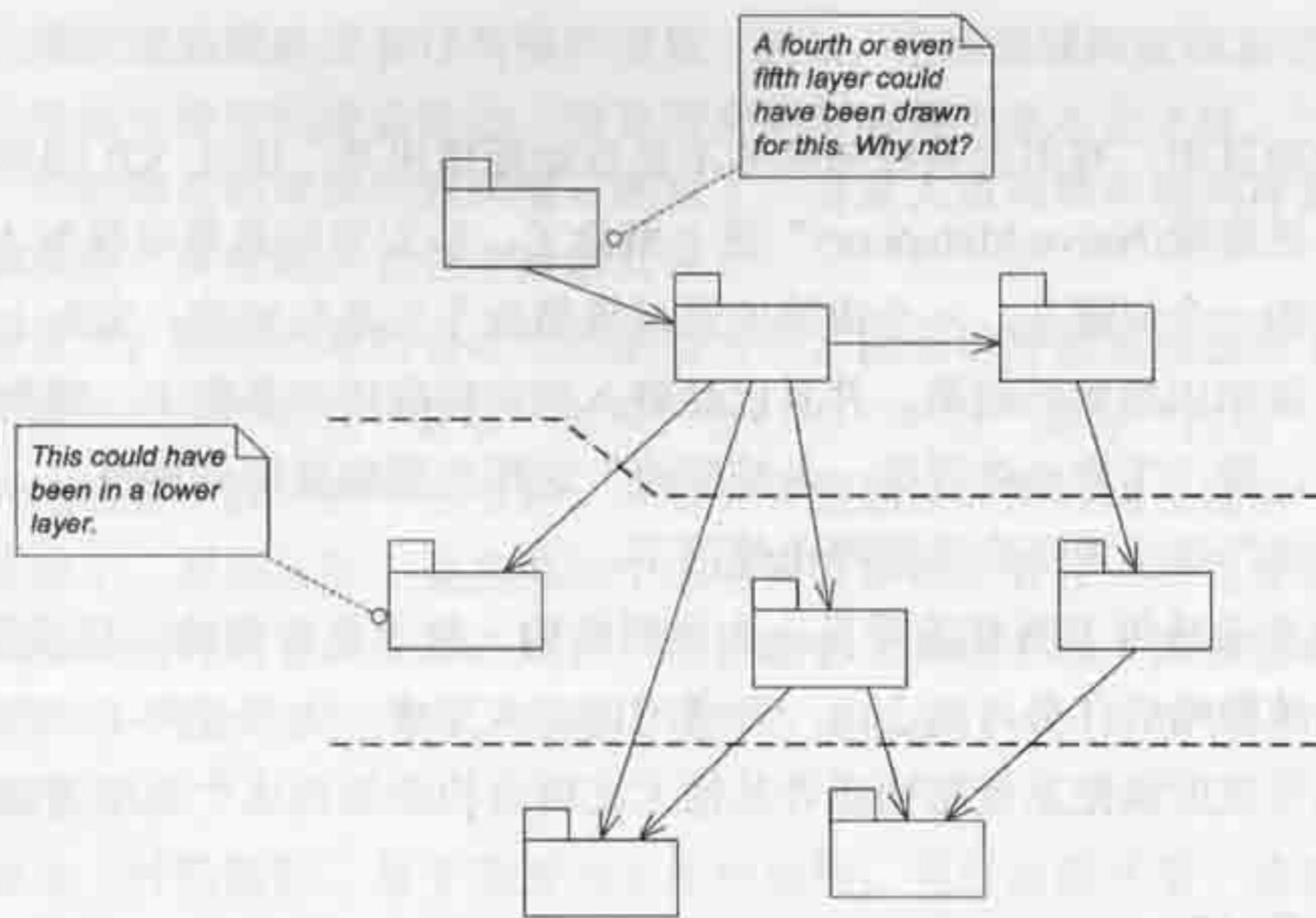


图 16-2 特定的分层：这些模块是作什么用的

在一个具有自然层次的模型中，我们可以围绕着主要职责来定义概念层，把分层和职责驱动设计这两个强大的原则结合起来。

这种职责的范围必须比通常指定给单个对象的职责范围要广泛得多(我们稍后将举例说明)。在设计独立的模块和聚合时，我们要对它进行重构，使之局限于一个主要职责的范围之内。对职责进行分组和命名可以增加模块化系统的可理解性，因为模块的职责更易于理解。分层和高层次职责二者的结合为我们提供了一种组织系统的原则。

### 分层模式

分层模式最适合于职责层结构，我们又把这种结构称为松散分层系统(Relaxed Layered System)(Buschmann et al. 1996)，它允许每一层中的构件访问所有比它低的层，而不是只访问紧接在它下面的那一层。

因此：

仔细考虑模型中概念之间的依赖关系，它们的变更速率，以及导致领域各个部分发生变化的来源。如果界定出了领域中的自然层次，那就把它们转换成大的抽象职责。这些职责可以用来描述系统设计的高层次目标。在每一层的职责范围内重构模型，使得每个领域对象、聚合以及模块的职责能够清晰地结合起来。

这是一种非常抽象的描述，但是看几个示例就能很快明白了。在本章开始介绍的卫星通信模拟器中，就是把它的职责进行了分层。我曾经看到过职责层结构在生产控制领域和财务管理领域中都获得了良好的效果。

下面的例子详细研究了职责层结构，让我们初步感受一下如何去发现一种大比例结



构，以及如何用这种结构来指导和约束我们的建模和设计工作。

### 示例：深入货运系统的分层

让我们把职责层应用到在前面几章例子中讨论的货运系统中，看看会有什么样的效果。

现在开发团队已经取得了相当大的进展，他们不仅创建了模型驱动设计，还精炼了核心领域。但是，当设计逐渐成形以后，他们在如何把各个部分结合起来时碰到了麻烦。他们正在寻找一种大比例结构，以便把系统的主要主题显示出来，并且让每个人对整个系统都获得同一个共识。

让我们在这里看一下模型中的一个典型部分，如图 16-3、图 16-4 所示。

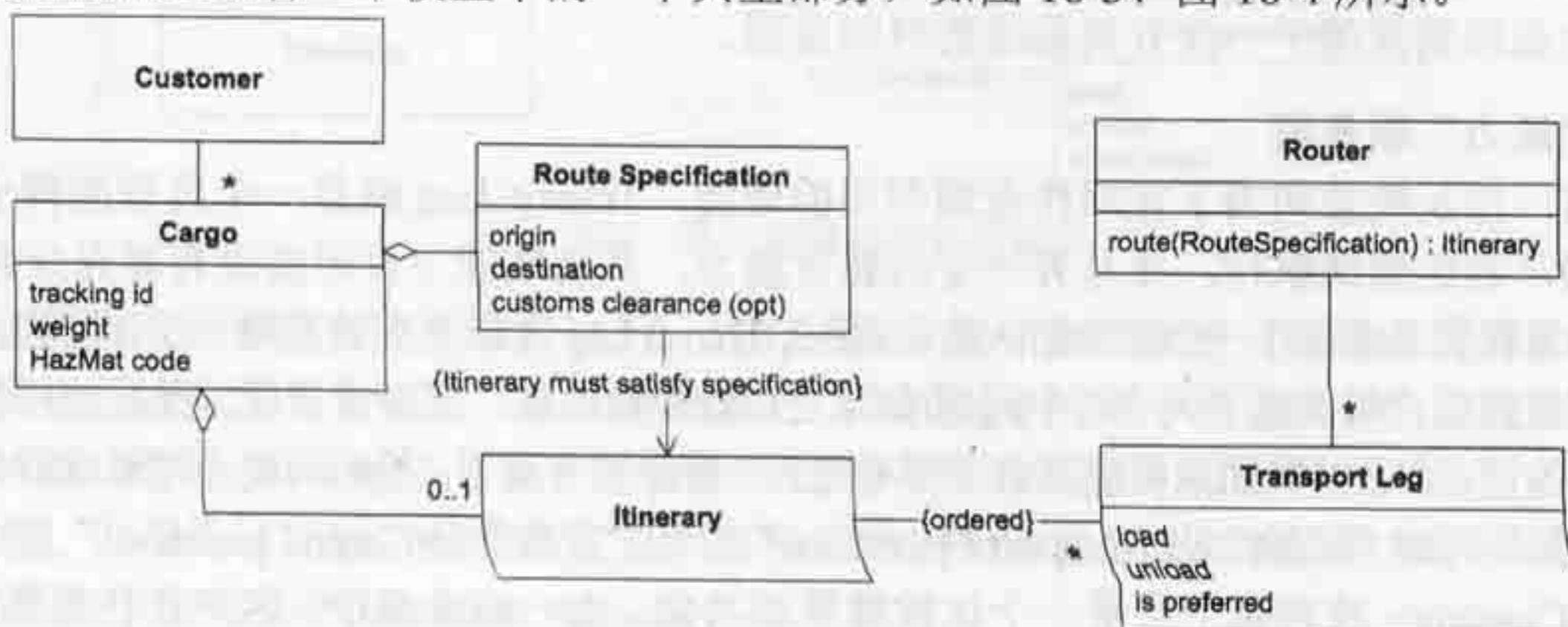


图 16-3 货运路线的一个基本运输领域模型

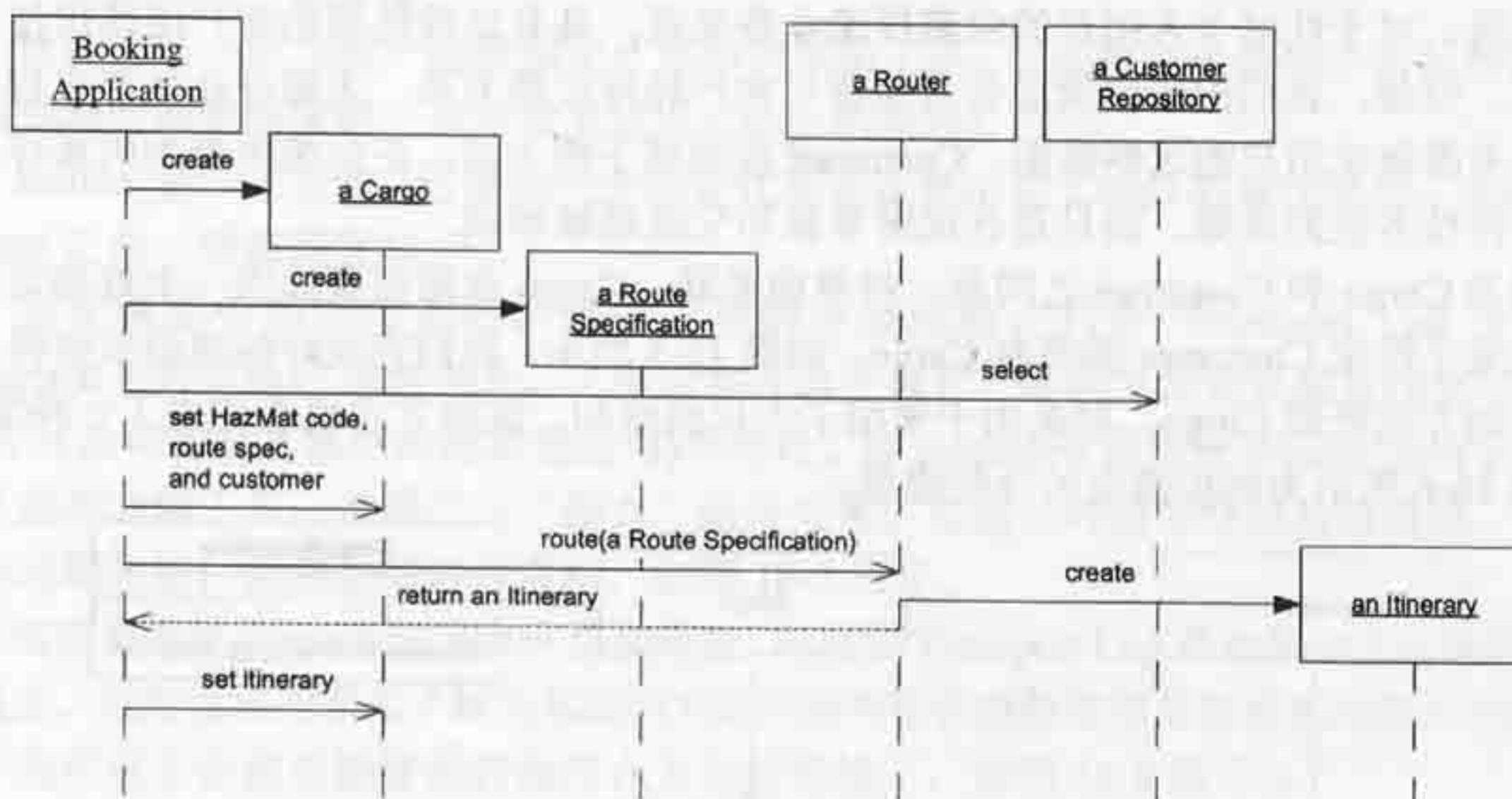


图 16-4 在预订期间制定货物运送航线的模型

团队成员涉足运输领域已经好几个月了，他们已经注意到系统的概念中存在一些自



然层次。在讨论运输计划(运输船和火车的预订航程)时可以不考虑运输工具上是否装载了货物。但是，在对货物进行跟踪时就必须考虑运载货物的运输工具。它们在概念上的依赖关系非常清楚。开发团队可以很容易地把系统分成两层：“作业(Operation)”，以及它们的支持基础(他们称之为“能力(Capability)” )。

### “作业”职责层

公司过去的、现在的和计划的行为都被归入作业层。最明显的作业对象是 Cargo，它是公司每天最关心的行为。Route Specification 是 Cargo 的一个主要部分，用来指示运送的需求。Itinerary 是可操作的运送计划。这两种对象都是 Cargo 的聚合部分，并且它们的生命周期依赖于一次有效运送的时间范围。

### “能力”职责层

这一层反映公司为了完成作业而利用的资源，Transit Leg 就是一个典型的例子。货轮按照计划的航线航行，并且有一定的载货能力，其装载能力有可能没有被充分利用。

如果我们考虑运作一个运输船队的话，那么 Transit Leg 实际上应该是属于作业层的。但是，这个系统的用户根本就不关心这个问题(如果公司既经营船队，又经营货运，并且想协调这两个业务的话，那么开发团队可能就必须得考虑另一种分层方案了，他们可能会把系统分成两个不同的层，例如“运输作业(Transport Operation)”层和“货物作业(Cargo Operation)”层)。

把 Customer 放到哪个层是一个比较棘手的决定。在一些业务中，客户往往是临时性的：他们只有在运送包裹时才会和系统发生关系，然后就消失了，直到下一次交易才又重新出现。对于针对个人用户的包裹投递业务来说，具有这种性质的客户应该仅仅与作业相关。但是，我们的假想货运公司希望与客户保持长期关系，大部分业务都来自于回头客。考虑商业用户的这些意图，Customer 应该属于能力层。正如您所看到的那样，这不是一种技术性的决策，而只是在试图掌握和交流领域知识。

因为 Cargo 和 Customer 之间是一种单向关联，Cargo 仓库需要提供一个查询功能，来找出某个特定 Customer 的所有 Cargo，如图 16-5 所示。我们有很好的理由来解释为什么以查询方式处理 Cargo，但是由于采用了大比例结构，这种方式现在变成了一种需求。

图 16-6 所示为初步确定的分层模型。

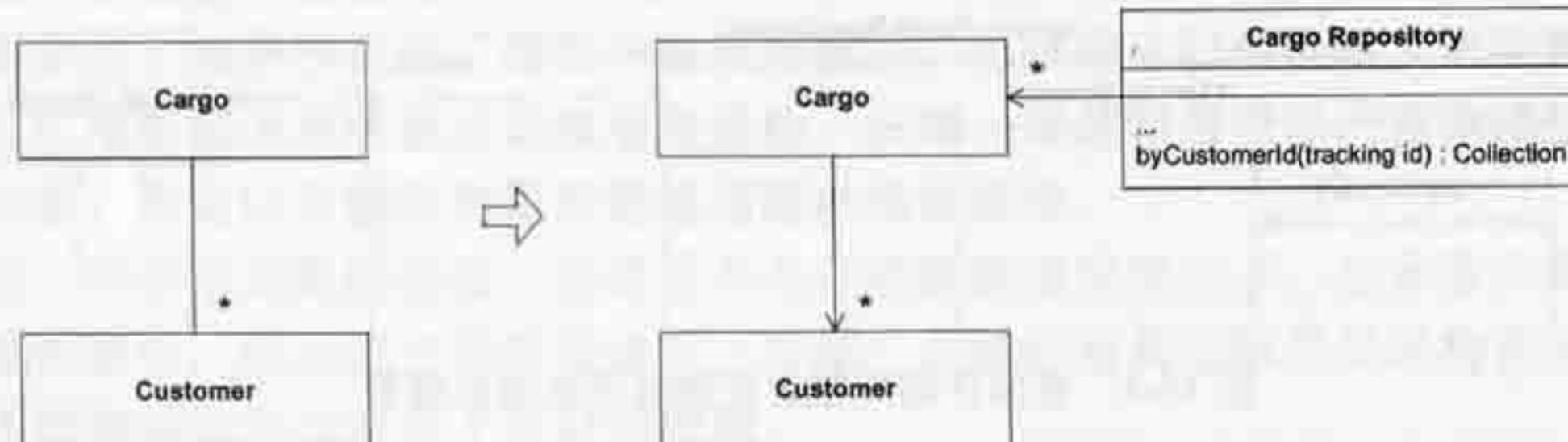


图 16-5 用查询代替会妨碍分层的双向关联

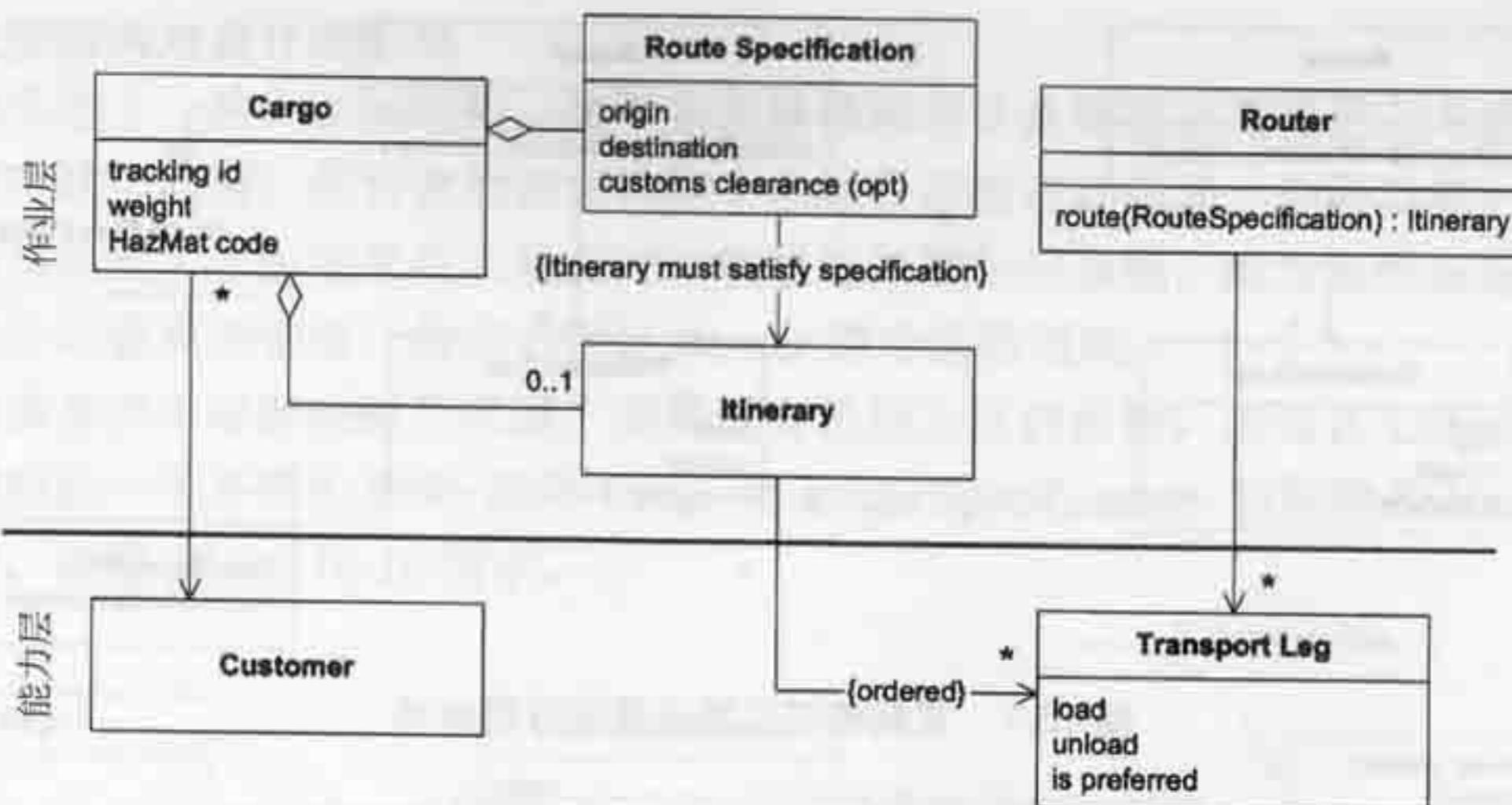


图 16-6 第一遍确定的分层模型

在弄清作业和能力之间的差别后，他们还要作进一步的改进。通过几周的试验，团队开始全力处理另一个问题。在很大程度上，最初划分的两个层所关心的都是实际的状况或计划。但是 Router(还有许多没有包括在这个例子中的其他元素)既不是作业实体的一部分，也不是计划的一部分，它是用来帮助用户作出修改计划的决策的。团队定义了一个新的“决策支持(Decision support)”职责层。

### “决策支持”职责层

软件的这一层为用户提供制定计划和决策的工具，它还可以自动地作出某些决策(例如当运输时间表改变时，自动改变运送 Cargo 的航线)。

Router 是一个服务，它帮助一个预订代理来选择运送 Cargo 的最佳路线。这样看来，Router 正是一种决策支持。

现在模型中的引用都符合这个三层的分层结构，只有 Transport Leg 中的“is preferred(是否首选)”属性例外。之所以存在这个属性，是因为公司更愿意使用它自己的(或者与公司签有互惠条约的其他公司)的货轮。is preferred 属性使 Router 倾向于选择这些首选的运输工具。该属性与“能力”毫无关系。它是一种指导性的决策策略。为了使用新的职责层，模型必须进行重构，如图 16-7 所示。

这次重构使 Route Bias Policy 更加清楚，同时让 Transport Leg 更加集中于运输能力的基本概念。以对领域的深入了解为基础的大比例结构往往能促使模型朝着更清晰的方向发展。

现在这个新模型能够很好地符合大比例结构了，如图 16-8 所示。

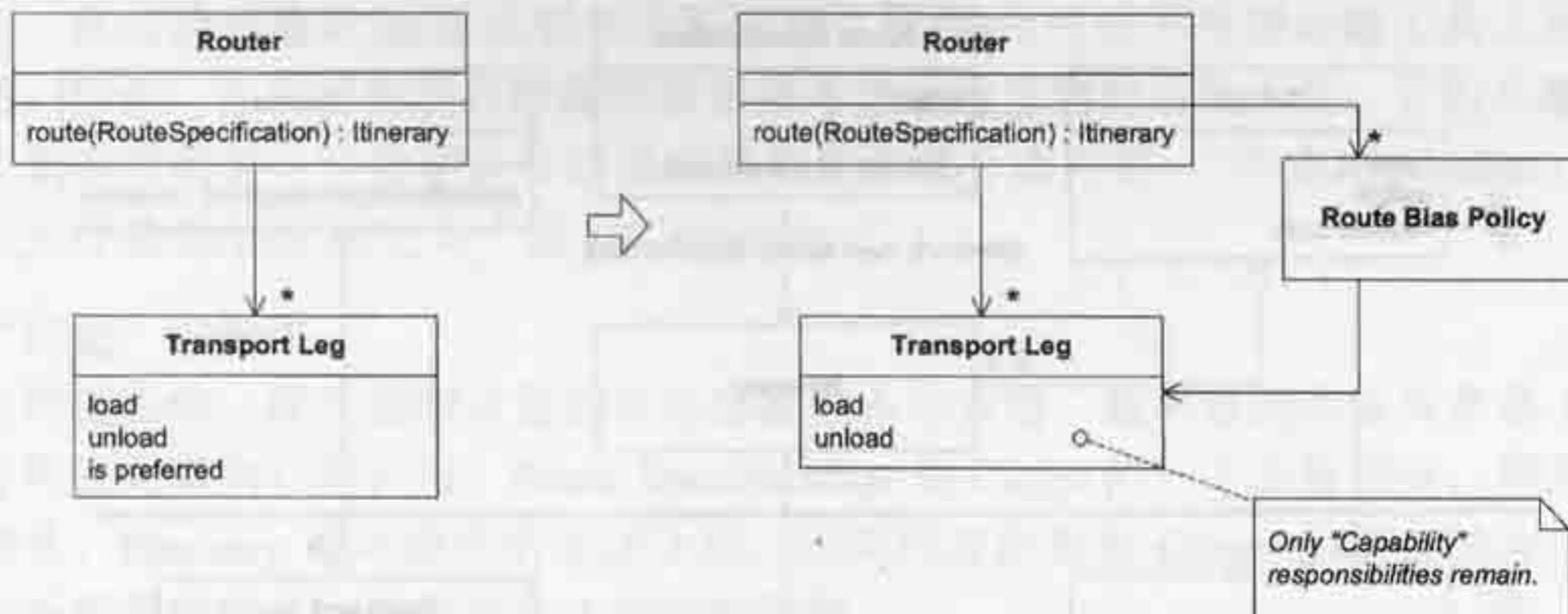


图 16-7 重构模型以符合新的分层结构

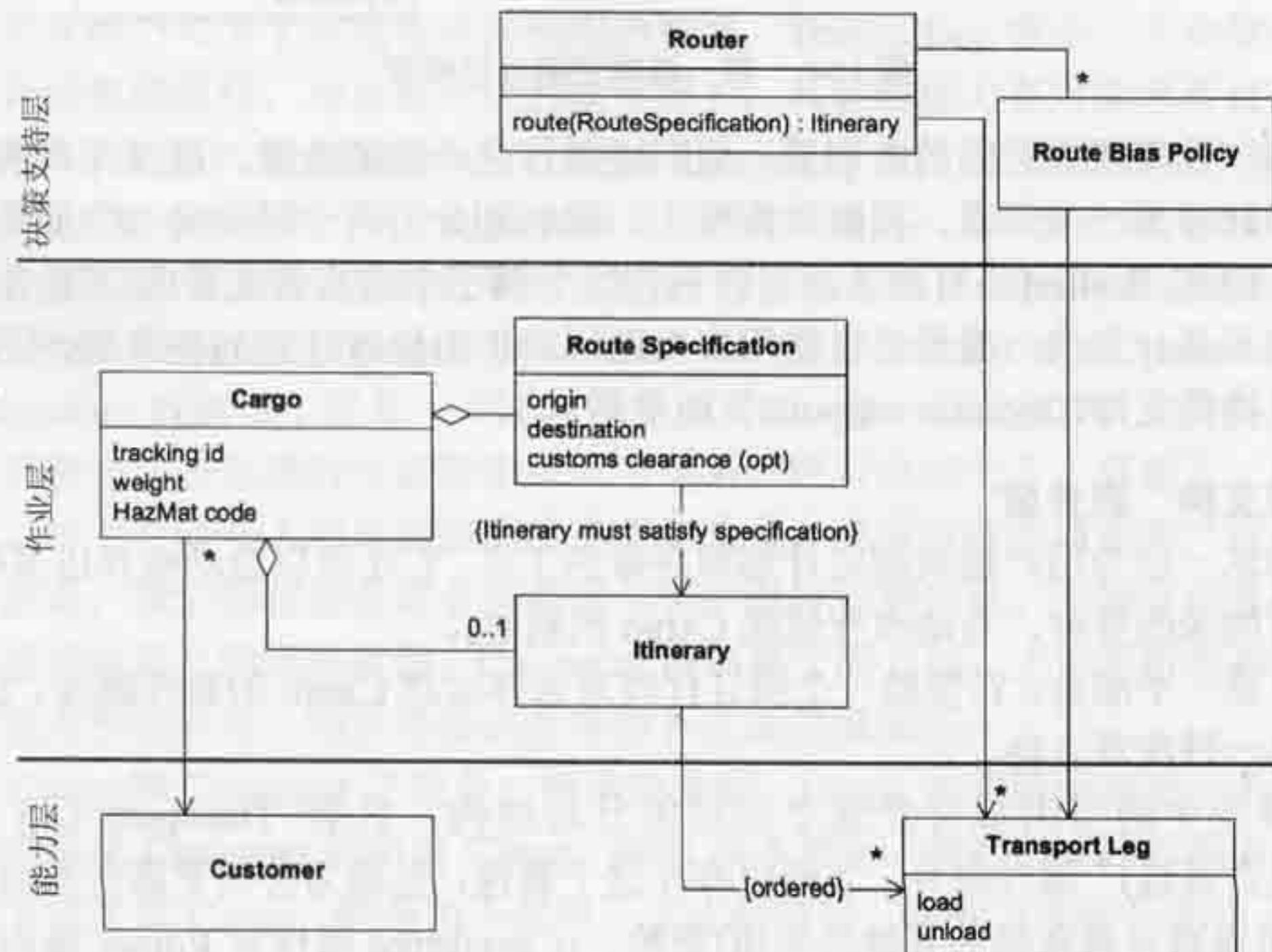


图 16-8 重组和重构以后的模型

如果开发人员对选定的层非常熟悉，那么他就能更容易地辨别出各个部分所承担的角色和依赖关系。大比例结构的价值随着复杂程度的增加而增加。

注意，尽管我用了一种改进的 UML 图来举这个例子，但是这只是一种表达分层的画法而已。UML 并没有这种标记方法，这是为了让读者有个清楚的认识而加上去的。如果代码是项目的最终设计文档，那么最好能够有一种工具来帮助我们按照分层结构来浏览类，或者至少按照层来提供类的信息。



### 大比例结构对设计的影响

一旦采用了一种大比例结构，接下来的建模和设计决策就必须要把它考虑进去。为了举例说明这个问题，假设我们必须给这个已经分层的设计添加一个新功能。领域专家曾经告诉过我们，在运送某些危险品时对航线选择有一些限制，因为某些运输船或港口可能不允许装载那些物资。我们必须让 Router 遵守这些规则。

有很多方法可以解决这个问题。如果没有使用大比例结构，那么让 Cargo 来负责合并这些规则是一个不错的设计，因为 Cargo 是 Route Specification 和危险品(HazMat)编码的所有者，如图 16-9、16-10 所示。

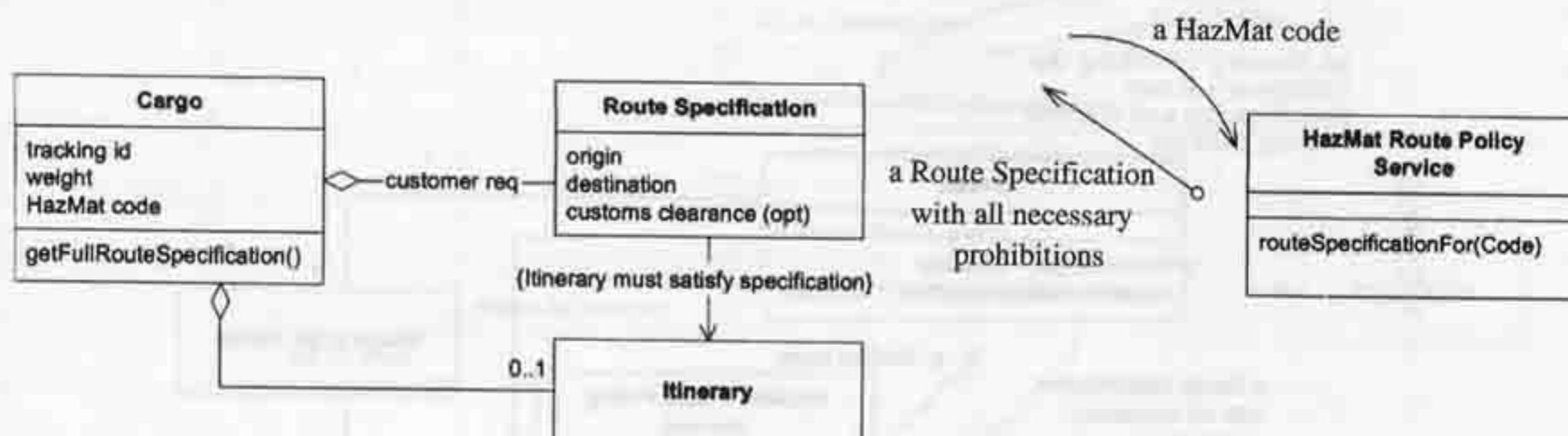


图 16-9 运送危险品的路线的可能设计

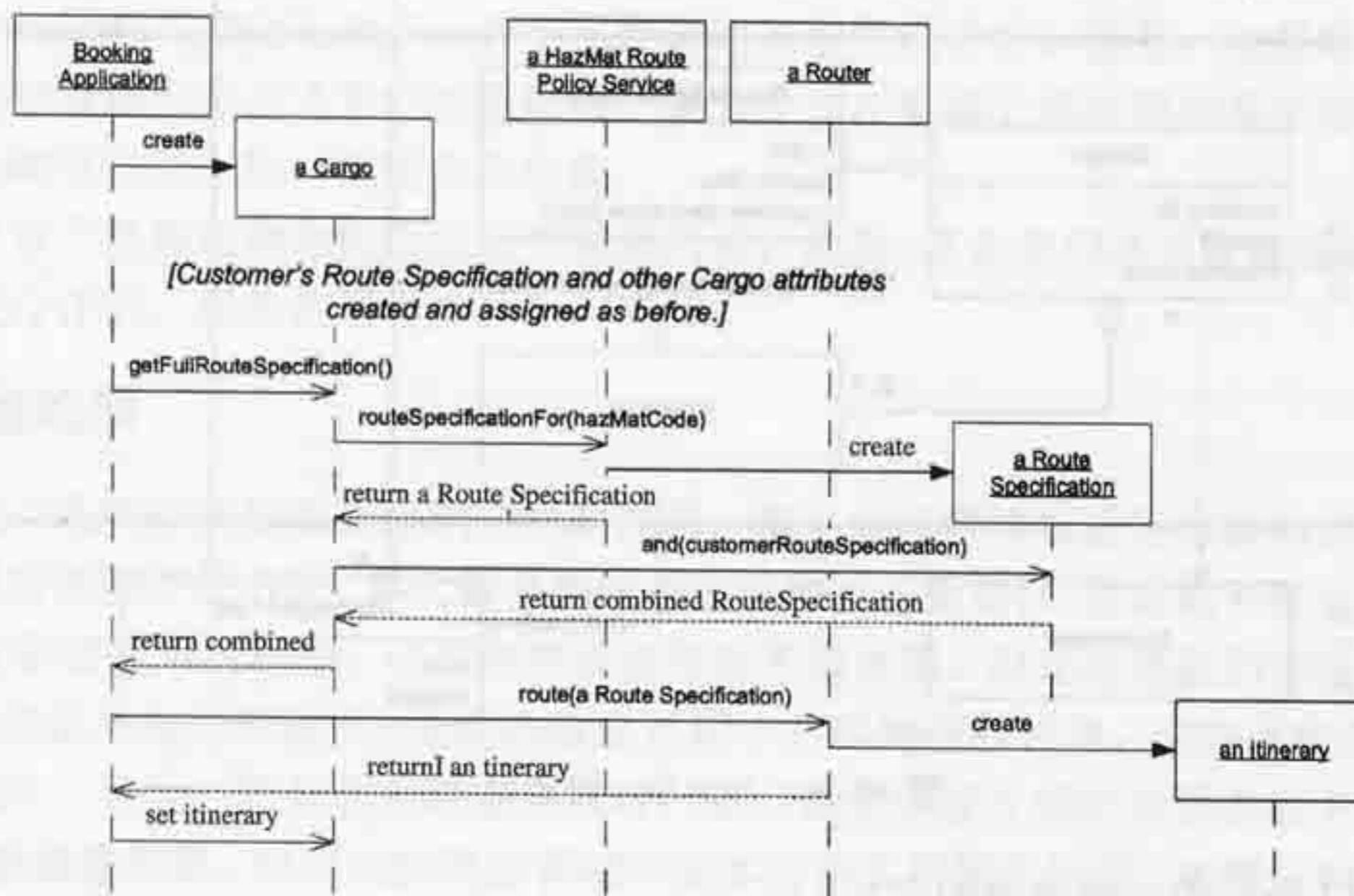


图 16-10 交互图

问题是这个设计不适合大比例结构。HazMat Route Policy Service 并没有什么问题，

它非常适合决策支持层的职责。问题出在 Cargo(它是一种作业对象)到 HazMat Route Policy Service(一种决策支持对象)的依赖关系上。只要项目遵循这个分层结构,就不能允许出现这种模型,因为那些以为设计遵循了分层结构的开发人员会被它弄糊涂。

可供选择的设计总是会有很多种的,这里我们只是需要另选一种遵循大比例结构的设计而已。HazMat Route Policy Service 是没有问题的,但是我们需要转移一些职责才能使用那些规则。试着让 Router 在搜索航线之前收集适当的规则,这意味着我们需要改变 Router 接口,把那些规则所依赖的对象包含进来。图 16-11 是一个可能的设计。

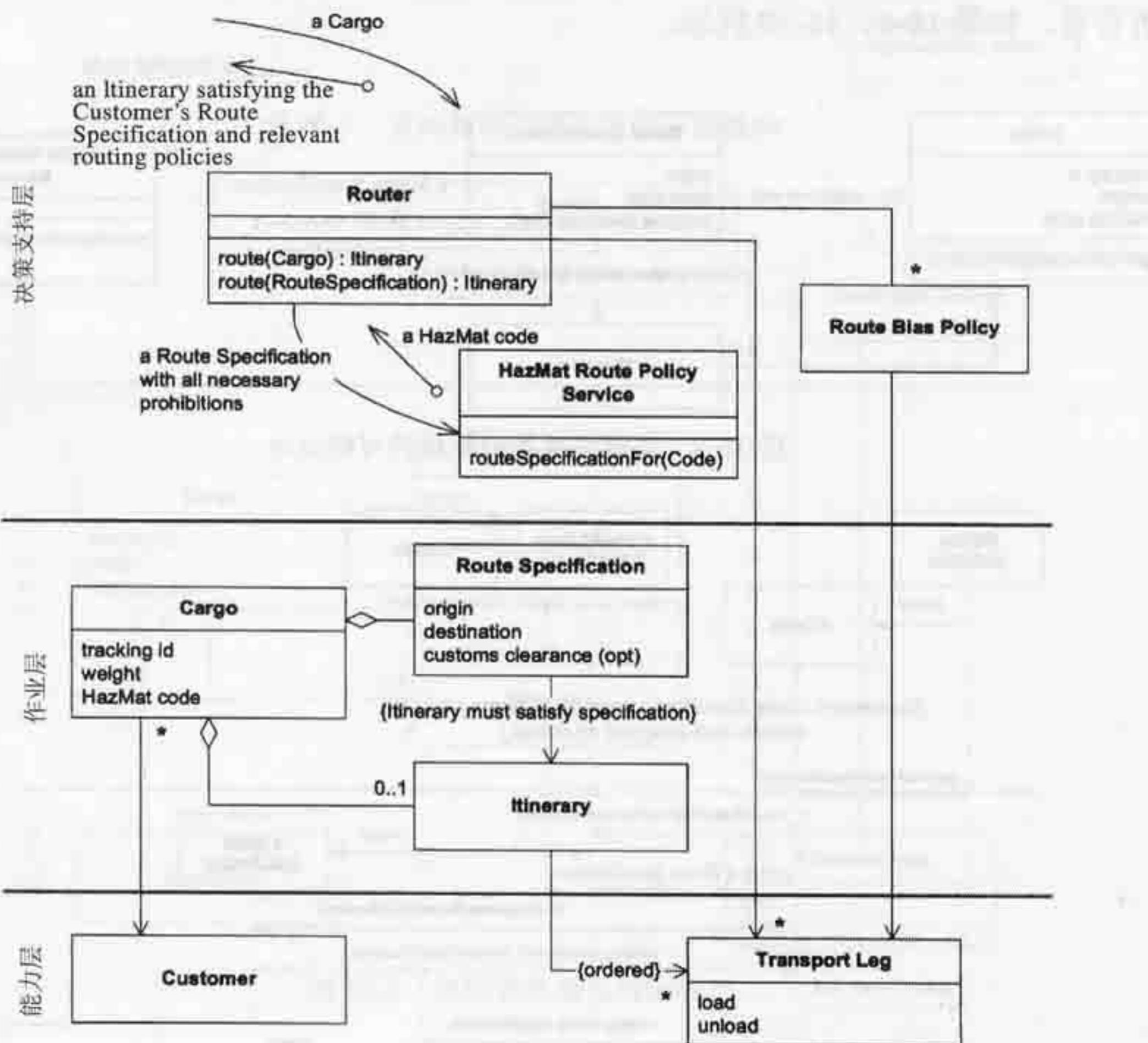


图 16-11 与分层一致的设计

图 16-12 所示为一种典型的交互。

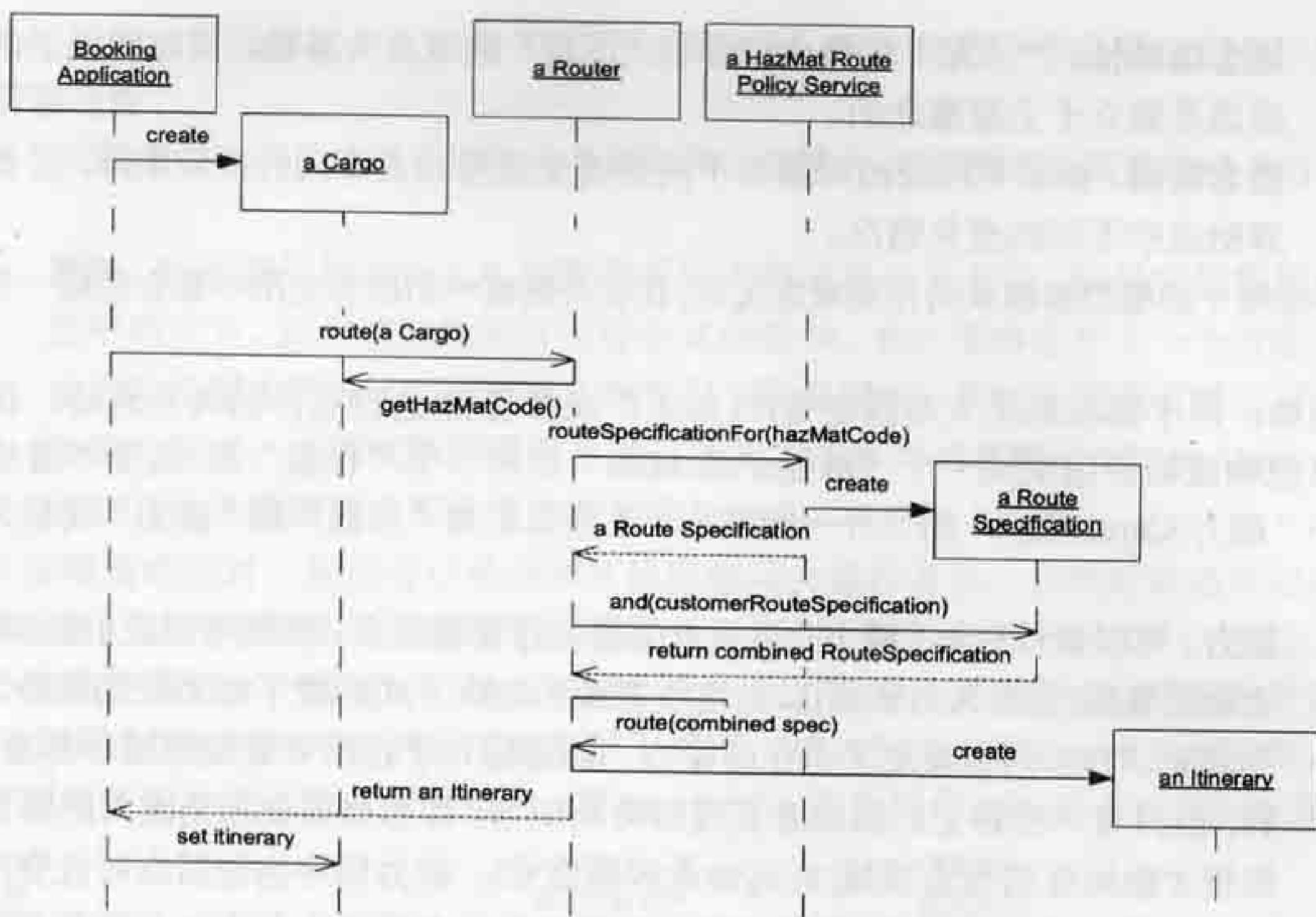


图 16-12 交互图

这个设计有优点也有缺点，并不见得会比其他的设计好。但是，如果项目中的每一个人都能够以统一的方式来作决策的话，那么设计在整体上将会更加易于理解，因此我们值得在细节设计上作一些适当的权衡。

如果这个结构迫使我们作出很多笨拙的设计选择，那么我们必须按渐进顺序的原则，对结构进行评估、修改甚至舍弃。

## 选择适当的层

寻找合适的职责层或大比例结构的过程，就是对问题领域进行理解和试验的过程。如果采用渐进顺序的方法，那么最开始选择的结构并不重要(尽管结构不好可能会增加工作量)。在渐进发展过程中，结构也可能会变得不堪使用。因此这里我们将给出一些指导原则，不论是从头开始设计，还是在渐进过程中对结构进行转换，这些原则都是适用的。

在划分、合并、拆分和重新定义层时，我们需要寻找并维护分层的如下一些特征。

- **能描述场景。**层应该描述出领域中基本的事实和优先次序。选择一种大比例结构与其说是一种技术决策，倒不如说是一种业务建模决策。层应该体现出业务中的优先次序。



- 概念依赖性。“上层”的概念应该以“下层”的概念为基础，同时较低层的概念应该是独立于上层概念的。
- 概念轮廓。如果不同层的对象有不同的改变速度或者不同的改变来源，层都能够容纳这些不同的变化情况。

并非每个新模型都得从头开始来定义层。在相关领域中的所有应用中都会出现一些确定的层。

例如，对于那些利用大型固定资产(如工厂或者货船)进行运作的商业活动，我们通常可以把物流软件组织成一个“潜能(Potential)”层和一个“作业”层(这里“潜能”是上例中“能力(Capability)”的另外一种叫法。下面我们将“潜能”和“能力”统称为“能力”)。

- 能力。可以做什么？“能力”不是考虑我们打算做什么，而是考虑我们能做什么。企业的资源(包括人力资源)以及这些资源的组织方式构成了能力层的核心。与供货商签订的合同也确定了企业的能力。能力层几乎在所有商业领域中都是存在的，但只有一些特定的领域才表现得特别明显，即那些需要相当庞大的固定资产投资才能运作的商业领域(如运输业和制造业)。能力同样也包括临时性资产，但是正如稍后讨论的那样，那些以临时性资产作为主要的业务推动力的商业领域可能会选择另一种层来体现这种特性。
- 作业。正在做什么？“作业”是我们运用那些能力来做的事情。和能力层一样，作业层应该体现真实的状况，而不是我们设想的状况。在这一层中我们要设法描述企业自身的动机和行为：我们在出售什么，而不是有什么让我们出售。通常，作业对象可以引用能力对象，甚至由能力对象组成，但是一个能力对象不能引用作业层中的对象。

在上述的那些领域中，“能力”和“作业”这两个层可以完全包括许多(也许是大部分)现有系统的内容(尽管这些系统可能会有一些完全不同的、意义更明确的划分)，能够跟踪企业的当前状态、现行的作业计划，以及结果报告和相关文档。但是，光有跟踪还是不够的，我们可以把另外一组职责集合起来，组织成作业层之上的“决策支持”层。

- 决策支持。应该采取什么行动或实施什么策略？这一层是用来为分析和决策提供支持的，它根据来自较低层(如能力层和作业层)的信息来进行分析。决策支持软件可以利用历史信息来为当前的和将来的作业主动寻找机会。

决策支持系统在概念上必须依赖于其他层(如作业层和能力层)，否则就会变成拍脑袋决策。很多项目使用数据仓库技术来实现决策支持，此时决策支持层就成了一个独立的限界上下文，与作业软件形成一种客户/供应商的关系。在其他项目中，决策支持层将与系统集成得更加紧密(正如先前的扩展示例那样)。层的一个内在优势是较低的层能够

不依赖于较高的层而存在。这样易于进行分期开发，或者在老的作业系统之上开发高层次的增强功能。

另一种情况是软件实施了详细的业务规则和合法需求，这些规则和需求构成了另一个职责层。

- **策略。**规则和目标是什么？规则和目标大都不是作业对象，但是它们能限制其他层中的行为。这种交互的设计可能会比较微妙。有时策略是作为一个方法参数传递到较低层中的方法，有的设计则使用了策略模式。策略和决策支持层可以很好地配合起来：决策支持层搜索各种可能的方法来实现策略层设定的目标；同时，这些方法又受到策略层设置的规则集的约束。

在实现策略层时，我们可以使用和其他层相同的编程语言，但有时候也可以用规则引擎来实现。这并不需要把策略层和其他层分别放到不同的限界上下文中。实际上，让二者严格地使用同一个模型可以降低不同实现技术之间的协调难度。如果规则所采用的模型与应用这些规则的对象所采用的模型不相同，那么不仅会增加开发的复杂度，还可能会束缚对象的功能以保持可管理性，图 16-13 列出了工厂自动化系统中实现的策略层。

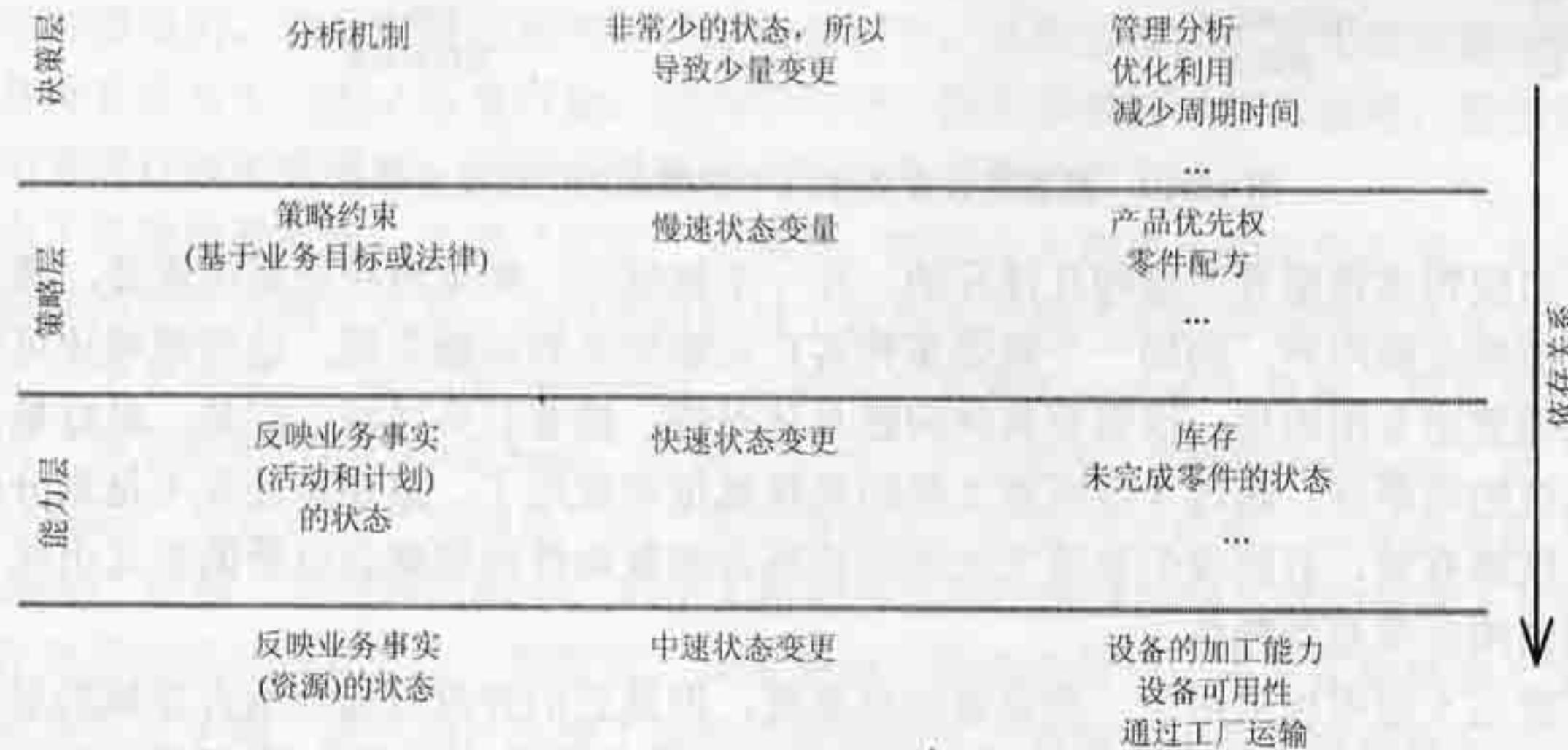


图 16-13 工厂自动化系统中的概念相关性和差异点

许多商业领域的功能都不是以工厂和设备为基础的。在金融服务或者金融保险业务中，功能在很大程度上由当前的业务情况来决定。一家保险公司是否有能力来承担一份新的保单所带来的风险，取决于其当前业务的多样性。能力层有可能与操作层合并，并发展为一种新的分层。

在这些情况中经常出现的一个层是对客户所作的承诺，图 16-14 列出了投资银行业务系统中实现的承诺层。



- 承诺。我们承诺的是什么？该层具有策略的性质，因为它声明的目标可以指导将来的作业；但是它也有作业的性质，因为承诺的出现和改变是正在进行的商业活动的一部分。

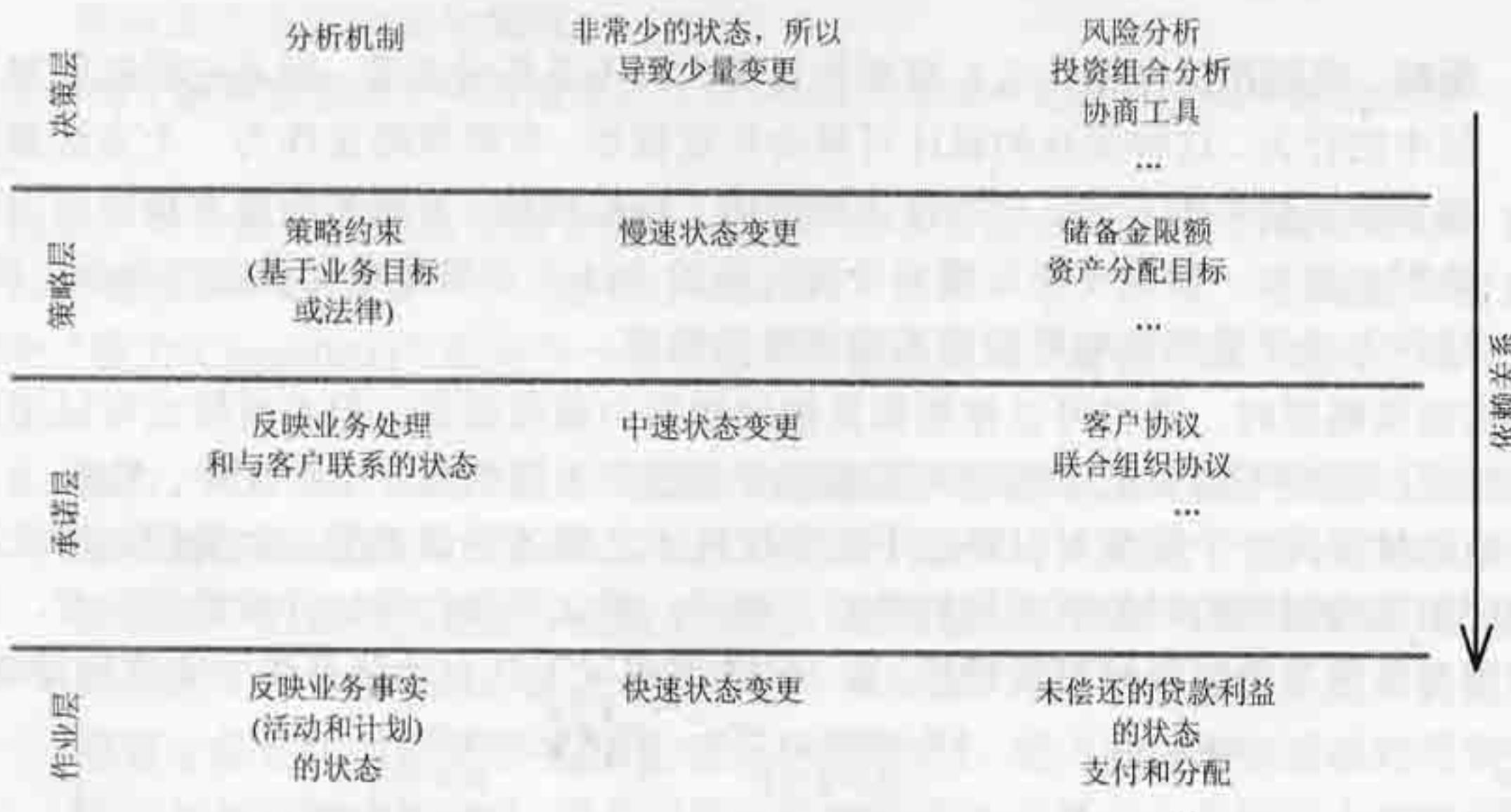


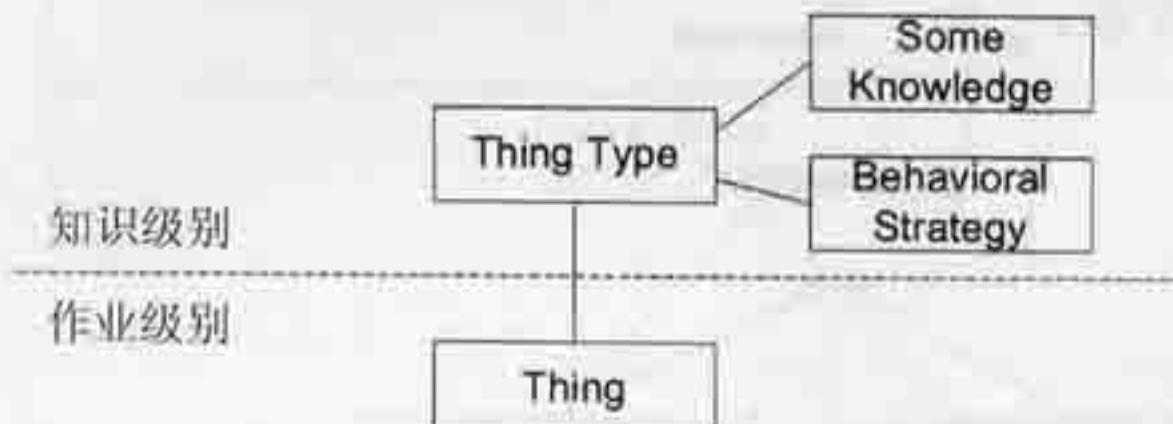
图 16-14 投资银行业务系统中的概念相关性及差异点

能力层和承诺层并不是相互排斥的。在一个领域中，如果两种层都很重要，那么它们就可能都会被用到，例如一个提供多种客户运输服务的运输公司。这些领域还可能会用到其他更加专用的层，这需要具体问题具体对待，摸着石头过河。但是，最好是保持分层系统的简单性，超过 4 层或者 5 层的系统就很难使用了。描述系统并不是划分的层次越多就越有效，否则我们想用大比例结构解决的复杂性问题就会以新的形式出现了。大比例结构必须相当精炼。

尽管这 5 层可以适用于一些企业应用系统，但是它们并没有捕捉所有领域的显著职责。对于其他的应用系统来说，照着这种结构去生搬硬套只会起反作用(但是总会有一些职责层适合于那些系统的)。对于一个与前面讨论的情况完全无关的领域来说，我们可能要另起炉灶才行。最终，我们必须运用自己的直觉，从某个地方开始，然后渐进地得到所需的结构。



## 16.4 知识级别



[知识级别]是一组描述其他组对象行为的对象。[Martin Fowler, Accountability, www.martin-fowler.com]

如果我们要让模型的某些部分对于用户具有足够的可塑性，同时又满足一些更大范围的规则，那么知识级别可以为我们提供帮助。知识级别可以为软件提供所需的可配置能力，在这些软件中，实体的角色及其相互关系可以在安装时甚至运行时发生改变。

在*Analysis Patterns*(Fowler 1996, 24~27)中，知识级别模式是在讨论企业内部的会计责任时出现的，它后来被应用到会计过账规则中。尽管该模式在好几章中都有提到，但是并没有作为专门的一章来讨论，因为它与书中的大多数模式都不相同。那些分析模式是用来进行领域建模的，而知识级别模式则是用来构造模型的。

为了具体地看问题，考虑“责任”的模型。组织由人和更小的组织组成，并定义了它们扮演的角色和相互关系。不同的组织对于这些角色和关系的管理规则是有很大区别的。在一个公司里，“部门(department)”可能由“主管(Director)”负责，而他要把工作情况向“副总经理(Vice President)”汇报。在另一个公司里，“模块(module)”由“管理员(Manager)”完成，而“管理员”向“高级管理员”汇报开发进度。还有所谓的“矩阵式(matrix)”组织，在这种组织中，每个人根据各自不同的目的向不同的管理人员汇报。

在典型的应用中都会作一些假设。如果使用了错误的假设，用户在数据输入字段时可能就会张冠李戴。由于用户改变了这些字段的语义，系统的所有功能都将无法正常运行。用户可能会设法绕过一些限制，或者把一些高级的功能给关掉。他们被迫去学习复杂的使用方法，把他们工作中的行为方式转变成软件所接受的行为方式。他们可能从未享受过优质的服务。

当系统不得不进行修改或替换时，开发人员迟早会发现一些功能的含义与其外表并不相符，它们对于不同的用户群(或者在不同的环境下)可能代表着完全不同的含义。如果不去掉这些交错重叠的用法，那么要对系统作任何改变都会是一个艰巨的任务，因为我们必须理解所有这些怪异的部分，并且对它们进行编码，才能把数据移植到更合适的系统中来。



### 示例：职工的工资和退休金(第1部分)

有个中等规模公司的人力资源部门用一个简单的程序来计算职工的工资和退休金，如图 16-15、图 16-16 所示。

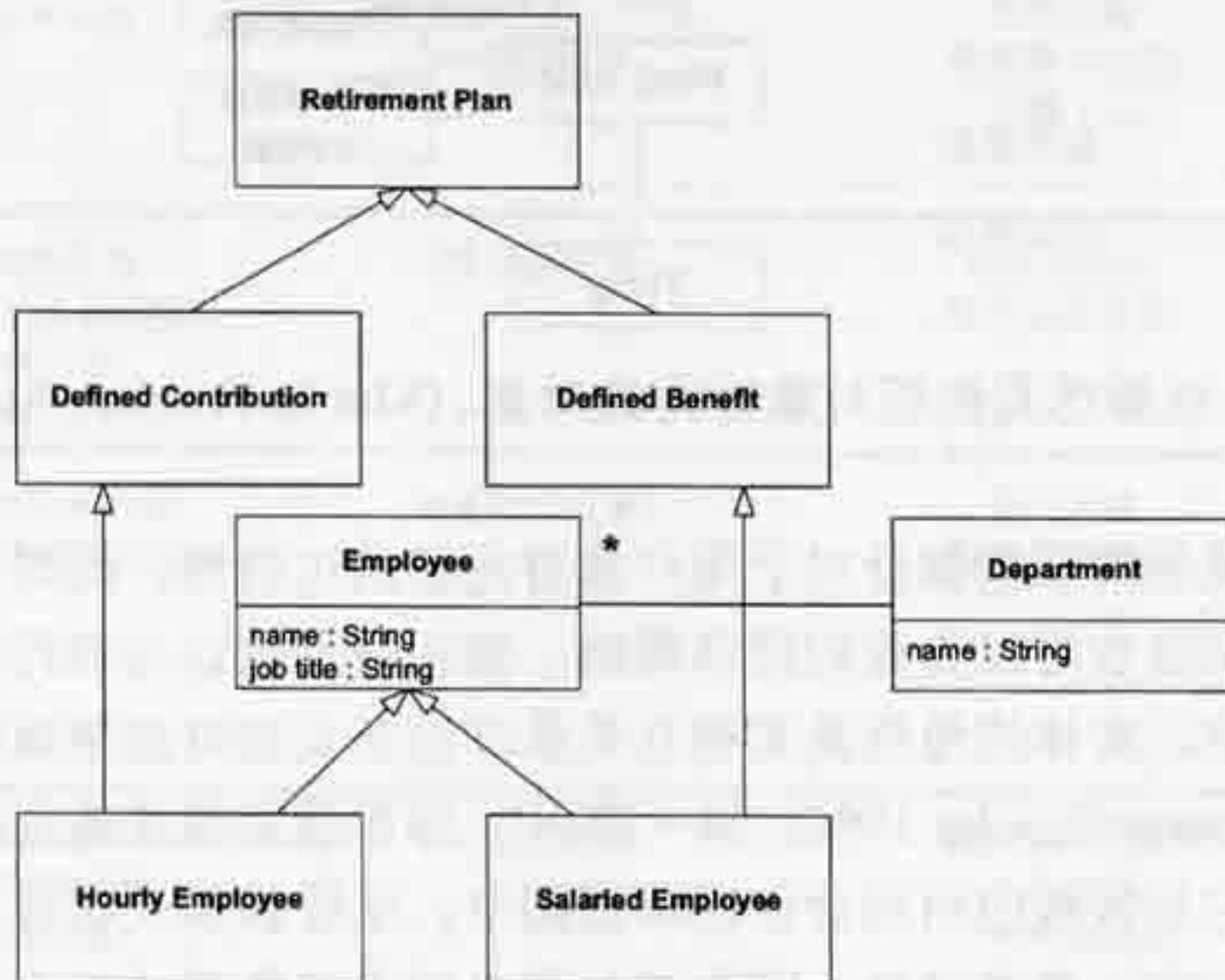


图 16-15 原模型对新需求有过多的约束

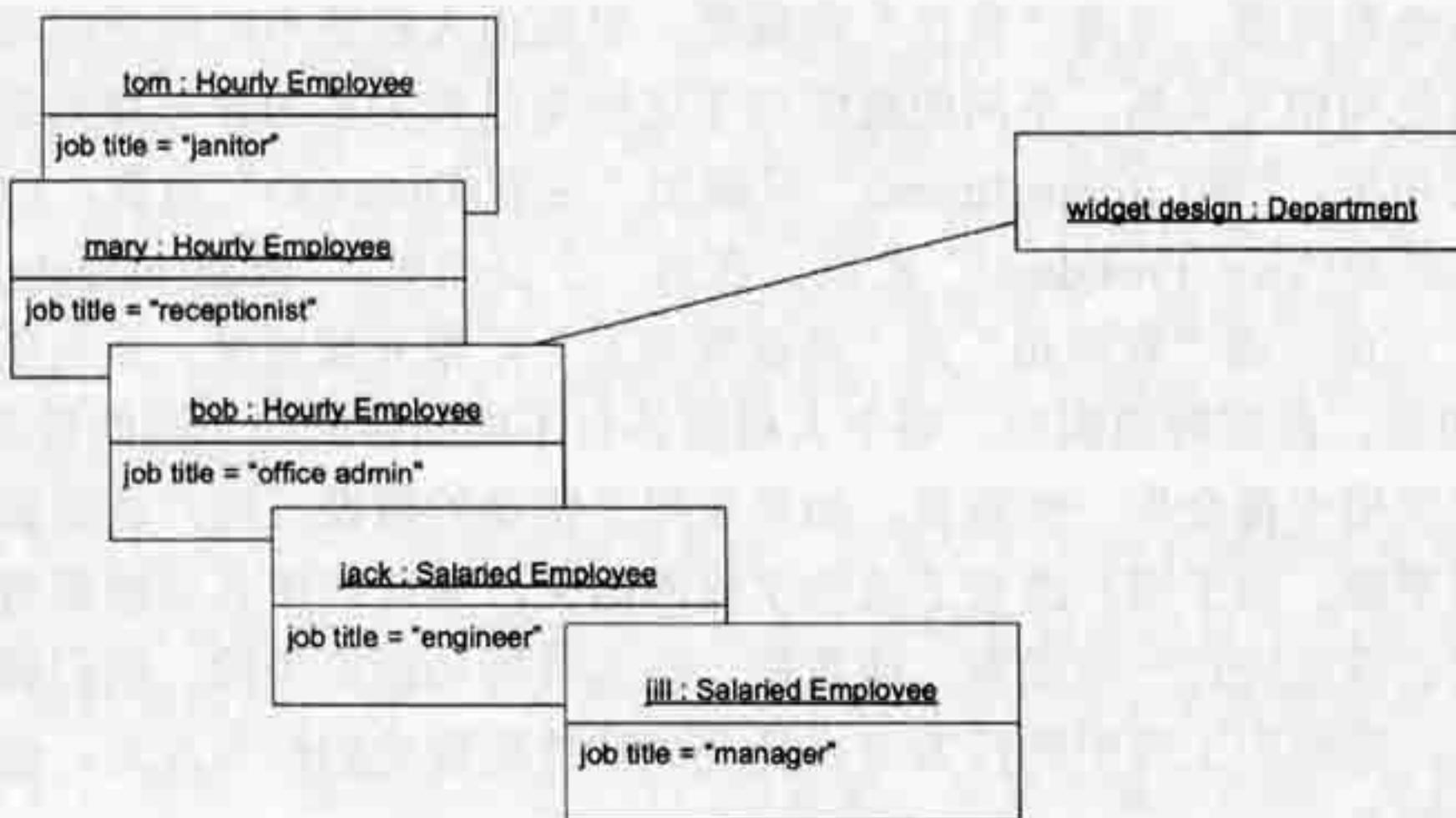


图 16-16 用原模型来表示一部分职工

但是现在，管理部门已经决定让办公室主任加入到“固定收益型(defined benefit)”退休金计划中来。问题是，办公室主任都是按钟点付给报酬的，而这个模型并不支持这种情况。因此必须对模型进行修改。



下面这个模型的方案非常简单：去掉了这些约束，如图 16-17 所示。这一方案也会产生一定错误，如图 16-18 所示。

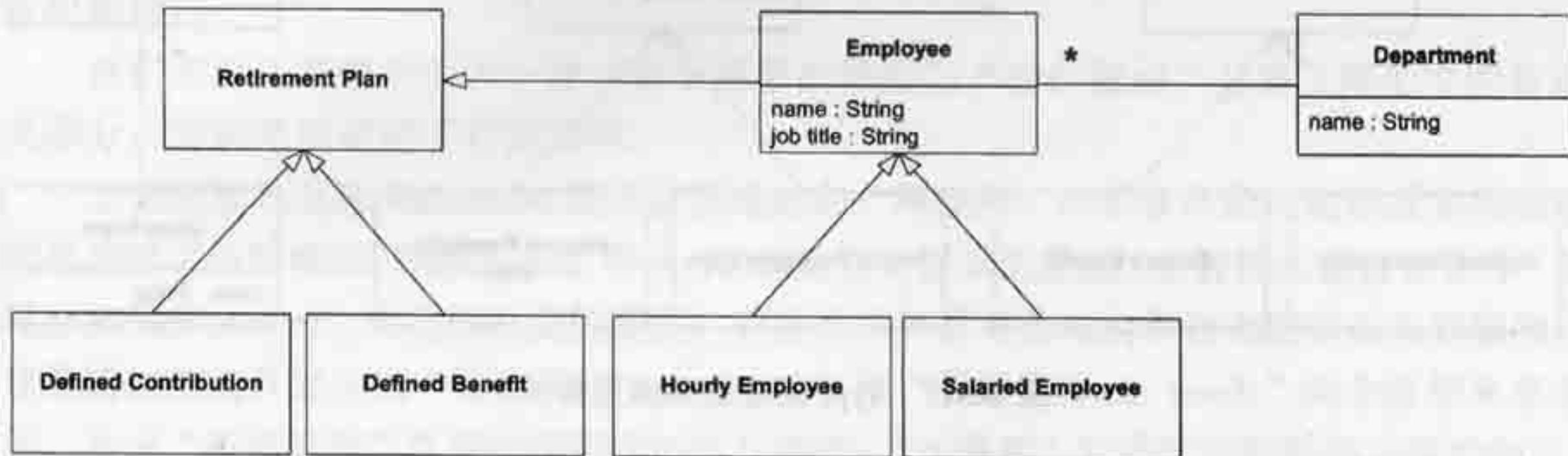


图 16-17 这个模型的约束过于宽松

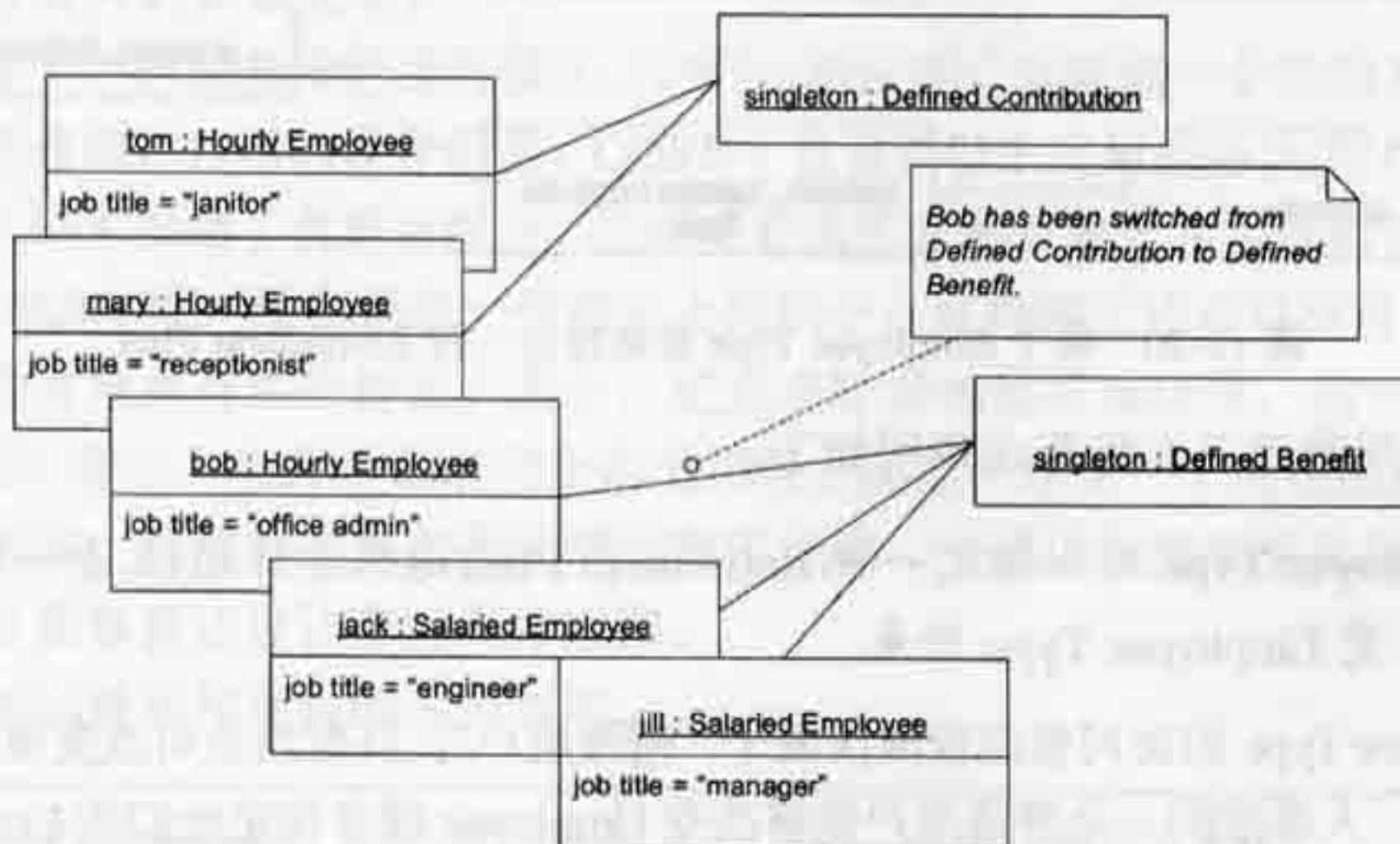


图 16-18 职工可能会关联到错误的计划上

这个模型中每一个职工都可以关联到两种退休金计划中的一种，所以每个办公室主任都可以从“限定拨款额式”换到“固定收益型”退休金计划中来。管理部门否决了这个模型，因为它没有反映出公司的策略。有的管理人员可以选择退休金计划，而另外一些却不能选择。甚至连看门的都可以选择。管理部门需要一个模型来实施下面这个策略：

办公室主任按时记酬，并且适用于固定收益型退休金计划。

这个选择提出的“职别(job title)”域代表一个重要的领域概念。开发人员可以把这个概念明确地构造成一个 Employee Type，如图 16-19、图 16-20 所示。

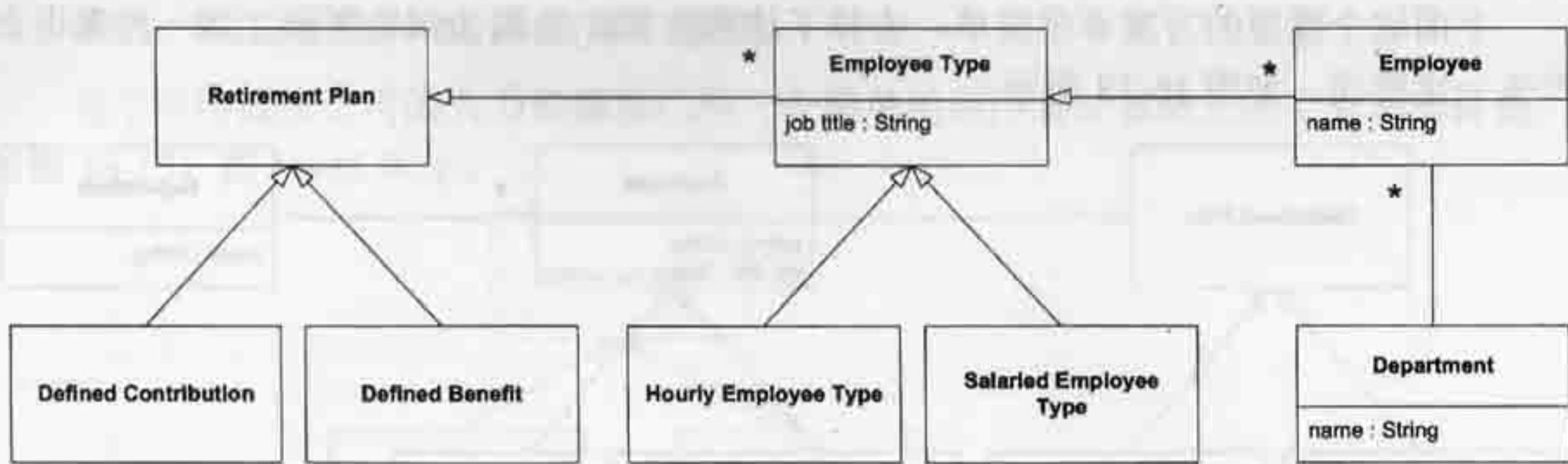


图 16-19 Type 对象能够满足需求

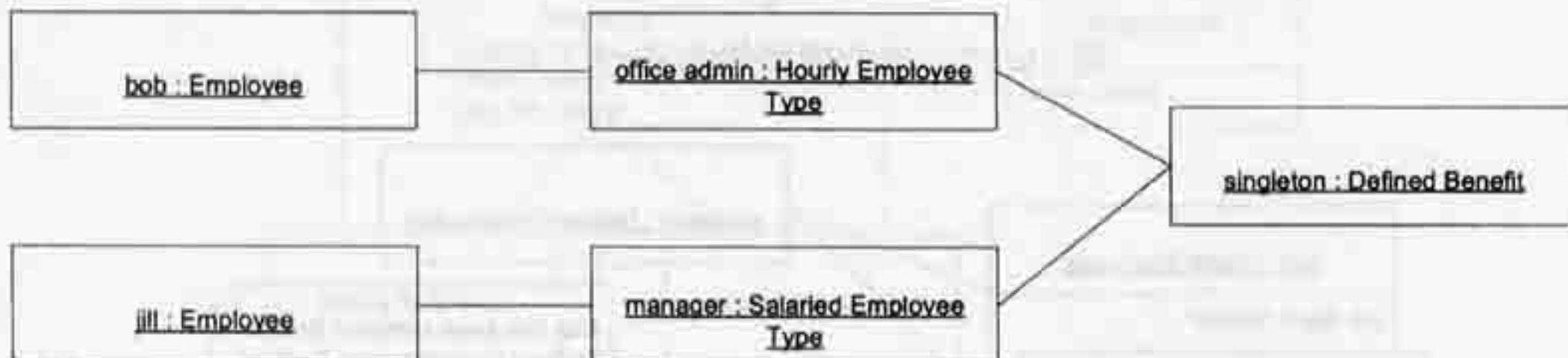


图 16-20 每个 Employee Type 都被指定一种 Retirement Plan

我们可以用通用语言将需求声明如下：

每个 Employee Type 可以指定一种 Retirement Plan(退休金计划)或者一种工资。

Employee 受 Employee Type 约束。

对 Employee Type 的访问修改权限仅限于“超级用户”，只有当公司改变策略时，超级用户才会去更改。人事部的一个普通用户能够改变 Employee 或者指定他们的 Employee Type。

这个模型满足了需求。开发人员感觉到一两个隐含的概念，但这种感觉也只不过是片刻的想法而已。他们还没有形成任何具体的构思，所以他们今天的工作到此为止。

静态模型可能会造成问题。但是如果系统过于灵活，以至于任何所有可能的关系都可以存在，那也不会好到哪里去。这样的系统不便于使用，也无法保证企业规则的实施。

为每一个组织开发完全定制的软件是不实际的，因为即使每个组织都愿意为定制软件付费，组织的结构也可能会经常发生改变。

所以这种软件必须提供选项，使用户能够通过配置来反映组织目前的结构。问题是，给模型添加这种选项会增加处理的难度。添加的灵活性越大，碰到的问题就越复杂。

在一个应用中，如果实体在不同情况下具有不同的角色和关系，那么开发的复杂性会急剧增加。不管是非常通用的模型还是完全定制的模型都不可能完全满足用户的需要。



最后开发出来的对象必须引用其他的类型来适应不同的情况，对象的属性在不同情况下可能会具有不同的使用方式。具有相同数据和行为的类可能会大量增加，来适应不同的装配规则。

我们可以在模型中引入一种“关于模型的模型”。“知识级别”分离了模型中的自定义部分，并清楚地描述了它的限制。

知识级别是反射(Reflection)模式在领域层的一种应用。许多软件的架构和技术基础结构都使用了反射模式，Buschmann 等人(1996)对它有详尽的描述。反射模式通过让软件“自我感知(self-aware)”来适应改变的需要，并且允许外界对其结构和行为进行访问和修改。这是通过把软件划分成“基础级别(base level)”和“元级别(meta level)”两个级别来完成的，其中“基础级别”负责应用程序的执行操作；“元级别”提供软件结构和功能的知识。

值得注意的是，我们并没有把该模式叫做知识“层(layer)”。虽然它与分层非常类似，但是反射包含双向的依赖关系。

Java 内置了一些最基本的反射模式，它们以协议形式来查询一个类的方法等。这样的机制允许程序查询它自己的设计信息。CORBA 也有类似于反射的协议(但作了一些扩展)。一些数据持久技术加强了自描述能力，以便在数据库表和对象之间实现部分自动映射。除了 Java 和 CORBA 外，还有其他一些技术上的例子。这种模式还可以应用在领域层上。

知识级别有两点有用的特征。第一，它重点针对的是领域应用，而不是我们所熟悉的反射的应用。第二，它并不追求完全的通用性。规范可能比一个通用的谓词更加有用，同样，用来约束一组对象及其关系的规则集可能比一个通用的框架更加有用。知识级别比较简单并且能够表达设计者的具体意图。

下面是知识级别和反射的术语对照：

Fowler 术语	POSA 术语 <sup>2</sup>
知识级别(Knowledge Level)	元级别(Meta Level)
作业级别(Operations Level)	基础级别(Base Level)

为了清楚起见，编程语言提供的反射工具不会用来实现领域模型的知识级别，因为那些元对象描述的是语言构造(如方法、属性)本身的结构和功能。但是，知识级别必须由普通对象构成。

因此：

创建一组对象来描述和约束基本模型的结构和能力。把这些对象分成两个“级别”，

<sup>2</sup> POSA 是 Pattern-Oriented Software Architecture 的简写，Buschmann et al. 1996。



其中一个级别非常具体，另外一个级别则反映用户或者超级用户能够定制的规则和知识。

像所有影响深远的观点一样，反射模式和知识级别模式提出来的观点是很令人兴奋的。但是不要滥用这些模式，因为它们虽然将操作对象从无所不能的要求中解脱出来，从而降低了复杂度，但是其间接作用确实又增加了一些模糊性。如果知识级别太复杂，系统的行为就会让开发人员和用户难以理解；用户(或者超级用户)将需要具有程序员的技能，甚至需要元级别程序员的帮助。如果他们犯了错误，应用程序也不会正常运行。

同样，数据迁移的基本问题也没有完全解决。当知识级别模式中的一个结构发生改变时，我们必须对现有的作业级对象进行处理。新老对象共存是可能的，但是无论如何都要进行仔细地分析。

所有这些问题给知识级别模式的设计者带来了较大的负担。该设计必须足够健壮，不仅能够处理在开发过程中碰到的情况，而且还能处理用户在今后配置软件可能遇到的情况。如果设计中的某个部分非常强调可定制能力，否则就会出错或者破坏整个设计，那么就可以应用知识级别，它可以解决这种很难用其他方式解决的问题。

#### 示例：职工的工资和退休金(第2部分：知识级别)

经过了一个晚上的休息，团队成员们都恢复了精神，其中已经有人开始接触到了一个棘手的设计问题：为什么有些对象是固定的，而其他对象却可以被随意修改呢？这些受限制的对象让他想起了知识级别结构，于是他决定试着把它作为一种查看模型的方法，如图 16-21 所示。他发现现有的模型已经可以用这种方法进行查看了。

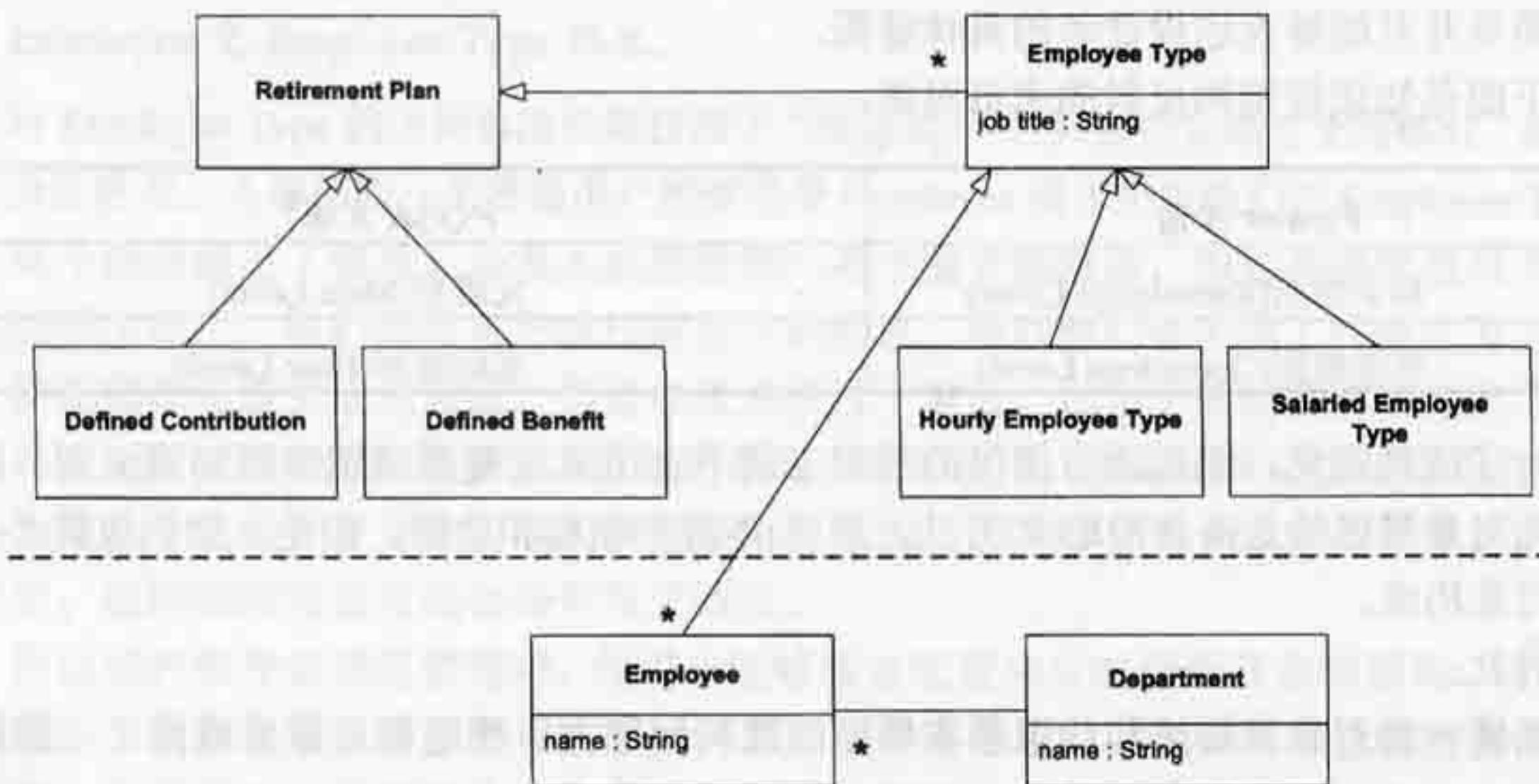


图 16-21 识别出隐含在现有模型中的知识级别



受限制的编辑对象在知识级别结构中，而日常的编辑对象在作业层。这是一种非常好的安排。所有在那条线上的对象用来描述类型或者一些长期策略。Employee Type 实际上是将行为施加在了 Employee 之上。

正当这个开发人员与同事讨论这个认识的时候，另一个开发人员也获得一个认识。她清楚地看到了模型可以通过知识级别来进行组织，这让她找到了前几天一直困扰着她的问题所在：两个截然不同的概念被混在了同一个对象中。她曾经在几天前听到过这个声明，但是没有引起重视：

为 Employee Type 指定一种 Retirement Plan 或者一种工资。

但是，这并不是一个真正用通用语言来表达的声明，因为在模型中没有“工资 (payroll)”一说。他们是用自己“需要”的而不是已有的语言来给出这个声明的。工资概念被隐含在模型之中，与 Employee Type 混在了一起。在知识级别结构被分离出来之前，它并不那么明显。而且，那个关键声明中的所有元素都处于同一个级别——只有一个例外。

基于这种认识，她对模型进行了重构以支持那个声明。

为了让用户能够控制对象之间的关联规则，开发队伍开发出了一个具有隐含知识级别的模型，如图 16-22、图 16-23 所示。

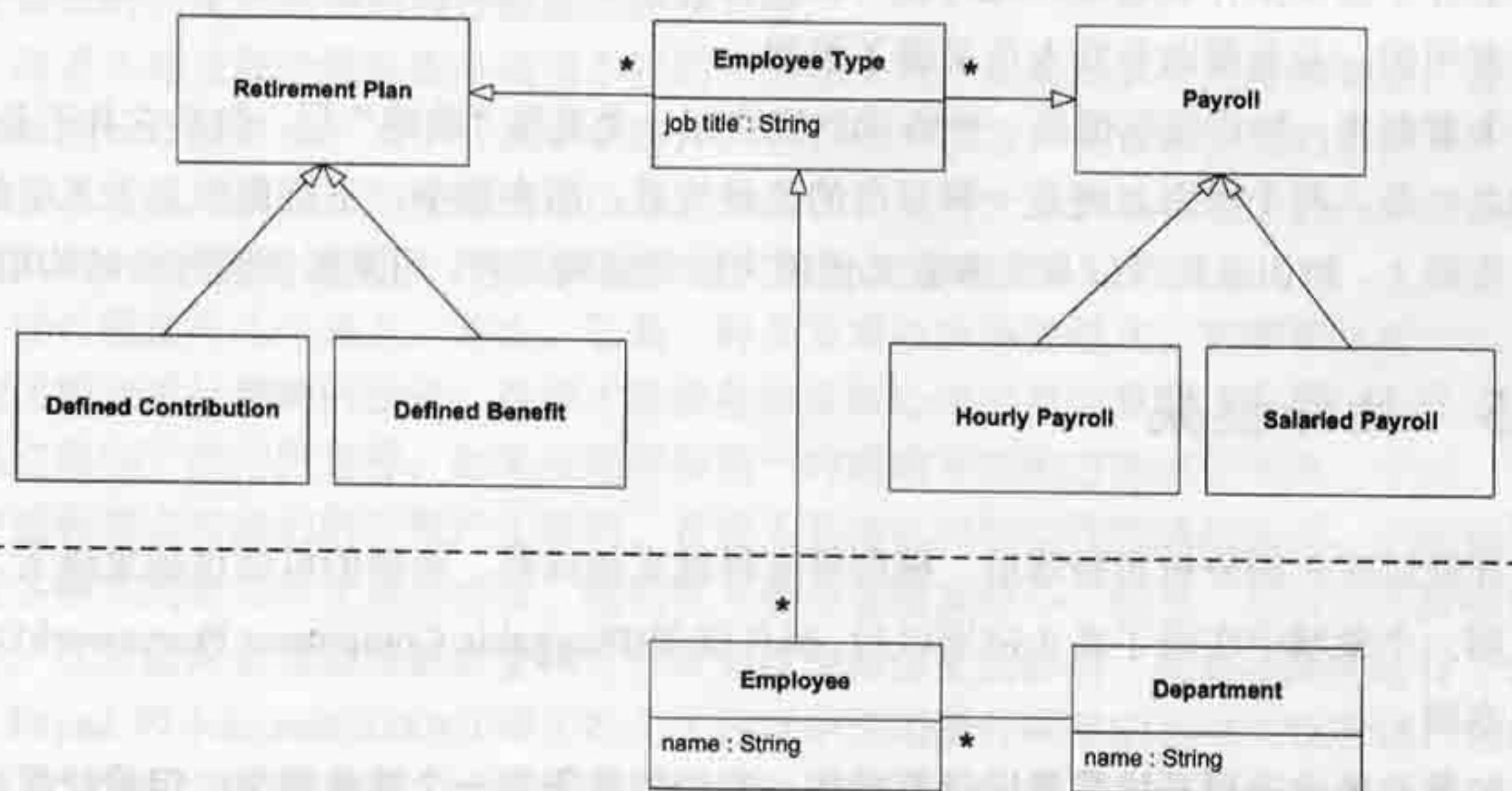


图 16-22 明确地把 Payroll 对象从 Employee Type 中分离

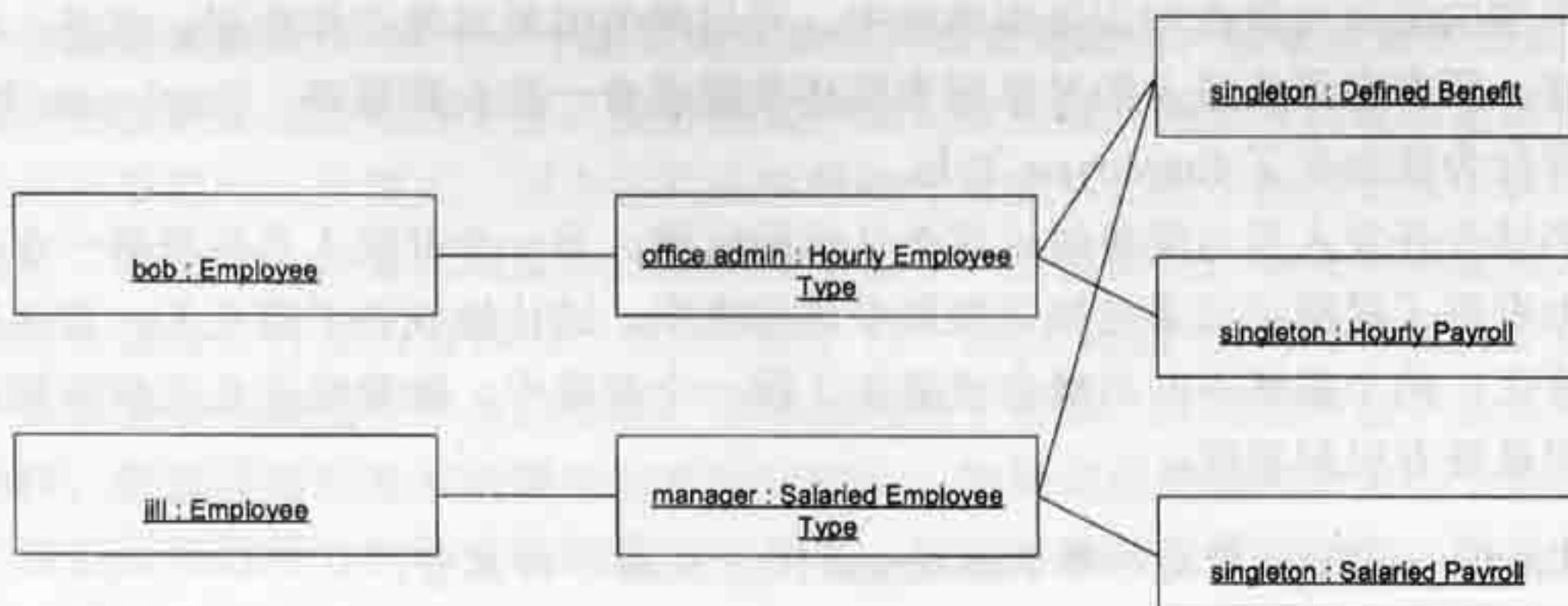


图 16-23 每个 Employee Type 都对应一个 Retirement Plan 和一个 Payroll

在上面的例子中，特有的访问限制和 thing-thing 类型的关系暗示了模型中的知识级别。只要发现了知识级别的存在，它提供的清晰结构就能够帮助我们获得其他的认识——通过分离 Payroll 来解决两种重要领域概念混在一起的问题。

像其他大比例结构一样，知识级别结构并不是绝对必要的。没有它对象也许可以照样工作，我们也许照样能发现 Payroll 并把它和 Employee Type 分离开来。有时，这种结构可能并不会带来什么效果，此时就可以把它丢弃了。但是目前，它看起来在这个系统中是有用的，并且帮助开发人员掌握了模型。

乍看起来，知识级别像是一种特殊的职责层，尤其像“策略”层，但是它并不是层。理由之一是，两个级别之间是一种双向的依赖关系，而在层中，下层是独立于上层的。

实际上，知识级别可以与大多数其他的大比例结构共存，用来提供额外的结构信息。

## 16.5 插件框架

在经过深入的分析和精炼后，模型将变得越来越成熟，突破的机会也越来越大。只有在同一个领域中实现了多个应用以后，插件框架(Pluggable Component Framework)才会浮出水面。

如果有多个应用系统需要进行互操作，它们都基于同一个抽象模型，但设计是相互独立的，那么我们就必须在多个限界上下文之间进行转换，这对系统的集成造成了限制。如果团队之间的工作不紧密配合，那么他们就无法使用共享内核。功能的重复和割裂增加了软件的开发和安装费用，互操作也将变得非常困难。



一些优秀的项目把设计分解成许多组件，每个组件负责实现一种功能。通常所有的组件都插入到一个中心 hub 上，这个 hub 支持所有用到的协议，并且知道如何与接口进行通信。还有一些其他的连接组件的模式。这些接口和中心 hub 的设计必须进行协调，以便 hub 能把不同的组件连接起来；但是组件的内部设计可以更加独立一些。

有几个广泛使用的技术框架提供了对这种模式的支持，但这只是次要的问题。只有当技术框架能解决一些重要的技术问题(例如组件分布或者在不同应用中共享组件)时，才需要使用它。插件框架的基本模式是对职责的一种逻辑组织。它很容易就能够应用到一个简单的 Java 程序中。

因此：

对接口和交互过程的抽象核心进行精炼，并创建一个框架来允许这些接口的不同实现能够被自由替换。插件框架还使得所有应用程序都能使用这些组件，只要它严格地遵循抽象核心的接口来进行操作。

高层次的抽象在整个系统范围内都可以通用的、可界定的；而特殊化只发生在模块中。中心 hub 是共享内核中的抽象核心。但是多个限界上下文都可以用同一个组件接口来封装，因此当系统中存在着许多来源不同的组件，或者需要用组件来封装已有的系统进行集成时，使用这种结构可能就会特别方便。

这并不是说每个组件都必须用不同的模型来实现。如果团队采用持续集成，那么他们可以在一个限界上下文中开发多个组件。他们也可以为一批紧密相关的组件另外定义一个共享内核。所有这些策略都可以共存于插件框架的大比例结构之中。在一些情况下还可以有一种选择，即用一种公布语言来作为中心 hub 的插入接口。

插件框架有几个缺点。首先，它是一种非常难以应用的模式。它需要对接口以及一个深层模型进行精确的设计，这样才能抓住抽象核心中必要的概念。另外一个较大的缺点是它限制了应用的选择。如果应用需要用一种截然不同的方法来使用核心领域，那么这种结构就会对我们的开发产生障碍。开发人员可以对模型作特殊化处理，但是如果不能改变各种不同组件所使用的协议，他们就不能改变抽象核心。结果，对核心的持续精炼过程，以及面向更深层理解的重构过程或多或少都会受到影响，而难以继续进行下去。

Fayad 和 Johnson(2000)介绍了在几个领域中使用插件框架的各种大胆尝试，包括对 SEMATECH CIM 的讨论。这些框架的成功取决于许多因素，其中一个最大的障碍可能是，我们要必须获得深刻的理解才能设计出有用的框架。插件框架不应该是最先应用到项目上的大比例结构，也不应该是第二个。这些最成功的例子都是在完整地开发了多个专门应用以后才得到一个成功的插件框架的。



### 示例：SEMATECH CIM 框架

在一个生产电脑芯片的工厂里，硅晶片从一台机器移到另外一台机器需要通过很多道工序，直到在它们上面掩膜印制微电路，然后进行光蚀刻后才能结束。该厂需要一种软件来对每块硅晶片进行跟踪，记录它被处理过的工艺过程，并且指导工厂的工人或自动化设备按照正确的顺序把它放到下一台机器上，以便执行下一道工序。我们称这种软件为制造执行系统(MES, Manufacturing Execution System)。

工厂里使用的数百台机器是从许多不同的厂家购买来的，工序中的每一步都仔细制定了相应的处理过程。要开发一个如此复杂的 MES 软件不仅让人望而生畏，而且开发的费用也高得惊人。为了解决这个问题，SEMATECH 工业协会开发了 CIM 框架。

CIM 框架非常庞大复杂，它包括了很多方面，但是这里只涉及到两个相关的方面。第一，该框架为半导体 MES 领域的基本概念定义抽象接口，换句话说，这是一种抽象核心形式的核心领域。这些接口定义包括了行为和语义，如图 16-24 所示。

如果一个设备制造商生产了一种新机器，他们就必须开发 Process Machine 接口的一种专用实现。如果他们坚持使用这种接口，他们的机器控制组件就能够插入到任何基于 CIM 框架的应用中(称为 CIM 应用)。

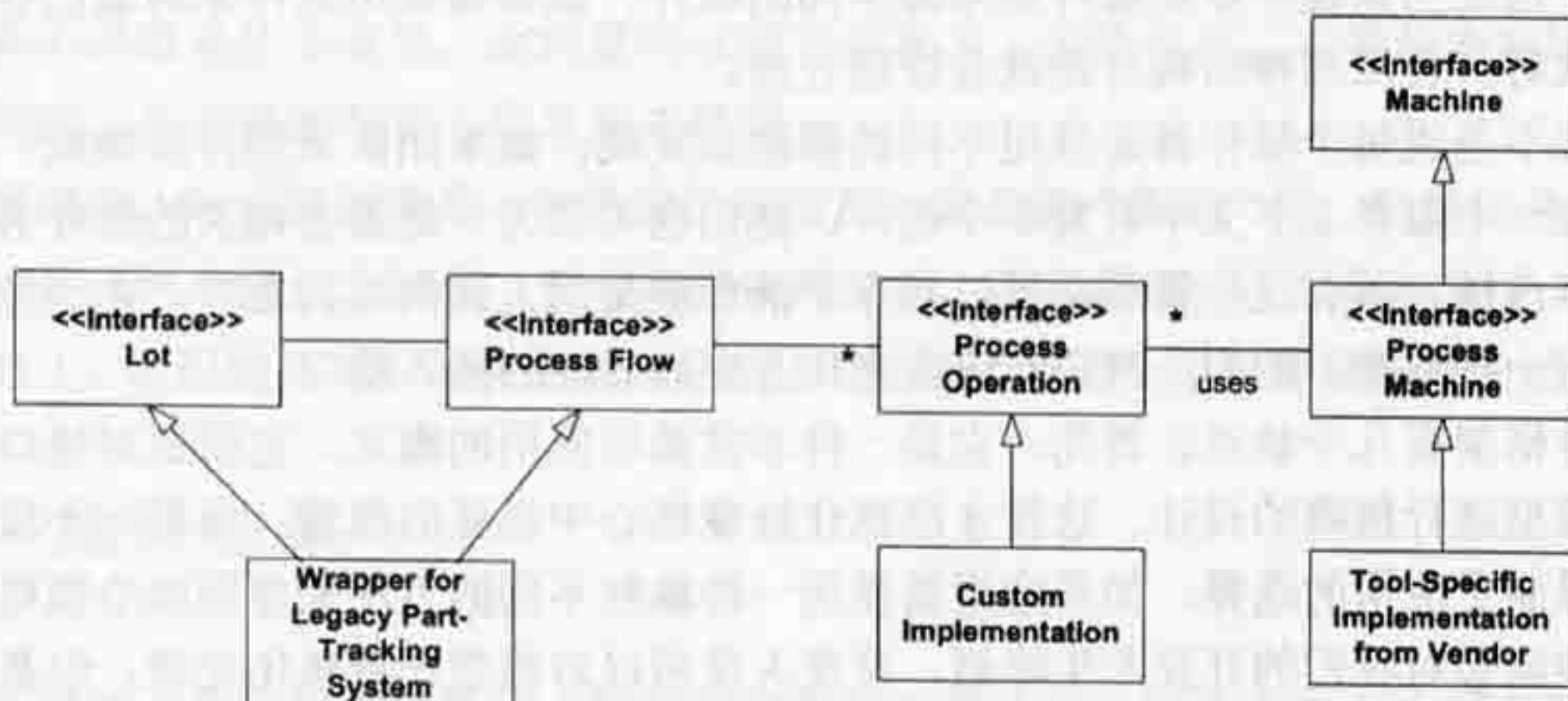


图 16-24 一个高度简化的 CIM 接口子集及其实现示例

根据这些接口，SEMATECH 定义了应用中这些组件的交互规则。任何 CIM 应用都必须实现一种协议，为实现了其中的一些接口的对象提供服务。如果 CIM 应用实现了这个协议，并且应用程序严格地遵循了抽象接口，那么应用程序就可以使用这些接口提供的服务，而不管它们是如何实现的。接口和协议的组合构成了一种具有严格限制的大比例结构。

图 16-25 显示了 MES 软件的工作过程。

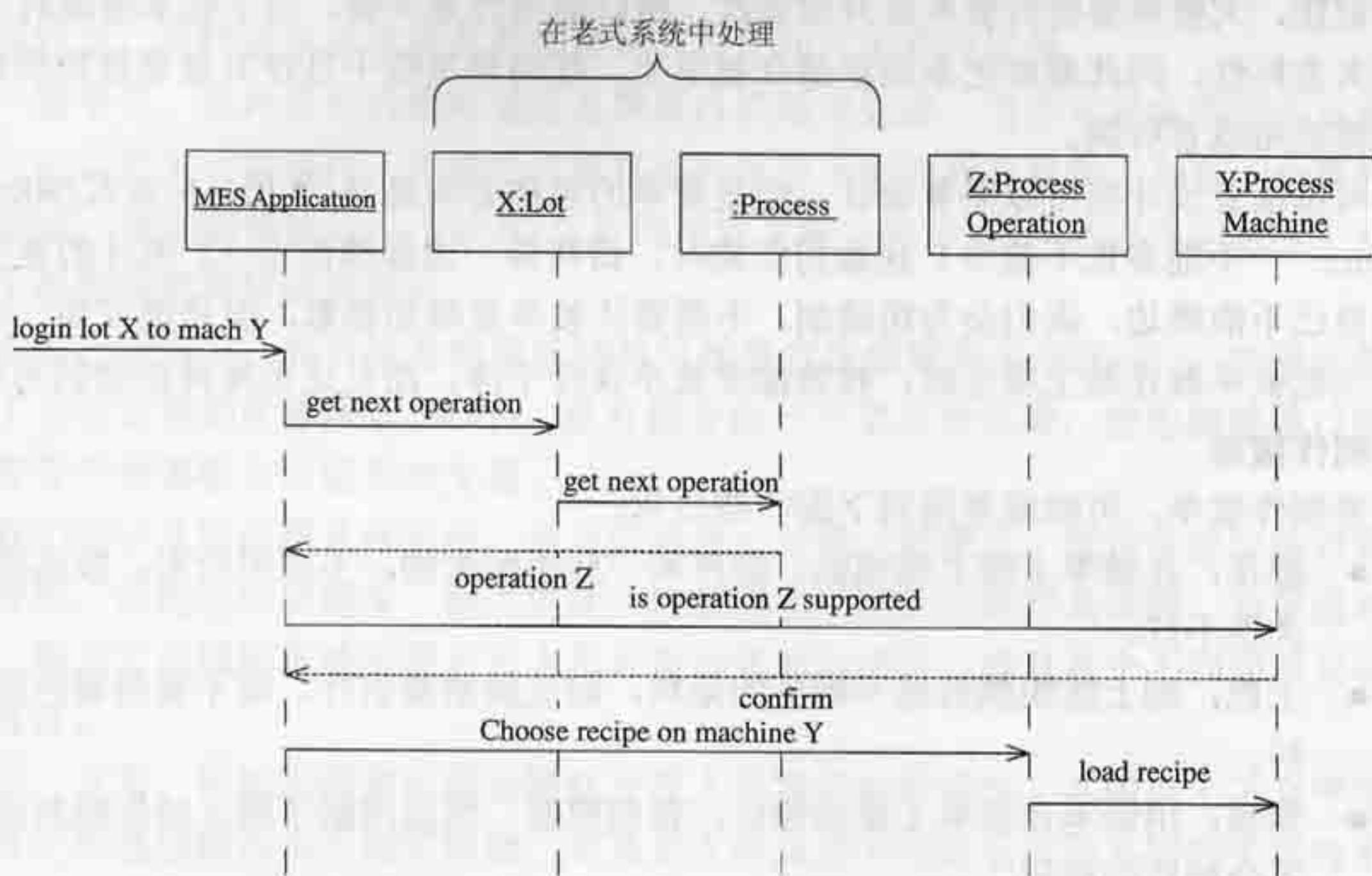


图 16-25 用户把晶片转到下一台机器并记录移动信息

该框架需要非常专门的基础结构需求。它与 CORBA 紧密关联，利用了 CORBA 提供的持久性、事务、事件以及其他服务。但是我们感兴趣的是插件框架的定义，它使得独立开发的软件能够平稳地集成到庞大的系统中来。没有人知道这种系统的所有细节，但是每个人都能理解系统的全貌。

### 一面由 4 万多张艾滋被单组成的大被单是如何由成千上万个独立工作的人完成的

几个简单的规则构成了一个艾滋纪念被单的大比例结构，而具体细节是由各个志愿者去完成的。注意下面的规则如何涉及整体任务(悼念死于艾滋病的人们)，使被单能够被缝合起来的特征，以及进一步处理被单的能力(例如折叠被单)。

#### 如何缝制每一块被单

【摘自 AIDS 语录纪念网站，[www.aidsquilt.org](http://www.aidsquilt.org)】

#### 设计被单

包括要悼念的人的名字。可以自由地加入其他信息，例如出生日期、死亡时间、出生地等。每张被单只限于悼念一个人。



### 选择材料

记住，大被单要被折叠和展开很多次，所以耐用性是关键。由于胶水会随时间推移而失去粘性，因此最好把东西都缝在被单上。使用轻重适中且没有延展性的织物，例如棉帆布或者府绸。

可以横着设计也可以竖着设计，但是被单的规格必须是 3 英尺×6 英尺(90cm×180cm)——不能多也不能少！在裁剪织物时，请在每一边都预留 2~3 英寸的宽度。如果自己不能缝边，我们会为您缝制。不需要往被单里填加棉絮，但是建议加一层衬垫。当把被单放在地上展示时，衬垫能使被单保持干净，而且还能维持织物的形状。

### 制作被单

要制作被单，可能需要用到下面一些技术：

- 缝花：在被单上缝上编织品、信件和一些小纪念物。不要用胶水，胶水的耐久性不好。
- 上色：刷上纺织颜料或不褪色的染料，耐洗烫油墨也行。请不要用褪色的颜料。
- 描模：用铅笔在被单上描出设计，得到模板，然后用刷子刷上纺织颜料或者不会褪色的标记。
- 拼贴：确保添加到被单上任何材料都不会弄破被单(避免用玻璃和金属片)。要避免使用体积很大的东西。
- 加照片：添加照片或者字母的最好方法是把它们影印到纯棉织物上，然后再把它缝到被单上(不要放在中间以避免折叠)。

## 16.6 结构的约束

本章讨论了多种大比例结构模式，从约束非常宽松的系统隐喻到约束非常严格的插件框架。当然，还有很多其他的结构，甚至在同一个通用结构模式中，如何确定规则的约束力都可以有多种选择。

例如，职责层表示了模型概念的一种划分以及它们的依赖关系，但是可以通过添加规则来定义各层之间的通信模式。

就拿一个加工设备来说，软件指挥机器将加工品的每个部分根据工艺方法进行加工。决策层给出正确的处理过程，然后在作业层上得到执行。但是，在工厂实际生产过程中出现错误是不可避免的，实际情况会与软件的规则不符。在这个时候，作业层就必须反映出现实世界的真实情况——也就是说，如果某个零件被偶然地放到错误的机器上进行



处理，作业层只能无条件地接受这个错误，而不能纠正这种情况。这种异常情况需要通知给更高的层，如让决策层运用其他策略来纠正这种错误。但是，作业层并不知道上层的任何信息。这种通信必须避免在低层和高层之间产生双向依赖。

一般来说，这种信号传输将通过某种事件机制来完成。当作业对象的状态发生变化时，它们会产生一些事件。决策层对象将监听来自下层的相关事件。当有一个事件违反了规则时，该规则将执行一个动作(这是规则定义的一部分)来作出适当的响应，或者产生一个事件向更高层请求帮助。

在银行示例中，有价证券价值的转移引起资产价值发生改变(作业)。如果这些价值超过了有价证券的配额限制(策略)，就可能会向一个交易商报警，使他能够通过购买或者出售资产来调整有价证券的交易。

我们可以具体问题具体对待，也可以设计一种统一的模式，供特殊层中的对象在交互时使用。结构的约束越多，模式的统一性就越高，设计就越容易理解。如果结构适合的话，那么它的规则就能促使开发人员开发出优秀的设计，而且各个不同的部分可以配合得很好。

另一方面，结构的约束可能会降低开发人员需要的灵活性。在限界上下文中采用非常特殊化的通信路径是不切实际的，尤其是在涉及到不同的系统和不同的实现技术时。

所以不要老想着去创建框架并且严格控制大比例结构的实现。大比例结构的最大作用是维护概念上的一致性，并且帮助我们了解领域。每个结构性规则都应该使开发更加容易实现。

## 16.7 重构到合适的结构

现在，过度依赖于预先设计的传统方法正在受到挑战，一些人会把大比例结构看成一种倒退，让项目退回到使用瀑布模型的痛苦时代。但是实际上，要找到一种有用的结构，惟一的方法就是要对模型和问题有非常深入的理解，而获得这种理解的一种可行方法就是采用迭代式的开发过程。

要坚持使用渐进顺序进行开发，那么团队就必须在项目的整个生命周期内大胆地重新考虑大比例结构的适用性。团队不应该被以前构思的结构捆住手脚，因为那个时候还没有人能够很好地理解领域或者需求。

遗憾的是，这种渐进性意味着在项目开始时是无法确定最终结构的，它也意味着在开发过程中必须进行重构才能利用新的结构。这种重构可能会非常昂贵和困难，但它是必要的。下面是一些用来控制开发费用并获取最大效果的通用方法。



### 16.7.1 最小化

保持低开发费用的关键是保持一个简单的、轻量级的结构。不要企图开发一种万能的结构。只关注最重要的部分，其他问题可以逐个去解决。

在开发早期，选择一种宽松的结构(例如系统隐喻结构或者分成几个职责层的结构)会很有帮助。最小化的宽松结构仍然可以提供轻量级的指导方针，帮助我们防止开发的混乱。

### 16.7.2 交流和自律

整个团队必须遵循所使用结构来进行开发和重构。为了做到这点，结构必须能被整个团队所理解。术语和关系必须加到通用语言中。

大比例结构能够为项目提供一份术语表，从而概括地对系统进行描述，并且使不同分工的人能作出协调的决策。但是，由于大多数大比例结构都是一种宽松的概念性指导，所以开发团队必须自律。

如果开发人员不遵从一致性原则，结构就会被逐渐破坏。例如，结构与模型细节或具体实现的关系没有在代码中显式地表达出来，或者功能测试不是依赖于结构构建的。另外，结构往往是抽象的，所以应用的一致性在一个大团队(或者多个团队)中可能难以维护。

仅仅通过谈话并不足以在大多数团队之间维护大比例结构的一致性。关键是要把结构融入到项目的通用语言中，并让每个人严格地使用这种语言。

### 16.7.3 结构重组产生柔性设计

结构的任何改变都可能会导致大量的重构。随着系统复杂性的增加和对系统理解的深入，结构也会得到发展。每一次改变结构，都会导致整个系统根据结构的新要求而作出相应改变。很明显，这需要做大量的工作。

实际情况并不像听起来那么糟糕。我注意到，有大比例结构的设计通常比没有大比例结构的设计更加容易转换，即使是从一种结构转换到另外一种结构(比如说从隐喻结构到分层结构)也不例外。我不能完全解释这种现象。部分原因可能是，如果能够理解一个系统的结构安排，那么对结构进行重新安排就会比较容易——而已有的结构正好为理解提供了帮助。还有一部分原因可能是维护初期结构的原则渗透到了系统的各个方面。但是我认为还有更多的原因，因为如果系统此前使用过两种结构，那么对它进行转换会更容易些。

新的皮夹克通常又硬又僵，但是穿上一天之后，肘部就会比较容易弯曲了。再穿几



次以后，肩部已经变得宽松，因而皮夹克就更加服帖了。穿了几个月以后，皮革会变柔软，穿起来就会觉得舒适自在了。对模型进行反复转换就像穿皮夹克一样，是同一个道理。新的知识被不断添加到模型中，那些关键的转换点被逐渐确定下来并得到了灵活的处理；同时，模型中的稳定部分也获得了简化，领域的更广泛的概念轮廓逐渐在模型结构中浮现出来。

#### 16.7.4 精炼为开发指路

另外一种应该施加到模型上的关键方法是持续精炼。它以不同的方式降低了改变结构的难度。首先把机制、通用子域以及其他辅助结构从核心领域中去掉，减少结构重组所涉及的元素。

如果可能的话，这些辅助元素的定义应该能使之简单地融入大比例结构之中。例如，在一个根据职责分层的系统中，通用子域的定义应该能够使之保持在一个特定的层中。如果使用插件框架，那么一个通用子域可能完全属于一个组件，或者可能是一组相关组件的一个共享内核。我们可能必须对这些辅助元素进行重构，以便能在结构中为它们找到合适的位置。但是，这些元素的移动与核心领域无关，并且影响的范围更小，因此改变起来更加容易。最终它们将变得更加次要，因此精炼的影响就更小了。

精炼原则和重构甚至可以应用到大比例结构本身。例如，在最开始时我们可以根据对领域的肤浅理解来分层，随着系统基本职责的进一步抽象，这些层会被逐步替换掉。这能够帮助我们深入地了解设计——这也是精炼的目标。精炼还是一种手段，它使得在大比例情况下管理系统更简单、更安全。



## 第17章

# 综合应用战略性设计

前面三章中我们介绍了许多关于领域驱动战略性设计的原则和技术。在一个庞大而复杂的系统中，您也许会在同一个设计中用到多种不同的技术。如何使一个大比例结构与上下文图很好地配合起来？怎样才能把这些技术很好地结合在一起？首先要做什么？其次做什么？再次呢？您打算如何来运筹您的战略？

## 17.1 大比例结构和限界上下文的结合

图 17-1 结合了大比例结构和限界上下文。

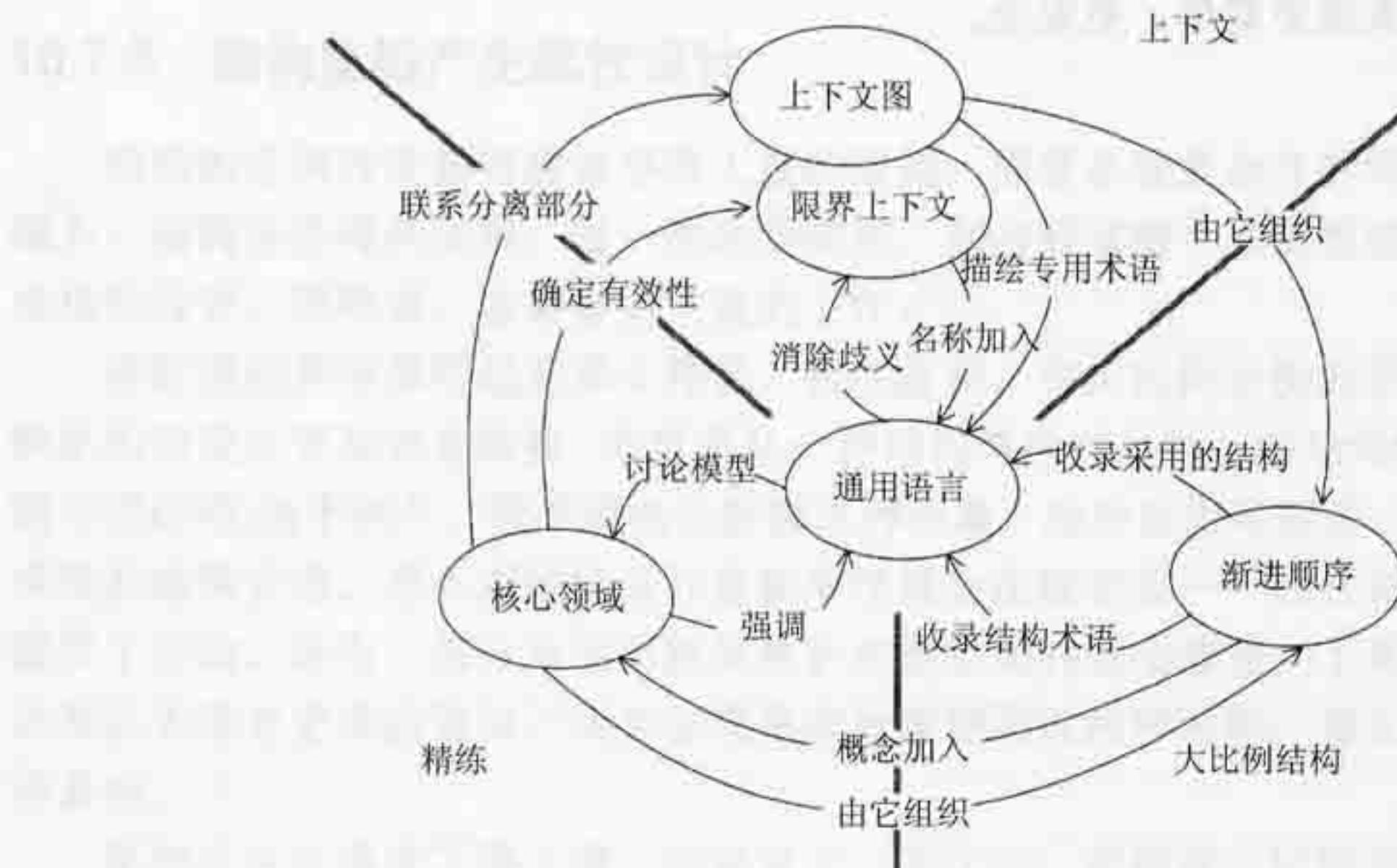


图 17-1 结合大比例结构和限界上下文



限界上下文、精炼和大比例结构这 3 个基本的战略性设计原则是不可互换的，在很多方面它们互为补充、相互影响。例如，一个大比例结构不仅可以存在于一个限界上下文中，还可以跨越多个限界上下文并且组织相应的上下文图。

前面使用职责层结构的例子都限制在一个限界上下文中，如图 17-2 所示。这是解释职责层最简单的方法，也是这个模式的通常用法。在这种简单的情况下，各层名称的含义都会受到它所在上下文的限制，就像上下文中模型元素的名称或者子系统接口的名称一样。

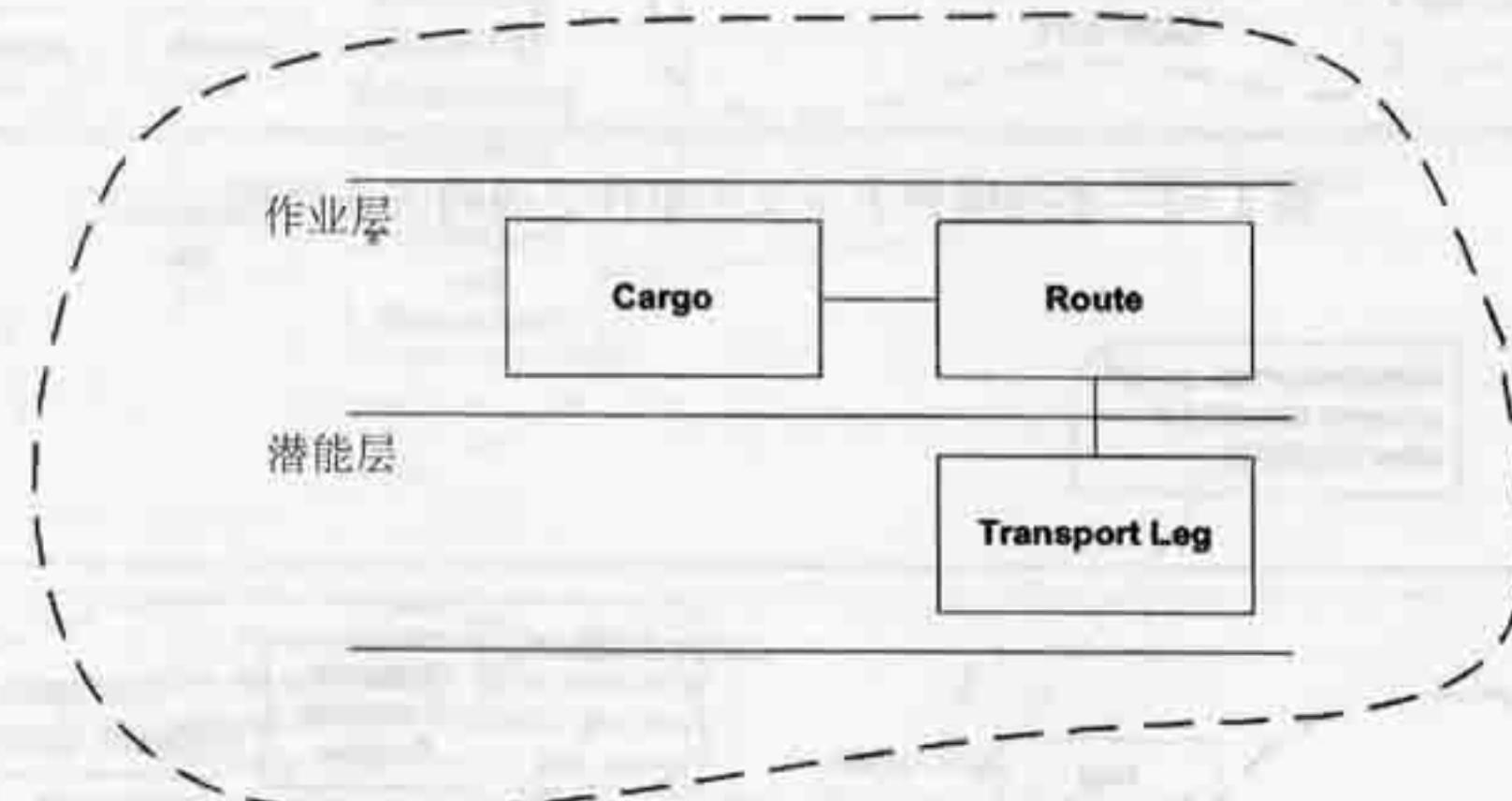


图 17-2 在一个限界上下文中创建一个模型

这种局部结构在一个复杂而一致的模型中是非常有用的，它提高了系统复杂性的上限，使我们能在限界上下文中维护更多的信息。

但是在许多项目中，更大的挑战是要知道怎样把各个完全不同的部分结合在一起，如图 17-3 所示。它们可能被划分到了不同的上下文中，但是各个部分在整个集成系统中的作用以及它们之间的相互关系是什么？搞清楚了这些问题，然后才能用大比例结构来组织上下文图。在这种情况下，结构的术语应用于整个项目(或者至少是项目中一些有明确边界的部分)。

假设我们想要采用职责层结构，但是老式系统与所需的大比例结构并不一致。此时我们是否必须得放弃这种分层结构呢？回答是否定的，但是我们必须确定这个系统在结构中的实际位置，如图 17-4 所示。事实上，分层结构可以帮助我们来刻画这个系统。老式系统提供的服务实际上可能只被限制在几个层次上。如果老式系统与特定的职责层结构完全相符，那么就很简明地描述出了系统的范围和角色的主要方面。

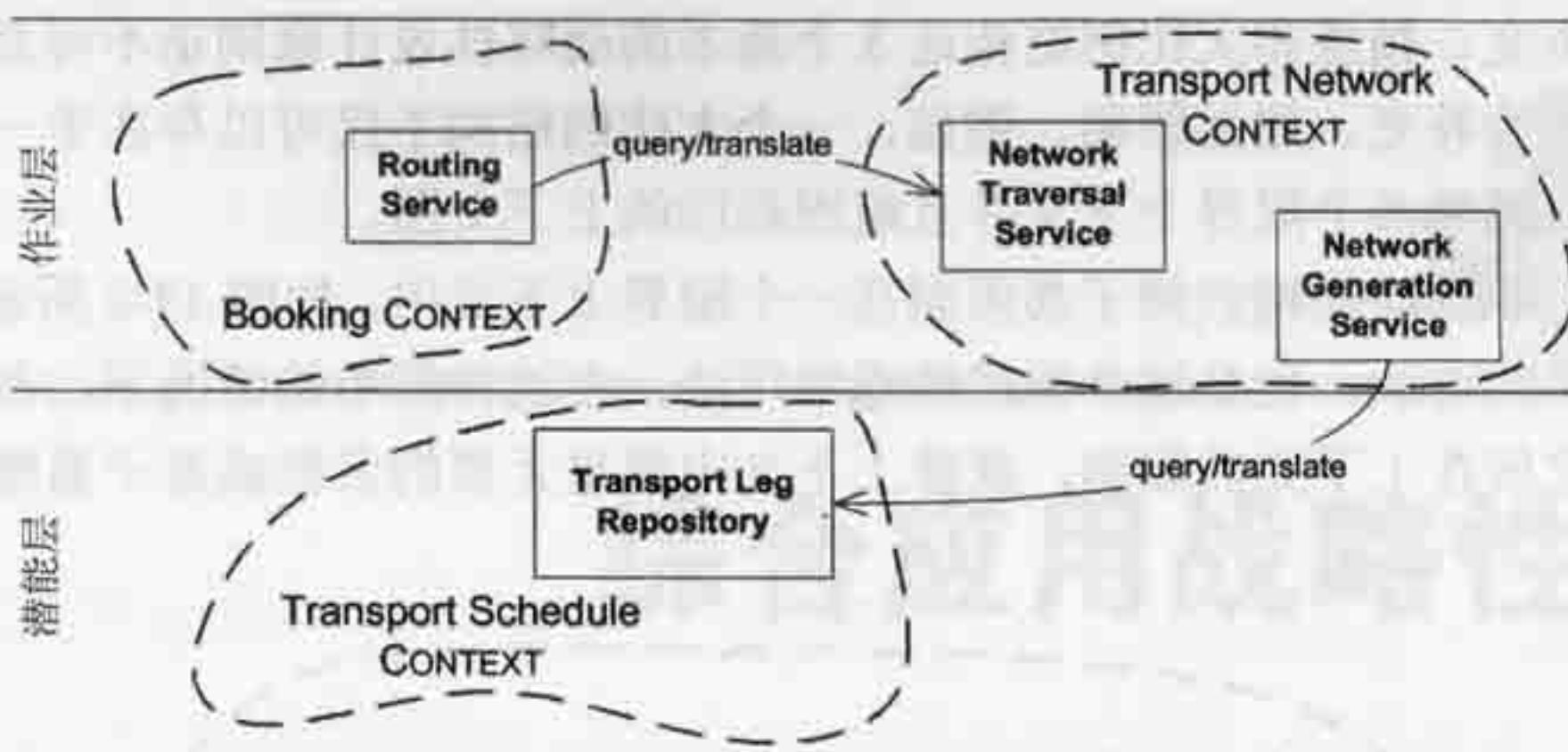


图 17-3 不同限界上下文的组件之间的关系结构

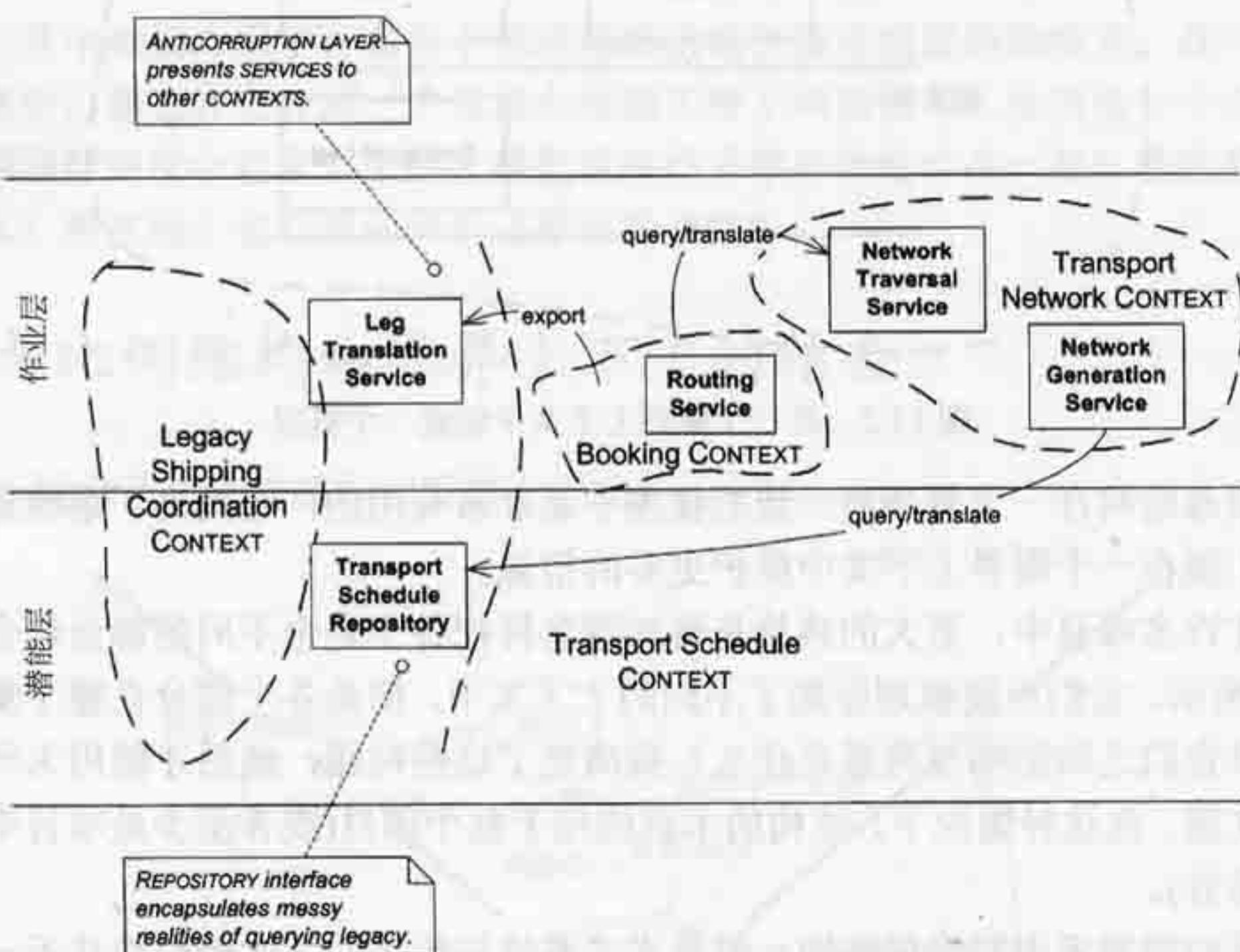


图 17-4 允许组件跨层的结构

如果这个子系统的性能可以通过一个外观来访问，那么我们也许可以将该外观提供的每一种服务都设计在一个层中。

在这个例子中，运输调度应用是一个老式系统，其内部结构表现为一个统一的环境。但是如果在一个项目中，有一个构建好的大比例结构跨越了这个上下文图，那么开发团队也可以根据他们熟悉的分层结构来定制模型，如图 17-5 所示。

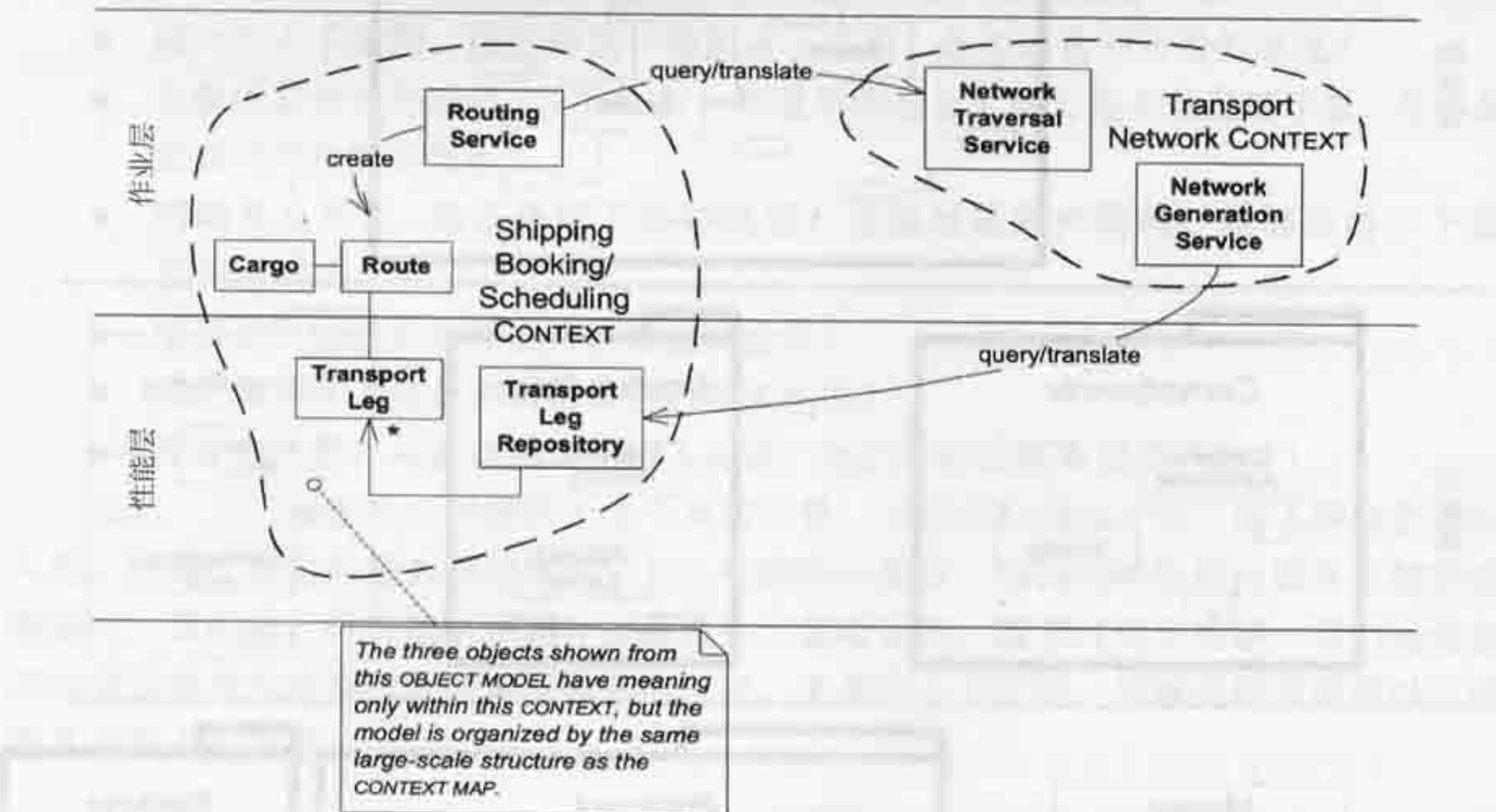


图 17-5 在一个上下文中和跨越上下文图使用的同一种结构

显然，由于每一个限界上下文都是它自己的名称空间，所以在一个上下文内可以用一个结构来组织对象模型，而在邻近的上下文中使用另一个结构，由它来组织上下文图。然而，我们应该把握上述方法的适度性，因为如果沿着这个思路走下去，大比例结构就会失去作为项目统一概念的价值。

## 17.2 大比例结构和精炼的结合

大比例结构和精炼的概念也是相互补充的。大比例结构能帮助解释核心领域内部的和通用子领域之间的关系，如图 17-6 所示。

与此同时，大比例结构本身也可能是核心领域的一个重要部分。例如，将能力层、作业层和决策支持层区分开来，能使我们对软件需要解决的商业问题获得更深入的认识。这种认识对于划分为多个限界上下文的项目尤其有用，因为它能够限制核心领域中模型



对象的含义，使其牵涉到的范围不会太大。

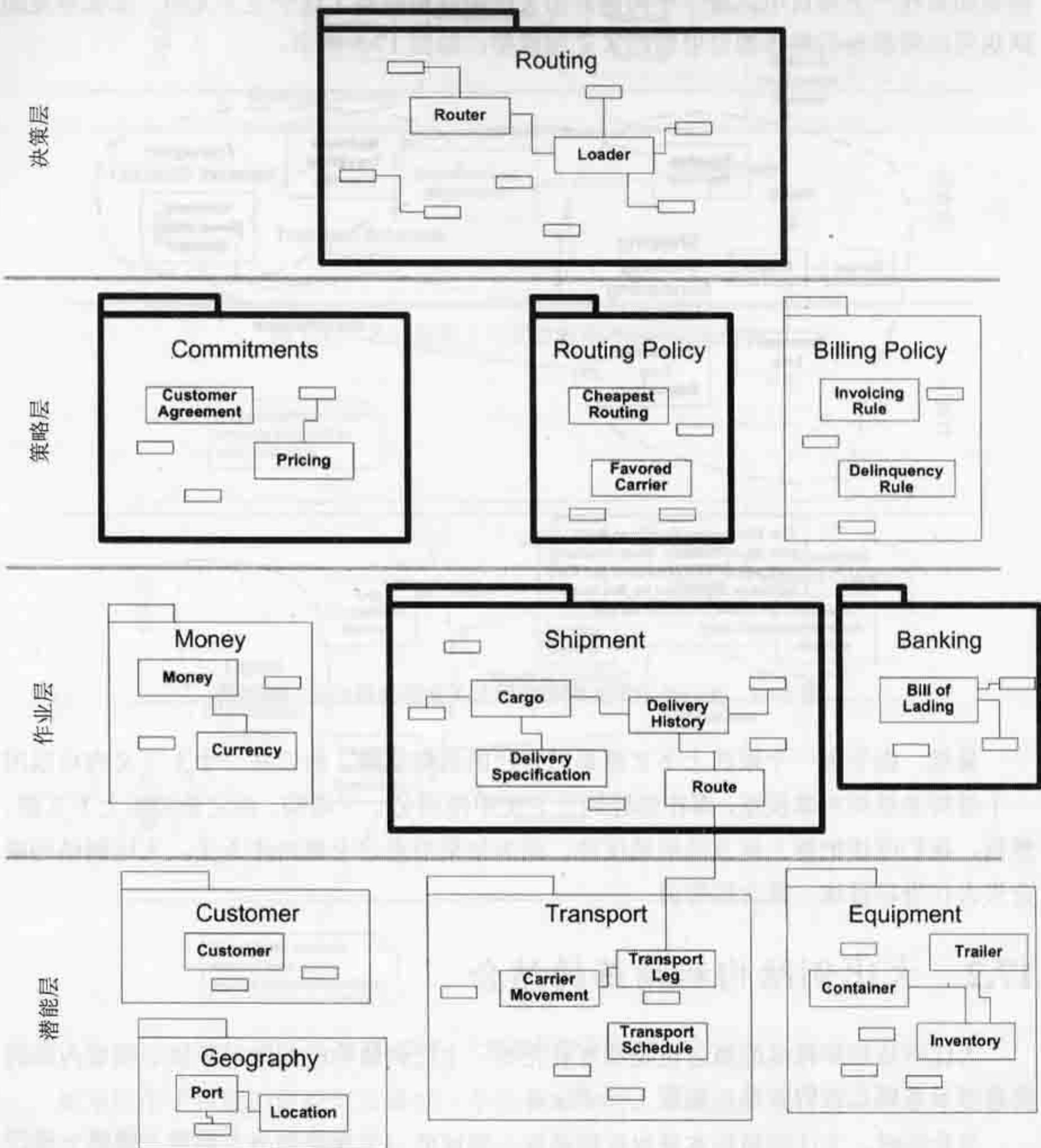


图 17-6 分层结构阐明了核心领域(粗体表示)和通用子领域的模块



## 17.3 首先进行评估

在制定项目的战略性设计时，我们必须首先对目前的情况进行评估。

- 画一个上下文图。能否画出一致的上下文图？是否还有不明确的情况？
- 注意项目中使用的语言。有没有一种通用的语言？语言是不是足够丰富，是不是能够对开发提供帮助？
- 明确重点所在。是否确定了核心领域？有领域愿景声明吗？能够写出一个愿景声明吗？
- 项目采用的技术是否支持模型驱动设计？
- 团队的开发人员是否具备必要的技术技能？
- 开发人员是否具备该领域的相关知识？他们对该领域有没有兴趣？

当然，我们现在不可能得到十全十美的回答。以后我们对这个项目的了解会更加深入的。但是，这些问题为我们提供了一个明确的出发点。当我们对这些问题有了初步的答案时，我们就已经开始认识到什么是最迫切需要做的。随着时间的推移，我们还会修正这些答案来反映新出现的情况和新的认识，尤其是上下文图、领域的愿景声明以及所有其他相关的信息。

## 17.4 由谁制定策略

在开始开发应用程序之前，架构通常要预先确定下来，而且制定架构的团队通常比做应用程序开发的团队在组织中要拥有更大的权力。但是，我们并非一定要遵循这种方式，因为这种方式通常并不是很有效。

战略性设计必须应用于整个项目。组织一个项目的方式有很多种，这里我不会作过多的介绍。但是，为了使决策过程更有效率，我们需要遵循一些基本的原则。

首先，让我们大致看看我曾经见到过的两种具有一定实用价值的组织风格。

### 17.4.1 在开发过程中自发产生

一个善于交流、严谨自律的团队可以在没有核心权威的情况下正常工作，并且遵循渐进顺序的模式来获得一系列原则性共识，使得项目的开发秩序能够自发地建立起来，而不是靠命令来强制推行。

这是极限编程团队使用的典型模式。从理论上讲，任意一个编程对子都可以根据自



己的认识完全自发地产生出一个结构。一般而言，如果团队中有一个人或几个人负责对大比例结构进行监督，会对保持结构的一致性很有帮助。如果是由一个进行实际编程的开发人员来充当这种非正式的领导者——他负责进行仲裁和沟通，但并不是决策的惟一来源，那么这种方法尤其有效。在我所见过的极限编程团队中，这种战略性设计的领导能力似乎都是自发产生的(通常是在教练(Coach)中产生)。不管这个领导是谁，他仍然是开发团队中的一员：所以，在这样的开发团队中至少要有几个能干的、能够作出重大设计决策的人。

如果一个大比例结构跨越了多个团队，那么那些联系密切的团队就可以开始进行非正式的合作了。在合作过程中，每个团队还是会产一些关于大比例结构的认识，其中有些设计问题还需要由各个团队的代表组成一个非正式委员会来进行讨论，由这个委员会对设计的影响进行评估，然后决定是采用、修改还是搁置这个设计。各个团队都力求能够在这种松散的合作关系下共同发展。当参与的团队相对较少、设计能力不相上下，而且各个团队的结构需求具有足够的相似性(从而能够使用同一个大比例结构)，并希望取得协调一致的时候，这种自发产生的风格就能取得较好的效果。

#### 17.4.2 以客户为中心的架构团队

当多个团队需要共享同一个战略时，采用集中的决策就比较具有吸引力了。虽然有些模型由于架构师脱离实际而导致失败，但这并不是一种必然现象。架构团队与其他应用程序开发团队的地位是平等的，他们帮助其他团队调整和协调大比例结构、限界上下文的边界，以及其他涉及到多个团队的技术问题。为了在这个过程中发挥作用，他们必须树立正确的观念，强调应用程序开发的重要性。

从组织结构上看，这种团队似乎与传统的架构团队很相似，但是实际上二者的行为有着很大的不同。架构团队的成员是真正的合作者，他们同开发人员一起寻找模式、与各个团队一起进行试验，对系统进行精炼，并融入到开发过程中来。

我曾多次见过这种情况：项目最终由一个架构师来领导，他负责完成下面列出的大部分工作。

### 17.5 制定战略性设计决策的 6 个要点

#### 1. 决策必须传达给整个团队

很明显，如果没有人知道这个战略并依此行事，那么战略性决策就毫无意义了。因此，架构团队必须成为团队的组织中心——要具有“权威”，这样规则才能得到贯彻施



行。然而讽刺的是，那些脱离实际的架构师们往往会被团队忽视或者架空。如果架构师缺乏来自实践中的反馈，又企图把他们自己的规则强加到实际应用中来，导致设计出来的方案不切实际，那么开发人员也别无选择，只能把他们绕过去。

在沟通良好的项目中，来自应用团队的战略性设计可以更高效地传达到每一个人。这是一种恰当的战略设计，是明智的集体决策。

不管是什么样的系统，我们都应该更加关注开发人员与战略的实际关系，而不是去关注管理部门所授予的权力。

## 2. 决策过程必须接受反馈

组织规则和大比例结构是非常微妙的，因此其创建和精炼需要我们对项目需求和领域概念有深刻的理解。这种深刻的理解只有应用程序开发团队的成员才会具有——这也解释了为什么尽管很多架构师才华横溢，但架构团队设计出来的结构却很少能为开发提供帮助。

与技术基础结构和架构不同，虽然战略性设计影响着整个开发过程，但是它本身并不需要很多代码；它所需要的是与应用团队的开发过程联系起来。有经验的架构师会听取来自不同团队的意见，并能够更容易地提出综合性的解决方案。

我曾经在一个技术架构团队中工作过，我们的成员与希望使用其框架的各个应用程序开发团队进行了密切的交流。这种交流使得我们的架构团队理解了开发人员所面临的挑战、获得了实践经验，同时又使得开发人员学到了处理框架使用过程中一些细微问题的知识。战略性设计同样需要这种紧密的反馈循环。

## 3. 制定计划必须顾及发展

有效的软件开发是一种动态过程。当开发团队必须作出改变时，如果高层次的决策早已确定下来的话，他们就不会有太多的选择。渐进顺序模式强调随着理解的深入对大比例结构进行改变，从而能够避免陷入这种困境。

如果预先定下的设计决策太多太死，开发团队可能就会举步维艰，失去了处理问题的灵活性。所以，协调原则是非常重要的。结构必须随着项目的开发过程发展和改变，而且它也不应过多地耗费应用程序开发人员的精力，因为他们的任务已经很艰巨了。

有了强有力的反馈支持，当我们在应用程序开发过程中遇到障碍或者发现意外机会时，我们就能得到更多的点子来应付这些问题。

## 4. 架构团队的成员不必都是最好的和最聪明的

架构的设计需要经验丰富的设计人员，而这样的人才可能相当缺乏。因此，管理者



往往把最有能力的开发人员调到架构团队和基础设施团队中来，以便让这些高级设计者充分发挥作用。这些管理者认为，能够有机会获得更大影响力或者去解决那些更有趣的问题，会让开发人员觉得很有吸引力。而且，加入精英团队也是一件很荣耀的事情。

这样做的结果是，通常只有那些技术最差的开发人员才被留下来做实际的应用程序开发。但是，创建优秀的应用程序是需要设计技巧的；这样做无异于自取其败。即使战略团队创造出了非常好的战略性设计，应用程序开发团队中也没有熟练的设计开发人员来实现它。

另一方面，管理者几乎从来不把那些设计技巧较差，但是在领域上具有丰富经验的开发人员吸收到架构团队中。战略性设计并不是一种纯技术性的任务；把那些具有丰富领域知识的开发人员排除在外，只会让架构师们裹足不前。此外，领域专家也是必须的。

所有的应用程序开发团队都需要优秀的设计者，所有试图进行战略性设计的团队都必须具备领域知识——这是非常关键的。我们可能只需雇用更多的高级设计人员；让开发人员兼职从事架构开发工作也会有所帮助。我相信有很多可行的方法，但是任何高效的战略团队都必须与高效的应用程序开发团队进行通力合作。

## 5. 战略设计者必须谦逊和追求简单

精炼和极简主义对于任何一个优秀的设计来说都是非常关键的，但是，极简主义对战略性设计来说甚至更加重要。即使是很小的设计失误都会给将来开发带来可怕的隐患。独立的架构团队必须更加小心，因为他们可能会更难察觉到摆在应用程序开发团队面前的障碍。同时，如果架构师对于自己的基本职责过于热衷，还可能会迷失方向。我曾多次见过这种情况，我自己也曾经犯过这样错误——那就是过于追求完美，好主意一个接着一个，结果太过讲究的架构反而对开发造成了负面影响。

我们必须自我约束来避免这种情况。凡是不能明显提高设计清晰性的事物都应该从组织原则和核心模型中剔除出去。事实上，几乎所有事物都会在某种程度上互相制约，所以设计中的每个元素都必须物有所值。我们应该谦逊地意识到，自己认为是最好的想法有可能会给别人带来麻烦。

## 6. 开发人员要能文能武，对象要职责专一

一个好的对象设计的关键是要给每个对象赋予一个明确而专一的职责，并且把对象之间的相互依赖减至最少。在软件中，我们要力图使不同团队(的模块)之间的交互尽可能地条理清晰。而在一个优秀的项目中，有很多人都会涉及到别人的工作——开发人员设计框架，架构师编写应用代码，所有人都能相互交谈。对开发人员能力的要求实际上



是多方面的。所以，应该让开发人员能文能武，而让对象职责专一。

前面我已经描述了战略性设计和其他类型设计的区别，从而使得我们能理解涉及到的任务。这里我必须指出，存在这样两种设计行为并不意味着一定要存在两种不同的人。在深层模型基础上创建一个柔性设计是一种高级的设计行为，但是，细节设计也相当重要，它必须由那些从事编码工作的人来完成。战略性设计源于应用设计，然而它必须知道系统行为的全貌才能进行，这可能会跨越多个团队。人们喜欢去寻找划分任务的方法，以便设计专家无需懂业务，领域专家也不需要懂技术。个人能够学习的东西是有限的，但是过于专业化又会脱离了领域驱动设计的思想。

### 17.5.1 技术框架同样如此

技术框架为我们提供了一个基础设施层，它使得应用无需去实现基本服务，并帮助我们把领域从其他概念中独立出来，从而能够极大地加快应用(包括领域层)的开发。但是，架构也会带来一些风险——它可能会限制领域模型在实现上的表达能力，以及在适应改变时的灵活性。即使框架结构的设计者并没有特意涉足到领域层或者应用层，这种风险仍然是存在的。

我们上面所说的针对战略性设计的原则，对于技术上的架构同样适用。渐进顺序、极简主义，以及应用程序开发团队的参与——这些原则都能够使服务和规则不断得到精化，使之能够真正地帮助应用程序开发而不造成障碍。如果架构师不遵照这种方式进行设计的话，他们要么就会抑制应用程序开发的创造力，要么就会发现开发人员为了解决实际问题，不得不绕开他们的结构来进行应用的开发。

下面这种态度肯定会使框架变成一堆废物。

#### 不要以为应用程序开发人员都是傻瓜

在团队分工时，如果认为一些开发人员不够聪明、无法胜任设计工作，那么这样的团队很可能会失败——因为他们低估了应用程序开发的难度。如果这些人不够聪明，那么就不应该让他们去开发软件；如果他们的确很聪明，那么企图用“傻瓜式”的工具来为他们提供方便只会造成障碍，因为那根本不是他们想要的工具。

这种态度也会损害团队之间的关系。我就曾经碰到过这样的情况——我所在的架构团队傲慢自大，结果每次交谈时我都得向应用程序开发人员道歉，为我的同事感到难堪(恐怕我永远无法改变这样的团队)。

请注意，对不相关的技术细节进行封装是必要的，这与我所反对的“傻瓜式工具”完全不同。框架可以为开发人员提供强大的抽象和工具，使他们从繁重的劳动中解放出来。这二者的差别很难用一种通用的方法描述出来；但是，框架设计者对工具(或者框架、



组件)使用者的期望(态度)就可以使我们看出区别所在。如果设计者看起来对框架的使用者非常尊敬,那么说明他们可能还没有误入歧途。

### 17.5.2 提防总体规划

一群由 Christopher Alexander 领导的建筑师提倡“在架构和城市规划领域中要逐步增长”。他们非常好地解释了总体规划失败的原因。

如果没有一个规划过程,俄勒冈州大学想拥有跟剑桥大学几乎同样庞大和谐的校园建设秩序就永远没有机会了。

总体规划是解决这种困难的传统方法。总体规划企图制定足够的指导方针来提供整个环境的一致性,同时又允许每幢建筑物具有不同的风格,并且留下开阔的空间来适应局部的需求。

……将来这个大学的各个部分将构成的一个和谐的整体,因为它们都是按照设计的规划来建立的。

……事实上总体规划失败了。因为它们创造的是一种极权的秩序,而不是有机的秩序。它们过于僵化,很难适应在社区生活中不可避免将要出现的自然变化和其他不可预见的变化。一旦发生改变……总体规划就过时了,并将不再使用。即使是遵循这个总体规划的地区……它们对于建筑物之间的连接、社区的规模、功能的平衡等也没有具体的规定,无法使所有局部建筑的风格和设计与环境很好地融为一体。

……(总体规划)这种方法很像是在一个小孩的填色本上填充色彩……这种过程最多也只能得到填色本中的老套图案。

因此,想要维持一种有机秩序(organic order),总体规划既过于精确,又不够精确。它在总体上过于精确:但在细节上又不够精确。

……总体规划疏远了它的用户,由于大多数重要的决定已经形成,社区成员无法对社区将来的建设产生什么影响。

——译自 *The Oregon Experiment*, 16~28 (Alexander et al. 1975)

Alexander 和他的同事们提倡为所有社区成员拟订一组原则,并应用到逐步发展的每个行动中,这样就产生了“有机秩序”,并且能很好地适应环境。

# 尾 声

能够工作在最前沿的项目，运用有意思的思想、使用好的工具来实验，应该是非常令人满意的，但是如果软件得不到广泛应用，对我而言那将是一无所获。事实上，真正的成功测试是看软件是否能够稳定地运行一段时间。这些年来，我一直借鉴曾经参与过的一些项目的开发经验。

我将在这里讨论其中的 5 个项目，每个项目都是在努力尝试领域驱动设计，虽然它们采用的设计并没有系统化，也不能称之为领域驱动设计。但所有的这些项目都提交了软件：一些项目设法开发一种模型驱动设计，但其中一个却没有这样做；一些应用程序在很多年内会不断被更改，但其中一个停滞不前，另一个也中途夭折。

第 1 章中描述的 PCB 设计软件的测试版本在业内引起轰动。遗憾的是，启动该项目的公司由于市场经营不善停止了该项目的开发。但现在还有少数保留了测试程序备份的 PCB 工程师在使用这个软件。就像任何缺乏支持的软件一样，它会被继续使用下去，直到该软件中集成的某个程序发生了重大的改变为止。

我写了“突破”以后的 3 年中，第 9 章所提到的借贷软件已经得到了茁壮地成长，并以同样的轨迹发展着。目前，从该项目中已经产生了一个独立的公司。在进行重新组织时，从一开始就领导整个项目的项目经理被辞退了，一些核心的开发人员也同他一道离去。新的团队有一些稍微不同的设计理念，他们并不完全是采用对象建模这种方式进行开发。但仍保留了具有复杂行为的领域层，并且开发团队仍然重视领域知识。7 年以后，软件继续被改进并且增加不少新的特性。该应用成为了这个领域的领头羊，为不断增长的客户机构进行服务，也成了公司最大的经济来源。



一片新种植的橄榄树林

当领域驱动的方法被普遍使用时，许多项目中的相关软件将会被迅速、高效地创建出来，项目可能会变得更加通用，它们也许不能被完全利用，更不要说提高到同时具有较早精炼出来的深层模型能力。我期望更多收获，但事实上这些成功多年来为用户提供了许多有用的价值。



7年后的橄榄树林

在一个项目中，我和一名开发人员合作，为客户开发一个可以作为其产品核心的应用程序。这些特征相当复杂，并以一种难以理解的方式结合在一起。我十分喜欢这个工作。我们使用一个抽象核心实现了一个柔性设计。当软件交接以后，每个人负责的任务也就结束了。由于情况变得如此突然，我预计支持可合并元素的设计特性可能会把人搞糊涂，或者可能会被更典型的事例逻辑所代替。这在一开始并不会发生。在我们提交软件时，软件模块包括一组全面的测试套件和一份精炼文档，新的团队可以利用这个文档



来指导他们开发。随着对问题的深入研究，他们会被该设计提供的可能想法而感到兴奋。当我在一年以后听到他们的评论，我意识到通用语言已经传遍了其他团队，并将继续发展下去。

又过了一年，我听到了一个不一样的故事：团队碰到了新的问题，开发人员在原有的设计基础上找不到任何方法来完成任务。他们被迫改变设计已经成为不争的事实。通过我的仔细研究，我发现用我们的模型可以不太灵活地解决这些问题。在这种情况下，通常可能需要突破到一个更深的模型，尤其像这个例子一样，当开发人员积累了该领域的丰富知识和经验的时候，更容易获得一个更深的模型。实际上，他们已经获得了急需的新认识，并在这些认识的基础上完成了模型的转换和设计。

他们谨慎、委婉地告诉我这件事情。我猜想，他们预计我会因为他们把我所做的那么多的工作丢弃掉而感到失望，事实上我并没有对我的设计被丢弃而感到伤感。一个成功的设计是不能固步自封的。一个人们所依赖的系统，如果把它封装起来，那么它将永远成为一个不可触摸的遗产。深度模型提供了可以产生新认识的洞察力，而柔性设计可以很容易适应正在发生的变化。他们提出的模型更加深刻，能够更好地和用户的需求相吻合；他们的设计解决了实际的问题。改变是软件的本质，并且这个程序在开发它的团队手中得以继续发展。

贯穿全书的运输系统例子，是以一个较大的国际集装箱运输公司开发的项目为基础的。在早期，项目的领导打算采用领域驱动方法，但是他们从来没有提供能够支持这种方法的开发文化。几个团队虽然都有对象开发的经验，但是他们的设计技术水平参差不齐，在着手创建模块时，只是通过团队领导的非正式合作和一个受客户重视的架构团队来自由协调。我们确实开发了一个有相当深度的核心领域模型，并且这里有一个可利用的通用语言。

公司文化严重抵触重复开发，而我们却迫不及待地推出了一个内部试用版本。因此，在后期暴露了不少问题，他们要冒更大的风险和花费更大的代价去解决它。模型驱动设计的一个本质特征是反馈信息，包括从实现中出现的问题到模型的更改，在某个时候，我们发现了模型中导致数据库性能出问题的地方。但到了那个时候，我们就会因为涉入太深而难以改变基本模型。相反，对代码所作的改变会使代码更加有效，然而代码和模型的联系却不紧密。初始版本也暴露了在基础结构上的缩性限制，这种限制增加了管理的难度。我们会采纳专家的意见来解决基础设施的问题，并且重新开发项目。但是在实现和领域模型之间的回路永远不会闭合。

几个团队递交的软件都是具有综合性能和表现力的模型。而另外的团队尽管使用了通用语言，但是提交的软件却将模型简化成了数据结构。因为不同团队开发的结果之间



是没有任何必然联系的，所以上下文图也许能够在很大程度上帮助我们。在通用语言中包含的核心模型确实能够帮助团队最终统一到一个系统上来。

虽然应用范围缩小了，但是该项目替代了一些老系统。尽管大多数的设计不是十分灵活，但整个项目通过共享一组概念结合在一起。它没有跟上发展的主要原因在于它自己。数年以后，它仍然会被用来服务于全天候的国际运输业务。虽然该项目会慢慢受到更加优秀的团队的影响，但是即使它在公司里应用得最广泛，最终还是会被淘汰掉的。因为该项目的文化从来没有真正吸收过模型驱动设计的思想。新的开发人员遵照老传统进行开发，所以现在在不同平台上进行的新开发只会间接受到我们所做工作的影响。

在一些情况下，像运输公司一开始制定的那些雄心勃勃的目标是不可信的。最好是实现一个我们知道如何交付的小应用。始终用最少的设计通用部分来完成简单的任务。这种保守的方法有它的好处，并且同时也考虑到了简化作用范围和快速响应的项目要求。但集成的模型驱动系统提供的价值是这些拼凑的应用所不能达到的。还有第3种方法，那就是依靠深度模型和柔性设计，领域驱动设计允许具有丰富功能的大系统逐步增长。

我将以 Evant 公司的故事来结束，这是个开发库存管理软件的公司，我曾经在这个公司担当过辅助支持的角色，并且为当时已经很强大的设计文化作出了一点贡献。其他人已经把这个项目作为极限编程的典范来编写，但是通常并没有强调该项目是完全领域驱动的。不断深化的模型在不断适应变化的柔性设计中得到提炼和表示。这个项目一直保持着兴旺，直到 2001 年 dot com 的衰败。随后由于投资资金的匮乏，公司收缩了，软件的开发大部分也都暂停了，看起来公司的末日即将来临。但是在 2002 年夏天，Evant 公司赢得了一个排名全球前十大的零售商的青睐。这个潜在的顾客喜欢该产品，但是需要改变设计，允许应用按比例增加一种大库存计划的操作，这是 Evant 公司的最后机会。

虽然减少到只有 4 个开发人员，但是这个团队仍然具有满足这个开发所要求的资本。他们的技术十分熟练，对领域的知识也十分熟悉，而且其中一位是大比例结构方面的专家。他们有一个十分有效的开发文化和一个柔性设计的代码库。那个夏天，这 4 个开发人员完成了巨大的开发工作量，使得系统能够同时处理数以十亿计的计划商品和数百个用户。由于出色地完成了这些性能，Evant 公司赢得了这个超级大客户不久之后，Evant 被另外一家公司收购，收购 Evant 的公司想利用它们的软件以及它们显示出来的能力来适应不断变化着的新需求。

领域驱动设计文化(也叫极限编程文化)在转变中幸存下来并且获得了新生。如今，模型和设计继续发展，比起两年以前我参与这个项目的时候有了更丰富的内容和更大的适应性。而且 Evant 团队的成员并没有被购并公司所同化，他们看起来正在鼓励公司的其他团队服从他们的领导，故事还没有完。



没有任何一个项目会把本书介绍的所有技术都用上。即便如此，从几个方面都可以辨认出任何一个采用了领域驱动设计的项目。首要的是要理解目标领域并把理解结合到软件中，其他工作的进行都以此为前提。团队成员不仅要有意识地使用项目上的语言，而且还要对其进行精炼。由于他们在不断地学习更多的领域知识，所以很难对现有的领域模型感到满意。他们把持续精炼视为一种机会，而把不合适的模型视为一种危险。由于作为产品去开发能够准确反映领域模型的软件不是一件很容易的事情，为此他们认真地运用设计技巧。不管碰到多少障碍，失败过多少次，他们仍然坚守自己的原则，振作精神，继续前进。

## 展望

气候、生态系统和生物科学过去常被认为是没有规律的，是和物理、化学形成鲜明对比的“软”领域。然而最近，人们已经认识到在这种“杂乱”的表象下实际上提出了一个有着深远意义的挑战——发现和理解这些非常复杂现象的秩序。具有“复杂性”的领域现在是一些前沿科学。纯粹技术上的任务在有才能的软件工程师看来通常才是最有趣和最富有挑战的，领域驱动设计为软件工程师敞开了一片同样富于挑战的新领域。商业软件并非一定是简单地把各个模块拼凑起来。而将复杂的领域转变成一个可理解的软件设计，对技术很棒的人来说确实是一个令人激动的挑战。

我们离由外行创造复杂软件的时代还差得很远。具有基本开发技术的程序员大军可以开发出某些软件，但是绝对不能开发出可以拯救公司于危难时刻的软件。工具开发人员需要的是一心想着如何扩展那些有才能的软件开发人员的开发能力和生产能力；需要更加有力的方法来探索领域模型并用可运行的软件把它表现出来。我期望能使用依据这种目的而设计出来的新工具和技术进行实验。

虽然改良过的工具十分有价值，但我们不能对此过于关注，而忽略一个事实洞察力，即创造一个好的软件是一种学习和思考的行为。建模需要想象力和自我训练的能力。能够帮助我们思考或避免我们分心的工具是好工具，而所谓能够自动把思想转换成软件产品的努力是天真的、不可能实现的。

与现在大多数项目所完成的系统相比，利用已有的工具和技术，我们能够创建出更加有价值的系统。我们能够写出一个易于使用的优秀软件，当软件扩展时不仅不会形成障碍，反而为我们创造新机会，并且为它的使用者提供了更多的价值。