第15章

精　炼

$$\nabla \cdot \mathrm{D} = \rho$$

$$\nabla \cdot \mathrm{B} = \mathrm{O}$$

$$\nabla \times \mathrm{E} = -\frac{\partial \mathrm{B}}{\partial t}$$

$$\nabla \times \mathrm{H} = \mathrm{J} + \frac{\partial \mathrm{D}}{\partial t}$$

这4个方程式，它们定义的术语，连同它们所依赖的数学基础，表达了19世纪经典电磁学的全部内容。

《电磁通论》——詹姆斯·克拉克·麦克斯韦，1873

如何把注意力集中于中心问题之上，而避免陷入到繁复的细枝末节中去呢？分层架构能将领域概念与计算机系统运行所需的技术逻辑分离开来，但是在一个庞大的系统里，即使是隔离的领域也可能复杂到难于管理。

精炼(distillation)是指将混合物中的不同成分分离出来，从中以某种形式提取出更有用、更珍贵的精华的过程。模型就是对知识的一种精炼。每一次重构都使我们获得更深入的了解，并逐渐将一些关键的领域知识和需要优先考虑的方面抽象出来。然后，我们就可以后退一步来从战略上进行考察了。本章将讨论如何将模型中大的方面区分出来，并将模型作为一个整体进行精炼。

就像很多化学精炼一样，模型精炼(例如通用子域(Generic Subdomain)和内聚机制(Coherent Mechanism))也可以分离出更为有用的副产品。但是，模型精炼的目的是希望

提取出一些特别有价值的部分，正是这些部分使我们的软件产品具有了差异性，使我们值得为之付出努力——这就是“核心领域(Core Domain)”。

对领域模型进行战略性精炼，可以：

- 帮助团队成员把握系统的整体设计，以及各个部分如何配合工作
- 得到一个便于交流的核心模型，其规模更易于控制，并可以加入通用语言
- 指导重构
- 使我们能集中精力处理模型中最有价值的部分
- 为外包、使用现成组件以及分配决策提供指导

本章将给出一套系统性的对核心领域进行战略性精炼的方法，这套方法说明了如何在团队中有效地共享视角，并且提供了在这个过程中使用的语言。

图 15-1 所示为战略性精练导航图。

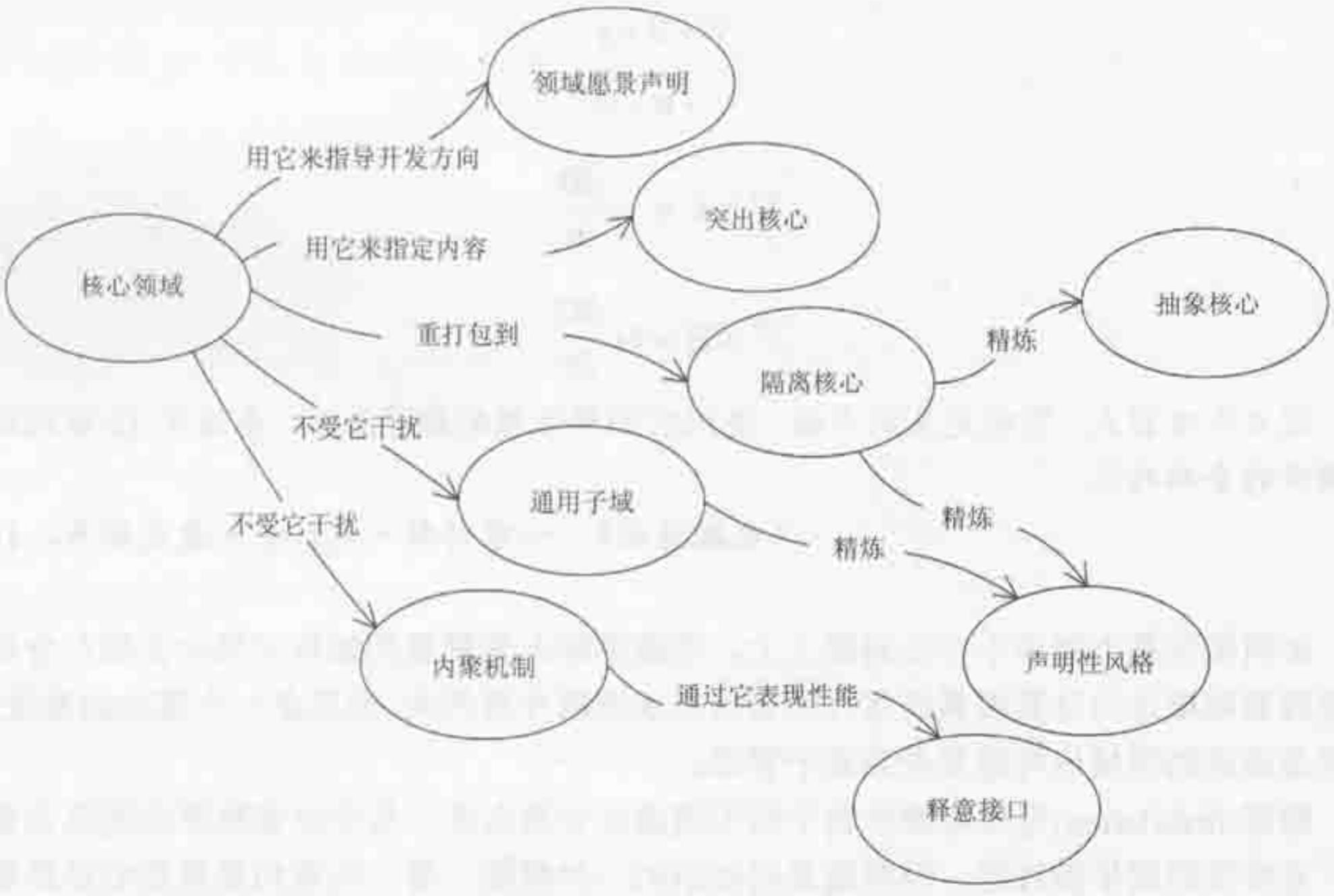


图 15-1　战略性精炼导航图

15.1　核心领域

园丁剪枝的目的是为了为主干的生长扫清障碍；同样，我们也将应用一套技术来排

除模型中的干扰，并把精力集中在模型中最重要的部分。

在设计一个大型系统时，我们会遇到很多有用的组件，它们都非常复杂，都是软件成功所不可或缺的，这样的组件实在太多了，以至于领域模型的精髓部分(它们才是真正的企业资产)反而会变得不那么明显甚至被忽视掉。

一个难以理解的系统是很难更改的，而且改变的效果难以预料。一旦超出开发人员所熟悉的领域，他们就会如坠云端(对于开发团队中新加入的成员来说这种情况尤其明显。但是，如果代码表达不清或者结构模糊，即使是团队中的老成员，在这种情况下也会觉得很麻烦)。这样就会使得开发人员只能胜任特定的任务。一旦开发人员把自己的工作限制在特定的模块，他们之间的知识交流就会进一步减少。这种工作的隔离使得我们难以平稳地将系统集成起来，也无法在开发人员之间灵活地分配工作。同时，由于开发人员不知道别人早已在其他地方实现了相同的功能，因此系统中会出现大量重复，而这又进一步增加了系统的复杂性。

对于那些难以理解的模型来说，上述的问题是必然会发生的。但是还有一种情况同样危险——那就是失去了对领域的整体认识。

客观地说，我们不可能将设计中的所有部分都一视同仁地进行精炼。我们必须分清轻重缓急。为了使领域模型真正成为资产，模型中关键的核心部分必须足够灵活和充分平衡来创建应用程序的功能。但可惜的是，技巧高超的开发人员往往倾向于使用技术基础结构来解决问题，或者是去解决那些无需专门的领域知识就能理解的、定义明确的领域问题。

无需专门的领域知识就能理解的问题似乎更容易引起计算机专家们的兴趣，他们认为，只有解决这些问题才能积累自己的专业技巧，同时也为自己的简历增光添彩。结果，核心模型往往是由一些不太熟练的开发人员装配完成的。他们和 DBA 一起创建一个数据方案，然后一个功能一个功能地编程实现，根本没有用到模型中的任何概念。但是，

正是那些包含了专门领域知识的核心模型，才使我们的系统真正具有了差异性，才使之成为一种真正的企业资产。

如果软件的核心模型设计或实现得非常拙劣，那么不管使用的基础结构多么先进，也不管提供的支持特性多么丰富，用户都不会觉得软件的功能有多么强大。产生这种严重问题的根源在于，项目对整体设计以及软件各个部分之间的相对关系缺乏清晰的认识。

我曾经参与过的一个相当成功的项目，这个项目在开始的时候就出现过上述的问题。项目的目标是开发一个非常复杂的银团贷款系统。大多数技术天才和技术高手都对数据库映射层和消息接口津津乐道，而业务模型却被交给一些刚刚涉足面向对象技术的新手们打理。

惟一例外的是，我们让一个经验丰富的面向对象开发人员来负责处理一个领域问题。他设计了一种为所有持久领域对象添加注释的方法。这些注释能够被很好地组织起来，使借贷商们可以看到自己或别人以前记录下来的决策思路。他还设计了一个友好的用户界面，用户可以直观地使用这个注释模型所提供的各种灵活特性。

这些特性不仅有用而且设计巧妙，所以它们成为了产品的一部分。

遗憾的是，这些特性只是一些外围的东西。这个开发天才建模了一种有趣的、通用的注释方法，清晰地实现了这些功能，并最终交付给了用户使用。但是同时，那些差劲的开发人员却把“借贷”模块这个关键任务的开发弄得一塌糊涂，整个项目差点因此而彻底失败。

在计划时，我们必须把资源分配到模型和设计最为关键的地方来。为此，我们必须明确指出哪些地方才是关键点，并保证所有开发人员在计划和开发阶段都能够理解它们。

核心领域是由模型中那些具有差异性的、在系统的目标中处于中心地位的部分构成的。核心领域是系统中最有价值的部分。

因此：

浓缩模型，找出核心领域，并且提供一种方法使我们能很容易地把它从众多的支持模型和代码中区分开来。将最有价值、最体现专门知识的概念凸现出来，让核心变小。

让最好的开发人员来处理核心领域，并根据需要调整人员的配备。尽力寻找核心的深层模型，并开发出一个能充分满足系统愿景的柔性设计。对任何其他部分的投入都必须经过考虑，看它们是否能为精炼出来的核心提供支持。

精炼核心领域不是一件容易的事情，但是精炼过程中需要我们作出的决策并不复杂。我们要努力使核心模型与众不同，同时又让设计的其余部分尽可能通用灵活。如果您想让设计中的某些方面秘而不宣，并凭借它们来取得竞争优势，那么它们就是核心领域——您无需浪费精力去隐藏其余部分。此外，无论何时，如果由于时间的限制，我们必须

在两个合适的重构之间进行选择的话，那么对核心领域影响最大的就是我们的首要选择。

本章介绍的模式将使核心领域更易于理解、使用和改变。

15.1.1 选择核心

我们考虑一下那些在模型中代表业务领域以及解决业务问题的部分。

核心领域的选择将取决于您的观点。例如，很多应用程序需要一个通用的货币模型，能够提供各种流通货币的汇率和换算。另一方面，需要支持货币交易的应用程序则可能需要一种更为精细的货币模型，并把它当作核心的一部分。但是，即使是在这种系统中，货币模型可能还是会有某个部分是很通用的。随着经验的增加和对领域理解的不断深入，我们可以继续对模型进行精炼，进一步将通用的货币概念分离出来，而只把模型中那些具有专门特征的方面留在核心领域中。

在一个运输应用系统中，表达核心的模型可能包括如何整理需要运送的货物；如何在转运集装箱时转移责任；以及如何安排一个特定的集装箱，使之通过不同的运输方式到达目的地。在投资银行业务应用中，它的核心可能包括在代理人和投资者中资产联合的模型。

一个应用程序中的核心领域很可能是另一个应用程序的通用支持组件。尽管如此，我们还是可以为一个项目(通常涉及整个公司)定义一个一致的核心。正如其他部分的设计一样，核心领域也是通过反复迭代来确定的。一些关系在刚开始时可能显得并不重要，而开始看起来似乎非常重要的对象后来却会发现原来只是充当支持的角色。

在后面的章节，尤其是通用子域的讨论中，我们将给出更多有关这些决策的指导思路。

15.1.2 谁来负责精炼工作

项目团队中大多数技术精湛的开发人员几乎都没有太多的领域知识，这限制了他们的工作范围，因而只能分配他们去开发支持组件。由于领域知识的缺乏而使熟练人员无法从事领域逻辑的构建工作，这无疑是一种怪圈。

要想打破这个怪圈，关键是要组织一个由优秀开发人员组成的团队，这些开发人员将对系统的长期目标负责，并且乐于在一个或多个资深领域专家的帮助下学习和积累相关的领域知识。如果严肃地看待的话，领域设计是一项有趣的、充满技术挑战的工作；我们要寻找的就是持有这种观点的开发人员。

短期地雇用一个外援专家，让他来帮助我们创建核心领域的具体细节——这种做法通常是不可行的，因为团队需要积累领域知识，而雇用临时成员将使团队变成一只装不

住水的漏桶。另一方面，如果有一个专家来为团队提供指导和顾问，那将是大有裨益的，他能帮助团队学习领域设计技巧，促进团队成员掌握他们可能尚未掌握的复杂原则来进行设计。

同理，购买核心领域就更加不大可能了。人们已经在建立专门的行业模型框架上做了一些工作，其中著名的例子就是，由半导体工业协会 SEMATECH 提出的实现半导体制造自动化的 CIM 框架。IBM 的 San Francisco 框架也在商业领域获得广泛应用。但是，尽管这种想法非常诱人，到目前为止其应用的结果仍然不太理想，也许只有它们的公布语言还算成功，因为它们为数据交换提供了方便(参见第 14 章)。Fayad 和 Johnson 编著的 *Domin-Specific Application Frameworks*(2000)一书对这种技术进行了概述。随着该领域的发展，将来会出现更多可行的框架。

虽然我们可以使用这些框架，但是必须注意一个更根本性的问题：定制软件中最有价值的部分来自于其对于核心领域的完全控制。设计良好的框架提供了高层次的抽象，我们可以对这些框架进行特殊化处理来解决特定的问题。框架可以帮我们缩短开发通用部件所需的时间，而把精力集中到核心部件的开发上来。但是，如果框架带来了更多的限制而不是帮助，那么可能有以下 3 种情况。

- 您正在丧失一项重要的软件资产。此时应该把框架从核心领域中排除出去。
- 框架的应用范围并不如您想的那么重要。此时应该重新确定核心领域的边界，使之成为模型中真正具有差异性的部分。
- 您的核心领域没有什么特殊需求。此时应该考虑采用一种低风险的解决方案，例如购买现成的软件，然后将其集成到您的系统中去。

不管采用什么方式，要开发一个独一无二的软件，我们必须依赖于一个稳定的团队，必须不断积累专门的领域知识，并将其融入到一个优秀的模型中去。这是没有任何捷径可走的。

15.2 精炼的逐步升级

本章余下部分将介绍各种精炼技术。应用这些技术并没有严格的顺序要求，但不同的技术对设计产生的影响程度会有所不同。

我们可以用一个简单的“领域愿景声明(Domain Vision Statement)”来描述领域的基本概念及其价值，这也是一种所需投入最少的技术。“突出核心(Highlighted Core)”能够增进交流并指导我们做出决策，同时它只需对设计作少量修改(甚至无需任何修改)。

更为激进一点的是进行重构和重新打包，它们显式地对通用子域进行划分，使其能

够被分别处理。内聚机制可以使我们得到一个通用、易懂、柔性的设计。只有消除了这些旁枝细节，我们才能解脱出来从而集中于核心的开发。

对“隔离核心(Segregated Core)”进行重新打包，能使核心立刻变得清晰起来——即使是在代码中它也清晰可见。这还能为将来在核心模型上的工作提供方便。

最为激进的技术当数“抽象核心(Abstract Core)”了，它以一种抽象的形式来表达基本的概念和关系，并且需要对模型进行大面积的重组和重构。

这些技术需要我们承担越来越重的责任。但是，刀是越磨越快的，对领域模型的不断精炼将使我们获得珍贵的软件资产，使项目进行得更快、更敏捷、更精确。

一开始，我们可以去除模型中最一般化的部分。将通用子域与核心领域进行对比可以使我们清楚地理解二者的含义。

15.3　通用子域

模型中的有些部分不能捕捉和体现专门知识，同时又会增加模型的复杂性。任何外围的东西都会使核心领域变得难以辨别和理解，使模型中充斥着众所周知的一般原则，或者与主要目标无关的辅助功能。当然，虽然这些都是一些通用元素，但它们对于系统的功能来说都是不可或缺的。

模型中有些部分的必要性是不言自明的。这些部分肯定应该属于领域模型，但是，它们所抽象出来的概念也是其他千千万万的商业应用同样需要的概念。例如，不同的行业(如运输业、银行业或者制造业)都需要某种形式的组织结构图。另外一个例子是，许多应用都需要跟踪应收款、费用分类帐和其他财务信息，这些信息都可以用一个通用的会计模型来进行处理。

处理领域的外围事情通常要花费大量的精力。我本人就见到过这种事情，有两个项目组都要重新设计带时区功能的日期和时间组件，开发这样的组件花了他们最好的开发人员数周的时间。虽然这样的组件是必须要做的，但它们并不是系统的核心。

即使您觉得一个通用模型元素非常重要，但还是应该将系统中附加值最高、最具有差异性的部分从整个领域模型中凸现出来，并对那些部分进行精心组织，使之尽可能地强大。如果核心与所有相关因素混合在一起时，模型的核心就很难突出了。

因此：

将那些与项目目标无关的内聚子域标识出来，然后把它们分离为通用模型并放到单独的模块中去。不要让这些子域包括任何专门领域知识。

分离出子域之后，它们的开发优先级别应该设置得比核心领域的优先级别低一些，

并避免让核心开发人员来负责这些任务(因为他们不会从这些任务中获得任何领域知识)。对于这些通用子域，也可以考虑采用现成的解决方案或者公开的模型。

在开发这些模块时，可以有以下几种选择。

选择 1：现成的解决方案

有时可以购买一个已实现的方案或者使用开放源代码。

优点

- 需要开发的代码较少。
- 维护责任由外部承担。
- 代码可能更成熟，使用范围广，因此比自己开发更加安全和完善。

缺点

- 在使用前，必须花功夫对它进行评价和了解。
- 质量控制问题。不能保证它的正确性和稳定性。
- 它可能过于复杂和通用，远远超出了您所需要的功能，这样集成的工作量可能比自己实现的要大。
- 外部代码与项目的集成往往不那么容易，二者的限界上下文可能会截然不同。即使限界上下文的差别不大，也很难顺利地从其他的模块中引用实体。
- 可能会引入对操作平台、编辑器版本等的依赖。

现成的子域解决方案值得我们去研究，不过它们往往也会带来不小的麻烦。我曾经见过一些成功的应用现成的子域解决方案的实例。例如，有的项目成功地使用了商业化的外部工作流系统，利用它提供的 API 钩子实现了非常精巧的工作流功能。我还见过有人成功地把错误日志模块非常巧妙地集成到了应用程序中。有时候，通用子域的解决方案是以框架的形式来提供的，它实现了一个非常抽象的模型，您可以对它进行定制，并集成到系统中来。子域组件越通用，其子域模型越精炼，使用起来就越有用。

选择 2：公开的设计或者模型

优点

- 比自己开发的模型更加成熟，而且反映了很多人的见识。
- 直接提供了高质量的文档。

缺点

- 可能不太符合您的需要，或者设计超过了您的需要

第 15 章
精　炼

二十世纪五六十年代的著名喜剧作曲家 Tom Lehrer 曾说过，在数学上获取成功的秘密就是，“抄袭！剽窃！不要放过任何人所做的工作……但务必要记住把它称之为借鉴、研究。”在领域建模中，这是一个好的建议，尤其是在处理通用领域问题的时候。

如果通用领域已经有了一个被广泛使用的模型能够满足需要时，运用这个模型就最有效，例如 *Analysis Patterns*(Fowler 1996)中的模型(参见第 11 章)。

如果通用领域已经有了一个高度形式化的、严谨的模型，那么使用它。会计学和物理学就是两个很好的例子。会计学和物理学不仅完备、简洁，而且到处都能被人们所理解，这也减少了现在和将来需要培训的负担(参见第 10 章)。

如果能确定一个简化的集合，它能够满足您的需要并保证内部统一，就不要非得去实现公开模型的所有部分。如果有一个可用的模型包含了丰富的经验和规范的文档，当然有正式的模型更好，那么再去重新进行设计就没有什么意义了。

选择 3：外包实现

优点

- 由于核心领域需要收集大量的知识，外包可以使核心团队腾出时间来处理核心领域。
- 无需不断扩大开发团队以完成更多的开发任务，同时又不会分散核心领域的信息。
- 强制我们使用面向接口的设计，并且有助于保持子域的通用性，因为我们需要把规范传递给外包人员。

缺点

- 因为需要对接口、编码标准和其他重要因素进行交流，所以仍然需要占用核心开发团队的时间。
- 为了使项目能完全掌握外包实现的代码，需要花大量精力来理解那些代码(当然，这比理解专门子域所需的精力应该要少一些，因为通用模型不需要了解特殊的背景知识)。
- 代码质量可能不同。是好是坏将取决于两个开发团队的水平。

自动化测试在外包中占有非常重要的地位。外包实现人员必须为他们提交的代码提供单元测试。这里有一种相当强大的方法，它可以帮助我们保证质量，阐明规范，使代码能够顺利集成——那就是为外包组件指定(甚至编写)自动化的验收测试。同时，“外包实现”可以与“公开的设计或模型”很好地结合起来使用。

选择 4：内部实现

优点

- 易于集成。
- 所得即所需，没有其他附加的东西。
- 可以指定临时分包人员。

缺点

- 维护和训练需要开销。
- 很容易低估开发这种软件包的时间和成本。

当然，这种方式也可以与“公开的设计和模型”很好地结合起来。

对于通用子域，我们可以设法利用外部的设计专家，因为他们不需要深入了解特殊的核心领域(而且他们也没有多少机会接触到核心领域)。机密性不是问题，因为在通用子域的模块中很少包括机密信息或者业务规则。有些人员不需要对领域有深入的了解，而通用子域正好减少了对他们进行培训的开销。

随着时间的推移，我相信我们对于核心模型的界定将会更加狭窄，而越来越多的通用模型将被实现为框架(至少是作为公开的模型或分析模式供我们使用)。目前，大部分的通用模型仍然需要我们自己来开发，但是把它们从核心领域中划分出来是很有价值的。

示例：两个关于时区组件的故事

让项目组里最好的开发人员，花费数周的时间，来解决在不同时区内保存和转换时间的问题——这种情况我见过两次。虽然我总是怀疑这样做的必要性，但是有时候它又是必须的，这两个项目几乎形成了鲜明的对照。

第一个项目要设计安排货物运输的行程。为了制定国际运输计划，能够准确地计算时间是非常重要的。由于所有运输计划都是以当地时间为准的，所以不进行时间转换就无法协调好运输过程。

在明确了这个功能需求后，团队继续进行核心领域的开发，并且在早期的迭代过程中使用了一个已有的时间类和一些空数据。随着系统的逐渐成熟，团队明显感到那个时间类已经无法满足需求了，各个国家时间的差异和国际日期变更线使问题变得很复杂。在需求日益明确以后，他们找了一些现成的解决方案，但没有哪个能满足要求。别无他法，他们只好自己创建一个。

由于这项任务需要进行研究和精心设计，团队领导希望指派他们当中最优秀的程序员来负责。但是，这项任务不需要任何运输领域的专门知识，也不会为运输领域提供什

么专门知识，所以他们选择由项目中的一个临时程序员来完成这项任务。

这个程序员并不是从零开始进行设计。他研究了几种现有的时区转换的实现，发现大多数都不能满足应用的需求。最终他发现 BSD Unix 的公共领域解决方案中有一个方案，它使用了精巧的数据库并已经用 C 语言实现了，于是他决定对其进行改编。他通过反向工程找出了其中的逻辑，并为这个数据库编写了一个程序。

虽然出现的问题比想象的要困难得多(例如，导入数据库时的特殊情况)，但代码还是编写完成并集成到核心中，产品最终提交了。

另外一个项目的情况则完全不同。这是一家保险公司正在开发的新的理赔处理系统，他们打算用这套系统来记录各种事件发生的时间(如撞车的时间、雹暴发生的时间等)。这种数据将以当地时间来记录，因此需要有时区的功能。

当我到达时，他们已经指派了一个年轻但很聪明的开发人员来完成这项工作。但是，当时应用的确切需求并没有确定下来，甚至连一次迭代过程都还没有尝试过。这个开发人员很尽职地开始了自己的工作——根据自己头脑中的先验认识来创建时区模型。

由于不知道到底需要哪些功能，所以他只好假定它能灵活地处理任何事情。解决这样的难题超出了这个程序员的能力，于是项目组又指派了另一个资深开发人员为他提供帮助。他们编写了非常复杂的代码，但是由于没有任何明确的应用来使用它们，因此根本不知道代码是否能正确运行。

由于各方面的原因，这个项目搁浅了，那些关于时区计算的代码从未被使用过。但是，即使是涉及到了时区的问题，在这个应用中也只需简单地保存当地时间并附加上当地所在时区可能就足够了，因为时间只是一种参考数据而并非理赔计算的依据——所以根本不需要进行任何转换。就算时区转换也是必需的，由于所有数据都是从北美地区收集来的，所以时区的转换相对而言并不复杂。

由于项目组过于关注时区的问题，结果造成了对核心领域模型的忽视。如果把同样多的精力放到核心领域上，那么他们可能已经设计出了这个应用程序的功能原型，并且可能已经初步得到了一个可以工作的领域模型。更进一步，所有对项目的远期目标负责的开发人员此时都应该已经对保险领域有所涉足，为团队增加了关键的领域知识。

有一件事这两个项目组都做对了——那就是把通用的时区模型从核心领域中明确地分离出来。如果将运输系统或保险系统与各自的时区模型关联起来，那么它们的核心将更难理解(因为时区模型的细节与核心是无关的)，同时也可能会使时区模块更难维护(因为维护人员必须了解核心以及核心与时区的关系)。

表 15-1 对两个项目的策略作了对比。

表 15-1　两个项目的策略比较

运输项目的策略	保险项目的策略
优点 • 从核心中分离了通用模型 • 核心模型比较成熟，所以资源的转移不会阻碍它的发展 • 确切地知道他们需要哪些功能 • 国际时间表提供关键的支持功能 • 由短期雇佣的程序员来负责通用模型的开发任务 缺点 • 分散了核心开发团队中优秀程序员的精力	优点 • 从核心中分离了通用模型 缺点 • 在核心模型尚未完成的情况下，关注其他问题将使小组继续忽视核心模型的开发 • 需求不清，因此只好企图开发一种万能程序——但实际上只要把北美的时区转换功能完成即可 • 指派长期的程序员来开发通用模型，他们本来应该成为领域知识的智囊团

我们的技术人员往往喜欢处理那些定义明确的问题(如时区转换问题)，而且很容易就能找到充分的理由来花时间解决它们。但是，如果严格地按照优先级别来考虑的话，通常核心领域才是应该首先处理的部分。

15.3.1　通用不一定可重用

注意，当我强调一个子域具有通用性时，并没有说它的代码具有可重用性。现成的解决方案是否能满足特定的需求姑且不说，如果您决定自己来实现代码，那么不管是采取外购还是内部开发的方式，都不应该去关注代码的重用性，因为这与精炼的目的背道而驰：您应该尽可能地把精力投入到核心领域问题的处理上来，只在必要的时候才去处理通用子域。

重用并不总是指代码的重用——模型重用(如重用公开的设计或模型)通常是一种更高级的重用。如果我们必须自己创建模型，那么这个模型对于以后相关项目的开发也会具有很高的价值。不过，即使模型中的概念在很多情况下都可以用到，我们也不必把模型开发得百分之百的通用。我们只要将业务中需要的部分建模和实现出来就可以了。

注意，虽然我们要尽量避免为了可重用性而进行设计，但是又必须严格遵循通用概念的思想。将与特定行业相关的元素引入到通用子域中来会造成两个不良的后果两个。第一，它会阻碍今后的开发。尽管您现在只需要子域模型的一小部分，但您的需求是会增加的。在设计中引入不属于模型概念的东西会导致系统很难灵活地进行扩展——除非我们对原有部分进行彻底改造，并且重新设计那些使用它们的模块。

第二，也是更重要的，那就是这些与特定行业相关的概念要么应该属于核心领域，要么应该属于它自己的比通用模型更有价值的专门子域。

15.3.2　项目风险管理

敏捷过程通常要求通过及早处理风险最高的任务来控制风险。特别地，XP 过程要求端到端系统能够立即运行起来。这种原型系统常常被用来验证某种技术体系的可行性。同时，由于通用子域通常比较容易分析，所以我们可以构造一个外围系统来处理那些辅助性的通用子域 ——这种做法似乎很诱人。但是要小心，这种做法可能会与风险管理的目的相悖。

项目面临的风险来自两个方面，在一些项目中存在着较大的技术风险，而另外一些项目中则存在着较大的领域建模风险。端对端系统只是实际系统中最困难的部分的一个雏形，因此它对风险的控制能力是有限的。我们很容易低估领域建模的风险，这种风险可能表现为错误地估计了系统的复杂度，缺乏业务专家的支持，以及开发人员对关键技术的掌握程度参差不齐等。

因此，除非团队具有熟练的开发技能而且对领域非常熟悉，否则不管核心领域多么简单，首次实现原型系统时都应该以核心领域的某个部分为基础。

这个原则也同样适用于任何试图及早处理高风险任务的过程：核心领域具有高风险，因为它经常出乎意料地难以处理，如果没有它，项目就不可能获得成功。

本章介绍的大多数精炼模式展示了如何改变模型和代码来精炼核心领域。然而，下面介绍的这两种模式，领域愿景声明和突出核心，则展示了补充文档能够以极小的投入来增进交流，提高和对模型核心的认识，并能把小组的精力集中到核心的开发上来。

15.4　领域愿景声明

在项目初期，模型通常还不存在，但项目的主要开发需求却早已确定。在开发一段时间以后，我们就需要解释系统的价值究竟在哪里——这并不需要对模型作深入研究。另一方面，领域模型的关键方面可能涉及到多个相互独立的限界上下文，从定义来看，这些模型是没有共同目标的。

许多项目团队为管理目的而编写了“愿景声明”。好的愿景声明能够将系统为企业所带来的价值清楚地罗列出来，有的还提到应该把创建出来的领域模型作为企业的一项战略性资产。但是，在项目启动以后，愿景声明文档往往会被置之高阁，在实际开发过程当中形同虚设，甚至根本就没有被技术人员阅读过。

领域愿景声明是根据愿景声明来编写的，但是它着重描述领域模型的本质以及对企业有何价值。在开发过程的所有阶段之中，管理层和技术人员都可以直接用它来指导资源的分配、建模的选择以及团队成员的培训。如果领域模型要为多个目标服务，那么领域愿景声明还能够说明它将如何满足这些目标的需要。

因此：

用大约一页纸的篇幅简短地描述核心领域及其所带来的价值(即“价值主张(value proposition)”)。忽略那些不能将模型与其他领域模型区分开来的问题。说明领域模型是如何维护和平衡各方利益的。不要泛泛而谈。尽早写出领域愿景声明，并且在获得新的认识后及时进行修改。

在对模型和代码进行精炼的过程当中，领域愿景声明可以作为一种方向标，指导开发团队沿着一个共同的方向前进。我们还可以与非技术团队的成员、管理人员，甚至与客户一起分享领域愿景声明(当然，其中包含商业秘密时例外)。

表15-2和表15-3分别列出了航空公司订票系统和半导体工厂自动化系统的愿景声明。

表 15-2　航空公司订票系统和愿景声明

部分领域愿景声明	虽然重要，但不属于领域愿景声明
航空公司订票系统 该模型能够支持乘客优先级别、订票策略以及灵活的权衡策略。乘客模型应该能够反映出航空公司积极发展与老客户关系的努力。所以，它应该以一种精简的形式来提供乘客的历史记录，他们参加过的特别活动，以及与战略性集团客户的合作关系等。 模型要反应不同用户的不同角色(例如乘客、代理商、管理人员)，以详细地描述客户关系并且为安全框架提供必要的信息。 模型应该支持高效的路线/座位搜索，并与其他已有的航班订票系统集成	**航空公司订票系统** 对于熟练用户来说，UI 应该简单高效，但对于新用户来说，UI 应该便于操作。 系统可以通过 Web 或其他类型的 UI 进行访问，并且可以把数据传递到其他系统。因此，系统将基于XML来设计接口，使用转换层来为 Web 页面提供服务或者转换到其他系统中。 要在客户机上缓存一个生动鲜艳的 logo，以便 logo 能在下次访问时迅速显示出来。 在客户提交预约以后，要在 5 秒钟内反馈可视的确认信息。 安全系统将鉴别用户身份，然后根据规定的用户角色，限制其访问的权限

表15-3　半导体工厂自动化系统的愿景声明

部分领域愿景声明	虽然重要，但不属于领域愿景声明
半导体工厂自动化 该领域模型将描绘原料和设备在晶片加工中的状况，它能够提供必要的审计跟踪功能，支持自动化生产线。 模型不包括在生产过程中需要的人力资源，但是允许通过配方下载来实现选择工序自动化。 工厂状况的描述应该让管理人员容易理解，从而加深他们的了解并提出更好的决策	半导体工厂自动化 该软件可以通过一个 servlet 来支持 Web 访问，但是其架构应该允许采用不同的访问接口。 尽可能使用行业标准技术，避免内部的开发及其造成的维护费用。尽可能使用外部专门技术。开源项目是首选的解决方案(例如 Apache Web 服务器)。 Web 服务将运行在一台专用服务器上。应用程序将运行于一个单独的专用服务器上

虽然领域愿景声明为开发团队指明了一个共同的开发方向，但通常在高层次的声明以及代码或模型的完整细节之间还需要建立一些联系纽带。

15.5　突出核心

领域愿景声明用几项主要条款来确定核心领域，但是具体的核心模型元素则要单独描述。除非团队内部的交流极其高效，否则光凭愿景声明是难以奏效的。

即使团队成员对核心领域的构成有了大致的理解，但不同的人所理解的元素不会完全一样，甚至同一个人的理解也会随着时间的推移而发生变化。我们应该把精力放在设计上，但是分辨和寻找模型的关键部分不仅会分散我们的精力，而且还需要我们对模型有全面的了解。因此，核心领域必须要显而易见。

对代码作重大的结构改变是确定核心领域的理想方式，但是这样做在短期内可能并不现实。实际上，如果开发团队对核心领域还缺乏明确认识的话，那么他们是很难对代码作出这样的重大修改的。

模型的结构改变(如划分通用子域和后面介绍的一些方法)可以通过模块体现出来。但是这种改变太过激进，而且难以直接实现，因此我们不能把它当成表达核心领域的惟一方法。

我们可能需要一种轻量级的解决方案，来作为那些激进技术的补充。例如，我们可能会受到一些限制，而无法从代码上对核心进行划分。现有的代码可能并没有明确地区

分出核心领域，但是我们必须了解核心，以便能达到更好的精炼和更有效的重构。进一步，从更高的层次上来说，通过精选几个图形或文档，我们就能为开发团队提供一些思路和切入点。

不管是使用 UML 建模的项目，还是只需少量外部文档、把代码作为主要模型仓库的项目(如 XP 项目)，都会出现这些问题。极限编程团队可能更精炼一些，使用这些补充技术的机会就更少，也更具有临时性(例如，把图画在所有人都能看得到的墙上)，但是，这些技术可以很好地融入到开发过程中。

模型中的专有部分，连同它的具体实现，只是对模型的一种反映，并不是模型本身的必要部分。任何技术，只要能让人容易地了解核心领域，都可以用来描述核心领域。这里有两种技术，可以作为解决这一类问题的代表。

15.5.1 精炼文档

通常，我会创建一个单独的文档来描述并解释核心领域。在文档中，我们可以给出一个最基本的概念对象的列表；可以用一组与这些对象相关的图表来展示它们之间最关键的关系；可以在某个抽象层次上(或通过实例)描述对象之间的基本交互过程。我们还可以使用 UML 类图或顺序图、领域中特有的非标准图表、措词严谨的解释，或者综合使用这些方法。精炼文档并不是一份完全的设计文档。它只是提供了一种最低限度的要求，用来描绘和解释核心部分，并提出对这些部分细加研究的理由。读者能够从中全面了解到各个部分是如何协作的，并能根据提示找到相关的代码，以便获取详细内容。

因此，一种描述突出核心的形式就是：

写一个非常简短的文档(3～7 页纸)，描述核心领域以及核心元素之间的主要交互过程。

但我们同样会碰到其他文档也有的常见问题。

- 文档可能没人维护。
- 文档可能无人阅读。
- 由于这种文档也是一种信息源，因此它可能会增加复杂性，这与它的目的相悖。

避免出现这些情况的最好方法是文档必须绝对精简。跳过所有普通细节，集中描述中心概念以及它们之间的相互作用，这样才会降低文档的更新速度，因为模型在这种层次上通常比较稳定。

这种文档要能够被团队中的非技术人员理解。它应该作为一种模型的共识，把人人都需要知道的东西描述出来；或者作为一种指南，使团队的每个成员都可以把这个文档

作为起点，开始他们对模型和代码上的研究。

15.5.2　把核心标记出来

我曾参与过一家大型保险公司的一个开发项目，第一天，我拿到了“领域模型”的一份副本，这是一份厚达两百页、花大价钱从一个行业协会中购买来的文档。我花了几天时间对它进行研读，结果脑子里充满了乱七八糟各种各样的类图，它们包括了所有的东西，从详细的保险单组成到非常抽象的人际关系模型。这些模型构造的质量，有的简直就只有中学生的水平，而有的确实特别优秀(一些甚至可以作为商业标准，至少我认为附带的文档达到了这么高的水准)。但是文档有两百页那么多，我该从哪里入手呢?

这个项目组非常崇尚创建抽象的框架，我的前任们的工作主要集中在一个非常抽象的人际关系模型上，它包括了人与人之间的关系、人与物的关系以及人与活动或协议的关系。事实上，他们对这些关系的分析非常透彻，并且他们对模型所作的实验也达到了学术研究项目的质量。但是，我们还是不知道该从哪里下手来处理这个保险业务的应用。

我的第一反应是先找出一个小的核心领域，然后对它进行重构，并逐渐引入我们要解决的其他复杂问题。但是管理部门对这种意见提出了警告。购买的这个文档具有绝对的权威性。它是整个行业的专家共同创造的成果，而且他们支付给协会的价钱无论如何都会比支付给我的高得多，所以他们不太可能会认真地考虑我要进行根本改变的建议。但是我知道我们必须首先对核心领域拥有一个共同的认知，然后才能把所有人的工作重心集中起来。

我并没有对这个模型进行重构，而是仔细地分析了这个文档。有一个业务分析员，他大体上知道很多关于保险业的知识并且特别了解我们要创建的应用需求，在他的帮助下，我从该文档中确定了几个部分，这才是我们真正需要处理的基本的、与众不同的概念。我给出了一个模型的导航图，清楚地显示出系统的核心以及它与辅助功能之间的关系。

从这种观点来看，我们其实已经开始进行了一种新的原型设计工作，并且迅速产生了一个简化的原型系统来论证一些必需的功能。

只需要通过在文档中添加一些页制表符和黄色的轮廓，就可以把核心标记出来了，然后这沓两磅重的纸也就变成了一种商业资产。

这种技术也可以用于对象模型图。喜欢使用 UML 图的开发团队可以用一个 stereotype 来确定核心元素；把代码作为模型惟一仓库的开发团队则可能使用注释，把这些核心元素组织成 Java Doc，或者在他们使用的开发环境中使用某种标记工具。采用什

么特殊的技巧并没有什么关系，只要开发人员能够毫不费力地区分哪些属于核心领域、哪些不属于核心领域就可以了。

因此，第二种描述突出核心的形式就是：

在模型的主要仓库中将核心领域的元素标记出来，而不是专门去阐述它的任务。使开发人员能够毫不费力地了解核心领域内部和外部是什么就可以了。

只需要用相当小的代价和维护成本，对于工作在这个模型上的开发人员来说，核心领域就已是清晰可见了，至少在模型构造方面足以辨别各个部分的组成。

15.5.3 把精炼文档作为开发过程的工具

从理论上说，在 XP 项目中，任意两个结对编程的程序员都可以改变系统中的任何代码。实际上，有些改变需要我们多加磋商和协调才能完成。当修改基础结构层时，改变的影响可能非常清楚；但是在领域层，它的影响可能就不那么明显了，因而常常被我们忽视。

核心领域的概念能帮助我们清楚地看到这种影响。对核心领域模型作出的改变将会产生巨大的影响。虽然一些广泛使用的通用元素发生改变之后可能需要我们修改大量代码，但是它们并不会产生明显的概念转移，这一点与核心的改变不同。

把精炼文档作为一种指南。当开发人员意识到精炼文档本身需要修改，以便保持与代码或模型的改变保持同步时，就会要求进行磋商。他们要么会从根本上改变核心领域的元素或关系，要么会改变核心的边界，把一些东西包含进来或排除出去。模型所发生的改变必须通过各种可用的沟通渠道(如分发一份新版本的精炼文档)发布到整个团队。

如果精炼文档给出了核心领域的主要轮廓，那么我们就可以把它当作一种指示器，用它来指示一个模型改变的重要程度。如果模型或代码的改变影响到精炼文档，那么就需要与团队中的其他成员进行磋商，在改变实施之后，需要立即通知团队的所有成员，并发布精炼文档的新版本。对核心外部(或者不包括在精炼文档中的细节)进行改变时，可以无需进行磋商或通知而直接实施这些改变。其他成员在后续的工作过程中会看到这些变化的。这样，开发人员就完全拥有了 XP 所要求的自治性。

尽管愿景声明和突出核心为我们提供了信息和指导，但是它们并没有实际地修改模型或代码本身。通用子域的具体划分去除了一些无关紧要的元素。下面介绍的模式将从结构上改变模型和设计本身，使核心领域更加清晰、更易于管理。

15.6　内聚机制

封装机制是面向对象设计的一种标准原则，它隐藏了方法中的复杂算法，并通过赋予有意义的名称来告诉人们这个对象“是什么(what)”而非“怎么做(how)”。这种技术使得设计更加容易理解和使用。可是它也存在一些先天不足。

有时候计算方法会非常复杂，以至于设计都受到了很大的冲击，模型中的概念变成了用“怎么做”来解释，而不是用“是什么”来解释。设计中产生了一大堆用来实现算法、解决问题的方法，而描述这个问题的方法却变得模糊不清。

“怎么做”的方法在模型中泛滥成灾，表明模型存在着某种问题。我们可以通过重构来获得更深的理解，使模型和设计的元素更适合于解决问题。最直接的方案是寻找一种能够简化计算机制的模型。但是，我们有时会发现，在计算机制中有些部分本身在概念上是内聚的。我们的计算方法可能并不需要把所有所需的功能全部包含进来，我们需要的也不是一种万能的计算机制。把那些内聚的部分提取出来，这样剩下的机制就更易于理解了。

因此：

把概念上的内聚机制分离到一个独立的轻量级框架中。要特别注意那些公式和具有详细文档的算法种类。用一个释意接口来说明该框架的功能。现在，领域中的其他元素就可以集中于描述问题 “是什么”了，而“怎样做”的复杂细节可以交给框架去完成。

这些分离出来的机制将被放到辅助功能模块中，而留下的核心领域将更紧凑、更具表达能力，并通过一种更接近说明性风格的接口来使用那些机制。

把一些标准的算法和公式界定出来之后，我们就把一部分复杂性从设计中提取出来，转移到了一系列经过深入研究的概念之中。这样，我们不用进行太多的反复试验，就能够满怀信心地实现一种解决方案了。我们还可以寻求外部帮助，可能有其他开发人员已经知道了这些概念，或者至少能够找到相关的资料。这个好处与公开的通用子域模型有点相似，但是，我们找到这些算法资料或者计算公式的机会更大，因为这类问题已经在计算机科学里得到了很多的研究。当然，多数情况下我们还是会需要设计新的算法。在设计算法时，要使算法专门处理计算问题，而不要让描述性的领域模型掺和进来。这是一种职责的分离：核心领域或通用子域用来表达一种事实、规则或者问题；而内聚机制则用来解决规则，完成模型指定的计算。

示例：组织结构图中的一种机制

我曾经在一个项目中参与过这种内聚机制的分离过程。那个项目需要一种相当精细

的组织结构图模型。这个模型用来描述人们之间的工作关系，确定他们属于组织的哪个分支，并提供了一个用来询问或回答相关问题的接口。这些问题大多数都是“在这个行政管理系统中，谁有权对此做出批准”或者“在这个部门，谁有能力解决这种问题”等。开发团队意识到，大部分搜索指定的人或关系的问题都涉及到对组织树的遍历。这其实就是一种需要用图(groph)来解决的问题。图论是一种相当成熟的理论，“图”就是一组用弧线(称为边)连接起来的节点集合，以及遍历图所需要的规则和算法。

负责这部分的开发人员已经做好了一个图的遍历框架，并把它实现为一种内聚机制。他在这个框架中使用了标准的图术语和算法，它们对于计算机专家来说非常熟悉，在教科书中也屡见不鲜。这个框架并没有实现成一个完全通用的模块，而只包含了我们的组织模型所需的各个功能，这只是图的概念框架的一部分而已。同时，由于使用了一个释意接口，因此以何种方式来获取答案并不是我们主要关注的问题。

现在这个组织模型可以用标准的图术语来作一些简单的声明：一个节点代表一个人，他们之间的关系是一条连接这些节点的边(弧线)。作出这样规定后，只要使用图框架所提供的机制就能够找出任意两个人之间的关系了。

如果我们把图的遍历机制和领域模型混在一起的话，那么会造成两个问题。首先，模型会与解决问题的特定方法绑定起来，从而限制了将来的选择。更重要的是，组织模型将会变得非常复杂和混乱。把机制和模型分离开来之后，我们就可以用一种更清楚的、声明性的风格来描述组织结构了。操纵图的复杂代码被分离在一个纯技术性的框架中，这个框架基于已被证明的算法，可以对它单独进行维护和单元测试。

内聚机制的另外一个例子是用一个框架来构造规范(SPECIFICATION)对象并支持其基本的比较和组合操作。通过使用这样的框架，核心领域和通用子域就能够用清晰、更易于理解的语言来描述它们的规范(参见第 10 章)，而进行比较和组合的复杂操作则可以留给这个框架去 完成。

15.6.1 通用子域与内聚机制

通用子域和内聚机制都是为了减轻核心领域的负担而提出来的。其不同之处在于二者担当责任的性质不同。通用子域是用一个描述性的模型来说明团队应该如何看待领域中的一些方面。这一点与核心领域无异，只不过它没有核心领域那么重要、那么专门化。内聚机制不是用来表示领域的，它负责解决一些由描述性模型提出的棘手的计算问题。

模型提出问题，而内聚机制解决问题。

实际上，除非您打算采用一种正式、公开的计算方法，否则这种区别通常不是绝对的，至少在刚开始的时候是这样。经过多次重构以后，这种计算方法可能会被精炼成一

种更单纯的机制，或者转变为一个通用子域，其中可能会包含一些以前未被发现的概念，使机制变得更加简单。

15.6.2　属于核心领域的机制

您可能认为机制应该总是从核心领域中分离出来。但是，如果机制本身就是一项资产，或者属于软件价值的一个关键部分，那么就是一种例外。一些高度专业化的算法有时就属于这种情况。例如，在物流运输应用中需要一个特别有效的计算时间表的算法，如果这个功能正好体现了软件的差异性的话，那么它在概念上就可以认为是核心的一部分。我曾经参与过一家投资银行的开发项目，在项目中用来计算利率风险的高度专业化的算法就被明确地放在了核心领域中。实际上，这些算法受到了非常严密的控制，甚至大多数核心开发人员都不了解它们。当然，这些算法很可能是一组风险预测规则的一种特定的实现。对它们进行更深入地分析后，可能会得到一个更深层的模型，并用一种封装机制将这些规则显式地表达出来。

但是，那些都是以后的事情，是对设计所作的下一步改进。要决定是否进行下一步设计，必须根据各方面的成本收益进行分析：完成新设计有多大的难度？理解和修改目前的设计有多大的难度？对期望完成这项任务的人来说，一个更先进的设计能带来多少简化？还有，有没有人对新模型有何想法？

示例：组织结构图和机制重新绑定

实际上，在我们完成那个组织模型一年之后，其他开发人员又对它进行了重新设计，以消除图框架与组织模型的分离。他们觉得这种分离增加了对象数量，而且把机制分离到一个单独的模块中是不可靠的。相反，他们在组织实体的父类中添加了节点的行为。尽管如此，他们还是保留了组织模型的声明性公共接口，甚至还保留了机制在组织实体内的封装。

像这样的反复是很常见的，但每一次反复都不是回到设计的起始点。最终结果通常是为了获取更深层的模型，使之能够更明显地区分功能、目标和机制。有用的重构在去除不必要的复杂性的同时，还保留了中间阶段的主要优点。

15.7　精炼到声明性风格

声明性设计和“声明性风格”是第 10 章的主题，但是在战略性精炼的问题上，这种设计风格值得特别提出来。精炼的价值在于能够看到正在做的工作：不被无关的内容干扰，直指系统的本质。如果辅助设计能在封装和实施计算方法的同时，为核心概念和规

则的表达提供一种简约的语言，那么核心领域的重要部分就可以获得声明性的风格了。

如果内聚机制能够通过释意接口进行访问，同时提供一套在概念上一致的断言和无副作用函数，那么它将会有更加有用。机制和柔性设计使核心领域能够使用含义清晰的声明，而不是调用晦涩难懂的函数。如果核心领域部分自身能够突破到一个更深层模型，并且能够成为一种语言，来灵活、简明地表达应用中最重要的情况，就更会带来意想不到的效果。

深层模型通常是伴随相应的柔性设计产生的。成熟的柔性设计将提供一组易于理解的元素，我们可以像单词造句那样，把这些元素明确地组合起来，以便完成复杂的任务或者表达复杂的信息。而且，用户代码也能获得声明性风格，并且能够变得更加精炼。

构造通用子域可以减少设计的混乱，而内聚机制则可以封装复杂的操作。它们将留下一个重心突出的模型，并减少那些对用户活动价值不大的东西对模型的干扰。但是，我们很可能尚未为那些不属于领域核心的部分找到合适的去处。“隔离核心”将采取一种直接的方式来从结构上划分核心领域。

15.8 隔离核心

模型中的元素有些服务于核心领域，有些则起着辅助的作用；核心元素与通用元素之间可能有着紧密的联系；核心概念之间的内聚性可能并不那么健壮或直观。所有混乱的关系使核心难以突出。如果设计者不能清楚地看出最重要的关系，就难以开发出一个良好的设计。

通过构造出通用子域，从领域中清除一些掩盖核心的细节，可以使核心更加直观。但是，把所有的子域都确定和清理出来是一项艰巨的工作，而且有些子域并不值得我们花那么多时间。而与此同时，最重要的核心领域仍与剩余的部分混在一起。

因此：

重构模型，把核心概念与辅助概念(包括定义模糊的概念)分离开来，加强核心的内聚力，同时减少核心与其他代码的联系。把所有通用元素和辅助元素分离出来放到其他模块中，即使这样的重构会分割联系紧密的元素也在所不惜。

这里使用的原则基本上与我们应用在通用子域上的原则一样，只是实施方向不同而已。内聚子域对于我们的应用来说是非常重要的，我们可以把它们界定出来，并划分到相关的模块中。如何处理剩下的元素(这是一堆大杂烩)也很重要，但其重要性要相对低一些。这些元素可以原地不动，也可以放到其他模块中去。最终，越来越多的剩余元素被分离到通用子域中。就目前来说，不管采用什么简单的分离方法都行，这样我们才能把重点放在隔离核心上。

分离隔离核心需要的步骤如下：

(1) 确定一个核心子域(这可以从精炼文档中得到)。

(2) 把相关类移至一个新的模块，并为模块指定一个与其概念相符的名称。

(3) 重构代码，把那些不能直接表示该概念的数据和功能分离出来(可能要分离为新的类)，并移到其他的模块中。尽量把它们放到概念上相关的模块中去，但是不浪费过多的时间在这个问题上来追求完美。把工作的重点放在精炼核心子域上，核心子域对其他的模块的引用必须突出、明确。

(4) 对得到的隔离核心模块进行重构，使其中的关系和交互作用更加简单易懂，同时尽量减少它与其他模块的关系(这也是这个步骤中重构的目标)。

(5) 对其他核心子域重复上面的步骤，直到得到隔离核心。

15.8.1　创建隔离核心的代价

对核心进行隔离有时候会破坏非核心类之间的紧密联系，使之变得模糊甚至复杂，但这并不会得不偿失，因为它可以使核心领域更加清晰、更容易处理。

隔离核心能增强核心领域的内聚力。分解模型的方式有很多种。在创建隔离核心时，为了获得核心领域的内聚性，我们有时可能会破坏一个本来内聚性很好的模块。这是一种净增益，因为企业应用软件最大的附加值来自于模型中企业特定的性质。

当然，分离核心的工作量是非常大的。我们必须知道，决定创建一个隔离核心有可能会牵涉到整个系统中的所有开发人员。

如果系统有一个很大的限界上下文，它对于系统非常关键，但是其中的核心部分却被大量辅助功能所掩盖，这时就需要考虑创建一个隔离核心了。

15.8.2　推进团队决策

和许多战略性设计决策所要求的一样，开发团队必须作为一个整体转移到隔离核心上来。这样的转移过程只有一支足够自律的和协调的团队才能完成，因此它需要经过团队的决策同意。最困难的地方在于，我们既要限制团队中的每个人都使用同一个核心定义，但同时又不能限制他们作出隔离核心的决定。由于核心领域和设计中的其他部分一样是不断推进的，在隔离核心之上的工作经验将使我们对哪些是核心元素、哪些是辅助元素产生新的认识。这些认识应该反馈到核心领域和隔离核心模块中来，使它们的定义得到进一步精化。

这意味着新的认识必须能够被团队即时共享，但是个人(或者编程对子)不能单方面对这些认识自行其是。无论采用什么决策过程，不管是全体通过还是领导拍板，它都必须足够敏捷，以便能及时修正开发的方向。团队内部的沟通必须非常高效，以保证每个人对核心的认识都是一致的。

示例：隔离货物运输模型的核心

我们从图 15-2 中所示的模型开始，把它作为货物运输调度软件的基础。

注意，这个模型同实际应用需求相比已经作了高度简化。作为例子，实际的模型会显得过于复杂。因此，尽管这个示例可能并不是很复杂，不足以让我们想起要去构造隔离核心，但是发挥您的想象力，把它看成是一个非常复杂的、很难解释的模型，而且不能够被当作一个整体来处理。

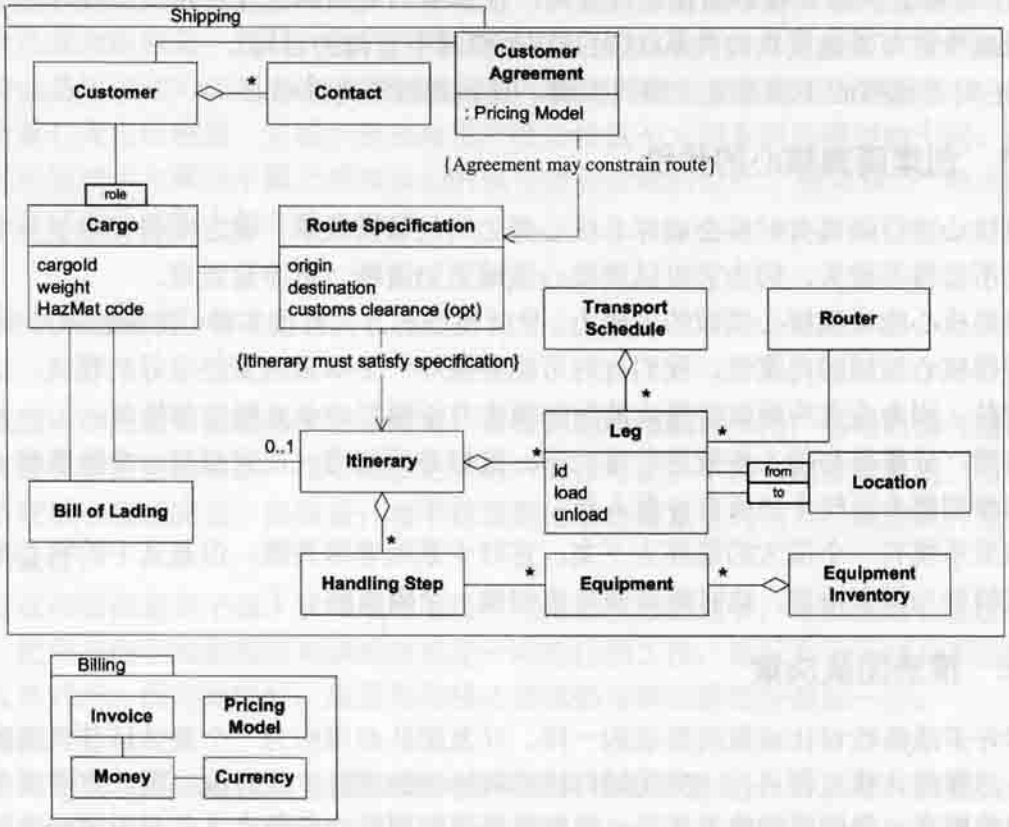


图 15-2　货物运输模型

好了，运输模型的核心是什么呢？通常我们可以找一个好的“底线”来开始分析。在此我们主要关心的是价格和货物。但是我们必须首先考虑领域愿景声明，下面是它的一段摘录。

……增加操作的可视性，并提供能够更迅速可靠地实现客户需要的工具……

这个应用不是为营销部门设计的，它将由公司的前端操作人员来使用，所以我们把所有同金钱有关的问题都归类为辅助功能(当然这些功能也很重要)。有人已经把其中的一些功能放到一个独立的包(Billing)中去了。我们可以保留它，并进一步确定它是起辅助性作用。

模型的中心需求是货物转运，即根据客户的需要运送货物。把与这些活动关系最直接的类提取出来，我们就能得到一个新的隔离核心。我们把它放在新建的 Delivery 包中，如图 15-3 所示。

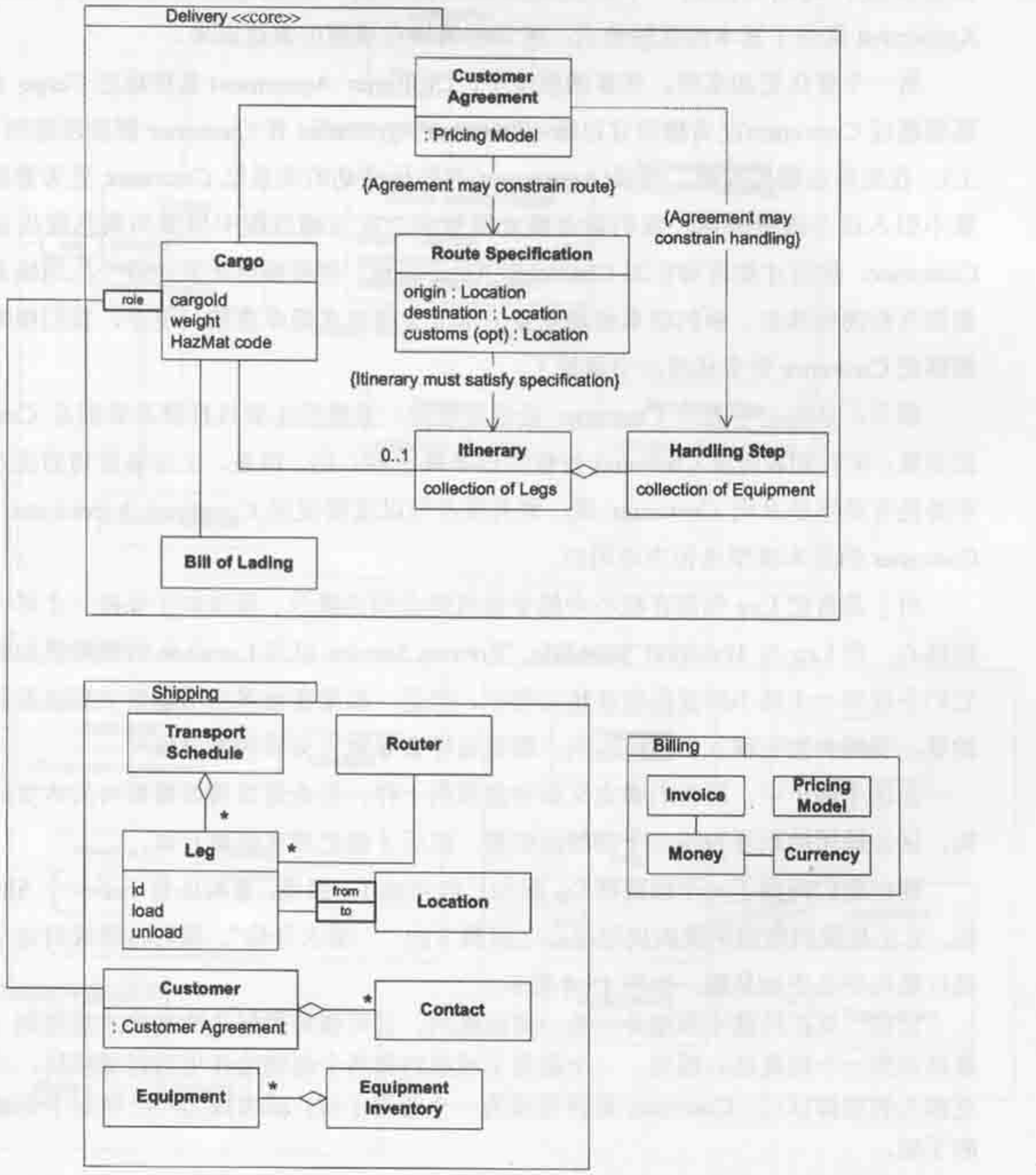


图 15-3　根据客户需要进行可靠运送是项目的核心目标

我们主要是把一些类移到了新的包中，但模型自身也有几处改变。

首先，Customer Agreement 约束着 Handling Step。这个联系在团队分离核心时通常都会发现。我们关注的一个焦点是货物能否高效、正确地运抵目的，而 Customer Agreement 提供了基本的运输要求，应该明确地在模型中表达出来。

另一个变化更加实用。在新的模型中，Customer Agreement 直接连在 Cargo 上，不需要通过 Customer(在货物预订以后，Customer Agreement 和 Customer 都必须连到 Cargo 上)。在实际运输过程时，协议(Agreement)与转运活动的关系比 Customer 更为紧密。如果不引入这个改变的话，我们就必须先根据客户在运输过程中扮演的角色找出正确的 Customer，然后才能查询它的 Customer Agreement，使得模型在表达各个应用场景时都显得有些拐弯抹角。新的联系使最重要的部分变得极其简单直接。现在，我们很容易就能够把 Customer 完全从核心中拿掉了。

那么，从核心中去掉 Customer 是否合理呢？系统的主要目标就是要满足 Customer 的需要，所以初看起来 Customer 好像应该是属于核心的。但是，在运输货物的交互过程中并没有经常涉及到 Customer 类，而且现在可以直接使用 Customer Agreement，所以 Customer 的基本模型是相当通用的。

对于是否把 Leg 保留在核心中的争论可能会相当激烈。我倾向于保持一个尽可能小的核心，而 Leg 与 Transport Schedule、Routing Service 以及 Location 的联系更加紧密，它们中任何一个都不需要保留在核心当中。但是，如果在很多应用场景中都涉及到 Leg 的话，我就会把它移入 Delivery 包，即使这样会导致它与那些类分隔开。

在这个例子中，所有的类定义都和前面的一样，但是通常精炼需要对类本身进行重构，区分通用的职责和特定于领域的职责，然后才能把后者隔离开来。

现在我们得到了一个隔离核心，重构已经完成了。但是，重构还留下了一个 Shipping 包，它正是我们前面所说的提取核心之后剩下的“一堆大杂烩”。我们将继续对这个模块进行重构使之更加易懂，如图 15-4 所示。

记住，现在的这个模型并不是一挥而就的，它可能需要好几次重构才能得到。我们最终得到一个隔离核心模块、一个通用子域模块和两个起辅助作用的领域模块。在得到更深入的理解以后，Customer 最终将成为一个通用子域，或者成为一个特定于运输应用的子域。

识别有用的、有意义的模块是一种建模行为(见第 5 章的讨论)。开发人员和领域专

家在战略性精炼上的合作将作为知识消化过程的一部分。

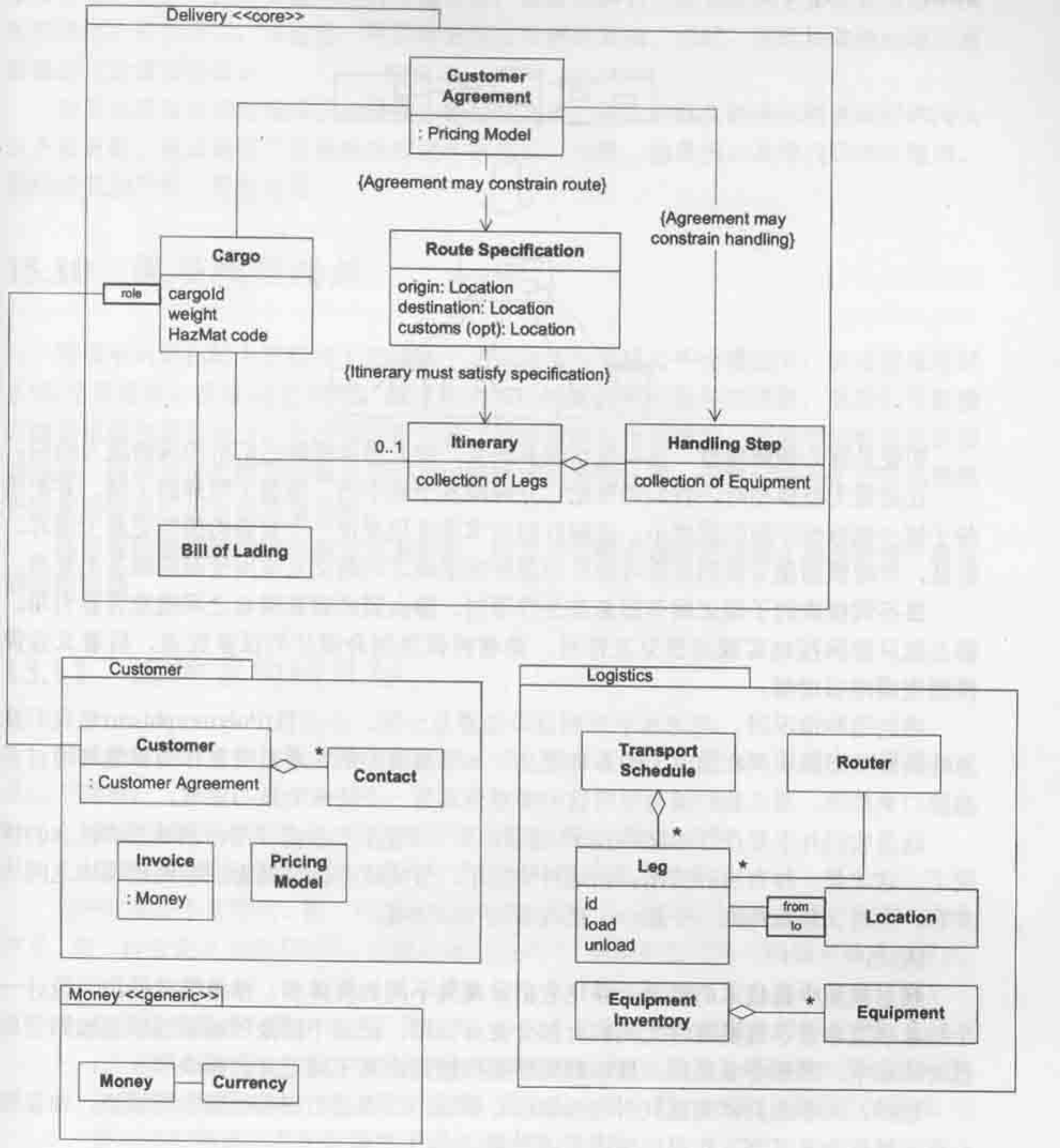
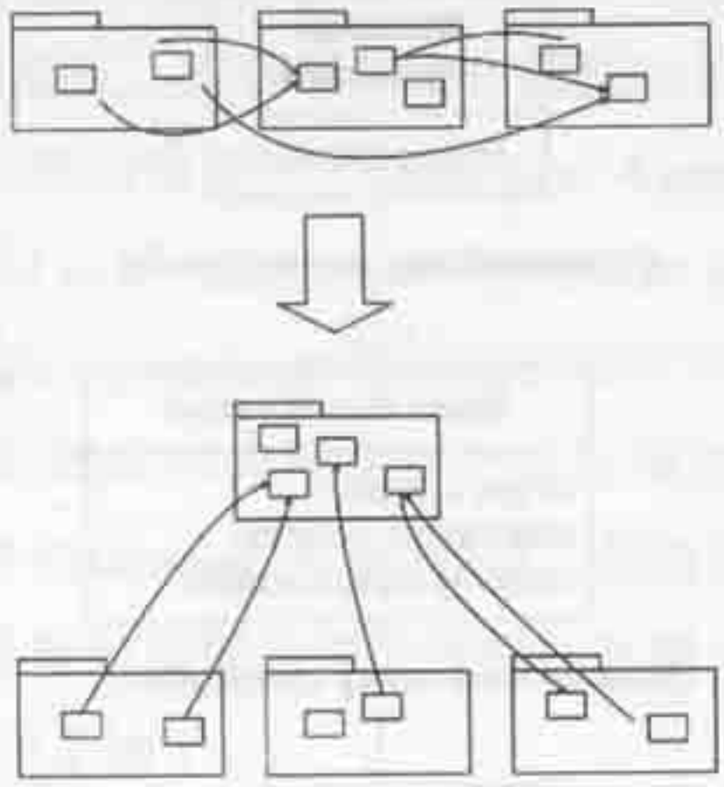


图 15-4　完成隔离核心后留下的非核心子域模块

15.9 抽象核心



即使是核心领域模型，也会包含很多细节，使人难以理解它们所构成的庞大结构。

在处理大型模型时，我们经常把它分解成多个较小的、更易于理解的子域，并把这些子域分别放到不同的模块中。这种打包技术常常用来使一个复杂的模型更易于操作。但是，有时候创建分离的模块可能反而会导致子域之间的交互作用变得模糊甚至复杂。

当不同模块的子域之间有很多交互作用时，要么就必须在模块之间建立许多引用，要么就只能间接地实现这些交互作用。前者将丧失划分模块的很多优点，后者又会使模型变得难以理解。

遇到这种情况时，考虑水平分割而不是垂直分割。多态性(Polymorphism)使我们能忽略抽象类型的实例在细节上的各种变化。如果模块中的大多数交互作用都能够通过多态接口来描述，那么我们就可以将这些类型分离为一个特殊的核心领域。

这里我们并不是在寻求某种技术上的技巧。只有在多态接口符合领域基本概念的情况下，这才是一种有用的技术。在这种情况下，分离这些抽象概念能够消除模块之间的关联，同时又精炼得到一个更小、更内聚的核心领域。

因此：

找出模型中最根本的概念，并把它们分离为不同的具体类、抽象类或接口。设计一个抽象模型来表示重要组件之间的大部分交互作用。把这个抽象的纲领性模型放到它自己的模块中，而那些专用的、具体的实现类则放到由其子域定义的模块中。

这样，大多数具体类就只引用抽象核心模块，而不会引用其他的专用模块。抽象核心将主要概念及其交互作用简洁地描述出来。

分离抽象核心并不是一个机械的过程。例如，如果把各个模块经常引用的所有类都

自动分离到一个独立的模块中，那么得到的结果可能会是一堆毫无意义的杂乱物。对抽象核心进行建模不仅需要深入了解关键概念，还要了解它们在系统的主要交互作用中所起的作用。换句话说，这也是一种面向更深层理解的重构。同时，构造抽象核心还经常需要进行大量重新设计。

如果在项目中同时使用了抽象核心和精炼文档，并且精炼文档也在随着理解的深入而不断更新，那么最终二者看起来应该非常相似。当然，抽象核心是用代码来实现的，因此将更加严密、更加完善。

15.10　深层模型精炼

精炼不只是在较大的粒度上把领域中的某些部分从核心中分离出来，它还意味着对子域(尤其是核心领域)进行精炼，通过连续的重构来获得更深入的理解，使我们不断接近深层模型和柔性设计。它的目标是使设计能够清晰地反映模型，而模型能够简洁地描述领域。深层模型把模型中最本质的方面精炼成简单的元素，使我们能够把这些元素组合起来去解决应用中的重要问题。

尽管深层模型的突破总是会带来好处，但是只有核心领域的突破才能改变整个项目的发展轨道。

15.11　选择重构的目标

如果您碰到一个结构极差的大型系统，该从哪里入手呢？在 XP 阵营中，答案往往是以下两种：

- 随便从哪个地方着手，因为所有部分都是要被重新构造的。
- 从存在问题的地方开始。重构完成特定任务所需的部分。

两种答案我不都赞同。第一种答案有些不切实际，除非参与项目的都是一些顶尖的程序员。第二种答案是治标不治本，回避最麻烦的部分，最终这会使得代码越来越难以重构。

我们既不能眉毛胡子一把抓，又不能头痛医头、脚痛医脚，那么应该怎么做呢？

- 如果想从存在问题的地方开始重构，那么要注意问题的根源是否涉及到核心领域(或者核心与辅助元素之间的联系)。如果确实如此，就要首先解决核心领域的问题。
- 如果可以随便从哪个地方开始重构，那么首先集中精力对核心领域进行重构，对核心进行隔离，并把辅助性子域提炼成通用子域。

这就是如何选择重构目标的最好办法。