第11章

应用分析模式

深层模型和柔性设计来之不易。我们必须通过大量的领域学习、大量的交谈以及大量的反复尝试才能取得进展。但是，有时我们也可以从经验中得到帮助。

当面对一个领域问题时，有经验的开发人员可能会看到一种熟悉的职责类型或者关系网。此时，他就能借用以前解决类似问题的方法来处理。他会回忆，原来尝试过哪些模型，哪个模型解决了问题？在实现的时候出现了哪些问题，是如何解决的？早期经验中反复尝试的情形忽然之间与眼前的问题联系起来。有些模式还被人们记录下来供人分享，使我们能够从这些积累的经验之中获益。

与我们在第 II 部分给出的基本构造块模式以及第 10 章给出的柔性设计原则相比，这些模式层次更高、更具有针对性，其中还使用了一些对象来描述概念。这些模式帮助我们节约了反复尝试所需的大量时间和精力，同时还让我们无需为一些微妙的问题付出太高的学费。利用这些模式，我们一开始就能得到一个具有较强的表达能力和可实现性的模型，从更高的起点来开始重构和实验。但是，这些模式不是现成的解决方案。

在 *Analysis Patterns: Reusable Object Models* 中，Martin Fowler 将他的模式定义如下：

> 分析模式是业务建模中用来表示一种常见构造的概念组。它可以仅仅与一个领域相关，也可以跨越多个领域。[Fowler 1997]

Fowler 描述的分析模式来自于领域经验，因此只要运用得当，它们都是非常实用的方案。如果我们面对着极具挑战性的领域，那么这些模式可以提供一个宝贵的起点，使我们能够基于这个起点来开始后续的迭代开发过程。“分析模式”这个名称强调了其本质是概念性的。分析模式不是技术性的解决方案；它们充当着向导的作用，来帮助我们在特定的领域中开发模型。

遗憾的是，这个名字没有传达出另一层意思，那就是这些模式还包含了对实现(包括一些代码)的重要讨论。Fowler 很清楚，光做分析而不考虑实际设计是不够的。在下面这个有趣的例子中，Fowler 甚至考虑到了一个特定的建模选择在软件部署以后对长期的系统维护所产生的影响。

他说道，当我们创建一个新的记账实务时，我们就创建了一个由过账规则新实例构成的网络。我们可以在不重新编译和构建的情况下这样做，同时仍然保持系统正常工作和运转。虽然需要加入一种新过账规则子类型的情况比较罕见，但这种情况是不可避免将要出现的。

在一个成熟的项目中，建模选择往往是根据应用经验的指导来做出的。在作选择之前，我们可能已经尝试了各个组件的多种实现方案，其中的一些实现方案可能已经融入到产品之中，甚至进入了维护阶段。这些经验将帮助我们避免许多问题。分析模式最大的好处就是，它能够将其他项目中的经验(包括对模型的理解、对设计方向和实现结果的广泛讨论等)描述出来。脱离模式上下文来讨论建模思想不仅会使模式难以应用于实际，还会出现分析与设计割裂的危险——这种现象与模型驱动设计背道而驰。

通过例子来解释分析模式的原则和应用比通过抽象的描述来解释要好。在本章中，我将给出两个例子，其中开发人员使用了“Inventory and Accounting(存货与记账)”一章(Fowler 1997)里一个具有代表性的小模型，其中的分析模式概括得恰好足够支持例子。显然，我们并不是要对分析模式进行分类，更不是要把示例模式从头到尾解释一遍。我们的目的是演示如何把分析模式集成到领域驱动设计过程中。

示例：利息计算之账目版

第 10 章演示了开发人员可以通过多种可能的方法寻找深层模型，来为一个特定的专业记账应用进行建模。下面是另一种场景。这一次，开发人员将从 Fowler 的 *Analysis Patterns* 一书中挖掘出有用的思路。

首先让我们复习一下。记账应用用来对贷款及其他有息资产进行跟踪，计算它们所产生的利息和手续费用，并跟踪借款人的付款情况。每天晚上金融公司都会运行一个批处理脚本(以下称之为 nightly batch)，将这些数据提取出来并传给老的记账系统，并指明每个数额应该过账到那个分类账中。这个设计能够工作，但是它使用起来非常困难，修改起来非常复杂，也不便于沟通，如图 11-1 所示。

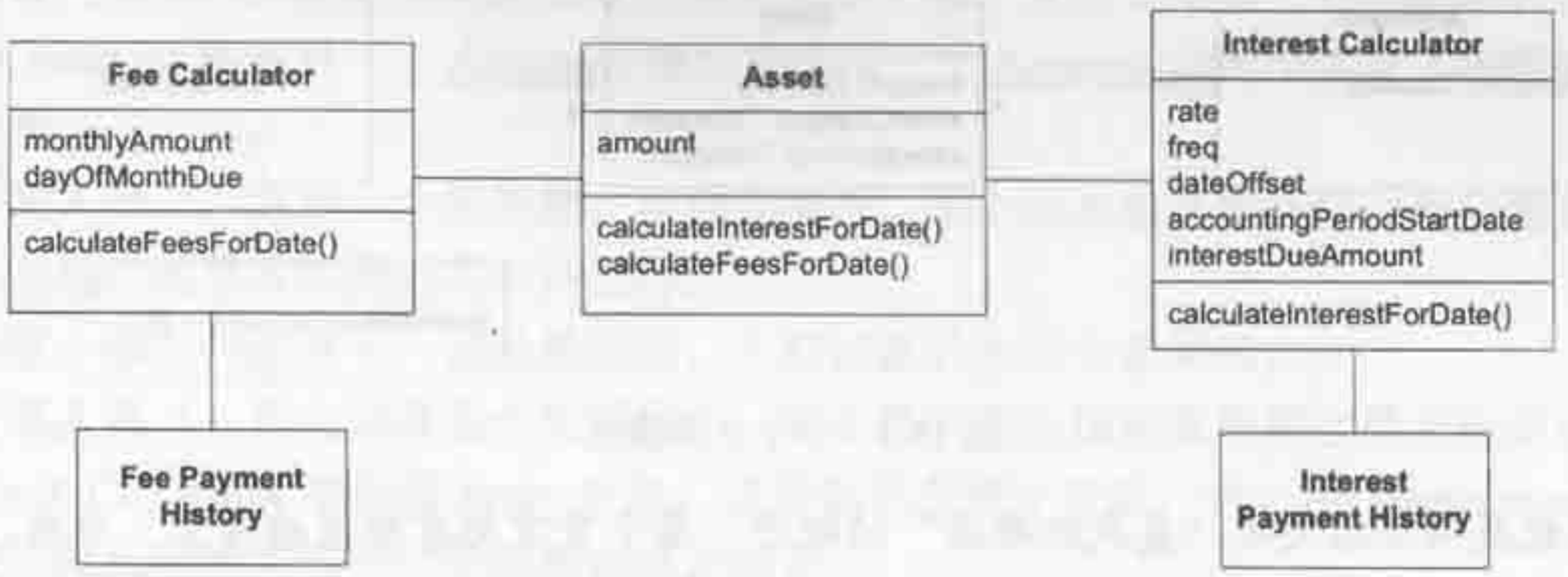


图 11-1　最初的类图

开发人员决定阅读 *Analysis Patterns* 的第 6 章，“Inventory and Accounting(存货与记账)”。下面是那一章中最相关部分的摘要。

Analysis Patterns 中的记账模型

各种商业应用都要跟踪账目。账目是用来保存数值的(通常是指钱)。在许多应用中，仅仅跟踪账目的总额是不够的。记录并控制账目的每一次修改相当重要。这也是记账模型(accounting model)最根本的出发点，图 11-2 所示为一个记账模型。

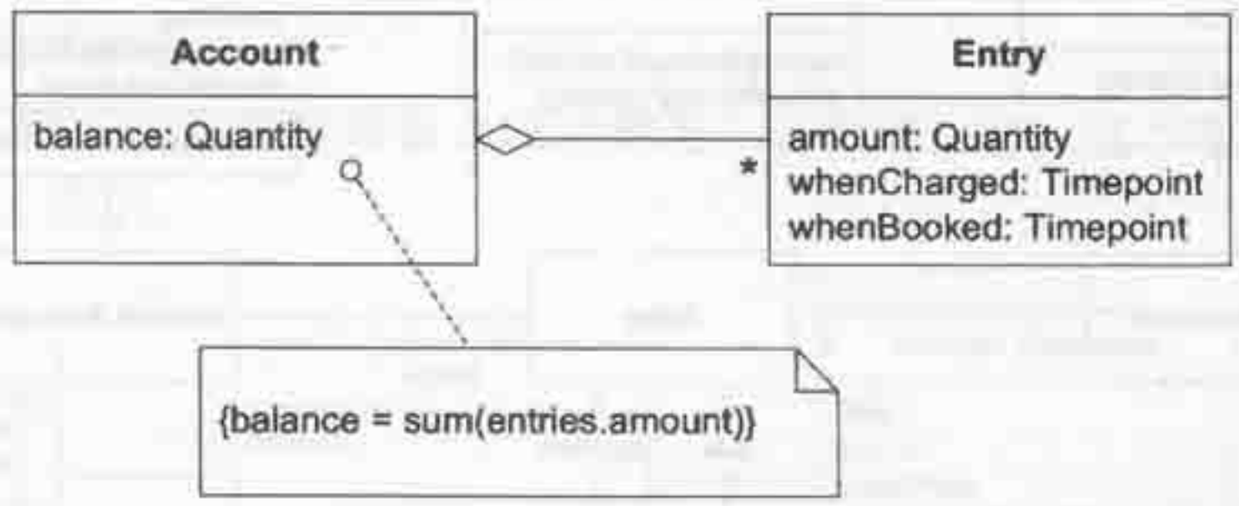


图 11-2　一个基本的记账模型

插入一个 Entry(条目)可以把数值加入到账目中，而删除一个数值则是通过插入一条“负的”Entry 来实现的。Entry 是永远不能删除的，因此整个历史都保留下来了。账目余额就是所有 Entry 的累加结果。在实现上，余额可以根据需要计算出来，也可以缓存起来。Account 接口对这种实现选择进行了封装。

记账的一个基本原则就是守恒性。钱不能无中生有，也不能不翼而飞。它只能从一个 Account 移动到另一个 Account，如图 11-3 所示。

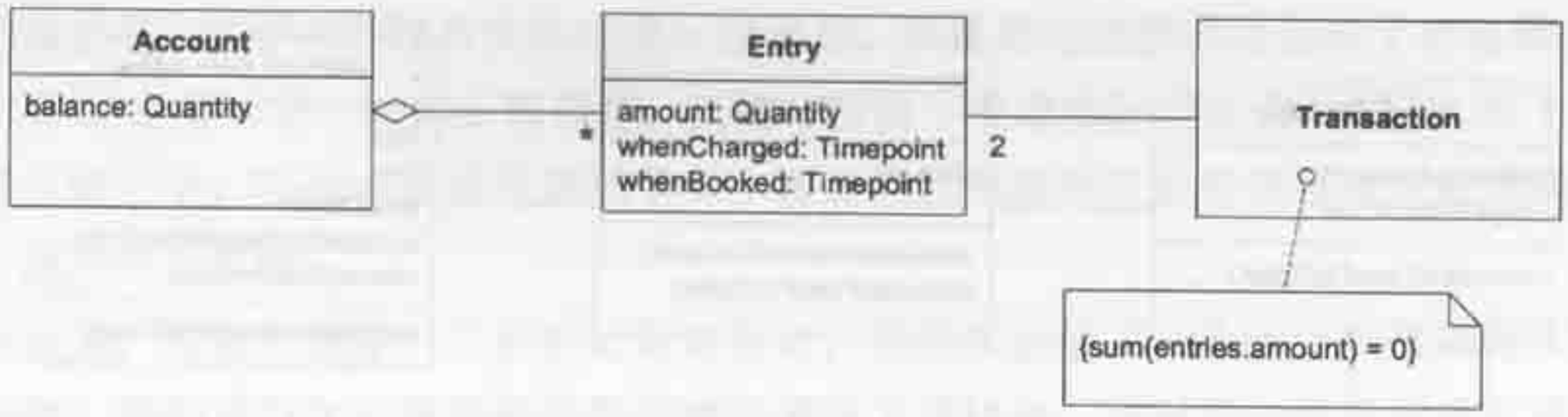


图 11-3　一个交易模型

这就是确立已久的“复式记账法”的概念：每个贷方都有相应的借方。当然，就像其他守恒原则一样，这个原则仅仅适用于封闭的系统，即所有的来龙和去脉都包含在系统中。许多简单的应用并不需要这样严格。

在 Fowler 的书中涉及到了这些模型更为详细的形式，并对设计权衡做了大量讨论。

通过阅读，开发人员(开发人员 1)获得了一些新的思路。她把这章给她的同事(开发人员 2)看了。开发人员 2 正在和她一起开发利息计算逻辑，并负责编写 nightly batch 程序。他们草拟出对模型的修改，将从书中看到的一些模型元素集成了进去，如图 11-4 所示。

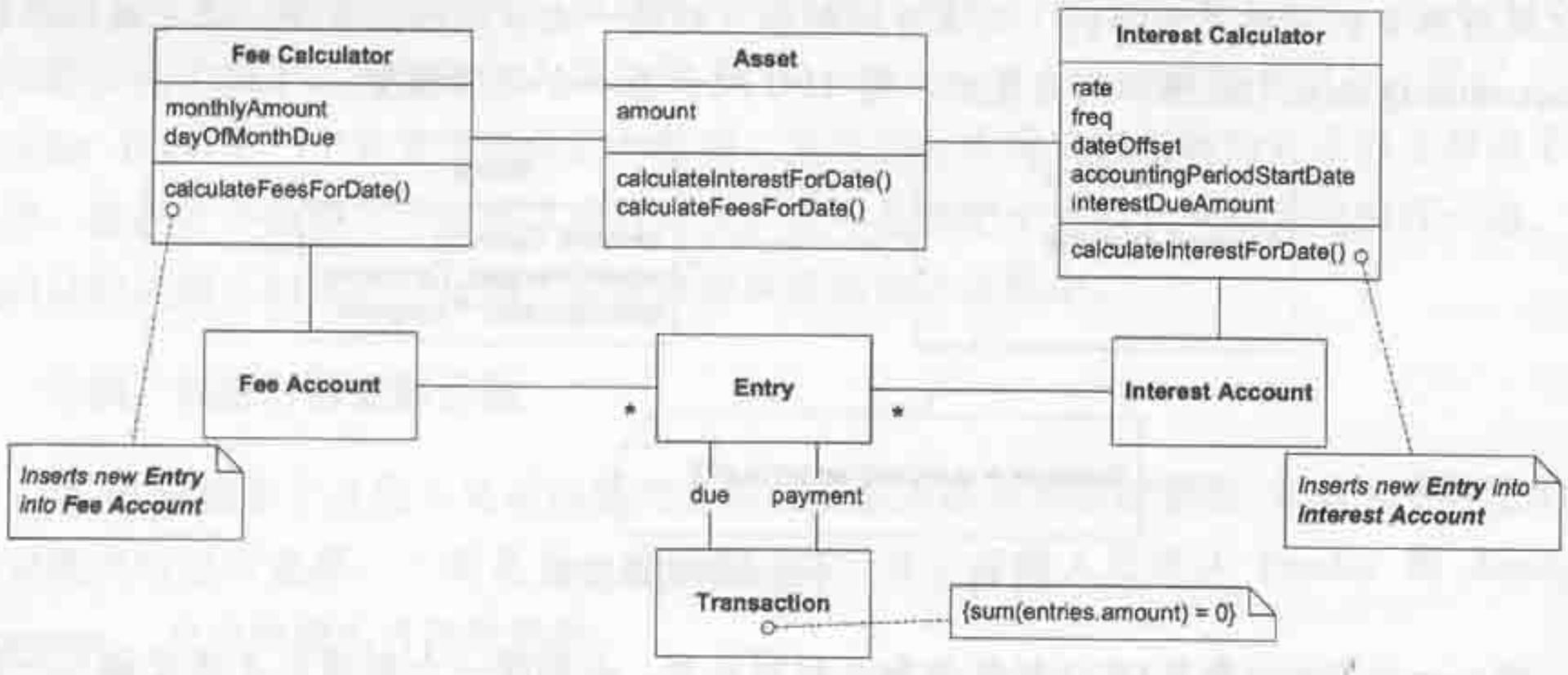


图 11-4　新提出的模型

然后他们把领域专家(简称专家)也拉了进来，一起讨论他们新的建模思路。

开发人员 1：用这个新模型，我们为每笔利息收益向 Interest Account 中加入一条 Entry，而不仅仅是调整 interestDueAmount。然后，另一条付款的 Entry 会把它销平。

专家：那我们现在就能看到所有应计利息和付款的历史了？这可是我们一直希望的功能。

开发人员 2：我怀疑 Transaction 的用法是否正确。从定义来看，Transaction 是将钱从一个 Account 移到另一个 Account，而不是在同一个 Account 的两个条目之间移动使之互相平衡。

开发人员 1：这是一个好问题。我也在担心，书上的意思很像是说一个交易是在一瞬间创建的。利息可以推迟几天再偿付。

专家：那些付款并不一定是推迟的。人们付款的方式非常灵活。

开发人员 1：那么这就是个死胡同了。我在想我们应该把某些隐含概念标识出来。让 Interest Calculator 来创建 Entry 对象，这样似乎更便于交流一些。而 Transaction 似乎能够把计算出来的利息和付款清晰地绑定起来。

专家：我们为什么需要把应计费用和付款绑定起来呢？它们是分开过账到记账应用的。Account 的平衡才是主要的。只要有了所有的 Entry，我们就什么都有了。

开发人员 2：您的意思是您并不跟踪别人是否已经偿付了利息？

专家：嗯，我们当然要跟踪，但是它不像一条应计费用对应一条付款那么简单。

开发人员 2：如果不再去考虑这个联系，就会简化很多事情。

开发人员 1：好的，这个如何？[把原来的类图拿过来开始动手修改。]另外，您几次提到了应计费用这个词。麻烦解释一下它的含义好吗？

专家：当然可以。应计费用就是当费用或收益发生的时候您就把它入账，而不考虑现金是什么时候发生转移的。因此，我们每天都计算利息，但是直到某个时候(比方说月底)我们才收到利息的付款。

开发人员 1：哦，我们确实需要“应计费用”这个词。好，这个呢？

结果如图 11-5 所示。

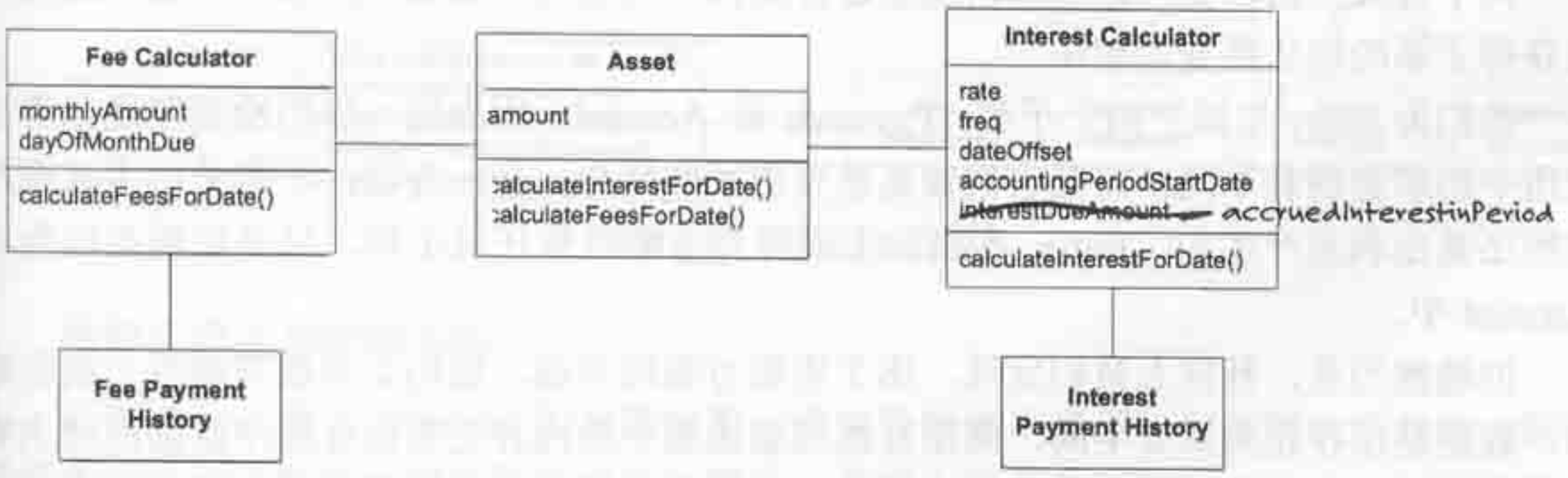


图 11-5　最初的类图，应计费用和付款是分开的

开发人员 1：现在我们把与付款相关的复杂计算全部甩掉了，而且引入了一个新的术语——应计费用，这个词能更好地揭示内涵。

专家：那我们还会有 Account 对象吗？我希望能够把所有信息都集中在 Account 中，包括应计费用、付款和余额。

开发人员 1：是吗？既然如此，也许这样能行[拿过另一张图动手画起来]。

修改结果如图 11-6 所示。

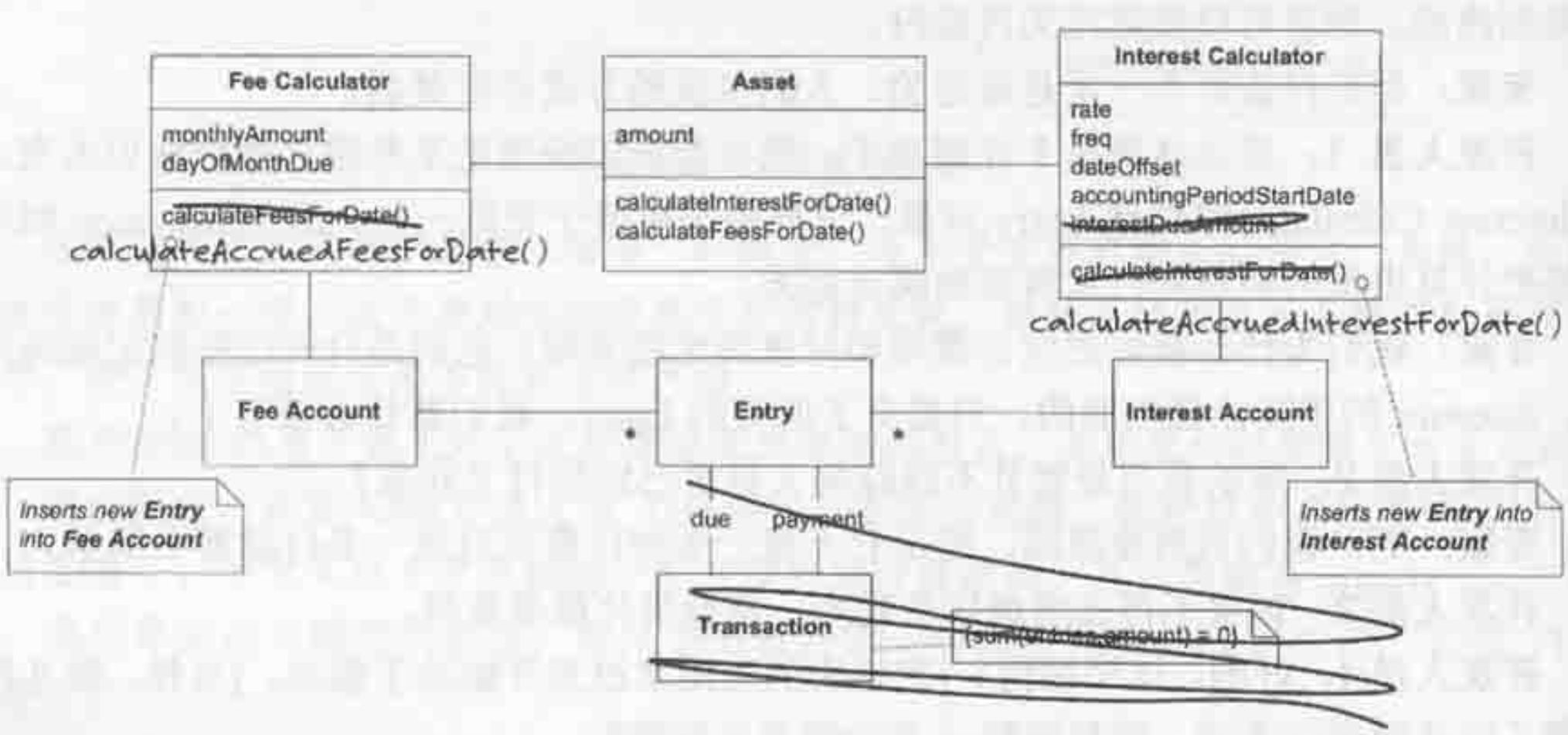


图 11-6　基于账目的图，其中没有 Transaction

专家：这个看起来确实不错！

开发人员 2：nightly batch 脚本很容易就能改成使用这些新对象。

开发人员 1：我得花上几天时间才能让新的 Interest Calculator 正常运行，还要改相当一部分测试。但是修改以后测试会更清晰易读。

两个开发人员回去开始根据新模型进行重构。当他们开始编码把设计付诸实践时，又获得了新的精化模型的想法。

他们为 Entry 生成了两个子类，Payment 和 Accrual，因为进一步的检查发现二者在应用中的职责稍有不同，而且它们都是重要的领域概念。另一方面，不管是由手续费产生的还是由利息产生的，Entry 在概念上或行为上都没有任何不同，只是出现在恰当的 Account 中。

但遗憾的是，开发人员们发现，出于实现方面的考虑，他们必须放弃最近一次的抽象。数据是保存在关系表中的，而项目规范要求表中的内容无需运行程序就能描述出来——这意味着手续费和利息的条目必须分别保存到不同的表中。根据所使用的特定的对象-关系映射框架，开发人员对此惟一能做的就是创建具体的子类(Fee Payment、Interest Payment 等)。如果使用一种不同的基础结构，他们也许可以避免这种笨拙的展开方法。最终结果如图 11-7 所示。

我把这个小花招放在这个大部分是虚构的故事中，是为了演示我们在现实世界中会不断遇到小问题。我们不得不作出恰当的折衷，然后继续前进，但不会因为折衷而偏离模型驱动设计的方向。

新的设计非常易于分析和测试，因为大部分复杂的功能都封装在无副作用函数中。剩下的命令的代码很简单(因为它只调用各种函数，并且用断言对其进行了刻画。

有时我们很难想到程序的某些部分也可以通过领域模型来获得帮助。这些部分在开始时非常简单，只要按部就班地实现就行了。它们看起来就像复杂的应用代码，而不是领域逻辑。分析模式在为我们揭示这些盲点时特别有用。

在接下来的例子中，开发人员对 nightly batch 程序产生了新的理解。以前他从不把这个黑盒看成是面向领域的。

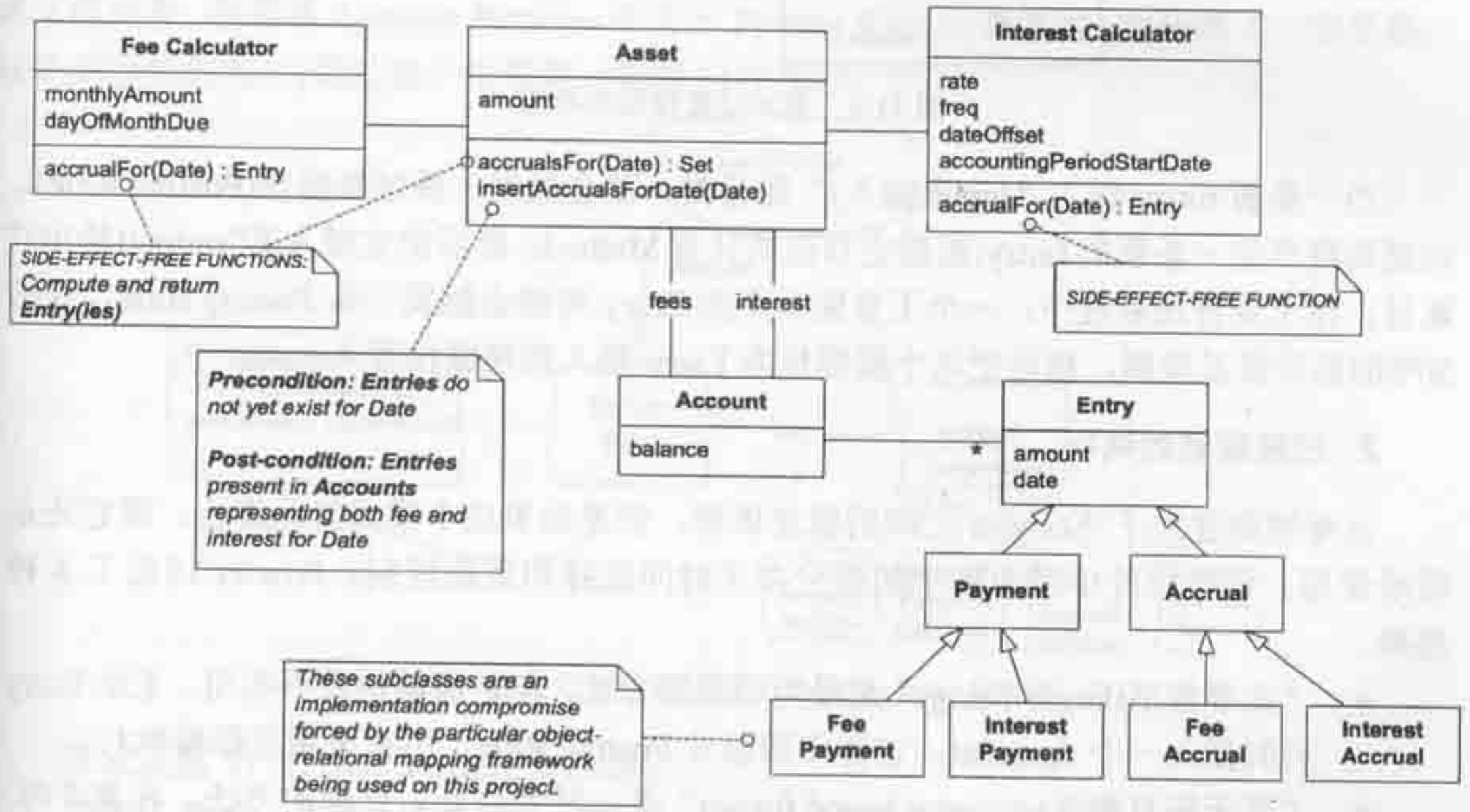


图 11-7　实现后的类图

示例：深入 nightly batch

几周以后，基于 Account 的改进模型已经实现得差不多了。但事情往往如此，新的更清晰的设计又使其他的问题变得更加明显。开发人员 2 在改写 nightly batch(使之能与新设计进行交互)时发现，批处理的行为和 *Analysis Patterns* 中的某些概念之间存在着联系。下面是他找到的相关概念的摘录。

1. 过账规则

记账系统常常为同一个基本财务信息提供多种视图。图 11-8 所示为基本过账规则的类图。一个账目可能用来跟踪收益，而另一个可能用来跟踪这些收益的估计税额。如果系统希望对估计税额的账目进行自动更新，那么这两个账目的实现就会变得相当纠缠不清。在某些系统中，大部分账目条目都是由类似的规则产生的；这样的系统中的依赖逻辑就像一团乱麻。即使在更小一些的系统中，交叉过账也是很不好处理的。要消除依赖关系的纠缠，第一步就是通过引入新的对象来使规则显现出来。

图 11-8　基本过账规则的类图

当一条新 Entry 进入“input(输入)”账目时，就会触发一条过账规则(Posting Rule)。该规则将产生一条新的Entry(根据它自己的计算Method)，然后把它插入其“output(输出)”账目。在工资管理系统中，一个工资账目中的 Entry 可能会触发一条 Posting Rule，算出30%的估计收益税额，然后把这个税额作为 Entry 插入到税额扣缴 Account 中。

2. 过账规则的执行

过账规则建立了 Account 之间的概念依赖，但是如果这个模式就此终止，那它还是很难使用。依赖设计中最为棘手的部分在于时间选择和更新控制。Fowler 讨论了 3 种选择。

- “贪婪激活(Eager firing)”是最为明显的方法，但是通常也最不实用。无论 Entry 何时插入一个 Account，它都立即触发 Posting Rule，所有更新立即将执行。
- “基于账目激活(Account-based firing)”是一种允许延迟处理的方法。在某个时刻，Account 将收到一条消息，然后它触发其 Posting Rule 来处理自上次激活以来新插入的所有 Entry。
- 最后，“基于过账规则激活(Posting-Rule-based firing)”的方法是由一个外部代理发起的，这个代理告诉 Posting Rule 需要激活。Posting Rule 负责查找自上次激活以来插入其输入 Account 中的所有 Entry。

虽然同一个系统可以混合使用多种激活模式，但每个特定的规则集合都需要清晰地定义出，谁是其执行的发起者，以及由谁负责查找输入 Account 中的 Entry。要成功地运

用这个模式，就必须将这 3 种激活模式加入通用语言——其重要程度不亚于模型的对象定义本身。这样做不仅消除了含糊不清之处，还使我们能够直接从 3 种定义清晰的方法中进行选择。这些激活模式标明了一个很容易被忽视的问题，同时也为我们提供了必要的词汇来进行清晰的讨论。

开发人员 2 需要一个倾听者来讨论他的新思路。于是他找到了自己的同事——开发人员 1。开发人员 1 原来的主要任务是对应计费用建模。

开发人员 2：我有点感觉到，nightly batch 开始变成了一个藏污纳垢的地方。有些领域逻辑被隐含到了脚本之中，而且它正在变得越来越复杂。我早就想通过模型驱动设计把领域层从脚本中分离出来了，这样可以把脚本本身实现为领域之上很简单的一层。但是我一直没有想出这个领域模型应该是什么样子，它看起来就是一些过程，并没有真正意义的对象。在阅读 *Analysis Patterns* 中关于 Posting Rule 的章节时，我获得了一些思路。这是我所想到的。[递过来一张草图，如图 11-9 所示。]

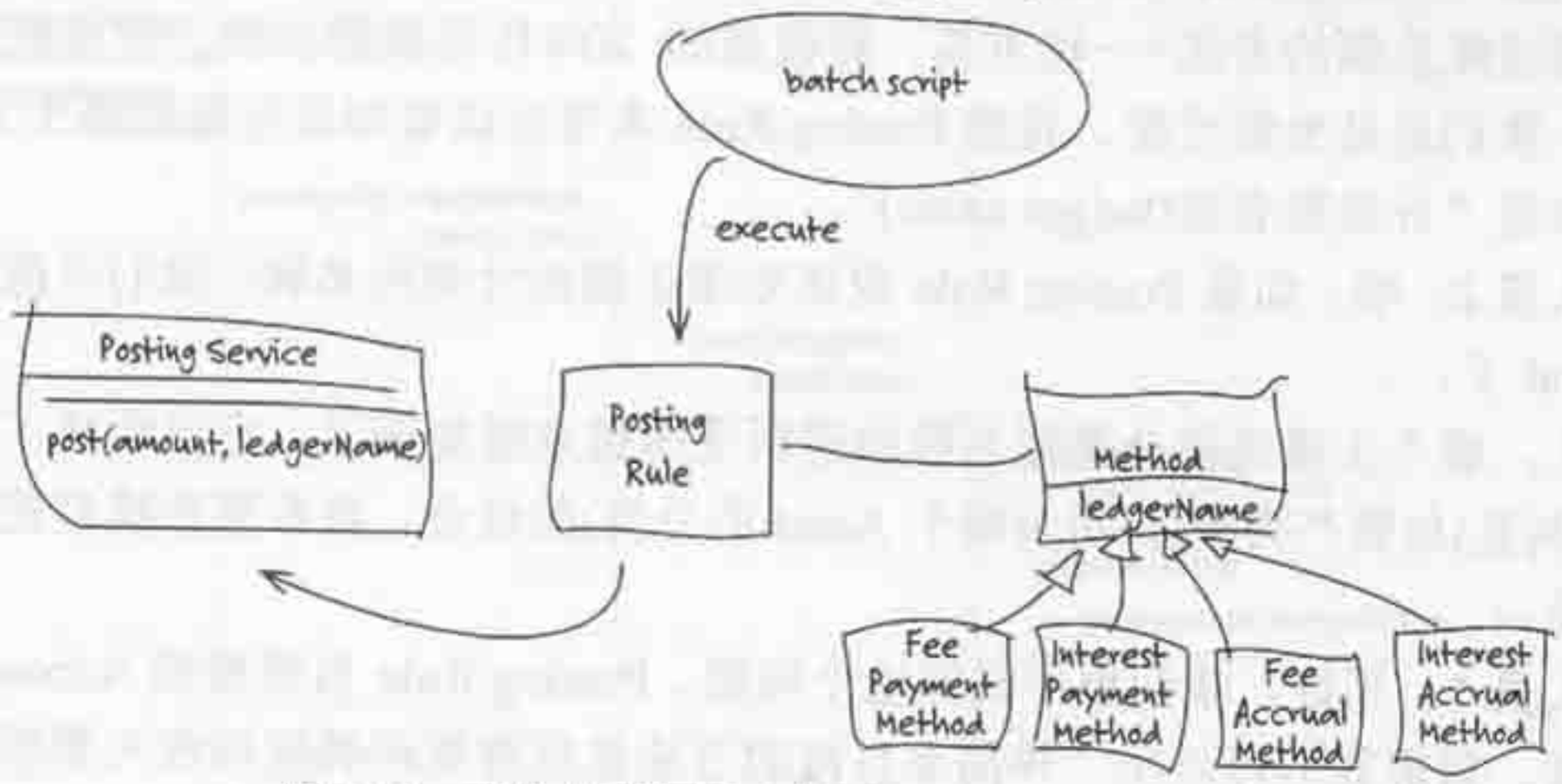


图 11-9　在批处理程序中使用 Posting Rule 的快照

开发人员 1：什么是 Posting Service?

开发人员 2：这是一个外观，它暴露记账程序中的 API，并将其呈现为一个服务。我实际上用它来简化批处理代码。它也提供了一个释意接口来过账到老式系统。

开发人员 1：有意思。那么，您准备对这些 Posting Rule 使用什么激活方式呢?

开发人员 2：我还没有想到那么远。

开发人员 1：我们可以对 Accrual 使用贪婪激活，因为它们实际上是由批处理通过 Asset 插入的。但是贪婪激活对于 Payment 行不通，因为 Payment 是在白天插入的。

开发人员 2：不管怎么说，我们都不希望把计算方法和批处理关联得那么紧。如果我们什么时候决定在另一个时机来触发利息计算，那事情就糟糕了。而且从概念上看，贪婪激活似乎有些不对。

开发人员 1：您这么说起来有点像“基于过账规则激活”。批处理通知每个需要执行的 Posting Rule，然后规则把合适的新 Entry 查找出来，再做其他的事情。这与您画出来的思路相当贴近。

开发人员 2：这样我们就避免了在批处理设计中产生一大堆依赖关系，它也不会失去控制了。听起来是正确的。

开发人员 1：我还有一点不太清楚，这些对象和 Account 及 Entry 是如何交互的？

开发人员 2：我也不太清楚。书中的例子在 Account 和 Posting Rule 之间创建直接链。这有某种合理性，但是我觉得它不太适合我们的情况。我们必须每次从数据中实例化这些对象，所以，要把 Account 和 Posting Rule 关联起来，就必须确定应该使用哪条规则。与此同时，Asset 对象知道每个 Account 的内容，因此也知道应用哪条规则。除此之外还有什么呢？

开发人员 1：我讨厌吹毛求疵，但是我觉得 Method 用得不正确。我想它在概念上是一个计算待过账总额的方法——比方说，将收益的 20%作为税额扣缴。但是我们的情况非常简单：我们总是全额过账。我想 Posting Rule 本身应该要知道过账到哪个 Account，也就是说知道“分类账名称(ledger name)”。

开发人员 2：哦，如果 Posting Rule 应该知道正确的分类账名称，我们可能根本就不需要 Method 了。

实际上，整个正确选择分类账名称的逻辑变得越来越复杂了。它已经成了收入类型(手续费或利息)与资产类型(公司为每个 Asset 的分类)的组合。我希望新模型能在这个地方有所帮助。

开发人员 1：那好，我们集中研究这个问题。Posting Rule 负责根据 Account 的属性选择分类帐。目前我们可以用一种简单直接的方法来处理资产类型和收入类型。将来您可以用一个对象模型来改进以便处理更复杂的情况。

开发人员 2：这个问题还要多想想。让我仔细考虑一下，把模式再看一遍，应该会获得新的领悟的。明天下午我能就这个问题再和您谈一下吗？

在接下来的几天里，两个开发人员得到一个模型，并对代码进行了重构，如图 11-10、图 11-11 所示。这使得批处理只是简单地遍历各个 Asset，向每个 Asset 发送一些自解析的消息，然后提交数据库事务。复杂性转移到了领域层，领域层的对象模型使问题既明显又更加抽象。

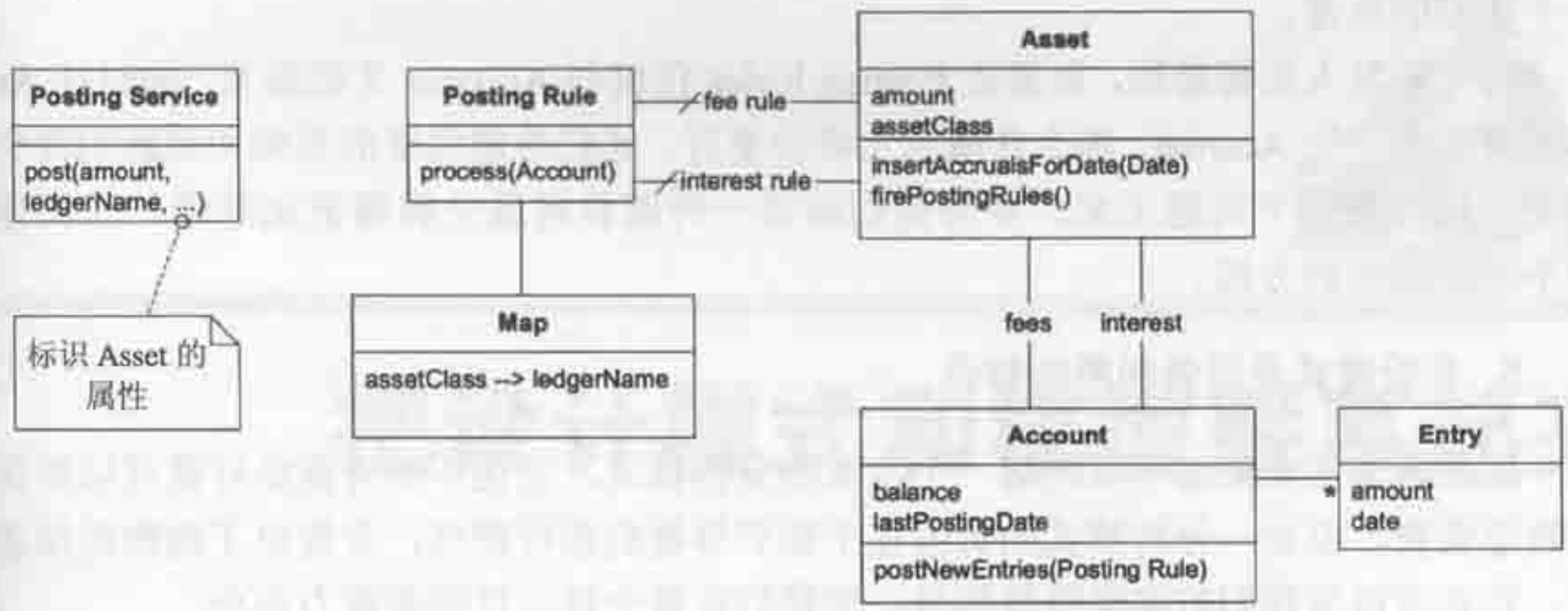


图 11-10　包含 Posting Rule 的类图

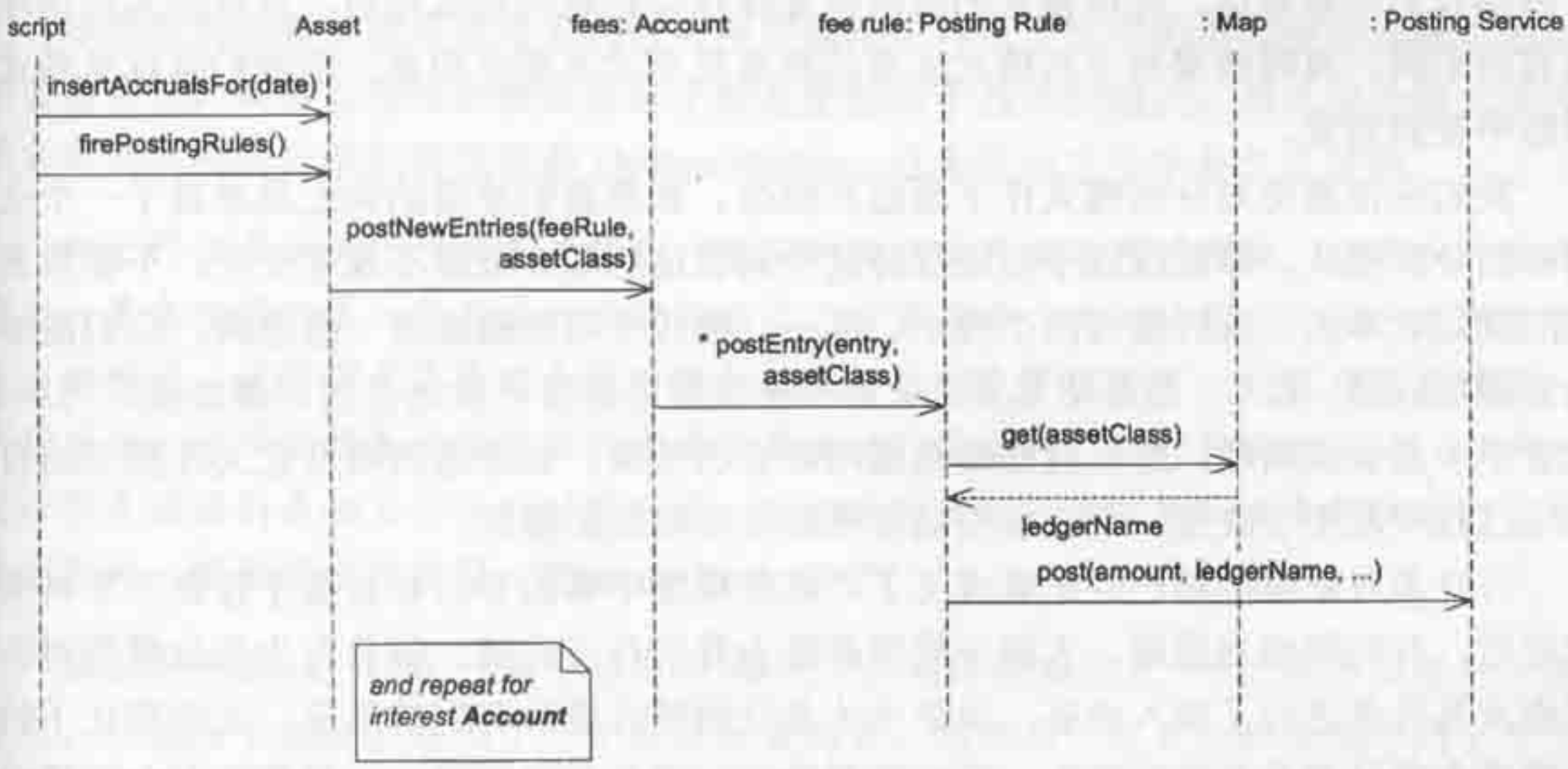


图 11-11　采用规则激活的顺序图

在细节上，开发人员们创建的模型和 *Analysis Patterns* 中给出的已经相去甚远了，但是他们觉得概念的本质还是保留在那里。他们还有些地方不太满意，那就是在 Posting Rule 的选择中会牵涉到 Asset。之所以会这样，是因为 Asset 知道每个 Account(手续费或利息)的基本信息，同时它也是脚本的一个很自然的访问点。如果让规则对象直接与 Account 关联起来，那么在每次实例化这些对象(每次运行脚本)时，都会要求 Asset 对象参与协作。他们没有这样做，而是让 Asset 对象通过一个 SINGLETON 来查询这两个相关的规则，然后把规则传递给合适的 Account。看起来它能使代码直接得多，因此这是

一个实用的决策。

两个开发人员都感到，如果让 Posting Rules 仅仅与 Account 关联起来，同时让 Asset 仍然聚焦于产生 Accrual，那么在概念上将会更好。他们希望后续的重构和理解的深化可以使他们回到这个问题上来，并为他们提供一种能将问题分解得更加清晰，而又使代码不失明显性的方法。

3. 分析模式是可供利用的知识

如果您非常幸运地可以利用一个现成的分析模式，它也不太可能恰好就可以解决您的特定需要。但是，分析模式的价值在于能引导我们进行研究，并提供了清晰的抽象词汇。它还可以为我们的实现提供指导，使我们在整个设计过程中省力不少。

所有分析模式都要融入到我们的大脑中来，协助我们消化知识，向更深层理解重构，并激励我们不断前进。分析模式的应用结果往往与记载的形式相似，只是因其应用环境而有所不同。有时结果与分析模式本身的联系甚至并不那么明显，但我们可以从模式的思想中受到启发。

我们应该避免对分析模式作下面这种修改。如果我们使用的词汇是来自于一个众所周知的分析模式，那就应该小心地保持这个词表达的基本概念不发生变化，不管其表面上的变化有多大。这样做有两个理由。其一，模式中可能蕴涵着一些理解，它们能够帮助您避免问题。其二，也是更重要的，如果您使通用语言中所包含的词都能被广为理解，或者至少是含义清晰，那么就能提高通用语言的效果。如果您的模型定义在模型的自然演化过程中发生了改变，那么模型名称也必须作相应的修改。

有许多对象模型都已经记载成文了，这些模型中有的专门针对某个行业中某种类型的应用，有的则相当通用。大部分模型都能为我们打开思路，但只有少部分模型对各种权衡及其后果进行了深入论证，而那些才是分析模式最为有用的部分。这些精化了的分析模式大部分都是非常珍贵的，它们能帮助我们避免重复开发。虽然我们不太可能对分析模式作全面的分类(如果有的话，我会觉得很惊讶)，但是针对特定行业的分类还是有可能的。我们可以广泛地分享那些跨应用的领域中出现的模式。

这种将知识组织起来重新应用的方式与通过框架或组件实现代码的重用是完全不同的，但是二者都能为我们提供一些较为隐蔽的思路。模型(甚至通用框架)都是作为一个整体来发挥作用的，而分析则是一个模型片断的工具包。分析模式着眼于最关键、最困难的决策，为我们阐明各种选择和方法。分析模式能对以后才出现的结果进行预测，这些结果如果靠我们自己来发现的话，代价可能会非常高昂。

第12章

把设计模式和模型联系起来

本书前面探索的模式都是专门针对模型驱动设计上下文中的领域模型问题。然而，到目前为止，许多已公布的模式实际上都是更着眼于技术问题的。那么设计模式和领域模式有何区别呢？我们首先看看 *Design Patterns* 这本开山之作中是怎么说的。

对于什么是模式、什么不是模式，不同的观点会产生不同的理解。这个人的模式在那个人看来可能只是一个基本的构造块而已。在本书中，我们将集中讨论一定抽象层次之上的模式。设计模式不是用来设计诸如链表、散列表等这些可以封装为类直接供人使用的东西，也不能用来为整个应用或者子系统进行复杂的、针对领域的设计。本书中的设计模式描述的是相互作用的对象和类，我们通过定制这些对象和类来解决特定上下文中遇到的一般性的设计问题。[Gamma et al. 1995]

Design Patterns 中的一些(不是所有)模式可以用作领域模式，不过需要调整一下重点才行。*Design Patterns* 给出了一个设计元素列表，这些设计元素能解决在不同上下文中遇到的常见问题。设计模式的动机以及模式本身都是用纯粹的技术术语进行描述的。但是，有一部分设计元素可以应用到更宽上下文的领域建模和设计中，因为它们所代表的一般性概念在许多领域中都会出现。

除了 *Design Patterns* 中给出的设计模式，这几年还陆续出现了许多其他的技术性设计模式，其中有些反映了领域中出现的深层概念。吸取这些设计模式的经验将大有裨益。为了在领域驱动设计中应用那些模式，我们必须同时从两个层面上来看待它们。在一个层面上，它们是代码中的技术设计模式；在另一个层面上，它们又是模型中的概念模式。

我们将用 Design Patterns 中的一个模式作为示例，说明如何把我们认为是设计模式的模式应用到领域建模中来。这个示例也能阐明技术设计模式和领域模式之间的区别所在。Composite(组合)和 Strategy(策略)演示了我们可以换一种思路，用某些经典的

设计模式来解决领域问题。

12.1 策略

有时领域模型中包含的流程(Process)并不是出自于技术性的考虑，而是在问题域中确实具有实际意义。当必须提供多种可供选择的流程时，如何适当选择就成了一个复杂的问题，再加上这些流程本身就很复杂，因此会导致事情失去控制。

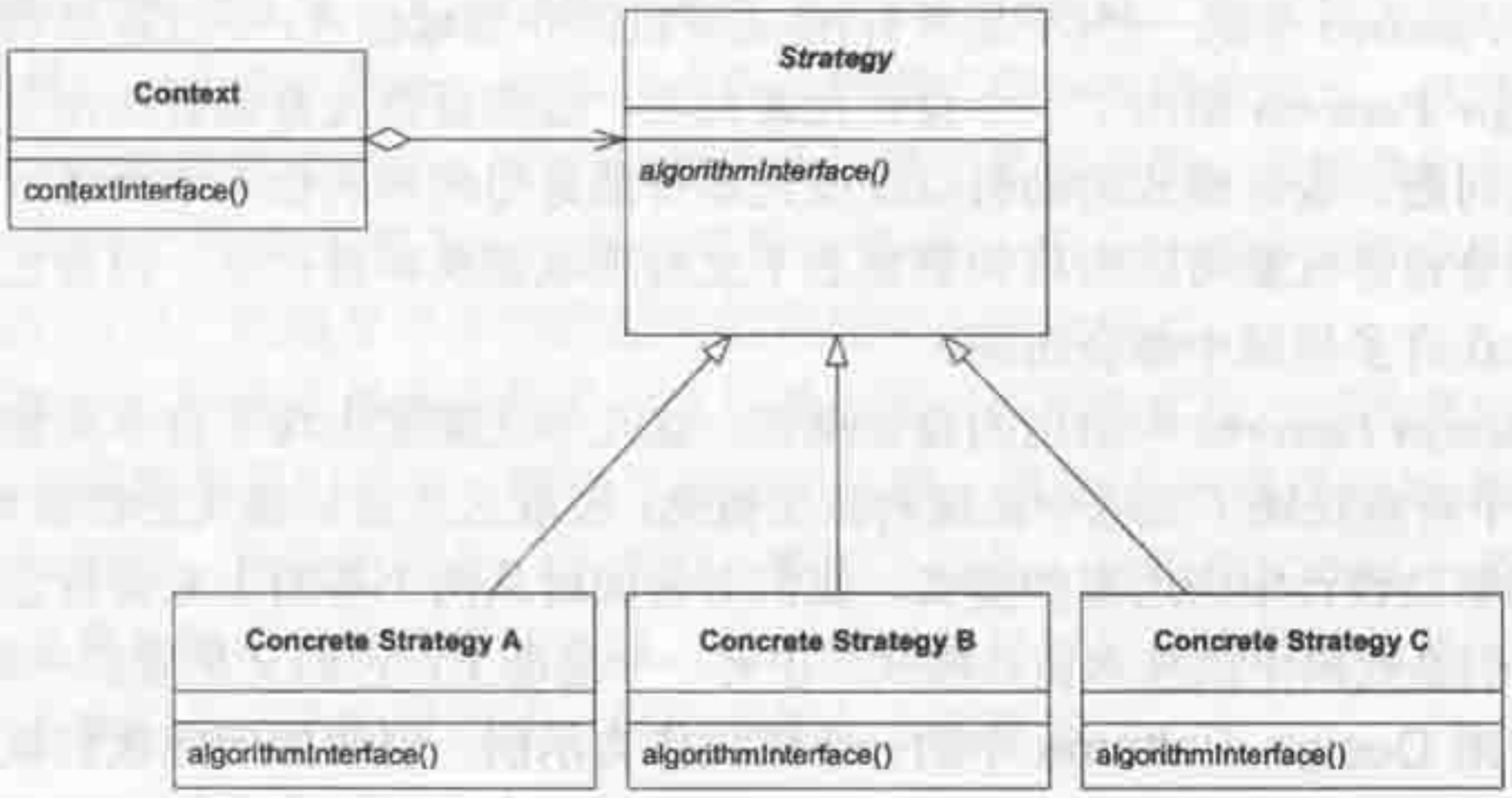
在对流程建模时，我们常常认为合理的流程不止一种。而一旦我们开始把各种选择描述出来，流程的定义就会变得既笨拙又复杂。当与行为的其他部分混合时，我们实际作出的行为选择会变得模糊不清。

我们希望把那些变体从流程的主体概念中分离出来，这样就能把主要流程和可选流程区分得更清楚了。软件设计阵营中确立已久的策略模式正是用来处理这种问题的，不过其着眼点是技术性的。在本节中，策略将被视为模型中的一个概念，并通过代码实现反映出来。与分离主要流程和可选流程一样，将流程中的易变部分与稳定部分解耦也是同样的问题。

因此：

将流程中易变的部分分离为模型中一个独立的“策略”对象，将规则及其控制的行为分离出来。然后，依据策略设计模式来实现这种规则或者可替换的流程。策略对象的不同版本代表了完成流程的不同方法。

根据传统视角将策略视为设计模式时，焦点放在替换不同算法的能力上。但是，将其视为领域模式时，焦点放在表达概念的能力上(通常描述流程或者策略规则)。



将一组算法分别封装起来，使之可以互相交换。策略使得算法的变化可以独立于使用它的客户[Gamma et al. 1995]

示例：航线搜索策略

我们把一个 Route Specification 传给 Routing Service，让它按规格的要求构造出详细的 Itinerary，如图 12-1 所示。这个服务是一个优化引擎，可以用来搜索最快速或者费用最低的路线。

上面的设置看起来不错，但是仔细查看航线代码就会发现，每处计算中都有条件判断，到处都是按最迅速或最廉价进行决策的代码。如果要加入新的规格来从两种航线之间作出更加精细的选择，那还会出现更大的麻烦。

一种方法是把那些调节参数分离到策略中，使之显式地描述出来，然后作为一个参数传给 Routing Service。

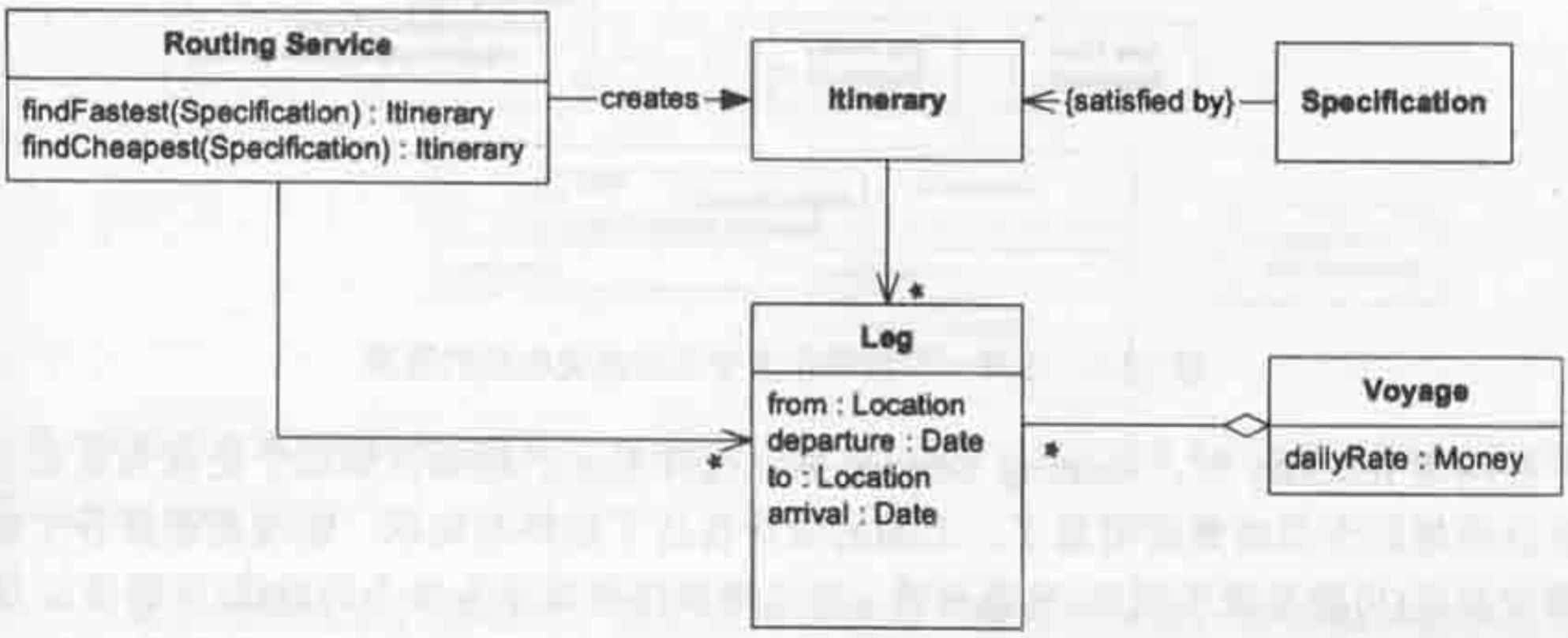


图 12-1　包含可选方法的服务接口需要条件逻辑

现在 Routing Service 可以用同一种方法来处理所有请求了，而无需再执行条件判断。它搜索一系列低开销的 Leg，航段的开销是由 Leg Magnitude Policy 来计算的。

这个设计具有 *Design Patterns* 中策略模式所具有的优点。从系统的通用性和灵活性这个层面来说，它使我们可以用策略来控制 Routing Service，还可以通过载入合适的 Leg Magnitude Policy 来进行扩充。图 12-2 演示的策略(最迅速或最廉价)只是两种最明显的策略，很可能还有速度和费用一起权衡考虑的策略。其他的因素也是完全有可能的，例如优先使用公司自身的运输能力来进行货物预订，而不是外包给其他运输公司。不使用策略也许同样能够实现这些修改，但是它们的逻辑会扰乱 Routing Service 的内部结构，也会使其接口变得庞大。解耦可以让 Routing Service 更加清晰而且易于测试。

第Ⅲ部分
面向更深层理解的重构

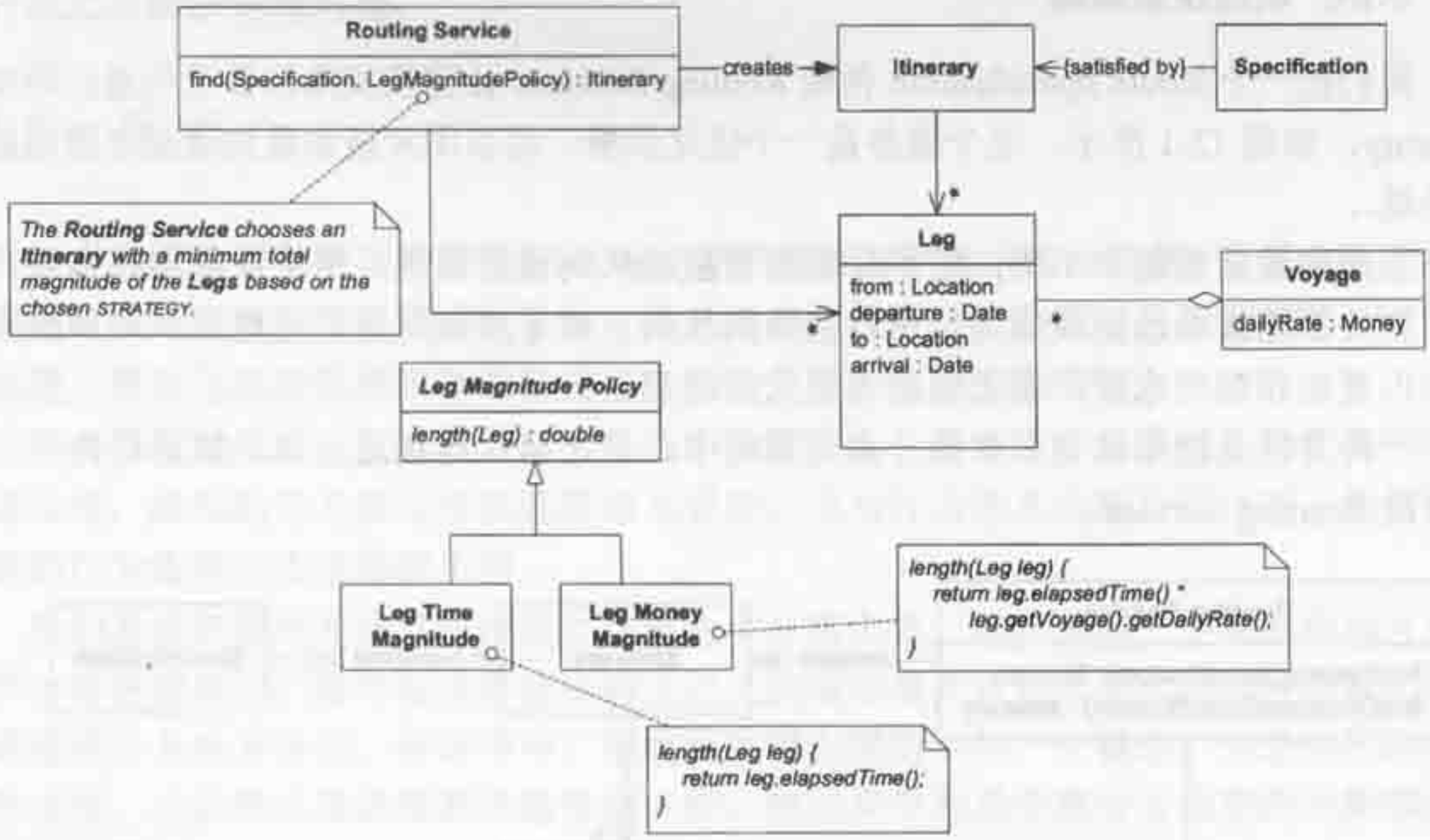


图 12-2　选择一个策略作为变元来决定航线的选择

在构造 Itinerary 时，Routing Service 据以选择 Leg 的规则在领域中是极为重要的，现在这些规则变得清楚而明显了。上面的设计传达了这样的知识：航线是根据各个航段的特定属性(可能是派生属性)来选择的，而这种属性被表示为单个的数值(开销值)。这使得我们可以用领域语言构造一个简单的声明，来定义 Routing Service 的行为：Routing Service 根据所选策略选择一个具有最小的 Leg 总开销值的 Itinerary。

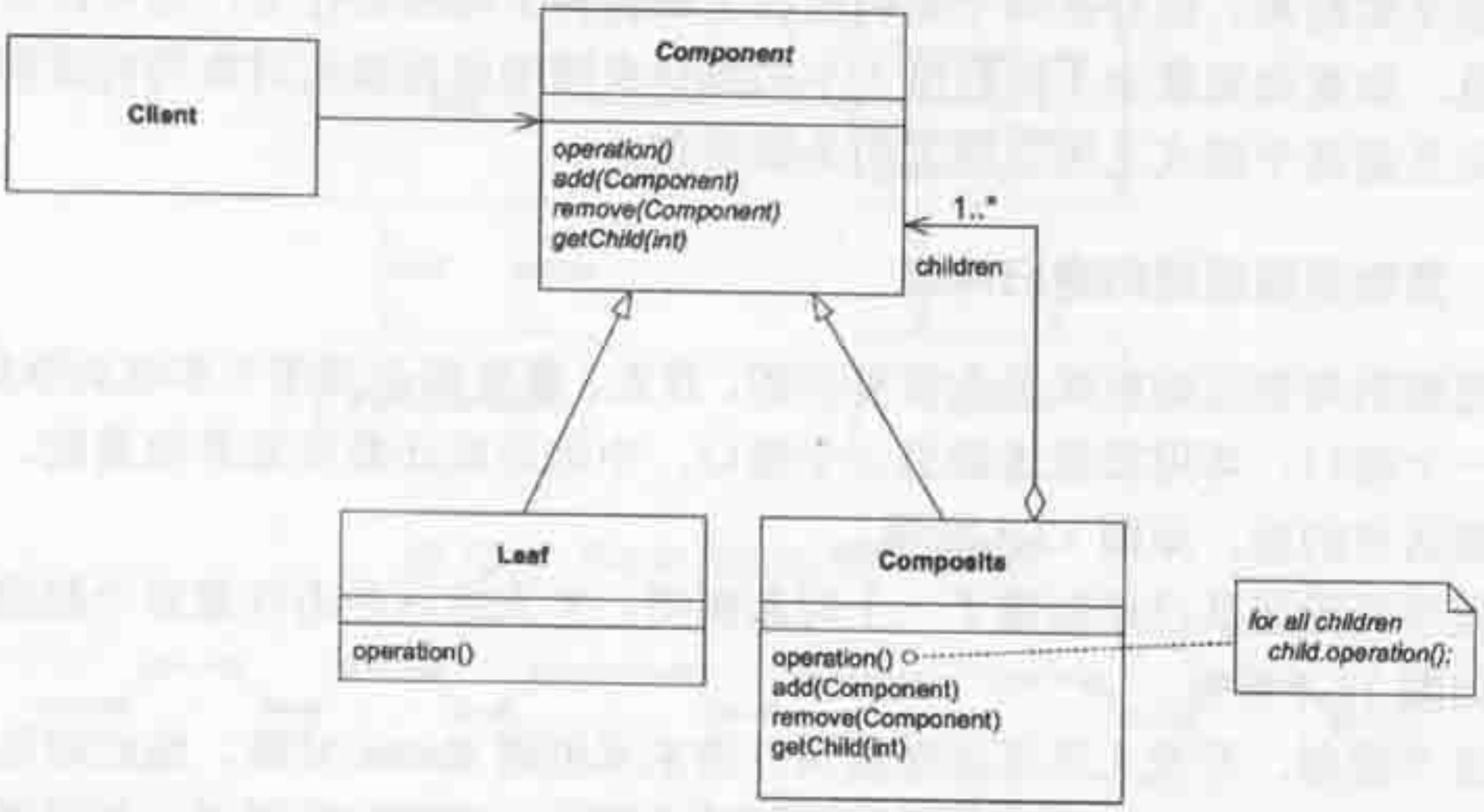
注意，这里的讨论假设了 Routing Service 在搜索 Itinerary 时实际上对 Leg 进行了计算。这种方法在概念上很简捷，将它作为一种原型实现也许合理，但是其效率可能低得令人无法接受。这个应用将在第 14 章“维护模型完整性”中重新讨论，在那里我们将使用同一个接口，但是 Routing Service 的实现完全不同。

当把技术设计模式应用到领域层时，我们必须多加入一个动机，或者说另一层含义。当策略与实际的业务策略对应得很好时，它就不再仅仅是一种有用的实现技术了(当然它作为实现技术也是很有价值的)。

设计模式的效果也是完全适用于领域层的。例如，在 *Design Patterns* 中，Gamma 等指出客户必须知道各个不同的策略，而这也是建模时的一个关注点。一个纯粹实现上的关注点是策略会增加系统中对象的个数。如果这是一个问题的话，我们可以把策略实现为一个无状态对象，通过上下文共享来降低开销。*Design Patterns* 中关于实现方法的

详尽讨论在这里同样适用，因为我们仍在使用策略。虽然我们的动机有些不同，会影响到某些问题的决策，但是设计模式中所蕴涵的经验仍然可以为我们所用。

12.2 组合



将对象组织为树形结构来描述部分-整体的层次关系。组合使客户对单个对象和对象组合作同样的处理[Gamma et al. 1995]

当对复杂的领域进行建模时，我们常常会遇到一种重要对象，它由多个部分组合而成，而那些部分又由部分的部分组合而成，部分的部分又由部分的部分的部分组合而成——这种组合有时甚至会任意深度地嵌套下去。在某些领域中，每个层次在概念上可能是独立的，但是在其他的情况下，我们可以认为部分和整体是同一种事物，只不过小一些而已。

如果这种嵌套容器关系没有在模型中反映出来，那我们就不得不在每个层次上复制它们的公共行为，同时嵌套结构也非常僵硬(例如，容器往往不能包含同一层次上的其他容器，而嵌套层数也是固定的)。客户必须通过不同的接口来与不同的层次打交道，即使它并不关心它们在概念上有何区别。遍历树形结构来收集信息变得非常复杂。

在领域层中应用任何设计模式时，第一个关注点应该是这个模式的思想是否能与领域概念真正很好地吻合起来。能递归地遍历一些对象可能会很方便，但是它们确实存在整体-部分关系吗？您有没有找到一种方法，能把所有部分都抽象为同一种概念类型？如果有，那么组合可以使模型的这些方面变得更加清晰，同时能使您能深入下去，慎重仔细地考虑设计模式中的设计和实现问题。

因此：

定义一个能包含组合中所有成员的抽象类型。在容器中实现信息查询方法时，根据其内容的集合来返回信息；在“叶”节点中实现那些方法时，则根据其自身的值来返回信息。客户只与抽象类型打交道，而无需区分元素是叶子还是容器。

从结构层面上看，组合是一个相对明显的模式，但是设计人员往往没有把这个模式的操作层面充实起来。组合在每个结构层次上都提供了同样的行为，也可以查询一些有意义的信息，如查询对象中不同粒度大小的部分来透明地反映出对象的构成情况。这种严格对称关系是这个模式之所以强大的关键所在。

示例：货物运输航线的递归构成

一个完整的货物运输航线是非常复杂的。首先，集装箱必须用卡车送到铁路终点站，然后送到一个港口，再用货轮送到另一个港口，中途可能还要转装其他货轮，最后通过陆上运输到达目的地，如图 12-3 所示。

应用程序开发团队已经创建了一个对象模型，来表达这些由任意多个航段串接而成的航线，如图 12-4 所示。

使用这个模型，开发人员可以根据预订请求来创建 Route 对象。他们对这些 Leg 进行处理来生成操作计划，即货物的装卸和转运步骤。这时他们发现了一些问题。

开发人员原来一直以为航线是任意多个航段串接而成的，如图 12-5 所示，航段与航段之间没有区别。

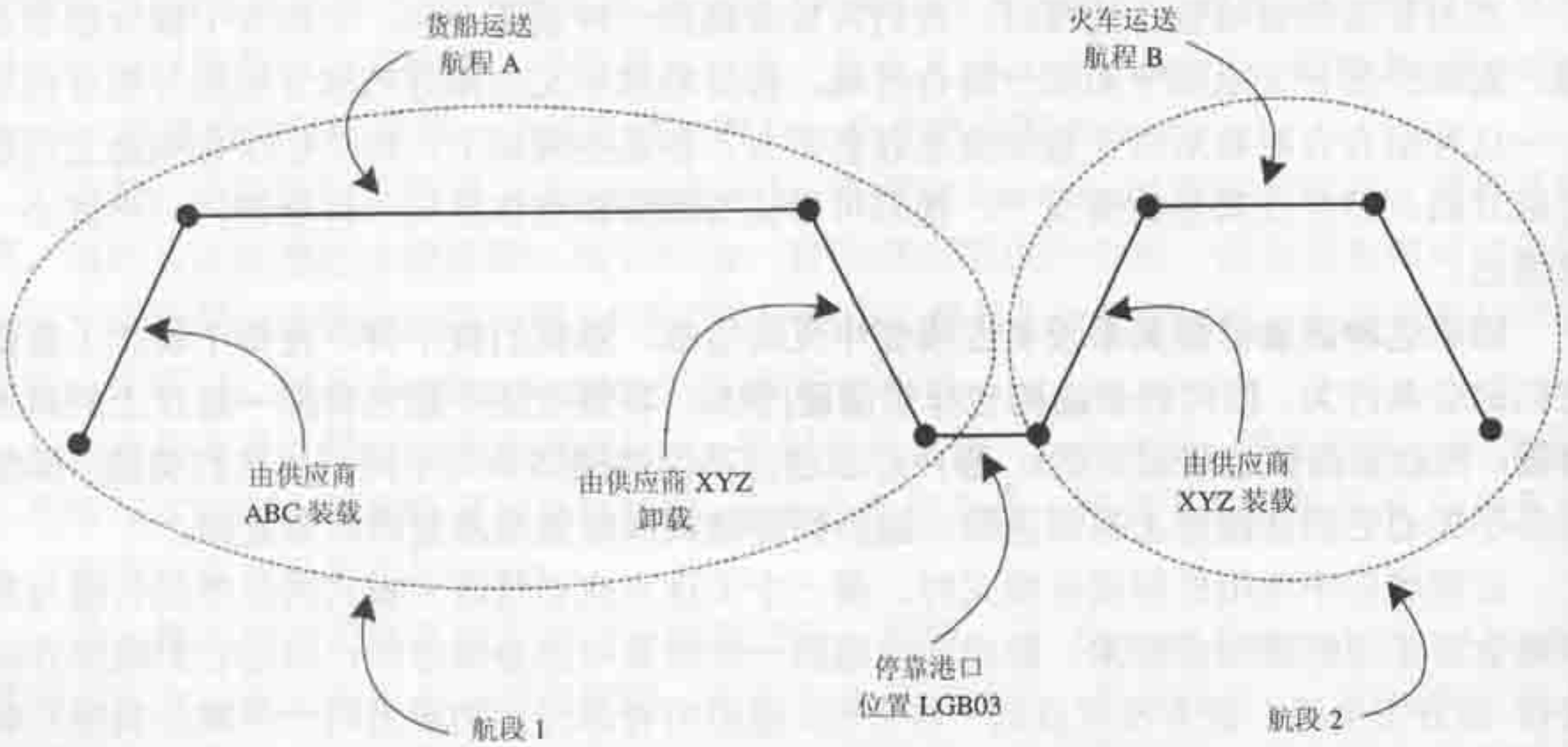


图 12-3　由“航段”构成“航线”的图解

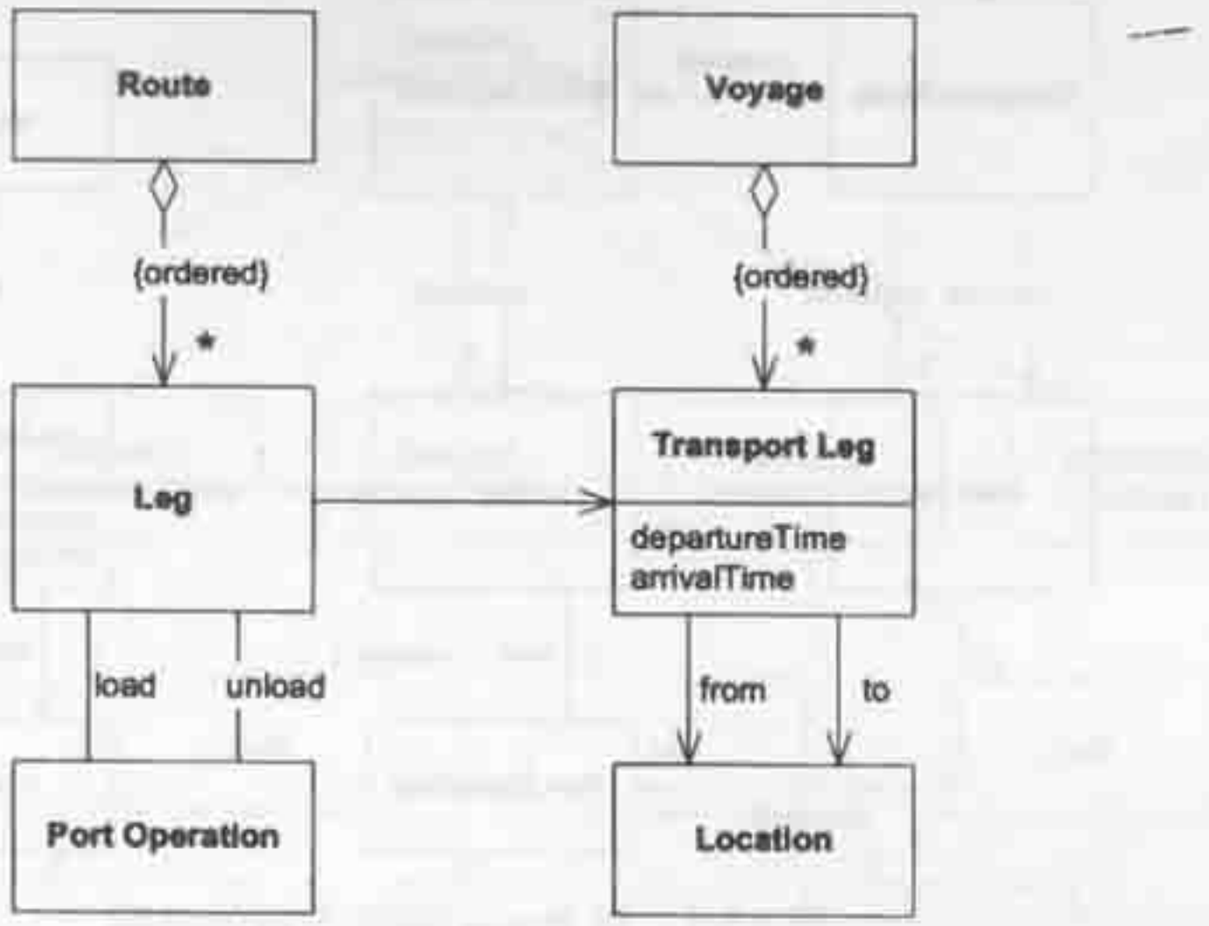


图 12-4　Route 由 Leg 构成的类图

图 12-5　开发人员头脑中的航线概念

但是领域专家却把航线看成是由 5 个逻辑区段(segment)串接而成的，如图 12-6 所示。

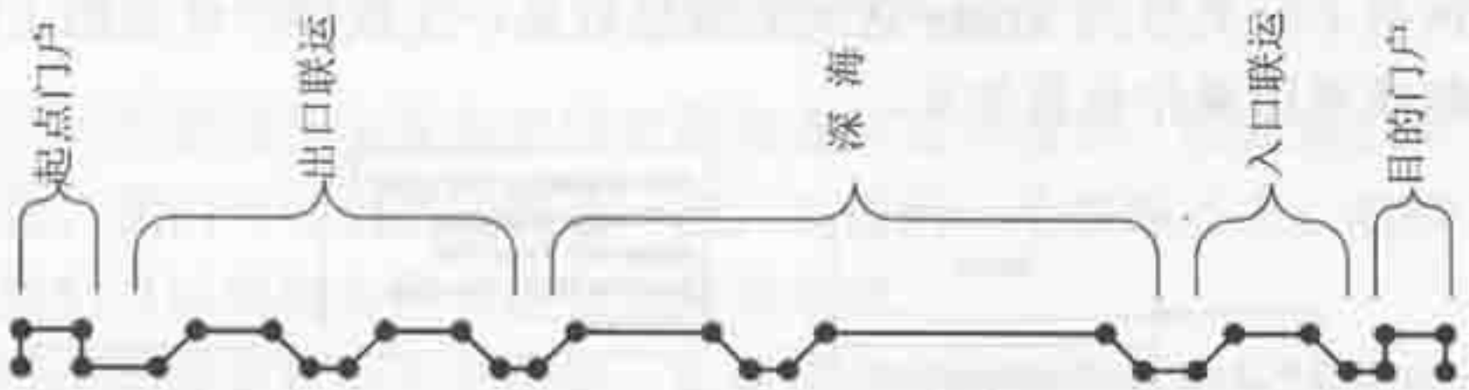


图 12-6　业务专家头脑中的航线概念

除此之外，这些子航线还可能是由不同的人在不同的时间内进行计划的，因此不能把它们一视同仁地看待。更深入地检查后发现，“门户航段(door leg)”与其他航段大不相同，它可能包括在当地雇佣卡车运输，甚至客户拖运，而不像铁路和航运那样有详尽的调度计划。

为了反映这些区段的不同之处，对象模型开始变得复杂起来，如图 12-7 所示。

这个模型在结构上并不坏，但是它不能用统一的方式来处理操作计划，因此实现代码(甚至行为的描述)也会变得相当复杂。其他复杂问题也开始浮现出来。遍历任一个航线都需要对不同类型的对象进行多次收集。

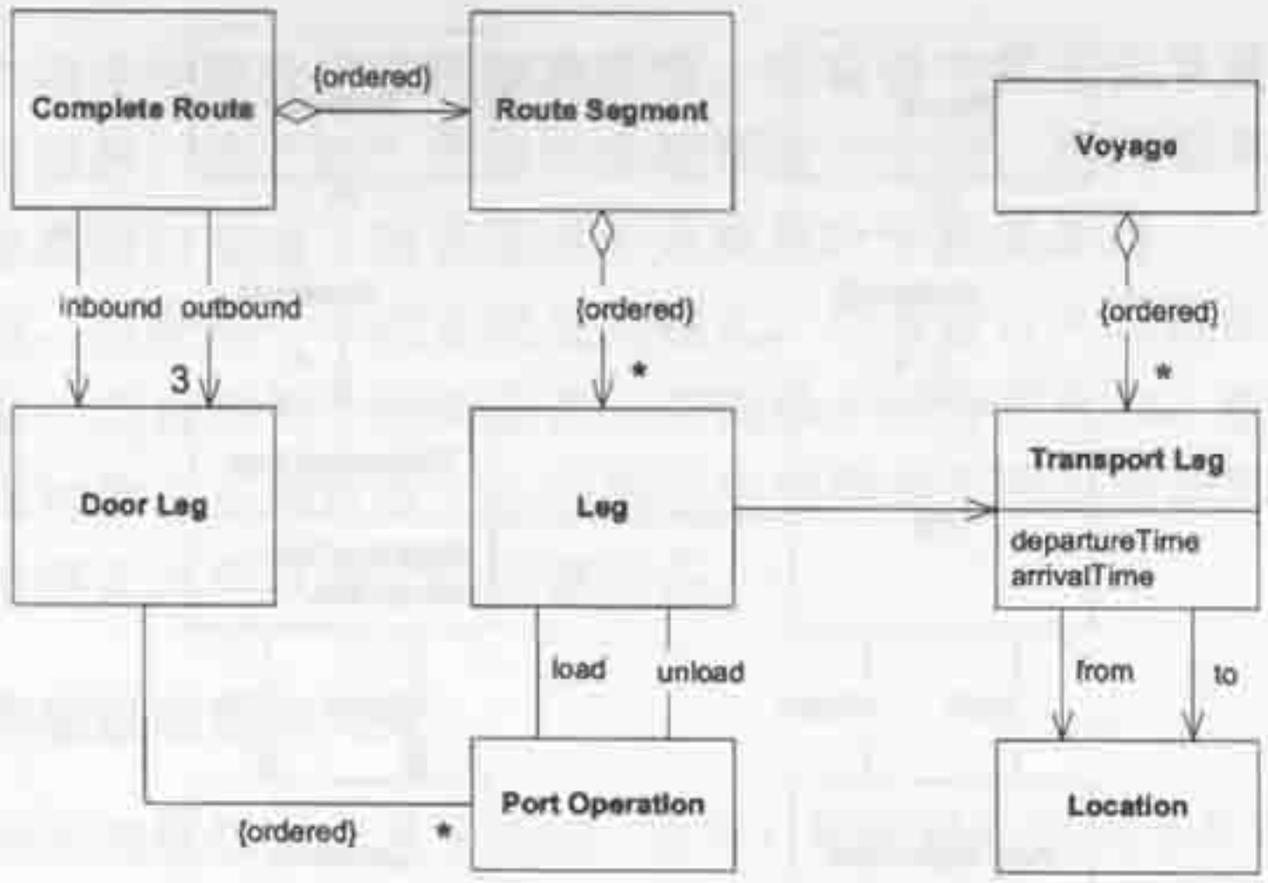


图 12-7　将 Route 细化之后的类图

运用组合。对于某些客户来说，这能使它们使用这个构造在不同的层次上作统一处理，因为航线是由(子)航线构成的。在概念上，这种看法是非常可靠的。Route 的每个层次都是集装箱从一个点移动到另一个点，一直分解到单个的航段(见图 12-8)。

现在的这个静态类图和前面那个相比，并没有告诉我们多少关于 door leg 和其他区段如何吻合起来的信息。但是这个模型不仅仅是一个静态类图。我们可以通过其他的图(图 12-9)来演示航线组合的信息，也可以通过代码来说明这一点(现在代码简单多了)。这个模型捕捉了所有不同类型的 Route 之间的深层联系。生成操作计划的工作重新变得简单了，其他的航线遍历操作也是如此。

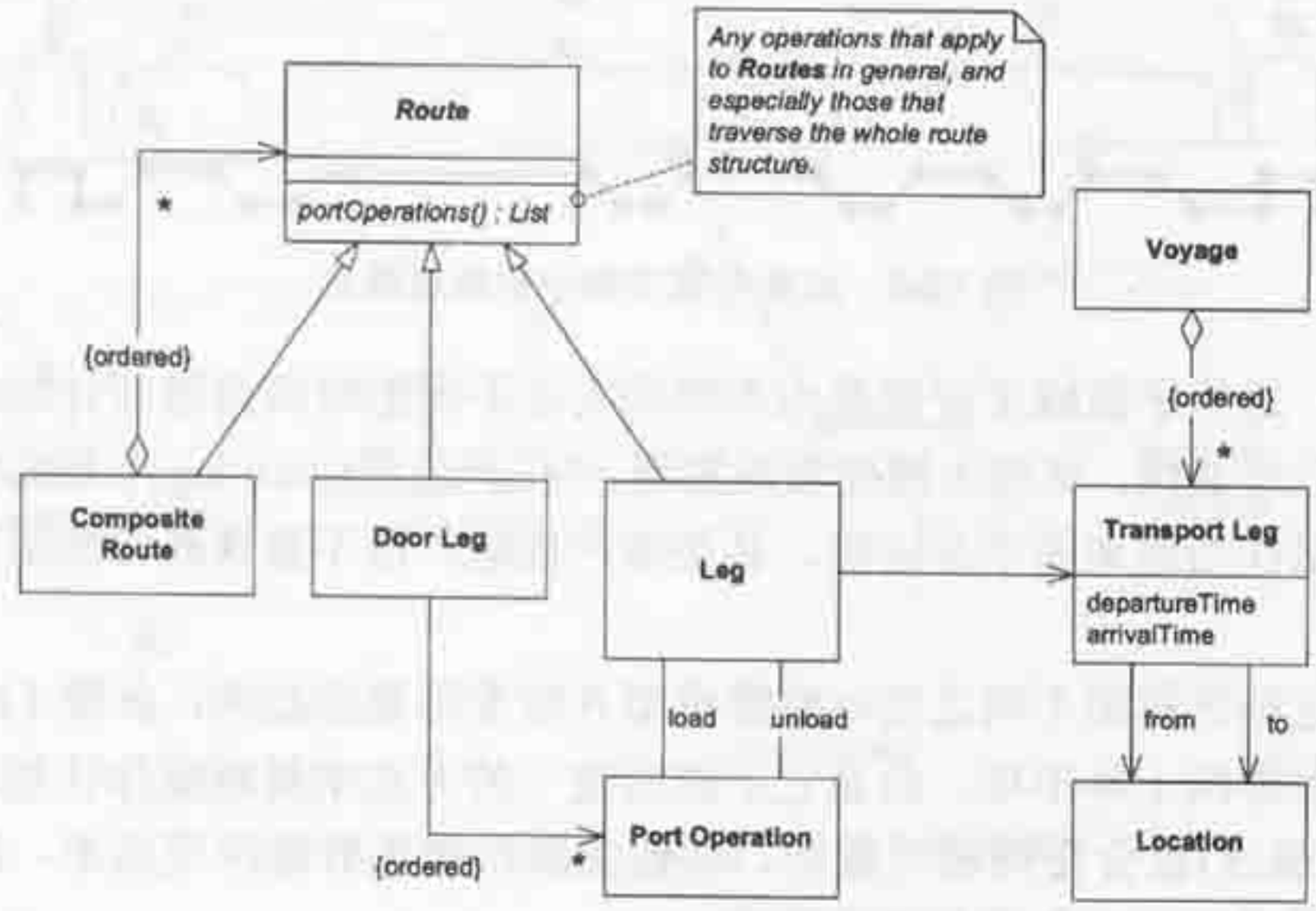


图 12-8　一个使用组合的类图

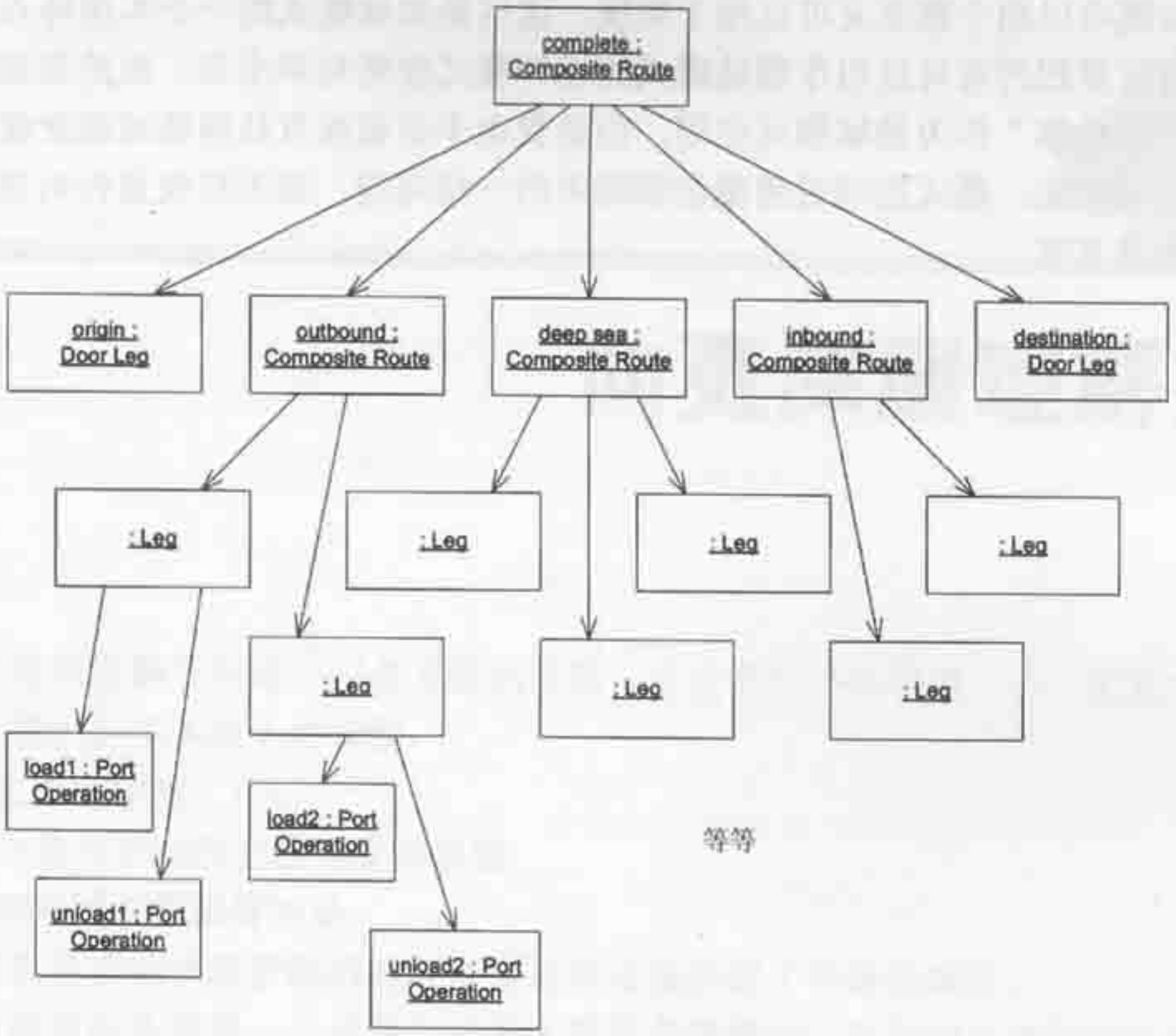


图 12-9　一个完整的 Route 的示例

由于两点之间的航线是由其他航线首尾相连拼接而成的，因此我们可以在不同细节层次上来实现它。我们可以把航线截去一段，再连到一个新端点上。航线可以任意地进行嵌套，因此我们可以获得各种可能会有用的选择。

当然，我们还不需要有那么多选择。如果现在并不需要这些航线区段和单独的 door leg，那么我们不用组合也能很好地完成任务。设计模式应该只在需要的时候才去使用。

12.3　为什么不用 Flyweight?

在前面(第 5 章中)我们提到过 Flyweight(享元)。您可能会认为享元也是一个可以应用于领域建模的模式的例子。事实上，享元是一个不适于领域建模的设计模式的例子。

如果我们需要多次使用同一批值对象，就像在住宅设计中电源插座的例子，那么把它们实现为享元是有意义的。享元是一种适用于值对象，但不适用于实体的实现选择。相比之下，组合不是一种实现选择，而是表示一个概念对象是由其他概念对象组成的。

因此，组合既可以用于模型又可以用于实现，这也是领域模式的一个本质特点。

我没有打算把所有可以用作领域模式的设计模式收集列举出来。虽然我想不出一个例子来把“解释器”作为领域模式使用，但是我也不会说没有任何领域概念能与之相吻合。惟一的要求是，模式应该说明概念领域中的一些问题，而不仅仅是针对某个技术问题的技术解决方案。

第13章

向更深层理解重构

向更深层理解重构是一个多方面的过程。这里我们不妨稍停一会，把其中的要点整理一下。我们必须关注 3 件事情：

- 扎根到领域之中；
- 总是用不同的方法来看待事物；
- 与领域专家保持对话。

在我们寻求领域理解的同时，也为重构过程搭起了更宽的舞台。

典型的重构场景是一个或两个开发人员坐在键盘前，认识到某些代码还可以优化，然后就立即对代码进行修改(当然还要用单元测试来验证修改的结果)。这种重构应该不断进行，但它还不是全部。

前面的 5 章扩展了重构的概念，使之超越了传统的微重构方法。

13.1 发起重构

向更深层理解重构可以有许多种开始的方法。对代码中的问题(某种复杂性或不协调性)作出响应就可能引起重构。开发人员感觉到问题的根源出在领域模型中——也许是遗漏了某个概念，也许某些关系是错误的。这种问题不能通过标准的代码变换来解决。

这里对于重构的看法与传统的重构有所不同。虽然代码看起来很整洁，但是如果模型所使用的语言看起来与领域专家的语言不搭边，或者新出现的需求不能自然地融入模型的话，我们也需要进行重构。学习也可以引起重构，因为当开发人员获得了更深入的理解时，他就会发现自己可以使模型变得更加明晰或有用。

找出问题的所在往往是最难，也是最不确定的部分。找到了以后，开发人员就可以

系统地搜索出组成新模型的元素。他们可以与同事和领域专家进行头脑风暴，也可以从分析模式或设计模式中吸取系统化的知识。

13.2 探索团队

无论问题是因何而起，下一步就是要找到精化模型的方法，使之能够更清晰自然地进行交流。有时候，这些修改可能都不大而且很明显，几个小时就能搞定，和传统的重构差不多。但有时候，为了获得一个新的模型，我们可能需要重构更多的次数，涉及到更多的人。

修改的发起者要选出几个其他开发人员，来组成一个探索团队。探索团队的成员应该善于找出那一类问题的答案，熟悉领域的知识，或者具有相当的建模技巧。如果有一些难以捉摸的问题，他们还应该保证领域专家也参与其中。四五个人一组一起到会议室或咖啡厅中讨论半个小时到一个半小时，画一些 UML 草图，然后把那些对象放到场景中进行走查。他们还要确保问题专家能够理解他们的模型，并认为那是一个有用的模型。如果他们发现了一些令人满意的结果，就可以回去开始编码了。他们也可能会决定用几天时间把问题再深入考虑一下，然后再回来看有没有新的思路。几天以后，小组重新集合起来，再继续他们的讨论。这一次，经过了前面几天的思考，他们会更加自信，也会获得一些结论。然后，他们回到计算机前，把新的设计编码出来。

有一些方法可以保证这个过程的效率：

- **自主**。为了探索一个设计问题，这种小团队可以随时组织起来，运作几天，然后再解散。不需要长期的、精致的组织结构。
- **界定范围，适度休息**。在几天的时间内开两到三次短会，应该就可以得到一个值得一试的设计。想把设计硬拽出来是于事无补的。如果陷入僵局，那可能是一次考虑得太多。选取一个更小的设计方面来集中解决。
- **运用通用语言**。有其他团队成员(特别是问题专家)一起参与的头脑风暴是运用和精化通用语言的一个好机会。运用通用语言能使之更加精化，最终由原来的开发人员拿回去并形式化到代码中。

本书的前面章节给出了几个对话记录，其中开发人员和领域专家一起来寻求更好的模型。完美的头脑风暴会议是动态的、非结构化的，而且具有惊人的效率。

13.3 前期工作

重复开发有时候是没有必要的。通过头脑风暴，我们可以找到遗漏的概念和更好的模型，对各方面的思想兼收并蓄，并与已有的知识结合起来。随着这些知识的不断消化，我们就能找到解决当前问题的答案了。

我们可以从书籍和领域本身的其他知识来源获得想法。虽然有的领域可能没有建立适于运行软件的模型，但人们可能已经将其中的概念组织起来了，并提供了一些有用的抽象。通过充分的知识消化过程，我们可以理解得更快、更深入，而且这些知识对于领域专家来说也会更加熟悉。

有时我们还可以从分析模式中吸取其他人的经验。分析模式也能在一定程度上帮助我们理解领域，但是它是特别针对于软件开发的，因此在应用分析模式的时候，应该让它与我们在领域中实现软件的经验直接结合起来。分析模式可以为我们提供微妙的模型概念，帮助我们避免许多错误。但是它们和菜谱不同，不是现成的解决方案，而是知识消化过程的一种原料。

随着各个片断相互吻合起来，我们就必须同时考虑模型和设计的关注点了。同样，这并不总是意味着所有东西都要从头做起。当一个设计模式既符合实现需要，又符合模型概念时，我们往往可以用它来解决领域层中的问题。

类似地，如果一个通用的形式化系统(如算术或谓词逻辑)与领域的某个部分相适应，我们也可以将那个部分分离出来，然后用它来改写形式系统的规则。这可以构造出非常紧凑而且易于理解的模型。

13.4 针对开发人员设计

软件不仅仅是针对用户的，它也是针对开发人员的。开发人员必须把代码与系统的其他部分整合起来。在一个迭代过程中，开发人员反复修改代码，通过向更深层理解重构来获得柔性设计，同时又从柔性设计中获益。

柔性设计能用来交流它的意图。这种设计使得运行代码的效果可以很容易地预测出来——因此预测修改代码的影响也很容易。柔性设计还有助于限制大脑过载，这主要是因为它减少了依赖和副作用。领域的深层模型是柔性设计的基础，只有在那些对于用户最为关键的地方，深层模型才提供了详细的细节。这使得柔性设计在那些需要频繁修改的地方非常灵活，而在其他地方又不失简单。

13.5 时机选择

如果您想在周全地证明了其合理性之后才动手修改，那么您就会等很久。项目的开销也已经很大了，而修改越是推迟就越是难以完成，因为目标代码会变得更加庞大，与其他的代码也牵涉得更深。

持续重构已经被视为一种“最佳实践”，但大多数项目团队仍然对于重构过于谨慎。他们看到修改代码会有风险，也需要开发人员投入时间；但是他们没有看到的是，维持一个糟糕设计会有更大的风险，设法使设计满足需求会需要投入更多的时间。当开发人员希望进行重构时，他们往往被要求证明其决定的合理性。虽然这看起来有道理，可是却使得一件本来就困难的事变得更加困难(困难到无法完成)，结果是压制了重构的进行，或者使之只能在暗中进行。软件开发不是一种可以完全预测的过程，我们无法准确地计算出执行一项修改会带来多少好处，或者不执行一项修改会引起多大的损失。

向更深层理解重构需要融入到其他活动中去，包括领域问题的探索过程、开发人员的教育学习，以及开发人员与领域专家的思维交会。因此，可以在以下情况下进行重构：

- 设计没有表达出团队目前对于领域的理解；
- 重要的概念被隐含在设计之中(而且您看到了使之显式化的方法)；
- 您看到了一个使设计的某些重要部分更具有柔性的机会。

这种积极的态度并不是说任何时候的任何修改都是合理的。不要在软件发布的前一天重构。如果一个“柔性设计”只是表现出技术上的优雅性，而不能切中领域的核心，那就不要引入它。不要接受一个不能说服领域专家使用的“深层模型”，无论它看起来多么优雅。不要把事情绝对化，但是对于有益的重构，让自己更加积极勇敢一点。

13.6 将危机视为机会

自达尔文引入进化论以后，进化的标准模型就是物种随着时间而逐步地、较为稳定地改变。在 20 世纪 70 年代，这个模型突然被替换为“间断性均衡”模型。它扩展了进化的观念，认为长时间的逐步改变或者稳定性会被短时间内爆发出来的激烈改变所打断，然后一切又归于一种新的平衡。虽然软件开发具有自然进化所缺乏的目的指向性(虽然在某些项目中这一点并不明显)，但是它同样也遵循进化的韵律。

传统意义上的重构听起来非常稳定，但向更深层理解重构却往往不是如此。模型经过一段时期稳定的精化后，会突然为我们带来一种撼动万物的新理解。这些突破不会每

天发生，但是使我们得到深层模型和柔性设计的大部分修改都是源自于此。

当突破出现时，情形往往不像是一个机会，而更像是一场危机。突然之间模型出现了一些明显的缺陷。模型的表达能力出现了一个大洞，或者某些关键区域含混不清。也许模型中的声明是完全错误的。

这意味着团队的理解已经达到了一个新的境界。从他们现在的更成熟的观点来看，模型很差劲，他们可以想出一个更好的模型。

向更深层理解重构是一个连续的过程。隐含的概念被识别出来并成为显式概念。模型的一些部分变得更具有柔性，也许还获得了声明性的风格。开发突然之间到达了突破的边缘，冲破僵局，获得了一个深层模型——然后，稳定的精化又重新开始了。