



第 III 部分

面向更深层理解的重构

本书的第Ⅱ部分为维护模型与实现之间的对应关系铺好了基础。通过使用一系列行之有效的基本构建块，并在语言表达上保持一致，我们就能够使得开发活动较为稳健地进行。

当然，真正的挑战在于，我们所寻找的模型必须要恰到好处——它既捕捉到领域专家所关注的各个微妙细节，又能够为切实可行的设计提供支持。最终，我们希望开发的模型能够反映对领域的深刻理解。这样才能使得软件更加贴近领域专家的预期结果，同时也更符合用户的需要。本部分将阐明这个目标及其实现过程，并说明我们可以应用哪些设计原则和模式，来满足应用以及开发人员自己的需求。

成功地开发有效的模型，可以归结为以下 3 点：

- 复杂精致的域模型是可以得到的，而且值得为之付出努力。
- 除非通过一个迭代过程对模型进行反复重构，并且由领域专家以及有兴趣学习领域知识的开发人员密切参与，否则很难开发出有效的模型。
- 实现和有效地使用这种模型需要非常复杂的设计技巧。

重构的层次

重构是指对软件进行重新设计，而不改变软件的功能。重构不要求我们预先作出各种详细的设计决定，而是对代码进行一系列小的、独立的修改，每次修改都保持功能不变，但使得设计更加灵活易懂。另外，自动化的单元测试使我们能够相对安全地对代码进行验证。这个过程使得开发人员可以无需对软件设计作远期预测。

但是，几乎所有关于重构的文献都是着眼于如何用机械的方法对代码进行调整，以便代码能够更加清晰易读，或者在非常细节的层次上有所改进。如果开发人员认识到可



以采用某种既定的设计模式，“模式重构(refactoring to patterns)¹”¹的方法可以为重构过程提出一个更高层次的目标。不过，这种方法本质上仍然是从技术层面上来考察设计质量的。

有些重构会对系统的生存能力产生极大的影响，它们有的是由于我们对领域产生了新的理解而引起的，有的则通过代码来阐明模型的含义。这种类型的重构并不能代替模式重构或者微重构(模式重构和微重构必须持续进行)，但是它们形成了一个新的重构层次：模型重构。模型重构是基于对领域的理解来执行的，它通常会包含一系列的微重构(micro-refactorings)，但是其动机绝不仅仅是调整代码的状态。相反，微重构提供了一种方便的、小尺度的重构，以便能逐步产生一个更加合理的模型。重构的目标是开发人员不仅要理解代码做了些什么，而且要理解它为什么要这么做，并能够在与领域专家交流的过程中表述出来。

重构分类表(Fowler 1999)包括了大部分常见的微重构情形，这些微重构所针对的问题基本上都可以从代码中直接观察出来。相比之下，当我们对领域产生了新的理解时，修改域模型的方法可以有多种多样，要汇编出一个完整的模型重构分类表是不可能的。

和任何探索活动一样，建模本质上是非结构化的。每次在获得了新的知识或者深入的理解之后，我们都应该将模型重构到更深的层次。虽然已有的成功模型(如第 11 章中所讨论的)对我们大有帮助，但是我们不应误入歧途，企图将领域建模简化为一本菜谱或者一个工具包。建模和设计是需要创造力的工作。接下来的 6 章将给出一些特定的方法来考虑如何改进一个域模型，以及如果通过设计来实现这个模型。

深层模型

对象分析的传统方法包括从需求文档中界定出名词和动词，然后把它们分别作为最初的对象和方法。许多人认为这种方法将问题过于简化，也许只有在向初学者传授对象建模时它才会有用。然而事实上，在构造最初的模型时，我们通常都是基于一些浅薄的认识，因此这些模型往往非常幼稚和肤浅。

例如，我曾经参与过一个运输应用系统的开发，最初，我关于对象模型的想法包括货轮和集装箱。货轮从一个地方航行到另一个地方。集装箱通过装载和卸载动作改变与货轮的关联关系。这些想法准确地描述了运输过程中的一些物理动作，但是结果它并没有成为运输业务软件中一个非常有用的模型。

¹ Gamma 等人(1995)简要提到了将模式作为重构的目标。Joshua Kerievsky 已经将模式重构发展到了更加成熟和有用的程度(Kerievsky 2003)。



最终，在与运输专家经过数月的工作和多次迭代之后，我们得到了一个与最初想法大相径庭的模型。对于外行而言，它并不那么浅显易懂；但是对于专家而言却非常贴切中肯。它所关注的焦点被转移到货物运输业务上来了。

货轮还是在那儿，但是被抽象为“船舶航次(vessel voyage)”的一种形式。“船舶航次”是一次特定的行程，它被调度给某些货轮、火车或其他运输工具。货轮本身是次要的，它可以在最后一分钟被替换掉(因为需要对它进行维护或者运输计划被推迟)；但是船舶航次仍然按计划进行。集装箱几乎整个儿从模型中消失了。虽然它确实以一种不同的、非常复杂的形式出现在货物装卸系统中，但在原来的运输业务系统中，集装箱变成了一个操作性细节。货物在法律责任上的转移取代了货物在地理位置上的变动。一些不那么明显的对象，例如“装货单”被显式定义出来。

每当有新的对象建模人员进入这个项目时，他们的第一个建议就是——缺少“货轮”和“集装箱”类。他们都是聪明人，只不过尚未经历寻找和发现对象的过程罢了。

深层模型(deep model)能够清晰地表达出领域专家所考虑的主要问题，以及与这些问题关系最为紧密的知识，同时它又剥离了领域中的一些表面现象。这个定义没有提到抽象的问题。一个深层模型通常包含抽象元素，但是它也可以包含具体元素，以便能够直接切入到问题的核心。

当模型确实与领域非常贴切时，它就会获得功能强大、简单易懂的特点。这种模型几乎都具有一个共同特点，那就是它们都使用业务专家喜欢使用的、简单的(虽然可能是抽象的)语言。

深层模型/柔性设计

在不断重构的过程中，设计本身也需要能够支持改变。第10章考察了一些使设计更容易进行的方法，我们可以用这些方法来修改设计，也可以将设计与系统的其他部分整合起来。

某些特点可以使设计更易于修改和使用。这些特点并不复杂，但是非常具有挑战性。“柔性设计(Supple design)”，以及如何获得柔性设计，将是第10章要讨论的话题。

幸而，如果对模型和代码所作的每次修改都反映了新的理解的话，那么这些反复进行的修改正好可以为我们在最需要改变的地方引入灵活性，同时也为我们提供一种获得柔性设计的简单方法。用旧了的手套在手指需要弯曲的地方会变得柔顺，而其他部分则是硬性的以起到保护作用。同样，虽然上述的建模和设计过程需要进行反复尝试，但是实际上每次的修改会变得越来越容易，我们也将随着这些修改而不断接近一个柔性设计。



柔性设计除了便于修改，还有利于模型本身的精化。模型驱动设计是靠两条腿来支撑的：深层模型使我们获得的设计具有更强的表达能力；同时，如果开发人员能够体验到设计的灵活性，并清晰地理解设计中发生的事情，那么设计实际上又能将他们的理解反馈回模型发现过程。这个反馈循环是非常关键的，因为我们所寻求的模型不仅仅是一系列漂亮的想法，而应该成为整个系统的基础。

发现过程

为了使我们创建的设计确实能够切合眼前的问题，我们必须首先用一个模型来捕捉领域中心部分的有关概念。第 9 章“隐含概念转变为显式概念”要讨论的话题就是：如何积极地寻找这些概念，并将它们带入到设计中去。

模型和设计关系非常紧密，因此当代码难以重构时，建模过程将陷入停顿。第 10 章“柔性设计”将讨论如何为软件开发人员，而不仅仅是自己编写软件，以便它能够高效地扩充和修改。这项工作与模型的进一步精化是密不可分的。它往往需要使用更高级的设计技术和更严格的模型定义。

当对发现的概念进行建模时，我们通常需要依靠自己的创造力，并反复进行尝试来寻找最好的建模方式。但是，有时候我们可以借用别人已经做好的模式。第 11、12 章讨论了“分析模式”和“设计模式”的应用。这些模式并不是现成的解决方案，但是它们可以为我们提供知识并缩小我们的研究范围。

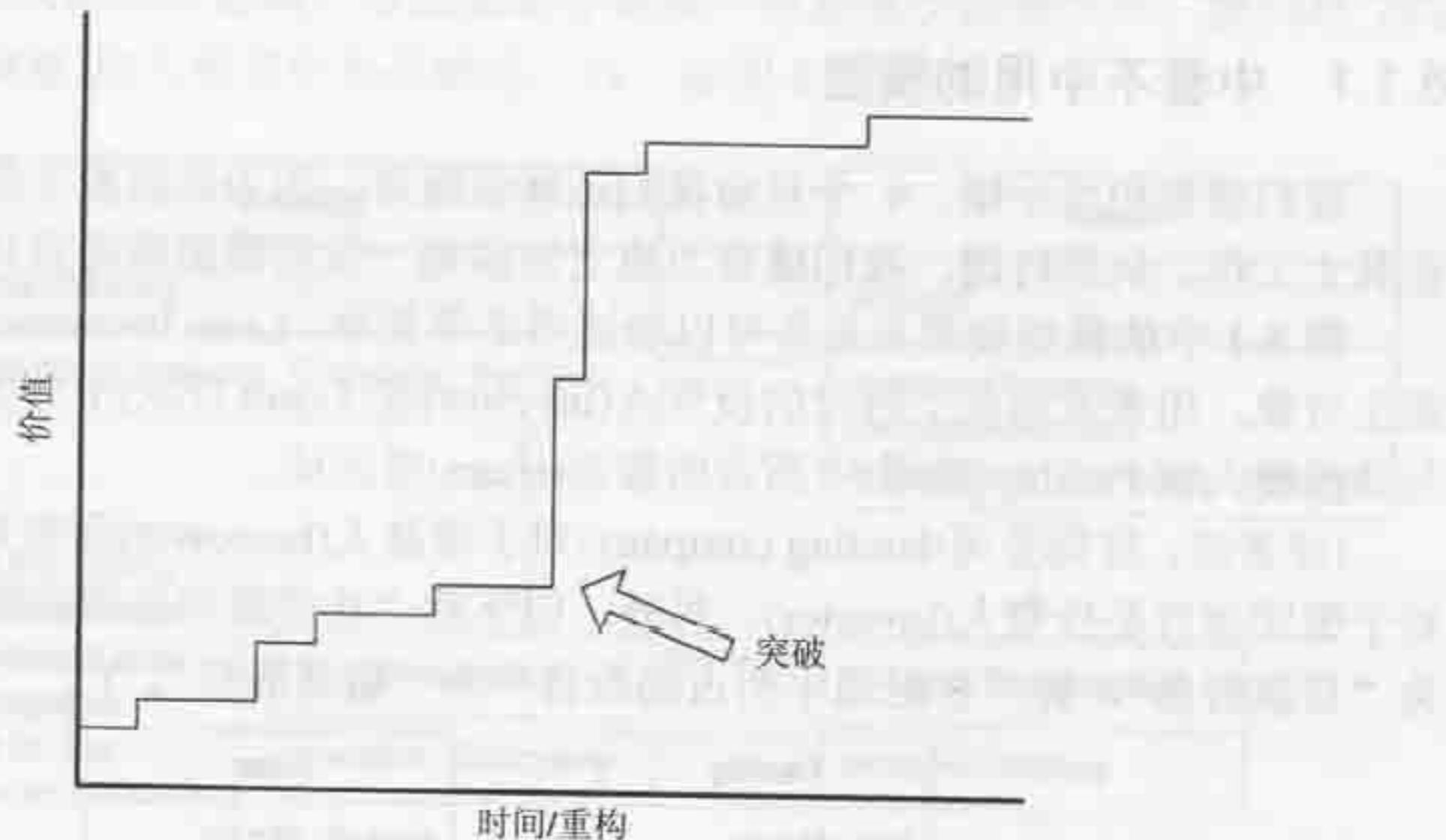
本书的第 3 部分一开始就会介绍在领域驱动设计中最令人兴奋的事件——突破。有时，当我们到达模型驱动设计阶段，并且概念都已经成为显示概念时，就会产生一个突破。此时，我们将有机会使软件的表达能力更强、功能更丰富，甚至出乎我们的意料。这可能意味着我们发现了某些新的特性，也可能意味着我们找到了一个简单、灵活的更深层次的模型，大块的代码将被替换为用这个模型来表达。虽然这种突破不会每天都发生，但却是非常宝贵的，因此当这种突破真的出现时，我们必须认识到它并把握住机会。

第 8 章讲述了一个真实的故事，在这个故事的项目中，不断深化的重构过程导致了一个突破。这种经历是无法事先规划的。但是，它为我们对领域重构进行思考提供了一个不错的场景。

第8章

突 破

重构的投入与回报是一种非线性的关系。通常每次细微的努力都会获得细微的改进，这些改进逐渐累积起来。它们与熵作斗争，成为避免模型变得陈腐僵化的第一道防线。然后，一些至关重要的理解会骤然间破土而出，为整个项目带来电击般的震动。



项目团队缓慢而稳健地吸收着领域知识，并将它们融入到模型中来。细微重构也许每次只涉及到一个对象：在这边调整一个关联，在那边移动一个职责；但通过一系列细微重构，深层模型就会随之逐渐浮现出来。

持续进行的重构通常也为一些飞跃性的质变作好了准备。代码和模型的每次精化都为开发人员提供了一个更清晰的视图，使他们有可能在理解上获得突破。经过一连串的修改之后，我们将得到一个更深层次、更符合实际和用户需求的模型。模型突然变得功能更强大、更清晰易懂，而模型的复杂性反而会降低。



这种突破并不是一种技术，而是一种事件。它的挑战之处在于，我们必须认识到正在发生的事情，并决定怎样对它进行处理。为了说明这种经历是一种什么样的感觉，我将讲述一个真实的故事，一个多年以前我工作过的项目，看我们是如何获得一个极具价值的深层模型的。

8.1 关于突破的故事

在纽约，经过一个漫长冬季的重构之后，我们得到了一个模型和一份设计：模型捕捉到了领域中的许多关键知识，而设计可以在应用中发挥一定的实际作用了。我们正在为某投资银行开发一个大型应用的核心部分，该应用将用来管理银行的银团贷款(syndicated loan)。

如果 Intel 想建造一座上十亿元的工厂，那么它所需要的贷款额实在太大了，以至于任何一家放贷公司都承担不起。因此，多个放贷公司会联合起来组成一个银团(syndicate)，通过共同投资来支持一个融通(facility)。通常会有一家投资银行充当银团的主席，对交易业务和其他服务进行协调。我们的项目就是要开发一个软件来跟踪和支持整个过程。

8.1.1 中看不中用的模型

我们感觉相当不错。4 个月前我们还麻烦缠身，因为此前留下的基础代码完全无法在其上工作。从那时起，我们就奋力将它转移到一致的模型驱动设计上来了。

图 8-1 中的模型使常见业务可以表述得非常简单。Loan Investment(贷款投资)是一个派生对象，用来表示某个特定的投资人(investor)在 Load(贷款)中分担的额度，这个额度与该投资人 in Facility(融通)中所占的股份(share)成正比。

(译者注：放贷公司(lending company)对于借款人(borrower)而言是放贷人(lender)，而对于银团而言是投资人(investor)。另外，以下称“在贷款中分担的额度(或所占的股份)”为“贷款股份”，称“在融通中所占的股份”为“融通股份”。)

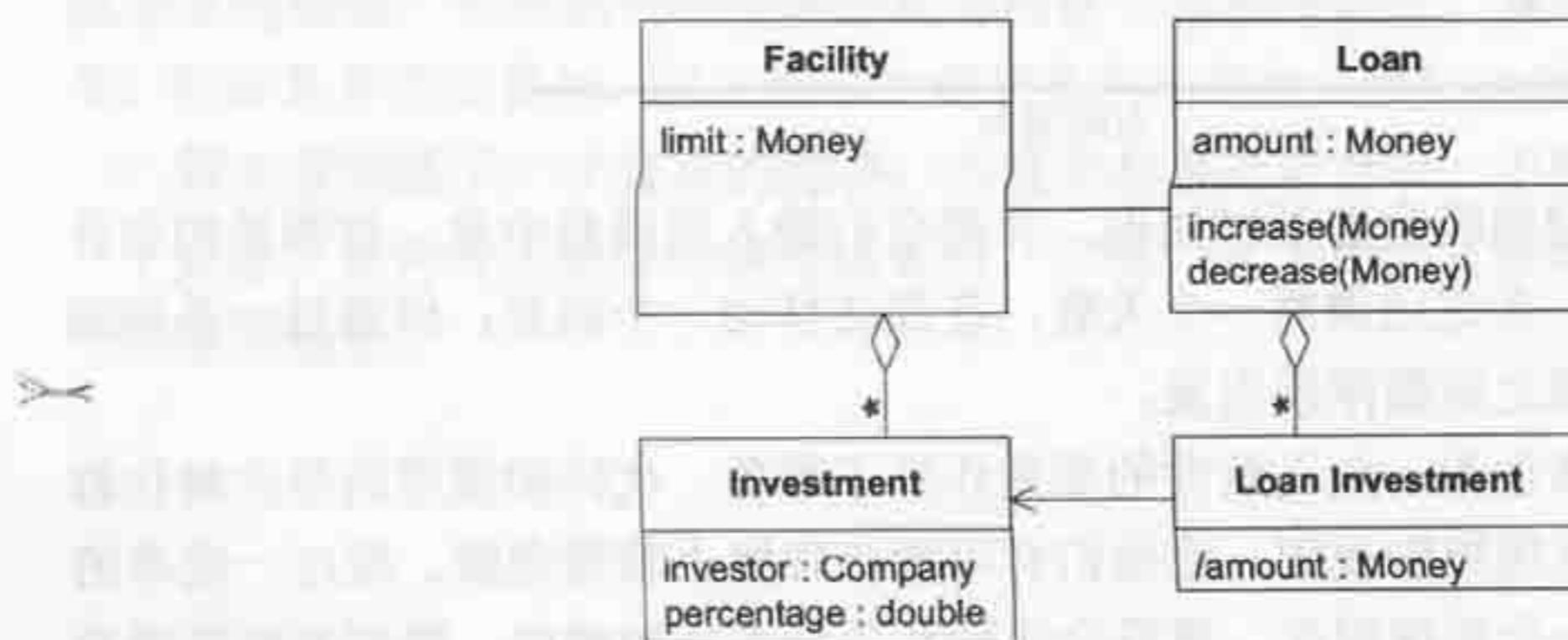


图 8-1 假设放贷人所占股份不变的模型



什么是“融通”

融通在这里指的并不是建筑物。在大多数项目中，我们都会从领域专家那里学到一些特殊的术语，这些术语将成为通用语言的一部分。在商业银行领域中，融通是指一个公司为借款而作出的承诺(commitment)。信用卡就是一种融通，它授权信用卡持有者在需要时借支一定的金额(借支金额的上限和利率都是预先规定好了的)。当您用信用卡来付账时，就产生了一笔未偿贷款(outstanding loan)，每次取钱都在支取您的融通，增加该贷款。最后，您必须偿还贷款本金，还可能要支付年费。年费是信用卡(融通)的持有费用，它与您贷款多少无关。

然而，有些迹象令我们感到不安。各种意料之外的需求在不断地影响我们，使设计变得更加复杂。一个明显的例子就是，我们渐渐领悟到在任何一次特定的贷款支取中，融通股份对各个放贷人需要提供多少贷款额仅仅起指导作用。当借款人要求支取贷款时，银团主席将要求所有成员支付各自的份额。

在收到银团主席的要求后，各个投资人通常按自己所持的股份出钱，但是它们往往还要与银团的其他成员进行磋商，以决定是多投资一点还是少投资一点。我们把 Loan Adjustments(贷款调整)加入模型中来反映这一点，如图 8-2 所示。

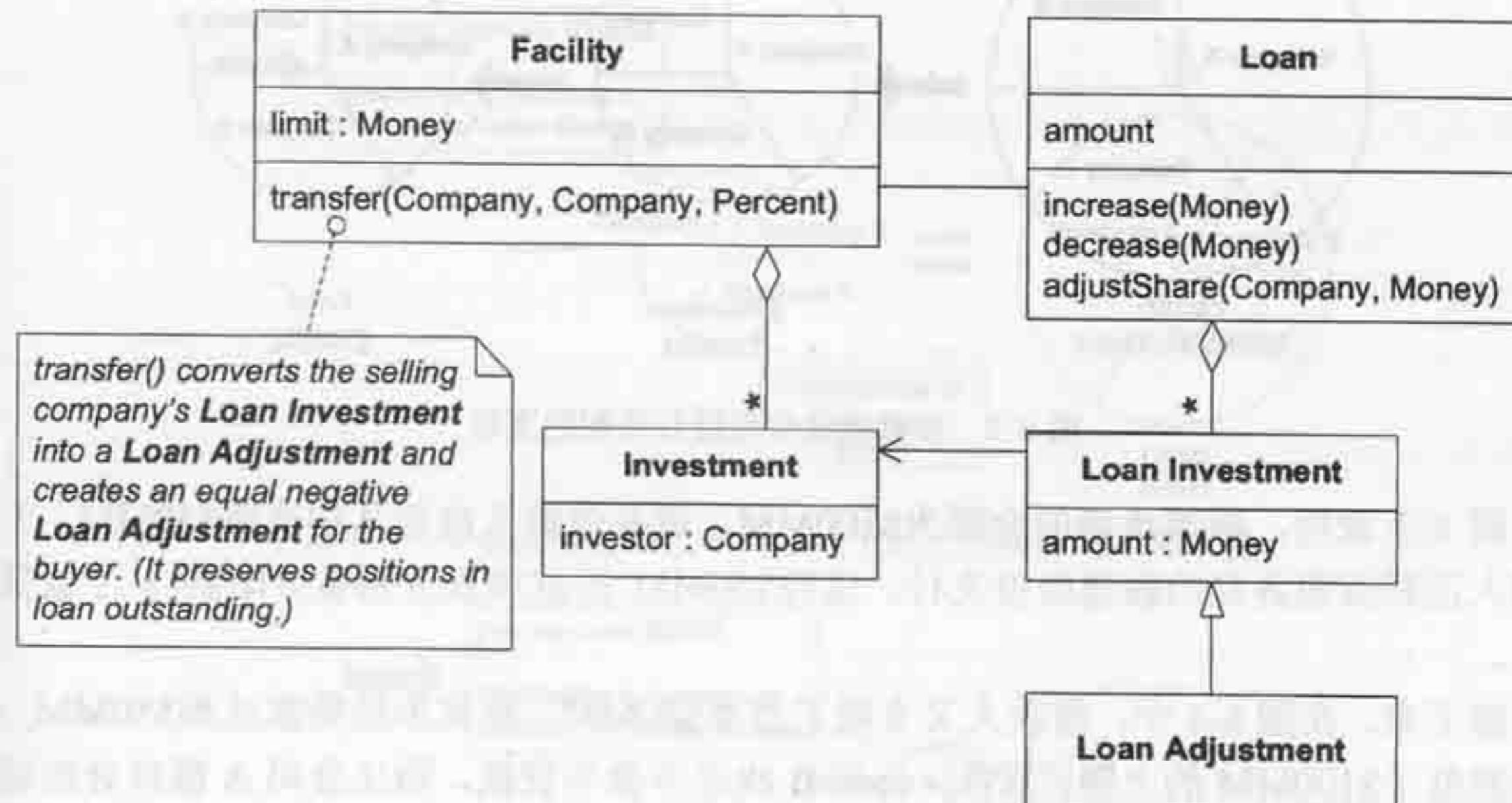


图 8-2 通过逐步改变模型来解决问题。Loan Adjustment 用来跟踪一个放贷人所占的贷款股份与它最初同意的融通股份之差

上面这种形式的精化使得模型能够随着各种事务规则的不断显露而变得更加清晰。



但是它同时也增加了模型的复杂性，而且似乎并不能迅速收敛，提供真正管用的功能。

更麻烦的地方在于，舍入计算会带来细微的不一致。虽然我们使用的算法越来越复杂，却一直没能解决这个问题。不错，在一项 1 亿美元的交易中，没人会在乎几个美分到哪里去了；但是银行家却不会信任一个无法精确地计算出每一个美分的软件。我们开始怀疑我们的困难只是某个根本设计问题的症状。

8.1.2 突破

过了一段时间，我们省悟到了问题的所在：我们的模型将融通股份和贷款股份捆绑到了一起，而这对于业务来说是不适当的。这个发现得到了广泛的响应。业务专家们都点头称是，并开始热心地帮助我们（我敢说，他们一定还在奇怪是什么让我们拖了这么久才想明白这一点）。我们迅速在白板上建立了一个新的模型。虽然细节问题尚未定型，但是我们已经知道新模型的关键特征了：贷款股份和融通股份必须能够相互独立地改变。按照这个理解，我们给出了一个新的模型，并用它对大量应用场景进行了走查。这个新模型看起来如图 8-3 所示：

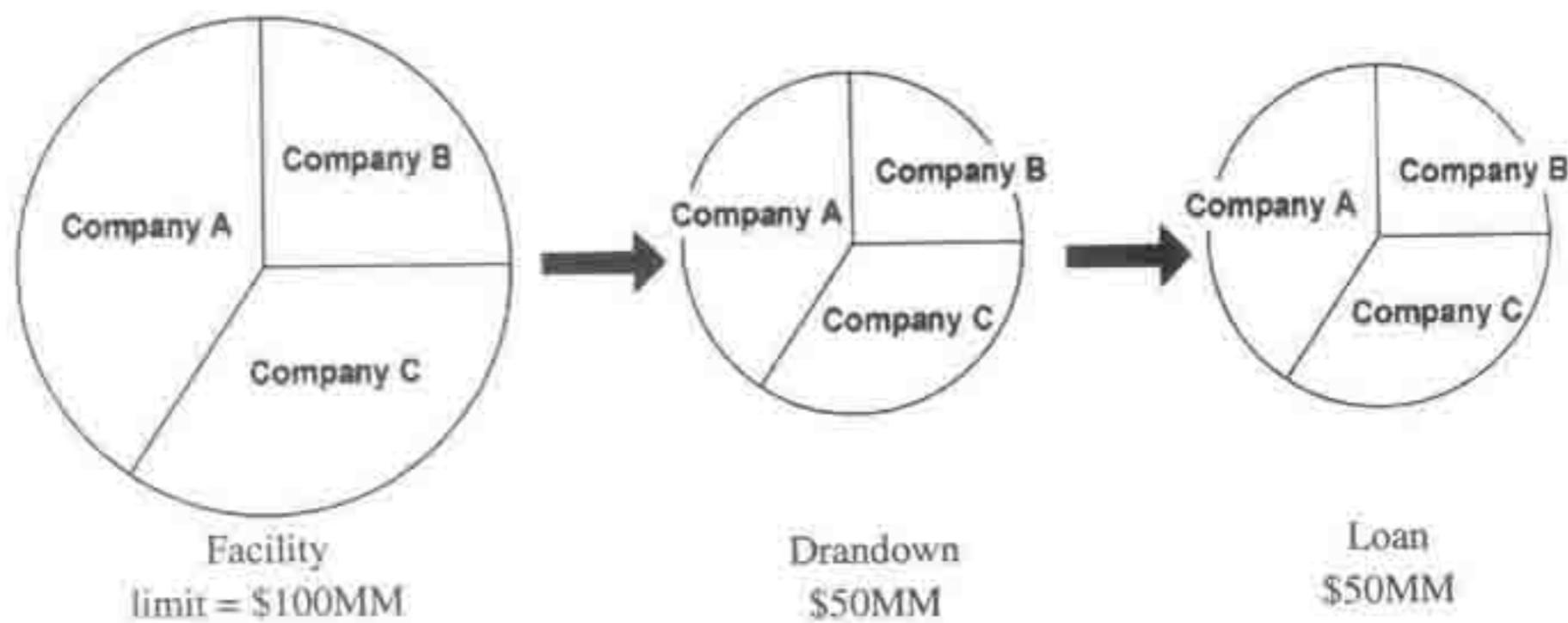


图 8-3 按融通股份来进行分配的支取

图 8-3 表明，融通承诺的金额为 \$100MM，现在借款人决定从中支取 \$50MM。3 个放贷人正好按照各自的融通股份支付，这样 \$50MM 的款项就分别被分配到 3 个放贷人头上。

接下来，在图 8-4 中，借款人又支取了另外 \$30MM，使其未偿贷款达到 \$80MM，但是仍然低于 \$100MM 的上限。这次，公司 B 决定不参与贷款，而让公司 A 额外分担这一部分。各个公司在这次支取中所占的份额反映了它们的投资选择。当支取额累计达到贷款额时，贷款股份不再与融通股份一致——这是普遍现象。

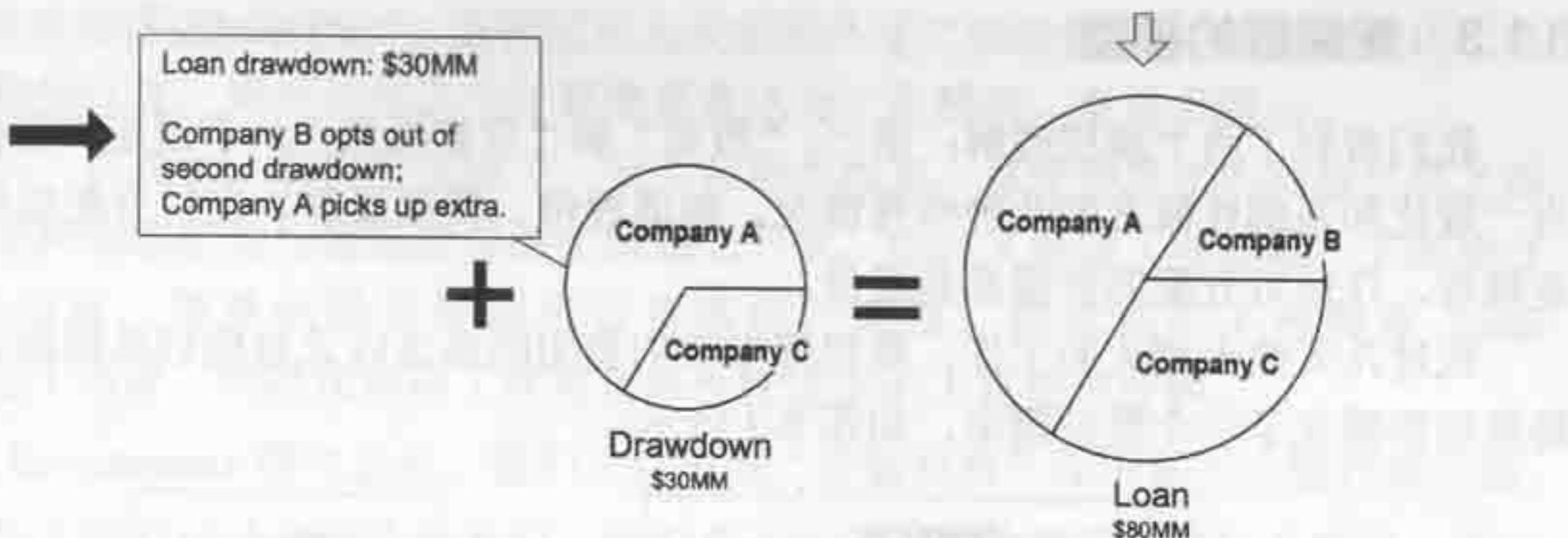


图 8-4 放贷人 B 决定不参加第二次支付

当借款人偿付贷款时，他的钱将按照各放贷人的贷款股份来分配，而不是按融通股份，如图 8-5 所示。同样，利息也是按照贷款股份来分配的，如图 8-6 所示。

另一方面，借款人必须为融通的持有权交付一定的手续费。手续费总是按融通股份来分配的，而不管钱实际上是由谁借出去的。贷款不会因为手续费支付而发生变化。甚至还有这样的情况，即各个放贷人在手续费中所占的份额是单独设定的，而与利息等的份额无关。

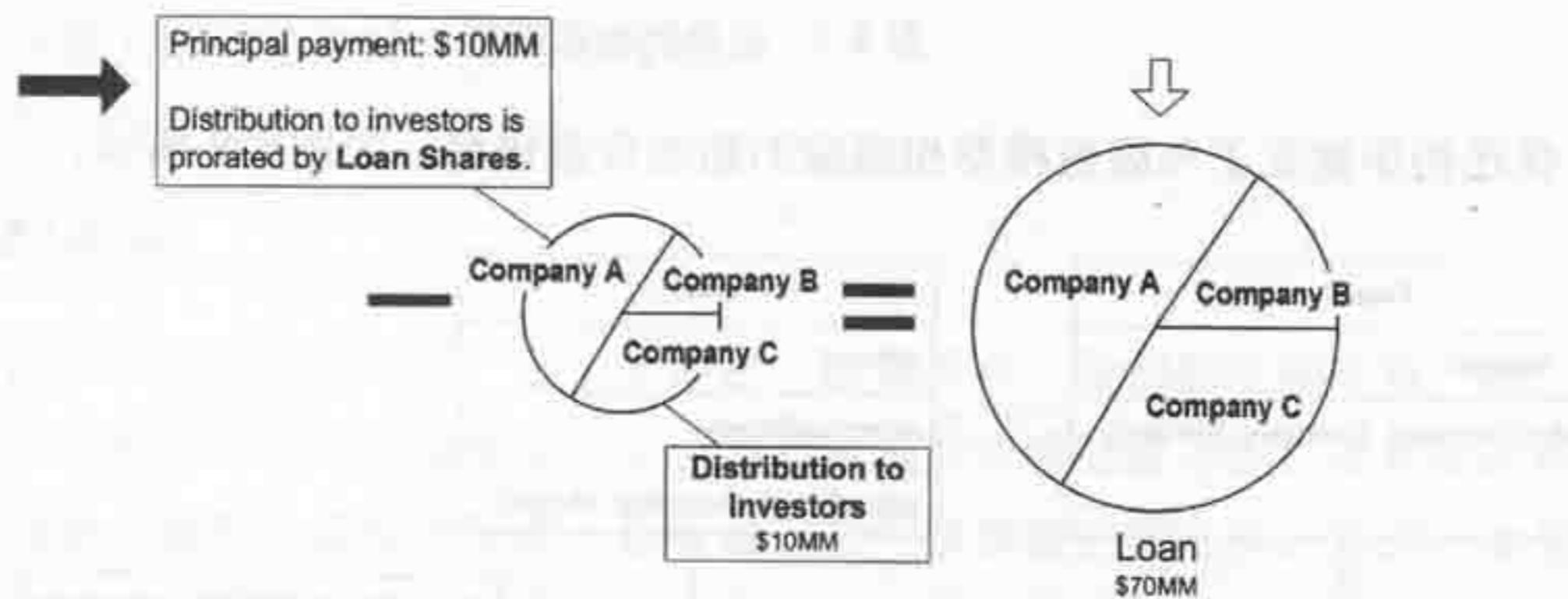


图 8-5 本金支付总是按未偿贷款中所占的股份比例来分配

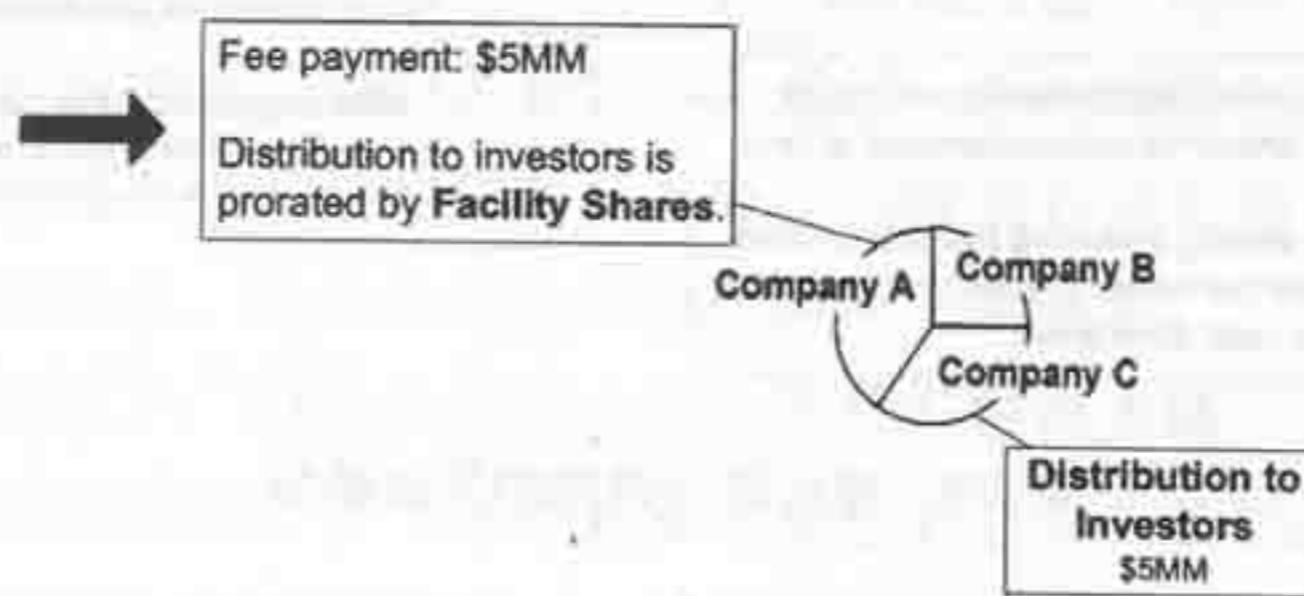


图 8-6 手续费总是按融通股份来分配



8.1.3 更深层的模型

我们得到了两个深层理解：第一，“投资”和“贷款投资”只不过是“股份”这个更为一般化和基础性概念的两种特殊情况。融通股份、贷款股份、支付分配股份，这些都是股份。任何可分配的价值都是股份。

经过几天热火朝天的工作，我根据专家们使用的语言以及与他们共同探讨过的一些场景初步建立了一个股份模型，如图 8-7 所示。

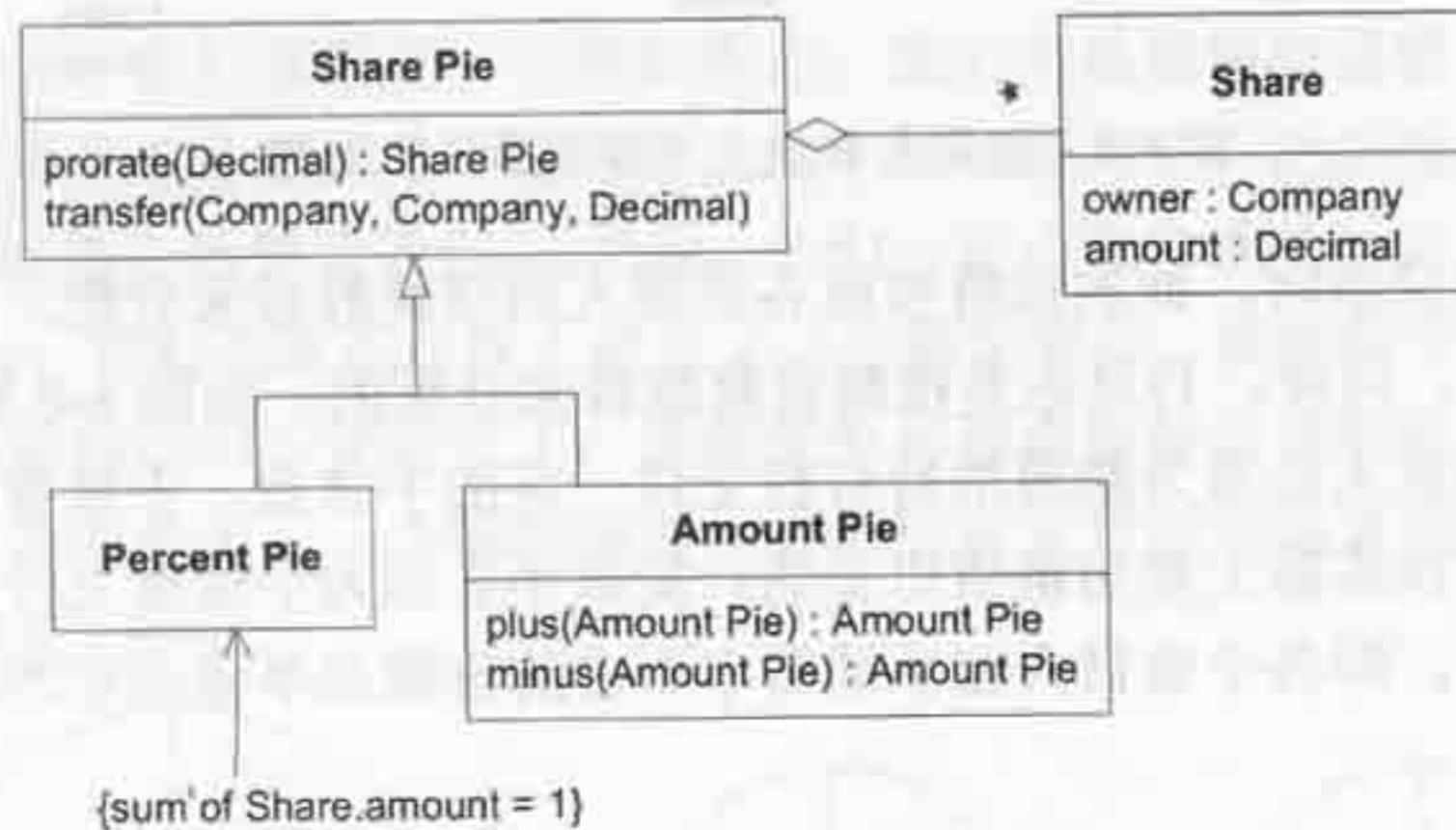


图 8-7 股份的抽象模型

我还初步建立了与股份模型相适应的新的贷款模型，如图 8-8 所示。

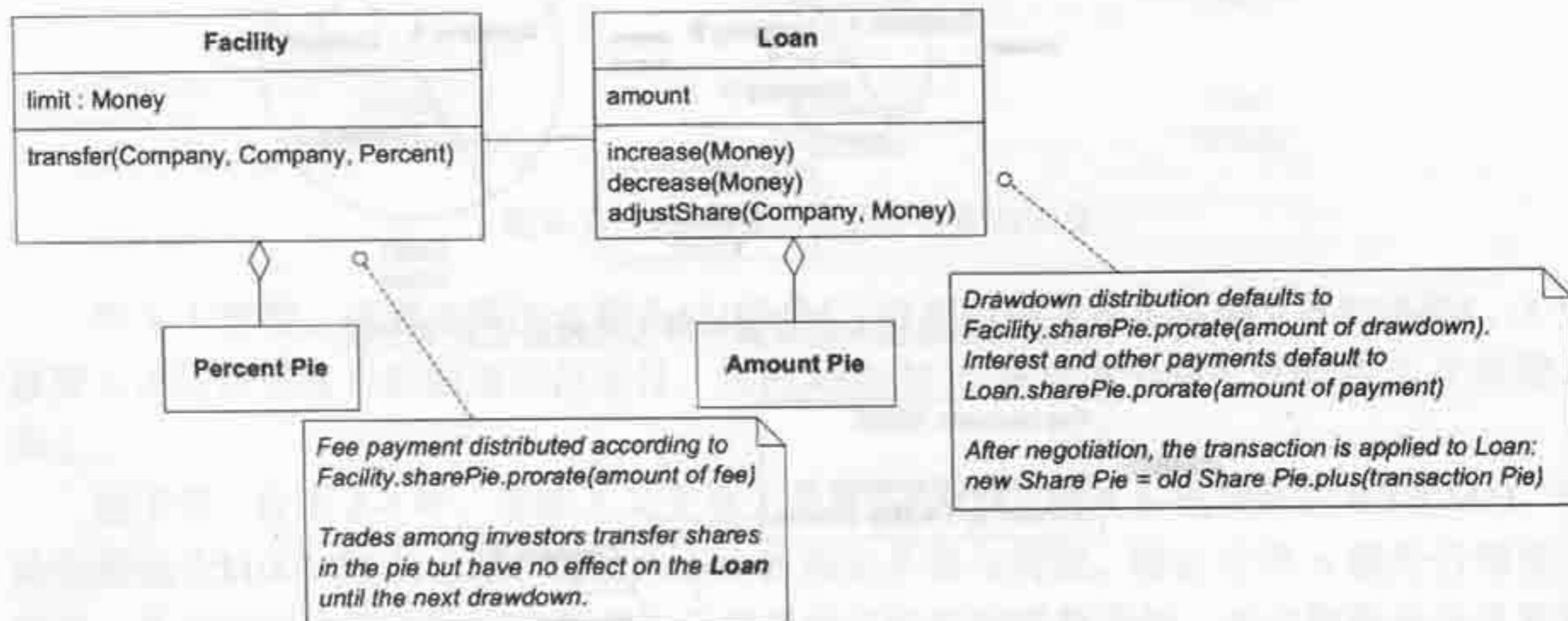


图 8-8 使用股份饼图的贷款模型

现在融通股份或者贷款股份不再需要用特殊对象来分别表示了，它们都被归结为更



为直观的“股份饼图(Share Pie)”。这种泛化为我们引入了“股份数学”，极大地简化了各种事务中的股份计算，同时使得这些计算更有表达力、更精确，更易于组合。

更重要的一点在于，新模型消除了一项不合理的约束，这使得原来的问题也随之消失了。新模型将贷款股份与融通股份的比例独立开来，同时又保留了对总额、手续费分配等的合理约束。贷款的股份饼图可以直接进行调整，因此我们不再需要 Loan Adjustment 这个对象，同时也消除了针对各种特殊情况的大量处理逻辑。

在 Loan Investments 消失之后，我们才认识到“贷款投资”并不是一个银行业术语。实际上，业务专家们已经多次告诉我们，他们并不知道“贷款投资”是什么意思。不过他们还是尊重我们的软件知识，并假设“贷款投资”在技术性的设计中是有用的。事实上，这个术语只不过是在我们对领域的理解还不完整的情况下杜撰出来的罢了。

利用这种新的看待领域的方法，我们突然之间可以更加轻松地遍历以前遇到过的所有场景了，而且感觉比以往的任何一次都要简单得多。同时，我们的模型图让业务专家们感到非常完美——他们以前总是感到模型对他们来说“太技术化”了。即使是一张在白板上画出的草图也能够让我们看到，最顽固的舍入计算问题已经被斩草除根了，因此我们无需再去使用那些复杂的舍入计算的代码。

我们的新模型工作得非常好，非常非常好。

我们都感到心力交瘁了！

8.1.4 冷静的决定

您也许会认为我们在那个时候一定十分得意。但是没有。我们的工期非常严峻，项目早就已经落在计划后面了，情况岌岌可危。我们最强烈的感受就是担心项目会延期。

重构的原则是，我们总是小步前进，总是保证每件事都能正常工作。然而，如果我们按这个新的模型对代码进行重构，那么将需要修改大量的支持代码，在这过程中几乎不会出现稳定的停止点。我们可以看到一些可行的细微改进，但是那些改进全都不能使我们与新模型的概念更加接近。我们还看到可以通过一系列细微的重构来达到目的，但是在过程中必须冻结系统的某些部分。另外，当时的自动化测试还没有广泛使用在这样的项目中，我们也没有，因此肯定会出现始料不及的缺陷。

现在我们该着手工作了，但数月的压力早已使我们精疲力竭。

此时，我们与项目经理举行了一次令人永生难忘的会议。我们的经理是一个睿智而大胆的人，他向我们问了一系列的问题。



Q1：如果采用这个新的设计，需要多久才能重新实现目前已有的功能？

A1：大概 3 个星期。

Q2：不用这个新设计可以解决问题吗？

A2：也许可以。但是无法保证。

Q3：如果现在不采用新设计，我们可以继续进入下一个发布版本吗？

A3：如果不作修改，我们的进度将会很慢。而且一旦有了安装基础，想再作修改将会困难得多。

Q4：采用新设计是一个正确的行动吗？

A4：我们知道目前的局面很不稳定，如果迫不得已的话，最好将就着应付过去。而且我们已经很累了。但是，毫无疑问，新设计是一种更简单的解决方案，而且更加贴近业务。从长远来看，它的风险更低。

他批准了我们的新设计，并且告诉我们他会控制好局面。作出这个决定需要相当的勇气和信心，这使我总是对他钦佩不已。

我们全力以赴，在 3 个星期后完成了任务。任务的工作量非常大，但是进展异常顺利。

8.1.5 成效

不可捉摸而又出人意料的需求变更终于停止了。舍入的逻辑变得稳定而有意义，虽然它实际上仍然非常复杂。我们交付了第 1 个版本，而开发第 2 个版本的思路也已经非常清晰了。我的神经衰弱也勉强有了好转。

在开发第 2 个版本的时候，股份饼图变成了整个应用的主旋律。技术人员和业务专家用它来讨论系统。市场人员用它来向潜在顾客解释特性。那些潜在或非潜在顾客都能立即领会它的含义，并使用它来讨论特性。股份饼图实际上已经成为通用语言的一部分，因为它触及到了银团贷款的核心问题。

8.2 时机

突破能使我们得到一个更深层次的模型。但是，当突破即将出现时，它却常常令人畏惧。相对于大部分重构而言，突破引发的是一种高回报、高风险的修改。这在时间上可能会不合适。

尽管我们希望工作总是能够平顺地进行，但事实却并非如此。为了转移到一个真正



的深层模型上，我们必须从根本上调整自己的思路，同时还要对设计作出重大修改。在许多项目的建模和设计工作中，大部分重大的进展都是由这些突破带来的。

8.3 着眼于根本

不要企图“制造”突破，那会使您无法动弹。通常，只有在执行了许多细微的重构之后才可能出现突破。我们的大部分时间都会花在细微的改进上，连续的每次精化使我们对模型的理解节节进步。

为了为突破创造条件，我们必须集中精力消化领域知识，并提炼出一套稳定的通用语言。寻找那些重要的领域概念，并将它们显式地体现到模型中(在第 9 章将会讨论这一点)。接着，精化设计使之更具有柔性(见第 10 章)，最后精炼模型(见第 15 章)。利用这些更容易把握的手段来增进我们的理解——这往往能成为突破的先驱。

不要犹豫作一些小的改进。即使这些改进并没有突破原来的通用概念框架，也会逐步深化我们的模型。不要企图看得太远，那样会反受其害。只要随时注意可能出现的机会就行了。

8.4 尾声：一连串的新理解

那次突破使我们走出了迷宫，但是故事尚未结束。深层模型为我们带来了意外的机会，使应用更加丰富，设计更加清晰。

在采用股份饼图的软件版本发布几周之后，我们注意到模型中另一个使设计变得复杂的地方。我们漏掉了一个重要的实体，结果它的职责不得不由其他对象来分担。具体地说，支取贷款、交纳手续费等业务都是由一些重要的规则来管理的，这些规则的逻辑被实现为不同的方法，然后勉强塞进了 Facility 和 Loan 类中。在“股份饼图”突破以前，这些设计上的问题几乎不会引起我们的注意。现在我们对领域的理解更加清晰了，它们才变得明显起来。现在我们注意到，在讨论时会蹦出一些模型中从未出现过的术语，例如“事务”(表示一次金融交易)。我们开始认识到，这些术语一定是被隐含在那些复杂的方法中了。

按照类似于前面描述的过程(不过谢天谢地，这次时间压力轻多了)，我们的理解又向前迈进了一步，并获得了一个更深层的模型。这个新模型使那些隐含的概念(如 Transaction)显露出来，同时又简化了 Position(这是一个包含 Facility 和 Loan 的抽象类)。现在我们能够更加容易地定义各种事务、事务规则、协商程序、审批流程等概念，而且



第Ⅲ部分 面向更深层理解的重构

描述这些概念的代码也相对更加不言自明了，如图 8-9 所示。

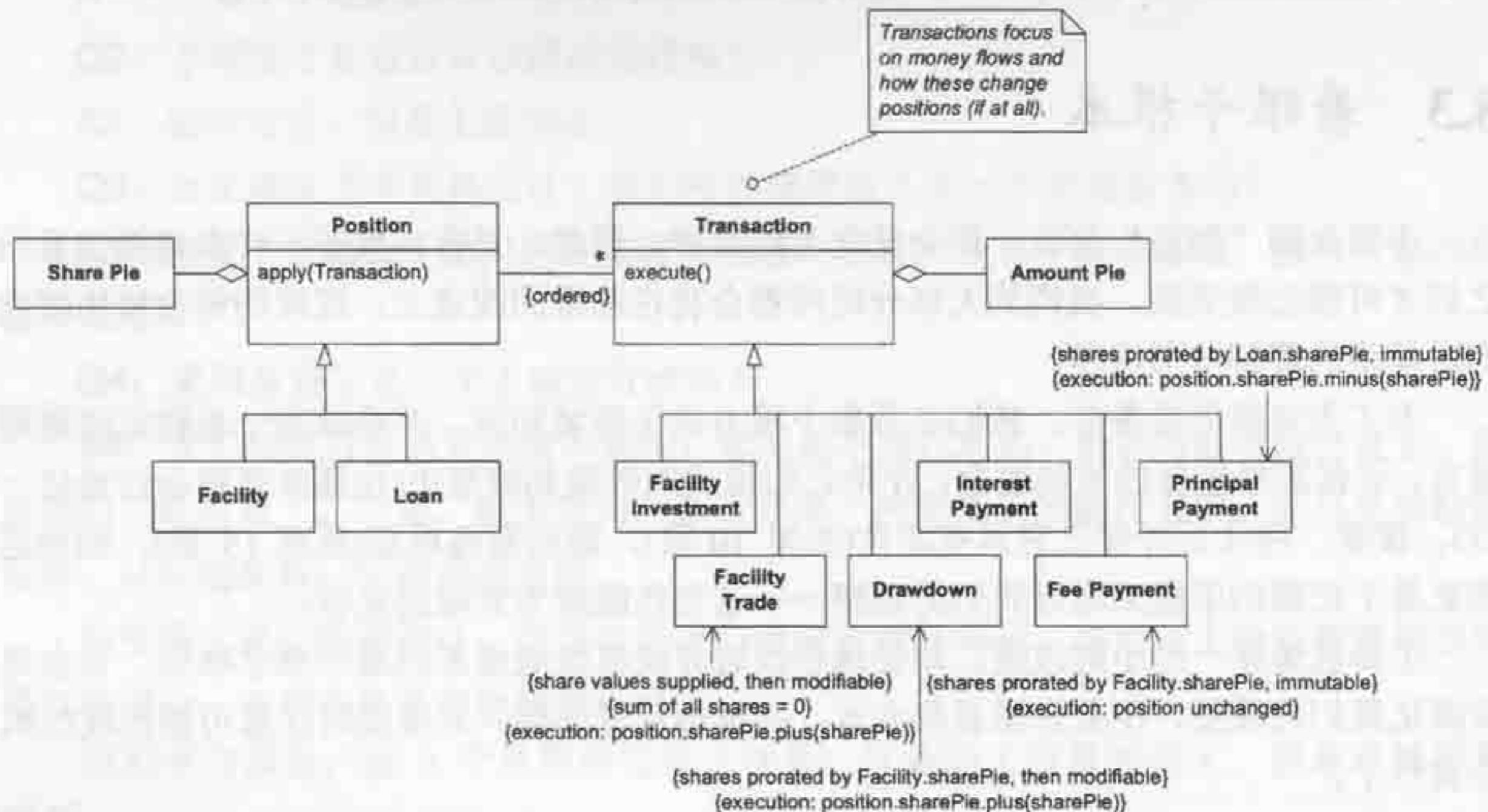


图 8-9 几个星期后再次突破得到的模型。我们很容易就能把 Transaction 上的约束精确地描述出来

连续突破是一种很常见的情况。在经过一次真正的突破并得到一个深层模型之后，设计变得更加清晰、简单，而新的通用语言又增进了沟通，于是又促成了下一次的建模突破。

在这个阶段，当其他的项目正在开始被积累的信息弄得举步维艰的时候，我们的开发步伐却正在快马加鞭。



第9章

隐含概念转变为显式概念

深层建模听起来非常不错，但是它具体是怎样进行的呢？深层模型包含领域的中心概念和抽象，能够简洁而灵活地表达用户活动、问题及其解决方案的本质知识，这也是深层模型的强大之处。在进行深层建模时，我们首先要设法在模型中将领域的本质概念表达出来；然后进行模型精化，不断消化知识、反复进行重构。实际上，当我们识别出一个重要的概念，并在模型和设计中将它显式地描述出来的时候，这个过程就已经开始了。

当开发人员识别出了某个概念(可能是在讨论时间接提到的，也可能是隐含在设计之中)的时候，就会对领域模型及其对应的代码执行一些转换，在模型中加入一个或多个对象或关系，将这个概念显式地描述出来。

这种隐含概念到显式概念的转变偶尔会形成一次突破，从而使我们得到一个深层模型。但是，突破一般不会那么快就出现。在突破到来之前，我们可能已经在模型中加入了多个显式的重要概念，通过一系列重构反复调整对象的职责，修改它们与其他对象的联系，甚至连名称都要几经修改。最后，突破终于发生，一切都变得清晰了。但是，我们首先必须通过某种方式将隐含概念识别出来，无论这种方式多么原始——然后突破过程才会开始。

9.1 概念挖掘

开发人员必须能够敏锐地捕捉到那些隐含概念所露出的蛛丝马迹。有时候，他们还不得不主动出击，将这些隐含概念搜寻出来。大部分隐含概念都是这样发现的：倾听团队的表达用语；仔细研究设计中不协调的地方，以及专家们似乎有些自相矛盾的说法；



阅读与领域相关的文献；此外还要进行大量的实验。

9.1.1 倾听表达用语

您可能有过这样的经历：用户多次谈到报表中的某项，那一项可能是由许多对象属性汇集而成的，也可能是通过直接查询数据库得到的。应用的另一部分也要收集同样的数据集，以便制作界面或报表，或者从中提取某种信息。但是，您却一直不觉得需要为它建立一个对象。也许，您从未真正地理解用户所说的某个特定的术语有何含义，也没有认识到它的重要性。

然后，您头脑中突然灵光闪现。报表中那一项的名字指明了一个重要的领域概念，于是您激动地与专家们讨论这个新发现。专家们也许会因为您终于想通了这一点而感到安慰，也许会开始打哈欠——因为他们自始至终都认为这样的理解应该是理所当然的。不管专家们怎么反应，您开始在图板上动手画起模型图来(您总是在图板中预先画上一些示意图)。用户们会试图纠正新模型的某些细节，但是您可以告诉他们，在这次的讨论中模型会有点变化。您和用户的互相理解加深了，用模型来演示一些特殊场景时也显得更加自然。领域模型所使用的语言变得更强大了。最后，您对代码进行重构使之符合新的模型，同时发现自己获得了一个更加清晰的设计。

倾听领域专家所使用的语言。有没有一些术语被用来简洁地描述非常复杂的事物？他们有没有(也许是委婉地)纠正您的用词？当您使用一个特定的短语时，他们的表情是否变得不再迷惑？这些都是线索，根据这些线索把那些概念识别出来，就能使模型获得改进。

这种思路和过去的“名词即对象”的观点是不同的。听到一个新单词只是开了个头，接下来我们还要与人交谈、消化知识，力图挖掘出一个清晰且有用的概念来。如果用户和领域专家使用了尚未在设计中出现过的词汇，那就是一个警告信号；如果开发人员和领域专家都使用了尚未在设计中出现过的词汇，那就是一个严重的警告信号。

也许，我们应该把这种警告看成机会。组成通用语言的词汇普遍存在于对话、文档、模型图，甚至是代码之中。如果一个术语没有在设计中出现过，那么把它包含到设计中去——这是一个改进模型和设计的机会。

示例：听出一个运输模型中缺失的概念

团队已经开发了一个正在工作的应用，用来预订货物。他们现在正在开始“作业支持”应用的开发，来帮助工作人员管理工作订单。装卸人员根据这些工作订单在运输的起点和终点装卸货物，或者在货轮之间转运货物。

预订系统用一个路线引擎来安排货物的行程。行程的每一航段(leg)都在数据库的表



中保存为一行，指定了装运该货物的船舶航次的 ID（特定船舶的特定航次），以及货物的装载地点和卸载地点，如图 9-1 所示。

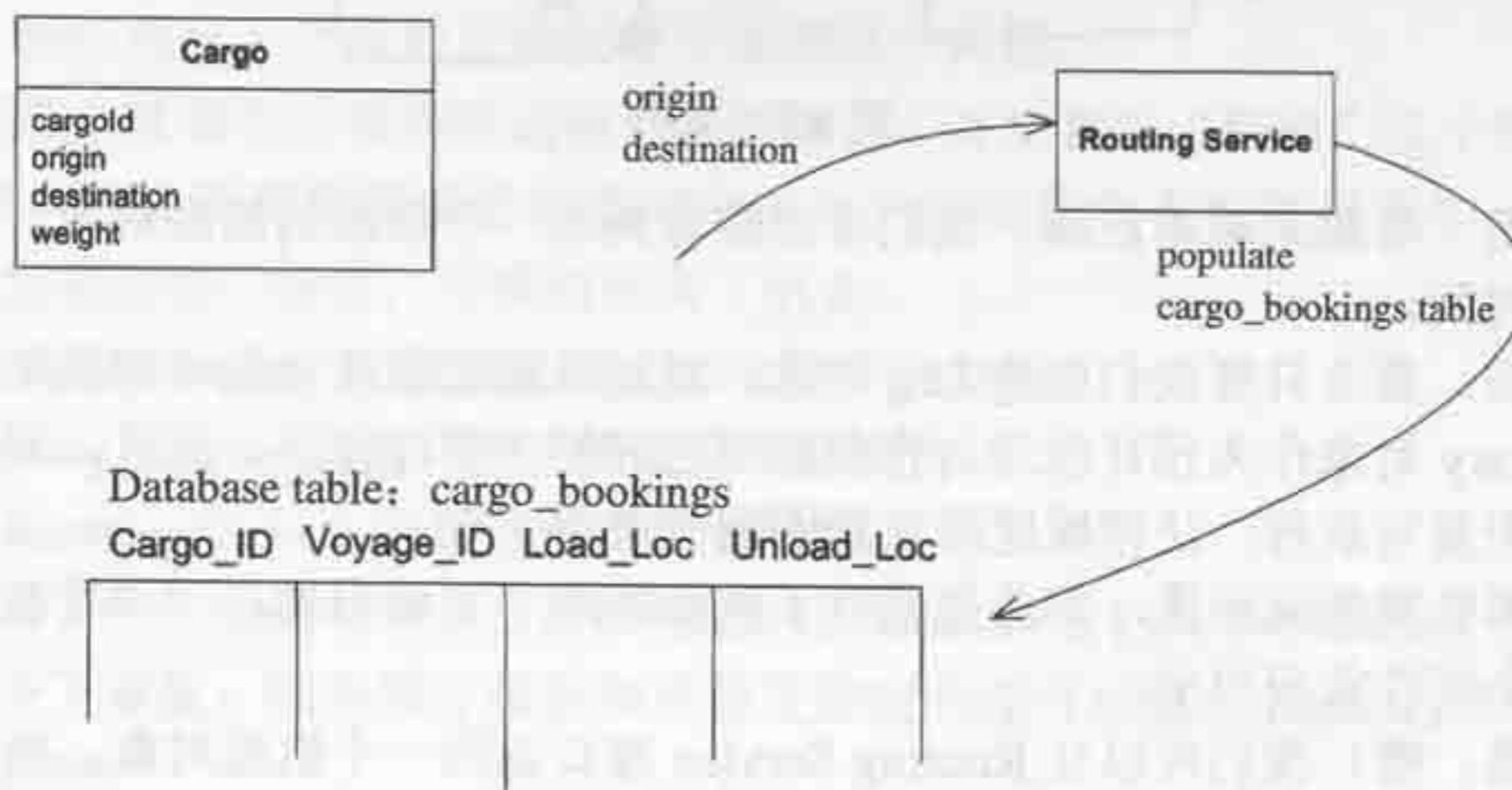


图 9-1 预订系统

让我们来听听开发人员和运输专家的对话(经过了高度缩简)。

开发人员：我希望确认“cargo bookings(货物预订)”这个表是否已经包含了作业应用所需的所有数据。

专家：工作人员需要货物的完整航程(itinerary)。现在货物预订表有哪些信息？

开发人员：货物 ID、船舶航次，以及每一航段的装货港和卸货港。

专家：日期呢？作业需要按预期的时间来进行装卸工作。

开发人员：哦，这可以从船舶航次的计划中查到。货物预订表已经经过了规范化处理。

专家：嗯，日期通常都是必需的。作业人员将用航程来安排后面的装卸工作。

开发人员：很好，他们绝对可以访问到这些日期。作业管理应用可以提供整个装货和卸货序列，以及各次装卸作业的日期。我猜这也就是您所说的“航程”。

专家：不错。航程是他们需要的主要信息。实际上，您知道，预订应用已经提供了一项菜单，用来将航程打印出来或者给顾客发电子邮件。您能设法利用那个功能吗？

开发人员：我想那只是个报表。我们不能把作业应用建立在那个报表之上。

[开发人员像是在思索什么，接着兴奋起来。]

开发人员：哦，航程实际上是把预订和作业连接起来了。

专家：对，还连接了一些客户关系。

开发人员：[在白板上画草图]那么您觉得是这样的吗？

结果如图 9-2 所示。

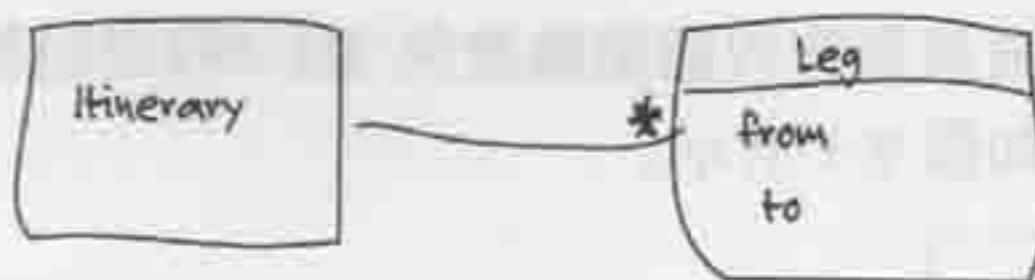


图 9-2 草图

专家：嗯，看起来基本正确。我们希望能看到每一个航段的船舶航次、装货和卸货地点，以及时间。

开发人员：那么只要我们创建 Leg 对象，就能从船舶航次计划中得到时间了。我们可以把 Itinerary 对象作为预订应用与作业应用之间的主要连接点。而且，我们还可以使用这些逻辑来重写航程，让领域逻辑重新回到领域层中来。

专家：那些我都搞不懂，但是您说对了航程的两个主要用处，一个是在预订应用的报表中，一个在作业应用中。

开发人员：嘿！我们可以让 Routing Service 接口返回一个航程对象，而不是把数据存到数据库表中。这样路线引擎就无需了解数据库表的细节了。

专家：啊？

开发人员：我的意思是，我可以让路线引擎仅仅返回一个 Itinerary，而由预订应用在保存其他预订信息时将 Itinerary 一起保存到数据库中去。

专家：您是说现在并不是这么做的吗？

于是开发人员回去与路线引擎的开发人员商量。他们按新的理解修改了模型和设计，并在必要的时候与运输专家联系。最后他们得到了如图 9-3 所示的模型图。

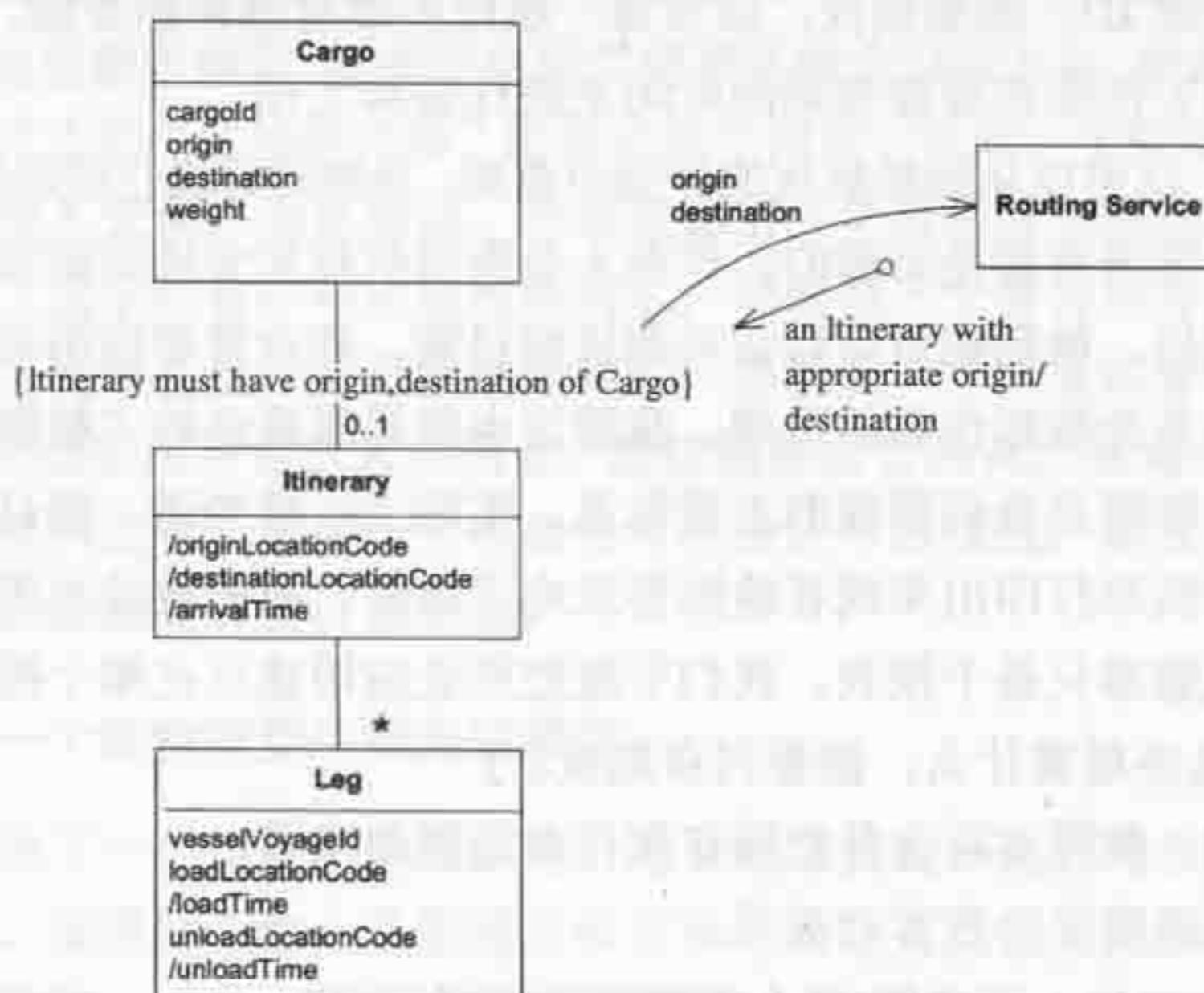


图 9-3 最终模型图



接着开发人员对代码进行重构以反映新模型。在一个星期内，他们对代码作出了一系列修改，每次修改都包含两到三次快速的连续重构。但是预订应用的航程报表还没有进行简化，他们将在下个星期开始时着手处理这个问题。

开发人员一直都在仔细倾听运输专家的谈话，并注意到“航程”这个概念对他来说有多么重要。虽然所有数据都已收集完毕，行为已经隐含在航程报表中，但是显式的 Itinerary 作为模型的一部分，为模型带来了机会。

通过重构将 Itinerary 对象显式描述出来的好处包括：

- Routing Service 的接口定义更具有表达能力；
- 把 Routing Service 与预订数据库表解耦；
- 澄清了预订应用与作业支持应用之间的联系(即它们共享 Itinerary 对象)；
- 减少了重复，因为预订报表和作业支持应用都可以通过 Itinerary 得到装货/卸货的时间；
- 将预订报表中的领域逻辑迁移到了隔离的领域层；
- 扩充了通用语言，使得开发人员与领域专家或者在开发人员内部讨论模型和设计时，能够更精确地进行沟通。

9.1.2 检查不协调之处

我们需要的概念并不总是浮在表面上，能通过交谈和文档显露出来。有的概念可能要通过挖掘和创造才能得到。而挖掘这些概念的地方，就是我们的设计中最不协调的地方。在那里，程序的功能很复杂、很难解释，而且每个新的需求都会增加其复杂性。

有时，我们很难认识到居然遗漏了某个概念。每个功能都有相应的对象去实现，但是我们发现某些对象的职责很不自然。有时，虽然我们认识到遗漏了什么，却总是找不到一个模型能解决这个问题。

这个时候，您必须积极地要求领域专家和您一起寻求答案。如果幸运的话，专家们可能会乐意考虑各种想法，并对模型进行验证。如果不那么幸运的话，您和您的开发人员伙伴就必须提出不同的想法，让领域专家对这些想法进行判断，并注意观察专家们的表情是赞同还是反对。

示例：利息计算之摸索版

下面这个故事以一家虚拟的金融公司为背景，该公司从事商业贷款和其他计息资产的投资业务。项目团队正在开发一个跟踪各项投资及其收益的系统，通过逐项添加特性来使系统不断完善。每天晚上金融公司都会运行一个批处理脚本组件(以下称之为 nightly



batch), 计算当天的利息和手续费, 然后将计算结果记录到公司记账软件的合适账号中去, 如图 9-4 所示。

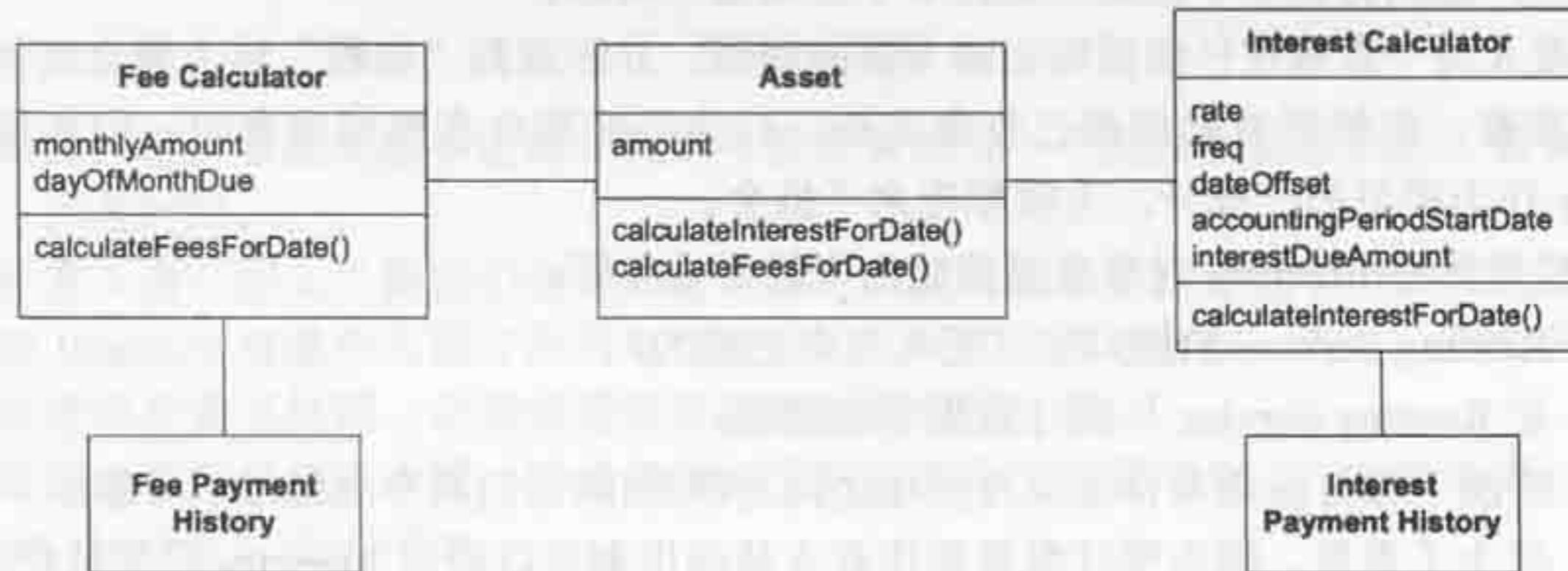


图 9-4 一个不协调的模型

nightly batch 脚本遍历每个 Asset(资产), 让它们按当天的日期计算各自的利息 (calculateInterestForDate())。然后, 脚本把返回值(收益额), 连同登记该结果的分类账 (ledger) 的名称, 传递给一个服务(服务提供了记账程序的公共接口), 由记账软件将收益额登记到指定的分类账。对各个 Asset 的每日手续费也进行类似地处理, 并登记到另一个分类账中。

一个开发人员一直在奋力处理利息计算问题, 这个问题已经越来越复杂了。她开始怀疑应该能找到一个模型来更好地完成任务。于是, 她请求她喜欢的领域专家和她一起来深究这个问题。

开发人员: Interest Calculator 像要失去控制了。

专家: 这个部分是很复杂。有很多情况我们还没有提到呢。

开发人员: 我知道。我们可以通过更换 Interest Calculator 来加入新的利息类型。但是, 现在最大的问题在于, 如果别人没有按时支付利息, 那么我们应该怎么处理这些特殊情况呢?

专家: 这并不是特殊情况。人们支付利息的方式可以非常灵活。

开发人员: 记得我们将 Interest Calculator 从 Asset 中分离出来对设计产生了很大的帮助。我们可能还要进一步分解 Asset。

专家: 好的。

开发人员: 我在想, 对于利息计算, 您们是不是有什么说法。

专家: 您指的是什么?

开发人员: 嗯, 比如说, 我们希望跟踪在一个会计期间(accounting period)尚未付款



的到期利息(interest due)。您们有没有什么术语来专指这种利息？

专家：哦，我们实际上并不是像那样的。利息收入(interest earned)和付款(payment)的过账(post)是完全分开的。

开发人员：那么您们不需要那些数字(尚未付款的到期利息)？

专家：嗯，有时候可能要看一下，但那并不是我们处理业务的方法。

开发人员：好吧，如果付款和利息是分开的，那我们也许应该这样来建模。这个怎么样？[在白板上画出，如图 9-5 所示的草图]

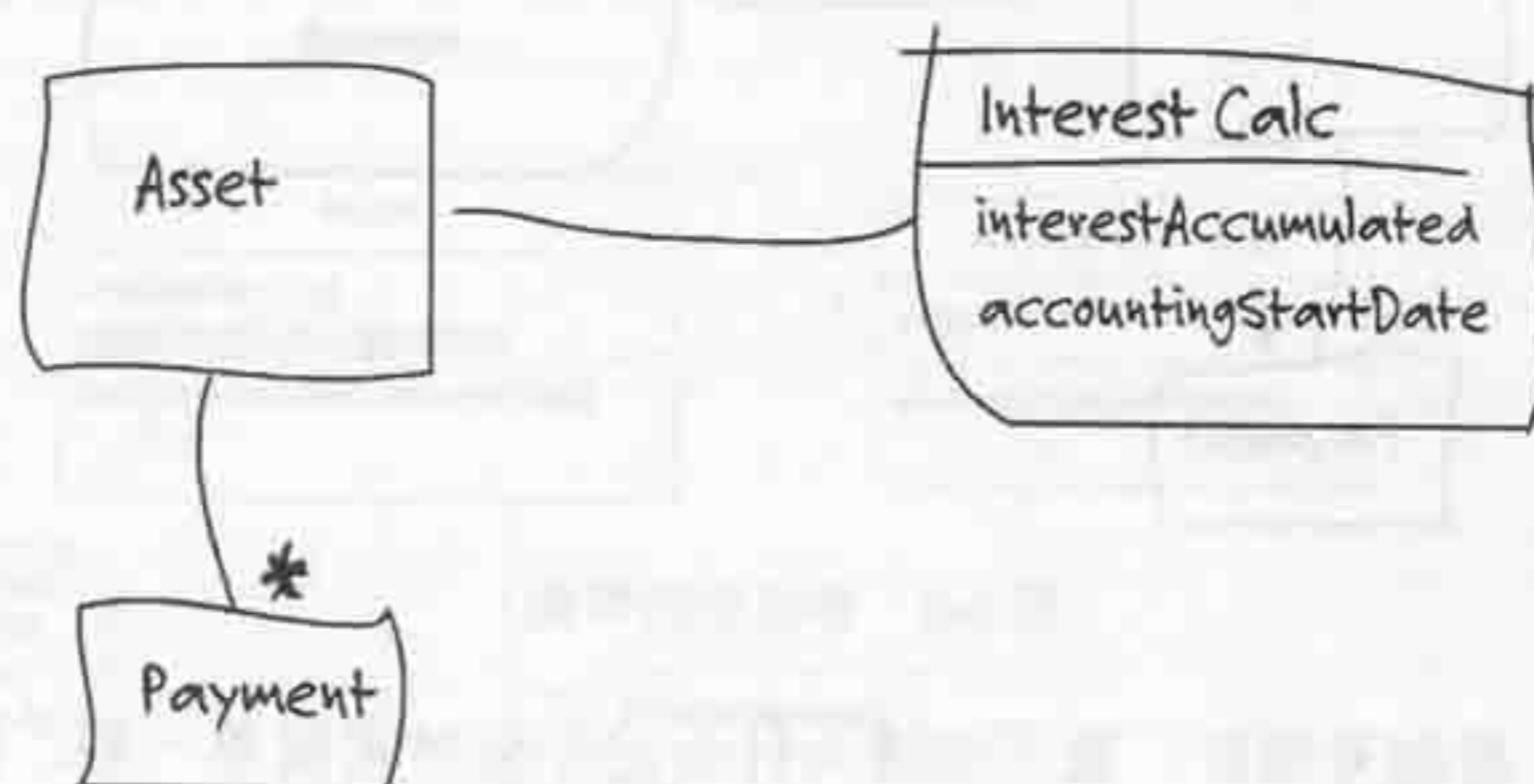


图 9-5 草图

专家：我觉得有点道理。但您只不过把它移动了一个地方而已。

开发人员：现在 Interest Calculator 只管跟踪利息收入，付款数字则是由 Payment 分开管理的。这并没有简化多少，但是它是不是更好地反映了您们的业务习惯呢？

专家：啊，我懂了。我们可以保留利息的历史吗？就像 Payment History(付款历史)一样。

开发人员：啊，利息历史是一个新特性。但是它本来应该在最初的设计中就被加进来的。

专家：哦。是这样，当时我看到利息和 Payment History 像那样分开，还以为您们要把利息分解开来，然后把它组织成类似于 Payment History 的结构。您对应计制会计(accrual basis accounting)了解吗？

开发人员：请解释一下。

专家：我们每天，或者是根据时间表，把应计利息(interest accrual)过账到分类账中。这和付款的过账是不同的。您在这个地方把应计利息累加起来有些不合适。

开发人员：您是说，如果我们用一个列表来跟踪记录所有的“应计费用”，就能够随时对它们进行累加或者根据需要进行“过账”。



专家：应该是在应计期日(accrual date)过账。对，累加可以在任何时候进行。手续费的处理跟这个差不多，当然，它要过账到另一个分类账中。

开发人员：实际上，如果只计算一天或者一段时期的利息，问题会简单得多。然后我们就可以依据那些应计费用来完成任务了。您觉得修改后的草图(如图 9-6 所示)怎么样？

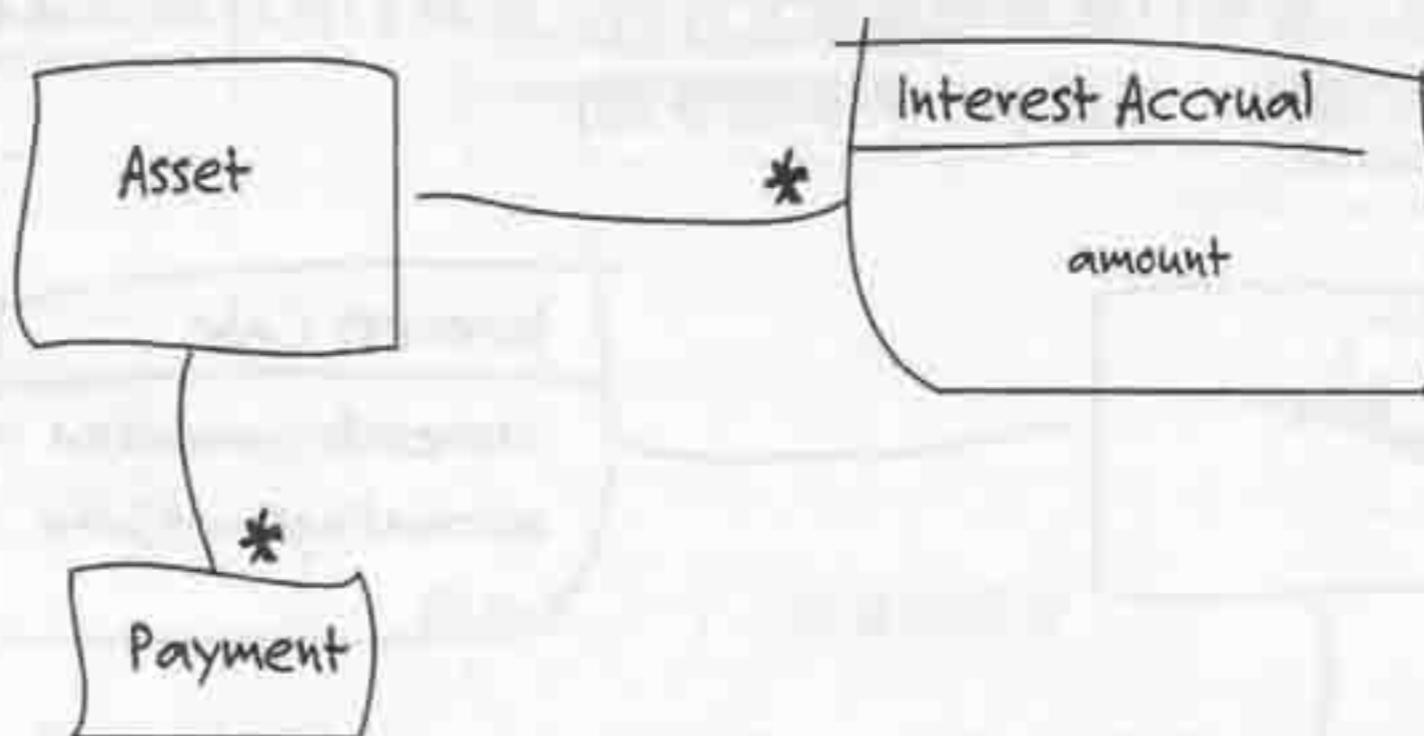


图 9-6 修改后的草图

专家：不错，看起来很好。我不知道为什么这个会对您简单一些。但是从根本上来说，资产之所以有价值，就是因为它能产生利息、手续费等。

开发人员：您说手续费是一样的？它们……是什么……过账到不同的分类账？

改进后的结果如图 9-7 所示。

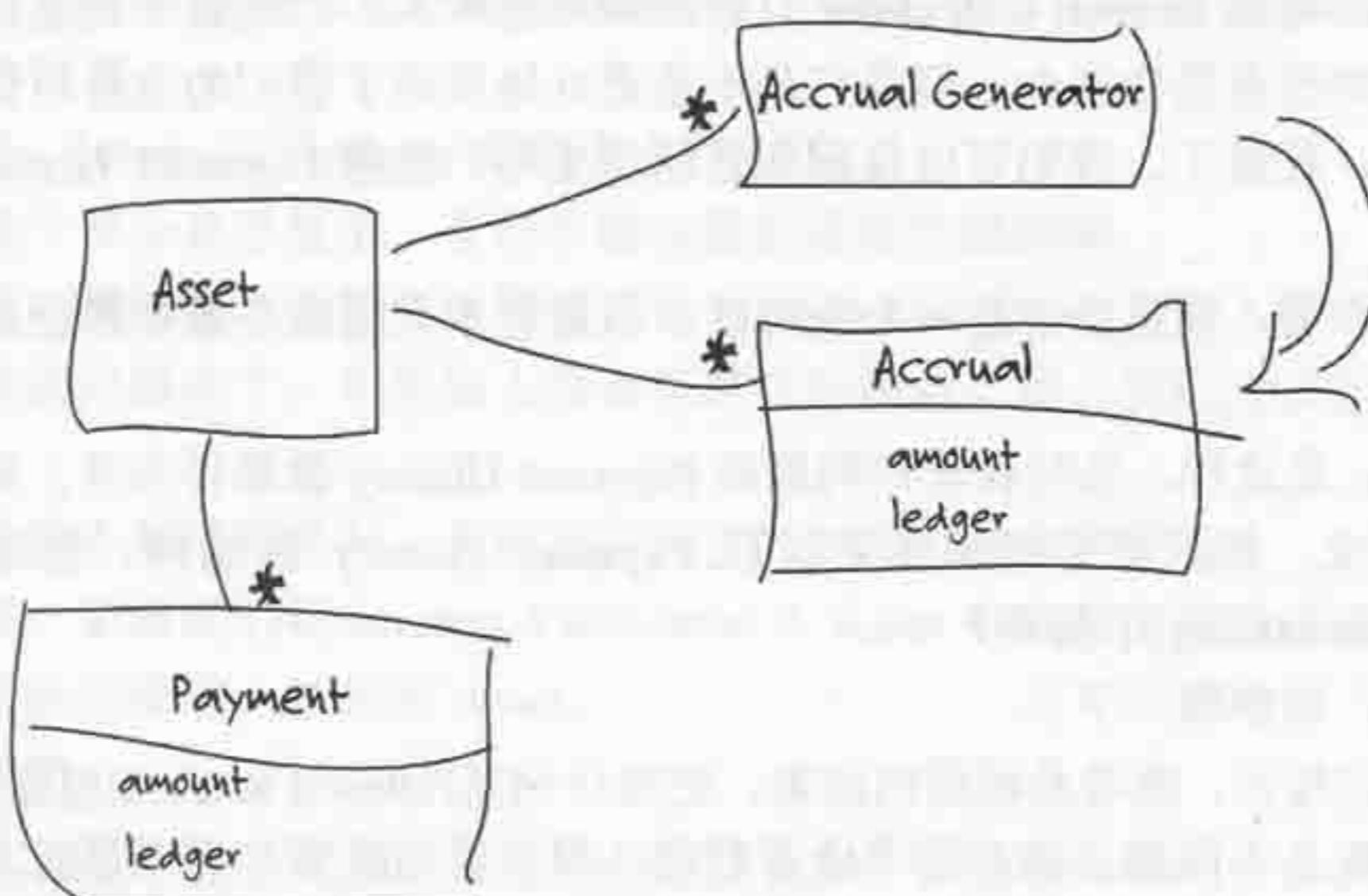


图 9-7 草图

开发人员：在这个模型中，我们把 Interest Calculator 中计算利息的逻辑，或者应该说是计算应计费用的逻辑，与跟踪这些费用的逻辑分离开来了。直到现在我才注意到，Fee Calculator 同 Interest Calculator 的功能十分雷同。此外，现在不同类型的手续费很容易就能加入了。

专家：是啊，以前的计算也是正确的，但是现在一切都能看清楚了。

由于 Calculator 类与设计的其他部分没有直接关联，因此这次重构非常容易。开发人员可以在几个小时内用新的术语重写单元测试，然后在第二天把新设计实现出来。她最终得到了图 9-8。

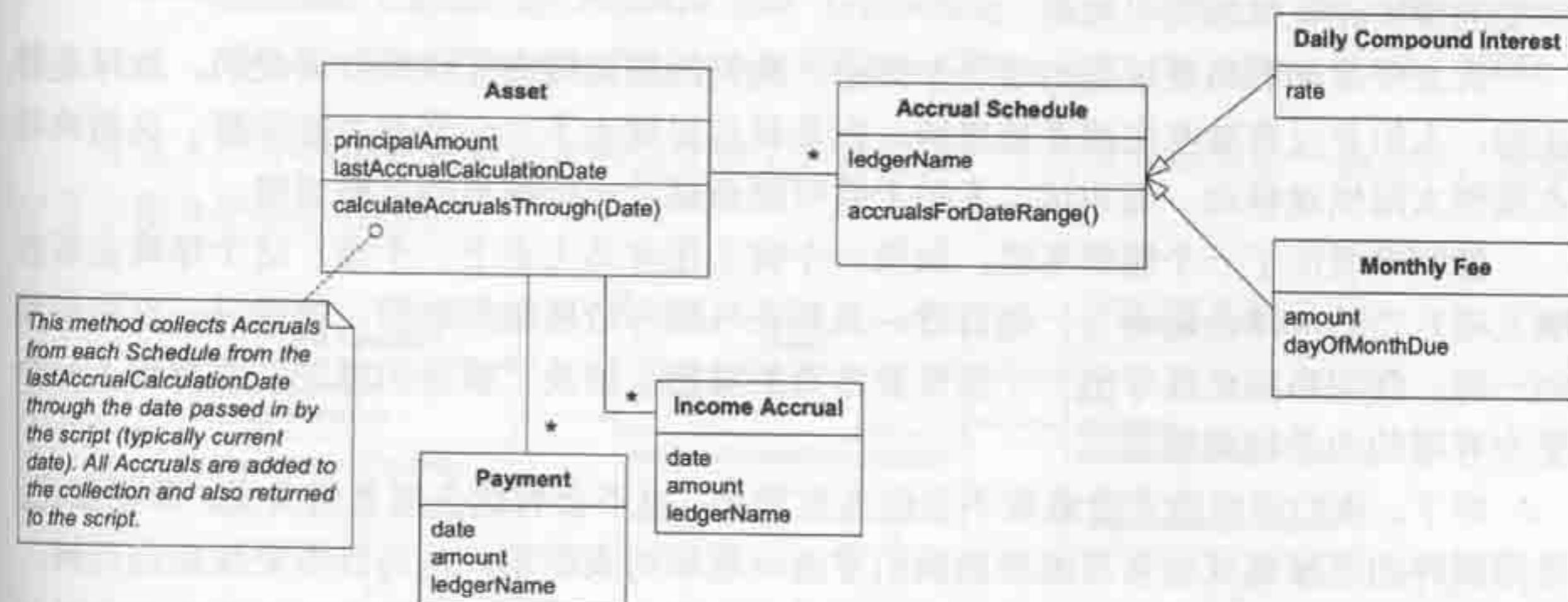


图 9-8 重构得到的更深层次模型

在重构后的应用中，nightly batch 脚本告诉每个 Asset 按日期计算应计费用 (calculateAccrualsThroughDate())。返回值是一个 Accrual 的集合，而各个金额都将过账到指定的分类账中。

新模型具有几个优点，这次重构：

- 用术语“应计费用”扩充了通用语言；
- 把应计费用与付款解耦；
- 将领域知识(如过账到哪个分类账)从脚本移到了领域层；
- 将手续费和利息统一起来，既符合业务，又消除了重复的代码；
- 提供了增加手续费和利息变种(都被视为一种 Accrual Schedule)的简单途径。

这次，开发人员不得不挖掘她需要的新概念。她看到了利息计算的不协调之处，并尽职尽责地努力找出了这个问题的深层答案。

她非常幸运地找到了一个聪明、积极的银行业专家作为自己的合作伙伴。如果专家



在合作中不那么主动的话，她在开始时可能会犯更多错误，还得更多地依赖于与其他开发人员的头脑风暴。虽然会慢一些，但还是有可能获得进展的。

9.1.3 研究矛盾之处

不同的领域专家具有各自的经验和需要，因此看待问题的方法也不尽相同。即使是同一个人提供的信息，经过仔细分析也可能会发现其中存在逻辑上的不一致。在我们深入挖掘程序需求的过程中会不断遇到这种讨厌的矛盾，它们为深层模型提供了重要线索。有些矛盾只是术语上的变化，或者是某种误解。但是，还有一些矛盾则是由于专家们所声明的事实相互抵触而引起的。

天文学家伽利略曾经提出过一个悖论。我们的感觉明白无误地告诉我们，地球是静止的：人们并没有被吹走或者被甩掉。但是哥白尼提出了一个强有力的论据，说明地球在围绕太阳快速移动。调和这二者的矛盾可能会揭示一些深奥的自然规律。

伽利略提出了一个假想实验。如果一个骑士在奔马上丢下一个球，这个球将会落在哪儿呢？当然，球会随着马一起前进，直到在马脚旁边接触到地面，就像马一直站着没动一样。伽利略据此推导出一个惯性参考系的雏形，解决了前面的悖论，并得到了一个更为有用的力学物理模型。

好了。我们遇到的矛盾通常不会这么有意思，也不会有这么深奥的含义。即便如此，采用同样的思维模式也常常能帮助我们穿透问题域的表面层，从而获得更深刻的理解。

想调和所有的矛盾是不可能的，甚至是不值得的(第 14 章研究了如何决定和应对这种结果)然而，即使将矛盾保留下，我们也应该深入地考虑为什么两种不同的声明能应用到同一个外部实体——这样的考虑能为我们带来启迪。

9.1.4 查阅书籍

在寻找模型概念时，别忽略了一些近在眼前的资源。在许多领域中，您都可以从书本中找到一些基本概念和一般常识的解释。虽然您还是必须和自己的领域专家来精炼与您的问题相关的部分，并且对那些知识消化，使之适应于面向对象的软件开发，但是看书能使您在开始时就获得一致而深刻的视角。

示例：利息图书赢利

让我们想象一下前面讨论的投资跟踪应用中可能出现的另一种场景。和前面一样，故事开始时开发人员感到设计变得越来越难以使用，特别是 Interest Calculator 这个类。但是，这次领域专家把自己的主要职责放在其他工作上，他对帮助软件开发项目可没有

多少兴趣。在这个场景里，开发人员不可能指望和领域专家来一场头脑风暴会议来帮她寻找潜藏在表象之下的被遗漏的概念。

于是，她去了书店。稍加浏览之后，她找到了一本自己比较中意的会计学入门书籍。她把书略读了一遍，发现里面有一整套系统的、定义良好的概念。其中一段文字激发了她的思考：

“应计会计制。在这种方法中，凡是本期发生的收益，无论是否已经收到现金，均作为本期的收益。凡是本期发生的费用，无论是否已经支付现金(或票据)，均作为本期的费用。所有到期债务，包括税金，都列入费用。”

——Suzanne Caplan 的 *Finance and Accounting: How to Keep Your Books and Manage Your Finances Without an MBA, a CPA or a Ph.D.*(Adams Media, 2000)

开发人员不再需要去重新发明会计学了。在与另一个开发人员讨论一番之后，她建立了如图 9-9 所示的模型。

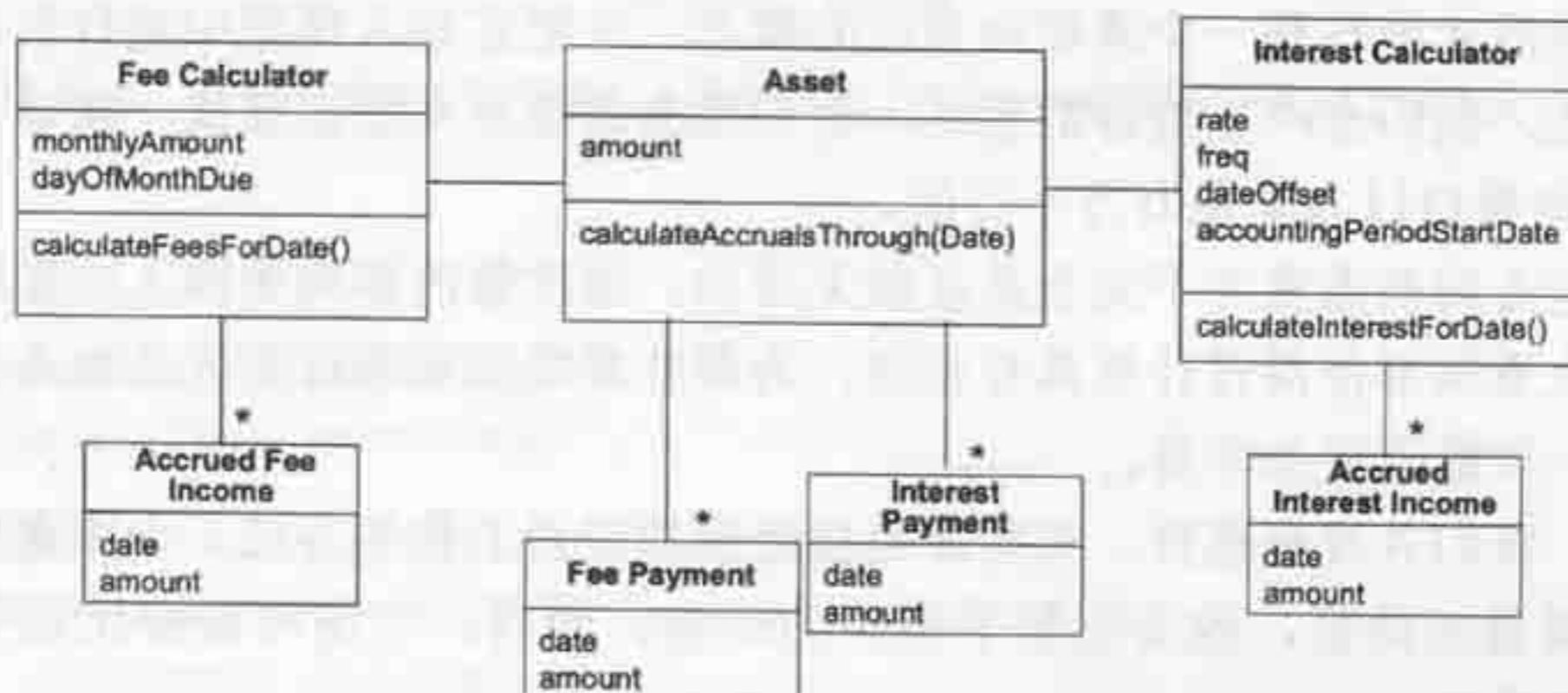


图 9-9 基于书本学习而得到的稍为深刻的模型

她还没有领会到收益是由 Asset 产生的，因此两个 Calculator 还是留在那里。分类账的知识还是在应用中而不是在它本应属于的领域层中。但是，她确实将付款和应计收入分离开来了，而这是问题最大的地方。她还将“应计”这个词引入了模型和通用语言。这个模型在以后的迭代中还会作进一步优化。

当她终于找到机会与领域专家进行讨论时，专家大吃了一惊。这是他碰到的第一个对他的工作表现出少许兴趣的程序员。由于所分配职责的关系，这个领域专家从未与她花时间坐下来讨论过她的模型，这与前一个例子中的场景截然相反。然而，由于掌握了相关的知识，开发人员可以提出更好的问题，从那以后，专家确实开始仔细倾听了，并相当卖力地迅速回答每一个问题。



当然，与领域专家交谈还是看书，这并不是一个二选一的命题。即使能从领域专家那里获得大量的支持，花点时间从文献中学习领域的理论知识也是值得的。虽然许多行业都不像会计学或者金融业这样具有精确的模型，但大部分领域都会有一些思想者对行业的通用准则进行组织和抽象。

开发人员还有一个选择，就是阅读曾经在这个领域从事过开发工作的软件专业人员的作品。例如，*Analysis Patterns: Reusable Object Models* (Fowler 1997)一书的第6章可能就会为她提供一个完全不同的方向，无论结果是好是坏。这样的阅读可能并不能提供现成的解决方案，但是可以为她提供一些全新的起点，以及在这些领域工作过的人们的经验总结，使她能够更好地开展自己的工作。这样可以使她免去重复制造轮子的工作。第11章“应用分析模式”将更深入地研究这个话题。

9.1.5 尝试，再尝试

本章给出的例子并没有演示出反复尝试的过程。在交谈中，我可能要尝试六七种不同的思路，然后才能找到一个清晰而有用的概念，并把它放入模型中进行尝试。随着经验和知识消化，我们会产生更好的想法，有些概念最终至少要被替换一次以上。一个建模/设计人员如果自以为是是万万不行的。

所有这些方向的改变并不完全是在做无用功。每次修改都将更深入的理解融入到模型之中，每次重构都使得设计更具有柔性，为那些最终需要容许变更的地方加入了灵活性，使得下一次修改更加容易。

说到底，我们并没有选择。实验才是检验模型能否工作的方法。企图避免设计上的失误会造成质量的降低，因为它基于更少的经验。而且，它很可能会比进行一系列的快速实验更加费时。

9.2 如何建模不太明显的概念

面向对象范型引导我们去寻求和创造各种概念。事物(甚至是一些像“应计费用”这样抽象的)及其行为就是对象模型的“血肉”。一些面向对象设计的入门级书籍讨论过“名词和动词”的方法。但是，其他类型的重要概念也可以显式地在模型中描述出来。

我将讨论3种这样的类型，在我初学面向对象时对它们并不熟悉。每学会一个这样的类型，我的设计水平就又进了一步。



9.2.1 显式的约束

约束构成了模型概念中特别重要的一种类型。它们通常是隐式地出现的，而将它们显式地表达出来能够极大地改进设计。

有时候，约束被很自然地隐含到一个对象或者方法中。一个“桶”对象必须保证这个不变量：桶所包含的内容不能超出它的容量，如图 9-10 所示。

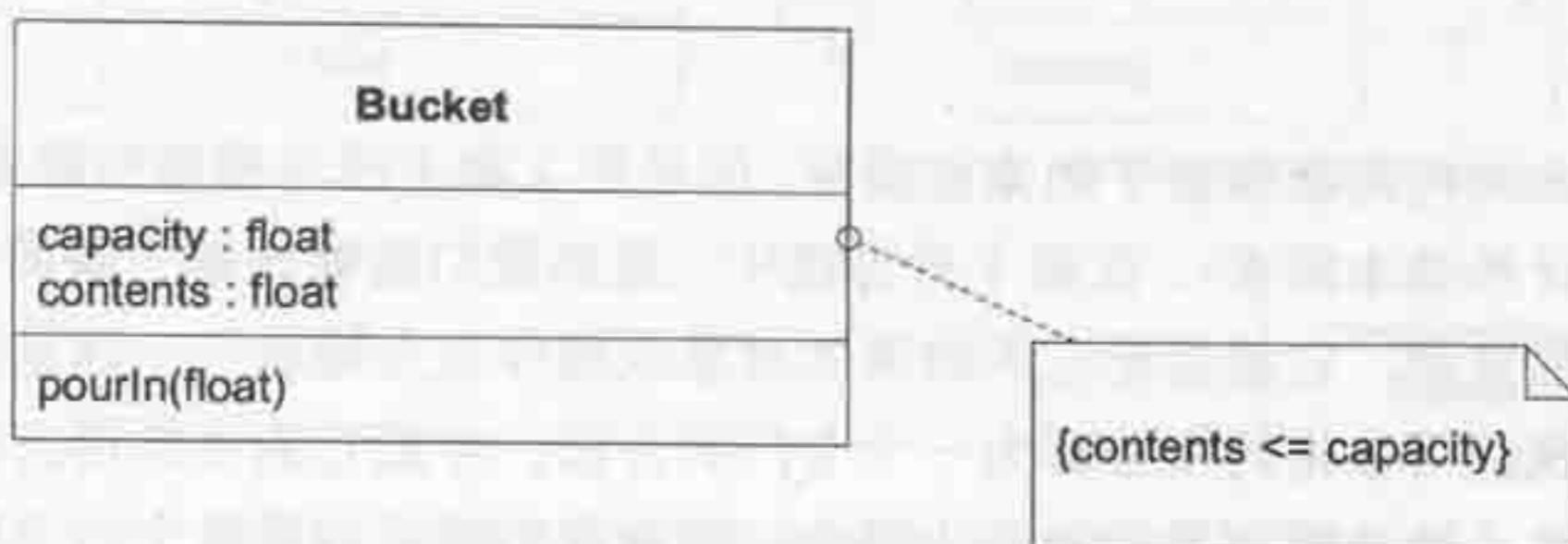


图 9-10 桶对象

为了保证这样一个简单的不变量，我们可以在每个能够改变内容的操作中使用逻辑判断：

```

class Bucket {
    private float capacity;
    private float contents;

    public void pourIn(float addedVolume) {
        if (contents + addedVolume > capacity) {
            contents = capacity;
        } else {
            contents = contents + addedVolume;
        }
    }
}
  
```

这里的逻辑非常简单，因此它代表的规则也显而易见。但是不难想象，在一个更加复杂的类中这个约束可能会丧失保证。让我们把这个约束分解到一个单独的方法中，并用方法的名称清晰而直观地表达出约束的意义。

```

class Bucket {
    private float capacity;
    private float contents;

    public void pourIn(float addedVolume) {
        ...
    }
}
  
```



```
    float volumePresent = contents + addedVolume;
    contents = constrainedToCapacity(volumePresent);
}

private float constrainedToCapacity(float volumePlacedIn) {
    if (volumePlacedIn > capacity) return capacity;
    return volumePlacedIn;
}
}
```

这两个版本的代码都保证了约束被满足，但是第 2 种方法与模型的联系更加明显(这是模型驱动设计的基本需求)。在第 1 种方法中，虽然我们能够理解一条简单的规则，但是如果规则更加复杂，它就会把它所约束的对象或操作完全掩盖——这是所有隐含概念的共同特点。我们可以将约束分解为一个专门的方法，并把它的目的用方法名清晰地描述出来，使之显式地体现到我们的设计当中。现在我们就可以用这个约束的名称来对它进行讨论了。这种方法还为约束的表达提供了空间。一条更为复杂的规则所对应的方法很可能会比方法的调用者(这里是 pourIn()方法)还长。这样，方法调用者能保持原来的简单性，并集中处理自己的工作，而约束则可以根据需要增加其复杂性。

虽然一个分离的方法可以为约束提供一些成长空间，但是在许多情况下，约束无法用一个方法轻松地描述出来。或者，即使方法非常简单，但它需要一些额外的信息，而从对象的主要职责来看，这些信息并不需要。这样的规则可能很难在一个已有的对象中找到安身之所。

下面是一些警告信号，表明约束正在误导它所属的对象的设计。

- 从对象的定义发现，判断约束所需的数据本来不应该属于这个对象；
- 相关规则在多个对象中出现，造成了代码的重复，或者使对象之间出现了本来不应出现的继承关系；
- 有许多设计和需求讨论都是围绕约束展开的，但是这些规则在实现上却被隐藏到了过程性的代码当中。

如果约束使对象的基本职责变得模糊，或者约束在领域中非常显眼而在模型中却若隐若现，那么您就可以将它分解出来，使之成为一个显式的对象，甚至将它建模为一系列对象和关系(*The Object Constraint Language: Precise Modeling with UML*[Warmer and Kleppe 1999]一书对这个问题提出了深入的、半形式的解决方案)。

示例：复查超额预约政策

在第 1 章中，我们讨论了一个运输业中的常见惯例：预订的货物量要超过额定运输



量10%(运输公司的经验表明，超额的部分可以用来补足后来被取消预订的部分，这样他们的货轮就能基本上满载航行了)。

Voyage 和 *Cargo* 之间关联的约束已经在模型和代码中显式地体现出来，我们通过加入一个新类来代表这种约束，如图 9-11 所示。

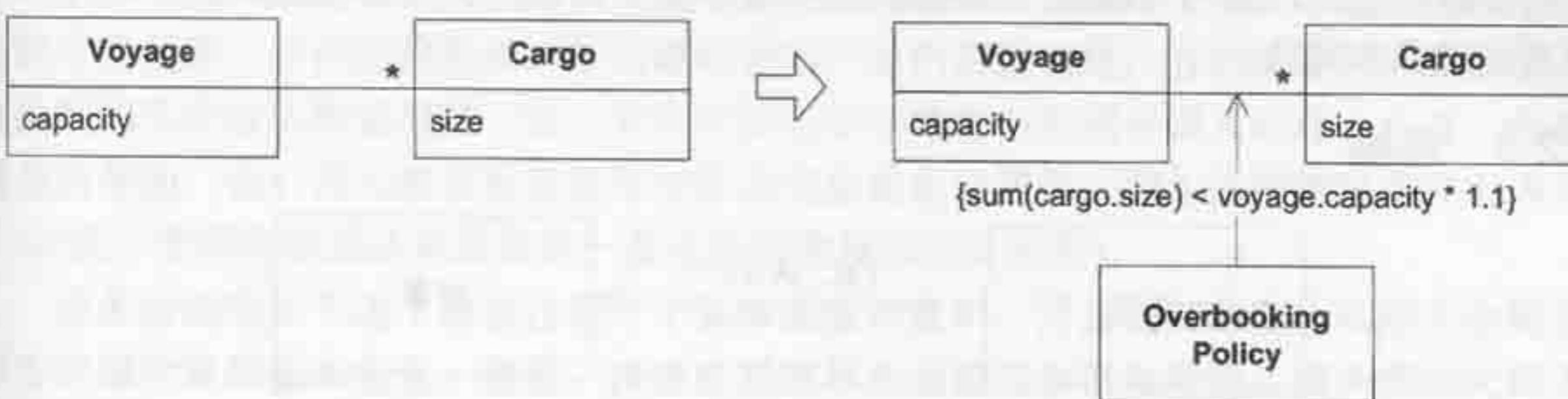


图 9-11 重构后的模型将超额预订策略显式地描述出来

要了解整个例子的代码和思考过程，可以参见第 1 章。

9.2.2 作为领域对象的流程

首先要说清楚，我们都知道要避免让过程成为模型中的一个突出方面。对象意味着将过程封装起来，而我们只需考虑对象的目的或意图。

我在这里要讨论的是存在于领域之中的流程，我们必须把它体现到模型中来。否则它们浮现出来时，往往会导致不协调的对象设计。

本章的第一个例子描述了一个用来调度货物路线的运输系统。路线处理流程中包含很多业务层面的东西。我们用一个服务来显式描述这个流程，同时将极度复杂的算法封装起来。

如果有多种方法来执行流程，那我们可以使用另一种方法：将算法本身(或者其关键部分)作为一个独立的对象。选择不同的流程也就是选择不同的对象，每个对象代表一个不同的“策略”(第 12 章将更详细地讨论在领域中使用策略的问题)。

流程是应该被显式地描述出来，还是应该被隐藏起来呢？区分的要点很简单：这个流程是领域专家所谈论的流程，还是仅仅是计算机程序机制的一部分？

约束和流程是两个涵盖范围很宽的模型概念，当我们用面向对象的语言进行编程时，这些概念并不会马上出现在我们的头脑中。但是，一旦我们开始将它们视为模型元素，它们就可以明显地深化我们的设计。

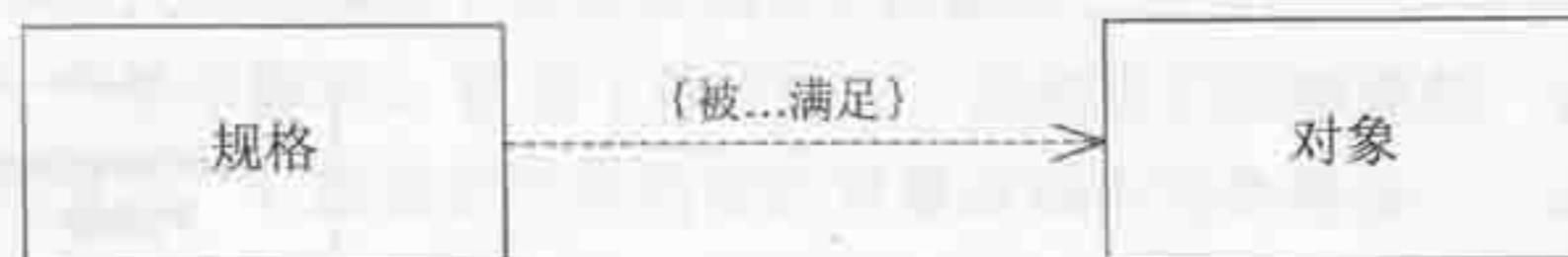
有些概念类型也非常有用，但涵盖的范围更窄。我将以一种更加特殊，但也非常常见的概念类型作为本章的结束。规格提供了一种精确地表达某些规则的方法，它将这些



规则从条件判断逻辑中抽取出来，并使之显式地体现到模型当中。

我和 Martin Fowler 一起开发了规格模式(Evans and Fowler 1997)。规格这个概念表面上非常简单，但是应用和实现起来却有些微妙，因此本节会给出大量的细节。在第 10 章，我们还会更详细地讨论这个模式，并对其进行扩充。在阅读完下面对这个模式所作的初步解释之后，您可以跳过“规格的应用和实现”一节，等到您真正想应用这个模式时再回过头来阅读。

9.2.3 规格



在各种应用中，我们都可以看到返回布尔值的测试方法。这些方法实际上是某些规则的一部分。我们可以将那些简单的规则实现为测试方法，例如 `anIterator.hasNext()` 或者 `anInvoice.isOverdue()`。在 `Invoice` 类中，`isOverdue()` 的代码就是计算一条规则(发票是否过期)的算法。例如，

```
public boolean isOverdue() {  
    Date currentDate = new Date();  
    return currentDate.after(dueDate);  
}
```

但是并非所有规则都这么简单。同样是 `Invoice` 类，另一条规则 `anInvoice.isDelinquent()` 可能也要判断发票是否已经过期(以此来判断发票是否拖欠)，但是那仅仅是这条规则要做的前期工作。宽限期政策可能是根据顾客的账户状况来确定的。有的拖欠发票可能正在准备发出第二次催款通知，而其他的则可能会被发送给某个讨债公司。此外，还要考虑顾客的付款历史、公司对不同产品线的政策等。发票本来是用来请求付款的，但是这个明显的事很快就会被大堆的规则计算代码所掩盖。同时，这还使得 `Invoice` 类依赖于许多领域类和子系统，而这些领域类和子系统并不是用来支持“请求付款”这个基本任务的。

此时，为了挽救 `Invoice` 类，开发人员往往会将规则的计算代码分解到应用层(这里是账单收集应用)。现在规则完全从领域层中分离出去了，留下一个呆板的数据对象，它没有将那些本质上应属于业务模型的规则表达出来。那些规则本来应该保留在领域层中，但把它们放到被其约束的对象(这里是 `Invoice` 对象)中又不合适。此外，规则计算方法中充斥着条件判断代码，使规则变得晦涩难懂。

使用逻辑程序范型的开发人员会用一种不同的方法处理这种情况。这种规则将被描述为“谓词(predicate)”。谓词是指计算结果为 true 或 false 的函数，它们可以通过 AND 和 OR 操作符连接起来，以表达更复杂的规则。通过谓词，我们可以将规则显式地声明出来，然后在 Invoice 中使用它们，前提是我们必须使用逻辑范型。

了解了这一点之后，人们对如何用对象来实现逻辑规则进行了尝试。这些尝试中，有的非常复杂，有的比较简单；有的雄心勃勃，有的脚踏实地；有的被证明极有价值，有的则因为实验失败被抛到一边；有些尝试甚至导致他们的项目误入歧途。只有一件事情是清楚的：他们的主要目标都是用对象来完整地实现逻辑，因为这种想法实在太诱人了(毕竟，逻辑程序设计本身就是一套完整的建模和设计范型)。

业务规则往往不适于放在任何一个实体或值对象中，而且规则的变化和组合会掩盖那些领域对象的基本含义。但是，将这些规则移出领域层会更加糟糕，因为领域代码不再能够表达模型了。

逻辑程序设计为我们提供了一种概念，即利用“谓词”来分离和组合规则对象。不过，要把这种概念完全用对象实现出来还是比较困难。而且这种概念也过于通用，在交流意图的时候它还不如一个更有针对性的设计。

幸运的是，我们并不需要完全实现逻辑程序设计，也能从中获得它的好处。大部分规则都可以划分成一些特殊情况。我们可以借鉴谓词的概念，为每种情况创建一个计算出布尔值的专门对象。那些失去控制的测试方法可以巧妙地扩充为自治的对象。这些对象对一些小的事实进行判断，并能分解到一个独立的值对象中。我们可以用这个新对象来计算其他对象，来判断它是否可以满足谓词，如图 9-12 所示。

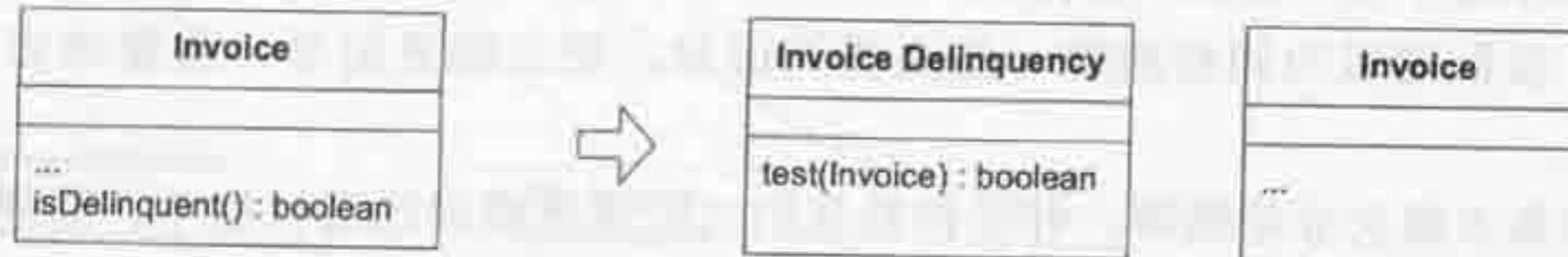


图 9-12 创建一个专门的对象

换言之，新的对象是一个规格。规格声明了施加在另一个对象(这个对象目前可能存在，也可以不存在)的状态之上的约束。规格有多种用途，其中有一种用途能够体现最基本的概念：规格可以对任何对象进行测试，看它是否符合给定的标准。

因此：

为特定目的创建显式的、类似于谓词的值对象。规格就是判断对象是否满足某些标准的谓词。



许多规格都是简单的、具有特定目的的测试，就象拖欠发票例子里的。当规则复杂时，我们还可以扩充这种概念，对简单的规格进行组合，就像用逻辑运算符将多个谓词组合起来一样(这种技术将在下一章中讨论)。在基本模式保持不变的情况下，规格的组合为我们提供了一种从简单模型中构造出复杂模型的途径。

我们可以用这种方法对拖欠发票的例子进行建模，用一个规格来声明拖欠的含义，并用它来对任意 Invoice 对象进行计算和作出判断，如图 9-13 所示。

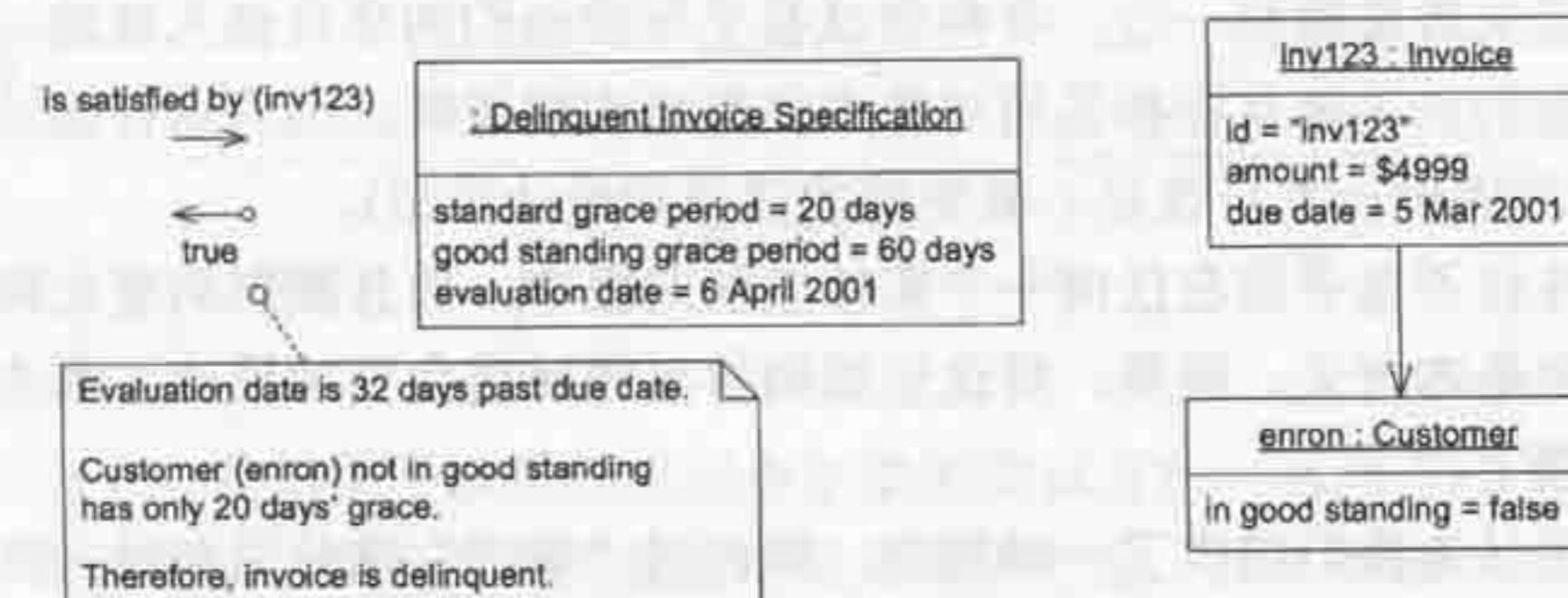


图 9-13 把一条更精细的拖欠规则分解为一个规格

规格使规则保持在领域层中。由于规则被实现为一个成熟的对象，因此这种设计能够更清晰地反映我们的模型。我们还可以用一个工厂来(在创建时)配置规格，配置数据可能来自于其他地方，例如顾客账号或者公司的策略数据库。让 Invoice 直接访问这些信息源会导致不恰当的关联，因为它们与 Invoice 的基本职责——“请求付款”是毫不相关的。在这个例子中，我们将创建 Delinquent Invoice Specification 对象，用它来评估一些 Invoice，然后抛弃它。我们只要提供一个特定的评估日期就可以把它创建出来——简化得很漂亮。我们可以为规格提供一些必要的信息，使之能更简单、直接地实现自己的功能。

规格的基本概念非常简单，并能帮助我们对领域建模问题进行思考。但是，模型驱动设计不仅要求我们的设计能够将概念表达出来，还应该能够被高效地实现。为了实现这个目标，需要更深入地研究如何应用这个模式。一个领域模式不仅仅是用 UML 图表达出来的优雅的想法，它还应该为模型驱动设计中的某个编程问题提供解决方案。

当我们的模式应用得恰当时，我们就可以深入下去，挖掘出一整套解决某种领域建模问题的思路。而且，这种寻找高效实现的经验的不断积累，也能使我们受益匪浅。下面对规格的讨论涉及到大量的细节，包括特性选择和实现方法。模式并不像菜谱那么死板，它只是让您从基本的经验出发来寻求自己的解决方案，同时也为您提供了一些语言来讨论手头的工作。

在首次阅读时，您可以把这些关键概念快速浏览一遍，以后碰到类似的情况时再回



过头来阅读这里的细节讨论，并从中汲取经验。然后，您就能着手寻找解决自己问题的方法了。

9.2.4 规格的应用和实现

规格最有价值的地方在于，它能够将一些看起来迥然不同的应用功能统一起来。出于以下3种目的之一，我们可能需要构造针对某个对象的状态的规格：

- 验证一个对象，看它是否满足某些要求或者已经准备就绪；
- 从一个集合中筛选出一个对象(如上面的例子中的查询过期发票)；
- 在创建一个对象时指定该对象必须满足的某种要求。

这3种用法(验证、筛选和按需创建)从概念层面上来说是完全一样的。如果不使用类似于规格这样的模式，同一条规则可能会表现得大不相同，甚至在形式上完全相反，从而丧失概念上的统一性。规格模式可以使我们使用一致的模型，虽然在实现上我们可能会有分歧。

1. 验证

验证是规格最简单的用法，也是能够最直接地展示其概念的用法，如图9-14所示。

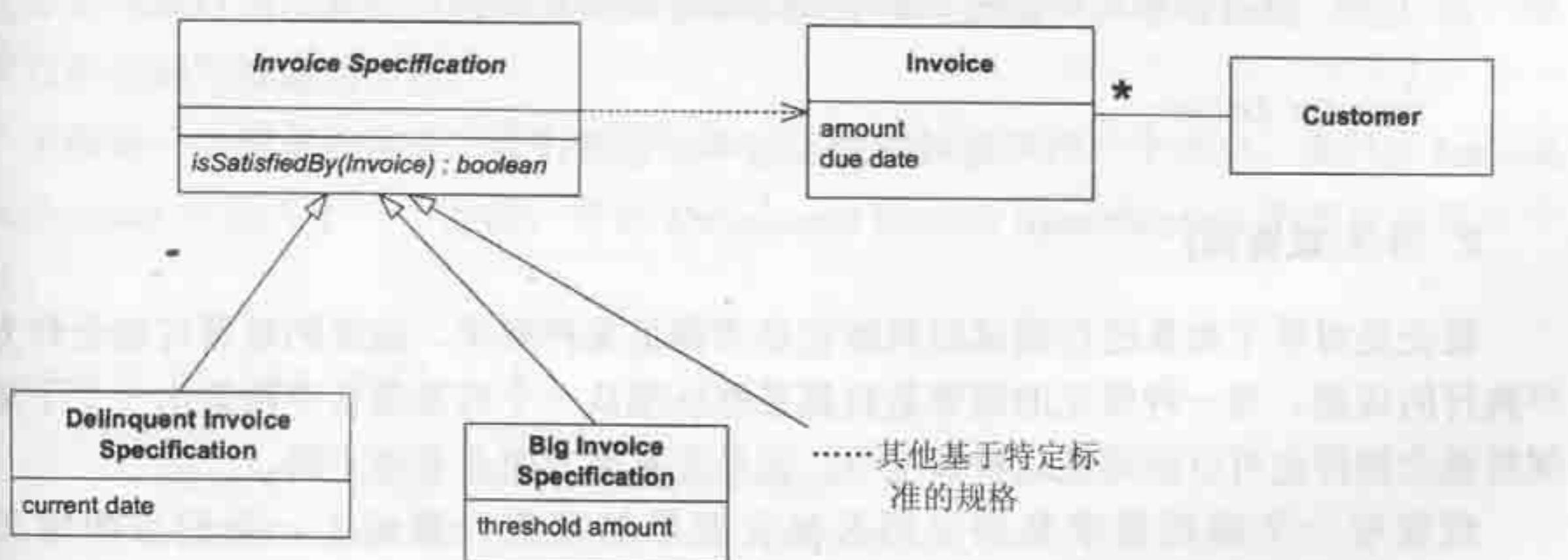


图9-14 一个用规格进行验证的模型

```

class DelinquentInvoiceSpecification extends
    InvoiceSpecification {
    private Date currentDate;
    // An instance is used and discarded on a single date

    public DelinquentInvoiceSpecification(Date currentDate) {
        this.currentDate = currentDate;
    }
}
  
```



```
public boolean isSatisfiedBy(Invoice candidate) {  
    int gracePeriod =  
        candidate.customer().getPaymentGracePeriod();  
    Date firmDeadline =  
        DateUtility.addDaysToDate(candidate.dueDate(),  
            gracePeriod);  
    return currentDate.after(firmDeadline);  
}  
}
```

现在，假设我们要实现这样的功能：当售货员打开一个欠账客户的信息时，系统会显示一个红色标记。我们只需在客户类中加入一个方法，例如：

```
public boolean accountIsDelinquent(Customer customer) {  
    Date today = new Date();  
    Specification delinquentSpec =  
        new DelinquentInvoiceSpecification(today);  
    Iterator it = customer.getInvoices().iterator();  
    while (it.hasNext()) {  
        Invoice candidate = (Invoice) it.next();  
        if (delinquentSpec.isSatisfiedBy(candidate)) return true;  
    }  
    return false;  
}
```

2. 筛选(或查询)

验证是对单个对象进行测试以判断它是否满足某种要求，验证的结果可能会作为客户执行的依据。另一种常见的需要是根据某些标准从一个对象集合中筛选出一个子集。规格概念同样也可以应用在这种情况下，但是在实现方面会有些不同。

假设有一个应用需求是列出所有拖欠发票的顾客。理论上，我们前面定义的 Delinquent Invoice Specification 也可以用于这个目的，但实际上我们可能不得不修改它的实现。为了演示二者在概念上是完全相同的，我们先假设 Invoice 的数量非常少，可能已经全部载入内存了。在这种情况下，我们完全可以直接重用实现验证功能的代码。Invoice Repository 可以用一个泛化的方法，根据规格对 Invoice 进行筛选：

```
public Set selectSatisfying(InvoiceSpecification spec) {  
    Set results = new HashSet();  
    Iterator it = invoices.iterator();  
    while (it.hasNext()) {
```



```
Invoice candidate = (Invoice) it.next();
if (spec.isSatisfiedBy(candidate)) results.add(candidate);
}

return results;
}
```

这样，客户用一行代码就能得到所有拖欠发票：

```
Set delinquentInvoices = invoiceRepository.selectSatisfying(
    new DelinquentInvoiceSpecification(currentDate));
```

上面的代码建立了隐藏在操作背后的概念。当然，Invoice 对象可能并不在内存中，它的数量有可能会达到成千上万。在典型的商业系统中，这些数据一般保存在关系数据库中。另外，我们在前面的章节中曾经指出，当与其他技术混杂使用时，模型会很容易失去方向。

关系数据库具有强大的查询能力。我们怎样才能既利用这种能力来高效地解决问题，同时又保持规格模型的清晰呢？模型驱动设计要求模型与实现保持同步，但它也给了我们选择实现的自由，只要这种实现能够如实地捕捉模型的含义。幸运的是，SQL 是一种非常自然的编写规格的方法。

下面是一个简单的例子，我们把查询和验证规则封装在同一个类中。我们在 Invoice Specification 中加入了一个方法，并在 Delinquent Invoice Specification 子类中实现这个方法：

```
public String asSQL() {
    return
        "SELECT * FROM INVOICE, CUSTOMER" +
        " WHERE INVOICE.CUST_ID = CUSTOMER.ID" +
        " AND INVOICE.DUE_DATE + CUSTOMER.GRACE_PERIOD" +
        " < " + SQLUtility.dateAsSQL(currentDate);
}
```

规格与仓储配合得非常好。仓储是一种构建机制，它提供了对领域对象的查询访问，并将数据库接口封装起来(参见图 9-15)。

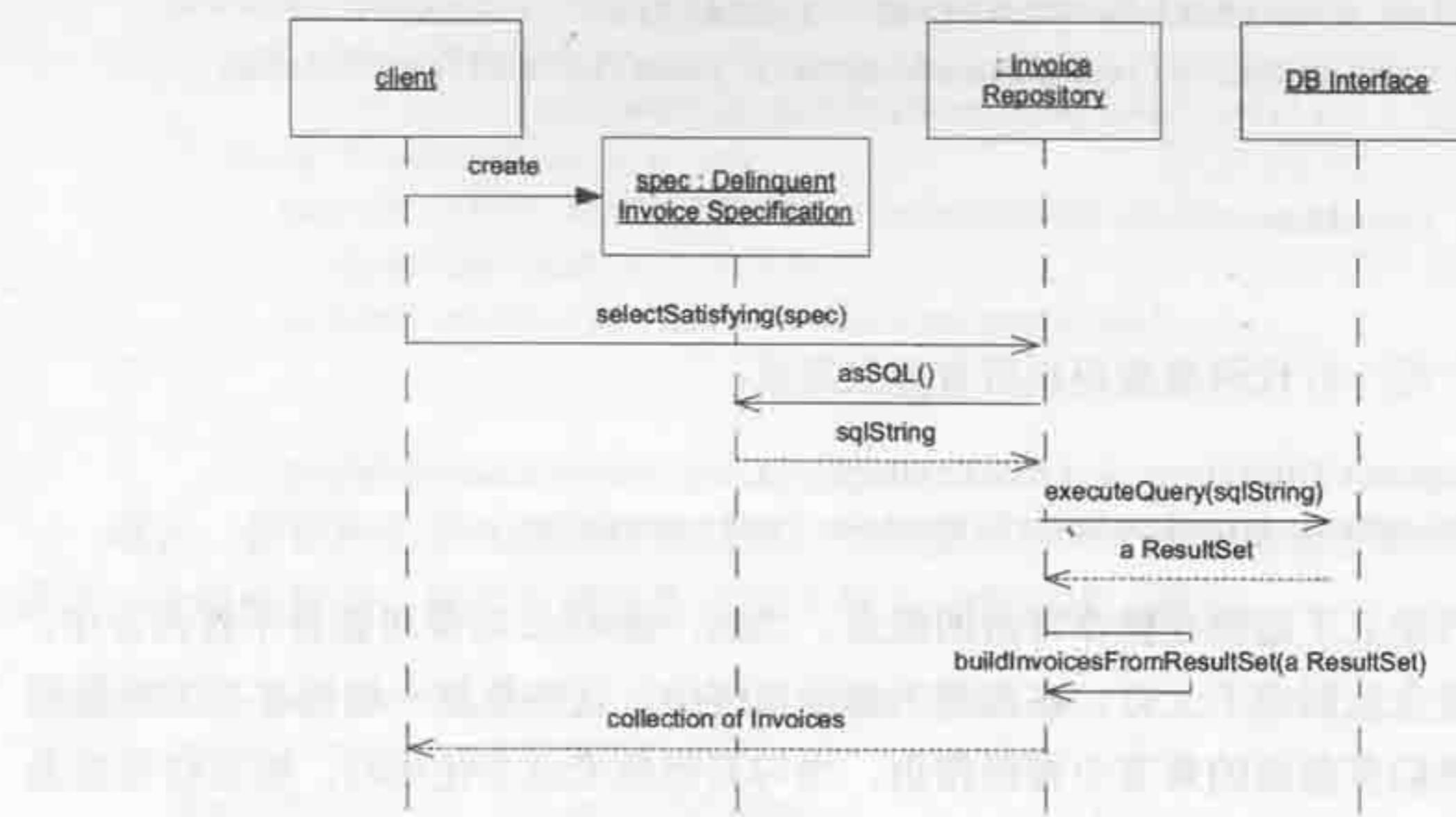


图 9-15 仓储和规格之间的交互

现在这个设计出现了一些问题。最重要的问题在于，表结构的细节被泄漏到了领域层，这二者应该通过一个将领域对象关联到关系表的映射层隔离开来。上面的代码隐式地复制了表结构的信息，这将降低 Invoice 和 Customer 对象的易修改性和可维护性，因为一旦(表与领域对象之间的)映射发生改变，我们就必须跟踪多个地方看是否要修改。但是，这个例子的目的正是为了简单演示如何把规则只放在一个地方(因此必须设法消除这种状况)。有些对象-关系映射框架提供了用模型中对象和属性来表达查询的方法，并在基础结构层中产生实际执行的 SQL 语句。如果使用这些框架，我们就能鱼与熊掌兼得了。

如果基础结构没有这个能力，我们也可以将 SQL 语句从有表达力的领域对象中分解出去，方法是在 Invoice Repository 中加入一个特化的查询方法。为了避免将规则嵌入到仓储中，我们必须用一种更通用的方法来表达查询，这种方法并不捕获规则，但是它能够被组合或者置于某种上下文之中将规则表达出来(在这个例子中使用的是双重分派(double dispatch))。

```
public class InvoiceRepository {
    public Set selectWhereGracePeriodPast(Date aDate) {
        //This is not a rule, just a specialized query
        String sql = whereGracePeriodPast_SQL(aDate);
        ResultSet queryResultSet =
            executeQuery(sql);
        ...
    }
}
```

```

        SQLDatabaseInterface.instance().executeQuery(sql);
        return buildInvoicesFromResultSet(queryResultSet);
    }

    public String whereGracePeriodPast_SQL(Date aDate) {
        return
            "SELECT * FROM INVOICE, CUSTOMER" +
            " WHERE INVOICE.CUST_ID = CUSTOMER.ID" +
            " AND INVOICE.DUE_DATE + CUSTOMER.GRACE_PERIOD" +
            " < " + SQLUtility.dateAsSQL(aDate);
    }

    public Set selectSatisfying(InvoiceSpecification spec) {
        return spec.satisfyingElementsFrom(this);
    }
}

```

Invoice Specification 中的 asSql()方法被替换为 satisfyingElementsFrom(InvoiceRepository)，在其子类 Delinquent Invoice Specification 中，这个方法的实现如下：

```

public class DelinquentInvoiceSpecification {
    // Basic DelinquentInvoiceSpecification code here

    public Set satisfyingElementsFrom(
        InvoiceRepository repository) {
        //Delinquency rule is defined as:
        // "grace period past as of current date"
        return repository.selectWhereGracePeriodPast(currentDate);
    }
}

```

上面的代码将 SQL 语句放在了仓储中，但是使用哪个查询则是由规格来控制的。规格虽然没有完全表达规则，但是表达了规则的本质声明(即超过了宽限期)。

现在仓储中包含了一个高度特化的查询(专门查找拖欠的发票)，这个函数很可能只用在这种情况下。虽然这是可以接受的，但由于过期发票的数量总是会比拖欠发票的数量多，因此我们可以作一些折衷处理，使仓储中的方法更通用，同时仍保持较好的性能，而规格也更加清晰易懂。

```

public class InvoiceRepository {

    public Set selectWhereDueDateIsBefore(Date aDate) {

```



```
String sql = whereDueDateIsBefore_SQL(aDate);
ResultSet queryResultSet =
    SQLDatabaseInterface.instance().executeQuery(sql);
return buildInvoicesFromResultSet(queryResultSet);
}

public String whereDueDateIsBefore_SQL(Date aDate) {
    return
        "SELECT * FROM INVOICE" +
        " WHERE INVOICE.DUE_DATE" +
        " < " + SQLUtility.dateAsSQL(aDate);
}

public Set selectSatisfying(InvoiceSpecification spec) {
    return spec.satisfyingElementsFrom(this);
}

public class DelinquentInvoiceSpecification {
    //Basic DelinquentInvoiceSpecification code here

    public Set satisfyingElementsFrom(
        InvoiceRepository repository) {
        Collection pastDueInvoices =
            repository.selectWhereDueDateIsBefore(currentDate);

        Set delinquentInvoices = new HashSet();
        Iterator it = pastDueInvoices.iterator();
        while (it.hasNext()) {
            Invoice anInvoice = (Invoice) it.next();
            if (this.isSatisfiedBy(anInvoice))
                delinquentInvoices.add(anInvoice);
        }
        return delinquentInvoices;
    }
}
```

上面的代码会降低一些性能，因为我们必须取出更多的 Invoice，然后在内存中对它们进行筛选。这样做更好地对职责进行了分解，但是其性能代价能否接受则完全取决于环境因素。有许多方法都可以实现规格和仓储之间的交互，不仅利用了开发平台的优势，也使得对象的基本职责适得其所。

有时候，为了提高性能或者加强安全，我们可以将查询实现为服务器上的存储过程。在这种情况下，规格只需传递存储过程允许的参数。虽然在实现上存在不同，但是它的



模型是完全一样的。凡是模型没有给出明确约束的地方，我们都可以自由选择实现的方法。使用存储过程的缺点在于编写和维护查询困难。

上面的讨论中尚未触及到如何将规格和数据库结合起来。这样做需要考虑到许多问题，在这里我就不讨论了。Mee 和 Hieatt 对设计仓储与规格时涉及到的一些技术问题进行了讨论(Fowler 2002)。

3. 按需创建(生成)

如果五角大楼需要一架新式的喷气战斗机，它会先写一个规格。规格会要求喷气机的速度达到 2 马赫，航程 1800 英里，成本低于 5 千万美元等。但是无论多么详细，规格都不是飞机的设计，更不是飞机本身。这份规格是给航空工程公司进行飞机设计用的。不同的竞争公司会提出不同的设计，并假定他们的设计能够满足规格所提出的要求。

许多计算机程序都能产生一些事物，我们也必须为这些事物指定规格。当我们把一幅图片放到字处理软件的文档中时，文字将会环绕在图片周围。我们必须指定图片的位置，以及文字环绕的风格。这样，文字处理软件才能把文字放到页面上的准确位置之上，使之符合我们指定的规格。

虽然乍看起来并不明显，但是这种规格同用于验证及筛选的规格的概念是完全相同的：我们正在为一个尚未产生的对象指定标准。但是，这种规格在实现上会有很大的不同。这种规格不是用来过滤已有对象，也不是用来测试某个已存在的对象的验证规格。这一次，我们要使用规格来创建或重新配置一整个或一批新对象。

如果不用规格，我们可以写一个生成器，通过一个过程或一系列指令来创建所需的对象。代码隐式定义了生成器的行为。

相反，如果用描述性的规格来定义生成器的接口，我们就能显式地约束生成器的产出。这种方法具有几个优点。

- 生成器的实现从接口解耦。规格声明了对输出的需求，但是没有定义如何达到这个结果；
- 接口显式地表达了它的规则，这样开发人员能够知道生成器将产生什么，而无需了解所有的操作细节。对一个已经定义好产品构造过程的生成器而言，惟一能够预测其行为的方法就是尝试各种情况，或者理解它的每一行代码。
- 接口更加灵活，或者说可以变得更具有灵活性，因为请求的声明是由客户来掌握的，生成器的职责仅仅是满足规格的要求；
- 最后，这个接口更易于测试——这一点同样重要。模型显式地定义了生成器的输入，而这个输入同时也是输出的验证。也就是说，传递给生成器接口的规格既可



以用来约束对象创建过程，又可以验证所创建的对象(如果其实现支持这一点的话)，这样我们就能保证创建对象的正确性(在第 10 章，我们将讨论一个断言的例子)。

按需创建可以是从头开始创建一个对象，也可以是对一个已经存在的对象进行配置，使之满足规格。

示例：化学品仓库打包机

有一个化学品仓库，其中各种化学品被装在类似于货车车厢的容器中。有的化学品是惰性的，随便放在什么地方都行；有的容易挥发，必须保存在特制的通风柜中；有的是爆炸性的，必须保存在特制的防爆柜中。还有一些关于在容器中如何混装化学品的规则。

我们的目的是写一个软件，来寻找一种将化学品保存到容器中的有效且安全的方法，如图 9-16 所示。

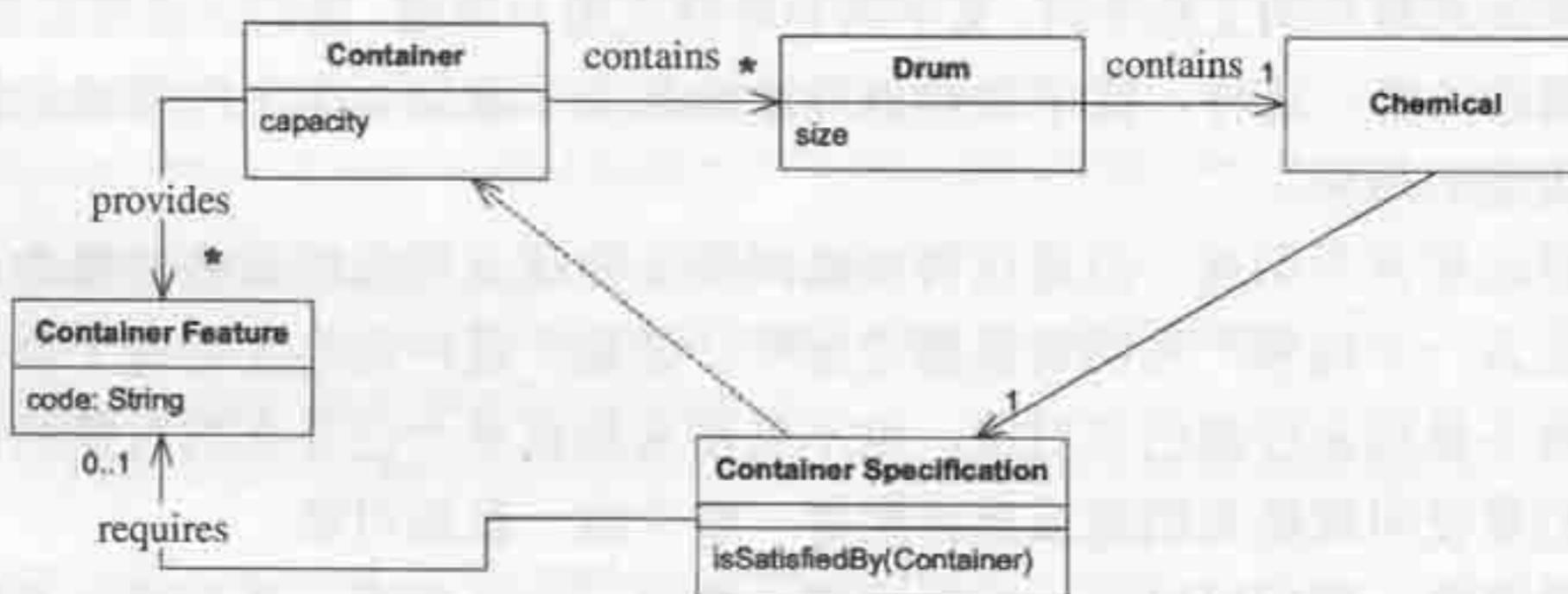


图 9-16 一个仓库存储模型

我们可以直接写一个过程来取出一件化学品，并把它放到一个容器中。但是，让我们还是从验证问题开始着手，这可以促使我们将规则显式地描述出来，也为我们提供了一种测试最终实现的方法。

每种化学品都有一个容器规格，如表 9-1 所示。

表 9-1 每种化学品都有一个容器规格

化 学 品	容 器 规 格
TNT	防爆容器
盐	
生物标本	绝不能与爆炸物混装
氨水	通风柜

现在，如果我们将这些规则实现为 Container Specifications，这样就应该能够对一个已打包容器的配置进行测试，如表 9-2 所示，看它是否能够满足这些约束。

表 9-2 测试结果

容器特性	内 容	是否满足规格
防爆	20 磅 TNT	是
	500 磅盐	
生物样本	50 磅生物样本	是
	氨水	否

Container Specification 的 isSatisfied() 方法用来检查容器是否具有所需的 ContainerFeature。例如，某种爆炸性化学品的容器规格可能需要具有“防爆”特性。

```
public class ContainerSpecification {
    private ContainerFeature requiredFeature;

    public ContainerSpecification(ContainerFeature required) {
        requiredFeature = required;
    }

    boolean isSatisfiedBy(Container aContainer) {
        return aContainer.getFeatures().contains(requiredFeature);
    }
}
```

下面是用来设置爆炸性化学品的客户代码：

```
tnt.setContainerSpecification(
    new ContainerSpecification(ARMORED));
```

Container 对象上的 isSafelyPacked()方法用来保证 Container 具有保存在该对象中的 Chemical 指定的所有特性。

```
boolean isSafelyPacked() {
    Iterator it = contents.iterator();
    while (it.hasNext()) {
        Drum drum = (Drum) it.next();
        if (!drum.containerSpecification().isSatisfiedBy(this))
            return false;
    }
}
```



```
        return true;  
    }
```

现在，我们可以写一个监控应用来对存货数据库进行监控，并报告存在的不安全隐患。

```
Iterator it = containers.iterator();  
while (it.hasNext()) {  
    Container container = (Container) it.next();  
    if (!container.isSafelyPacked())  
        unsafeContainers.add(container);  
}
```

但是这并不是要求我们编写的软件。让业务人员知道这个程序当然很好，但是我们的任务是设计一个打包机(packer)。现在我们得到的是一个打包机的测试程序。利用这些对领域的理解和基于规格的模型，我们可以为服务定义一个清晰简单的接口，让服务接收 Drum 和 Container 的集合，然后将它们按照规则打包。

```
public interface WarehousePacker {  
    public void pack(Collection containersToFill,  
                     Collection drumsToPack) throws NoAnswerFoundException;  
  
    /* ASSERTION: At end of pack(), the ContainerSpecification  
     * of each Drum shall be satisfied by its Container.  
     * If no complete solution can be found, an exception shall  
     * be thrown. */  
}
```

现在，我们的任务就是要设计一个优化的约束求解算法，来完成 Packer 服务的职责。Packer 已经从应用的其他部分解耦了，因此其中的实现机制不会对这个模型的设计产生影响(参见第 10 章中的“声明风格设计”和第 15 章中的“内聚机制”)。然而，打包的控制规则并没有从领域对象中抽取出来。

示例：仓库打包机的工作原型

编写优化逻辑使得仓储打包软件转起来是一项艰巨的工作。已经组织了一小组开发人员和业务专家来着手研究这个问题，但是尚未开始编码。同时，另一个小组正在开发一个应用，该应用允许用户从数据库中取出存货数据并提交给 Packer 处理，并展示打包结果。这个小组正在力图为预期的 Packer 进行设计，但是，他们能做的也只是实现一个用户界面原型，以及一些数据库集成代码。他们无法为用户提供一个具有实际含义的界

面，因此也不能从用户那里得到有效的反馈。同理，Packer 小组也是在闭门造车。

利用在前面的例子中得到的领域对象和服务接口，应用团队认识到他们可以实现一个非常简单的 Packer，来帮助他们继续自己的开发过程，并使得两个小组的工作能够并行地开展下去。同时，用户反馈循环也可以建立起来了——用户反馈必须要有一个能运转起来的端到端系统才能真正产生效果。

```
public class Container {  
    private double capacity;  
    private Set contents; //Drums  
  
    public boolean hasSpaceFor(Drum aDrum) {  
        return remainingSpace() >= aDrum.getSize();  
    }  
  
    public double remainingSpace() {  
        double totalContentSize = 0.0;  
        Iterator it = contents.iterator();  
        while (it.hasNext()) {  
            Drum aDrum = (Drum) it.next();  
            totalContentSize = totalContentSize + aDrum.getSize();  
        }  
        return capacity - totalContentSize;  
    }  
  
    public boolean canAccommodate(Drum aDrum) {  
        return hasSpaceFor(aDrum) &&  
            aDrum.getContainerSpecification().isSatisfiedBy(this);  
    }  
}  
  
public class PrototypePacker implements WarehousePacker {  
    public void pack(Collection containers, Collection drums)  
        throws NoAnswerFoundException {  
        /* This method fulfills the ASSERTION as written. However,  
         * when an exception is thrown, Containers' contents may  
         * have changed. Rollback must be handled at a higher  
         * level. */  
  
        Iterator it = drums.iterator();  
        while (it.hasNext()) {  
            Drum drum = (Drum) it.next();  
        }  
    }  
}
```



```
Container container =
    findContainerFor(containers, drum);
container.add(drum);
}

public Container findContainerFor(
    Collection containers, Drum drum)
    throws NoAnswerFoundException {
Iterator it = containers.iterator();
while (it.hasNext()) {
    Container container = (Container) it.next();
    if (container.canAccommodate(drum))
        return container;
}
throw new NoAnswerFoundException();
}

}
```

不可否认，上面的实现代码还存在许多缺陷。它可能会将盐打包到特殊容器中，结果后面要打包危险化学品时也许会发现空间不够。显然，它没有考虑利润优化的问题。但是，许多优化问题无论怎样都无法解决得十全十美。这个实现确实遵循了我们在前面声明过的规则。

通过工作原型来摆脱开发僵局

这个团队必须等另一个团队的代码写出来后才能继续工作；两个团队都必须等到代码完全集成之后才能测试他们的组件，或者获得用户反馈——这种阻塞可以通过建立关键组件的模型驱动原型来缓解(原型并不需要满足所有需求)。如果实现从接口解耦，那么不管是什么实现，只要能运行起来，就能有灵活性，使开发工作并行地开展下去。待时机成熟之后，我们再把原型替换为更有效的实现。同时，在开发过程中，原型也为系统的其他部分提供了可供交互的对象。

有了这个原型，应用程序开发人员就可以全速前进了，所有与外部系统集成的工作也在顺利进行。这个原型还使 Packer 开发团队能够获得领域专家的反馈，因为他们现在可以与原型交互以确认自己的想法，这有助于澄清各项需求及其优先顺序。Packer 团队决定接管这个原型，并对它进行调整以测试各种想法。

他们还随时保持接口与最新设计的同步，以推动应用和一些领域对象的重构。这样能够尽早地解决集成问题。

当复杂的 Packer 最终就绪的时候，代码集成也就轻而易举了，因为 Packer 是根据一



个定义精良的接口来编写的——这个接口同时也是交互系统的实现依据。应用根据它和断言来编写并与原型交互。

专家们花了数月的时间才得到一个正确的优化算法。用户在与原型进行交互时提出的反馈意见给了他们很大的帮助。同时，原型也为系统的其他部分提供了在开发期间可供交互的对象。

这里，我们用例子演示了一个“可工作且最简单的东西”。由于模型更加精巧，因此最终是可以实现的。我们可以用二三十行易于理解的代码来实现一个复杂组件的原型。如果不使用模型驱动的方法，那么原型会更难懂，而且更难以升级(因为 Packer 与设计的其他部分可能会关联得更紧密)。在这样的情况下，开发原型可能会更加费时。