

第一部分

让领域模型发挥作用



这张表示 18 世纪中国的地图表示整个世界。中间占据了绝大多数空间的部分是中国，周围寥寥几笔勾勒了其他的国家。这是适用于那个社会的世界模型，它有意地偏重于内部。这张地图表现的世界视图对于外国人是没有太大用处的，当然它也无法适用于现代中国。地图就是模型，每个模型都代表了我们所感兴趣的现实或观点的某些方面。模型是一种简化，它对现实进行阐述，只是抽象出与解决手头问题有关的方面而忽略掉无关的细节问题。



每个软件程序都会与其用户的活动或兴趣相关。用户在其中使用程序的主要环境称为软件的领域(domain)。一些领域会涉及到物质世界：航线预订程序的领域涉及到现实中登机的人。一些领域是无形的：财务程序的领域就是货币和金融。软件领域通常很少与计算机有关系，然而也有例外：源代码控制系统的领域就是软件开发本身。

要建立对于用户活动有价值的软件，开发团队必须瞄准与这些活动相关的知识主体。所要求的知识广度可能令人望而生畏，信息的容量和复杂度也是令人难以想象的。而模型正是处理这种过载负担的工具。模型是知识的一种有选择的简化和有意识的组织形式。一个合适的模型能够了解信息的含义并聚焦于问题本身。

领域模型并不是某种特殊的图(diagram)，而是图所要表达的思想。它并不仅仅是某个领域专家头脑中的知识，而是对相关知识进行严格的组织与选择性抽象。一个图能够描绘并传达一个模型，同样，认真编写的代码以及英语句子都可以做得到。

领域建模并不是尽可能地制作一个逼真的模型。即使在一个可触及的真实世界事物的领域中，我们的模型只是一个仿真的创造物。它也不是仅仅给出必要结果的软件机制的构建。它更像是电影制作，松散地表现具有特定目的的现实。即使是纪录片也不会展示未经修饰过的实际生活。就像一个电影制片人选择经验中的方方面面并将它们用一种特殊的方式展现出来，告诉大家一个故事或一个论点，领域建模人员也是如此。

领域驱动设计中模型的作用

在领域驱动设计中，模型的选择取决于 3 个基本的用途：

- **模型与设计核心的相互塑型。**正是模型与实现之间密切的联系使得模型与现实相关并且保证对于模型的讨论分析能够应用于最终产品——可运行的程序。这种模型与实现之间的绑定对于软件的维护和继续开发也很有帮助，因为可以根据对模型的理解来解释代码(参见第 3 章)。
- **模型是所有团队成员所使用语言的核心。**由于模型与实现是相互绑定的，因此开发人员可以用这种语言来讨论程序。他们能够在没有翻译的条件下与领域专家交流。又由于这种语言是基于模型的，我们的自然语言能力也能够用来细化模型本身(参见第 2 章)。
- **模型用来提炼知识。**模型是团队在组织领域知识和辨别最感兴趣的原理时一致同意的方式。在我们选择术语、分解概念并将它们相互联系起来时，模型能够反映出我们是怎样考虑领域问题的。开发人员与领域专家将信息放置于模型这种形式



中，这样，公用的语言可以使他们的合作更加高效。模型与实现之间的绑定使得在反馈到建模过程时，软件前期版本的一些经验也同样适用(参见第 1 章)。

下面的 3 章将着手考察这些影响的意义和价值，以及它们相互关联的方式。通过这些方式使用模型能够给具有丰富功能的软件的开发提供很大帮助，否则可能会需要巨大的投资。

软件的核心

软件的核心是它为用户解决领域相关问题的能力。其他的一些特征，尽管它们也许是必需的，但也是用来支持这个核心目的的。当领域非常复杂的时候，这个任务将非常艰巨，需要有才能和技术的人们共同努力。开发人员需要进入领域之中补充业务知识，他们必须磨练建模技巧来掌握领域设计。

然而，这并不是大多数软件项目要优先考虑的方面。大多数开发人员都不大愿意学习他们所处理的特定领域的知识，更少有人愿意承诺去提高他们的建模技巧。技术人员则喜欢能够锻炼他们技术能力的可计量问题。领域相关的工作非常繁杂，并且需要很多难懂的新知识，而这些并不是计算机科研工作者能力范围内的。

技术人员应该在精心描述的框架下工作，用技术解决领域问题。学习建模和领域的任务留给其他人。必须首先处理软件核心的复杂性问题，否则可能会偏离初衷。

在一次电视谈话节目的采访中，喜剧演员 John Cleese 讲述了在拍摄电影 *Monty Python and the Holy Grail* 时发生的一件事。他们反复拍摄一个特殊的场景，但是不知为什么总觉得不是很有趣。最后，他休息了一会，与同事的喜剧演员 Michael Palin 商量并提出了一个细微的改变。他们又进行了一次拍摄，最后达到了预期的效果并顺利收工。

第二天早上，Cleese 观看影片剪辑员作的对前一天工作的初步剪辑结果。当看到他们昨天一直研究的那个场景时，Cleese 发现它很没有趣味，因为使用的是早些时候的一个镜头。

他问影片剪辑员为什么没有使用指定的最后一个镜头。剪辑员回答说：“不能使用它，拍摄时有人走动。” Cleese 一遍又一遍地观看这个场景，始终没有发现有什么地方不对。最后，剪辑员暂停了影片，指出图片的边上能够看到一个外套的袖子。

影片剪辑员注意的是他自己专业方面的内容。他想到的是其他影片剪辑员看到电影时会从技术完美的角度来评价他的工作。在这个过程中，场景的核心问题丢掉了。

幸运的是，这个有趣的场景被通晓喜剧的导演还原了。同样的道理，当模型开发偏离了正确走向时，团队中理解领域中心问题的领导者能够使得它归位。



本书将说明领域开发能够培养复杂的设计技巧。大多数软件领域的杂乱性实际上是一种有趣的技术挑战。事实上，在许多科学学科中，当研究者着手处理现实世界的繁杂时，“复杂性”是最令人兴奋的问题。当面对一个从未被形式化过的复杂领域时，一个软件开发人员也会有同样的期望。创建一个清晰易懂能够简洁地解决其复杂性的模型是一件令人兴奋的事情。

开发人员可以使用系统化的方式来寻找并创建有效的模型。对于杂乱的软件应用程序，设计方法能够使其有序发展。这些技巧的培养可以使得一个开发人员更有价值，即使在一个最初并不熟悉的领域。

在开始学习如何使用领域驱动设计之前，我们先来回顾一下什么是领域。领域是软件工程的一个子集，它关注的是软件系统所处理的业务逻辑。领域驱动设计（Domain-Driven Design，简称 DDD）是一种设计方法，它通过识别和建模核心概念来解决复杂的业务逻辑问题。领域驱动设计强调通过分析核心概念来构建模型，从而提高系统的可维护性和可扩展性。这种方法的核心思想是：通过深入理解业务领域，我们可以构建出更符合业务需求的系统。领域驱动设计通常与持续集成、重构等实践结合使用，以确保系统能够随着业务需求的变化而不断进化。领域驱动设计特别适用于处理高度动态、变化频繁的业务逻辑，如金融交易系统、医疗保健系统等。通过将业务逻辑抽象为领域模型，领域驱动设计可以帮助开发者更好地理解业务需求，从而构建出更可靠、更高效的软件系统。

第1章

消化知识

几年前，我曾经着手设计一个用于设计印刷电路板(printed-circuit board, PCB)的专用软件工具。我并不了解任何关于电子硬件方面的知识。为此，我拜访了一些 PCB 设计人员，然而他们在 3 分钟内就让我晕头转向。那么我应该了解多少才能够开始编写这个软件呢？我当然不敢奢望在交付期限到达之前变成一个电子工程师。

我们试着让 PCB 设计人员精确地告诉我们软件所需要完成的工作，然而这么做并不是一个好主意。他们是非常优秀的电路设计人员，但是他们的软件概念通常涉及到阅读一个 ASCII 文件，将其分类并加入注释，然后做成一个报告。这样做显然不能够取得他们所希望的生产率的飞跃。

前几次的会议让人气馁，但是在他们要求的报告中也有一些闪光点。他们总是谈及 net 和其中的各种细节问题。在这个领域中，net 是可以连接 PCB 中任意数目元件(component)并将电子信号(signal)传送到它所连接的任何元件的金属导线。我们得到了领域模型的第一个元素如图 1-1 所示。



图 1-1 模型的第一个元素 net

在讨论他们希望软件能做工作的时候，我开始为他们画图。我使用了一种非正式的对象交互图来对情景进行初步描绘，如图 1-2 所示。

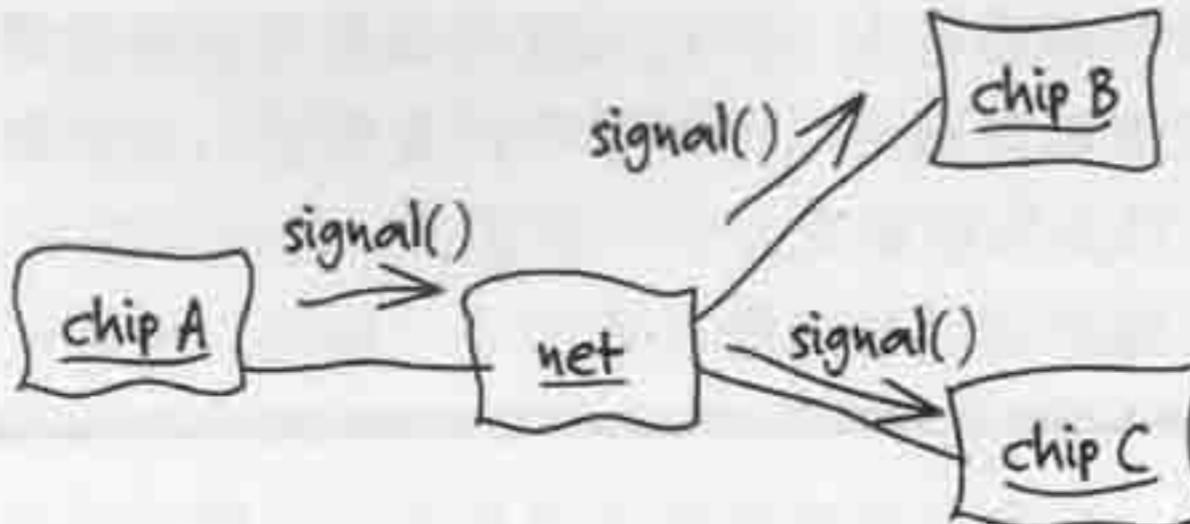


图 1-2 非正式的对象交互图

PCB 专家 1：元件不一定必须是集成电路(chip)。

开发人员(我)：那么我们就只把它们称为“元件”怎么样？

专家 1：我们叫它们“元件实例”(Componet instance)，因为可能有很多相同的元件。

专家 2：net 方框看起来就像一个元件实例。

专家 1：他没有使用我们的符号。我猜想每样东西都会是一个方框。

开发人员：抱歉，是这样的。我想我应该更清楚地解释这个符号。

他们经常修正我的一些工作，在这个过程中我开始学习相关的知识。我们消除了他们在术语和技术主张中有差异或含糊不清的地方，他们也从中学到了东西。他们现在能更一致和精确地对事物进行解释，我们开始共同开发一个模型：

专家 1：说一个信号到达一个 ref-des 是不够的，我们还必须知道引脚(pin)。

开发人员：ref-des？

专家 2：就是一个元件实例。ref-des 是我们在使用一种特殊工具时所使用的名称。

专家 1：总之，net 是将一个实例的特定引脚与另一个实例的引脚相连。

开发人员：我们可以说一个引脚只属于一个元件实例并且只与一个 net 相连吗？

专家 1：是的。

专家 2：并且，每个 net 都有一个拓扑结构，用来决定 net 元素连接的方式。

开发人员：好的，这样画如何？

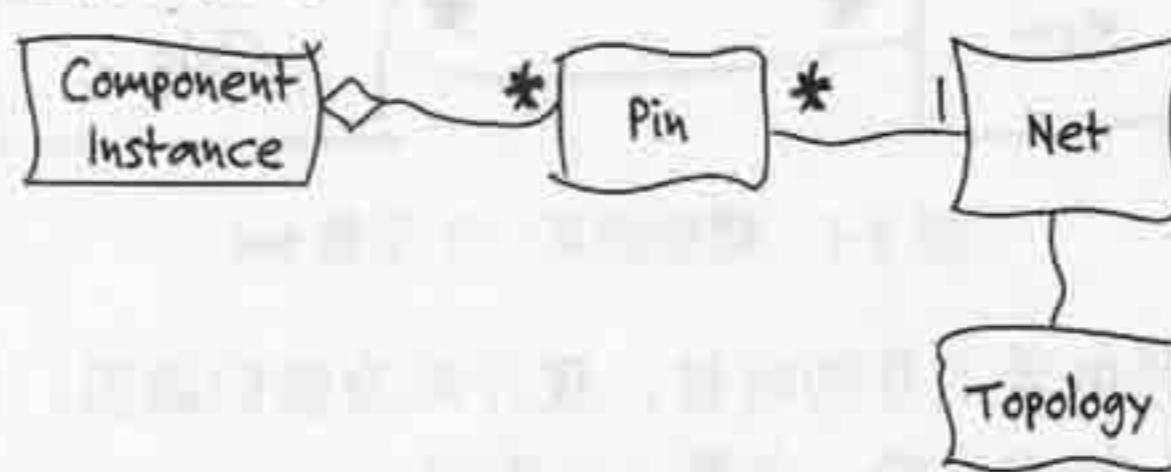


图 1-3 修改后的图

为了让我们的讨论更加集中，我们在一段时间内集中研究一个细节特征。“探针模拟”



能够跟踪一个信号的传播，这样可以探测设计中可能涉及到的问题的各个方面。

开发人员：我已经明白了信号是如何被 net 传送到附属的引脚上的，但是它们怎样可以传送得更远呢？拓扑结构和它有关系吗？

专家 2：没有。是元件推动信号继续前进。

开发人员：我们肯定不能对一个集成电路的内部行为进行建模，那样太复杂了。

专家 2：我们不需要这么做。我们可以使用一种简化的方式来表示通过一个元件从某些引脚将信号推动(push)到其他引脚。

开发人员：是这样吗？

[经过反复的尝试和挫折，我们最后共同绘制了一个草图，如图 1-4 所示。]

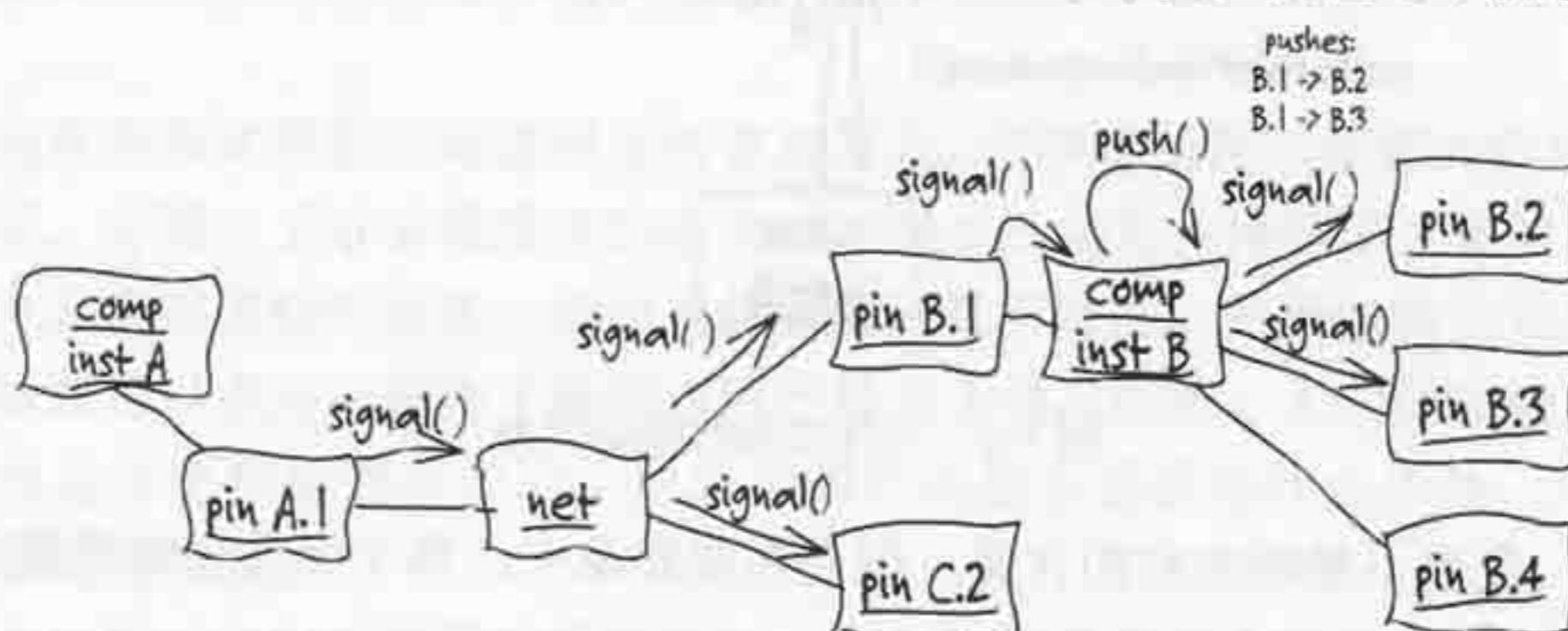


图 1-4 最终草图

开发人员：但是您们究竟需要从计算中得到哪些东西呢？

专家 2：我们需要寻找一些较长的信号延迟——就是说，任何超过 2 或 3 跳的信号路径。这是一个经验法则。如果路径过长，信号可能不会在时钟周期内到达。

开发人员：超过 3 跳……因此我们需要计算路径的长度。那么怎样算作一跳呢？

专家 2：信号每经过一个 net，称为一跳。

开发人员：那么我们可以一直传递跳数，net 可以递增跳数，就像图 1-5 中所示。

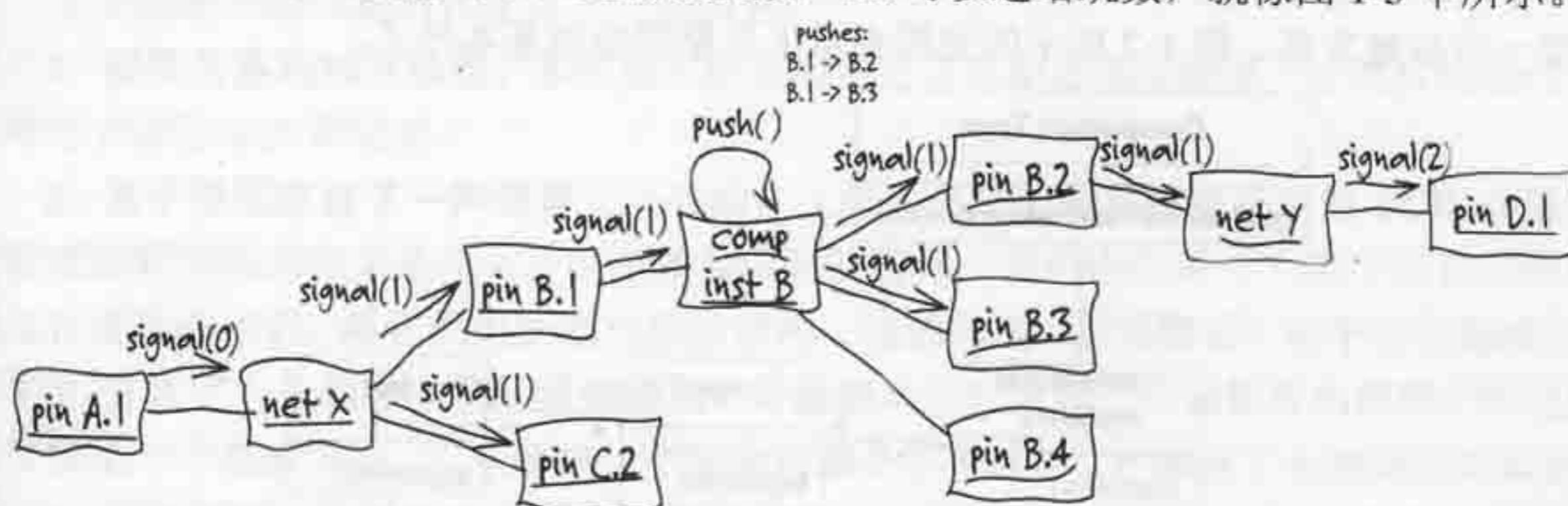


图 1-5 细化后的图

开发人员：我惟一不清楚的部分是这种“推动”来自什么地方呢？我们要为每个元件实例存储那些数据吗？

专家2：一个元件所有实例的推动行为都是一样的。

开发人员：那么元件的类型将决定推动的方式。对于每个实例都是一样的吗？

图1-6表述了元件实例的推动行为。

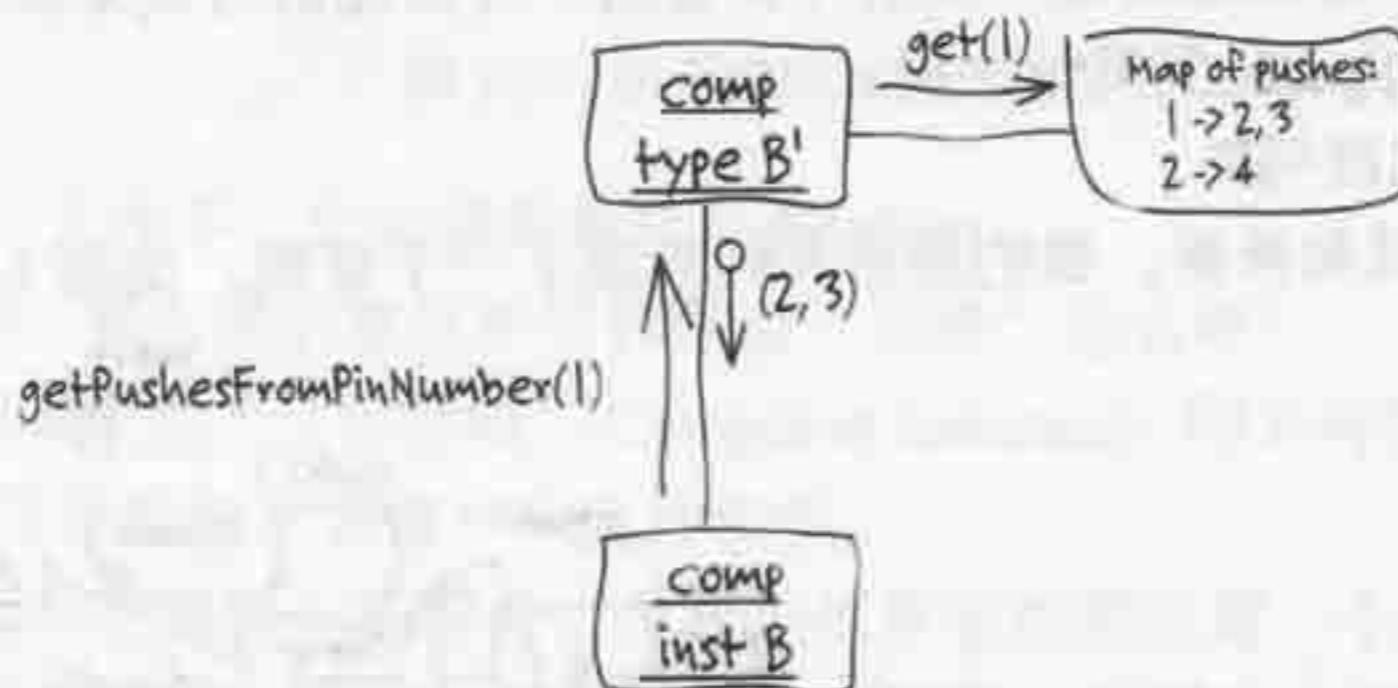


图1-6 元件实例的推动行为

专家2：我也不能确定它的含义，但是在我想象中，每个元件存储的推动方式应该就是这个样子。

开发人员：对不起，我问得太过细节了，我只是想把它理解得更透彻一些……那么现在的问题是，拓扑结构与它有什么关联呢？

专家1：探针模拟是不需要拓扑结构的。

开发人员：那么我现在可以不对它进行研究，对吗？等我们接触到那些特征时再来讨论它。

这样就完成了模型的建立(实际会有更多的问题)。在自由讨论中进行细化，在询问下进行解释。随着我对领域的理解和他们对于模型如何在问题解决中发挥作用的理解，模型一步步地发展。图1-7所示的类图表示了早期模型的基本形式。

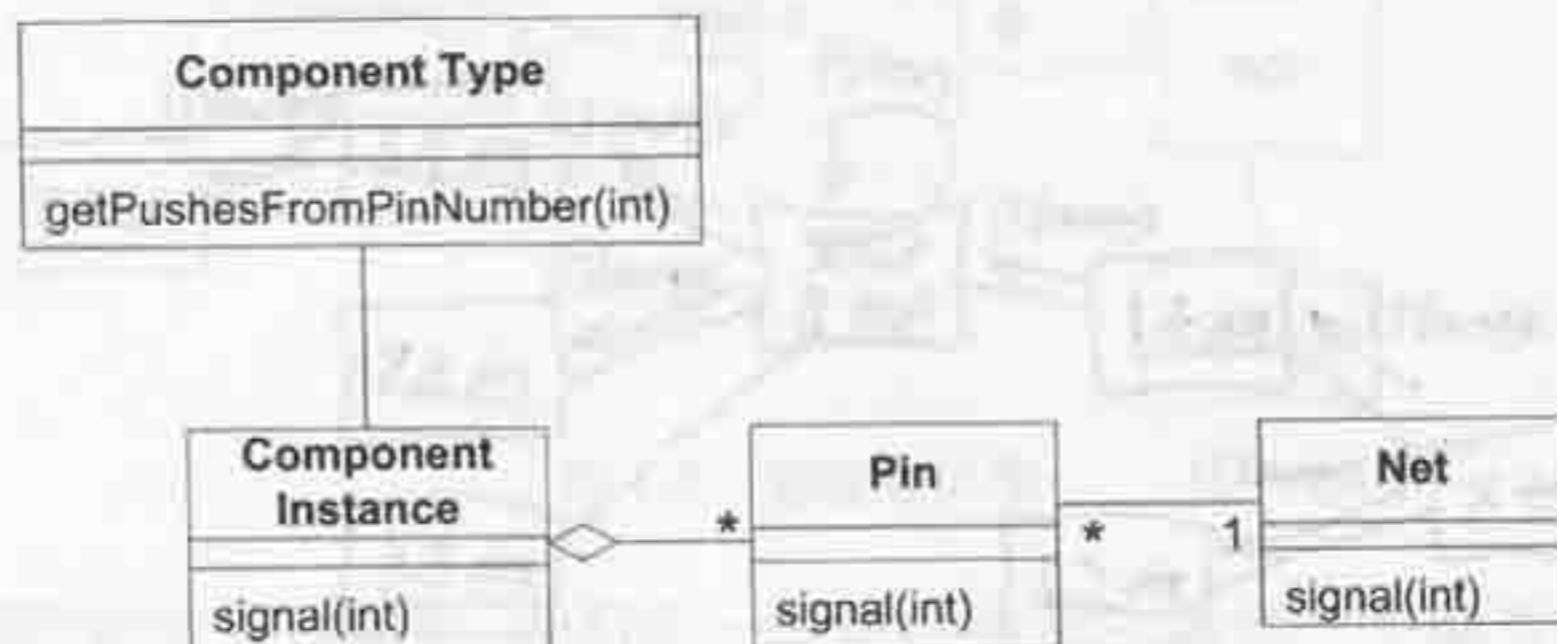


图1-7 早期模型



经过若干次这样的过程，我觉得对工作已经有了充分的了解，可以开始进行一些编程工作了。我编写了一个非常简单的原型，它是由一个自动测试框架所驱动，我避过了所有的基础结构。没有持久化内容，也没有用户界面，这使得我能够专注于行为本身。再过几天我就可以演示一个简单的探针模拟。尽管它使用的是虚拟的数据，并且向控制台写的是原始的文本，它还使用 Java 对象对路径长度进行了实际的计算。那些 Java 对象反映了领域专家和我自己共同使用的模型。

原型的具体化使得领域专家更加清楚模型的含义以及它与软件的功能是如何关联的。从那时起，我们的模型讨论更加具有交互性，他们能够看到我如何将新近获得的知识组合到模型中并写入软件。他们也从原型中获得更加具体的反馈信息，与自己的想法对照。

模型中的内容要比我们在这里展示的更加复杂，它涉及到我们要解决的 PCB 相关问题领域的知识。它统一了许多描述中的同义词和微小的偏差，排除了大量的工程人员理解但是并没有直接相关性的事实，例如元件的实际数字特征。像我这样的软件专业人员看到这个图后便可以在几分钟内了解到该软件是关于哪方面的。人们能够通过一个框架来组织新的信息并更快地进行学习，用来推测什么是重要的而什么不是，并更好地与 PCB 工程人员进行交流。

当工程人员描述他们所需要的新特征时，我让他们带我进入场景来看对象是如何进行交互的。当模型对象不能完成一个重要的场景时，我们通过“头脑风暴”过程对新的对象或对旧的对象进行改变，对它们知识进行消化。我们对模型进行精化，代码也随之改进。几个月过后，PCB 工程人员得到的工具比他们预期的更加丰富。

1.1 有效建模的因素

上文描述的成功案例具备以下一些因素：

1. 模型与实现相互绑定。未经加工的原型建立了早期必需的联系，在随后的迭代中始终对它进行维护和完善。
2. 基于模型生成了一种语言。一开始，工程人员需要向我解释基本的 PCB 问题，我需要解释类图的含义是什么。但是随着项目的进行，我们中的每个人都可以自如地使用来自模型的术语，将它们组织成与模型结构一致的句子，不需翻译，也不会引起歧义。
3. 开发了一个包含丰富知识的模型。对象都具有行为和一些强制性的规则。模型并不仅仅是一个数据方案，它是解决一个复杂问题必不可缺的。它捕获了各种类型的知识。
4. 提炼模型。在模型变得更加完善的过程中，一些重要的概念被加入其中。同样重

要的是，如果概念被证明没有用处或并不重要，则应该丢弃。如果一个不必要的概念与一个需要的概念结合在一起，新的模型将会分离出有用的概念而丢弃另一个。

5. 头脑风暴与实验。与草图和头脑风暴方式相结合的语言使得我们的讨论成为模型实验室，在这里，会对数以百计的实验变种进行尝试和鉴定。当团队审查场景时，口头的表述提供了一种测试模型的快速方式，因为耳朵能够快速地发现表述中的明确性、简易性和不协调的部分。

头脑风暴和大量实验的创造力使得我们能创建一个知识丰富的模型并进行提炼，基于模型的语言以及实现的反馈循环中的约束则提供了支持。这种知识消化(knowledge crunching)过程将团队的知识转化为有价值的模型。

1.2 知识消化

金融分析人员经常对数字进行消化。他们筛选大量的细节数据，对它们进行绑定与重新组合，寻找潜在的意义和能够反映出真正重要因素的简单表示方法——这种理解常常成为金融决策的基础。

高效的领域建模人员就是知识的消化器，他们对大量信息中的相关部分进行探查。他们尝试了一个又一个组织方式，寻找一种对冗杂信息有意义的简单视图。许多模型在实验后被放弃或改变，当发现一组能够明确所有细节意义的抽象概念时，这项工作就获得了成功。这个精练结果就是对已发现的最相关特定知识的精确表达。

知识消化并不是一个孤立的活动，它是由开发团队与领域专家共同合作，由开发人员领导的。他们一起接收信息并对知识进行消化，使之成为有用的形式。原始的资料一般来自于领域专家的意见、现有系统的用户、相关老式系统的技术团队的经验或同一领域中的另一项目。一般是以项目或业务中的文档的形式，以及大量的会谈而得到这些资料的。早期的版本或原型为团队提供了反馈经验，并修改相关阐释。

在老式的瀑布方法中，业务专家与分析人员会谈，分析人员提取摘要，进行抽象后将结果转达给程序员，由程序员对软件进行编码。这种方法并不成功，因为它完全没有反馈机制。分析人员全权负责模型的创建，但仅仅根据从业务专家得到的信息来进行工作。他们没有机会从程序员那里学习新知识或从软件的早期版本获得经验。知识只是单向流动，并没有交互的累积。

其他的项目使用一种迭代过程，但是它们也未能有效创建新知识，因为他们没有对知识进行抽象。开发人员让领域专家描述一项需要的功能然后构造它。他们将结果展示给专家并询问下一步的工作。如果程序员对该领域较为熟悉，他们便可以使得软件保持



能够继续扩展的良好状态，但是如果程序员对于该领域并不感兴趣，他们只能知道应用程序应该做什么，却不了解其背后的原理。这样做虽然能够建立一个有用的软件，但是项目永远不会具有能够从前期特征能推导出更加强大的新特征的能力。

优秀的程序员会自然地开始进行抽象来开发一个模型。但是当建模工作仅仅是在技术基础上进行，并没有领域专家的合作时，这些概念是幼稚的。这种单薄的知识产生的软件能够完成基本的工作，却缺乏与领域专家思考方式之间深刻的联系。

在团队成员一起讨论模型的过程中，他们之间的交互也会发生变化。领域模型的不断精化使得开发人员不断地学习他们所需要的重要业务原理，而不是机械地产生新的功能。领域专家经常通过提炼他们认为必需的知识来精化他们的理解，他们也会逐渐理解软件项目所需的概念严格性。

所有这些都将使得团队成员成为更加有能力的知识消化人员。他们把无关因素分离出去，将模型重新构造成更加有用的形式。因为程序员和分析人员都介入其中，模型变得组织有序并且抽象合理，也为实现提供了很大的支持。又因为领域专家介入其中，模型反映出业务的深层知识。其抽象出来都是真正的业务原则。

随着模型的完善，它逐渐成为项目中信息流的组织工具。模型关注的是需求分析，它与编程和设计相互影响。在一个良性的循环中，它能够加深团队成员对于模型的理解，使得他们对模型了解得更加清晰并对其进行进一步的改进。模型永远都不会是完美的，它们会不断地发展。模型对领域来说必须是实用的。它们必须十分精确，使得应用程序易于实现和理解。

1.3 持续学习

在编写软件的过程中，我们对知识的需求永无止境。项目涉及的知识是零散的，它们分散在许多人的头脑中和文档之中，并且还会与其他信息混合在一起，这样使得我们甚至不知道我们真正需要的知识在哪里。看上去有较少技术障碍的领域也会造成假象：我们意识不到不了解的知识到底有多少。这种无知常常导致我们作出错误的假设。

与此同时，所有的项目都会透露出所需的知识，那些学习到一些新知识的人们能够继续前进。重组会打散团队，知识也会再次分散。仅仅交付了代码而没有传递知识的工作方式使得一些关键的子系统无据可依。使用典型的设计方法时，代码和文档都不能用有效的方式表达出人们辛苦得到的知识，因此当这种口头上的惯例由于任何原因被打破时，知识也就遗失了。

高效率的团队依靠学习(Kerievsky 2003)，有意识地增长自己的知识。对于开发人员



来说，这意味着提高技术知识以及一般的领域建模技能(像本书中介绍的这些)。但它也包括努力学习他们所从事的特定领域。

这些进行自学的团队成员形成了一个稳定的核心人群来关注涉及关键部分的开发任务(要了解该方面的更多知识，请阅读第 15 章)。这个核心团队头脑中积累的知识使得他们成为更加高效的知识消化人员。

阅读到这里，请先停下来问自己一个问题。您曾经学习过有关 PCB 设计过程的知识吗？尽管这个例子只是该领域的一个表面处理，在进行领域模型讨论时也应该学习一些相关知识。我进行了大量的学习，当然我并不是要学习如何成为一个 PCB 工程人员，这并不是我的目的。我试着与 PCB 专家进行交谈，了解关于应用程序的主要概念，并明确我们要创建的内容。

实际上，我们的团队最后发现探针模拟对于开发工作来说是一个低优先级的工作，这个特征最后会逐渐被抛弃。模型中具有这种特征的部分用于在元件之间传递信号并计算跳数。应用程序的核心转移到了其他地方，模型将那些方面变成了关键的部分。领域专家也学习到了更多的知识并对应用程序的目的更加明确(第 15 章深入讨论了这些问题)。

尽管如此，早期的工作还是很必要的。关键的模型元素被保留下来，更重要的是，对知识进行消化的工作使得随后的工作更加高效：团队成员、开发人员以及领域专家等都获得了所需的知识；开始通用一种语言；在实现中结束了反馈回路。要开始发现一些新的东西了。

1.4 知识丰富的设计

在模型(例如 PCB 示例)中所捕获的知识类型超越了“寻找名词”。业务活动与规则对于一个领域来说与其涉及的实体同样重要：领域也会包含各种类别的概念。对知识的消化能够产生出反映这种理解的模型。在模型发生改变的同时，开发人员重构实现来表达模型，使得应用程序能使用得到的知识。

超越实体和值之上的变化可能会对知识消化产生剧烈的影响，因为业务规则之间可能会存在不一致。领域专家通常都没有意识到他们的思考过程有多么复杂，在他们的工作过程中，他们导引所有这些规则，协调冲突，并填补常识缺陷。软件则不能进行这些工作。正是通过与软件专家密切合作，进行知识消化，才使得规则明晰化，进一步充实，彼此协调一致或排除在范围外。



示例：提炼隐藏的概念

我们从一个非常简单的领域模型开始学习，这个模型是一个关于一个航次中预订货物的应用程序，如图 1-8 所示。

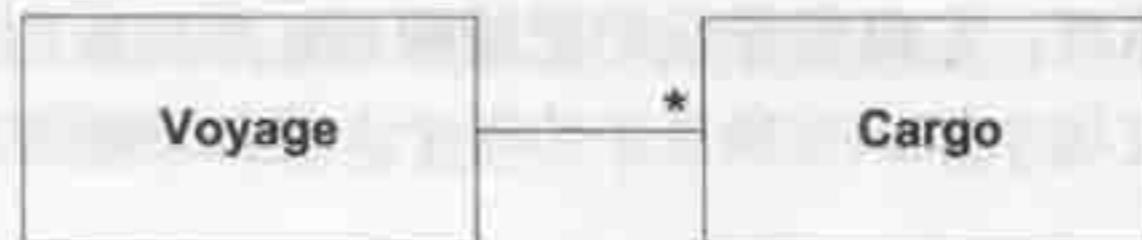


图 1-8 航运模型

我们可以规定预订程序的任务是将每件货物(Cargo)与一个航次(Voyage)联系起来，记录并跟踪这种关系。到现在为止，该程序运行得一直还不错。应用程序代码的一部分可能会使用下面的方式：

```

public int makeBooking(Cargo cargo, Voyage voyage) {
    int confirmation = orderConfirmationSequence.next();
    voyage.addCargo(cargo, confirmation);
    return confirmation;
}
  
```

因为通常都会存在临时取消的订单，航运业的标准操作方式都会接受比一次航程所能装载容量更多的货物预订。这被称作“超额预订”。有时候会使用一个简单的百分比来表示，例如预订容量的 110%。在一些情况下会使用复杂的规则，某些主要客户或某些种类的货物优先。

下面是航运领域中所有业务人员都了解的航运业基本策略，但是这些基本规则可能并不为软件团队中所有的技术人员所知。

需求文档中包含下面的句子：允许 10% 的超额预订。

现在的类图(图 1-9)与代码如下所示：

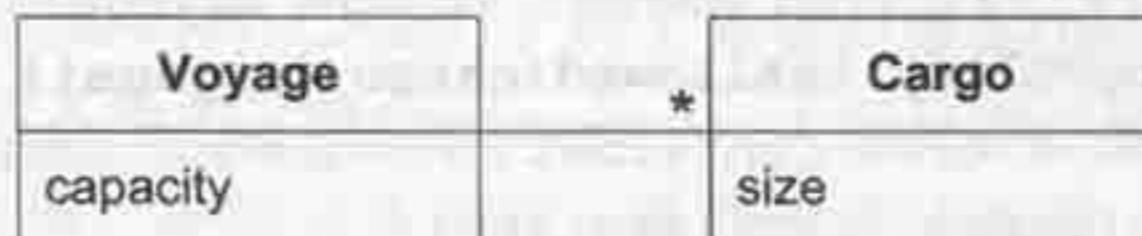


图 1-9 类图

```

public int makeBooking(Cargo cargo, Voyage voyage) {
    double maxBooking = voyage.capacity() * 1.1;
    if ((voyage.bookedCargoSize() + cargo.size()) > maxBooking)
        return -1;
    int confirmation = orderConfirmationSequence.next();
  
```



```
    voyage.addCargo(cargo, confirmation);
    return confirmation;
}
```

现在一个重要的业务规则隐藏在应用程序方法的警戒子句中。后面，在第 4 章中，我们将介绍分层架构的原理，它将指导我们把超额预订规则放到一个领域对象中，但是现在我们集中讨论如何使这个知识对项目中的每个人更加清晰可用。我们将会得到一个相似的解决方案。

- 任何业务专家都不大可能阅读代码来核对规则，即使在开发人员的指导下。
- 一个非业务人员的技术人员，要将需求文档与代码联系起来是困难的。

如果规则更加复杂，那么上述情况更不可能。

我们可以通过更改设计来更好地捕获知识。预订超额规则是一种策略(strategy)。策略是一种设计模式(Gamma et al. 1995)。就像我们所知道的，通常会根据要求替换掉不需要的规则。但是我们要捕获的概念确实与策略的意义相符合，这也是领域驱动设计的重要推动力(参见第 12 章“把设计模式和模型联系起来”)。

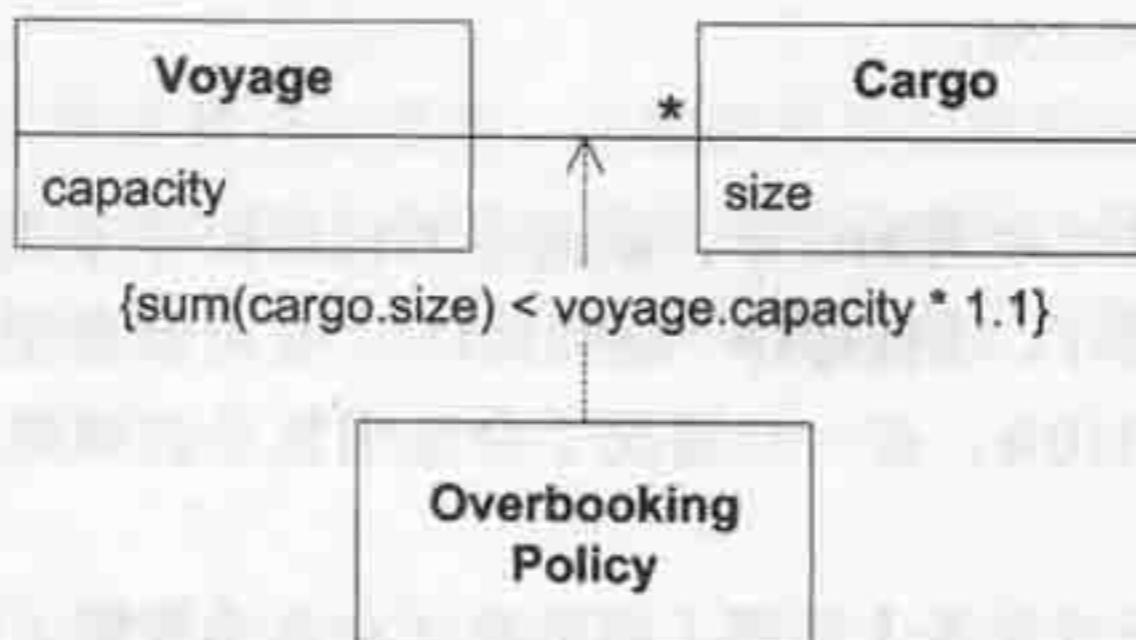


图 1-10 添加策略

代码如下：

```
public int makeBooking(Cargo cargo, Voyage voyage) {
    if (!overbookingPolicy.isAllowed(cargo, voyage)) return -1;
    int confirmation = orderConfirmationSequence.next();
    voyage.addCargo(cargo, confirmation);
    return confirmation;
}
```

新的 Overbooking Policy 类包含如下方法：

```
public boolean isAllowed(Cargo cargo, Voyage voyage) {
    return (cargo.size() + voyage.bookedCargoSize()) <=
```



```
(voyage.capacity() * 1.1);  
}
```

大家都很清楚超额预订是一个独特的策略，该规则的实现是独立的。

现在，我并不建议大家将这种精细的设计应用到领域的各个细节中。第15章“精炼”中会深入讨论如何聚焦于重要问题，将其他问题隔离或最小化。这个例子的意思是要说明一个领域模型和其相应的设计能够被用来保护和分享知识。更加明确的设计有以下几个优点：

- 为了使设计进入这个阶段，程序员与其他有关人员都了解了超额预订的本质，它是一个重要的特殊业务规则，并不仅仅是一个含糊的计算问题。
- 程序员可以向业务专家展示技术工件，甚至代码，但应该是领域专家(在指导下)可以理解的，这时反馈回路结束。

1.5 深层模型

有用的模型很少停留在表层关系上。当我们开始理解领域以及应用程序的需要时，我们通常会放弃开始看起来比较重要的表层模型元素，或者会改变对它们的看法。巧妙的抽象能够使得开始不为我们所知的事情核心的元素显露出来。

前面的例子所讲到的项目将会在本书中作为示例引用：一个集装箱装运系统。本书中的例子对于那些不熟悉装运的用户也是容易理解的。但是在一个实际的系统中，团队成员必须进行持续的学习来充实自己，因为一个实用并清晰的模型通常需要在领域和建模技术方面的完善技巧。

在这个项目中，因为每一批装载的货物都首先要登记重量，我们可以开发一个模型，它能够描述船运货物以及它的航线等。这些都是必要和有用的，然而领域专家还是感觉不满意。我们忽视了他们考虑业务的方式。

终于，通过几个月的知识消化，我们了解了货物的搬运(handling)、货物装卸/loading and uploading)和货物到各地的运输(movement)，大部分都由转包商或公司的操作人员执行。装运专家的意见是，各部分之间有一系列的责任传递。它是一个管理法律和实际的责任传递的过程，这个传递行为从托运人(shipper)到本地运输商(carrier)，在运输商之间传递，最后到达收货人(consignee)。很多情况下，在一些重要步骤进行时，货物则放在仓库里。在其他一些时间里，货物可能会经历一些复杂的实际过程，而这些与装运公司的业务决策并无关系。在处理航程的物流之前，应该落实提单等法律文件以及支付过程。



对于装运业务的更深一步理解并不需要将航线对象删除，但是模型发生了很大的改变。我们对装运的理解从将集装箱在各地之间运送改变成在实体之间传输货物传递责任。处理这些责任传递的特征不再是仅仅与装车作业相联系，而是由一个模型所支持，该模型来自于对这些操作和责任之间重要关系的理解。

知识消化是一个探索过程，这个过程是永无止境的。

这个阶段的探索过程可能需要数周时间，或者更长。如果在开始时就将所有的东西都删掉了，那么一切就是未知的而且没有经验可供参考。因此，最好从最简单的着手——从一个简单的例子入手——然后逐步地增加复杂性。这样，你就可以通过自己的经验来学习，而不是通过别人的讲解。当然，这并不意味着你不能向别人学习。事实上，向别人学习是非常重要的。然而，最好的学习方法是通过自己动手实践。通过实践，你可以更好地理解模型的各个方面，并且能够更深入地掌握它们。通过实践，你还可以发现自己的不足之处，并且通过不断改进自己的模型，从而不断提高自己的能力。

这个阶段的探索过程可能需要数周时间，或者更长。如果在开始时就将所有的东西都删掉了，那么一切就是未知的而且没有经验可供参考。因此，最好从最简单的着手——从一个简单的例子入手——然后逐步地增加复杂性。这样，你就可以通过自己的经验来学习，而不是通过别人的讲解。当然，这并不意味着你不能向别人学习。事实上，向别人学习是非常重要的。然而，最好的学习方法是通过自己动手实践。通过实践，你可以更好地理解模型的各个方面，并且能够更深入地掌握它们。通过实践，你还可以发现自己的不足之处，并且通过不断改进自己的模型，从而不断提高自己的能力。

这个阶段的探索过程可能需要数周时间，或者更长。如果在开始时就将所有的东西都删掉了，那么一切就是未知的而且没有经验可供参考。因此，最好从最简单的着手——从一个简单的例子入手——然后逐步地增加复杂性。这样，你就可以通过自己的经验来学习，而不是通过别人的讲解。当然，这并不意味着你不能向别人学习。事实上，向别人学习是非常重要的。然而，最好的学习方法是通过自己动手实践。通过实践，你可以更好地理解模型的各个方面，并且能够更深入地掌握它们。通过实践，你还可以发现自己的不足之处，并且通过不断改进自己的模型，从而不断提高自己的能力。

这个阶段的探索过程可能需要数周时间，或者更长。如果在开始时就将所有的东西都删掉了，那么一切就是未知的而且没有经验可供参考。因此，最好从最简单的着手——从一个简单的例子入手——然后逐步地增加复杂性。这样，你就可以通过自己的经验来学习，而不是通过别人的讲解。当然，这并不意味着你不能向别人学习。事实上，向别人学习是非常重要的。然而，最好的学习方法是通过自己动手实践。通过实践，你可以更好地理解模型的各个方面，并且能够更深入地掌握它们。通过实践，你还可以发现自己的不足之处，并且通过不断改进自己的模型，从而不断提高自己的能力。

第2章

交流及语言的使用

在一个软件项目中，领域模型可以成为项目成员所使用的通用语言的核心。模型是建立在项目成员头脑中的一组概念，它使用术语及关系来反映领域的内涵。这些术语和相互的关系规定了适合于领域的语言语义，这种语义对于技术开发来讲是足够精确的。这也是将模型贯穿到开发活动中，并将模型与代码进行绑定的关键。

基于模型的交流并不局限于使用统一建模语言(UML)中的图。为了更有效地使用模型，我们需要充分地使用每一种交流手段。基于模型的交流提高了编写文档的实用性，非正式的图和偶然的交谈在敏捷过程中重新被强调。它改善了代码本身及其测试之间的交流。

在一个项目中，语言的使用虽然是很小的一方面，但它却十分重要……

2.1 通用语言

首先写下一个句子，
接着将它截成小段；
然后将这些小段全部混合，并将它们分类选出
就像他们是偶然出现的那样：
短语的顺序没有任何影响。

——Lewis Carroll, “Poeta Fit, Non Nascitur”

要想创建一个充满柔性的、知识丰富的设计，团队需要一种通用的共享语言，还需



要使用该语言完成很多真实的实验，尽管这在软件项目中并不经常发生。

领域专家对于软件开发的技术行话的理解通常都非常有限，但是他们会使用他们自己领域的行话——很可能有很多种“风味”。而另一方面，开发人员能够理解并使用描述性的功能术语来讨论系统，而并不使用专家们的语言。或者开发人员可能建立一个能够支持他们设计的抽象结构，而领域专家并不理解。从事于同一问题但不同部分的开发人员会产生他们自己的设计概念和描述领域的方式。

由于这种语言上的差异，领域专家对他们所需要的只能含糊地进行描述。开发人员费力地去理解领域中对他们来说陌生的东西，也只能含糊地理解领域专家的思想。团队中很少有人设法去精通两种语言，于是他们之间的信息流遇到了瓶颈，不能够准确地进行转换。

在一个没有通用语言的项目中，开发人员不得不对来自领域专家的信息进行转换。领域专家则在开发人员与其他领域专家之间进行信息转换。开发人员彼此之间的信息甚至也需要转换。这些转换混淆了模型的概念，可能导致重构代码时的失败。这种间接的交流掩盖了结构上的分裂——不同的团队成员使用不同的术语而却不能意识到这一点。这会导致产生不可靠的软件，它们不能够彼此适应。转换工作使得知识与思想不能够相互影响，从而无法深入理解模型。

当一个项目的语言存在断层时，会面临一系列的问题。领域专家使用自己的行话，而技术团队成员却按照设计的思路调整和使用自己的语言去讨论领域。

天天进行讨论时所使用的术语与嵌入到代码(一个软件项目最重要的最终产品)中的术语是分裂开来的。甚至可能是同一个人，在编写文档或者代码时，也会使用跟交流讨论时完全不同的语言，这会导致对于领域的某些深入描述只是短时间内存在，却无法反映到代码或文档编写中。

语言转换减弱了交流的效果，使得知识积累也不尽如人意。

然而任何一种方言都不能够成为通用语言，因为它们都无法满足所有需求。

语言转换的操作成本加上可能产生误解的风险使得代价过大。项目需要一种比所有语言更加健壮的通用语言。通过团队的不懈努力，将领域模型作为通用语言的核心，会在团队交流和软件实现之间搭建桥梁，并发挥核心作用。这种语言在团队的工作中得到广泛应用。

通用语言(Ubiouitous Language)这个词汇包含了类的名称与重要的操作。“语言”这个词，包括了对模型中已经明确的规则进行讨论所使用的术语，还包括说明模型所展示的高级组织原理的术语(如上下文图(Context Map)和大比例结构，将在第14章和第16章



中进行讨论)。最后,团队应用于领域模型的模式名称将进一步地充实这种语言。

模型的相互关系成为所有语言的组合规则,单词和短语的含义反映了模型的语义。

基于模型的语言不仅被开发人员用来描述系统中的工件,还用来描述任务和功能。这个模型提供开发人员和领域专家彼此交流以及领域专家之间讨论需求、开发计划和特性时使用的语言。这种语言的使用越普遍,理解起来就越容易。

至少,这是我们需要努力的方向。但是在初期,模型可能会很简单,不能完全满足这些任务,它可能缺乏像领域行话那样丰富的语义。但又不能够直接使用这些行话,因为它们可能包含模糊与矛盾。它们可能缺少开发人员编写代码时所需的柔性和灵活性特征,也许因为它们并不把那些看作为模型的一部分,或者因为编码只是为了隐式地体现领域中的概念,编码风格只是程序性的问题。

尽管顺序看起来是循环的,但是能够产生有用模型的知识消化过程只依赖于团队所使用的基于模型的语言。坚持使用通用语言可以迫使模型的弱点暴露出来。团队便可以进行实验来寻找不灵活的术语或组合的替代品。由于语言中存在隔阂,新的单词将被引入讨论之中。这些关于语言的改变将在领域模型中也引起相应的改变,并且让团队对类图进行更新,将代码中的类和方法重命名,甚至当术语的意义改变时对行为进行改变。

通过在实现的上下文中通用这种语言,开发人员能够指出其中的不精确性与存在的矛盾,促使领域专家进一步发现可行的替代品。

当然,领域专家可能在通用语言的范围之外发表意见来进行解释并给出更广泛的上下文。但是在模型涉及的领域,他们应该只使用这种语言并在发现不灵活或不完整甚至错误的时候加强注意。借助于广泛地使用这种基于模型的语言,直到最终满意的时候,我们可以得到一个完整的,并且能理解的模型,这个模型由一些组合起来用于表达复杂思想的简单元素组成。

因此:

要将模型作为语言的骨干。团队在所有的交流与代码中都应该练习使用这种语言。在图、文档编写,尤其是发表意见过程中都使用相同的语言。

通过用替代表达方法进行实验来消除实际中遇到的困难,它们反映了替代模型的含义。然后按照新的模型重新进行编码,并对类、方法和模块进行重命名。在交谈中辨明术语中含糊的词义,就像我们对普通的词语意义达成一致意见那样。

要意识到通用语言中的变化也是模型中的变化。

领域专家应该排除掉传达领域意思时不易使用或不适当的术语或结构;开发人员应该注意防止可能导致设计失败的模糊性或不一致性。

使用了通用语言,模型就不仅仅是一个设计工件了。它整合到开发人员与领域专家



所做的每一件事中。语言用动态的形式传送知识。使用语言进行讨论使得图与代码有了真实的含义。

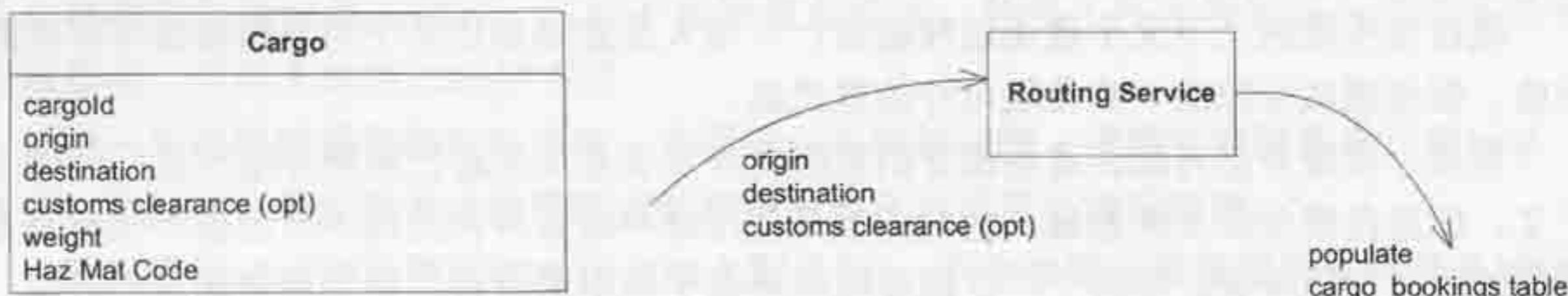
在用通用语言进行讨论时假设只有一个模型。第 14 章“维护模型完整性”讲述处理不同模型(不同语言)的共存以及如何使模型不会分裂。

通用语言是不在代码中出现的设计方面的主要载体——代码是组织整个系统的大比例结构(参见第 16 章)，限界上下文(Bound Context)用来定义不同系统与模型的关系和其他被应用于模型和设计中的模式。

示例：设计货运路线

下面的两则对话有着细微但也很重要的差别。在每个场景中，注意观察讲话者有多少是从业务层面谈论软件要做什么，又有多少是从技术层面谈论它的工作机理的。用户与开发人员使用的是同一种语言吗？如果要对应用程序必需完成的操作进行讨论，这样的语言足够丰富吗？

场景 1：最小化的领域抽象



Database table: cargo_bookings

Cargo_ID	Transport	Load	Unload

图 2-1 场景 1

用户：那么，当我们改变了清关点(customs clearance point)时，就需要重做整个路线计划。

开发人员：是的。我们要删除装运表(shipment table)中所有包含该货物 id 的行，然后我们将出发地、目的地和新的清关点传给路线服务(Routing Service)，它会填充数据库



表(table)。Cargo 中应该有一个布尔值，这样我们可以知道在装运表中是否有数据。

用户：删除行？好的，不管它是什么意思。总之，如果我们之前根本没有指定清关点，我们也同样需要这样做。

开发人员：当然，无论什么时间，您改变了出发地、目的地或清关点(也包括第一次输入这些信息)，我们都要检查是否已产生了装运数据，如果已经有了就将其删除，再让 Routing Service 重新生成新的数据。

用户：可以，但是如果以前的海关清关操作恰好是正确的，我们不希望这样做。

开发人员：哦，这很容易。每次仅仅让 Routing Service 重新写入或不写入会更加容易。

用户：是的，但是对我们来说，每当有一个新的航线，就需要为其制定所有的辅助计划，这是一件很费力的工作，因此我们希望在不必要时不要重新安排航线。

开发人员：啊，明白了。那么，如果您第一次输入一个清关点，我们需要查询数据表来找到以往的清关点，然后将它和新的比较。这样我们就知道是否需要重新安排了。

用户：不用去比较出发地和目的地，因为它们发生变化，航线总会改变。

开发人员：好的，我们会注意的。

场景 2：在讨论过程中使用领域模型

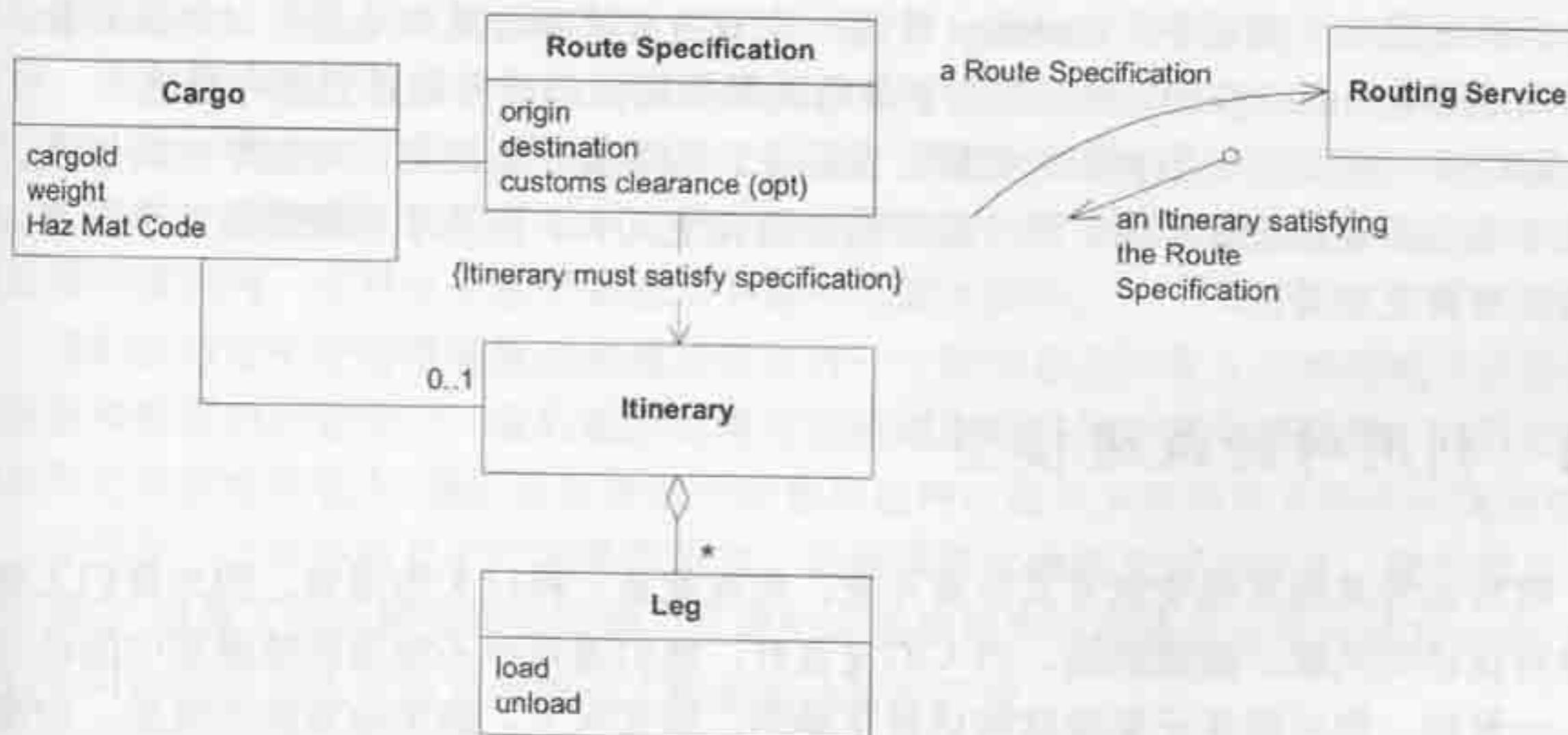


图 2-2 场景 2

用户：那么，当我们改变了清关点时，就需要重做整个路线计划。

开发人员：是的。当您改变路线说明(Route Specification)中的任何一个属性时，我



们将删除旧的航线(Itinerary)并让 Routing Service 根据新的 Route Specification 生成一个新的 Itinerary。

用户：如果之前没有指定一个清关点，我们也要同时做这项工作。

开发人员：当然，任何时候您改变 Route Specification 的任何内容时，我们都将重新生成 Itinerary。这包括第一次输入信息。

用户：可以，如果以前的清关操作恰好是正确的，我们不希望进行这项工作。

开发人员：哦，没有问题。如果仅仅让 Routing Service 每次都制定一个新的 Itinerary 会更容易。

用户：是的，但是对于我们来说，给新的 Itinerary 制定辅助计划是一件很费力的工作，因此我们希望在不必要时不重新安排航线。

开发人员：哦。那么我们需要对 Route Specification 添加一些功能。这样，无论何时您改变 Specification，我们会判断 Itinerary 是否满足它。如果不满足，我们就让 Routing Service 重新产生 Itinerary。

用户：不用考虑出发地或目的地的变化，因为 Itinerary 总会随它们的变化而变化。

开发人员：好的，但是如果每次只作比较的话，情况更加简单。只有在 Route Specification 不再被满足的时候才会生成 Itinerary。

第二段对话传达了领域专家的更多意图。用户在两个对话中都使用了词语 itinerary，但是只有在第二个对话中，itinerary 才是一个双方能够精准具体地进行讨论的对象。它们明确地讨论 route specification，而不是每次都用属性和过程来对它进行描述。

这两段对话特意采用相似的结构。实际上，因为要不断地对应用程序的特性及讨论过程中出现的误解进行解释，第一段对话会显得更冗长。而基于领域模型术语的第二段对话则显得更加简洁。

2.2 利用对话改进模型

将对话与其他其他交流方式分离开来，简直就是一种巨大的浪费，因为我们人类本来就有说话的天赋。遗憾的是，当人们发言时，他们通常并不使用领域模型中的语言。

一开始，您可能并不觉得这句话是正确的，而实际上，确实也有例外发生。如果今后您去参加一个需求或设计的讨论会时，多注意倾听。您会听到使用内行话(或外行话)对特性进行的描述；会听到关于技术工件和具体功能的讨论；当然，您也会听到来自领域模型的术语；很显然，那些源自内行话的公共语言中的名词可以在编码时当作对象，因此这些术语会被提及。但是您能从中听到任何使用当前领域模型中的关系和交互等术



语来描述的措辞吗？

改进一个模型的最好方式之一就是通过对话进行模型探索，尝试罗列各种可能的模型变化，大声描述不同的结构，这样不当之处很容易被听出来。

“如果我们给 Routing Service 提供出发地、目的地和到达时间，它便可以查询该货物必需的停靠点，然后……将它们保留在数据库中。”（含糊并且过于技术化）

“把出发地、目的地等都提供给 Routing Service，这样我们就能够得到一个 Itinerary，它里面有我们需要的所有信息。”（更加完整，但是有点罗嗦）

“Routing Service 能够找到满足 Route Specification 的 Itinerary。”（简洁明确）

我们必须与单词和短语打交道，利用我们的语言能力建模，这就和我们必须利用画图来进行视觉和空间推理一样；就像我们利用我们的分析能力进行系统的分析和设计，然后进行编码。这些思考的方式互为补充，结合使用它们才能找到有用的模型和设计。在所有这些方式中，使用语言进行实验常常是最容易被忽视的（本书的第III部分将研究这种发现过程并讲述它在几个对话中的影响）。

实际上，我们的大脑看上去很擅长于处理口语的复杂性（对于像我一样的非专业人士来说，可以查阅 Steven Pinker 所写的 *The Language Instinct* [Pinker 1994]）。例如，当拥有不同语言背景的人们因为商业原因走到一起时，如果没有通用语言，他们就会发明一种混杂语(pidgin)。混杂语并不像发言者所使用的初始语言那样面面俱到，但是它对手头的任务非常适合。当人们交谈的时候，他们就可以很自然地发现他们语言的解释和含义的差别，进而消除这种差别。他们发现这种语言中的粗糙之处，并使它变得顺畅。

有一次，我去上一堂强化性的西班牙语课。课堂上的规则是一个英语单词都不能够讲。开始时并不顺利，感觉非常不自然并且需要很大的自制力。但是逐渐地我和同学们都说得非常流利，这种水平在书面练习中是不可能达到的。

我们在讨论中使用领域模型的通用语言时——特别是在开发人员和领域专家仔细推敲场景和需求的讨论中——我们逐渐对语言越来越熟悉，使用越来越流利，并将它的细微差别之处教给其他人。我们很自然地开始使用这种以前从未在图和文档中出现的语言。

要想在一个软件项目中产生通用语言，说起来要比做起来容易得多，我们要设法努力实现它。就像人类的视觉和空间能力让我们在观察图形时能够快速地传输和处理信息，我们也可以利用语法和语义方面的才能来驱动模型开发。

因此，下面的一段话可作为通用语言模式的补充：

结合模型来讨论系统。使用模型的元素和元素之间的交互来大声描述场景，按照模型允许的方式把概念组合在一起。找到更简单的方式来表达要表达的内容，然后将这些意见应用到图形和代码中。



2.3 一个团队，一种语言

技术人员常常觉得没必要让业务专家与领域模型相接触。他们说：

“这对他们来说太抽象了。”

“他们不懂对象。”

“这样我们就得用他们的术语来收集需求。”

这是我从使用两种语言的团队中听到的一些原因，让我们忘掉这些吧。

当然，系统设计过程中存在一些领域专家不关心的技术部分，但是模型的核心却会让他们更加感兴趣。太抽象了？您如何知道您进行的抽象是合理的呢？您对领域的理解有他们深入么？有时候需要从底层用户那里收集一些特殊的需求，他们在表述这些特殊需求的时候可能会使用一些具体的术语，但是领域专家则能够对该领域进行深度思考。如果高级领域专家不能够理解模型，那么该模型是有问题的。

刚开始时，用户们讨论要建模的系统所必需具有的能力时，并没有成形的模型供他们使用。但是一旦他们与开发人员们一起采用这种新的想法进行工作时，朝共享模型进军的探索过程也就开始了。初期模型可能并不容易使用并且也不完整，但是它会逐渐地被改进。随着新语言的进化，领域专家必须花费时间来掌握它，并且及时更新那些还有价值的旧文档。

当领域专家在与开发人员或其他领域专家的讨论中使用这种语言时，很快就能发现模型中不适合他们的需求或完全是错误的地方。由于基于模型的语言要求精确度，因此领域专家(在开发人员的帮助下)也会发现自己想法中存在矛盾或模糊之处。

开发人员与领域专家可以通过各种场景一步一步地使用模型对象来非正式地走查模型。每一次讨论都是开发人员与用户专家共同研究模型，深化彼此的理解并细化概念的机会。

领域专家可以在书写用例时使用模型的语言，也可以在说明验收测试时更直接地使用模型。

有时候人们会反对使用模型语言来收集需求。难道需求不应该独立于实现它们的设计吗？这种看法忽视了一个现实问题，那就是所有的语言都是基于某个模型的。词语的意义是难于捉摸的。领域模型基本上是从领域专家的行话中得来，但是已经进行了整理，具有精确严密的定义。当然，如果这些定义与领域中公认的含义偏差较大，领域专家应该提出反对意见。在一个敏捷过程中，需求逐渐演化为项目进行下去，因为在一个应用程序充分确定之前几乎没有可用的知识存在。这种演化过程的一部分体现为通过精化通用语言对需求的重新组织。

语言的多样性是必要的，但是领域专家与开发人员之间不应该有语言上的障碍(第



14章“维护模型完整性”讲述在同一个项目中多个模型共存的情况)。

当然,开发人员确实会使用一些领域专家不能够理解的技术术语,因为技术人员需要大量行话来讨论一个系统的技术方面。毫无疑问,用户也会使用一些超出应用程序的有限范围或者开发人员不能理解的特殊行话。但是这些都是语言的扩展。对于反映不同模型的相同领域,这些方言不应该包含可替换的词汇。

图2-3显示了通用语言的构成。

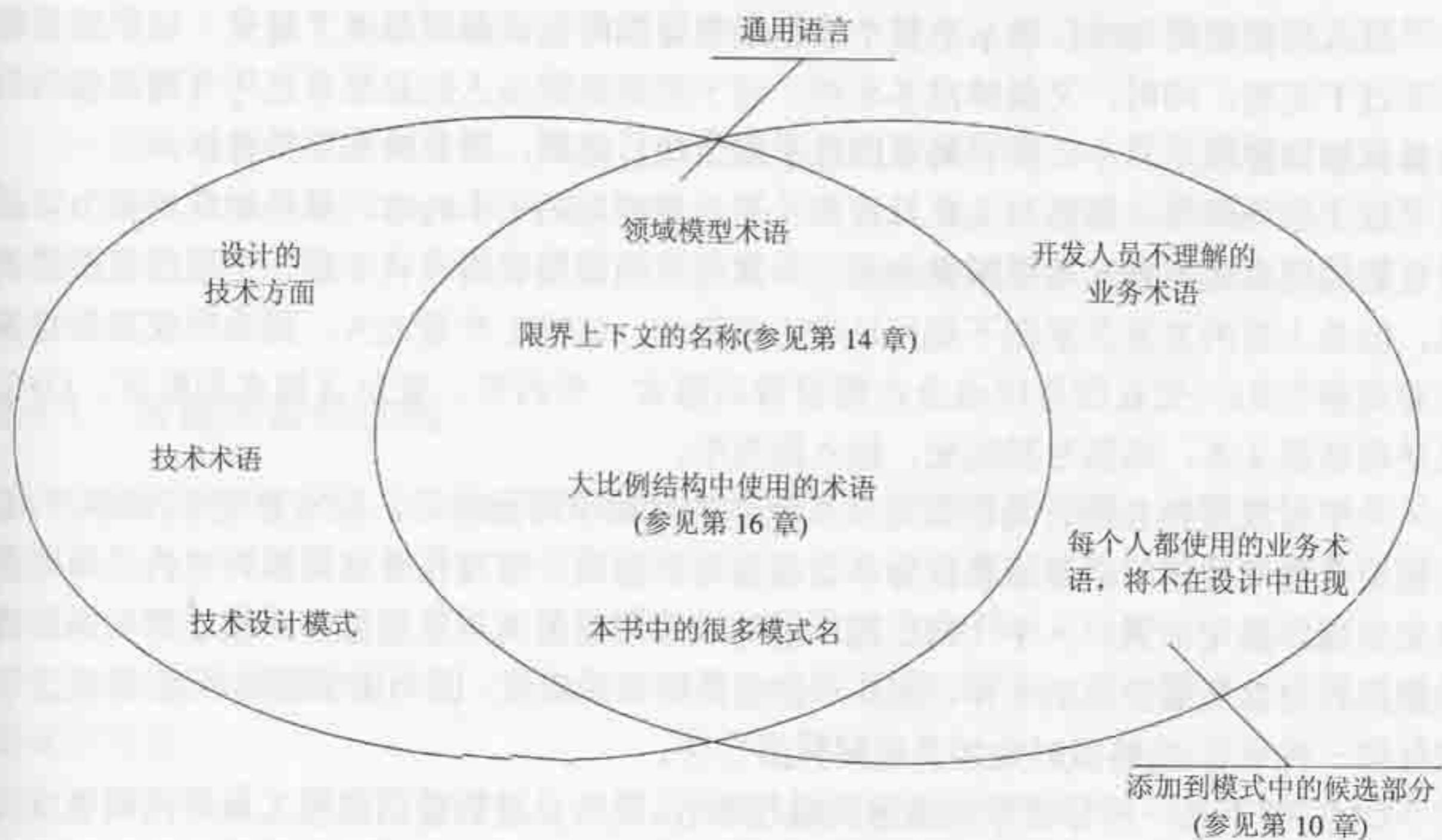


图2-3 行话的交集产生了通用语言

有了通用语言,开发人员之间的会谈、领域专家之间的讨论以及代码本身的表达都基于同一种语言,这来自于一个共享的领域模型。

2.4 文档和图

无论何时,在会议中讨论软件设计时,如果不在书写板或草图上画图,我几乎就无法工作。我所画的很大一部分是UML图,主要是类图与对象交互图。

有些人习惯于直观的表示方法,图形能够帮助他们抓住一些信息。UML图可以很好地表达对象之间的通信关系,并且能够明确地显示对象之间的交互作用。但是它们并不能够表达这些对象在概念上的定义。在会议上,当我画下草图的时候会通过发言进一步



充实这些含义，有时，这些定义会在跟其他参与者谈话时提出来。

简单、非正式的 UML 图能够确定讨论的主题。针对手头问题，在一个图中画出 3 到 5 个重要的对象，可以使每个人都能够集中在要点上。每个人都使用这些对象之间的关系视图，更重要的是，共享这些对象的名称。在 UML 图的帮助下，口头讨论会变得更加高效。当人们尝试不同的想法时，图形会发生改变，草图就会增添交谈中的一些意见。毕竟，UML 代表“统一建模语言”。

当人们需要用 UML 表示出整个模型或设计的时候，麻烦就多了起来。很多对象模型图过于完整，同时，又漏掉很多东西。过于完整是因为人们总想要把所有将要编码的对象都放到建模工具中。所有的东西过于细节化，结果，只见树木不见森林。

除了这些细节，属性与关系只占到一个对象模型的一半内容。那些对象的行为以及对它们的约束则不容易用图例来表示。对象交互图能够说明设计中的一些错综复杂的关系，但是大量的交互关系则不能用这种方法表示，这样工作量太大，因为不仅要创建图还要理解它们。交互图有时也会在模型背后隐含一些内容。要包含约束和断言，UML 还是得依靠文本，用括号括起来，插入图形中。

一个对象所担负的行为职责可以从操作的名称中得到暗示，在对象交互(或顺序)图中也会含蓄地展示，但是这些行为不会被说明。因此，这项任务就得靠补充的文本或会谈来完成。换句话说，一个 UML 图不能够表示模型的两个重要方面：模型所代表的概念意义和对象所要完成的工作。这并不会给我们带来麻烦，因为谨慎使用英语(西班牙语或任何一种语言)能够很好地担负起解释的任务。

UML 并不是一种非常令人满意的编程语言。我所见过的使用建模工具的代码生成功能进行的尝试从来没有一次能够达到预期目的。如果您仅仅局限于 UML 的功能当中，很多情况下会遗漏掉模型的最关键部分，因为一些规则并不适合以线框为主的图。另外，代码生成器不能够利用上面所说的文本注释。如果您确实使用类似 UML 的图形语言去编写一个可执行程序，那么这时的 UML 图便仅仅只是程序本身的另一种视图，而其“模型”的真实含义被您给遗弃了。如果使用 UML 作为实现语言，则仍然需要其他的手段来清晰地表达模型的含义。

图形是交流和进行解释的一种手段，它们也很容易引发头脑风暴。小而精的图能够很好地达到这些目标；那些想表述完整对象的大而全的图常常不便于交流和解释；其中的细节会让读者迷失方向，找不到真正的含义。它会使我们远离被对象所包围的模型图，或甚至 UML 数据资料库的包围圈。它引导我们朝使用简化的图来表述对象模型的重要概念部分的方向努力。而此对象模型对于理解设计是必不可少的。本书中的图都是我在项目中使用的一些很典型的例子。它们非常简单，并具有解释的功能，当要阐明某些观



点时还会使用一些非标准的注释。它们描述了设计约束，但是它们不是面面俱到的设计规约。它们体现的是思想的骨架。

设计的重要细节能够在代码中体现出来。一个良好的实现应该是透明的，能够展示它所依据的模型(这也是下一章以及本书剩余部分的主题)。互为补充的图和文档能够将人们的注意力集中到一些关键的问题上。使用自然语言进行讨论可以消除含义上的细微差别。因此与典型的 UML 图处理事物的方式不同，我更喜欢使用一种不同的处理手段。与其使用一个带有文本注释的图，我更愿意编写一个文本文档，里面用经过挑选的简化图作为示例。

一定要记住模型并不是图。图的目的是帮助表达和解释模型。代码能够反映设计的细节，编写风格良好的 Java 代码与 UML 一样具有表达力。经过精心挑选和组织的图能够帮助人们集中注意力并有指导作用，当然它们不能因被强制去完整地表达模型或设计的意义而变得含糊。

2.4.1 书面的设计文档

口头的交流在语义上弥补了代码的太过精密和太过细节的问题。尽管谈话很重要，能够将个人与模型联系起来，但任何规模的团队还是很需要能够提供稳定性和共享能力的书面文档。然而编写能够真正帮助团队生产优秀软件的书面文档是一个挑战。

一旦一个文档固定下来，它将失去与项目流程之间的联系，并将滞后于代码或项目语言的发展。

有许多编写文档的可行方法，本书后面会讨论几种具体的文档，它们针对一些特殊的需求，但是我不会罗列出一个项目应该使用的所有文档。有两个常用的评价文档的指导方针。

1. 文档应该对代码和发言作补充

每个敏捷过程对于文档编写都有自己的原则。极限编程提倡不使用任何额外的设计文档，让代码自己来表达其含义。可运行的代码不会说谎，而文档则可能并不真实。可运行代码的行为是没有模糊性的。

极限编程仅仅关注一个程序中的活跃元素和可执行测试。甚至代码中添加的注释也不能影响程序行为，因此它们常常无法与活跃代码和其驱动模型同步。外部的文档和图形不能够影响程序的行为，因此它们也常常被剔除。在另一方面，口头的交流和书写板上临时使用的图形可以降低混淆产生的可能性。这种将代码作为通信介质的做法促使了开发人员努力保持代码的整洁和明确性。



但是将代码作为设计文档确实有它的局限性。它会使读者负担过多的细节问题。尽管代码的行为是明确的，但并不意味着是明显的。一个行为背后的含义会很难表达出来。换句话说，仅仅通过代码来编写文档与使用综合的 UML 图一样，都含有一些相同的基本问题。当然，团队内部大量的口头交流能够给代码阅读提供一些上下文环境的分析和指导，但也是暂时和局限的，并不只是开发人员需要理解模型。

文档不应该去做代码已经做得很好的工作。代码已经提供了所有的细节问题，它是一种精确的程序行为说明。

其他文档需要阐明代码的含义，并洞察其大比例结构，将注意力集中到一些核心元素上。当程序语言不支持某概念的直接实现时，文档可以阐明设计意图。编写的文档应该作为代码和谈话的补充。

2. 文档应该服务于一个活跃稳定的工作流

在我为一个模型编写文档时，我对细心选择的一些子模型作了图示，并在边上作了注释。我定义了一些类及其功能，并将它们放入一个用自然语言能够提供的上下文含义的框架中。这些图形可能是临时的——甚至是手工绘制的。从节省劳力来看，手绘的图形对于偶然和临时的问题很有优势。它们对交流非常有好处，因为它们通常都正确地反映了模型思想。

一个设计文档的最大价值在于解释模型的概念，帮助我们定位到代码中的细节，也许还能够洞察到模型打算使用的风格。根据团队的原则，整个设计文档应该与墙上的草图一样简单，或者可以更实际一点。

文档必须涉及到项目的活动。验证这一点最简单的方法就是观察文档与通用语言的相互作用。文档所用的语言是不是人们在项目中所使用的呢(现在)？文档是否用嵌入到代码中的语言书写的呢？

观察通用语言并关注它的变化。如果设计文档中所解释的术语没有在会谈与代码中体现出来，则该文档还没有完成它的任务。也许文档太过庞大或复杂，也可能是它没有集中于重要的主题，使得人们没有阅读这些内容或没有发现这些问题。如果它没有对通用语言产生任何影响，那么可能是出了一些问题。

相反，您可能见到过通用语言发生了改变，而文档滞后的情况。这是因为文档可能看起来与人们不太相关或并不需要进行更新。它应该被安全地归档；如果继续保持为活跃文件，则可能产生混淆而危害项目。如果一个文档的角色并不重要，一直保持它的更新将是劳动力的一个巨大浪费。



通用语言还使得其他的文档(如需求说明)更加简洁并减少了模糊性。领域模型在反映最相关的一些业务知识的同时,应用程序的需求成为模型中的场景,通用语言可以使用直接与模型驱动设计相关的术语来描述这样的场景(参见第3章)。结果就是,需求说明可以用更加简洁的方式书写,因为它们不需要传达模型背后包含的业务知识。

通过保持文档的最小化并聚焦于代码和会谈的补充,可以保持文档与项目的联系。以通用语言及它的发展作为选择文档的指导,使得文档与项目活动相交织并一直发挥作用。

2.4.2 执行的基础

现在我们来分析一下为什么选择XP,它基本上全部依赖于可执行代码和它的测试。本书的大部分内容是通过模型驱动设计来让代码表达含义的(参见第3章)。编写良好的代码有利于提供信息,但是它所传达的信息并不能确保是正确的。一段代码所表示的真实行为是无法改变的,但是一个方法的名称可能是模糊的、有误导性的,或与方法的内部相比较已经过期。测试所得到的断言是精确的,而代码的不同名称和组织结构则不是这样。良好的编程风格使得它们之间的联系尽可能直接化,但是在自律方面仍然需要锻炼。编码时需要一丝不苟的态度,这样才能使得代码不仅能够完成正确的任务并且能够正确表达含义。

歧义的消除是声明性设计这种方法的一个最主要的功能(在第10章中有相关讨论),在这种方法中,程序元素目的的陈述决定了它在程序中的实际行为,它对从UML中生成程序的做法也有一定的驱动,尽管通常这并不会产生很好的效果。

虽然代码可能有误导作用,但它仍然比其他文档更为基础。使用当前标准技术对编码的行为、意图和消息进行矫正,需要规程以及某种关于设计的思考方法(在第III部分中有详细讨论)。要有效地进行交流,代码必须基于编写需求的同一种语言——也就是开发人员之间以及他们同领域专家之间交谈所使用的同一种语言。

2.5 说明性模型

本书的核心内容在于模型应该是实现、设计和团队交流的基础。将模型与它们分隔开是一种冒险的行为。

模型在讲授有关领域知识的时候也是一种非常有价值的教学手段。驱动设计的模型是领域的一种视图,它也有助于学习其他视图,这些视图可能仅仅用作教学工具或是交



流一般的领域知识。在这些时候，人们可以使用传达与软件设计无关的其他模型的图片或语言。

使用其他模型的一个特殊原因是范围限制。驱动软件开发过程的技术性模型被大规模削减，使得它能用最小的规模完成所有功能。而说明性模型可以包括领域的更多方面，来提供说明严格受范围限制的模型的上下文。

说明性模型为针对特定主题进行更多的讨论方式提供了自由的空间。领域专家使用形象的隐喻进行更清晰的解释，指导开发人员并在专家之间进行协调。说明性模型还帮助人们在完全不同的、多样化、多层次的解释中进行学习。

说明性模型不必成为对象模型。通常情况是两者不相同才最好。在这些模型中避免使用 UML 实际上更有帮助，可以避免一种错误的印象——觉得说明性模型与软件设计是一致的。尽管说明性模型与驱动设计的模型通常都是对应的，但没有确切的相似性。为了避免混淆，就必须分清这种区别。

示例：航运操作与路线

考虑一个跟踪航运公司货物的应用程序，如图 2-4 所示。模型包括如何将港口操作和轮船航次组装进一个货运(路线)计划中的详细视图。但是对于刚入门的人来说，类图并没有多大的辅助说明作用。

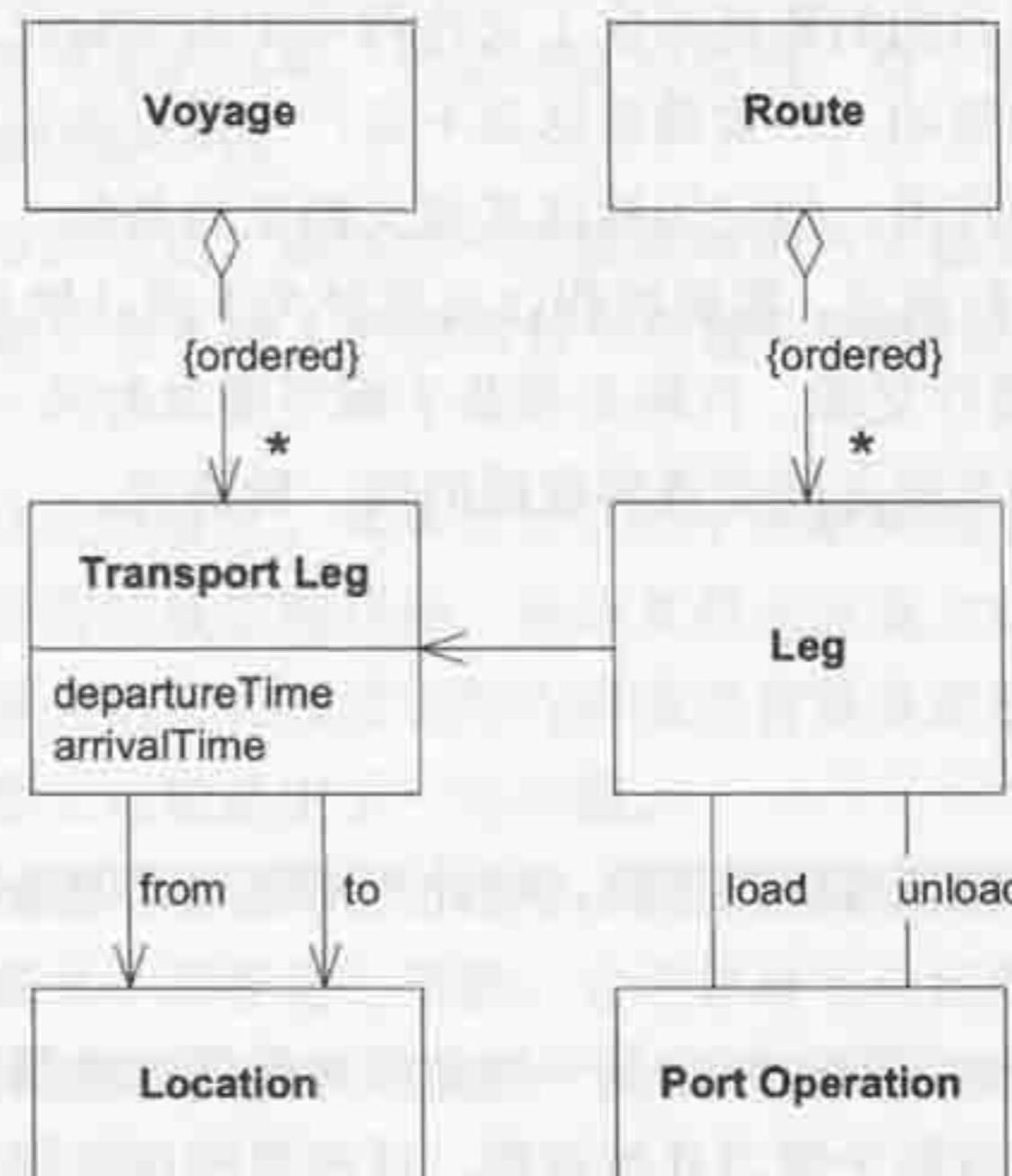


图 2-4 航运路线的类图



在这种情况下，说明性模型可以帮助团队成员理解类图的实际意义。图 2-5 所示为查看同一概念的另一种方式。

图 2-5 中的每一根线或者表示一次港口操作(装载或卸载货物)，或者表示放在仓库中的货物，或者代表运送中的货物。这与类图在细节上并不是对应的，但是强调了领域的关键点。

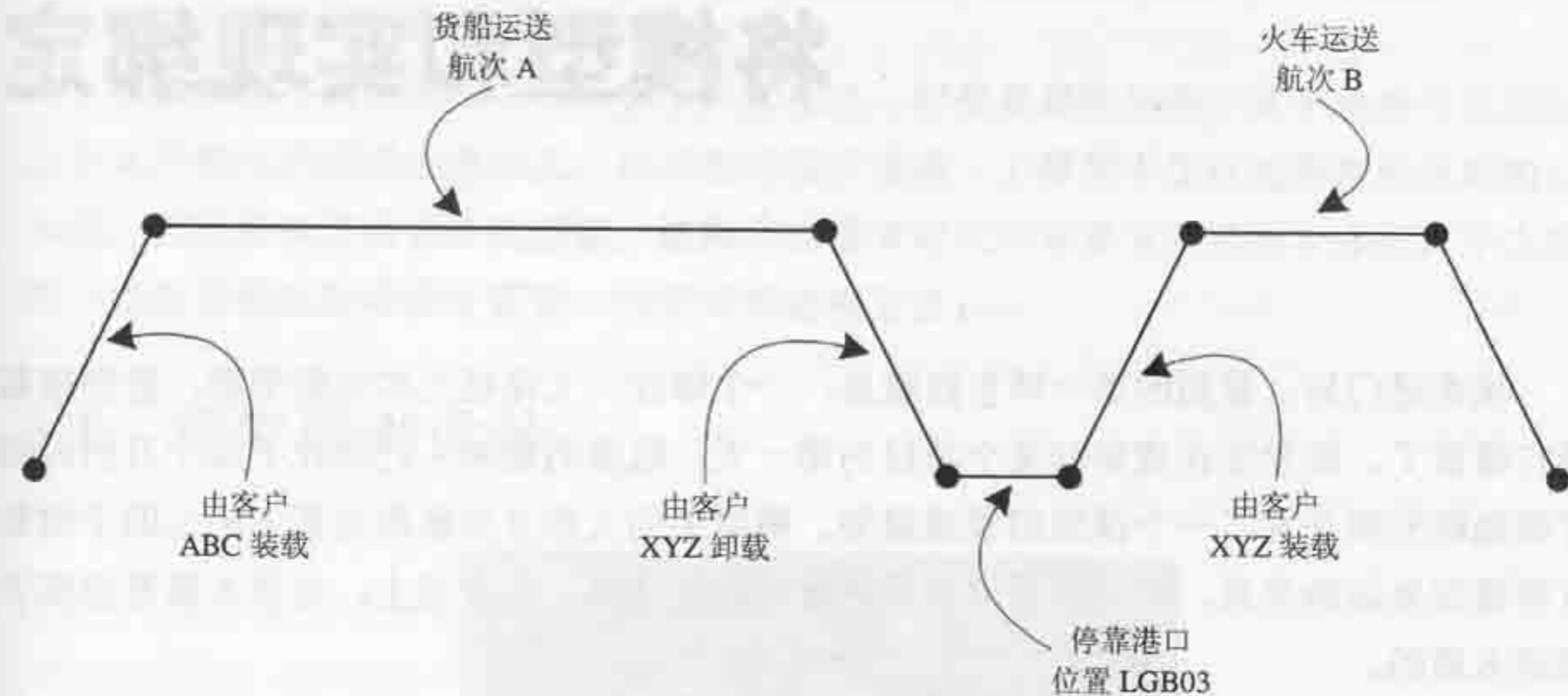


图 2-5 航运路线的说明性模型

这种类型的图形带有它所表示模型的自然语言解释。它能够帮助开发人员和领域专家理解更加严格的软件模型图形。一起查看两种图形要比单独地看任何一种更容易理解。



第3章

将模型和实现绑定

我走进门后，看到的第一样东西就是：一个印在一大张纸上的完整类图，整面墙都被它覆盖了。这发生在我参与某个项目的第一天，这里的聪明人已经花了几个月的时间仔细地研究和开发了一个详细的领域模型。模型中的大部分对象都与其中的三四个对象有着错综复杂的关系，并且关系网中几乎没有自然边界。在这点上，分析人员是忠实于领域本质的。

在这张墙一般大小的图形里，模型确实收集到了不少知识。经过一段时间的学习，我学到了很多(尽管这种学习很难找到方向——更像是在随机地浏览网页)。但是在学习中，我遇到了更多的困难——根本无法了解到应用程序的代码和设计。

当开发人员开始实现应用程序的时候，他们发现彼此纠缠的关系根本无法转换成事务完整性操作中的可存储、可检索的单元，尽管分析人员能够将它们搞清楚。注意，该项目使用了一个对象数据库，因此开发人员甚至不用面对将对象映射到关系表之类的挑战。从根本上说，这个模型并没有提供对实现的指导。

因为这个模型是“正确”的，这是技术分析人员和业务专家深入合作的结果，所以开发人员得出结论：基于概念的对象不能够成为他们设计的基础。因此他们继续开发特别的设计。他们的设计确实使用了一些与数据存储中相同的类名称和属性，但并不基于已经存在的(或者任何)模型。

这个项目有一个领域模型，但是一个模型如果对于可运行软件的开发没有直接帮助，那么这种纸上的模型又有什么用处呢？

几年过后，我看到另一个完全不同的开发过程也导致了相同的结果。这个项目要将一个已经完成的 C++ 应用程序替换成由 Java 实现的新设计。旧的应用程序是在没有考虑



任何对象建模的基础上生成的。像这样的旧应用程序的设计，是在已有的代码上添加一个又一个功能，并不考虑其通用化或抽象性。

可怕的是，这两个开发过程的最终产品竟是如此相似！都具有所要求的功能，但是非常庞大且难于理解，最后逐渐无法进行维护。尽管有几处的实现比较直接，但您无法通过阅读代码来理解系统的目的。除了想象的数据结构外，两个过程都没有很好地利用它们开发环境中的对象范型(paradigm)。

模型具有不同的种类并且承担不同的任务，即使是那些局限于某个软件开发项目的上下文环境中的模型也是如此。领域驱动设计提倡一个模型不仅仅能够帮助早期的分析人员，并且可以作为设计的基础。这种方法要求对代码有着重要的暗示作用。不太明显的一点就是领域驱动设计需要一种不同的建模方法……

3.1 模型驱动设计



曾经被用作计算星体位置的星盘，就是天空模型的一种机械实现

将代码与作为基础的模型紧密关联，会让代码具有含义并使模型起到相应的作用。

一个中世纪的天体仪

古希腊天文学家发明了星盘，后来中世纪伊斯兰科学家对它进行完善。一个旋转的网络结构(rete)代表天球上固定星体的位置。可替换的金属板上刻着当地的球坐标系统，代表不同纬度的视图。相对于盘子来转动 rete，能够计算出一年中任何一天中天体的位置。反过来，给出一个恒星或太阳的位置，可以计算出时间。星盘就是天空的面向对象模型的机械实现。

那些没有领域模型，只是靠编写代码来完成一个又一个功能的项目，不能获得前两章中讨论的知识消化与交流带来的益处。复杂的领域会使他们的工作陷入泥沼。



另一方面，一些复杂的项目确实尝试使用了少许的领域模型，但是他们没有维护好模型与代码之间紧密的联系。他们所开发的模型，开始或许还可以作为有用的探索工具，但是逐渐变得与项目无关甚至会产生误导作用。所有花费在模型上的精力都不会让人们对于设计的正确性放心，因为它们两个是有差别的。

这种连接可能在很多情况下被打断，而打断却常常又是一种有意识的选择。很多设计方法学都提倡一种分析模型，这种分析模型与设计截然不同，通常是由不同的人员开发的。把它叫做分析模型，是因为它是对业务领域进行分析得出的产物，它将业务领域中的概念组织起来，但并没有考虑它在软件系统中所处的位置。分析模型只是一种理解的工具；与实现相混合则被认为是搅浑了清水。在后面的工作中，创建了设计并且与分析模型仅仅保持一种松散的对应。分析模型不是根据头脑中的设计问题创建的，因此它对于那些需要来说是不实际的。

在这种分析中会有知识消化的过程，但是在编码时大多数知识都被遗弃了，开发人员被迫为设计进行新的抽象。因此分析人员获得的、以及在模型中嵌入的知识不能保证会被保留或被重新发现。在这一点上，维持任何设计和松散连接的模型之间的映射都是不合算的。

纯粹的分析模型甚至不能够达到其理解领域的主要目标，因为重要的发现往往出现在设计/实现过程中。非常特殊的、不曾预料的问题总会发生。预先的模型可能会去深入研究一些无关的主题，也会忽视掉一些重要的主题，另外的主题却用了许多对于应用程序没用的方式表现出来。结果是编码工作开始不久，纯粹的分析模型会被丢弃，很多基础的东西得重新返工。再次做同样的工作时，如果开发人员意识到分析是一个隔离的过程，建模可能会使用一种不太认真的方式。如果管理者意识到分析是一个独立的过程，开发团队可能就不会有足够的机会与领域专家接触。

不管是因为什么原因，一个缺乏设计基础概念的软件最多只能是一个能够做有用的事情却不能清楚解释它的行为的机械装置。

如果一个设计，或者它的核心部分，不能够映射到领域模型上，那么这个模型是没有价值的，软件的正确性也是值得怀疑的。同时，模型与设计功能之间复杂的映射常常难于理解，并且在实践中，当设计发生变化时也不可能维护。在分析与设计之间存在致命的隔阂，从任何(分析和设计)活动中获得的知识都无法提供给另一方。

一个分析人员必须用容易理解和易于表达的方式从领域中收集基本概念。设计必须指明一组能够被项目中使用的编程工具构造的组件，这些组件必须能够在目标部署环境中有效地执行，并且能够正确地解决应用程序出现的问题。

模型驱动设计(Model-Driven Design)抛弃了分裂分析模型与设计的做法，而是寻找一



个单独的模型来满足这两方面的要求。将纯粹的技术问题放到一边，设计中的每个对象都担当一个模型中描述的概念上的角色。这就需要我们对选择的模型有更多的要求，因为它必须满足两个完全不同的目标。

对领域进行抽象有很多方式，要解决一个应用问题通常也有很多种设计。这使得绑定模型与设计成为可能。这种绑定不能够以削弱分析、基于技术性考虑作出折衷为代价。我们也不能够接受拙劣的设计，这些设计虽然反映了领域的概念却舍弃了软件设计的原则。这种绑定方式要求一个模型在分析与设计方面都取得良好的效果。当一个模型看起来对于实现不实用时，我们必须重新寻找一个新的模型。当模型没有忠实地表达领域的关键概念时，我们也要重新寻找一个新的模型。这样，建模与设计过程就成为了单个迭代循环。

强制性要求将领域模型与设计紧密地联系起来，使得在各种可能的模型中选择更加有用的模型时又添加了一条新的准则。它要求认真地思考并花费多次迭代和重构，这样才可以得到相应的模型。

因此：

应该设计出能够非常精确地反映领域模型的部分软件系统，这样映射就会明显。如同您尽力让模型更深层次地反映领域一样，再次研究模型并修改，使得它能够让软件更加自然地实现。需要这样的单个模型，除了要支持一种健壮的通用语言之外，还要很好地完成这两个目的。

从模型中得到的术语可以用于设计和职责的基本分配。代码成为模型的表达形式，因此对代码的改变可能也变成对模型的一种改变。它的影响将相应波及到项目中的其他活动。

要将实现与模型严格地绑定在一起，通常需要支持建模范型的软件开发工具和语言，例如面向对象的编程。

有时，不同的子系统会有不同的模型(参见第14章)，但是遍及开发工作的方方面面，从代码到需求分析，对于系统的一个特定的部分只有一个模型可以应用于其中。

单一的模型减少了发生错误的可能性，因为现在的设计是被仔细考虑过后的模型的直接产品。设计，甚至代码本身，都与模型保持着联系。

开发一个能够捕获问题并提供实用设计的单个模型并不像所说的那么简单。您不能够仅仅简单地拿过一个模型，然后将它转化成可使用的设计。这个模型需要仔细地制作以得到实用的实现。设计和实现技术可以使得代码更有效地表达一个模型(参见第II部分)。知识消化过程必须探索模型选项，并将它们精化为实用的软件元素。开发便成为将模型、设计和代码精化为一个单一活动的迭代过程(参见第III部分)。



3.2 建模范型和工具支持

要使模型驱动设计发挥作用，必须在人为误差的范围内严格保证一致性。如图 3-1 所示要使模型与设计之间的紧密联系成为可能，使用一个由允许对模型中的概念创建直接模拟的软件工具支持的建模范型是很必要的。

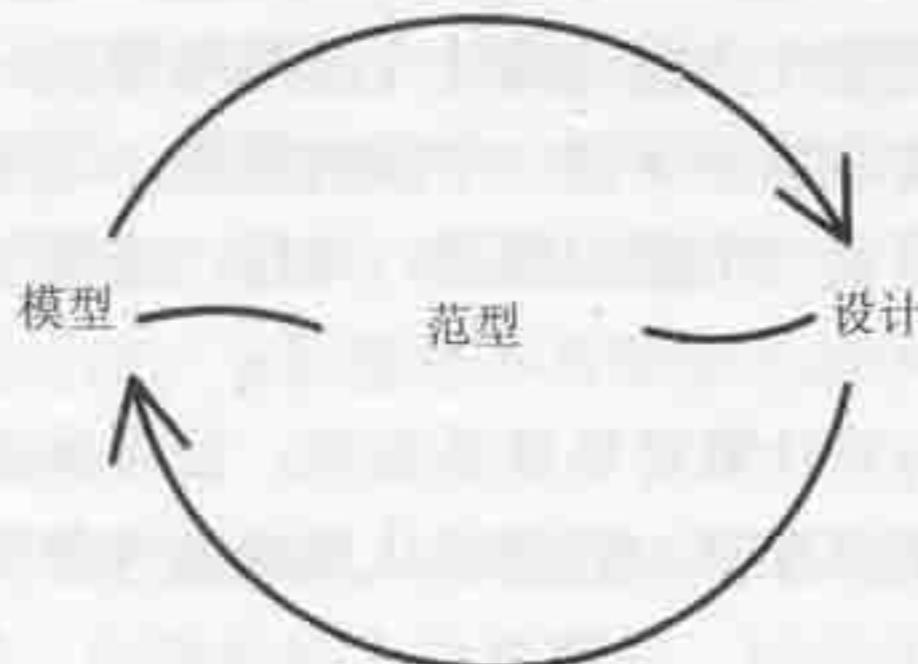


图 3-1 模型、范型和设计的关系

面向对象编程的功能强大，因为它基于一个建模范型并可提供对模型构造的实现。对程序员来说，对象真实地存在于内存中，与其他对象相互关联，它们被组织成类并且通过消息提供可用的行为。尽管许多开发人员仅仅受益于应用对象技术来组织程序代码，然而对象设计的真正突破点在于用代码来表示模型的概念。Java 与其他工具都允许建立与概念上的对象模型类似的对象和关系。

尽管 Prolog 的使用程度从来没有像面向对象语言那样广泛，但它却是模型驱动设计的一种自然选择。在这种情况下，范型是逻辑，模型则是一组应用于其上的逻辑规则与事实。

模型驱动设计在使用像 C 之类的语言时，功能会受到很大的限制，因为没有与纯粹的过程语言相对应的建模范型。这些语言都是过程化的，由程序员告知计算机应该执行的一系列步骤。尽管程序员可能会考虑领域中的概念，但程序本身只是一系列对数据的技术操作。结果可能是有用的，但是程序并没有捕获到太多的含义。过程化语言通常支持复杂的数据类型，这些复杂的数据类型与领域中更加自然的概念相对应，但是它们只是一些经过组织的数据，它们并没有捕获到领域的活动的方面。结果就是，用过程化语言编写的程序不会采用领域模型中概念性的连接方式，而是基于预先安排好的执行路径将复杂的函数连接在一起。

在听说面向对象编程之前，我通过编写 FORTRAN 程序来解决数学模型，这是 FORTRAN 有优势的领域。数学函数是这类模型最主要的概念组件，FORTRAN 可以清



晰地表达它们。尽管如此，还是无法捕获这些函数之外的一些高层含义。大多数的非数学领域都不能够用过程化语言进行模型驱动设计，因为领域本身不能够像用数学函数或过程中的步骤那样来表达概念。

面向对象的设计，这种现在在大多数大型项目中占主导地位的范型，是本书中使用的主要方法。

示例：从过程化到模型驱动

正如第 1 章所讨论的，一个印刷电路板(PCB)可以被看作是连接各种元件的引脚的电导体的集合(称为 net)。(一张)印刷电路板上通常会有上万个 net。一种叫作 PCB 布局工具的专用软件，可以找到使得所有的 net 彼此不相交或不相互干扰的物理安排。它通过满足由设计人员规定的大量约束来限定布局的方式，达到优化路径的目的。尽管 PCB 布局工具相当完善，但它们还存在一些缺点。

一个问题就是这成千上万的 net 中每一个都拥有各自的布局规则。PCB 工程人员将许多 net 自然分组，认为应遵守同样的规则。例如，一些 net 构成总线，如图 3-2 所示。

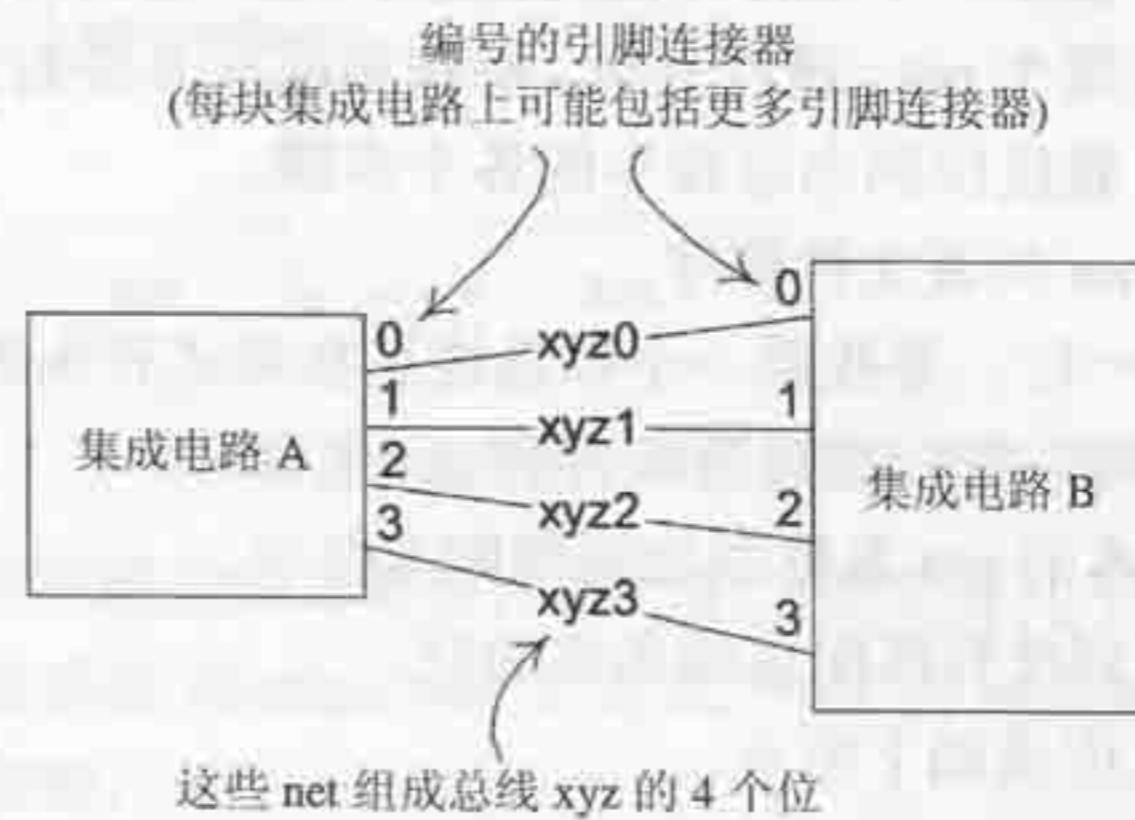


图 3-2 总线和 net 的说明图

通过将 net 堆砌到一个总线，大概一次 8、16 或 256 个，工程人员将工作量降低到了一个易于管理的规模，这样做提高了生产率并且减少了错误。问题是，布局工具中没有类似于总线这样的概念。规则不得不应用于上万个 net，每次只能处理一个 net。

一种机械的设计

无计可施的工程人员在布局工具的这种限制下，只得编写分析布局工具数据文件的脚本并将规则直接插入文件，然后每次将它们应用于整个总线。

布局工具在一个 net 列表文件中存储每个电路连接，如下所示：



第1部分 让领域模型发挥作用

Net Name	Component.Pin
Xyz0	A.0, B.0
Xyz1	A.1, B.1
Xyz2	A.2, B.2
...	

它在一个文件中以下面的格式存储布局规则:

Net Name	Rule Type	Parameters
Xyz1	min_linewidth	5
Xyz1	max_delay	15
Xyz2	min_linewidth	5
Xyz2	max_delay	15
...		

工程人员对 net 使用一种严格的命名约定,这样对于数据文件按字母顺序排序可以使得同一个总线中的 net 位于一个已排序的文件中。然后他们编写的脚本便可以对文件进行分析并根据总线修改每个 net。进行分析、操作和编写文件的实际代码过于冗长和难于理解,因此在例子中,我仅仅列出过程中的各个步骤。

- (1) 根据 net 名称将 net 列表文件排序。
- (2) 读取文件中的每一行,寻找第一个以总线名称形式开头的行。
- (3) 对于名称匹配的每一行,通过分析得到 net 名称。
- (4) 将带有规则文本的 net 名称添加到规则文件中。
- (5) 从步骤(3)重复直到没有匹配总线名称的行。

因此总线规则的输入应该如下所示:

Bus Name	Rule Type	Parameters
Xyz	max_vias	3

添加了 net 规则的文件结果如下所示:

Net Name	Rule Type	Parameters
Xyz0	max_vias	3
Xyz1	max_vias	3
Xyz2	max_vias	3
...		



我猜想第一个编写这样的脚本的人仅仅只有这样一个简单的需要，并且只有这是唯一的需求时，那么这样的脚本会很有意义。但是在实际环境中，会有许多个脚本。它们会进行重构来共享排序和字符串匹配的函数，并且如果语言支持函数调用来封装细节，这些脚本看起来会与像上面总结的步骤十分相似。然而，它们仅仅是文件的操作。不同的文件格式(有若干种)需要从头做起，即使它们所使用的对总线分组和应用规则到总线的概念是相同的。如果您想要更多的功能和交互性，则每一步都要进行很多工作。

编写脚本的工作人员努力要做的是为工具的领域模型补充“总线”的概念。他们的实现是通过排序和字符匹配来推断总线的存在，但是并没有明确地对概念进行处理。

模型驱动设计

前面的讨论已经描述了领域专家用来考虑问题时所使用的概念。现在我们需要将这些概念更明确地组织进模型，作为软件的依据，如图 3-3 所示。

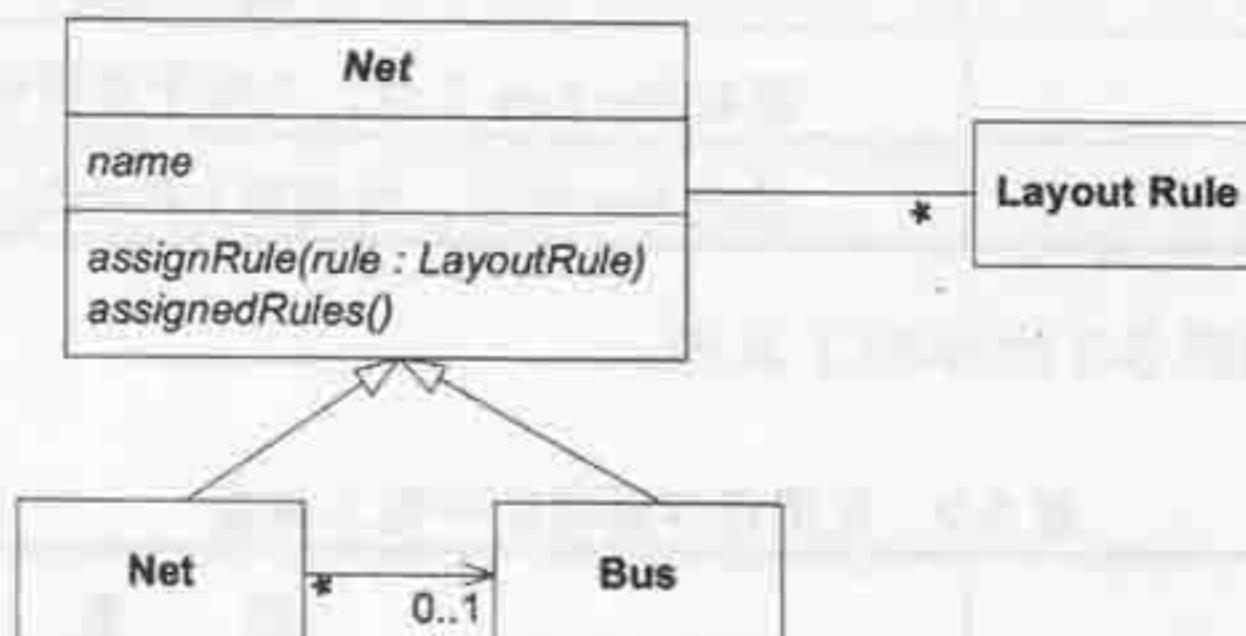


图 3-3 面向布局规则高效分配的类图

用面向对象的语言实现这些对象后，核心的功能变得几乎不重要了。

`assignRule()`方法能够在 `AbstractNet` 上实现，`Net` 的 `assignedRules()` 方法使用了它自己的规则及其 `Bus` 的规则。

```

abstract class AbstractNet {
    private Set rules;

    void assignRule(LayoutRule rule) {
        rules.add(rule);
    }

    Set assignedRules() {
        return rules;
    }
}

class Net extends AbstractNet {
  
```



```
private Bus bus;

Set assignedRules() {
    Set result = new HashSet();
    result.addAll(super.assignedRules());
    result.addAll(bus.assignedRules());
    return result;
}
```

当然，应该有大量的支持代码，但是这些代码已涵盖了脚本的基本功能。应用程序需要导入/导出逻辑，我们将它们封装进一些简单的服务当中，如表 3-1 所示。

表 3-1 封装了导入/导出逻辑的服务

服 务	职 责
Net List import	读取 Net List 文件，为每个条目创建 Net 实例
Net Rule export	给出 Net 集合，向规则文件中写入所有附加的规则

我们还需要一些如表 3-2 所示的工具类。

表 3-2 应用程序需要的一些工具类

类	职 责
NetRepository	提供通过名称访问 Net 的能力
InferredBusFactory	对于给定的 Net 集合，使用命名约定推断总线，创建实例
BusRepository	提供通过名称访问总线的能力

现在，启动应用程序，用导入数据初始化仓储：

```
Collection nets = NetListImportService.read(aFile);
NetRepository.addAll(nets);
Collection buses = InferredBusFactory.groupIntoBuses(nets);
BusRepository.addAll(buses);
```

可以对每个服务及仓储进行单元测试。更重要的是，核心领域逻辑也能被测试。下面是最核心行为的一段单元测试(使用了 JUnit 测试框架)：

```
public void testBusRuleAssignment() {
    Net a0 = new Net("a0");
    Net a1 = new Net("a1");
    Bus a = new Bus("a"); //Bus is not conceptually dependent
```

```

    a.addNet(a0);           //on name-based recognition, and so
    a.addNet(a1);           //its tests should not be either.

    NetRule minWidth4 = NetRule.create(MIN_WIDTH, 4);
    a.assignRule(minWidth4);

    assertTrue(a0.assignedRules().contains(minWidth4));
    assertEquals(minWidth4, a0.getRule(MIN_WIDTH));
    assertEquals(minWidth4, a1.getRule(MIN_WIDTH));
}

```

一个交互式的用户界面可以表现一组总线，可以允许用户为其指派规则，或者为实现向后兼容性从规则文件中读取数据。外观(Facade)使得对于任一接口的访问变得简单。它的实现与下面的测试相对应：

```

public void assignBusRule(String busName, String ruleType,
    double parameter){
    Bus bus = BusRepository.getByName(busName);
    bus.assignRule(NetRule.create(ruleType, parameter));
}

```

最后完成：

```
NetRuleExport.write(aFileName, NetRepository.allNets());
```

服务向每个 Net 调用 assignedRules()，然后将它们完全展开。

当然，如果只有一个操作(如示例所示)，基于脚本的方法可能更实用。但是在实际中，通常有 20 个或更多的操作。模型驱动设计很容易伸缩，并且可以包含组合规则以及其他方面也有增强。

第二个设计也适合于测试。它的组件都有经过良好定义的接口并可以进行单元测试。对脚本进行测试的惟一方法是通过比较端到端的文件输入/输出。

一定要记住这种设计并不只是出现在一个单独的步骤中。它采用多次的迭代进行重构和知识消化，将领域的重要概念精炼为一个简单深刻的模型。

3.3 突出主旨：为什么模型对用户很关键

在理论上，您可以向用户呈现任何一个系统视图，无论其底层是什么情况。但在实际中，不匹配可能会轻则导致混乱，坏则产生 bug。考虑一个简单的例子，在 Microsoft



Internet Explorer 的最新版本中，书签的重叠模型是如何让用户产生误解的。¹

Internet Explorer 的用户认为，Favorites 是一个 Web 站点的名称列表，在不同会话中保持不变。但是实现时是将一个 Favorite 看作一个包含 URL 的文件，它的文件名被放置在 Favorites 列表中。如果 Web 页面的标题包含对 Windows 文件名来说非法的字符时就会出现问题。假设一个用户要存储一个 Favorite 并且键入如下名称“Laziness: The Secret to Happiness”。这时就会出现一个错误信息“A filename cannot contain any of the following characters: \ / : ? “ < > |”。那么用什么文件名呢？另一方面，如果 Web 页面标题已经包含了一个非法字符，Internet Explorer 会悄悄地将它去掉。这种情况下，丢失的数据可能是无害的，但是这毕竟不是用户所期望的。对大多数应用来说，悄悄地改变数据是完全不能接受的。

模型驱动设计要求只在一个模型下进行工作(如在任何一个单独的上下文中，第 14 章中将会有相应讨论)。大多数的建议和示例都是针对将分析模型与设计模型相分离的问题，但是这里我们遇到的问题来自两个不同的模型：用户模型与设计/实现模型。

当然，绝大多数情况下，没有经过整理的领域模型视图大多都不便于用户使用。但是尝试在 UI 中建立一个模型的映像(而不是领域模型)可能会导致混乱，除非映像非常完美。如果 Web Favorites 实际上只是一个快捷文件的集合，那么应该将这个事实告知用户以消除对模型的误解。这样不仅减少了一些特征的混淆，用户也可以根据他所了解的文件系统的知识来处理 Web Favorites 相关问题。例如，他可以使用文件浏览器组织这些文件，而不是使用 Web 浏览器中提供的拙劣的工具。见多识广的用户还能够更进一步地开发更灵活的方法，在文件系统中存储 Web 快捷方式。通过删除有误导作用的多余模型，应用程序的功能会得到大大提高并更加清晰。在程序员感觉旧的模型已经足够好时，为什么还要让用户学习新的模型呢？

还有一种可选的方法，就是用另一种方式存储 Favorites，比如说将它们存储在一个数据文件中，这样它们就能够用自己的规则表示了。假设这些规则就是应用于 Web 页面的一些命名规则。这又会提供一个单独的模型。这个模型可以告知用户所有关于 Favorites 的 Web 站点命名方法。

当一个设计基于一个反映了用户和领域专家所关心的基本内容的模型时，相较于其他设计方法来说，这种方法能够在更大程度上向用户揭示设计的主旨。揭示模型的主旨使用户有更多的机会来挖掘软件的潜能并产生一致性的可预测行为。

¹ Brian Marick 向我提起过这个例子。



3.4 实践型建模人员

制造是软件开发的一种通用隐喻。从这个隐喻中可以得到一个推论：高度熟练的工程人员进行设计，较低水平的劳动力装配产品。这种隐喻使得很多项目陷入困境，原因很简单——软件开发完全就是设计。所有的团队中每个成员都有专门的角色，但是将分析、建模、设计和编程的职责完全隔离起来，会阻碍模型驱动设计。

在一个项目中，我的任务是协调不同的应用程序团队并帮助开发可以驱动设计的领域模型。但是管理部门认为建模人员就应该进行建模工作，编写代码是一种技能的浪费，因此我被禁止编程或与程序员处理细节上的问题。

在一段时间内情况看起来还好。我与领域专家及不同团队的开发领导们共同工作，我们消化知识并精化出了一个优秀的核心模型。但是这个模型并没有起作用，原因有两个。

第一，模型的一些意图在交接的时候遗漏了。模型的整体效果受细节的影响很明显（将在第II和第III部分讨论），并且这些细节问题并不总能够在一个UML图或一次普通的讨论中体现出来。如果我能够卷起袖子与其他开发人员共同工作，提供一些作为示例的代码，并提供一些紧密的支持，团队就应该能够理解模型的抽象并根据它们进行工作。

另一个原因是模型与实现和技术之间的相互作用所得到的反馈过于间接。例如，模型的某些方面在我们的技术平台上被证明广泛效率低下，可是几个月时间内我都没有得到这个反馈信息。相对较小的改进能够修复这些问题，但是到那时它却没有起作用。开发人员可以很自如地用他们的方式来编写能够运行的软件——并不需要借助模型，这些模型就是被使用，也已经简化成了一些数据结构。开发人员若把婴儿随洗澡水一起扔出去，还能怎么选择呢？他们不能再继续按照象牙塔中架构师的指令去冒险了。

像往常一样，这个项目的初始环境倾向于不让建模人员进行过多的参与实现。对于在项目中使用的大部分技术，我已经有很丰富的实践经验。在我的角色改变之前，我在同一个项目中甚至领导过一个小的开发团队，因此我对项目中的开发过程与编程环境非常熟悉。如果将建模人员与实现过程分离，即使我有上述种种优势，也无法高效地工作。

如果编写代码的人认为他们不对模型负责，或者并不理解如何让模型在一个应用程序中发挥作用，那么模型对于软件来说根本就没有用。如果开发人员没有意识到改变代码会同时改变模型，那么他们的重构工作只会减弱模型的作用而不是增强。其间，当一个建模人员与实现过程相分离的时候，他/她永远不会学会（或者说是错过了）对实现上的一些约束的感觉。模型驱动设计的基本约束——模型要支持高效的实现和模型要抽象关键的领域知识——已经丢失了一半，最终的模型变得不再实用。最后，如果分工阻碍了协作，在对模型驱动设计进行编码时，不能传递细节，那么经验丰富的设计人员的知识



和技术都不能够传递给其他的开发人员。

要求实践型的建模人员并不意味团队成员不能有专门的角色。每个敏捷过程，包括极限编程，都定义了团队成员的角色，另外还会有一些趋向于自然出现的非正式专业。当模型驱动设计、建模和实现中相互关联的两个任务需要被分离时，问题就会出现。

整体设计的有效性对细粒度设计和实现决策的质量和一致性非常敏感。对于模型驱动设计来说，代码的一部分就是模型的表达，改变代码的同时也应该对模型作相应改变。程序员就是建模人员，不管他喜欢不喜欢。因此最好将项目组织成利于程序员更好地建模的形式。

因此：

负责建模的技术人员必须花时间接触代码，而不论他或她是否在项目中担当主要角色。任何负责代码修改的人员都必须学习通过代码表达模型。每个开发人员都必须参与一些级别的模型讨论中，并与领域专家接触。负责不同工作的人员都必须自觉地和接触代码的人交流，通过通用语言动态交换模型想法。

建模与编程的完全分离是不可行的，然而大型的项目仍然需要技术领导协调高层设计和建模，并帮助作出最为困难和关键的决策。第IV部分“策略性设计”将教您处理这样的决策，同时激发更多的创造性思维，定义高层技术人员的角色和责任。

领域驱动设计通过把模型实现来解决应用程序中的问题。通过知识消化，团队从大量无秩序的信息中提炼出实用的模型。模型驱动设计与模型和实现紧密关联。通用语言是开发人员、领域专家和软件之间信息流通的渠道。

基于对核心领域的基本理解，最终得到的是能够提供丰富功能的软件。

正如上面说过的，模型驱动设计要成功，受详细设计决策影响很大，这些决策是下面几个章节的主题内容。