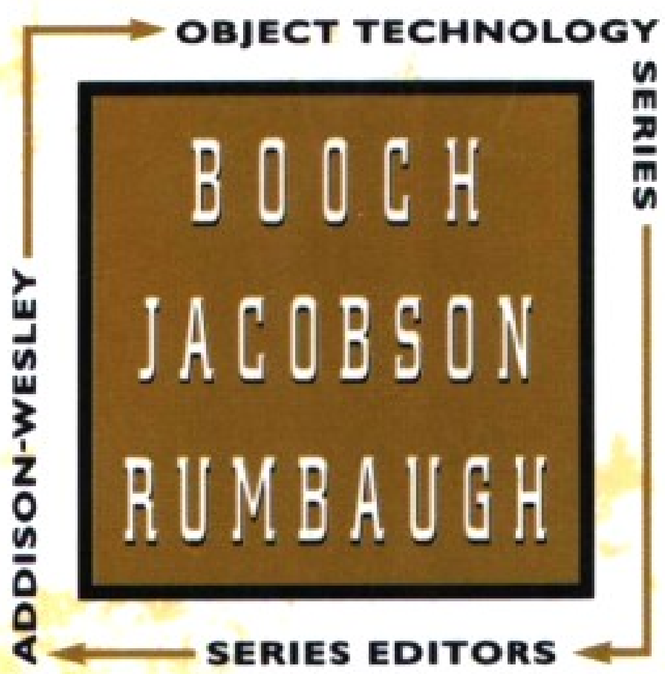
软件工程技术丛书

统一软件开发过程

The Unified Software Development Process



Ivar Jacobson
(美) Grady Booch 著
James Rumbaugh

周伯生 冯学民 樊东平 译

机械工业出版社
China Machine Press
Addison-Wesley

《UML 参考手册》

ISBN 7-111-08220-6/TP · 1610
页数：440 页　　定价：69.00 元

《UML 用户指南》

ISBN 7-111-07564-1/TP · 1192
页数：356 页　　定价：48.00 元

即将出版

《UML和模式应用：
面向对象分析与设计导论》

The Unified Software Development Process

统一软件开发过程

这本划时代的著作透彻地介绍了用于软件开发的统一过程，并把重点放在从实践的角度讨论如何使用统一建模语言（UML）进行建模。统一过程超出了面向对象分析和设计的范畴，清楚地说明了支持整个软件开发生命周期的一系列经过证实的技术，其结果是一个用况驱动的、以构架为中心的、迭代和增量式的基于构件的过程。

统一过程充分利用了作为工业标准的统一建模语言(UML)。本书用UML模型阐明这种已实际应用的新过程，证明符号和过程是如何互相补充的。作者清晰地描述了模型中所用的不同层次的构造的语义和符号。在模型的语境中，描述了诸如用况、参与者、子系统、类、接口、主动类、过程、线程、节点以及大部分关系等构造。熟悉本书作者过去工作的对象技术实践者和软件工程师，将把这本《统一软件开发过程》作为学习当前最好的软件开发实践的有效途径。

本书的作者Ivar Jacobson、Grady Booch和James Rumbaugh是统一建模语言的创始人。世界公认，这三人对开发对象技术做出了许多重大的贡献，包括对象工厂（OOSE）过程、Booch方法和对象建模技术（OMT）等，他们三人目前都在Rational软件公司任职。

适用水平：中、高级

ISBN 7-111-07572-2
9 787111 075721

HZ BOOKS
华章图书

www.china-pub.com
北京市西城区百万庄南街1号　100037
购书热线：(010)68995265，8006100280 (北京地区)

ISBN 7-111-07572-2/TP · 1200
定价：45.00 元

软件工程技术丛书

统一软件开发过程

Ivar Jacobson

（美） Grady Booch 著

James Rumbaugh

周伯生 冯学民 樊东平 译

机械工业出版社
China Machine Press

本书是由UML的三位创始人Ivar Jacobson，Grady Booch，James Rumbaugh亲自撰写的。全书给出了一种以UML作为建模语言进行软件开发的过程指导。书中的内容不是UML固有的组成部分，因为UML只是一种建模语言，并不包括过程指导。实际上，UML独立于过程的特点可以使之用于不同的软件开发过程。但是本书介绍的软件开发过程是三位作者在开发UML时一直在头脑中思考的内容，因此很切合UML的特点。本书对于如何运用UML的概念进行软件开发提供了详细指导，适合参与软件开发的各类人员使用，尤其适合软件项目开发组成员阅读。

Ivar Jacobson, Grady Booch, James Rumbaugh: The Unified Software Development Process.

Copyright © 1999 by Addison Wesley Longman, Inc.

Chinese edition published by arrangement with Addison Wesley Longman, Inc.

All rights reserved.

本书中文简体字版由美国Addison Wesley公司授权机械工业出版社独家出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

版权所有，侵权必究。

本书版权登记号：图字：01-1999-3680

图书在版编目(CIP)数据

统一软件开发过程/（美）雅各布森（Jacobson, I.），（美）布谢（Booch, G.），（美）朗博（Rumbaugh, J.）著；周伯生等译. - 北京：机械工业出版社，2002.1

(软件工程技术丛书)

书名原文：The Unified Software Development Process

ISBN 7-111-07572-2

I. 统… II. ①雅… ②布… ③朗… ④周… III. 软件开发 IV.TP311-52

中国版本图书馆CIP 数据核字（2001）第047726号

机械工业出版社（北京市西城区百万庄大街22号 邮政编码 100037）

责任编辑：姚 蕾

北京市密云县印刷厂印刷 · 新华书店北京发行所发行

2002年1月第1版 · 2002年6月第4次印刷

787mm × 1092mm 1/16 · 24.25印张

印数：15 001-19 000册

定价：45.00元

凡购本书，如有倒页、脱页、缺页，由本社发行部调换

译者序

随着计算机硬件性能的不断提高和价格的不断下降，其应用领域也在不断扩大。人们在越来越多的领域希望把更多、更难的问题交给计算机去解决。这使得计算机软件的规模和复杂性与日俱增，因而软件技术不断地受到新的挑战。20世纪60年代软件危机的出现就是因为系统的复杂性超出了人们在当时的技术条件下所能驾御的程度。此后在软件领域，从学术界到工业界，人们一直在为寻求更先进的软件方法与技术而奋斗。每当出现一种先进的方法与技术，都会使软件危机得到一定程度的缓和。然而这种进步又立刻促使人们把更多、更复杂的问题交给计算机去解决。于是又需要更先进的方法与技术。

开发一个具有一定规模和复杂性的软件系统和编写一个简单的程序大不一样。其间的差别，借用G. Booch的比喻，如同建造一座大厦和搭一个狗窝的差别。大型的、复杂的软件系统的开发是一项工程，必须按工程学的方法组织软件的生产与管理，必须经过分析、设计、实现、测试、维护等一系列的软件生命周期阶段。这是人们从软件危机中获得的最重要的教益。这一认识促使了软件工程学的诞生。编程仍然是重要的，但是更具有决定意义的是系统建模。只有在分析和设计阶段建立了良好的系统模型，才有可能保证工程的正确实施。正是由于这一原因，许多在编程领域首先出现的新方法和新技术，总是很快地拓展到软件生命周期的分析与设计阶段。

面向对象方法正是经历了这样的发展过程，它首先在编程领域兴起，作为一种崭新的程序设计范型引起世人瞩目。继Smalltalk-80之后，20世纪80年代又有一大批面向对象的编程语言问世，标志着面向对象方法走向成熟和实用。此时，面向对象方法开始向系统设计阶段延伸，出现了如Booch86、GOOD（通用面向对象的开发)、HOOD（层次式面向对象的设计）、OOSD（面向对象的结构设计）等一批OOD（“面向对象的设计”或“面向对象的开发”的缩写）方法。但是这些早期的OOD方法不是以面向对象的分析（OOA）为基础的，而主要是基于结构化分析。到1989年之后，面向对象方法的研究重点开始转向软件生命周期的分析阶段，并将OOA和OOD密切地联系在一起，出现了一大批面向对象的分析与设计（OOA&D）方法，如Booch方法、 Coad/Yourdon方法、Firesmith方法、Jacobson的OOSE、Martin/Odell方法、Rumbaugh等人的OMT、Shlaer/Mellor方法等等。截至1994年，公开发表并具有一定影响的OOA&D方法已达50余种。这种繁荣的局面表明面向对象方法已经深入到分析与设计领域，并随着面向对象的测试、集成与演化技术的出现而发展为一套贯穿整个软件生命周期的方法体系。目前，大多数较先进的软件开发组织已经从分析、设计到编程、测试阶段全面地采用面向对象方法，使面向对象无可置疑地成为当前软件领域的主流技术。

各种面向对象的分析与设计方法都为面向对象理论与技术的发展做出了贡献。这些方法各有自己的优点和缺点，同时在各自不同范围内拥有自己的用户群。各种方法的主导思想

以及所采用的主要概念与原则大体上是一致的，但是也存在不少差异。这些差异所带来的问题是，不利于面向对象方法向一致的方向发展，也会给用户的选择带来一些困惑。为此，Rational公司的G. Booch和J. Rumbaugh决定将他们各自的方法结合起来成为一种方法。1995年10月发布了第1个版本，称作“统一方法”（Unified Method 0.8）。此时OOSE的作者I. Jacobson也加入了Rational公司，于是也加入了统一行动。1996年6月发布了第2个版本UML0.9。鉴于统一行动的产物只是一种建模语言，而不是一种建模方法（因为不包含过程指导），所以自0.9版本起，改称“统一建模语言”（Unified Modeling Language）。在此过程中，由Rational公司发起成立了UML伙伴组织。开始时有12家公司加入，共同推出了UML1.0版本，并于1997年1月提交到对象管理组织（OMG）申请作为一种标准建模语言。此后，又把几家分头向OMG提交建模语言提案的公司扩大到UML伙伴组织中，并为反映他们的意见对UML进一步做了修改，产生了UML1.1版本。该版本于1997年11月4日被OMG采纳。此后UML还在继续改进，目前最新的版本是UML1.3。

关于UML的历史、发起的动机、目标、权衡的问题等，这里不想做更多的介绍，因为读者很快会从这套丛书中的《UML用户指南》的前言中看到更详细的叙述。这里想着重指出的是以下三点：第一点是UML的三位发起人G. Booch、J. Rumbaugh和I. Jacobson是从事面向对象研究的著名专家，他们各自的方法和著作在该领域均具有很大的影响。第二点是众多的大公司加入了UML阵营，对UML的制定和推广提供了强有力的支持。第三点是UML经过数年的努力终于被OMG采纳，成为该组织承认的一种标准建模语言。总之，UML是吸收多种方法的成果、凝结了许多组织和个人智慧的产物。

UML是一种用于对软件密集型系统进行可视化、详述、构造和文档化的建模语言，主要适用于分析与设计阶段的系统建模。UML最主要的特点是表达能力丰富。因为它从各种OOA&D方法中吸取了大量的概念，并在“UML语义”、“UML表示法指南”、“对象约束语言规格说明”等UML文献中对这些概念的语义、图形表示法和使用规则做了完整而详细的定义。可以说，UML对系统模型的表达能力超出了以往任何一种OOA&D方法。当然，随之而来的问题是，它的复杂性也超出了以往任何一种方法。

UML的问世受到计算机软件界的广泛重视，因为它代表了一种积极的方向——多种方法相互借鉴、相互融合、趋于一致、走向标准化。建模语言的标准化将为软件开发商及其用户带来诸多便利。因此，在美国等国家已有大量的软件开发组织开始用UML进行系统建模。学习和使用UML已经成为一种潮流。我国软件界对UML也相当关注。许多研究人员和技术人员已在数年前开始学习和研究UML。更有许多人想学习UML，但苦于找不到合适的书籍。由于UML的复杂性，仅通过UML的标准文献来学习和使用它确实不是一件轻松的事。以往国内外也曾发表过一些介绍或评述UML的著作或论文，但是与UML的丰富内容相比，这些介绍远不能满足读者的要求。

值得高兴的是，UML的三位主要设计者G. Booch、J. Rumbaugh和I. Jacobson现在已亲自撰写了这套详细阐述UML的著作，由Addison Wesley于1999年出版。这套著作对UML进行了详细、深入而准确的介绍和论述，而且语言生动、深入浅出、实例丰富、图文并茂。这是一套教会读者掌握和使用UML的教材和指导手册，而不是枯燥的标准文献。对于想学

习和使用UML的广大读者，这是一套难得的好书。为了使中国的读者能够更好地从中受益，我们在机械工业出版社华章公司的恳切建议下，分头翻译了这三本书，即《UML用户指南》、《UML参考手册》和《统一软件开发过程》。

三本原著都是由这三位作者合著，既各自独立，又有很强的内在联系。其中《UML用户指南》介绍了UML的基础知识，包括UML的术语、规则和语言特点，以及如何运用该语言去解决常见的建模问题，初学者学习UML最好从阅读该书开始。《UML参考手册》对UML的组成和概念做了详细的介绍，包括这些概念的语义、语法、表示法和用途，是一本适合软件专业人员使用的方便而全面的参考读物。《统一软件开发过程》给出了一种以UML作为建模语言进行软件开发的过程指导。其内容不是UML固有的组成部分，因为被OMG采纳的UML只是一种建模语言，并不包含过程指导。实际上UML是独立于过程的，可以用于不同的软件过程。但是该书介绍的软件开发过程是三位作者在开发UML时一直在头脑中思考的，因此很切合UML的特点。该书对于如何运用UML的概念进行软件开发提供了详细指导，适合软件专业人员使用。

鉴于UML本身以及这套著作的重要意义，译者在翻译这些著作时采取了特别慎重和严谨的态度。力求准确和通顺。在翻译过程中，一个重要问题是要使这套书中的专业术语的中文译法取得一致。这三本书的译者以往曾分别开展过一些与UML有关的研究和写作，对有些术语的译法互有差异。本次翻译工作中，所有译者在机械工业出版社华章公司的组织下进行了多次讨论、研究和交流，首先对所有专业术语的译法统一意见，达成共识。其中某些术语的译法颇难定夺：既要确切反映英文本意，又要符合中文习惯，还要避免与国内已习惯于与其他英文词对应的中文相混淆。经过反复切磋，大部分问题都得到满意的解决。对个别有争议的问题，在充分讨论的基础上采取放弃己见，服从大局的态度。如此形成了一个译法一致的词汇表。此后在翻译过程中还经常以各种交流方式进行磋商和沟通。最终使这套丛书能以一致的面貌呈献给读者。我们也希望这些工作能为UML术语今后在中文翻译中的统一贡献一份力量。

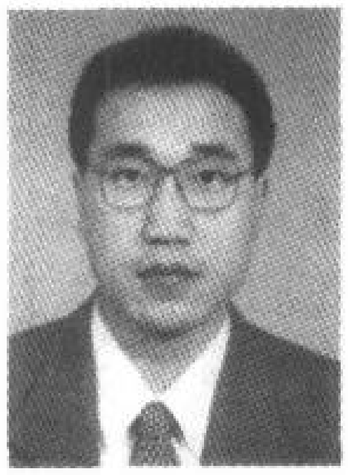
在科技著作的翻译中，保证准确和通顺的关键因素不仅仅是外文水平，还取决于译者真正了解所涉及的技术内容。这套著作的内容远远超出了UML的标准文献，因为除了介绍UML的语法、语义、使用规则之外，其中还包含许多学术思想、技术策略和实践经验。在翻译中遇到的许多疑难问题，是通过进一步研究UML以及有关的学术和技术问题而得到解决的，从而避免了许多讹误。因此，这套著作的翻译不仅是文字方面的工作，还包含译者在技术上的研究。我们希望这些研究最终通过较准确的翻译文字使读者受益。同时诚恳地希望广大读者对可能存在的疏漏和错误之处给予批评和指正。

译　者

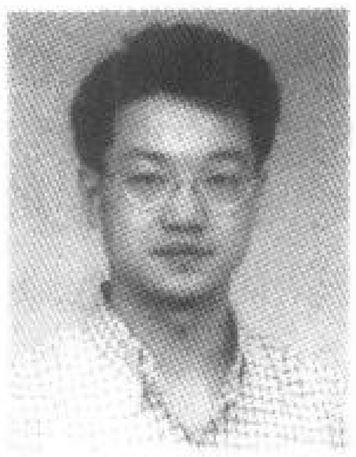
2000年11月于北京

译者简介

周伯生，男，1935年生。北京航空航天大学计算机科学与工程系教授，计算机软件专业博士生导师，专攻软件工程与过程工程。1988年创办北航软件工程研究所；1984年2月～1986年7月，任国家科委与美国ISSI公司合作项目总体组组长；1986年1月～1995年12月，任国家七五和八五科技攻关软件工程及环境总体组成员；1991年10月以来，任北航与美国FunSoft公司“过程工程环境”合作项目技术负责人。



冯学民，男，1968年生。北京航空航天大学软件工程研究所讲师，主要从事软件工程方面的研究及应用项目的开发。



樊东平，男，1972年生，博士。北京航空航天大学计算机科学与工程系讲师，主要从事软件工程及其支持环境方面的研究。

前　言

一些人持有这样的观点，专业公司应该围绕受过严格技能训练的人员组建。这些人员了解他们所要做的工作，并且能正确地去做！他们在工作时，几乎不需要从组织获取相关的方针或者规程上的指导。

然而，这种想法在绝大多数情况下都是错误的，对于软件开发就更是如此。的确，软件开发人员都是经过严格训练的专业人员，但该专业毕竟还很年轻。因此，开发人员需要有组织的指导，在本书中，我们称之为“软件开发过程”。此外，因为在编写这本书的过程中，我们将以前的零散方法集中在一起，所以我们觉得更正确的叫法应该是“统一过程”。这种统一不仅仅集中了三位作者的工作，还有机地结合了众多的致力于UML研究与应用的企业和个人（也包括Rational软件公司中众多的相当关键的人员）的成果。它从数以百计的用户组织（在客户站点上使用过过程的较早版本）的实践经验中吸取了大量重要的信息。

举例来说，一个交响乐队的指挥在一场表演中要做的事情很少，仅仅是告诉演奏者何时开始演奏并帮助他们协调一致。作为指挥的他或她所需要做的工作如此之少，是因为他在排练和演出的准备过程中对乐队进行过指导，而且乐队里的每个成员不但能够非常熟练地演奏自己的乐器，而且还能够独立于其他成员进行演奏。在这里，我们更深层的用意是想说明，每个音乐家都遵循着由作曲家早已安排好的一个“过程”。正是乐谱提供了大量的对演奏加以指导的“方针和规程”。相反，软件开发人员不是独立行动的，而是与其他开发人员和用户相互协作的。他们没有“乐谱”可以遵循，直到有了一个过程（*process*）。

尤其在其软件系统负有重大使命（比如金融、空中交通管制、国防和电信系统等）的公司或组织里，对过程的需求就更加迫切。在这里，我们的意思是，业务的成功管理或公共任务的执行依赖于支持它的软件。这类软件系统正变得更加复杂，要求投入到市场的时间要尽量短，对它们的开发也相应地变得更加困难。基于以上原因，软件工业需要一个指导开发人员的过程，就像交响乐团需要一个作曲家的乐谱来指导演奏一样。

什么是软件开发过程？

一个过程定义了为达到某个确定的目标，需要什么人在什么时间以何种方式做何种工作。对于软件工程而言，其目标是构造一个新的软件产品或者完善一个旧的软件产品。一个有效的过程为有效地开发高质量的软件提供准则，它获取并提出当前技术条件下可行的最佳实践方案。因此，它可降低风险并增强预见性。总的效果是要发扬一种共同的构想与文化。

我们需要一个可用于指导顾客、用户、开发人员和项目经理等参与者的过程。我们需要一个能将软件产业有史以来第一次有机地结合在一起的最好的过程——任何旧的开发过程都

无法做到这一点。最后，我们需要一个广泛适用的过程，以便所有的项目相关人员都能理解自己在所进行的开发工作中的角色。

一个软件开发过程还要能随时间的推移不断地进化。在进化过程中，它能及时调整自己以适应技术、工具、人员以及组织模式的变化。

- **技术**：过程必须建立在当时可以使用的实用技术（如编程语言、操作系统、计算机系统、网络能力、开发环境等）基础之上。例如，20年前可视化建模技术还不是主流技术，因为它的代价太高。因此，那时的过程建造者不得不采取手工绘图的方式，这大大限制了过程发起者把模型引入过程的程度。
- **工具**：过程和工具必须同时发展。工具是过程的一部分。换一个角度来讲，一个广泛应用的过程能对支持它的工具的开发起到促进作用。
- **人员**：过程建造者必须将运作过程所需的技能作一定的限制，使得这些技能是现有开发人员已经具备的或者通过培训能迅速掌握的。在许多领域中，现在都可以嵌入曾经需要广泛技能的技术，比如在基于计算机的工具中要检查模型图的一致性。
- **组织模式**：软件开发人员并不是像交响乐演奏家那样各自独立的专家。他们远非一百多年前Frederick W. Taylor在其《科学的管理》中所论及的自动化工人。过程建立者要让过程适合当前的实际情况——例如，虚拟的组织形态，通过高速线路远距离工作，小公司中的部分业主、支取薪水的雇员、合同工和外协转包商等的结合，以及软件开发人员的持续短缺。

过程工程师需要权衡以上四种因素。此外，这种权衡不仅现在必须存在，而且还要保持到将来。过程建造者必须将过程设计成可以进化的，就像软件工程师设法开发不仅在目前可用而且还能在未来岁月里不断进化的软件。一个过程要经历数年的时间才能达到稳定和成熟的水平，能够经得起商业产品开发的严格考验，并将其使用风险保持在合理的水平。不仅没有通过实践充分检验过的过程有风险，而且开发新产品本身也要冒巨大的风险。基于对以上因素的考虑，过程可以保持相对稳定。在没有充分权衡技术、工具、人员和组织的情况下使用过程将会有相当大的风险。

本书的目的

本书主要阐述当我们在开发统一建模语言（UML）时经常会想到的软件过程。UML给了我们一种标准的方法，对软件密集型系统中的制品进行可视化描述、详细说明、构建、文档化以及相互交流。当然，我们都意识到这样一种语言必须应用到软件开发的全过程。UML不是目标，而是手段。最终的目标是健壮的、有弹性的和可扩充的软件应用系统。要达到这样的目标，既需要过程，又需要语言，而本书编写的目的就是要阐明过程。虽然我们提供了有关UML的简要附录，但这个附录并不是全面的或详尽的。对于详细的UML指南，请查阅《UML用户指南》一书[11]。对于全面的UML参考，请查阅《UML参考手册》一书[12]。

本书的读者

参与软件开发的任何人员都可使用统一软件开发过程。其中主要是为处理生命周期活动（需求、分析、设计、实现和测试——即生成UML模型的工作）的开发组成员编写的。因此，举例来说，本书与分析人员和最终用户（他们详细描述系统所需的结构和行为）、应用开发人员（他们设计满足这些需求的系统）、编程人员（他们将设计转换成可执行代码）、测试人员（他们验证与确认系统的结构和行为）、构件开发人员（他们创建与收录构件）以及项目和产品经理等都有关系。

本书读者最好是基本掌握了面向对象概念。如果具有软件开发以及面向对象程序设计语言方面的经验就更好，但这并不是必需的。

本书的学习方法

本书用大部分的笔墨阐述在UML中认为是重点的活动——需求、分析和设计。正是在这些重点活动中，过程开发了复杂软件系统的**构架**（*architecture*）。然而，我们将开发过程作为一个整体来对待，虽然缺乏细节。最后要运行的仍然是一个可执行程序。要达到这一目标，需要全体开发人员的共同努力和项目相关人员的大力支持。正如你将要看到的，过程取决于大量不同的活动，会产生大量的制品并必须加以追踪。对于所有这些活动，必须加以管理。

任何一本书都无法完整地涵盖一个综合而完整的生命周期过程。这样一本书必须包括设计指南、制品模板、质量指示、项目管理、配置管理、度量等内容，在此无法一一列举。但随着在线访问的产生，前面无法枚举的东西现在已可利用，可以当成新的开发指令加以更新。为此我们向你推荐“Rational统一过程”，它是一个可从Web下载的软件产品，可用来指导软件开发组进行更有效的软件开发实践（要了解更多的信息，请参见http://www.rational.com）。Rational统一过程涵盖了整个软件生命周期，拓展了本书所阐述的统一过程的范围，并提供了本书所没有包含的或者是一笔带过的其他工作流，例如业务建模、项目管理和配置管理等。

统一过程的历史

统一过程是稳定的，因为它是经过30年的发展和实际运用后推出的最终产品。像一个产品那样，它的发展历程（如图1）从“对象工厂过程”开始（1987年发布），经过“Rational对象工厂过程”（1997年发布），到“Rational统一过程”（1998年发布）。它的发展受到了诸多因素的影响，在这里并不一一列举这些因素（我们实际上也并不了解所有这些因素），而把这个问题留给软件考古学家去研究，我们将描述Ericsson方法、Rational方法以及其他一些因素对统一过程的影响。

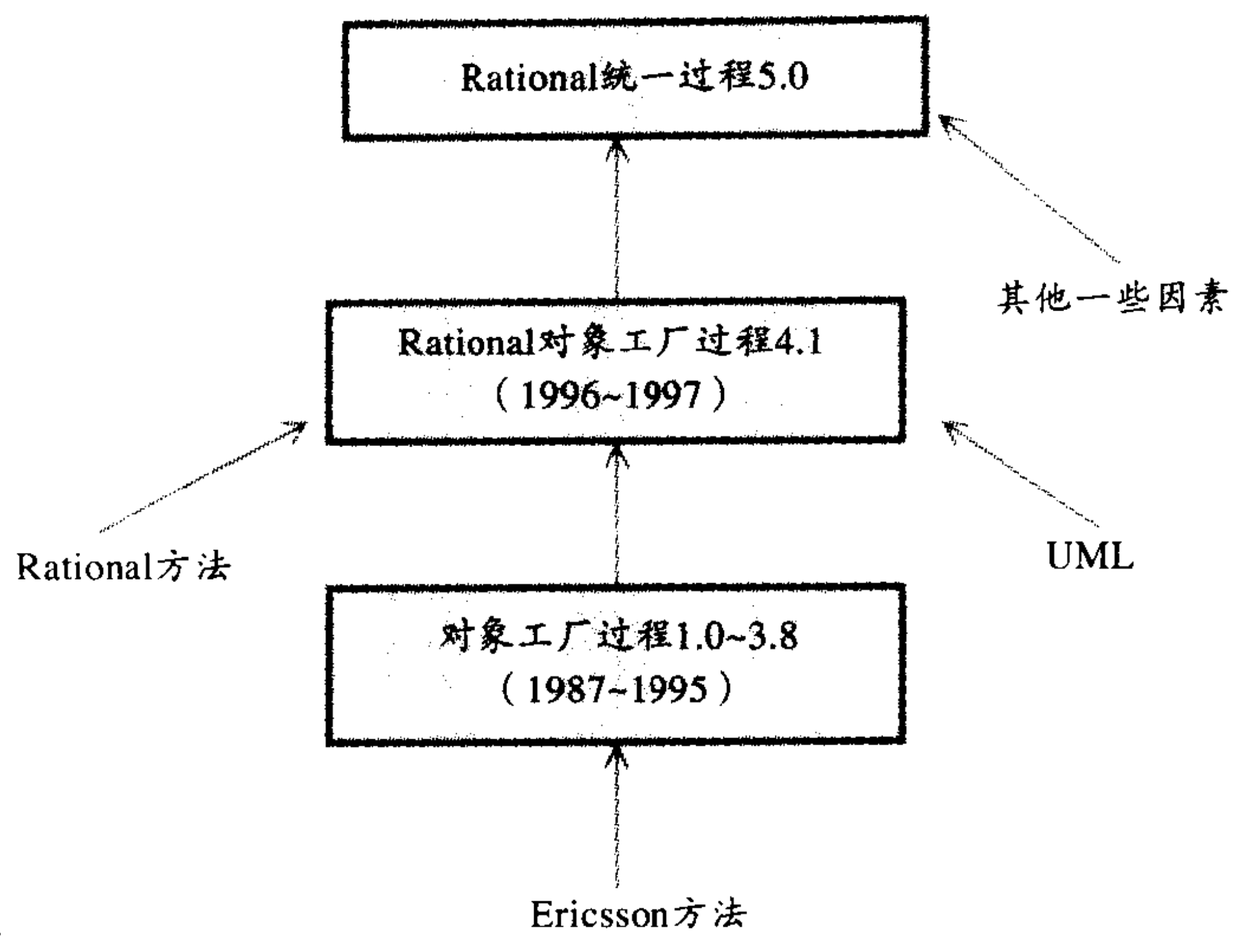


图1 统一过程的发展历程（在灰色的矩形框中指出了产品的版本号）

Ericsson方法

统一过程有着很深的渊源。用Peter F. Drucker的话来说，过程是一个“基于知识的创新”。他忠告我们：“新知识的出现与将其精华转化为可用的技术之间存在着相当大的一段距离，因而，在这一新的技术以产品、过程或服务的形式在市场上出现之前，还会经历另一段相当长的时期”。[1]

产生这段相当长的前导期的原因之一是，基于知识的创新是将很多不同类型的知识融汇贯通，而这是要花时间的。另一个原因是人们需要时间来吸收那些新的思想，将其传播给他人，才能使新思想真正有效。

要阐明统一过程的发展历程，首先让我们回到1967年去具体地概括一下Ericsson（爱立信公司）所取得的成果[14]、[15]、[16]。Ericsson将整个系统模型化成相互联系的模块（在UML中，称之为“子系统”，并当做“构件”实现）的集合。他们用底层的模块装配成较高层的子系统，使整个系统更易于管理。通过遍历以前曾详细描述过的业务实例（现在称为“用况”）来发现这些模块。对于每一个用况，识别出其实现所需的相关模块。了解了每个模块的职能后，他们就编制每一个模块的规格说明，设计出一批带有接口的静态模块图，并将其组合成子系统。这些模块图直接对应于UML的类图或包图的简化版本——称之为“简化”是因为它仅仅表示了用于通信的关联。

设计活动的第一个工作产品是软件的**构架描述**（*architecture description*）。它基于对最关键的需求的理解，简要描述每个模块以及如何将它们组合成子系统。一组模块图描述了模块及其相互联系。在其联系上，信号（即一种消息）用于模块之间的通信。所有消息都在消息库中逐个地加以描述。软件构架描述和消息库是指导开发工作的关键文档，同时也可用于向

客户介绍整个系统。在那时（1968年），客户还不大习惯软件开发人员以类似于工程蓝图的方式向客户介绍软件产品。

对于每个用况，工程师既要准备顺序图，还要准备协作图（现在，在UML中这方面得到了进一步的发展）。这些图描述了模块是如何通过动态通信来实现该用况的。工程师以状态图（只包括状态和转换）和状态转换图（UML活动图的简化版本）的形式编制规格说明。这种使用具有良好定义的接口的模块来设计系统的方法是成功的关键。例如，现在通过另一个提供相同接口的模块来替换其中的某个模块，可以为一个新客户创建该系统的一个新配置。

现在，模块不只是子程序或者源代码构件；它们被编译成可执行的模块，逐个地被安装到目标机器中，并被确定能与其他的可执行模块一起运行。此外，对一个运行中的（如电话呼叫）系统，必须能够在系统运行时就能安装新模块或者替换旧模块。这样的系统不能因为要做更改而停止，就好比不能给一辆正以60英里时速行驶的轿车替换轮胎一样。

本质上讲，这种方法就是我们今天所说的**基于构件的开发**（*Component-Based Development*, CBD）。Ivar Jacobson是该开发方法的创始人。经过“对象前时代”的多年努力，他指引着该方法进化成为一种软件开发过程。

规格说明与描述语言（SDL）

这一时期具有重大意义的进展是：1976年，CCITT（国际电话电报咨询委员会，电信领域的国际标准化组织）公布了用以描述电信系统中功能行为的**规格说明与描述语言**（*Specification and Description Language*, SDL）。该标准受Ericsson方法的重大影响，用一系列相互关联的模块来刻画一个系统，各模块之间仅仅通过消息（标准中称之为“信号”）来相互通信。每一个模块都拥有一系列的“过程”，它们是由SDL术语表示的主动类。一个过程拥有实例，非常类似于面向对象术语中的类。过程实例通过消息产生交互。SDL所推荐使用的图可以看成是UML中的类图、活动图、协作图和顺序图的特化。

因此，SDL是一种特化的对象建模标准。SDL被定期地更新，现在仍有10 000多个开发人员在使用，并且仍得到一些软件工具供应商的支持。SDL是在20多年前开发的，远超前于所处的时代。但是，SDL是在对象建模还没有成熟的时候开发的，看来它将会被UML所替代，而后者在1997年已成为一种标准。

对象工厂

1987年，Ivar Jacobson 离开了Ericsson公司，并在斯德哥尔摩建立了“Objectory AB”。在后来的八年时间里，他和助手们一起开发了一个称为对象工厂（Objectory，它是“Object Factory”的缩写）的过程产品，并将该产品推广到电信以外的许多行业和瑞典以外的许多国家。

尽管Ivar Jacobson在Ericsson的工作中已经提出了用况（*use case*）的概念，但直到现在

它才有了一个名称（是在1987年的OOPSLA讨论会上提出的），并随着图示化技术的发展，这种思想才扩展到多个不同的应用领域。正是用况使得开发过程更加清晰；正是构架引导了开发人员并让项目相关人员也走到了开发的前台。

各种相继的工作流表征为一系列的模型：需求—用况、分析、设计、实现和测试。一个模型是对一个系统的一种刻画。在该系列中模型之间的关系对开发人员而言非常重要，可看作是从模型序列的一端到另一端某个功能特征的演进路线。事实上可跟踪性已成为用况驱动的开发过程的先决条件。开发者可以通过模型顺序跟踪到对应于某个用况的源代码，或者当出现问题时能够进行逆向跟踪。

对象工厂过程在发展中产生了一系列的版本，从1988年的Objectory 1.0版本到1995年的第一个在线版本（Objectory 3.8版本）（参阅[2]中的Objectory介绍）。

需要着重指出的是，对象工厂产品本身也被看成是一个系统。这种将过程描述成一个系统产品的方法为从早期的（对象工厂）版本开发新的（对象工厂）版本提供了良好的途径。这种设计对象工厂的方法可以很轻松地加以裁剪以适应不同开发组织的需要。事实上，对象工厂软件开发过程的一个独特的特征是：其本身就是工程化的。

对象工厂的开发实践也为如何去巧妙地设计业务通常运作的过程提供了经验。同样的原理是适用的，并在1995年的一本书中进行了讨论[3]。

Rational方法

Rational软件公司在1995年秋季引进了Objectory AB，并在已有的软件开发过程基础之上统一了基本规则，从而获得了新的突破。Rational公司开展了大量的软件开发实践，其中很多是对对象工厂已有内容的补充。

例如，James E. Archer Jr. 和Michael T. Devlin在1986年的回忆[4]中曾提到："Rational于1981年开始研制一个能提高大型软件开发效率的交互式环境。"他们还进一步提到，在这项工程中，面向对象设计、抽象、信息隐藏、可复用性以及原型化等技术都显得很重要。

自1981年以来，大量的书籍、文章和内部文献资料都详细地描述了Rational的开发过程。然而，对于过程的建立最具贡献意义的两个因素也许是对构架和迭代式开发的强调。例如，1990年Devlin写了一篇关于构架驱动的迭代式开发过程的前瞻性文章；Philippe Kruchten（在Rational公司中负责构架实践方面的工作）也发表过一些有关迭代和构架的文章。

我们引用其中一点，一篇有关构架的文章一般都表达了四个方面的视图：逻辑视图、过程视图、物理视图和开发视图，另外再加上一种用用况或者场景阐述前面四种视图的视图[6]。根据Kruchten在几个大型项目中的实践经验，拥有一组视图比试图把所有东西都塞进一种视图中更有价值。多视图使得项目相关人员和开发人员都能通过适当的角度去发掘他们所想要的东西。

一些人曾认为迭代式开发有些混乱无序和漫无边际。在迭代中引进四阶段法（初始、细化、构造和移交）可以设计出更好的结构和更好的迭代控制过程。阶段法强制迭代按照一

定的顺序进行。阶段的详细计划编制和阶段内迭代的具体排序是Walker Royce和Rich Reitman的小组共同努力的结果，此外也要归功于 Grady Booch和Philippe Kruchten的长期参与。

Booch是Rational公司创立和发展的见证人。1996年，他在一本书中声明了两条关于构架和迭代的“首要原则”：

- “构架驱动方式的开发方法通常是开发最复杂的软件项目的最好方法。”
- “一个成功的面向对象的项目必须采用一种迭代增量式的过程。”[7]

Rational对象工厂过程：1995~1997

在合并前，对象工厂3.8版本已经说明了作为产品的软件开发过程是如何进行开发和建模的。它设计了软件开发过程的原始构架，建立了一套记录过程结果的模型。在用况建模、分析和设计等方面取得了很好的进展，但在不同于用况的需求管理、实现和测试等其他方面却没有取得太大进展。此外，在项目管理、配置管理、实施以及开发环境的准备（工具和过程选取）等方面它也很少涉足。

现在，我们将Rational的经验和实践加入，从而形成了Rational对象工厂过程4.1版本。特别是，新版本中加入了阶段和受控的迭代方法。构架以构架描述的形式清晰地展现，并成为软件开发组织的“圣经”。此外，建立了构架的精确定义，把它看成是系统组织的重要部分，并把它描述为模型的构架视图，从而使迭代式开发方法从相对一般的概念发展成为把构架放在首位的风险驱动的开发方法。

当前，UML还在继续发展，并用作Rational对象工厂过程（Rational Objectory Process, ROP）的建模语言。本书的三位作者是UML的创始人。以Philippe Kruchten为首的过程开发组，通过加强项目管理（例如，基于Royce的研究成果[8]），弥补了ROP的一些不足。

统一建模语言

在20世纪90年代初期出现了众多的面向对象方法，因此需要一种统一的和一致的可视化语言来表达这些方法，这种需求在相当长的一段时间里是非常明显的。

举例来说，在这段时间里，Grady Booch提出了Booch方法[9]，James Rumbaugh是通用电气公司对象建模技术（Object Modeling Technique, OMT）研发中心的首席开发人员[10]，1994年10月他加入到Rational公司时，两人与Rational公司的客户一道，开始统一他们的方法。1995年10月他们发布了UM（统一方法）的0.8版本。差不多在同一时间，Ivar Jacobson也加入到Rational公司。

这三个人一起工作，发布了UML（统一建模语言）0.9版本。该版本的发布带动了其他的方法学研究人员和众多的公司，包括IBM、HP和Microsoft等，他们为标准的进化做出了应有的贡献。1997年11月，在经历了标准化过程后，UML 1.1版本由对象管理组织（Object

Management Group, OMG）公布为标准。详细信息请参阅《UML用户手册》[11]和《UML参考手册》[12]。

Rational对象工厂过程的所有模型都使用UML。

Rational统一过程（RUP）

在这段时间里，Rational公司收购或者兼并了其他一些软件工具公司。每个公司都带来了在过程方面的一些专门技术，从而进一步发展了Rational对象工厂过程：

- Requisite公司为Rational带来了需求管理方面的经验。
- SQA 公司为配合其测试工具曾经开发了一个测试过程，给Rational在这方面增加了进一步的经验。
- Pure-Atria公司为Rational增加了配置管理方面的经验。
- Performance Awareness公司为Rational增加了性能测试和负载测试方面的经验。
- Vigortech公司为Rational带来了数据工程方面的专门技术。

该过程还基于[3]扩展了针对业务建模的新工作流，用于从业务过程提取为该业务服务的软件的需求。该过程也涉及到由用况（基于Objectory AB所做的工作）所驱动的用户界面设计。

到1998年中期，Rational对象工厂过程已经完全成熟，能够支持整个软件开发生命周期。这样做的结果是，它统一了广泛的来自不同来源（既包括上面所提到的三位作者提出的方法和思想，还包括许多Rational 和UML得以导出和发展的其他来源）的贡献。1998年6月，Rational公司发布了该产品的新版本——Rational统一过程（RUP）5.0版本[13]。这个Rational公司专有过程的很多内容现在以本书的形式第一次向公众公开。

名称的变化反映了统一是从多个角度进行的：开发方法的统一，使用统一建模语言以及很多方法论研究人员（不仅包括Rational公司中的研究人员，还包括几百个多年以来一直应用该过程的客户站点中的研究人员）的研究成果的统一。

致谢

本书的编写、出版和发行是许多人员辛勤劳动的结晶，在此列举他们的名字并向他们致以诚挚的谢意。

为本书的编写做出贡献的人员

Birgitte Lφnvig 准备了Interbank系统示例，并用所有模型来说明，这是本书的主要示例。

Partrik Jonsson从Rational对象工厂文献中选取材料，并按所建议的章节顺序对它们加以整理。他还帮助准备了示例。在此过程中，他还就如何更好地介绍统一过程发表了自己的一些意见。

Ware Myers从开始策划时就参与了本书的编写，是他将主要作者提供的最初手稿改写得更通俗易懂。

在审阅人员中，我们尤其要感谢Kurt Bittner、Cris Kobryn、和Earl Ecklund, Jr。此外，我们要特别感谢Walker Royce、Philippe Kruchten、Dean Leffingwell、Martin Griss、Maria Ericsson和Bruce Katz。审阅人员还包括Pete McBreen、Glenn Jones、Johan Galle、N. Venu Gopal、David Rine、Mary Loomis、Marie Lenzi、Janet Gardner以及一些没有留下姓名的审阅者，在此也向他们表示感谢。

Rational公司的Terry Quatrani对本书第1章到第5章的英文进行了修改完善。Karen Tongish对全书进行了编辑。在此，谨向两位表示感谢。

我们还要特别感谢Stefan Bylund，他审阅了全书的草稿并详细地提出了改进意见，其中很多都被我们采纳。他为本书质量的提高做出了巨大贡献。

多年来帮助我们的人员

我们还要感谢多年来一直帮助我们“建立正确的过程”，并通过不同方式支持我们工作的人们。特别要感谢下面一些人：Stefan Ahlquist、Ali Ali、Gunilla Andersson、Kjell S. Andersson、Sten-Erik Bergner、Dave Bernstein、Kurt Bitter、Per Bjork、Hans Brandtberg、Mark Broms、Stefan Bylund、Ann Carlbrand、Ingemar Carlsson、Margaret Chan、Magnus Christerson、Geoff Clemm、Catherine Connor、Hakan Dahl、Stephane Desjardins、Mike Devlin、Hakan Dyrhage、Susanne Dyrhage、Staffan Ehnebom、Christian Ehrenborg、Maria Ericsson、Gunnar M. Eriksson、Iain Gavin、Carlo Goti、Sam Guckenheimer、Bjorn Gullbrand、Sunny Gupta、Marten Gustafsson、Bjorn Gustafsson、Lars Hallmarken、David Hanslip、Per Hedfors、Barbara Hedlund、Jorgen Hellberg、Joachim Herzog、Kelli Houston、Agneta Jacobson、Sten Jacobson、Paer Jansson、Hakan Jansson、Christer Johansson、Ingemar Johnsson、Patrik Jonsson、Dan Jonsson、Bruce Katz、Kurt Katzeff、Kevin Kelly、Anthony Kesterton、Per Kilgren、Rudi Koster、Per Kroll、Ron Krubeck、Mikael Larsson、Bud Lawson、Dean Leffingwell、Rolf Leidhammar、Hakan Lidstrom、Lars Lindroos、Fredrik Lindstrom、Chris Littlejohns、Andrew Lyons、Jas Madhur、Bruce Malasky、Chris McClenaghan、Christian Meck、 Sue Mickel、Jorma Mobrin、Christer Nilsson、Rune Nilsson、Anders Nordin、Jan-Erik Nordin、 Roger Oberg、Benny Odenteg、Erik Ornulf、Gunnar Overgaard、Karin Palmkvist、Fabio Peruzzi、 Janne Pettersson、Gary Pollice、Tonya Prince、Leslee Probasco、Terry Quatrani、Anders Rockstrom、Walker Royce、Goran Schefte、Jeff Schuster、John Smith、John Smith、Kjell Sorme、 Ian Spence、Birgitta Spiridon、Fredrik Stromberg、Goran Sundelof、Per Sundquist、Per-Olof Thysselius、Mike Tudball、Karin Villers、Ctirad Vrana、Stefan Wallin、Roland Wester、Lars Wetterborg、Brian White、Lars Wiktorin、Charlotte Wranne、Jan Wunsche。

此外，多年来许多人为本书的主要作者提供了个人支持，为此向他们表示衷心的感谢。他们是：Dines Bjorner、Tore Bingefors、Dave Bulman、Larry Constantine、Goran Hemdal、

Bo Hedfors、Tom Love、Nils Lennmarker、Lars-Olof Noren、Dave Thomas以及Lars-Erik Thorelli。

最后尤其要感谢以下人员

Mike Devlin，Rational软件公司的总裁，感谢他始终坚信对象工厂过程可以作为一种产品，从而帮助全世界的软件开发人员开发软件，还要感谢他一直支持利用有效的软件过程作为软件工具开发的“驱动器”。

最后，我们还要感谢Philippe Kruchten，Rational统一过程（RUP）的主管，以及Rational过程组的全体成员，是他们将对象工厂中最好的东西与Rational公司的丰富经验及UML综合集成在一起，并最大程度地保留了各自的有价值的东西。此外，如果没有Philippe的个人承诺和坚忍不拔的意志，Rational过程组就不可能完成构造世界上所能见到的最好的软件过程这项任务，我们也不可能成功地完成本书的编写。

过程的突破

通过本书和相关的书籍、在线版本以及工具，我们可以得出这样的结论：软件开发过程已经成熟。统一过程从众多的来源获得灵感。现在，它已被广泛采用。它提供了一种使管理人员、开发人员和项目相关人员能从中获得对统一软件开发过程了解的公共媒体。

然而，我们仍有大量的工作要做。开发人员必须学习统一的工作方法，项目相关人员和管理人员必须支持他们。对于许多软件组织而言，这种突破仅仅是一种潜在的可能，而你可以将其变为现实。

Ivar Jacobson
Palo Alto，California
December 1998
ivar@rational.com

参考资料

[1] Peter F. Drucker, “The Discipline of Innovation,” *Harvard Business Review,* May-June, 1985; reprinted Nov.-Dec. 1998, pp. 149–157.

[2] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Övergaard, *Object-Oriented Software Engineering: A Use-Case Driven Approach,* Reading, MA: Addison-Wesley, 1992.

[3] Ivar Jacobson, Maria Ericsson, and Agneta Jacobson, *The Object Advantage: Business Process Reengineering with Object Technology,* Reading, MA:

Addison-Wesley, 1995.

[4] James E. Archer Jr. and Michael T. Devlin, "Rational's Experience Using Ada for Very Large Systems," *Proceedings of the First International Conference on Ada Programming Language Applications for the NASA Space Station,* June, 1986.

[5] Philippe B. Kruchten, "The 4 + 1 View Model of Architecture," *IEEE Software,* November 1995, pp. 42–50.

[6] Grady Booch, *Object Solutions: Managing the Object-Oriented Project,* Reading, MA: Addison-Wesley, 1996.

[7] Walker Royce, *Software Project Management: A Unified Framework,* Reading, MA: Addison-Wesley, 1998.

[8] Grady Booch, *Object-Oriented Analysis and Design with Applications,* Redwood City, CA: Benjamin/Cummings, 1994.

[9] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen, *Object-Oriented Modeling and Design,* Englewood Cliffs, NJ: Prentice Hall, 1991.

[10] Grady Booch, James Rumbaugh, and Ivar Jacobson, *The Unified Modeling Language User Guide,* Reading, MA: Addison-Wesley, 1998.

[11] James Rumbaugh, Ivar Jacobson, and Grady Booch, *The Unified Modeling Language Reference Manual,* Reading, MA: Addison-Wesley, 1998.

[12] Philippe Kruchten, *The Rational Unified Process: An Introduction,* Reading, MA: Addison-Wesley, 1998.

[13] Ivar Jacobson, *Concepts for Modeling Large Real Time Systems,* Chapter 2, Dissertation, Department of Computer Systems, The Royal Institute of Technology, Stockholm, Sept. 1985.

[14] Ivar Jacobson, "Object-Orientation as a Competitive Advantage," *American Programmer,* Oct. 1992.

[15] Ivar Jacobson, "A Large Commercial Success Story with Objects, Succeeding with Objects," *Object Magazine,* May 1996.

第一部分

统一软件开发过程

第一部分介绍统一软件开发过程的一些主要概念。

第1章对统一软件开发过程进行简单描述，强调它是用况驱动、以构架为中心、迭代和增量的。该过程使用统一建模语言（UML），该语言能够产生与长期用于其他技术学科的蓝图相似的草图。该过程使基于可重用构件（即具有明确定义接口的软件）来实现一个开发项目的大部分成为可能。

第2章介绍四个要素：人员、项目、产品和过程，并描述它们之间的关系，这些对了解整本书的内容很关键。为了理解这里所阐述的过程，有必要了解制品（*artifact*）、模型（*model*）、工作人员（*worker*）和工作流（*workflow*）等关键性概念。

第3章进一步详细讨论用况驱动开发的概念。用况是一种用来获得正确需求并用这些需求来驱动开发过程的方法。

第4章描述构架在统一过程中的作用。构架确定要做什么；构架还建立软件组织的主要层次，并侧重于确立系统的框架。

第5章强调采用迭代和增量的（*iterative and incremental*）方法进行软件开发的重要性。这意味着在实际操作中首先要处理系统的风险部分，尽早确定一个稳定的构架，接着在后续的迭代中再补充常规的部分，每一次迭代都得到一个增量版本，直至得到最后的版本。

第二部分对统一软件开发过程进行更为深入的探讨。分章描述需求、分析、设计、实现和测试等核心工作流。在统一软件开发过程四个阶段的不同迭代过程中，这些工作流将在下面的第三部分中用作实际的活动。

第三部分具体说明如何完成每个阶段的任务：首先在初始阶段获得一个业务案例，在细化阶段创建构架和制定方案，在构造阶段对构架进行扩充以得到一个可以发布的系统，在移交阶段要确保系统在用户的环境中能够正常运作。在这一部分需要重用核心工作流，并针对每个阶

段对其进行剪裁、组合，以使能够获得预期的结果。

然而，一个组织的根本目标不在于拥有好的软件，而是要运作其业务过程或嵌入式系统，使其能够根据市场需求，以合理的成本尽快生产出高质量的产品或提供高质量的服务。对于企业或政府来说，软件是在产品和服务方面大幅度降低成本和缩短生产时间的战略性武器。如果没有良好的组织过程，就不可能对风云变幻的市场作出快速的反应。在每周7天、每天24小时运营的全球经济活动中，如果没有软件，其中的大部分活动将无法运作。因此，一个良好的软件开发过程对于任何组织的成功都是一个关键因素。

第1章

统一过程的特点：用况驱动、以构架为中心、迭代和增量的

目前，软件的趋势是朝着更庞大、更复杂的系统发展。其中部分原因是计算机的性能逐年增强，用户的期望值随之增大。这种趋势也受到用于信息交流的因特网用途扩展的影响——从纯文本到格式化的文本，到图片，到图表，到多媒体。随着对一种软件产品及其在某些方面可能有所提高的后续版本的了解，我们对更为复杂的软件系统的渴望也在增加。我们希望软件能够更好地满足我们的需要，但这只会使软件更加复杂。简而言之，我们希望得到更多、更好的服务。

我们也希望能更快生产出软件。对市场而言，时间是另一个重要的驱动力。

然而，要实现上述目标具有相当的难度。我们采用的开发软件的过程不能满足我们开发功能强大、复杂的软件系统的需求。如今多数的开发人员仍然沿用25年前的方法来开发软件系统，这就是问题的根本所在。除非我们能够改进这种开发方法，否则将无法实现开发现今所需的复杂软件系统的目标。

软件问题可以归结为开发人员面临的将一个大型软件所包含的各个部分集成为一个整体的难度。软件开发团体需要一种受控的工作方式；需要一个过程来集成软件开发的方方面面；需要通用的方法或过程来完成如下工作：

- 指导一个群组活动的顺序。
- 布置每个开发人员和整个群组的任务。
- 确定开发何种制品。
- 提出监控和测量一个项目的产品和活动的准则。

具有明确定义和管理完善的过程是区分高效项目和不成功项目的关键。（参见2.4.4节“过程的价值”。）统一软件开发过程是30多年来经验的总结，是软件问题的一种解决方案。本章对整个统一过程进行概述，后续章节将会详细论述该过程所包含的每个元素。

1.1 统一过程概述

首先，统一过程是一个软件开发过程。软件开发过程（*process*）是一个将用户需求转化为软件系统所需要的活动的集合（如图1-1所示）。然而，统一过程不仅仅是一个简单的过程，而是一个通用的过程框架，可用于各种不同类型的软件系统、各种不同的应用领域、各种不同类型的组织、各种不同的功能级别以及各种不同的项目规模。

统一过程是基于构件的（*component-based*），即所构造的软件系统是由软件**构件**（附录A）通过明确定义的**接口**（附录A）相互连接所建造起来的。

统一过程使用统一建模语言（*Unified Modeling Language*, UML）来制定软件系统的所有蓝图。事实上，UML是整个统一过程的一个完整部分，它们是共同发展起来的。

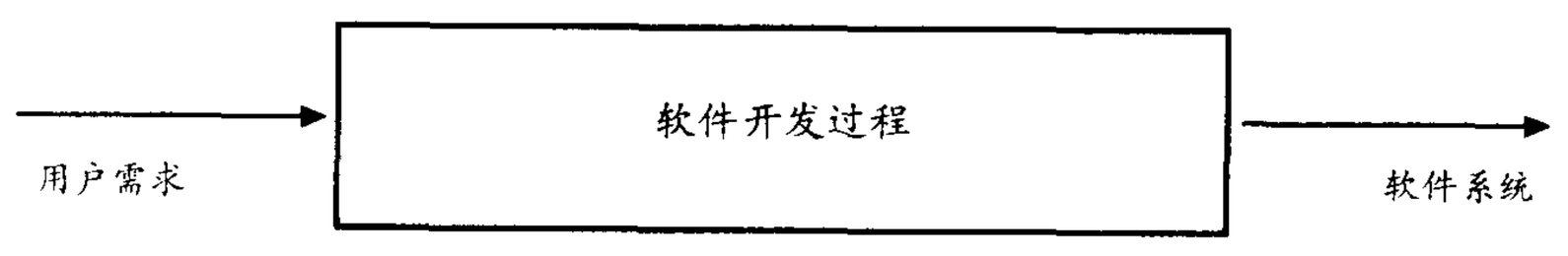


图1-1　软件开发过程

统一过程的突出特点可以由以下三个关键词来体现——用况驱动、以构架为中心、迭代和增量的。这是统一过程所特有的。

在后续的三节中，我们将详细介绍这三个关键词。然后在本章的其余部分对过程进行概述，包括过程的生命周期、阶段、版本、迭代、工作流和制品。这一章的总体目标是介绍最重要的概念和对过程进行全面的概括。读完本章后，你应该知道过程是什么，但没有必要完全理解。本书的其他章节将逐步交代过程的细节。第2章将介绍软件开发的四个元素：人员、项目、产品和过程。然后对三个主要概念分章进行介绍。上述内容为本书的第一部分。第二部分和第三部分是本书的核心，将详细介绍统一过程的各种工作流。

1.2　统一过程是用况驱动的

软件系统是为了服务于它的用户而出现的。因此，为了构造一个成功的软件系统，必须了解其预期的用户所希望和需要的是什么。

用户（*user*）这个术语所指的不仅仅是人，也可以是其他系统。从这个意义来说，术语用户代表了与所开发的系统进行交互的某个人或某件事（例如，被提及的系统之外的另一个系统）。一个人使用自动取款机（ATM）就是一个交互的例子：他（或她）插入取款卡，然后对取款机屏幕上出现的询问信息作出恰当的应答，便可以提取一定数额的现金。作为对用户的取款卡和应答的响应，系统执行一系列有序的**动作**（附录A），提供给用户一个有价值的结果，也就是取出现金。

这种交互就是一个**用况**（附录A；另见第3章）。用况是能够向用户提供有价值结果的系统中的一种功能。用况获取的是功能需求。所有的用况合在一起构成**用况模型**（附录B；另见2.3节），它描述了系统的全部功能。该模型代替了传统的系统功能说明。传统的系统功能说明回答了这样一个问题，即“系统应该做什么？”在问题中间增加三个词“*for each user*”（译为“为每个用户”），该问题变为“系统应该为每个用户做什么？”这样就可以刻画出用况方法的特点。这三个词包含了十分重要的涵义。它促使我们要从系统对用户的价值方面来考虑问题，而不仅仅限于提供强大的功能。然而，用况不只是一种确定系统需求的工具，它们还能驱动系统设计、实现和测试的进行，也就是说，用况可以驱动开发过程。基于用况模型，开发人员可以创建一

系列实现这些用况的设计和实现模型。开发人员可以审查每个后续建立的模型是否与用况模型一致。测试人员测试实现以确保实现模型的构件正确实现了用况。因此，用况不仅启动了开发过程，而且使其结合为一体。*用况驱动*（*use-case driven*）表明开发过程是沿着一个流——一系列从用况得到的工作流前进的。用况被确定，用况被设计，最后用况又成为测试人员构造测试用例的基础。

虽然用况确实可以驱动过程，但我们不能孤立地选择用况。它们与系统构架是协调发展的。也就是说，用况驱动系统构架，系统构架反过来影响用况的选择。因此，系统构架和用况会随着生命周期的延续而逐渐完善。

1.3 统一过程是以构架为中心的

软件构架的作用实际上与建筑物构架所起的作用类似。我们可以从结构、服务设施、供暖、供水、供电等各种不同的角度来审视一座建筑物。这使得施工人员在开始施工前就可以全面了解该建筑的全貌。与此类似，软件系统的构架也从不同的角度描述了即将构造的系统。

软件构架概念包含了系统中最重要的静态和动态特征。构架是根据企业的需要逐渐发展起来的，受到用户和其他项目相关人员需求的影响，并在用况中得到反映。但是，它也受到其他许多因素的影响，如软件应用平台（像计算机体系结构、操作系统、数据库管理系统、网络通信协议等）、是否有可重用的构造块（如**图形用户界面框架**，见附录C）、如何考虑实施问题、如何与遗留系统集成以及非功能性需求（如性能、可靠性）等。构架刻画了系统的整体设计，它去掉了细节部分，突出了系统的重要特征。因为“究竟什么是重要的”部分依赖于判断，而判断又来自于经验，所以，构架的价值也就依赖于执行该任务的人的素质。然而，过程可以帮助构架设计师确定正确的目标，如易理解性、适于将来变化的柔性以及可重用性。

用况和构架是如何相关的呢？每一种产品都具有功能和表现形式两个方面，只具备其中的一个方面是不够的。这两个方面必须恰当权衡才能得到成功的产品。这里功能与用况相对应，表现形式与构架相对应。用况和构架之间是相互影响的，这是一个“鸡与蛋”的问题。一方面，用况在实现时必须适合于构架；另一方面，构架必须预留空间以实现现在或将来所有需要的用况。事实上，构架和用况必须并行进化。

于是，构架设计师可以通过某种*表现形式*（*form*）来刻画一个系统。这种表现形式（即构架）必须设计得使系统能够进化，不仅要考虑系统的初始开发，而且要考虑将来的发展。为了能够找到这样一种表现形式，构架设计师必须从全面了解系统的主要功能（即主要用况）入手。这些主要用况的数量可能只占所有用况的5%～10%，但它们是十分重要的，这些用况构成了系统的核心功能。概括来说，构架设计师应该遵循以下步骤：

- 首先，从不是专门针对用况的那部分构架（如平台）开始，创建一个粗略的构架轮廓。尽管这部分构架与用况无关，但构架设计师在创建构架轮廓之前必须对用况有一个全面的了解。
- 其次，构架设计师需要着手处理已经确定的重要用况子集，这些用况代表了待开发系统的

主要功能。详细描述每个选定的用况，并通过**子系统**（附录A；另见3.4.4节）、**类**（附录A）和**构件**（附录A）来实现。

- 随着用况的描述趋于完善，构架的更多部分便会显现出来，从而也使更多的用况趋于完善。

继续这个过程，直到确信得到了一个稳定的构架为止。

1.4 统一过程是迭代和增量的过程

开发一个商用软件产品是一项艰巨的任务，可能会持续几个月甚至一年以上。将这项工作划分为较小的部分或袖珍项目（*mini-project*）是切实可行的。每个袖珍项目都是一次能够产生一个增量的迭代过程。迭代是指工作流中的步骤，而增量是指产品中增加的部分。为了获得最佳的效果，迭代过程必须是受控的（*controlled*）；也就是说，它们必须按照计划好的步骤有选择地执行。这就是称它们为袖珍项目的原因。

开发人员基于两个因素来确定在一次迭代过程中要实现的目标。首先，迭代过程要处理一组用况，这些用况合起来能够扩展所开发产品的可用性。其次，迭代过程要解决最突出的风险问题。后续的迭代过程建立在前一次迭代过程末期所开发的制品之上。它是一个袖珍项目，在每次迭代过程中，根据一些用况继续完成后续的开发工作（分析、设计、实现和测试），并以可执行代码的形式来实现这些用况。当然，一个增量不一定是对原有制品的增加，尤其在生命周期初期，开发人员可能是用更加详细和更加完善的设计来代替最初简单的设计。在以后的阶段，增量通常是对原有制品的增加。

在每次迭代过程中，开发人员标识并详细描述有关的用况，以选定的构架为向导来创建设计，用构件来实现设计，并验证这些构件是否满足用况。如果一次迭代达到了目标（通常如此），开发工作便可以进入下一次迭代。如果一次迭代未能达到预期的目标，开发人员必须重新审查前面的方案，并试用一种新的方法。

为了在开发过程中达到最佳的经济效益，项目组应设法选择实现项目目标所需要的迭代过程，并按照一定的逻辑顺序来排列这些迭代过程。一个成功的项目将会沿着这条笔直的路线进行下去，不会与开发人员最初规划好的路线有较大的偏离。当然，对于不可预见的问题，需要增加迭代次数或改变迭代顺序，开发过程将需要更多的工作量和更长的时间。将不可预见的问题减到最少是降低风险的目标之一。

受控迭代过程具有很多好处：

- 受控迭代过程将成本风险降低为获得一次增量所需要的费用。如果开发人员需要重复某次迭代过程，该组织损失的只是对这次出错的迭代过程的投入，而不是整个产品的价值。
- 受控迭代过程可以降低产品不能按计划投放市场的风险。通过在开发过程初期确定风险，处理这些问题所花费的时间出现在进度表的初期，此时，开发人员通常比开发末期要从容。在传统的方法中，在系统测试阶段才首次发现疑难问题，而解决这些问题所需要的时间常常超过项目进度表中预留的时间，因此几乎总是延期交付。

- 受控迭代过程可以加快整个项目的进展速度，这是因为有了清晰、简洁的目标，而不是在一个漫长且不断变化的进度下工作，开发人员的工作效率也会更高。
- 受控迭代过程确认一个经常忽略的事实，即用户需要和相应的需求无法预先清晰地说明。它们通常在后续的迭代过程中精确化。受控迭代的操作模式使项目开发更能适应项目需求的变化。

用况驱动、以构架为中心以及迭代和增量开发的概念是同等重要的。构架提供了一种结构来指导迭代过程中的工作，而用况则确定了目标并驱动每次迭代的工作。去掉三个主要概念中的任何一个，都会严重降低统一过程的价值。它就像一张三条腿的凳子，缺少一条腿凳子便会翻倒。

上面已经介绍了三个主要的概念，下面将对整个过程、它的生命周期、制品、工作流、阶段和迭代进行介绍。

1.5 统一过程的生命周期

统一过程是在重复一系列组成系统生命周期的循环，如图1-2所示。每次循环都以向用户提供一个产品**版本**（附录C；另见第5章）作为终结。

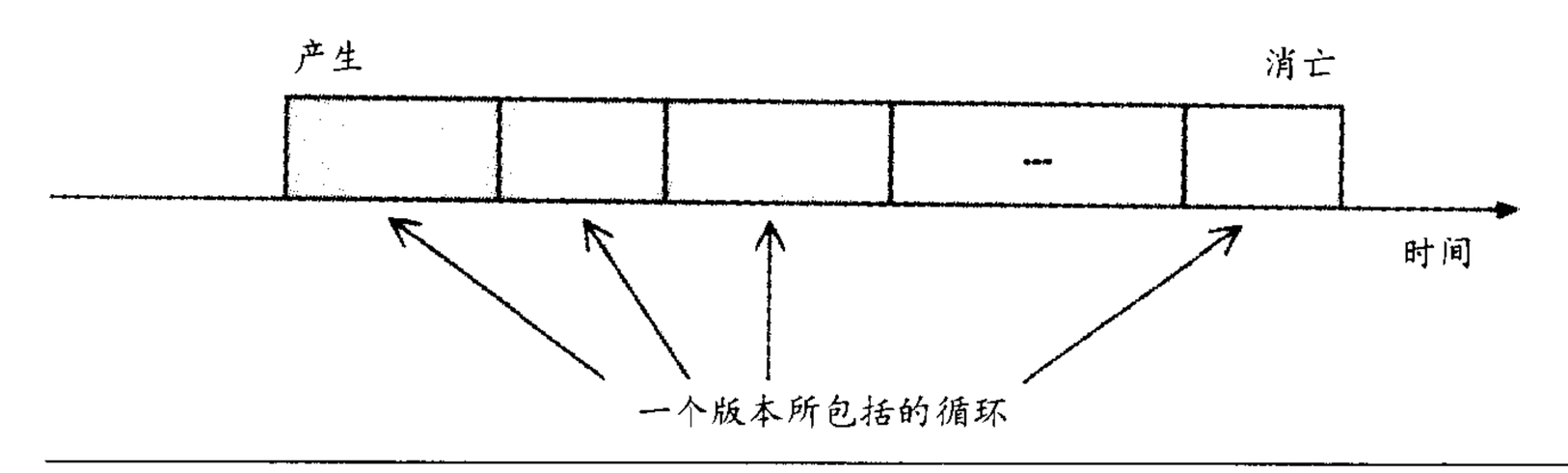


图1-2 由产生到消亡的循环组成过程的生命周期

每次循环都包括四个阶段：初始、细化、构造和移交。每个**阶段**（附录C）又进一步细分为上面提及的多次迭代过程，如图1-3所示。

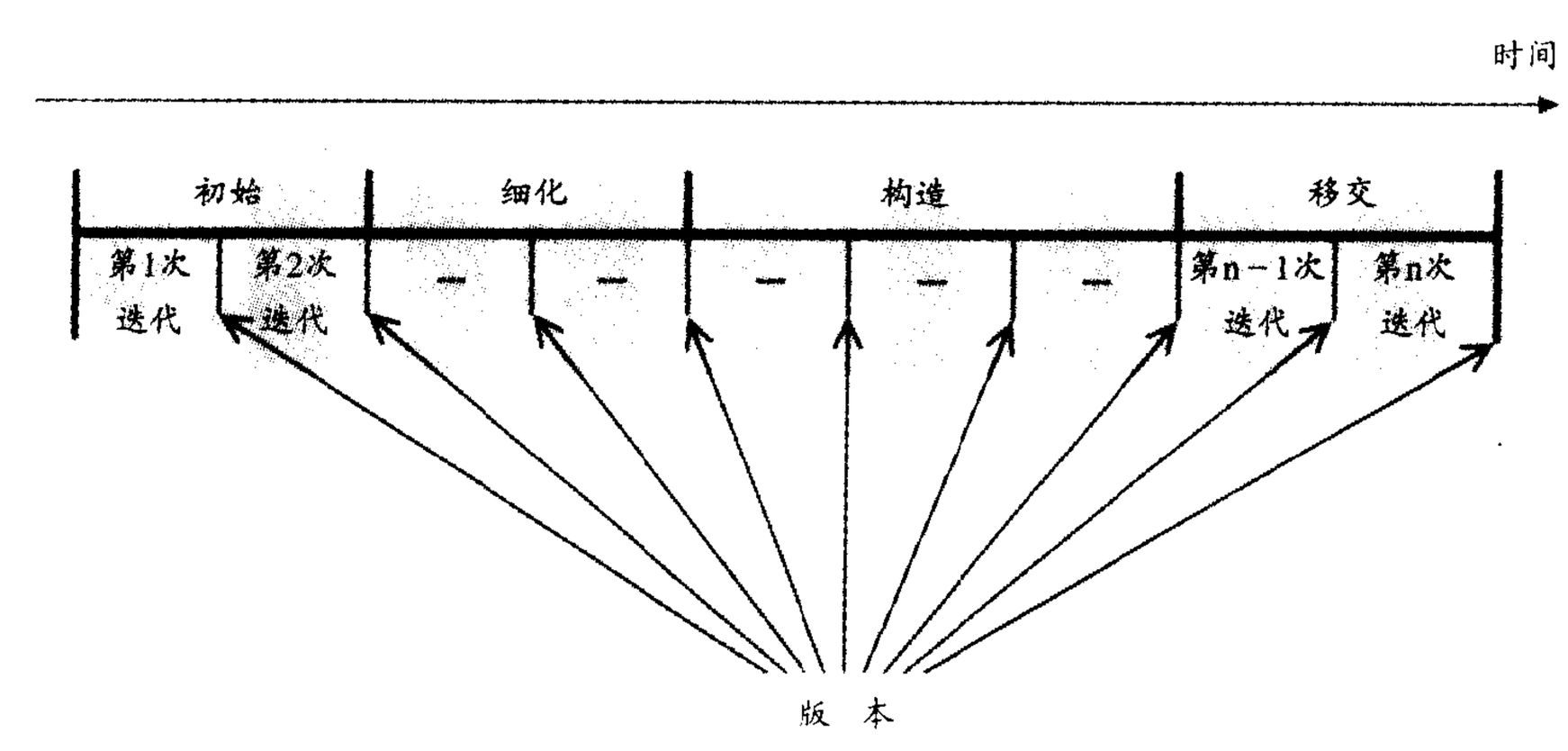


图1-3 一次循环所包含的阶段和迭代

1.5.1 产品

每次循环都产生系统的一个新的版本，每个版本都是一个准备交付的产品。它包括由能够编译和运行的构件所体现的源代码体、各种手册和相关的交付品。然而所完成的产品不仅要满足用户的各种需求，还需要满足所有产品相关人员的各种需求。软件产品应该不仅仅是可以运行的机器码。

所完成的产品包括需求、用况、非功能性需求和测试用例，还包括构架和可视化的模型（即用统一建模语言UML建立的模型化制品）。实际上，它包括本章中讨论的所有元素，因为正是这些元素使所有产品相关人员（客户、用户、分析人员、设计人员、实现人员、测试人员和管理人员）能够详细说明、设计、实现、测试和使用一个系统。而且，正是这些元素使这些产品相关人员能够使用和逐代更新这一系统。

尽管对于用户来说可执行的构件是最重要的制品，但是只有这些是不够的。这是因为环境在不断变化。操作系统、数据库系统和作为基础的计算机都在不断发展。此外，随着更好地理解了任务，需求本身可能会发生变化。事实上，需求变化对于软件开发来说是不可避免的。最后，开发人员必须着手进行一轮新的循环，管理人员必须为此提供经济支持。为了高效完成这一循环过程，开发人员需要该软件产品的所有表示（见图1-4）：

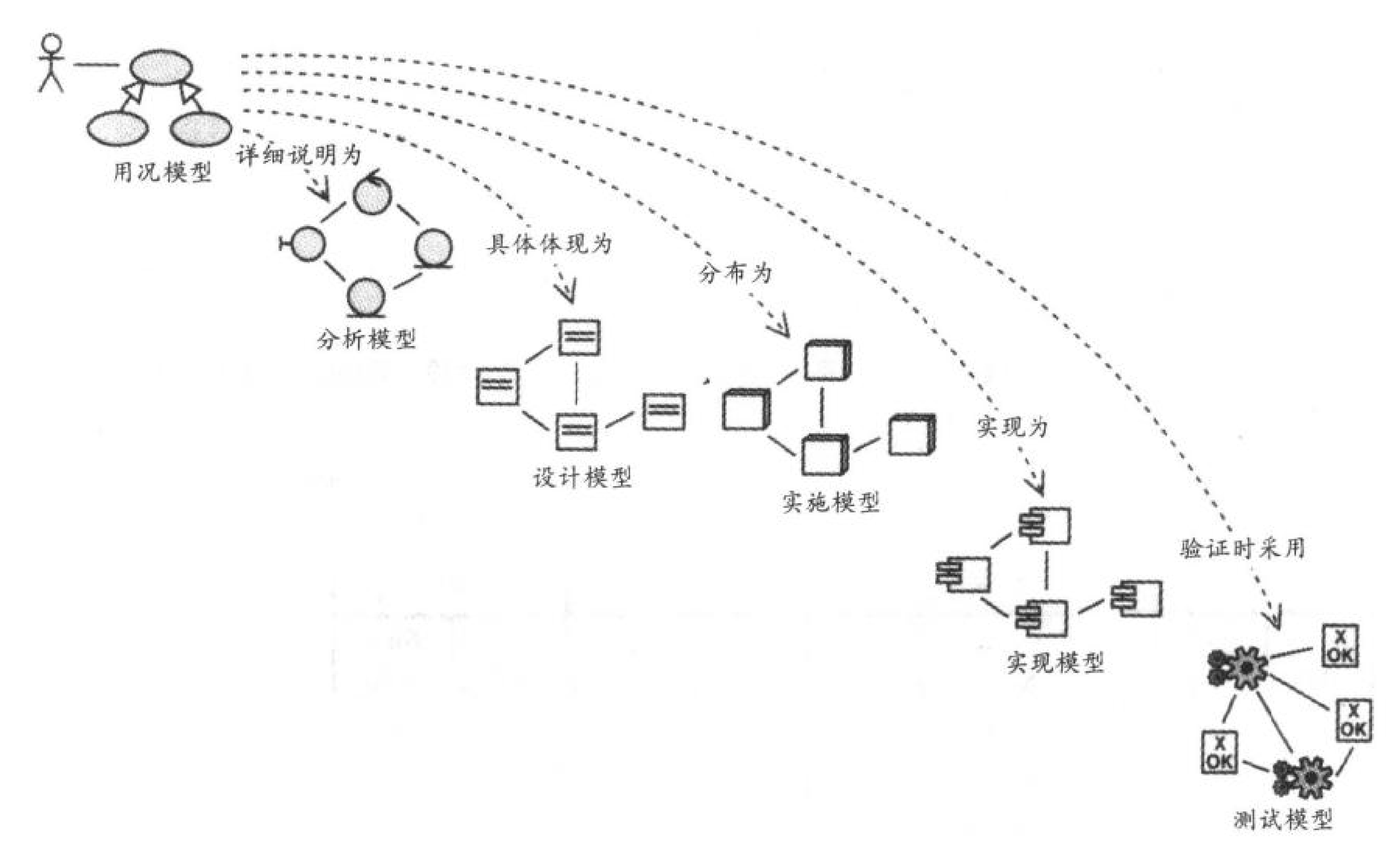


图1-4 统一过程的模型（很多模型之间具有依赖关系，图中虚线表示了用况模型和其他模型之间的依赖关系）

- 用况模型，包含所有用况及其与用户之间的关系。
- 分析模型，它有两方面的作用：更详细地提炼用况，将系统的行为初步分配给提供行为的

一组对象。

- 设计模型，将系统静态结构定义为子系统、类和接口，并定义由子系统、类和接口之间的**协作**（附录A；另见3.1节）所实现的用况。
- 实现模型，包括构件（表现为源代码）和类到构件的映射。
- 实施模型，定义计算机的物理节点和构件到这些节点的映射。
- 测试模型，描述用于验证用况的测试用例。
- 当然，还包括构架的表示。

系统可能还包括一个描述系统业务语境的领域模型或业务模型。

所有这些模型都是相关的，它们合起来表示整个系统。一个模型的元素到其他模型的链表明了模型中的元素前后间存在**跟踪**依赖关系（附录A；另见2.3.7节）。例如，（用况模型中的）用况可以跟踪到（设计模型中的）用况实现，再跟踪到（测试模型中的）测试用例。可跟踪性有助于系统的理解和修改。

1.5.2 一次循环所包含的阶段

每次循环都要经历一定的时间，这段时间又可以分为四个阶段，如图1-5所示。通过一组模型，所有与该产品相关的人员都可以看到这些阶段中要发生的事情。在每个阶段，管理人员或开发人员又可以将本阶段工作进一步细致地划分为多次迭代过程以及每次迭代所产生的增量。每个阶段都以一个**里程碑**（附录C；另见第5章）作为结束标志。通过获得一组可用的制品来定义每个里程碑；也就是说，使某些模型或某些文档达到预先规定的状态。

里程碑具有很多用途。最关键的用途是管理人员能够在进入下一阶段之前据此作出某些意义重大的决定。里程碑还能够使管理层和开发人员在开发工作通过这四个关键点时监控工作的进展情况。最后，通过跟踪在每个阶段所投入的时间和其他开销，我们可以得到大量的数据。这些数据对估计其他项目的时间和人员需求，对设计项目过程中的人员需求，以及控制项目的进展等都是很有用的。

图1-5左侧列出了开发过程的工作流——需求、分析、设计、实现和测试。曲线图近似描述了（不应过分按字面意义理解）工作流在每个阶段中完成的情况。每个阶段通常又进一步细分为多次迭代过程或袖珍项目。一次典型的迭代过程经历全部五种工作流，图1-5中细化阶段显示了这一情况。

在*初始阶段*（*inception phase*），将一个好的想法发展为最终产品的一个构想，提出了该产品的业务实例。本质上，该阶段回答下面的问题：

- 系统向它的每个主要用户提供的基本功能是什么？
- 该系统的构架看起来是什么样子？
- 开发该产品的计划如何？开销多大？

包含最关键用况的简化用况模型可以回答第一个问题。在这个阶段，构架是试验性的，通常只是一个包括主要子系统的大致的轮廓。在这个阶段，要确定最主要的风险及其优先次序，要对细化阶段进行详细规划，并对整个项目进行粗略估算。

在*细化阶段*（*elaboration phase*），详细说明该产品的绝大多数用况，并设计出系统的构架。系统构架和系统本身的关系是至关重要的。可以用一种简单的方法来说明这种关系：构架就像

是由皮肤所覆盖的骨架，在皮肤和骨骼之间只有很少可令骨架产生基本动作的肌肉（即软件）。系统则是包括了骨架、皮肤和肌肉的整个躯体。

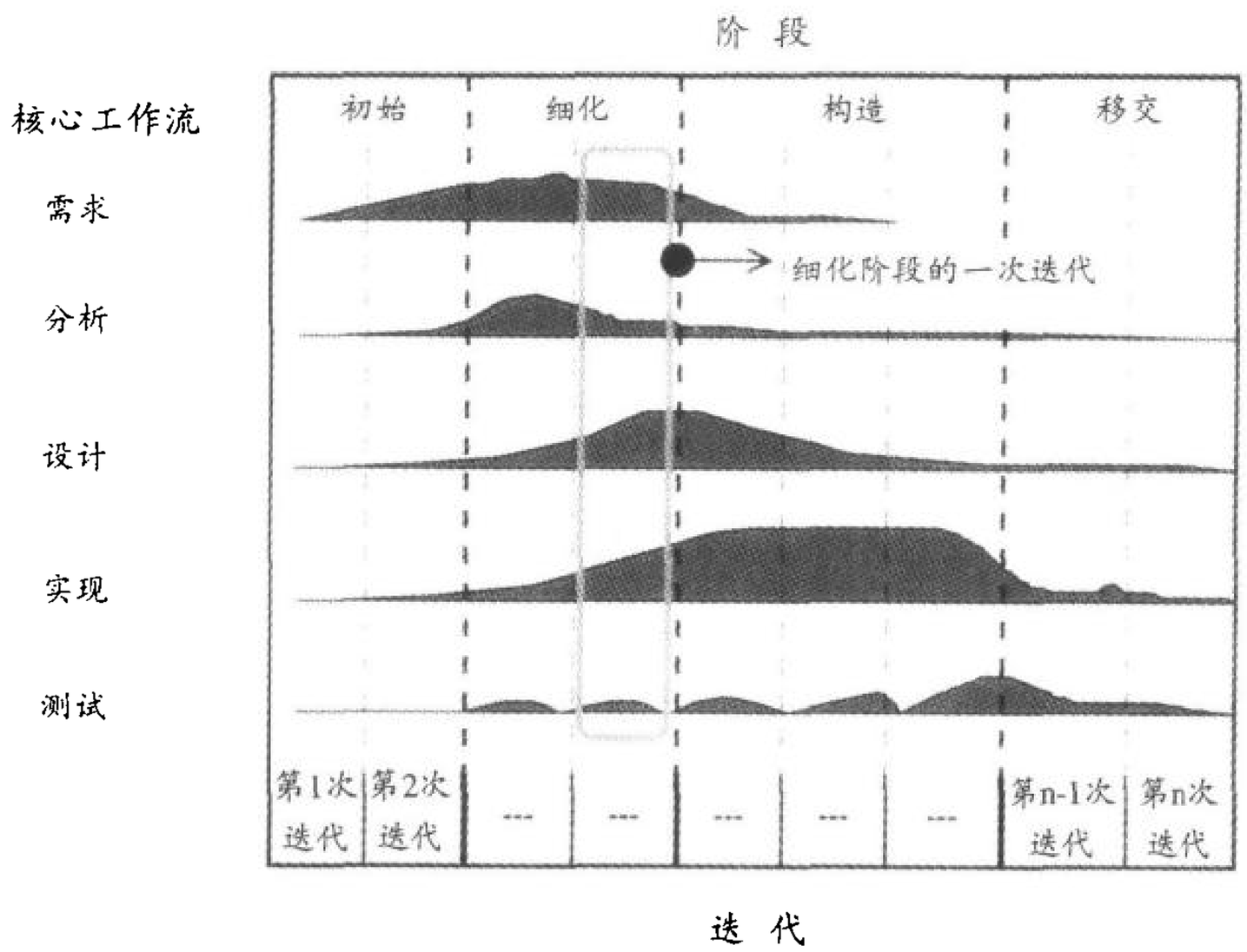


图1-5 在四个阶段（初始、细化、构造和移交）中发生的五种工作流——需求、分析、设计、实现和测试

所以，构架表示为系统中所有模型的不同视图，合起来表示整个系统。这意味着构架包括了用况模型、分析模型、设计模型、实现模型和实施模型的视图。实现模型的视图包含了一些构件，以证明该构架是可以运行的。在开发的这个阶段所确定的最关键用况都要在这个阶段具体化。该阶段的结果是构架**基线**（附录C；另见4.4节）。

在细化阶段末期，项目经理要规划完成项目的活动，估算完成项目所需的资源。这里的关键问题是：用况、构架和计划是否足够稳定可靠，风险是否得到充分控制，以便能够按照合同的规定完成整个开发任务。

在构造阶段（*construction phase*），将构造出最终产品——为骨架（构架）增加肌肉（完成的软件）。在这个阶段，构架基线逐渐发展成为完善的系统。在初始阶段形成的构想进化成为一个准备移交给用户团体的产品。在开发的这个阶段，将消耗所需的大部分资源。虽然因为开发人员可能发现更好地构造系统的方法，所以他们可能会向构架设计师建议对构架进行细微变动，但系统的构架是稳定可靠的。在这个阶段的末期，产品将包括管理层和客户对发布的版本达成共识的所有用况。但是，产品不可能是完全没有缺陷的。很多缺陷将在移交阶段发现和改正。在里程碑处要回答的问题是，早期交付给某些客户的产品是否完全满足了用户的需求？

移交阶段（*transition phase*）包括了产品进入β版后的整个阶段。在β版期间，少数有经验的用户试用该产品并报告产品的缺陷和不足。开发人员则改正所报告的问题，并将部分改进建议嵌入到用于更大用户范围的通用版本。移交阶段包括诸如制作、用户培训、提供在线支持以及

改正交付之后发现的缺陷等活动。维护部门通常将这些缺陷分为两类：对检验最近一个δ版本的操作产生重要影响的缺陷和可以在下一个正式版本中改正的缺陷。

1.6 一个综合的过程

统一过程是基于构件的。它采用新的可视化建模标准，即统一建模语言（UML），它依赖三个关键概念——用况、构架以及迭代和增量开发。为了使这些概念能够发挥作用，需要一个包括多个方面的过程，该过程要考虑到生命周期、阶段、工作流、风险缓解、质量控制、项目管理和配置管理。统一过程建立了一个集成所有这些方面的框架。这个框架就像是一把伞，在它之下，工具厂商和开发人员可以生产工具来支持过程自动化，支持专门的工作流，构造所有不同的模型，完成跨越生命周期和所有模型的集成工作。

本书的目的在于从工程角度、从三个主要的概念（用况、构架以及迭代和增量开发）以及从基于构件的设计和UML的运用等方面来描述统一过程。本书将介绍四个阶段和不同的工作流，但不太具体涉及管理问题，如项目规划、资源规划、风险缓解、配置管理、捕获度量数据和质量控制等问题，此外本书只是简要地讨论过程的自动化问题。

第2章

软件开发的四个要素：人员、项目、产品和过程

软件**项目**的最终结果是一种产品（附录C），产品在其开发期间是由许多不同类型的人员建造的。指导参与项目人员工作的是软件开发过程，它是一种模板，阐明了完成项目所需的各个步骤。过程一般可以通过使用一种或一组工具自动完成。参见图2-1。

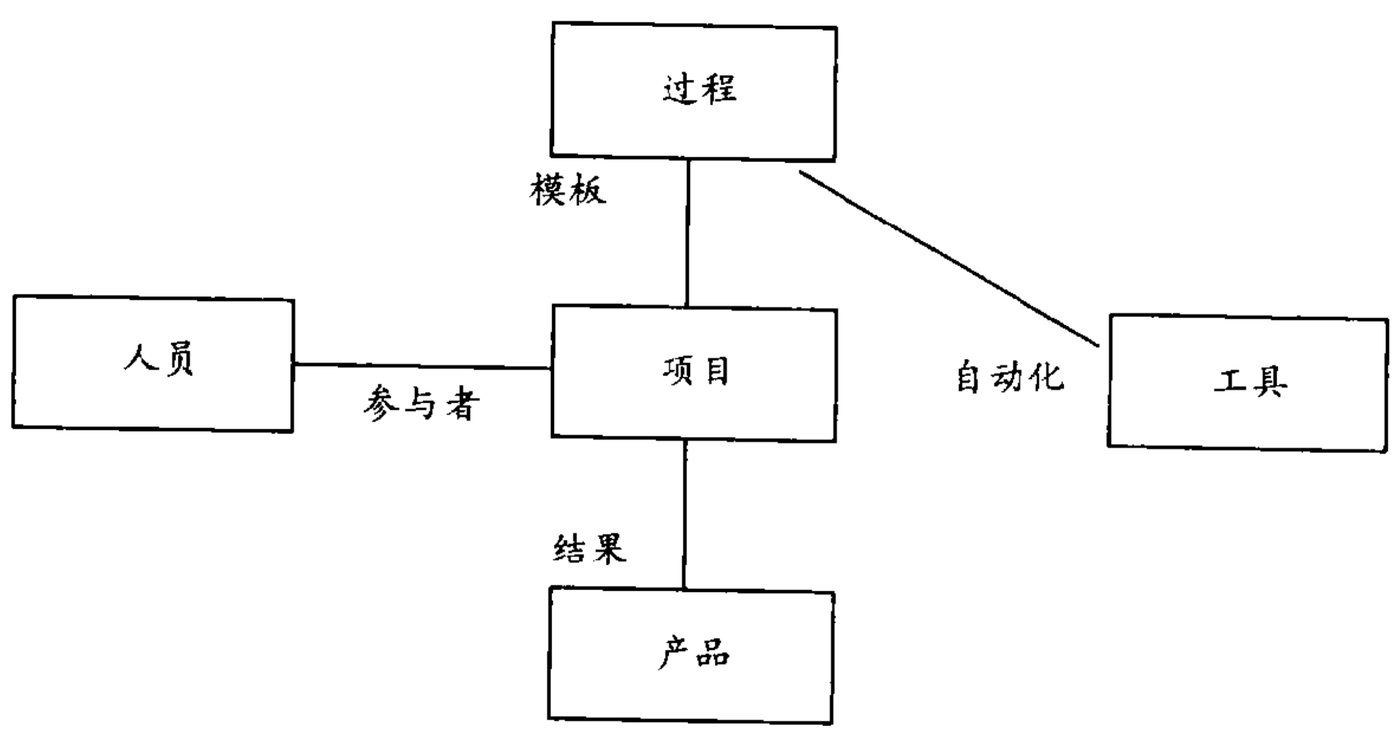


图2-1 软件开发的四个要素

本书通篇将使用人员、项目、产品、**过程**（附录C）和工具等五个术语，这些术语的定义如下：

- 人员（*people*）：构架设计师、开发人员、测试人员和相应的管理人员，此外还有用户、客户和其他项目相关人员都是软件项目的主要推动者。此处人员是指实际的人，而不是后面将要介绍的工作人员（*worker*）的抽象概念。
- 项目（*project*）：管理软件开发的组织元素。项目的成果是可以发布的产品。
- 产品（*product*）：在项目的生命周期内创建的制品，如**模型**（附录A）、源代码、可执行代码和文档。
- 过程（*process*）：软件工程过程是指将用户需求转化为产品所需要的完整活动集合的定义。一个过程是一种用来创建项目的模板。
- 工具（*tool*）：用于使过程中定义的活动自动化的软件。

2.1 人员至关重要

人员将参与软件产品的开发，并贯穿其整个生命周期，他们或对产品提供资金，或进行规

划、开发、管理、测试、使用，或从中受益。因此，指导产品开发的过程必须是面向人员的，也就是说，要便于人们运用。

2.1.1 开发过程影响人员

组织和管理软件项目的方式会极大地影响参与项目的人员。诸如项目可行性、风险管理、开发组结构、项目进度和项目易理解性等概念均起着重要作用：

- 项目可行性（*project feasibility*）：大多数人都不愿意接手确信为不可行的项目——没有人希望随着这条船沉没。我们在第1章中看到，迭代的开发方法允许在早期对项目的可行性进行评估，不可行的项目可以在较早的阶段终止，从而缓解精神压力。
- 风险管理（*risk management*）：在相似的情况下，当人们觉得没有对风险进行透彻分析和采取缓解措施时，会变得心神不宁。在早期阶段探查主要的风险可以缓解该问题。
- 开发组结构（*team structure*）：人员在6～8人的项目组里工作效率最高。由一个小型项目组来评估一种具体的风险、开发一个子系统（附录A）或完成一次迭代的过程，为能够产生有意义结果提供了机会。在子系统和构件（附录A；另见第10章）之间具有明确定义接口（附录A；另见第9章）的良好构架，使这种划分工作的方法成为可能。
- 项目进度表（*project schedule*）：当人们相信项目进度表不实际时，士气会十分低落——因为人们不愿意去做明知无论如何努力都不能获得预期结果的工作。在初始阶段和细化阶段所用的技术，使开发人员对项目的最终结果应该是什么，即发布的产品应该完成什么功能，能产生一个好的想法。因为对产品应该完成什么功能有了一种良好的感觉，所以对于为了实现目标所需要的工作量和时间，能够提出一个切实可行的项目计划，从而可以缓解具有“我们永远无法完成”想法的人员的问题。
- 项目易理解性（*project understandability*）：人员希望知道他们要做什么，希望了解整个项目规划。构架描述为参与项目的每个人提供了一个概貌。
- 成就感（*sense of accomplishment*）：在迭代的生命周期中，人员会得到频繁的反馈，并使该迭代过程妥善结束，从而加快了工作的节拍。快速的工作步伐和频繁的完成工作增加了人们的成就感。

2.1.2 角色会发生变化

因为软件开发的关键活动是由人员来完成的，因此需要有工具和统一建模语言（目前在UML中已经做到）（附录C）来支持统一开发过程，以便使人员效率更高。这样一个过程可使开发人员在兼顾投放市场时间、质量和成本的基础上构造出更好的软件；能够帮助他们详细说明更好满足用户需要的需求；能帮助他们选择一个构架，使所建立的系统具有投资效益且可及时修改。一个好的软件过程还有助于构造更复杂的系统。我们在第1章曾经提到，随着现实世界变得越来越复杂，客户也会随之需要更为复杂的软件系统。业务过程和相应的软件将会具有更长的生命周期。因为在整个生命周期之内，现实世界会持续变化，所以软件系统也必须设计成适

于在一段较长的时间内持续发展。

为了理解和支持这些更为复杂的业务过程，并在软件中付诸实现，开发人员将会与很多其他类型的开发人员一起工作。为了使越来越大的群组工作更有成效，亟需一个过程来提供指导。这样的指导会使开发人员“更加聪明地工作”，即限定开发人员努力为客户增加有价值的东西。这种指导的第一步是建立用况模型，使我们将精力集中在用户需要做什么上。第二步是建立构架，以允许系统随着时间的推移而持续进化。第三步是尽可能多地购买或重用软件。只要按照一致的方式将可重用的构件和新开发的元素集成起来，就可最终建成系统。

在未来的几年内，大多数软件人员的工作与他们所支持的任务会更加密切，并且由于有了自动化过程和可重用构件，因此他们能开发出更为复杂的软件。在可以预见的将来，人员对于软件开发将至关重要。最终要靠合适的人员才能使我们成功。这个问题归结为使人员更为高效，允许他们做只有人才能做的工作——具有创造性、发现新的机会、运用判断能力、与客户或用户进行交流以及了解快速变化的世界。

2.1.3　将“资源”转化为“工作人员”

在软件开发组织中，人员担任着很多不同的职务。为了使人员胜任这些职务，需要对他们进行教育和有针对性的培训，然后精心分派职务并使他们在监督和指导下工作。当软件开发组织将某个人从潜在的“资源”转化为“工作人员”的具体职务时，它面临着一个很实际的任务。

我们选择**工作人员**（附录C）一词代表可能赋予人员的和他们所接受的职务[4]。工作人员类型（*worker type*）是指一个人在软件开发中所充当的角色，例如用况描述人员、构架设计师、构件工程师和集成测试人员。我们不使用术语角色（而是使用术语工作人员）主要出于两个原因：第一，角色在UML中具有精确的和不同的意义，而工作人员的概念十分具体；第二，我们需要把工作人员个体看作单个个人所担任的职务，以此来思考问题。我们还要使用术语角色来讨论工作人员的角色。在不同的工作流中，一个工作人员可以充当多种角色。例如，构件工程师这个工作人员可能参与多个工作流，在每个工作流中充当一个具体的角色。

每个工作人员（如工作人员实例）负责一个完整的活动集合，如子系统设计中所涉及的各种活动。为了高效地工作，工作人员需要了解执行这些活动所需要的信息，需要理解他们所担负的角色与其他工作人员所担负的角色之间的关系。同时，如果他们要完成自己的工作，所用的工具必须恰当。这些工具不仅能帮助工作人员执行他们自己的活动，而且能屏蔽与他们无关的信息。为了实现这些目标，统一过程正式描述了在过程中人员可以承担的职务——即工作人员。

图2-2描述了在一个项目中个体可能是不同的工作人员。

一个工作人员也可以由一组协同工作的个体来充当。例如，构架设计师可以由一个设计组来充当。

在开发软件的过程中，每个工作人员都有一些具体的职责并执行一个活动集合。

在一个项目中，当把资源分配给项目中的工作人员时，项目经理需要确定每个人的能力，

并使他们与这些工作人员岗位所需要的能力相匹配。这不是一件容易的工作，尤其是在第一次使用统一过程时。资源（即真实的人）的技能必须与该项目所需要的各种工作人员岗位所规定的能力相匹配。一些工作人员岗位所需要的能力可以通过培训获得，但另外一些工作人员岗位所需要的能力只有通过经验才能得到。例如，用况描述人员所需要的技能可以通过培训学到，但是构架设计师所需要的技能则需要通过经验才能获得。

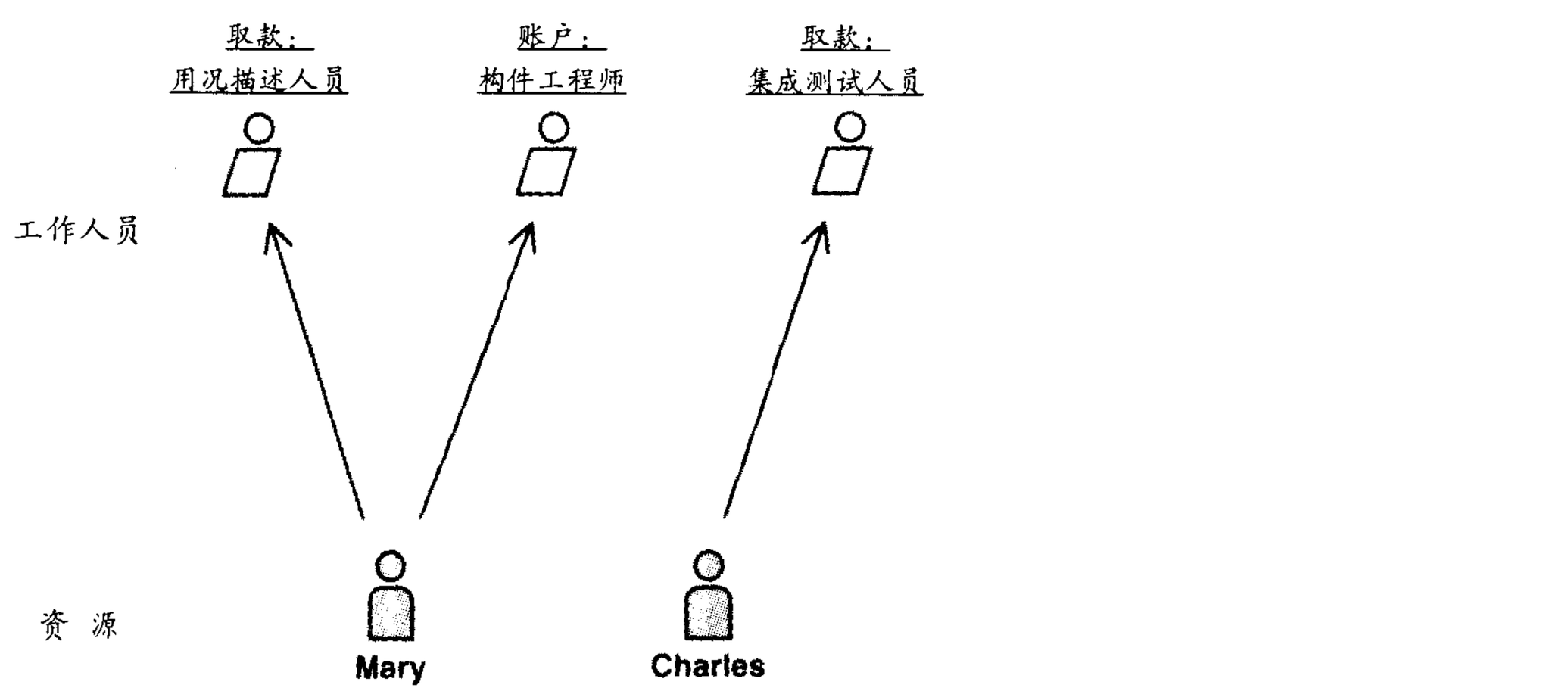


图2-2 工作人员和实现他们的资源

在项目的生命周期内，一个人可以作为多种不同的工作人员。例如，Mary 开始作为一个用况描述人员，后来作为一个构件工程师。

当分配资源时，项目经理应该尽量减少制品在资源间的交接次数，以便使过程的流程尽可能是无缝的。例如，“取款”用况中的用况描述人员（附录A，另见第7章）将会获得许多有关“账户”类（附录A）职责的知识，所以他或她将成为“账户”类构件工程师逻辑上的候选者。另一种选择是培养一位新人来承担这项工作，这样做是可行的，但是由于存在信息损失或误解等风险，因此，这样做不是最有效的。

2.2 项目创造产品

一个开发项目将会得到一种产品的新版本。生命周期中的第一个项目（即第一个开发周期的项目，有时称为“非先验性项目”）开发和发布最初的系统或产品。后续的项目周期将在很多版本之上延续系统的生命周期。参见参考资料[9]和[10]对项目管理所作的更为详细的介绍。

在项目的整个生命周期中，项目组在项目实施的过程中必须综合考虑变化、迭代和组织模式这三方面的因素：

- 一系列变化（*a sequence of change*）：系统开发项目产生产品，但沿着这一路径的过程是一系列的变化。在工作人员经历各个阶段和各次迭代时，必须牢记项目生命周期中的这种

实际情况。每个周期、每个阶段和每次迭代都将系统从一种情况变化为另一种情况。第一个开发周期是个特例，它使系统从无变为有。每次循环产生一个*系统版本*（*release*），并且在一系列循环之外，变化还将继续多代（*generations*）。

- *一系列迭代*（*a series of iterations*）：在一个周期的每个阶段，工作人员通过一系列的迭代完成该阶段的活动。每次迭代都会实现一些有关的用况或降低某些风险。在一次迭代中，开发人员将处理一系列的工作流：需求、设计、实现和测试。因为每次迭代都会经历所有这些工作流，所以，可以把一次迭代看作是一个*袖珍项目*（*miniproject*）。
- *组织模式*（*an organizational pattern*）：一个项目包括一组人员，他们被指派在业务约束（即时间、成本和质量）的范围内实现某种结果。人员作为不同的工作人员进行工作。“过程”的意义在于提供一种模式，在该模式内人员作为工作人员来完成一个项目。这种模式或模板指明了项目所需要的工作人员的类型以及项目将产生的制品。该过程还提供了一些指南、启发性知识以及文档实践，以帮助承担任务的人员完成他们的工作。

2.3 产品不仅仅是代码

在统一过程的语境中，所开发的产品是一个软件系统。这里的术语*产品*（*product*）不是单指所交付的代码，而是指整个系统。

2.3.1 什么是软件系统？

一个软件系统是机器代码、是可执行代码吗？的确如此，但什么是机器代码呢？它是一种描述！一种可以被计算机“阅读”和“理解”的二进制形式的描述。

一个软件系统是源代码吗？或是说，它是一种由程序员编写并能够被编译器阅读和理解的描述吗？是的，可以这样回答。

我们可以继续以这种方式，从子系统、类、**交互图**（附录A）、**状态图**（附录A）以及其他制品等方面来询问关于软件系统设计的类似问题。它们是系统吗？是的，它们都是系统的一部分。那么有关需求、测试、销售、生产、安装和操作又是什么呢？它们是系统吗？是的，它们也是系统的组成部分。

系统是以机器形式或人们可读形式给机器、工作人员和其他项目相关人员提供表示的所有制品。这里机器是指工具、编译器或目标计算机。工作人员包括经理、构架设计师、开发人员、测试人员、市场营销人员、行政管理人员以及其他人员等。项目相关人员包括投资者、用户、销售人员、项目经理、生产线经理、生产人员以及代理等。在本书中，除非明确地区分每一类人员，否则采用术语*工作人员*来统一表示这三类人员（即机器、工作人员和项目相关人员）。

2.3.2 制品

制品（附录C）是一个通用的术语，表示在系统开发过程中由工作人员创建、生产、修改或使用的各种信息。制品的一些例子如UML图和与之相关的文本、用户界面草图和**原型**（附录

C；另见第7章和第13章）、构件、测试计划（参见第11章）和测试规程（参见第11章）。

主要存在两类制品：工程制品和管理制品。本书集中讨论在过程的各个阶段（如需求、分析、设计、实现和测试）所创建的工程制品。

然而，软件开发也需要管理制品。有些管理制品生命周期较短——它们只能在项目的生命周期内存在。属于这类制品的有业务案例和开发计划（包括发布和迭代计划）、一个用来将个别人员指派为工作人员（如项目中不同的职位或职责）并安排工作人员的各种活动的计划。这些制品是用文本或图来描述的，由项目开发组用各种可视化手段向项目相关人员详细描述他们的承诺。管理制品还包括开发环境说明书——开发人员所需要的过程自动化软件和硬件平台以及存放工程制品所需要的资料库。

2.3.3 系统包含一组模型

用在统一过程中的最引人关注的制品是模型。每个工作人员都需要有一个独特的系统视角（如图2-3所示）。当设计统一过程时，要标识出所有的工作人员以及他们可能需要的各种视角。所收集的全部工作人员的视角构成一个更大的单位，也就是模型；按照这种方法，一个工作人员可以从模型中得到任何特定的视角。

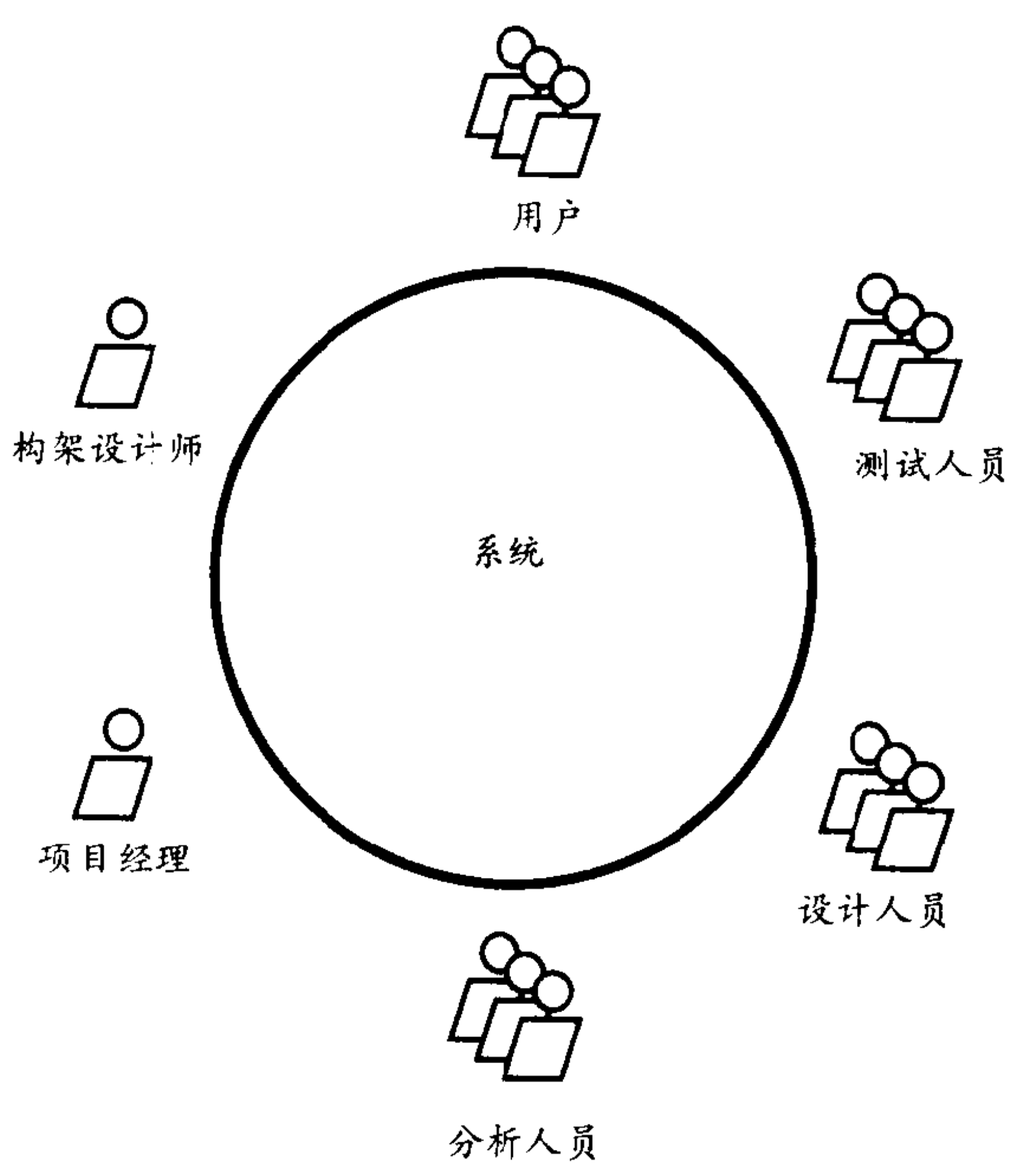


图2-3 参与软件开发的工作人员（其中一些是单一类型的工作人员；另一些是多种类型和多个对象的工作人员）

因此，构造系统就是一个构造模型的过程，即采用不同的模型来描述系统所有不同视角的过程。为系统选择模型是开发组所要做的最重要的决定之一。

在第1章中曾介绍过统一开发过程的主要模型（见图2-4）。

图2-4 统一过程中的主要模型集合[⊖]

统一过程给出了经过仔细选择的模型集合，并用它来启动过程。模型集合向所有的工作人员（包括客户、用户和项目经理）阐明该系统。模型的选用应能满足工作人员对信息的需要。

2.3.4 什么是模型？

模型是对系统的一种抽象，从某个视点、在某种抽象层次上详细说明被建模的系统[1]。例如，一种视点是系统的规格说明视图或设计视图。

模型是对构架设计师和开发人员构造的系统的抽象。例如，对功能需求建模的工作人员认为用户处于系统之外，而认为用况处于系统之内。他们不关心系统从内部看起来像什么，而只关心对于用户来说它能做什么。构造设计模型的工作人员只考虑结构元素，如子系统和类；他们考虑这些元素如何在给定的语境中工作，又如何协同工作来为用况提供功能。他们明白这些抽象的东西如何工作，并在他们的头脑中具有一种特定的解释。

2.3.5 每个模型是系统自包含的视图

模型是系统的语义闭合的抽象。它是一个自包含的视图，即用户不需要其他信息（即从其他模型）就可解释该系统。

自包含的概念意味着当触发一个用该模型描述的事件时，开发人员希望在系统中产生的效果只能有一种解释。除了从系统的角度考虑之外，模型还要描述系统与周围环境间的交互。这样，除了被模型化的系统之外，模型还要包含描述其与环境相关部分的元素，即它的**参与者**（附录A；另见第7章）。

大多数工程模型是由精心选择的UML子集定义的。例如，用况模型由用况和参与者所组成。这基本上是所有观察者都需要了解的。设计模型描述了系统所包含的子系统和类以及它们如何进行交互来实现用况。对于参与者发出的一系列外部激励系统应该作出什么反应，用况模型和设计模型给出了两种不同的、但相互一致的解释。它们之所以不同，是因为它们是为完成不同任务的不同工作人员设计的。用况模型是系统的外部视图；而设计模型是系统的内部视图。用况模型捕获系统的应用，而设计模型表示系统的构造。

2.3.6 模型的内部

模型总是用来标识被模型化的系统。那么，这种系统元素就是其他元素的容器。**高层子系**

⊖ 在UML术语中，这些“包”表示统一过程中的《业务实体》（或《工作单元》），而不是对一个特定系统建模的模型元素。另见7.1节中附加说明段中的解释。

统（附录B）表示所构造的系统。在用况模型中，系统包含用况；在设计模型中，系统包含子系统、接口和类。系统还包括用来确定所有涉及的子系统和类之间的协作（附录A），可能还包括其他元素，如状态图或交互图。在设计模型中，每个子系统本身可能就是一个具有相似结构的容器，这意味着在这种模型中存在着元素之间的层次关系。

2.3.7 模型间的关系

一个系统包含了不同模型中模型元素之间的所有**关系**（附录A）和约束[1]。因此，一个系统不仅是其模型的集合，而且也是模型间关系的集合。

例如，用况模型中的每个用况与分析模型中的协作具有一定关系（反之亦然）。这样一种关系在UML中称为跟踪依赖，或简称为跟踪（附录A）。如图2-5所示，图中跟踪用单向箭头表示。

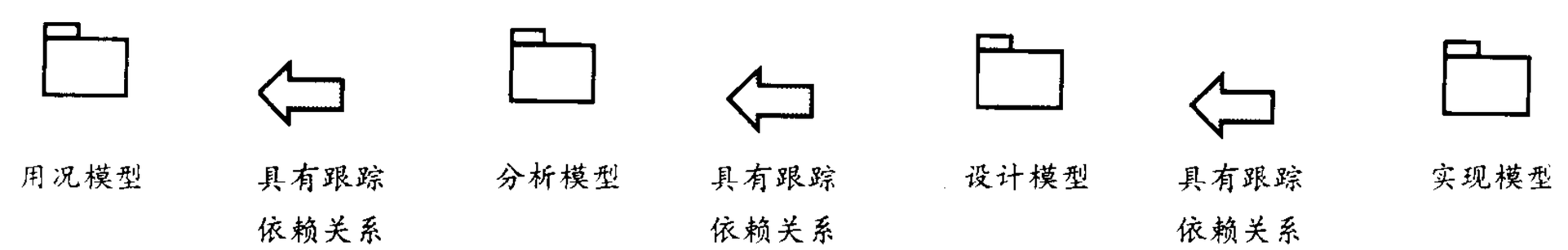


图2-5 模型通过跟踪关系紧密连接

在设计模型中的协作和分析模型中的协作之间，以及在实现模型中的构件和设计模型中的子系统之间，也都存在跟踪依赖关系。于是，可以用跟踪关系把一个模型中的元素连接到另一个模型中的元素上。

两个模型元素间存在连接关系的事实不会因为它们在模型内部所执行的动作而变化。不同模型中的元素间的跟踪关系增加了非语义信息，有助于我们了解相关的模型；它们只是使模型相互连接。在软件开发中，由于易理解性和变更的传播等原因，跟踪能力是非常重要的。

2.4 过程指导项目

过程（*process*）是一个使用很频繁的术语，它在不同的语境（如业务过程、开发过程、软件过程）中具有不同的含义。在统一过程的语境中，过程是指软件开发业务中的关键“业务”过程，即开发和支持软件的组织中的关键“业务”过程（关于设计软件开发业务，参见[2]）。在该业务中还有其他过程，如与产品的用户交互的支持过程，以订单开始并交付产品的销售过程等。不过本书的重点在于开发过程[3]。

2.4.1 过程：一个模板

在统一过程中，过程（*process*）是指通过创建模板的实例来重用一个模板的概念。可以把它比作在面向对象范式中用来创建对象的类。过程实例（*process instance*）是项目（*project*）的同义词。

在本书中，软件开发过程（*software development process*）定义了一个完整的活动集合，这

些活动将用户需求转化成一组前后一致的、表示一种软件产品的制品集合，并将以后的需求变更转化成新的一组前后一致的制品集合。

这里，需求（*requirement*）一词是广义用法，意思是“需要”。开始不必完全了解这些需要。为了能更完整地捕获这些需求或需要，可能必须更加充分地了解客户的业务以及他们的用户的工作环境。

过程的增值结果是一个前后一致的制品集合，它是代表包含软件产品在内的一个或一系列应用系统的基线。

过程是对一组活动（而不是这组活动的执行）的定义。

最后，过程不仅包括第一个开发周期（第一个版本），而且包含之后最常见的周期。在后续的版本中，过程的实例获取需求中的增量变化，并对制品集合产生增量变化。

2.4.2 相关活动组成工作流

我们是从工作流的角度来描述过程的，此处的工作流是指一组活动。这些工作流来源于何处呢？我们不是把过程分割成许多小的相互作用的子过程来得到工作流，也不是使用传统的流程图来描述如何把过程分解为更小的子过程。这些方法用于设计工作流结构时效率不高。

我们采用的方法是，首先确定参与该过程的各种工作人员，然后标识在过程中需要每种工作人员创建的制品。当然，这种标识不是瞬间就能完成的，统一过程需要依靠大量的经验来找出合适的制品和工作人员的集合。一旦标识出该集合，就可以描述过程如何在不同的工作人员间流动以及工作人员如何建立、生产和使用彼此的制品。在图2-6中显示了一个**活动图**（附录A），它描述用况建模中的工作流。注意，“**泳道**”（每个工作人员都有一条，见附录A）中的工作怎样从一个工作人员流向另一个工作人员，以及在这种流动中的工作人员怎样执行活动（由齿轮来表示）。

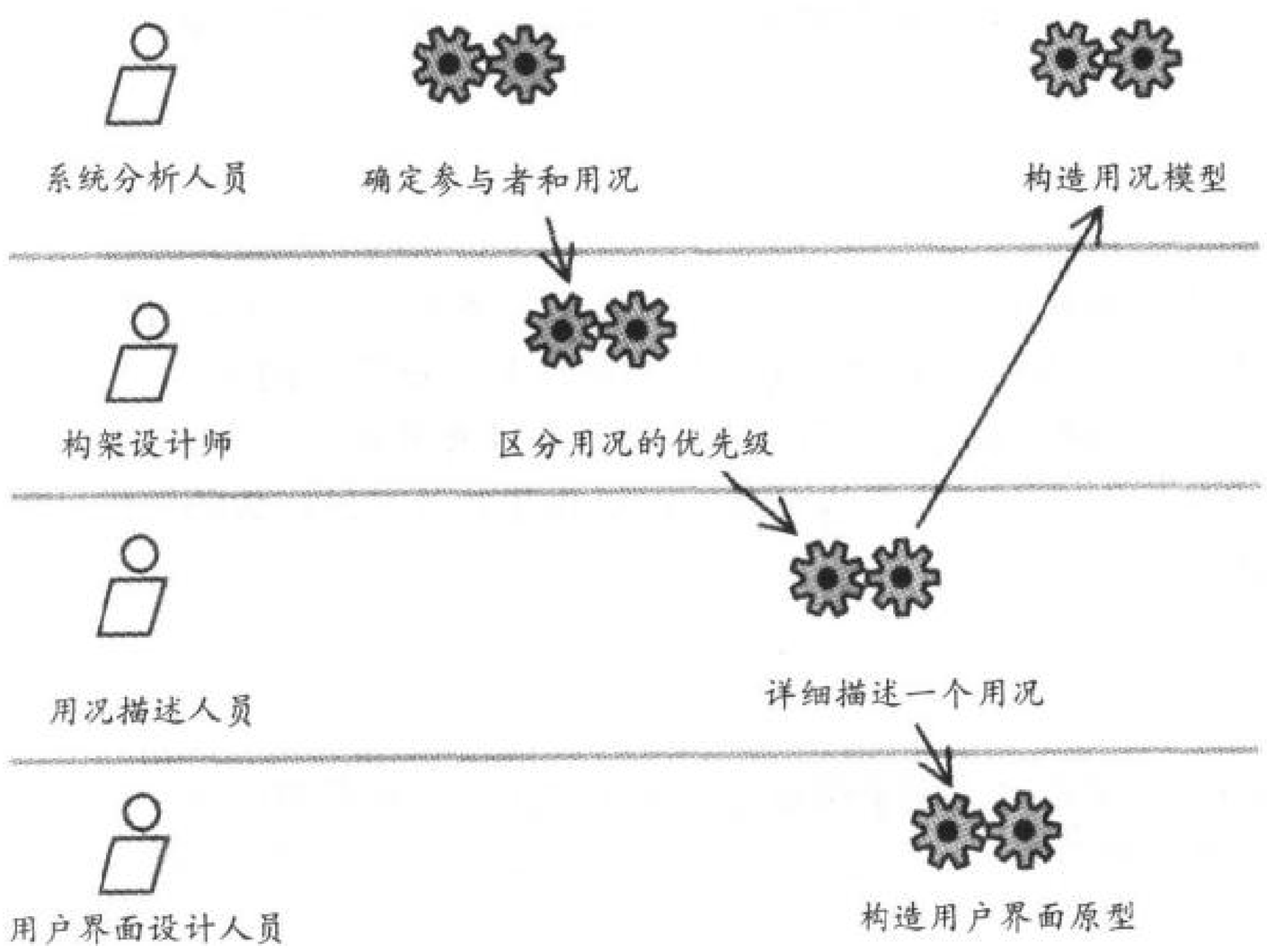


图2-6 “泳道”中有工作人员和活动的工作流

现在我们可以很容易地找出这些工作人员被激活时需要执行的活动。这些工作人员的活动对于充当工作人员的某个人来讲是有意义的工作。而且，从这些描述中能立刻了解工作流中是否需要多次涉及某个工作人员。

换句话说，我们用称为**工作流**（附录C）的片断来描述整个过程。在UML术语中，工作流是协作的一个**构造型**（附录A），其中的工作人员和制品是参与者。因此，一个工作流中所涉及的工作人员和制品也可以（并且通常是如此）参与到其他的工作流中。我们用图2-7中所给的符号表示工作流。

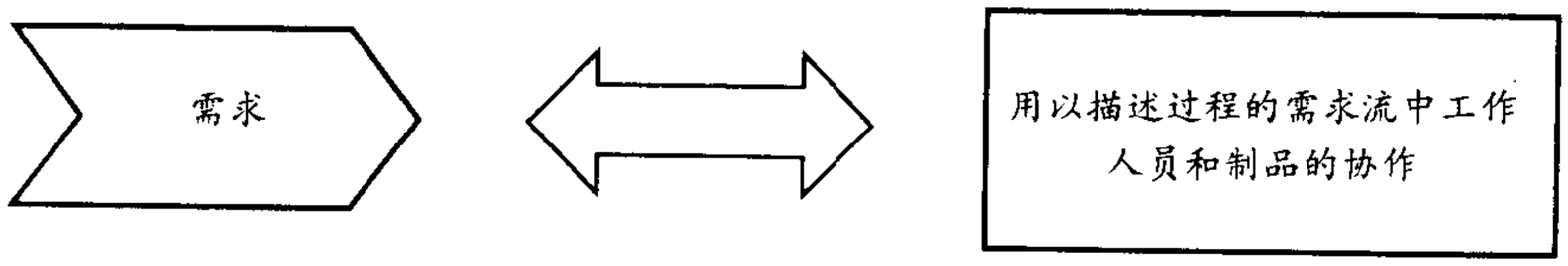


图2-7 “鱼形”符号是工作流的简单表示法

工作流的一个例子是需求工作流，它包括下列工作人员：系统分析人员、构架设计师、用况描述人员和用户界面设计人员；它还包含下列制品：用况模型、用况及其他。工作人员的其他例子还有构件工程师和集成测试人员。制品的其他例子有**用况实现**（附录B；另见第8、9章）、类、子系统和接口。

2.4.3 过程具体化

没有哪个软件开发过程能够适用于任何情况！过程间有所不同是因为它们存在于不同的语境中，要开发不同类型的系统，要满足不同种类的业务约束（如进度、成本、质量和可靠性）。因此，现实世界中的软件开发过程必须是可修改的和可配置的，以便满足一个具体项目或具体组织的实际需要。统一过程被设计为可具体化的（关于设计一个过程，参见[6]）。它是一个通用过程，也就是说是一个过程框架。每个使用统一过程的组织最后都要将它具体化以满足该组织的实际情况（如应用种类和平台等）（关于过程的具体化，参见[8]）。

可以将统一过程具体化以适应不同应用和不同组织的需要。同时，至少在一个组织内部，过程应该是相对一致的。这种一致性使构件在运用时可以互换，使人员和经理在项目之间可以方便地调换，使对成果的度量具有可比性。

造成过程间差别的主要因素有：

- 组织因素（*organizational factor*）：组织的结构、组织的文化、项目组织和管理、具备的能力和技能、过去的经验和现存的软件系统。
- 领域因素（*domain factor*）：应用领域、支持的业务过程、用户群体以及竞争对手可提供的产品。
- 生命周期因素（*life cycle factor*）：投放市场的时间、软件的预计生命周期区间、软件开发技术和人员的专业技能、计划中未来的版本。
- 技术因素（*technical factor*）：程序设计语言、开发工具、数据库、框架和底层的“标准”构架、通信和分布。

这些都是原因，那它们会产生什么结果呢？你可在原有的统一过程中去掉工作人员和制品，以更好地适应成熟度较低的开发组织；也可在原有的统一过程中增加新的工作人员或制品，以使该过程用于你的项目时更为有效。你还可能改变你认为特殊的制品的描述方式，用一种不同的结构对其进行描述。我们的经验是，人们在最初的项目中都会按照统一过程的建议去做。随着时间的推移，人们获得了更多的经验，便进行局部的扩展，开发出适合他们自己的过程。

在设计统一过程中，什么是允许具体化的呢[6]？答案很简单，但最初却不容易理解。统一过程的前身Objectory本身实际就是使用用况、协作和类等对象来设计的。当然，这里的类不是软件对象而是业务对象，即工作人员和制品。它们可在不改变过程设计的情况下具体化或相互转化。在后续的章节里，当描述工作流时，你会发现我们使用对象来描述它们。所用的对象就是工作人员和制品。

2.4.4 过程的价值

在开发组内部或开发组之间共用一个过程具有很多益处：

- 开发组中的每个人都能理解在开发产品时他或她该做的工作。
- 开发人员可以更好地了解其他开发人员正在做的工作——无论是在同一项目的初期或末期，或者在同一公司相类似的项目中，或者在不同的地理位置，甚至是在其他公司的项目中。
- 主管和经理，甚至那些看不懂代码的人员，都能根据构架图了解开发人员正在做的工作。
- 开发人员、主管和经理不必学习一个新的过程就可在项目或部门间进行调动。
- 公司内部的培训可以标准化。培训可以在学院或短期班中进行。
- 软件开发的过程是可重复的，就是说可以足够准确地设定工作进度和估算成本以满足预期的要求。

尽管共用过程有这么多好处，有些人仍然坚持认为它不能解决“真正困难的问题”。对此，我们的回答很简单，“当然不能”。仍然需要人去解决问题。但是一个好的过程可以帮助人们组成优秀的群组。我们可以把它比作对一次军事行动的组织。战役的进行总是需要落实到每个具体执行任务的人身上，但结果却取决于他们的组织的效能。

2.5 工具对于过程不可或缺

工具支持现代的软件开发过程。现在，不使用工具支持的过程而进行软件开发是不可思议的。过程和工具是相互配套的：工具对于过程不可或缺[5]、[7]。

2.5.1 工具对过程的影响

工具支持对过程具有巨大影响。工具适于将重复的任务自动化，使事情更有条理，管理大量的信息，并指导你沿着一条特定的开发路径前进。

如果工具的支持能力有限，过程就得依靠大量的手工工作并因此不太形式化。在实践中，

大部分形式化工作都不得不推迟到实现活动。如果生命周期中的一致性没有工具的自动化支持，则将很难保证及时更新模型和实现，迭代和增量开发就会更加困难。要么最终无法保持一致，要么为保持一致需要进行大量的手工工作来更新文档。后者使整个群组不得不手工地进行所有的一致性检查，从而大大降低群组的效率。这即使可能，也是非常困难的，而且会在开发出的制品中包含大量的错误。用这样的手工方法需要更长的交付时间。

开发支持工具是为了使活动全部或部分自动化，以提高效率和质量，缩短交付时间。在引入支持工具时，我们便会得到一个不同的、更为形式化的过程，进而可以引进没有工具无法进行的新的活动。这样，在整个生命周期的工作就更加精确：可以使用一种形式化的建模语言（如UML）来保证每种模型本身以及与其他模型间的一致性；也可以使用一种模型，并从中生成其他模型的组成部分（如设计到实现或相反）。

2.5.2 过程驱动工具

过程，无论是显式还是隐式定义，都确定了工具的功能，即工具的用况。当然，“过程”的实际情况是我们需要工具的唯一原因。这些工具将尽可能使过程自动化。

使过程自动化的能力取决于对每个工作人员需要哪些用况和对哪些制品进行管理有一个清晰的了解。自动化过程为所有工作人员并发工作提供了有效的方法，也为检查所有制品的一致性提供了一种方法。

实现自动化过程的工具应该简单易用。为了使这些工具具有高可用性，工具开发人员需要仔细考虑进行软件开发所采用的方式。例如，工作人员如何处理某一任务？他怎样才能意识到工具可以从哪方面帮助他？哪些任务需要多次重复因而值得自动化？哪些任务很少出现因而不值得包含到工具中？工具该如何引导工作人员把时间花在只有他才能做的重要任务上，而把重复性的、工具可以做得更好的任务留给工具去做？要回答这些问题，工具必须便于工作人员理解和使用。而且，为了使工作人员认为值得花时间去学习，工具必须确实能大大提高生产率。

对于为什么要制定简单易用的目标存在一些特殊原因。工作人员应能试验不同的选择，应能很容易地对每种选择的候选设计进行揣摩。他们应能选择一种方法进行试验，如果弄清楚这种方法是不可行的，能够轻易地从这种方法变化到另一种方法。工具应能使工作人员尽可能多地重用构件；而不需要对每种尝试的方法都不得不从头再来。总而言之，工具必须既要支持对重复性活动和由一系列模型和制品表示的信息管理的自动化，还要鼓励和支持作为重要开发活动核心的各种创造性活动。

2.5.3 平衡过程和工具

因此，开发过程时如果不考虑怎样将它自动化，则开发过程只会具有理论意义。开发工具时如果不知道它们用来支持什么样的过程（框架），则开发工具可能是无用功。过程和工具之间需要保持平衡。

一方面，过程驱动工具的开发；另一方面，工具指导过程的开发。过程的开发和它的工具支持必须并行发展。在发布过程的每个版本时，必须发布相应工具的新版本。每一次发布都必

须保持这种平衡。要使平衡接近于理想化需要进行多次迭代，而且后续的迭代必须以两次版本发布间用户的反馈作为指导。

这种过程和工具的关系是另一个“鸡与蛋”的问题。先有哪一个呢？就工具的情况而言，近数十年来多是先有工具；而过程则还没有开发完善。因此，工具并不像用户预想的那样可以很好地用在未经仔细计划的过程中。我们中许多人对工具的信心有所动摇。软件开发仍然是一种效率不高的手工工作。换句话说，过程必须从工具中学习，而工具必须支持一个计划周密的过程。

我们试图用最清晰的语句表达这一点：过程自动化（工具）的成功开发若没有它们所要支持的过程框架的并行开发是不可能完成的。这一点大家一定要明白。如果你头脑中还存在任何疑问的话，反问一下自己如果不知道银行的业务过程是什么，能不能为它们开发出IT支持产品。

2.5.4 支持UML的可视化建模

我们刚刚说明了工具在实现过程的作用时很重要。

下面看一个在统一过程支持环境的语境中这种工具的有代表性的例子。我们的例子是针对UML的建模工具。UML是一种可视化语言。因此，它应有与许多制图工具包相同的特点，如内嵌编辑环境、格式化、缩放、打印、色彩填充和自动排版。此外，UML定义了句法规则，详细说明了如何组合使用语言的各种元素。所以，工具必须能够保证这些规则得到遵守。这种功能超出了普通制图软件包所具有的功能，普通制图软件包不需要遵守什么规则。

不幸的是，执行这种句法规则毫无例外地会使工具成为不可用的。例如，在模型编辑中，模型会经常存在句法错误，在这种情况下工具应允许存在句法错误。例如**顺序图**（附录A；另见第9章）中的消息允许在类的任何操作定义之前存在。

UML包括许多同样需要得到支持的语义规则，这些规则可并入到建模工具中，作为立即执行或在要求下响应的例程，这些例程遍历模型并检查常见的错误或查找语义或句法上的不完备性。总之，建模工具需要融入更多的UML知识，需要允许开发人员用UML创造性地进行工作。

使用UML作为标准语言，市场将提供比其他建模语言更好的工具支持。这种更为理想的支持部分来自UML的精确定义，另外也归因于UML如今已被广泛认可为正式的工业标准。工具厂商的竞争焦点不再是针对支持多种不同的建模语言，而在于谁的产品能最好地支持UML。这种新的游戏规则对于软件的用户和客户都更为有利。

但是，UML只是建模语言，它并没有定义如何使用UML去开发软件系统的过程。建模工具也不必强制执行某一过程，但如果用户要使用某一个过程，则工具应能支持这个过程。

2.5.5 支持整个生命周期的工具

工具可以支持**软件生命周期**（附录C）的每一个方面：

- 需求管理（*requirements management*）：用来存储、浏览、检查、跟踪、定位一个软件项目的各种需求。需求可以附带一种状态，工具可以允许跟踪某个需求到生命周期中的其他制品，如用况或测试用例（见第11章）。

- 可视化建模（*visual modeling*）：用来使UML的使用自动化，即对应用进行可视化建模并组装。可以与程序设计环境集成在一起来使用这种工具，以确保模型和实现之间的相互一致性。
- 程序设计工具（*programming tool*）：用来提供一系列的工具，包含编辑器、编译器、调试器、错误检测器和性能分析器。
- 质量保证（*quality assurance*）：用来测试应用与构件，即记录和运行驱动图形用户界面（GUI）和构件接口的测试用例。在迭代的生命周期中，回归测试远比在常规开发时重要。自动化测试用例对于提高生产效率至关重要。另外，许多应用还需要进行强度和负载测试。这种应用的构架怎样才能同时容纳多至10 000个并发用户的使用呢？在你把系统部署到这10 000个用户之前，就会希望知道这个问题的答案。

除了这些面向功能的工具，还有其他跨越生命周期的工具。这些工具包括版本控制、配置管理、缺陷跟踪、文件编制、项目管理和过程自动化。

2.6 参考资料

[1] OMG Unified Modeling Language Specification. Object Management Group, Framingham, MA, 1998. Internet: www.omg.org.

[2] Ivar Jacobson, Martin Griss, and Patrik Jonsson, *Software Reuse: Architecture, Process and Organization for Business Success,* Reading, MA: Addison-Wesley, 1997.

[3] Watts S. Humphrey, *Managing the Software Process,* Reading, MA: Addison-Wesley, 1989.

[4] Ivar Jacobson, Maria Ericsson, and Agneta Jacobson, *The Object Advantage: Business Process Reengineering with Object Technology,* Reading, MA: Addison-Wesley, 1995.

[5] Ivar Jacobson and Sten Jacobson, "Beyond methods and CASE: The software engineering process with its integral support environment," *Object Magazine,* January 1995.

[6] Ivar Jacobson and Sten Jacobson, "Designing a Software Engineering Process," *Object Magazine,* June 1995.

[7] Ivar Jacobson and Sten Jacobson, "Designing an integrated SEPSE," *Object Magazine,* September 1995.

[8] Ivar Jacobson and Sten Jacobson, "Building your own methodology by specializing a methodology framework," *Object Magazine,* November–December 1995.

[9] Grady Booch, *Object Solutions: Managing the Object-Oriented Project,* Reading, MA: Addison-Wesley, 1996.

[10] Walker Royce, *Software Project Management: A Unified Framework,* Reading, MA: Addison-Wesley, 1998.

第3章
用况驱动过程

统一过程的目标就是指导开发人员有效地实现并实施满足客户需求的系统。其效率是按照成本、质量和交付时间来衡量的。从评估客户需求到实现之间的步骤并不简单。首先，客户需求不易识别。这就要求我们具备某种捕获用户需求的手段，以使参与项目的每个人能够清楚地了解这些需求。然后必须能设计一个可以满足这些需求的可行的实现方案。最后，必须测试系统以保证客户需求得以全部实现。鉴于这一过程的复杂性，我们将其描述为可以逐步构造工作系统的一系列工作流。

在第1章曾提到，统一过程是用况驱动、以构架为中心、迭代和增量的。本章将探讨统一过程用况驱动的一面。用况驱动首先是在[1]中介绍的，然后在[2]中作了进一步的讨论。以构架为中心和迭代增量的特点将分别在第4章、第5章中集中介绍。把它们分开来讨论，是希望使用况驱动开发的概念更为简单明了。基于同样的目的，对于如何准备实施模型、设计高度统一的子系统、开发高质量的实现构件（参见10.3.2节）以及完成其他各种测试等主题，在此都不作进一步的探讨，因为这些问题都不能帮助解释用况和用况如何驱动开发工作，所以将它们都推迟到第二部分再进行深入探讨。

图3-1表示了统一过程中的一系列工作流和模型。首先，开发人员捕获客户需求作为用况模

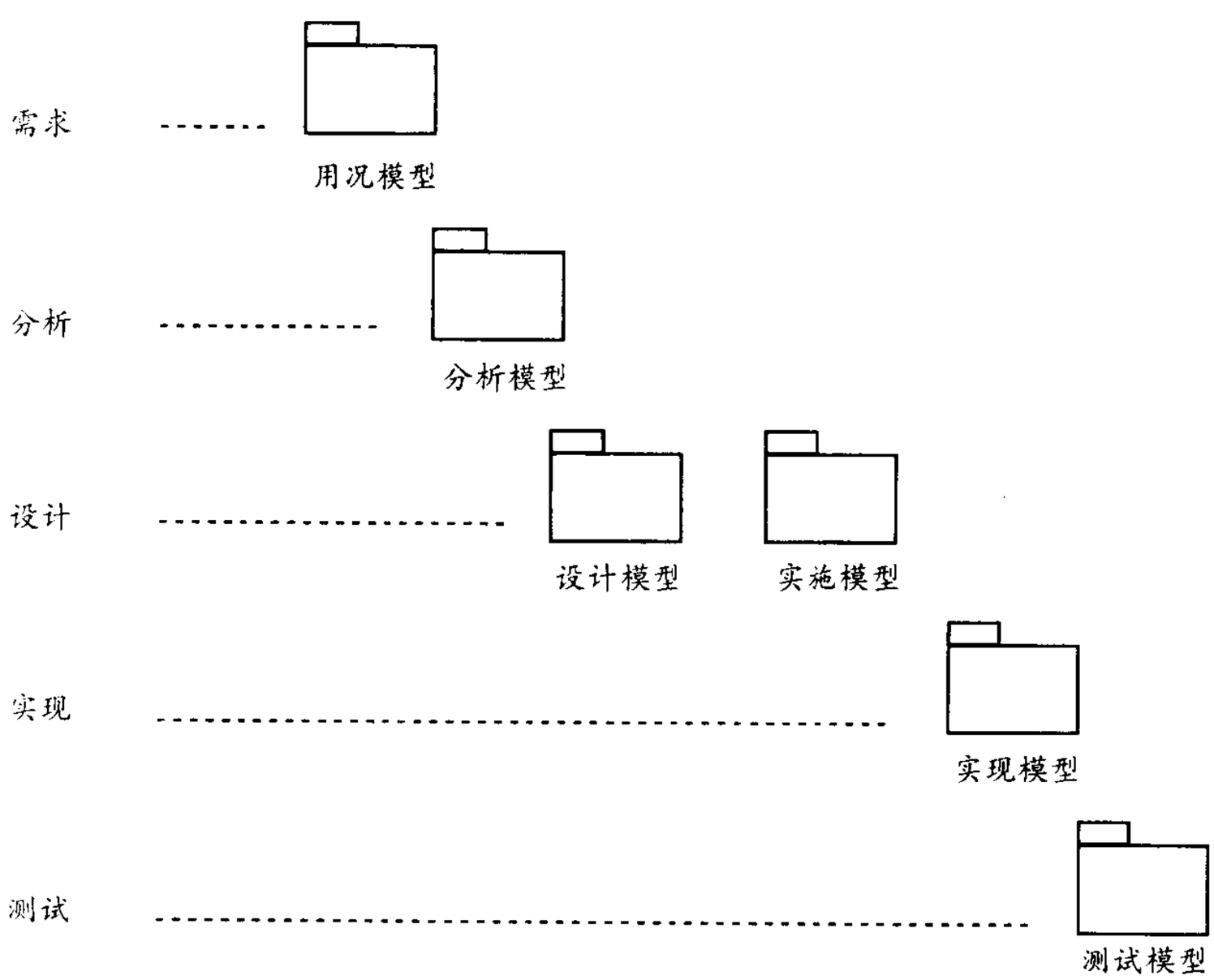


图3-1 统一过程是从需求到测试（左侧，从上到下）的一系列工作流。工作流发展为各种模型：从用况模型到测试模型

型中的用况。然后分析并设计系统来满足这些用况，这样便首先创建了一个分析模型，然后是设计模型和实施模型；进而在实现模型中实现该系统，实现模型中包括所有的代码，即构件。最后，开发人员准备一个测试模型来验证系统是否能够提供用况中描述的功能。所有的模型通过跟踪依赖关系而互相关联。从机器解释的角度看，实现模型是最为形式化的，而用况模型的形式化成分最少。也就是说，部分实现模型可以通过编译和连接成为可执行代码，而用况模型则主要是通过自然语言来描述的。

用况几乎普遍用于捕获软件系统（尤其是基于构件的系统）的需求[6]。但用况不只是捕获需求的工具，它们还能够驱动整个开发过程。在寻找和确定类、子系统和接口（附录A）时，在寻找并确定测试用例时，在规划开发迭代和系统集成（见第10章）时，均可以将用况作为主要输入。对于每一次迭代，用况驱动完成一整套工作流（从需求捕获，经由分析、设计和实现，到测试），并把这些不同的工作流结合在一起。

3.1 用况驱动开发概述

需求捕获有两个目标：发现真正的需求（参见第6章）并以适于用户、客户和开发人员的方式加以表示。“真正的需求” 是指在实现时可以给用户带来预期价值的需求。“以适于用户、客户和开发人员的方式”主要是指对需求的最后描述必须能够让用户和客户理解。这是需求工作流的主要难点之一。

一个系统通常有多种用户。每种用户表示为一个参与者。参与者在与用况进行交互时使用系统。用况是系统为了向参与者提供某些有价值结果而执行的动作序列。系统的所有参与者和用况构成用况模型[3]、[4]。

在分析和设计期间，用况模型经由分析模型转化为设计模型。简单地说，分析模型和设计模型都是由**类元**（附录A）和说明如何实现用况的用况实现集合（见第8、9章）组成的。类元一般是“似类的”东西。例如，它们具有**属性**和**操作**（附录A），可以用**状态图**（附录A）进行描述，有些可以实例化，可以参与协作等。UML具有很多种类元。子系统、类和接口都是这种结构中类元的例子。其他的类元有参与者、用况、构件和**节点**（附录A；另见9.3.7节）。

尽管分析模型本身是一种模型，但它是需求的详细的规格说明，并可作为设计模型的切入点。开发人员使用该模型将需求工作流中描述的用况精化为概念性类元（与将要实现的设计类元相对应）间的协作，以便更准确地理解这些用况。分析模型还用于创建使用大量可重用构件的、健壮的柔性系统（包括构架）。分析模型与设计模型不同，它是一个概念模型而不是实现蓝本。分析模型可能是暂时的，只是存在于前几次迭代中。然而，在某些情况下，尤其对于大型、复杂系统，分析模型会存在于系统的整个生命周期。这种情况下，分析模型的用况实现与设计模型中相应的用况实现之间存在一种无缝关系（通过跟踪依赖关系）。分析模型中每个元素都可以从实现它的设计模型的元素中跟踪到。（分析模型的作用及其与设计模型的关系将在8.1 ~ 8.3

节中详细讨论。）

设计模型具有以下特点：

- 设计模型是有层次的，但也包括跨越层次的关系。这些关系在UML中很常见：**关联**、**泛化**和**依赖**（附录A）。
- 用况实现是协作的构造型。协作表示类元在做某件有益的事情（如实现用况）时如何参与和充当角色。
- 设计模型还是实现的蓝图。在设计模型的子系统和实现模型的构件之间存在直接的映射关系。

开发人员使用用况模型作为输入来创建分析模型[5]。用况模型中的每个用况都实现为分析模型中的用况实现。用况和用况实现的对偶性支持需求和分析之间的无缝可跟踪性。通过对用况进行处理，开发人员可以确定参与实现用况的类。例如，“取款”用况可以通过分析“取款”类、“账户”类、“分配”类及其他几个无需在此进行标识的类来实现。开发人员将用况中定义的**职责**（附录A）分配为类的职责。在我们的例子中，“账户”类包括诸如“从账户上取一定数额的现金”的职责。这样，我们能确保得到一个类的集合，这些类合在一起可以实现真正是用户所需要的用况。

然后，开发人员设计类和用况实现，以便更充分地利用实现系统的产品和技术（如对象请求代理、图形用户界面（GUI）构造工具和数据库管理系统）。按照子系统对设计类进行分组，并定义这些子系统间的接口。开发人员还要进一步建立实施模型，其中包括定义可计算节点上系统的物理结构和验证用况能够实现为可运行于这些节点上的构件。

然后，开发人员把设计好的类实现为实现模型中的文件构件（源代码）集合，从中能够产生（即编译和连接）如DLL、JavaBeans和ActiveX等可执行代码。用况有助于开发人员决定实现和集成构件的次序。

最后，在测试工作流期间，测试人员验证系统确实能够实现用况中所描述的功能和满足系统需求。测试模型包含测试用例（和其他内容，这些内容将在第11章中讨论）。测试用例定义了输入、运行条件和结果的集合。很多测试用例可直接从用况中得到，因此，在测试用例和相应的用况之间存在跟踪依赖关系。这意味着测试人员将验证系统能够做用户需要它做的事，即能够执行用况。任何过去测试过软件的人员都测试过用况，尽管那时不用该术语描述这项工作[8]。统一过程新的和不同于其他方法的特点在于，测试可以在开发周期的初期进行规划。一旦捕获到用况，就可以详细说明用况的测试（“黑盒”测试），并决定实现、集成和测试它们的次序。当用况在设计阶段实现后，可以进一步细化用况的测试（“白盒”测试）。每种执行用况的方式（即用况实现的每条路径）都是一个候选的测试用例。

至此，我们已经将统一过程表达为一系列步骤，很像过去的瀑布方法。我们这样做只是希望使问题在这一点上得到简化。第5章中将看到如何用更为有趣的迭代和增量的方法来实现这些步骤。事实上，这里所描述的就是一次迭代。整个开发项目就是一系列的迭代，其中每次（但也许是第一次）迭代都经历需求、分析、设计、实现和测试等工作流。

在详细探讨不同的工作流之前，让我们先仔细看看用况的益处。

3.2 为什么使用用况？

用况具有益处且被普遍采用有多种原因[6]，其中两个主要原因是：

- 它们着眼于为用户增加价值，提供了一种捕获**功能需求**（附录C；另见第6、7章）的系统而且直觉的方法。
- 它们可驱动整个开发过程，因为大部分活动如分析、设计和测试都是从用况开始执行的。设计和测试可根据用况进行规划和协调。当构架经过第一组迭代后稳定下来时，这种特性在项目中尤为明显。

3.2.1 为了捕获附加在需求上的价值

按照Karl Wieger的说法，“用况所提供的视角补充了软件工程的最终目标：创造的产品可以让客户做有用的工作。”[8]。之所以如此有以下几个原因。

首先，用况的构造有利于识别满足用户目标的软件，用况是系统向其用户提供的增值功能。通过收集各种用户的观点，能够捕获他们所需要完成工作的用况。相反，如果我们虚拟出一组很好的系统功能，而没有考虑各个用户使用的用况，我们就会发现很难断定这些功能是否重要或很好。我们不知道：它们辅助谁进行工作？需要满足什么业务？能够对业务增加多少价值？

最好的用况是能够为使用该系统的业务提供最大的价值。为业务提供负面价值或允许用户做他或她并不能做的事情的用况不是真正的用况。因为这种用况指出了必须禁止使用系统的方法，所以我们称之为“不良用况”。这种用况的例子如允许银行的储户将他人账户上的存款转移到自己的账户上。为业务提供很少价值或无价值的用况基本上不会用到，它们是不必要的“无用用况”。

正如在第1章中提到的，多数人发现只是简单的问系统需要做什么并不能帮助他们获得正确的答案。取而代之，他们设法得到一张系统功能的列表，乍一看似乎很有用，但仔细研究后发现它们相对用户需要并非必要。在“你希望系统做什么？”的问题中增加三个词“*for each user*”（译为“为每个用户”）该问题变为“你希望系统为每个用户做什么？”这种用况策略似乎只有很细微的差别，但它却产生了具有很大差别的结果。它使我们将注意力集中于了解系统怎样才能适用于每个用户，指导我们找出每个用户所需要的功能，它还有助于我们避免提出用户根本就不需要的多余功能。

其次，用况很直观。用户和客户根本不懂复杂的符号，相反，简单的英语（如自然语言）可以用于大多数场合，这使它更易于阅读用况描述和提出修改建议。

捕获用况涉及到用户、客户和开发人员。用户和客户是需求专家。开发人员的作用是与用户和客户进行讨论，并帮助他们表达自己的需求。

用况模型用来与用户和客户在“系统应该为用户做什么”方面达成共识。可以认为用况模型是所有可能使用系统（用况）的方式的完整的规格说明，它可以用作与客户签订合同的一部分。用况模型通过定义系统必须为用户做的事情来界定系统。参考资料[12]中介绍了一种构造用况（模型）的值得关注的方法，参考资料[11]中对用况进行了全面的介绍。

3.2.2 为了驱动过程

正如我们所说过的，用况驱动意味着一个开发项目是通过从用况初始化开始的一系列工作流进行的。这些用况有助于开发人员找出类。这些类是开发人员在阅读用况说明以寻找适于实现这些用况的类时获得的[⊖]。这些用况还能帮助我们开发用户界面，使之更易于用户执行它们的任务。随后，这些用况实现要经过测试以证明这些类的**实例**（附录A）能够执行这些用况[8]。

用况不仅启动一个开发过程，而且将其结合为一个整体，如图3-2中所示。

图3-2 用况将核心工作流结合为一体。背景中着色的椭圆表示用况是如何将这些工作流连为一体的

我们还要确保捕获正确的用况，以便用户得到他们真正需要的用况。解决这一问题最好的方法当然是在需求阶段彻底做好，但这往往是不够的。一个可以执行的系统允许我们进一步确认这些用况是否是实际的用户需求。

用况有助于项目经理规划、分配和监控开发人员所执行的多个任务。对于每个用况，项目经理可以标识出一些任务。标识、设计和测试每一个用况都分别是一个任务。项目经理可以估算这些任务所需要的工作量和时间。根据用况确定的任务有助于项目经理估计项目的规模和所需要的资源，然后把这些任务分配给该项目的开发人员。项目经理可以让其中一个开发人员在需求阶段负责确定五个用况，另一个开发人员负责设计三个用况，第三个开发人员负责为两个用况确定测试用例。

用况是保证所有模型具有可跟踪性的一种重要机制。需求阶段的一个用况可以跟踪到在分析和设计阶段的实现，还可以跟踪到参与实现它的所有类、构件（间接的）以及最后用来验证该用况的测试用例。这种可跟踪性是管理项目的一个重要方面。当用况发生变化时，必须对相应的实现、类、构件和测试用例进行核查以便更新（参见[10]）。用况和其他模型元素之间的可跟踪性，可以更加容易地保持系统的完整性和保证系统与需求变更的一致性。

3.2.3 为了设计构架以及其他

用况有助于进行迭代开发。可能除了项目中第一次迭代之外，每次迭代都是由用况驱动而经历所有工作流的，即从需求到设计和测试，进而得到一个**增量**（附录C）结果。因此，每次开发的增量都是一个用况集合的具体实现。换句话说，在每次迭代中，都会确定并实现一些用况。

用况还有助于设计构架。在最初的几次迭代中，通过选择实现适当的用况集合（对构架很关键的用况），便可以用一个稳定的构架来实现一个系统，该构架可以用于多个后续的开发周期。第4章中将继续讨论这一问题。

用况还可作为编写用户手册时的起点。因为每个用况说明了一种使用系统的方法，因此用况是说明用户怎样与系统进行交互的理想起点。

⊖ 这是一种简化。事实上，每个用况可能用到几个已经开发好的类（子系统等），所以我们要改变这些用况以便适用于这些可重用的构造块。这将在4.3节中讨论。

通过估计执行用况时通过不同路径的频度，可以估计出哪条路径需要最佳性能。对某些用途来说，这样的估计可以用来衡量底层硬件的处理器能力或优化某类应用时的数据库策略。类似的估计还可以用来获得良好的可用性，即当设计用户界面时，特别注意选择最重要的路径。

3.3 捕获用况

现在简单看一下所有的任务是如何通过工作流来执行的。如前面所述，我们将着重于用况驱动方面。以下章节侧重讨论捕获功能性的需求作为用况，当然还需要捕获其他类型的需求。

在需求工作流期间，我们标识用户和客户的需要作为需求。将功能性需求表示为用况模型中的用况，其他的需求或者“附加到”与之有关的用况上，或者放到一个单独的列表中，或者以其他某种方式描述。

3.3.1 用况模型表示功能性需求

用况模型帮助客户、用户和开发人员在如何使用系统方面达成共识。大多数系统具有多种类型的用户。每类用户表示为一个参与者（*actor*）。参与者在与用况进行交互时使用系统。系统所有的参与者和所有用况组成用况模型[2]、[3]。一张**用况图**（附录A；另见7.4.1节）描述部分用况模型，显示带有关联关系（表示出一对参与者和用况之间的交互）的用况和参与者的集合（如图3-3所示）。

举例：自动取款机（ATM）系统的用况模型

参与者“银行储户”使用自动取款机系统从账户中取款，或存款到账户中，或在不同的账户间转账。这可以由图3-3中的三个用况表示，这三个用况与参与者之间的关联表示了它们之间的交互。

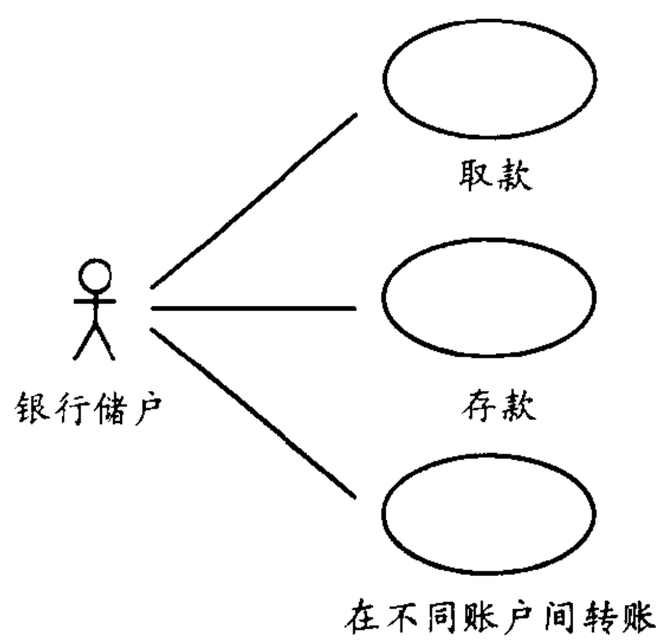


图3-3 具有一个参与者和三个用况的用况图举例

3.3.2 参与者是系统的环境

所有的参与者无需只代表人。参与者可以是与该系统发生交互的其他系统或外部硬件。每个参与者在与系统交互时均充当一系列相应的角色。一个真正的用户可以作为一个或几个参与

者，当他们与系统进行交互时发挥这些参与者的作用。几个用户可以作为一个用户的不同出现而充当同一个参与者。例如，银行的储户可以有成千上万个人。

参与者在执行用况时通过向系统发送**消息**（附录A）或从系统接收消息来与系统进行通信。当我们明确了参与者和用况应该分别做什么时，便可以在参与者的职责和系统的职责之间进行清晰划分，以便帮助我们限定系统的范围。

通过考虑哪些用户将使用该系统以及哪些其他的系统必须与该系统发生交互，就可以发现并详细说明所有的参与者。每一类用户或交互系统都表示为参与者。

3.3.3 用况确定系统

确定用况是为了满足用户在使用系统时的需要。用况模型捕获系统的所有功能性需求。我们给出用况的精确定义如下：

用况规定了一个动作序列（可以有多种实现），系统可以执行这些动作并产生出一个对于特定参与者有价值的可见结果。

通过考虑用户如何使用系统完成他们的工作来发现用况（关于基于目标构造用况的方法，参见[10]）。每个能够对用户增值的系统使用方式就是一个候选用况。然后对这些候选用况进行详细说明、改变、划分为更小的用况或结合成为更加完整的用况。当以客户、用户和开发人员都能理解的方式正确地捕获了全部功能性需求时，用况模型便基本上完成了。在操作期间（如一个用况实例），由一个用况执行的动作序列是一条完成该用况的具体路径。可能存在多条路径，并且很多路径可能是相似的，它们都是由用况规定的动作序列的不同实现。为了使用况模型易于理解，应该将相似的实现路径组成一个用况。当我们说规定和描述一个用况时，意味着规定和描述了实际上定义为相同用况的各种路径。

举例："取款"用况

完成该用况动作序列的一条路径是非常简单的，在此简化为：

1) 银行储户表明自己的身份。

2) 银行储户选择从哪个账户取款，并确定取款金额。

3) 系统从账户上减掉该数量的金额，发给该储户相应金额的货币。

用况还可用来说明**非功能性需求**（附录C；另见第6章），如对某个用况特定的性能、可用性、准确度和安全性等需求。例如，下面的需求可以附加到"取款"用况上：银行储户从选择取款数量到得到相应的货币的响应时间在所有用况的95%中应该小于30秒。

总之，所有的功能性需求确定为用况，而很多非功能性需求可以附加到这些用况上。事实上，用况模型是以一种易于管理的方式组织需求的工具。客户和用户可以理解并用一种一致的、非冗余的方式来表达他们的需要。开发人员可以将捕获需求的任务在他们之间进行分工；然后将这些结果（原始用况）用作分析、设计、实现和测试系统时的输入。

3.4 实现用况的分析、设计和实现

在分析和设计期间，将用况模型经由分析模型转换到设计模型，即转换到类元的结构和用况

实现。目标是为了有效地实现用况，以便系统能够提供合适的性能，并在将来能够进一步改进。

本节将介绍如何经由分析来开发一个设计以实现用况。在第4章和第5章，我们将会看到构架、迭代和增量开发如何帮助我们卓有成效地开发一个能够满足新需求的系统。

3.4.1 根据用况建立分析模型

随着经过分析的用况越来越多，分析模型会逐渐完善起来。对于每次迭代，选择一组在用况模型中实现的用况。我们将系统构造成为一些类元（分析类）以及这些类元间关系的结构。我们还要说明实现这些用况的协作关系，即用况实现。然后，在下一次迭代中，挑选另外一组用况去实现，并进而将它们加入到前一次迭代所选的用况中。在5.3节和12.6节，将讲述如何为第一次迭代确定和选择最关键的用况集合，以便在系统生命周期的初期构造一个稳定的构架。

一种切实可行的处理方法是为某次迭代标识并描述用况，然后阅读一遍每个用况的说明（如3.3.3节中的举例），并考虑实现该用况需要哪些类元和关联。对于一次迭代中的所有用况进行同样的处理。根据我们在生命周期中所处的阶段和所处理的迭代种类，可能已经存在一个构架来指导我们发现新的类元和重用已有的类元（参见4.3节）。每个类元在一个用况实现中充当一个或几个角色。每个类元角色详细说明该类元参与实现某个用况的职责和属性等。我们可以把这些角色看作是类元的“萌芽”。事实上，在UML中角色本身也是类元。例如，可以把类的角色认为是类的视图。这样，它便包括了它的类所包括的内容，即职责和属性等——但只限于在用况实现中对其角色有意义的那些内容。描述类的角色的另一种方式是，把类的角色看成是将该类通过一个滤波器后所剩余的部分，该滤波器将屏蔽该类中没有共同职责的所有其他角色。出于简单考虑，在本章中只是简单地说明这种角色的概念。在本书的第二部分将详细讨论所有的类元。

举例：分析模型中用况的实现

在图3-4中，描述了“取款”用况如何通过一个带有《跟踪》依赖关系的协作（即一个用况实现）来实现（其中，跟踪是一种由符号“《 》”表示出的依赖关系的构造型），在这个用况实现中有四个类参与并充当角色。如图中所示，虚线椭圆表示用况实现或协作。

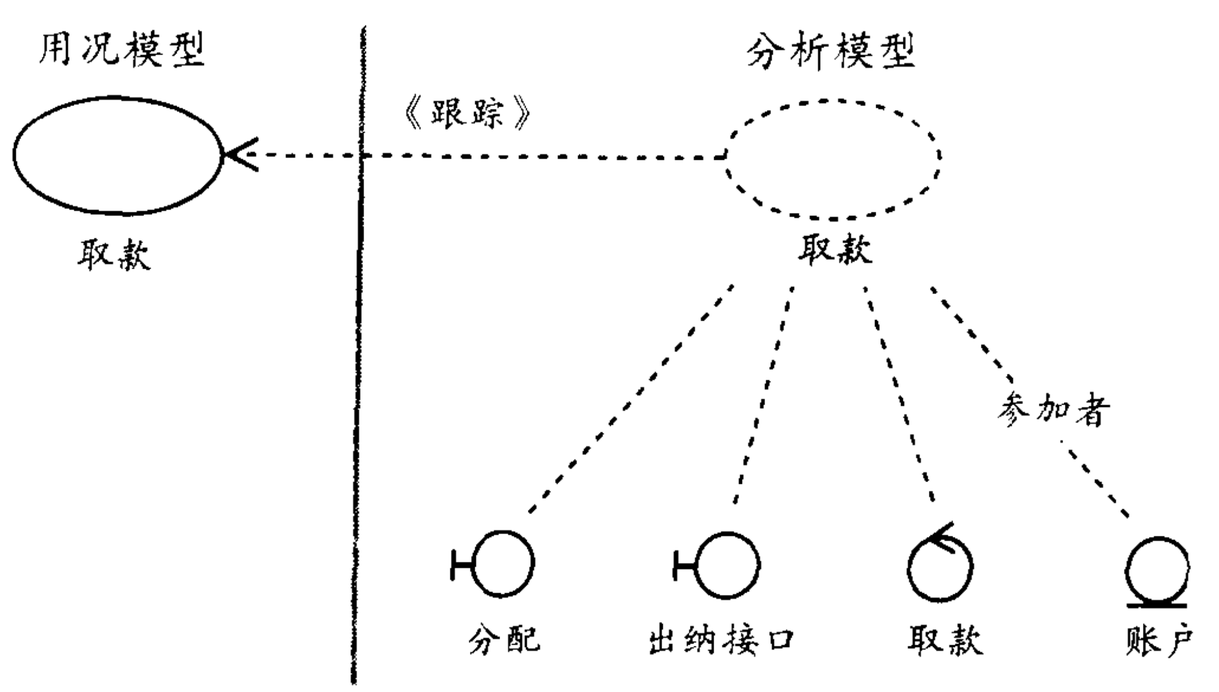


图3-4 参与“取款”用况实现的分析类。“分配”和“出纳员接口”是边界类，“取款”是控制类，“账户”是实体类

通常，我们从考察一些用况、为它们建立用况实现并确定类元的角色开始。然后对更多的用况重复同样的工作并确定新的类元角色。之后的一些角色可以认为是已确定类元的新的或变更的角色，而另一些角色需要新的类元。然后再考察一下最初的那些用况，经过这样一些考察与变换，逐渐构造出一个稳定的分析模型。因此，每个类元可以参与多个用况实现并从中充当一定的角色。

分析构造型

在分析模型中使用了关于类的三种不同构造型：《边界类》、《控制类》和《实体类》。“分配”和“出纳员接口”是边界类，一般用于建立系统与其参与者（即用户和外部系统）间交互的模型。“取款”是控制类，通常用于表示协调、排序、事务处理以及其他对象的控制，并且它还经常用于封装与特定用况（如本案例中的“取款”用况）有关的控制。“账户”是实体类，一般用于建立长效且持久的信息模型。于是，这些类的每一种构造型分别封装不同类型的行为（或功能，如果你希望如此）。因此，构造型有助于我们构造出健壮的系统，在第8章将更为详细地讨论这些问题。另外，构造型还能帮助我们发现可重用的资源，这是因为实体类对于很多用况、进而对于很多不同的应用常常是通用的。用这三种构造型建造的各种分析类在[2]中进行了详细的描述。构造型已经应用了多年并得到广泛的认同，现在已被UML采纳[12]。

举例：在分析模型中参与多个用况实现的类

在图3-5中，左边是ATM系统（与图3-3中相同）的用况集合，右边是对应的系统结构，在本例中，是实现用况的分析类。系统结构在**类图**（附录A）中进行建模。类图一般用来显示类以及类之间的关系，但也可以用来显示子系统和接口（在3.4.3节中讨论设计时将会看到这一点）。为简单起见，已经选用不同的底纹来表示一个类参与哪个用况实现并在其中充当什么角色。

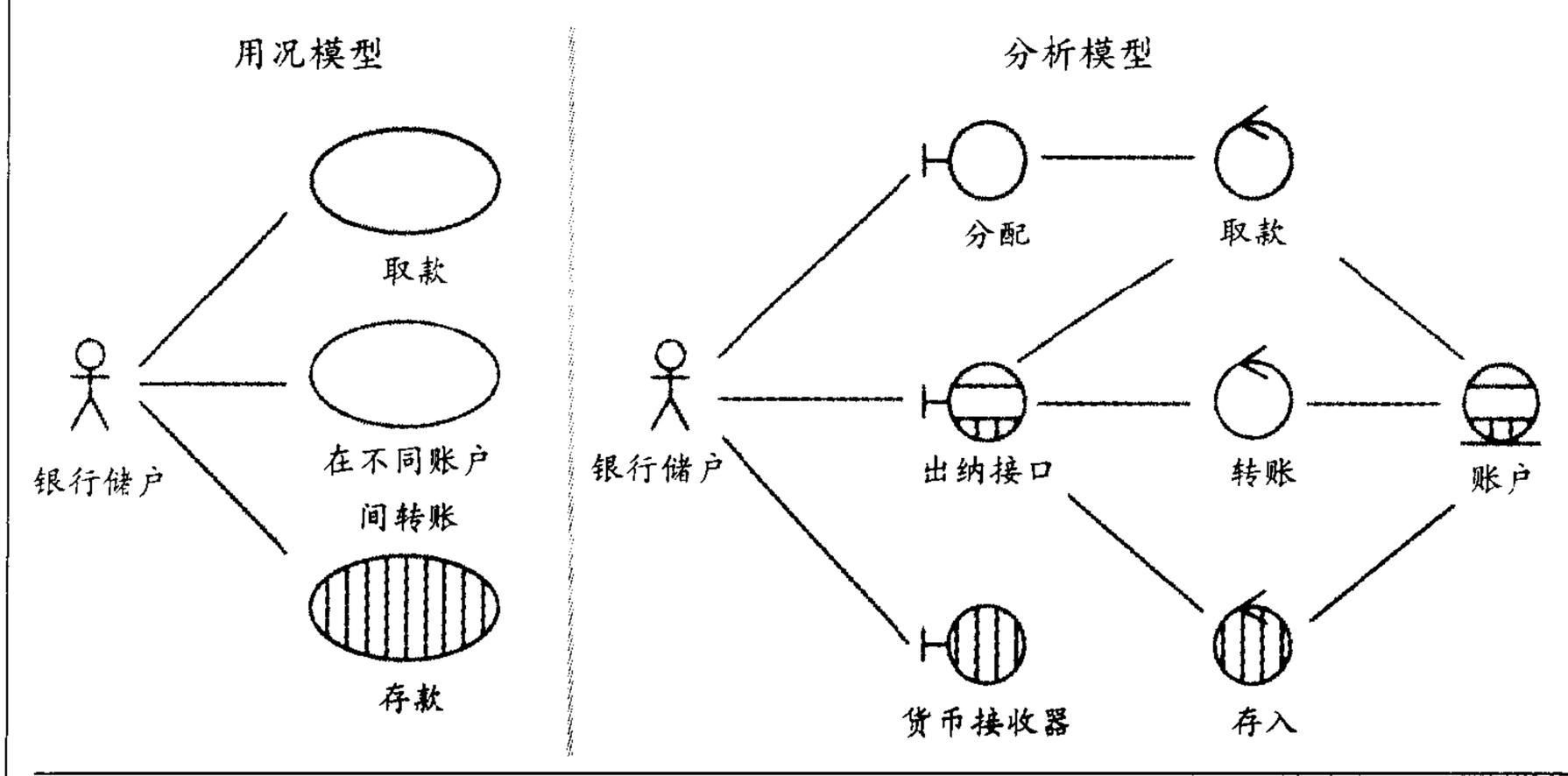


图3-5　图中显示了每个用况（左图）如何实现为分析类结构（右图）。例如，“出纳员接口”类、“取款”类、“账户”类和“分配”类全部参与了“取款”用况的实现。“出纳员接口”类和“账户”类参与并在全部三个用况实现中充当角色。其他的类只参与了一个用况的实现，即它们只充当一个角色

ATM系统（图3-5）的类图是通过对三个用况进行描述并接着寻找实现每个用况的方法来建立的。可以按照下面所介绍的方法来做：

- 三个用况的实现“取款”、“转账”和“存款”涉及到边界类“出纳员接口”和实体类“账户”。每个用况的执行均开始于“出纳员接口”的一个**对象**（附录A）。然后把任务传递给前面提及的协调大多数用况的控制对象。这种对象的类对于每个用况是唯一的。于是，“取款”类将参与“取款”用况等。“取款”对象请求“分配”去分发货币，并请求“账户”对象去减少余额。
- “转账”对象请求“在不同账户间转账”的用况实现中所涉及的两个“账户”对象去更新余额。
- “存入”对象通过“货币接收器”接受现金并请求“账户”对象去增加余额。

至此，我们经过努力在当前的迭代中确定了一个稳定的系统结构。我们已经明确了所涉及的类元的职责，并且建立了类元之间的关系。然而，我们还没有确定在实现用况时必须进行的迭代细节。我们已经创建了结构，但现在需要在结构上叠加每个用况实现所要求的不同的迭代模式。

正如在前面所提及的，每个用况均对应一个用况实现，每个用况实现都包含一个充当不同角色的类元集合。理解交互模式意味着要说明如何执行或运行（或实例化）一个用况实现。例如，当一个银行储户取款时，即当他或她执行“取款”用况时会发生什么。我们知道“出纳员接口”类、“取款”类、“分配”类和“账户”类将参与实现该用况，也知道它们将承担什么职责。但是，我们仍然无从知道它们——更确切地说是那些类的对象——将怎样交互来完成用况实现，这就是我们下一步需要弄清楚的。在分析中我们主要使用**协作图**（附录A）来建立对象间交互的模型。（UML还使用顺序图来建立交互模型，在3.4.3节中讨论设计时再介绍这一点。）协作图与类图相似，但它显示的是实例和**链**（附录A）而不是类和关联。它通过对交换消息进行编号来显示出对象如何顺序地或并行地进行交互。

举例：在分析模型中使用协作图来描述用况实现

在图3-6中，用协作图描述了取款的用况实现是如何通过一组分析对象来执行的。

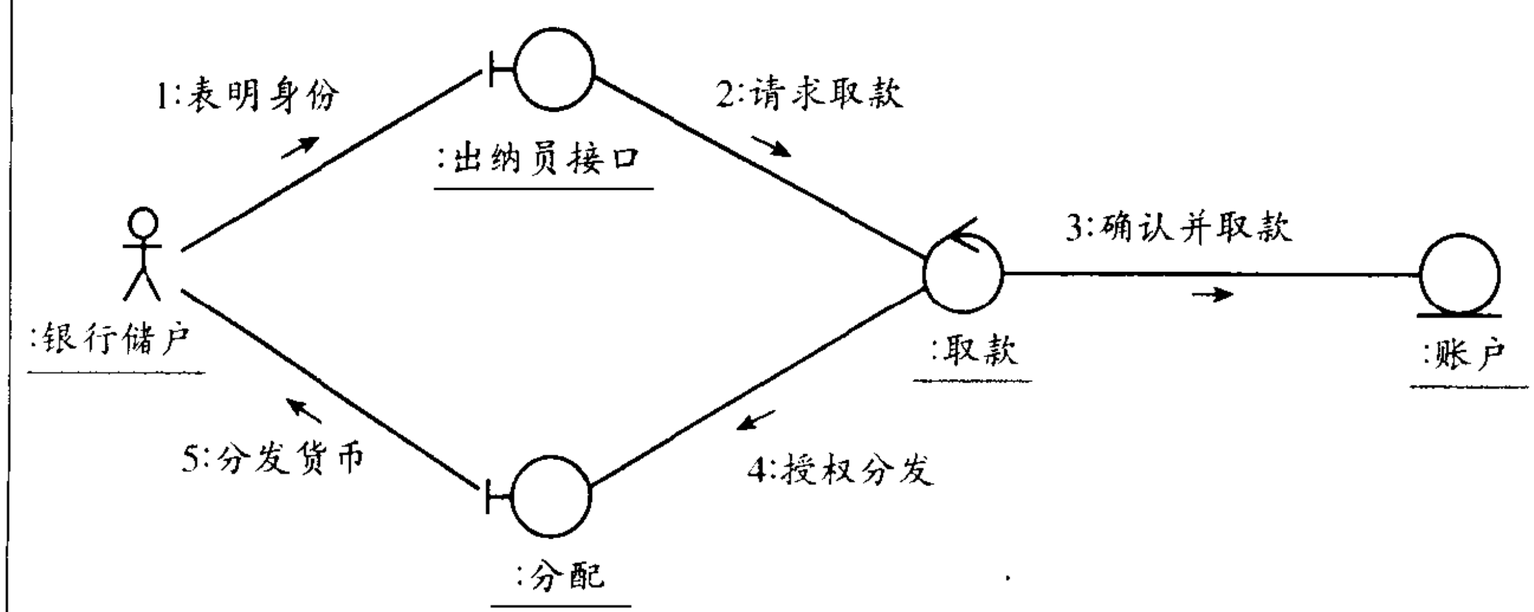


图3-6 分析模型中“取款”用况实现的协作图

图中显示了在执行用况时工作重点如何在对象间转移以及消息如何在对象间传递。从一个对象发送的消息触发接收对象作为工作重点，并执行它的对应的类的一种职责。

消息名称表明了调用对象与被调用对象进行交互时的意图。在设计期间，这些消息将具体化为由对应的设计类（参见3.4.3节）所提供的一个或多个操作。

作为对协作图的补充，开发人员还可以用文本来解释对象如何执行用况的事件流。可以有多种方式描述实现，如使用结构化文本或伪码。

举例：用况实现的事件流描述

现在根据图3-6中所示的交互对象和参与者来说明“取款”用况的实现。

银行储户选择取款并激活“出纳员接口”对象。银行储户表明自己的身份并确定从哪个账户取多少现金（1）。“出纳员接口”验证银行储户的身份并请求“取款”对象执行事务（2）。

如果银行储户的身份合法，便请求“取款”对象确认该银行储户有权从指定账户中取出确定数量的货币。“取款”对象通过“账户”对象验证该请求来予以确认，如果该请求合法便取出相应数量的货币（3）。

然后“取款”对象授权“分配”去分发银行储户所请求数额的货币（4）。然后银行储户接收他或她所请求数量的货币。

注意，这个简单的例子只显示了没有任何复杂情况时用况实现的一条路径。然而，如果账户内余额太少而不允许取款时，便会出现复杂情况。

以下我们将分析每个用况，并由此确定参与每个用况实现的所有类的角色。下面首先讨论如何分析每个用况。

3.4.2 每个类必须履行其所有的协作角色

一个类的职责就是该类在所有用况实现中所充当角色的汇集。将角色集中起来并去掉其中重叠的部分，就可得到该类的所有职责和属性的规格说明。

负责分析和实现用况的开发人员的责任是确定类的角色。负责一个类的某个开发人员把这个类的所有角色汇集为该类的一个完整的职责集合，然后将它们集成为一个一致的职责集合。

负责分析用况的开发人员必须确保这些类能够恰当地实现用况。如果某个类发生了变化，该类的开发人员必须验证它仍然能够在用况实现中充当所要求的角色。如果用况实现中的某个角色发生了变化，用况开发人员必须将这种变化通知该类的开发人员。因此，角色可以帮助类的开发人员和用况的开发人员维持分析的完整性。

3.4.3 根据分析模型建立设计模型

设计模型是通过使用分析模型作为主要输入而被创建的，并应使它适用于所选的实现环境，如对象请求代理、GUI（图形化用户界面）的构造工具或者数据库管理系统。它还应适用于重用**遗留系统**（附录C）或其他为该项目开发的框架。这样，分析模型便成为设计模型的原始依据，

而设计模型则成为实现的蓝图。

与分析模型类似，设计模型也要定义类元（类、子系统和接口）、这些类元间的关系以及实现这些用况的协作（即用况实现）。然而，从一定意义上讲，在设计模型中定义的元素是分析模型中定义的更加概念化的元素的“设计副本”，因为前者（设计）的元素适应于实现环境，而后者（分析）的元素则不然。换句话说，设计模型在本质上是更加“实际的”，而分析模型则是更加“概念的”。

分析模型中的用况实现可以跟踪到设计模型中的用况实现。

举例：分析模型和设计模型中的用况实现

图3-7中描述了“取款”用况是如何由分析模型和设计模型中的用况实现予以实现的

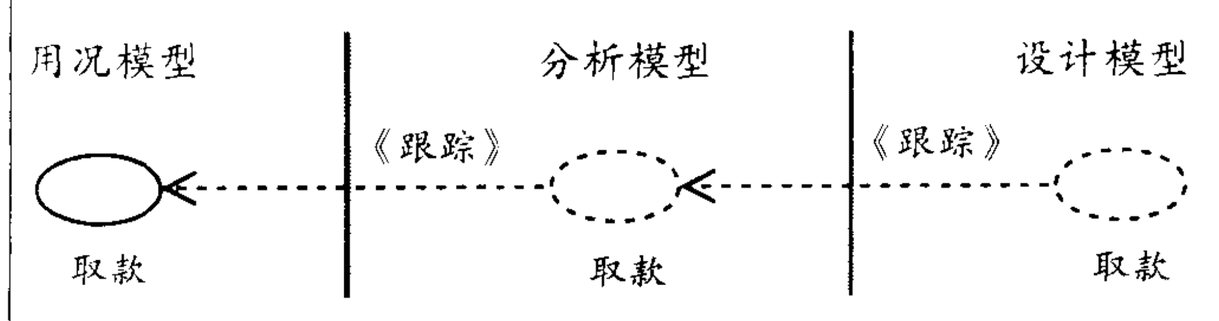


图3-7 不同模型中的用况实现

不同模型中的用况实现用于不同的目的。回想一下3.4.1节（图3-4）中的分析类“出纳员接口”、“取款”和“分配”都参与了实现分析模型中的“取款”用况。然而，当设计这些分析类时，会确定和导出更多应用于实现环境的精化后的设计类，如图3-8所示。

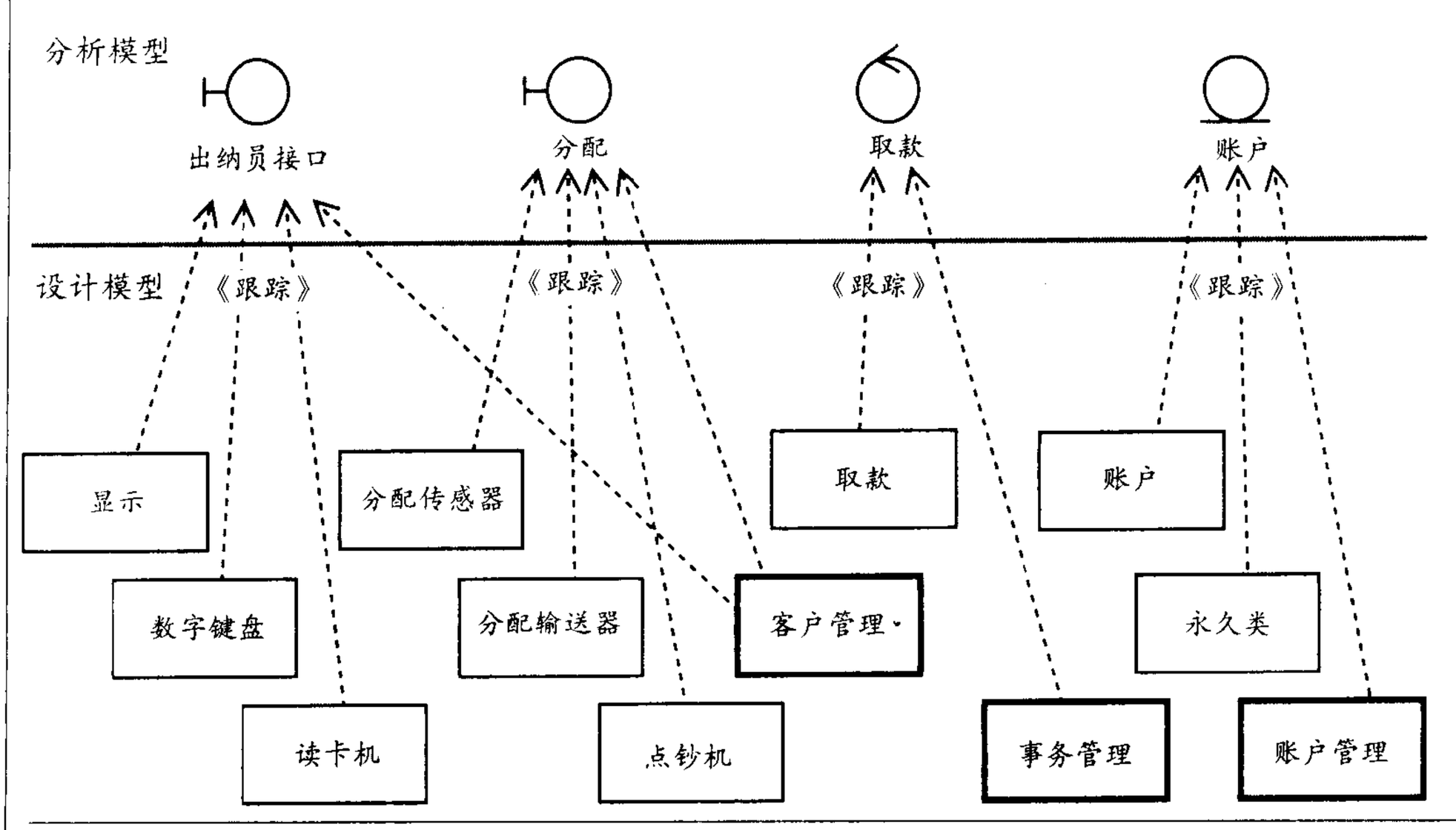


图3-8 设计模型中的设计类跟踪到分析模型中的分析类

例如，名为“出纳员接口”的分析类是由以下四个设计类：“显示”、“数字键盘”、“读卡

机”和“客户管理”（它是主动类，所以用粗边框表示；参见附录A）来设计的。

注意，图3-8中大多数设计类通常只跟踪一个分析类。这对于支持与应用领域有关的一个应用或一组应用而设计的设计类来说是正常的。所以，由分析模型所定义的系统结构在设计期间自然保留下来，尽管可能需要作一些折衷处理（例如，当“客户管理”参与“出纳员接口”和“分配”类的设计时）。此外，主动类（“客户管理”、“事务管理”和“账户管理”）表示系统分布时（将在4.5.3中进行讨论）由它组织其他（非主动）类的工作过程。

因此，设计模型中“取款”用况的实现需要说明该用况是如何根据对应的设计类来实现。图3-9描述了一个类图，它是用况实现的一部分。

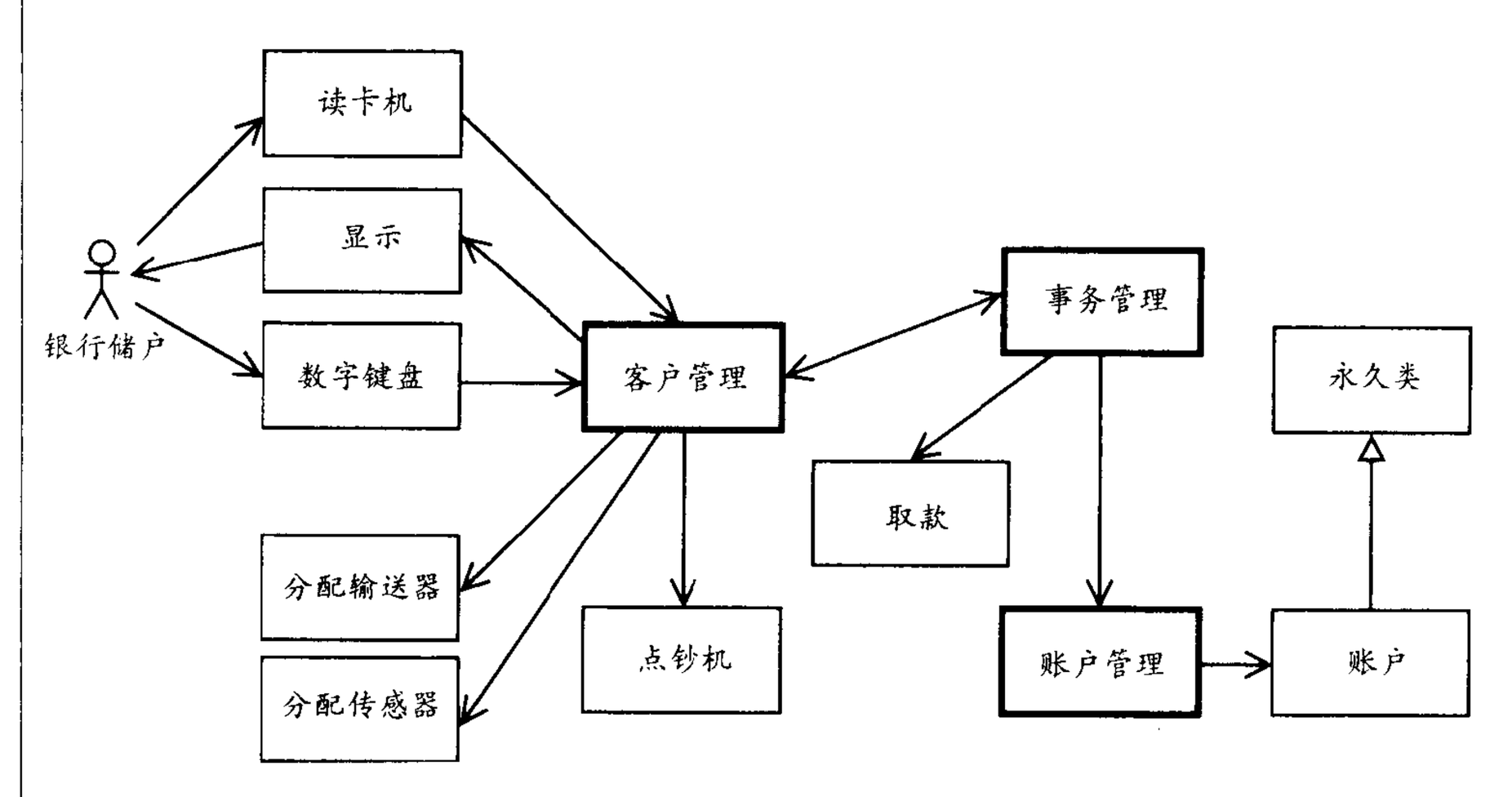


图3-9 设计模型中部分实现“取款”用况的类图。每个设计类均参与并充当用况实现中的角色

很显然，这张类图比分析模型中的类图（图3-5）展现了更多的细节。由于设计模型应该适应于实现环境，因此这些细节是需要的。

与分析（图3-6）中类似，在设计模型中，我们必须详细标识设计对象之间在实现用况时发生的交互。我们主要采用顺序图来建立设计过程中对象间交互的模型，如图3-10所示。

顺序图显示，当执行用况并在对象间传递消息时，顺序图开始于左上角，然后显示如何在对象间移动。从一个对象发出的消息触发接收对象作为工作重点，并执行对象所属类的一个操作。

将该顺序图与“分析副本”（即图3-6中的协作图）进行比较是一种有意义的练习。事实上，协作图中的前两个消息（“1：表明身份”和“2：请求取款”）是由图3-10的顺序图中的所有消息来设计的。把设计模型与分析模型进行比较就为我们提供了一个线索去理解设计模型中引入的复杂性和大量细节。

正如在协作图中所示，开发人员还可以使用文本对顺序图进行补充，解释设计对象如何进行交互来执行用况的事件流。

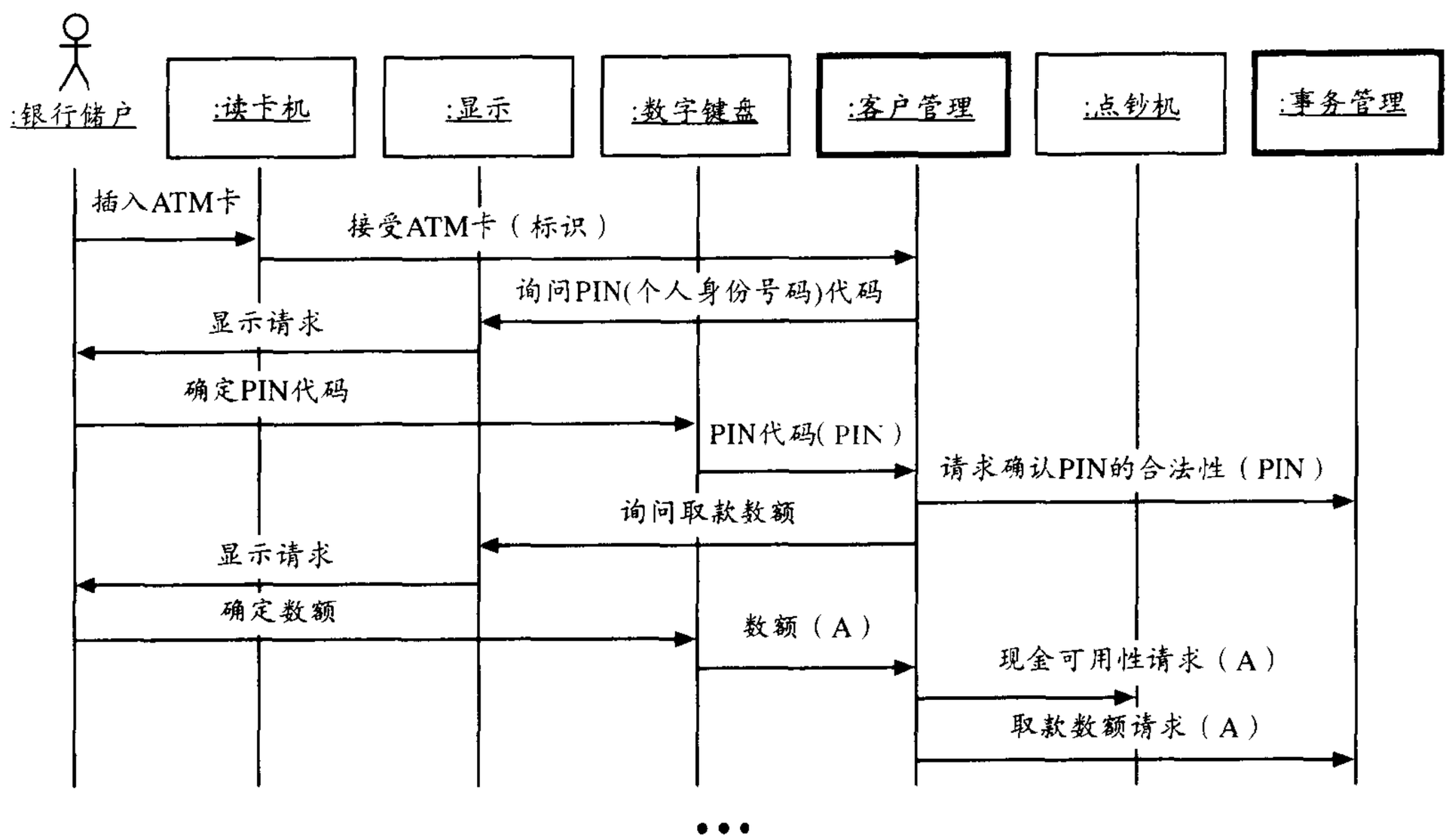


图3-10 设计模型中部分实现“取款”用况的顺序图

从这个例子中可以看到，设计模型似乎包含了很多类。因此，需要用一种方式来组织这些类。可以按照子系统来对类进行处理，下一节我们将把注意力转移到这个问题。

3.4.4 按子系统对类分组

对于具有成千上万个类的大系统，仅用类来实现用况是不可能的：这是因为系统太大，如果没有较高层次的分组将很难把握。这些类可以按照子系统进行分组。子系统是对类或其他子系统在语义上有益的分组。子系统提供和使用一个接口的集合。这些接口定义了子系统（参与者、其他子系统和类）的语境。

底层的子系统称为**服务子系统**（附录B；另见第9章），这是因为它们的类实现了一种服务（有关服务概念的更详细说明，参见8.4.5.1节）。服务子系统构成了一个可管理的任选的（或可能任选的）功能单位。从整体来讲，有可能在客户系统中只安装一个服务子系统。服务子系统也可以用来建立可能一起发生变化的一组类的模型。

子系统可以自底向上或自顶向下进行设计。当自底向上进行处理时，开发人员基于已确定的类来考虑和设计子系统；他们所设计的子系统将这些类封装到明确定义功能的单元内。如果开发人员选择自顶向下进行处理，构架设计师在确定其他类之前首先要确定高层的子系统和接口。然后开发人员处理各个子系统的任务，确定并设计子系统内的类。子系统是在第1章引入的，

在第4章和第9章还会进一步讨论。

举例：按照子系统对类分组

如图3-11所示，开发人员将类分组为三个子系统。这些子系统的选择依据是：所有提供用户界面的类归入一个子系统，所有与账户处理有关的类归入一个子系统，用况的专用（特定）类归入一个子系统。将所有用户界面类放入到“ATM接口”子系统中的好处在于，针对“事务管理”子系统，该接口子系统可以用其他提供相同功能的子系统来代替。备选的“ATM接口”子系统的用户界面可能有很不相同的实现，也许设计为能够接受或分发硬币而不是纸币。

“事务管理”子系统中的用况专用类（如“取款”类）都设计成一个单独的服务子系统。实际上，每个这样的服务子系统都包含多个类，但在我们所讨论的简单例子中没有表示出来。

图3-11还显示了子系统间的接口。一个圆代表一个接口。从类到接口的实线表示由该类提供此接口。从类到接口的带箭头的虚线表示该类使用此接口。为简单起见，没有表示由参与者提供或使用的接口；这里只是使用了一般的关联关系。

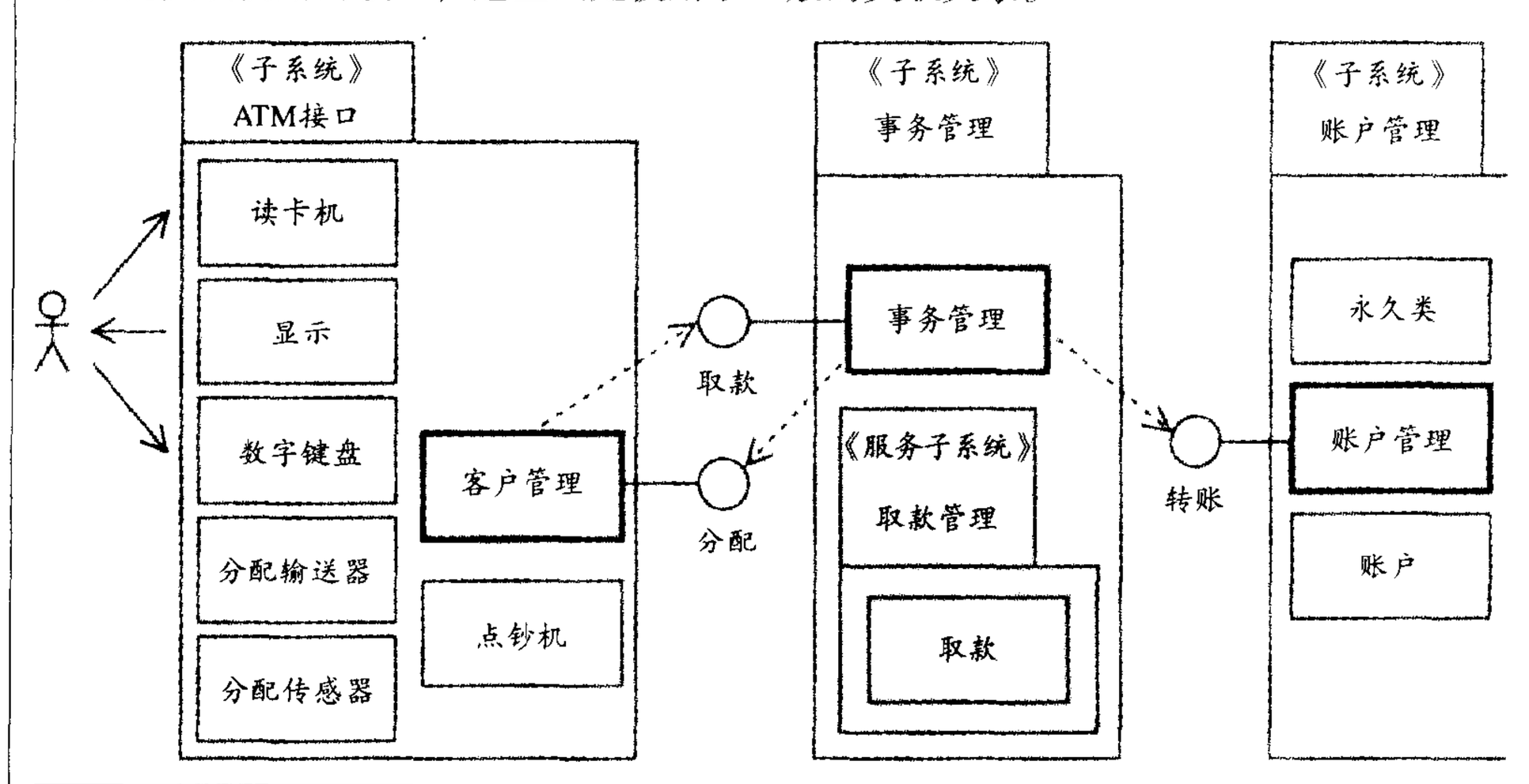


图3-11　这个简单的ATM例子划分为三个子系统和一个服务子系统（“事务管理”子系统内部灰色的部分）

“转账”接口定义了在账户间转账、从账户中取款以及向账户中存款的操作。“取款”接口定义了从某个账户中请求取款的操作。“分配”接口定义了由其他子系统（如“事务管理”子系统）使用的将货币分发给银行储户的操作。

3.4.5　根据设计模型建立实现模型

在实现工作流期间，我们需要开发能够产生可执行系统的制品：可执行的构件、文件构件（源代码，shell脚本等）、表构件（数据库元素）等。一个构件是系统中一个实际的且可替换的部分，它符合并提供接口集合的实现。实现模型由构件组成，包括所有的**可执行体**（附录A；另

见第10章），如ActiveX构件和JavaBeans构件，以及其他类型的构件。

举例：实现模型中的构件

图3-12描述了设计模型（在3.4.3节中介绍）中实现设计类的构件。

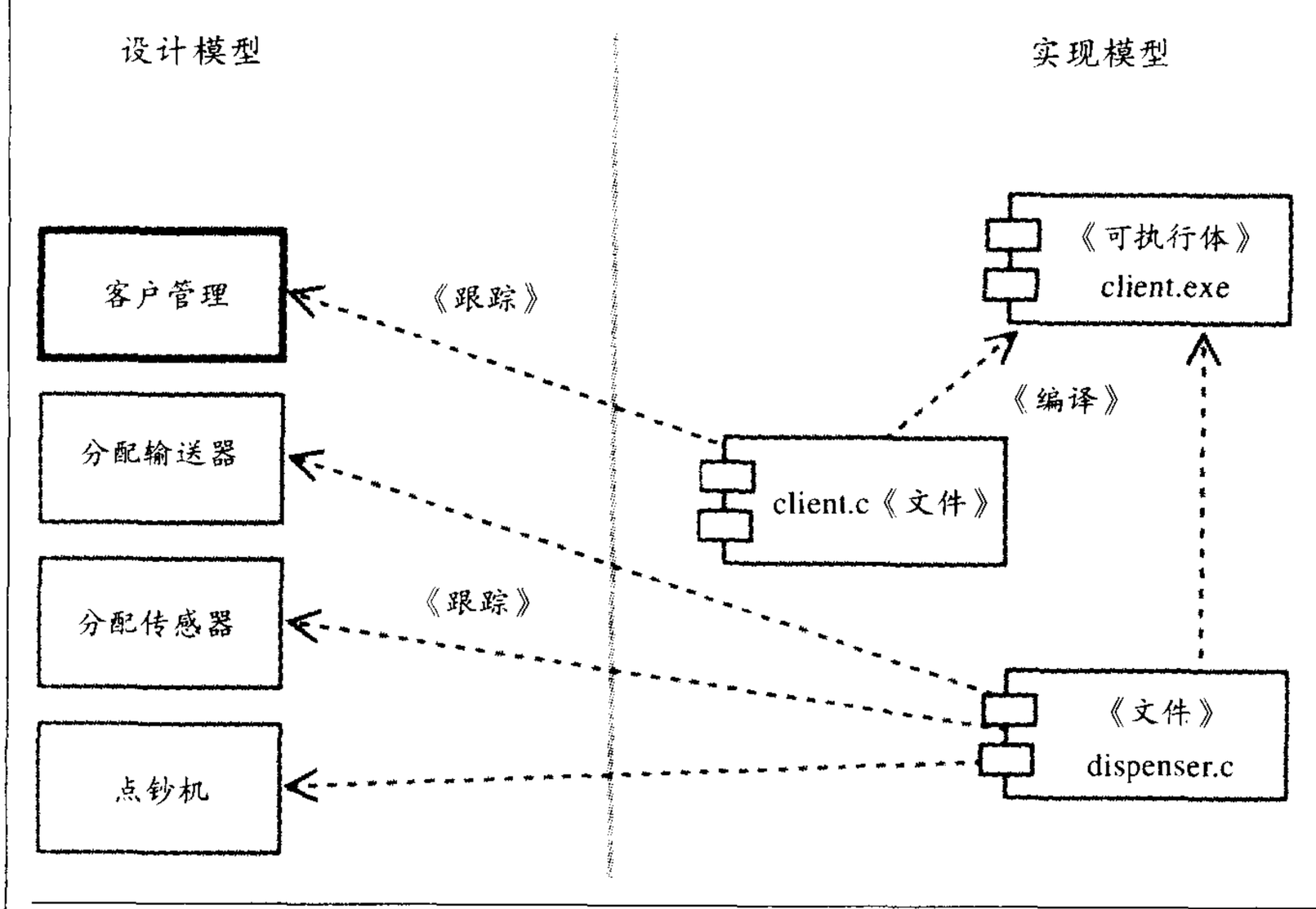


图3-12 实现设计类的构件

例如，“dispenser.c”文件构件包含三个类（“分配输送器”，“分配传感器”和“点钞机”）的源代码（因此是这三个类的具体实现）。该文件构件通过编译并与“client.c”文件构件连接成为可执行的“client.exe”构件。

构件设定了一个由其接口定义的构架语境。它也是可替换的，表明开发人员可以将一个构件用另一个可能更好的构件代替，只要这个新的构件提供或要求相同的接口即可。通常有一种简明的方式将设计模型中的一个服务子系统实现为可分配到实施模型中节点上的若干构件。如果服务子系统总是被分配到实施模型中同一类节点上，那么它可以由一个构件来实现。如果服务子系统被分配到多个节点上时，可能要根据节点的类型将服务子系统分为多个部分（通常分为一些类）。在这种情况下，该服务子系统的每部分都将作为一个构件来实现。

举例：作为构件实现的服务子系统

假如对于ATM的例子选择了**客户机/服务器**（参见9.5.1节）解决方案，那么就需要有“取款管理”服务子系统的一部分（图3-11），它包含部署在客户机和服务器端上的“取款”类。所以，“取款管理”服务子系统就应该实现“在客户机端取款”和“在服务器端取款”两个构件。

如果用面向对象的程序设计语言实现构件，类的实现也很简便。每个设计类对应一个实现中

的类，如C++类或Java类。根据具体程序设计语言的约定，每个文件构件可以实现多个这样的类。

但是实现工作不只是开发源代码以建立一个可运行的系统。负责实现构件的开发人员在将它交付进行集成和系统测试之前还要完成单元测试。

3.5 用况的测试

在测试期间，需要验证系统是否正确实现其规格说明。建立一个包括**测试用例**和**测试规程**（附录C；另见第11章）的测试模型，并执行测试规程以确保系统按照预期的方案工作。测试用例是测试输入、运行条件和针对具体目标制定的预期结果的集合，例如按照一条特定的路径完成一个用况或验证与所规定的需求的一致性。测试规程是关于如何对具体的测试用例进行设置、运行和评估结果的规格说明。测试规程也可以从用况中得到。对发现的缺陷要进行分析并定位问题所在。然后对这些问题按照其重要性排定先后次序，并进行相应的改正工作。

在3.3节中我们首先捕获用况，接着在3.4节中分析、设计并实现用来实现用况的系统。现在介绍如何测试是否正确实现了用况。在某种意义上，这并不是什么新的东西。开发人员一直在测试用况，甚至在术语用况出现前就是如此。以对用户有意义的方式来测试系统一直是测试系统功能的实用方法。然而，从另一方面来说，这是一种新的方法。新在我们开始设计系统之前就确定了一些测试用例（即需求工作流期间的用况），并确保我们的设计真正实现了这些用况。

举例：根据用况确定测试用例

图3-13中给出了一个测试用例“取款—基本流”，图中说明如何测试“取款”用况的基本流。

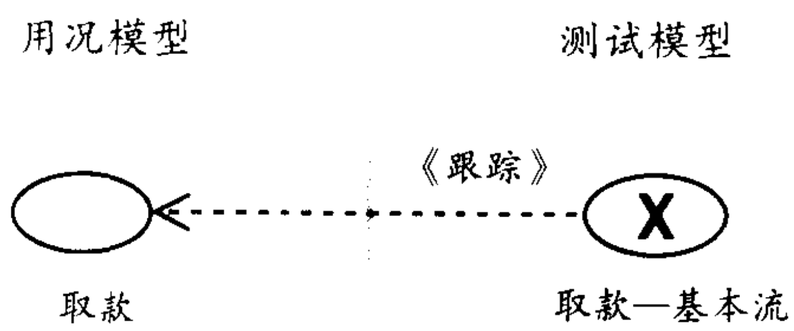


图3-13　测试模型中的测试用例说明如何测试用况模型中的用况（“取款”）

注意，在这里引入了一个新的、测试用例的构造型，即内部带有“×”的用况符号。这样做是为了在图中（参见第11章）用它表示测试用例。

测试用例规定了输入、预期结果以及有关验证“取款”用况基本流的其他条件。

输入：

- 银行储户的账户12-121-1211上余额为350美元。
- 银行储户正确表明本人的身份。
- 银行储户请求从账户12-121-1211上取款200美元。
- ATM中有足够的货币（至少为200美元）。

结果：
• 银行储户的账户12-121-1211上余额减为150美元。
• 银行储户从ATM获得200美元现金。
条件：
在该测试用例运行期间，不允许其他用况（实例）对账户12-121-1211进行访问。

注意，该测试用例是基于3.3.3节中列举的“取款”用况的说明而建立的。通过尽早确定用况可以提前规划测试活动并考虑有用的测试用例。在设计阶段，当我们了解了更多的有关系统如何执行这些用况的细节时，便可以进一步细化相应的测试用例。实际上，一些工具可以根据设计模型生成测试用例，但运行测试所需要的测试数据仍然必须人工输入。

用况测试可以从参与者的角度来执行，此时将系统作为一个黑盒看待；或者从设计的角度执行，此时所构造的测试用例用来验证用况实现中涉及到的类的实例能够完成它们应该做的工作。一旦需求基本稳定，便可以对黑盒测试进行标识、详细描述和规划。

还有其他类型的测试，如系统测试、验收测试和用户文档测试。在第11章中将对测试进行进一步的介绍。

3.6 小结

用况驱动过程。在需求工作流期间，开发人员可以将需求表示为用况，接着项目经理便可以根据开发人员确定的用况来规划项目。在分析和设计期间，开发人员按照类或子系统来创建用况实现。然后开发人员实现构件，并将构件集成到每次实现用况集合的增量结果中。最后，测试人员验证系统是否正确实现了用户的用况。换句话说，用况将所有开发活动连接成为一个整体，并驱动了开发过程，这也许是用况驱动方法最大的好处所在。

用况对一个项目提供了很多益处。然而，它们不是万能的。下一章我们将讨论统一过程的另一个很重要的方面——以构架为中心。

3.7 参考资料

[1] Ivar Jacobson, “Object-oriented development in an industrial environment,” *Proceedings of OOPSLA'87*, Special issue of *SIGPLAN Notices* 22(12): 183–191, December 1987.

[2] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Övergaard, *Object-Oriented Software Engineering: A Use-Case Driven Approach*, Reading, MA: Addison-Wesley, 1992 (Revised fourth printing, 1993).

[3] Ivar Jacobson, “Basic use case modeling,” *ROAD* 1(2), July–August 1994.

[4] Ivar Jacobson, “Basic Use Case Modeling (continued),” *ROAD* 1 (3), September–October 1994.

[5] Ivar Jacobson, "Use cases and objects," *ROAD* 1(4), November–December 1994.

[6] Ivar Jacobson and Magnus Christerson, "A growing consensus on use cases," *Journal of Object-Oriented Programming*, March–April 1995.

[7] Ivar Jacobson and Sten Jacobson, "Use-case engineering: Unlocking the power," *Object Magazine*, October 1996.

[8] Karl Wieger, "Use cases: Listening to the customer's voice," *Software Development*, March 1997.

[9] E. Ecklund, L. Delcambre, and M. Freiling, "Change cases: Use cases that identify future requirements," *Proceedings, Conference on Object-Oriented Programming Systems, Languages, & Applications (OOPSLA'96)*, ACM, 1996, pp. 342–358.

[10] Alistair Cockburn, "Structuring use cases with goals," *Report on Analysis & Design* (ROAD), 1997.

[11] Geri Schneider and Jason Winters, *Applying Use Cases: A Practical Approach*, Reading, MA: Addison-Wesley, 1998.

[12] OMG Unified Modeling Language Specification. Object Management Group, Framingham, MA, 1998. Internet: www.omg.org.

第4章
以构架为中心的过程

在第3章中，为了简化问题，曾指出用况本身就能够表明经过需求、分析、设计、实现及测试而获得一个系统的方法。然而，软件开发并不仅仅是盲目地依赖用况驱动工作流就能完成的。

只依靠用况是不够的。要得到一个可用的系统还需要考虑更多的因素，这里“更多的”这个词就是指构架。可以把系统的构架看作是所有工作人员（即开发人员和其他项目相关人员）必须达成或至少能够接受的共同目标。构架为我们提供了整个系统的清晰的视角，这对控制系统的开发是必要的。

我们需要一个构架来描述对我们最为重要的模型元素。如何判断哪些元素是重要的呢？其重要性就在于能够指导我们系统的开发工作，不仅在这个周期而是在整个生命周期。这些对构架重要的模型元素包括子系统、依赖、接口、协作、节点和**主动类**（附录A；另见9.3.2节）。它们描述了系统中的基础部分，据此可以更为有效地理解、开发并改良这个系统。

可以把开发一个软件项目和建造一座能停放一辆汽车的车库进行比较。首先，建造者需要考虑用户希望怎样使用该车库。其中的一个用况需要遮蔽汽车，即要能把车驶入、停放在里面，以后又能再把它开出来。用户是否还想把车库用作其他用途呢？假如他还希望把车库用作一个家用工作间，那么建造者就得想到采光要求——要设计几扇窗户和一盏电灯。许多工具都需要用电，因此还要设计几个电源插座并提供足够的电量。从某种意义上讲，建造者是在创建一个简单的构架。这一切他可用大脑去思考，因为他以前曾见过车库，也见过工作台，知道一个人在工作台边工作需要什么。建造者的方法不是盲目的，因为他已经熟悉了一个典型的小车库的构架。他只要把这些部分组织起来，以适合车库的具体需要就行了。如果建造者从未见过车库而盲目地关注车库的用途，可能会造出一座很怪的建筑。所以，他不仅要考虑车库的功能，还要考虑它的外部表现形式。

要建造一所具有10个房间的房子、一座教堂、一家购物中心或一幢摩天大楼，情况是很不一样的。现在有许多建造这类大型建筑的方法，需要一组建筑设计师来设计它们。设计组成员必须彼此了解构架的进度，也就是说他们需要把自己的工作用其他组员能够明白的形式记录下来。还要用一种非专业人员（业主、用户和其他项目相关人员）可以理解的方式表示出来。最后，还得通过施工图纸将构架告知建筑商和建材供应商。

类似地，大部分软件系统（或软件系统加上支持它的硬件）的开发都要预先考虑周全，并以后续的开发人员和其他项目相关人员都可用的形式记录这些想法。而且，这些想法（这个构架）并不是一开始就完全成熟：构架设计师在初始和细化阶段经过几次迭代才能逐渐使其完善。实际上，细化阶段的主要目标就是以可执行的**构架基线**（附录C）的形式建立一个可靠的构架。

这样，在进入构造阶段时，就具备了构造整个系统的坚实基础。

4.1 构架概述

我们需要构架。不错，但“软件系统的构架”到底是指什么呢？当人们探究软件构架的概念时，不禁会想起盲人摸象的寓言故事。大象只是每个盲人偶然间摸到的一条大蛇（象鼻子）、一根粗绳（象尾巴）或一棵小树（象腿）。与此类似，构架的概念，至少当简化为一句话的定义时，在作者头脑中所想的就像寓言中所说的一样。

我们仍把软件构架比作房屋建筑。在客户看来，一座建筑通常是一个单一的单元。建筑设计师可能因此发现制作一座建筑的比例模型加上几幅不同视角的建筑图纸很有用处。这些图纸通常都是简图，却能让客户看懂。

然而，建筑物的修建在施工阶段还需要其他工种的工作者，如木工、小工、泥瓦匠、吊顶工、水管工、电工等。他们都需要更为详细和专门的建筑施工图纸，而且所有这些图纸之间必须保持一致。例如，通风管和水管就不能标定在同一位置。建筑设计师的职责就是创建整个建筑物设计中最重要的方面。于是，建筑设计师绘制出一套描述建筑物方方面面的建筑图纸，如挖掘工作中的地基。结构工程师决定支撑楼架柱子的尺寸，地基承受墙、地面和房顶的重量。这种结构包括电梯、水、电、空调、卫生等系统。然而，这些建筑图纸对于建筑工人的工作来说还不够详尽。为此，很多专业领域的建筑制图人员绘制和起草能够反映有关材料选择、通风子系统、电力子系统、供水子系统等细节的图纸和详细说明。建筑设计师对工程全面负责，而其他各类设计人员负责补充细节问题。一般情况下，建筑设计师是把建筑的各个方面集成为一个整体的专家，但不是其中每个领域的专家。当绘制完所有图纸以后，建筑图纸只包括了建筑物最重要的部分。建筑图纸是所有其他图纸的不同视图，但它们和其他图纸是一致的。

在施工过程中，不同的工作者使用建筑图纸——详细图纸的不同视图——以获得对建筑物的全面了解，但他们要靠详细的施工图纸才能完成他们的工作。

就像建筑物一样，软件系统是一个单一的实体，但软件构架设计师和开发人员发现从不同视角展示该系统有助于更好地理解其设计。这些视角被解释为系统的模型**视图**（附录A和附录C），视图合在一起展示了构架。

软件构架包括对以下四个方面所作的决策：

- 软件系统的组织。
- 构成系统的结构元素和各元素之间的接口，以及由元素间各种协作所规定的各元素的行为。
- 结构元素和行为元素合成为逐渐增大的子系统。
- 指导这种组织的**构架风格**（附录C）：元素以及它们的接口、协作和组合。

然而，软件构架不只涉及结构和行为，还涉及到使用、功能、性能、柔性、重用、可理解性、经济性和技术约束以及折衷方案、美学等。

在4.4节中，将用更为具体的术语讨论软件构架的概念，并说明如何使用统一过程来表示它。这里先简单解释一下什么是**构架描述**（附录C）。前面已经介绍过构架可以表示为多种模型的视图：用况模型视图、分析模型视图、设计模型视图等。这组视图与[3]中所讨论的4+1视图是吻合的。因为一个模型的视图是对该模型的一种抽取或是对它的一个切片，例如，用况模型视图看起来就像是用况模型本身，它包括对构架重要的参与者和用况。同样，设计模型的构架视图看起来就像是设计模型，但它只包含用来实现构架的重要用况的设计元素（参见12.6.2节）。

构架描述没有什么不可思议之处，它看起来像是带有所有系统模型（不同模型之间有些区别，以后再提及）的一个完整的系统描述，但它比较小。它有多小呢？构架描述没有绝对的规模大小，但根据我们的经验，对于有很多类的大型系统，构架描述大约应该在50页～100页。这个范围适用于单一的**应用系统**（附录C）；**应用系统套件**（附录C）的构架描述要比这个范围更大一些。

4.2 为什么需要构架？

一个大型、复杂的软件系统需要一个构架设计师，以便开发人员可以向着一个共同的目标努力。软件系统很难想象，因为它并不存在于我们的三维世界里。从某些方面来讲，它通常是无先例可循的或独一无二的。它经常采用未经证实的技术或各种技术的新颖组合，并把现有的技术推向极至。而且，在构造时还必须适应将来一系列的巨大变化。随着系统的日益复杂，“设计问题超过了计算的算法和数据结构：设计和确定整个系统的结构便成了一类新的问题”[1]。

此外，经常存在用现行系统来完成规划中系统某些功能的情况。在没有文档或文档极少的情况下，要弄清楚系统能做什么以及开发人员可以重用哪些遗留代码，更增加了开发的复杂性。

我们需要一个构架是基于以下原因：

- 理解系统
- 组织开发
- 鼓励重用
- 进化系统

4.2.1 理解系统

对于一个从事系统开发的组织，系统必须为所有相关人员所理解。要使现代的系统为人们所理解是个很大的挑战，其原因很多：

- 它们包含复杂的行为。
- 它们在复杂的环境中运作。
- 它们使用的技术很复杂。
- 它们经常把分布式计算、商用产品和平台（例如操作系统和数据库管理系统）以及可重用构件和框架结合在一起。

- 它们必须满足个人和组织的需求。
- 在某些情况下，系统过于庞大以致于管理层不得不把开发工作分为在地理上经常处于不同地点的多个项目，也增加了协调的难度。

而且，这些因素也在不断变化。所有这一切形成了一种潜在的难以理解的格局。

以构架为中心进行开发（附录C），可以防止出现这种无法理解的现象。因此，对于构架描述的第一个需求就是：它必须使开发人员、管理人员、客户以及其他项目相关人员能够详细理解所需要做的工作，以利于他们参与系统的开发。在第3章曾提到的模型和图会对此有所帮助，可用它们来描述构架。随着人们对于UML的日益熟悉，会发现使用UML来对构架进行建模，可以更容易掌握构架的概念。

4.2.2 组织开发

软件项目组织越庞大，开发人员协调工作所付出的开销也就越大。当项目分散在不同地点时，这种交流的开销也会增加。通过将系统划分为带有明确定义接口的子系统，并让一个开发组或个人负责每个子系统，无论他们是在同一幢大楼里还是在不同的地域，构架设计师都可以减少负责不同子系统的开发组之间的交流的工作量。一个“好的”构架应该明确定义这些接口，尽可能减少子系统间的通信。一个良好定义的接口，可以有效地向双方的开发人员“传达”他们需要了解的对方小组正在进行的工作。

稳定的接口允许该接口双方的软件独立地进化。合适的构架和设计**模式**（附录C）有助于发现子系统间恰当的接口。例如，边界控制实体模式（参见3.4.1节）有助于把用况行为、边界类和共性类所关心的不同问题分离出来。

4.2.3 鼓励重用

让我们打个比方来解释构架对重用的重要性。管线产业很早就已经标准化。管线承包商从标准化的零部件“获益匪浅”。管道工只需从总是可以组装在一起的标准化零部件中选取零部件，而不必费力去搭配来自于各地的不同尺寸的“新型”零部件。

像管道工一样，“提倡重用”的开发人员知道**问题领域**（附录C），而且知道构架所确定的哪些构件是合适的。开发人员了解如何装配这些构件来满足系统需求和实现用况模型。当能够得到可重用构件时，就使用它们。就像标准化的管道设施一样，可重用的软件构件经过设计和测试以便能够组合使用，这样构造软件就能够节省时间和降低成本，其结果是可以预测的。就像在管线产业中推行标准化用了几个世纪的时间一样，软件构件标准化也需要积累经验；但是，我们预计大量的“构件化产品”将在今后几年内出现。实际上，此项工作开展得不错。

软件产业要达到如很多硬件领域已经达到的标准化水平，好的构架和明确的接口是实现这一目标的关键步骤。好的构架为开发人员提供了可以在其上开展工作的稳定的骨架。构架设计师的任务就是定义这个骨架和开发人员使用的可重用子系统。通过精心设计得到可重用的子系统，并使它们可以装配起来使用[2]。一个好的构架有助于开发人员知道在哪里能有效地寻找到可重用的元素以及发现合适的可重用构件。UML将会加速构件化产品的进程，因为标准建模语

言是构造特定领域可重用构件的先决条件。

4.2.4 进化系统

有一件事情很确定，那就是任何有相当规模的系统都需要不断进化。甚至它处在开发过程中就需要这种进化。在投入使用后，变化着的环境也会要求对其进一步完善。在这两个过程中，系统应该易于变更；也就是说，开发人员应该可以改变部分设计和实现，而不必担心这种改变会对整个系统产生非期望的效果。大多数情况下，他们应该可以在系统中实现新功能（即用况），而不会对现有的设计和实现造成太大的影响。换句话说，系统本身应该对变化具有一定的柔性（或称容变能力）。另一种表述这一目标的方式是说，系统应该能够适度地进化。相反，构架拙劣的系统经常会随着时间的推移和很多补丁程序的使用而出现功能退化，直至最后无法有效地进行更新。

举例：爱立信AXE系统——关于构架的重要性

爱立信AXE电信交换系统最初是在20世纪70年代早期运用我们的构架原则的初期版本开发的。软件构架描述是一个很重要的制品，它指导了系统生命周期内的全部开发工作。当时的构架得到了多个现在已并入到统一过程中的原则的指导。

其中的一个原则就是功能模块化。将类和等价的设计元素分组为客户可任意选择的功能块或服务子系统（即使这些元素是发给所有客户的）。服务子系统具有很强的内聚性。对系统的变更通常只局限于一个服务子系统，很少影响到多个服务子系统。

另一个原则是把对接口的设计和对服务子系统的设计分开。目标是实现“可插拔的”设计，其中的几个服务子系统支持相同的接口。这样，便可以实现服务子系统之间的互换而无需改变服务子系统的客户（只依赖于接口，而不依赖于服务子系统的实际代码）。

第三个原则是直接将设计中的服务子系统映射到实现中的一个或多个构件上。服务子系统中的构件可分布在不同的可计算节点上。每一个运行服务子系统的处理节点恰好有一个构件。于是，如果服务子系统在中央计算机（比如一台服务器）上运行，那么，针对这个子系统就恰好有一个构件。如果服务子系统既要在客户机上又要在服务器上执行，就应该构造两个构件。这个原则简化了软件在出现不同安装时的变化处理。

还有一个原则是，服务子系统间耦合度要低。信号是服务子系统间进行通信的唯一方式。因为信号是异步的（即无等待发送语义），所以它们既支持封装也支持分布。

由于最初和后续的开发都是由设计良好的构架指导，因此AXE系统一直到现在还在使用，有100多个客户和几千个安装点。经过更新，预计它还可以使用几十年。

4.3 用况和构架

我们曾经暗示过，用况和构架之间存在着某些相互作用。第3章大致介绍了如何开发能为用户提供恰当用况的系统。如果系统提供了恰当的用况（具有高效能、高质量和高可用性），那么

用户可用该系统来完成自己的任务。但如何实现这一目标呢？正如前面已经提到的，就是要构造一个允许无论现在还是将来都能有效地实现用况的构架。

首先来看一下什么影响构架（如图4-1所示），然后再看看什么影响用况，以便能弄清楚这种相互作用是如何产生的。

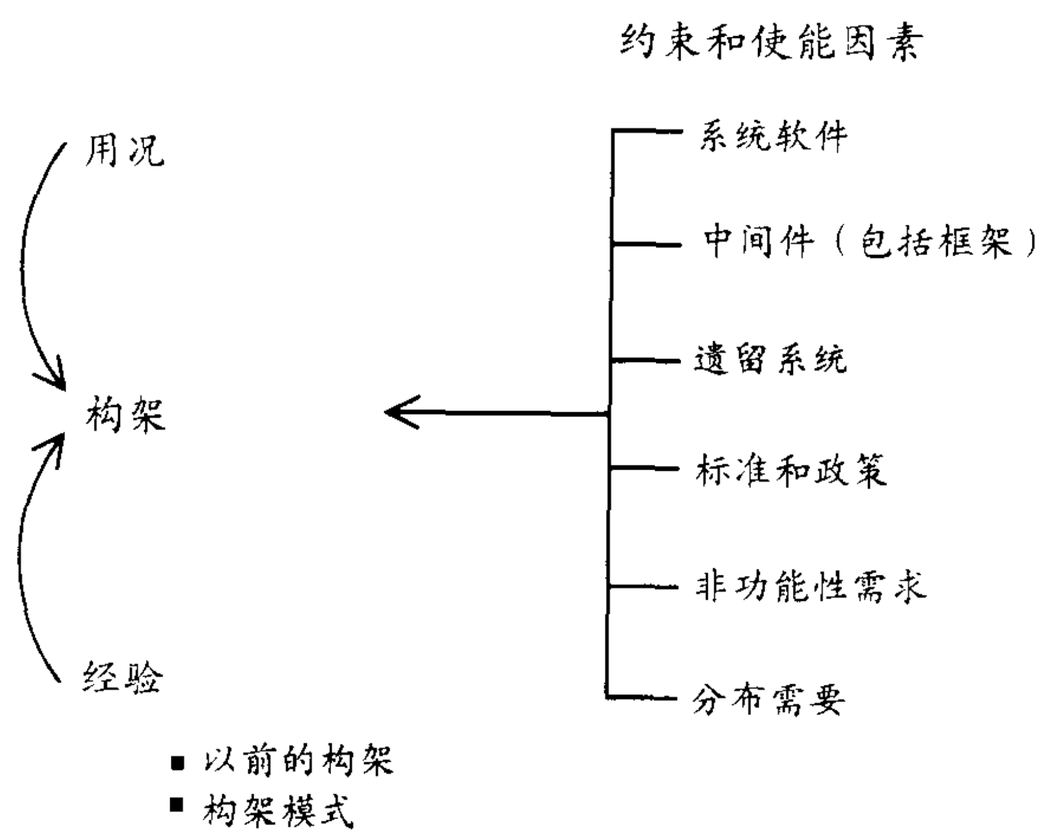


图4-1 不同种类的需求和产品影响构架，而不仅仅是用况影响构架。先前工作中的经验和确认为构架模式的结构也有助于设计构架

前面曾提到过，构架受我们希望系统所支持的用况的影响；用况是构架的驱动力。归根结底，在实现用况时需要一个合适的构架。在最初的迭代中，选择几个最有助于设计构架的用况。这些对于构架十分重要的用况是指在最近和将来的发布中用户最需要的那些用况。

然而，构架不仅会受到对构架重要的用况的影响，还会受到下列因素的影响：

- 希望软件产品构造在哪些系统上，如操作系统或具体的关系数据库管理系统。
- 希望使用哪些中间件（**中间件层**；附录C）产品。例如，需要选择一个对象请求代理（ORB）来创建图形用户界面，这里，ORB是一种将消息透明地调度和转发给分布在异构环境中对象的机制[6]，或是一个与平台无关的框架，即一个“预制”的子系统。
- 希望在系统中使用哪些遗留系统。通过在构架中使用遗留系统，如一个现行的银行系统，可以重用许多现有的功能，但还需要调整构架来与“老”产品相互配合。
- 需要适应哪些政策和公司标准。例如，可选择OMG的接口定义语言（IDL）[7]来确定类的所有接口，或选择远程通信规范TMN[8]来说明系统中的对象。
- 通用的非功能性需求（即非用况专用的需求），例如可用性、恢复时间或内存使用等方面的需求。
- 分布需要说明系统是如何分布的，也许可以通过客户机/服务器构架来分布系统。

可以把图4-1中右侧的几项看作是能使我们以某种方式来建立构架的约束和使能因素。

构架是在细化阶段的迭代过程中被创建的。以下是一个简化的、不太成熟的方法，是一种思维模型。开始先确定构架的高层设计，例如一个**分层**构架（附录C）。然后在第一次迭代的几次**构造**（附录C；另见第10章）中逐步确立该构架。

在第一次构造中，处理的是**通用应用**部分（附录C），它们对于所讨论的领域是通用的，而不是专用于计划开发的系统，即并非专用于我们选择使用的系统软件（系统软件层；附录C；另见9.5.1.2.2节）、中间件、遗留系统、标准和政策。我们要决定在实施模型中包括哪些节点以及这些节点应该如何进行交互；要决定如何处理一般的非功能性需求，如可用性需求。在第一次构造中，对应用有一个大致的了解也就足够了。

在第二次构造中，处理构架中的**专用应用**部分（专用应用层；附录C）。首先选取一个与构架相关的用况集合，捕获需求，并对它们进行分析、设计、实现和测试，最后得到一个新的、用构架实现的子系统，来支持所选的用况。在第一次构造中实现的对构架至关重要的构件可能也要进行某些改变（当时我们没有从用况的角度考虑）。为了实现用况，需要对变化的构件和新的构件进行开发，使构架更适合于所选的用况。然后再进行下一次构造，依次下去，直到完成这次迭代过程。如果这次终结刚好出现在细化阶段的最后，则该构架也就应该稳定了。

当建立了一个稳定的构架后，就可以通过在构造阶段实现其余的用况来实现系统的全部功能。在开发构造阶段中实现的用况时主要是以客户和用户需求作为输入（参见图4-2）。但这些用况也会受到细化阶段所确定的构架的影响。

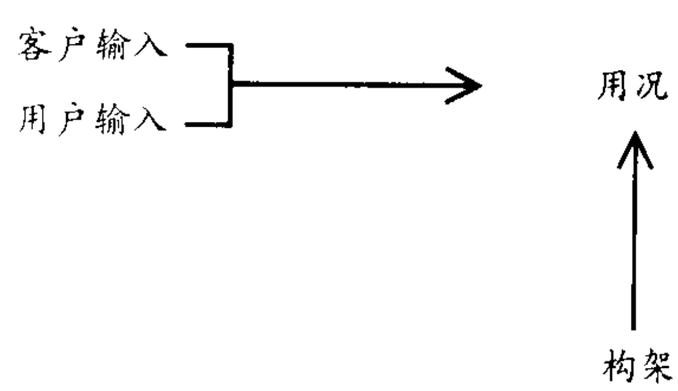


图4-2 基于客户和用户的输入来开发用况。然而，用况也受到已经存在的构架的影响

在捕获新的用况时，可以利用对已经存在的构架的知识来更好地完成我们的工作。在评估每个所选用况的价值和成本时，也是根据现存的构架进行的。一些用况很容易实现，而另一些用况实现起来则较为困难。

举例：使用况适应于已经存在的构架

客户希望获得一种监控处理器负载的功能。可以将这个要求说明为测量计算机高优先级负载的用况。要实现这个用况，需要对现在所用的实时操作系统进行一些改动。而开发组则建议，通过一个独立的外部设备呼叫系统并测量应答时间来实现所需的功能。于是，客户得到了一种更为可靠的方法，而开发组也可避免对关键的底层构架进行改动。

我们可以与客户进行协商，以确定是否可以改变用况，以使用况和设计结果与已有的构架更加一致，从而使实现更为简化。这种一致性意味着必须考虑现存的子系统、接口、用况、用况实现、类等。通过使用况与构架保持一致，便可以利用现存的资源有效地创建新的用况、子系统和类。

所以，一方面构架受到我们希望系统支持的用况的影响。用况驱动构架。另一方面，当我们将需求捕获为用况时，可以利用对构架的知识来更好地完成任务。构架指导用况（如图4-3所示）。

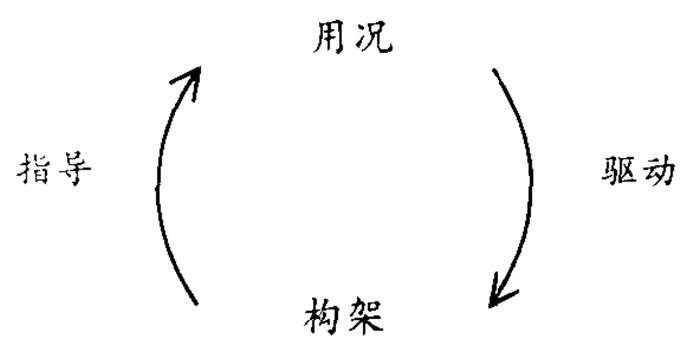


图4-3　用况驱动构架的开发，构架指导用况的实现

用况和构架哪一个最先出现呢？这是一个典型的“鸡与蛋”的问题。这种问题最好通过迭代来解决。首先，在很好地了解**领域范围**（附录C）的基础上建立一个临时的构架，但不考虑具体的用况。接着选取几个重要的用况，并进一步使构架能够支持这些用况。然后再选取更多的用况并建立更加完善的构架，依此类推。在每一次迭代中，都选取并实现一组用况来确认构架，如果必要，对构架进行改进。随着每次迭代的进行，我们还在所选用况的基础上进一步实现构架的专用应用部分。因此，在通过迭代实现整个系统时，用况有助于我们逐渐完善构架。这就是用况驱动开发的一种益处。第5章将接着讨论这种迭代方法。

总之，一个好的构架允许我们无论现在还是将来都能有效地补充适当的用况

4.4　建立构架的步骤

构架主要是在细化阶段的迭代过程中发展起来的。每次迭代的进行都像第3章中所概括的那样，从需求开始，然后是分析、设计、实现和测试，但侧重于与构架相关的用况和其他需求。细化阶段的最终结果是一条构架基线——一个只有很少软件“肌肉”的系统骨架。

哪些用况对构架更为重要呢？12.6节中将详细阐述这一问题。现在这样解释就可以了：对构架重要的用况就是那些有助于降低最重大风险的用况，那些对系统的用户最为重要的用况，那些有助于实现所有重要功能而不遗留任何问题的用况。实现、集成和测试构架基线使构架设计师及其他工作人员确信，到目前为止构架确实是可操作的。这是无法通过“纸上”的分析和设计来证实的。可操作的构架基线向那些能提出反馈的工作人员给出了一种能工作的证明。

4.4.1　构架基线是一个“小的、皮包骨架的”系统

在细化阶段最后，我们从构架的角度开发出了代表最重要的用况及其实现的系统模型。正如4.3节“用况和构架”中所讨论的，我们也确定了要遵守哪些标准，要使用哪些系统软件和中间件，要重用哪些遗留系统，以及有哪些分布需要。这样，我们就获得了用况模型、分析模型、设计模型以及其他模型的早期版本。这些模型的集合（参见图4-4）就是构架基线；它是一个

"小的、皮包骨架的"系统[⊖]。它包含构造阶段末期性能完善的系统中所有模型的各种版本，包括与最终系统相同的子系统骨架、构件和节点，但不是所有的"肌肉部分"都存在了。然而，它们确实具有行为，而且是可执行的代码。"皮包骨架的"系统逐渐进化为性能完善的系统，有可能要对结构和行为做一些细微的改变。改变较小的原因是，在细化阶段最后已经定义了一个稳定的构架；如果没有达到这个目标，细化阶段必须进行下去直到达到该目标为止。

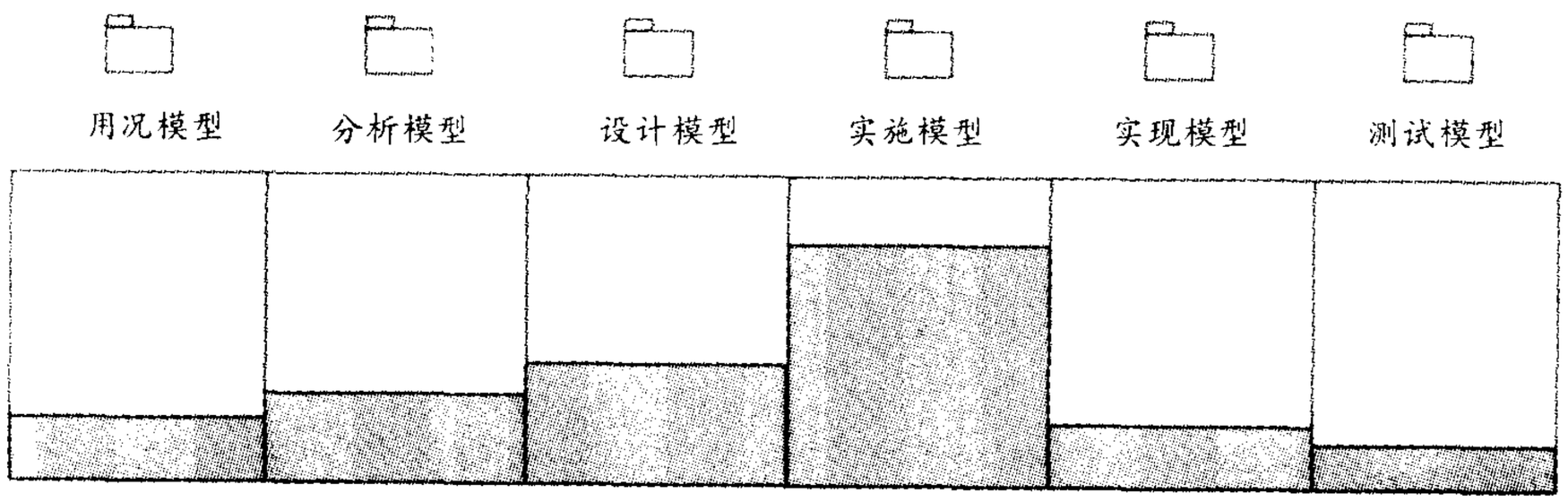


图4-4 构架基线是以描述构架为主的系统内部发布的版本

在图4-4中，每种模型的阴影部分表示在细化阶段最后已经开发出的模型版本，即属于构架基线的那部分模型版本。整个矩形（带阴影和不带阴影的部分）表示在移交阶段末期开发出的模型版本，即代表客户版本的基线（读者不要从图4-4中显示的阴影部分的大小而得出诸多结论，它们在这里只是用作解释之用）。在构架基线和客户版本基线中间还有几个代表模型新版本的**内部版本**（附录C）的基线。可以把这些新版本解释为以构架基线为起点逐渐累加增量的结果。每个模型的新版本都是由前一个版本进化得来的。当然，图4-4中的不同模型不是相互独立地进行开发。例如，用况模型的每个用况都对应着分析模型和设计模型中的某个用况实现，或测试模型中的测试用例。过程和节点结构必须满足用况需要的性能。（否则，就需要改变用况模型或实施模型，也许通过改变主动类分配到节点的方式可以获得更好的性能。实施或设计模型的这种变化可能要求改变用况模型。）就像2.3.7节中所讲述的那样，不同模型中的模型元素是通过《跟踪》依赖来建立相互之间的关联。

然而，构架基线（即细化阶段末期系统的内部版本）不仅仅靠模型制品来表示，它还包括构架描述。这个描述实际上是同时建立的，甚至经常比产生构架基线中模型版本的活动更早。构架描述的作用是在系统的整个生命周期内指导整个开发组——不只用于当前周期的各次迭代，而且还用于所有以后的周期。它是开发人员目前和将来都要遵循的标准。既然构架应该是稳定的，那么标准也应该是稳定的。

构架描述可以有不同的形式，可以是对组成构架基线的模型（版本）的抽取，也可以是以一种便于阅读的形式对这些抽取的重写，我们将在4.4.3节"描述构架"中继续讨论该问题。不过，它应包括组成构架基线的模型的抽取（或视图）。在以后的阶段中，随着系统的进化和模型

⊖ 这样讲不是很确切。在细化阶段的最后所得到的用况模型版本，包括对构架重要的用况，也包括为了确定一个业务案例而需要说明的其他用况（最高可达80%）。这样，构架基线的用况模型和分析模型中包含的信息就比图4-4中所示的更丰富。然而，为了讨论的目的，我们做了这样的简化。

的增大，它还会包括模型新版本的视图。假定构架基线已经发展为一个稳定的构架——即对构架重要的模型元素已经稳定下来，在随后的迭代中不再改变——构架描述也将是稳定的，而且在每个时刻都包括系统的模型视图。

这里吸引人之处就是在第一个生命周期的细化阶段，在大约只投资了30%的初期产品发布时，就可能开发出一个稳定的构架。这个构架将成为系统剩余生命周期的基础。由于对基础的每个改动代价都很昂贵，在某些情况下也非常麻烦，因此，在开发工作的初期阶段得到一个稳定的构架是至关重要的。一方面，为某一特定系统开发构架是创建新的东西；另一方面，人们已经开发构架多年，已经积累了开发好的构架的经验和知识；有许多经过多年进化而得到的通用“方案”，包括结构、协作和实际的构架，每个有经验的构架设计师都应熟悉这些方案。这些方案通常称为“模式”，如[4]中描述的构架模式和[5]中描述的设计模式。通用模式是构架设计师所依赖的资源。

4.4.2 使用构架模式

建筑设计大师Christopher Alexander关于如何用“模式语言”对建筑和社区设计中重要的原则进行系统化和实用化的观点，启发了很多面向对象领域的人士定义、收集和测试各种软件模式[10]。“模式界”把模式定义为“一种在设计问题中常见的解决方案”。其中许多设计模式已经收录到书籍中，并用标准文档模板来表示模式。这些模板为每种模式命名，概括问题和产生问题的原因，并根据参与类的协作和类中对象的相互作用来提供解决方案。这些模板还给出了模式在某些程序设计语言中具体应用的实例，给出了模式的变体、运用该模式的好处和结果以及对相关模式的引用等。在Alexander之后，人们建议软件工程师应该了解许多标准模式的名称和意义，并运用它们实现更好、更易于理解的设计，如虚包（Facade）、内部修饰（Decorator）、委托监测者（Proxy Observer）、策略（Strategy）和访问者（Visitor）等设计模式已被广泛引用和应用。

模式界还运用具有细微差异的文档模板的模式概念来收集用于解决常见构架问题的标准方案。这些模式中包括层次（Layers）、管道和过滤（Pipes and Filters）、代理（Broker）、黑板（Blackboard）、水平-垂直元数据（Horizontal-Vertical Metadata）和MVC。其他建立的模式用于分析（“分析模式”）、实现（将通用面向对象的结构映射到语言的特定方面（如C++和Smalltalk）的“惯用语”），甚至用于有效的组织结构中（“组织模式”）。典型地，设计模式可以较为直接地映射到面向对象的程序设计语言上。在C++、Java和Smalltalk中都有实例，而构架模式主要针对系统或子系统和接口，实例并不一定包括代码。有关良好的分类方案参见[9]。

我们从模型驱动的角度，将模式定义为协作模板，一种可以按照**模板**（附录A）中的定义进行具体化的通用协作。于是，设计模式变成了可以用交互图解释其行为的类和实例间的协作。使用协作模板是因为这样的方案相当通用。用继承、扩展和其他机制对模式进行具体化（例如，说明模板中出现的类名称、类的数量等）。在对协作模板具体化时，多数情况会产生可直接跟踪到用况的具体的协作。关于设计模式的进一步处理参见[5]。

构架模式的使用与此类似，但主要是针对粒度更粗的子系统甚至系统的结构和交互。存在

很多种构架模式，这里只简单讨论一些最值得关注的模式。

代理模式[4]是一种管理对象分布的通用机制，它允许对象通过一个代理调用其他的远程对象，该代理将调用请求转发到包含被请求目标对象的节点和过程。这种转发的完成是透明的，就是说调用者无需知道被引用对象是否为远程的。代理模式经常利用委托设计模式，它提供一个具有与远程对象相同接口的本地代理对象，使分布式的通信形式和细节透明化。

还有其他一些模式有助于我们了解所构造系统的硬件，并帮助我们基于此硬件来设计系统，如客户机/服务器模式、三层（Three-Tier）模式和端对端（Peer-to-Peer）模式。这些模式定义了实施模型的结构，并建议构件应该如何分布到节点上。在4.5节中将举例说明如何把客户机/服务器模式运用到第3章所提到的ATM系统上。在这个例子中，客户机/服务器分布中的客户机节点执行所有的用户界面代码和一些针对每台实际的ATM业务逻辑（控制类）的代码。服务器节点则包含了实际的账户和每个事务需要得到验证的业务规则。

层次模式适用于多种系统。该模式定义了如何分层组织设计模式，就是说一层的构件只能参与恰好处于其下层中的构件。这种模式很重要，因为它简化了理解和组织开发复杂系统的工作。层次模式降低了依赖，因为低层不必关心高层的细节和接口。而且，它有助于确定什么可以重用，并提出了一个有助于我们决定买什么或构造什么的结构。

具有分层构架的系统在顶层具有单独的应用子系统。这些是在低层子系统的基础上（如框架和类库）构造的。如图4-5所示。通用应用层包含的子系统不是专用于一个单独的应用，而是在相同的领域或业务内能被很多不同的应用所重用。建立下面两层构架时不必考虑用况的细节，因为这些层不是针对具体业务的。而上面两层构架则要根据与构架相关的用况（这些层是用于具体业务的）来建立。

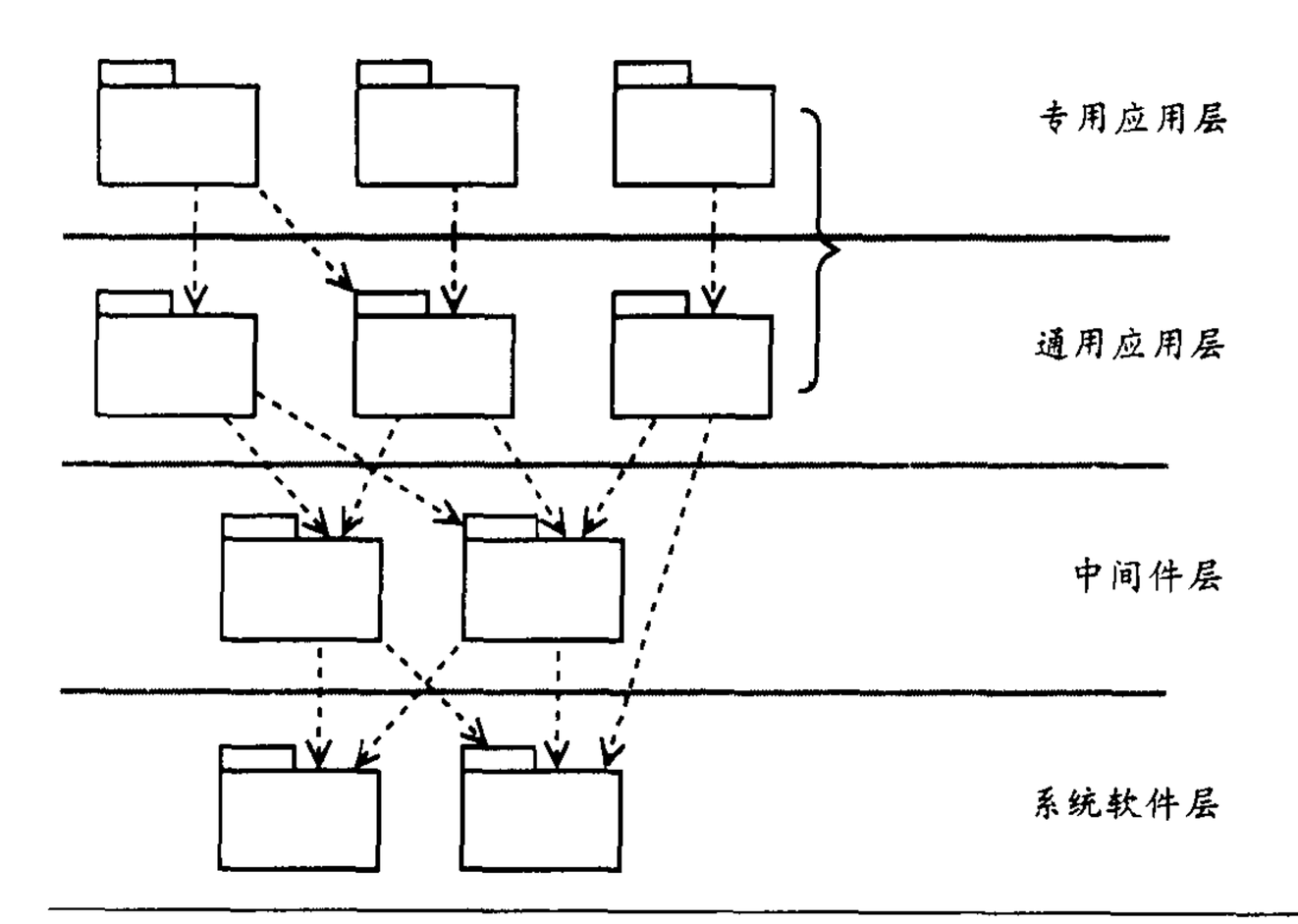


图4-5 分层构架通过多层子系统来组织系统

层次（附录C）是具有相同程度的通用性和接口易变性的子系统集合：低层通常用于多个应用，必须具有较稳定的接口；而高层更适用于具体应用，接口稳定性较差。既然低层提供的接口一般很少改变，那么高层的开发人员可以基于稳定的低层来构造。不同层次中的子系统可重

用用况、其他低层子系统、类、接口、协作和低层的构件。许多构架模式可用于一个单独的系统中。构成实施模型的模式（例如，客户机/服务器模式、三层模式或端对端模式）可以与层次模式相结合，其中层次模式可以辅助构成设计模型。用来处理不同模型中结构的模式彼此间通常是正交的。甚至处理同一模型的各种模式相互间通常也能很好结合。例如，代理模式和层次模式可以很好地结合在一起，两者都用于设计模型。代理模式处理如何解决透明对象分布的问题，而层次模式则用来显示如何组织整个设计。实际上，应该将代理模式实现为中间件层的一个子系统。

注意，有时候一种模式是占主导地位的。例如，在分层系统中，层次模式定义了整个构架和任务分解（将层分配给不同组），而管道和过滤可用于一个或多个层次内部。相反，在管道和过滤系统中，将整个构架表示为过滤模式间的流，而分层将被明确地用于一些过滤程序。

4.4.3　描述构架

正如在4.4.1节中所观察到的那样，细化阶段开发的构架基线以构架描述的形式保留下来。这个描述来自于细化阶段所产生的不同模型版本，如图4-6所示。构架描述是一种抽象，或者说是构架基线中模型的视图集合（也许进行了细微的改写以便它们更具可读性）。视图包括对构架重要的元素。当然，许多组成构架基线的模型元素会出现在构架描述中，然而并不是所有的元素都是如此。因为要得到一个可操作的基线，可能需要开发一些对构架不重要但对生成可执行代码有必要的模型元素。因为构架基线不只是用于开发构架，也在一定程度上用于说明系统需求，以便制定出详细的规划，该基线的用况模型还可以包含更多的不只是对构架视图有意义的用况。

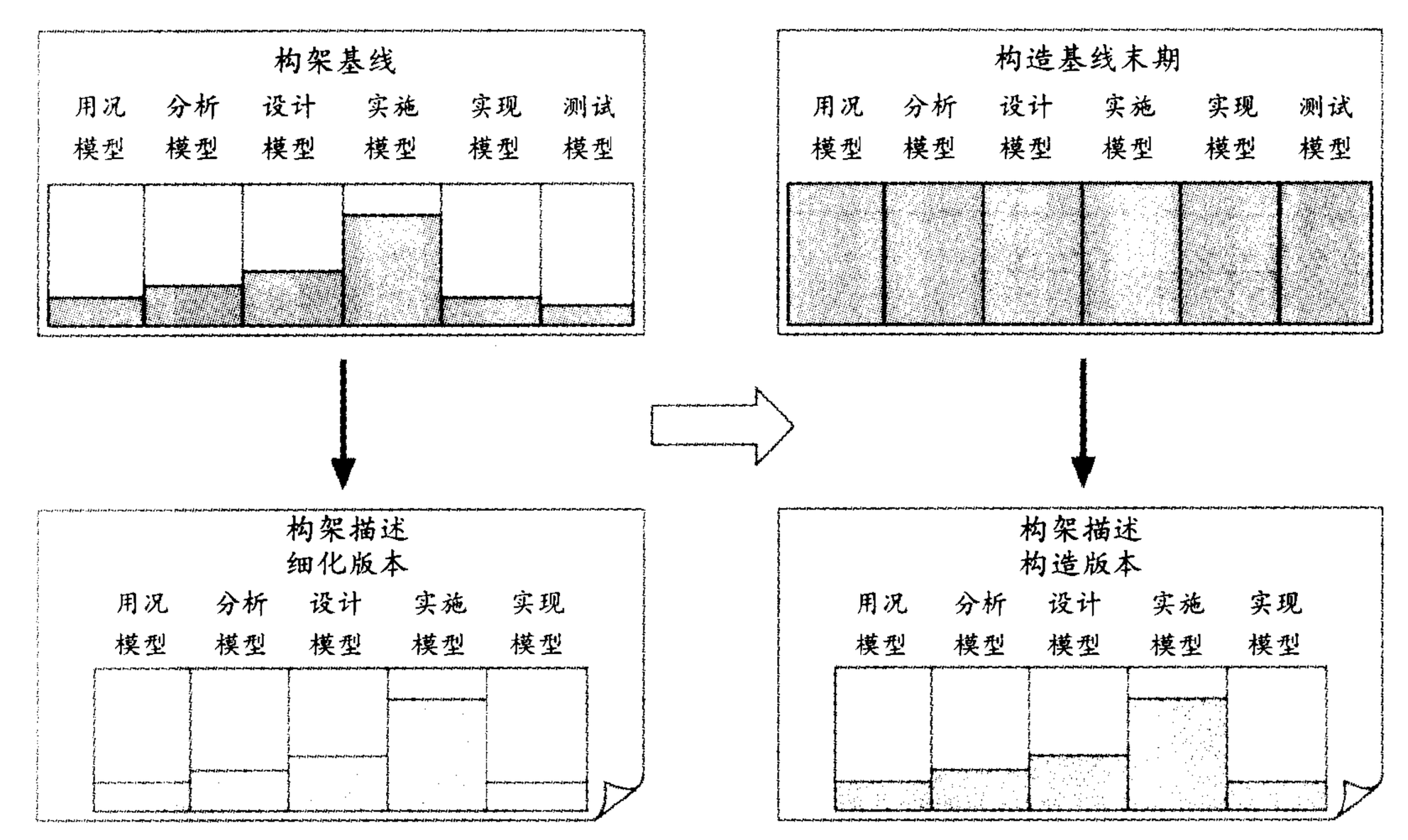


图4-6　在构造阶段，各种模型趋于完成（右上角填充阴影的图形）。因为大多数构架在细化阶段定义，因此构架描述没有明显变化（右下图）。构架的细微变化确实发生（由不同的填充图案表示）

在整个系统生命周期内，构架描述会不断更新，以反映与构架有关的变化和补充。这些变化通常很小，大致包括：

- 发现了新的抽象类和接口。
- 为现存的子系统添加新的功能。
- 升级到可重用构件的新版本。
- 重新安排处理结构。

构架描述本身也可能需要修改，但不必增加规模，只是对有关内容进行更新（如图4-6所示）。

正如前面所讨论的，构架描述可展示模型的视图，包括用况、子系统、接口、一些类和构件、节点以及协作。构架描述还包括用况没有描述的对构架有意义的需求。这些其他的需求是非功能性的，作为补充需求来说明，例如，提到**分布**和**并发**（附录C；另见9.3.2节）时涉及到安全及重点约束的那些需求。构架描述还应包括对平台、遗留系统和将使用的业务软件的简单说明，如Java中用于对象分布的远程消息请求（Remote Message Invocation，RMI）。而且，说明实现通用**机制**（附录C；另见9.5.1.4节）的框架也很重要，例如，在关系数据库中存储或检索一个对象。这些机制可在多个用况实现中得到重用，因为它们就是为实现可重用的协作而设计的。构架描述还应该对所有使用过的构架模式进行归档。

构架描述强调了最重要的设计问题，并将它们明确表示出来供他人探讨和提出反馈。然后需要对这些问题进行讨论、分析，并最终加以解决。例如，这些分析可能包括评估负载性能或内存需要，以及设想将来可能会危及到构架的需求。

尽管构架描述考虑得很周全，但它终究是一种高级的视图。一方面，不要指望它能阐明一切；而且它不应该把参与者淹没在太多的细节中。它是一张行车路线图（系统运作），而不是整个系统的详细说明。另一方面，它必须体现出每个参与者的需要，所以即使有100多页也不算多。如果把人们的所有需求以易于理解的方式纳入其中，可能需要使用一个巨大的文档。这就是构架描述应该做到的：它应该包括开发人员为了完成他们的工作所需要的内容。

在阅读构架描述时，我们可能会觉得对一些子系统它只涉及表面问题，而对另一些子系统它却详细地确定了它们的接口和协作。这种区别对待的原因是：详细说明的子系统对构架十分重要，应该处于构架设计师的控制之下（参见12.4.2节和14.4.3.1节）。

这也有助于探讨什么是非构架部分。绝大多数属于子系统或服务子系统私有的操作、接口和属性的类（对系统其他部分是隐含的），对于构架都不重要。作为其他子系统变体的子系统从构架角度来看也不重要。经验表明，只有不到10%的类与构架有关。其余的90%的类并不重要，因为它们对于系统的其他部分来讲是不可见的，其中一个类的改变不会影响到服务子系统外部的任何实质性的东西。大部分用况实现对构架也不重要，因为它们没有对系统施加任何额外的约束。这就是构架设计师通过为数不多的用况及其他需求就可以设计出构架的原因。大部分用况实现代表易于实现的简单的附加性能，尽管系统所提供的大部分功能是由它们构成的。关键在于：一旦选定了构架，绝大部分系统功能实际上就很容易实现了。

构架描述不包括那些只是为了确认或验证构架所需要的信息。因此，它不包括测试用例和测试规程，也不包括测试模型的构架视图。这些内容都不属于构架范畴。然而，正如图4-6所示

的，构架基线包含所有模型的版本，其中也有测试模型的版本。这样，作为构架描述基础的基线便会经历测试——所有的基线都如此。

4.4.4 构架设计师创建构架

构架是由构架设计师和其他开发人员共同创建的。他们致力于实现一个高性能、高质量的系统，而且要功能强大、可测试、对用户友好、可靠、高可用、精度高、可扩展、可变更、健壮、可维护、易携带、安全、可靠而且经济。他们清楚自己需要在这些约束范围内求生存，还要从中找到折衷方案——这就是需要构架设计师的原因。构架设计师对此承担最主要的技术责任，他们选取构架模式和现存产品，并安排子系统的依赖，使其能从不同侧面分别考察问题。在此分别考察问题是指在创建一种设计方案时，当其中某个子系统发生变化时不会对其他子系统造成影响。

根本目标是以当前的技术状况和该应用可承受的成本、用最佳的方式来满足应用的需要，换句话说，也就是能够在现在和将来有效地实现应用功能（即用况）。这里构架设计师得到了UML和统一过程的支持。UML具有强有力的结构来系统地阐明构架，而统一过程则详细地介绍了构造良好构架的原则。即使如此，所选定的构架最终还是基于技术和经验进行判断的结果。构架设计师将负责做出这一判断。细化阶段末期，当构架设计师将构架描述提交给项目经理时，这就意味着，“现在我清楚了我们可以构造出系统而不会遇到太大的技术难题。”

合格的构架设计师需要具备两种能力。一种是他所从事的领域知识，因为他必须明智地与所有项目相关人员配合，而不只是开发人员。另一种就是软件开发知识，甚至到最基本的编码能力，因为他必须向开发人员说明构架，协调他们的工作，了解他们的反馈。如果构架设计师具有类似于目前正在开发的系统的经验也是很有价值的。

构架设计师在软件组织中处于一个非常重要的职位，他承担的任务十分艰巨。他不应该是项目经理，因为这个职位除了构架还有其他许多工作要做。他确实必须对管理全力支持，既要创建构架，又要予以实施。他还必须有充分的适应能力以便能够从开发人员和其他项目相关人员获得有用的反馈。这些就是对构架设计师所应具备的能力的简单概括。对于庞大的系统，一个构架设计师是不够的，成立一个构架设计小组来开发和维护构架可能更为合理。

开发构架需要大量的时间。将这段时间安排在开发日程的最前面，可能会令那些已经习惯于把大部分时间用于实现和测试的项目经理感到不安。然而，经验表明，有一个好的构架作为指导，后面的各阶段会大大缩短整个开发周期，这就是第5章中要涉及的内容。

4.5 最后是构架描述！

关于构架是什么已经讨论了很久却没有给出实质性的例子。现在举一个具体的例子来看看构架究竟是什么样。不过，在举例之前需要解释一下为什么这样做不太容易。

我们说过，构架描述只是对系统模型的适当抽取（即不增加任何新的内容）。构架描述的第一个版本就是对第一个生命周期中细化阶段末期的模型版本的抽取。如果不想将这些抽取改写

为更易读的形式，则构架描述看起来与系统的一般模型非常相似。这种外部特征意味着用况模型的构架视图看起来就像一般的用况模型。唯一的区别就是用况模型的构架视图只包括对构架重要的用况（参见12.6.2节），而最终的用况模型包含所有用况。设计模型的构架视图也是如此。它看起来像是一个设计模型，但它只实现对构架有意义的用况。

很难举出例子的另一个原因是只有针对实际的系统谈论构架才有意义，而且如果希望详细讨论一个系统的话，它必须是一个非常小的系统。现在我们用第3章中简单的ATM例子来解释构架视图所需要的内容，将通过比较视图中的内容和系统全部模型中的内容来介绍构架描述。

构架描述包括五部分，每部分说明一个模型：用况模型视图、分析模型视图（不一定总有）、设计模型视图、实施模型视图和实现模型视图。但构架描述不包括测试模型视图，因为它对描述构架不起作用，它只是用来验证构架基线。

4.5.1 用况模型的构架视图

用况模型的构架视图展示了最重要的参与者和用况（或者是这些用况的脚本）。关于ATM系统的用况模型见3.3节“捕获用况”。

举例：ATM系统中用况模型的构架视图

在ATM的例子中，“取款”是最重要的用况。没有它，也就没有实际的ATM系统。“存款”和“在不同账户间转账”用况对一般的银行储户来讲不太重要。

所以在定义构架时，构架设计师认为“取款”用况要在细化阶段完全实现，而其他用况（或用况中的一部分）对于构架的目标意义不大（实际操作中，做出这样的决定还为时过早，不过这里只是为了讨论方便）。

因此，用况模型的构架视图应该显示出“取款”用况的完整描述。

4.5.2 设计模型的构架视图

设计模型的构架视图展示了设计模型中对构架最为重要的类元：最重要的子系统和接口，还有一些很重要的类（主要是主动类）。它还展示了最重要的用况是如何按照这些类元实现的，即如何实现用况的。在4.5.3节中介绍实施模型时还会讨论到主动类（在此，将主动类分配到节点上）。

举例：ATM系统中设计模型的构架视图

在3.4.3节中，我们确定了三个主动类：“客户管理”、“事务管理”和“账户管理”（图4-7）。它们都被包含在设计模型的构架视图中。

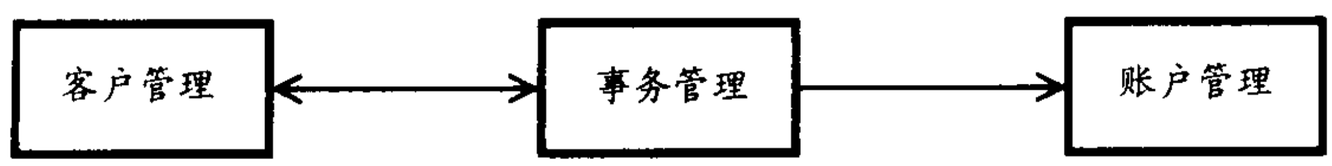


图4-7 ATM系统中设计模型构架视图的静态结构。这是一个描述主动类的类图。

而且，在3.4.4节中，我们已经对三个子系统（“ATM接口”、“事务管理”和“账户管理”）有所了解，参见图4-8。这些子系统是用于实现“取款”用况的，所以它们对构架很重要。设计模型还包括很多其他的子系统，但这里不进行讨论。

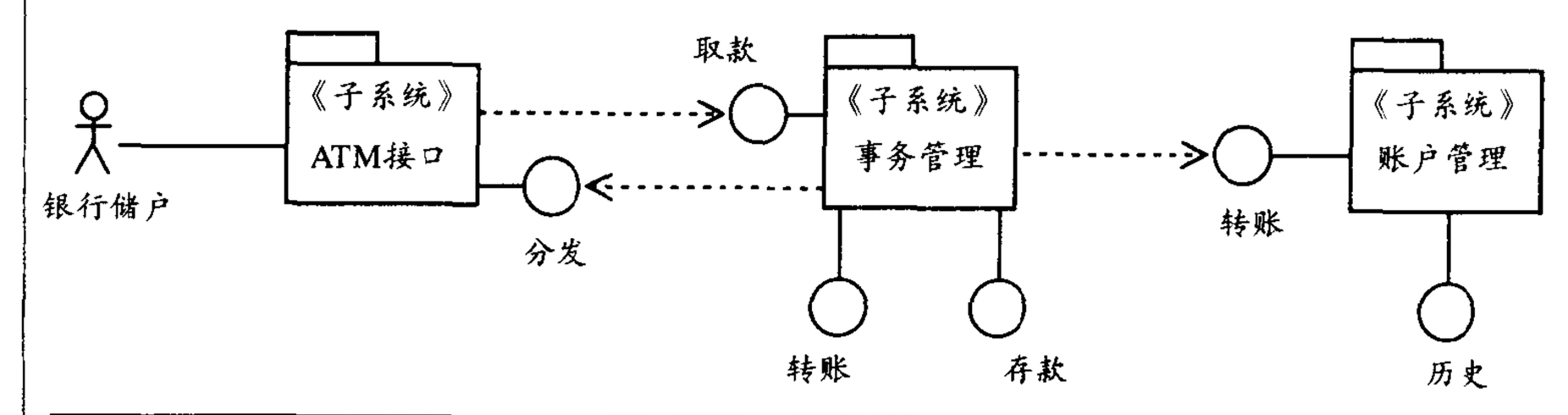


图4-8 ATM系统中设计模型构架视图的静态结构。这是描述子系统及其接口的类图。

“ATM接口”子系统处理银行储户的所有输入和输出，如打印收据和接受银行储户的指令。“账户管理”子系统中保存所有长期账户的信息，用于处理所有账户事务。“事务管理”子系统包含用况专用行为的类，如“取款”用况的专用行为。在3.4.4节的例子中曾提到过用况专用的类通常包含在不同的服务子系统中，例如在“事务管理”子系统中，就有“取款”类、“转账”类和“存入”类的服务子系统（图4-8中未表示出来）。实际上，每个这样的服务子系统通常包括几个类，但我们的例子很简单。

图4-8中的子系统通过接口彼此提供一定的行为，如由“账户管理”提供的“转账”接口。“转账”、“取款”和“分配”接口已经在3.4.4节中进行了说明。还有“转账”、“存入”和“历史”接口，但这些接口没有包括在这个例子所涉及的用况中，所以不对其进行解释。

只有静态结构是不够的，还需要说明设计模型中对构架重要的用况是如何通过子系统实现的。因此，这里从子系统和参与者相交互的角度再次说明“取款”用况，如图4-9中的协作图所示（附录A）。子系统所属类的对象相互之间进行交互来执行一个用况实例。对象间相互传递消息，这些交换如图中所示。消息携带有指明子系统接口所属操作的名称，由“::”符号表示（如“取款::执行（数额，账户）”），这里“取款”是“事务管理”子系统中的类所提供的接口）。

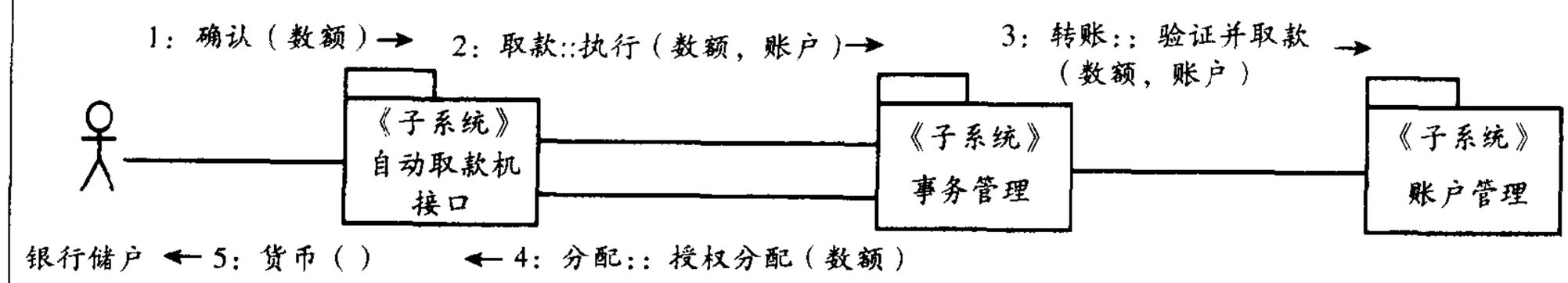


图4-9 协作执行“取款”用况的子系统

下面内容简要说明了用况实现中的流。内容基本上与3.4.1节（用况实现描述）中的描述相同，但这里指的是子系统而不是类。

前提条件：银行储户有一个可以用于ATM的银行账户。

1) 参与者“银行储户”选择取款并向“ATM接口”表明自己的身份，这可以通过使用具有编号的磁卡和PIN（个人身份号码）来实现。银行储户还要说明取多少现金并从哪个账户中取。这里假定“ATM接口”子系统能够确认身份。

2) “ATM接口”请求“事务管理”子系统取款。“事务管理”子系统负责将提取现金的整个动作序列作为一个原子事务来执行，以便从账户中扣除取款金额，并将现金分发给银行储户。

3) “事务管理”请求“账户管理”子系统取款。“账户管理”子系统决定能否取出现金，如果能，则从账户中扣除取款的金额并返回一个应答，表明可以执行取款动作。

4) “事务管理”授权“ATM接口”子系统分发货币。

5) “ATM接口”将现金分发给“银行储户”。

4.5.3 实施模型的构架视图

实施模型根据相互连接的节点定义实际的系统构架。这些节点是软件构件能够在其上运行的硬件单元。通常我们所了解的实际系统构架看起来就像着手开发系统之前的样子。于是，便可以在需求工作流期间尽早将节点和连接在实施模型中进行模型化。

在设计期间，需要确定哪些类是主动的，即确定线程或过程。还要确定每个主动对象应该做什么，这些主动对象的生命周期应该怎样，以及这些主动对象如何通信、同步和共享信息。将主动对象分配到实施模型的节点上。在将主动对象分配给节点时，需要考虑节点的性能（如它们的处理能力和内存大小）以及连接的特点（如带宽和可用性）。

实施模型的节点和连接以及分配给节点的主动对象可以在**实施图**（附录A）中表示出来。这些图还可以表明如何将可运行的构件分配给节点。例子中的ATM系统分布在三个不同的节点上。

举例：ATM系统中实施模型的构架视图

“银行储户”通过“ATM 客户机”节点访问系统，该节点通过访问“ATM应用服务器”来执行事务（见图4-10）。而“ATM应用服务器”又利用“ATM数据服务器”对账户执行具体

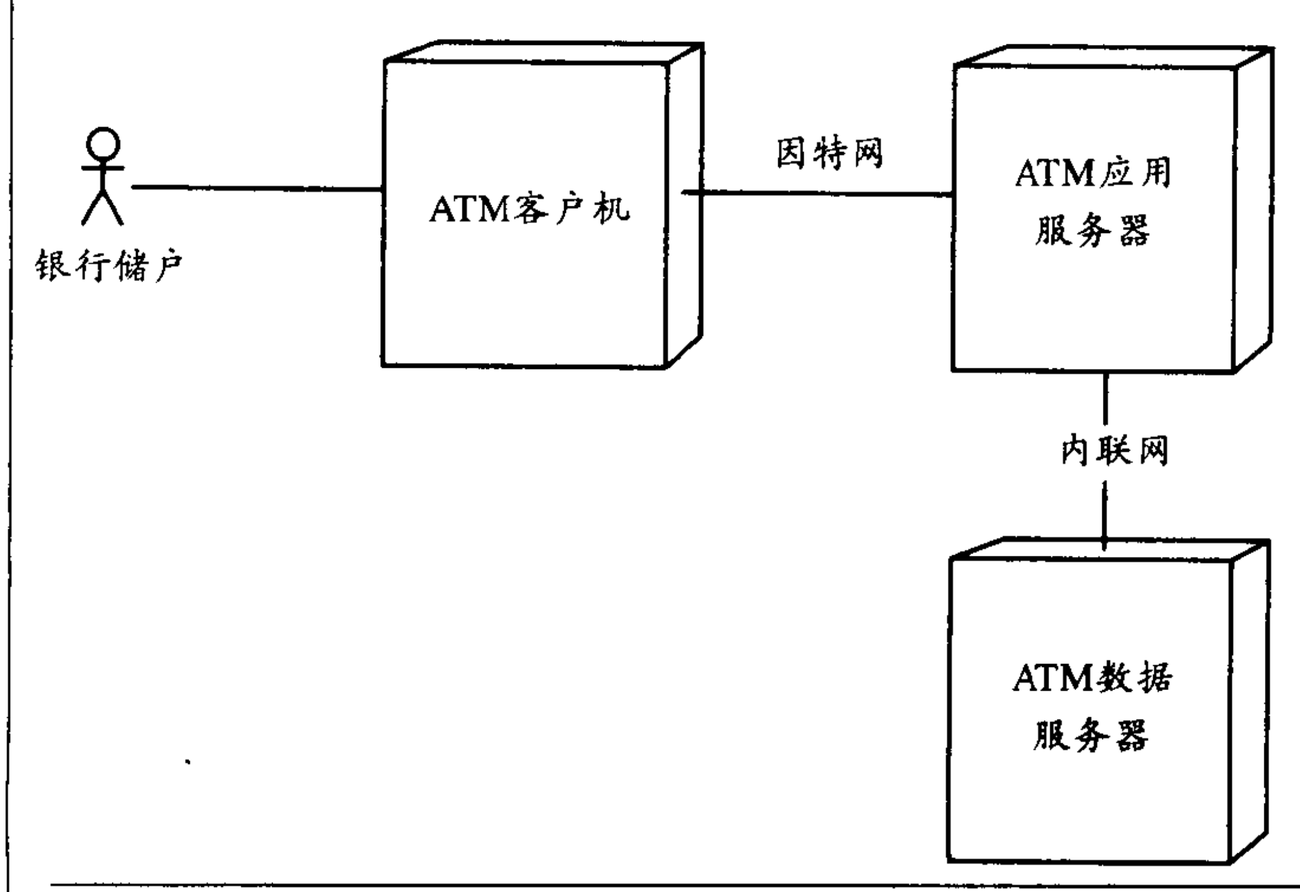


图4-10 实施模型定义了三个节点：ATM 客户机、ATM应用服务器和ATM数据服务器

的事务。这不仅对于“取款”用况（已将其归为对构架重要的用况）是事实，而且对其他用况（如“存款”和“转账”用况）也是如此。在3.4.3节中，曾介绍过选择了哪些类来实现“取款”用况。

当定义这些节点时，可以将功能部署到节点上。为简单起见，将每个子系统（见图4-8）作为一个整体部署在一个节点上。“ATM接口”子系统部署在“ATM 客户机”节点上，“事务管理”子系统部署在“ATM应用服务器”上，“账户管理”子系统部署在“ATM数据服务器”上。因此，这些子系统中的每个主动类（参见3.4.4节和图3-10）都部署在相应的节点上，并具体表现为运行于该节点上的一个过程。在过程空间保存的每一个这样的过程维护该子系统中其他（一般的，非主动的）类的对象。主动对象的部署如图4-11中所示。

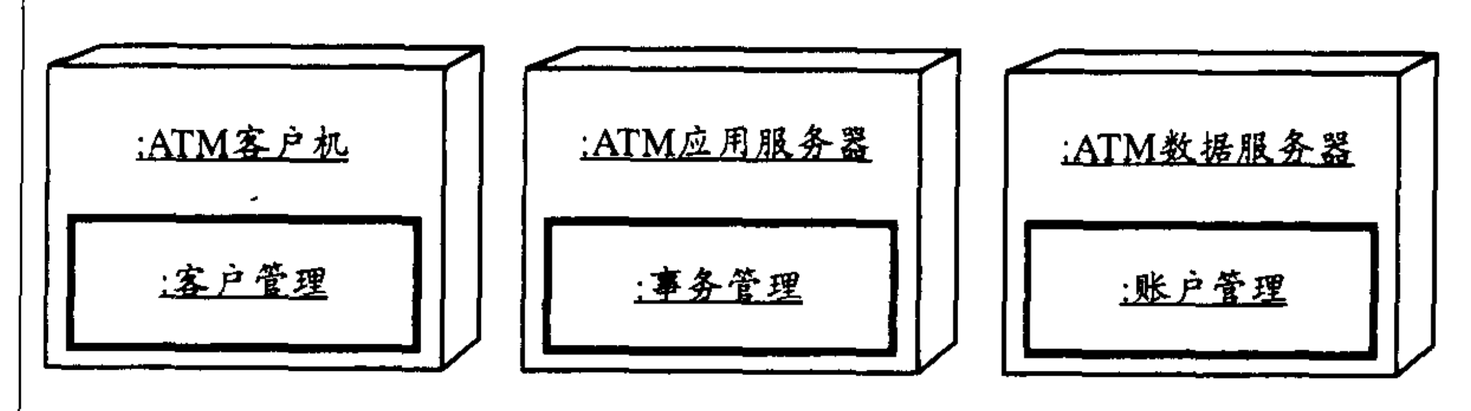


图4-11 实施模型的构架视图。ATM系统的主动类分布到几个节点上。主动类用粗边框的矩形表示

这是一个系统分布的简单例子。在实际的系统中，分布必然更为复杂。对于分布问题的另一种方案是使用某些用于对象分布的中间件，如对象请求代理（ORB）。

4.5.4 实现模型的构架视图

实现模型是从设计模型和实施模型直接映射得到的。每个设计服务子系统通常会为它所安装的节点类型产生一个构件（但并不总是如此）。有时候，同一个构件（如缓冲区管理构件）可能会在几个节点上实例化和运行。一些语言（如JavaBeans）提供了封装构件的结构。否则，要将类组织到表示所选构件集合的代码文件中。

在3.4.5节中，曾提到过“取款管理”服务子系统有可能实现为两个构件：“在服务器端取款”和“在客户机端取款”。“在服务器端取款”构件可以实现“取款”类，而“在客户机端取款”构件可以实现一个“取款代理”类。在我们的简单例子中，这些构件中的每一个构件将只实现一个类。在实际的系统中，每个服务子系统应该有更多的类，所以一个构件要实现多个类。

4.6 三个应关注的概念

4.6.1 什么是构架？

构架是构架设计师在构件描述中详细说明的内容。从技术角度看，构架描述使构架设计师

可以控制系统的开发。软件构架侧重于系统的重要结构元素，如子系统、类、构件和节点，以及这些元素通过接口实现的协作。

用况驱动构架，使系统提供预期的用途和功能，以满足合理的性能指标。构架应该综合权衡各方面的因素，但需要有足够的柔性以便包容新的功能，还需要它能支持对现有软件的重用。

4.6.2 如何获得构架？

构架是在细化阶段通过需求、分析、设计、实现和测试等多次迭代开发出来的。对构架重要的用况以及其他一些输入可用来实现构架基线，或称“骨架”系统。这里其他一些输入包括系统软件需求、中间件、所用的遗留系统、非功能性需求等。

4.6.3 如何描述构架？

构架描述是系统各种模型的视图，即是用况、分析、设计、实施和实现模型的视图。构架描述叙述了系统的组成部分，这对所有开发人员和其他项目相关人员理解该系统是十分重要的。

4.7 参考资料

[1] David Garlan and Mary Shaw, *Software Architecture: Perspectives on an Emerging Discipline,* Upper Saddle River, NJ: Prentice-Hall, 1996.

[2] Ivar Jacobson, Martin Griss, and Patrik Jonsson, *Software Reuse: Architecture, Process and Organization for Business Success,* Reading, MA: Addison-Wesley, 1997.

[3] P.B. Kruchten. "The 4+1 view model of architecture," *IEEE Software,* November 1995.

[4] F. Buschmann, R. Meurier, H. Rohnert, P. Sommerlad, M. Stal, *A System of Patterns,* New York: John Wiley and Sons, 1996.

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software,* Reading, MA: Addison-Wesley, 1994.

[6] OMG, Inc. The Common Object Request Broker: Architecture and Specification (CORBA), Framingham, MA. 1996.

[7] ISO/IEC International Standard 10165-4 = ITU-T Recommendation X.722.

[8] ITU-T Recommendation M.3010, Principles for a Telecommunication Management Network.

[9] Thomas J. Mowbray and Raphael C. Malveau, *CORBA Design Patterns*, New York: John Wiley and Sons, 1997.

[10] Christopher Alexander, Sara Ishikawa, Murray Silverstein, with Max Jacobsen, Ingrid Fiksdahi-King, Shlomo Angel, *A Pattern Language: Towns, Buildings, Construction,* New York: Oxford University Press, 1977.

第5章
迭代和增量过程

软件过程要做到更为有效，需要具有一系列非常清晰的**里程碑**（附录C），以便为项目经理和项目组的其他人员提供一个准则，他们需要以此来认可产品周期是否可以从一个阶段进入下一个阶段。

在每个阶段中，过程需要经历一系列的**迭代**和增量（附录C）来满足这些准则。

初始阶段的根本准则是可行性，它通过以下方式达到：

- 确定并降低影响系统可行性的关键**风险**（附录C；另见12.5节）。
- 通过用况建模将需求的关键子集转化为候选构架。
- 在宽限条件下对成本、工作量、进度和产品质量进行初步估计。
- 还是在宽限条件下，开始对从经济上考虑值得做的项目创建业务案例（有关业务案例的细节参见第12～16章）。

细化阶段的根本准则是在经济框架内构造系统的能力，它通过以下方式达到：

- 确定并降低对系统构造有重大影响的风险。
- 确定代表待开发功能的大部分用况。
- 将候选构架扩展为可执行的基线部分。
- 制定足够详细的**项目计划**（附录C）来指导构造阶段。
- 在较严的条件下进行估计，以便调整项目的商业报价。
- 对值得做的项目确定业务案例的最终方案。

构造阶段的根本准则是使系统能够在用户环境下完成最初的操作，它通过以下方式达到：

- 通过一系列迭代，得到周期性的构造和增量，这样在完成该阶段时，系统可行性便会以可执行的形式明确表现出来。

移交阶段的根本准则是一个达到最终操作能力的系统，它通过以下方式达到：

- 修改产品以缓解在早期阶段没有发现的问题。
- 修正**缺陷**（附录C，另见11.3.6节）。

一般来说，统一过程的目标之一是使构架设计师、开发人员和其他项目相关人员了解到早期阶段的重要性。为了实现这一目标，我们最好引述一下Barry Boehm几年前的忠告：

我无须过分强调生命周期的构架里程碑[对应于我们细化阶段的里程碑]对你的项目和你的职业生涯有多关键。如果你还没有满足LCA（Life Cycle Architecture，生命周期构架）里程碑的准则，不要进行全面开发。重新召集所有项目相关人员研究并制定一个新的、能够完全满足LCA准则的项目计划。

其中关于阶段和迭代的问题将在第三部分进行更详细的描述。

5.1 迭代和增量概述

正如在第3、4章中指出的那样，软件开发过程应该是用况驱动和以构架为中心的，这是统一过程三个关键点中的两个关键点。这些方面对使用统一过程开发产品有明显的技术影响。软件开发过程是用况驱动的意味着在驱动中产生最终产品的每个阶段都能回顾一下为用户实际上做了什么，从而使开发人员能确保系统满足用户的真正需要。以构架为中心意味着开发工作侧重于建立在早期阶段指导系统构造的构架模式，保证不仅对当前产品版本、而且对整个产品的生命都有一个顺利的发展。

在用况和构架间达到恰当的平衡很像产品开发中功能和表现形式的平衡，这将随着时间的前进而逐步实现。二者哪个在先是一个“鸡和蛋”的问题，如同我们在4.3节中介绍的一样。在漫长的进化过程中，几乎经过了无数次的迭代过程才出现了“鸡和蛋”。同样，在短暂的软件开发过程中，开发人员通过一系列迭代有意识地做到在用况和构架间的平衡。因此，迭代和增量开发方法构成统一过程的第三个关键方面。

5.1.1 以细小的步骤开发

第三个关键点提出了以细小的可管理的步骤开发软件产品的策略：

- 计划一小步。
- 说明、设计和实现一小步。
- 集成、测试和运行每次迭代（一小步）。

如果你对这一步感到满意，就进行下一步。在每步之间会得到反馈，并以此来调整下一步的侧重点。然后开始另一步，接着再一步。当处理完计划的所有步骤后，便开发出了可以向客户和用户发布的产品。

早期阶段的迭代主要是关注确定项目的范围，消除关键风险和建立构架基线。然后，当我们开始着手实现项目，逐渐地降低了剩余的风险并实现了构件后，迭代的形式就会发生改变并产生增量结果。

一个软件开发项目是将用户需求的“增量”（或变化）转化为软件产品的增量（或变化）（参见2.2节）。使用迭代和增量的方法，这种变化的积累是一点一点完成的。换句话说，我们将该项目分解成许多袖珍项目，每一个袖珍项目都作为一次迭代。每次迭代均包括软件开发项目所具备的一切：规划，通过一系列工作流进行处理（需求、分析和设计、实现、测试），最后准备发布。

但是，迭代不是一个完全独立的实体，它是项目的一个阶段，它在很大程度上是作为项目的一部分而得到的。我们说它是一个袖珍项目是因为它本身不是客户、用户或高层管理人员要求我们去做的。另外，每个这样的袖珍项目都像过去的瀑布模型，因为它处理的是瀑布模型的活动。我们可以将每次迭代标注为一个“袖珍瀑布”。

迭代生命周期以内部版本的形式（尽管是初步的）交付实际的结果，其中每个版本均会增

加一个增量并表明所关注的风险得以降低。这些版本可以展示给客户和用户，从而获得有价值的反馈以确认工作成果。

规划人员设法对迭代进行排序以得到一条有序的途径，使早期的迭代为后期的迭代提供认知基础。项目的早期迭代增进了对需求、问题、风险和**方案领域**（附录C）的了解，而后期迭代产生的附加增量最终构成了**外部版本**（附录C），即客户产品。对规划人员来说，最终的成功是一个一直前进的迭代系列；即从来不必因为在后期的迭代中了解到的一些问题而回退2~3个迭代去修补模型。我们不想爬一堆正在溶化的雪，前进两步便会滑下一步。

总之，生命周期是由一系列迭代组成的。有些迭代，特别是早期的迭代，有助于我们了解风险、确定可行性、构建软件的初始核心并建立业务案例。而其他迭代，特别是后期的迭代，则不断增加增量结果，直至得到一个可以对外发布的产品。

迭代有助于管理层规划、组织、监控和控制项目。迭代被组织到四个阶段中，每个阶段都要考虑配备人员、提供资金、安排进度以及入口和出口准则等问题。在每个阶段开始，管理人员可以决定如何来执行，必须交付什么结果和必须降低哪些风险。

5.1.2 迭代不是什么

有些经理认为“迭代或增量”是对“探路”的美称。他们担心这些词只不过隐藏了开发人员不知道他们正在做什么的事实。在初始阶段，甚至在细化阶段的早期，这可能有些道理。例如，如果开发人员还未消除关键或重大的风险，那么这个断言就是正确的；如果他们还没有证明基础的概念或建立构架基线，那么这个断言是正确的；如果他们还未领会如何能够实现最关键的需求，那么这个断言也是正确的。确实，他们可能不知道他们在做什么。

他们假装确实知道在做什么会有好处吗？根据不充分的信息进行规划有好处吗？遵循一个不可靠的计划会有好处吗？当然没有好处。

因此，我们强调一下迭代生命周期不是什么：

- 它不是任意的探路。
- 它不是开发人员的游戏围栏。
- 它不是只影响开发人员。
- 它不是开发人员一遍遍重复设计同样的东西直到偶然遇到一个有用的东西。
- 它不是不可预见的。
- 它不是计划和管理失败的理由。

实际上，受控的迭代根本不是任意的，它是有计划的。它是项目经理可以用来控制项目的工具。在生命周期早期，它降低了可能会对开发进度存在威胁的风险。迭代之后得到的**内部版本**（附录C）可以获得项目相关人员的反馈，以尽早对项目进程进行修改。

5.2 为什么采用迭代和增量的开发方法？

原因可以表示为两个英文词：“better software”（更好的软件）。详细一点来说，就是为了达

到用来控制开发的主要里程碑和次要里程碑。更详细一些来说，就是：

- 为了尽早处理关键风险和重要风险。
- 为了建立一个构架来指导软件开发。
- 为更好地处理不可避免的需求以及其他变化而提供一个框架。
- 为了随时间而递增地构建系统，而不是在临近结束时才一下子建成一个系统，那时候的改变需要付出很大的代价。
- 为了提供一个开发过程，使所有工作人员可以更高效地工作。

5.2.1 降低风险

就像任何工程活动一样，软件开发也会遇到风险。管理领域预言家Peter F. Drucker认为："风险是将现有资源投向未来期望结果的委托行为中固有的"[2]。在软件开发中，我们通过在开发中尽可能早地确定风险并做出迅速处理来解决这个现实问题。风险是会导致损失或危害的一种潜在的可能性。风险是构成危险的因素、事件、要素或进程，其危害程度是不确定的。在软件开发中，我们可以将风险定义为一个需要关注的问题，它在一定程度上可能会危及项目的成功。例如：

- 我们最初考虑的**对象请求代理**（附录C）可能无法处理每秒钟1000个"远程客户账户"对象的查询请求。
- 一个实时系统可能需要一些没有在细化阶段确定的数据输入。它可能需要通过大量的但无法确知细节的计算来处理这些数据。它可能在一个较短但目前尚不能确定的时间内发出一个命令信号。
- 一个电话交换系统可能必须在几个毫秒内响应由客户电讯业务公司定义的各种输入。

软件领域所需要的，就像Barry Boehm多年前所写的一样，是一个过程模型，它"创建的是软件过程的风险驱动方法，而不是文档驱动或代码驱动的过程"[3]。统一过程满足这些准则，因为它在最初两个阶段（初始和细化阶段）中便致力于解决主要的风险，并在构造阶段的早期按照风险的重要顺序逐个解决其余的风险。它在早期阶段通过迭代标识、管理并降低风险。因此，未经确定的或忽略的风险不会在后期突然出现并危害整个项目。

降低风险的迭代开发方法与**瀑布方法**（附录C）相比没有什么相似之处。瀑布模型显示出通过一系列步骤的单向开发流程：需求、分析、设计、实现和测试。按照这种方法，在项目到达实现、**集成**（附录C）和测试阶段时，将会涉及到所有的开发人员。在集成和测试阶段，问题才开始在他们周围爆发出来。项目经理将被迫重新分配人员——通常是更有经验的开发人员——去解决这些问题，然后工作才能继续进行下去。然而，因为所有开发人员都已经有分配好的任务，项目经理发现很难"腾出"最有资格的人员去解决这些新发现的问题。更为困难的是，重新分配经验丰富的开发人员去"清理残余问题"常常让剩下的缺少经验的开发人员停下来等待。最终导致项目超期，成本增加。最坏的情况是，竞争对手首先占领了市场。

如果按风险和开发时间画成一张图，如图5-1所示，就会发现迭代开发在最早的迭代过程便开始着手降低关键风险。在工作到达构造阶段时，就已经没有什么重大风险，工作得以顺利进

行。相反，使用瀑布模型，重大风险直到代码集成阶段“大爆发”时才得到处理。

按照Capers Jones的说法，大约2/3的软件项目未能进行合适的风险评估[4]，所以存在巨大的改进空间！在开发过程尽早着手处理风险应该是首要的步骤。

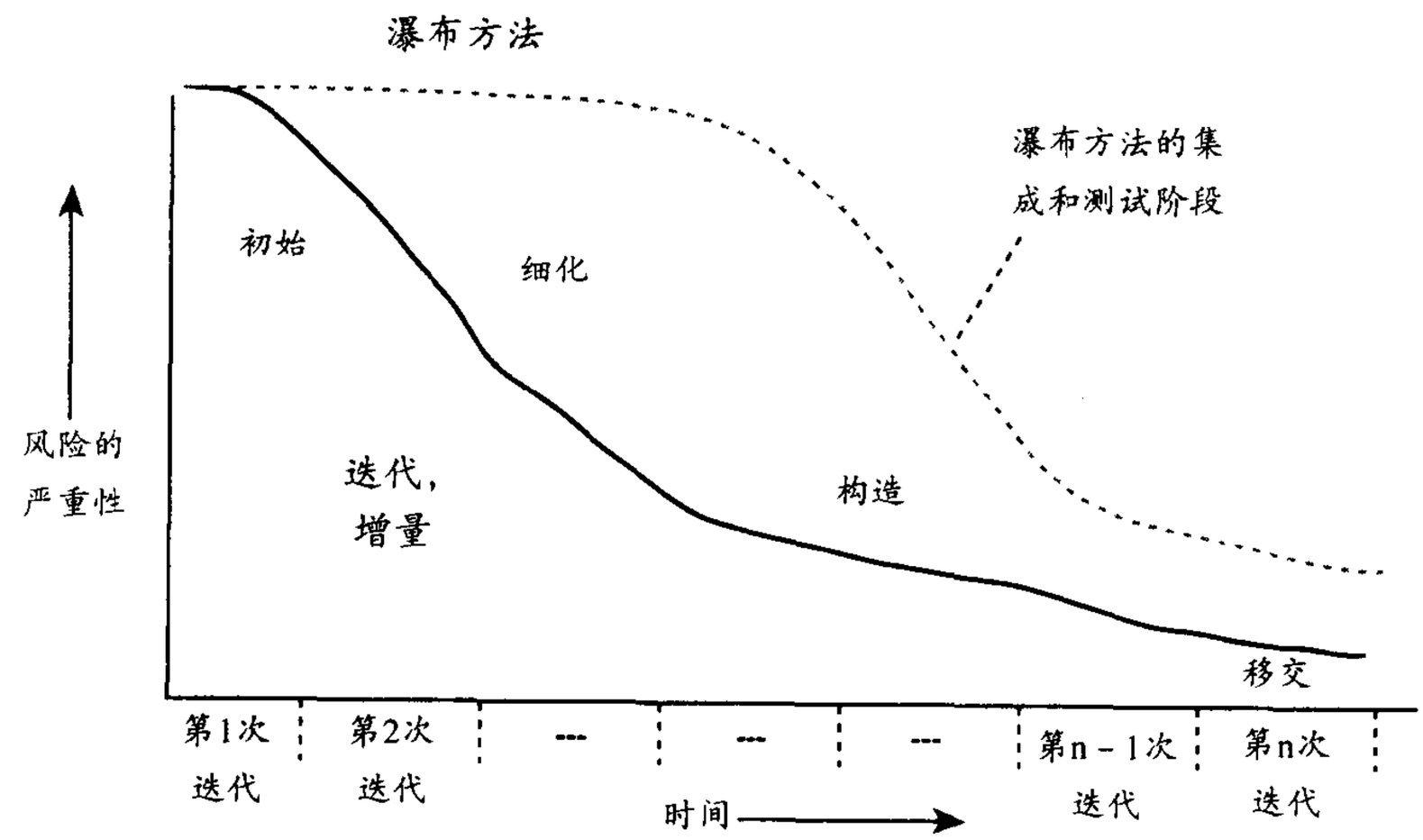


图5-1 与瀑布开发不同，迭代开发在早期标识并降低关键风险。在瀑布开发中，最关键的风险直到集成和系统测试时才进行处理（虚线表示）。图中最下部所示的迭代只与迭代和增量的方法有关

5.2.2 获得一个健壮的构架

得到一个健壮的构架本身是早期阶段迭代的结果。例如，在初始阶段，寻找一个能满足关键需求、克服关键风险和解决开发中主要问题的核心构架。在细化阶段，我们建立构架基线，指导进一步的开发。

一方面，这些阶段的投资还是比较少的，能够负担得起这些确保构架健壮性的迭代。例如，在细化阶段的第一次迭代后，可以对构架进行初步评估。此时，如果发现需要对构架进行改变以满足重要用况的需要和非功能性需求，我们还能负担得起。

如果沿用瀑布方法，到发现需要对构架进行改变时，我们已经在开发上投资太多以致于改变构架会导致严重的经济损失。而且，我们将临近承诺的交付日期。由于成本和进度的制约，我们将不能主动对构架进行大的改变。通过在细化阶段侧重于创建构架，可以避免这种进退两难的局面。在生命周期早期，我们便将构架稳定在基线级，此时，成本仍然比较低，而且在进度上我们仍有伸缩的余地。

5.2.3 处理不断变化的需求

对用户来说，一个可以操作的系统（尽管它还不能完全运行）比一个仅存在上百页文档的系统更容易理解。另外，如果存在的只是文档，他们很难了解项目的进展情况。因此，从用户

和项目相关人员角度来看，通过一系列可执行的版本或“构造”来进化产品，比罗列一堆难以理解的文档更为有效。一个构造是系统的部分可运行版本，它展示了系统能力的子集。每次迭代可以通过一系列构造以达到原先规划的结果（即增量）。

在早期阶段，系统部分地运行就可使用户和其他项目相关人员提供建议并指出可能被忽略的需求。计划——预算和进度表——还没有最终确定下来，开发人员可以容易地进行修正。在单向的瀑布模型中，用户直到集成和测试阶段才能看到可运行的系统。那时即使是那些有价值或看起来很小的改变，几乎总会不可避免地增加投资或拖延进度。因此，迭代生命周期更易于客户在开发周期中尽早了解需要附加或改变的需求，也更容易使开发人员进行相应的修改。归根到底因为他们是按照一系列迭代进行系统构造的，所以响应反馈或进行修改只是对一个增量的改变。

5.2.4 允许灵活改变

采用迭代和增量的方法，开发人员可以解决前期构造中发现的问题，并能立刻改变来纠正这些问题。通过使用这种方法，可以逐渐发现问题，开发人员能够很容易地纠正这些问题。在瀑布模型中，在“爆发”的集成阶段大量涌现出的错误报告经常会中断项目的进展。如果这种中断很严重，项目可能会终止，开发人员由于压力而垂头丧气，项目经理焦头烂额，其他管理人员也手足无措。相反，一系列可运行的构造结果会为所有人带来成就感。

测试人员、手册编写人员、工具开发人员、配置管理人员和质量保证人员都可以调整他们自己的计划来适应进化中的项目进度表。开发人员在项目早期首次遇到会导致严重迟滞的问题时，便察觉到了它们的存在。他们有时间去调整自己的进度。如果问题一直隐藏到测试时才发现，这时再重新制定有效的进度就太晚了。

当质量保证人员测试过一次迭代后，项目经理、构架设计师和其他项目相关人员可以根据预定义的准则对其进行评价。他们可以决定这次迭代是否得到了恰当的增量，风险是否完全消除。这种评价使项目经理可以确定这次迭代是否成功。如果是，他们便可以批准开始下一次迭代。如果迭代只是部分成功，他们可以延长本次迭代或在下一次迭代中继续解决尚未解决的问题或进行必要的返工。在极端的情况下，评价结果是完全否定的，他们可以终止整个项目。

5.2.5 实现持续的集成

在每次迭代的最后，项目组表明它们已经降低了一些风险。每次迭代后，项目组均交付递增的功能，这使与项目相关的所有人员能够清楚地看到项目的进展情况。

频繁的构造迫使开发人员在有规律的时间间隔内终结迭代——以一种可执行软件的形式来终结。构造的体验使得他们或其他任何人不再持有“已完成90%”的态度。当代码或其他制品（附录C）的数量意味着产品几乎完成时这种态度就会出现。然而，在缺乏可以运行的构造时，最困难的工作可能还在后面。这些问题在集成和测试系统之前可能仍然不会发现。相反，因为进行连续的迭代，它所产生的一系列结果能准确表明该项目的进展状态。

即使开发人员在早期迭代过程中未能得到计划的结果，他们仍有时间在后续的内部版本中

重试和改进模型。因为他们首先着手处理关键问题，所以有不少机会改进他们的解决方案。

图5-2（最初在[5]中提出）中的粗线表明迭代开发是如何进行的。在进度表的开始，这时项目代码只有2%到4%，将一个增量（或构造）组合起来。这种尝试会出现一些问题，由图中进度线的下降表示，但它们很快得到解决，项目继续向前进行。此后，项目进行频繁的构造。每次构造都可能导致进程中一次暂时的停顿，因为与整个产品的最终集成相比，增量相对较小，所以恢复起来很快。

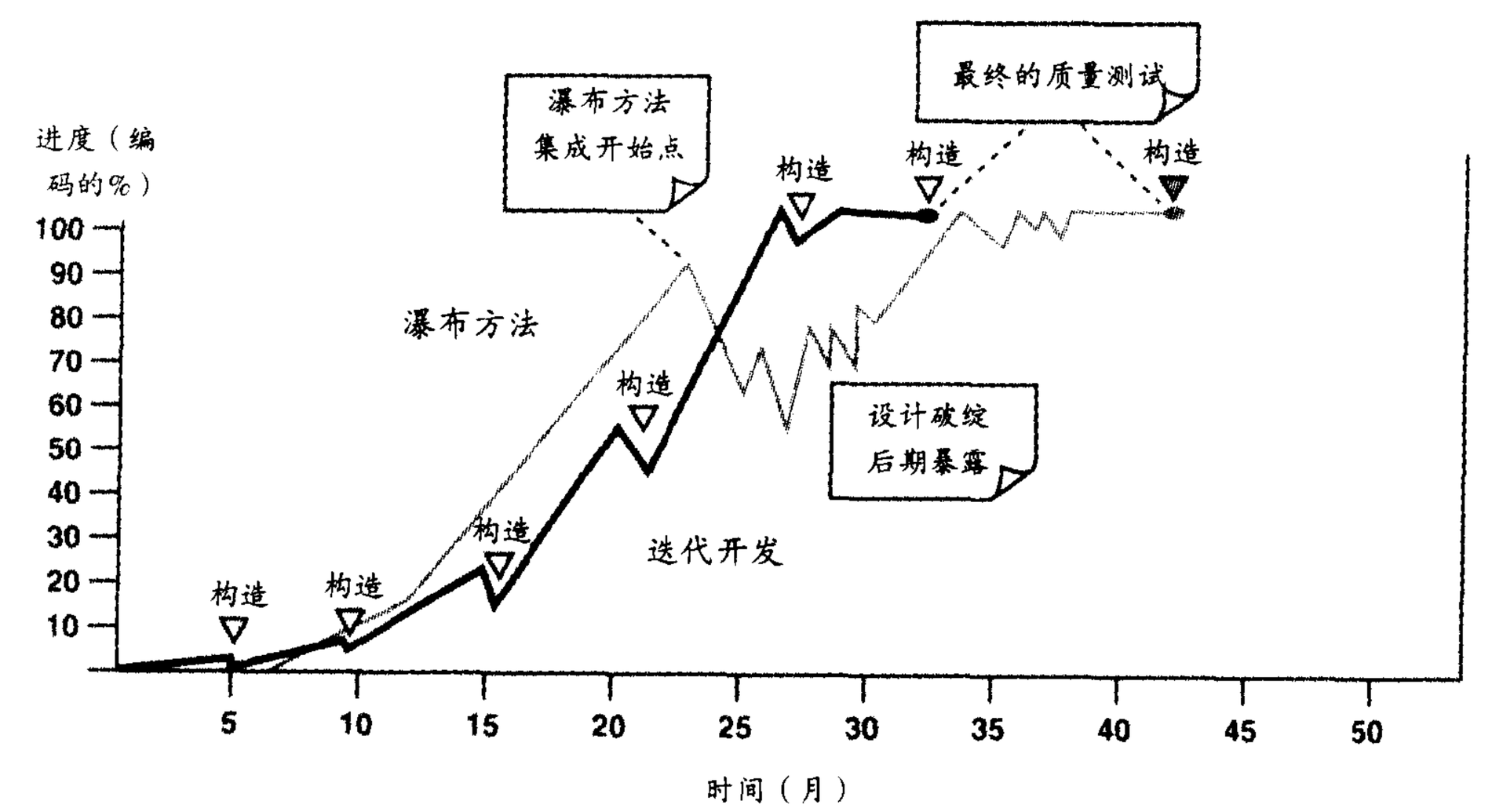


图5-2 在瀑布方法中（细线），开发人员直到完成了需求、分析和设计才开始实现工作。他们反映实现过程进展顺利，因为没有中间构造来说明他们的工作有误。实际上问题隐匿得很深，直到集成和测试阶段才完全暴露出来（设计破绽后期暴露）。在迭代开发中，实现开始得较早，频繁的构造不仅能尽早发现问题，而且发现的问题数量不大，也易于处理

如图所示，在瀑布方法中，接近交付日期时一次性的集成暴露出大量问题。此时因为问题太多和不可避免的忙乱，导致很多问题的修正无法考虑周全。查找并纠正问题常常耽搁了按计划日期交付。因此，迭代开发能够比瀑布开发在更少的规划时间内完成；而且，“瀑布项目”的产品可能是脆弱的，即它可能难以维护。

5.2.6 尽早得到有关知识

经过几次迭代之后，开发组的每个人都对不同工作流的含义有了深入的理解。他们知道有了需求后该做什么，分析后该做什么。大大降低了“分析停顿”（分析花费的时间太多）的风险。

此外，培训新的人员会更容易一些，因为他们能够通过工作本身得到训练。项目无须只为帮助人们理解过程是什么而设置专门的演练。他们可以直接进入与任务相关的工作。假如他们已经得到了适当的培训而且与前期已进入该项目的一些人共事，他们很快就会跟上进度。如果

新手不能理解要点或做错了，他们的错误对整个项目的最后进度也不会造成重大影响，因为在进行下次构造时问题就会显现出来。

迭代方法还有助于项目处理非技术性风险，如组织风险。例如，开发人员可能没有尽快了解：

- 如何使用对象请求代理建立应用程序。
- 如何使用测试或**配置管理**（附录C）的工具。
- 如何根据软件开发过程进行工作。

在项目迭代过程中，开发组开始了解这些新技术、工具和过程。在后续迭代中随着该小组更多地使用它们，对其更加精通。随着项目通过迭代有了进展，开发组逐渐发展壮大，可能开始只有5～10人，然后增加到25人，最后达到上百人。随着开发组的逐渐发展，可以利用开发组的核心成员对新加入的组员进行指导。迭代方法允许最初的开发组在大部分开发人员加入前对过程和工具进行适当调整。

通过阶段性的迭代工作，开发人员能够更好地满足实际用户需要和降低风险。通过增量式构造，所有有关人员都能够注意他们的进展程度。通过减少后期困难，加速了上市时间。此外，这种迭代方法不仅对开发人员，而且最终对用户和他们的经理都是有益的。经理们通过关注完成的迭代能够看到实际的进展。

5.3 迭代方法是风险驱动的

风险是威胁或阻碍项目成功的一个不定因素。它是“一个项目将经历不希望的事件（如进度延期、成本超支或彻底放弃）的可能性”（参见[4]中的术语表）。

依据风险及其重要性的先后顺序确定、排序并执行迭代。在以下几种情况下都需要这样做：在评价新技术时；为满足客户的需要——需求（无论是功能性的或非功能性的）而工作时；在早期阶段正在建立一个健壮的构架时（它允许进行改变而不必冒重新设计某些部分的风险）。的确，我们可以组织迭代来减少风险。

其他严重的风险是性能（速度、容量、准确度）、**可靠性**、可用性、系统界面统一性、适应性和**可移植性**（附录C）方面的问题。许多这类风险直到实现和测试底层功能时才暴露出来。这就是为什么需要通过迭代首先在初始和细化阶段、然后一直到编码和测试阶段都要探查风险的原因。这样做的目的就是在早期迭代中确定风险。

一份值得关注的观察结果表明：原则上，所有技术风险都能够映射到用况或用况脚本。这里，映射（*map*）的意思是：如果具有功能性和非功能性需求的用况得以实现，就能够降低风险。这不仅适用于与需求和构架有关的风险，而且适用于检验底层的硬件和软件。通过仔细选取用况，我们能够覆盖构架的全部功能。

降低风险是初始和细化阶段迭代工作的核心。在构造阶段，在很大程度上风险已经降低到了一个常规的级别，即已经能够由一般的开发活动来解决。我们应设法对迭代过程进行排序，

以使每个构造可以在前一个构造上建造。这样做可以设法避免风险，特别是那些因迭代顺序不妥所造成的风险，此时可能需要重新进行前面的几次迭代。

5.3.1 迭代降低了技术风险

风险可以划分为很多种（[3]和[4]）。然而，对我们来说只需提出一些就足够了，不必一一枚举。风险可以确定为四大类：

1) 与新技术有关的风险：

- 进程可能要分布到很多节点上，可能会产生同步问题。
- 有些用况可能依赖还未充分发展的计算技术，例如，自然语言识别或Web技术的使用。

2) 与构架有关的风险。这些风险非常重要，所以在我们设计统一过程时以一种准则来处理它们；即统一过程的细化阶段和在该阶段中的构架迭代给出了该过程中明确处理构架的地方。通过尽早建立能承受风险的构架，可以消除不易适应于变化的风险；从而消除了后面不得不大量返工的风险。这个能抗风险的构架是健壮的。良好的应变能力是构架**健壮性**（附录C）的特征。及早获得一个健壮构架的另一个好处是表明在哪里适合采用可重用构件，使我们在项目早期就可以考虑购买而不是自己开发这些构件。它还可以降低太迟发现系统过于昂贵而无法构造的风险。例如：

- 最初选择的用况无法帮助我们确定需要使用后面的用况进化系统的子系统结构。例如，在细化阶段的早期迭代中，可能没有注意到多个参与者将通过不同的接口使用同一个用况。如“取款”的几个接口：一个是使用图形用户界面和个人电脑；另一个是在网络上使用通信协议。如果设计时只满足了这些用况中的一个，所得到的构架可能没有内部接口来增加新的交互方式。此处的风险是将难以进化该系统。
- 某些为重用设计的框架（附录C）实际上没有在创建它们的原始项目之外应用过。此处的风险是这样一个框架不能很好地与其他框架配合或它并不易于重用。
- 计划使用的新版操作系统可能没有达到我们必须依赖的质量要求。此处的风险是在等待供应商升级该操作系统时，可能需要推迟我们自己的软件的发布时间。

3) 与创建适当的系统（支持其任务和用户）有关的风险。这种风险强调了确定功能性需求和非功能性需求的重要性，本质上意味着确定具有合适用户界面的恰当用况。尽早确定最重要的功能并确保尽早予以实现是非常重要的。这里要按照满足用户需要和性能需求的重要性来安排用况的顺序。我们综合考虑行为和能力（如性能）。在选择用况时，我们根据它们的潜在风险（例如用况可能存在性能问题）来决定处理它们的顺序。特别在初始和细化阶段，某些需求（和表达它们的用况）和其中隐匿的风险之间密切相关。开发组选择的用况会影响到所开发的构架。例如：

- 用况“跟踪呼叫”可以使电话用户将呼叫重新定向到另一个号码。是否将这种重定向应用于所有的呼叫呢？如何处理“唤醒呼叫”？电话用户很可能就在他或她的基本号码处，因而不希望呼叫改向。

4) 某些与性能有关的风险。例如：

- 用况响应时间必须小于1秒。
- 并发用况实例的数量每小时超过10 000个。

对诸如此类问题的确定，很大程度上依赖于人们的经验。因为一个人不可能具备所有这些必需的经验，所以有必要由一些人研究所提议的系统，列出可能出现的问题，并集中在一起进行风险标识。这些讨论的目的不是去解决问题，只是为了确定问题并且区分它们在初始和细化阶段的迭代过程中需要进一步研究的次序。

5.3.2 管理对非技术性风险负责

非技术性风险是那些机警的管理人员能够发觉并解决的风险。这类风险的实例包括：

- 开发组织目前缺少在待开发项目的某些特殊方面有经验的人员。
- 开发组织计划使用一种新的语言来实现系统的某些部分。
- 客户方提出的进度看起来非常紧，除非每一步都能够毫无差错地顺利完成。
- 只有在以前没有接触过的承包商能及时交付特定的子系统时，开发组织才能够满足用户提出的进度要求。
- 客户不能在交付日期规定的时限内改变某些已经核准的东西。

这类风险超出了本书的范围。在此只需说明：软件组织应该确定它们，应制定管理措施跟踪每个风险范围内的开发工作，并确保在这类风险发生时具体负责的经理能够采取行动去缓解风险。

5.3.3 处理风险

一旦确定了风险并对其排定先后次序后，开发组下一步就是决定如何逐个进行处理。开发组基本上有四种选择：避免、限制、减轻或监控。

- 也许通过重新规划项目或改变需求，有些风险能够而且应当得到避免。
- 其他的风险应加以限制，即进行约束，以使这些风险只影响项目或系统的一小部分。
- 有些风险可以通过试验看它们是出现还是消除而采取缓解措施。如果风险出现了，有利的一面是开发组对风险有了更多的了解，就可以找到一种方法去避免、限制或监控它。
- 然而有些风险无法缓解，开发组只能监控它们，看它们是否出现了。如果某个风险确实出现了，开发组必须按照**应急计划**（附录C）进行处理。如果出现了“项目杀手”型风险，必须保持镇定并仔细考虑我们的处境：把这个项目继续进行下去，还是应该取消计划呢？此时，我们只花费了有限的时间和金钱。我们知道，一个“项目杀手”型风险是可能发生的，这就是为什么要进行早期迭代的原因。所以，我们做的是很有益的工作，可以在所有开发人员进入项目前发现这种量级的风险。

处理风险需要花费时间，避免或限制风险需要重新进行规划或返工。减轻风险可能需要项目小组构造一些东西来暴露风险。监控风险涉及到选择建立并执行一种监控机制。减轻或监控风险需要付出相当大的开发工作量（即时间）。因为处理风险要花费时间，一个项目组织很少能同时处理所有风险，这就是为什么区分迭代先后次序是必要的。这就是我们所说的风险驱动的

迭代开发，这是一种有效的风险管理方法。

5.4 通用迭代过程

正如我们已经看到的，因为开发人员在每个阶段面临的挑战不一样，因此开发周期中不同阶段的迭代过程明显不同。这一节的目的是从通用的观点介绍迭代的概念：它是什么，如何进行规划，如何确定其次序，一次迭代的结果是什么。在第三部分，将分章介绍四个阶段中每个阶段的迭代过程。

5.4.1 迭代是什么

一次迭代（*iteration*）是一个能够产生内部版本的袖珍项目——或多或少要经历所有的核心工作流。这是对迭代概念的一种直觉认识。然而，为了能够描述迭代中所进行的工作而不只是停留在表面，我们对这一定义进行扩展。

可以把迭代看作是一个工作流，这表明它是一种使用和生产制品的工作人员（附录C）之间的协作。在统一过程中，我们区分**核心工作流**和**迭代工作流**（附录C）。到现在为止，我们已经熟悉了五种核心工作流：需求、分析、设计、实现和测试。这些核心工作流只是出于教学目的，帮助我们描述迭代工作流。因此，关于核心工作流的构成没有什么奇妙的；采用另外一组核心工作流可能同样简单，例如把分析和设计结合在一起的核心工作流[㊀]。核心工作流被用来简化对更具体工作流的描述，就像抽象类有助于描述具体类一样。这些更具体的工作流就是迭代工作流。第6～11章将详细描述核心工作流，第12～16章将介绍使用核心工作流来描述迭代工作流。

在图5-3中，我们描述了每种迭代工作流的一般元素。它们都经历了五种核心工作流，而且都是以规划活动开始、以评估结束。在第三部分，我们将介绍四种原型迭代，每一种针对统一过程的一个阶段。每次迭代都重用了核心工作流的描述，只是方式不同。

与传统的瀑布模型相比有什么不同呢？每种核心工作流都是一组工作人员与制品间的协作。但迭代过程间存在交叠。工作人员和制品可能参与多种核心工作流。例如，构件工程师参与分析、设计和实现等三个工作流。最后，通过对所选的核心工作流子集进行彼此叠加、并添加如规划和评估等特定的活动，来创建迭代工作流。

早期迭代侧重于了解问题和技术。在初始阶段，迭代过程关注的是获得一个业务案例[㊁]。在细化阶段，迭代的目的是进行构架基线的开发。在构造阶段，迭代过程通过每次迭代中的一系列构造创造产品，直到得到准备交付给用户组织的产品。但是，每次迭代都按照同样的模式，如图5-3所示。

每次迭代在其结束时进行评估。其中的一个目的就是判定是否提出了新的需求或者现有需

㊀ 工作流不应与并发过程相混淆。工作流是对创建描述有帮助的协作过程。

㊁ 在初始阶段，当研究具体的技术问题时，一次迭代可能会遵循一个简化的工作流。

求的变化是否影响到后续的迭代。在规划下一次迭代的细节时，开发组还会考察仍然存在的风险将如何影响后续的工作。

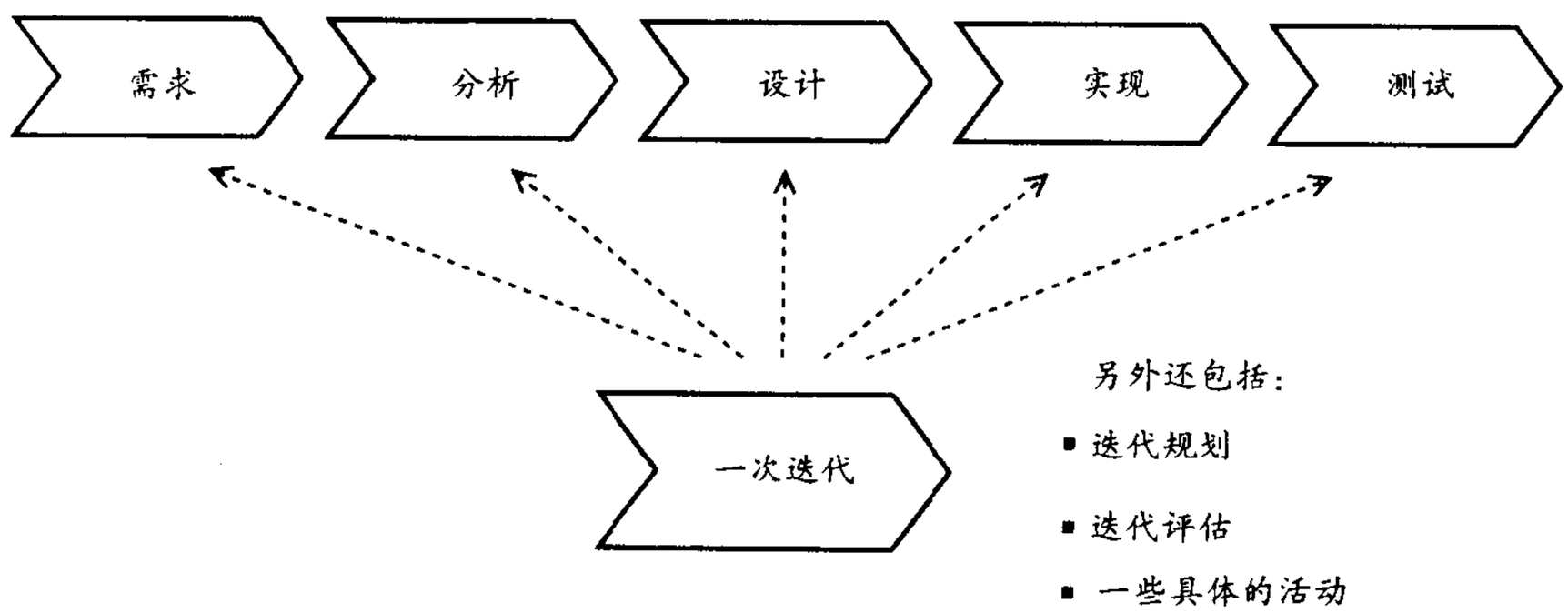


图5-3 每次迭代都要经历五种核心工作流。它由规划活动开始，以评估结束

这里值得特别强调的一个活动是**回归测试**（附录C）。在完成一次迭代前，需要确保不会破坏以前迭代中所实现的系统的其他任何部分。回归测试在迭代、增量的生命周期中尤为重要，因为每次迭代都会对前面的增量进行相当的补充和修改。我们注意到如果没有合适的测试工具，完成如此大规模（因为每次迭代都产生一个构造）的回归测试是不现实的。

项目经理在当前迭代的目标未达到之前不应同意开始下一次迭代。否则，就会导致必须改变计划来适应新的情况。

5.4.2 规划迭代过程

迭代生命周期比瀑布方法需要更多的规划和思考。在瀑布模型中，所有的计划经常是在降低风险和确定构架前就预先制定好了的。制定的计划基于很多不确定性，因而缺少真实性。相反，迭代方法在初始阶段并不对整个项目进行详细规划，而只是对最初的几步进行规划。直到在细化阶段建立了事实的基础，项目组才试图对构造和移交阶段进行规划。当然，在最初两个阶段有工作计划，但不太详细。

除了在项目的最开始，通常规划工作要考虑以前的迭代结果、与新的迭代相关的用况选取、出现在下一次迭代中的风险现状以及模型集合的最新版本。它以准备发布内部版本作为结束。

在细化阶段后期，已经有了对项目的其余部分进行规划并为构造阶段的每次迭代确定详细计划的基础。其中第一次迭代的计划将会十分清晰；而后面的迭代不太详细，它们将在早期迭代中积累的成果和知识的基础上不断进行修正。同样，也应该制定移交阶段的计划，但要根据开发组从构造阶段迭代中得到的经验或教训来进行修改。这种计划能使迭代开发得到控制。

5.4.3 确定迭代次序

自然界的进化是在没有预先进行规划的情况下发生的。但软件迭代却不是这样。可以这样

认为，用况设定了一个目标；而构架建立了一个模式。记住这个目标和模式，开发人员便可以规划出进行产品开发的顺序。

规划人员设法对迭代进行排序以便得到一条顺畅的路径，这里早期迭代为后期迭代提供了认知基础。项目的早期迭代增进了对需求、问题、风险和解决方案的了解，而后期迭代得到的附加增量最终组成了外部版本，即客户产品。对规划人员来说，最终的成功是一个一直前进的迭代系列，不必因为在后期的迭代中了解到的情况而回到早期的迭代结果去修补模型。

在一次迭代将要结束而另一次迭代即将开始时，迭代过程可能出现交叠，如图5-4所示。在结束前一次迭代并准备发布时，可以开始规划并提早进行下一次迭代的工作。但是不能交叠得太多，因为上一次迭代毕竟是下一次迭代的基础。应该记住，结束一次迭代意味着在开发组内已经获得了一次结果，即对这次迭代中的所有软件进行了集成并可在内部发布。

在很大程度上，我们规划的迭代顺序依赖于技术因素。但最重要的目标是确定工作顺序，以便能够尽早做出最重要的决定，即涉及新技术、用况和构架的决定。

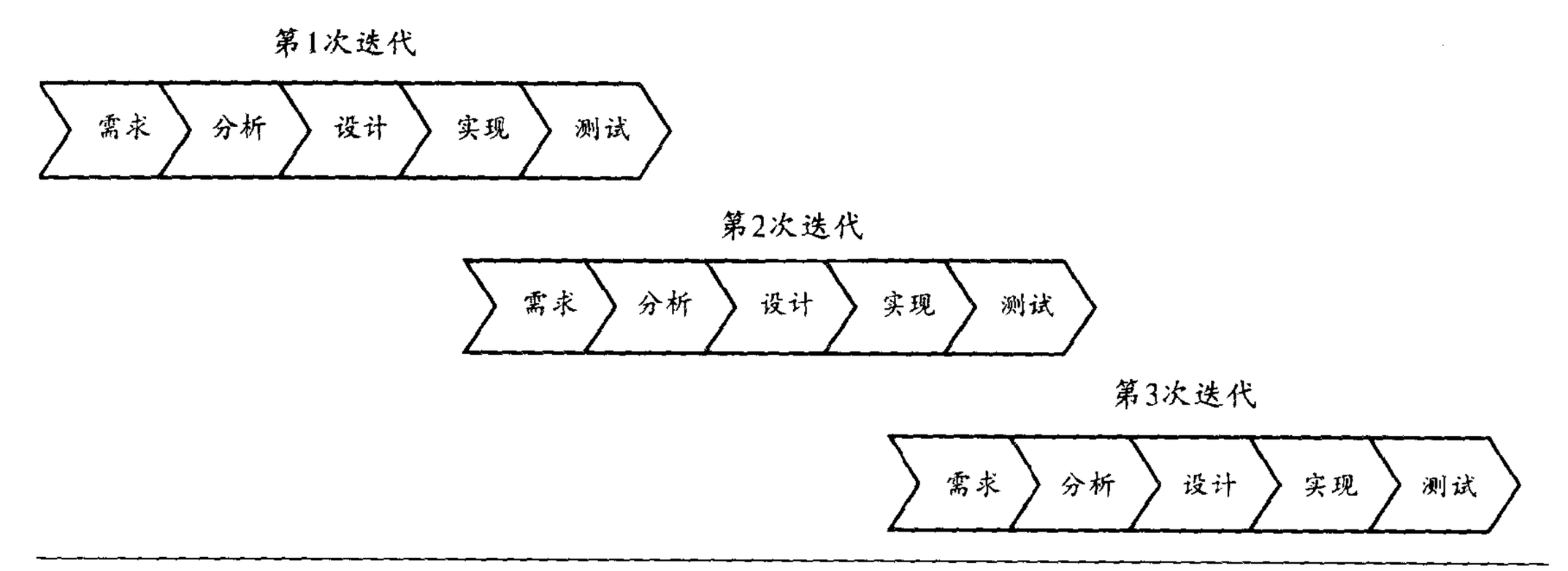


图5-4 迭代经历从捕获需求到测试的所有工作流

5.5 一次迭代产生一个增量结果

一个增量是一次迭代的内部版本与下一次迭代的内部版本之间的差别。

在迭代过程的最后，代表系统的模型集合处于一种特定的状态。这种状态或状况称为基线（*baseline*）。每种模型都已经达到一条基线；每个基本模型的元素处于基线状态。例如，每次迭代最后的用况模型均包含了一个用况集合，代表迭代过程已经实现需求的程度。这个集合中的有些用况已经完成了，而另一些用况只是部分完成了。同时，设计模型已经达到了与用况模型一致的基线状态[⊖]。设计模型的子系统、接口和用况实现也处于彼此相互一致的基线状态。为了在项目中有效地处理多条基线，开发组织需要在一条基线中维持所有制品一致的和兼容的版

⊖ 不是所有用况都需要进行设计，因此，这里术语“一致的”只是指那些经过设计的用况。

本。在进行迭代开发时，我们不能过分强调配置管理工具的有效性。

在迭代序列内的任一点，有些子系统已经完成了。它们包含所有规定的功能，而且已经得以实现并经过了测试。其他的子系统只是部分完成了，还有一些子系统仍然空着，尽管他们确实有桩程序（stubs），也可以运行并与其他子系统集成在一起。因此，用更精确的术语来说，一个增量是两次相继的基线之间的差别。

在细化阶段我们已经特别提到过建立构架基线，确定对构架有重要影响的用况，然后把这些用况按照协作来实现。这就是我们确定大多数子系统和接口（至少是那些对构架有意义的子系统和接口）的方法。只要确定了大多数的子系统和接口，就可以对它们增添血肉，即编写实现它们的代码。其中有些工作在发布构架基线前就完成了，并将继续贯穿所有的工作流。然而，大多数的细化工作是在构造阶段的迭代过程中进行的。

当临近移交阶段时，模型间和模型内的一致性程度提高了。可以通过反复完善模型来构造增量，对最终增量的集成就成为这个可发布的系统。

5.6 在整个生命周期上的迭代

四个阶段中的每一个阶段都以一个主里程碑结束，如图5-5所示[1]：

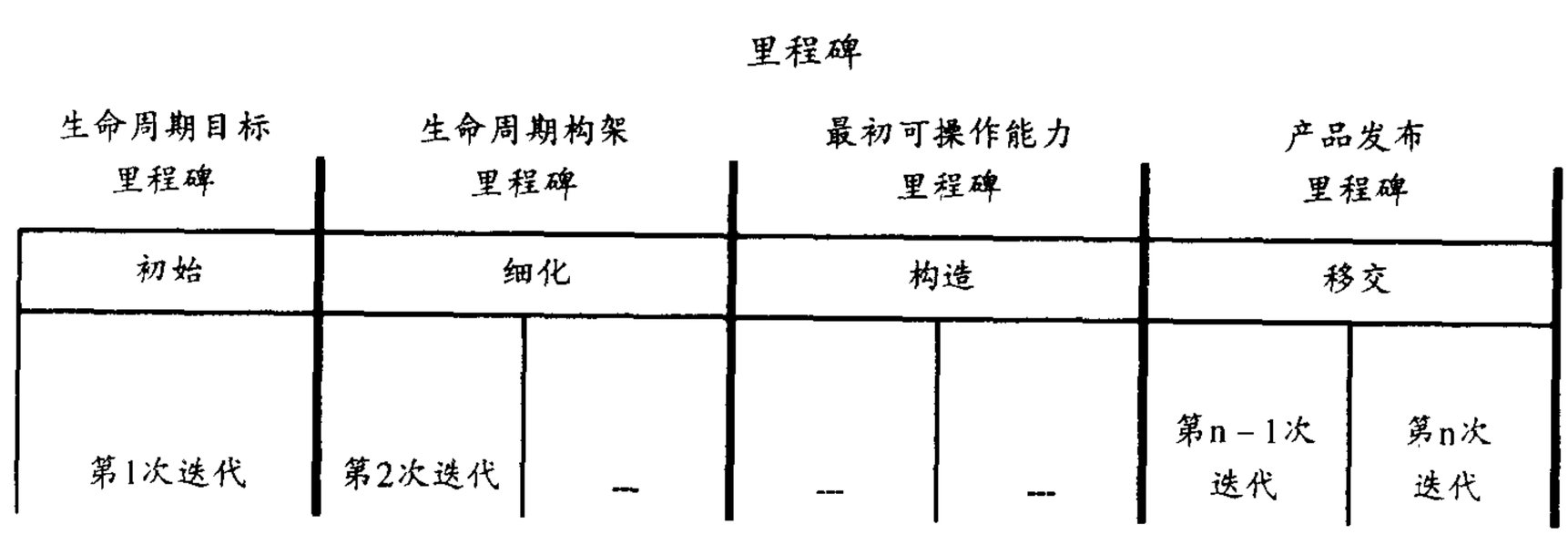


图5-5 四个阶段包含了产生主里程碑的迭代，管理层在此处做出重要的业务决定。（迭代的次数是不固定的，它是随不同项目变化的。）

- 初始：生命周期目标
- 细化：生命周期构架
- 构造：最初的可操作能力
- 移交：产品发布

每个主里程碑的目标是为了确保不同的工作流模型在产品的生命周期内以一种平稳的方式进化。“平稳的”意思是指影响这些模型的最重要的决定（即与风险、用况和构架有关的决定）是在生命周期的早期做出的。之后，应该能够高质量地以递增细节的方式进行工作。

初始阶段的主要目标是设定产品应该做什么的范围，降低最不利的（关键）风险，并建立初始业务案例，从业务的角度表明项目的可行性。换句话说，我们旨在为项目建立生命周期目标。

细化阶段的主要目标是建立构架基线，捕获大多数需求，并降低比最不利风险稍好一点的风险，即建立生命周期构架。在这个阶段最后，我们能够对成本和进度进行估计，并能够对构造阶段进行相当详细的规划。此时，我们应该能够参与投标。

构造阶段的主要目标是开发整个系统，并确保产品可以开始移交给客户，即产品达到最初的可操作能力。

移交阶段的主要目标是确保得到一个准备向用户社团发布的产品。在这个阶段，需要培训用户如何使用该软件。

在每个阶段内有更小的里程碑，也就是适用于每个迭代过程的准则。每次迭代均会产生结果（即模型制品（*model artifact*））。因此，在每次迭代的最后，用况模型、分析模型、设计模型、实施模型、实现模型和测试模型都会得到新的增量。新的增量将与前一次迭代的结果集成为一个模型集合的新版本。

前一节中曾讨论过，在每个阶段结束时的主里程碑处，经理做出继续或不继续前进的重大决定，并确定进度、预算和需求。在次里程碑处，经理和开发人员决定如何进行后续的迭代。

次里程碑（一次迭代最后发布内部版本的时刻）是朝阶段结束的主里程碑目标前进中规划好的步骤。主、次之分主要在于业务级别。开发人员反复确定风险和构造软件制品直至达到主里程碑。在每个主里程碑处，管理层对开发人员已经完成的工作进行评价。因此，每跨越一个主里程碑就代表一个重要的业务决策和按计划（至少）对下一阶段工作提供资助的承诺。可以把主里程碑看作是管理层和技术领域的结合点。

这些划分有助于管理层和其他项目相关人员对在低成本的初始和细化阶段所完成的工作进行评价，然后再决定是否进入到高成本的构造阶段。

一个软件开发项目可以粗略地划分为两大块：初始和细化阶段与构造和移交阶段。在初始和细化阶段，我们建立业务案例，缓解最主要的风险，创建构架基线，并精确规划项目的其余

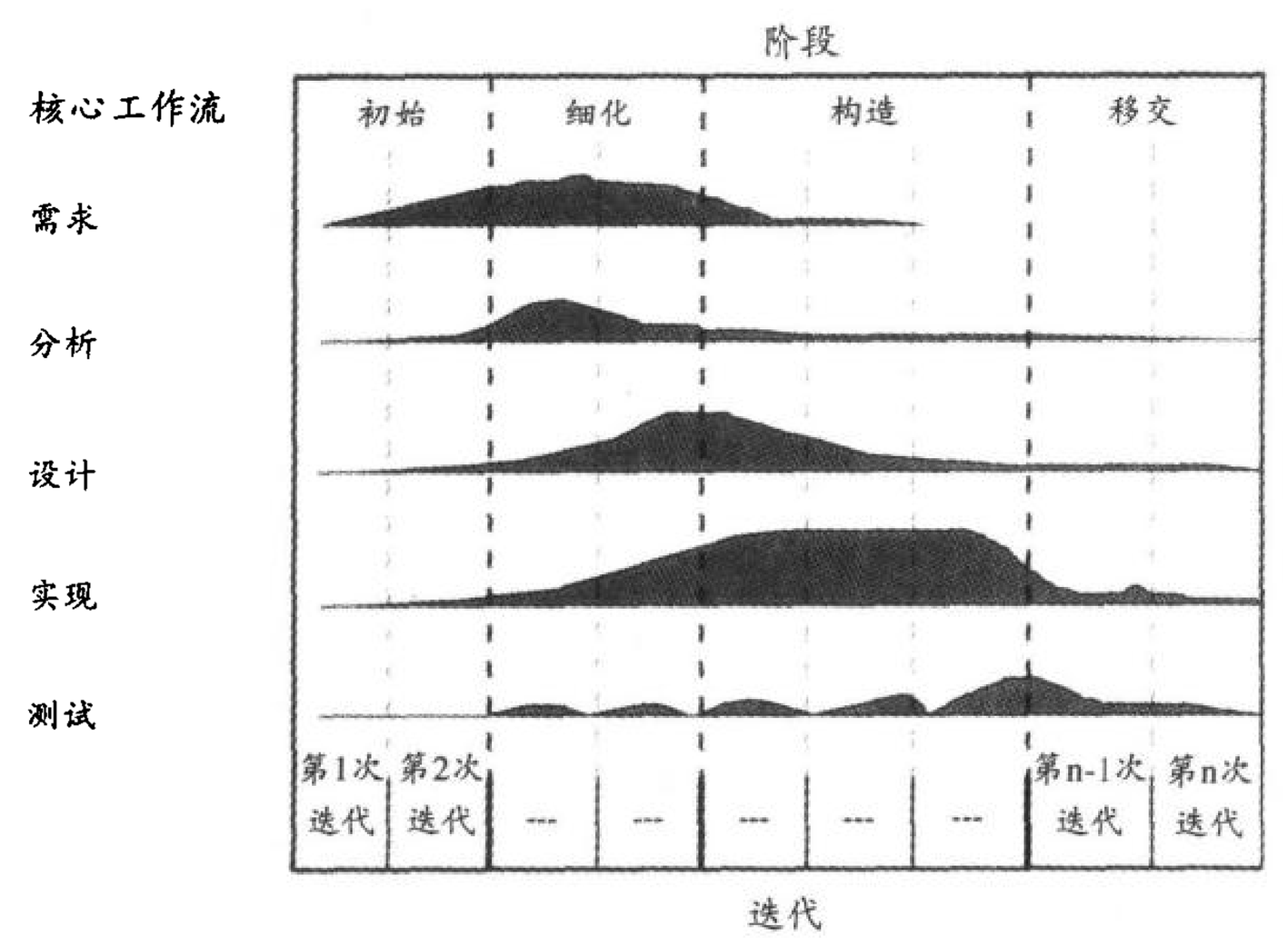


图5-6 重点在迭代过程中发生转移，从捕获需求和分析到设计、实现和测试

部分。由一个较小的、低成本的开发组来完成这项任务。

然后项目进入到构造阶段，此时的目标是实现规模经济。此时，参与项目的人数有所增加，他们通过在细化阶段建立的构架基线上进行构造来开发大部分的系统功能；并尽量重用现有的软件。

虽然每次迭代都要经历需求、分析、设计、实现和测试工作流，但不同阶段的迭代侧重点不同，如图5-6所示。在初始和细化阶段中，绝大部分工作集中在捕获需求、进行初步的分析与设计上。在构造阶段中，重点则转移到详细设计、实现及测试上。尽管图5-6中没有表示出来，但早期阶段中项目管理和为项目创造环境的任务也很艰巨。

5.7 由迭代过程来进化模型

迭代过程是以逐渐递增的方式构造出最终模型。每次迭代在经历需求、分析、设计、实现和测试时，会对每种模型增加一些内容。其中有些模型（如用况模型）在初期会得到更多的关注；而其他模型（如实现模型）则在构造期间成为注意的重点，如图5-7所示。在初始阶段，开发组也许会先创建对于支持证明概念的原型所必须的模型部分，包括（U）用况模型、（A）分析模型和（D）设计模型中最重要的元素，还包括（D）实施模型、（I）实现模型和（T）测试模型中的一些元素。在这个阶段中，大多数实现素材都是初步的，如图5-7所示，因此还有许多工作要做。

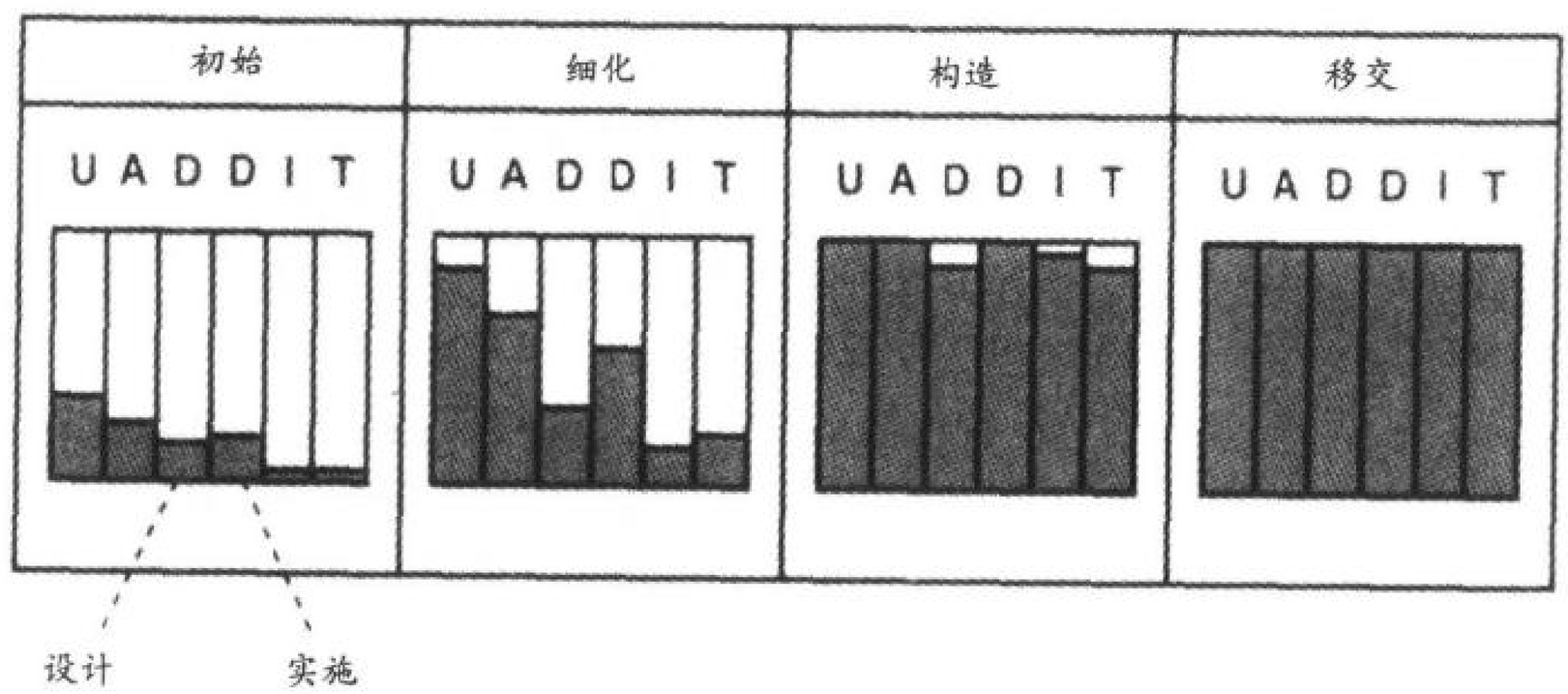


图5-7 所有模型中的工作都在四个阶段上持续进行，由模型中增加的填充色表示。构造阶段结束时所有模型基本上很完整了。但是，在移交阶段将这些模型交由用户社团使用时还需要对它们进行调整

在细化阶段，代表工作已完成的深色区域进展相当明显。而在这一阶段末期，开发组已经捕获了大约80%的用况模型（U）和实施模型（第二个D），但只有不到10%的用况已经“构造成为”系统，并对其功能进行了实现（I）和测试（T）。在细化阶段结束时，用况模型和实施模型必须是完整的。否则，我们将无法充分了解需求和实现的前提条件（包括构架），也就无法对构

造阶段进行精确规划[⊖]。

构造阶段完成了U、A、D、D、I和T的大部分工作，这是规划所预期的，因为完成的准则是一个可以移交给用户社团的完整的系统实现。之后，在移交阶段将系统移交并投入使用时，还需要进行细微的修改和调整。

5.8 迭代对开发组织具有挑战性

很多软件组织倾向于直接进入编码，因为他们的经理所关注的是代码的行数。他们往往拒绝进行变更，因为变更减缓了编码速度。他们对分析、设计或编码的重用不感兴趣，因为他们的经理关注的是新的代码。

转向迭代开发则向这些组织的工作实践提出了挑战。它要求改变工作态度。开发组织的侧重点将不得不从统计代码行转移到降低风险和建立构架功能基线上来。项目经理必须重新审视一下他们应该测量什么。他们需要通过自己的行动表明他们判断进度的依据是处理的风险、确定的用况和实现这些用况的构件。不然的话，开发人员会很快恢复到他们只注重代码行数的老传统上。

运用迭代、增量的方法要注意以下重要结论：

- 为了在初始阶段获得业务案例，开发组织应该侧重于缓解关键风险并证明概念的可行性。
- 为了在细化阶段末期进行具有业务价值的投标，开发组织必须清楚合同规定要构造什么（通过构架基线和需求来表示），并确信其中没有隐匿的风险（即未充分探明的增加成本和进度的因素）。
- 为了降低成本、减少缺陷和缩短上市时间，开发组织必须使用可重用的构件（这是在对规划的系统所在领域进行研究的基础上进行早期构架开发的自然结果）。
- 为了避免交付延期、预算超支和产品质量低劣，开发组织必须“首先抓住要害”。
- 为了避免在交付时所构造的产品过时，开发组织不能固执地认为不应作任何修改。分阶段迭代的方法使开发过程能够进一步沿着开发踪迹来处理变更。

迭代和增量的开发方法不仅需要一种新的管理项目的方式，而且需要支持这种新方法的工具。如果没有工具的支持，在每次构造和每个增量中，要处理系统中全部制品同时发生变更的情况几乎是不可能的。采用这种开发模式的组织，需要有工具来支持不同工作流，也需要有工具来支持配置管理和版本控制。

5.9 参考资料

[1] Barry Boehm, “Anchoring the software process,” *IEEE Software,* July 1996, pp.

⊖ 第4章曾经指出，在细化阶段末期，用况模型和分析模型的完成程度比图5-7所示的要低。造成这个差异的原因是，在第4章我们专门侧重于介绍构架而没有考虑其他需要完成的工作（即，要更多理解用况，以便能够开发业务案例）。

73–82.

[2] Peter F. Drucker, *Management: Tasks, Responsibilities, Practices,* New York: Harper & Row, 1973.

[3] Barry W. Boehm, "A spiral model of software development and enhancement," *Computer,* May 1988, pp. 61–72. (Reprinted in Barry W. Boehm, *Tutorial: Software Risk Management,* IEEE Computer Society Press, Los Alamitos, CA, 1989.)

[4] Capers Jones, *Assessment and Control of Software Risks,* Upper Saddle River, NJ: Prentice-Hall, 1993.

[5] Walker Royce, "TRW's Ada process model for incremental development of large software systems," *Proceedings, 12th International Conference on Software Engineering,* 1990, pp. 2–11.

第二部分

核心工作流

既然理解了作为统一过程关键实践的基本概念，我们便可以开始详细介绍每种工作流了。第三部分将讨论软件开发过程的几个阶段及其迭代过程。

在第二部分中将分章对核心工作流进行逐个介绍。第6章和第7章讨论需求；第8章讨论分析；第9章研究设计；第10章描述实现；最后在第11章介绍测试。术语核心工作流是一种抽象，已在第5章中对它进行了详细描述。在迭代期间，将着重介绍工作流的具体使用，这是第三部分讨论的主题。

按照我们的计划，分开描述核心工作流可能会误导读者，但我们希望通过解释能避免这种可能性。首先，逐一描述核心工作流会给人留下这种印象，整个软件开发过程从项目开始到结束似乎只执行一遍核心工作流序列。读者可能认为，核心工作流是一个一次性经历的过程，很像原来的瀑布过程。其次，不经意的读者可能得出结论认为，每种核心工作流都是开发过程中一个不可分割的步骤。

这些想法都是错误的。我们分章描述核心工作流只是作为一种方法来详细而全面地介绍每种工作流，完全是基于教学目的。

关于第一个问题，即可能与瀑布方法相混淆的问题。我们确实是按顺序执行这五种工作流，但每次迭代中都要执行一次，而不是整个项目只经历一次。这样，如果在四个阶段需要进行7次迭代，就会执行这些工作流7次。退一步说，在开始的初始阶段可能不会用到全部五种工作流；也就是说，在前几次迭代期间可能不会涉及到后面的工作流，如实现和测试。但统一过程的原则很清楚，每次迭代过程中都要执行根据具体迭代需要确定的工作流。

关于第二个问题，即每种核心工作流都是开发过程中一个不可分割的步骤问题。的确，我们在描述每种核心工作流时基本没有涉及其他的工作流。但是，工作流之间是通过生产和使用彼此的制品而相互影响的。在第二部分中，我们试图通过强调每种工作流的基本活动来对工作

流的描述进行简化，这也是出于教学的目的。我们没有探究核心工作流之间活动的交替问题。当然，这些交替工作对于迭代式软件开发过程是必要的，第三部分将会对此进行详细讨论。例如，当开发人员处理某个具体的设计问题时，他可能发现需要在分析和设计工作流之间来回交替进行工作。

在第三部分，将讨论如何把第二部分中分别介绍的工作流在一个可运作的项目中组合起来。例如，一个包含部分工作流的集合可能适用于早期阶段，而在构造阶段则需要所有工作流的集合。

第6章
捕获需求：从构想到需求

某些美国土著部落世代以来一直制造着一种独木舟，这种独木舟是用挖空的园木制成的。独木舟制造者们一开始在水边寻找已经倒向水面的几英尺粗的大树。他们在大树旁边生起一堆火，并将烧热的炭平铺在圆木的顶头。烧焦的木头更容易用石头工具挖空。经过几天的雕琢，独木舟看来似乎即将完成了，造船者这时要将它拖入浅水中。在大多数情况下，第一次努力所得到的结果基本上都是船在水中翻了，因为它不能保持平衡。然后还要做更多的工作。他们使用那些粗钝的石头工具继续建造他们的船，直至有一天，有人坐在船上俯身从水中捕到了一条鱼而船没有翻。只有这时他们才认为已经完工了。这样的技艺世代相传，作为经验一直流传至今。

当一个软件开发人员的“部落”听到要开发一个新系统时，他们面临的情况完全不同。首先，开发人员不是系统的未来用户。他们无法直接得到有关“独木舟”运行情况的反馈。其次，系统的需求和约束并没有扎根于他们的经验中，虽然可能从童年时代就不断使用这样的产品。相反，他们不得不要自己去发现需求。

我们将这种发现活动称为捕获需求（*requirements capture*）。了解需要建造什么样的过程，通常是在非常困难的情况下进行的。事实上，这一点确实很困难，以致于有些项目组会在确认编码需要做什么（这是很难的一步）之前就开始进行编码（相对来说这相当简单）。

6.1 为什么捕获需求很困难

专业软件开发人员通常是为其他人而不是为自己构造软件——他们是为软件的用户构造系统。开发人员常说，“哈，用户必定知道他们需要什么。”但是，你只要稍微试着从用户那里收集需求，马上会发现他们并不是最佳的信息源。原因之一是，任何一个系统都会有很多用户（或不同类型的用户），同时每个用户只知道他或她如何使用系统，而没有任何人知道系统的整个情况。用户不知道系统作为一个整体怎样操作效率更高。大多数用户也不知道他们的哪些工作可以交由软件来完成。坦白地说，用户常常不知道需求是什么，或者不知道如何以一种精确的方式描述需求。

解决问题的传统方法是指派一名协调人员（分析人员）从每个用户那里获得需求清单，并希望分析人员能用这种方法了解全部需求，并将其整理为完整、正确、一致的需求规格说明。典型的情况是，分析人员用几百甚至几千张纸把需求记录在文档中。如果你相信人类的头脑可以以“系统必须……”的形式给出上千页需求清单、而又能做到相对一致的话，那就太不切实际了。而且，这些需求规格说明也很难转变为设计和实现规格说明。

即使有分析人员的帮助，用户也不完全理解软件系统到底应该实现什么功能，他们直到系统基本完成时才知道自己的需求。随着开发工作的进行，当用户、协调人员以及开发人员本人能看到系统将是什么样子时，才能更好地理解用户的真正需要，并提出很多改变系统的建议。这些建议多数是很好的，但要实现它们会对项目进度和成本产生重大影响。

这些年来，我们错误地认为：用户知道需求是什么，我们所要做的就是与他们交谈，从他们那里得到需求。当然，我们开发的系统要支持用户，并且确实可以从用户那里得到用户与系统交互的情况。但是，更重要的是系统要支持任务（*mission*），因为系统是为执行任务而构造的，系统应该为应用它的业务和客户提供价值。通常，很难确定或理解系统的这种价值到底是什么，有时也不可能使系统确保这种价值。更糟的是，由于世界是不断变化的，这种令人难以捉摸的价值在整个项目的过程中很可能也会改变：业务本身可能发生变化，构造系统使用的技术可能发生变化，构造系统的可用资源（人员、经费）也可能发生变化，等等。

即使明白了这些，捕获需求仍旧相当困难，软件产业界很长时间以来也一直在寻求一种好的系统过程来解决该问题。本章将集中介绍有关捕获需求的问题。

6.2 需求工作流的目的

需求工作流的主要目的是致力于开发正确的系统。要做到这一点，就要足够详细地描述系统需求（也就是系统必须达到的条件或能力），使客户（包括用户）和开发人员在系统应该做什么、不应该做什么方面达成共识。

这样做的一个最大的挑战是，我们假设客户不是计算机专家，但必须能读懂并理解需求捕获工作得到的结果。为了迎接这种挑战，我们必须用客户语言（*the language of the customer*）来描述这些结果。因此，在结果中引入形式化描述和结构并介绍系统内部运作的细节时，一定要非常小心。

需求工作流的结果也有助于项目经理规划迭代和客户版本（这些将在第三部分中进行讨论）。

6.3 需求捕获概述

每个软件项目都是唯一的。这种单一性源于系统、客户、开发组织、技术等情况都各不相同。同样，捕获需求的起始点也有所不同。在有些情况下，我们是从开发一个业务模型开始，或者是从一个已由其他组织开发的业务模型开始（见6.6.1节“什么是业务模型”）。在另一些情况下，软件是一个嵌入式系统，它并不能直接支持一个业务。此时可能有一个简单的对象模型（如领域模型）作为输入（见6.5.1节“什么是领域模型”）。在其他情况下，客户已经开发了一个不以对象模型为基础的完整而详细的需求规格说明，我们可以以此作为起点，并在其基础上协商对系统的改进问题。

另一个极端是，客户对他们的系统到底要做什么只有一个模糊的概念，这样的概念可能是来源于高层管理人员提出的一个景象陈述。在这两个极端之间是各种各样两者相结合的情况。我们考虑这样一种存在“模糊概念”的起始点，并介绍在本书的后续部分所使用的例子。

举例：Interbank财团考虑的计算机系统

一个假定的金融机构“Interbank财团”面临着非常规业务、新的竞争和WWW（万维网）对业务能力增长所带来的重大变化。这个财团计划开发新的应用来适应迅速变化的金融市场。它已经命令它下属的软件开发部门“Interbank Software”着手开发这些应用。

Interbank Software 决定与主要银行储户合作设计一个结账与支付系统。这个系统将通过互联网在买主和卖主之间发送订单、账单和支付费用。银行开发这个系统的动机是想通过降低支付过程费用来吸引新的客户。系统通过互联网自动处理支付请求（而不是由出纳员手工完成），银行还能降低它的工资费用。

买主和卖主的动机都是为了降低成本、减少书面工作和缩短处理时间。例如，他们再也不需要通过传统的邮寄方法传送订单或账单。账单的支付将在买卖双方的计算机之间进行处理。买主和卖主还能更好地了解他们的账单和支付状况。

起始点可能是一个模糊的景象描述，也可能是一个详细的需求规格说明；如此大的差异要求分析人员的需求捕获方法能适应不同的情况。不同的起始点带来了不同种类的风险，所以分析人员要选择一种能够最大限度降低这些风险的方法。第三部分中将详细讨论风险缓解问题。

尽管起始点不同，但在大多数情况下一些确定的步骤都是可行的，这就允许我们建立原型工作流。这个工作流包括以下步骤，实际上这些步骤不是单独执行的：

- 列举出候选需求。
- 理解系统的语境。
- 捕获功能性需求。
- 捕获非功能性需求。

我们将在下面的段落中简要描述这些步骤。

列举出候选需求

在系统的生命周期中，客户、用户、分析人员和开发人员都会提出很多好的想法，这些想法可能会成为真正的需求。我们保留这些想法的清单，把它们作为一系列候选需求，有可能在将来的系统版本中选择实现它们。当有新的项目加入时，这份特征清单（*feature list*）就会增长；当有些特征成为真正需求并转换为其他制品（如用况）时，这份特征清单便会缩减。这份特征清单仅仅是用来规划工作。

每个特征有一个简短的名字和简要的解释或定义，这些信息对于产品规划期间讨论该特征已足够了。每个特征都有一系列规划值，其中可能包括：

- 状态（例如，建议的、批准的、在实现中的、确认过的状态）；
- 估计实现成本（按照资源类型和人-小时）；
- 优先级（例如，关键的、重要的、次要的优先级）；

• 在实现特征时的风险关联级别（例如，关键的、重要的、普通的风险级别；见第5章）。

使用这些规划值与其他方面的因素（见第12章）来估计项目规模，并决定如何将项目分为一系列迭代过程。

与特征相关的优先级和风险级别可以用来决定在哪个迭代过程中实现这个特征（在第三部分中进行讨论）。此外，当对一个特征的实现进行规划时，可能跟踪到一个或多个用况或补充需求（已经简短讨论过）。

理解系统的语境

大多数参与软件开发的人员都是与软件有关的某方面的专家。但是，为了正确地捕获需求和正确地建立系统，关键开发人员——特别是构架设计师和一些高级分析人员——需要牢固把握系统所处的语境。

开发人员至少有两种方法表达系统所处的语境：领域建模和业务建模。领域模型是把语境中的重要概念描述为领域对象，并将这些对象彼此连接起来。确定这些对象并给这些对象命名能够帮助我们建立一张术语表，以确保参与该系统的每个人都能够较好地进行交流。然后在分析和设计系统时，领域对象将有助于我们确定一些类。不久大家将会看到，可以将业务模型描述为领域模型的超集，因为它不只是包含了领域对象。

业务建模的目的是描述过程——存在的或可以察觉到的——以便于了解它们。业务建模是本书唯一要用到的业务工程中的一部分[3]。目前只要了解业务工程与业务建模非常相象也就足够了，两者都有改善组织的业务过程的目的。

分析人员在建立业务模型时，了解了很多有关软件系统语境的信息，并将这些信息在业务模型中描述出来。业务模型详细说明了系统将支持哪些业务过程。除了确定业务所涉及的业务对象或领域对象以外，业务模型还要确立每个过程所需要的能力：工作人员，他们的职责以及他们将执行的操作。当然这些信息对确定用况是至关重要的，我们已简短地讨论过这一点。事实上，业务工程方法是业务应用中用来捕获需求的最系统的过程[3]。

构架设计师和项目经理共同决定是否建立领域模型，或自始至终只建立一个完整的业务模型，还是根本就不建立这两种模型。

捕获功能性需求

确定系统需求的简明方法是基于用况（在第7章中“用况”的应用更为广泛）。这些用况既捕获功能性需求，也捕获非功能性需求，而且这些需求对于各个用况都是具体的。

下面简要概述一下用况如何帮助我们捕获正确的需求。每个用户都希望系统能为他或她做些什么，也就是执行某个具体的用况。对于用户来说，用况是使用系统的一种方法。因此，如果分析人员能够描述出所有用户需要的用况，那么就清楚了系统应该做什么。

每个用况表示一种使用系统的方法（例如，为了在业务过程中支持用户）。每个用户需要几个不同的用况，每一个用况代表他或她使用系统的不同方法。捕获系统确实需要的用况，如那些支持业务或能使用户非常舒服地工作的用况，要求我们能透彻理解用户和客户的需求。为了做到这一点，需要理解系统的语境，与用户讨论各种建议等。

作为用况的附件，分析人员还必须确定在执行用况时，用户界面看起来应该是什么样子。

开发该用户界面规格说明的最好方法是勾画出几个构想草图来表明需要传递的信息，与用户讨论这些草图，并整理出具体的可视化结果或原型交由用户验证。

捕获非功能性需求

非功能性需求确定了系统的性质，如环境和实现约束、性能、平台依赖性、可维护性、可扩展性和可靠性等性质。可靠性（*reliability*）是指如可用性、准确度、平均故障间隔时间、每1000行代码中（KLOC）的缺陷、每个类中的缺陷等特征。性能需求为功能性需求增加了一些条件，如速度、吞吐量、响应时间和内存使用。大多数性能需求只与某一个用况有关，并且应该与这个用况联系起来（作为标记值）（附录A）。在实际应用中，这意味着要在一个恰当的语境中描述这些需求，也就是要在用况说明中进行描述（也许是在单独的特殊需求部分中描述）。

举例：支付账单用况的特殊需求

性能需求

当卖主发出支付账单时，在90%的情况下，系统应该在1秒内对这个请求进行验证。验证时间必须限定在10秒内，除非网络连接被断开（在这种情况下，应该通知用户）。

有些非功能性需求参照了现实世界中的现象，如银行中的账户。这些需求最初可以在系统语境模型相应的业务对象或领域对象中捕获。接着，当确定了用况及其实际运作的“概念”以后，这些非功能性需求便与这些概念相联系。这里的“概念”既是指在术语表中用于用况说明的非形式化术语（见第7章），也是指在分析模型中更加形式化的类（见第8章）。为了简单起见，在这一章中我们采取前一种情况，即这些需求与术语表中的概念有关。

最后，有些非功能性需求非常通用，不能与一个特定的用况或现实世界中一个特定的类相联系。这时，应该将它们归入到**补充需求**（附录C）列表中单独处理。

总结

为了能有效地捕获需求，分析人员需要一组技术和制品来帮助他们更好地描述系统以便加速后续的工作流。我们将所有这些制品称为需求集合（*set of requirements*）。于是，过去传统的需求规格说明将由一组制品（即用况模型和补充需求）所取代。用来设置系统语境所需要的制品是业务模型和领域模型，如图6-1所示。

要做的工作	得到的制品	
列出候选需求	特征清单	
理解系统的语境	业务模型或领域模型	
捕获功能性需求	用况模型	定义传统的需求规格说明
捕获非功能性需求	补充需求或个别用况（针对用况的具体需求）	定义传统的需求规格说明

图6-1 不同制品（右栏中所示）组成了所有需求的集合。要做的工作会影响到一个或多个制品。注意，用况还包含“因用况而异的”非功能性需求

由于需求是不断改变的，所以需要某些方法对它们进行可控地更新。在迭代过程中可以做到这一点，每次迭代都会将一些变化反映在需求集合中，但变化的数量通常会随着我们进入到构造阶段和需求趋于稳定而减少。在6.4节中将详细介绍这些内容。之后，我们将集中精力把系统语境描述为领域模型（参见6.5节）或业务模型（参见6.6节）。最后我们将讨论补充需求（参见6.7节）。

将需求捕获为用况是一个范围很大的主题，第7章将回来讨论这个主题。

6.4 需求在软件生命周期中的作用

用况模型是在几个开发增量的基础上得到的，期间各个迭代过程将增加新的用况或对已存在的用况规格说明增加细节。

图6-2显示了需求捕获工作流和得到的制品在不同阶段及相应的迭代过程中是如何呈现不同形态的（参见第12章）：

- 在初始阶段，分析人员确定了大部分用况，以限定系统和项目的范围，并详细说明最重要的用况（少于10%）。
- 在细化阶段，分析人员捕获大多数剩余的需求，以便开发人员能够估计所需要的开发工作量。目标是捕获大约80%的需求，并在细化阶段结束时描述大部分用况。（注意，此时只能将这些需求的大约5%～10%实现为构架基线。）
- 在构造阶段，捕获（并实现）其余的需求。
- 在移交阶段，基本上不再捕获任何需求，除非需求发生了改变。

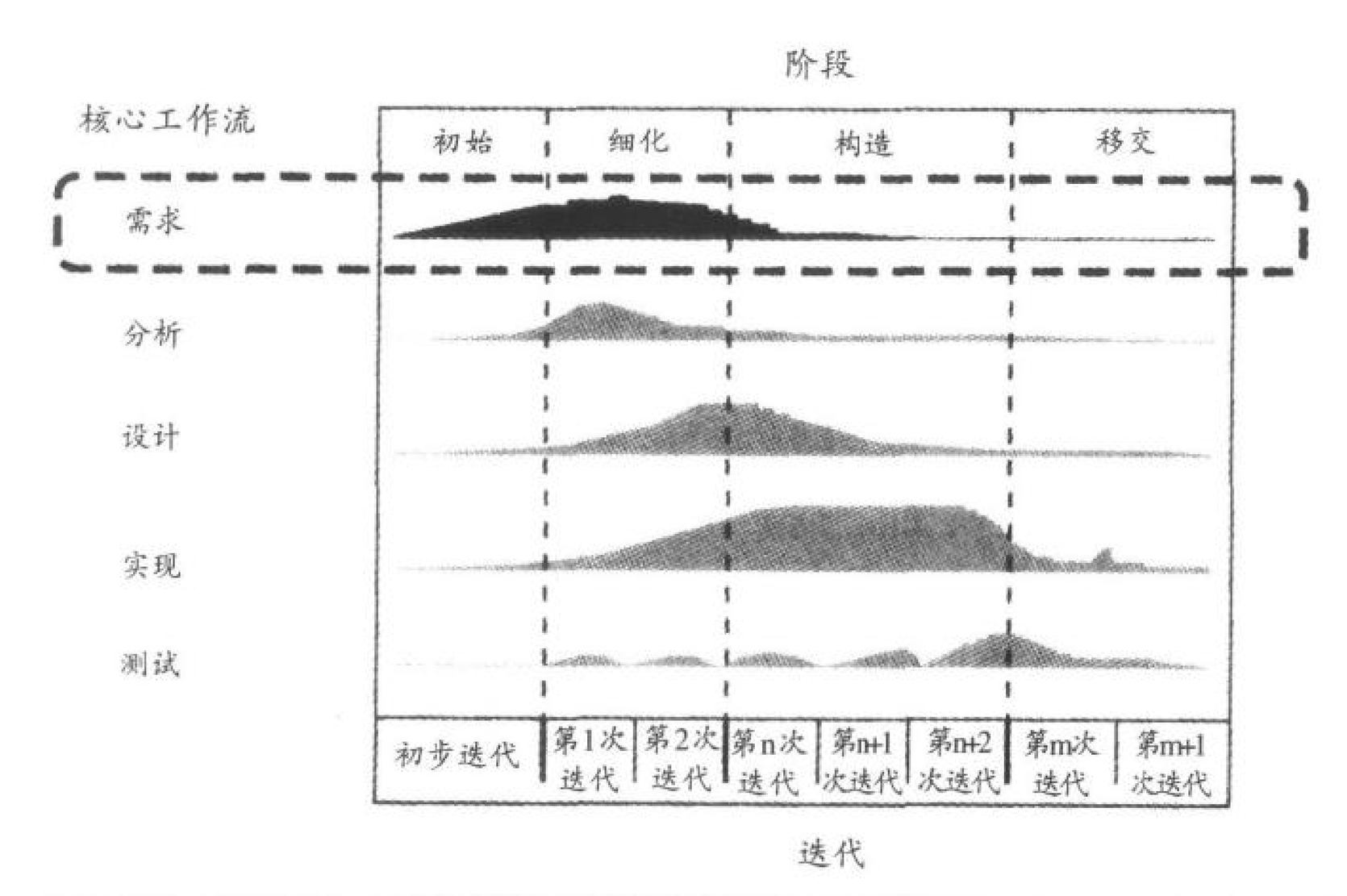


图6-2 需求主要是在初始和细化阶段获得的

6.5 运用领域模型来理解系统的语境

6.5.1 什么是领域模型？

领域模型能捕获系统语境中最重要的对象类型。领域对象代表系统工作的环境中存在的“事情”或发生的事件[2，5]。

很多的领域对象或类（用更加精确的术语）可以从需求规格说明中找到或者通过拜访领域专家得到。领域类有三种典型的形式：

- 业务对象，表示业务中可操作的东西，例如订单、账户和合同。
- 系统需要处理的现实世界中的对象和概念，如敌机、导弹及其弹道。
- 将要发生或已经发生的事件，例如飞机抵达、飞机起飞和午餐休息。

UML图（尤其是类图）描述了领域模型。

这些图向客户、用户、评审人员和其他开发人员阐明领域类，以及它们之间是如何通过关联彼此相关的。

举例：领域类：“订单”、“账单”、“条目”和“账户”

系统将通过互联网在买主和卖主之间发送订单、账单和支付费用。系统帮助买主准备订单，卖主对该订单估价并发送账单，然后买主确认账单并从买主的账户到卖主的账户实现支付。

订单是买主向卖主提出的购买多种条目的请求。每个条目在订单中“占一行”。订单具有如提交日期、交货地址等属性。见图6-3中的类图。

账单是卖主发给买主的对应于货物或服务订单的支付请求。账单具有如数量、提交日期和最后支付日期等属性。一份账单可能是多张订单的支付请求。

账单的支付是通过将买主账户上的存款转到卖主账户上实现的。账户具有如余额和所有者等属性。所有者属性标识拥有该账户的人。

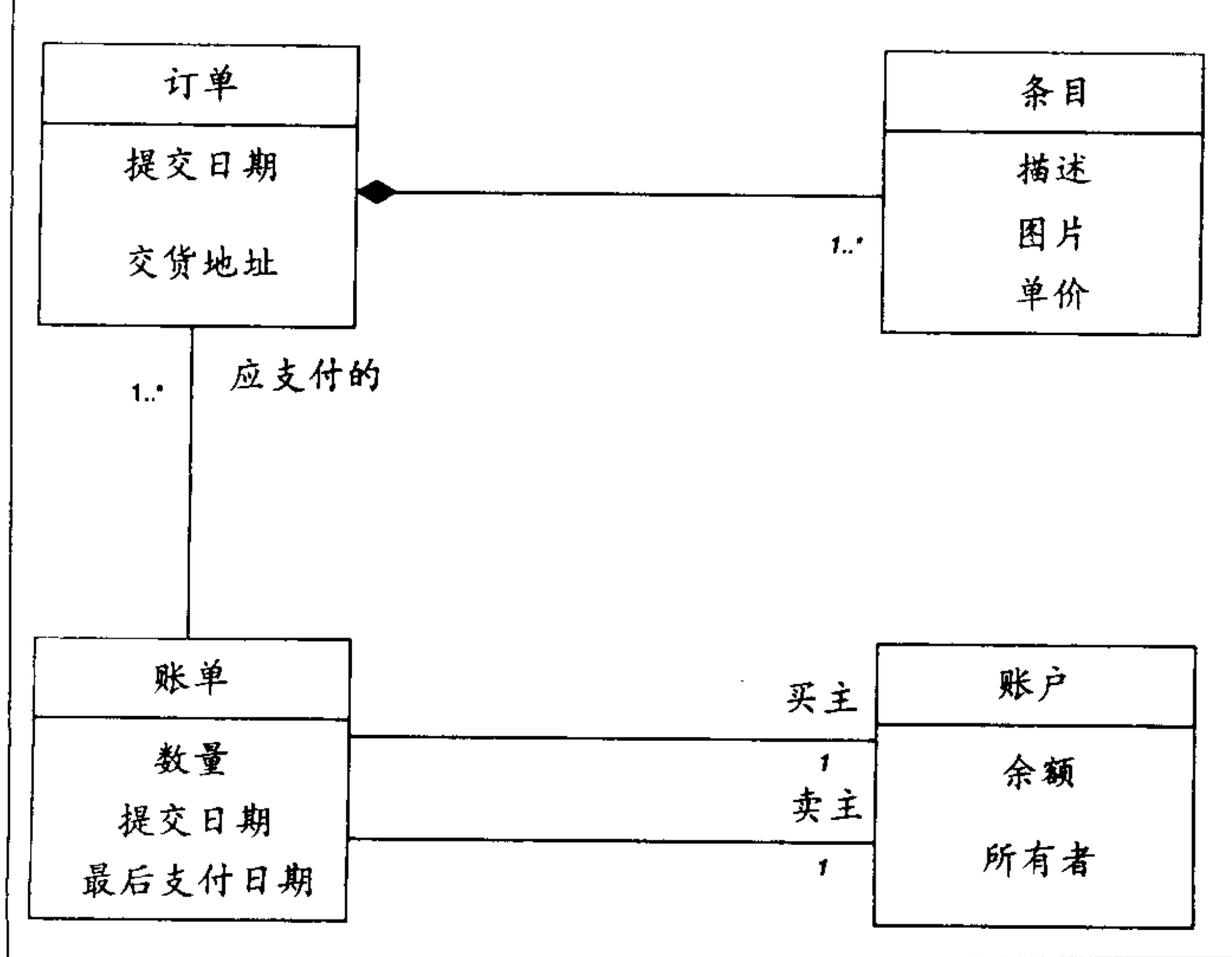


图6-3 领域模型中的类图捕获系统语境中最重要的概念

UML符号

类（矩形）、属性（类矩形下半部的文字）和关联（类矩形之间的连线）。关联路径一端的文字解释一个类相关于其他类所充当的角色，也就是关联的角色。多重性——在关联路径一端的数字和星号——表明在这一端有多少个类的对象与另一端的一个对象相联。例如，图6-3中连接类“账单”和“订单”的关联在“订单”类一端具有多重性“1..*”。这意味着每个“账单”对象可能是一个或多个“订单”对象的支付请求，如由“应支付的”关联**角色**（附录A）所显示。

6.5.2 建立领域模型

领域模型通常是在讨论会上由领域分析人员完成的，他们用UML和其他建模语言把结果文档化。为了组建一个高效的群组，这个讨论会上应该包括领域专家，还应有以建模见长的其他人员。

领域建模的目的是理解和描述在领域语境中最重要的类。规模适度的领域模型通常需要10个～50个这样的类。规模更大的领域可能需要更多的类。

由分析人员为该领域所选取的几百个候选类将作为术语表中的定义保存下来；否则，领域模型会变得很大，需要花费很大的工作量，这不是过程中这个阶段所期望的。

有时候，如对于非常小的业务领域，没有必要为其建立对象模型，只要一张张术语表也就足够了。

术语表和领域模型有助于用户、客户、开发人员和其他项目相关人员使用统一的词汇。统一的术语对与其他人共享信息十分必要。如果存在太多的混淆概念，尽管不是不可能进行设计，但也会使工程非常困难。为了开发任何规模的软件系统，现代的工程师一定要将所有参与者的语言“融合”为一种一致的语言。

最后，还有一个关于领域建模的忠告就是次序性。很容易首先对系统的内部建模而不考虑其语境 [7]。例如，一些领域对象在系统中可能具有简明的表示，而某些领域分析人员却会陷入到确定这种表示细节的陷阱里。在这种情况下，重要的是要牢记领域建模的目的是为了有助于理解系统的语境，并进而理解由该语境所产生的系统需求。换句话说，领域建模应该有助于理解系统在其语境中要解决的问题。系统解决这个问题的内部方法将在分析、设计和实现工作流中讨论（参见第8章、第9章和第10章）。

6.5.3 领域模型的使用

在确定用况和分析模型时要用到领域类和术语表。它们用于以下场合：

- 描述用况和设计用户界面时，第7章中将讨论其中的某些问题。
- 分析期间，提出所要开发的系统的内部类，第8章中将讨论其中的某些问题。

实际上，还有更加系统的方法来确定用况和发现系统内部的类——建立一个业务模型。我们将会看到，领域模型实际上是更为完整的业务模型的一个特例。因此，建立业务模型是建立领

域模型的更为有效的替代方法。

6.6 使用业务模型来理解系统的语境

业务模型是理解一个组织中业务过程的技术。但如果你所使用的系统与我们中大多数人所关心的业务无关怎么办呢？例如，开发起搏器、防抱死刹车系统（ABS）、相机控制器或电信系统时，我们应该怎么做呢？在这些情况下，我们可能还会围绕着将要开发的软件系统建立一个系统模型。这个系统（人体的一部分、汽车的一部分、相机、交换机）是嵌入式软件系统的“业务系统”。它参与了需要简要概括的较高层的系统用况。目的是确定软件用况和软件所支持的相关业务实体，所以只需建立模型来理解系统语境就行了。这项活动的结果是根据对有关“业务系统”功能的理解而导出的领域模型。

有两种类型的UML模型支持业务建摸：用况模型和对象模型[6]。两者均是在对UML具体业务扩展中定义的。[⊖]

6.6.1 什么是业务模型？

首先，业务用况模型是分别从与业务过程和客户对应的业务用况和业务参与者的角度来描述公司的业务过程。与软件系统的用况模型相似，业务用况模型从使用的角度来表示系统（这里为业务），并概括了它如何向其用户（这里指客户和合作伙伴）提供有价值的功能 [3，4，6]。

举例：业务用况

Interbank财团的例子提供了一个业务用况，它涉及在买主和卖主之间发送订单、账单和支付费用——“销售：从订单到交货”。在这个业务用况中，买主知道要买什么和从哪儿买。在以下序列中，Interbank在这个业务用况中扮演经纪人，将买主和卖主彼此联系并为账单的支付提供以下安全例程：

1) 买主订购货物或服务。

2) 卖主交付货物或提供服务。

3) 卖主给买主开账单。

4) 买主支付费用。

在这种语境中，买主和卖主是Interbank的业务参与者，他们使用Interbank所提供的业务用况。

一种业务一般提供很多个业务用况，Interbank的业务也不例外。我们将讨论其中两个用况，以获得恰当的语境，但我们不讨论其他的过程。

在“借贷处理：从申请到支付”的业务用况中，银行储户向Interbank提交借贷申请，并从银行收到贷款。

⊖ 根据我们的经验，可以用扩展的活动图来建立业务模型。这比用况模型更精细，而比对象模型更易于为非计算机专业人员所理解。详见《软件学报》1997年第6期增订本“可视化建模语言VPML”。——译者注

该银行储户代表银行的普通客户。买主和卖主是银行更为具体的储户。

在“转账、取款和存款”的业务用况中，银行储户在账户间转账、从某个账户提款和向其中某个账户存款。该业务用况还允许银行储户进一步设置自动转账。

用况图描述了业务用况模型（参见第4章和第7章）。

业务对象模型是业务的内部模型。它描述了如何由一组工作人员使用一些业务实体和工作单元来实现每个业务用况。每个业务用况的实现可以用交互图（参见第4章和第9章）和活动图（如第7～11章中的工作流图）来表示。

业务实体代表如账单这样的事物，工作人员可以在业务用况中访问、检查、处理、产生或使用它。一个工作单元是一个实体集合，它对最终用户来说是一个可认知的整体。

业务实体和工作单元用于表示同一类型的领域类概念，如订单、条目、账单和账户。因此，我们可以绘制一张非常类似于图6-3的业务实体图。而后用其他图描述工作人员、它们之间的交互以及如何使用业务实体和工作单元，如图6-4所示。

每个工作人员、业务实体和工作单元可能会参与多个业务用况的实现。例如，“账户”类很可能参与所有以下三个业务用况的实现：

- 在“借贷处理：从申请到支付”的用况中，借贷得到的贷款将转入到某个账户上。
- 在“转账、取款和存款”的用况中，从账户中取款或将钱存入账户，或在两个账户间转移。
- 在“销售：从订单到交货”的用况中，将款额从买主的账户转移到卖主的账户。

举例：“销售：从订单到交货”的业务用况

在“销售：从订单到交货”的业务用况中，工作人员执行以下几步（参见图6-4）：

1) 买主通过与卖主接触订购货物或服务。

2) 卖主通过“支付处理者”向买主发送账单。

3) 卖主向买主交付货物或提供服务。

4) 买主通过“支付处理者”付款，即将款额从买主的账户转移到卖主的账户。

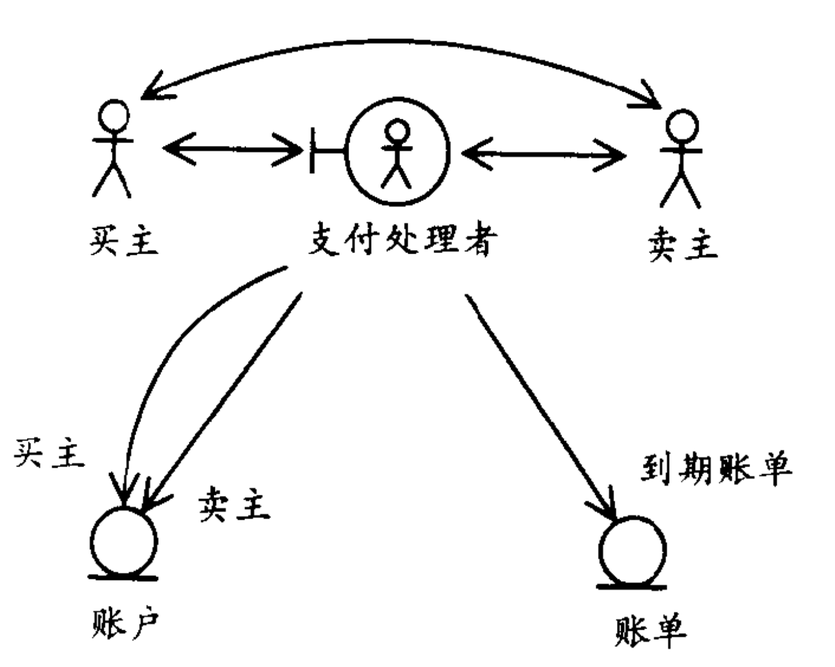


图6-4　买主、卖主和支付处理者参与了“销售：从订单到交货”的业务用况。支付处理者将存款从一个账户转到另一个由账单所指定的账户

辅助实现第2步和第4步的支付处理者是银行职员。这些任务可以由一个信息系统自动完成。

买主和卖主使用（自动的）“支付处理者”，因为工作人员能为他们提供价值。工作人员（即“支付处理者”）通过给买主发账单并跟踪未付款账单来为卖主提供增值服务。同时工作人员（即“支付处理者”）通过简化支付过程和提供更好的账单支付概况和可用性来为买主提供增值服务。

6.6.2 如何建立业务模型

业务模型分两步来建立：

1) 业务模型建模人员应该建立一个业务用况模型，用于确定业务的参与者和参与者使用的业务用况。这个业务用况模型使建模者能更好地理解业务向其参与者提供的价值。

2) 建模者应该建立一个业务对象模型，其中包括工作人员、业务实体和工作单元，三者结合在一起共同实现业务用况。业务规则和施加于业务上的其他条例与这些不同的对象相关联。目标是尽可能高效地（即迅速、准确和低成本）创建用来实现业务用况的工作人员、业务实体和工作单元。

业务建模和领域建模在某些方面非常相像。实际上，可以将领域建模看作是业务建模的一个简单变体，这时我们只关注领域类或工作人员要处理的业务实体[㊀]。因此，领域类和业务实体是非常相象的概念，我们使用这两个术语时不再加以区分。

但是，业务建模和领域建模之间还是存在一些重大差异，说得严重些，这些差异体现在更加完整的业务建模方面：

- 领域类的获得来源于一些领域专家的知识，也可能来源于与待开发系统相似的系统的有关知识（如其他的领域类和需求规格说明等）。另一方面，业务实体的得来是以业务的客户为起点，然后确定业务用况，并最终确定这些业务实体。在业务建模方法中，每个实体必须通过在业务用况中使用才能激发。这两种方法通常会得到不同的类、关联、属性和操作的集合。领域建模方法中的类可以跟踪到领域专家的经验。业务建模方法中的每个模型元素可以跟踪到客户的需要。
- 领域类具有属性，但通常没有或仅有较少的操作。业务实体则不同。业务建模方法不仅要确定实体，还要确定将参与实现业务用况（使用了业务实体）的所有工作人员。此外，我们要确定这些工作人员是如何通过每个实体提供的操作来使用实体的。这些实体所提供的操作都来源于客户并可以跟踪而返回到客户。
- 在业务建模中确定的工作人员，将作为导出所构造的信息系统的第一批参与者和用况集合的起点。这就允许我们在信息系统中经由工作人员和业务用况来跟踪每个用况，并返回到业务的客户，第7章将详细介绍这些内容。此外，每个用况均可跟踪到实现该系统的构件，

㊀ 因为领域模型是业务模型的一个简单变体，因此，在第7~11章中，我们仅仅列出后者作为各种核心工作流的输入。

如在第3章所描述的。因此我们可以得出结论：将业务建模和统一过程的软件工程方法结合起来，允许我们经过业务过程、工作人员和用况一直到软件代码来跟踪客户的需要。但是，当只使用领域模型时，在领域模型和系统用况之间没有明显的跟踪路线。

用于描述统一过程的业务建模方法(第一部分)

我们所提出的为客户公司建模的业务建模方法，基本上与我们在本书中描述的软件工程的统一过程方法相同。因此，当我们描述软件工程统一过程(参见第2章)时，对UML进行适合具体业务方面的扩展。这不仅有很强的理论基础，而且也非常实用。它是一种自举或自反的工作，揭示了业务建模方法的优点与弱点。

因此，统一过程是软件开发业务的业务用况。在软件开发业务内部，按照一系列联动的工作流对过程进行组织，或如我们所说的“实现”。这些联动的工作流包括需求(如在本章和第7章中讨论的)、分析(第8章)、设计(第9章)、实现(第10章)和测试(第11章)。每种工作流都是统一过程业务用况的部分实现，并从以下几方面进行描述：

- 工作人员，如系统分析人员和用况描述人员。
- 业务实体，或称为制品，如用况和测试用例。
- 工作单元，同样也是制品，如用况模型和构架描述。

统一过程的工作人员、业务实体和工作单元以及彼此之间的重要关系也在UML类图中进行描述。

(本方法的其他部分将在第7章中继续讨论)。

6.6.3 根据业务模型确定用况

分析人员使用业务模型作为输入，运用一种系统的技术来建立一个暂时的用况模型。

首先，分析人员把每个将成为信息系统用户的工作人员和业务参与者（即每个客户）确定为一个参与者（*actor*）[⊖]。

举例：买主参与者

买主应用“结账与支付系统”来订购货物或服务并支付账单。因此，买主既是客户也是参与者，因为他通过“订购货物或服务”用况和“支付账单”用况来使用系统去订购和付账。

每个将成为信息系统用户的工作人员（和业务参与者）都需要获得系统的支持。这种支持需求是通过遍历所有工作人员（每次确定一个）来确定的。对每个工作人员，要确定他所参与的所有不同的业务用况实现。工作人员在每个用况实现中充当一个角色，就像类在每个用况实现中充当角色一样。

一旦确定了所有的工作人员或业务参与者所充当的角色（即为它参与的每个业务用况实现确定一个角色）后，便可以为信息系统的系统参与者确定用况。业务中的每个工作人员和业务参与者对应于信息系统的一个参与者。对于每个工作人员或业务参与者角色，需要有一个对应

⊖ 在不会与业务参与者发生混淆的情况下，用术语“参与者”来代表系统参与者。

于系统参与者的用况。

因此，确定一个临时性的用况集合最简明的方法是为与每个工作人员和业务参与者的角色相对应的参与者创建一个用况。所以，对于每个业务用况，将存在一个针对每个工作人员和业务参与者的用况。然后分析人员可以对这些临时性用况进行细化和调整。

分析人员还必须决定有多少工作人员或业务参与者的任务应该由信息系统（作为用况）自动完成，并重新整理这些用况以便更好地适应参与者的需要。注意，不是所有任务都适于自动完成。

举例：根据业务模型确定用况

接着前面的例子，我们假定一个临时性的用况“购买货物或服务”，当买主在业务用况“销售：从订单到交货”中充当业务参与者时，该用况将支持参与者“买主”。经过进一步分析表明，最好是将用况“购买货物或服务”实现为几个不同的用况，如“订购货物或服务”和“支付账单”。将这个临时性用况分为几个较小的用况的原因是：买主不希望以不间断的动作序列执行“购买货物或服务”用况；而希望等到交付货物或提供服务之后再支付账单。因此，将支付序列表示为一个单独的用况“支付账单”，并于交付货物之后再执行这个用况。

至此，我们已经介绍了如何使用领域模型或业务模型为系统的语境建模。然后又介绍了如何从业务模型中获得用况模型，即得到一个捕获了系统所有功能性需求和大部分非功能性需求的用况模型。有些需求可能不与任何特定的用况相关联，称为*补充需求*（*supplementary requirement*）。

6.7 补充需求

补充需求是指不与任何特定的用况相关联的非功能性需求，每个这样的需求可能会影响到几个用况或根本不影响任何一个用况。性能、接口和物理设计需求以及构架、设计和实现等约束是非功能性需求的实例[1]。补充需求可以被捕获作为传统的需求规格说明中陈述的需求（即作为需求的一个清单）。然后，连同用况模型一起用于分析和设计。

接口需求（*interface requirement*）详细描述了系统必须与之交互的外部项目的接口，其中包括对于格式和时限的约束或其他与此交互有关的因素。

物理需求（*physical requirement*）详细描述了系统必须具有的物理特性，如它的材料、形状、大小和重量。这种类型的需求可以用来代表硬件需求，如所需的物理网络配置。

举例：硬件平台需求

服务器

SUN SPARC 20 或 PC Pentium

客户机

PC（最低为Intel 486 处理器）或 Sun Sparc 5

设计约束（*design constraint*）是对系统设计进行限制，如可扩展性和可维护性约束，或有关重用遗留系统或其中的主要部分的约束。

实现约束（*implementation constraint*）是对系统的编码或构造进行说明或限制。例如，所需要的标准、实现指南、实现语言、数据库完整性策略、资源范围和操作环境。

举例：文件格式约束

“结账与支付系统”的1.2版将支持长文件名。

举例：软件平台约束

系统软件

客户机操作系统：Windows NT 4.0、Windows 95或Solaris 2.6

服务器操作系统：Windows NT 4.0 或Solaris 2.6

因特网软件

Netscape Communicator 4.0 或 Microsoft Internet Explorer 4.0

此外，经常存在其他需求（*other requirement*），如法律方面的需求和规章制度方面的需求。

举例：其他需求

安全性

传输必须是安全的，这意味着只有经授权的人才可以访问信息。被授权的人只能是拥有账户的银行储户和系统管理员等参与者。

可用性

“结账与支付系统”每个月的停机时间必须少于1小时。

易掌握

90%的买主学会（通过所提供的逐步操作的使用说明书）提交简单订单和支付简单账单的时间不能超过10分钟。简单订单只有一个条目。简单账单是对简单订单进行支付的账单。

6.8 小结

至此，我们已经全面深入地介绍了有关捕获需求的内容；了解了业务模型和领域模型如何帮助定义系统的语境和如何从业务模型中获得用况。我们还知道了用况可用于捕获需求（下一章我们还要讨论这个问题）。在后续的章节中，将介绍如何使用用况和补充需求来帮助我们分析系统，创建系统构架，设计、实现和测试系统，以确保系统能满足客户的需求。

6.9 参考资料

[1] IEEE Std 610.12.1990.

[2] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Övergaard, *Object-Oriented Software Engineering: A Use-Case–Driven Approach,* Reading, MA: Addison-Wesley, 1992 (Revised fourth printing, 1993).

[3] Ivar Jacobson, Maria Ericsson, and Agneta Jacobson, *The Object Advantage: Business Process Reengineering with Object Technology,* Reading, MA: Addison-Wesley, 1994.

[4] Ivar Jacobson, "Business process reengineering with object technology," *Object Magazine,* May 1994.

[5] James Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen, *Object-Oriented Modeling and Design,* Englewood Cliffs, NJ: Prentice Hall, 1991.

[6] OMG Unified Modeling Language Specification. Object Management Group, Framingham, MA, 1998. Internet: www.omg.org.

[7] Alan M. Davis, *Software Requirements: Objects, Functions, and States,* Englewood Cliffs, NJ: Prentice Hall, 1993.

第7章
捕获需求作为用况

7.1 引言

需求捕获阶段的主要工作是建立待开发系统的模型，使用用况是建立这种模型的合适方法。这是因为功能性需求可以很自然地构造为用况，而大多数非功能性需求都具体对应于单个用况，从而也可以在这个用况语境中进行处理。

剩下的为多数或全部用况所公用的非功能性需求，可以保存到一个单独的文档中作为补充需求。这些需求在第6章中已经介绍过，本章不再进行讨论，后面当用于分析、设计、实现和测试工作流中时还会提到。

用况提供了一种系统而直观的方法来捕获功能性需求，并特别强调要为每个用户或外部系统增值。使用用况会迫使分析人员考虑谁是用户以及可以通过用况来完成什么业务或任务。但正如我们在第4章中所介绍的，如果这就是用况能做到的全部，并不会得到如此广泛的接受。它们在驱动其他开发工作方面的关键作用是其能被大部分现代软件工程方法所接受的重要原因[8]。

这一章将详细阐述用况和参与者，并给出比第3章中更为严格的定义。

我们按三步来描述需求工作流（在第8～11章中将以同样的方法描述所有其他工作流）：

- 在需求工作流中产生的制品。
- 参与需求工作流的工作人员。
- 需求捕获工作流，包括每个活动的细节。

首先，我们仔细观察一下图7-1中所示的工作人员和制品。

用于描述统一过程的业务建模方法（第二部分）

我们要确定参与每个工作流的工作人员和制品。一个工作人员代表一个可以赋予一个人或一个群组的职位，并规定所需的职责和能力（参见2.1.3节）。

制品是该系统中的工作人员在工作时所创建、生产、改变或使用的各种描述和信息。制品可以是模型、模型元素或文档。例如在需求工作流中，制品主要是用况模型及其用况。而在统一过程的业务模型中，制品是业务实体或工作单元。

每个工作人员负责一组制品。在图中用从工作人员到对应制品的名为“负责”的关联表示(参见图7-1)。为了使这种图更加直观，对大多数制品采用特殊的符号表示。代表文档的制品用特定的文档符号表示，代表模型或模型元素的制品用它们对应的UML符号表示。

注意，为了在统一过程的业务模型中使用这些特殊的符号，我们引入表示文档、模型、模型元素的构造型。每个这样的构造型都是《业务实体》或《工作单元》构造型的子类型。

我们将说明工作人员如何在工作流中合作，从中可以看到工作重心如何从一个工作人员转向另一个工作人员，可以看到他们在执行活动时如何处理制品(另见2.4.2节)。在这部分中，我们还将更详细地考察每个活动。一次活动是一个工作人员在工作流中执行的一项工作，即该工作人员执行的一项操作。

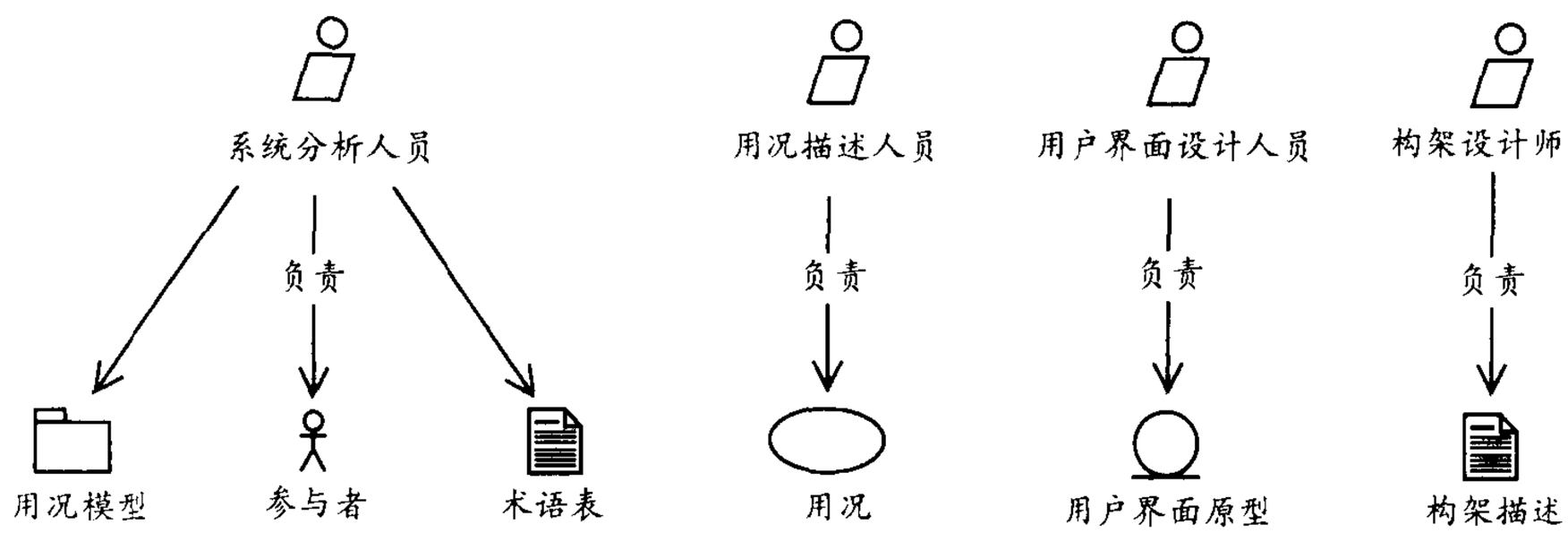


图7-1 将需求捕获为用况时所涉及的工作人员和制品

7.2 制品

需求捕获中用到的主要制品是用况模型，包括用况和参与者。可能还有其他种类的制品，如用户界面原型。

第3章曾介绍过这些制品，这里将给出更为精确的定义，它与[12]中所给出的定义是一致的。然后，我们将放宽这种形式，说明如何在统一过程的实践中应用这些构造。这里所给出的定义提供了一个坚实的基础，实践中无须拘泥于这种形式。这里所以介绍这些定义是出于以下原因：

- 在形式化描述某些类型的用况时（例如，使用活动图或状态图描述用况时），这些定义可能是很重要的。这对具有很多状态和复杂状态转移的用况特别有用。
- 更易于确定恰当的用况并一致地描述它们。实际上，即使描述时没有选用这种有效的形式化方法（如参与者或用况），在头脑中有这种概念对保证描述完整性和一致性也很有好处。
- 这对能够说明参与者和用况构造与其他UML构造的关系非常重要。

7.2.1 制品：用况模型

用况模型可以使软件开发人员和客户在需求（即系统必须满足的条件和能力）方面达成共识[6]。用况模型充当客户和开发人员之间的协议，并作为分析、设计和测试的基本输入。

用况模型是一种包括参与者、用况以及它们之间关系的系统模型（参见图7-2）。

用况模型可能非常大，你不能“一口将它吞下去”，所以需要某种方法来帮助我们从较小的

局部来领会该模型。UML允许我们用各种图来展现用况模型，这些图从不同角度、以不同目的来表示参与者和用况。还要注意，假如用况模型很大，也就是说它包括了大量的用况或参与者，在模型中引入包来控制其规模是很有用的。这或多或少是对用况模型的若干扩展，但在本书中将不再进行介绍。

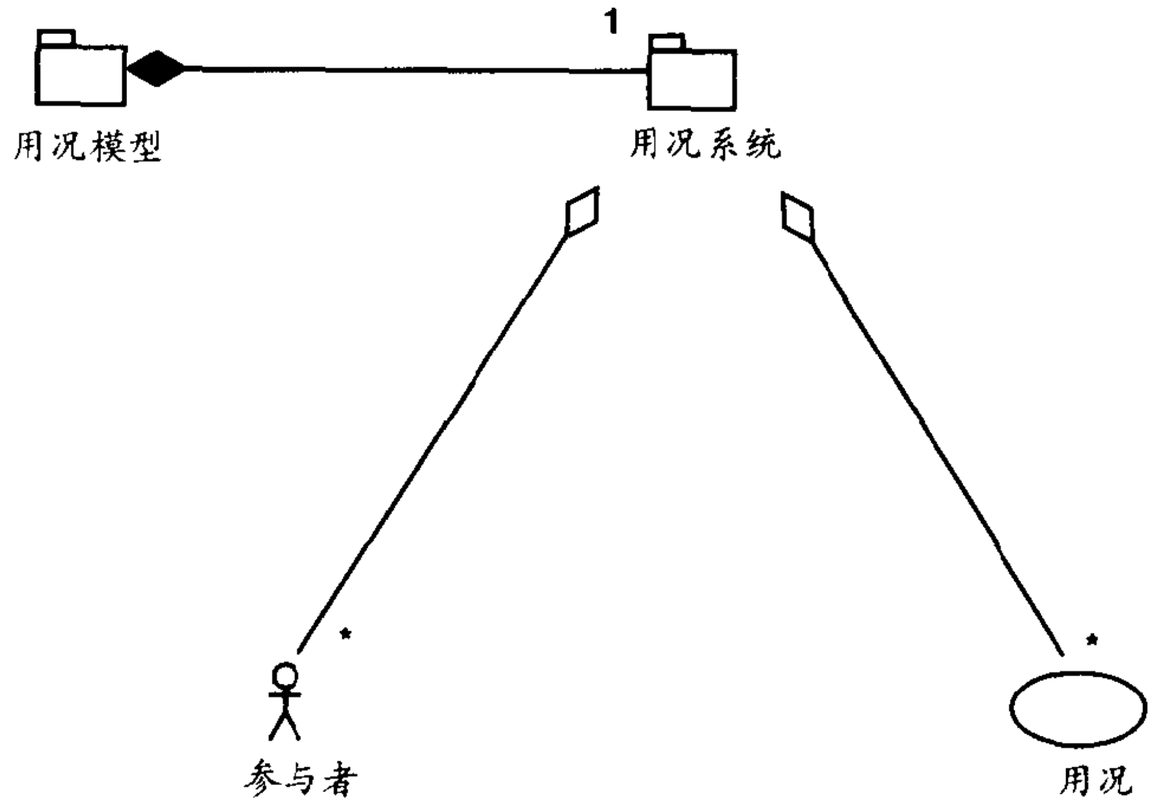


图7-2 用况模型及其内容。用况系统代表模型的顶层包

7.2.2 制品：参与者

用户模型描述了系统能为每种类型的用户做些什么。每种类型的用户表示为一个或几个参与者。每个与该系统进行交互的外部系统也表示为一个或多个参与者；这包括外部设备（如定时器），可以认为它们处于系统的外部。因此，参与者表示系统外部与系统进行协作的参与者。只要确定了系统的所有参与者，就确定了系统的外部环境。

正如第6章中所介绍的，参与者在业务中通常与工作人员（或业务参与者）相对应；每个（工作人员的）角色定义了该工作人员在特定的业务过程中要做些什么。工作人员所充当的角色可以用于导出（如果提供了适当的工具，实际上就可以生成）对应的系统参与者所要充当的角色。

举例：参与者

Interbank的“结账与支付系统”与使用该系统进行订货、确认订单和支付等任务的用户进行交互。那么，这类用户可以表示为“买主”参与者（图7-3）。

图7-3 “买主”参与者

一个参与者在与之进行协作的用况中充当一个角色。每当一个具体用户（人或其他系统）

与系统相交互时，相应的参与者实例就充当这样一个角色。因此，一个参与者实例是与该系统进行交互的一个具体用户。任何适合做某个参与者的实体都可以作为该参与者的实例。

7.2.3 用况

参与者使用系统的每种方式都可以表示为一个用况。用况是系统能够向其参与者提供增值的功能“块”。更严格地说，用况确定了一个与系统参与者进行交互、并可由系统执行的动作序列，其中包括各种不同的候选序列。

因此，一个用况是一种规格说明，它规定了动态“事物”（这里指用况的实例）的行为。

举例：“取款”用况

在第3章中曾介绍过“取款”用况，该用况使参与者实例“银行储户”可以通过ATM取款（图7-4）。

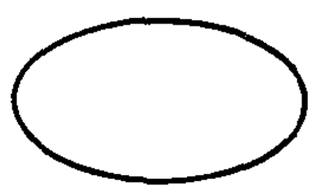


图7-4 “取款”用况

“取款”用况确定了几种可能的用况实例，即几种不同的合理方式，例如由系统执行用况，或需要与所涉及的参与者实例相交互。例如，一个叫Jack的人首先输入他的个人识别码1234，选择取款220美元，而后得到这笔钱，系统执行一个用况实例。如果Jack输入同样的个人识别码，选择取款240美元，并得到这笔钱，此时系统执行另一个用况实例。当Jack申请取款480美元时，由于账户余额不够或个人识别码错误等原因，系统拒绝该申请，这时系统执行第三个用况实例。

在UML术语表中，一个用况是一个类元，这意味着它具有操作和属性。因此，用况说明可以包括状态图、活动图、协作图和顺序图（详见附录A，“UML概述”）。

状态图从状态和状态间转移的角度说明了用况实例的生命周期。每个转移都是一个动作序列。活动图也是通过描述每个转移中发生的动作序列来更加详细地说明其生命周期。协作图和顺序图用于描述一个典型的参与者实例与一个典型的用况实例之间的交互关系。实际上，在描述用况时并不总是需要这样形式化，我们将在7.4.3节“活动：细化用况”中进行讨论。但是，头脑中对用况有这种更为精确的理解，对构造用况说明很有帮助。

一个用况实例是一个用况的行为（或执行）。另一种表达是，一个用况实例是系统“遵循某个用况”所执行的那个东西。在执行用况实例时，它与参与者实例相交互，并执行其用况规定的一个动作序列。这个序列在状态图或活动图中说明；它是该用况的一条路径。可能存在很多的路径，而且多数路径可能很相似。这些就是该用况的候选动作序列。通过一个用况的这样一条路径可以描述如下：

1) 用况实例被初始化并进入开始状态。

2) 由参与者发出的外部消息激发。

3) 通过执行一个动作序列转移到其他状态。这个序列包括内部计算、路径选择和消息输出（给某个参与者）。

4)（在新的状态）等待由参与者发出的另一个外部消息。

5)（再次）由新消息所激发，依此类推。可能要经历很多状态直至用况实例结束。

如前所述，最常见的是参与者实例引用用况实例。但也有可能是系统中的事件引用该用况实例，例如，一个设置好的定时器触发时就会出现这种情况（前提是如果将定时器看作是处于系统的内部）。

像所有的类元一样，用况也具有属性。这些属性表示在用况执行期间用况实例使用和处理的值。这些值只局限于该用况实例，即它们不能用于其他的用况实例。例如，可以认为“取款”用况具有属性“个人识别码”（PIN）、“账户”和“取款金额”。

用况实例并不与其他用况实例相交互。用况模型中唯一的交互发生在参与者实例与用况实例之间[10]。其原因是我们希望用况模型简单、直观，以便与最终用户和其他项目相关人员进行富有成效的讨论而不至于陷入细节。我们不想处理用况之间的接口、并发和不同用况实例之间的其他冲突（如共享其他对象）。我们把用况实例视为原子的，即每个用况实例要么完全执行要么根本不执行，而不受任何其他用况实例的干扰。因此，每个用况的行为都可以独立地解释，从而使用况建模更加简单。将用况视为原子的，与某个解决用况冲突的事务处理程序无关。这样做只是为了确保我们能够读懂用况模型。

我们承认在对系统的不同使用之间确实存在冲突问题。这些问题不能在用况建模中得到解决，而是要推迟到分析和设计阶段来解决（在第8章和第9章再分别讨论），那时将用况实现为类或子系统间的协作。例如在分析模型中，我们可以清晰地描述一个单独的用况如何参与几个用况的实现，以及如何解决用况之间潜在的冲突问题。

对大系统建模

本书侧重于如何对一个单独的系统建模。但在很多情况下，我们需要开发规模较大的系统，这个系统实际上是由其他系统构造而成的，我们称之为*系统的系统*。

举例：世界上最大的系统之一

人类曾构造的规模最大的系统可能有10亿个用户，那就是全球的电信网络。

例如，当你从旧金山打电话到斯德哥尔摩时，这个电话可能经过了20多个系统，包括本地交换机、国际间交换机、卫星系统和传输系统等。每个这种类型的系统大约要花费每年1 000人的成本来开发，而其中大部分是软件开发工作。

令人惊异的是，在我们打这样的电话时系统通常能正常运转。

系统如此复杂，并且涉及到不同的人、公司和国家，为什么它能正常工作呢？主要原因在于整个网络（即网络体系结构）的每个接口均由一个唯一的组织ITU（国际电信联合会）进行标准化。

ITU规定了该网络中各种类型节点间的接口以及这些接口的精确语义。

构造系统的系统所依靠的技术与用于构造全球电信网络的技术类似[9]。首先是通过其用

况来确定整个系统。接着按照相互协作的子系统来设计。然后将整个系统的用况划分为相互协作的子系统的用况，这些子系统通过接口相互联系。对这些接口进行精确定义，之后每个不同的子系统可以由不同的开发组织独立进行开发（本身作为一个系统）。UML支持这种构架，对统一过程进行扩展可以用来开发这类系统[13]。

实际上，用况不但可以用于确定系统，还可用于确定其他较小的实体，如子系统或类。子系统或类又包括两部分（每部分描述一个视角）：规格说明和实现。规格说明根据用况描述子系统或类为其环境提供什么功能；实现部分描述子系统或类如何进行内部构造以便实现其规格说明。这种环境通常是由其他子系统和类组成的。不过，如果我们不想明确提到环境，也可以由参与者来代表。

当我们希望把一个子系统当作一个系统看待时，就可以使用这种方法。例如：

- 希望使用与开发其他子系统所用技术不同的技术来开发该子系统。只要该子系统提供了恰当的用况，并使用和支持确定的接口，我们就可以做到这点。
- 希望与其他子系统分开来管理该子系统——也许地理位置是相互分离的。

7.2.3.1 事件流

可以将每个用况的事件流捕获为该用况的动作序列的单独文本描述。因此，事件流规定了在执行确定的用况时系统要完成的工作。事件流还规定了在执行用况时系统如何与参与者进行交互。

从管理的观点来看，一个事件流描述包括一个动作序列的集合，该动作序列适于修改、评审、设计、实现和测试，并适合作为用户手册中的一节或一小节来描述。

7.4.3节“活动：细化用况”中给出了一个用况事件流描述的例子。

7.2.3.2 特殊需求

特殊需求是汇集了一个用况的所有需求（如非功能性需求）的文本描述。它们主要是与用况有关的非功能性需求，需要在后续的工作流（如分析、设计或实现工作流）中进行处理。

7.4.3节“活动：细化用况”中给出了一个与用况有关的特殊需求的例子。

7.2.4 制品：构架描述（用况模型视图）

构架描述包括用况模型的构架视图，描述了对构架重要的用况（如图7-5所示）。

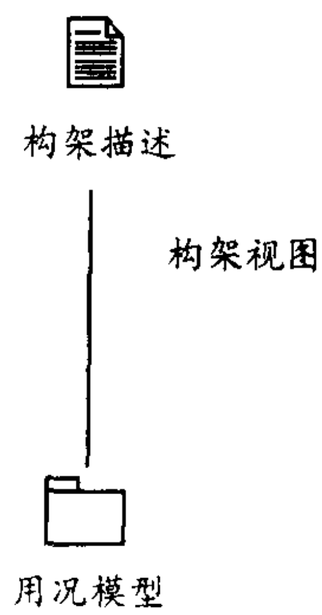


图7-5 构架描述

用况模型的构架视图应该包括描述某些重要和关键性功能的用况，或涉及某些必须在软件生命周期中尽早实现的重要需求的用况。该构架视图在区分迭代中用况开发（即分析、设计和实现）的优先次序时用作输入。该内容在第一部分和第三部分中有详细的描述（分别参见第4章的4.3节和第12章的12.6节）。

通常，相应的用况实现可以在分析和设计模型的构架视图中找到（参见第4章的4.5节，第8章的8.4.5节以及第9章的9.3.6节）。

7.2.5 制品：术语表

术语表可以用来定义分析人员（和其他开发人员）在描述系统时所用的重要术语和公用术语。术语表有益于开发人员之间就各种概念和观点的定义达成一致，以便能从总体上降低由于误解而产生的风险。

术语表经常可以从业务模型或领域模型中导出，由于它不是很形式化（它不包括类或明确的关系），因此，它更容易维护，而且能更直观地与外部合作伙伴（如用户和客户）进行讨论。此外，术语表更侧重于所构造的系统而非系统的语境，业务模型或领域模型通常就是这样。

7.2.6 制品：用户界面原型

用户界面原型有助于在需求捕获期间理解和确定由人充当的参与者和系统之间的交互。这不仅有助于开发更好的用户界面，而且有助于更好地理解用况。在描述用户界面时，可能还会用到其他制品，如用户界面模型和屏幕示意图。

有关参与者和用况的细节参见[2，4，6，10，11，12]，关于用况界面设计的信息参见[14]。

7.3 工作人员

在本章开始，我们曾介绍过在用况建模期间所得到的制品。下面来看看负责各种制品的工作人员。

如第2章中所讨论的，工作人员是一个可以赋给“真正的”人的职位。每个工作人员均是对其职责和预期行为的说明。一个工作人员与一个个人并不相同；在一个项目中，可能会指派一个人作为几个不同的工作人员。工作人员也不对应于开发组织中的具体头衔或职位——这是一个不同的问题。确切地说，工作人员代表具有业务用况（这里指软件工程的统一过程）中所需要的某种能力的人的抽象。在为项目配备人员时，工作人员表示某个人需要在项目中作为工作人员承担某项工作的知识和能力。我们可以指出参与用况建模的三类工作人员（每一类都具有自己的操作集和所需要的不同职责）：系统分析人员、用况描述人员以及用户界面设计人员。所有这些工作人员在本书中统一命名为分析人员。另外，还有其他的工作人员，如评审人员，但在本书中它们不是重点。

7.3.1 工作人员：系统分析人员

系统分析人员（*system analyst*）对模型化为用况的需求的完整集合负责，其中包括可

以由用况确定的所有功能性需求和非功能性需求。系统分析人员负责界定系统，确定参与者和用况，并确保用况模型是完整的和一致的。出于一致性考虑，系统分析人员可以使用术语表，以便在捕获需求阶段使公用术语、观点和概念得到统一。系统分析人员的职责如图7-6中所示。

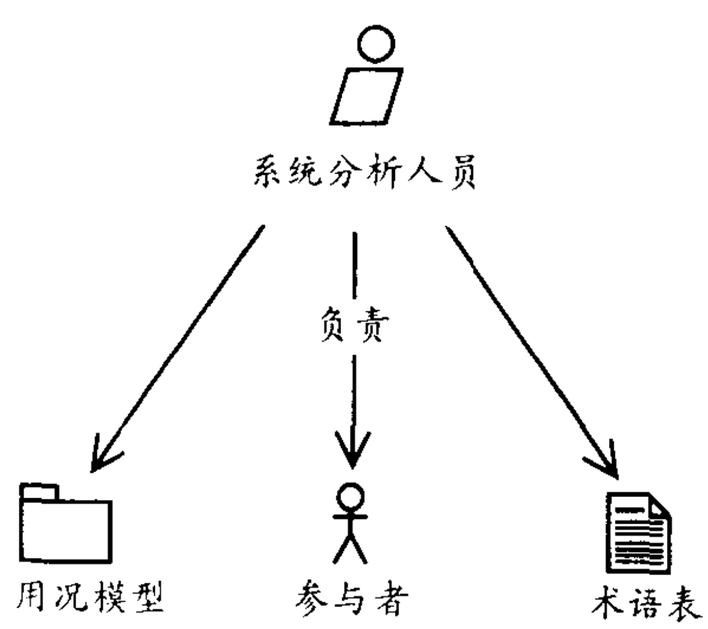


图7-6 将需求捕获为用况时系统分析人员的职责

尽管系统分析人员负责确定用况模型和其中包含的参与者，但他或她并不对每个单独的用况负责。这是另外一种职责，由用况描述人员负责（参见7.3.2节）。在需求捕获中，系统分析人员还作为建模的领导者和协调者。

每个系统有一个系统分析人员。实际上，这类工作人员需得到一个群组（在专题研讨会上或类似的事件中）的支持，其中包括很多也作为分析人员工作的其他人员。

7.3.2 工作人员：用况描述人员

捕获需求的工作一般不可能由一个人单独完成。相反，系统分析人员需要其他工作人员的协助，其中每个人负责对一个或多个用况进行详细描述。这些工作人员称为用况描述人员（*use-case specifier*）（参见图7-7）。

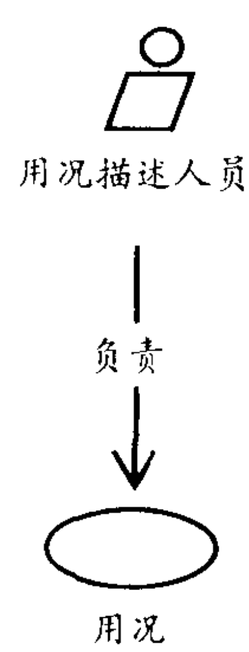


图7-7 将需求捕获为用况时用况描述人员的职责

每个用况描述人员需要与他或她负责的用况的真正用户密切接触。

7.3.3 工作人员：用户界面设计人员

用户界面设计人员（*user-interface designer*）对用户界面进行可视化定型[⊖]。它可能涉及为某些用况开发用户界面原型，通常每个参与者对应一个原型（参见图7-8）。因此，比较合适的是让每个用户界面设计人员为一个或多个参与者定型用户界面。

图7-8 将需求捕获为用况时用户界面设计人员的职责

7.3.4 工作人员：构架设计师

构架设计师（*architect*）也参与需求工作流，以便描述用况模型的构架视图（参见图7-9）。

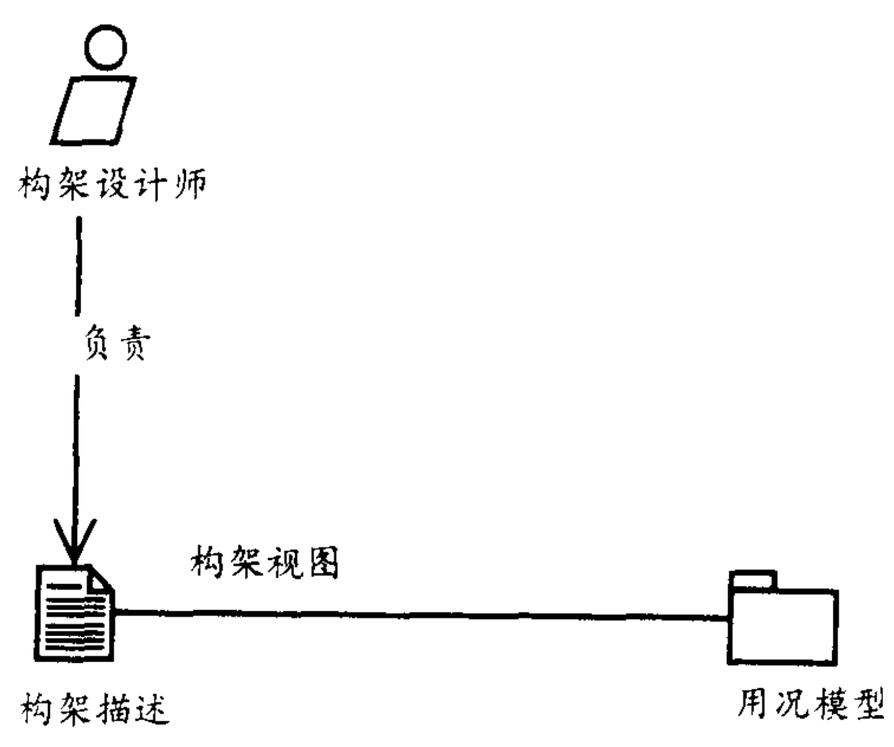


图7-9 将需求捕获为用况时构架设计师的职责

用况模型的构架视图是规划迭代开发的重要输入，如7.4.2节“活动：区分用况的优先级”所述。

7.4 工作流

前面的各章节从静态角度介绍了需求捕获工作流，现在用活动图（图7-10）来描述它的动态行为。

⊖ 用户界面设计是指用户界面的可视化定型，不是用户界面的实际实现。实际实现是由其他开发人员在设计和实现工作流期间完成的。

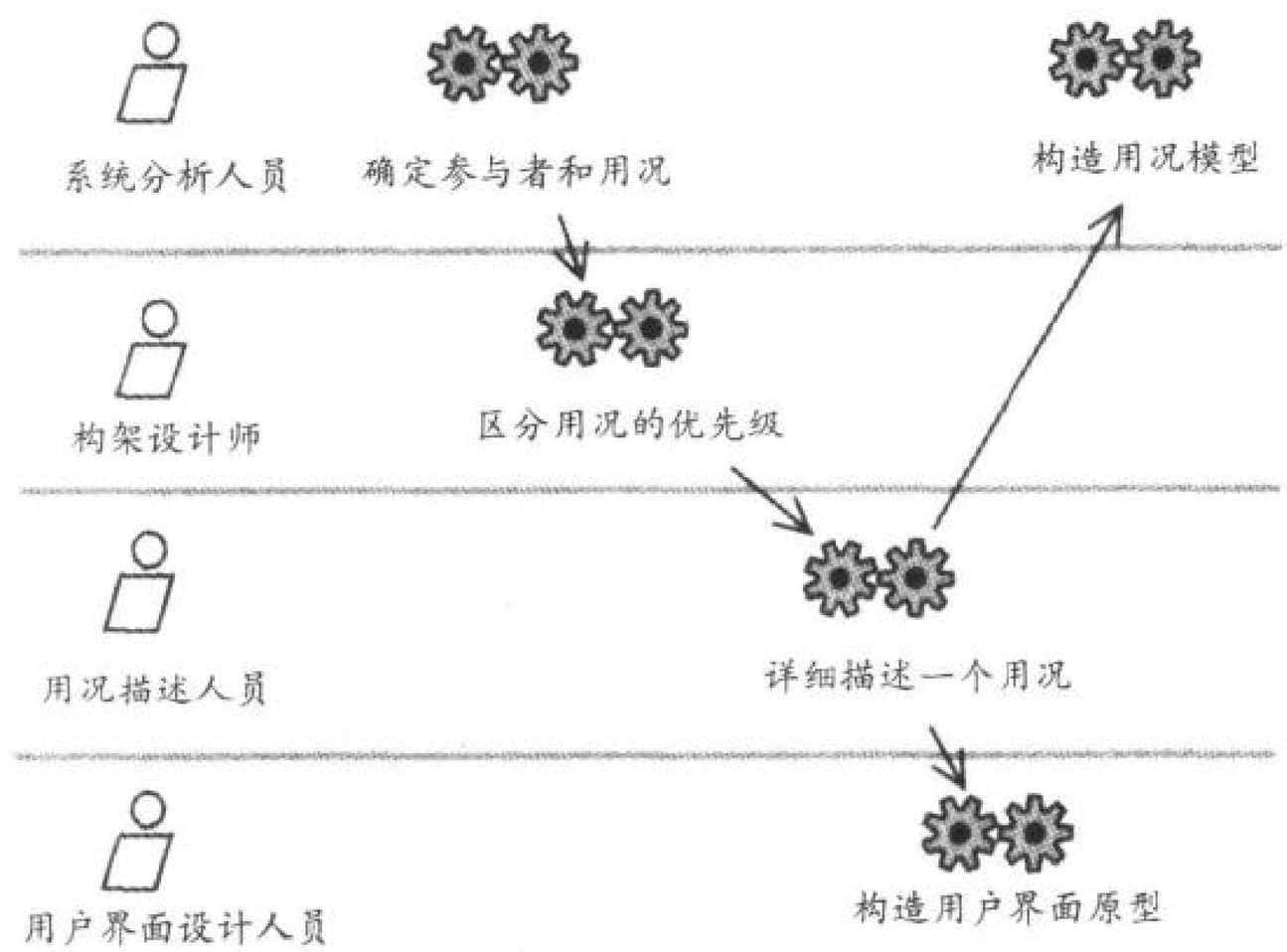


图7-10 将需求捕获为用况的工作流，其中包括参与的工作人员及其活动

图中使用泳道来表明每类工作人员所执行的活动；每个活动（用齿轮表示）与执行它的工作人员处在同一条泳道中。当工作人员执行活动时，它们会创建或改变制品。我们将工作流描述为一个活动序列，其中的活动是经过排序的，以便一个活动产生的输出作为下一个活动的输入。但是活动图展示的只是一种逻辑流。在实际的生命周期中，并不需要按顺序执行各项活动。相反，可以按任意方式执行，只要产生“等价的”最终结果。例如，我们可以从确定用况（活动“确定参与者和用况”）开始，然后设计用户界面（活动“构造用户界面原型”），这时可能会发现需要增加新的用况（所以我们又跳回到活动“确定参与者和用况”，因此破坏了严格的顺序），依此类推。

因此，一个活动可能会重复多次，每次重复可能只需要执行该活动的一部分。例如，在重复活动“确定参与者和用况”时，唯一的新结果可能是确定了一个补充的用况。因此，活动之间的路径只是表明了活动的逻辑顺序——将执行一个活动产生的结果用作执行另一个活动的输入。

首先，系统分析人员（由一组分析人员支持的个人）执行“确定参与者和用况”活动来构造包含确定参与者和用况的用况模型的最初版本。系统分析人员应确保所开发的用况模型捕获了作为该工作流输入的需求，即特征表和领域模型或业务模型。然后构架设计师将确定对构架重要的用况，作为区分将在当前迭代中开发的用况（很可能还有其他需求）优先次序的输入。然后，用况描述人员（几个人）对所有确定了优先次序的用况进行描述。几乎与此同时，用户界面设计人员（几个人）根据用况为每个参与者设计合适的用户界面。然后，系统分析人员通过定义用况间的泛化关系重新构造用况模型，以使其尽可能易于理解（我们将很快讨论到“构造用况模型”活动中的泛化关系）。

经过该工作流的第一次迭代结果得到了用况模型、用况和所有相关的用户界面原型的最初版本。后续迭代的结果构成了这些制品的新版本。记住，所有制品都是经多次迭代逐步完成和完善的。

用况建模中的各种活动在项目的不同阶段具有不同的形式（参见6.4节）。例如，系统分析人员在初始阶段执行“确定参与者和用况”活动时，会标识出很多新的参与者和用况。但在构造阶段执行这一活动时，系统分析人员会尽可能对参与者和用况集合做最小的改变，例如，创建新的用况图以便从一个特定的视角更好地描述用况模型。下面我们将介绍在细化阶段的迭代过程中出现的典型活动。

7.4.1 活动：确定参与者和用况

我们确定用况和参与者的目的是：

- 从其环境中界定系统。
- 概述谁和什么（参与者）将与系统进行交互，概述从系统预期得到哪些功能（用况）。
- 捕获和定义术语表中的公用术语，它们是对系统功能（如用况）进行详细说明的基础。

确定参与者和用况是捕获恰当需求的最为重要的活动，这就是为什么需要系统分析人员负责的原因（如图7-11所示）。但系统分析人员[⊖]不可能独自完成这项工作，他或她需要从由他或她所领导的有客户、用户以及其他分析人员参加的建模专题讨论会中获得输入。

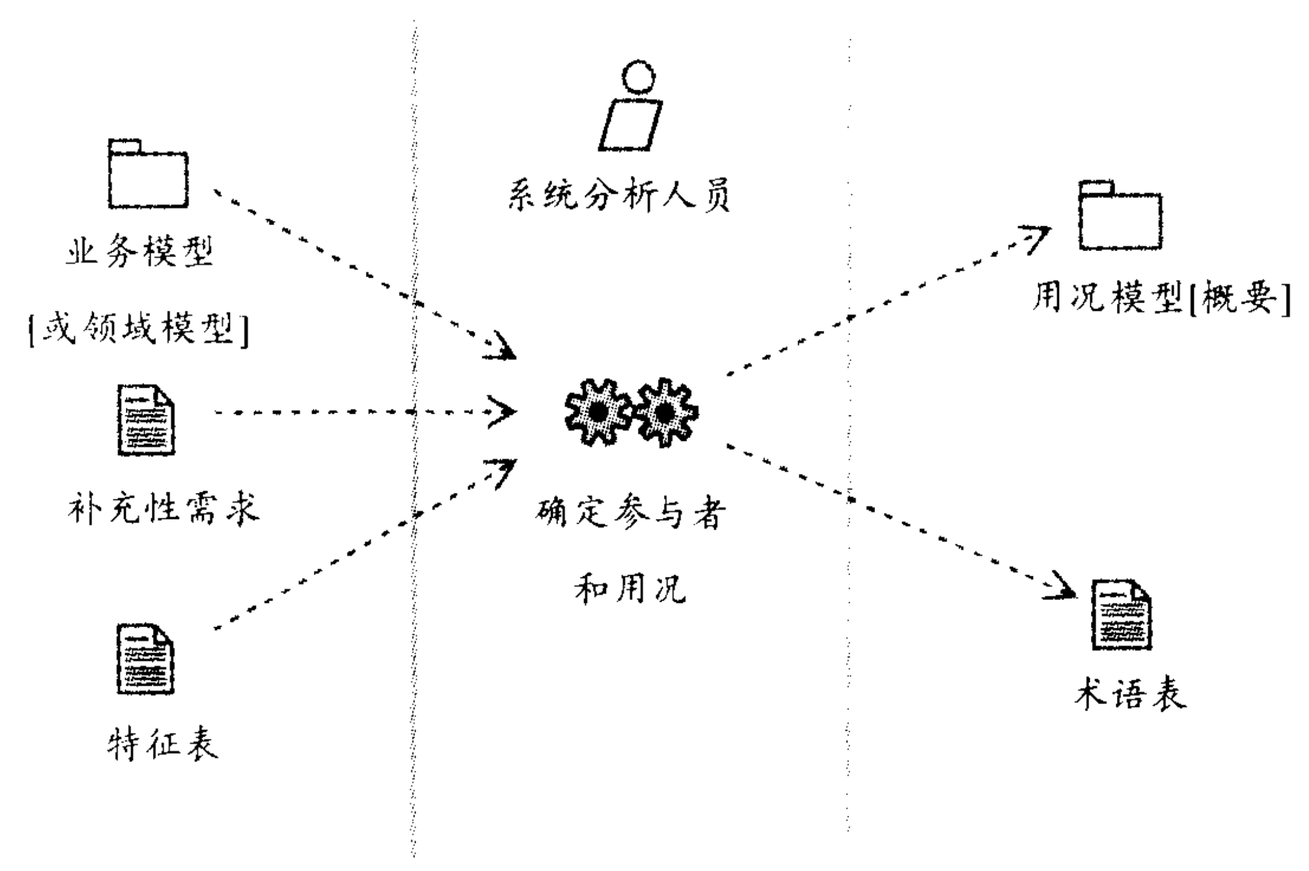


图7-11 用于确定参与者和用况的输入及其结果

有时开始时就可能已经有了一个业务模型。如果是这样，该小组或多或少地可以“自动”得到用况模型的最初草案。其他情况下，可能需要从领域模型开始，或者唯一的输入是一个简要的景象陈述或是一个包含了所需要的一般特征的详细的需求规格说明。另外，还可能给出无法分配到单个用况上的补充需求作为输入。有关各种输入制品的讨论请参见第6章。

这个活动包括四个步骤：

- 确定参与者。
- 确定用况。

⊖ 实际上，这里是指“一个担当系统分析人员的人”。区分一个真人和他或她所担当的工作人员看起来有些笨，但是它确实能使最终的描述更加清晰。我们也用同样的方法来讨论其他的工作人员。

• 简要描述每个用况。

• 从整体上描述用况模型（这一步还包括准备一张术语表）。

这些步骤不必按照特定的顺序执行，而经常会并发执行。例如，一旦确定了新的参与者或用况，就可以对用况图进行更新。

这次活动的结果是包括新的或改变了的参与者和用况的新版用况模型。所得到的用况模型制品应该从表面上进行简明扼要地描述或图示，以便可以对其中的每个用况进行详细描述，这些在接下来的活动“细化用况“中完成。图7-12就是这样一个用况图的说明（经过多次迭代来完善和重构）。后面将对其进行更加详细的介绍。

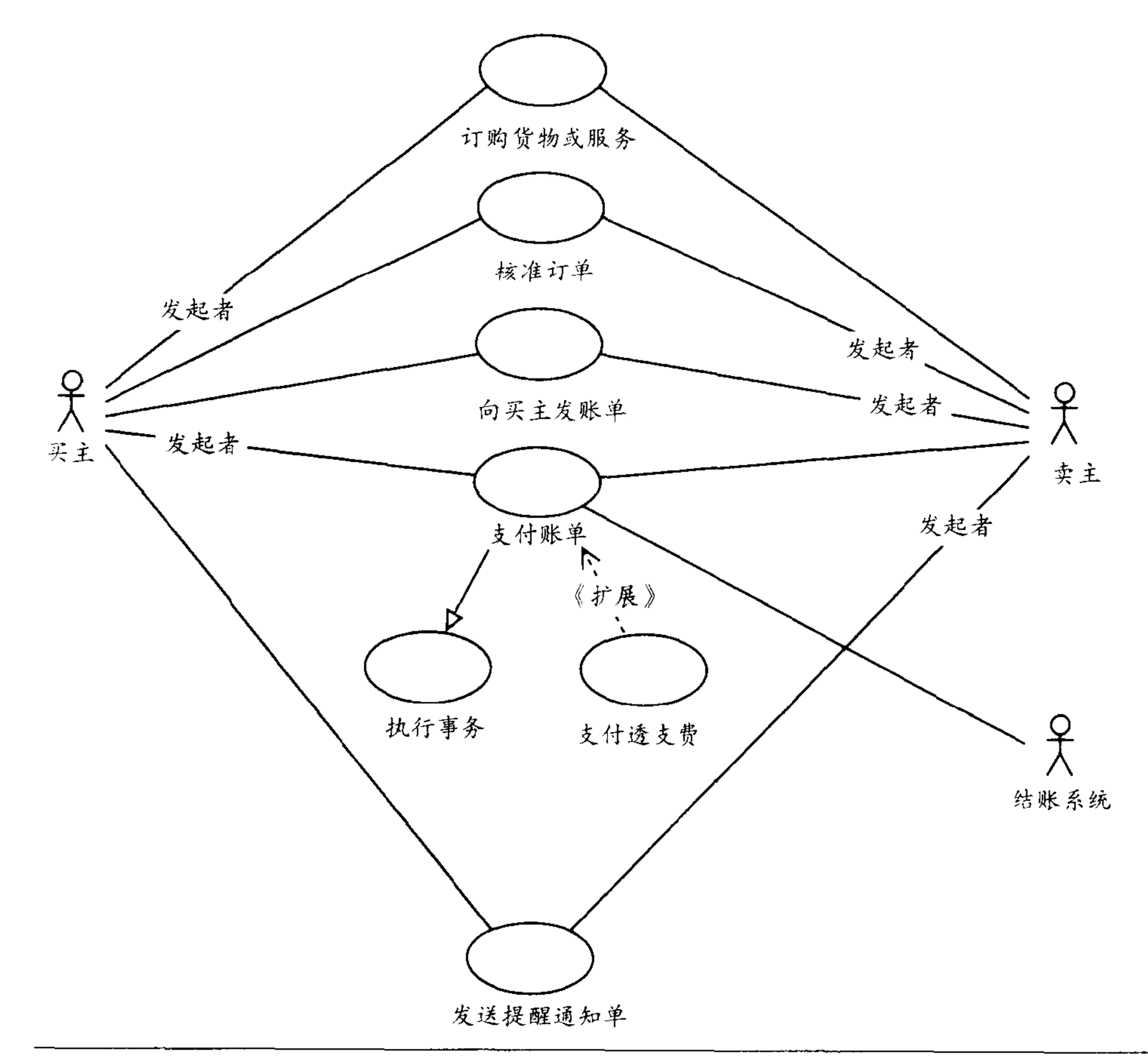


图7-12 支持业务用况“销售：从订单到交货”的“结账与支付系统”中的用况。关联上的角色发起者表明哪个参与者启动这个用况

7.4.1.1 确定参与者

确定参与者的任务依赖于起始点。

当开始存在一个业务模型时，确定参与者很简单。系统分析人员可以为业务中的每个工作人员建议一个参与者，也可以为每个使用该信息系统的业务参与者（即每个业务的客户）建议一个参与者（参见第6章的6.6.3节）。

否则，不管有没有领域模型，系统分析人员都要与客户一起来确定用户，并设法将他们组织成不同的类别来表示参与者。

在这两种情况下，需要将表示外部系统的参与者与用于系统维护和操作的参与者确定下来。

在挑选候选参与者时有两个标准非常有用：首先，应该能至少确定一个用户来扮演候选参与者。这有助于我们只获得相关的参与者而避免只是我们想象中的“幻象”的参与者。其次，与系统相关的不同参与者实例所充当的角色之间的重叠应该最少。我们不希望两个或多个参与者实质上充当相同的角色。如果出现了这种情况，应设法将这些角色合并到对应于一个参与者的唯一的角色集合中；或者设法确定另一个更一般化的参与者来充当重叠参与者的公共角色。于是，这个新的参与者可以使用泛化关系进行特化。例如，“买主”和“卖主”是“银行储户”参与者的特化。通常最初几次确定的参与者数目太多而且有较多的重叠，一般需要经过几轮探讨才能最终确定恰当的参与者集合及泛化关系。

系统分析人员为这些参与者命名，并简要描述每个参与者的角色以及参与者使用系统做什么。为参与者确定有关的名字对于表达所期望的语义十分重要。对每个参与者的简要说明应该概括其需求和责任。

举例：“买主”、“卖主”和“记账系统”参与者

“买主”

按照业务用况“销售：从订单到交货”中所描述的，“买主”表示负责购买货物或请求得到服务的人。这个人可能是一个个人（即不隶属于任何公司）或一个业务组织中的某个人。货物和服务的“买主”需要使用“结账与支付系统”来发送订单和支付账单。

“卖主”

“卖主”表示出售、交付货物或提供服务的人。卖主使用该系统来查看新的订单，并发送订单确认、账单和支付提醒通知单。

“记账系统”

“结账与支付系统”向“记账系统”发送事务验证信息。

这一步的结果是一个更新了参与者集合的用况模型制品的新版本，其中每个参与者都有一个简要说明。此时，这些经简要说明的参与者便可以用作确定用况的起始点。

7.4.1.2　确定用况

当起始点是业务模型时，可以按照第6章的6.6.3节“根据业务模型确定用况”中讨论的方法来确定参与者和用况。需要为参与业务用况实现和使用该信息系统的每个工作人员所充当的每个角色设计一个用况。另外，系统分析人员通过专题讨论会与客户及其用户共同商讨来确定用况。系统分析人员逐个检查参与者，并为每个参与者设计候选用况。例如，可以通过访谈和连环画来了解需要用况做些什么，参见[16]。参与者通常需要用况支持其工作，来创建、改变、跟踪、删除或研究用于业务用况中的业务对象（如“订单”和“账户”）。参与者可能要将某些外部事件或其他类似的方法告知系统，或反之，即参与者可能需要系统告知自己某些已经发生的事件（例如账单过期）。可能还需要有附加的参与者来执行系统启动、终止或维护等工作。

一些候选用况可能不适合作为单独的用况，而最好作为其他用况的一部分。记住，我们应

设法创建易于按一个单元来进行修改、评审、测试和管理的用况。

为每个用况选择一个名字，由此能想到为参与者增值的具体动作序列。用况的名字通常以一个动词开始，并应该反映出参与者和系统之间的交互所要实现的目标。在我们例子中有如“支付账单”和“订购货物或服务”等用况。

有时很难确定用况的范围。用户与系统的交互序列可以在一个用况中确定，也可以在参与者逐个引用的几个用况中确定。在决定一个候选用况是否应该作为一个单独的用况时，必须考虑它本身是否完整，或是否总是作为另一个用况的延续。我们曾介绍过用况可以为它们的参与者增值（参见7.2.3节 “用况”）。更确切地说，用况会产生一个对具体参与者有价值的可见结果。这种关于“好”用况的实用指南有助于确定用况的适用范围。

注意，这个指南中的两个关键词组，*有价值的结果*和*具体参与者*，表示出确定用况的两个有用的准则：

- *有价值的结果*（*result of value*）：每个可以成功执行的用况都应该对其参与者提供某些价值，使参与者实现预定目标[3]。有时候，参与者愿意为获得价值回报而付出代价。注意，一个用况实例（如打电话）可能涉及多个参与者。此时，“有价值的可见结果”准则应该针对最初的参与者。这个“有价值的结果”准则有助于避免确定太小的用况。

举例：“支付账单”用况的范围

“结账与支付系统”提供了一个称为“支付账单”的用况，买主使用该用况来确定他或她已经订购或收到的货物账单的支付日期。然后，“支付账单”用况会在预定日期完成支付。

“支付账单”用况包括确定支付日期和完成支付。如果将该用况分为两部分，一部分是确定支付日期，另一部分是完成支付。其中“确定支付日期”本身不会增值，只有支付了账单（此时不必担心会收到提醒通知单），才会获得增加的价值。

- *具体参与者*（*particular actor*）：通过确定向真正的用户提供价值的用况，可以确保用况不会变得太大。

至于最初确定的参与者和用况，在用况模型稳定之前通常需要进行多次重构和评估。

第4章曾介绍过，一个系统的用况和构架是通过迭代逐步开发的。一旦得到了一个构架，后面捕获的新用况一定要适合这个现存的构架。不适合所选构架的用况应该加以改变以便更好地融入该构架，或者改进构架以便更易于补充新的用况。例如，最初确定用况时，头脑中可能已经有了一个与具体用户的交互。一旦确定了GUI（图形用户界面）框架，可能需要修改相应的用况，尽管这种适应性修改通常很小。

有时可能需要进行较大的改变。系统分析人员可能会提出一种方法，通过指定一个模拟器作为重要的参与者来监控系统的加载，它要向系统请求用况。用这种方法，可以测量响应时间和其他性能需求；也可以测量用况在系统内部队列中的暂停（等待）时间。这两种方法可能会给出相似的值，但是根据现存的构架来实现它们的代价可能相差很大。于是，系统分析人员可能需要与客户重新协商需求（用况等），以便构造一个更好的、开发人员易于实现和维护的系统。客户也可以通过重新商议需求获益，而且可能会以高质量、低成本尽早获得相应

的功能。

7.4.1.3 简要说明每个用况

分析人员在确定用况时，有时候会潦草地写下一些词来说明每个用况，有时只是记下用况的名字。之后，他们首先用几句话来简要描述每个用况，概括用况的动作；然后逐步说明当用况与其参与者交互时系统需要完成什么任务。

举例："支付账单"用况的初始描述

简要说明

"买主"使用"支付账单"用况来确定账单支付时间。而后，"支付账单"用况在预定日期实现支付。

初始的逐步描述

在对用况进行初始化之前，"买主"已经收到了账单（由另一个称为"买主账单"的用况发送），而且已经收到了订购的货物或服务：

1) 买主检查要支付的账单并核对是否与原始订单一致。

2) 买主确定通过银行支付账单的时间。

3) 在预定的支付日期，系统检查买主的账户里是否有足够的存款。如果有足够的存款，就完成该事务。

至此，我们已经简要介绍了参与者和用况。但只是孤立地描述和理解每个用况是不够的，还需要从整体上来把握。我们需要说明用况和参与者如何彼此相关以及如何一起组成用况模型。

7.4.1.4 从整体上描述用况模型

我们准备用图和说明从整体上解释用况模型，尤其是想说明用况如何彼此相关以及用况如何与参与者相关。

对图中应该包括哪些内容没有严格的规定。我们选择那些能更明确说明系统的图。例如，可以画图来显示参与一个业务用况的若干用况（见图7-12），或表明由一个参与者所执行的若干用况。

在同时描述几个用况时为确保一致性，建立一张术语表是可行的。这些术语可能来自于（并可跟踪到）领域模型或业务模型中的类（第6章中曾介绍过）。

用况模型无需是这里所描述的平面模型，还可以将它组织成称为*用况包*的用况簇[12]。

这一步的输出也是用况模型的综合说明，它从整体上解释用况模型，描述参与者与用况如何进行交互，并描述用况之间如何彼此相关。在用况模型的UML表示中，综合说明是模型本身的标记值（参见附录A的A.1.2节）。

举例：综合说明

"结账与支付系统"（见图7-12）中用况模型的综合说明大致如下。该说明中包括带有序号的注释，并在例子的最后对注释内容进行了解释。

买主使用"定购货物或服务"用况查看订单条目和价格，汇总并提交订单。

可能当时或稍后，货物或服务会连同账单一起提交给买主。

买主启动“支付账单”用况，认可收到的账单并确定支付时间。在预定的支付日期，“支付账单”用况自动将存款从买主的账户转到卖主的账户上。{注释1}

另外，如果出现透支，“支付账单”用况将由“支付透支费用”用况进行扩展。{注释2}

现在，我们看看卖主如何使用该系统。卖主可能会使用“确认订单”用况查看、建议更改和确认收到的订单。确认后的订单将与订购的货物或服务一起发送出去（在我们的用况模型中没有对此进行描述；这是在“结账与交付系统”之外实现的）。

之后，当货物或服务送达时，卖主通过“买主账单”用况给买主开账单。在开账单时，卖主可能需要使用一个合适的折扣率，还可能将几个账单合并为一个账单。

如果买主到期没有支付，卖主会得到通知并使用“发送提醒通知单”用况。系统也可以自动发送提醒通知单，但我们选择另一种解决方案，即卖主先检查一遍提醒通知单之后再发送出去，以避免给任何客户带来麻烦。{注释3}

注释

1) 我们曾介绍过，用况模型并不仅仅是一个用况清单。它还描述了这些用况之间的泛化关系。例如，“支付账单”用况中支付事务的动作序列可以由多个用况共享（尽管图7-12中只显示了一个泛化关系）。这种共享可以用一个称为“执行事务”的独立用况来表示，该用况通过泛化关系被多个用况（如“支付账单”用况）重用。泛化意味着将“执行事务”用况中所描述的动作序列插入到“支付账单”的动作序列中。当系统执行“支付账单”用况的实例时，该实例也会按照在“执行事务”用况中描述的行为执行。

2) 在系统执行“支付账单”用况的实例时，其动作序列可能会扩展为包括在扩展用况“支付透支费用”中描述的动作序列。刚才简单提到泛化和扩展关系是为了说明可以将用况模型构造得更容易确定和理解系统功能性需求的完整集合；详见[7]。

3) “发送提醒通知单”说明了在业务用况中简化更正途径的一个用况。这样更正途径可以帮助过程“重新步入轨道”，也许会防止在与客户交互时出现的小问题发展成为真正的难题。因此，参与者还需要用况（或基本用况的备选用况）来帮助他们纠正偏离的过程路径。这类用况通常占整个系统职责的相当一部分，用以处理很多可能出现的偏离问题。

讨论

建造用况模型可以有多种方法，这个例子只是阐明了其中的一种方法。下面讨论一下我们在确定用况模型时所做的一些权衡。如果买主可以在因特网上浏览可获得的货物或服务目录，接着进行在线订购并立刻获得确认又将如何。还需要单独的“确认订单”用况吗？答案是否定的，因为对订单的直接确认应该包含在“订购货物或服务”用况中。

但在这个例子中，我们假设在对订单进行分析之后，卖主或者确认订单或者会建议一种其他备选方案。例如，卖主可能会建议一组用途相同但更便宜或能更快送达的备选货物。这时，发送订单的实际顺序实际上可能是一系列的步骤，包括卖主提出备选订单并随后予以确认，如下所示：

1) 由买主发出初始订单。

2) 卖主将备选订单建议发送给买主。

3) 买主发出最终订单。

4) 卖主向买主发送订单确认。

这些步骤包含在“订购货物或服务”和“确认订单”两个用况中。为什么不用四个不同的用况或只用一个用况呢？首先看看为什么没有把这些步骤描述为一个大的用况。我们不希望迫使买主和卖主实时进行交互，但希望他们能够向对方提交请求而无需互相等待或同步。可以假设卖主希望收集到多个来自于不同买主的新订单，然后一并对它们进行分析和确认。如果这四步作为一个用况，就不可能实现上述目标；因为每个用况都被视为是原子的，因此总是要完成一个用况之后，再开始执行下一个用况。所以，在等待将备选订单发送回去的时候（例子中的步骤2），卖主不可能有多个待处理的初始订单（步骤1之后）。

当然，这四个步骤可以对应四个不同的用况，但初始订单和最终订单（步骤1和3）很相似，因而可以表示为“订购货物或服务”用况的备选用况。毕竟，我们不希望这组用况超出我们的理解范围。用况太多会使用况模型在作为分析和设计的输入时难以理解和使用。

同样，建议备选订单（步骤2）包括对订单中的某些部分进行确认并对其他部分提出备选建议，而确认订单（步骤4）包括确认订单的所有部分。这两步可以表示为一个“确认订单”用况，允许卖主确认订单中的一部分或对其他部分提出备选建议。所以，用一个称为“订购货物或服务”的用况提供对应于步骤1和步骤3的服务，而另一个称为“确认订单”的用况对应于步骤2和步骤4。

在建立综合用况模型时，应该邀请开发组之外的人员（如用户或客户）进行非正式的评审，来确定是否满足以下各条，并据此确定是否批准该用况模型：

- 是否已将所有必需的功能性需求捕获为用况。
- 每个用况的动作序列是否是正确的、完整的和易于理解的。
- 是否已经确定了一些价值很小或根本没有价值的用况。果真如此，则应该重新考虑这些用况。

7.4.2 活动：区分用况的优先级

这个活动的目的是为区分用况优先级提供输入，以决定其中哪些用况需要在早期的迭代中进行开发（即分析、设计、实现等），以及哪些用况可以在随后的迭代中进行开发（图7-13）。

可以在用况模型的构架视图中获得这些结果。然后与项目经理一起对该视图进行研究，并将该视图用作规划一次迭代中所要开发的内容时的输入。注意，这个规划还需要考虑其他非技术性因素，如待开发系统的业务或经济方面的因素（参见第12章的12.4.2节）。

用况模型的构架视图应该描述对构架重要的用况。详细的资料请参见7.2.4节“制品：构架描述（用况模型视图）”。

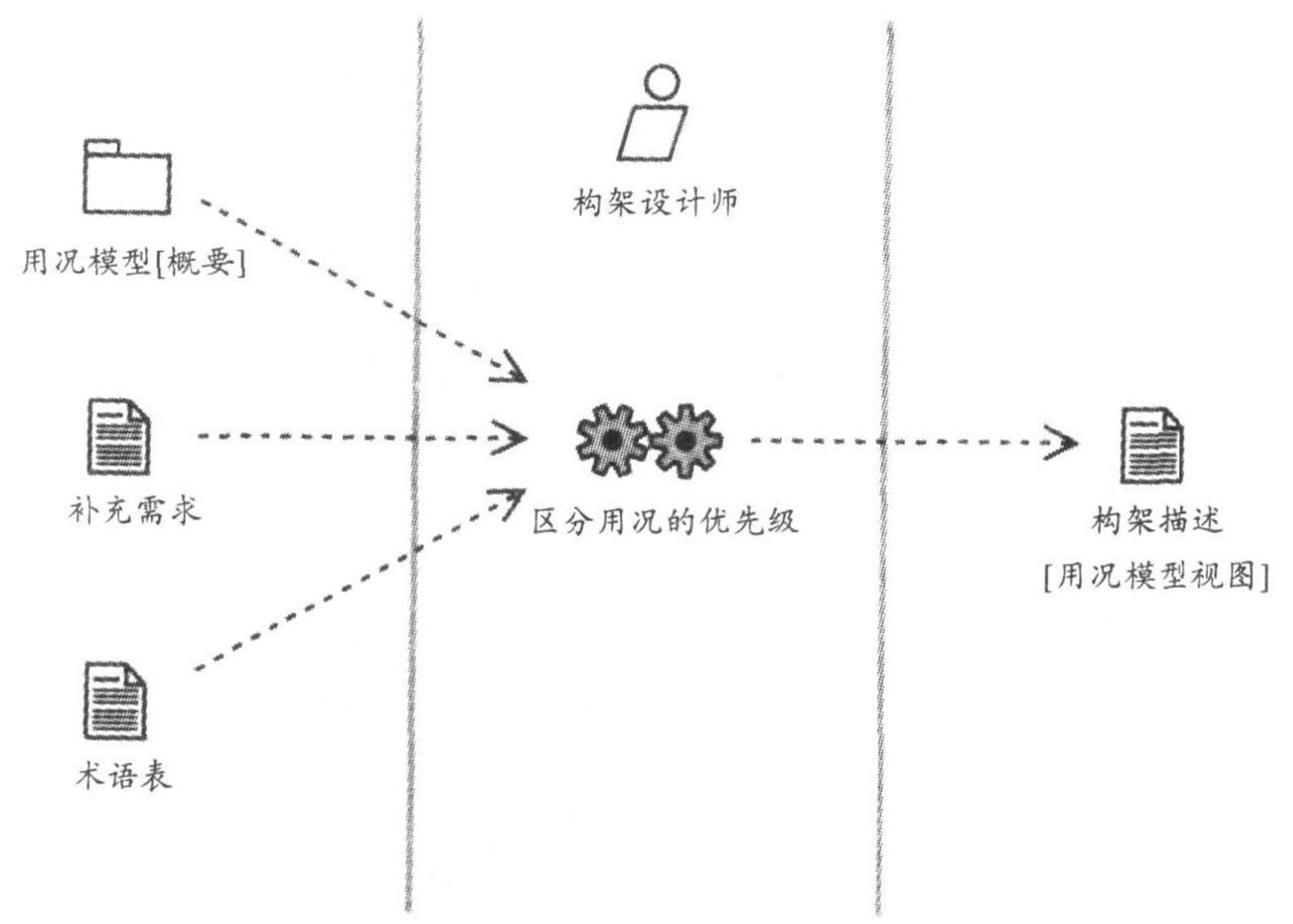


图7-13 区分用况优先级的输入及其结果

7.4.3 活动：详细描述一个用况

细化每个用况的主要目的是为了详细描述其事件流，包括用况如何开始、结束以及如何与参与者进行交互（如图7-14所示）。

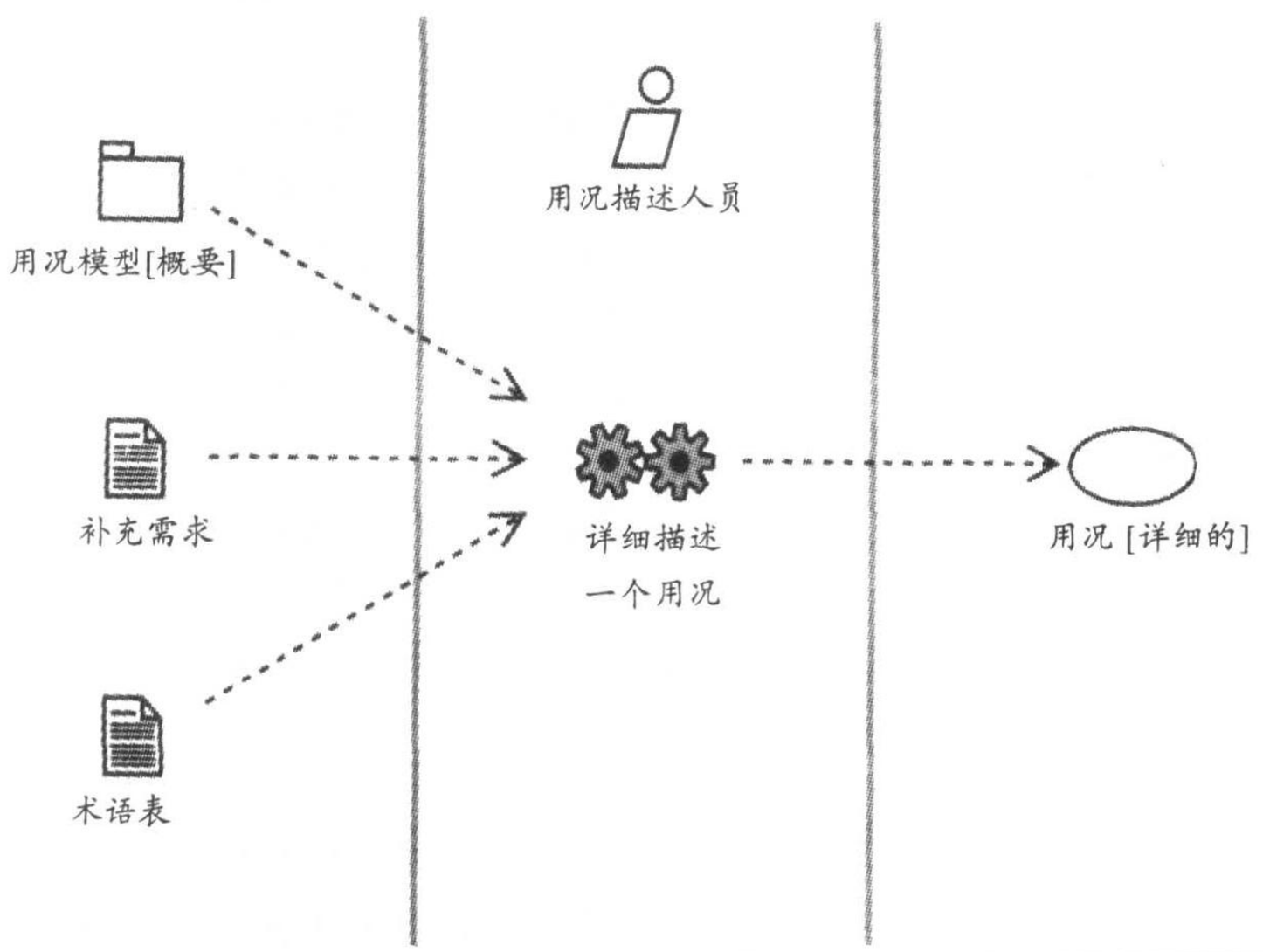


图7-14 详细描述每个用况的输入及其结果

此时，以用况模型和相关联的用况图作为起始点，每个用况描述人员就可以详细描述每个用况。他们将每个用况的逐步描述细化为精确的动作序列规格说明。在本节中将看到：

- 如何组织描述以确定用况的所有备选路径。
- 用况描述中应包括哪些内容。
- 必要时如何将用况描述形式化。

每个用况描述人员需要与用况的真正用户密切合作。用况描述人员需要与用户面谈，可能要记下用户对用况的理解，与他们讨论有关用况的各种建议，并请他们评审用况说明。

该活动的结果是以文本和图表示的某个具体用况的详细说明。

7.4.3.1　构造用况说明

我们曾经提到过，用况定义了用况实例可能进入的状态以及在这些状态间的转移（参见图7-15）。每个这样的转移都是当一个事件（如消息）触发时由一个用况实例所执行的动作序列。

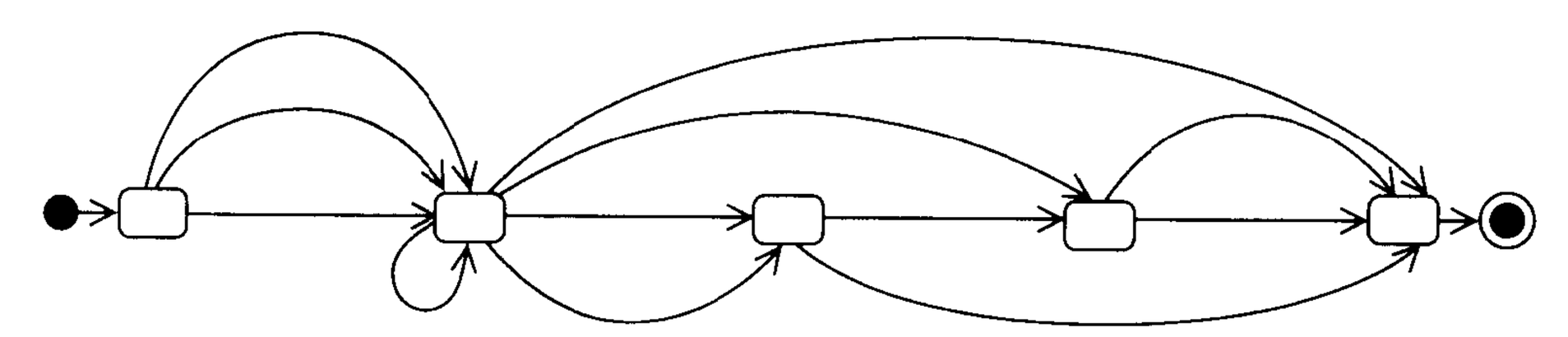


图7-15　可以认为用况有一个初态（最左面的圆角矩形）、中间状态（后面的圆角矩形）、终态（最右面的圆角矩形）和状态间的转移（参见附录A的状态图符号）。带箭头的直线代表基本路径，带箭头的曲线代表其他路径

图7-5所示的状态转移图可以变得非常复杂。尽管如此，我们必须尽量简单而精确地描述可能的状态转移（动作序列）。一种可行的技术是选择一条从初态到终态的完整的基本路径（图7-15中带箭头的直线），并在说明中用一小节对这条路径加以描述。而后可以将其他路径（带箭头的曲线）描述为基本路径的备选或偏离路径，每条路径在单独的一小节中描述。但有时候备选或偏离路径很小，可以将它们直接插入基本路径的描述中。根据常识，确定是否将备选路径描述直接插入基本路径描述，或是将它创建为单独的一小节。记住，目标是使说明精确、易读。无论选择什么技术，我们仍要描述所有的备选路径，不然就不能算确定了这些用况。

出现基本路径的备选、偏离或异常路径有很多原因：

- 参与者可能选择不同的路径来完成用况。例如，在执行“支付账单”用况时，参与者可能决定支付或拒付账单。
- 如果用况涉及多个参与者，其中一个参与者的动作可能会影响其他参与者的动作路径。
- 系统可能检测到来自于参与者的错误输入。
- 有些系统资源可能出现了故障，于是导致系统无法正常工作。

所选的基本路径应该是一条“正常”路径，即用户认为最有可能遵循的和能够给参与者带来最显著价值的路径。一般地，这样一条基本路径几乎不包括很少需要由系统加以处理的异常和特殊路径。

举例："支付账单"用况的路径

请注意，该文本是如何从本章前面只有一个用况说明纲要时（参见7.4.1.3节"简要说明每个用况"）的草稿改变而来的。这种改变反映出在对用况建模时如何细化用况，而完整的用况说明实际上可能更大、且包含更多的路径。

前提条件：买主已收到订购的货物或得到预定的服务，且至少有一份账单来自于该系统。此时，买主计划确定支付账单的时间。

事件流

基本路径

1) 买主通过浏览从系统接收到的账单来激活用况。系统检查每份账单的内容是否与早先收到的订单确认（作为"确认订单"用况的一部分）一致，并以某种方式将结果通知买主。订单确认表明可以提供哪些条目，何时、何地、以何种价格提供。

2) 买主确定通过银行支付账单的时间，而系统产生一个支付申请以便将存款转到卖主的账户上。注意，买主不能对同一账单确定两次支付时间。

3) 之后，如果买主的账单上有足够的存款，支付事务会在预先确定的时间完成。在事务执行期间，按照抽象用况"执行事务"（由"支付账单"用况使用）中所描述的，存款从买主的账户转到卖主的账户上。买主和卖主将会得到事务执行结果的通知。银行对这笔交易收取一定的费用，由系统直接从买主的账户上扣除。

4) 该用况实例结束。

备选路径

在步骤2中，买主可能改为请求系统发送拒付账单给卖主。

在步骤3中，如果账户上没有足够的存款，该用况将取消支付事务并通知买主。

后置条件：当账单支付完毕或取消账单支付以及没有存款可转时，该用况实例结束。

因为用况需要得到开发人员、客户及用户的理解，所以，应该像这个例子中所示的那样，用简明的语言来描述用况。

7.4.3.1.1 用况说明中包括的内容

如前面的例子所示，用况说明必须包括以下内容：

- 用况说明应该将初态（见图7-15）定义为前提条件。
- 用况如何以及何时开始（即第一个动作的执行；步骤1）。
- 动作必须按所要求的顺序（如果需要）执行。这里，顺序是通过顺序编号定义的（步骤1~4）。
- 用况如何以及何时结束（步骤4）。
- 用况说明应该将可能的终态定义为后置条件（见图7-15）。
- 不允许的执行路径。步骤2中的注释提醒我们一条不可能发生的路径——两次付账是不允许的。它是用户不能选取的路径。
- 直接插入基本路径描述的备选路径描述。步骤3中所有的动作是只有当账户上有足够的存款时才执行的动作。

- 从基本路径描述中抽取出来的备选路径描述（见上例中关于“备选路径”的说明）。
- 系统与参与者之间的交互以及它们彼此交换的信息（步骤2和3）。典型的例子是步骤2中买主确定账单支付时间以及步骤3中将事务执行结果通知买主和卖主。换句话说，就是要描述用况的动作序列，描述这些动作如何由参与者激发，以及如何要求参与者通知其执行结果。
- 系统中对象、值以及资源的用法（步骤3）。换句话说，就是要在用况使用中描述动作序列，给用况属性赋值。典型的例子是步骤3中将存款从买主的账户转到卖主的账户，另一个例子是步骤1中确认账单和订单的用法。
- 注意，必须明确地描述系统要做的工作（执行的动作）和参与者要做的工作。要能够区分系统和参与者的不同职责，否则用况说明就不够精确不足以作为系统规格说明使用。例如，在步骤1中写的“系统检查每份账单的内容是否与早先收到的订单确认一致”，以及步骤3中的“这笔费用由系统从买主的账户上扣除”。

以后在分析和设计阶段，用况的属性可以用作一种启发，以确定类和属性，例如，从用况的账单属性可以导出称为“支付账单”的设计类。在分析和设计期间，还应该考虑那些用于多个用况的对象，但在用况模型中无需考虑这一点。相反（正如在7.2.3节“用况”中讨论的那样），应该通过禁止用况实例间的交互和访问彼此的属性的实例来保持用况模型简单。

至此，我们主要讨论了功能性需求以及如何将它们表示为用况，但还需要详细说明非功能性需求。大多数非功能性需求是与具体用况有关的，如规定速度、可用性、准确度、响应时间、恢复时间或内存使用等需求，这些是系统执行一个给定用况时所要求的。将这些需求作为特殊需求（UML中的标记值）附加到所讨论的用况上，并可以在用况说明的一个独立小节中进行描述。

举例：特殊（性能）需求

在卖主发出账单支付请求时，在90%的情况下，系统应该在1秒钟内对请求的验证做出响应。验证的时间绝对不能超过10秒钟，除非网络中断（这时应该通知用户）。

如果该系统需要与其他系统进行交互（不是由人充当参与者），就有必要详细说明这种交互（如参考标准通信协议）。这项工作必须在细化阶段的早期迭代中完成，因为系统间通信的实现常常会对构架产生影响。

当认为用况说明已经是易于理解的、正确的（即捕获了正确的需求）、完整的（例如，它们描述了所有可能的路径）和一致的时候，用况说明就已经完成了。在需求捕获末期举行的评审会上，分析人员、用户和客户一起对用况说明进行评价。只有客户和用户才能验证这些用况是否是正确的。

7.4.3.2 对用况说明进行形式化描述

图7-15表明了用况实例如何通过转移从一种状态转变到另一种状态。通常，当用况只有几种状态时，无需明确地描述这些状态，而可以选用“支付账单”的例子中所使用的风格。然而在描述用况时，为了确保包含所有可能的情况，在头脑中考虑一下该用况的状态机是个很好的主意。但在复杂的实时系统中，用况可能非常复杂，以致于有必要使用更为结构化的描述技术。

例如，参与者和用况间的交互作用可能要经历很多个状态和可选择的转移，以致于几乎无法保持文本用况说明的一致。于是，使用可视化建模技术来描述用况就可能非常有用。这种技术有助于系统分析人员更好地理解用况：

- UML的状态图可以用于描述用况的状态以及这些状态间的转移；如图7-16所示。
- 可以用活动图按照动作的顺序更为详细地描述状态间的转移。可以将活动图看作是SDL状态转移图（一种用于电信领域的成熟技术）的一般形式[15]。
- 交互图可以用于描述用况实例如何与参与者实例进行交互。因此，交互图显示了用况和所参与的参与者（或参与者们）。

有关对状态图、交互图和活动图的解释参见[2，11，12，17]。

举例：使用状态图来描述用况

图7-16是“支付账单”用况的状态图。图顶端的黑点代表用况的开始，这是用况实例在实例化时状态机的起点。从黑点引出的箭头表明状态机在实例化时直接移动到的位置，此时是第一种状态“浏览”。状态由圆角矩形表示。状态转移由从一种状态到另一种状态间的箭头表示。

首先，用户浏览账单（参见前面“支付账单”例子中的步骤1），而后确定支付账单时间（参见步骤2）或拒付账单。对确定了支付时间的账单在预定日期进行支付时，该用况从状态“已确定账单支付时间”转移到“账单已支付”（参见步骤3）。当用况到达“账单已支付”或“账单被取消”状态后便终止（中心带黑点的圆圈）。

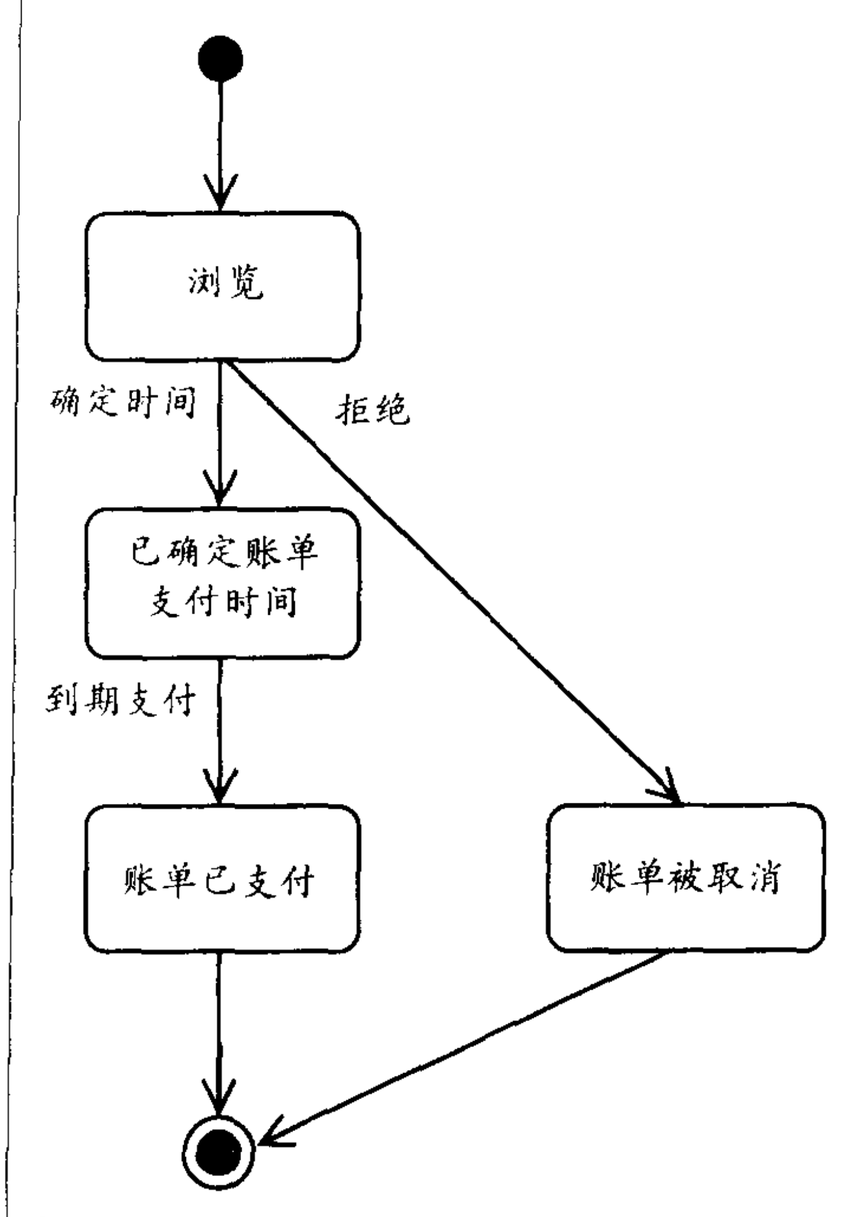


图7-16 “支付账单”用况的状态图显示了它的实例如何通过一系列状态转移（箭头）在几个状态（圆角矩形）上移动

注意，在用况语境中使用这些图时，有时可能会得到很大、很复杂的图，以至于难以读懂。例如，一个单个的用况可能包含很多状态而难以用有意义的方式来命名。对于不是软件开发人员的项目相关人员来说，要读这些图尤为困难。另外，制作一些详细的图并使它们与系统的其他模型保持一致需要付出很高的代价。

因此，我们的忠告是谨慎使用这类图，而通常使用用况的纯文本描述（即事件流描述）也就足够了。另外，在多数情况下，文本描述和图可以相互补充。

7.4.4 活动：构造用户界面原型

该活动的目的是构造用户界面的原型（如图7-17所示）。

到目前为止，系统分析人员已经建立起用况模型，确定了用户是谁以及他们要用系统做什么。这些都已经反映在用况图、用况模型的综合描述以及每个用况的详细描述中。

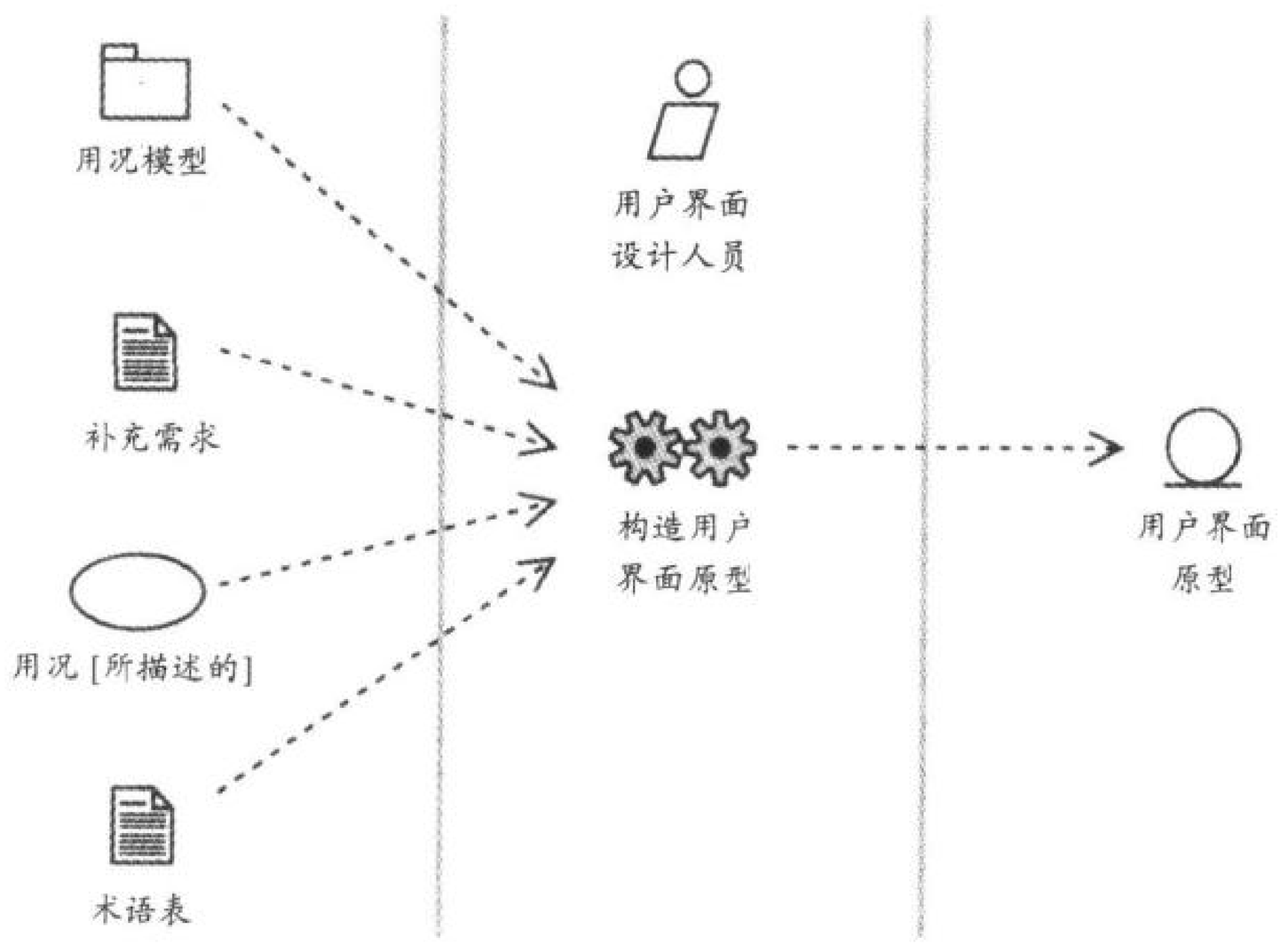


图7-17　构造用户界面原型的输入及其结果

现在我们需要设计用户界面，以便使用户能有效地执行用况。我们分几步来完成这项任务。首先，从用况入手，并设法了解为使每个参与者能启动用况需要用户界面提供些什么，即进行逻辑用户界面的设计。然后，再进行实际的用户界面设计并开发出一个原型，以说明用户如何使用系统来执行用况[1]。在决定如何实现用户界面之前首先确定需要用户界面做些什么，可以迫使我们在设法实现它们之前首先理解需求[1]。

该活动的最终结果是一组用户界面简图和用户界面原型，从而为最关键的参与者确定用户界面的外观和感觉。

7.4.4.1 进行逻辑用户界面设计

当参与者与系统交互时，他们使用和处理表示（用况）属性的用户界面元素。这些属性通常是术语表中的术语（如账户余额、支付日期和账户所有者）。参与者所感受到的这些用户界面元素是图标、列表、文件夹或二维地图中的对象，并通过选择、拖动或说话（语音控制）来对它们进行处理。用户界面设计人员通过遍历参与者能访问的所有用况并为每个用况确定适当的用户界面元素，每次为一个参与者确定和说明这些元素。一个用户界面元素可能会参与多个用况[⊖]，在每个用况中充当一个角色。因此，必须将该用户界面元素设计为能充当所有这些角色。每个参与者都应该对以下问题做出回答：

- 需要哪些用户界面元素来启动这些用况？
- 用户界面元素彼此之间应该如何相关？
- 如何将它们用在不同的用况中？
- 它们看起来应该像什么？
- 应该如何处理它们？

为了确定参与者可以访问的每个用况中需要哪些用户界面元素，我们可以询问以下问题：

- 哪些领域类、业务实体或工作单元适合作为该用况的用户界面元素？
- 参与者用哪些用户界面元素完成工作？
- 参与者可以激发哪些动作，能够做那些决定？
- 参与者在激发用况的动作前需要哪些指南和信息？
- 参与者需要向系统提供什么信息？
- 系统需要向参与者提供什么信息？
- 所有输入/输出参数的平均值是多少？例如，参与者在一个会话期内一般处理多少账单，平均账户余额是多少？我们需要这些数字的粗略估计，因为要针对这些数据来优化图形用户界面（虽然我们必须保证其有足够宽的范围）。

举例：用于“支付账单”用况中的用户界面元素

对于“支付账单”用况，我们将说明如何确定参与者需要用到的屏幕上的用户界面元素以及参与者在工作时需要哪类指南。

参与者一定会用到像“账单”（来源于领域类或业务实体）这类用户界面元素。因此，账单是一个用户界面元素，如图7-18所示。注意，账单包括预定日期、支付数额和目的账户等属性。所有这些属性都是决定是否支付账单的参与者所需要的。

另外，在参与者决定要支付哪些账单时，他或她可能希望密切注意账户中的存款数量以避免透支。这就是参与者所需的那种指南和信息的例子。在账单支付期间，用户界面必须显示这些账单，并显示已确定支付的账单如何影响账户余额（如买主账户关联所表示的，该关联是从第6章中的领域类关联买主得来的）。于是，账户是另一个用户界面元素。账户余额以及它随着账单的支付如何变化，由图7-18中的账户属性和“账户”到“账单”用户界面元素之间的支付关联表示（见图7-18）。

⊖ 大致类似于一个类可以参与多个协作以实现不同的用况，该类在每个这样的协作中充当一个角色。

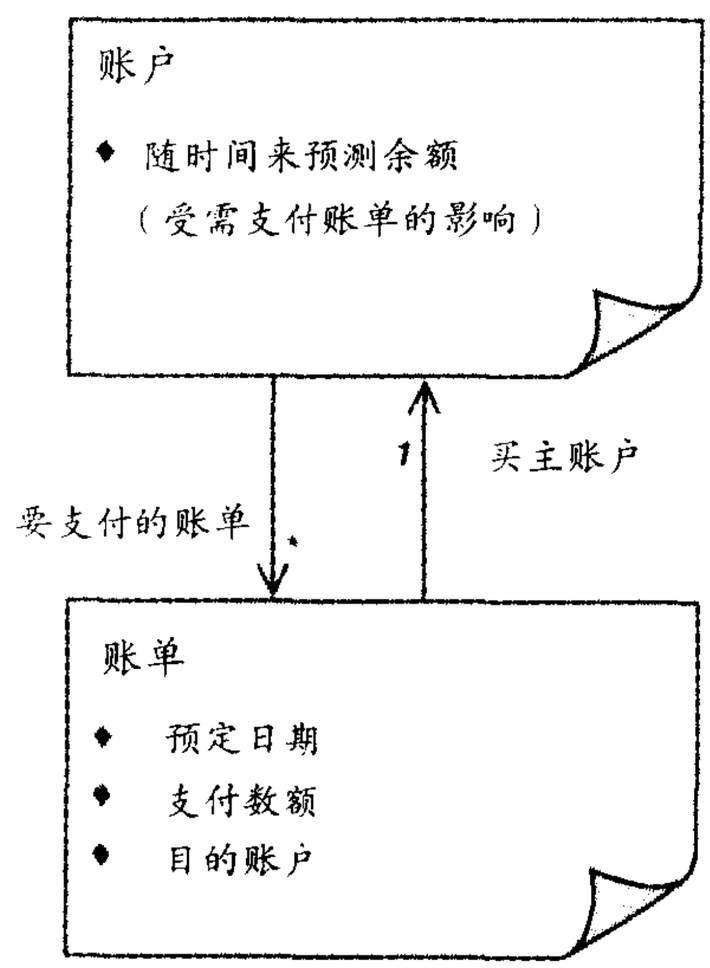


图7-18 “账单”和“账户”类的用户界面元素以及它们的一些属性

一种实用的处理方法是将用户界面元素表示为可粘贴注解[⊖]（如图7-18所示），将它们粘贴在白板上，并对它们进行调整来说明用户界面的外观。然后用户界面设计人员描述参与者在处理这些用况时如何使用这些元素。使用可粘贴注解的优点在于可以展示所需要的数据量。而且，可粘贴注解看起来并不是固定的——很容易来回移动它们，甚至将它们丢弃——这使得用户在提议改变时较为方便。

因此，用户界面设计人员要确保每个用况都可以通过其用户界面元素进行访问。另外，还要确保一个参与者可以访问的整个用况集合具有非常完整、易用而且一致的用户界面。

至此，我们介绍了参与者需要用户界面提供些什么。下面我们来看看实际的用户界面是如何提供这些内容的。

7.4.4.2 进行实际的用户界面设计和构造原型

首先，用户界面设计人员绘制出构成参与者有效用户界面的用户界面元素集的简图。然后，他们描绘将各种用户界面元素组成完整的用户界面所需要的附加元素。这些附加元素可能包括用户界面元素的容器（如文件夹）、窗口、工具和控件；参见图7-19。这些简图可以在逻辑用户界面设计阶段确定了可粘贴注解之后（或有时是同时）进行绘制。

基本用况建模

详细的用况说明是设计用户界面的出发点——有时候是非常有效的出发点。问题是用况说明中常常含有关于用户界面的隐式选择。随后，在用户界面设计人员设计用况的合适用户界面时，可能会受到这些隐式选择的限制。但是，为了给这些用况建立最好的用户界面，必须设法避免陷入这种陷阱。例如，“支付账单”用况开始于“买主通过浏览接收到的账单来激活用

⊖ 从形式化着眼，可以将这种“可粘贴注解”看做是UML中类的一种构造型。但有关用户界面元素的形式化（对象）模型的详细介绍已经超出了本书的范围。

况……”。这样的描述可能会误导用户界面设计人员建立一个包含参与者可以浏览的接收账单的列表。但这可能并不是检查所收到账单的最好方法。相反，买主可能会发现以一种不太明显的方法浏览账单更容易些，例如通过按支付日期水平排列和按支付金额垂直排列的图标显示。

Larry Constantine 提出了一种关于解决用户界面隐式选择问题的方法[14]。他建议用况描述人员首先准备一个只包含基本用况的简要的用况说明，其中不包括任何用户界面隐式选择。那么前面的例子就可以重写为“买主通过检查收到的账单来激活用况……”然后用况界面设计人员可以将这些基本用况用作建立用户界面的输入，而不会受到任何隐式选择的牵制。

这个例子简单阐明了从用况说明中抽取其基本含义时所需要的内容。实际上，要做到这一点不只需要替换说明中某个词。重要的是要避免对以下问题过早做出决定：

- 用于展示某个用户界面元素的技术，如是否使用文本域或列表。
- 参与者的输入顺序，如在输入一个属性后再输入另一个属性。
- 输入和输出所需的设备，例如用鼠标选择某些信息或用监视器显示这些信息。

然后，如果合适的话，用况描述人员可以再快速检查一遍用况说明，以便在基本风格的说明中增加前面所省略的细节。

举例：实际的用户界面设计和原型

用户界面设计人员绘制了以下实际设计的草图，来可视化地显示已确定支付时间的账单对账户余额的影响。账单显示为白色的梯形，当决定对其支付时会减少账户余额。用户界面设计人员之所以选用梯形，是因为它的宽度足够显现并可进行选择，而且它还可以用尖点来表明支付发生的时间。确定了支付时间的账单，如房租账单，在预定日期会使账户余额减少，如图7-19所示。草图还表明了账户中的存款是如何增加的，例如当账户的户主得到薪金收入时。另外，图中还显示了如滚动条和缩放按钮等用户界面控件。

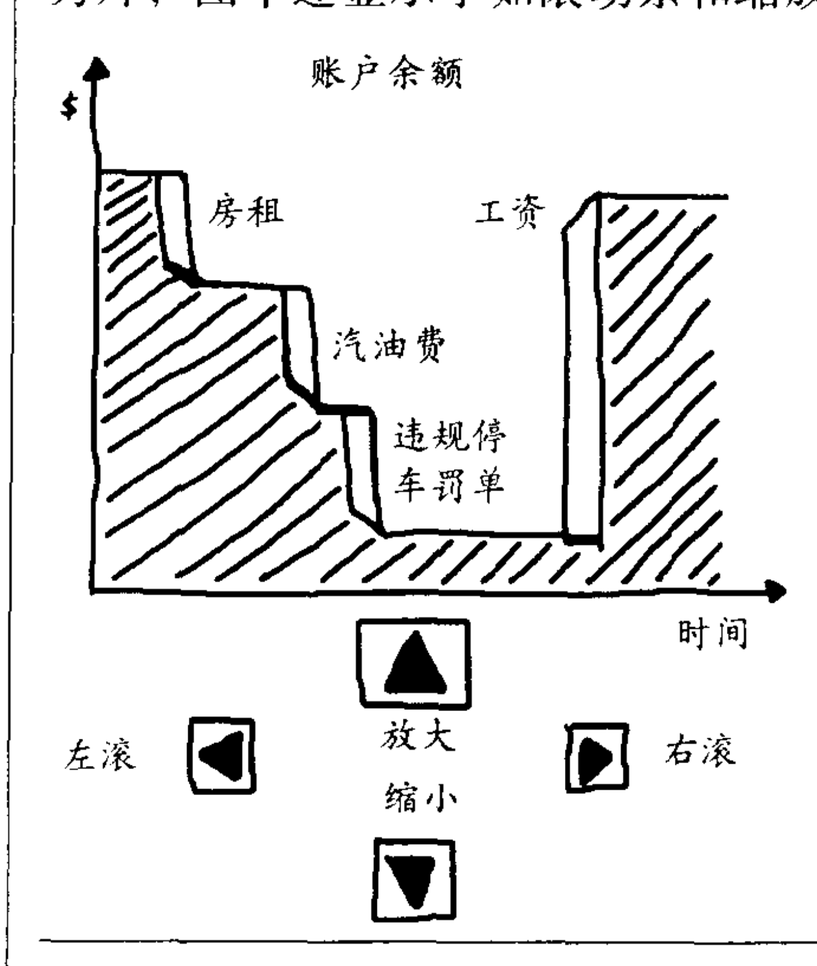


图7-19 对有关界面元素“账户”和“账单”所提议的用户界面。界面显示了账单支付和储蓄凭证将如何引发账户余额的减少和增加。左、右滚动按钮用于移动“时间窗口”，使参与者可以了解账户余额。放大、缩小按钮用于调整图的大小

现在，我们准备为最重要的用户界面元素集构造可执行的原型。这些原型可以用快速原型开发工具进行构造。

可能会有多个原型，或许每个参与者对应一个原型，用来验证每个参与者可以执行他或她所需要的用况。原型设计工作必须按预期的反馈进行调整。在获得很多可用性价值时（例如，为最重要的参与者构造的原型），就可以建立可执行的GUI原型；而在没有得到多少有用信息时，则使用纸上的草图。

通过对原型和草图的早期评审来确认用户界面，可以避免出现很多错误，而以后要改正这些错误需要付出昂贵的代价。原型还能揭示用况说明中的疏漏，以便在设计用况之前对其进行改正。评审人员应该验证每个用户界面是否满足以下条件：

- 允许参与者进行适当的导航。
- 提供一致的外观和感觉以及一致的用户界面处理方法，如Tab键的顺序和加速键。
- 遵循有关的标准，如颜色、按钮大小、工具栏的放置。

注意，实际用户界面的实现（与这里建立的原型相反）是与系统的其他部分并行进行构造的，即在分析、设计和实现工作流期间构造。于是，此处所建立的用户界面原型将作为用户界面的规格说明，并采用符合产品质量的构件来实现该规格说明。

7.4.5 活动：构造用况模型

构造用况模型是为了：

- 提取可以由更具体的（用况）说明所使用的通用的和共享的（用况）功能说明。
- 提取可以扩展更具体的（用况）说明的补充的或可选的（用况）功能说明。

在执行该活动前，系统分析人员已经确定了参与者和用况，用图对它们进行了描述，并从整体上说明了用况模型。用况描述人员已经建立了每个用况的详细说明。这时，系统分析人员可以重新构造用况的完整集合，以使模型更易于理解和处理（见图7-20）。分析人员则应该确定共享的行为以便进行扩展。

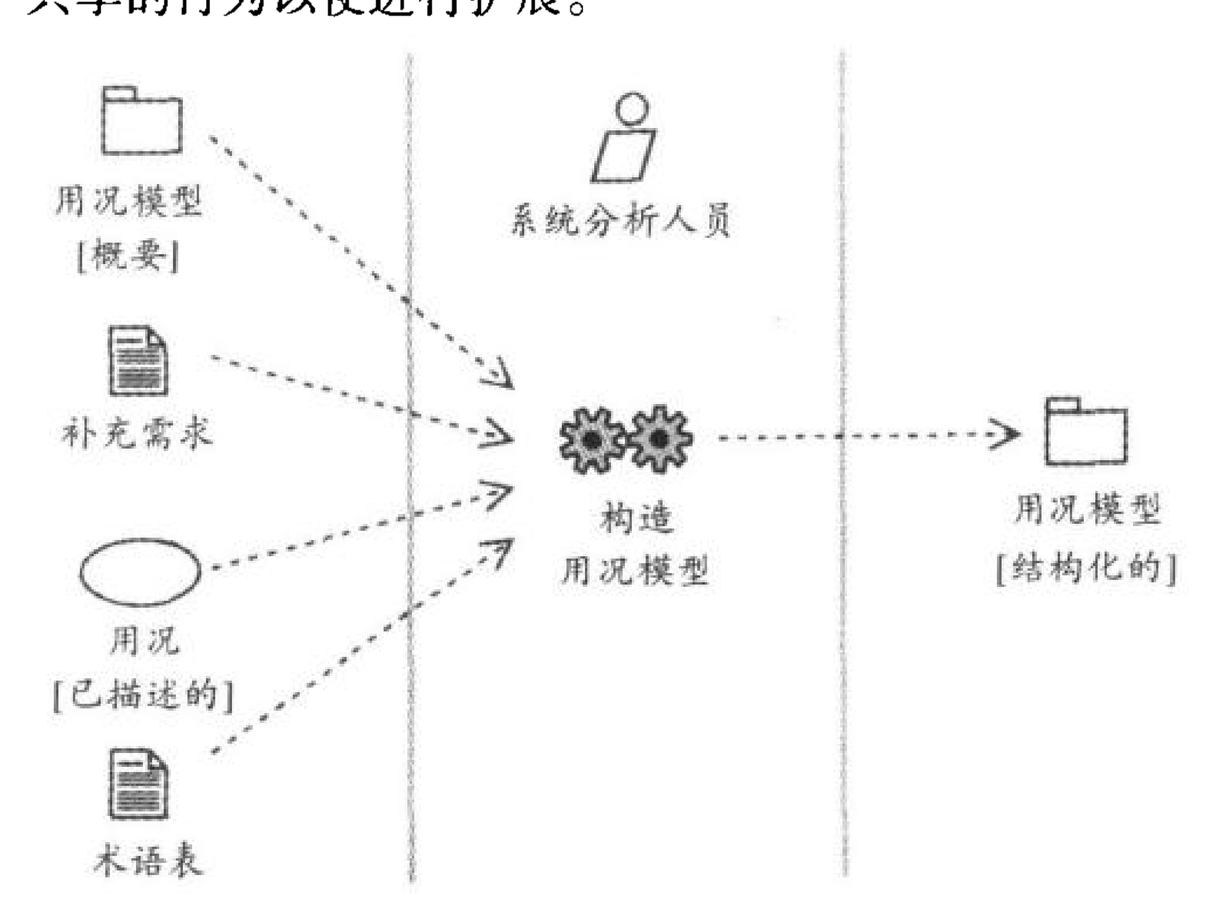


图7-20 确定用况模型中泛化关系的输入及其结果

7.4.5.1 确定共享的功能性说明

在确定并概括每个用况的动作时，还应该确定由多个用况公用的或共享的动作或部分动作。为了减少冗余，可以将这些共享的部分提取出来并放到一个可以由原先的这些用况重用的单独用况中进行描述。我们用泛化（[7]中称之为使用关系）表示这种重用关系。用况之间的泛化是一种继承，因为被泛化用况的实例可以执行泛化用况中描述的所有行为。换句话说，从用况A到用况B的泛化表明用况A的实例也包括由B所确定的行为。

举例：用况之间的泛化

本章前面的图7-12中曾经介绍过，“支付账单”用况是由“执行事务”用况泛化得到的。因此，“支付账单”中描述的动作序列继承了“执行事务”用况中描述的动作序列（参见图7-21）。

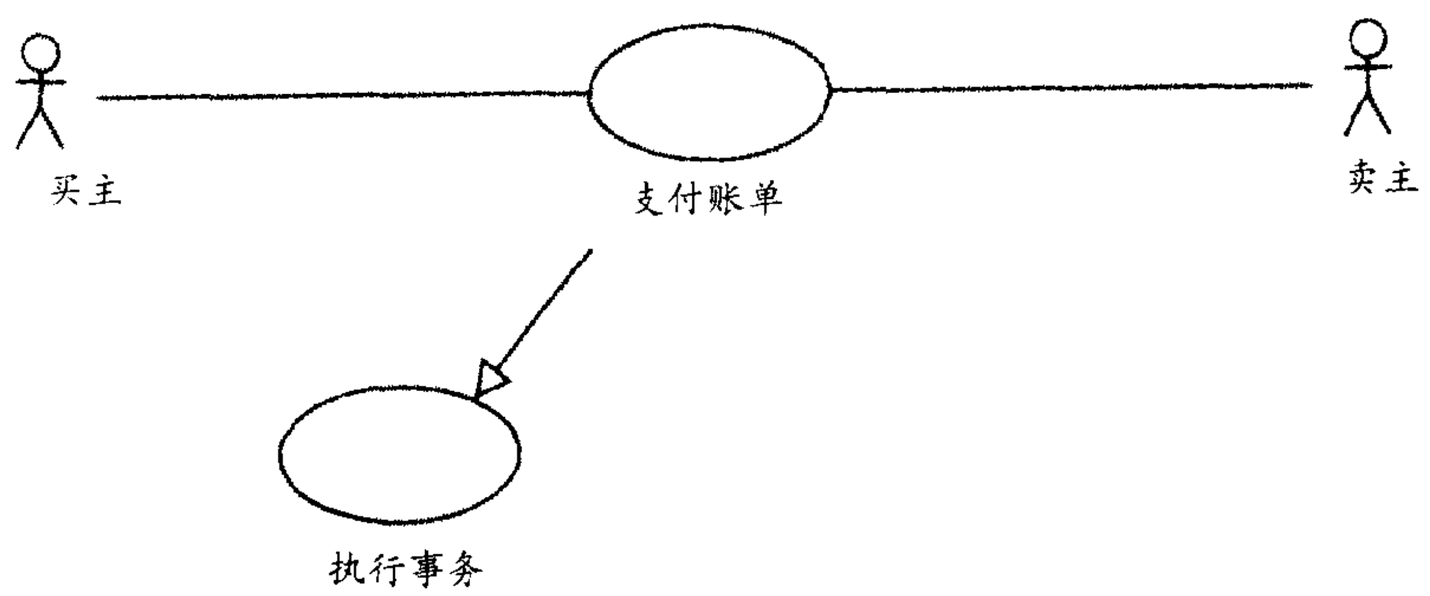


图7-21 “支付账单”用况和“执行事务”用况之间的泛化关系

泛化可用来简化对用况模型的处理和理解，并在整理客户所要求的完整用况时重用“半成品”用况。这样的完整用况称为具体用况（*concrete use case*）。它们由参与者初始化，它们的实例组成了一个由系统执行的完整的动作序列。“半成品”用况的存在只是为了其他用况的重用，并被认为是抽象用况（*abstract use case*）。一个抽象用况本身不能实例化，但一个具体用况的实例可以引用它所重用的抽象用况确定的行为。为了便于理解，我们称这种实例为“真正的”用况，它们是参与者在与系统交互时能够感知到的。

举例：“真正的”用况

在由“执行事务”用况泛化为“支付账单”用况时，可以将“真正的”用况概念化，如图7-22所示。

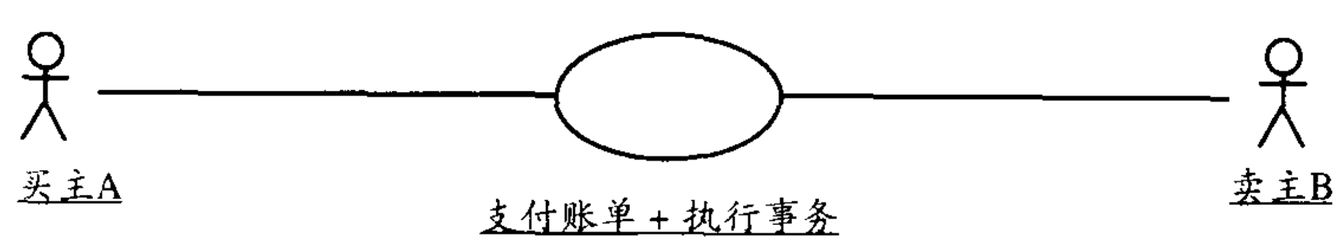


图7-22 由“支付账单”用况和“执行事务”用况形成的“真正的”用况实例，就如参与者实例“买主A”和“卖主B”所感知到的那样

这种“真正的”用况是两个用况（一个具体的用况和一个抽象的用况）泛化后得到的结

果。这种真正的用况代表了与系统交互的参与者可感知到的用况实例的行为。如果模型中包含了更具体的由“执行事务”用况泛化而来的用况，那么就应该存在更为真实的用况。这些真正用况的规格说明之间可能存在着重叠，这些重叠是“执行事务”用况中确定的内容。

注意，这个例子没有揭示出全部事实，因为还存在一个扩展了“支付账单”用况的用况（“支付透支费用”），由此又产生了其他真正的用况。下一节将对此进行讨论。

7.4.5.2　确定补充和可选的功能说明

用况之间的另一种关系是*扩展关系*（*extend relationship*）[7]。扩展模型对用况的动作序列进行了补充。扩展“表现为”补充到用况的原始说明中的内容。换句话说，从用况A到用况B的扩展关系表明用况B的实例可能包含（限于扩展中规定的具体条件）由用况A确定的行为。个别目标用况的多个扩展所确定的行为可能会发生在某个单个的用况实例中。

扩展关系包括扩展的条件和对目标用况中扩展点的引用（即在用况中可能进行补充的位置）。一旦（目标）用况的实例到达了扩展关系引用的扩展点，就可以对扩展关系的条件进行判定。如果满足条件，用况实例所执行的序列将扩展为包含扩展用况的序列。

举例：用况之间的扩展关系

回忆一下本章前面的图7-12以及7.4.1.4节“从整体上描述用况模型”中所给的例子，其中，“支付账单”用况通过“支付透支费用”用况进行扩展。于是，如果出现透支（这是扩展的条件），在“支付透支费用”用况（图7-23）中描述的动作序列就会插入到“支付账单”用况描述的序列中。

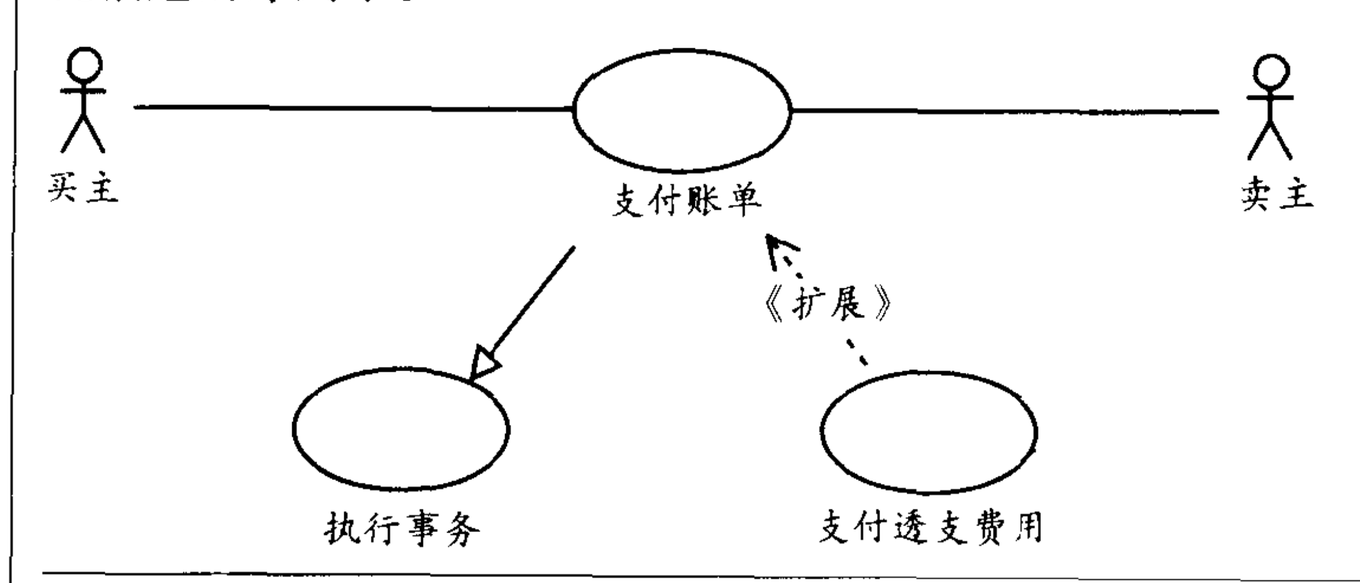


图7-23　“支付账单”用况和“支付透支费用”用况之间的扩展关系

注意，通过扩展关系，可以进一步深入讨论参与者所能感知到的真正用况。通过将扩展用况“支付透支费用”的扩展关系应用到目标用况（即由“执行事务”用况泛化得到的“支付账单”用况），可以得到另一个由三个用况合并成的真正用况（见图7-24）。

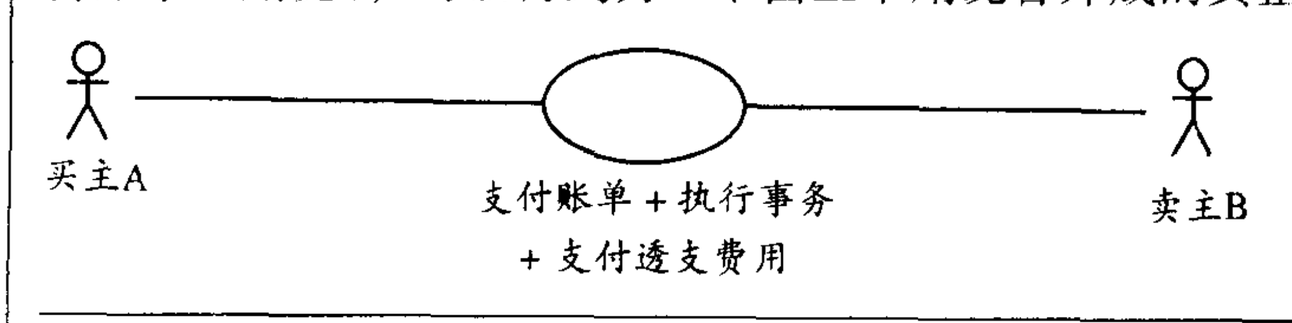


图7-24　由参与者实例“买主A”和“卖主B”所感知到的“真正的”用况实例，它由“支付账单”用况和“执行事务”用况加上“支付透支费用”用况一起构成

可以说真正的用况本身就是用况，它们是通过对模型中用况应用泛化和扩展关系所得到的。真正的用况是那些给用户带来价值的用况。于是，在7.4.1节“活动：确定参与者和用况”中提到的好用况的准则（向一个具体的参与者提供一个有价值的可见结果的用况）只是针对真正的用况而言的。因此，必须为好的具体用况、抽象用况和扩展用况确定单独的准则。具体用况不应该描述与其他具体用况共享的（重要）行为。抽象用况通过提供可共享和可重用的行为以便更易于确定具体用况。扩展用况对其他用况确定了补充行为，无论这些行为是可选的还是强制性的。

为了理解泛化和扩展关系，我们已经介绍了真正用况的概念。一旦已经确定具体的、抽象的和扩展的用况，就可以将它们组合起来得到真正的用况。但当开始对一个新系统建模时，一般是从相反的方向进行处理，即从真正的用况开始确定共享行为以便从抽象用况中区分出具体用况，并确定补充行为，把它看作其他用况的扩展。

有关泛化（也称作使用关系（*uses relationship*））和扩展关系以及何时使用哪种关系的更为全面的讨论请参阅[7]。

7.4.5.3 确定用况之间的其他关系

用况之间还存在着其他关系，例如包含关系（*include relationship*）[12]。为简单起见，可以将这种关系视为一种逆扩展关系，对用况提供明显的和无条件的扩展。另外，当包含一个用况时，将对所包含用况的行为序列和属性进行封装，而且禁止改变或访问，即只有包含用况的结果（或功能）可供利用，这与使用泛化关系相比是不同的。本书将不再过多地叙述这些关系的细节，而只是给出一些注意事项：

- 用况的结构以及它们的关系应尽量反映真正的用况（正如前面所讨论的）。这种结构偏离真正的用况越多，就越难理解用况及其用途；这不仅对外部的合作伙伴（如用户和客户）是如此，对开发人员自己也是如此。
- 需要将每个单独的用况视为一个单独的制品。需要某些人（即用况描述人员）负责对其进行说明；并且在后续的工作流（分析和设计）中，需要用单独的用况实现来实现这些用况（将在第8章、第9章中介绍）。为此，用况不能太小或太多，否则将需要大量额外的管理开销。
- 避免从功能上分解用况模型中的用况。最好是在分析模型中通过精化每个用况来完成这项工作。这是因为，由用况定义的功能会被按照面向对象的方式分解为概念化分析对象之间的协作，第8章将对此进行介绍。如果需要，这种分解会使人们更加深入地理解需求。

7.5 需求工作流小结

在本章和前一章中，我们介绍了如何捕获系统的需求而得到：

- 设置系统语境的业务模型或领域模型。
- 捕获功能性需求和具体针对个别用况的非功能性需求的用况模型。用况模型由综合说明、

一组图以及每个用况的详细说明来描述。

- 代表用户界面设计的每个参与者的用户界面草图和原型的集合。
- 通用的、非具体用况专用的补充需求规格说明。

这些结果对于后续的工作流（分析、设计、实现和测试工作流）是一个非常好的开端。用况将驱动开发工作逐次迭代地通过这些工作流。针对用况模型中的每个用况，在分析和设计阶段将确定与各个用况相匹配的用况实现，在测试阶段将为各个用况确定一组测试用例。这样，用况就将不同的工作流无缝地连接起来。

第8章将转移到工作流链中的下一个环节（分析工作流），此时将按照对象对用况重新进行形式化描述，以便对需求获得更好的理解，也为系统的设计和实现作好准备。

7.6 参考资料

[1] Ahlqvist Stefan and Jonsson Patrik, "Techniques for systematic design of graphical user interfaces based on use cases," *Proceedings OOPSLA 96.*

[2] Grady Booch, James Rumbaugh, and Ivar Jacobson, *Unified Modeling Language User Guide.* Reading, MA: Addison-Wesley, 1998.

[3] Alistair Cockburn, "Structuring use cases with goals," *Report on Analysis and Design (ROAD),* 1997.

[4] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Övergaard, *Object-Oriented Software Engineering: A Use-Case–Driven Approach.* Menlo Park, CA: Addison-Wesley, 1992. (Revised fourth printing, 1993).

[5] Ivar Jacobson, Maria Ericsson, and Agneta Jacobson, *The Object Advantage—Business Process Reengineering with Object Technology.* Menlo Park, CA: Addison-Wesley, 1994.

[6] Jacobson I., "Basic use case modeling," *Report on Analysis and Design (ROAD),* July–August, vol. 1 no. 2, 1994.

[7] Jacobson I., "Basic use case modeling (continued)," *Report on Analysis and Design (ROAD),* vol. 1 no. 3, September–October 1994.

[8] Ivar Jacobson and Magnus Christerson, "Modeling with use cases—A growing consensus on use cases." *Journal of Object-Oriented Programming,* March–April 1995

[9] Jacobson I., K Palmqvist, and S. Dyrhage, "Systems of interconnected systems," *Report on Analysis and Design (ROAD),* May–June 1995.

[10] Ivar Jacobson, "Modeling with use cases—Formalizing use-case modeling," *Journal of Object-Oriented Programming*, June 1995.

[11] James Rumbaugh, Ivar Jacobson, and Grady Booch, *Unified Modeling Language Reference Manual,* Reading, MA: Addison-Wesley, 1998.

[12] OMG Unified Modeling Language Specification. Object Management Group, Framingham, MA, 1998. Internet: www.omg.org.

[13] Ivar Jacobson, Martin Griss, and Patrik Jonsson, *Software Reuse: Architecture, Process and Organization for Business Success.* Reading, MA: Addison-Wesley, 1997.

[14] L.L. Constantine and L.A.D. Lockwood, *Software for Use: A Practical Guide to*

the Models and Methods of Usage-Centered Design. Reading, MA: Addison-Wesley, 1999.

[15] CCITT, Specification and Description Language (SDL), Recommendation Z.100. Geneva, 1988.

[16] John Carrol, *Scenario-Based Design,* New York: John Wiley & Sons, 1995.

[17] David Harel, Michal Politi, *Modeling Reactive Systems With Statecharts: The STATEMATE Approach*, New York: McGraw-Hill, 1998.

第8章
分　　析

8.1 引言

在分析阶段，我们通过精化和组织需求捕获阶段所描述的需求来对其进行分析。这样做的目的是为了更精确地理解需求，也是为了得到一个易于维护且有助于确定系统结构（包括其构架）的需求描述。

在正式论述本章内容之前，先回想一下需求捕获的结果。我们记得捕获需求的首要原则是使用客户的语言（参见6.2节）。而且，如在第7章中讨论的，我们认为用况为这种语言奠定了一个良好的基础。但是，即使我们与客户就系统功能达成了共识，但很可能还存在着尚未解决的系统需求问题。这是由于在捕获需求过程中使用了直观但不精确的客户语言而不得不付出的代价。为了弄清楚需求捕获阶段所描述的系统需求可能遗漏哪些“未解决的问题”，有必要回顾一下与客户有效地交流系统功能的方法：

1) 用况必须尽量保持彼此独立。为了做到这一点，就不能陷入到有关用况之间干扰、并发与冲突的细节中，这些情况经常出现在用况之间竞争系统内部共享资源时（参见7.2.3节）。例如，“存款”和“取款”用况都要访问同一储户的账户。或者，当一个参与者混合使用多个可能产生无法预测行为的用况时，就可能出现冲突。例如，当一个电话用户使用“预置唤醒呼叫”用况后，接着使用“呼入改向”用况去对另一个电话用户进行电话唤醒呼叫。所以在需求捕获阶段，用况之间的干扰、并发与冲突问题可能没有得到解决。
2) 用况必须使用客户语言来描述。为了做到这一点，在用况说明中应主要使用自然语言，并慎用诸如状态图、活动图和交互图（参见7.4.3.2节）等较为形式化的符号。但是，仅仅使用自然语言大大削弱了表达能力；在需求捕获阶段，许多本来可以用更形式化的图符来精确描述的细节问题，可能因此得不到解决，或者只能进行模糊地描述。
3) 每个用况的构造都是为了构成完整而且直观的功能规格说明。为了做到这一点，需要对用况（也就是针对需求）进行组织以便直观地反映系统提出的“真实”用况。例如，不能为了消除冗余而将用况设计得太小、太抽象或太不直观。尽管这样做是可能的，但我们必须在用况说明的易理解性与可维护性之间进行权衡（参见7.4.5.3节）。因此，关于所描述的需求间的这种冗余问题可能无法在需求捕获期间解决。

由于存在着上述未解决的问题，分析的主要目的就是通过进一步深入分析需求来使问题得以解决，但与需求捕获相比，最主要的差别在于它可以使用开发人员的语言来描述这些结果。

因此，我们可以在分析阶段更多地探究系统内部，从而解决用况间的干扰以及类似的问题（上述第1条）。在分析阶段，我们还可以采用更为形式化的语言对系统需求中的细节问题（上述第2条）进行描述，我们称之为“精化需求”。

此外，在分析阶段，我们可以以易于理解、易于组织、易于更改和易于重用（总之，易于维护）的方式来组织需求（上述第3条）。这种结构（基于分析类和分析包）与需求中确定的结构（基于用况）是正交的。然而，这两种不同的结构间具有无缝的可跟踪性，因此，可以按照不同的详细程度跟踪同一需求的不同描述，而且易于保持彼此间的一致。实际上，这种无缝的可跟踪性是在用况模型中的用况和分析模型中的用况实现之间定义的；在本章后面将详细讨论这个问题（参见表8-1）。

表8-1 用况模型与分析模型的简单比较

用况模型	分析模型
使用客户的语言进行描述	使用开发人员的语言进行描述
系统的外部视图	系统的内部视图
通过用况来构造；提供外部视图的结构	通过构造型类和包来构造；提供内部视图的结构
主要用于客户与开发人员之间签署合同时明确系统应该做什么，不应该做什么	主要为开发人员使用，以理解如何构造系统，即怎样设计和实现系统
需求中可能存在冗余和不一致等问题	需求中不应该存在冗余和不一致等问题
捕获系统的功能，包括对构架重要的功能	概述如何实现系统的功能，包括对构架层重要的功能；是设计阶段的切入点
定义在分析模型中进一步进行分析的用况	定义用况实现，每个用况实现代表对用况模型中一个用况的分析

最后，通过分析得到的需求结构还可用作构造整个系统（包括其构架）的基本输入，因为我们希望实现一个整体可维护的系统，而不仅仅是描述系统需求。

在本章中，我们将通过分析、精化和组织需求来更细致地说明我们的用意。首先我们对分析做一个简单的“定位”（参见8.2节），描述在软件生命周期的不同阶段中分析的作用（8.3节）。然后介绍分析过程中涉及的制品（8.4节）和工作人员（8.5节）（参见图8-1）。最后，我们将描述分析工作流（8.6节）。

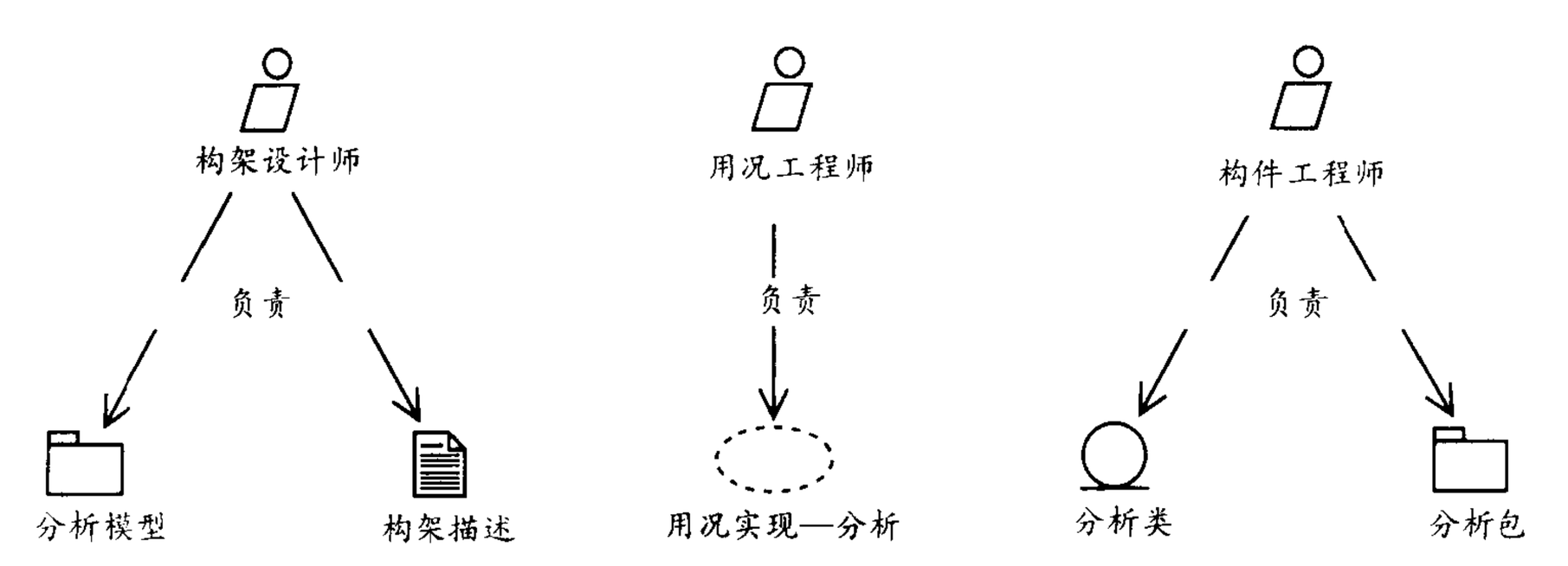


图8-1 分析中涉及的工作人员与制品

8.2 分析概述

分析阶段中所用的语言是基于一种称为分析模型（*analysis model*）的概念层对象模型。分析模型有助于我们按前面所提出的路线（8.1节）精化需求，并探究系统的内部，包括其内部的共享资源。实际上，内部资源可以表示为分析模型中的对象，例如“存款”和“取款”用况都要访问的储户账户。而且，分析模型提供了更强的表达能力和形式化方法，例如用于描述系统动态结构的交互图。

如前面所介绍的，分析模型还有助于我们组织需求（8.1节），并能提供一种侧重于可维护性的结构，例如具有对需求变化的柔性及可重用性。（本章后面的部分将介绍建立对需求变化具有柔性并包含可重用元素的分析模型的原则。）这种结构不仅有益于需求的维护，而且还可作为设计和实现活动（在第9章和第10章中介绍）的输入。我们在构造系统并决定进行设计和实现时，应该尽量维持这种结构。基于以上原因，尽管分析模型本身就是一种模型，但还是可以将它看作是设计模型的切入点。在设计阶段维持分析模型的这种结构，我们会得到一个整体上也可以维护的系统：它对于需求变化具有柔性，并包含构造同类系统时可重用的元素。

但是，这里应该着重指出的是：分析模型是进行抽象的过程，应避免去解决某些问题和处理某些需求，最好将它们推迟到设计和实现阶段（附录C；也可参见8.2.1节）去完成。因此，分析模型中提出的结构不可能一直维持下去，需要在设计与实现期间对其进行一定的处理和折衷，在第9章和第10章中将讨论这些问题。“维持结构”在实际操作中行不通的原因很简单，即在设计阶段要考虑实现平台：程序设计语言、操作系统、预制框架、遗留系统等。从效益来看，当过渡到设计模型并构造系统时，通过修改分析模型的结构可以获得一个较好的构架。

8.2.1 为什么分析不同于设计与实现

你现在也许会问，为什么不在设计和实现系统的同时进行需求分析呢？原因在于：设计与实现远比分析（即精化和组织需求）复杂，因此需要分开来考虑。在设计阶段，我们必须构造系统并确定其表现形式（包括其构架）。该表现形式实现了系统的所有需求；包括了经过编译并集成到系统的可执行版本中的代码构件；在长期运行过程中可以维护该表现形式——它能经受时间、变化和进化的考验；并且具有完整性。

因此，在设计阶段，我们必须对系统应如何处理诸如性能和分布等需求作出决定，并回答诸如“如何优化这个规程使其执行时间不超过5毫秒？”和“如何在网络节点上部署这段代码而不会加重网络的通信负载？”等问题。在设计阶段还有很多其他类似的问题必须加以处理，例如，如何有效地利用诸如数据库和对象请求代理等已有构件，如何将这些构件集成到系统构架中去，以及如何恰当地使用程序设计语言等。这里我们不一一列举设计和实现阶段将出现的所有额外的问题，这些将在第9章和第10章中进行讨论。我们希望已经阐明了我们的观点，即设计与实现远不只是通过精化和组织需求来对其进行分析；实际上，设计与实现要构造能够实现所有需求（包括非功能性需求）的系统。可以给出这样一种概念：分析模型中所做的抽象与设计

模型中的丰富细节进行比较，它们的模型元素数量的比例一般为1：5。

因此我们简单地认为，在开始进行设计与实现之前，必须对需求有一个精确而细致的了解，达到（通常情况下）客户不必关心的精确和细致的程度。此外，如果已经具有一个可以用作构造系统的输入的需求结构，那就更好了。所有这些都是通过分析得到的。

简而言之，通过进行分析，我们实现了对所关心问题的分离：通过界定在后续的设计与实现活动中应解决的问题和应做出的决定来准备和简化这些活动。而且，通过分离这些有关问题，开发人员在软件开发工作的初期就能“面对难题”，从而可以避免因试图一次解决太多问题（包括因需求不清或理解有误而根本无法解决的问题）而可能引起的开发工作的停顿。

8.2.2 分析目的：小结

以分析模型的形式来分析需求是很重要的，其原因如前面所介绍的：

- 分析模型产生的需求规格说明比从需求捕获得到的结果（包括用况模型）更为精确。
- 分析模型使用开发人员的语言来描述，因此可以引入更为形式化的方法，用于探究系统内部的运作方式。
- 分析模型以易于理解、易于编制、易于改变的方式，总之，以易于维护的方式来组织需求。
- 分析模型可以看作是设计模型的切入点（尽管分析模型本身就是一种模型），因此，它是设计与实现阶段中构造系统时的基本输入。这是因为系统整体上应该具有可维护性，而不仅仅是其需求的描述。

8.2.3 何时使用分析的具体实例

除了前面所介绍的之外，我们将给出更为具体的例子来说明何时使用分析以及如何利用其结果（即分析模型）：

- 通过单独地进行分析，而不是将它作为设计与实现的一个完整部分来执行，可以用较低的成本完成大部分系统分析工作。然后可以利用分析的结果来规划后续的设计与实现工作；可以是多个连续的增量，其中每个增量只设计与实现系统的一小部分；也可以是多个并发的增量，可能是由在地理位置上分开的开发组设计和实现的。如果没有分析的结果，确定和规划这些增量可能更困难。
- 分析提供了系统的概貌，而通过研究设计或实现的结果来了解系统可能会更困难一些，因为期间引入了太多的细节（回忆一下8.2.1节所介绍的1：5的比例）。这样的系统概貌对于参与系统的新成员和维护系统的开发人员来说是非常有价值的。例如，一个组织使用类似于第3章和第4章中描述的原则开发了一个大型系统（该系统包含上千个服务子系统）之后，为了更好地理解这个已开发完成的系统，我们引入一个分析模型。首席信息主管（CIO）总结经验时说：“由于得益于分析模型，我们现在培养一个构架设计师只需要两年而不再是五年。”对于较小的系统，培训时间以月而非以年计，但比例应该是相同的。

- 系统的某些部分存在备选的设计或实现。例如，生死攸关的系统（如飞机控制系统或铁路控制系统）可能由几个不同的程序组成，这些程序对同一操作并发地进行运算，只有当这些计算得出相同的结果时，才会去执行一个重要的操作。另一个例子是，客户希望两个或多个供应商（或分销商）在不同地点提供软件；客户希望两家竞争的软件机构针对同一个规格说明进行投标。通常，当使用不同的技术（如程序设计语言或可以在不同平台上运行的构件）不止一次地实现系统的某个部分时就会出现这种情况。那么，分析模型能够为这些备选的实现提供一个概念上精确而统一的视图。很显然，在这种情况下，对分析模型的维护应该贯穿系统的整个生命周期。
- 系统构造时使用了一个复杂的遗留系统。可以按照分析模型对该遗留系统或其中的一部分进行重组，以便开发人员无须钻研其设计与实现的细节就可以了解该遗留系统，并且用该遗留系统作为一个可重用的构造块来构造新系统。对这样一个遗留系统的设计和实现全部进行重组可能是非常复杂和昂贵的，而且并无多大益处，特别是当该遗留系统无需修改而是采用过时的技术实现时。

8.3 分析在软件生命周期中的作用

最初的细化迭代重点在于分析（参见图8-2）。它有助于建立一个可靠且稳定的构架，并利于深入理解需求。之后，在细化阶段的末期与构造阶段，当构架已经稳定并且需求明确后，重点便转移到设计与实现上。

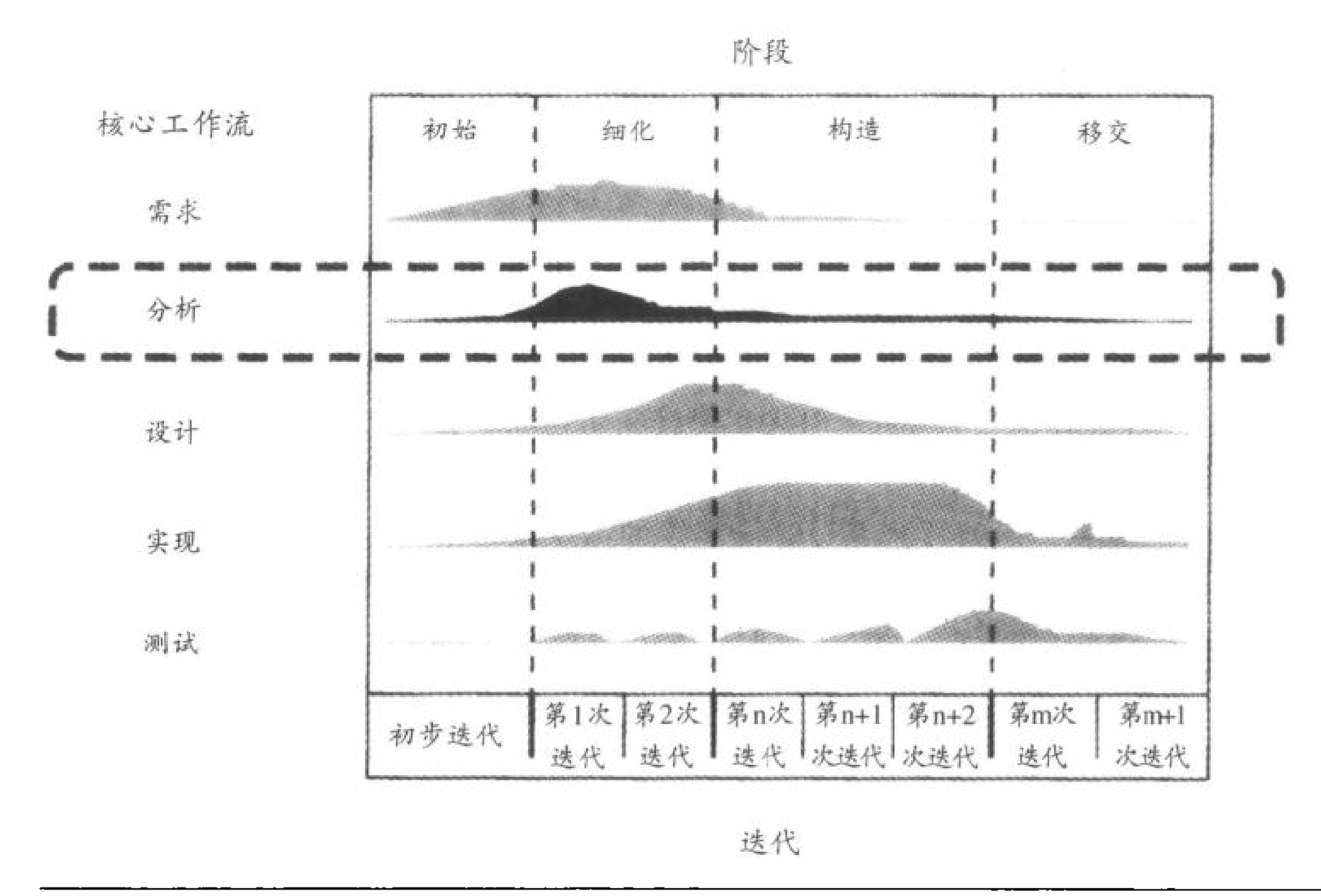


图8-2 分析的焦点

对于每个项目，都应该以某种方式达到分析的目的和目标。但是，具体如何看待和使用分析，对于每个项目可能各不相同。下面，我们来看三种不同的基本方式：

1) 项目使用分析模型（本章后面将详细介绍）来描述分析结果，并在整个软件生命周期内维护该模型的一致性。可以在项目每次迭代中保持这种一致性，以获得8.2.3节中所概括的益处。
2) 项目使用分析模型来描述分析结果，但仅将其视为过渡性的中间工具，可能重点在细化阶段。此后，在构造阶段，随着设计与实现活动的展开，就不再维护分析模型。而是将任何遗留的“分析问题”作为所得到的设计模型中设计工作的一个完整部分予以解决（第9章将集中介绍设计模型）。
3) 项目根本就不使用分析模型来描述分析结果，而是将分析需求作为需求捕获或设计的一个完整部分。第一种方式要求用况模型具有更为形式化的方法。如果用户能理解这一结果，则将无可非议（尽管我们认为这种情况很少出现）。第二种方式会使设计工作复杂化，其原因在8.2.1节中已经说明。然而，如果需求非常简单而且/或者得到了透彻理解，或者如果系统的表现形式（包括其构架）很容易确定，又或者如果开发人员对需求有一种直觉且正确的理解，并能以相当简明的方式构造出一个满足这些需求的系统，那么，这种方式就是可行的。同样，我们认为这种情况也很少出现。

当在前两种方式中进行选择时，我们必须权衡在多次迭代以及不同阶段维持分析模型的花费。因此，我们有必要确定一个恰当的效益折衷方案，并决定是停止更新分析模型（也许可以在细化阶段刚结束时），还是在系统剩余的生命周期内继续保持并维护分析模型。

关于第三种方式，我们认为，项目不仅可以避免维护分析模型的花费，而且还可以避免最初引入分析模型所付出的代价（如培训开发人员使用该模型以及使他们从中获取经验所投入的时间与资源上的代价）。但是，正如我们前面所指出的，在项目初期使用分析模型所带来的收益至少会超过引入分析模型所付出的代价；因此，这种方式只在极少数情况下（如对于非常简单的系统）使用。

8.4 制品

8.4.1 制品：分析模型

在8.2节的开始曾介绍过分析模型。由分析模型添加的结构可以用图8-3所示的层次结构来定义。

分析模型由代表该模型顶层包的分析系统来表示。使用其他分析包可将分析模型组织成为更易于管理的若干部分，这些部分代表了对系统设计中的子系统或可能是整个系统的某一层的抽象。分析类则代表了对系统设计中的类或子系统的抽象。在分析模型中，用况是通过分析类及其对象实现的，并由分析模型中的各种协作来表示，并标志为*用况实现—分析*。在8.4.2节～8.4.5节中将详细介绍分析模型中的制品。

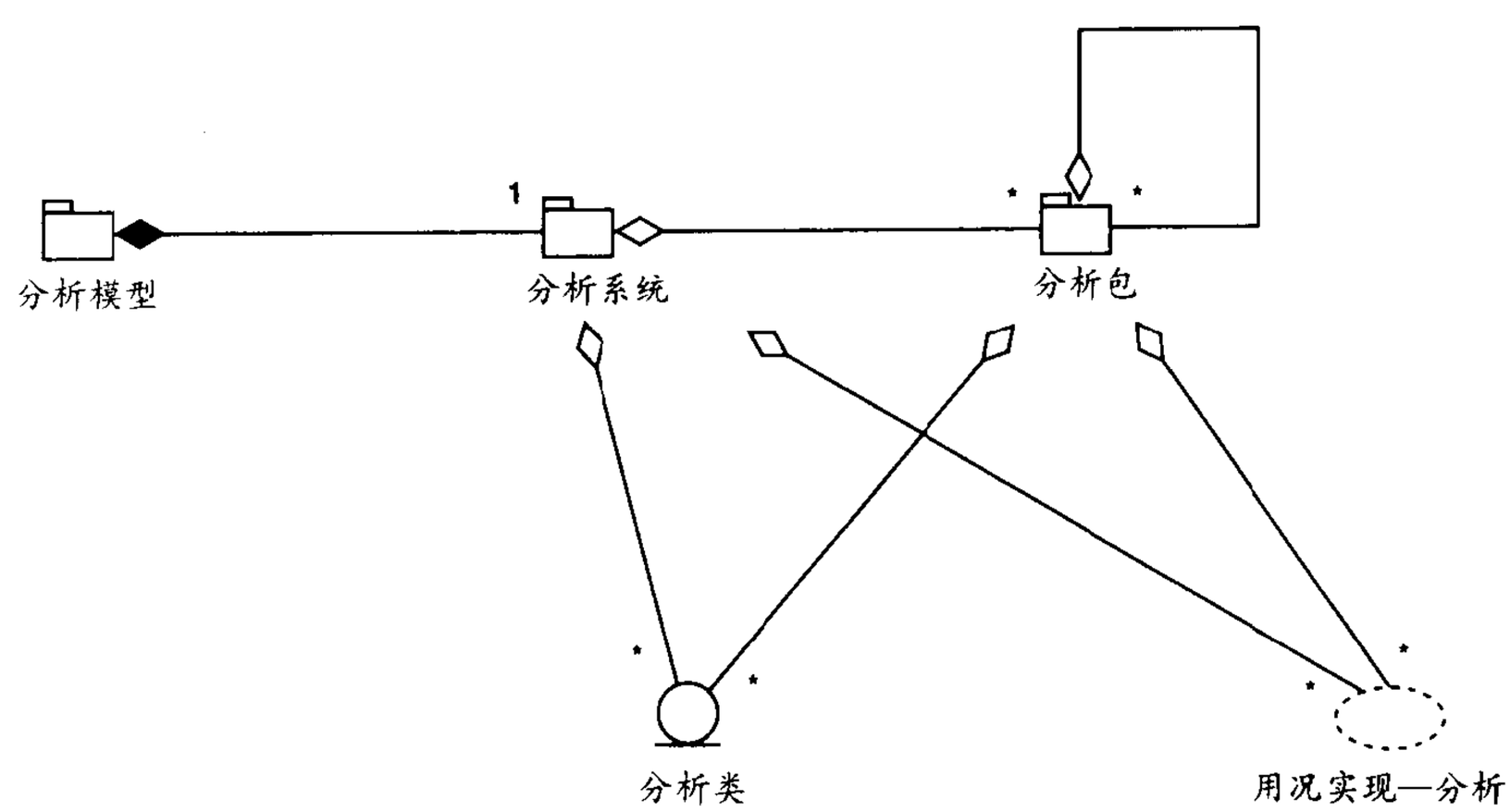


图8-3 分析模型是包含分析类和用况实现的分析包的层次结构

8.4.2 制品：分析类

分析类代表了对系统设计中的一个或几个类或若干子系统的抽象。这种抽象具有如下特点：

- 分析类侧重于处理功能性需求，而把非功能性需求推迟到后续的设计与实现活动中将这些需求标识为类的具体需求时再实现。这使得分析类在问题域语境中更加突出、更加“概念化”，而且与其对应的设计与实现类相比具有更大的粒度。
- 分析类很少根据操作及其特征标记来定义或提供接口，而是通过较高的、非形式化层次的职责来定义其行为。职责是对由类定义行为的内聚子集的文本描述。
- 分析类定义属性，但这些属性还处于较高的层次。通常这些属性的类型是概念性的，而且应从问题领域来考虑；而设计与实现类的属性类型通常就是程序设计语言的类型。而且，分析期间定义的各种属性通常在设计与实现阶段对应各种类。
- 分析类涉及到关系，尽管这些关系和设计与实现阶段的对应部分相比更加概念化。例如，关联的导航性在分析中并不十分重要，但在设计时却是必需的。又如，可以在分析中使用泛化，但在设计阶段如果得不到程序设计语言的支持就无法使用泛化。
- 分析类总能符合三种基本构造型中的一种：边界类、控制类或实体类（参见图8-4）。每一种构造型都有具体的语义（很快会进行说明），从而得到一种有效、一致的方法来确定和描述分析类，并有助于创建健壮的对象模型与构架。然而，以这种清晰、直观的方式来构造设计与实现中的各种类是非常困难的。这是因为它们要处理非功能性需求，它们“存在于解决方案领域的语境中”，而且通常使用程序设计语言的语法及类似的低层技术来描述。

这三种构造型在UML中已经标准化，并已用来帮助开发人员区别对待不同的类[3]。每种构造型都有自己的符号，如图8-5所示。

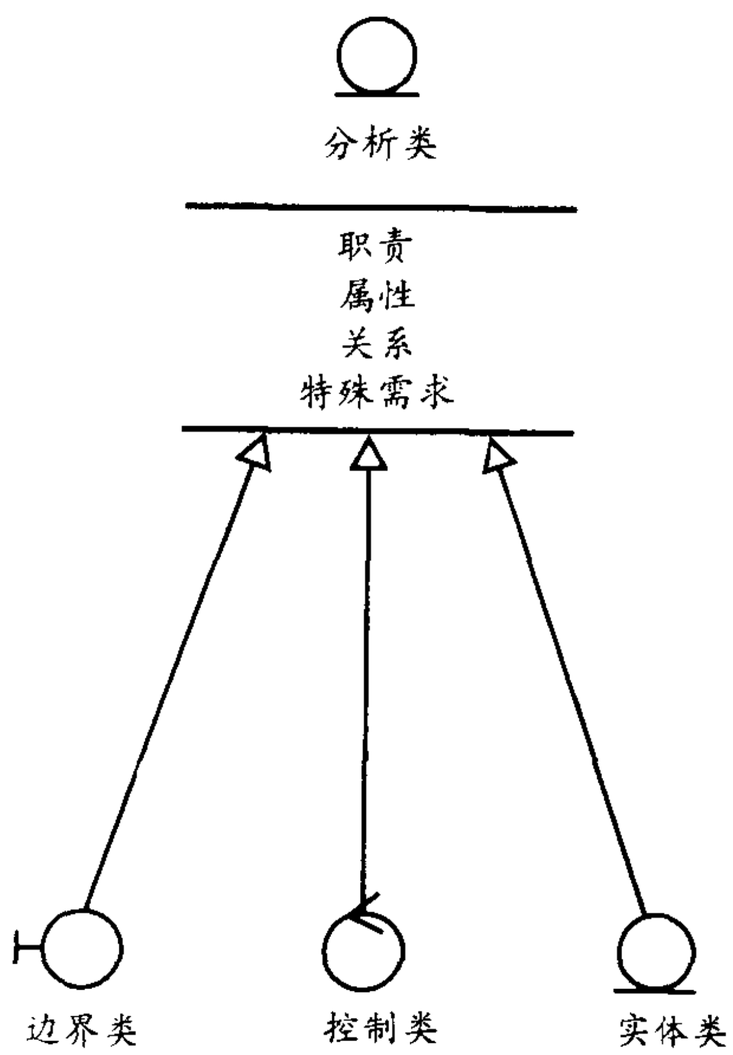


图8-4 分析类的关键属性和子类型（即构造型）

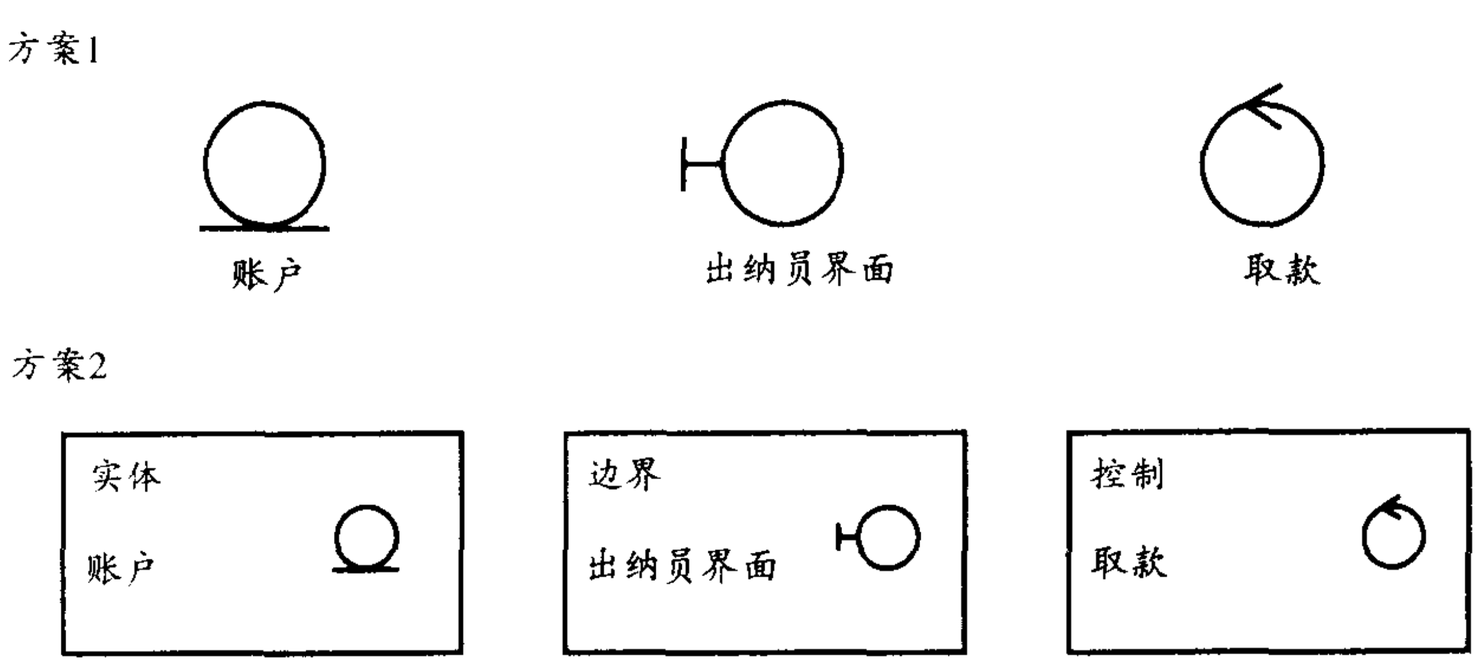


图8-5 UML提供三种供我们在分析阶段使用的标准类的构造型

8.4.2.1 边界类

边界类用于建立系统与其参与者（即用户和外部系统）之间交互的模型。这种交互通常包括接收来自用户和外部系统的信息与请求以及将信息与请求提交到用户和外部系统。

边界类对系统中依赖于参与者的部分建模，也就是说，边界类用于阐明和收集系统的边界需求。这样，通常就把用户界面或通信接口的变化隔离在一个或多个边界类中。

边界类经常代表对窗口、窗体、窗幕、通信接口、打印机接口、传感器、终端以及（可能是非面向对象的）API（应用程序编程接口）等的抽象。边界类仍然应该保持在一个较高的概念化层次上，例如，不要去深究用户界面中每个窗口部件。注意，只要边界类能够说明通过交互（即在系统与参与者之间来回传递的信息与请求）所实现的目标就足够了，而不必描述如何具体实现这种交互，因为这是在后续的设计与实现活动中所考虑的问题。

每个边界类至少应该与一个参与者有关，反之亦然。

举例："支付请求用户界面"边界类

下面称为"支付请求用户界面"的边界类用于支持"买主"参与者与"支付账单"用况之间的交互（参见图8-6）。

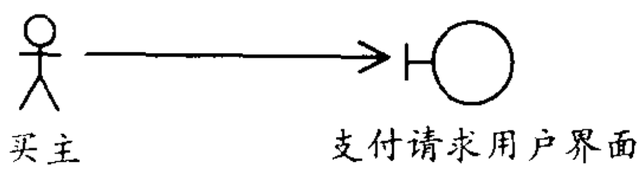


图8-6　"支付请求用户界面"边界类

"支付请求用户界面"允许用户浏览应付款的账单，核对账单的细节，然后向系统请求支付账单（通过确定支付时间）。"支付请求用户界面"还允许用户废弃买主不愿支付的账单。

下面我们举例说明该边界类如何与系统的"内部"（即控制类与实体类）发生关联。

8.4.2.2　实体类

实体类用于对长效且（经常是）持久的信息建模。实体类主要是对诸如个体、实际对象或实际事件的某些现象或概念的信息及相关行为建模。

大多数情况下，实体类是直接从业务对象模型（或领域模型）中相应的业务实体类（或领域类）得到的。然而，实体类与业务实体类之间的主要区别在于：前者表示由所研究的系统处理的对象，而后者通常表示业务（和问题领域）中的对象。因此，从某种意义上来说，实体类反映的信息有助于开发人员设计和实现系统，包括对系统持久性的支持。而业务实体类（或领域类）只是描述了系统的语境，它所捕获的信息根本不能在系统中处理。

实体对象不一定是被动的，有时可能具有与它所表示的信息有关的复杂行为。实体对象能够将变化与它们所表示的信息隔离开。

实体类经常表示为一种逻辑数据结构，并有助于理解系统所依赖的信息。

举例："账单"实体类

下面称为"账单"的实体类用于表示账单。该实体类与用户用来浏览和处理账单的"支付请求用户界面"边界类相关联；如图8-7所示。

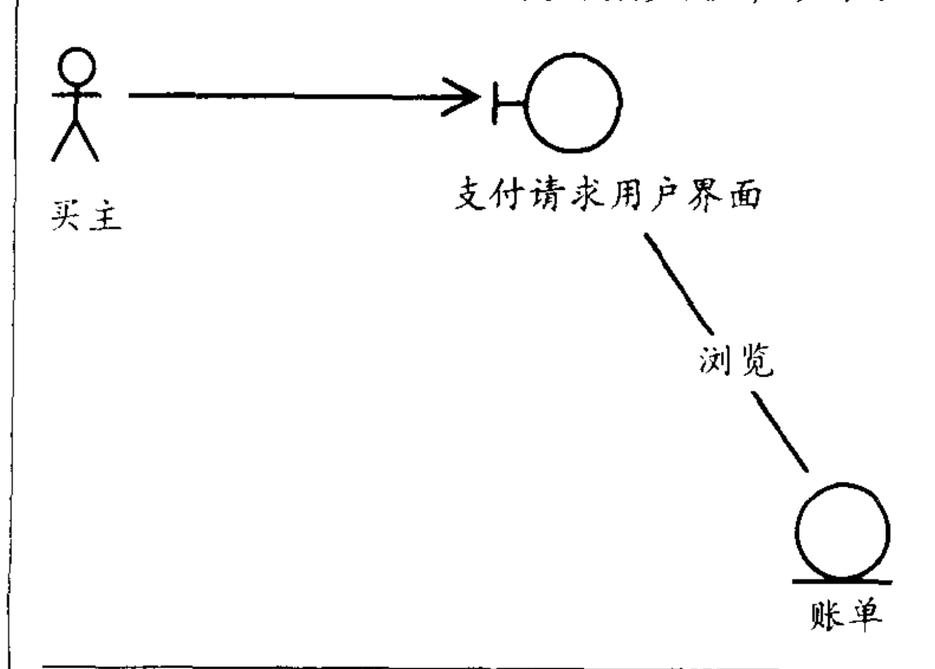


图8-7　"账单"实体类以及它与"支付请求用户界面"边界类的关系

8.4.2.3 控制类

控制类代表协调、排序、事务处理以及对其他对象的控制，经常用于封装与某个具体用况有关的控制。控制类还可用来表示复杂的派生与演算，如业务逻辑，它们与系统存储的任何具体的长效信息（即某个具体的实体类）都没有关系。

系统的动态特征通过控制类来建模，因为控制类处理和协调主要的动作和控制流，并将任务委派给其他对象（即边界对象和实体对象）。

注意，控制类并不封装与参与者交互有关的问题，也不封装与系统处理的长效、持久信息有关的问题，这些问题分别是由边界类和实体类封装的。而控制类对控制、协调、排序、事务处理、有时是涉及其他多个对象的复杂业务逻辑进行封装，因而能将它们与变化隔离开来。

举例："支付调度程序"控制类

为了精化前面介绍的例子，引入一个称为"支付调度程序"的控制类，它负责在"支付请求用户界面"边界类和"账单"实体类之间进行协调，如图8-8所示。

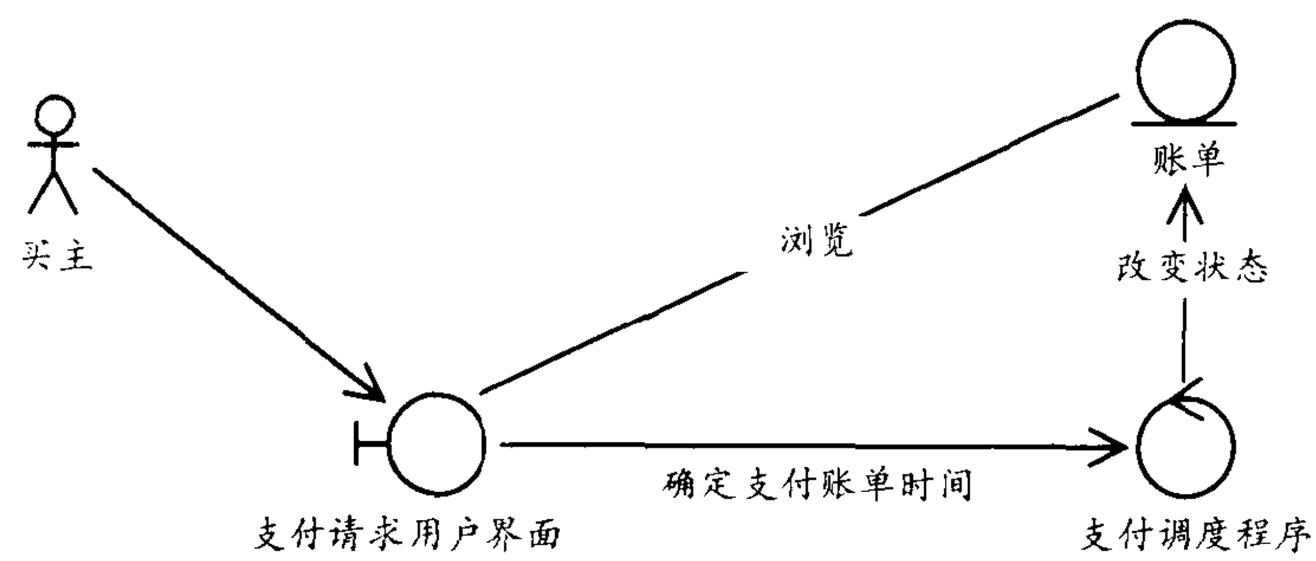


图8-8 引入"支付调度程序"控制类以及它与边界类、实体类的关系

"支付调度程序"接受支付请求（例如，支付账单请求）和支付账单日期。之后，在账单支付日期，"支付调度程序"通过请求在两个相应的账户之间转账来完成支付任务。

8.4.3 制品：用况实现—分析

*用况实现—分析*是分析模型内部的一种协作，它描述了如何根据分析类及其交互的分析对象来实现和执行一个具体的用况。于是，一个用况实现便可以很直接地跟踪到用况模型中一个具体用况（参见图8-9）。

用况实现包括事件流的文本描述、反映参与用况实现的分析类的类图以及按照分析对象的交互作用描述特定流实现或用况脚本的交互图（参见图8-10）。另外，因为这里的用况实现是按照分析类及其对象进行描述的，所以，它很自然地侧重于功能性需求。于是，就像分析类一样，可以将非功能性需求推迟到在后续的设计与实现活动中将它们标明为实现的具体需求时再进行处理。

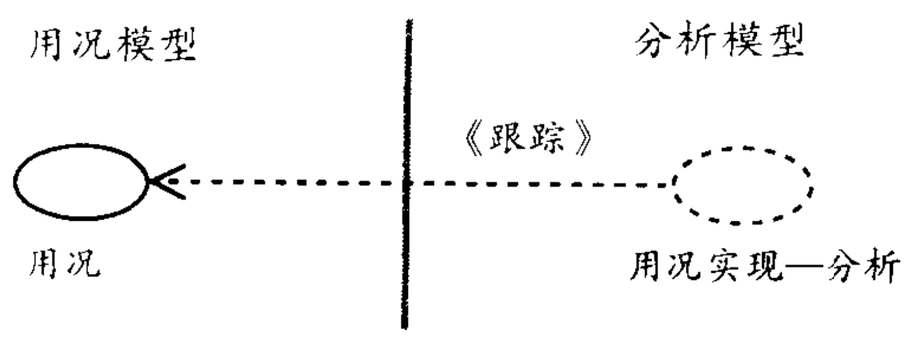


图8-9　分析模型中的用况实现—分析与用况模型中的用况之间存在可跟踪性

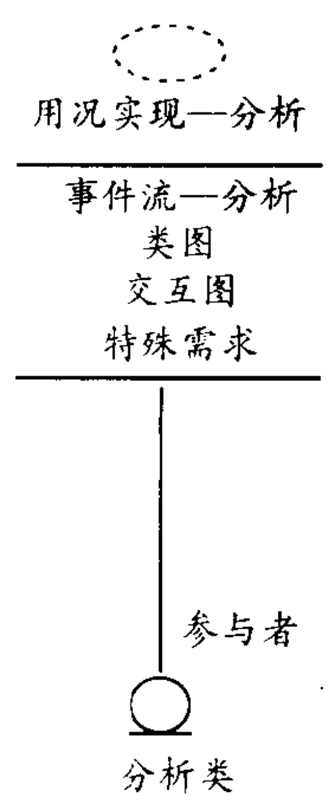


图8-10　用况实现—分析中的关键属性与关联

8.4.3.1　类图

一个分析类及其对象经常参与多个用况实现，而一个具体类的某些职责、属性及关联通常只与一个用况实现相关。因此，在分析期间，协调不同的用况实现可能具有的类及其对象上的所有需求是重要的。为了做到这一点，我们为用况实现附加上类图，以表明参与用况实现的类及其关系（参见图8-11）。

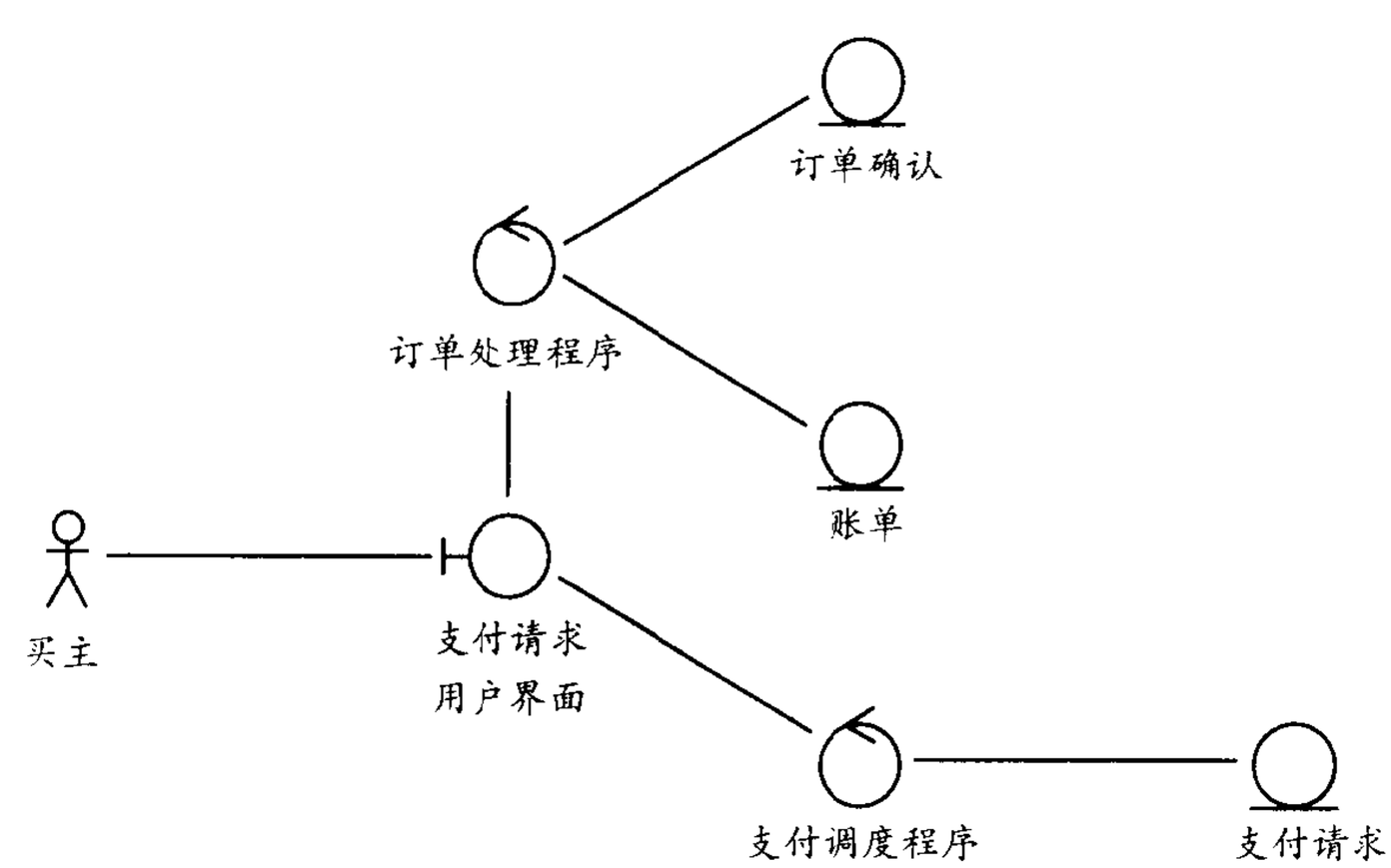


图8-11　“支付账单”用况实现的类图

8.4.3.2 交互图

当参与者向系统发送某种形式的消息而激活用况时，开始执行该用况中的动作序列。此处如果我们探究系统的“内部”实现，则边界对象将接收来自参与者的消息。然后边界对象向其他对象发送一个消息，并使有关对象与之交互从而实现该用况。在分析阶段，建议使用协作图来描述用况的实现，因为我们主要关注的是确定需求和对象的职责，而不是确定详细的按时间顺序的交互序列（这种情况应该用顺序图来表示）。

在使用协作图时，通过在对象之间建立链接并在其上附加消息来表明对象间的交互。消息名称应该能够反映出在与被引用对象交互时引用对象的意图。

举例：表明用况实现的协作图

图8-12所示是实现“支付账单”用况第一部分的协作图。

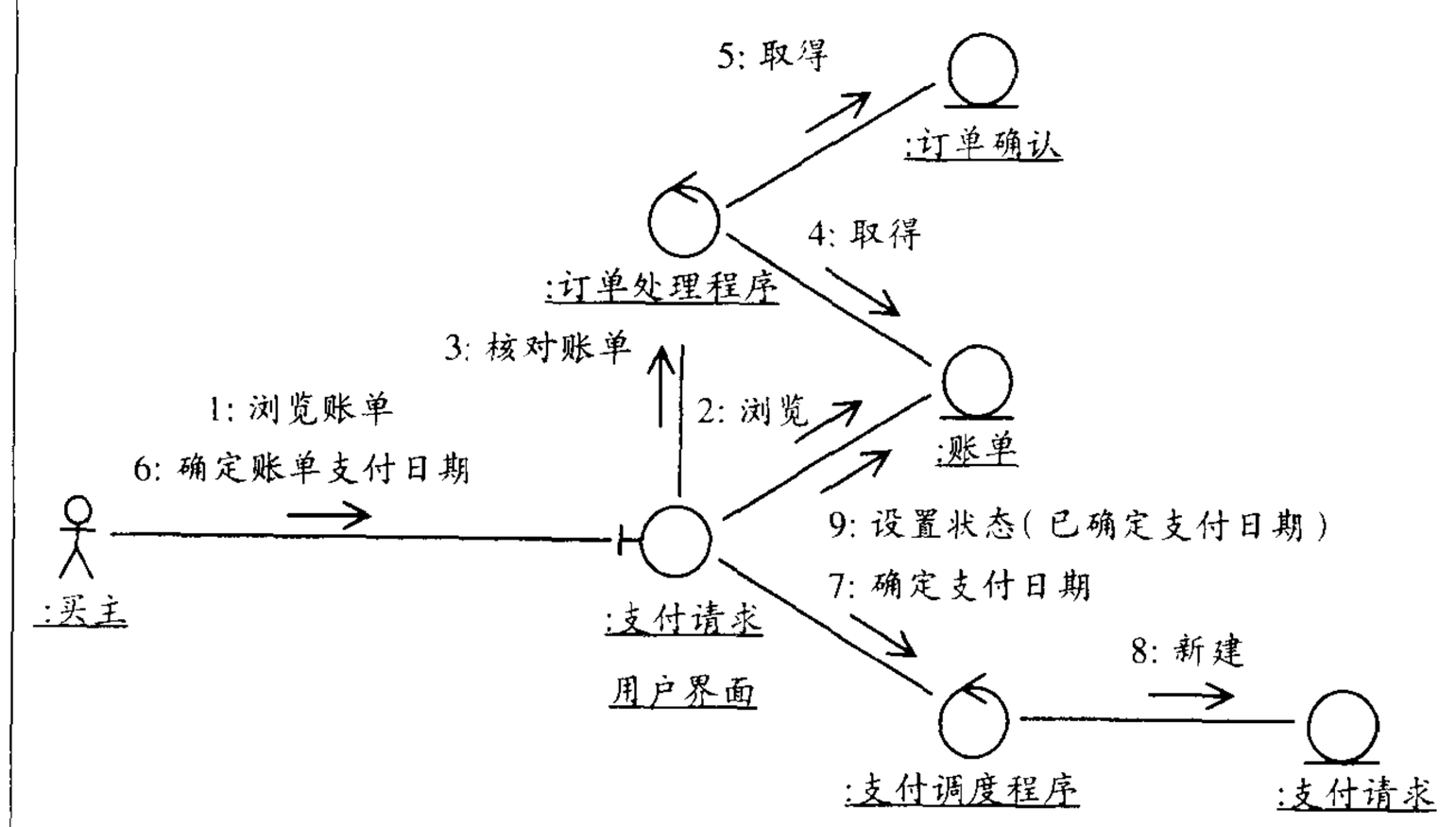


图8-12 实现“账单支付”用况的协作图

对于用况实现中分析对象的创建与终结，不同对象具有不同的生命周期（参见附录C）：

- 一个边界对象无须专用于一个用况实现，例如，如果它必须出现在窗口中并参与两个或多个用况实例时就是如此。但是，边界对象经常是在单个用况实现的内部创建和终结的。
- 一个实体对象通常不专用于一个用况实现。而且，一个实体对象的生命周期较长，在其终结前经常会参与多个用况的实现。
- 控制类通常对与具体用况有关的控制进行封装，这表明控制对象是在用况开始时创建，而在用况结束时终结。然而也有例外，如当一个控制对象参与多个用况实现，或当几个控制对象（属于不同控制类的）参与一个用况实现，以及当用况实现根本不需要任何控制对象时。

8.4.3.3 事件—分析流

描述用况实现的图（尤其是协作图）本身不易读懂，因此，补充一些解释性的文本可能很

有效。文本应该按照对象（尤其是控制对象）来写，正是它们之间的相互作用实现了用况。但是，文本不应该提及任何有关对象的属性、职责以及关联等内容，因为它们变化得很频繁以致不易维护

举例：用于说明协作图的事件—分析流

下面的文本描述补充说明了前面例子中的协作图（参见图8-12）：

买主通过“支付请求用户界面”浏览系统管理的账单，找出已收到的账单（1，2）。“支付请求用户界面”使用“订单处理程序”核查与“订单确认”有关的账单（3，4，5），然后向买主显示账单列表。核查的限定条件取决于买主组织建立的业务规则，其中可能包括对价格、发货日期及附有订单确认的账单内容的比较。“订单处理程序”对象运用这些业务规则来确定向“订单确认”提问哪些问题（见图中“取得”消息4，5）以及“账单”对象如何分析答案。“支付请求用户界面”对不可信的账单会以某种方式为买主做上标记，例如用不同的颜色来显示。

买主通过“支付请求用户界面”选择账单，并确定支付账单时间（6）。“支付请求用户界面”调用“支付调度程序”规划账单支付（7）事务。然后，“支付调度程序”创建一个支付请求（8）。“支付请求用户界面”接着将该账单的状态改为“已确定日期”（9）

在预定的日期，“支付调度程序”启动支付工作（图中未显示）。

将以上描述与第7章7.2.3.1节中所描述的用况事件流进行比较很有意义。第7章中所描述的是用况的外部可见行为，而此处的描述则侧重于说明系统如何根据相互协作的（逻辑）对象来实现用况。

8.4.3.4　特殊需求

特殊需求是对收集到的一个用况实现上的所有非功能性需求的文本说明。其中有些需求在需求工作流期间（第6章和第7章中介绍过）已经以某种形式被捕获，只需针对**用况实现—分析**将它们进行一定变化即可。但有些需求则是新的或随着分析工作的进行而派生出来的。

举例：一个用况实现的特殊需求

下面的例子是“支付账单”用况实现的特殊需求：

- 当买主要求查看所收到的账单时，在屏幕上显示账单所需要的时间不能超过0.5秒。
- 使用SET标准来进行账单支付。

8.4.4　制品：分析包

分析包提供了一种方法，以可管理分块的方式对分析模型的制品进行组织。分析包可以包括分析类、用况实现及其他分析包（递归地）。参见图8-13。

分析包应该具有强内聚性（即它们所包含的元素应紧密相关）与弱耦合性（即彼此间的依赖应尽可能小）。

此外，分析包还具有以下特点：

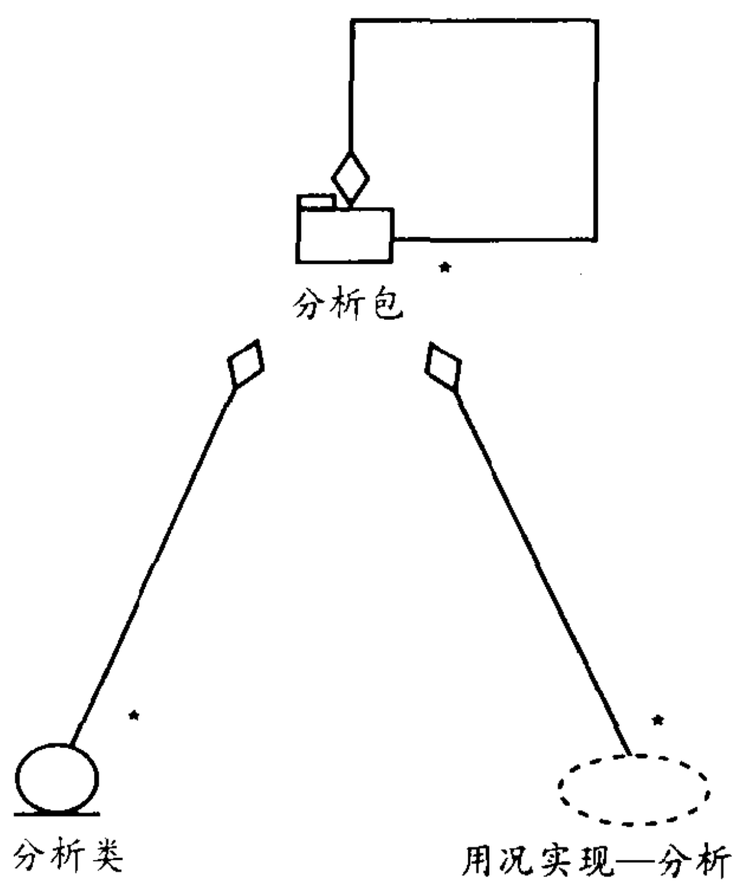


图8-13 分析包的内容

- 分析包可以表示对分析内容的分割。例如对一个大型系统，可以对某些分析包分别进行分析，这些包可能是由具有不同领域知识的不同开发人员并行开发的。
- 分析包应基于功能性需求与问题领域（即应用或业务）来创建，并且应能被具有该领域知识的人所理解。分析包不应该基于非功能性需求与方案领域来开发。
- 分析包很可能成为设计模型中两个高级应用层中的子系统，或在子系统中分布。有时候，一个分析包甚至可以表达设计模型中的整个最高层。

8.4.4.1 服务包

除了向参与者提供用况之外，每个系统还向其客户提供一组服务。客户抽取适当的服务组合以便向他或她的用户提供必需的用况来完成业务：

- 一个用况确定了一个动作序列：由参与者启动一个线程，接着进行参与者与系统的交互，当参与者获得返回值后，该线程完成并终止。通常，用况不是孤立存在的。例如，“给买主开单”用况假定另一个用况已创建了一个账单，而且买主的地址和其他客户数据都是可以存取的。
- 一个服务表示用于多个用况的、功能上紧密相关的动作集合（即一个功能包）。系统的客户通常购买不同的服务组合以便向其用户提供必需的用况。系统要么提供完整的服务，要么根本不提供任何服务，从这种意义上来讲服务是不可分割的。
- 用况针对用户，而服务针对客户。用况与服务是交叉的，即一个用况需要来自多个服务的动作。

在统一过程中，服务的概念是由服务包支持的。服务包在按照系统提供的服务而组织的分析包（容器）层次结构中处于较低层。关于服务包需要注意以下几点：

- 服务包包含了一组功能上相关的类。
- 服务包是不可分割的。每个用户要么得到服务包中的所有类，要么根本得不到其中的任何一个类。

- 在实现用况时，可能会有一个或多个服务包参与其实现。而且，一个具体的服务包通常会参与几个不同的用况实现。
- 一个服务包经常很少与其他服务包相关。
- 一个服务包往往只与一个或少数几个参与者有关。
- 一个服务包所定义的功能，在设计与实现时可作为一个单独交付的单元来处理。因此，一个服务包可以表示为该系统的某些“附加”功能。如果去掉一个服务包，那么，需要该服务包参与实现的用况也将失效。
- 服务包可能是互相排斥的，或者它们可以表示同一种服务的不同方面或不同形式。例如，“英式英语拼写检查”与“美式英语拼写检查”可能是同一系统提供的两种不同的服务包。
- 服务包构成了后续的设计和实现活动的基本输入，并将按照服务子系统来辅助构造设计和实现模型。特别是，服务子系统对于将系统分解为二进制和可执行的构件具有深远的影响。

通过按系统提供的服务来构造系统，就为个别服务的变化做好了准备，因为这种变化可能只局限于相应的服务包。这样产生的就是一个对于变化具有柔性的健壮系统。

在8.6.1.1.2节中给出了一种确定服务包的方法，同时列举了几个服务包的例子。

注意，组织分析模型中制品的一般方法仍然是使用前面章节中所介绍的普通分析包。但这里我们引入《服务包》构造型以便能够明确地标明代表服务的那些包。在大型系统中（具有很多包），尤为重要的是要能够以一种简易的方式来区分不同类型的包。这也是对如何在UML中有效地使用和利用构造型的例证。

8.4.4.1.1 服务包是可重用的

前一节曾讨论过，服务包具有很多好的特点，如内聚性（参见附录C）、不可分割性、弱耦合性、可单独交付等。这使得大多数服务包成为一个系统内和跨越（或多或少）相关的几个系统进行重用的首选。更确切地说，以一个或多个实体类（参见8.4.2.2节）为中心的提供服务的服务包，在支持相同的业务或领域的不同系统中很可能得到重用。这是因为实体类是从业务实体类或领域类中得到的，从而使实体类和相关的服务能够在整个业务或领域内以及大多数支持它的系统中得到重用，而不只是针对一个具体的系统。如果一个服务包可用于几个不同的用况实现，那么服务包和服务对用况来说就是正交的。当服务包位于具有公用和共享功能（见8.6.1.1.3节）的通用层中尤其如此。那么，这样的服务包可能会在系统的多个不同应用（配置）中得到重用。这里系统的每个应用均提供多个用况，每个用况的实现都需要该服务包。服务包可以跟踪到设计阶段的服务子系统（参见9.3.4.1），并进而跟踪到实现阶段的构件（参见10.3.2节）。基于和服务包同样的原因，这些构件也是可重用的。因此，服务包便成为分析期间针对重用进行构造的主要工具，它还影响到系统的设计与实现，并最终获得可重用的构件。

8.4.5 制品：构架描述（分析模型的视图）

构架描述包括**分析模型的构架视图**（参见附录C），它描述了对构架重要的制品（参见

图8-14）。

图8-14 包含分析模型的构架视图的构架描述

通常认为分析模型中的下列制品对构架是重要的：

- 从分析模型到分析包及其依赖关系的分解。在设计和实现期间，这种分解往往会影响到处于高层的子系统，因此，它对整个构架而言通常是很重要的。
- 关键的分析类（如封装问题领域中重要现象的实体类）；封装重要通信接口和用户界面机制的边界类；封装具有较大覆盖范围的重要事件序列的控制类（即那些协调重要用况实现的序列）；以及通用的、主要的、与其他分析类关系紧密的分析类。从对构架重要性的角度，通常只考虑抽象类就够了，不必再考虑其子类。
- 实现某些重要的关键性功能的用况实现；涉及多个分析类因而有较大的覆盖范围、可能跨越几个分析包的用况实现；或侧重于需要在软件生命周期中尽早开发的某个重要用况并因此应该在**用况模型的构架视图**（参见附录C）中建立的用况实现。

8.5 工作人员

8.5.1 工作人员：构架设计师

在分析工作流期间，构架设计师对分析模型的完整性负责，确保分析模型作为一个整体的正确性、一致性和易读性（参见图8-15）。对于大型、复杂的系统，经过若干次迭代之后，这种职责可能要求更多的维护工作，并且有关的工作可能变成相当常规性的。在这种情况下，构架设计师可以把这项工作委托给另外的工作人员，很可能是一个“较高层”的构件工程师（参见8.5.3节）。然而，构架设计师仍要对构架的重要部分（构架描述）负责。其他的工作人员要对分析模型中的高层包负责，它们需要与构架描述相适应。

这种分析模型当且仅当实现了用况模型中描述的功能时才是正确的。

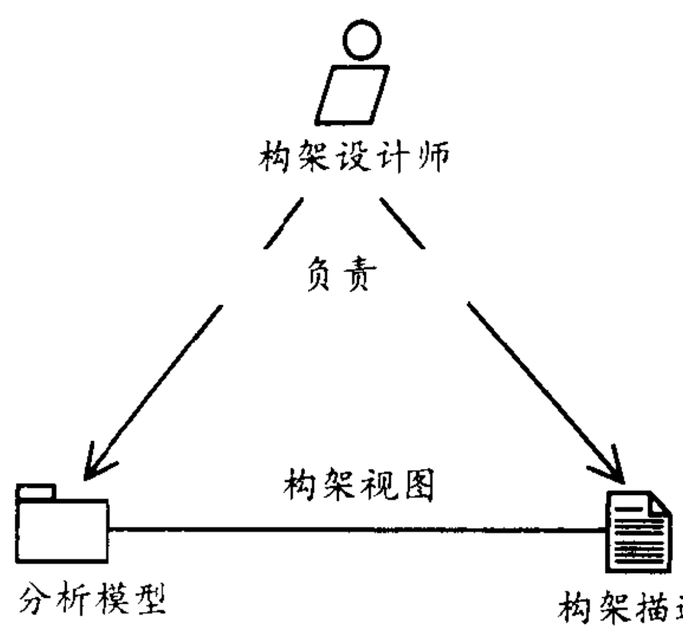


图8-15 分析阶段中构架设计师的职责

构架设计师还要对分析模型的构架负责，也就是说，要对按模型的构架视图中描述的、对系统构架重要部分的存在负责。因为该视图是系统的构架描述的一部分。

注意，构架设计师并不对分析模型中各种制品的持续开发和维护负责。相反，这些是相应的用况工程师和构件工程师的职责（参见8.5.2节和8.5.3节）。

8.5.2 工作人员：用况工程师

用况工程师负责一个或几个用况实现的完整性，确保它们实现了各自的需求（参见图8-16）。一个用况实现必须并且只需正确实现用况模型中相应用况的行为，包括确保描述用况实现的文本说明和图的易读性，并能满足它们的用途。

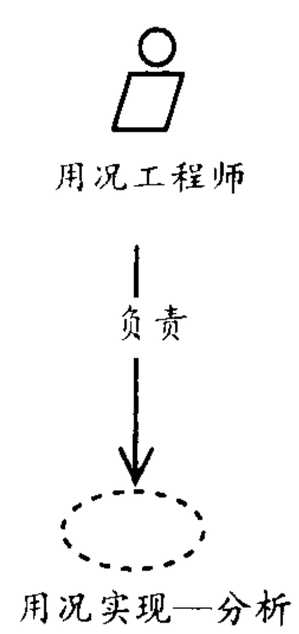


图8-16 分析阶段中用况工程师的职责

注意，用况工程师不负责用况实现中所使用的分析类和关系。这些是相应的构件工程师的职责（参见8.5.3节）。

在下一章我们会看到，用况工程师还要对用况实现的设计负责。于是，用况工程师既要对用况的分析负责，又要对用况的设计负责，这有助于从分析无缝地过渡到设计。

8.5.3 工作人员：构件工程师

构件工程师定义并维护一个或多个分析类的职责、属性、关系和特殊需求，确保每个分析

类能实现来自它所参与的各个用况实现的需求（参见图8-17）。

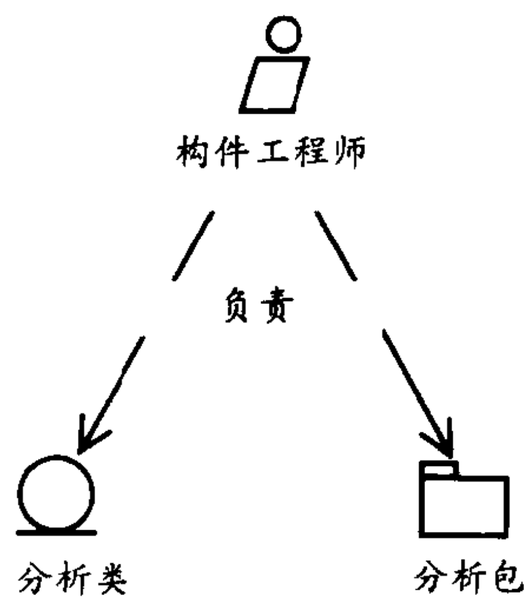


图8-17 分析阶段中构件工程师的职责

构件工程师还要维护一个或多个分析包的完整性，包括确保它们内容（例如类和类之间的关系）的正确性以及对其他分析包依赖的正确性和最小性。

通常让一个负责分析包的构件工程师同时负责该分析包所包含的分析类。而且，如果在分析包和相应的设计子系统（参见8.4.4节）之间有直接映射关系（简明跟踪性），则构件工程师还应当对这些子系统负责，并在设计和实现该分析包时运用分析阶段中所获得的知识。如果没有这种直接映射关系，则需要另外的构件工程师参于设计和实现该分析包。

8.6 工作流

在本章开始，我们从静态角度描述了分析工作。现在，我们用活动图来探究其动态行为（参见图8-18）。

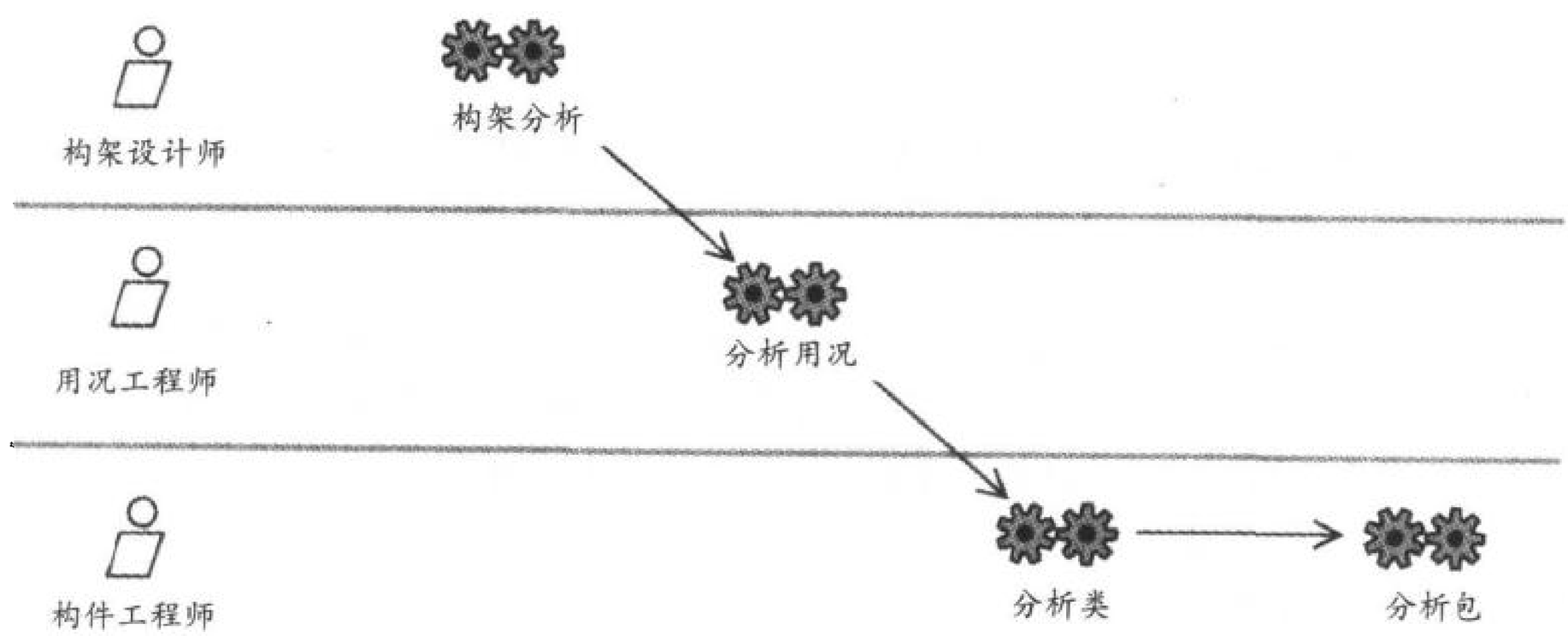


图8-18 分析阶段中的工作流，包括参与的工作人员和他们的活动

分析模型（如在本章开始定义的分析模型）的建立是由构架设计师通过确定主要分析包、显见的实体类和公用需求启动的。而后，用况工程师通过规定参与分析类的每个类的行为需求

来实现每个用况。然后这些需求又由构件工程师通过为每个类创建一致的职责、属性和类之间的关系确定下来并集成到每个类中。在分析期间，随着分析模型的进化，构架设计师不断确定新的分析包、类和公用需求，而构件工程师负责对单个分析包不断进行精化和维护。

8.6.1 活动：构架分析

构架分析的目的是通过确定分析包、显见的分析类和公用的特殊需求来概述分析模型和构架（参见图8-19）。

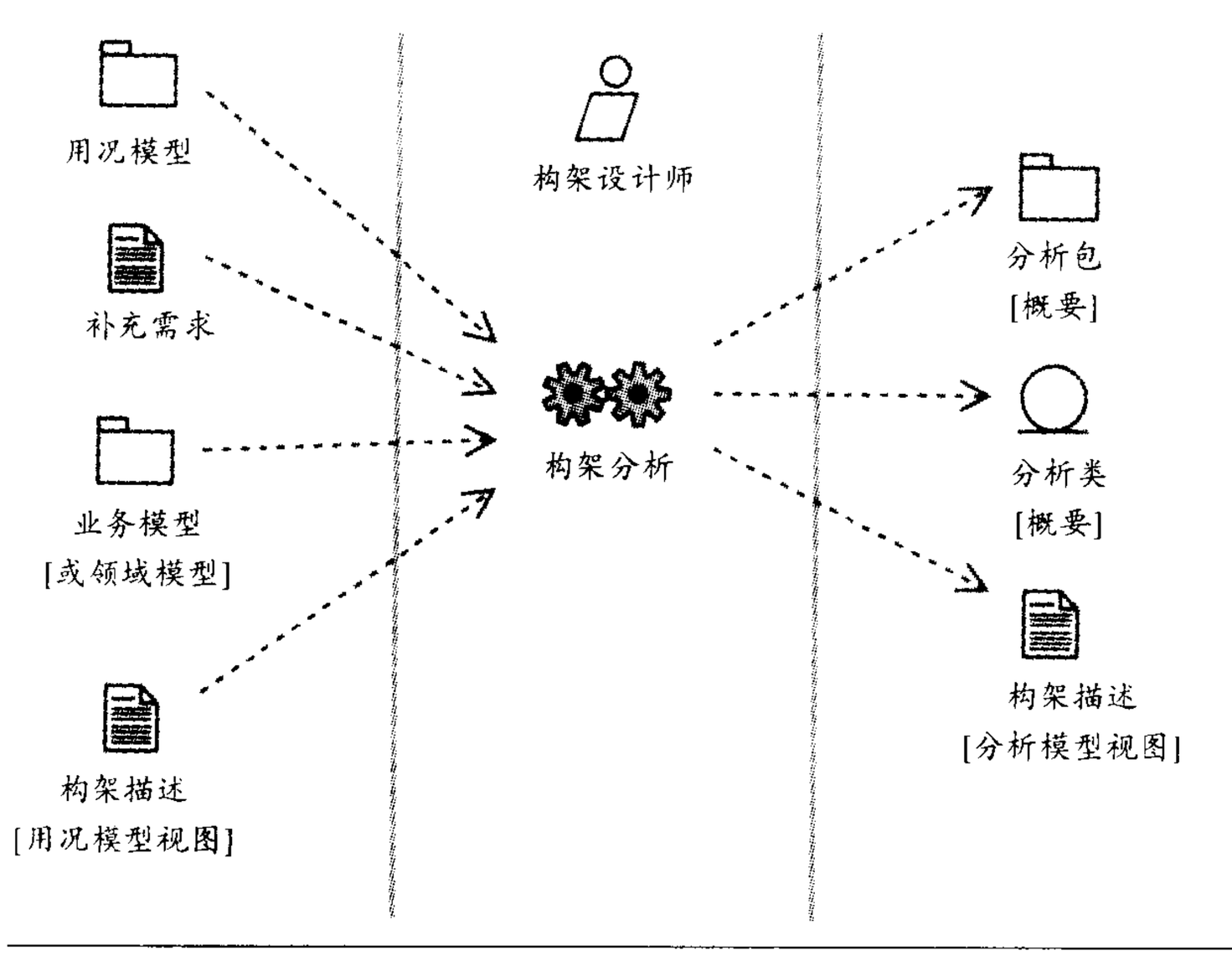


图8-19 构架分析的输入和结果

8.6.1.1 确定分析包

分析包提供了一种方式，用于将分析模型组织成更小、更可管理的块。它们或者开始就被确定为一种划分分析任务的方法，或者随着分析模型的进化和“发展成为”一个需要分解的大型结构时再被确定。

最初确定分析包自然是基于功能需求和问题领域，即所考虑的应用或业务。因为要将功能需求捕获为用况，所以一种确定分析包的简单方式是将一些用况的主要部分分配给一个具体包，然后在该包中实现相应的功能。以下是将用况恰当地分配给具体包的准则：

- 需要支持一个具体业务过程的用况。
- 需要支持系统的一个具体参与者的用况。
- 通过泛化和扩展关系建立用况的关联。在用况彼此间具有特化或相互“扩展”关系时，用况集合具有强内聚性。

这种包会将变化分别局限于一个业务过程、一个参与者行为或一个紧密相关的用况集合中。这种方法在最初有助于将用况分配到包中。用况通常并不局限于一个包内，而是跨越几个包。

因此，随着分析工作的进行，当用况实现为类（可能存在于不同的包中）之间协作时，就会得到一个更加精化的包结构。

举例：确定分析包

这个例子阐明Interbank Software如何从用况模型中确定分析包。用况“支付账单”、“发送提醒通知单”和“给买主开单”都参与了同一个“销售：从订单到交货”的业务过程。因此，可以将它们放到一个分析包中。

但是Interbank Software应能针对不同客户的需求提供不同的系统。一些客户只作为买主来使用系统，另一些客户只作为卖主，还有一些客户既是买主又是卖主。因此采取将卖主需要的用况实现与买主需要的用况实现分开的措施。这里最初将业务过程“销售：从订单到交货”作为一个分析包的设想进行调整来满足客户的需要。于是根据客户的需要可以向他们分别交付两个分析包：“买主账单管理”和“卖主账单管理”（参见图8-20）。

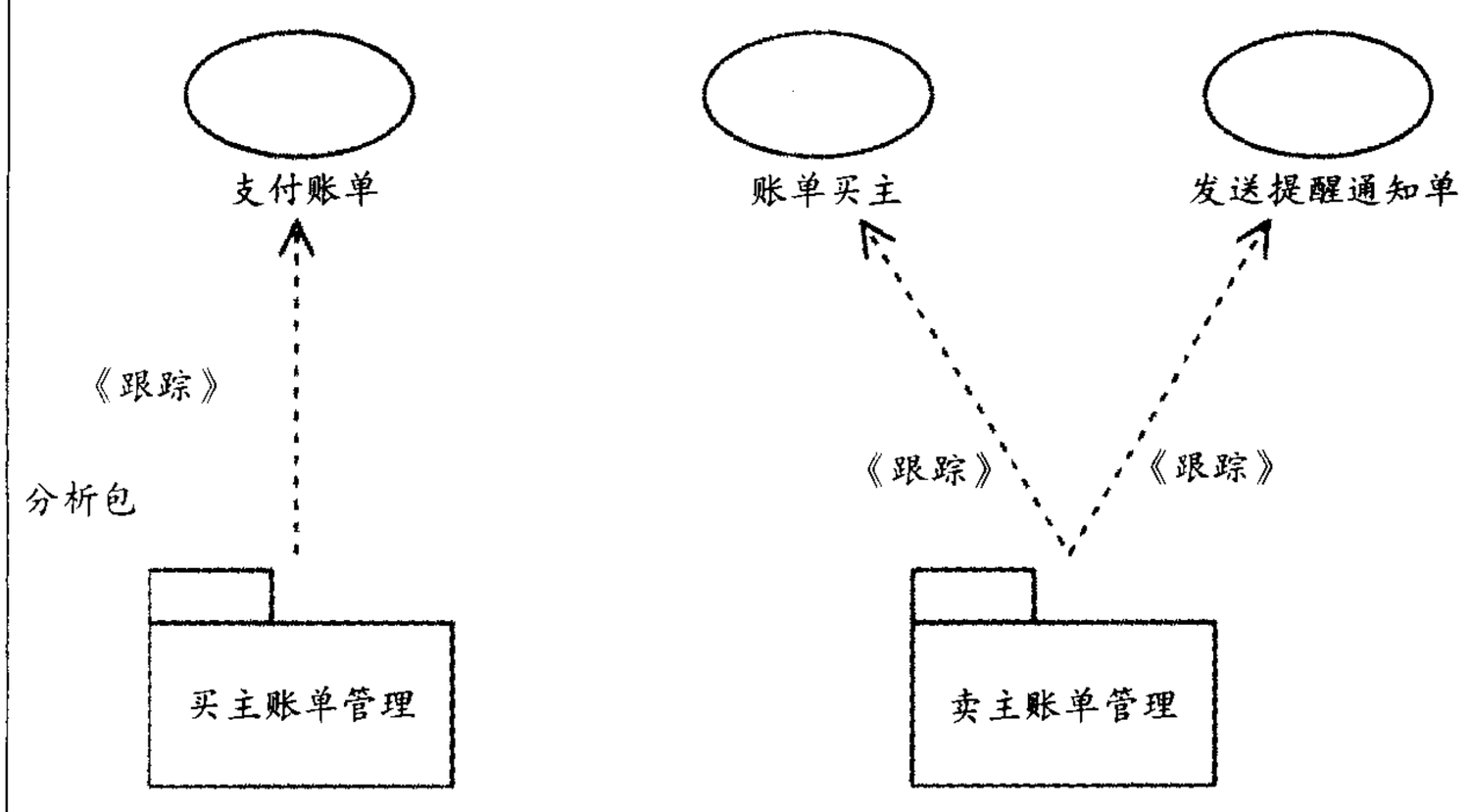


图8-20 从用况中确定分析包

注意，其他用况也支持业务过程“销售：从订单到交货”，但为了简化例子，此处忽略。

8.6.1.1.1 处理分析包之间的共性

正如前面章节确定的，在包间经常会发现共性的内容。例如，当两个或更多的分析包需要共享同一个分析类时就是如此。处理这种情况的恰当方法是抽取出共享类，把它作为一个单独的分析包，或只是放到其他包的外面，然后让其他包依赖于这个通用的包或类。

这种表示共性的共享类很像是实体类，且可跟踪到领域或业务的实体类。因此，如果领域或业务的实体类是共享的，就值得对它们进行研究，而且通用分析包是在分析阶段中才确定的。

举例：确定通用分析包

这个例子阐明Interbank Software如何从领域模型中确定通用分析包。每个领域类“银行储户”和“账户”都表示现实世界中重要而复杂的实体。Interbank认识到这些类需要复杂信息系统的支持，而且可以与其他更为具体的分析包共享。因而Interbank Software为每个类创

建单独的“账户管理”包和“银行储户管理”包（参见图8-21）。

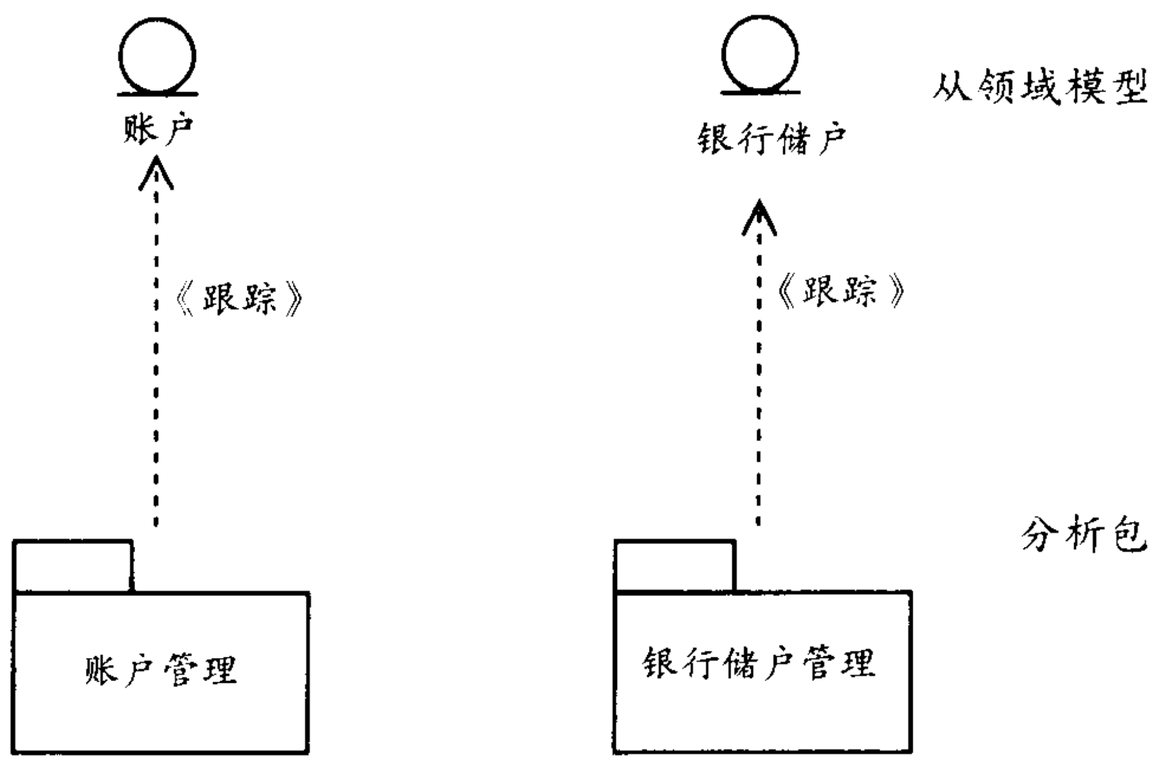


图8-21 从领域类中确定通用分析包

注意，“账户管理”包和“银行储户管理”包可能会包含很多分析类，例如分别与账户管理和银行储户管理有关的控制类和边界类。所以，这些包不可能只包含一个或几个可跟踪到相应领域类的实体类。

8.6.1.1.2 确定服务包

恰当的服务包通常是在分析工作的后期、当对功能需求有了清晰的理解并且大多数分析类已经存在的时候确定的。同一服务包中的分析类都是用于相同的服务。

通常按以下步骤来确定服务包：

- 为每个可选的服务确定一个服务包。这种服务包是一个定制的单元。

举例：可选的服务包

大部分使用Interbank系统的卖主都希望系统具有发送提醒通知单的服务。这种服务在（可选的）用况“发送提醒通知单”中描述。有些卖主希望一旦存在过期账单就自动发送提醒通知单，而另一些卖主愿意有过期账单时先得到通知，而后再决定是否发送提醒通知单。这种可变性可以表示为两种可选的且彼此只能专用的服务包：“自动发送提醒通知”用于系统自动发出提醒通知单，而“人工发送提醒通知”首先通知卖主，而后由他决定是否与买主联系；参见图8-22。当卖主不需要提醒通知支持时，交付的系统中就不要附带任何服务包。这些服务包包含在“卖主账单管理”包中。

- 为每个可能成为可选的服务确定一个服务包，即使每个客户都一直希望得到该服务。因为服务包中包含功能相关的类，所以这将得到一种将大部分变化局限于个别服务包中的包结构。这种准则也可以描述为：为由功能相关的类所提供的每个服务确定一个服务包。例如，当出现下述情况时，类A和类B功能上是相关的：
- A中的变化很可能要求B中有所变化。
- 删除A使B成为多余的。

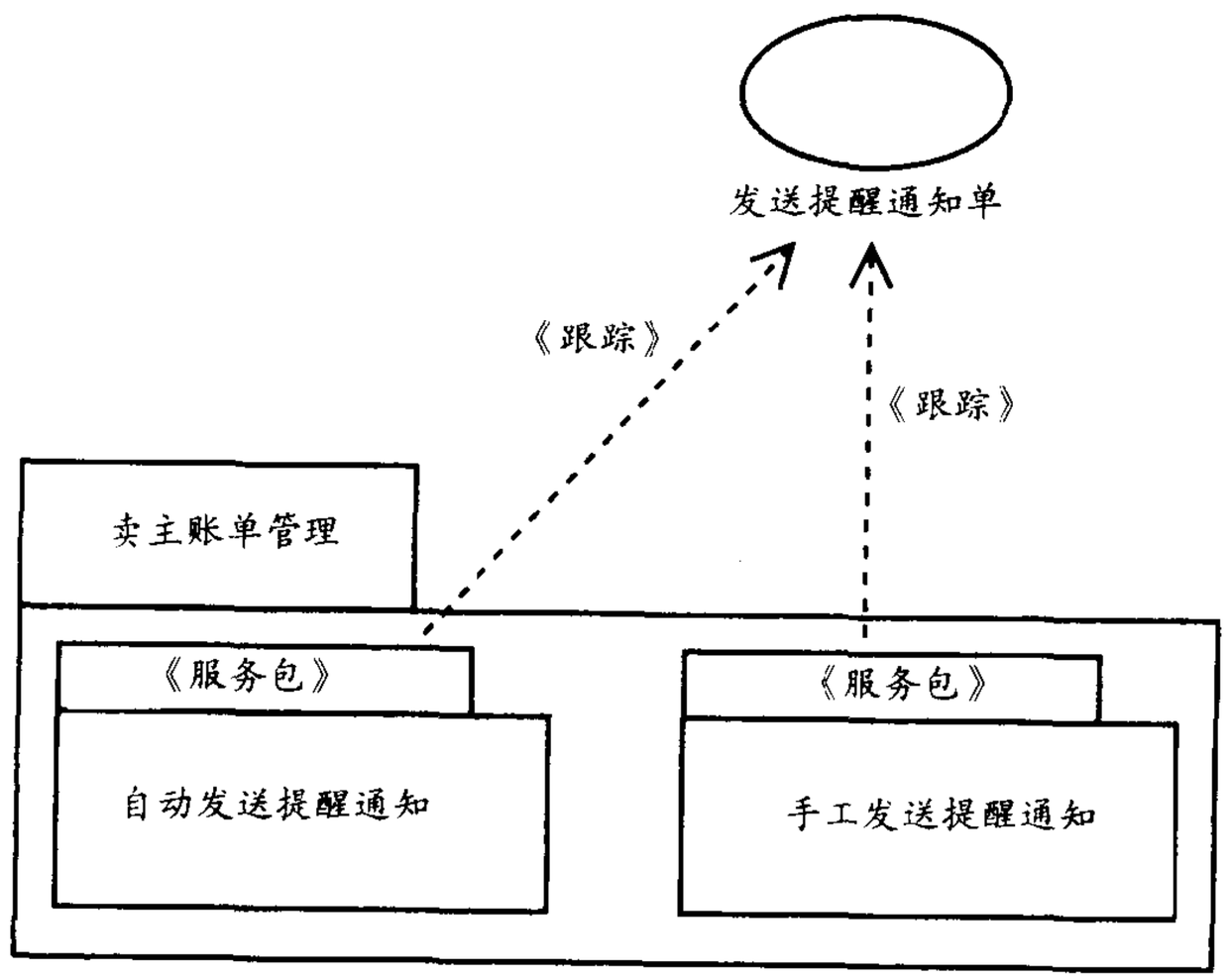


图8-22 局限于“卖主账单管理”包内的“自动发送提醒通知”和“人工发送提醒通知”服务包

- A的对象可能通过几个不同的消息与B的对象频繁地进行交互。

举例：确定封装功能相关的类的服务包

“账户管理”包包括一个称为“账户”的通用服务包，用于当需要转账和提取事务历史等活动时访问账户。这个包还包括另一个称为“风险管理”的服务包，用于估计与某个特定账户相关的风险。这些不同的服务包是公用的，由几个不同的用况实现使用。参见图8-23。

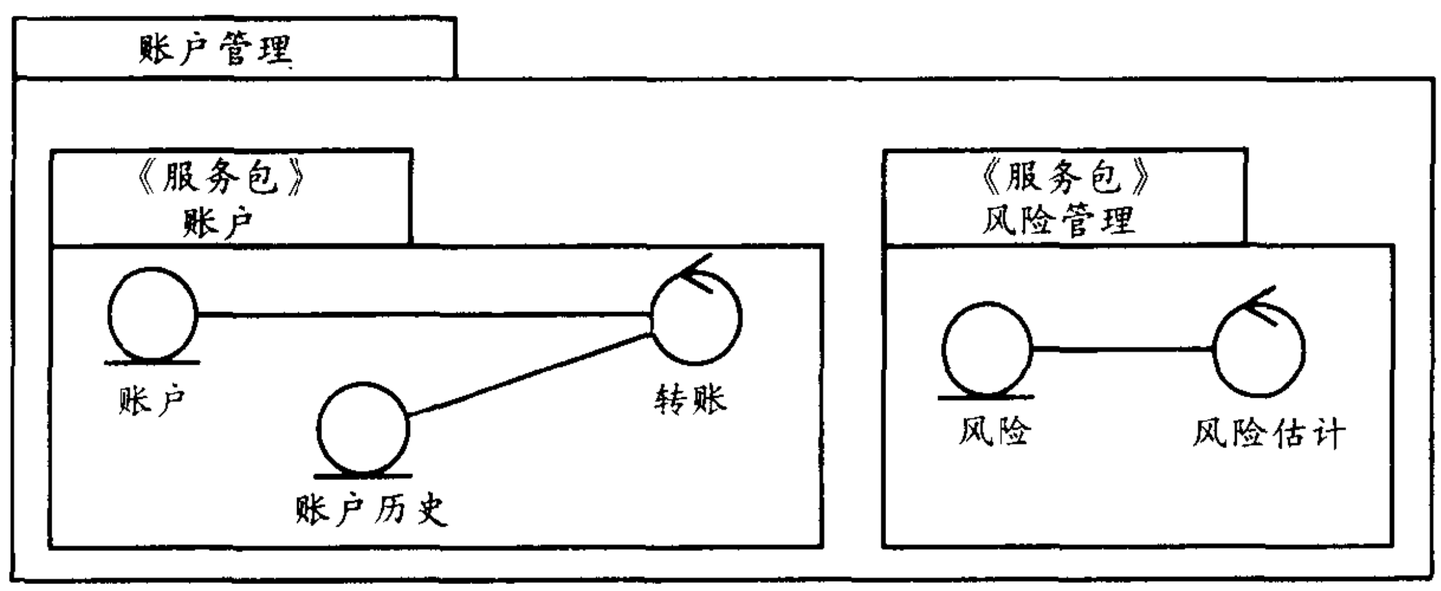


图8-23 “账户”服务包和“风险管理”服务包，每个服务包均封装了功能相关的类

8.6.1.1.3 定义分析包间的依赖

如果分析包的内容间彼此关联，就应该定义分析包间的依赖。依赖的方向应该与关系的（导航）方向相同。

我们的目标是确定相对独立和低耦合度、但具有高内聚性的包。高内聚、低耦合使包更易于维护，这是因为改变一个包中的某些类将主要影响该包本身的类。因此，减少不同包中的类

之间的关系的数目是明智之举，因为这样降低了包间的依赖。

为了阐明这种依赖，对分析模型划分层次是有益的，可以将专用应用包放在一个较高层，而将多个通用包放在一个较低层。这样划分就使专用功能和通用功能区别得更加清楚。

举例：分析包间的依赖和层

“账户管理”包包含几个由其他包中的类所使用的类，如“账户”。“买主账单管理”和“卖主账单管理”包中的类都可使用“账户”类。因此，这些包依赖于“账户管理”包（参见图8-24）。

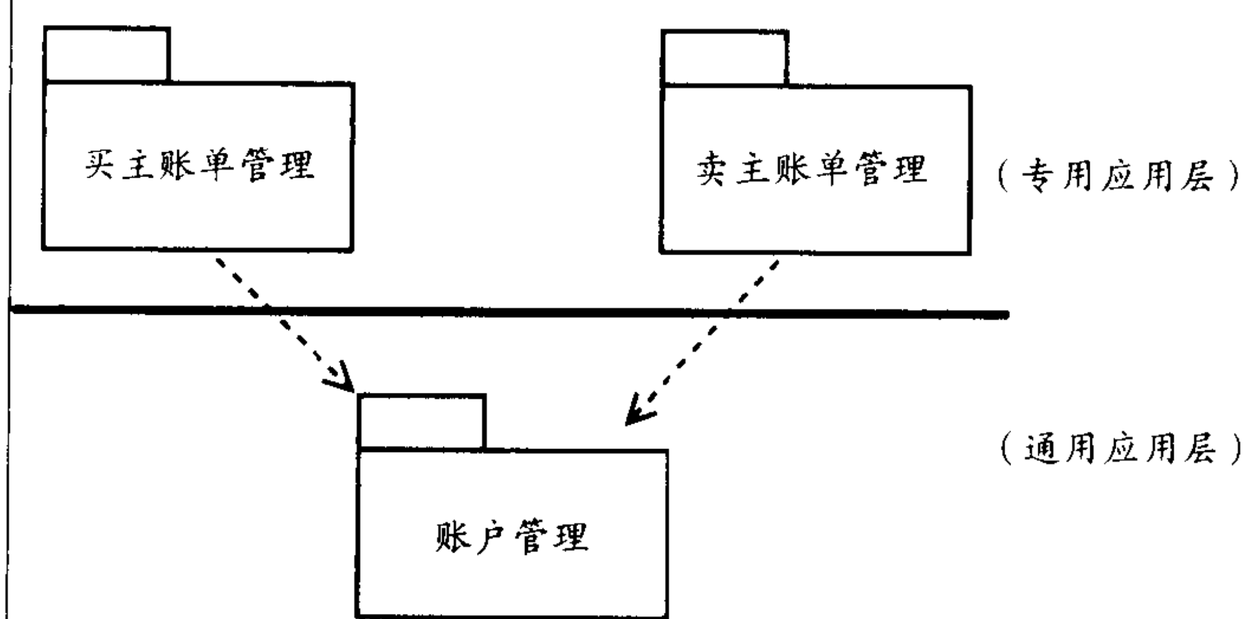


图8-24　分析包间的依赖和层次

一般来说，在设计和实现期间，我们将进一步精化这些层次，并通常在考虑实现环境和非功能性需求时还要增加更多较低级的层次。

8.6.1.2　确定显见的实体类

通常恰当的方法是，基于在需求捕获阶段定义的领域类和业务实体，对最重要的（10到20个）实体类提出初步建议。然而，大部分类是（在用况分析活动中，参见8.6.2节）创建用况实现时定义的。因此，应该注意不要在这个阶段定义过多的类，以至于陷入过多的细节。获得一个对构架重要的类的初步框架就足够了（参见8.4.5节）。否则，当后面使用用况来确定真正需要的实体类（即那些参与用况实现的类）时，很可能要重做大部分工作。不参与用况实现的实体类是多余的。

领域模型中领域类之间（或业务模型中业务实体之间）的聚合和关联可用于确定相应实体类之间的暂时的关联集合。

举例：由领域类确定实体类

账单是第6章提到的一个领域类。我们要用这个领域类来设计其中一个最初的实体类。作为起点，我们可以为“账单”类设计相同的属性：数量（amount），提交日期（data of submission），以及支付的最后期限（last date of payment）。我们还可以根据领域模型定义实体类间的关联，如“订单”类和“账单”类间的可支付关联。

8.6.1.3　确定公用的特殊需求

特殊需求是在分析过程中出现的需求，捕获它以便在后续的设计和实现活动中进行适当处

理是十分重要的。例如，对以下各方面的限制或约束就是公用的特殊需求的实例：

- 持久性
- 分布与并发
- 安全性
- 容错性
- 事务管理

构架设计师负责确定公用的特殊需求，以便开发人员能够引用它们作为个别的用况实现和分析类的特殊需求。在某些情况下，特殊需求不能预先确定，而只能在对用况实现和分析类进行探究时才能确定。还应注意，常常有对一个类或用况实现确定几个不同的特殊需求的情况。

为进一步支持后续的设计和实现，应该确定每个公用的特殊需求的关键特征。

举例：确定特殊需求的关键特征

一个持久性需求具有以下特征：

- 大小范围（*size range*）：保持持久性的对象的大小范围。
- 容量（*volume*）：保持持久性的对象的数目。
- 持久性时间段（*persistency period*）：对象一般需要保持持久性的时间段。
- 更新频率（*update frequency*）：对象的更新频率。
- 可靠性（*reliability*）：诸如对象在硬件或软件崩溃时是否能保存下来的可靠性问题。

因此，每个特殊需求的特征将针对引用特殊需求的每个类和用况实现来加以限定。

8.6.2 活动：分析用况

分析一个用况的目的在于（参见图8-25）：

- 确定分析类，其对象要执行用况的事件流。
- 将用况的行为分配给相交互的分析对象。
- 捕获用况实现中的特定需求。

用况分析的另一术语是*用况精化*（*use-case refinement*），我们将每个用况精化为分析类的协作。

8.6.2.1 确定分析类

在这一步，我们确定控制类、实体类和边界类，它们对于实现用况和表示它们的名称、职责、属性和关系是必需的。

需求中描述的用况对于确定分析类有时还不够详细。有关系统内部的信息在需求捕获期间一般没有多大意义，因而在此省略。因此，为了确定分析类，你可能必须就系统内部来精化用况说明。这种精化应该在事件的文本流中被捕获，用以作为用况实现的分析描述。

以下是确定分析类的一般性准则：

- 通过详细研究用况说明和已有的领域模型来确定实体类，接着考虑用况实现中需要涉及和处理的信息。但应该注意，那些最好能捕获为属性的信息（参见8.6.3.2节），那些最好能

归为边界类或控制类的信息，或者用况实现中并不需要的信息，都不应该作为实体类来建模。

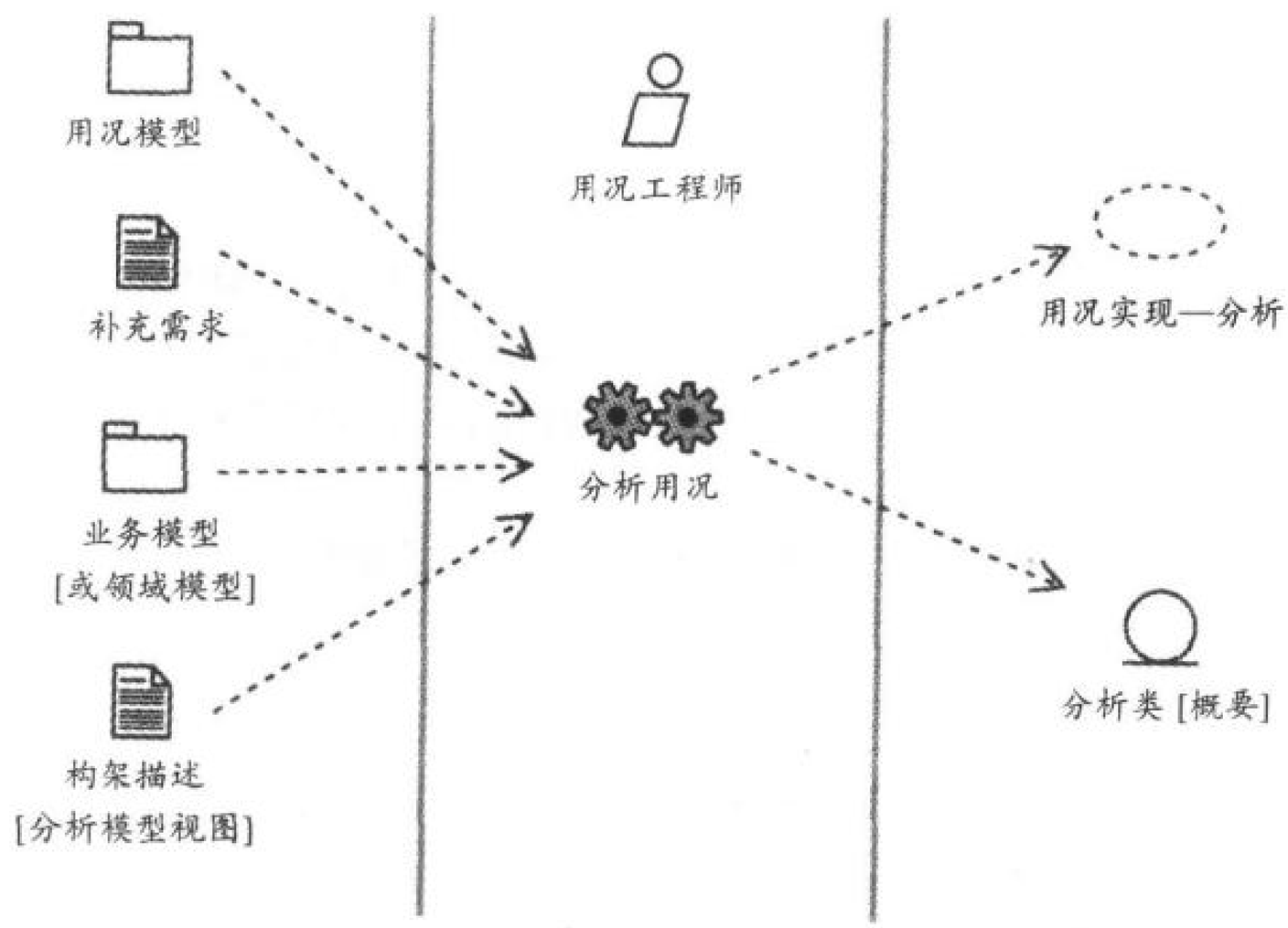


图8-25 分析用况的输入和结果

- 为每个由人充当的参与者确定一个主要边界类，并使这个类代表与参与者相交互的用户界面中的主窗口。如果参与者已经与某个边界类进行交互，就应该考虑重用那个类以使用户界面（参见附录C）具有良好的可用性并尽量减少每个参与者需要与之进行交互的主窗口数。而且通常认为这些主要边界类是更为基本的边界类的集合。
- 为每个初期建立的实体类确定一个基本边界类。这些类代表（人）参与者在执行用况期间相交互的用户界面中的逻辑对象。然后按照各种可用性准则对这些基本边界类进行精化，以便能有助于创建一个“良好的”用户界面。
- 为每个由外部系统充当的参与者定义一个主要边界类，使其代表通信接口。注意，系统的参与者可能是与当前系统进行交互的任何软件或硬件单元，如打印机、终端、报警设备、传感器等。如果将系统通信分为几个协议层，那么该分析模型可能有必要对其中的某些层加以区别。此时，应该为每个需要关注的层确定单独的边界类。
- 确定一个控制类，负责处理用况实现中的控制和协调关系，然后按照用况实现的需求对该控制类进行精化。例如，在某些情况下，控制最好能封装在边界类中，尤其是当参与者处理大部分控制时最好采用这种方法。这时并不需要控制类。在其他情况下，如果控制很复杂，最好将它封装在两个或多个控制类中。这时，需要对控制类进行划分。

当执行到这一步时，当然应该对分析模型中业已存在的分析类加以考虑。有些分析类很可能在当前的用况实现中得到重用，并且此时几乎同时实现了几个用况，这样将更易于确定参与多个用况实现的分析类。

将参与用况实现的分析类收集到一个类图中，并使用这个类图表明用况实现中所用到的

关系。

8.6.2.2 描述分析对象间的交互

当有了一个实现用况所需要的分析类的框架之后，就需要描述对应的分析对象间如何进行交互。这可以通过使用协作图来实现，协作图中包含了参与的参与者实例、分析对象以及它们的链接。如果该用况具有不同的或独特的流或子流，通常值得为每个流创建一幅协作图。这会使得用况实现更为清晰，而且也能够提取出表示通用和可重用的交互关系的协作图。

创建协作图是从用况流的初期开始的，然后一次一步地通过用况流并决定实现该步需要哪些分析对象和参与者实例进行交互。通常可以很自然地确定对象在用况实现的交互序列中所处的位置。对于协作图应注意以下几点：

- 需要通过从一个参与者实例向一个边界对象发送一个消息来引入用况。
- 每个在先前步骤中确定的分析类应至少有一个分析对象参与某个协作图。否则，由于该分析类不参加任何用况实现，因此它就是多余的。
- 不要将消息指定给操作，因为并没有为分析类确定操作。消息应该表示出引用对象在与被引用对象交互时的意图。这种“意图”是接收对象的职责的根据，甚至可能成为该职责的名称。
- 图中的链接一般应是分析类间关联的实例。这些关联或者已经存在，或者由链接定义对关联的需求。这一步应该概括出所有显见的关联并反映在与用况实现有关的类图中。
- 图中的次序不应作为主要的关注点，并且当它难以维护或者造成图的混乱时，可以考虑将它去掉。相反，对象和每个特定对象的需求（对消息的捕获）间的关系（链）应该是主要的关注点。
- 协作图应该处理所实现的用况中的所有关系。例如，如果用况A通过泛化关系特化出用况B，则实现用况A的协作图需要引用用况B的实现（如引用其协作图）。

在某些情况下，可以采用文本描述对协作图进行补充，尤其是当用很多图实现同一个用况或用若干图来表示若干很复杂的流时即是如此。这些文本描述应该在用况实现的事件流分析中捕获。

8.6.2.3 捕获特殊需求

在这一步，我们捕获对一个用况实现的所有需求，如非功能性需求；这个用况实现是在分析阶段中确定的，但应该在设计和实现阶段中进行处理。

举例：用况实现的特殊需求

由用况“支付账单”的实现所得到的特殊需求包括如下要求：

- “账单”类必须是持久的。
- “订单处理程序”类必须能每小时管理10 000个事务。

在捕获这些需求时，尽可能参照已由构架设计师确定的公用特殊需求。

8.6.3 活动：分析类

分析一个类的目的（参见图8-26）在于：

- 依据分析类在用况实现中的角色来确定和维护它的职责。
- 确定和维护分析类的属性及其关系。
- 捕获对该分析类实现的特殊需求。

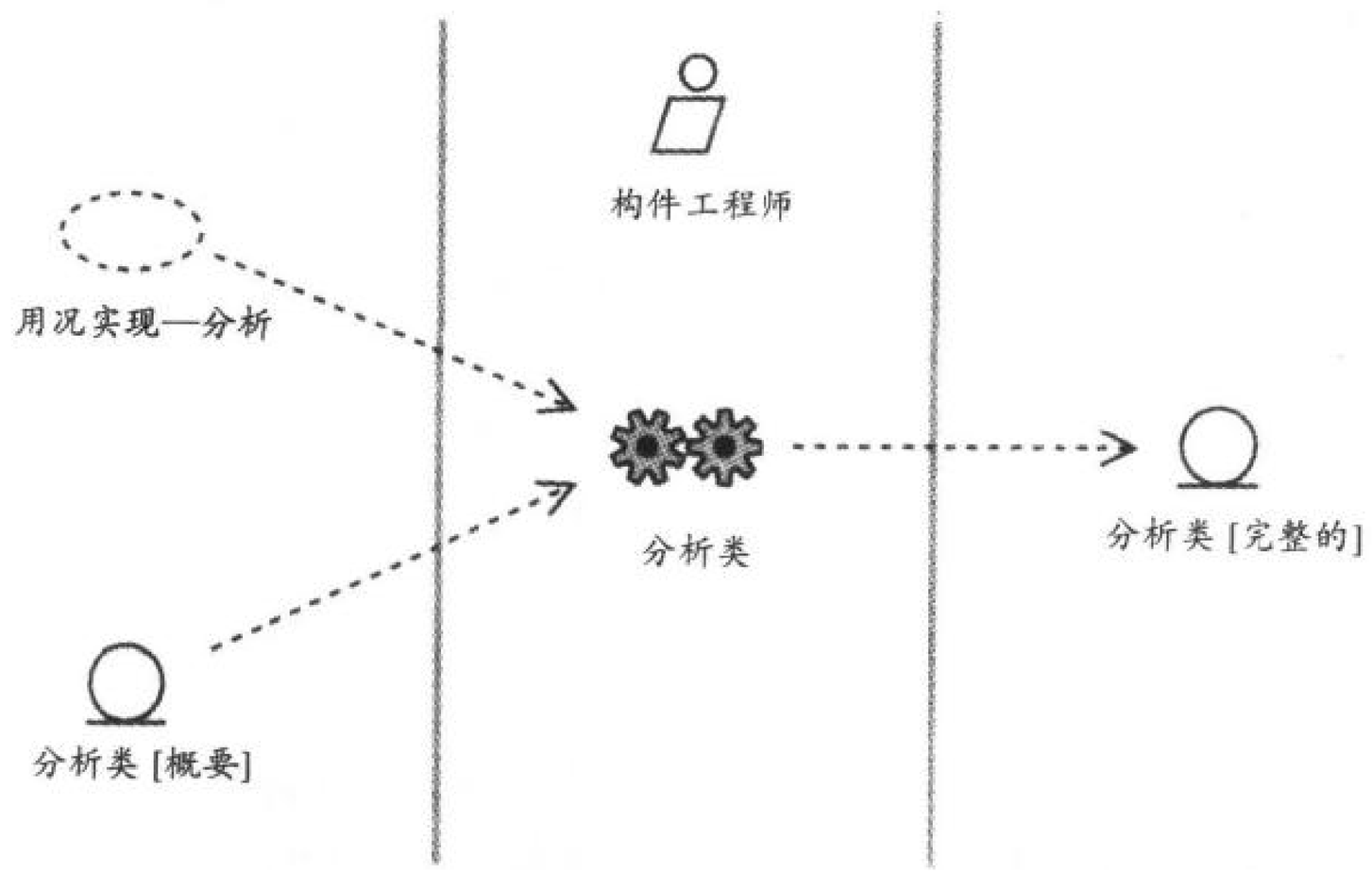


图8-26 分析类的输入和结果

8.6.3.1 确定职责

通过组合一个类在不同用况实现中所充当的所有角色可以用来汇集该类的职责。通过研究它们的类图和交互图可以确定该类参与的所有用况实现。还要记住，关于类的每个用况实现的需求，有时是在用况实现的事件流分析中用文本进行描述的。

举例：类的角色

“账单”对象是在用况“给买主开单”中创建的。卖主执行该用况来请求买主支付订单（订单是在用况“订购货物或服务”中创建的）。在这个用况期间，账单传递给买主，随后由他决定支付账单。

“支付”在“支付账单”用况中实现，此时“支付调度程序”对象调度“账单”对象来确定支付日期。之后，支付账单，并关闭“账单”对象。

然而需要注意的是，参与用况“给买主开单”和用况“支付账单”的是“账单”对象的同一个实例。

有多种方式可以用来汇集类的职责。一种简单的方法是每次从一个角色中抽取出职责，再附加上每次基于一个用况实现得到的补充职责或改变现存的职责。

举例：类的职责

“支付调度程序”类有以下职责：

- 创造一个支付请求。

- 跟踪已确定支付时间的支付，当实施支付或取消支付时发送一条通知。
- 在规定的支付日期启动金额的转账。
- 当账单已确定了支付日期和已经支付（即关闭）时通知“账单”对象。

8.6.3.2 确定属性

属性详细说明了分析类的特性，它通常由类的职责直接或间接决定（如在前面的步骤中所介绍的）。在确定属性时，应记住以下一般性的指南：

- 属性的名称应是一个名词[1，2]。
- 记住，属性的类型在分析阶段应该是概念上的，并且如果可能的话，它们不应受到实现环境的限制。例如，分析阶段中“数量”可能是一种合适的类型，而设计阶段中它所对应的类型应为“整型”。
- 在选择一种属性类型时，尽量重用已存在的类型。
- 一个简单的属性实例不能由多个分析对象共享。如果需要这样，属性要在它自己的类中进行定义。
- 如果一个分析类由于自己的属性变得过于复杂而难以理解，可以将其中的一些属性分成它们自己的类。
- 实体类的属性通常很明显。如果要一个实体类跟踪到一个领域类或一个业务实体类，这些类的属性可作为重要的输入。
- 与由人充当的参与者进行交互的边界类，其属性经常表示由该参与者所操作的信息项，例如带标记的文本域。
- 与由系统充当的参与者进行交互的边界类，其属性经常表示通信接口的特性。
- 控制类由于其生命周期短暂，因此很少具有属性。然而在用况实现期间，控制类可能会有代表聚集或派生价值的属性。
- 有时并不需要形式化的属性。相反，对由分析类处理的特性进行简单说明就足够了，并且还可将它放到该类的职责描述中。
- 如果一个类具有很多属性或复杂属性时，你可以在一个只显示属性栏的单独的类图中加以说明。

8.6.3.3 确定关联和聚合

分析对象通过协作图中的链彼此进行交互。这些链通常是分析对象的相应类之间关联的实例。因此，构件工程师应该研究这些用于协作图中的链以便确定需要哪些关联。这些链可以表明对象间所需的引用和聚合。

类之间的关系数目应尽量少。它主要不是现实世界中那种能作为聚合或关联来建模的关系，而是为了响应各种用况实现的需求而必须存在的那种关系。分析阶段的重点不在于通过聚合或关联来对最佳搜索路径建模。这项工作最好放在设计和实现阶段进行处理。

构件工程师还要定义关联的多重性、角色名称、自关联、关联类、有序角色、限定角色和*n*元关联（参见附录A）。参阅[2，3]。

举例：分析类间的关联

账单是对一张或多张订单的支付请求（参见图8-27）。这可以表示为一个带有“1..*”多重性的关联（一般至少有一张订单与账单关联）和角色名称“支付订单”。

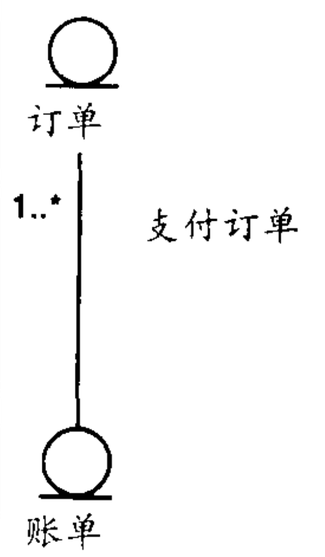


图8-27 账单是对一张或多张订单的支付请求

当对象表示以下事物时应该使用聚合：

- 实际中相互包容的概念，如汽车包含驾驶员和乘客。
- 相互间存在组成关系的概念，如汽车由发动机和车轮等部件组成。
- 构成对象的概念性集合的概念，如家庭由父亲、母亲和孩子组成。

8.6.3.4 确定泛化

在分析阶段，从几个不同的分析类中抽取共享和公用的行为时应该使用泛化。泛化应处于较高的概念层，其主要目的是使分析模型易于理解。

举例：确定泛化

“账单”类和“订单”类具有相似的职责，二者都是更一般对象“贸易”的特化；参见图8-28。

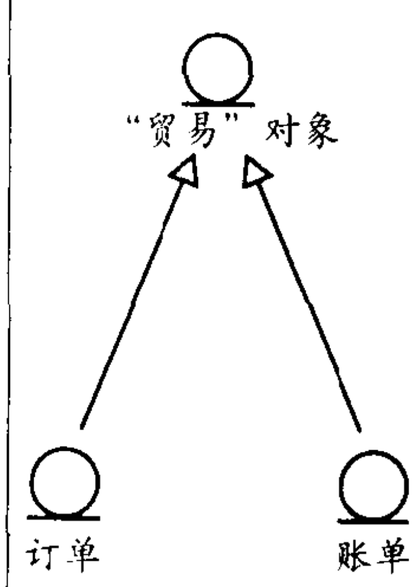


图8-28 “贸易”对象是“账单”和“订单”的泛化

在设计阶段，应该对泛化关系进行调整，以便更好地适应所选的实现环境，如程序设计语言。泛化有可能消失，而变为由其他关系（如关联）来实现。

8.6.3.5 捕获特殊需求

在这一步，我们将捕获在分析阶段中确定的、但应该在设计和实现阶段中进行处理的分析

类的所有需求（如非功能性需求）。在执行这一步时，务必要研究用况实现的特殊需求，因为它们可能包含了与分析类有关的补充（非功能性的）需求。

举例：捕获与分析类有关的特殊需求

对“账单”类的持久性需求的特征限定如下：

- 大小范围（*size range*）：每个对象2到24K字节
- 容量（*volume*）：最大为100 000
- 更新频率（*update frequency*）：
  - 创建/删除：1 000次/天
  - 更新：30次/小时
  - 读取：访问1次/小时

在捕获这些需求时，应该尽可能参照已由构架设计师确定的公用特殊需求。

8.6.4 活动：分析包

如图8-29所示，分析一个包的目的在于：

- 确保该分析包尽可能与其他分析包无关。
- 确保该分析包能够达到实现一些领域类或用况的目的。
- 描述依赖以便能够对未来变化的影响进行估计。

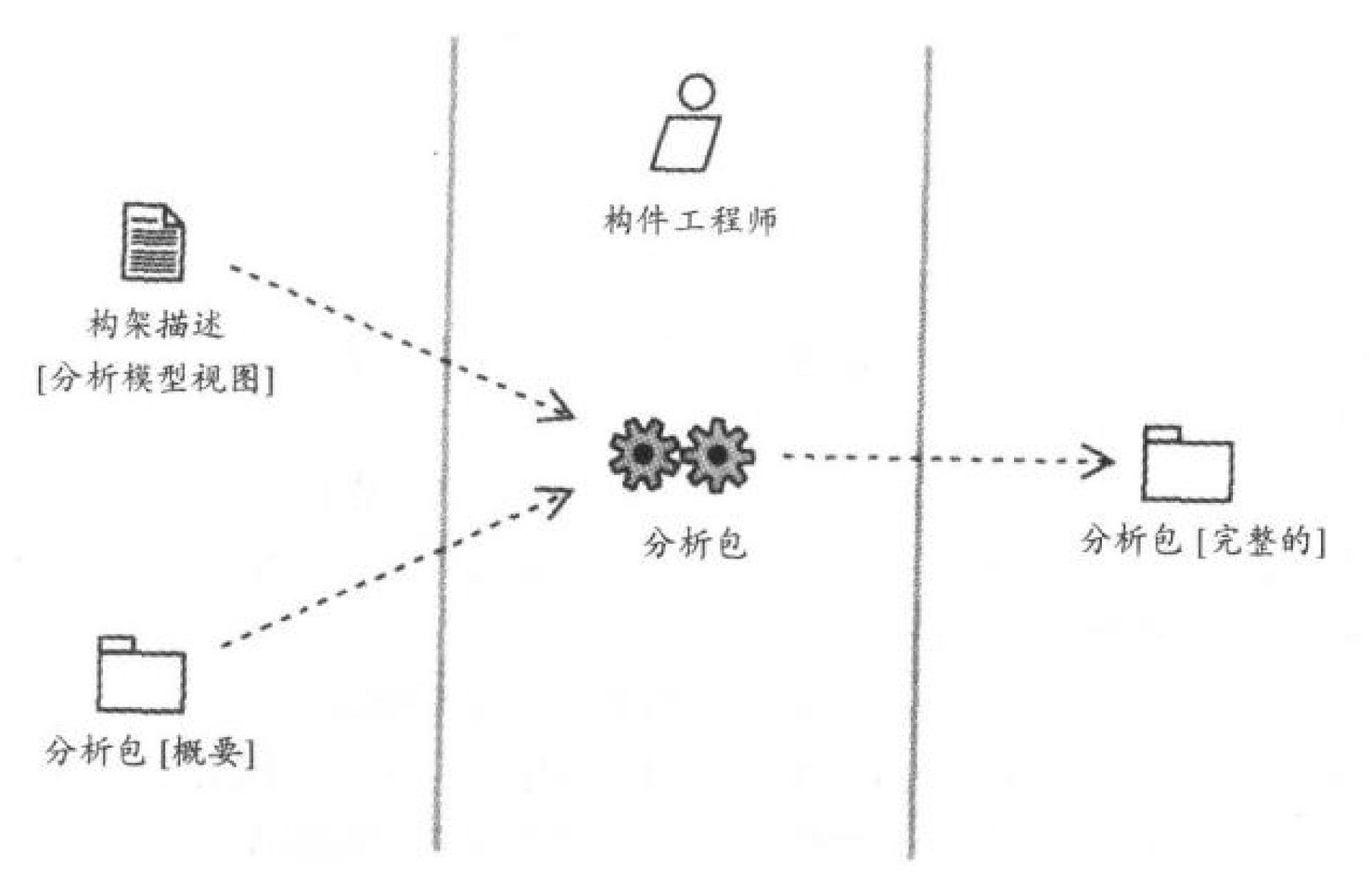


图8-29 分析包的输入和结果

下面是这种活动的一些通用的指南：

- 定义和维护该包与其他包（含有与该包相关联的类）的依赖。
- 确保包中包含恰当的类。通过只包含功能相关的对象来设法增强包的内聚性。
- 限制对其他包的依赖。如果包中包含了过于依赖其他包的类，应考虑对这些类进行重新

定位。

举例：包的依赖

“卖主账单管理”包中包含“账单处理”类，它与“账户管理”包中的“账户”类相互关联。这需要两个包之间具有对应的依赖关系（参见图8-30）。

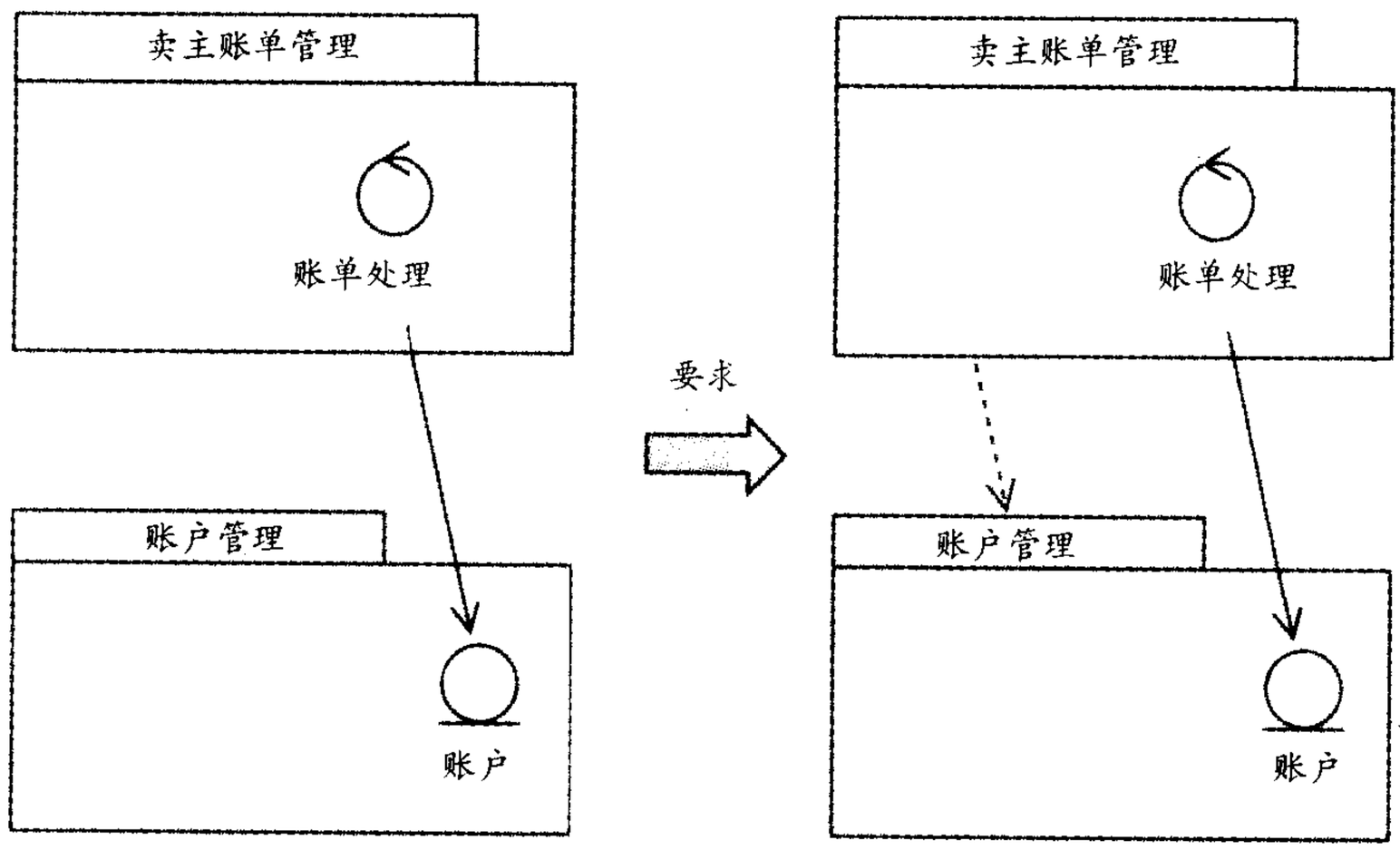


图8-30　所要求的包的依赖关系

8.7　分析小结

分析工作流（参见附录C）的结果是分析模型，它是通过精化和组织需求来对其进行分析的概念性对象模型。分析模型包括以下元素：

- 分析包和服务包以及它们之间的依赖关系和内容。分析包可以将变化局限于一个业务过程、一个参与者行为或一个紧密相关的用况集合中。服务包能够将变化局限于系统所提供的个别服务中，而且它们为分析阶段中构造重用提供了主要工具。
- 分析类、它们的职责、属性、关系和特殊需求。每个控制类、实体类和边界类都将变化局限于它们所代表的（构造型的）行为和信息内。用户界面或通信接口中的变化通常被局限在一个或多个边界类中；由系统处理的长效且经常为持久的信息通常被局限在一个或多个实体类中；涉及到多个（边界或实体）对象的控制、协调、排序、事务处理以及有时为复杂事务逻辑的变化通常被局限在一个或多个控制类中。
- **用况实现—分析**描述如何按照分析模型中的协作以及它们的特殊需求来精化用况。用况实现将变化局限在用况中，因为如果用况发生了变化，则它的实现也需要改变。
- 分析模型的构架视图，包括对构架重要的元素。**构架视图**（参见附录C）将变化局限在构架中。

分析模型被认为是后续设计活动的主要输入，这将在后面的章节中进行介绍。当为此而使用分析模型时，不要改变通过处理非功能性需求的主要部分和其他与实现环境有关的约束设计系统时定义的结构。更具体地说，分析模型会对设计模型产生如下的影响：

- 分析包和服务包会分别对专用应用层和通用应用层的设计子系统和服务子系统产生重要影响。多数情况下，包与相应的子系统间存在一对一的（同构）跟踪关系。
- 分析类将用作设计类的规格说明。在对具有不同构造型的分析类进行设计时，需要采用不同的技术和技巧；例如，实体类的设计通常要用到数据库技术，而边界类的设计通常要用到用户界面技术。分析类及其职责、属性和关系将用作创建设计类中相应的操作、属性和关系的（逻辑）输入。另外，当考虑采用诸如数据库和用户界面技术时，在分析类中捕获的大多数特殊需求将由相应的设计类进行处理。
- **用况实现—分析**有两个主要目的：一是有助于针对用况创建更为精确的规格说明。取代用状态图或活动图来详细描述用况模型中的用况，将每个用况描述为分析类间的协作，将得到该系统的一个详尽的形式化的需求规格说明。**用况实现—分析**还可作为设计用况时的输入。它们有助于确定那些需要参与相应的**用况实现—设计**的设计类。它们还有助于在设计对象间概括出一个初步的交互序列。另外，当考虑采用诸如数据库和用户界面技术时，在**用况实现—分析**中捕获的大多数特殊需求将由相应的**用况实现—设计**进行处理。
- 分析模型的构架视图在创建设计模型的构架视图时用作输入。（不同模型的）不同视图中的元素很可能彼此可以相互跟踪。这是由于构架重要性的概念倾向于经由跟踪依赖而在不同模型间进行平滑过渡。

8.8 参考资料

[1] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Övergaard, *Object-Oriented Software Engineering: A Use-Case–Driven Approach,* Reading, MA: Addison-Wesley, 1992. (Revised fourth printing, 1993.)

[2] James Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen, *Object-Oriented Modeling and Design,* Englewood Cliffs, NJ: Prentice Hall, 1991.

[3] OMG Unified Modeling Language Specification. Object Management Group, Framingham, MA, 1998. Internet: www.omg.org.

第9章 设　计

9.1 引言

在设计阶段，我们将构造系统，并获得实现所有需求（包括非功能性需求和其他约束）的系统组织（包括系统构架）。需求分析结果（即分析模型）是设计的基本输入（参见表9-1）。分析模型提供了对需求的详细理解。更重要的是，分析模型提供了一个在构造系统时需要尽可能保持的系统结构（参见第8章，8.2节）。尤其是，设计的目的在于：

- 深入理解与非功能性需求和约束相联系的编程语言、构件重用、操作系统、分布与并发技术、数据库技术、用户界面技术、事务管理技术等相关问题。
- 通过对单个子系统、接口和类的需求捕获，为后续的实现活动创建适当的输入和出发点。
- 能够把实现工作划分成更易于管理的各个部分，而且尽可能并发地由不同的开发组去开发。这一点在无法基于需求获取（包括用况模型）或分析（包括分析模型）的结果来划分实现工作时是很有用的。例如当不易获取需求和分析结果时就是如此。
- 在软件生命周期的早期捕获子系统之间的主要接口。这一点在理解系统构架和使用接口作为保持不同开发组之间同步的手段时都是很有用的。
- 通过使用通用的符号，可以可视化地刻画和思考设计。
- 建立对系统实现的无缝抽象，把实现看成是设计的直接精化。它不改变结构，只填入“血肉”。这使得应用代码生成以及在设计和实现之间的双向工程等技术成为可能。

表9-1　分析模型和设计模型的简要比较

分析模型	设计模型
概念模型，因为它是系统的一个抽象并回避了实现问题	物理模型，因为它是实现的蓝图
对设计是通用的（适用于多种设计）	对设计不是通用的，针对特定的实现
对类型有三种（概念性的）构造型：控制类《control》、实体类《entity》、边界类《boundary》	对类型有任意数量（物理的）构造型，依赖于实现语言
不太形式化	比较形式化
开发费用较低（是设计费用的1/5）	开发费用较高（是分析费用的5倍）
层数少	层数多
动态的（但并不特别关注时序）	动态的（特别关注时序）
勾画系统的设计轮廓，包括系统的构架	进行系统的设计，包括系统的构架（其中的一个视图）
主要通过研讨会等方式创建	主要是在双向工程环境中通过“可视化程序设计”来创建；设计模型和实现模型需要开发成“双向工程”的（在第10章阐述）

（续）

分析模型	设计模型
可能不需要在整个软件生命周期中都做维护	在整个软件生命周期中都应该维护
定义作为构造系统基本输入的构架，包括创建设计模型。	在尽可能保持需求模型所定义结构的前提下构造系统

在本章及后续章节中，将阐述如何达到以上目标。我们将采用与分析工作流类似的方法来讨论设计工作流（参见图9-1）。

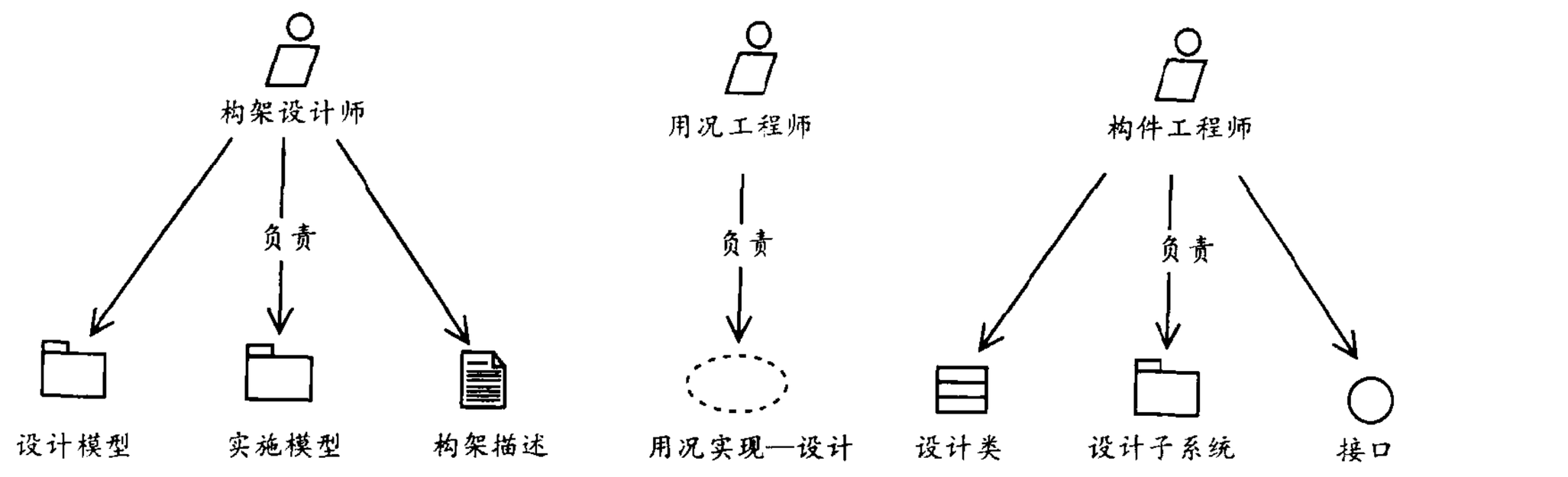


图9-1 包含在设计中的工作人员和制品

9.2 设计在软件生命周期中的作用

设计工作集中在细化阶段的末期到构造阶段的初期（参见图9-2），它将产生合理而稳定的构

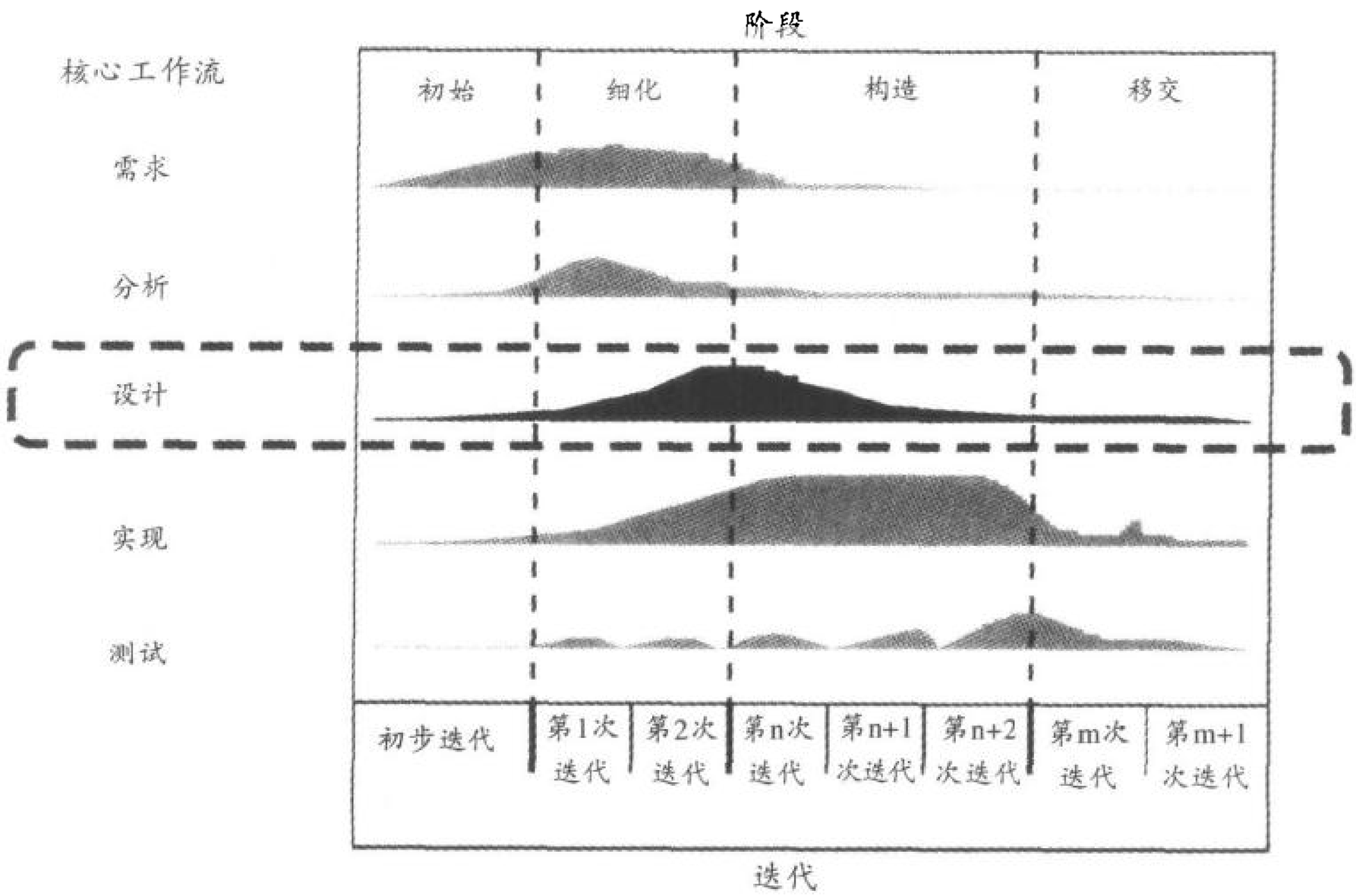


图9-2 设计的焦点

架，并创建实现模型的蓝图。在随后的构造阶段，当系统构架已经稳定并且需求已经被很好理解后，重点将转向系统的实现。

设计模型非常接近实际的系统，在整个软件生命周期里很自然地要保持并维护好设计模型，尤其是在双向工程中，设计模型可用来可视化地刻画系统实现并支持图形化编程技术。

9.3 制品

9.3.1 制品：设计模型

设计模型是一个用于描述用况物理实现的对象模型，重点关注功能性需求和非功能性需求，同时也关注与实现环境有关的、并最终影响系统的其他约束。另外，设计模型作为系统实现的抽象，因而用作系统实现中活动的基本输入。

设计模型定义了如图9-3所描述的层次结构。

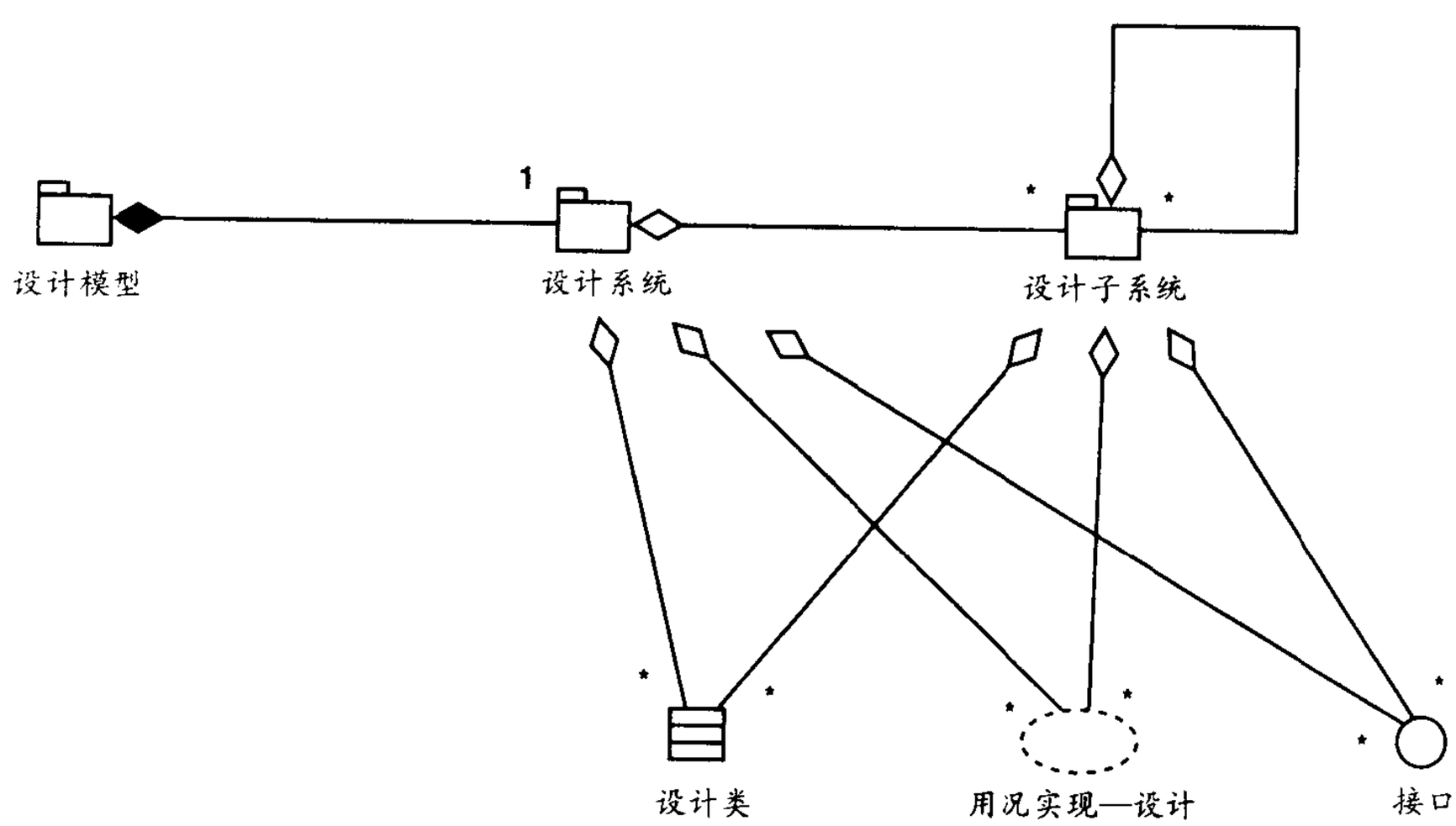


图9-3 设计模型是一个包含设计类、**用况实现—设计**和接口等设计子系统的层次结构

设计模型由设计系统来表示；设计系统表示模型的高层子系统。利用其他子系统是将设计模型划分成更易于管理的功能块的一种方法。

设计子系统和设计类是系统实现中子系统和构件的**抽象**（参见附录C），这些抽象是简明的，它表示设计和实现间的简单映射。

在设计模型中，设计类及其对象实现了用况。这体现在设计模型内部成分的协作上，并称之为用况实现—设计（*use-case realization—design*）。需要注意的是，**用况实现—设计**不同于**用况实现—分析**。前者描述一个用况是如何用设计对象的交互来实现的，而后者则描述一个用况是如何用分析对象的交互来实现的。

下面将详细讨论设计模型中的各种制品。

9.3.2 制品: 设计类

设计类是系统实现中的一个类或相似构造的无缝抽象（参见图9-4）。抽象的无缝性表现在如下几个方面：

- 使用与编程语言相同的语言来描述设计类。因此，操作、参数、属性、类型等都用选定的编程语言的语法来详细说明。
- 经常需要指明设计类的属性和操作的可视性。如采用普遍用于C++中的*public*、*protected*、*private*等关键字。

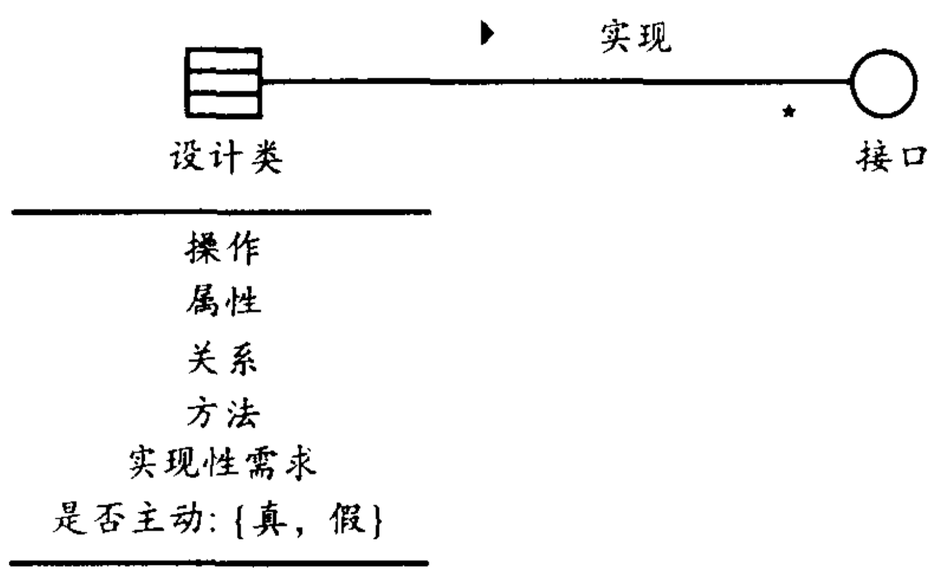


图9-4 一个设计类的关键属性与关联

- 当实现一个设计类时，该类与其他类之间的关系应该有直截了当的含意。例如，泛化或泛化的任何构造型与编程语言中相应的泛化（或继承）的含义有对应的语义。也就是说，关联和聚合经常映射到实现中相应类的变量（属性），从而在对象之间提供引用关系。
- 设计类的方法（即操作的实现）直接映射到该类的实现（即编码）中相应的方法。如果在设计中详细说明了方法，则它们一般是用自然语言或伪代码来表达的，因而也可以作为方法实现中的注释。当然，这是设计和实现之间不多的几个主要抽象之一，且因我们推荐由同一个开发人员担任设计和实现一个类的工作，因而这种注释并非特别必要（参见9.4.3节）。
- 一个设计类可以通过标上实现性需求（*implementation requirement*），而把这些需求推迟到后续的实现活动中处理。从而可以将那些不宜在设计模型中确定的问题推后，比如那些与该类的编码有必然联系的问题。
- 一个设计类通常给出构造型，它可以无缝地映射到某种指定编程语言所提供的结构。例如，一个将用VB实现的设计类通常表达为《class module》、《form》、《user control》等构造型。
- 如果编程语言支持，一个设计类能够实现（并因此提供）接口。例如，一个表示Java类的设计类能够提供一个接口。

- 一个设计类可以是主动的，这意味着该类的对象维护它们自己的控制线程，并与其他的主动对象并发运行。然而，设计类通常都是非主动的，这意味着它们的对象“运行”在其他主动对象控制的特定空间之下。此外，有关这方面的详细语义依赖于所用的编程语言以及并发或分布技术。需要注意的是，主动类和非主动类的语义是明显不同的：主动类可以驻留在它们自己的过程模型中，而不是在设计模型中。特别是当有许多主动类而且其对象有复杂的交互时，比如在一些实时系统中，就显得尤其明显。

举例：“账单”设计类

图9-5说明在设计中详细描述的“账单”设计类。在分析阶段提出的“账户”属性已经转化成与一个“账户”类的关联。

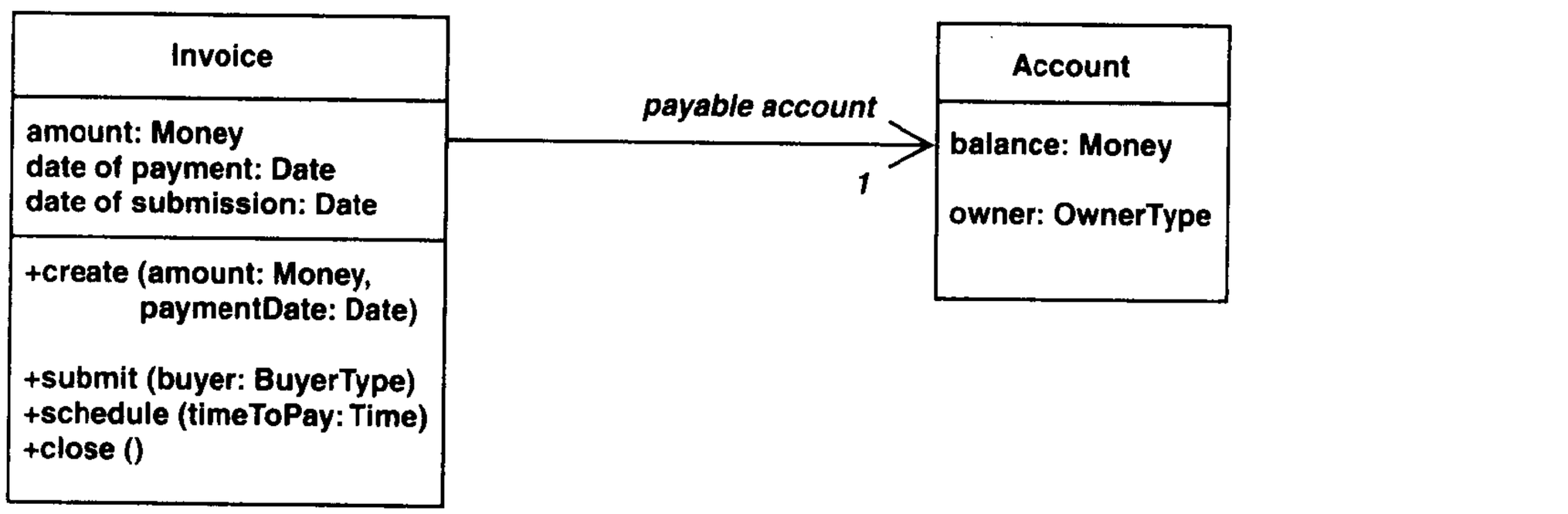


图9-5 拥有属性、操作和一个将“账户”对象与“账单”对象相联系的关联的设计类

9.3.3 制品：用况实现—设计

用况实现—设计是在设计模型内的协作关系，以设计类及其对象为基础，描述了一个特定用况的实现和执行。一个 *用况实现—设计* 可以直接跟踪到需求模型中的一个 *用况实现—分析*（参见图9-6）。注意，一个**用况实现—设计**也因此能够通过**用况实现—分析**跟踪到用况模型中的某个用况。

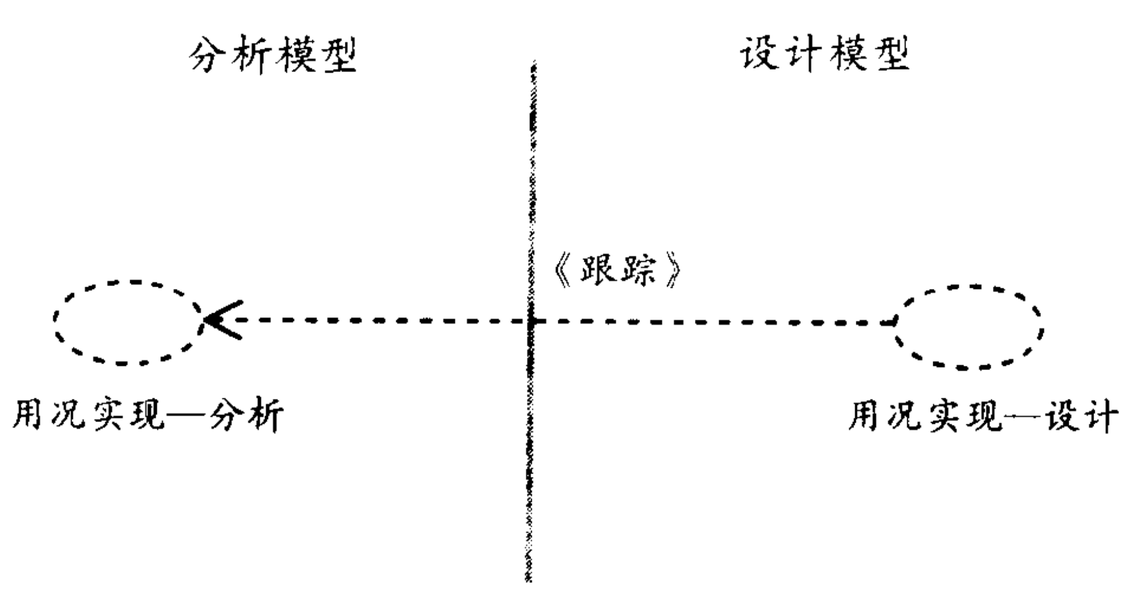


图9-6 存在于不同模型中用况实现之间的跟踪关系

当分析模型只是为了创建一个好的设计，并不需要在整个软件生命周期中维护时，就不会有**用况实现—分析**。这时**用况实现—设计**将直接跟踪到用况模型中的用况。

用况实现—设计有如下内容：1）文字性的事件流描述；2）描述参与的设计类的类图；3）根据设计对象之间的交互，描述特定流的实现或用况脚本的交互图（参见图9-7）。如果需要，这些图也能描述包含在用况实现中的子系统和接口（例如，包含参与的设计类的子系统）。

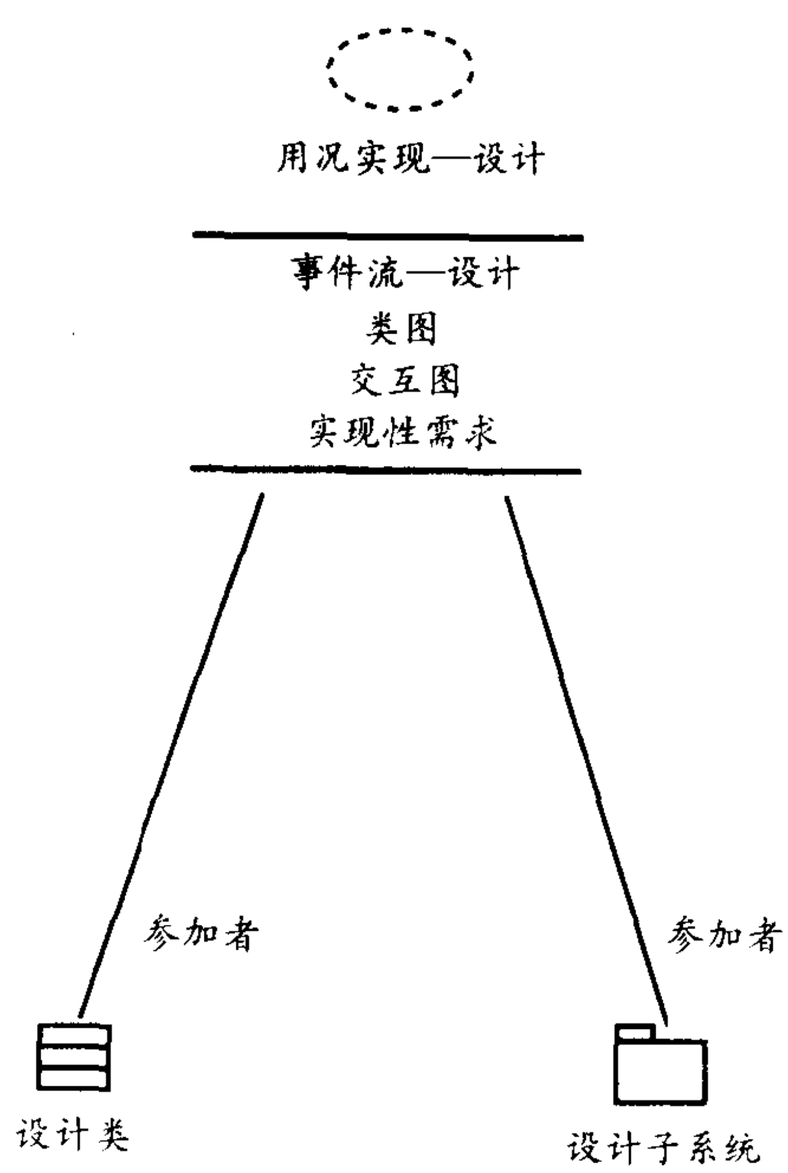


图9-7 用况实现—设计的关键属性和关联

用况实现—设计为相应的**用况实现—分析**提供了物理实现，而且它还处理在**用况实现—分析**中所捕获的大部分非功能性需求（如一些特殊的要求）。此外，**用况实现—设计**能像设计类那样，可以将一些需求标为实现性需求，而将其推迟到后续的实现活动中去处理。

9.3.3.1 类图

一个设计类和它的对象以及含有该设计类的子系统，经常会参与多个用况的实现。但也可能是一个特定的设计类的一些操作、属性和关联关系只跟一个用况实现相关。因此，协调好对一个类及其对象的所有需求以及与不同用况实现有关的子系统的需求间的关系就非常重要。为了处理好这一关系，我们用跟一个用况实现相联系的类图来描述参与的类、子系统以及它们之间的关系。采取这种方法，可以保持对用况实现中的参与元素进行跟踪。

类图的示例在9.5.2.1节和9.5.2.3节中给出。

9.3.3.2 交互图

当一个参与者通过向系统发送一些消息而调用某个用况时，就开始执行这个用况中的活动序列。如果从系统内部考虑，将有一些设计对象来接收该参与者的消息，然后这些设计对象再调用其他的设计对象，从而通过所有参与对象的交互来实现和执行该用况。在设计过程中，因

为我们关注的重点是按时间顺序排列的交互序列，所以倾向于采用顺序图。

在某些情况下，在顺序图中包含子系统，以此来说明参与某个用况实现的是哪些子系统，同时很可能还要说明参与用况实现的子系统所提供的接口。通过这种方法，在对参与的子系统进行内部设计之前，就可在较高的层次上对用况进行设计。比如，在软件生命周期早期进行子系统的内部设计之前就需要确定子系统的接口时，这种方法就很有用。

使用顺序图，可以用对象或子系统的生命线之间的消息传送来说明对象的交互关系。当我们说一个消息被一个子系统生命线所“接收”，实际上是指子系统内的某个类的对象接收了这个消息。而当一个消息从子系统生命线“发送”出去，实际上是指子系统内的某个类的对象发送了这个消息。消息的名称应该指出被调用对象的一个操作，或者是由这个对象所提供（实现）的接口。

交互图的示例在9.5.2.2节中给出。

9.3.3.3 事件流—设计

一个用况实现的各种图（尤其是交互图）通常很难理解。因此，用来解释和补充图和它们的标签的文字性描述的**事件流—设计**就非常有用。这些文字应该按照执行该用况而进行交互的对象或者按照参与的子系统来书写。然而，这些描述信息不应该提及任何有关对象的属性、操作或关联，否则将很难维护，因为设计类的属性、操作和关联经常变化。此外，如果图中使用了接口，还不应该提及任何接口操作。按照这样的方法，我们将能把因为所描述的图（尤其是交互图中表现在消息上的操作）的变化而导致对**事件流—设计**描述的修改降低到最小程度。

如果存在多个顺序图描述同样的用况实现，或者存在表示复杂流的图，则**事件流—设计**就特别有用。在进行**事件流—设计**时，要注意以下各点：

- 一个用况实现的**事件流—设计**并不局限于某个特定的顺序图，因此它可以用于描述几个图是如何关联的。
- 一个顺序图的标签（例如在一个活动激活时动作的时标或描述）局限于该图。然而，如果标签很多，会让图看起来很混乱。在这种情况下，一些标签的文字可以放到**事件流—设计**中。

当同时使用**事件流—设计**和标签时，它们应该相互补充。

关于**事件流—设计**描述的示例在9.5.2.2节中给出。

9.3.3.4 实现性需求

实现性需求是对某个用况实现的需求（如非功能性需求）集的文字描述。这些需求是在设计中捕获的，但最好是在实现时处理。其中某些需求可能已经在前面的工作流中捕获，因而只不过是把它们转化为一个用况实现。然而，其中某些需求也可能是在设计工作的进展过程中发现的全新的或者导出的需求。

关于**用况实现—设计**的实现性需求的示例在9.5.2.5节中给出。

9.3.4 制品：设计子系统

设计子系统提供了一种将设计模型中的制品组织成为更易于管理的功能块的方法（参见图9-8）。一个设计子系统可以包含设计类、用况实现、接口和其他的子系统（递归地）。而且，一个子系统

能够提供代表其与外界进行交互的功能接口。

一个子系统应该是紧耦合的，也就是说，它的内部元素的关系应该非常紧密。子系统间应该是松耦合的，也就是说，它们之间的相互依赖性或者是对相互的接口的依赖性应该是最小化的。

此外，所设计的子系统应该具有如下的特点：

- 子系统可以表示对设计问题的分解。例如，在一个大系统中，某些子系统可以由不同的、拥有各自设计技巧的开发组独立地（很可能并行地）进行设计。

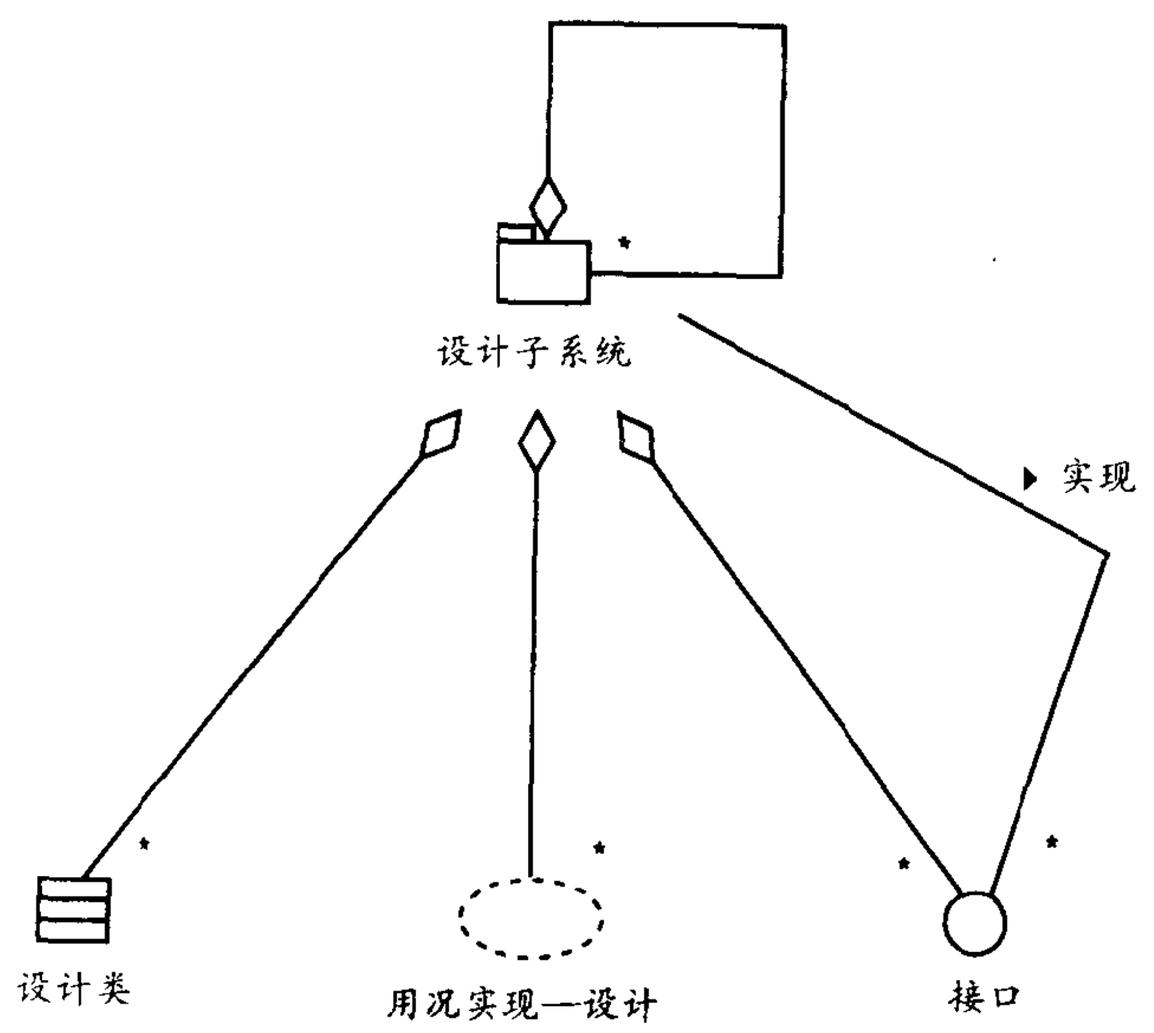


图9-8 子系统的内容和关键关联

- 在设计模型中最高的两个应用层以及它们的子系统可以直接跟踪到分析包或分析类。
- 子系统可以表示系统实现中的粗粒度构件；即由若干细粒度构件构成、可提供若干接口的构件，比如那些详细说明个别实现类的构件，它们可以将自己表达成在不同节点使用的可执行体、二进制代码或其他类似的实体。
- 子系统可以通过包装来表示可重用的软件产品。因而子系统在设计模型中可以用来说明可重用软件产品的集成。这样的子系统驻留在中间件层或系统软件层中。
- 通过包装，子系统可以用来表示整个遗留系统或部分遗留系统。子系统因而可以用来将遗留系统合并到设计模型中。

在9.5.1.2 节中，我们将对子系统、子系统的语用以及如何识别子系统及其接口的方法做更详细的阐述。

9.3.4.1 服务子系统

服务子系统在设计子系统层次结构中的较低层使用，这与分析模型中使用服务包有相同的原因。也就是说，通过把个体服务的变化限制在相应的服务子系统，来预防这些变化带来的不良影响。欲了解服务的详细情况，请参见第8章8.4.4.1节。

对服务子系统的识别是基于分析模型中的服务包，它们之间通常存在一对一（同构）的跟

踪关系。因此，服务子系统非常普遍地存在于系统最高的两个层次中（例如，专用应用层和通用应用层）。然而，服务子系统比相应的服务包要处理更多的问题，这基于如下原因：

- 服务子系统需要通过接口和它们的操作来提供服务。
- 服务子系统中以设计类取代了分析类，因此可以处理许多非功能性需求和其他与实现环境相关的约束。总之，相对于对应的服务包，服务子系统往往包含更多的类，并且为了控制它们的规模，需要将它们进一步分解成更小的子系统。
- 一个服务子系统常常对应于实现中的某个二进制代码或可执行的构件。但在某些情况下，一个服务子系统需要分割，并将每个部分分别置于不同的节点上，这也就意味着，每个节点需要实施一个二进制代码或可执行的构件。在这种情况下，该服务子系统需要进一步分解为更小的子系统，每个子系统封装有在某个特定的节点所需要的功能。

有关服务子系统的示例将在9.5.1.2.1节中给出。

需要说明的是，组织设计模型中的服务制品的一般方法仍然是使用在前面章节中所讨论的普通的设计子类。然而，我们在这里引入一种“服务子系统”《service subsystem》构造型，它能清楚地标志那些表示服务的子系统。这在大型系统（拥有许多子系统）中是非常重要的，它能用一种简单的方法来区分不同类型的子系统。关于这一点，请回顾一下第8章介绍的“服务包”《service package》构造型，两者有着相似的动机。

9.3.5　制品：接口

接口用于详细说明由设计类和子系统所提供的操作（参见图9-9）。

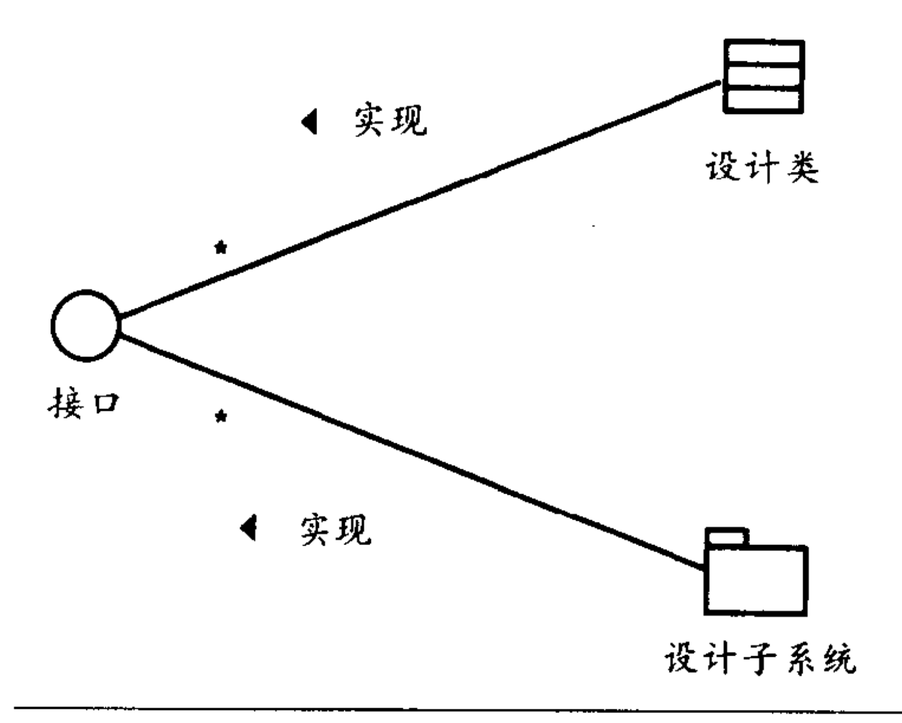


图9-9　接口的关键关联

一个提供接口的设计类，也必须提供实现该接口的操作的方法。一个提供接口的子系统，也必须包含提供该接口的设计类或者其他子系统（递归地）。

接口提供了一种方法，把基于操作的功能说明与基于方法的具体实现区分开来。这样的区分使得任何依赖或者使用接口的客户不必依赖于接口的具体实现。一个接口的具体实现（如某个设计类或者子系统）可以在不影响客户的情况下用别的实现来替换。

子系统间的绝大部分接口对于系统构架的意义非常重大，因为它们定义了子系统间是如何

交互的。在某些情况下，在软件生命周期中，在实现子系统的功能之前，尽早确定稳定的接口关系也是有用的。这些接口可以看作是对开发组在设计子系统时的需求，它们还可以用作并发地开发各个子系统的各个不同开发组间的同步工具[2]。

在9.5.1.2.4节中，我们将对接口、接口的语用以及如何识别接口的方法进行更详细的描述。

9.3.6 制品：构架描述（设计模型的视图）

构架描述包含**设计模型的构架视图**（参见附录C），它描述对构架有重要意义的制品（参见图9-10）。

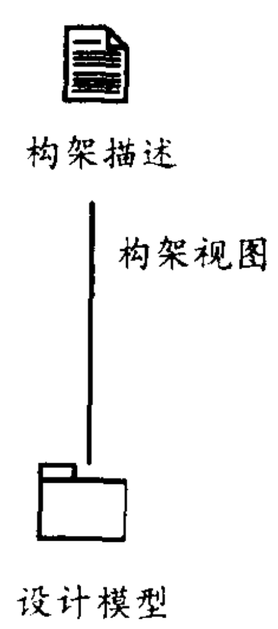


图9-10 含有设计模型的构架视图的构架描述

通常认为在设计模型中有着重要构架意义的制品如下：

- 将设计模型划分为子系统以及它们的接口和相互依赖关系的分解。这些分解通常对构架是非常重要的，因为子系统和它们的接口组成了系统的基础结构。
- 核心的设计类，如那些可以跟踪到对构架有重要意义的分析类的设计类、主动类[⊖]，以及那些处于中心地位的、通用的、表示通用设计机制、且与其他设计类有许多联系的设计类。通常，把一个抽象类（而不是其子类）看成是对构架有重要意义的制品就足够了，除非该子类具有一些不同于抽象类的有趣的、对构架有重要意义的行为。
- 实现某些需要在软件生命周期早期开发的重要而且关键的功能的**用况实现—设计**。它们包含许多设计类，因而具有很广的覆盖范围，甚至可能跨越几个子系统；或者只包含前面提到的关键的设计类。通常的情况是，对应的用况可以在设计模型的构架视图中找到，并且对应的**用况实现—分析**可以在分析模型的构架视图中找到。

在9.5.1.2、9.5.1.3以及9.5.1.4这三节中，我们将给出一些示例来说明在设计模型的构架视图中可以包含哪些东西。

9.3.7 制品：实施模型

实施模型是一个描述系统物理分布的对象模型，它是按照如何在各个计算节点当中分布功

⊖ 如果主动类驻于它们自己的处理模型中（早前已经提到过），则将改在处理模型的构架视图中去描述它们。

能来进行描述的（参见图9-11）。因为系统分布对设计有很大的影响，所以实施模型用作设计和实现活动的基本输入。

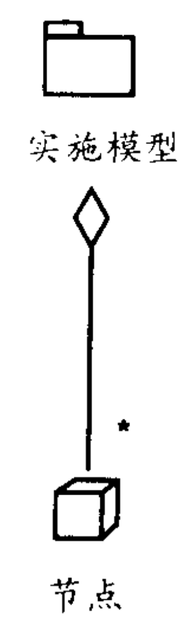


图9-11　含有节点的实施模型

如下是对实施模型的几点解释：

- 每个节点表示一个计算资源，通常表示一个处理器或者类似的硬件设备。
- 节点拥有表示相互间的通信方式（如因特网、内联网和总线等）的关联。
- 实施模型能够描述几种不同的网络配置，包括测试和模拟配置。
- 实施到一个节点上的构件定义了这个节点的功能（或者处理）。
- 实施模型本身表示软件构架与系统构架（硬件）之间的映射关系。

在9.5.1.1节中，我们给出若干实施模型的示例。

9.3.8　制品：构架描述（实施模型的视图）

构架描述包含**实施模型的构架视图**（参见附录C），用以描述对构架有重要意义的制品（参见图9-12）。

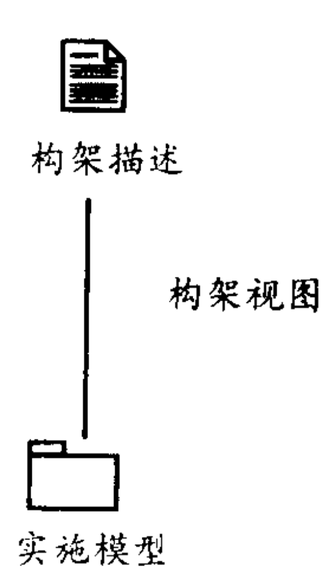


图9-12　含有实施模型的构架视图的构架描述

鉴于其重要性，实施视图的所有方面都要在构架视图中得以体现，其中包括在实现过程中得到的构件在节点上的映射关系。

在9.5.1.1节中，我们给出一些示例来说明在实施模型的构架视图中可以包括哪些东西。

9.4 工作人员

9.4.1 工作人员：构架设计师

在设计过程中，构架设计师负责保持设计和实施模型的完整性，以确保模型整体上是正确的、一致的和易读的（参见图9-13）。如同对分析模型那样，对于大型复杂系统，需要单独安排一个工作人员来负责设计模型（也就是设计系统）的顶层子系统的设计。

当模型实现了那些且仅仅是那些在用况模型、补充需求以及分析模型中描述的功能时，就可以认为该模型是正确的。

构架设计师还对设计和实施模型的构架负责，也就是说，要对该模型的构架视图中所描述的对构架有重要意义的部分的存在性负责。要记住这些视图是系统构架描述的一部分。

需要说明的是，构架设计师并不负责设计模型中的各种制品的继续开发和维护。这些工作由相应的用况工程师和构件工程师来负责完成（参见9.4.2和9.4.3节）。

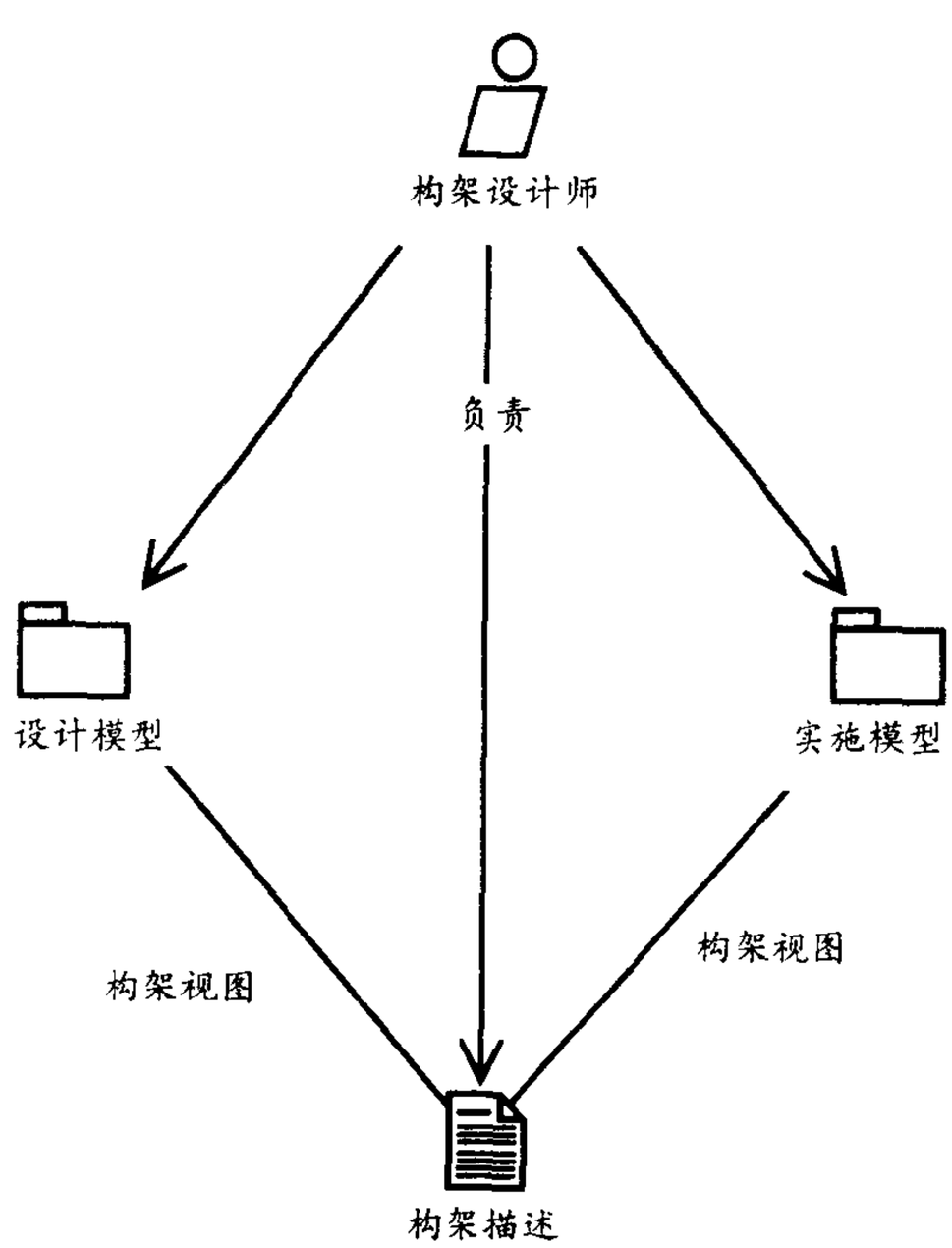


图9-13 构架设计师在设计中的职责

9.4.2 工作人员：用况工程师

用况工程师负责一个或多个**用况实现—设计**的完整性，以确保它们实现特定的需求（参见图9-14）。一个**用况实现—设计**必须正确地实现那些且仅仅是那些在分析模型中相应的**用况实现—分析**的行为以及在用况模型中相应用况的行为。

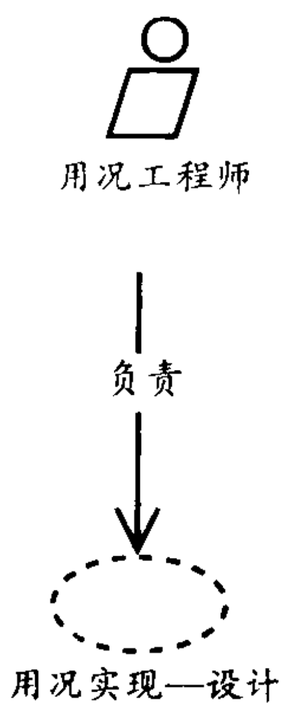


图9-14 用况工程师在设计中的职责

这包括使所有描述用况实现的文字说明和图形易读且符合它们的用途。

需要说明的是，用况工程师并不对用况实现中所使用的设计类、子系统、接口以及关系负责。相反，这些是相应的构件工程师的职责。

9.4.3 工作人员：构件工程师

构件工程师负责定义和维护一个或多个设计类的操作、方法、属性、关系以及实现性需求，以确保每个设计类实现由它所参与的用况实现对它提出的需求（参见图9-15）。

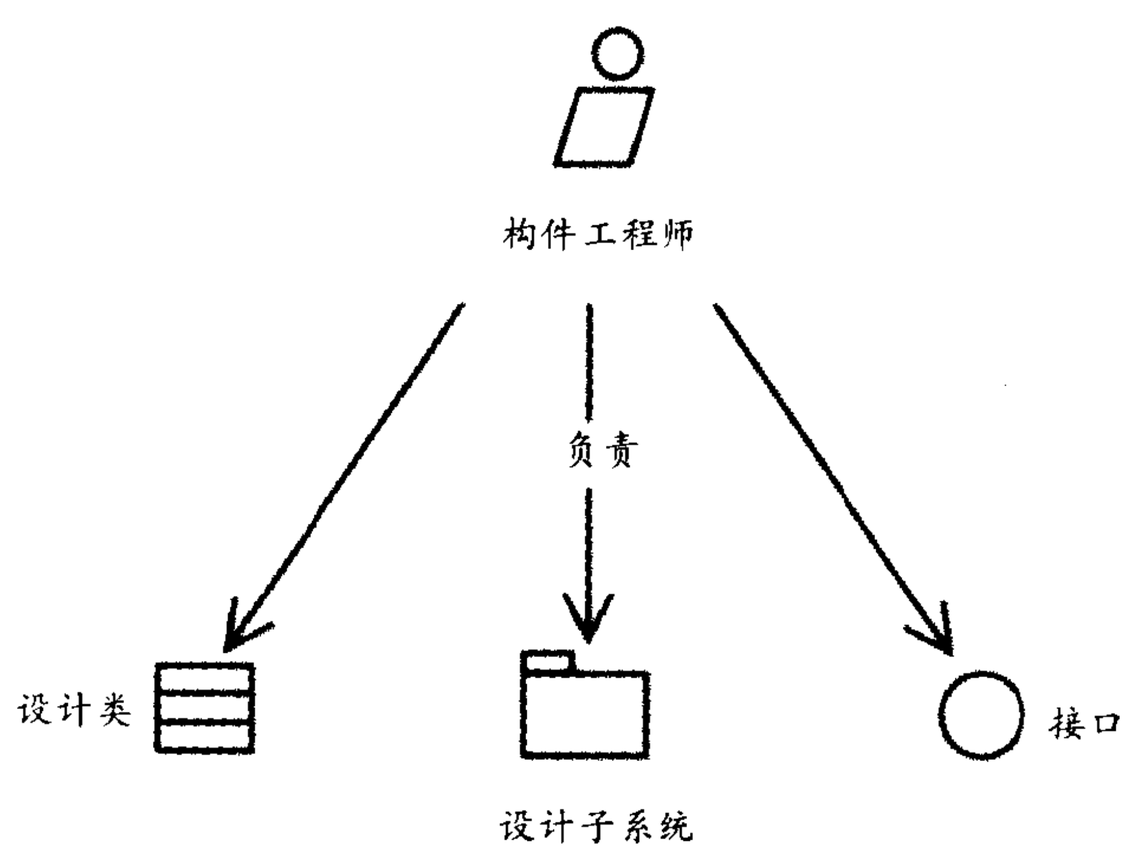


图9-15 构件工程师在设计中的职责

构件工程师还负责保持一个或多个子系统的完整性。这包括确保子系统的内容（如类及它们之间关系）是正确的，确保它们对其他子系统或接口的依赖是正确的和最小化的，还确保它们能正确地实现所提供的接口。

让负责子系统的构件工程师同时也负责该子系统所包含的模型元素常常是合适的。此外，为了实现平滑无缝的开发，由同一个构件工程师把设计模型中的制品（比如设计类和子系统）

带入实现工作流并负责实现也就是很自然的事了。

9.5 工作流

在本章前面的若干章节中，我们曾经描述在静态条件下的设计工作。现在，要用活动图来说明系统的动态行为；参见图9-16。

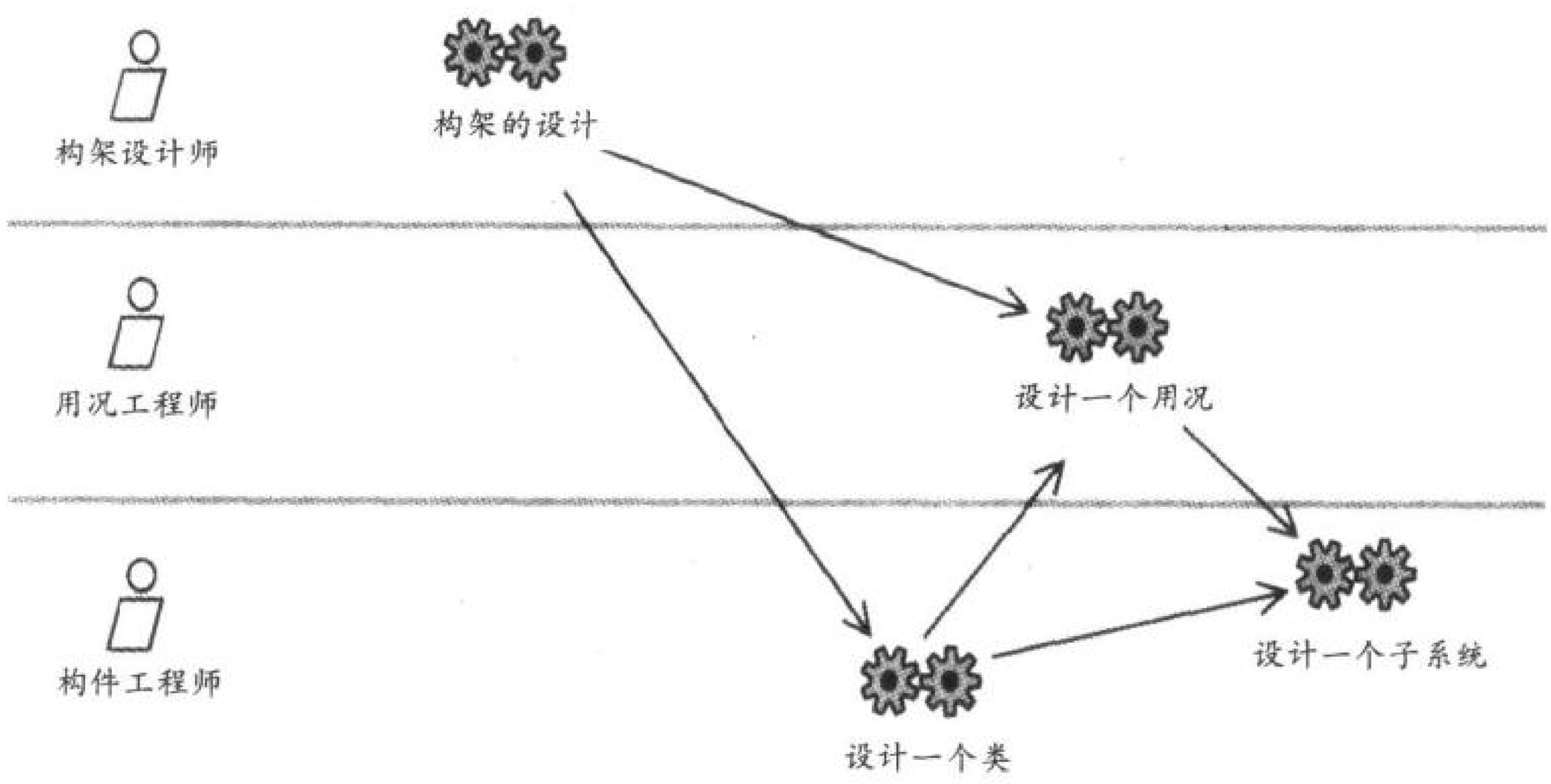


图9-16 设计中包括参与的工作人员及其活动的工作流

设计模型和实施模型（在本章的前面已经定义）的创建是由构架设计师启动的，他们勾画出实施模型的节点、设计模型中的主要子系统及其接口、包括主动类在内的重要的设计类以及通用的设计机制等。然后，用况工程师通过参与的设计类或子系统及其接口来实现每个用况。用况实现的结果规定了参与用况实现的每个类和子系统的行为需求。这些需求由构件工程师说明，并通过创建每个类的一致的操作、属性和关系，或者通过创建每个子系统所提供的接口的一致的操作，把这些需求集成到每个类中。

在设计工作流的整个过程中，随着设计模型的进化，开发人员要识别新的子系统、接口、类和通用的设计机制的候选方案，负责各个子系统的构件工程师要精化和维护这些子系统。

9.5.1 活动：构架设计

构架设计的目的是通过对如下内容的识别来勾画设计和实施模型及其构架（参见图9-17）：

- 节点及其网络配置。
- 子系统及其接口。
- 对构架有重要意义的设计类，如主动类。

- 处理共性需求的通用设计机制，如在对分析类和**用况实现—分析**的分析过程中捕获的有关系统持久性、分布特征、性能等方面的特殊需求。

在整个构架设计活动过程中，构架设计师要考虑各种重用的可能性，比如可重用相似系统的一部分或通用的软件产品。要把所设计的子系统、接口和其他设计要素都合并到设计模型中。构架设计师还要维护、精化和更新构架描述以及设计模型和实施模型的构架视图。

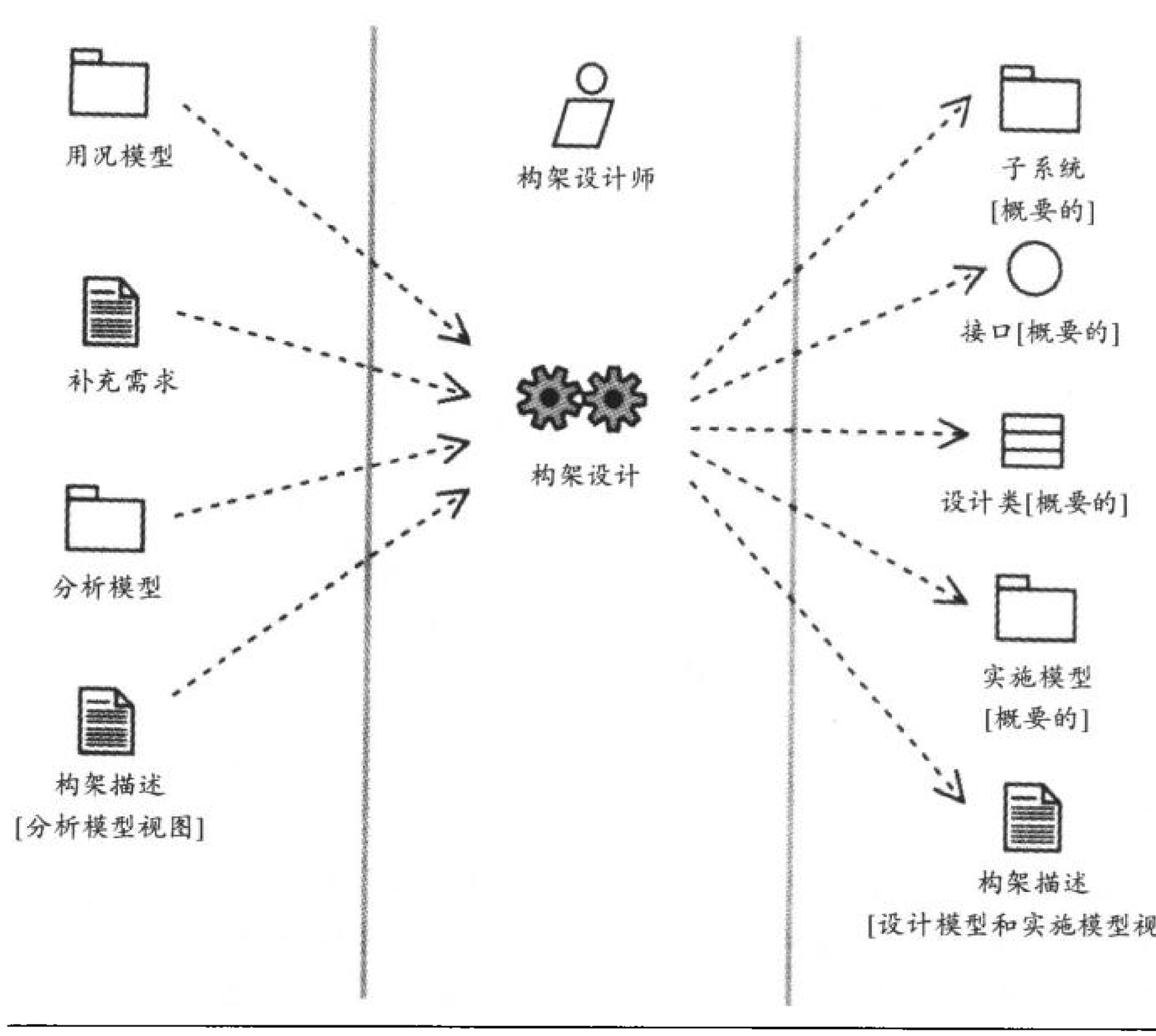


图9-17 构架设计的输入和结果

9.5.1.1 识别节点和网络配置

物理网络的配置通常对软件构架有很大影响，这包括所需要的主动类以及网络节点间的功能分布。通常的网络配置使用三层模式，即把系统分离为客户（用户交互）层、数据库功能层和业务/应用逻辑层。简单的客户/服务器模式是三层模式的一个特例，在这种模式中，业务/应用逻辑放到了其他层的一层（即客户层或数据库层）中。

网络配置包括如下几个方面：

- 包含哪些节点以及这些节点在处理能力和存储容量方面有些什么要求？
- 节点间采取何种连接方式，采用什么样的通信协议？
- 连接方式和通信协议有些什么特征（比如带宽、有效性和质量）？
- 是否需要有容错处理能力、错误恢复模式、处理移植、数据备份等要求？

在了解了节点和连接方式的局限性和可能性后，构架设计师才能综合选择各种技术，比如，

采用对象请求代理和数据复制服务等来简化系统分布的实现。

举例：Interbank系统的网络配置

Interbank系统将运行在三个服务器节点和大量客户节点上。首先，其中一个服务器节点是为买主准备的，另一个服务器节点是为卖主准备的，因为买卖双方各自都需要为它们的业务对象和处理准备一个中心服务器。终端用户通过客户端节点来访问系统，例如通过买主客户端。节点间使用因特网和内联网的TCP/IP来进行通信，参见图9-18。

第三个服务器是为银行自己准备的，这才是账单交付实际发生的地方（也就是说，资金在账户之间转账的地方）。

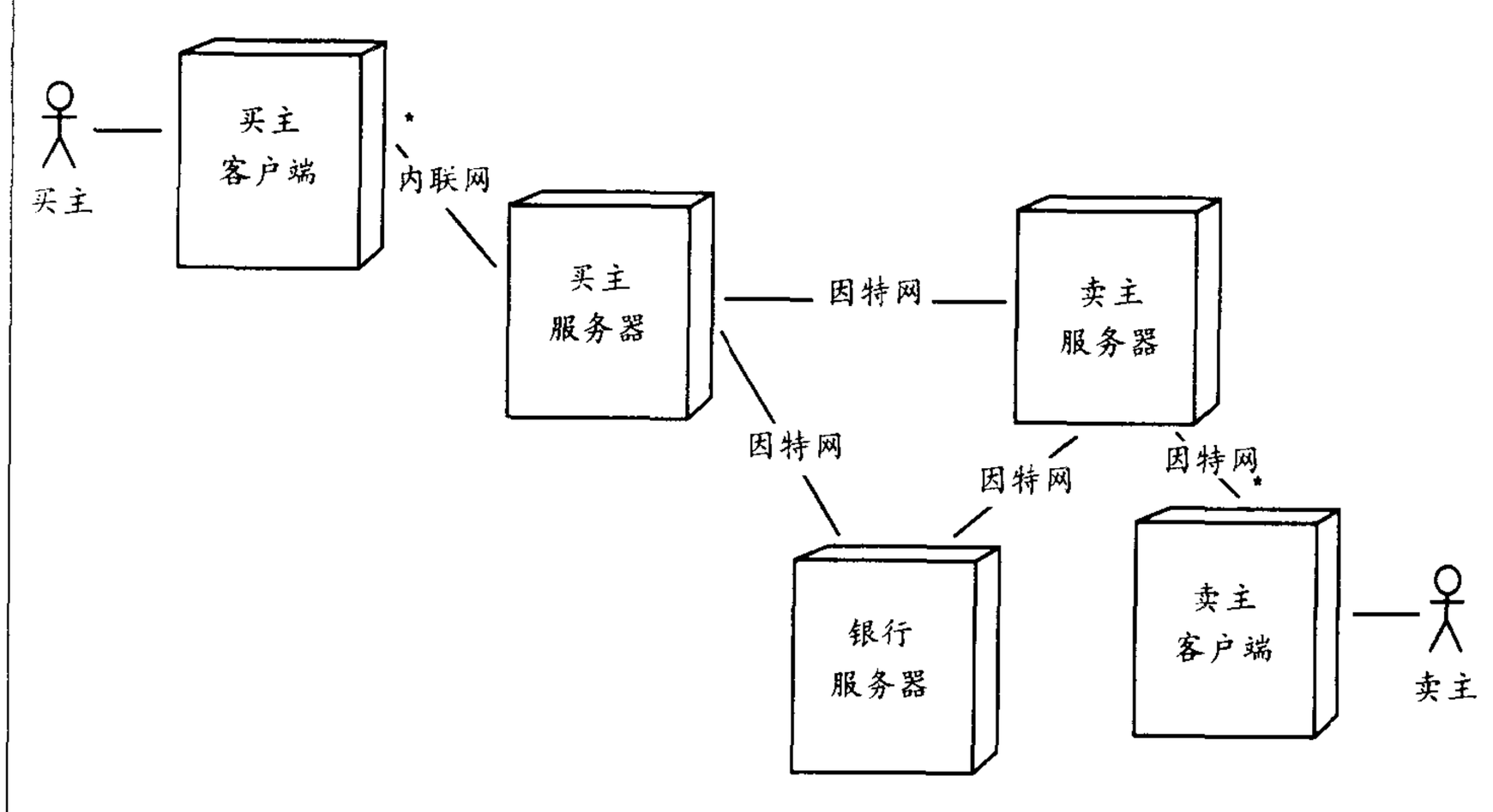


图9-18 Interbank系统的实施模型

每种网络配置，包括用于测试和模拟的特殊配置，都应该用单独的实施图来描述。通过给定的网络配置，就可以开始考虑如何在节点当中分布功能了（参见9.5.1.3.2节）。

9.5.1.2 识别子系统及其接口

子系统提供了一种将设计模型组织成可管理片的方法。它们既可以通过细分设计工作的方式来初步识别，也可以随着设计模型进化和“增长”成一个需要被分解的较大的结构时再去发现。

还需要说明的是，并不是所有的子系统都是由当前的项目组自己开发的，相反，一些子系统代表重用的产品和其他公司的现有资产。在设计模型中包括这样的子系统就可考察和评估重用的机会。

9.5.1.2.1 识别应用子系统

在这一步，我们将识别专用应用层和通用应用层中的子系统（也就是最高的两个层次中的子系统），参见图9-19。

如果在分析中找到适当的分析包分解，我们就能够尽可能地利用这些包去识别设计模

型中的相应的子系统。这对于服务包的处理尤其重要，这样就不用分解系统结构，而仅根据所提供的服务来识别相应的服务子系统。然而，这种对子系统的初步识别将在设计过程中进行少许改进，以处理与系统的设计、实现和实施有关的问题，我们将对此做简要的介绍。

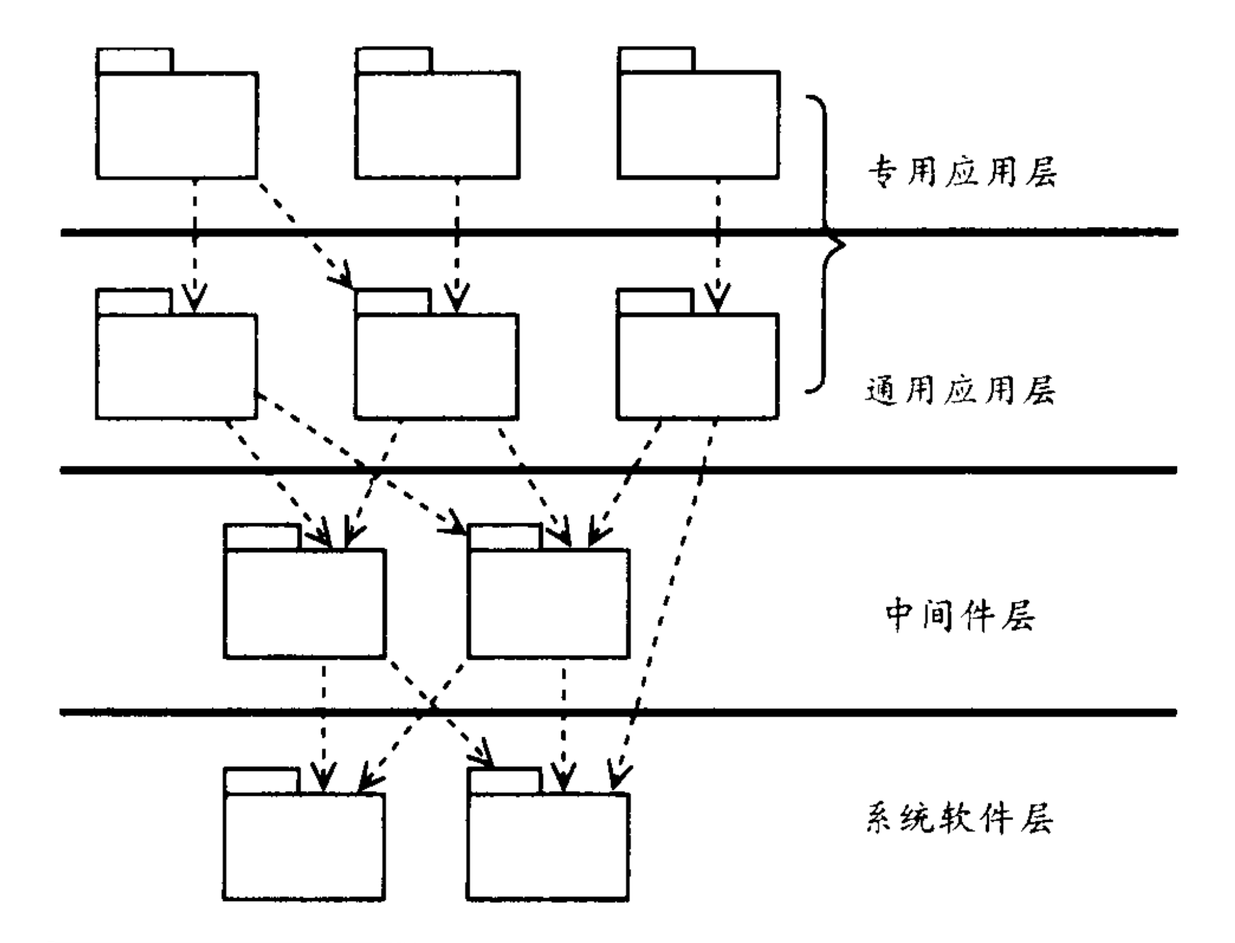


图9-19 专用应用层和通用应用层中的子系统

举例：从分析包识别设计子系统

分析模型中的“买主账单管理”包和“账户管理”包用于识别设计模型中相应的子系统（参见图9-20）。

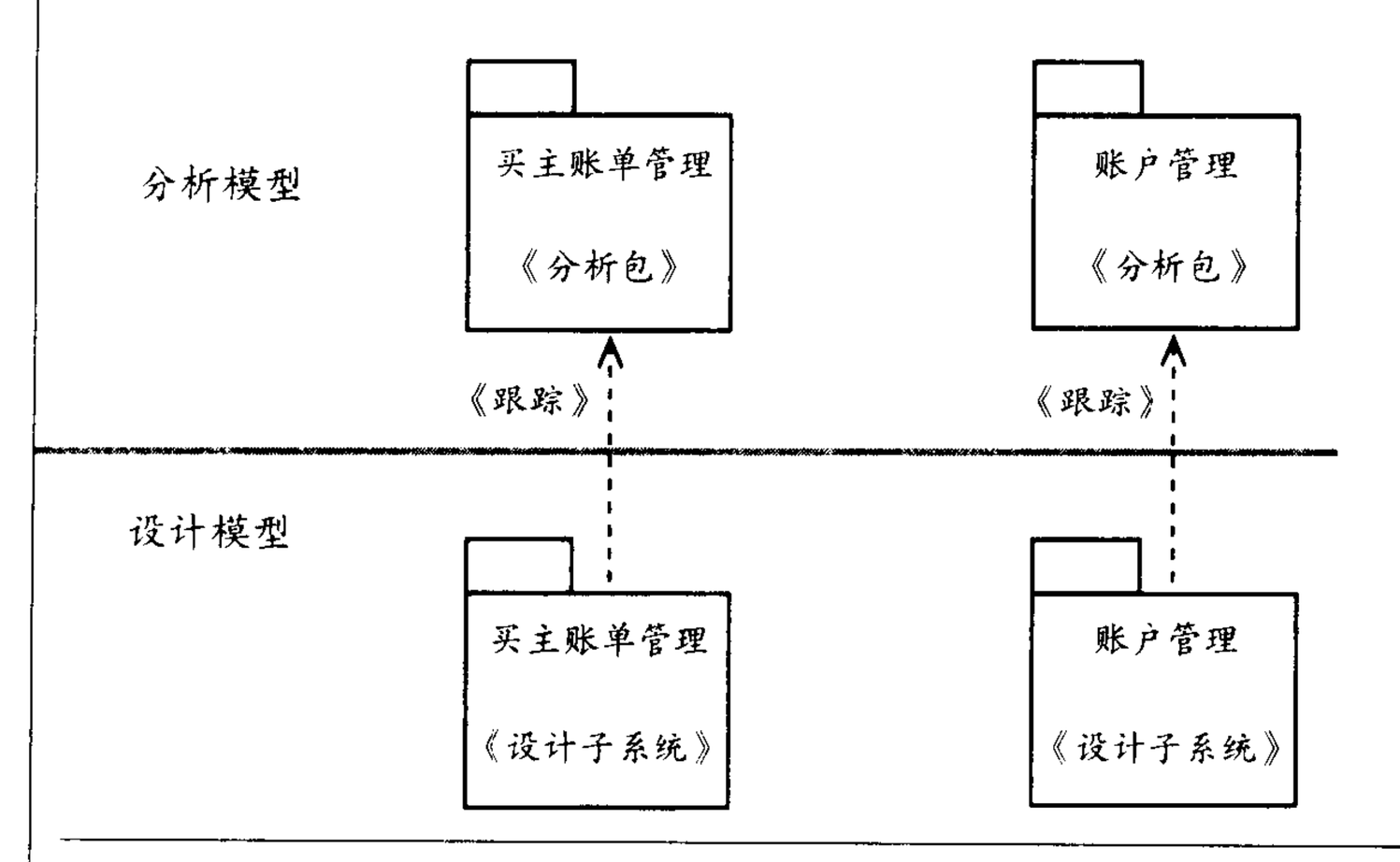


图9-20 寻找基于已有分析包的子系统

此外，位于分析模型中的“账户管理”包中的“账户”和“风险”服务包用于识别设计模型中相应的服务子系统（参见图9-21）。

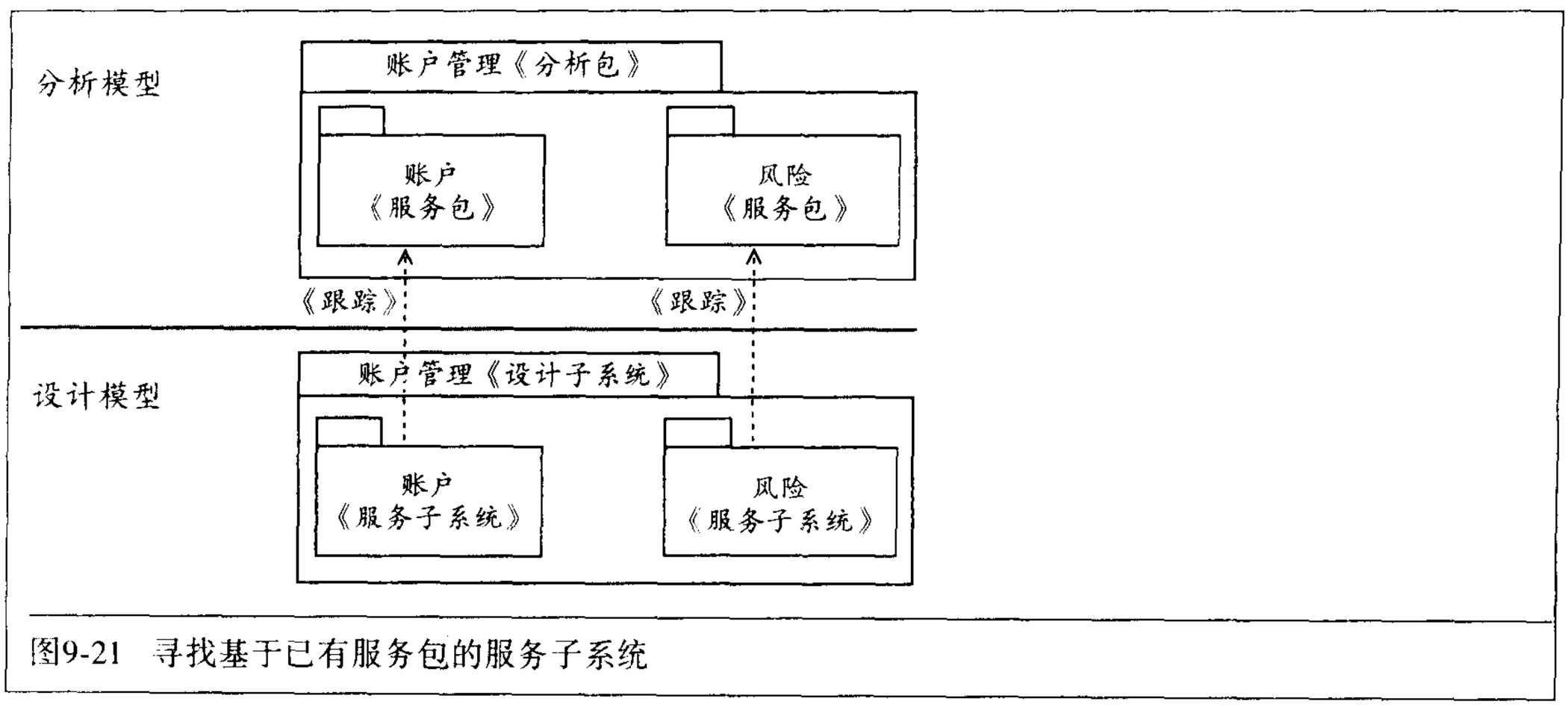


图9-21 寻找基于已有服务包的服务子系统

举例：精化子系统以处理共享功能

在分析了与账单相关的用况后，支付账单的服务包看起来好像属于“买主账单管理”子系统，因此，Interbank软件认为要在“买主账单管理”子系统中实现所有的支付账单的服务包。开发人员进而意识到，其他几个尚未设计的用况可以从一个通用支付服务子系统获取。因此，他们决定在一个名为“支付调度管理”的服务子系统中保留支付功能。这就意味着，“买主账单管理”子系统可以利用“支付调度管理”服务子系统中的支付调度功能。稍后，当预定的支付时间到期时，管理子系统就会通过“账户管理”子系统把款额从一个账户转移到另一个账户（参见图9-22）。

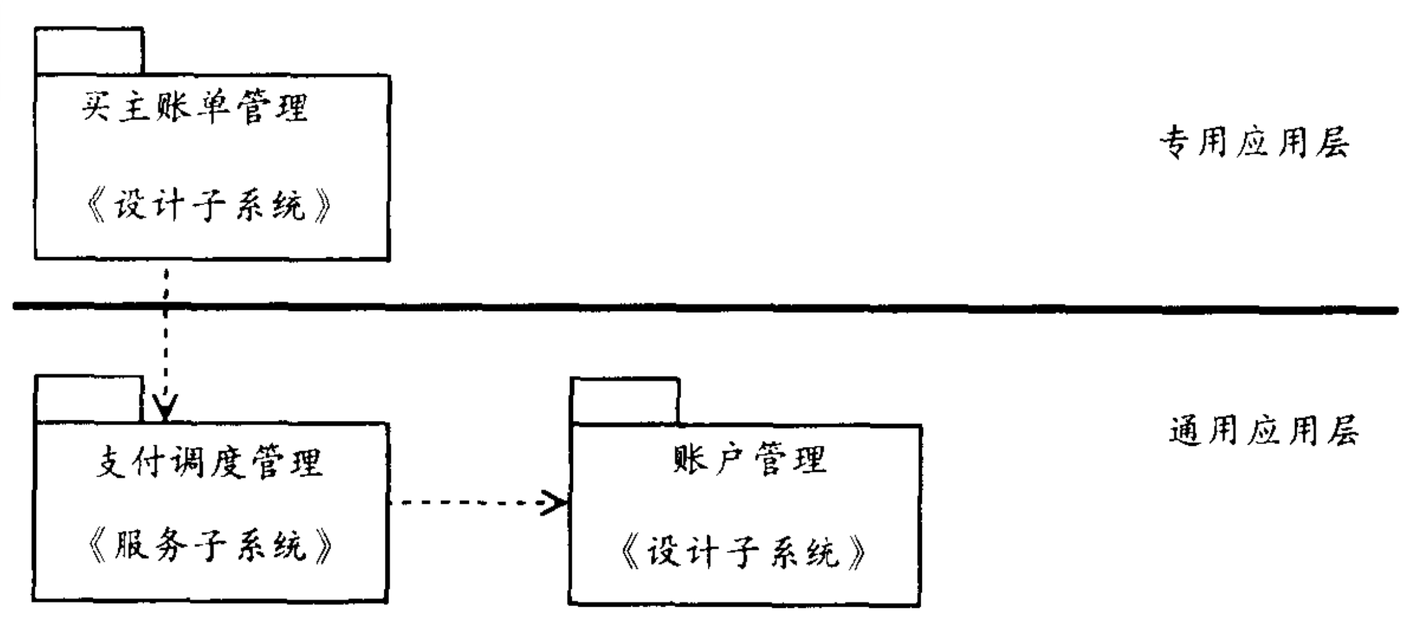


图9-22 在设计过程中，发现“支付调度管理”服务子系统以提供能够被多个用况实现所使用的通用服务

与分析模型中前面的分析包相比，在下面的情况下需要对这种初步的子系统分解做进一步的精化：

- 当前面的分析包中某部分映射到它本身的某个子系统时（例如，当发现这部分能被其他的几个子系统共享和使用时）。

- 当前面的分析包的某些部分通过可重用的软件产品来实现时。这样的功能可以分配到中间件或者系统软件子系统（参见9.5.1.2.2节）。
- 当前面的分析包不能恰当地反映工作的分解时。
- 当前面的分析包不能反映与遗留系统相结合时。这时，可将遗留子系统或者它的某部分打包成为一个单独的服务子系统。
- 当前面的分析包不能直接实施到某个节点时。这时，就需要将一些子系统做进一步的分解，使其成为更小的子系统，使得每个子系统可以分配到单个的节点上。然后，需要对这些更小的子系统进行精化，以使网络流量最小。

举例：在节点间分布一个子系统

在实施时，“买主账单管理”子系统需要分布到几个不同的节点中。为了处理这种情况，该子系统又进一步分解成三个子系统：“买主用户界面”、“支付请求管理”以及“账单管理”（参见图9-23）。基于这三个更小的子系统创建的构件就能够分别实施到“买主客户端”、“买主服务器端”和“卖主服务器端”的节点上。

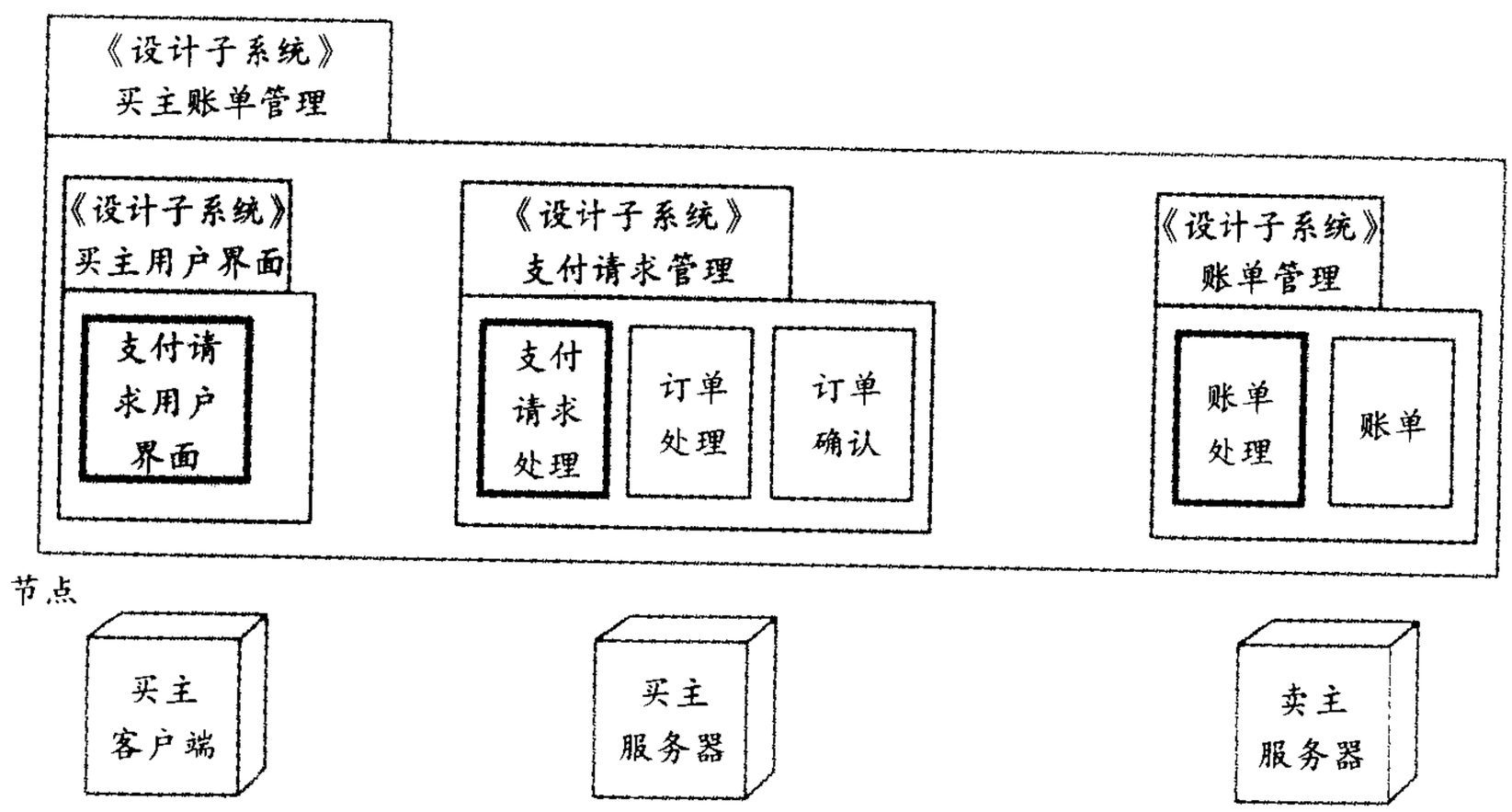


图9-23 一个子系统递归地分解成三个新的子系统以便处理实施问题

9.5.1.2.2 识别中间件和系统软件子系统

中间件和系统软件是一个系统的基础，因为所有的功能都是基于下列各种软件之上的：比如操作系统、数据库管理系统、通信软件、对象分布技术、图形用户界面设计工具、事务管理技术[3]（参见图9-24）。选择并集成所获取或构造的软件产品是初始和细化阶段所关注的两个要点。构架设计师要确认所选择的软件产品适合构架，并提供一种性能价格比合算的系统实现方案。

需要提醒的是：当获取到中间件和系统软件后，就很难控制对它们的改进。此时重要的是要维持适当的行动自由，并避免步入对那些项目很难影响的产品和卖主完全依赖的困境。要尽力减少对已获取的产品的依赖，并因此减少使用它们会产生的风险，以防止它们在将来作种种修改，在需要的时候还可以选择不同的卖主。

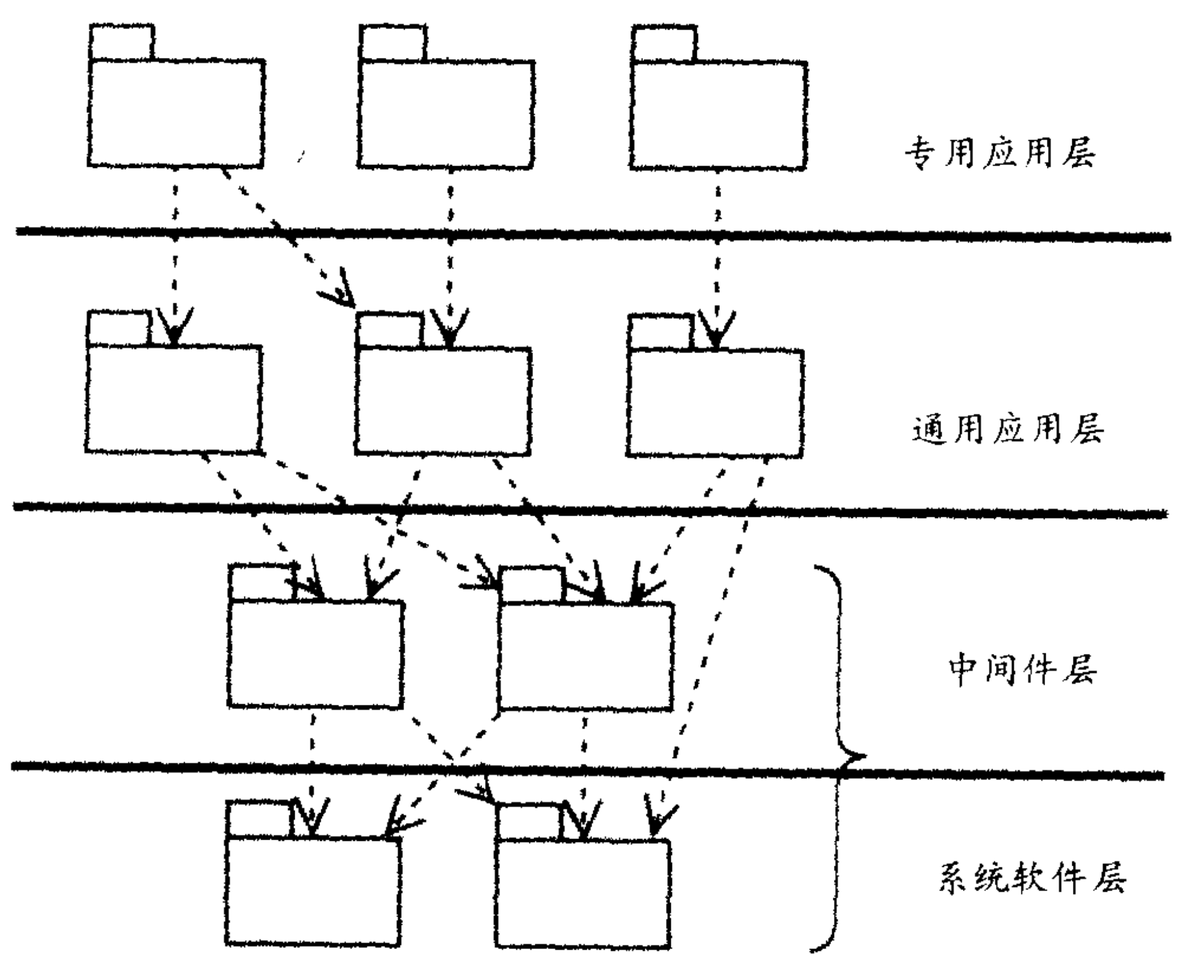


图9-24 中间件层和系统软件层中的子系统封装了已获取的软件产品

控制依赖性的一种方法是把每个已获取的软件产品看成是一个与系统其他部分有清晰接口的独立的子系统。例如，如果需要实现一种透明的对象分布机制，就应该通过定义一个到能够实现这种机制的子系统的严格的接口来实现这种机制。这就保证了在不同的产品间作出选择的自由度，从而消除了更新系统所需要的额外花费。

举例：建造中间层时使用Java

由Interbank软件公司开发的几个应用子系统的实现需要运行在不同类型的机器上，如PC和UNIX工作站，因此需要跨平台互操作。Interbank软件公司选择使用中间件来实现这种互操作特性，这些中间件包括：Java的抽象视窗调试工具（AWT）、远程消息请求（RMI）以及Applet包。在图9-25中将这些Java包表示成一些子系统。这些子系统建立在Java虚拟机上，Java虚拟机是安装在一台计算机上的能够执行Java代码的解释器。在我们的示例中，用一种Web 浏览器来装载获取Applet的Web页面。

在最底层，Interbank软件公司将系统软件建造在支持因特网通信的TCP/IP协议的基础上。

Java Applet子系统是一个允许Interbank创建Java *applet*的中间件。

每个用户界面窗口都使用Java Applet类和其他的用户界面类（如由AWT子系统提供的Lists）来设计。Java抽象视窗调试工具包（java.awt）用于创建平台独立的用户界面。AWT包包括多种类（如域、滚动条和检查框等类）。

Java RMI是汇总在标准Java类库中的用于对象分布的一种机制。在这里选择了RMI处理对象分布是为了简化示例并避免几种技术和语言的混淆。CORBA 或者 ActiveX/DCOM解决方案也可以用于对象分布的实现。

图9-25阐述了如何将Java服务组织到中间件层的子系统中。每个子系统包含一些类，将这

些类设计成能互相协同地提供一定的服务。类似地，如电子表格、多媒体、图像处理、安全管理、持续性、分布特性、推理机、过程和并行处理支持等ActiveX构件也可以被表示为子系统。为标准的数据类型或基础类创建一个单独的子系统供所有其他的子系统输入和引用（例如，java.lang提供有众多的Java核心类，如布尔型、整型、浮点型），也常常是一种实际的做法。

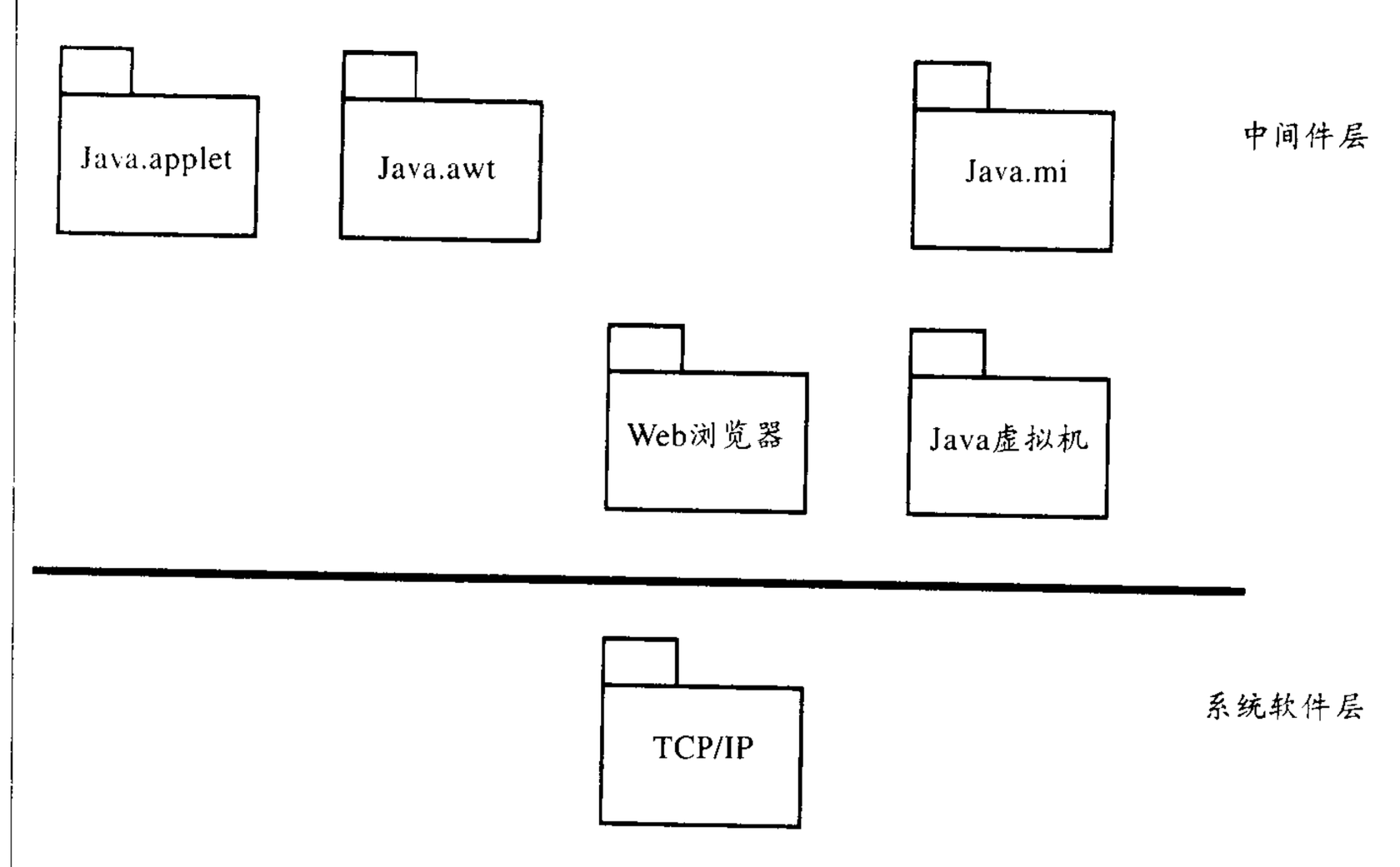


图9-25 Java中间件提供了与平台无关的图形用户界面(java.awt) 以及分布式对象计算(java.rmi)。该图没有显示任何子系统之间的依赖关系（这些将在图9-26中给出）

9.5.1.2.3 定义子系统间的依赖关系

如果子系统的内容相互有关，就应该定义子系统之间的依赖关系。依赖的方向与关系（导航）的方向相同。如果依赖关系需要在子系统的内容搞清楚前就勾画出来，我们就把相应于该设计子系统的分析包之间的依赖关系看成是这种依赖关系，这种依赖关系可能与设计模型中的相似关系类似。如果使用了子系统之间的接口，则依赖关系就应该指向接口，而不应该指向子系统本身（参见9.5.1.2.4节）。

举例：依赖关系与层

图9-26描述了Interbank系统中的子系统以及其中部分子系统之间的（初始）依赖关系。

注意，某些依赖关系（尤其是在中间件层和系统软件层的依赖关系）可能或多或少地隐含在实现环境中（Java即如此）。但当依赖关系对系统构架有影响，尤其是当它们“穿越”层次边界时，在设计模型中对它们加以清楚地标记，并像图9-26那样在类图里对它们加以描述就很有用。用况工程师和构件工程师在设计用况、类和子系统时都会查阅这样的图。

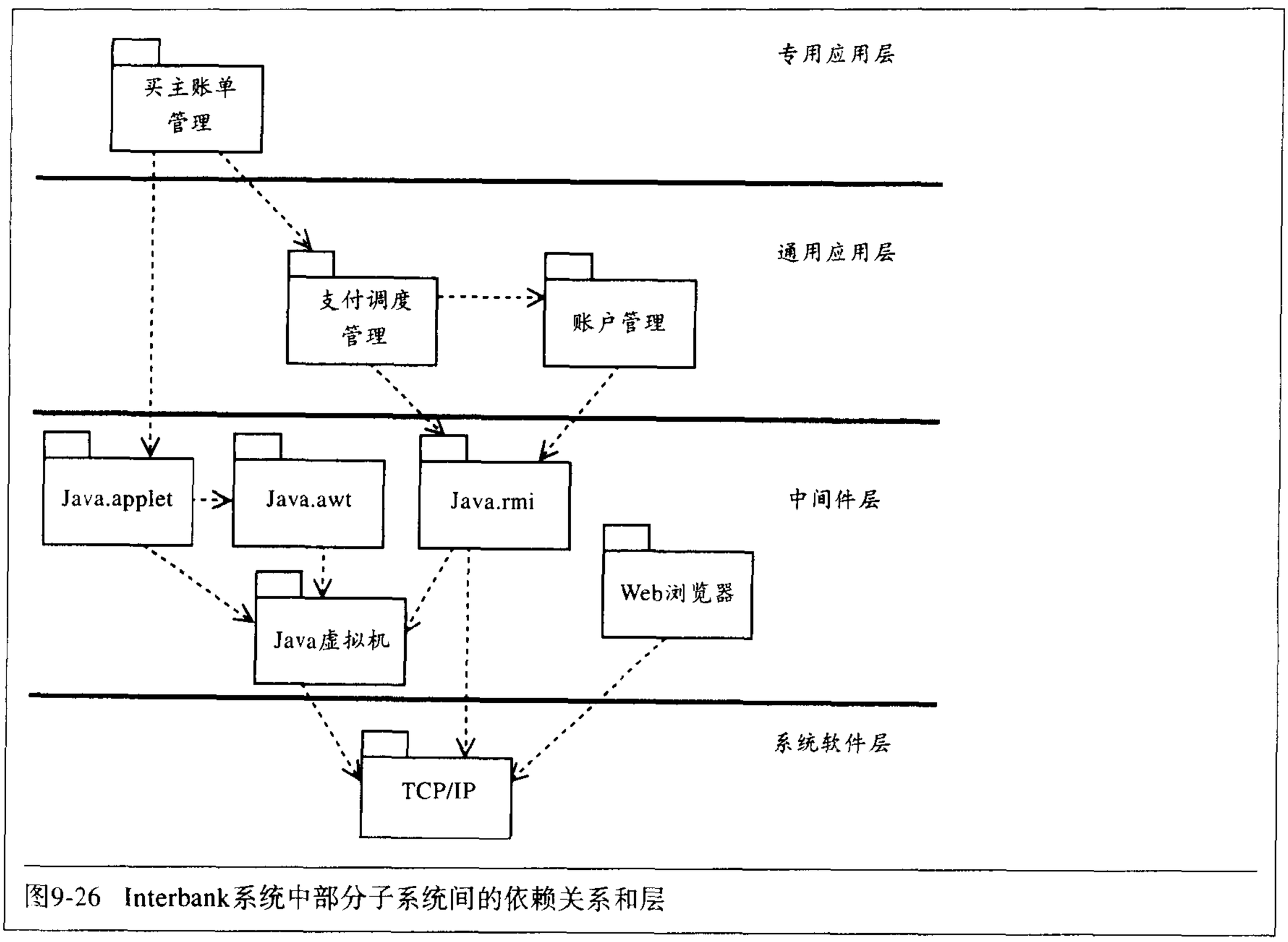


图9-26 Interbank系统中部分子系统间的依赖关系和层

9.5.1.2.4 识别子系统接口

由子系统提供的接口定义了来自外部子系统的访问操作。这些接口由子系统内部的类提供，或由子系统内部的其他子系统（递归地）提供。

在弄清楚子系统的内容之前，为了初步勾画子系统接口，我们需要从前一步发现的子系统间的依赖关系入手。当有一个依赖关系直接指向某个子系统时，就有可能表示这个子系统需要提供一个接口。此外，如果存在一个可以从子系统跟踪的分析包，则任何从该分析包的外部引用的分析类都隐含地需要由该子系统提供相应的接口（如下面示例所示）。

举例：寻找基于分析模型的候选接口

“账户”服务包中含有一个名为“账户转账”的分析类，它由该包外的类引用。于是我们就识别出一个初步的接口称之为“转账”，它由设计模型中相应的“账户”服务子系统提供（参见图9-27）。

使用这种方法，我们在设计模型的最高两层初步识别出图9-28所示的接口。

“账户管理”子系统提供“转账”接口用于实现账户间资金的转移。该接口与“账户管理”内的“账户”服务子系统所提供的是同一接口（参见前一个例子）。“支付调度管理”子系统提供“支付请求”接口用于对支付款进行调度。买主账单管理子系统提供“支付调度接收”接口用于从卖主接收新的账单。当一个新账单被送给买主时，“给买主开单”用况就使用该接口。

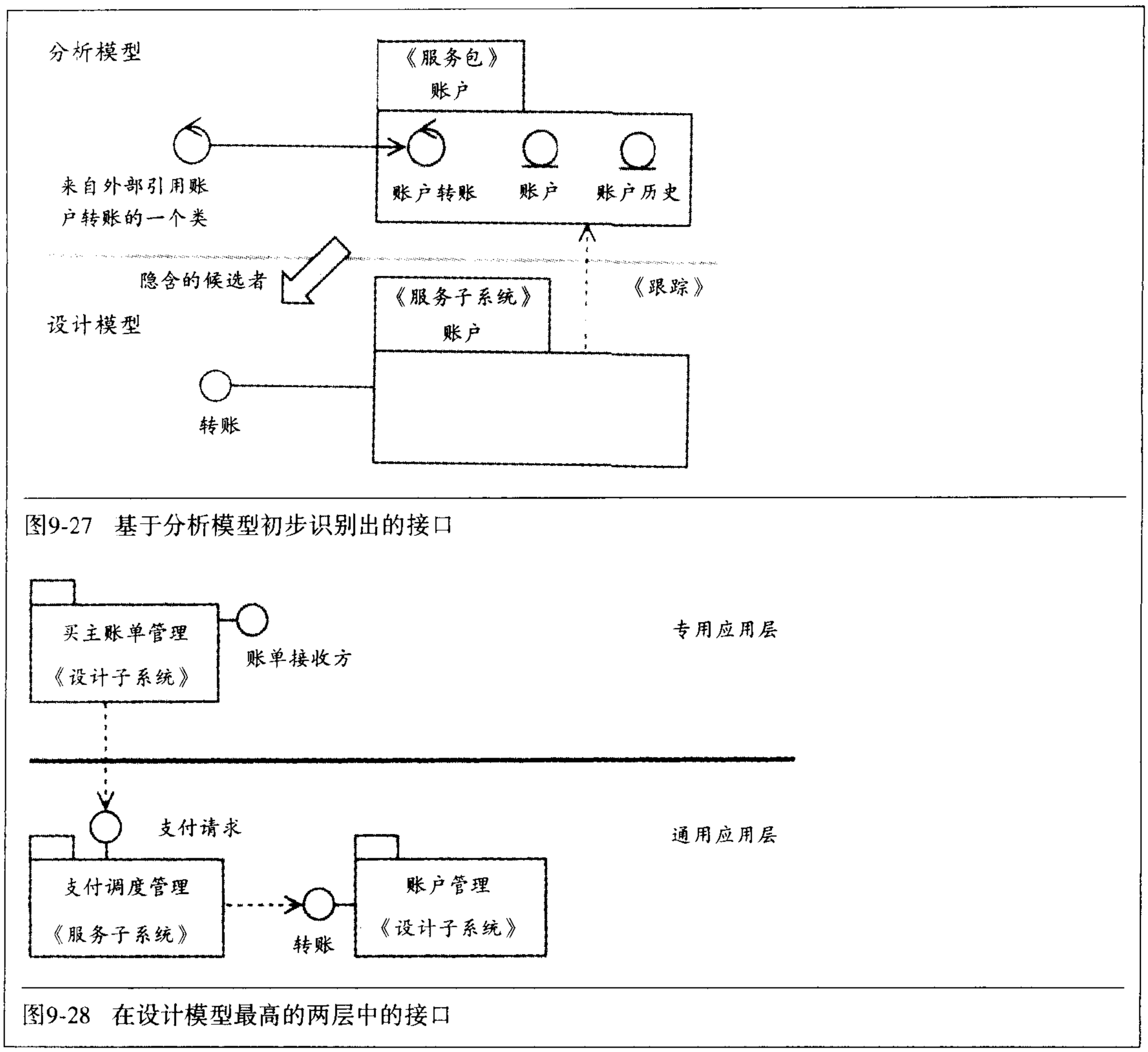


图9-27 基于分析模型初步识别出的接口

图9-28 在设计模型最高的两层中的接口

注意，这里对接口的识别使我们对前面9.5.1.2.3节所描述的子系统间的依赖关系进行了精化；现在依赖关系就体现在接口上，而不是体现在子系统上。

关于最低两层（也就是中间件和系统软件层）的接口的识别，就显得相对简单一些。因为那些层次的子系统封装了软件产品，而这些产品常常已经有某种形式的预定义接口。

然而，仅仅识别出接口是不够的，还必须识别每个接口需要定义的操作。这通过在子系统和接口的基础上进行用况设计来实现，具体情况请参见后面的用况设计活动，如9.5.2.3节和9.5.2.4节所述。在用况设计时，将陈述需要由接口定义的操作需求。9.5.4.2节描绘了怎样为每个接口处理和综合来自不同的用况实现的需求。

9.5.1.3 识别对构架有重要意义的设计类

在软件生命周期的早期识别对构架有重要意义的设计类来启动设计工作，常常是很实际的。然而，绝大多数设计类要在类设计活动中设计类时才能识别出来，并在用况设计活动结果的基

础上精化（参见9.5.2节和 9.5.3节）。基于这个原因，开发人员应该注意，在这一阶段既不要识别太多的类，也不要陷入太多的细节，只要初步地勾画出对构架有重要意义的类也就足够了（参见9.3.6节）。否则，在后面使用用况来调整设计类（即调整参与用况实现的类）时，大量工作将需要重做。不参与用况实现的设计类是多余的。

9.5.1.3.1 根据分析类识别设计类

根据在分析过程中发现的对构架有重要意义的分析类，可以初步地勾画出一些设计类。而且，这些分析类间的关系可以用于识别相应的设计类间的临时的关系集。

举例：从分析类勾画设计类

从分析模型中的“账单”实体类初步勾画出“账单”设计类（参见图9-29）

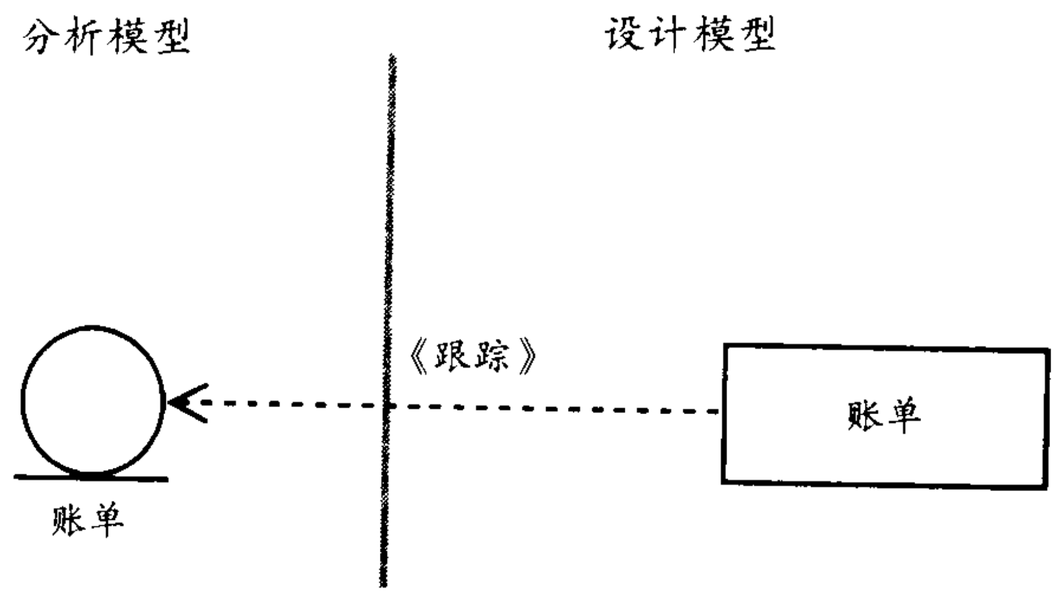


图9-29 从“账单”实体类初步勾画出“账单”设计类

9.5.1.3.2 识别主动类

构架设计师还通过考虑系统的并发性需求，识别该系统所需要的主动类。例如：

- 不同的参与者与系统交互时，会有性能、吞吐量、有效性等需求。比如，如果某个参与者对响应时间有很高的要求，那么就需要有一个专用的主动对象来管理从该参与者获得输入和向该参与者提供输出；该主动对象不会因为其他主动对象负载太重而终止活动（前提是有足够的处理器能力和内存资源）。
- 在节点上的系统分布。主动对象需要支持在几个不同节点上的分布，比如，每个节点可能至少需要一个主动对象，并且需要单独的主动对象来处理节点间的相互通信。
- 其他的需求，例如对系统的启动和终止、活跃性、死锁预防、饥饿预防、节点重构、连接容量等需求。

通过对其主动对象的生命周期以及主动对象如何通信、同步和信息共享等的考虑来勾画每个主动类。然后把主动对象分布到实施模型的节点上。当把主动对象分布到具体节点时，需要考虑节点的容量（例如它们的处理器和内存大小）以及连接特征（例如带宽和有效性）。在进行这样的分布时，一条基本的准则是：网络通信量对系统所需的计算资源（包括硬件和软件）有实质的影响，因此需要谨慎控制。这可能对设计模型有重要的影响。

为了初步勾画出主动类，可以把分析的结果和实施模型作为输入，然后通过主动类把相应的分析类（或其一部分）的设计映射到节点上。

举例：利用分析类勾画主动类

如前所述，Interbank系统需要分布到“买主”客户端、“买主”服务器端和“银行”服务器端等节点上。此外，我们已经识别出“支付请求用户界面”、“订单处理”、“订单确认”和“账单”等分析类（参见图9-30所示）。现在，我们发现某个买主对“订单确认”和“订单处理”分析类所提供的功能感兴趣，但这些功能所需要的计算能力比“买主”客户端节点所能提供的计算能力要更高。因此，需要把这两个分析类的主要部分分布到“买主”服务器端，并由“买主”服务器端节点上的一个单独的主动类（“支付请求处理”）管理。

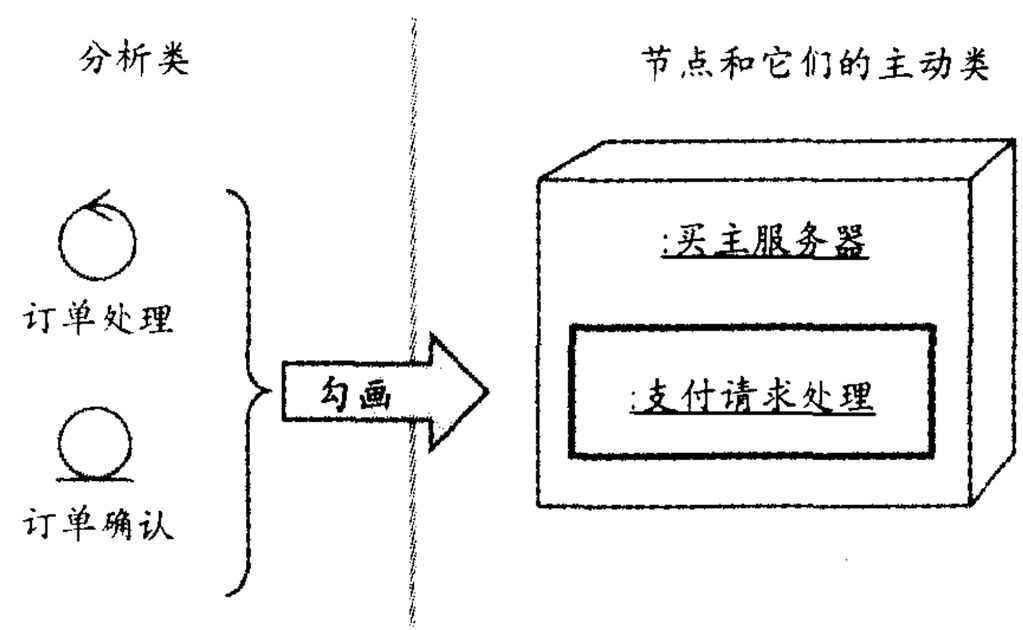


图9-30 通过对分析类“订单处理”和“订单确认”以及“买主”服务器端节点进行综合考虑，勾画出来主动类“支付请求处理”。这就决定了当对分析类进行设计时，相应的设计类的主要部分要分配到“买主”服务器端节点。然后，在这个节点上识别“支付请求处理”类以便对那些设计类的控制线程进行封装

另一种勾画主动类的可行方案是利用早前识别的子系统，并通过识别子系统内的一个主动类而将该子系统全部分配到某个特定的节点上（参见9.5.1.2.1节）。请回忆一下，要实现这种可能性，可能需要对子系统做进一步分解。

任何表示重要处理的主动类都可以作为一个在实现过程中被识别的可执行构件的候选。因此，在这一步从主动类到节点的分配是在实现过程中从可执行构件到节点的分配的重要输入。此外，当一个子系统全部分配到某个节点时，这个子系统的所有组成部分常常都是属于分配到这个节点的可执行构件。

9.5.1.4 识别通用的设计机制

在这一步，研究共性需求，例如在**用况实现—分析**和分析类的分析过程中识别的特定需求，并决定如何通过给定的可行的设计和实现技术来处理这些需求。研究的结果是一个通用的设计机制集，它们表明自己是设计类、协作或者甚至子系统，类似于[1]中所描述的内容。

需要做处理的需求通常与下面的问题有关：

- 持久性
- 透明的对象分布
- 安全特征
- 错误检测与恢复

• 事务管理

在某些情况下，并不能预先知道具体的机制，而是在探究用况实现和设计类时才能得到具体的机制。

举例：一种用于透明的对象分布的设计机制

某些对象（如“账单”对象）需要被几个节点访问，因此必须把它们设计成一个分布系统。Interbank软件公司决定通过使每个待分布的类成为抽象的Java类（即java.rmi.UnicastRemoteObject，它支持远程消息请求RMI）的一个子类来实现这样的对象分布，如图9-31所示。RMI是Java用于实现透明的对象分布的技术（也就是说，对象按照这样的方式（即客户对象不必知道它驻留在哪里）来分布）。

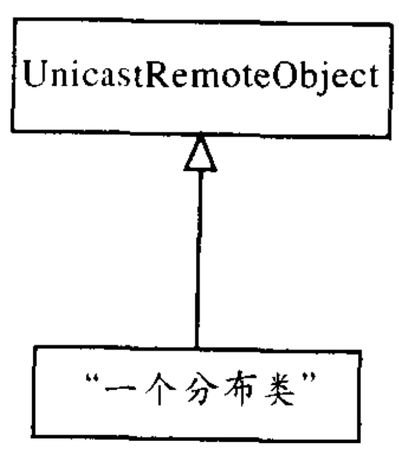


图9-31 任何需要被分布的类应该是UnicastRemoteObject的子类

举例：用于处理持久性的设计机制

某些对象（例如“账单”和“订单”对象）需要持久存在。要处理这样的要求，Interbank软件公司可以使用对象数据库管理系统、关系数据库管理系统或者普通的二进制文件。哪一种方法最好，取决于对象需要的访问方式和更新方式，以及哪一种方法在未来更容易实现和升级。关系数据库通常提供较好的表格数据处理性能，而对象数据库则提供较好的复杂对象结构处理性能。关系数据库管理系统比对象数据库系统更成熟，但采用关系数据库管理系统建造系统的代价较高，而且在将来当对象数据库足够成熟时肯定需要更新成对象数据库。无论选择哪种方案，构架设计师需要把处理持久性问题的设计机制编制成文档。

最后一个示例阐明每种机制通常有几种不同的处理方法，而且每种方法都有其优点和缺点。如果没有哪一种单一机制能适应全部情况，就可能需要提供多种机制，并为每种情况选择最合适的一种机制。

构架设计师还应该识别共性的协作关系，把它作为工作模式，并在设计模型内由几个用况实现所使用。

举例：用于几个用况实现中的共性的协作关系

在处理用况及其实现的工作过程中，构架设计师识别了一种模式，其中“交易”对象（如“订单”或“账单”）由某一个参与者创建并传送给另一个参与者：

• 当“买主”决定从卖主订购一些产品或服务时，买主就调用“产品/服务订购”用况，

该用况允许买主对订单加以描述，并电传给卖主。

- 当“卖主”决定要买主支付账单时，就调用“给买主开单”用况，给买主电传一张账单。

如图9-32中的协作图所示的是一个可以通过共性的协作关系表达的共性行为。

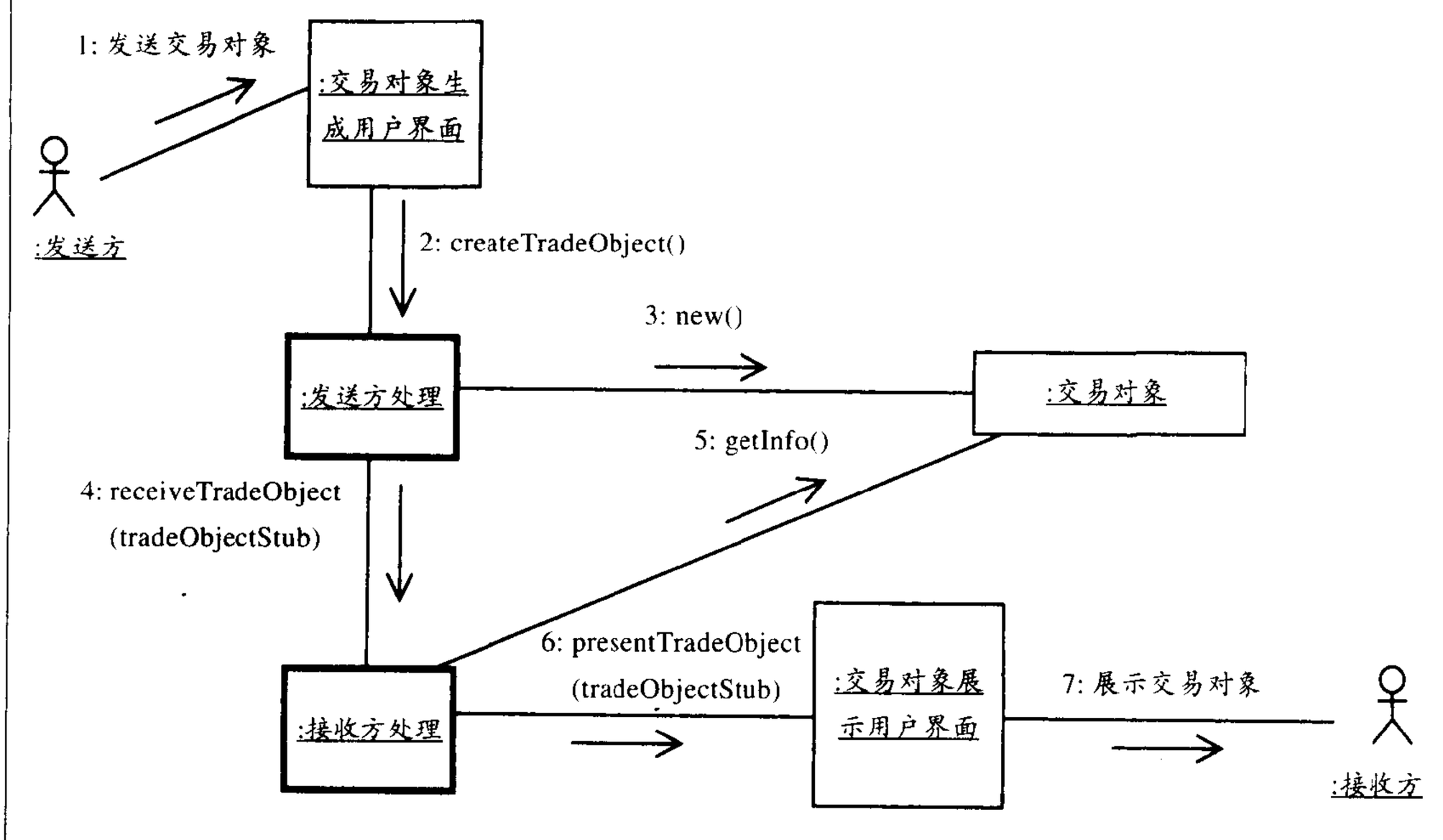


图9-32 举例说明共性协作关系的协作图，用于创建、发送、接收和展示交易对象

首先，“交易对象生成用户界面”从“发送方”参与者处接收信息，作为创建“交易”对象（1）的输入（括号中的数字对应图9-32中的相应数字）。然后，“交易对象生成用户界面”要求“发送方处理”创建一个“交易”对象（2）。“发送方处理”要求相应的“交易对象”类创建一个实例（3）。“账单处理”向“接收方处理”提交“交易”对象的参考信息（4）。“接收方处理”在需要时可以查询“交易”对象以获得更多的信息（5）。“接收方处理”然后把用

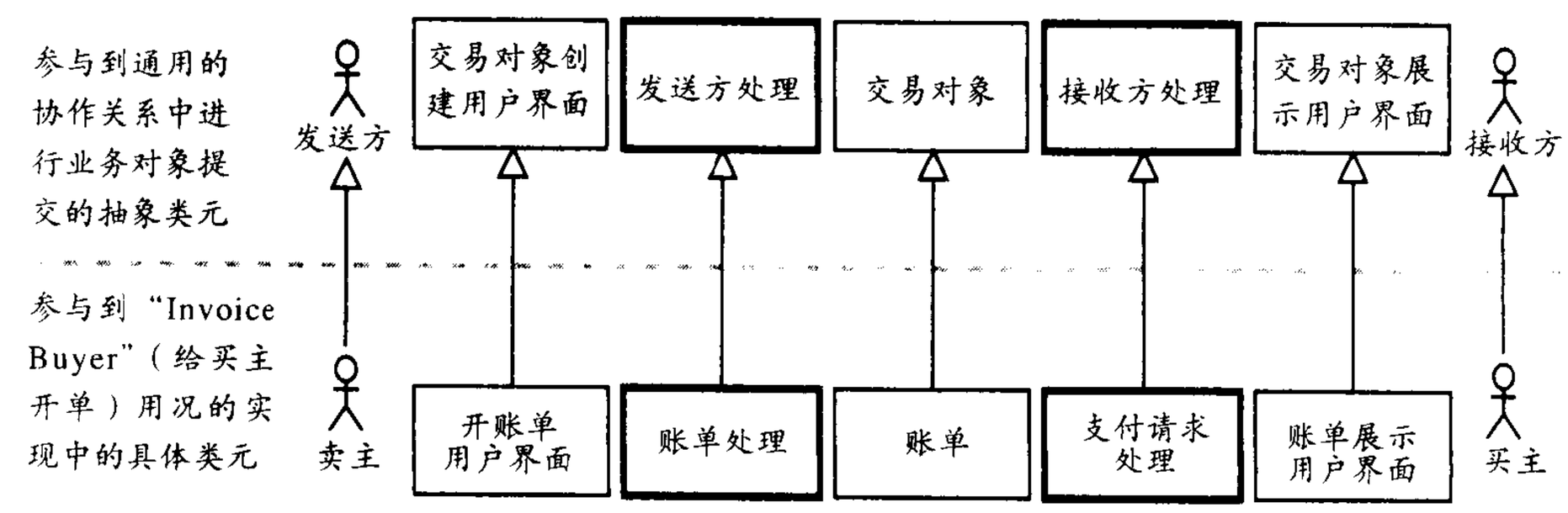


图9-33 参与某个共性协作关系的抽象类元由参与特定用况实现的具体类元加以子类型化

于展示的“交易”对象发送给“交易对象展示用户界面”（6），于是就把“交易”对象展示给“接收方”参与者。

然后，例如在实现“给买主开单”用况时，参与到这个共性协作关系中的每一个抽象类元就可被子类型成具体类元，如图9-33所示。

这样，“给买主开单”用况的实现就只需要参考该共性协作关系，但是，如果抽象类元的（虚拟）操作是通过经子类型化后的具体类元的方法来实现，就不需要复制或作专门的研究。

在实现“产品/服务订购”用况时可以遵循同样的方法。

请注意，使用泛化并不是使用共性协作关系的唯一形式。例如，参数化协作关系（即参数化的类）的模式也是共性的，并且可以通过给具体的类定义参数的方法来使用。

在细化阶段，应该识别和设计绝大多数的泛化机制。通过这样细致的工作，构架设计师常常能够设计出一套机制，它能解决大多数复杂的设计问题，并使大多数用况变得相当简单，且直接面向构造阶段的实现。与软件产品获取相关的机制就自然成为中间件层的首选，其他机制很有可能定位在通用应用层。

9.5.2 活动：设计一个用况

设计一个用况的目的是为了：

- 识别设计类或子系统，其实例需要去执行用况的事件流。
- 把用况的行为分布到有交互作用的设计对象或所参与的子系统。
- 定义对设计对象或子系统及其接口的操作需求。
- 为用况捕获实现性需求。

9.5.2.1 识别所参与的设计类

在这一步，我们将识别实现用况所需要的设计类（参见图9-34），步骤如下：

- 研究参与相应的**用况实现—分析**中的分析类，识别那些由构件工程师在类设计过程中创建的或者由构架设计师在构架设计过程中创建的分析类所跟踪对应的设计类。
- 研究相应的**用况实现—分析**的特殊需求，识别那些由构架设计师在构架设计过程（即通用机制）中所发现的或者由构件工程师在类设计过程中所发现的实现特殊需求的设计类。
- 最终应该识别所需要的类，并且应该指定一些构件工程师来负责执行这一任务。
- 如果一些针对特定用况设计的设计类仍然没有找到，用况工程师应该将该信息通报构架设计师或者构件工程师。应该识别所需要的类，并应指定一个构件工程师来负责执行这一任务。

收集类图中参与某个用况实现的设计类。使用这个类图来表示在用况实现中所使用到的关系。

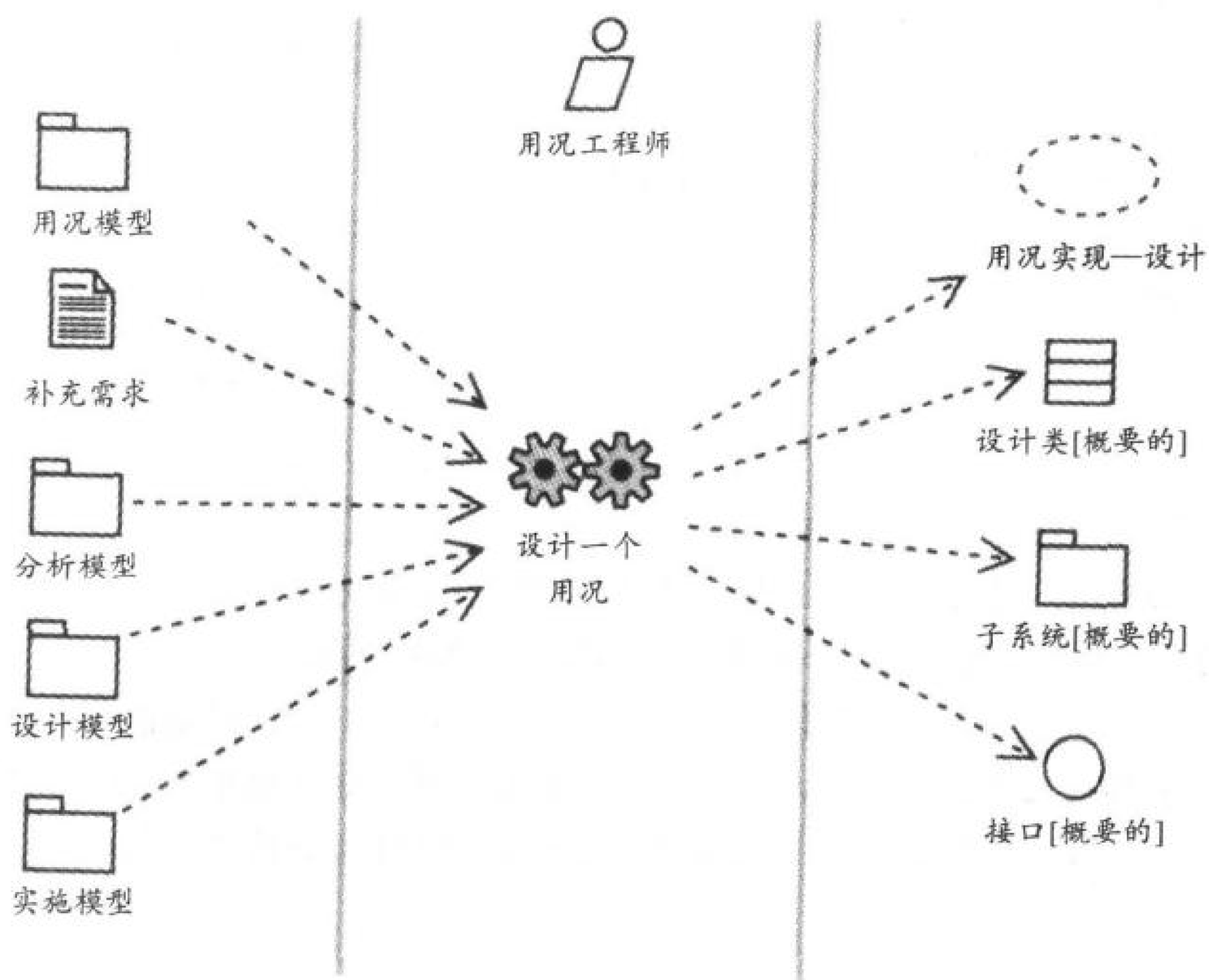


图9-34 用况设计的输入和结果。分析模型中相应的**用况实现—分析**是这个活动的基本输入

举例：参与“支付账单”用况实现的类

图9-35所示的是一个类图，它描述参与“支付账单”用况实现的类及其相互之间的关联。

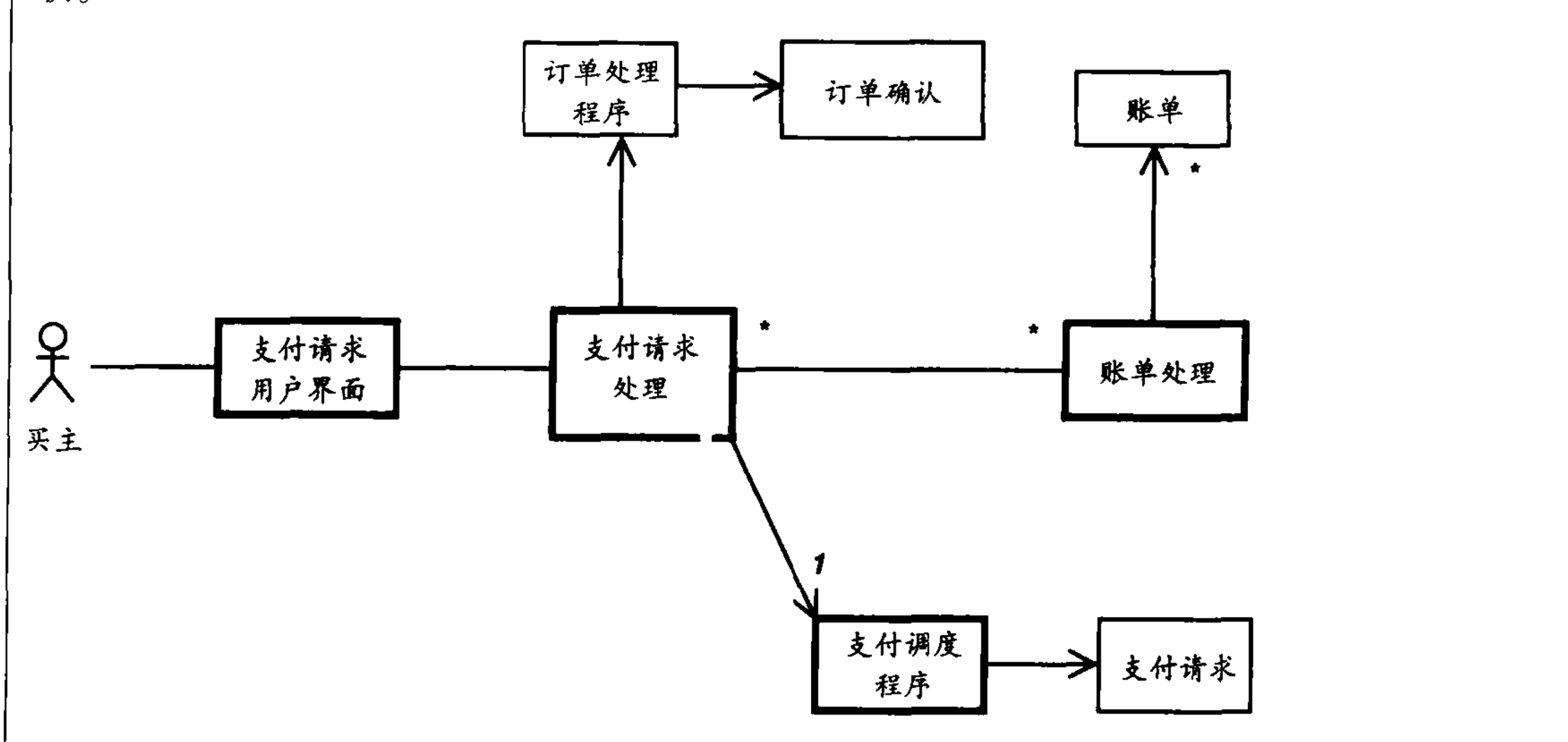


图9-35 参与“支付账单”用况实现的类及其关联。图中，主动类的边框较粗

某些主动类（像“支付请求处理”和“账单处理”）通过从发送者到接收者（如账单从卖主到买主）的不同节点之间传递业务对象，来保证Interbank系统的运行。

9.5.2.2 描述设计对象间的交互

当有了实现用况所需要的设计类的轮廓之后，就需要详细描述相应的设计对象是如何交互的。这可以通过使用包含参与的参与者实例、设计对象以及它们之间的消息传递的顺序图来实现。如果用况拥有不同且独特的流或子流，那么为每个流创建一个顺序图常常是值得的。这可以使用况的实现更加清晰，并且使得用顺序图来表示通用的和可重用的交互成为可能。

首先要做的工作是研究相应的**用况实现—分析**。尽管增加了许多新的设计对象，**用况实现—分析**仍可用于获取设计对象间所需的消息序列的轮廓。在某些情况下，甚至可以尝试将**用况实现—分析**的协作图转换成相应的顺序图的初步轮廓。

顺序图的创建过程是：从用况流的初始点开始，逐个分析改用况流上的每一步，并决定哪些设计对象和参与者实例间的交互需要实现。在绝大多数情况下，对象都可以很自然地在用况实现中找到各自在交互序列中的位置。下面是在创建这样一个顺序图时应该注意的几点：

- 用况由从参与者实例到设计对象传递的消息所调用。
- 对于前面所识别出来的每个设计类，应该至少有一个设计对象参与到某个顺序图中。
- 消息在**对象生命线**（参见附录A）间传递以便实现用况。一个消息在被负责对象类接收的构件工程师识别后，可以拥有一个临时的名称，这个名称将是某个操作的名称。
- 图中的序列应该是首先要关注的，因为在用况实现时**用况实现—设计**是主要的输入。了解对象间消息传送在时间上的先后顺序是非常重要的。
- 利用标签和**事件流—设计**来补充顺序图。
- 顺序图应该处理要实现的用况的所有关系。例如，如果用况A通过泛化关系将另一个用况B特化，那么实现用况A 的顺序图就可能需要参考用况B的实现（例如参考一个顺序图）。需要说明的是，这种参考关系也很可能发生在相应的**用况实现—分析**中。

举例：用于“支付账单”用况第一部分的顺序图

图9-36所示的是用于“支付账单”用况第一部分的顺序图。

补充顺序图的**事件流—设计**的描述可以如下所述：

买主通过“支付请求用户界面”applet和“支付请求处理”应用程序来使用系统，浏览收到的账单。“支付请求处理”利用“订单处理程序”按照有关的订单确认信息来核对账单，然后“支付请求用户界面”向买主显示账单的清单。

买主通过“支付请求用户界面”选择某一账单并确定支付时间，“支付请求用户界面”将这一请求传递给“支付请求处理”。“支付请求处理”要求“支付调度程序”确定账单支付时间，然后“支付调度程序”产生一个支付请求，“支付请求处理”应用程序就调用“账单处理”应用程序将账单的状态置为“已确定支付时间”状态。

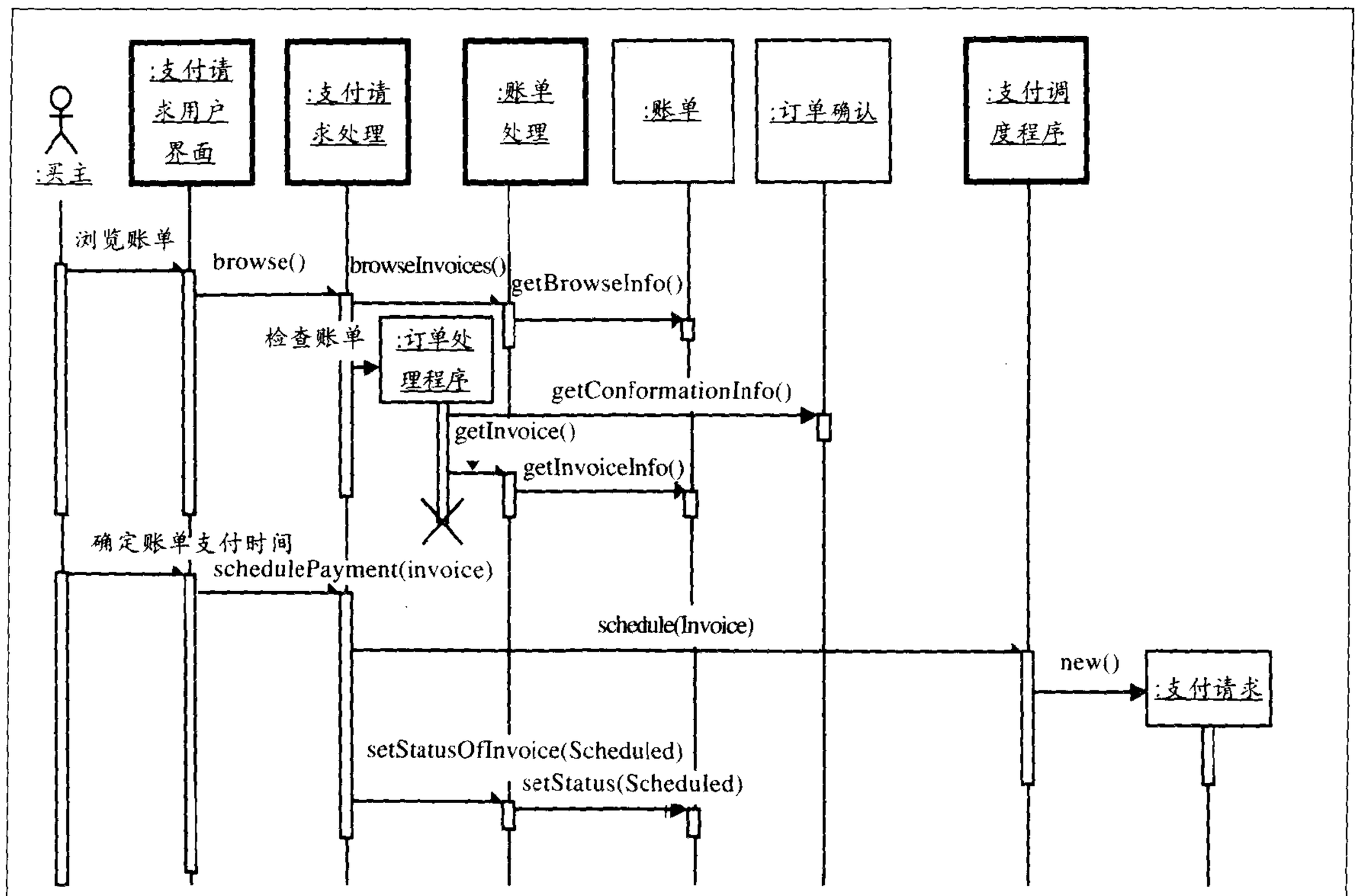


图9-36 用于完成“支付账单”用况实现的一部分的设计对象的顺序图

当对交互图进行细化时，很可能会发现用况的新的可选路径。这样的路径可以在该图中用标签或者在它们自己的交互图中作出阐述。当加入更多的信息时，用况工程师将经常发现新的异常，这些异常是在需求捕获或分析中无法考虑到的。这些例外的类型包括：

- 当节点或者连接停止工作时的超时处理。
- 人或者机器等参与者可能提供的错误输入。
- 由中间件、系统软件或者硬件产生的出错消息。

9.5.2.3 识别参与的子系统和接口

迄今为止，我们已经将用况设计成为类及其对象的协作。然而，有时候，用用况所参与的子系统或子系统接口来设计一个用况会更为合适。例如，在自顶向下的开发过程中，在设计系统内部之前，需要捕获子系统及其接口的需求。或者，在某些场合，应该很容易将一个子系统及其内部设计替代为另一个具有不同内部设计的子系统。在这些情况下，一个**用况实现—设计**可以在子系统层次结构的多个层次上加以描述（参见图9-37）。

首先，需要识别实现用况所需的子系统，步骤如下：

- 研究参与相应的**用况实现—分析**中的分析类。识别包含那些分析类的分析包（如果有的话）。然后识别能够跟踪到那些分析包的设计子系统。
- 研究相应的**用况实现—分析**的特殊需求。识别实现那些特殊需求的设计类（如果有的话）。

然后识别包含那些类的设计子系统。

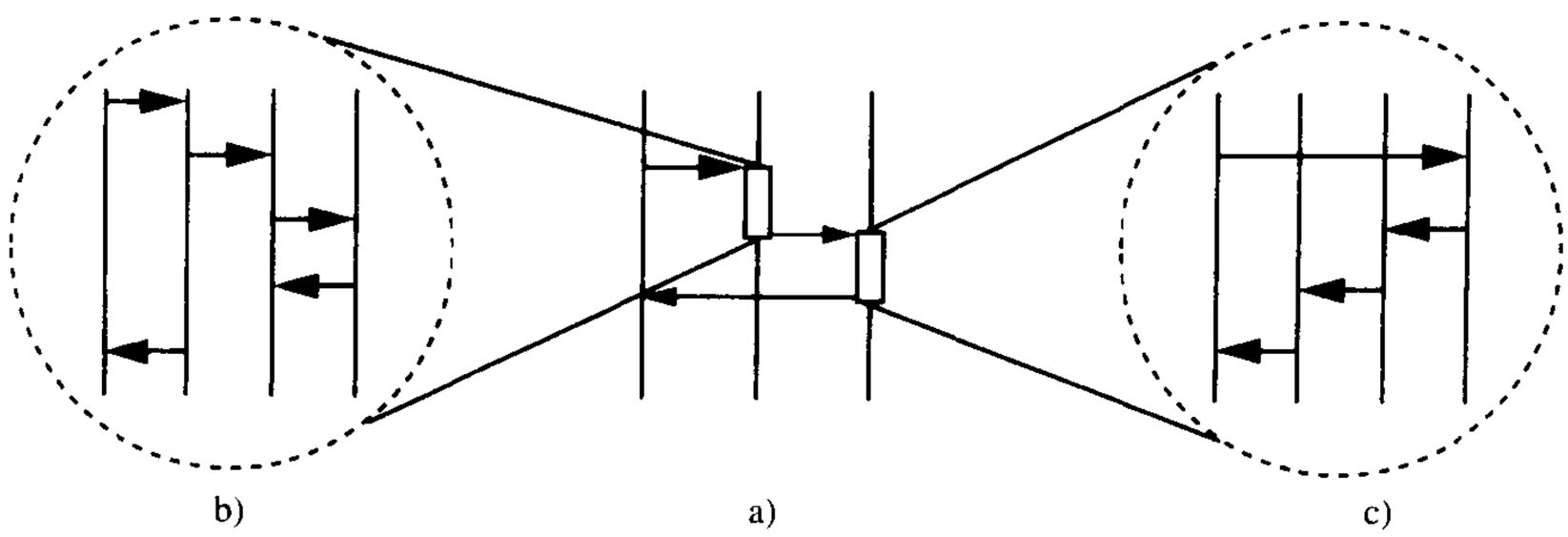


图9-37 中间的图（a）中的生命线代表子系统，并显示由子系统接收和发送的消息。另外两个图（b,c）代表这些子系统内部的设计，并显示这些子系统内部的元素是如何接收和发送消息的。图（a）可以在图（b,c）之前进行设计

收集与用况实现相关的某个类图中参与该用况实现的所有子系统，用这个类图来表示那些子系统和在用况实现中所用到的任何接口之间的依赖关系。

举例：参与“支付账单”用况实现的子系统和接口

图9-38所示的是涉及“支付账单”用况第一部分的一个类图，它包含子系统、接口以及它们之间的依赖关系。

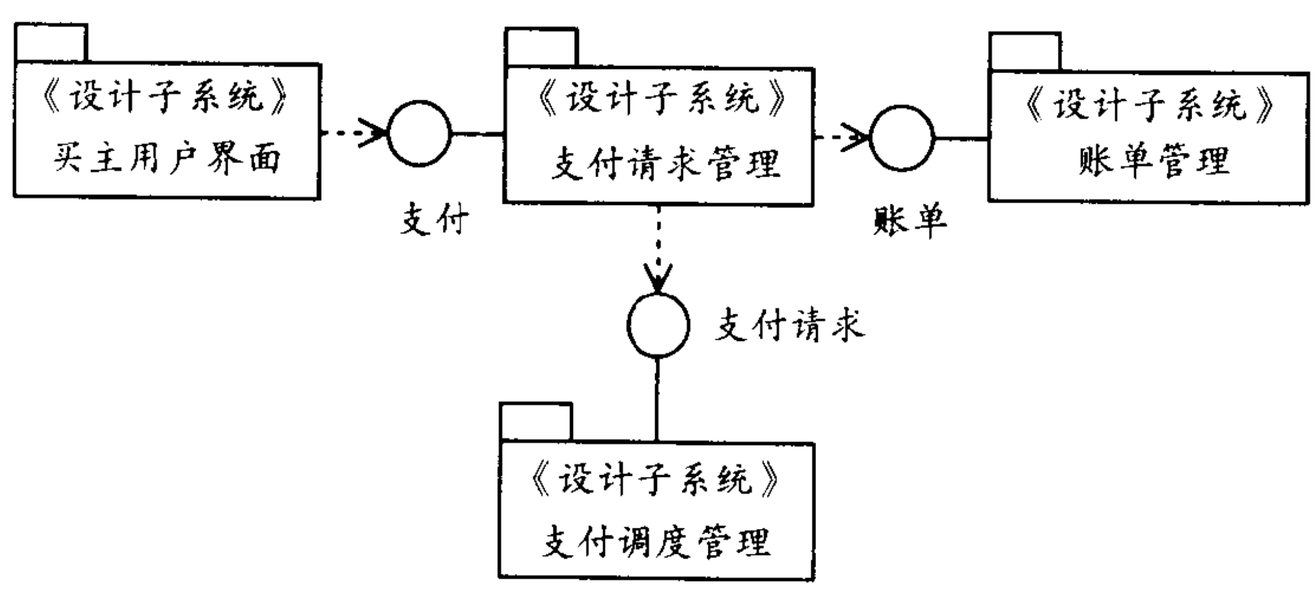


图9-38 带有子系统、接口以及它们之间的依赖关系的类图

9.5.2.4 描述子系统间的交互作用

当有了实现用况所需要的子系统的轮廓时，就需要描述它们所包含的类的对象在子系统层上是如何交互的。这可以通过使用包含参与的参与者实例、子系统以及它们之间的消息传递的顺序图来刻画。当这样做时，我们采用类似于9.5.2.2 节所描述的方法，只是在如下几个方面有所不同：

- 顺序图中原来设计对象的**生命线**（参见附录A）代表子系统的生命线。
- 9.5.2.3节中识别的每个子系统在一个顺序图中应该至少有一条生命线表示它；
- 当一个子系统提供多个接口时，需要分清哪个接口被用在哪个消息中。因此，如果一个消

息被赋给某个接口的一个操作，就需要考虑这个消息是否与提供这个操作的接口相匹配。

9.5.2.5 获取实现性需求

在这一步，我们将捕获基于一个用况实现的所有需求，比如在设计中识别的、但应该在实现中处理的非功能性需求。

> **举例：基于“支付账单”用况的实现性需求**
>
> 下面是一个由“支付账单”用况实现而引起的实现性需求的示例：
>
> “支付请求处理”（主动）类的一个对象，应该能够在任何单个买主未察觉的延迟时间内同时处理10个不同的买主客户端的请求。

9.5.3 活动：设计一个类

设计一个类的目的是为了创建一个设计类；这个设计类能够实现其在用况实现中以及非功能性需求中所要求的角色（参见图9-39）。这包括维护该设计类本身以及它的如下一些方面的内容：

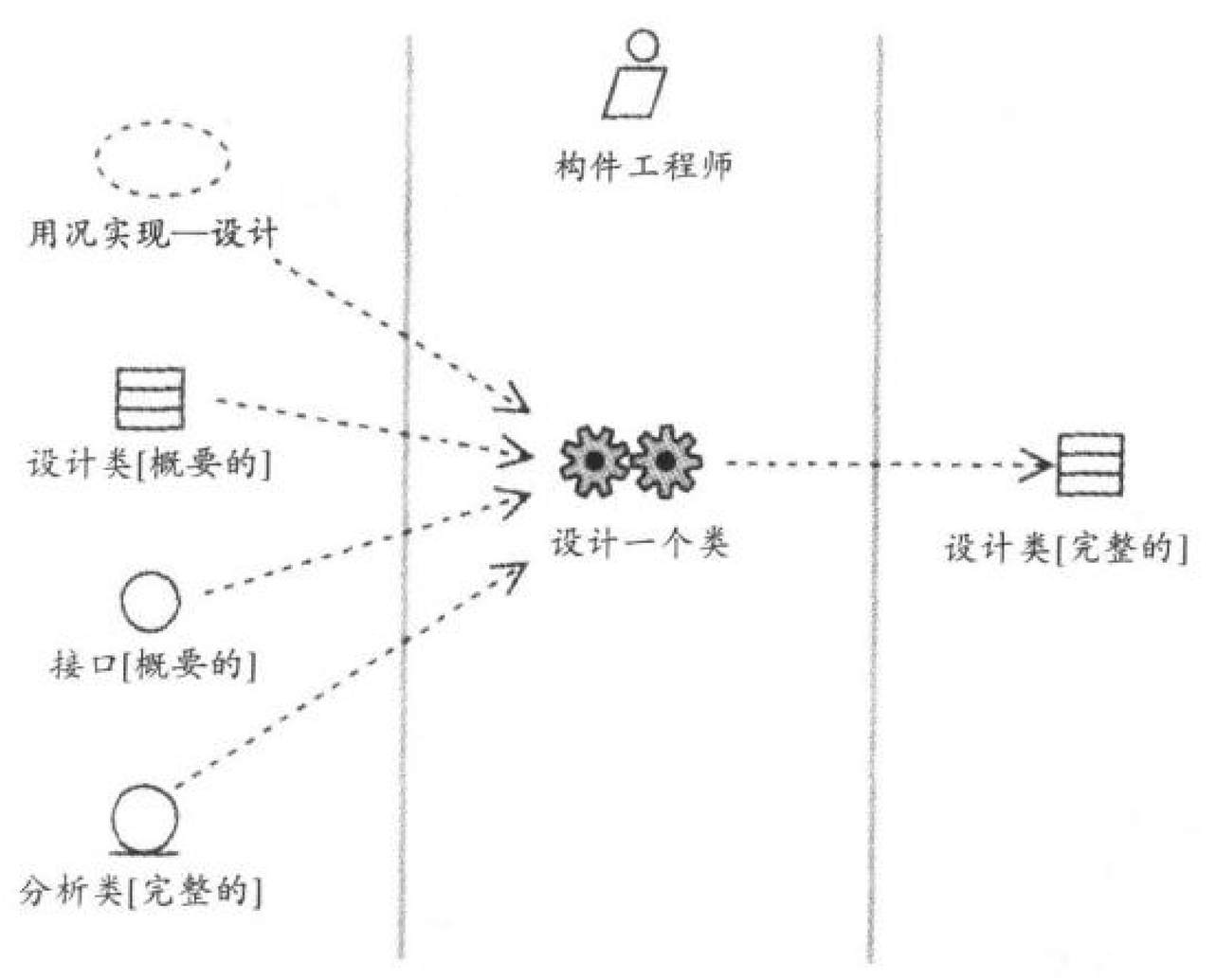


图9-39 设计一个类的输入和结果

- 操作
- 属性
- 所参与的关系
- 实现操作的方法
- 强制状态
- 对任何通用设计机制的依赖
- 与实现相关的需求
- 需要提供的任何接口的正确实现

9.5.3.1 勾画设计类

首先，需要根据给定分析类或接口作为输入来勾画出一个或几个设计类。当给定某个接口作为输入时，指定某个设计类来提供这个接口通常是简单易行且直截了当的。

当给定一个或多个分析类作为输入时，使用的方法依赖于分析类的构造型：

- 边界类的设计依赖于所采用的特定的接口技术。例如，在Visual Basic中设计的边界类暗指在用户界面中构造型为《form》（译成“表格”）的设计类和其他表示《Controls》（译成“控件”，可能是ActiveX 控件）的设计类。还需要说明的是，在某些流行的用户界面设计工具中，用户界面可以可视化地直接在屏幕上创建，从而使得相应的设计类的创建变得含蓄隐蔽。任何现存的用户界面原型都是本步骤的基本输入。
- 表示持久信息的实体类（或有其他持久性需求的类）的设计经常暗指使用特定的数据库技术。例如，创建用来映射到关系数据模型中的数据表的设计类。本步骤可以通过采用当前已有的建模工具来部分地实现自动化，虽然这样做也可能十分棘手且需要采用持久性策略（如数据库表示与存储）。采用该策略可能存在许多设计问题，尤其是在把面向对象设计模型映射到关系数据模型时更是如此。在开发过程中要解决这些问题，可能需要额外的工作人员（如数据设计人员）、活动（如数据库设计）和模型（如数据模型）。这些内容超出了本书所讨论的范围。
- 控制类的设计很棘手。因为它们封装了时序、与其他对象的协调、有时甚至是纯粹的业务逻辑问题。我们需要考虑下面的问题：
  - 分布问题：如果时序需要被分布到网络中的几个不同节点当中并加以管理，那么在这些节点上需要单独的设计类来实现控制类。
  - 性能问题：在各个节点上使用单独的设计类来实现控制类不一定合理，相反，可以由相同的设计类来实现控制类，这些相同的设计类实现一些相关的边界类或实体类。
  - 事务问题：控制类通常封装事务，它们相应的设计也必须与当前使用的事务管理技术相结合。

应该指出这一步所识别的设计类与它们所遵循（设计实现）的相应的分析类之间的跟踪依赖关系。当按下面的步骤进一步精化时，牢记设计类的“起源”是非常重要的。

9.5.3.2 识别操作

在这一步，我们将识别需要由设计类来提供的操作，并用编程语言的语法加以描述。其中包括详细说明每个操作的可见性（比如，C++中用*public*、*protected*、*private*表示）。本步骤的重要输入有：

- 设计类所跟踪的任何分析类的职责。每个职责通常暗含一个或多个操作。此外，如果为职责描述输入和输出，那么这些输入和输出可以用作形参和操作结果值的初步轮廓。
- 设计类所跟踪的任何分析类的特殊需求。回想一下，那些需求通常需要在设计模型中被处理，并可能需要与通用设计机制和数据库等技术相结合。
- 设计类需要提供的接口。接口的操作也需要由设计类来提供。
- 该类所参与的**用况实现—设计**（参见9.5.2节）。

一个设计类的操作需要支持该类在不同的用况实现中所扮演的所有角色。通过遍历用况实

现，并查看类和它的对象是否包含在该用况实现图中和实现的**事件流—设计**描述中，以此来发现这些角色。

举例："账单"类的操作

"账单"用况参与多个用况的实现，比如"支付账单"、"发送提示通知单"、"给买主开账单"等用况的实现。每一个这样的用况实现都读取或改变账单对象的状态。"给买主开账单"用况创建并提交账单；"支付账单"用况确定账单时间；等等。因此，每一个这样的用况实现按不同的方式使用"账单"对象，换句话说，"账单"类和它的对象在这些用况的实现中扮演了不同的角色。

首先，构件工程师考虑用一个名为"setStatus"的操作来实现这些状态的改变，这个操作拥有一个参数来指示所期望的活动与目标状态（例如，setStatus(Scheduled)）。但是随后他们决定宁愿为每个状态转移实现一个显式操作。进而，他们决定既对"账单"类、也对"交易对象"类（"账单"类是"交易对象"类的一个子类型，参见图9-40）采用这种方法。因此，"交易对象"类支持如下的操作（每个操作会改变交易对象的状态）：创建(Create)、提交(Submit)、调度(Schedule)和关闭(Close)。然而，这些操作仅仅是定义一个特征标记的虚拟操作。每个"交易对象"类的子类型将具体定义实现这些操作的方法。

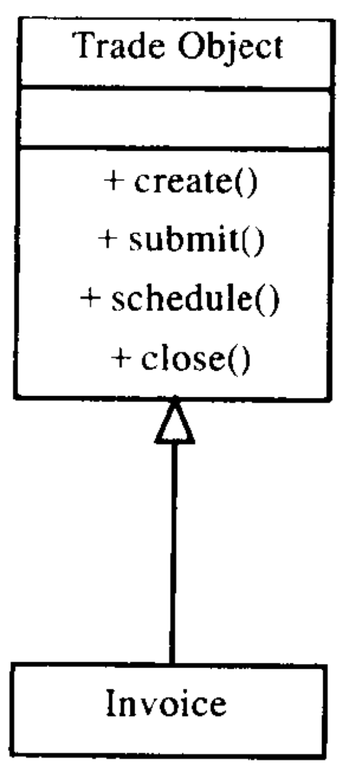


图9-40 "账单"（Invoice）类、它的超类型"交易对象"（Trade Object）以及它所支持的操作

对于某些类，它们的对象的行为高度依赖于对象的状态，这些类最好用状态图来描述（参见9.5.3.7节）。

9.5.3.3 识别属性

在这一步，我们将识别设计类所需的属性，并用编程语言的语法来加以描述。一个属性详细说明了一个设计类的特性，并经常被类的操作所隐含和调用（如前一步所考虑的）。在识别属性时要牢记如下的通用指导方针：

- 考虑设计类所跟踪的任何分析类的属性。有时，这些属性隐含着设计类需要有一个或多个属性。
- 用编程语言来约束可用的属性类型。
- 当选择一个属性类型时，尽可能重用已经存在的属性类型。

- 单个的属性实例不能被多个设计对象共享。如果需要共享，就需要将该属性定义成一个单独的类。
- 如果一个设计类因为其属性的原因变得复杂而难于理解，那么其中的一些属性可以分离出来变成它们自己的类。
- 如果一个类存在大量的或者复杂的属性，可以在一个单独的、只显示属性部分的类图中进行阐述。

9.5.3.4 识别关联和聚合

在顺序图中设计对象相互作用，这种对象间的交互通常需要它们对应的类之间有关联。因此，构件工程师应该研究顺序图中的消息传递，以确定需要哪些关联。关联的实例可用于保持与其他对象之间的参考关系，这些实例还可用来将对象分组成为聚合，从而可以将消息发送给它们。

类之间关系的数量应该最小化。那些现实世界中的主要关系并不一定要被模型化为关联或聚合，但不同用况实现所需要的那些关系应该被模型化为关联和聚合。还需要注意的是，因为性能问题需要在设计过程中处理，所以可能需要通过关联和聚合对最佳检索路径进行建模。

在识别和精化关联和聚合时，应牢记如下的通用指导方针：

- 考虑包括在相应的分析类（或类族）中的关联或聚合。有时候，这些（分析模型中的）关系隐含着设计类需要包括一个或多个（设计模型中的）相应的关系。
- 在当前使用的编程语言的支持下，精化关联多重性、角色名称、关联类、有序角色、合格角色和n元关联。例如，当生成编码时，角色名称可以变成设计类的属性，从而限定角色名称的格式。或者，一个关联类可以变成其他两个（起源）类之间的一个新类，因此需要在“关联”类和其他两个类之间有一个具有合适多重性的新关联。
- 精化关联的导航性。考虑使用了关联的交互图。设计对象之间的消息传送方向隐含了它们的类之间的关联的相应的导航性。

9.5.3.5 识别泛化

应该用与编程语言所定义的语法相同的语法来使用泛化。如果编程语言不支持泛化（或者继承），那么可以用关联或聚合来替代，以提供从更特定的类的对象到更共性的类的对象的委托关系（比如，发送要求完成某项工作的消息，将工作完成，返回一个确认消息）

举例：设计模型中的泛化

“账单”和“订单”都按类似方法改变状态并支持类似的操作。它们都是一个更一般的“交易对象”的特化（参见图9-41）。

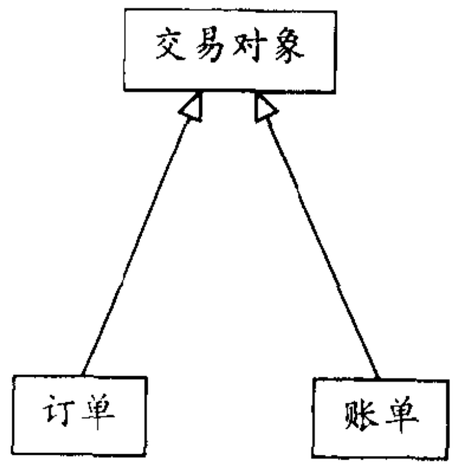


图9-41 “交易对象”泛化了“账单”和“订单”。注意，这些泛化关系也存在于分析模型中对应的分析类之间

9.5.3.6 描述方法

在设计过程中，方法是用来详细说明操作是如何实现的。例如，一个方法可以详细说明一种用于实现一个操作的算法。只要合适，方法可以用自然语言或者伪代码来详细说明。

然而在设计过程中，在绝大部分的时间里都不对方法加以详细说明；相反，方法是在实现过程中直接用编程语言创建的。这是因为同一个构件工程师既要设计又要实现类，从而没有必要进行诸如方法的规格说明等规格说明书的移交工作。

注意，如果设计工具支持从设计类的方法到代码的平滑生成，那么方法就可以在设计工具中用编程语言直接加以详细说明。但是，这些并不是设计活动要处理的问题，而是实现活动需要考虑的问题。请参见第10章中有关类的实现活动（参见10.5.4节）的详细讨论。

9.5.3.7 描述状态

某些设计对象是由状态控制的，也就是说，当它们接收到一个消息时，它们的状态决定了它们的行为。在这种情况下，用状态图来描述一个设计对象的不同状态转移就很有意义。因此，这样的一个状态图是相应设计类的实现的一个有价值的输入。

举例："账单"类的状态图

"账单"对象在创建、提交、调度和关闭时改变状态。对所有交易对象，这些状态改变遵循严格的顺序。例如，在提交账单前，不可能确定账单支付时间。这样的状态改变顺序可以用一个状态图来定义，如图9-42所示。

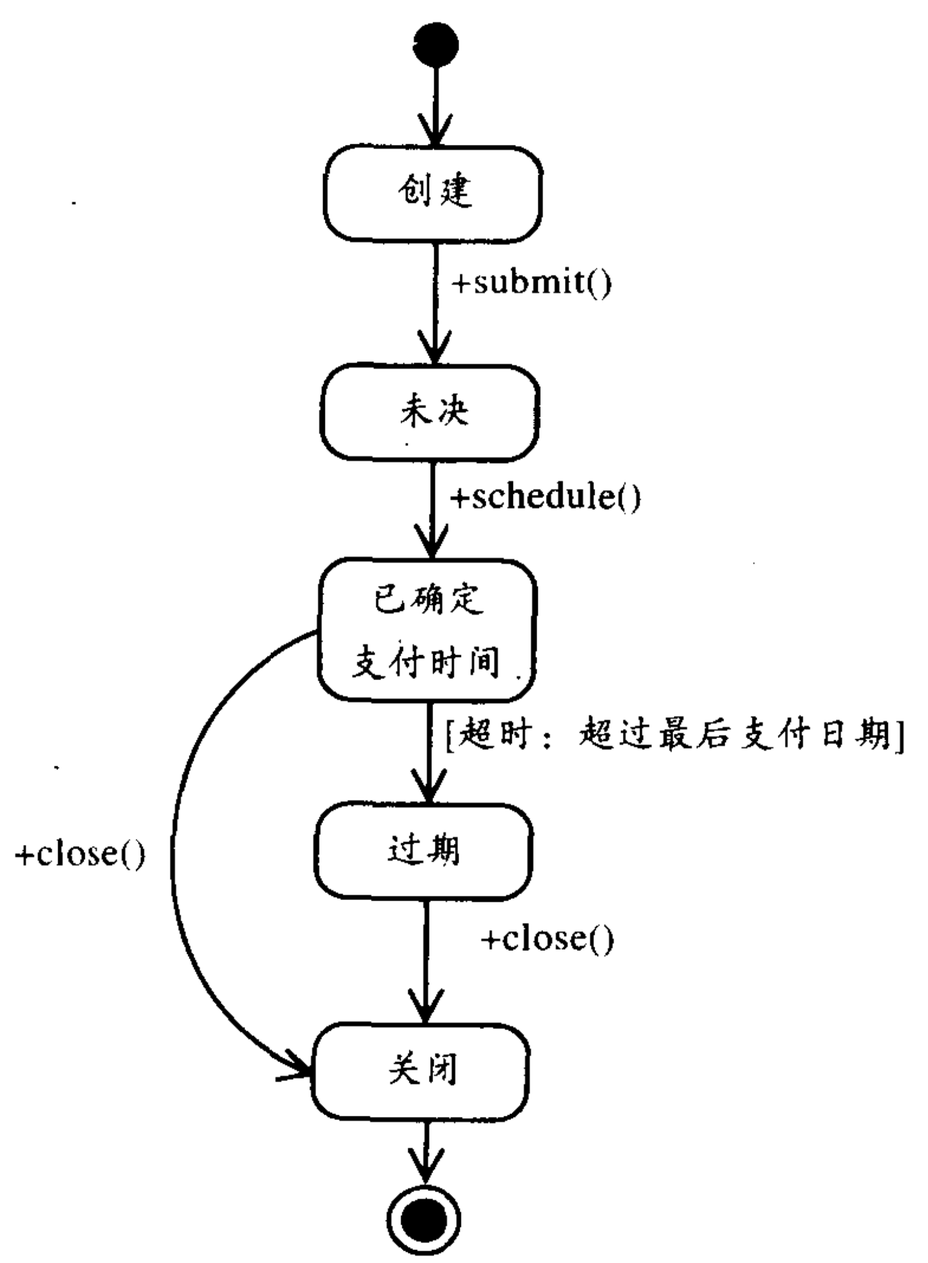


图9-42 "账单"类的状态图

当卖主要求买主对一个订单付款时，就创建一个账单。然后把账单提交给买主，状态变成“未决”。当买主决定付款时，账单状态就变成“已确定支付时间”，然后，如果买主没有按时付款，账单状态就变成“过期”。最后，当账单被付款时，状态就变成“关闭”。

9.5.3.8 处理特殊需求

在这一步，我们将处理在前面步骤没有考虑到的任何需求。在处理时，要研究类所参与的用况实现所引起的需求，这些需求可能陈述了对设计类的（非功能性）需求。另外，还需要研究对设计类可以跟踪到的任何分析类的特殊需求，这些需求通常需要由设计类来处理。

举例：使用一种设计机制来处理特殊需求

需要从多个节点（从“买主服务器”和“卖主服务器”两者）来访问“账单”对象。“账单”不是一个主动类，但它必须设计得适合于分布系统。在我们的例子里，通过把“账单”定义成一个抽象的Java类——java.rmi.UnicastRemoteObject（它支持远程消息请求RMI）的子类来实现对象的分布（参见图9-43）。请注意，这种设计机制是由构架设计师在构架设计活动中识别和阐述的。

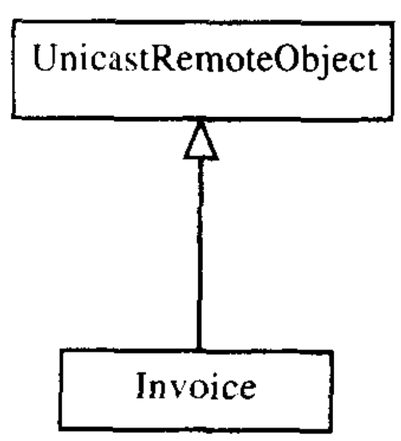


图9-43 “账单”(Invoice)类的对象必须是分布式的，这是通过使它们成为UnicastRemoteObject的子类的方式来实现的

当处理这些特殊需求时，如果可能，要使用任何被提及到的（如构架设计师所识别出的）通用设计机制。

然而，把对某些需求的处理推迟到实现阶段可能是合适的。这样的一些需求应该被标注为对设计类的实现性需求。

举例：对一个主动类的实现性需求

下面是对“支付请求处理”类的实现性需求的一个示例。

- 一个“支付请求处理”类的对象应该能在买主未察觉的延迟时间内同时处理10个不同买主客户端的请求。

9.5.4 活动：设计一个子系统

设计一个子系统的目的是为了（参见图9-44）：

- 确保该子系统尽可能地独立于别的子系统或它们的接口。
- 确保该子系统提供正确的接口。

• 确保该子系统实现其目标，即提供其接口所定义操作的正确实现。

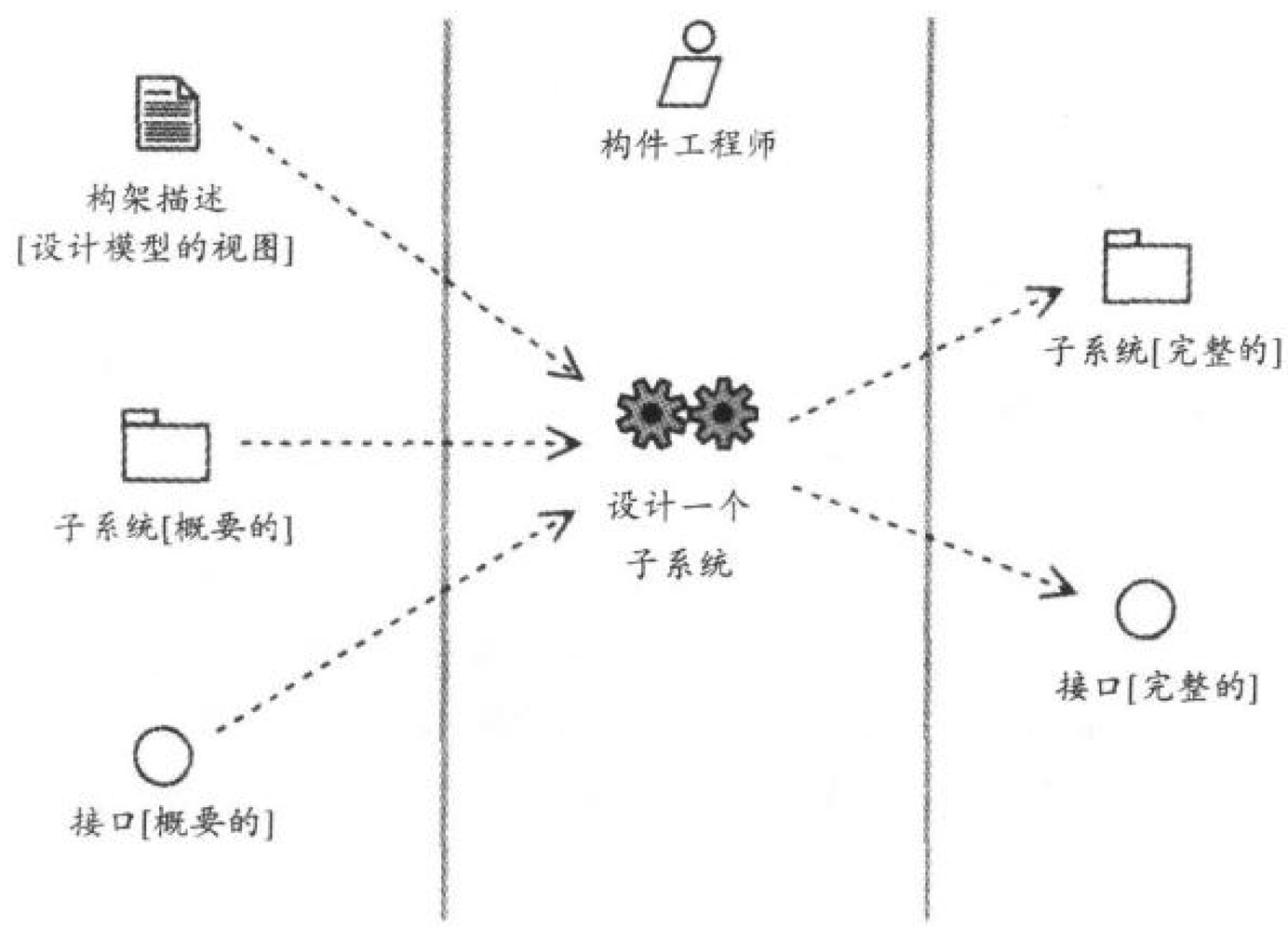


图9-44 设计一个子系统的输入和结果

9.5.4.1 维护子系统的依赖关系

应该定义和维护从一个子系统到其他子系统（它们所包含的元素与该子系统内的元素相关联）的依赖关系。然而，如果那些（其他）子系统提供接口，（使用）依赖性就应该直接定义到那些接口。因为一个子系统可能被别的包含有另外内部设计的子系统所替代，而在那种情况下接口并不需要被替换，所以对一个接口的依赖比对一个子系统的依赖更可取。

要力图将对其他子系统或接口的依赖关系降低到最小限度，要考虑将这些子系统中那些太依赖于其他子系统的类进行重新定位。

9.5.4.2 维护由子系统所提供的接口

由子系统所提供的接口所定义的操作需要支持该子系统在不同用况实现中所扮演的所有角色。即使接口是由构架设计师勾画的，随着设计模型的进化与用况的设计，这些接口可能需要由构件工程师进一步精化。请回想一下，一个子系统及其接口可能被用况工程师用于不同的用况实现中，从而对接口提出了进一步的要求（参见9.5.2.4节）。

接口及其操作的维护方法类似于9.5.3.2节中所述的设计类及其操作的维护方法。

9.5.4.3 维护子系统的内容

当一个子系统提供由它的接口所定义操作的正确实现时，这个子系统就达到了它的目的。即使子系统的内容是由构架设计师勾画的，随着模型的进化，这些内容可能需要由构件工程师来进一步精化。与此相关的共性问题包括：

- 对于由该子系统所提供的每一个接口，在该子系统内部应该有能提供该接口的设计类或者其他的子系统。
- 为了阐明一个子系统的内部设计是如何实现它的所有接口或者用况，可以根据该子系统所

包含的元素创建协作关系。调整该子系统所包含的元素可以实现这一点。

举例：在子系统内提供接口的设计类

“买主账单管理”子系统提供“账单”接口，负责该子系统的构件工程师决定让“账单”类实现这个接口，参见图9-45。一个可选的实现方案是让其他的一些设计类来实现该接口，然后使用该“账单”类。

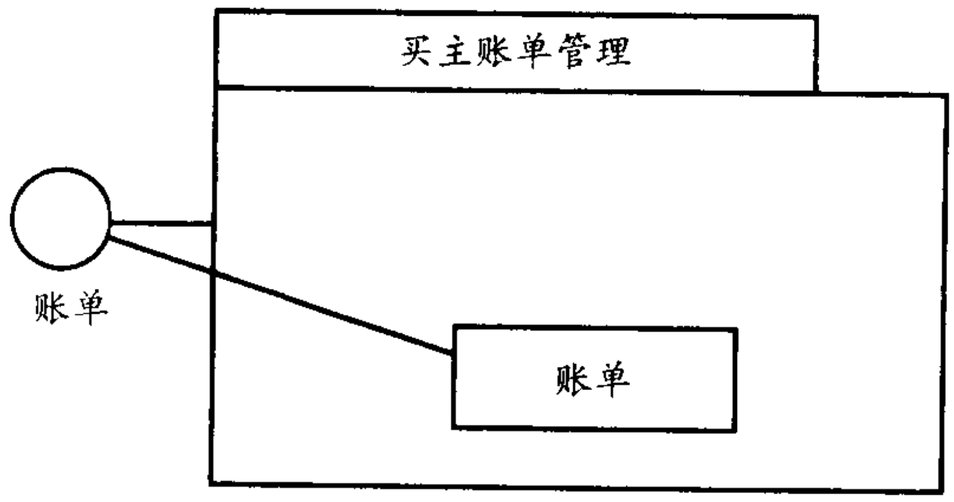


图9-45 “账单”类提供“账单”接口（这个“账单”接口是由“买主账单管理”子系统来提供的）

9.6 设计小结

设计的主要结果是设计模型，设计模型受分析模型的影响，它要极力保持系统的结构，并用作实现的蓝图。设计模型包括如下一些元素：

- 设计子系统和服务子系统以及它们的依赖关系、接口和内容。居于最高两个层次（即专用应用层和通用应用层）的设计子系统的轮廓基于分析包（参见9.5.1.2.1节）。设计子系统的部分依赖关系的轮廓基于相应分析包的依赖关系（参见9.5.1.2.3节）。部分接口的轮廓基于分析类（参见9.5.1.2.4节）。
- 设计类，包括主动类以及它们的操作、属性、关系和实现性需求。一些对构架有重要意义的设计类的轮廓基于对构架有重要意义的分析类（参见9.5.1.3.1节）。部分主动类的轮廓基于分析类（参见9.5.1.3.2节）。通常，在勾画设计类的轮廓时，分析类用作设计类的规格说明（参见9.5.3.1节）。
- **用况实现—设计**，它描述如何用设计模型内的协作关系来设计用况。通常，在勾画设计**用况实现—设计**的轮廓时，**用况实现—分析**用作规格说明（参见9.5.2节）。
- 设计模型的构架视图，包括对构架有重要意义的元素。在早前曾指出，在勾画设计模型中对构架有重要意义的元素时，分析模型中对构架有重要意义的元素用作规格说明。

设计还产生实施模型，它描述了系统分布应具备的任何网络配置。实施模型包括：

- 节点以及它们的特征和连接关系
- 主动类到节点的初步映射
- 实施模型的构架视图，包括对构架有重要意义的元素。

设计模型和实施模型被看成是后续的实现和测试活动的主要输入，我们将在后面的章节中介绍这些内容。特别要注意的是：

- 设计子系统和服务子系统将由包含实际构件（例如源代码文件、脚本、二进制代码、可执行程序等）的实现子系统来实现。这些实现子系统将一对一地（同构地）对应到设计子系统。
- 设计类将由包含源代码的文件构件来实现。在单个的文件构件中实现多个设计类是常见的（虽然这要依赖于所采用的编程语言）。同样地，在创建可执行构件时，承担繁重处理任务的主动设计类将作为一个输入。
- 当按很小的、可管理的步骤来规划和实施实现工作时，将使用**用况实现—设计**来产生一系列的“构造”。每个构造将主要实现一组用况实现或者其中的一部分。
- 当采用把可执行构件实施到节点上的方法来对系统进行分布时，将用到实施模型和网络配置。

9.7 参考资料

[1] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Reading, MA: Addison-Wesley, 1994.

[2] Ivar Jacobson, Stefan Bylund, Patrik Jonsson, Staffan Ehnebom, "Using contracts and use cases to build plugable architectures," *Journal of Object-Oriented Programming*, June, 1995.

[3] Ivar Jacobson, Martin Griss, and Patrik Jonsson, *Software Reuse: Architecture, Process and Organization for Business Success*, Reading, MA: Addison-Wesley, 1997.

第10章 实　　现

10.1 引言

在实现阶段，将基于设计的结果，探讨如何用源代码、脚本、二进制代码、可执行体等构件来实现系统。

幸运的是，系统构架的绝大部分都是在设计过程中捕获的。实现的主要目的是从总体上充实构架和系统。更明确地讲，实现的目的如下：

- 规划每次迭代中所要求的系统集成。采用增量式开发方法，即采取一系列细小且易于管理的步骤来实现一个系统。
- 通过把可执行构件映射到实施模型中的节点的方式来分布系统。这主要基于设计过程中发现的主动类。
- 实现设计过程中发现的设计类和子系统。特别是要将设计类实现为包含源代码的文件构件。
- 对构件进行单元测试，然后通过编译和连接把它们集成为一个或多个可执行程序，之后再送去做集成和系统测试。

在本章以及后续章节，将阐明如何开展实现工作以及哪些工作人员和制品会参与其中（参见图10-1所示）。所采取的方法与设计工作流的方法几乎相同。

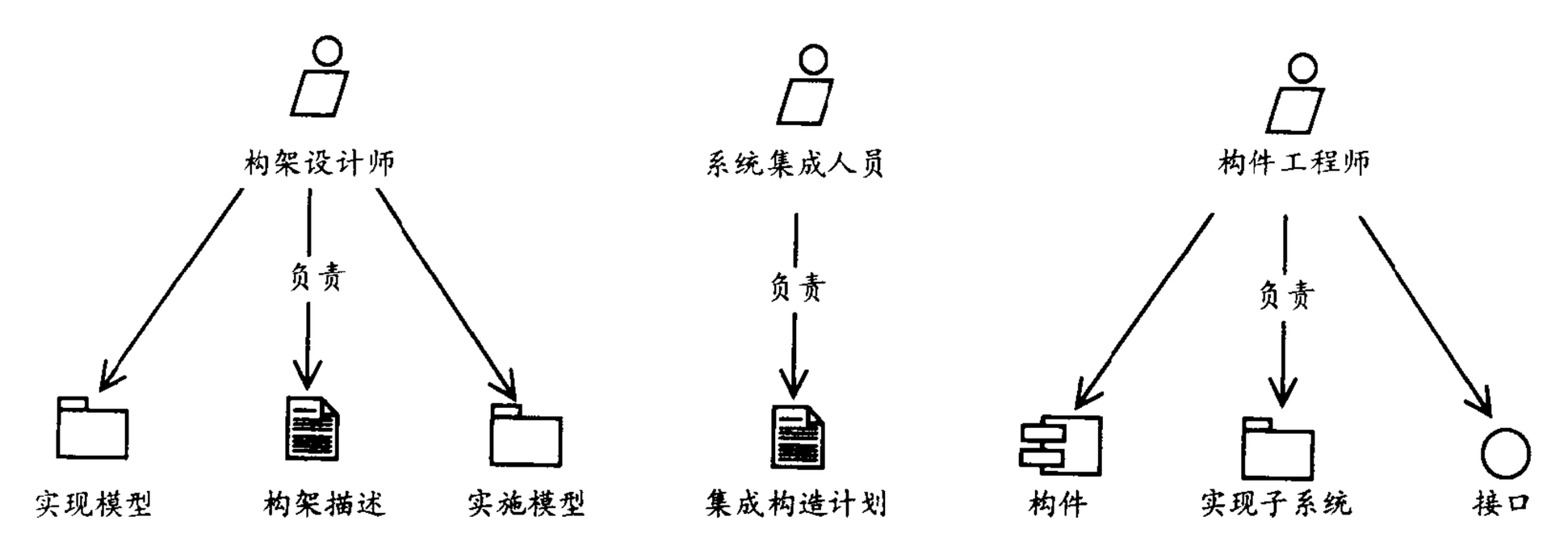


图10-1实现阶段中的工作人员和制品

10.2 实现在软件生命周期中的作用

实现主要集中在构造迭代阶段。细化阶段也要进行实现活动，目的是创建可执行的构架基

线；而移交阶段的实现活动是处理那些在系统beta版发布时发现的最新缺陷（如图10-2中移交区域中的尖峰所示）。

由于实现模型阐述如何用构件和实现子系统来表示系统的实际实现，因此在整个软件生命周期中很自然地要维护实现模型。

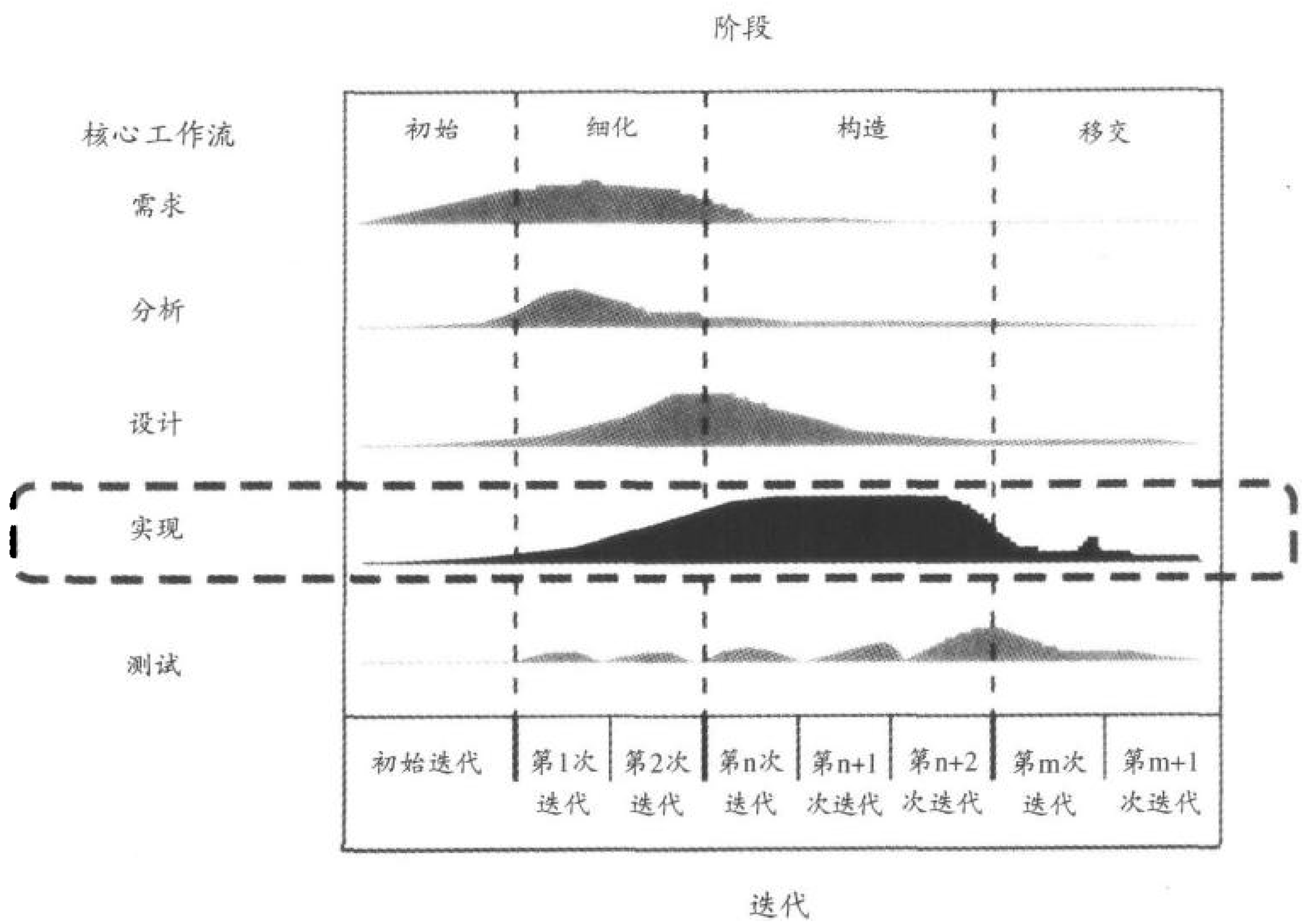


图10-2 实现的焦点

10.3 制品

10.3.1 制品：实现模型

实现模型描述如何用源代码文件、可执行程序等构件来实现设计模型中的元素（比如设计类）。实现模型还描述构件是如何通过相应的结构和模块化机制（适用于实现环境和所用的编程语言）组织起来的、以及这些构件之间是如何相互依赖的。

实现模型定义了如图10-3所示的一种层次结构。

用表示模型顶层子系统的实现系统来表达实现模型，因此利用其他子系统是将实现模型组织成为更易于管理的功能块的一种方法。

下面将详细讨论实现模型中的各个制品。

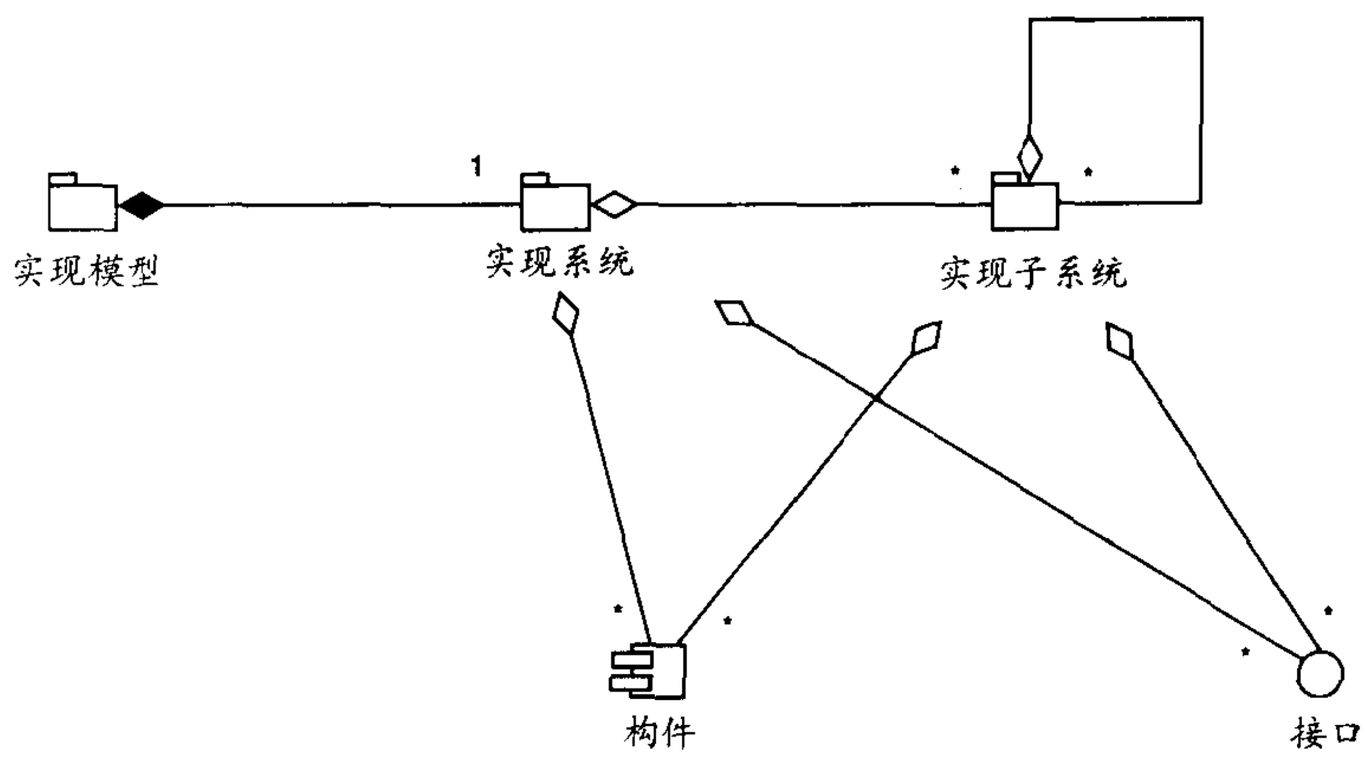


图10-3 实现模型是一个包含构件和接口的实现子系统的层次结构

10.3.2 制品：构件

构件是模型元素（如设计模型中的设计类）的物理包[5]。构件有如下一些标准构造型：

- 《可执行体》表示一个可以在节点上运行的程序。
- 《文件》表示一个含有源代码或数据的文件。
- 《库》表示一个静态库或者动态库。
- 《表》表示数据库中的一个表。
- 《文档》表示一个文档。

当在一个特定的实现环境中创建构件时，可将这些构造型加以修改以反映它们所代表的实际构件。构件具有如下特征：

- 构件拥有到它们所实现的模型元素的跟踪关系。
- 一个构件实现多个元素（如设计类）的情况很普遍。然而，如何准确地创建这样的跟踪关系，依赖于如何用给定的编程语言进行源代码文件的构造和模块化。

举例：可跟踪到设计类的构件

在Interbank系统中，设计类“Account Transfers”是在AccountTransfers.java源代码构件中实现的。这是因为为每个类创建一个“.java”源代码文件是Java语言的一种惯常用法，虽然这并不是一种强制要求。总之，这是通过设计和实现模型间的跟踪依赖关系来建模的 (参见图10-4所示)。

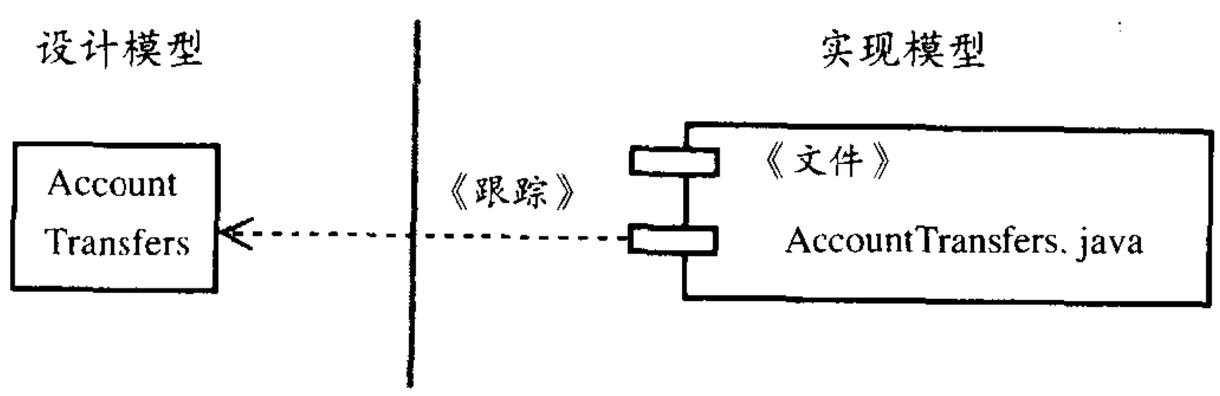


图10-4 构件和设计类间的跟踪依赖关系

举例：设计和实现中的接口

在Interbank系统中，“Account Transfers”设计类提供了一个“Transfers”接口。这个接口也是由实现“Account Transfers”类的AccountTransfers. java构件提供的（参见图10-5）。

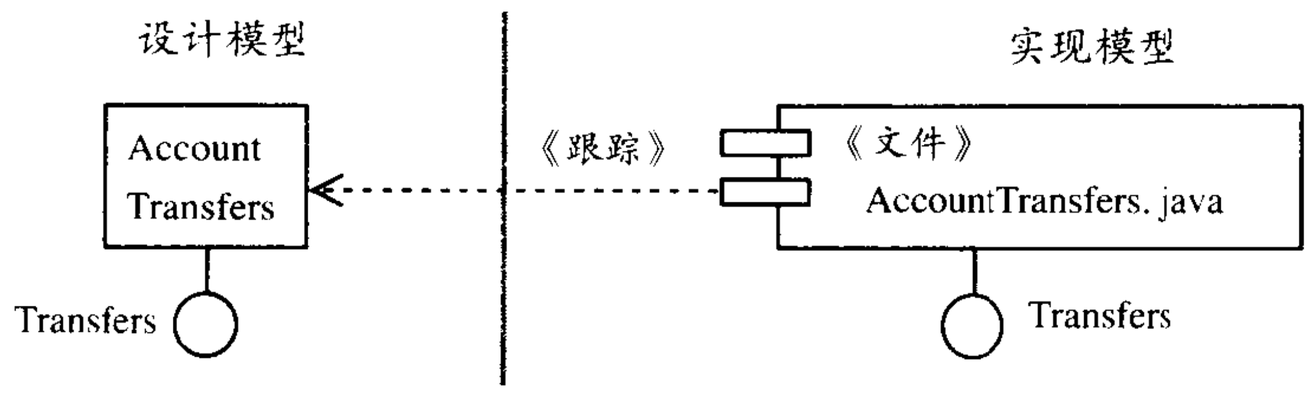


图10-5 构件提供与其所实现的类相同的接口

- 构件提供与其所实现的模型元素相同的接口。
- 构件间可能存在编译依赖，以指示哪些构件是编译某个特定的构件所需要的。

举例：构件间的编译依赖

在Interbank系统中，如图10-6所示，AccountTransfers. java（文件）构件编译成一个叫做AccountTransfers. class（可执行体[⊖]）的构件。

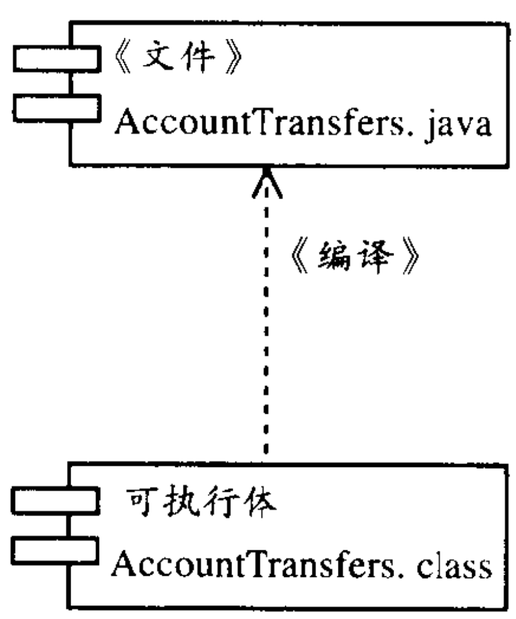


图10-6 两个构件间的编译依赖

10.3.2.1 桩

桩是一种具有概略的或具有特定目标的实现的构件，可用来开发或测试另一个依赖于该桩的构件；在[1]中给出了类似的定义：桩可用于使系统的每个新的（中间的）版本所需要的新构件数量最小化，从而简化集成问题和集成测试（参见10.5.2节“集成系统”）。

10.3.3 制品：实现子系统

实现子系统提供一种把实现模型的制品组织成更易于管理的功能块的方法（参见图10-7）。一

⊖ 更具体地讲，在Java虚拟机中的构件实际上是可解释的。

个子系统可以包含构件、接口及其他的子系统（递归地）。此外，一个子系统可以实现（并因此可以提供）接口，这些接口以操作形式展现它们所实现的功能。

有一点非常重要，那就是要明白实现子系统是由实现环境中具体的“包机制”所体现的，比如：

- Java中的包(*package*)[2]。
- Visual Basic 中的项目(*project*)[3]。
- C++项目中文件的目录(*directory*)。
- 在集成开发环境（如Rational Apex）中的子系统(*subsystem*)。
- 在可视化建模工具（如Rational Rose）中的构件视图包(*component view package*)。

因此，当在具体的实现环境中出现时，实现子系统（*implementation subsystem*）的概念将获得一些精确的语义。而本章在通用层面上讨论实现，从而可适用于大多数实现环境。

实现子系统与设计模型中的设计子系统（参见第9章，9.3.4节“制品设计子系统”）有非常紧密的联系。事实上，一个实现子系统与相应的设计子系统应该是一对一的跟踪关系。请回忆一下设计子系统已经作出的定义：

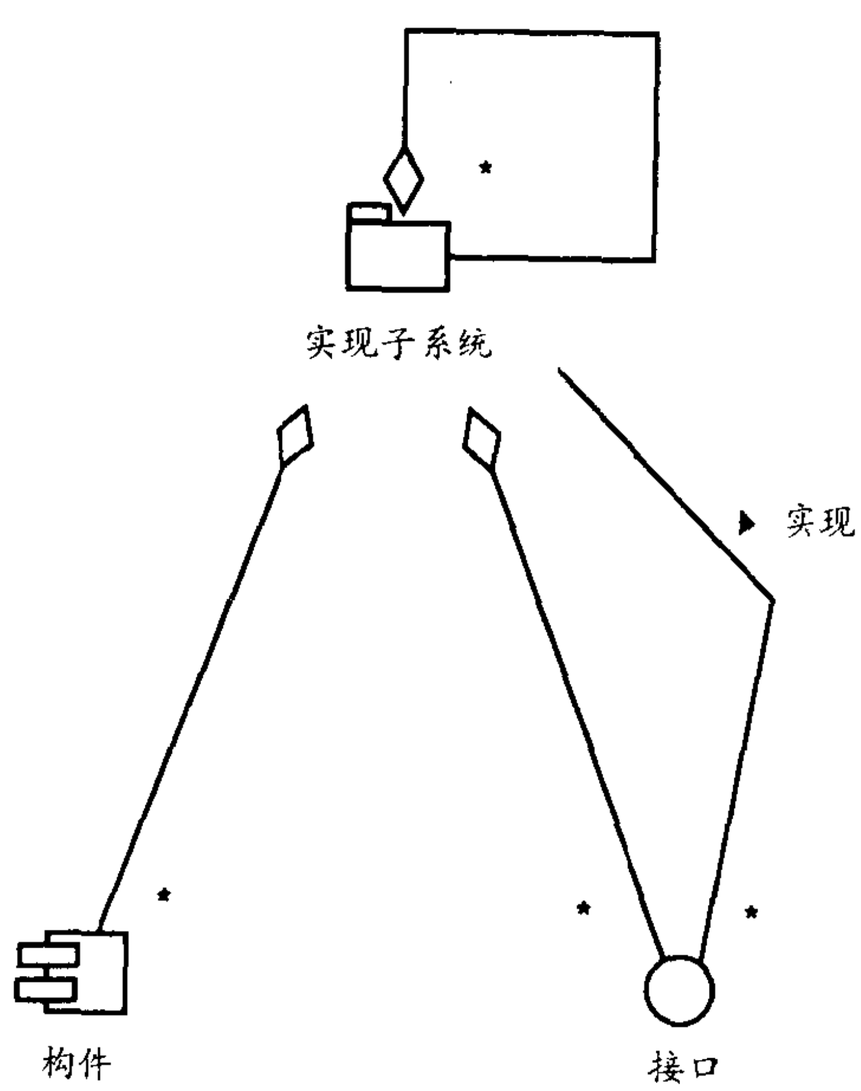


图10-7 子系统的内容和关键关联

- 对其他子系统或其接口的依赖。
- 子系统会提供什么样的接口。
- 子系统内的哪些设计类或其他设计子系统（递归地）应该为子系统提供本该由该子系统提供的接口。

上述几点对于相应的实现子系统非常重要，原因如下：

- 实现子系统应该定义与其他（相应的）的实现子系统或接口的类似的依赖关系。
- 实现子系统应该提供相同的接口。

- 实现子系统应该作出定义：在该子系统内，哪些构件或者递归地包含于该子系统中的其他的实现子系统应该提供本该由该子系统提供的接口。此外，这些包含于其中的构件（或者实现子系统）应该能跟踪到它们所实现的设计子系统中的相应的类（或者设计子系统）。

图10-8阐明了设计和实现模型间的关系。

子系统提供和使用接口的方法的任何变化或者接口本身的变化，都在设计工作流中加以描述（请参见第9章，9.5.4节 “活动：设计一个子系统”）。因此，虽然这些变化也会影响实现模型，但它们不在实现工作流中加以处理。

请注意，刚才描述的映射关系对设计模型中的服务子系统也成立。总之，与某个服务子系统对应的实现子系统将驻留在实现模型层次中的较低层，并完成与服务子系统相同的目标，也就是说，通过提供服务来构造相应的系统。为此，与设计模型中的服务子系统有跟踪关系的实现子系统很可能封装提供各种各样系统服务的可执行构件。

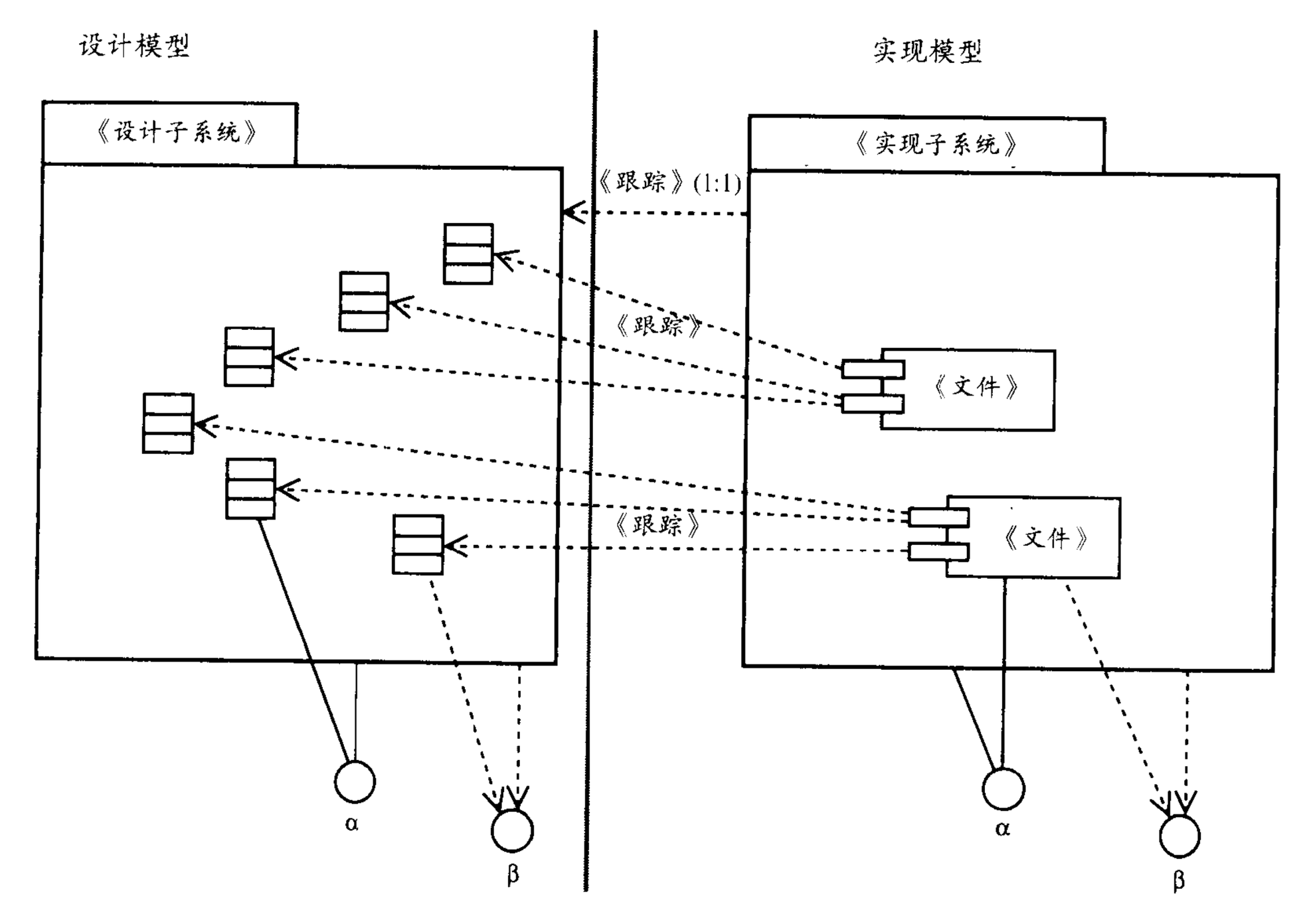


图10-8 实现模型中的实现子系统与设计模型中的设计子系统存在一对一的跟踪关系。不同模型中的子系统提供相同的接口(α)并依赖于相同的接口(β)。实现子系统中的构件实现设计子系统中的类

10.3.4 制品：接口

在第9章曾对接口作过详细描述。然而，同样的接口可用在实现模型内，用来详细说明由构件和实现子系统所实现的操作。此外，如先前所述，构件和实现子系统对接口有“使用依赖关系”

(参见图10-9)。

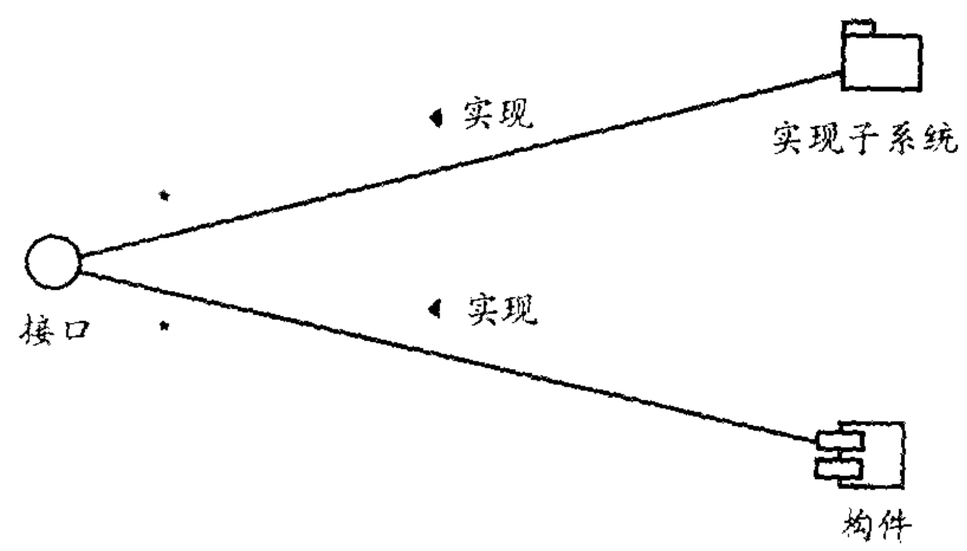


图10-9 接口的关键关联

实现（且因此提供）一个接口的构件必须按正确的方式实现该接口所定义的全部操作。提供接口的实现子系统也必须包含提供该接口的构件或提供该接口的其他子系统（递归地）。

举例：提供接口的实现子系统

Interbank系统中有一个叫“AccountManagement”的实现子系统（一个Java包），它提供“Transfers”接口（参见图10-10）。

图10-10 “AccountManagement”（账户管理）子系统提供“Transfers”（转账）接口

图10-11中给出了“转账”接口的Java代码。

```
package AccountManagement;

// provided interfaces:

public interface Transfers {

        public Account create(Customer owner, Money balance,

                                        AccountNumber account_id);

        public void Deposit (Money amount, String reason);

        public void Withdraw (Money amount, String reason);

}
```

图10-11 为“账户管理”子系统所提供的“转账”接口编写的Java代码

10.3.5 制品：构架描述（实现模型的视图）

构架描述包含**实现模型的构架视图**（参见附录C），阐述了对构架有重要意义的制品（参见图10-12）。

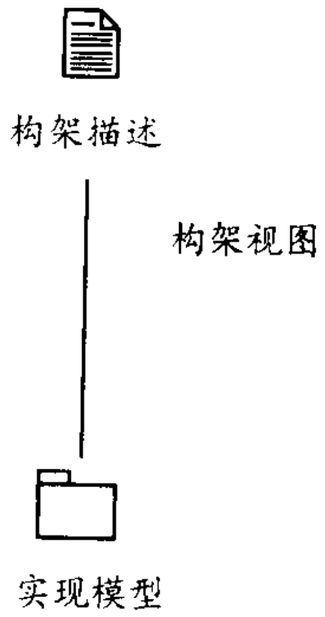


图10-12 实现模型的构架视图

通常认为构架模型中的下列制品对构架有重要意义：

- 实现模型分解成子系统、子系统接口以及它们之间的依赖关系的分解。一般来讲，这样的分解对于构架具有非常重要的意义。然而，因为实现子系统与设计子系统有一对一的跟踪关系，并且设计子系统很可能在设计模型的构架视图中作了描述，所以通常并不需要在实现模型的构架视图中再对实现子系统进行描述。
- 关键的构件，如那些可跟踪到对构架有重要意义的设计类的构件、可执行构件以及通用的、核心的、实现通用设计机制的构件(许多其他构件要依赖于它)。

在10.5.1.1节“确定对构架有重要意义的构件”中，我们提供了一个示例来说明哪些制品应该包括在实现模型的构架视图中。

10.3.6 制品：集成构造计划

软件按照可管理的步骤增量地构造是非常重要的，这样，在每一步中要解决的集成或测试问题就很少。每一步的结果称之为“构造”，它是系统的一个可执行版本，通常是系统的某个特定的部分。在后续的构造创建之前，对每个构造进行集成测试（参见第11章中的描述）。为防备出现失败的构造（比如，没有通过集成测试），要将每个构造置于版本控制之下，由此才可能回卷到某个早期的构造。这种增量式的开发方法的优点是：

- 可以将系统的一个可执行版本早早地创建出来，而不必要等待一个更完整的版本才创建可执行版本。集成测试也很早就开展，可执行版本可用于向内部项目成员和外部项目相关人员证明系统的特征。
- 缺陷在集成测试过程中较易定位，因为每次只对已存在的构造增加或精化较少且可管理的一些部分。更理想的状况是，缺陷很可能只与新增加或被精化的部分有关（当然并不总是这样）。
- 集成测试比系统测试更彻底，因为集成测试能够专注于更小、更便于管理的部分。

一般来讲，**增量式集成**（附录C）是对于系统集成而言的，而受控的迭代式开发是针对软件开发而言的。两者都专注于小的、可管理的功能增量。

把增量式集成置于迭代式开发语境中，每次迭代将至少产生一个构造。然而，某些特殊的迭代中要实现的功能往往太复杂，以致于不能集成到一个单一的构造中，在该次迭代中就会产生一系列的构造，每个构造向系统添加一个小的增量，以一个可管理步骤把系统的建造工作向前推进。每次迭代则将产生一个较大的增量追加到系统，其间可能积聚了好几个构造（参见第5章）。

集成构造计划（*integration build plan*）阐述在一次迭代中所需的构造序列。更明确地讲，该计划将为每个构造作如下的描述：

- 希望在构造中实现的功能。这是一个用况或场景的清单，或者是其中的一部分（在前面章节中曾作过阐述）。这样的清单也可能参考其他的补充需求。
- 实现模型中的哪些部分受到该构造的影响。这是为实现构造所希望的功能而需要的子系统和构件的清单。

在10.5.2节“集成系统”中，我们将描述一种创建集成构造计划的系统化方法。

10.4 工作人员

10.4.1 工作人员：构架设计师

在实现阶段，构架设计师负责实现模型的完整性，并保证模型在整体上是正确的、一致的和可读的。同分析和设计一样，对于那些大型复杂的系统来讲，需要一个单独的工作人员来负责实现模型顶层子系统（也就是实现系统）的实现。

当实现模型实现了那些且仅仅是那些在设计模型以及所有（相关的）补充需求中所描述的功能时，就说该模型是正确的。

构架设计师还对实现模型的构架负责，即对该模型的构架视图中描述的对构架有重要意义的部分的存在性负责。请牢记，该视图是系统构架描述的一部分（参见图10-13）。

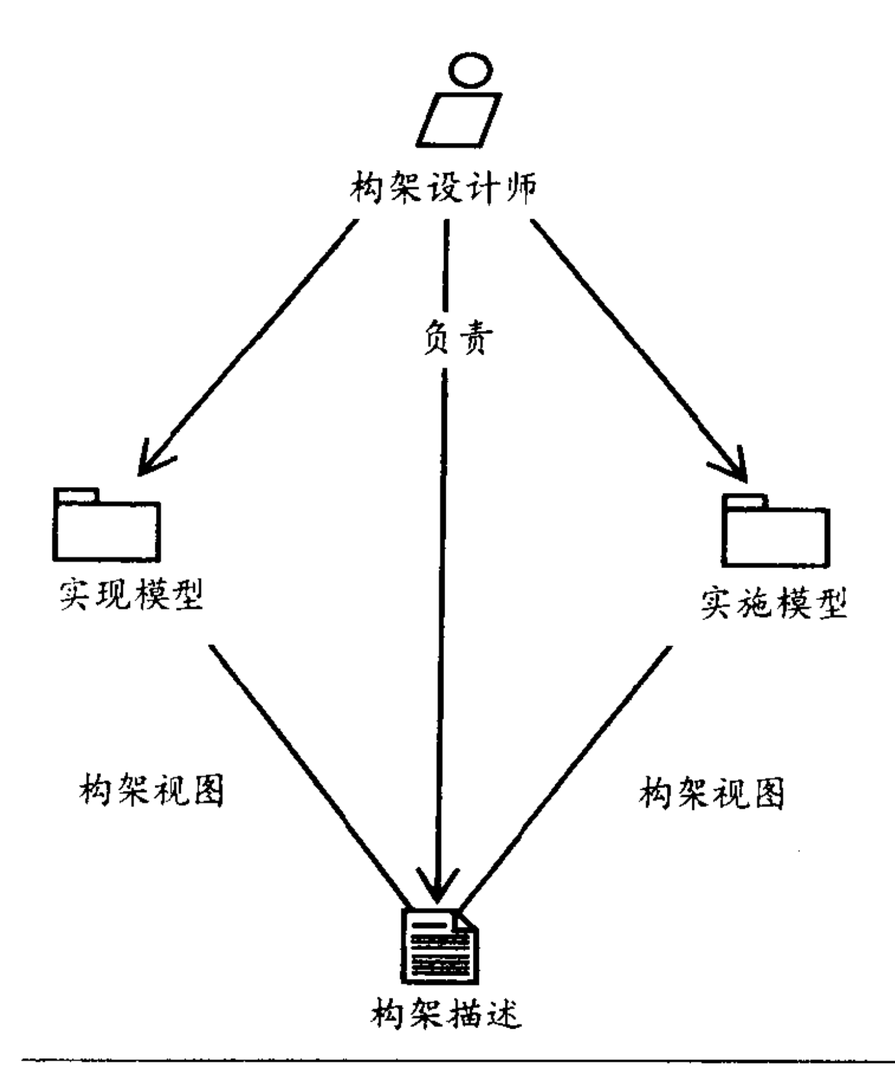


图10-13 构架设计师在实现中的职责

最后，实现的一个重要结果就是将可执行构件映射到节点上。构架设计师负责这种映射，这种映射在实施模型的构架视图中进行了描述。具体细节请参考第9章，9.3.7节“实施模型”。

注意，构架设计师并不负责实现模型范围内不同制品的继续开发和维护，这些工作由相应的构件工程师负责完成。

10.4.2 工作人员：构件工程师

构件工程师定义和维护一个或多个文件构件的源代码，确保每个构件实现正确的功能（如设计类所阐述的功能）。

构件工程师通常还要维护一个或多个实现子系统的完整性。因为实现子系统与设计子系统是一一对应的，所以这些子系统的大多数变化在设计期间就加以了处理。但是，构件工程师需要保证实现子系统的内容（如构件）是正确的，需要保证它们对其他子系统或接口的依赖性是正确的，并需要保证它们正确地实现了它们所提供的接口。参见图10-14。

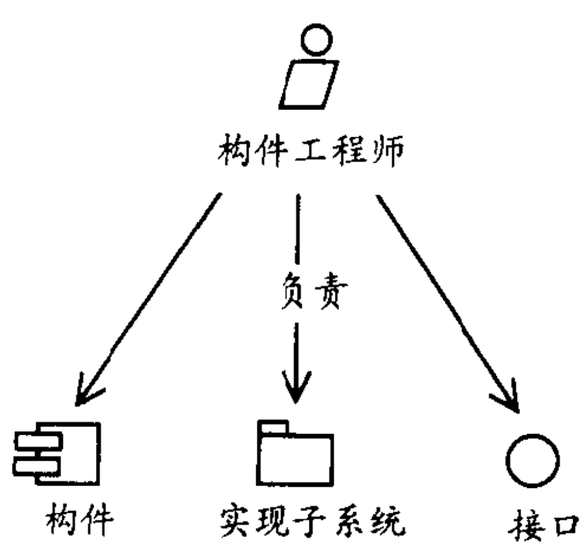


图10-14　构件工程师在实现中的职责

对于负责一个子系统的构件工程师来讲，他通常也适宜于负责这个子系统所包含的模型元素（例如，构件）。而且，为获得一个平滑无缝的开发，由一个构件工程师在整个设计和实现工作流中负责某个子系统及其内容是很自然的。因此，构件工程师应该在他或她的职责范围内，既要设计类，又要实现类。

10.4.3 工作人员：系统集成人员

集成系统的工作超出了构件工程师的职责范围，而应由系统集成人员负责。系统集成人员的

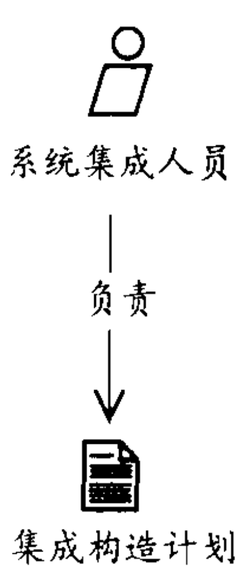


图10-15　系统集成人员的职责

职责包括规划在每次迭代中所需的构造序列，并在每个构造的各部分已经实现时对它们进行集成。规划的结果是一个集成构造计划。参见图10-15。

10.5 工作流

在前面的若干章节中，我们用静态术语描述了实现工作。现在，我们要用活动图来说明动态行为（参见图10-16）。

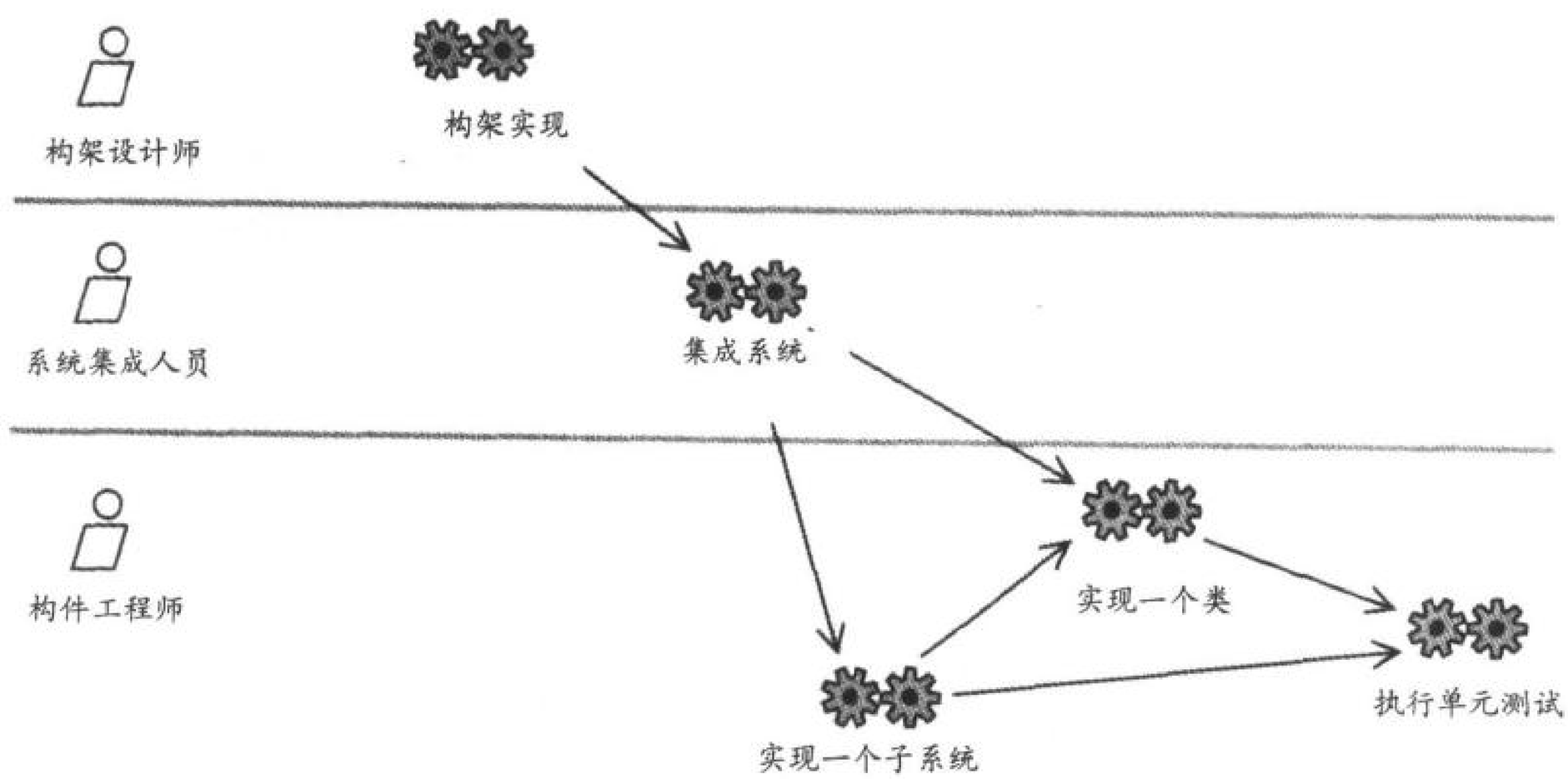


图10-16 实现中的工作流，包括参与的工作人员及其活动

实现阶段的主要目标是实现系统。这是由构架设计师[⊖]通过勾画实现模型的关键构件而启动的。然后，系统集成人员规划当前作为一个构造序列的迭代所需的系统集成。对每一次构造，系统集成人员描述所需实现的功能和将影响实现模型的哪些部分（如子系统和构件）。然后由构件工程师来实现在构造中对子系统和构件的需求；所得到的构件经过单元测试，提交给系统集成人员进行集成。然后，系统集成人员将新的构件集成到某个构造中，并交给集成测试人员人进行集成测试（参见第11章）。然后开发人员开始启动后续构造的实现，并考虑以前构造中的缺陷。

10.5.1 活动：构架实现

构架实现的目的是通过如下途径勾画实现模型及其构架：

- 识别对构架有重要意义的构件，例如可执行构件。
- 在相关的网络配置中将构件映射到节点上。

⊖ 即由许多个人共同合作担当构架设计师这个角色，很可能是一个构架小组。

回想一下，在构架设计过程中（参见第9章，9.5.1节“构架设计”），要勾画设计子系统、它们的内容以及接口。在实现过程中，我们使用与设计子系统一一对应的实现子系统，并提供相同的接口。因此对实现子系统及其接口的识别就显得价值不大，在此就不再讨论。相反，在实现阶段，主要的任务是在实现子系统范围内创建实现相应的设计子系统的构件。

在该活动中，构架设计师维护、精化并更新构架描述以及实现模型和实施模型的构架视图。

10.5.1.1 确定对构架有重要意义的构件

通常的实际做法是在软件生命周期的早期确定对构架有重要意义的构件，以便启动实现工作（参见图10-17）。然而，当实现类时，许多构件（特别是文件构件）的初始创建工作相当琐碎，因为这些构件基本上只是提供将实现类包装成源代码文件的一种工具。由于这个原因，开发人员必须注意在这一阶段不能确定太多的构件或作过分详细的设计。否则，当实现类时，许多工作不得不重做。事实上，只要勾画出一个对构架有重要意义的构件的初始的轮廓就足够了（参见10.3.4节 “构架描述（实现模型的视图）”）。

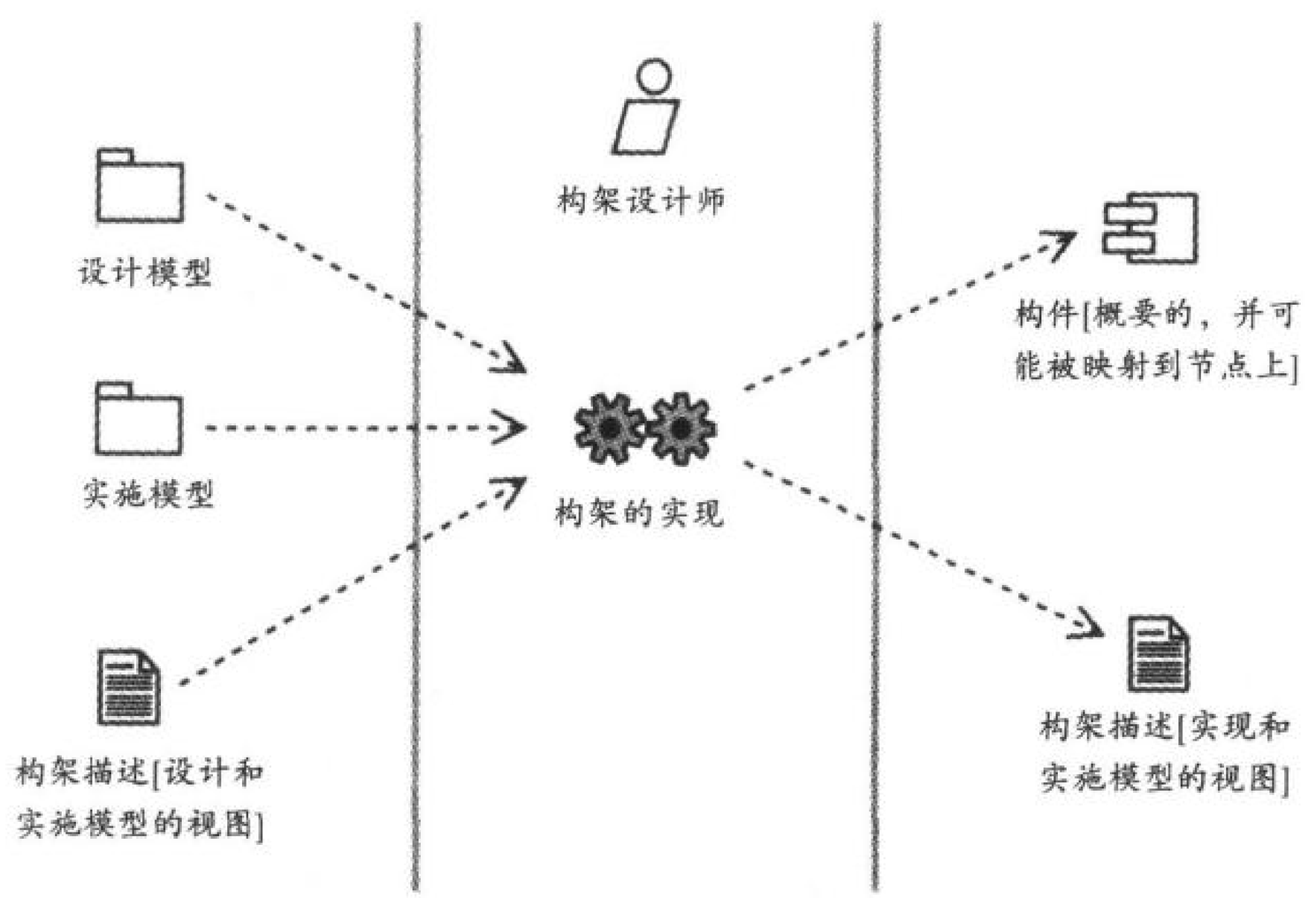


图10-17 构架实现的输入和结果

10.5.1.1.1 确定可执行构件并把它们映射到节点上

为了识别能够实施到节点上的可执行构件，我们要考虑在设计阶段得到的主动类，为每个主动类分配一个可执行构件，并将它标记为一个任务繁重的进程。这也许还要包括确定为创建该可执行构件所需的其他文件或二进制构件，尽管这项工作是相对次要的任务。

举例：确定可执行构件

在设计模型中，有一个叫做“支付请求处理”的主动类。在实现中，我们确定一个相应的可执行构件，叫做“支付请求处理”（参见图10-18）。

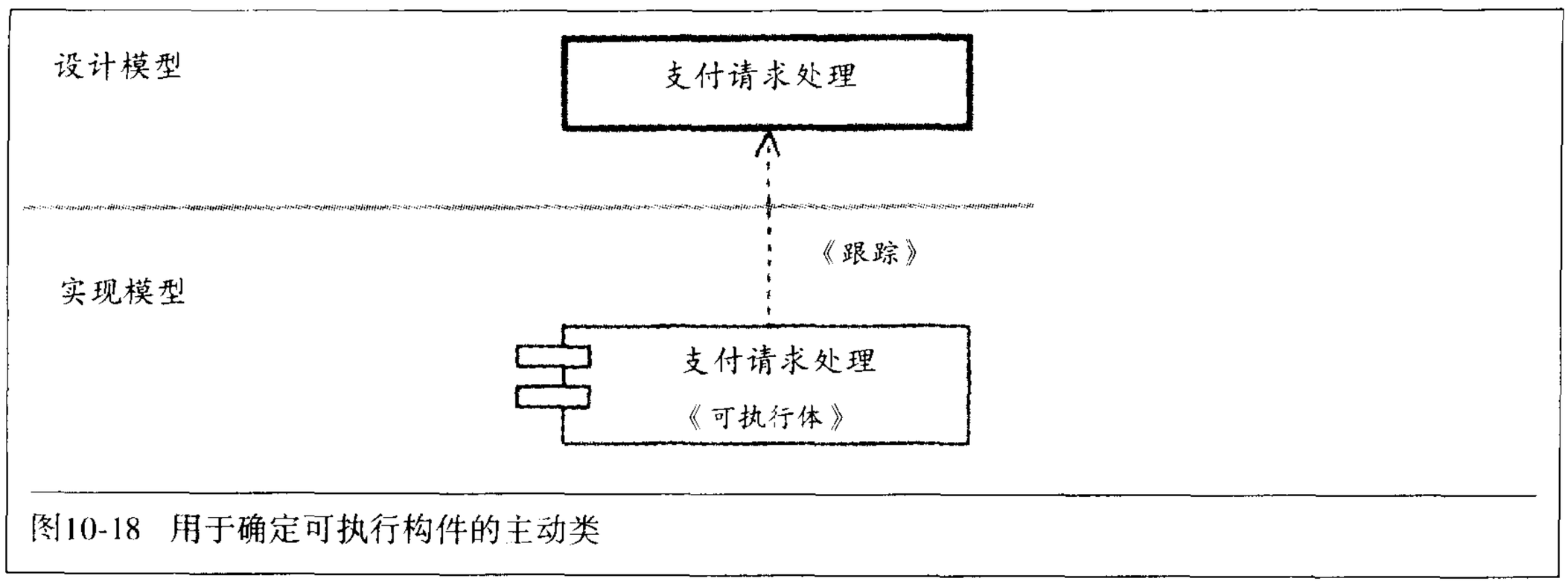


图10-18 用于确定可执行构件的主动类

接着考虑设计模型和实施模型，我们检查是否存在可分配给节点的主动对象。如果有，那么对应于该主动类的相应构件必须实施到相同的节点上。

举例：把构件实施到节点上

“支付请求处理”类的主动对象要被分配到“买主”服务器节点上。

由于“支付请求处理”构件实现“支付请求处理”类（参见上一个例子），因此它也要实施到“买主”服务器节点（参见图10-19）。

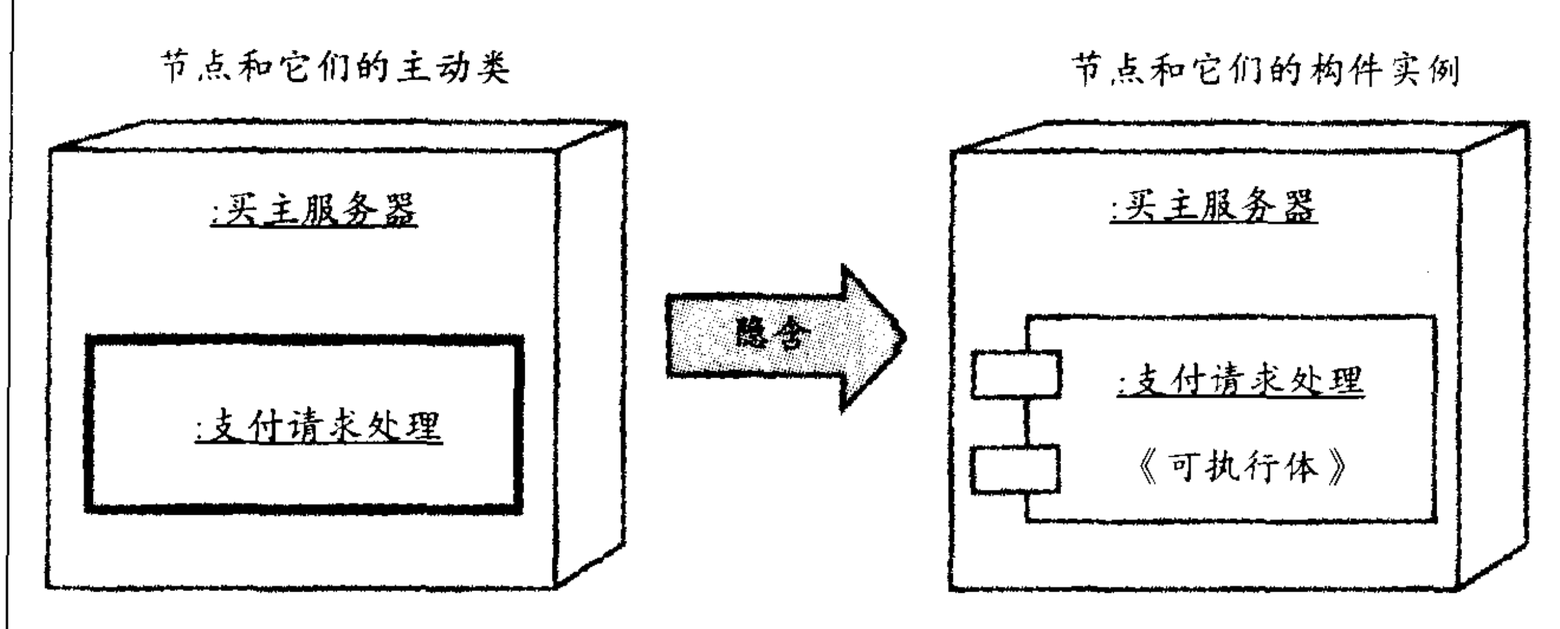


图10-19 分配到节点的主动类隐含相应的构件实施

构件到节点的映射对于系统构架来说非常重要，应该在实施模型的构架视图中加以描述。

10.5.2 活动：系统集成

系统集成的目的是为了：

- 创建集成构造计划，描述迭代中所需的构造和对每个构造的需求（参见图10-20）。
- 在进行集成测试前集成每个构造。

10.5.2.1 规划后续的构造

在这一节中，我们讨论如何规划后续构造的内容，不管是从某个以前的构造开始，还是从零开始。我们还假设已有许多用况或场景（如通过用况的路径）以及其他要在当前迭代中实现

的需求。

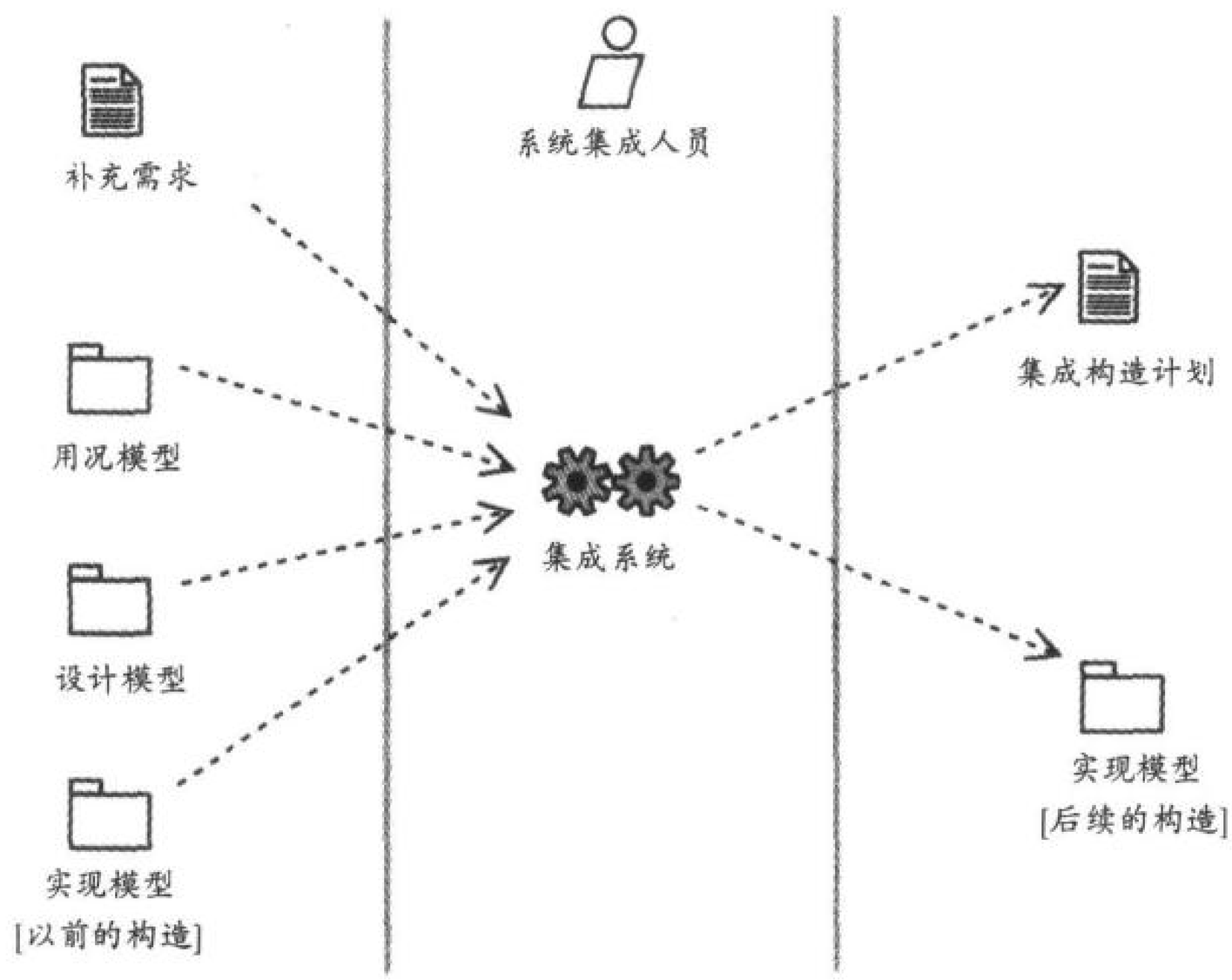


图10-20 系统集成的输入和结果。设计模型中“**用况实现—设计**”是该活动的基本输入。

用于后续构造的一些准则包括：

- 后续的构造应该通过实现完整的用况或它们的场景来向以前的构造添加功能。（为简单起见，在这一节的其余部分我们将用用况（*use case*）代表用况或场景（*use case and/or scenario*））。构造的集成测试要基于这些用况，而测试完整的用况比测试它们的片段要容易（参见第11章）。
- 构造不能包括太多的新增或被精化的构件。否则，很难集成这个构造并执行集成测试。如果必要，可以采用构造中介绍的方法，即可以用桩来实现一些构件，从而将新构件的数量最少化（参见10.3.2.1节“桩”）。
- 这个构造应该以先前的构造为基础，并应向上扩展到层次化的子系统层次结构的边界。这就是说，初始的构造应该在较低层次开始（例如中间件层和系统软件层）。然后，后续的构造向上扩展到通用应用层和专用应用层。这样安排的理由是显而易见的，即在所要求的较低层次上的构件已经存在且已具有正确的功能之前，很难实现在更高层次上的构件。

牢记这些准则，就可以开始评估要实现的需求，比如要实现的用况（或它们的场景）。请注意，在按照一种适当的方式遵循这些准则的过程中，很可能需要一定的折衷办法。例如，实现一个完整的用况（bullet one）可能需要很多新的构件（bullet two），但是如果该用况对于当前的构造来说很重要，则无论如何也要实现它。在任何情况下，重要的是要正确地确定将要在一个构造中实现的需求，而将剩余的需求分配给未来的构造实现。

对于每一个要实现的潜在用况，要按照如下的步骤去做：

1) 通过识别设计模型中相应的**用况实现—设计**来考虑该用况的设计。请回顾通过跟踪依赖关系，**用况实现—设计**可以跟踪到用况（也因此隐含着可以跟踪到该用况的任意场景）。
2) 识别参与**用况实现—设计**中的设计子系统和设计类。
3) 识别出实现模型中可跟踪到第2步中所发现的那些设计子系统和设计类的实现子系统和构件。这些就是在实现用况时所需要的实现子系统和构件。
4) 要考虑实现这些需求时对当前构造顶层的这些实现子系统和构件所带来的影响。请注意，如第3步所描述的那样，这些需求是从设计子系统和类的角度加以详细描述的。按照早前所描述的准则来评估这样的影响是否可以接受。如果可以，就把这个用况的实现安排到下一个构造中；否则，就把它遗留给将来的某个构造。

这些结果应该在制定集成构造计划时捕获，并与具体负责这些受影响的实现子系统和构件的构件工程师保持良好的沟通。然后，构件工程师就可在当前的构造中着手实现这些实现子系统和构件的需求（如10.5.3节“实现一个子系统”和10.5.4节“实现一个类”所描述），并对它们进行单元测试（如10.5.5节“执行单元测试”所描述）。然后，将实现子系统和构件移交给系统集成人员进行集成（参见10.5.2.2节 “集成一个构造”）。

10.5.2.2 集成一个构造

如上述步骤所述，如果对构造做了仔细的规划，构造集成就相当容易。通过汇集合适版本的实现子系统和构件，对它们进行编译，然后把它们连接，就可得到一个构造。请注意，在分层次结构的系统里，编译工作可能需要自下而上地进行，因为从较高层次到较低层次可能存在编译依赖关系。

所得到的构造如果通过了集成测试并且是一个迭代内最后创建的构造，就会安排进行集成测试和系统测试（参见第11章）。

10.5.3 活动：实现一个子系统

实现一个子系统的目的是确保一个子系统履行它在每个构造中的角色，如在集成构造计划中所陈述的。这意味着要保证在构造中要实现的需求（比如场景或用况）以及那些影响子系统的需求能通过子系统内部的构件和其他子系统（递归地）正确地加以实现（参见图10-21）。

10.5.3.1 维护子系统的内容

当通过子系统内部的构件和其他子系统正确地实现了在当前构造中要实现的需求以及那些影响子系统的需求时，一个子系统就实现了它的目标。

即使子系统的内容（例如它所包含的构件）是由构架设计师勾画的，随着实现模型的进一步完善，它仍然需要由构件工程师来精化。这种精化包括以下两个方面：

- 当前构造所需要的相应的设计子系统内的每个类应该由实现子系统中的构件来实现（参见图10-22）。

请注意，如果一个设计子系统包含当前构造所需的其他的设计子系统（递归地），那么其方法是类似的：每个这样包含的设计子系统应该由相应的包含其他实现子系统的实现子系统来实现。

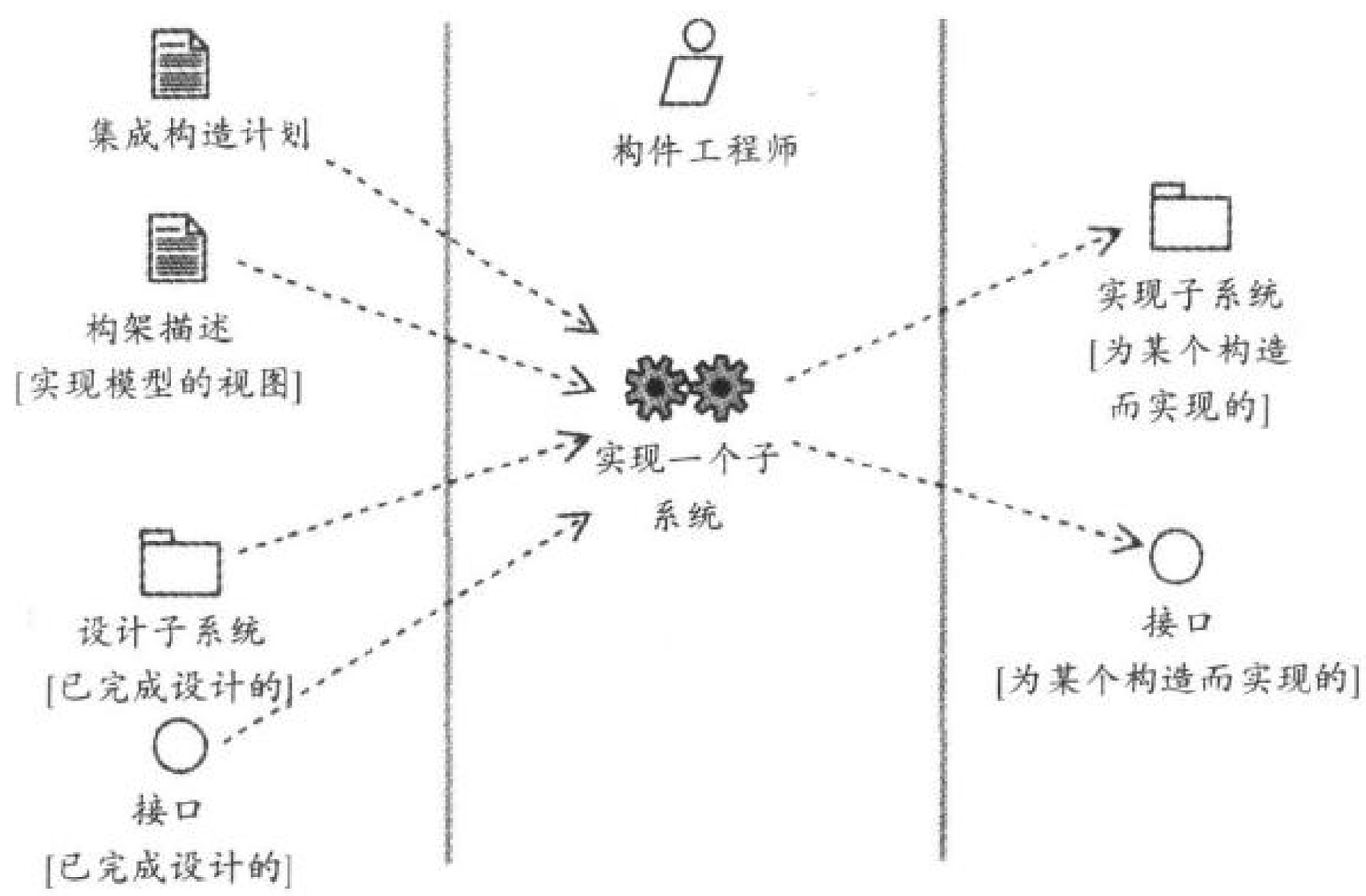


图10-21 子系统实现的输入和结果

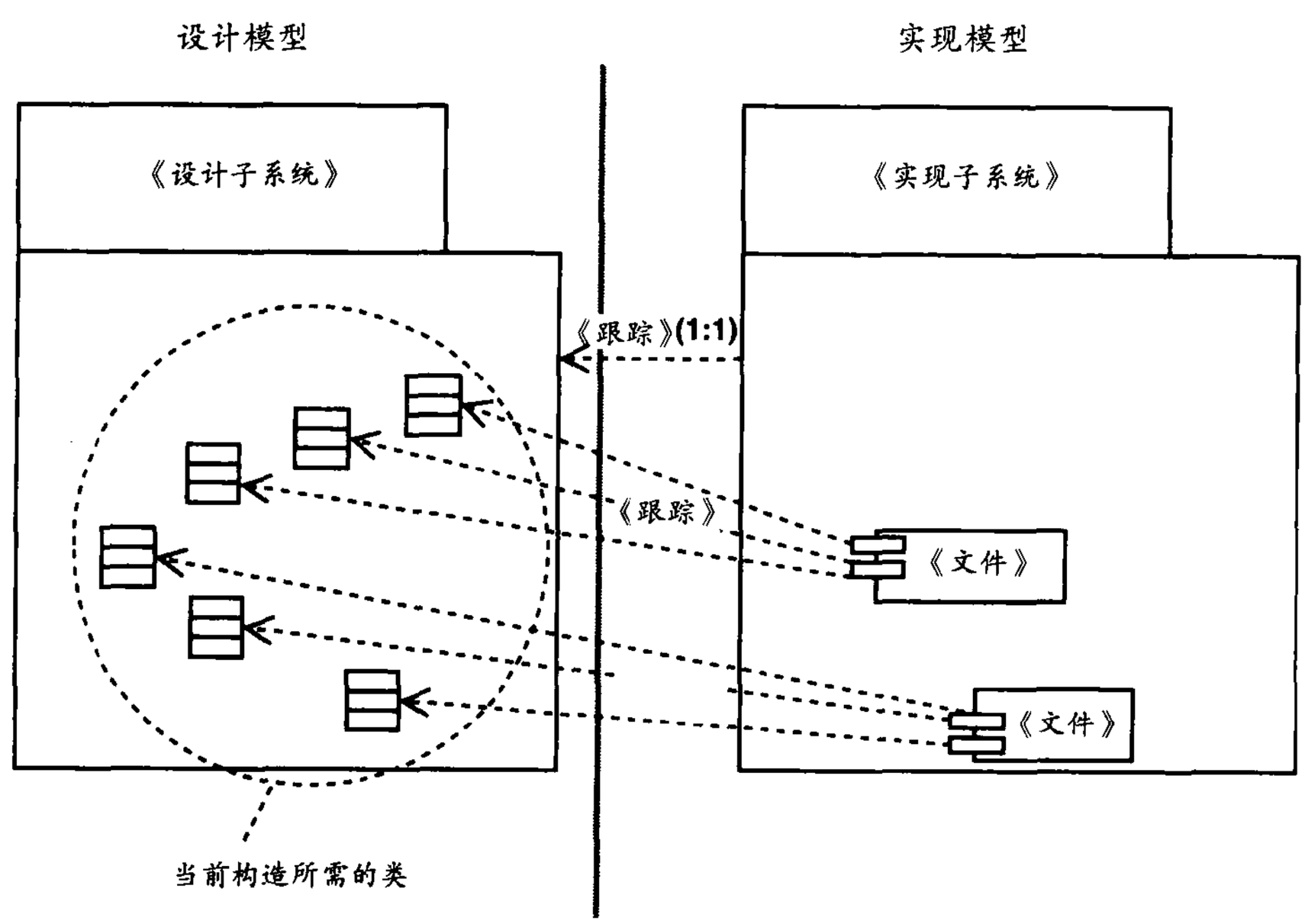


图10-22 通过构件实现在某个构造中所需的设计类

- 当前构造所需的相应的设计子系统所提供的每个接口也应该由实现子系统提供。因此，该实现子系统必须包含提供该接口的构件或实现子系统（递归地）（参见图10-23）。

至此，构件工程师可以开始实现子系统内所需要的构件（如10.5.4节“实现类”所述），并对它们进行单元测试（参见10.5.5节 “执行单元测试”）。

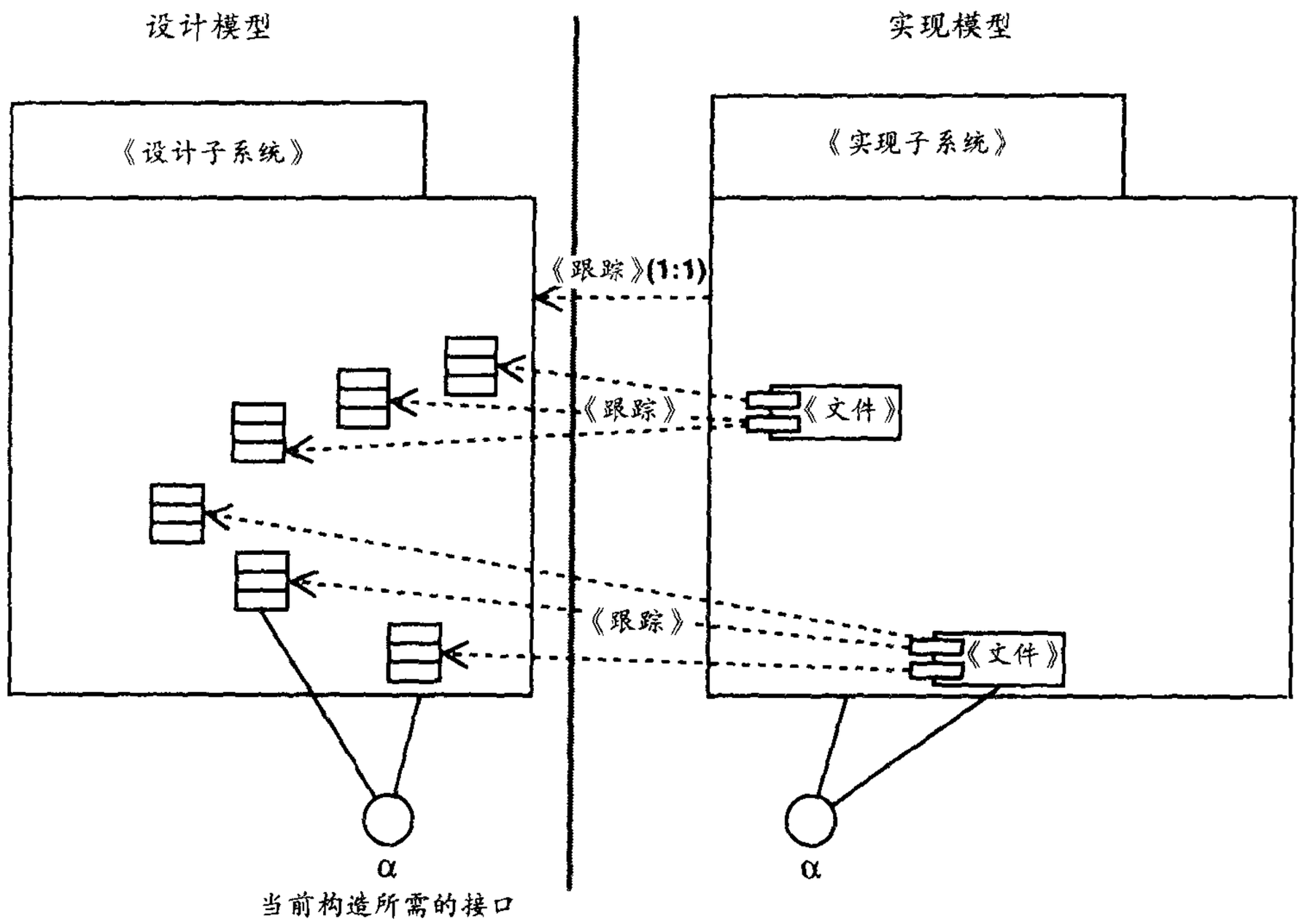


图10-23 在某个构造（α）中所需的接口也应该由实现子系统提供

然后，将所得到的子系统交由系统集成人员进行集成（参见10.5.2节“集成系统”）。

举例: 提供接口的子系统

“买主账单管理”子系统在当前构造中需要提供“账单”接口。负责该子系统的构件工程师决定由“账单处理”构件实现这个接口（参见图10-24）。

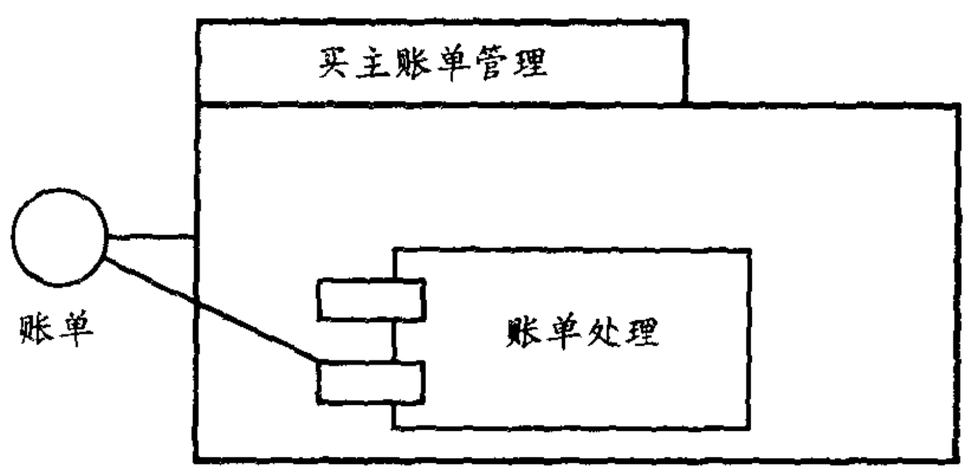


图10-24 “账单处理”构件提供“账单”接口

10.5.4 活动：实现一个类

实现一个类的目的是为了在一个文件构件中实现一个设计类。包含下列任务（参见图10-25）:

- 勾画出将包含源代码的文件构件。

- 从设计类及其所参与的关系中生成源代码。
- 按照方法实现设计类的操作。
- 确保构件提供与设计类相同的接口。

尽管这里没有提到，但该活动还应包括处理已实现类的各方面的维护问题，例如对类进行测试后要进行缺陷修复。

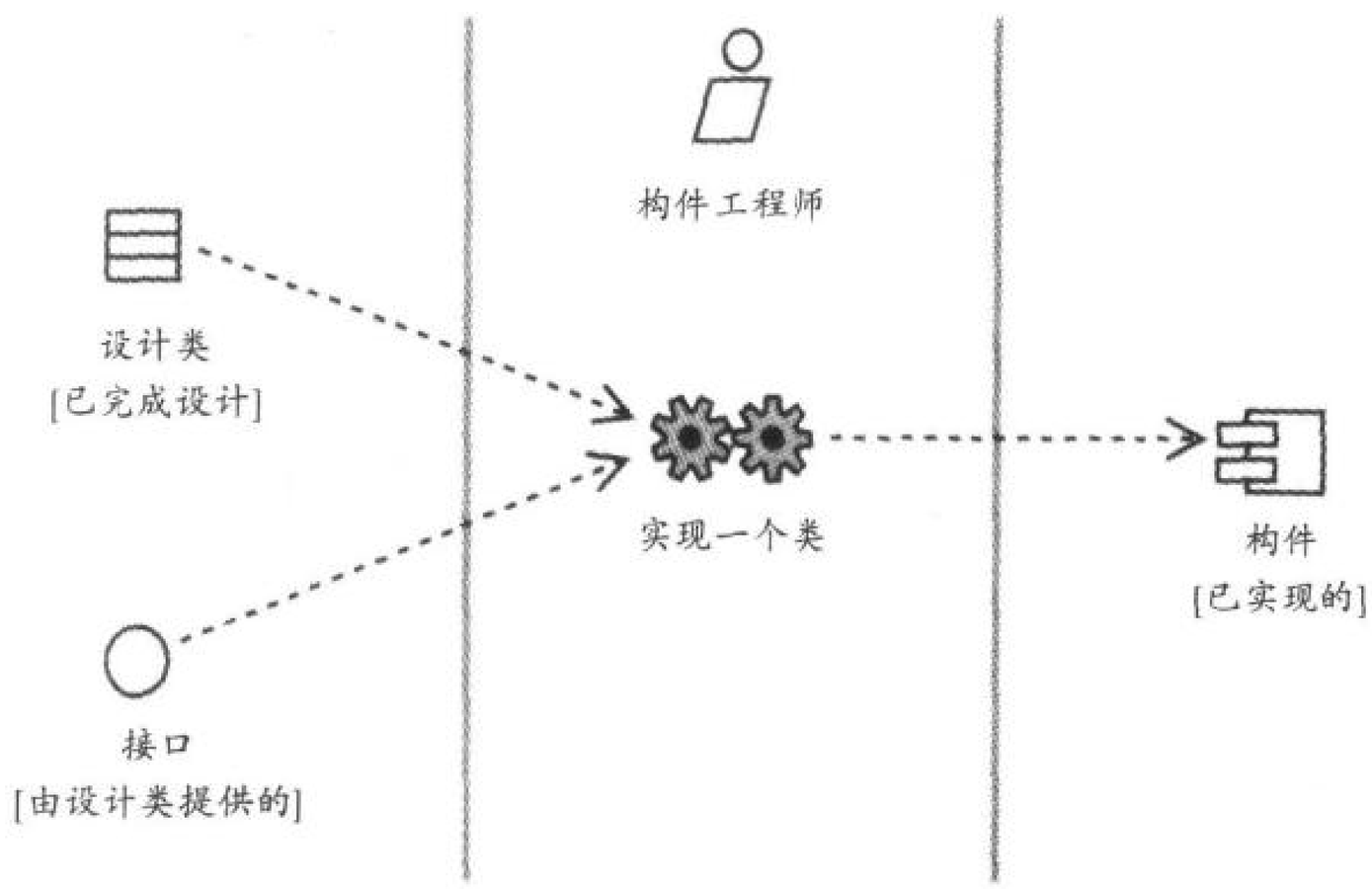


图10-25 实现一个类的输入和结果

10.5.4.1 勾画文件构件

实现设计类的源代码存放在文件构件中。因此必须勾画出文件构件并考虑其作用域。在一个文件中实现几个设计类是很常见的，但是，请注意，正在使用的文件模块化方法和编程语言约定将对如何勾画文件构件加以限制（参见10.3.2节 “构件”）。例如，当使用Java时，要为每个类的实现创建一个“.java”格式的文件构件。一般地，所选的文件构件应该支持系统的编译、安装和维护。

10.5.4.2 从设计类中生成源代码

在设计过程中，设计类及其关系的许多细节都是用编程语言的语法描述的，这就可以生成部分源代码，从而使实现类的工作简单明了。特别是，这种方法还支持类的操作和属性以及类所参与的关系的实现。但是，只生成了操作的特征标记，而操作本身仍有待于实现（参见10.5.4.3节 “实现操作”）。

还应注意，可以非常巧妙地从关联和聚合中产生代码，其实现方法在很大程度上取决于所用的编程语言。例如，具有单向导航特性的关联可用对象间的“引用”来实现；这个引用将作为引用对象的一个属性，而属性名就是关联另一端的角色名。关联另一端的多重性决定（引用的）属性类型应该是一个简单指针（如果多重性小于等于1）或是一个指针集合（如果多重性大于1）[6]。

10.5.4.3 实现操作

在设计类中定义的每个操作都必须实现，除非该操作是“虚拟的”（或“抽象的”）并且这样的操作应由该类的后裔（例如子类型）实现。我们用术语“方法”来表示操作的（method）实现[5]。在实际存在的文件构件中，方法的实例有Java中的*方法*（*method*）[2]、Visual Basic中的*方法*（*method*）[3]和C++中的*成员函数*（*member function*）[4]。

实现一个操作包括选择合适的算法和支持的数据结构（例如方法的局部变量），然后对算法所需的动作进行编码。请回忆一下，在设计类的设计过程中，方法可能已经用自然语言或伪代码描述（尽管这很少见并经常是浪费时间，参见9.5.3.6节），但是，任何“设计方法”当然应该用作这一步的输入。同时注意，任何为设计类描述的状态会影响到实现操作的方式，因为在收到消息时，设计类的状态决定其行为。

10.5.4.4 使构件提供正确的接口

所得到的构件应该提供和它所实现的设计类相同的接口。其示例可见10.3.2节“构件”。

10.5.5 活动：执行单元测试

执行单元测试的目的是为了把已实现的构件作为个体单元进行测试（参见图10-26）。主要有以下几类单元测试：

- **规格说明测试**（*specification testing*）或“黑盒测试”，验证单元外观上可观察的行为。
- **结构测试**（*structure testing*）或“白盒测试”，验证单元的内部实现。

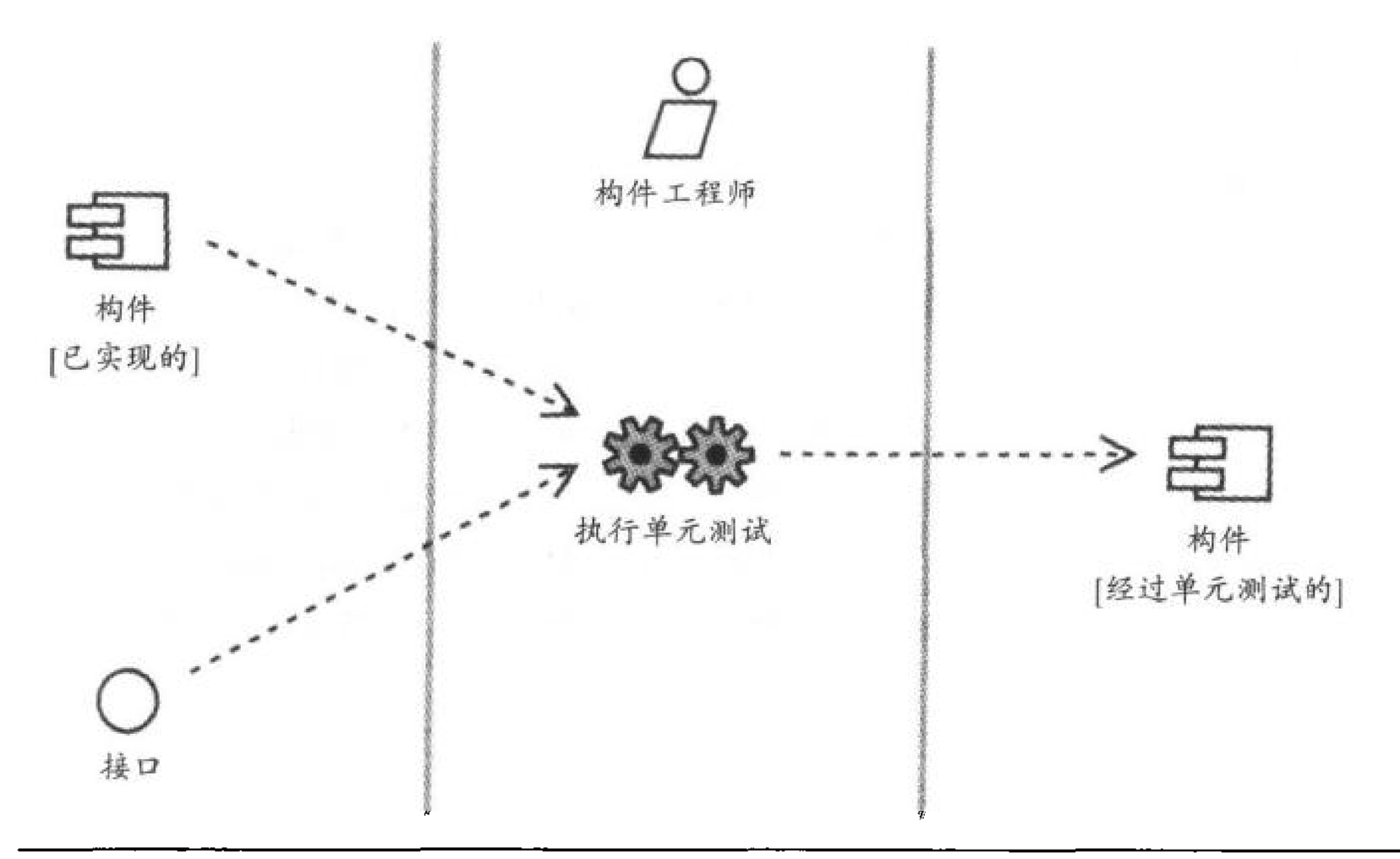


图10-26 单元测试的输入和结果

注意，对于某些单元，可能还要执行许多其他类型的测试，比如性能、内存使用情况、负载和容量等方面的测试。同时，还必须执行集成和系统测试，以保证将几个构件集成在一起后可以正确地工作（参见第11章）。

10.5.5.1 执行规格说明测试

规格说明测试是在不考虑构件内部如何实现的情况下验证构件的行为。因此规格说明测试

是某个构件处在特定状态下，给定一个输入，观察返回的结果（输出）。可能的输入、初始状态和输出的组合范围通常是很大的，要测试所有的组合是不可能的。因此可将输入、输出和状态划分为等价类（*eguivalence classes*）。一个等价类是一组输入、状态和输出值，可以预料这个对象具有类似的行为。通过测试一个构件的每个等价类组合的输入、输出和状态，几乎可以取得与测试所有独立值组合时相同的和有效的测试覆盖，从而极大地减少了测试工作量。

举例：等价类

一个账户的状态有三个等价类：余额为零、余额为负（或许是透支）、余额为正。同样地，输入变量可以分为两个等价类：零和正数。输出变量也可分为两个等价类：取款额为正或者取款额为零。

构件工程师可以基于试探法选择出下列的测试值：

- 每个等价类允许的范围内的正常值，例如从账户中支出4、3.14、5 923美元；
- 等价类的边界值，例如取款为0、最小正值（如0.000 000 01）和最大的可能值；
- 等价类合法边界之外的值，例如取出比合法值更大或者更小的数目；
- 非法值，例如取款值为 - 14和*A*。

选择测试时，构件工程师应该力求覆盖输入状态和输出的所有组合，例如从下列状况中提取14美元：

- 账户中有 - 234.13美元，结果为取出0美元；
- 账户中有0美元，结果为取出0美元；
- 账户中有13.125美元，结果为取出0美元；
- 账户中有15美元，结果为取出14美元。

这四个测试用例的净结果是在所有合法的状态（余额为正和余额为负）和输出（取款额为正和取款额为零）等价类组合中，从每个等价类中取出一个值进行测试。然后，构件工程师应该选择具有类似状态（可能是 - 234.13、0、3和15美元）和输出值（0和14美元），但从同一个输入等价类中取出不同值（如3.14美元）的组合所组成的测试用例进行测试。

然后，构件工程师准备从输入值的其他等价类取值，组成类似值域的测试用例进行测试。例如可以试图从输入值域中取出0、4、3.14、5 923、0.000 000 01、37 000 000 000 000 000 000 000 000 (如果这是最大的可能值)、37 000 000 000 000 000 000 001、 - 14和 *A*美元等值进行测试。

10.5.5.2　执行结构测试

结构测试是有意识地验证构件的内部工作。在结构测试过程中，构件工程师一定要测试所有代码。这意味着每条语句至少要执行一次。构件工程师还应保证测试代码中绝大部分感兴趣的路径，包括最常执行的路径、最关键的路径、某算法中最少了解的路径以及那些有高度风险的路径。

举例：一个方法的Java源代码

图10-27是一个由“账户”类所定义的取款方法的（简化的）实现。

当测试这些代码时，必须确保所有if 语句的true 和 false 分支都判断过，并且所有代码都执行过。例如，我们可以执行如下测试：

- 从一个余额为100美元的账户中取款50美元时，系统执行第10~13行；
- 从一个余额为10美元的账户中取款 - 50美元，系统执行第21行；
- 从一个余额为10美元的账户中取款为50美元，系统执行第19行；
- 当系统执行语句*balance = balance - amount* 时，触发一个异常，系统执行第14~17行。

```
public class Account {
1     // In this example the Account has a balance only
2     private Money balance = new Money (0);
3     public Money withdraw(Money amount) {
4                 // First we must ensure that the balance is at least
5                 // as big as the amount to withdraw
6                 if balance >= amount
7                 // Then we check that we will not withdraw negative amount
8                         then { if amount >= 0
9                                 then {
10                                        try {
11                                                balance = balance – amount;
12                                                return amount
13                                                }
14                                        catch (Exception exc) {
15                                        // Deal with failures reducing the balance
16                                        // … to be defined …
17                                }
18                        }
19                        else {return 0}
20                }
21                else {return 0}
22        }
  }
```

图10-27 为由“账户”类所定义的一个简单的取款方法而编写的Java代码

10.6 实现小结

实现阶段的主要结果是实现包含以下元素的实现模型：

- 实现子系统及其依赖关系、接口和内容。
- 构件（包括文件构件和可执行构件）以及它们之间的依赖。构件经过了单元测试。
- 实现模型的构架视图，包括对构架有重要意义的元素。

当可执行构件被映射到节点时，实现还对实施模型的构架视图进行了仔细的精化和改进。

正如我们将在下一章所述，实现模型对于以后的测试活动来说是主要的输入。更确切地说，在测试期间，每个特定的来自实现的构造都经过了集成测试，也有可能还经过了系统测试。

10.7 参考资料

[1] IEEE, Std 610.12-1990.

[2] Ken Arnold and James Gosling, *The Java™ Programming Language*, Reading, MA: Addison-Wesley, 1996.

[3] Anthony T. Mann, *Visual Basic5—Developer's Guide*, Indianapolis, IN: SAMS Publishing, 1997.

[4] Bjarne Stroustrup, *The C++ Programming Language*, Third Edition, Reading, MA: Addison-Wesley, 1997.

[5] The Unified Modeling Language for Object-Oriented Development, Documentation set, ver. 1.1, Rational Software Corp., September 1997.

[6] James Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design*. Englewood Cliffs, NJ: Prentice Hall, 1991.

第11章 测　试

11.1 引言

在测试工作流中，通过测试每一个构造（包括内部构造、中间构造以及将要向外部发布的系统最终版本）来验证实现的结果。[2]中提供了对测试的一个很好的一般性综述。更明确地讲，测试的目的是为了：

- 规划每一次迭代需要的测试工作，包括集成测试和系统测试。迭代中的每一个构造都需要作集成测试，而系统测试仅需在迭代结束时进行。
- 设计和实现测试，采取的方法是创建用来详细说明要测试什么的测试用例，并创建用来详细说明如何执行测试的测试规程，如果可能，还要创建用来使测试自动化的可执行的测试构件。
- 执行各种测试并系统地处理每个测试的结果。发现有缺陷的构造要重新测试，甚至可能要送回给其他核心工作流（如设计和实现），这样才能修复严重的缺陷。

在本章中，我们将阐明如何开展测试工作以及哪些工作人员和制品会参与到其中（参见图11-1）。所采取的测试工作流与实现工作流几乎相同。

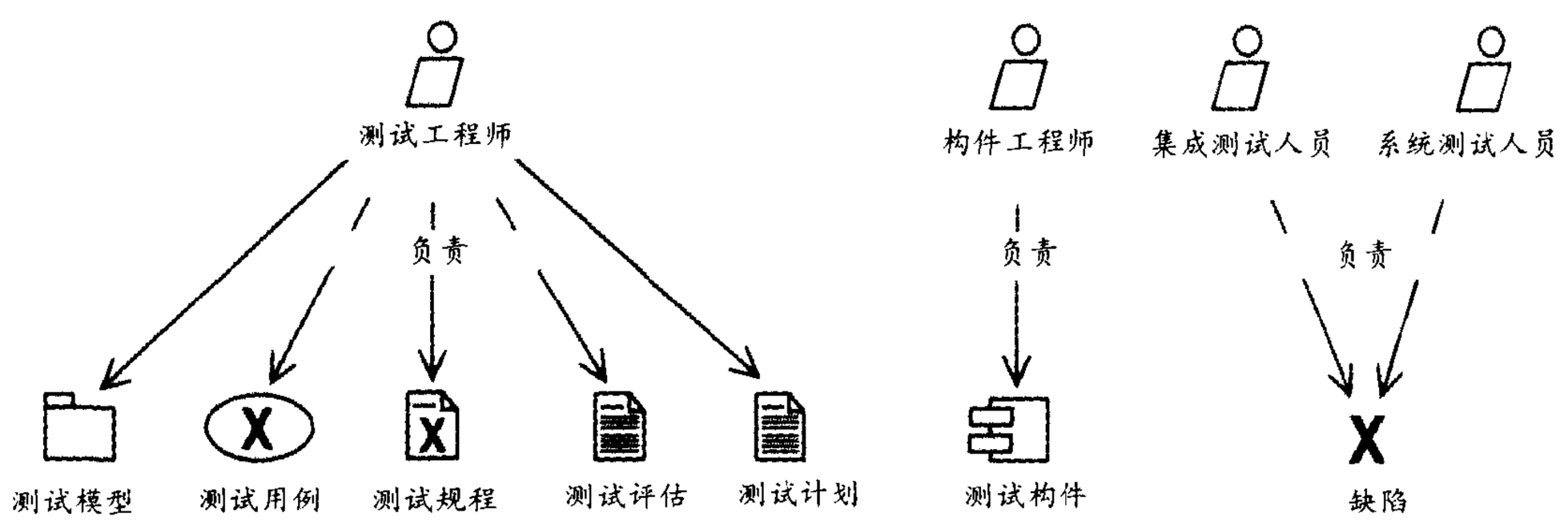


图11-1　参与测试的工作人员和制品

11.2 测试在软件生命周期中的作用

当确定系统范围后，就可在初始阶段制定初始的测试计划。然而，测试主要应用在对每个

构造（作为实现结果）进行集成测试和系统测试时。这意味着测试既是细化阶段（此时测试可执行的构架基线）的焦点，也是构造阶段（此时实现了系统的绝大部分）的焦点。在移交阶段，焦点转向修复在早期使用中发现的缺陷，并进行回归测试。参见图11-2。

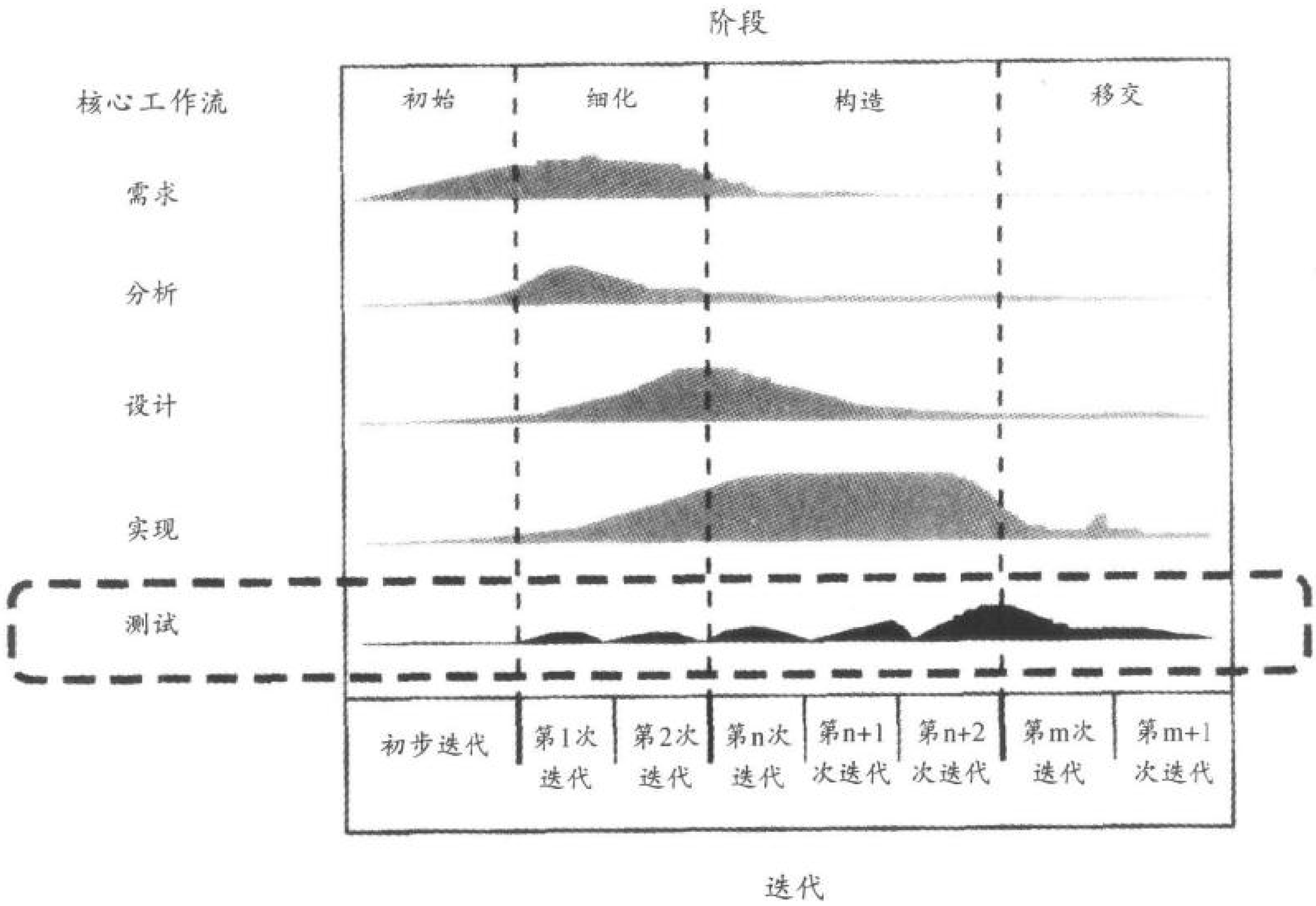


图11-2 测试的焦点

由于开发工作的迭代性，因此一些说明如何测试早期构造的测试用例也可用作说明如何测试后续构造的回归测试用例。在迭代中对回归测试的需要逐步增长，这意味着后期迭代将包括大量的回归测试。因此，虽然测试模型会通过如下方式不断进化，但是在整个软件生命周期中仍要很自然地维护测试模型。测试模型的进化方式包括以下几种：

- 移走过期的测试用例（以及相应的测试规程和测试构件）。
- 把一些测试用例改造成回归测试用例。
- 为每一个后续构造创建新的测试用例。

11.3 制品

11.3.1 制品：测试模型

测试模型主要描述怎样通过集成测试和系统测试对实现模型中的可执行构件（如构造）进行测试。测试模型还能描述如何测试系统的某些特殊方面（比如，用户界面是否可用和协调，

或者系统的用户手册是否能达到目的）。测试模型是测试用例、测试规程和测试构件的集合（如图11-3所示）。

测试模型中的制品将在后面几节中详细描述。请注意，如果测试模型很大，包含了大量的测试用例、测试规程或测试构件，那么在模型中引入包来管理其规模是行之有效的。这或多或少是测试模型的一种无关本质的延伸，在本章中不再讨论这方面的内容。

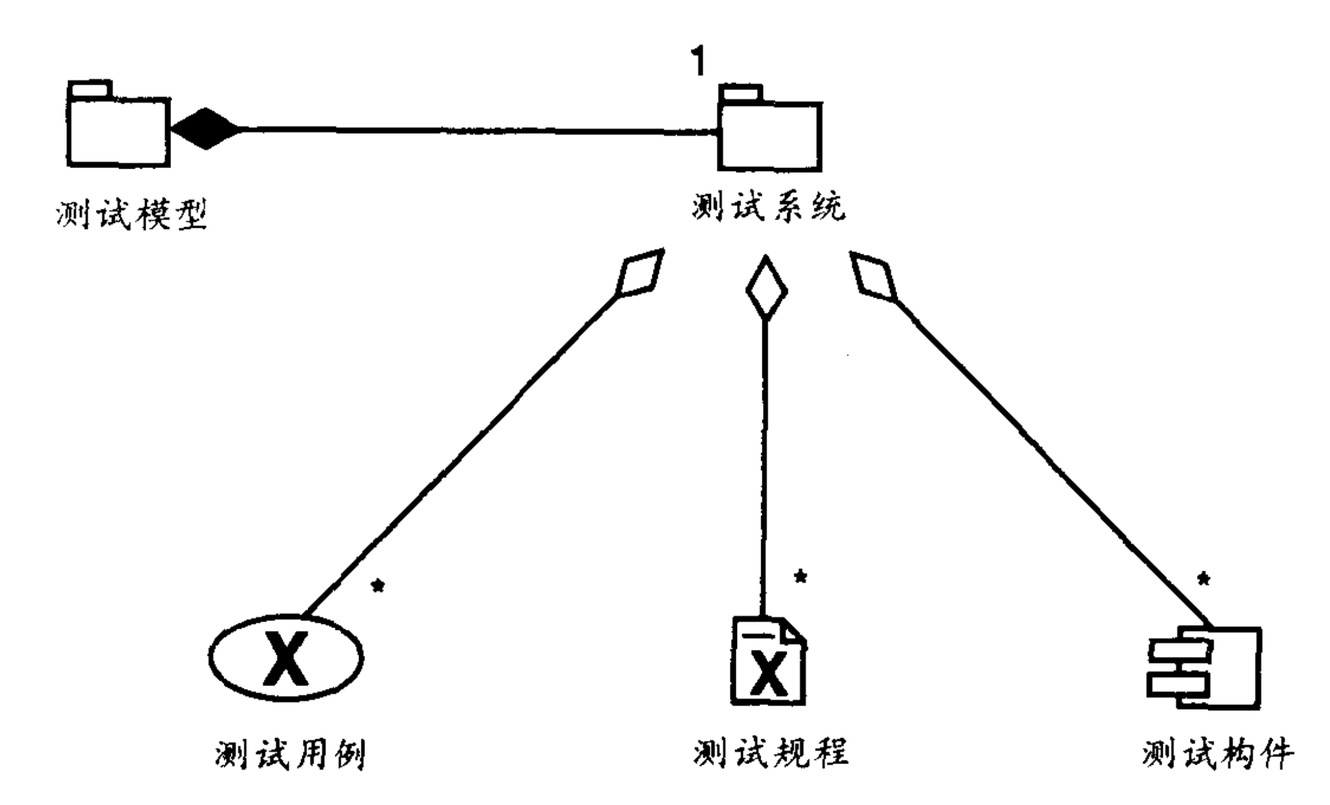


图11-3 测试模型

11.3.2 制品：测试用例

测试用例详细说明测试系统的一种方法，包括用哪些输入或结果去测试什么样的内容，以及在何种条件下进行测试（参见图11-4）。实际上，测试内容可以是任何系统需求或需求集合，它们的实现证明执行一个测试是可能的且代价不会太大。以下是一些常用的测试用例：

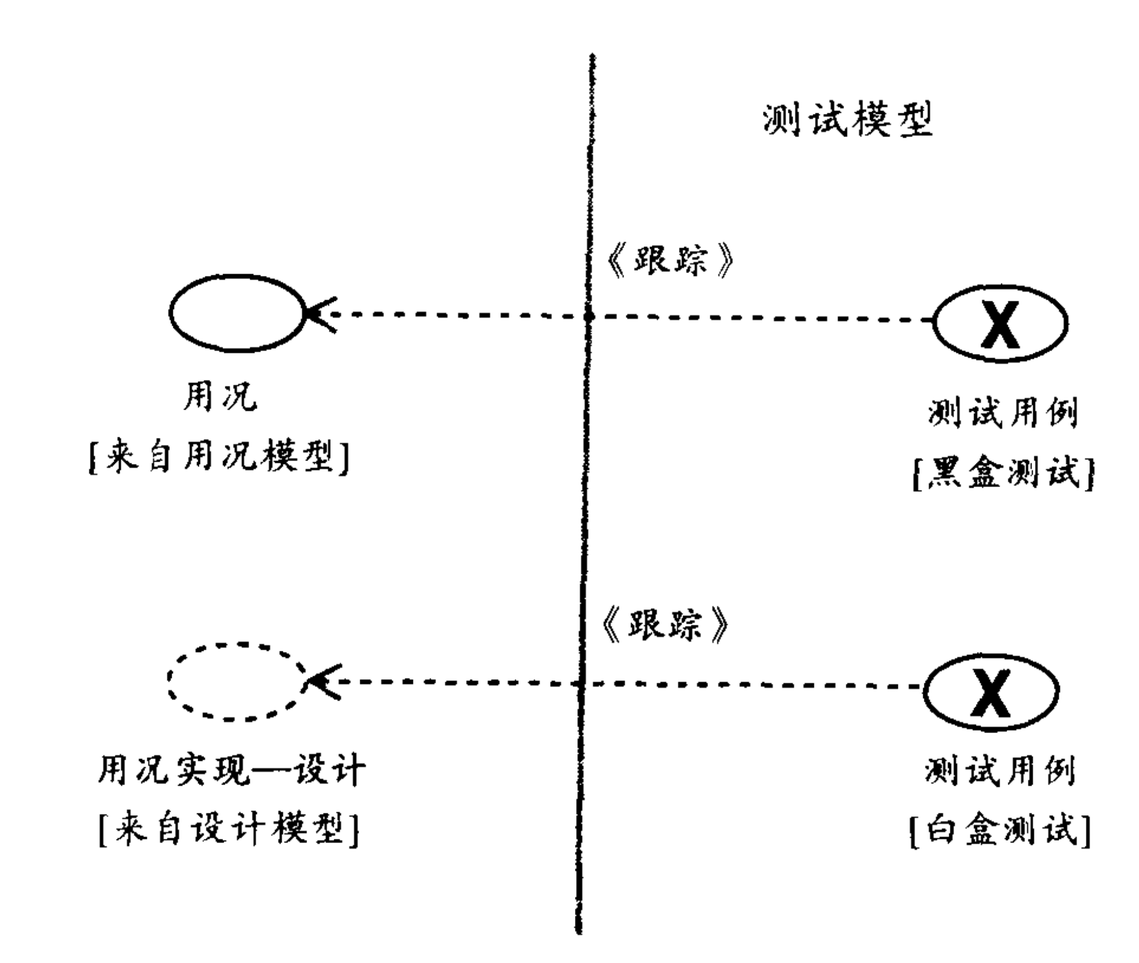


图11-4 测试用例可派生自（因此也可跟踪至）用况模型中的某个用况或设计模型中的某个用况实现

- 详细说明怎样测试一个用况或用况的一个特定场景的测试用例。这样的测试用例包括验证参与者和系统之间的交互结果，验证用况所阐述的前置条件和后置条件是否满足，以及验证是否遵循用况所阐述的动作序列。应注意，基于用况的测试用例典型地阐明了系统的“黑盒”测试（即对系统外观上可观察到的行为的测试）。
- 指出怎样测试一个**用况实现—设计**或该实现的特定场景的测试用例。这样的测试用例可以包括验证实现用况的构件之间的交互作用。应注意，基于用况实现的测试用例典型地阐明了系统的“白盒”测试（即对系统构件之间的内部交互作用的测试）。

举例：测试用例

测试工程师们建议用一系列测试用例来测试“支付账单”用况，每个测试用例验证用况的一个场景。其中的一个测试用例是对一张300美元的山地摩托车订单的账单的支付，他们称这个测试用例为“支付山地摩托车300美元”。

作为一个完整的测试用例，它必须详细说明输入、期望结果以及其他与验证该用况场景相关的条件。

输入

- 一张已存在的且有卖主“Crazy Mountaineer, Inc”签字的山地摩托车的有效订单。摩托车的订价是300美元，包括运输费和送货费。
- 一张买主已收到的山地摩托车的订单确认通知书（ID是98765）。经确认后的摩托车价格是300美元，包括运输费和送货费。
- 一张买主已收到的账单（ID是12345）。账单符合山地摩托车的订单确认通知书，是系统中当前仅有的一张账单。账单上的金额共计300美元，且处于“未决”状态。账单指向卖主的收款账户22-222-2222，该账户的当前余额为963 456.00美元。
- 买主的账户11-111-1111有余额350美元。

结果

- 账单状态应设为“关闭”（表示账单已支付）。
- 买主的账户11-111-1111应显示余额为50美元。
- 卖主账户的余额应增加至963 756.00美元。

条件

- 在实施该测试用例的过程中，不允许其他用况（实例）访问买主和卖主的账户。

要注意，一些测试用例可能相似，仅仅在单个的输入值或结果值上有所区别，这往往是为了支持对同一用况的不同场景的验证。遇到这样的情况，合适的做法是用矩阵来详细说明测试用例，每个测试用例表示成一行，输入值和结果值的值域表示成列。这样一个矩阵能够为相似的测试用例提供良好的概貌，并且为后续的测试规程和测试构件的创建提供有用的输入（参考11.3.3节和11.3.4节）。

可用其他测试用例详细说明系统整体性的测试，例如：

- 安装测试（*installation test*）验证系统能够装在客户的平台上，而且验证该系统在安装时运行正常。

- 配置测试（*configuration test*）验证系统在不同的配置下（比如不同的网络配置下）运行正确。
- 否定测试（*negative test*）试图使系统出错以发现它的弱点。测试工程师们要识别一些特殊的测试用例，这些测试用例试图采取在设计时没有加以考虑的方式来使用系统以发现它的弱点，比如，使用不正确的网络配置，使用不足的硬件容量，或使用一个“不可能的”工作负荷。[⊖]
- 强度测试（*stress test*）识别在遇到资源不足或资源冲突情况下的系统问题。

通过考虑以上的系统用况可以发现许多这样的测试用例，在11.5.2节给出了更多的细节。

11.3.3 制品：测试规程

测试规程（*test procedure*）指出怎样执行一个或几个测试用例或者其中的一部分。比如，测试规程可以是对某个人发出的如何手工执行测试用例的指示，也可以是一份详细说明用来描述如何手工地与自动测试工具交互作用以创建可执行的测试构件（参见11.3.4节）。

怎样执行一个测试用例可以由一个测试规程详细说明，但是通常可对不同的测试用例重用同样的测试规程，并对同一测试用例重用不同的测试规程（参见图11-5）。

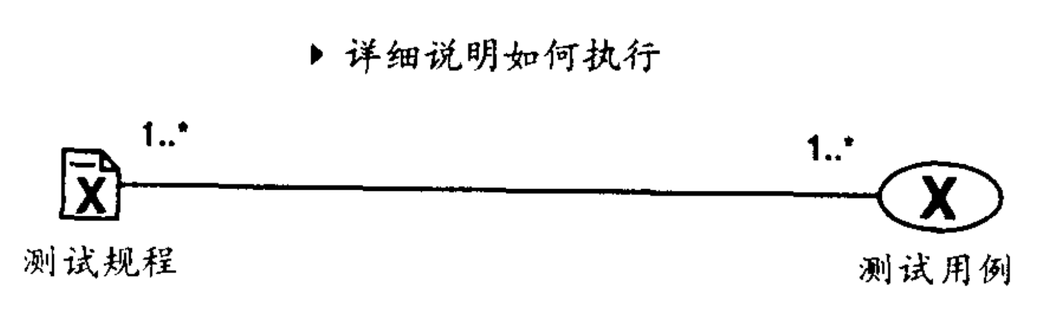


图11-5 测试规程和测试用例之间存在多对多的关联

举例：测试规程

需要有一个测试规程来指导一个人执行上个例子中讨论的“支付山地摩托车300美元”的测试用例（参见11.3.2节）。测试规程的第一部分描述如下（方括号中的文字不包括在说明中，因为它已包括在测试用例中）：

支持的测试用例：“支付山地摩托车300美元”

1) 从主窗口选择“**浏览账单**”（Browse Invoices）菜单。打开“账单浏览查询”对话窗口。
2) 在“账单状态”区，选择“**未决**”（Pending）并点击“**查询**”（Query）按钮。显示“查询结果”窗口。验证在测试用例[ID 12345]中详细说明的账单列在了“查询结果”窗口中。
3) 双击选择要支付的账单。显示所选账单的“账单详细情况”窗口。验证下列情况：

- 状态是未决
- 支付日期为空
- 订单确认序列号与测试用例中规定的序列号[ID 98765]相匹配

⊖ 也叫“滥用用例”。

- 账单金额与测试用例中的金额[300美元]相匹配
- 账户与测试用例中的账户[22-222-2222]相匹配

4) 选择“**准予支付**”（Authorize to Pay）框对该账单的支付进行初始化，显示“支付账单”对话框。

5) 其他。（说明通过用户界面给系统一定的输入是如何执行“支付账单”用况的完整路径，还要说明在系统输出中验证什么。）

请注意，该测试规程同样可以用于其他输入值和结果值不同的类似的测试用例（也就是方括号中的值）。还应注意，该测试规程与“支付账单”测试用例的事件流描述是相似的（参见7.4.3.1节），但是测试规程还包括额外的信息，如从测试规程中选择什么输入数据和怎样把这些数据输入到用户界面以及要验证什么。

举例：通用测试规程

当测试设计人员设计测试规程时，他们注意到一些用况（和用来验证它们的测试用例）的测试是通过对用况初始时的对象进行验证而展开的，比如把接收到的账单同订单确认相比较。

然后测试设计人员就设计一段叫做“验证业务对象”的通用测试规程来测试这一序列。测试规程要求首先应该创建要测试的对象（从一个文件中读取数据，然后在数据库中创建它们）。然后，测试规程规定应该调用用于验证业务对象的主动对象（比如在“支付账单”用况中的“订单处理程序”）。最后，测试规程规定要把验证结果同预期结果（在上面提到过的文件中描述）相比较。

测试设计人员还建议设计一段叫做“验证调度”的测试规程来规定怎样测试账单调度并稍后验证这种调度是否会引发账户的转账。

测试设计人员知道一些用况执行账户转账，因此他们创建一段叫做“验证账户转账”的测试规程来规定怎样验证账户转账。

11.3.4 制品：测试构件

测试构件（*test component*）自动执行一个或几个测试规程或者其中的一部分（参见10.3.2节）。如图11-6所示。

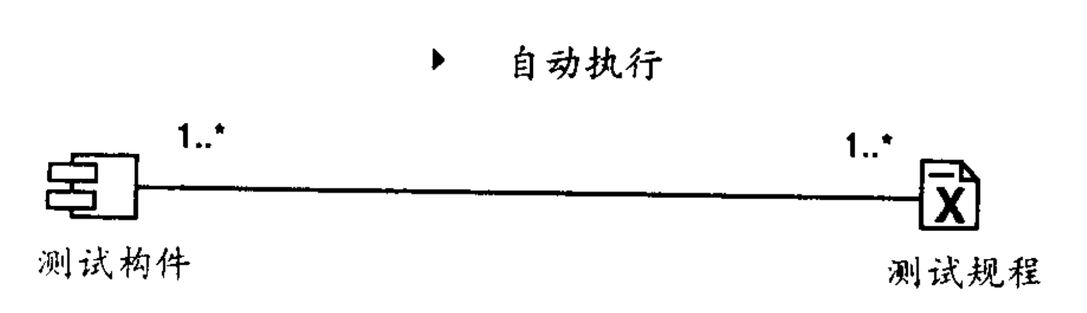


图11-6 测试规程和测试构件之间存在多对多的关联

测试构件可用脚本语言或编程语言来开发，或者通过测试自动化工具来记录。

测试构件通过提供测试输入、控制和监控被测试构件的执行以及尽可能报告测试结果等手段，来对实现模型中的构件进行测试（参见第10章）。测试构件有时也叫测试驱动程序（*test*

driver）、测试道具（*test harness*）[1]以及测试脚本（*test script*）。

请注意，测试构件可以通过对象技术来实现。如果几个测试构件存在复杂的内部交互，或者与实现模型中的普通构件有复杂的交互，则一个单独的“测试设计模型"（与设计模型相似，参见9.3.1节）可用于对测试构件建模并描述它们的高层视图。尽管这样的模型可能有用，但在本书中不打算进一步加以讨论。

11.3.5 制品：制定测试计划

测试计划描述测试策略、资源和进度安排。测试策略包括为每次迭代定义要执行的测试类型和目标、测试和代码覆盖的水平，以及带有某种特定运行结果的测试所占的百分比。

11.3.6 制品：缺陷

缺陷即系统异常，比如软件故障的征兆，或者在复查会议上发现的问题。可用缺陷来捕获开发人员需将其作为系统中问题的征兆而进行登记并跟踪和解决的那些东西。

11.3.7 制品：评估测试

测试评估是对测试工作各项结果的评估，比如测试用例覆盖率、代码覆盖率以及缺陷的状态。

11.4 工作人员

11.4.1 工作人员：测试设计人员

测试设计人员的任务是保证测试模型的完整性，保证其模型实现其目的。测试设计人员还要规划测试，也就是说由他们来决定适当的测试目标和测试进度。而且，测试设计人员要选择和描述测试用例以及相应的测试规程。在测试完成后，他们还要负责对集成和系统测试进行评估。参见图11-7。

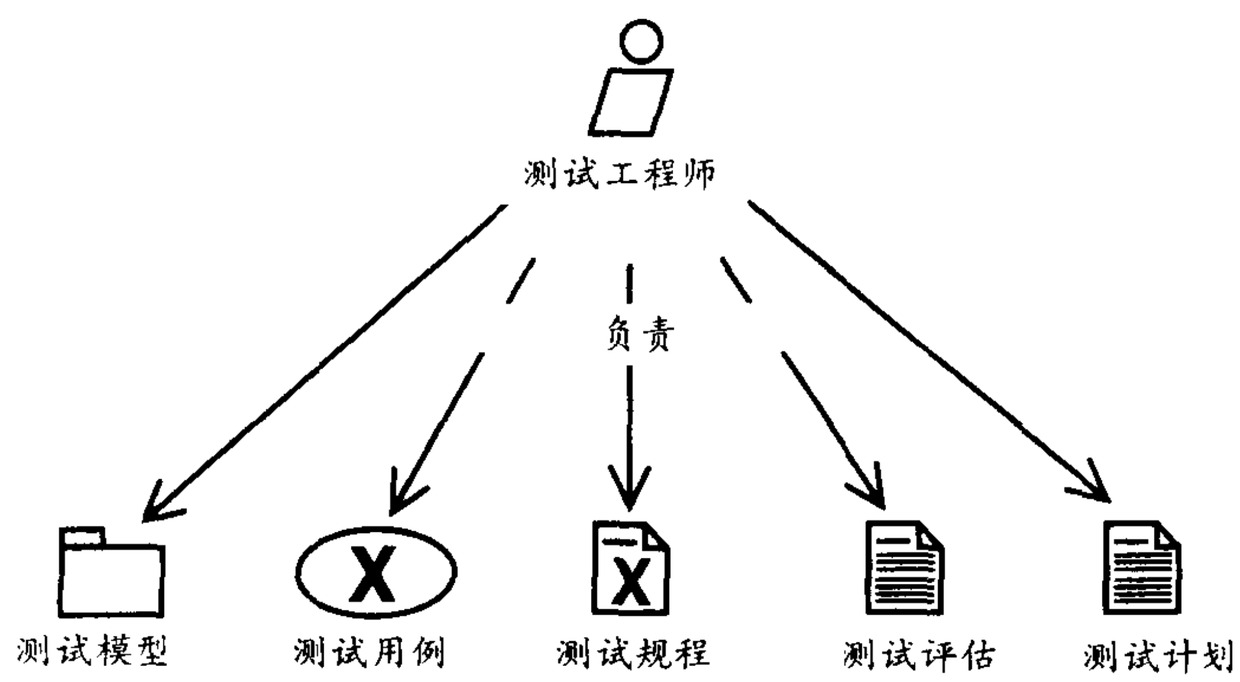


图11-7 测试工作流中一个测试设计人员的职责

应注意，测试设计人员实际上并不执行测试，而是把注意力集中在测试准备和评估上。而由另外两类工作人员（集成测试人员和系统测试人员）执行测试。

11.4.2 工作人员：构件工程师

构件工程师负责测试构件，以便自动执行一些测试规程（并非所有的测试规程都能自动执行）。这是因为这些构件的创建可能需要扎实的编程技能（即构件工程师在实现工作流中也需要的那些技能，参见10.4.2节）

11.4.3 工作人员：集成测试人员

集成测试人员对实现工作流中产生的每一个构造进行集成测试（参见10.5.2节）。集成测试验证那些集成在一个构造中的构件的运行是否正常。因此，集成测试通常由那些说明如何测试**用况实现—设计**的测试用例导出（参见11.3.2 节）。

集成测试结果是由集成测试人员所报告的缺陷。

请注意，集成测试人员所测试的是由系统集成人员在实现工作流中创建的结果（即构造）。因此，有些项目会选择同一员工同时充当这两个角色，以减少所需的知识覆盖。

11.4.4 工作人员：系统测试人员

系统测试人员负责对作为一个完整迭代的（可执行）结果的构造（参见10.5.2节）进行所需的系统测试工作。系统测试主要用于验证参与者与系统之间的交互。因此，系统测试通常由那些详细说明如何测试用况的测试用例导出，但也要采用其他各种类型的测试，以保持系统测试的整体性（参见11.3.2节）。

系统测试结果是由系统测试人员所报告的缺陷。

基于系统测试的特点，担当系统测试的人员并不需要对系统内部运作有深入的了解，而是应该熟悉该系统外观上可观察到的行为。这样，某些系统测试可以由其他项目成员来执行，比如用况说明人员或者甚至可以是外部人员（如beta用户）。

11.5 工作流

在前面章节中，我们用静态术语描述了测试工作。现在用一个活动图（参见图11-8）来说明其动态行为。

测试的主要目的是为了执行并评估测试模型所描述的测试。这是由在每次迭代中规划测试事宜的测试工程师发起的。接着，测试工程师描述所需的测试用例以及执行这些测试用例的测试规程。然后，如果可能，构件工程师要建立起使某些测试规程自动化的测试构件。当每个构造从实现工作流中发布时，就会按上面的步骤进行测试。

用这些测试用例、测试规程及测试构件作为输入，集成测试人员和系统测试人员测试每一

个构造并捕获发现的所有缺陷。接着，将这些缺陷反馈给其他工作流（如设计和实现），还要反馈给测试工程师，以对测试结果进行系统的评估。

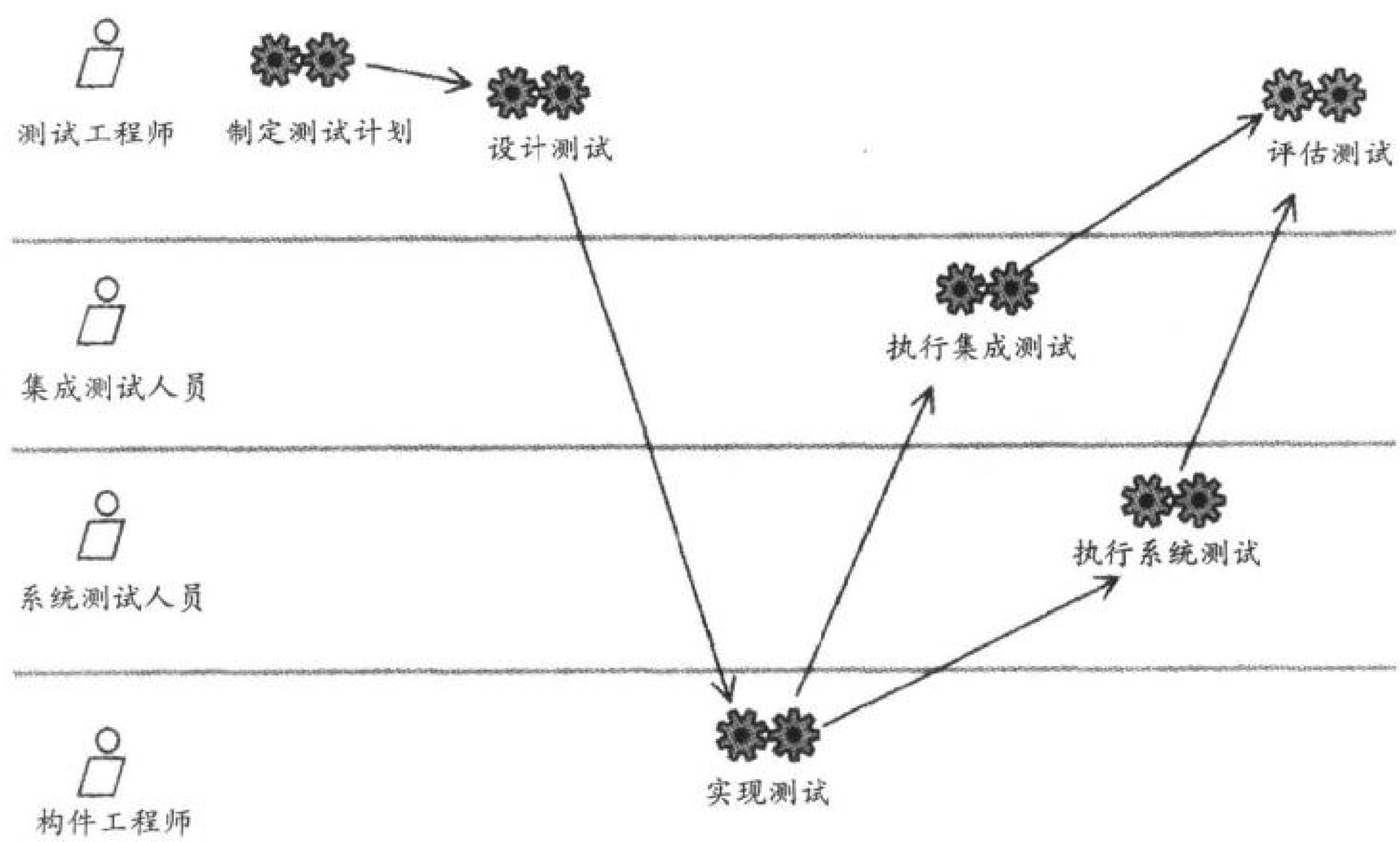


图11-8 测试期间的工作流，包括参与的工作人员及其活动

11.5.1 活动：制定测试计划

制定测试计划的目的是为了按如下方式规划一次迭代中的测试工作（参见图11-9）：

- 描述测试策略。
- 估计测试工作所需的人力以及系统资源等。
- 制定测试工作的进度。

测试工程师在准备测试计划时会利用一系列输入。用况模型和补充性需求帮助他们决定适当的测试顺序，并评估进行测试所需的工作量。测试设计人员也会用其他制品作为输入，比如设计模型。

测试设计人员为迭代开发一套通用测试策略，即进行何种测试，如何进行测试，何时进行测试，以及如何判断测试工作是否成功。

举例：细化阶段中最后一次迭代的系统测试策略

至少75%以上的测试应是自动的，剩下的测试由手工完成。对每一主题用况的正常流程和三种可选流程加以测试。

成功标准：通过90%的测试用例，所有中优先级到高优先级的缺陷都已经修复。

每个测试用例、测试规程和测试构件的开发、执行和评估都需要花费时间和金钱。系统不可能完全被测试。因此，我们应该识别那些在提高质量[3]方面有最好的投资回报的测试用例、

测试规程以及测试构件。一般的准则是：所开发的测试用例和测试规程能以最少的重复来测试最重要的用况并对风险性最大的需求进行测试。

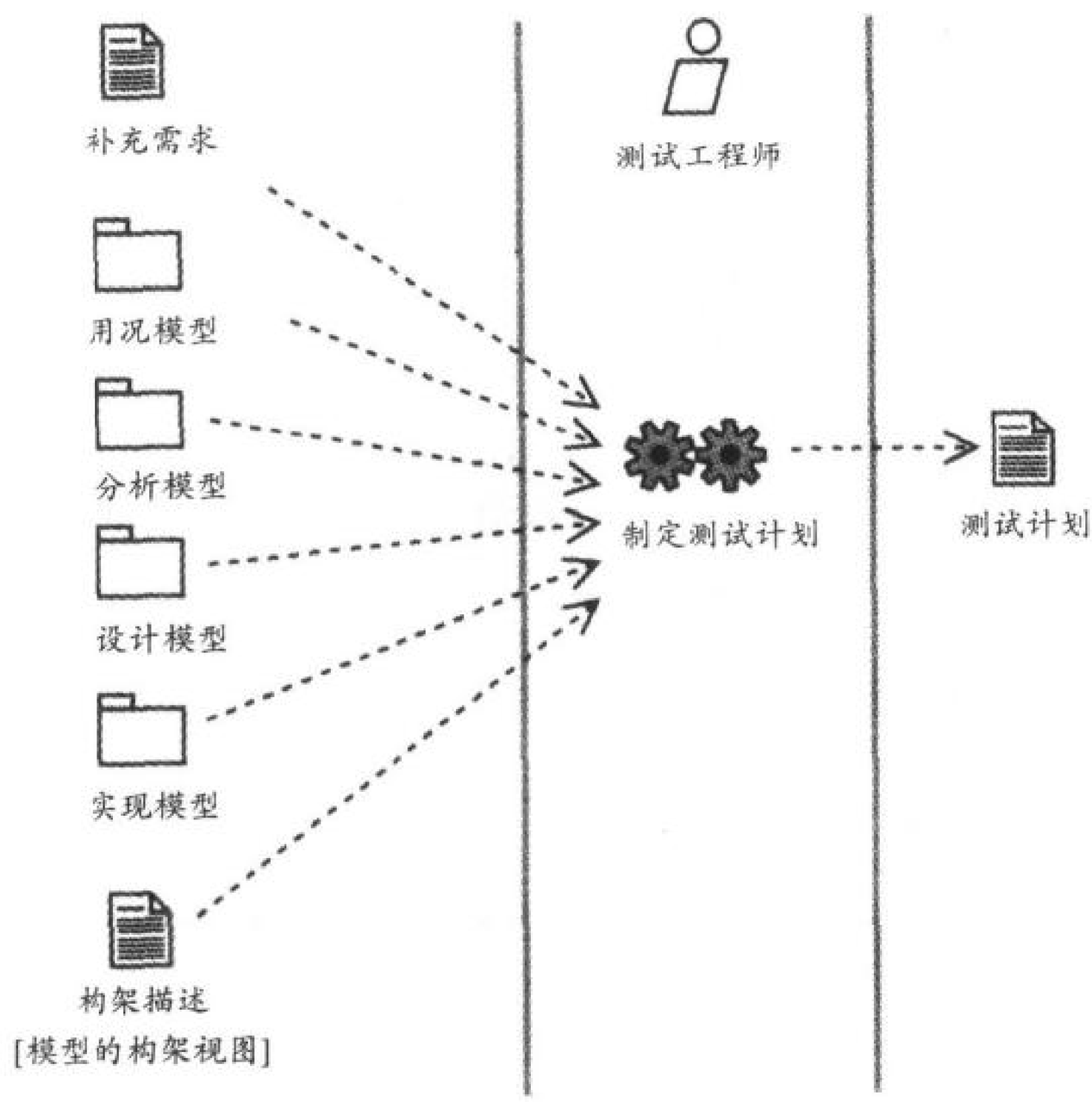


图11-9 制定测试计划的输入和结果

11.5.2 活动：设计测试

设计测试（参见图11-10）的目的是为了：

- 识别并描述每个构造的测试用例。
- 识别并构造用于详细说明如何进行测试的测试规程。

11.5.2.1 设计集成测试用例

集成测试用例用于验证构件被组装成构造后相互之间能否正常交互。大多数集成测试用例可由**用况实现—设计**导出，因为用况实现描述了类和对象（构件亦如此）如何交互。

测试工程师应设计一套测试用例，以便事半功倍地完成测试计划中的目标。为此，测试工程师尽力寻找一套尽量不重叠的测试用例，每一个测试用例都通过用况实现来测试一个感兴趣的路径或场景。

当设计测试人员创建集成测试用例时，他们首先考虑把用况实现的交互图作为输入。测试设计人员寻求一种可产生感兴趣的场景的参与者输入、输出和系统初始状态的组合；这些场景使用了参与到图中的各种类（因此也是构件）。

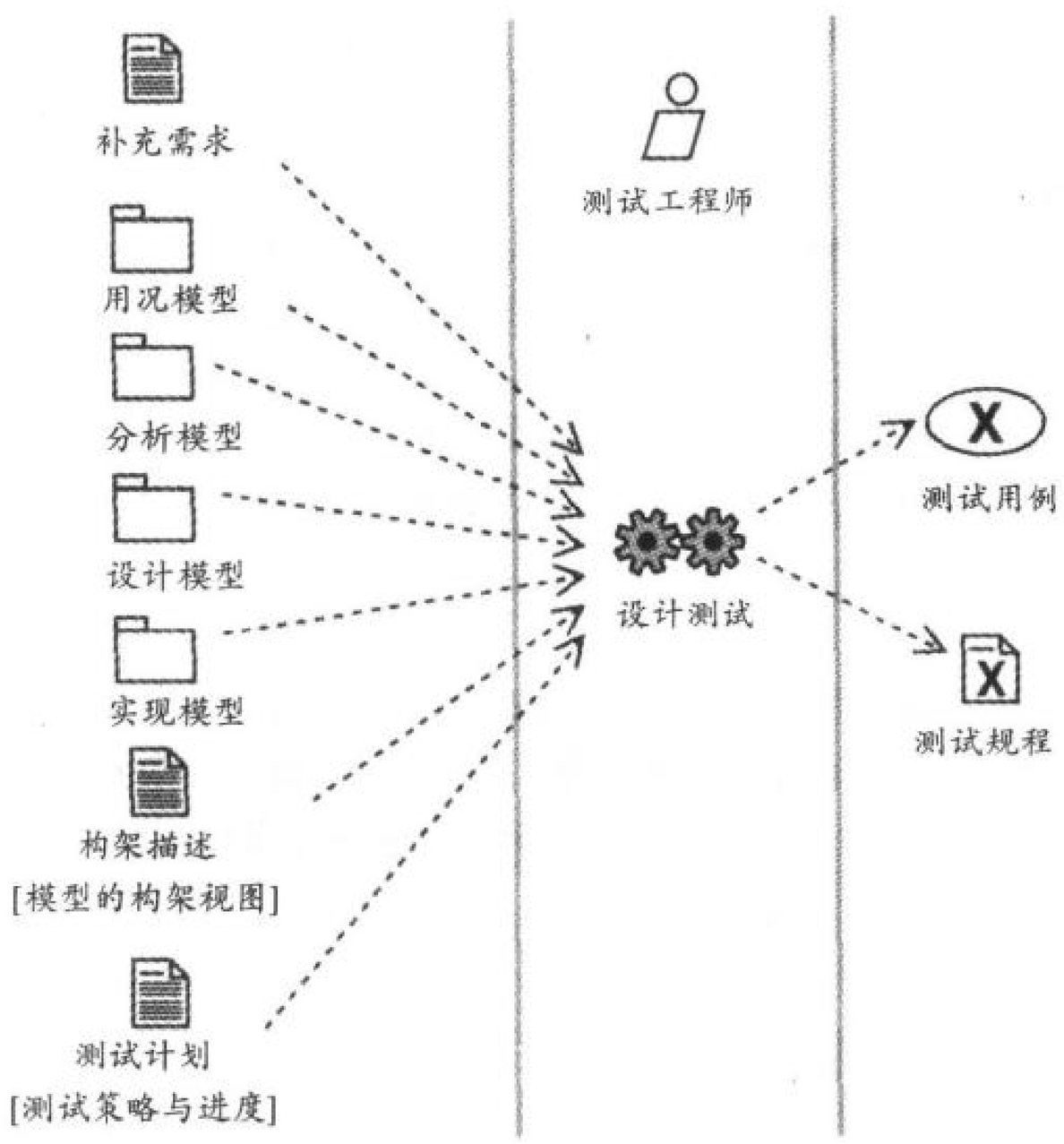


图11-10 设计测试的输入和结果

举例：集成测试用例

测试设计人员从考虑顺序图开始，该图是“支付账单”用况的**用况实现—设计**的一部分。图11-11是该顺序图的第一部分（参见9.5.2.2节）。

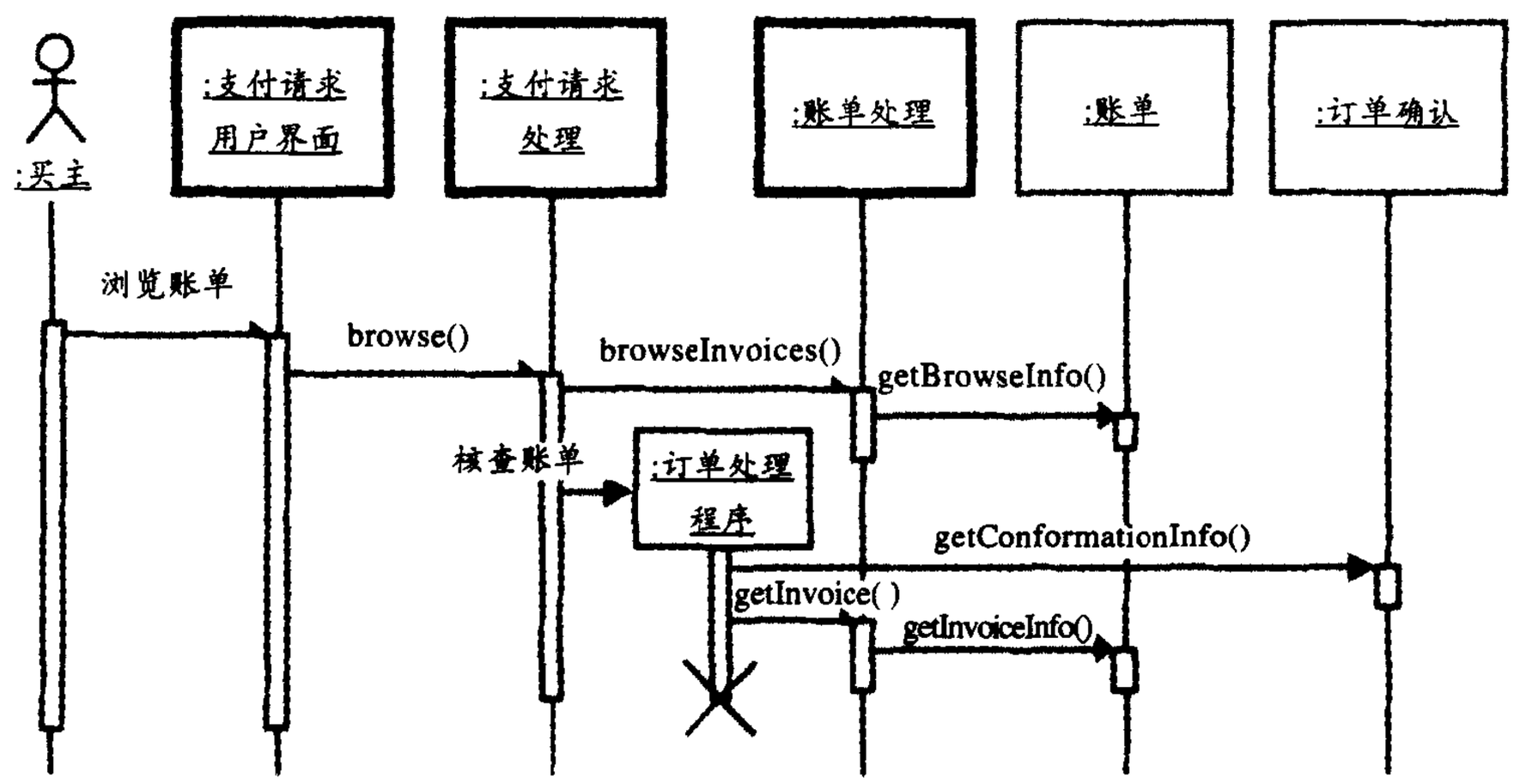


图11-11 “支付账单”用况的**用况实现—设计**的顺序图的第一部分

请注意，可能会有几个不同的“序列”贯穿于一个顺序图中，这些序列依赖于该系统的初始状态和来自参与者的输入等因素。例如，在图11-11所示的顺序图中，我们注意到如果系统中没有“账单”，许多信息就不会发送出去。

从图11-11所示的顺序图导出的测试用例，应描述如何通过捕获所要求的系统初始状态以及来自参与者的输入等使顺序得以发生的因素，来测试贯穿于图中的一个感兴趣的顺序。

稍后，当执行相应的集成测试时，可以捕捉到系统内对象间的实际交互（比如，通过创建跟踪打印输出或通过单步执行），然后将其同交互图进行比较，两者应相同，否则，就是发现了缺陷。

11.5.2.2 设计系统测试用例

系统测试用于测试系统功能整体上是否正确。每个系统测试主要测试在不同条件下的用况组合。这些条件包括不同的硬件配置（处理器、基本内存、硬盘等）、不同程度的系统负载、不同数量的参与者以及不同规模的数据库。测试设计人员在设计系统测试用例时，应对以下用况组合的优先级进行排序：

- 那些对并行功能需要的用况组合。
- 那些可能被并行执行的用况组合。
- 如果并行运行，有可能相互影响的用况组合。
- 包含多进程的用况组合。
- 经常性的、并可能以复杂和不可预知的方式消耗系统资源（如进程、处理器、数据库以及通信软件）的用况组合。

因此，许多系统测试用例可通过考虑用况、尤其是考虑它们的事件流和特殊需求（比如性能需求）来发现。

11.5.2.3 设计回归测试用例

一些从早期的构造中得到的测试用例可用于后续构造的回归测试中，但并非所有的测试用例都适合于回归测试。要保持适用性并继续为系统质量保证作贡献，测试用例必须很灵活，能够适应被测试软件的变化。这种从回归测试用例中产生的灵活性要求做一些额外的开发工作，这意味着我们要仔细权衡，在值得这样去做时才将其转化成回归测试用例。

11.5.2.4 识别并构造测试规程

测试设计人员可按测试用例一步步地开展工作，同时提出每个测试用例的测试规程。设法尽可能多地重用已存在的测试规程，这意味着也许需要对它们加以修改，使其可用来详细说明如何执行一个新的或发生了变化的测试用例。测试设计人员还应尽可能地创建可被几个测试用例重复使用的测试规程。这使得测试设计人员可对许多测试用例快速且正确地使用简明的测试规程集。

绝大部分测试用例将测试来自多个服务子系统的多个类（这是因为测试用例基于用况或用况实现），但是如果可能，对每个测试规程应详细说明如何测试仅来自于一个服务子系统的类；然而，多个测试规程可以详细说明如何测试一个服务子系统。每个测试用例可包含多个测试规程，也许是一个测试规程对应测试用例中待测的一个服务子系统。保持测试规程和服

务子系统间的一对一的关系使得测试规程更容易维护。当一个服务子系统发生变化时，对相关测试规程的影响就只局限于那些用于验证该服务子系统的测试规程，其他的测试规程将不受影响。

举例：测试规程

“账户”服务子系统提供账户间的资金转移功能，该功能包含在多个用况实现中，比如那些为“支付账单”和“账户间转账”而设计的用况实现。一个称为“验证转账”的测试规程将详细说明如何测试账户间的资金转移。这个由“验证转账”所详细说明的测试规程具有三个输入参数：两个账户实体和一笔待转移的款额，并通过查询相关的两个账户在转账前后是否保持了平衡来验证转账的正确性。

测试设计人员为“支付账单”用况建立了8个测试用例，为“账户间转账”用况建立了14个测试用例。“验证转账”测试规程详细说明了那些测试用例中的部分测试用例或所有的测试用例是如何实施的。

11.5.3 活动：实现测试

实现测试的目的是为了尽可能地建立测试构件以使测试规程自动化（并非所有的测试规程都能自动化）。参见图11-12。

用测试规程作为输入来创建测试构件：

- 当使用测试自动化工具时，要执行或详细说明由测试规程所描述的动作，随后记录这些动作，并产生出一个测试构件，如一个Visual Basic的测试脚本。
- 当明确地编程测试构件时，要把测试规程作为编程工作的主要规格说明。要注意，这样的编程工作也许会需要具有高超编程技巧的编程人员。

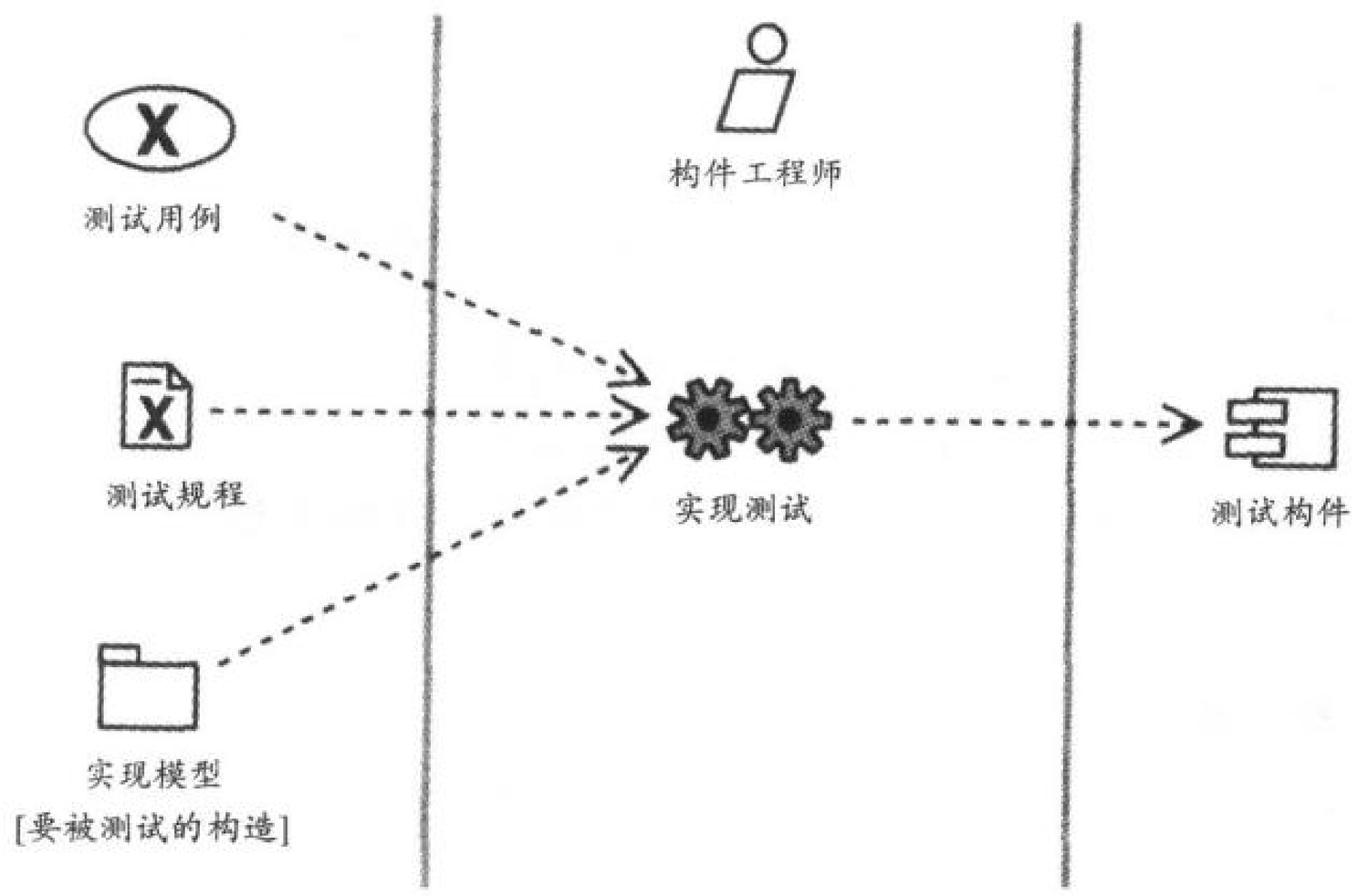


图11-12 测试实现的输入和结果

测试构件常常使用大量待测试的输入数据，并且产生大量的输出数据作为测试结果。清晰、直观地显示这些数据是非常有用的，因为这样可以正确地对其加以说明，而且测试结果也可以得到解释。可使用电子表格和数据库应用来达到这个目的。

11.5.4 活动：执行集成测试

在该活动中，执行在迭代内创建的每个构造所需要的集成测试（参见11.5.2.1节）（参见10.5.2节），并捕获其测试结果。参见图11-13。

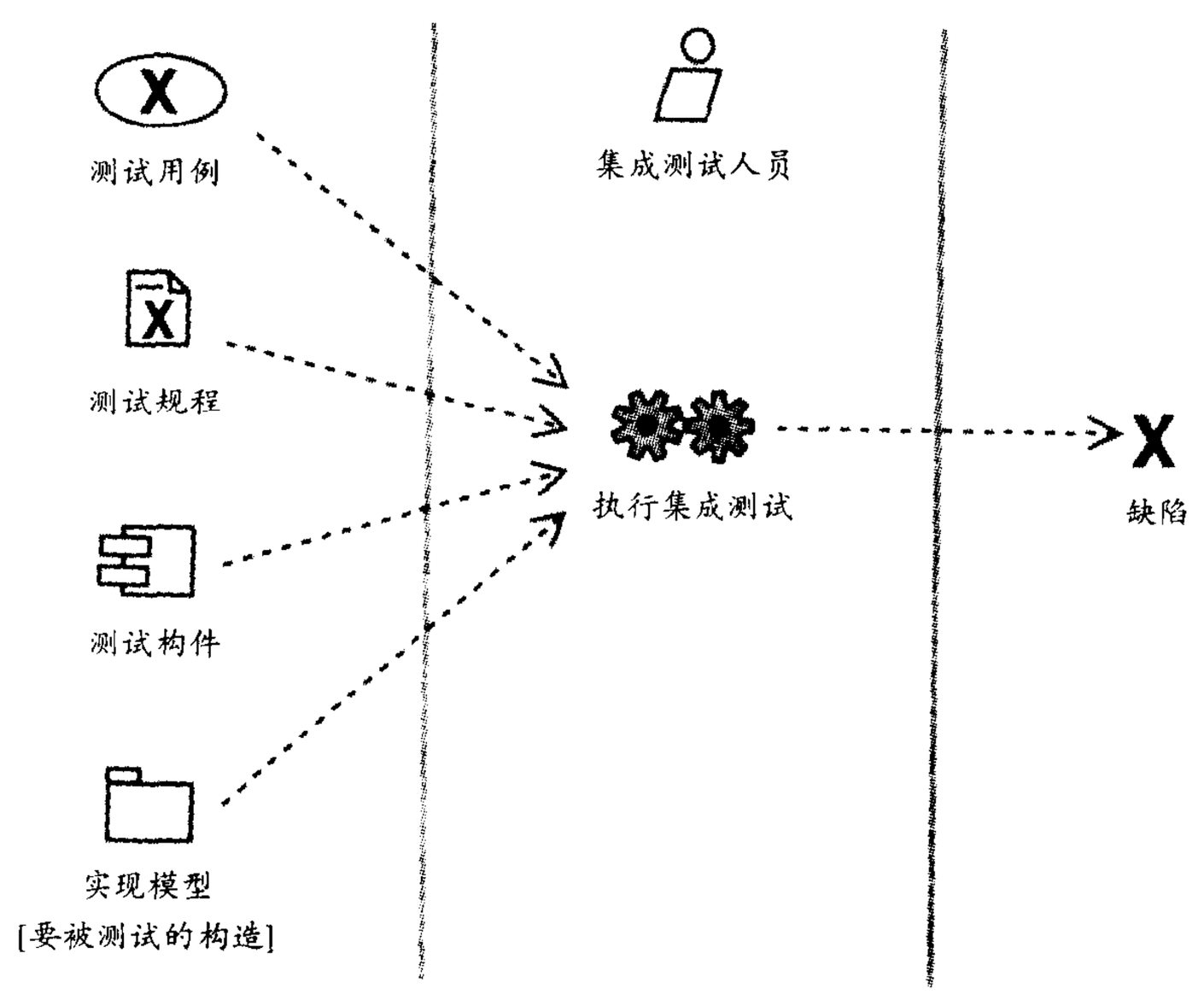


图11-13 集成测试的输入和结果

集成测试按以下顺序进行：

1) 对每一个测试用例通过手工方式执行测试规程或者通过运行任意的测试构件自动地执行测试规程，实现与构造相关的集成测试。

2) 将测试结果和预期结果相比较，并研究偏离预期结果的测试结果。

3) 把缺陷报告给相关的构件工程师，他负责可能含有该缺陷的构件。

4) 把缺陷报告给测试设计人员，他们将利用这些缺陷对整个测试工作的结果进行评估（如11.5.6节所述）。

11.5.5 活动：执行系统测试

系统测试的目的是为了执行在每一次迭代中需要的系统测试（参见11.5.1.2节），并且捕获其测试结果（参见图11-14）。

当集成测试表明系统满足当前迭代中所确定的集成质量目标集时（例如，95%的集成测试用

例的执行结果是所期望的），就可以开始进行系统测试。

系统测试的执行与集成测试类似（参见11.5.4节）。

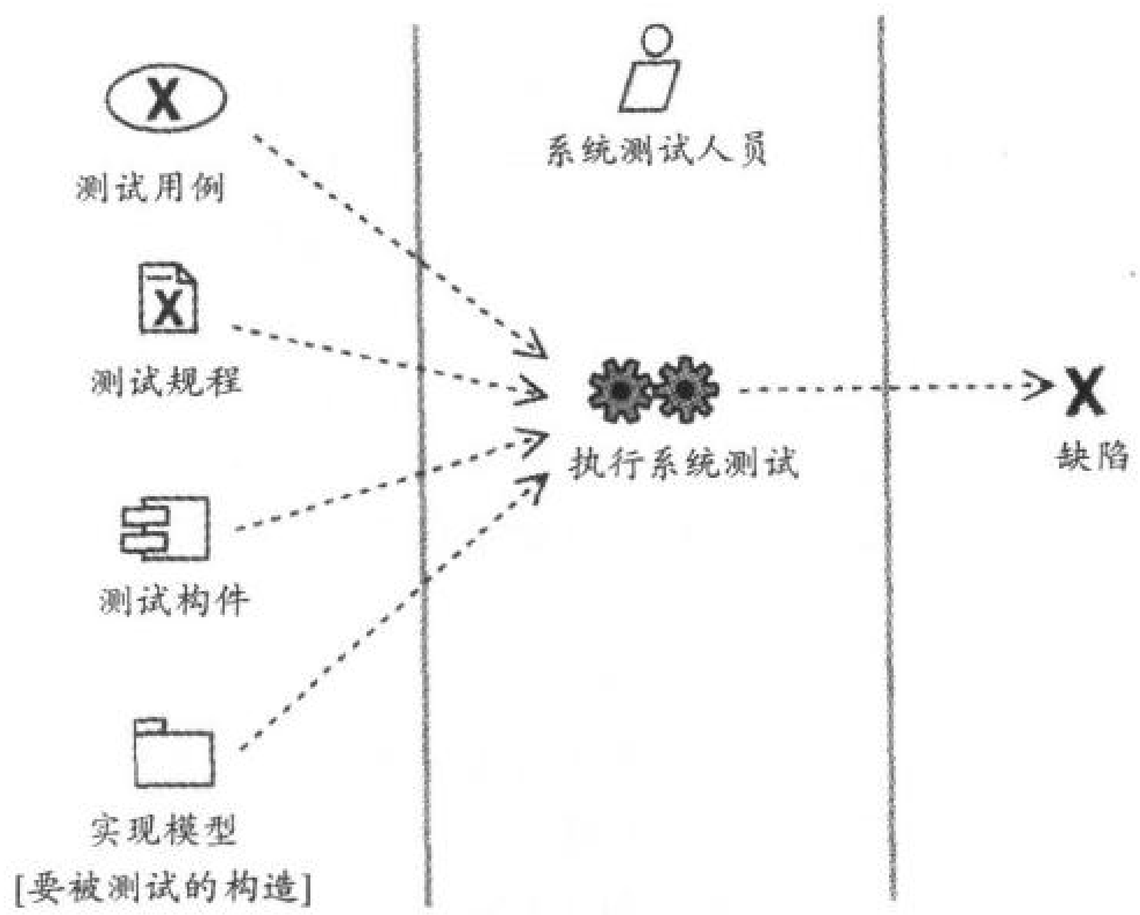


图11-14 系统测试的输入和结果

11.5.6 活动：评估测试

测试评估的目的是为了对一次迭代内的测试工作作出评估（参见图11-15）。

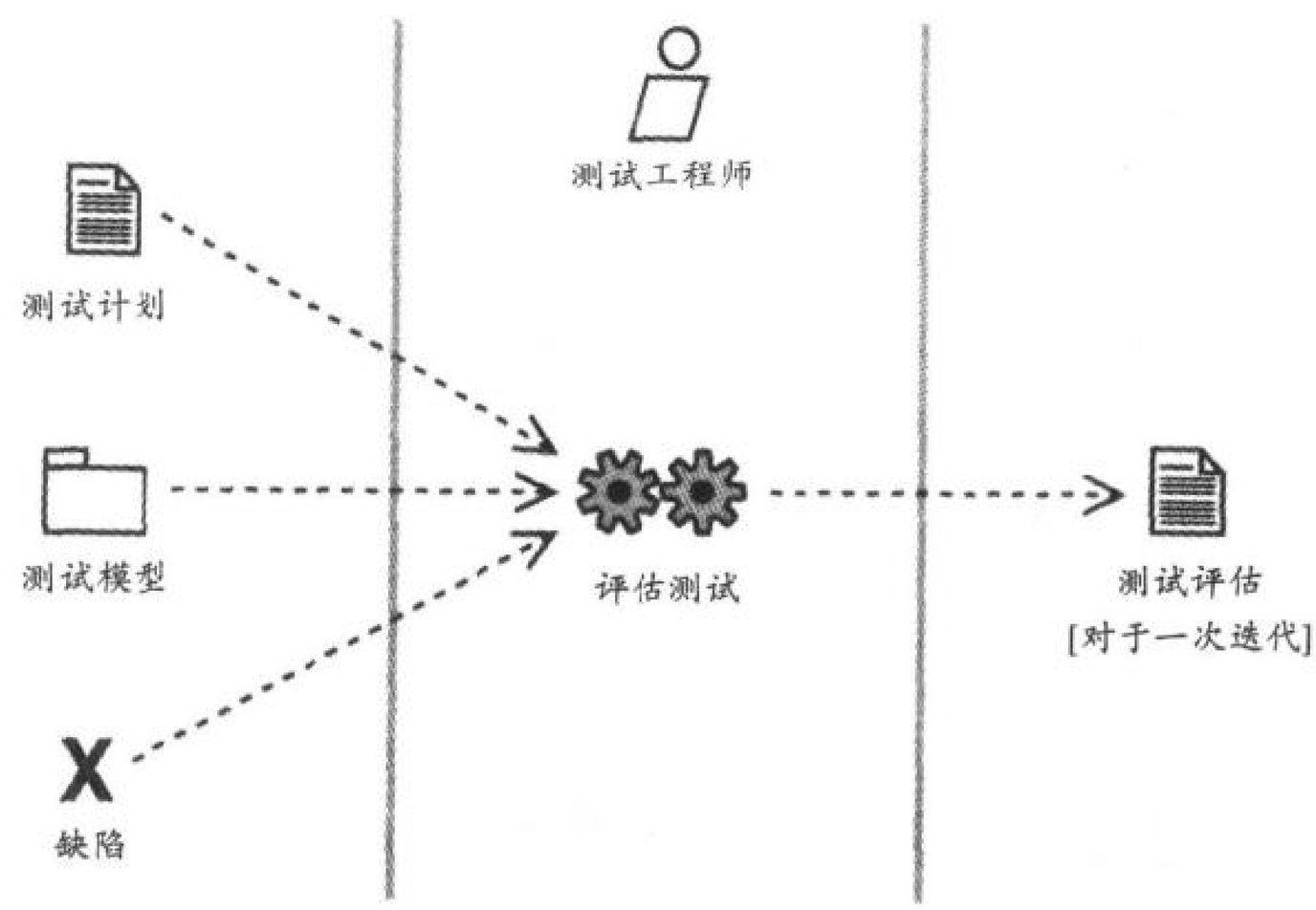


图11-15 测试评估的输入和结果

测试设计人员将测试工作的结果和测试计划勾画的目标进行对比，并对其作出评估。他们准备了一些度量标准，用来确定软件的质量水平，并确定还需要进一步做多少测试工作。

测试工程师尤其看重两条度量标准：

- 测试完全性（*testing completemess*）：由测试用例的覆盖率和测试构件的覆盖率导出，用来显示测试用例和代码已执行的百分比。
- 可靠性（*reliability*）：基于对已发现缺陷的趋势分析和按照预期结果进行测试的趋势。

为确定系统的可靠性，测试设计人员创建了缺陷趋势图，以阐明特定类别的缺陷（如新的或者严重的缺陷）在时间跨度上的分布。测试设计人员还创建了能够描述在时间跨度上测试成功率（即已达到预期结果的测试的比率）的趋势图。

缺陷趋势通常都遵循一个模式，即在各个项目中都会重复。例如，当一个测试刚开始时，新的缺陷一般增加很快，过一段时间就会相对平稳，最后开始减少。到那时，再把实际趋势和以前项目中相似的趋势进行比较，就可能预测出还需要多少工作量才能达到令人满意的质量水平。

基于缺陷趋势分析，测试设计人员可能提出进一步的建议，例如：

- 如果可靠性测量指出系统还不够成熟，就要执行额外的测试来查找更多的缺陷。
- 如果对当前迭代所设置的质量目标过高，就要降低一下测试准则。
- 把系统中看上去质量过关的部分分离出去，并把它们作为当前迭代的结果交付出去。没有达到质量标准的部分必须加以修改，然后再进行测试。

测试设计人员在测试评估描述中阐述测试的完整性、可靠性并提出行动建议。

11.6 测试小结

测试的主要结果是测试模型，它描述了系统是怎样被测试的。测试模型包括：

- 测试用例，规定了在系统中要测试什么。
- 测试规程，规定了怎样执行测试用例。
- 测试构件，使测试规程自动化。

测试还产生测试计划、对已执行的测试的评估以及可以向其他核心工作流（如设计和实现工作流）反馈的缺陷。

11.7 参考资料

[1] IEEE, Std 610.12-1990.

[2] Bill Hetzel, *The Complete Guide to Software Testing,* Second Edition, Wellesley, MA: QED Information Sciences, Inc., 1988.

[3] Robert V. Binder, "Developing a test budget," *Object Magazine*, 7(4), June 1997.

第三部分

迭代和增量的开发过程

软件系统在其生命周期中要经历几个开发周期。每一个开发周期会给客户和用户提供产品的一个新版本，其中产品的第一个版本是最难开发的。产品的第一个版本奠定了系统的基础以及系统的构架，是对有可能存在严重风险的新领域的探索。一个开发周期随着它在整个软件生命周期中所处位置的不同有着不同的内容。在后期版本中，如果系统构架有了较大变化，即意味着开发过程早期阶段需要做更多工作。但如果系统最初的构架是可扩展的话，那么在后期版本中新的项目只是建立在已存在产品基础之上；也就是说，产品的后期版本将建立在其早期版本基础之上。

越来越多的人们相信，在每个开发周期的早期解决问题往往比把问题留到晚期去解决更为有利。于是他们把“迭代”（*iteration*）这个术语应用到初始阶段和细化阶段中的问题解决序列以及构造阶段中的每个构造系列。

风险的到来并没有任何明显的提示，必须要识别风险，限定风险范围，监控风险状况，并尽可能地缓解风险。最好首先处理最重大的风险。同时，还必须仔细考虑迭代的顺序以首先解决最重要的问题。总之，要先做最难做的事情。

在第二部分中我们曾分别描述了每一个工作流。比如，图6-2 集中描述了需求工作流在各个不同阶段的进展情况。类似地，图8-2是关于分析的，图9-2是关于设计的，图10-2是关于实现的，图11-2是关于测试的。

本部分将介绍根据处于软件生命周期位置的不同，这些工作流是如何以不同的方式组合起来的。首先，我们将在第12章描述所有阶段的共同之处，即跨越各个阶段的东西，如制定迭代计划、制定评估标准、建立风险清单、对用况的优先级排序以及对迭代进行评估等。随后的几章将集中介绍每个阶段。

在初始阶段（第13章），活动主要集中在第一个工作流（即需求工作流）中，有少部分工作延续到第二和第三工作流（即分析和设计工作流）。该阶段的工作几乎不涉及最后两个工作流（即实现和测试工作流）。

在细化阶段（第14章），虽然活动仍然着重于完成需求，但第二和第三工作流（即分析和设计工作流）中的活动更趋活跃，为构架的创建打下基础。为了建成可执行的构架基线，有必要包含最后两个工作流（即实现和测试工作流）中的一些活动。

在构造阶段（第15章），需求工作趋于停止，分析工作也减少了，大部分工作属于最后三种工作流（即设计、实现和测试工作流）。

在移交阶段（第16章），工作流的混合程度依赖于接收测试或β测试的反馈。例如，如果β测试没有覆盖实现中的缺陷，则重复进行的实现和测试工作流中的活动会相当多。

在最后一章，即第17章，将回到本书的主题上来。在该章中，我们将展现工作流、阶段、迭代等是如何共同构成一个设计良好的过程以便开发执行关键任务的软件。这一章还用一些篇幅介绍如何处理这些关系以及一个组织如何从现有过程转变成统一过程。

第12章
一般的迭代工作流

这一章，我们回到曾在第5章讨论过的一般的迭代的概念。本章的内容就是从四个阶段中发生的不同迭代里提炼出一般的模式以表征所有迭代的特征。

我们使用这种一般的模式作为基础来创建具体的迭代，而在各个阶段，迭代内容随着该阶段要达到的目的的不同而有所变化（参见图12-1）。

一般的迭代工作流包括五个核心工作流：需求、分析、设计、实现和测试。它也包括在这些工作流之前的计划和在这些工作流之后的评估（第6~11章分别介绍了每一个核心工作流）。本章我们将集中讨论计划、评估以及在所有工作流中都普遍存在的其他活动。

计划在整个开发周期中是必要的。在计划开始之前，首先需要知道要做什么。五个核心工作流提供了一个起点。风险管理通过实现相应的用况集合来识别并缓解风险，是计划的另一个重要的方面。当然，没有对所需资源的估计，就不可能完成计划。最后，还必须对每一次迭代和阶段的执行进行评估。

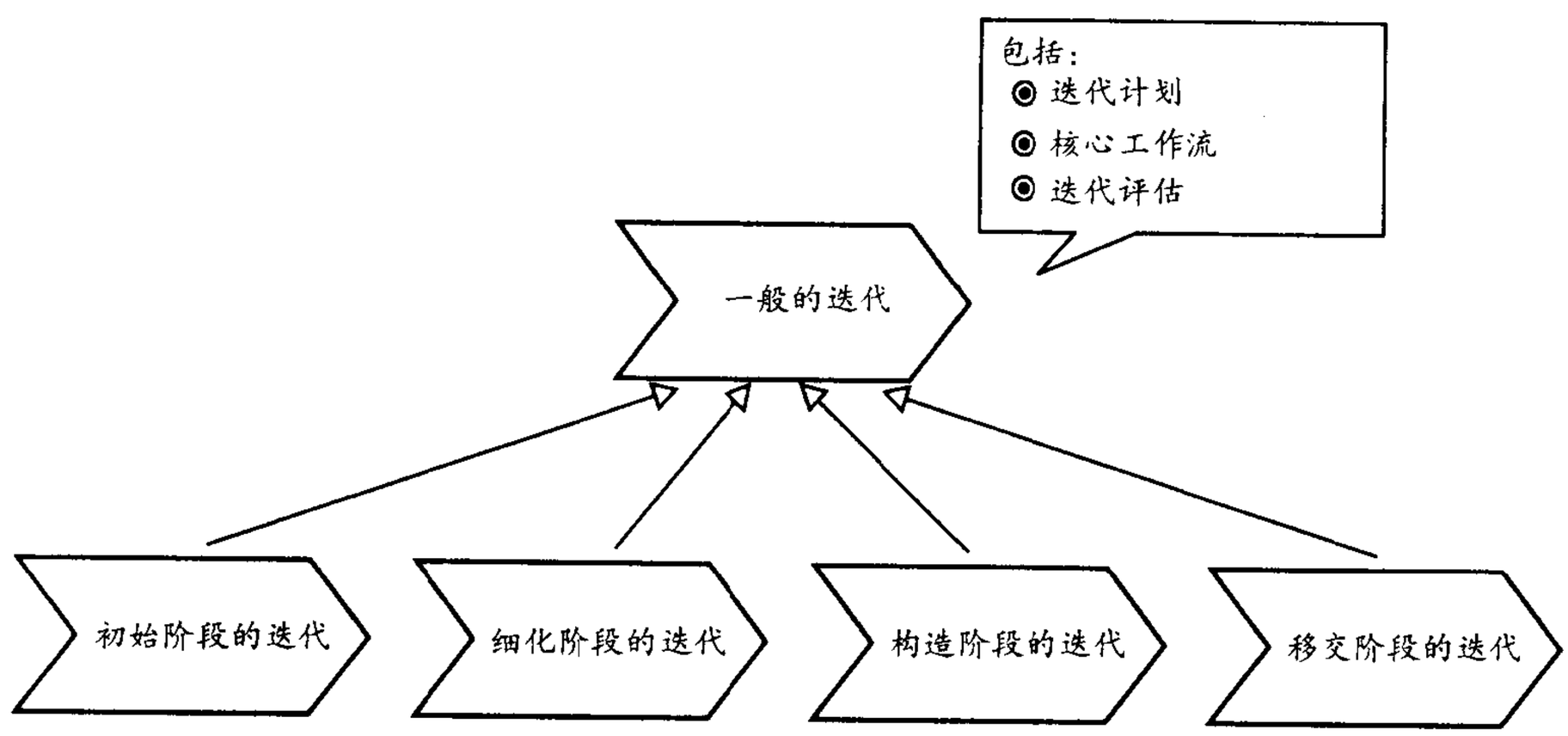


图12-1 一般的迭代工作流用于描述各阶段中具体的迭代工作流。(本图的意图是用来阐述在第三部分确立的方法，在第12章中将阐述一般的迭代，在第13~16章中将详细描述每一个阶段的迭代)

12.1 对平衡的需要

在软件开发项目整个生命周期中的每一时刻，有许多不同的活动序列发生。比如，实现新

的功能，进行构架设计，接收用户反馈的意见，缓解风险，计划将来的工作，等等。面对这种复杂的情况，必须时刻注意保持这些不同的活动流序列的平衡与同步。

开发人员把整体上看来十分复杂的工作分解成一些小的部分，对这些部分的理解往往比对整体的理解要容易得多。如同我们曾在第一部分所勾画的那样，在软件开发生命周期中，开发人员把工作分解成不同的阶段，对每个阶段又再细分成多次迭代。

在每一次迭代中，该项目要力争在运行于迭代中的动作序列间达到平衡。这也就意味着我们应在每一次迭代中做该做的事情。什么是该做的事情取决于我们在生命周期中所处的位置。项目的任务就是为每一个活动序列选择该做的事情去做。在决定活动序列的平衡时，确保其相对的重要性也是重要的，这样就能有效地进行优先级排序并保持同步。没有达到这种平衡和有效的执行，可能导致许多迭代增量式开发过程失败。

在早期的迭代中，主要处理关键风险、关键用况、构架问题、开发环境的选择以及所有面向调研的活动，而后期的迭代则处理实现、测试、性能评估问题以及系统实施等面向开发的活动。将所有这些活动相互关联起来是一项微妙的平衡工作，正是这种微妙性使得软件的开发特别困难。

了解这些不同的活动序列并保持其平衡正是我们在每一次迭代中所要做的工作。在统一过程中，一部分活动序列被确定和描述成核心工作流。还有一部分活动序列是没有被正式确定的，但对这些活动序列也可像对工作流一样认真加以处理。比如：

- 与客户就新的需求交换意见。
- 为顾客提供报价。
- 通过建立一个业务模型了解系统的情况。
- 计划和管理项目。
- 建立并管理开发环境，即过程和工具。
- 管理风险。
- 把一个产品实施到客户环境中。
- 响应用户的反馈。

12.2 阶段是开发工作的第一次划分

划分软件开发过程的第一步是将其按时间顺序分成四个阶段：初始阶段、细化阶段、构造阶段、移交阶段。每一个阶段又可以进一步分解成一次或多次迭代。本章将详细介绍这些阶段和迭代的共性，接下来的四章将分别详细介绍每一个阶段。

12.2.1 初始阶段确立系统可行性

初始阶段的主要目标是建立业务用况，即伴随项目的用况。随着可用信息的增加，在细化阶段将进一步开发用况。初始阶段并不对所计划开发的系统进行彻底的研究，而是仅仅挑出一小部分用况来支持最初的业务用况。我们可采取如下四步来构造业务用况：

1) 限定所计划开发的系统的范围，即定义系统的边界并且确定与外部相关系统的接口。

2) 描述或勾画系统的候选构架，尤其是系统中全新的、带有风险的或是较难实现的部分。这一步仅针对系统的构架描述，很少针对可执行的原型。构架描述包含最初的各种模型视图。目的是要确信在下一阶段我们能够建立一个稳定的目标系统构架。并不在初始阶段建造构架，仅仅确信能够建立而已；建造构架是细化阶段的主要工作。

3) 识别出那些影响系统建造能力的重大风险，并且断定是否可以（也许是在后续阶段）找到方法缓解这些风险。在初始阶段，只考虑那些影响可行性的风险，也就是对系统的成功开发有威胁的风险，而把偶尔发现的次要风险列入风险清单，在细化阶段再加以仔细考虑。

4) 通过建造一个**证明概念原型**，向潜在的用户或客户表明，待开发的系统能够解决他们想要解决的问题或者能够支持其业务目标。在初始阶段，可以通过建造一个原型来表明潜在的用户或客户问题的解决方案。原型集中表现了新系统在使用上的基本思想，如用户界面或关心的新算法。这种原型是带有探索性的，也就是说，它只证明一种可能的解决方案，但可能不会发展成最终产品，通常是抛弃型的原型。相对而言，在细化阶段开发的构架原型则可能是进化型的原型，即能在下一阶段进行进一步进化的构架原型。

作出这些努力是想说明，从经济角度看，开发这个产品是值得的。以上步骤表明：在相当大的限度内，系统可能提供与建造系统所需要的投资相称的收益或其他价值。换句话说，对业务用况进行了初步分析，在下一阶段将对其进行进一步的精化。

这么做的目的是将初始阶段中计划的时间、投入的人力和资金减少到最小程度，直到发现系统是切实可行时为止。如果是一个较大的新系统，且处于未知领域，就意味着可能要投入相当多的时间和精力，并且可能要持续进行多次迭代。而对于一个已知领域内的大家都熟悉的系统或者是对已有系统进行升级，则风险和不可预知的东西将会很少，从而使得初始阶段在几天内就可以完成。

12.2.2 细化阶段关注“做的能力”

细化阶段的主要产品是一个稳定的构架，用于在后续过程中指导系统的开发。细化阶段还对目标系统进行研究，以求能高度准确地制定构造阶段的计划。为了实现构架和高度准确的费用估算这两个目标，开发组应做如下一些工作：

1) 创建构架基线，使其覆盖系统中对构架有重要意义的功能和对项目相关人员来讲十分重要的特性，这些内容已经在第4章中描述过。基线由模型的各种制品、构架描述和可执行的实现组成，不仅表明能够建立一个稳定的构架，而且还包含该构架本身。

2) 识别出重大的风险。这些风险有可能对下一个阶段的计划、费用和进度产生负面影响。要从时间和资金两个方面将这些风险缓解。

3) 通过可靠性（缺陷率）和响应时间等质量属性来规定系统可以达到的水平。

4) 捕获功能性需求中大约80%的用况，使其足够可以制订构造阶段的计划。（我们将在本章稍后的部分说明80%意味着什么。）

5) 在由业务实践所设定的界限内，准备一份有关进度、所需要的人员和费用的标书。

需求和构架（包括分析、设计和实现）是初始阶段和细化阶段的主要工作。

12.2.3 构造阶段建造系统

构造阶段的一般目标是由其主要里程碑指出的：具有最基本的可操作能力，即建造出能进行β测试的产品。本阶段用到的开发人员比其他阶段要多，时间也比其他阶段要长。这就是为什么在进入这个阶段前要将所有重要内容都搞清楚的原因。该阶段通常比别的阶段要经历更多次的迭代。

在很多情况下，构造阶段有可能会因为需求、分析和设计工作做得不好而花费过多的时间。开发人员删改系统的结果是必须花费比原计划更多的时间来满足需求和减少错误。使用多阶段和迭代增量式开发的这种软件工程方法的最大的优点之一，就是使你可以在整个生命周期中保持时间和资源的均衡分配（参见12.1节）。

构造阶段的活动包括：

1) 将对用况的识别、描述和实现扩展到全体用况。

2) 完成分析（在构造阶段开始时，可能有一半以上的用况等待加以分析）、设计、实现和测试（可能剩余90%）。

3) 保持构架的完整性，在必要时要对其进行修改。

4) 监控在上两个阶段发现的关键和重大的风险。如果这些风险确实存在，则应设法缓解其影响。

12.2.4 移交阶段转移到用户环境

移交阶段开始时通常伴随着β版本的产生，即开发组织将基本可操作的软件产品交付给真正用户群的代表。在用户组织苛刻的环境中对产品开发状况所进行的验证，通常要比在开发环境内更严格。

移交阶段的活动包括：

- 准备活动，如场所准备。
- 建议用户更新软件将使用的环境（硬件、操作系统、通信协议等）。
- 为产品准备用户手册以及其他文档。在构造阶段中，我们为使用β版本的用户准备预备文档。
- 调整软件，使其适于在用户环境参数下操作。
- 在收到β测试反馈后，纠正所发现的缺陷。
- 修改软件中没有预见到的问题。

移交阶段伴随着产品正式版本的产生而结束。但在项目组完成项目前，开发组长必须针对以下两个目标进行总结：

- 寻找、讨论、评估并记录可提供给将来参考的经验教训。
- 记录在软件的新版本中或新一代软件中会用到的一些事件。

12.3 再论一般的迭代

我们要区分核心工作流和迭代工作流。核心工作流是：需求、分析、设计、实现和测试

（在第6~11章中加以描述）。在统一过程中，这些核心工作流发生多次，而在瀑布模型中，这些核心工作流理论上只发生一次。核心工作流在每一次迭代中重复发生。但在每次重新发生时，它们在具体内容上有所不同，而与该迭代的中心问题有关。

12.3.1 在每次迭代中重复的核心工作流

一般的迭代由五个工作流组成：需求、分析、设计、实现和测试；另外它还包括计划和评估（参见图12-2）。

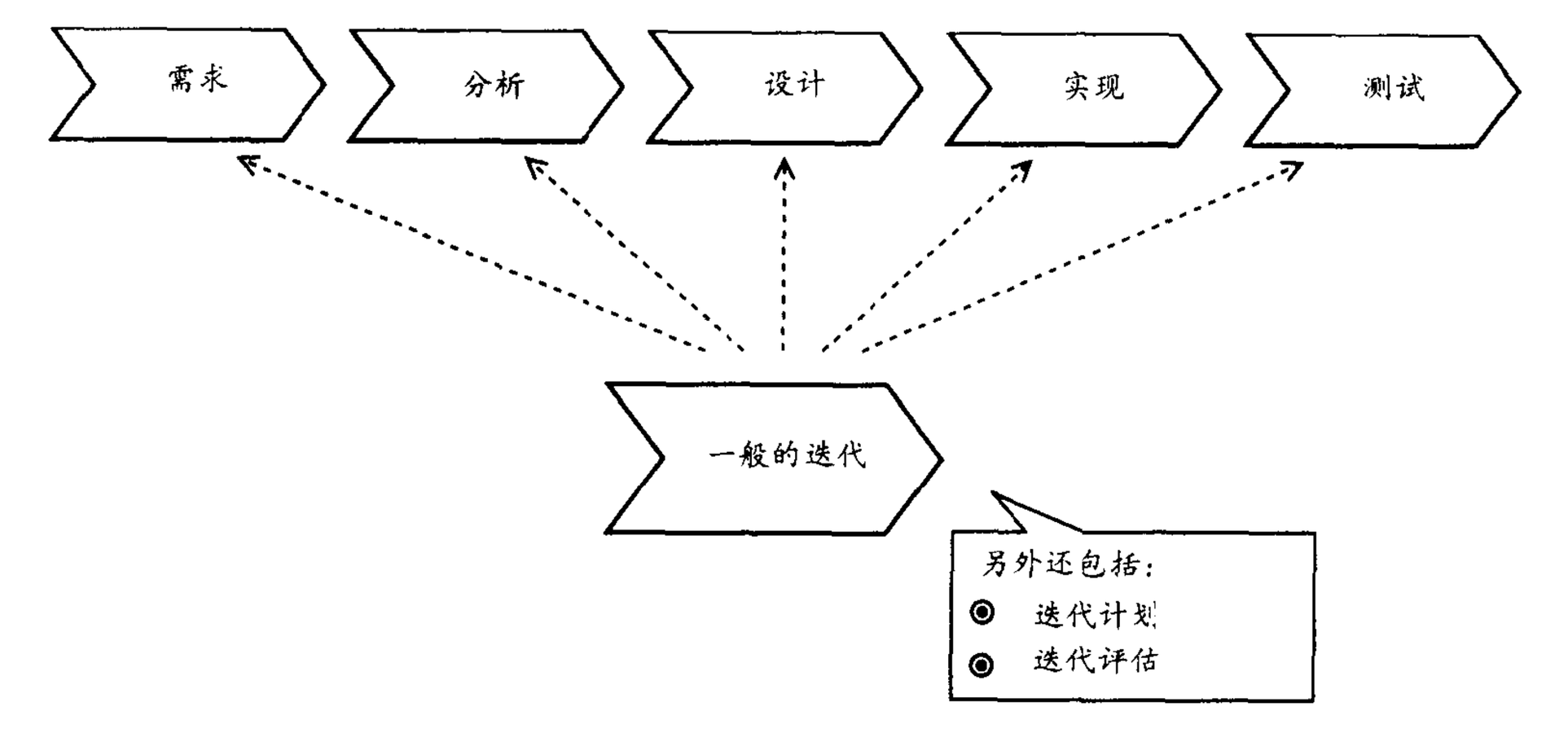


图12-2 五个核心工作流在每次迭代中重复进行。对于每次迭代，先制定计划，然后进入五个核心工作流，最后进行评估

在12.4~12.7节中，我们将讨论在迭代之前的计划制定工作；在12.8节中讨论对每一迭代的评估。然后在第13~16章中详细介绍这五个核心工作流在每一阶段是如何应用的。

12.3.2 参与到工作流中的工作人员

在谈及软件开发时，我们已不止一次用“复杂”来形容。图12-3提供了对开发过程的一个概貌。即使如此，该图也并不简单，并且我们知道在实际工作中将比该图还要复杂得多。本章既不打算详细描述每个工作人员是怎样开发他所负责的制品，也不准备详细说明这些制品是什么。图中的小齿轮代表工作，活动间的箭头代表现实中的关系，详细内容请见第6~11章。

另外，图12-3将给你留下在每一次迭代中发生什么的印象。例如，在左上角的开始处，系统分析人员识别出用况和参与者，并且将它们构造成用况模型。接着，用况描述人员详细说明每一个用况，用户界面设计人员建立用户界面原型。构架设计师考虑风险等因素，将用况在迭代中的开发顺序按优先级排序。

可以看出，在整个开发过程中，需求“周期”内执行的特定活动将随着迭代位置的不同而有所变化。例如，在初始阶段，工作人员将用况的详细描述和优先级排序限定于仅在初始阶段需要用到的一小部分用况上。前四个工作流开展时尽管有一些重叠，但主要还是按时间顺序执

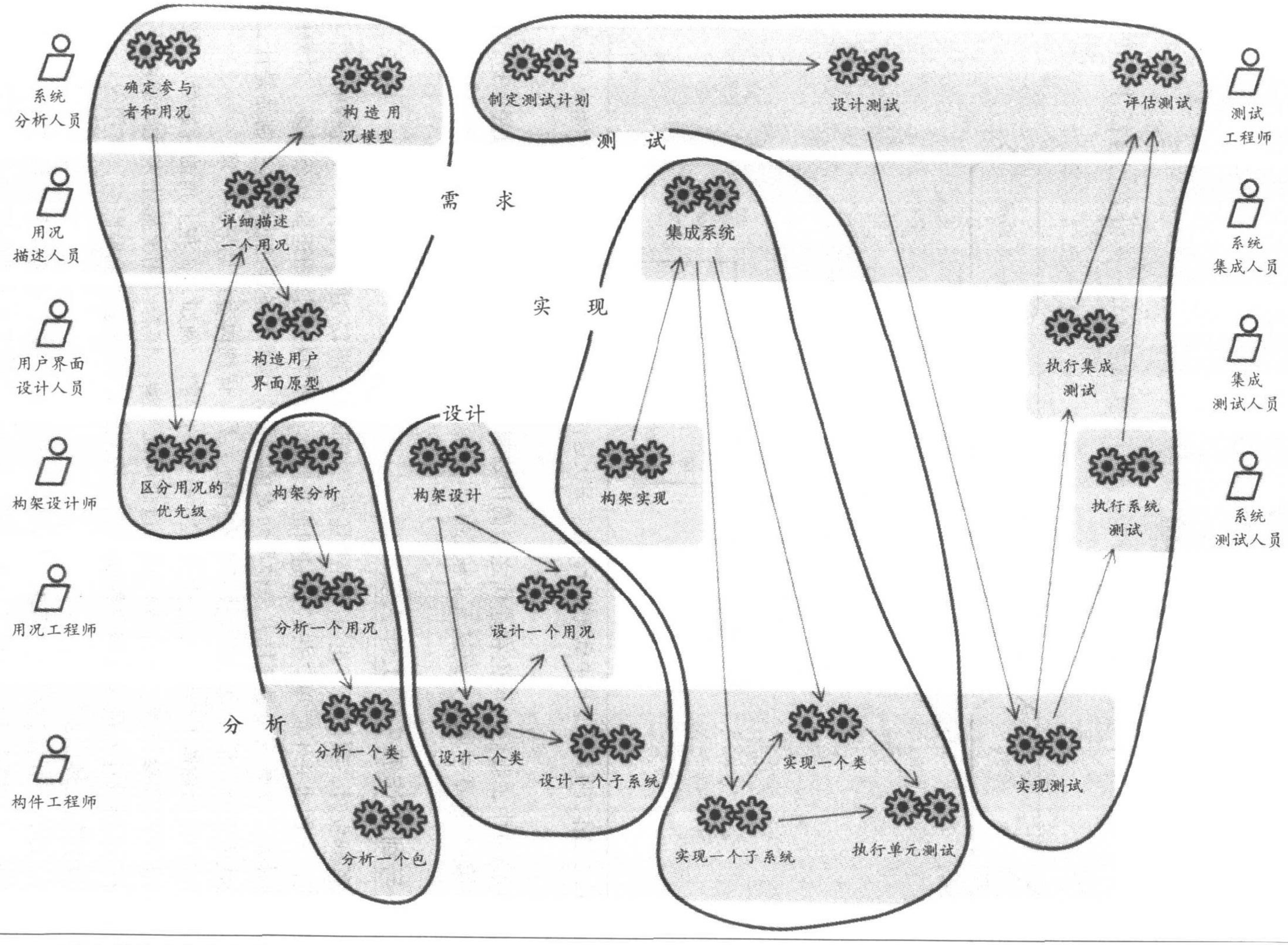


图12-3 工作人员的头衔纵向排列在左右两边，并用“泳道”加以识别；时间从左至右递增；包含工作人员和他们所执行的活动的核心工作流圈在手画的“圆框”内。例如，构件工程师在分析工作流中分析一个类和一个包，在设计工作流内设计一个类和一个子系统，在实现工作流中实现一个类和一个子系统并执行一个单元测试

行的。参与分析工作流的三个工作人员将在设计工作流中继续开展工作。

然而，随着测试工程师对测试内容进行规划，测试工作流将很早就开始进行。只要在实现工作流中一有足够的细节可以利用，测试工程师就开始设计测试。随着对已经经过了单元测试的构件进行集成，系统测试人员和集成测试人员将针对集成的多个层次的结果进行测试。测试工程师将评估他所规定的测试是否已经足够充分。

在后面的几章中，我们将向图12-3中添加每一个阶段需要表示的迭代工作流。这些工作流均仿照这个一般的流程图的模式。由于不同阶段的侧重点有所不同，因此相应的迭代工作流在不同程度上会有所变化。

12.4 计划先于行动

在初始阶段开始时要知道以下三件事：

- 在四个阶段中，将按一系列的迭代来进行项目的开发。
- 我们有一些以前的开发人员搜集的关于目标系统的信息（这些信息将启动项目的进行）。
- 我们自己拥有一些关于这个领域和所做过的类似系统的背景信息。

从这些信息中，我们知道必须为项目和每一次迭代制定计划（项目计划和迭代计划）。刚开始时，我们对要做什么所知有限，项目计划和迭代计划都只含有少量的细节。而随着初始阶段和细化阶段工作的开展，项目计划和迭代计划将变得越来越详细。

首先描述怎样为阶段和迭代制定计划以及怎样对迭代进行评估。在12.5节中，我们将讨论风险对计划的影响；在12.6节中，我们将讨论如何通过选择合适的用况来消除风险。最后，在12.7节中，我们将讨论资源的分配。

12.4.1 制定四个阶段的计划

统一过程规定了每一个阶段所涉及的内容。项目计划的任务就是将这些规定演化为具体的术语。

- 时间分配。要决定每一个阶段应分配多少时间以及准备完成的日期。这些时间尽管设定得十分精确，但在初始阶段开始时并不十分准确，然而它们会随着所知内容的增加而趋于稳定。毕竟，直到细化阶段结束时才制定严格的标书。
- 主要里程碑。当达到预先设定的准则时，标志着一个阶段的结束。最初，这些准则只是一种猜测，但是通过对在该领域内过去积累的经验、目标系统与早期版本的差异程度、新系统要达到的性能要求以及软件开发组织的开发能力等的逐渐了解，对这些准则也会逐渐有所把握。
- 每个阶段的迭代。在每一个阶段中，我们通过一次或多次迭代推进工作。粗糙的项目计划中包含了每次迭代的一般特征。
- 项目计划。项目计划勾画了一个整体的“路线图”，包括进度表、主要里程碑日期和准则，以及由阶段分解来的迭代。

可以预料，初始阶段的第一次迭代是非常困难的。因为除了迭代本身之外，还必须处理以

下几个方面：

- 裁剪统一过程使其适用于这个项目，并且选择使过程自动化的工具。
- 开始汇集具有项目所要求的各种才能的人员。
- 构建各种关系以形成一个高效的开发组。
- 了解该领域的一些知识。通常这个领域对于开发组而言是全新的。
- 对项目特性的了解，非先验性的开发要比对一个已存在的产品的升级困难得多。
- 使开发组熟悉将应用于裁剪后的过程和项目的开发工具。

12.4.2 制定迭代计划

每一个阶段由一次或几次迭代组成。制定迭代计划和制定阶段计划的步骤大致相似，包括如下内容：

- 迭代进度表。决定允许每一次迭代需要的时间及其完成的日期，它在刚开始时是十分粗略的，随着所了解内容的增加其精确性有所增加。
- 迭代内容。项目计划用术语勾画出所计划的迭代，而随着迭代开始日期的临近，我们将对要做些什么制定更详细的计划。迭代的内容基于以下几个方面：
  - 在迭代期间，哪些用况将至少完成一部分。
  - 哪些技术风险需要进行识别、转化为用况并将它们缓解。
  - 已经发生的任何需求变化或者可能已经纠正的缺陷。
  - 要部分地或完整地实现哪一个子系统。（这一点根据所考虑阶段的不同而有所变化。例如，在细化阶段，我们找出大部分的子系统以及所有对构架有重要意义的类。在构造阶段，我们向子系统中加入越来越多的功能，形成更完整的构件。）

当前迭代的计划是十分详细的，而下一次迭代的计划则随着所了解内容的增加而越来越详细。后期迭代的细节有可能受当前所能利用的知识的限制而比较粗略。

- **次要里程碑**（参见附录C）。达到预先制定的准则（最好是在计划的日期），标志每一次迭代的完成。
- **迭代计划**（参见附录C）。每一次迭代过程的活动都在细粒度迭代计划中进行了记录。在每一次迭代的初期，分配一些可用人员作为工作人员。

在每个阶段中，计划要进行的迭代的数目基本上随着目标系统的复杂度的不同而变化。一个非常简单的项目可能在每个阶段中只进行一次迭代，而复杂一点的项目也许需要更多的迭代。例如：

- **初始阶段**（参见附录C）：一次迭代，主要用来规定系统的范围。
- **细化阶段**（参见附录C）：两次迭代，其中第一次迭代用于建造构架的轮廓，第二次迭代用于完成构架基线。
- **构造阶段**（参见附录C）：两次迭代，以确保所产生的增量执行起来有令人满意的效果。
- **移交阶段**（参见附录C）：一次迭代。

随着所考虑项目的规模的增大、复杂度的提高以及涉及到的因素的增多，可以预料开发组的规模会随之扩大。此时，将会有更多的迭代，而每次迭代耗费的时间根据系统的规模从一个星期到三个月不等。

12.4.3 长远考虑

持续时期较长的系统会有各种各样的变化，如新技术、系统环境的新接口以及更先进的平台等。同时，计划人员也应该检查业务是否要适用于其他组织（如附属公司或合作伙伴等）。但对将来的推测不能过于荒谬。如果你已经看到在将来系统必须作某些更改，你就不会希望建立一个刚性的系统构架，然而另一方面，你也不希望在系统中引入一些不必要的柔性（例如根本就不会用到的柔性）。

12.4.4 制定评估准则

相对于传统的项目而言，迭代的时间比较短。为了避免运行的不确定性，项目经理除了要制定开发进度外，还要确定每一次迭代的主要目标。为了达到这些目标，项目经理建立指明迭代结束的准则。这些准则（例如，在给定点生成最起码的特性）给出迭代的侧重点，这有利于迭代过程的及时结束。准则的例子包括：

- 以用况的形式表示的功能性需求。
- 对用况附加这些用况所应用的非功能性需求。
- 将不属于特定用况的非功能性需求列在补充性需求中（在6.7节已作过详细描述）。

比如，早期迭代的目标之一可能是要解决用户需求表达上的歧义性问题。准则用可测量的条款（如性能）和可观察到结果的条款（如希望获得的特征）来说明计划人员希望迭代实现的内容。

项目经理提前为每一次迭代和阶段本身制定评估准则。每一个准则都需要有一个清晰的结束点，以便开发人员自己也明白他们是否已完成了工作。另外，这些结束点提供了里程碑，供项目经理估计工作进度。

总的来说，评估准则分为两类：能验证需求的准则和更一般的准则。

就像在第11章中所介绍的，测试工程师从初始阶段开始就已加入了开发。他们识别用况的哪些特性可以通过测试来确认。他们计划用来定义集成测试、回归测试以及系统测试的测试用例。在迭代开发中，测试周期也是迭代的。迭代中创建的每一个构造都是测试的目标。测试工程师增加并精化为每一构造所执行的测试，因此聚集成大量测试以便在后期用于回归测试。早期的迭代比后期的迭代引入了更多的新功能和新测试。随着构造的继续集成，新测试的数目有所减少，而回归测试的数目不断增加以确认集成起来的系统的实现。因而，早期的构造和迭代需要更多的测试计划和设计，而后期的迭代则通过测试的执行和评估来衡量。

一般的准则不能归约到可由测试人员所能测试的代码路径。然而，这些代码首先可以在原型中观察到，随后可以在一系列的构造工作和迭代中观察到。用户、项目相关人员、开发人员可以通过对屏幕和图形化用户界面的浏览，了解到模型制品中静态信息所不能表达的信息。

评估准则描述如何验证迭代中的需求是否已被正确地开发。它用可观察到的或可验证的条款说明项目经理想要迭代达到的目标。他们的原始灵感就是有关其构想的陈述。评估准则进一步将这种构想细化为各种各样的用况、用况场景、性能需求以及测试用例，具体地表达后续增

量应该是什么样子。

12.5 影响项目计划的风险

制定新系统的开发计划很大程度上受所感知的风险的影响。因此，初始阶段最初的工作之一就是要建立风险清单。刚开始时，由于缺乏信息，因此可能会遇到一些困难，但可能对那些决定是否能够建造系统的关键风险还是有一些感性认识。随着早期开发工作的进行，将逐渐搞清楚重大风险是什么。为了预计进度、费用并达到质量目标，必须缓解这些风险。

12.5.1 管理风险清单

模糊地认识到软件开发包含着风险是一回事；把风险展现出来，让大家都能看到且在其指导下对其采取相应的措施则又是另一回事，这就是风险清单的目的。这并不像把一些文件从抽屉中或从计算机的文件夹中清除掉那么简单。为了处理风险，需要了解其全部内容，包括风险清单中风险惟一的标志符。这些内容包括：

- 描述：从简短的描述开始，随着所了解内容的增多，逐渐增加更多的内容。
- 优先级：为风险分配优先级，开始时分为关键风险、重大风险和一般风险。随着风险清单的扩大，也可以增加一些新的类型。
- 影响：风险会影响系统或项目的哪些部分？
- 监控者：由谁负责跟踪持续发生的风险？
- 职责：由什么个人或小组负责让该风险消退？
- 应急措施：如果风险确实发生，应采取什么措施？

对一定规模的项目，最后可能会发现几百个风险。在大型项目中，风险应该以条目的形式存放在数据库中以便高效地排序和查找。开发组不能马上将所有的事件都视为重点。这是进行迭代开发的一个原因之一。风险按严重程度或者它们对系统的影响和侵袭的程度进行排序。我们已经多次强调过，首先要处理的是可能使项目开发失败的风险。有一些风险不是简单地就能解决的，它们会在风险清单上保留一段时间。一些开发组发现：将风险清单中前十个风险作为处理的重点是十分有帮助的。

风险清单并不仅仅是一个静态工具。随着风险的不断发现，风险清单的内容不断增加。一旦风险被消除或风险已经通过了其可能发生的某一开发进程点，则将在风险清单中把这些风险记录剔除。项目经理要召开定期会议（通常同迭代评估一起进行），检查最重大的风险的状态；其他领导则针对次要的风险召开局部性的会议。

12.5.2 影响迭代计划的风险

在初始阶段要识别关键风险，开发组要尽力缓解这些风险。开发人员研究这些风险的性质以便准备迭代计划。例如，为了获得制定计划的足够信息，开发人员将开发一小部分与风险有关的用况，并在**证明概念原型**中实现这些用况。接着，对原型输入一些数据，开发人员发现原

型产生了错误的输出（如过早的激发），这就是关键风险。（这些输入要在规定的输入范围之内，可能是在范围的边缘，而获得的仍然是错误的输出。）

除了那些最严重的风险对项目的成功有影响之外，所有其他风险对进度、费用和质量也都存在着一定的影响。其中有些风险比较严重，必须考虑在原计划的基础上延长时间或增加投入的人力，或者在风险带来的后果产生之前消除这些风险。几乎在所有的情况下，对进度有影响的风险同时也影响到工作量和费用。在某些情况下，虽然有的风险对进度或费用几乎没有什么影响，而对其他因素如质量、性能等有一定影响。

12.5.3 风险处理进度安排

一般的原则是要有计划地针对风险采取措施。阶段以及阶段中的迭代都提供了风险处理进度安排的机制。例如，在初始阶段的迭代中，制定计划来处理那些影响系统构造能力的风险。也就是说，如果可能就消除这些风险，或者至少有一个应急计划。

根据我们的经验，采取其他办法来代替风险调度，其效果并不理想。如果在早期没有有意识地针对风险作出一定的处理，这些风险就会在后来的集成测试和系统测试中经常出现。到那时，解决任何严重问题，可能都需要对系统作大量修改，因而会延误几个星期甚至更长的交付时间。在迭代开发方法中，从第一阶段开始的原型、构造和制品的建造过程中，如果逐步暴露风险，则仍有时间缓解这些风险。

要意识到某些类型的风险是很难识别和描述的。鉴于多方面的原因，一些风险可能模糊不清。这通常是因为人们没有仔细去寻找，或因为有关人员被那种肯定能在预定的时间和费用内完成开发的想法所蒙蔽。如果有些风险只是一闪而过，则项目不会把对这些风险的处理安排到迭代过程中，并以某种顺序来消除。

不管是什么原因，在一些项目中，尤其是开发组对风险管理还没有什么经验时，有一些风险可能被忽视，直到开发的后期才发现。开发组有一定的实践经验之后，就能够将风险按项目进展的逻辑顺序进行排序。例如，在构造阶段，如果存在对第二次迭代的进度有影响的风险，则最后应在该阶段的第一次迭代内消除这个风险。这样做的目的是为了使得构造阶段的每一次迭代最终都能够按计划平安无事地进行，那种在项目进展过程中不能尽快克服风险的情况是不可能发生的。

12.6 用况优先级排序

在本节中，我们将讨论单次迭代内用作驱动器的用况的选择。我们知道，统一过程的每一次迭代都是由一些用况驱动的。实际上，更准确地说，一次迭代是由用况的场景集合所驱动的，这是因为在早期的迭代中，没有必要使用整个用况，仅仅采用一些与手头上的任务相关的场景或路径。有时，当谈到用况选择时，我们指的是选择了与迭代相关的场景。

导致这一选择的工作就称为“*对用况进行优先级排序*”（参见7.4.2节）。按照用况或用况的场景给在迭代中处理的顺序赋予不同的优先级，这些用况应被分别排列在几次迭代中。在早期的迭代中，对部分用况（或用况的场景）进行排序，仍有许多用况因尚未被识别而没有被排序。

所有识别出来的用况都进行排序，这些排序生成了一张用况排序清单。

控制这一排序是一种风险。我们按照用况实现的风险的顺序来排列这些用况。在这里所说的风险指的是广义上的风险。例如，在后期阶段中改变系统的构架是我们要设法避免的风险；不能建立正确的系统是我们在早期希望通过捕获真正的需求而加以消除的风险。这个选择的过程是风险驱动的。12.5.1节中曾介绍过，将风险的详细说明置于风险清单中，而在消除风险时，将每个风险转换成一个用况，这个用况将根据风险的级别插入到风险清单中的适当位置。

在早期迭代中，我们致力于对与系统范围和构架相关的风险进行用况优先级排序。在后期的迭代中，我们选择新的用况填充到构架中，这时的构架已经拥有了许多的功能。我们往结构上填入更多的“肌肉”，后续的用况按一定的逻辑顺序加入，而这种逻辑顺序对应于用况排序清单中的排列顺序。例如，那些功能级别相对较低而需要其他用况的用况就要后于其他用况而被开发。参见5.3节有关将风险驱动作为迭代方法的工作方式的讨论。在后续的三个小节里，我们将处理三种风险：特定风险、构架风险和需求风险。

12.6.1 针对特定产品的风险

这类风险就是在5.3.1节中曾讨论过的技术风险。我们把这些风险转换成用况，当正确实现这些用况时，就能消除这些风险。当实现时，每一个风险都映射到一个能够消除该风险的用况。对这些风险的处理并没有正式纳入统一过程，因此不得不逐个识别这些风险。这里所说的“正式纳入统一过程” 是指统一过程提供一个处理某一类型风险的特定位置。例如，某些构架风险在初始阶段加以考虑，其他风险在细化阶段加以考虑，这在12.6.2节曾作过讨论。没有正式纳入统一过程的风险需要逐个处理，在它们的出现将影响开发工作的进展之前就要将它们消除。

12.6.2 没有正确获得构架的风险

最严重的风险之一是没有能建造一个在随后的阶段或在其生命周期中能逐步进化的系统，即没有建立一个有弹性的构架。当确定拥有一个正确的且可冻结的（除了在构造阶段的细微变动）构架时，就在初始阶段和细化阶段处理该风险。这就是在前面所谈到的把对某类风险的寻找和处理融入到统一过程中的真实意图。在这种情况下，比如，在初始阶段和细化阶段对构架问题进行明确的处理。

怎样判定哪些用况是获得正确构架的最重要的用况呢？应如何消除那些影响建立稳定构架的风险？要解决这些问题，可以搜寻对构架有重要意义的用况，这些用况覆盖系统所要完成的主要任务和功能。可以向自己提出这样的问题：我们为什么要建立这个系统？

这一问题的答案在关键用况（即那些对系统用户最重要的用况）中发现。此外，有很重要的非功能性需求（如性能和响应时间等）的用况也属于关键用况。这些用况通常有助于发现系统的骨架，在系统骨架之上再添加所需要的其他功能（参见7.2.4节）。

其他类型的用况有：

- 次要的。这些用况支持关键用况。它们涉及一些次要的功能，例如对操作数据的管理和编辑。既然项目相关人员对看到一些数据输出有浓厚的兴趣，就像在12.6.3节中描述的事务

费用的例子，那么这些用况在早期就需要进行开发，但大部分这类用况只对构架有中度的影响。将这些用况排列在较高的优先级，这是因为我们希望消除不能得到正确需求的风险。

- 附属的（有了也好）。这些用况不是构架和关键风险的要害。这个层次的用况很少进入初始阶段和细化阶段的迭代。如果进入了，也仅仅是因为偶然地为了补充关键用况或重要用况。
- 可选择的。一些用况尽管不是时常出现，却可能是关键的或重要的用况。需要处理这些用况，因为当它们出现时将影响构架。

此外，我们希望确定已经遍历了所有可能影响构架的用况。我们不希望在某个角落里遗漏任何功能，以至于发现太晚而不能拥有一个稳定的构架。需要广泛地覆盖那些可能影响构架的用况。广泛覆盖是十分重要的，不只是为了构架，还为了确保可以在第一个生命周期内精确地预测开发产品所需要的费用。我们必须避免因发现太晚以至于无法增添新发现的功能的风险。

这就是为什么需要在细化阶段覆盖大约80%的用况的原因。在这里，“覆盖”的含义是指了解用况以及它们可能对系统造成的影响。实际上，虽然要平均识别80%左右的用况并将它们包含在用况模型中，但我们发现通常没有必要对所有用况都加以详细描述。在一些典型的项目中，我们只需要描述一部分用况，其中一些描述可能十分简要，只用几行文字，只要足以说清楚在这个阶段中需要了解的内容。在相对简单的项目中，我们可以在处理需求时描述一小部分用况。在带有高风险的较大型项目中，我们会发现详细描述80%甚至更多的用况是一种明智的选择。

12.6.3 没有正确获得需求的风险

另一个严重的风险是没有能获得用户真正期望的功能的系统。对这种风险的处理也列入统一过程之中。在细化阶段结束时，我们希望确定：正在建立的是正确的系统。对这些风险的寻找不能往后拖延，因为在构造阶段开始投入大量的费用。哪些用况是用来确保开发的系统是用户所需要的？哪些用况是确保系统能在其他阶段继续进化的，即可向系统添加产品第一个版本所需要的各种功能？不能在某个角落里遗漏任何功能。需要了解所建立的系统可以继续发展。

答案的第一部分当然是要保证需求工作流的正确性，为此可以创建一个业务模型（或者在某些情况下，创建一个较为有限的领域模型）。答案的第二部分是，通过早期的迭代和原型，构建用户要求的系统并尽可能早地获得用户的反馈。当然最后只有通过真正的使用才能证明是否构建了正确的系统。

举例：“记账”及“支付”系统

在“记账”及“支付”系统中，我们假设银行会确定通过服务赚钱是十分重要的。银行可能希望从每一笔交易中收取一小部分费用。实现收费相对于“订购货物与服务”、“订单确认”、“买方账单”、“支付账单”等核心用况来讲是新添加的功能。然而，从结构的角度看，收费功能对于实现正确的构架是十分重要的。收费可以通过其他用况的扩充来处理；也可以与包含收费的几个用况相结合。对于开发人员而言，收费功能是十分普通的，他们在以前已经实现过收费的功能。然而，从客户角度看，在软件交付前就能正确执行收费用况是极其重

要的。正是基于这个原因，收费功能应作为高风险处理，所以它变得十分重要。

因此，当项目经理考虑迭代的顺序时，他不得不权衡收费功能的重要性。一方面，如果他发现当前的收费实例是一项十分简单的功能，对开发人员并没有什么挑战性，他可能会决定不需要开发人员在细化阶段来处理，并有可能将收费功能推迟至构造阶段来考虑。另一方面，如果他发现收费体现了一系列复杂的内部问题（从其他用况中分离出来），他应把对收费功能的处理规划成细化阶段的迭代的一部分。这些“复杂的问题”之一可能是客户需要早一点看到收费问题已经解决。

12.7 所需要的资源

可能你已经感觉到基于阶段的软件开发的迭代计划有相当多的优点，但也存在几个困扰你的问题：

- 初始阶段和细化阶段将要花费多少费用（根据工作量以及所需人员的资质两个方面）？
- 这些阶段所需的经费来自何处？
- 初始阶段和细化阶段将花费多少时间？
- 早期阶段要经过多少个月的延续才能进入大家认为的真正的软件开发过程（即构造阶段）？

12.7.1 项目间存在广泛的差异

目标软件系统在它们准备进行开发前当然存在着很大的差异，这并不是什么秘密。让我们来列举以下四个例子：

1) 未知领域的一个非先验性的全新产品。没有人对要做什么了解多少，这时基本上没有经验可以参照，只能依靠一些有经验的开发人员作一些猜想。在这种情况下，从某种意义上说，需要使用该系统的人有责任为初始阶段和细化阶段提供经费。这些阶段是基于费用的，需要提供经费以进行开发，具有作研究工作的性质。在初始阶段或细化阶段以固定的预算和进度来掌握和控制项目是不够的。在初始阶段中，决定系统的范围，寻找候选的构架，识别关键的风险，生成业务案例等都是需要耗费时间的任务。同样，为了达到细化阶段的目标，也就是要使项目进展能够达到可以制定构造阶段计划，将耗费更多的时间。
2) 在某一领域中已经做过的产品。虽然该领域的早期产品可以充当例子，但没有可重用的构件。这些原有的产品给候选的构架提供指导，但可能还要花几天时间来确认原来的构架是否与现在的产品真正相适应。在这种情况下，初始阶段可能会十分短暂（几个星期）。它可能只需要一两个有经验的全职人员，但它可能要利用小型开发组中有经验的开发人员作为咨询对象来提供知识。因为这种类型的产品已经开发过，主要的风险可能并不存在，但可能有必要花费几天时间对这一点加以证实。

3) 旧的系统已经存在，但要对它进行移植或升级，例如从主机模式移植成客户机/服务器模式。在一定程度上，部分旧系统的代码可在新系统中加以封装后再使用。在初始阶段中，开发组必须找到候选的构架，因为其他组织已经重用过旧的产品，开发组知道这样做是可行的。如果软件开发组织准备做一些过去没有做过的工作，那么所需的费用和时间就不得而知了。这时，就必须设计构架基线，必须从用况以及寻找子系统（其中的一个子系统可能封装了旧系统不需改变的部分）开始，着手识别新老系统之间的接口。

4) 构件已经存在于商业市场或开发组织内部。软件开发组织希望新系统中有相当大的一部分（50%到90%）能够由这些构件组成，但这些构件之间可能存在间隙而需要一些新的代码来衔接。项目要求识别并详细描述重用构件和新构件之间的接口以及外部系统与用户之间的接口。从构件开始开发也需要耗费人力和时间，开发组也有可能遇到风险。然而，总的来说，基于构件的开发比从底层开始开发要更快并且更节省费用。

并不是要对这些例子进行严格的分类，相反，它们之间可能存在着重叠，因此必须基于初始状态去考虑。开发组织在应用领域有多少经验？可以利用多少重用构件？这样的项目建议是否比发布已有产品的新版本包含更多的好内容？这样做是否可以推动当前技术的发展？是否是分布式系统（这里只说明了在单一平台上工作的情况）？

12.7.2 一个典型的项目

先不管不同的初始状态所导致的不确定性，一个中等规模的项目最初的开发周期将大致如图12-4所示去分配人力和安排进度。相对于基于瀑布模型的开发而言，为了设计和安排构架以及降低风险，基于阶段的开发把工作转移到了开发周期的前期。

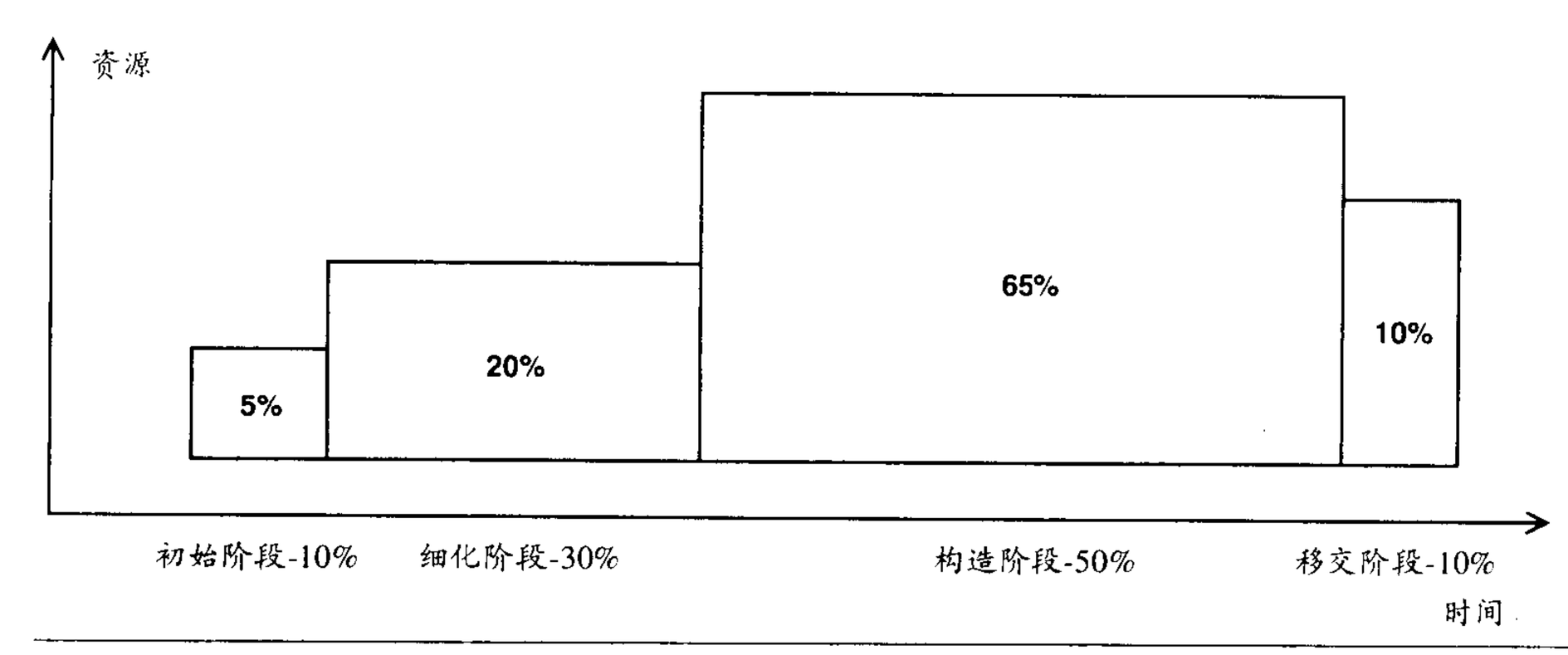


图12-4 构造阶段所需的工作量和时间最多，但其他三个阶段也需要相当数量的时间和工作量

12.7.3 复杂项目具有更大的需要

假定有一个更大、更复杂的项目，它需要新的功能，采用分布式的构架，或者是严格的实时操作，这说明该项目还将使用新的潜在技术，此时会产生什么影响呢？其结果很可能要进行更多次迭代，要在初始阶段和细化阶段投入更多的时间和工作量。结果这两个阶段的资源可能

如图12-5所示那样有所增长。

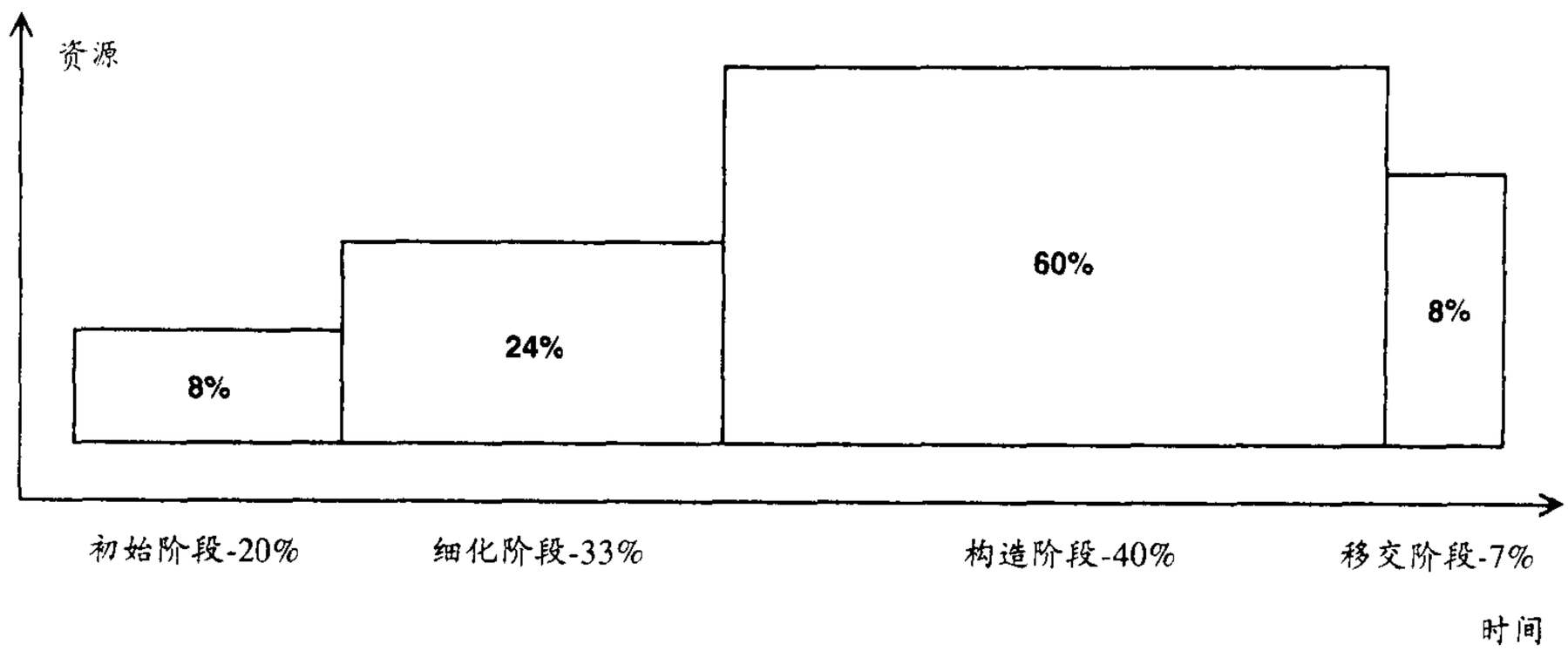


图12-5 在一个难度更大的项目中，相对后面的阶段而言，前面的阶段需要做更多的工作。前期阶段主要会花费更多的时间

要强调的是，在这些图中所表示的百分比都是假定的，不应武断地用于真正的项目。这些图只是想指出一点：对于现有项目不了解的部分越多，在初始阶段和细化阶段就要花费更多的时间和工作量。然而，例如在这个例子中，在初始阶段和细化阶段所增加的时间比另外两个阶段所增加的时间要多；对投入的工作量并无相同的影响，因此，虽然资源也有所增加，但无需增加相同的比例。同时，投入项目的工作量和时间的增加应作为一个整体来考虑。

框架的使用明显缩短了构造阶段，但对构造阶段以前的阶段影响不大。如果需要考虑重用的话，初始阶段和细化阶段将更长，但作为框架可以直接利用的功能不需要进行分析和设计，所以从整体上而言，项目需要的时间和工作量会减少。

12.7.4 新的产品线要求经验

在大多数情况下，对于全新的或是复杂的系统，开发组需要获取自身所没有掌握的信息。而这种信息的自然资源是在目标系统所在领域有着丰富知识的人员。甚至当有详细的需求可供利用时，开发组也需要会见这些人员以创建构架和汇集风险。寻找具有丰富知识的开发人员可以说是战斗的一半。普通用户可能只对整个过程中他们自己所在的部分有所了解，而并不知道计算机系统能做些什么。

新产品线刚开始工作时经常出现的失败是试图在没有重用知识的情况下进行工作。因为有关公司实际运作方式的大部分知识是驻留在人的头脑里，而不是驻留在文档（通常也不是用任何方式读出的）中，重用知识基本上体现为有经验的人员的“重用”。这种失败是因为将新人分派到最初的项目组中，而不是选派在执行公司业务方面有经验的人员（如果本身不在新的产品线中）去做这项工作。

当公司计划开发一条新的产品线时，他们知道这项工作需要最有竞争力和经验的人员。然而，公司同时又会想到具有这种才干的人员对公司现有业务的运行也是十分重要的。他们必须满足现有客户的需求。公司必须保持现有资金的流动。

这种左右为难的状况并不是轻易可以解决的。这批有经验人员的主管心顾两头，显然，现有的生产线和新生产线都十分重要。实际上，许多高级主管已在现有生产线上工作多年，从心理上，他们已经与其相融合了。通常，大部分管理人员把他们手下的重要人员放到新的项目中会使他们现在负责的业务冒一定的风险。结果是新的开发组中只有少数几个有经验的人员，并且他们可能还不是管理和技术方面的最好的领导者。他们是那些管理人员愿意放走的人。这些新的小组领导不得不增加大量新人，这些新人通常是直接从大学招聘的。

除了缺乏经验之外，这些直接从大学招聘的新的开发人员将抱着很特殊的心态。他们总是将公司已经做过的东西视为过时的，只有新的才是最好的。这些新的开发人员并没有注意到他们没有在学校学到的东西的重要性，如生产方法和生命周期管理。

我们注意到公司往往不能为新产品的开发配备开发所需要的有才干的全职人员。这一不足往往到产品的第二版才能得以弥补。从本质上讲，第一代产品的结果只是一个有关领域的实验而已，尽管这可能并不是公司的初衷。

12.7.5 资源消耗所需费用的支付

要支持即使是很小的开发组在最初的两个阶段进行工作，也需要花费资金。而实际上，获取经费以支持这两个阶段的困难已经成为迫使软件组织提前着手进行构造阶段的因素之一，有时这导致了严重的失败。那么，这些经费来自于什么地方呢？

- 如果软件组织生产的是用于出售的产品，那么经费来自企业的一般管理费用，并且处于管理的控制之中。然而从更长远来看，经费应该来自顾客或客户。换句话说，软件组织必须在现有的价格中包含开发今后产品的费用。
- 如果软件组织是为内部客户生产产品，那么最初两个阶段的费用来自于组织自身的一般管理费用，或者来自从客户转账过来的资金，或者来自高层管理人员分配的资金。后两种筹措资金的方法说明，软件组织外部的管理人员必须理解开发工作早期阶段的价值，并拨款支持他们。
- 如果软件组织是为单个的公司类客户生产产品，那么前两个阶段的费用可能来自于其本身的一般管理费用。然而，只有组织将这些花费列入过去的标书中，才有可供使用的经费。如果下一个项目满足软件组织的正常的业务活动，则这些经费来源已经足够。如果目标项目在这些正常的限制下看起来具有风险，则应由客户提供最初两个阶段的费用。

事实上重要的工作是在最初的两个阶段完成的，并且要完成它需要耗费时间和经费。进一步来讲，除了经费之外，完成这些工作还需要客户的合作。而合作（软件组织从客户方提供的人员那里获取所需要的信息）也需要经费（尽管这部分经费可能不被正式地记入账户）。

12.8 迭代和阶段的评估

如果要全部实现迭代开发方法的优点，项目需要在每一次迭代和阶段的结束点对其所完成的内容进行评估。一般由项目经理负责评估工作，这样做并不只是为了评估迭代本身，还是为

了以下两个更深的目的：

- 为了利用开发组在本次迭代中所学到的知识来重新制定下一次迭代计划，并作任何必要的变动。
- 为了修改过程，选择合适的开发工具，扩大人员培训，并按照迭代评估所获得的经验对过程改进提出建议。

评估的第一个目标是检验已完成的内容是否满足预定的评估准则。第二个目标是根据迭代或项目计划审查所取得的进展。

- 开发工作是否符合经费预算和进度安排？
- 通过对原型、构件、构造的测试或者观察，或者通过项目相关人员所关心的增量操作，来判断其结果是否满足质量要求？

理想的情况是，项目将满足这些准则。项目经理将评估结果分发给项目相关人员，并且将文档归档。要用新的文档来报告新的评估结果时，并不需要更新原有的文档。

12.8.1 没有达标时的处理

评估工作做起来并不是那么简单。通常，迭代不会充分达到准则。项目不得不在下一次迭代（或者在某一特定的迭代）中来完成这些工作。这些工作涉及：

- 修改或扩充用况模型。
- 修改或扩充构架。
- 修改或扩充正在开发的子系统。
- 搜寻更深层的风险。
- 为开发组增添某些技能或背景知识。

否则，需要分配更多的时间来完成已有的计划。如果是这种情况，则需要延长第一次迭代的时间，且应该指定一个固定的完成日期。

12.8.2 准则本身

然而，还有一点需要考虑，就是评估准则本身。也许，在还没有掌握全部可用的相关信息时，开发组已经制定了准则。在迭代中，可能会发现额外的需求或发现原来列出的需求是不必要的。因此，评估人员不仅仅只是检测是否达到了准则，还要对准则进行必要的修改。

12.8.3 下一次迭代

主要里程碑（参见附录C）标志着一个阶段的结束，标志着不仅是项目组而且是项目相关人员（尤其是用户方的投资部门及其代表）都一致认可项目已达到里程碑准则，并同意进入下一个阶段。

在评估的基础上，项目经理（由一部分在本次迭代或阶段中工作的人员充当助手，增加一些将安排到下一次迭代开发的人员）做如下一些工作：

- 确定工作已经就绪，可以进入下一次迭代。

- 如果需要返工，确定要进行的迭代是后面的哪一次迭代。
- 为下一次迭代制定详细计划。
- 更新下一次迭代以后的迭代计划，不要求太详细。
- 更新风险清单和项目计划。
- 把该迭代的实际耗费和进度与原计划相应的部分进行比较。

需要指出的是，在基于构件的开发中，度量（即完成的代码行数）并不是工作进展可靠的指标。因为开发人员可以重用已设计的构造块（子系统、类、构件），所以，只要写很少的新代码，就能使工作有一个很好的进展。

12.8.4 模型集的进化

基于阶段的迭代开发过程的一个关键特性是模型集的进化。将这个进化过程与瀑布模型相对比，在瀑布模型中，首先是完成需求，然后是分析，等等。在迭代开发中，各种模型如同一个协调的集体在阶段之间共同进化。在早期的迭代中，一些模型先于另一些模型进化。例如，用况模型先于实现模型进化。整个系统由所处的一个状态向更高级的另一个状态进化，来代替相互独立的模型的进化，这一点在图5-7中已表示出来了。每一次迭代（也可能是一个迭代中的每一个构造）都代表整个系统状态的进展，这一进展反映为模型集的逐步进化直至完成为止。在每一次评估中，这一进化进展的程度是评估组要考虑的一个重要指标。

下一章将回到项目的开始，考虑初始阶段自身的优点。

第13章
初始阶段启动项目

初始阶段的全部目的是要启动项目。在初始阶段之前，你可能只是有一个不错的想法。如果该想法是一个新领域的全新想法，则这将是一件激动人心的事；如果该想法只是业已存在的产品的一个新版本，则心情会十分平静。初始阶段可能相当简单，只需要由一个开发人员收集一些对产品的构想陈述，用不同的图勾画构架，并生成一个合理的业务案例。当然，初始阶段也有可能像一个完整的研究项目那么复杂。关键在于，我们并没有将初始阶段仅仅定型为一种模式。经过初始阶段，即使系统是全新的，也应该限定待解决问题的范围，并且给自己以足够的信心——即开发这个系统既是值得的，也是可能的。

当然，这种信心并不仅只属于一个人。初始阶段所做的工作给客户（或是其代表）、开发组织以及其他项目相关人员都提供了保证：构架设计师和开发人员能够缓解关键风险，明确地表达候选构架，并生成最初的业务案例。

13.1 初始阶段概述

初始阶段的目标是生成具有必要内容的业务案例，以证明启动项目是正确的。为了生成这一案例，首先必须划定目标系统的范围。知道了这个范围，就可以弄清当前项目所包括的内容。了解哪些内容是构架所必须包括的，就可以定义关键风险的寻找范围，并用它来规定业务案例中费用、开发进度以及投资回报等因素的评估范围。

要尽量获取足够多的构架内容，以确信存在能够支持系统范围的构架。这也是“候选构架”所要表达的意图。

要设法避免彻底的失败。许多复杂项目陷入困境是因为这些项目遇到了关键风险，这些风险或许是直到系统集成和测试时才发生的，并且不能在经费预算和进度安排所允许的范围内予以缓解。不管你是否喜欢，这种不确定性或风险始终存在。我们只能通过及早缓解风险、缩减风险发生的范围或者通过增加时间或资源（即经费）的投入来避免那些无法缓解的风险。

设法建立最初的业务案例，以便于从经济学角度考察预想的系统，也就是要对费用、开发进度和投资回报及早作出估计。此时要考察下面两个问题：

- 由软件产品的使用或出售所获得的利润是否超过用于开发的费用？
- 是否能及时进入市场（或内部使用）以获取利润？

尽力给项目组提供开发环境——过程和工具。当然，软件开发组织的开发环境是多年努力的结果，大部分在给定的项目启动之前就已经具备了。在初始阶段所要努力解决的问题是：根据

待开发系统的类型和即将从事这一开发的软件开发组织所具有的能力来裁剪统一过程。

13.2 初始阶段初期

开始制定计划（*plan*），扩展系统构想（*vision*），并开始收集评价准则（*evaluation criteria*）。

13.2.1 初始阶段开始之前

在初始阶段开始之前，你对所要做的事情已有一定的了解。例如，某人（客户组织、内部市场人员或者甚至是内部开发组织中的开发人员）有一个想法并且证明它是合理的，就开始启动某些工作。尽管在初始阶段开始之前所做事情的程度各有不同，跨越了一个很宽的范围，但是仍然可以将其分成三种不同的情况：

- 开发上市产品的软件开发机构。这些软件开发机构开始开发时所具有的信息量可能十分丰富。市场人员和管理人员可能对下一个产品进行了相当多的研究，而且已经有一些开发人员在从事探索性研究。换句话说，他们实际上已经做了一些初始阶段的工作。
- 为同一公司的其他部门开发系统的软件开发机构，即典型的内部开发机构。从广义上讲，我们会遇到两种不同的情况。第一种情况，一个部门发觉需要一个软件系统并且请软件开发机构来开发这一系统。提出这一需要的部门将会对它所需要的系统进行描述，然而这些描述可能很少涉及对软件的要求。换句话说，软件开发机构在初始阶段起步时几乎没有掌握什么信息。第二种情况，管理部门觉得需要一个在公司范围内使用的系统，并且已安排一个业务工程小组对系统应该支持的内容进行了研究。在这种情况下，可能在开始初始阶段时就对需求有了一个很好的了解。
- 为客户开发系统的软件开发机构。对计划的初始要求往往包含着相当多的细节，其需求描述或许有很多页。而在另一种情况下，客户关系不很正规，其客户对其需要可能只有大致的构想。

如果产品是从早期版本进化而来的，那么计划这个新项目的第一次迭代要做的大部分工作可能在前一个周期的最后一次迭代中已经完成。然而，如果是一个新的大型项目，创始人希望看到他们的思想得到进一步的发展。他们可能已经对所要完成的工作量、初始阶段所需要的经费以及开发进度有了一个大概的估计。

13.2.2 制定初始阶段计划

在项目开始时，你可能会陷入进退两难的境地。有经验的开发人员会建议你制定计划，但你又没有制定计划所需要的大量信息。这时，你可以通过问自己“需要做什么”来搞清楚要计划的内容是什么。这些内容将在13.3节和13.4节中介绍。现在，可以采取如下步骤开始你的工作：

- 将开发人员在项目开始之前所搜集的信息集中起来。
- 把这些信息组织整理以便于使用。

- 召集一批知道如何使用这些信息的人员。
- 研究提出需要进一步了解的内容（不是针对所有的四个阶段，而是针对初始阶段中十分有限的几个目标）。

换句话说，将工作限定在该阶段需要完成的几个关键目标上，这一点已在13.1节中进行了总结。然后，制定一个试验性的计划，以澄清与这些初始目标有关的需求，并在相应的用况中详细描述。制定用来创建一个候选构架的计划，该计划要将构架完善到恰能足以确定该项目是切实可行的，通常这个计划仅是构架视图的一个草图。要尽快地设法确定究竟需要一次迭代，还是在某些情况下需要两次迭代。

要记住，最初的计划只是试验性的（*tentative*）。随着所掌握的信息量的增多，要对计划进行修改，以包含新发现的内容。当然，要尽量在管理部门（或客户）最初规定的开发进度和开发经费的限制范围内完成初始阶段的工作。因为管理部门只是根据有限的知识设定这些数字，所以在该阶段要让管理部门（和项目相关人员）了解实际的进展情况。这可能会证明有必要对这些数字作出修改。

13.2.3 扩展系统构想

在生命周期开始时，开发组可以利用的信息可能只是比一页纸的内容稍多一点的构想陈述。它包括一个特征清单（参见6.3节）、少量性能信息以及与开发人员可能遇到的风险有关的少量知识，也许还会模糊地提到可能的构架（也许仅仅是“客户机/服务器”之类的名词）以及从经济学角度的一个大致预测（如1千万美元和两年时间）。

初始阶段的开发组包括项目经理、构架设计师、一两个有经验的开发人员、一个测试工程师（尤其是在可能出现大量测试问题的情况下），很可能还包括客户或用户的代表。第一步工作是扩展构想陈述，使其可以指导初始阶段的工作。

这一点说起来很容易，但做起来却很困难。这就是为什么要牵涉到利益的各方代表的原因；这就是为什么参加的人员应该有经验的原因；这就是为什么要统筹考虑不同的观点，而不仅仅只是折衷的原因；这也就是为什么要花一点时间来考虑这一复杂的任务的原因。要记住我们不是在寻求多数人能够接受的一致意见，而是在寻找最好的答案。正确的答案来自于领导，来自于负责该项目经费的管理人员，也来自于受学识渊博的专家指点的领导。

如果要开发的产品与开发组曾经开发的产品类似，开发人员可能只需寻找其相似点，这一工作可能只需花费几天时间。如果是一个已有的简单产品的二次开发这一类情况，则其初始阶段只需要一天，而一个非先验性项目的初始阶段可能要花费几个月的时间。

如果要开发的产品是具有高度创造性的系统，则其初始阶段的时间可能会非常冗长，费用也会非常昂贵。

13.2.4 设立评价准则

当项目经理掌握了足够多的信息来开始为第一次迭代制定详细计划时，他也将制定表明迭代达到预期目标的评价准则。由于信息的缺乏，这一针对首次迭代的“详细”计划实际上可能

是十分粗糙的。如果是这种情况，评价准则可能是相当通用的。随着迭代的进行和所掌握信息量的增多，项目经理会精化这些准则。针对初始阶段的四个目标，我们提供一些通用的评价准则。

确定系统范围

毫无疑问，对系统的最初构想提出了关于系统范围的一些想法，但没有精确地定义系统的范围。在初始阶段，项目划出一条确切的界限，指出什么是属于目标系统之内的以及什么是属于目标系统之外的。然后要确认外部的参与者（参与者可能是人或其他的系统），并确认系统和参与者的交互作用。这就在一个较高的层次上说明了项目的这一交互作用的性质，其要点包括：

- 是否清楚哪些部分在系统之内？
- 是否识别了所有的参与者？
- 是否弄清楚了跟参与者的各个接口（用户界面和通信协议）的共性？
- 在系统范围之内的什么部分能作为一个功能系统支持它自身？

解决本阶段所需要的需求中的不确定性问题

初始阶段开始时的需求可能处在泛泛的构想和大量的文字描述之间。然而，这些初始需求可能比较含糊。评价的要点包括：

- 达到该阶段目标所需的有限数量的用况需求（功能性或非功能性的）是否已经被识别和详细描述？
- 补充需求（参见6.7节）中的需求是否已被识别和详细描述？

建立候选构架

就经验而言，初始阶段的开发人员能够很快将注意力集中到新的功能或是需要崭新性能的功能上。在这些功能中，开发人员不得不选出一些可能使整个系统开发陷入困境的新功能。对于这些新功能，开发人员至少要开发出一个可用的构架。评价准则包括：

- 是否能够满足用户的需要？
- 构架是否可用？（根据候选构架所能达到的程度来考虑这一标准。这是因为目前还没有原型，例如可根据构架“承诺”要做的工作来对构架描述作出评价。）

要回答这一问题，必须考虑几个要点。建立构架所采用的技术（数据库、网络等）是否恰当和可行？构架是否高效？构架能否使用资源？构架是否可靠并具有容错能力？构架是否健壮并具有弹性？随着需求的增加，构架能否平稳地进化？

缓解关键风险

关键风险是指那些会危害项目成功的风险。评价的要点包括：

- 是否已识别出所有的关键风险？
- 已识别的风险是否已经缓解或者是否已制定出缓解的计划？

“缓解”并不是意味着在这个阶段将风险彻底消除。它可能表示客户因不愿面对项目可能失败而愿意修改相关的需求；它也可能表示项目组预见到解决风险的途径，但直到随后的某个阶段才设计出细节；另外，它还可能表示项目组发现减小风险发生可能性的方法或项目组发现在风险发生时可以将严重程度降至最低的方法；它还可能表示已制定出计划以应付风险的发生。以上这些推断都不是简单、机械的判断结果，而往往介于上述几种情况之间。初始阶段的主要

目标是在软件开发过程中提供一个时间点，让项目经理、高级管理人员和项目相关人员（包括客户）能够进行业务判断。

判断初始业务案例的价值

评价的要点是：初始业务案例是否足以说明继续进行该项目的合理性？

初始业务案例涉及到该项目主管人员所负责的另外的要求准确判断的项目领域。在一个熟悉的领域中，你在初始阶段结束时很可能就有了可观的业务案例数。然而，在一个全新的具有一定难度的领域中，你可能只有一个很宽的，仅供作出基本判断的数字范围。

13.3 原型的初始迭代工作流

在初始阶段，进行三组不同的活动。一是制定迭代计划，如12.4~12.7节和13.2.3节所述；二是五个核心工作流，这部分内容在13.3.1节中简单提及并在13.4节中详细讨论；三是选择适合于该项目的开发环境，如13.3.2节中所述。另外，13.5节中描述了初始业务案例，13.6节介绍了对这一阶段的评估。

13.3.1 五个核心工作流引言

初始阶段的主要目标是建立业务案例，即来自所从事项目的业务角度的案例。为了达到这一目标，需要找出目标系统的范围，勾画出构架，识别出决定项目成功与否的关键风险，并且制定缓解风险的计划。如果目标是一个新型的系统，则必须通过建立原型（概念证明原型，通常会被抛弃）来证明系统及其应用。原型也可被有选择地用于管理和缓解风险：识别出高风险区，并且将系统中处理复杂功能或已知性能或可靠性问题的关键部分原型化。例如，在一个容错的金融交易系统中，你可能希望尽早建立一个错误机制的原型。

初始阶段的大部分工作都在第一个工作流（需求）中完成，如图13-1所示；其中有一些工作在分析和设计工作流中进行；在最后的实现和测试工作流中只有少量的工作。在这个阶段中，所关注的是要证明新的概念，而不是要确保这种探索性的原型在任何细节处都能正确运行。

系统分析人员识别出用况和参与者以定义系统的范围。构架设计师对这些用况进行优先级排序，选出与候选构架有关的用况。他或她为候选构架准备最初的构架描述。用况说明人员通过这些识别出的用况来详细描述一些路径。这些详细描述的用况是那些与弄清楚系统范围和候选构架以及关键风险有关的用况，也就是与建立初始业务案例有关的用况。如果项目的开发是非先验性的，这步工作的一个成果就是最初的用况模型；如果是在一个已存在项目的基础上的进一步开发，那么这步工作的一个成果就是用况模型的一个新版本。此外，还可能要准备一个用况优先级清单，这在第12章中已讨论过。

在分析工作流中，我们为要在初始阶段中处理的大部分用况和用况的场景建立最初的分析模型。分析模型对于清楚地理解用况是十分重要的，对于理解构架的第一个草案的根据也是非常重要的。

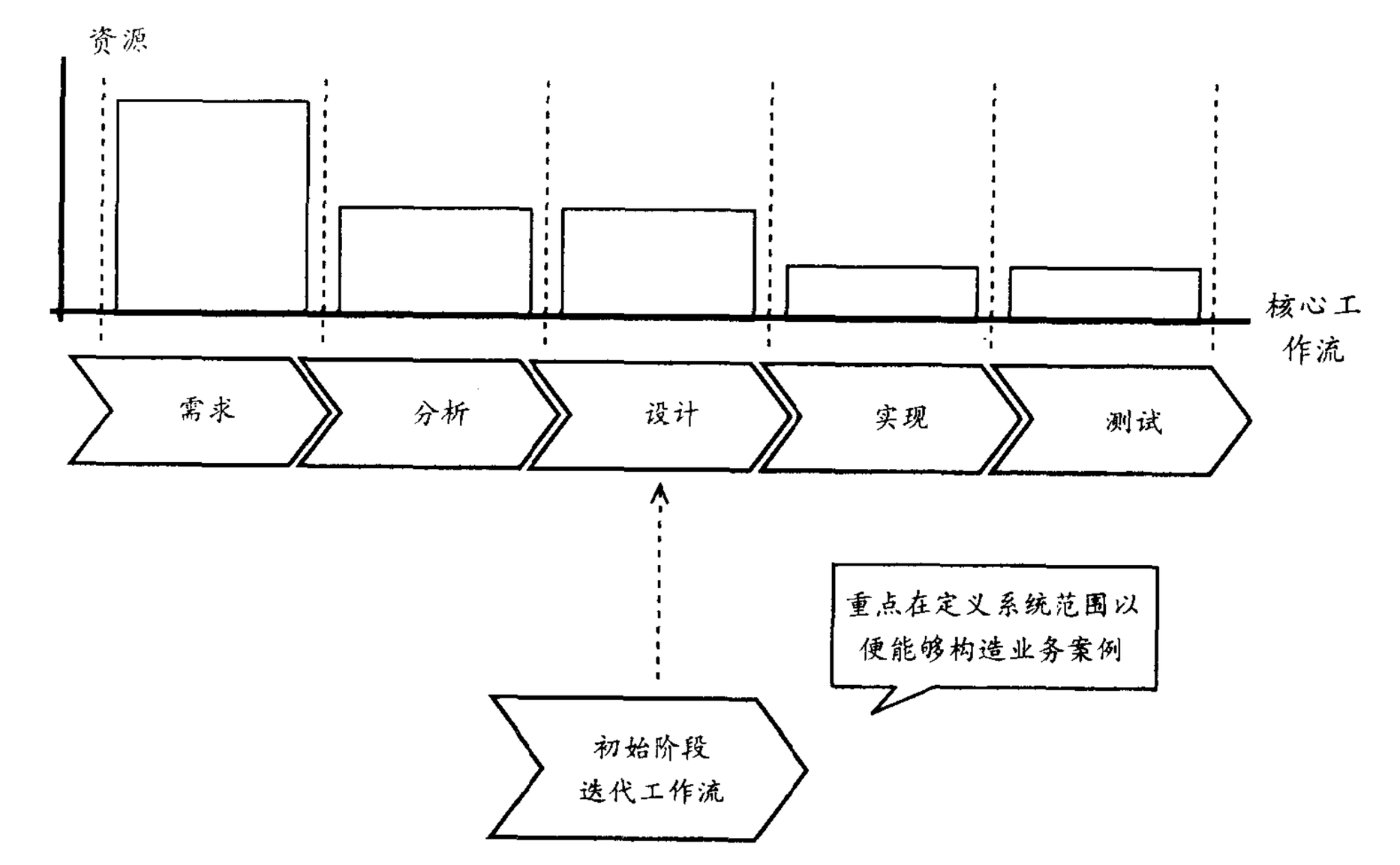


图13-1 初始阶段中一次迭代工作按五个核心工作流进行。矩形框的相应大小仅仅表示哪些工作流需要更多注意，哪些工作流则可较少关心

如果系统是新的或是无先例的，初始阶段中的开发组可能会通过迅速地集中某些东西来开发一个以证明关键想法的探索性原型。该原型的目标是要表达有关的思想，而不是要进化为最终的实现；换句话说，它有可能被抛弃。在某些情况下，倘若开发组发现构件工程师能够毫无问题地实现一个满足所需性能的构件算法，则可能只需要证实存在一种可以进行一种新计算的算法就足够了。

一旦项目经理认为候选构架看上去可以运作，并且风险低于标准或者已经缓解，该迭代就可以以“候选构架描述”（包括模型视图的概要）的形式结束。根据项目和项目相关人员的经验，在很好地建立构架时，这样的情况就会发生。

13.3.2 把项目融入到开发环境中

开发环境由过程、执行过程的工具以及为项目提供的服务所组成。开发环境还包括过程的配置和改善，工具的选择、获取和制造，技术服务，培训以及咨询等。

工具包括支持需求、分析、设计、实现和测试五个核心工作流的工具，也包括项目管理(计划、估计和跟踪)工具、配置和变动管理工具、文档生成以及过程在线工具。除了那些为了达到通用目的而从市场上获得的工具之外，为了满足特定的需要，可能会在项目内或者在公司内开发专用的工具。服务是指系统管理、数据备份和远程通信等。

在初始阶段，要致力于使软件开发组织的开发环境支撑项目的开展。随着项目遇到对于工具支持和其他服务的更大需求，该工作会一直持续到细化阶段。项目组织实现其软件开发环境的工作是与该阶段的其他工作同时进行的。

13.3.3 找出关键风险

在第12章的12.5节讨论了识别和缓解影响开发的风险。初始阶段的任务是要识别和缓解关键风险。关键风险是使项目不能实现的风险。在13.2.4节中，我们讨论了要缓解关键风险。寻找这一级别的风险是初始阶段期间的头等大事。如果发现存在这样的风险，但又不能找到缓解它的办法或者不能制定抑制该风险的计划，就必须考虑放弃这个项目。

13.4 执行五个核心工作流——从捕获需求到测试

本节将详细介绍在本阶段所要做的工作。描述涵盖了对一个非先验性的产品（即从头开始的一个新产品）在初始阶段所要做的工作。对于一个已有产品的进一步开发通常要简单得多。

每一次迭代都是一个最小化的瀑布模型，经历了从需求到测试的所有五个核心工作流。在第二部分用单独的章节分别描述了五个核心工作流，其中需求工作流用了两章。本节将用每个子节分别讨论每个工作流，但在初始阶段将对它们加以综合考虑。为了表示什么是重点，在图13-2中画了两个圈。一个圈圈出了寻找系统范围的活动，另一个圈圈出了与理解候选构架有关的活动。

图中的工作人员代表开发期间人们所充当的角色。然而，如果在这个阶段开发组比较小，则其成员将充当多个工作人员的角色。

13.4.1 捕获需求

初始阶段的重点主要放在第一个核心工作流即需求工作流上。这一工作流包括识别和详细描述与本阶段相关的用况。它包括以下几个主题（这些在第6章中已有所介绍）：

1) 列出作为系统特征清单候选的需求。

2) 理解系统语境。

3) 以用况的形式捕获有关的功能性需求。

4) 捕获有关的非功能性需求。

列出候选需求

系统特征通常来自于客户或用户在以前的系统或类似系统中所获得的经验。对于具有很大销售量的产品而言，新的特征来自于市场的需要。另外，开发机构的工作人员也会构思出众多可能的特征。一些特征来源于软件机构和用户之间的交互，一些特征来源于市场。来源于各处的特征就成为了特征清单的候选需求，这些内容已经在第6章中作过介绍。

理解系统语境

如果客户有一个业务模型（或领域模型），则开发人员在初始阶段就可以对它进行处理。如果客户目前还没有这么一个模型，就应该鼓励客户开发一个，尽管这样做所需要的时间和资源通常会远远超过处理单个项目所需要的时间和资源，但这是一个积极进取的方案。

应该为业务或技术系统识别出用况（它们将告诉你要支持什么过程），还应确认工作人员和核心工作成果（工作单元和业务实体）。总而言之，在进入详细描述有关用况的建模之前，应该

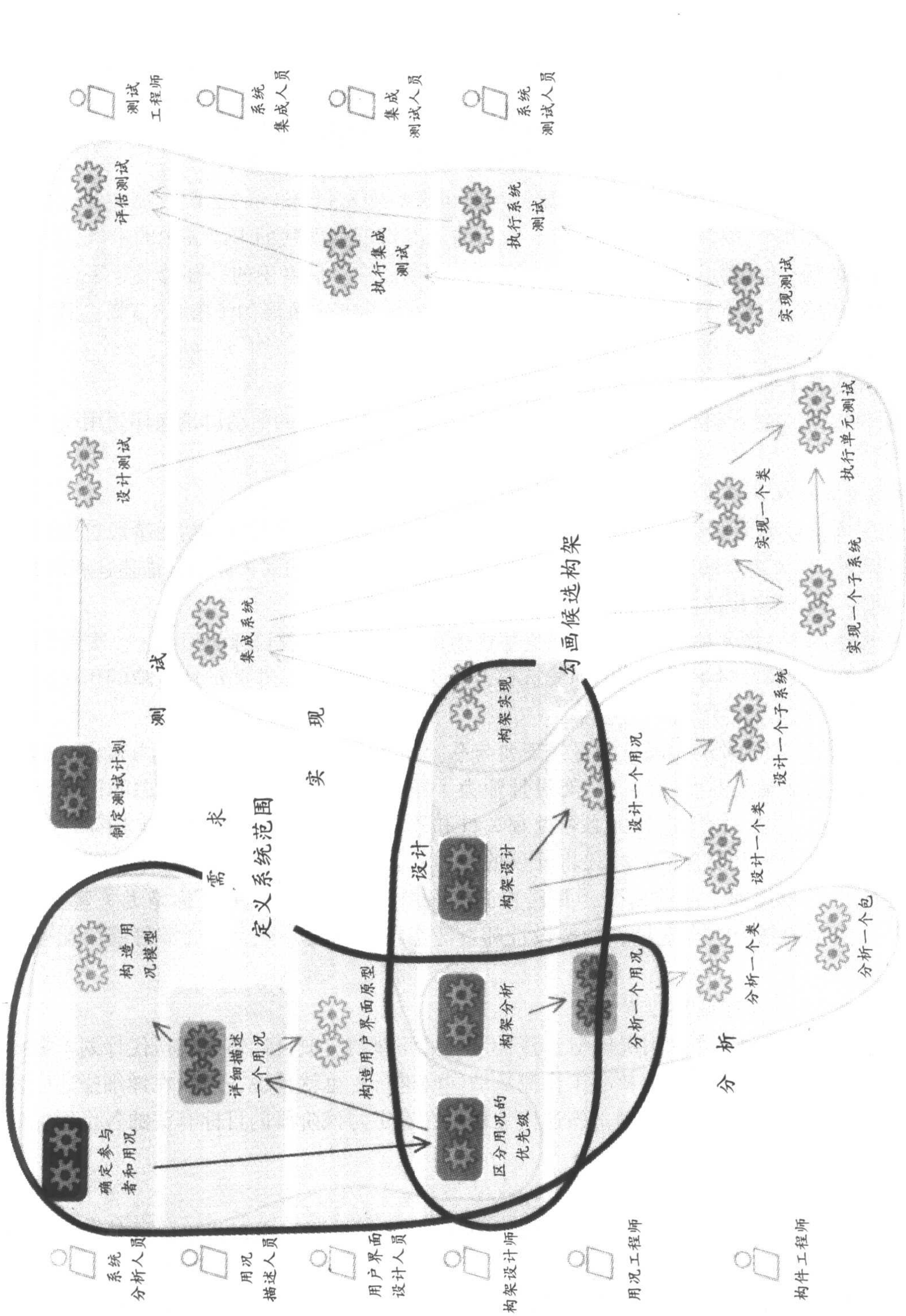


图13-2 粗边线所圈定的部分重点是初始阶段的主要活动：定义系统范围和勾画候选构架。不用在意该图的外观细节，它仅是对在第6章到11章中所阐述的工作人员和制品的一个简单的示意

已经完成业务模型的50%~70%的工作。如果你没有搞清楚支持软件开发的过程，你就无法开发软件。

捕获功能性需求

功能性需求以用况来表示，这一点在第7章和13.4.1.1节中讨论。

捕获非功能性需求

特定用况的非功能性需求也与它们所适用的用况相联系(见6.3节和7.2.3.2节)。那些针对业务和领域模型中某个特定对象的非功能性需求，成为与用况模型相关联的术语表中的术语；而那些数量上少得多的更具共性的非功能性需求，则成为辅助需求。后者中的一部分对于平台、系统软件和中间件构架的选择十分重要，在本阶段中它们也将是非常重要的（参见4.3节“用况及构架”）

13.4.1.1 以用况捕获需求

现在我们回到图13-2，讨论其中的活动。本节按照图13-2中所示的活动讨论怎样用用况来捕获需求。

确定参与者和用况（参见7.4.1节）

“完全的”需求或用况模型的完成都超出了初始阶段的资源。开发人员在初始阶段首先必须整理完成本阶段工作所需要的用况片断，其次，要像第7章中所阐明的那样详细地描述这些用况。在12.6节中简述了如何寻找这些用况。

此外，初始阶段要将大量的工作限定在主要描述那些影响该阶段目标的用况上。要忽略枝节性问题，或者跳过那些对于系统范围和候选构架并不重要的用况、非关键性风险的用况以及对初始业务案例影响不大的用况。

为了整理初始阶段处理的用况，项目经理要与系统分析人员和构架设计师通力合作。系统分析人员主要考虑系统范围的设定。构架设计师为了排除那些不需要进一步考虑的用况而要对整个用况作通盘考虑。构架设计师必须处理关键和重大的风险，确认那些制定构架计划所需要的用况，处理一些主要在下一阶段将开展的工作。在系统的某些部分，如果构架设计师在先前的系统中已经用到了这些用况，并且知道这些用况对于本阶段的目标是无关紧要的，则他只需投入少量的精力。在这里，构架设计师要考虑所有用况，至少要弄清哪些用况可以暂不考虑。

区分用况的优先级（参见7.4.2节）

下一步，构架设计师要考虑用况模型，并且为项目经理制定项目计划（或迭代计划）提供输入数据。这一工作与每一阶段的核心工作流活动同时进行；也就是说，可以在详细描述已发现的用况的同时制定将来的迭代计划。该迭代计划是在确定与该阶段的目标有关的各个用况的优先级排序之后的最终结果。

详细描述一个用况（参见7.4.3节）

然而，在接受这种工作限制的现实的同时，我们要反复强调完成在初始阶段处理的用况的所有相关内容（即设定系统范围所需的用况、制定缓解关键风险的计划以及制定基于构架基线的工作计划）的重要性。在绝大多数情况下，人们总是认为已经理解了需求，但实际上他们却忽视了一些关键的需求。他们倾向于认为详细描述所需用况的好处与所花费的

费用相比是不值得的。然而相反的是，那样做是值得的，始终提供用况的详细描述是非常必要的。

我们的目标是要知道将工作进行到什么程度。然而，这并不表示要用到非常多的用况。通常，由于不需要在本阶段生成构架原型，所以根本不需要经历实现和测试。然而，如果希望通过一个抛弃型的原型来证明核心的思想，就不得不考虑为了原型而解释一小部分**用况群**（附录C）。你可能要在用况模型中详细描述其中的一部分用况群，例如10%的用况群。

这样的说法表明，可能会找出许多用况，但仅仅详细描述其中的一部分，并进而对其中的有关部分进行更为详细的描述。例如，如果我们选出最终用况的50%作为可能与本阶段相关的用况，然后对于每个用况只找出其20%的场景进行进一步详述，那么，就可以计算出需要详细描述的用况仅占整个用况群的10%。关键在于要尽力降低初始阶段的开发费用，加快初始阶段的开发进度。各个项目的实际百分比依赖于该项目的复杂程度或该项目的独特情况。

如第7章中所述，该阶段所需要的所有功能性需求都以用况来表示。

构造用户界面原型（参见7.4.4节）

本阶段对其不感兴趣。

构造用况模型（参见7.4.5节）

在这里不是讨论的重点。

13.4.2 分析

一般来说，分析工作流的目的是分析并细化需求，并将这些需求构造成用作设计模型第一个剪辑的对象模型。这一阶段的成果是生成初始的分析模型。用该分析模型来精确定义用况和帮助生成候选构架（进一步建立构架基线是细化阶段的事情）。这就表示在初始阶段只完成分析模型的一小部分（如5%）。实际上，我们也许会把对初始阶段分析模型特点的刻画作为第一次粗略的剪辑，这仅仅是得到分析模型的构架视图的第一步工作而已。

构架分析（参见8.6.1节）

初始阶段的任务是：为了实现该阶段的目标而挑选出需要仔细考虑的用况或场景——主要是理解和细化这些用况或场景。给定了这些最初的用况或场景集合，构架设计师就可以为系统的某些部分构造出分析模型的第一个简单版本。在这里不需要非常广泛，也不需要非常完美。下一阶段并不期望简单地在此基础上建造项目。实际上，开发人员可能会抛弃它，只是希望用它来抛砖引玉。

分析一个用况（参见8.6.2节）

在某些情况下，逐个地考虑用况是不够的，用况模型在同一时间只能处理一个用况。事实上，用况在系统内部共享资源，如数据库、计算资源等。分析模型显示出这些共享资源。因此，通常必须进行非常充分的分析才可以解决这一矛盾。

在该迭代工作流中，我们可以分析并因而细化在13.4.1.1节中详细描述的部分用况（用况群的10%）。我们可以详细描述其中的一半，即约为用况群的5%。

分析一个类和分析一个包

如果这两项工作都要在本阶段中完成，则它们是以最小化的形式被完成的。

13.4.3 设计

本阶段设计工作流的主要目的是为包含在初步构架描述中的候选构架勾画出设计模型。

如果需要开发一个演示原型，可以通过使用预制好的模块、4GL语言或任何表明这一思想的快速开发技术来完成。演示不熟悉的用户界面和算法，以使所有项目相关人员相信，该项目值得进一步开发。向用户代表演示这一原型以确保满足用户的需要，并且如果需要的话，可以做一些修改以更好地满足他们的要求。

构架设计(参见9.4.1节)

设计工作流的目标是：开发初始的或第一个设计模型轮廓，完成设计模型构架视图的第一步工作。该视图通过子系统/类间的协作，来实现在较早前的需求分析工作流中识别出的用况。我们关心的是对子系统/类间的接口（最多定义它们的一些操作）的识别，即使这仅是设计的一个概要。由于这些接口是构架的核心，因此非常重要。此外，需要识别通用设计机制，用于实现所识别出的用况的子系统的基础层。选出在中间件层中使用的系统软件和框架 (参见9.5.1节)。即使设计模型非常粗略，也应该进行这一初步的设计，因为在本阶段我们将只进行到候选框架描述的程度。

设计模型不仅实现由指定的用况所表示的功能性需求，还实现非功能性需求（如性能），这可能存在风险。

如果要将目标系统实施到节点，构架设计师将设计实施模型的一个小规模的版本，例如，仅限于有挑战性的性能要求的节点和节点间的连接。这时，项目经理可以要求用况工程师针对存在挑战性的两个节点的部分功能以及它们之间的交互作用进行建模。

设计一个用况(参见9.4.2节)

在初始阶段中，设计用况的工作是最小化的。

设计一个类和设计一个子系统

同样，如果这两项工作都要在本阶段中完成，则它们是以最小化的形式被完成的。

13.4.4 实现

实现工作流的活动范围依赖于项目经理早期所作的决定。他或她会不会使初始阶段在候选构架描述这一点上停下来？

一方面，一些人认为只有参与者看到原型的运行，才能确信候选框架的功能。只有原型的各个部分确实开始运行，才能确信排除了风险。另一方面，为了将初始阶段的人力和时间减至最少，只要一有看上去可以工作的构架描述，就在初始阶段中停止工作流。而决定将候选构架开发到什么程度是项目经理的职责。

在他或她以正常的方式中止候选构架描述的情况下，接下来的实现工作流就不需要了。

然而，也有可能需要一个演示原型（一个抛弃型原型）。那样的话，尽管项目比较小，也需要随后的实现工作流。

13.4.5 测试

如图13-2所示，与分析、设计和实现工作流中的活动并行进行的是，测试工程师努力使自己了解目标系统的一般性质，考虑进行什么样的测试，并制定一些临时性的测试计划。然而由于探索性的演示原型主要是用作说明而不是供进一步使用，所以如图11-2和13-1所示，在初始阶段期间不进行大量的测试。但项目经理可能会发现进行少量的测试是十分有利的。

13.5 构造初始业务案例

在初始阶段后期，当发现项目有了一个候选构架并且能够缓解关键风险时，通过对项目的资源需求、投资金额以及市场（或内部）接受程度的综合考虑，就可以将这些构想转变成经济上的条款。业务案例的一方面是业务标书，另一方面是产品的最终使用所产生的经济效益。

13.5.1 对业务标书的勾画

对业务标书的估计公式依赖于最终产品的“规模”。然而，初始阶段末期的估计规模（只开发了用况群的一小部分）可能与最终的规模相差很大，如50%。此时，对费用的估计可能与项目最终的实际费用可能同样相差50%。

例如，将用况作为估计费用的度量标准。设计、实现和测试一个用况的“人时数”可能处于一百小时到几千小时之间。一个给定的用况处于上述范围的哪一点依赖于以下几个因素：

- 风格。如果开发者在单个用况中加入了更多的特征，则用况在功能上会比一般的用况强大，但它可能要耗费更多的人时。
- 目标系统的复杂度。系统越复杂，所花的费用可能就越多（给定规模）。要弄清楚是否可通过减少系统的功能来简化系统。
- 规模。一个小系统的用况通常比大系统的用况更容易理解。这种系统规模系数可能对每一用况的“人时数”费用有10倍左右的影响。
- 应用程序的类型。例如，对比客户机/服务器平台上的MIS应用，分布式环境中严格的实时系统和容错/高可用性的系统在每一用况的费用方面是MIS应用的用况费用的3~5倍。

以上所列并不详尽。开发机构和项目相关人员可根据他们自己的经验来缩减用于估计的变量。在缺乏经验的情况下，建议你根据惯例来作出估计。然而，由于学习新方法，使用新工具，采用统一过程的其他新特性，会增加费用和时间的估计值。在使用统一过程一两次以后，开发机构就会发现他们的费用大幅度下降，从开发时间到市场都有很大的改善，系统的质量和可靠性也都有所提高。

在初始阶段末期，项目组通常不会建立最终的业务案例。这是因为收集到的事实还不够用来作出最终的业务案例。首先，直到细化阶段的末期，统一过程才提出“确定的标书”。这就是在初始阶段称业务案例为初始业务案例的原因。这一初始业务案例仅需足以（实际上，它是相当粗糙的）证明进入细化阶段是可行的就可以了。例如，针对个人电脑市场目前的状态，不必进行初始阶段来查明不存在将另一常规字处理软件推向市场的业务案例。然而，在公司确实比

较在意项目的情况下，业务案例就始终是一个焦点，它通常要进行大量的调查研究工作。

人员需求以及早期迭代自身的进度安排都必须有资金上的支持。项目经理要在早期迭代中记录各种数据，用来作为估计未来项目所需的人力和开发进度的基础，但这一点通常都不受重视。因此，开发组应该对初始阶段进行合适的度量与跟踪。这些数据为估计下一个项目的初始阶段中迭代的数据奠定了基础，并通过判断下一个项目比作为基准的项目复杂还是简单来对这些数据进行修改。

13.5.2 对投资回报的估计

这种业务估计构成了业务案例的一个方面。而另一方面，并不存在用来计算软件所能带来的收益的简单公式。对于供市场出售的软件，卖出的软件数量、软件的销售价格以及软件可以持续销售的时间均是要进行考虑和判断的市场因素。如果软件是供内部使用的，则可要求受软件影响的部门去估计他们可能节约的费用量。通常，费用估计的偏差非常大，但这种估计至少提供了衡量收益和支出的基础。

对于商业性软件产品，业务案例应该包括对于项目的一系列的假设以及在这些假设都正确的情况下投资回报的数量等级。为了确保业务案例将会取得一定成效，管理部门通常设定一个关于投资回报的较大的障碍系数。

此时，可以大体上建立一个业务案例；也就是说，看上去进行这个项目是值得的或是可以盈利的，且项目会取得成功。就像在初始阶段所下的结论，我们所需要的不是确切的数字，而是对系统在软件开发组织和客户经济能力所及范围之内的认识。此时，你还没有用金融术语来表达业务案例所必需的详细信息。为了完成业务案例，你必须等到获得合理、稳定、精确的标书和进度安排。在细化阶段的末期，当你可以更精确地定义这一项目时，你可以再次检查这些假设。

13.6 评估初始阶段中的迭代

如13.2.4节所述，在初始阶段开始时，一旦有了足够的可用信息，项目经理便会制定评价准则来评价第一次迭代和整个阶段可否结束。由于初始阶段是慢慢结束的，因此项目经理也可派一个小组（可能只有两个人）来进行评价。评估小组通常包括客户或用户代表。对于规模较大的项目，可能还有必要包括所有受到影响的项目相关人员的代表。结果可能证明一些准则不是在原计划之内可以完成的。以下是准则的一些示例：

- 本阶段需要达到的用况模型范围的程度。
- 采用**探索性概念证明原型**来演示效果。
- 尚未发现所有关键风险的可疑性。
- 已列出的关键风险还没有被充分地缓解或者没有被突发事件处理计划充分覆盖的事实。

项目经理在执行下一次迭代之前采用这些评价准则，并且修改受影响的迭代计划和进度安排。例如，在随后的迭代中，可能需要额外的拥有某种技能或知识的开发人员加入开发组。

初始阶段评估的最重要的成果是决定继续开发还是取消开发。可以检查这一阶段的几个目标（系统范围、关键风险、候选构架），从而决定继续开发还是取消开发。在决定进行进一步开发之前，必须经过这第一个主要里程碑。一旦发现继续努力毫无意义时，可以取消开发。然而，决定不能被武断地作出。继续开发或中止开发都必须取得项目相关人员的同意，特别是要取得投资方和用户代表的认可。尤其是当有可能取消开发时，用户也许能够找到绕过障碍的方法。

13.7 制定细化阶段的计划

在初始阶段的末期，我们开始考虑细化阶段的费用和进度安排，因而开始制定细化阶段的计划。我们希望能够细化全部需求的80%；不希望遗漏任何对构架来讲比较重要的内容。这么做既是为了能够生成一个比初始阶段更精确的标书（初始阶段受有限数据的限制），也是为了选取构架。确定业务标书通常需要大约80%的用况群。在这80%之中，为了很好地理解需求，通常需要分析其中的50%。

为了获得构架基线，需要用到用况的80%以确保没有遗漏任何重要的东西。在这80%之中，选出整个用况群的重要部分，以此为基础来设计构架基线。这些重要的用况占整个用况群的百分比比已找出的80%要小，例如 40%的用况，而其中的平均只有20%的用况才是真正重要的。这两个百分比的乘积仅仅只占用况群的8%。换句话说，通常，不足整个用况群的10%的用况阐明了需要了解的全部重要用况。我们用这一部分来驱动构架基线上的工作，包括构架描述和所有模型的版本。

如果需要的迭代不止一次，就以这种方式启动细化阶段中所需要的迭代。可以假定只有一次迭代，但在复杂的情况下可能有多次迭代。要决定在每次迭代中做什么工作是恰当的，哪一个工作需要实现和测试工作流，以及因而需要缓解哪一种风险。

经验表明，在初始阶段完成的大部分设计和实现内容，如**探索性概念证明原型**，并不适合于作为下一个阶段建造工作的基础。

到现在为止，你想要对所有百分比进行跟踪可能有一些困难。因此，为了便于参考，表13-1中列出了所有的百分比。

表13-1 涉及用况的工作

	已完成的业务模型	已识别的用况	已描述的用况	已分析的用况	已设计、实现和测试的用况
初始阶段	50%~70%	50%	10%	5%	概念证明原型，占很小的百分比
细化阶段	几乎100%	80%或更多	40%~80%	20%~40%	少于10%
构造阶段	100%	100%	100%	如果保持就是100%	100%
移交阶段					

注：该表中的数据仅作为一个粗略的指示。我们在识别和讲一些有关用况的话之间作了区分，而且对它们作过更全面的阐述，见“描述一个用况的活动”（7.4.3节）。分析用况在“分析一个用况的活动”（8.4.2节）一节中作了阐述。最右边的一列指示在各阶段末期的基线中有多少用况。

13.8 初始阶段的可交付内容

初始阶段完成以下一些内容：

- 一个特征清单。
- 描述系统语境的业务（或领域）模型的第一个版本。
- 代表用况模型、分析模型、设计模型的第一个版本的首次剪辑。一些实现模型和测试模型的基本内容；补充需求的第一个版本。
- 包含用况、分析、设计和实现模型的视图轮廓的候选构架描述的第一个草案。
- 可能有一个**探索性概念证明原型**，用来演示新系统的用途。
- 一个初始的风险清单和一个用况优先级清单。
- 开始制定整个项目计划，包括为阶段制定的通用性计划。
- 业务案例的第一个草案，它包括：业务语境和成功准则（收益、市场认可以及项目估算）。

项目相关人员现在对项目的可行性以及概貌有了相当多的了解。用况之间的优先级顺序已经建立。项目经理已经有了足够的信息为下一阶段制定详细的计划。这一阶段获得的成果将在细化阶段进一步精化，我们将在第14章中作详细介绍。

第14章 细化阶段构造构架基线

当进入细化阶段时，也随之通过了一个主要里程碑，它标志着三项成就：

- 已经阐明了一个初始的构架（即候选构架），这意味着我们知道如何为目标系统建立一个构架，该构架包含了系统中新颖的以及困难的部分。
- 已经识别出了最严重的风险（即关键风险），并且通过对这些风险的研究，对建立系统的可行性充满信心。
- 已经建立了有足够描述信息的初始业务案例以便进入第二阶段，而且已经取得了项目相关人员尤其是投资方的同意。

14.1 细化阶段概述

细化阶段的目标是：

- 要捕获大部分尚未开发的需求，并以用况形式表示功能性需求。
- 要建立合理的构架基础（即构架基线），用来指导构造和移交阶段的工作，并进一步用来指导产品将来的升级。
- 要继续监控剩余的关键风险，并要识别出重大风险，以使我们可以估计出风险对业务案例尤其对费用的影响。
- 要向项目计划中加入更详细的描述。

为了实现这些目标，我们使用系统的一个“广而不深的”视图。在技术风险占主导地位或者是最主要的风险的情况下，我们可能需要更深入地探索如何建立合理的构架。对于大型项目，我们可能同时需要采用作为系统热点的“深而不广的”视图。系统构架设计师应尽早识别出系统中最危险的部分，并且启动原型/导航“探险程序”来识别和管理风险。

我们基于对整个系统的理解而做出构架上的决定：系统的范围、系统的功能性需求和非功能性需求（如性能）。此外，在做出这些决定时，必须保持需求（如用况和用况模型中所表达的）和构架之间的平衡。因为这两者的开发是相互关联且相互影响的（参见4.3节）。

细化阶段中最主要的焦点是阐明构架基线。这个焦点将对80%的用况进行细致分析并讨论影响这个目标的风险。本阶段开始借助开发环境的支持，这不仅是为了开展本阶段的工作，还为了替构造阶段做准备。到该阶段末期，我们已聚积起制定构造阶段计划所必需的信息。同样，到那时我们将拥有足够的信息建立可靠的业务案例，并使初始阶段所启动的工作运转起来。

14.2 细化阶段初期

细化阶段开始时，从初始阶段获得一个细化阶段的计划、一个部分细化了的用况模型以及

一个候选构架的描述。可能还会有初始的分析模型和设计模型。然而，尽管这些模型提供了一些指导，但不能指望重用这些模型。实际上，细化阶段的一个任务就是充实这些模型，虽然不是彻底的，但要达到获得构架基线的程度。

可能还有一个**概念证明原型**，用来演示系统的用途。然而，不能指望在该原型的基础上进行开发。建立该原型的目的通常是为了以最快的方式建立可行的系统，而不是为下一阶段提供开发基础。

14.2.1 制定细化阶段计划

从初始阶段末期传送过来的细化阶段计划可能是没有被彻底完成的。通常，直到细化阶段开始时才能完全知道细化阶段中可利用的资源。在有些情况下，在初始阶段结束和细化阶段开始之间有一段间隔。随着对资源、开发进度和可用人员的进一步了解，项目经理会修改原来的迭代计划和阶段计划。

14.2.2 组建开发组

并不需要对开发组在初始阶段做的所有工作都做记录。因此，项目经理将根据需要尽可能多地安排该组的成员进入细化阶段，他们将部分地承担“小组存储器”的职能。此外，在细化阶段，其他类型的人员也会加入到开发组。例如，需要具有可重用构造模块（可能适用于项目）工作经验的人员。细化阶段开发组的规模通常比初始阶段开发组的规模要大。新的人员加入到开发组，项目经理从这些人员中会选出一部分进入构造阶段，这部分人员有可能成为设计小组的领导。

14.2.3 改变开发环境

根据初始阶段的开发情况和细化阶段的期望，项目经理继续对开发环境进行调整，这一点曾在13.3.2节首次进行论述。

14.2.4 设立评价准则

迭代或细化阶段所要达到的特定准则对于不同的项目在总体上各不相同，但可以根据该阶段的目标来考虑。

14.2.4.1 扩展需求

评价准则是：

- 是否已经识别出设计构架基线、识别重大风险以及支持业务案例和标书所需要的需求、参与者和用况？
- 为了达到本阶段的目标，是否已经对这些目标进行了详细而充分的描述？

14.2.4.2 建立基线构架

评价准则是：

- 可执行的构架基线是否不仅满足了正式捕获的那些需求，而且还满足了项目相关人员观察工作基线后所体会到的需要？

- 构架基线看上去是否足够健壮，使之可以承受构造阶段以及后续版本可能需要的额外特征的考验？

14.2.4.3 缓解重大风险

评价准则是：

- 关键风险是否已经得到了适当的缓解（即关键风险已被消除或者已有缓冲风险的突发事件处理计划）？
- 是否已经识别出所有的重大风险？（参见12.2.2节和12.5节中的“重大风险”）
- 对重大风险的研究是否已经到达了可生成标书的程度？
- 是否可在构造阶段正常地排除仍保留在风险清单上的风险？

14.2.4.4 判断业务案例的价值

评价准则是：

- 是否对项目进行了充分的定义并确定了标书的费用、开发进度和质量？
- 业务案例是否能提供投资回报或达到业务所规定的投资障碍率？
- 简而言之，是否为已定价的合同（或内部开发的等价物）做好了准备？

14.3 原型的细化迭代工作流

如图14-1所示，原型迭代由五个工作流组成。在第二部分中曾描述过这些工作流，但本章只关心它们在细化阶段的作用。我们再次部分并行地开展四组活动。第一组是核心工作流；第二

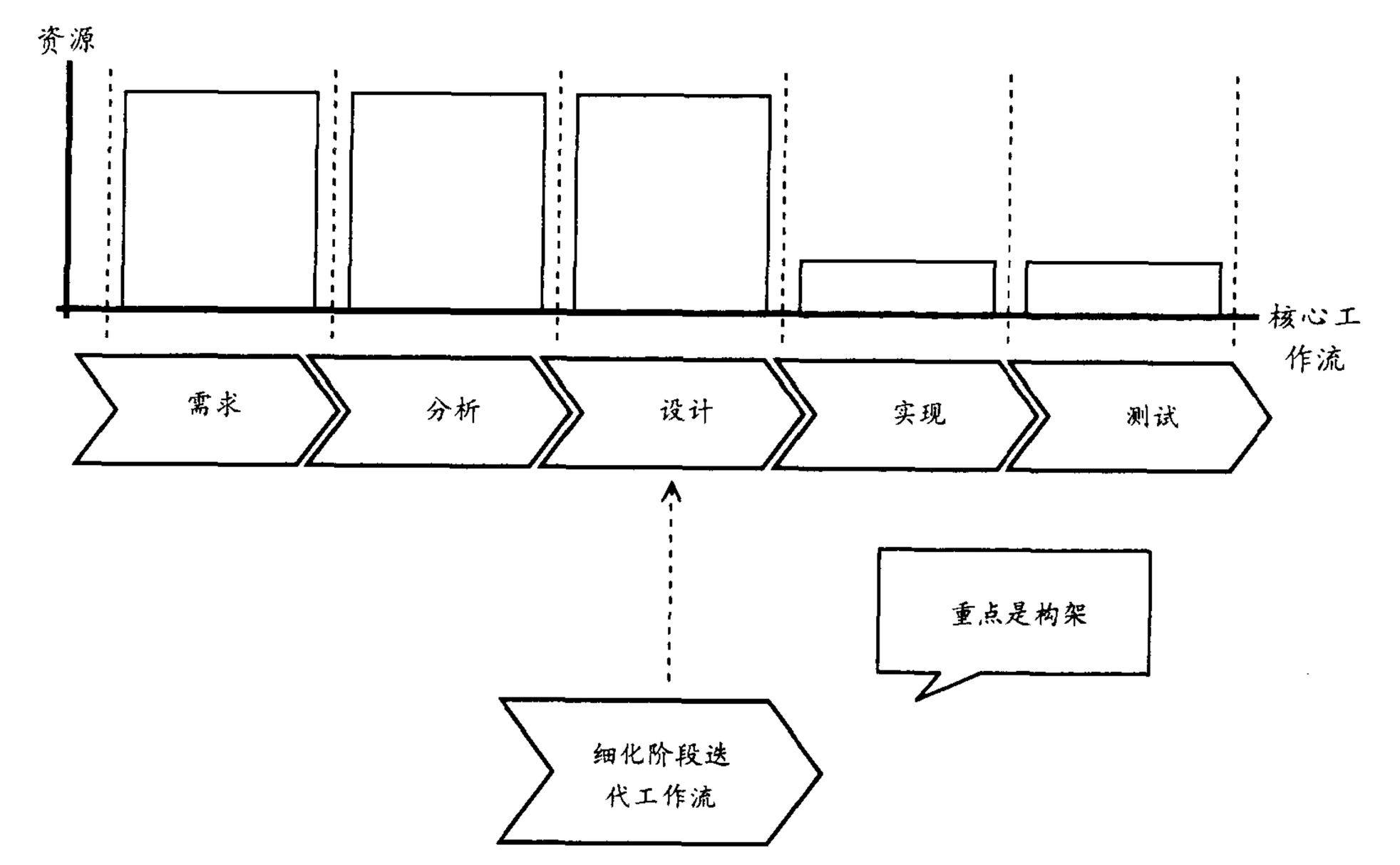


图14-1 细化阶段中一次迭代的工作按五个核心工作流进行。绝大部分工作在需求捕获、分析和设计期间完成；我们需要了解绝大部分需求并设计相应的系统。相比较而言，实现和测试仅需很少的资源

组是制定迭代计划，在12.4~12.7节和14.2.1节中介绍；第三组是评估，在12.8节和14.6节中介绍；第四组是进一步准备开发环境，最初曾在13.5节中做过描述。本节仅概览一下五个核心工作流以及它们在细化阶段所起的作用，在14.4节中再进行详细介绍。

在“构架设计”期间，仅仅捕捉、分析、设计、实现和测试与构架有关的需求，很少留意那些没有重大构架意义的细节部分，而是将这些细节留给构造阶段去处理。构架基线仅仅是一个骨架系统，对于它本身而言，除了要详细描述构架可以作为一个整体运行这一点之外，没有别的内容。根据系统的范围、风险、新颖度、使用技术的复杂程度以及开发人员的经验，用一次、两次或者在极端情况下用多次迭代开发出构架基线。

在本阶段，研究风险的目的不是要消除风险，而是要将风险降低到构造阶段可以接受的程度。换一种说法就是在细化阶段通过实现构架来解决在构架层面的技术风险。“构造阶段可以接受的程度”表示什么意思呢？所要表达的意思就是，对于风险，只要找到可以缓解风险的方法，并能够估算出缓解风险所需耗费的人力和时间就可以了。直到实现包含风险的用况时（有时是在细化阶段，通常是在构造阶段，偶而直到移交阶段），才真正消除了该风险。

14.3.1 捕获并细化绝大多数需求

在这里，捕获“绝大部分需求”是什么意思呢？在13.7节中，当开始制定细化阶段计划时，就对这个问题展开了讨论。在该节中曾提到目标是要识别约80%的用况。我们可能详细描述用况群的40%~80%。我们不需要识别所有的用况，也不需详细描述已识别出的所有用况，这是因为前面的经验告诉我们，可以很容易地（在构架方面上）设计出某些子系统，不包含任何意外的风险，并且可以准确地报价。请参见第6章和第7章。

在详细描述的用况群中，我们大约选出其中的一半通过分析进行仔细的检查。我们可能发现，为了进行设计、实现和测试，为了获取构架和缓解风险，需要考虑的仅是这一半用况中的部分场景，见表13-1。目的是要捕获达到本阶段目标所必要的需求。

14.3.2 开发构架基线

如第8章、第9章和第10章中所介绍的，构架设计师对用况进行优先级排序，并进行构架层次上的分析、设计和实现工作。而另一些工作人员则进行分析和设计工作，即分析类和包（参见第8章），并设计类和设计子系统（参见第9章）。

测试工程师则集中精力建立测试环境，对构件进行测试，并对有构架意义的用况实现的整个基线进行测试。

14.3.3 开发组较小时的迭代

在细化阶段中，开发组的规模仍然较小，但需要进行迭代并尝试不同的解决方案（技术、框架、结构等）。如果项目具有挑战性，可能在获得一个稳定的构架之前需要3~4次迭代。随后，在构造阶段，当已有了许多人员以及成千上万行代码时，需要基于稳定的构架开展工作，并增量地开发系统。

如果系统比较小或比较简单，一次迭代就足够了。但是如果系统庞大而复杂，一次迭代仅仅可能是第一步而已。额外的迭代需要考虑诸如系统的复杂性、需要起指导作用的构架基线以及风险的严重性等因素。

迭代会不停地进行，直到构架达到一个稳定的水平，即该构架可以说明系统是可接受的，并且达到很少发生变化的程度。

14.4 执行五个核心工作流——从捕获需求到测试

细化阶段的工作是在初始阶段工作的基础上进行的。在初始阶段，只需证明可以在以后的阶段中建立构架。而在细化阶段，将真正地开发构架。我们将再次回到以前做过的工作，但可能会发现没有什么可重用的内容。现在，不仅要找出表示关键风险的用况，还要找出对构架有重要意义的用况。其次，为了提供准确的标书，要涉及大部分的用况。此外，我们所面临的任务是要在细化阶段末期建立一个可执行的构架基线，一个可在构造阶段添加内容的稳定的基线。因此，我们要比初始阶段更注重质量和可扩展性。

在本次迭代开始时，要仔细检查风险并识别出用况（如第12章所述）。为了找出对构架有重要意义的需求并收集生成标书所需要的足够多的信息，需要涉及大约80%的需求。通常需要在这些用况中找出与开发构架基线相关的10%的用况。

在下面的假想项目中，我们假定一个可在一次迭代中完成构架基线的中等复杂的系统，并且假定是非先验性开发。我们已经指出，项目经理已经有了从初始阶段传递过来的制定项目计划的素材和具有一定内容的首次迭代的迭代计划。下一步工作是与构架设计师和高级开发人员合作，向迭代计划中加入细节内容。

我们按照五个工作流对这些内容进行讨论。这些工作流的顺序可能会使你认为要依次执行这些工作流，但这些工作流内的工作可能是同时进行的，如图14-2所示。

14.4.1 捕获需求

在本书中，我们会寻找用况，对用况进行优先级排序，详细描述并且构造用况（对需求的详细讨论，参见第6章和第7章）。

14.4.1.1 确定用况和参与者

系统分析人员识别出在初始阶段没有发现的用况和参与者（参见7.4.1节）。为了达到本阶段的目标，我们需要了解80%的用况，但并不需要详细描述那么多用况。我们可能识别出了几乎全部的用况（80%），但只需要详细描述其中的一部分，且只分析经过详细描述的用况中的一部分。这里所说的“了解”是指了解什么是对构架有重要意义的，并且确定没有遗漏任何可能影响构架和标书的用况。

要捕获多少用况也依赖于需要达到的准确程度。例如，如果想要一个确定的标书，可能需要详细描述更多的用况，需要详细描述的用况可能达到全部用况的80%。对于一些复杂的系统，可能要识别出几乎所有的用况，并且详细描述其中的80%以上的用况。如果是我们自己为项目提

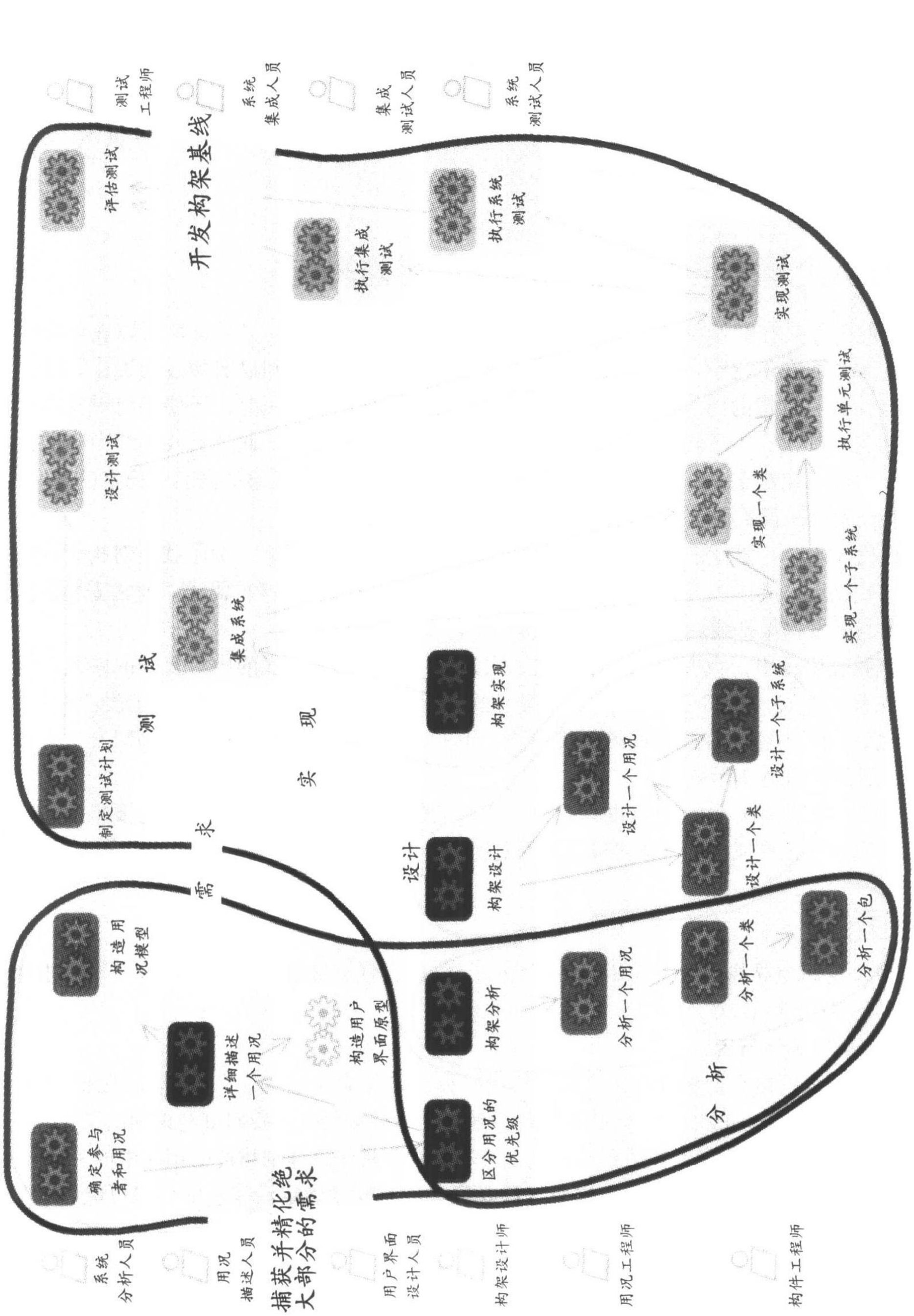


图14-2　手画粗边线所圈定的部分中集中了细化阶段的主要活动

供经费，可能只停留在一个较低的百分比上。停留在一个较低的百分比当然会增加风险。在细化阶段投入较少的时间和人力，在管理上则将面对更大的风险。为了一点眼前的利益而要面对潜在的风险是十分不明智的。

14.4.1.2 构造用户界面原型

在需求捕获期间发生的另一个活动是确定用户界面（参见7.4.4节）。

在细化阶段期间，我们要注意那些对构架有意义的接口。然而，很少出现用户界面唯一的情况。如果确实如此，必须建立我们自己的某种用户界面框架。例如，当所开发的系统本身就是一个用户界面框架时就是如此。又如，具有唯一通信协议的系统，其协议对构架性能和响应时间都有重要意义。

即使从构架观点来看用户界面没有什么重要意义，实现一个用户界面还有一个理由，即可以从真正的用户那里了解系统是否合适。然而，只有在初始阶段不能证明系统的价值时，才应该原型化用户界面。一般来讲，没有必要在细化阶段原型化用户界面。

14.4.1.3 区分用况的优先级

在初始阶段准备好的局部用况模型的基础上，我们建议按照两条线进行开发：添加更多的用况和按构架基线开展工作（参见7.4.2节）。首先，要花一定的时间来寻找更多的用况，接着把注意力转移到构架上。然而，我们还必须协调这两条线。我们的决定受风险优先级和开发顺序的影响（参见第7章7.4.2节“区分用况的优先级”和第12章12.6节“用况优先级排序”）。构架设计师根据用况模型生成用况模型视图，该视图是构架描述的一部分。

14.4.1.4 详细描述一个用况

用况描述人员要详细描述那些彻底理解需求所必需的用况以及建立构架基线所必需的用况。有关内容请参见第7章7.4.3节“详细描述一个用况”。在细化阶段，我们将把精力主要集中在对构架有重要意义的复杂用况的初步描述上。对于选出的将进行详细描述的用况，通常也不是详细描述其所有部分，而是描述在本阶段需要用到的一些场景。尽量不描述没有必要描述的内容。然而，正如先前所提到的，在一些复杂的情况下，有必要描述全部场景和用况，即几乎100%的用况群。

14.4.1.5 构造用况模型

系统分析人员审查自己所做的工作，并寻找其中的相似点、可简化点及改善用况模型结构的机会。分析人员使用诸如扩展和泛化（参见7.4.5节）等机制，建立一个结构良好且易于理解的模型。例如，减少重复部分，从而使该模型易于修改、扩展和维护。然而，有时分析人员在此时还没有发现一个好的结构，可能要在以后的迭代中做了进一步的工作之后再做决定，有时一直要等到用况进入分析和设计工作流时再做决定。

举例：构造用况模型

开发人员在开发用况模型时，他们发现几个用况的实现有相似之处。例如，“订购商品或服务”、“确认订单”、“给买主账单”、“赠送纪念品”等用况，这些用况均涉及在买主和卖主之间递送交易对象。这个普通行为的可重用用况是“提交交易对象”，该用况可通过重新构造用况模型而引入。当开发人员继续开发这些用况时，他们均重用了“提交交易对象”的实现。在这里，交易对象包括账单、订单、订单确认，它们以相似的方式改变状态，支持相似的操

作，在买主和卖主之间传递。由于存在这些相似点，从而可以基于一个抽象类即交易对象开发这些用况。

14.4.2 分析

我们在初始阶段提出了分析模型的一个草案（参见第8章中关于分析的详细讨论）。现在，我们将在此基础之上进行开发，但我们会发现不得不废弃其中相当大的一部分。在细化阶段，我们需要开发的是对构架有重要意义的用况，以及为便于更好地理解标书的基本内容而需要加以细化的复杂用况。

本节中的工作包括：构架分析、用况分析、类分析和包分析。我们需要很好地分析对构架有重要意义的用况。这些用况占用况群的比例通常少于10%。我们还要通过分析这些用况，从而更准确地了解它们，洞悉它们之间的相互作用。总的来说，需要考虑详细描述的用况群的50%。

调整需求以适应构架

当我们捕获进一步的需求信息（完成额外的用况）时，我们利用构架开发的外延性知识，就能更富技巧地完成这项工作（参见4.3节）。当我们对每一个新提议的需求或用况进行价值和费用的评价时，我们的依据仍是已经拥有的构架基线。构架告诉我们，有些需求很容易实现，而有些则不然。

当研究一项新需求将造成的后果后，我们也许会发现，对需求做些改变（没有语义变化或仅仅只有很少语义变化的改变）将大大简化实现的难度。产生这种结果的原因是在需求中进行这样的改动会使得设计与现存构架更加和谐。我们要与客户就需求的这种改动进行磋商。

当分析、设计、实现和测试该系统时，需要保持设计中的任何改变同已有的设计模型中的构架的一致性。进行这样的调整意味着必须考虑已经存在的子系统、构件、接口、用况实现以及主动类等。这样就可以在一个已有的设计上有效地进行一次新的设计。

举例：重新协商需求

假设我们是一个向私营客户出售一种名为PortfolioPlus软件包的公司，该软件包能够分析股市走势。在三年前，用户对股市价格手工操作的改变非常满意。而随着对万维网研究的深入，客户的要求也越来越高。他们可以毫不费力地得到报价，非常轻松而且几乎是实时获得的。为了竞争，我们要使PortfolioPlus能够同步地获得股市报价。

由于PortfolioPlus的构架，对其进行改变以使其能直接收听报价器将意味着我们会有很多的工作要做。一种较便宜的方法是使用我们早期开发的API函数从Excel电子表格中获得输入。一个简单而有效的办法是执行PortfolioPlus的一条宏指令，即既要下载一个Excel电子表格，还要从我们的Web站点上获得一个新版本，其上有用户所需要的报价。我们的工作可以归纳为三个任务：

- 当PortfolioPlus客户需要时在我们的Web站点上生成电子表格。
- 使得Web站点适应预期数量的PortfolioPlus用户。
- 写出取得电子表格的宏指令。

举例：现实生活

这个示例说明了真正意义上的重用，展现了协商可能对项目产生的影响。

通过使用构架与客户进行需求协商，公司就能以低成本、高质量来构造系统。一个典型的有某电信公司参与其中的例子阐明了这种改进。

客户已经以类似于用况的形式准备了一份详尽的需求清单。这份清单估计出开发系统所需要的费用，它发现照常规而言，构造这个系统需要花费25人/年。软件供应者向电信公司展示：通过修改需求从而使其与已存在的构架相一致，可能会得到相类似的系统，尽管不是完全相同的系统。按已存在的构架仿造，有可能节约90%的开发费用。

电信公司决定在该构架基础上进行开发。它获得了标准产品的系统，只是根据特定的需求做稍微的修改而已。这将节省20人/年以上的投入。另外，还不必支付软件和硬件维护所需的运行费用，相反只需花费与维护标准产品差不多的维护费用。

客户最初所需的与最后同意购买的之间的差异就在于卖主如何构造用况的结果不同。对用户界面的轻微改动、管理主进程的不同方式以及测量和表达通信流量的不同方法等节省了90%的费用。另外，客户获得了比要求更多的功能。事实上，其价格很低，这是因为从前的用户已经为此支付了大量的费用。软件销售商将这些额外的功能保持低价，这是因为已经实现并测试过这些功能。

14.4.2.1 构架分析

初始阶段所进行的构架分析只达到确认构架可行性的程度（参见8.6.1节），通常并不是十分深入的。在细化阶段，必须加深对构架的分析从而可以支持比较完整的构架，即可执行的构架基线。

为了达到这一目的，构架设计师对系统进行初步（高层）的分割而成为分析包，他是根据用况模型的构架视图、相关的需求、术语以及由业务模型（或简化的领域模型）的形式所表达的可用的领域知识来开展工作的。他可以使用分层的构架，识别出专用应用包和通用应用包；从问题的观点来看，这些都是最重要的分析包。一旦在用况模型的构架视图中看到“驱动器”用况，构架设计师就可以确认出显而易见的并对构架有重要意义的服务包和分析类。

构架设计师在收集用况需求以考虑构架时，他在寻找支持用况实现所需要的基本机制。他识别出所需的通用分析机制（参见8.6.1.3节），这些机制包括通用协作关系（参见第3章）和通用包。通用协作关系包括诸如出错恢复以及事务处理等特征。通用包是指诸如持久性、图形用户界面以及对象分布等特征。

现在到了构架设计师改进分析模型视图的时候了。

14.4.2.2 分析用况

分析模型中所描述的许多用况并没有被很清楚地理解（参见8.6.2节）。用况必须根据存在于需求世界但不必直接实现的分析类进行精化。对于那些复杂的用况以及那些对别的用况有影响的用况（也就是说，别的用况赖以存在的用况）而言，尤其需要这种精化。例如，对于可以接收信息的用况，必须有其他用况提供该信息。

因此，那些对构架重要的用况以及需要重点了解的用况，可按分析类的形式加以精化。而对那些对构架不重要的用况以及对理解需求意义不大的用况，则不进行分析或精化。对于这些用况，用况工程师仅仅需要了解它们是什么以及确定它们确实不存在别的什么影响。在构造阶段实现这些用况时，他们将会知道如何处理这些用况。

对那些比较重要的或复杂的用况，也不需要进行特别详细的描述，只要能让系统分析人员了解用况所勾画的内容（即构架基线和业务案例）就行了。如果为了了解用况在系统中的作用而需要考虑80%的用况群并详细描述不超过40%的用况群，那么在分析中需要考虑的用况通常会更少，这是因为其中一些用况对业务案例几乎没有什么影响（参见表13-1中给出的百分比值）。

然后用况工程师开始寻找实现这些用况的分析类。他们要用到构架设计师所识别出的具有重要构架意义的分析类，给这些分析类赋予一定的职责。分析一个用况的大量工作是要遍历用况模型中的每一个用况，并根据类以及类的职责对该用况进行更详细的描述。用况工程师同时也要说明类之间的联系以及这些类的属性。

举例：类的职责

当开发人员在第一次迭代中处理“支付账单”用况时，他们提出一个调度和实现支付的类“支付调度程序”，该类有如下一些职责：

- 创建一个支付请求。
- 在到期时启动资金的转移。

在随后的迭代中，开发人员会发现更多的职责。添加这些职责并不需要重新构造类。在一个好的分析模型中，应该可以添加一些新的职责，而并不需要摒弃一些已经存在的职责，更不需要在随后的迭代中重新构造那些已经发现的类。随后当开发人员拓展这些类的工作范围时，例如在细化阶段的第二次迭代中，开发人员可能会发现某个类有更多的职责，这个类应该能够在不重新进行构造的情况下容纳这些职责。这些职责有：

- 跟踪那些已经列入计划的支付项。
- 当做过安排进行支付并已支付（关闭状态）时，开出一张账单。

在这个用况分析工作的基础上，构架设计师选出那些对构架有重要意义的类。这些类成为分析模型构架视图的基础。

14.4.2.3 分析一个类

构件工程师对前一阶段识别出来的类进行精化，他们对根据不同用况而分配给类的各种职责进行合并，确认出可用的分析机制并且找出每一个类是如何运用这些机制的（参见8.6.3节）。

14.4.2.4 分析一个包

正如在上面的构架分析中所提到的，构架设计师要考虑系统的服务并将类分组成服务包，这是在构架分析活动中完成的。通过将这些类分组成服务包，构件工程师承担起这些包的职责，并且对其加以精化和维护（参见8.6.4节）。

14.4.3 设计

在本阶段，我们通常设计和实现不超过10%的用况群。这个比例只是在本阶段识别出的用况

群的一小部分。在细化阶段中，我们是在构架这一层次进行设计的，这就表明只是设计对构架有重要意义的用况、类和子系统。分析阶段的包以及设计阶段的子系统对于定义构架视图都是至关重要的。尽管各种类元对构架可能有或可能没有重要意义，但包和子系统通常对构架都是有重要意义的（参见第9章）。

14.4.3.1 构架设计

正如在设计模型的构架视图中所描述的（参见9.3.6节），构架设计师要负责对系统的构架有重要意义的各个方面的设计。设计模型的构架视图包括子系统、类、接口以及包含在用况模型视图中对构架有重要意义的用况的实现。设计的其他方面的工作就落在用况工程师和构件工程师身上。

构架设计师识别分层的构架（包括通用的设计机制）、子系统及其接口、对构架有重要意义的设计类以及节点配置。这些分别在以下四个小节中讨论：

• 识别分层的构架

构架设计师继续进行在初始阶段启动的工作，并设计分层的构架。他再次回到系统软件层和中间件层（参见9.5.1.2.2节），选出要用的那些最终产品。他可以将其所在公司过去已经开发出来的产品重用，并指明与这些部分的接口是什么。构架设计师在较低层次选择产品，作为与早先步骤（参见14.4.2.1节）中发现的分析机制相对应的设计机制的实现。这些“设计机制”是指支持目标系统运行所在的操作系统、程序设计语言、数据库系统、对象请求代理等所包含的各种机制。实现环境界定了产品所使用的设计机制。这些设计机制或者是通过购买产品实现或者是自行建造，通常是中间件的子系统以及分层构架中的系统软件，可以与分析工作流并行建造或选择。构件工程师将根据这些设计机制进行设计。

举例：用于对象分布的Java RMI

Java RMI 用于对象分布，也就是说，java.rmi包将在细化阶段中用于实现“提交交易对象”用况。

• 识别子系统及其接口

构架设计师接着继续进行构架较高层次（接近应用层）的开发。基于分析模型中的包，构架设计师识别出将要包含到设计模型中的子系统。通常，构架设计师试图使分析模型中的每一个服务包成为设计中的一个服务子系统；处于较高层次的分析包变成设计模型中的设计子系统。

该方法在某些情况下运作很好，而在另一些情况下，分析和设计之间并不是如此简单的对应关系。例如，在某些情况下，一个分析包并不映射为一个设计子系统，而映射为一个遗留系统（或其中的一部分）。这种映射并不是简单的一对一的映射。遗留系统可能实现几个分析包或是其中的一部分，或者一个分析包可能映射为几个不同的遗留系统，也就是说，是多对多的关系。

在另一些情况下，构架设计师可能选出一些可重用的构造块（如框架），这些构造块或者是内部开发的，或者是为外部销售商提供的。这些块可能与分析模型所建议的包结构相匹配。因此，构架设计师可能不得不为构架设计选择子系统结构，该结构可能与构架分析工作中选出的结构稍有不同。

• 识别对构架有重要意义的设计类

构架设计师把对构架有重要意义的分析类转变成设计类。随着所找出的设计类的数量的增多，他挑出那些对构架有重要意义的设计类，如主动类，并在构架说明中对它们进行描述。

• 如果是分布式系统，识别出节点及其网络配置

构架设计师通过研究所需要的线程和进程以及处理机和其他设备的物理网络，考虑系统所需要的并发性和分布性。那些已经设计的用况，尤其是那些交互图中所示的用况，是构架设计师在进行这项工作时十分重要的输入。构架设计师将交互图中所使用的对象分配给主动类，而这些主动类又被依次分配给处理器和其他设备。该步骤既进行逻辑意义上的功能分布，也进行物理意义上的功能分布。

构架设计师要准备设计模型构架视图的一个新版本以及实施模型构架视图的一个新版本，这两者均包含在构架说明中。

14.4.3.2 设计一个用况

对构架有重要意义的用况按照设计子系统、服务子系统或设计类进行设计（参见9.5.2节）（其他被识别、细化并分析过的用况不在本阶段进行设计）。这项活动与我们在分析阶段所做的活动（分析用况的活动）类似，但也存在一些区别。在分析中，我们希望分析和精化用况，以获得一个健壮的、对将来的改变具有弹性并且可以重用的规格说明，这一规格说明将作为设计的第一个方案。我们将寻找已被识别的分析类的职责。

在设计中，我们所做的工作要具体得多。从分析过渡到实现的过程中，构件工程师必须使分析模型适用于可运行的设计模型，这是因为随后的工作要受到设计机制的限制。然而，分析包和类确实提供寻找子系统和设计类的直接方法。一旦找到这些子系统和设计类，就不仅需要描述设计元素所有的职责，还要详细描述它们之间的交互作用。

在分析中，我们描述了执行一个用况时焦点是怎样从一个元素转移到另一个元素的，并使用了几种不同的交互图来表现这一转移。在设计中，我们也要详细描述用于通信的操作。在设计中，我们还需要考虑重用那些可重用的子系统、框架或遗留系统，并说明它们提供哪些操作。如果这些内容难以理解，那么设计也将难以理解。

该活动的结果是一组“**用况实现——设计**”，其中每一个制品对应一个对构架有重要意义的用况。

14.4.3.3 设计一个类

在前面的步骤中，我们设计了参与用况实现的类。要注意这些类通常没有彻底完成，它们将在以后的迭代中参与更多的用况实现。构件工程师将每个类的不同角色合并成一个一致的类，这一内容曾经在10.3节中讨论过。

14.4.3.4 设计一个子系统

构件工程师对来自构架设计的子系统进行设计。在这些步骤期间，必要时构架设计师会更新设计模型的构架视图。

14.4.4 实现

该工作流从考察对构架有重要意义的设计元素开始，实现和测试对构架有重要意义的构件，

其结果就是构架基线。构架基线常常是由少于10%的用况群实现的。在本节将讨论构架实现的各项活动，即实现一个类、实现一个子系统以及集成系统（参见第10章）。

14.4.4.1 构架实现

基于设计模型的构架视图以及实施模型的构架视图，我们识别出实现服务子系统所需的构件。该可执行构件与其运行时所在网络中的节点相匹配。构架设计师接着在实现模型的构架视图中对此加以说明（参见10.5.1节）。

14.4.4.2 实现一个类和实现一个子系统

在设计工作流中，构件工程师设计出一组与创建构架基线相关的类。该基线将是我们打算建造的系统的一个早期的可执行版本。在本工作流中，构件工程师用文件构件实现这些类（实现来自设计模型的一个服务子系统，通常需要一个或多个构件）。“执行单元测试”活动确保每一构件可作为一个单元来工作（参见10.5.3节和10.5.4节）。

14.4.4.3 集成系统

基于将在该迭代中实现的一小部分用况，系统集成人员在集成构造块计划中制定出一个集成顺序，接着将子系统及其相应构件增量地集成为可执行的构架基线（参见10.5.2节）。

举例：三个构造块

系统集成人员提出了三个初始构造块，如图14-3所示。

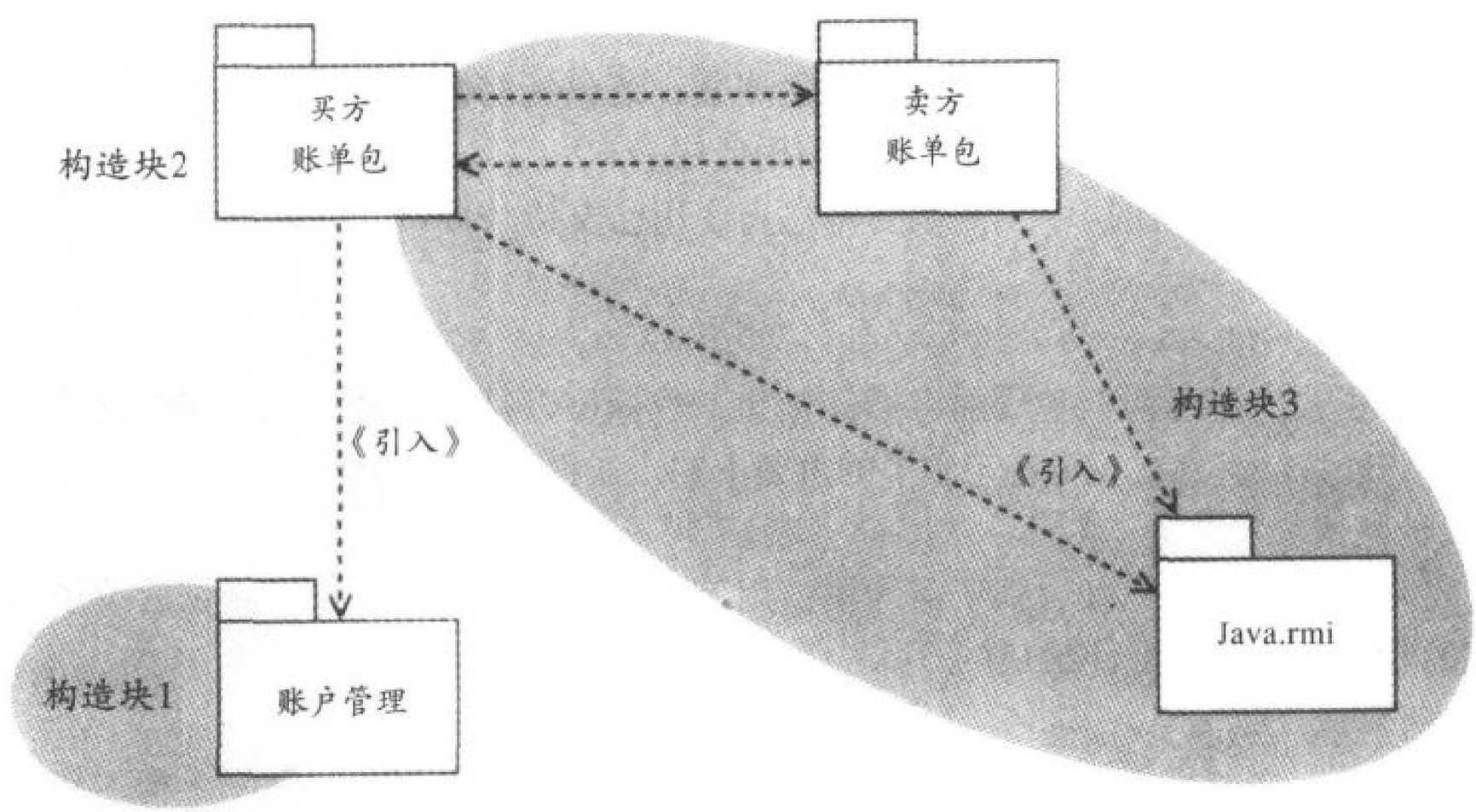


图14-3 首次迭代包含三个构造块。请注意，构造块2中也包括了构造块1的结果

图14-3 的示例包含三个构造块，其中的每一个构造块都经过了集成测试：

1) “账户管理”子系统（封装了旧的银行系统）的类。
2) “买主账单包”（包含在“支付账单”用况中）的类，加上第一个构造块。

3) “卖主账单包”的类和“买主账单包”（包含在“给买主发送账单”用况中）的类。这些子系统刚开始时是在“提交交易对象”抽象用况的通用类中，后来，这些通用类被转移到在其他几个子系统中可重用的一个子系统中。这个构造必须与Java RMI包进行集成。

在初始阶段没有工具仍能够设法开展工作（虽然有些痛苦），而在细化阶段不用工具显然是不切实际的。例如，现在开始有了要管理的版本，就需要配置管理工具。要在已有的基础上开展工作。在进入细化阶段之前，人们可以用笔和纸进行工作，然而到了细化阶段，如果不把构架基线置于工具的控制之下，人们将无法完成开发工作。

再举一个例子，工作人员会使用“生产”语言，如Java及工具集，其好处在于可以试验开发环境，还可以使工作人员熟悉新工具和新方法。在这种情况下，第一个增量很可能利用最终系统的基层（即中间件和系统软件）来构造；而原型很可能进化为真正的系统。

14.4.5 测试

测试的重点是要确定在所有水平（既有服务子系统也有设计子系统）和所有层次上（从系统软件层到专用应用层）的子系统及其接口是可运行的，参见第11章。当然，我们仅能测试可执行的构件。如果这些构件可以运行，则基本上可以相信别的东西（在其他模型中）也是可用的。

从较低的构架层开始测试，这表示要测试对象的分布、储存、检索（持久性）和并发以及系统较低层的其他机制。测试不仅包括对功能的测试，还包括对性能的测试。大多数层次本质上并不需要进行测试，需要测试的是高层如何利用低层。

在专用应用层和通用应用层中，测试要考虑当使用已支持的接口合并新的子系统时，如何有效地扩大系统的规模。

14.4.5.1 制定测试计划

测试工程师要选择评价构架基线的目标。例如，目标可以是能否在一个给定的响应时间内执行一个用况场景，一直到在一个特定水平的负载下的工作情况（参见11.5.1节）。

14.4.5.2 设计测试

基于这些目标，测试工程师识别所需的测试用例并准备测试规程，以便测试后续子系统的集成和后期的整个基线（参见11.5.2节）。

14.4.5.3 执行集成测试

当测试构件后，这些构件就可以用于集成测试。集成测试人员要测试每一个构造块（参见11.5.4节）。

14.4.5.4 执行系统测试

当系统（由对构架有重要意义的用况定义）被集成起来之后，系统测试人员就开始对系统进行测试。这个系统（最终系统的一个版本）就是构架基线。系统测试人员要将缺陷反馈给构件工程师或构架设计师进行修正（参见11.5.5节）。

测试工程师重新审视系统测试结果从而验证他们是否完成了最初的目标，或者决定为了达到那些目标，测试用例要如何进行必要的修改。

举例：不一致的转账造成了主要风险

系统测试发现大部分功能似乎能够达到期望的质量目标，但有一个例外，即当测试人员执行“支付账单”用况时，有些结果不正确。“账户管理”子系统中由遗留系统包装而成的功能对一些非整数的金额的转账不会产生预期的结果，如134.65美元，而其他数额的转账运行正确，如124.54美元。测试人员提出要对该问题引起注意，构架设计师将这一问题标为主要风险。项目经理任命一个小组着手解决这一问题。

14.5 产生业务案例

缓解风险和构建构架基线的根本理由是让项目进行到开发组可进入构造阶段的程度。此时，开发组有充分的信心能在业务范围内构造产品。本质上，有两种业务限制，其一是对一个给定质量下的进度、人力和费用的估计，其二是投资回报（或一些相当的指标）表明目标系统在经济上将是成功的。

初始阶段结束时，软件组织只是在一个非常广泛的范围内对业务案例的价值做出判断，至少新颖、大型或难度较大的项目是如此。到细化阶段末期，软件组织增进了对项目的了解，限制已非常严格，从而能在更苛刻的业务实践限制下准备标书和生成业务案例。

14.5.1 准备业务标书

开发组期望在类似于业务关系的基础上开展构造阶段的工作，而不论业务安排是与外来客户签订的协议，或是与本公司内另一部门的关系，又或是要卖给大量客户的产品的开发。

我们注意到，软件估计是以项目大小和软件组织的生产能力为基础的。软件组织的生产力来源于该组织在以前开发项目所积累的经验，项目大小则由细化阶段所获得的内容确定。为做到真实的估计，要保证在细化阶段能达到很好地了解构造阶段工作内容的程度。为此，首先需要构架基线；其次，如果项目存在大量风险，就要对风险进行研究，直到能估计出克服它们需要多少时间和人力。

14.5.2 更新投资回报

业务案例从本质上讲就是：软件产品从使用或者销售中所获得的收益是否能抵消开发所投入的费用？产品能否及时地投入市场（或内部使用）从而获得利润？

软件组织现在能以业务标书的形式估计构造和移交阶段的费用，标书体现了业务案例的一方面情况；但是另一方面，并不存在计算软件将能获得多少收益的简洁公式。

对上市软件而言，售价、销售量、销售期都是值得项目经理考虑和判断的问题。对内部使用的软件而言，应要求使用该软件的部门估计所期望的节省效果。估计潜在收益的误差通常很大，但是这种估计至少可以帮助权衡得失。

14.6 评估细化阶段的迭代

每次迭代结束时，都应根据这次迭代开始前所制定的迭代计划进行评估，如在14.2.5 节所述。评估小组评审每次迭代的结果，验证基线是否真正代表构架起到了完成原定目标和缓解风险的作用。

如果有多次迭代，那么第一次迭代的结果就是在构架上的第一次剪辑。该剪辑可能相当好地描述了不同模型（用况、分析、设计、实施和实现）的构架视图。第二次迭代的结果是在构架上的第二次剪辑，最后一次迭代则提供构架基线。每次迭代完成后，项目经理通常要与评估小组一起按事先确定的准则进行评价。如果项目没有满足这些准则，项目经理则要重新调整有关工作，如第12章所述。在细化阶段中，这意味着将未完成的活动带入下一次迭代。

因为项目经理让客户和其他项目相关人员认可每个最小里程碑的成果，因而通常在到达主要里程碑时（阶段结束时）他们很少提出异议。相对瀑布模型而言，迭代增量式过程使项目组与客户具有更紧密的关系，客户也就有机会将改进意见及时反馈到开发模型中。

在细化阶段结束时，评估小组将使项目相关人员相信，细化阶段已经缓解了重大风险，已经建立了一个稳定的构架基线，系统能够依据项目计划和构造阶段的标书进行构建。

14.7 制定构造阶段计划

细化阶段快结束时，项目经理开始详细计划构造阶段的第一次迭代，并用非常概括性的术语记下剩余的部分工作。迭代次数依赖于项目大小和复杂程度，项目经理通常制定2~3次迭代，当任务庞大而复杂时，则制定4次或更多次迭代。在每次迭代中，他们勾画出大量的构造块，每次都是在已完成的工程上增加少量任务而得到新的构造块。

项目经理仍在风险清单上记下许多风险。在构造和迭代序列进行之前，项目经理对剩余风险的处理顺序做出安排，争取在构造或迭代顺序中风险突发之前对风险加以缓解。实施原则仍是在这些风险扰乱构造顺序之前缓解它们。

细化阶段仅仅完成每一模型的一部分。为了完成对模型的补充，项目经理要考虑应以怎样的次序处理剩余的用况和场景。这种考虑导致了构造和迭代顺序的产生。在大型项目中，为了增加开发人员来缩短整个项目的进度，项目经理对工作加以分类，以使员工能并行工作。开发大型的工业化系统意味着项目组必须寻求并行的工作途径，因为项目有时间限制。项目组要寻求部署大量人员共同工作的途径，通常以二十的倍数增加人员。

这种方法以构架基线中的子系统为基础，在设计工作流（由分析包所激发）的不同层次上发现子系统。子系统有接口，一个较高层次的目标就是识别并定义这些接口。接口是整个构架的核心。随着对子系统和接口的识别与详细描述，我们已为并行工作创造了很好的条件。

由单独的人员负责一个设计子系统中的一个服务子系统，由一个组负责一个设计子系统。

如果个人或小组想要在工作过程中保持适当程度的独立性，那么用于限定他们的活动范围

的接口必须是固定的。为了强调其重要性，一般称接口的规格说明为协议，一份协议把当前的开发人员和后续各周期的开发人员通过接口联系起来。此后，只要不违反协议，开发人员可以用另外一个子系统来替代当前的子系统。构建通过接口协议（或相当物）互相连接的子系统的原理同构建系统的系统是一样的，在7.2.3 节中已经介绍过后面这个问题。

14.8 关键的可交付内容

可交付的内容有以下八项：

- 描述系统语境的一个相当完善的业务（或领域）模型。
- 所有模型（用况、分析、设计、实施及实现）的新版本。（在细化阶段结束时，完成将近10%的用况，而在分析模型中可能包括更多的用况（在某些情况下达到80%），从而可以确定已经正确地理解了需求。大多数用况已经被详细地了解，并确信没有遗漏任何对构架有重要意义的用况，从而我们可以估计这些用况的开发费用。）
- 可执行的构架基线。
- 构架说明，包括用况、分析、设计、实施和实现模型的各种视图。
- 更新过的风险清单。
- 构造阶段和移交阶段的项目计划。
- 初步的用户手册（可选）。
- 完整的业务案例（包含业务标书）。

第15章 构造阶段形成初步可运行能力

本阶段的主要目的是生成软件产品的一个初步可运行（发布）版本，即“beta版本”。该产品达到可应用的质量，且满足需求。构造阶段应在业务计划的限制范围内开展构造工作。

在细化阶段结束时，目标系统的开发进行到可执行构架的程度，并将严重的和重大的风险数目减少到构造阶段所能接受的水平。在细化阶段中，项目组为设计模型和实施模型中对构架有重要意义的模型元素搭建好了基础，这些基础包括子系统、类、构件以及它们之间的接口，还包括对构架有重要意义的用况的实现。此时，在详细描述的基础上所完成的用况占用况群的将近10%。请回想一下，几乎所有的需求都已捕获（通常大约为80%），但根据细化阶段的目标，不必详细说明所有这些需求，这些工作是构造阶段所要讨论的内容。

15.1 构造阶段概述

开发组在构造阶段的工作是从一个可执行的构架基线着手，通过一系列的迭代和增量过程，开发出一个准备在用户环境中初步运行的软件产品。这一过程，通常称为beta测试。该过程详细描述剩余的用况和场景，修改构架描述；必要时，通过附加的迭代，对分析、设计和实现模型加以补充和完善。它集成子系统并加以测试，最后集成整个系统并进行测试。

当工程从细化阶段过渡到构造阶段时，工作重点会有所变化。初始和细化阶段可比喻成“研究”，而构造阶段则类似于“开发”。重点从收集构建项目所需要的基本知识，转向在给定的费用、工作量和进度范围内实际构造系统或产品。

在构造阶段中，项目经理、构架设计师和高层开发人员应该确保：对用况进行优先级排序；对用况加以分组并形成构造和迭代；按照规定的顺序开发并避免返工。

通过持续地更新来保持“风险清单”的实时性，这样才能反映项目当前的实际风险，目标是在该阶段结束时，所有的风险都得到缓解（运行过程中或移交阶段中发生的风险除外）。构架设计师应密切关注构造是否始终符合构架，必要时可修改构架以适应在构造工作中发生的变化。

15.2 构造阶段初期

细化阶段结束时，项目经理制定构造阶段计划。当他或她得到投资方的授权后，可以根据现实情况修订计划。特别要注意下面两种经常可能发生的情况。

其一是细化阶段和构造阶段可能出现的时间空隙。投资方可能会立即授权项目经理及其成员继续工作以保持他们对该项目的详尽了解。不幸的是，从建议的提出、标书的认可到协议的签订或其他授权之间，也许会有好几个月的时间空隙。在此期间，项目经理和开发组成员分散到其他工作，从而导致在得到继续进行的许可后，人员不能召集齐全。最糟糕的情况是，由一个新的项目经理以及大部分新的人员来接手构造阶段的工作。

另一种情况则是资金和进度比项目在细化阶段所计划的要少。项目规模也许能缩小到与资金和进度相一致，但也可能做不到。所要指出的是，在构造阶段开始时，现实环境已或多或少地不同于细化阶段末期制定计划时所依据的环境。无论是否喜欢（通常是不会喜欢的），在某种程度上，项目经理将不得不重新制定计划。几乎在每种情形下，他都得从细化阶段开始调整计划以符合可用的人力资源和项目相关人员所设定的进度。

15.2.1 本阶段人员安排

构造阶段的工作是在构架（即对构架有重要意义的模型元素）上展开的。构架设计师在细化阶段识别的服务子系统并不完善，它们仅实现了核心用况及其核心场景。项目经理会指派别的人员去完成这项工作。

带有子系统和接口陈述的构架基线，是项目经理划分工作的依据。每一个子系统会由一位担当构件工程师职责的开发人员来负责。按第9章的说法，通常负责某个子系统的开发人员还要负责该子系统中的所有类。通常并不采用一个开发者负责一个类的做法，那样做，工作的划分会相当繁琐。

按照构造阶段的主要工作，设置如下的岗位：用况工程师、构件工程师、测试工程师、系统集成人员、集成测试人员以及系统测试人员。同细化阶段相比，员工数量大约增加一倍。表13-1表明，每个工作流都有大量的工作（分析工作流中60%的工作，设计、实现和测试工作流中90%的工作）留待构造阶段来完成。

除了上面列出的工作人员外，构架设计师在这一阶段仍然会起作用，尽管他们投入的时间通常很少。此外，因为有20%的需求分析还没有完成，系统分析人员和用况说明人员仍将继续工作。

15.2.2 设立评价准则

由迭代或构造阶段产生的特定的评价准则对每个项目都是唯一的。事实上，这些准则已经在开发用况的过程中制定出来。在前面的章节中曾提到过，用况本身就是符合功能需求的，同时它们还含有非功能需求，如性能需求。用况可以用来缓解风险。每个构造或迭代都实现一组用况，因此，对这一组用况的评价就基于与其相关的功能性或非功能性需求。

这些与用况相关的评价准则也使得开发人员在完成特定的构造或迭代后能更清楚地了解自己的工作情况。

此外，制定构造阶段所需要的评价准则还需要准备别的一些补充材料，例如：

用户资料

为支持最终用户，在构造阶段需要准备文字资料的第一个剪辑。如用户指南、帮助文件、版本信息、用户手册和操作手册等。评价准则是：这些文字资料能否为用户提供充分的支持？

教程产品

为支持最终用户，在本阶段还需要准备教程的第一个剪辑，如幻灯片、提示信息、示例和辅导材料等。评价准则是：在移交阶段，它们能否为用户提供充分的支持？

从总体上来讲，对于构造阶段，评价准则是：初步可运行能力是否足够成熟与稳定，能否达到把beta版软件交给用户团体时不会让软件组织或者用户团体遭受无法接受的风险？

15.3　原型的构造迭代工作流

原型迭代包括五个工作流，如图15-1所示。在第二部分对工作流进行详细描述，本章仅考虑它们在构造阶段中的作用。我们再次部分并行地开展四组活动。第一部分是五个核心工作流；第二部分是迭代计划，在12.4节~12.7节及15.2节中详细讨论；第三部分是业务案例，在13.5节和15.5节中描述；第四部分是评估，在12.8节和15.6节中描述。本节先提供对五个工作流的概览，下一节再详细讨论。

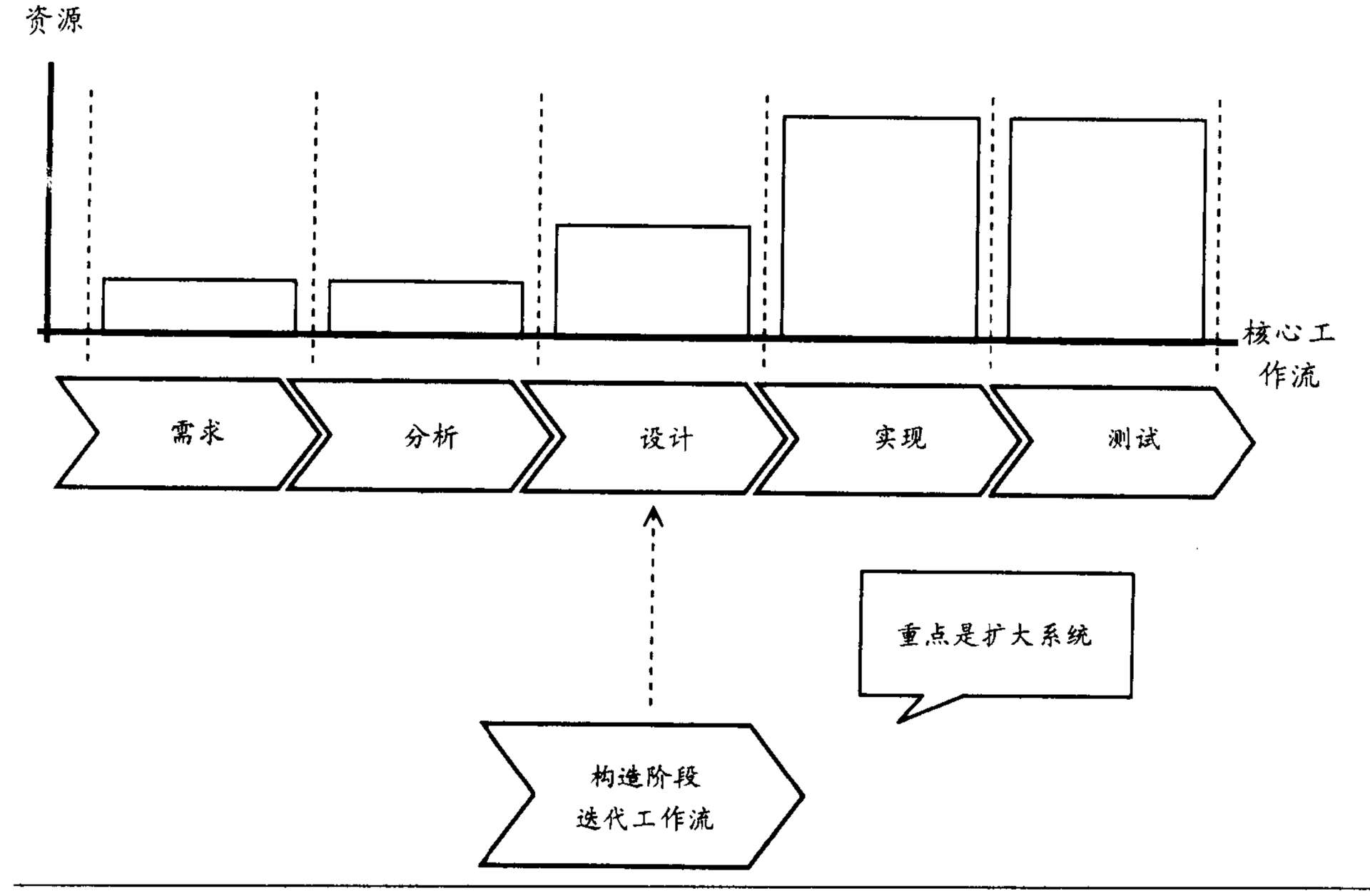


图15-1　构造阶段中的迭代工作按五个核心工作流（需求、分析、设计、实现和测试）进行。剩余的需求在该阶段详细描述和分析，但工作量很小。绝大部分工作已在前两个阶段完成。设计扮演着重要的角色，这正是在本阶段的任务，实现和测试工作流的绝大部分工作开始进行。（矩形框的大小仅用于指示，根据不同的项目会有所变化。）

在构造阶段的早期迭代中，初始工作流获得较多的关注，往后则逐渐减少。在整个构造迭代中重点逐渐转移。例如，如果画一条钟形的曲线描述连续的工作量，曲线的峰顶将逐渐从左向右转移，每次相继的迭代都逐渐把注意力放到实现工作流。

建造系统

到目前为止，需求和构架都很稳定，重点在于完成所有用况的用况实现，设计所需的子系统和类，并用构件实现它们，然后既进行单独的测试，又构造在一起进行测试。在每个构造中，开发人员都获得由项目经理和系统集成人员给定的用况集，并实现它们。

用况驱动、以构架为中心和迭代式的开发方法按相当小的增量构造软件，每次向前面增量的聚集增加一个增量，且总是保持一个可执行的构造。它将日益增多的增量进行排序，开发人员就用不着返回到前面的几个增量并重新实现它们。

用增加相当小的增量的方法来开发软件使得项目更容易管理。它将分析、设计、实现和测试工作流的范围局限在单个增量内所发现的少量的问题和难点上。它还可最大限度地隔离风险和缺陷，并将修复工作局限于单独的构造中，使得它们既容易发现又便于处理。

尽管前期的各个阶段已经研究并缓解了严重的和重大的风险，但项目管理仍然在风险清单上保留着许多风险。另外，在开发人员进行开发与迭代时，在用户进行实验时，新的风险会随之出现。例如，大多数程序设计语言已经沿用了很长一段时间，一个项目也理所当然地认为它计划采用的编译器的功能是满足要求的。而事实上，语言在发展，编译器也在更新，一个更新过的编译器可能包含缺陷。在一个有8万行源码的项目中，项目经理最终会发现在后继测试中，重复性错误都是由编译器引起的。于是开发工作将拖延更长的时间，直到编译器的生产厂家发现了那些难以捉摸的错误并加以修正为止。

15.4 执行五个核心工作流——从捕获需求到测试

前一章节概述了构造阶段的内容。在本节以图15-2为指南进行更详细的讨论。与前几章节相同，本节的内容也是按五个工作流来组织的。尽管排列有先后次序，但各工作流中的工作实际上是并行的，如图15-2所示。

在本阶段的工作过程中，工作人员参与到第二部分所勾画的工作流中，在构架基线或后续迭代的所有制品的基础上构造系统。

在细化阶段中项目大约设计和实现了不到10%的用况，仅够建造构架基线使用。现在，项目在构造阶段所面临的任务是往构架“骨骼”中加入“血肉”，也就是要往第4章和第5章所描述的模型中填充内容。特别地，请看图5-7，它展示了在四个阶段的进展中模型的填充过程。每一个结构和迭代都向进化的模型结构中增添了更多的用况、类、子系统和构件。

以下各小节将按图15-2所示顺序依次讲述各个工作流，但工作人员不必按照这样的次序进行工作。例如，在每次迭代中，会提早制定测试计划以启动迭代所需的测试工作，因而对测试计划的处理可以先于对用况、分析、设计和实现的细化。但这种并行工作并没有在以下章节中得到体现。此外，需要反复强调的是，每次迭代都要重复这五个工作流。

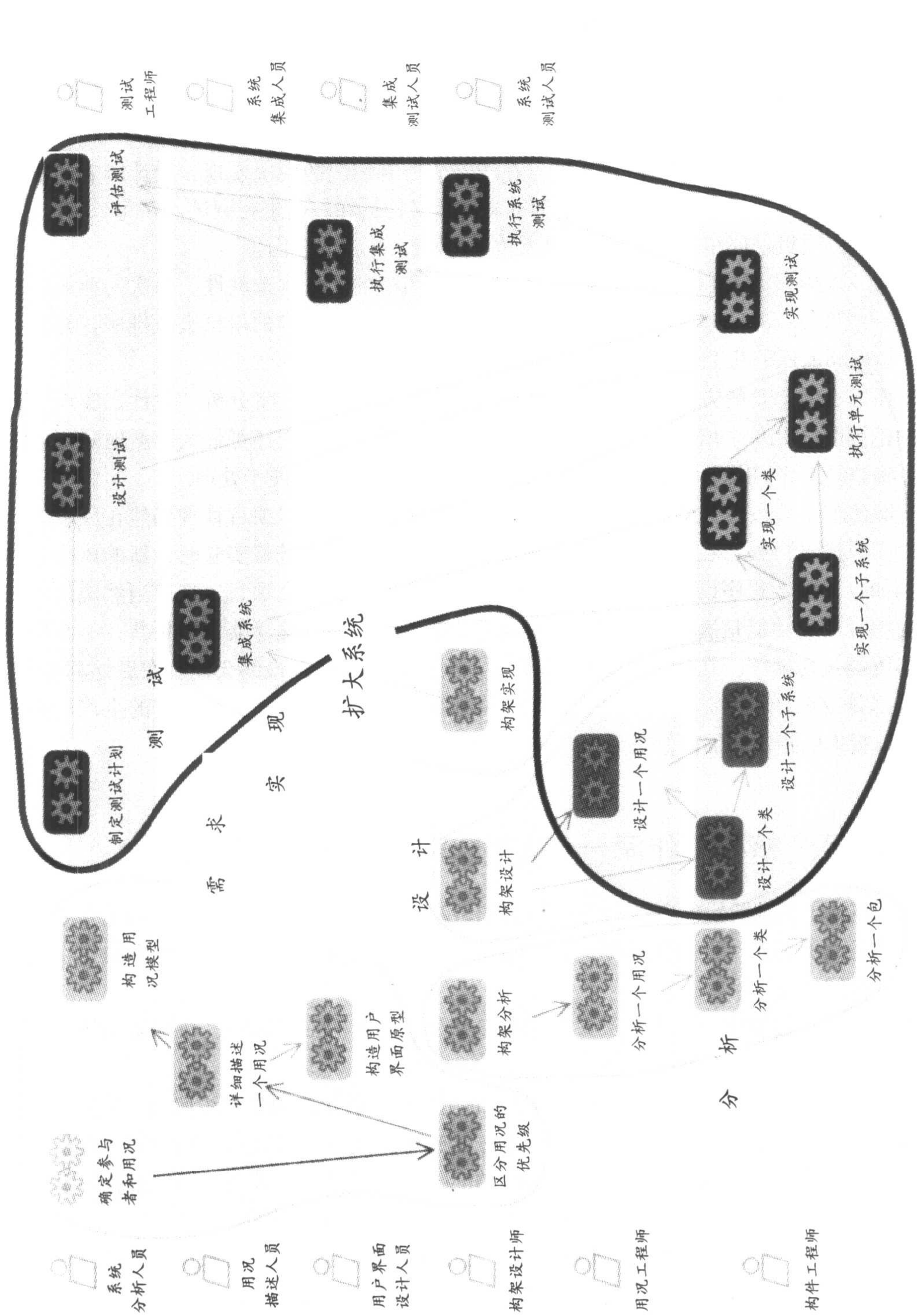


图15-2 手画粗边线所圈定的部分重点是构造阶段的主要活动

15.4.1 捕获需求

介绍了构造阶段的概要之后，我们转而去寻找用况和参与者、原型化用户界面以及详细描述并构造用况。在细化阶段，定义了所有的用况和参与者，也理解了用况群的80%，但仅深入描述用况群的20%，从中再挑出少许的用况（那时需要实现少于10%的用况），因为建立构架基线仅需少量的细节。当然，在构造阶段，要千方百计地实现一个初步可运行的系统，因此，不得不全面地展开需求捕获工作（即要识别并细化100%的用况）。

确定参与者和用况

通常只有少数（不到20%）参与者和用况留到构造阶段来定义。如果必要，由系统分析人员更新用况模型中的参与者和用况。

构造用户界面原型

一般来讲，不在初始阶段和细化阶段构造用户界面原型，除非需要用于演示的目的或出现了新型界面。而现在，用户界面就非要设计不可了。工作量则由所建系统的类型确定。

对某些系统，尤其是其中的一些用况需要相当复杂的用户界面的系统，不建立原型就很难把握用户界面的设计。因此，建立一个原型（或者，如果需要的话，建立多个原型）并请用户试用，依据用户的反馈来修改，直到该原型最终满足用户的要求。请注意，设计用户界面是需求分析工作流的任务，而不是设计（即使在叫法上是设计）的任务，且必须在进入下一个工作流之前完成。这样设计出的原型就是用户界面的规格说明（细节参见第7章）。

对于需要大批量出售的系统，即使其用户界面不复杂，也要为它设计原型，因为在这种情况下更换一个不成功的界面开支实在太大。

区分用况的优先级

细化阶段曾对开发构架基线所需的用况进行了优先级排序，在本阶段定义用况时，同样要按顺序把它们加进优先级清单中去。

详细描述一个用况

按剩余用况和用况场景的重要性，由用况描述人员完成细化工作。

构造用况模型

系统分析人员也许希望改善用况模型的结构，但系统构架在本阶段已较为稳定，任何改动都要首先考虑还没有处理的用况。每一个做过修改的用况在分析和设计模型中也要有相应的变动。

15.4.2 分析

本小节再次考虑构架分析活动，对在细化阶段初始化的用况、类和包进行分析。在细化阶段，仅关注那些对整体构架和业务标书至关重要的用况。为使读者明白我们的意图，在细化阶段我们分析了40%的用况，而这只是决定业务标书的用况的典型值——80%的一半。请不要单纯地理解数值本身，因为对于每个项目该比例会有所不同。在构造阶段，要考虑所有用况，但可能并没有必要因此而扩大分析模型。

如第8章所讨论的，存在这样的情况，对于一个新产品，经过第一次细化阶段后的分析模型

不再继续使用。而在另一些情况中，项目经理会在构造阶段中继续使用分析模型，直到项目结束甚至超出产品生命期。因为后一种情况较为复杂，所以在后续的工作中我们默认其存在。细化阶段和构造阶段的本质区别在于：在构造阶段，完成分析模型。细化阶段结束时所得到的模型很大程度上是构架视图，离真正的构架还有很大的一段距离，它只是整个分析模型的一部分。在构造阶段末期，整个分析模型才真正完成，构架视图只是其中一个很小的子集。

构架分析

到细化阶段结束时，构架设计师已准备好分析模型的构架视图，除了发生影响构架的修改而需要对构架进行更新外，在构造阶段很少进行构架分析活动。

分析一个用况

在细化阶段，构架设计师仅使用那些对构架有重要意义的、并应用于分析模型的构架视图的用况。在构造阶段的每次迭代中，我们要用该迭代所包含的用况来扩展分析模型。

分析一个类

构件工程师继续他们在细化阶段开始的工作。

分析一个包

构架设计师在细化阶段识别包，并在构造阶段将其提炼到新用况所需要的程度。在构造阶段，构件工程师负责包的维护。

15.4.3 设计

在本阶段，通常要设计并实现余下的、没有被用于开发构架基线的90%的用况。再次强调，当考虑设计工作流时，设计工作流和其他工作流会重复出现在每一次迭代中。

构架设计

在构造阶段，构架设计师通常并不是增加设计子系统或服务子系统，这些部分都已存在，诚然，它们在构架基线中是以“骨骼”形式存在的。

构架设计师也可以增加子系统，前提是它们与已经存在的子系统相似，或者可以与已经存在的子系统相互替换。例如，对于一个通信协议，已存在一个子系统，那么增加一个不需要新接口的子系统是允许的。

服务子系统的行为通常可以由一个用况的一部分或一组相关的用况导出。或者说，服务子系统担当支配性的角色，以帮助实现单个的用况。服务子系统的40%~60%的职责来自这样一个用况。当比例太高时，比如说80%时，项目经理就会考虑剩下的20%是在同一构造中实现，还是留给后面的构造去实现。这时，授权完成该构造是个很不错的想法，即使该子系统的其他部分源自优先级更低（但占相对支配地位）的用况。虽然这些余下用况优先级较低并且可能被推迟处理，但当同时考虑所有用况时，可能会更有效地完成整个包的功能。

一方面，此时处理低优先级的用况使得构件工程师能够一次性地开发出整个子系统。因为具有能同时检查整个子系统的优点，所以就有较大的可能来保证该子系统结构的正确性。如果他或其他人不完成整个子系统，而要在稍后的构造中回头处理该子系统的这些优先级相对较低的用况，那么他会发现可能已经出现了矛盾，而要对整个系统进行改动。也许会因为原有子系

统与新增的、由低优先级用况所导出的代码不能很好地协同工作，因而不得不重写代码。因此，如果一些以后才被设计的用况的影响很明显，不妨将这些用况包括进来一起进行处理，这一点很重要，理由是一份工作最好一次做完而不是迭代多次。

另一方面，在早期构造中处理低优先级用况与通常的尽早检验高优先级用况的惯例相冲突。一般的处理办法是：如果方便且时机恰当，就选择少数低优先级的用况和高优先级的用况一起处理。即使如此，我们的目标仍是保证设计正确，而不是要实现和测试所有被选择的低优先级用况。项目经理应该将那些低优先级的用况的设计、实现和测试工作，推迟至稍后的由其优先级所确定的构造中。

构架设计师要改进设计和实施模型的构架视图以反映构造阶段的经验。然而，通常在细化阶段末期已完成架构，现在所要做的是对其进行更新。关于该工作流的其他活动（用况、类以及子系统设计）的详细情况，请参见第9章。在这里，有足够的理由可以说明设计是构造阶段的一个主要焦点（实现当然更是），如图15-1所示。其最终结果是设计模型和实施模型，两者在整个开发期间都要加以维护，甚至会延续至未来版本的开发周期中。设计模型是实现模型和实现本身的“蓝图”。

15.4.4 实现

该工作流首先从设计模型出发，实现并执行所有构件的单元测试。一定次数的迭代后的结果连同系统集成和测试就形成了初步可运行的版本。本小节内容包括构架的实现活动：实现类和子系统，执行单元测试并集成系统。

正是在该工作流内，项目完成构造阶段的绝大部分工作，构造第10章中所描述的构件。项目对每一构件多次执行迭代开发，获得一个一个构造，并对每个构造不断填入代码，直至本阶段结束，所有构件的开发即告完成。

构架实现

到此时，构架牢固地建立起来。构架设计师的任务只是在必要时更新构架。

实现一个类和实现一个子系统

构件工程师对实现模型中的类和子系统加以实现。他们实现建造构造所需要的尚未完全完成的类。

执行单元测试

构件工程师负责对其所开发的构件执行单元测试，必要时，纠正构件的设计和实现。

集成系统

系统集成人员要制定一个集成计划，勾画出构造的顺序。该计划阐明构造要实现的用况或者用况中的场景。这些用况和场景导出子系统和构件。

系统集成人员发现，从分层构架体系的较低层次的构造开始是比较可取的，例如系统软件层和中间件层（见图4-5）。往后的构造则向通用应用层和专用应用层扩展。没有低层功能支持是很难把构造集成起来的。

例如，在一个化学公司里，软件开发组织计划平均每两周进行一次增量开发。而据说微软

的项目在达到具有代码构造的程度后，每天完成一个构造。每个构造要进行测试（新增代码的测试和修改代码的回归测试），以保证开发代码的可用性。这种每天一次的构造过程核实每天提交的代码单元，因此与前一天是兼容的，使得开发人员不能中断构造。然而同时它也减少了项目组织的长期压力，因为集成问题在测试中被发现（通常是在每天晚上），并很快就得到解决。每天完成一次构造听起来会给项目增加相当大的时间上的压力，但事实上并非如此。开发人员在代码已准备妥善后再提交，提交还未准备妥善的代码是没有意义的，那样做有可能破坏构造。然而，每个开发人员确有时间上的压力，因为他要及时提交代码以适应构造集成计划。

对一个组织而言，每个构造周期的长短是一件值得好好考虑的事情。一个指导性的原则是：以足够的频度进行构造，就会得到相应的好处。

为使构造较小，系统集成人员通常需要用“桩模块”和“驱动模块”来代替尚未构造的构件。桩模块是一种简单元素，它仅能对一种激励或该构件可能接受的所有激励做出反应；驱动模块则用于提供激励。因为桩模块和驱动模块都很简单，不会引入额外的复杂性。

系统集成人员开始考虑以怎样的次序集成各个构件以生成一个满足功能性需求的、可测试的配置，并将其发现归档到构造集成计划内，以说明每个构造在什么时候需要进行集成和测试。系统集成人员按照构造计划将已完成的类和未完全实现的类合并成为可执行的构造。他继续建造后续的更大的构造，集成测试人员对其进行测试和回归测试。在每次迭代的最后一步和阶段本身的最后时刻，系统集成人员连接整个系统，系统测试人员对其进行测试和回归测试。

这样的安排给出了在每次迭代中构造的顺序和构造阶段中迭代的顺序。因为分层构架中高层的编译依赖于低层，所以系统集成人员一般自下而上地进行编译。

15.4.5 测试

测试工程师的工作，如第11章所述，是发现可有效测试的内容、设计测试环境、测试规程并对其进行测试。这些工作在构造阶段获得了结果。如图15-1所示，测试是构造阶段的重要一环。因为执行单元测试是构件工程师的任务，测试工程师仅提供技术支持。构件工程师及其同事（集成测试人员和系统测试人员）负责对构造（即每次迭代之后的增量以及最后的构造，也就是系统的完整版本）进行测试。

制定测试计划

测试工程师选择每个构造及系统本身（最后的构造）的测试目标。

设计测试

测试工程师找出如何测试构造集中的需求的方法，以验证可测试的需求的正确性。为此，他们要准备测试用例和测试规程。他们从以前构造的测试用例和测试规程中选出那些依旧能使用的测试用例和测试规程并加以修改，用于对后续构造的回归测试。测试工程师按测试计划中原先所确立的步骤，验证要集成在一起的构件的正确性。集成测试的目的在于验证待测试构件之间的接口的正确性以及和构件在一起工作的实际情形。

执行集成测试

集成测试人员按照测试规程运行测试用例，构造通过测试后，系统集成人员再增加新的构

造，到可测试时，测试人员再进行测试，如此反复。当集成测试人员发现错误时，就记录下来并通知项目经理，由项目经理（或者他指派的人员）决定下一步该如何做。指派的人员可能是具备相关知识的系统集成人员。例如，下一步可能是同一构架中进一步的工作，把对缺陷的修复延期到下一个构造；或者，尤其在错误非常严重的情况下，要派专门人才加以研究。

执行系统测试

当持续的构造过程到达一次迭代的末期时，就形成了一个局部性的系统，并进入到系统测试人员的权限内。系统测试人员按照测试规程运行测试用例，如果发现错误，同样要记录下来并通知项目经理或其所指派的人员以寻求解决方案。

构造阶段的最后一次迭代结束时，系统测试人员测试初步可运行版本，然后将错误反馈给项目经理，由项目经理分配给相应的构件工程师负责修复。

评价测试

当进行集成和系统测试时，测试工程师依据测试计划中的原定目标（可能被后续迭代修改），在每次构造完成时，评审测试结果。评价测试的目的是确保测试达到目标。如果某个测试没有达到目标，则要修改测试用例和测试规程，以达到测试目标（参见11.5.6节）。

15.5 控制业务案例

在细化阶段结束时准备好业务标书的目的之一是为项目经理和项目相关人员提供实施构造阶段所需的指南。在每次迭代后，项目经理将实际进展与原计划的进度、人力和费用进行比较，并审查项目开发过程的数据、代码质量、数据库大小以及其他一些指标。

完成代码的行数并不是基于构件的开发的好的度量标准。因为这种开发方法的目的之一是重用类和构件，构件工程师和其他员工几乎不用编新的代码，就能在构造和迭代上取得良好的进展。在这样的情形下，检验工作完成状况的一个更好的指标是构造和迭代按照计划的完成情况。

如果差异（尤其是负面的差异）不小，项目经理就要采取纠正行动。同样地，如果差异相当大，就要召开有项目相关人员参加的评审会。

在构造阶段，随着项目经理对产品费用和实际性能拥有更深入的了解，他会觉得有必要更新业务案例并与项目相关人员探讨新的情况。

15.6 评估构造阶段的迭代

基于测试结果以及其他评价准则（包括15.2.2节描述的那些评价准则）的评审，项目经理和评估小组着手实施以下事项：

- 审查迭代中的工作是否有与原计划相矛盾的内容。
- 计划在后面的哪一次迭代中实现未完成的工作。
- 确定构造是否完成以决定是否进入下一次迭代。

- 更新风险清单。
- 制定下一次迭代的计划。
- 为下一次迭代之后的迭代更新计划。
- 在本阶段的最后一次迭代结束时，确定产品是否已通过系统测试并且已达到初步可运行能力。
- 授权进入移交阶段。
- 更新项目计划。

15.7 制定移交阶段计划

项目组不能期望像以往各阶段那样，预先为移交阶段制定详细的计划。当然，小组成员知道要将beta版（或相当物）发放给选定的用户以征求意见。移交阶段的部分内容，如选择beta测试人员，复制可执行代码和准备测试指令等，都可以计划得稍微详细一些。

项目组极少能提前预知将会收到的反馈（如风险、问题、错误、建议等）。如果小组具有beta测试经验，就能预计大概的情形，也会估计出需要多少人去处理beta测试人员发现的问题。

15.8 关键的可交付内容

可交付的内容包括：

- 移交阶段的项目计划。
- 可执行软件本身——初步可运行版本，它是构造阶段的最后一个构造。
- 包括系统模型在内的所有制品。
- 维护并最低限度地更改构架描述。
- 指导beta版用户的足够详细的初步的用户手册。
- 反映本阶段后期情形的业务案例。

目的是希望被标为“完善的”可交付内容确实是完善的。但是在移交阶段，当用户方对软件进行操作时，可能会发现其中的一些实际上并不正确，这些错误将会被纠正。

第16章
移交阶段完成产品发布

当项目进入移交阶段时，系统已具备可初步运行能力。

此时，项目经理相信系统已能在用户环境中运转，尽管不是很完美。例如，在构造阶段末期的系统测试中，一些问题、风险和缺陷尽管没有显现出来，但在用户环境中可能会显现出来。用户可能在后来才发现遗漏了一些他们所需要的特征。如果这些特征非常重要，并与现存产品是一致的，项目经理可能会同意增加它们。然而，这些改变必须很小，且其引入不会严重影响项目计划。如果一个新提议的特征确实影响了计划，对它的处理就要格外谨慎。在大多数情况下，我们认为最好是把它们加入到特征清单中，并且把它们留到下一个开发周期去处理，也就是说，留到系统下个一版本的开发中去处理。

本阶段的基本目标是：

- 要满足初始阶段所确立的需求，让项目相关人员满意。
- 处理在用户环境中运转时遇到的所有问题，包括纠正beta用户或验收测试人员所反馈的缺陷。

验收测试是客户的职责，尽管一些客户把它委托给专门的验收测试组织。

16.1 移交阶段概述

本阶段重点是要在运行环境中安置产品。根据产品与其市场的关系的性质的不同，项目实现这一目标的具体途径会有所不同。举例来说，如果一个产品将要进入市场，项目组就会将一个beta版本发布给位于有代表性的“beta客户”站点的典型用户。如果一个产品将给一个单独的客户（或者可能是一个大型组织内的许多站点），则项目组就只在单个站点上安装产品。[⊖]

项目组监控从运行站点反馈回来的信息，用来完成以下工作：

- 发现系统是否真正实现了业务及其用户的需求。
- 发现没有预见的风险。
- 留意未解决的问题。
- 寻找错误。
- 修复用户文档中不清楚和有遗漏的内容。
- 把重点放在用户显得无能为力和需要信息或培训的地方。

⊖ 我们可能提到其他两种可能：alpha (α) 测试和第三方确认。alpha (α)测试和beta (β) 测试很相似，它是在开发该软件的公司的内部、由除了开发该软件的组织以外的其他组织进行的。执行alpha (α)测试的人员是实际的用户；在第三方确认中，专门从事测试的公司按照与客户签订的合同完成验收测试。

基于这些反馈信息，项目组修改系统或者相关的制品。通常，开发组要进行软件的产品化、包装、实施和系统的移交等工作。

本阶段并不是要对产品进行重新整合。项目组和客户应该在初期阶段中就处理了重大的需求变化。确切地讲，项目组要寻找在构造阶段被忽略的细小错误，这些错误可以在现有的产品基线内得到纠正。

在用户关系上，项目组也可以在建立良好的产品运行环境和培训用户有效地使用产品等方面提供帮助。它可以帮助这些用户在运行新系统的同时并行地运行将被取代的旧系统，这样有助于将旧数据库转换到新的配置环境中。

针对产品发布给大量用户使用（“包装后在市场上销售”）的情况，要将以上的服务合并到安装程序中（安装程序是产品的一部分），并通过桌面帮助的方式来补充。

移交阶段随着产品的发布而结束。

16.2 移交阶段初期

软件是在大量的业务安排下进行生产的。不用了解许多细节，可以把这些安排归为两种类型：

由软件供应商开发，在市场中销售给多个客户

有时这个市场非常大，例如个人电脑市场。有时会比较小，例如为软件生产者生产可重用构件，或者生产可适应每一种安装情况的预制程序。在这些市场中，尽管规模可以改变，但因为存在一对多的关系（供应商和许多的客户），将会影响移交阶段的工作。

软件开发机构与单个客户签订的开发合同

这种模式又分为几种不同的情况，如：

- 软件组织为单站点上的单个客户开发程序。
- 软件组织为拥有许多下级单位和站点的客户开发程序。
- 任务外包的软件公司，如EDS、Computer Science 或者Anderson Consulting，有时候在刚开始时是给单站点或单客户开发软件，但后来会对其加以改造以满足其他站点或客户的需求。
- 在公司现定的财务安排下，为同一公司内的其他部门开发软件的内部软件开发组织。

移交阶段的组织和用户/客户的关系在这些不同的模式下会有所不同。在这些模式下，随着初步可运行版本通过内部的系统测试以及在构造阶段结束时主要里程碑的评估的完成，移交阶段便开始了。然而，在移交阶段的项目组要准备额外的制品，如安装脚本、数据转换程序或者数据移植程序；或者要超出自身的职责，要对在构造阶段产生的制品加以修改，以准备好用于发布的可执行程序。

16.2.1 制定移交阶段计划

项目处于构造阶段时，很难预先制订移交阶段计划的大量细节。另一方面，项目经理知道

本阶段将获得在构造阶段开发的beta版本（或其等价物），并发布给选定的用户进行测试。这一认识为移交阶段的早期规划提供了基础。众所周知的工作包括：生产beta版本，准备beta测试文档，选择beta用户，等等。另一方面，还有一些鲜为人知的因为beta测试反馈而产生的工作。如果项目经理想要预留一些人手的话，他可以把这些人先分配到其他项目中去工作，但一旦出现移交方面的问题时能启用他们，这样可以节省人力。

在制完移交阶段计划时，项目经理希望从构造阶段获得的初步可运行版本经过beta测试后只需要做很少的修改工作。事实上，如果一个项目是在迭代的基础上进行的，就能满足这样的要求。这种开发过程允许开发人员在早期的迭代中进行试验，并在早期的增量测试和运行观察中发现这些迭代中的概念性错误。同样地，随着构造工作的逐步进行，普通的错误会被发现并加以纠正。简而言之，早期的重复工作是有好处的。在移交阶段，仿效迭代式开发，重做的工作应小于5%。然而，项目经理应当预料到这个比例会大于零。至少，一些冗长多余的信息和错误会出现在所有的屏幕上。一个项目如果忽视可能的返工会造成如下影响：

- 进度安排太紧，从而导致“欲速则不达”的综合症。
- 在构造阶段末期缺乏大量的系统测试和评估。
- 不能将重点放在仍然处于移交阶段的应该加以考虑的工作上。
- 过分担忧因为返工要耗费大量的物力。
- 认为返工是很“糟糕”的事情，是无能的一种表现。

在制定移交阶段计划时，可能会出现什么才是“足够好“的软件这一问题。有一个不争的事实，那就是没有一种软件产品是十全十美的。举个例子，产品在发行时仍存在着一定百分比的缺陷，有一些需求会推迟到稍后的版本中去实现，或者会有一些由beta用户发现的需求在移交阶段因缺乏条件而难以实现。对于“足够好”问题，有三个“答案”。

首先，统一过程的阶段和迭代有意识地去识别风险，捕获正确的需求，并合理地制定项目计划。在统一过程中，开发组尽力搞好与用户和客户的关系。其结果是，初步可运行版本，或者说beta版本，能够很好地接近开发方和投资方的期望。

其次，既然项目组和项目相关人员都不期望初步可运行版本没有一丝一毫的缺陷，他们都应该会允许为移交阶段安排时间和资源。

第三，既然项目组和项目相关人员通力合作并达成以上两个共识，其结论是，可能会延长移交阶段或者将不希望做的工作延期到下一个开发周期。

16.2.2 移交阶段人员安排

由于这个阶段的重点是给用户一个发布版本，首先要对产品进行beta测试和验收测试，然后才使用它。同构造阶段相比，移交阶段尽管在着重点上有细微的不同，但在人员需求方面却非常相似。

分析beta测试和验收测试的反馈信息并对之作出反应，需要一些更面向服务而不是更面向开发的人员。即使只考虑一个细小的改善所带来的好处，也可能需要专门的人员来处理，他们不仅对系统的绝大部分都精通，而且对系统所采用的应用特征也要十分熟悉。而且，当测试一旦发现

了缺陷，研究其错误的根源需要非常深入地去了解整个系统，或者至少要非常深入地去研究导致错误发生的部分。还有，虽然在构造阶段初步撰写了用户和课程资料，但在送到普通用户手中之前，常常需要由技术文档编写人员重写。在本阶段，构架设计师必须能随叫随到，但仅仅是为了确保和维护构架的完整性，而且有时（尽管此种情况很少）需要考虑对构架作很小的改变。

16.2.3 设立评价准则

在构造阶段的系统测试之后，确信产品满足可初步运行的能力，或者说，满足需求规格说明书中的各项条款。对于移交阶段，只要评价这一阶段出现的问题。基本上存在以下五个方面的问题：

1) beta用户是否使用了所有的关键功能？例如，那些由全部用况所表述的、包含在该领域内已成功运行的产品中的功能。
2) 类似地，产品是否通过了由客户进行的验收测试？测试准则在开发组织和客户双方同意的合同中确定。此外，验收测试常常是在一个双方都同意的时间段内以生产模式运行软件。
3) 用户资料是否达到可验收质量？
4) 是否已经准备好所需要的课程资料（如果必要的话，包括教师指南）？
5) 最后，客户和用户是否对产品表示满意？在16.4.6节中将对此问题作进一步的讨论。

16.3 核心工作流在本阶段中扮演了很小的角色

在本阶段，五个工作流的活动频度都处于低水平。因为几乎所有的工作都在构造阶段完成

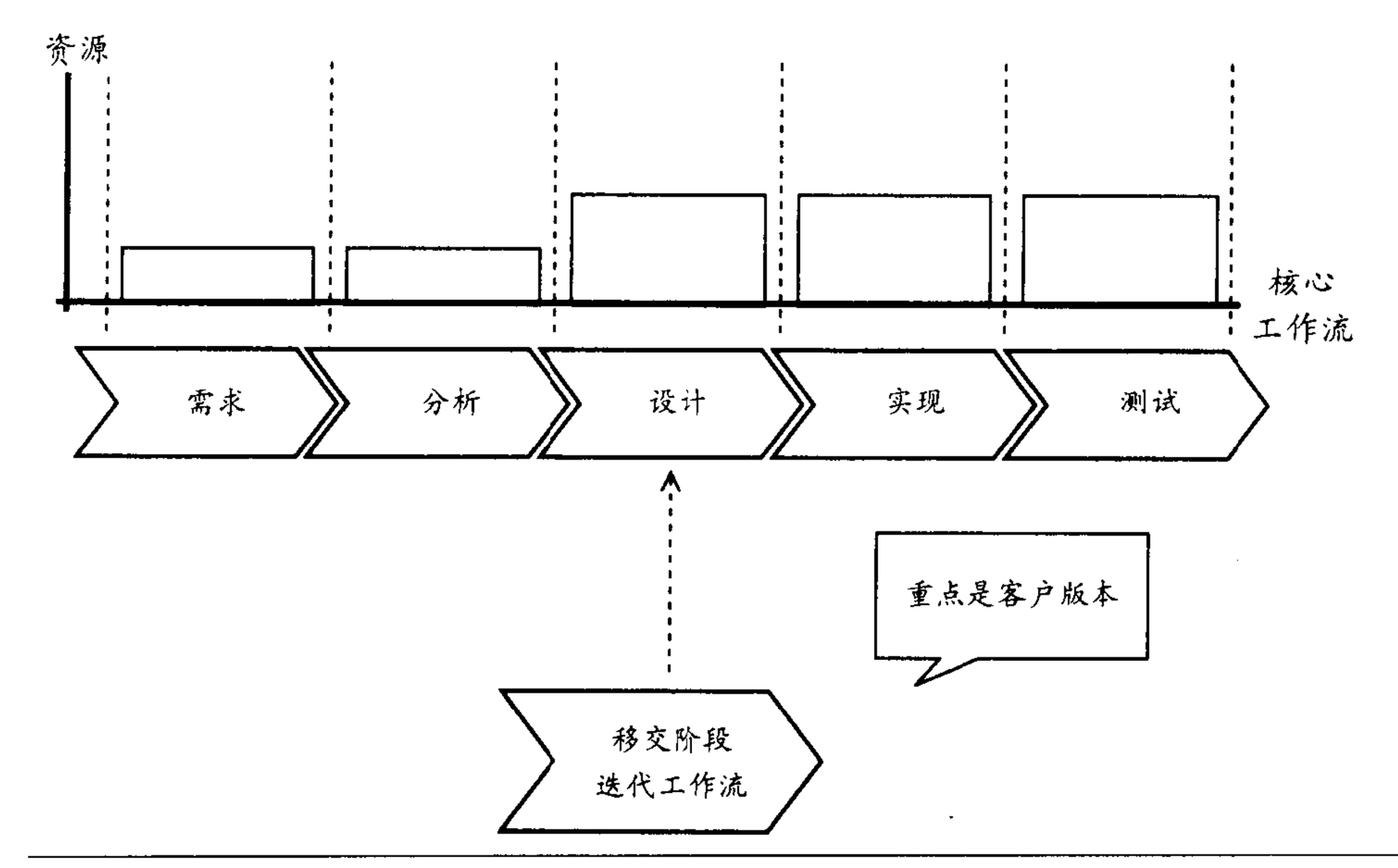


图16-1　移交阶段中的迭代工作按与早前各阶段相同的五个工作流进行。（图中矩形条的大小仅是示意性的，不同的项目会有所不同。）在本阶段，实现和测试工作流备受关注，正是在这里，对在beta版试用和验收测试中发现的缺陷进行搜索、识别、修复和再测试。在很少情况下，需要通过重新设计来修复缺陷，从而导致设计工作流的细微增加，而需求一般很少受到影响

了。在本阶段的活动水平处于很低状态，只要能够纠正在使用环境的测试中发现的问题即可。然而，这并不是说在本阶段没有大量的工作可做（详见16.4节），而是几乎没有与需求、分析和设计工作流相关的工作。在移交阶段设计活动通常很少，在任何情况下，它们大都是些很小的设计改进，以改正问题或者缺陷，或者实现最后的（通常是非常小的）完善。现在的重点已经转移到对缺陷的修复，以消除在早前的使用中发生的故障，确保修复工作本身的正确性，并通过回归测试确保修复工作没有引入新的缺陷。

在前面章节中讨论过原型迭代的五个工作流。本章只关注它们在移交中扮演的角色。说得更明白一些，我们将部分并行地开展四个活动集。一是迭代中的五个核心工作流；第二是制定迭代计划，如12.4~12.7节中所述；第三是对业务案例的进一步考虑，将在16.5节中讨论；第四是评估，在12.8节和16.6节中阐述。在本节中，只给出移交阶段的概述，下一节将增加细节。

16.4 在移交阶段要干些什么

项目按如下顺序执行移交活动：

- 以构造阶段产生的（可初步运行的）版本为基础，准备实际的beta（或者验收测试）版本。
- 在站点上安装（或安排安装）该版本，并在该站点上做一些附加的工作，例如从旧系统移植数据。
- 根据测试站点的反馈采取行动。
- 让目前已改正过的产品能适应用户环境。
- 完成项目制品。
- 决定项目何时结束。

这些活动的顺序在细节上会根据该项目是为市场还是为特定客户开发的而有所差别。在第一种情况下，将有很多潜在的用户，beta用户的选择和指导就将是一个值得考虑的问题。此外，作为真正的用户，他们不会遵循预定的测试进度。他们随心所欲地使用产品，并反馈回发现的问题。

在第二种情况下，客户可能为初始安装和验收测试选择一个站点，在该站点上，将遵循一个正式的、为众人所接受的、系统化的规程。报告结果主要是与正式的规格说明相违背的部分。如果问题超出了已有的合同所确定的范围，则各方都会同意在验收后再作进一步的安排。

根据软件产品是否诞生于一个全新的领域还是在已有的或者相似的产品上的改善，活动的安排顺序也会有所不同。再次，如第二章中所讨论的那样，对软件开发组织来讲，这是一个专门研究对他们所面对的环境如何进行处理的机会。

16.4.1 发行beta版本

大多数beta测试用户将是非常有经验的。移交组织希望他们能利用相当原始的文档进行工作，

但也会为他们提供特殊的指南，用以指导他们如何反馈测试和观察所发现的问题。在移交阶段早期，项目组将早已准备好的beta用户或验收测试人员所需要的文档放在一起，把特殊的beta用法说明补充到文档中。然后选择beta用户，并将beta版本及其相关文档分发给他们。

16.4.2 安装beta版本

在beta测试站上的活动和验收测试站上的活动是不同的。通常存在大量的beta站点，移交人员并不在这些点出现，他们必须发布特定的用法说明，例如怎样安装测试软件，怎样对其进行操作，beta客户应该注意哪些问题，以及如何报告发现的错误和问题。如果该版本是以前软件的升级或替代，移交文档就必须提供用法说明，例如beta用户如何移植数据或将数据库转换成新的版本。还可能要提供beta版本和旧软件在一段时间内并行运行的用法说明。

相反，在验收测试中，相关的项目成员可能出现在测试点。将会有正式的验收测试文档，但以非正式的沟通来补充。如果可能，将故障和问题就地解决，或者在必要时连同适当的限定要求一起返回给项目成员。

16.4.3 响应测试结果

为了决定如何采取行动，项目组要收集并分析测试结果。结果可能会分为以下两种：相对较小的代码故障，只要简单地浏览一下即可将其改正（虽然在某些情况下可能会比较难）；非常重要的问题，其影响可能非常广泛，甚至可能涉及到系统构架的改变。

故障

导致故障的首要因素可能来自构件缺陷，但是该缺陷可能会追溯到设计、分析或更早期模型中的缺陷。如果发现这种追溯比较困难，移交人员可能会从构件工程师或其他进行早期工作的人员那里寻求支持。无论如何，能胜任的人员修复缺陷，测试人员则对其进行测试和回归测试。另外，项目组要考虑如下问题：

- 该缺陷是否与其他尚未发现的缺陷有联系？
- 不改动构架或设计，缺陷是否可以得到纠正？
- 纠正是否没有带来新的问题？

影响面更大的问题

beta测试中的有些难题不是仅靠“故障修复”就能解决的。例如，它们有可能要进行附加的带有测试的迭代。它们也有可能通过一种称为“修改控制委员会”的机制对变化、改进或者特征进行正规化处理。通过该机制，一旦作了修改，将强调保持配置控制的重要性。如前面所声明的，重要的改变会使系统耗费额外的资源，延迟移交，甚至可能会改变构架。所有这样的改变，如果可能，应该推迟到下一个开发周期。

在修改问题和缺陷时，保持构架的完整性非常重要。构架设计师会跟随整个移交过程。他按“紧跟模式”的方式工作，确保对问题和缺陷的修复遵循构架的完整性。他（或构架组）要确保不会采取为图方便但会危害构架的方法来解决问题。为了实现这个目标，有时要求做一些

构架调整。如果该阶段的活动确实影响了构架，构架设计师则会更新构架描述。

当然，几乎没有哪个软件组织生产出的产品是十全十美的。基于这一理由，项目经理必须从业务案例的角度掂量对特定条款进行返工时的花费和时间延迟。

16.4.4 让产品适应不同的用户环境

在早前曾提到过，软件组织按两种广义的关系面向市场并提供产品：上市产品（一对多）和单--客户关系（一对一）。任何一种关系都可能有更进一步的任务：移植数据或将旧系统中的数据库转换到新系统中。

市场关系

市场可能是高度分化的目标集合，开发组必须为不同的用户准备不同的可执行版本。一些变化的因素包括国家、语言、币制和其他一些度量单位。如果产品是在网络中不同的操作节点上运行的，该产品可能不得不为每一个节点做出修改。

使用第一批通用版本的用户很可能不如beta用户有经验，所以开发组要完善原始的beta测试文档，来满足普通用户的需要。

推向市场的产品常常是由用户或公司内的系统管理员安装的。在其他的情况下，由当地供应商负责安装。开发组要提供给安装者可以使用的安装规程，并提供在线帮助时使用的脚本。这些规程可能非常复杂，例如，在产品安装到通过内部网相连的大量个人计算机或工作站的情况。如果产品的某些部分安装在不同节点，并且这些节点必须以预定方式进行安装，此时，安装过程就会变得更加复杂。不同类型节点可能需要不同的安装规程。

单一客户关系

按合同所安排的方式将系统移交给单个用户，在很大程度上遵循与上面刚列举的相同的模式。然而，也存在一些差异：

- 客户代表很可能已经参加了早期阶段，并对增量提供了一些反馈信息。
- 客户代表在软件承包方的提议下观测过最后的系统测试。
- 客户代表已经参加了一些标志通过构造阶段里程碑的评估会议。
- 软件组织可能帮助客户将系统安装到客户方的初始化站点上。至于那些大的、复杂的系统，可能会协助客户完成大部分安装工作。
- 承包方代表会观测验收测试并且当场作出一些切实可行的更改。至于那些更较困难的问题，他们会把细节带回到自己的组织，在专家的帮助下进行改正或修复。
- 在客户和承包方觉得该系统已经满足了他们以前所一致同意的需求后，验收测试就结束了。当然，他们会碰到一些额外的需求或者需求变化，这些就形成新的开发合同内容。

数据移植或转换

通常会有一个现存系统要被替换，从而需要制定数据移植或数据库转换规程。旧数据可能也是在由同一个软件组织开发的软件中，这样就很容易把它转移到新产品中去。然而，如果数据是在由另外的提供者（可能是一个竞争者）开发的软件中，那么移植数据就会有一些困难。

职责可能包括：

- 用新系统代替旧系统有两种情况，或者用新系统完全接手原有任务，或者两个系统同时运行直到用户感到新系统已经运行正常。
- 把数据从旧系统转移到新系统，有时候需要改变数据格式。
- 除了对数据转移提供指导，文档也要包括一些测试，以使用户能够确认安装已经成功完成。
- 程序组要在帮助行中提供进一步的诠释信息，尤其是那些防止用户安装失败所需要的信息。

16.4.5 完成制品

直到项目完成了全部制品（包括模型和构架描述），移交阶段才结束。我们特别强调，模型集合应当在构造阶段结束时完成，而不是直到移交阶段结束时才突然地加以填充完善。更确切地讲，它是在所有迭代和所有阶段中逐渐进化的，如12.8.4节所述。在移交阶段，如果必要的话，会对它加以修改。在移交的最后阶段，通过实际应用已经证明所有制品彼此都保持了一致。

16.4.6 项目什么时候结束

移交阶段的结束，不仅是在所有工作和制品都完成时，而且是在客户“满意”时。那么什么时候用户才满意呢？这要取决于市场关系。

对于用于市场销售的产品，项目经理认为当项目组从beta版本获得反馈并对这些反馈信息进行处理后，大多数的用户就会满意了。那时，再发布一个通用版本。当然，在大多数情况下，环境和产品会持续地进化，但软件组织对这种进化的响应要在稍后的版本里，或者如果是大的调整则要在另一个开发周期里。因为成功的产品和客户系统通常都是先经过小的扩充，然后再进行大的修订，所以实际上“项目”是永远不会结束的。移交阶段要在项目组把继续维护的任务交给一个支持组织的时候才结束。

对于与特定用户签订合同的产品，项目经理认为要在系统通过验收测试后，客户才算满意。这一点，当然取决于在原始合同（在详细描述后签订）中对需求的解释，也取决于在后续阶段又在其上做过增改的需求的解释。客户可能要同软件供应商签订某种类型的维护支持协议。客户也可以自己维护，或者同第三方签订合同。具体的细节问题可能不同，但有一点是相同的，那就是移交阶段已经结束了。

16.5 业务案例的完成

详细描述阶段结束时，在业务标书中提到了移交阶段的花费和进度安排。到移交阶段结束时，已经获得了判断业务案例价值所需要的进一步信息。

16.5.1 控制进度

项目经理把实际进度与为阶段所规划的进度、人力和费用进行比较。

在移交阶段结束时，项目预算也结束了，项目经理会召集人员回顾：就整个工程而言，与计划的数字相关的实际时间进度、个人工时、费用、缺陷率以及公司可能采用的其他的一些度量，以达如下目的：

- 看一看项目是否达到了计划的目标。
- 确定为什么没有达到计划目标的原因（如果是这样的话）。
- 把项目的度量数据添加到公司的度量数据库中，以备将来使用。

16.5.2 业务计划的审查

这个计划预示了该项目是否会取得经济上的成功。同样地，有两种模式：基于合同的产品开发和基于市场的产品开发。在第一种情况下，很容易定义成功的概念：即合同的定价是否大于项目实施的费用？当然，这个相对简单的问题可能由于相关的考虑而变得很复杂，如公司是否能够成功地开拓一个新的业务领域或者一些由这个合同所产生的构件和子系统能否在将来的合同中减少费用。

在第二种情况下，业务计划就更为复杂。项目成功与否以产品是否达到公司基于在开发过程中所投入的资金而确定的最低收益率来衡量。在移交阶段和项目完成的后期，所有的数据，例如未来的销售水平还不确定。但与项目开发初期相比，此时公司对它所付出的代价有较好的了解，对未来的发展前景有更好的认识。负责人安排对所有可用的数据进行收集整理，并召集一个小组对这些数据进行审查。

16.6 评估移交阶段

项目经理召集一个小组对移交阶段进行评估，并从整体上对开发周期进行事后分析。这次评估在两个方面不同于前面几个阶段完成时所做的评估。首先，移交阶段是最后一个阶段，不存在将尚未完成的工作传递给后续阶段的问题。然而，在有一定规模的系统中，会存在一个产品组，有用的思想可以传递给该组。其次，尽管这是当前开发周期中的最后一个阶段，但以后还会有开发周期，评估小组应当把这次评估中的有用的发现记录下来，以备后续开发之用。

这些发现可被归纳为两类，以下小节将进行具体讨论。

16.6.1 对迭代和阶段的评估

一方面，如果项目开发组织有效地实施了前三个阶段，则移交阶段就能顺利进行，并按计划和预算圆满地完成任务。beta测试只能发现项目组成员可以很容易地纠正的常规错误。评估小组要找出一些对产品组或者下一个生命周期要处理的项目是很有价值的注意事项。

另一方面，如果项目组没有识别出所有的重要风险，没有设计出满足要求的构架，或者没有实现能提供所需系统的设计，那么这些不足会在移交阶段变得十分令人头疼。这些不足所产生的后果是，项目经理可能不得不延长移交阶段的时间（而即使这样做也只能实现一个仅能满足最低要求的系统），可能不得不费尽周折以期将“足够好”的系统推入用户市场，可能不得不将需求说明中最初所确定的一些特征推迟到以后的发行版本。

这些不足当然就成为最终评估小组的素材。他们实际上是对整个项目——包括所有的这四个阶段——进行评估。他们将记录一些不太令人满意的工作经验。从过程的观点来看，评估的目的是使项目在下一个发行版本中进行得更好。特别是，这样的行为不应成为负面个人行为的基础。这些行为应该基于其他的证据。如果评估小组感觉他们的发现可能被误用，那么他们就会尽力降低调子。获得充分的事实对组织未来的成功是很重要的。

16.6.2 项目的事后分析

与评估相反，事后分析主要是致力于分析项目组织正确地做了哪些事情，错误地做了哪些事情。目的是为更有效地组织后续项目和更成功地执行开发过程提供记录；例如：

- 应记录与所开发系统相关的要点，以备维护工作者和下一版本开发人员使用。例如，常常要记录选择目前正在使用的这一设计的原因，而拒绝其他可能的设计的原因对于以后的开发人员来说也是有用的。
- 应当全面地考虑与开发过程本身相关的要点。
- 需要进一步的综合培训吗？
- 什么领域需要专门培训？
- 专门的辅导是否需要继续？
- 本项目使用统一过程（参见第2章）中某些特定的方面所获取的经验为未来的项目提供了有益的见解吗？

16.7 制定下一版本或升级版本开发计划

几十年来软件的工业生产经验表明，几乎没有长期不变的软件。支撑软件产品的硬件和操作系统在不断地发展，业务和政府环境在改变，越来越多的商业、工业、政府部门和个人领域卷入软件领域。当上述因素改变时，几乎全部的软件系统就会立即进入一个新的开发周期。这个新的周期将重复初始、细化、构造和移交等阶段。

16.8 关键的可交付内容

本阶段的可交付内容与构造阶段所罗列的内容十分相似（参见第15章），但是，此时它们已经被修正，而且完成了如下的交付内容：

- 可执行软件本身，包括安装软件。
- 合同、许可证、弃权声明书和保证书等法律文档。
- 完成的和改正过的产品发布基线，包括系统的所有模型。
- 完成的和更新过的构架描述。
- 最终用户、操作员和系统管理人员使用的手册和培训材料。
- 客户支持链接和WEB链接，从中可以找到更多的信息，如报告缺陷，找到修复和升级的信息等。

第17章 统一过程的运用

正如本书前述各章节所阐明，开发具有重大意义的软件系统的工作非常复杂。在最后一章，作为回顾，我们再一次把众多相互缠绕的概念集中到一起。我们的意图是希望可以比前述章节更加简明地告诉大家，这些概念是如何联系并一起发挥作用的。

另外，我们把本章的内容拓展到两个新的领域。首先，如果软件过程本身很复杂，那么相应地这个过程的管理也很复杂；其次，从没有过程或者非常原始的过程转变为统一过程决非一件容易的事情。

17.1 统一过程帮助你解决复杂性问题

首先，统一过程有初始、细化、构造和移交等四个阶段。与最初的开发周期的阶段不同，由于软件系统的不断发展，出现了新的版本；而由于大的修订，又出现了新的产品。在这些阶段中，包含着相互缠绕的工作流、构架、风险管理、迭代和增量等问题，例如：

- 用况驱动的软件开发，经过的工作流有：需求、分析、设计、实现和测试。
- 由构架引导的软件开发，其结构和行为元素的框架可以使系统很好地进化。
- 使用构造块和构件进行的软件开发，可以支持广泛的重用。
- 通过迭代和迭代内的构造进行的软件开发，形成产品的增量式增长。
- 风险控制下的软件开发。
- 可视化并用统一建模语言（UML）记录的软件开发。
- 在里程碑进行评估的软件开发。

定位过程的四个里程碑是：生命周期的目标、生命周期的构架、可初步运行的能力和产品发布。[1]

17.1.1 生命周期的目标

第一个里程碑通过提出如下问题，阐明产品生命周期的目标：

- 是否弄清楚了系统的范围？是否已经确定什么要在系统内而什么要在系统外？
- 是否与项目相关人员在系统的关键需求上取得了一致？
- 是否找到实现这些特征的构架？
- 是否识别了对于项目成功至关重要的风险？是否找到了缓解风险的办法？
- 产品在实际使用时能否产生应有的价值以证明对其投资是正确的？
- 开展这样的项目，对你们的组织而言是否可行？

- 项目相关人员是否认同这些目标?

回答以上问题是初始阶段要做的事情。

17.1.2 生命周期的构架

第二个里程碑通过提出以下问题，阐明产品生命周期的构架：

- 创建了可运行的构架基线吗？它是否具有弹性且很健壮？它是否能随着产品生命周期而进化?
- 是否已经标识风险并且已经将主要风险降低到保证不会扰乱项目计划的程度?
- 是否已经制定了一个对进度、费用和质量估计现实的项目计划?
- 项目能否像现在所计划和估价的那样有适当的投资回报?
- 是否得到了项目相关人员的赞同?

回答以上问题是细化阶段要做的事情。

17.1.3 可初步运行的能力

第三个里程碑确定产品已达到可初步运行的能力，即要确定是否已使产品达到了在用户环境中可初步运行的水平，尤其是是否达到了可作beta测试的水平?

这是第三个里程碑的一个关键问题。当接近该里程碑时，已有了构架基线，已研究过风险，并且已有了项目计划和相应的资源。因此，我们就在这里建造产品。如果已经确定了生命周期目标和生命周期构架，建造工作会进展得很顺利。这个阶段的一个重要任务是安排构造和迭代的顺序。一个好的顺序意味着每个新的迭代的先决条件已经到位了；一个好的顺序能避免走回头路，避免由于后来才知道的某些东西而对一个早期的迭代进行返工。

尽管努力避免这些问题的出现，但一些问题仍然会发生。在此所要做的就是要克服这些颇费时日的问题。建造系统是构造阶段要做的事情。

17.1.4 产品发布

第四个里程碑确定产品已准备好，可以自由地发布给用户，即要确定该产品是否能够在典型的用户环境中成功地运行?

达到可初步运行能力的要求之后还要做一些工作，即要进行beta测试和验收测试，以及纠正在工作环境中运行时所暴露出来的问题和缺陷。如果用户对产品感到满意，就发布这个产品。

这些任务是移交阶段要做的事情。

17.2 主题

经历上述这四个主要的里程碑并把它们联结起来是要讨论的主题，这涉及到诸如需求、构架、基于构件的开发、统一建模语言（UML）、迭代以及风险管理等主题。（所列主题并非以它

们的重要性为序，它们都非常重要，而且它们必须共同发挥作用。）

正确地获得需求

通过用况建模、分析、反馈机制等获得正确的需求。统一过程的最好开端是用况驱动。

正确地获得构架

所有大型项目都必须是以构架为中心的。构架将系统划分为许多部分，并使各部分协同运作。构架确定了各个部分之间的接口，使各开发组能够在接口的各自一边独立开展工作，并保证其开发工作的正确性。构架的监督使得项目能纳入正确的技术轨道。

使用构件

有很多因素使得以构件为基础的开发成为可能，构架的稳定的接口（标准组之间的标准接口也一样）就是其中之一。可重复使用的构造块减少了开发费用，使产品能够更快的投入市场，还可以提高质量。

用统一建模语言（UML）来进行思考和沟通

软件开发远不是编写代码。统一建模语言（UML）以及统一过程的其他一些特征使得软件开发成为一门工程学科。UML是一种图形语言，软件开发人员可以用它进行构思、可视化描述、分析、沟通和记录。

如果没有像UML这种标准的沟通方式，人们在理解别人或别的组在做什么时，就会产生很大的困难。在通过各个阶段，以及进行以后的发布版本和新一代产品的开发纵向地传递信息时，也会存在很大的困难。

迭代

迭代和构造带来了很多好处：小块的工作、小的开发组、与风险管理紧密相联，以及频繁的检查点和反馈点。

风险管理

识别风险——严重的、重要的、常规性的——将它们列在风险清单中，并在开发过程中，将它们在发生之前加以缓解。

17.3 通过管理引导向统一过程的转化

将一个公司或公共企业从一种老的方式转变为新的方式，远远超过了管理和控制一个正在进行的过程的任务。目前，商业和政府领域里有很多机构，几乎很少采用软件过程的方式。当然，它们开发软件，因此有自己的方法，但它们不是采用有组织的、可重复的方式。还有其他很多公司，它们有某种软件过程，但是也许是含糊的，不是很满意的。它们也开发软件，有时还取得了不小的成功，但是它们认为一定还有更好的方法。事实上，的确有更好的方法。这些公司如何向更好的方式迈进呢？

一个低级的专业人员并不能实现这一转变。一旦该组织决定转向一种新的方式，他就可能是该方式的挑战者。但这显然是软件工程部门的高层主管的职责。他有责任去领导这一转变，因为这将影响整个公司。他必须知道有更好的方式，并且明白那是一个应该拥有的好东西。他

必须认识到竞争使这件事变得更紧迫。也许他会从其组织已经步入软件过程领域的杰出的主管人员那里获得启发。

17.3.1 行动案例

由于新的方式将会应用到软件开发机构的其他部分甚至整个公司，所以首席主管要赢得其他主管人员的支持。他可以通过一个“行动案例陈述”来做到这一点。这种陈述要阐明现在所采用的方式并不够好，它费用太高，质量太低，最重要的是推向市场的时间太长。即使是在内部使用的软件，也不能像较好的软件那样很快地取得竞争优势。每年盈利5%已算不上好了。在成本、进度和质量这三个方面获得更大进展现在是可能的，其他公司已经证实了这一点。

对于行动案例的陈述引出了“行动”是什么的疑问。按通常的做法，软件开发机构并不去研究开发方法、工具、构造块和构件之类的事情。这些事情由其他机构来做。例如说，有的公司专门生产和出售构件。也有像SAP之类的机构来做预制系统，它能根据你的事务需要来加以调整（预制系统是一个很大的框架，在卖主的帮助之下，它做一些很小的改变就可以适应你的特殊需要）。有很多像Rational软件公司一样的公司，它们支持统一过程。软件主管的任务就是找到在哪里可以获得它们的公司所需要的帮助。

假设可供的选择不多，对大部分软件公司来讲，只有一条路可走，那就是以构件为基础的软件开发。无论是打算自己做大量的工作，像银行界、保险业、国防、电信以及其他的一些行业，还是打算寻找预制系统，在可以重复使用的构造块上进行系统建造，软件主管可根据自己的情况作出选择。在多数情况下，有一些或者有大量的软件是在内部开发的。此时有人必须去获取构造块，有人必须使这些构造块适宜于公司的需要。这些都要经历一个软件过程。在很多情况下，软件主管会发现统一过程可以解决他们的部分需要。

17.3.2 重建工程指南的宣传和贯彻

最先运用这一方法的主管详细地解释了为什么公司要转向一个改进的软件过程，在此并没有为重建工程指南做广告的意思。指南内容包括：

- 当前的业务情况及其正在发生变化的事实。
- 顾客现在想要什么。
- 公司面临的竞争。
- 公司面临的挑战。
- 这些挑战所带来的对自身的诊断。
- 不变革将引发的风险。
- 尤其是关于软件过程，公司必须做些什么 [2]。[⊖]

支持的保证

⊖ 在[2]中，介绍了当重建工程指南应用于重用时的内容，构成这种指导基础的心理学同样可以用于软件过程。

项目经理必须确信有持续的财力支持。除了要考虑其他事情之外，当一个项目经理检查新过程的初始应用时，该保证包括初步培训和指导以及转变需求时的持续培训。不要在缺少有经验的人的情况下试图转向新的过程。指导者必须知道如何去做，他们可以是外部人员或内部员工。运用新过程开始一个新项目完全依赖于四个方面的综合支持：过程、工具、培训和指导者。

现有项目的持续性

一个具有一定规模、又有许多项目正在进行的公司，当前的项目和近期要开始的项目必须沿用现在的过程。并非所有人都会立刻接受培训，并非所有项目都能够应付因较大的变化而产生的混乱。

同时，重建工程指南应该清楚，那些继续进行现有项目的经理和工作人员不能被抛在一边。他们将会轮流接受培训并被指派做统一过程下的项目。这一转变需要时间。公司必须继续现有的业务，以便为将来筹措所需的经费。公司培训其他人，这样他们就可以参加对以后事情的讨论，他们不能把自己视为局外人，否则，会严重破坏转变工作。

对未来要有自信心

参与这一转变的人都是软件方面的专业人员。该领域自诞生以来已经发生了巨大变化，而历史却很短。有些人已经有些跟不上了。专心致力于管理的中层管理更是感到了这一压力。如果他们能对自己的未来充满信心，将有利于这一转变。具有照顾自己历史传统的公司在这方面有优势，但应留心所有事情的重要性。

17.3.3 实现转变

软件主管面临着实现问题。

支持者

第一个问题是，如何通过一种组织形式来实现这一转变。既然软件主管人员很可能已经全力投入，就需要一个技术过硬的工程师来支持这日复一日的变化，即指导这一彻底的转变。支持者需要了解统一过程，要获得这种了解，就要接受一些培训和直接的咨询。

支持者也许是一个技术上很胜任的项目经理，也许只是一个有项目管理经验的构架设计师。他已经学习过统一过程，并相信它会帮助公司，还愿意为此去克服重重困难；同时，他还信任发起这一转变的主管和那些参与人员，知道如何与主管和技术人员一起工作，是一个擅长信任别人的人；他通常对方法和过程充满兴趣，但同时，又可以相当熟练地用他自己独特的方法而不必一切重来。更确切地讲，支持者使统一过程适应第一个项目的特殊需要。

支持者无需具有曾在一个统一过程中贯穿始终的经验。事实上，自他在该组织中工作一段时间以来，他可能还不会拥有这样的机会。他只要知道这样做的含义是什么，只要乐于支持并参与即可——当然，要有项目经理和指导者的帮助，这在下面要讨论。

第一个项目的经理

除了支持者（顺便说一句，如果无法找到所描述的特殊人才，那么就从现有的人手中找到一个最好的，并给他一切支持），负责的软件经理还需要一个有特殊才能的项目经理。他需要一个彻底认为采用新的过程并将其贯穿始终是非常重要的人。面对采用新过程时可能产生的混乱，

项目仍要顺利地开展。

指导者

培训不是一切，尤其是对采用新方法还没有什么把握。因此，支持者和项目经理需要指导者（内部的或外部的），他必须已经历过采用统一过程的项目，有这方面的经验。指导者不需要管理经验，尽管有管理经验也不错。他需要有两方面的特殊才能：一是能够预见到项目中的问题，该能力当然是建立在其以前曾经参加过采用统一过程的项目的基础之上；二是必须能够与其将遇到的各种人员很好地合作，尤其是支持者，还有项目经理、项目有关人员以及发起该工作的主管。

从何下手

第一个项目必须是一个结果有价值的真正项目。我们的经验表明，假拟的演练项目并不值得提倡，它们倾向于成为一潭死水，不管其结果如何，都不会给人以深刻印象。第一个项目应该是任务重要、但在进度安排上并不过分苛刻的项目。你也许会说，所有重要的项目总是要求很短的时间进度！

事实上，在统一过程下开展工作比其他老方式下要快得多。我们很早就开始寻找风险，很早就设计构架，因此，建造过程实际上进行得非常快。此外，第一个项目会有指导者来指导。他们之所以称为指导者，是因为他们曾经参加过整个统一过程，他们能正确地推动开发过程。

为一个真正的项目寻找一个全面的方法事实上是比较安全的，虽然这并不意味着同时会有太多的新东西要做。过程当然是新东西。与过程很好集成的工具当然也是新东西。但是，一种新的操作系统、一种新的数据库技术、一种新的程序设计语言以及一种新的分布式系统平台都不是新东西。不要在同一时间堆积过多的新东西，尤其是不要把那些与其他技术接合不太紧密的技术放在一起。根据情况，你可以采用一种或多种新技术，但不要在一开始就承担一个大系统。

进一步的考虑

经验告诉我们一些更多的事情：

- 并非总是要按照这里所介绍的方法去做，但它确实是采用统一过程的一种系统的途径。
- 一个非常渐进的方法可能会陷入相反的陷阱。因为进展几乎是不可见的，所以会逐渐失去支持，导致转变过程失败。
- 有时候，人们用一些支离破碎的方法也能够在软件过程的再工程中取得最终的成功，这是事实，但是你不能依靠它，它耗时长，而且经常失败。
- 无论如何，在有效的管理控制之下完成转变并留心它的全过程是最好的。因缺乏控制而造成的失败是很难修复的。

17.4 统一过程专题

正如本书所描述的，统一过程并非是该主题的唯一词汇。事实上，有两件非常重要的事情

需要说明。首先，它是一个框架，要加以裁剪以适应大量的变数：开发系统的规模、系统发挥作用的领域、系统的复杂性以及项目组织及其人员的经验、技能或者过程水平。第二点，在你看来这本书已叙述得很详细了，但实际上只是对工作过程的一个总的看法。实际上，为了应用它，你需要大量更进一步的信息。在下两个小节将讨论统一过程的这两个专题。

17.4.1 裁剪过程

软件系统、产品以及建造它们的组织都在不断地变化。因此，统一过程中存在的某些常量，例如说四个阶段和五个工作流，也有许多变化的因素。例如，这些阶段相对地应该有多长？在不同的环境下各个阶段迭代多少次是合适的？候选构架（或者严重风险的缓解、构架基线、业务案例等）可以在哪一点完全建立起来？

类似这样的问题的答案取决于系统的规模、应用的性质、项目组织的领域经验、系统的复杂性、项目组的经验以及软件管理的能力水平，甚至包括项目相关人员合作的有效程度。

举一个例子，如果一个系统相当小，项目组在这方面又很有经验（曾经做过早期的版本或者是相似的产品），则初始阶段会很短。开发组（也许项目相关人员也是这样）知道在开发过程中没有很严重的风险，知道一个先前使用过的构架可以重新利用。结果是只要花几天就确定了系统的范围，并确保在这一范围内不会出现新的风险，使得初始阶段顺利完成。

另外举一个例子，如果系统庞大而复杂，并且对项目组来讲是全新的，可以预料初始阶段将和细化阶段一样，会非常漫长且会经历更多的迭代。

这些主题的不同之处的数量跟要建的系统的数量一样多。那是经验的来源，从那里，项目经理非常明智地吸收了其他软件人员的工作经验以及其他项目相关人员的知识。总之，可以根据你的具体环境来“裁剪过程”[3]。

17.4.2 填充过程框架

本书中所描述的统一过程只是对你所需要的用于管理一组人协调一致地开展工作的过程的一个概述。本书的目的在于使实践者和管理者对这个过程有所了解，而并不能提供给开发组成员用于指导他们日常工作的详细的方针。它为该过程所产生或使用的一些制品命名，还有一些别的制品没有命名。它并不提供文档模板，也不识别所有不同类型的员工（因为太多）。它一次次地提及这样的事实，那就是工具在阶段实施过程中可以发挥巨大的作用。实际上，大量常规性的工作可以由这些工具来完成，这将比没有工具支持的人要完成得更快、更准确。然而本书没有详细地说明这些工具是什么，用来干什么以及如何使用。实质上，本书包含了基本的思想，它描述了一些工作流、一些制品、一些工作人员及他们的一些活动、以及统一建模语言（UML）的应用等。

还远不止这些。至于更高级的版本，需要看一看RUP（Rational Unified Process）。该过程产品是可以在线查找的知识库，总共大约有1800页的资料。它通过给每个组员提供生产效率提高指南、模板和用于重要开发的辅助工具，来提高开发组的生产效率。该过程和Rational公司的工具相结合。事实上，工具和过程一起开发，其内容也不断更新。该过程还由一个全面的教程集

支持，是指导者提供个人指导的基础。

RUP（Rational Unified Process）是存在已久的ROP (Rational Objectory Process)的进一步发展。如果你使用过后者，就可以很轻松地升级到新的RUP。

17.5 联系更广泛的社团

除了能够在项目内部提供更有效的工作环境，以及提供与用户和项目相关人员之间更有效的联系之外，统一过程还为更为广泛的社团提供更令人满意的接口：

- 教育社团　教育者和培训者能够将他们的课程集中于学生按统一过程进行工作时所需要知道的东西，以及他们用统一建模语言进行构思和可视化描述时所需要的东西。
- 软件社团　构架设计师、开发人员以及其他人员能在项目甚至公司之间转换，而不需要花很长的时间进行调整以适应新公司的独特习惯。
- 重用社团　项目可以很容易地重用子系统和构件，因为它们是用统一建模语言描述的。
- 工具社团　项目要有更有效的工具支持，因为统一过程的巨大市场为工具开发提供了更好的经费支持。

17.6 采用统一过程的好处

通过用况的进化以及用户和过程工程人员的一致努力，我们在早期就确立了稳定的需求；因为统一过程是**用况驱动的**（附录 C），所以能更有效地从需求转到后续的工作流（分析、设计等）。

固定项目所要做的工作；在项目进行过程中加以指导；并为未来产品的升级提供指导。因为统一过程是以构架为中心的，所以所有这些都通过一个生命周期构架来实施，而且该构架是可以理解的、有弹性的、健壮的、在未来的岁月里是可以进化的。

因为统一过程是**迭代的**（附录 C）和增量式的，所以开发人员和所有其他项目相关人员都可以从成功的构造和迭代实践中学习。

因为统一过程是**风险驱动的**（附录 C），所以要减小那些可能对项目的成功造成危害或者对预算或进度造成危害的严重风险。

因为统一过程是基于构件的，所以，通过重用构造块，可以加快开发过程的速度，减少成本，提高产品质量。

因为统一过程不只是对单个开发人员的指导，而是一个工程化的过程，所以它能使项目组的每个成员（包括所有项目相关人员）协调一致地进行工作。

拥有了阶段、迭代、里程碑以及评估的过程框架后，就为项目相关人员提供了许多可以了解开发过程并找出正在开展的工作的切入点。知道正在进行什么，并通过他们的应用常识所获知的信息，项目相关人员能对系统的改进提出反馈建议。最为重要的是，由于早期阶段考虑了像构架和风险这样的关键方面，所以可以在项目还有时间可充分利用时输入他

们的想法。

在30年实践工作的基础上，统一过程集中了几个思想界领袖和若干有经验的组织的经验。它是许多应用和技术的统一，例如：

- “可视化的” 统一过程中使用的可视化模型和制品是用统一建模语言描述的，这带来很多的优点，例如广泛的软件重用和软件蓝图。
- “工具化的” 事实上统一过程和标准语言为更广泛的工具研制提供了财力上的支持，反之，又使得过程更加有效。
- 可裁剪的 它是一个过程框架，而不是严格的过程。针对不同应用领域和组织的需要可以作一些专门的处理。
- 可扩充的 在开发活动中，统一过程并不把用户限定在一条路上，例如，对于风险分析，用户可以采用其他的信息来源加以指导。统一过程所做的是提供风险分析在过程中的逻辑位置，这种情况下，风险分析基本上会“及早”而不是“过晚”发生。

统一过程从很多方面强化了软件开发组织。最重要的是它提供了项目组可以协同工作的途径。而且，它还提供了项目组与用户及其他项目相关人员一起工作的途径。

17.7 参考资料

[1] Barry Boehm, “Anchoring the software process,” *IEEE Software*, July 1996, pp. 73–82.

[2] Ivar Jacobson, Martin Griss, Patrik Jonsson, *Software Reuse: Architecture, Process and Organization for Business Success*, Reading, MA: Addison-Wesley, 1997.

[3] Walker Royce, *Software Project Management: A Unified Framework*, Reading, MA: Addison-Wesley, 1998.

附录A
UML综述

A.1 引言

软件工程的一些古老的分支很久以前就发现用画图的方式来表达设计很有效。在软件出现的早期，编程人员就把他们的概念用各种各样的图，或者更进一步地用模型的方式具体化了。软件开发社团需要一种用模型进行沟通的方法，这种方法不仅用于项目组内部成员之间，还要用于与外部的项目相关人员以及未来几代的开发人员进行沟通。这需要一种语言，不仅是为了方便相互之间的沟通，还要提供一个框架，供每个开发人员在其中进行独自的思考和分析。此外，他们不用把所有的东西长年累月地保留在自己的脑袋里，而只需把所有的东西记录在纸上或者进行电子化记录。统一建模语言（UML）是一种标准的软件建模语言，它是一种用于对一个软件系统的制品进行可视化描述、详细描述、构造以及文档化的语言。最主要的是，UML使得开发者能够用标准化的蓝图或者图表的方式对他们所加工的产品进行可视化描述。例如，用于捕捉需求的特征化的符号或图标是：一个椭圆表示一个用况，而一个稻草人表示使用该用况的一个用户。类似地，设计中用到的主要图标是一个表示类的矩形框。这些图标不过是一种图符，说得更精确些，是一种语法。在A.2节中，我们会给出UML图符的概览。查阅参考资料[2]可获取更为广泛的信息，查阅参考资料[3]可获取用户指南。然而，在这些图符的背后，UML给其指定了特定的含义，也就是语义。我们在A.3节中通过对UML语言的主要术语提供简单定义，给出了这些语义的一个简要综述。对于UML语义的论述需要查阅参考资料[2]和[3]。另一个阐述UML表示法和语义的参考资料是有关UML的一个公共的文集（参考资料[1]），虽然它本质上太形式化。其他两本有关UML的一般性资料是[4]和[5]。

A.1.1 词汇

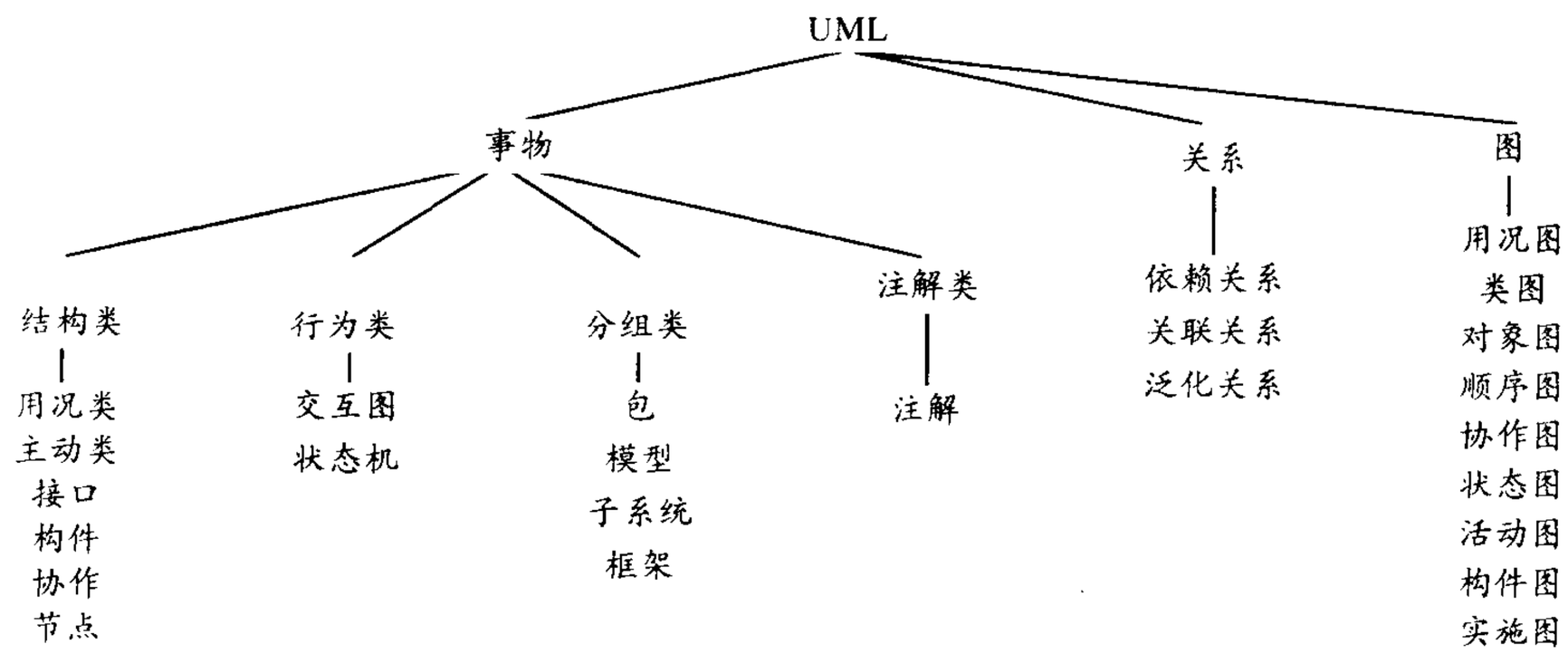
UML给开发者提供一些词汇，包括三种范畴：事物、关系和图。我们在这里提及它们，仅仅是为了给你们一点有关该语言基本结构的感觉。

事物共有四类：结构类、行为类、分组的和注解类。其中，结构类的事物主要有七种：用况、类、主动类、接口、构件、协作以及节点；行为类的事物有两种：交互与状态机；分组类的事物有四种：包、模型、子系统和框架；注解类的事物主要就是注解。

在第二个范畴即关系中，我们发现三种进一步的分类：依赖关系、关联关系和泛化关系。

在第三个范畴即图中，UML提供了九类图：用况图、类图、对象图、顺序图、协作图、状态图、活动图、构件图和实施图。

UML词汇如图A-1所示。



图A-1 UML词汇

A.1.2 扩展机制

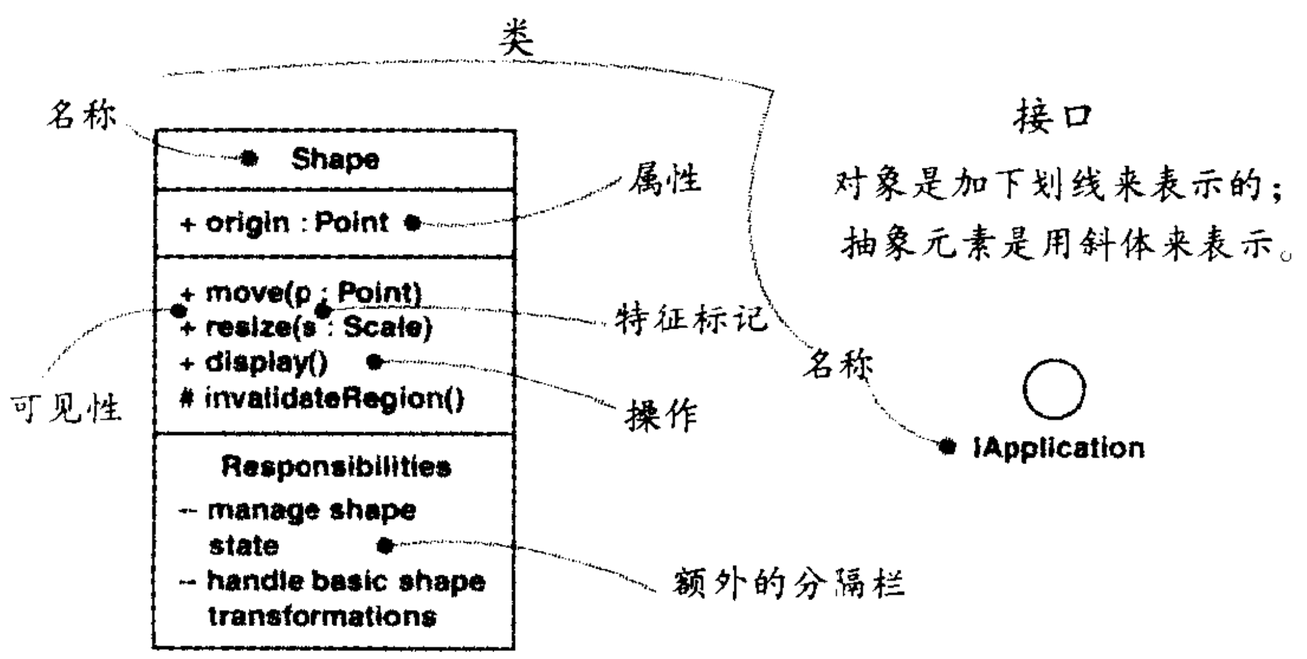
UML提供扩展机制（*extensibility mechanisms*），允许其用户精练它的语法和语义。UML由此可以在必要的时候加以裁剪，以适合特定的系统、项目或者开发过程。关于裁剪方面的例子，请参阅附录B。

扩展机制包括构造型、标记值以及约束。构造型通过扩展或者精练已存在元素（如事物、关系等，参阅A.1.1节）的语义的方式来提供一种定义新元素的方法。而标记值提供了一种定义已存在元素的新特性的方法。最后，约束提供了一些对元素及其特性的强制性规则（例如一致性规则或业务规则）。

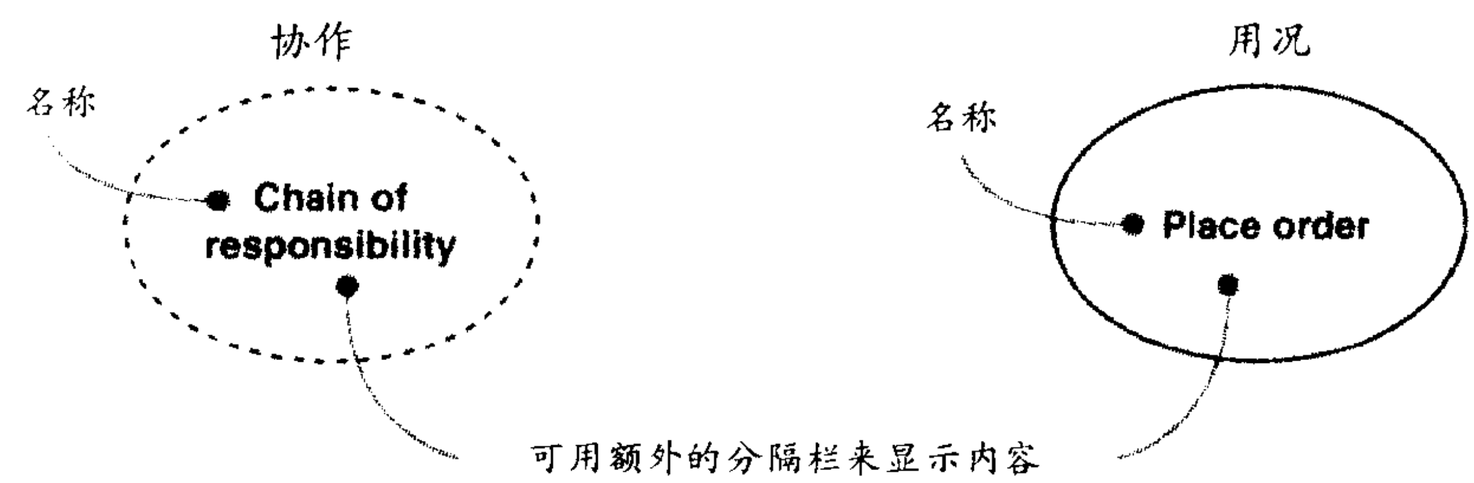
A.2 图符

本附录余下的部分摘自于Grady Booch、James Rumbaugh和Ivar Jacobson所著的《UML用户指南》一书[3]。

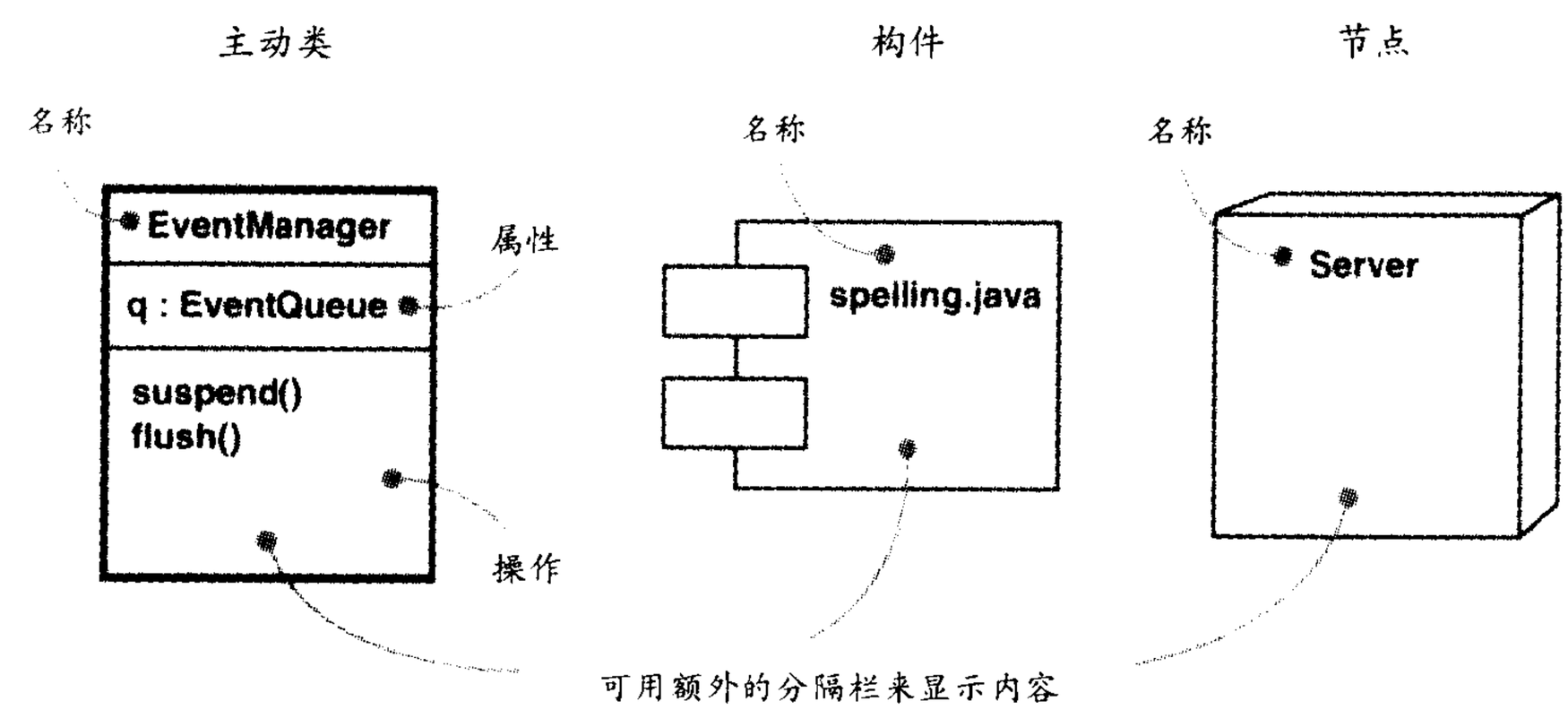
A.2.1 结构类的事物



图A-2 类和接口

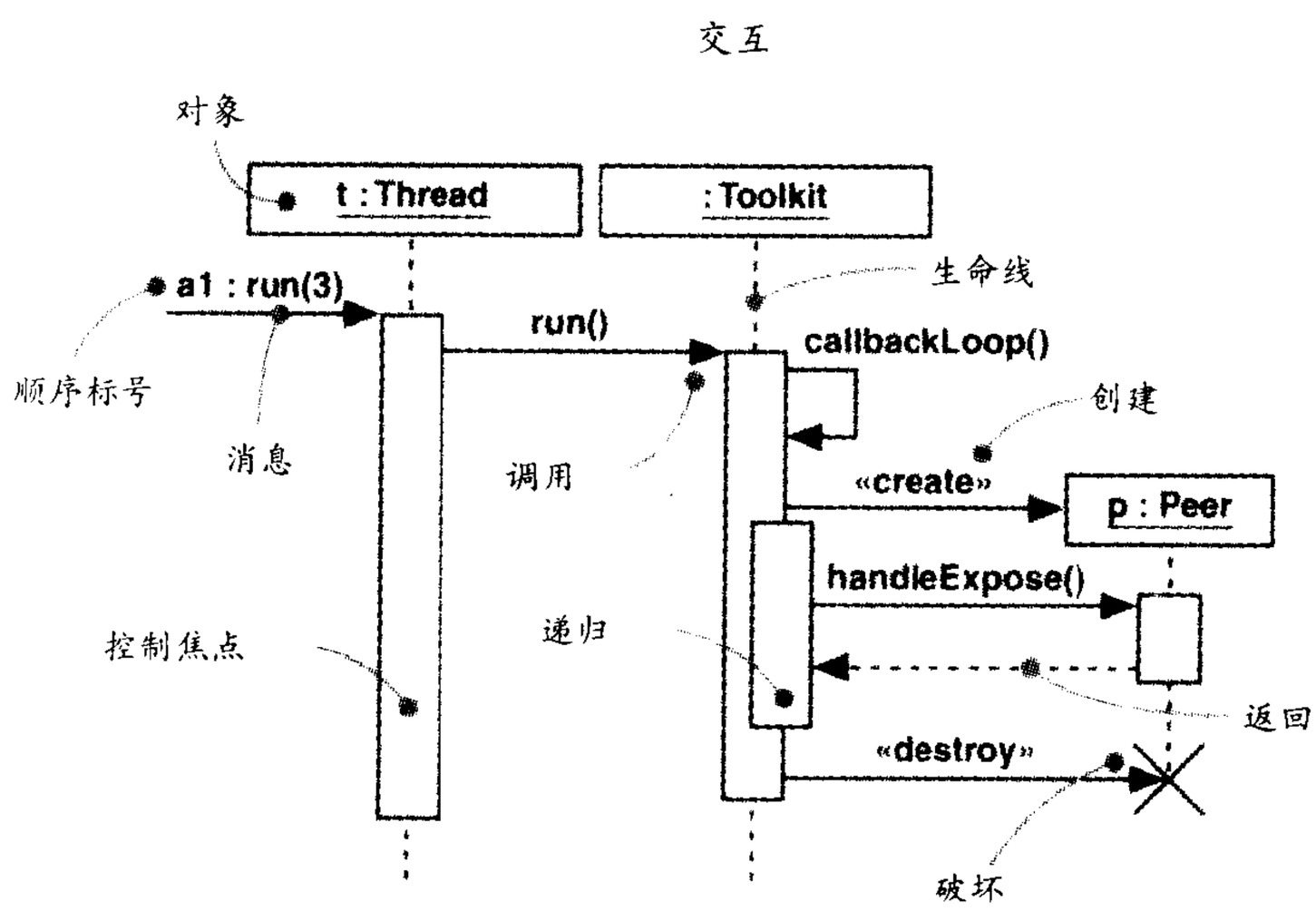


图A-3 用况和协作。（注意，举例来说，如果适当，可以将用况的名称放在符号的外面。）

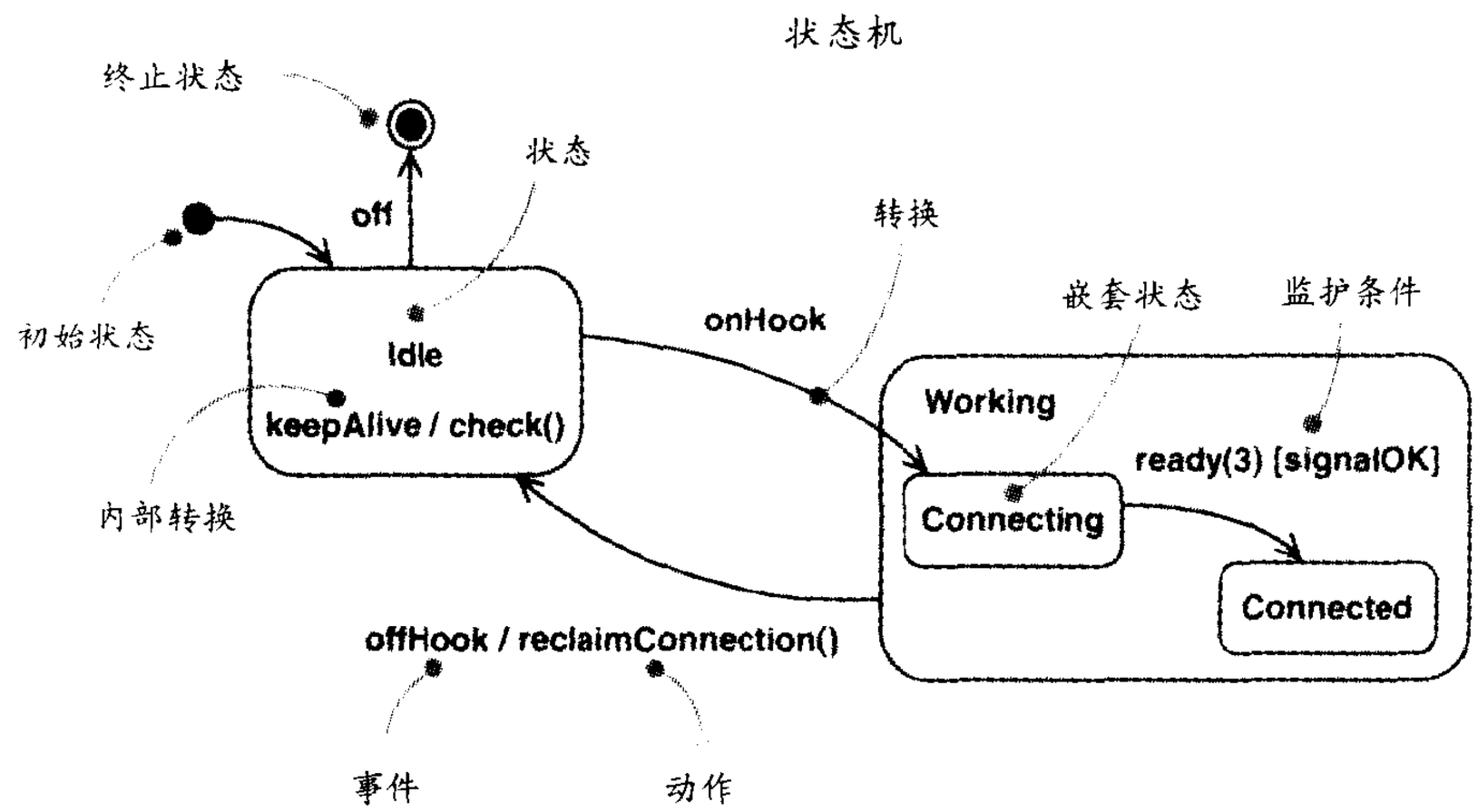


图A-4 主动类、构件和节点

A.2.2 行为类的事物

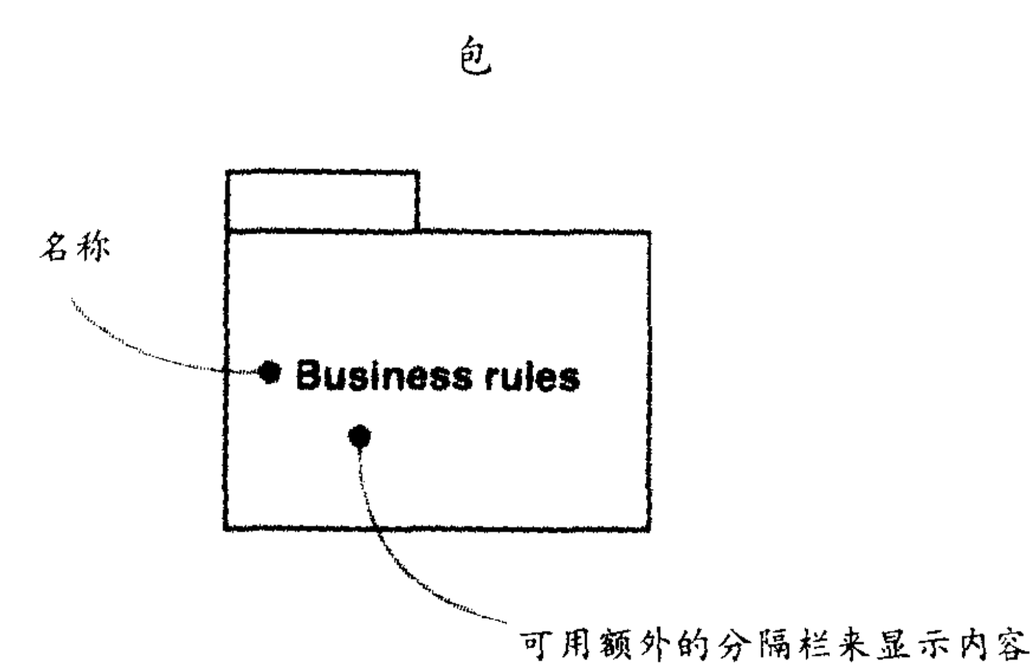


图A-5 交互



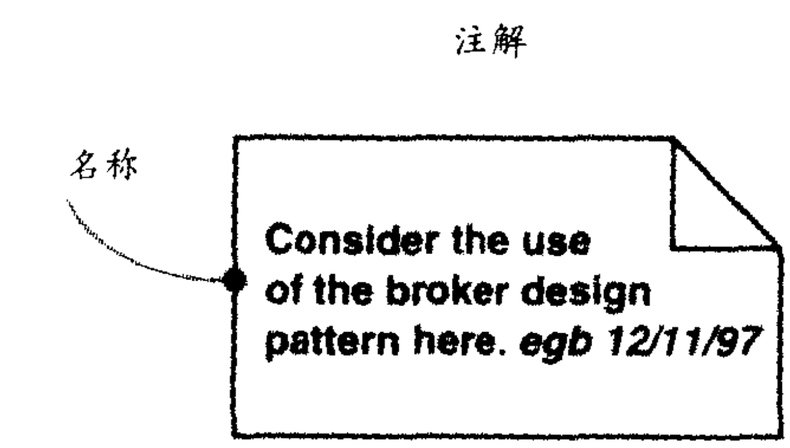
图A-6 状态机

A.2.3 分组类的事物



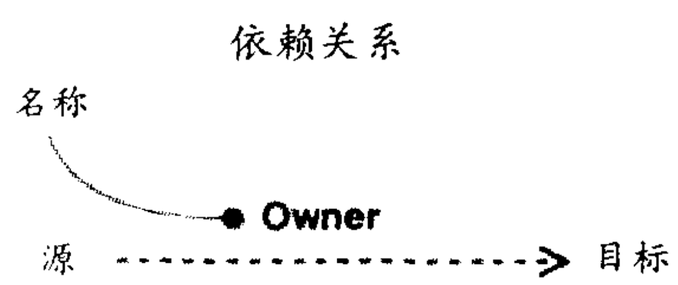
图A-7 包

A.2.4 注解类的事物



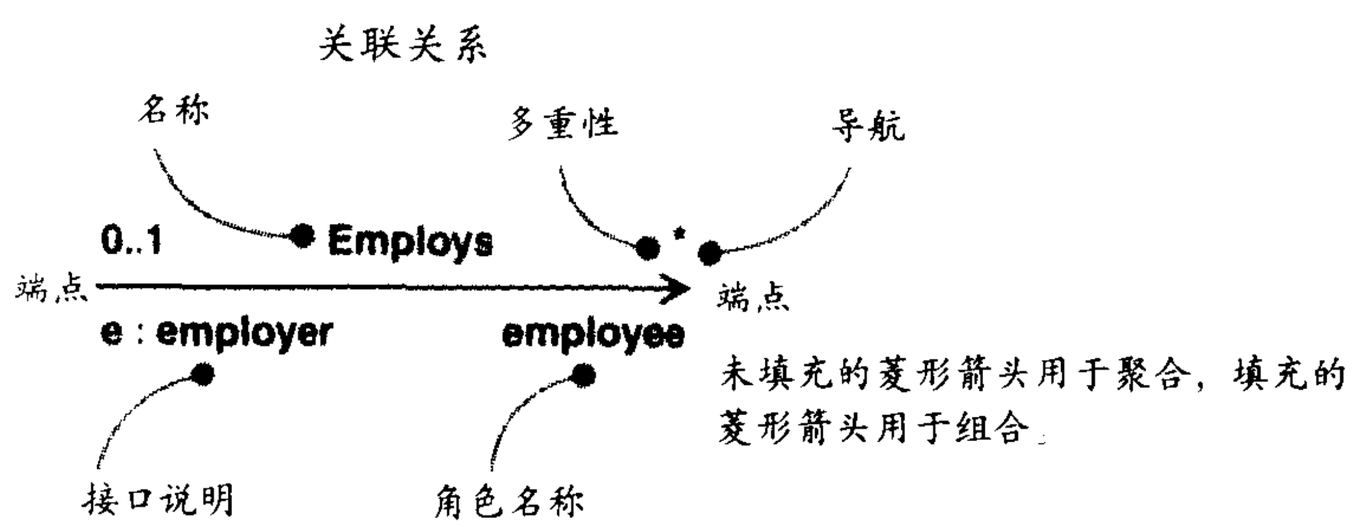
图A-8 注解

A.2.5 依赖关系



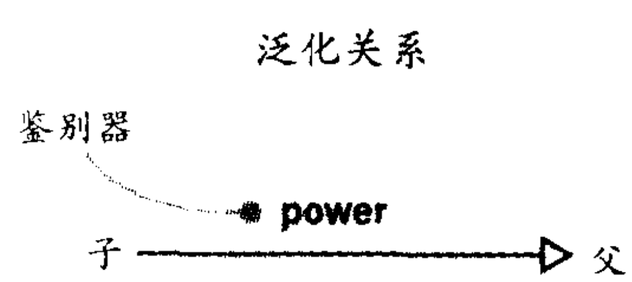
图A-9 依赖关系

A.2.6 关联关系



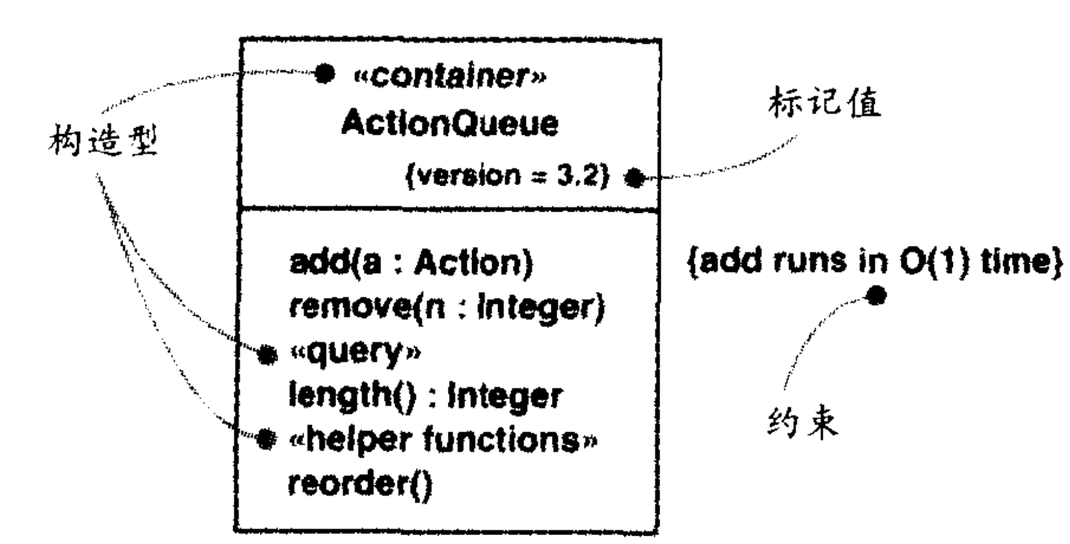
图A-10 关联关系

A.2.7 泛化关系



图A-11 泛化关系

A.2.8 扩展机制



图A-12 扩展机制

A.3 术语表

abstract class（**抽象类**）不能直接实例化的类。

action（**动作**）一条可执行语句的规格说明；一条可执行语句是一个计算过程的抽象。通过向一个对象发送消息或修改某一属性的值来执行一个动作，并引起状态的改变。

action state（**动作状态**）表示执行一个原子动作的状态，典型示例如对一个操作的调用。

activation（**激活**）一个动作的执行。

active class（**主动类**）其实例为主动对象的类。参见进程（*process*）、任务（*task*）、线程（*thread*）。

active object（**主动对象**）拥有一个进程或线程并能对控制活动初始化的对象。

activity（**活动**）展示某种行为的状态。[⊖]

activity diagram（**活动图**）一种描述从活动到活动的流（活动流）的图；活动图描述系统的动态视图。可以将活动图看作状态图的一个特例；在这种状态图中，大部分状态或全部状态是动作状态，并且大部分或全部状态变化是由源状态中动作的完成来触发的。

actor（**参与者**）用况的使用者在与用况发生交互作用时所扮演的角色。

adornment（**修饰**）根据元素的规格说明而加入到基本图形表示方法中的细节。

aggregate（**聚集**）在聚合关系中表示整体的类。

aggregation（**聚合**）关联的一种特殊形式，用以说明聚集（整体）和一个构件部分（部分）之间的整体-部分关系。

association（**关联**）用来描述一组表达对象之间连接的链的结构关系；同时也可表示包含实例间连接的两个或多个类元之间的语义关系。

association class（**关联类**）具有关联特性和类特性的建模元素。关联类既可看作具有类特性的关联，又可看作具有关联特性的类。

association end（**关联端点**）一个关联的端点，它将关联连接到类元。

asynchronous action（**异步动作**）发送对象无需停下来等待结果的请求。

attribute（**属性**）类元的命名特性，它描述了特性实例的取值范围。

binary association（**二元关联**）两个类之间的关联。

binding（**绑定**）通过为模板参数设定参数值而从模板创建元素。

cardinality（**基数**）集合中元素的数目。

class（**类**）对一组具有相同属性、操作、关系和语义的对象的描述。

class diagram（**类图**）描述一组类、接口、协作及其关系的图；类图论述系统的静态设计视图；给出说明（静态）元素集合的图。

classifier（**类元**）[⊜] 描述结构和行为特征的机制。类元包括接口、类、数据类型、构件和

⊖ 在《UML用户指南》（Grady Booch、James Rumbaugh、Ivar Jacobson合著）一书中，作者将活动定义为“一个状态机中正在行进的非原子执行”。由此可见，动作是不能中断的原子执行，活动是可以中断的非原子执行，两者都展示了某种行为。——译者注

⊜ 接口类、数据类型、构件和节点都可被看作是广义的类，因此将“classifier”译成类元。——译者注

节点。

client（**客户**）向其他类元请求服务的类元。

collaboration（**协作**）多个类、接口和其他元素协同工作的群体，它所提供的协同行为要比所有这些元素单独提供的行为之和更强大；一个元素（例如一个用况或一个操作）如何由一组按照特定方式充当特定角色的类元和关联来实现的规格说明。

collaboration diagram（**协作图**）强调发送和接收消息的对象结构组织的交互图；显示由实例及其相互间的链构成的交互作用的图。

comment（**注释**）对单个元素或元素集合的注解。

component（**构件**）系统中一个物理上可替代的部分，它符合并提供一组接口的实现。

component diagram（**构件图**）显示一组构件及其关系的图；构件图展示了系统的静态构件视图。

composite（**组合体**）通过组合关系与一个或多个类相关联的类。

composition（**组合**）整体与部分间具有很强从属关系和一致生命周期的聚合的一种形式；具有非固定多重性的部分可在组合体建立后创建，部分一经创建便与组合体共存；这些部分也可在组合体消亡前被显式地移去。

concrete class（**具体类**）可以直接实例化的类。

concurrency（**并发**）两个或多个活动在同一时间间隔内发生。并发可以通过交叠执行或同时执行两个或多个线程来实现。

constraint（**约束**）对UML元素语义的扩展，允许增加新的规则或修改现存的规则。

container（**容器**）包含其他对象的对象，并提供对其内容进行存取或迭代的操作。

containment hierarchy(**包容层次**)由元素及元素间存在的包容关系组成的命名空间的层次关系。

context（**语境**）基于特定目的的相关元素的集合，如详细说明一种操作。

datatype（**数据类型**）一种取值彼此不同的类型。数据类型包括基本的内建类型（如数字型和字符串型）和枚举类型（如布尔型）。

delegation(**委派**)一个对象在对某个消息做出应答时向另一个对象发送消息的能力。

dependency（**依赖**）两个事件之间的语义关系，其中一个事件（独立事件）的变化会影响另一个事件（非独立事件）的语义。

deployment diagram（**实施图**）一种描述一组节点及其相互关系的图。实施图论述系统的静态实施视图。

diagram（**图**）元素集合的图形表示，通常表达为顶点（代表事物）和弧（代表关系）的连通图。

distribution unit（**分布单元**）作为分配给一个任务或一个处理器的一组对象或一组构件的集合。

element（**元素**）一个模型的原子成分。

event（**事件**）对可用时间和空间来定位的某一重要事情的规格说明；在状态机的语境中，一个能触发状态转换的激励就是一个事件。

executable（**可执行体**）可以在某个节点上运行的程序。

export（**示出**）在包的语境中，使元素从它的封闭命名空间对外部可见。

extensibility mechanism（**扩展机制**）以受控的方法来扩展UML的三种机制（构造型、标记值和约束）之一。

facade（**虚包**）虚包是一个构造型包，它只包含对属于其他包的模型元素的引用；它用于提供一个包中某些内容的“公共”视图。

fire（**激发**）执行一个状态转换。

focus of control（**控制期**）顺序图中的一种符号，用于表示对象直接执行或通过一个附属操作执行某个动作的时间段。

framework（**框架**）一种为特定领域内的应用提供可扩展模板的构架模式。

generalization（**泛化**）一种特化/泛化关系，即特化元素（子类型）的对象可以替代泛化元素（超类型）的对象。

guard condition（**监护条件**）为使相关联的转换激发而必须满足的条件。

import（**引入**）在包的语境中，表示某个包中的类可以被指定的包（包括递归嵌入的包）引用的依赖关系。

inheritance（**继承**）将特殊元素与一般元素的结构和行为相结合的机制。

instance（**实例**）抽象的具体表示；可以在其上施加一组操作并具有存储该操作效果的状态的一个实体；对象的同义词。

interaction（**交互**）在特定的语境中，为实现特定目标，一组对象之间相互交换各种消息的行为。

interaction diagram（**交互图**）一种表示交互的图，由一组对象及其关系（包括对象之间发送的消息）组成；交互图展示了系统的动态视图；强调对象间交互的几类图（包括协作图、顺序图和活动图）的通用术语。

interface（**接口**）操作的集合，用于规定一个类或一个构件所提供的服务。

interface inheritance（**接口继承**）对一个更为特殊元素接口的继承，不包括对实现的继承。

lifeline（**生命线**）参见对象生命线（*object lifeline*）。

link（**链**）对象间的语义关系；是关联的一个实例。

link end（**链端点**）关联端点的实例。

location（**位置**）构件在节点上的布局。

message（**消息**）对象之间通信的规格说明，用来传送活动引发的预期结果的信息；消息实例的接收通常看作一个事件实例。

metaclass（**元类**）其实例仍是类的类。

method（**方法**）操作的实现。

model（**模型**）一个系统在语义上封闭的抽象。

multiple classification（**多重分类**）泛化的一种语义变体，此时，一个对象可以直接属于多个类。

multiple inheritance（**多重继承**）泛化的一种语义变体，其中一个类型可以有多个超类型。

multiplicity（**多重性**）对一个集合可以假定的允许基数范围的规格说明。

n-ary associationn（**n元关联**）n个类之间的关联。当n=2时，该关联是二元的。参见二元关联（*binary association*）。

name（**名称**）对事物、关系或图的称谓；用来标识元素的字符串。

namespace（**命名空间**）模型中可用于定义和使用名称的部分；在一个命名空间内，每个名称都具有唯一的意义。

node（**节点**）运行时存在并代表可计算资源的物理元素，通常至少具有内存，有时也具有处理能力。

note（**注解**）对一个元素或元素集合的注释。

object（**对象**）参见实例（*instance*）。

Object Constraint Language, OCL（**对象约束语言**）一种用于表示无副作用约束的形式化语言。

object diagram（**对象图**）描述某一时刻一组对象及其相互关系的图；对象图展示了系统的静态设计视图或静态过程视图。

object lifeline（**对象生命线**）顺序图中的一条线段，表示对象在该线段所表示的这段时间内存在。

operation（**操作**）类中任何对象为了实现某种行为而请求的服务的实现。

package（**包**）一种用于对元素进行分组的通用机制。

parameter（**参数**）一种值可变化、传递或返回的变量的规格说明。

persistent object（**永久对象**）当创建对象的进程或线程消失后仍然存在的对象。

postcondition（**后置条件**）在操作完成后必须为“真”的约束。

precondition（**前提条件**）调用操作时必须为“真”的约束。

primitive type（**简单类型**）预定义的基本类型，如整型或字符串型。

process（**进程**）一种相互之间可以并发执行的高权限控制流[⊖]。

property（**特性**）描述元素特征的命名值。

realization（**实现**）类元之间的一种语义关系，其中一个类元详细说明一种契约，另一个类元保证该契约的执行。

receive（**接收**）对来自发送者对象的消息实例的处理。

receiver（**接收者**）处理来自发送者对象的消息实例的对象。

relationship（**关系**）元素间的语义联系。

responsibility（**职责**）一个类型或类的契约或责任。

role（**角色**）参与特定语境的一个实体的特有行为。

scenario（**脚本**）一种阐明行为的特定动作序列。

scope（**作用域**）赋予一个名称特定意义的语境。

⊖ “process”是指可以独立并发运行的程序，“thread”是指在“process”内并发运行的程序，因此将“process”译成高权限控制流，将“thread”译成低权限控制流。——译者注

send（**发送**）消息实例从发送者对象到接受者对象的传递。

sender（**发送者**）把消息实例传递到接收者对象的对象。

sequence diagram（**顺序图**）一种强调消息时间顺序的交互图。

signal（**信号**）在实例间传递的异步激励的规格说明。

signature（**特征标记**）行为特征的名称和参数。

single inheritance（**单继承**）泛化的一种语义变体，其中一个类型只能有一个超类型。

specification（**规格说明**）一个特定的构造块的语法和语义的正文陈述；对某事是什么或做什么的说明性描述。

state（**状态**）对象在其生命周期内的一种条件或一种情况，期间该对象满足某个条件、执行某个活动或等待某个事件。

statechart diagram（**状态图**）描述一个状态机的图；状态图展示了系统的动态视图。

state machine（**状态机**）一种详细描述对象在其生命周期内对事件作出响应所经历的状态序列的行为，包括对事件作出的响应。

stereotype（**构造型**）对UML词汇的扩展，允许从现存的构造块衍生地创建专门用于解决特定问题的新构造块。

stimulus（**激励**）一个操作或一个信号。

subsystem（**子系统**）一组元素，其中一些元素制定由所包含的其他元素提供的行为的规格说明。

subtype（**子类型**）在泛化关系中，将超类型特化的类型。

supertype（**超类型**）在泛化关系中，将子类型泛化的类型。

supplier（**提供者**）一种类型、类或构件，它所提供的服务可由其他类型、类或构件来调用。

swim lane（**泳道**）活动图中为组织动作的职责而划分的区域。

synchronous action（**同步动作**）一种使发送对象暂停以等待结果的请求。

system（**系统**）为完成特定目标而组成的子系统集合，可用一组模型从不同的角度来描述。

tagged value（**标记值**）对UML元素性质的扩展，允许在该元素的规格说明中创建新的信息。

task（**任务**）通过一个程序、一个动态模型或控制流的某种其他表示执行的一条简单路径；一个线程或一个进程都可看作一个任务。

template（**模板**）参数化的元素。

thread（**线程**）一种相互之间可以在同一进程中并发执行的低权限控制流。

trace（**踪迹/跟踪**）表示两个元素之间的渊源或处理关系的依赖关系，这两个元素表示同一个概念，但不存在从一个元素衍生另一个元素的特定规则。

transient object（**暂时对象**）仅在创建该对象的线程或进程执行期间才存在的对象。

transition（**转换**）两种状态之间的关系，表明当发生某个特定事件以及满足某个特定条件时，处于第一种状态的对象将执行某一特定动作而进入第二种状态。

type（**类型**）类的构造型，用于说明对象的领域以及适用于该对象的操作（不是方法）。

usage（使用（关系）） 一个元素（客户）为了提供正确的功能或实现而要求另一个元素（提供者）存在的依赖关系。

use case（用况） 一组动作序列及其变体的描述，系统执行这些动作及其变体并对特定的参与者产生可观测的、有价值的结果。

use-case diagram（用况图） 描述一组用况、参与者以及两者之间关系的图，用况图展示了系统的静态用况视图。

view（视图） 从给定的视角或视点看到的模型的一个投影，它忽略了与该视角或该视点无关的实体。

visibility（可见性） 一个名称怎样才能被其他对象见到或使用的性质。

A.4 参考资料

[1] OMG Unified Modeling Language Specification. Object Management Group, Framingham, MA, 1998. Internet: www.omg.org.

[2] James Rumbaugh, Ivar Jacobson, and Grady Booch, *The Unified Modeling Language Reference Manual,* Reading, MA: Addison-Wesley, 1998.

[3] Grady Booch, Jim Rumbaugh, and Ivar Jacobson, *The Unified Modeling Language User Guide,* Reading, MA: Addison-Wesley, 1998.

[4] Martin Fowler, *UML Distilled,* Reading, MA: Addison-Wesley, 1997.

[5] Hans-Erik Eriksson, Magnus Penker, *UML Toolkit,* New York: John Wiley & Sons, 1998.

附录B
针对统一过程的UML扩展

B.1 引言

本附录阐述了统一过程所要求的UML扩展。这些扩展用构造型和标记值的形式来描述，也就是按照UML所提供的扩展机制以及用于描述其中一些构造型的图符来加以描述。那些不属于或不同于UML标准扩展（[1]和[2]所描述的）的构造型将用星号（“*”）来标识。

对于UML的综合情况，请参阅附录A。

B.2 构造型

见表B-1。

表 B-1

构 造 型	应 用 于	简 要 描 述
用况模型	模型	一个包含参与者、用况以及它们之间关系的模型；一个描述系统应为其用户做些什么以及在哪些约束限定之下的模型
用况系统	顶层包	用况模型的顶层包。(《用况系统》是《顶层包》的一个子类型)
分析模型	模型	是一个对象模型，其目的是：① 精确地描述需求；② 按某种方式构造它们，以便于对它们的理解、准备、改变，总之，以便于对它们的维护；③ 作为在设计和实现中塑造系统（包括它的构架）的基本输入
分析系统	顶层包	分析模型的顶层包。(《分析系统》是《顶层包》的一个子类型)
控制类	类	一个在分析模型内表示协作、顺序以及控制其他对象的类，并经常用于封装与某个特定用况相关的控制
实体类	类	一个在分析模型内用于对长久性以及通常是永久性的信息进行建模的类
边界类	类	一个在分析模型内用于对系统与它的参与者（说得更精确些，与用户和外部系统）之间的交互进行建模的类
用况实现—分析 *	协作	一种在分析模型内的协作关系。它阐述了一个特定的用况是如何用分析类（即控制、实体和边界类）以及它们的交互分析对象来实现和执行的
分析包 *	包	一个包，它提供了一种以可管理块的方式来组织分析模型中的制品的方法。一个分析包可能包含分析类（即控制、实体和边界类）、**用况实现—分析**和其他的分析包（递归地）
服务包 *	包	分析包的一种变型，在分析包层次结构（在分析模型中）的较低层次上使用，用来根据它所提供的服务来构造系统

（续）

构造型	应用于	简要描述
设计模型	模型	一个描述用况物理实现的对象模型，重点在于如何将功能性和非功能性需求连同其他与实现环境相关的约束反映到正在考虑中的系统
设计系统	顶层子系统	设计模型的顶层子系统（《设计系统》是《顶层包》的一个子类型）
设计类 *	类	一个设计类在系统实现中表示某个类或者类似构造的一个“无缝抽象”
用况实现—设计 *	协作	一种在设计模型内的协作关系。它阐述了一个特定的用况是如何用设计类、设计子系统以及它们的对象来实现和执行的
设计子系统	子系统	一个子系统，它提供了一种以可管理块的方式来组织设计模型中制品的方法。一个设计子系统可能包含设计类、**用况实现—设计**和其他的设计子系统（递归地）
服务子系统 *	子系统	设计子系统的一种变型，在设计子系统层次结构（在设计模型中）的较低层次上使用，用来根据它所提供的服务来构造系统
实施模型 *	模型	一个对象模型，它以如何将功能分布到计算节点当中的方式来阐述系统的物理分布
实现模型	模型	一个模型，它阐述设计模型中的元素（例如设计类）是如何用构件（例如源代码文件和可执行体）来实现的
实现系统	顶层子系统	实现模型的顶层子系统。（《实现系统》是《顶层包》的一个子类型）
实现子系统	子系统	一个子系统，它提供了一种以可管理块的方式来组织实现模型中制品的方法。一个实现子系统可能包含构件、接口和其他实现子系统（递归地）
测试模型 *	模型	一个描述实现模型中的可执行构件（例如构造）是如何通过集成和系统测试来进行测试的模型
测试系统 *	顶层包	测试模型的顶层包。（《测试系统》是《顶层包》的一个子类型）
测试构件 *	构件	一个自动执行一个或多个测试规程或者其中一部分的构件

B.3 标记值

见表B-2。

表 B-2

标记值	应用于	简要描述
纵览描述	用况模型	用于从整体上解释用况模型的文字描述
事件流	用况	用况动作顺序的文字描述
特殊需求	用况	收集有关某个用况在它的事件流中没有被捕获到的所有需求（例如非功能性需求）的文字描述
特殊需求	分析类（即控制、实体和边界类）	收集有关某个分析类的非功能性需求的文字描述。这些需求在分析中被详细描述，但最好在设计和实现中加以处理
事件流—分析	**用况实现—分析**	对定义用况实现的图（以及它们的标签）加以解释和补充的文字描述
特殊需求	**用况实现—分析**	收集有关某个用况实现的需求（例如非功能性需求）的文字描述。这些需求在分析中被详细描述，但最好在设计和实现中加以处理

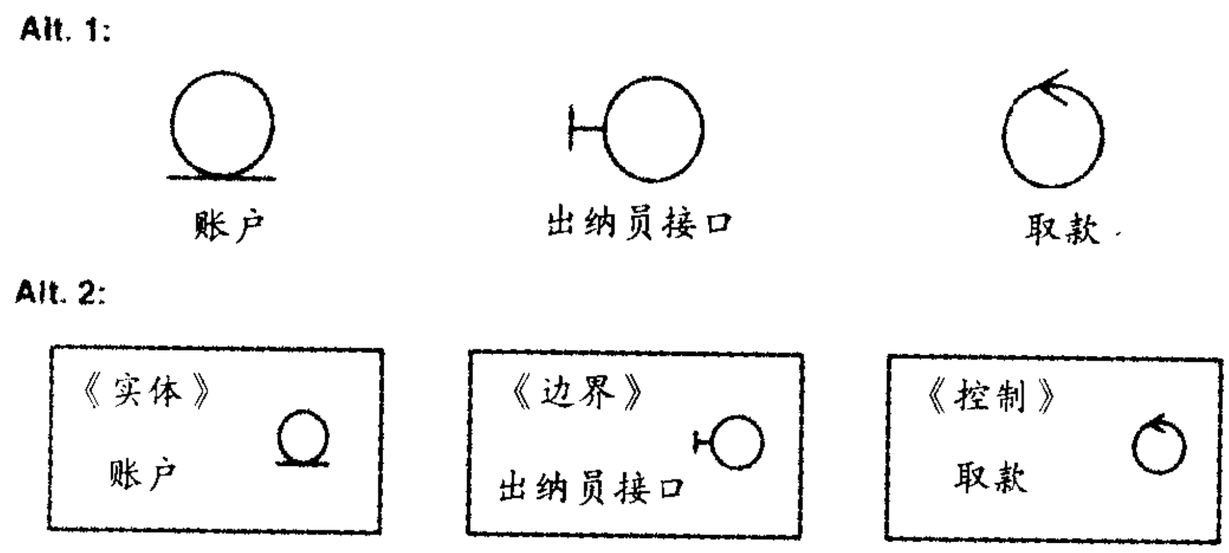
（续）

标 记 值	应 用 于	简 要 描 述
实现性需求	设计类	收集有关某个设计类的需求（例如非功能性需求）的文字描述。这些需求在设计中被详细描述，但最好在实现中加以处理
事件流—设计	**用况实现—设计**	对定义用况实现的图（以及它们的标签）加以解释和补充的文字描述
实现性需求	**用况实现—设计**	收集有关某个用况实现的需求（例如非功能性需求）的文字描述。这些需求在设计中被详细描述，但最好在实现中加以处理

B.4 图符

在B.2节中提到的绝大部分构造型都没有引入任何新的图符，可以在这些构造型所应用到的符号中，改用书名号（“《”和“》”）括住构造型关键字的方式来描述这些构造型。

然而，控制、实体和边界类引入了新的符号，如图B-1中所描述的那样。



图B-1 用于分析中的三种标准类构造型

B.5 参考资料

[1] OMG Unified Modeling Language Specification. Object Management Group, Framingham, MA, 1998. Internet: www.omg.org.

[2] James Rumbaugh, Ivar Jacobson, and Grady Booch, *Unified Modeling Language Reference Manual*, Reading, MA: Addison-Wesley, 1998.

附录C
常用术语

C.1 引言

该附录收录并定义了描述统一过程的常用术语，但有关统一建模语言UML和统一过程对UML的具体扩展的术语未收录在内。关于这两部分术语的简要定义，请参考附录A “统一建模语言概述”和附录B“统一过程对统一建模语言的具体扩展”。

C.2 术语

abstraction（抽象） 一个实体与其他实体相区别的基本特征。抽象定义了从观察者角度来考察事物的边界。

activity（活动） 工作人员在工作流中执行的有形工作单元，它具有以下特点：（1）隐含明确定义的工作人员职责；（2）基于明确定义的输入（一组制品），产生明确定义的输出（另一组制品）；（3）在项目计划中，当把任务分配到个人时，代表一个清晰界定的工作单元。对工作人员来说，也可以看作是执行一个操作。参见制品（*artifact*）、工作人员（*worker*）。

analysis (workflow)（分析（工作流）） 一种核心工作流，其主要目标是通过细化和结构化在需求捕获中描述的需求方法来分析这些需求。这样做的目的是：（1）更精确地理解需求；（2）得到一个易于维护且有助于确定整个系统结构（包括其构架）的需求描述。

application system（应用系统） 向最终用户提供一组有内在联系的用况的系统。

application system suite（应用系统套件） 一组不同的、可以协同工作的应用系统的集合，目的是为某些参与者增加价值。参见应用系统（*application system*）。

application-general layer（通用应用层） 系统中在一定的业务或领域内可以重用的部分（包或子系统）。该层可以被专用应用层所使用。参见专用应用层（*application-specific layer*）。

application-specific layer（专用应用层） 系统中与特定应用有关、不能被其他部分（子系统）共享的部分（包或子系统）。该层可以使用通用应用层。参见通用应用层（*application-general layer*）。

architectural baseline（构架基线） 细化阶段结束时发布的基线，着重于系统的构架。参见细化（*elaboration*）、构架（*architecture*）、基线（*baseline*）。

architectural pattern（构架模式） 定义了某种结构或行为的模式，通常是一个特定模型的构架视图。例如层次模式、客户/服务器模式、三层模式和对等模式，其中每一种模式都对实施

模型定义了某种结构，并且建议如何把构件（功能）分派到它的节点上。参见模式（*pattern*）、构架视图（*architectural view*）。

architectural prototype（构架原型） 本质上是一个可执行的原型，它着重实现模型的构架视图且着重体现原型的构件。如果构架原型是进化的，则通过一个更为完整但原型化的（或略图似的）的构架描述（包括所有的构架视图），明确地建立构架原型的基线。参见进化原型（*evolutionary prototype*）、基线（*baseline*）、构架描述（*architecture description*）、构架视图（*architectural view*）。

architectural style（构架风格） 共享类似的高层结构和核心机制的系统，可以说具有相似的构架风格。

architectural view（构架视图） 对系统中一个特定模型的结构与行为的投影，侧重于该模型中有关构架的重要方面。

architectural view of the analysis model（分析模型的构架视图） 包含分析类、包和用况实现的系统构架视图；它主要展示系统需求的精化与结构化。当设计与实现该系统的构架时，应尽可能地保留该视图中的结构。参见设计模型的构架视图（*architectural view of the design model*）、实现模型的构架视图（*architectural view of the implementation model*）。

architectural view of the deployment model（实施模型的构架视图） 包含构成硬件拓扑结构（用于运行系统）节点的系统构架视图；展示组成物理系统的部件的分布、交付和安装的视图。

architectural view of the design model（设计模型的构架视图） 包含构成系统方案领域的词汇的设计类、子系统、接口和用况实现的系统构架视图；也包含构成系统并发与同步的机制的线程与进程的视图；展示非功能性需求（包括系统的性能、可扩展性和吞吐量等需求）的视图。

architectural view of the implementation model（实现模型的构架视图） 包含用于组装和发布物理系统的构件的系统构架视图；展示系统发布配置管理的视图，其中相对独立的构件可以按照不同方式组装成一个能运行的系统。

architectural view of the use-case model（用况模型的构架视图） 包含与构架有关的重要用况的系统构架视图。

architecture（构架） 对以下一系列重要问题的决策的总和：软件系统的组织；对组成系统的结构元素、接口以及这些元素在协作中的行为的选择；由这些结构与行为元素组合成更大的子系统的方式；用来指导将这些元素、接口、它们之间的协作以及组合等组织起来的构架风格。软件构架不仅涉及到结构与行为，而且涉及使用（关系）、功能、性能、适应性、重用、可理解性、经济和技术约束及其权衡以及美学考虑等。

architecture description（构架描述） 包括模型的构架视图的系统构架的描述。参见构架视图（*architectural view*）、用况模型的构架视图（*architectural view of the use-case model*）、分析模型的构架视图（*architectural view of the analysis model*）、设计模型的构架视图（*architectural view of the design model*）、实施模型的构架视图（*architectural view of the deployment model*）、实现模型的构架视图（*architectural view of the implementation model*）。

architecture-centric（以构架为中心） 在软件生命周期的语境中，以构架为中心意味着将系统构架用作构思、构造、管理和改善该系统的主要制品。

artifact（制品） 具有以下特征的有形信息：（1）由工作人员在执行活动的过程中创建、修改和使用；（2）代表一定的职责范围；（3）被置于独立的版本控制之下。一个制品可以是一个模型、一个模型元素或一个文档。参见工作人员（*worker*）、活动（*activity*）。

baseline(基线) 一组经评审并通过的制品：（1）表示一致同意的可进一步进化和开发的基础；（2）只有通过正式的规程（如配置管理和变更管理）才能对它进行修改。参见构架基线（*architectural baseline*）、配置管理（*configuration management*）。

build（构造） 系统的一个可执行版本，通常用于该系统的某个特定部分。开发是通过一系列的构造过程完成的。

business process（业务过程） 为业务中某个个体客户产生可视且可度量的有价值的结果所需要的全部活动。

cohesive（内聚性） 实体（如系统、子系统或包）将其各个部分聚集在一起的能力。

component-based development, CBD（基于构件的开发） 由构件装配成软件密集型系统的创建和实施，以及对这类构件的开发和获取。

concurrency（并发） 几个独立作业（线程、进程）同时共享一个硬件设备（如处理器）时发生。

configuration management（配置管理） 定义和维护制品的配置和版本的任务。配置管理包括制品的基线定义、版本控制、发布控制、状态控制和存储控制。参见制品（*artifact*）、基线（*baseline*）。

construction phase（构造阶段） 软件生命周期的第三个阶段，在该阶段中，从可执行的构架基线得到软件，直至将该软件移交给用户。

contingency plan（应急计划） 描述在某些风险发生时如何应对的计划。

core workflow（核心工作流） 需求、分析、设计、实现或测试工作流中的一种。参见工作流（*workflow*）、需求（*requirements*）、分析（*analysis*）、设计（*design*）、实现（*implementation*）、测试（*test*）。

customer（客户） 订购一个需要全新构造或以持续版本构造的系统的个人、组织或团体。*customer*与*client*是同义词。

defect（缺陷） 系统的异常表现，如测试中发现的软件故障的征兆，或在评审会上发现的问题。参见测试（*test*）。

design (workflow)(设计(工作流)) 一种核心工作流，其主要目标是针对非功能性需求和方案领域建立模型，并为系统的实现和测试作准备。

design mechanism（设计机制） 设计模型中大量的设计类、协作、甚至子系统，用以实现共性需求，如持久性、分布和性能。

developer（开发人员） 参加核心工作流的工作人员，如用况工程师、构件工程师等。参见核心工作流（*core workflow*）。

distribution（分布） 几个独立作业（线程、进程）分别在不同的硬件设备（处理器）上运

行时的状况。

domain area（领域范围） 由该领域内从业者能够理解的一系列概念和术语所表征的知识或活动的范围。

elaboration phase（细化阶段） 软件生命周期的第二个阶段，在该阶段定义系统的构架。

engineering artifact（工程制品） 在核心工作流中创建的制品。参见核心工作流（*core workflow*）。

evolutionary prototype（进化原型） 经过逐步进化和精化并最终成为待开发系统一部分的原型。它一般要进行配置管理。参见配置管理（*configuration management*）。

exploratory prototype（探索原型） 仅仅以探讨为目的的原型，当这些目的达到后，该原型将不再使用。它一般不进行配置管理。参见配置管理（*configuration management*）。

external release(外部发布(版本)) 面向不属于该项目及其成员的客户和用户的发布(版本)。参见发布(版本)（*release*）。

forward engineering（正向工程） 在软件开发的语境中，通过映射到特定的实现语言，把模型转换成代码。参见逆向工程（*reverse engineering*）。

framework（框架） 微构架，它为特定领域内的系统提供不完整的模板。例如，可以是一个未来要扩展的、重用的或既要扩展又要重用的子系统。

functional requirement（功能需求） 在不考虑物理约束时，确定一个系统必须能够完成的动作的需求；确定一个系统的输入/输出行为的需求。参见需求（*requirement*）。

green-field project（非先验性项目） 一个没有先例的项目。参见项目（*project*）。

implementation (workflow)（实现（工作流）） 一种核心工作流，其基本目标是利用构件（如源代码、脚本、二进制代码、可执行代码等）来实现系统。

inception phase（初始阶段） 软件生命周期的第一个阶段，在该阶段中，对提出的有关开发的基本思想进行充分论证，以确保可以进入细化阶段。

increment（增量） 系统中一个较小的、可管理的部分，通常指两次相邻的构造之间的差异。每次迭代至少产生一个（新的）构造块，从而向该系统增加一个增量。然而，一次迭代可能创建一系列构造块，每个构造块都向该系统增加一个小的增量。这种迭代将向该系统增加一个（可能由多个构造块组成的）较大的增量。参见构造（*build*）、迭代（*iteration*）。

incremental integration（增量式集成） 在软件生命周期的语境中，包含系统构架的连续集成以产生一个发布版本的过程，每个新的版本都包含了对其他版本的增量式改进。

integration（集成） 参见系统集成（*system integration*）。

internal release(内部发布(版本))仅仅在内部面向项目及其成员而不面向客户与用户的发布版本。参见发布(版本)（*release*）。

iteration（迭代） 按照专门的（迭代）计划和评估标准产生一个（内部或外部）发布版本所进行的一组明确的活动。参见发布（*release*）、内部发布（*internal release*）、外部发布（*external release*）。

iteration plan（迭代计划） 一次迭代的细粒度计划。该计划规定这次迭代按时间和资源估算的预期花费以及用制品来表征的期望输出。该计划还规定在该迭代中谁应该做什么以及应该

按照什么样的顺序做。这将通过为该迭代分派个体到工作人员岗位和描述详细的工作流等措施来完成。参见迭代（*iteration*）、制品（*artifact*）、工作人员（*worker*）、迭代工作流（*iteration workflow*）。

iteration workflow（迭代工作流） 一种工作流，它表示对核心工作流（如需求捕获、分析、设计、实现和测试）的集成。对包括参与的工作人员、由工作人员执行的活动及工作人员生产的制品的迭代描述。参见工作流（*workflow*）。

iterative（可迭代体） 在软件生命周期的语境中，包含管理一连串可执行发布版本的过程。

layer（层） 系统中由包或子系统明确定义的部分。参见专用应用层（*application-specific layer*）、通用应用层（*application-general layer*）、中间件层（*middleware layer*）、系统软件层（*system-software layer*）。

legacy system（遗留系统） 从项目里“继承”下来的现存系统。通常是指采用了过时的实现技术创建的旧系统，当该项目构造新的系统时，它必须以整体或部分的形式合并或重用。参见项目（*project*）。

life cycle（生命周期） 参见软件生命周期（*software life cycle*）。

major milestone（主要里程碑） 管理层作出重要的业务决策的里程碑。项目的每个阶段都由一个主要里程碑标志结束，此时，经理要对是否继续该项目作出重要决策并确定该项目的进度、预算和需求。可以把主要里程碑作为一个同步点，在该同步点满足了一组明确定义的目标，生产了制品，对是否进入下一阶段作出了决策，而且综合考虑了管理领域与技术领域的各种问题。参见阶段（*phase*）、项目（*project*）、制品（*artifact*）。

management artifact（管理制品） 一种非工程制品的制品，如由项目经理创建的项目计划。参见工程制品（*engineering artifact*）。

mechanism（机制） 对一个共性问题或共性需求的通用解决方案。例如在设计模型中提供持久性或分布设施的设计机制。

middleware layer（中间件层） 为实用性框架和与平台无关的服务（如分布式对象计算和异构环境下的互操作）提供可重用构造块（包或子系统）的软件层。例如，对象请求代理、用于创建图形用户界面的未确定平台框架，或一般的实现通用设计机制的产品。参见系统软件层（*system-software layer*）、对象请求代理（*object request broker*）、用户界面（*user interface*）、设计机制（*design mechanism*）。

milestone（里程碑） 参见主要里程碑（*major milestone*）、次要里程碑（*minor milestone*）。

minor milestone（次要里程碑） 两个主要里程碑之间的中间里程碑。例如，当一个迭代结束时或当迭代中的一个构造块最后完成时的里程碑。参见主要里程碑（*major milestone*）、迭代（*iteration*）、构造（*build*）。

model-driven（模型驱动） 在软件生命周期的语境中，模型驱动意味着系统开发是根据具有特定目标的不同模型组织的，模块中的各个元素彼此有关（可以跟踪）。

nonfunctional requirement（非功能性需求） 确定系统特性（如环境和实现约束、性能、平台依赖、可维护性、可扩展性以及可靠性）的需求。详细描述功能需求的物理约束的需求。参见需求（*requirement*）、性能需求（*performance requirement*）、可靠性（*reliability*）、功能需

求（*functional requirement*）。

nontechnical risk(非技术风险)与管理制品以及与可用资源（例如人员）、他们的能力、交付日期等方面有关的风险。参见管理制品（*management artifact*）、风险（*risk*）、技术风险（*technical risk*）。

object request broker（对象请求代理） 对分布在异构环境中的对象间实现透明地消息调度和转发的机制。参见分布（*distribution*）。

pattern（模式） 在某一给定语境中，对一个共性问题的通用解。

performance requirement（性能需求） 将行为条件施加到功能需求上的需求，如速度、吞吐量、响应时间、内存使用。参见功能需求（*functional requirement*）、需求（*requirement*）。

phase（阶段） 开发过程中两个主要里程碑之间的时间跨度。参见主要里程碑（*major milestone*）、初始（*inception*）、细化（*elaboration*）、构造（*construction*）、移交（*transition*）。

portability（可移植性） 将一个运行在特定环境下的系统转化为运行在另一个环境下的系统的难易程度。

problem domain（问题领域） 问题（通常是要由系统解决的问题）定义的领域范围。系统的客户通常对问题的领域比较了解。参见领域范围（*domain area*）、客户（*customer*）。

process（过程） 参见业务过程（*business process*）、软件开发过程（*software development process*）、统一过程（*Unified Process*）。

project(项目) 使一个系统经历整个生命周期的开发工作。参见软件生命周期（*software life cycle*）。

project plan（项目计划） 一个勾画出项目整个“行进图”的计划，包括进度、主要里程碑日期和准则，以及阶段到迭代的分解。参见项目（*project*）、主要里程碑（*major milestone*）。

prototype（原型） 参见用户界面原型（*user-interface prototype*）、构架原型（*architectural prototype*）、进化原型（*evolutionary prototype*）、探索原型（*exploratory prototype*）。

regression test（回归测试） 对以前已经测试过的构造块或其一部分进行的复测。回归测试主要是为了验证在“新构造块”中添加了“新功能”后，“旧构造块”中的“旧功能”仍能工作。参见测试（*test*）、构造（*build*）。

release（发布(版本)） 一组相对完整、一致的、交付给内部或外部用户的制品，可能包括一个构造块；这样一组制品的交付。参见制品（*artifact*）、构造（*build*）。

reliability（可靠性） 系统在实际运行环境下正常运转的能力。例如，可以用系统的可用性、准确性、平均故障时间、每千行代码的缺陷数、每个类的缺陷数来度量。

requirement（需求） 系统必须满足的条件或能力。

requirements (workflow)（需求（工作流）） 一种核心工作流，其基本目标是使开发导向正确的系统。为了达到这个目标，需要把系统需求描述得足够明确，使客户（包括用户）与开发人员就系统“应该做什么和不应该做什么”取得共识。参见需求（*requirement*）、客户（*customer*）和开发人员（*developer*）。

reverse engineering（逆向工程） 在软件开发的语境中，通过从特定的实现语言出发的映射，把代码转换成模型。参见正向工程（*forward engineering*）。

risk（风险） 项目遭遇危险或失败的变数。风险可能是项目遇到非期望的事件，如进度拖延、费用超支或项目完全取消。参见技术风险（*technical risk*）、非技术风险（*nontechnical risk*）。

risk-driven（风险驱动） 在软件生命周期的语境中，风险驱动意味着每个新发布版本都集中于解决或减少对项目成功影响最大的风险。

robustness（健壮性） 一个实体（通常是一个系统）对变化的适应性。

software development process（软件开发过程） 软件开发业务的业务过程（或业务用况）。为将客户需求转换为代表软件产品的一组协调一致的制品所需要的活动全集；在后期将需求变化转化为该软件产品的新版本。参见业务过程（*business process*）、统一过程（*Unified Process*）。

software life cycle（软件生命周期） 按如下顺序的四个阶段的循环：初始、细化、构造、移交。参见初始（*inception*）、细化（*elaboration*）、构造（*construction*）、移交（*transition*）。

solution domain（方案领域） 定义（问题的）方案领域的范围，通常，解表示一个系统的设计与实现。方案领域通常为系统开发者所理解。参见领域范围（*domain area*）、开发人员（*developer*）。

supplementary requirement（补充需求） 不能与某个特定用况或某个特定现实世界中的类（如领域或业务实体类）相关联的一般性需求。

system integration（系统集成） 将系统的部分构件编译并连接成为一个或多个可执行体（仍然是构件）。

system-software layer（系统软件层） 包含计算和联网基础设施的软件层，如操作系统、数据库管理系统、专用的硬件接口等。这是软件层次结构中的最底层。参见中间件层（*middleware layer*）。

systemware（系统软件） 参见系统软件层（*system-software layer*）。

technical risk（技术风险） 与工程制品以及与实现技术、构架、性能等方面相关的风险。参见工程制品（*engineering artifact*）、构架（*architecture*）、性能需求（*performance requirement*）、非技术风险（*nontechnical risk*）。

test (workflow)（测试（工作流）） 一种核心工作流，其基本目标是通过测试每一个构造块（包括内部构造块、外部构造块以及移交给外部单位的最终发布版本）来验证实现的结果。参见实现（*implementation*）、构造（*build*）、内部版本（*internal release*）和外部版本（*external release*）。

test case（测试用例） 用于测试系统的一个案例的规格说明，包括测试的目的、输入、结果和条件。

test evaluation（测试评价） 对测试工作结果的评价，包括测试用例覆盖率、代码覆盖率和缺陷状态。参见测试（*test*）、测试用例（*test case*）、缺陷（*defect*）。

test plan（测试计划） 描述测试策略、资源和进度的计划。

test procedure（测试规程） 对于如何完成一个或多个案例或其一部分的规格说明。参见测试用例（*test case*）。

transition phase（移交阶段） 软件生命周期的第四个阶段，在该阶段，软件移交给用户方。

Unified Modeling Language, UML（统一建模语言） 软件的统一建模语言，它是一种对软件密集型系统的制品进行可视化、详述、构造和文档化的语言。统一过程所使用的语言。一种使开发人员可以按照标准化蓝图或图从而使他们的产品（制品）可视化的语言。参见制品（*artifact*）、统一过程（*Unified Process*）、开发人员（*developer*）。

Unified Process（统一过程） 基于统一建模语言的软件开发过程，它是迭代的、以构架为中心、用况驱动和风险驱动的。围绕四个阶段（初始、细化、构造和移交）和五个核心工作流（需求捕获、分析、设计、实现和测试）来组织的过程。采用业务模型来描述的过程，而业务模型是由三种基本的构造块（工作人员、活动和制品）来构造的。参见软件开发过程（*software development process*）、统一建模语言（*Unified Modeling Language*）、迭代（*iterative*）、以构架为中心（*architecture-centric*）、用况驱动（*use-case driven*）、风险驱动（*risk-driven*）、阶段（*phase*）、初始（*inception*）、细化（*elaboration*）、构造（*construction*）、移交（*transition*）、核心工作流（*core workflow*）、需求（*requirement*）、分析（*analysis*）、设计（*design*）、实现（*implementation*）、测试（*test*）、工作人员（*worker*）、活动（*activity*）和制品（*artifact*）。

use-case driven（用况驱动） 在软件生命周期的语境中，用况驱动意味着为建立该系统的期望行为和为与该系统的其他项目相关人员对这种行为进行交流而将用况作为主要制品。它也意味着用况是该系统的分析、设计、实现与测试的基本输入，包括该系统构架的创建、验证和确认。参见分析（*analysis*）、设计（*design*）、实现（*implementation*）、测试（*test*）、构架（*architecture*）。

use-case mass（用况群） 用况模型中所有用况动作的全集。

user（用户） 与系统进行交互的人。

user interface（用户界面） 用户与系统进行交互的界面。

user-interface prototype（用户界面原型） 主要是一个可执行的用户界面原型，但在开发的早期阶段，可能只包含画在纸上的草图、屏幕上的位图以及其他类似的成分。

view（视图） 从某个给定视角或视点出发、而忽略了与该视角或该视点无关的实体所得到的模型的投影。

visual modeling（可视化建模） 以标准化蓝图或图将工作产品（制品）可视化。参见制品（*artifact*）。

waterfall approach（瀑布方法） 将开发安排成线性工作序列的系统开发方法，例如按照如下的顺序：需求捕获、分析、设计、实现和测试。参见需求（*requirement*）、分析（*analysis*）、设计（*design*）、实现（*implementation*）、测试（*test*）。

worker（工作人员） 赋予一个个人或一个群组的职位，需要承担一定的职责和具备相应的能力，诸如执行某些活动和开发某些制品。参见活动（*activity*）、制品（*artifact*）。

workflow（工作流） 一个（或部分）业务用况的实现。可以通过活动图来描述，图中包括参与的工作人员、由工作人员执行的活动及工作人员生产的制品。参见核心工作流（*core workflow*）、迭代工作流（*iteration workflow*）。

索引

Alexander Christopher, 54
alpha (α)测试, 307
Beta（发布）版本, 11
Beta（发布）版本, 构造阶段, 248, 294, 295
Beta（发布）版本, 移交阶段, 248, 308~310, 311~312
Boehm Barry, 67
Constantine Larry, 125
Drucker Peter F., 67
Ericsson（爱立信）AXE系统, 49~50
Java语言, 183~184, 225, 289
Jones Capers, 68
Proxy（代理服务器）设计模型, 54
Rational统一过程（RUP）, 324~325
ROP（Rational Objectory Process）, 325

A

安装测试, 230
安装产品, 312~313

B

“白盒”测试, 28, 230
包
  服务, 145~146, 152~154, 171
  图符, 330
  用况, 114
  参见分析包
包含关系, 129~130
边界类, 34, 139~140, 144
  设计, 197
  识别, 155~156
  图符, 340
编程语言, 167, 222~223
编译依赖, 208
标记值, 339
标准化, 48
补充需求, 89, 97~98

C

参与者, 18~19, 27
  角色, 31, 102~103
  确定, 111~112
  识别, 270, 283~285
  特殊的/独特的, 113
  系统, 96~97
  业务, 93~94, 96~97
  用户界面, 123~124
测试, 227~242
  alpha = α（测试）
  beta = β（测试）
白盒, 230
初始阶段, 292~294
单元, 223~226
迭代, 74~75, 228~229, 238
工作流, 4, 234~235
工作人员, 233~234
构造阶段, 28, 230
规格说明, 223~224

黑盒, 28, 230
集成, 275
计划编制与评价/评估, 233, 234~235, 240~242
结构, 224~226
矩阵（用例）, 231
目的, 227
驱动器和桩（模块）, 304~305
生命周期,
系统, 238, 240
细化阶段, 208, 233, 304
验收, 309, 311~312
用况, 42~43, 230
测试工程师, 233, 253
测试规程, 42, 231~232, 238~239
构件, 233, 239~240
测试用例, 42~43, 229~231
测试模型, 8, 28
回归, 238
集成, 236~237, 240~241
系统, 238, 240
层次, 55, 289
分析包, 154, 160
设计子系统, 184~185
层次模式, 55~56
产品, 8, 12
规模, 275
和客户, 312~313
课程, 298
软件系统, 16~19
新的, 260~261
抽象用况, 127, 129
初始阶段, 11, 65, 79, 264~279
测试, 275
迭代, 251, 268~270
分析模型, 273
风险, 246, 267~268
和细化阶段, 277
计划编制与评价/评估, 265~267
可交付内容（东西）, 278
目标, 77
评估, 276~277
软件组织, 264~265
设计模型, 274~275
生命周期, 318
实施模型, 274
实现阶段, 274
系统范围, 267
需求, 90, 270~273
业务案例, 245~246, 264~265, 268, 275~276

D

代码, 16
测试, 224~226
构造, 304
设计类, 222~223
单一客户产品, 312~313
单元测试, 223~226
道具 测试, 233
等价类, 224
第三方确认
迭代, 6~7, 9, 65~80
发布版本, 65~66
风险管理, 67~69, 71~73
构架, 50~51, 68
构造, 68~70, 217~218
和瀑布方法, 67, 70, 74, 75
基线, 76
集成, 69~70
计划编制与评价/评估, 75~76, 252~255
阶段, 75, 79~80
看作袖珍项目（小项目）, 16, 65
模型, 78~79

评估, 262~263
生命周期, 77~78
协作, 71
用况优先级, 255~256
增量, 76
迭代工作流, 74~75
初始, 251, 268~270
构造, 298~299, 305
通用化/泛化, 245~246, 248~251
细化, 281~283, 294~295
移交阶段, 310, 315~316
动作, 4
对象, 22
参见类,
分布, 55, 189
分析, 141, 143, 156~157
领域, 88, 91
设计, 170, 193~194
实体, 140
业务, 95
用况的实现, 34~36
主动的, 61~62, 187, 217

F

发布版本, 7, 16
迭代, 65, 66
内部的, 53, 66
外部的, 66
参见Beta发布版本
反馈, 13
泛化
分析类, 160
设计类, 199
图符, 331
协作, 191
用况, 127~128
范围系统, 267
非功能性需求, 32, 89
初始阶段, 270
用况, 105, 120
“非先验性项目”, 15
费用/花费/花销, 261~262
测试, 235
分析模型, 137
业务标书/业务报价, 275~276, 293
分析包, 144~147, 161~162
设计子系统, 179~181
识别, 149~152
细化阶段, 288
依赖（性）, 153~154
分析类, 137~141
边界类, 34, 139~140, 144
泛化, 160
共享, 151
构造型, 197
关联和聚合, 159~160
和设计类, 185~187, 191
角色与职责, 138, 158~159
控制类, 33, 141, 143
类图, 142
识别, 155~156
实体类, 34, 140, 143, 147, 154
属性, 138, 159
用况, 287~288
分析模型, 8~9, 132~163
初始阶段, 273
费用/花费, 137
服务包, 152~153
工作人员, 147~149
构架描述, 146~147
构造阶段, 302~303
和构造型, 34
和设计模型, 134~136, 168, 185
和用况, 27, 33~36, 155~158, 287~288

接口, 185~186
目的, 135
生命周期, 135~136
事件流描述, 144
特殊需求, 144~145, 155, 157~158, 160
细化阶段, 285~288
需求, 132~134, 137
用况实现, 36~39, 143~144, 166, 168
制品, 137~147
分组, 327, 330
风险管理, 13
初始阶段, 247
迭代, 66~68, 71~74
风险清单, 254~255
构架, 72~73, 177~178
构造阶段, 299
关键风险, 267~268, 270
计划编制, 254~258
细化阶段, 279
需求, 73, 178~179
用况优先级, 255~256
否定测试, 231
服务包, 145~146
分析模型, 152~153
设计模型, 171
服务子系统, 39~42, 49
测试规程, 238
构造阶段, 302~304
设计模型, 171
实现模型, 210

G

跟踪依赖, 8, 18~19
分析模型, 132
设计和实现模型, 208
用况, 30
工程制品, 17
工具, 12, 22~25
初始阶段, 269
细化阶段, 292
工作单元, 94
工作流, 9, 20~21, 101
测试, 4
分析模型, 149
其中的制品, 74
设计模型, 177~178
实现模型, 215~213
通用迭代, 245~246, 248~251
需求, 90, 108~111, 171
用况, 29~30
与测试有关的, 234~235
参见核心工作流；迭代工作流
工作人员, 13~15, 16~17
测试, 233~234
迭代, 249~251
分析模型, 147~149
构造阶段, 297
和制品, 20~21
角色, 102~103
如……的参与者, 102
设计模型, 175~176
实现模型, 213~215
需求, 106~109
业务建模, 95, 97, 101
移交阶段, 309
功能规格说明, 4
功能模块化, 49
功能性需求, 29~30
看作用况, 31~32, 88~89
构架, 45~63
初始阶段, 245~246, 265~267, 273
迭代, 66
分层/层次化, 55, 289
分析模型, 146~147, 150~155

风险, 72~73, 177~178
功能, 47~48
构造, 50
构造阶段, 302~304
和用况, 5, 30~331, 49~52, 72, 112
模式, 54~56
软件, 46
设计模型, 59~61, 173~174, 177~191
生命周期, 319~320
实施模型, 61~62
实现模型, 62, 212~213, 216~217, 291
细化阶段, 246~247, 289~290
需求, 286~287
易理解性, 13
用况模型, 59
子系统, 48, 49
构架风格, 46
构架基线, 52~54, 56
细化阶段, 248, 282, 283, 287
用况群, 277~278
构架描述, 47~48, 56~59
分析模型, 146~147
设计模型, 172~173
实施模型, 174~175
实现模型, 212~213
用况模型, 59, 105, 116
构架设计师, 58~59
初始阶段, 268, 272
分析模型, 147
构造阶段, 296, 297
设计模型, 175
实现模型, 213
需求, 108, 109
移交阶段, 313
构件, 40~42
测试, 233, 239~240
接口, 210
图符, 329
文件, 222~223
依赖（性）, 208
可执行体, 216~217
构件工程师
测试, 224~226, 233, 304
分析模型, 148, 301~302
设计模型, 176
实现模型, 213~215
构想陈述, 266
构造, 49, 68~70
构造阶段, 304~306
集成, 213~214, 217~218
细化阶段, 291~292
参见增量
构造阶段, 11, 66, 79, 294~306
测试, 304~305
迭代工作流, 298~299, 305
分析模型, 302~303
风险, 299
和移交阶段, 305~306
计划编制与评价/评估, 297~298, 304~305
可交付（内容/东西）, 306
目标, 77, 248
评估, 305
设计模式, 302~304
生命周期, 319
实现模型, 304~305
细化阶段, 294~295, 296~297, 299, 301
需求, 301~302
业务案例, 305
构造型, 21, 338~340
测试用例, 42
分析模型, 33, 139, 197
服务包, 145
服务子系统, 171
构件, 206~207

设计类, 168
图符, 340
业务模型, 100
参见类
关联, 159~160
产生代码, 222
分析类, 199
图符, 330
关系, 18~19, 28
包括, 129~130
泛化, 127~128
分析类, 138, 160
扩展, 128~129
设计类, 167
使用, 127
图符, 330~331
管理, 320~324
管理类制品, 17
规格说明测试, 223~224
过程软件开发, 3, 12, 19~25
工具, 23~25
工作流, 20~21
生命周期, 7
实例, 19
特化, 21~22
通用的, 22
用况驱动的, 4
自动化, 23~24

H

核心工作流, 74~75, 78, 248~251
初始阶段, 270~275
构造阶段, 299~305
细化阶段, 283~293
移交阶段, 311
"黑盒"测试, 28, 230
候选测试用例, 28
候选构架, 265~266
候选用况, 32~33
回归测试, 74~75, 228~229, 238
活动, 100, 108~110
活动图, 20, 103
用况说明/用况描述, 120

J

机器码, 16
机制, 57
基线、构架, 52~54, 56
细化阶段, 248, 282, 283, 287
用况群, 277~278
集成测试, 213~214
设计, 236~237
执行, 240~241, 292, 305
集成测试人员, 234
集成构造计划, 213~214
构造阶段, 304~306
实现（阶段）, 217~219
细化阶段, 291~292
计划编制, 251~253
初始阶段, 265~266
迭代, 75~76, 252~255
风险, 254~258
构造阶段, 294~295
细化阶段, 277
移交阶段, 308~310
价值……的结果, 113
交互图
分析模型, 141~144
设计模型, 169~170
用况描述, 120
角色
参与者的, 112
工作人员的, 13~15, 96, 102~103
类, 33, 36, 92, 159

脚本测试, 233
阶段, 7~8, 9~11
迭代, 75, 79~80
计划编制, 251
评估, 262~263
接口
设计模型, 168, 173, 201~202, 290, 291
实现模型, 208~209, 210~211, 219~220
图符, 328
协议, 294~295
需求, 97
用户, 89, 106, 108, 160, 301
子系统, 39~53, 48, 185~186, 194~196
节点, 61
构件, 217
设计模型, 178, 197
实施模型, 173~175
图符, 329
细化阶段, 289
子系统, 181~182
结构性测试, 224~226
结构性事物, 328~330
经理, 320~322
矩阵测试, 230
具体（化）用况, 127, 129
聚合, 159~160, 199
开发过程, 也见过程, 软件开发
开发环境, 269
开发组结构, 12~13, 71, 287
可靠性, 89
可行性, 项目, 13, 246, 269
可执行构件, 216~217
客户（端）, 307, 313~314
客户/服务器模式, 55
课程产品, 298
扩展机制, 327~328, 332, 338~340
控制类, 33, 141, 143
设计, 197
识别, 156
图符, 340
扩展关系, 128~129
类
边界, 139~140, 143, 156~157, 197
等价（类）, 223~224
共享, 151
构造型, 34
角色, 33, 36, 92, 159
控制, 141, 143, 156, 197
领域, 91~92, 95, 151
实体, 140, 143, 147, 154, 155, 197~198
实现, 220, 222~223, 291
图符, 328, 340
用况, 30
职责, 28, 36, 288
子系统, 39~40
也见主动类；分析类；设计类；构造型
类图, 34, 38
分析模型, 142
领域模型, 91
设计类, 169~170
类元, 27
抽象/具体, 190~191
角色, 33
如……的用例, 103
里程碑, 9, 64, 77
次要的, 77, 252
主要的, 78, 251, 318~319
领域类, 91~92, 95
分析包, 151~152
实体类, 154
领域模型, 参见业务模型
路径用况, 32, 103, 194

基本的/基础的, 118~119

M

描述, 构架, 47~48, 56~59
分析模型, 146~147
构架基线, 53~54
设计模型, 274, 173~174
实施模型, 174~175
实现模型, 212~213
用况模型, 59, 106, 116
模板, 54, 55
模式构架, 54~56
模型, 17~19
和迭代, 78~79
基线, 76
集/集合, 263, 314
模型制品, 78

N

内部（发布）版本, 53, 66

P

配置测试, 231
评估, 262~263
评估, 参见评价/评估, 253
初始阶段, 266~268, 276
迭代, 262~263
构造阶段, 297~298, 304, 305
细化阶段, 280~281, 204~205
移交阶段, 309, 315~316
瀑布模型方法, 67, 70, 74, 75, 264

Q

契约、接口, 294~295
强度测试, 231
驱动程序测试, 232, 304
缺陷, 233, 242, 312

R

人员, 12~15
软件, 16~19
生产安排, 308
生命周期, 22
系统, 182~184
组织, 264~265
软件开发过程 参见过程软件开发, 16, 222~223

S

上市产品（用于市场销售的产品）, 312~313
设计类, 36, 41, 167~168, 196~201
操作, 198~199
泛化, 199
方法, 199
勾画, 197~198
构件跟踪, 207
关联和聚合, 199
接口, 172, 202
类图, 169~170
识别, 185~187, 191~193
实现, 220, 222~223
特殊需求, 201
细化阶段, 289, 290
属性, 198~199
状态图, 200
设计模式, 54
设计模型, 8, 18, 164~205
初始阶段, 274~275
对象分布, 189
工作流, 177~178
工作人员, 175~176
构架, 59~61, 173~174, 177~191

构造阶段, 302~304
协作, 189~191
和分析模型, 134~136, 168, 185
和实施模型, 27, 173~175
和实现模型, 40~42
接口, 289, 290
节点和网络配置, 178~179, 197, 290
目的, 164~165
生命周期, 165~166
事件流描述, 170
细化阶段, 288~290
永久性/持久化, 189
用况, 36~39, 166, 168~170, 189~196, 290~291
制品, 166~175
子系统, 170~171, 179~186, 194~195, 201~202
设计约束, 98
生存能力/可行性, 64
生命线, 195
生命周期, 7~11, 15~16
测试, 227~228
迭代, 77~78
分析模型, 135~136
工具, 24~25
构架, 319~320
目标, 318
设计模型, 165~166
实现模型, 205~206
需求, 90
时间分配, 251, 252
用况, 275
实例
过程, 19
类, 30
用况, 103~104
实施模型, 9
初始阶段, 274
构架, 61~62
构件, 41
和设计模型, 27, 173~175
实体类, 33, 140, 143
服务包, 146
设计, 197~198
识别, 154, 156
图符, 340
实现模型, 8, 27, 205~226
操作, 222
初始阶段, 274
工作流, 215~216
工作人员, 213~215
构架, 62, 212~213, 216~217, 291
构件, 206~208, 210
构造阶段, 304~305
和设计模型, 27, 167~168, 207~213, 217, 220
和用况, 40~42
集成, 213~214, 217~218
接口, 208~209, 210
目的, 205
生命周期, 205~206
细化阶段, 291~292
需求, 168, 170, 172, 201
子系统, 208~210, 219~222, 291
实现约束, 98
使过程特定化, 21~22
使用关系, 127
事件流描述, 36~37
事物UML, 327
视图, 18, 46, 47
手册/指南用户, 31, 298
受控迭代, 6~7
书名号（《 》）, 33
术语表制品, 106

数据移植, 313
顺序图, 37~39, 103
标签, 170
集成测试用例, 236~238
设计模型, 169~170, 193~194

T

特殊参与者, 113
特殊需求
设计类, 201
需求模型, 144~145, 155, 157~158, 160
用况, 105, 120
特征, 87~88, 270
通用过程, 21
通用设计机制, 188~191
统一过程, 3~11
迭代和增量, 6~7
定义, 3
概述/综述, 318~319
构架, 5
生命周期, 7~11
术语, 341~348
特化, 324~325
优点, 325~326
主题, 319~321
转化, 320~324
统一建模语言 (UML), 3, 320, 327~340
标准化, 48
词汇, 422
符号（表示法）, 92, 328~331
可视化建模, 24~25
扩展机制, 327~328, 332, 338~340
术语, 332~337
投资回报, 276, 294
外部版本, 66
网络配置, 178~179, 290
文件型构件, 222~223
物理需求, 97
系统测试, 238, 240, 293
系统测试人员, 234
系统的系统（基于别的系统开发出的系统）, 104~105
系统分析员, 106~107, 110
初始阶段, 268, 272
构造阶段, 297
需求, 113
系统集成人员, 215, 303~305
系统软件, 182~184
细化阶段, 11, 65~66, 79
测试, 292~293
迭代, 281~283
分析模型, 285~288
风险, 281
构架, 246~247
和构造阶段, 294~295, 296~297, 299, 301
计划编制与评价/评估, 277, 280~281
可交付内容（东西）, 295
目标, 77, 279~280
评估, 294~295
设计模型, 288~290
生命周期, 319~320
实现模型, 291~292
需求, 90, 280, 282~286
业务案例, 281, 293~294
相互作用/交互作用, 329
项目, 12, 19
进度/进度表, 13
可行性, 13, 246, 269
类型, 258~260
袖珍项目（小项目）, 6
周期, 15~16
项目经理, 253, 262
构造阶段, 296~297, 305

细化阶段, 294~295
移交阶段, 308~309, 314
转变到统一过程, 322~324
项目相关人员, 16~17
消息, 31, 169~170
分析对象, 157
接口, 195
协作图, 36~37, 103
分析模型, 141~144, 156~157
和顺序图, 37
行动案例陈述, 321
行为类事物, 327, 328
性能风险, 72
性能需求, 88, 120
袖珍项目（小项目）, 6, 65
需求, 20, 85~98
补充需求, 89, 97~98
不确定性, 267
初始阶段, 272~273, 90, 89~90, 105, 120
非功能性需求, 32, 89, 105, 120
分析模型, 132~134, 137
风险, 73, 178~179
工作流, 109~111, 129
工作人员, 106~109
功能性需求, 29~30, 31~32, 88~89
构架, 286~287
构造阶段, 301~302
候选需求, 87~88
领域模型, 88, 91~93, 90, 280
起始点, 86~87
生命周期, 90
实现性需求, 168, 170, 172, 201
细化阶段, 90, 280, 282~286
需求集合, 89
业务模型, 88, 93~97
制品, 101~106
需求管理, 22
参见特殊需求;用况

Y

验收测试, 309, 311~312
业务案例, 64
初始阶段, 245~246, 264~265, 268, 275~276
构造阶段, 305
细化阶段, 281, 293~294
移交阶段, 314~315
业务标书/业务报价, 275~276, 293
业务类, 140
业务模型, 8, 93~97
初始阶段, 270
和领域模型, 88, 91~93
寻找参与者, 111~112
制品, 100
依赖（性）
分析包, 153~154
跟踪, 8, 18~19, 30, 133, 213
构件, 208
图符, 330
子系统, 184~186, 201
移交阶段, 11, 64, 79, 307~316
beta（发布）版本, 308~310, 311~312
迭代, 311
工作人员, 309
和构造阶段, 305~306
计划编制, 308~310
可交付内容（东西）, 316
客户及产品, 312~312
目标, 78, 307~308
评估, 315~316
评价/评估, 310
软件, 308
业务案例, 314~315
“泳道”, 20

遗留系统, 50
分析模型, 135
设计模型, 171
永久性/持久化, 189, 197
用户, 4
用户界面设计人员, 108, 109
用户界面原型, 106, 108, 122~126
构造阶段, 301
规格说明, 89
物理的, 124~126
细化阶段, 285
用户手册, 31, 298
用况, 4
包, 114
参与者, 31, 111~112
测试, 42~43, 230
抽象/具体, 127, 129
定义, 32
发现, 113
泛化, 127~128
分类, 256~257
分析包, 149~151
分析模型, 27, 43~48, 155~158, 287~288
服务包, 145~146
工作流, 273
构造, 217~218
和实现模型, 40~42
看作类元, 103
可跟踪性, 30
扩展关系, 128~129
路径, 32, 103, 119~120, 194
时间分配, 275
识别, 260
实例, 103~104
事件流描述, 105~106, 116~121
特殊需求, 105, 120
图符, 328
详细描述, 116~118, 272~273, 283~285, 301
需求, 26~29, 31~32, 85, 132~134
业务, 93~94, 96~97
优先级排序, 116, 255~256, 272, 285, 301
与构架, 5, 30~331, 49~52, 72, 112
与设计模型, 28, 36~37, 191~195, 290~291
子系统, 104~105, 302~303
用况工程师
分析模型, 147~149
设计模型, 176
用况描述, 113~121
活动图, 121
结构化, 118~119
内容, 118~122
纵览, 综合, 114~116
用况描述人员, 107, 107, 108
初始阶段, 268
构造阶段, 297
用况模型, 4, 8~9, 18, 101~102
构架描述, 59, 106, 116
和分析模型, 134
基本的, 124
结构化, 126~129, 285, 301
用况图, 31
用况实现
测试用例, 230
分析模型, 36~39, 143~144, 166, 168
符号, 33~34
接口, 186
类图, 34
设计模型, 36~39, 166, 168~170, 189~196
事件流描述, 36~37
特殊需求, 157~158
协作图, 36~37, 39

业务, 94~95
优先级清单用况, 255~256, 301
有价值的结果, 113
“鱼形”符号, 21
预制系统, 321
原型
初始阶段, 268~270
概念的验证, 247
用户界面, 106, 108, 122~126, 285, 301
源代码

Z

增量, 6~7, 76
构造阶段, 299
集成, 213~214
用况, 30
“真正的”用况, 127, 129
参见迭代
支持者, 322
职责 类, 28, 36
分析类, 138, 157~159, 288
指导者, 323
制品, 16
测试模型, 228~233
分析模型, 137~147
工作流, 74
和工作人员, 20~21
模型, 78
设计模型, 166~175
实现模型, 206~213
术语表, 106
需求, 101~106
需求集合, 89
业务模型, 100
移交阶段, 313
质量保障（保证）, 24~25
中间件, 50, 182~184
重建工程指南, 321~322
重用/复用, 48~49
周期 参见生命周期
注解/注释, 327, 330
注释, 330
桩, 208, 304
状态机, 329
状态图, 27, 103
设计类, 200
用况描述, 118, 121~122
资源, 15
子系统, 39~40
层次, 55
服务, 39~42, 239, 302~303
高层, 18
共享功能性, 181
构架, 48, 49
构造阶段, 302
接口, 185~186
设计模型, 170~171, 179~186, 194~195, 201~202, 188, 290
实现模型, 208~210, 219~222, 291
依赖（性）, 184~186, 201
用况, 105
自动化过程, 23~24
自约束, 18
组织, 264~265
组织, 迭代, 80
组织, 类型, 264~265
组织, 模式, 16