软件工程技术丛书

设计系列

美国
"软件开发"杂志
效率大奖获奖作品

面向模式的软件体系结构
卷1：模式系统

Pattern-Oriented Software Architecture
Volume 1: A System of Patterns

Frank Buschmann　Regine Meunier
（德）Hans Rohnert　Peter Sommerlad　著　贲可荣　郭福亮　赵皑　等译
Michael Stal

WILEY
机械工业出版社
China Machine Press

Pattern-Oriented Software Architecture
Volume 1: A System of Patterns

面向模式的软件体系结构
卷1：模式系统

软件模式根本改变了开发人员对软件应该如何设计、构建和文档化的思考方式。任何一位认真求索的程序员都应该将本书放到自己的书架上作为必备的参考用书。

Amazon.com评价：

为了从软件工程的更大范围内前瞻模式的潜能...本书指出了如何将模式应用于各个层次，从用于特定计算机语言（如C++）的惯用法到设计模式，再到对改善整个软件工程周期都有用的高层模式。

Amazon.com客户评价：

关于模式的一本杰作。该书是《设计模式》（Gamma等著）的参考读物。

一本适用于每一个软件构架师的著作...它确实能使我们缩短所需的设计时间。

这是一本特别的模式专著...我不但曾论证“四人帮”的书是20世纪90年代最重要的编程著作，我也认为...本书是一本更加深入、更加成熟的著作。

CVu 1997年的评价：

《设计模式》（Gamma等著）已是一本广为传阅的经典著作，我认为《面向模式的软件体系结构》也应该成为经典。《设计模式》覆盖了面向对象开发中的一个抽象层，而这本书覆盖了三个层次：高层体系结构模式、设计模式和语言特定的惯用法。

封面设计 陈子平

ISBN 7-111-11182-6
9 787111 111825

HZ BOOKS
华章图书

网上购书：www.china-pub.com
北京市西城区百万庄南街1号 100037
购书热线：(010)68995259，8006100280 (北京地区)
总编信箱：chiefeditor@hzbook.com
ISBN 7-111-11182-6/TP · 2666
定价：45.00 元

软件工程技术丛书

设计系列

TP311.12
40

Frank Buschmann　Regine Meunier
(德)　Hans Rohnert　Peter Sommerlad　著
Michael Stal

贲可荣　郭福亮　赵皑　等译

面向模式的软件体系结构
卷1：模式系统

Pattern-Oriented Software Architecture
Volume 1: A System of Patterns

北方工业大学图书馆
00524093

机械工业出版社
China Machine Press

本书系统地介绍了模式的概念并讨论了模式描述的原则；给出了来自不同应用领域的8个软件体系结构模式；还给出了8个设计模式，这些模式分别针对在定义了软件系统的整体结构之后遇到的典型问题；对惯用法模式进行了阐述；论述了将模式组织成模式系统的重要性；讨论了模式如何嵌入到软件体系结构中，特别是对软件体系结构及其基本原则的理解。此外，本书还介绍了模式历史、相关工作和整个模式团体，并对模式的未来发展作了展望。

本书力图做到既是教材又是参考指南，帮助软件开发人员以一种新的方式考虑软件体系结构，并提供了一些技术来解决特定的再现设计问题，对软件开发的初学者和专家都有帮助。把本书用作软件工程课程的指南，可以给学生提供大型软件设计的完整的新观点。把本书用作参考手册，可以提供全面的技术并随查随用。本书包含了涉及模式实际应用的许多指导原则和约束限制。本书可作为计算机专业高年级本科生、研究生的软件工程教材或参考书，亦可作为软件开发人员的参考手册。

Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad and Michael Stal: Pattern-Oriented Software Architecture, Volume 1: A System of Patterns (ISBN: 0-471-95869-7).

Authorized translation from the English language edition published by John Wiley & Sons, Inc.

Copyright © 1996 by John Wiley & Sons Ltd.

All rights reserved.

Simplified Chinese language edition copyright © 2003 by China Machine Press. All rights reserved.

本书中文简体字版由约翰-威利父子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：01-2002-0816

图书在版编目（CIP）数据

面向模式的软件体系结构 卷1：模式系统 / 布希曼（Buschmann, F.）等著；贲可荣等译.
-北京：机械工业出版社，2003.1

(软件工程技术丛书-设计系列)

书名原文：Pattern-Oriented Software Architecture, Volume 1: A System of Patterns

ISBN 7-111-11182-6

I. 面… II. ① 布… ② 贲… III. 数据结构 IV. TP311.12

中国版本图书馆CIP 数据核字（2002）第088419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

北京第二外国语学院印刷厂印刷 · 新华书店北京发行所发行

2003年1月第1版第1次印刷

787mm × 1092mm 1/16 · 19印张

印数：0 001-5 000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

软件体系结构的模式描述了在特定设计语境中出现的设计问题的特殊重现，并为这种方案提供了一个被证明良好的一般计划。方案计划通过描述它的关键组件、它们的责任和相互关系以及它们结合的方式来指定。

模式有助于利用资深软件工程师的经验来构建软件。模式总结在软件开发中现存的、被证明良好的经验，再用来促进好的设计实践。每个模式处理一个软件系统的设计或实现中一种特定类型的重复出现的问题。模式可以用来构建具有特定属性的软件体系结构。

保存模式文档已被证明是很好的设计经验，它们提炼并提供一种方式来重用从实践经验中获得的设计知识。模式为设计原则提供一种公共的词汇和理解。它们提高了对设计问题及其解决方案的讨论效率。

模式是软件体系结构文档化的一种方式。当设计一个软件系统时，它们可以进行形象描述。模式支持采用定义属性来构造软件。模式提供一个功能行为的基本框架，有助于实现应用程序的功能。模式还清楚描述了软件系统的非功能需求，如可修改性、可靠性、可测试性和可重用性。模式有助于建立一个复杂的不同种类的软件体系结构。每个模式提供组件的预定义集、它们的作用以及它们之间的关系。

一个模式提供了一个问题族的一般解决方案的计划，而不是可以使用的预制模块。模式有助于相似单元的创建。模式有助于解决问题，但它不能提供彻底的方案。

模式有助于管理软件复杂度。每个模式描述一种已被证明的方法以处理指出的问题：所需组件的种类、它们的作用、要隐藏的细节、必须看到的抽象以及每一个是如何工作的，等等。

本书对软件开发的初学者和专家都有帮助。本书可以帮助初学者在适当规模的项目上像专家一样工作，而不需要积累多年的经验。本书可以支持专家去设计具备已定义属性的大型复杂软件，并可以促使他们学习别的专家的经验。

本书力图做到既是教材又是参考指南，帮助软件开发人员以一种新的方式考虑软件体系结构，并提供一些技术来解决特定的再现设计问题。把本书作为软件工程课程的指南，可以给学生提供大型软件设计的完整的新观点。把本书作为参考手册，可以提供全面的技术并随查随用。本书包含了涉及模式实际应用的许多指导原则和约束限制。

本书可作为计算机专业高年级本科生、研究生的软件工程教材或参考书，亦可作为软件开发人员的参考手册。

本书第3、4章由郭福亮翻译，第5~8章由赵皑翻译，其余章节由贲可荣翻译。全书由贲可

荣审校。俞立军、孙宁、王淑雪、吴铁洲、周娟、石国景等参与了本书的翻译工作，在此表示感谢。

由于各种原因，译稿难免存在错误和疏漏。欢迎读者批评指正。

译　者

2002年8月8日

译者介绍

贲可荣　男，1963年8月生，江苏省海安县人。现任海军工程大学教授。

1983年在苏州大学数学系获理学学士学位，1986年在南京大学数学系获理学硕士学位。1986年8月至1990年3月在海军工程大学计算机系任教。1994年6月在国防科技大学获工学博士学位。博士期间，由国防科技大学计算机学院陈火旺院士指导，主修计算机软件。1994年12月任海军工程大学计算机系副教授。1995年起担任计算机应用技术专业硕士生导师、海军工程大学学位评定委员会委员。经教育部批准，2000年3月至2001年3月任武汉大学软件工程国家重点实验室访问学者；2000年被教育部确定为海军首批十名骨干教师之一。2000年12月晋升为教授。

主要译著有《净室软件工程——技术与过程》(电子工业出版社，2001年6月)，《能力成熟度模型(CMM)：软件过程改进指南》(电子工业出版社，2001年7月)，《基于项目的软件工程——面向对象研究方法》(机械工业出版社，2002年6月)，审校《实用软件测试指南》(电子工业出版社，2003年1月)，参加《计算机科学技术百科全书》(清华大学出版社，1998)的编写。先后承担国家自然科学基金项目、国家863项目和军队科研等五个项目。在“中国科学”、“软件学报”、“计算机科学”等刊物和会议发表论文50余篇。成果获军队科技进步奖。主要研究方向：软件可靠性、软件质量保证技术、形式化方法等。兼任中国计算机学会计算机理论专业委员会委员、中国造船工程学会电子技术学术委员会委员，《海军工程大学学报》、《舰船电子工程》等刊物编委。

郭福亮　男，1963年1月生，河北省滦南县人。现任海军工程大学教授。

1982年在海军工程学院数学专业获理学学士学位，1987年在北京航空学院计算机系软件工程研究生班毕业。1987年8月至1996年8月在海军工程学院计算机系任教。2001年1月在华中科技大学计算机学院获硕士学位。2001年9月起在华中科技大学攻读博士学位。由华中科技大学计算机学院卢炎生教授指导，主修计算机软件与理论。1996年12月任海军工程学院华中地区军队院校计算机技术服务中心副教授。1998年起担任计算机应用技术专业硕士生导师。2001年12月晋升为教授。

曾主编过《微型计算机维护维修实用技术》(华中理工大学出版社，1997年11月)，主要译著有《自己动手装网络Linux》(华中理工大学出版社，1998年12月)。在“中国科技文库”、“计算机应用研究”、“图像工程”、“军事”等刊物和会议发表论文20余篇。先后承担多项军队科研项目，其中两项成果获军队科技进步奖。主要研究方向：软件可靠性，数据挖掘技术等。

赵皑　女，1973年1月生，河北张家口人。现任海军工程大学讲师。

1994年在国防科技大学计算机系获工学学士学位，1997年在国防科技大学计算机系获工学硕士学位。1997年3月至今在海军工程大学任教。主要研究方向：计算机网络及应用、多媒体技术。

主要译著有《JAVA 2 类库》（机械工业出版社，2001年1月）。

前　言

本书讨论了软件体系结构的模式，简称模式。近年来，模式已引起广泛关注，讨论模式的形式包括专题讨论会、研究班、电子论坛、期刊论文、专著等。模式团体甚至开始举行它们自己的会议。所有这些有关模式的热情讨论看起来像是达到了“对象波”的顶峰。

为什么关于模式如此令人激动呢？很可能是因为它们在收集熟练的设计者和软件工程师的经验方面起到了“根基”的作用。这样的专家对许多再现设计问题已有了解决方案。模式以易于获得的方式和所希望的良好书写的格式捕获了这些已证实的解决方案。

我们希望本书对软件开发的初学者和专家都有帮助。本书可以帮助初学者在适当规模的项目上像专家一样工作，而不需要积累多年的经验。本书可以支持专家去设计具备已定义属性的大型复杂软件，并可以促使他们学习别的专家的经验。本书会帮助这两种人找到对一个特定设计问题的充分考验的解决方案。

本书力图做到既是教材又是参考指南，帮助软件开发人员以一种新的方式考虑软件体系结构，并提供一些技术来解决特定的再现设计问题。把本书用作软件工程课程的指南，可以给学生提供大型软件设计的完整的新观点。把本书用作参考手册，可以提供全面的技术并随查随用。本书包含了涉及模式实际应用的许多指导原则和约束限制。

以规范形式记录设计知识可以追溯到Christopher Alexander[⊖]。他在建筑学领域率先提出模式概念。他的著作《*The Timeless Way of Building*》说明了如何将模式应用到房屋构建以及邻近地区和整个城市的规划之中。他的工作的基础主题是居住场所的设计，不仅要注重功能性和流行趋势，而且要讲究舒适感和欣慰感。良好设计的建筑可以展现清晰可辨而又难以描述和量化的内在特质。简而言之，这样的建筑拥有“没有名称的质量”。

把这种方法改用到软件工程中的早期试验主要偏重于Alexander的风格。最近软件界经历了寻求更适合软件设计的风格形态的试验。对模式尝试了几种不同的描述形式，但至今没有达成一致。

尽管我们把相当多的精力用于寻找一种描述模式的好方法，但是开发“模式风格”理论并不是本书的主要目标。这当然不是开始模式工作的原动力。在1991年，我们以一种直截了当的方式记录了第一个模式。随着模式系统文档的逐步增长，很快就发现单个模式不能单独存在。模式揭示了丰富的相互关系集。这是出版本书的驱动因素之一，而不是在某个时候把模式文档化并作为系列论文出版。采用图书这种方法的缺点在于，在成书之前需要一个较长的酝酿思考阶段。尽管模式成名已经几十年了，但是花费如此之长的时间来提出一个好的模式描述仍然让我们吃惊。

⊖ Christopher Alexander是一位开业建筑师和城市规划师，同时他也是美国加州大学伯克利分校的建筑学教授以及环境建筑中心的主任。他创建了建筑学的理论，基于模式的构造和使用来进行构建和规划。他的模式理论本身、他的理论实践以及对这种途径的批评和讨论已由牛津大学出版社以系列丛书的形式出版。

其他四位作者经历了同样的现象。1994年秋天，Erich Gamma、Richard Helm，Ralph Johnson和John Vlissides[⊖]出版了重要著作《设计模式——可复用面向对象软件的基础》。尽管设计模式的思想不再新颖，但是“GoF”（简称为“四人帮”）的著作仍然给出了对良好描述的面向对象程序设计模式的最初分类。

我们的方法和“四人帮”的方法略有差别，尽管有许多相似之处并有部分重叠。“GoF”的著作集中于设计层模式，而我们的模式跨越几个抽象层次。范围从高层体系结构模式经设计模式到低层惯用法。我们也关注面向对象以外的问题，并试图结合对模式描述技术的最新认识。我们的整体目标是使用模式来在软件体系结构的更广泛的语境中提供帮助。我们把这种方式叫做面向模式的软件体系结构。我们谈到了模式系统，模式不仅仅被收集在不同种类的容器中，而且也根据相应的标准分组。“GoF”的著作通过把模式划分为“创建型”、“结构型”和“行为型”等开始了分类工作。我们下一步试图根据已获得的更好标准（如交互式和适应性系统、工作的组织、通信和访问控制等）来对模式分组。

我们想要鼓励我们模式系统的用户与其同事共享我们的研究成果。模式共享建立了设计问题的通用词汇表。它使得正在成长的模式团体的成员以更有效的方式来确认、命名和讨论问题及其解决方案。在系统设计中达到“提速”是在工作中使用模式的一个更加重要的理由。

我们的模式系统并不完全。如此多的模式用一本书是不可能记录下来的。随着技术的演化，新的模式也跟着演化。我们希望读者能针对你们的特定需要对我们的模式系统进行扩展、修改和剪裁。缺少的模式应该添加，不需要的模式可以忽略，其他的模式可以更改。

如果你对本书风格和内容方面的改进有任何评论、批评或建议，请大胆提出。我们也欢迎用我们描述的模式来报告经验。你可以写信并通过John Wiley & Sons.出版公司转交给我们，或发电子邮件到patterns@mchp.siemens.de。

我们对于本书包括的绝大多数模式的最初版本已经在因特网上进行了讨论。我们的动机不是为了获得免费广告或赠送模式。相反，我们想支持出版业的一个新趋势，即在刊印之前尽早将材料展示给团体成员，这样对每一个部门都有益。我们欣赏这些经验并感谢所有的参与者。但是这并不意味着我们关闭了本书的公开电子讨论。邮件列表仍然存在并仍然欢迎读者参与。预订指南可以在模式主页上找到。它的URL是：

http://www.hillside.net/patterns/

这个URL也是所有模式方面（比如已出版的和待出版的著作、模式会议、模式论文等）最重要的信息资源。

本书结构

第1章系统地介绍了模式的概念并讨论了模式描述的原则。第2章~第4章给出了我们的模式目录的内容。

⊖ 他们四人常被称为“四人帮”，或简称为“GoF”（Gang-of-Four）。——编辑注

体系结构模式是最高层模式。它们旨在提供整个系统体系结构的构架。第2章给出了来自不同应用领域的8个体系结构模式。

第3章给出了8个设计模式，这些模式分别针对在定义软件系统的整体结构之后遇到的典型问题。例如，我们的设计模式处理以下情况：构建控制复杂性的组件，分布组件间工作量，组织组件间通信。

第4章是模式内容的第三部分也是最后一部分。它处理惯用法，一种与语言相关的模式。但是本书主要谈到其他人的工作而不是对我们自己的惯用法进行文档化，并且仅仅提供了一个惯用法作为一个具体实例。本书没有对我们自己的惯用法进行描述，理由很简单——已经有了很多诸如C++和Smalltalk语言的惯用法可供使用。我们不是仅对这些模式进行改述，我们要有选择地引用原始资源。

在第5章中，我们证实了把模式组织成模式系统是很重要的。这样的系统在如下几个方面对模式的作者和使用者都有帮助：为手头的情形寻找正确的模式，填充模式汇集中的缺口，理解模式之间的关系并演化模式系统。

在第6章中，我们讨论了如何将模式嵌入到软件体系结构中。特别是我们讨论了对于软件体系结构及其基本原则的理解，我们展示了模式是如何支持这些原则的。

第7章讨论模式历史、相关工作和整个模式团体。结束本书前，第8章给出了对于模式未来发展的展望。

本书以附录结尾，附录包括符号表、常用词汇表、参考文献和模式索引。

致谢

我们向所有以不同方式对本书的形成起到过帮助作用的人士表示感谢，不仅因为这是一个惯例，而且更是因为我们真心诚意希望这样做。

我们感谢Joelle Coutaz、Wilhelm Gruber、Claus Jäkel、Doug Lea、Oscar Nierstrasz、Laurence Nigay、Frances Paulisch、Wolfgang Pree、Uwe Steinmüller、John Vlissides和Walter Zimmer，他们对我们工作的早期版本进行了讨论和修订。Ralph Johnson和他在Urbana-Champaign伊利诺斯大学的体系结构读书小组的成员，其中包括John Brant、Michael Chung、Brian Foote、Don Roberts和Joseph Yoder，他们仔细评审了我们的绝大部分模式描述。他们为我们提供了许多评论以及改进建议。我们也感谢山腰组（Hillside Group）的支持和鼓励。

我们在每个模式描述最后以单独一节感谢对这个模式的改进提供过帮助的人们。

特别感谢James Coplien、Joseph Davison、Neil Harrison和Douglas Schmidt。他们详细评审了我们的材料，帮助我们设计和润色本书的最终内容。

我们的夏季学生Marina Seidl和Martin Botzler和我们一起经历了一些早期试验。也特别感谢Franz Kapsner和Hartmut Raffler在管理上的支持，感谢德国慕尼黑西门子公司合作研究和开发部软件工程实验室的支持。

Francis Glassborow和Steve Rickaby力求提高我们有限的英语写作能力，并帮助消除了最差的“德语习语”——这不是一项容易的任务。

最后，我们感谢编辑Gaynor Redvers-Mutton以及John Wiley & Sons出版公司内所有使本书按时出版的人士。

读者指南

本书的编排使读者能从头至尾地阅读。万一你想在本书中选择你自己的阅读路线，可以参考下面给出的提示。

第1章“模式”，给出了软件体系结构的全面解释。一切都建立在这些讨论之上，所以你应该首先阅读这一章。阅读单个模式的顺序取决于你。想掌握某个特定模式的关键思想，你只要阅读它的语境、问题和解决方案等小节。广泛的交叉引用会指导你理解模式间的相互关系。

如果模式对你而言是全新的知识，我们建议你首先阅读基本和简单的模式，即那些易于理解并在许多良好结构的软件系统中出现的模式。例子包括：

- 管道和过滤器体系结构模式
- 代理设计模式
- 转发器-接收器设计模式

你也可以利用本书寻找你当前工程中遇到的设计问题的解决方案。利用第5章“模式系统”中的模式综述作为你研究的指南，再查看那些你选为潜在解决方案的模式的详细描述。

其他章节——第6章“模式和软件体系结构”、第7章“模式团体”和第8章“模式将走向何方”——可以以任何顺序阅读，尽管书中给定的顺序是我们认为最适合读者阅读的。

索引中的页码为英文原书面码，与书中边栏的页码一致。

目　录

译者序

译者介绍

前言

读者指南

第1章　模式 ……1

1.1　什么是模式 ……1

1.2　模式是如何构成的 ……5

1.3　模式类别 ……7

1.3.1　体系结构模式 ……7

1.3.2　设计模式 ……7

1.3.3　惯用法 ……8

1.3.4　与软件开发结合 ……9

1.4　模式间关系 ……9

1.5　模式描述 ……11

1.6　模式和软件体系结构 ……13

1.6.1　模式作为智力构造块 ……13

1.6.2　构造异构体系结构 ……13

1.6.3　模式和方法 ……14

1.6.4　实现模式 ……14

1.7　总结 ……14

第2章　体系结构模式 ……16

2.1　引言 ……16

2.2　从混沌到结构 ……17

2.2.1　层 ……18

2.2.2　管道和过滤器 ……30

2.2.3　黑板 ……41

2.3　分布式系统 ……55

2.3.1　代理者 ……56

2.4　交互式系统 ……70

2.4.1　模型-视图-控制器 ……70

2.4.2　表示-抽象-控制 ……83

2.5　适应性系统 ……97

2.5.1　微核 ……98

2.5.2　映像 ……110

第3章　设计模式 ……128

3.1　引言 ……128

3.2　结构化分解 ……129

3.2.1　整体-部分 ……129

3.3　工作的组织 ……141

3.3.1　主控-从属 ……142

3.4　访问控制 ……151

3.4.1　代理 ……152

3.5　管理 ……160

3.5.1　命令处理器 ……160

3.5.2　视图处理程序 ……169

3.6　通信 ……178

3.6.1　转发器-接收器 ……179

3.6.2　客户机-分配器-服务器 ……189

3.6.3　出版者-订阅者 ……197

第4章　惯用法 ……201

4.1　引言 ……201

4.2　惯用法能够提供什么 ……201

4.3　惯用法与风格 ……202

4.4　在哪里可以发现惯用法 ……204

4.4.1　计数指针 ……205

第5章　模式系统 ……210

5.1　什么是模式系统 ……210

5.2　模式分类 ……211

5.2.1　模式类别 ……212

5.2.2　问题类别 ……212

5.2.3　分类图式 ……213

5.2.4　比较 ……214

5.3 模式选择 ……214
5.4 作为实现指南的模式系统 ……216
5.5 模式系统的演化 ……218
5.5.1 模式描述的演化 ……218
5.5.2 作者研讨会 ……218
5.5.3 模式采掘 ……219
5.5.4 新模式集成 ……219
5.5.5 删除过时模式 ……220
5.5.6 扩展组织图式 ……220
5.6 总结 ……222
第6章 模式和软件体系结构 ……223
6.1 引言 ……223
6.1.1 软件体系结构 ……223
6.1.2 组件 ……224
6.1.3 关系 ……225
6.1.4 视图 ……225
6.1.5 功能属性和非功能属性 ……226
6.1.6 软件设计 ……227
6.1.7 小结 ……227
6.2 软件体系结构中的模式 ……227
6.2.1 方法学 ……228
6.2.2 软件过程 ……228
6.2.3 体系结构风格 ……229
6.2.4 框架 ……230
6.3 软件体系结构启用技术 ……231
6.3.1 抽象 ……231
6.3.2 封装 ……231
6.3.3 信息隐藏 ……232
6.3.4 模块化 ……232
6.3.5 事务分离 ……232
6.3.6 耦合和内聚 ……232
6.3.7 充分性、完整性和原始性 ……233
6.3.8 策略和实现的分离 ……233
6.3.9 接口和实现的分离 ……233
6.3.10 单一引用点 ……234
6.3.11 分而治之 ……234
6.3.12 小结 ……234
6.4 软件体系结构的非功能属性 ……234
6.4.1 易修改性 ……235
6.4.2 互操作性 ……236
6.4.3 效率 ……236
6.4.4 可靠性 ……236
6.4.5 可测试性 ……237
6.4.6 可重用性 ……237
6.5 总结 ……238
第7章 模式团体 ……239
7.1 起源 ……239
7.2 领军人物和他们的著作 ……240
7.3 团体 ……240
第8章 模式将走向何方 ……243
8.1 模式采掘 ……243
8.1.1 软件体系结构的模式 ……243
8.1.2 组织模式 ……244
8.1.3 特定领域模式 ……244
8.1.4 模式语言 ……244
8.2 模式组织与索引 ……244
8.3 方法与工具 ……245
8.4 算法、数据结构和模式 ……246
8.5 形式化模式 ……247
8.6 最后评述 ……247
符号 ……248
词汇表 ……252
参考文献 ……258
模式索引 ……270
索引 ……273

第1章 模　式

……很久以前，飘荡在广漠无际的太空中的一组随意的原子受到严重伤害，并以特别不同的模式将它们粘附在一起。这些模式迅速学会复制自身（这是模式如此不凡的部分原因）且继续对它们飘过的每个行星造成巨大的麻烦。这是生命在宇宙中的开始……

Douglas Adams，《*The Hitchhiker's Guide to the Galaxy*》

模式有助于我们利用熟练的软件工程师的集体经验来构建软件。它们捕捉软件开发中现存的、充分考验的经验，再用于促进好的设计实践。每个模式处理一个软件系统的设计或实现中一种特殊的重复出现的问题。模式可以用来构建具有特定属性的软件体系结构。

本章对软件体系结构的模式是什么，以及它们如何帮我们构建软件等问题给出了一个全面深入的说明。

1.1 什么是模式

当专家求解一个特殊问题时，他们一般不会发明一种和已有解决方案完全不同的方案来处理这个问题。他们往往想起已解决过的相似问题，并重用其解法的精华来解决新问题。这种“专家行为”，即同时考虑一对问题-求解方案，这一点在很多不同领域中都是共同的，比如建筑学[Ale79]、经济学[Etz64]和软件工程[BJ94]领域就是如此。这是解决任何一类问题或社会交互的一种自然手段[NS72]。

下面是取自建筑学领域的一对问题-解决方案的一个很好的直观例子：

例子　窗户位置[AIS77]

每个人都喜爱靠窗的座位、凸窗、有着低窗台的大窗户、舒适的椅子放在它们边上……。一个房间如果没有一个这样的位置，你自然不会觉得舒适或完美……。

如果房间没有“位置”适宜的窗户，房里面的人会受到以下两种强制条件的折磨：

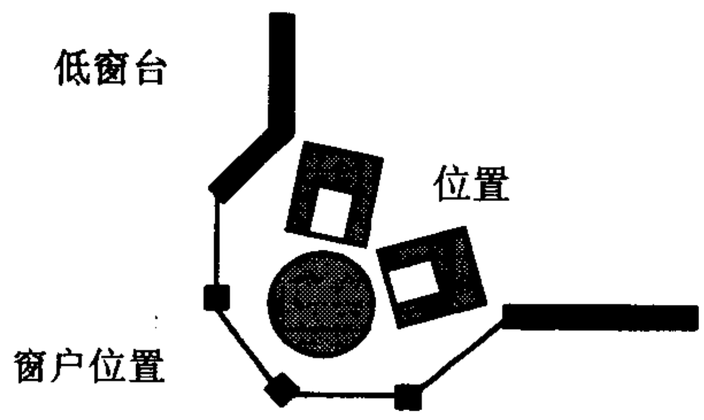
(1) 他想舒适地坐下来。

(2) 他想朝向光线。

很明显，如果让你感到舒适的位置——房间中你最想坐的位置——距离窗户较远，你会无法克服这种冲突。

因此，如果你每天在那个房间都要呆上一段时间，那么至少让这个房间的一个窗户处于一个适宜的“窗户位置”。

从特定的一对问题-解决方案中进行抽象并提炼出公共要素可以形成模式：“这些成对的问题-解决方案倾向于形成相似的问题-解决方案的系列，且每一个系列体现一种在问题和解决方



案中的模式”[Joh94]。建筑师Christopher Alexander在《建筑的永恒方法》(The Timeless Way of Building)[Ale79]一书中，定义了如下的术语“模式”(*pattem*)：

> 每个模式是一条由三部分组成的规则，它表示了一个特定环境、一个问题和一个解决方案之间的关系。
>
> 作为建筑领域的一个元素，每个模式是一个特定环境，在该环境中重复出现的一个特定强制条件体系(system of forces)以及给这些强制条件自求解提供一个特定的空间配置之间的相互关系。
>
> 作为语言的一个元素，一个模式是一个说明，它说明了如何使用这个空间配置，不断求解给定的强制条件体系，只要环境是与它相关的。
>
> 简单地说，模式是在同一时间里发生在世界上的一件事物和如何创建这个事物以及我们何时必须创建它的规则。它既是一个过程，又是一个事物；既是一个活生生事物的描述，又是产生那个事物的过程描述。

我们在软件体系结构中也发现了许多模式。软件工程中的专家从实践经验中获知这些模式并在开发具有特定属性的应用中遵循这些模式。专家们使用模式有效而出色地解决设计问题。在详细讨论这些内容之前，让我们看一个著名的例子：

例子 模型-视图-控制器(Model-View-Controller, MVC)(参见2.4.1节)

在开发人机界面软件时考虑这种模式。

用户界面容易改变需求。例如，当扩展一个应用程序的功能时，必须修改菜单以访问新功能，而且用户界面必须适合特定的客户。一个系统可能经常以不同的“视觉和感觉”标准运行在另一个平台上。甚至升级一个新的窗口系统版本就意味着需要更改你的代码。总之，经久耐用的系统的用户界面是一个我们不断变动的目标。

3

如果用户界面和功能核心紧紧交织在一起，则建立一个具有所要求灵活性的系统是昂贵的而且易于出错。这导致开发和维护几个实质上不同的软件系统，一个系统针对一个用户界面的实现。确保改动随后散布到许多模块中。总之，当开发这样一个交互软件系统时，你必须要考虑两方面的因素：

- 用户界面应该是易于改变的，在运行期间也是可能改变的。
- 用户界面的修改或移植应该不影响应用程序的功能核心的代码。

为解决这个问题，把一个交互应用程序划分成三个部分：处理、输出和输入：

- 模型(*model*)组件封装核心数据和功能。模型独立于特定的输出表示或输入行为。
- 视图(*View*)组件给用户显示信息。视图获得来自模型的待展示数据。一个模型可以有多

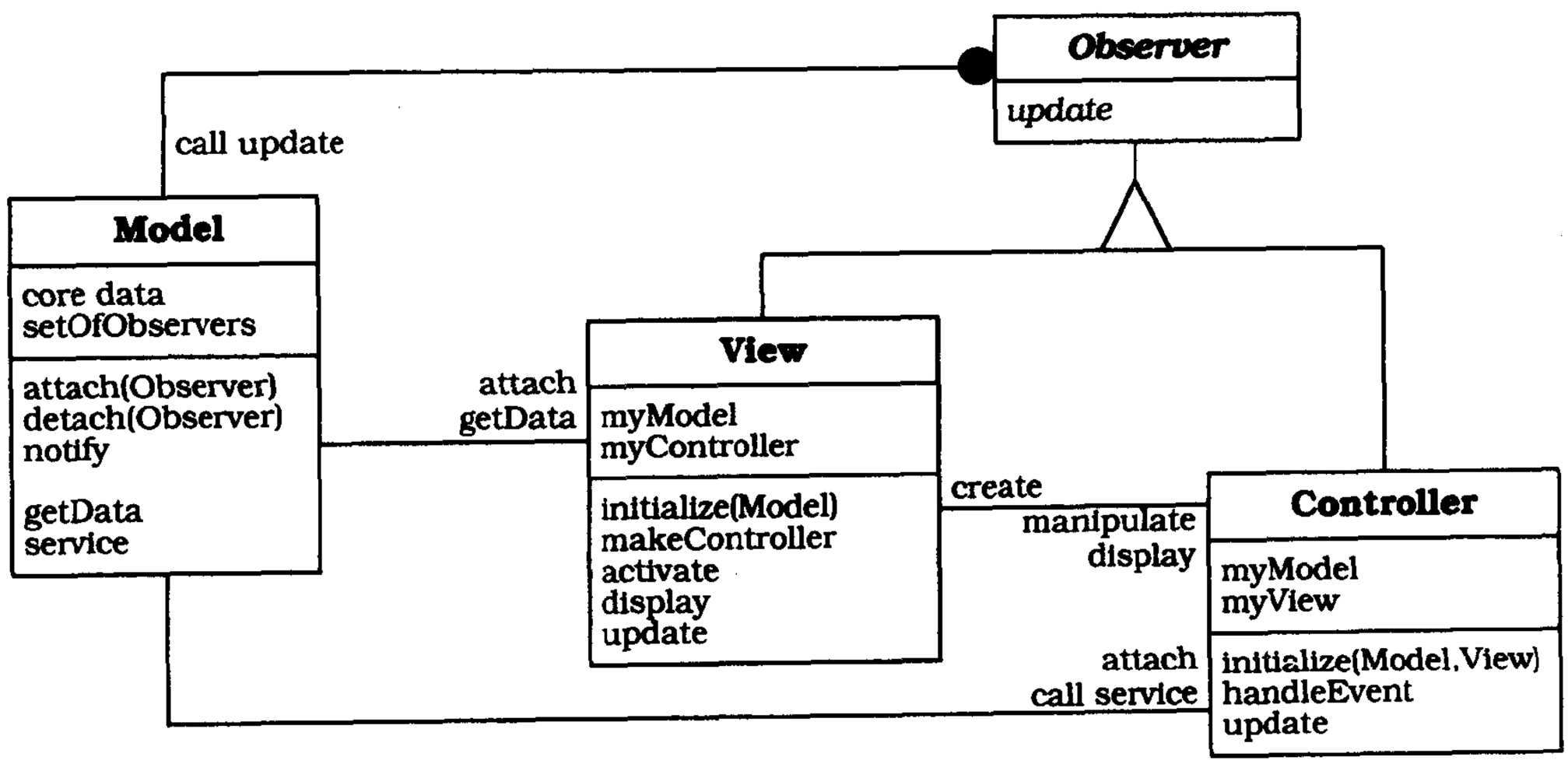
个视图。

- 每个视图都有一个相关的控制器（*cantroller*）组件。控制器接受输入，通常就是像鼠标移动、鼠标点击或键盘的输入等事件。事件被转换为服务请求，而服务请求又被传送给模型或视图。用户仅仅通过控制器与系统交互。

将模型与视图及控制器组件分离允许同一模型有多个视图。如果用户通过一个视图的控制器改动了模型，则所有其他依赖于这个数据的视图应该反映这个改动。为达到这一目的，数据一旦改动，模型就通知所有视图。视图依次从模型获得新数据并更新显示信息。

这种解决方案可以改动应用程序的一个子系统而不对其他子系统产生重要影响。例如，把一个非图形化用户界面改为图形化用户界面不需要修改模型子系统。你可以添加一个对新的输入设备的支持而不需要影响信息显示或功能核心。该软件的所有版本可以处理同一模型独立于特定“视觉和感觉”的子系统。 4

下面的OMT（对象模型技术）类图[⊖]描述了这种解决方案：



从下面这个介绍性的例子中，我们可以导出软件体系结构模式的几个属性[⊖]：

一个模式关注一个在特定设计环境中出现的重现设计问题，并为它提供一个解决方案。在我们的例子中，问题是支持用户界面的可变性。开发人机交互软件系统时，这个问题就会出现。你可以通过严格区分职责来解决这个问题：将应用程序的核心功能从其用户界面中分离出来。

各种模式用文档记录下现存的经过充分考验的设计经验。它们不是人工发明或创造的。它们“提炼并提供一种手段来重用从有经验的实践者获得的设计知识”[GHJV93]。那些熟悉足够 5

⊖ 对于分析和设计方法对象建模技术（OMT）及其表示法的小结，参见本书第8章之后的“表示法”。详细描述可见[RBPEL91]。

⊖ 如果不另外声明，我们把术语“模式”（*pattern*）和“软件体系结构模式”（*pattern for software architecture*）作为同义词使用。

多的模式的人“能立即把它们应用到设计问题中而不需要重新发现它们”[GHJV93]。取代仅存在于少数专家头脑中的知识，模式使得这类知识更容易获得。你可以使用这样的专家知识为一个特定任务来设计高质量软件。例如，模型-视图-控制器模式体现了开发人员许多年来开发交互式系统的经验。许多著名的应用程序已经应用了模型-视图-控制器模式——对许多Smalltalk应用程序来说，它是经典的体系结构，并构成了几个应用程序框架（如MacApp[Sch86]或ET++[WGM88]）的基础。

模式明确并指明处于单个类和实例层次或组件层次之上的抽象[GHJV93]。典型情况下，一个模式描述几个组件、类或对象，并详细说明它们的职责和关系以及它们之间的合作。所有的组件共同解决模式关注的问题，而且通常比单个组件更有效。例如，模型-视图-控制器模式描述了三个合作组件的三元组，每个MVC三元组也同系统的其他的MVC三元组合作。

模式为设计原则提供一种公共的词汇和理解[GHJV93]。如果仔细选择模式名称，则这个名称会成为广泛传播的设计语言的一部分。它有助于设计问题及其解决方案的有效讨论。它们去掉了对一个特殊问题用冗长而复杂的描述来解释其解决方案的需求。代之以你可以使用一个模式名称，并解释解决方案的哪个部分对应模式的哪个组件，或者对应它们之间的哪个关系。例如，自20世纪80年代早期，名称“模型-视图-控制器”和相关模式就被Smalltalk团体所熟悉，并被许多软件工程师所使用。当我们说“软件体系结构遵循模型-视图-控制器模式”时，我们所有熟悉该模式的同事立即就想到了它的基本结构的要领和应用的特性。

模式是为软件体系结构建立文档的一种手段。设计一个软件系统时，这些模式可以在你脑海中描述出一个构想（Vision）。这避免了在扩展和修改初始体系结构时或修改系统代码时违背这个最初的构想。例如，如果你知道一个系统是根据模型-视图-控制器模式构建的，你也就知道如何给它扩展一个新功能：使核心功能与用户输入和信息显示分离。

6

模式支持用已定义的属性来构造软件。模式提供一个功能行为的基本骨架，从而有助于实现应用程序的功能。例如，有的模式用于维护合作组件之间的一致性，有的模式提供透明对等的进程间通信。另外，模式清楚描述了软件系统的非功能需求，如可更改性、可靠性、可测试性或可重用性。例如，模型-视图-控制器模式支持用户界面的可更改性和核心功能的可重用性。

模式有助于建立一个复杂的和异构的软件体系结构。每个模式提供组件、作用以及相互关系的预定义集。它可以用于指定具体软件结构的特定方面。模式“可作为更复杂设计的构造块”[GHJV93]。这种使用预定义设计人工制品的方法提高了设计的速度和质量。理解并应用良好书写的模式比起你寻找自己的方案来要节省时间。这并不是说样本模式一定会比你自己的方案好，但是，至少如本书所介绍的那样一个模式系统有助于你评价和评估方案的选择。

但是，尽管模式确定了解决一个特定设计问题的基本结构，但是它还是没有给出完整详细的方案。一个模式提供了某问题族的一般解决方案的图式（Schema），而不是可以使用的预制模块。你必须根据当前设计问题的特定需求来实施这个图式。模式有助于相似单元的创建。这些单元在更宽泛的结构上是相似的，但在详细层面上往往很不一样。模式有助于解决问题，但它不能提供完整的解决方案。

7

模式有助于管理软件复杂度。每个模式描述一种用来处理所关注问题的已证明是可行的方法，这些问题包括：所需组件的种类、它们的作用、要隐藏的细节、应为可视化的抽象，以及

各要素是如何工作的，等等。当你遇到有一个模式覆盖的具体设计的情形下，就没有必要浪费时间为你的问题创建一种新的解决方案。如果你正确实现了模式，那么你就可以依靠它提供的方案。例如，模型-视图-控制器模式有助于你分离一个软件系统的不同用户界面方面并为它们提供适当的抽象。

我们以如下定义来结束本节：

一个软件体系结构的模式描述了一个出现在特定设计语境中的特殊的再现设计问题，并为它的解决方案提供了一个经过充分验证的通用图式。解决方案图式通过描述其组成组件、它们的责任和相互关系以及它们的协作方式来具体指定。

1.2 模式是如何构成的

前一节中进行的讨论使我们采纳一个构成每个模式基础的三部分图式：

语境：问题出现的场景。

问题：在那个语境中出现的再现问题。

解决方案：已被证实的问题的解决方案。

图式作为一个整体表示一类规则，用来建立在一个给定语境、这个语境中出现的一个特定问题以及对这个问题的适当解决方案之间的关系。这个图式的三个部分是紧密结合的。但是，为了详细了解这个图式，我们必须阐明语境、问题和解决方案的具体含义。

1. 语境

通过描述问题提出的场景，语境扩展了清晰的问题-解决方案的二分法。一个模式的语境可以相当概括，例如“开发具有人机界面的软件”。另一方面，语境可以结合特定模式，例如“实现模型-视图-控制器三元组的变更-传播机制”。 8

为一个模式说明正确的语境是困难的。我们发现确定一个模式可能应用到的所有场景，不论是一般的还是特殊的，实际上是不可能的。一个更实际的方法是列出特殊模式关注问题可能出现的所有已知场景。这并不保证包括与一个模式相关的所有场景，但至少它给出了有价值的指导。

2. 问题

模式描述图式的这个部分描述了在给定语境中重复出现的问题。它以一个一般的问题规格说明开始，抓住它的本质——我们必须解决的具体设计问题是什么？例如，模型-视图-控制器模式关心用户界面经常变化的问题。这个一般的问题陈述用一个强制条件（*force*）集来表示。该词最初是从建筑学和Christopher Alexander那里借用来的，模式组织使用术语“强制条件”来说明问题要解决时应该考虑的各个方面，比如：

- 解决方案必须满足的需求——例如，对等进程间通信必须是高效的。
- 你必须考虑的约束——例如，进程间通信必须遵循特定协议。
- 解决方案必须具有希望的特性——例如，软件更改应该是容易的。

前一节的模型-视图-控制器模式指出了两个强制条件：它必须易于修改用户界面，但软件的功能核心不能被修改所影响。

一般地，强制条件从多个角度讨论问题并有助于你了解它的细节。强制条件可以相互补充或相互矛盾。例如，系统的可扩展性与代码的最小化构成了两个相互矛盾的强制条件。如果希望你的系统可扩展，那么你应倾向于使用抽象超类。如果你想使代码最小化，例如用于嵌入式系统，你就不能承受抽象超类的奢侈。但更重要的是，强制条件是解决问题的关键。它们平衡得愈好，对问题的解决方案就愈好。所以，强制条件的详细讨论是问题陈述的重要部分。

9

3. 解决方案

模式的解决方案部分给出了如何解决再现问题，或者更恰当地说是如何平衡与之相关的强制条件。在软件体系结构中，这样的解决方案包括两个方面。

第一，每个模式规定了一个特定的结构，即元素的一个空间配置。例如，模型-视图-控制器模式的描述包括以下语句："把一个交互应用程序划分成三部分：处理、输出和输入。"

该结构关注解决方案的静态方面。由于这样的结构可以被看成是微型体系结构[GHJV93]，因此和任何软件体系结构一样，它由组件及其相互关系构成。在这个结构里，组件作为构造块，每个组件有一个已定义的责任。组件之间的关系决定它们的位置。

第二，每个模式规定了运行期间的行为。例如，模型-视图-控制器模式的解决方案部分包括以下陈述："控制器接收输入，而输入往往是鼠标移动、点击鼠标按键或键盘输入等事件。事件转换成服务请求，这些请求再发送给模型或视图。"

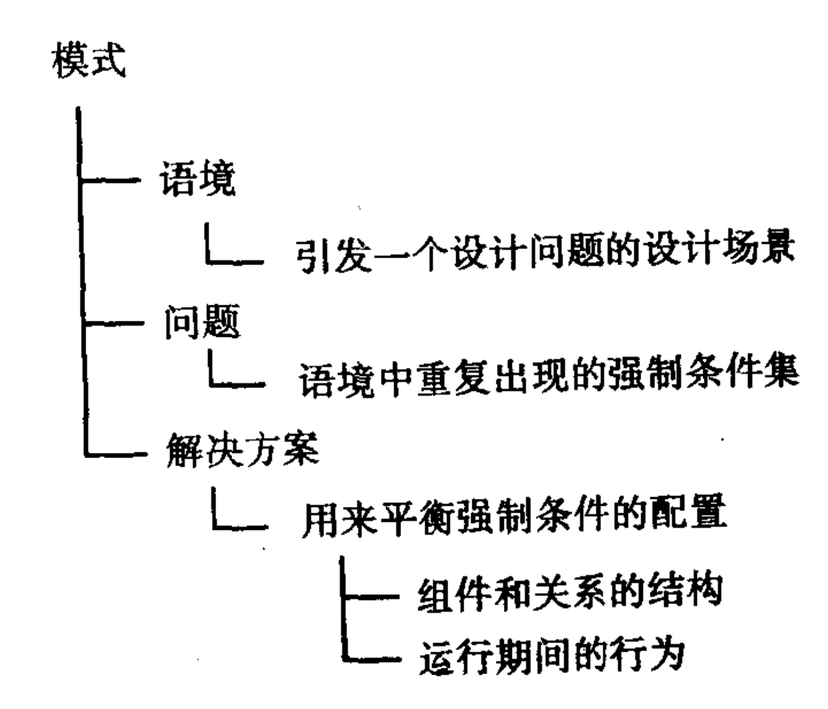
运行期间的行为关注解决方案的动态方面。模式的参与者是如何协作的？它们之间工作是如何组织的？它们彼此如何通信？

解决方案不必解决与问题相关的所有强制条件，注意到这一点很重要。可以集中于特殊的强制条件，而对于剩下的强制条件进行部分解决或完全不解决，特别是强制条件相互矛盾时。

正如我们在前一节所提到的，模式提供一个解决方案图式而不是完整详细的人工制品或蓝图。你应该能在多次实现中重用这些解决方案，但它的本质仍要保留。模式是智力构造块。应用一个模式后，一个体系结构应该包括为由模式指定的角色提供一个特殊结构，但要按目前问题的特殊需要进行调整和剪裁。一个给定模式的两个实现不可能是完全相同的。

10

下图总结了整个图式：



这个图式抓住了模式独立于其领域的本质。把它用作描述模式的模板看上去是显然的。它早已成为许多模式描述的基础，正如在[AIS77]、[BJ94]、[Cope94c]、[Cun94]和[Mes94]中所给出的。这使我们确信上面的形式给理解、共享和讨论一个模式带来了便利。

1.3 模式类别

对现有模式的更近一步的观察揭示它们包含各种范围的度量和抽象。有些模式有助于把一个软件系统分解成子系统。另一些模式支持子系统和组件的细化或它们之间关系的细化。其他模式有助于实现特定编程语言中的特殊设计方面。模式的范围也从与领域无关的模式，比如去耦交互组件模式，遍及到关注特定领域的模式，如商业应用中的交易策略或电信中的呼叫路由。 11

为了细化我们的分类，我们把模式分成三种类型：

- 体系结构模式
- 设计模式
- 惯用法

每一种类型都由具有相似规模或抽象程度的模式组成。

1.3.1 体系结构模式

根据一些整体构建原理来建立可行的软件体系结构。我们用体系结构模式来描述这些原理。

体系结构模式（*architectural pattern*）表示软件系统的基本结构化组织图式。它提供一套预定义的子系统，规定它们的职责，并包含用于组织它们之间关系的规则和指南。

体系结构模式可作为具体软件体系结构的模板。它们规定一个应用的系统范围的结构特性，以及对其子系统的体系结构施加的影响。所以体系结构模式的选择是开发一个软件系统时的基本设计决策。

本章开头的模型-视图-控制器模式是体系结构模式最著名的例子之一。它为交互软件系统提供了一种结构。

1.3.2 设计模式

软件体系结构的子系统，以及它们之间的关系，通常由几个更小的体系结构单元构成。我们用设计模式来描述它们。 12

设计模式（*design pattern*）提供一个用于细化软件系统的子系统或组件，或它们之间关系的图式。它描述通信组件的公共再现结构，通信组件可以解决特定语境中的一个一般设计问题[GHJV95]。

设计模式是中等规模的模式。它们在规模上比体系结构模式小，但又独立于特定编程语言或编程范例。设计模式的应用对软件系统的基础结构没有影响，但可能对子系统的体系结构有

较大影响。

许多设计模式提供分解更复杂的服务或组件的结构。其余的设计模式则关心它们之间有效的合作，比如下面的模式：

1. 名称

观察者[GHJV95]或出版者-订阅者。

2. 语境

一个组件使用另一个组件提供的数据或信息。

3. 问题

改变组件的内部状态可能引入合作组件之间的不一致性。为重建一致性，我们需要一种机制来交换这种组件间的数据或状态信息。

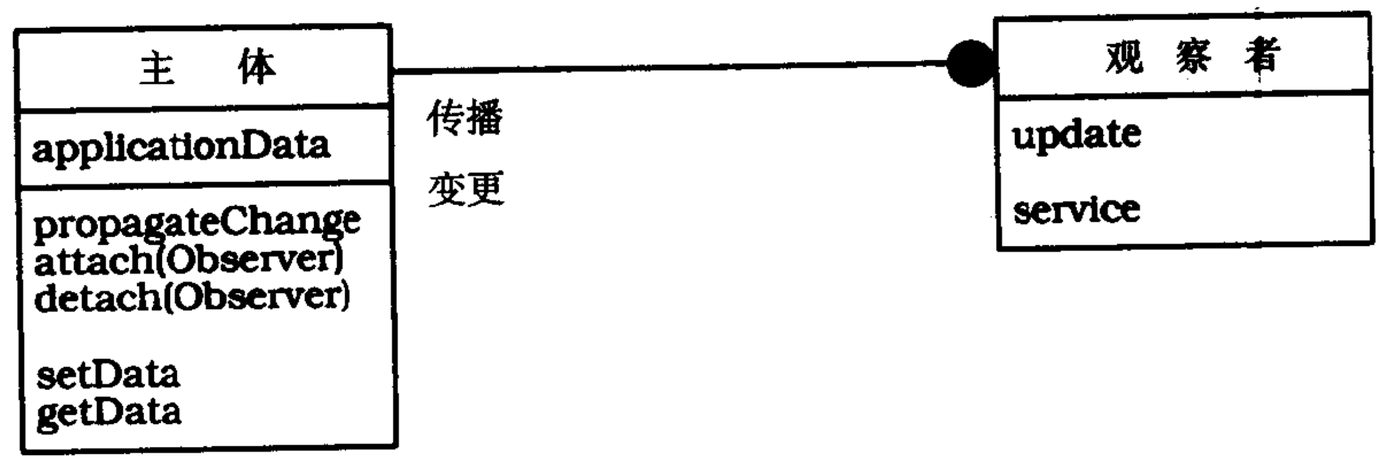
有两种强制条件与该问题相关：

- 组件应该松散耦合——信息提供者不应该依赖于其协作者的细节。
- 依赖于信息提供者的组件并不是预先知道的。

4. 解决方案

实现信息提供者之间的变更-传播机制——目标（*subject*）——和依赖于它的组件——观察者（*observer*）。观察者可以用这种机制动态地注册或取消注册。当主体改变其状态时，它启动变更-传播机制来重建与所有已注册的观察者的一致性。变更通过请求一个特定的对所有观察者公用的更新功能来传播。为实现变更-传播——从主体到观察者的数据和状态信息的传递——你可以使用拉-模式（*pull-model*），推-模式（*push-model*），或它们的结合。

13



1.3.3 惯用法

惯用法处理特定设计问题的实现。

惯用法（*idiom*）是具体针对一种编程语言的低层模式。惯用法描述如何使用给定语言的特征来实现组件的特殊方面或它们之间的关系。

惯用法代表最低层模式。它们关注设计和实现方面。

大多数惯用法是针对具体语言的——它们捕获现有的编程经验。经常出现这样的情况：同一个惯用法对不同的语言看起来不一样，有时候一种惯用法对一种编程语言有用而对另一种语言

却无意义。例如，C++团体使用引用-计数惯用法来管理动态分配的资源；而Smalltalk提供无用单元收集机制，所以不需要这样的惯用法。

下面的例子处理C++中的关键性操作：赋值。模式被称为计数体，其描述大部分来自于[Cope94a]。后面我们描述了计数指针模式，它包含计数体模式作为一个变体。 14

1. 名称

计数体[Cope94a]。

2. 语境

类的接口从它的实现中分离出来。一个句柄类描述与用户的接口类。其他的类包括实现，称为主体（*body*）。句柄传送成员函数调用给主体。

3. 问题

C++中的赋值是作为逐项赋值递归定义的，递归终止时复制。在Smalltalk中，如果捆绑了复制，它会更高效并且语言的精髓会更多。详细地说，你需要平衡三个强制条件：

- 在存储需求和处理时间这两个方面上，主体复制都是昂贵的。
- 通过使用指针和引用可以避免复制，但这留下一个问题——谁负责清理对象？它们也在内部类型和用户定义类型间留下了用户可见的区别。
- 如果通过句柄之一修改共享主体，则有关赋值的共享主体是语义错误的。

4. 解决方案

往主体类中添加一个引用计数有助于内存管理。将内存管理加入到句柄类，特别是加入到初始化、赋值、复制和销毁的实现中。通过使主体自复制，减少初始主体的引用计数，可以修改主体的状态以便中断主体共享，这是任何修改操作的职责。

该解决方案避免没有理由的复制，从而导致更高效的实现。当主体状态通过任何一个句柄被修改时共享被中止。共享在参数传输的更通用的场合中得到保留。避免特殊指针和引用类型，Smalltalk语义与此相近。无用单元收集可以基于这种模型来实现。

1.3.4 与软件开发结合

理想情况下，这种分类有助于你为给定的设计问题预先选定潜在有用的模式。它们与重要的软件开发活动相关。体系结构模式可以用在大粒度设计的开始，设计模式可以用在整个设计阶段，惯用法可以用在实现阶段。关于这些问题的更详细的讨论可以在5.2节“模式分类”中找到，那里还讨论了其他分类图式。 15

1.4 模式间关系

对许多模式的进步考察揭示出：不管初始印象如何，它们的组件和关系并不总是像开始出现的那样是“原子的”。一个模式解决一个特定问题，但它的应用会带来新的问题。其中一些问题可以被其他模式解决。所以，特殊模式中的单个组件和关系可以由更小的模式来描述，它们

可以通过包含它们的更大的模式来集成。

例子

模型-视图-控制器模式的细化

模型-视图-控制器模式把核心功能从人机交互中分离出来以提供可调整的用户接口。但是，应用这种模式带来了一个新问题。视图，有时候甚至是控制器，都依赖于模型状态。它们之间的一致性必须得到维护，即一旦模型状态改变了，我们必须更新所有与其相关的视图和控制器。但是，我们不能失去改动用户接口的能力。前一节中的观察者模式有助于我们解决这个问题——该模型体现了主体的角色，而视图和控制器担任观察者的角色。 ❑

大多数软件体系结构模式产生的问题可以用更小的模式来解决。模式往往并不孤立存在。Christopher Alexander使用带有理想主义的术语来进行阐述："每个模式依赖于它包含的更小模式和包含它的更大模式"[Ale79]。

16 一个模式也可以是另一个模式的变体。一般认为，一个模式与它的变体描述用来解决非常相似的问题的解决方案。这些问题往往在包含它们的一些强制条件之间变化，而不是一般特征。这将在下面的例子中进行描述。

例子

模型-视图-控制器模式的文档-视图变体

考虑一个使用模型-视图-控制器模式的交互文本编辑器的开发。在这样一个文本编辑器中，很难将视图功能与控制器功能分开。假设你用鼠标选择了文本并把它从常用字体变为黑体。文本选择是一个控制器动作，它不会导致对模型的变动。选中的文本仅仅作为另一个控制器动作的输入，这里改变的是选中文本的外观。但是，文本选择是可见的——选中文本以高亮显示。在一个严格的模型-视图-控制器结构中，控制器必须能够自身实现这个"类似视图"的行为，或者必须与视图协作，所选中的文本出现在该视图中。两种解决方案都需要增加一些不必要的实现开销。

在这样的情形下，应用模型-视图-控制器模式的文档-视图变体会更好一些，后者把视图和控制器功能统一在一个组件中，即文档-视图模式的视图中。文档组件直接对应模型-视图-控制器三元组的模型。但是，在使用文档-视图变体时，我们失去了独立更改输入输出功能的能力。 ❑

模型也可以在处于同一个抽象层次的更复杂的结构中进行结合。这在原先的问题包含了可以由单个模式平衡的更多强制条件时出现。在这种情况下，应用几个模式可以解决这个问题。每个模式解决强制条件的一个特定子集。

例子

透明的对等进程间通信

假设你必须使用高性能对等进程间通信来开发一个分布式应用程序。必须平衡下面的强制条件：

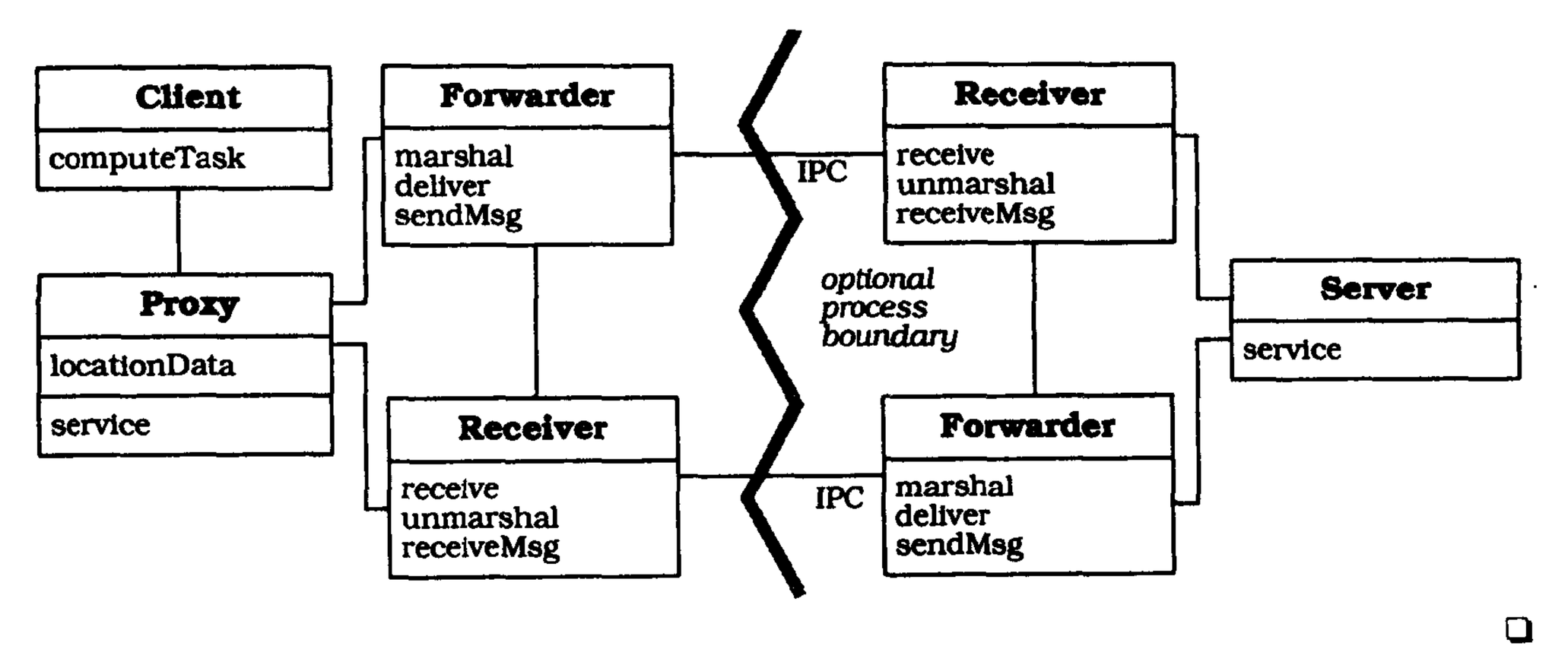
- 进程间通信必须是高效的。我们不希望把时间花在搜索远程服务器上。
- 希望从特定的进程间通信机制中独立出来。这种机制必须是可交换的，同时又不对客户机或服务器构成影响。

17

- 客户机不应该知道或者不应该依赖于它们的服务器的名称和位置。取而代之，它们彼此应该就像在同一个进程中那样进行通信。

任何孤立的单个模式都解决不了这个问题，但两个模式的结合可以达到这个目的。转发器-接收器模式决定了第一个和第二个强制条件，它提供了一种通用接口来发送和接收跨越进程边界的消息和数据。该模式隐藏了具体的进程间通信机制的细节。替换该机制仅影响系统的转发器和接收器。另外，模式为服务器提供了名称到地址的映射。

代理模式决定了第三个强制条件。在该模式中，客户机与处于同一进程中的服务器的代表通信。这个代表，即远程代理（*remote proxy*），它知道服务器的细节（如服务器的名称），并向服务器发送每一条请求。



所有这三种关系——细化、变体和组合——都有助于有效使用模式。细化支持模式的实现，组合有助于组成复杂的设计结构，变体有助于在给定设计情形下选择正确模式。

你可以在[Zim94]中找到模式间关系的补充讨论。 18

1.5 模式描述

如果我们要理解和讨论模式，就必须以适当形式描述模式。好的描述有助于我们立即抓住模式的本质——模式关心的问题是什么，以及提出的解决方案是什么？好的模式还给我们提供实现一个模式的必要细节，并考虑它的应用效果。

模式也应该以统一方式来描述。这有助于我们对模式进行比较，尤其在我们为一个问题寻求可选择的解决方案时。

我们在本章前面所讨论的基本的语境-问题-解决方案结构以一个好的起点为我们提供了一个满足上述需求的描述格式。它捕获了模式的主要特征，为你提供了关键思想。因此我们将我们的描述模板基于这种结构。

然而，仅仅基于语境-问题-解决方案图式来描述模式是不够的。如果我们要共享并讨论模

式，就必须对模式命名——最好采用比较直观的名称。这种名称也要表达一个模式的本质。一个好的模式名称是很重要的，它将成为设计词汇表的一部分[GHJV93]。

我们为模式描述加入一个介绍性的例子来帮助解释这个问题及其相关的强制条件。当讨论一般模式的解决方案和实现情况时，我们反复引用这个例子。

我们进一步使用图表和场景来解释解决方案的静态方面和动态方面。我们也包含了模式的实现指南，这些指南有助于我们将一个给定的体系结构转化成使用模式的一个体系结构。我们加入实例代码并列出模式的成功应用来增强其可信度。

我们也描述模式的变体。变体为我们提供解决问题的其他可选方案。但是，我们不像原始模式那样详细地描述变体——我们仅进行简短描述。

19

有关模式优点和潜在不足的讨论突出了应用的效果。这提供给我们信息，以帮助我们决定使用哪种模式来为特定问题提供一个适当的解决方法。我们也交叉引用其他的相关模式，或者因为它们细化了当前模式，或者因为它们关心一个相似的问题。

通过把所有可获得的和合适的信息陈列出来，我们应该能理解一个模式，并能正确地应用和实现它。

最后，我们要感谢对设计特殊模式有帮助的相关人员。写模式是困难的。获得一个清楚的模式描述需要好几轮的评审和修改。世界上许多专家对我们这项工作给予了帮助，在此我们致以诚挚的感谢。如果我们知道模式“发现者”——即最先描述模式的人的名字，我们也对他们表示感谢。

所以我们的模式描述模板如下：

名称 模式的名称和一个简短的摘要。

别名 模式的其他名称，如果知道的话。

例子 用来说明问题存在和需要模式的一个真实世界的例子。

在整个描述中，在需要或有用的地方，我们用例子来说明解决方案和实现的各个方面。有关例子的文本用➥符号标志开始，用❑符号标志结束。

语境 模式可以应用的情形。

问题 模式解决的问题，包括其相关强制条件的讨论。

解决方案 以该模式为基础的基本解决方案原理。

结构 模式结构方面的详细的规格说明，包括针对每个参与组件和一个OMT类图[RBPEL91]的CRC卡片[BeCu89]。（参看本书后面“符号”中的相关部分）

20

动态特性 描述模式运行期间行为的典型场景。

我们进一步用对象消息序列图表来描述相关场景（参看本书后面“符号”中的相关部分）。

实现 实现模式的指南。

这些仅仅是建议，而不是一成不变的规则。通过添加不同的、额外的或更详细的步骤，或通过对步骤的重新排序，我们可以改写实现以满足你的要求。我们给出C++、Smalltalk、Java或pSather的代码片断来描述可能的实现，通常是描述例子问题的细节。

已解决的例子 针对解决没有包括在“解决方案”、“结构”、“动态特性”和“实现”小节中的例子的一些重要方面所进行的讨论。

变体 模式变体或特例的简短描述。

已知使用 从已存在的系统中给出模式使用的例子。

效果 模式提供的优点和模式存在的潜在不足。

参见 参考那些解决相似问题的模式，并且参考另一些模式，它们有助于我们细化正在描述的模式。

1.6 模式和软件体系结构

判断模式取得成功的一个重要准则是它们在多大程度上达到了软件工程的目标。模式必须支持复杂的、大规模系统的开发、维护以及演化。它们也必须支持有效的产业化的软件生产，否则它们就只是停留在有趣的智能概念上，而对于构造软件没有什么用途。

1.6.1 模式作为智力构造块

我们已经知道，在开发一个软件系统时，模式是处理受限的特定设计方面的有用智力构造块。 21

因此，模式针对软件体系结构的一个重要目标——用已定义属性进行特定的软件体系结构的构造。再次考虑模型-视图-控制器模式。它提供了一个结构，用于交互应用程序的用户界面的裁剪。

软件体系结构的一般技术，比如使用面向对象特征（如继承和多态性）的指南，并没有针对特定问题的解决方案。绝大多数现有的分析和设计方法在这一层次也是失败的。它们仅仅提供建造软件的一般技术，例如“从实现中分离策略”[RBPEL91]。特定体系结构的创建仍然基于直觉和经验。

模式使用特定的面向问题的技术来有效补充这些通用的与问题无关的体系结构技术。注意，模式不会舍弃软件体系结构的现有解决方案——相反，它们填补了一个没有被现有技术覆盖的缺口。

1.6.2 构造异构体系结构

单个模式不能完成一个完整的软件体系结构的详细构造——它仅仅帮助你设计应用程序的某一方面。然而，即使你正确设计了这个方面，整个体系结构仍然可能达不到期望的所有属性。为“整体上”达到软件体系结构的需求，我们需要一套丰富的涵盖许多不同设计问题的模式。可获得的模式越多，能够被适当解决的设计问题也会越多，并且我们可以更有力地支持构造带有已定义属性的软件体系结构。

另一方面，可获得的模式越多，就越难获得对它们的概述。如我们已经指出的那样，模式之间有许多关系。当应用一个模式时，你想知道哪个其他模式有助于细化这个模式所引入的结构。你也想知道你可以与哪个其他模式结合使用。

为了有效使用模式，我们需要将它们组织成*模式系统*（*pattern system*）。模式系统统一描述模式，对它们分类，更重要的是，说明它们之间如何交织。模式系统也有助于你找到正确 22

的模式来解决一个问题或确认一个可选解决方案。这和模式目录（*pattern catalog*）相反，在模式目录中每个模式描述的多少与别的模式无关。模式系统有助于我们使用模式全体所提供的效能。

1.6.3 模式和方法

好的模式描述也包含它的实现指南，你可将其看成是一种微方法（*micro-method*），用来创建解决一个特定问题的方案。通过提供方法的步骤来解决软件开发中的具体再现问题，这些微方法补充了通用的但与问题无关的分析和设计方法，比如Booch[Boo94]和“对象建模技术”[RBPEL91]。5.4节“作为实现指南的模式系统”详细讨论了这个问题。

1.6.4 实现模式

从模式与软件体系结构的集成中产生的另一个方面是用来实现这些模式的一个范例。目前的许多软件模式具有独特的面向对象风格。以下是一个吸引人的结论：我们能够有效实现模式的惟一方式是使用面向对象编程语言。但是，我们相信这个结论是错误的。

一方面，许多模式，包括本书中的那些模式，确实使用了诸如多态性和继承性等面向对象技术。策略（Strategy）模式[GHJV95]和代理（Proxy）模式是这种模式的例子。

另一方面，面向对象特征对实现这些模式并不是最重要的。例如，代理模式通过放弃继承性而失去了一小部分简洁性。在C中实现策略模式可以通过采用函数指针来代替多态性和继承性。

在设计层次，大多数模式只需要适当的编程语言的抽象机制，如模块或数据抽象。因此，你可以用几乎所有的编程范例并在几乎所有的编程语言中来实现模式。另外，每种编程语言都有它自己特定的模式，即语言的惯用法。这些惯用法捕获了现有的有关该语言的编程经验并为它定义了一个编程风格。

23

总之，我们可以说：没有单个的范例或语言可以用来实现模式。模式可以与构造软件体系结构用到的每一个范例进行集成。

1.7 总结

模式为开发具有已定义属性的软件提供了一种有前途的方法。它们把现有的设计知识文档化，有助于你为设计问题找到适当的解决方案。模式存在于各种不同的规模和抽象之中，并覆盖软件开发的许多不同的和重要的领域。模式彼此交织——你可以用它们来细化其他较大规模的模式，并且你可以将它们组合以解决更复杂的问题。它们关注软件体系结构的重要方面并对现有技术和方法进行补充。你可以用任何编程范例来集成它们并且可以用几乎任何一种编程语言来实现它们。总之，模式全体提供了一个“智力工具箱”（*Mental todbox*）来帮助你构造满足应用程序的功能需求和非功能需求的软件。

模式已经被成功地运用。我们在商业领域[EKM+94]、自动化领域[BM95]和电信领域

[Sch95]的应用程序中都可以找到它们的应用。它们在应用程序框架（如ET++[WGM88]或InterViews[LCITV92]），以及在诸如C++元信息协议[BKSP92]的运行时环境中，均发挥了重要的作用。

但是，为了发挥模式的全部效能，我们需要提供可以跨越单个模式范围的技术上和方法上的支持。我们将在第5章“模式系统”中考察这些方面。

24

第2章

体系结构模式

层式蛋糕

2cL可可香草甜酒

2cL杏黄色白兰地酒

2cL双层奶油

把可可香草甜酒倒入一个咖啡杯。加杏黄色白兰地酒时应仔细地让它流过贴于玻璃杯内侧的汤匙背面。以同样的方式加入奶油。独立层不能被混合。当阅读层模式时就可以喝了。

体系结构模式描述了软件系统基本的结构化组织方案。它们提供了一套预先定义好的子系统来制定它们的职责，包括用于组织它们之间的规则和指南。

本章给出以下8种体系结构模式：层（Layers）、管道和过滤器(Pipes and Filters)、黑板(Blackboard)、代理者（Broker）、模型-视图-控制器（Model-View-Controller）、表示-抽象-控制（Presentation-Abstraction-Control）、微核（Microkernel）、映像（Reflection）。

2.1 引言

体系结构模式代表了模式系统中的最高等级模式，它有助于明确一个应用的基本结构。后面的每个开发活动（例如，子系统的详细设计，系统不同部分之间的通信和协作，以及它后期的扩展）都遵循这种结构。

每个体系结构模式都有助于获得一个特定的全局系统属性，如用户接口的适应性。可以将有助于支持相似属性的模式进行分类。本章把模式划分为以下四类：

- *从混沌到结构*。此类模式有助于避免一个组件或对象的“海洋”。特别地，它们支持把整个系统任务以受控方式分解成可协作的子任务。这一类包括层模式、管道和过滤器模式以及黑板模式。
- *分布式系统*。该类包含一种模式，即代理者模式，并涉及到其他种类中的两个模式，即微核模式、管道和过滤器模式。代理者模式为分布式应用提供了一个完整的基础结构。它的基础体系结构很快就被对象管理组（OMG）所标准化［OMG92］。微核模式及管道和过滤器模式仅认为分布是一个次要的考虑因素，所以这两种模式只被列在相应的主分类下面。不过关于这两个模式分布情况的详细描述仍将在2.3节“分布式系统”中讨论。
- *交互式系统*。该类由两种模式构成，即因Smalltalk而闻名的模型-视图-控制器模式和表示-抽象-控制模式。这两个模式都支持具有人机交互特征的软件系统的构建。
- *适应性系统*。映像模式和微核模式强有力地支持应用的扩展以及它们对进化技术和变更功

能需求的适应性。 26

注意：这种分类并不彻底。它只针对目前我们所描述的体系结构模式，而在需要添加更多体系结构模式时可能有必要定义新的种类——第5章“模式系统”对此进行了更深入的讨论。

体系结构模式的选择应该由当前应用的一般属性来确定。例如,自问一下，你提议的系统是不是一个交互式系统，或是存在许多细微差别变化的系统。模式的选择应该更多地受到应用中非功能需求（例如可更改性或可靠性）的影响。

在决定一个特定体系结构模式前多考察几种选择也是有用的。例如，表示-抽象-控制模式（PAC）和模型-视图-控制器模式（MVC）都可用于交互系统。类似地，映像模式和微核模式都支持软件系统进化需求的适应性。

不同的体系结构模式导致不同的结果，即使针对同一个问题或相似的问题也是如此。例如，MVC体系结构往往比PAC体系结构更有效率，另一方面，PAC在支持多任务和特定任务用户接口方面做得比MVC更好。

但是，绝大多数软件体系结构不能仅依据一个体系结构模式来构建。它们必须支持几种用不同体系结构模式来说明的系统需求。例如，你需要进行一项设计，它既有在异构计算机网络中组件分布的灵活性，又要求用户接口的适应性。你必须结合几个模式来构建这样的系统——在这种情况下，合适的模式是代理者模式和MVC模式。代理者模式提供了组件分布的基础结构，而MVC模式的模型在代理者基础结构中担当服务器的角色。类似地，控制器担任客户机的角色，视图将客户机和服务器的角色结合，它作为模型的客户机，并作为控制器的服务器。

但是，一个特定的体系结构模式，或几个模式的组合，并不是一个完整的软件体系结构。 27
它保持了软件系统的一个结构化框架，但需要进一步说明和定义，其中包括把应用功能集成到
框架的任务，以及组件及其相互关系的细化，这些工作可能要在设计模式和惯用法的协助下完
成。体系结构模式的选择，或几个模式的结合，仅仅是设计一个软件系统的体系结构的第一步。 28

2.2 从混沌到结构

在开始设计一个新系统之前，我们从客户那里收集需求并转换成规格说明。这些活动要比想像的复杂得多。最近一本由Michael Jackson写的书［Jac95］描述了该主题。

我们乐观地假定新系统的需求是良好定义的而且是稳定的。接下来主要的技术任务是定义系统的体系结构。这个阶段意味着寻找一种高层划分法来把系统分成多个组成部分。我们往往关注各个方面的整体转换，并将工作重点集中于将混乱组织成可工作结构的问题。Ralpha Johnson把这种情形称为“混沌球”［Joh96］。这常常是我们开始所遇到的情况，而且我们必须把这种状况转换成一个有组织的结构。

由于某些原因，沿应用领域中能看到的线切这个球无济于事。一方面，得到的软件会包含许多与该领域没有直接关系的组件。经理和助理的功能性就是一个很典型的例子。另一方面，我们需要的远不只是一个可工作的系统——它应该具备一些属性，如可移植性、可维护性、可理解性、稳定性等，而这些属性与应用的功能性不直接相关。

我们描述如下三种体系结构模式：层、管道和过滤器以及黑板，它们提供不同类型的高层系统划分。

- 层模式有助于构建这样的应用：它能被分解成子任务组，其中每个子任务组处于一个特定的抽象层次上。
- 管道和过滤器模式为处理数据流的系统提供了一种结构。每个处理步骤封装在一个过滤器组件中。数据通过相邻过滤器之间的管道传输。重组过滤器可以建立相关系统族。
- 黑板模式对于无确定性求解策略的问题比较有用。在黑板模式中有几个专用子系统收集其知识以建立一个可能的部分解或近似解。

29

层模式描述了体系结构划分中传播最广的原则。我们在系统体系结构文档中看到的许多模块图都隐含着分层体系结构。但是，实际的体系结构或者是不同范例（它自身不能被细分）的混合，或者是彼此间没有明确体系结构界限的协作组件的隐含集合。在这种情形下，我们试图更加严格地描述并列出真正层次系统的特征。

相反，管道和过滤器模式用得比较少，但它对于需要渐增式地处理数据流的领域很有吸引力。意外的是，以这种方式建模的一些系统族并不是采用这种范例的好的选择，我们忽略了这个模式可以更好使用的一些领域。在模式描述中，我们对这个主题进行更深入的扩展。

黑板模式来自人工智能领域。我们把该范例作为模式描述是因为它背后的思想值得在更宽广的语境中考察。在结构很差的（或者简单地说是新的和不成熟的）领域中，我们只有关于如何处理特殊问题的零碎知识。黑板模式给出了这样一种方法，即综合这样的零碎知识来获得解决问题的方法，哪怕这些零碎知识是局部最优的或没有保证的。当应用领域随时间推移而成熟之后，设计者经常放弃黑板体系结构而开发出支持更接近解决方法的体系结构，其中处理步骤是由应用结构预先定义好的。

30

2.2.1 层

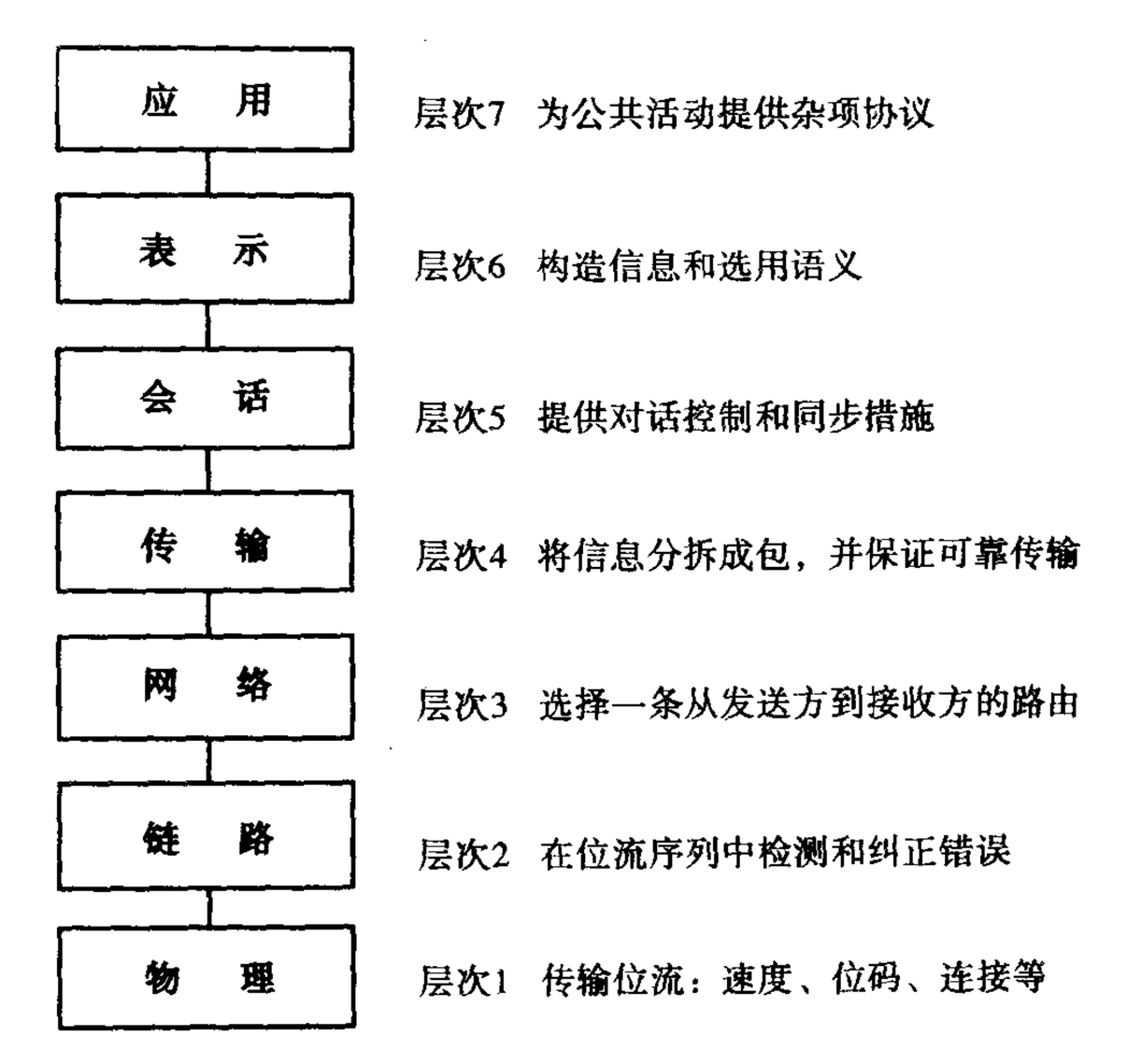
层（*layer*）体系结构模式有助于构建这样的应用：它能被分解成子任务组，其中每个子任务组处于一个特定的抽象层次上。

1. 例子

网络协议可能是层体系结构的最好例子。这样的协议由一套描述计算机程序之间如何跨越机器边界进行通信的规则和约定来组成。定义所有消息的格式、内容和意义。所有场景都被详细地描述，描述方法常常是顺序图表。协议规定了各个抽象层次上的协定，范围从位传输细节到高层应用逻辑。因此，设计者使用几个子协议并将它们安排在各层当中。每层处理通信的一个特定方面，并利用相邻较低层的服务。国际标准化组织（ISO）定义了下面的体系结构模型，即ISO 7层模型[Tan92]。

31

与采用单块来实现协议相比，分层方法被视为一种更好的实践，因为分别实现概念上不相同的问题可获得一些好处。例如，帮助团队开发以及支持增量编码和测试。使用半独立部件也能使后期的单个部件的交换变得更容易。更好的实现技术（如新的语言或算法）可以通过简单地重写代码的划界部分加入进来。



ISO是一个重要的参照模型，而被认为是“Internet协议组”的TCP/IP协议是普遍运用的网络协议。我们用TCP/IP来描述分层的另一个重要理由是：不同语境中独立层的重用。例如，TCP可以由不同的分布式应用（如`telnet`或`ftp`）来使用。

2. 语境

一个需要分解的大系统。

3. 问题

假设你正在设计一个系统，它的显著特征是混合了低层与高层问题，这里的高层操作依赖于低层操作。系统的一部分处理低层问题，如硬件陷阱，传感器输入，从文件中读入（二进制）位或线路上读入电子信号。在谱系的另一端有用户可见的功能，如多用户“地牢”游戏的界面或高层策略（如电话付账价目）。一个通信流的典型模式由两部分组成，一是高层到低层移动的请求，二是对请求的应答，有关事件的输入数据或通知沿反方向传输。

这样的系统往往需要一些与其垂直子划分正交的水平构建。即几个操作处在同一个抽象层但彼此之间很大程度上是独立的。可以在描述ISO 7层模型的图中看“和/并”出现的地方。

提供的系统规格说明描述了在某种程度上的高层任务，并规定了目标平台，并希望能移植到其他平台上。系统的外部边界要指定一个先验条件，如系统必须附带的功能接口等。高层任务到平台的映射不是直接的，主要由于它们过于复杂而不适合直接使用平台提供的服务来实现。 32

在这样的情形下需要平衡下列强制条件：

- 后期源代码的改动应该不影响到整个系统。它们应被限制在一个组件内且不影响其他组件。
- 接口应该是稳定的，甚至可以用标准件来限定。
- 系统的各个部分应该可以替换。组件可以被别的实现方法来替代且不影响系统的其他部分。可以给出一个低层平台但可以承受未来的改动。这样的基本变动往往需要改动代码并重新

编译，系统的重新配置也可以在运行期间使用管理界面来完成。调整高速缓存或缓冲区大小就是这种改变的一个例子。一个可变动性的典型形式是一客户组件动态地转变成一个服务的不同实现，而这一点在开始未必能行。考虑到一般变更的设计是优秀系统演化的一个主要助推器。

- 以后可能有必要建立其他系统，这些系统具有和当前设计的系统同样的低层问题。
- 相似职责应分组以提高可理解性和可维护性。每个组件应该是内聚的——如果一个组件实现分散的问题，它的整体性就会丧失。分组和内聚同时是互相制约的。
- 没有“标准的”组件粒度。
- 复杂组件需要进一步分解。
- 跨组件边界可能影响性能，例如当有大量的数据必须通过几个边界传输时，或在需要跨越多个边界的地方。
- 系统可以由一个程序员组来创建，工作界限必须划分清楚——这是在体系结构设计阶段经常要监督的需求。

33

4. 解决方案

从高层看解决方案是非常简单的。将系统分成适当层次，按适当次序放置。从最低抽象层（称为第1层）开始，这是系统的基础，以梯状把抽象层J放在J-1层的顶部，直到顶层功能——称它为第N层。

注意，这里没有描述实际设计的层的顺序，仅给出概念化观点。它也没有说明某层J是不是一个需要进一步分解的复杂子系统，或者是否仅需把J+1层的需求转换为J-1层而自身不需要做什么：但很重要的是，在某一层中所有用到的组件工作在同一个抽象层。

第J层提供的绝大多数服务由第J-1层提供的服务组成。也就是说，实现每一层的服务的一种策略是有意义地组合低层的服务。第J层的服务可以依赖第J层的其他服务。

5. 结构

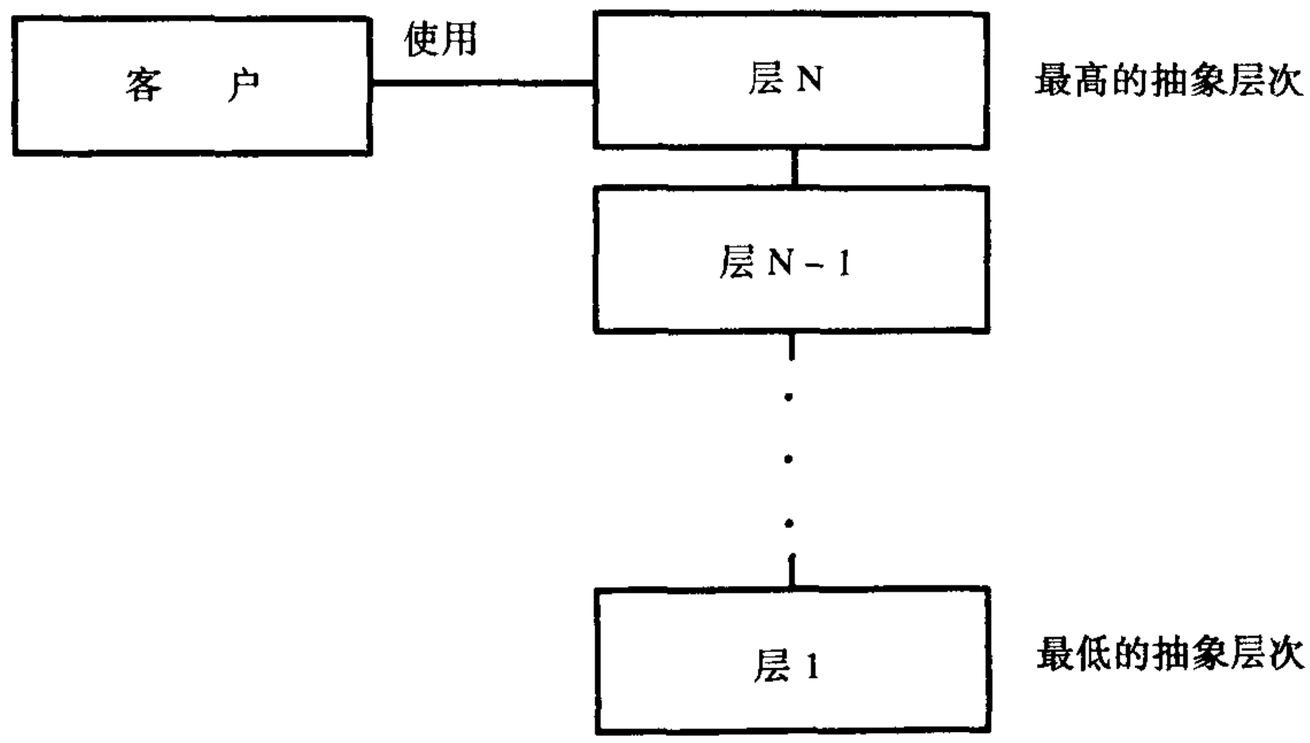
一个独立层可以由以下CRC卡片来描述。

类 层J	协作者 • 层J－1
责任 • 提供了由层J+1使用的服务 • 委派了任务给层J－1	

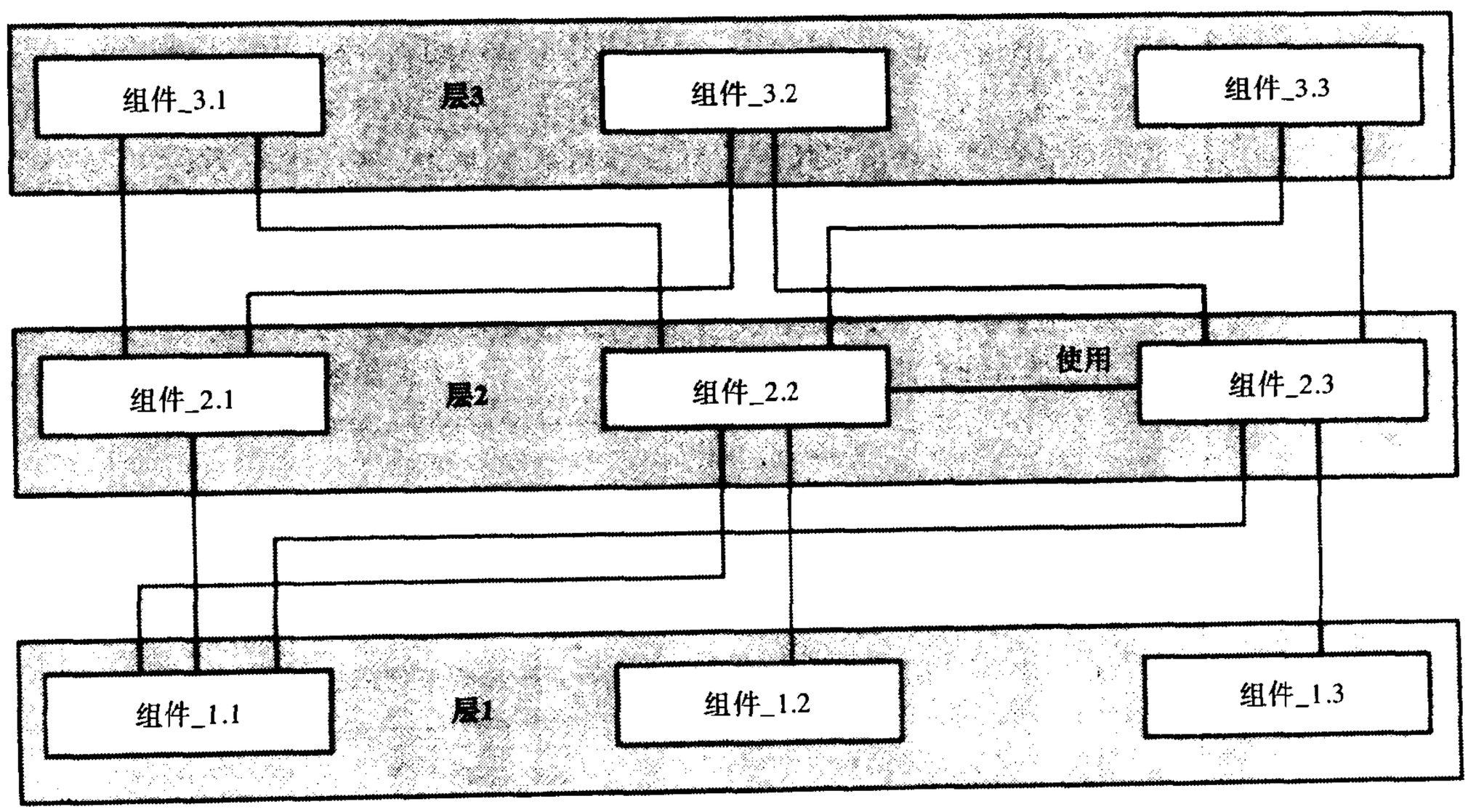
层模式的主要结构特征是第J层的服务只被第J+1层使用——层之间没有更进一步的直接依赖关系。可以将这种结构与一个堆栈或甚至一个洋葱相比较。每个独立层都要防止较高层直接访

34

问较低层。



更细致地检查一个独立层可以看出它们是由不同组件构成的复杂实体。在下图中，每层由三个组件构成。中间层有两个组件交互。不同层中的组件彼此直接调用——其他设计通过加入统一接口保护每一层。在这样的设计中，组件_2.1不再直接调用组件_1.1，而改为调用第1层中面向需求的接口对象。在实现部分，我们讨论了直接访问的利弊。 35



6. 动态特性

下面的场景是分层应用动态行为的原型。这并不意味着在每一个体系结构中会涉及每种场景。在简单的分层体系结构中，可能只看到第一个场景，但绝大多数分层应用包括场景I和场景II。由于篇幅有限，我们没有给出这个模式的目标消息顺序图表。

场景I 可能是最知名的。一个客户端向层N发出一个请求。由于层N自身不能完成这个请

求，它就调用层N-1相应的子任务。向层N-2、层N-3、…直到层1发送进一步请求，层N-1完成相关任务。这里，最底层服务是最后完成的。必要的话，通过从层~层2，从层2~层3等等直到到达层N，这样把不同的需求反馈回去。实现部分的例子代码描述了这些思想。

这样自顶向下通信的一个特点是层J往往会把层J+1的一个请求转换成几个请求发给层J-1。正是由于这样的情形，即层J处于一个比层J-1更高的抽象层次，因此必须把高层服务对应到更加原始的服务。

场景II　描述了自底向上通信——一个从层1开始的动作链，例如在一个设备驱动程序探测输入的时候，驱动程序把输入转换成内部格式并报告给层2，层2开始解释，等等。在这种方式下数据向上移动直到到达最高层。自顶向下消息和控制流常常描述为“请求”，而自底向上调用可以称为“通知”。

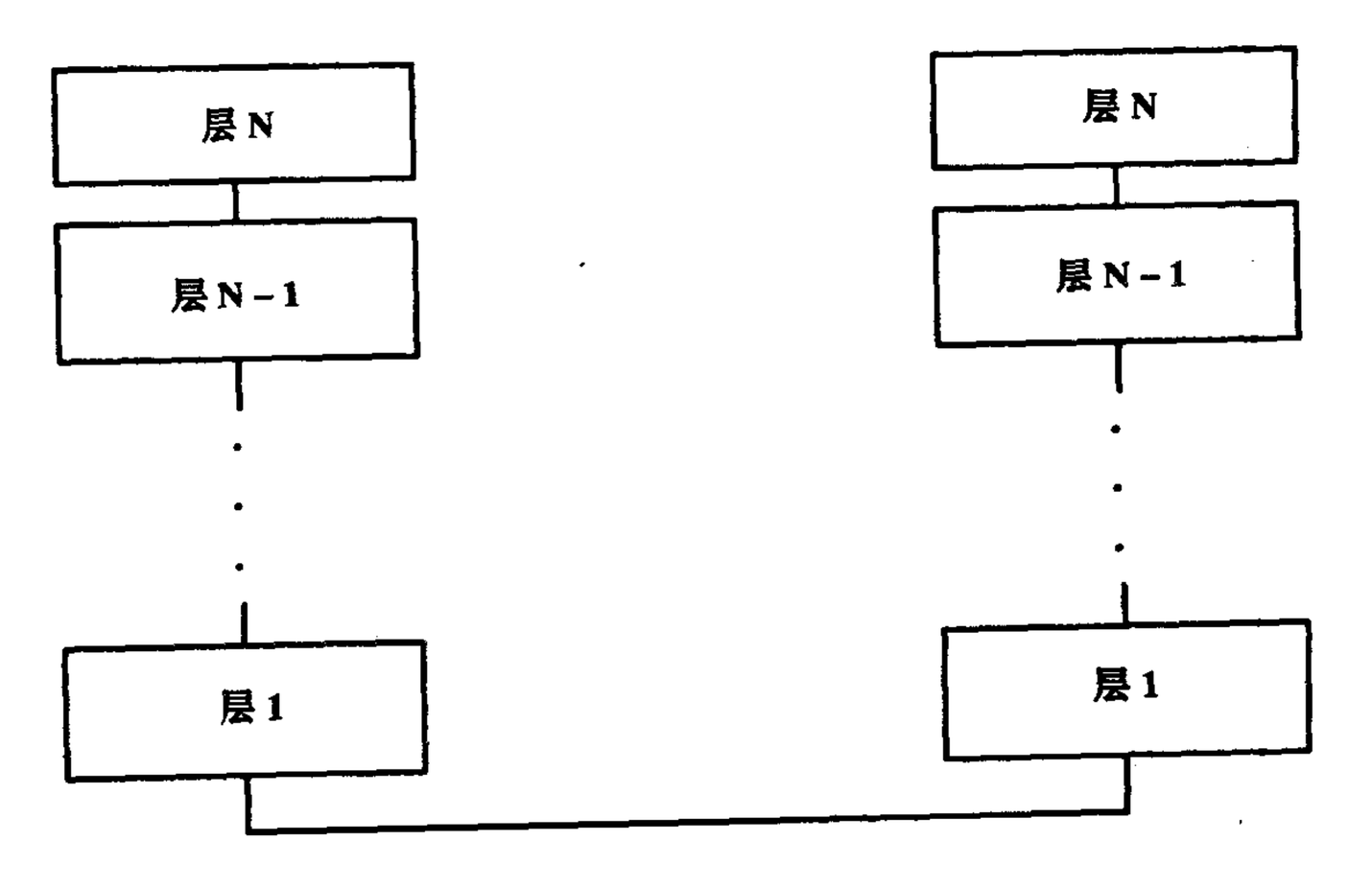
如场景I中提到的一个自顶向下请求往往在低层变出几个请求。相反，几个自底向上的通知也可以压缩成一个结构上更高的通知，或保持1:1的关系。

场景III　描述了请求只需经历层的一个子集的情形。如果层N-1可以满足请求的话，一个顶层（层N）请求可能只需到达（相邻层）层N-1就可以了。例如，层N-1作为高速缓存，一个来自层N 的请求可以被满足，而毋须发送请求到层1，也毋须从本地发送请求至远程服务器。注意，这样的缓存层保留状态信息，而只是向前的请求往往是无状态的。无状态层往往有简化程序的优点，尤其是考虑重新进入时。

36

场景IV　描述了和场景III类似的情形。一个事件在层1被探测到，但到层3停住不再向层N传送。例如，在一个通信协议中，一个重发的请求可能来自一个不够耐心的客户，他在这之前已请求过数据。在此期间服务器已发出了应答，且应答和重发请求交叉。这种情况下，服务器端的层3应该注意到这一点并截取重发请求而不采取进一步措施。

场景V　包括彼此能互相通信的两个层N的堆栈。这个场景在通信协议中很知名，其堆栈称为“协议堆栈”。在下图中，左边堆栈的层N发出一个请求。请求通过层向下移直到到达第1层，再发送给右边堆栈的层1，然后在右边堆栈中通过层向上移动。对请求的响应沿反向路径直到它到达左边堆栈的层N。



协议堆栈的更详细内容见“已解决的例子”小节，在那里我们会用TCP/IP作为例子讨论几个通信协议问题。

37

7. 实现

下面的步骤描述了分层体系结构定义的一个逐步细化方法。该方法并非对所有应用都是最好的——有时自底向上或“溜溜球”方法比它更好。参见步骤5中的讨论。

注意，并非下述步骤均是必须的——这依赖于具体应用。例如，几种实现步骤的结果可能很有影响，甚至由以后必须遵守的规格说明所规定。

(1) 为把任务分组成层而定义抽象准则。这个准则往往是来自平台的概念上的间隔。有时你会遇到其他抽象范型，例如特定领域的个性化程度，或概念复杂度。例如，一种棋类游戏应用程序可以由下面这些层组成，自底向上如下所列：

- 游戏的基本单元，如一个象（bishop）。
- 基本移动，如移车（castling）。
- 中盘战术，如西西里（Sicilian）防御。
- 游戏整体策略。

在美式足球中这些层次分别对应后卫球员、快攻、两分钟训练的比赛编排，以及一个最终的完整游戏计划。

在软件开发的真实世界中，我们往往使用抽象准则的混合体。例如，硬件的间隔可以规定低层的形状，而概念复杂性控制高层。使用混合模式分层原理的现成例子如下，按自顶向下顺序排列：

- 用户可见元素
- 特定应用模块
- 公共服务层
- 操作系统接口层
- 操作系统（分层系统本身，或根据微核模式进行结构化）
- 硬件

38

(2) 根据抽象准则定义抽象层数。每个抽象层次对应模式中的一层。有时从抽象层次到层的映射是不明显的。在确定是否把特定方面分成两个层或是将它们合并成一个层时，要权衡利弊。过多的层则增加不必要的开支，而层太少会导致一个很差的结构。

(3) 给每个层命名并指定它们的任务。正如客户所知，最高层的任务是整个系统的任务。所有其他层的任务只作为较高层的助手。如果我们采用自底向上方法，则较低层提供一个基础结构使得能在其上构建较高的层。但是这种方法需要大量的经验，并且在域中预见使得能在可以定义来自较高层特定请求前找到正确的较低层抽象。

(4) 指定服务。最重要的实现原则是层间彼此要严格分离，感觉上就是没有一个组件跨越多于一个层。回过来说，有争议的是，由层J所提供功能的错误类型是编程语言类型固有的，还是层J 定义的类型，或是共享数据定义模块采用的类型。注意，层间共享模块放松了严格分层的原则。

把较多的服务放在高层往往要比低层好，这是由于开发员不必学习大量只有细微差别的原

本书中的代码清单反映了空白字符布局的个人风格：回车符放在开、闭花括号的前后，代码相对花括号缩进。然而，这种布局方式不是C#应用程序所必须采用的。清单2.3显示了另一种代码布局方式，其中使用了不同的空白字符。无所谓采用哪种格式，只需适合于作者就好。

清单 2.3　另一种空白字符布局方式

```
Class
HelloWorld
{
    static void Main()
    { System.Console.WriteLine("Hello World!");}
}
```

如果编译并运行清单 2.3，就会看到结果与清单 2.1 的结果一样：程序输出字符串 Hello World！新的空白字符布局方式对编译器编译代码的能力丝毫不产生影响；也对代码在运行时执行的行为不产生任何影响。

2.4　用 Main()函数开始程序

清单 2.1 所示的应用程序定义了一个包含函数 Main( )的类。Main( )函数是 C# 应用程序的重要部分，因为 Main( )函数是程序开始执行的地方。

编写的每一个C#应用程序都必须有一个含Main( )函数的类。Main( )函数称之为应用程序的入口，C# 应用程序从 Main( )函数的代码开始执行。如果代码包含多个类，那么只有一个类可以包含函数Main( )。如果忘记定义Main( )函数，那么就会收到来自编译器的几个错误信息，如图 2.3 所示。

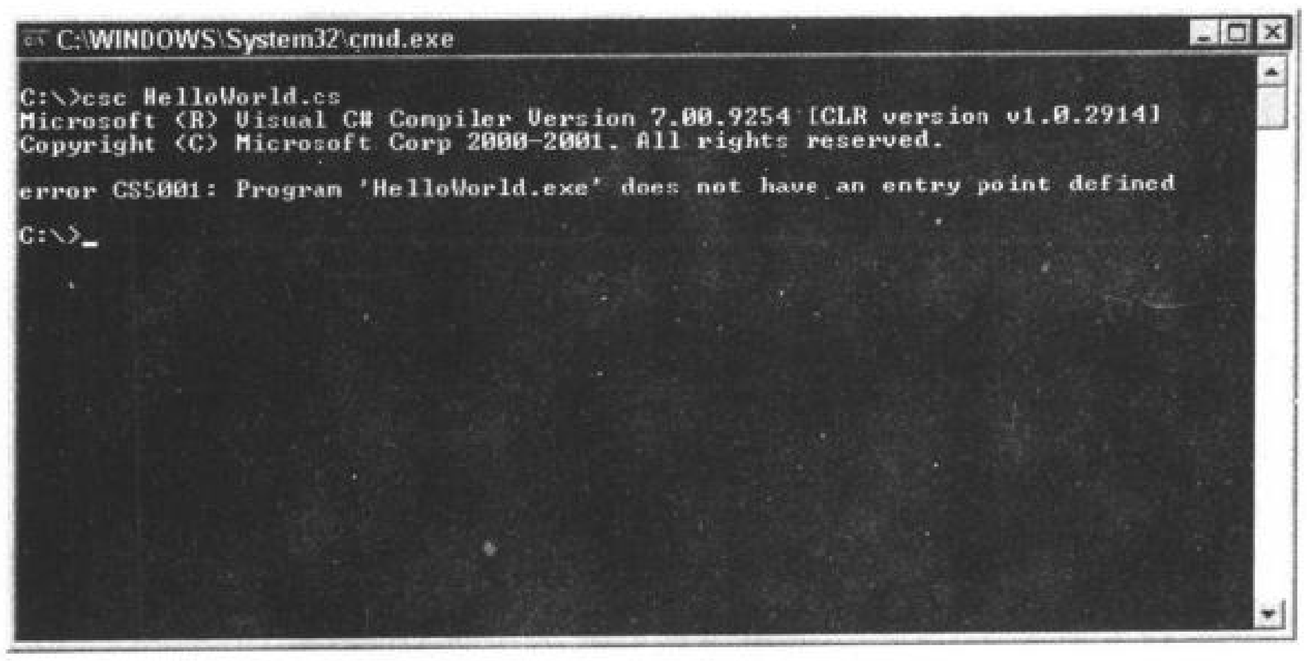


图 2.3　缺少 Main( )函数将导致生成编译器错误

清单 2.1 中定义的 Main( )函数不返回任何内容（因为 void 关键字的缘故），并且不带参数（因为括号内为空的缘故）。实际上，C# 编译器接受 Main( )函数的四种可能的构造：

- public static void Main( )
- public static void Main(string[] Arguments)
- public static int Main( )

式［Sch94］描述了回调使用与事件分路传输相结合的一种面向对象实现方式。命令模式［GHJV95］给出了如何把回调函数封装成第一类对象。

可以将高层与低层分离到一定程度。下面是一个如何用面向对象技术实现的例子。相对接口而言，通过对高层编码，高层可以从低层特殊实现的变体中分离出来。在下面的C++代码中，该接口是一个基类。低层实现更容易替换，甚至在运行时都可以。在例子代码中，一个层2的组件和层1提供者交谈，但不知道和层1的哪个实现交谈。层的“连线”在主程序中完成了，但当通常将公共部分放到连接管理组件中。主程序也通过调用顶层服务来担任客户机的角色。

41

```
#include <iostream.h>

class L1Provider {
public:
    virtual void L1Service() = 0;
};
class L2Provider {
public:
    virtual void L2Service() = 0;
    void setLowerLayer(L1Provider *l1) {level1 = l1;}
protected:
    L1Provider *level1;
};
class L3Provider {
public:
    virtual void L3Service() = 0;
    void setLowerLayer(L2Provider *l2) {level2 = l2;}
protected:
    L2Provider *level2;
};

class DataLink : public L1Provider {
public:
    virtual void L1Service(){
        cout << "L1Service doing its job" << endl;}
};
class Transport : public L2Provider {
public:
    virtual void L2Service() {
        cout << "L2Service starting its job" << endl;
        level1->L1Service();
        cout << "L2Service finishing its job" << endl;}
};
class Session : public L3Provider {
public:
    virtual void L3Service() {
        cout << "L3Service starting its job" << endl;
        level2->L2Service();
        cout << "L3Service finishing its job" << endl;}
};

main() {
    DataLink dataLink;
    Transport transport;
    Session session;
```

```
        transport.setLowerLayer(&dataLink);
        session.setLowerLayer(&transport);

        session.L3Service();
42  }
```

程序输出如下：

```
L3Service starting its job
L2Service starting its job
L1Service doing its job
L2Service finishing its job
L3Service finishing its job
```

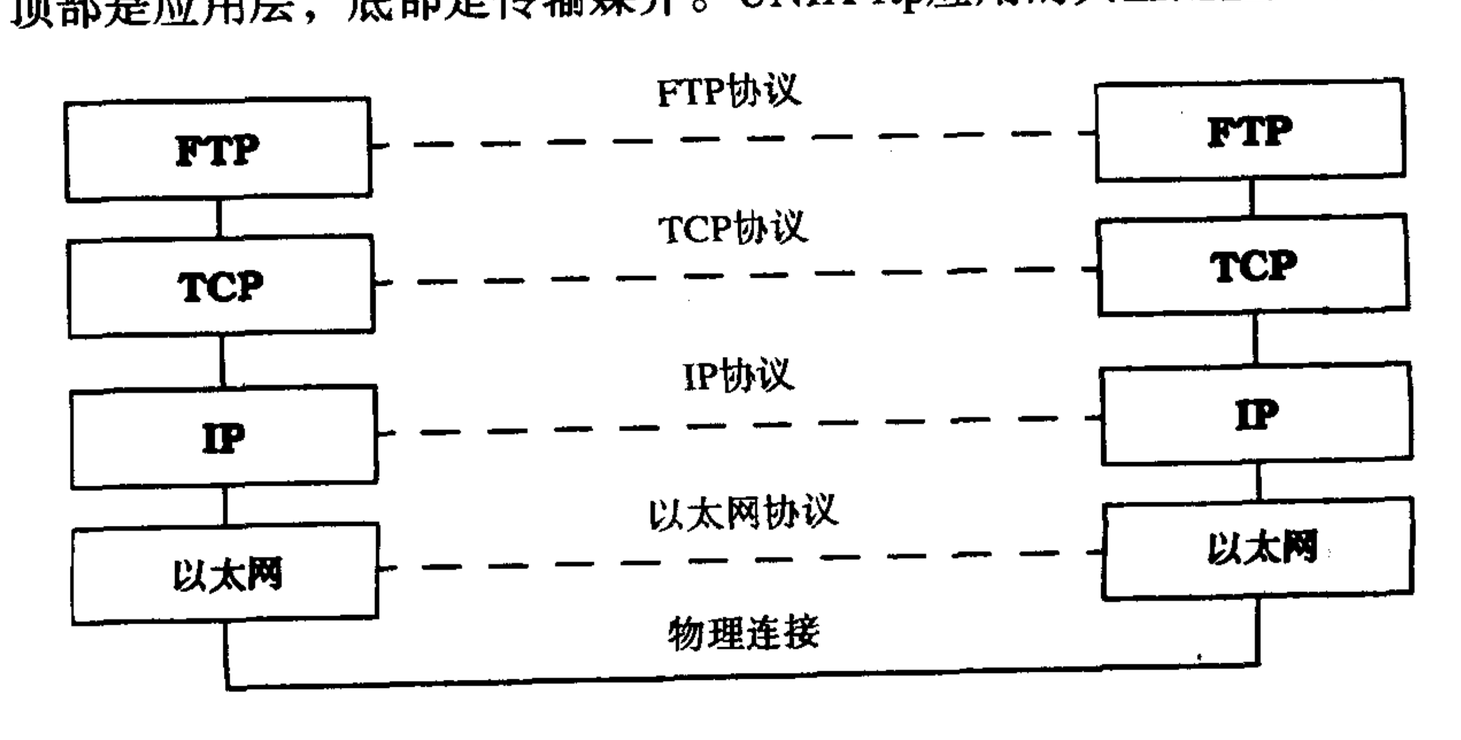
对于把消息既要向上又要向下传送的层的通信堆栈，明确把低层连接到高层通常更好。所以我们再次引入基类，例如：例子代码中的类L1Provider、L2Provider、L3Provider，附加的L1Parent、L2Parent以及L2Peer。类L1Parent提供了层1的类访问下一高层的接口，例如，为了返回结果，发送确认或传输数据流。类似的论点也支持L2Parent。L1Peer提供的接口可用来将消息发送给层1在其他堆栈中的同等模块。所以，层1实现类继承于两个基类：L1Provider和L1Peer。第二层实现类继承于L2Provider和L1Parent，它提供第二层的服务并能作为层1对象的父类。最后，第三层次实现类继承于L3Provider和L2Parent。

如果编程语言在语言级能区分继承和子类型，如Sather［Omo93］和Java［AG96］所示，上面的基类可以通过把数据压入子类并在那儿实现所有方法以将其转换成接口。

(10) 设计一种错误处理策略。对层式体系结构来讲，错误处理在处理时间和编程工作方面代价较大。一个错误既可以在它出现的层内处理也可以输送到下一个更高层。后一种情况，低层必须把错误转换成对更高层有意义的错误描述。根据经验，尽可能在最低层处理错误。这防止了高层被许多不同错误和大量的错误处理代码陷入困境。至少应试着把相似错误类型归并成更一般的错误类型，并仅传播这些更一般的错误。如果不这么做，高层可能面对应用了高层不能理解的低
43 层抽象的错误信息。谁没有看到过所有的隐含错误信息都被上托到最高层——用户层？

8. 已解决的例子

使用最广泛的通信协议——TCP/IP——并不严格按照ISO模型，而只由四层构成：TCP和IP构成中间层，顶部是应用层，底部是传输媒介。UNIX ftp应用的典型配置如下图所示：



TCP/IP有几个与我们的讨论相关的有趣方面。对应的层采用虚拟协议以对等方式通信。意味着这两个TCP实体按一定格式彼此互送消息。从概念观点看，在上图中它们沿标有“TCP协议”的虚线通信。说这个协议是“虚拟的”，因为实际上一个TCP消息在图中从左到右传输最先由左边的IP实体处理。这个IP实体把这个消息看成一个数据包，以头作前缀，将之发送到局部以太网接口。以太网接口随之加上自己的控制信息并通过物理连接发送这些数据。在接收端，局部以太网和IP实体分别去掉以太网和IP相应的头。图中右边的TCP实体从它左边的对等层接收TCP消息，如同虚线发送过来一样。

TCP/IP和其他通信协议的一个显著特征是，标准化功能接口是次要关心问题。部分由这样的下述事实导致：来自不同售主的TCP/IP实现彼此有意不同。售主通常不提供单个层，但提供整套协议的完整实现。其结果是，每个TCP实现生成一固定的核心功能集，但免费提供更多的功能，比如增加灵活性或性能。因为如下两个理由，这种松散性对应用开发人员没有影响。首先，由于严格遵守虚拟协议，不同的堆栈彼此互相理解。其次，应用开发人员使用TCP或UDP顶部的层。这样的上层有一个固定接口。Sockets和TLI是这种固定接口的例子。 44

假设使用一个TCP/IP堆栈顶部的Socket API。Socket API由系统调用`bind()`、`listen()`或`read()`构成。Socket实现概念上处于TCP/UDP的顶部，但也使用低层，如IP和ICMP。这个对严格分层原理的违背对于协调性能是值得的，并且当所有的从sockets到IP的通信层在OS核中建立时还可以进行调整。

独立层的行为和层间流动的数据包的结构在TCP/IP中的定义比在功能接口中严格得多。这是由于不同的TCP/IP堆栈彼此间必须互相理解——它们是日益增长的异构Internet的主力。协议规则准确描述了一个层在特定环境下如何行动。例如在处理一个原先已发过新来的重传消息时，它的行为要准确描述。数据包规格说明主要关心添加给信息的头和尾。规定头和尾的大小，以及其子域的意义。例如，在头中，协议堆栈编入诸如发送者，目的，使用协议，时限信息，顺序号和校验码等信息。关于TCP/IP的更多信息，见例子［Ste90］。要知道更多的详细资料，可查阅始于［Ste94］的系列丛书。

9. 变体

松散分层系统（*Relaxed Layered System*）。这是层模式的一种变体，这种模式的层间关系约束较少。在一个松散分层系统中每个层可以使用比它低的所有层的服务，而不仅仅是相邻低层。一个层也可以不分不透明——这意味着它的某些服务仅对相邻高层可见，而其他服务可被所有更高层可见。在松散分层系统中，灵活性和性能的增加是对可维护性丧失的回报。这个代价往往比较昂贵，在放弃开发人员要求的捷径之前必须仔细考虑。这些捷径常常更多地在基础结构系统中看到（相比应用软件中而言），如Unix操作系统或X Window系统。主要原因是基础结构系统相比应用系统而言修改较少，而且它们的性能往往比可维护性更重要。 45

通过继承分层（*Layering Through Inheritance*）。这种变体可在一些面向对象系统中找到，并且在［BuCa96］有描述。该变体中，低层作为基类实现。一个向低层请求服务的高层从低层实现中继承，所以可以对基层服务提出请求。这种方式的一个好处是高层可以根据需要修改低层服务。缺点是继承关系把高层与低层紧紧捆绑在一起。例如，如果C++基类的一个数据布局改变了，所有的子类必须重新编译。这种由继承引入的偶然依赖关系也称*脆弱基类问题*(*fragile*

base class problem)。

10. 已知使用

虚拟机。我们可以把较低层次说成是一个虚拟机，它用来将低层次细节或各类硬件与较高层次隔离开来。例如，Java虚拟机（JVM）定义了一种二进制代码格式。用Java编程语言写的代码转换成平台中立的二进制代码（也称为字节代码），交付给JVM来解释。JVM本身是一个特殊平台——对不同操作系统和处理器有不同的JVM的实现。在保持平台独立性的同时，这种两步转换过程允许平台中立的源代码和人不可读的二进制代码的交付[⊖]。

API。一个应用程序编程接口（API）是一个封装了低层常用功能的层。一个API通常是一个功能规格说明的平坦集合，如Unix系统调用。“平坦”在这里的意思，举例说，是用于访问UNIX文件系统的系统调用不会同用于存储分配的系统调用分离开来——只能从文档那里知道`open()`或`sbrk()`属于哪个组。我们在其他层中找到上面系统调用，如带有诸如`printf()`或`fopen()`的C标准库［KR88］。这些库提供了不同操作系统之间可移植性的好处，提供了诸如输出缓冲或格式化输出等附加的高层服务。它们完成这些职责往往很低效[⊖]，且可能有更多的硬规定行为，相反传统的系统调用会给出更大的灵活性——而且有更多的出错和概念不匹配，这主要是由于高层应用抽象和低层系统调用之间的差距导致的。

46

信息系统（IS）。来自商务软件领域，它往往使用两层体系结构。底层是一个数据库，拥有公司特殊数据。通过在该数据库顶部并行地开展应用工作来完成不同任务。主机交互系统和颇受赞誉的客户机-服务器系统常常使用这种体系结构。由于用户接口和数据表示的紧密耦合，导致其共享问题，在它们之间引入第三层——领域层——这一层模拟了问题域的概念结构。由于顶层仍然混合了用户接口和应用，这一层也可分开，导致一个四层体系结构。从高到低分别是：

- 表示
- 应用逻辑
- 领域层
- 数据库

有关商务建模的更多信息参见［Fow96］。

Windows NT［Cus93］。该操作系统是根据微核模式来构建的。NT执行程序组件对应微核模式的微核组件。NT执行程序（Executive）是一个松散分层系统，正如在“变体”小节所述。它有以下几个层：

47

- 系统服务：子系统和NT执行程序之间的接口层。
- 资源管理器层：它包含了对象管理器、安全引用监视器、过程管理器、I/O管理器、虚拟存储管理器和局部过程调用等模块。
- 内核：它关心一些基本功能，如中断和意外处理、多处理器同步、线程调度和线程分配。
- HAL（硬件抽象层）：该层隐藏了不同处理器系列机器之间的硬件差异。
- 硬件。

⊖ Java字节码可转换成ASCII表示，而ASCII表示是一种面向对象的汇编程序代码。这种代码可读，但比较难读！

⊖ 高层中的输入/输出缓冲常常会有相反的效果——性能要好于随便直接使用底层系统调用。

Windows NT放松了层模式的原则，因为效率方面的考虑，内核和I/O管理器是直接访问基础的硬件。

11. 效果

层模式有几个**优点**：

层的重用。如果一个独立层体现了一个良好定义的抽象且有良好定义和文档化的接口，该层就可在多个语境中被重用。但是，除了不重用已存在层的代价较高，开发人员往往宁愿重写这个功能。他们认为现有层不能准确符合他们的目的，分层会导致高性能处罚——况且他们会做一项更好的工作。一项经验研究指出，已存在层的黑盒重用会显著地减少开发工作量且减少缺陷数［ZEWH95］。

标准化支持。清晰定义和共同接受抽象层能促进标准化任务和接口的开发。同一接口的不同实现可以替换使用。这样可以让你使用不同层的不同售主的产品。一个众所周知的标准化接口的例子是POSIX编程接口［IEEE88］。

局部依赖性。层之间的标准化接口往往限制被改动层的改动代码的影响。硬件、操作系统、窗口系统、特殊数据格式等等，它们的变动往往只影响一层，不用改变其他层就可以适应被改变层。这支持了系统的可移植性。可测试性也支持得很好，因为你可以测试系统中独立于其他组件的特殊层。

48

可替换性。独立层实现不需要太费劲就可以被语义上等价的实现所替换。如果层之间连接在代码中是硬连线的（hard-wired），这可以用新层的实现的名称来更新。甚至可以采用用于接口适应的适配器模式［GHJV95］来以一个不同接口的实现替换一个旧的实现。另一种情况是动态替换，例如，可以用桥接模式［GHJV95］来实现，并修改指针适应时的实现。

硬件替换或添加是描述可替换性的主要例子。例如，一个新的硬件I/O设备通过安装正确的驱动程序就可投入使用，驱动程序可以插入或替换旧的。高层不受替换的影响。诸如以太网的一种传输媒介能用令牌环替换，在这种情况中，高层不需要改动它们的接口，并可以和以前一样继续向低层请求服务。但是如果想对两个在接口和服务不完全匹配的层之间切换，则必须在这两层之上建立一个隔离层。可替换性的好处在于增长的编程工作的代价，但可能降低运行性能。

层模式也有如下一些**不足**：

更改行为的重叠。层的行为改变时会出现一个严重问题。例如，假设在网络化应用底部替换一个10M/s（Megabit/Sec）的以太网层并改为把IP放在155M/s ATM的顶部[⊖]。由于I/O存储性能的极限，局部终端系统不能足够快地处理进来的数据包以跟上ATM的高数据速率。但是带宽密集型应用（如医疗图像或视频会议）能从全速ATM中获益。并行发送多个数据流是避免上述低层极限的高层解决方案。类似地，IP路由器，它在Internet中发送包，它能通过多CPU系统分层运行于高速ATM网络的顶部以实现IP包的并行处理。

49

总之，高层往往不受低层改动的影响。这就允许系统通过去掉低层和/或代之以更快的解决

⊖ ATM（异步传输模式）与普通的低速网络（如以太网和令牌环）相比，提供了很高的数据速率（从155 Mbps~2.4 Gbps）和功能（如服务质量保证）。此外，ATM能够模拟LAN中的以太网的行为，后者允许它以无缝方式集成到现成的网络之中。更多的有关ATM的信息参见[HHS94]。

方案（如硬件）等透明性调整。如果不得不在许多层上做相当数量的重复工作以合并外观上的局部变动，那么分层便成为一个缺点。

降低效率。说起来一个分层体系结构的效率往往要低于整体结构或一个“对象的海洋”。如果在上层中的高层服务很大程度上依赖于最低层，则所有的相关数据必须通过一些中间层转换，且可以转换若干次。同样，由低层产生的所有结果或错误信息也传送到最高层。例如，通信协议通过添加消息头和尾从高层传输消息。

不必要的工作。如果低层执行的某些服务执行了多余或重复工作，而这些工作并非高层真正需要，这对性能的影响是负面的。通信协议堆栈中的多路分解是这种现象的一个例子。几个高层需求导致相同输入位序列被读多次，因为每一个高层需求对位的不同子集感兴趣。另一个例子是文件传输中的错误纠正。通用目的的低层传输系统最先写，并提供很高的可靠度，但它可能更经济甚至是强制性地把可靠性建立在较高层中，例如通过使用校验位。有关这些权衡的细节和分层系统中功能模块放在哪儿的更进一步的讨论可参见[SRC84]。

难以认可层的正确粒度。层数太少的分层体系结构不能完全发挥这种模式在可重用性、可更改性和可移植性上的潜力。相反，层过多会引入不必要的复杂性和层间分离的冗余以及变元和返回值传输的开销。层粒度的确定和层任务的分配是困难的，但对体系结构的质量是很关键的。如果潜在客户范围内的应用适应所定义的层，则可以仅使用一个标准体系结构。

50

参见

组合消息（*composite message*）。Aamod Sane 和Roy Campbell[SC95b]描述了通过层传输的消息的面向对象封装。一个组合消息是一个由头、有效负载和嵌入包构成的包。所以组合消息模式是组合模式［GHJV95］的一种变体。

*微核*体系结构可以看作是特殊的分层体系结构。见“已知使用”小节中有关Windows NT的讨论。

*PAC*体系结构模式也强调提高抽象层次。但是，整个PAC是PAC节点的一棵树，而不是一层接一层的竖型结构。PAC体系结构模式强调每个节点由三个组件（即*表示*、*抽象*和*控制*）组成，而层模式没有对一个独立层的任何子划分进行规定。

致谢

这个模式由Paulo Villela进行了仔细评审，他在早期草案中指明了许多方向。Douglas Schmidt在ATM讨论中提供了有价值的支持。

51

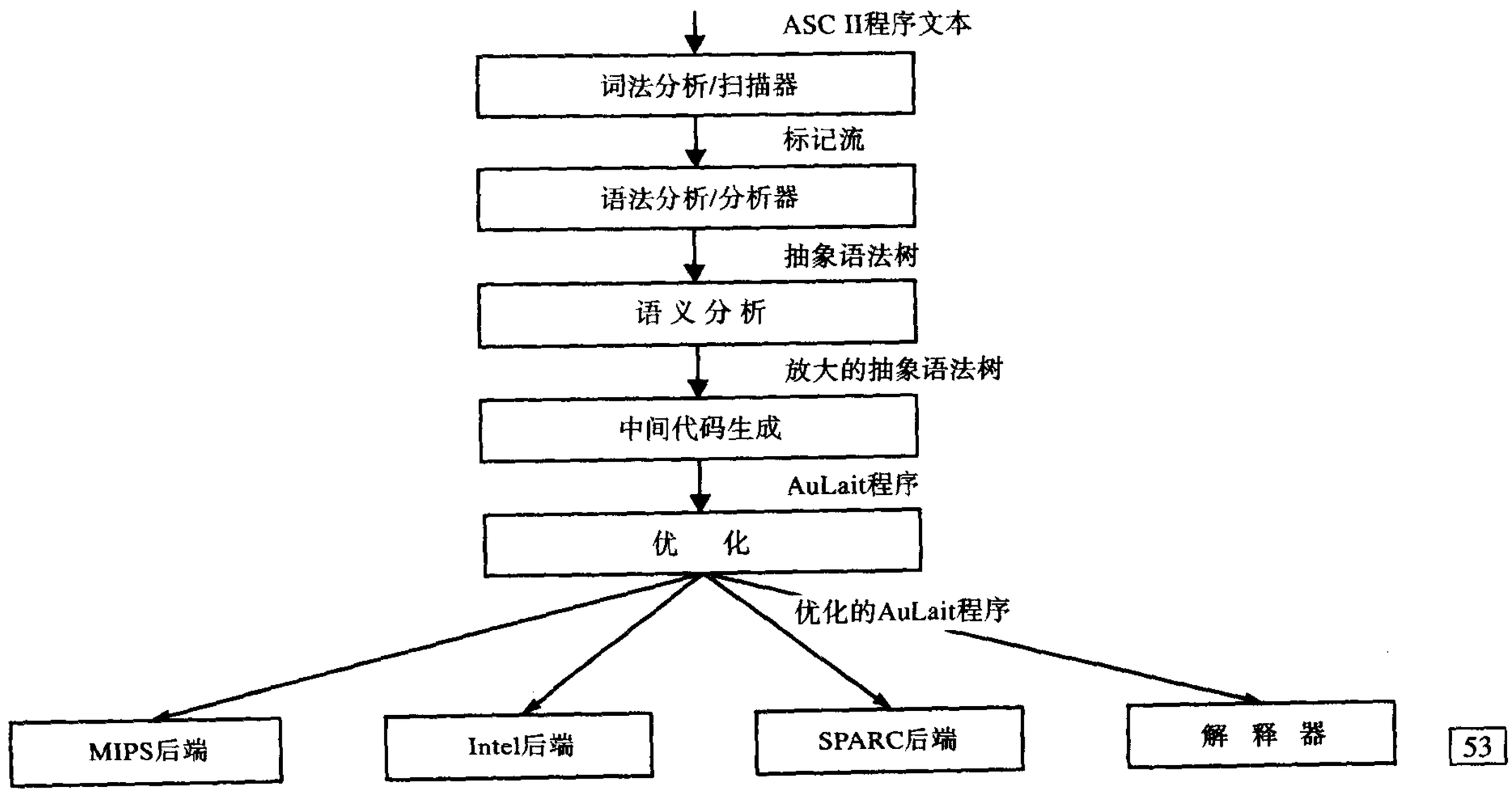
2.2.2 管道和过滤器

管道和过滤器（*Pipes and Filters*）体系结构模式为处理数据流的系统提供了一种结构。每个处理步骤封装在一个过滤器组件中。数据通过相邻过滤器之间的管道传输。重组过滤器可以建立相关系统族。

1. 例子

假设我们已定义了一种新的程序设计语言叫做Mocha（Modular Object Computation with

Hypothetical Algorithms，带假设算法的模块对象计算）。我们的任务是为这种语言创建一个小巧的编译器。为了支持现有的和将来的硬件平台，我们定义了一种运行于虚拟机Cup（Concurrent Uniform Processor，并发一致处理器）上的中间语言AuLait（Another Universal for Intermediate Translation，中间转换的另一种通用语言）。Cup通过解释器或特殊平台后端来实现。AuLait解释器在软件中模拟Cup。一个后端会把AuLait代码转换成一个特定处理器的机器指令以获得最佳性能。



概念上，从Mocha转换成AuLait由如下几阶段构成：词法分析、语法分析、语义分析、中间代码生成（AuLait）和中间代码优化（可选）［ASU86］。每个阶段有定义良好的输入和输出数据。编译过程的输入是表示Mocha程序的ASCII字符序列。系统中的最后步骤——是后端或解释器——采用二进制AuLait代码作为它的输入[⊖]。

2. 语境

处理数据流。

3. 问题

设想正在建立一个必须处理或转换输入数据流的系统。把这样的系统作为单个组件实现是不容易的，这有几个原因：系统必须由几个开发人员创建，整个系统任务自然就分解成几个处理阶段，而且需求很容易变动。

所以你要通过替换或重新排列处理步骤来为将来的灵活性作规划。通过加入这样的灵活性，采用现存处理组件构建系统簇是可以办到的。系统的设计——尤其是处理步骤的内部连接——必

⊖ 名字、人或事件中的任何相似纯属巧合，绝非故意。

须考虑如下因素：

- 未来系统的升级通过替换处理步骤，或重组步骤，甚至通过用户，应该可以做到。
- 不同的语境中小的处理步骤要比大的组件更易于重用。
- 不相连的处理步骤不共享信息。
- 存在不同的输入数据源,例如网络连接或一硬件传感器提供温度读数。
- 可以用多种方式给出或存放最终结果。
- 明确存放中间结果以便在文件散乱目录中进一步处理，如果由用户来做，则易于出错。
- 暂不取消多重处理步骤，例如，在并行或准并行中。

54

处理步骤的分解是否容易主要依赖应用领域和待解决的问题。例如，一个交互式的事件驱动系统就不能分成若干序贯阶段。

4. 解决方案

管道和过滤器体系结构模式把系统任务分成几个序贯的处理步骤。这些步骤采用通过系统的数据流连接——一个步骤的输出是下一个步骤的输入。每个处理步骤由1个过滤器组件实现。过滤器消耗和转发增长的数据——在产生任何输出之前消耗它的所有输入——以达到低延迟而且能够真正地并行处理。系统的输入由诸如文本文件等数据源（*data source*）提供。输出流入数据汇点（*data sink*），如文件，终端，动画程序等等。数据源、过滤器和数据汇点由管道顺序连接起来。每个管道实现相连处理步骤间的数据流动。通过管道联合的过滤器序列叫做处理流水线（*Processing pipeline*）。

5. 结构

过滤器组件是流水线的处理单元。过滤器丰富、提炼或转换它的输入数据。过滤器通过计算和增加信息来丰富数据，通过浓缩或摘录信息来提炼数据。通过以别的表示形式交付数据来转换数据。一个具体的过滤器实现要兼备如下三条基本原理：

- 随后的流水线单元从过滤器中拉出输出数据。
- 前面的流水线单元把新的输入数据压入过滤器。
- 最常用的，过滤器以循环方式工作，从流水线中拉出其输入数据并且将其输出数据压入流水线。

前两种情况表示所谓的被动过滤器，而最后一种情况表示一种主动过滤器[⊖]。一个主动过滤器从把自身作为独立程序或线程开始处理。一个被动过滤器组件通过作为一个函数（拉）或作为一个过程（推）被调用而激活。

55

管道表示过滤器之间的连接，数据源和第1个过滤器之间的连接，最后一个过滤器和数据汇点之间的连接。如果两个主动组件之间是相连的，则管道就使它们同步。这个同步是由先进先出缓冲器完成。如果活动是由链接着的过滤器中的一个控制，则管道可以通过过滤器对被动组件的直接调用来实现。但是，直接调用使过滤器重组更困难。

⊖ 注意，按此定义，所有UNIX过滤器都是主动的。被动过滤器可以是一个陌生的概念。我们引入它以说明管道和过滤器模式不需要语境转换和数据转化的开销也可实现，而且仍然视其为一个有用的概念。

类 过滤器	协作者 • 管道
责任 • 获得输入数据 • 在其输入数据上执行一个函数 • 供给输出数据	

类 管道	协作者 • 数据源 • 数据汇点 • 过滤器
责任 • 转化数据 • 缓冲数据 • 同步主动邻居	

数据源表示系统的输入，它提供一系列相同结构或类型的数值。由文本行组成的文件，或发送数字序列的传感器便构成这种数据源的例子。流水线的数据源既可主动地把数据值推入第一个处理阶段，也可在第一个过滤器拉出时被动地提供数据。

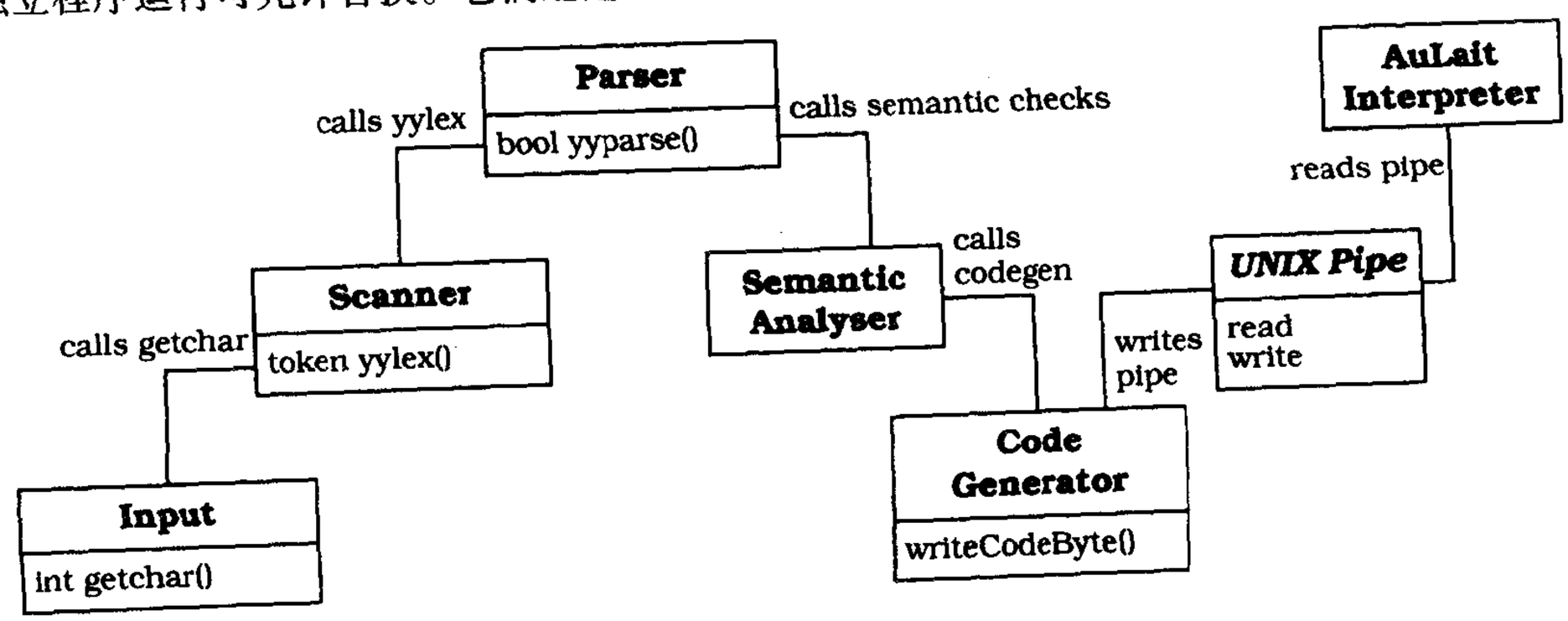
数据汇点收集来自流水线终端的结果。数据汇点可能有两种变体。主动数据汇点把前面处理阶段的结果拉出来，而被动数据汇点则允许前面的过滤器把结果推或写进出。

类 数据源	协作者 • 管道
责任 • 将输入递送到处理流水线	

类 数据汇点	协作者 • 管道
责任 • 消耗输出	

56

➡在我们的Mocha编译器中，我们用UNIX工具`lex`和`yacc`来完成编译器的开始两个阶段[ASU86]。两个工具生成函数——`yylex()`和`yyparse()`——用来嵌入到程序中。函数`yyparse()`主动控制编译器的前端。当需要进一步地输入标记时，它调用`yylex()`。与其他前端阶段的连接组成许多过程调用，嵌入到语法行为规则中，且不仅仅是简单的数据流。这样的嵌入式调用要远比创建一个明确抽象语法树表示且沿管道传输要高效得多。返回端和解释器作为独立程序运行可允许替换。它们通过一个UNIX管道连接到前端。❑

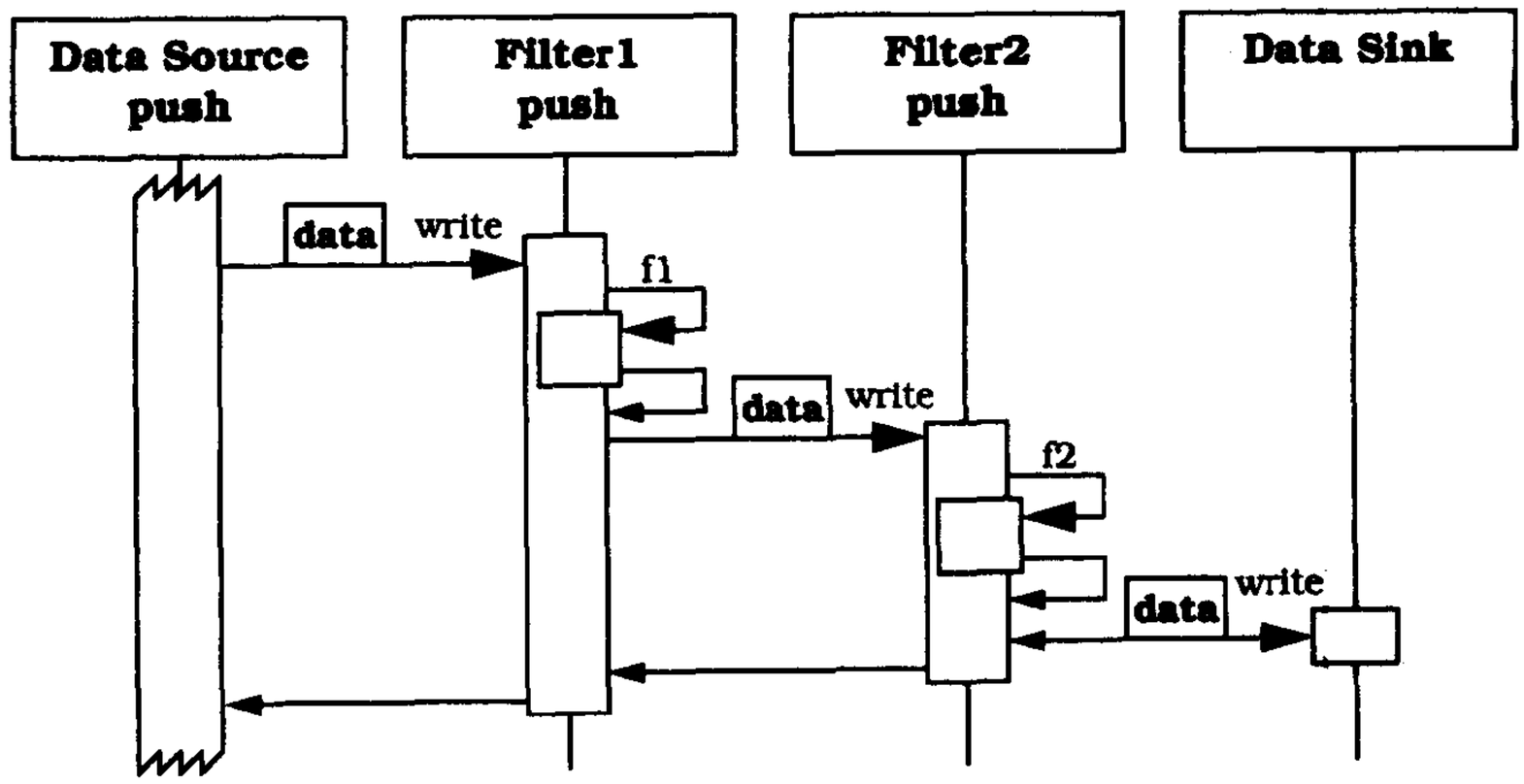


6. 动态特性

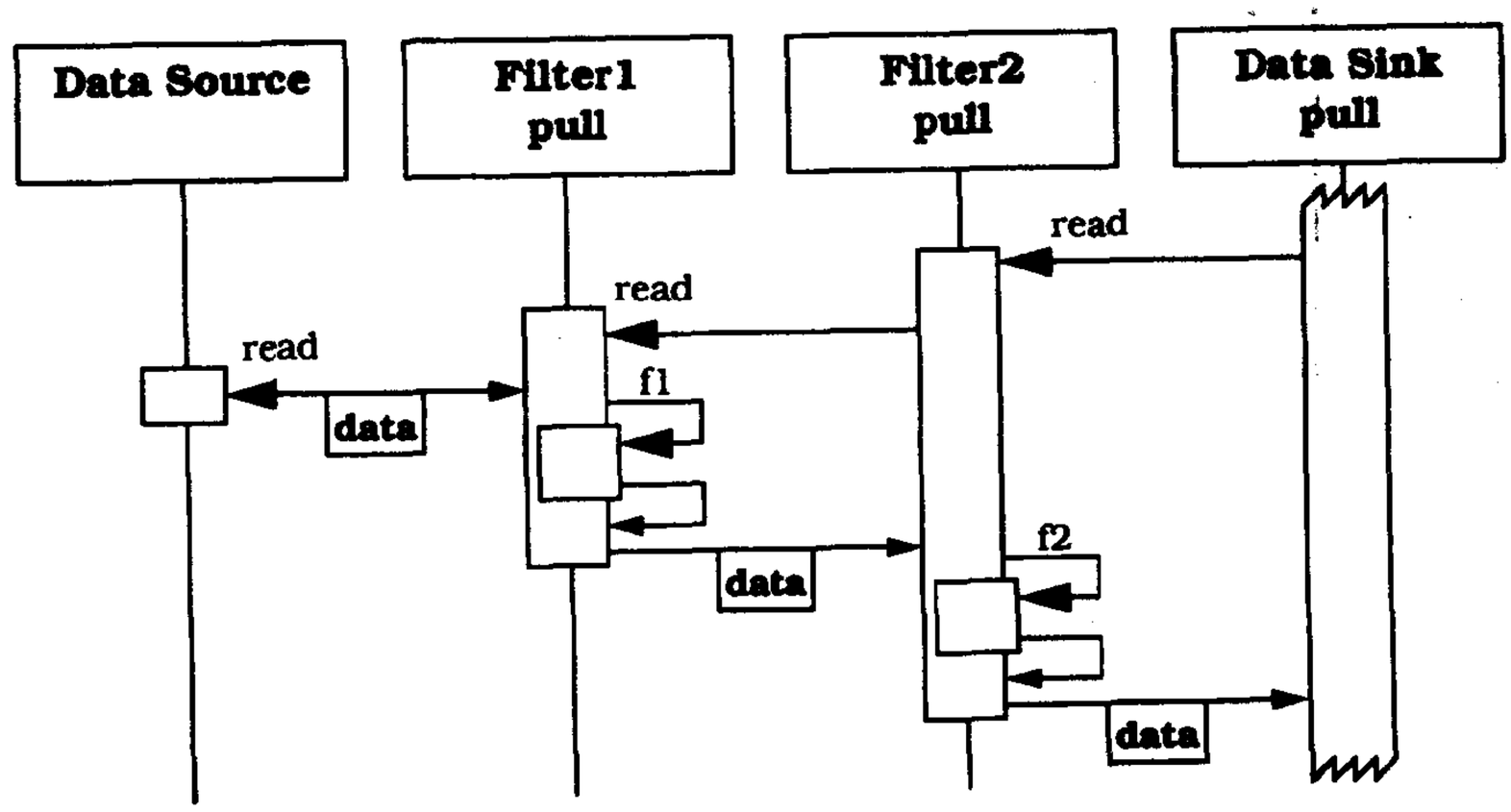
下面的场景给出了相连过滤器之间的控制流的不同选项。假设Filter1在它自己的输入数据上计算函数f1，而Filter2计算出函数f2。最先三个场景给出了直接调用相连流水线组件的被动过滤器，用不同组件控制活动——所以不存在明确的管道组件。最后的场景给出了最常见的情况，即所有的过滤器都是主动的，其间有同步管道。

57

场景I 给出一个推进流水线，其中活动从数据源开始。过滤器活动通过向被动过滤器写数据而启动。



场景II 给出一个拉出流水线，这里控制流始于向数据汇点调用数据。

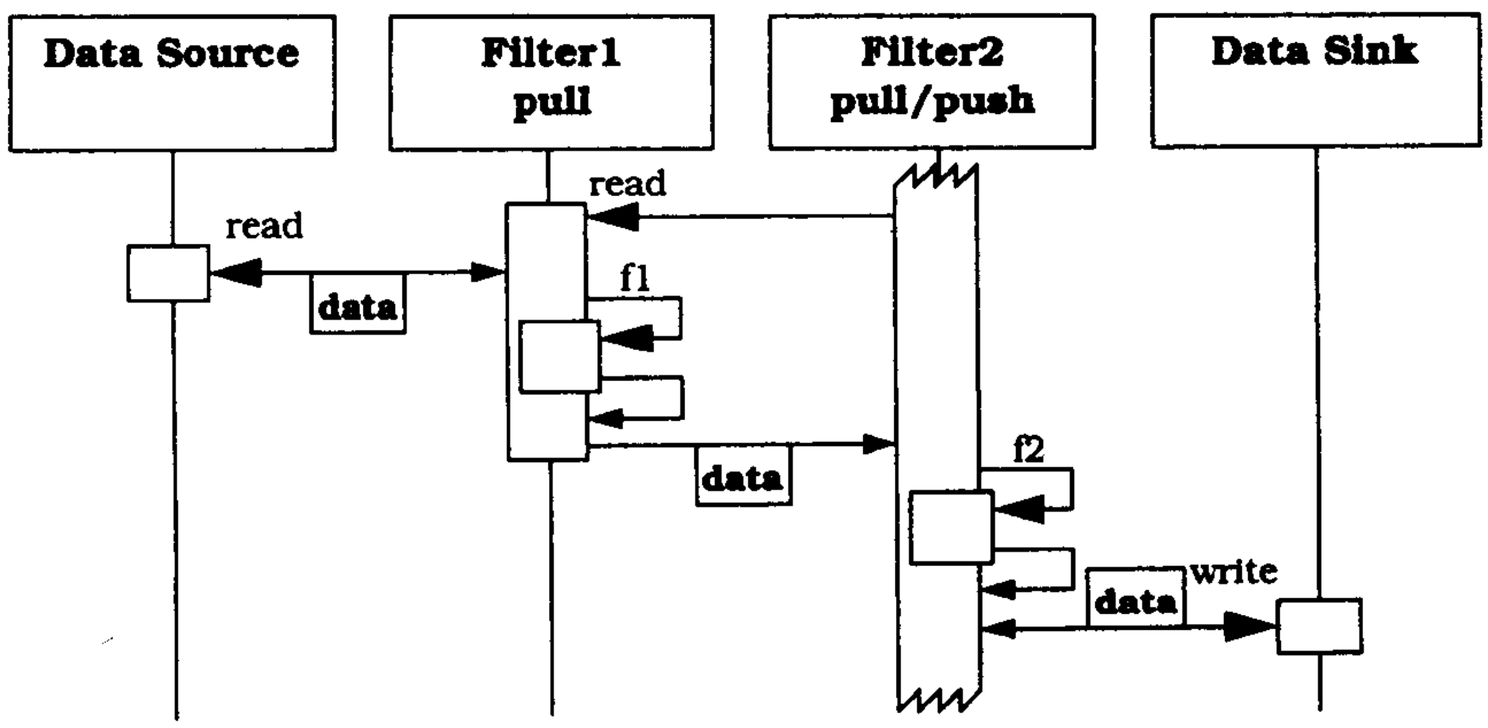


场景III 给出混合了被动数据源和数据汇的推-拉流水线。这里第二个过滤器起主动作用并启动处理。

58

场景IV 给出了一个更复杂的但又是典型的管道和过滤器系统的行为。所有的过滤器都循环地主动拉出、计算并推入数据。所以，每个过滤器以它自己的控制线程运行，比如视为一个

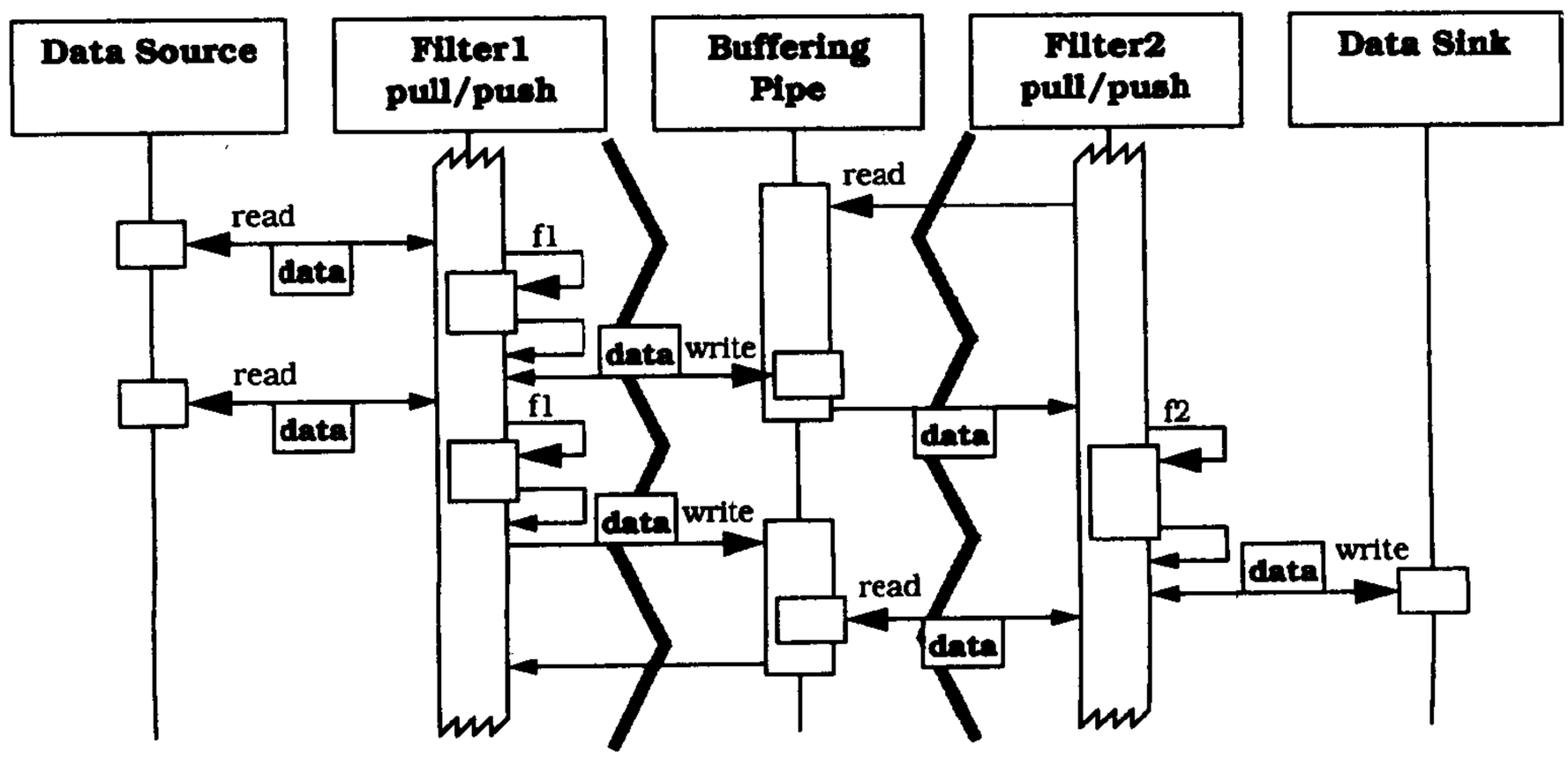
独立的处理器。过滤器被它们之间的缓冲管道同步化。为简单起见，假设管道只缓冲一个值。这个场景也给出了你如何用过滤器获得并行执行。



这个场景中出现了以下步骤：

- Filter2通过读管道来试图获得新数据。由于没有获得数据，数据请求就挂起Filter2的活动——缓冲池是空的。
- Filter1从数据源拉出数据并执行函数f1。
- 接着Filter1把结果压入管道。
- 现在Filter2可以继续了，因为获得了新的输入数据。Filter1也可以继续，因为它没有被管道中满的缓冲池阻塞。
- Filter2计算f2并把结果写入数据汇点。
- 与Filter2活动并行，Filter1计算了下一个结果，并试图把结果压入管道。这个调用被阻塞，因为Filter2没在等待数据——缓冲池是满的。
- Filter2现在开始读已经从管道中获得的新的输入数据。这释放了Filter1，使得它现在也能继续它的处理。

59



7. 实现

实现一个管道和过滤器体系结构是直接的。你可以使用一个系统服务（如消息队列或UNIX管道）来进行管道连接，或选择其他诸如直接调用实现，正如下面第3步~第6步所描述的。这些步骤中的设计决定是紧密相关的，所以可以给它们一种不同于这里给出的顺序。数据源和数据汇点的实现没有详细说明，因为它紧密遵循管道和过滤器的指南。

(1) *把系统任务分成一系列处理阶段。*每个阶段必须只依赖其前一阶段的输出。通过数据流将所有阶段在概念上相连起来。如果你计划通过替换处理步骤来开发一个系统簇，或者如果你正在开发一个组件工具箱，在这一点上你可以考虑改动或重组一些处理阶段。

➡在Mocha编译器中，主要的分离处在创建AuLait的前端和后端之间。前端的进一步结构化为我们提供了扫描器、语法分析器、语义分析器和代码生成器等阶段。我们决定不是去建立一棵详尽的抽象语法树，从语法分析器到语义分析器，而代之以把对语义分析器（sa）和代码生成器（cg）的调用嵌入到yacc的语法规则：

60

```
addexpr  :   term
         |   addexpr '+' term
             { sa.checkCompat($1,$3); cg.genAdd($1,$3); }
         |   addexpr '-' term
             { sa.checkCompat($1,$3); cg.genSub($1,$3); }
```

这意味着我们需要建立一个过滤器，它由语法分析器、语义分析器和代码生成器阶段组成。扫描器，作为一个独立的过滤器组件，处于被动等待状态直到被语法分析器调用。这样我们把函数yylex()连入到前端程序。 ❑

(2) *定义沿每个管道传输的数据格式。*定义一个统一格式可获得最大的灵活性，因为它使过滤器的重组变得容易。在绝大多数UNIX过滤器程序中，数据格式是线性结构的ASCII文本。但是这可能会带来效率问题。例如，浮点数的文本表示也许效率太低而不能沿管道传输，因为需要在ASCII码和浮点表示之间进行反复转换。如果你想既拥有灵活性，又能选择不同的数据表示，则创建一个转换过滤器组件对语义等价的数据进行转换。

你也必须定义如何标识输入结束。如果一个系统服务用于管道连接，则一个输入结束错误条件可能就足够了。对其他的管道实现你可以使用一个特殊数值来标识输入结束。值0、-1、$、control-D、control-Z等都是标识输入结束的常用例子。

➡前端输入是一个Mocha程序，其形式是ASCII字符流或文件。从扫描器传到语法分析器标记是用整数值表示的。函数yylex()返回的或者是所扫描字符的ASCII代码，或者是超出ASCII范围的标记的代码，如一个Mocha关键词。输入结束用0值表示。在前端和后端之间使用的数据格式或解释器是由AuLait字节代码的定义提供的。 ❑

(3) *决定如何实现每个管道连接。*这个决定直接确定了把过滤器作为主动组件实现还是被动组件实现。相连管道进一步定义了一个被动过滤器是由压入数据还是由拉出数据启动。管道连接的最简单情形是相邻过滤器间压入或拉出一数据值的直接调用，如同动态特性部分中前三个场景所给出的。但是如果在过滤器之间使用直接调用，当你想要重组或重用过滤器的组件时，就不得不改动代码。这样的过滤器也很难独立开发和测试，因为测试框架需要调用过滤器组件。

61

使用同步化相邻主动过滤器的分离管道机制可以提供一种更灵活的解决方法。如果所有的管道使用相同的机制，过滤器的随意重组是可能的。管道支持先入先出缓冲以连接相邻过滤器，过滤器每次计算生产和消费数量不等的数据。许多操作系统提供过程内通信服务，如队列或管道等，可用来连接主动过滤器程序。如果不能获得这样的服务，则可以把过滤器作为独立线程实现，管道作为队列来使数据的生产者和消费者同步。

➡因为在Mocha编译器的后端需要灵活性，所以我们在前端和后端之间使用UNIX管道机制。这也允许我们储存编译的中间结果——即AuLait代码——在一个通过另一个后端来进一步分析或转换的文件中。❑

(4) 设计和实现过滤器。过滤器组件的设计同时基于它必须完成的任务以及相邻管道。实现被动过滤器，对拉出动作可作为函数，对推入动作可作为一个过程。在流水线程序中主动过滤器既可以作为一个过程实现也可以作为一个线程实现。

过程之间关联转换的代价和地址空间之间拷贝数据的需要，会严重影响性能。管道的缓冲区大小是一个需要考虑的附加参数。在最多关联转换情况下，一个小缓冲区表现最差。在多个关联转换和数据转换的总开销一定的情况下，用小的主动过滤器组件，可以获得高度的灵活性。

如果想很容易就重用过滤器，以某种方式控制它们的行为是最重要的。已有几种技术可用来把参数传递给过滤器。例如，UNIX过滤器程序允许许多选项在命令行上传输。另一种方法是在执行时使用过滤器可获得的全局环境或仓库。这可以由操作系统，外壳或配置文件来支持。应该仔细考虑过滤器的灵活性和易于使用之间的平衡。根据经验，一个过滤器应该做好一件事。 62

➡Mocha前端从标准输入读程序源代码并在标准输出上创建一个AuLait程序。前端阶段由直接调用来通信。也为AuLait运行创建一个优化程序作为独立的过滤器程序。AuLait解释器可视为一个数据汇点，而预计的后端是产生目标代码作为输出的附加过滤器阶段。❑

(5) 设计出错处理。由于流水线组件不能共享任何全局状态，所以，错误处理很难做到且往往被忽略。至少，错误探测应该是可能的。UNIX为错误消息定义了一个特殊的输出管道stderr，它由大多数为此目的提供的过滤器程序所使用。这样的方式可以表示诸如输入数据、资源受限之中的错误。但是，当过滤器并发运行时，一个单独的错误管道可以以一种不明显的、不可预测的方式混合来自不同组件的错误消息。

如果一个过滤器在其输入数据中探测到错误，它可以忽略输入直到一些清楚表明的分隔出现。例如，如果希望输入的某一行包含一个数值但没包含，则过滤器可以跳到下一输入行。如果可能输入不正确的数据，并且可以容忍不准确的结果，这种方法是有帮助的。

一个基于管道和过滤器模式的系统，很难给出错误处理的一般策略。例如，考虑这样的情况，一个流水线已消费了其输入的3/4，也已产生一半的输出数据，并且某些中间过滤器失败。在许多系统中惟一的解决方法是重启流水线并希望它能无失效地完成。

流水线的再同步可能是一个高级系统的目标，在该系统中，过滤器递增地处理数据。一种选择是引入特殊标记值来给输入数据流做标记。这些标记不变地传到输出。然后，再输入的正确阶段可以重启管道（在失效后）继续处理。另一种选择是用管道来缓冲已消费的数据，如果过滤器失败了就用管道来重启流水线。 63

➡在这个简单的编译器中，我们把错误发送到标准错误管道。涉及语法分析器满足，当探

测到语法错误它能跳过标记，直到扫描程序识别出“；”语句分隔符。如下语法规则中给出的例子便是这样做的，它忽略了在“import”语句中出现的语法错误。

```
import    :    FROM identifier objidentlist ';'
          |    FROM error ';'
                    { mochaerror(errs[E_IMPRT]); yyerrok; }
```

在这个规则中，yacc的特殊标记error匹配所有未识别出的标记直到找到一个分号。语句yyerrok是一个特殊动作，它用在出现语法错误时把语法分析器复位到正常模式。 ❑

(6) *建立处理流水线*。如果你的系统处理单个任务，你可以用一个标准化的主程序来建立流水线并开始处理。这类系统可以从直接调用流水线中获益，其中主程序调用主动过滤器来开始处理。

可以通过提供外壳或其他终端用户能力为过滤器组件集创建各种流水线来增加灵活性。通过允许中间结果存放在文件中，并把文件作为流水线输入加以支持，这样的外壳可以支持流水线的增量式开发。不需要限制到诸如UNIX提供的纯文本外壳，甚至可以用“拖放”交互为流水线的可视化创建开发一个图形化环境。

➥我们的编译器由UNIX外壳命令创建，它建立编辑器或解释器流水线。

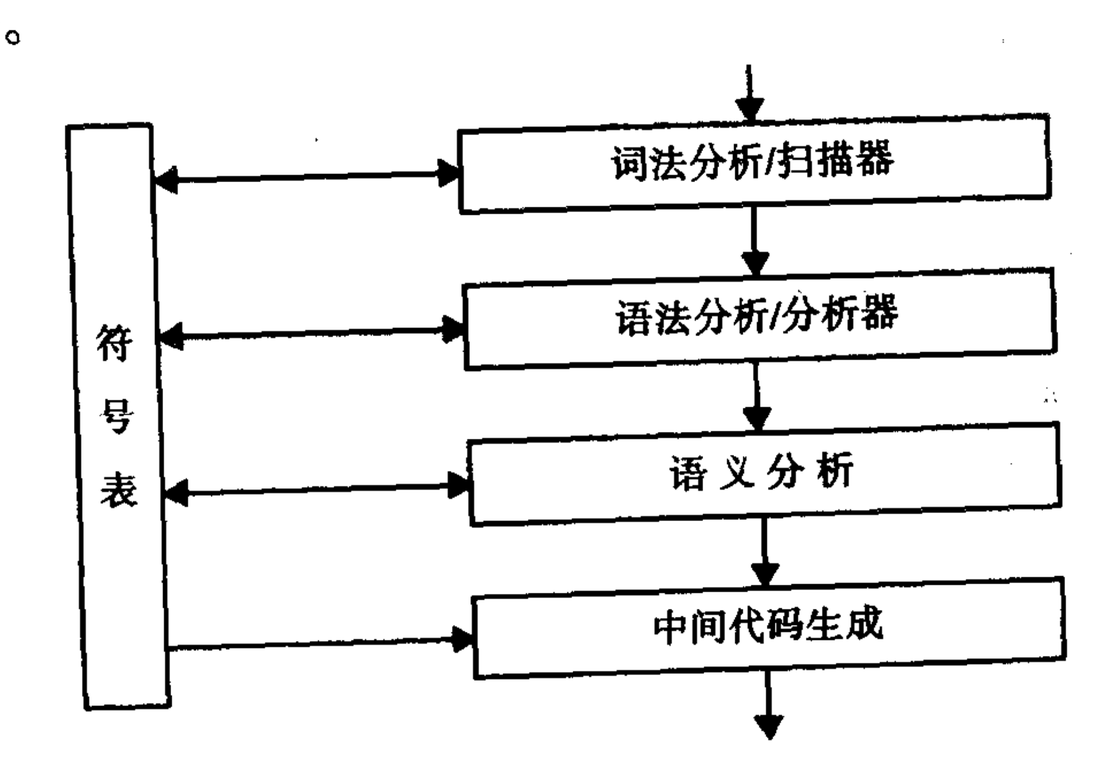
```
# compile and optimize a Mocha program for a Sun
$ Mocha <file.Mocha | optauLait | auLait2SPARC >a.out

# interpret a Mocha program
$ Mocha <file.Mocha | cup
```

Mocha是前端程序，优化器叫做optAuLait，后端按命名习惯叫做AuLait2machine。解释器在它实现了虚拟机之后称为cup。 ❑

64

8. 已解决的例子

Mocha编译器中我们并不严格遵循管道和过滤器模式，我们是把编译器的所有阶段的实现都作为用管道连接的独立过滤器程序。这样做有性能的原因，并且还因为，与第三个强制条件相比，这些阶段共享一个全局状态——符号表。有时它可能通过把全局信息作为附加数据沿流水线传输来移除共享全局状态的需要。但是，这包含更复杂的数据结构和流水线中数据量的增加，增加了性能惩罚。待处理数据由简单类型（如文本行）构成，这样复杂的附加数据结构不得不借助过滤器编码和解码。

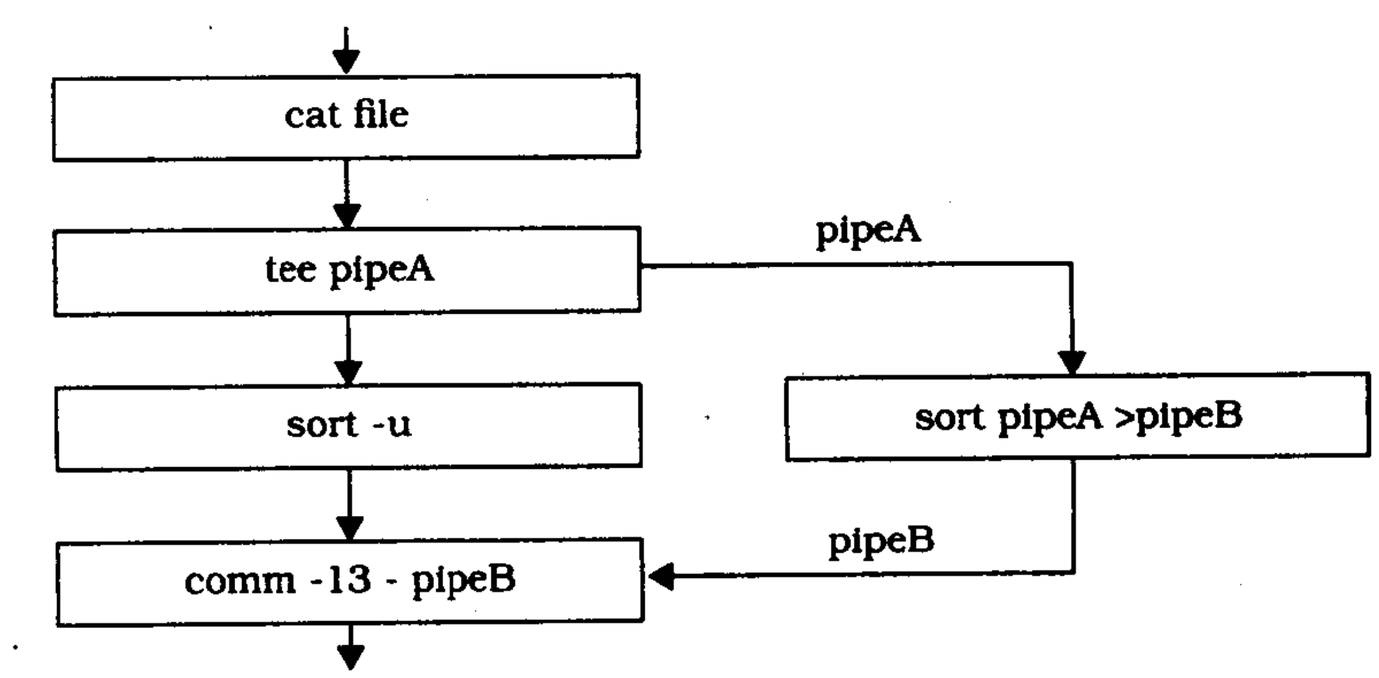


我们把最先四个编译阶段结合成一个单独的程序，因为它们都要访问和修改符号表。这允许把扫描器和语法分析器之间的管道连接用简单的函数调用来实现。后端、解释器或调试器也可以从访问符号表获益。所以，我们按照现有许多编译器的例示，把一些符号表信息，如名称和源行号，编成用于调试的二进制代码。注意这样的符号表信息会大大增加了编译程序的规模。 65

9. 变体

*准备并加入流水线系统。*管道和过滤器模式的单输入单输出过滤器规范可以变化以允许过滤器有不止1个输入或不止1个输出。于是处理就可以用一个有向图来建立，有向图有时甚至包含反馈循环。这种系统的设计，尤其是带反馈循环的系统，需要一个坚实的基础来解释和理解完整的计算——适合使用形式化方法的严格的理论分析和规格说明，以此来证明系统终止并生成预期结果。但是如果我们限定到简单有向无圈图，创建有用系统还是可能的。例如UNIX过滤器程序tee，允许你沿管道把数据写入文件或另一个“命名的”管道中。一些过滤器程序允许把文件或已命名管道和标准输入一样作为输入使用。例如，为建立在文本文件中出现不止一次的所有行的分类列表，我们构建如下外壳程序：

```
# first create two auxiliary named pipes to be used
mknod pipeA p
mknod pipeB p
# now do the processing using available UNIX filters
# start side fork of processing in background:
sort pipeA > pipeB &
# the main pipeline
cat file | tee pipeA | sort -u | comm -13 - pipeB
```



66

10. 已知使用

UNIX［Bac86］常常使用管道和过滤器范型。命令外壳和许多过滤器程序的有效性使这种系统开发方式更普及。对软件开发人员来讲，作为一个系统，在一个“传统”的UNIX系统上最频繁的任务（如程序编译和文档创建）由流水线完成。UNIX流水线的灵活性使该操作系统成为一个过滤器程序的二进制重用和应用集成的合适平台。

CMS流水线［HRV95］是对用来支持管道和过滤器体系结构的IBM大型计算机的操作系统的扩展。CMS流水线的实现遵循CMS的习惯，并定义一个记录来作为沿管道传输的基本数据类

型，以取代一个字节或ASCⅡ字符。CMS流水线以同UNIX一样的方式提供了重用和集成平台。由于CMS操作系统不像UNIX一样使用一个统一形式的I/O模型，所以CMS流水线定义了设备驱动器来作为数据源或数据汇的行为，它允许对流水线中特定I/O设备进行处理。

LASSPTool［Set95］是一个支持数字化分析和图形的工具集。该工具集主要由能用UNIX管道组合的过滤器程序构成。它包含采用旋钮或游标针对类似数值输入的图形化输入设备、数值分析和数据提取用过滤器，以及从数值数据流生成动画的数据汇。

11. 效果

管道和过滤器体系结构模式有以下**长处**：

可能，但不一定需要中间文件。通过把中间结果放在文件中，无需管道即可用独立程序计算结果。这种方法使目录乱七八糟，并且如果在每次运行系统时你不得不建立你的处理步骤，这种方法是易于出错的。此外，它取消结果的增量计算和并行计算。使用管道和过滤器消除了中间文件的需要，但允许在流水线中使用T形连接（T-junction）来研究中间数据。

通过过滤器交换增加了灵活性。过滤器有一个简单的接口，该接口允许它们在处理流水线中方便地交换。即使过滤器组件以推或拉方式彼此直接调用，一个过滤器组件的替换仍然是直接的。例如在编译器例子中用`lex`生成的扫描器可以很容易地用更加高效的手编函数`yylex()`来替换，该函数能够执行同样的任务。由于流水线中的增量计算,过滤器交换在运行时一般是不可能的。

67

通过重组增加了灵活性。结合过滤器组件的重用，主要好处是允许你通过重排过滤器或添加新过滤器来创建一个新的处理流水线。一个没有数据源或数据汇的流水线可以作为一个过滤器嵌入到更大的流水线。应该计划协调系统平台或周围基础结构来支持这种灵活性，正如UNIX中由管道机制和外壳所提供的。

过滤器组件的重用。对重组的支持使得过滤器组件易于重用。如果把每个过滤器都作为主动组件来实现，而基础平台和外壳允许终端用户轻易构建流水线，则可进一步提高重用程度。

流水线的快速原型。前述优点使它易于从现有过滤器中搭建数据处理原型。在用流水线实现了主要系统功能后你可以逐步优化它。例如，可以通过为实时处理阶段开发特定过滤器，或用更有效的管道连接实现流水线来做这项工作。当然，原型流水线可以成为最终系统，如果它完全能够执行所要求的任务。高度灵活性的过滤器（如UNIX工具`sed`和`awk`）强化了这种原型方式。

并行处理提高效率。在一个多处理器系统或网络中，并行地启动一个主动过滤器组件是可能的。如果流水线中的每个过滤器递增地消耗和生成数据，它们就可以并行地执行它们的功能。

应用管道和过滤器模式有如下**不足**：

共享状态信息或者昂贵或者不灵活。如果处理阶段需要共享大量的全局数据，应用管道和过滤器模式或者低效或者不能提供该模式的所有优点。

68

并行处理获得的效率往往是一种假象。这有几个原因：

- 过滤器之间传输数据的代价比起单个过滤器负担的计算的代价来相对要高。对于使用网络连接的小的过滤器组件或流水线尤其如此。

- 一些过滤器在产生输出之前消耗所有输入，或者是任务所要求的（如存储），或者是过滤器代码写得很差，如没有使用应用程序允许的增量式处理。
- 线程或进程之间的关联转换在单处理器机器上通常是一个昂贵的操作。
- 过滤器对管道的同步化可能要经常终止或启动过滤器，尤其当管道仅有一个小缓冲区时。

数据转换额外开销。对所有过滤器输入和输出使用单数据类型来获得高度灵活性导致数据转换额外开销。考虑进行数值计算并使用UNIX管道的一个系统。这样的系统必须在每个过滤器中把ASCⅡ字符转换成实数以及将实现转换成ASCⅡ字符。一个简单的过滤器，如两个数相加，它的绝大部分处理时间消耗在格式转换上。

错误处理。正如我们在“实现”小节的第5步中解释的，错误处理是管道和过滤器模式的惟一弱点。至少应为错误报告定义一个共同策略，并贯穿在系统中。一个具体的错误恢复或错误处理策略依赖于你需要解决的任务。如果你期望流水线在“关键任务”系统中使用且不可能重启流水线或忽略错误，你就应该考虑用其他体系结构（如层）来构建你的系统。 69

参见

层模式更适合于需要可靠操作的系统，因为它比管道和过滤器模式更容易实现错误处理。但是层模式缺少对组件的易于重组和重用的支持，而这是管道和过滤器模式的主要特征。

致谢

这个模式依赖于我们学习、使用和讲授UNIX中获得的经验，所以我们感谢UNIX第1版的设计者和前辈，他们发明并建立了管道和过滤器的使用。

主动和被动流水线组件的差别受到PLoP'95论文“流水线设计模式”［VBT95］的影响。

我们亦感谢Ken Auer、Norbert Portner、Douglas C. Schmidt、Jiri Soukup和John Vlissides，他们对这种模式的［PLoP94］版本提出了有价值的批评和建议。 70

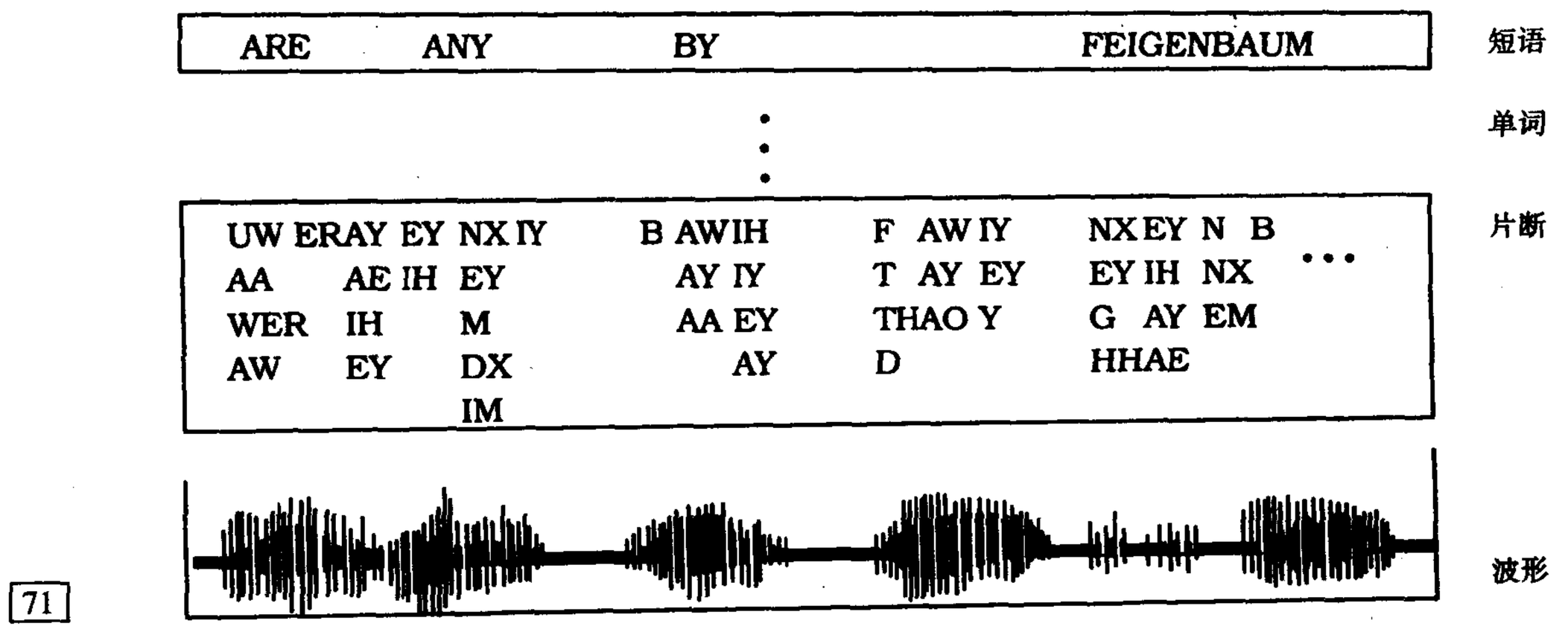
2.2.3 黑板

黑板（*Blackboard*）体系结构模式对于无确定性求解策略的问题比较有用。在黑板模式中有几个专用子系统收集其知识以建立一个可能的部分解或近似解。

1. 例子

考虑一个语音识别的软件系统。系统的输入是记录为一段波形的语音。系统不仅接受单个词，也接受严格遵守语法和特殊应用所需词汇的完整句子，如数据库查询。预期输出是对应英文段落的机器表示。所做转换需要声波-语音的、语言的和统计的专门知识。例如，一个过程将波形分成在语音语境中有意义的片断，如音素[⊖]。在处理序列的另一末端，另一个过程检查候选短语的语法。两个过程都工作在不同的领域。

⊖ “音素”是口语声音的最小单位。“音素”不同于“语音”，后者是能传达一个明显意义的口语中最小划分单位。“语音”可用不同的音素来表达。例如，德语的“r”音和英语卷舌“r”音是不同的音素，但属于相同的语音[Fel84]。



71

该图主要取自［EHLR88］。输入是底部的波形，输出由短语“are any by Feigenbaum”组成。

我们暂且假设对识别语音没有能结合所有必要过程的协调一致的算法——我们在效果小节对这一主题作进一步讨论。刻画问题时把各种糟糕情况都考虑进来：所讲语言的模糊，杂音数据和讲话者在诸如词汇、语音和语法方面的个人特色。

2. 语境

一个不成熟的领域，其中没有相近的已知方法或可行的方法。

3. 问题

黑板模式针对那些在把原始数据转换成高层数据结构（如图、表或英语短语等）方面没有可行的确定的求解方法的问题。视觉、图像识别、语音识别和监视等领域就包含这类问题。它们由一个问题刻画，一旦分解成若干子问题，则跨几个领域的专业知识。部分问题的求解方法需要不同的表示和范型。在许多情况下不存在预先确定的策略用于“部分问题解法”应如何与它们的知识相结合。这和功能分解相反，功能分解中计划几个求解步骤，这样它们的活动顺序是强制编码的（hard-coded）。

在某些上述问题领域中，可能不得不采用不确定的或近似的知识来求解。每个转换步骤也可以产生用几个不同的求解方法。在这样的情况下，对于大多数情况找到最优解法，或次优解法，对其余情况没有解，这通常就足够了。所以一个黑板系统的局限是不得不仔细做文档，而且如果重要的决定要依赖于它的结果，则结果必须验证。

下面几条强制条件影响了这类问题的解决方案：

- 在合理的时间内，解空间的完全搜索是不可行的。例如，你在1000个单词的词汇量中考虑一个最多10个词的短语，单词排列数达到1000^{10}。
- 由于领域还不成熟，所以你需要有对同一个子任务进行不同算法的试验。由于这个原因，单个模块应该是易于替换的。
- 可以有求解部分问题不同的算法。例如，波形中的语音切片的探测和基于词或词组的短语

72

生成是无关的。

- 输入，涉及中间环节和最终结果，有不同的表示，并且算法依据不同的范型来实现。
- 一个算法输入往往是其他算法的输出。
- 不确定数据和近似解也包含在内。例如，说话往往包含暂停和外来声音。这些很大程度上歪曲了信号。信号解释的过程也是易于出错的。识别目标竞争选项可能出现在过程的任何阶段。例如，很难辨别“till”和“tell”。单词“two”和“too”甚至有相同的发音。英语中其他的例子还有很多。
- 使用不相关算法引起潜在的平行性。如果可能你应避免严格顺序解法。

对这种复杂的不确定性问题，采用人工智能（AI）系统取得了一定的成功。在“经典”专家系统结构中，系统的输入和中间结果保持在工作存储器中。使用存储器的内容借助推理机和知识库来推出中间结果。重复这样的操作步骤直到满足了某些完成条件。

这类专家系统对语音识别系统是不充分的。原因有以下三点：

- 所有的部分问题都用同样的知识表示来解决。但是，语音识别中包含的组件工作在这样的知识领域：尽可能宽地区分一个波形的分割及候选短语的语法分析。所以它们需要不同的表示。
- 专家系统结构只提供一个推理机来控制知识的运用。不同表示的不同部分问题需要独立的推理机。
- 在一个“经典”专家系统中，控制隐含在知识库的结构当中，例如，基于规则系统的规则的顺序。这和许多AI系统中的观点是一致的，即“问题求解在于搜索”和“知识用于修剪和指导搜索”。这意味着要在节点中包含问题解的搜索树中搜索，并利用知识来指导从搜索树的根——这里所有解都是可能的——到单个叶子的搜索路线。

对语音识别问题，观点“问题求解在于专家调集他们的知识”更合适。也就是说，知识的表示片断必须在一个适当时机应用，而不是按预先确定的顺序应用。

4. 解决方案

黑板体系结构背后的思想是合作工作于一个公共数据结构上的独立程序集。每个程序专门用来解决整个任务中的一个特定部分，所有的程序一起工作以解决问题。这些专用程序彼此独立。它们彼此不互相调用，它们的活动也没有预先确定顺序。实际上，系统的执行方向主要由前进的当前状态确定。一个中心控制组件评估处理的当前状态，协调专用程序。该有向数据控制机制涉及到机会主义的问题求解，这使得采用不同算法来试验成为可能，而且允许试验驱动的启发方法来控制处理。

在问题-求解过程中系统通过部分求解方案的组合、更改和否定来工作。每一个这样的解均表示一部分问题和其解法的一定阶段。所有可能解的集合称为解空间，并被组织成几个抽象层次。解的最低层次由输入的一个内部表示构成。整个系统任务的潜在解决方案处于最高层次。

选择名称“黑板”是因为记住这样一种情形，即人类专家坐在真实黑板的前面并一起工作来解决一个问题。每个专家独立评估解法的当前状态，并可在任何时候到黑板上添加、更改或删除信息。人们往往要决定接下来谁去访问黑板。在我们描述的模式中，如果可用的组件超过一个，仲裁者（*moderator*）组件决定程序执行的顺序。

5. 结构

把系统分成一个叫做黑板的组件和一个控制组件，黑板组件由知识源集合构成。

黑板是中心数据仓库。解空间的元素和控制数据都存储在这里，我们对所有能在黑板中出现的数据元素使用术语“词汇”。黑板提供了一个接口使所有知识源能由它读出和写到它上面去。

解空间的所有元素都能出现在黑板上。对在问题求解过程中构建并放在黑板上的解，我们使用术语“假设”或“黑板入口”。在过程后期拒绝的假设要从黑板中删除。

一个假设往往有几个属性，例如它的抽象层次，即它和输入在概念上的距离。低抽象层次的假设的表示和输入数据表示仍然很类似，而高抽象等级的假设处在和输出同一个抽象层次上。别的假设属性有假设真值的估计度或由假设覆盖的时间区间。

指出假设之间的关系往往是有用的，如“部分”或“支持”。

➡语音识别例子的解空间由声波-语音的言语片断构成。抽象层次分为符号参数、声波-语音片断、音素、音节、单词和短语。

75 音节真实度是根据音节的理想音素序列和假设的音素间的匹配质量来估计的。

语音信号的波形记录在一个时间轴上，对应2.2.3节“例子”小节图中的X轴。每种解法都有一个属性，该属性明确规定了它所描述的X轴上的间隔。

黑板可以看作一个三维问题空间，言语的时间线在X轴上，抽象等级的递增在Y轴上，所选解决方案在Z轴上［Nii86］。 ❑

*知识源*是指求解整个问题特定方面的分离的独立的子系统。它们一起模拟了整个问题领域。它们中没有哪个能单独解决系统的任务——只有通过集成几个知识源的结果才能创建一个求解方案。

➡在语音识别系统中，我们为以下部分问题明确给出求解方案：定义声波-语音片断，创建音素、音节、词和短语。对这些部分问题中的每一个，我们定义一个或几个知识源。例如，在单词等级一个知识源可以从相邻音节创建单词，而在同一等级的另一个知识源根据邻近词对单词进行验证。

注意，从波形到短语的转换不一定是一个严格的顺序过程。整个波形不必首先转换成片断，所有片断再转换成音素，接着转换成音节和词，再创建短语。一部分波形是可能已经被转换成词，另一部分在单词级可能已经被拒绝又返回到音素级，第三种情况根本不去分析，直到有足够的证据来说明存在短语等级的解决办法。 ❑

知识源之间不直接交流——它只是从黑板读或写。所以它们不得不理解黑板上的词汇。我们在“实现”小节探索了这方面的内容。

知识源往往在两个抽象等级上起作用。如果一个知识源实现前向推理，一个特殊解法被转化成更高层次解法。反向推理用到的一个知识源为支持一种解法在较低等级探索，并且，如果

76 推理过程不能给出对该解决方案的支持，则可反过来参考一个更低等级。

每个知识源负责了解在什么条件下对解法有用。所以知识源分成条件部分和行动部分。条件部分评估求解过程的当前状态，即写在黑板上的内容，用以确定它是否有用。行动部分生成能导致黑板内容改变的结果。

类 黑板	协作者
责任 • 管理中心数据	

类 知识源	协作者 • 黑板
责任 • 评价其自身的可应用性 • 计算一个结果 • 更新黑板	

➥我们的语音识别系统有多个知识源，这些知识源将处在同一层次上几个假设连续不断地转换成下一个更高等级上的单个假设。例如，一个短语从词的选择来创建，它们一起横跨过某一时间区间对应某个短语。其他知识源在相同等级预测新的假设。例如，一个知识源用来预测语法上先于或后于某一给定短语的可能单词。我们也定义了一个知识源来验证基于下一较低等级信息上的所预测的假设。这估算了一个预测词和跨越同一时间段片断集之间的一致性。 ❑

控制组件运行一个循环来监视黑板上的改动并决定接下来采取什么动作。它根据一个知识应用策略安排知识源评估和行动。这种策略的基础是黑板上的数据。

策略可以依赖控制知识源。这些特殊的知识源不必直接对黑板上的解法有用，但执行作出控制决定所基于的计算。典型的任务是进展可能性的估计，或知识源执行的计算代价。它们的结果称为控制数据并也放在黑板上。 77

类 控制	协作者 • 黑板 • 知识源
责任 • 监视黑板 • 安排知识源活动	

黑板可能达到没有知识源可应用的状态，理论上这种情况是可能发生的。在这种情况下，系统传送结果失败。在实际应用当中，更可能做的事是在每个推理步骤引入几个新的假设，以及可能在下一步“推翻”的假设数。所以问题限制到采取哪个选项而不是找到一个可应用知识源。

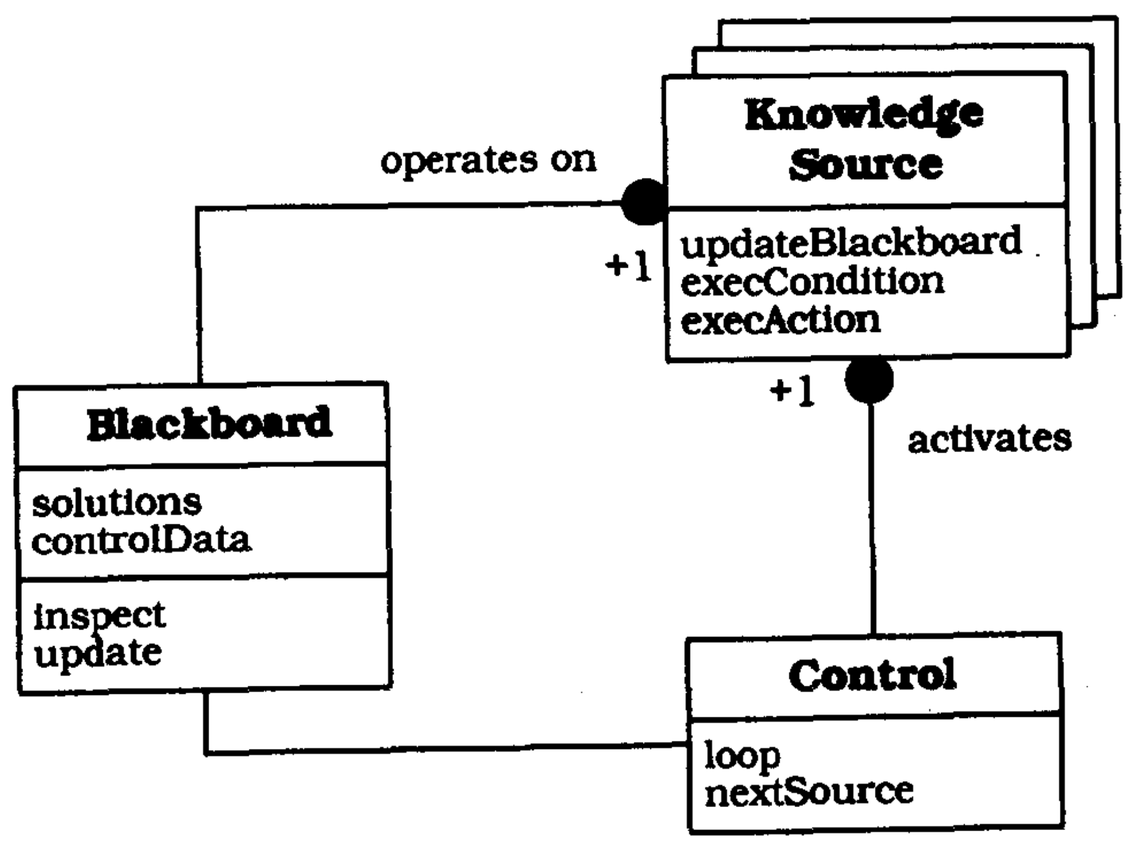
在控制组件中有一个特定知识源或一个过程确定系统何时应停止，最终结果是什么。一旦找到一个可接受[⊖]假设，或者系统的空间资源或时间资源被耗尽，则系统停止。

⊖ 在“实现”小节，我们考虑何时接受或拒绝顶层解决方案的问题。

下图描述了黑板体系结构中的三个组件之间的关系。黑板组件定义了两个过程：inspect和update。知识源调用inspect并检查黑板上的当前解。update用于改动黑板上的数据。

78

控制组件运行一个循环来监视黑板上的改动并决定接下来采取什么动作。我们把负责这个决定的过程叫做nextSource()。



6. 动态特性

下面的场景描述了黑板体系结构的模式，它基于我们的语音识别例子：

- 启动控制组件的主循环。
- 控制组件调用nextSource()过程来选择下一个知识源。
- nextSource()通过观察黑板首先确定哪个知识源是潜在的贡献者。在这个例子中我们假设候选知识源是分片、音节创建和单词创建。
- nextSource()唤醒每个候选知识源的条件部分。在这个例子中，分割、音节创建和单词创建的条件部分检查黑板以确定它们对解决方案的当前状态是否有贡献及如何作贡献。
- 控制组件选择一个知识源到好状态（oke）中，并选择一个假设或一个假设集作为工作的基础。在本例中选择的作出是根据条件部分的结果。在其他情况下，选择也是基于控制数据。它把知识源的动作部分运用于假设。在我们的语音识别例子中，假设音节创建是最有希望的知识源。音节的动作部分检查黑板状态，创建一个新的音节并更新黑板。

79

7. 实现

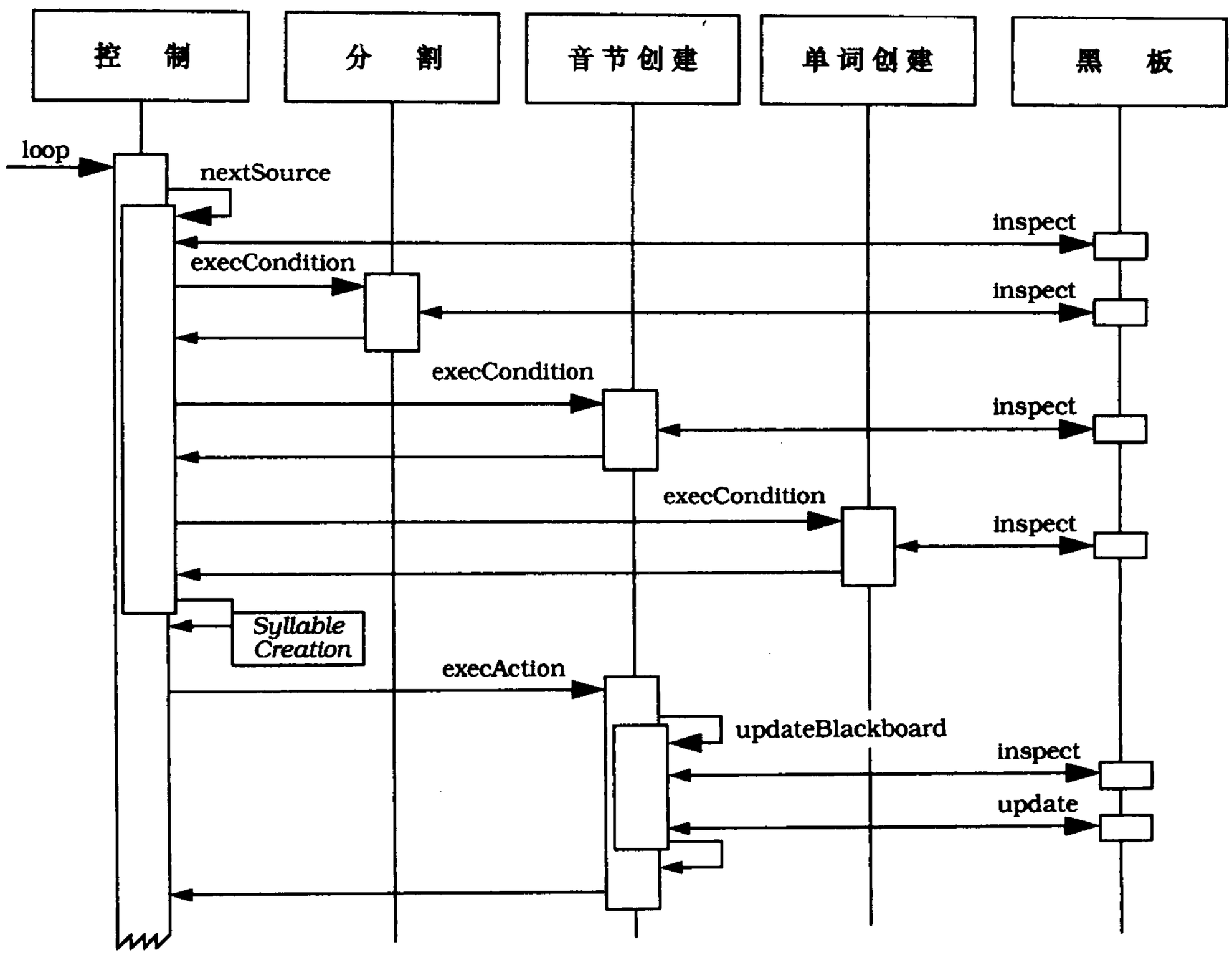
为实现黑板模式，需要实施以下步骤：

(1) 定义问题：

- 详细指出问题领域和找出一个解决方案所需知识的一般领域。
- 详细检查系统的输入。确定输入的任何特殊属性，如噪声内容或一个主题的变化——即输入是否包含有规律的模式它随时间而缓慢变化？

80

- 定义系统的输出。详细指明正确需求和失败-安全行为。如果你需要结果信度的估计，或者有这样的情形，在其中系统向用户请求进一步的资源，记录这些内容。
- 细化用户是如何与系统交互的。



➥在语音识别领域中，对系统很重要的知识领域是声学、语言学和统计学。输入是一个说话者的声音信号序列。数据是嘈杂的。如果系统允许说话者对某一段重复几遍，输入就包含“主题的变体”，如前所述。预期的输出是对应所说短语的一个书面的英文短语。用于数据库查询接口时，系统能容纳偶然的误解。如果我们在10%情况下不得不重复一个查询，这系统仍然是有用的。 ❑

(2) *定义问题的解空间*。我们把中间的和顶层解放在一边，部分和完整解放在另一边。顶层解处于最高抽象等级。其他等级的解是中间解。完整解解决整个问题，而部分解解决问题的一个部分。注意完整解可以属于中间等级，部分解可以是顶层的。

➥在语音识别中，完整的顶层解是相对于已定义的词汇和语法来讲正确的短语。完整中间解是声波-语音序列或描述整个口语片断的语言学元素序列。解的部分就是元素本身。 ❑

所以执行以下步骤：

- 准确指出是什么构成了顶层解决方案。
- 列出解决方案的不同抽象等级。
- 将解决方案有序地放入一个或多个抽象层次中。
- 找出能独立工作的完整解的细分，例如短语的词或一幅图或一个领域局部。

81

(3) 将求解过程分成几个步骤：

• 定义解是如何转换成更高等级的解决方案的。

• 描述在同一抽象等级上如何预测假设。

• 细化如何通过找出在其他等级中对它们的支持来验证预测的假设。

• 详细指明可以用于排除部分解空间的知识类型。

➡为把有关音节层的解转换单词层的解，我们提供一个词典，它把1个音节与所有发音包含该音节的词相关联。 ❑

语法的和统计学的知识在对词序列搜索剪枝时是有用的。例如，一个形容词通常后跟另一个形容词或一个名词，这样的启发知识可以用于减少计算时间。

(4) 把知识分成和一定子任务相关联的专门知识。这些子任务往往对应特定领域。系统可能有一些子任务对某些未定情形由人类专家来作出决定，甚至由人类专家来替换缺省的知识源。在下述意义下，知识源必须是完整的：对绝大多数输入短语，至少存在一个知识源动作的可能序列，它能导出一个存在的可接受的解。

➡知识源的例子诸如分片，音素创建，音节创建，词创建,短语创建,词预测和词验证。 ❑

(5) 定义黑板的词汇。详细拟定解空间的最初定义和解的抽象等级。找到解的一种表示，它允许所有的知识源从黑板上读并对黑板有贡献。这并不意味着每个知识源必须理解每个黑板入口，但每个知识源必须能确定它是否能用一个黑板入口。如果有必要，提供在黑板入口和知识源内部表示之间的转换的组件。这允许知识源可以很容易替换，彼此的表示和范型互相独立，并在同一时间利用各自的结果。

82

➡在语音识别例子中，每个假设有一个统一的属性-值结构。一些属性必须包含在所有的假设中，而其他是可选的。元素名称、抽象等级和由假设所覆盖的时间区间都在所需属性之中。真实度的估计是可选的。例如，黑板可以包含如下登录：

```
ABOUT+FEIGENBAUM+AND+FELDMAN+] (phrase) (48:225) (83).
```

依据抽象等级，每个知识源能确定它是否能在一个假设下工作。例如，知识源负责分片，并不理解黑板登录中的符号“+”和“]”。通过读属性抽象等级的值，它知道假设是一个短语，因而它不检查其他属性。 ❑

为评估黑板的内容，控制组件必须能理解它。所以黑板的词汇不能只定义一次，但词汇定义随知识源和控制组件的定义一起进化。设计过程中的某些点上，词汇必须稳定；允许与知识源稳定接口的开发。

(6) 指出系统的控制。控制组件实现一个机会主义的问题求解策略，以决定允许哪个知识源来对黑板进行改动。这个策略的目的是构建一个当作结果来接受的假设。但什么时候一个假设可以接受？由于一个假设的正确性在严格意义上是不能验证的，我们的目标是构建可能在解空间中的、最可信的、完整的顶层解。

一个假设的可信度是它正确的可能性。我们估计一个假设的可信度是通过考虑它的所有貌似真实的选项，以及从输入数据中得到的每个选项的支持度。可信度等级可以是一个从0~100之间的某数值。一个假设是可接受的,如果它处在顶层，是完整的,并且其评估所得可信度达到一个

阈值,如85。为找出一个可接受的假设,系统删除低可信度的假设,并检查与输入数据一致的互相支持的假设簇。 83

最简单的情况是当黑板一旦被改动，控制策略就咨询所有知识源的条件部分，并随机选择一个可应用知识源来采取行动。但是，这种策略往往过于低效，正如接近可接受假设的改进是缓慢的一样。好的控制策略的设计是系统设计的最困难部分。它往往由尝试组合几种机制和几个部分策略的冗长过程构成。策略模式［GHJV95］在这里对支持控制策略的替换是有用的，甚至在运行期也是有用的。成熟的控制策略可以由一个专门的基于知识系统来实现。

下面的机制优化了知识源的评估，这样就提高了控制策略的效果和性能：

- 把对黑板的改动分类成两种类型。一种改动指出包含一个新的可应用知识源集合的所有黑板改动，另一种指出其他改动。在第二类改动之后，控制组件选择一个未对所有条件部分的再一次调用的知识源。
- 黑板的关联种类随可能应用的知识源集而一同改动。
- 控制的聚焦。焦点既包含黑板上的可以继续工作的部分结果，也包括具有优先性的知识源。
- 创建一个队列，队列中的知识源按其执行应用等待来分类。通过使用一个队列，你可以保存有关知识源的有价值的信息而不是在每次黑板改动后将它丢弃。

控制策略利用启发式知识确定启动哪一个可应用知识源。启发式知识是基于经验和猜测的规则。记住，好的启发式方法经常有效，但并不始终有效。这里列出一些能被控制策略所用的启发式的例子：

- 以优先顺序排列可应用知识源。这样一个优先级计算的基础是对知识源条件部分和可能的其他信息（如利用知识源获得改进的潜力），以及其应用代价的评估。控制组件要考虑知识源的贡献来决定优先级。在这种情形下，在决定哪个会对黑板作出改动之前，它必须执行所有可应用知识源的动作部分。如果系统使用一个队列，则每个知识源的优先级以其入口存储。对黑板的改动可导致优先级的改动或队列中知识源的删除。 84
- 提出低等级或高等级假设。如果这是所用的惟一策略，则控制策略不再是机会主义的，而是实现正向链接或反向链接。
- 提出覆盖问题大部分的假设。
- “孤立体（island）驱动”。这种策略假定，一个特殊假设是一个可接受解的部分，并被视为“确定性孤立体”。于是，优先激活以这个假设为基础的知识源，而且删除持久搜索更高优先级的别的假设的需求。

如果控制组件展示复杂和独立子任务，则为每个子任务定义一个控制知识源。例如，对可应用知识源的优先级计算本身就能作为一个专门的控制知识源来实现。

(7) *实现知识源*。根据控制组件的需要，把知识源分成条件部分和动作部分。为维护知识源的独立性和可替换性，不能作出任何和其他知识源或控制组件有关的假设。

可以在同一个系统中用不同的技术来实现不同的知识源。例如，一个可以是基于规则的系统，另一个是神经网络，第三个是惯用函数集。这意味着知识源自身可以根据各种体系结构模式或设计模式来组织。例如，一个知识源可以用层模式设计，而另一个可以根据映像模式来构建。如果你倾向于用面向对象技术来开发系统，但你的知识源采用另一种范型来实现，采用外 85

观模式［GHJV95］来“包装”它们是有意义的。

8. 变体

产生式系统。这种体系结构用在OPS语言中［FMcD77］。在该变体中，子程序用条件-动作规则来表示，数据在工作存储器中全部可获得。条件-动作规则由指明条件的左端和指明动作的右端共同构成。只有当条件满足并选择了这条规则之后才执行这个动作。选择是由“冲突消解模块”作出的。一个黑板系统可以视为原始产生式系统形式主义的基本扩展：规则两端允许是任意程序，提高了工作存储器的内部复杂性。冲突消解采用了复杂调度算法。

仓库。该变体是黑板模式的泛化。该变体的中心数据结构称为仓库。在一个黑板体系结构中，中心数据结构的当前状态，同控制组件一起，最后激活知识源。相反，仓库模式不指明内部控制。一个仓库体系结构可以被用户输入或一个外部程序所控制。例如，一个传统的数据库，可以看成一个仓库。工作在该数据库上的应用程序对应于黑板体系结构中的知识源。

是仓库系统而不是黑板系统的例子在[SG96]中给出：“编程环境往往由工具集、一个程序共享的仓库和程序段构成。甚至被习惯视为流水线体系结构的应用，更准确地可以用仓库系统来解释……”。例如，编译器，过去总是描述成流水线，有时也用流水线模式来实现[⊖]。现代的编译器有一个拥有共享信息（如符号表和摘要语法树）的仓库。编译阶段对应在仓库上的知识源操作。这种体系结构提供了增量问题求解办法：

86

- *扫描器*读出尚未定义的标识符。
- *语法分析器*识别出由标识符描述的语法单元。
- *代码生成器*随后突增并创建对应机器码，如果有的话。

9. 已知使用

HEARSAY-Ⅱ。最早的黑板系统是20世纪70年代早期的HEARSAY-Ⅱ语音识别系统。它是作为对一个文献数据库的自然语言接口来开发的。它的任务是回答对文档的查询，从人工智能出版物的摘要集合中检索文档。系统的输入是语音信号，从语义上解释这些信号，并转化成一个数据查询。［EM88］给出该项目的详细描述和追忆的观点。HEARSAY-Ⅱ有选择的方面亦可作为这种模式的运行例子。下面的段落讨论了它的控制方面。

在HEARSAY-Ⅱ中，知识源的条件部分标识了适合于该知识源行为的黑板上假设的配置。该子集称作激励框架。例如，生成短语假设的知识源的条件部分寻找相邻的词或短语假设。条件部分也预测一个知识源将要执行的适当动作的形式化描述，称为反应框架。例如基于音节的词假设器的反应框架意味着它的动作将在词这一级生成假设，假设在X轴上覆盖的区间至少包含激励框架。

HEARSAY-Ⅱ的控制组件由以下几个部分构成：

- *控制焦点数据库*，它包含一张黑板改动的原始改动类型表，以及对每个改变类型可能执行的那些条件部分。原始改动类型的例子，如“新音节”或“自底向上创建的新词”——说明了出现在黑板上的一个新词而且它是用较低等级假设做出的推论。

87

⊖ 有关流水线体系结构的更多信息，参见管道和过滤器模式，那里，我们详细解释了按照管道和过滤器构造编译器常常不是一个好主意的理由。

- 调度队列，它包含指向知识源条件部分或动作部分的指针[⊖]。
- 监视器，它跟踪黑板上每个改动。基于对应的原始改动类型，监视器把可应用条件部分的指针插入到调度队列中。如果一个条件部分被实际执行了而且计算后的反应框架也不为空，则把相匹配的动作部分的指针放入调度队列中。
- 调度器，它使用试验驱动启发方法来计算在调度队列中等待的条件部分和动作部分的优先权。这个估计是基于特定的激励框架和反应框架。它也考虑整个黑板状态信息，比如，在相同的X轴区间上的几个竞争假设哪个对来自较低等级的假设支持最好。调度器最终选择对执行具有最高优先级的条件部分或动作部分[LeEr88]。

HEARSAY-Ⅱ的设计者为他们的知识应用策略结合了几种解决求解的技术。第一种是自底向上方法，其中直接从数据来综合，逐渐达到抽象层次。第二种是自顶向下策略，其中递归产生较低等级的假设直至产生最低等级上的假设序列，后者能对原始输入进行测试。对这些方式的正交，HEARSAY-Ⅱ使用“生成和测试”策略，其中一个知识源生成假设，由另一个知识源评估它们的有效性。

HASP/SIAP。设计HASP系统用来探测敌人的潜艇。在该系统中，水中听音器阵列通过收集声纳信号来监视海域。黑板系统解释了这些信号［Nii86］。HASP是一个基于事件的系统，意思指特定事件的出现意味着新信息是可获得的。黑板用作随时间演化的“态势板”。由于不断地收集信息，新的和不同的信息就存在冗余。HASP处理多重输入流。除了来自水中听音器的低层数据，它也接收从智能或其他来源收集的态势的高层描述。 88

CRYSALIS。设计这个系统用来推断来自X射线衍射数据的蛋白质分子的三维结构[Ter88]。系统引入了几个黑板体系结构的特征。黑板分成几个称为画板的部分。每个画板有它自己的词汇和层次。可以通过知识源来限制对某些画板的访问。CRYSALIS使用一个数据画板和一个假设画板。知识源组织成等级。只有最低等级包实际创建和修改假设的数据源。其他等级由控制知识源构成。CRYSALIS是第一个使用基于规则系统来控制的黑板系统。

TRICERO。这个系统监视飞机活动。它把黑板体系结构扩展成分布式计算［Wil84］。针对部分问题设计了4个完整独立的专家系统运行在4台分离的机器上。

泛化。在1977~1984年间，推广面向应用的黑板系统产生了框架来方便建立黑板应用。但是，没有标准的方法来做这件事。

SUS。在［THG94］中描述了一个最近的项目“软件理解系统”，它特别对我们作为软件模式作者的观点感兴趣。SUS的目的是支持软件的理解和对可重用资源的搜索。在一次匹配过程中，系统将来自模式库中的模式与正在分析的系统进行比较。SUS递增地建立已分析过的接着可以评审的软件的一个“模式对应关系”。

10. 已解决的例子

下面我们给出了处理步骤的摘录，这些处理步骤是HEARSAY-Ⅱ理解短语“Are any by Feigenbaum and Feldman?”中用到的，正如［EHLR88］中所描述的。

⊖ 调度队列没有隐念地确定应该被删除的元素的队列，就像LIFO队列或FIFO队列所做的那样。相反，调度器通过重复地计算优先级来确定序列。因而，根据我们的术语，HEARSAY-II的“调度队列”是一个容器而不是一个队列。

89 为简要刻画例子中被激活的知识源：

- 在任何创建新建设的知识源活动之后，RPOL作为高优先级任务立刻运行。RPOL使用基于新假设的分级信息以及与新假设相关联的假设的分级信息，来计算新假设的整体分级。
- PREDICT分析一个短语并在该语言中预测那短语之前或之后（并紧挨短语）的所有词。
- VERIFY试图在预测一个词的短语上下文中验证预测词的存在性或拒绝某个预测词。如果该词被验证了，必须同时为它生成一个置信等级。这由知识源RPOL来完成。
- CONCAT完成一个短语的生成，该短语由验证的词及其预测短语构成。扩展短语包含一个置信等级，而该等级又是基于预测短语等级和被验证的词的等级。如果一个被验证的词早已和其他短语相关联，CONCAT试图以预测短语来分析那个短语。如果成功了，则创建一个短语假设来表示两个短语的合并。

为方便理解，我们已简化了原始描述，并且省略了知识源条件部分的详细执行。RPOL的执行也被省略了。VERIFY知识源的执行往往紧跟着PREDICT知识源的执行。所以这两个知识源组合一个步骤。

为帮助你理解以下处理步骤序列和图，这里是我们所使用符号的解释：

- 词或短语后面括号中的数字标明了它的可信度等级。
- “[”标明口述短语的开始，“]”标明它的结束。
- KS代表“知识源”。

目前黑板上最高等级的假设是这样的：

[ARE+(97), [ARE+REDDY(91), FEIGENBAUM+AND+FELDMAN+] (85),

和[ARE+ANY(86).

90 步骤17是需要考虑的第一个步骤：

```
Step 17: KS PREDICT&VERIFY
Stimulus: FEIGENBAUM+AND+FELDMAN+](85) (phrase).
Action: Predict eight preceding words;
reject one: DISCUSS;
find three already on the blackboard: CITE(70),
ABOUT(75), BY(80);
verify four: CITES(65), QUOTE(70), ED(75), NOT(75).
```

一个假设的等级并不只是调度器用来指定等待知识源激活的优先级的参数。特殊情况下，假设的长度也是很重要的。所以可信等级是85的短语FEIGENBAUM+AND+FELDMAN+]优于等级是91的短语[ARE+REDDY和等级是86的短语[ARE+ANY，因为它更长。

从步骤18到步骤24，揭示了FEIGENBAUM+AND+FELDMAN+](85)的其他词扩展。作为探查的结果，短语BY+FEIGENBAUM+AND+FELDMAN+](84)被认为是最可信的。

```
Step 18: KS CONCAT
Stimulus: BY(80) (word),
FEIGENBAUM+AND+FELDMAN+](85) (phrase).
Action: Create phrase:
BY+FEIGENBAUM+AND+FELDMAN+](84).

Step 19: KS CONCAT
```

```
Stimulus: ABOUT(75) (word),
FEIGENBAUM+AND+FELDMAN+](85) (phrase).
Action: Create phrase:
ABOUT+FEIGENBAUM+AND+FELDMAN+](83)

Step 20: KS PREDICT&VERIFY
Stimulus:
ABOUT+FEIGENBAUM+AND+FELDMAN+](83)(phrase).
Action: Predict one preceding word;
verify: WHAT(10).

Step 21: KS CONCAT
Stimulus: CITE(70)(word),
FEIGENBAUM+AND+FELDMAN+](85)(phrase).
Action: Create phrase:
CITE+FEIGENBAUM+AND+FELDMAN+](83) (phrase).
```

91

```
Step 22: KS PREDICT&VERIFY
Stimulus: CITE+FEIGENBAUM+AND+FELDMAN+](83)(phrase).
Action: Predict ten preceding words;
reject five: ABSTRACTS, ARE, BOOKS, PAPERS, REFERENCED;
find two already on the blackboard: ANY(65), THESE(25);
verify three: ARTICLE(25), WRITTEN(25), ARTICLES(10).
```

如果先于该短语的所有10个词的预测已被拒绝，则短语假设本身也会被拒绝。

```
Step 24: KS CONCAT
Stimulus: NOT(75) (word),
FEIGENBAUM+AND+FELDMAN+](85).
Action: Create phrase:
NOT+FEIGENBAUM+AND+FELDMAN+](83).

Step 25 KS CONCAT
Stimulus: ANY(65) (word),
BY+FEIGENBAUM+AND+FELDMAN+](84) (phrase).
Action: Create phrase:
ANY+BY+FEIGENBAUM+AND+FELDMAN+](82).
[ARE+ANY+BY+FEIGENBAUM+AND+FELDMAN+](85)
is also created from [ARE+ANY(86) and
BY+FEIGENBAUM+AND+FELDMAN+](84).
```

短语有时是一个完整的句子，所以它是用于口语输入解释的候选。

在下面的图中，如果在一单个处理步骤中一个假设由另一个假设导出，则这两个假设之间有条弧。在弧上标记处理步骤的顺序号。虚线弧指出步骤17之前就已经在黑板上的假设。

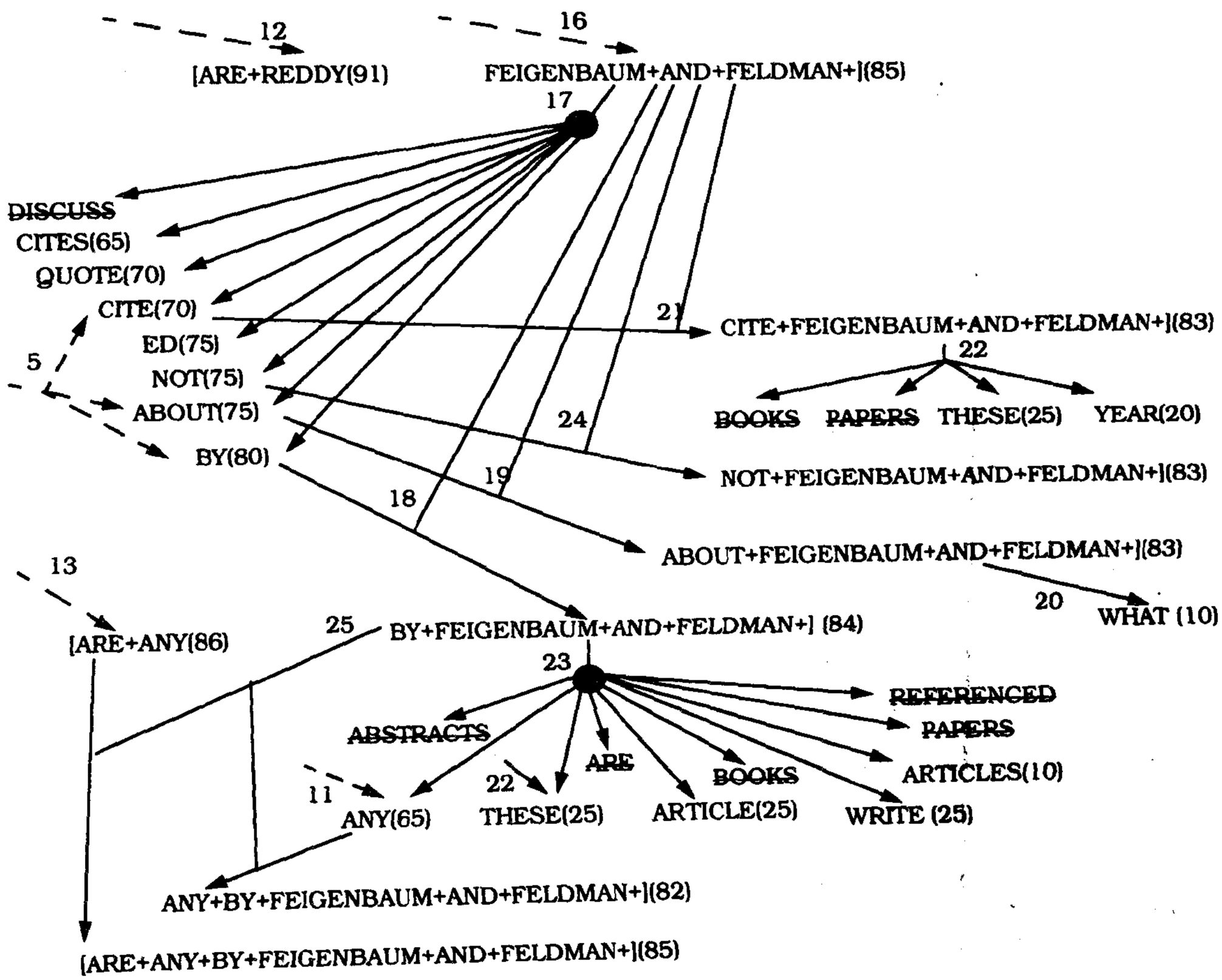
11. 效果

用于问题分解和知识应用的黑板方法有助于解决“问题”小节中所列的大部分强制条件：

试验。在领域中如果没有独立方法存在，而且对解空间的完全搜索也是不可行的，则黑板模式用不同的可能算法来进行试验，并也允许试用不同的控制启发方法。

92

对可更改性和可维护性的支持。由于单个知识源，控制算法和中心数据结构被严格分离，



所以黑板模式支持可更改性和可维护性。然而，所有的模块可以通过黑板来通信。

可重用的知识源。知识源是某类任务的独立专家。黑板体系结构有助于使它们可重用。重用的先决条件是知识源和所基于的黑板系统理解相同的协议和数据，或者在这方面相当接近而不排斥协议或数据的自适应程序。

93 支持容错性和健壮性。在黑板体系结构中，所有的结果都只是假设。只有那些被数据和其他假设强烈支持的才能生存。这提供了对噪声数据和不确定结论的容忍。

黑板模式有一些不足：

测试困难。由于黑板系统的计算没有依据一个确定的算法，所以其结果常常不可再现。此外，错误假设也是求解过程的部分。

不能保证有好的求解方案。一个黑板系统往往只能正确解决所给任务的某一百分比。

难以建立一个好的控制策略。控制策略不能以一种直接方式设计，而需要一种试验的方法。

低效。黑板系统在拒绝错误假设中要承受多余的计算开销。但是，如果没有确定的算法存在，则与根本不存在系统相比，低效总比没有强。

昂贵的开发工作。绝大多数黑板系统要花几年时间来进化。我们把这归因于病态结构问题

领域和定义词汇、控制策略和知识源时的粗放的试错编程方法。

缺少对并行机制的支持。黑板体系结构不能避免采用了知识源潜在并行机制的控制策略的使用。但是它不提供它们的并行执行。对黑板上的中心数据的并发访问也必须是同步的。

总之，黑板体系结构允许知识的解释使用。它评估可选动作，选择对当前情形最优的动作，然后应用最有希望的知识源。只要对该问题没有充分的详尽的算法可用，这种深思熟虑的花费就是值得的。一旦有了这样的算法，通常它提供更多的性能和效果。黑板体系结构因此使自己成为最适合不成熟领域，在这样的领域中试验是有帮助的。通过研究并获得经验之后，可以进化到更好的算法使你能够使用一个更有效率的体系结构。 94

这出现在语音识别领域。例如，在HARPY系统（HEARSAY-Ⅱ的后继系统）中，绝大多数知识被预先编译成统一的结构来表示所有可能的口语短语［EHLR88］。所有中间层次的替换，如片断到单音，单音到词，词到短语等被编译成单个的非常大的有限的马尔可夫（Markov）网络。于是解释器用这个结构比较口语短语的片断来寻找与分割的语音信号最近似的网络路径。所使用的搜索技术，称作定向（beam）搜索，与词格结合，是动态编程的启发式形式。听觉的和语言学的知识不再通过黑板结合，而是通过“极大似然”计算。一个窗口略过输入并不断地将新结果加到输出。这允许语音识别以实时方式使用。［Mar95］给出了对较简单的语音识别产品的最近更新。有关语音识别的更详细内容，参见［HAJ90］、［Rab86］和［Rab89］。

致谢

我们的黑板模式大部分是基于从HEARSAY-Ⅱ语音识别系统中抽象出来的特征。在AI文化中找到术语“黑板”的最初引用，是在Newell和Simon［NS72］有关西洋棋、象棋和定理证明程序等组织问题的文本中。对黑板体系的最综合的描述和讨论在［EM88］和［Cra95］中。

感谢西门子公司语音处理组的Harald Höge，我们用他的工作解释了该领域中的最新进展。 95

2.3 分布式系统

最近硬件技术发展有两个主要趋势：

- 带有多CPU的计算机系统逐渐进入小型办公场所，尤其是运行诸如IBM OS／2 Warp、微软Windows NT或UNIX等操作系统的多处理系统。
- 用局域网连接成百上千个不同种类的计算机已变得很平常。

现在，即使是小公司也在使用分布式系统。但是是什么优势使分布式系统受到如此众多的欢迎呢？Tanenbaum[Tan92]对此进行了如下解释：

经济性。将PC与工作站连接起来的计算机网络，其价格／性能比要高于大型机。

性能与可扩展性。根据SUN 微系统公司的理念——“网络即计算机”，分布式应用程序能够利用网络上可获得的资源。通过使用联合起来的几个网络节点的计算能力，可以获得性能方面的极大提升。另外，至少在理论上，多处理器和网络是易于扩展的。

固有分布性。有些应用程序是固有分布式的，如遵循客户机-服务器模型的数据库应用程序。

可靠性。大多数情况下，网络上的一台机器或多处理器系统中的一个CPU的崩溃不会影响到系统的其余部分。中心节点（文件服务器）是明显的例外，但可以采用备份系统来保护。

然而，分布式系统有一个显著的缺点[Tan92]："分布式系统与集中式系统相比需要完全不同的软件"。这就是为什么诸如对象管理组（OMG）这样的合作团体和诸如微软这样的公司都开发了自己的分布式计算技术的主要技术原因。

97 我们在该类别中引入与分布式系统相关的三种模式：

- 管道和过滤器模式为处理数据流的系统提供了一种结构。每个处理步骤封装在一个过滤器组件中。数据通过相邻过滤器之间的管道传输。重组过滤器可以建立相关系统族。

这种模式与用于分布相比，更经常用于构建一个应用程序的功能内核，因此我们在一个不同类别中描述它——见2.2节"从混沌到结构"。

- 微核（*Microkernel*）模式应用于必须能够适应变更系统需求的软件系统。这种模式把最小功能内核同扩展功能和特定客户部分分离开来。微核也可作为插入到这些扩展中和协调其协作的套接字。

微核系统采用客户机-服务器体系结构，在该结构下，客户机和服务器在微核组件之上运行。但是，这类系统主要优点是在设计中考虑自适应和更变。我们因此将该模式的描述放在另一个类别——见2.5节"适应性系统"。

诸如微软OLE（对象连接和嵌入Object Linking and Embedding）[Bro94]和OMG的CORBA (公用对象请求代理体系结构，Common Object Request Broker Architecture) [OMG92]这样的平台共用了一个相同的软件体系结构，从该结构中我们抽象出代理者模式：

- 代理者模式可以用于构建带有隔离组件的分布式软件系统，该组件通过远程服务调用进行交互。代理者组件负责协调通信，诸如转发请求以及传送结果和异常。

有三组开发者可以受益于使用代理者模式，这三组开发者是：

- 那些正在使用现有代理者系统并对了解该系统结构有兴趣的人。
- 那些想构建代理者系统的"剪裁"版的人，该版本不带有完整OLE或CORBA的所有铃和哨。
- 那些计划实现一个完美的代理者系统并因此需要深入描述代理者体系结构的人。

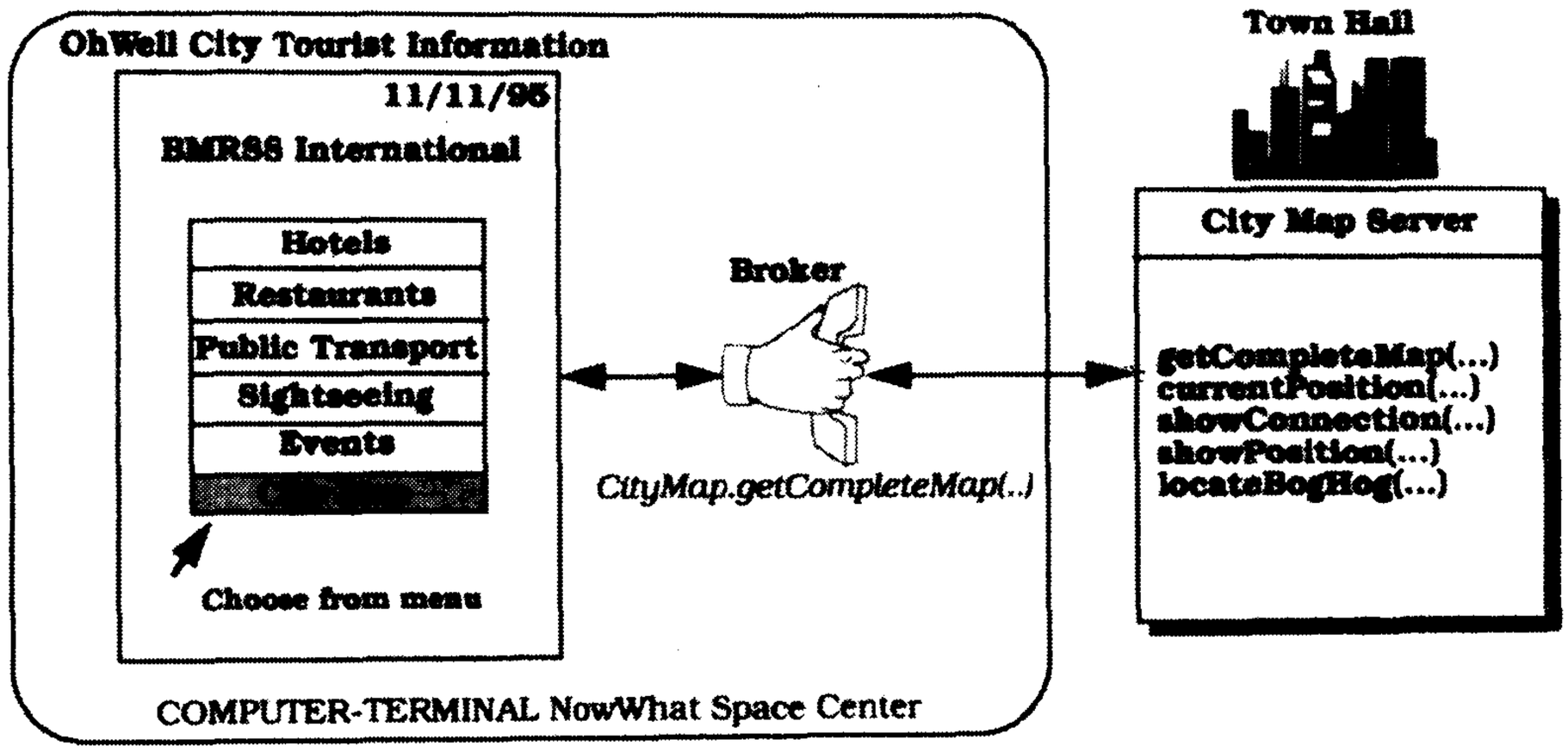
98

2.3.1 代理者

代理者（*Broker*）结构模式可以用于构建带有隔离组件的分布式软件系统，该软件通过远程服务调用进行交互。代理者组件负责协调通信，诸如转发请求，以及传送结果和异常。

1. 例子

假设我们正在开发一个城市信息系统（CIS），它计划运行于广域网上。网络上有些计算机负责一项或多项支持有关事件、餐厅、宾馆、历史纪念馆或公共运输信息的服务。计算机终端与网络相连接。城市的游客可以在计算机终端上通过使用万维网（WWW）浏览器检索他们感兴趣的信息。这种前端软件支持从适当的服务器以在线方式检索信息并显示在屏幕上。数据分布在网络上，并不是都在终端上维护。



我们希望系统能够不断地改变并生长，因此单个服务之间应隔离。另外，终端软件应该能在不必知道服务器地址的情况下获得服务。这就使我们能移动、复制或迁移服务。一种解决办法是安装一个独立的网络，它连接所有终端和服务器，形成一个内联网（intranet）系统。然而这样的方法有几个缺点：并不是所有信息提供者都想连接到一个封闭的内联网，而且更重要的是，有效服务应该在世界各地可访问。因此，我们决定使用因特网作为实现CIS系统的一个更好的解决方案。 99

2. 语境

你的环境是带有独立协作组件的分布式的并可能是异构的系统。

3. 问题

构建一个复杂的软件系统，它是独立的和互操作的组件集合而不是一个整体的应用程序，从而有较大的灵活性、可维护性和可变更性。通过将功能分割成独立的组件，系统潜在地变得可分布和可扩展。

然而，当分布式组件相互通信时，需要一些过程间通信手段。如果组件自身处理通信，最终系统面临几个相关性和局限性问题。例如，系统变得依赖于所使用的通信机制，客户机要知道服务器的位置，在很多情况下该解决方案仅限于使用一种编程语言。

用来添加、移动、交换、激活和定位组件的服务也是必需的。使用这些服务器的应用程序不应该依赖于特定系统的细节以确保在异构网络中的可移植性和互操作性。

从开发者的角度来讲，开发集中式系统的软件和开发分布式系统的软件应该没有本质差别。使用一对象的应用程序只看该对象所提供的接口。不需要知道对象的实现细节或其物理位置。

使用代理者体系结构来权衡以下强制条件：

- 组件应该能够访问其他组件通过远程、地点透明的服务调用提供的服务。
- 需要在运行期间交换、添加或移动组件。
- 该体系结构应该向用户隐藏特定系统和特定实现的细节。

100

4. 解决方案

引入代理者组件可较好地做到客户机和服务器的隔离。服务器向代理者注册自己并使其服务通过方法接口能为客户机所使用。客户机通过代理者发送请求访问服务器功能。代理者的任务包括定位合适的服务器，将请求转发到服务器并向客户机回送结果和异常。

通过使用代理者模式，应用程序能够简单地通过向合适的对象发出消息调用访问分布式服务，而不是把重点放在低级进程间通信。另外，代理者体系结构很灵活，在这种模式下允许对对象动态改变、添加、删除和重定位。

代理者模式降低了开发分布式应用程序的复杂性，因为分布性对开发者来说变得透明了。通过引入对象模型达到这个目标，在这个模型下分布式服务被封装对象中。代理者系统提供了两种核心技术集成的途径：分布技术和对象技术。它也将对象模型从单一应用程序扩展到由运行在异构机器上的并可以用不同编程语言来编写的独立组件组成的分布式应用程序。

5. 结构

代理者体系结构模式由6种参与组件构成：客户机、服务器、代理者、桥接、客户机端代理和服务器端代理。

服务器[⊖]实现了通过接口展示功能的对象，这里接口由操作和属性组成。这些接口或者通过接口定义语言（IDL）或者通过二元标准来获得。“实现”小节对这些方法进行了对比。接口一般聚合了语义上相关的功能。有两种类型的服务器：

101

- 为许多应用领域提供公共服务的服务器。
- 为单一应用域或任务实现特定功能的服务器。

➡在CIS例子中，服务器由提供访问HTML（超文本标记语言）网页功能的WWW服务器组成。WWW服务器用httpd (超文本转换协议)过程实现，该过程在特定端口等待输入请求。当一个请求到达服务器时，所请求的文档和所有附带的数据使用数据流被发送到客户机。HTML网页包含在网络主机上可以远程执行操作的文档和CGI(公共网关接口)脚本，网络主机是一台远程机器，客户机从这台远程主机上接受HTML网页。CGI脚本用来让用户填写表格和提交查询，例如查找宾馆空房间的搜索请求。为了在客户机WWW浏览器上显示动画，将Java ‘applets’ 集成到HTML文档之中。例如，Java applets之一在城市地图上生动显示了一个地方和另一个地方之间的路线。Java applets运行在虚拟机的顶层，虚拟机是WWW浏览器的一部分。CGI脚本和Java applets互不相同：CGI脚本在服务器上执行，而Java applets被发送到WWW浏览器从而在客户机上执行。❑

客户机程序是访问至少一台服务器的服务的应用程序。为了调用远程服务，客户机向代理者转发请求。在一个操作已经执行后，他们将收到来自代理者的应答或异常。

客户机与服务器之间的互操作基于一种动态模型，它意味着服务器也可以做客户机。这种动态互操作模型与传统的客户机-服务器计算的概念不同，客户机和服务器的作用并不是静态定义好的。虽然其他的实现也是可能的，但是从实现的观点来看，你可以把客户机看作应用程序

⊖ 在这种模式描述中，服务器负责实现服务。在一个面向对象方法中，每个服务通过一个或更多对象实现。我们使用“服务器”这个术语是为了强调用面向对象的观点来看，一个服务器对其他组件好像是一个对象。

而把服务器看作库。特别指出的是客户机不必知道他们访问的服务器的位置。这点很重要，因为这将允许增加新的服务，也可以移动现有的服务到其他地方，即使系统处在运行期间。 102

➡在代理者模式的语境中，客户机是可以利用的WWW浏览器。他们并不直接连接到网络上。相反，他们依赖于因特网提供者，他们提供接入因特网的网关，如服务商CompuServe。WWW浏览器用调制解调器或租借线路连接到这些工作站。连接后，他们就可以检索来自httpd服务器的数据流，解释这些数据并开始执行诸如在显示屏上显示文档或执行Java applets的动作。 ❑

类 客户机	协作者 • 客户机端代理 • 代理者
责任 • 实现用户功能 • 通过客户机端代理向服务器发送请求	

类 服务器	协作者 • 服务器端代理 • 代理者
责任 • 实现服务 • 用本地代理者注册自身 • 通过服务器端代理将应答和异常发回客户机	

代理者是负责从客户机到服务器的请求传送的信使，也是返回到客户机的应答和异常传送的信使。基于接受者的惟一系统标识符，代理者必须有某种手段对请求的接受者定位。代理者向客户机和服务器提供包括用来注册服务器和调用服务器方法的操作的APIs（应用程序编程接口）。

当请求发到由本地代理者㊀维护的一个服务器时，代理者将直接把请求传送到服务器。如果服务器当前未激活，代理者将其激活。所有来自服务执行的应答和异常由代理者转发到发出请求的客户机。如果指定的服务器由另一个代理者代理，则本地代理者将寻找一条到远程代理者的路径并使用该路径转发请求。这就需要代理者要能互操作。 103

根据整个系统的需求，诸如名字服务㊁或列集支持㊂这样的附加服务将可以集成到代理者中来。

➡CIS例子中的代理者是因特网网关和因特网基础结构本身的结合。客户机和服务器之间的任何信息交换必须经过代理者传递。客户机通过使用称为URLs（通用资源定位器）的惟一外部数据标识符来指定所需的信息。通过使用这些标识符，代理者能够定位所需的服务并将请求循线递送到适当的服务器机器上。如果添加一个新服务器，它必须由代理者注册。客户机和服务器使用因特网提供者的网关作为到代理者的接口。 ❑

㊀ 在该模式的描述中，我们区别了本地代理者和远程代理者。本地代理者运行在当前正在考虑的机器上，远程代理者运行在远程网络节点上。

㊁ 名字服务提供名字和对象之间的关联。为了解析一个名字，名字服务确定哪一个服务器与给定的名字相关联。在代理者系统的语境中，名字仅在与名字空间相关时才有意义。

㊂ 列集是数据语义不变的转换，转换与机器无关，如ASN.1（抽象语义符号）或ONC XDR(外部数据表示)。散集则是进行相反的转换。

类 代理者	协作者 • 客户机 • 服务器 • 客户机端代理 • 服务器端代理 • 桥接
责任 • 注册(取消注册)服务器 • 提供APIs • 传递消息 • 错误恢复 • 通过桥接而与其他代理者互操作 • 定位服务器	

客户机端代理代表了客户机和代理者之间的一个层。该附加层提供了透明性，这样对客户机来说，一个远程对象看上去好像在本地。更详细地说，代理者允许对客户机隐藏具体的实现细节，诸如：

104

- 用来在客户机和代理者之间传送消息的进程间通信机制。
- 存储体的创建与删除。
- 参数和结果的列集。

在许多情况下，客户机端代理将特定的作为代理者体系结构模式一部分的对象模型翻译成用来实现客户机的编程语言的对象模型。

服务器端代理通常与客户机端代理相似。区别在于它们是负责接受请求、解开来到的消息、散集参数、调用适当的服务等。另外，还用作在向客户机发送它们之前对结果和异常进行列集。

类 客户端代理	协作者 • 客户机 • 代理者
责任 • 封装特定系统的功能 • 客户机与代理者之间的协调	

类 服务器端代理	协作者 • 服务器 • 代理者
责任 • 在服务器内协调服务 • 封装特定系统的功能 • 服务器与代理者之间的协调	

当结果或异常从服务器返回时，客户机端代理从代理者接受到发送来的消息，散集数据后并将它转发给客户机。

➡在CIS的例子中，诸如Netscape WWW浏览器和httpd服务器，提供了用来与因特网提供者的网关通信的内在能力，这样，在这种情况下我们就不必为代理者担心。 ❑

网桥(Bridges)[⊖]是用来隐藏两个代理者互操作时的实现细节的可选组件。假设代理者系统运

⊖ 我们称这些组件为网桥，遵循了CORBA2规格说明中的OMG术语。

行在异构的网络上。如果在网络上发送请求，不同的代理者必须在所使用的不同的网络和操作系统的独立通信。网桥建立了一个将所有特定系统细节封装起来的层。 105

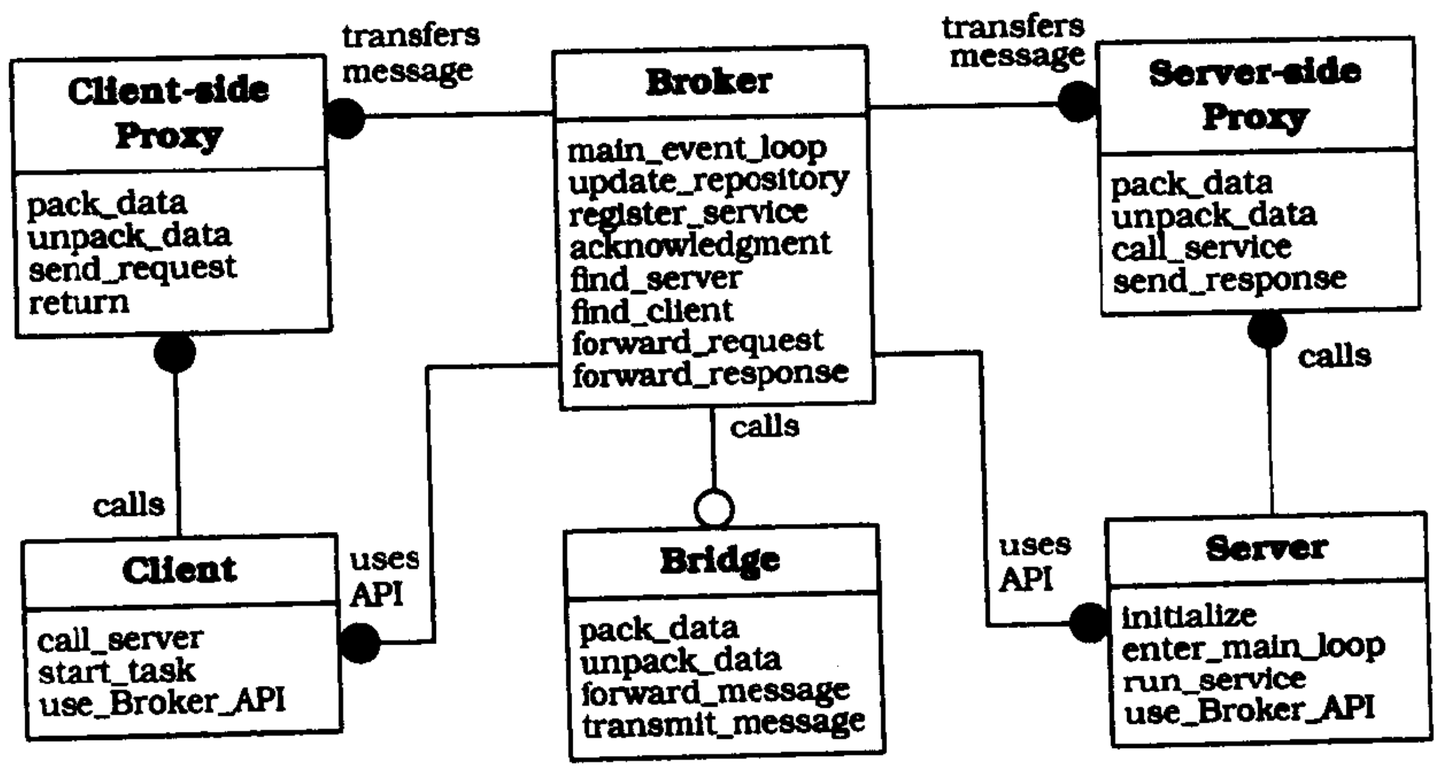
➡CIS例子中并不需要网桥，因为所有的httpd服务器和WWW浏览器实现了用于远程数据交换所必需的协议，例如http（超文本传输协议）或ftp(文件传输协议)。 ❑

类 网桥	协作者 • 代理者 • 网桥
责任 • 封装特定网络的功能 • 协调本地代理者与远程代理者的网桥	

有两种不同种类的代理者系统：采用直接通信的系统和采用间接通信的系统。为了达到较好的性能，一些代理者的实现仅建立了客户机与服务器之间通信链，而剩下的通信由参与的组件之间直接进行——消息、异常和应答在客户机端代理和服务器端代理之间传送而不使用代理者作为中间层。这种直接通信的方法要求服务器和客户机使用并理解相同的协议。在该模式描述中，我们关注间接代理者变体，在这种模式下，所的消息都通过代理者传递。这种客户机-分配器-服务器模式描述了代理者模式的直接变体的重要方面。

➡CIS的例子实现了间接通信变体，因为浏览器和服务器只能通过使用网间网关协作。但是，GIS中只有一个地方使用了直接通信变体——从网络上下载的Java applets可以直接连接到WWW服务器，它们使用一个套接字从WWW服务器得到。 ❑ 106

下图给出了代理者模式中涉及的对象：



6. 动态特性

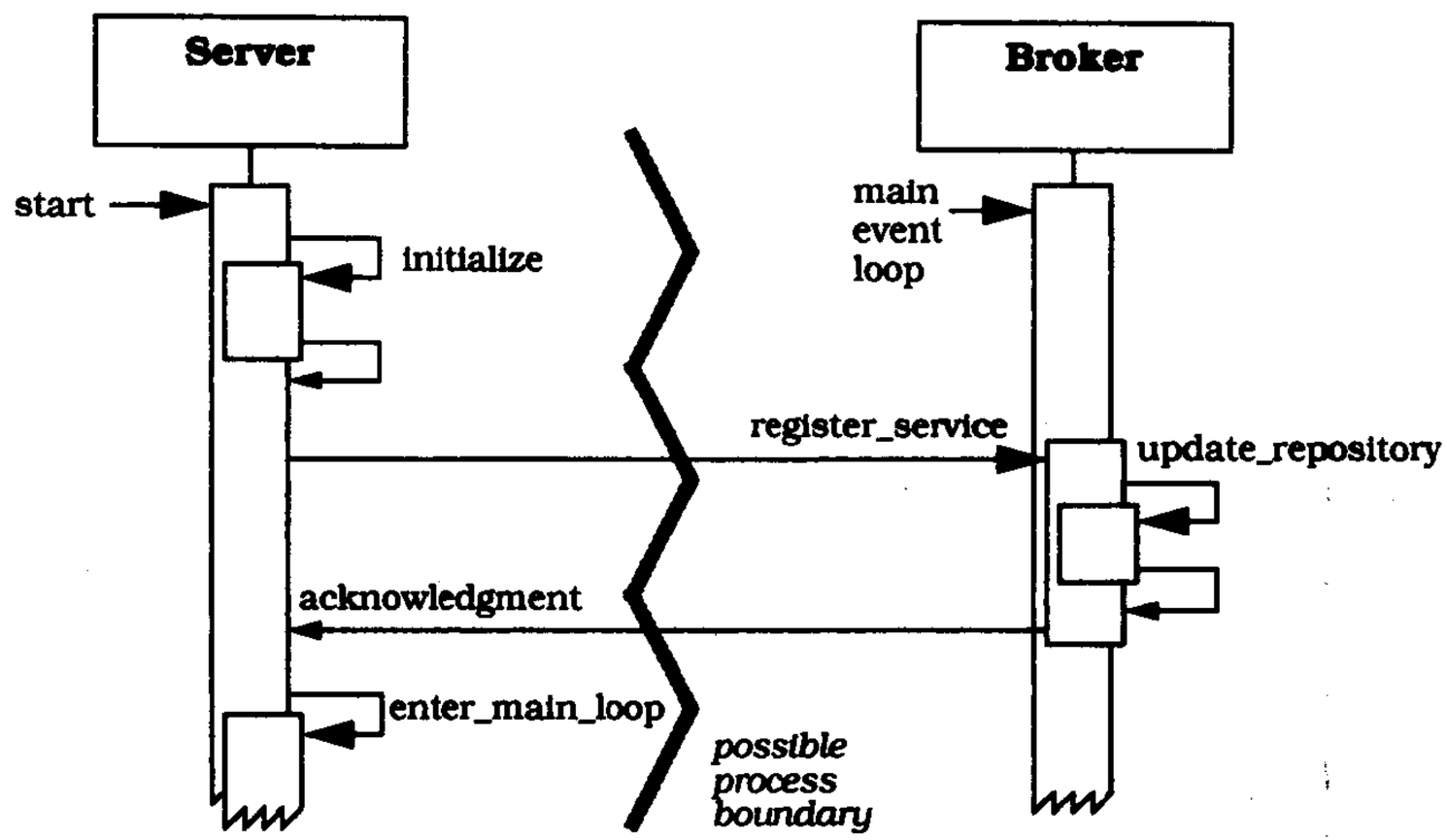
本节关注代理者系统的操作中的最相关的场景。

场景I 说明了当服务器向本地代理者组件注册自己时的行为：

• 在系统的初始化阶段代理者被启动。代理者进入其事件循环并等待消息的到来。

- 用户或其他实体启动了一个服务器应用程序。首先，服务器执行其初始化代码。初始化完成后，服务器向代理者注册自己。
- 代理者收到来自服务器的新来的注册请求。从消息中提取出所有必需信息并将其存储到一个或多个仓库中。这些仓库用来定位并激活服务器。发回一个确认。
- 从代理者收到确认后，服务器进入其主循环等待新来的客户机请求。

107



场景II 说明了当一个客户机向本地服务器发送请求时的行为。在这个场景中，我们描述了一个同步调用，在这个调用中，客户机处于阻塞状态直到得到了服务器的回应。代理者也可能支持异步调用，允许客户机不必等待回应而进一步执行任务。

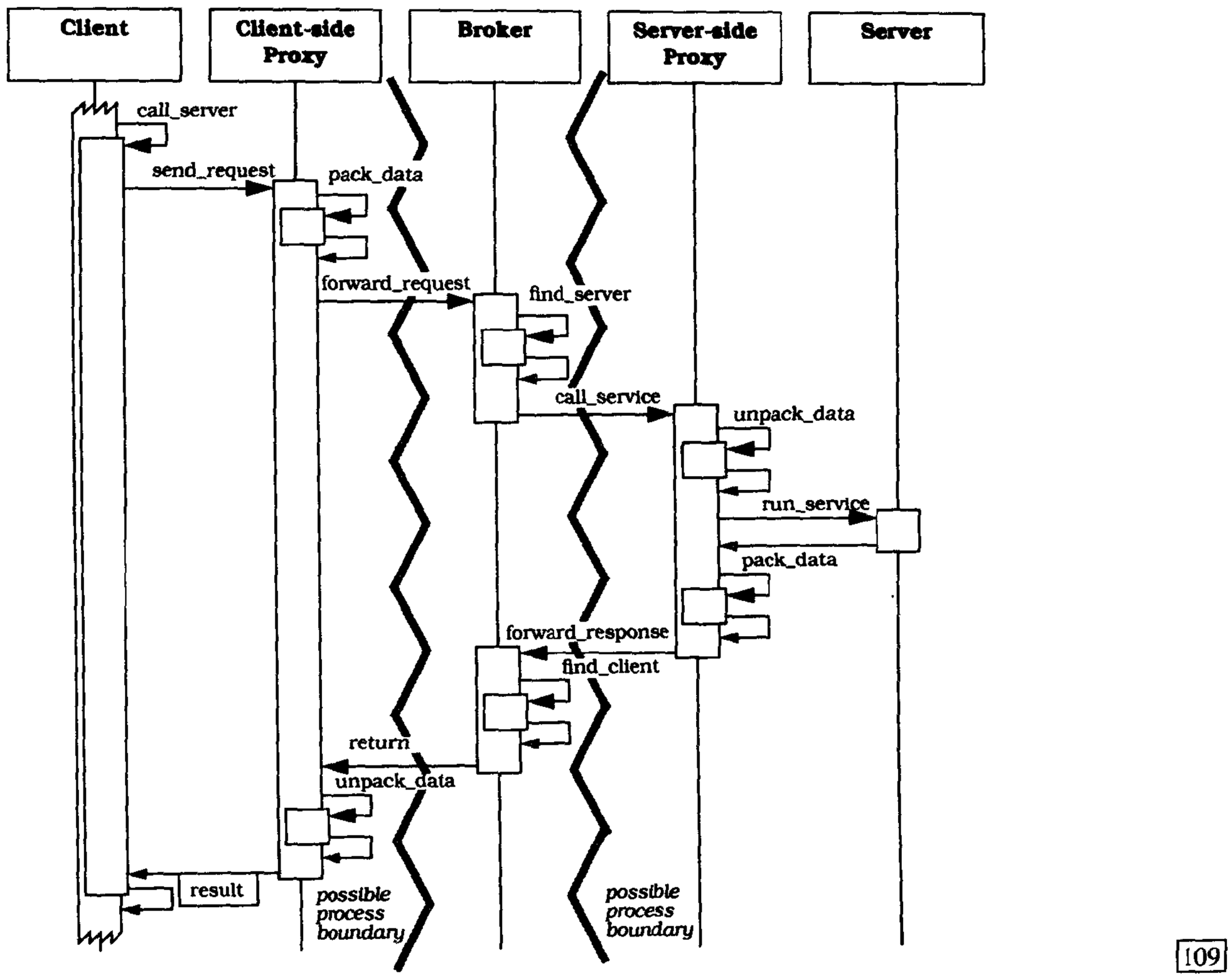
- 启动客户机应用程序。在程序执行期间，客户机调用远程服务器对象的方法。
- 客户机端代理将所有参数和其他相关信息打包成一条消息并将该消息发送到本地代理者。
- 代理者在其仓库中查找到所要求的服务器的地址。由于服务器是本地可用的，代理者将消息转发到相应的服务器端代理。远程情况下，见下面的场景。
- 服务器端代理打开所有参数和其他信息，如期望调用的方法。服务器端代理调用适当的服务器。

108

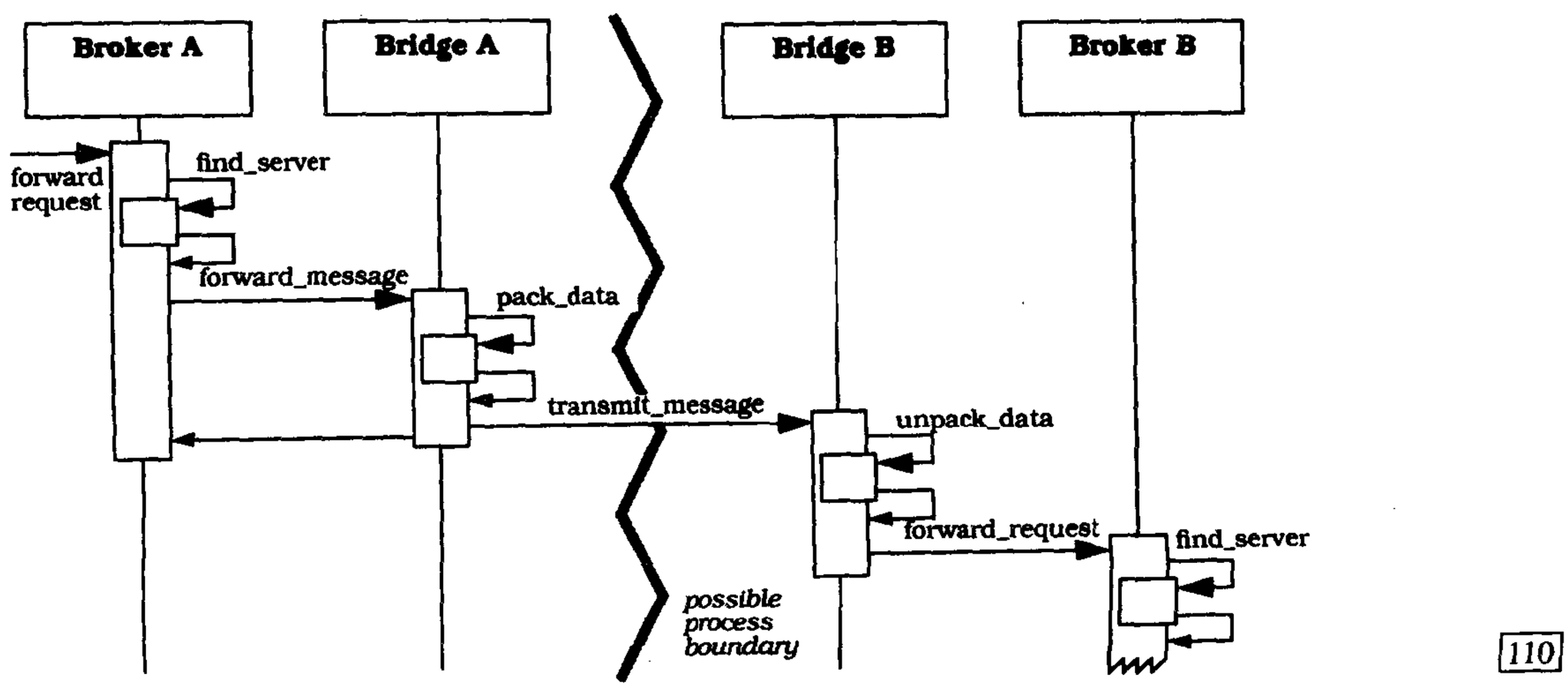
- 服务器执行完毕后，服务器将结果返回到服务器端代理，它将其打包成带有其他相关信息的消息并传递给代理者。
- 代理者将回应转发到客户机端代理。
- 客户机端代理收到回应，打开结果并返回到客户机应用程序。客户机进程继续其计算。

场景III 说明通过桥接组件不同代理者进行交互：

- 代理者A收到新来的请求。它通过在仓库中查找定位负责执行特定服务的服务器。由于对应的服务器在另一个网络节点处可用的，因此，该代理者发送请求到远程代理者。
- 消息从代理者A发送到网桥A。这个组件负责将该消息从由代理者A定义的协议转换成网络特定的但可以被参与的两个网桥所理解的公共协议。消息转换后，网桥A将消息发送到网桥B。



- 网桥B将新来的请求从网络特定格式映射到代理者B特定的格式。
- 当请求到达时，代理者B执行所有必需的行动，正如该场景中的第一步中所描述。



7. 实现

为了实现这个模式，执行以下步骤：

(1) 定义一个对象模型，或使用一个现有的模型。所选择的对象模型对正在开发的系统的其他部分有重要的影响。每个对象模型必须指定实体，诸如对象名、对象、请求、值、异常、支持的类型、类型扩展、接口和操作。第一步中你应该只考虑语义问题。如果对象模型必须是可扩展的，则为下一步系统增强作准备。例如，指定一个基本对象模型并说明如何使用扩展将其系统地细化。关于这个主题更多的信息参见文献［OMG92］。

基础计算模型的描述是设计对象模型的关键。你需要描述服务器对象状态的定义、方法定义、怎样为执行选择方法和服务器对象如何产生和销毁。服务器对象的状态及其方法实现不应该直接被客户机访问。客户机仅能通过向本地代理者传送请求间接改变或读取服务器的状态。用这样的接口与服务器实现的隔离，所谓“远程”接口就变成了可能——客户机使用客户机端代理作为服务器接口，该接口完全与服务器实现隔离，从而与服务器接口的具体实现相隔离。

(2) 确定系统应该提供哪种组件互操作性。可以通过指定一个二元标准或引入高级接口定义语言（IDL）来设计互操作性。IDL文件包括服务器提供给它的客户机的接口的文本描述。二元方法需要得到编程语言的支持。例如，在微软对象链接与嵌入 (OLE)［Bro94］中，二元方法表是可用的。这些表由指向方法实现的指针组成，并能使客户机间接使用指针来调用方法。访问OLE对象仅被了解这些表的物理结构的编译器或解释器支持。

与二元方法相反，IDL方法更加灵活，表现在IDL映射可以用任何编程语言实现。有时这两种方法是结合使用的，如在IBM的系统对象模型(SOM)[Cam94]中所示。

111

IDL编译器使用IDL文件作为输入并产生编程语言代码或二元码。生成的代码的一部分被服务器用来与其本地代理者通信，另一部分被客户机用来与其本地代理者通信。代理者将使用IDL规则说明来维护关于现存服务器实现的类型信息。

一旦互操作性作为二元标准提供，对象模型的每个语义概念都必须与一个二元表示相关联。然而，如果你为互操作性提供一个接口定义语言，那么可以将语义概念映射到编程语言表示。例如，对象句柄可以用C++的指针来表示而数据类型被映射到适当的C++类型。

还有一个问题——代理者系统应该何时以接口定义语言展示接口，什么时候用二元标准展示接口？第一种方法的基本原理是为代理者的实现获得更多的灵活性——代理者体系结构的每种实现均可定义它自己的用于代理者和其他组件交互的协议。提供到本地代理者协议的映射是IDL的任务。遵循二元标准时，你需要定义二元表示法，比如，调用远程服务的方法表。这样常常更加高效，但当客户机和服务器通信时要求所有代理者实现相同种类的协议。

(3) 指定代理者组件提供客户机与服务器合作的API。在客户机端，必须提供构建请求，将请求发送到代理者并接受回应等功能。确定客户机是否仅能静态调用服务器操作，允许客户机在编译期间绑定调用。如果也想允许服务器动态调用[⊖]，这将对API的大小或数量有直接的影响。例如，你需要某种询问代理者关于现有服务器对象情况的方法。你可以借用元层次元素来实现，如映象模式中所描述的。

112

⊖ 动态请求是运行期间使用API函数以及类型信息动态的构造方法调用（call）。相反，静态请求是在源代码中的实际编码。

必须向客户机提供操作，这样将使它们能够在运行期间构建请求。服务器实现使用了API函数主要是为了向代理者注册。代理者使用仓库来维护信息。这些仓库可以作为外部文件提供，这样服务器就可以在系统启动前注册自己。另一种方法是把仓库作为代理者组件的内部组成来实现。在此，代理者必须提供允许服务器在运行期间注册的API。由于代理者要在请求到达时确认这些服务器，因此一个恰当的确认机制是必要的。换句话说，代理者组件负责将服务器对象标识符与服务器对象的实现关联起来。因此，代理者的服务器端API必须能够产生系统惟一的标识符。

如果客户机、服务器和代理者作为不同进程来运行，API函数需要基于一个客户机、服务器和本地代理者之间的进程间通信的有效机制。

(4) 用代理对象来对客户机和服务器隐藏实现的细节。在客户机端，本地代理对象代表了客户机调用的远程服务器对象。在服务器端，代理用来实现客户机的作用。代理对象有以下责任：

- 客户机端代理将进程调用打包成消息并将这些消息转发到本地代理者组件。另外，它们从本地代理者接受回应和异常并将它们发送到调用客户机。必须指定用于在代理者和代理者间通信的内部消息协议来支持这一点。
- 服务器端代理接受来自本地代理者的请求并调用相应服务器的接口实现中的方法。它们依据内部消息协议在打包后将服务器回应和异常转发到本地代理者。

特别指出的是，代理常常是相应客户机或服务器过程的一部分。 113

代理通过使用其自身的过程间通信机制与代理者组件通信来隐藏实现细节。它们也可实现参数和结果的列集和散集到／从一个系统无关的格式。

如果采用IDL方法解决互操作性，那么代理对象是自动可用的，因为可以由IDL编译器产生。如果采用二进制方法，代理对象的创建和删除可以动态发生。

(5) 与第3步和第4步并行地设计代理者组件。这一步我们描述了如何开发作为客户机到服务器之间来回传送的消息的信使的代理者组件。为了提高整个系统的性能，某些实现并不通过代理者传递消息。在这些系统中，大部分工作是由代理完成，而代理者仍然负责建立客户机与服务器之间的初始通信链。客户机和服务器之间的直接通信只在两者都使用相同协议时才有可能。我们把这种系统称为直接通信代理者系统（见“变体”小节）。

在设计和实现期间，系统性重复如下步骤：

(5.1) 指定与客户机端代理和服务器端代理交互详细的在线协议。做出请求、回应和异常到你的内部消息协议间的映射计划。在线协议中，内部消息协议处理较高级结构（如参数值、方法名和返回值）到相应的由基本的过程间通信机制指定的结构间的映射。

(5.2) 本地代理者必须对网络中参加进来的每台机器都是可用的。如果请求、回应或异常在网络节点之间传送，相应的本地代理者必须使用一个在线协议互相通信。使用网桥来对代理者隐藏诸如网络协议和操作系统特性的细节。代理者也必须保持一个仓库来定位远程代理者或网关，以向其转发消息。你可以将寻找远程代理者的路由信息编码成服务器或客户机标识符的一部分。广播通信是另一种（可能效率不高）定位服务器或客户机所在网络节点的方法。 114

(5.3) 当客户机请求服务器的方法时，代理者系统负责将所有结果和异常返回到开始的客户机。换句话说，系统必须记住哪一个客户机发出了请求。在直接通信变体（见“变体”小节）

中，没有必要记住一个调用的请求者，因为客户机和服务器是用一个通信通道直接连接起来的。在间接代理者系统中，你可以通过记住请求发出者的不同方式来选择。例如，你可以指定客户机的地址为请求或消息的一个附加的、不可见的参数。

(5.4) 如果代理（见第4步）没有提供对参数和结果的列集和散集（编组和复组）的机制，你必须将该功能包括在代理者组件中。

(5.5) 如果你的系统支持客户机和服务器之间的异步通信，你需要在代理者（broker）或代理(proxy)中提供消息缓冲区作为临时的消息存储器。

(5.6) 代理者中包括一个目录服务用来将本地服务器标识符和相应的服务器物理地址关联起来。例如，如果基础的过程间通信协议是基于TCP／IP的，你可以使用因特网端口数作为物理的服务器地址。

(5.7) 当你的体系结构要求系统惟一的标识符在服务器注册期间动态产生时，代理者必须提供名字服务用来实例化这些名字。

(5.8) 如果你的系统支持动态方法调用（见第3步），代理者就需要某种维护有关现存服务器的类型信息的手段。客户尽可以使用代理者APIs来动态建立一个请求访问这个信息。你可以通过实例化映象模式来实现这种类型信息。这样，元对象就维护了可以通过元对象协议访问的类型信息。

115

(5.9) 考虑出现失效的情况。在分布式系统中，两种层次的错误可能发生：

- 诸如服务器这样的组件可能运行到出错条件。这与你在执行通常非分布式应用程序时所遇到的错误类型是一样的。
- 两个相互独立的进程之间的通信可能失败。这种情况就更加复杂了，因为通信组件是异步运行的。

当与客户机通信、其他代理者或服务器失败时，规划代理者的行为。例如，有些代理者重复发送请求或回应多次直到通信成功。如果你使用至多一次语义（*at-most-once semantic*）[⊖]，你必须保证一个请求仅执行一次，即使再次被发送。千万不要忘记客户机试图访问不存在或没有权限访问的服务器的情况。出错处理是实现分布式系统的重要主题。如果你忘记用系统的方法处理错误，那么客户机应用程序和服务器的测试和调试将成为极其乏味的工作。

(6) 开发IDL编译器。一旦你通过提供一个接口定义语言实现互操作性，你需要为你所支持的每种编程语言建立一个IDL编译器。IDL编译器将服务器接口定义转化成编程语言代码。使用许多编程语言时，最好将编译器开发成一个框架，以允许开发者添加他自己的代码生成器。

8. 已解决的例子

例子CIS系统提供了不同种类的服务。例如，一个隔离的服务器工作站提供了所有与公共传输相关的信息。另一个服务器负责收集并公布宾馆空房信息。游客也许会对来自几家宾馆的检索信息感兴趣，所以我们决定在单一的工作站上提供这个数据。每家宾馆都可以与这个工作站连接并进行更新。

116

⊖ 支持至多一次语义时，你的系统必须确保任何请求要么失败，要么执行一次。如果你用其他语义实现，例如至少一次语义，相同请求可以被多次重发和执行。这种策略仅适用于幂等服务，此处，通过多于一次地执行一种服务，整体一致性不可被破坏。幂等服务的典型例子是为变量赋初值的功能。

游客能够使用CGI脚本在因特网的任何地方以在线方式订房间。房间预定费以在线方式用信用卡支付。为了安全的原因我们可以为上述事物引入加密机制。另外的httpd服务器可用来提供诸如订飞机票或火车票，纪念馆和其他名胜的订票或信息检索的附加服务。

每个CIS终端都运行一个WWW浏览器。这将允许我们使用便宜的PC和因特网PC作为终端。httpd服务器运行在快速的UNIX和Windows NT工作站上以确保短的响应时间。

9. 变体

直接通信代理者系统。你有时可以放松客户机只能通过本地代理者转发请求的限制以提高效率。在这样的变体中客户机就可以直接与服务器通信。代理者告诉客户机服务器提供了哪一条通信通道。从而客户机可以建立与被请求的服务器之间的直接链接。在这样的系统中，代理接管了代理者的责任来处理绝大部分通信活动。类似论据可用于离台（off-board）通信：这里，在适当的时候使用网桥，客户机直接与远程代理者相通，这一点与向其本地代理者发送请求以转发到远程服务器的代理者正好相反。

消息传送代理者系统。这种变体适合于以数据传输为重点的系统，而不是实现远程过程调用抽象[⊖]的系统。使用这种变体，服务器用消息的类型来确定它们必须做什么，而不是提供客户机能调用的服务。在该语境中，消息是原始数据附加说明消息类型、结构和其他相关属性的额外信息的序列。

交易器系统。客户机的请求通常被恰好转发到惟一标识的服务器。在某些环境中，服务是客户机发送其请求的目标而服务器则不是。在交易器系统中，代理者必须知道哪个服务器可以提供这项服务，并将请求转发到适当的服务器。因此，客户机端代理使用服务标识符取代服务器标识符来访问服务器功能。相同请求可能被转发到不止一台服务器而实现相同服务。 117

适配器代理者系统。你可以使用额外的层将代理者组件的接口隐藏到服务器，以增强灵活性。该适配器层是代理者的一部分，它负责注册服务器并与服务器交互。通过提供多于一个的适配器，你可以支持不同的服务器粒度和服务器定位策略。例如，如果所有被应用程序访问的服务器对象位于相同的机器上，并作为库对象实现，一个特定的适配器可以用来将对象直接链接到应用程序。另一个例子是使用面向对象的数据库维护对象。由于数据库是负责提供方法和存储对象的，就没有必要严格地登记对象。在这样的场景下，你可以提供一个特殊的数据库适配器。参见［OMG92］。

回叫代理者系统。只要不是实现一个客户机产生需求而服务器消费需求的主动通信模型，你也可以使用反应式模型。反应式模型是事件驱动的，并且对客户机与服务器不加区别。一旦事件到来，代理者将调用已注册的组件的回叫方法对事件作出反应。该方法的执行可能会产生新的事件转而导致代理者触发新的回叫方法调用。这种变体的具体细节见［Sch94］。

有几种方法将上面的变体联合起来。例如，你可以实现一个直接通信代理者系统并将它与交易器变体相结合。在这样的系统中，新来的客户机请求将导致代理者从提供所请求的服务的服务器中选择一个服务器。接着代理者建立一个客户机和被选服务器之间的直接链接。

⊖ 代理者提供的RPC（远过程调用）接口是典型的使用消息传送接口建立的。

10. 已知应用

COBRA。代理者体系结构模式用来详细说明由对象管理组（OMG）定义的公共对象请求代理体系结构（Common Object Broker Architecture，COBRA）。COBRA是一种处理异构系统上分布式对象的面向对象技术。它给出了一种接口定义语言来支持客户机与服务器对象［OMG92］的互操作性。许多COBRA实现成为直接通信代理者系统的变体。例如，IONA Technologies' Orbix[Iona95]。

118

IBM SOM/DSOM。［Cam94］代表了COBRA风格的代理者系统。与许多其他COBRA实现相反的是，它通过将COBRA接口定义语言与二元协议结合起来实现了互操作性。SOM的二进制方法支持从现有二元父类衍生出子类。你可以以一种编程语言实现SOM中的某个类而在另一种语言中由该类衍生出一个子类。

微软的**OLE 2.x**技术提供了使用代理者体系结构模式的另一个例子。与COBRA用接口定义语言确保互操作性相比，OLE 2.x定义了一个用来展示并访问服务器接口的二元标准［Bro94］。

万维网（WWW）是世界上最大的可用代理者系统。诸如HotJava、Mosaic和Netscape行为与代理者相似的超文本浏览器和WWW服务器发挥了服务提供商的作用。

ATM-P。在西门子公司建立基于ATM（异步传输方式）的电信交换系统的内部项目中我们实现了消息传送代理者系统变体［ATMP3］。

11. 效果

代理者体系结构模式有以下重要**优点**：

定位透明性。由于代理者负责定位服务器通过使用了惟一的标识符，所以客户机就不需要知道服务器的位置。类似地，服务器也不需要关心调用它的客户机的位置，因为它们是从本地代理者组件接收所有请求的。

组件的可变性和可扩展性。如果服务器发生改变而其接口保持不变，这将对客户机没有功能上的影响。修改代理者的内部实现而并不是它提供的API，除了性能发生改变外，对客户机和服务器没有影响。用于在服务器和代理者之间，客户机与代理者之间以及在代理者之间交互的通信机制发生变化，或许要求你重编译客户机、服务器或代理者。但是，你不需要改变其源代码。使用代理和网桥的一个重要原因是为了便于实现改变。

119

代理者系统的可移植性。代理者系统通过使用诸如API、代理和网桥这样的间接层，来对客户机和服务器隐藏操作系统和网络系统。需要移植时，将代理者组件及其API移植到新平台上并重编译客户机和服务器程序在绝大多数情况下这就足够了。根据层式体系结构模式，推荐将代理者组件构建成层状结构。如果最低层对代理者的其他部分隐藏系统特定细节，你只需要将最低层移植过去而不需要移植整个代理者组件。

不同代理者系统之间的互操作性。如果不同代理者系统理解消息交换的公共协议，那么它们就可以互操作。该协议是通过网桥实现并处理的，它负责将特定的代理者协议翻译成公共协议，以及将公共协议翻译成特定代理者协议。

可重用性。创建新的客户机应用程序时，可以基于现有系统来建造应用程序的功能。设想

你正在开发一个新的商业应用程序。如果提供诸如文本编辑、可视化、打印、数据库访问或电子制表功能的组件已经是可用的，你就不需要亲自来实现这些功能。将这些服务集成到你的应用程序中就可以了。

代理者体系结构模式有以下**不足**：

效率受限。使用代理者实现的应用程序通常比组件分布是静态和已知的应用程序要慢。直接依赖于用来在进程间通信的具体机制的系统在性能上也高于代理者体系结构，因为代理者引入了间接层使之具有可移植性、灵活性和可变更性。

容错性较差。与非分布式软件系统相比，代理者系统提供的容错性较低。假设一个服务器或代理者在程序执行期间失效，那么所有依赖于该服务器或该代理者的应用程序将不能成功地继续进行。你可以通过组件复制来提高可靠性。 120

以下方面既给出了**优点**也给出了**不足**：

测试和调试。从测试过的服务开发出来的客户机应用程序将更健壮并易于测试。然而，由于牵涉到很多组件，测试和调试代理者系统是一项乏味的工作。例如，客户机和服务器之间的协作失败可能有两个原因——要么服务器运进入了错误状态，要么客户机和服务器之间的通信线路的某个地方出了问题。

参见

转发器-接收器模式封装了两个组件之间的过程间通信。在客户机端，转发器从客户机接受一个请求和收件人并将其映射到所使用的IPC（inter-process communication，过程间通信）工具。服务器端的接受者解开该消息并将其发送到服务器。这种模式中没有代理者组件。很容易实现一个比代理者模式小的工具，但也缺少灵活性。

代理模式有几种情况，远程情况是其中之一。远程代理常常用于连接转发器。代理封装了服务器的接口和远程地址。转发器得到消息后并将其转化为IPC层代码。

客户机-分配器-服务器模式是直接通信代理者变体的一个不重要的版本。分配器分配、打开，并维护客户机与服务器之间的直接通道。

中介者设计模式［GHJV95］用星形配置代替了对象间连接的网，在星形配置下，中心的中介者组件通过定义与对象通信的公共接口封装了聚集行为。与使用代理者模式一样，中介者模式使用通信集线器，但也有几点重要区别。代理者模式是一个大规模的基础结构范例——它并不用来建立单一的应用程序，而是作为整个应用程序族的平台。它并不局限于处理本地计算，它分配和监视请求而不关心请求的发送者或请求的内容。相反，中介者模式通过检测一个请求关于什么以及它可能来自何处封装应用程序语义——接着才能确定应该做什么。它可能向发送者返回一个消息，按自己的意愿完成请求，或者包括多于一个其他组件。 121

致谢

我们衷心感谢OOPSLA'95的并发和分布式系统模式专题讨论会的与会代表，他们评审了这个代理者模式。特别感谢Jim Coplien、David Delano、Doug Schmidt和Steve Vinoski，他们评审了代理者描述的早期版本并提出了宝贵的建议。 122

2.4 交互式系统

当今的系统主要是通过图形用户接口来达到与用户的高度交互。其目的是为了增强应用程序的可用性。可用的软件系统提供了对其服务的便利的访问方法，因而能使用户快速学习该应用程序并产生结果。

详细说明这种系统的体系结构时，主要的挑战是保持功能内核独立于用户接口。交互式系统的内核基于系统功能需求，通常保持稳定。然而，用户接口常常要经受变化和改建。例如，系统可能必须支持不同的用户接口标准，特定用户关于“式样和感觉”的比喻，或必须对其进行调整以适应客户业务过程的接口。这就需要能支持用户接口改建而对特定应用程序或底层软件的数据模型不产生重要影响的体系结构。

我们描述了两种模式，它们能够为交互式软件系统提供一个基本的结构化组织：

- 模型-视图-控制器（*Model-View-Controller, MVC*）模式将一个交互式应用程序分为三个组件。模型包含核心功能和数据。视图向用户显示信息。控制器处理用户输入。视图和控制器共同构成了用户接口。变更传播机制确保了用户接口和模型之间的一致性。
- 表示-抽象-控制（*Presentation-Abstraction-Control, PAC*）模式以合作agent的层次形式定义了交互式软件系统的一种结构。每个agent负责应用程序功能的某一特定方面，并且由表示、抽象和控制三个组件构成。这种细分将agent的人机交互部分与其功能内核和它与其他agent的通信分隔开来。

123

MVC很可能为交互式软件系统提供了最著名的体系结构方面的组织。MVC是由Trygve Reenskaug [RWL96]首创并第一个在Smalltalk-80环境下实现[KP88]。它为许多交互式系统和具有图形用户接口的软件系统的应用框架提供了基础，例如，MacApp[App89]，ET++[Gam91]，当然也包括Smalltalk库。甚至微软的基础类库[Kru96]也遵循MVC的原则。

然而，我们的目的并不是解释Smalltalk MVC的实现——为了给出基本原理的更清晰的理解，我们略去了很多Smalltalk的 MVC实现细节。几乎没有读者会自己去创建一个新的MVC的框架，但很可能要使用现有的框架或依据MVC的主要原则划分他们的应用程序。

PAC的应用没有MVC广泛，但这并不意味着它不值得描述。作为构建交互式应用程序的一个可供选择的方法，PAC尤其适用于由几个自给子系统组成的系统。PAC也用于MVC没有解决的问题，如怎样有效组织功能内核的不同部分与用户接口之间的通信。Joelle Coutaz[Cou87] 最先描述了PAC。PAC的第一个应用是在人工智能领域［Cro85］。

124

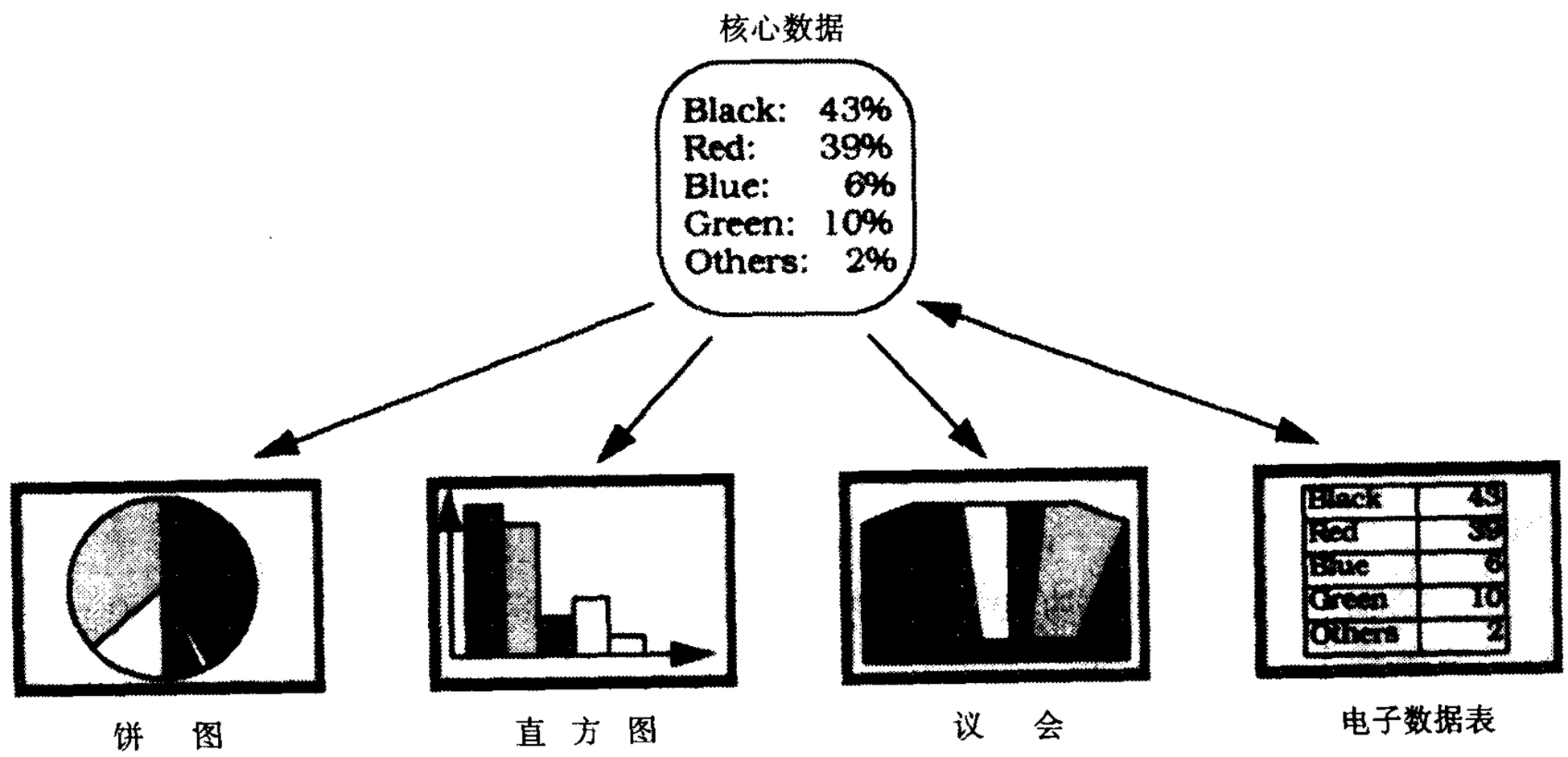
2.4.1 模型-视图-控制器

模型-视图-控制器（*Model-View-Controller, MVC*）体系结构模式将一个交互式应用程序分为三个组件。模型包含核心功能和数据。视图向用户显示信息。控制器处理用户输入。视图和控制器共同构成了用户接口。变更-传播机制确保了用户接口和模型之间的一致性。

1. 例子

考虑一个采用比例表示的用于政治选举的一个简单信息系统。它提供了一个输入数据的电

子数据表和表示当前结果的几种图表。用户可以通过图形接口与系统交互。所有信息显示必须立即反映出选举数据的变化。



在对系统没有较大的影响条件下，它应该能够集成新的数据表示方法，如政党间的议员席位的分配。系统应该也能被移植到具有不同的“式样和感觉”标准的平台上，如运行Motif的工作站或运行微软Windows 95的PC。 125

2. 语境

具有灵活人-机接口的交互式应用程序。

3. 问题

用户接口尤其容易改变需求。当你扩展应用程序的功能时，你必须修改菜单以访问这些新功能。客户可能要求一个特殊的用户接口匹配，或系统需要移植到有不同“式样和感觉”标准的另一个平台上。甚至将你的开窗口的系统升级到新版本时就意味着要改写代码。因此，长寿系统的用户接口平台表示一个移动中的目标。

不同的用户对用户接口提出了相互冲突的要求。打字员通过键盘输入信息。经理主要想使用点击图标和按钮的系统。因此，对几种用户接口范型的支持当然应该合并起来。

如果用户接口是与功能内核紧密交织在一起的，建立具有所需的灵活性的一个系统是昂贵的而且易于出错。它将导致需要开发并维护几个本质上不同软件系统，每个用户接口对应一个实现。确保变更跨多个模块传播。以下强制条件影响了解决方案：

- 相同的信息在不同的窗口中有不同的表示，例如，直方图和饼图。
- 应用程序的显示和行为必须立即反映出对数据的操作。
- 用户接口应易于改变，甚至在运行期间也应该可以修改。
- 支持不同的“式样和感觉”标准或移植用户接口不应影响应用程序内核的代码。

4. 解决方案

模型-视图-控制器（MVC）首先在Smalltalk-80编程环境中引入［KP88］。MVC将交互式应用程序分为三个区域：处理、输出和输入。

126 模型组件封装了内核数据和功能。模型独立于特定输出表示法或输入方式。

视图组件向用户显示信息。视图从模型获得数据。可能有模型的多个视图。

每个视图都有一个相关的控制器组件。控制器接受输入，通常作为将鼠标移动、鼠标按钮的活动或键盘输入编码的事件。事件被翻译成模型或视图的服务器请求。用户仅仅通过控制器与系统交互。

模型与视图和控制器组件的分离将允许同一个模型的多个视图。如果用户通过一个视图的控制器改变了模型，所有依赖于该数据的其他视图应该反映出这种变化。因此一旦模型的数据发生了变化，模型要通报所有视图。视图反过来从模型恢复新数据并更新所显示的信息。这种变更-传播机制在出版者-订阅者模式中描述。

5. 结构

模型组件包含应用程序的功能内核。它封装了相应的数据并输出执行特定应用程序处理的过程。控制器代表用户调用所有这些过程。模型也提供访问其数据的函数。这些数据由获得待显示的数据的视图组件使用。

变更-传播机制维护了一个模型中相依组件的注册表。所有的视图还有挑选出来的控制器注册它们有关变更的待通知的需求。模型状态的改变触发变更-传播机制。变更-传播机制是模型与视图和控制器之间的惟一链接。

127

视图组件向用户呈现信息。不同的视图用不同的方法呈现模型的信息。每个视图定义了一个被变更-传播机制激活的更新过程。更新过程被调用时，视图就会恢复从模型来的待显示的当前数据值，并将其显示在屏幕上。

在初始化阶段，所有视图都与该模型相关，并向变更-传播机制注册。每个视图将创建一个合适的控制器。视图与控制器之间的对应是一对一的关系。视图通常提供允许控制器操作显示器的功能。这对不影响模型的用户触发操作如卷动是有用的。

类 模型	协作者 • 视图 • 控制器
责任 • 提供应用程序的功能核心 • 注册相关的视图和控制器 • 公布有关数据变更的相关组件	

控制器组件接受作为事件的用户输入。这些事件如何发送到控制器取决于用户接口平台。为了简化起见，让我们假设每个控制器实现了一个由相关事件调用的事件处理过程。事件被翻译成对模型或相关视图的请求。

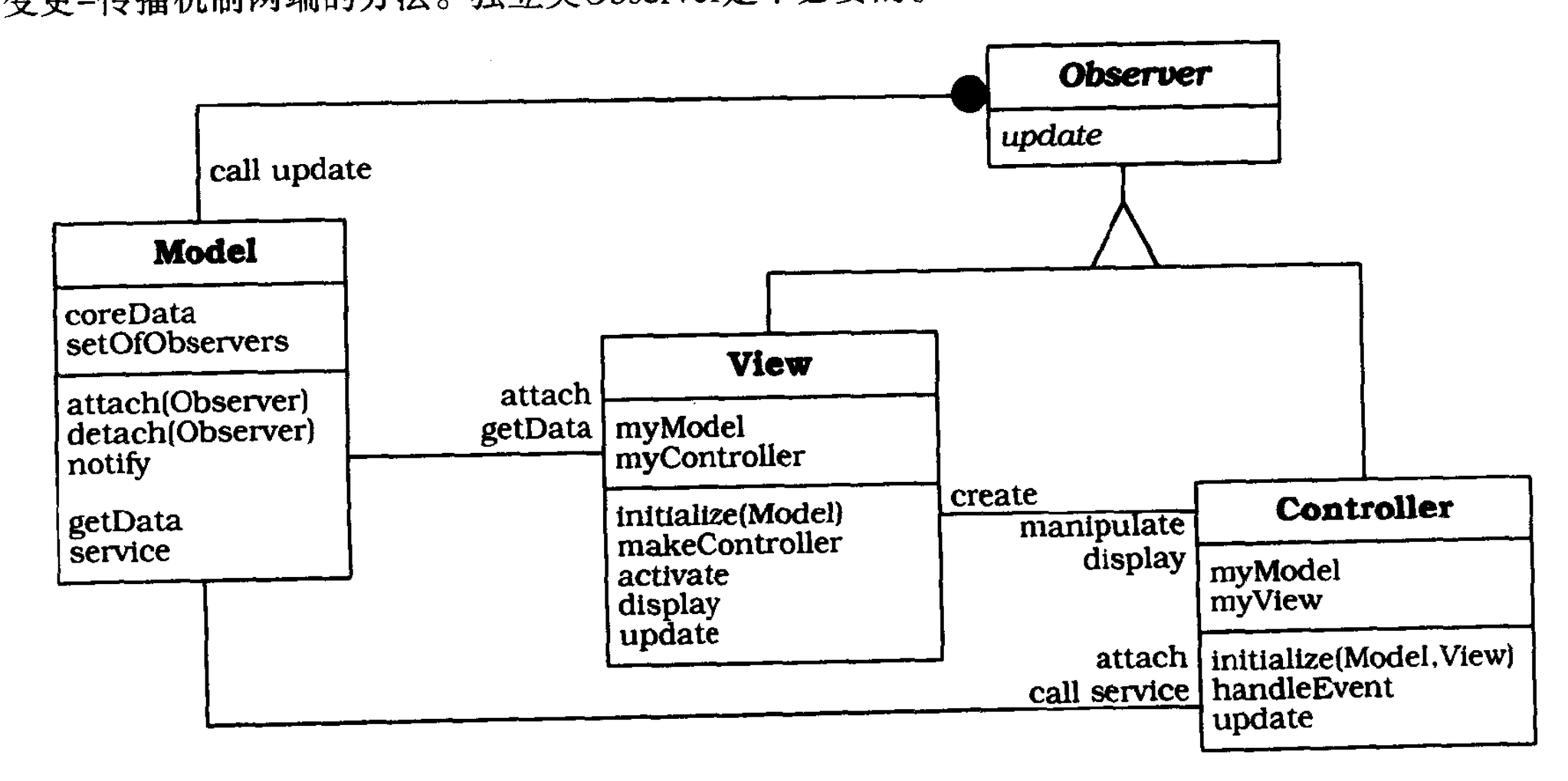
如果控制器的行为依赖于模型的状态，控制器向变更-传播机制注册自己并实现一个更新过程。例如，当模型的改变使一个菜单输入能用或禁用时，这是很必要的。

类 • 视图	协作者 • 控制器 • 模型
责任 • 创建和初始化相联的控制器 • 对用户展示信息 • 实现更新过程 • 以模型恢复数据	

类 • 控制器	协作者 • 视图 • 模型
责任 • 接受用户输入作为一个事件 • 将事件翻译成对模型的服务请求，或对视图的展示请求 • 如果需要的话，实现更新过程	

128

MVC的面向对象实现将为每个组件定义一个独立的类。在C++实现中，视图和控制器类共享一个公共父类，该父类定义了更新的接口。这在下图中给出。在Smalltalk中，类Object定义了变更-传播机制两端的方法。独立类Observer是不必要的。



➡在我们的例子系统中，模型保存了每个政党累积的选票并允许视图检索选票数。它进一步向控制器输出数据操作过程。❑

我们定义了几个视图：直方图、饼图和表。图型视图使用了不影响模型的控制器，而表视图连接到用做数据入口的控制器。

你也可以使用MVC模式为交互式应用程序建立一个框架，与在Smalltalk-80环境下所作的类似［KP88］。这样一个框架为诸如菜单、按钮或列表等常用用户接口元素提供了预先准备的视图和控制器的子类。为了实例化一个应用程序的框架，你可以使用组合模式［GHJV95］将现有

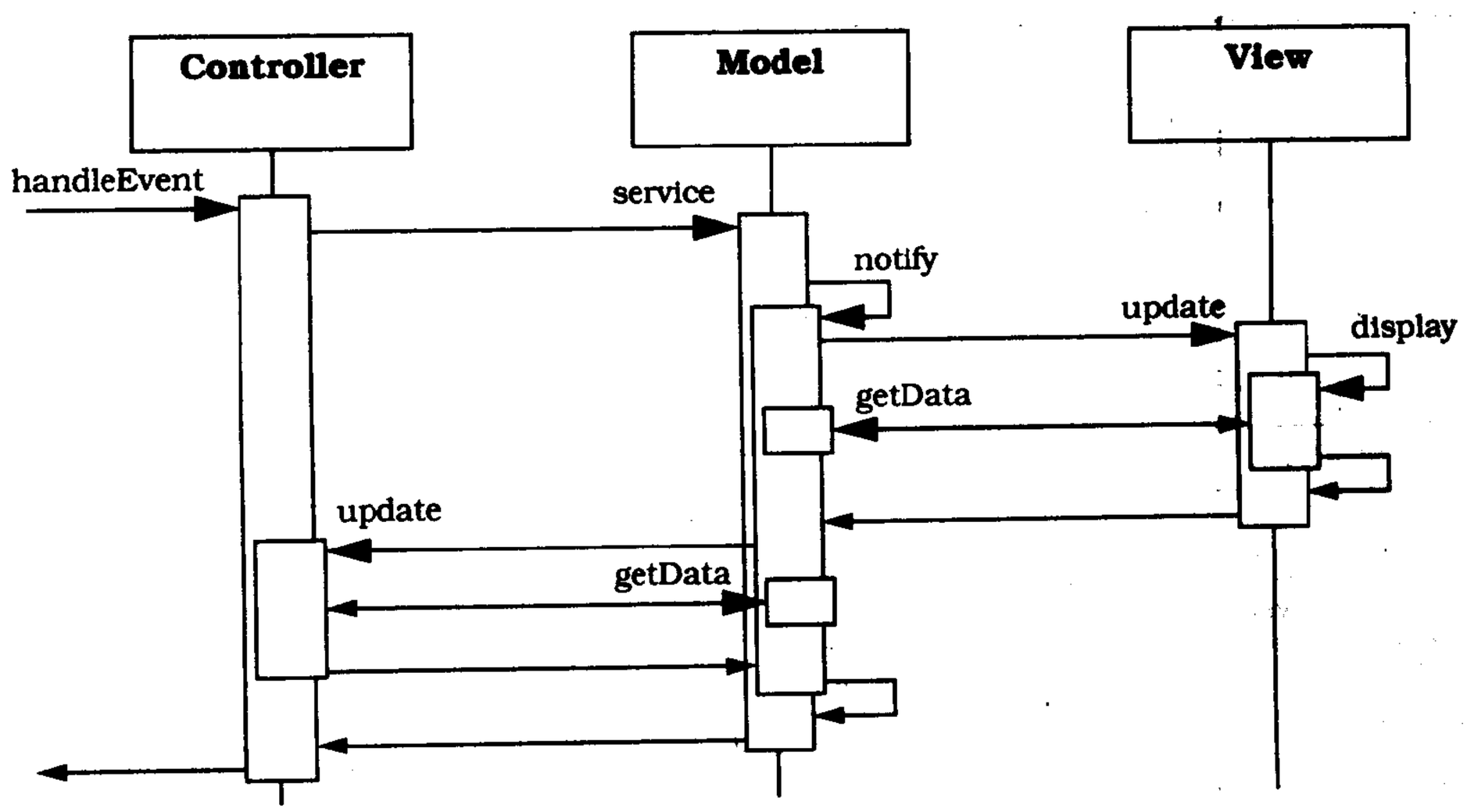
129 用户接口元素分等级地结合起来。

6. 动态特性

以下场景描绘了MVC的动态行为。为了简单起见，在图中只给出了一个视图-控制器对。

场景I 给出了用户输入如何导致触发变更-传播机制的模型的改变。

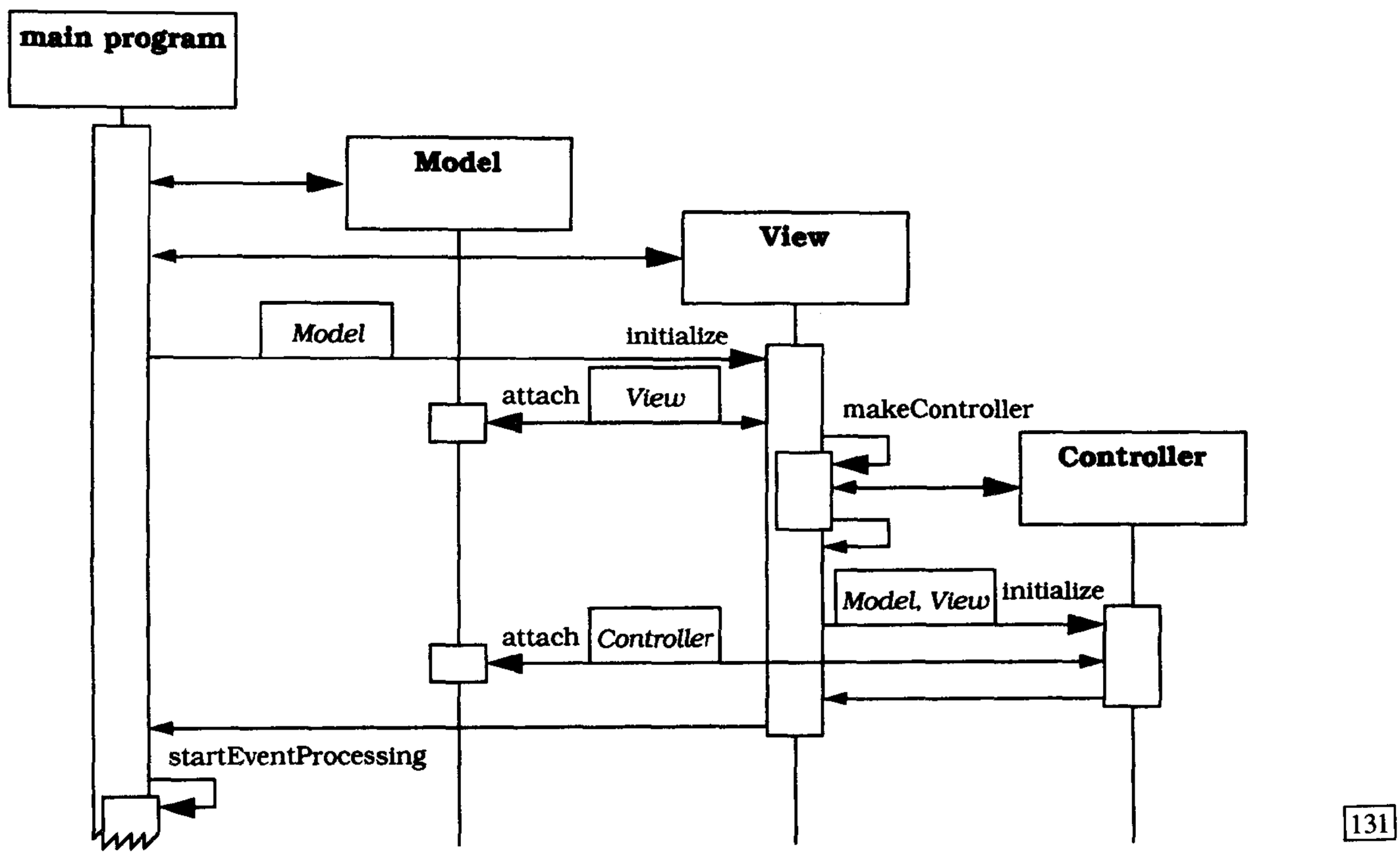
- 控制器在其事件处理过程中接受用户输入，解释事件，并启动模型的服务过程。
- 模型执行所请求的服务。这将导致其内部数据的变化。
- 模型通过调用其更新过程通知所有向变更-传播机制注册的视图和控制器所发生的改变。
- 每个视图向模型请求变化的数据并将其重新显示在屏幕上。
- 每个注册的控制器从模型检索数据来能用或禁用某个用户功能。例如，激活存储数据的菜单入口可以是对模型的数据修改的结果。
- 最初的控制器恢复控制并从其事件处理过程返回。



130

场景II 给出了MVC三位一体的结构是如何初始化的。这些代码通常位于模型、视图和控制器的外部，例如在主程序中。视图和控制器的初始化为模型打开每个视图时同样发生。将出现以下步骤：

- 创建模型实例，继而初始化其内部数据结构。
- 创建视图对象。它取一个对模型的引用作为模型初始化参数。
- 视图通过调用附属过程支持模型的变更-传播机制。
- 视图通过创建它的控制器继续初始化。它向控制器的初始化过程发送对模型和对自身的引用。
- 控制器也通过调用附属过程来支持变更-传播机制。
- 初始化后，应用程序开始处理事件。



7. 实现

下面的第1步~第6步是写一个基于MVC的应用程序的基本步骤。第7步~第10步描述了产生更高自由度的附加主题，并引导他们实现高度灵活的应用程序或应用程序框架。

(1) 人机交互从核心功能中分离出来。分析应用程序领域并将核心功能从设想的输入和输出行为中分离出来。设计你的应用程序的模型组件来封装内核所需的数据和功能。提供访问待显示数据的功能。确定模型功能的哪一部分应该通过控制器向用户展示，并给模型添加相应的接口。

➡我们的例子中的模型在两个等长的列表中存储了政党的名字和相应的选票[⊖]。提供了两种访问列表的方法，每种方法都创建了一个迭代程序。模型也提供了改变选举数据的方法。

```
class Model{
    List<long>   votes;
    List<String> parties;
public:
    Model(List<String> partyNames);

// access interface for modification by controller
    void clearVotes(); // set voting values to 0
    void changeVote(String party, long vote);

// factory functions for view access to data
```

⊖ 与政党名称相关联的数组作为关键字，选票作为信息或许是一种更为真实的实现，但是这样会使例子代码过度膨胀。

```
    Iterator<long>   makeVoteIterator(){
        return Iterator<long>(votes);
    }
    Iterator<String> makePartyIterator(){
        return Iterator<String>(parties);
    }
// ... to be continued
}
```

❑

(2) 实现变更-传播机制。对此采用出版者-订阅者设计模式并指定模型的出版者的作用。用具有对观察者对象引用的注册来扩展系统。提供过程以允许视图和控制器赞成和不赞成变更-传播机制。模型的通知过程调用所有观察对象的更新过程。所有改变模型状态的模型的过程在变化执行后调用通知过程。

132

➡正确的C++使用建议应该定义一个抽象类Observer来拥有更新接口。视图和控制器继承于Observer。第1步中的Model被扩展成拥有对当前观察者引用的一个集合，并且两种方法——attach()和detach()——允许观察对象赞成和不赞成变更-传播机制。修改模型状态的方法将调用方法notify()。

```
class Observer{ // common ancestor for view and controller
public:
    virtual void update() { }
// default is no-op
};

class Model{
// ... continued
public:
    void attach(Observer *s) { registry.add(s); }
    void detach(Observer *s) { registry.remove(s); }
protected:
    virtual void notify();
private:
    Set<Observer*> registry;
};
```

我们在注册表中所有的Observer对象上迭代实现方法notify()，并调用其更新方法。我们没有提供独立的功能来为注册表创建一个迭代器，因为它只在内部使用。

```
void Model::notify(){
    // call update for all observers
    Iterator<Observer*> iter(registry);
    while (iter.next()){
        iter.curr()->update();
    }
}
```

❑

133

方法changeVote()和clearVote()在选举数据改变后调用notify()。

(3) 设计并实现视图。设计每个视图的外观。指定并实现一个画图过程来将视图显示在屏幕上。这个过程从模型获得待显示的数据。画图过程剩下的部分主要依赖于用户接口平台。例如，它将调用画直线或再现文本的过程。

实现更新过程来反映模型的变化。最简单的方法是简单调用画图过程。画图过程继续进行并获得视图所需的数据。对需要频繁更新的复杂视图而言，这种更新直截了当地实现是低效率的。在这种情况下有几种优化策略。一种是向更新过程提供额外的参数。这样视图就可以决定是否有必要重画。另一种方法是当还有事件也可能需要视图重画时，只安排但不执行视图的重画。当没有更多事件需要时，视图就可以重画了。

除更新和画图过程之外，每个视图需要一个初始化过程。初始化过程支持模型的变更-传播机制并建立起与控制器之间的关系，如第5步中所示。控制器初始化以后，视图将自己显示在屏幕上。平台或控制器或许需要附加的视图能力，比如实现视图窗口缩放的过程。

➡我们为选举系统使用的所有视图定义了一个公共基类View。模型与控制器之间的关系用与相应的访问方法相对应的两个成员变量来表示。View的构造函数通过支持变更-传播机制建立了与模型之间的关系。析构函数又通过不支持该机制将其删除。View还提供了一个简单的未优化的update()实现。

```
class View : public Observer {
public:
    View(Model *m) : myModel(m), myController(0)
        { myModel->attach(this); }
    virtual ~View() { myModel->detach(this); }
    virtual void update() { this->draw(); }
    // abstract interface to be redefined:
    virtual void initialize() ;// see below
    virtual void draw() ;      // (re-)display view
// ... to be continued below
    Model *getModel() { return myModel; }
    Controller *getController() { return myController; }
protected:
    Model        *myModel;
    Controller   *myController; // set by initialize
};

class BarChartView : public View {
public:
    BarChartView(Model *m) : View(m) { }
    virtual void draw();
};

void BarChartView::draw(){
    Iterator<String> ip = myModel->makePartyIterator();
    Iterator<long> iv = myModel->makeVoteIterator();
    List<long> dl; //for scaling values to fill screen
    long     max = 1;// maximum for adjustment

    // calculate maximum vote count
    while (iv.next()) {
        if (iv.curr() > max ) max = iv.curr();
    }
    iv.reset();
    // now calculate screen coordinates for bars
    while (iv.next()) {
        dl.append((MAXBARSIZE * iv.curr())/max);
    }
```

134

```
        // reuse iterator object for new collection:
        iv = dl; // assignment rebinds iterator to new list
        iv.reset();

        while (ip.next() && iv.next()) {
            // draw text: cout << ip.curr() << " : " ;
            // draw bar: ... drawbox(BARWIDTH, iv.curr());...
        }
    }
```

BarChartView类的定义说明了我们系统的一个特殊视图。它重定义了draw()来将选举数据显示成直方图。 ❑

(4) 设计并实现控制器。对应用程序中的每个视图，指定回应用户动作的系统的行为。假设基础平台将用户的每个动作作为一个事件来发送。控制器用一个专用过程来接受并解释这些事件。对一个重要的控制器来说，这种解释依赖于模型的状态。

135

控制器的初始化将其绑定到它的模型和视图并使之能进行事件处理。如何实现它取决于用户接口平台。例如，控制器可能向窗口系统注册自己的事件处理过程视为一次回叫。

➡我们例子中的绝大部分视图不需要任何特定的事件处理——它们仅用来显示。因此，我们定义一个带有空的handleEvent()方法的基类Controller。构造函数将控制器连接到其模型而析构函数又将其分开。

```
class Controller : public Observer {
public:
    virtual void handleEvent(Event *) { }
        // default = no op

    Controller( View *v) : myView(v) {
        myModel = myView->getModel();
        myModel->attach(this);
    }

    virtual ~Controller() { myModel->detach(this); }
    virtual void update() { } // default = no op
protected:
    Model    *myModel;
    View     *myView;
};
```

我们省略了独立控制器初始化方法，因为视图和模型之间的关系已经由其构造函数建立起来了。 ❑

调用功能内核将控制器与模型紧密地链接起来，因为控制器变得依赖于特定应用程序的模型接口。如果你计划修改功能，或想提供可重用的控制器并因而希望控制器独立于特定的接口，那么就要应用命令处理器设计模式。模型起到命令处理器模式的供应者的作用。命令类和命令处理组件是控制器和模型之间附加的组件。MVC控制器在命令处理器中起到控制器的作用。

136

(5) 设计并实现视图-控制器关系。视图在其初始化期间通常创建与其关联的控制器。当你建立视图和控制器的类层次时，应用工厂方法设计模式［GHJV95］并在视图类中定义方法makeController()。需要不同于其超类的控制器的每个视图都要重新定义工厂方法。

➡在我们的C++例子中，View基类实现了方法initialize()。该方法反过来调用工厂方

法makeController()。我们不能够将对makeController()的调用放进视图类的构造函数中，因为这样子类所重定义的makeController()就不能像所期望的那样获得调用。仅有的一个需要特殊控制器的View子类是TableView。我们重定义makeController()来返回一个Controller以使从用户那里接受数据。

```
class View : public Observer {
// ... continued
public:
//C++ deficit: use initialize to call right factory method
    virtual void initialize()
        { myController = makeController();}
    virtual Controller *makeController()
        { return new Controller(this); }
};

class TableController : public Controller {
public:
    TableController(TableView *tv) : Controller(tv) {}
    virtual void handleEvent(Event *e) {
    // ... interpret event e,
    //     for instance, update votes of a party
        if(vote && party){ // entry complete:
            myModel->changeVote(party,vote);
        }
    }
};
class TableView : public View {
public:
    TableView(Model *m) : View(m) { }
    virtual void draw();
    virtual Controller *makeController()
        { return new TableController(this); }
};
```

❑ 137

(6) 实现MVC的准备（*set-up*）。准备代码首先初始化模型，进而创建和初始化视图。初始化之后，事件处理开始，通常进入一个循环，或是包括一个循环的过程，如来自X工具箱的XtMainLoop()。因为模型应该保持特定视图和控制器的独立性，该准备代码应该置于外部，例如，放在主程序中。

➡在我们的简单例子中，主函数初始化了模型和几个视图。事件处理程序将事件发送到表视图的控制器，允许选举数据的进入和改变。

```
main() {
    // initialize model
    List<String> parties;    parties.append("black");
    parties.append("blue "); parties.append("red  ");
    parties.append("green"); parties.append("oth. ");
    Model m(parties);

    // initialize views
    TableView *v1 = new TableView(&m);
    v1->initialize();
    BarChartView *v2 = new BarChartView(&m);
```

```
v2->initialize();
// now start event processing ...
```

❑

(7) 创建动态视图。如果应用程序允许动态打开和关闭视图，那么提供用来创建组件是个不错的想法。例如，这个组件也可以在最后一个视图被关闭后负责终止应用程序。可以用视图管理器设计模式来实现这个视图管理组件。

(8) “可插入”控制器。视图与控制方面的分离支持一个视图的不同控制器的组合。这种灵活性可以用来实现操作的不同模式，如相对于专家而言的偶然用户，或采用忽略任何输入的控制器创建只读视图。这种分离的另一种用法是将新的输入和输出设备与一个应用程序集成在一起。例如，给残疾人用的导盲设备的控制器就可以利用现有模型和视图的功能并可以方便的加入系统之中。

138 ➡我们的例子中仅有类TableView支持多个控制器。默认控制器TableController允许用户输入选举数据。如果只为了显示的目的，TableView可以设定成控制器忽略所有用户输入。下面的代码给出了一个控制器如何被另一个控制器所取代的。特别指出的是，setController返回了以前使用的控制器对象。在此，控制器对象不再使用，因此被立即删除。

```
class View : public Observer{
// ... continued
public:
    virtual Controller *setController(Controller *ctlr);
};

main()
// ...
    // exchange controller
    delete v1->setController(
        new Controller(v1)); // this one is read only
// ...
    // open another read-only table view;
    TableView *v3 = new TableView(&m);
    v3->initialize();
    delete v3->setController(
        new Controller(v3)); // make v3 read-only
    // continue event processing
// ...
}
```

❑

(9) 层次化视图和控制器的基础结构。基于MVC的框架实现了可重用的视图和控制器类。对经常层次化使用的用户接口元素（如按钮、菜单、或文本编辑器）来说一般都是这样实现的。于是，一个应用程序的用户接口主要靠结合预先定义好的视图对象来构建。可以使用组合模式［GHJV95］来创建层次式静态视图。如果同时激活多个视图，可能有几个控制器同时关心事件。例如，对话框内的按钮对鼠标点击有反应，但对键盘上敲的‘a’没用响应。如果父辈对话视图也包含一个文本域，‘a’会被发送到文本视图的控制器。事件以某种已定义序列形式分配到所有活动控制器的事件处理过程。可使用职责链模式［GHJV95］来管理事件的委托。如果职责链
139 建立得合适，控制器将把未处理的事件传送到父辈视图的控制器或兄弟视图的控制器。

(10) 进一步去除系统依赖性。建立一个精心选择的视图和控制器类的集合框架是代价昂贵的。你可能想让这些类的平台相互独立。这在某些Smalltalk系统中已做到了。你可以通过应用桥接模式［GHJV95］为系统与基础的平台之间提供另一个间接层。视图使用名为*display*的一个类作为窗口的抽象，而控制器使用*sensor*类。

抽象类*display*定义了创建窗口，绘制线和文本，改变鼠标光标外观等方法。*sensor*抽象定义了独立于平台的事件，并且每个具体的*sensor*子类将特定系统事件映射到独立于平台的事件。对每个支持的平台，要实现封装了系统特性的具体的*display*和*sensor*子类。

抽象类*display*和*sensor*的设计是重要的，因为它既影响了结果代码的效率也影响了可以在不同平台上实现的具体类的效率。一个方法是使用直接由所有用户接口平台提供仅有的具备非常基本功能的*sensor*和*display*抽象。另一个极端是让*display*和*sensor*提供更高层次的抽象。这样的类的移植要付出更多的努力，但可以使用来自用户接口平台的原有代码。第一种方法产生出跨平台看上去相似的应用程序，而第二种方法产生与特定平台的指导方针更加一致的应用程序。

9. 变体

文档视图。这种变体放松了视图与控制器的隔离。在几种GUI平台中，窗口显示和事件处理是紧密交织在一起的。例如，X Window系统报告与窗口相关的事件。你可以通过牺牲控制器的可交换性将MVC中的视图和控制器的职责结合在一个组件中。这种结构常称为文档-视图体系结构[App89]、[Gam91]、[Kru96]。文档组件对应于MVC中的模型也实现了变更-传播机制。文档-视图的视图组件结合了MVC中控制器和视图的职责，实现了系统的用户接口。就像MVC中一样，文档和视图组件的松散耦合使相同文档可以同时有多个同步的但不同的视图。 140

10. 已知应用

Smalltalk[GR83]。模型-视图-控制器模式是最著名的使用实例是Smalltalk环境下的用户接口框架[LP91]、[KP88]。建立MVC以构建可重用的用户接口组件。这些组件被组成Smalltalk开发环境的工具所共享。然而，MVC范型被证明对于采用Smalltalk开发的其他应用程序也是有用的。VisualWorks Smalltak环境通过经由*display*和*sensor*类分离视图和控制器支持不同的“式样和感觉”标准，正如“实现”小节第10步中所描述的。

MFC [Kru96]。模型-视图-控制器模式的文档-视图变体被集成到Visual C++环境中——微软基础类库用来开发Windows应用程序。

ET++ [Gam91]。应用程序框架ET++也使用文档-视图变体。典型的基于ET++应用程序实现了它自己的文档类和一个相应的视图类。ET++通过定义一个封装了用户接口平台附属物的类`WindowPort`建立了“式样和感觉”独立性，就同类*display*和*sensor*所做的一样。

11. 效果

模型-视图-控制器的应用程序有以下**优点**：

同一模型的多视图。MVC将模型严格与用户接口组件分离。因此多视图可以在单一的模型中实现和使用。运行期间，可以同时打开多个视图，并且视图可以动态地打开和关闭。

同步化视图。模型的变更-传播机制确保了所有加入的观察者在正确的时间被告知应用程序数据的变化。它将所有相互依赖的视图和控制器同步起来。

“可插入”的视图和控制器。MVC的概念的分离允许你交换模型的视图和控制器对象。用户接口对象甚至可以在运行期间替换。

141

“式样和感觉”的可交换性。由于模型独立于所有用户接口代码，因此MVC应用程序到新平台的移植并不影响应用程序的功能内核。你只需为每个平台实现合适的视图和控制器组件。

框架潜力。正如“实现”小节第7步~第10步所概述的，基于这个模式的应用程序框架是可能的。不同的Smalltalk开发环境已经证明了这一点。

MVC的不足如下所示：

增加了复杂性。严格遵循模型-视图-控制器结构并不总是创建交互式应用程序的最好方法。Gamma[Gam91]指出对菜单和简单文本元素采用分离的模型、视图和控制器组件增加了复杂性而没有获得多少灵活性。

潜在过多的更新因素。如果一个单一的用户动作就导致许多更新，那么模型应该跳过不必要的变化通知。有可能并不是所有的视图都关心模型中发生的变化传播。例如，具有图符化窗口的视图直到窗口恢复其正常大小时才需要更新。

视图和控制器之间的紧密连接。控制器和视图是分离的但又是紧密相关的组件，它妨碍了它们的单个重用。一个视图离开其控制器使用好像是不可能的，而反之亦然，例外情况是共享忽略所有输入的控制器的只读视图。

视图和控制器与模型的紧密耦合。视图和控制器组件都是直接调用模型。它表明模型接口的改变很可能使视图和控制器的代码无效。如果系统使用多个视图和控制器，这个问题就被放大了。你可以通过采用命令处理器模式解决这个问题，如“实现”小节中所描述的，或是采用其他一些间接方法。

142

视图中数据访问的低效率。依赖于模型接口，视图可能需要做多次调用来获得所有要显示的数据。如果更新很频繁，不必要的对来自模型的未改变的数据的请求将弱化系统性能。视图内数据缓冲改善了响应时间。

移植时对视图和控制器更改是不可避免的。所有的用户接口平台的附属物被封装在视图和控制器中。然而，对每个组件也包含特定平台无关的代码。因此，MVC系统的移植因而需要在重写之前分离依赖于平台的代码。在MVC框架或大的任务应用情况下，可能需要另外的平台附属物的封装。

借鉴现代的用户接口工具使用MVC的困难性。如果不关心可移植性问题，使用高级的工具箱或用户接口软件可以排除MVC的使用。翻新工具箱组件或用户接口输出布局工具通常是很昂贵的。附加的包装也许是最小需求。另外，许多高级工具或工具箱定义了自己的控制流并且在内部处理事件，如显示弹出式菜单或卷动窗口。最后，高级用户接口平台可能已经为每种用户活动解释了事件并提供了回调。图中，由工具箱提供了绝大部分控制器的功能，就不需要独立的组件了。

参见

表示-抽象-控制模式采用了不同的方法来将系统的用户接口方面从其功能内核隔离出来。它的对应于MVC中模型的抽象组件，同视图与控制器一起被组合成一个表示组件。抽象组件和表示组件之间的通信由控制组件隔离。表示组件和抽象组件之间的交互并不仅限于调用更新过

程，如在MVC中所做的一样。

致谢

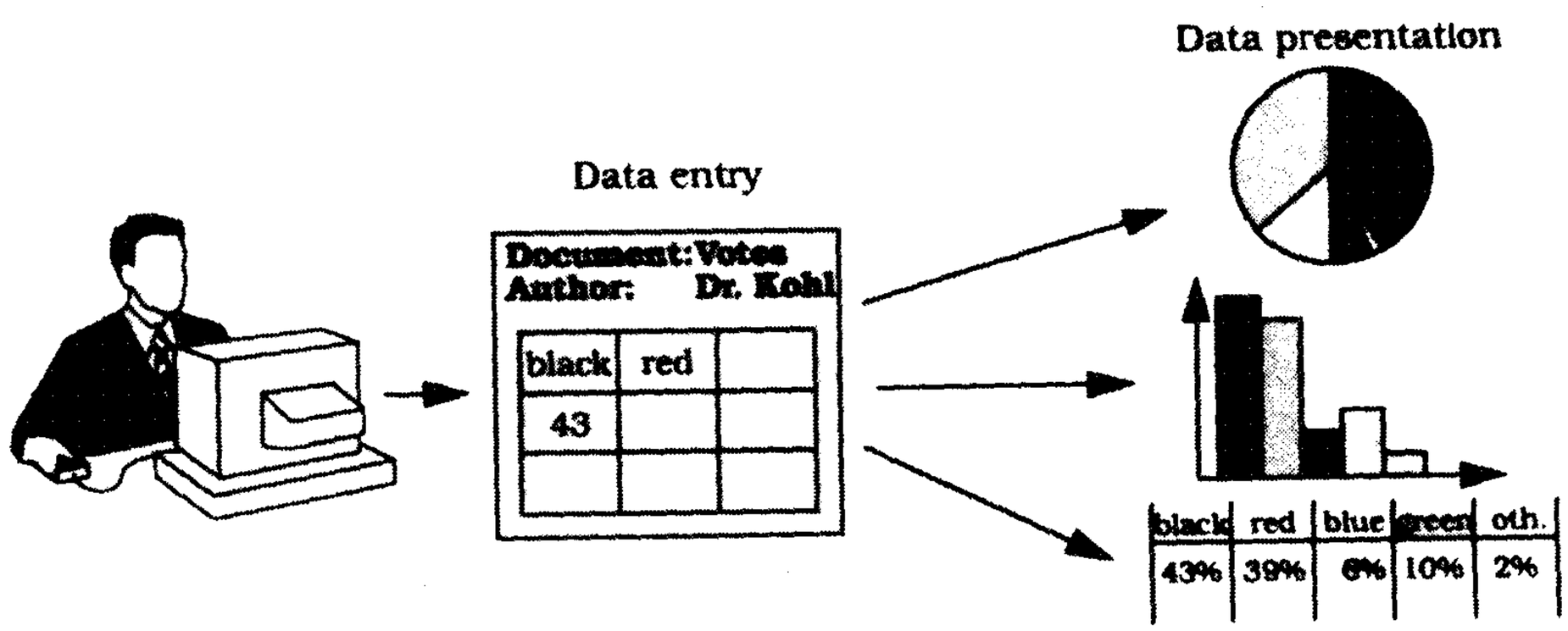
Trygve Reenskaug创建了MVC并将其引入到了Smalltalk环境中［RWL96］。 143

2.4.2 表示-抽象-控制

表示-抽象-控制（*Presentation-Abstraction-Control, PAC*）体系结构模式以合作agent的层次形式定义了交互式软件系统的一种结构。每个agent负责应用程序功能的某一特定方面，并且由表示、抽象和控制三个组件构成。这种细分将agent的人机交互部分与其功能内核和它与其他agent的通信分隔开来。

1. 例子

考察一个简单的用比例表示的政治选举信息系统。这里提供了用于输入数据的电子数据表格和几种表示当前状态的图表。用户通过图形接口与软件进行交互。



当然，不同版本适用于不同需求的用户界面。例如，一种版本支持数据的附加视图，如政党的国会席位分配。

2. 语境

在agent的协助下开发一个交互式应用程序[⊖]。 145

3. 问题

交互式系统经常被视为一个协作agent的集合。人机交互中描述的agent接受用户的输入并显示数据。其他的agent维护系统的数据模型并提供对这些数据进行操作的功能。附加的agent负责诸如错误处理或同其他软件系统的通信的任务。除了系统功能的横向分解外，我们常常还做纵

⊖ 在该模式的语境中，一个agent表示一个信息处理组件，组件包括事件接受程序和转发程序，保持状态的数据结构，以及处理输入事件，更新自身状态的处理器，组件可以产生新事件[BaCo91]。Agent可以小至单个对象，亦可复杂至一完整的软件系统。在该模式描述中，我们使用术语agent和PAC agent表示相同意思。

向分解。例如，产品规划系统（Production Planning Systems, PPS），不同于产品规划和以前的指定产品计划的执行。可以为这个任务定义一个独立的agent。

在这种协作agent体系结构中，为某一特定的任务指定一个agent，所有的agent共同提供整个系统的功能。这种体系结构也包含了横向分解和纵向分解。下面的特性影响了问题的解决方案：

- Agent常常维护它们自己的状态和数据。例如，在PPS系统中，产品的规划和实际的产品控制可以作用在不同的数据模型上，一个协调规划与模拟，另一个为有效生产做性能优化。但是，单个agent之间必须有效协作来提供应用程序的全部功能。为了达到该目标，它们需要一个交换数据、消息和事件的机制。
- 由于它们各自的人机交互常常差别较大，交互agent提供它们各自的用户接口。例如，将数据输入到电子数据表格是使用键盘输入来完成的，而图形对象的操纵是使用标定设备来完成的。
- 系统随时间而进化，它们的表示特性特别易于变化。图形的使用以及最近的多媒体特色，即是用户接口广泛变化的实例。对单个agent的变更或者用新的agent对系统的扩展，都不应该影响整个系统。

4. 解决方案

以PAC agent树状层次结构构建交互式应用程序。应有一个顶层agent，几个中间层agent，较多的底层agent。每个agent都负责应用功能的某一特定方面，且由表示、抽象和控制等三部分组成。整个层次图反映了agent之间的传递依赖性。每个agent都依赖所有的较上层的agent直至顶层的agent。

146

Agent的表示组件提供了PAC agent的可视行为；其抽象组件维护了构成agent基础的数据模型，并提供对这些数据进行操作的功能；其控制组件连接表示与抽象组件，并提供该agent与其他PAC agent进行通信的功能。

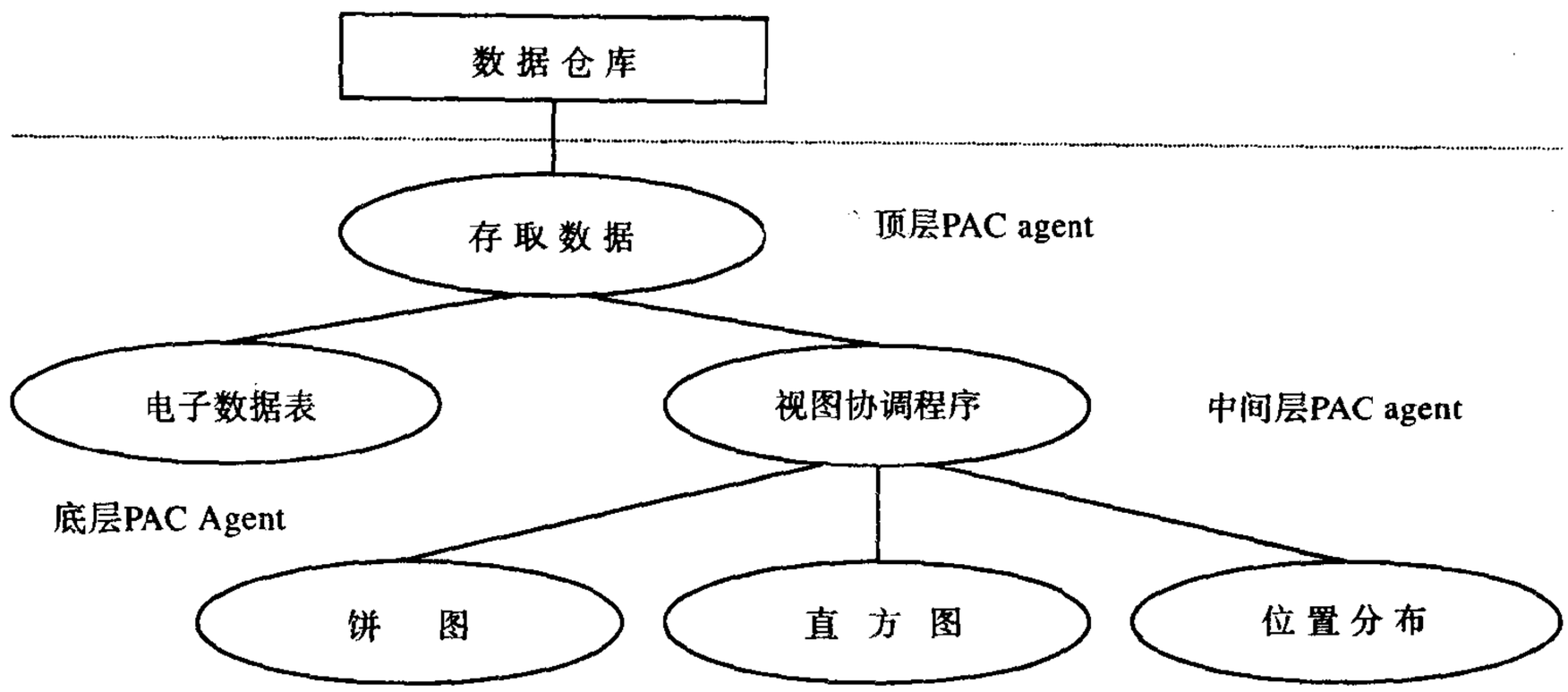
顶层PAC agent提供系统的核心功能。绝大部分其他PAC agent依赖于或操作于这个功能核心。进一步，顶层PAC agent包括不能分配给某一特定子任务的用户接口，如显示应用程序信息的菜单条和对话框。

底层PAC agent表达了独立语义概念的，如电子数据表和图表，系统的用户可以基于这些概念进行工作。底层的agent把这些概念交给用户并支持用户可以执行于这些agent上的所有操作，如放大或移动一个图表。

中层的PAC agent表达了低层agent的组合或低层agent之间的关系。例如，一个中层agent可保持相同数据的几个视图，如建筑用CAD系统中的一个地板平面图和一栋房屋的外部视图。

➡政治选举信息系统定义了一个顶层PAC agent，该agent的可对系统的数据仓库进行存取。数据仓库本身并不是应用程序的一部分。在底层我们规定了4个PAC agent：一个用于输入数据的电子数据表agent，三个用于表示3种数据图表的视图agent。应用程序具有一个中层PAC agent，它协调三个底层视图agent 以保持其一致性。电子数据表agent直接同顶层的PAC agent相连。系统的用户仅同底层agent交互。

147



5. 结构

顶层PAC agent的主要职责是提供软件的全局数据模型。这项工作由顶层agent的抽象组件完成。抽象组件的接口提供操纵数据模型和检索信息的功能。抽象组件中的数据表示是与媒体无关的。例如，在建筑CAD系统中，墙、门、窗等采用能反映其实际大小的厘米或英寸来表示，而不采用用于显示目的的象素来表示。这种媒体独立性支持PAC agent对不同的环境的适应性，使用时，无需对其抽象组件作大的变更。

顶层agent的表示组件通常没有什么职责，它可能包括对整个应用程序而言是共同的一些用户接口元素。在某些系统中，比如网络通信管理程序[TS93]，根本没有顶层表示组件。

顶层PAC agent的控制组件具有三个职责：

- 允许低层agent 使用顶层agent的服务，主要是存取和操纵全局数据模型。来自低层agent的新来的服务请求被提交给抽象组件或表示组件。
- 协调PAC agent的层次。它保存了关于顶层agent和低层agent之间的连接信息。控制组件使用这些信息来确保顶层agent和低层agent之间的正确协作与数据交换。 148
- 保存了用户与系统的交互信息。例如，它可以检查一个由用户触发的有关数据模型的操作是否可执行。此外，可以跟踪调用功能，以提供对功能核心操作的历史或取消/重做服务。

➡在政治选举信息系统的例子中，顶层PAC agent 的抽象组件提供了对底层数据仓库的特殊应用接口。它实现了对选举数据的读写功能。它还实现了操作这些选举数据的所有功能，如计算投影和座位分布的算法。它还进一步包括了维护这些数据的功能，如数据更新和一致性检查。控制组件组织其与低层agent的通信与协作，低层agent指视图协调程序和电子数据表agent。该顶层PAC agent不包括表示组件。 ❑

*底层PAC agent*描绘了应用领域的一个具体语义概念，如网络流量管理系统[TS93]中的一个邮箱或移动机器人系统中的一面墙[Cro85]。这个语义概念可以低至底层的一个简单绘图对象，如一个圆，亦可复杂至柱状图表，该图表总结了系统中的所有数据。

底层PAC agent的表示组件给出了对应于语义概念的一个具体视图，并提供了对用户所能运行的功能的存取。表示组件在内部保存了视图的信息，如它在屏幕上的位置。

底层PAC agent的抽象组件同顶层PAC agent的抽象组件具有相似的职责，既维护具体agent的数据。与顶层PAC agent的抽象组件不同的是：没有其他PAC agent依赖这些数据。

149 底层PAC agent的控制组件维护抽象组件和表示组件的一致性，以避免它们之间的直接依赖关系。它可视为一适配器并执行接口和数据适应功能。

底层PAC agent的控制组件与较高层agent通信以交换事件和数据。输入事件（例如“关闭窗口”请求）被传送到底层agent的表示组件，而输入的数据被传送到底层agent的抽象组件。输出事件和数据，如出错消息，被发送到相关的高层agent。

由底层PAC agent描述的概念，如例子中的直方图和饼图，在用户可操作的最小单元的意义上是不可分的。对于选举信息系统而言，这意味着用户只能把直方图作为一个整体来操作，如通过改变纵坐标轴的缩放比例因子。他们不能改变直方图中的单个条图的大小。

底层PAC agent 并不局限于仅提供应用领域的语义概念。你也可以指定底层agent来实现系统服务。例如，这里可能有一个允许系统与其他应用程序协作并监视这种协作的通信agent。

➡考察政治选举信息系统中的直方图agent。它的抽象组件存放图中的选举数据，并维护特定图表信息，如数据的表示顺序。表示组件负责在窗口中显示直方图，并提供可以运行其上的所有功能，如放大、移动和打印等。控制组件作为表示组件和抽象组件之间的迂回层。控制组件也负责直方图agent与视图协调程序agent的通信。 ❑

*中层PAC agent*可以完成两种不同的任务：合成与协作。例如，当复杂图形中的每个对象都由一个独立的PAC agent表示时，一个中层agent将这些对象分组形成一个合成的图形对象。中层

150 agent定义了一个新的抽象，该抽象同时包含其组件的行为和加到合成对象的新特性。中层agent的第二个作用是维护低层agent间的一致性，例如协调相同数据的多个视图。

抽象组件维护了中层PAC agent特殊数据。表示组件实现了它的用户接口。控制组件具有和底层PAC agent及顶层PAC agent的控制组件相同的职责。

➡政治选举信息系统定义了一个中层PAC agent 。它的表示组件提供了一个允许用户产生选举数据视图（如直方图或饼图）的调色板。抽象组件维护了关于所有当前活动视图的数据，每个当前活动视图都由它自己的底层agent实现。控制组件的主要职责协调所有的从属agent。它把有关顶层agent的数据模型发生的变化的输入通知传送到底层agent，并组织它们的刷新。它也包

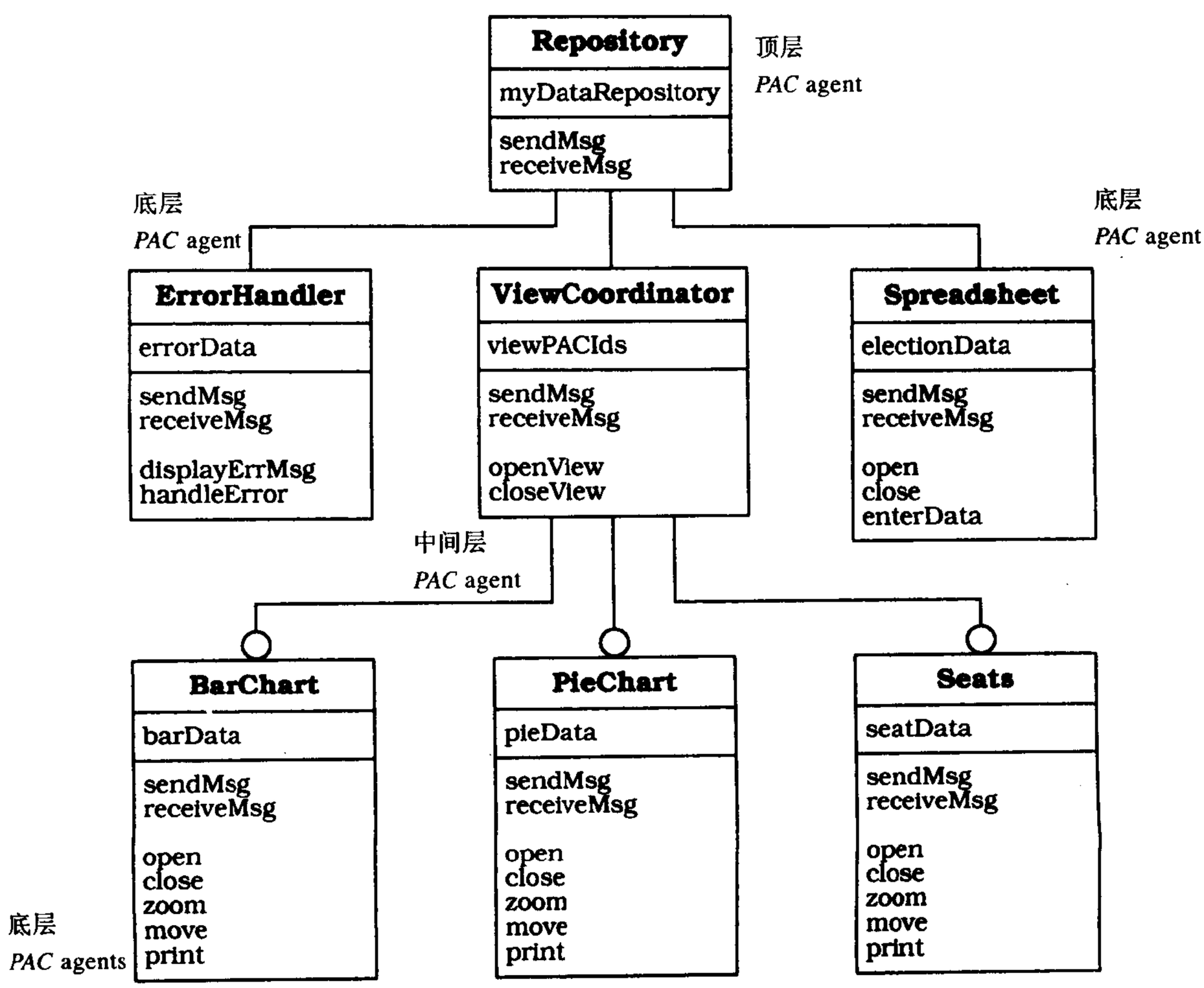
151 括应用户请求创建和删除底层agent的功能。 ❑

政治选举信息系统的如下的OMT图阐明其PAC层次。但是它只列出了控制和协调PAC层次所必须的功能，或者，其他PAC agent或用户可存取的功能。我们通过运用合成消息模式[SC95b]保持PAC agent的接口较小。所有输入服务请求、事件和数据都被一个叫做 `receiveMsg()` 的一个函数处理。该函数解释消息并把它们发送到预期的接收者，该接收者可能是该agent的抽象组件或表示组件或其他的 agent。类似的，函数`sendMsg()`被用来将服务请求、事件和数据打包并发送到其他的agent。另一个方法是提供一个包括agent提供的所有服务的agent特色的接口。我们将在实现小节讨论这两种方法的结论。

类 顶层agent	协作者 • 中间层agent • 底层agent
责任 • 提供系统的功能核心 • 控制PAC结构层	

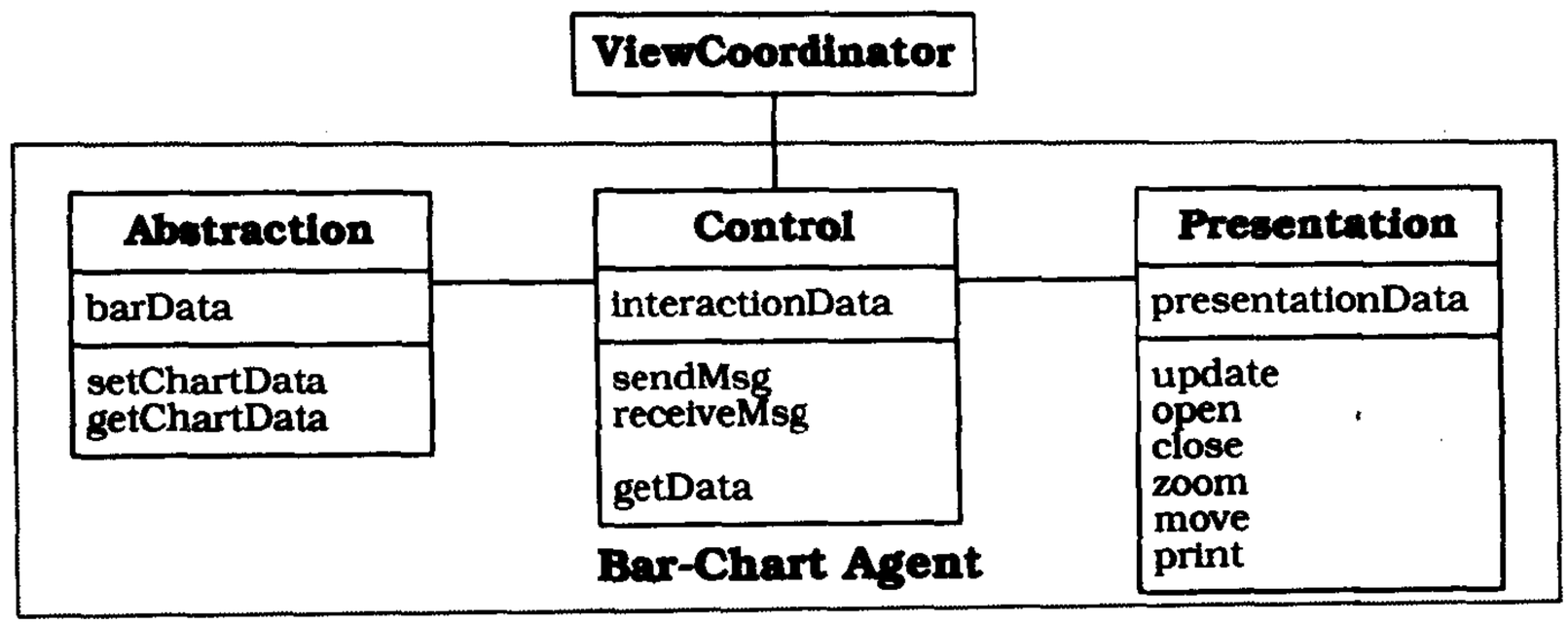
类 中间层agent	协作者 • 顶层agent • 中间层agent • 底层agent
责任 • 协调低层PAC agent • 将低层agent组合成较高层抽象的惟一单元	

类 中间层agent	协作者 • 顶层agent • 中间层agent
责任 • 提供软件或系统服务的具体视图，包括其相关的人机交互	



152

一个PAC agent 的内部结构如下所示，它使用了我们的例子中的直方图agent：



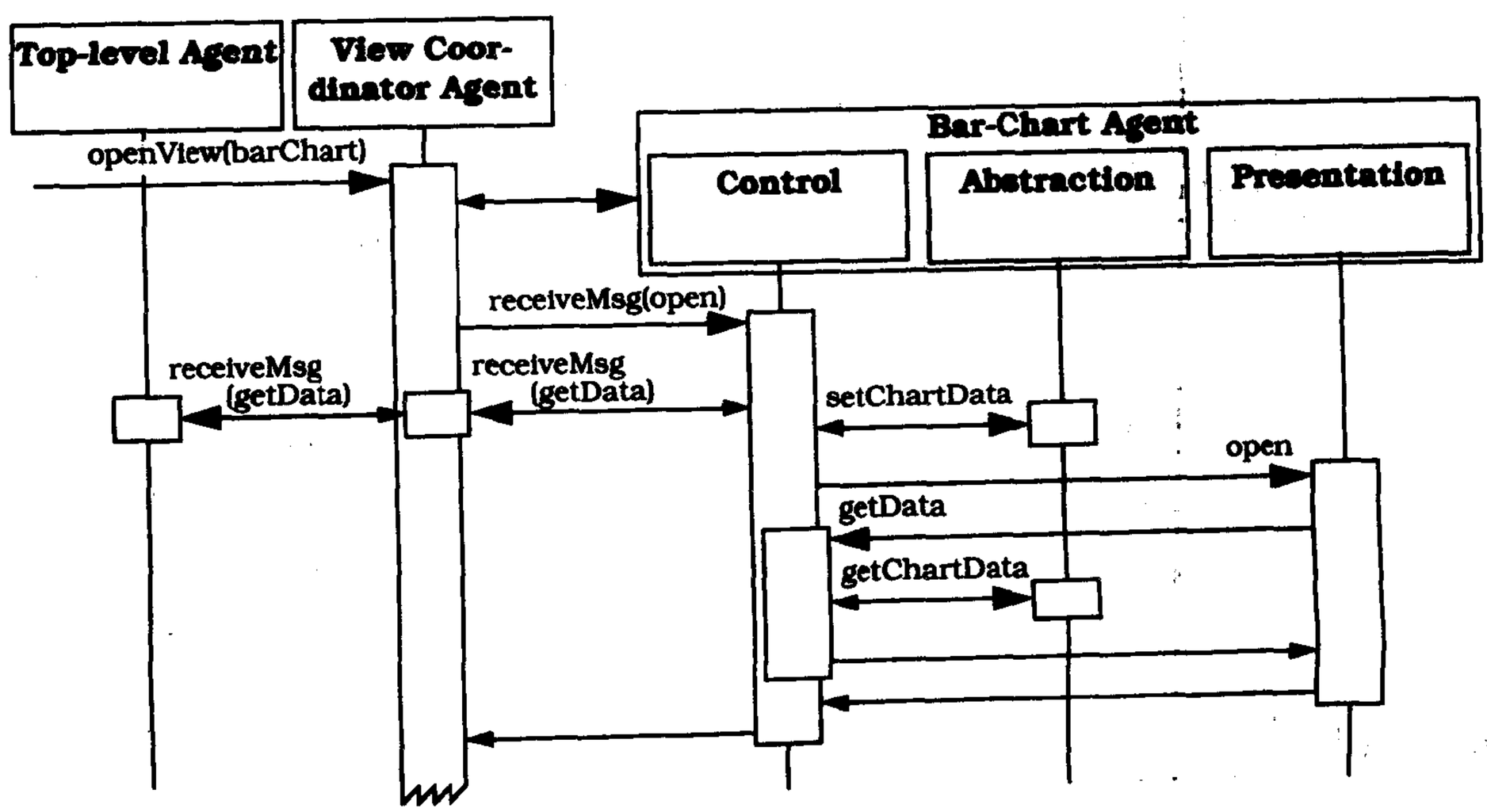
6. 动态特性

我们将以两个场景说明一个PAC体系结构的行为，两个场景均基于选举系统。

场景I 描述了打开选举数据新的直方图视图时不同PAC agent之间的协作。该场景也包括了直方图agent的内部行为的一个更详细的描述。具体分为五个阶段：

- 一个用户要求视图协调程序agent的表示组件打开一个新的直方图。
- 视图协调程序agent的控制组件实例化用户所期望的直方图agent。
- 视图协调程序agent发送一个‘open’事件到新的直方图agent的控制组件。
- 直方图agent的控制组件首先检索来自顶层PAC agent的数据。视图协调程序agent协调底层和顶层agent。返回到直方图agent的数据被存放到它的抽象组件。直方图agent的控制组件调用表示组件显示直方图。
- 表示组件在屏幕上创建一个新的窗口，通过控制组件发出请求检索从抽象组件得到的数据，

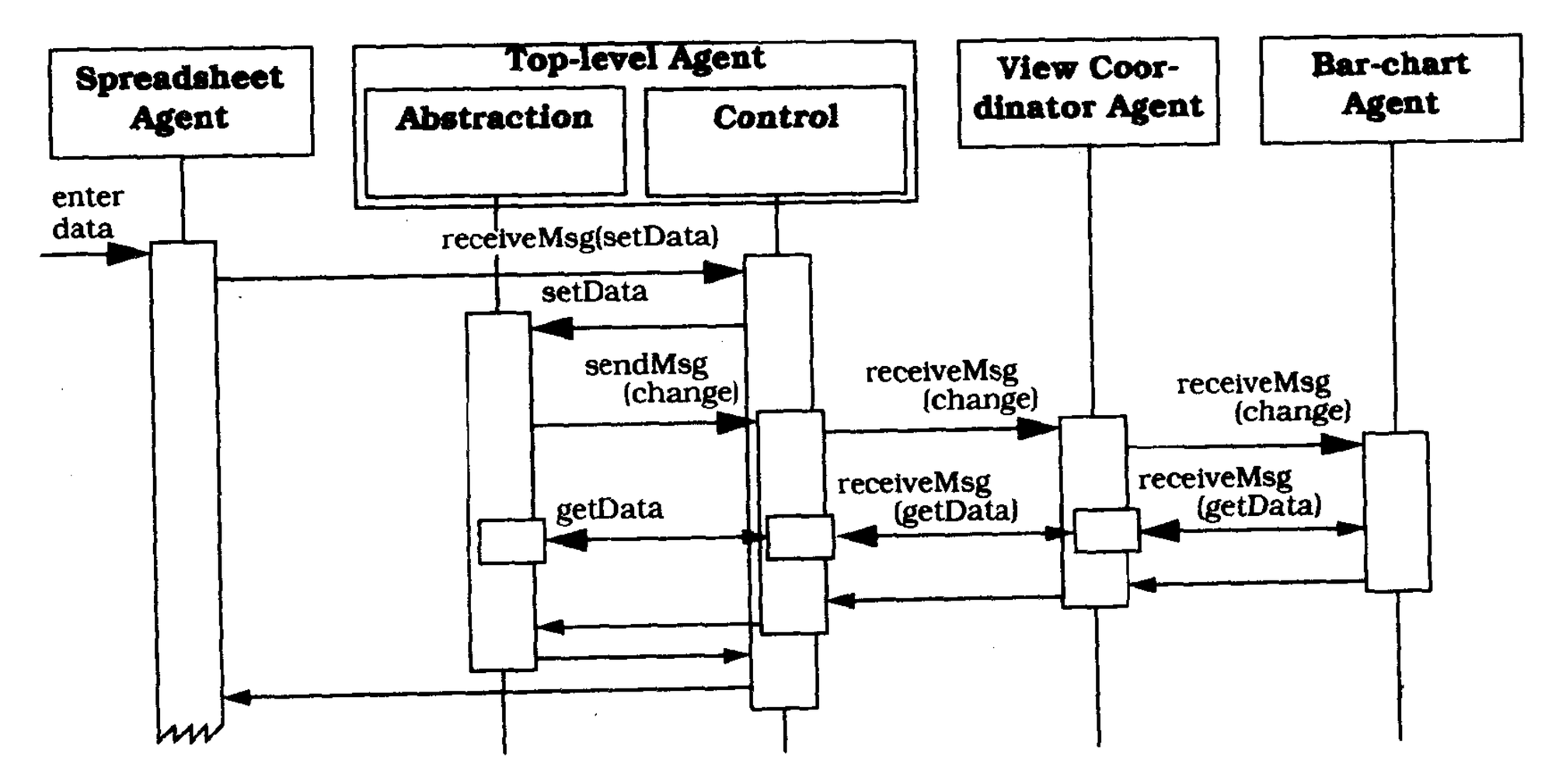
153 并最后将它显示在新窗口中。



显然，这里可以进一步优化，比如在视图协调程序中高速缓存顶层数据，或先调用底层的表示组件再存储数据。但是，在这一点上，我们的重点是解释该模式的基本思想。

场景II 说明了新的选举数据输入后系统的行为，提供了顶层PAC agent的内部行为的周密审视。它分五步：

- 用户向电子数据表录入新数据。电子数据表agent的控制组件将数据输送到顶层agent。
- 顶层agent的控制组件收到数据并通知顶层抽象组件更新相应数据仓库。顶层agent的抽象组件要求其控制组件更新所有依赖于这些新数据的agent。因此，顶层PAC agent的控制组件通知视图协调程序agent。
- 视图协调程序agent的控制组件把更新通知提交给它所负责协调的所有视图PAC agent。 154
- 与以前场景一样，所有视图PAC agent都更新它们的数据并恢复它们展示的图像。



7. 实现

为实现PAC体系结构，执行如下的10个步骤，必要时重复其中的一步或几步。

(1) 定义一个应用模型。分析该问题领域并将该问题影射到适当的软件结构。执行这一步时，不要考虑组件队PAC agent的分布。将精力集中在寻找一个应用领域的正确分解和组织上。为完成这一步，回答如下问题：

- 系统应提供哪些服务？
- 哪些组件可完成这些服务？
- 各组件之间的关系如何？
- 各组件之间的如何协作？
- 各组件对什么数据进行操作？
- 用户如何与软件进行交互？

描述模型时要遵循一种适当的分析方法。

(2) 为组织PAC层次定义一般策略。在这一点上我们还未定义独立的agent，但可以描述组织 155

协作agent层次的一般指南。要遵循的规则之一是最低共同祖先规则。当一组低层agent所依赖的服务或数据是由另一个agent提供时，我们尽量确定这个agent是这一组低层agent所形成子树的根。以此类推，只有提供全局服务的agent才上升到最顶层。例如，选举系统中的所有agent都依赖于中心数据仓库。于是，中心数据仓库的功能由顶层PAC agent提供。如果所有agent中只有一小部分依赖于中心数据仓库，我们应该把这一小部分归结到一个子树中，再定义一个处理数据仓库的agent作为该子树的根。

第二个方面是考虑层次的深度。大多数的PAC体系结构包括好几个中层PAC agent。例如，在移动机器人系统[Cro85]中，低层agent形成环境，此环境又进一步组成工作区——这些内容将在“已知应用”小节移动机器人系统中详细描述。通常，层次越深越能更好地反映一个应用程序到自包含概念的分解。另一方面，深的层次运行时常常效率较低且难于维护。寻找一个系统到PAC agent的适当分解对于能够得到该体系结构的优势是很重要的。

(3) *明确说明低层*PAC agent。确定分析模型中的代表系统功能核心的部分内容。它们很可能是维护系统全局数据模型的组件和直接操作这些数据的组件。同时明确整个应用程序公共用户接口元素，如关于系统信息的菜单条或者对话框。这一步标识的所有组件都将是顶层agent的一部分。

(4) *明确说明底层*PAC agent。确定分析模型的组件，这些组件代表了用户可执行操作或观察表示的系统的最小独立单元。在我们的例子中，这些单元是表示选举数据的各种图表以及用于录入这些数据的电子数据表。

对于每个这样的单元，确定与它们关联的提供人机交互的组件。我们的例子中的直方图需要一个显示图形的窗口和操纵图形的功能，比如放大和打印。每个语义概念（如直方图）及其用户界面组件类图形组成一个独立的底层agent。

156

(5) *明确说明系统服务的底层*PAC agent。通常一个应用程序包括与其最初的主题并不直接相关的额外服务。在我们的例子中，我们定义了一个错误处理器。其他系统可能提供与别的系统通信或为配置用途的服务。每个这样的服务，包括它们的人机交互，可以作为一个独立的底层agent来实现［Baco91］。

(6) *明确说明组成低层*PAC agent*的中间层*PAC agent。通常，几个低层agent一起形成一个高层语义概念，用户可以基于这个概念进行操作。

文献[Cro85]描述的移动机器人系统中，几堵墙、地点和路线PAC agent形成了一个环境。系统的用户可以为环境中的移动机器人指定新的环境和任务。环境被显示在屏幕上，用户可以基于这些显示执行诸如滚动和缩放的操作。因此，一个环境是有其自己的功能和人机交互的一较高层概念的。这样的概念被作为独立的agent来实现。它们提供了它们自己的人机交互，并且基于它们组成的较低层agent来操作。

➡我们的选举例子并没有提供在单独的图、表和电子数据表之上的语义概念。从而我们并没有定义组成其他PAC agent的PAC agent。 ❑

(7) *明确说明与较低层*PAC agent*协作的中间层*PAC agent。许多系统提供同一个语义概念的多个视图。例如，在文本编辑器中你可以找到一个文本文档的“版面设计”和“编辑”视图。当一个视图中的数据发生变化时，其他视图中的数据也必须更新。这样的协作组件，提供了它

们自己的人机交互，当你为分析样式建模时，这些组件你可能已经明确过了。例如菜单输入和相应的回叫函数。选举例子系统中的视图协调程序agent就是这样的一个中间层agent。为了实现与多视图合作的agent，你可以运用视图处理器模式。

注意，视图不是应用程序必须要协作的惟一方面。例如，“已知应用”小节[TS93]中描述的网络交通管理系统实现了一个agent，该agent可以协调电信网络中执行不同并发工作的系统。 157

(8) 从人机交互分离出核心功能。对每个PAC agent，引入了表示组件与抽象组件。提供agent的用户界面的所有组件，如给用户显示的图像，像屏幕坐标、菜单、窗口、对话框等特殊表示数据形成了表示部分。所有维护核心数据或其上操作的组件形成了抽象部分。

通过运用外观模式[GHJV95]可以为一个PAC agent抽象组件和表示组件提供一个统一的接口。控制组件输出了抽象接口和表示接口的其他组件可以使用的那部分。

对某些PAC agent来说，详细的描述其表示或抽象部分可能有些困难。例如，顶层PAC agent经常不提供表示组件。[Baco91]建议顶层表示组件的实现可以作为一个一般的几何关系管理者，来维护低层PAC agent 的表示组件之间的空间关系。你可以运用命令处理器模式来进一步组织表示组件。这样，你就可以调应用户有关延期或提前执行的请求，并提供特别agent撤销/重做服务。

某些抽象组件，尤其是出现在低层agent中的，其操作常常基于其他PAC agent提供的数据。在这种情况下，你可以不去具体指定一个抽象组件，或者设计应用程序满足抽象组件仅作为一个数据缓存来提供服务。在第一种情况下，集中精力实现数据复制的组件，以及保持这些复制品的一致性功能。在后一种情况下，节省了PAC agent间的额外通信量，例如当一个窗口移动后刷新一个视图时。

最后，引入介于抽象组件和表示组件之间控制组件，从而避免了它们之间的直接依赖关系。控制组件是作为一个适配器来实现的[GHJV95]。它通过完成表示组件与抽象组件间的接口和数据自适应将两个组件联系在一起。在这一步中，不要考虑涉及到agent与其他PAC agent之间通信控制组件的部分。这是控制组件的一个不同角色，因而应与作为agent内部的抽象组件和表示组件中间的角色区别开来。 158

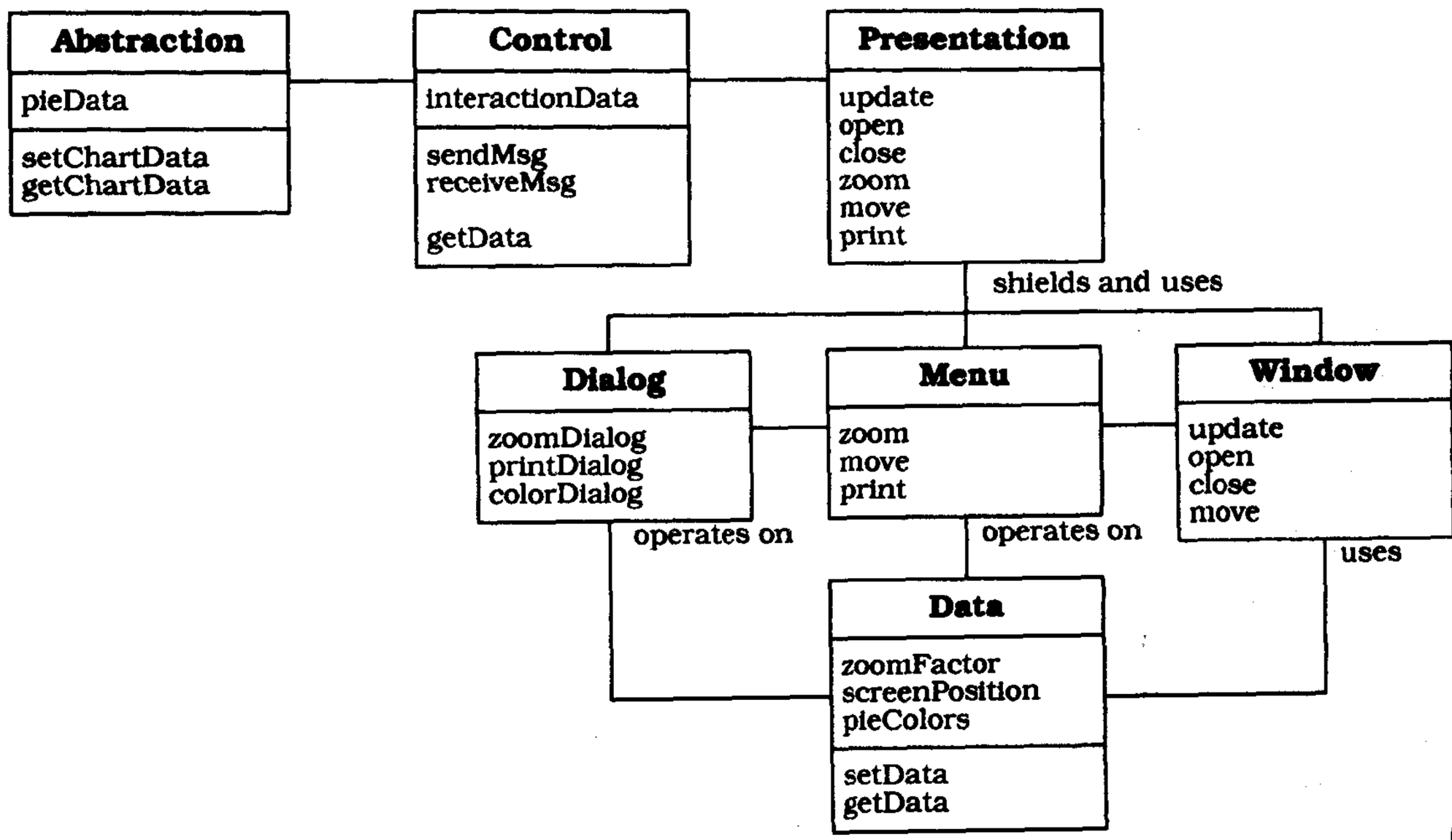
➡为了阐述我们的例子中的这一步，我们细化了例子中的直方图agent，正如在“结构”小节中描述的那样。抽象组件保存了显示在直方图中的选举数据的一个副本。 ❑

表示组件从结构上分成提供窗口、菜单、对话框功能和提供维护特别表示的数据这两部分。为保护客户不受这种结构的影响，我们提供了一个外观[GHJV95]。

饼图PAC agent的控制组件是简单的。它只把来自表示组件的读数据请求递送到抽象组件。与高层agent的通信在下一步处理。 159

(9) 提供外部接口。为了与其他agent协作，每个PAC agent发送并接受事件和数据。实现这个功能是控制组件的一部分。

在一个agent之内，输入的事件或数据被递送到期望的接受者。接受者可能是agent的抽象组件或表示组件，但也可能是较低层或较高层的agent。例如，我们的信息系统的视图协调程序agent定期接受来自顶层PAC agent变化通知并把它们递送到视图agent。它也接受来自较低层agent的递送给顶层agent的请求。换句话说，控制组件是中介者——你可以使用中介者模式



[GHJV95]来实现这个角色。

实现与其他agent通信的一种方法是运用组合消息模式[SC95b]。这就使得一个agent的接口保持的较小。它同样允许agent独立于其他agent的特定接口，独立于特定的数据格式、列集、散集、分割和重聚集方法。但是，运用组合消息模式要求控制组件解释输入的消息。它决定用它们作什么：是调用抽象组件或表示组件呢？还是把这个消息传递给另一个agent。这个功能通常非常复杂且难以实现。

第二种可选方案是提供一个公共接口，该公共接口将每个agent的服务表示为一独立的功能。当这些功能被调用时它们“知道”如何去处理事件和数据。与组合消息求解方案相比，这种方案减少了控制组件的内在复杂性，但它引入了agent之间的额外依赖性，这些agent依赖其他agent的特定接口。此外，在这种方法中一个agent的接口可以“迅速扩张”。例如，一个中间层agent必须提供顶层agent的所有功能，这些功能被相关的低层agent调用。反过来也一样，中间层agent必须提供与其相关的低层agent的所有服务，这些服务被顶层agent调用。结果使一个agent的接口变得复杂且难以维护。

通过使用注册功能可以将一个PAC agent以一种灵活和动态的方式与其他PAC agent相联，正如在出版者-订阅者模式所介绍的那样。例如，如果在我们的选举系统中创建了的直方图agent的一个新的实例，它将以视图协调者agent的名义被动态地注册。

160

如果一个PAC agent依靠由其他PAC agent维护的数据或信息，那么你应该提供一个变更-传播机制。这种机制应该包括所有的agent和层次图的所有等级以及两个方向上的工作。当一个agent内的数据发生变化时，它的抽象组件就开始了变化传播。控制组件将改变通知传送给所有相关的PAC agent，但经常也传送到表示组件。来自其他agent的变化通知引起抽象组件和表示组件更新它们的内部状态。实现这种变更-传播机制的方法之一是使用出版者-订阅者模式。另

一种方法是把改变传播同发送和接收事件、消息和数据的一般功能结合起来。参见下面的例子代码。

对所有PAC agent而言，这些通信和协作功能的接口都是相同的。这样做支持了PAC agent重新配置和重用以及加入新的PAC agent对应用程序的扩展。

➡在我们的选举例子中，视图协调者agent控制组件提供了如下的接口：

```
enum ViewKind { barChart, pieChart, seats };
    // type of available views of election data
class DataSetInterface { /* ... */ };
    // Common interface for datasets, messages, and
    // events, according to the specifications of the
    // Composite Message pattern [SC95b]
class PACId { /* ... */ };
    // Provides a handle to a PAC agent
class VCControl {
    // Data member specifications
    PACId        parent;  // higher-level agent
    List<PACId>  children;// lower-level agents
    // More data member specifications ...
private:
    void attach(PACId agent, parentAgent = 0);
    void detach(PACId agent);
        // Registration functionality for connecting
        // dependent view agents and the top-level agent
        // with the view coordinator agent.
    DataSetInterface sendMsg(DataSetInterface data);
        // Sending events, messages, or data to other PAC
        // agents including change notifications
    void openView(ViewKind kind);
    void closeView(PACId agent);
        // Opening and closing views including
        // creation, registration,and deletion
        // of bottom-level agents displaying charts
public:
    DataSetInterface receiveMsg(DataSetInterface data);
        // Receiving events, messages, or data from other
        // PAC agents including change notifications
};
```

161

函数sendMsg()和receiveMsg()返回带有消息发送和接收的应答的对象。 ❑

(10) *将各层次连接起来*。实现了单个的PAC agent之后，你就可以建立最后的PAC层次了。把每个PAC agent同它直接协作的较低层PAC agent连接起来。

提供动态地创建和删除低层PAC agent的PAC agent，而这些低层PAC agent具有动态扩展或减少PAC层次的功能。例如，在我们的信息系统中，如果用户要打开一个特定视图，则视图协调者agent创建一个新的视图PAC agent，在用户关闭显示这个视图的窗口时，删除这个PAC agent。

9. 变体

许多大型的应用程序——特别是交互式应用程序——都是多用户系统。因此，在设计这样的软件系统时多任务成为一个要考虑的主要方面。如下两个PAC变体针对这一强制条件。

*作为主动对象的*PAC agent。许多应用程序，特别是交互式应用程序，都得益于多线程。移

动机器人系统[Cro85]是一个多线程PAC体系结构的例子。每一个PAC agent用一个主动对象来实现，该主动对象寄身于它自己的控制线程中。像主动对象（Active Object）和半同步/半异步（Half-Sync/Half-Async）[Sch95]的设计模式有助于你实现这样的体系结构。

PAC agent作为进程。为了支持位于不同进程或远程机器上的PAC agent，使用代理来本地化表示这些PAC agent，以避免直接依赖于它们的物理位置。使用发送者-接收者模式或客户-分配器-服务器模式来实现PAC agent之间的进程间通信(IPC)。

162

由于IPC效率较低，你也可以考虑组织不同进程内的PAC层次的相干子树。于是，在执行一特定任务当中紧密协作的agent位于同一进程内。PAC Agent间的IPC被最小化，且只有协作不同子树，以及访问顶层PAC agent的服务是必要的。

10. 已知应用

网络通信量管理。该系统在[TS93]中已描述过。它展示了电信网络的通信量。每15分钟所有被监视的交换单元都要把它们当前的通信量情况报告给一个控制点，数据在控制点处存放，分析和显示。这有助于确定潜在的瓶颈和预通信量超载。系统包括以下功能：

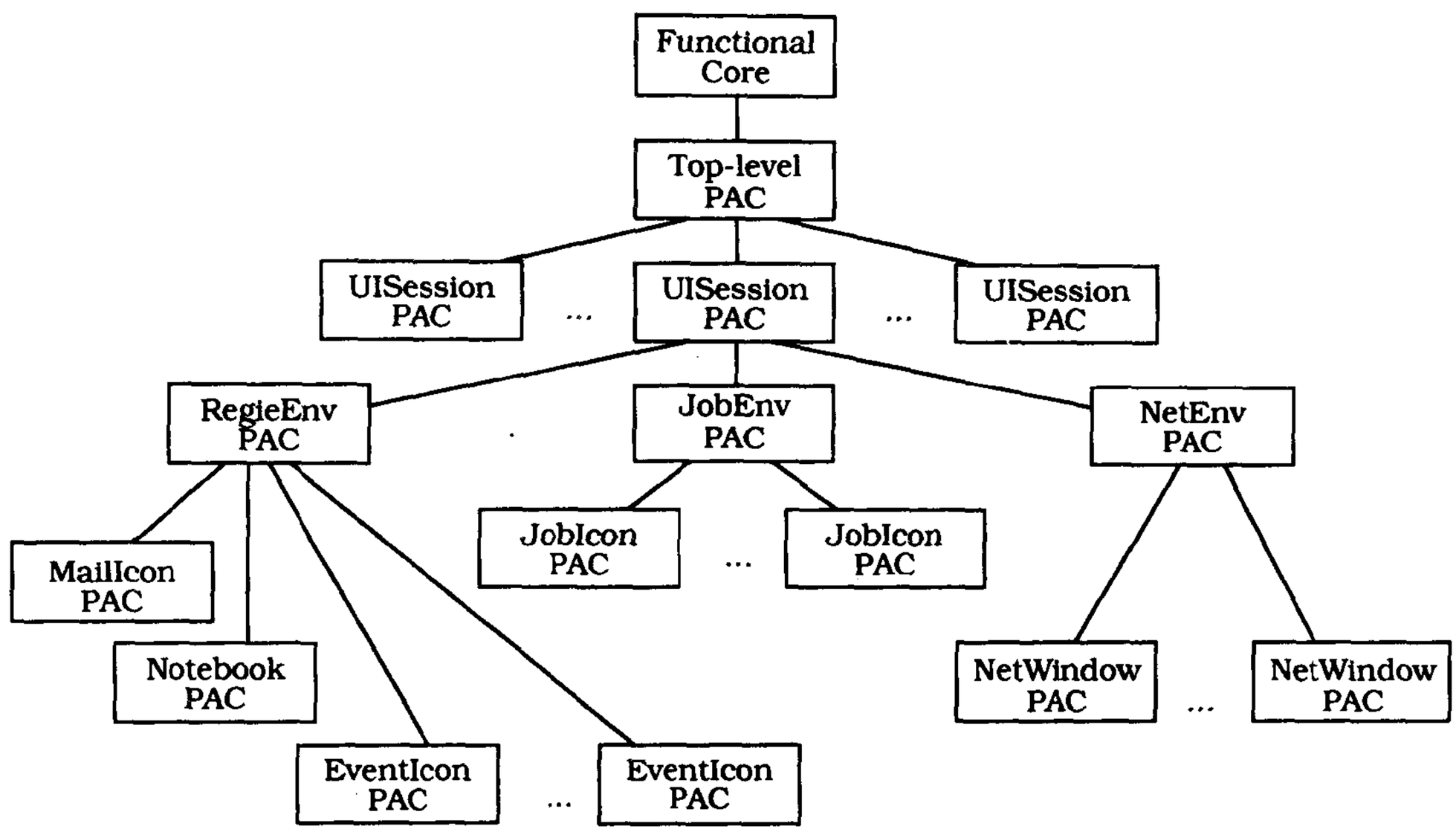
- 从交换单元收集通信量数据。
- 阈值检查与溢出例外的产生。
- 网络例外的日志与跟踪。
- 电信流量和网络例外的可视化。
- 显示整个网络的各种用户可配置视图。
- 通信量数据的统计评估。
- 存取历史通信量数据。
- 系统管理与配置。

系统的设计与实现遵循了表示-抽象-控制模式。系统的每一个功能可通过它自己的底层PAC agent 所表示。这里有专门的agent负责网络的每一个视图，负责系统可以完成的每一项工作，负责系统提供的额外服务，比如邮件或帮助。三个中间层PAC agent协调这些底层agent，其中每个对应如下三种应用功能之一：视图、工作和额外服务。在图中，它们分别由三个agents、NetEnv、JobEnv、RegieEnv来表示。另外一个中间层PAC agent组织用户话路。顶层PAC agent协调单个的用户话路，并与系统的功能核心通信。功能核心的实现独立于PAC层次，也许是因为它合并了遗产软件。系统的PAC agent层次是动态的。例如，如果一个用户开始了一个新的话路，则创建一个相应的 UISession agent并在顶层agent进行注册。在该话路的最后删除该agent。

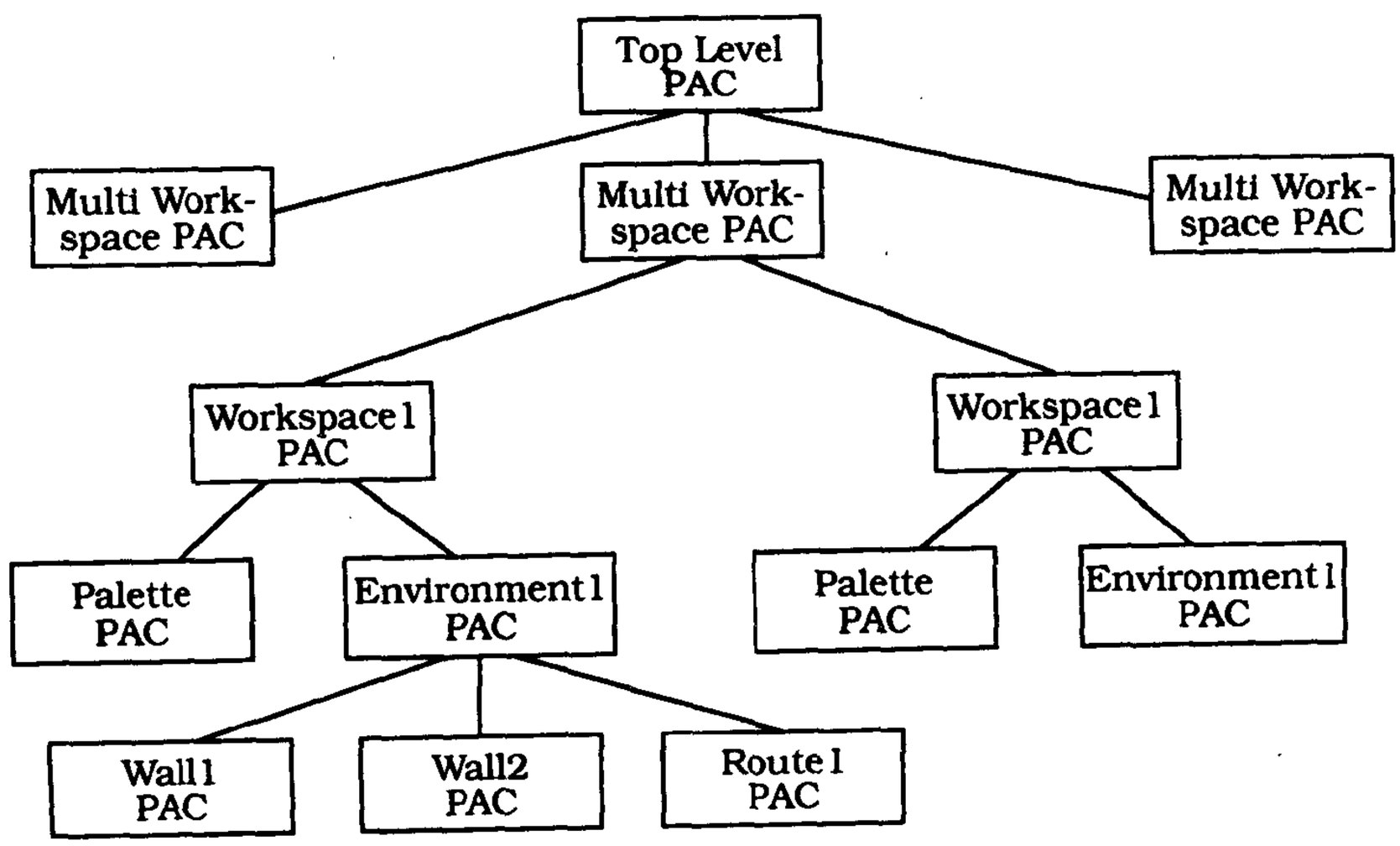
163

移动机器人。这个系统[Cro85]允许一个操作员同一个移动机器人交互，该机器人航行在一个由墙、器材和人（入侵者或偶然的受害者）组成的封闭且危险的环境中。机器人航行时使用自己的传感器和来自系统操作员的信息。软件允许的操作员从事以下活动：

- 给机器人提供它所工作的环境、环境中的位置以及位置间的路线的描述。
- 随后修改环境。
- 详细描述机器人的任务。
- 控制任务的执行。
- 观察任务的进展。



环境中的每面墙、每条路线、每个位置都由它自己的底层PAC agent来表示。这些agent共同来构建可视化环境。环境由中间层PAC agent来表示。它们控制构成成分墙、路线、位置PAC agent。用户可以在一个环境上使用控制在“调色板”PAC agent中实现，这里的agent也就位于层次结构的底层。环境PAC agent和调色板PAC agent形成机器人的一个工作区。这个工作区由它自己的一个中间层PAC agent来表示。为了支持同一环境的多个视图，一个多工作区PAC agent协调同一工作区的多个视图。位于层次结构的顶层的PAC agent封装了应用程序的功能核心，它是一个用于操纵和控制机器人的基于规则的智能监督者。 164



11. 效果

表示-抽象-控制体系结构模式有如下**优点**：

事务分离。应用领域的不同语义概念用独立的agent分别表示。每个agent维护它自己的状态和数据，与其他PAC agent合作，但又独立于其他的PAC agent。单个PAC agent要提供它自己的人机交互。这允许为应用程序的每一个语义概念或任务开发特定的独立于其他语义概念或任务的数据模型和用户接口。

支持变化和扩展。一个PAC agent的表示组件或抽象组件的变化不影响系统中的其他agent。这允许你独立地修改或调整一个PAC agent所基于的数据模型，或改变它的用户界面，比如，从命令外壳改为菜单和对话框形式。

165

新的agent可以容易地集成到一个现存的PAC体系结构中而无需对现存的PAC agent做太多的改变。所有PAC agent通过一个预先定义的接口核心进行通信。此外，现存的agent可以动态地注册新的PAC agent以确保通信和合作。例如，往政治选举信息系统中加入一个新的视图PAC agent，我们只需以适当的调色板手段（palette field）扩展视图合作者PAC agent中的表示，该调色板手段允许用户创建新的视图。处理这个新PAC agent，将它以新的合作者PAC agent身份注册，将变化和事件向它传播，所有这些功能均为随时可用的。

支持多任务。PAC agent可以容易地分配到不同的线程、进程或机器上。扩展PAC agent的适当的IPC功能只影响它的控制组件。

多任务也方便了多用户应用程序。例如，在我们的信息系统中，当数据录入人员用新的选举数据更新了数据库之后新闻广播员可以给出最新估计。对共享数据仓库或其控制组件去考虑串行化或同步，都是必要的。

这个模式的**不足**如下所示：

增加了系统复杂性。一个应用程序中的每个语义概念都作为它自己的PAC agent来实现可能导致一个复杂的系统结构。例如，如果一个图形编辑器中的每一个图形对象（如一个圆或一个矩形）均实现为其自己的PAC agent，那么系统将淹没在agent的汪洋大海。Agent也必须协作和受控，从而需要额外的协作agent。仔细考虑设计粒度的层次，何处停止将agent细化到越来越多的底层agent。

复杂的控制组件。在一个PAC系统中，控制组件是一个agent的抽象组件和表示组件之间的通信媒介，也是不同的PAC agent之间的通信媒介。因此，控制组件实现的质量对于agent之间的有效协作，以至对于系统体系结构的整体质量都是至关重要的。控制组件的个体角色应该明显地相互区分开来。这些角色的实现应该不依赖于其他agent的特定细节，比如它们具体的名字或一个分布系统的物理位置。控制组件的接口应独立于内部细节，以保证一个agent协作者不依赖于它的表示组件或抽象组件的特定接口。实现必要的接口和数据改造是控制组件的责任。

166

效率。PAC agent之间的通信开销可能会影响系统效率。例如，如果一个底层agent从顶层agent检索数据，那么沿PAC层次的从下而上路径上的所有中间层agent都要参与数据交换。如果agent是分布的，那么数据传递也需要IPC，还需要数据的列表，散集，分段存储和重新聚集。

这里存在严重的潜在陷阱。我们将在下面的讨论中考虑何时该使用何时不该使用表示-抽象-控制模式。

可应用性。应用程序的原子语义概念越小，那么它们用户接口的相似性就越多，该模式可用性方面就越差。例如，一个图形编辑器，其中文档中的中每一单独对象都由它自己的PAC agent来表示，将可能导致一个难以维护的复杂的细粒度结构。另一方面，如果原子语义概念相当大，且需要它们自己的人机交互，PAC提供一个可维护和可扩展的结构，而且在不同系统任务之间事务明确分离。

参见

模型-视图-控制器模式（Model-View-Controller，MVC）也把软件系统的功能核心与信息显示和用户输入处理区分开来。但是，MVC定义它的控制器作为一个实体负责接受用户输入并把它转化为内部语义。这意味着MVC有效地把用户存可取部分（PAC中的表示部分）划分成视图和控制。它缺少介于其间的控制组件。进一步，MVC没有把一个系统的自力更生的子任务划分成为合作但松散耦合的agent。 167

致谢

PAC最初由Joelle Coutaz在[Cou87]中描述过。基于PAC实现的系统之一是移动机器人应用程序[Cro85]。实现PAC的进一步有价值的指导可以参考[BaCo91]和[CNS95]。

我们感谢Joelle Coutaz和Laurence Nigay与我们进行的富有成果的讨论及有价值的建议，这些讨论与建议对我们形成这些模式的描述很有帮助。Steve Berczuk、Brian Foote、Ralph Johnson、Tim Ottinger、David E.DeLano和Linda Rising仔细评审了早期的PAC版本并为我们提供了详细的用于改进的反馈意见。 168

2.5 适应性系统

系统随时间演化——添加新的功能和更改现有的服务。它们必须支持新版本的操作系统、用户接口平台或第三方组件和库。适应新的标准或硬件平台也是必要的。在系统设计和实现过程中，客户可能要求新的特征，这种要求通常都很迫切并且此时开发进程已处于后期阶段。还可能需要随客户不同而提供不同的服务。

因此，应对变化的设计成为我们在描述一个软件系统的体系结构时应该主要考虑的方面。一个应用程序首先应支持对先前版本的自修改和自扩展。变化不应影响核心功能或关键设计抽象，否则系统将难以维护且适应不断变化的需求的成本太高。

本节描述了两个在设计期间应对变化的模式：

- 微核模式应用于必须能够适应变更系统需求的软件系统。这种模式把最小功能核心同扩展功能和特定客户部分分离开来。微核也可作为插入到这些扩展中并协调其协作的套接字。
- 映像模式为动态地改变软件系统的结构和行为提供了一种机制。它支持诸如类型结构和函数调用机制等基本方面的修改。在这种模式中，一个应用程序可分成两部分。一个元层次提供所选系统属性的相关信息并使软件含自述信息。一个基本层次包括应用程序逻辑。它的实现建立在元层次之上。改变保存在元层次上的信息会影响其后的基本层次上的行为。

开发微核模式来支持小的、有效的和可移植的操作系统的设计，并用新服务来支持其扩展。 169

微核为几个现代操作系统，如Chorus[Cho90]、Mach[Tan92]和Windows NT[Cus93]充当基本体系结构。微核模式提供了一个“即插即用”的软件环境，使得你容易地连接扩展部分并把它们同系统的核心服务集成在一起。采用特定组件来封装平台附属物。尽管现今在操作系统领域之外只有几种应用程序运用这种模式的原理，但是我们相信微核结构非常诱人，它非常适合用在下面这种系统上，即要求对不同平台具有高度适应性并能满足客户定制需求的系统。

映像模式采取不同的方法。使用映像模式来设计一个系统，使这个系统维护了自身的信息并且使用这种信息来保持可变性和可扩展性。特别地，一个映像系统在实现方面处于开放状态，以支持特定的结构和行为方面（如类型结构、函数调用机制或特别服务的实现）的改造、变更和扩展。映像的原理得到了广泛的支持，这种支持来自不同的程序设计语言（如CLOS[Kee89]和Smalltalk[GR83]），来自多种操作系统（如Apertos[Yok92]），甚至来自大型的工业应用程序。诸如CORBA[OMG92]和微软的OLE[Bro94]等现代平台也使用了映像模式的某些原理。

170

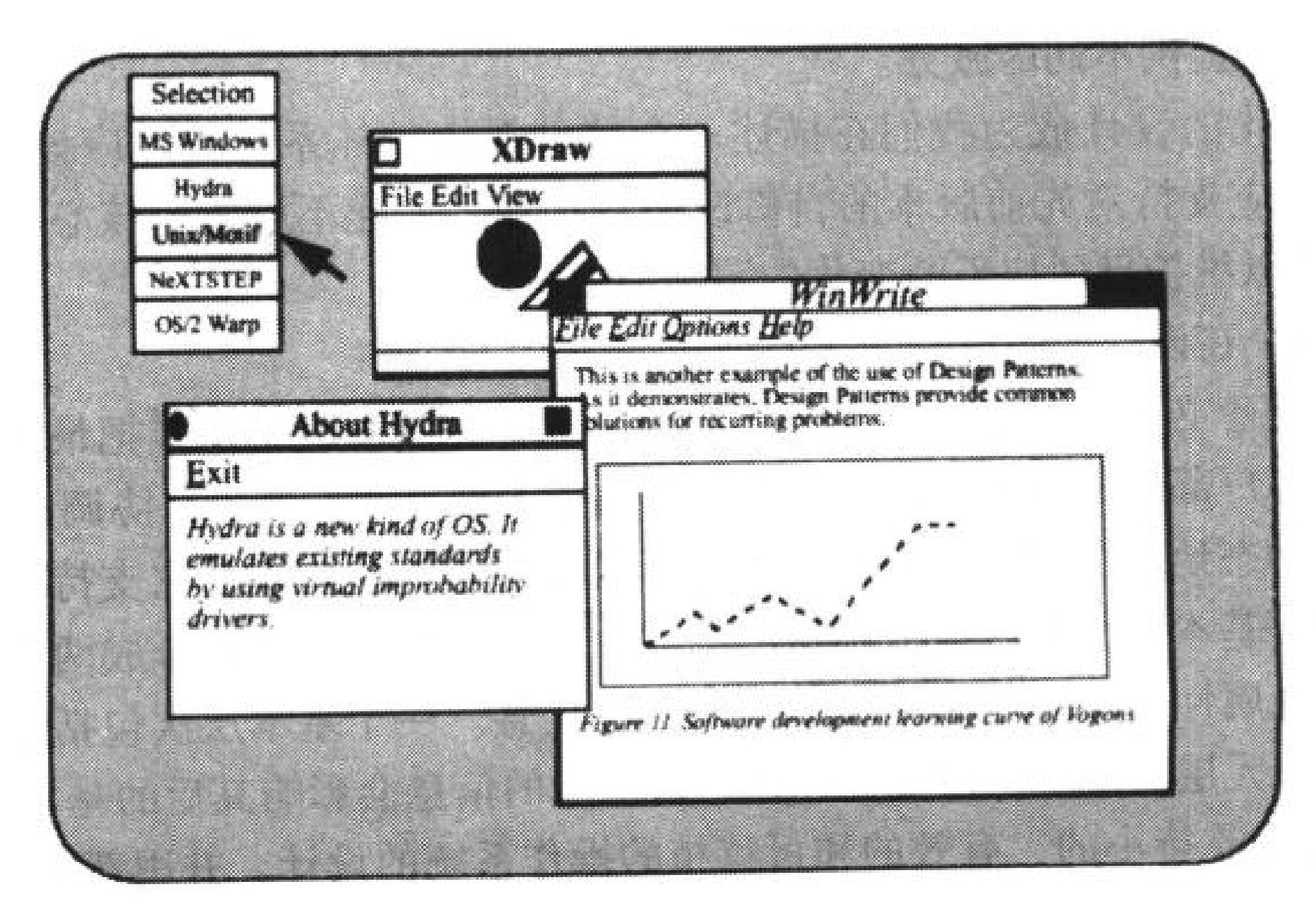
2.5.1 微核

*微核（Microkernel）*体系结构模式应用于必须能够适应变更系统需求的软件系统。这种模式把最小功能核心同扩展功能和特定客户部分分离开来。微核也可作为插入到这些扩展中并协调其协作的套接字。

1. 例子

设想我们想为名叫Hydra的台式电脑开发一种新型操作系统。我们的开发小组已经详细描述了要达到的设计目标的一个列表。一个需求是这个创新的操作系统必须很容易地移植到相关的硬件平台上，必须能够容易适应未来的发展。它也必须能够运行在诸如NeXTSTEP、Microsoft Windows和UNIX系统等其他流行的操作系统等上所编写的应用程序。一个用户应该能够在开始一个应用程序之前从一弹出菜单中选择它所需要的操作系统。Hydra将在它的主窗口中显示所有当前运行的应用程序：

171



为了模拟这些操作系统，Hydra将集成实现了Hydra的功能核心的特定视图的特殊服务器。一个视图表示建立在核心功能顶部的抽象层。由一组服务器模拟微软Windows的过程即是这种视图的例子。由于诸如多媒体、基于笔的计算和万维网等新技术可能变得越来越重要，所以Hydra设计目标时应考虑它们的易集成性，同时要考虑总体功能的适应性、进化和提高。

2. 语境

使用类似的编程接口的几个应用程序的开发建立在相同核心功能上。

3. 问题

为需要满足下述要求的应用领域开发软件不是一个简单的任务，要求是能适应宽谱的相似标准和技术。众所周知的例子是诸如操作系统和图形用户界面[⊖]这样的应用程序平台。这样的系统通常有一个较长的生存期，有时达10年或10年以上。在这么长的时间里，新的技术出现了，旧的技术改变了。因此，在设计这样的系统时下面的要求需要特别考虑：

- 应用程序平台必须适应连续的硬件和软件进化。
- 应用程序平台应该是可移植的、可扩展的和可适应的使之易于集成不断出现的技术。

这种应用程序平台的成功进一步依赖于它们运行用现有标准编写的应用程序的能力。为了支持宽范围的应用，需要基础应用程序平台功能的多个视图。换句话说，诸如一个操作系统或一个数据库的应用程序平台也应该可以模拟其他的属于相同应用领域的应用程序平台。

➥例如，设计Hydra来运行原本为流行的操作系统（如微软 Windows或OS/2 Warp）开发的应用程序。 ❑

这又导致以下的强制条件：

- 在你的领域中的应用程序需要支持不同的但又相似的应用程序平台。
- 应用程序可以按类分组，每组以不同方式使用相同的功能核心，需要基于应用程序平台模拟现有标准。

172

提供了一个领域的功能核心的一个应用程序平台是其客户机的专有资源。为了避免性能问题和保证可伸缩性，你的解决方案必须考虑一个额外的强制条件：

- 应用程序平台的功能核心应该分离出一个占有最小的存储空间的组件和消耗的处理能力仅尽可能地小的服务。

4. 解决方法

将应用程序平台上的基本服务封装到一个微核组件之中。该微核包含使独立进程中运行的其他组件之间相互通信的功能。它也负责维护诸如文件或进程那样的系统范围内的资源。此外，它提供了使其他的组件存取其功能的接口。

不得不增加其规模或复杂性的又不能在微核内实现的核心功能应将其分到内部服务器中。

外部服务器实现了它们自己的基础微核视图。为了构造这个视图，他们使用通过微核的接口获得的机制。每个外部服务器是独立进程，该进程自身表示了一个应用程序平台。因此，一

⊖ 在现存的文献中，微核系统主要用来描述与操作系统设计相关的内容。但是，我们相信这种模式也可用于其他领域。例如，金融应用或数据库系统[Woo96]。由于使用微核实现操作系统需要用到许多知识，因此我们的例子将主要集中于这个特殊领域。

个微核系统可以被视为一个集成了其他应用程序平台的应用程序平台。

173 客户机通过使用由微核提供的通信能力与外部服务器通信。

5. 结构

微核模式定义了五种参与组件：

- 内部服务器（*Internal servers*）
- 外部服务器（*External servers*）
- 适配器（*Adapters*）
- 客户机（*Clients*）
- 微核（*Microkernel*）

微核表示了该模式的主要组件。它实现了诸如通信手段或资源处理那样的主要服务。其他的组件全部或部分建立在这样的基本服务之上。它们通过使用一个或多个由透明的微核功能组成的接口来间接完成这项工作。

许多系统特定的附属物被封装在微核内。例如，大多数的硬件相关部分对其他的参与者来说是隐藏的。微核的客户仅能看到基础应用领域和平台特性的特殊视图。

微核也负责维护像进程或文件那样的系统资源。它控制和协调对这些资源的访问。

总之，微核实现了被我们称为机制（*mechanism*）的原子服务。以这些机制为基本出发可以

174 构造更加复杂的、称为策略（*policy*）的功能。

类 微核	协作者 • 内部服务器
责任 • 提供核心机制 • 提供通信能力 • 封装系统从属物 • 管理和控制资源	

➡在Hydra中，我们要支持其他操作系统中的UNIX系统V和OS/2 Warp。实现Hydra的进程模型时我们面临一个问题。在UNIX中通过克隆一个现存的进程，复制整个地址空间，来实现诸如创建一个新的子进程的系统调用。OS/2 Warp以完全不同的方式处理进程的创建，其中它并不复制其父进程的地址空间。换句话说，OS/2 Warp和NUIX为进程提供不同策略。因此设计Hydra来提供诸如创建进程的机制和克隆现存进程空间的机制的基本服务。实现UNIX系统V的进程模型和实现OS/2 Warp的进程模型可以以各种方式结合起来。 ❑

一个内部服务器（也称为一个子系统）扩展了微核所提供的功能。它表示了提供附加功能的一个独立组件。微核通过服务请求调用内部服务器的功能。因此，内部服务器可以封装某些基础的硬件或软件系统上的附属物。例如，支持特定图形卡的设备驱动程序就是内部服务器的好的候选。

类 内部服务器	协作者 • 微核
责任 • 实现附加服务 • 封装某些系统 • 特性	

设计目标之一应该是使微核尽可能小以减少存储需求。另一个目标是提供快速执行机制以减少服务执行时间。因此，额外和更复杂的服务由内部服务器实现，必要时微核会激活或装载它们。可以认为内部服务器是微核的扩展。注意，内部服务器只能由微核组件访问。 175

一个外部服务器（也称为个性化服务器）是使用微核实现以应用领域为基础的它自己的视图的一个组件。正如前面已提过的，一个视图表示建立在原子服务顶部的一个抽象层，这些原子服务由微核提供。不同的外部服务器为特定应用领域实现不同的策略。

外部服务器通过像微核那样的输出接口来展现它的功能。每个这样的外部服务器运行在一个独立的进程上。它通过使用微核提供的通信手段接收来自客户机应用程序的服务请求，解释这些请求，执行相应的服务，并把结果返回到它的客户机。服务的实现依赖于微核机制，于是外部服务器需要访问微核的编程接口。

类 外部服务器	协作者 • 微核
责任 • 为其客户机提供编程接口	

➡在Hydra中，我们要实现一个OS/2 Warp外部服务器和UNIX系统V外部服务器。这些服务器都使用基础微核机制来实现OS/2 Warp和UNIX系统V系统调用完全集。 ❑

一个客户机是一个同外部服务器精确关联的应用程序。它只存取外部服务器提供的编程接口。

如果一个客户机需要直接存取它的外部服务器的接口，那么问题就出现了。每个客户机不得不使用可用的通信手段与外部服务器相互作用。

因此，每个与外部服务器的通信都必须硬化编码到客户机代码之中。但是，客户机与服务 176

器如此紧密偶合将导致很多缺点：

- 这样的系统不能很好地支持可变更性。
- 如果外部服务器模拟现存的应用程序平台，为这些平台开发的客户机应用程序不加以修改将不能运行。

于是，我们引入客户机与其外部服务器之间的接口以避免客户机的直接依赖性。适配器（*adapter*）——也称仿真器（*emulator*）——代表了客户机及其外部服务器的接口，允许客户机以一种可移植性的方式访问它们的外部服务器的服务。它们是客户机地址空间的一部分。如果外部服务器实现了一个现存的应用程序平台，那么相应的适配器模仿了这个平台的编程接口。因此，为这个仿真平台编写的客户机程序可以不加修改地编译和运行。适配器也使客户机程序避开了微核的具体实现细节。

一旦客户机向外部服务器请求一个服务时，适配器的任务是把调用提交给适当的服务器。为了达到这个目的，适配器使用了微核提供的通信服务。

类 客户机	协作者 • 适配器
责任 • 表示一个应用	

类 适配器	协作者 • 外部服务器 • 微核
责任 • 隐藏系统从属物，如来自客户机的通信手段 • 请求代表客户机的外部服务器的方法	

➥由于被一个适配器封装了，因此与OS/2 Warp的外部服务器相关联的Hydra 客户并不知道
177 它是运行在一个本地OS/2 Warp系统上还是运行在提供了OS/2 Warp外部服务器的一个微核系统上。它只是像以前那样使用OS/2系统调用。幕后发生的事被适配器隐藏了。❑

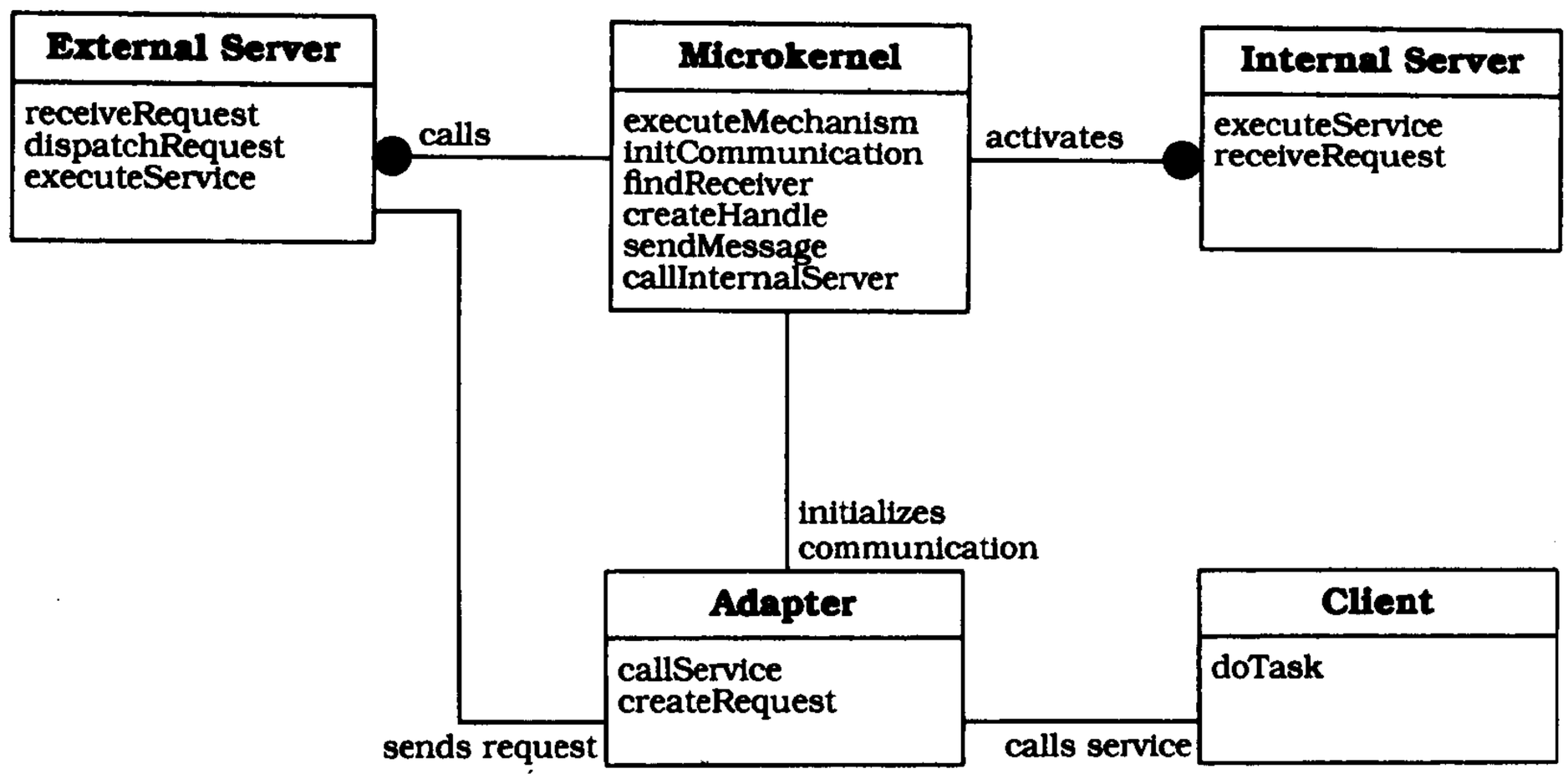
下面的OMT图给出了一个微核系统的静态结构。它的中心组件——微核与外部服务器、内部服务器和适配器协作。每个客户机关联于一个适配器，该适配器用作客户机与其外部服务器间的桥梁。内部服务器只能由微核组件访问。

6. 动态特性

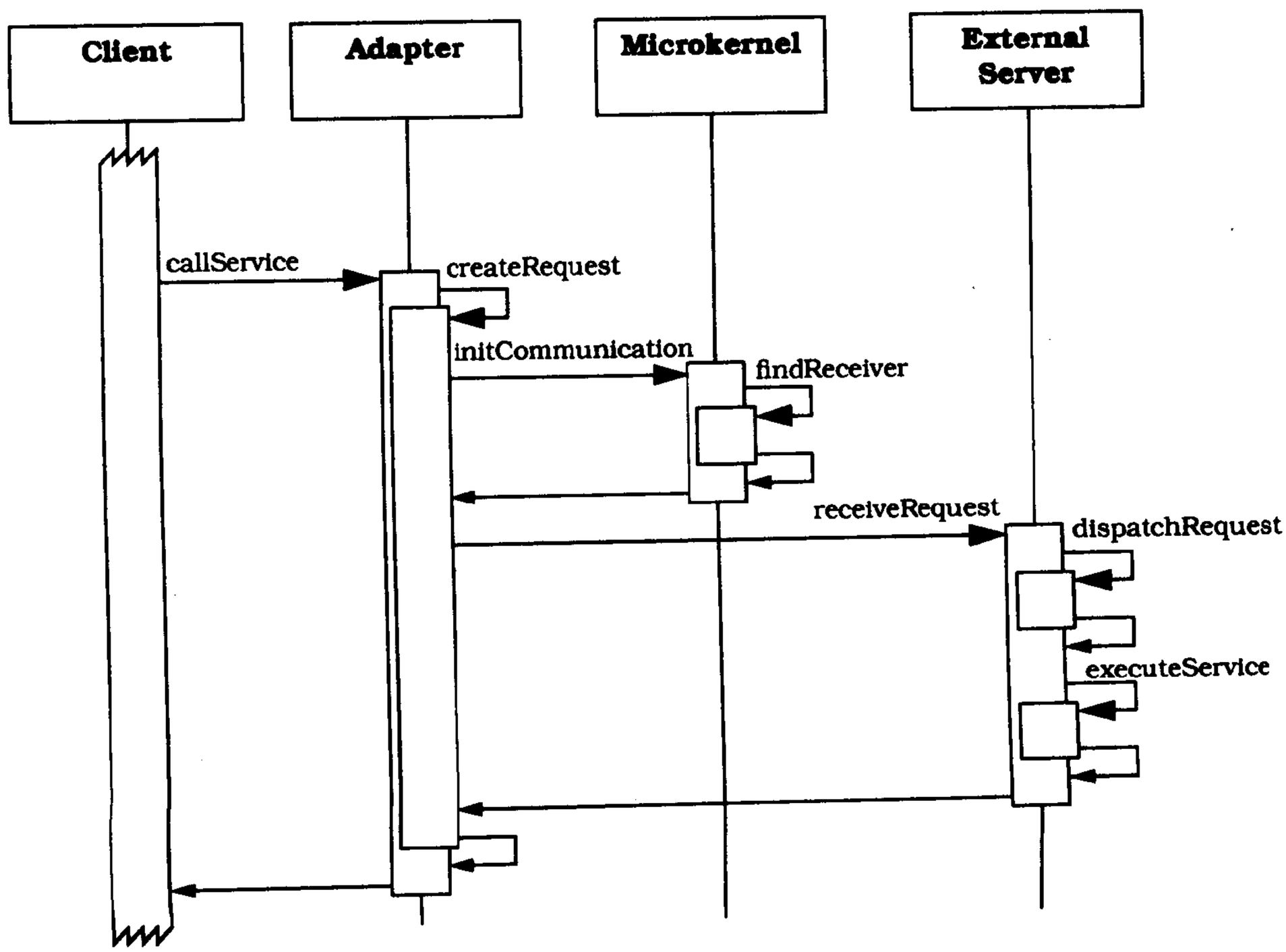
微核系统的动态行为依赖于它为进程之间的通信所提供的功能。在下面的场景中，我们假定远程过程调用的有效性。第一个场景也假定外部服务器不能访问微核接口——这之后的情况在第二个场景中简述。

场景I 展示了当一个客户机调用它的外部服务器的服务时的行为：

- 在客户机程序的控制流中的某一点处通过调用适配器客户机向外部服务器请求一个服务。
- 178 适配器构造一个请求并要求微核同外部服务器的通信连接。
- 微核确定外部服务器的物理地址并把该地址返回给适配器。
- 在检索到这个信息以后，适配器建立一个与外部服务器的直接通信连接。

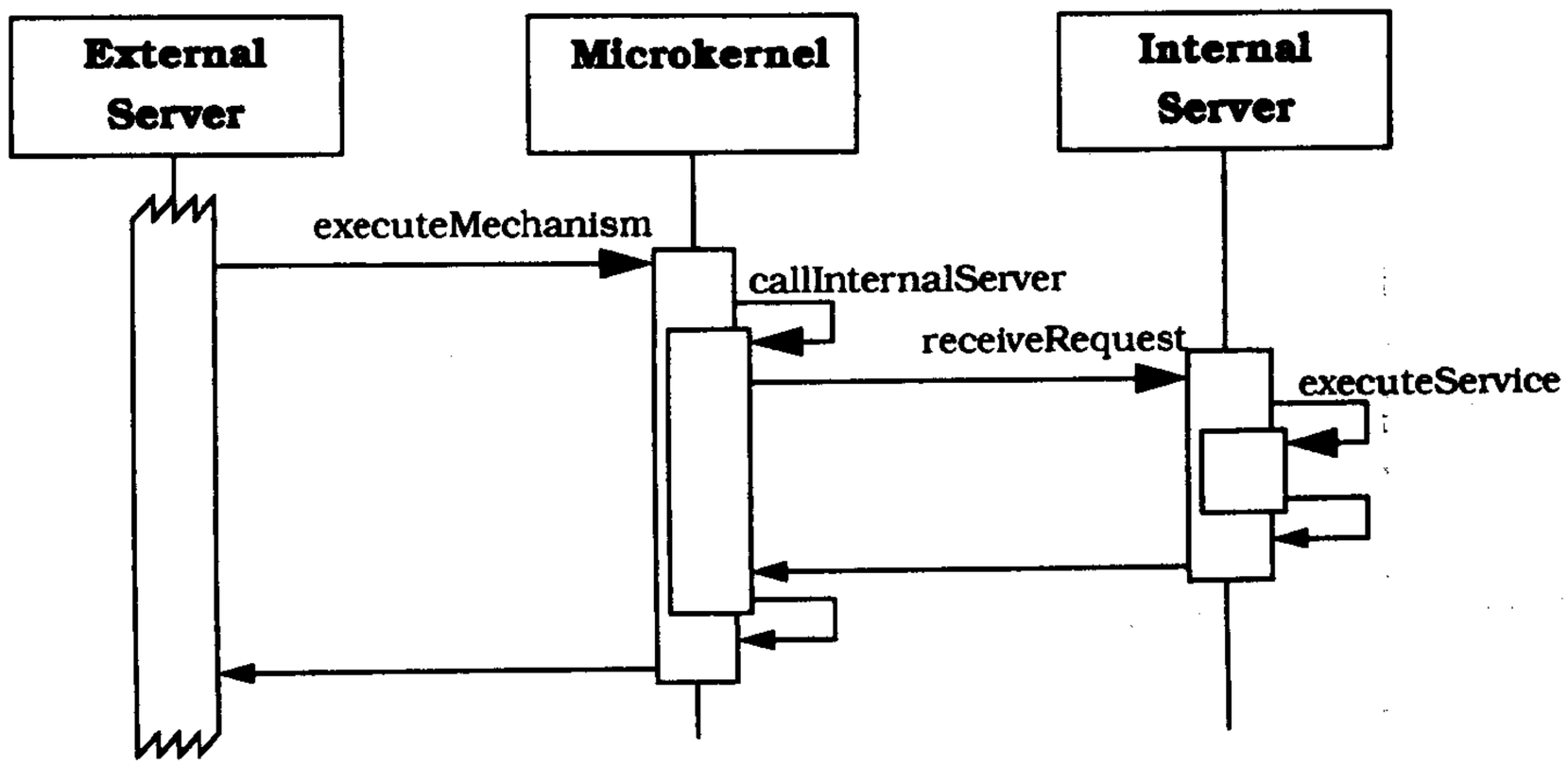


- 使用远程过程调用，适配器向外部服务器发送一个请求。
- 外部服务器收到这个请求，打开该消息并把任务分配给它的一个方法。完成了请求的服务以后，外部服务器把所有的结果和状态信息发回适配器。
- 适配器返回到客户机，客户机回过来继续执行它的控制流。



场景II 展示当一个外部服务器请求一个由内部服务器提供的服务时一个微核体系结构的行为。在这个场景中，我们假定内部服务器是用一个独立的进程来实现的。它可以像一个被动态地连接到微核的共享库那样被交替实现。

- 外部服务器向微核发送一个请求。
- 微核的编程接口的一个过程被调用来处理这个服务请求。在方法执行期间，微核向一个内部服务器发送一个请求。
- 收到请求以后，内部服务器执行被请求的服务并将所有的结果发回微核。
- 微核再把结果返回给外部服务器。
- 最后，外部服务器更正结果，并继续执行它的控制流。



7. 实现

为了实现一个微核系统，执行以下步骤：

(1) *分析应用领域*。如果你已经知道了外部服务器需要提供的办法，或如果你具备了有关你要实现的外部服务器的详细知识，开始步骤2。如果不是，进行领域分析，并明确为实现外部服务器所必需的核心功能。然后开始步骤3。

180

(2) *分析外部服务器*。分析外部服务器要提供的方针。然后你应该能够确定在你的应用领域你所需要的功能。

➡对Hydra，我们已经知道它必须实现的外部服务器：UNIX系统V、OS/2 Warp、Microsoft Windows和NeXTSTEP。因此，我们分析它们的编程接口以确定它们提供的服务。这个需求分析的结果为实现台式电脑的操作系统所必需的服务和服务分类的列表。❑

(3) *分类服务*。可能的话，把所有的功能分成语义独立的类别。

建立不直接同应用领域相关但又是实现系统基础结构所必需的操作的类别。这些操作的部分可以作为移居到内部服务器的候选者。

➡例如，在Hydra中被操作系统领域预定义的核心操作类别是存储管理、进程管理、I/O低层服务和通信服务。❑

下面的类别与应用领域的核心概念不直接相连：页、处理程序进程、文件系统、硬件和软

件驱动程序。在Hydra的实现中需要这些类别，但它们可以被迁移到内部服务器中去。

(4) 划分类别。把类别区分成应该是微核的一部分的服务和应该像内部服务器一样可获得的服务。需要为这种划分建立准则。例如，最好在微核组件内部实现实时、常用或硬件相关的操作。

➡在Hydra中，微核提供了诸如进程管理、存储管理、通信、低层I/O等服务。这些功能是实时的，并被其他所有组件使用，同时也封装了系统从属物。因此，它应该是微核的一部分。诸如页面缺陷处理程序、驱动程序或文件系统等所有额外服务由内部服务器来实现。 ❑ 181

(5) 为在步骤1明确的每一个类别寻找操作和抽象一致的完全集。记住微核提供的是机制而不是方针。外部服务器提供的每一条方针必须使用微核通过其接口提供的服务来实现。

➡不同的操作系统（如NUIX或Microsoft Windows）以不同的方式处理诸如创建进程或线程的操作。在Hydra中，两者都要支持。因此，我们为管理进程和线程提供了一个完全的基本机制集合。例如，我们为以下操作提供了服务：

- 创建和中止进程和线程。
- 中止和重启进程和线程。
- 读出或写入进程地址空间。
- 锁定和处理异常。
- 管理进程或线程间的关系。
- 线程同步和协调。

❑

(6) 确定请求传递和恢复的策略。详细描述微核应该为组件间的通信提供的手段。可以在几个可选方案中挑选，例如，异步通信与同步通信。组件间通信的关系应为一对一、多对一或多对多的关系。你集成的通信策略依赖于应用领域的需求。在许多情况下诸如消息传递或共享内存，低层通信手段是可行的，可以在它们之上建立更复杂的通信机制。为获得实现通信机制的更多信息，比较诸如转发器-接收器和客户机-分配器-服务器设计模式。

➡Hydra提供了两个基本的通信手段：

- 同步远程过程调用（RPC）。RPC让一个客户机能够调用一个远程服务器上的服务就好像它们是被本地过程调用所实现。这种机制有必要支持RPC，例如请求的打包与解包或消息穿过进程边界的传递，对调用者和调用的服务器来说都是隐藏的。 182
- 异步邮箱。一个邮箱是一种消息缓冲器。允许组件集从邮箱中读消息，允许另一个组件集往邮箱中写消息。可以允许一个组件进行这两项活动。 ❑

(7) 构造微核组件的结构。如果可能的话，使用分层模式来设计微核以区分微核的系统特定部分和系统无关部分。将微核接触其他组件的服务放在最顶层，使用低层来隐藏系统附属物，避免高层直接可见。

➡在Hydra项目中，我们决定使用面向对象技术来实现微核：

- 最低层由低级对象组成，这里低级对象隐藏了诸如总线体系结构的特殊硬件细节，使微核的其他部分不可见。
- 在中间层主要的服务由系统对象来提供，例如，负责存储管理的对象和管理进程使用的对象。

• 最上层由微核对外展示的所有功能组成，并表示了微核为进程服务的网关。 ❑

(8) 为了详细定义微核的编程接口，需要决定从外部如何存取这些接口。必须明确地考虑微核是作为一个独立进程来实现还是作为一个由其他组件物理上共享的模块来实现。如果是后者，可以使用传统方法调用来运用微核的方法。

如果作为一个独立的进程来实现微核，那么要求现存的通信手段从组件向微核传递请求。在这种情况下，需要知道内核代表了一个独占的资源，因此可能成为一个瓶颈。为了提高整体性能，你可以在微核内提供等待输入请求的多线程，并使用同一个线程或其他的线程来执行相应的服务。如果设计了这样的一个多线程系统，那么要确信内部数据的一致性得到了保证。

183

➥由于Hydra代表了一个操作系统，因而它的微核组件是每个用户进程的一部分。因此，服务是传统的系统调用可访问的。将这些功能进行逻辑分组放入支持诸如文件系统操作或进程管理那样的API（应用程序编程接口）中去。

请求一个Hydra系统调用将导致一个系统陷阱。软件例外由微核中的一个特别陷阱处理程序来处理。陷阱处理程序分析导致系统陷阱的中断类型，并把工作委托给它的一个内部服务对象来完成。完成服务以后，调度对象决定下一步应该执行那一个可获得的线程并对它指定一个处理单元。 ❑

(9) 微核负责管理所有的诸如存取部件、设备或设备语境（图形用户接口实现之中的输出区句柄）那样的系统资源。微核维护有关资源的信息并允许以一种协作和系统的方式存取它们。如果组件要存取一个资源，它们使用一个惟一的标识符（句柄）而不是直接地访问资源。微核有创建这些句柄和提供句柄和资源之间映射的任务。这个映射可以使用哈希表来实现。资源管理不仅仅是只提供一个映射，与此同时，微核也必须执行资源的共享、加锁、分配和回收等策略。

➥在Hydra内，句柄指的是作为资源类的实例的对象。每一个这样的对象提供了一致的接口来控制对特定资源的存取。

我们的资源对象展示了如下的遵循Windows NT方法的接口：

```
class Resource {
     String name;                    // Name of object
     void OpenHandle();              // open handle to object
     Handle IterateHandles();        // iterate over handles
     Body pointerToObject;           // pointer to real object
     ...                             // (much, much more ...)
}
```

❑

184

(10) 设计和实现像独立进程或共享库那样的内部服务器。执行这一步时要与步骤7~9相并行，因为某些微核服务需要访问内部服务器。它有助于区分主动服务器和被动服务器：

• 主动服务器类似进程来实现。
• 被动服务器类似于共享库。

被动服务器总是用直接调用它们的接口方法来请求，主动服务器需要不同的手段。主动服务器进程用事件循环来等待输入请求。如果它通过一个可利用的通信手段接收到一个请求，那么它就解释并执行代表调用者的一个服务。注意，内部服务器由微核独占访问——不允许其他的组件调用内部服务器的服务。

➡在Hydra中，我们提供了设备驱动程序，鉴定服务器和页面缺陷处理程序等其他的组件，并作为内部服务器来实现它们。

图形卡驱动程序开发成共享库，是因为它只代表客户机行动。相反，页面缺陷处理程序是独立的进程。它们始终驻留在主存储器中，而且不能被交换到外存中。 ❑

(11) 实现外部服务器。外部服务器包括的所有策略都基于微核的编程接口中的可获得的服务。一个外部服务器接收请求，分析它们，执行相应的服务并把结果发回到调用者。执行服务时，外部服务器可以调用微核中的操作。

因此，每个外部服务器作为一个提供了其自身服务接口的独立进程来实现。一个外部服务器的内部体系结构依赖于它所包含的策略。

详细定义外部服务器如何向它们的内部过程分配请求。例如，它们可以集成一个分配器组件，该分配器组件执行一个主要事件循环并等待输入请求。一个请求到达时，分配器对其解包，解释请求并通过一个回调机制调用相应的过程。如果你设计一个外部服务器作为应用程序框架的话，这是很有用的。有关事件驱动方法的描述可参见反应器模式[Sch94]。 185 ❑

➡我们要为Hydra开发如下的外部服务器：

- 微软的 Win32和Win16 API的完整实现，要允许用户运行Windows NT，Windows 3.11 和Windows 95应用程序。
- 由IBM OS/2 Warp 2.0提供的完整功能。
- OpenStep的实现。
- 由X/Open 描述的所有相关 UNIX系统V接口。 ❑

(12) 实现适配器。适配器的主要任务是向它的客户机提供被转发给外部服务器的操作。当客户调用外部服务器的一个功能时，适配器把所有的相关信息打包成一个请求并把请求转发给相应的外部服务器。然后适配器等待服务器的回应，并最后使用组件间通信手段把控制返回给客户机。

可以设计适配器或者作为编译期间静态地链接到客户机的普通库，或者作为按需动态地链接到客户机的共享库。可以把适配器视为准确代表一个外部服务器的一个代理。因此，可以使用代理模式来实现适配器。通过允许适配器自身执行某些API操作而不是把请求转发给外部服务器，或者在缓存中存储几个客户机请求再转发它们，用这种方法可以优化适配器。公共请求的应答也可以存放在这里。有关缓存的优点和缺点可参见代理模式。

必须决定一个适配器应该是否对所有的客户机都是可用的，或者每个客户机是否都与其自己的适配器相关联。第一种方法导致较小的存储，而第二种方法可以导致较快的反应时间。

➡如果我们用Microsoft Windows外部服务器来设计Hydra，那么所有同这个服务器关联的客户机应用使用Win16或Win32 API。在一个本地Windows 3.11系统中，所有的API作为一个共享库集合都是可用的。但是，在Hydra中，Windows客户机和Windows服务器都是独立的进程。由于我们想不加修改地在Hydra上能运行Windows应用程序，我们需要提供相同的环境。因此，我们实现一个适配器作为Windows客户机和Windows服务器之间的桥接。当一个客户机调用一个Win16或Win32 API函数时，由一个适配器来处理调用，该适配器将一个请求转发给Windows外部服务器。于是，现存的Windows客户机可以在Hydra系统上被编译和执行。 186 ❑

(13) 为准备运行的微核系统开发客户机应用程序或使用现存的客户机应用程序。当为一个特定的外部服务器创建一个新的客户机时，它的体系结构仅受限于外部服务器提出的约束。即客户机依赖于它们的外部服务器实现的策略。

➡在Hydra中，我们可以通过经由Microsoft Windows适配器获得Microsoft Windows外部服务器的服务来开发Microsoft Windows应用程序。 ❑

8. 已解决的例子

Hydra的开发完成不久，我们被要求集成模拟了Apple MacOS操作系统的外部服务器。为了在Hydra之上提供一个MacOS模拟，如下的活动是必需的：

在Hydra微核的顶部建立一个外部服务器，该微核实现了MacOS提供的所有编程接口，包括Macintosh用户接口的策略。在其主循环中，MacOS服务器等待输入请求，这些请求存放在一个具体分配给该MacOS服务器的消息端口中。服务器把这些请求从消息端口中提取出来，解释它们并把它们分配到内部过程。这些过程模拟了MacOS环境中的典型策略。

提供一个适配器，它被设计成一个库，动态地链接到客户机。对每一个在本地MacOS系统中可用的API函数，必须由库提供一个句法上相同的过程。每一个这样的过程负责把请求的类型、主题、发送者和接受者的标识符打包成一个消息。然后它在微核中调用过程sendMessage，微核回过来把消息存储进MacOS服务器的消息端口。

187

实现MacOS要求的内部服务器。例如，一个内部服务器提供网络协议AppleTalk。必须修改微核来调用这些代表MacOS服务器的附加内部服务器。

9. 变体

带有间接的客户机-服务器连接的微核系统。在这个变体中，要向一个外部服务器发送一个请求或消息的一个客户机要求微核提供一个通信信道。请求的通信信道建立以后，客户机和服务器之间用微核作为消息基干来间接通信。使用该变体导致一个所有请求都要通过微核的体系结构。例如，当安全性需求迫使系统控制参与者间的所有通信时，你就可以运用它了。

分布式的微核系统。在这个变体中，一个微核也可以充当一个消息基干负责发送消息到远程机器或从远程机那里接收消息。分布式系统中的每台机器都使用它自己的微核实现。从用户的角度来看，整个系统表现为一个单一的微核系统——分布式微核系统对用户保持透明。一个分布式微核系统允许你在一个计算机或微处理器网络中分布服务器和客户机。为了达到这个目标，在一个分布式实现中微核必须包括用于相互通信的额外服务。

10. 已知应用

Mach操作系统[Tan92]是由卡内基-梅隆大学开发的，它的第一个版本发布于1986年。Mach微核打算形成一个其他操作系统可以效仿的基础。使用Mach作为系统内核的商用操作系统之一是NeXTSTEP。

Amoeba操作系统[Tan92]由两个基本要素组成：微核本身和用来实现大多数Amoeba的功能的服务器（子系统）的集合。内核提供了四个基本服务：进程和线程的管理，系统存储的低级管理，点对点通信服务和组通信服务，和低级I/O服务。内核不能提供的服务必须由服务器进程来实现。这将导致内核规模的减少并提高了灵活性。

188

Chorus[Cho90]是一个最初由法国研究所INRIA特地为实时应用开发的商用微核系统。UNIX系统V作为一个外部服务器是可用的。

Windows NT[Cus93]是微软为高性能服务器开发的操作系统。从体系结构观点上看，Windows NT无疑是一个微核系统。它提供了三个外部服务器：一个OS/2 1.x 服务器、一个POSIX服务器和一个Win32服务器。

MKDE（Microkernel Datenbank Engine）系统[Woo96]引入了一个遵循微核模式的数据库引擎。在这个系统中，微核负责提供基本的服务，如物理数据存取、数据缓存和事务管理等。各种外部服务器运行在微核之上，并且提供了基础微核不同的概念视图。一个概念视图表示了一个依照给定数据模型的数据抽象，例如一个关系型SQL数据库的数据模型。诸如会计系统的应用可以使用外部服务器来访问数据库。MKDE实现了分布式微核变体来支持分布式环境。

11. 效果

微核模式具有如下重要**优点**：

可移植性。微核系统具有很好的移植性，有两个原因：

- 如果你把微核系统装到一个新的软件或硬件环境中，那么大多数情况下你不需要重装外部服务器或客户机应用程序。
- 把微核迁移到一个新的硬件环境中时只需要修改硬件相关部分。

189

适应性和可扩展性。一个微核系统的最大的长处之一是它的适应性和可扩展性。如果你需要实现一个额外的视图，所有你需要做的是添加一个外部服务器。以额外能力扩展一个系统只需要对内部服务器进行添加或扩展。

策略与机制分离。微核组件提供了使外部服务器得以实现它们的策略所必须的所有机制。策略与机制的严格区分提高了整个系统的可维护性和可改变性。它也允许你添加新的实现它们自己的特定视图的外部服务器。如果微核组件用作实现策略，那么由外部服务器实现视图的限制将是不必要的。

如果我们考虑微核体系结构的分布式微核变体，那就会有如下进一步的优点：

可扩展性。一个分布式微核系统可用于计算机网络的操作系统或数据库系统的开发，或用于本地存储的多处理机开发。如果你的微核系统工作在一个网络机上，当你向一个网络添加一台新机器时，可以很容易将微核系统扩展成新的配置。

可靠性。为获得可靠性，有用性和容错性很重要[Tan92]。一个分布式微核体系结构支持可用性，因为它允许你在多个机器上运行同一个服务，提高了可用性。因此，如果一台服务和一台机器失败了，失败未必对一个应用程序产生影响。容易支持容错性是因为分布式系统允许你隐藏失败而不让用户知道。

透明性。在一个分布式系统中，组件可以被分布在整个网络的客户机中。在这样一个配置中，微核体系结构允许每个组件访问其他组件而无需知道它们的位置。内部进程同服务器的通信的所有细节由适配器和微核隐藏而不让客户知道。

190

微核体系结构框架也有一些**不足**：

性能。如果我们将设计用来提供特定视图的单片软件系统与一个支持不同视图的微核系统相比较，在大多数情况下前者的性能将比后者好。因此，我们将不得不为适应性和可扩展性付

出代价。但是，如果为了性能，将微核系统内部的通信进行了优化，那么这种代价不可以被忽视[Tan92]。

设计和实现的复杂性。开发一个基于微核的系统不是一个简单的任务。例如，有时分析或预测一个微核组件必须提供的基本机制可能就非常困难。此外，在需求分析和设计期间，区分机制和策略需要精通领域知识并付出相当的努力。

参见

代理者模式适合于由交互组件和去耦组件组成的分布式软件系统。在代理者模式中，客户机使用远程过程调用或消息传递访问由服务器提供的服务。与微核体系结构框架相比，代理者模式集中于网络上的分布式处理。这些模式之间的进一步区分是一个代理者系统内的模块耦合通常不如一个微核系统中的模块耦合得那样紧密。因此，当你开发一个分布式微核系统的时候，可以把这两个模式结合起来使用。

映像模式提供了两层的体系结构。一个基本层对应微核和内部服务器的组合。一个元层使得基本层次功能的行为可以动态地变化，例如改变资源管理或组件间通信的策略。此外，这种元层允许集成对基本层服务的特定用户扩展。这正对应于一个微核体系结构中的外部服务器所提供的服务。同微核模式相比，元层次的修改在特定接口的元对象协议[MOP]的协助下间接完成。这可让用户描述一个变化，检查其正确性，并自动地将变化集成到元层次。在最近操作系统的开发中，映像模式和微核模式通常是结合在一起的[Zim96]。

191

层模式和微核模式之间的关系是两方面的。首先，一个微核系统也可以视为是层模式的一个变体。依靠内部服务器，微核实现一个虚拟机。内部服务器处于最低层，也属于虚拟机。由虚拟机执行的应用程序包括外部服务器和适配器以及代表了虚拟机顶部的层。一个外部服务器和一个适配器一起可以被视为是微核顶部的第二虚拟机。被提供的每个个性对应一个单独的第二层虚拟机。客户机应用程序形成层次结构中的最高层，并使用特定的个性。

第二，对某些应用程序而言，这两种模式之一都可以使用。考虑商业应用程序的体系结构[Fow96]。一个非常通用的方法是把这些系统分成三层：

- 最底层包括数据库管理系统。
- 中间层包含商业逻辑。
- 最高层包含了不同的商业应用程序。

如果这些应用程序可以被分成不同的种类，你可以改为引入一个微核负责来实现核心的业务逻辑。这个微核可以额外地封装访问DBMS的功能到内部服务器。外部服务器可以提供微核机制的不同视图，包括以不同的方式获取一个特定商业种类的具体功能的业务逻辑。于是，实际的商业应用就是客户机。但是，如果所有的客户机程序建立在以业务逻辑为基础的同一个视图之上，就不应使用微核模式。

192

2.5.2 映像

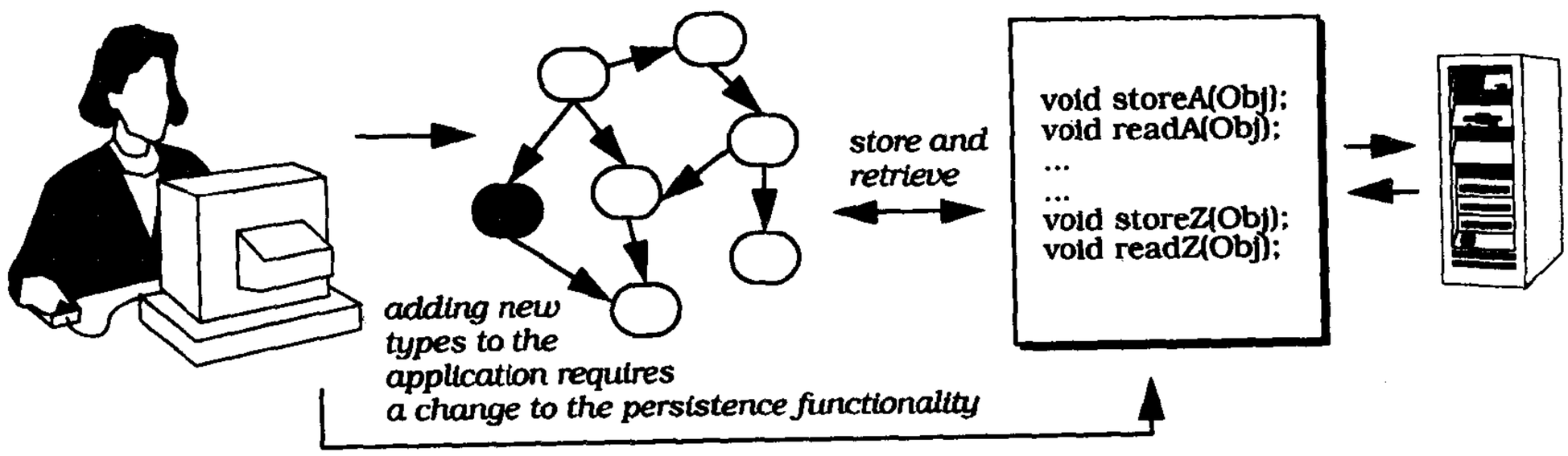
映像（*Reflection*）体系结构模式为动态地改变软件系统的结构和行为提供了一种机制。它支持诸如类型结构和函数调用机制等基本方面的修改。在这种模式中，一个应用程序可分成两

部分。一个元层次提供所选系统属性的相关信息并使软件含自述信息。一个基本层次包括应用程序逻辑。它的实现建立在元层次之上。改变保存在元层次上的信息会影响其后的在基本层次上的行为。

别名 开放实现（Open Implementation），元层次体系结构（Meta-Level Architecture）

1. 例子

考虑一个需要向磁盘写入对象和再从磁盘读出对象的C++应用程序。由于持久性并不是C++的固有特色，因此我们必须指定如何在应用程序中存储和读出每种类型。对该问题的许多解决方案，比如实现特定类型的读写方法，代价昂贵且易于出错。例如，当我们改变应用程序的类的结构的时候，我们必须同时修改这些方法。



对缺少持久性的其他的解决方案产生了其他问题。例如，我们可以为持久性对象提供一个特定的基类，以继承存储和重载的读方法，由该基类导出应用程序。类结构的改变需要我们在现存的应用程序类中修改这些方法。持久性和应用程序功能牢牢地交织在一起。 193

取而代之，我们需要开发一个独立于特定类型结构的持久性组件。但是，为了存储和读出任意的C++对象，我们需要动态的对它们的内部结构进行存取。

2. 语境

建立一个支持它们自己对先验信息修改的系统。

3. 问题

软件系统随时间而演化。它们必须开处于可更改状态以响应对技术和需求的修改。设计一个在很大程度上满足不同需求先验信息的系统是一个首要任务。一个好的解决方案是描述一个可随时修改和扩展的体系结构。可得系统可以按要求适应更改需求。换句话说，我们需要为变化和演化进行设计。有几个要求与这个问题相关：

- 更改软件单调乏味，易于出错，而且常常是代价昂贵。大范围的修改通常涉及许多组件，甚至一个组件内的局部修改也可能影响系统的其他部分。必须仔细的实现和测试每次更改。主动地支持和控制其自身修改的软件可以更有效和更安全的被更改。
- 自适应的软件系统通常具有一个复杂的内部结构。与变化有关的方面被封装到独立的组件中。应用服务的实现涉及到错综复杂的许多小组件[GHJV95]。为了使这样的系统可维护，

我们宁愿对系统的维护者隐藏这中复杂性。

- 要使一个系统更改所必需的技术(如参量化、分成子类、混入、复制和粘贴)越多，其更改就越笨拙和复杂。应用到各类变更的统一机制是易于使用和修改的。
- 变化可以是任何规模的，从为常用命令提供快捷键到为一个特定的用户调整应用程序框架。
- 甚至软件系统的基本方面都可以发生变化，例如组件间的通信机制。

194

4. 解决方案

使软件系统自明，并且可以适应和更改的系统的结构和行为。这导致一个体系结构分成两个主要部分：一个*元层次*和一个*基本层次*。

元层次提供了一个软件的自表示来给出软件自身结构和行为的知识，元层次由所谓的*元对象*(*metaobjects*)组成。元对象封装和表达了有关软件的信息。例子包括类型结构、算法，以及函数调用机制。

基本层次定义了应用程序逻辑，其实现使用元对象来保持这些可能要更改的方面的独立性。例如，基本层次组件可能仅仅通过一个元对象来进行软件间的通信，而该元对象实现了一个特殊用户定义的函数调用机制。改变这个元对象就改变了基本层次组件通信的方式，而无需修改基本层次的代码。

指定了操作元对象的接口。它被称为*元对象协议*(MetaObject Protocol, MOP)，它允许客户机描述特殊的变化，例如上面提到的函数调用机制元对象的修改。元对象协议本身负责检查变化规格说明的正确性，并完成这个变更。通过元对象协议的元对象的每次操作都影响了随后的基本层次行为，就像函数调用机制例子那样。

➡对位于例子应用程序的基本层次的持久性组件而言，我们指定提供运行期间的类型信息的元对象。例如，为了存储一个对象，我们必须知道它的内部结构和它的所有数据成员的设计。以这些可获得的信息，我们可以递归地迭代任何给定的对象结构并使其可分解成内置类型的序列。持久性组件“知道”如何去存储这些信息。如果我们改变运行期间的类型信息，那么我们也改变了存储方法的行为。例如，不再持久的对象类不再存储。对每种方法采用相似的策略，我们可以构造一个能读出和存储任意数据结构的持久性组件。 ❑

195

5. 结构

元层次（*meta level*）由一组元对象组成。每个元对象封装了一个关于基本层次的结构，行为或状态的一方面的所选信息。这样的信息有三个来源：

- 它可以由系统运行期间的环境提供，如C++类型标识对象[DWP95]。
- 它可以提供用户定义的，如前面一节中的函数调用机制。
- 它可以从运行期间的基本层次检索到，例如关于当前计算状态的信息。

所有的元对象一起提供一个应用程序的自表示。元对象使得原本间接可获得的信息变得直接可存取而且可修改。几乎每一个系统的内部都可以以这种方式来描述。例如，在一个分布式系统中，可能有提供基本层次组件的物理位置信息的元对象。其他的基本层次组件可以使用这些元对象来确定它们的通信伙伴是远程的还是局部的。它们可以选择最有效的函数调用机制来与它们通信，函数调用机制本身可能由其他的元对象提供。进一步的例子包括类型结构、实时

约束，进程内通信机制和事务协议。

但是，用元对象你表示什么依赖于什么应该是自适应的。只有可能要更改的系统细节或客户与客户之间有差异的内容要由元对象来封装。在一个应用程序的整个生存周期内期望系统的某些方面保持稳定是不现实的。

一个元对象的接口允许基本层次存取它维护的信息或它提供的服务。例如，一个提供分布式组件定位信息的元对象将提供存取该组件的名字和标识符的函数，它处在的进程的有关信息，以及进程运行其上的主机信息。实现函数调用机制的一个元对象将提供激活某个特定收件人的某一具体函数的方法，包括输入输出的参数传递。一个元对象不允许基本层次修改其内部状态。操作只有通过元对象协议或其自己的计算才是可能的。 196

*基本层次(base level)*模拟并实现了软件的应用程序逻辑。它的组件表示了系统提供的各种服务以及它们下面的数据模型。基本层次也描述了它所包含的组件的基本协作和结构关系。如果该软件包括一个用户接口，这也是基本层次的一部分。

基本层次使用元对象提供的信息和服务，如组件的位置信息和函数调用机制。这样基本层次就保持灵活性——它的代码独立于可能要发生改变和调整的内容。使用元对象的服务，基本层次组件不需要关于它的通信伙伴的具体的位置的硬码信息——它们向相应的元对象咨询这个信息。

基本层次组件或者与它们所依赖的元对象直接相连，或者通过特别的检索函数向它们提交请求。这些函数也是元层次的一部分。如果基本层次和元对象之间的关系是相对静止的，那么第一种类型的连接比较好。例如，如果一个对象需要它自身的类型信息，那么基本层次组件总是咨询同一个元对象。如果被基本层次使用的元对象动态地变化，那么使用第二种类型的连接，就像在持久性组件的存储过程中的情况。

类 基本层次	协作者 元层次
责任 • 实现应用程序逻辑 • 使用由元层次提供的信息	

类 元层次	协作者 • 基本层次
责任 • 封装可更改的系统内部成分 • 提供一个接口以方便对元层次的修改	

197

*元对象协议（MOP）*作为元层次的外部接口，并使一个映像系统的实现以一种已定义的方式来获得。元对象协议的客户机程序可以描述使用基本层次的元对象或其关系的修改，客户机程序可能是基本层次组件、其他的应用程序或授权的用户。元对象协议本身负责执行这些变更。这提供了一个对其自身修改的有明确控制的映像应用程序。

为了继续上述例子，一个用户可以描述一个用于基本层次组件之间通信的新的函数调用机制。作为第一步，用户用这个新的函数调用机制的代码提供了这个元对象协议。然后元对象协

议执行了这个变更。这一点是可以做到，例如，通过产生一个包括这个新机制的用户已定义执行代码的合适元对象，编译产生的元对象，动态地把它与应用程序链接起来，并把所有的“旧”的元对象的引用刷新为“新”元对象的引用。

元对象协议通常被设计为一个独立的组件。这样做可支持对几个元对象操作的函数的实现。例如，封装了有关分布式组件的定位信息的元对象的修改最终要求相应的函数调用机制元对象的一个更新。如果我们把这种变更的责任委托给元对象本身，则它们之间的一致性难以维护。元对象协议对这样执行的每一次修改都有一个较好的控制，因为它的实现与元对象分离。

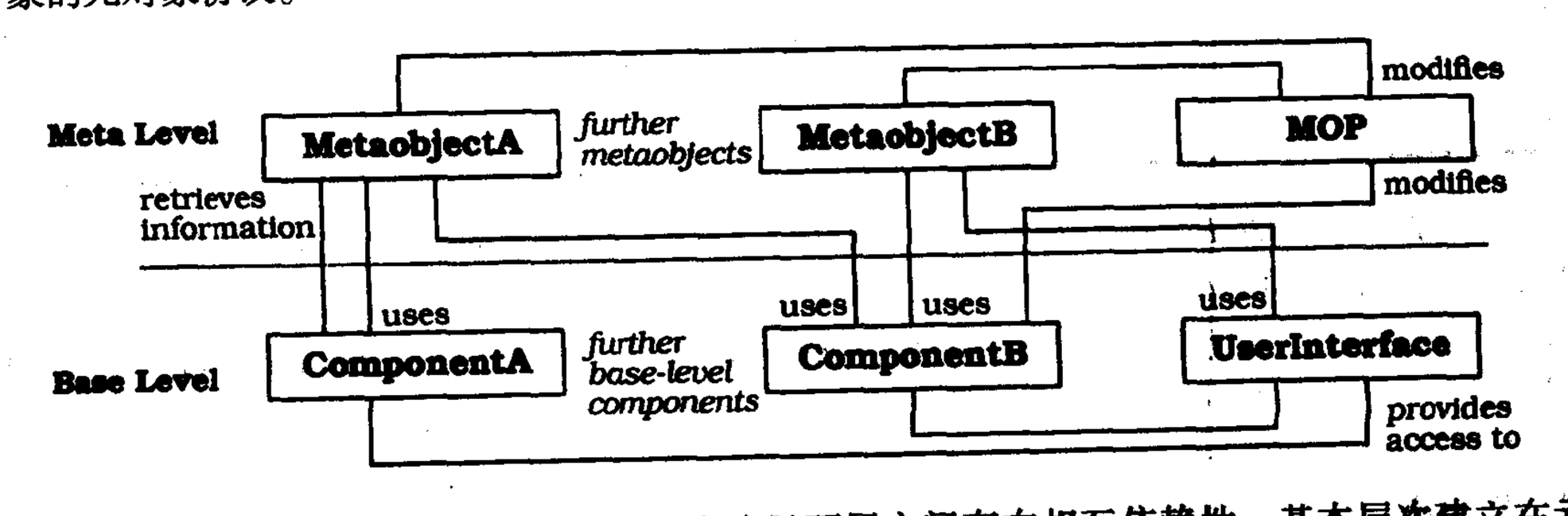
类 元对象协议	协作者 • 元层次 • 基本层次
责任 • 为描述对元层次的变更提供一个接口 • 执行已描述的变更	

198

为了执行这些变更，元对象协议需要进入元对象的内部。如果被进一步授权改变基本层次对象与元对象间的连接，也需要访问基本层次组件。提供这种访问的方法之一是允许元对象协议直接操作它们的内部状态。另一个较安全但效率较低的提供这种访问的方法是为元对象和基本层次组件提供一个用于它们的操作的特别接口，只有通过元对象协议才能访问。

由于基本层次实现明确地建立在由元对象提供的信息和服务上，因此元对象提供的信息和服务的改变对基本层次的后继行为有直接影响。在我们的例子中，我们改变了基本层次组件的通信方法。然而，与通常的修改相比，系统的改变无需修改基本层次代码。

映像体系结构的一般形式与分层系统非常相似。元层次与基本层次是两个层次，其中每一个都提供其自身的接口。基本层次层描述了开发应用的功能的用户接口。元层次定义了修改元对象的元对象协议。



但是，与一个分层体系结构相比较，在这里两层之间存在相互依赖性。基本层次建立在元

层次之上，反之亦然。元对象实现了例外情况被执行的行为时便有了后者的一个示例。这种必须被执行的例外过程通常依赖于计算的当前状态。元层次从基本层次，通常从提供中断服务的不同组件检索这个信息。在一个纯粹的分层体系结构中，层之间的这种双向依赖性是不允许的。每个层只能建立在它下面的层之上。 199

➡在我们的持久性组件例子中，我们描述了能对我们的应用程序中的类型结构提供自省存取的元对象——也就是说，它们可以存取应用程序的结构或行为的信息，但不能修改它。我们可以得到一个给定类型或对象的名称、大小、数据成员和超类的有关信息。额外的元对象描述了允许一个客户机实例化任意类型的对象的一个函数。例如，当我们从一个数据文件中重建一个对象结构时，使用这个函数。元对象协议包括添加新的，以及修改现存的运行期间的类型信息的功能。

持久性组件的主体独立于我们的应用程序的具体类型结构。例如，存储过程仅仅实现了把一个给定对象结构递归地分解成一系列内置类型的一般算法。如果它需要关于用户定义类型的内部结构信息，它向元层次咨询。直接存储内置类型的数据成员。所有其他的数据成员被进一步分解。 ❑

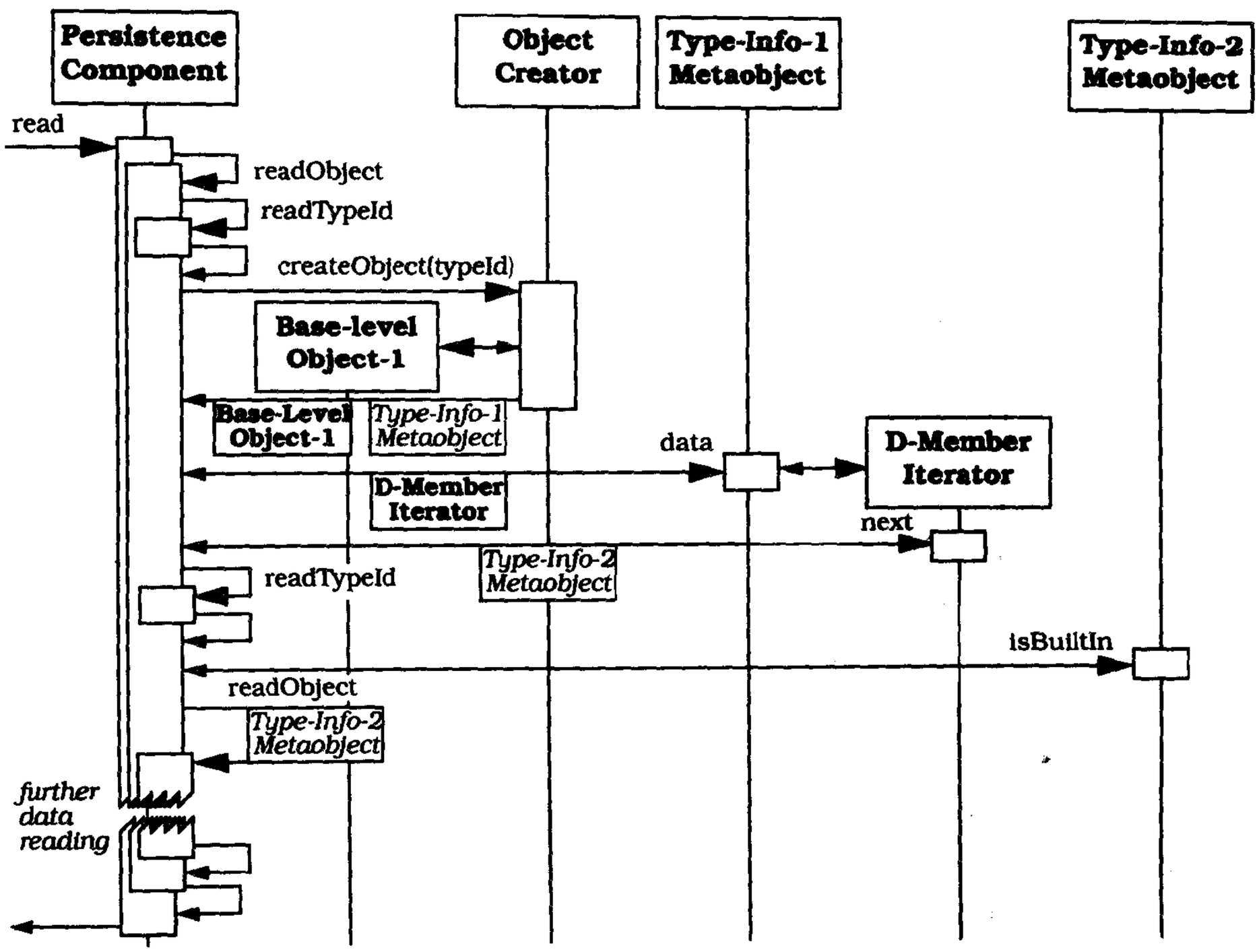
6. 动态特性

一般地描述映像系统的动态行为几乎是不可能的。于是我们基于持久性组件例子提出了两种场景。元对象协议细节和它所包含的元对象可参见“实现”小节。

场景I 例示了读存储在磁盘文件中的对象时基本层次和元层次之间的协作。所有的数据按适当的顺序存储，每个对象前有一个类型标识符。该场景还对特殊情况别进行了抽象，比如对象结构中的读字符串，静态成员以及恢复周期。该场景被分成六个阶段：

- 用户要读存储的对象。请求被提交给持久性组件的`read()`过程，同时提交的还有存储该对象的数据文件的名称。
- 过程`read()`打开数据文件并调用读第一个类型标识符的内部`readObject()`过程。
- 过程`readObject()`调用负责对象创建的元对象。“对象创建器”元对象实例化确定类型的一个“空”对象。它返回一个句柄给该对象，并且返回一个句柄给对应的运行期间类型信息（RTTI）元对象。 200
- 过程`readObject()`请求从它对应的元对象读出的对象的数据成员之上的一个迭代器。过程在对象的数据成员之上迭代。
- 过程`readObject()`读出下一个数据成员的类型标识符。如果类型标识符表示一个内置类型（一种我们未说明的情况）基于对象中数据成员的大小和偏移，过程`readObject()`从文件到数据成员直接分配下一个数据项。否则递归调用`readObject()`。如果该数据成员是一个指针，那么这个递归开始于一个“空”对象的创建。如果不是一个指针，则递归调用`readObject()`对包含在该数据成员中的对象的现存布局进行操作。 201
- 读完数据之后，`read()`过程关闭数据文件并把新对象返回给提出要求的客户机程序。

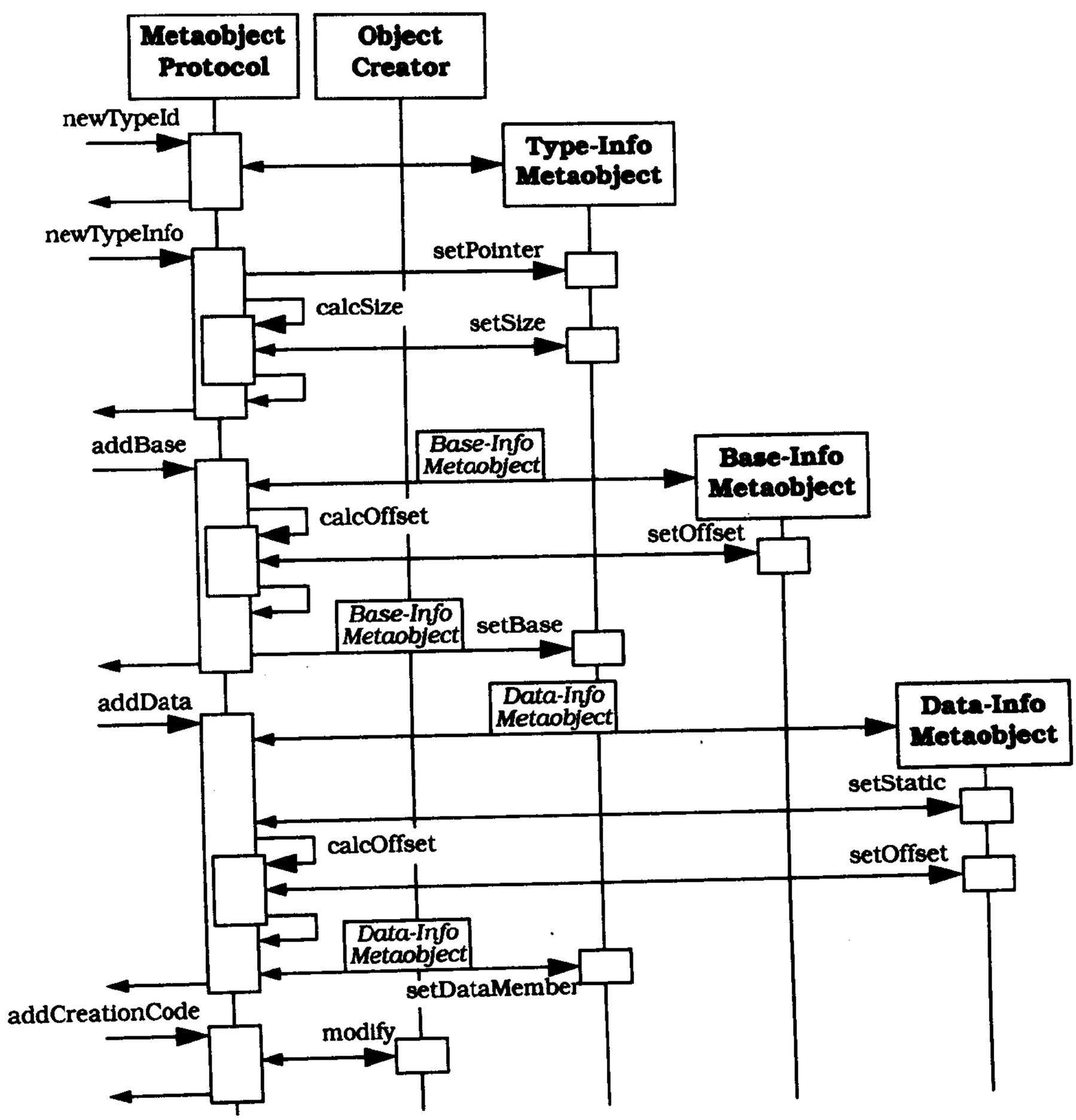
场景II 描述了向元层次添加类型信息时元对象协议的使用。考虑由某应用程序使用的一类库，该应用程序以新类型将旧版本更改为一新版本。为了存储和读出这些类型，我们必须用新的对象来扩展元层次。添加这些信息可以由用户，或自动地用工具来完成。为了简单起见，我



们将类type_info和extTypeInfo统一起来，正如“实现”小节所描述的那样。这个场景被分成六个阶段，为每种新类型执行的：

- 一个客户机程序调用元对象协议针对应用程序中的每个新类型描述运行期间类型信息。类型的名字作为变元来传递。
- 元对象协议为该类型创建type_info类的一个元对象。这个元对象也作为一个类型标识符。
- 客户机程序调用元对象协议添加扩展类型信息。这包括设置类型大小、是否为一指针及它与其他类型的继承关系。为了处理继承关系，元对象协议创建类baseInfo的元对象。这为一个特定的基类的type_info对象和它在新类型中的偏移地址保持了一个句柄。
- 在下一步中，客户机程序描述了新类型的内部结构。每一个数据成员的名字和类型提供给元对象协议。元对象协议为每一个数据成员创建类dataInfo的一个对象。它为成员类型、名称以及它是否为一个静态数据成员等信息保持了指向type_info对象的一个句柄。如果数据成员是静态的，那么dataInfo对象也保持了数据成员的绝对地址，否则保持的是新类型中的偏移地址。
- 客户机程序调用元对象协议来修改包括新类型作为一个数据成员的现存类型。为每种类型添加了适当的数据成员信息。由于这一步与前一步非常相似，我们在下面的对象消息序列图中不再说明它。
- 最后，客户机程序调用元对象协议来调整“对象创建器”元对象。持久性组件在读持久性

数据时必须能够实例化新类型的对象。基于以前添加的类型信息，元对象协议自动地为新类型对象的创建形成代码。它进一步将新代码集成到“对象创建器”元对象的现存实现中，编译修改的实现，并把它与应用程序链接起来。 202



203

7. 实现

下面的指导方针有助于实现一个映像体系结构。如有必要迭代某些步骤。

(1) 定义应用程序的模型。分析问题领域并把它分解成一个适当的软件结构。回答下述问题：

- 软件应该提供什么样的服务？
- 哪些组件可完成这些服务？
- 组件之间的有什么关系？
- 组件之间如何合作与协作？

• 组件对什么数据进行操作?

• 用户如何与软件交互?

描述模型时要遵循一个合适的分析方法。

➥在我们的C++磁盘存储例子中，持久性组件是一个仓库管理应用程序[Coad95]。我们确定表示物理存储的组件，例如仓库、过道、箱柜等。我们也确定了表示定单和物品的组件。它是一个在系统崩溃后我们可以以一个有效状态重新计算的需求。因此，仓库的物理结构及其物品的数量都有必要做持久性保存。我们需要两个组件来完成这一任务。一个持久性组件提供存储和读出对象的功能。一个文件处理器负责上锁、打开、关闭、解锁和删除文件，以及读写数据。 ❑

(2) *确定变化的行为*。分析前一步中开发模型并确定应用程序的哪一些服务可能要发生变化哪些服务保持稳定。不存在描述系统中什么可以变化的一般规则。某一方面能否发生变化依赖于很多因素，如应用领域、应用的环境及其顾客与用户。在一个系统中可能发生变化的方面在其他的系统中可能保持稳定。下面的例子是经常发生变化的系统某些方面：

• 实时约束[HT92]，如截止期，时间防护协议和检测最终期限失效的运算法则。

204 • 事务协议[SW95]，例如记账系统中的乐观的和悲观的传输协议。

• 进程间通信机制[CM93]，如远程过程调用和共享存储。

• 例外情形中的行为[EKM+94]、[HT92]，如实时系统中的错过截止期的处理。

• 应用程序服务算法[EKM+94]，如具体国家的VAT计算。

开放实现分析和设计方法[KLLM95]对这一步很有帮助。

➥为保持持久性组件例子的简单性，我们未考虑应用行为的自适应问题。 ❑

(3) *当系统变更后，确定系统的不影响基本层次实现的结构方面*。例子包括一个应用程序的类型结构[BKSP92]，它所基于的对象模型[McA95]，或异构网络中的分布式组件[McA95]。

➥持久性组件的实现必须独立于特定应用的类型。这需要取得运行时的类型信息，比如每种类型的名称、大小、继承关系和内部布局，以及它们的数据成员的类型、顺序和名称等。 ❑

(4) *确定对如下两方面提供支持的系统服务*：一是步骤2确定的应用程序服务的可变性，二是步骤3确定的结构化细节的无关性。例如，实现C++中的可恢复的例外需要明确获得这种语言的例外处理机制。下面列出基本系统服务其他例子：

• 资源分配

• 无用单元收集

• 页面交换

205 • 对象创建

➥持久性组件在读持久性对象时必须能够实例化任意类型的类。 ❑

(5) *定义元对象*。为前三步中明确的每一方面定义适当的元对象。封装由几个领域无关的设计模式所支持的行为，如体现者[Zim94]、策略、桥接、访问者和抽象工厂[GHJV95]。例如，函数调用机制的元对象可作为策略对象来实现，组件的多重实现可用桥接模式来实现。访问者允许你集成新的功能而无需修改现存结构。有时你可能找到支持这一步的适当的邻域特定的模式，例如开发分布式系统的接受器和连接器（Acceptor and Connector）模式[Sch95]。另一个例子是

可分的检查器（Detachable Inspector）模式[SC95a]，它支持诸如调试器和审查者的运行期间设备的添加。可分析审查者建立在访问者模式之上。封装结构化信息和状态信息由诸如容体化器（Objectifier）[Zim94]和状态（State）[GHJV95]的设计模式所支持。

➡为持久性组件提供运行类型信息的元对象可按如下方式组织：

使用C++标准库类type_info来确定类型[DWP95]。它的接口提供获取类型的名称，比较两种类型，确定它们的系统内部顺序的功能。应用程序中的每种类型都由类type_info的一个实例来表示。

```
class type_info {
    //...
private:
    type_info(const type_info& rhs);
    type_info& operator=(const type_info& rhs);
public:
    virtual       ~type_info();
    int           operator==(const type_info& rhs) const;
    int           operator!=(const type_info& rhs) const;
    int           before(const type_info& rhs) const;
    const char*   name() const;
};
```

运行期间的类型信息系统的其他类都不是C++标准中的一部分。

一个类extTypeInfo提供了一种访问方法，以获取有关一个类的大小，超类和数据成员的信息。客户机程序也可以确定某类型是否是内置的或者是不是一个指针。 206

```
class extTypeInfo {
    // ...
public:
    const bool    isBuiltIn() const;
    const bool    isPointer() const;
    const size_t  size() const;
    baseIter*     bases(int direct = 0) const;
    dataIter*     data(int direct = 0) const;
};
```

方法bases()返回类baseIter的一个对象，该对象或者是一给定类型的所有基类上的迭代器或者只是它的直接基类上的迭代器。如果该类型是内置的，该方法返回一个NULL迭代器。类似地，方法data()返回类dataIter的一个对象。它或者在一个给定类型的所有数据成员之上迭代，其中数据成员包括通过继承得到的，或者只是在为这种类型具体声明过的数据成员之上迭代。如果该类型是内置的，则此方法返回一个NULL迭代器。

类BaseInfo提供了获得一个类的基类的类型信息的函数，也确定了它在类布局中的偏移。

```
class BaseInfo {
    // ...
public:
    const type_info*  type() const;
    const long        offset() const;
};
```

类DataInfo包括返回一个数据成员的名字，该数据成员的偏移及其相关的type_info对

象的函数。

```
class DataInfo {
    // ...
public:
    const char*        name() const;
    const type_info*   type() const;
    const bool         isStatic() const;
    const long         offset() const;
    const long         address() const;
};
```

207

(6) 定义元对象协议。支持元层次定义的和控制的修改和扩展，同时支持基本层次组件和元对象之间关系的修改。

实现元对象协议有两种选择：

- 将它与元对象集成在一起。每个元对象提供操作它的元对象协议的函数。
- 用一个独立的组件实现元对象协议。

后一种方法的优点是映射应用程序的每一次修改的控制定位在一个中心点。操作几个元对象的函数易于实现。此外，一个独立的组件可以保护元对象免遭非法的存取和修改，如果它的实现遵循诸如外观模式[GHJV95]或整体-部分模式这样的模式。单件惯用法[GHJV95]有助于保证元对象协议只被实例化一次。

如果作为一个独立的组件来实现，那么元对象协议通常不作为定义元对象类的基类——它只对它们进行操作。如果它运用于每个元对象，那么在把描述元对象协议成一个基类才有意义，这里从基类导出具体元对象类。

➡我们提供了一个类MOP，它为持久性组件例子中的元层次定义元对象协议。它作为一个单件来实现，并且可直接操作于前一步所表示的所有类的内部结构之上。

类型信息可由两个函数访问。

```
const type_info* getInfo(char* typeName) const;
const extTypeInfo* getExtInfo(char* typeName) const;
```

第一个函数允许客户机程序访问一个对象的标准类型信息。第二个函数访问扩展类型信息，这里的信息是我们专门为运行期间信息系统定义的。我们需要这个函数，因为标准类type_info的对象没有提供对用户定义信息的访问。所有其他的类型信息（诸如关于基类的）都是通过extTypeInfo对象来访问的。

208

新的类型信息元对象可由两个函数初始化，一个实例化type-info对象和另一个创建extTypeInfo对象。

```
void newTypeId(char* typeName);
void newTypeInfo(char* typeName,
                 bool builtIn, bool pointer);
```

newTypeInfo()函数也计算和设置一个类型的大小。如果没有的其他类包含对一个类型的对象的引用，函数deleteInfo()删除有关该类型的所有可用信息。

```
void deleteInfo(char* typeName);
```

我们为添加新的类型信息和修改现存类型信息定义了四个函数。函数addBase()和deleteBase()分别添加和删除基类信息，而函数addData()和deleteData()分别添加和删除数据成员信息。

```
void addBase(char* typeName, char* baseName);
void addData(char* typeName,
             char* memberType,char* memberName);
void deleteBase(char* typeName, char* baseName);
void deleteData(char* typeName, char* memberName);
```

执行变更之前，所有的函数进行一致性检查。例如，为了能够设置基类信息，相应的type_info和extTypeInfo对象必须是可用的。

有两个函数支持“对象创建器”元对象的修改。

```
void addCreationCode(char* typeName);
void deleteCreationCode(char* typeName);
```

在内部，元对象协议需要函数来计算类型大小及基类和数据成员的偏移。这些函数是与编译器相关的，因此在使用不同的编译器时必须被改变。策略模式[GHJV95]提供了一个支持改变这些函数的一种方法。为保持type_info和extTypeInfo对象，元对象协议保持了两个映射，tMap和eMap。这些映射提供了添加、删除和查找元素的函数。 209

元对象协议的绝大多数函数可以直接实现。计算偏移和大小以及操作“对象创建器”元对象需要在实现方面付出更多的努力。下面的代码定义了addBase()函数。

```
void MOP::addBase(char* typeName, char* baseName) {
    BaseInfo* base;
    // Is extended type information for type typeName
    // and type information for type baseName available?
    if ((!eMap.element(typeName)) ||
                        (!tMap.element(baseName)))
        // error handling ...

    // Instantiate the baseInfo object for type baseName
    base = new BaseInfo(tMap[baseName]);
    // Calculate the offset of the base class.
    base->baseOffset = calcOffset(typeName, baseName);
    // Add the new baseInfo object to the list of
    // bases within the extTypeInfo object for
    // type typeName
    eMap[typeName]->baseList.add(base);
}
```

❑

健壮性是在实现元对象协议时要主要考虑的因素。更该规格说明书出现的错误应该查出，只要有可能。变更也应该是可靠的。例如，添加新基类和数据成员时上述元对象协议检测相应的类型信息元对象的可用性。在删除其类型信息以前，它也检查一个类型是否用作一个基类或数据成员。

健壮性也意味着保持一致性。例如，如果我们向一个特定类型添加一个数据成员，我们必须重新计算所有类型的大小，这些类型包括作为一个基类或一个数据成员的正更改类型。此外，任何修改都应该只影响系统的发生变化的那些部分。最后，元对象协议的客户机程序没有义务

将变更集成到元层次。理想情况，一个客户机程序只描述一次变更，而且由元对象协议负责其集成。这就避免了对源代码的直接操作。

210

(7) 定义基本层次。根据步骤1开发的分析模型实现系统的功能核心和用户接口。

使用元对象保持基本层次的可扩展性和可修改性。将每个基本层次组件与提供系统信息的元对象连接起来，它们所依赖的系统信息，如类型信息，或者系统信息提供它们所需要的服务，如持久性组件的对象创建。为了处理系统服务，使用诸如策略、访问者、抽象工厂和桥接[GHJV95]的设计模式，或者诸如信封-信件（Envelope-Letter）[Cope92]惯用法。例如，策略模式的语境类组件表示基本层次组件和策略类谱系元对象。当应用访问者模式时，元对象是访问者，而且对象结构表示了基本层次组件。

向基本层次组件提供函数以维持与其相关的元对象的关系。元对象协议必须能够修改基本层次和元层次之间的任何关系。例如，当用一个新的元对象取代一个旧的元对象时，元对象协议必须更新对替换的元对象的所有引用。元对象协议或者是直接操作基本层次组件的内部数据结构，或者使用基本层次组件提供的一个特别接口。

如果不事先知道所要使用的元对象，那么用适当的检索函数提供元层次或元对象协议，比如持久性组件例子中的getInfo()和getExtInfo()函数。

元对象常常需要有关计算的当前状态的信息。例如，持久性组件例子中的“对象创建器”必须知道它应该实例化什么样的类型。这条信息或者可以作为一个参数传递给元对象，该元对象可以从其他元对象检索到它，或者该元对象可以从相应的基本层次组件检索到它。

元对象的改变影响与它们连接的基本层次组件伴随的行为。基本层次与元层次关系的改变只影响一个具体的基本层次组件，该组件维护修改的关系。

211

➡持久性组件中read()方法的实现遵循了“动态特性”小节描述的第一场景。该方法为从数据文件读对象实现了一个通用的递归算法。该方法咨询元层次以获得有关如何去阅读用户定义类型的信息。所读的内置类型或串被固化在它的实现之中。为了得到类型信息，read()咨询元对象协议的getInfo()和getExtInfo()函数。为了创建任意类型的对象，read()直接与“对象创建器”元对象连接。

store() 方法的结构类似于read()方法的结构。它先打开要读的数据文件，然后调用一个内部的storeObject()方法以存储对象结构。最后，store()关闭数据文件。

实现store()的最具挑战的部分是在待存储的对象结构中的循环检测——它在避免存储副本和运行进入无穷递归时非常重要。为了做到这一点，该方法以惟一的标识符标识该结构，在存储此对象之前也存贮该标识符。如果我们返回这样标识过的一个对象，那么我们只存储了它的标识符。

如下简化了的代码说明了storeObject()方法的结构。它对几个细节进行了抽象，比如静态数据成员的存储。

```
void Persistence::storeObject
                    (void* object, char* typeName) {
    type_info*          objectId;
    extTypeInfo*        objectInfo;
    baseIter*           iterator;
```

```
    // Get type information about the object to be stored
    objectId      = mop->getInfo(typeName);
    objectInfo    = mop->getExtInfo(typeName);
    iterator      = objectInfo->data();

    // Mark the object to avoid storing duplicates
    markObject(object);

    // Object is of built-in type?
    if (objectInfo->isBuiltIn())
        storeBuiltIn(object, objectId);

    // Object is of type char*?
    else if (!strcmp("char*", objectId->name()))
        storeString(object);

    // Object is a pointer != NULL?
    // *(char**)object means that we interpret the
    // generic pointer object as a pointer to an address
    else if ((objectInfo->isPointer()) &&
                        (!(*(char**)object)))
        // Dereference the pointer
        storeObject(*(char**)object,
            iterator->curr()->type()->name());

    // Object is a user-defined type with data members
    else while (!iterator->atEnd()) {
        // If not marked, store the data member,
        // else store the marker
        if (!marked((char*)object +
                iterator->curr()->offset()))
            storeObject((char*)object +
                iterator->curr()->offset(),
                iterator->curr()->type()->name());
        else
            storeMarker((char*)object +
                iterator->curr()->offset());

        iterator->next();
    };
    delete iterator;
};
```

212

❑

8. 已解决的例子

在前一小节我们解释了持久性组件例子中的映像体系结构。我们如何提供运行期间的类型信息的问题仍未解决。

不像语言CLOS和Smalltalk，C++并不能很好地支持映像——只有标准类type_info提供映射能力：我们可以确定和比较类型。提供扩展类型信息的一种解决方案是在编译过程中包含一个特别步骤。在这个编译过程中，我们从应用程序的源文件中收集类型信息，为实例化元对象产生代码，并把该代码连接到应用程序。类似产生“对象创建”元对象。用户为实例化每种类型的一个“空”对象指定代码，工具箱为元对象产生代码。系统的某些部分是独立于编译器的，如偏移和大小的计算。

213

正如在代码例子中说明的那样，我们使用类型和数据成员的指针及地址算法、偏移和大小去读出和存储对象。由于这些特征被认为是有害的，例如招致过多地写对象代码的危险，因此持久性组件必须非常仔细地去实现和测试。

9. 变体

带几个元层次的映像。有时元对象之间相互依赖。例如，考虑持久性组件。改变某个特定类型的运行期间的类型信息需要你更新“对象创建器”元对象。为了协调这些变化，你可以引入独立的元对象，以及概念上的元层次的一个元层次，或者换一句话说，一个元元层次。理论上这将导致一个无穷的映像塔。这样的软件系统有无穷多个元层次，其中每一个元层次都由它上层的元层次控制，每一个元层次都有它自己的元对象协议。实际上，绝大多数现存的映像软件只有1~2个元层次。

带有几个元层次的编程语言的例子是RbCl[IMY92]。RbCl是一个解释性语言。RbCl的基本层次对象由几个元层次对象表示。这些元层次对象由驻留在RbCl的元元层次的解释器来解释。RbCl的元对象协议允许用户修改元对象，这里的元对象表示RbCl基本层次对象，表示RbCl元对象的解释器的元元层次行为元对象协议。

10. 已知应用

CLOS。这是一个映像程序设计语言[Kee89]的经典例子。在CLOS中，为对象定义的操作称为*类属函数*（*generic functions*），它们的处理就是类属函数调用。类属函数调用分成三个阶段：

- 系统首先确定可用于一个给定调用的所有方法。
- 然后按优先序降序排列所有这些可用的方法。
- 最后系统顺序执行可用方法的列表。注意，CLOS中对应于一个给定调用，可能执行不止一种方法。

214

CLOS的元对象协议中定义了类属函数调用的过程[KRB91]。从根本上说，它执行了元层次类属函数的一个确定的序列。贯穿CLOS元对象协议，用户可以修改这些类属函数或它们调用的元对象的类属函数来改变一个应用程序的行为。

MIP[BKSP92]是C++的一个运行期间类型信息系统。它主要用来对一个应用程序的类型系统进行自省存取。一个C++软件系统的每种类型由一个元对象集合来表示，而这里元对象提供有关该类型，该类型与其他类型的关系以及它的内部结构的一般信息。所有信息在运行期间都是可获得的。MIP的功能被分成四层：

- 第一层包括能使软件明确和比较类型的信息和功能。这一层对应于C++的标准运行期间类型标识能力［SL92］。
- 第二层提供有关应用程序的类型系统的更详细信息。例如，客户机程序可以获得有关类的继承关系信息，或者有关它们的数据和函数成员的信息。这些信息可以用来浏览类型结构。
- 第三层提供了有关数据成员的相对地址的信息，提供了创建用户定义类型的“空”对象的函数。为了同第二层联合在一起，这一层支持对象I/O。
- 第四层提供完整的类型信息，如关于一个类的友元，数据成员的保护，或函数成员的变量和返回类型。这一层支持灵活的进程件通信机制的并发，或像审查者那样的工具的开发，

这需要有关一个应用程序的类型结构的非常详细的信息。

MIP的元对象协议可让你描述和修改提供运行期间类型信息的元对象。它为MIP功能的每一层提供了合适的函数。

MIP是作为一个库类集实现的。它也包括了一个收集有关应用程序类型信息的工具箱，并实例化相应的元对象生成代码。这些代码被链接到使用MIP的应用程序并在主程序开始时被执行。工具箱可以与开发包为C++应用程序的“标准”编译过程集成在一起。一个特定的接口允许用户为每个单独的类或类型度量可用的类型信息。 215

PGen[THP94]是一个为C++做的基于MIP的持久性组件。它允许一个应用程序存储和读任意C++对象结构。

用来解释映像模式的例子主要基于MIP和PGen。尽管持久性组件的描述作了简化，但是元对象的类声明和元对象协议充分地反映了MIP和PGen的原始结构。

NEDIS。小汽车销售商处理系统NEDIS[Ste95]使用映像来支持它的针对特别客户和特别地区的特定需求的调整。NEDIS包括一个称为运行期间数据字典的元层次。它提供了以下服务和系统信息：

- 类的具体属性的性质，如它们的允许值的范围。
- 检查与它们所需性质相比的属性值的函数。NEDIS使用这些函数估计用户输入，例如确认一个日期。
- 类属性的默认值用来初始化新对象。
- 在出错事件中描述系统的行为的函数，如属性无效输入或意外的“零”值。
- 特殊地区的功能，如税款计算。
- 有关软件的“视觉和感觉”的信息，如输入界面的布局或在用户界面中使用的语言。

运行期间数据字典作为一个持久性数据库来实现。一个特别的接口可以使用户修改它所提供的任何服务和信息。一旦运行期间数据字典发生了变化，那么就采用特别的工具检查，并最终重建其一致性。运行期间数据字典在开始运行软件时加载。由于安全的原因，NEDIS在运行时不能被修改。 216

OLE 2.0[Bro94]提供了揭示和获得有关OLE对象和及其接口类型信息的功能。这些功能可以被用来动态地获得OLE对象的结构化信息，并创建OLE接口的调用。例如，Visual Basic[Mic95]的运行环境在动态地调用一个对象前检查调用一个对象的方法的正确性。Corba[OMG92]描述了一个类似的概念。

使用映像体系结构的语言和例子的进一步例子包括Open C++[CM93]、RbCl[IMY92]、AL-1/D[OIT92]、R2[HT92]、Apertos[Yok92]和CodA[McA95]。甚至在[IMSA92]中可以找到更多的例子，但要注意，尽管所有的例子提供了映像能力，并不是所有的例子真正像这个模式描述的那样实现了映像体系结构。

11. 效果

映像体系结构有以下的**优点**：

不直接对源代码进行修改。在修改一个映像系统时，你不需要接触现有源代码。相反，你可以通过调用元对象协议的一个函数来描述一个变化。在扩展一个软件时， 你向元层次传递的

新代码作为元对象协议的一个参数。元对象协议本身负责集成你的变更请求：它执行元层次代码的修改和扩展，并且，如有必要的话，重新编译变化了的部分，在它在执行时将它们链接到应用程序。

*更改一个软件系统变得容易了。*元对象提供了更改软件的一个安全和统一的机制。它对用户隐藏了诸如访问者、工厂和策略的使用的所有的具体技术。它也隐藏了一个可变化应用程序的内部复杂性。用户不直接面对封装了特定系统方面的大量的元对象。元对象协议也控制每一个修改。一个设计良好的、健壮的元对象协议有助于预防应用程序的基本语义的无计划变更[Kic92]。

217 *支持许多种类的变更。*元对象可以封装系统行为、状态和体制的每一个方面。因此，一个基于映像体系的结构潜在地支持几乎所有种类或规模的变更。甚至基本系统方面也可以被改变，如函数调用机制或类型结构。在映像技术的帮助下，调整软件以去满足环境的具体需要或集成特殊顾客的需求，也是可以做到的。

但是，映像体系结构也有一些显著的**不足**：

*在元层次修改可能会带来故障。*即使最安全的元对象协议也不能阻止用户描述不正确的修改。这些修改可能引起对软件或其环境导致严重破坏。危险修改的例子包括改变数据库模式而没有在使用它的应用程序中终止该对象的执行，或把代码传递给包含语义错误的元对象协议。类似地，指针运算中的隐错可能引起对象代码被重写。

因此，一个元对象协议的健壮性就特别重要[Kic92]。变更规格说明中的潜在错误应该在变更执行之前被检测到。每次变更对软件的其他部分只能有局部影响。

*增加了组件的数目。*一个映像软件系统可能包括比基本层次组件多的元对象。在元层次上封装的因素越多，元对象便越多。

*较低的效率。*映像软件系统通常比非映像系统要慢。这是由基本层次和元层次之间的复杂关系引起的。当基本层次不能决定如何继续计算时，它向元层次寻求帮助。这种映像能力需要额外处理：信息检索，改变元对象，一致性检查，并且两层之间的通信降低了系统的整体性能。你可以通过优化技术部分减轻这种性能负面影响，比如编译系统时直接把元层次代码注入基本
218 层次。

*并非软件的所有潜在变更都得到支持。*尽管一个映像体系结构有助于可变更软件的开发，但是只有通过元对象协议实现的变更才被支持。结果，它不可能简便地把所有未预料到的变更集成到一个应用程序，例如对基本层次代码改变或扩展。

*并非所有语言都支持映像。*一个映像体系结构在诸如C++的某些语言中难以实现，这类语言很少支持甚至不支持映像。C++仅提供了类型标识。C++中的映像应用程序通常建立在诸如处理任意对象的指针运算那样的语言构件上，而且需要动态修改元层次代码的支持工具。但是，这是单调乏味且易于出错的。在这样的语言中发挥出映像模式的全部功能也是不可能的，比如向一个类中动态添加新方法就不可能。但是，即使在一个没有提供映像能力的语言中，也可能建立可变更和可扩展的映像系统，如C++系统NEDIS[EKM+94]、MIP[BKSP92]和Open C++[CM93]。

参见

<u>微核</u>体系结构模式通过用额外的或客户特定的功能为扩展软件提供的一个机制来支持调整

和变更。这种体系结构的中心组件——微核——作为插入这类扩展和协调它们的协作的一个套接字。修改可以通过更换这些“可插入”部件来实现。

这种模式的一个较早版本出现在[PloP95]中。

致谢

关于映像的最早论著之一是Brian Cantwell Smith的博士论文[Smi82]。该论文在过程语言的语境中描述了映像。有关映像概念的概述参见[Mae87]。

我们感谢PLoP'95工作组1的成员，尤其是Douglas C. Schmidt和Aamod Sane，他们为该模式的早期版本的改进提出了宝贵的批评和建议。我们也非常感谢AG 通信系统的Linda Rising和David E. DeLano，以及地处Urbana Champaign的伊利诺伊大学的Brian Foote和Ralph Johnson。他们对该模式的早期版本的详细评论有助于形成这里的描述。

219

第3章 设计模式

我们都知道设计经验的价值。有多少次设计过程中，我们都有一种似曾相识的感觉，即觉得自己以前已经解决了某个问题，但又不知道确切的地点或具体的解决方法？如果能记得先前的问题和它的详细解决方法,就能复用先前的经验而不需要重新发现它。

Erich Gramma等4人合著,《设计模式——可复用面向对象软件的基础》

一个设计模式描述通信组件的一种通用-重现结构，该结构能在特殊语境中解决一般的设计问题[GHJV95]。

本章，我们给出八种设计模式：整体-部分、主控-从属、代理、命令处理器、视图处理程序、转发器-接收器、客户机-分配器-服务器、出版者-订阅者。

221

3.1 引言

设计模式是中等规模的模式，在规模上它比体系结构模式要小，但在级别上比特定编程语言的惯用法要高。设计模式的应用不会影响软件系统的基本结构，但对子系统的体系结构会产生很大影响。

我们将设计模式分成几类相关模式，就像我们对体系结构模式分类一样：

- 结构化分解。此类别包含的模式支持将子系统和复杂组件适当地分解为相互合作的部分。整体-部分模式是我们在此类别中能体会到的最普遍的模式。它被广泛应用于构造复杂组件。
- 工作的组织。此类别包含的模式定义了组件之间如何协作来共同解决复杂问题。我们描述的主控-从属模式有助于构成服务所需的容错或计算精度的计算。它同时支持将服务分解成相互独立的部分并能并行执行。
- 访问控制。这种模式保护和控制对服务或组件的访问。我们在这里描述的代理模式允许客户机与组件的代表通信而不是与组件本身通信。
- 管理。此类别包含的模式将同类对象集、同类服务集、同类组件集视为一个整体来处理。我们描述两种模式：命令处理器模式处理用户命令的管理和协议，而视图处理程序模式描述在一个软件系统中如何管理视图。
- 通信。此类别中的模式有助于构成组件间的通信。有两种模式用于处理进程间通信问题：转发器-接收器模式处理对等通信，客户机-分配器-服务器模式描述在一个客户机-服务器结构中的位置透明通信。

222

出版者-订阅者模式有助于合作组件之间保持数据一致性任务。出版者-订阅者模式直接对

应文献[GHJV95]中的观察者模式。因此，我们只给出这种模式的精髓，并着重描述出版者-订阅者模式的一种重要变体：事件通道。

本章包含的设计模式仅仅涵盖了设计软件系统时可能发生问题的一小部分。我们可以并且也应该用更多的设计模式来扩充现有模式集，例如在[GHJV95]中的那些设计模式。如果更多的设计模式被加进来，可能在组织它们时有必要定义新的类别。在第5章“模式系统”中，我们对这个主题进行了详述。

所有设计模式都共有的一个重要特性是它们独立于特殊的应用领域。它们涉及应用程序功能的构造，而不涉及这些应用程序功能本身的实现。

大多数设计模式独立于特殊的程序设计范型。通常，它们能够方便地用面向对象方式实现，但是我们的所有设计模式已经足够通用，以至于能够适合于更加传统的程序设计实践，如过程型程序设计。

3.2 结构化分解

如果从结构上把子系统和复杂组件分解为较小的独立组件，而不将其保留成单一集成的大块代码，操作起来会比较容易。这样，改动易于完成，扩展易于集成，而设计也更加易于理解。 223

在本节中，我们将描述支持组件结构化分解的设计模式：

- 整体-部分（*Whole-Part*）设计模式有助于聚合组件形成一种语义单元。一种聚合组件（即整体），封装其构成组件（即部分），组织它们之间的协作，并为其功能提供一个公共接口。直接访问各个部分是不可能的。

整体-部分模式有着广泛的应用。几乎每一个软件系统都包含组件，甚至整个子系统也可以通过使用该模式来进行组织。包含关系的层次结构，在其某一变体中尤其适合于使用整体-部分设计模式。

另一个大家熟知的、有助于结构化分解的设计模式是组合[GHJV95]。

- 组合（*Composite*）模式将对象组织为树型结构，表示部分-整体层次。组合允许客户机与独立对象和组合对象进行同样交互。

注意，像整体-部分模式和组合模式这样的模式没有提供特定子系统或组件的结构化分解。仍需根据正在开发的应用程序的需求，详细说明一个组件结构中的参与者（participant）。

这些模式仅仅提供子系统和复杂组件分解的一般技术。例如，组合描述如何建立层次结构，从而允许客户机忽略组合对象与独立对象在层次中的差别。

在此类别中的各种模式详细说明了如何实现组件间的特殊关系，比如组装-部分或容器-内容。它们也详细描述了在这种结构中的特殊组件必须具备的几种职责。 224

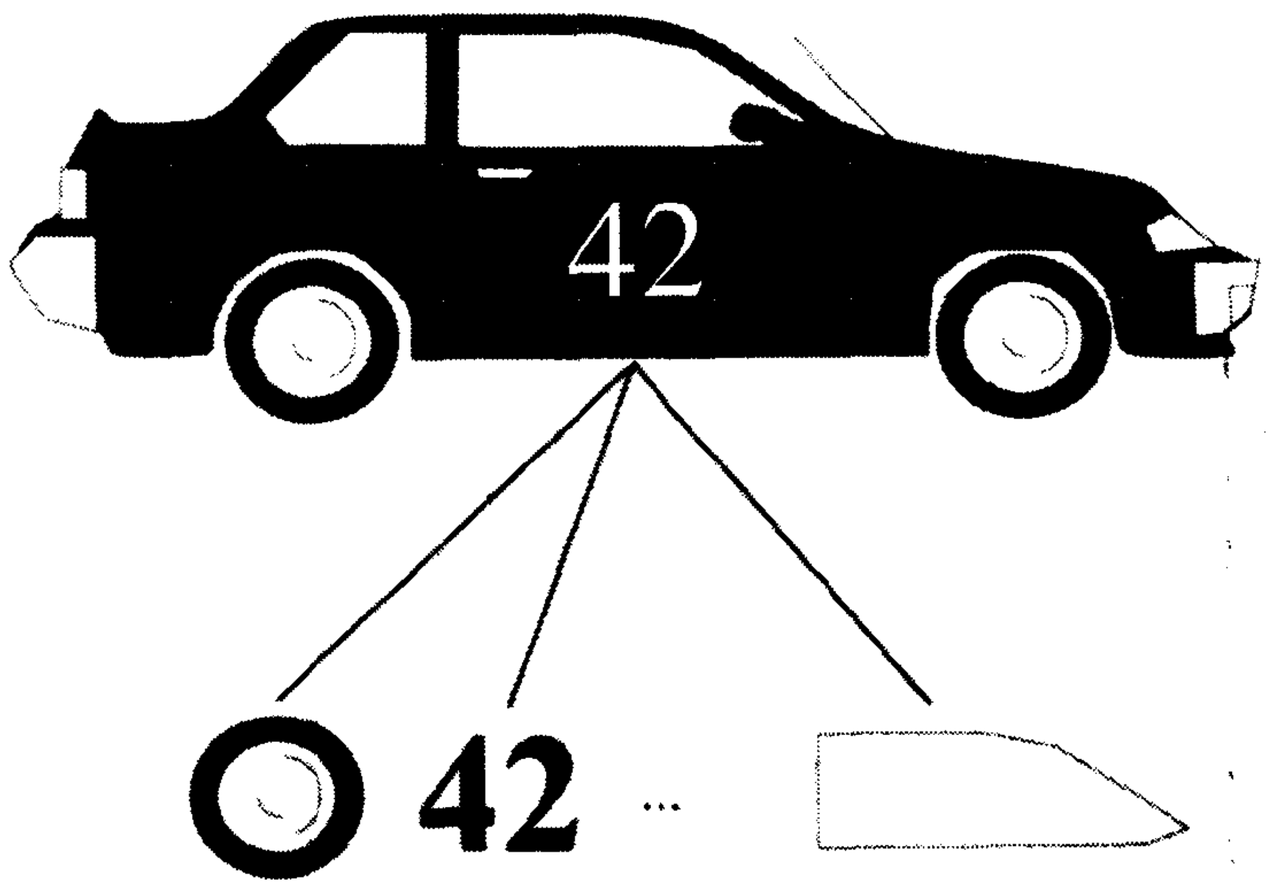
3.2.1 整体-部分

整体-部分（*Whole-Part*）设计模式有助于聚合组件形成一种语义单元。一种聚合组件（即

整体）[⊖]，封装其构成组件（即部分），组织它们之间的协作，并为其功能提供一个公共接口。直接访问各个部分是不可能的。

1. 例子

一个用于二维或三维建模的计算机辅助设计系统（CAD），允许工程师们交互地设计图形对象。在这样的系统中，大多数图形对象被设计为其他对象的组合。例如，一个小汽车对象聚合几个较小的对象，例如轮子和窗户，它们本身又由更小的对象组成，例如圆形和多边形。将汽车视为一个整体，汽车对象负责实现对汽车操作的功能，如旋转或绘制。



225

2. 语境

实现聚合对象。

3. 问题

在几乎所有的软件系统中，都存在由其他对象组合而成的对象。例如，考虑一个化学仿真系统中的一个分子对象，它可以实现为一个分离的原子对象图形。这种聚合对象不能表示松散耦合的组件集合。相反，它们形成的单元大大超过它们各个部分的集合。在这个例子中，分子对象将具有诸如其化学性质这样的属性，以及诸如旋转这样的方法。这些属性和方法把分子（而不是把组成分子的单独原子）当做一个语义单元。分子示例说明了一种典型情况，在其中，聚合揭示了单个部分不明显或不可见的行为——各个部分的组合导致新的行为出现。这种行为称为突发行为（*emergent behavior*）。例如，考虑到一个分子能参与的化学反应——不能仅通过分析它的单个原子来确定化学反应。

当我们为这样的结构建模时，必须权衡下面的强制条件：

- 一个复杂的对象要么应该被分解成较小的对象，要么由现存的对象组成，以便在其他聚合类型中支持可重用性、可变更性和构成对象的可重组性。

⊖ 在这个描述中，模式所涉及的名字都采用楷体表示，例如“*整体*”，以区别于名词“整体”和名为“*整体*”的组件。

• 客户机应该把聚合对象视为原子对象，不允许对其构成部分进行任何直接访问。

4. 解决方案

引入一个封装了较小对象的组件，并防止客户机直接访问这些构成部分。访问该封装对象功能的惟一方法是通过为该聚合体定义一个接口，允许聚合体作为一个语义单元出现。

整体-部分模式的一般原则适用于三种关系类型的组织：

• 组装-部分（*assembly-parts*）关系，用于区分一个产品与其组成部分或者子配件——例如我们前面例子中的分子和原子之间的关系。所有的组成部分根据组装的内部结构紧密地集成在一起。子配件的数量和类型被事先定义，并不能被更改。 226

• 容器-内容（*container-contents*）关系，其中聚合对象表示为一个容器。例如，一个邮包可以包含不同的内容，比如一本书、一瓶酒或一张生日贺卡。与组装-部分关系中的部分相比，这些内容较为松散。甚至可以动态地添加或删除内容。

• 收集-成员（*collection-member*）关系，有助于相似对象成组——例如一个组织和它的成员的关系。该收集提供诸如对其成员进行迭代和对成员执行操作的功能。不区分收集的单个成员——均被同等对待。

这些关系模仿真实世界中对象间的关系。当用软件实体对它们建模时，哪种关系更适合并不总是很明显。一个分子可以视为一个由不同原子组成的组装部件，也可以视为一个容器，其内容是原子。哪一种关系更合适，由聚合体所希望的使用决定。

注意到这些类别定义了对象间的关系而不是数据类型间的关系是十分重要的。

5. 结构

整体-部分模式引入两种参与者类型：

整体对象表示一些我们称为部分的较小对象的聚合。它形成其组成部分的一个语义分组，协调和组织它们的合作。基于这个目的，整体利用部分对象的功能来实现服务。

整体的某些方法可能仅仅是为特殊的部分服务设置占位符。调用这种方法时，整体仅调用相关的“部分”服务，并向客户机返回结果。

➥CAD系统中的每个图形对象都包含向用户提供版本信息的一个部分。当客户机调用方法`getVersion()`时，该请求就被转发到部分的适当方法。 ❑ 227

整体的其他服务实现了构建在由部分提供的几个较小服务上构建的复杂策略。

➥考虑缩放一组二维图形对象。为此，要确定所有组内成员的最小环绕矩形。计算这些矩形的并集，将得到组本身的最小环绕矩形。其中心代表了缩放操作的中心。为了完成缩放方法的执行，组对象调用所有部分的缩放操作，把缩放中心和缩放百分比作为参数传递。 ❑

整体可以额外提供不调用任何部分服务的功能。

➥考虑到诸如集合这样的收集的实现。设置对象提供诸如`getSize()`的功能，以便返回当前包含的元素的数目。考虑到性能因素，`getSize()`可以通过引入缓存策略来实现。额外的数据成员`size`存储集合中当前元素的大小。一旦添加或删除元素，`size`的值也相应地进行更新。如果客户机调用`getSize()`，则函数返回`size`的值，而不需要访问集合中的任何元素。 ❑

对外部客户机来讲，只有整体的服务是可见的。整体还充当一个包装器（wrapper），它围绕其组成部分，保护它们免遭未经受权的访问。

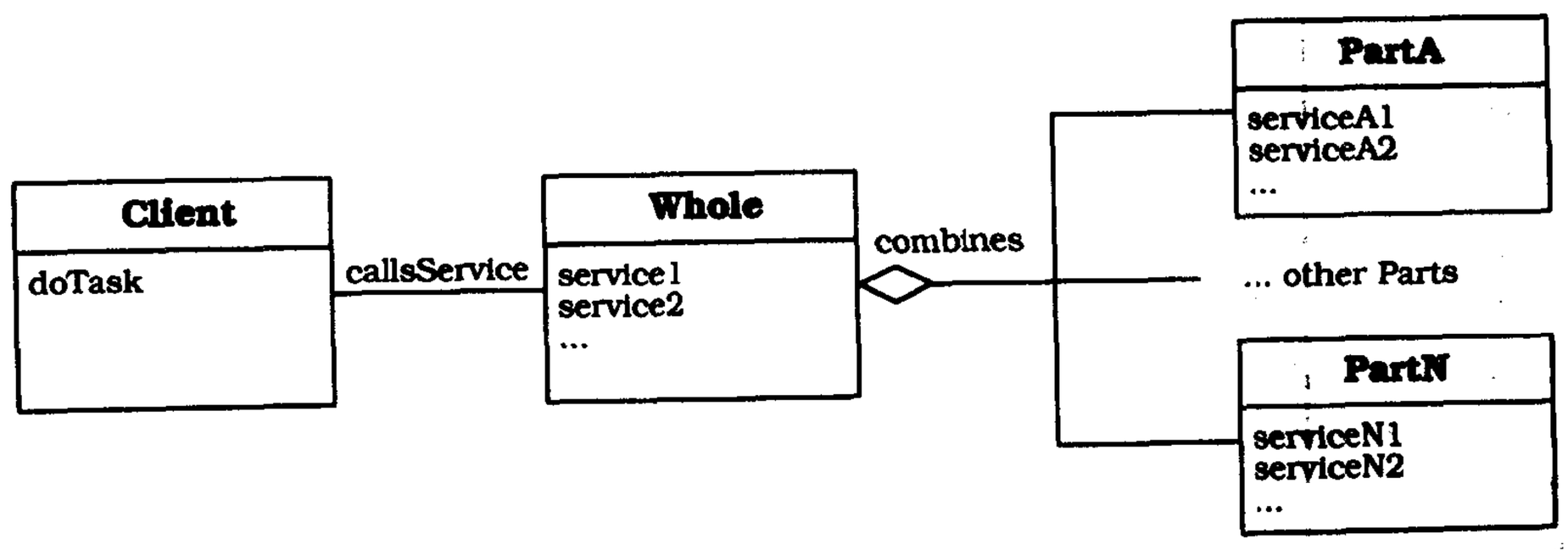
每个部分对象恰好被嵌入在一个整体中。两个或两个以上的整体不能共享同一个部分。每个部分的创建或销毁在其整体的生命周期内完成。

类 整体	协作者 • 部分
责任 • 聚合几个较小的对象 • 提供建立在部分对象顶部上的服务 • 围绕其组成部分充当一个包装器	

类 部分	协作者 –
责任 • 表示一个特殊对象及其服务	

228

如下的OMT图描述了整体及其部分的静态关系：



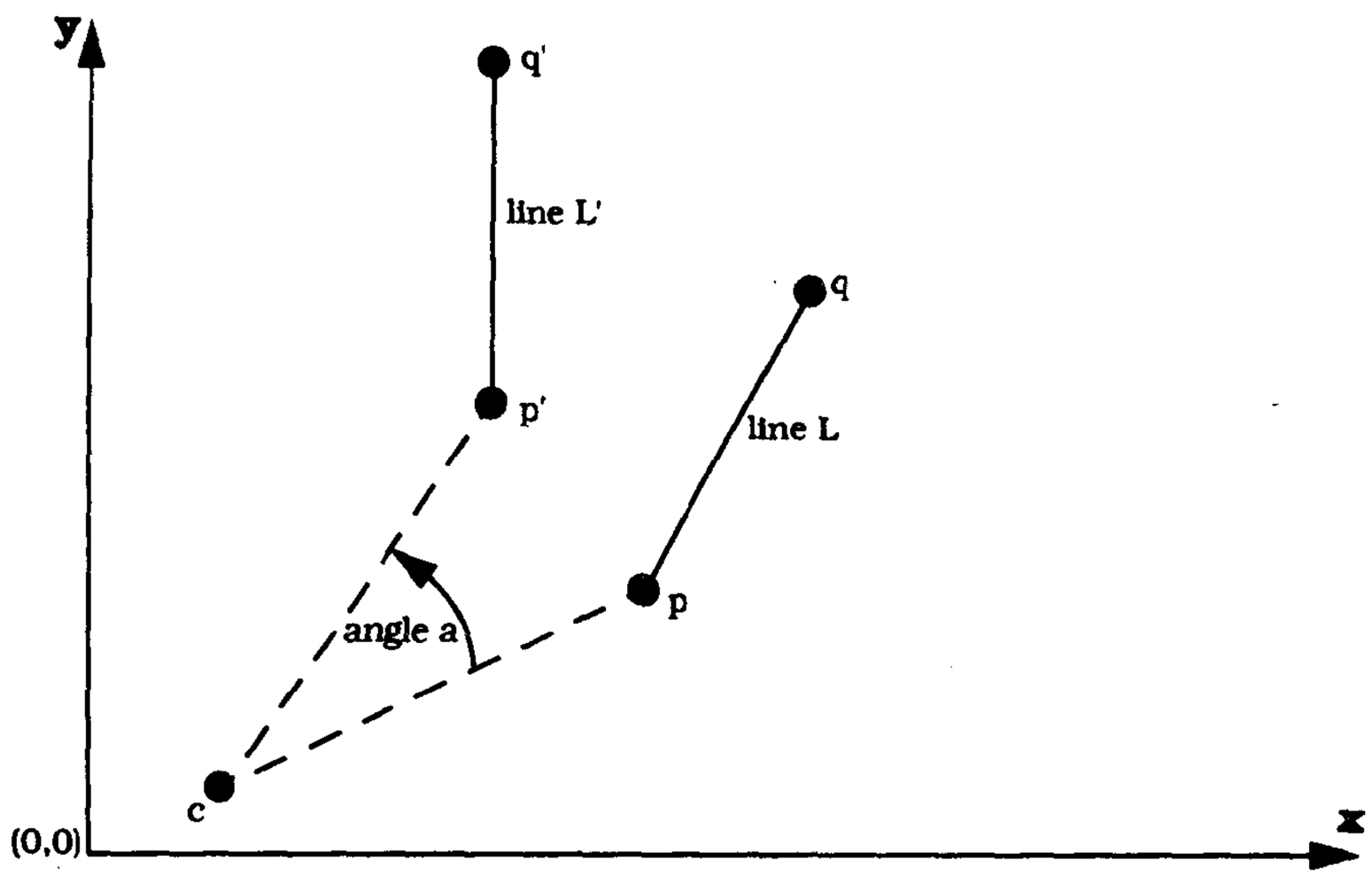
6. 动态特性

下面的场景描述整体–部分结构的行为。我们把一个CAD系统中线段的二维旋转作为例子。线段作为整体对象，它包含的两个点p和q作为部分。客户机要求线段对象绕c点旋转，旋转的角度作为参数传递。因为线段的旋转可以基于单点旋转，所以线段对象只需调用它的端点旋转就足够了。旋转后，线段在屏幕上重新绘制自己。为简洁起见，该场景并未说明旧的线段是如何从屏幕上被删除，也未说明drawLine（画线）方法是如何检索到新端点的坐标。

p点绕中心c以角度a旋转，可以用以下公式计算：

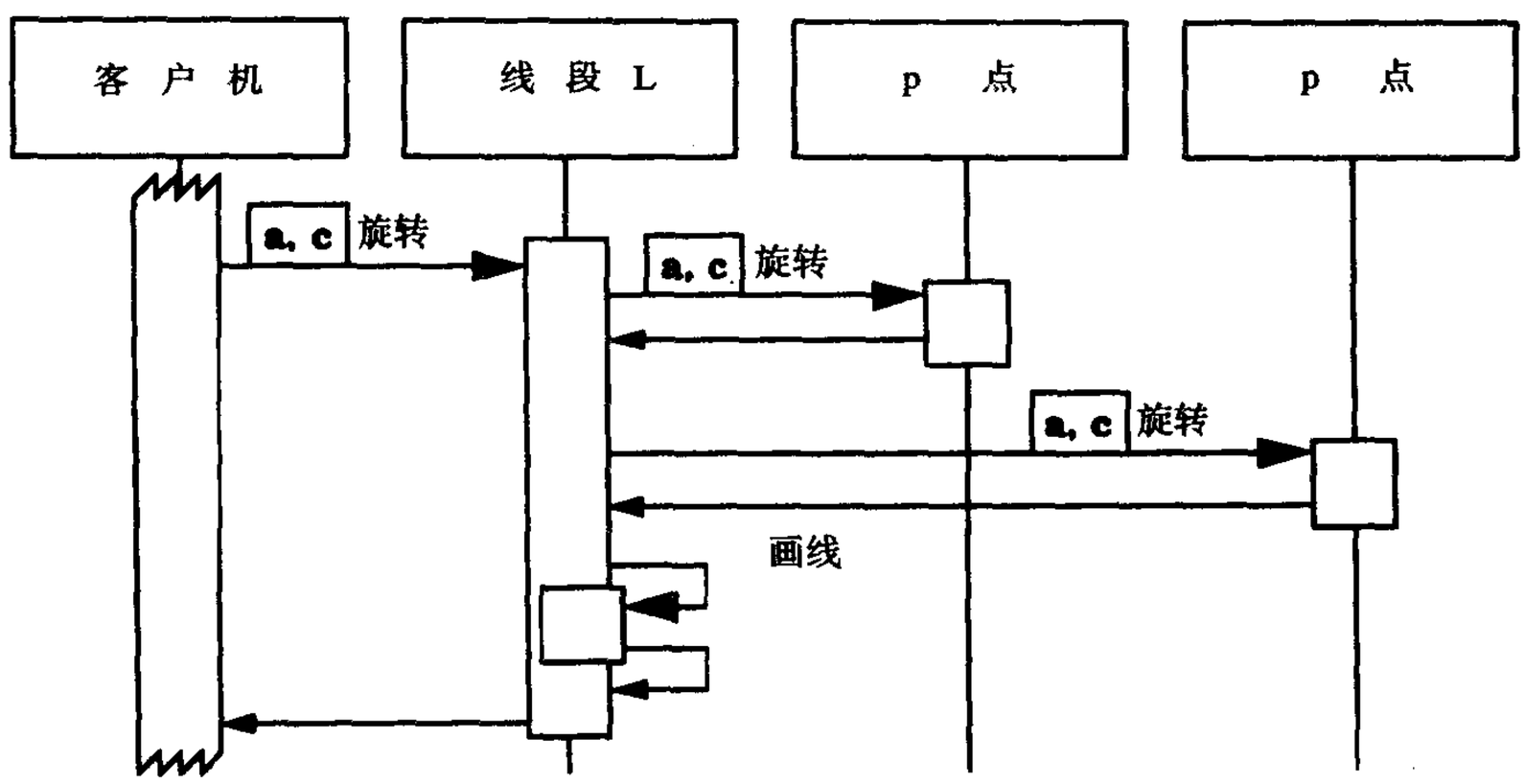
$$\vec{p}' = \begin{bmatrix} \cos a & -\sin a \\ \sin a & \cos a \end{bmatrix} \cdot (\vec{p} - \vec{c}) + \vec{c}$$

229 在下图中，说明了p点和q点组成线段的旋转。



该场景由四个阶段组成：

- 客户机调用线段L的rotate（旋转）方法，角度a和旋转中心c作为参数传递；
- 线段L调用p点的rotate方法；
- 线段L调用q点的rotate方法；
- 用p′和q′作为线段L端点的新位置重绘自己。



230

7. 实现

为了实现整体-部分结构，应用以下步骤：

(1) *设计整体的公共接口*。分析整体必须提供给客户机的功能。在这个步骤中仅仅考虑客户机的观点。把整体看作是没有分解成部分的原子组件，编译共同组成整体的公共接口的方法列表。

(2) 整体分解为部分，或者从现存的部分中合成。有两种方法聚集所需要的部分——一种是从现存的部分自下而上聚集成一个整体，另一种方法是将整体自上而下分解为较小的部分：

- 自下而上的方法允许从松散耦合的部分组成整体，在以后实现其他类型的整体时将重用这些部分。这种方法的缺点在于用现存部分来覆盖整体所需功能的各个方面的困难性。结果，不得不经常实现“粘贴”来填补部分的合成和由整体提供的接口之间的缺口。
- 自上而下的方法使覆盖整体的所有功能成为可能。划分为部分是由于受整体为其客户机提供服务的驱动，避免了实现粘贴代码的需求。然而，严格地应用自上而下的方法经常导致部分紧密耦合而不能在其他语境重用。

经常利用两种方法的混合。例如，可以先遵循自上而下的方法直到产生的结构能重用现有部分为止。

(3) 如果沿用自下而上的方法，使用组件库或类库中现存的部分并详细描述它们之间的协作。如果利用现存的部分不能覆盖整体的所有功能，则需要详细说明附加的部分，以及其与剩余部分的集成。也许需要使用自上而下的方法来实现这些缺少的部分。

231

(4) 如果沿用自上而下的方法，将整体的服务分解为较小的协作服务，并将这些协作服务映射到分离的部分。例如，在转发器-接收器设计模式中，一个转发器组件负责汇集IPC信息，并将它们分发给接收器。因此，可以将一个转发器分解成两个部分，一个负责汇集而另一个负责信息的发送。

注意，将整体分解为部分通常有几种方法。例如，一个三角形可以被说明为不在同一直线上的三个点，或者三条直线，或者一条直线和一个点。根据经验，选择提供实现整体服务的最容易的方法作为分解策略。例如，如果打算将隐藏线段算法应用于三角形，就应该将它们作为线段的组成部分来实现。

(5) 用部分的服务来详细说明整体的服务。在前两个步骤所建立的结构中，整体被表示成协作部分的集合，它们分别有独立的职责。必须详细说明整体使用哪个部分功能来对客户机的请求提供服务，哪个请求执行它自身的服务。

调用部分服务有两种可能方法：

- 如果客户机的请求被转发给部分服务，部分不需要使用整体执行语境的任何知识，只需依赖它自身环境。这种转发导致整体与部分之间的松散耦合。甚至可以被当作运行在不同进程中的活动对象来实现。
- 委派方法要求整体将它自己的语境信息传递给部分。当需要把部分紧密嵌入在整体的环境中时，委派十分有用。例如，必须模拟部分与整体之间的实现继承时，需要委派。

决定所有的部分服务是否仅仅只能被它们的整体调用，或者部分之间是否也可以互相调用。通常部分由它们的整体激活。然而，有时部分之间需要交互。例如，考虑诸如表示天文星系集合的整体这样的一个模拟对象。如果需要确定该星系的运动，仅仅只考虑“大爆炸”是不够的——还必须把星系之间的引力考虑进来。解决这个问题需要多种方法，在这些方法中，部分之间互相交互。另外一个例子可以由链表来说明，在链表中每个元素包含对其相邻元素的引用。

232

可以在中介者（Mediator）设计模式[GHJV95]中找到关于部分之间交互的更进一步的讨论。

(6) 实现部分。如果部分自身是整体-部分结构，转到步骤1递归设计。如果不是，从库中重

用现存的部分，或者如果它们的实现是直截了当的并且不需要更进一步的分解，就只实现它们。

(7) 实现整体。基于在前面步骤中开发的结构实现整体的服务。通过从整体中请求它们的服务来实现依赖于部分对象的服务。还必须在这一步中实现不依赖于部分对象的服务。

实现整体时，必须考虑所有给定的限制，比如基数性质。例如，一个水分子恰好由两个氢原子和一个氧原子组成。在部分之间同样存在约束。考虑一个邮包对象及其内容——内容的大小不可能超出邮包的大小。

还必须管理部分的生命周期。因为部分随着其整体产生也必须随着其整体而消失，所以整体必须负责建立和删除部分。

"已解决例子"小节提供了实现整体-部分结构的具体例子。

8. 变体

共享部分。该变体放宽了每个部分必须与一个确定的整体相关联的限制，允许多个整体共享同一个部分。被共享的部分的寿命与其整体的寿命相耦合。例如，考虑由一个邮件头和几个附件组成的电子邮件信息。该信息的接收器能够解压缩附件并把它们打包成新的信息。即使原始的信息被删除了，它的部分——附件——依然存在。在这种情况下，用部分本身或者一个中心管理组件，来负责管理部分的生命周期。在诸如C++这样的程序设计语言中，可以通过引用计数策略来达到此目的——这将在计数指针惯用法中解释。 233

接下来的三个变体描述了我们在前面"解决方案"小节引入的整体到部分之间关系的实现：

组装-部分。在这个变体中，整体可能是一个表示较小对象组装的对象。例如，小汽车的CAD表示可能是轮子、窗户、车身面板等的组装。组成部分可以递归地按组装-部分关系分解——一个轮子本身可能是由诸如圆这样的部分组成的一个整体。递归地应用整体-部分关系将得到一棵树，并且如果允许共享的部分也可能得到有方向的无循环图。组装-部分结构是固定的，因此在执行期间，它不支持对部分的添加或删除。它们仅仅支持同类型的两个部分的交换。

容器-内容。在这个变体中，一个容器负责维持不同的内容。例如，一个电子邮件信息可能包含一个邮件头、邮件体和可供选择的附件。与组装-部分变体相比，一个容器组件可以动态地添加或删除其内容。

收集-成员变体是容器-内容变体的一个特例，其中部分对象都有同样的类型。部分彼此间的联系通常不紧密，也不相互依赖。当实现诸如集合、列表、映像和数组等汇集时，可以应用该变体。此外，这种模式支持在所有成员上完成迭代功能和在部分或全部成员上执行操作的功能。

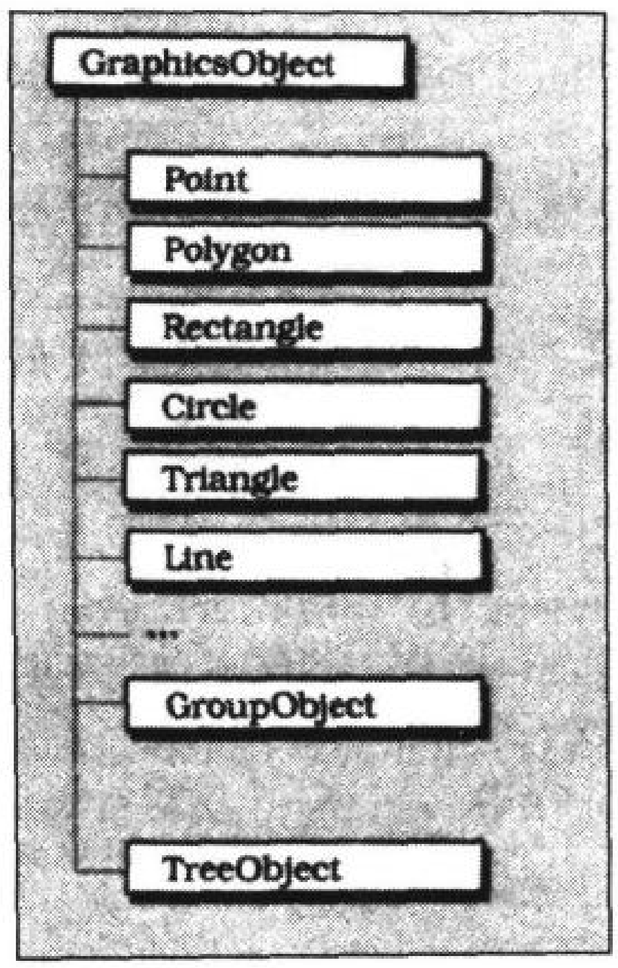
在文献[GHJV95]中引入了组合模式。它可以应用于整体-部分层次，在其中，整体和部分可以同等对待——也就是说，两者都实现了同样的抽象接口。

9. 已解决的例子

在CAD系统中，我们决定定义一个Java软件包为图形对象提供基本的功能。类库由诸如圆、直线等原子对象组成，用户可以利用这些原子对象组成更为复杂的实体。我们直接实现这些类而不是使用标准的Java包`awt`（抽象开窗工具箱，Abstract Windowing Toolkit），因为`awt`不提供我们所需要的功能。 234

对象使用虚拟坐标而不是物理的屏幕坐标来隐藏诸如屏幕分辨率等系统附加属性。抽象的

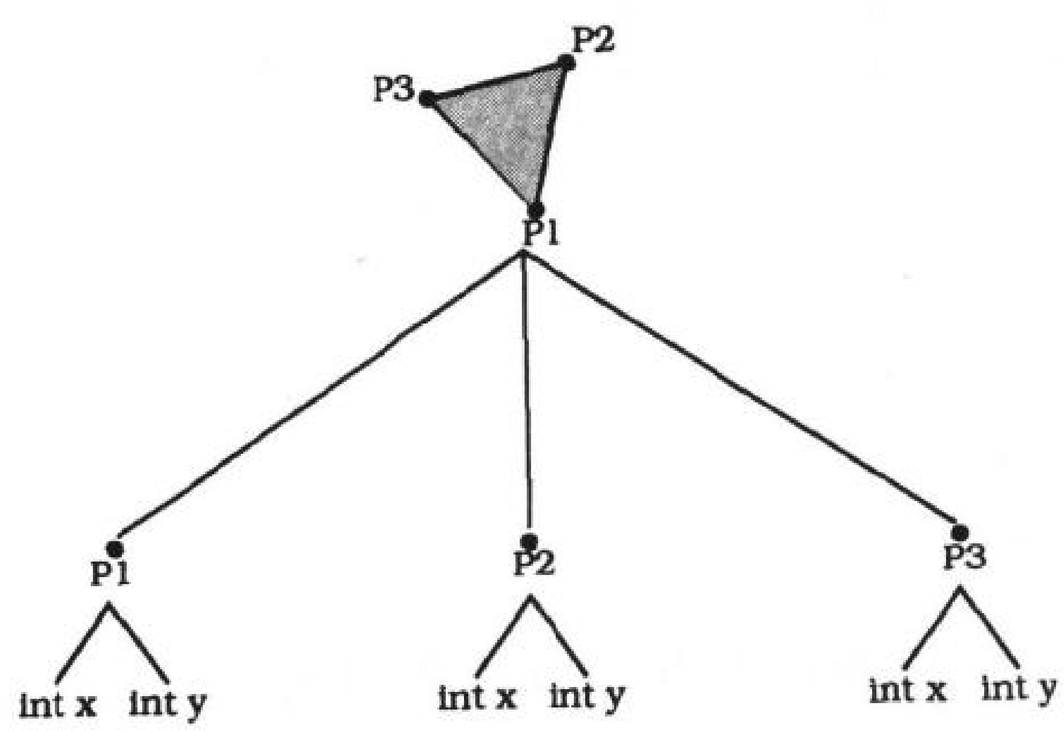
基类GraphicsObject定义诸如绘制、旋转和打印等公共方法。所有其他类要么由GraphicsObject导出，要么由它的子类之一导出。提供诸如Triangle这样的特殊类型图形对象的类的实现，使用组装-部分变体。




```
abstract class GraphicsObject {
    abstract public void dump();
    abstract public void
                rotate(int xc, int yc, double angle);
    // much more ...
}
```

类Triangle是GraphicsObject子类的一个例子。每个三角形恰好由三个不在一条直线上的笛卡儿坐标点组成。因此，一个三角形对象担任一个包含三个点作为部分对象的整体的角色。所以，类Triangle的实现基于点Point的实现。例如，旋转一个三角形可以通过旋转它的角来完成。所以旋转方法是整体服务的例子，使用由部分提供的操作。三角形与其角之间的组装-部分关系用下图说明：

235

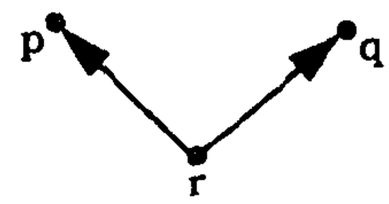


调用方法rotate用于一点时，旋转的中心作为参数传递。如果该中心和那一点重合，该方法不做任何事情，否则它将围绕中心以给定的角度旋转该点：

```
class Point extends GraphicsObject {
    int x;
    int y;

    public static boolean isCollinear
        (Point p, Point q, Point r) {
        // using long arithmetic to avoid overflow:
        return ( (long)(p.x - r.x) * (q.y - r.y) -
                 (long)(p.y - r.y) * (q.x - r.x)) == 0;
    }
    public Point(int xCoord, int yCoord) {
        x = xCoord;
        y = yCoord;
    }
    public void dump() {
        System.out.print("POINT ");
        System.out.println("(" + x + "/" + y + ")");
    }
    public boolean isEqual(Point aPoint) {
        return ( (x == aPoint.x) &&
                 (y == aPoint.y));
    }

    public void rotate(int xc, int yc, double angle) {
        if (isEqual(new Point(xc,yc)))
            return;
        else {
            double cosA = Math.cos(angle);
            double sinA = Math.sin(angle);
            double dx = x - xc;
            double dy = y - yc;
            x = (int) Math.round( cosA * dx -
                                  sinA * dy +
                                  xc        );

            y = (int) Math.round( sinA * dx +
                                  cosA * dy +
                                  yc        );
        }
    }
}
```

236

Triangle的构造函数必须检查作为参数传递的三个点是否在同一直线上，这是三角形作为整体的限制性检查的一个例子。三个点p、q、r不在同一直线上的充要条件是它们定义了一个二维的向量空间：



如果是这样，每个点z必须满足下面的条件：

$$\exists(\lambda,\mu \in \Re);\begin{bmatrix} p_x - r_x & q_x - r_x \\ p_y - r_y & q_y - r_y \end{bmatrix} \times \begin{bmatrix} \lambda \\ \mu \end{bmatrix} = z$$

对每个点z方程有解当且仅当矩阵行列式为非零：

$$((p_x - r_x)\cdot(q_y - r_y)) - ((q_x - r_x)\cdot(p_y - r_y)) \neq 0$$

如果不是这样，三点在同一直线上，并且构造函数出现一个异常。

```
class PointsAreCollinear extends Exception {}
class Triangle extends GraphicsObject {
    Point p1;
    Point p2;
    Point p3;
    public Triangle(Point po1, Point po2, Point po3)
        throws PointsAreCollinear
    {   // check if these points are collinear.
        // If yes, raise an exception
        if (Point.isCollinear(po1, po2, po3))
            throw new PointsAreCollinear();
        p1 = po1; p2 = po2; p3 = po3;
    }
    public void dump() {
        System.out.println("TRIANGLE");
        System.out.print("Point 1: ");
        p1.dump();
        System.out.print("Point 2: ");
        p2.dump();
        System.out.print("Point 3: ");
        p3.dump();
    }
    public void rotate(int xc, int yc, double angle) {
        p1.rotate(xc, yc, angle);
        p2.rotate(xc, yc, angle);
        p3.rotate(xc, yc, angle);
    }
}
```

237

我们使用收集-成员变体实现不同的图形对象组。之所以能使用该变体，是因为一个组不需要它的成员的具体子类型——相反它能把它的每个成员当作类GraphicsObject的实例来处理。类GroupObject包含诸如增加图形对象和通过所有组成员迭代等功能。注意，类GraphicsObject并不严格遵守组合变体[GHJV95]。造成这种现象的原因是部分对象与整体对象的类型不同。整体是类GroupObject的一个实例，部分则不是——我们为了这个目的已引入类GroupObject。可选方案很可能是扩充GraphicsObject使之具有增加元素的功能，不论导出类是否实现了组对象。

如果诸如rotate这样的方法被这样一个组调用，则组将在它的所有成员上递归地调用该方法。

```
class GroupObject extends GraphicsObject {
    private Vector members = new Vector();
    public int size() { // number of members
```

238

```
        return members.size();
    }
    public GraphicsObject objectAt(int pos) {
        return (GraphicsObject)(members.elementAt(pos));
    }
    public void addObject(GraphicsObject aShape) {
        members.addElement(aShape);
    }
    public void rotate(int xc, int yc, double angle) {
        for (int i = 0; i < members.size(); i++) {
            objectAt(i).rotate(xc, yc, angle);
        }
    }
    public void dump() {
        System.out.println("GROUP with " + size() +
                           " members: ");
        for (int i = 0; i < members.size(); i++) {
            objectAt(i).dump();
        }
    }
}
```

想像一下，一个用户建立了不同的图形对象，用鼠标选择它们，并将它们插入到一个组，并告知对象编辑器围绕（0，0）以π/4的角度旋转该组。编辑器随后将执行类似于下面的一段代码：

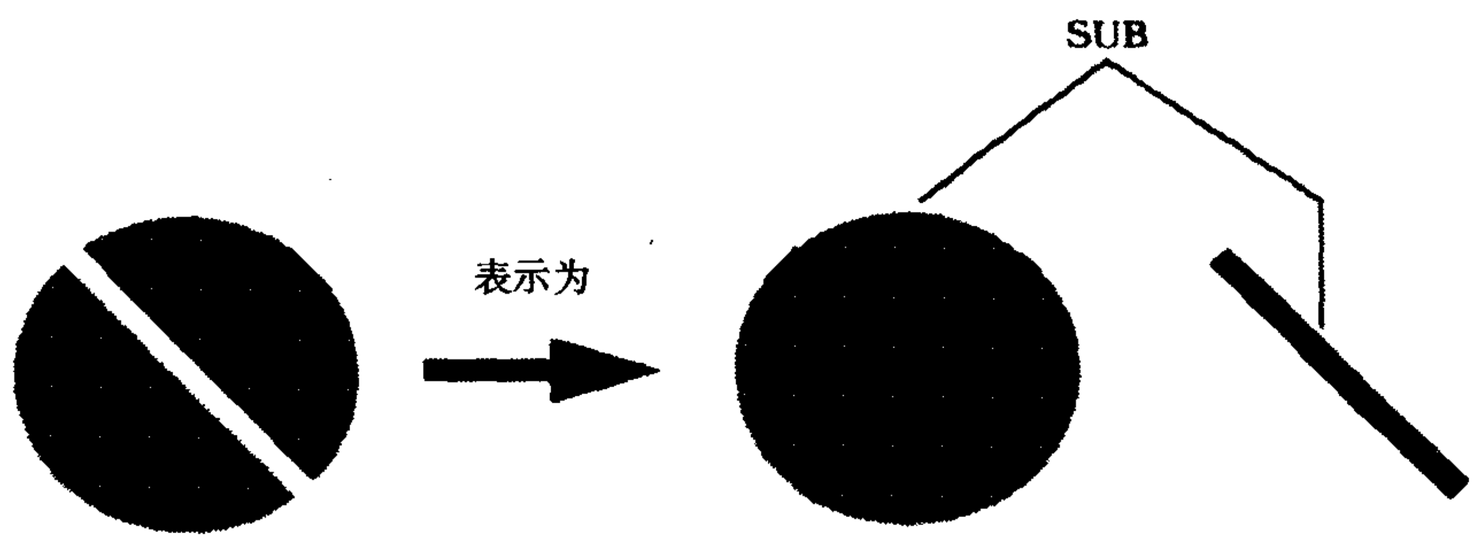
```
Point p1 = new Point(10,10);
Point p2 = new Point(10,20);
Point p3 = new Point(20,10);
Triangle t = new Triangle(p1,p2,p3);
Circle   c = new Circle(new Point(0,0), 10);
Rectangle r = new Rectangle(new Point(-5,-5),
                            new Point(+5,+5));
Line l = new Line(new Point(1,1), new Point(10,5));
GroupObject g = new GroupObject();
g.addObject(t);
g.addObject(c);
g.addObject(r);
g.addObject(l);
g.rotate(0,0,java.lang.Math.PI/4);
```

前面我们已经引入的类支持诸如圆或三角形等简单形状，同样支持这种图形对象分组。为了创建更为复杂的形状，类TreeObject的实例支持使用算子的图形对象的组合。例如，带有一个矩形孔的圆可以用二叉树来描述，左子树定义圆，右子树定义矩形。节点由算子SUB和附加数据组成。算子SUB定义两个图形相减。在本例中，几何图形矩形从圆中减去，形成了一个带洞的圆。 239

树形对象实现部分-整体的容器-内容变体。整体由简单形状通过几何公式计算得出的复杂形状给出。公式中的图形对象和算子表示部分。当一个操作（例如移动）被TreeObject实例调用时，整体对组成它的所有子形状转发该请求。

10. 已知应用

许多面向对象应用的关键抽象都服从整体-部分模式。例如，一些图形编辑器支持不同类型



的数据组合形成多媒体文档。通常根据组合设计模式[GHJV95]来实现这些编辑器。在CAD或动画系统中，构造中的项由组装-部分结构来表示。能够被分层次地构建并能够表示语义单元的一个应用的几乎所有方面可以成为整体-部分模式在它的某个变体中应用的根据。

大多数**面向对象的类库**提供诸如列表、集合和映射等收集类。这些类实现收集-成员和容器-内容变体。参见[SNI94]和[Lea96]中的例子。

诸如Fresco或ET++[Gam91]等图形用户界面工具箱使用整体-部分模式的组合变体。

11. 效果

240 整体-部分模式具有如下**优点**：

部分的可变更性。整体封装部分并因此使其客户机不能见到这些部分。这就使修改整体的内部结构而不影响客户机成为可能。部分的实现也将发生彻底改变而不必修改其他部分或客户机。

事务的分离。一个整体-部分结构支持对事务的分离。每个事务由一个分离的部分来实现。因此，通过由较简单的服务组合它们相比将它们当作单块集成单元来实现，会较容易地实现复杂策略。

可重用性。整体-部分模式支持两个方面的重用。首先，整体的部分能够重用于其他聚合对象。其次，在整体中对部分的封装阻止客户机在它的源程序中“分散”部分对象的使用——这就支持整体的重用。

整体-部分模式有如下**不足**：

间接的行为降低了效率。因为整体在其部分周围建立了一层包装，它也在一个客户机请求和实践它的部分之间引入了一个间接的附加层。与单一结构相比，这就会导致额外的运行期间的开销，尤其当部分自身又作为整体-部分结构来实现时。

分解为部分较为复杂。由不同的部分合成合适的一个整体通常是很困难的，尤其是采用自底向上方法的时候。这是因为理想的拆分依赖于很多问题，例如给定的应用领域、待模拟的结构和要求由整体提供的功能等。

参见

根据文献[GHJV95]，在下述情况中可应用组合设计模式：

想要表示对象的整体-部分层次。

想让客户机能够忽略组合对象和单个对象之间的差别。客户机将统一看待组合结构中的所有对象。

241 组合模式是整体-部分模式的一个变体，当你面临这两种需求时，你应该考虑到这一点。

外观（*Facade*）设计模式[GHJV95]有助于为复杂的子系统提供简单的接口。客户机使用该接口而不必直接访问子系统的不同部分。然而，外观结构并不强行封装部分——客户机也可以直接访问部分。与整体-部分结构的另一个区别是虚包并不从简单的部分服务中组成复杂的服务——它们仅仅进行必要的接口转化并向合适的部分转发客户机请求。

致谢

感谢我的同事Peter Graubmann，他对该模式描述提出了富有成果的建议和评价。 242

3.3 工作的组织

通常通过几个组件合作来实现复杂的服务。在这种结构内如何优化组织工作，需要考虑几个方面的因素。例如，每个组件都应该有一个明确定义的责任以及一个基本策略，这个策略用来提供不应该涉及许多不同组件的服务。

在组织复杂服务的实现时，可以应用几个通用的原则。事务分离，方针和实现的分离，以及“分而治之”方法都是这种原则的例子（参见第6章“模式及软件体系结构”）。针对组织特殊种类服务的模式就是基于这种能够实现的技术。

在本节，我们描述了一种在系统内组织工作的模式：

- 主控-从属（*Master-Slave*）模式支持容错性、并行计算以及计算准确性。主控组件将工作分配给相同的从属组件并从从属组件返回的结果中计算出最终的结果。

主控-从属应用“分而治之”原则。工作被分割成几个独立处理的子任务。整体服务的结果通过使用每个局部处理操作提供的结果计算得出。主控-从属模式被广泛地应用于并行及分布式计算领域。

另一个主控-从属模式的应用实例就是所谓的“三重模块冗余”（triple modular redundancy）原理。在这种方法中服务的执行被委派给三个独立的组件，其中至少有两个必须提供相同的结果才会被认为有效。 243

职责链、命令及中介者模式[GHJV95]同样也属于这个范畴：

- 职责链（*Chain of Responsibility*）模式通过给予多于一个对象机遇来处理请求，避免了将请求的发送者与其接收者耦合在一起。链接接收对象，然后沿链路将请求进行传递直到有一个对象能够处理它为止。
- 命令（*Command*）模式将请求作为对象封装起来，允许以不同请求对客户机进行参数化、对请求排队或记录请求以及支持可撤销的操作。
- 中介者（*Mediator*）模式定义了一个对象，它封装了一组对象交互的方式。中介者通过明确地阻止对象的相互引用促进了松散耦合，而且允许独立地改变它们的交互。

像主控-从属模式、职责链模式和中介者这样的模式为工作的组织提供了一种通用的协作技术以及结构框架，这种模式类似于针对子系统及组件的结构分解的模式（参见3.2节“结构分解”）。

采用这些模式解决某个特殊问题（例如，为了实现矩阵乘法而使用主控-从属模式）还要受

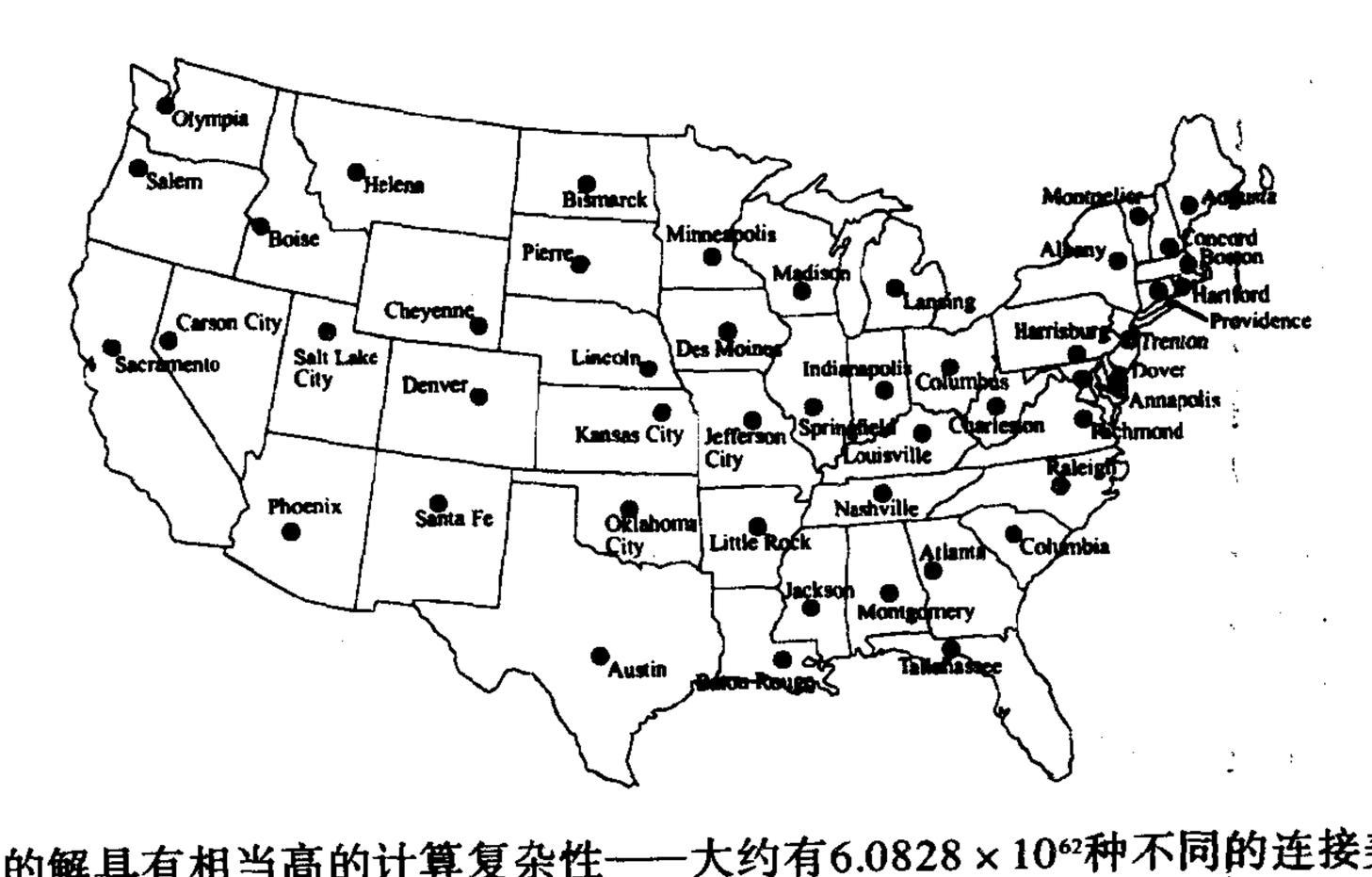
244 到开发过程中应用程序的具体设计活动的影响。

3.3.1 主控-从属

主控-从属设计模式支持容错性、并行计算以及计算准确性。主控组件将工作分配给相同的从属组件并从从属组件返回的结果中计算出最终的结果。

1. 例子

图论中，旅行商（traveling-salesman）问题是众所周知的。在一组给定的位置间找到一条最佳的往返路径，比如每个位置恰好访问一次的最短路径。



这个问题的解具有相当高的计算复杂性——大约有6.0828×10^{62}种不同的连接美国各州首府的旅行方式！通常，n个位置的旅行商问题的解答最多有$(n-1)!$个可能的路线。因为旅行商问题是NP-完全的[GJ79]，所以要想找到最优的解就无法回避这样的高度复杂性。

因此，大多数现有的实现旅行商问题的近似最优方案是比较一组固定的路径。最简单的方法是随机选取路径进行比较，希望能够发现一条足够接近最优路径的最佳路径。然而我们要确信被考察的路径是以随机的、独立的方式选取的，则选择的路径数就十分庞大。 245

2. 语境

将工作分成在语义上相同的子任务。

3. 问题

“分而治之”是用来解决多种问题的通用原则。工作被分割成几个同等的独立处理的子任务。整个计算的结果通过使用每个局部进程提供的结果计算而来。在实现这个结构时会出现如下几方面的问题：

- 客户机不应该意识到计算是基于“分而治之”原则的。
- 客户机和子任务的处理都不应依赖于划分工作及汇聚最终结果的算法。
- 使用不同的但语义上相同的实现来处理子任务是非常有益的，例如可以提高计算的准

确性。

• 处理子任务有时需要合作，例如在使用有限元方法模拟应用程序当中。

4. 解决方案

在单个子任务的处理和服务的客户机之间引入合作实例。

主控（*master*）组件将工作划分为几个同等的子任务，将这些子任务委派给几个独立的但语义上相同的从属（*slave*）组件，并从这些从属组件返回的结果中计算出最后的结果。

这个通用原则在以下三个应用领域可以找到：

• 容错。一个服务的执行可以被委派给几个复制的实现。可以检测并处理服务执行的失效。

• 并行计算。一个复杂的任务可以被划分成一组固定数目的并行执行的相同子任务。最终的结果来自处理这些子任务所得到的结果。 246

• 计算准确性。服务的执行被委派给几个不同的实现。可以检测并处理不精确的结果。

为所有的从属组件提供一个公共接口。让所有服务的客户机仅仅同主控组件进行通信。

➡我们决定通过比较一组固定数目的路径来逼近旅行商问题的解。选择路径的策略很简单——随机地选取。我们采用面向对象的并行程序设计语言pSather的一个早期版本来实现这个思路简单的程序[MFL93]。程序在Thinking Machines公司的带有64个处理器的CM-5计算机上运行。

利用CM-5多处理器体系结构的特点，可以并行计算不同路径的长度。因此我们把实现路径长度的计算视为从属的工作。每个从属组件都把待比较的一组路径作为输入，随机地选取路径并返回发现的最短路径。主控组件事先确定需要被实例化的从属组件数目，规定每个从属实例需要比较的路径数目以及开始的从属实例，并从所有返回的路径中选择一条最短路径。也就是说，从属组件提供局部的最佳路径，而主控组件决定出总体的最佳路径。 ❑

5. 结构

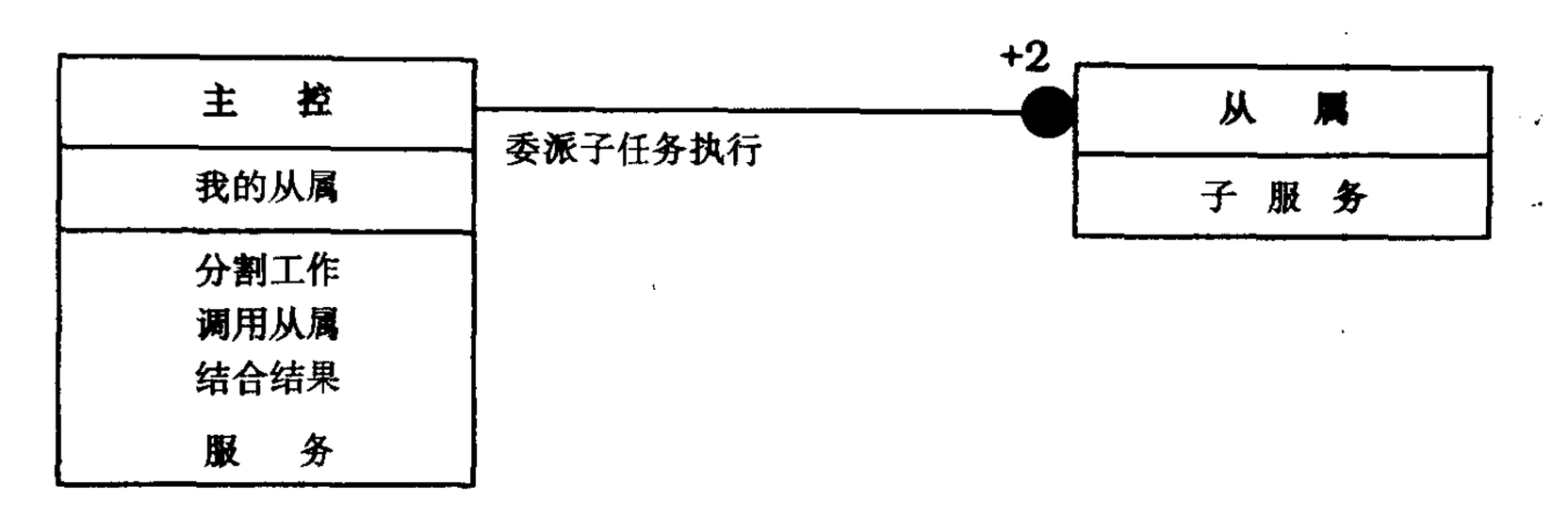
主控组件提供一种能够利用“分而治之”原则求解的服务。它提供了允许客户机访问这个服务的接口。在内部，主控组件执行将工作划分成几个相同子任务的功能，启动并控制它们的处理，并从所有得到的结果中计算出最后的结果。主控组件还要保存对受委托去处理子任务的所有从属实例的引用。

从属组件提供一个子服务，它可用来处理由主控组件定义的子任务。在主控-从属结构中，至少有两个从属组件实例连接到主控组件上。 247

类 主控	协作者 从属
责任 • 把工作分割并分给几个从属组件 • 开始从属组件的执行 • 用从属组件返回的子结果计算出总的结果	

类 从属	协作者 –
责任 • 实现由主控组件使用的子服务	

可以通过如下的OMT图来阐述由主控-从属模式定义的结构。

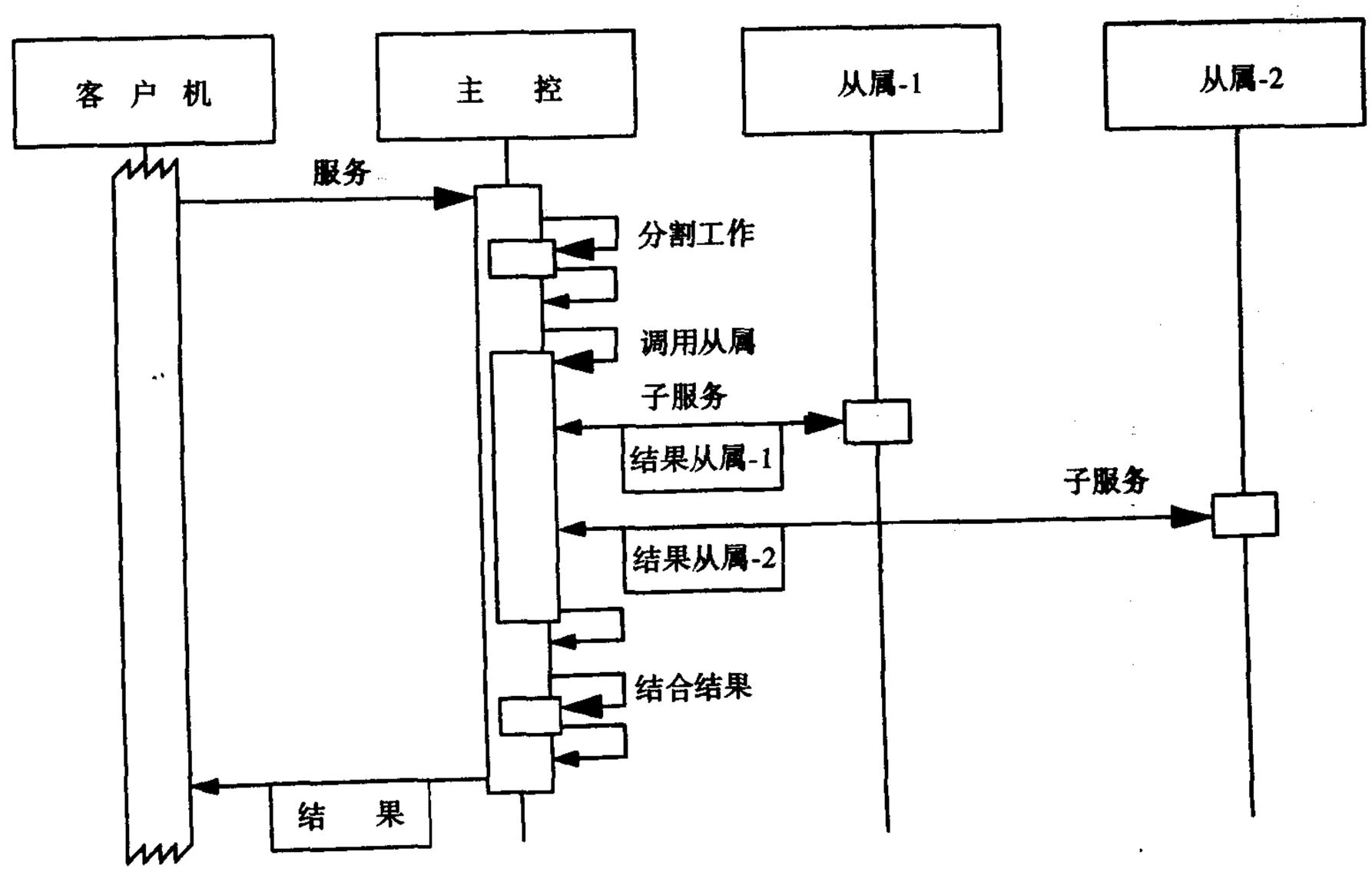


6. 动态特性

在下列我们所假定的场景中，为简单起见，一个接一个地调用从属组件。然而，当从属组件并发地被调用时，主控-从属模式释放所有的能量，例如将它们指派给几个独立的控制线程。这个场景包括六个阶段：

- 一个客户机请求来自主控组件的服务。
- 主控组件将工作分割成几个相同的子任务。
- 主控组件将这些子任务委托给几个从属实例，开始它们的执行并等候它们返回结果。
- 从属组件执行子任务的处理并将它们计算的结果返回给主控组件。
- 主控组件从从属组件返回的局部结果中计算出整个任务的最终结果。
- 主控组件将这个结果返回给客户机。

248



7. 实现

主控-从属模式遵循如下五个步骤。注意，这些步骤是从特殊问题中抽象出来的，在支持容

错、并行计算及计算准确性这些特殊情况模式的应用时，或者将从属组件分给几个进程或线程时，就要考虑这些特殊的问题。这些方面将会在“变体”小节进行讨论。

(1) 划分工作。规定任务的计算如何被分割成一组相同的子任务。确定处理子任务所必需的子服务。

➡对并行的旅行商程序而言，我们可以分割问题以便为从属组件提供一个往返路径并计算出它的代价。然而，对于像CM5这样的带有SPARC节点处理器的机器，这样的分割或许太过细小。监控这些并行执行以及将多个参数传递给它们的代价不但不会提高速度，反而会降低算法的整体性能。

更为有效的解决方案是定义一个子任务，用它来确定所有路径的特殊子集中的最短路径。这个解决方案考虑到在CM5上只有64个可用处理器的情况。可用处理器的数量限制了并行处理的子任务数。为了找出由每个子任务比较的路径数，我们由可用处理器的数量来划分待比较的所有路径数。 ❑ 249

(2) 结合子任务结果。说明整体服务的最终结果是如何借助于处理单独子任务所得结果而计算得到的。

➡每个子任务仅返回待比较的所有路径的一个子集中的最短路径。 ❑

(3) 说明主控与从属间的合作。为步骤1确定的子服务定义一个接口。它由从属组件实现并且由主控组件使用，用来委托单独子任务的处理。

从主控到从属传送子任务的一种选择是当调用子服务时作为一个参数包含它们。另一种选择是定义一个仓库，主控在此处放置子任务而从属可以由此处获得这些子任务。当处理一个子任务时，单独的从属可以在分离的数据结构上工作，或者所有的从属可以共享一个数据结构。从属明确地返回它们的处理结果并作为一个返回参数，或从属把结果写到一个主控可以检索到的独立仓库之中。

要判断这些选择中哪种选择最好，需要依赖许多因素。例如，传递子任务到从属的代价、复制数据结构的代价以及几个从属在共享数据结构上操作的代价。初始问题也影响待作出的决定。当从属修改它们操作所基于的数据时，需要为每一个从属提供它自己初始数据结构的副本。如果它们不修改数据，则所有从属都可以工作在一个共享的数据结构之上，例如，在实现矩阵乘法时。

➡对于旅行商程序，让每个从属操作在表示所有城市及其连接的图形的副本上。在初始化从属时，将创建这些副本。有一种可供选择的办法——从属从一个共享的图形表示中读取——不选择这种方法的原因在于CM5内部网络上的这种通信负载严重降低了应用程序的性能。 250

从属对主控的接口由一个函数来定义，该函数取待评估的随机路径数作为输入参数。函数返回找到的最佳路径，这可以由一个类TOUR的实例来表示。

```
random_perms(numberPerms : INT) : TOUR
```

random_perms()中的术语perms代表排列，因为我们把往返路径表示为n个节点的排列，而这些节点表示待访问的n个城市。 ❑

(4) 实现从属组件，依据是前面步骤中开发的规格说明。

➡类TSP是小应用程序的设计中心。它包含一个构造函数，作用是创建一个随机旅行并更

新迄今为止找到的最短旅行，以及前面步骤说明的random_perms()函数。类COMPLETE_GRAPH表示操作其上的TSP实例的图形结构。类RANDOM表示一个随机数发生器。代码部分是不完全的，摘自一个已经投入运行的应用程序。

```
class TSP is
    -- Data structures
    best_tour, current_tour     : TOUR;
    graph                       : COMPLETE_GRAPH;
    random                      : RANDOM;
    -- Constructor for the slave that initializes
    -- the return value, creates the graph structure,
    -- and creates the random number generator.
    create() : TSP is
        res            := new;
        res.graph      := COMPLETE_GRAPH::create;
        res.random     := RANDOM::create;
    end; -- create
    -- Construct a number of randomly selected tours and
    -- return the tour with the lowest costs
    random_perms(numberPerms : INT) : TOUR is
        i : INT := 1;
        while i <= numberPerms loop
            construct_random_tour;
            update_optimum;
            i := i+1;
        end; -- loop
        res := best_tour;
    end; -- random_perms
    -- Construct a new random tour and calculate its costs
    construct_random_tour is -- not shown here
    end; -- construct_random_tour
    -- Update the optimal tour if the currently evaluated
    -- tour is better than the current optimum
    update_optimum is
        if current_tour.cost < best_tour.cost then
            best_tour     := current_tour;
        end; -- if
    end; -- update_optimum
end; -- class TSP
```

251

注意：在update_optimum中的赋值假设深度备份的语义，或者在赋值后 current_tour将指向一个新的TOUR对象。否则，在修改current_tour时，construct_random_tour()要损坏best_tour。初始的程序通过交换best_tour和 current_tour所引用的两个TOUR对象来解决这个问题。 ❑

(5) 实现主控，依据是步骤1~3中开发的规格说明。

有两种可选的方法能把任务划分成子任务。第一种选择是把工作分成数量固定的子任务。这种方法最适用于当主控委托从属执行一个完整任务时。这种方法通常用在主控-从属模式被用来支持容错或者计算准确性应用时，不然的话大量的并行工作总是被固定下来并被称为先验知识（priori）。第二种选择是尽可能地定义必要多的子任务。例如，在旅行商程序中，尽量多地定义那些处理器能够承担的子任务。

通过应用策略模式[CHJV95]，可以支持用于细分任务的算法交换。我们进一步讨论应当在“变体”小节考虑的问题。

启动从属组件，控制它们的执行，并搜集它们的结果的代码与许多因素相关。从属组件能按顺序执行吗？它们能在不同的进程或线程上并发运行吗？它们是彼此独立还是协同工作？关于这一点，我们将在“变体”小节中详细说明。

主控的最终结果要依靠从从属组件收集来的结果。这个算法可以遵循不同的策略，正如“变体”小节中所描述的那样。为了支持其动态交换和变化，可以再次应用策略模式[CHJV95]。 252

必须处理可能的错误，正如从属组件执行失效或启动线程的失效。这些内容将在“变体”小节中详细讨论。

在主控-从属结构中，只有一个主控组件。可以应用单件模式[CHJV95]来确保这个属性。

➡在旅行商程序中，我们用一个类CM5_TSP的对象表示主控。它给其客户机提供一个函数best_tour()，客户机返回由整个主控-从属结构访问的最佳往返路径。best_tour()函数取待生成的路径数和处理器数作为参数使用。

函数distribute()给所有的处理器复制图形和一些附加的数据结构。从而按我们说明的实现次序依次工作。“@j”意味着在处理器j上执行这个操作。函数distribute()创建与可获得处理器数相同多的新从属。函数random_perms()启动从属。函数update_optimum()从从属返回的局部最优路径中选一个最佳路径。

协调从属的策略是异步地开始它们，稍后使它们同步，尤其当我们要选择一个最佳路径时。为了实现这一行为，我们使用“未来”原则。未来是一个变量，它在不同的控制进程或线程中定义了异步计算的一个值。当稍后存取该变量时，同步是可以做到的。因为pSather支持未来，我们就为从属使用一个未来数组来协调它们的并行执行。由于篇幅的原因，在此不举例说明对象的创建。在例子中用到的有关pSather版本的更详细内容，参见[Lim93]。

```
class CM5_TSP is
     -- Data structures. Shared variables in pSather
     -- correspond to static members in C++
     shared n : INT                -- Number of Cities
     shared P : INT;               -- Number of processors
     shared T : ARRAY{TSP};        -- The slave array
     shared best_tour : TOUR       -- The best round trip
-- Assign a slave to each available processor
distribute is
     -- Create the slave instances
     i : INT := 1;
     while i <= P loop
          -- initializes T[j];
          copy_graph()@j;
          i := i+1;
     end; -- loop
end; -- distribute
-- Launch the slaves
random_perms(t : INT) is
     i, j, jobs_per_proc : INT;
     -- Calculate how many tours each slave must visit
     -- Assume that P divides t
```

253

```
jobs_per_proc     := t/P;
-- Define a monitor
m := MONITOR{TOUR}:=MONITOR{TOUR}::new;
-- Launch each slave at its processor
i := 1;
while i <= P loop
    m :- T[i].random_perms(jobs_per_proc)@i;
    i := i + 1;
end; -- loop
-- wait until the slaves finish with their
-- computation and take the results of the slaves
-- in whatever order they are returned
j := 1;
while j <= P loop
        current_tour := m.take;
        update_optimum();
        j := j + 1;
    end; -- loop
end; -- random_perms
-- Select the optimal tours from the trips the slaves
-- returned
update_optimum is -- not shown here
end; -- update_optimum
-- Return the optimal tour from t randomly created
-- ones with help of P slaves.
best_tour(t, p : INT): TOUR is
    P := p;
    -- Create the slaves, launch them, determine
    -- the best trip visited, and return this tour
    -- to the client calling the master
    distribute;
    random_perms(t);
    update_optimum;
    res := best_tour;
  end; -- best_tour
end; -- class CM5_TSP
```

254 ❑

8. 变体

主控-从属模式有三个应用领域：

用于容错的主控-从属。在这个变体中，主控只是委派一个服务的执行到一个固定数量的复制的实现，每一个实现都代表一个从属。一旦第一个从属工作终止，产生的结果就又返回到主控的客户机。只要至少一个从属能够不失败，就支持容错，客户机可得到一个有效结果。主控能处理所有从属失败的情况，例如，由客户机操作引起的异常或返回一个特别的“异常值”[Cun94]。主控用超时来检测从属的失效。然而，该变体不能帮助主控自身失败的情况——它是关键的组件，必须“处于激活状态”以使这种结构正常工作。

用于并行计算的主控-从属。支持并行计算是主控-从属模式最普遍的用途。在这个变体中，主控把复杂任务分成一些相同的子任务，其中每一个子任务分别由一个独立的从属并行执行。主控从从属返回的结果中计算出最终结果。主控包含划分整个任务并计算最终结果的策略。

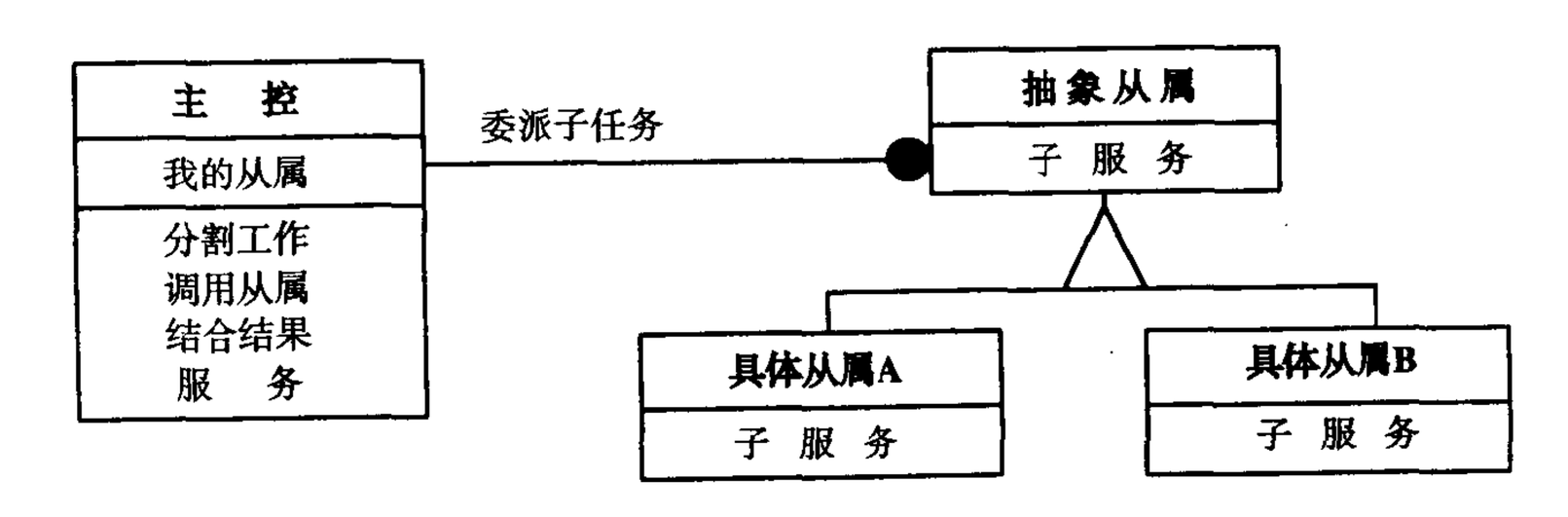
子划分任务和协同从属的算法紧紧依赖支持程序运行的硬件体系结构。例如，在一个带有通用处理器的分布式存储机器上，粒度通常要比在SIMD（单指令多数据流Single Instruction

Multiple Data）机器上大得多。影响算法的另一方面是机器的拓扑结构和其处理器相互连接的速度。主控和从属之间的合作也依赖机器是否存在共享和分布式存储等方面的情况。工作的划分更进一步受那些列在从属中作为线程变体（见下面）的问题所影响，主控与从属之间的合作受在“实现”小节的第3步所列问题的影响。

在主控计算出最终结果之前，它必须等待所有从属完成其子任务的执行。为使主控摆脱一个一个单独地同步于每个从属的任务，文献[KSS96]引入界限（*barrier*）的概念。用终止主控等待的那个从属初始化界限，接着它挂起主控执行直到所有主控控制的从属均终止。相反，pSather 例子在增强方式中工作，一旦从属终止运行，`random_perms()`方法就能得到它的结果。 255

*用于计算准确性的*主控-从属。在该变体中，服务的执行至少委派给三种不同的实现。每一种实现都有一个独立的从属。主控等待所有从属工作的完成，并对其结果进行表决来检测和处理非准确性。这种表决可以遵循不同的策略。例如，主控可从最多数量从属返回的结果中选择，所有结果取平均，或者使用一个使所有的从属产生不同结果的异常值[Cun94]。

为了提供不同的从属实现，我们用一个附加的抽象类来扩充主控-从属模式的结构。这要对所有从属实现定义一个公用接口。于是，不同的从属实现都是来源于这个抽象基类。



为了实现从属存在更多的变体：

作为进程的从属。为了处理位于分离进程中的从属，可以用两个额外的组件来扩充初始的主控-从属结构[Bro96]。主控包含一个跟踪所有为主控工作的从属的顶端组件（*top component*）。为了保持主控和顶端组件独立于分布式从属的物理地址，远程代理代表主控进程中的每个从属。可以应用转发器-接收器或客户机-分配器-服务器模式来实现进程间通信。 256

作为线程的从属。在这个变体中，每个从属在其自身的控制线程内实现[KSS96]。在这个变体中，主控创建线程，启动从属，并在继续其自身的计算之前等待所有的线程完成。主动对象模式[Sch95]有助于实现这样一个结构。

在这个变体中，主控必须处理两个问题：当一个线程无法创建时会发生什么？应该创建多少个线程？第一个问题的解决方法是直接调用从属的服务而不是在独立的线程中启动它们。性能将受到影响，但结果是正确的。最佳的线程数量依赖于可用的处理器数和每个线程所要求的工作量。太多线程在它们的创建和销毁时产生开销，在内存消耗中也是这样。文献[KSS96]建议采取不同的策略进行试验，以“比处理器数目多一些的线程”开始。

*与从属协作的*主控-从属。一个从属的计算可能依赖于其他从属的计算状态，例如，用有限

个元素执行模拟时。在这种情况下，所有从属的计算都必须有规则地挂起，因为每个从属必须使自己与它所依赖的从属协作，这之后该从属重新开始它们各自的计算。

有两种方法实现这样一种行为。首先，可以在从属的内部包含用于从属协作的控制逻辑。这可将主控从实现这些协作的任务中解脱出来，但可能降低总体结构的性能。从属将独立地停止它们的执行并保持空闲直到它们所依赖的从属准备好协作。

第二种选择是让主控在从属之间保持依赖和控制从属的协作。在规则的时间间隔内，主控挂起所有从属，恢复它们计算的当前状态，向依赖于该数据的所有从属转发该数据并继续所有从属的执行。

257

9. 已知应用

文献[KSS96]列出了用于并行计算的主控-从属模式的三个具体例子。

- 矩阵乘法。积矩阵中的每一行可由一个单独的从属计算得出。
- 变换-编码一个图像。例如计算一个图像中每个8×8像素块的离散余弦变换（Discrete Cosine Transform, DCT）。由一个独立的从属计算出每一块。
- 计算两个信号的互相关性。通过对于信号中的所有样本的迭代，计算样本及其相关物之间的均方距离，并求距离之和而得出。可以将样本上的迭代划分成几部分，对于每个划分单独计算距离平方及其和。对这些划分的和再求和，得到最终的总和。由一个独立的从属完成每个局部求和。主控组件定义划分，启动从属，并计算最终的总和。

基于Linda原理[Gel85]，文献[KR96]中描述的**工作池模型**（workpool model）应用主控-从属模式实现对并行计算的进程控制。程序员可以向一个工作池指派一组“工人”。每个工人提供相同的服务，并在一个独立的进程或线程中实现。客户机向工作池发送请求，由工作池在相关联的工人的协助下处理这些请求。请求本身是一个像矩阵乘法这样的函数，可以在工人的帮助下将它的执行并行化。该函数对应于主控-从属模式中的主控组件。

群(Gaggles)[BI93]的概念建立在主控-从属模式的原则之上，用来处理面向对象软件系统中的“复数性”（pluralitys）。一个群代表一组重复服务对象。从客户机接收服务请求后，群将该请求转发给它包含的一个服务对象。每个服务对象可以是原子的，这意味着它执行该服务并回送一个结果；服务对象也可以是另一个群，它本身又代表一组重复服务对象。

258

文献[Bro96]中列出了主控-从属设计模式的几个应用，这些应用全都是考虑分布式从属的。应用实例包括分布式设计规则检查系统Calibre™ DRC-MP和CheckMate IC验证工具，两者都来自于良师图形公司（Mentor Graphics）。

也可以使用“分而治之”的方法将一个大数分解成素数因子（*prime factors*）。由于这个问题是密码学的核心问题，而政府对它很感兴趣，并且它需要大量的计算资源，因此可以通过因特网来解决这个问题。一个站点进行任务划分并将子任务分给愿意提供计算时间和使用他们的机器的人。

10. 效果

主控-从属设计模式有以下几个**优点**：

可互换性和可扩充性。通过提供一个抽象的从属类，可以在不对主控进行重要改变的情况

下交换已有的从属实现或增加新的从属。客户机不受这种变化的影响。如果采用策略模式[GHJV95]来实现，那么当改变用来向从属分配子任务和计算最后结果的算法时，情况也是一样。

事务分离。主控的引入将从属和客户机代码与下述事务代码分离，这些事务包括划分工作，委派工作给从属，从从属那里收集结果，计算最终的结果，以及处理从属失效或者不准确的从属结果等。

效率。如果精心实现的话，用于并行计算的主控-从属模式可以加速计算一个特殊服务的性能。不过，必须考虑并行计算的代价（见下面的不足）。

主控-从属模式有以下3个**不足**：

可行性。主控-从属体系结构并不总是可行的。必须要划分工作，拷贝数据，启动从属，控制其执行，等待从属的结果并计算最终结果。所有这些活动都要消耗处理时间和存储空间。

对机器的依赖性。用于并行计算的主控-从属模式在很大程度上依赖于运行程序的机器的体系结构——详情参见“变体”一节。这可能降低主控-从属结构的可变性和可移植性。

难以实现。实现主控-从属结构是不容易的，尤其对于并行计算而言。必须考虑许多不同的方面并且要仔细地实现，例如怎样进行工作的子划分，主控与从属怎样协作，怎样计算最终的结果。还必须处理像从属执行失效，主控与从属间通信失效，以及启动并行从属失效这样的错误。要实现用于并行计算的主控-从属模式，通常需要有关正在开发的系统的目标机体系结构的全面知识。 259

可移植性。因为存在对于基础的硬件体系结构的潜在依赖，所以主控-从属结构迁移到其他一个机器上是困难的，或者甚至可以说是不可能的。这个不足特别体现在用于并行计算的主控-从属模式中，类似的情况还出现在运行在CM5计算机上的简单旅行商程序中。

参见

这种模式的较早版本出现在[PLoP94]。

用于并行计算服务的主控-从属模式[Bro96]为实现一个主控-从属结构提供了附加的洞察力。主控-从属结构不同于在这里描述的结构，因为它集中描述了作为进程从属的变体。

《线程程序设计》（Programming with Threads）[KSS96]一书详细描述了作为线程从属的变体。

对象组[Maf96]是一种用于组通信的模式，它支持分布式应用程序中的容错。它对应于用于容错的主控-从属变体并对其实现提供了附加的细节内容。对象组模式为分布在网络化计算机上的一组复制对象提供了一个本地代理。将一个请求广播到组的所有对象。只要一个组成员成功终止，这个请求就会取得成功。

致谢

感谢Ken Auer、Norbert Portner、Douglas C. Schmidt、Jiri Soukup和John Vlissides，他们对于主控-从属模式的[PLoP94]版本的改进提出了有价值的批评和建议。特别感谢Phil Brooks 和Jürgen Knopp对这个新版本的贡献。 260

3.4 访问控制

有时，一个组件甚至一个完整子系统不能或者不应该被它的客户机直接访问。例如，不是

所有的客户机都有权使用组件的服务，或检索一个组件提供的特殊信息。

在本节中，我们描述一个帮助保护对一个特殊组件进行访问的设计模式。

- 代理（*Proxy*）设计模式使一个组件的客户机与一个组件代表而不是组件本身通信。引入这样一个占位符有许多用途，包括提高效率、易于存取和防止越权访问等。

文献[GHJV95]也描述了代理模式。我们的描述与此不同，不同点在于它将模式所基于的通用原则与其具体应用情况相分离，而我们将它描述为一个变体。我们也给出了没有被“四人帮”（Gang-of-Four）版本涵盖的代理模式的几个新变体。

代理模式应用广泛。几乎每个分布式系统或分布式系统的基础结构都局部地使用该模式表示远程组件，例如OMG-Corba[OMG92]。代理的一个较新的应用是万维网[LA94]，在那里它被用来实现代理服务器。

在[GHJV95]中描述的另外两个模式——外观和迭代器也属于这一范畴。

- 外观（*Façade*）模式为子系统中的一组接口提供了一个统一接口。外观定义了使子系统更容易使用的高级接口。
- 迭代器（*Iterator*）模式提供了连续访问一个聚集对象的成员，而不暴露其基础表示的方法。

261 像代理模式一样，外观和迭代模式有广泛的应用。

外观保护了一个子系统的组件，免遭其客户机直接访问。反过来也一样，客户机也不依赖子系统的内部结构。一个外观组件给实现服务的子系统组件递送新来的服务请求。因此，外观比代理有更大的粒度，它防护了对单个组件的访问。

在一个面向对象的程序或类库中，几乎每一个容器类都提供迭代器。一个迭代器定义了客户机能够遍历或访问一个容器元素的次序。例如，访问一个二叉树的所有元素，可以为先序、

262 中序和后序遍历来定义迭代器。

3.4.1 代理

代理（*Proxy*）设计模式使一个组件的客户机与一个组件代表而不是组件本身通信。引入这样一个占位符有许多用途，包括提高效率、易于存取和防止越权访问等。

1. 例子

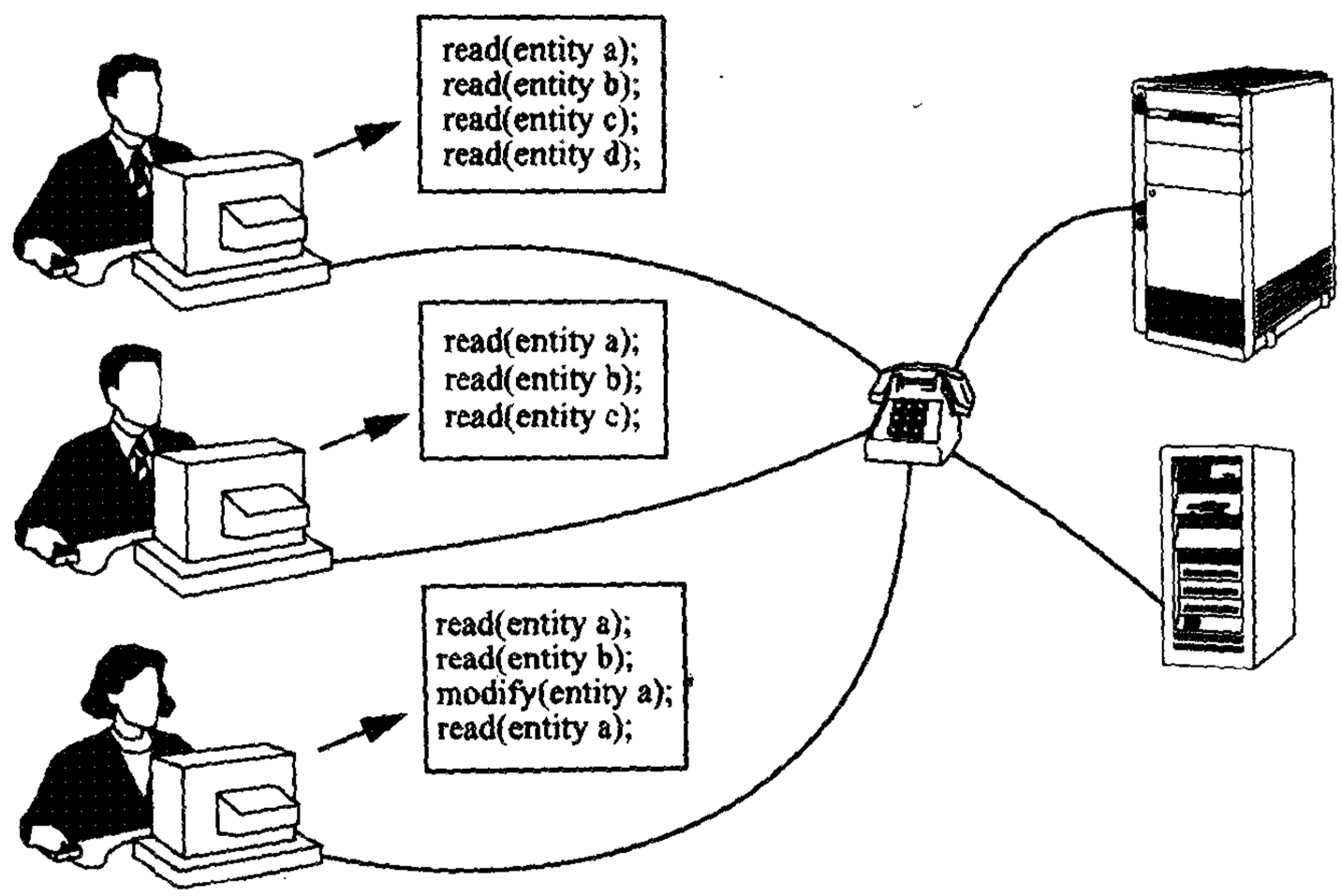
公司工程人员通常参考数据库以便得到关于资料提供者、可用部分及蓝图等信息。每次远程访问可能代价昂贵，因为许多访问都是相似的或者相同的而且经常是重复的。这种情况清楚地提供了最优访问时间和范围。但是我们不想用这种最优化来加重工程师编写应用程序代码的

263 负担。使用的优化和类型的存在性对用户和程序员来说应该是非常透明的。

2. 语境

一个客户机需要访问另一个组件[⊖]的服务。直接访问在技术上是可行的，但不是最好的途径。

⊖ 在此有意非常含糊地使用“组件”这个词。由于上述原因，“组件”可以指你不想直接访问的任何事物。这种组件的例子包括：普通的局部对象、一个外部数据库、Web上的一个HTML页或一个嵌入在文本文档中的图像。



3. 问题

直接访问一个组件通常是不合适的。我们不想将其物理地址强制编码到客户机，对组件进行直接和无限制的访问可能是低效的或者甚至是不安全的。需要额外的控制机制。这种设计问题的一个解决方法是权衡下面的全部或部分强制条件：

- 访问组件应该是运行期间高效、花费合适并且对客户机和组件两者都安全的。
- 对客户机而言，访问组件应该是透明的、简单的。特别是客户机不必改变它过去呼叫任意其他可直接访问组件的呼叫行为和语法。
- 客户机应该清楚地意识到对远程客户机访问的可能的性能或经济方面的损失。完全透明性会使各种服务之间的花费差别变得不明显。

4. 解决方案

让客户机与组件代表而不是组件本身通信。这种代表——称为代理——提供组件接口并执行附加的前期处理和后期处理，例如访问控制检查或制作原件的只读副本等，见如下小节。

5. 结构

原件（*original*）实现一种特别的服务。这种服务包括从诸如返回或显示数据的简单操作到复杂的数据检索功能或包含深一层组件的计算。

客户机（*client*）对特定的任务负责。为完成客户机的工作，它以一种访问代理的间接方式调用原件的功能。客户机不必改变它过去呼叫本地组件的呼叫行为和语法。 264

因此，*代理*（*proxy*）提供了和原件相同的接口，而且保证了对原件的正确访问。为完成这种功能，代理保持对它所表示的原件的引用。通常代理和原件之间是一对一的关系，但是对这个通用模式的两个变体——远程和防火墙代理，这种规则也有例外。要获得更多信息，参见“变体”小节。

抽象原件（*abstract original*）提供了通过代理和原件实现的接口。在像C++这样的语言中，在子类型定义和继承之间没有显著的差别。代理和原件二者都来自于抽象原件。当访问原件时，客户机针对这个接口进行编码。

类 客户机	协作者 • 代理
责任 • 利用代理提供的接口来请求特殊服务 • 完成它自己的任务	

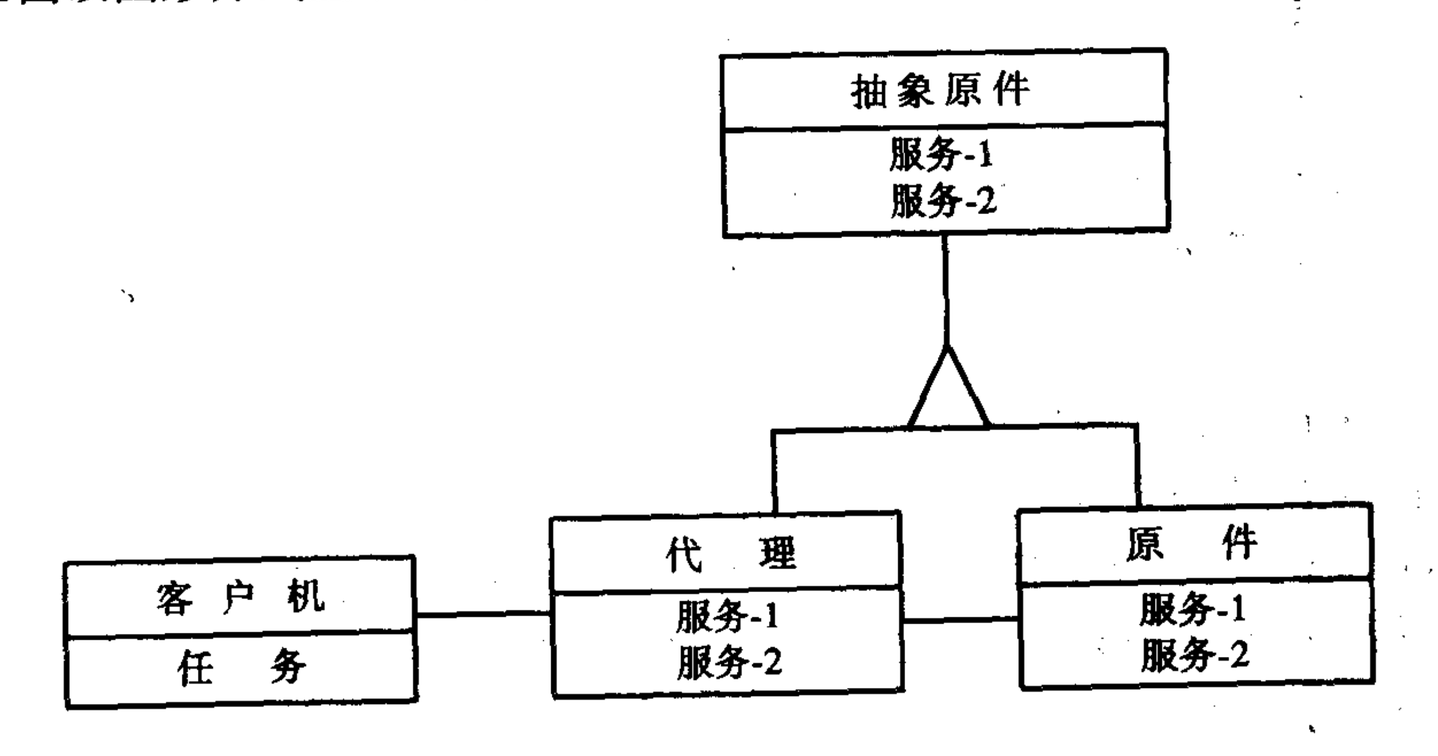
类 抽象原件	协作者 –
责任 • 对代理和原件，服务作为一个抽象基类	

类 代理	协作者 • 原件
责任 • 对客户机提供原件接口 • 确保安全、有效和正确地访问原件	

类 原件	协作者 –
责任 • 实现一个特殊服务	

265

下面的OMT图以图形方式显示了各类之间的关系。

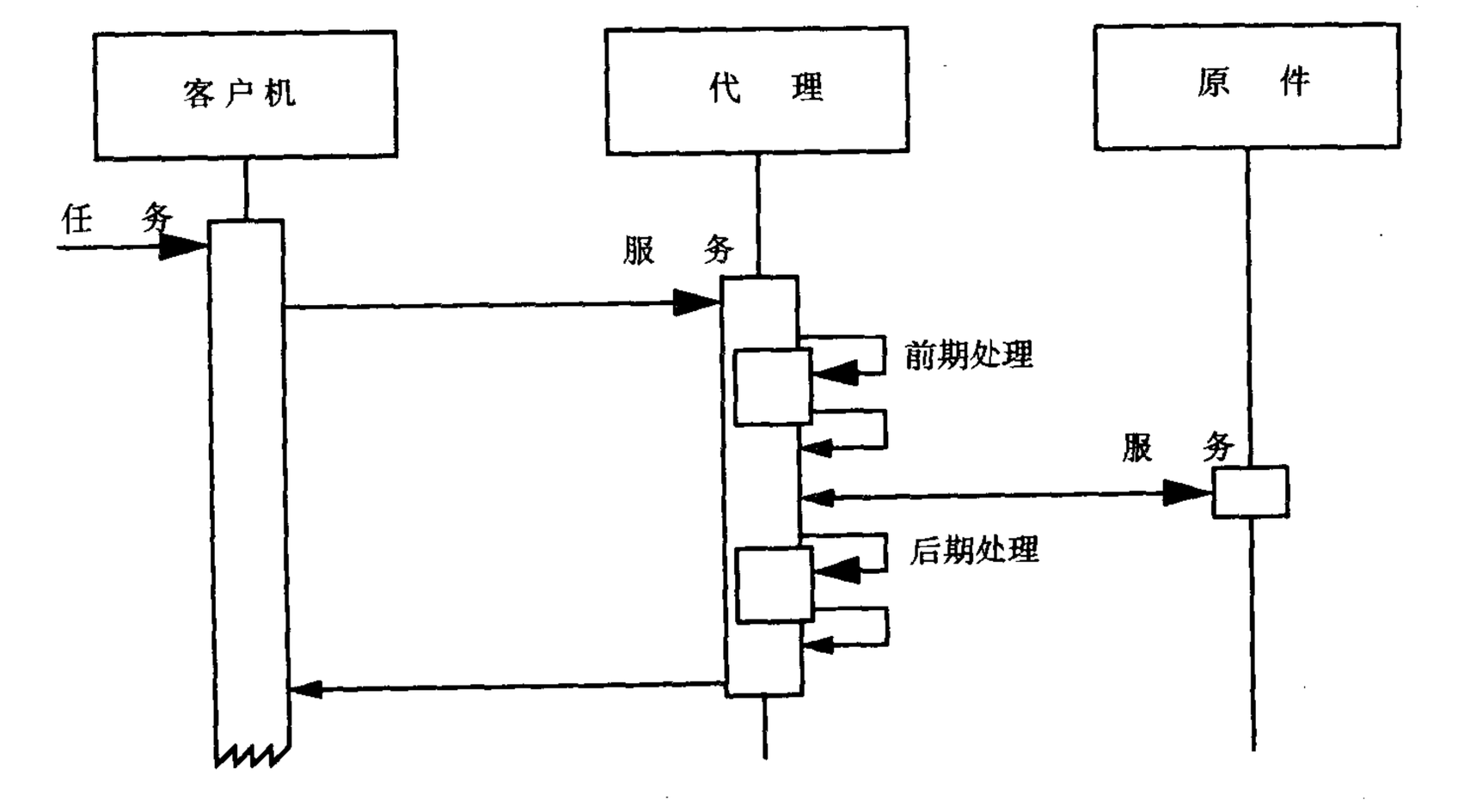


6. 动态特性

下图显示了代理结构的一种典型的动态场景。注意这些操作在代理内部执行，而不是依赖它的实际特征——更多信息可参见“变体”小节。

- 当客户机执行它的任务时，它要求代理实施一种服务。
- 代理接受新来的服务请求并对其进行前期处理。这种前期处理包括诸如查找原件地址这样的操作，或检查本地存储器，以便查看被请求的信息是否已经可以获得。
- 如果代理必须参考原件来实现这种请求，那么它会使用合适的通信协议和安全措施将该请求转发到原件。
- 原件接受这种请求并实现它。原件将响应发回给代理。
- 代理接收响应。将它传送到客户机之前或之后，代理可实施附加的后期处理操作，例如存储结果、调用原件的析构函数或释放一个资源锁。

266



7. 实现

为了实现代理模式，需要执行如下步骤：

(1) 为处理对一个组件的访问控制确定所有职责。将这些职责依附在一个单独的组件——代理上。这一步的详细描述将在“变体”小节中给出。

(2) 如果可能，引入一个抽象基类来详细说明代理和原件两种接口的共同部分。从这种抽象基类导出代理和原件。如果代理和原件之间的相同接口是不可行的，那么可以使用一个适配器[GHJV95]来保证接口的适应性。使代理接口适应原件接口会让客户机以为有相同的接口，用于适配器和原件的一个公共基类或许也是可能的。

(3) 实现代理的功能。为此检查第1步中指定的角色。

(4) 解除原件及其客户机所承担的现已迁移到代理中的职责。

(5) 通过给代理一个指向原件的句柄将代理和原件关联起来。这种句柄可以是一个指针、一个引用、一个地址、一个标识符、一个套接字和一个端口等。

(6) 删除原件及其客户机之间所有的直接关联。用对代理的相似关系取代它们。

267

8. 变体

下面，我们描述一般代理模式的七种变体。首先总结出单个变体最适合的那种情况：

- 远程代理。远程组件的客户机应该与网络地址和进程间通信协议相屏蔽。
- 保护代理。必须防止越权访问组件。
- 缓存代理。多重本地客户机可以共享远程组件传回的结果。
- 同步代理。必须同步对一个组件的多重同时访问。
- 计数代理。必须防止组件的偶然删除，或者收集使用情况的统计数字。
- 虚拟代理。处理或装入一个组件的代价是昂贵的，但该组件的部分信息可能就已足够了。
- 防火墙代理。本地客户机应该与外部世界相隔离。

下面的段落将详细描述每个变体的特征和实现细节。

远程代理（*Remote Proxy*）封装和保持了原件的物理位置。它同样实现了与原件间执行实际通信的IPC（进程间通信，Inter-Process Communication）的程序。对于每个原件，依据原件服务所需的每地址空间来实例化代理。对于复杂的IPC机制，可以将同原件通信的职责转移到转发器组件来细化代理，正如转发器-接收器模式中所描述的。类似地，将一个接收器组件引入到原件中。

考虑到效率，我们将远程代理分为三种情况：

- 客户机和原件在同一个进程中。
- 客户机和原件在同一台机器的不同进程中。
- 客户机和原件运行在不同机器上的不同进程中。

268

第一种情况是简单的：我们不需要一个与原件通话的代理。对于第二种和第三种情况，我们会将远程地址的字段放入代理中，地址通常包括机器ID、端口或者进程号以及对象ID。第二种情况显然不需要机器ID。如果想存储一个机器ID所占用的很少几个字节，记住第二种和第三种之间的分化会使代理的代码变得复杂。开发这种分化逻辑的努力通常得不到证实，除非进程间通信的手段在两种情况下不同，此时强调这种分化逻辑。尽管那样，我们仍然可以在顶端添加一个薄层来隐藏三种情况的分化，从而简化使用寻址方案的代码。抽象原件的存在使运行在三种情况下的客户机对此完全透明。

在高性能的应用程序中，我们在提交一个场外请求前经常想要确定在应用层中进行通信是否昂贵。在这种情况下，一个远程代理解释了这个信息。

保护代理（*Protection Proxy*）保护原件免受越权访问。为完成此功能，代理检查每个客户机的访问权限。可以通过使用平台所提供的访问控制机制很容易地达到此目的。如果合适和可能，试着给出每个客户机到其他组件的许可集。访问控制列表是这个概念的常用实现手段。

为了实现缓存代理（*Cache Proxy*），用临时保存结果的一个数据区来扩充代理。开发一种策略来保持和更新缓存。当缓存满了并且需要释放空间给新的输入时，可以采用几种策略。例如，可以删除使用最少的缓存输入，或实现一个“移到前端”的策略——这是最容易实现而且足够有效的。在这个策略中，当一个客户机访问一个缓存输入时，它被移到一个双链表的前端。当新的输入必须被加入缓存中时，输入可以从链表的尾部进行删除。

同样必须注意“缓存无效”的问题——当原件中数据发生改变时，在别处存储的这种数据的副本将变得无效。如果你的应用程序时效性很强，总是有最新的数据，这种情况就会相当严重，你可以宣布一旦其输入的任何原件副本发生变化，整个缓存无效。相反地，你可以利用“直写”

269

（write-through）策略，众所周知该策略来自微处理器缓存设计，用于较小粒度的控制。一旦原件被修改，所有它的副本也都同样被修改。注意，假如有多于一个的副本，或者副本在远处时，情况就变得复杂了，相对于微处理器缓存，此处情况要简单一些。如果你的客户机可以接收少量过期的信息，你就可以用其截止日期来标注每个单独的缓存输入。这种策略的例子包括万维网浏览器。

同步代理（*Synchronization Proxy*）控制了多个客户机同时访问。每次只有一个或一定数目的客户机能够访问原件，如果这一点是非常重要的，那么代理可以通过信号灯实现排斥操作[Dij65]。要不然，它可以利用操作系统提供的任何的同步方法。你还可以区分读和写访问。在前面的情况，可以采取更加自由的策略，例如，没有写操作被激活或在等待时，允许任意数量的读操作。操作系统文献是一种研究这种机制的好的资料源。

计数代理（*Counting Proxy*）可以被用来收集使用情况统计，或者为自动删除过时的对象而去实现一种众所周知的技术——引用计数技术。为此，这种计数代理保留存在于原件中的引用数，并且当这个数变成零时删除这个原件。需要确保对应于每一个原件恰好有一个计数代理，对原件的每次访问是通过一个定义好的各自代理的界面进行的。同样应该牢记，单独的引用计数无助于寻找意料之外的引用彼此的代理的其他孤立组件循环问题。

计数指针惯用法说明了在C++中实现计数代理的一个不同方法。在那里，引用计数器在其原件或者其自身的对象内部，而不是在句柄或代理中。惯用法也讨论了为什么有些C++实现使用涉及引用计数器的另一个间接层次，并在一个句柄对象被创建或删除时更新引用计数器。

虚拟代理（*Virtual Proxy*），也称为惰性构造（*lazy construction*），假设一个应用程序引用了次要的存储器，例如硬盘。这个代理不会透露原件是否被完全装入或者关于它的提纲式信息是否有效。在要求时执行原件丢失部分的装入。 270

当一个服务请求到达并且代理中存在的信息不足以处理请求时，从磁盘中装入所需数据并将请求转发到新创建或扩充的原件。如果原件已经完全装入，则仅仅转发请求。这种转发做起来应是透明的，以至于客户机总是使用相同的接口，而与原件是否在主存中无关。当不再需要原件或其中的一部分时，客户机或其关联的模块的职责是通知代理。代理随后释放分配的空间。当几个客户机引用同一个原件时，增加同步代理和缓存代理变体的能力也许是合适的。

防火墙代理（*Firewall Proxy*）包含与具有潜在敌意环境进行通信所必需的连网技术和必需的保护代码。通常，在防火墙机器上，防火墙代理作为一个守护进程（daemon process）被执行，这个机器也被称为“代理服务器”。所有将请求传递到外部世界的客户机引用这个代理。这个代理工作在幕后，通过检查输出的请求和引入的应答来确保内部安全和访问规则。当一个请求不遵从这些规则时，或它的资源被耗尽时，它就拒绝访问。提供给客户机一种错觉，即对外部访问没有任何阻碍，并且不需费力就可以登录到防火墙机器上。同样地，安全得到了维护，例如保护用户账户不受外部的攻击。因特网上的服务器给人一种错觉，即代理就是客户机。这使得在防火墙背后的网络内部结构被隐藏起来。

防火墙代理的一个显著特征是用户需要客户机软件的“代理”版本。例如，标准的ftp软件必须被与代理关联的另外版本所替代而不是直接访问目标机。结果是当这些服务的等价代理版本可获得时才可以使用新的服务。 271

因为所有的通信流都通过防火墙代理，所以它构成一个潜在的瓶颈并为最优化（例如缓存）提供了一个理想场所。它同样为附加任务（例如登录和核算）提供了一个理想位置。有关防火墙设计的更多信息，参见文献[CZ95]。

9. 已解决的例子

可能经常需要使用多于一个的上述代理变体——也许需要某代理扮演几个上述角色并完成相应的职责。首先，作出选择，方法是先挑选出期望的角色（例如虚拟和缓存），然后考虑把这些角色结合成一个代理。

如果结合它们所得的代理膨胀太多，就把它分离成为较小的对象。一个例子是分解复杂的连网代码成为一个转发器-接收器结构——参见转发器-接收器模式。在这种情况下，该代理仅仅保留了原件的位置信息以及本地还是远程的决定。

可以通过使用具有远程和缓存代理变体特点的代理来解决远程数据访问问题。实现这样的混合模式代理可以通过使用整体-部分模式来完成。

结合代理的一部分是缓存。它包含了一个存储区和用于更新和查询缓存的策略。利用“最少使用”策略并调整缓存大小，可以削减外部访问的开销。如何解决缓存无效的问题取决于是否已经控制数据库。如果已经控制，当对应的原件数据库输入被修改时，可以安排单独的缓存输入无效。如果没有，对结合代理的缓存的每次访问都必须检查输入创立是否仍然有效。

结合代理的另一部分维护原件的名称和地址并执行实际的IPC。如果原件是关系数据库，它把客户机请求转化为SQL查询并把结果转化成所需的格式。如果它是另一种类型的组件，就使用转发器-接收器模式。

272

10. 已知应用

代理模式通常与转发器-接收器模式结合使用以实现“存根”（stub）概念[LPW94]。

NeXTSTEP。代理模式被用在NeXTSTEP操作系统中以提供远程对象的本地存根。当一个特定服务第一次访问远程对象时代理就被创建了。在NeXTSTEP操作系统中，一个代理对象的职责是对收到的请求及其参数进行编码，并将它们转发到它们相应的远程原件。

OMG-CORBA[OMG92] 使用代理模式有两个目的。第一个目的称为“客户机-存根”或IDL-存根，保护客户机避免其服务器和对象请求代理的具体实现。第二个目的称为“IDL-骨架”，由对象请求代理自身使用以便将请求转发到具体的远程服务器组件。

Orbix[Iona95],一个具体的OMG-CORBA实现，使用远程代理。一个客户机通过指定其惟一标识符可以与原件相捆绑。在C++语言的例子中支持`bind()`调用返回一个C++指针，使用标准的C++函数调用语法，客户机可利用这个指针来调用远程对象。

万维网代理[LA94] 描述了通常运行在防火墙机器上的CERN HTTP服务器的情况。这个代理允许防火墙里面的人并发访问外部世界。通过缓存最近传输的文件来提高效率。

OLE。在微软，OLE[Bro94]服务器可以作为动态链接到客户机地址空间的库来实现，也可以作为一个独立的进程来实现。代理被用来对客户机隐瞒一个特殊服务器是本地还是远程的信息。当客户机调用位于其自身地址空间的服务器时，它将直接调用服务器实现。如果服务器不是位于客户机的地址空间，那么一个代理取参数，将它们打包并产生一个远程过程调用到远程

服务器。在服务器中处理另一个代理——用OLE术语称为“存根”——接收请求，解开参数包，将它们压入堆栈并调用适当的服务器方法。如果程序调用返回一个结果，将这个结果打包并传回到客户机代理。这个客户机代理解开结果包并将它返回到客户机，客户机仍不知道服务器是本地的还是远程的。

273

11. 效果

这里所描述的代理模式的一个问题是并非所有的强制条件都可以同等地获得解决。传统的做法是重点考虑容易处理和达到一定的效率，正如在第一个和第二个强制条件中所陈述的那样。当用户或程序员需要保留对微调的明确控制时会发生什么呢，由第三个强制条件来请求吗？一种可能是通过消除抽象超类而在源代码层次上求它的镜像。然后程序员总会察觉到手边的对象是否是“真实的事情”[U2]或者只是一个代理。然而，这又会违反强制条件1和强制条件2。

代理模式具有如下**优点**：

提高效率和降低成本。虚拟代理变体帮助实现一个“按需装入”的策略。这避免了不必要的磁盘导入并且通常能够加速应用程序。一个类似的观点也应用于缓存代理变体中。然而，很明显，代理的附加开销也许会产生相反的效果，这取决于应用程序——参见下面的不足之处。

将服务器组件的位置与客户机相分离。将所有本地信息和定位功能放入一个远程代理变体中，客户机不会因为服务器的迁移或者连网基础结构的变动而受到影响。这将使客户机代码变得更加稳定和可重用。但是注意：一个远程代理的简单实现仍然将原件的位置强制连到其代码中。这样做的优点是它通常可以提供更好的运行性能。如果这种灵活性损失是重要的，则可以考虑往代理中引入一个动态查询图式，正如在客户机-分配器-服务器模式中所描述的那样。

将内务处理代码从功能中分离出来。更概括地说，这个优点适用于所有的代理变体。一个代理减轻了客户机的负担，而这些负担本不属于客户机待执行的任务。

274

代理模式有如下两个**不足**：

间接方法导致低效率。所有的代理都引入一个间接方法的附加层。效率的损失与客户机的清晰结构和通过缓存或者通过利用代理所取得的惰性构造所增加的效率相比，通常是可以忽略的。虽然如此，还是应该彻底地检查这种对于代理模式的每个应用程序的效率都有的影响。

过分采用复杂策略。小心像缓存或者按需装入这样的复杂策略——它们并不总是值得的。这样的例子出现在原件是高度动态时，例如在飞机订票或其他票务预约系统中。这里，由于原件数据的变化，复杂的带有失效机制的缓冲存储可能引入新的开销，这种开销使预定的目的失效。通常，只有粗粒度的实体才能证实所得缓存维护工作是正确的。

参见

装饰（*Decorator*）模式[GHJV95]在结构上与代理模式非常相似。具体组件——代理模式中的原件——执行了一些通过装饰调用的行为，装饰是代理模式中的代理。两个类均由一个公共的基类继承所得。装饰与代理模式之间的主要差别在于其中的目的不同。装饰增加功能，或更一般地说，为动态选择功能除了具体组件的核心功能之外给出选项。代理从非常具体的内务处理代码中释放了原件。

致谢

文献[GHJV95]同样描述了代理设计模式。特别地，它们描述了四种变体：远程、虚拟、保护代理以及“灵巧引用”，后者是计数、虚拟和同步代理等情况的一个结合体。

我们感谢PLoP'95第三工作组的成员为改善这个模式的早期版本提出了有价值的批评和建议。Ken Auer作为这个模式的“领军人”，针对模式再分解成两层模式语言问题给出了关键性的建议，这在[PLoP95]中有专门描述。

275

3.5 管理

系统必须经常处理相似服务类型的对象集，或者相似复杂组件的对象集。一个例子是从用户或其他系统引入的事件，必须适当地解释和调度。当交互系统必须以多种不同方法给出特定应用数据时，表现为另一个例子。必须适当处理这些视图，可以以单独形式也可以以群组形式。

在一个结构良好的软件系统中，独立的“管理者”组件通常用于处理这种同类的对象集。我们描述这种类型的两种设计模式：

- 命令处理器（*Command Processor*）模式将一个服务请求与其执行分开。一个命令处理器组件管理作为独立对象的请求，调度它们的执行，并且提供额外的服务，例如存储供以后撤销用的请求对象。
- 视图处理程序（*View Handler*）模式帮助管理软件系统中的视图。视图处理程序组件允许客户机打开、处理和消除视图，协调视图之间的依赖性，并组织它们的更新。

命令处理器模式和命令模式[GHJV95]均运用了将服务请求封装到命令对象的概念。然而，命令处理器模式将命令模式嵌入到处理命令对象管理的一个结构中。[GHJV95]同样描述了一个管理模式——备忘录：

- 备忘录（*Memento*）模式可以捕获并具体化一个对象的内部状态而无需破坏封装，以便其状态以后能够恢复。

备忘录有助于管理特定组件的状态。例如，当一个先前执行的操作被撤销时，一个组件的状态可能需要恢复。另一个例子出现在当一个客户机需要访问组件状态，但组件的封装又不能被破坏时。备忘录可以为客户机提供当前状态的一个备份。

276

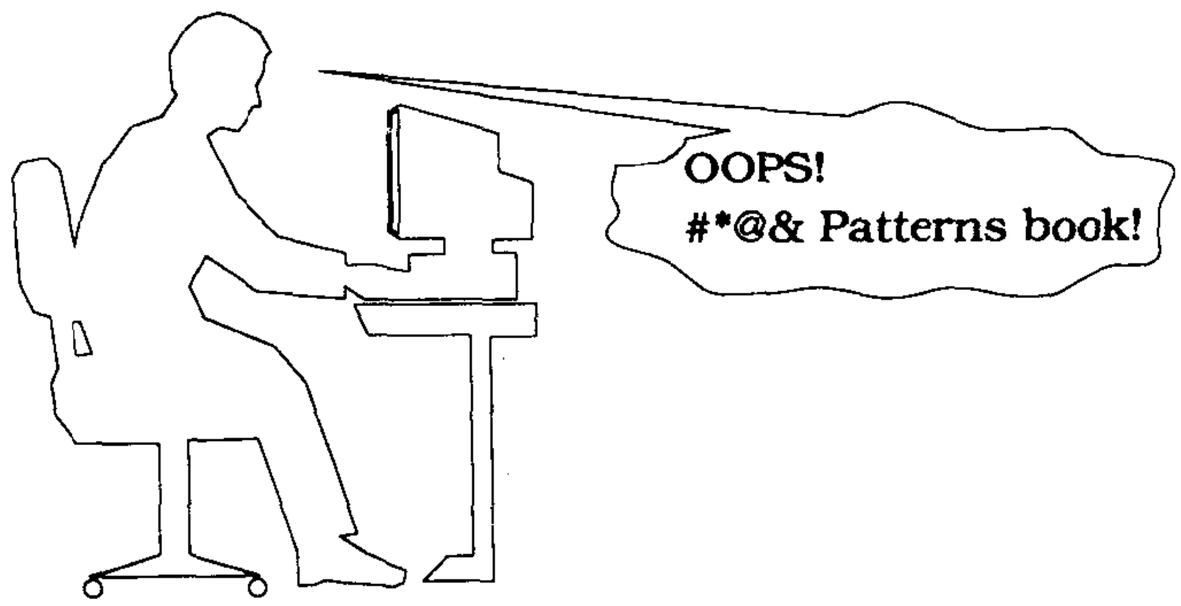
3.5.1 命令处理器

命令处理器（*Command Processor*）设计模式将一个服务的请求与其执行分开。一个命令处理器组件管理作为独立对象的请求，调度它们的执行，并且提供额外的服务，例如存储供以后撤销用的请求对象。

1. 例子

一个文档编辑器通常提供处理用户误操作的方法。一个最简单的例子是撤销多个最近的变更。一个更吸引人的解决方案是能够撤销多重变更。我们希望开发这样的编辑器。为了这里讨

论方便，我们简称它为TEDDI。



TEDDI的设计包括多级的撤销机制和允许未来的增强机制，例如增加新特征或者操作的批处理模式。

TEDDI的用户界面提供了几种交互手段，例如键盘输入和弹出式菜单。程序必须定义一个或多个可以为每次人机交互自动调用的回调（*callback*）过程。

2. 语境

应用程序需要灵活的和可扩充的用户界面，或者是提供与用户函数执行相关联的服务，例如调度或者撤销。

277

3. 问题

包括较多特征的应用程序受益于其结构良好的解决方案，这种解决方案将其界面与其内部功能对应起来。这可以支持不同的用户交互模式，例如对于初学者的弹出菜单，对于有经验用户的快捷键，或者是通过脚本语言进行应用程序的外部控制，等等。

经常需要实现用户请求执行的超出系统核心功能的服务。例如撤销，重做，组请求的宏，活动登录，或者请求调度和暂停。

下列强制条件塑造了解决方案：

- 不同用户希望以不同的方式运行应用程序。
- 应用程序的升级不应该破坏现有代码。
- 诸如撤销的额外服务对所有请求而言应该一致地实现。

4. 解决方案

命令处理器模式建立在[GHJV95]中命令设计模式之上。两种模式都遵循这样的思想：将请求封装在对象之中。一旦用户调用应用程序中的特定功能，请求会立刻变成命令对象。命令处理器模式特别地说明了命令对象是如何被管理的。在“参见”小节中将更进一步讨论命令模式与命令处理器模式之间的差别。

模式描述的中心组件——命令处理器，维护了所有的命令对象。命令处理器安排命令的执行，可以存储它们供以后撤销之用，并且可能提供其他的服务，例如为了测试目的而记录命令顺序。每个命令对象将其任务的执行委托给应用程序功能核心内的供应者组件。

5. 结构

抽象命令（*abstract command*）组件定义了所有命令对象的接口。这个接口至少由执行一条命令的过程组成。由命令处理器实现的额外服务要求进一步适用于所有命令对象的接口过程。例如，TEDDI的抽象命令类定义了一个附加的撤销方法。

278

对于每个用户函数，我们从抽象的命令中导出命令组件（*command component*）。一个命令组件通过使用零个或多个供应者组件（*supplier component*）实现抽象命令的接口。在执行之前，TEDDI的命令保存了相关的供应者组件的状态，万一撤销可以恢复它。例如，删除命令就负责保存被删文本的内容及其在文档中的位置。

类 抽象命令	协作者 –
责任 • 定义执行命令的统一接口 • 为命令处理器的服务（例如撤销和登录）扩展接口	

类 命令	协作者 • 供应者
责任 • 封装一个功能请求 • 实现抽象命令接口 • 使用供应者执行一个请求	

控制器（*controller*）代表了应用程序的接口。它接受请求（例如“粘贴文本”），并创建相应命令对象。命令对象随后被发送至命令处理器中执行。TEDDI的控制器维护了事件循环并将引入的事件映射到命令对象。

命令处理器（*command processor*）管理命令对象，调度它们并且开始它们的执行。这是实现与命令执行相关的附加服务的关键组件。命令处理器保持了特殊命令的独立性，因为它只使用了抽象的命令接口。在TEDDI字处理器情况中，命令处理器也同样存储已执行的命令以备后面的撤销操作使用。

供应者（*supplier*）组件提供了执行具体命令所需的绝大多数功能（即，这些功能与具体命令类相关，而与抽象命令类相反）。相关联的命令通常共享供应者组件。在要求一个撤销机制时，供应者通常提供一种存储和恢复其内部状态的手段。用来实现内部文本表示的组件是TEDDI中的主要供应者。

279

下图给出了模式组件间的主要关系。它说明了撤销是由命令处理器提供的一个附加服务的例子。

6. 动态特性

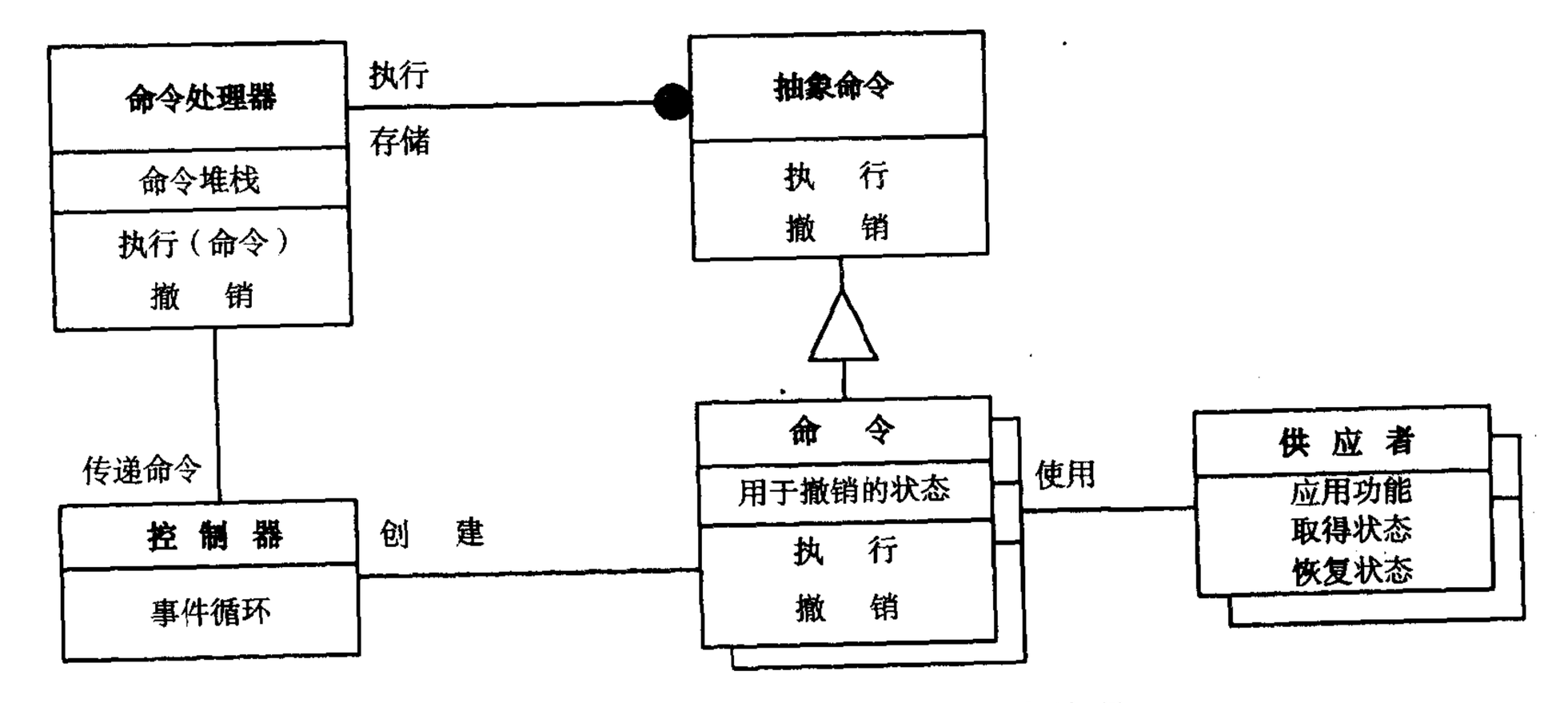
下图给出了一个关于命令处理器模式实现撤销机制的典型场景。所选单词用大写字母开头的一个请求送达，执行，然后撤销。发生如下步骤：

- 控制器在用户的事件循环中接收来自用户的请求并创建一个“用大写字母开头”的命令对象。

类 控制器	协作者 • 命令处理器 • 命令
责任 • 接受服务请求 • 把请求转化成命令 • 把命令传递到命令处理器	

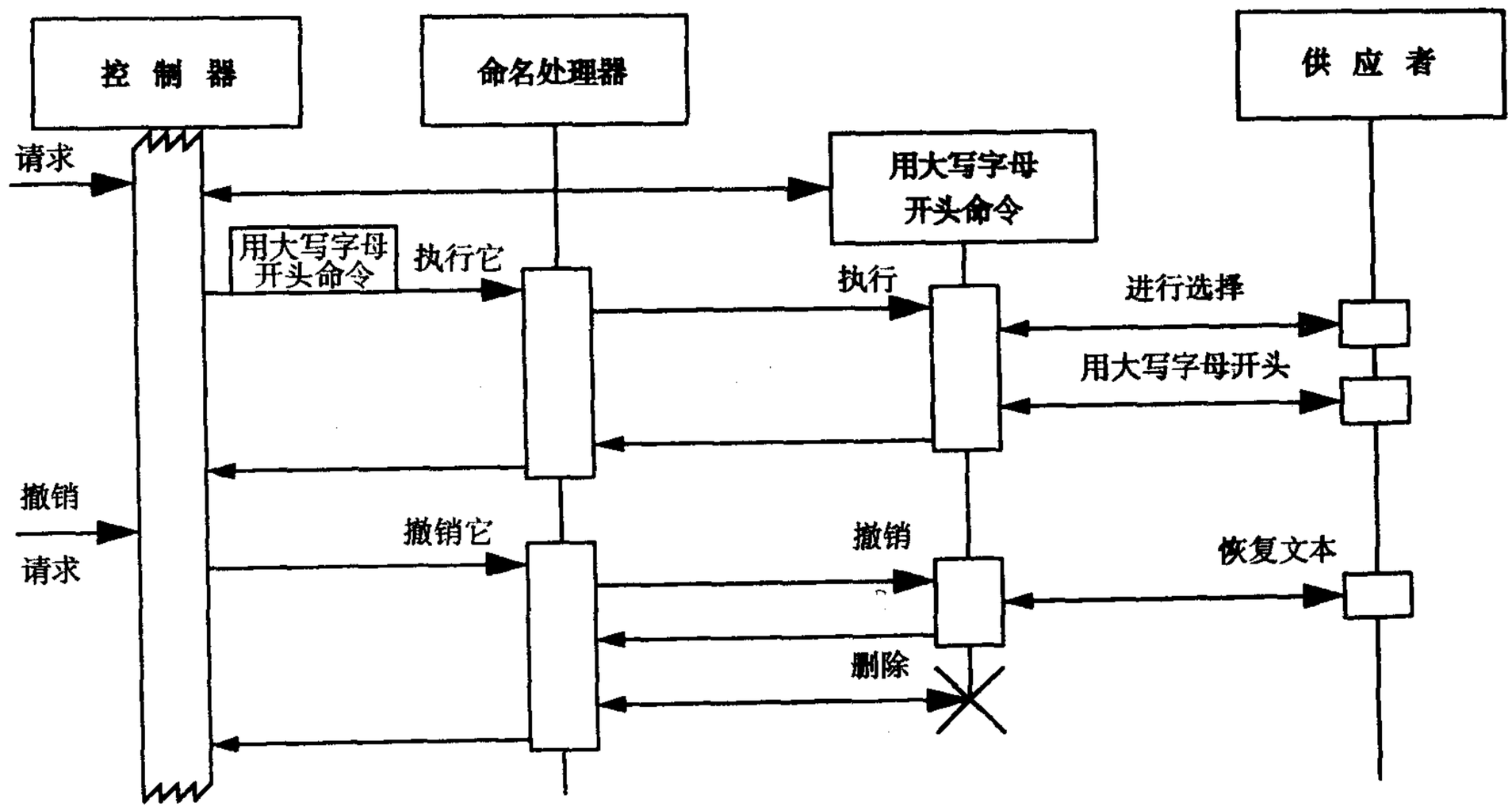
类 命令处理器	协作者 • 抽象命令
责任 • 激活命令执行 • 维持命令对象 • 提供与命令执行相关的额外服务	

类 供应者	协作者 –
责任 • 提供特殊应用的功能	



• 控制器将新的命令对象传递到命令处理器中执行并进一步处理。
• 命令处理器激活命令的执行并存储它供以后撤销之用。
• 用大写字母开头命令从其供应者中检索当前选择的文本，存储这个文本及其在文档中的位置，并要求供应者实际开始大写化选择。
• 在接收一个撤销请求后，控制器传递这个请求到命令处理器。命令处理器调用所有最近命令的撤销过程。

- 大写化命令重新设置供应者到前一步状态，在其原来位置替换存储的文本。
- 如果没有进一步的请求行为或可能的命令，命令处理器将删除命令对象。



280
~
281

7. 实现

为了实现这种模式，执行下列步骤：

(1) 定义抽象命令接口。抽象命令类隐藏了所有特殊命令的细节。这个类常常指定执行命令所需的抽象方法。它同样定义了由命令处理器提供的实现附加服务所需的方法。例如，用于登录命令的“getNameAndParameters”方法。

➡针对TEDDI中的撤销机制，我们区分三种命令类型。用枚举对它们建模，因为命令类型可以动态变化，正如步骤3所示：

没有变化。无需撤销的命令。鼠标移动属于该类。

正常。可以被撤销的命令。在文本中替换一个单词是一个正常命令的例子。

不许撤销。不能被撤销的命令，以防止先前执行的正常命令被撤销。

如果我们希望文本“人称正确”并且用“他/她”替换所有出现的“他”，TEDDI需要存储文档中所有对应的位置以便以后撤销。全局替换的潜在高存储要求是该命令为什么属于“不许撤销”类型的原因。

```
class AbstractCommand {
public:
     enum CmdType { no_change, normal, no_undo };
     virtual ~AbstractCommand();
     virtual void doit();
     virtual void undo();
     CmdType getType() const { return type;}
     virtual String getName() const { return "NONAME";}
          // gives name of command for selection
          // in undo/redo menu
```

```
protected:
    CmdType type;
    AbstractCommand(CmdType t=no_change): type(t){}
};
```

当用户选择“撤销”时，用方法getName()向用户显示最近的命令。 ❑ 282

(2) 对于应用程序支持的每个请求类型，设计命令组件。将一个命令捆绑到它的供应者上有几种选择方法。供应者组件可以在命令中强制编码，或者控制器可以给供应者提供命令构造函数作为一个参数。第二种情况的例子是一个多文档编辑器，其中命令与特殊文档对象相连。

➥TEDDI的“删除”命令是将表示文本的对象作为它的第一个参数。删除字符的范围由两个附加参数指定：

```
class DeleteCmd : public AbstractCommand {
public:
    DeleteCmd(TEDDI_Text *t, int start, int end)
        : AbstractCommand(normal) , mytext(t) ,
            from (start) , to (end) {/*...*/}
    virtual ~DeleteCmd();
    virtual void doit();
        // delete characters in mytext
        // between from and to and save them in delstr
    virtual void undo();
        // insert delstr again at position from
    String getName() const { return "DELETE " + delstr;}
protected:
    TEDDI_Text *mytext;// plan for multiple text buffers
    int from,to;       // range of characters to delete
    String delstr;     // save deleted text for undo
};
```

方法doit()的实现调用了TEDDI_Text供应者对象的方法deleteText()。 ❑

一个命令对象也许会要求用户提供进一步的参数。例如，TEDDI中的“装入文本文件”命令就激活对话框以请求待装入文件的名字。在这种情况下，事件处理系统必须把用户输入传递到命令，而不是传递到控制器。因此，在创建和执行命令期间要求用户交互的命令需要额外的保护。事件处理系统的设计超出了这个模式的范围，它必须能够处理这样的情况。

可撤销命令可以用备忘录模式[GHJV95]来存储其供应者的状态，以便以后撤销而不违反封装。 283

(3) 通过提供宏命令增加了灵活性，宏命令结合了几个连续的命令。应用这种组合模式[GHJV95]可以实现这样一个宏命令组件。

➥在TEDDI中，我们实现一个宏命令类，以允许用户定义经常用到的命令序列的快捷方式：

```
class MacroCmd : public AbstractCommand {
public:
    MacroCmd(String name, AbstractCommand *first)
        : AbstractCommand( first->getType()),
                        macroname(name){/*...*/}
    virtual ~MacroCmd();
    virtual void doit();
        // do every command in cmdlist
```

```
    virtual void undo();
        // undo all commands in cmdlist in reverse order
    virtual void finish(); // delete commands in cmdlist
    void add(AbstractCommand *next) {
        cmdlist.append(next);
        if (next->getType() == no_undo) type = no_undo;
        /*... */}
    String getName() const { return macroname;}
protected:
    String macroname;
    OrderedCollection<AbstractCommand*> cmdlist;
};
```

一个MacroCmd的命令类型依赖被加进宏的命令。一个附加的no_undo类型的命令会阻止完整宏命令的撤销。否则，撤销功能会在cmdlist中以逆序迭代，撤销所有正常命令并跳过所有no_change类型的命令。 ❑

(4) 实现控制器组件。命令对象由控制器创建，例如，利用“创建型”模式——抽象工厂（Abstract Factory）和原型（Prototype）[GHJV95]。然而，因为控制器已经从供应者组件中分离出来，所以这个附加的对控制器和命令的分离是可选的。一个普通菜单控制器提供原型模式的应用例子。这样的控制器对于每次菜单输入包含了一个命令原型对象，并且每当用户选择菜单输入，都传递这个对象的一个备份到命令处理器。如果这样的菜单控制器可以用宏命令对象进行动态配置，我们就可以容易地实现用户定义的菜单扩充。

284

➡在TEDDI中，用户交互是在控制器中由回调过程进行处理的。一次回调创建相应的命令对象并将它传递给命令处理器。TEDDI利用一个全局变量theCP来指代单独的命令处理器组件。

```
void TEDDI_controller::deleteButtonPressed(){
    AbstractCommand *delcmd =
        new DeleteWordCommand(
            this->getCursor(),// pass cursor position
            this->getText());    // pass text
    theCP->perform(delcmd);
}
```

在启动时，用事件处理系统注册回调deleteButtonPressed()。 ❑

(5) 实现对命令处理器的附加服务的访问。一个用户可访问的附加服务通常由一个特殊命令类实现。命令处理器提供了“do”方法的功能。直接调用命令处理器的接口同样是一个选择。其他本质的服务（例如命令登录）由命令处理器自动执行。

➡类UndoCommand提供了对TEDDI的撤销机制的访问。这个类的实现用到了命令处理的内部机制，因此被称为它的友员。注意，UndoCommand对象不能被命令处理器存储，它属于no_change范畴。

```
class UndoCommand : public AbstractCommand {
public:
    UndoCommand()
        : AbstractCommand(no_change){}
    virtual ~UndoCommand();
    virtual void doit() { theCP->undo_lastcmd(); }
};
```

UndoCommand的方法doit()要求命令处理器撤销最后正常执行的命令。类RedoCommand提供了相反的功能。其方法doit()使命令处理器重新执行撤销命令。 ❑ 285

(6) 实现命令处理器组件。命令处理器从控制器中接收命令对象并对它们负责。对每个命令对象，命令处理器从调用do方法来开始执行。例如，一个在C++中实现的命令处理器负责删除那些不再有用的命令对象。

应用单件设计模式［GHJV95］以确保只有一个命令处理器存在。

➥对于TEDDI，我们利用两个栈实现一个多级撤销/重做操作，这两个栈一个用于执行命令，另一个用于撤销命令：

```
class CommandProcessor {
public:
    CommandProcessor();
    virtual ~CommandProcessor();
    virtual void do_cmd(AbstractCommand *cmd){
        // do cmd and push it on donestack
        cmd->doit();
        switch(cmd->getType()){
        case AbstractCommand::normal:
            donestack.push(cmd); break;
        case AbstractCommand::no_undo:
            donestack.make_empty();
            undonestack.make_empty();
            // Fall through:
        case AbstractCommand::no_change:
            // take responsibility for command objects:
            delete cmd;
            break;
        }
    }
    friend class UndoCommand; // special relationship
    friend class RedoCommand; // special relationship
private:
    // this method is only used by UndoCommand
    virtual void undo_lastcmd();
        // pop cmd from donestack,
        // undo it, and push it on undonestack
    // this method is only used by RedoCommand
    virtual void redo_lastundone(){
        AbstractCommand *last = undonestack.pop();
        if (last) this->do_cmd(last);
    }
private:
    Stack<AbstractCommand*> donestack,undonestack;
};
```

❑ 286

8. 变体

传播控制器功能。在这个变体中，控制器的功能分布在几个组件中。例如，每个用户界面元素（比如一个菜单按钮）被激活时可以创建一个命令对象。然而，控制器的作用不只限于图形用户界面的组件。

与解释器模式相结合。在这个变体中，脚本语言为每个应用程序提供了一个可编程界面。

脚本解释器的语法分析器组件起到了控制器的作用。运用这个解释器模式[GHJV95]，并由命令对象建造抽象语法树。在解释器模式中命令处理器是客户机。通过激活命令来执行解释功能。

9. 已知应用

ET++ [WGM88]提供了支持无限、有界和单一撤销和重做操作的命令处理器的框架。抽象类Command实现一个状态机，来跟踪每个命令的执行状态。这个状态机被用来检测一个命令是否被执行或撤销。控制器作用分布于一个ET++应用的事件处理程序对象层次中。

MacApp [App89]利用命令处理器设计模式来提供可撤销的操作。

InterViews [LCITV92]包括了一个动作类，这是一个提供命令组件功能的抽象基类。

ATM-P [ATM93]实现了一个命令处理器模式的简化版本。它使用命令类的一个层次来轮流传递命令对象，有时穿过进程边界。命令对象的接收器决定了如何和何时执行它。每个进程执行它自己的命令处理器。

SICAT [SICAT95]实现命令处理器模式以便在控制程序和图形SDL编辑器中提供定义良好的撤销功能。

287

10. 效果

命令处理器模式有如下**优点**：

激活请求方法的灵活性。用来请求一个功能的不同的用户界面元素可以产生相同类型的命令对象。这样容易将用户的输入重新映射到应用功能。这有助于创建一个适应用户偏好的应用程序界面。一个例子是文本编辑器，它可以提供不同的控制模式，例如WordStar或一个emacs键盘。

请求数量和功能的灵活性。控制器和命令处理器独立于单个命令功能而执行。改变一个命令的实现或者引入一个新的命令类都不会影响命令处理器或者应用程序中的其他不相关部分。例如，由现存的命令建造更多复杂的命令是可行的。除了宏机制外，这种复合命令可以预编程，因此扩充这个应用程序不需要修改功能核心。

编程与执行相关的服务。中央命令处理器容易增加与命令执行相关的服务。高级命令处理器可以将命令记录或存储到一个文件以备以后检查或重放。命令处理器可以编排命令并在稍后调度它们。如果命令应该在一个指定时间执行，或者按照优先级处理它们，或者它们将在分离的控制线程中执行，则这个功能非常有用。一个附加的例子是一个由几个并发应用程序所共享的单独的命令处理器，它提供了一种具有记录和重新运行命令的事务处理控制机制。

应用层的可测试性。命令处理器对于应用程序测试来说是一个理想的入口点。如果与上述第二种变体的解释器模式[GHJV95]结合，可以用脚本语言书写回归测试并在功能核心更改后得以应用。另外，记录由命令处理器执行的命令对象可用来分析错误情况。如果持久存储已执行命令的序列，它可以在错误更正后被重新应用，或者重用于回归测试中。

288

并发性。命令处理器设计模式允许在分离的控制线程中执行命令。响应性得到改善是因为控制器不会等待到一个命令执行完成。然而，当应用程序的全局变量（例如在一个供应者组件中）被几个并行执行的命令访问时，这就要求同步。

命令处理器模式也有一些**不足**：

效率损失。对能将组件分离的所有模式而言，附加的间接方法耗费了存储空间和时间。可以直接执行服务请求的控制器不会产生效率损失。然而，用新的请求扩充这样一个直接控制器，改变服务的实现，或者实现一个撤销机制，都需要更多的努力。

潜在的过多的命令类。一个具有丰富功能的应用程序可得出许多命令类。我们可以采取很多方法处理这种情况的复杂性：

- 围绕抽象将命令分组。
- 通过将供应者对象作为一个参数来传递，以统一非常简单的命令类。
- 预先编程宏命令对象，它们依靠少数几个低层命令的结合。

获得命令参数的复杂性。有些命令对象在执行之前或在执行期间检索来自用户的附加参数。这种情况使事件处理机制变得复杂，事件处理机制需要将事件传递到不同的目的地，例如控制器和某些被激活的命令对象。

参见

在文献[GHJV95]中，命令处理器模式建立在命令设计模式之上。两种模式都阐述了把服务请求封装成命令对象的思想。命令处理器提供了更多的用于处理命令对象的细节。命令处理器模式的控制器在命令模式中起客户机的作用。控制器决定使用哪条命令，并为每个用户请求创建一个新的命令对象。

然而，在命令模式中，客户机配置一个带有一个命令对象的调用者，执行这个命令对象能够解决几个用户请求。命令处理器接收来自控制器的命令对象并担当调用者的角色，执行命令对象。来自命令处理器模式的控制器担当客户机的角色。命令处理器模式的供应者对应于接收器，但是我们并不严格要求一条命令正好对应一个供应者。 289

致谢

最初对ET++[WGM88]的命令处理器类的研究推动了这个模式的描述。西门子公司的SICAT团队[SICAT95]指出了执行期间当一个命令需要获得来自用户的附加参数时出现的事件处理问题。 290

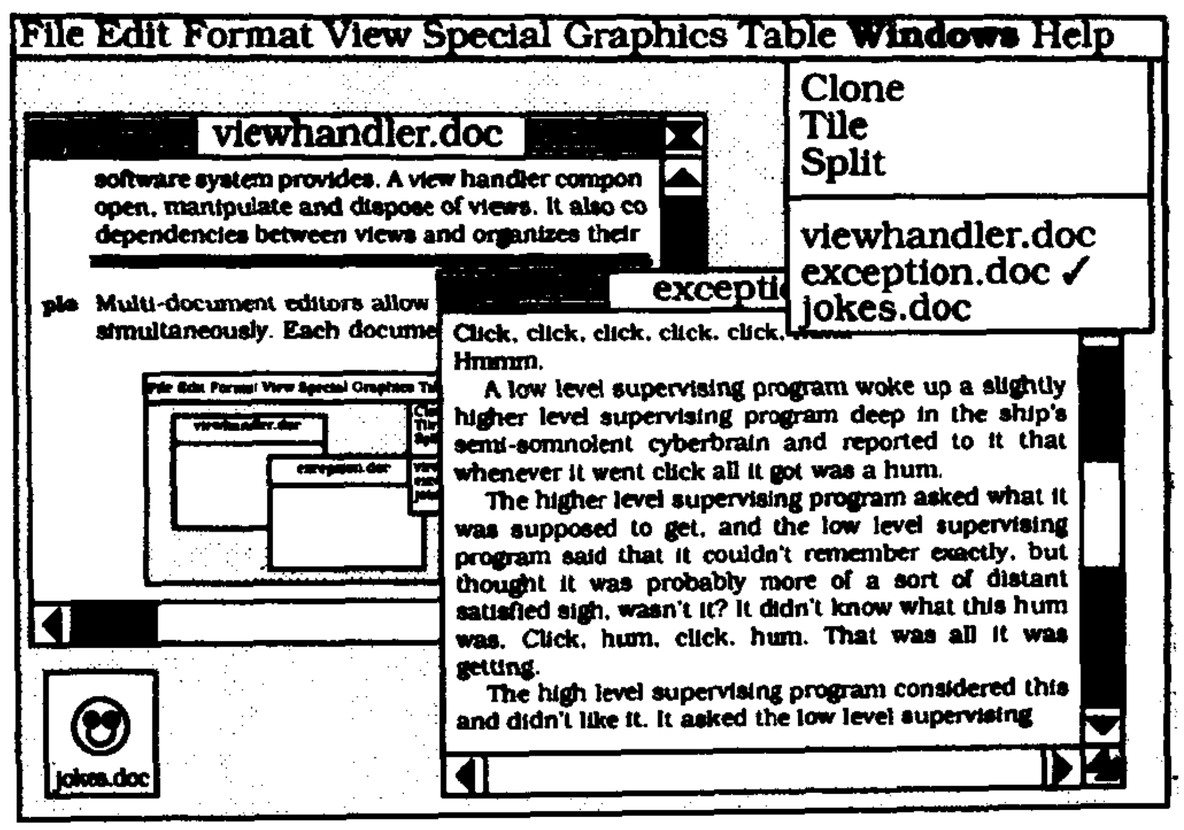
3.5.2 视图处理程序

视图处理程序（*View Handler*）设计模式有助于管理软件系统提供的所有视图。一个视图处理程序组件使得客户机能够打开、使用和关闭视图。它还协调视图间的依赖关系并组织它们的更新。

1. 例子

多文档编辑器允许多个文档同时工作。每个文档在各自的窗口内显示。

为了能有效地使用这种编辑器，用户需要对处理窗口的支持。例如，他们也许想要复制一个窗口与同一文档的几个独立视图一起工作。用户通常在退出编辑器前不关闭所打开的窗口。跟踪所有打开的文档并彻底关闭它们是系统的任务。一个窗口内的变化有可能也会影响到其他的窗口。因此，我们需要用于在窗口间传播变更的一个有效的更新机制。



2. 语境

提供特殊应用数据的多视图或者支持多文档工作的软件系统。

3. 问题

291 支持多视图的软件系统通常需要用于管理这些视图的附加功能。用户希望能够方便地打开、使用以及关闭视图，比如窗口及其内容。用户必须能够协调视图，以便对其中一个视图的更新能自动地传播给相关的视图。如下几个强制条件可以解决这个问题：

- 从用户角度，管理多视图应该简单方便，而对系统内客户机组件而言，也应如此。
- 单个视图的实现不应该互相依赖，同时不与用来管理视图的代码相混合。
- 视图的实现可以变化，并且可以在系统生存期中可以加入视图的附加类型。

4. 解决方案

将视图管理从需要表示或控制特定视图的代码中分离出来。

视图处理程序（*view handler*）组件管理由软件系统提供的所有视图。它提供了打开、协调及关闭特定视图所必需的功能，同样也提供了处理视图的功能——例如，“平铺显示”所有视图的命令，也就是说，将它们有次序地排列。

特定视图与它们的表示和控制一起封装在分离的视图组件内——针对每类视图有一个组件。*供应者*提供带有必须要呈现的数据的视图。

如同模型-视图-控制器模式所提出的，视图处理程序模式适合于从功能核心中分离出表示的思想。它本身并不为软件系统提供整体结构——它仅仅从模型和视图组件中除去管理整个视图及其相互依赖的职责。模式将这种职责赋予视图处理程序。例如，一个视图不需要管理其子视图。因此，视图处理程序模式具有比模型-视图-控制器模式更好的粒度——它有助于改进模型及其相关视图间的关系。

可以把视图处理程序组件看成是一个抽象工厂[GHJV95]及一个中介者[GHJV95]。称为抽象工厂是因为客户机不需要知道特定的视图是如何创建的。称为中介者是因为客户机不需要知道
292 视图是如何协调的。

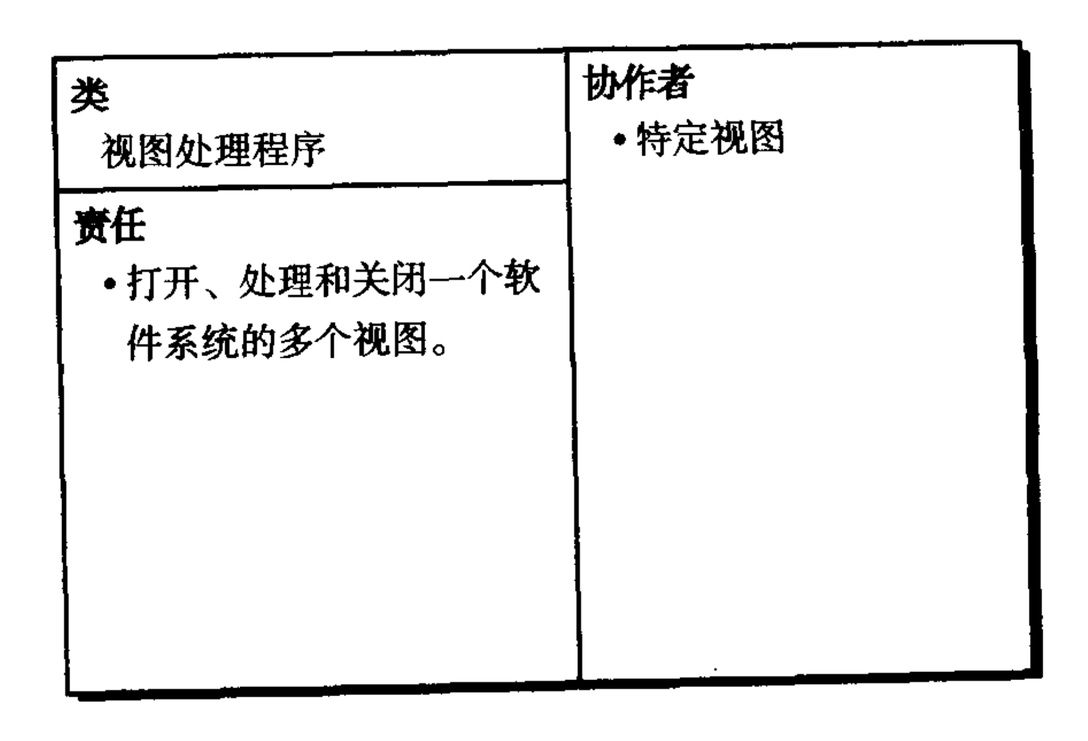
➡在文档编辑器的实例中，我们为每种类型的文档窗口提供一个视图组件。系统提供窗口，用来编辑文档、打印预览及浏览“简略的”文档页。视图处理程序管理这些视图。除了窗口的创建及删除外，视图处理程序还提供将特定的窗口放在前景、复制前景窗口以及平铺所有打开的窗口使它们不至于重叠等功能。窗口的供应者是被显示的文档。可能存在一个文档的多个同时视图，以及可以被显示的多个文档。 ❑

5. 结构

视图处理程序是这个模式的中心组件。它负责打开新的视图，并且客户机可以说明他们想要的视图。视图处理程序实例化相应的视图组件，维护它的正确的初始化，并要求显示自身的新视图。如果被请求的视图已经打开，视图处理程序将这个打开的视图放在前景窗口。如果被请求的视图打开但是却被最小化，视图处理程序让视图显示它的实际大小。

视图处理程序同样也提供关闭视图的功能，包括单个视图和所有当前打开的视图，这在退出应用程序时也是需要的。

然而，视图处理程序的主要责任是提供视图管理服务。实例包括快速地将特定视图放在前景窗口，平铺所有视图，将单个视图分成几个部分，刷新所有视图，复制视图以获得同一文档的多个视图。如果管理功能的实现跨越许多不同的视图组件，那么就很难组织这样的管理功能。



293

视图处理程序的一个附加职责是协调。视图之间可能会存在依赖性，例如几个视图显示一个复合文档的不同部分就会出现这种情形，如ET++中的VObjectText对象[WGH88]。这样的视图在平铺显示时应当相互挨个放置在一起。如果用户修改文档的其中一个视图，那么也必须按预先设定的顺序对其他视图进行更新，例如，显示最多全局信息的视图应该最先更新。然后再更新其他的视图。

抽象视图组件定义了所有视图的公共接口。视图处理程序用这个接口来创建、协调以及关闭视图。系统底层平台用该接口来执行用户事件，如重新设定窗口大小。抽象视图的这个接口必须为视图可以实现所有可能的操作提供相应的功能。

特定视图组件是从抽象视图派生而来，并实现抽象视图的接口。此外，每个视图都实现其自身的显示功能。视图从视图供应者处检索数据，准备显示这个数据并将数据呈现给用户。打开或更新一个视图时调用这个显示功能。

类 抽象视图	协作者
责任 • 定义用来创建、初始化、协调和关闭一个特定视图的接口	

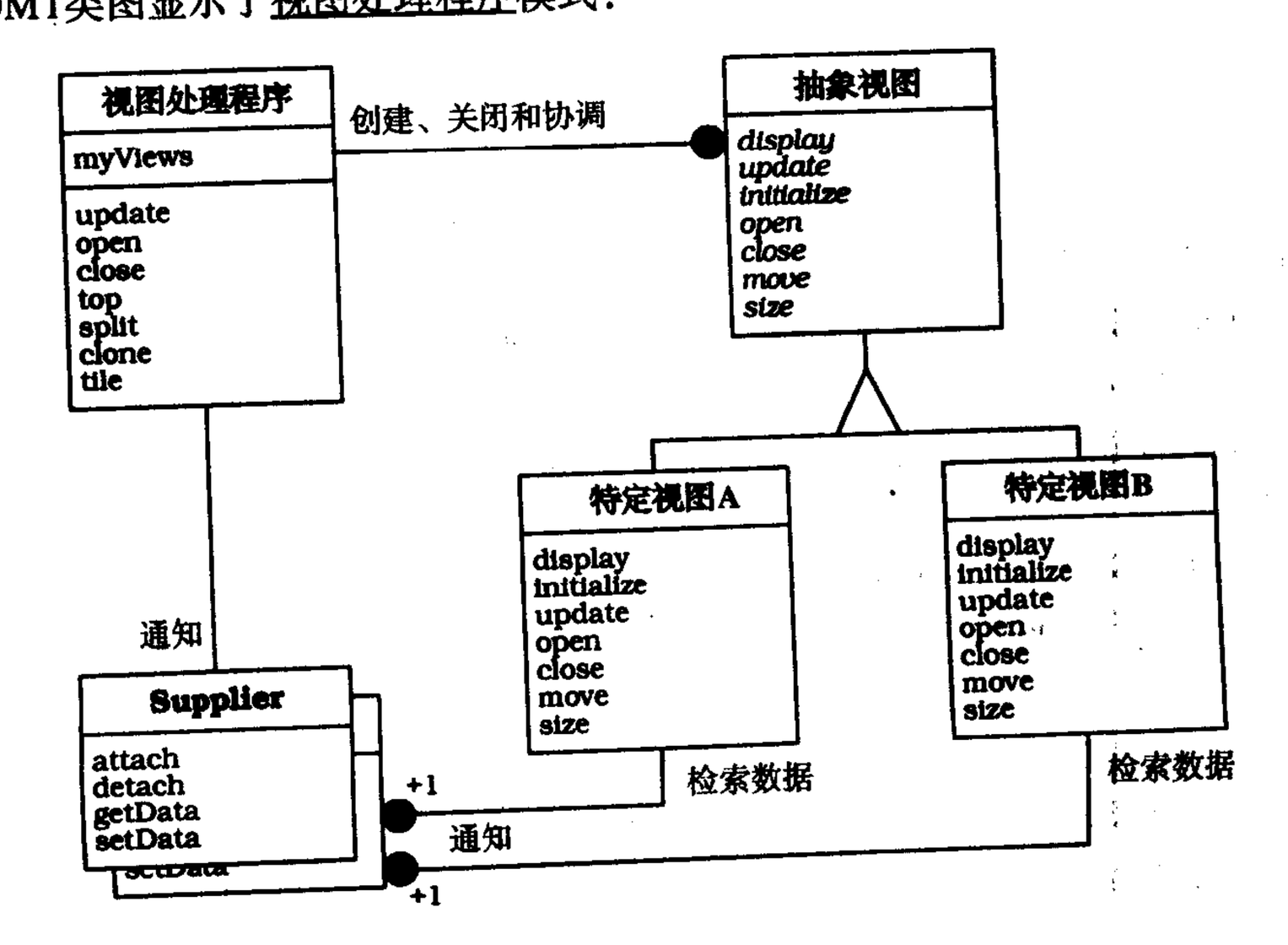
类 特定视图	协作者 • 供应者
责任 • 实现抽象接口	

供应者组件提供了由视图组件显示的数据。供应者提供了允许客户机（比如视图）获得和更改数据的一个接口。供应者通知相关组件有关其内部状态的变化。这些相关组件或是单个的视图，或是在视图处理程序组织更新时视图处理程序自身。

294

类 供应者	协作者 • 特定视图 • 视图处理程序
责任 • 实现抽象视图的接口——对每个视图都有一个类对应到系统	

下面的OMT类图显示了视图处理程序模式：

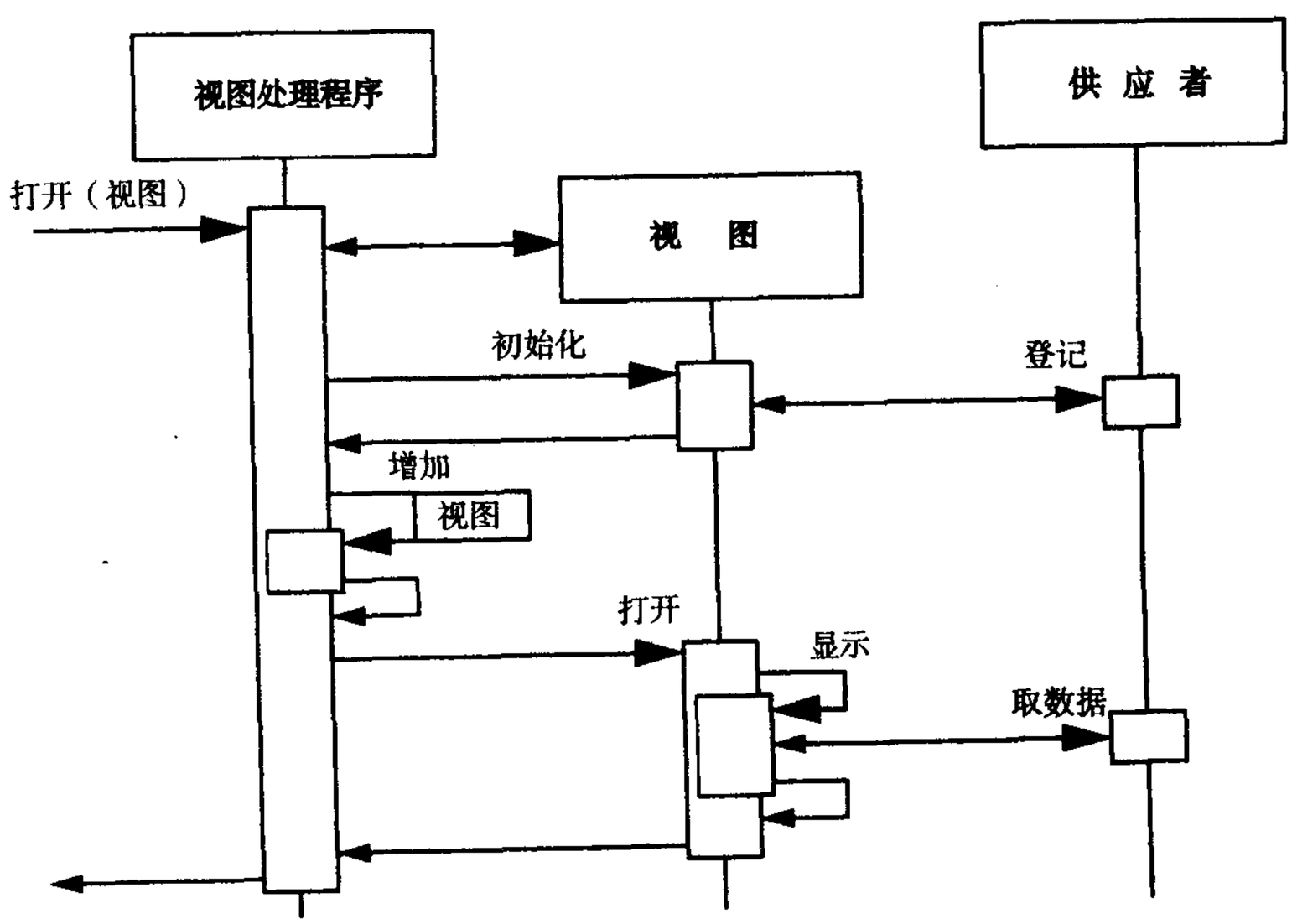


6. 动态特性

我们选择两种场景来例证视图处理程序模式的行为，这两种场景是：视图创建及平铺显示。两种场景均假定每个视图在自己的窗口中显示。 295

场景I 显示视图处理程序如何创建一个新视图。该场景由四个阶段组成：

- 客户机——可以是用户或是系统的另一个组件——调用视图处理程序打开一个特定视图。
- 视图处理程序实例化和初始化想要的视图。视图向其供应者的变更-传播机制登记，就像出版者-订阅者模式规定的那样。
- 视图处理程序将新试图添加到打开视图的内部列表中。
- 视图处理程序调用视图来显示自身。视图打开一个新窗口，从供应者那里取回数据，准备显示该数据，并将它提交给用户。

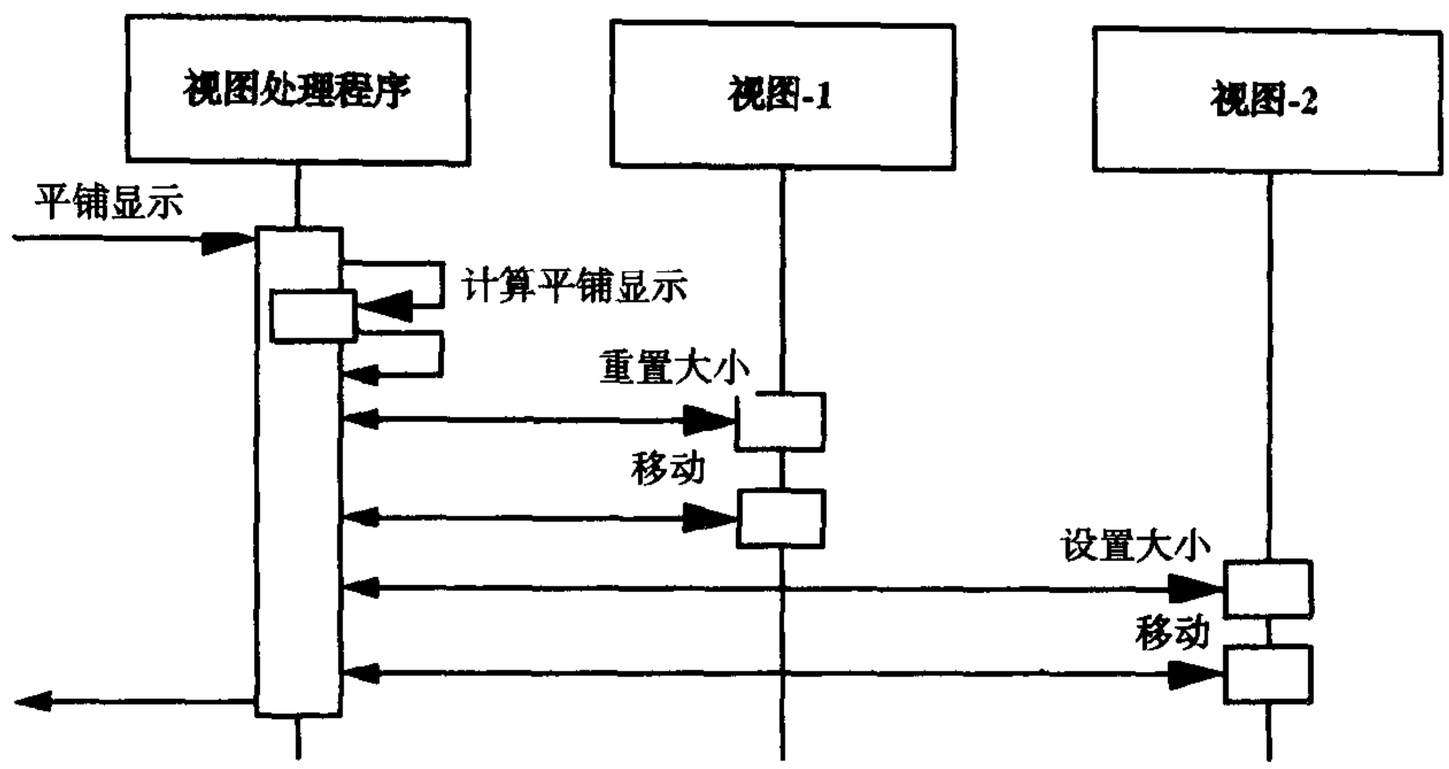


场景II 描述了视图处理程序如何去组织视图的平铺显示。为简单起见，假定只打开两个视图。该场景分为三个阶段：

- 用户调用命令以平铺显示所有打开的窗口。该请求被送往视图处理程序。
- 视图处理程序为每个打开的视图计算新的大小及位置，并调用其重置大小及移动过程。 296
- 每个视图变更其位置和大小，设置相应剪切区，并刷新其显示给用户的图像。假定视图存储了它们显示的图像。如果不是这样，视图必须在重新显示自身之前从与之相关的供应者那里获取数据。

7. 实现

可以将视图处理程序结构的实现分成四步。我们假定供应者已经存在，并且包括合适的变更-传播机制。



(1) 确定视图。规定要提供的视图的类型以及用户如何控制每个单独的视图。

(2) 为所有的视图指定一个公共接口。这应该包括打开、关闭、显示、更新和处理一个视图的功能。这个接口还可以提供初始化视图的功能。例如，它可以用特殊供应者的数据来配置视图。在一个抽象类中封装接口。对于某些功能（例如视图更新）来说，通常可以提供缺省实现的功能。

➡对于文件编辑器实例而言，我们规定AbstractView类。AbstractView类的保护接口包括用于显示和删除窗口的方法，以及用于显示窗口内容的方法。公用接口包括用于打开、关闭、移动、设置大小、拖曳和更新视图的方法以及一个初始化方法。

297

```
class AbstractView {
protected:
    // Draw the view
    virtual void displayData() = 0;
    virtual void displayWindow(Rectangle boundary) = 0;
    virtual void eraseWindow() = 0;

public:
    // Constructor and Destructor
    AbstractView() {};
    ~AbstractView() {};
    // Initialize the view
    void initialize() = 0;
    // View handling with default implementation
    virtual void open(Rectangle boundary) { /* ... */ };
    virtual void close() { /* ... */ };
    virtual void move(Point point) { /* ... */ };
    virtual void size(Rectangle boundary) { /* ... */ };
    virtual void drag(Rectangle boundary) { /* ... */ };
    virtual void update() { /* ... */ };
};
```

❑

(3) 实现视图。为每个在步骤1中已明确的视图的特定类型从AbstractView类中派生出一个单独的类。实现接口的特定视图部分，比如实例中的displayData()方法。当某些方法的缺省实现不能满足特定视图的需求时，重载那些方法。

如果视图处理程序实现特定的协调和更新策略，视图必须通知它有关可能影响其他视图的所有事件。例如，当重置视图大小时，其他视图先前隐藏的部分会变得可见。如果视图处理程序协调这些视图的更新，就必须通知它重置大小这个事件。出版者-订阅者模式有助于实现这样的更改通知。

➡在我们实例中实现了三个视图类：EditView、LayoutView和ThumbnailView，就像在解决方案小节中规定的那样。我们不需要为它们的实现而重载从AbstractView类继承而来的缺省实现。❑

(4) 定义视图处理程序。可以像工厂方法[GHJV95]一样实现创建视图的功能。客户机可以指定它们想要的视图，但它们不能控制如何创建这个视图。视图处理程序负责实例化及初始化正确的视图组件。 298

视图处理程序在内部维持对打开视图的引用。迭代器模式[GHJV95]有助于实现这个功能。视图处理程序也可以维持视图的附加信息，比如屏幕上窗口的当前位置及大小。视图处理程序的管理功能（如复制窗口的操作）使用了这样的信息。

视图处理程序可能需要实现特殊应用的视图协调策略。例如，一个视图有可能给出有关其他视图的信息，比如记录动画模拟信息。平铺显示应该将这两个相互依赖的视图叠放在一起，如果两个视图都被最小化，打开其中的一个，另一个也应该被打开。

更新策略是视图协调的另一个实例。例如，将更高的优先权赋予特定视图的更新，有可能是非常必要的。例如，用于显示报警的视图可能需要在打开其他视图前被更新。在这种情况下，供应者将这些更改通报给视图处理程序，而不是依赖的视图。视图处理程序使用其更新策略将这些需求转发给受到影响的视图。用于协调视图更新的视图处理程序通常在它们的公用接口中提供更新功能。

为使协调策略可交替，可用策略模式[GHJV95]来实现它们。中介者设计模式[GHJV95]有助于实现视图协调，例如将刷新请求广播给所有打开的视图。使用单件模式[GHJV95]确保视图处理程序类只被实例化一次。

➡文档编辑器实例中的视图处理程序提供了打开和关闭视图，以及平铺显示视图，将视图放在前景窗口、复制视图等功能。在内部，视图处理程序维持对所有打开视图的引用，包括它们的位置及大小信息以及它们是否被最小化的信息。

```
class ViewHandler {
    // Data structures
    struct ViewInfo {
        AbstractView* view;
        Rectangle     boundary;
        bool          iconized;
    };
    Container<ViewInfo*> myViews;
    // The singleton instance
    static ViewHandler* theViewHandler;
    // Constructor and Destructor
    ViewHandler();
    ~ViewHandler();
public:
```

299

```
    // Singleton constructor
    static ViewHandler* makeViewHandler();

    // Open and close views
    void open(AbstractView* view);
    void close(AbstractView* view);

    // Top, clone, and tile views
    void top(AbstractView* view);
    void clone(); // Clones the top-most view
    void tile();
};
```

下面的代码说明了新视图的创建。defaultBoundary是类Rectangle的对象，它为每个新窗口定义了缺省位置及大小。代码实现了“动态特性”小节中的场景I。

```
void ViewHandler::openView(AbstractView* view){
    ViewInfo*     viewInfo = new ViewInfo();

    // Add the view to the list of open views
    viewInfo->view          = view;
    viewInfo->boundary      = defaultBoundary;
    viewInfo->iconized      = false;
    myViews.add(viewInfo);

    // Initialize the view and open it
    view->initialize();
    view->open(defaultBoundary);
};
```

□

8. 变体

带有命令对象的视图处理程序。这个变体使用命令对象[GHJV95]来保持视图处理程序独立于特定视图接口。视图处理程序创建适当的命令并且执行它，而不是直接调用视图功能。命令本身知道如何在视图上进行操作。例如，我们可以规定一个平铺显示命令，执行它时，首先调用视图的度量尺寸函数然后是移动函数。另外也可以在创建命令后将它们传递给命令处理器，由它来负责命令的正确执行，但是也可以允许附加的功能，比如撤销运行的命令。

300

9. 已知应用

Mac机窗口管理器[App85]。窗口管理器是Mac机工具箱的一部分，工具箱可以与视图处理程序组件相比，其接口为窗口分配、窗口显示、鼠标定位、窗口移动和度量尺寸以及更新区域维护提供了功能。它也提供了一个数据结构，用来构成每个Mac机窗口的基础。部分接口用Pascal描述如下：

```
TYPE WindowRecord = RECORD
     port:          GrafPort;      {window's grafPort}
     windowKind:    INTEGER;       {window class}
     visible:       BOOLEAN;       {TRUE if visible}
     {more record elements ...}
     refCon:        LONGINT        {window's reference value}
END;
```

```
FUNCTION NewWindow( {lots of parameters} ) : WindowPtr;
PROCEDURE CloseWindow(theWindow: WindowPtr);

PROCEDURE SelectWindow(theWindow: WindowPtr);
PROCEDURE HideWindow(theWindow: WindowPtr);
PROCEDURE ShowWindow(theWindow: WindowPtr);

PROCEDURE BringToFront(theWindow: WindowPtr);
PROCEDURE SendBehind(theWindow, behindWindow: WindowPtr);

FUNCTION FindWindow(thePt: Point;
                    VAR whichWindow: WindowPtr) : INTEGER;

PROCEDURE MoveWindow(theWindow: WindowPtr;
                     hGlobal, vGlobal: INTEGER; front: BOOLEAN);
PROCEDURE DragWindow(theWindow: WindowPtr;
                     startPt: Point; boundsRect: Rect);
PROCEDURE SizeWindow(theWindow: WindowPtr;
                     w,h: INTEGER; fUpdate: BOOLEAN);

PROCEDURE BeginUpdate(theWindow: WindowPtr);
PROCEDURE EndUpdate(theWindow: WindowPtr);
```

Mac机窗口管理器并不提供在多个或所有窗口上操作的功能，它只支持处理单个窗口。因此Mac机窗口管理器被视为低层的视图处理程序组件。 301

Microsoft Word[Mic93b]。Microsoft Word字处理系统提供复制、分割、平铺显示窗口以及将打开的窗口放置在前景窗口的功能。退出Word时关闭所有打开的窗口；如果窗口包括已经改变但未存储的数据时会显示对话框来请求期望的动作。它提供了一个例子，显示视图处理程序系统如何呈现给用户，以及它所能够提供的功能。

10. 效果

视图处理程序模式具有如下的**优点**：

视图的统一处理。所有的视图共享公共接口。因此，视图处理程序及系统所有其他的组件都能够一致地处理和操作所有的视图，与它们显示的内容以及实现方法无关。

视图的可扩充性及可变更性。在带有抽象库的继承层次中的视图组件组织支持新视图的集成而不改变现有的视图和视图处理程序。由于单个的视图被封装在分离的组件中，因此变更它们的实现不会影响系统的其他组件。

特殊应用的视图协调。由于视图被中心程序管理，因此有可能实现特殊的视图协调策略。

视图处理程序模式也有如下的一些**不足**：

有限的适用性。只有当系统必须支持许多不同的视图，而视图彼此具有逻辑上的相关性，或者视图可能由不同的供应者或输出设备配置时，使用视图处理程序模式才是值得的。如果系统必须实现特定的视图协调策略，视图处理程序模式也具有使用价值。除了这些应用外，视图处理程序模式只会带来额外的实现工作并且会增加系统内部的复杂性。

效率。如果视图处理程序负责组织视图更新，则视图处理程序组件会在需要创建视图的客户机之间以及变更通知的传播链内引入间接方法层。这将导致性能的损失。然而，大部分情况下这些损失是可以忽略的。 302

参见

模型-视图-控制器(MVC)体系结构模式提供了从输入和输出行为中分离功能的基础结构。从MVC角度来看，视图处理程序模式是模型及其相关联视图之间关系的细化。

表现-抽象-控制体系结构模式根据视图处理程序模式的原理实现了多个视图的协调。用于创建及协调视图的中间层PAC agent与视图处理程序相对应。给用户呈现数据的底层视图PAC agent代表了视图组件。

致谢

303 特别感谢Dirk Riehle，他认真地评审了这个模式的早期版本。

3.6 通信

当今只有少数媒体和大型软件系统在单机上运行——绝大多数都使用计算机网络。原因如下：

- 分布式系统能够更好地实现网络资源的共享和利用。
- 快速但价格昂贵的服务器可能承担中央服务（例如数据库管理系统），而花费不多的工作站可以远程访问这些服务。
- 公司内部工作本质上是分布式的，因此实现业务逻辑的分布式软件系统与这种工作组织相匹配。

这种应用的分布式有一个重要的先决条件。分布式的子系统必须要相互协作，因此需要一种相互通信的手段。没有通信的分布式系统是难以想像的。然而，通信的问题在于有太多的机制可供选择。以UNIX为例，可能要用到TCP/IP、套接字、TLI(传输层接口，Transport Layer Interface)或RPC(远程过程调用，Remote Procedure Call)等等。

使用通信设备常常是强制连接到现有应用程序，引发各种问题。以后再想要改变通信机制是困难的甚至是不可能的，因为分布式系统直接依赖于所使用的通信机制。可移植性是另一个重要问题。最后，在通信设备允许的条件下，子系统可以从网络的一个节点移动到另一个节点。

有几种方法可以降低分布式系统组件及通信机制之间的耦合。在这里，两个最主要的方面是封装和位置透明性。通信设备的封装意味着对用户隐藏底层通信机制的细节。这经常通过在低层通信设备上提供抽象编程接口来实现。位置透明性可以让你的应用程序不知道物理位置而访问远程组件。

305

本节我们给出两种针对这些主题的模式：

- 转发器-接收器（*Forwarder-Receiver*）设计模式为对等交互模型的软件系统提供了透明的进程间通信。它引入转发器和接收器用于从下层通信机制中分离出对等体。
- 客户机-分配器-服务器（*Client-Dispatcher-Server*）设计模式在客户机与服务器之间引入一个中间层——分配器组件。通过名字服务器实现位置透明性，并且隐藏客户机与服务器间建立通信连接的细节。

转发器-接收器设计模式提供了封装，而客户机-分配器-服务器模式提供了位置透明性。如果需要支持封装和位置透明性这两个方面，可以将这两种模式相结合。

保持合作组件的相容性是通信中的另一个问题。这个问题与系统是否由分布式组件组成无关。只要是用几个组件合作解决某一特殊问题就需要考虑相容性问题。

本节我们描述一种有关这个问题的模式：

- 出版者-订阅者模式有助于合作组件的状态保持同步。要达到目标，它能单向传播变更：一个出版者通知许多订阅者有关其状态的变更。

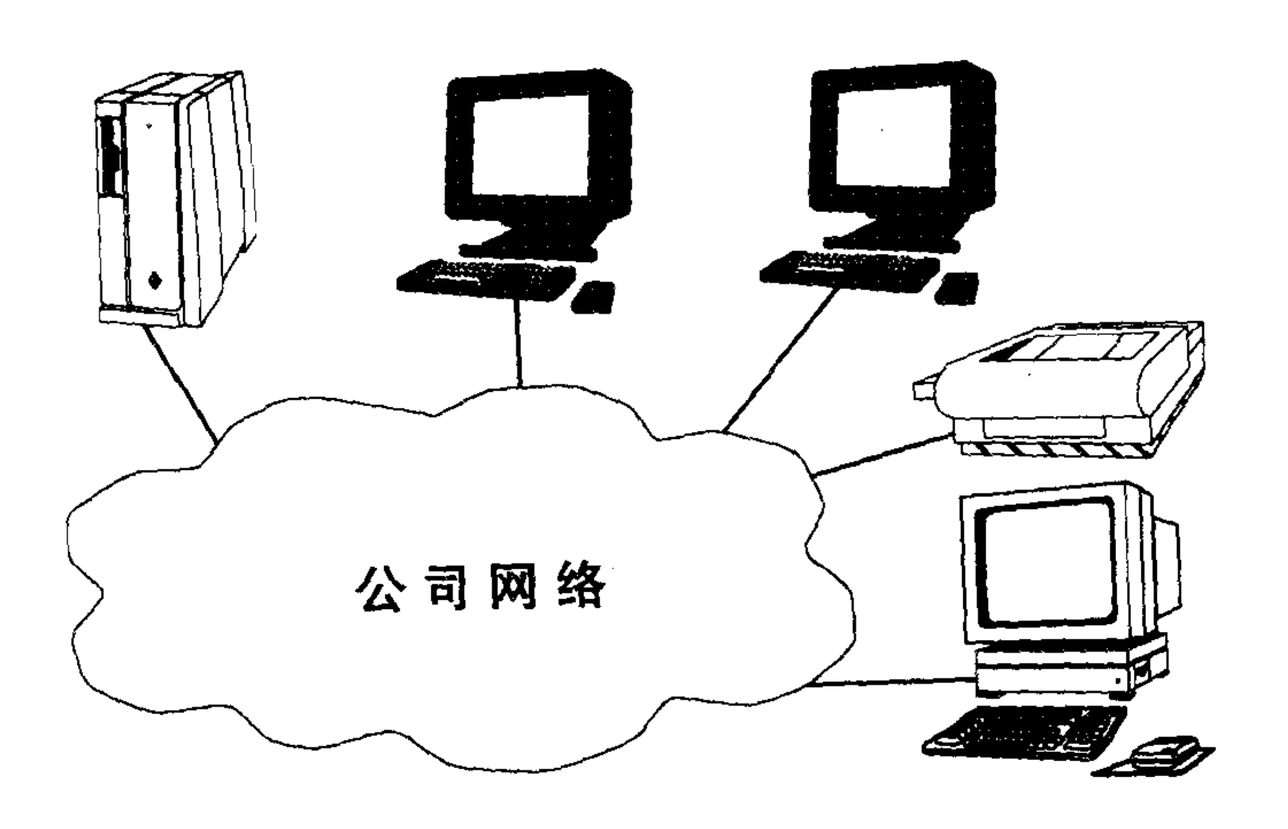
注意，出版者-订阅者模式在[GHJV95]中曾经以观察者这个名称被描述过。我们并不想用另一种形式及风格来重复现有的工作，因此我们只综述这个模式的重要部分。然而，我们还描述了出版者-订阅者的一个重要变体——事件通道（Event Channel），它并未包含在“四人帮”版本中。在许多使用代理者体系结构的分布式系统中都使用了这个变体，它同样还用在诸如OMG-Corba[OMG92]的分布式系统的基础结构中。 306

3.6.1 转发器-接收器

转发器-接收器（*Forwarder-Receiver*）设计模式为对等交互模型的软件系统提供了透明的进程间通信。它引入转发器和接收器用于从底层通信机制中分离出对等体。

1. 实例

DwarfWare公司为计算机网络管理提供了应用软件。在新的项目中，一个开发小组为网络管理已经定义了基础结构。在其他的组件中，系统由Java语言编写的agent进程组成，它可以运行在每一个可利用的网络节点上。这些agent负责观测和监督事件及资源。另外，它们允许网络管理员改变和控制网络行为，例如更改路由表。要实现信息的交换，以及管理命令的快速传播，每个agent都以对等的方式与远程agent相连接，根据需要充当客户机或服务器。由于基础结构需要支持广泛的不同的硬件和软件系统，因此对等体之间的通信不能仅依赖进程间通信的一个特殊机制。



307

2. 语境

对等通信。

3. 问题

用于建立分布式应用的最常用方法是利用现有的低层进程间通信(IPC)机制，如TCP/IP、套接字或消息队列。几乎所有的操作系统都提供这些机制，当它们与高层机制（如远程过程调用）相比时是非常有效的。然而这些低层机制通常会引入下层操作系统与网络协议间的相关性。通过使用一种特殊的IPC机制，最后的解决方案限制了可移植性，束缚了支持异构环境的系统能力，并且使以后更改IPC机制变得困难。

转发器-接收器模式在权衡如下强制条件时是有用的：

- 系统应该具备通信机制可交换性。
- 组件合作遵循一种对等模型，在此模型中发送者只需要知道其接收器的名字。
- 对等体间的通信不应对系统性能产生主要影响。

4. 解决方案

分布式对等体进行协作来解决一个特殊问题。一个对等体可充当一个客户机来请求服务，或充当一个服务器来提供服务，或身兼两职。通过将特殊的系统功能封装在单独的组件中，来隐藏发送或接收消息的下层IPC机制的细节。有关这个功能的实例是将名字映射成为物理位置，通信信道的建立，或者消息的列集和散集。

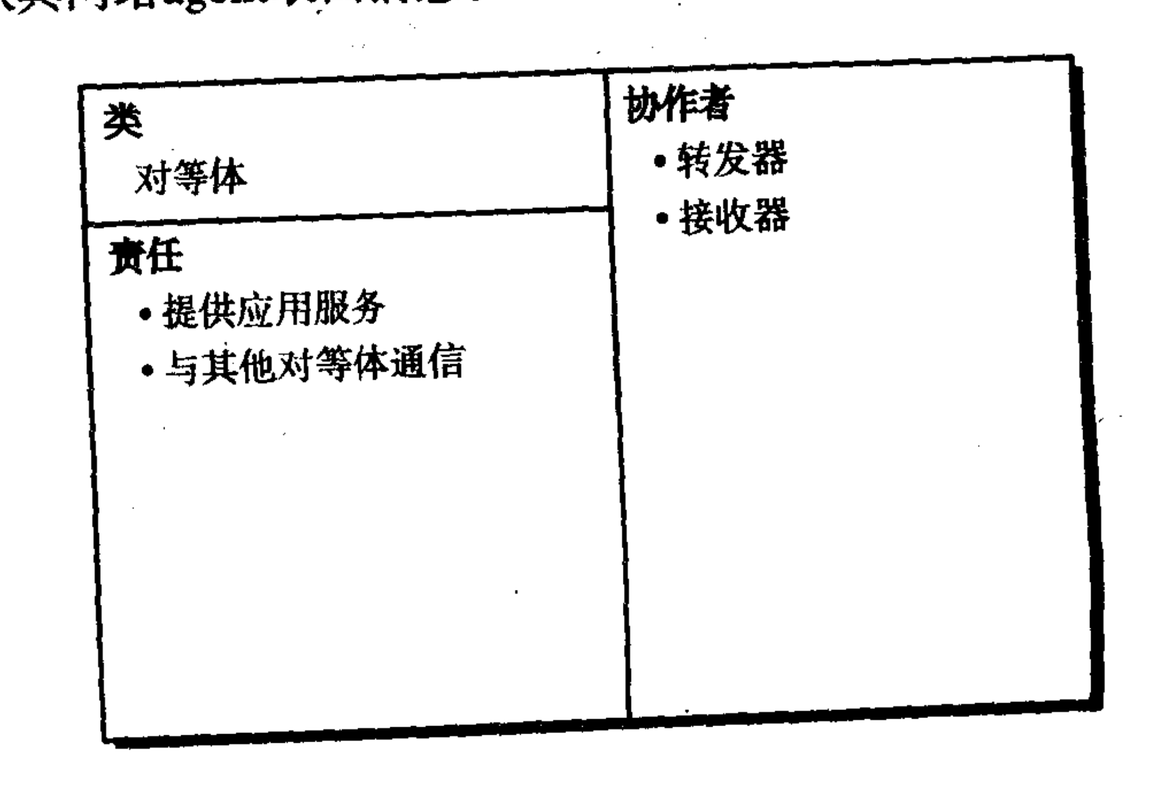
5. 结构

转发器-接收器设计模式由三种组件构成：转发器、接收器和对等体：

对等体组件负责应用任务。要实现它们的任务，对等体需要与其他的对等体通信。它们可能处于不同的进程甚至不同的机器上。每一个对等体知道需要进行通信的远程对等体的名字。它使用转发器向其他对等体发送消息并且使用接收器接收由其他对等体传送来的消息。这些消息或者是一个对等体发送给远程对等体的请求或者是对等体传送到请求发起者的响应。

308

➡在DwarfWare实例中的对等体是运行在网络节点上的agent。它们不停地监控网络事件和资源，侦听来自远程的agent的新来消息。每个agent都可以和任何其他的agent相连接来交换消息和请求。网络管理基础结构将所有其他的agent与网络管理员控制台相连接。网络管理员的任务是控制网络活动及事件。为了达到这个目的，管理员通过使用可利用的网络管理工具向网络agent发送请求或者从其网络agent取回消息。 ❑



转发器组件跨越进程边界发送消息。一个转发器提供一个通用接口，该接口是一个特殊IPC机制的抽象，并且包括列集消息功能和传送消息的功能。转发器还包含从名字到物理地址的映射。当转发器向远程对等体发送消息时，该转发器通过使用其名字到地址的映射来确定接收端的物理地址。在发送的消息中转发器指定自己对等体的名字，这样远程对等体能够向消息发起者发送一个响应。

➡在我们实例中存在几种不同形式的消息：

- 命令消息指示接收者执行诸如改变其主机路由表的行动。
- 情报消息包含网络资源及网络事件上的数据。
- 响应信息允许agent应答消息的到达。

转发器组件负责向远程网络agent转发所有这些消息而不引入对下层IPC机制的任何依赖。 ❑

接收器组件负责接收消息。一个接收器提供一个通用接口，该接口是一个特殊IPC机制的抽象。它包括接收消息的功能和散集消息的功能。

➡实例中的接收器等待代表其agent进程的新来消息。一旦消息到达，它们将接收到的数据流转换成一个通用的消息格式并将它们转发给接收器agent进程。 ❑

类 转发器	协作者 • 接收器
责任 • 为发送消息提供一个通用接口 • 列集和传送消息到远程接收器 • 将名字映射到物理地址	

类 接收器	协作者 • 转发器
责任 • 为接收器消息提供一个通用接口 • 接收和散集从远程转发器来的消息	

转发器-接收器设计模式的静态关系如下图所示。

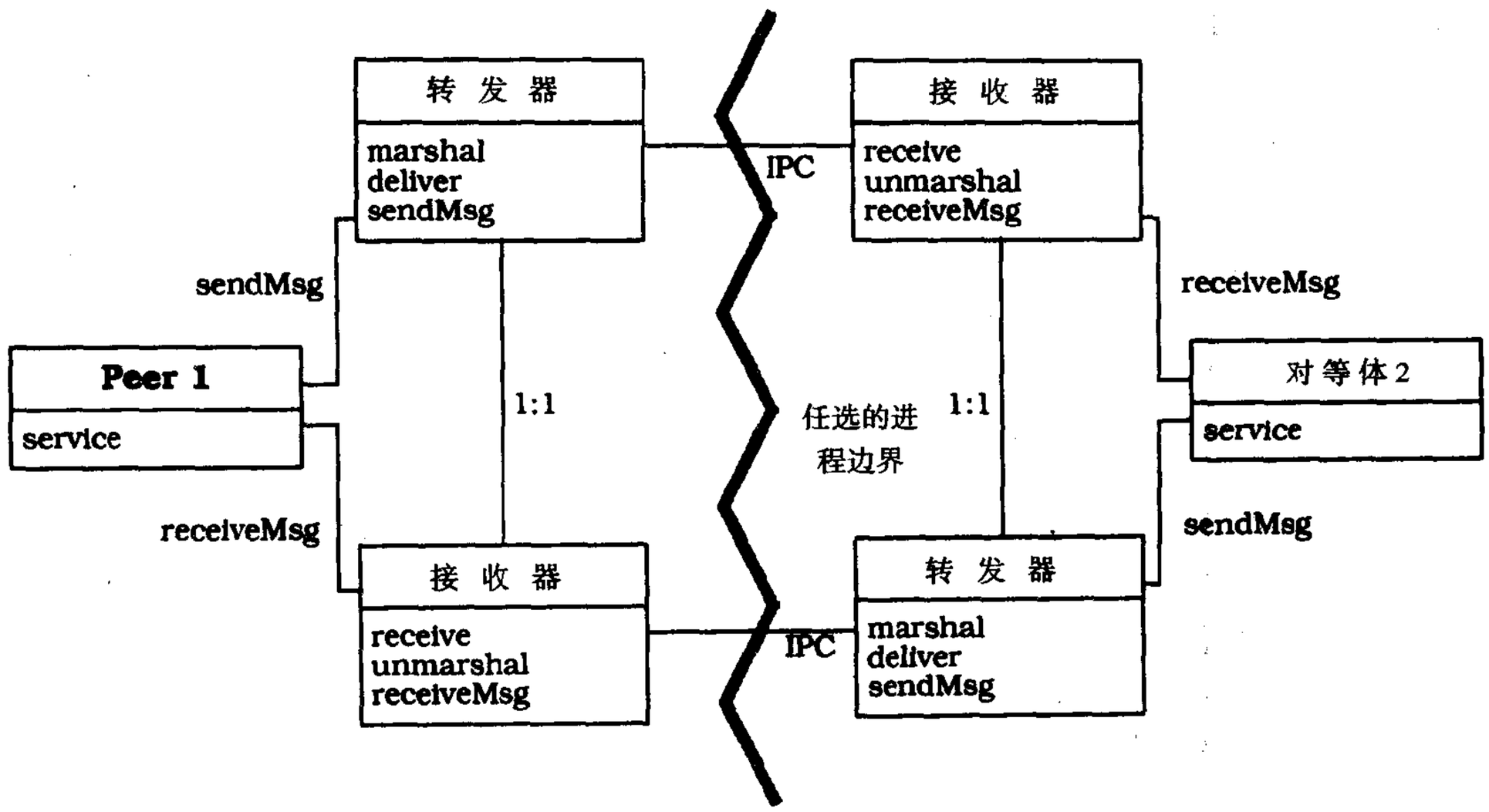
要将消息发送给远程对等体，对等体调用其转发器的方法sendMsg，将消息作为变元传递。方法sendMsg必须将消息转换成为下层IPC机制能够识别的格式。为此，它调用marshal. sendMsg 使用deliver把IPC消息数据传送给远程接收器。

当对等体想接收一条来自远程对等体的消息时，它调用其接收器的receiveMsg方法，然后返回消息。receiveMsg调用receive，后者使用下层IPC机制的功能来接收IPC消息。消息收到后，receiveMsg调用unmarshal将IPC消息转换成为对等体能够理解的格式。

6. 动态特性

下面的场景说明了使用转发器-接收器结构的实例。两个对等体P1及P2相互通信。为此，P1使用转发器Forw1和接收器Recv1。P2处理所有用转发器Forw2和接收器Recv2传送的消息：

- P1从远程对等体P2请求服务。为此，P1向它的转发器Forw1发出请求，并且指定接收者



的名字。

- Forw1确定远程对等体的物理位置并且列集消息。
- Forw1将消息传递给远程接收器Recv2。
- 早些时候P2已经请求其接收器Recv2等候即将到来的请求。现在，Recv2接收来自Forw1的消息。
- Recv2散集消息并将它转发给它的对等体P2。
- 此时，P1调用其接收器Recv1来等候一个响应。
- P2执行被请求的服务，并将结果及接收者P1的名字发送给转发器Forw2。转发器列集结果并将它传送给Recv1。
- Recv1接收来自P2的响应，散集响应并将它传送给P1。

311

312

7. 实现

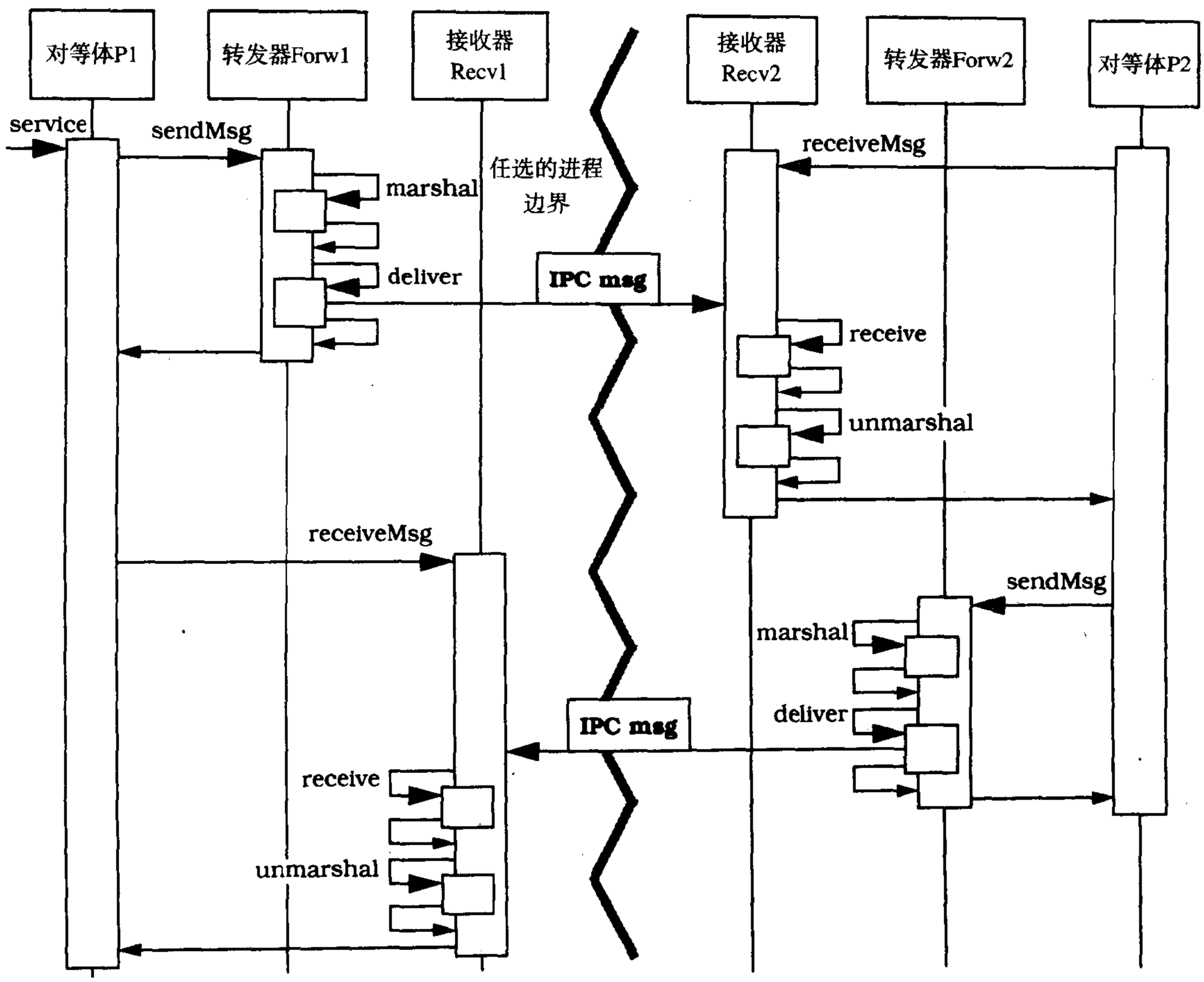
要实现转发器-接收器设计模式，迭代如下的步骤：

(1) *指定名字到地址的映射*。因为对等体通过名字引用其他的对等体，所以需要引入一个适当的名字空间。名字空间定义在给定语境中名字必须符合的规则和约束。例如，可以规定所有的名字恰好由15个字符组成并且由大写字母开头，如“PeerVideoServer”。还可以是按UNIX风格的路径名字来构建名字，如“/Server/VideoServer/AVIServer”。

一个名字不一定是指单一地址——它也可以是指一组地址。当一个对等体发送带有表示一组远程对等体的目标名字的消息时，消息被发送给组内每个成员。甚至可以引入一个分层结构使得一组成为另一组的组成员。

(2) *指定在对等体及转发器间使用的消息协议*。该协议规定了转发器收到其对等体信息数据的明细结构。对接收器及其对等体之间使用的消息协议执行同样的任务。

➡DwarfWare实例过于简化，它没有涉及到出错处理，以及诸如将数据分割成多个数据包的



通信细节。对等体调用其转发器时，它们使用类Message的对象。接收器接收到一个消息时，它将一个Message对象返回到它的对等体。实例中消息仅仅包括发送者及消息数据，它们均用统一编码字符串来表示。消息并不包括接收者的名字，因为发送者将这个名字作为额外的变元传送给它的转发器。这使得我们可以将同样的消息发送给多个接收者。

```
class Message {
     public String sender;
     public String data;
     public Message(String theSender, String rawData) {
          sender = theSender;
          data   = rawData;
     }
}
```

313

我们还需要远程对等体的转发器与接收器间的通信协议。转发器发送给一个远程接收器的消息也包括发送者的名字。

每个消息被作为字节序列传送，其中前四个字节指定消息的总长度，其后的字节包括消息发送者及信息数据本身。 ❑

```
        {
                radio = new Radio();
                // Set other member variables...
        }

        //---------- Delegate to Radio ----------
        public void RadioOn()
        {
                radio.On();
        }

        public void RadioOff()
        {
                radio.Off();
        }
}
```

用户现在可以用包含/委托方式实现完整的radio功能。现在是增加AlarmClock功能的时候了。首先，很快加上一个radioAlarm私有变量来决定当警报不存在的时候，收音机是否开始警报或蜂鸣器是否发出响声：

```
class ClockRadio : AlarmClock
{
        private bool radioAlarm;
        // Declare other member variables...
        public ClockRadio()
        {
                radioAlarm = false;
                // Set other member variables...
        }
        //---------- New ClockRadio functionality -------
        public void SetRadioAlarm(bool useRadio)
        {
                radioAlarm = useRadio;
        }
}
```

因为用户想在AlarmClock中覆盖SoundAlarm( )函数，故需要改变SoundAlarm( )方法，即声明它是受保护的，进一步地，因为用户想在Run( )函数中产生多态性行为，故要求该方法是虚拟的：

```
class AlarmClock
{
        private int currentTime;
        private int alarmTime;

        protected virtual void SoundAlarm()
        {
                Console.WriteLine("Buzz!");
        }

        // Other Methods...
}
```

```
class Registry {
    private Hashtable hTable = new Hashtable();
    public void put(String theKey, Entry theEntry) {
        hTable.put(theKey,theEntry);
    public Entry get(String aKey) {
        return (Entry) hTable.get(theKey);
    }
}
```

现在我们引入Forwarder类。Forwarder类的构造函数需要一个指定对等体逻辑名的string变元theName。当对等体调用sendMsg方法时，发生如下事件：

- 方法sendMsg调用marshal将消息theMsg转换成为字节序列。
- 调用deliver。这个方法在本地库中查询与远程对等体theDest相关联的物理位置。

为此，全局类fr包含作为Repository实例的数据成员fr.reg。deliver打开一个套接字端口，与远程对等体相连接，传送消息，并关闭所有套接字。

```
class Forwarder {
    private Socket s;
    private OutputStream oStr;
    private String myName;
    public Forwarder(String theName) { myName = theName;}
    private byte[] marshal(Message theMsg) { /* ... */ }
    private void deliver(String theDest, byte[] data) {
        try {Entry entry = fr.reg.get(theDest);
            s = new Socket(entry.dest(),entry.port());
            oStr = s.getOutputStream();
            oStr.write(data);
            oStr.flush();
            oStr.close();
            s.close();
        }
        catch(IOException e) { /* ... */ }
    }
    public void sendMsg(String theDest, Message theMsg) {
        deliver(theDest, marshal(theMsg));
    }
}
```

316

它有助于将转发器的责任彼此分开，如列集、消息发送和库。所有这些功能可以被分解成为具体的IPC机制。使用整体-部分设计模式来封装转发器的一个分离部分组件的每个职责。

(5) 实现接收器。封装接收器中用来接收IPC消息的所有功能。给接收器提供一个通用接口，该接口从特殊IPC机制的细节中抽象出来。接收器需要包括接收及散集IPC消息的功能。使用整体-部分设计模式，每个职责可以封装到接收器的分离部分组件之中。(见步骤4)。

设计接收器时，需要特殊考虑其他两个方面。由于所有的对等体异步运行，因此必须要决定在消息到达之前接收器是否应该阻塞：

- 如果是的话，接收器等待新来的消息。只有在收到消息时它才将控制权返回给其对等体。换句话说，除非消息接收成功，否则对等体不能继续工作。如果对等体依赖于这个新来的消息才能继续工作，则这个行为是合理的。
- 除此以外，应该实现无阻塞的接收器，它允许对等体去指定超时值(参见步骤2)。如果在指

定的时间内消息没有到达，接收器给其对等体返回一个异常。

如果下层IPC机制不支持无阻塞的I/O，则可以在对等体内使用一个独立的线程来处理通信。

接收器内使用不止一个通信信道是另一个重要的设计问题。这样的接收器可以多路分解通信信道——它们处于等待状态直到一个消息到达某个信道，然后将消息返回给其对等体。如果出现多个消息同时到达的情况，接收器会提供一个内部的消息队列来缓冲消息。是否能多路分解依赖于下层IPC机制。例如，UNIX系统调用select可让一个进程去等待文件和套接字描述符集合上的事件。如果IPC机制不支持多路分解，可以在接收器内提供多线程，其中每个线程负责一个特殊的通信信道。有关多路分解事件更详细的细节，可参见反应器模式[Sch94]。

317

➡在我们的实例中，类Receiver提供接收器的组件。如果一个对等体实例化一个接收器，它调用构造函数并将自己的名字作为参数传递。接收器使用这个名字去确定哪个套接字端口应该用于消息接收。当对等体需要取回消息时，它调用其Receiver对象的receiveMsg方法，它回过来又调用receive。方法receive做两件事：

- 从全局库里取回套接字端口数之后，打开一个服务器套接字并且等待与远程对等体的连接。
- 一旦与第二个套接字建立连接，从通信信道读出新来消息及其大小。receive将数据返回给receiveMsg。

最后，receiveMsg调用unmarshal将字节序列转换成为Message对象并将该对象返回给对等体。

```
class Receiver {
    private ServerSocket srvS;
    private Socket s;
    private InputStream iStr;
    private String myName;
    public Receiver(String theName) { myName = theName;}
    private Message unmarshal(byte[] anArray) { /* .. */ }
    private byte[] receive() {
        int val;
        byte buffer[] = null;
        try {
            Entry entry = fr.reg.get(myName);
            srvS = new ServerSocket(entry.port(), 1000);
            s = srvS.accept();iStr = s.getInputStream();
            val = iStr.read(); buffer = new byte[val];
            iStr.read(buffer);
            iStr.close(); s.close(); srvS.close();
        }
        catch(IOException e) { /* ... */ }
        return buffer;
    }
    public Message receiveMsg() {
        return unmarshal(receive());
    }
}
```

318

(6) 实现应用对等体。将对等体分割成两个集合——客户机与服务器。两个集合的交可以非空。如果一个对等体充当客户机，则它将消息发送给一个远程对等体并等待响应。收到响应，

继续完成它的任务。充当服务器的对等体不停地等待新来的消息。这样的消息到达时，它执行一个依赖于所接收消息的服务，并发送一个响应给请求发起者。注意服务器也可以是其他服务器的客户机。还有可能服务器与客户机动态地变化它们的角色。

两个对等体间的通信不一定总是双向的。有时仅需要由一个对等体给另一个对等体发送消息而不需要响应——单向通信。这里，对等体发送一个消息然后接着工作。消息的接收者从其接收器取回消息，但并不向消息的发起者发送响应。可以使用单向通信来实现发送者及接收者之间的异步通信。

➡下面是一个对等体充当服务器的实例：

```
class Server extends Thread {
    Receiver  r;
    Forwarder f;
    public void run() {
        Message result = null;
        r = new Receiver("Server");
        result = r.receiveMsg();
        f = new Forwarder("Server");
        Message msg = new Message("Server","I am alive");
        f.sendMsg(result.sender, msg);
    }
}
```
❑

(7) 实现启动配置。系统启动时，转发器和接收器必须用一个有效的名字到地址的映射进行初始化。引入一个单独的启动例程来创建库并输入所有的名字/地址对。这样的配置例程可以从外部文件读入这些名字/地址对，改变映射时不需要接触到源代码。

如果软件系统允许不同的对等体具有不同的名字到地址的映射，启动配置必须能够根据这个需求来初始化库(见步骤4)。

319

如果需要配置能够被动态地改变，则实现用来在运行期间修改库的附加功能。

➡在DwarfWare实例中，我们引入下面的配置类，允许我们向中央库注册服务器和客户机：

```
class Configuration {
    public Configuration() {
        Entry entry = new Entry("127.0.0.1",1111);
        fr.reg.put("Client",entry);
        entry = new Entry("127.0.0.1",2222);
        fr.reg.put("Server",entry);
    }
}
```
❑

8. 已解决的例子

在网络管理的基础结构中，一个公共协议确定请求、情报消息及响应的格式。如果一个agent想要从远程agent取回信息，比如当前资源争用，它给接收者发送一个消息。接收者从其接收器取回消息，将请求的信息打包成为响应并将响应发送给消息发起者。当agent传送一个命令信息时，接收者接收到消息，对它解释并执行适当的命令。接着便告诉发送者它是否能够成功地执行命令。所有相关的信息都使用图形接口显示在网络管理员的平台上。为了增加适用性，网络中的每台机器均可主持网络管理控制平台。

9. 变体

无名字-地址映射的转发器-接收器。有时性能问题比能够封装下层IPC机制细节更为重要。要实现这一点，可以取消转发器及接收器内名字到物理位置的映射。在这种配置下，对等体需要告诉其转发器接收者的物理位置。然而，这个变体可能会极大地降低更改IPC机制的能力。

320

10. 已知应用

软件开发工具包TASC[TASC91]支持工厂自动化系统的分布式应用中转发器-接收器结构的实现。

作为REBOOT项目[Kar95]一部分开发出的用于柔性制造的物流控制软件，使用转发器-接收器结构有利于一个高效的IPC。

ATM-P交换系统[ATM93]使用转发器-接收器设计模式去实现静态分布式组件间的IPC，例如进程管理与通信agent间的通信。

转发器-接收器设计模式被用来实现分布式Smalltalk环境BrouHaHa[Stee91]内的进程间通信。

11. 效果

转发器-接收器设计模式有两个**优点**：

有效的进程间通信。这个模式提供了非常有效的进程间通信。组件间的通信以对等方式来构建，其中IPC消息的每个转发器都知道其潜在接收器的物理位置。因此，转发器不需要定位远程组件。然而，IPC功能与对等体的分离引入了间接方法的附加层。然而，相对于实际IPC的时间消耗，这个开销在大多数情况下应该忽略不计。

KPC设备的封装。所有具体IPC设备的附属物都被封装在转发器和接收器内。下层IPC机制的改变不会影响应用程序的其他组件，尤其不会影响通过转发器和接收器相互通信的对等体。

但是，转发器-接收器设计模式有一个很大的**不足**：

不能支持灵活的组件重新配置。如果对等体的分布在运行期间变更，那么转发器-接收器系统是很难适应的。这样的改变潜在地影响所有对等体与“迁移后”的对等体的协作。正如客户机-分配器-服务器设计模式中描述的那样，这个问题可以通过为转发器-接收器结构增加一个中心分配器组件来解决。

321

参见

客户机-分配器-服务器设计模式为软件系统提供透明的进程间通信，其中编译时组件的分布是不知道的，也许在运行期间动态地改变。可以按照下面的描述将这个模式与转发器-接收器设计模式结合应用。

客户机-分配器-服务器设计模式可以被实例化，方法是转发器充当客户机而接收器充当服务器。当一个对等体请求其转发器发送消息时，转发器促使分配器将接收者名字映射到它的物理位置并建立与远程接收器的通信信道。这样的安排允许对等体在运行期间不用注册便可迁移到其他位置，然后用分配器进行再次注册。

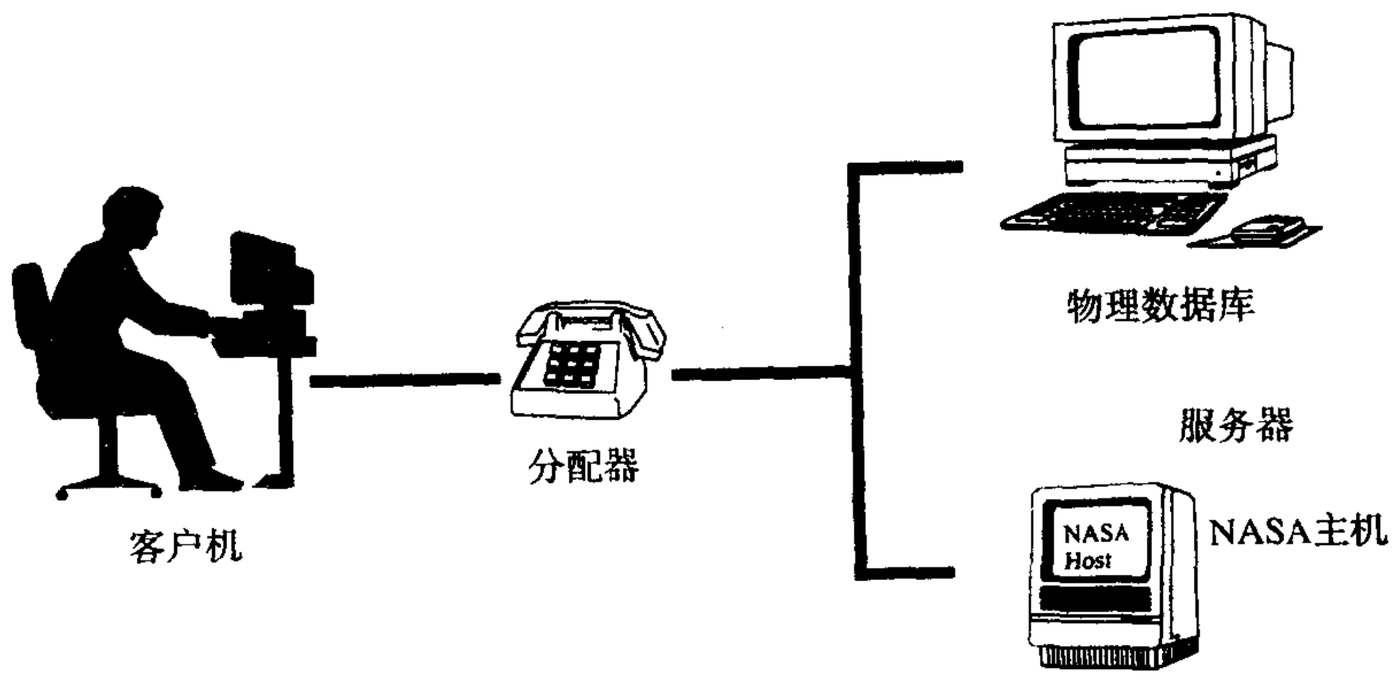
322

3.6.2 客户机-分配器-服务器

客户机-分配器-服务器（*Client-Dispatcher-Server*）设计模式在客户机与服务器间引入一个中间层——分配器组件。借助名字服务实现位置透明性，并且隐藏客户机与服务器间建立通信连接的细节。

1. 例子

设想我们为检索新科学信息开发一个软件系统ACHILLES。信息提供者位于局域网和遍布世界各地的广域网上。要访问某一信息提供者，就有必要指定其位置及要执行的服务。信息提供者接收到来自客户机应用的请求时，它运行适当的服务并给客户机返回被请求的信息。



2. 语境

一个软件系统集成了一组分布式服务器，这些服务器在本地运行或是分布在整个网络上。 323

3. 问题

当一个软件系统使用分布在网络中的服务器时，它就必须提供一种用来相互通信的手段。通常通过运用现有的通信设备，组件间的连接在发生通信前已经被建立。然而，组件的核心功能应该与通信机制的细节相分离。客户机不需要知道服务器的位置。这样就可以动态地改变服务器的位置，并提供针对网络或服务器失效的恢复能力。

我们必须权衡如下强制条件：

- 一个组件应该能够使用独立于服务供应者位置的服务。
- 实现服务用户的功能核心的代码应该与用来建立同服务供应者连接的代码相分离。

4. 解决方案

提供一个分配器组件充当客户机与服务器间的中间层。分配器实现名字服务，它允许客户机用名字来指定服务器而不要提供物理位置，由此实现位置透明性。此外，分配器还要负责建立客户机及服务器间的通信信道。

将服务器加入到为其他组件提供服务的应用程序中去。每个服务器惟一地由其名字指定，并通过分配器同客户机建立连接。

客户机依赖分配器去定位某一特定服务器并同服务器建立通信链路。与传统的客户机-服务器计算技术相比，这里，客户机和服务器的角色可以动态地互换。

5. 结构

客户机的任务是去执行特殊领域的任务。为了执行处理任务，客户机访问由服务器提供的操作。在发送请求给服务器之前，客户机向分配器请求信道。客户机利用这个信道同服务器通信。

324 服务器为客户机提供一组操作。它既可以注册自身，也可以通过其名字和地址向分配器进行注册。一个服务器组件可以与客户机位于同一台计算机，也可以位于网络上某个节点处。

类 客户机	协作者 • 分配器 • 服务器
责任 • 执行系统任务 • 请求从分配器到服务器连接 • 调用服务器的服务	

类 服务器	协作者 • 客户机 • 分配器
责任 • 为客户机提供服务 • 向分配器注册自己	

分配器提供在客户机和服务器间建立通信信道的功能。要实现这一点，需要取出服务器组件的名字并将这个名字映射成为服务器组件的物理地址。分配器使用现有的通信机制为服务器建立通信链路并将通信句柄返回给客户机。如果分配器无法启动与被请求服务器间的通信链路，则它将通知客户机它所遇到的错误。

为了提供它的名字服务，分配器实现用来注册和定位服务器的功能。

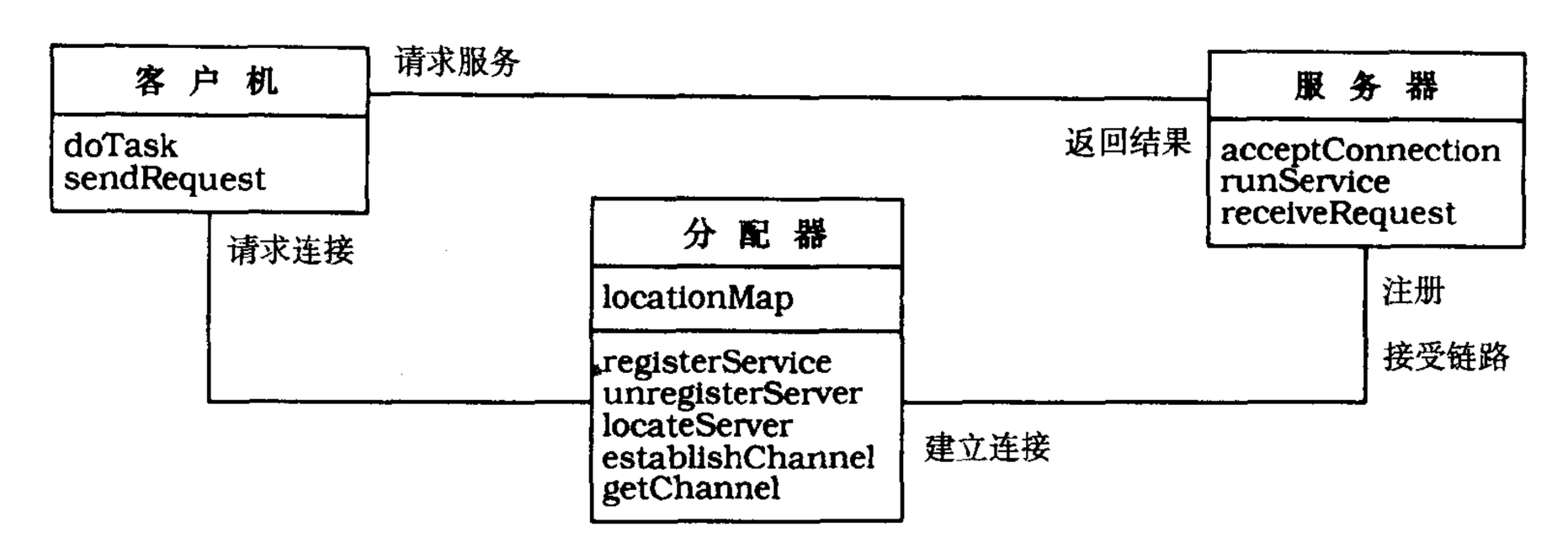
类 分配器	协作者 • 客户机 • 服务器
责任 • 建立客户机与服务器之间的通信信道 • 定位服务器 • 注册（或注销）服务器 • 维护服务器位置的一个映射	

325

客户机、服务器和分配器间的静态关系如下图所示：

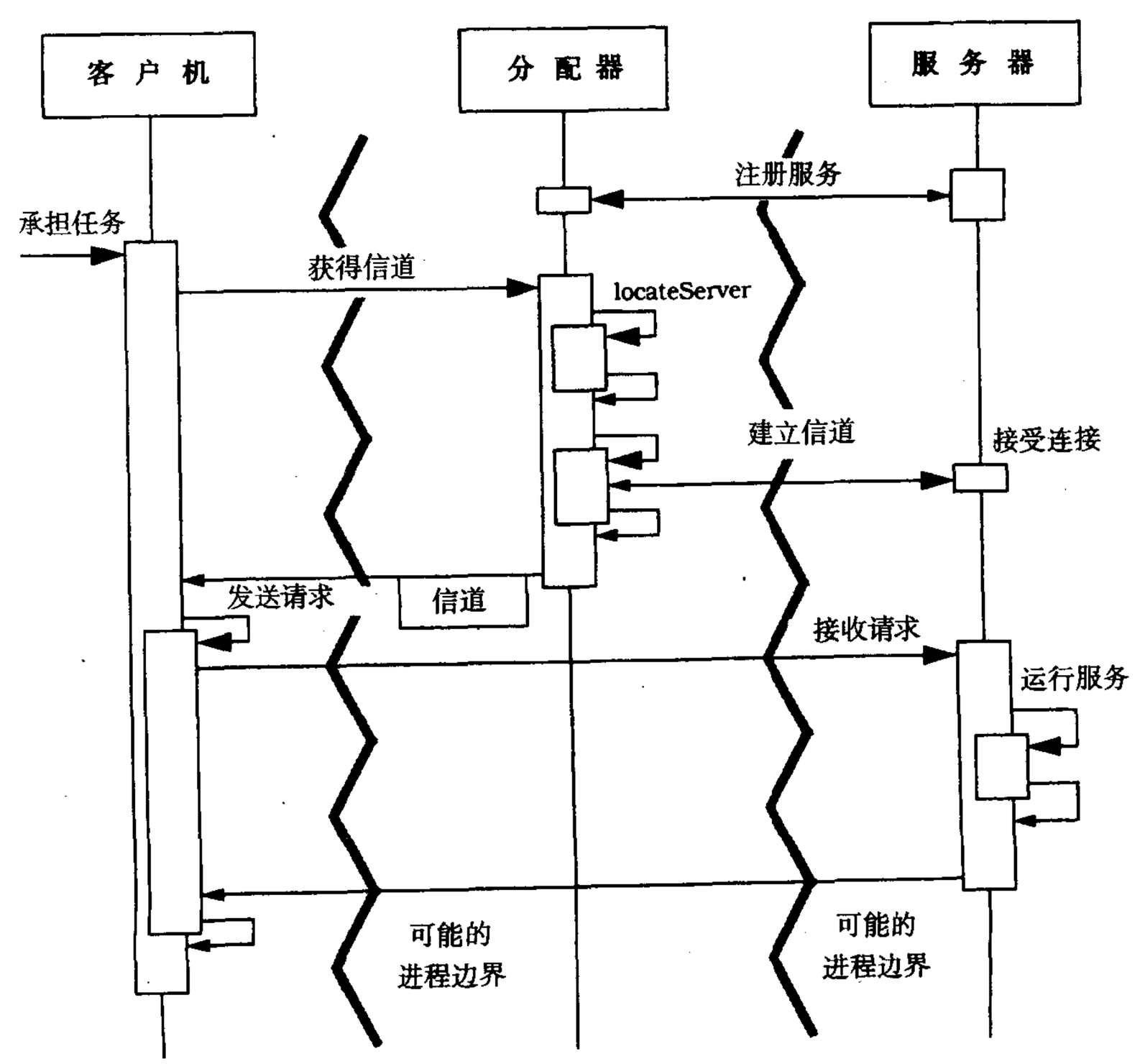
6. 动态特性

客户机-分配器-服务器设计模式的一个典型场景包括如下几个阶段：



- 服务器向分配器组件注册自己。
- 稍后，客户机向分配器请求同指定的服务器建立通信信道。
- 分配器在其注册表中查找同客户机指定名字相关联的服务器。
- 分配器建立同服务器相连的通信链路。如果分配器成功启动连接，则它将通信信道返回给客户机。如果不成功，它给客户机发送一条出错消息。
- 客户机使用通信信道直接给服务器发出一条请求。
- 识别新来的请求后，服务器执行适当的服务。
- 服务执行完成时，服务器将结果发送回客户机。

326



7. 实现

要实现客户机-分配器-服务器结构，应用如下的步骤。不一定需要按照给定的次序，因为其中有些步骤是相互关联的。

(1) 将应用分成服务器和客户机两部分。定义哪个组件应该作为服务器实现，并确定哪些客户机访问这些服务器。可以预先定义这些划分，因为构建中的应用程序可能要集成现有的服务器。这样在某种程度上已经确定了客户机和服务器的分离。因为客户机也可以充当服务器，反过来也是如此——它们的角色并没有预先定义而且运行期间可以改变。

327

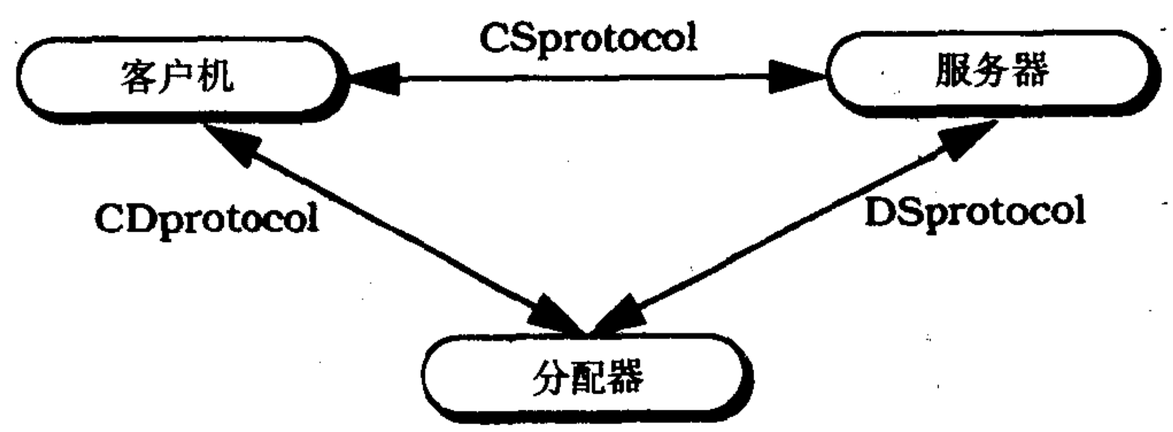
(2) 确定需要哪些通信设备。为客户机与分配器之间、服务器与分配器之间、客户机与服务器之间的交互选择通信设备。每种连接都可以使用不同的通信机制，也可以为三种连接选择同一种通信机制。使用单一的通信设备可以降低实现的复杂度。然而，有时这种方式是不可能或是不可行的。有可能是因为性能的问题。例如，如果处于同一台机器上的分配器和客户机访问它，那么共享存储器是进程间通信的最快方法。在这个实例中，客户机可以通过使用共享存储器同分配器进行通信，但是服务器与分配器，以及客户机与服务器可以使用套接字进行通信。服务器可以分布在不同的机器上，使用套接字是客户机与服务器通信的最好选择。

在那里，现有的服务器不得不被集成到应用程序中，选择适当的通信设备就要由这些服务器已经使用的机制来决定。

如果所有的组件都位于同一个地址空间内，组件间的交互只能依赖于常规的过程调用接口。

328

(3) 指定组件间的交互协议。考虑下面的图：



协议指定了行为的有序序列，这些行为用来对两个组件间通信信道以及被传送消息或数据的结构进行初始化及维护。客户机-分配器-服务器模式包含三种不同的协议。

我们需要服务器和分配器间的交互协议*DSprotocol*。协议针对两个主题：指定服务器如何向分配器注册，以及确定建立与服务器连接的通信信道所必需的行为。

客户机与分配器间的交互协议*CDprotocol*定义了当客户机请求分配器建立同特殊服务器的连接时所发生的交互。如果由于网络或服务器的问题而致使通信建立失效，分配器会通知客户机发生的故障。在报告错误之前，分配器会再试几次试图建立通信链接。

*CSprotocol*指定客户机与服务器如何进行相互交流。这个交互过程包括如下几个步骤：

- 客户机使用先前在客户机与服务器之间建立的通信信道给服务器发送消息。要实现这一点，客户机和服务器需要共享有关发送和接收消息的语义及语法的公共知识。
- 服务器接收消息，对其进行解释并调用其中的某个服务。完成服务后，服务器给客户机发送一个返回消息。

• 客户机从消息中抽取出服务结果并继续完成它的任务。

(4) *确定如何命名服务器*。四字节的因特网IP地址方案是不合适的，因为它没有提供位置透明性。如果使用IP地址，客户机就会依赖于服务器的具体位置。需要引入惟一确定服务器但又不带有位置信息的名字。例如，使用字符串，如“ServerX”，或预先定义的常量，如ID_SERVER_X。这些独立于位置的名字由分配器映射成为物理位置(见步骤5)。

329

(5) *设计和实现分配器*。使用可获得的通信设备确定应该如何实现步骤3中引入的协议。例如，如果分配器位于客户机的地址空间内，局部过程调用应该用于*CDprotocol*。对于其他情况或协议，就要使用诸如TCP端口或共享存储器这样的设备。

借助某些通信机制，可供利用的通信信道也许是受限的资源。例如，套接字描述符数量就要受到操作系统中描述符表规模的限制。关于这点有几种方法。例如，每个服务器可以分配自己的套接字，限制可能存在的服务器数量。客户机请求到达时，分配器将服务器套接字描述符返回给客户机。或者，分配器可以暂时将客户机请求存储在内部消息队列中。然后提供一个套接字端口，此处服务器可能询问是否有新请求到达。当服务请求到达时，服务器打开一个新的套接字并将新的套接字描述符传递给分配器。接着分配器将信息转发给客户机。客户机与服务器间的交互完成以后，服务器关闭其套接字描述符。

基于为服务器选定的通信机制和使用的鉴别方案，定义请求、响应和出错消息的详细结构。

一个分配器包括一个用来将服务器名字映射成为物理位置的库。服务器位置的表示依赖于用来实现客户机-服务器通信的下层机制。例如，物理位置可以用套接字端口、TCP端口、共享存储器句柄或其他一些合适的方案来描述。

需要考虑性能问题。多个客户机通过一个分配器访问多个服务器时，分配器显然构成一个瓶颈。如果想要改善响应和执行时间，就使用多线程。例如，可以在分配器中提供一个线程池。一个请求到达时，将一个线程同该请求相关联，可以并行地处理多个请求。

330

(6) *根据你期望的解决方案和你做出的有关分配器接口的决策来实现客户机与服务器组件*。配置系统，向分配器注册服务器，或者让服务器自身动态地注册和注销。采取与步骤5相同的策略来优化性能。

8. 已解决的例子

在科技信息示例ACHILLES中，一个TCP端口数量以及主机的因特网地址被合并用来惟一地指定一个服务器。客户机连接到分配器并通过使用诸如“NASA/HUBBLE_TELESCOPE”这样的标识符来询问服务器位置。系统预先定义所有消息的结构：带有固定长度的消息标头随后带有一组随机的原始数据。所有用来解释原始数据的必要信息（如其长短或格式）都包括在消息标头中。每个标头也包含消息的发送者和接收者。消息用顺序号来标记，以便消息接收者将新来的包依照正确顺序重新结合起来。当服务器收到请求时，它将信息从消息中抽取出来，如要调用的服务。例如，在客户机的消息中有可能包括以下的信息：“HUBBLE_DOC_RECEIVE, ANDROMEDA.jpg”。服务器确定被请求的文件是否可用并将包含图片的消息发送给客户机。

9. 变体

分布式分配器。可以通过引入分布式分配器来取代网络环境中的单个分配器组件。在这个

变体中，当分配器接收到远程机器上的客户机对服务器的请求时，在目标节点上建立同分配器的连接。远程分配器启动与被请求服务器的连接并将通信信道发回第一个分配器。然后信道被返回给客户机。另一种可能性是允许客户机在远程机器上直接同分配器通信。但是，这样就限制了位置透明性，因为客户机必须要知道它们想要访问的每个服务器的网络节点。在使用分布式分配器变体前，考虑代理者体系结构模式的使用。

331

由客户机管理客户机-分配器-服务器间的通信。在这个变体中，分配器只给客户机返回服务器的物理位置而不是建立到服务器间的通信信道。然后由客户机负责管理与服务器的所有通信活动。可以使用这个变体去提高总体性能，或者因为可获得的通信设备并不要求建立一个明确的通信链路。

采用异构通信的客户机-分配器-服务器。在客户机与服务器间仅使用一种通信机制进行通信是不可能的。有些服务器可能使用套接字，而其他一些服务器却使用管道。这导致一个客户机-分配器-服务器模式变体，其中分配器可以支持多种通信机制。在这个变体中，每一个服务器向分配器来注册自身，并指定它支持的通信机制。当客户机请求到一个特定服务器的通信信道时，分配器使用服务器指定的通信设备建立通信。

客户机-分配器-服务。在这个变体中，客户机关心服务而不关心服务器。当分配器接收到请求后，它在其库内查询哪些服务器提供指定的服务，并与其中一个服务供应者建立连接。如果连接失败，它试着去访问另一个提供相同服务的服务器，前提是存在这样一个服务器。

➡下面的Java代码示例描述了客户机-分配器-服务器的变体。所有的客户机、服务器和分配器处于同一地址空间内。

类Dispatcher使用一个向量的哈希表作为名字服务库。哈希表中的一个条目可供每一个服务名字使用。每个条目由提供相同类型服务的服务器向量组成。服务器通过指定服务器名字和新的服务器实例向分配器注册。当客户机向分配器请求一个特定服务时，分配器在其库中查询所有可供使用的服务器。然后随机选取其中一个并将服务器引用返回给客户机。

332

```
import java.util.*;
import java.io.*;

// Exception thrown by the dispatcher:
class NotFound extends Exception {}
class Dispatcher {
    Hashtable registry = new Hashtable();
    Random rnd = new Random(123456); // for random access

    public void register (String svc, Service obj) {
        Vector v = (Vector) registry.get(svc);
        if (v == null) {
            v = new Vector();
            registry.put(svc, v);
        }
        v.addElement(obj);
    }
    public Service locate(String svc) throws NotFound {
        Vector v = (Vector) registry.get(svc);
        if (v == null) throw new NotFound();
        if (v.size() == 0) throw new NotFound();
```

```
        int i = rnd.nextInt() % v.size();
        return (Service) v.elementAt(i);
    }
}
```

抽象类Service表示可用的服务器对象。执行构造函数时，它自动地向分配器注册服务器对象。

```
abstract class Service {
    String nameOfService; // service name
    String nameOfServer; // server name
    public Service(String svc, String srv) {
        nameOfService = svc;
        nameOfServer  = srv;
        CDS.disp.register(nameOfService,this);
    }
    abstract public void service(); // service provided
}
```

从抽象类Service中派生出具体的服务器类。因此，它们必须实现抽象方法service。这些具体类的实例必须要在其自身构造函数内调用基类的构造函数，所以可以自动地对它们注册。 333

```
class PrintService extends Service {
    public PrintService(String svc, String srv) {
        super(svc,srv);
    }
    public void service() { // test output
        System.out.println("Service " + nameOfService
                + " by " + nameOfServer);
        // here the service code would be implemented
    }
}
```

客户机向分配器请求对象引用，然后使用这些引用去调用适当的方法实现。

```
class Client {
    public void doTask()
    {   Service s;
        try { s = CDS.disp.locate("printSvc");
              s.service();
        }
        catch (NotFound n) {
              System.out.println("Not available");
        }
        try { s = CDS.disp.locate("printSvc");
              s.service();
        }
        catch (NotFound n) {
              System.out.println("Not available");
        }
        try { s = CDS.disp.locate("drawSvc");
              s.service();
        }
        catch (NotFound n) {
              System.out.println("Not available");
        }
    }
}
```

类CDS定义应用的主程序。它实例化分配器、某些服务器和一个客户机。然后调用客户机的事件循环体：

```
public class CDS {
    public static Dispatcher disp = new Dispatcher();
    public static void main(String args[]) {
        Service s1 = new PrintService("printSvc","srv1");
        Service s2 = new PrintService("printSvc","srv2");
        Client client = new Client();
        client.doTask();
    }
}
```

334

程序启动时，显示如下输出：

```
Service printSvc by srv2
Service printSvc by srv1
Not available
```

用户开始应用程序时，调用类CDS的静态方法main。两个服务S1和S2使用同样的名字向分配器disp注册。然后通过调用client.doTask()创建和开始客户机。客户机请求分配器两次定位服务“PrintSvc”，以及一次定位服务“DrawSvc”。通过使用一个随机数发生器，分配器返回用一个特定名字注册的服务对象。因此客户机的第一次服务调用在样本输出中是指不同的服务对象。由于服务“DrawSvc”不可用，当客户机请求分配器去定位合适的服务器时就会发生错误。

10. 已知应用

Sun公司**远程过程调用**（Remote Procedure Calls, RPC）[Sun90]的实现是基于客户机-分配器-服务器设计模式的原理。它实现了*分布式分配器变体以及由客户机管理的客户机-分配器-服务器间的通信变体*的结合。端口映像程序进程担当分配器的角色。一个启动RPC的进程随后成为客户机，而接收进程成为服务器。当客户机进程调用远程过程时，它连接到目标机上的端口映像程序进程。这是可能的，因为所有的端口映像程序都使用同样的TCP/UDP端口来接收请求。端口映像程序向客户机返回请求服务的TCP/UDP端口，接着客户机端口与远程服务器建立一个直接的信道。

OMG Corba(公用对象请求代理体系结构，Common Object Request Broker Architecture)规范[OMG92]使用客户机-分配器-服务器设计模式的原理来细化和实例化代理者体系结构模式。

335

11. 效果

客户机-分配器-服务器设计模式具有如下的**优点**：

服务器的可交换性。在客户机-分配器-服务器设计模式中，软件开发者不需要修改分配器组件或者客户机，就能改变服务器或者增加新的服务器，而这么做往往是必要的。如果一个服务器的新实现是可获得的，则服务器首先注销自己，然后用新的实现重新注册自己。

位置和迁移透明性。客户机不必知道服务器的位置——它们不依赖于任何位置信息。因而，服务器可以动态迁移到其他的机器上。当然，在客户机与其连接时如果迁移服务器就会不成功。

重新配置。直到系统开始运行，或者甚至在运行期间，开发者均能改变有关服务器应该运行在哪个网络节点上的决定。因此客户机-分配器-服务器设计模式为以后将软件系统转变为分布式系统做好准备。

容错性。当网络或服务器发生故障时，一个不同的网络节点上的新服务器被激活，而丝毫不会影响客户机。这使得系统具有更强的健壮性和容错性。

客户机-分配器-服务器设计模式具有如下的**不足**：

由间接方法和显式连接建立而带来的低效率。客户机-分配器-服务器模式的性能依赖于由分配器引入的总开销，包括定位和注册服务器以及显式建立连接等行为带来的开销。另一种方法是取消分配器而将服务器位置强制编码入客户机中。然而这也导致几个缺点。例如，客户机将会直接依靠服务器的位置，因而降低了服务器的可交换性。

对分配器组件接口改变的敏感性。因为分配器起着中心的作用，所以软件系统对于改变分配器接口是很敏感的。 336

参见

可以将转发器-接收器设计模式与客户机-分配器-服务器模式结合起来，以隐藏进程间通信的细节。虽然客户机-分配器-服务器模式可以通过支持位置透明性将客户机与服务器分离开来，但是它没有封装下层通信设备的细节。要实现这一点，可以在服务器与客户机、客户机与分配器以及服务器与分配器之间引入转发器和接收器。

接受器（*Acceptor*）与连接器（*Connector*）模式[Sch96b]阐述了另一种从连接处理中分离出连接结构的方法。Schmidt的模式比我们的方法更为分散，它使用了一个集中式分配器。Schmidt的模式中每个被动接收连接的地点都可以提供接受器工厂族。这些接受器负责构造服务处理程序，它们是指向指定应用服务的入口点。

可以定义不同的接受器去辨别不同的连接策略，如同步与异步，并且使用不同的服务策略，如在分离的进程或线程中并发运行或者在单一进程中反应性地多路分解。连接器模式比接受器模式具有“双倍”的功效——根据地点主动地启动连接装置。客户机-分配器-服务器模式类似于一个微型代理者，它带有可以实现动态重定位服务器的名字服务。

致谢

感谢所有参与PLoP'95作者专题讨论会[PLoP95]的人员，他们提供了有价值的建议和评论。 337

3.6.3 出版者-订阅者

出版者-订阅者（*Publisher-Subscriber*）设计模式有助于保持合作组件状态的同步化。为达到这一点，它能实现单向更改传播：一个出版者通知任意数目的订阅者有关对于其状态的更改。

1. 别名

观察者，依赖

本节我们给出基于[GHJV95]中的观察者模式的简短的模式描述，使得我们可以给出附加的观点和变体。

2. 问题

问题通常出现在某处有数据改变的情况下，而许多其他组件依赖于这个数据。经典的实例是用户接口元素：当某个内部数据元素改变时，所有依赖于这个数据的视图都必须被更新。我们可以通过引入直接调用传播更改的依赖来解决这个问题，但是这个解决方案不易改变而且不可重用。我们要寻找一种更为通用的可以适用于不同环境的变更-传播机制。

解决方案要平衡如下强制条件：

- 必须将在特定组件中状态的改变通知一个或更多组件。
- 先前并不知道相关组件的数目及身份，甚至随着时间的推移这些也可以改变。
- 由从属物公开轮询获取新信息是不可行的。
- 当引入变更-传播机制时，信息的出版者及其从属物不能被紧密地耦合在一起。

3. 解决方案

一个专用的组件代替出版者([GHJV95]中称为目标（*subject*））。所有依赖于出版者中更改的组件都是它的订阅者([GHJV95]中称为观察者（*observer*））。

339 出版者维持当前订阅的组件登记。只要一个组件想要成为一个订阅者，它使用由出版者提供的订阅接口。类似地，它也可以取消订阅。

只要出版者改变状态，它会给它的所有订阅者发送通知。订阅者依次自行取回已更改的数据。

在模式的实现中，模式提供如下的自由度：

- 如[GHJV95]中描述的那样，可以引入抽象基类去使不同的类成为出版者或订阅者。
- 出版者可以决定哪个内部状态改变并会通知其观察者有关情况。它也可以在调用`notify()`之前将几个更改排队。
- 一个对象可以成为多个出版者的一个订阅者。
- 一个对象可以同时担任出版者和订阅者的角色。
- 可以根据事件类型区分订阅者和随后的通知。这使得订阅者只得到关于它们感兴趣事件的消息。
- 当出版者通知其订阅者时，它可以发送选定的数据更改的细节，或仅仅发送一个通知并授权订阅者由其自身选择更改的内容。

更概括地说，我们区分压入与拉-模式。在推-模式（*push model*）中，当出版者通知其订阅者时，它发送所有更改的数据。订阅者无权选择是否和何时取回数据——它们仅仅只是接收。在拉-模式（*pull model*）中，当出版者发送更改通知时仅仅发送最少的信息——订阅者负责取回所需的数据。有些变化有可能处于这两个极端之间的中间带。

推-模式有非常稳固的动态特性，而拉-模式提供了更多的灵活性，但是这是以出版者与订阅者之间较多的消息数为代价的。

对于复杂的数据改变，推-模式是一个不好的选择，尤其是当出版者给订阅者发送一个大的
340 但又不是订阅者感兴趣的包时。即使仅将描述数据改变的特性压入一个包中，带来的开销也是很大的。此时，使用拉-模式并让订阅者查找发生了什么样的数据改变。连续地查找有关数据变更细节的过程可以像决策树那样来组织。

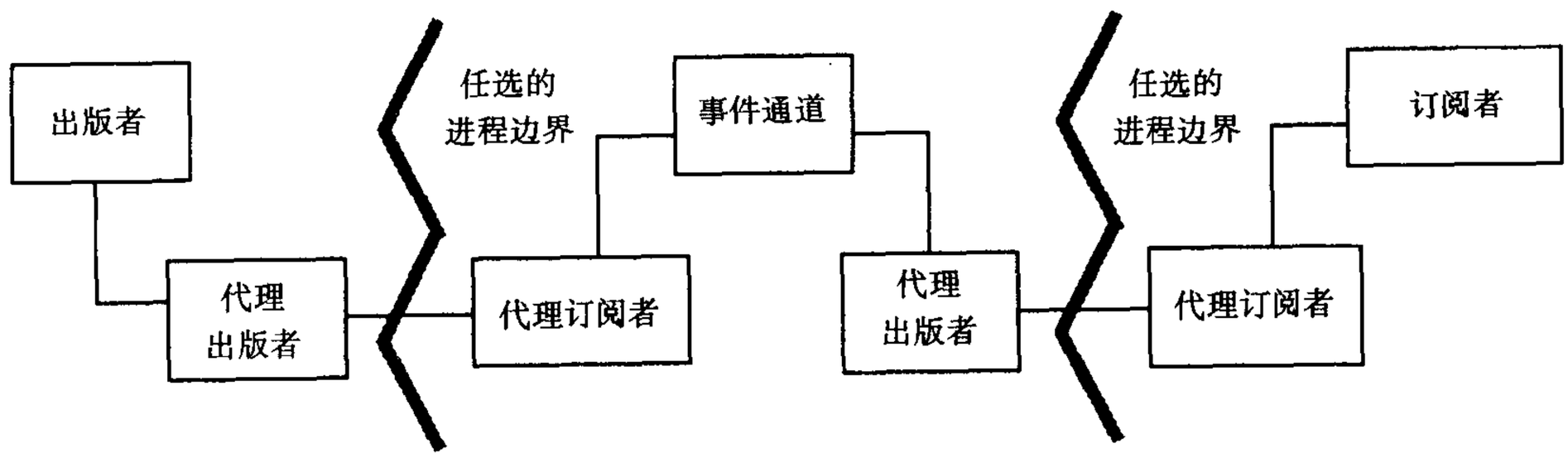
通常，当多数情况下订阅者需要出版的信息时，推-模式是一个较好的选择。只有当单个订阅者可以决定是否和何时需要一段特定信息时才使用拉-模式。

4. 变体

值班员（*Gatekeeper*）。出版者-订阅者模式同样也可以运用到分布式系统中。在这个变体中，进程中的一个出版者实例通知远程订阅者。出版者可以交替地在两个进程中扩展。进程中的一个组件发送消息，而在接收进程中一个单件“值班员”通过勘察指向进程的入口点来多路分解它们。值班员通知事件处理订阅者它们注册的事件何时发生。反应器模式[Sch94]详细地描述了这个方案。

事件通道（*Event Channel*）变体由OMG在其事件服务规格说明[OMG95]中提出并且以分布式系统为目标。这个模式严格地分离出版者和订阅者。例如，可以存在多于一个出版者，并且订阅者仅希望得到有关更改事件的通知，而不希望知道出版者的身份——订阅者不关心到底是哪个组件的数据被更改了。同样地，出版者也不关心哪个组件正在订阅。

在这个变体中，创建事件通道并将其置于出版者和订阅者之间。对于出版者而言，事件通道作为订阅者出现，而对于订阅者而言，事件通道则作为出版者出现。一个订阅者向事件通道注册，如下图所示。它请求一个管理实例去创建一个“代理出版者”，并跨越进程边界同本地的“代理订阅者”连接。类似地，在出版者和事件通道间创建一个“代理出版者”，并且在事件通道的另一边创建一个“代理出版者”。



按照这种方式，出版者、事件通道以及订阅者可以存在于不同的进程中。给事件通道提供一个缓冲区去进一步分离出版者和订阅者。当来自出版者的消息到达时，事件通道不需要立即通知订阅者，但是可以实现其自身的通知策略。 341

甚至可以链接几个事件通道。这样做的原因在于事件通道可以提供附加性能，例如过滤事件或固定的时间内在内部存储事件并将其发送给在那段时间内订阅的所有组件。这通常被称为“服务质量”。于是，一条链可以汇集一个系统需要的性能——这条链总结了由单个事件通道组成的性能，这类似于UNIX的管道。

事件通道变体有足够的能力包容多个出版者以及分类的事件。

出版者-订阅者模式的另一种变体使用生产者-消费者合作风格。其中生产者提供信息，而消费者接收这个信息用做进一步处理。通常，通过在生产者与消费者间放置一个缓冲区来分离它们。生产者向缓冲区写入数据而不需要考虑消费者。而消费者随意地从缓冲区中读出数据。

惟一的同步是实现检查缓冲区上溢或下溢。当缓冲区满时生产者暂停，如果由于缓冲区为空而无法读出数据时消费者会等待。出版者-订阅者模式与生产者-消费者变体间的另一个不同在于后一种变体中通常生产者与消费者保持1:1的关系。

342 只有更为复杂的模式（如事件通道）可以模拟多于一个生产者或消费者的生产者-消费者关系。只要少数的生产者可以直接或是间接地通过允许连续写入缓冲区来提供数据。多个消费者的情况要稍微复杂一些。当一个消费者从缓冲区内读取数据时，事件通道并没有将数据从缓冲区内删除，只是让消费者从中读出数据。让这个消费者误以为数据被使用过，然后删除了，而其他的消费者则会认为数据仍然存在而且并未被读取过。迭代器是实现这个特性的较好方法。每个消费者在缓冲区内都有自己的迭代器。缓冲区上迭代器的位置反映出相应的消费者已经读
343 取缓冲区到了什么程度。当所有的读取都结束后，缓冲区的数据会在延迟的迭代器之后被清除。

第4章
惯 用 法

"一个什么?" 他问。
"一个S.E.P."
"S...?"
"… E.P."
"那是什么?"
Douglas Adams,《生活、宇宙和万事万物》

惯用法是特定程序设计语言中的低层模式。惯用法描述如何用给定语言的特征来实现组件的特定方面或组件之间的关系。

在本章中，我们给出使用惯用法的一个概观，说明惯用法如何定义一种程序设计风格以及从哪里可以找到这些惯用法。我们主要提出他人的工作而不是记录我们自己的惯用法。我们给出计数指针（*Counted Pointer*）惯用法作为一个完整的惯用法例子进行描述。 345

4.1 引言

惯用法表示低层模式。与关注一般结构原理的设计模式相比，惯用法描述如何在一种程序设计语言中解决具体实现问题，如C++中的内存管理问题。惯用法可能也直接针对一种特殊设计模式的具体实现。因此，我们不能在设计模式和惯用法之间划出一条清晰的界限。惯用法可以针对与语言的使用相关的低层问题，如命名程序的元素、源文本格式化或选择返回值。这些惯用法与那些由程序设计指南特别关注的领域比较接近或部分重叠。总而言之，可以说惯用法展示了程序设计语言特征的应有的使用，因此，惯用法有助于对程序设计语言的教学。

一种程序设计风格可以由用于实现一种解决方案的语言构造方式来刻画，例如：使用的循环语句类型、程序元素的命名、甚至源代码的格式。每当实现决策导致一种特定的程序设计风格的时候，彼此分离的任何一个方面都能够形成一种惯用法。这些相关惯用法的汇集定义了一种程序设计风格。

就像软件体系结构的所有模式一样，惯用法使开发人员之间的交流变得更容易，而且加速了软件的开发和维护。项目组的惯用法汇集就形成了公司的智力财富。

4.2 惯用法能够提供什么

学习一种新的程序设计语言并不因为掌握了它的语法而终止。用一种给定的语言解决一个

特定的程序设计问题，总存在许多方法。某些方法具有比较好的风格或更好地利用了语言优点。必须了解并掌握那些能提高效率和代码质量的小技巧以及难以言表的规则。

346

一个惯用法有助于用常用的程序设计语言解决经常遇到的问题，如内存管理、对象创建、方法命名、为了易读性而进行的源代码格式化、特定库构件的有效使用等。

在解决这些问题当中，有几种获取专门知识的方法。一种方法是阅读由经验丰富的程序员编写的程序。这迫使你去思考他们的风格，并促使你在自己的代码中模仿这种风格。由于理解他人的代码并不总是一件容易的事，因此这种方法需要花费很长的时间。如果现有一组惯用法可以供你学习，那么使用一种新的程序设计语言进行高效编程就会容易得多，因为惯用法能教会你如何利用语言的特点高效地解决特定的问题。

因为每个惯用法都有惟一的名字，所以它们为软件开发人员之间的交流提供了一种工具。一组在一起工作了很长时间的有经验的软件开发工程师可以用他们自己的惯用法分享经验。对于一个新加入这个组的人员来说，理解并掌握这些隐含的惯用法将困难重重。因此明确惯用法和它们的使用是一个好主意，比如，尽力对使用的惯用法进行命名和归档。

相对于许多设计模式而言，惯用法在程序设计语言之间的“可移植性”比较差。例如，Smalltalk汇集类的设计中结合了许多专门针对这门语言的惯用法。它们依赖于在C++中没有出现的一些特点，比如无用单元收集或元信息。一个早期的C++类库——NIHCL[GOP90]，通过模仿Smalltalk的汇集类来实现C++程序的汇集类。例如，每个有对象存储在汇集中的类必须由NIHCL根类`Object`继承得到。另外，由于存储管理完全依靠程序设计人员，从而导致使用NIHCL的汇集类比使用Smalltalk的汇集类要困难得多。现在的C++类库（如Generic++[SNI94]）舍弃了这种方法，并通过使用C++模板机制实现不同于NIHCL的汇集类。这样的模板汇集类可以存储任何给定类型的数据，甚至非对象类型数据。

347

4.3 惯用法与风格

就像其他专家一样，有经验的程序设计人员在工作时会使用模式。由单独一个程序设计人员编写的好程序将包含他的模式集的许多应用。理解程序设计人员所使用的模式会使理解他们的程序变得容易许多。

但是坚持一贯的风格非常困难，即使是对有经验的程序设计人员而言也是如此。如果小组中的程序设计人员使用不同的风格，那么他们应该对他们的程序商定一种单一的代码风格。例如，考虑下面的C/C++代码段，这两段代码都实现了一个用于“C风格”字符串的字符串拷贝函数：

```
void strcopyKR(char *d, const char *s) {
    while (*d++=*s++);
}

void strcopyPascal(char d[], const char s[]){
    int i ;
    for (i = 0; s[i] != '\0'; i = i + 1)
    {
```

```
        d[i] = s[i];
    }
    d[i] = '\0'; /* always assign 0 character */
} /* END of strcopyPascal */
```

这两个函数计算得到相同的结果——它们将字符串s中的字符拷贝到字符串d中，直到一个字符的值为0。一个编译器甚至可以对两个例子都生成相同的经过优化的机器代码。在Kerninghan和Ritchie[KR88]的传统简洁的C风格中，strcopyKR()函数使用指针作为数组参数的别名。strcopyPascal()函数可以由具有某种语言背景（如Pascal语言）的程序设计人员来编写，其中想以链式数据结构来使用指针。两种实现有它们各自的风格。喜欢哪种风格，或自己的版本看起来像哪一种风格，依赖于自身的经验、背景、喜好和许多其他因素。如果一个程序同时具备这两种风格，那么对它的理解和维护要比仅包含单一风格的程序难得多。这是理解程序风格的先决条件，如在strcopyKR()中看起来很奇异的while循环。 348

团体风格指南是小组在开发整个程序过程中形成一致风格的一种途径。不幸的是，他们许多人使用各自的规则，如“所有的注释必须另起一行”。这就意味着它们不在模式表中——它们给出了解决方案或规则，但没有对这一问题进行陈述。这种风格的另一个缺点是它们很少就如何解决经常发生的编码问题给程序员以具体的建议。

我们认为包含了众多惯用法的风格指南作用更大。它们不仅提供规则，而且还给出了依据这些规则解决问题的见解。他们对惯用法进行命名，并因此可以彼此交流。例如，说出并记住“你应该在这里使用目的展示选择器”[Bec97]要比“应用规则§7-42并因此改变你的方法名”容易得多。然而这种风格指南还不多。进一步的问题是，如果不小心将来自不一致的风格的惯用法应用到一个程序当中，那么这些相互冲突的惯用法就不能很好地融合在一起。

这里有一个风格指南惯用法的例子，它出自Kent Beck所著的《*Smalltalk Best Practice Patterns*》[Bec97]：

1. 名称

缩排控制流（Indented Control Flow）

2. 问题

如何缩排消息？

3. 解决方案

将0个或1个自变量消息放在与其接收器收到时同样的一些行上。

```
foo isNil
2 + 3
a < b ifTrue:[...]
```

如果每个消息含有至少两个关键字，则将消息的关键字/自变量对放到各自的行上，并缩排一个制表键位。

```
a < b
    ifTrue:[...]
    ifFalse:[...]
```

□

不同的惯用法集适用于不同的领域。例如，可以用继承和动态绑定的面向对象风格编写C++程序。在某些领域，比如实时系统中，需要一种更“有效”的不使用动态绑定的风格。因此，对于大公司而言，它们拥有众多从事不同领域应用程序开发的开发小组，单一风格指南肯定是不合适的。一种风格指南不能也不应该涵盖所有风格。

349

一组相关的惯用法会使我们在程序中形成一种一致的风格。这种单一风格将加速开发进程，因为不必花费大量的时间去考虑那些包含在你的惯用法中的简单问题，例如如何格式化一段代码。此外，一致的风格对程序的开发和维护也有所帮助，因为它使程序设计更容易理解。

4.4 在哪里可以发现惯用法

一种程序设计语言的程序设计风格不在本书的论述范围之内，因为这样的风格和惯用法本身就可以写整整一本书。我们建议在学习一本好的语言书时，应该仔细阅读导言部分，以便从收集一组要使用的惯用法开始学习这门语言。作为练习，在对自己的模式编制文档时，可以重新描述这些书中所给出的准则以符合一个模式模板。这将有助于理解什么时候应用这些准则，从而可以很容易地确定一个准则解决什么问题。

某些设计模式以更普遍的方式指出了程序设计问题，它们也可以作为惯用法的来源。如果从一种特定语言的程序设计角度来看这些模式，会看到一些嵌入的惯用法。例如，单件设计模式[GHJV95]给出了两种不同于Smalltalk和C++的惯用法：

名称

单件（Singleton）（C++）

问题

用C++实现单件设计模式[GHJV95]，确保运行期间恰好存在一个类实例。

解决方案

350

使类的构造函数成为私有的。声明一个静态成员变量`theInstance`代表那个惟一存在的类实例。在类的实现文件中将该指针初始化为零。定义一个公有的静态成员函数`getInstance()`，该函数用于返回`theInstance`的值。当第一次调用`getInstance()`时，它将用`new`创建惟一的实例并将它的地址赋给`theInstance`。

例子

```
class Singleton {
static Singleton *theInstance;
Singleton();
public:
static Singleton *getInstance() {
        if (! theInstance)
            theInstance = new Singleton;
        return theInstance;
    }
};
//...
Singleton* Singleton::theInstance = 0;
```

与之相应的单件的Smalltalk版本解决相同的问题，但是解决方案不同，因为Smalltalk的语言概念与C++语言完全不同。Smalltalk的单件设计模式如下：

名称

单件（Singleton）（Smalltalk）

问题

用Smalltalk实现单件设计模式[GHJV95]，确保运行期间恰好存在一个类实例。

解决方案

覆盖类方法new会产生一个错误。添加一个类变量TheInstance保存那个惟一的实例。实现类方法getInstance，该方法用于返回TheInstance的值。当第一次调用getInstance时，它将用super new创建惟一的实例并将它赋给TheInstance。

例子

```
new
    self error: 'cannot create new object'

getInstance
    TheInstance isNil ifTrue: [TheInstance := super new].
    ^ TheInstance
```

351

能够形成几种不同的C++编码风格的惯用法在多本专著中都有论述，例如Coplien的《*Advanced C++*》[Cope92]、Barton和Neckman的《*Scientific and Engineering C++*》[BN94]和Meyers的《*Effective C++*》[Mey92]。

从"*Smalltalk Report*"的Kent Beck专栏所给出的惯用法中，可以看到Smalltalk程序设计知识的一个好的汇集。他的论文汇编《*Smalltalk Best Practice Patterns*》以书的形式出版。Beck采用与Smalltalk类库一致的编码模式定义了一种程序设计风格，所以可以把这个模式汇编视为Smalltalk风格指南。他的许多模式相互依存，所以除了作为一种风格指南之外，他的汇编也被看成是一种模式语言。

还可以看自己的程序代码或同事的代码，阅读代码并筛选出曾使用过的模式。可以用这样的"模式挖掘"来建立一个你自己的程序设计语言的风格指南，并进而成为你们小组的智力财富。对每一个惯用法命名，其风格指南就为开发人员之间彼此交流提供一种语言。它还能为新加入小组的开发人员提供一种教学辅助。

352

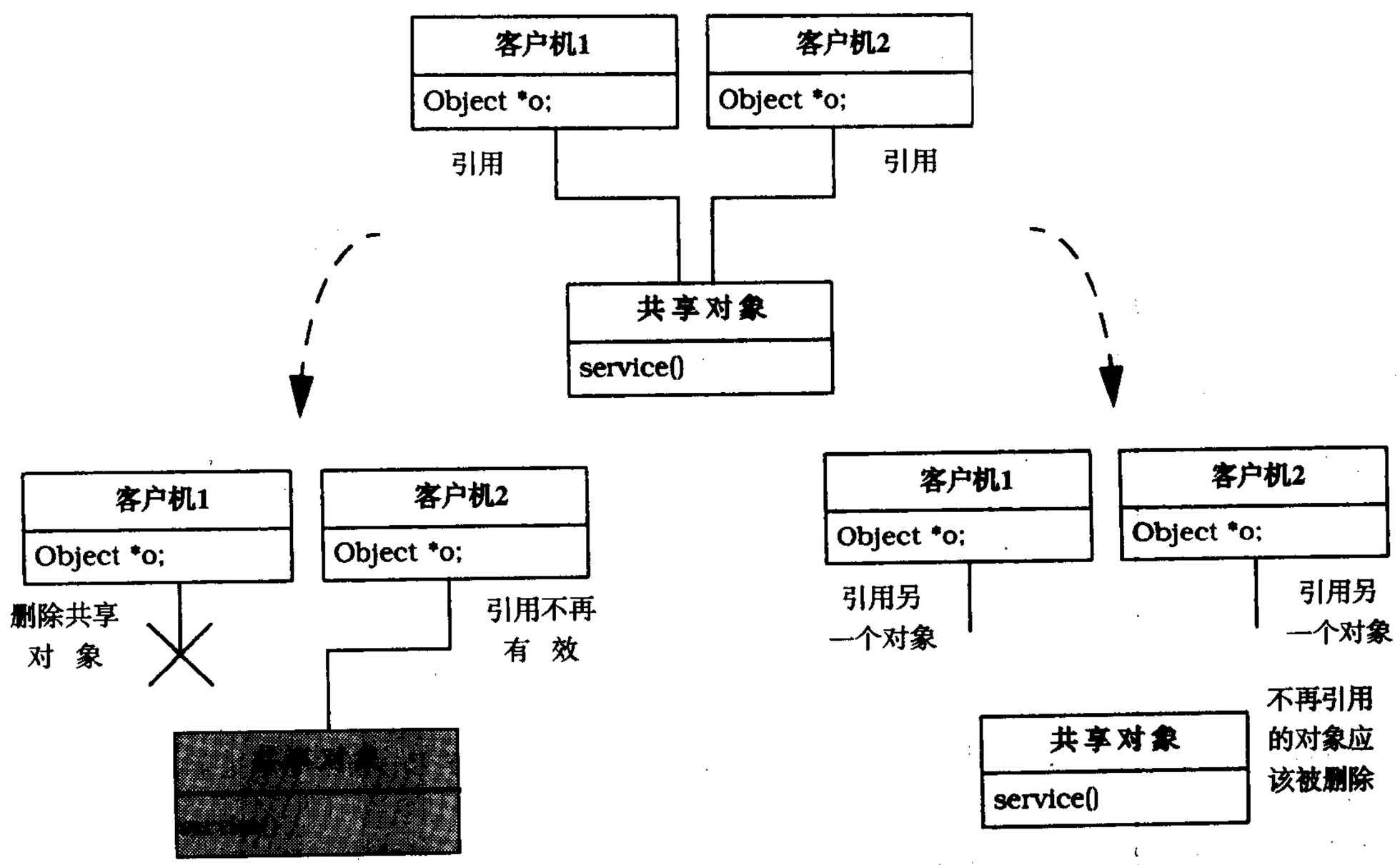
4.4.1 计数指针

计数指针（*Counted Pointer*）惯用法[Cope92]使C++中的动态分配共享对象的内存管理更为容易。它引入了实体类的一个引用计数器，其中的计数器由句柄对象更新。客户机只有凭借句柄通过重载operator->()来访问实体类对象。

1. 例子

在使用C++进行面向对象的开发时，内存管理是一个很重要的课题。一旦一个对象由客户机

共享，其中的每个客户机都拥有一个对该对象的引用。此时存在两种可能引发问题的情况：一个客户机可能删除该对象，同时另一客户机仍持有该对象的引用；或者所有的客户机都可能“忘记了”它们的引用，但没有将该对象删除。



2. 语境

对动态分配类实例的内存管理。

3. 问题

在任何面向对象的C++程序中都必须将对象作为函数的参数来传递。采用指针或对对象的引用作为参数是一种典型用法。这样就可以利用多态机制。然而，自由地四处传递对象的引用可能导致上图所示的情况——不知道引用是否仍然有效或是否仍然需要。

一种避免因使用了指针和引用而引起问题的办法就是完全不使用它们，并代之以值来传递对象，正如通常使用整数那样。C++允许创建这样的程序，而且编译器将自动销毁超出其使用范围的值对象。

然而，这种解决方案并非对所有的程序都有效，原因有三点。首先，如果通过值传递的对象非常大，那么对它们的每次拷贝在运行期间和内存消耗方面的代价也是非常昂贵的。其次，也许需要创建动态对象结构（如树或有向图），而这在C++中仅仅使用值对象几乎是不可能的。最后，可能需要有计划地共享一个对象，例如在几个汇集中同时存储它。

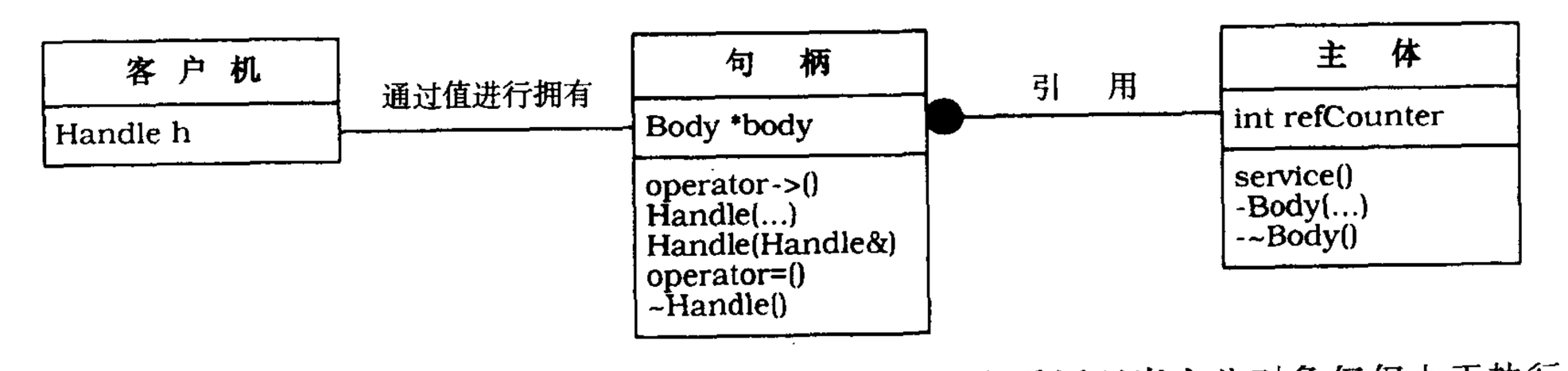
如果必须处理指向类的动态分配对象的引用或指针，那么需要注意以下强制条件：

- 对一个类而言，通过值传递对象是不合适的。

- 几个客户机也许需要共享同一对象。
- 要避免“悬挂”引用（即引用一个已被删除的对象）。
- 如果已不再需要一个可共享对象，它就应被销毁以便有效利用内存并释放其他被占用的资源。
- 解决方案应该不需要过多的有关每个客户机的附加代码。

4. 解决方案

计数指针惯用法通过引入引用计数方便了对共享对象的内存管理。通过引用计数器来扩展共享对象的类，被称为主体（*Body*）。为了跟踪所使用的引用，第二个类句柄（*Handle*）是惟一允许拥有对主体对象引用的类。程序中所有句柄对象都通过值进行传递，并因此被自动地分配和销毁。句柄类维护主体对象的引用计数器。通过重载句柄类中的`operator->()`，可以按照语法使用其对象，就好像它们是指向主体对象的指针。 354



本求解方案的一种变化形式参见下面“变体”一节，它适用于当主体对象仅仅由于执行原因才被共享的情况。

5. 实现

为了实现计数指针惯用法，按如下步骤执行：

(1) 声明主体类的构造函数和析构函数为私有的（或受保护的）以阻止对它的不受抑制的实例化和删除。

(2) 声明句柄类为主体类的友元，并因此提供对主体内部访问的句柄类。

(3) 用一个引用计数器扩展主体类。

(4) 向句柄类中增加一个指向主体对象的数据成员。

(5) 通过拷贝主体对象指针和增加共享主体对象的引用计数器，实现句柄类的拷贝构造函数及其赋值操作符。实现句柄类的析构函数，以减小引用计数器以及在计数器为0时删除主体对象。

(6) 实现句柄类的箭头操作符如下：

```
Body  * operator->() const { return  body ; }
```

355

并且声明它为公有成员函数。

(7) 用一个或几个构造函数扩展句柄类，其中这些构造函数用于创建它所引用的初始化的主体实例。每个这样的构造函数都初始化引用计数器为1。

6. 样本代码

在下面的C++代码中使用计数指针惯用法：

```
class Body {
public:
// methods providing the bodies functionality to the world
     void service() ;
     // further functionality...
private:
     friend class Handle;
     // parameters of constructor as required
     Body(/*...*/) { /* ... */ }
     ~Body() { /* ... */ }
     int refCounter;
};

class Handle {
public:
     // use Body's constructor parameters
     Handle(/*...*/) {
          body = new Body(/*...*/);
          body->refCounter = 1;
     }
     Handle(const Handle &h) {
          body = h.body;
          body->refCounter++;
     }
     Handle & operator=(const Handle &h) {
          h.body->refCounter++;
          if (--body->refCounter) <= 0)
               delete body;
          body= h.body;
          return *this;
     }
     ~Handle() {
          if (--body->refCounter <= 0)
               delete body;
     }
     Body* operator->() { return body; }
private:
     Body *body;
};
// example use of handles ...
Handle h(/* some parameter */);
// create a handle and also a new body instance
{    Handle g(h); // create just a new handle
     h->service(); g->service();
} // g goes out of scope and is automatically deleted

h->service(); // still possible
// after h goes out of scope the body instance is
// automatically deleted.
```

356

7. 变体

引用计数的常见应用类似于计数指针，被用于较大的主体对象的性能改进。这种变体在[Cope92]中称为引用计数惯用法（*Reference Counting Idiom*），在[Cope94a]中称为计数主体（*Counted Body*）。在这种变体中，一个客户机可能会误认为使用的是它自己的主体对象，即便它

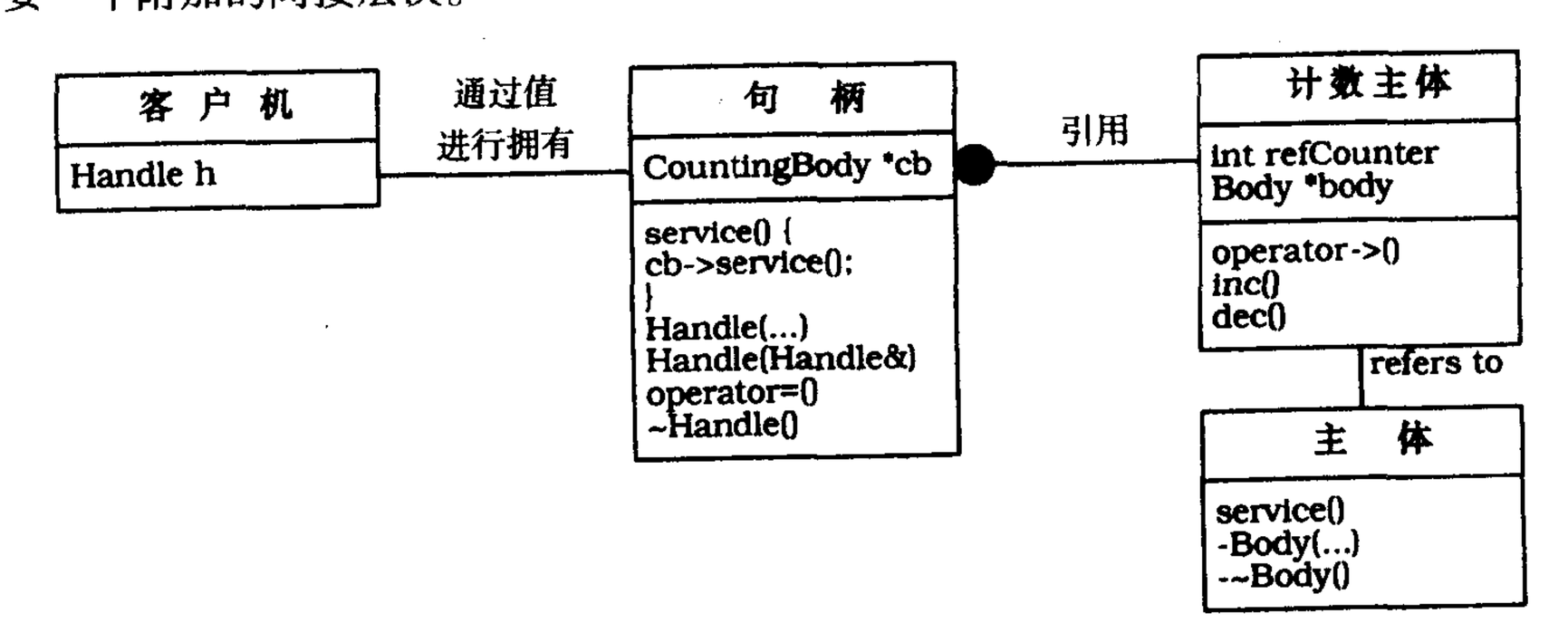
正与其他客户机共享该对象。每当一个操作可能改变共享主体对象的时候，句柄创建一个新的主体实例并在以后的处理中使用这个拷贝。为了实现这种功能，仅仅重载operator->()是不够的。相反，句柄类复制了主体类的接口。句柄类中的每种方法都代表着对它所引用的主体实例的执行。如果还有其他客户机共享该主体对象，那么改变主体对象的方法就会创建该对象的一个新拷贝。

参见

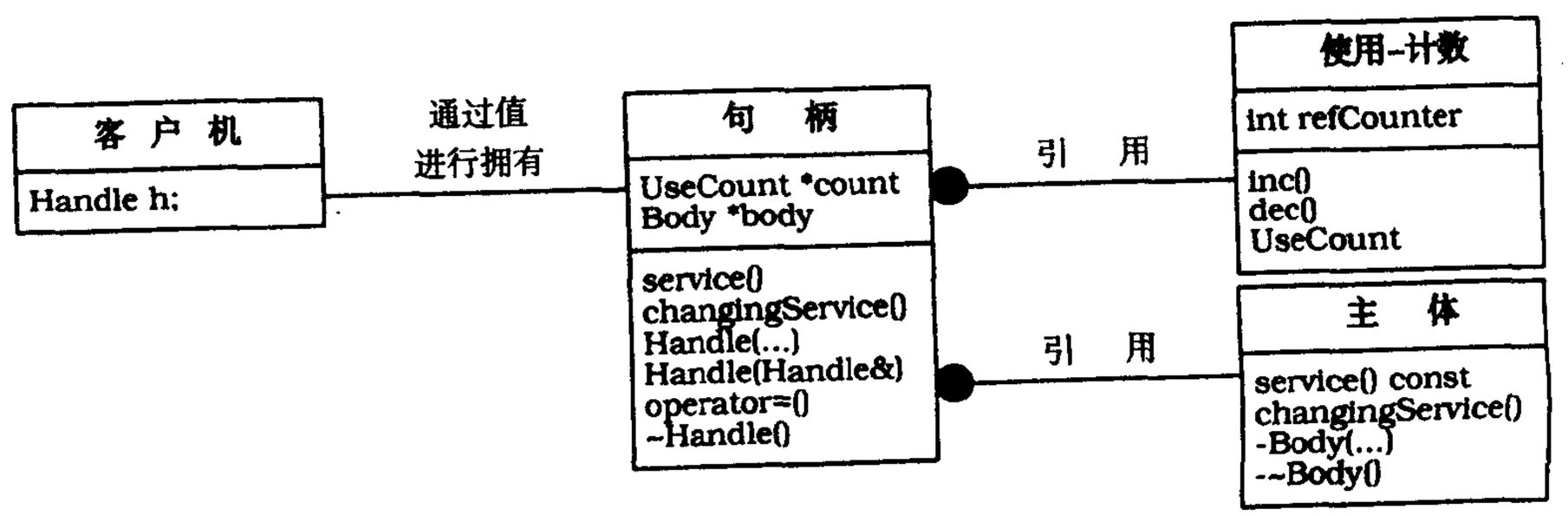
Bjarne Stroustrup [Str91]讨论了几种句柄类的扩展方式。如果主体类作为一个模板参数传递，并与句柄模板类协作，例如主体类为句柄类提供对引用计数器的访问，则句柄可以作为一个模板来实现。

由计数指针惯用法提供的解决方案有一个缺点，那就是为了引入引用计数器必须改变主体类。Coplien和Koenig给出了两种方法以避免这一改变。

James Coplien [Cope92]提出计数指针惯用法及其几种变体。例如，主体类不期望有导出类，可以将它嵌在句柄类中。另一种变体是用一个引用计数器类来包装现存的类，如下图所示。于是这个包装器类就形成了计数指针惯用法的主体类。当客户机访问主体类对象时，这个解决方案还需要一个附加的间接层次。 357



Andrew Koenig给出该主题的又一个变体，它允许向类中添加引用计数而不需要改变它们[Koe95]。他为使用计数定义了一个独立的抽象概念。这样句柄就有两个指针：一个指向主体对象，另一个指向使用-计数（use-count）对象。使用-计数类可以用来实现各种主体类的句柄。这种解决方案的句柄对象需要的空间是其他计数指针变体的两倍，但是，他访问如同对主类的变更一样直接。



358

第5章 模式系统

模式不是一座孤岛。

——*Richard Helm*《个人通信》

模式系统（*pattern system*）将一个个单独的模式捆绑在一起。模式系统描述它的组成模式怎样与系统中其他模式相联系，这些模式怎样实现，以及怎样支持用模式进行软件开发。模式系统是表示和构建软件体系结构的有力工具。

本章，我们详述模式系统，包括我们在本书描述的所有模式，并且它对于集成其他模式（诸如文献 [GHJV95]、[PLoP94]和 [PLoP95]中的那些模式）以及你自己的模式是开放的。

359

5.1 什么是模式系统

模式并不是孤立存在的——它们之间相互依赖。无论怎样表示，包含所有模式的像目录那样清晰的列表并不能反映它们之间的多样关系。而实际上，模式应该交织在模式系统中。

模式系统将它的组成模式捆绑在一起。模式系统描述模式怎样相互联系和怎样相互补充。模式系统同样支持模式在软件开发中的有效使用。

Christopher Alexander 用术语“语言”代替“系统”来描述同样的概念（文献[Ale79]第185页）：

> [模式语言] 的要素是模式。有一个模式的结构，它描述模式本身是其他更小模式的模式。还有一些嵌入在模式内的规则，它们描述创建模式的方式，以及有其他模式存在时排列模式的方式。
>
> 但是，这样的话，模式既是要素又是规则，因此规则和要素是难以区别的。模式是要素。并且每一种模式也是一种规则，它描述要素可能的排列方式，这些要素是它们自身或是其他模式。

实际上，模式系统可以与语言进行比较。模式就像语言的词汇，而实现和合成模式的规则构成语言的语法。

我们更倾向于使用术语“模式系统”而不是“模式语言”。模式语言意味着它的组成模式覆盖了一个特定领域的每一个重要方面。模式语言对软件体系结构而言必须是计算完全的：对软件系统实现和构造的每一方面来说，必须至少有一种模式一定适用——必须没有缺口或者空白。这样的模式语言在一些小的和著名的领域内存在。这里有两个例子，一个是Crossing Chasms [PLoP95]将面向对象应用连接到关系数据库，另一个是CHECKS [Cun94]用于信息完整性。但是，我们描述的模式仅仅包括软件体系结构构造的一个特定方面。它们全部不是计算完全的，即使用我们知道的所有其他的相关模式进行扩充也是如此。因为我们描述模式是怎样捆绑在一起的，所以它对我们来说，不仅限于模式的目录，但又远小于模式语言。我们定义“模式系统”

360

如下：

软件体系结构的模式系统是一个软件体系结构模式的汇集，它包括模式在软件开发中实现、组合和实际使用的指南。

软件体系结构模式系统的主要目标是支持高质量的软件系统开发。所谓“高质量”是指系统既实现其功能需求又实现其非功能需求。为了达到这个目标，一个模式系统必须满足以下的需求：

- *应该包括足够的基本模式*。我们需要可以用来详细描述系统基本结构的模式，可以支持我们细化系统的模式，可以帮助我们用具体的程序设计语言来实现软件体系结构的模式。
- *应该统一描述它所有的模式*。描述的形式必须既捕获模式的本质又对其细节进行准确叙述。这种形式必须能进一步支持模式与其他模式进行比较。
- *应该揭示模式间的各种关系*。模式系统必须能够识别什么样的模式是通过模式细化得到的，什么样的模式是通过模式揭示得到的，模式可以和哪些模式相结合以及可以使用什么样的可选模式。
- *应该组织它的组成模式*。用户应该可以很快找到能帮助他们解决具体设计问题的模式，并且他们可以根据不同的模式探索不同的解决方法。
- *应该支持软件系统的构造*。模式系统应该说明怎样应用并实现它的组成模式。 361
- *应该可以进行自我演化*。随着技术的不断发展，模式系统也将逐步发展。现有的模式可能会改变，它们的描述将改进，新的或者缺少的模式将被增加，现有的模式甚至可能“死亡”。

本书中的模式和其他人写的模式已经满足了第一个要求——我们能提供一个大且有用的模式集合。这些模式涵盖模式系统的全部范围，能处理很多软件体系结构中的问题。

我们的模式描述模板也满足模式系统的这一需要(参见第1章“模式”)。它允许我们画出模式的“大框图”，用来详细描述它的具体结构和动态特性以及指导实现所描述的这种模式。对模式系统而言最重要的是，我们的描述模板能够显示模式与其他模式怎样相连，它可被哪些其他模式细化和结合，它可以揭示哪些变体，以及哪些其他模式以不同方式解决相同的问题。

但是，模式系统不仅仅是用模板描述的多个模式的汇集。我们必须为模式定义一个有用的组织图式，并且指导用户选择模式并使用模式来构造软件系统。最后，我们必须确保该模式系统是开放式的，可以自我演化。

5.2 模式分类

模式系统包含的模式越多，理解和使用模式系统就越困难。如果软件开发者必须阅读、分析和仔细理解每个模式才能找到他们所需的模式，则这个模式系统作为一个整体是没有用的，即使它的组成模式非常有用。为了在模式系统中很方便地处理所有模式，将它们分成相关模式组是很有帮助的。一个可以支持使用模式进行软件系统开发的模式分类图式必须具有下列特性：

- *分类图式应该是简单的、易学的*，而不是复杂的、难理解和难使用的。 362

- 分类图式应该包括少而精的分类标准，而不需要根据每个理论上可能的模式属性来组织一个多维的模式空间。
- 每个分类标准应该反映模式的自然属性，例如反映模式处理问题的类型，而不是反映诸如模式是否属于一种模式语言这样的人为标准。
- 分类图式应该给用户提供一个“路标”，帮助他们找到一组可能适用的模式，而不是死板的“抽屉式”的图式，试图支持用户找到一个“正确的”模式。
- 分类图式应该对于新模式的集成是开放的，而不需要对现有分类进行重构（refactoring）。

要保持我们的分类图式简单，它应该基于如下两个分类标准：模式类别（*pattern category*）和问题类别（*problem category*）。

5.2.1 模式类别

在我们的分类图式中，最基本的分类标准是模式类别。我们区分体系结构模式（*architectural pattern*）、设计模式（*design pattern*）和惯用法（*idiom*）（参见第1章“模式”）。所有3个类别都在软件开发过程中与重要的阶段和活动相关：

- 当具体指定一个应用的基本结构时，体系结构模式可用在粗粒度设计的开始阶段中。
- 设计模式适用于粗粒度设计的结束阶段，在需要细化并且扩展软件系统的基本体系结构时，例如，决定子系统之间的基本通信机制。设计模式在指定局部设计方面同样适用于详细设计阶段，例如对组件的多种实现所需要的支持。
- 惯用法用于在实现阶段将一个软件体系结构转换成一个使用某种特定语言编写的程序。

363 注意：虽然上述准则在许多场合正常工作，但是它们不是不可改变的规则。例如，如果你想要用对子系统的层式化抽象的单件实例化，例外就会发生。应该首先考虑到单件（Singleton）模式[GHJV95]，然后再考虑怎样用层模式构造子系统。

5.2.2 问题类别

我们的第二个分类标准提供模式系统的一个面向问题的视图。每个模式解决一个可能在软件系统开发过程中出现的特定问题。例如，转发器-接收器模式描述怎样在分布式组件之间实现对等通信，客户机-分配器-服务器模式描述怎样在分布式系统中实现位置透明性。由特定问题抽象得来的问题类别揭示了一些相关问题。例如，转发器-接收器和客户机-分配器-服务器处理实现进程间通信，或者更一般的情况下组件之间的通信时出现的问题。问题类别直接对应具体的设计环境。因此对模式来说，它是一种有用的模式分类标准。我们定义以下的问题类别：

- 从混沌到结构，包括那些支持将整个系统任务适当分解成相互合作子任务的模式。
- 分布式系统，包括那些为有组件分布在不同进程中的系统或有组件分布在几个子系统和组件内的系统提供基本结构的模式。
- 交互式系统，包括那些用来帮助构建人机交互系统的模式。
- 适应性系统，包括为应用程序的扩展和改进提供基本结构的模式，以响应演化和变更功能

需求。

- 结构化分解，包括那些支持将子系统和复杂组件适当分解成相互合作部分的模式。
- 工作的组织，包括那些定义组件怎样协作以提供一种复杂服务的模式。
- 访问控制，包括那些防止与控制对服务和组件访问的模式。
- 管理，包括那些用来对同类的对象、服务和组件的集合进行全面处理的模式。
- 通信，包括那些有助于在组件间组织通信的模式。
- 资源处理，包括那些有助于管理共享组件和对象的模式。

364

然而，某些模式还是不能归入某个单一的问题类别。这些模式解决几个问题——一个主要问题和几个次要问题。我们把这些模式归入所有相关的问题类别。例如，我们将管道和过滤器模式归入“从混沌到结构和分布式系统”问题类别。

5.2.3 分类图式

模式类别和问题类别相互交织形成一个二维的模式分类图式——对每种模式来说，我们能定义它相应的模式和问题类别。

图式本身是很简单的、有表现力并且容易学习。这里只有两个分类标准。它们对应着软件开发过程中的两个主要方面：必须执行的一般性开发活动和必须解决的具体问题。两个标准也反映出模式的自然属性——规模范围和问题处理。

你可能注意到本书的结构反映出我们的分类图式。第2~4章对应于模式类别，并且每一章又根据不同的问题类别被进一步组织。下表给出了我们对模式的分类概述。

365

	体系结构模式	设计模式	惯用法
从混沌到结构	层（参见2.2.1节）		
	管道和过滤器（参见2.2.2节）		
	黑板（参见2.2.3节）		
分布式系统	代理者（参见2.3.1节）		
	管道和过滤器（参见2.2.2节）		
	微核（参见2.5.1节）		
交互式系统	MVC（参见2.4.1节）		
	PAC（参见2.4.2节）		
适应性系统	微核（参见2.5.1节）		
	映像（参见2.5.2节）		
结构化分解		整体-部分（参见3.2.1节）	
工作的组织		主控-从属（参见3.3.1节）	
访问控制		代理（参见3.4.1节）	
管理		命令处理器（参见3.5.1节）	
		视图处理程序（参见3.5.2节）	
通信		出版者-订阅者（参见3.6.3节）	
		转发器-接收器（参见3.6.1节）	
		客户机-分配器-服务器（参见3.6.2节）	
资源处理			计数指针（参见4.4.1节）

这个分类图式对其他模式也同样适用。例如，反应器（Reactor）和客户机-服务器（Client-Server）[PLoP94]是用于构建分布式系统的体系结构模式。组合消息（Composite Message）[SC95b]是一个针对通信问题的设计模式。句柄-主体（Handle-Body）[Cope92]是一个用于防止访问服务的惯用法。

我们的分类图式还是可扩展的——参见5.5节“模式系统的演化”我们可以增加新的模式和问题类别，以便对那些不能被归入现有类别的模式进行分类。以这种方法扩展图式不会妨碍我们现有的模式分类。

366

5.2.4 比较

对于定义组织模式来说，我们的分类图式并不是惟一的。或许最著名的图式描述在[GHJV95]中。与我们的图式一样，“四人帮”的图式是二维的：目的（*purpose*）和范围（*scope*）。以下的段落是从“四人帮”的书中摘录下来的。

> 第一个准则，称之为目的，反映出模式干什么。模式可以具有创建、结构或者行为的目的。创建模式关系到对象创建的过程。结构模式处理类或对象的组成。行为模式刻画类或对象相互作用的方式和分配职责。
>
> 第二个准则，称之为范围，规定这种模式主要适用于类还是对象。类模式处理类及其子类之间的关系。这种关系通过继承建立，因此它们是静止的——在编译时固定。对象模式处理对象关系，这种关系可在运行时改变并且更具有动态性。

根据这种分类图式，组合[GHJV95]和整体-部分是结构对象模式，而解释器[GHJV95]是行为类模式。

然而，我们认为结构模式和行为模式之间的区别太模糊。问题种类更具有表现力。它们明确地命名了在建造软件系统时开发者必须处理的具体问题范围。而且，“四人帮”的范围准则不能够帮助软件开发者选择模式。这是因为：它不与任何具体的设计环境或活动有关，也不适合非面向对象的模式，例如层模式或管道和过滤器模式。

模式的其他组织图式在参考文献[EKM+94]、[Zim94]和[BM94]中提到。[EKM+94]建立在问题类别之上，例如各种事务处理或者克服在面向对象的应用和关系数据库之间的差距，其做法与我们的图式如出一辙。[Zim94]侧重于模式之间的关系，如在它的解决方案中有“模式A使用模式B”或“模式A相似于模式B”等表述。

367

[BM94]是我们在本书中讲述的图式的前身。它是三维的。前二维——称为“粒度”和“功能属性”——直接与我们的模式和问题类别对应。第三维“结构化原理”描述模式推荐解决方案的基础技术原理。例如，整体-部分模式基于策略和实现的分离[RBPEL91]。但是，如同“四人帮”图式的范围准则一样，结构化原理准则在选择模式时并不重要，因此我们在定义新的分类图式时舍弃了它。

5.3 模式选择

基于我们的分类图式，我们的模式描述模板（参见第1章“模式”），以及模式间的关系，我

们可以定义以下的简单过程来选择一种具体的模式。它包括如下7个步骤：

(1) 具体指定问题。为了能找到可以帮助你解决具体问题的模式，你必须首先精确描述问题：主要问题是什么，它的强制条件是什么？如果主要问题有几个方面，例如定义一个分布式交互系统的基本体系结构时，可以将问题分成子问题来处理。分别描述每个子问题及其强制条件。对每个子问题，努力寻找可以解决该问题的模式。

例如，让我们假设你的问题是定义一个交互式文本编辑器的基本结构。该系统应该便于移植到不同的用户界面库上和适应不同用户定义的风格。我们将使用该例子来说明其余的模式选择步骤。

(2) 选择符合你正在进行的设计活动的模式分类。对我们的例子来说，需要定义文本编辑器的基本结构。因此我们选择体系结构模式分类。

368

虽然这一步不要求涉及设计问题的详细知识，但是它已经有效限制了那些可能适用于设计问题的模式的数量。

(3) 根据设计问题的自然特性选择合适的问题分类。每个问题分类广泛地总结了包含在该分类中的模式所解决的问题类型。在我们的文本编辑器例子中，我们可以选择交互式系统问题分类，此处我们找到模型-视图-控制器模式(MVC)和表示-抽象-控制模式(PAC)。如果没有问题分类能够与具体的设计问题相匹配，尽可能选择可替换的问题分类(步骤7)。

(4) 比较问题描述。在已选取问题分类中的每个模式将会针对具体问题的一个特别方面，不论是单个模式还是几个模式的组合都能够帮助解决具体问题。选择那些其问题描述和强制条件与你的设计问题能够最佳匹配的模式。这一步第一次要求具有所解决设计问题的专业知识。例如，对一个文本编辑器来说，我们或许将选择模型-视图-控制器模式。MVC和PAC都支持系统用户界面的改变。但是，因为文本编辑主要由一组关系紧密的功能组成，而不是几个独立的子域，所以我们的编辑器不需要由PAC推荐的基于agent的结构。

如果所选问题分类中的模式的确不能解决具体设计问题的某个方面，尽可能选择一个可替换的问题分类(步骤7)。

(5) 比较优点和不足。这步调查到目前为止使用所选模式的效果。挑选那些能为你的需要提供便利，而其不足对你影响最小的模式。因为我们已经为我们的文本编辑器的体系结构选择了一个特定的模式，所以我们跳过这一步。

(6) 针对你的设计问题，选择可以最好地实现解决方案的变体。就我们的文本编辑器例子而论，视图和控制器的功能通常紧紧交织在一起。因此我们选择MVC中的文档视图变体来具体指定我们的编辑器的体系结构。

369

除非你在第3步或第4步遇到问题，否则现在你已经完成了模式选择。

(7) 选择一个可替换的问题分类。如果没有合适的问题分类，或如果被选择的问题分类不包含你能使用的模式，试着选择一个可以进一步概括你的设计问题的问题分类。这个分类包括的模式特例化后，能够帮助你解决问题。然后回到步骤4“比较问题描述”。

很多模式是经过不同问题分类的模式特例化得到的。例如，针对通信问题的组合消息模式[SC95b]，基本上是结构化分解问题分类中组合模式[GHJV95]的特例化。如果你面临的问题由组合消息解决，但是得到它是不可能的，那么你可能会改用组合模式来替代它。

如果经过步骤2、3和步骤4没有得到结果，甚至在选择了可替换的问题分类后也没得到结果，你应该停止在这个模式系统中的搜寻——这个模式系统不包含能够帮助你解决设计问题的模式。你应该查找其他的模式语言、系统或者目录去看看它们是否包含你能使用的模式，或者不使用模式解决你的设计问题。

当实现或者细化一个你已经选择的模式时，不需要使用搜索过程。我们的模式描述的实现部分直接引用那些模式，它们自然补充被实现的模式。

5.4 作为实现指南的模式系统

在我们的模式描述中提供了用来指定模式实现的步骤和指南。这些模式描述帮助完成一项转化任务，即将不包含模式的软件体系结构转化成为包含模式的软件体系结构。实现步骤可以被看成是使用模式处理具体问题的一种*微方法*（*micro-method*）。

370

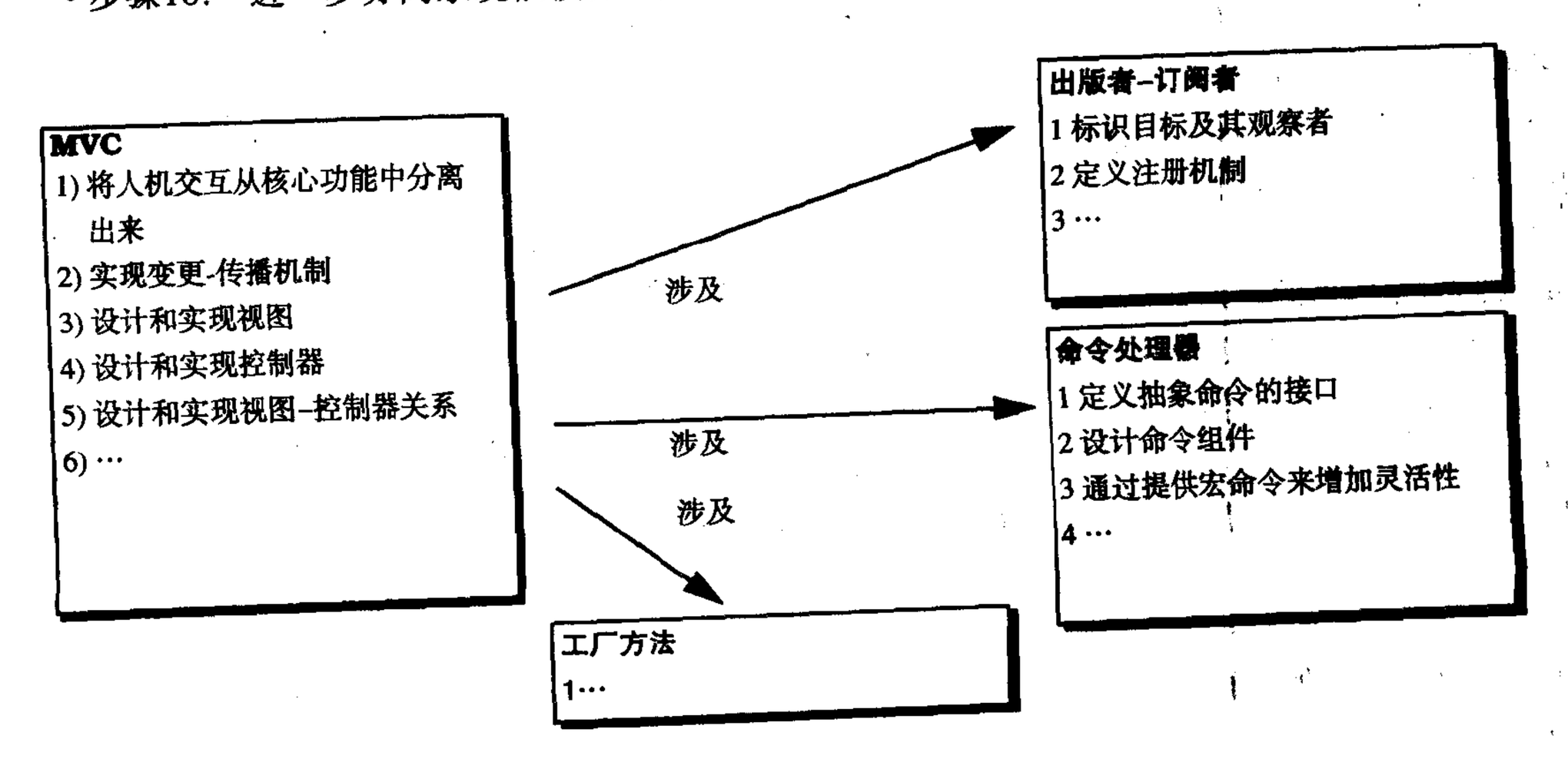
就像模式本身一样，它们的实现步骤是交织在一起的——它们经常引用其他模式来补充被描述的模式。每当另一个模式被引用时，它的实现步骤就可以被使用。

例子

实现模型-视图-控制器体系结构。

模型-视图-控制器的实现部分涉及7种其他模式：

- 步骤2:“实现变更-传播机制”建议使用出版者-订阅者设计模式（参见3.6.3节）。
- 步骤4:“设计和实现控制器”涉及命令处理器设计模式（参见3.5.1节）。
- 步骤5:“设计和实现视图-控制器关系”涉及工厂方法设计模式[GHJV95]。
- 步骤7:“动态视图创建”建立在视图处理程序设计模式（参见3.5.2节）的基础上。
- 步骤9:“层次视图和控制器的基础设施”使用组合模式[GHJV95]和职责链模式[GHJV95]。
- 步骤10:“进一步分离系统依赖性”建议使用桥接模式[GHJV95]。



371

上述例子揭示所有模式的实现步骤为软件设计和实现共同形成一套可扩展的准则。单个模式的实现步骤是它的构造块。可将其他模式的实现步骤加入其中，这里的其他模式是指在你实现模式时所涉及到的模式。因此你能通过递归应用所有涉及其解决方案的模式的实现步骤来解决复杂问题。

解决具体问题的焦点集中在如何将模式的实现准则区别于现有的分析和设计方法，如Booch[Boo94]、Coad/Yourdon[CY91]、“对象建模技术”[RBPEL91]或者Shlaer/Mellor[SM88]。它们只提供了软件开发的一般性的且与问题无关的指南，如“标识系统建模所需的对象/类”[Kar95]。具体的体系结构的构建（例如模型-视图-控制器体系结构）仍然基于你自己的经验和直觉。

你可能想知道怎样由单个模式实现步骤派生得到完整的准则。检查我们的模式系统通常可以发现对软件开发的支持相当不完全，力度也很小。我们仅仅覆盖了系统中至少包括一种模式的软件体系结构的那些问题领域。对于很多问题领域，我们的准则并不支持，因为我们不提供针对这些问题的模式。例子包括：组件创建，事件处理，事务处理，用关系数据库连接面向对象的应用程序，应用程序扩展新功能，等等。

但是，我们设计的模式系统应该是可扩展的（参见5.5节“模式系统的演化”）——对还没有覆盖到的问题领域，可以用模式来扩展。以这种方法集成一个新模式时，我们同样定义它与其他模式的关系。它集成了新模式的实现步骤和现有相关模式的实现步骤。因此，每个新模式扩展了整个模式系统的指南，使它们变得更有力，更具体，并且包括更多软件体系结构的问题领域。 372

然而，即使是非常全面的模式系统，也不能够和不应该覆盖软件体系结构的每一个问题领域。总有空白点——设计问题没有对应可使用的模式。[Cope96]坚持“宽广的设计空间非常有助于用普通技术实现著名范例”的观点。例如，不需要使用模式来描述一般的模块接口或者过程的用法。单个模式的实现准则同样也不能解决软件开发中的一般性问题，如提供整个过程和软件生存周期模型。模式并不因此为软件开发定义一个新方法来替代现有的方法。相反，在解决特定问题和具体问题时，它们用指南来补充一般性的且与问题无关的分析和设计方法。

因此我们建议采用以下的实用步骤来使用模式去开发软件系统：

(1) 使用你喜欢的一种方法去定义软件开发的整个过程和在每个开发阶段内执行的详细活动，如Booch[Boo94]、Coad/Yourdon[CY91]、“对象建模技术”[RBPEL91]、Shlaer/Mellor [SM88]、职责-驱动-设计 [WBWW90] 或者“统一方法”[BR95]。

(2) 使用适当的模式系统去指导解决具体问题方案的设计和实现。一旦这个模式系统包含一个能够解决你面临的设计问题的模式，你就可使用那个模式的实现步骤来解决此问题。如果问题涉及其他模式，递归地应用这些模式和它们相关的实现步骤补充你的原始模式的实现。

(3) 如果模式系统不包括你设计问题的模式，力求从你知道的其他模式源中寻找一个模式。

(4) 如果没有模式可供使用，应用你所用方法的分析和设计指南。这些指南至少可以为你手中的设计问题提供一些有用的支持。

这种简单的方法避免了还要定义另外的设计方法。它结合了由现有分析和设计方法所捕捉 373

的软件开发经验和由模式描述的具体设计问题的实际解决方案。

5.5 模式系统的演化

即使是最成熟的模式系统也不会保持静止状态。知识随着时间不断演化——新技术发展了，现有的技术进步了或者过时了。新模式将因此出现，而现有的模式可能会“死亡”。出现的新模式必须集成到模式系统中以保持其是最新的。如果过时的模式不再被使用，则一定要删除它。甚至单个的模式描述会随时间而改变——阐明问题的特定方面和加入进一步知道的用法。一旦新模式集成到系统中，或者一种现有的模式被删除，那么现有模式之间的关系就一定要更新。

在模式系统演化的语境中，必须考虑几个问题：模式描述的演化，“模式采掘”，新模式集成到系统中，过时模式的删除和组织图式的扩展。下面将讨论这些问题。

5.5.1 模式描述的演化

不断改进并稳定每个模式在模式系统内的描述对于保持系统有用性是非常重要的。模式越成熟，它属于模式系统的时间就越长，成功应用的机会也就越多。每当一种模式被使用时，从应用中获得的经验应该被用作模式及其描述的关键性的评审依据。

374 这样的评审将重新认识由模式提供的额外的好处以及模式潜在的不足和局限。你可能也认识到需要对模式的结构和动态特性进行轻微修改，或者在模式描述中集成新的变体。

例子

代理

代理是一个模式及其描述演化的好例子。在[GHJV95]中的原始描述列举了3种变体：远程代理、虚拟代理和保护代理，其具体细节也与一般原理的描述交织在一起。在[PLoP95]中我们提出了另一种描述，它将代理模式的一般原理与其具体使用的细节分离开。我们提出了另外4种代理：高速缓存代理、防火墙代理、计数代理和同步代理。基于从代理描述的许多评审中得到的反馈信息，我们进一步改进了模式描述。我们加深了对本质的描述，同时增加了各种变体的技术信息。这个改进过程的成果是可以在本书中找到代理模式。 ❑

每当你成功地应用了模式，你就可以通过扩充它的已知使用列表来进一步固化模式。已知使用列举得越多，用户根据这些用法描述确定相似的设计环境的机会也就越大。在这样的情况下，用户能从模式的“参考应用”中得到以前的经验，从而直接受益。

5.5.2 作者研讨会

每个模式评审都应该遵循一种结构化的形式。其目标是为进行建设性改进获得尽可能多的反馈。非结构化的评审不够系统化——内容以任意和无关的顺序增长，改进的很多方面没有经过讨论或者仅仅进行了简要讨论。

我们建议模式评审的形式采纳一种书面作品（尤其是诗作）的评审形式。召开一次作者研讨会，当进行模式评审时它遵循以下形式：

375

- 模式由包括其作者和一群熟悉模式描述内容的评审人一起讨论。研讨会现场的支持人（moderator）帮助参加者遵守研讨会的规定。
- 模式描述的作者阅读他们从模式描述中所选择的段落。
- 两个评审人总结来自他们个人观点的描述。
- 在不同的阶段，首先讨论的是模式的优点，然后是它的缺点，最后是剩下的其他方面。在讨论中，模式描述的作者仅是“事实上”与会——在讨论中，作者既不积极参与，评审人也不直接针对作者发言。评审人应该像模式描述的作者不在现场一样讨论模式描述。尽管如此，还是允许作者对讨论做记录。
- 讨论之后，作者向评审人提问以澄清一些特殊的陈述。
- 以作者对讨论作出最后评论作为会议结束。

模式描述的改进基于作者研讨会的讨论结果。在[PLoP94]和[PLoP95]中的所有模式，以及我们书中的大多数模式，都在作者研讨会上进行过讨论。

5.5.3 模式采掘

解决一个具体的设计问题并不总是有一种合适的模式适合于它。在这样的情况下，用“采掘”新模式来解决这样的问题非常有效，特别是你要经常面对它们的时候。以下的经验已被证明是可行的：

(1) 能找到至少3个例子证明一个特殊的设计或实现问题可以使用相同的解决方案图式得以有效地解决。例子应该全部来自不同的现实世界系统，并且所有系统应该由不同的团队开发。

376

(2) 抽取出解决方案图式。从具体应用的特定细节中抽象出一般的解决方案图式。使用一个合适的模式描述模板，描述解决方案图式针对的问题，以及与问题相关的强制条件。列举源于“已知应用”的解决方案图式的例子。

(3) 宣布这个解决方案图式为“候选模式”。

(4) 召开作者研讨会来改进候选模式的描述并且与你的同事共享你的研究成果。

(5) 在一个现实世界软件开发项目中应用候选模式。

(6) 如果候选模式的应用是成功的，宣布这个候选模式成为一个正式的模式，并把它集成到模式系统中。通过召开另一次作者研讨会来改进它的描述。把这个新应用添加到模式的已知应用列表之中。

如果候选模式的应用失败了，改进模式的描述并试着再次应用它。或者，考虑完全地抛弃候选模式并寻找一个更好的解决原始问题的方案。

5.5.4 新模式集成

当把一个新模式（不论是一个现有的模式还是“采掘”得到的模式）集成到模式系统中时，你需要进行两项活动：

(1) 指定新模式与模式系统中其他模式的关系，以及从现有模式到新模式的所有关系。

(2) 通过把模式归入相应的模式和问题类别来为这种模式分类。如果你不能把新模式归入现有的类别，就扩展相应的组织图式(参见下面第6小节)。

377

5.5.5 删除过时模式

由于技术演化，模式可能变得过时。原因有以下几点：

- 问题的消失。过去必须被明确处理的一个问题可能现在已经由正在使用的编程语言或者系统环境处理。例如，在C++中引入无用单元收集后使得几个处理共享对象的C++惯用法变得多余。
- 更好的选择。一个新的解决特定设计问题的方法变得可使用，它优于处理相同问题的现有模式。
- 技术演化。一个新的范例，编程语言和风格的演化，开发系统种类的一个变化，都可以使现有模式变得过时。

主程序和子例程（Main Program and Subroutines）[PLoP95]是一个过时模式的例子。它建议将应用的功能分解成一组“嵌套的过程集合”。当结构化程序设计刚出现并且程序量较小的时候，主程序和子例程是一种有用的模式，因为它帮助程序员考虑系统分解。一个程序只要不是一整块代码就被认为是好的结构化程序。

现今，几乎所有程序都使用子例程，甚至是结构化很差的子例程。主程序和子例程是否是系统主要的体系结构原则，已不再是质量的标志。这是因为系统在大小和功能的复杂性方面持续增长。它们变得越来越分布式，许多还提供了图形用户界面。然而，复杂系统除了由主程序和子例程描述的这个原则外，还需要其他的体系结构原则——这个曾经有用的模式已经变得过时。

什么时候应该从模式系统中删除“垂死的”模式？当开发新软件系统时，当然不应该使用这种模式。但是，为了维护以前的系统，有必要仍然使用它。这样的系统可能遵循过去的程序设计惯例。因为“流行”模式的应用可能打破原来系统的体系结构版本，它们的应用经常没有意义。我们必须使用“老式”且符合现有体系结构的模式。因此必须当一个过时的模式不可能用在任何未来的软件开发或系统维护的时候，才能将它从模式系统中删除。

378

5.5.6 扩展组织图式

随着模式系统的演化，可能有必要修改它的组织图式。例如，我们可能需要增加新的模式分类。就像在本书中定义的那样，我们的模式系统只包括在软件开发过程中具有一般应用特性的模式。我们不提供领域特定的模式来指定在特定应用领域内的工作组织，而且我们也不提供在分析阶段可以应用的领域特定的模式。

为了集成这样的模式，我们可以增加一个“分析模式”类别。或者，我们可以定义新的问题分类，例如为组件创建定义新的问题分类。新问题分类必须使用[GHJV95]中的模式来扩展模式系统，新的问题分类如下所示：

- 创建包括帮助实例化对象和递归对象结构的模式。

• 服务变体包括支持对象或组件的行为改变的模式。

• 服务扩展包括帮助动态地给对象或对象结构添加新服务的模式。

• 适应性提供帮助接口和数据转换的模式。

所有其他的“四人帮”模式都可以被归入现有的问题类别中。下述表格显示了如何将“四人帮”的模式集成到我们的模式系统中。为了区分“四人帮”模式和我们的模式，我们的模式用宋体显示他们的模式用楷体显示。

379

	体系结构模式	设计模式	惯用法
从混沌到结构	层（参见2.2.1节）	解释器	
	管道和过滤器（参见2.2.2节）		
	黑板（参见2.2.3节）		
分布式系统	代理者（参见2.3.1节）		
	管道和过滤器（参见2.2.2节）		
	微核（参见2.5.1节）		
交互式系统	MVC（参见2.4.1节）		
	PAC（参见2.4.2节）		
适应性系统	微核（参见2.5.1节）		
	映像（参见2.5.2节）		
创建		抽象工厂	单件
		原型	工厂方法
		生成器	
结构化分解		整体-部分（参见3.2.1节）	
		组合	
工作的组织		主控-从属（参见3.3.1节）	
		职责链	
		命令	
		中介者	
访问控制		代理（参见3.4.1节）	
		外观	
		迭代器	
服务变化		桥接	模板方法
		策略	
		状态	
服务扩展		装饰	
		访问者	
管理		命令处理器（参见3.5.1节）	
		视图处理程序（参见3.5.2节）	
		备忘录	
适应		适配器	
通信		出版者-订阅者（参见3.6.3节）	
		转发器－接收器（参见3.6.1节）	
		客户机-分配器-服务器（参见3.6.2节）	
资源处理		享元	计数指针（参见4.4.1节）

380

另一个可能的扩展是增加新的分类准则，例如由[GHJV95]定义的范围(Scope)，或在[BM94]

中描述的启用技术（Enabling Technique），它们定义的原则成为特定模式的基础。但是，这是真的有用吗？我们并不这样认为。首先，一个多维的图式变得超载了。用户面对着许多不同的分类准则，而众多的分类准则使得模式系统很难理解和使用。第二，在选择模式时，更多的准则要求掌握有关当前设计问题的更多细节知识。

更进一步，仅当现有模式组变得非常庞大并因而很难处理时，我们才会为分组模式考虑更加细致的准则。你的目标应该一直是帮助用户获得模式系统中模式的概观，并指导模式的选择，而不是提供一套完整和详细的分类，可以覆盖模式所暴露的每一个属性。

5.6 总结

在各种规模和各种抽象程度中都存在软件体系结构的模式。它们可以用在软件开发的不同阶段并处理各种不同的问题。它们也彼此展示不同的关系。一个相关模式集合所带来的好处远大于集合中的每个单独模式所带来的好处的总和。

为了利用这种模式集合的优势，我们需要把它们组织成模式系统。模式系统以便利的方式处理数量众多的模式。它统一描述所有的模式。通过模式分类，提供对它所包含模式的概观。它通过提供合适的搜索策略来支持模式的选择。它提供一套准则来支持用模式对软件系统进行开发。最后，模式系统支持它自己的演化。

381 我们的模式系统包括在软件开发过程中一般的应用性的模式，从软件系统的基本体系结构的规格说明到使用一种具体的编程语言来实现具体的设计问题。可用针对软件体系结构的构造深层问题的模式来扩充模式系统，如来自[GHJV95]、[Sch95]、[Cope92]的那些模式以及在[PLoP94]和[PLoP95] 中描述的许多模式。用这些模式扩充我们的模式系统为解决许多重新设计和实现问题提供了具体和实际的支持。

我们也能用特定领域的模式来扩充我们的模式系统，例如[PLoP95]中的交换系统模式。对特殊的应用领域，用模式覆盖大部分软件开发过程（从分析到实现）成为可能。这样的模式系

382 统成为建造软件系统的强有力的工具。

第6章
模式和软件体系结构

标牌上写着："拿住棍子中间，在嘴里弄湿尖的一端，插入到牙缝中，钝的一端紧挨着牙龈。轻轻地来回移动"。

Wonko清醒地说，"如果一种文明世界失去了自己的头脑以至于对如何使用牙签也需要详细说明的话，它将不再是我能够生活和保持头脑清醒的一种文明世界。"

——*Douglas Adams《So Long, and Thanks for All the Fish》*

模式是构造高质量软件体系结构的一个重要工具。但是，软件体系结构的几种其他的技术、方法和过程都已经有了。模式怎样建造在这些技术、方法和过程之上？怎样补充它们？模式能定义软件体系结构的技术发展状态吗？

在这一章里，我们讨论模式怎样集成到软件体系结构的更多领域。但是，本章没有提供对软件体系结构的完整概况。

383

6.1 引言

在讨论模式是如何和软件体系结构集成在一起之前，需要刻画我们对于这个领域特性的理解。因此在这一节里我们简要讨论一些与软件体系结构原则有关的重要方面。我们给出以下术语的定义：

- 软件体系结构（Software Architecture）
- 组件（Component）
- 关系（Relationship）
- 视图（View）
- 功能属性（Functional Property）
- 非功能属性（Non-Functional Property）
- 软件设计（Software Design）

6.1.1 软件体系结构

通观全书，我们使用"软件体系结构"这个术语却没有做任何进一步解释，这是因为我们认为你已经对这个术语的含义有了一个直观的理解。可是软件体系结构对于我们真正意味着什么？

软件体系结构（*software architecture*）是对子系统、软件系统组件以及它们之间相互关系的描述。子系统和组件一般定义在不同的视图内，以显示软件系统的相关功能属性和非功能属性。系统的软件体系结构是一件人工制品。这是软件设计活动的结果。

384

6.1.2 组件

组件（*component*）是软件系统的一个封装部分。组件有一个接口。对于系统的结构，组件就像积木一样。在编程语言层次，组件可表示为模块、类、对象或是一组相关函数。

以下表格显示了3种不同的组件：

```
DEFINITION MODULE
  CoreData;

FROM Sys IMPORT
 ObjType,
 ObjID;

EXPORT QUALIFIED
 PROCEDURE
  newObj():ObjType;

 PROCEDURE
  loadObj(ID:ObjID):ObjType;

 PROCEDURE
  storeObj(obj:ObjType);

END CoreData.
```

MODULA-2定义模块

```
class Random {
private:
   int seedA;
   int seedB;
public:
   Random(int seed);
   ~Random();
   int random_card(int max);
};
```

C++类定义

```
float sin(float x) {
  /* ... */
};

float cos(float x) {
  /* ... */
};

float square(float x) {
  /* ... */
};

float square_root(float x) {
  /* ... */
};
```

C函数

注意，组件可以有非常不同的性质。例如，在代理者模式中，我们提到“代理者组件”。根据这种模式的实现方式，代理者组件可能是一个链接库或是一个独立的进程。术语“组件”——至少第一眼看来——与它用源代码表示的最终情况无关。

我们有时对术语“组件”的使用过于宽松。例如，当我们谈到“客户机组件”时，我们有意想要暂时忘记客户机是怎样实现的。相反，我们想将精力集中于另一个不同的问题，例如，指定客户机怎样才能充分利用模式提供的服务。

385

但是，我们怎样才能原则上对组件进行分类呢？在这里我们列举两种不同的方式。例如，[PW92]区分3 种不同种类的组件，称为*元素*：

- 处理元素
- 数据元素
- 连接元素

处理元素提供包含被转变信息的数据单元的转化。在任何时候，连接元素既可以是处理元素，也可以是数据元素，或者两者皆是。它是将不同部分连接在一起的“胶水”。

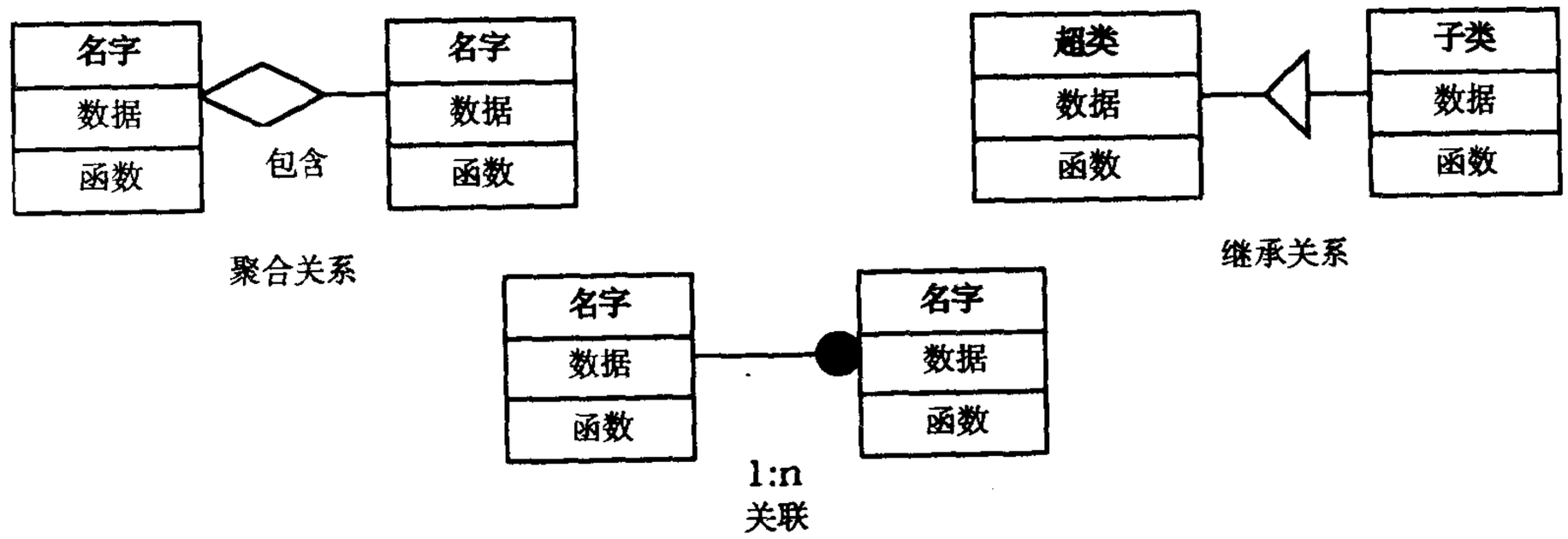
另一种为面向对象程序设计范例而开发的组件分类如下：

- 控制器组件
- 协作者组件
- 接口组件
- 服务提供者组件
- 信息持有者组件
- 构造用组件

6.1.3 关系

关系（relationship）[⊖]表示组件之间的连接。关系可能是静态的，也可能是动态的。静态关系可以直接用源代码显示，它们负责在体系结构内放置组件。动态关系处理临时的连接和在组件间的动态交互。从源代码的静态结构中是不易看出动态关系的。 386

聚合和继承是静态关系的例子。对象的创建、对象之间的通信和数据传输通常是动态关系。一个临时关系的例子是及时将一个对象在某点插入一个容器并且事后删除它。下图用OMT符号表示了3 种静态关系 [RBPEL91]。

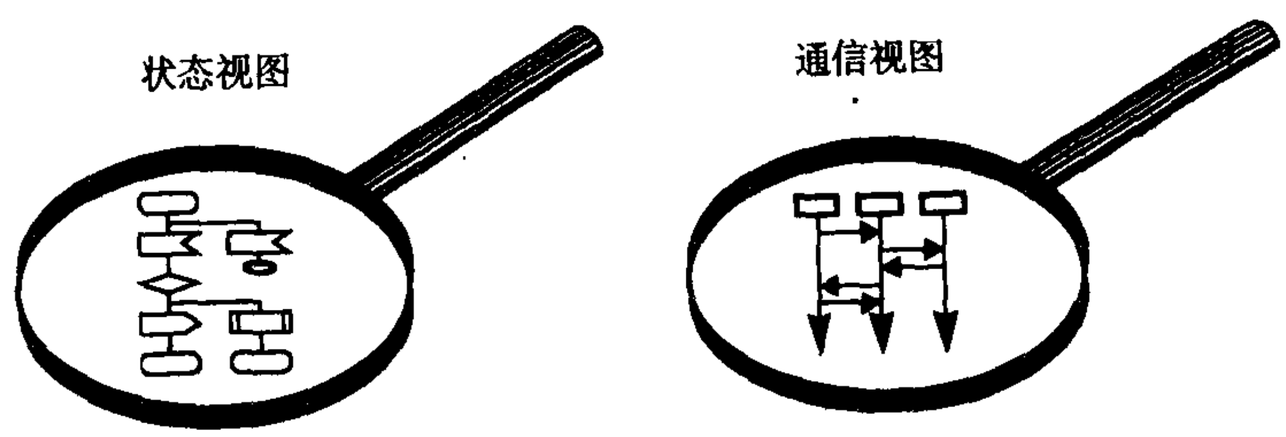


组件之间的关系对软件体系结构的总体质量有很大的影响。例如，如果在一种软件体系结构中，关系支持组件的变化，那么这种软件体系结构对可变性的支持要远远好于那种组件的任何变化都会影响其客户和协作者实现的软件体系结构。关系的重要性在大多数软件体系结构的新近定义和讨论中都有清楚说明[SG96][PW92] [KMS+92]。

6.1.4 视图

视图（*view*）代表一个软件体系结构的部分方面，这个部分方面专门显示一个软件系统的特定属性[⊖]。

视图的例子有组件的状态视图，或者组件之间关系的通信或者数据流视图。 387



⊖ 软件体系结构的其他定义使用术语“连接器”（Connector）而不是关系（relationship）[SG96]。

⊖ 注意，尽管在这里使用术语“视图”（View），但是却与我们模式中的视图组件（View component）没有直接关系。

[SNH95]提议使用以下4 种不同视图来描述软件体系结构：

- 概念上的体系结构：组件，连接器……
- 模块体系结构：子系统，模块，出口，入口……
- 代码体系结构：文件，目录，库，包含文件……
- 执行体系结构：任务，线程，进程……

在[Kru95]中采用了一种类似的方法。由选中的用例增强的4种不同的视图描述了软件体系结构。

- 逻辑视图：设计的对象模型，或一个相应模型（如实体关系图）。
- 进程视图：并发和同步情况。
- 物理视图：软件到硬件的映射及其分布式情况。
- 开发视图：在软件开发环境中的软件静态组织。

在两种方法之间有着明显的重叠。例如，概念上的体系结构与逻辑视图看上去非常相似。其他视图彼此之间映射得并不好。例如，模块和代码体系结构合起来才覆盖了开发视图，但是也可能覆盖了另一些情况。用两种方法描述同一个体系结构的例子是很有意思的。

388

6.1.5 功能属性和非功能属性

在讨论软件体系结构时，我们经常会听到“非功能属性”这个术语。与此相反，“功能属性”仅仅被隐含地假设。

功能属性（*functional property*）用来处理系统功能性的特定方面，并且通常与特定的功能需求相关。功能特性可以通过特定的功能使用户直接可看到应用程序，也可以通过它的实现来描述，例如用来计算功能的算法。

开发者过去习惯于专心给软件提供一定的功能属性，而今天，非功能属性变得越来越重要：

非功能属性（*non-functional property*）定义了未被功能属性描述覆盖的系统特征。非功能属性通常解决与一个软件系统的可靠性、兼容性、开销、易用性、维护或者开发有关的方面。

在6.4节“软件体系结构的非功能属性”中我们详细讨论了以下非功能属性：

- 易修改性（changeability）
- 互操作性（interoperability）
- 效率（efficiency）[⊖]
- 可靠性（reliability）
- 可测试性（testability）
- 可重用性（reusability）

389

当设计软件体系结构时，非功能属性非常重要。首先，软件系统随时间演化。它们必须相应地改变技术、需求和系统环境。因此仅仅恰当分解全部应用任务是远远不够的——系统还必须为变化、扩展和适应做准备。如果软件系统没有完成，尤其当它的寿命很长时，维护起来将会

⊖ 在本书中我们认为效率是非功能属性。然而，效率约束也可作为功能需求的一部分，例如在实时系统中。类似参数为其他非功能需求所拥有，而这些非功能需求在客户明确要求时可以变为功能需求。

变得困难和昂贵。其次，软件系统的功能性必须遵从总体需求，例如，为了它的总体可操作性、可靠性或效率。为了满足这种需求，需要恰当地设计软件体系结构。

6.1.6 软件设计

软件设计（software design）是以系统的软件体系结构为目标的软件开发者所执行的活动。我们所关心的是在给定的功能属性和非功能属性内指定软件系统的组件和组件之间的关系。

对于系统高层结构子划分，传统的做法是使用“软件体系结构”、“软件体系结构设计”或“粗粒度设计”这样的术语；而对于更详细的计划，则使用术语“设计”或“详细设计”。如前所述，我们用“软件设计”和“软件体系结构”分别表示建造一个软件系统的整体活动以及由此而得到的人工制品。

现在，许多开发者更喜欢采用术语“软件体系结构”而非“软件设计”来表示由设计活动而得到的所有人工制品。他们这样做的目的是想要表示：他们不仅仅只是将系统功能分解为一组协同工作的组件，而且更进一步他们要构造一个软件体系结构。他们想表示他们的注意力明确地集中在软件系统组件的适宜构建上，而关于这些组件，它们的附带责任，它们的功能性和接口，它们的内部结构，它们之间的多种关系以及它们的协作方式——所有这些都明确考虑了非功能属性(如易修改性和可移植性)。它们不再同意高层的设计决定可以独立于低层决定的思想。 390

6.1.7 小结

本节的简短讨论已经显示出软件体系结构设计已经远远不是一项在有限范围内的简单活动。它包括软件工程的技术层面、方法层面和处理层面。它明确针对多产软件开发和维护的需要，并对软件系统的最终质量有很大影响。

在下一节中，我们将表明模式是如何解决软件体系结构的需要以及它们与现有方法是如何联系在一起的。

6.2 软件体系结构中的模式

我们关于模式的工作与软件体系结构，面向对象或者过程的分析、设计和程序设计等其他工作有着非常密切的联系。

我们的模式建立在设计者和程序员近三四十年来在软件开发中得到的极其广泛的实际经验基础之上。我们描述的模式都不是人工构造出来的，它们既不是我们也不是别人构造的——它们是随着时间演化得来的。软件开发者认识到：使用某个特定方法解决一个问题要比使用其他方法好，因此它们一而再，再而三地使用这种解决方法。我们描述的一些模式已经存在很久了。例如，管道和过滤器模式在20世纪60年代就有了，模型-视图-控制器模式在20世纪70年代后期也出现了[KP88]。没有这些实际经验，模式将不会存在。

模式同样也明确建立在为结构化程序设计而开发出的许多原理基础之上，模式并非仅致力

391 于对象技术。在20世纪70年代发展起来的许多程序设计原理构成了我们的模式基础。我们将在6.3节“软件体系结构启用技术”中讨论模式与这些原理之间的关系。

模式的另一个目标是使用可预测的非功能属性建造软件系统。因此模式建造也就基于这样的原则：为重用进行软件开发并且使用重用开发软件，为变更而进行设计，等等。我们在6.4节“软件体系结构的非功能属性”中也讨论了模式和重要的非功能属性之间的关系。

6.2.1 方法学

关于模式的一个普遍问题是模式怎样与现有的分析设计方法相联系，如Booch方法[Boo94]、Coad/Yourdon方法[CY91]、对象建模技术 [RBPEL91]或者Shlaer/Mellor方法[SM88]。在我们有模式以前，这些方法作为“设计问题”的解决方案。近年来，人们对于方法学越来越关注——或者是在一定程度上过于依赖它们的想法。例如，Michael Jackson在[Jac95]中写到：

> 集中于问题而导致的失败已经损害了很多工程。但它导致了对开发方法演化的更多危害。因为我们不讨论问题，不分析问题，也不对问题进行分类，所以我们陷入幼稚的信仰中，认为有适于解决所有开发问题的通用开发方法。我们期望这些方法是能治百病的万能药。当然这是不可能的。方法的价值与它的通用性成反比，这是一个很好的经验法则。一个能解决所有问题的方法在解决任一特定问题时给你的帮助将是非常小的。

如果我们对期望不加限制，则不难预料人们将对模式产生相似的抱怨，James Coplien最近在[Cope96]中写到：

> 我隐藏模式的一个原因就是害怕设计者将因为这些模式的设计解决办案而首先注意它们。这在对象范例刚形成时也发生。其实很多设计问题可以通过著名范例解决，好的设计者在他们的工具箱里总是带着那些著名范例——不要总是试图使用最新的工具，即使它们是最强有力的。
>
> 不久前，我们已经努力使用对象工具来解决一切问题。模式引导我们走出呆板的对象设计方法，常常通过并不存在的范例带我们进入那些被处理得很好的结构。对我来说，那是在设计的黑暗角落里模式闪光的地方。对我来说，模式在设计空间内只覆盖了一些小洞：更宽的设计空间很好地适用于著名范例的公共技术，我们应该想办法在适合的地方使用那些范例。

392

通过降低我们的期望，我们可同时从模式和方法学的使用中获益。方法学为建造高质量软件提供了很多有用的步骤和指南。我们的模式的实现小节基本上遵循这些步骤，适应模式解决具体问题的需要。另外，这些方法定义了软件开发的全过程，你可以结合你对模式的使用改编并扩展它们。模式通过一组具体技术来补充现有的分析和设计方法，用于解决那些非常特别却又反复重现的设计问题。

记住，无论是模式、方法学还是它们的结合都不会为你建造良好体系结构提供“黄金大道”。将会有一大堆的设计问题等着你用自己的方式去解决。

6.2.2 软件过程

方法学的综合应用会引起软件进程的更加糟糕的问题。在项目中实施瀑布过程到底会引起

多大的危害呢？一个已定义的过程有它的优点，但是当它引起过大的组织机构开销或者执行并不适合你的工程目标的工作时，它就成了负担。一个努力适合所有项目的过程怎样才能适合你自己工程的特殊情况呢？如果不允许你回到设计阶段，你如何使用在实现期间获得的至关重要的洞察力来重新设计你系统的已定义部分呢？你应该不允许使用任何方法学或过程来严格指定用于进行设计和实现的方法。

393

模式在这里有什么帮助呢？我们愿意把模式集成到一种增量式的提交过程中，这个过程避免了与开发阶段的严格分离。面向对象的分析和设计方法学倾向于模糊各阶段间的边界。我们希望能够使这种增量式的并且有时是循环往复的工作更加具有可预测性。例如，如果模式能够帮助产生更好、更稳定的设计，则我们可以限定经过各阶段的循环次数以及将重设计限制到系统的定义良好的部分上。

我们经常被问到，在分析期间，高层或低层设计期间，或者甚至在实现期间，模式应该在开发的哪一点使用模式？ 对此问题并没有惟一正确的答案，但有一条经验是：使用高层体系结构模式要早于中层设计模式，惯用法又在它们之后使用。5.2节“模式分类”更详细地讨论这个问题。

6.2.3 体系结构风格

1992年，Dwayne E. Perry和Alexander L. Wolf引入了体系结构风格的概念：

体系结构风格（*architectural style*）根据软件系统的结构组织定义了软件系统族。体系结构风格通过组件应用的限制及其与构建有关的组成和设计规则来表现组件和组件之间的关系。

一般说来，体系结构风格为一个软件系统及其怎样建造该系统的相关方法表示一种特殊的基本结构。体系结构风格也包括什么时候使用它所描述的体系结构、它的不变量和特例以及其应用的效果等信息。

394

例子

多阶段体系结构风格[PW92]

多阶段体系结构风格由处理元素和在处理元素之间交换的数据元素组成。例如，一个编译程序的多阶段风格包括：

- 处理元素：词汇分析器、语法分析器、语义分析器、优化器、代码生成器。
- 数据元素：字符、标记、短语、关联短语、注释短语、对象代码。

如果多阶段体系结构风格能够被顺序地组织，则它也使用如下连接元素：

- 连接元素：过程调用和参数。

体系结构风格的形式由加权属性和体系结构元素之间的关系表示。例如，在编译程序中优化器和注释短语必须要一起被发现，但它们只是受偏爱的元素而非强制性的要求。体系结构元素也被其他各种各样重要的观点（例如处理过程的观点）所制约。

例如，在编译程序里，强制词汇分析器（lexer）接受字符序列C，以便产生标记序列T，并且保持字符和标记之间的有序对应：

词汇分析器: C → T, 此处 T 保持 C

必须为给定体系结构风格中的每个元素指定处理限制条件。进一步的限制条件是为组件间的连接、数据流和计算状况定义的。所有限制条件一起强有力地决定了使用多阶段顺序体系结构风格的软件系统的具体体系结构。 ❑

体系结构风格在[SG96]和[SNH95]中也提出过。体系结构风格非常类似于我们的体系结构模式。事实上，每一个体系结构风格都可以描述成一种体系结构模式。例如，多阶段体系结构风格对应管道和过滤器模式。另一方面，体系结构风格在以下几个重要方面不同于模式：

395

- 体系结构风格只描述应用程序的总体结构框架。而软件体系结构模式存在于各种各样的规模范围内，从定义应用程序基本结构的模式(体系结构模式)开始到描述怎样用给定的编程语言实现特定设计问题的模式(惯用法)结束。
- 体系结构风格彼此独立，但模式依赖于它所包含的较小模式，依赖于相互作用的模式和包含它的较大模式[Ale79]。
- 模式比体系结构风格更加面向问题。体系结构风格从不依赖实际设计环境的观点来表示设计技术。模式表示非常具体的递归设计问题并且提出解决它的方法，所有这些来自问题出现语境中的观点。

6.2.4 框架

框架是软件体系结构的另一种重要方法：

框架（*framework*）是一个试图实例化说明的部分完整的软件(子)系统。它为一个(子)系统族定义体系结构并提供创建它们的基本构造块。它也定义具体功能特性需要改进的地方。在面向对象的环境中框架由抽象类和具体类组成。

框架的实例化包含现有类的组成以及对现有类的再分类。在具体领域中的应用程序框架被称为*应用程序框架*（*application framework*）。

根据文献[Pree94]中的定义，应用程序框架由*冰点*（*frozen spots*）和*热点*（*hot spots*）组成。冰点定义软件系统的总体体系结构——它的基本组件和组件之间的关系。这些内容在应用程序框架的任意实例化中都保持不变。热点表示应用程序框架中专门针对单个软件系统的那些部分。热点设计成通用的——它们可以适应正在开发的应用程序的需要。

396

当用应用程序框架创建一个具体的软件系统时，其热点根据系统特定需要和需求而被特例化。为了实现应用程序框架的适应性和易修改性，你可以不局限于面向对象技术(如继承性和多态性)——你也可以使用模式[Ta194]。例如，在InterViews框架中[LCITV92]应用抽象工厂模式[GHJV95]来创建特定“视觉和感觉”的用户接口对象，并且在ET++ 框架中[WGM88]跨越不同的窗口系统来获得可移植性。Unidraw [VL90]应用命令模式[GHJV95]实现可撤销的命令。

从应用程序框架的角度来看，模式可以被视为它们的构造块。从模式的角度来看，应用程序框架可以被视为给定应用领域中完整软件系统的一个模式。

6.3 软件体系结构启用技术

软件的构造基于几个基本原理。因为随着时间的推移有关原理已经变得模糊，所以我们称这些原理为*启用技术*（*enabling techniques*）。技术的发展已经能够实现这些已被广泛接受的原理，在某种程度上说区分原理和技术之间的差别正在变得越来越困难。因此我们采取简单的方法，将两个术语作为同义词使用。

所有启用技术都独立于具体的软件开发方法，并且它们中的大多数在许多年以前就已经为人所知。它们主要是在20世纪70年代关于结构化程序设计的出版物中被发展和提出的。经典的参考文献是由Parnas和他的同事合作的论文——见[Par79]和[PCW85]。尽管启用技术的重要性已经被认识很久了，但是它们对于成功软件开发的重要性只是在最近几年得到了提高，它们正与软件体系结构的新兴学科强有力地链接在一起。软件体系结构的模式明确建立在这些原理之上，有许多模式尤其集中在一个特殊的原理上。下面的小节归纳了软件体系结构的一些最重要的启用技术：

397

- 抽象（Abstraction）
- 封装（Encapsulation）
- 信息隐藏（Information Hiding）
- 模块化（Modularization）
- 事务分离（Separation of Concerns）
- 耦合和内聚（Coupling and Cohesion）
- 充分性、完整性和原始性（Sufficiency, Completeness and Primitiveness）
- 策略和实现的分离（Separation of Policy and Implementation）
- 接口和实现的分离（Separation of Interface and Implementation）
- 单一引用点（Single Point of Reference）
- 分而治之（Divide-and-Conquer）

6.3.1 抽象

抽象是人们处理复杂问题的基本原理之一。Grady Booch 把抽象定义为“对象的基本特性，这种特性将对象和所有其他类型的对象区分开，并因此提供清晰定义的相对于观众观点的概念性边界”[Boo94 ^]。用“组件”替换“对象”可以获得抽象更普通的定义。抽象有几种存在形式，如实体抽象、动作抽象、虚拟机抽象和偶然性抽象[SS86 ^]。针对这个原理的模式包含层模式和抽象工厂模式[GHJV95]等。

398

6.3.2 封装

封装将构成抽象结构和行为的抽象元素分组，并且把不同的抽象彼此分开。封装提供抽象之间的清晰界限。例如，转发器-接收器模式封装了进程间通信机制的实现细节。封装增强了像易修改性和可重用性这样的非功能属性。

6.3.3 信息隐藏

信息隐藏涉及对客户隐藏组件的实现细节，以便更好地处理系统复杂性并且使组件之间的耦合减少到最小。任何与客户正确使用组件无关的组件细节都应该被组件隐藏。整体-部分模式准确使用了这个原理。封装的原理经常被用作信息隐藏的方法。信息隐藏也可以使用接口和实现分离的原理，这些内容在本节内稍后描述。

然而，将哪些内容隐藏在组件中有时取决于应用程序。在一个应用程序中客户不需要知道的方面可能在另一个应用程序中要外在可视。例如，为了调整性能，在一个系统中直接访问组件的内部数据结构可能是必要的。当在其他的系统中，组件的性能已经足够使用时，这样的访问就不必要了。

映像的概念放松了信息隐藏的原理[Smi82]。为了给适应和变化提供更多的灵活性，映像模式以定义好的方法开放软件系统或组件的实现[Kee89]。不过，信息隐藏仍然是软件工程领域内基本和最重要的原理之一。

399

6.3.4 模块化

模块化涉及到对软件系统有意义的分解，并且将其分成子系统和组件。模块化的主要任务是决定如何从物理上将实体打包以形成应用程序的逻辑结构。模块化的主要目标是通过在程序中引入定义良好的且经过证实的边界来处理系统复杂性。模块的作用类似于应用程序功能或职责的物理容器。模块化与封装原理密切相关。解决模块性的模式例子包括层模式、管道和过滤器模式以及整体-部分模式。

6.3.5 事务分离

在软件系统内不同的或无关的责任应该彼此分离，例如把它们放到不同的组件中。用来解决某个具体任务的协作组件应该与涉及其他任务计算的组件分离。如果一个组件在不同的情况下扮演不同的角色，这些角色在组件内应该是相互独立且彼此分离的。在我们的模式系统中，几乎每一个模式都以某种方式应用了这一基本原理。例如，模型-视图-控制器模式分离了内部模型、对用户的表示和输入处理等事务。

6.3.6 耦合和内聚

耦合和内聚最初是作为结构化设计方法的一部分引进到原理中的。耦合专注于模块交互方面，而内聚则强调模块内部的特性。

耦合是为了测量从一个模块连接到另一个模块所建立的关联强度而制定的度量标准。强耦合使一个系统错综复杂，因为当一个模块与其他模块紧密相连时它会难以理解、改变或更正。可以通过设计系统模块之间的弱耦合来降低复杂度。

400

内聚是为了测量单个模块内功能和元素之间的连接程度而制定的度量标题。内聚有好几种形式。最好的形式是功能内聚，其中模块或组件的元素“一同工作以提供一些界定良好的行为”[Boo94]。最差的形式是偶然性，其中完全无关的抽象被归入相同的模块。其他类型内聚性——

逻辑性内聚、暂时性内聚、过程性内聚、通信性内聚、顺序性内聚和非正式性内聚性——在[Ba185]中都有描述。

我们的所有用于在组件间组织通信的设计模式都使用这个原理，例如客户机-分配器-服务器模式和出版者-订阅者模式。

6.3.7 充分性、完整性和原始性

文献[Boo94 ^]规定“软件系统的每个组件应该是充分、完整和原始的”。“充分”意味着组件应该捕获抽象的那些特性，它们对于允许与组件进行有意义的且有效率的交互是必要的。“完整”意味着组件应该捕获其抽象的全部相关特性。对于“原始性”，Booch认为组件能够执行的所有操作都应该是容易实现的。对解决给定问题来说，充分性和完全性是每种模式的主要目标。很多模式也是相对原始和容易实现的，例如策略模式[GHJV95]。

6.3.8 策略和实现的分离

软件系统的组件应该处理策略或实现，但并非要求一个组件同时处理这两方面的需求：

- 策略（*policy*）组件用于处理对语境敏感的决定，有关信息的语义和解释的知识，将许多脱节的计算汇集成一个结果或者选择参数值。
- 实现（*implementation*）组件用于处理一种说明非常完全的算法的执行，在这个算法中不需要作出对语境敏感的决定。语境和解释是外部的，通常通过变元提供给组件。

401

因为它们与语境无关，所以纯粹的实现组件易于重用和维护，而策略组件经常是与具体应用相关并服从于变化的。

如果在一个软件体系结构内不能把策略和实现分成不同的组件，那么至少应该在组件中对策略和实现的功能特性进行分离。策略模式[GHJV95]就是针对这个原理。

6.3.9 接口和实现的分离

任何组件都应该由两部分组成：

- 一是接口部分，它定义了由组件提供的功能特性并说明怎样使用它。组件的客户可访问这个接口。这种类型的输出接口通常由功能特征标记组成。
- 二是实现部分，它包括由组件为功能特性提供的实际代码。实现部分也可包括只在组件内部使用的附加功能和数据结构。实现部分不能被组件的客户访问。

这个原理的主要目标是保护使用组件的客户免受实现细节的困扰，并且只为客户提供组件接口的使用说明和指南。此外，这个原理还能让你独立于其他组件的使用来实现某个组件的功能特性。像封装一样，接口和实现的分离是实现信息隐藏的技术，该原理表明“客户只应该知道他需要知道的东西”。

402

接口和实现的分离也支持易修改性——如果组件接口与组件实现分离，那么组件的变更就会容易得多。分离防止变更直接影响客户。例如性能调整，在组件变更不需要调整组件接口的情况下，该原理尤其减轻了更改组件行为或表示的任务。接口和实现的分离可使用桥接模式

[GHJV95]来解决。

6.3.10 单一引用点

软件系统内的任何条目都应该仅仅被声明和定义一次。这个原理的主要目标是避免不一致问题。

由于某些软件系统的设计原理和实现方法，因此尽管许多编程语言(例如C++)[ES90]要求一个单独的定义点，但是实际上它们允许甚至要求有几个声明点。就C++而言，这主要是由于受到传统编译器和连接器技术的限制。对于程序员而言，其结果则是在人工维护一致性的过程中增加了工作量。

6.3.11 分而治之

这个既来自于古代政治学又来自于组合算法(例如合并-分类)的原理是很有名的。我们多次在软件体系结构中使用这个原理。例如，从上至下的设计中，把一项任务或者组件分成可被独立设计的更小部分。整体-部分模式在模式级实现这种技术。尽管这种技术比通用的整体-部分技术更明确，但是其他的模式还是倾向于采用这种细分技术。例如，微核模式细分那种可能曾经是单一代码块的代码。分而治之也经常提供一种实现事务分离的方法。

6.3.12 小结

列表中的原理可以更进一步扩展，例如包括由Trygve Reenskaug[Ree92]提议的适合面向对象的软件开发总则。但是，它们基本上是本节中提出的原理的变体。

403

注意到并不是所有总则都是互相补充的——有一些是对立的，这一点很重要。这方面的例子是接口和实现的分离原理以及单点引用的原理。第一个原理——当使用传统技术时——实现一个特定的功能至少需要两个引用点，一个在组件的接口部分，另一个在组件的实现部分。这与单点引用的原理的严格解释相矛盾。当然，由实现产生接口可能是一个解决办法，并且这种办法已经使用在更现代的方法中。

其他原理紧密联系，例如抽象和封装。在一个软件系统内，一个特定实体的适当抽象要求封装全部元素，而这些元素在单个组件或模块里组成它的结构。

6.4 软件体系结构的非功能属性

软件系统的非功能属性对其开发和维护、总体可操作性以及它对计算机资源的使用有很大影响。就像系统的功能属性一样，它们对应用和应用体系结构的质量有着同样的影响。软件系统越大越复杂且寿命越长，它的非功能属性就越重要。软件体系结构的模式明确考虑了这些非功能方面。

在本节中，我们讨论与模式有关的软件体系结构的一些最重要的非功能属性：

- 易修改性（changeability）
- 互操作性（interoperability）

- 效率（efficiency）
- 可靠性（reliability）
- 可测试性（testability）
- 可重用性（reusability）

404

6.4.1 易修改性

大型的工业软件和商业软件系统通常有较长的寿命，有时20 年或者更长。很多这样的应用程序在最初的开发阶段之后并非保持不变——它们在使用的过程中还在不断地被改进。现有需求要更改，新的应用要增加进来。在更改一个应用程序时，为了降低维护的开销和工作量，为了修改和演化方面的考虑而设计体系结构是非常重要的。

Parnas在文献[Par94]中非常生动地描述了软件的老化:

> 程序，像人一样，会变老。我们不能够阻止老化，但我们可以弄清引起老化的原因，采取措施来限制它的影响，暂时性地修复由老化引起的损害，并为软件最终不再可用做好准备。

他列举了软件变老的两个原因：

- 没有升级——如果软件没有经常更新，它会变老。
- 无知的胡乱更改——一些不理解最初计划的人对软件进行的胡乱更改会逐渐破坏了软件的体系结构。

在另一本书中， Parnas 增加了两个更进一步的原因：

- 软件从开始就是不灵活的。
- 由于开发文档的缺乏，随着时间的推移，造成对软件系统产生错误的理解。

正如文献[Par94]所描绘的那样，软件老化所带来的开销，仅仅通过引入新的特色、降低性能和可靠性的方式来跟上市场发展的步伐已经越来越行不通了。这些情况可以通过采用准确的文档，在引入更改时保留体系结构，认真评审和为更改预留空间的方式来防止。

我们认为易修改性包含4个方面：

- 可维护性。这主要体现在问题的修复上：在错误发生后“修复”软件系统。为可维护性做好准备的软件体系结构往往能做局部性的修改并能使对其他组件的负面影响最小化。

405

- 可扩展性。这一点关注的是使用新特性来扩展软件系统，以及使用改进版本来替换组件并删除不需要或不必要的特性和组件。为了实现可扩展性，软件系统需要松散耦合的组件。其目标是实现一种结构，它能使你在不影响组件客户的情况下替换组件。支持把新组件集成到现有的体系结构中也是必要的。
- 结构重组。这一点处理的是重新组织软件系统的组件及组件间的关系，例如通过将组件移动到一个不同的子系统而改变它的位置。为了支持结构重组，软件系统需要精心设计组件之间的关系。理想情况下，它们允许你在不影响实现的主体部分的情况下灵活地配置组件。
- 可移植性。它使软件系统适用于多种硬件平台、用户界面、操作系统、编程语言或编译器。为了实现可移植，需要按照硬件无关的方式组织软件系统，其他软件系统和环境被提取出

来放到特定的组件（如系统和用户界面库）中。

这种为变更而设计的软件系统也支持为不同的用户提供不同的结构，它比没有考虑变更设计的软件系统要好。许多模式针对易修改性，例如映像模式和桥接模式[GHJV95]。

最后，针对变更设计说一句告诫的话。随着模式使用的增加，我们已经看见有人做得太过分了。类不再简单。代码的每“大块”都非常灵活并且能适合许多不同的环境。这样的灵活性，无论如何都是要付出很大代价的。灵活的软件经常会使用间接方式或者不断增加的存储消耗来消耗更多的资源。它也在编码过程中需要更多的思考和更多的工作。好的设计者因此试着提前确定软件中哪些部分需要非常灵活以处理可预知的变化，哪些部分保持相对稳定。如果经过证明，它们是错误的，仍然可以通过仔细地重构系统的某些部分来引入另外的灵活性，或通过使用支持变更设计的模式来完成。这方法相比从一开始就考虑整体易修改性的工程技术更加经济实惠。

406

6.4.2 互操作性

作为系统组成部分的软件不是独立存在的。它经常与其他系统或自身环境相互作用。为了支持互操作性，软件体系结构必须为外部可视的功能特性和数据结构提供精心设计的软件入口。程序和用其他编程语言编写的软件系统的交互作用就是互操作性的问题，这种互操作性也影响应用的软件体系结构。代理者模式可能是模式解决互操作性问题的最明显的例子。

6.4.3 效率

效率为软件的执行而处理可获得资源的使用问题，并考虑它是如何影响响应时间、吞吐率和存储开销的。效率不仅仅是使用复杂算法的问题。对组件及其耦合来说，恰当地分配责任是在给定的应用程序中为提高效率而执行的重要的体系结构活动。

效率在分布式软件系统中也扮演着重要角色。IPC(进程间通信) 机制奠定了分布式应用的基础，它必须以极快的速度转发消息和数据。像转发器-接收器这样的模式重点处理效率问题。然而，由于许多模式引入了解决问题的间接附加层，因此最终结果可能会降低效率而非提高效率。

407

6.4.4 可靠性

可靠性是软件系统在应用或系统错误面前，在意外或错误使用的情况下维持软件系统的功能特性的基本能力。可靠性可以分为两个方面：

- 容错。其目的是在错误发生时确保系统正确的行为，并进行内部“修复”。例如在一个分布式软件系统中失去了一个与远程组件的连接，接下来恢复了连接。在修复这样的错误之后，软件系统可以重新或重复执行进程间的操作直到错误再次发生。
- 健壮性。这里说的是保护应用程序不受错误使用和错误输入的影响，在遇到意外错误事件时确保应用系统处于已经定义好的状态。值得注意的是，和容错相比，健壮性并不是说在错误发生时软件可以继续运行——它只能保证软件按照某种已经定义好的方式终止执行。

软件体系结构对软件系统的可靠性有巨大的影响。例如软件体系结构采用这样的方式支持

可靠性：在应用程序内部包含冗余，或集成监控组件和异常处理。主控-从属模式提供了模式如何支持可靠性的特定方面的实例。

6.4.5 可测试性

随着软件系统越来越庞大并且越来越复杂，尤其是一些工业软件，测试工作变得越来越困难和越来越昂贵。软件系统需要从其体系结构上得到支持以减轻对其正确性的评估——因为在大多数情况下正确性的验证仍然达不到要求。支持可测试性的软件结构可以更好地进行错误检测和修复，也可以临时性地集成正在调试的代码和正在调试的组件。

尽管我们所描述的模式还不能进行精确的测试，但它们对软件系统的可测试性有着巨大的影响。例如，命令处理器模式通过记录日志和重现用户命令对象促进了用户交互层次的可测试性。代理者模式使分布式系统中单独的客户机和服务器组件的测试变得更加容易。这种体系结构将组件从它们的通信伙伴和它们使用的通信机制中释放出来。 408

然而，代理者模式使得客户机和服务器之间的协同测试更为复杂，因为它需要引进一些附加组件来支持它们的独立性。与客户机和服务器的强耦合实现相比，调试由客户机向服务器发送消息中的错误将更为困难。这主要是因为其他一些组件涉及到列集和散集数据以及穿越进程边界发送消息。

6.4.6 可重用性

可重用性是目前软件工程领域中讨论最热门的话题之一。它号称能减少开发软件系统的费用和时间，并且能够开发出更高质量的软件 [Kar95]。Adele Goldberg曾经把重用定义为“在现有基础上实现想达到目标的行为“[Go191]。可重用性有两个主要方面——使用重用进行软件开发和为重用进行软件开发：

- *使用重用进行软件开发*是指重用现有的组件和来自以前项目或商业库、设计分析、设计说明或代码组件的结果。这些可重用的人工制品将稍做修改或不做任何修改集成到正在开发的应用程序中去。使用重用进行软件开发要求软件体系结构的构造允许“插入”预制的结构和代码组件。使用重用进行软件开发的目的是支持软件组装，也就是说对现有组件进行适当调整以适应开发要求，并实现“胶水”组件将它们连接起来构成应用系统。
- *为重用进行软件开发*的重点集中在产生那些既是目前软件开发的一个组成部分，又有可能在未来项目中重用的组件。这要求软件体系结构允许自包含部件，这样不仅正在开发的应用中可用而且在其他系统内重用时不需要做很大的修改。 409

尽管模式不能明确地解决可重用性的问题，但是几乎每种支持易修改性的模式也支持可重用性。例如，模型-视图-控制器模式就支持视图和控制器的交换和模型的可重用性。

一些非功能属性的实现需要相似的体系结构技术，例如设计可重用性和易修改性。其他的非功能属性服务于一个相似的总体目标：例如，设计可移植性和互操作性是为了实现软件系统与环境的集成，而可靠性和效率是为了实现它的普遍可用性[Ba185]。

非功能属性可能在相互补充的同时又彼此矛盾。例如，当复制应用程序的功能特性以实现

容错性时，最后得到的结构与没有冗余的结构相比，通常是低效的和昂贵的。当为软件体系结构定义非功能需求时，需要仔细考虑它们之间彼此依赖又相互独立的关系。同时你也需要列出不同非功能需求之间的优先级清单，定义在发生冲突时优先考虑的需求。

虽然非功能属性在软件体系结构中非常重要，但是它们的完成却是难以度量的。只为一些这样的属性，例如可重用性和易修改性，就已经具体指定了一个软件体系结构必须满足的详细标准[Kar95]。因此，估算软件体系结构实现一个给定非功能属性的程度仍然主要依赖于软件工程师的经验。

410

6.5 总结

模式很好地适应了软件体系结构的现有方法：

- 它们明确建立在用来构造定义良好的软件系统的启用技术（如信息隐藏和接口和实现的分离）的基础上。
- 它们强调非功能属性（如易修改性和可靠性）的重要性。
- 它们通过使用用来解决递归设计和实现问题的指南，来补充现有的与问题无关的软件开发过程和方法。

模式为你从软件体系结构中获得好处同样作出了重要贡献：

- 它们帮助认识公共范例，使得软件系统之间的高层关系可以被理解，使得新应用可以被作为变体建立在老系统 的基础之上。
- 它们为正在开发的软件系统寻找合适的体系结构提供支持。
- 它们为在设计选择方案中作出原则性选择提供支持。
- 它们帮助分析和描述复杂软件系统的高层属性。
- 它们为软件系统的变更和演化提供支持。

模式在支持系统性地构造带有已定义的功能属性和非功能属性的高质量软件系统方面向前迈进了一大步。模式还提供了一种实用的与方法和进程无关的方式，用来解决软件开发者每天面对的很多设计和实现问题。

411

第7章 模式团体

每一个伟大的运动都必然会经历三个阶段：嘲笑、讨论、采用。

——*John Stuart Mill*

许多软件开发者用文档形式记录下他们熟悉的模式并与世界范围内的同行共享他们的研究成果。他们组织在一起形成了一个团体，分享对软件模式的共同兴趣。但是其体由谁组成了这个模式团体呢？它来自哪里？它的领导人物又是谁？

这一章给出了模式团体内“名人录”的概述。 413

7.1 起源

建筑大师Christopher Alexander为今天大多数模式方法的构建打下了基础。他和位于加利福尼亚州伯克利的环境结构中心的成员，耗费20多年的时间开发了一套使用模式构造建筑的方法。这种“建筑和设计中的全新态度”出现在已出版的丛书[Ale79]、[AIS77]、[ASAIA75]和[ANAK87]中。Alexander描述了覆盖各种规模和抽象的250多个模式，从构筑城镇和社区到铺设道路和装修房间。他也为描述模式定义了基本的“语境-问题-解决方案”结构，称为“Alexander形式”。最近，一些模式作者已经开始使自己稍微远离Alexander，因为他们感到他对模式的观点不能直接翻译到软件模式中。他们承认Alexander工作的重要性，但是他们想要走自己的路。抛开这个讨论不谈，无论如何，Alexander的著作值得每个对模式感兴趣的人阅读。

软件开发领域中的模式先驱者是Ward Cunningham和Kent Beck。他们阅读Alexander的书并深受其中思想的鼓舞，因而致力于把他的想法引入到软件开发中。Ward 和 Kent 的最初5个模式处理的是用户接口设计——他们的模式每个任务的窗口、少数方格、标准方格、名词和动词以及短菜单标志着模式在软件工程中的诞生[Cope95]。自从这些模式发表以来，Ward和Kent已经写了很多模式。Ward在开发商业系统主要是会计应用程序方面获得了经验。用于信息集成的CHECKS模式语言[Cun94]是这项工作的一个结果。Kent则将注意力集中在Smalltalk惯用法上。他的模式将作为丛书出版，第1卷《*Smalltalk Best Practice Patterns, Volnme1: Coding*》[Bec96]正准备发行。Kent也是“*Smalltalk Report*”杂志中Smalltalk惯用法栏目的固定专栏作家。

第一本正式出版的关于在软件开发过程中使用模式的著作是Erich Gamma于1991年所写的博士论文[Gam91]。这部著作用德语写成，因此在中欧之外没有取得很多承认。在同行中，Erich首先描述了怎样以简洁的方式使用面向对象机制来解决在应用框架开发过程中遇到的典型设计问题。在他的论文里你可以找到文献[GHJV95]中描述的大约一半模式的早期版本。 414

7.2 领军人物和他们的著作

四位软件设计专家——被称为模式团体的“四人帮”（Gang-of-Four）——他们为模式在软件工程中获得广泛承认铺平了道路。Erich Gamma、Richard Helm、Ralph Johnson和John Vlissides是模式方面的开山之作《*Design Patterns—Elements of Reusable Object-Oriented Software*》的作者[GHJV95]。尽管我们最初独立地收集自己的模式，但是我们的模式还是经常建造在“四人帮”的模式之上，与此同时编辑最初的“四人帮”目录的工作也在进行。我们也与“四人帮”共享我们对于模式的总体观点，例如关于模式系统与模式语言。我们采用和他们相似的方式描述我们的模式，并且力图把他们的模式集成到我们的模式系统中。

James O. Coplien是另一位模式方面的主要专家。在1991年他出版了受到业界广泛认同的C++教材《*Advanced C++ Programming Styles and Idioms*》[Cope92]。虽然他没有使用术语“模式”，也没有用模式形式来描述他的思想，但是他是一位惯用法尤其是C++惯用法的先驱者。他目前正致力于模式方面的工作，他的模式主要针对组织和软件开发项目的构建，以及人员在其中的作用[PLoP94]。他最近和John Vlissides在“*C++Report*”杂志上开设了一个有关模式的专栏。

Douglas C. Schmidt是模式团体中另一位值得注意的人物。几年以前，他还是一个博士生的时候，他就开始着手做自适应的通信环境（Adaptive Communication Environment, ACE）框架。ACE支持分布式应用的构造[Sch96]。他是许多模式的作者，主要是关于分布式和高速连网的主题[Sch94][Sch95]。Douglas的模式被广泛应用在很多工业通信软件系统中。

415

Robert Martin描述适合在C++中使用的模式。他的模式可被划分到介于设计模式和惯用法之间的某个类别中[PLoP94]。他从他所开发的应用程序中抽取这些模式，但是并不需要现存模式活动的预先知识——他仅仅知道对他正在解决的问题而言，他的这些模式代表好的解决方法。

Peter Coad也致力于模式方面的工作，最近他出版了一本包含了他的研究成果的著作[Coad95]。书中包含了大约200种模式，其中绝大多数模式都是试图帮助分析一个给定的应用领域并且使用面向对象技术建造应用程序。他的某些模式也属于我们的设计模式范畴。Peter Coad是早先公开提出模式主题的人之一[Coad92]。

Wolfgang Pree为框架开发寻找设计模式的结构原则[Pree94]。Wolfgang将结构原则分为7种所谓的“元模式”。他对设计模式的观点主要集中在适用于框架开发的结构原则上，而不是集中在有助于解决特定设计和实现问题的具体解决方案上。

从上面的叙述中可以了解到很多关于模式的著作都是可以获得的。由于版面的问题，有很多关于模式的出版物在此没有列举。在不久的将来会有更多的论文、会议录、特别发行的各类期刊杂志和著作出现。

7.3 团体

我们以及所有在本章中提及的人们正在努力致力于研究和使用模式。许多来自世界各地的软件工程师正在用模式记录下他们的经验并与他人分享这份财富。与他们分享我们的模式对我

们来说既有帮助又充满乐趣。

这个模式团体最近建立了自己的论坛，程序设计的模式语言（Pattern Languages of Programming, PLoP）会议。它的会议录作为一套丛书出版。PLoP'94[PLoP94]和PLoP'95[PLoP95]会议录已经出版。PLoP在欧洲还有一个分支机构——EuroPLoP，它的会议录也将作为丛书的一部分。PLoP 和EuroPLoP在以下几个方面不同于其他会议： 416

- 注重可行性。会议寻找已被证明的能够解决具体问题的方法的模式描述，而不是提出最新的科学成果。
- 勇于抛弃初创性。模式作者不必是他们所描述解决方案的原初的开发者。
- 非匿名的评论。"引导"提议而不是评论提议。"引导人"与提交论文的作者接触并与他们讨论提议。其目的是改进论文，使得它可以在会议评论时被接受并且尽可能少遭受拒绝。
- 作者研讨会代替呈文。所有模式都经过由会议参加者组成的作者研讨会的讨论，而不是在开放论坛里由他们的作者提出。
- 仔细编辑。作者有机会从作者研讨会获得反馈，并且所有模式在出现在最终会议录中以前，都已经被反复修改过。

为了讨论模式和有关模式的问题，模式团体提供几个邮件列表和一个万维网页。模式主页的URL 是：

http://www.hillside.net/patterns/

这个网页提供了即将来临的模式事件和关于模式的现有书籍，并提供了相关的链接，链接到其他的Web网页，例如在http: //c2.com/ppr的波特兰模式仓库，由Ward Cunningham维护。

关于模式还有几个因特网邮件列表。例如，patterns@cs.uiuc.edu讨论人们希望共享的具体模式，patterns-discussion@cs.uiuc.edu讨论与模式有关的问题，例如"模式是什么？"和"怎样描述模式？"。还有几个其他相关的邮件列表，其中一个讨论"四人帮"模式，一个讨论我们的模式。你可以在模式主页上找到关于可使用的邮件列表和怎样订阅它们的详情。 417

模式团体的非官方指导委员会是Hillside公司，即大家所熟知的"山腰组"。 Hillside公司是一个非赢利组织，其中的成员包括Ward Cunningham和Kent Beck、"四人帮"、Grady Booch和James O. Coplien。山腰组的主要目标是在软件开发过程中传播模式的使用、领导模式团体，并且对软件工程这个新学科中的新来者给予支持。山腰组的"精神之父"是Kent Beck。山腰组也是PLoP 和EuroPLoP会议的组织者和发起者。

如你所见，这里有世界范围内规模较大的模式团体，许多在软件工程和软件体系结构方面的领军人物是它的成员。模式团体的大多数成员在软件产业工作，并且是在设计和建造大规模应用程序过程中有经验的软件开发者。学术界的成员多数也参与了工业项目——他们不仅要教怎样建造软件系统，而且他们也要亲自干。通过参加模式团体你能够利用所有这些经验，从那些已经被证明、随时可以实际应用的模式中获得经验。你也可以通过写你自己的模式与其他专家分享自己在软件开发过程中的经验。

模式团体是计算机科学领域的惟一一个基于为了描述已经证实的知识而对文字形式、模式形式感兴趣的团体。它将具有不同背景和专业领域的人带到了一起。最有趣的是，模式形式使

得讨论并与其他领域专家分享这样的知识成为可能，甚至是与软件工程领域内的新来者和初学者分享这样的知识都是可能的。

如果你不是模式团体的一份子，我们邀请你参加这个团体。访问模式主页、订阅模式邮件列表、翻看各种各样的模式书籍、参加PLoP 或者EuroPLoP 会议、获取你自己类似模式的经验并与来自全世界的专家分享这些信息。你将肯定会受到很多积极的“啊哈！”的影响。

418

第8章 模式将走向何方

这是星际飞船的航行计划。它的5年任务是：探索奇异的新世界。寻找新的生命和新的文化。在没有人到过的地方，勇往直前……

——星际旅行：首卷

© Paramount Pictures 1966-1968

在写作本书期间，模式是每个人思维中最活跃的部分。人们狂热地谈论模式和模式将给软件开发带来的好处。但是模式将走向何方？未来研究的目标是什么？

本章描述我们对模式未来的看法。

8.1 模式采掘

虽然已经可以为很多领域提供大量在广度和深度上得以抽象的模式，但是采掘新模式仍是未来一项重要的工作。

8.1.1 软件体系结构的模式

一般来说，像面向对象设计、用户接口编程和分布式计算这样几个软件的特殊领域都可以使用多种不同的模式来很好地描述。然而，其他领域还没有被模式所覆盖，或仅被一小部分模式所覆盖。例如安全和事务处理系统、并行和科学计算及容错。填补这些空白将是未来的重要工作。

我们认为模式是一种智力工具，一些有经验的开发者建议首先寻找那些不能直接适合正在设计的应用领域的模式。在有些时候，有可能归纳出模式的关键思想并把它传递到另一个领域，这将导致一个新模式或原件变体的出现。

由于使用普通编程语言来捕获经验，因此惯用法将是另一项重要的工作。今天，只有在Smalltalk和C++语言中才有足够可用的惯用法。填补Pascal或C语言中的这个缺口将有助于程序员更有效地使用这些语言。

一项激动人心和得到广泛承认的活动是为Java编写惯用法。很多软件开发专家将这种相对新的编程语言吹捧为未来的语言。然而，在为Java编写惯用法之前，必须首先学习Java，而且理解它的细节并不是那么容易的。用来反映正在不断增长的使用Java的编程经验的惯用法将为想要有效学习它的正确用法的所有人提供极大的帮助。这样的惯用法将形成一门极好的教学课程，帮助开发者避免跌入Java的陷阱。

除使用一种程序设计语言以外，程序员的生产力依赖于使用库、框架(如微软公司的微软基础类库)或所谓的“中间件”平台(如对象代理者)。有效理解和使用这些平台可以并且应该能被

合适的模式集合所支持。它们中的许多并不在今天的出版物里存在。但是，当模式的优点正在为越来越多的开发者所了解时，我们希望看见这种从实际经验中得出的模式集合。将来的框架文档可能会包含用来描述怎样有效地使用这种框架的模式。

8.1.2 组织模式

模式早已覆盖软件开发的各个方面，而不仅仅限于设计和实现。一个例子就是James O. Coplien提出的组织模式的汇集[Cope94b]。它们描述了怎样为软件开发项目的管理构建组织和项目以提供合适的支持。

组织模式的例子是设计师控制产品[Cope94b]。它针对由多个人设计一个产品缺乏简洁性和内聚性的事实。这种模式说明在较大的工程项目里你应该建立一个设计师角色。设计师应该建议并且控制开发者，并与他们保持紧密的联系，同时保持与用户的紧密联系。

其他领域，如怎样组织需求分析，还没有被很多模式所覆盖。对这样的活动进行模式采掘有助于使整个软件开发过程更有效和更多产。

8.1.3 特定领域模式

应用领域(例如电信)是模式的一个潜在的广阔领域。具体的领域知识正逐渐以模式形式被记载。这样的模式获取领域的结构，即它的组成实体、它们的关系，以及非常重要的一点——怎样组织工作。例如，在AT&T的开发人员已经开始为在电信领域方面的交换系统收集模式。他们开发了100多种模式，其中8种在文献[PLoP95]中公布。另一个例子是用于公布和讨论商业应用中模式的因特网邮件列表(businesspatterns@cs.uiuc.edu)。其他特定领域的模式是为工厂自动化、仓库管理、会计、医疗保健和电信网络管理而编写的。

421

但是，大多数针对具体领域的模式是秘密的——它们表现公司怎样建立特殊种类应用的知识和经验，参考它们是不可能的。但我们相信随着时间的推移，越来越多的知识将变得公开。在很长的一段时期内，分享经验通常比努力保守秘密更有效。

8.1.4 模式语言

完整模式语言的开发是一个乐观而且值得做的目标。这样的语言能为不同领域内出现的所有设计问题提供解决方案。Christopher Alexander 声称已经在建筑领域进行了这样的工作[AIS77]。模式语言已经在小的软件设计子领域中存在，例如描述信息集成的CHECKS模式语言[Cun94]。看到模式团体沿着这条道路走了很远是一件令人激动的事情。

即使我们不能在严格的意义下达到完整性，覆盖不同领域设计空间的实质部分也将有利于模式语言的发展。例如，在文献[GHJV95]中提到的“四人帮”模式被认为包含了相当数量——可能有一半——的通用设计模式，这些模式出现在面向对象的设计过程中少数合作类的粒度级上。

422

8.2 模式组织与索引

今天大多数的工作集中在开发模式和模式语言上。经过近几年的发展，模式团体已经为软

件体系结构、设计和实现创建了大范围的模式。本书中谈到的模式包括可得到的模式，以使很多在邮件列表上和在PLoP 会议上讨论过的模式，这反映出已经有越来越多的需要用文档形式记录下的专门技术知识。

这个可得到的模式仓库增长得越快，以整体方式处理模式，寻找并使用某个特殊模式就越困难。因此我们需要一种合适的覆盖所有模式的组织方法。模式之间的关系必须明确，模式必须分类，相同模式的多种描述必须统——例如代理模式，它在“四人帮”模式系统中和在我们的系统中都存在。

我们希望我们在模式系统上的工作将为组织模式提供有用的起始点。另一个这样的起始点是Ward Cunningham的波特兰模式仓库。这为软件开发的不同方面提供模式语言，例如CHECKS [Cun94]和许多起源于Kent Beck [Bec94]的Smalltalk 程序设计模式。

模式团体在PLoP'95中采取了一种非常有趣的方法——“模式地图”。作者将他们写的模式与其他作者的相关模式相连接。他们在纸上写模式名，将纸片放在主会议室的地板上，用绳子将每个模式与相关模式连结起来。模式世界的第一幅图片就这样画好了，虽然这是用一种相当不正式、特别和不协调的方式来完成的。尽管如此，大约300个不同模式就这样被连接起来。

山腰组在试图对连接模式进行更严肃认真的尝试时使用这张地图作为输入。1996 年初在加拿大的一个山地旅馆，他们写了150多个所谓的“pattlet”——即模式摘要，这些模式摘要包括模式名、简要的问题描述、解决方案的关键思路和模式完整描述的参考。最重要的是，所有这些模式摘要使用几种不同的关系类型连接在一起。大多数模式摘要直接对应于我们在本书中定义的关系，例如细化关系。其他关系是新的，例如“对比”关系描述初一看很相似，但实际上完全不同的两种模式之间的差别。 423

所有山腰组的模式摘要都可在万维网上得到。令人遗憾的是，当我们写完这本书时，网页还没有做好。关于这种有趣的模式索引的具体细节，我们请您留意模式主页，你能在下面的地址找到这些内容：

http://st-www.cs.uiuc.edu/users/patterns/patterns.html

山腰组也为用新的模式摘要扩展它们的模式索引确定了一个程序。这允许你写自己的模式抽象，将它与其他的模式摘要连接并把它集成到索引中去。索引随时间而增长，伴随着每个新的模式摘要的增加，将画出一张更完整的模式世界图。

但是，尽管做了这么多有希望和有趣的工作，在我们能够建造可以有效支持高质量软件开发的真正成熟的模式系统之前，仍然有许多事情要做。我们需要使用模式的更具体的经验并对它们的组织方式进行更多的研究。

8.3 方法与工具

越来越多的人正在着手做模式工具。例子包括再工程工具SUS(Software Understanding System，软件理解系统)[THG94]，或软件开发环境FACE(Framework Adaptive Composition Environment，框架自适应组合环境)[ME96]。其他人正在为建造针对特殊模式的预制代码框架库 424

工作[Sou94]。所有这些方法的目标是为模式和模式使用自动化提供尽最多的CASE工具支持。工具制作的工作将在未来继续进行下去。支持使用模式的软件开发方法也在讨论中。它们的目标是指导软件开发者选择一个可在具体开发活动中使用的模式。其他工作集中在为选择、应用和结合模式以及将它们集成到现有软件体系结构中确定通用准则。

然而，很多有经验的软件开发者怀疑这些工具和方法的有效性。他们的第一个论点是如果你不理解模式本身，就没有方法和工具可以帮助你。第二点，他们争论说模式是智力积木并且故意留下空白，让开发者填上。每种模式必须经过调整以适应开发应用的需要。因此，一种模式的两个实现不太可能相同。你因而不能为模式提供羽翼丰满的预制代码，也不能使它的实例化完全自动化。

将几种模式合并成一个异型结构是更加复杂的。它不只是将不同模式的组件按照一种特殊的次序连接起来。你还需要经常将不同模式组件的责任合并到一个独单组件中，并且把模式组件的责任加到现有你设计的组件中。如果连接错误，由此产生的结构可能引入另外的复杂性并且失去每个独立模式支持的属性。最后，一种模式能否被使用取决于具体设计问题和与它们相关的强制条件。

总而言之，成功地使用模式仍然需要软件开发者的智力技能。我们相信精心设计的模式浏览器或万维网工具在帮助开发者寻找和使用模式时比曾经可能是一个完全集成的“模式支持”的软件开发环境更有效。

不论这场争论的结果如何，仍然有很多人确信模式工具和方法的有效性。在未来的日子里，人们将更进一步地讨论这样的工具和方法问题，毫无疑问，更多调查研究工作正在开展，更多的工具和方法将被开发出来。

425

8.4 算法、数据结构和模式

模式有助于捕获专家的现有知识并使用它来寻找软件设计中再现问题的解决方案。一个相似的目标曾导致对于基本算法和数据结构进行集中的查找。鉴于模式主要集中于体系结构的问题，算法和数据结构针对计算的问题(如查找和排序)。令人遗憾的是，软件开发者不得不既要找到一个合适的体系结构又要解决计算的问题。只有结合了模式的使用，抽象数据类型和算法才能帮助开发者解决他们的具体问题。

在模式和算法之间有一种双重的关系。一方面，当实例化一个特殊的模式时，我们必须实现它的参与者的所有服务和他们的协作。有些服务可能非常复杂。这时算法和数据结构开始发挥作用——它们为实现这样的服务提供方法。另一方面，设计模式和惯用法可以支持算法和数据结构的实例化。

我们期望未来的研究能进一步澄清模式和算法的组合用法。作为第一步，现有的算法和数据结构可以用模式形式描述。格式可从为描述模式而引进的格式中派生而来。这在与模式描述相比时需要进行一些修改和扩展。例如，需要增加关于复杂性分析的附加部分。其他部分（如结构）可以被修改或删除。

尽管如此，算法和数据结构对相似的描述图式仍然适应得很好。例如，在给定的语境中，两者都用来解决问题。相同的算法可能导致几种变体。像模式实例化一样，使用特殊的算法暗示着特定的效果。一个算法可以细化其他算法。把算法聚合成系统并把它们与具体的问题分类联系起来也可能是有用的。总起来说，算法描述展现的许多属性也可应用到模式描述中。我们希望新的算法目录出现，它将用一种统一和系统化的方式来描述算法和数据结构。 426

虽然你能用相同的图式来描述算法和模式，但是算法和模式是同一硬币的两个面——算法帮助解决计算的问题，而模式描述系统结构的元素。

8.5 形式化模式

学术界专门讨论如何使模式形式化。形式化的支持者争论说：它允许更准确的模式描述，特别是关于它们的结构、动态性和具体语义。今天，形式化模式能够比非形式化模式描述更好地支持模式工具的开发。因此我们期望在不久的将来可以了解在形式化模式中的许多工作。

但是，如同用工具和方法一样，很多软件开发者不同意这些论点。正是使问题描述形式化使得把一种模式与一个具体的设计问题相匹配变得很艰难，这是因为问题通常不是形式化的。使解决方案形式化使得抓住模式的关键思想并且创建有效的变体变得很艰难。形式化的解决方案可能不必要地缩小了模式的可应用性。相反，形式化可能使得模式太概括而不能使用。另外，我们不知道形式化是否适于描述模式的优点和不足。

所有这些方面对理解模式和决定模式是否有助于解决一个具体设计问题是非常重要的。相似的争论还存在于模式的实现准则中。程序员需要的是他们能理解的具体信息并将这些信息直接传递到他们自己的代码中，而不需要一个给人深刻印象的公式。模式是智力积木，它可能有无数个不同的具体实现。然而，形式主义仍然倾向于非常精确地描述具体问题，但是不允许这样一种变体，即它是本来就嵌入到每个模式中的。形式化方法在软件开发过程中有它们自己的位置——我们仅仅认为它们不适用于模式。 427

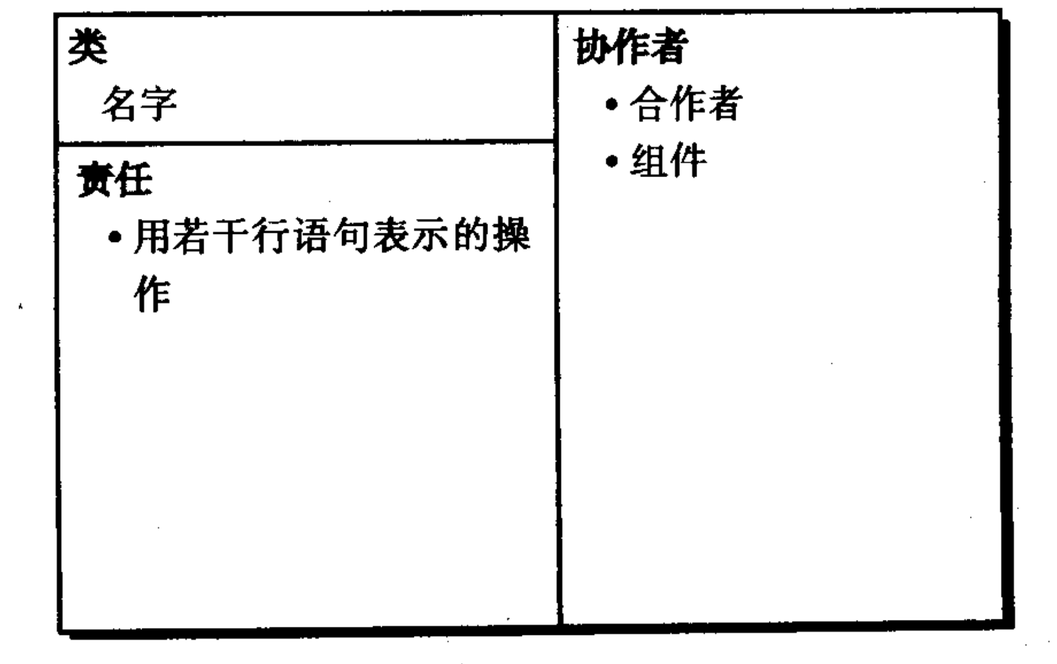
8.6 最后评述

模式揭示了专家们多年来获得的关于软件构造的知识。因此模式的所有工作都应该集中在使这个宝贵资源得到广泛应用。每个软件开发者在建造软件系统时都应该做到有效地使用模式。当这个目的达到时，我们就可以庆祝既在每个单个模式中又在所有模式中体现出的模式所反映的人类智能。 428

符　号

类-责任-协作者卡片

类-责任-协作者卡片（Class-Responsibility-Collaborator cards, 简称CRC卡片）[BeCu89]以一种非形式化的方式标识和指定应用的对象或组件——尤其是在软件开发的早期阶段。

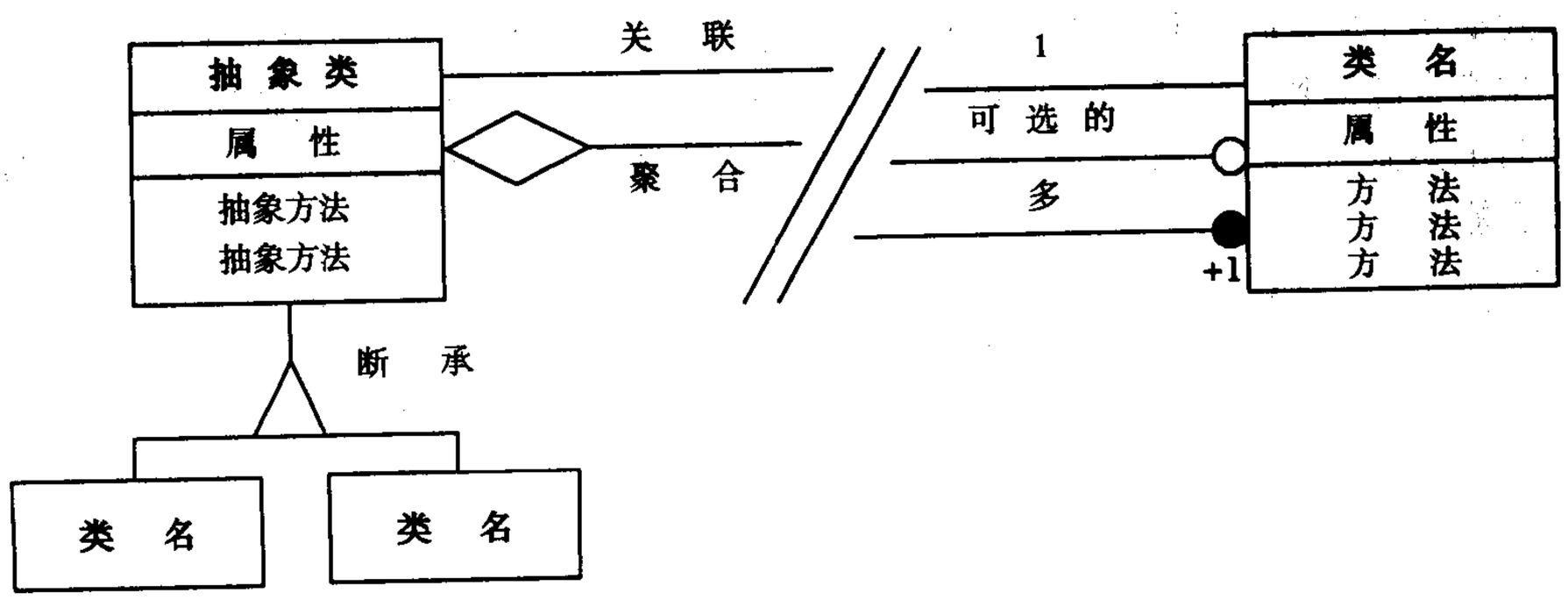


一张CRC卡片描述一个组件、一个对象或一个对象类。这张卡片由三部分组成，分别描述组件名字、它的责任以及其他协作组件的名字。术语“类”的用法是历史性的[Ree92]，我们也把CRC卡片用于其他种类的组件或单个的对象。

对象建模技术

对象建模技术（Object Modeling Technique, OMT）[RBPEL91]是一个广泛使用的面向对象的分析和设计方法。 OMT由对象模型、动态模型和功能模型这三个模型组成。我们只为对象模型引入这个符号，以表现相互作用组件的静态结构。对象模型描述了对象或类，它们的属性、方法和关系。我们也使用盒子表示针对其他组件的OMT中的类。OMT通过组件间的连线来表示组件之间的关联、聚合和继承关系。OMT对象模型的基本概念符号如下图所示：

429



组件

用矩形表示组件名及其可选的属性和操作。抽象组件和相应的抽象方法用斜体[⊖]表示。

方法

方法名被写在组件矩形中。它们表示组件的操作。我们用斜体[⊖]表示抽象方法，即那些仅为实现多态而提供接口的方法。

属性

属性名被写在组件矩形中。它们表示组件的数据槽。

关联

连接组件的线。关联可以是可选的(用空心圆表示)，也可以是多重的(用实心圆表示)。在关联末端的数字表示它的基数。除聚合和继承之外，组件的关联用来表示组件的任意关系。传递关系一般不画。

聚合

在关联线段的末端有一个菱形用来表示聚合关系，关联另一端的合作者组件包含在该组件内。

继承

在关联线段的中间有一个三角形用来表示继承关系，三角形的顶点指向超类。 430

对象消息序列图

消息序列图（Message Sequence Charts, MSC)是用于设计和指定当前操作实体间（如进程或硬件元素）协议的标准符号[GR92][GGR93]。MSC符号在电信领域进行了标准化并且集成到SDL语言中。它指定了给定范围内实体间的信号流。不过，我们并没有遵循SDL/MSC的标准符号，而是将MSC符号改写成适合于演示模式的参加者之间对象或组件的交互作用。我们称其为对象消息序列图（Object Message Sequence Charts, OMSC）符号。 431

对象

在OMSC中，对象或组件用矩形表示。用模式中组件的名字来标记这个矩形。在OMSC内发送或接收消息的对象，用垂直条与矩形的底部相连。

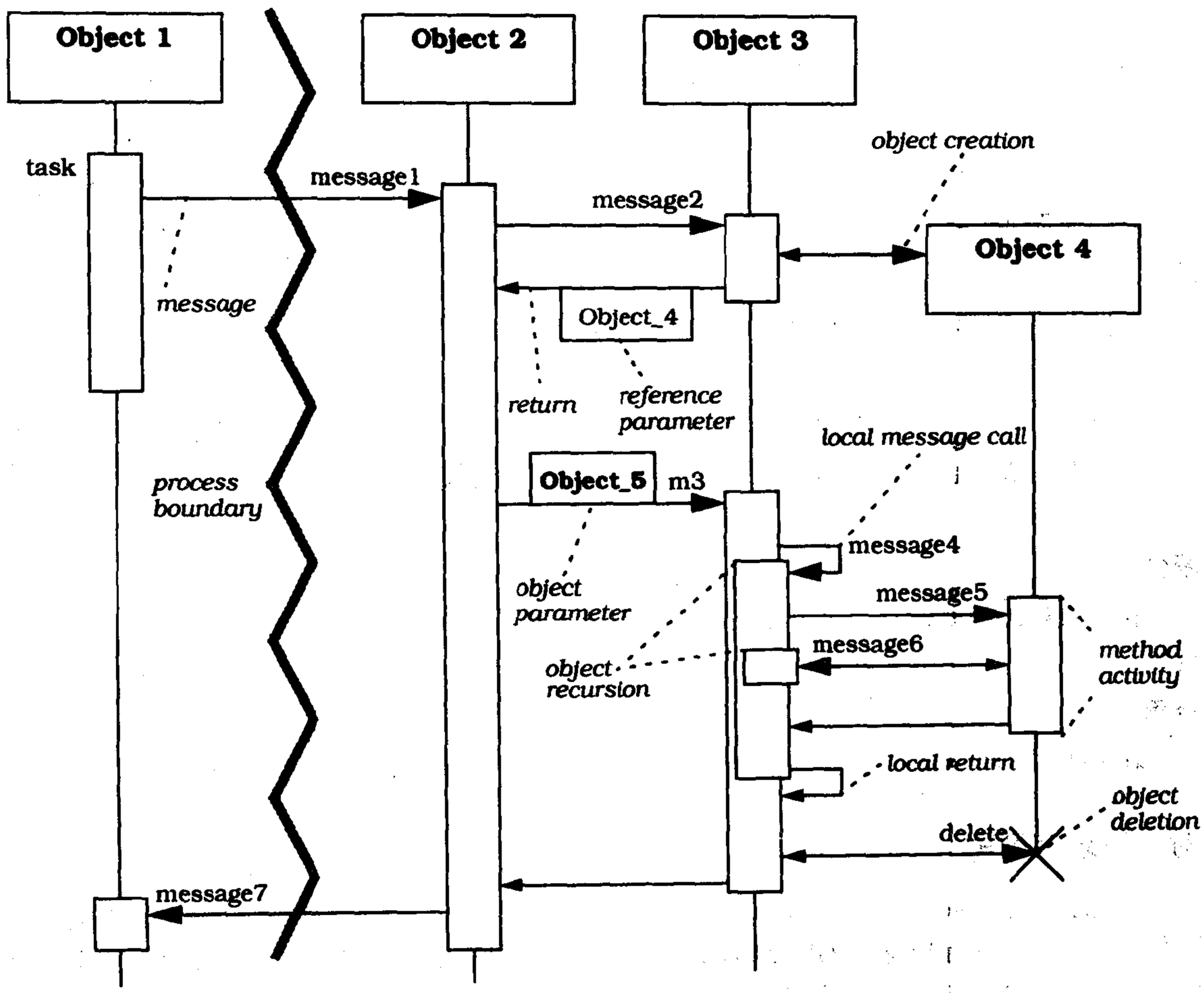
时间

时间从上向下流动。时间轴没有刻度。

消息

对象之间的消息用箭头表示。使用时，这些箭头用方法名在头部标记。为了表示控制流返回发送者，我们用小箭头扩展了标准MSC符号。如果一个激活的方法没有发送其他相关的消息，

⊖ 当保持英文原文时，用斜体表示；当翻译成中文时，用楷体表示。——编辑注



用一条有两种类型箭头的双箭头线段表示。

对象活动

为了表示用于执行一个特定函数、过程或方法的对象活动，在对象的垂直条上放一个矩形。对象可以给自己发送消息以激活其他方法。这种情况可用稍向右偏移的嵌套矩形来表示。

参数

当参数对于理解一个OMSC符号很必要的时候，它们才会被显式地指出。消息的参数用显示在箭头之上的矩形表示，而返回参数在返回箭头之下。如果沿箭头传送的是参数对象的责任，则对象的名字用**粗体**显示。如果仅是对象的引用作为参数传递，则它的名字用*斜体*[⊖]显示。

对象生存周期

在许多情况下我们假设所有相关的对象已经存在，并且对应的矩形已在OMSC的上方画好。

⊖ 当保持英文原文时，用斜体表示；当翻译成中文时，用楷体表示。——编辑注

如果OMSC显示对象创建，用没有标记的箭头指向OMSC中的矩形来表示。如果对象不再存在，用一个叉号结束对象垂直条来表示。这个符号对应着C++中的构造函数调用和析构函数调用。

地址空间

一条粗折线表示地址空间或进程边界。穿过这条边界的消息是通过IPC机制进行传递的。通常以异步方式对待这些消息，并且在发送和接收对象的同时继续处理消息。阻塞发送者直到远程过程返回的跨越进程边界的远程过程调用，是一个例外。

432

词 汇 表

词汇表包含了我们在本书中经常用到的许多术语。所有术语都与软件体系结构的特定方面有关。我们已经忽略了许多仅用于某一方面的术语，例如在黑板模式中从人工智能那里借用来的术语。当我们感到这样的术语需要解释时，我们在相关语境中定义它而不是将它包含在词汇表里。我们也忽略像“模式”、“软件体系结构”或者“惯用法”这样的术语，因为在书中我们已经使用了大量篇幅来解释它们。

Abstract Class（抽象类）

一种类，它不会实现在其接口中定义的所有方法。抽象类为其子类定义了公共抽象。

Abstract Component（抽象组件）

一种组件，它为其他组件指定一个接口。抽象组件不但可以像抽象类一样被明确地给出，而且可以通过使用与其他组件的接口被隐含地给出，例如一个C++模板函数的类参数。抽象组件是开发多态性和实现灵活系统的基础。为了避免将模式限制成面向对象的实现，这个术语的用法与抽象类相同。

Abstract Method（抽象方法）

一个接口，针对必须由子类定义的类操作。

API（应用程序接口）

软件平台的外部接口，例如操作系统，它被建造在它之上的系统或应用程序所使用。

Application（应用程序）

433 用来完成客户需求的一个程序或程序集。

Application Framework（应用框架）

一个特定领域中完整应用程序的框架。

Associative Array（相关数组）

通过任意关键值而不是整数进行索引的数组。哈希表演示实现相关数组的一种方式。

Class（类）

面向对象语言中的一个基本构造块。类指定并且封装它的内部数据结构及其实例或对象，的函数。通过继承，类的描述可以建造一个或更多其他类。

Client（客户机）

在我们的描述中，客户机是指利用其他部件提供的函数开发出的组件或子系统。

Collaborator（协作者）

与另一个组件合作的组件。CRC卡片的组成元素。

Component（组件）

软件系统的一个已封装好的部分。组件有一个接口，用于提供对其服务的访问。组件用作系统结构的构造块。在程序设计语言级，组件可以被表示为模块、类、对象或一组相关函数。一个没有实现其接口所有元素的组件被称为抽象组件。

Concrete Class（具体类）

一种类，它来自可以被实例化的对象。与抽象类相反，在具体类中实现了所有的方法。这个术语用来区分从抽象超类导出的具体类。

Concrete Component（具体组件）

一种组件，它实现了在其接口中定义的全部元素。与区分具体类与抽象类一样，它用于区分组件和仅定义其接口的抽象组件。

Container（容器）

对包含有许多元素的数据结构的通称。例如，列表、集合和数组都是容器。

CRC Card（CRC卡片）

类-责任-协作者卡片。一种设计工具和符号。我们也使用CRC卡片描述不是类的那些组件。 434

Demultiplexing（多路分解）

将引入的数据从输入端口发送到其对应的接收者的一种机制。在输入端口和接收者之间有一种1:N的关系。多路分解通常适用于引入事件和数据流。逆操作被称为多路复用。

Design（设计）

由软件开发者执行的活动，其结果是制定系统的软件体系结构。很多时候，这种活动使用术语“设计”作为这个活动的结果的名字。

Domain（领域）

表示与一个主题有关的概念、知识和其他术语。经常用“应用领域”来表示一个应用针对的问题领域。

Drag and Drop（拖放）

由现代的GUI支持的一种用户活动。拖放允许用户通过选中图形对象，并将其拖至屏幕上另一个地方来对图形对象进行操作。例如，打印文档可以通过选择文档并将其拖至打印机图标来完成。

Dynamic Binding（动态绑定）

直到运行时才将操作名称(消息)与相应代码(方法)关联起来的机制。它用来实现面向对象语言中的多态性。

Framework（框架）

想要对其进行实例化的半完成的软件(子)系统。框架定义(子)系统族的体系结构并提供创建它们的基本构造块。为了适于实现特定的功能，它也定义了自己的一部分。在面向对象环境中，框架由抽象类和具体类组成。这样一个框架的实例化包括组合现有类和将现有类划分成子类。

Functional Property（功能属性）

系统功能特性的特殊方面，通常与指定的功能需求有关。功能特性既可通过特殊的函数使应用程序用户直接可见，也可表示它的实现方面，就象算法用于计算函数一样。

GUI（图形用户界面/图形用户接口）

Hardwiring（硬布线）

用非常固定的方式编码，例如使用数字或字符串替代变量。因为这种数字本身没有给出线索来理解它来自哪儿和它是为什么，因此这些数字也被称为“幻数”（magic numbers）。

435

Inheritance（继承）

面向对象语言的一种特性，它允许由现有类派生新类。继承定义了实现重用、子类型关系，或两者都定义。单继承或者多继承都是可能的，这依赖于编程语言。

Inlining（内嵌）

编译时的代码扩展，它插入函数或过程体的代码来替换用于调用函数的代码。嵌入长函数体可能导致代码“臃肿”，并带来存储消耗和页面调度的负影响。

Instance（实例）

起源于特殊类的一个对象。经常作为面向对象环境中对象的同义词使用。这个术语也可用在其他语境中(参见实例化)。

Instantiation（实例化）

通过某个模板创建一个新实例的机制。该术语可用在几种语境中。对象是由类进行实例化得到的。对C++模板进行实例化以创建新类或新功能。对应用框架进行实例化以创建应用。短语“对模式进行实例化”有时是指通过描述取得模式并且填写必要细节以适合一个特殊应用。

Intercession（调解）

由系统自己增加或修改系统的结构、行为或者状态。

Intranet（内联网）

公司内部的计算机广域网络。这样一个网络可以防范外来访问，并为公司范围内的信息交流、协同工作和工作流提供平台。

Introspection（自省）

由系统自己检查系统的结构、行为和状态等所选方面的问题。

IPC（进程间通信）

IPC 机制的例子有内存共享、管道、消息队列和网络通信等。

Message（消息）

消息用于对象或进程间通信。在面向对象的系统中，术语“消息”用来描述选择和激活一个对象的操作或方法。消息同步是指发送者一直等待直到接收者完成被激活的操作。进程通常

436

进行异步通信，即发送进程继续执行，不必等待接收者回答。远程过程调用(RPC)是同步进程间通信的一种手段。

Method（方法）

表示由对象执行的操作。在类中定义方法。该术语也被用在“软件开发方法”中，它由一组在开发过程中被工程师使用的规则、指南和符号组成。

Mix-In（混入）

一个“小”类通过多继承将附加的接口和功能特性加到类中。混入也表示通过从类中继承来添加这样的功能特性。

Module（模块）

软件系统的语法或概念实体。经常用作组件或子系统的同义词。有时，“模块”也表示编译单元或文件。其他作者将这个术语作为“包”的等价词，指的是代码体有它自己的名字空间。我们对这个术语的解释如第一个句子所述。

Multiple Inheritance（多继承）

其中一个类有很多超类的继承。

Non-functional Property（非功能属性）

系统的特性没有被其功能描述所覆盖。非功能属性通常针对有关可靠性、兼容性、效率、开销、易用性、系统维护或开发等方面的问题。

Object（对象）

面向对象系统中可识别的实体。对象通过执行一个方法(操作)响应消息。对象可以包含数据值以及对其他对象的引用，它们一起确定对象的状态。因此对象包括状态、行为和身份标识。

Off-board Communication（场外通信）

跨越机器边界的通信。注意：术语“进程间通信”描述的是另一种通信类型，这种通信类型依赖于进行通信的进程是在一台机器中还是在不同的机器中。不同类型的通信在延迟、吞吐量和出错概率等方面会有所不同。

437

On-the-wire Protocol（在线协议）

“在线协议”定义高层通信工具(例如DCE、CORBA或网络OLE)如何将消息、对象、数据和其他实体转换到缓冲区中，这种“在线协议”可以跨越电线来传递。术语“电线”如今也包括诸如微波，光纤和无线电传送等传输媒介。

Peer-to-peer（对等）

在分布式系统中，对等体是相互通信的进程。与客户机-服务器体系结构中的组件相反，对等体既可以是客户机，又可以是服务器，还可以二者兼备，并且其角色可以动态改变。

Platform（平台）

用于实现系统的硬件或软件的总和。软件平台包括操作系统、库和框架。平台实现了可在

其上运行应用程序的虚拟机。

Polymorphism（多态）

其中的一个名字可以表示不同事情的概念。一个函数名在不同的时段可限制到不同的操作，或一个变量名可限制到不同类型的对象。这个概念有助于实现基于抽象的灵活系统。在面向对象语言中，多态通过操作的动态绑定机制来实现。这暗示着代码的一个固定部分可能根据与它协作的对象不同而有不同的行为。

Relationship（关系）

组件之间的连接。关系可以是静态的也可以是动态的。静态关系直接用源代码表示。它们在体系结构内处理组件的布局。动态关系处理组件间的交互作用。从源代码或图中不可能容易地看出这种动态关系。

Responsibility（责任/职责）

在特定的语境中对象或组件的功能特性。责任通常被指定为一组操作。责任部分是CRC卡片的一个元素。

Role（角色）

在相关组件的语境内组件的责任。即使在单个模式中，被实现的组件也可以有不同的角色。

S.E.P.

可表示如下3个概念之一：其他人问题（Somebody Else's Problem）、软件工程过程(Software Engineering Process)或使用模式的软件工程(Software Engineering with Patterns)。

438

Server（服务器）

由客户机请求触发的组件或子系统。当客户机请求到达时，服务器试着完成请求，服务器可以自己完成请求或将子任务委派给其他组件完成。

Single Inheritance（单继承）

其中一个类只有最多一个直接超类的继承。

Subsystem（子系统）

一组能够完成一项给定任务的相互协作的组件。在软件体系结构中，子系统被认是单独实体。它通过与其他子系统和组件的交互来完成指派给它的任务。

Superclass（超类）

另一个类可以继承的类。

System（系统）

软件或硬件的集合，执行一个或多个任务。系统可以是平台、应用程序或者两者兼而有之。

System Family（系统族）

一组解决相似任务的相关系统。系统族中的系统共享它们的体系结构和实现的大部分内容，这是因为每个系统都是由相同的框架得来。当单个系统随时间逐步演化时，它所发布的不同版

本也建立了系统族。

Unicode（统一的字符编码标准）

使用16位编码的字符表示标准。统一的字符编码标准包括适合所有可书写语言的字符，以及标点符号、数学符号和其他符号的表示。

439

参考文献

[Ada79] D. Adams: *The Hitchhiker's Guide to the Galaxy*, page 2^6, Pan Books Ltd., London, 1979

[Ada84] D. Adams: *So long, and Thanks for All the Fish*, Chapter 31, Pan Books Ltd., London, 1984

[AG96] K. Arnold, J. Gosling: *The Java Programming Language*, Addison-Wesley, 1996, see also http://java.sun.com

[Ale79] C. Alexander: *The Timeless Way of Building*, Oxford University Press, 1979

[ANAK87] C. Alexander, H. Neis, A. Anninou, I. King: *A New Theory of Urban Design*, Oxford University Press, 1987

[ASAIA75] C. Alexander, M. Silverstein, S. Angel, S. Ishikawa, D. Abrams: *The Oregon Experiment*, Oxford University Press, 1975

[AIS77] C. Alexander, S. Ishikawa, M. Silverstein with M. Jacobson, I. Fiksdahl-King, S. Angel: *A Pattern Language – Towns·Buildings·Construction*, Oxford University Press, 1977

[ASU86] A. Aho, R. Sethi, J. Ullman: *Compilers – Principles, Techniques, and Tools*, Addison Wesley, 1986

[ATM93] Siemens AG: *ATM-P: Komplexspezifikation*, internal document, 1993

[App85] Apple Computer Inc.: *Inside Macintosh, Volume I*, Cupertino, CA, 1985

[App89] Apple Computer Inc.: *Macintosh Programmers Workshop Pascal 3.0 Reference*, Cupertino, CA, 1989

[Bac86] M.J. Bach: *The Design of the UNIX Operating System*, Prentice Hall, 1986

[BaCo91] L. Bass, J. Coutaz: *Developing Software for the User Interface*, Addison-Wesley, 1991

[Bal85] H. Balzert: *Die Entwicklung von Software-Systemen*, B·I· Wissenschaftsverlag, Mannheim Wien Zürich, 1985

[Bec94] K. Beck: *Patterns and Software Development*, Dr. Dobb's Journal, 19(2), pp. 18–23, February 1994

[Bec97] K. Beck: *Smalltalk Best Practice Patterns*, Prentice-Hall, 1997

[BeCu89] K. Beck, W. Cunningham: *A Laboratory For Teaching Object-Oriented Thinking*, Proceedings of OOPSLA '89, N. Meyrowitz (Ed), Special Issue of

SIGPLAN Notices, Vol. 24, No. 10, pp. 1–6, October1989

[BI93] A.P. Black, M.P. Immel: *Encapsulating Plurality*, Proceedings of ECOOP '93, pp. 57–79, [ECOOP93]

[BJ94] K. Beck, R. Johnson: *Patterns Generate Architectures*, Proceedings of ECOOP '94, pp. 139–149, [ECOOP94]

[BKSP92] F. Buschmann, K. Kiefer, M. Stal, F. Paulisch: *The Meta-Information-Protocol: Run-Time Type Information for C++*, Proceedings of IMSA '92, pp. 82–87, [IMSA92]

[BM94] F. Buschmann, R. Meunier: *A System of Patterns*, Proceedings of PLoP '94, pp. 325–343, [PLoP94]

[BM95] F. Buschmann, R. Meunier: *Building a Software System*, Electronic Design, February 20, 1995

[BN94] J.J. Barton, L.R. Nackman: *Scientific and Engineering C++ – An Introduction with Advanced Techniques and Examples*, Addison-Wesley, 1994

[Boo94] G. Booch: *Object-Oriented Analysis and Design With Applications*, Second Edition, Benjamin/Cummings, Redwood City, California, 1994

[BR95] G. Booch: *Unified Method for Object-Oriented Development*, Version 0.8, Rational Software Corporation

[Bro94] K. Brockschmidt: *Inside OLE 2*, Microsoft Press, 1994

[Bro96] Phil Brooks: *Master-Slave Pattern for Parallel Compute Services*, submitted to the 1996 Conference on Object-Oriented Technologies and Systems (COOTS)

[BuCa96] R.J.A. Buhr, R.S. Casselman: *Use Case Maps for Object-Oriented Systems*, Prentice Hall, 1996

442

[Cam94] F.R. Campagnoni: *IBM's System Object Model*, Dr. Dobb's Journal, Special Report, #225 Winter 1994/95, pp. 24–28

[Cho90] Chorus systemès: *Chorus Kernel v3.2, Implementation Guide*, CS/TR-90-5

[CNS95] J. Coutaz, L. Nigay, D. Salber: *Agent-Based Architecture Modelling for Interactive Systems*, The Amodeus Project, ESPRIT Basic Research Action 7040, System Modelling/WP53, April 1995

[CM93] S. Chiba, T. Masuda: *Designing an Extensible Distributed Language with a Meta-Level Architecture*, Proceedings of ECOOP '93, pp. 482–501, [ECOOP93]

[Coad92] P. Coad: *Object-Oriented Patterns*, Communications of the ACM, Vol. 35, No. 9, September 1992

[Coad95] P. Coad with D. North and M. Mayfield: *Object Models – Strategies, Patterns, & Applications*, Yourdon Press, Prentice Hall, 1995

[Cope92] J.O. Coplien: *Advanced C++ – Programming Styles and Idioms*, Addison-Wesley, Reading, MA, 1992

[Cope94a] J.O. Coplien: *The Counted Body Idiom*, Pattern Mailing List Reflector, Feb 1994

[Cope94b] J.O. Coplien: *Generative pattern languages: An emerging direction of software design*, C++ Report, SIGS Publications, July-August 1994

[Cope94c] J.O. Coplien: *A Generative Development-Process Pattern Language*, Proceedings of PLoP '94, pp. 183–237, [PLoP94]

[Cope95] J.O. Coplien: *The History of Patterns*, see http://c2.com/cgi/wiki?HistoryOfPatterns

[Cope96] J.O. Coplien: Pattern Mailing List Reflector, V96 #35, April 1996

[Cou87] J. Coutaz: *PAC, an Object Oriented Model for Dialog Design*, Human-Computer Interaction – INTERACT '87 proceedings, H.-J. Bullinger and B. Shackel (Eds), pp. 431–436, Stuttgart, Germany, Elsevier Science Publishers B.V. (North-Holland), 1987

[Cra95] Iain Craig: Blackboard Systems, Ablex Publishing Corporation, Norwood, New Jersey, 1995

[Cro85] J. Crowley: *Navigation for an Intelligent Mobile Robot*, IEEE Journal of Robotics and Automation, Vol. RA-1, No. 1, pp. 31–41, March 1985

443

[Cun94] W. Cunningham: *The CHECKS Pattern Language of Information Integrity*, Proceedings of PLoP '94, pp. 145–155, [PLoP94]

[Cus93] H. Custer: *Inside Windows NT*, Microsoft Press, 1993

[CY91] P. Coad, E. Yourdon: *Object-Oriented Analysis*, Prentice Hall, second edition, 1991

[CZ95] D. Chapman, E. Zwicky: *Building Internet Firewalls*, O'Reilly & Associates, 1995

[Dij65] E.W. Dijkstra: *Solution of a Problem in Concurrent Programming Control*, CACM, Vol. 8, No. 9, p. 569, Sept. 1965

[DWP95] ANSI document X3J16/95 0088 WG21/N0688: *Programming Language C++*, draft working paper, July 1995

[ECOOP92] O. Lehrmann Madsen (Ed.): ECOOP '92 – *European Conference on Object-Oriented Programming*, Proceedings of 6th European Conference, Utrecht, The Netherlands, June/July 1992, Lecture Notes in Computer Science 615, Springer-Verlag, Berlin Heidelberg New York, 1992

[ECOOP93] O. Nierstrasz (Ed.): ECOOP '93 – *Object-Oriented Programming*, Proceedings of

7th European Conference, Kaiserslautern, Germany, July 1993, Lecture Notes in Computer Science 707, Springer-Verlag, Berlin Heidelberg New York, 1993

[ECOOP94] M. Tokoro, R. Pareschi (Eds.): *ECOOP '94 – Object-Oriented Programming*, Proceedings of 8th European Conference, Bologna, Italy, July 1994, Lecture Notes in Computer Science 821, Springer-Verlag, Berlin Heidelberg New York, 1994

[ECOOP95] W. Olthoff (Ed.): *ECOOP '95 – Object-Oriented Programming*, Proceedings of 9th European Conference, Åarhus, Denmark, August 1995, Lecture Notes in Computer Science 952, Springer-Verlag, Berlin Heidelberg New York, 1995

[EHLR88] L.D. Erman, F. Hayes-Roth, V.R. Lesser, D.R. Reddy: *The Hearsay-II Speech-Understanding System: Integrating Knowledge to Resolve Uncertainty*, ACM Computing Surveys 12 (2), pp. 213–253, 1980, reprinted in *Blackboard Systems*, pp. 31–86, [EM88]

[EKM+94] R. Eisenhauer, S. Kumsta, F. Miralles, K. Möbius, U. Steinmüller, P. Stobbe, C. Vester: *Architektur-Handbuch für Software-Architekten*, Siemens Nixdorf Informationssysteme AG, internal report, 1994

[EM88] R. Engelmore, T. Morgan (Eds): *Blackboard Systems*, Addison-Wesley, 1988 444

[ES90] M.A. Ellis, B. Stroustrup: *The Annotated C++ Reference Manual*, Addison-Wesley, 1990

[Etz64] A. Etzioni: *Modern Organizations*, Prentice-Hall, 1964

[Fel84] K. Fellbaum: Sprachverarbeitung und Sprachübertragung, Springer-Verlag, Berlin Heidelberg New York Tokyo, 1984

[FMcD77] C. Forgy, J. McDermott: *OPS: a domain-independent production system language*, Proceedings of the Fifth International Joint Conference on Artificial Intelligence IJCAI-77, pp. 933-939

[Fow96] M. Fowler: *Object Blueprints: Patterns in Systems Analysis*, Addison-Wesley, to appear

[Gam91] E. Gamma: *Objektorientierte Software-Entwicklung am Beispiel von ET++: Klassenbibliothek, Werkzeuge, Design*, Dissertation, Universität Zürich, 1991

[Gel85] D. Gelernter: *Generative Communication in LINDA*, ACM Transactions on Programming Languages and Systems, Vol. 7, No. 1, pp. 80–112, Jan. 1985

[GGR93] J. Grabowski, P. Graubmann, E. Rudolph: *The Standardization of Message Sequence Charts*, in Software Engineering Standards Symposium, Brighton, UK, 1993

[GHJV93] E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns: Abstraction and Reuse of Object-Oriented Design*, Proceedings of ECOOP '93, pp. 406–

431, [ECOOP93]

[GHJV95] E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns – Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995

[GJ79] M. Garey, D. Johnson: *Computers and Intractability – A Guide to the Theory of NP-Completeness*, W.H. Freeman and Company, New York, 1979

[Gol91] A. Goldberg: *Object-Oriented Project Management*, Tutorial TOOLS Europe, Paris, 1991

[GOP90] K. Gorlen, S. Orlow, P. Plexico: *Data Abstraction and Object-Oriented Programming in C++*, John Wiley & Sons, 1990

[GR83] A. Goldberg, D. Robson: *Smalltalk-80: the language and its implementation*, Addison-Wesley, 1983

445

[GR92] J. Grabowski, E. Rudolph: *Message Sequence Charts (MSC) – A Survey of the new CCITT Language for the Description of Traces within Communication Systems*, 1992

[HAJ90] X.D. Huang, Y. Ariki, M.A. Jack: *Hidden Markov Models for Speech Recognition*, Edinburgh University Press, Edinburgh, 1990

[HHS94] R. Händel, M.N. Huber, S. Schröder, *ATM Networks – Concepts, Protocols, Applications*, 2nd Edition, Addison-Wesley, 1994

[HRV95] J. Hartmann, C. Reichetzeder, M. Varian: *CMS Pipelines*, http://www.akh-wien.ac.at/pipeline.html

[HT92] Y. Honda, M. Tokoro: *Soft Real-Time Programming through Reflection*, Proceedings of IMSA '92, pp. 12–23, [IMSA92]

[IEEE88] IEEE: *Portable Operating System Interface for Computer Environments (POSIX)*, 1003.1, Sept. 1988

[IMSA92] A. Yonezawa, B.C. Smith (Eds.): Proceedings of the *International Workshop on New Models for Software Architecture '92 – Reflection and Meta-Level Architecture*, Tokyo, Japan, 1992

[IMY92] Y. Ichisugi, S. Matsuoka, A. Yonezawa: *RbCl: A Reflective Object-Oriented Concurrent Language without a Run-time Kernel*, Proceedings of IMSA '92, pp. 24–35, [IMSA92]

[Iona95] IONA Technologies Ltd: *Orbix Programmer's Guide*, compare also http://www.iona.ie/, Dublin, Ireland, 1995

[Jac95] M. Jackson: *Software Requirements & Specifications – a lexicon of practice, principles and prejudices*, Addison-Wesley, 1995

[Joh94] R. Johnson: *An Introduction to Patterns*, Report on Object Analsysis & Design, Vol.1, No. 1, SIGS Publications, May-June 1994

[Joh95] R. Johnson: private communication

[Joh96] R. Johnson: private communication

[Kar95] E-A. Karlsson (Ed.): *Software Reuse – A Holistic Approach*, John Wiley & Sons, 1995

[Kee89] S.E. Keene, *Object-Oriented Programming in Common Lisp – A Programmer's Guide to CLOS*, Addison-Wesley, 1989 446

[Kic92] G. Kiczales: *Towards a New Model of Abstraction in Software Engineering*, Proceedings of IMSA '92, pp. 1–11, [IMSA92]

[KLLM95] G. Kiczales, R. DeLine, A. Lee, C. Maeda: *Open Implementation – Analysis and Design™ of Substrate Software*, Tutorial #21 of OOPSLA '95, October 1995

[KMS+92] A. Kausche, M. van Meegen, A. Schappert, P. Sommerlad, K. Bergner, B. Rumpe: *Exploration Field Automated Software Development – State-of-the-Art Report*, Siemens AG, internal technical report, Munich, 1992

[Koe95] A. Koenig: *Another handle variation*, Journal of Object-Oriented Programming (JOOP), SIGS Publications, November-December 1995

[KP88] G.E. Krasner, S.T. Pope: *A cookbook for using the Model-View-Controller user interface paradigm in Smalltalk-80*, Journal of Object-Oriented Programming, 1(3), pp. 26–49, August/September 1988, SIGS Publications, New York, NY, USA, 1988

[KR88] B. W. Kerninghan, D.M. Ritchie, *The C Programming Language*, 2nd edition covering ANSI-C, Prentice Hall, 1988

[KR96] J. Knopp, M. Reich: *A Data Model For Architecture Independent Parallel Programming*, Workshop on High-Level Programming Models and Supportive Environments at the IEEE International Parallel Processing Symposium, Honolulu, 1996

[KRB91] G. Kiczales, J. des Rivières, D. Bobrow: *The Art of the Metaobject Protocol*, MIT Press, 1991

[Kru95] P.B. Kruchten: *The 4 + 1 View Model of Architecture*, IEEE Software, November 1995, pp. 42–50

[Kru96] D. Kruglinski: *Inside Visual C++*, Microsoft Press, 1995

[KSS96] S. Kleiman, D. Shah, B. Smaalders: *Programming with Threads*, SunSoft Press, Prentice Hall, 1996

[LA94] A. Luotonen, K. Altis: *World-Wide Web Proxies*, WWW94 Conference, 1994, see also http://www.w3.org/pub/WWW/Daemon/

[LCITV92] M. Linton, P. Calder, J. Interrante, S. Tang, J. Vlissides: *InterViews Reference Manual*, CSL, Stanford University, 3.1 edition, 1992

447 [Lea96] D. Lea: *Collections, a Java package*, http://g.oswego.edu/dl/, 1996

[LeEr88] V.R. Lesser, L.D. Erman: *A Retrospective View of the Hearsay-II Architecture*, in *Blackboard Systems*, Proc. of IJCAI-77, pp. 790–800 and Technical Report CMU-CS-78-117, reproduced in *Blackboard Systems*, pp. 87–121, [EM88]

[Lim93] C.-C. Lim: *A Parallel Object-Oriented System for Realizing Reusable and Efficient Data Abstractions*, PhD dissertation, TR-93-063, International Computer Science Institute, Berkeley, CA, 1993, see also http://www.icsi.berkeley.edu/~sather/psather.html

[LP91] W.R. LaLonde, J.R. Pugh: *Inside Smalltalk, Volume II*, Prentice-Hall, 1991

[LPW94] K.-P. Löhr, I. Piens, T. Wolff: *Verteilungstransparenz bei der objektorientierten Entwicklung verteilter Applikationen*, OBJEKTspektrum 5/1994, pp. 8–14, SIGS Publications, München, Germany, 1994

[Mae87] Pattie Maes, *Concepts and Experiments in Computational Reflection*, in Proceedings of OOPSLA '87, pp. 147–155, 1987

[Maf96] S. Maffeis: *The Object Group Design Pattern*, 2nd USENIX Conference on Object-Oriented Technologies and Systems (COOTS), Toronto, Ontario, Canada, 1996

[Mar95] J. Markowitz: *Talking to Machines*, Byte, December 1995, pp. 97–104

[McA95] J. McAffer: *Meta-level Programming with CodA*, Proceedings of ECOOP '95, pp. 190–214, [ECOOP95]

[ME96] T.D. Meijler, R. Engel: *Making Design Patterns explicit in FACE, a Framework Adaptive Composition Environment*, submitted to EuroPLoP '96

[Mes94] G. Meszaros: *Pattern: Half Object + Prototocol (HOPP)*, Proceedings of PLoP '94, pp.129–132, [PLoP94]

[Mey92] S. Meyers: *Effective C++ – 50 Specific Ways to Improve Your Programs and Designs*, Addison-Wesley, 1992

[MFL93] S. Murer, J. Feldman, C. Lim: *pSather: Layered Extensions to an Object-Oriented Language for Efficient Parallel Computation*, International Computer Science Institue, TR-93-028, Berkeley, CA, 1993

[Mic93b] Microsoft Corporation: *Microsoft Word*, User's Guide, 1993

[Mic95] Microsoft Corporation: *Microsoft Visual Basic*, Programmer's Guide, 1995

[Nii86] H.P. Nii: *Blackboard Systems, Part I and II*, The AI Magazine, vol. 7, nos 2 (pp. 38–53) and 3 (pp. 82–106), 1986

448

[NS72] A. Newell, H.A. Simon: *Human Problem Solving*, Prentice-Hall, 1972

[OIT92] H. Okamura, Y. Ishikawa, M. Tokoro: *AL-1/D: A Distributed Programming*

System with Multi-Model Reflection Framework, Proceedings of IMSA '92, pp. 36–47, [IMSA92]

[OMG92] Object Management Group: *The Common Object Request Broker: Architecture and Specification*, OMG Document Number 91.12.1, Revision 1.1, 1992

[OMG95] Object Management Group: *CORBAservices: Common Object Services Specification*, OMG Document Number 95-3-31, 1995

[Omo93] S.M. Omohundro: The Sather programming language, Dr. Dobb's Journal, 18(11):42–48, October 1993, see also http://www.icsi.berkeley.edu/Sather/

[Par79] D.L. Parnas: *On the criteria to be used in decomposing systems into modules*, CACM, Vol. 15, pp.1053–1058, Dec. 1972

[Par94] D.L. Parnas: *Software Aging*, IEEE Proceedings of the 16th International Conference on Software Engineering, 1994

[PCW85] D.L. Parnas, P.C. Clements, D.M. Weiss: *The Modular Structure of Complex Systems*, IEEE Transactions on Software Engineering, Vol. SE-11, No. 3, March 1985

[PLoP94] J.O. Coplien, D.C. Schmidt (Eds.): *Pattern Languages of Program Design*, Addison-Wesley, 1995 (a book publishing the reviewed Proceedings of the First International Conference on Pattern Languages of Programming, Monticello, Illinois, 1994)

[PLoP95] J.O. Coplien, N. Kerth, J. Vlissidis (Eds.): *Pattern Languages of Program Design*, Addison-Wesley, 1996 (a book publishing the reviewed Proceedings of the Second International Conference on Pattern Languages of Programming, Monticello, Illinois, 1995)

[PP90] ParcPlace Systems Inc.: *Objectworks\Smalltalk Release 4.1 User's Guide*, ParcPlace Systems, 1992

[Pree94] W. Pree: *Meta Patterns – A Means For Capturing the Essentials of Reusable Object-Oriented Design*, Proceedings of ECOOP '94, pp 150–162, [ECOOP94]

[Pree95] W. Pree: *Design Patterns for Object-Oriented Software Development*, Addison-Wesley, 1995 449

[PST96] G. Parulkar, D. Schmidt, J. Turner: *$a^I t^P m$: a Strategy for Integrating IP with ATM*, Proceedings of SIGCOMMM, ACM, Aug/Sep 1996, see also http://siesta.cs.wustl.edu/~schmidt/

[PW92] D.E. Perry, A.L. Wolf: *Foundations for the Study of Software Architecture*, ACM SIGSOFT, Software Engineering Notes, Vol. 17, No. 4, pp. 40–52, October 1992

[Rab86] L.R. Rabiner et al: *An Introduction to Hidden Markov Models*, IEEE ASSP Magazine, Vol 3, pp. 4–16, January 1986

[Rab89] L.R. Rabiner: *A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition*, Proceedings IEEE, Vol 77, No 2, pp 257–285, 1989

[RBPEL91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen: *Object-Oriented Modeling and Design*, Prentice Hall, 1991

[Ree92] T. Reenskaug: *Intermediate Smalltalk, Practical Design and Implementation*, Tutorial, TOOLS Europe '92, Dortmund, 1992

[RWL96] T. Reenskaug, P. Wold, O.A. Lehne: *Working with Objects: The OOram Software Engineering Method*, Manning Publications Company, 1996

[SC95a] A. Sane, R. Campbell: *Detachable Inspector/Removable cout: A Structural Pattern for Designing Transparent Layered Services*, Proceedings of PLoP '95

[SC95b] A. Sane, R. Campbell: *Composite Messages: A Structural Pattern For Communication Between Components*, OOPSLA '95 Workshop on Concurrent, Parallel, and Distributed Object-Oriented Systems, Austin TX, 1995, see also http://siesta.cs.wustl.edu/~schmidt/OOPSLA-95/index.html

[Sch86] K.J. Schmucker: *Object-Oriented Programming for the Macintosh™*, Hayden Book Company, Hasbrouck Heights, New Jersey, 1986

[Sch94] D.C. Schmidt: *Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching*, Proceedings of PLoP '94, pp. 529–545, [PLoP94]

[Sch95] D.C. Schmidt: *A System of Reusable Design Patterns for Communication Software*, Theory and Practice of Object Systems, Special Issue on Patterns and Pattern Languages, S.P. Berczuk (Ed), John Wiley and Sons, 1995

[Sch96] D.C. Schmidt: *ACE – The ADAPTIVE Communication Environment*, see http://siesta.cs.wustl.edu/~schmidt/ACE.html

450 [Sch96b] D.C. Schmidt: *Acceptor and Connector – Design Patterns for Initializing Network Services*, submitted to EuroPLoP '96

[Set95] J. Sethna: *LASSPTools: Graphical and Numerical Extensions to Unix*, http://www.lassp.cornell.edu/LASSPTools/LASSPTools.html

[SG96] M. Shaw, D. Garlan: *Software Architecture – Perspectives on an Emerging Discipline*, Prentice Hall, 1996

[SHA96] D. C. Schmidt, T. Harrison, E. Al-Shaer: *Object-Oriented Components for High-speed Network Programming*, Department of Computer Science, Washington University, 1996, see also http://siesta.cs.wustl.edu/~schmidt/

[SICAT95] Siemens AG: *SICAT Steuerprogramm: Entwurfsspezifikation*, internal document no. P30308-A6331-A000-02-D8

[SL92] B. Stroustrup, D. Lenkov: *Run-Time Type Identification for C++*, ANSI C++

standards document No. X3J16/92-0028, 1992

[SM88] S. Shlaer, S.J. Mellor: *Object-Oriented Systems Analysis – Modeling the World In Data*, Yourdon Press, Prentice Hall, 1988

[Smi82] Brian C. Smith, *Reflection and Semantics in a Procedural Language*, PhD thesis, Massachusetts Institute of Technology, 1982

[SNH95] D. Soni, R. Nord, C. Hofmeister: *Software Architecture in Industrial Applications*, in Proceedings of the 17th International Conference on Software Engineering, pp. 196–207, Seattle, Washington, ACM Press, April 1995

[SNI94] Siemens Nixdorf Informationssysteme AG: *Generic++ 2.0, Portable C++ Foundation Class Library*, User manual, October 1994

[Sou94] J. Soukup: Implementing Patterns, Proceedings of PLoP '94, pp.395–412, in [PLoP94]

[SRC84] J. Saltzer, D. Reed, D. Clark: *End-To-End Arguments in System Design*, ACM Transactions on Computer Systems, Vol. 2, No. 4, pp. 277–288, Nov. 1984

[SS86] E. Seidewitz, M. Stark: *Towards a General Object-Oriented Software Development Methodology*, Proceedings of the First International Conference on Ada Programming Language Applications for the NASA Space Station, Lyndon B. Johnson Space Center, Texas, NASA, 1986

[Stee91] D. Steel: *Distributed Object Oriented Programming: Mechanisms & Experience*, Proceedings of TOOLS USA '91, pp. 27–35, Prentice Hall, 1991 451

[Ste90] W.R. Stevens: *UNIX Network Programming*, Prentice Hall Software Series, 1990

[Ste94] W.R. Stevens: *TCP/IP Illustrated, Volume 1, The Protocols*, Addison-Wesley, 1994

[Ste95] U. Steinmüller: private communication

[Str91] B. Stroustrup: *The C++ Programming Language*, Second Edition, Addison-Wesley, 1991

[Sun90] Sun Microsystems, Inc.: *Sun OS Documentation Tools, Formatting Documents*, March 1990

[SW95] R.J. Stroud, Z. Wu: *Using Metaobject Protocols to Implement Atomic Data Types*, Proceedings of ECOOP '95, pp. 168–189, [ECOOP95]

[Tal94] Taligent Inc.: *Taligent's Guide To Designing Programs – Well-Mannered Object-Oriented Design in C++*, Addison-Wesley, 1994

[Tan92] A.S. Tanenbaum: *Modern Operating Systems*, Prentice Hall, 1992

[TASC91] Siemens AG: *Toolkit for Autonomous Software Components Communication*,

Systemdokumentation, internal document, 1991

[Ter88] A. Terry: *Using Explicit Strategic Knowledge to Control Expert Systems*, originally published in 1985, reproduced in *Blackboard Systems*, pp. 159–188, [EM88]

[THG94] R. Thomson, K.E. Huff, J.W.Gish: *Maximizing Reuse During Reengineering*, Proceedings of the Third International Conference on Software Reuse, Rio de Janeiro, Brazil, pp. 16–23, IEEE Computer Society Press, 1994

[THP94] W.F. Tichy, J. Heilig, F. Newbery Paulisch: *A Generative and Generic Approach to Persistence*, C++ Report, SIGS Publications, January 1994

[TS93] C. Traving, H. Stadtherr: *Building a Traffic Management System with C++*, Proceedings of the C++ User Group Technical Conference, Munich, 1993

[U2] U2: *even BETTER than the REAL THING*, Island Records Ltd., 1991

[VBT95] Allan Vermeulen, Gabe Beged-Dov, Patrick Thompson: *The Pipeline Design Pattern*, OOPSLA '95 Workshop on Design Patterns for Concurrent, Parallel and Distributed Object-Oriented Systems, see also

452 http://siesta.cs.wustl.edu/~schmidt/OOPSLA-95/html/papers.html

[VL90] J. Vlissides, M. A. Linton: *Unidraw – A framework for building domain-specific graphical editors*, acm Transactions on Information Systems, Vol. 8, No. 3, pp. 237–268, July 1990

[WBWW90] R. Wirfs-Brock, B. Wilkerson, L. Wiener: *Designing Object-Oriented Software*, Prentice Hall, 1990

[WGM88] A. Weinand, E. Gamma, R. Marty: *ET++ – An Object-Oriented Application Framework in C++*, in Proceedings of OOPSLA '88, pp. 46–57, San Diego, 1988

[Wil84] M.A. Williams: *Hierarchical Multi-expert Signal Understanding*, Technical Report ESL-IR201, ESL Inc, Sunnyville, CA, 1984. Reproduced in *Blackboard Systems*, pp. 387–415, [EM88]

[Woo96] D.W. Woodward: *Ein Microkernel für die Datenbank*, Software-Entwicklung, AWi Verlag, April 1996, S. 28–31

[Yok92] Y. Yokote: *The New Mechanism for Object-Oriented System Programming*, Proceedings of IMSA '92, pp. 88–93, [IMSA92]

[ZEWH95] S.H. Zweben, S.H. Edwards, B.W. Weide, J.E. Hollingsworth: *The Effects of Layering and Encapsulation on Software Development Cost and Quality*, IEEE Transactions on Software Engineering, Vol. 21, No. 3, pp. 200–208, 1995

[Zim94] W. Zimmer: *Relationships Between Design Patterns*, Proceedings of PLoP '94, pp. 345–364, [PLoP94]

[Zim96] C. Zimmermann: *Objektorientierte Konzepte: Entscheidungsschichten zwischen Anwendung und Betriebssystemkern—Zwischenspiele*, iX, Februar 1996, S.146–151 453

模式索引

Abstract Factory（抽象工厂）……206, 211, 284, 292, 380, 397, 398
Acceptor（接受器）……206, 337
Active Object（主动对象）……162, 257
Adapter（适配器）……49, 158, 267, 380

Blackboard（黑板）……26, 29, **71-95**, 366, 380
Bridge（桥接）……40, 49, 140, 206, 211, 371, 380
Broker（代理者）……26, 98, **99-122**, 191, 306, 331, 335, 337, 366, 380, 385
Builder（生成器）……380

Chain of Responsibility（职责链）……139, 244, 371, 380
Client-Dispatcher-Server（客户机-分配器-服务器）……106, 121, 163, 182, 222, 256, 274, 306, 322, **323-337**, 364, 366, 380
Client-Server（客户机-服务器）……366
Command（命令）……41, 244, 276, 278, 289, 300, 380
Command Processor（命令处理器）……136, 142, 158, 222, 276, **277-290**, 301, 366, 371, 380
Composite（组合/组成）……51, 129, 139, 224, 234, 238, 240, 241, 284, 367, 370, 371, 380
Composite Message（组合消息）……51, 152, 160, 161, 366, 370
Connector（连接者）……206, 337
Counted Body（计数体）……357
Counted Pointer（计数指针）……14, 15, 234, 270, **353-358**, 366, 380

Decorator（装饰）……275, 380
Dependents（从属物）……339
Detachable Inspector（可分析审查者）……206
Document-View（文挡-视图）……17, 140, 141, 369

Envelope-Letter（信封-信件）……211
Event Channel（事件通道）……223, 341
Exceptional Value（异常值）……255, 256

Facade（外观）……40, 86, 158, 159, 208, 242, 261, 380
Factory Method（工厂方法）……137, 298, 371, 380
Flyweight（享元）……380
Forwarder-Receiver（转发器-接收器）……18, 121, 162, 182, 222, 232, 256, 268, 272, 273, 306, **307-322**, 337, 364, 366, 380, 399

Half-Sync/Half-Async（半同步-半异步）……162

Handle-Body（句柄-主体/句柄-实体）……15, 366

Indented Control Flow（缩排控制流）……349
Interpreter（解释器）……287, 288, 367, 380
Iterator（迭代器）……261, 299, 380

Layers（层）……26, 29, **31-51**, 69, 70, 85, 120, 183, 192, 199, 364, 366, 367, 380, 398, 400

Main Program and Subroutines（主程序和子例程）……378
Master-Slave（主控-从属）……222, 243, 244, **245-260**, 366, 380
Mediator（中介者）……121, 160, 233, 244, 292, 299, 380
Memento（备忘录）……276, 283, 380
Meta-Level Architecture（元层次体系结构）……193
Microkernel（微核）……26, 38, 47, 51, 98, 169, **171-192**, 219, 366, 380
Model-View-Controller（模型-视图-控制器）……3, 9, 10, 12, 16, 17, 22, 26, 123, **125-143**, 167, 292, 303, 366, 369, 371, 380, 391, 400
MVC 参见Model-View-Controller

Object Group（对象组）……260
Objectifier（体现者）……206
Observer（观察者）……13, 223, 306, 339
Open Implementation（开放实现）……193

PAC 参见Presentation-Abstraction-Control
Pipes and Filters（管道和过滤器）……26, 29, 41, **53-70**, 86, 98, 365, 366, 367, 380, 391, 400
Presentation-Abstraction-Control（表示-抽象-控制）……26, 51, 123, 143, **145-168**, 303, 306, 369, 380
Prototype（原型）……284, 380
Proxy（代理）……18, 23, 104, 105, 113, 121, 162, 186, 222, 256, 261, **263-275**, 366, 375, 380
Publisher-Subscriber（出版者-订阅者）……13, 16, 41, 127, 132, 160, 161, 223, 298, 306, **339-343**, 366, 371, 380

Reactor（反应器）……41, 186, 318, 341, 366
Reference Counting Idiom（引用计数惯用法）……357
Reflection（映像）……26, 40, 85, 112, 115, 169, 191, **193-219**, 366, 380, 399

Singleton（单件）……208, 253, 286, 299, 364, 380
Singleton (C++)（单件（C++））……350
Singleton (Smalltalk)（单件（Smalltalk））……351
State（状态）……206, 380
Strategy（策略）……23, 40, 84, 206, 209, 211, 252, 259, 299, 380

Template Method（模板方法）……380

View Handler（视图处理程序）……138, 157, 222, 276, **291-303**, 366, 371, 380

Visitor（访问者）……206, 211, 380

Whole-Part（整体-部分）……208, 222, 224, **225-242**, 272, 317, 366, 367, 368, 380, 399, 400
Window Place（窗口位置）……2

索　引

索引中的页码为英文原书页码，与书中边栏的页码一致。

A

Abstract Class（抽象类），433
Abstract Command（抽象命令），278, 282
Abstract Component（抽象组件），433,
Abstract Data Type（抽象数据类型），426
Abstract Method（抽象方法），433
Abstract Original（抽象原件），265
Abstract View（抽象视图），294
Abstraction（抽象），147, 158, 398
  Criterion（准则/标准），38
  Level（层次），34, 39, 75, 176
Access Control（访问控制），222, 261, 365
Active Filter（主动过滤器），55, 57, 62
Active Server（主动服务器），185
Adams，Douglas（人名），1, 291, 345, 383
Adaptable Systems（自适应系统），26, 169, 364
Adaptation（自适应），379
Adapter（适配器），118, 177, 186
Agent（智能主体），307
AI（人工智能）。参见Artificial Intelligence
Alexander, Christopher（人名），xii, 360, 414
Algorithm（算法），426
American Football（美式足球），38
Analysis Patterns（分析模式），379
API（应用程序编程接口）。参见Application Programming Interface
Apple
  AppleTalk，188
  MacApp，124, 287
  Macintosh，187
  Macintosh Window Manager，301
  MacOS，187
Application（应用/应用程序），433
Application Framework（应用框架），186, 396, 434
Application Platform（应用平台），172, 177
Application Programming Interface（应用程序编程接口），46, 184, 433
Architectural Pattern（体系结构模式），xiv, 12, 25, 26, 363
Architectural Style（体系结构风格），394
Artificial Intelligence（人工智能），73, 124
Assembly-Part（组装-部分），226, 234, 236
Associative Array（相关数组），434
Asynchronous Transfer Mode（异步传输模式），49
ATM（异步传输模式）。参见Asynchronous Transfer Mode
At-most-once Semantic（至多一次语义），116
Atomic Service（原子服务），174

B

Backend（后端），53
Barrier（栅栏/屏蔽），255
Base Level（基本层次），195, 197, 199, 211
Beck, Kent（人名），414
Binary Standard（二元标准），111
Blackboard（黑板），75
  Entry（输入），75
  Vocabulary（词汇），75, 82
Black-box Approach（黑盒方法），40
Body（主体/整体），354, 357
Bottom-up Approach（自底向上方法），38, 231
Bridge（桥接/网桥），105, 110
Broker（代理者），101, 103
  Repository（仓库/库），113
Business Application（业务应用），192
Business Modeling（业务建模），47

C

C Standard Library（C标准库），47
C++（一种语言），41, 129, 193. 204, 213, 215, 219, 346, 347, 349, 350, 403, 420
Cache（缓存），269
  Invalidation（无效/作废），269
Cache Proxy（缓存代理），269
Callback（回调/回叫），41, 118, 185, 277, 285
Cardinality Property（基数性质），233

CGI（通用网关接口）。参见Common Gateway Interface
Changeability（易修改性），92, 100, 119, 194, 217, 241, 302, 336, 405
Change－propagation Mechanism（更改-传播机制/变更-传播机制），127, 130, 131, 132, 161, 339
  Registry（注册表/配置表），132, 339
  Subscription（订阅/预定），131, 133
  Unsubscription（取消预定），133
  Update（更新），134
Chorus（人名），170, 189
Class（类），434
Class-Responsibility-Collaborator Card（类-责任-协作者卡片），429, 434
Client（客户机），101, 102, 173, 176, 264, 319, 324, 327, 331, 434
Client-Server computing（客户机-服务器计算），102, 324
Client-side Proxy（客户机端代理），104, 113
CLOS（一种语言），170, 213, 214
Coad, Peter（人名），416
Collaborator（协作者），434
Collection-Member（汇集-成员），227, 234
  Command（命令），278, 283
  Command Processor（命令处理器），278, 279, 286
  Common Gateway Interface（公用网关接口），102
Common Object Request Broker Architecture（公用对象请求代理体系结构），98, 118, 170, 217, 273, 306, 335
Communication（通信），222, 305, 365
  Channel（通道/信道），188, 308, 317, 324, 331
  Facility（设施/设备），174, 176, 328
  Link（链接/连接），315, 325
  Mechanism（机制），100, 314, 324
  Path（路径），188
  Protocol（协议），328
Component（组件/构件），169, 385, 434
  Communication（通信），324
  Cooperation（合作/协作），243
  Interoperability（可互操作性），111
  Relationships（关系），182
Computational Accuracy（计算准确性），247, 256
Computer Network（计算机网络），305, 307, 323
Concrete Class（具体类），434
Concrete Component（具体组件），434
Concurrency（并发），248, 289
Configurability（可配置性），336
Connector（连接器/连接程序），386
Container（容器），434
Container-Contents（容器-内容），227, 234
Control（控制），75, 77, 83, 147, 158, 166
Controller（控制器），127, 128, 135, 279, 284
Coplien, James O.（人名），xv, 415, 421
COBRA（公用对象请求代理体系结构）。参见Common Object Request Broker Architecture
Counting Proxy（计数代理），270
Coupling and Cohesion（耦合和内聚），400
CRC-card（CRC卡片）。参见Class-Responsibility-Collaborator Card
Creation（创建/建立），379
Cunningham, Ward（人名），414, 423

D

Data（数据）
  Flow（流、流程），55
  Model（模型），148
  Sink（汇点），55, 56
  Source（源），55, 56
  Stream（流），54, 310
  Structure（结构），426
Database（数据库），172, 189
Debugger（调试器），206
Debugging（调试），121
Delegation（委派/委托），232
Demultiplexing（多路分用/多路复用），317, 337, 435
Descriptor Table（描述符表），330
Design（设计），435
Design Pattern（设计模式），xiv, 12, 221, 363
Device Context（设备语境），184
Device Driver（设备驱动程序），175
Directory Service（目录服务），115
Dispatcher（分配器），321, 324, 325, 330
Distributed（分布式）
  Server（服务器），323
  Service（服务），101
  Systems（系统），26, 97, 305, 364
Distributed Smalltalk（分布式Smalltalk），321
Distribution（分布），100
Divide and conquer（分而治之），243, 246, 403
Document-View（文档-视图），140
Domain（域/领域），435

Domain Analysis（领域分析），180
Domain-specific Patterns（特殊领域模式），421
Drag and Drop（拖放），435
Dynamic Binding（动态绑定），435
Dynamic Client-Server Model（动态客户机-服务器模型），102
Dynamic Lookup（动态查找），274

E

Economics（效益），97
Efficiency（效率/效能/有效性），50, 68, 69, 94, 120, 140, 167, 218, 241, 259, 274, 289, 302, 321, 336, 407
Emergent Behavior（紧急行为），226
Emulator（仿真程序/模拟器），177
Enabling Technique（启用技术），397
Encapsulation（封装），143, 289, 305, 321, 399,
Error Handling（出错处理），43, 63, 69
ET++（一种语言），124, 141, 240, 287, 294
Ethernet（以太网），44
Event（事件），41
  Handling（处理），130, 139, 283, 289
  Loop（循环），138
Event Channel（事件通道），341
Event-driven System（事件驱动系统），55
Exception Handling（异常处理），205
Exchangeability（可交换性），259, 308, 336
Expert System（专家系统），73
Extensibility（可扩展性），119, 190, 259, 302, 406
External Interface（外部接口），160
External Server（外部服务器），173, 176, 185

F

Fault Tolerance（容错），93, 120, 190, 246, 336, 408
Filter（过滤器），55, 62
  Recombination（重组合），62
  Synchronization（同步），62, 69
Firewall Proxy（防火墙代理），271
Flexibility（灵活性/适应性），67, 68, 100, 126, 190, 284, 288, 321
Forwarder（转发器），309, 315
Forwarding（转发），232
Four-layer Architecture（四层体系结构），47
Fragile Base Class Problem（脆弱基类问题），46
Framework（框架），129, 139, 140, 142, 396, 420, 435
Fresco（工具箱），240
From Mud to Structure（从混沌到结构），26, 29, 364
Frozen Spot（冰点），396
Function Call Mechanism（函数调用机制），195, 196
Functional Core（功能核心），132, 172
Functional Property（功能属性），389, 435

G

Gamma, Erich（人名），xii, 415,
Gang-of-Four（四人帮），xiii, 221, 379, 415
Garbage Collection（无用单元收集），205, 347
Gatekeeper（值班员），341
Gateway（网关），114
Generic Function（类属函数），214
Generic Function Invocation（类属函数调用），214
Generic++（类库），347
GoF（四人帮）。参见 Gang-of-Four
Graph Theory（图论），245
Graphical User Interface（图形用户界面），123, 172, 435
Gray-box Approach（灰盒方法），40
GUI（图形用户界面）。参见Graphical User Interface

H

Handle（句柄；处理），354, 357
Hardwiring（硬布线），435
HEARSAY-II（语音识别系统），87, 89
Helm, Richard（人名），xii, 359, 415
Heuristics（启发式），84
Hierarchical Component Structures（层次组件结构），224
Hillside Group（山腰组（某团体名）），418, 423
Horizontal Structuring（横向结构），32
Hot Spot（热点），396
HTML（超文本标记语言）。参见Hypertext Markup Language
Human-computer Interaction（人机交互），132, 146, 157, 158
Hypertext Markup Language（超文本标记语言），102
Hypothesis（假设），75

I

IBM OS/2 Warp（操作系统），97, 173, 186
Idiom（惯用法/习语），xiv, 14, 345, 363, 420
IDL（接口定义语言）。参见Interface Definition Language
IDL Compiler（IDL编译器），116
Implementation（实现），402
Information Hiding（信息隐藏），399
Information Provider（信息提供者），323
Infrastructure Systems（基础设施系统），46
Inheritance（继承），436
Inlining（内联/内嵌），436
Inspector（审查器），206
Inspiration（激励/启发），40
Instance（实例），436
Instantiation（实例化/例示），436
Interactive Systems（交互系统），26, 123, 364
Intercession（仲裁/调解），436
Inter-component Communication（组件间通信），186
Interface（接口/界面），402
Interface Definition Language（接口定义语言），101, 111
Internal Server（内部服务器），173, 175, 185, 192
Internet（因特网），45
Interoperability（互操作性），120, 407
Interpreter（解释器），53
Inter-process Communication（进程间通信），62，100，178, 196, 205, 308, 328, 337, 436
  Asynchronous（异步），182, 317, 319
  Asynchronous Invocation（异步调用），108
  Broadcast（广播），115
  Connection Policy（连接策略），337
  Direct（直接），106, 114
  Dynamic Invocation（动态调用），112, 115
  Indirect（间接），106
  Off-board（场外），117
  One-way（单向），319
  Protocol（协议），106
  Static Invocation（静态调用），112
  Synchronous（同步），182
  Synchronous Invocation（同步调用），108
  Two-Way（双向），319
InterViews（内部视图），287
Intranet（内联网），100, 436
Introspection（自省/内部检查），200, 215, 436
IPC（进程间通信）。参见Inter-process Communication

J

Jackson, Michael（人名），392
Java，43, 102, 234, 420
Java Virtual Machine（Java虚拟机），46
Johnson, Ralph（人名），xii, 415
JVM（Java虚拟机）。参见Java Virtual Machine

K

Knowledge Source（知识源），75, 76, 82, 85
Action-part（行为-部分），77, 85
Application Strategy（应用策略），77
Condition-part（条件-部分），77, 85

L

LAN（局域网）。参见Local Area Network
Layer（层/层次），34, 39, 48, 118, 183, 199
Layer Cake（层次块），25
Lazy Construction（惰性构造），270
Library（库），420
Life, the Universe and Everything（生命，万物一切），345
Linda（人名），258
Load-on-demand（按需装入），274
Local Area Network（局域网），97
Location Transparency（位置透明性），119, 305, 336
Logging（登录/记日志），278, 285, 288
Lowest Common Ancestor（最小公共祖先），156

M

Mach（操作系统），170, 188
Macro（宏/宏指令/宏定义），278, 284, 288
Mailbox（邮箱），183
Maintainability（可维护性），92, 100, 405
Management（管理），222, 276, 365
Marshaling（列集/编组），104, 232, 308
Martin, Robert（人名），416
Master（主控文件/总的），246, 247, 250, 252
Mechanism（机制），174, 182, 190
Memory Management（内存管理），346, 354

Mental Building-block（智力构造块/智力积木），21
Message（消息），436
Backbone（主干网/中枢），188
Buffer（缓冲），115，183
Call（调用），101
Header（标题/头），331
Passing（传递），182
Port（端口），187
Protocol（协议），313
Queue（队列），308，317，330
Reception（接收），317
Message Sending（消息发送），315
Message Sequence Chart（消息序列图/消息顺序图），431
Message Transfer（消息传递），311
Meta Level（元层次），195，196，199
Metaobject（元对象），115，195，196，206
Metaobject Protocol（元对象协议），115，195，198，208
Method（方法），437
Method Tables（方法表），111
Methods（方法），23，424
Booch（人名），23，372，392
Coad/Yourdon（人名），372，392
Object Modeling Technique（对象建模技术），23，372，392
Shlaer/Mellor（人名），372，392
Microkernel（微核），173，174，183
Microsoft（微软），
MFC（微软基础类库）。参见Microsoft Foundation Class Library
Microsoft Foundation Class Library（微软基础类库），124，141
Object Linking and Embedding（对象链接与嵌入），98，111，119，170，217，273
OLE（对象链接与嵌入）。参见Object Linking and Embedding
Win16（操作系统），186
Win32（操作系统），186
Windows3.11（操作系统），171，186
Windows 95（操作系统），125，186
Windows NT（操作系统），47，97，170，186，189
Word（字处理软件），302
Migration Transparency（迁移透明性），336
Mill, John Stuart（人名），413
Mixed-mode Proxy（混合方式代理），272
Mix-In（混入），437
Model（模型），126，127，132
Modularization（模块化），400
Module（模块），437
Monitoring（监控），307
Mostly Harmless（通常无害），291
MSC（消息序列图）。参见Message Sequence Chart
Multiple Inheritance（多继承），437
Multiprocessing System（多处理系统），97
Multi-tasking（多任务），162，166
Multi-threading（多线程），330
Multi-user System（多用户系统），162
Mutual Exclusion（互斥），270

N

Name（命名/名字），
Mapping（映射），308，313，320
Repository（仓库），330
Service（服务），104，115
Space（空间/间隔），313
Named Pipe（命名管道），66，332
Network Failure（网络失效），324
Network Management（网络管理），307
Networking Protocol（网络协议），31
NeXTSTEP（操作系统），171，188，273
NIHCL（C++类库），347
Non-blocking I/0（非阻塞输入/输出），317
Non-functional Property（非函数性质），389，404，437
Notification（通知），36
NP-complete（NP-完全），245

O

Object（对象），437
Composition（组装/组合），225
Creation（创建），205，211，347
I/0（输入/输出），215
Model（模型），101，111，205
Object Management Group（对象管理组），97，118
Object Message Sequencing Chart（对象消息顺序图），431
Object Modeling Technique（对象建模技术），429
Object Technology（对象技术），23，101
Observer（观察者），339
Off-board Communication（场外通信），437
OMG（对象管理组）。参见Object Management Group

OMSC（对象消息顺序图）。参见Object Message Sequencing Chart
OMT（对象建模技术）。参见Object Modeling Technique
One-way Coupling（单向耦合），41
On-the-wire Protocol（在线协议），114, 438
OpenStep（开放阶段），186
Operating System（操作系统），97, 169, 172
Opportunistic Problem Solving（机会主义的问题求解），74
Orbix，273
Organization of Work（工作机构），222, 243, 364
Organizational Patterns（组织模式），421
Original（原件/原始），264
OSI 7-layer Mode（开式系统互联7层模型），31

P

PAC Agent（表示-抽象-控制智能主体），145, 146, 148, 158
PAC Hierarchy（PAC层次），146, 155
Page Swapping（页面交换），205
Parallel Computing（并行计算），246, 255
Parallel Processing（并行处理），68, 69
Parnas, David（人名），405
Part（部分/部件），227, 231, 233
Passive Filter（被动过滤器），55, 57, 62
Passive Server（被动服务器），185
Pattern（模式），xi, 1, 2, 3, 5, 8, 21, 411, 426
  Catalog（目录），23
  Category（范畴/类别），11, 363, 368, 379
  Classification（分类），362, 423
  Classification Schema（分类图式/分类概要），365, 379
  Combination（组合），17
  Context（语境/上下文），8
  Description（描述），19
  Evolution（演化/进化），374
  Forces（强制条件/约束力），9
  Form（窗体/形式/表单），19
  Formalization（形式化/定型化），427
  Implementation（实现），23, 370
  Language（语言），360, 422
  Mining（采掘），352, 376, 420
  Problem（问题），8, 9
  Properties（属性），5
  Refinement（细化/精化），16
  Relationships（关系），16
  Selection（选择），368
  Solution（解决方案），8, 10
  System（系统），xiii, 22, 359, 360
  Variation（变体/变化），16
Pattern Home Page（模式主页），xiv, 417
Pattern Languages of Programming（程序设计的模式语言），416
Pattern Mailing Lists（模式邮递清单），417
Peer（对等/对等体），308, 319
Peer-to-peer（对等），438
Peer-to-peer Communication（对等通信），44, 307
Performance（性能），97, 191
Persistence（持久性），193, 195, 200, 204
Personality（个性），176
Pipe（管道），55, 56, 61
Platform（平台），438
PLoP（程序设计模式语言）。参见Pattern Languages of Programming
Plug'n Play（即插即用），170
Policy（策略），174, 182, 190, 401
Polling（轮询），339
Polymorphism（多态性/多态），354, 438
Portability（可移植性），120, 189, 260, 406
Portland Pattern Repository（波兰模式仓库），423
Pree, Wolfgang（人名），416
Presentation（表示），147, 158
Problem Category（问题分类），364, 369, 379
Problem-Solution Pair（问题-解决方案对），2, 3
Processing Pipeline（处理流水线/处理管线），55, 64
Producer-Consumer（生产者-消费者），342
Production System（生产系统），86
Programming Style（程序设计风格/编程风格），345, 346
Protection Proxy（保护代理），269
Protocol Stack（协议栈），37
Proxy（代理），264, 265
Proxy Server（代理服务器），271
pSather（一种语言），247
Publisher（出版者），339
Pull Model（拉-模型），40, 55, 61, 340
Push Model（推-模型），40, 55, 61, 340

Q

Quality-of-service（服务质量），342

R

Rapid Prototyping（快速原型），68
Real-time Constraint（实时限制），204
REBOOT（项目名），321
Receiver（接收器），310,317
Redo（重做），278,286
Reenskaug，Trygve（人名），403
Reference Counter（引用计数器），355
Reference Counting（引用计数），270,354,357
Regression Testing（回归测试），288
Relationship（关系），386,438
Reliability（可靠性），97,190,408
Remote Access（远程访问），305
Remote Procedure Call（远程过程调用），178，182，205，305,308,335
Remote Proxy（远程代理），268
Remoting of Interfaces（界面的远程化），111
Replay（回放），288
Repository（仓库），86,315
Request（请求），36,309
  Dispatching（分配），185
  Retrieval（检索），182
  Transmission（传输），182
Requirement Specification（需求规格说明），29
Resource（资源），
  Allocation（分配），205
  Handling（处理），174,365
  Management（管理），184
  Sharing（共享），305
  Utilization（利用），305
Response（响应），309
Responsibility（责任），438
Restructing（重构），406
Reuse（复用/重用），48,68,93,120,136,241,409
Robustness（健壮性），93,210,218,408
Role（角色），438
Rollback（回退/重新运行），288
Routing（路由选择），103
  Table（表），307
RPC（远程过程调用）。参见Remote Procedure Call
RTTI（运行期类型信息）。参见Run-time Type Information
Run-time Data Dictionary（运行期数据字典），216
Run-time Type Information（运行期类型信息），115，193，195,200,205,206,215

S

S.E.P（其他人的问题；软件工程过程；使用模式的软件过程），345,438
Sather（一种语言），43
Scaleability（可伸缩性），97,100,190
Scheduling（调度），277,278,288
Schmidt, Douglas C.（人名），xv,415
Scripting Language（脚本语言），287
SDL（一种语言），287,431
Semaphor（信号灯），270
Separation of Concerns（事务分离），165,241,259,400
Separation of Interface and Implementation（接口与实现的分离），402
Separation of Policy and Implementation（策略和实现的分离），401
Server（服务器），101,319,324,327,331,439
  Failure（失效/故障），324
  Registration（注册），103,324,331
  Registry（注册表），326
Server-side Proxy（服务器端代理），105,113
Service（服务），
  Extension（扩展），379
  Handler（处理程序），337
  Request（请求），175,183
  Variation（变体/变化），379
Shared Library（共享库），180,186
Shared Memory（共享内存），182,205,328,330
Shell（外壳/命令解释程序），63,64
SIMD（单指令多数据）。参见Single Instruction Multiple Data
Single Inheritance（单继承），439
Single Instruction Multiple Data（单指令多数据），255
Single Point of Reference（单一引用点），403
Slave（从属的/次要的），246,247,250,251
Smalltalk（一种语言），124，126，129，140，141，213，347，351,420
So Long, and Thanks for All the Fish（书名），383
Socket（套接字），45,308,315,328,330,332
  Descriptor（描述符），330
  Port（端口），330
Software Aging（软件老化），405
Software Architecture（软件体系结构），21,27,384,420
Software Design（软件设计），390
Software Process（软件过程），393
Solution Space（解空间），74,81

SOM（系统对象模型）。参见System Object Model
Speech Recognition（语音识别），71
SQL（结构查询语言），189
Standardization（标准化），48
Star Trek（星际旅行），419
Static Library（静态库），186
Step-wise Refinement（逐步细化），38
Structural Decomposition（结构分解/结构化分解），222、223，364
Structured Programming（结构化程序设计），391
Style Guide（风格指南），349
Subject（目标），339
Subscribers（订阅者），339
Subsystem（子系统），175，439
Sufficiency，Completeness and Primitiveness（充分性、完全性和基本性），401
Superclass（超类），439
Supplier（供应者），278，279，292，294
Synchronization（同步），141，289
Synchronization Proxy（同步代理），270
System（系统），439
  Evolution（演化/进化），194
  Family（族/系列），439
System Call（系统调用），175，184
System Object Model（系统对象模型），111，119
System Resource（系统资源），173
System-specific Dependencies（系统特定依赖性），174
System-unique Identifier（系统惟一标识符），113

T

TCP/IP（传输控制协议/因特网协议）。参见Transfer Control Protocol/Internet Protocol
Tee and Join（T型连接），66
Telecommunication（电信），421
Testability（可测试性），288，408
Testing（测试），94，121
The Hitchhiker's Guide to the Galaxy（书名），1
Thread（线程），62，183，248，257，330
Three-layer Architecture（三层体系结构），47，192
TLI（传输层接口）。参见Transport Layer Interface
Tool（工具），424
Top-down Approach（自顶向下方法），231
Trader（交易器），117
Transaction（事务（处理）），
  Control（控制），288
  Protocol（协议），204
Transfer Control Protocol/ Internet Protocol（传输控制协议／网际协议），32，44，305，314
Transparency（透明性），190
Transport Layer Interface（传输层接口），45，305
Traveling-salesman Problem（旅行商问题），245，247，250，253
Triple Modular Redundancy（三重模块冗余），243
Two-layer Architecture（二层体系结构），47，191

U

UDP（用户数据报协议）。参见User Datagram Protocol
Uncertain Knowledge（不确定知识），72
Undo（撤销），277，281，284，285
Unicode（统一的字符编码标准），313，439
UNIX（操作系统），46，63，67，97，171，186，189，305
  Filter（过滤器），61，62，66
  Pipe（管道），57，60，69
Unmarshaling（散集/复组），308
User Datagram Protocol（用户数据报协议），45
User Interface（用户界面/用户接口），126，146，169，199

V

Vertical Structuring（垂直结构），32
View（视图），127，128，134，291，292，294，297，387
View Coordination（视图协作），299
View Handler（视图处理程序），292，293，298
View Management（视图管理），293
Virtual Machine（虚拟机），53，192
Virtual Protocol（虚拟协议），44
Virtual Proxy（虚拟代理），270
Vlissides，John（人名），415

W

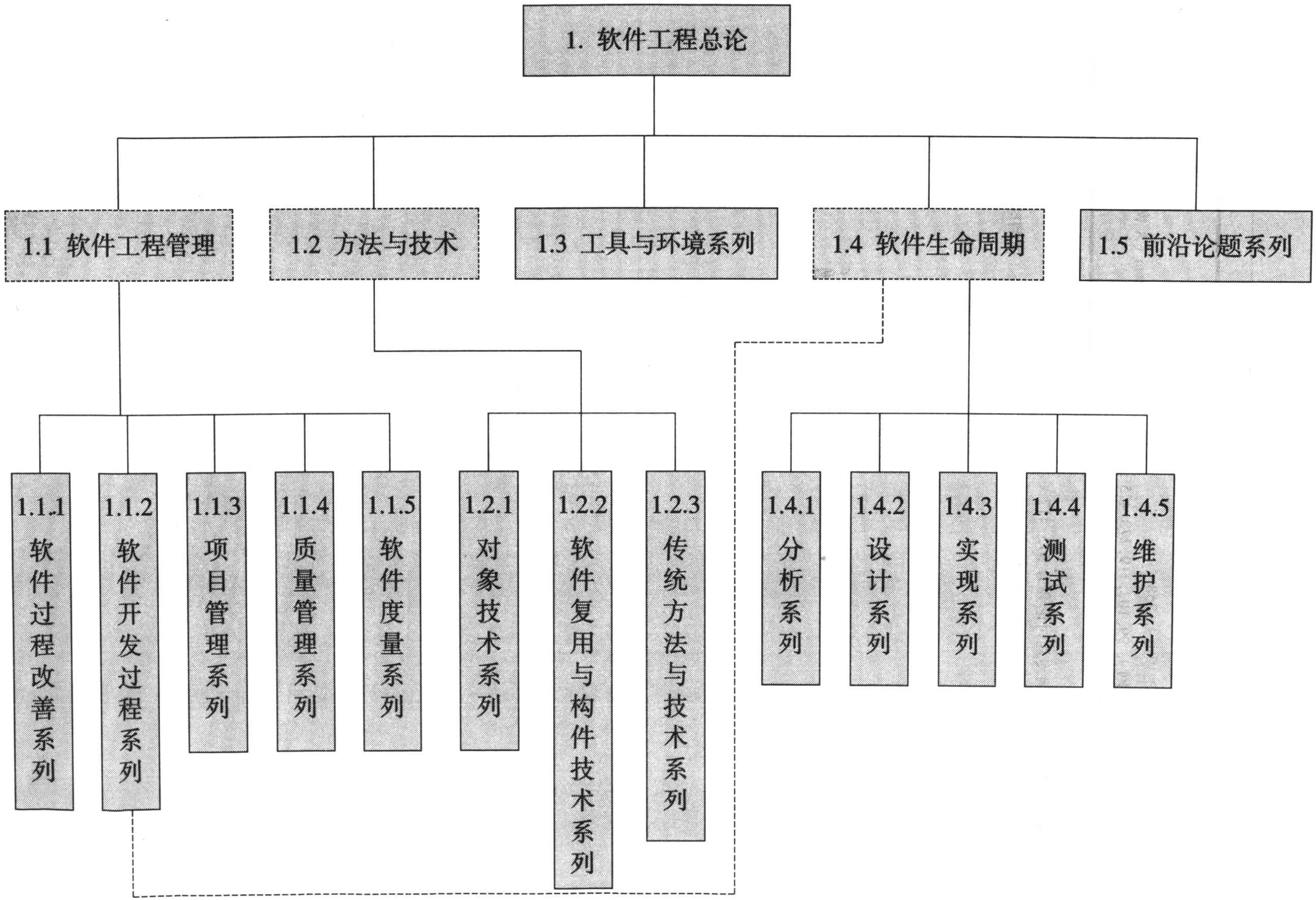
WAN（广域网）。参见Wide Area Network
White-box Approach（白盒方法），40

Whole（整体/部分），227, 231, 233
Wide Area Network（广域网），99
Workpool Model（工作池模型），258
World Wide Web（万维网），99, 102, 119, 273, 425
Wrapper（包装器），357
Wrapping（包装），143
Writer's Workshop（作者专题讨论会），375, 417
WWW（万维网）。参见World Wide Web

X

X Window System（X 窗口系统），46

Y

Yo-yo Approach（溜溜球方法），38

软件工程技术丛书结构图