PEARSON
Addison Wesley

面向对象分析与设计（第3版）

Object-Oriented Analysis and Design with Applications Third Edition

[美] Grady Booch Robert A. Maksimchuk
Michael W. Engle Bobbi J. Young, Ph.D. 著
Jim Conallen Kelli A. Houston

王海鹏 潘加宇 译

人民邮电出版社
POSTS & TELECOM PRESS

● 封面设计：胡萍丽

本书的英文原书长期以来都是面向对象技术的重要参考书，而面向对象技术已经成为主流软件开发的事实标准。在这本值得期待的第3版中，读者可以学会如何通过统一建模语言UML 2.0来应用面向对象方法。

Grady Booch是IBM院士（IBM fellow）和6本面向对象编程畅销书的作者。他是世界知名的面向对象技术发起人和UML创始人之一。

Robert A. Maksimchuk是Unisys Chief Technology Office研究部门的主管，关注新兴的建模技术。他是《UML数据库设计应用》和《UML初学者指南》的合著者之一。

Michael W. Engle是Lockheed Martin公司工程部门的主要成员。在许多不同的应用领域，他对完整的系统开发过程都具有丰富的技术和管理经验，包括从项目启动一直到部署和支持。作为一名系统架构师，他在复杂系统的开发中使用面向对象分析与设计技术。

Bobbi J. Young, Ph.D.是Unisys Chief Technology Office研究部门的主管。她在IT行业有着多年的经验，供职于一些商业公司和美国国防部的签约公司。

Jim Conallen是IBM Rational的Model Driven Development Strategy团队的一名软件工程师，他积极参与了将Object Management Group（OMG）的Model Driven Architecture（MDA）应用于IBM Rational的建模工具的工作。

Kelli A. Houston是IBM Rational的一名IT咨询专家。她是IBM的方法架构师，也是负责集成IBM方法的团队成员。

本书新增的内容包括：

- UML 2.0的详细介绍，包括这种表示法最基本的元素和高级元素，同时强调了重要的区别；
- 用5章的篇幅大大加强了对建模的关注，每一章都强调了整体开发生命周期中的一个阶段；
- 让复杂系统变得有条理的新方法，包括利用OOAD和UML来处理系统架构的新方法；
- 关于对象模型中一些被广泛误解的基本元素的基础概念，如抽象、封装、模块化和层次结构；
- 关于如何分配开发团队资源以及管理开发复杂软件系统相关风险的建议；
- 关于主要面向对象编程语言（如Java和C++等）的附录。

ISBN 978-7-115-20837-8

9 787115 208378 >

ISBN 978-7-115-20837-8/TP

定价：79.00 元

分类建议：计算机 / 软件工程

人民邮电出版社网址：www.ptpress.com.cn

面向对象分析与设计

（第3版）

[美] Grady Booch Robert A. Maksimchuk
Michael W. Engle Bobbi J. Young, Ph.D. 著
Jim Conallen Kelli A. Houston

UMLChina 王海鹏 潘加宇 译

人民邮电出版社
北京

图书在版编目（CIP）数据

面向对象分析与设计：第3版 /（美）布奇（Booch, G.）等著；王海鹏，潘加宇译. —北京：人民邮电出版社，2009.8
ISBN 978-7-115-20837-8

Ⅰ. 面… Ⅱ. ①布…②王…③潘… Ⅲ. 面向对象语言－程序设计 Ⅳ. TP312

中国版本图书馆CIP数据核字（2009）第071216号

版权声明

Authorized translation from the English language edition, entitled Object-Oriented Analysis and Design with Applications (Third Edition), 9780201895513 by Grady Booch, Robert A. Maksimchuk, Michael W. Engle, Bobbi J. Young, Ph.D., Jim Conallen, Kelli A. Houston, published by Pearson Education, Inc, publishing as Addison Wesley Professional, Copyright © 2007 by Pearson Education, Inc.
All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc. CHINESE SIMPLIFIED language edition published by PEARSON EDUCATION ASIA LTD., and POSTS & TELECOMMUNICATIONS PRESS Copyright © 2009.
本书封面贴有 Pearson Education（培生教育出版集团）激光防伪标签。无标签者不得销售。

面向对象分析与设计（第3版）

◆ 著　[美] Grady Booch　Robert A. Maksimchuk
Michael W. Engle　Bobbi J. Young, Ph.D.
Jim Conallen　Kelli A. Houston

译　王海鹏　潘加宇

责任编辑　刘映欣

◆ 人民邮电出版社出版发行　北京市崇文区夕照寺街14号
邮编　100061　电子函件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京顺义振华印刷厂印刷

◆ 开本：800×1000　1/16
印张：32.75
字数：732千字　2009年8月第1版
印数：1－3 000册　2009年8月北京第1次印刷

著作权合同登记号　图字：01-2007-5294号
ISBN 978-7-115-20837-8/TP

定价：79.00元

读者服务热线：(010)67132705　印装质量热线：(010)67129223
反盗版热线：(010)67171154

内容提要

本书是 UML 创始人 Grady Booch 的代表作之一，书中介绍的概念都基于牢固的理论基础。同时，本书又是一本注重实效的书，面向架构师和软件开发者等软件工程实践者的实际需要。本书通过大量例子说明了基本概念，解释了方法，并展示了在不同领域的成功应用。全书分为理论和应用两部分。理论部分深刻剖析了面向对象分析与设计（OOAD）的概念和方法。应用部分连续列出了 5 个不同类型、不同领域的应用，描述如何从初始阶段到移交阶段将 OOAD 理论和方法应用到项目中。应用部分所涉及的领域包括系统架构、数据获取、密码分析、控制系统和 Web 开发，还给出了一些关于重要问题的有效建议，包括分类、实现策略和高性价比的项目管理。书中的表示法采用最新的 UML 2.0，因此本书是学习 UML 2.0 不可多得的参考书。

本书作者基于长期丰富的经验，提出了改进的对象开发方法，用于解决系统和软件开发者面临的复杂问题，非常适合实际系统和软件的开发者、系统分析师或构架师、项目经理阅读。本书主要阐述了软件开发的方法，也可以作为高等院校软件工程和高级编程课程的教材使用。

特别的 Booch，特别的书

以下几点也许是您购买本书的理由。

1．读过 Robert C. Martin 的 *Agile Principles* 的读者，很容易注意到该书前言的第一句话，“Bob，你说过去年就能写完这本书的——Claudia Frers 在 1999 年 UML World 大会上抱怨。”“这本书”指的是 Robert C. Martin 在 20 世纪 90 年代的代表作 *Designing Object-Oriented C++ Application using the Booch Method*。*Agile Principles* 本来是作为 Designing 一书的第 2 版。由此可见，Robert C. Martin 深受 Booch 的影响。

2．Grady Booch 是最早提出面向对象分析设计方法的方法学家之一。20 多年来，Grady Booch 一直担任 Rational 公司的首席科学家，随着 Rational 成长，收购别人，被别人收购，CEO 换了又换，他也没有离开。Grady Booch 是 UML 三友中唯一的 IBM 院士。

3．Grady Booch 非常“与时俱进”，其 Blog（http://www.ibm.com/developerworks/blogs/page/gradybooch）更新频繁，即使在心脏主动脉瘤动手术期间，他也一直在病床上更新 Blog。他还热衷于 Second Life，认为 Second Life 是一条协作软件开发的新道路。Grady Booch 在 Second Life 上的化身叫做 Alem Theas。Dr. Dobb 在给他颁发程序设计杰出奖（Excellence in Programming Award）的时候，就是在 Second Life 上给 Alem Theas 颁奖。

4．本书是真正展示 Booch 思想的书。我们可以在 UML 三友署名的 UML 系列书籍封面上看到 Grady Booch 的名字，但其中大多数工作是由 James Rumbaugh 完成的。

5．本书分为理论和应用两部分。理论部分（概念和方法）的叙述朴实无华，标题简洁：“复杂性”、“对象模型”、“类和对象”、“分类”、“表示法”、“过程”……用平实的语言把内容剖析得很透彻。应用部分连续列出了 5 个不同类型、不同领域的应用，描述如何从初始阶段到移交阶段把前面所授方法应用到项目中。

- 基于卫星的导航系统　聚焦于系统架构；
- 列车交通控制系统　聚焦于系统需求；
- 人工智能解密系统　聚焦于分析；
- 气象站数据采集系统　聚焦于分析到初步的设计；
- 雇员休假跟踪 Web 应用系统　聚焦于详细设计和实现。

6．书中的表示法采用最新的 UML 2.0，画图工具是 IBM Rational Software Architect 和 Sparx Systems Enterprise Architect。也就是说，Grady Booch 使用了非 IBM 公司的 UML 工具来写自己的书。

潘加宇

2009 年 5 月

前　言

人类渴望得到精神上的宁静、美学上的成就、家庭的安全、正义和自由。这一切都不能通过工业化的生产效率来直接满足。但是，生产效率让人们得到充足的物质享受，而不至于与匮乏苦苦斗争。这为精神、美学和家庭事务赢得了时间，也使得社会能够将一些特殊的技能赋予司法机构以及维护权利的机构。

Harlan Mills
DPMA and Human Productivity

作为计算机专业人员，我们努力地去构建能工作而且有用的系统；作为软件工程师，我们面临着在计算资源和人力资源有限的条件下创建复杂系统的任务。面向对象（OO）技术已经发展为管理许多不同种类的系统中内在复杂性的手段。对象模型已被证明是非常有力和统一的概念。

对第 2 版的改动

在本书第 2 版出版以后，我们看到了一些重要的技术进步，其中一些突出的进步如下。

- 与因特网的高带宽、无线连接已经非常普遍；
- 纳米技术已经出现，并开始提供有价值的产品；
- 机器人在火星表面漫步；
- 计算机生成的特效使得在电影中能够完全逼真地再现任何想象中的世界；
- 出现了个人气垫船；
- 手机已无处不在，使用非常方便；
- 获得了人类基因图谱；
- 面向对象技术已经在工业软件开发中成为主流技术。

在世界各地都能见到面向对象技术被使用。但是，我们仍然遇到许多人，他们还没有采用面向对象的开发方式。对于这两类人，本书的新版本都很有价值。

对于面向对象分析与设计（OOAD）的新手，本书提供了下列信息。

- 面向对象的概念支持和演进式的观点；
- 如何在系统开发生命周期中应用 OOAD 的例子；
- 对系统和软件开发中使用的标准表示法统一建模语言（UML 2.0）的介绍。

对于有经验的 OOAD 实践者，本书从不同的角度提供了价值。

- 即使对于有经验的实践者，UML 2.0 也是新的。这里可以看到表示法方面的重要区别。
- 根据前一版本所收到的反馈，更加关注建模。
- 通过本书的概念部分的学习，可以了解在面向对象的世界中，“为什么事情是像它们现在的样子”。许多人可能从没研究过面向对象（OO）概念本身的发展，即使有所了解，在初次学习 OO 方法时，也许未能理解其重要性。

本书这一版和以前的版本有 4 项主要区别，如下所示。

- UML 2.0 已经正式得到了通过，第 5 章将介绍 UML 2.0。为了加强读者对这种表示法的理解，特别区分了它的基本元素和高级元素。
- 这一版在应用程序的章节中引入了一些新的领域和背景。例如，应用程序的领域范围很广，包括从高级系统架构到基于 Web 的系统的设计细节等各种不同层次的抽象。
- 在前一版出版时，作为 OO 编程的概念来说，C++相对还是比较新的。读者告诉我们，这种强调不再是主要的考虑。现在有大量的 OO 编程和技术书籍及培训，还有许多为 OO 开发而设计的编程语言。因此，大部分关于编码的讨论被删除了。
- 最后，响应读者的要求，这一版更关注 OOAD 建模方面。应用程序章节将展示如何利用 UML，其中每一章强调了整个开发生命周期中的一个阶段。

本书的目标

本书在面向对象系统构建方面提供了实用指导。它的具体目标是：

- 提供对对象模型的基础概念及其发展变化的正确理解；
- 帮助读者掌握面向对象分析和设计的表示法和过程；
- 介绍在不同的问题域中面向对象分析和设计的实际应用。

本书介绍的概念都基于牢固的理论基础，但本书首先是一本注重实效的书，面向架构师和软件开发者等软件工程实践者的实际需要。

读者对象

本书既是为计算机专业人员也是为学生编写的。

- 对于实际系统和软件的开发者，本书将展示如何高效地利用面向对象技术来解决实

际问题。

- 对于系统分析师或架构师，本书将利用面向对象的分析与设计，提供一条从需求到实现的途径。我们帮助分析人员或架构师提高识别能力，以区分不好的面向对象的结构与好的面向对象的结构，并在现实情况反常时权衡可选的设计方案。也许最重要的就是，我们提供了一些让复杂系统变得有条理的新方法。
- 对于项目经理，本书可以帮助他们更好地理解开发团队的资源分配、软件品质、测量指标以及管理与复杂软件系统相关的风险。
- 对于学生，本书提供了一些必要的指导，使得学生能够开始掌握复杂系统开发的科学与艺术中的一些重要技巧。

本书不仅适合于专业研讨班和个人学习使用，也适合作为高等院校本科生和研究生课程的教材。因为它主要阐述了软件开发的方法，所以非常适合软件工程和高级编程等课程，也可以作为涉及具体面向对象编程语言的课程的补充阅读材料。

本书的组织结构

本书分成 3 个主要部分：概念、方法和应用，其中穿插了大量的补充材料。

概念

第 1 部分研究软件的内在复杂性及其表现方式。本书将对象模型作为一种手段来帮助我们管理这种复杂性，详细地研究了对象模型的基本元素——抽象、封装、模块化、层次结构，讨论了“什么是类”以及“什么是对象”等基本问题。由于确定有意义的类和对象是面向对象开发中的关键任务，因此我们花了相当多的时间来研究分类的本质。具体来说，我们研究了生物学、语言学和心理学等其他学科中的分类方法，然后将这些经验应用到发现软件系统中类和对象的问题上。

方法

第 2 部分基于对象模型提出了复杂系统开发的一种方法。针对面向对象的分析与设计，首先提出了一套图形表示法（即 UML），然后是一个通用的过程框架。还研究了面向对象开发的实践，具体来说，就是它在软件开发生命周期中的位置以及它对于项目管理意味着什么。

应用程序

第 3 部分提供了一组 5 个不简单的例子，涉及不同问题域：系统架构、控制系统、密码分析、数据获取和 Web 开发。之所以选择这些问题域，是因为它们是软件工程师实践过程中遇到的复杂问题的代表。展示某些原则如何应用于简单的问题是很容易的，但是因为我们关注的是为现实世界构建有用的系统，所以我们对如何将对象模型应用于复杂应用程序更加感

兴趣。软件系统的开发不同于按菜谱做菜，因此我们强调应用程序的增量式开发，这种开发以一些正确的原则和良好的模型作为指导。

补充材料

本书中穿插了大量的补充资料。多数章节中都有补充材料，这些材料对重要的主题提供了相关的信息。本书包括了一个关于面向对象编程语言的附录，其中总结了一些常见语言的特征；还提供了常用术语的词汇表，以及一个扩展的分类参考书目，列出了关于对象模型的参考资料。

工具说明

读者总是会问创建本书中的图使用了什么工具。我们主要使用两个很好的工具来画图：IBM Rational Software Architect 和 Sparx Systems Enterprise Architect。为什么不只用一个？市场的实际情况是，没有哪一种工具可以做所有的事情。实践 OOAD 的时间越长，最后就会发现有些特别的情况是所有工具都不支持的。（在这种情况下，可能需要寻求基本的绘图工具来展示你的想法。）但是，不要让这些很少的情况阻止你使用健壮的 OOAD 工具，如我们提到的这两种工具。

本书的阅读方法

本书可以一页一页地读，也可以不按现有的组织形式阅读。如果想对对象模型中的基本概念或面向对象开发的动机有较深的理解，那么就应该从第 1 章开始依次读下去。如果只对面向对象开发分析与设计中的表示法和过程感兴趣，就从第 5 章和第 6 章开始阅读。第 7 章对使用这种方法管理项目的管理者来说特别有用。如果对针对特定问题域的面向对象技术的应用程序更感兴趣，则可以从第 8～12 章中任选一章或者全部阅读。

致　谢

我把本书献给我的妻子 Jan，感谢她的爱和支持。

在第 1 版和第 2 版的写作过程中，一些人促成了我的面向对象开发思想。对于他们的贡献，我特别要感谢，他们是：Sam Adams、Mike Akroid、Glenn Andert、Sid Bailin、Kent Beck、Dave Bernstein、Daniel Bobrow、Dick Bolz、Dave Bulman、Kayvan Carun、Dave Collins、Damian Conway、Steve Cook、Jim Coplien、Brad Cox、Ward Cunningham、Tom DeMarco、Mike Devlin、Richard Gabriel、William Genemaras、Adele Goldberg、Ian Graham、Tony Hoare、Jon Hopkins、Michael Jackson、Ralph Johnson、James Kempf、Norm Kerth、Jordan Kreindler、Doug Lea、Phil Levy、Barbara Liskov、Cliff Longman、James MacFarlane、Masoud Milani、Harlan Mills、Robert Murray、Steve Neis、Gene Ouye、Dave Parnas、Bill Riddel、Mary Beth Rosson、Kenny Rubin、Jim Rumbaugh、Kurt Schmucker、Ed Seidewitz、Dan Shiffman、Dave Stevenson、Bjarne Stroustrup、Dave Thomas、Mike Vilot、Tony Wasserman、Peter Wegner、Iseult White、John Williams、Lloyd Williams、Niklaus Wirth、Mario Wolczko 和 Ed Yourdon。

本书的相当一部分实践来自我参与并在世界各地开发的复杂软件系统，这些系统的开发公司包括 Alcatel、Andersen Consulting、Apple、AT&T、Autotrol、Bell Northern Research、Boeing、Borland、Computer Sciences Corporation、Contel、Ericsson、Ferranti、General Electric、GTE、Holland Signaal、Hughes Aircraft Company、IBM、Lockheed、Martin Marietta、Motorola、NTT、Philips、Rockwell International、Shell Oil、Symantec、Taligent 和 TRW。我曾有机会与数百名专业软件工程师和他们的经理合作，我要谢谢他们的帮助，是他们让本书与真实世界的问题相关。

特别要感谢 Rational 对我的工作的支持。还要谢谢 Tony Hall，他的卡通画给本书带来了亮点，否则这本书就只是一本乏味的技术书籍。最后，我要谢谢我的 3 只猫——Cammy、Annie 和 Shadow，在我写作的许多个深夜，它们总是陪伴着我。

——Grady Booch

我要感谢我的家人，他们必须忍受我参与编写这本书的漫长日子。感谢我的父母，他们培育了我高尚的职业道德。感谢 Mary T.O' Brien，她为我提供了这个机会，这才使我开始了大量的后续工作。感谢 Chris Guzikowski 帮助推动这项工作直至完成。我要感谢合著者，感谢你们允许我加入这项工作，也感谢你们在这个项目中的努力工作和贡献。最后，我要衷心

感谢 Grady 多年前编写的本书的第 1 版，这本书是关于面向对象分析与设计最早的、最基础的书之一。

——Bob Maksimchuk

我想表达对家人的感激，他们给了我爱和支持，这是我所有努力的基础。感谢 Grady 给我机会，让我能够在他的经典著作的第 3 版中做出贡献。最后，我要感谢 Bob Maksimchuk 在我成为一名作者的过程中所给予的指导。

——Mike Engle

我要将本书献给我的母亲 Jean Smith，她鼓励我参加这项工作。我也要表达我对家人 Russell、Alyssa 和 Logan 的爱和感激，感谢他们的支持和鼓励。感谢 Bob Maksimchuk 和 Mike Engle，是他们让我有机会参与这项工作。

——Bobbi J. Young

我要特别感谢我的丈夫 Bob 和两个孩子——Katherine 和 Ryan，他们的爱和支持给了我真正的灵感。

——Kelli A. Houston

感谢我们的审稿者，特别是 Davyd Norris 和 Brian Lyons。感谢 Addison-Wesley 所有参与本书的其他工作人员，特别是 Chris Zahn，他不仅参与了这项工作，而且保持了这项长时间的工作的连贯性。

作者简介

Grady Booch 在软件架构、软件工程和建模领域的创新工作是世界知名的。从 1981 年 Rational 公司创建开始，他就一直担任该公司的首席科学家。Grady 于 2003 年 3 月成为了 IBM 院士（IBM Fellow）。

Grady 是统一建模语言（UML）最早的开发者之一，也是几个 Rational 产品的最早开发者之一。Grady 曾担任世界各地一些复杂的软件密集型项目的架构师和架构指导者。

Grady 是 6 本畅销书的作者，包括 *UML Users Guide* 和 *Object-Oriented Analysis with Applications*。Grady 发表了几百篇有关软件工程的技术文章，其中包括在 20 世纪 80 年代早期发表的文章，这些文章最先提出了面向对象设计的术语和实践。他曾在世界各地演讲和咨询。

Grady 是美国计算机协会（ACM）、美国电气电子工程师学会（IEEE）、美国科学促进会（AAAS）、有社会责任的计算机专家协会（CPSR）的成员。他是 IBM 院士、ACM 院士、世界技术网络院士，也是软件开发论坛梦想家。Grady 是敏捷联盟、Hillside 集团和软件架构师世界学院的创始委员会成员，也是 Northface 大学的顾问委员会成员。

Grady 于 1977 年从美国空军学院获得学士学位，于 1979 年从加州大学圣巴巴拉分校获得电子工程科学硕士学位。

Grady 与他的妻子和他的猫生活在科罗拉多。他的兴趣包括阅读、旅行、唱歌和弹奏竖琴。

Robert A. Maksimchuk 是 Unisys Chief Technology Office 的一名研究主管。他关注新出现的建模技术，目的是提升 Unisys 3D—可视企业建模框架的战略方向。Bob 为这项任务带来了不同行业的大量系统工程、建模、面向对象分析与设计的专业知识。他是 *UML for Mere Mortals* 和 *UML for Database Design* 的合著者，也写了许多文章。他曾经周游世界各地，在各种技术论坛上作为重要演讲者发言，举办关于 UML 和面向对象开发的研讨会和培训。Bob 是电气电子工程师学会（IEEE）和国际系统工程学会（INCOSE）的成员。

Michael W. Engle 是洛克希德马丁公司的首席工程师。他有超过 26 年的技术和管理经验——从项目启动到运营支持，涵盖了完整的系统开发生命周期。利用系统工程师、软件工程师和系统架构师的背景，Mike 运用了面向对象技术，为复杂的系统开发提供创新的开发

方式。

Bobbi J. Young, Ph.D.是 Unisys Chief Technology Office 的一名研究主管。她有着多年的 IT 行业从业经验，与商业公司和国防部合同供应商一同工作。Young 博士是--名咨询师，她在项目管理、企业架构、系统工程和面向对象分析与设计方面提供现场指导。在她的职业生涯中，她关注于系统生命周期过程和方法学，同时也关注企业架构。Young 博士拥有生物学、计算机科学和人工智能学位，她获得了管理信息系统的博士学位，也曾是美国海军预备役的一名指挥官（已退伍）。

Jim Conallen 是 IBM Rational 的模型驱动开发战略小组的一名软件工程师。在这个小组中，他积极参与，将对象管理集团（OMG）的模型驱动架构（MDA）计划应用于 IBM Rational 的模型工具中。Jim 在基于资产的开发和可复用资产规范（RAS）领域也很活跃。Jim 经常在会议上演讲，也经常写文章。他的专业领域是 Web 应用开发。

他开发了 UML 的 Web 应用扩展（WAE）。这是对 UML 的一种扩展，让开发者能够利用 UML 在合适的抽象和细节层面上对 Web 应用的架构进行建模。这项工作是 IBM Rational Rose 和 Rational XDE Web Modeling 功能的基础。

Jim 与人合著了两个版本的 *Building Web Applications with UML*，第一个版本采用微软公司的 ASP 技术，后一个版本采用 J2EE 技术。

Jim 的经验也来自于加入 Rational 之前的工作，那时他曾是独立的咨询师、Peace Corps 的自愿者和大学讲师。他还是 3 个孩子的父亲。Jim 从 Widener 大学获得了计算机和软件工程的学士学位和硕士学位。

Kelli Houston 是 IBM Rational 的 IT 咨询专家。她是 IBM 内部方法的方法架构师，负责编写方法并集成 IBM 的方法。除了方法架构师的角色，Kelli 还在 IBM 内部领导 Rational Method Composer（RMC）特别兴趣小组（SIG）工作，为客户和 IBM 内部咨询师提供有效使用 RMC 方面的咨询和现场指导服务。

目　录

第1部分　概　念

第1章　复杂性 …… 2
1.1　复杂系统的结构 …… 2
1.1.1　个人计算机的结构 …… 3
1.1.2　植物和动物的结构 …… 3
1.1.3　物质的结构 …… 4
1.1.4　社会机构的结构 …… 4
1.2　软件固有的复杂性 …… 5
1.2.1　定义软件复杂性 …… 5
1.2.2　为什么软件在本质上是复杂的 …… 5
1.3　复杂系统的5个属性 …… 8
1.3.1　层次结构 …… 8
1.3.2　相对本原 …… 8
1.3.3　分离关注 …… 9
1.3.4　共同模式 …… 9
1.3.5　稳定的中间形式 …… 9
1.4　有组织和无组织的复杂性 …… 10
1.4.1　复杂系统的规范形式 …… 10
1.4.2　处理复杂性时人的能力的局限 …… 12
1.5　从混沌到有序 …… 12
1.5.1　分解的作用 …… 13
1.5.2　抽象的作用 …… 16
1.5.3　层次结构的作用 …… 16
1.6　复杂系统的设计 …… 17
1.6.1　作为科学和艺术的工程 …… 17
1.6.2　设计的含义 …… 17
1.7　小结 …… 19

第 2 章 对象模型 ····· 20
2.1 对象模型的演进 ····· 20
2.1.1 程序设计语言的换代 ····· 20
2.1.2 第一代和第二代早期程序设计语言的拓扑结构 ····· 22
2.1.3 第二代后期和第三代早期程序设计语言的结构 ····· 23
2.1.4 第三代后期程序设计语言的结构 ····· 23
2.1.5 基于对象和面向对象的程序设计语言的结构 ····· 24
2.2 对象模型基础 ····· 26
2.2.1 面向对象编程 ····· 29
2.2.2 面向对象设计 ····· 29
2.2.3 面向对象分析 ····· 30
2.3 对象模型要素 ····· 30
2.3.1 抽象的意义 ····· 31
2.3.2 封装的意义 ····· 35
2.3.3 模块化的意义 ····· 38
2.3.4 层次结构的意义 ····· 41
2.3.5 类型的意义 ····· 46
2.3.6 并发的意义 ····· 47
2.3.7 持久的意义 ····· 49
2.4 应用对象模型 ····· 51
2.4.1 对象模型的好处 ····· 51
2.4.2 开放式问题 ····· 52
2.5 小结 ····· 52
第 3 章 类与对象 ····· 53
3.1 对象的本质 ····· 53
3.1.1 什么是对象，什么不是对象 ····· 53
3.1.2 状态 ····· 55
3.1.3 行为 ····· 57
3.1.4 标识符 ····· 60
3.2 对象之间的关系 ····· 62
3.2.1 链接 ····· 62
3.2.2 聚合 ····· 64
3.3 类的本质 ····· 65
3.3.1 什么是类，什么不是类 ····· 65
3.3.2 接口和实现 ····· 66
3.3.3 类的生命周期 ····· 68

3.4 类之间的关系 …… 68
3.4.1 关联 …… 69
3.4.2 继承 …… 70
3.4.3 聚合 …… 78
3.4.4 依赖关系 …… 79
3.5 类与对象的互动 …… 79
3.5.1 类与对象的关系 …… 79
3.5.2 类与对象在分析和设计中的角色 …… 79
3.6 创建高品质的类与对象 …… 80
3.6.1 评判一种抽象的品质 …… 80
3.6.2 选择操作 …… 81
3.6.3 选择关系 …… 82
3.6.4 选择实现 …… 83
3.7 小结 …… 84
第4章 分类 …… 85
4.1 正确分类的重要性 …… 85
4.1.1 分类的困难 …… 86
4.1.2 分类的增量和迭代本质 …… 87
4.2 确定类和对象 …… 88
4.2.1 经典方法和现代方法 …… 89
4.2.2 面向对象分析 …… 92
4.3 关键抽象与机制 …… 97
4.3.1 确定关键抽象 …… 97
4.3.2 识别机制 …… 99
4.4 小结 …… 101

第2部分 方 法

第5章 表示法 …… 104
5.1 统一建模语言 …… 104
5.1.1 简单历史回顾 …… 104
5.1.2 模型与多重视图 …… 105
5.1.3 图分类 …… 105
5.1.4 在实践中使用图 …… 107
5.1.5 概念模型、逻辑模型和物理模型 …… 108
5.1.6 工具的角色 …… 108
5.1.7 面向对象开发的产品 …… 108

5.1.8 规模上的伸缩 …… 109
5.1.9 UML 的语法和语义 …… 109
5.1.10 UML 2.0 信息资源 …… 110
5.2 包图 …… 110
5.2.1 基本概念：包表示法 …… 110
5.2.2 基本概念：元素的可见性 …… 111
5.2.3 基本概念：依赖关系 …… 111
5.2.4 基本概念：包图 …… 113
5.2.5 高级概念：导入和访问 …… 114
5.3 组件图 …… 117
5.3.1 基本概念：组件表示法 …… 117
5.3.2 基本概念：组件图 …… 118
5.3.3 基本概念：组件接口 …… 119
5.3.4 基本概念：组件实现 …… 121
5.3.5 高级概念：组件的内部结构 …… 122
5.4 部署图 …… 123
5.4.1 基本概念：工件表示法 …… 123
5.4.2 基本概念：节点表示法 …… 124
5.4.3 基本概念：部署图 …… 124
5.5 用例图 …… 126
5.5.1 基本概念：执行者 …… 126
5.5.2 基本概念：用例 …… 127
5.5.3 基本概念：用例图 …… 127
5.5.4 高级概念：<<include>>和<<extend>>关系 …… 129
5.5.5 高级概念：泛化 …… 133
5.6 活动图 …… 133
5.6.1 基本概念：动作 …… 134
5.6.2 基本概念：开始和停止 …… 134
5.6.3 基本概念：判断节点和合并节点 …… 134
5.6.4 基本概念：分区 …… 135
5.6.5 高级概念：分叉、结合和并发 …… 135
5.6.6 高级概念：对象流 …… 138
5.6.7 高级概念：其他元素 …… 138
5.7 类图 …… 138
5.7.1 基本概念：类表示法 …… 138
5.7.2 基本概念：类关系 …… 140

5.7.3 高级概念：模板（参数化）类 …… 142
5.7.4 高级概念：可见性 …… 143
5.7.5 高级概念：关联端名称和限定符 …… 144
5.7.6 高级概念：约束 …… 145
5.7.7 高级概念：关联类和注解 …… 148
5.8 序列图 …… 149
5.8.1 基本概念：对象与交互 …… 149
5.8.2 基本概念：生命线与消息 …… 149
5.8.3 高级概念：销毁事件 …… 151
5.8.4 高级概念：执行说明 …… 151
5.8.5 高级概念：交互使用 …… 153
5.8.6 高级概念：控制结构 …… 153
5.9 交互概述图 …… 155
5.9.1 基本概念：框 …… 155
5.9.2 基本概念：控制流元素 …… 155
5.9.3 基本概念：交互图元素 …… 156
5.10 组合结构图 …… 157
5.10.1 基本概念：组合结构的部分 …… 157
5.10.2 基本概念：组合结构的部分与接口 …… 157
5.10.3 基本概念：组合结构连接器 …… 158
5.10.4 高级概念：协作 …… 158
5.11 状态机图 …… 159
5.11.1 基本概念：初始状态、最终状态和简单状态 …… 160
5.11.2 基本概念：转换与事件 …… 160
5.11.3 高级概念：状态活动——入口活动、执行活动和出口活动 …… 162
5.11.4 高级概念：控制转换 …… 162
5.11.5 高级概念：复合状态与嵌套状态 …… 163
5.11.6 高级概念：并发与控制 …… 165
5.11.7 高级概念：子状态机状态 …… 168
5.11.8 高级概念：其他状态机图元素 …… 169
5.12 时间图 …… 170
5.12.1 基本概念：更多相同之处 …… 170
5.12.2 基本概念：布局 …… 170
5.12.3 基本概念：事件 …… 171
5.12.4 基本概念：约束 …… 171
5.12.5 高级概念：另一种表示形式 …… 172

5.12.6 高级概念：事件与消息 …… 172
5.13 对象图 …… 173
5.13.1 基本概念：对象 …… 173
5.13.2 基本概念：对象关系 …… 173
5.13.3 高级概念：端点名称和限定符 …… 174
5.14 通信图 …… 175
5.14.1 基本概念：对象、链接和消息 …… 175
5.14.2 基本概念：顺序表达式 …… 176
5.14.3 高级概念：消息与同步 …… 177
5.14.4 高级概念：迭代子句和警戒条件 …… 178
5.15 小结 …… 179
第 6 章 过程 …… 181
6.1 首要原则 …… 181
6.1.1 成功项目的特征 …… 181
6.1.2 追求理性的开发过程 …… 184
6.2 宏观过程：软件开发生命周期 …… 186
6.2.1 概述 …… 187
6.2.2 宏观过程的内容维：科目 …… 188
6.2.3 宏观过程的时间维：里程碑和阶段 …… 190
6.2.4 宏观过程的时间维：迭代 …… 195
6.2.5 发行计划 …… 196
6.3 微观过程：分析与设计过程 …… 198
6.3.1 概述 …… 198
6.3.2 抽象层次 …… 199
6.3.3 活动 …… 200
6.3.4 产品 …… 201
6.3.5 微观过程与抽象层次 …… 203
6.3.6 识别元素 …… 205
6.3.7 确定元素间的协作 …… 208
6.3.8 确定元素间的关系 …… 211
6.3.9 详细确定元素的语义 …… 213
6.4 小结 …… 216
第 7 章 实战 …… 217
7.1 管理和计划 …… 217
7.1.1 风险管理 …… 218
7.1.2 任务计划 …… 218

7.1.3 开发评审 …… 219
7.2 人员配备 …… 220
7.2.1 资源配置 …… 220
7.2.2 开发团队角色 …… 221
7.3 发布版本管理 …… 223
7.3.1 配置管理和版本控制 …… 223
7.3.2 集成 …… 224
7.3.3 测试 …… 224
7.4 复用 …… 225
7.4.1 复用的元素 …… 225
7.4.2 建立复用制度 …… 225
7.5 质量保证和度量 …… 226
7.5.1 软件质量 …… 226
7.5.2 面向对象度量 …… 226
7.6 文档化 …… 229
7.6.1 开发遗产 …… 229
7.6.2 文档化的内容 …… 229
7.7 工具 …… 230
7.7.1 工具种类 …… 230
7.7.2 组织上的意义 …… 231
7.8 特殊主题 …… 231
7.8.1 领域特定问题 …… 231
7.8.2 采纳面向对象技术 …… 232
7.9 面向对象开发的好处和风险 …… 233
7.9.1 面向对象开发的好处 …… 233
7.9.2 面向对象开发的风险 …… 233
7.10 小结 …… 235

第3部分 应 用

第8章 系统架构——基于卫星的导航 …… 238
8.1 先启 …… 238
8.1.1 卫星导航系统的需求 …… 239
8.1.2 定义问题的边界 …… 240
8.1.3 决定使命用例 …… 243
8.1.4 决定系统用例 …… 245
8.2 精化 …… 248

8.2.1 开发一个好的架构 …… 248
8.2.2 定义架构开发活动 …… 248
8.2.3 验证所建议的系统架构 …… 249
8.2.4 分配非功能需求和阐明接口 …… 255
8.2.5 规定系统架构及其部署 …… 257
8.2.6 分解系统架构 …… 259
8.3 构造 …… 265
8.4 后移交 …… 265
8.4.1 添加新的功能 …… 265
8.4.2 改变目标硬件 …… 266
第 9 章 控制系统——交通管理 …… 267
9.1 先启 …… 268
9.1.1 列车交通管理系统的需求 …… 268
9.1.2 决定系统用例 …… 270
9.2 精化 …… 273
9.2.1 分析系统功能 …… 274
9.2.2 定义 TTMS 架构 …… 277
9.2.3 从系统工程到硬件和软件工程 …… 279
9.2.4 关键抽象和机制 …… 281
9.3 构造 …… 282
9.3.1 消息传送 …… 283
9.3.2 列车日程计划 …… 285
9.3.3 显示信息 …… 288
9.3.4 传感器数据采集 …… 289
9.3.5 发布版本管理 …… 289
9.3.6 系统架构 …… 290
9.3.7 子系统规格 …… 291
9.4 后移交 …… 293
第 10 章 人工智能——密码分析 …… 294
10.1 先启 …… 295
10.1.1 密码分析需求 …… 295
10.1.2 定义问题的边界 …… 296
10.1.3 黑板框架的架构 …… 297
10.1.4 知识源的分析 …… 299
10.2 精化 …… 299
10.2.1 黑板对象 …… 300

10.2.2 依赖和认定 …… 301
10.3 构造 …… 303
10.3.1 设计黑板对象 …… 304
10.3.2 设计知识源 …… 308
10.3.3 设计控制器 …… 313
10.3.4 集成黑板框架 …… 314
10.3.5 添加新的知识源 …… 317
10.4 后移交 …… 318
10.4.1 系统增强 …… 318
10.4.2 改变需求 …… 320
第 11 章 数据采集——气象监测站 …… 321
11.1 先启 …… 321
11.1.1 气象监测站需求 …… 321
11.1.2 定义问题的边界 …… 322
11.1.3 场景 …… 330
11.2 精化 …… 331
11.2.1 气象监测系统用例 …… 331
11.2.2 架构框架 …… 338
11.3 构造 …… 340
11.3.1 帧机制 …… 340
11.3.2 发布计划 …… 343
11.3.3 传感器机制 …… 344
11.3.4 显示机制 …… 346
11.3.5 用户界面机制 …… 346
11.4 后移交 …… 349
第 12 章 Web 应用——休假跟踪系统 …… 350
12.1 先启 …… 350
12.1.1 需求 …… 350
12.1.2 用例模型 …… 352
12.2 精化 …… 353
12.2.1 部署视图 …… 354
12.2.2 逻辑视图 …… 355
12.2.3 进程视图 …… 357
12.2.4 实现视图 …… 359
12.2.5 用例视图 …… 359
12.3 构造 …… 362

12.3.1 用户体验模型 ····· 362
12.3.2 分析和设计模型 ····· 364
12.3.3 实体 ····· 372
12.3.4 控制器 ····· 379
12.3.5 Web 页面和用户界面 ····· 380
12.4 移交和后移交 ····· 384
附录 A 面向对象编程语言 ····· 386
A.1 语言进化 ····· 386
A.2 Smalltalk ····· 389
A.2.1 概述 ····· 390
A.2.2 例子 ····· 390
A.2.3 参考文献 ····· 393
A.3 C++ ····· 393
A.3.1 概述 ····· 394
A.3.2 例子 ····· 394
A.3.3 参考文献 ····· 397
A.4 Java ····· 397
A.4.1 概述 ····· 398
A.4.2 例子 ····· 399
A.4.3 参考文献 ····· 402
附录 B 进一步阅读 ····· 403
注解 ····· 409
术语表 ····· 430
分类书目 ····· 438

概念

Isaac Newton 爵士（艾萨克•牛顿）私下向一些朋友承认：他知道重力的表现，但不知道重力的原理！

——Lily Tomlin
The Search for Signs of Intelligent Life in the Universe

在面向对象（OO）技术发展的早期，许多人最初是通过程序设计语言来接触“OO”的。他们发现了这些新的语言可以做的事情，并尝试应用这些语言来解决实际的问题。随着时间的推移，语言得到了改进，开发技术不断演进，出现了一些最佳实践，面向对象方法学正式诞生了。

今天，面向对象开发是一个丰富而强大的开发模型。本部分将回顾支撑这一切的底层理论，并深入讨论面向对象开发模型中各项工作的原理。

第1章

复杂性

一名医生、一名土木工程师和一名计算机科学家在一起争论什么是这个世界上最古老的职业。医生说："在圣经中，上帝用亚当的一根肋骨创建了夏娃。这显然需要外科手术，所以我当然可以宣称我的职业是世界上最古老的职业。"土木工程师打断道："但是在'创世纪'中更早的部分，描述了上帝从混沌中创造了天堂和人间的秩序。这是首次应用土木工程，也是土木工程最伟大的应用。因此，亲爱的医生，您错了，我的职业才是世界上最古老的职业。"计算机科学家斜靠在她的椅子上，微笑着，然后充满自信地说："啊哈，但是，你们觉得是谁创造了那片混沌？"

"系统越复杂，就越容易全面崩溃"[5]。建筑师一般不会想要为一幢 100 层的大楼添加一个新的地下室，因为这样做成本会很高，无疑将失败。但让人吃惊的是，软件系统的用户在提出类似的改动时，都不会多想一下。相反，他们会说，这只是一个简单的编程问题。

由于我们不能控制软件的复杂性，所以导致了项目延迟、超出预算以及在声称的需求中存在缺陷。我们常常把这种情况称为软件危机，但老实说，持续了这么长时间的问题必须称之为正常情况。不幸的是，这种危机导致了人力资源（最宝贵的商品）的浪费以及许多机会的丧失。没有足够好的开发者来创建用户需要的所有新软件，而且在任一组织机构中，相当一部分开发者必须经常维护或看护老的软件。考虑到软件对大多数产业化国家经济基础的间接和直接贡献，同时考虑到软件可以极大增强个人的能力，我们不能让这种情况继续下去。

1.1 复杂系统的结构

怎样才能改变这种可怕的现状？由于根本问题来自于软件固有的复杂性，所以我们建议先研究其他学科中复杂系统是如何组织的。实际上，如果看看周围的世界，就会发现许多成功的、相当复杂的系统。其中一些系统是人类的作品，诸如航天飞机、英法海底隧道和大型商业组织等。自然界中有许多更为复杂的系统，如人类的循环系统和古巴辣椒的结构。

1.1.1 个人计算机的结构

个人计算机是一个具有相当复杂度的设备。大多数个人计算机由同样的主要部件组成：中央处理器（CPU）、显示器、键盘和某种二级存储设备——通常是 CD 或 DVD 驱动器和硬盘驱动器。我们可以任取其中一个部件并进行进一步分解。例如，CPU 通常包括主存储器、算术逻辑单元（ALU）以及一条连接外围设备的总线。这些部分又可以被进一步分解，ALU 可以被分解为寄存器和随机控制逻辑，而它们又由更为基础的部件如 NAND 门、反相器等组成。

这里我们看到了复杂系统的层次化特征。个人计算机能正常发挥功能是因为它的每个主要部件之间协同工作。这些分离的部件形成一个逻辑整体。实际上，我们之所以能够理解计算机的工作方式，是因为可以将它分解为能够独立研究的部件。因此，我们可以独立地研究显示器的操作和硬盘驱动器的操作。类似地，我们可以在不考虑主存储器子系统的情况下研究 ALU。

复杂系统不仅仅是层次化的，而且这种层次也代表了不同的抽象级别，一层构建于另一层之上，每一层都可以分开来理解。在每一个抽象层都可以发现有一组设备协作，为更高的抽象层提供服务。可选择某个抽象层来满足特定的需求。例如，如果我们追踪主存储器中的一个时钟问题，可能会查看计算机的逻辑门级架构，但是如果我们要找的是电子表格应用中一个问题的根源，这个抽象层就不合适了。

1.1.2 植物和动物的结构

在植物学中，科学家们试图通过研究植物的形态，即植物的外观和结构，来理解植物之间的相似与不同。植物是多细胞的生物，其通过各种植物器官系统之间的合作实现光合作用和蒸发等复杂的行为。植物包含三种主要的结构（根、茎和叶），每种结构都有特定的不同结构。例如，根包括支根、根毛、根尖和根冠。类似地，叶的横切面表明它有表皮、叶肉和维管束。这些结构又由一些细胞构成，在每个细胞内部还可以发现另一层的复杂度，包括叶绿体、细胞核等。像计算机的结构一样，植物的各部分组成了一种层次结构，层次结构中的每一层都有着各自的复杂性。

同一层抽象中的所有部分之间以某种定义良好的方式进行交互。例如，在最高的抽象层，根负责从泥土中吸收水分和矿物质；根与茎交互，茎将这些原材料送到叶子；叶子利用茎输送的水分和矿物质，通过光合作用制造养料。

在给定层的内外之间总有清晰的边界。例如，可以说叶子的各个部分协同工作，作为一个整体提供叶子的功能，其与根的各组成部分之间很少有或没有直接的交互。简而言之，在不同抽象层的不同部分之间，存在着清晰的分离关注。

在计算机中，我们会在 CPU 和硬盘驱动器的设计中找到 NAND 门。类似地，在植物结构层次的各个部分之中，也有大量相同的特点。这是造物主实现简洁表示的方式。例如，细胞是植物中所有结构的基本单元，植物的根、茎、叶最终都是由细胞构成的。但是，尽管这

些基本要素都是细胞，而这些细胞却有许多不同的类型。例如，有的细胞有叶绿体而另一些细胞没有叶绿体，有的细胞有不透水的细胞壁而另一些细胞的细胞壁是半透膜，甚至还有活细胞和死细胞的差别。

在研究植物的形态时，我们找不到某一个独立的部分负责一个大的过程的一小步，如光合作用。实际上，没有集中的部分直接协调较低层各部分的活动。相反，我们看到的是一些独立的部分，它们各自为政，每一部分都展示出相当复杂的行为，每一部分都对许多较高层的功能做出贡献。只有通过这些部分之间的共同协作，我们才能看到植物较高层的功能。复杂性科学把这称之为“突现行为”，即整体的行为大于部分行为之和[6]。

简单地看一下动物学，会发现多细胞动物具有和植物相类似的层次结构：一些细胞构成了组织，几种组织协作构成了器官，一组器官构成了系统（如消化系统），等等。我们再次注意到了造物主的简洁表达：所有动物的基本构成单元都是细胞，就如同细胞是所有植物的基本构成单元一样。当然，植物和动物是有区别的。例如，植物细胞由刚性的细胞壁包围，但动物细胞不是这样。尽管存在着这些差异，但毫无疑问，这些结构都是细胞。这是跨领域的共性的一个例子。

在细胞水平之上的一些机制也是植物和动物所共有的。例如，动植物都利用某种脉管系统在器官内传输养料，同一物种的不同个体之间都表现出性的差异。

1.1.3 物质的结构

从解剖学到核物理，各个领域的研究都为我们提供了许多非常复杂的系统的例子。除了这两条原则，我们还发现了另一种结构上的层次结构。宇航员们研究由星团组成的银河系，恒星、行星和碎块构成了银河系。类似地，核物理学家也在关注一种结构上的层次结构，不过这种结构在尺度上完全不同。原子由电子、质子和中子组成，电子似乎是一种基本粒子，而质子、中子和其他粒子则由更基本的粒子（被称为夸克）组成。

我们又发现了在这样大的层次结构中的一种普适机制，一种共性。具体来说，宇宙中似乎只存在4种相互作用力：重力、电磁力、强相互作用和弱相互作用。许多涉及这些基本相互作用的物理定律，如能量守恒定律和动量守恒定律，既适用于银河系也适用于夸克。

1.1.4 社会机构的结构

作为复杂系统的最后一个例子，让我们来看看社会机构的结构。一群人聚在一起，完成一些个人无法完成的任务。有些机构是临时的，有些机构存在的时间超出了人的寿命许多倍。随着组织机构变得越来越大，我们看到了一种特有的层次结构。跨国公司包含许多子公司，子公司又包含许多部门，部门又包含分支机构，分支机构又包含各地的办公室，等等。如果组织机构能长期存在下去，这些部分之间的边界可能会改变，随着时间的推移，会出现新的、更稳定的层次结构。

这种大型组织机构中各个部分之间的关系就像计算机、植物甚至银河系各部分之间的关系一样。确切地说，一个办公室内的雇员之间的交互程度要强于不同办公室的雇员之间的交

互程度。负责处理邮件的员工一般不会与首席执行官打交道，而是会经常与收发室的其他员工打交道。这里同样存在着适用于不同层次的共同机制。员工和首席执行官都由同一个财务机构发薪水，都使用一些公共设施，如公司的电话系统，来完成他们的任务。

1.2 软件固有的复杂性

一颗垂死的恒星正处在塌缩的边缘，一名儿童在学习如何阅读，白细胞向病毒发起进攻——这是真实事件的几个例子，它们包含着真正可怕的复杂性。软件也可能包含巨大复杂性的元素，但是这里的复杂性基本上是另一种类型。Brooks 曾指出："爱因斯坦认为自然界必定存在着简单的解释，因为上帝不是反复无常或随心所欲的。软件工程师没有这样的信仰来提供安慰。许多必须控制的复杂性是随心所欲的复杂性。"[1]

1.2.1 定义软件复杂性

我们确实注意到，某些软件系统并不复杂。这些大多是可以被遗忘的应用，它们是由一个人提出、构建、维护和使用的，这个人通常是编程新手或独立工作的专业开发人员。但这并不是说所有这些系统都是拙劣和不优雅的，我们也不是想贬低它们的创造者。这些系统的目的通常很有限，生命周期也很短。我们可以扔掉它们，用全新的软件代替它们，不必尝试复用、修复它们或者扩展它们的功能。这样的应用开发起来通常难度不大，但比较乏味，因此，我们对学习设计这样的应用兴趣不大。

与之相反，我们对所谓的"工业级"的开发挑战的兴趣要大得多。我们发现，这些应用表现出非常丰富的行为，它们存在于反馈式系统中，由真实世界的事件驱动或发出驱动事件。对这些应用来说，时间和空间都是稀有资源。这些应用维护着数百万条信息记录的完整性，同时允许并发的更新和查询。这些系统发出命令并控制真实世界的实体，例如，控制空中交通和铁路交通。这样的软件系统通常具有很长的生命周期，随着时间的推移，许多用户渐渐依赖于这些软件系统的正常工作。在工业级软件的世界里，我们也会看到一些框架，它们简化了特定领域的应用的创建，还能看到一些程序模仿了某方面的人类智能。尽管这些应用通常是研发的产品，但它们的复杂性一点也不差，因为它们是增量式开发和探索式开发的方法和基础。

工业级软件的特征是，单个开发者要理解其设计的所有方面非常困难，几乎是不可能的。武断地说，这些系统的复杂性超出了人类智能的范围。不幸的是，我们所说的这种复杂性似乎是所有大型软件系统的基本特征。从根本上来说，我们可以掌握这种复杂性，但不能消除这种复杂性。

1.2.2 为什么软件在本质上是复杂的

正如 Brooks 所指出的："软件的复杂性是一个基本特征，而不是偶然如此"[3]。我们认

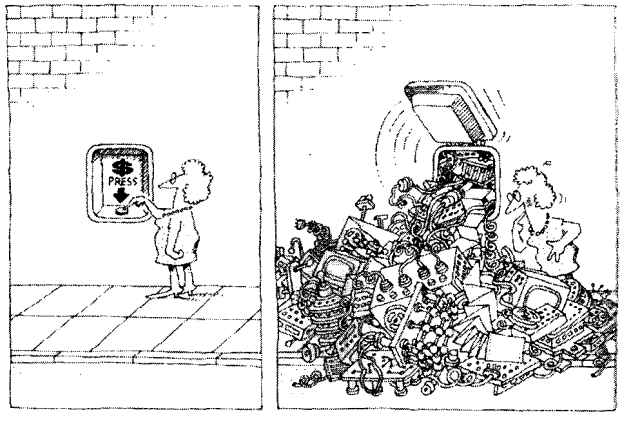
为这种固有的复杂性有4个原因：问题域的复杂性、管理开发过程的困难性、通过软件可能实现的灵活性以及刻画离散系统行为的问题。

1. 问题域的复杂性

我们在软件中试图解决的问题常常涉及不可避免的复杂性，在其中我们可以发现数不清的竞争性需求，甚至是相反的需求。请考虑一下一架多引擎飞机的电子系统的需求、一个蜂窝式移动电话交换系统的需求或一个自动化机器人的需求。这些系统的基本功能已经很难理解了，现在还要加上所有的（常常是隐含的）非功能需求，如可用性、性能、成本、健壮性和可靠性。这种无限制的外部复杂性是导致Brooks所说的任意复杂的原因之一。

这种外部复杂性通常源自于系统用户和系统开发者之间的“沟通困难”：用户常常发现很难用开发者能够理解的形式对他们的需求给出准确的表述。在某些情况下，用户只是对想要的软件系统有一个模糊的想法。这既不是系统用户的错，也不是系统开发者的错，出现这种情况是因为这些人都缺乏另一个领域的经验。用户和开发者对问题的本质有着不同的看法，并根据解决方案的本质做出了不同的假定。实际上，即使用户对他们的需求知道得很清楚，我们目前也没有什么好方法来准确地记录下这些需求。常见的描述需求的方法是用一大段文字，偶尔配有一些插图。这样的文档是难以理解的，可能产生不同的解读，而且经常包含一些设计方案，而不是基本需求。

更麻烦的是，软件系统在开发过程中经常发生需求改变，主要是因为软件开发项目本身改变了问题的规则。看到早期的产品如设计文档和原型之后，在安装好并使用系统之后，在强制使用所有功能之后，用户会对他们的需求有更好的理解和表述。同时，这个过程也帮助开发者了解了问题域，使他们能够问出更好的问题，从而照亮系统期望行为中的黑暗角落。



软件开发团队的任务就是制造简单的假象

因为大型的软件系统是一项投资，所以我们不能够忍受在每次需求发生变化时，就抛弃掉原有的系统。不管是否有计划，系统都会随时间的推移而演化，这种情况常常被错误地称为软件维护。准确地说，在我们修正错误时，这是维护；在我们应对改变的需求时，这是演

化；当我们使用一些极端的手段来保持古老而陈腐的软件继续工作时，这是保护。不幸的是，事实表明相当一部分软件开发资源被用在了软件保护上。

2. 管理开发过程的困难性

软件开发团队的基本任务就是制造简单的假象——让用户与大量的，通常是任意的外部复杂性隔离开来。当然，在软件系统中，规模大并不是太好的事情。我们追求的是通过发明一些聪明的、强大的方法来实现少写代码，从而给我们一种简单的假象，另外也复用已有的设计和代码的框架。然而，就算是系统需求的数量有时候也是无法逃避的，迫使我们要么编写大量的新软件，要么以创新的方式复用已有的软件。就在几十年前，只有几千行的汇编程序就已经是软件工程能力的极限了。但在今天，常常看到交付系统的代码规模达到几十万甚至几百万行（而且这些都是用高级语言写的）。没有哪个人能完全理解这样一个系统。即使以有意义的方式对我们的实现进行分解，也会得到数百个或数千个独立的模块。这样的工作量要求我们启用开发团队，而且理想情况下团队越小越好。但是，不论团队的规模有多大，团队开发总会面临一些重要的挑战。有更多的开发人员就意味着更复杂的沟通，因此更难以协调，特别是当开发团队在地理位置上是分散的时候，而实际情况又常常如此。对于开发团队来说，主要的管理挑战总是维持设计的一致性和完整性。

3. 软件中随处可能出现的灵活性

一家造房屋的公司通常不会自己经营林场，砍伐树木以获取原木。我们也很少看见一家建筑公司建造一个现场的钢铁厂，为新的建筑提供定制的大梁。但在软件行业，这种情况却经常发生。软件提供了非常大的灵活性，所以开发者几乎有可能表达任何形式的抽象。但是，这种灵活性变成了一种难以置信的诱人的属性，因为它也迫使开发者打造几乎所有的初级构建模块，高层的抽象将建立在这些初级构建模块之上。建筑行业对原材料的品质有着统一的编码和标准，但软件行业却很少有这种标准。结果，软件行业还是一种劳动密集型的产业。

4. 描述离散系统行为的问题

如果向空中抛出一个球，我们可以肯定地预测出它的路径，因为我们知道在正常的情况下，某些物理定律会起作用。如果因为我们在抛球时用的力大了一些，结果它就在飞行到一半的时候突然停下来，然后直接往上冲，那么我们将感到非常惊奇[1]。但是在一个调试得不太好的模拟球的运动的软件中，类似这样的行为却很容易发生。

在大型应用中，可能有成百上千个变量以及多个控制线程。系统中的这些变量、它们当前的值、当前的地址和每个过程的调用栈一起构成了应用当前的状态。因为我们是在数字计算机上执行软件，所以我们的系统具有离散的状态。与此形成对比的是，像抛球运动这样的模拟系统是连续的系统。Parnas 指出：“当我们说系统是由连续函数描述的时候，我们是说它

[1] 实际上，由于混沌的存在，即使是简单的连续系统，也可以展示出非常复杂的行为。混沌引入了一种随机性，使我们不能准确预测系统将来的状态。例如，给定两滴水在水流开始处的状态，我们无法准确预测它们在水流结束时相互之间的关系。在各种系统中都发现了混沌，如气象、化学反应、生物系统甚至计算机网络。幸运的是，所有混沌系统中似乎都有更深层次的有序，以一种名为“吸引子”的模式存在。

不会包含任何隐含的惊奇。输入中的小变化总是会导致输出中相应的小变化”[4]。而在另一方面，离散系统从本质上来说具有有限数量的可能状态。在大的系统中，由于组合的缘故，导致可能状态的数目变得非常大。我们试图以分离关注的方式来设计我们的系统，这样，系统某部分的行为对其他部分行为的影响就能降至最低。但是有一个事实仍未改变，即离散系统中的状态转换不能够用连续函数来建模。软件系统之外的每个事件都有可能让系统进入一个新的状态，而且，状态与状态之间的转换关系并非总是确定的。在最坏的情况下，外部的事件可能会破坏系统的状态，因为它的设计者没有考虑到事件之间的相互作用。如果一艘船的推进系统由于计算溢出而失效，其原因是某人在维护系统中输入了错误的数据（一次真正的事故），我们就理解了这个问题的严重性。在地铁、汽车、卫星、航空交通控制、仓库等系统中，与软件相关的系统故障大量上升。在连续系统中，这类行为不太可能发生，但在离散系统中，所有外部事件都有可能影响系统内部状态的任何部分。当然，这是对系统进行大量测试的主要原因，但除了那些极其微不足道的系统之外，穷尽可能的测试是无法做到的。既然数学工具和我们的智能都不能够对大型离散系统的完整行为进行建模，关于系统的正确性，我们必须满足于可接受的信心级别。

1.3 复杂系统的5个属性

考虑到这种复杂性的本质，我们认为对于所有复杂系统，存在5个共同的属性。

1.3.1 层次结构

在Simon和Ando工作的基础上，Courtois提出：

“复杂性常常以层次结构的形式存在。复杂的系统由一些相关的子系统组成，这些子系统又有自己的系统，如此下去，直到达到某种最低层次的基本组件。”[7]

Simon指出，“许多复杂系统都有几乎可分解的层次结构，正是这一事实让我们能够理解、描述甚至‘看到’这样的系统和它们的组成部分”[8]。确实，我们似乎只能理解那些有层次结构的系统。

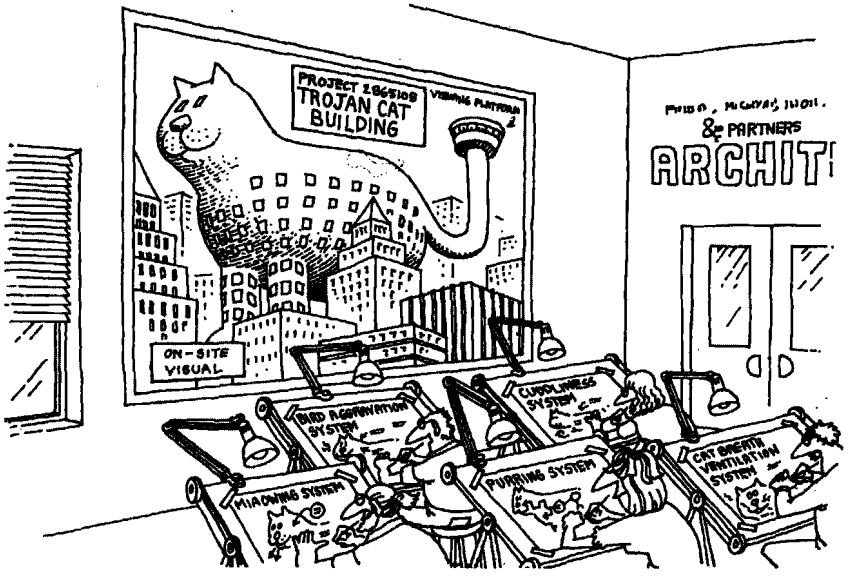
复杂系统的架构是它所有的组件以及这些组件之间的层次结构的函数，认识到这一点很重要。“所有系统都有子系统，所有系统都是更大系统的组成部分……一个系统所提供的价值肯定来自于各个组成部分之间的相互关系，而不是来自于单个的组成部分”[9]。

1.3.2 相对本原

关于复杂系统中基础组件的实质，我们的经验表明：

“选择哪些作为系统的基础组件相对来说比较随意，这在很大程度上取决于系统观察者的判断。”

对于一个观察者来说很基础的东西，对另一个观察者可能具有很高的抽象层次。



复杂系统的架构是它所有的组件以及这些组件之间的层次结构的函数

1.3.3 分离关注

Simon 将层次系统称为“可分解的”，因为它们可以被分成一些可标识的部分，他称它们是“几乎可分解的”，因为这些部分并不是完全独立的。这引出了所有复杂系统的另一个共同属性：

“组件内的联系通常比组件间的联系更强。这一事实实际上将组件中高频率的动作（涉及组件的内部结构）和低频率的动作（涉及组件间的相互作用）分离开来。”[10]

组件内部作用和组件间作用的差异让我们在系统的不同部分之间实现“分离关注”，让我们能够以相对隔离的方式来研究每个部分。

1.3.4 共同模式

前面曾提到，许多复杂系统都是以一种经济的表达方式来实现的。于是 Simon 指出：

“层次结构通常只是由少数不同类型的子系统按照不同的组合和安排方式构成的。”[11]

换言之，复杂系统具有共同的模式。这些模式可能涉及小组件的复用，如细胞，或者大一些的结构如脉管系统，在植物和动物中都存在。

1.3.5 稳定的中间形式

前面曾提到，复杂的系统趋向于随时间而演变。准确地说，“如果存在稳定的中间形式，从简单系统到复杂系统的演变将更快。”[12]用更夸张的词来说：

“复杂系统毫无例外都是从能工作的简单系统演变而来的……从头设计的复杂系统根本不能工作，也不能通过打补丁的方式使其工作。必须从头开始，从能工作的简单系统开始。”[13]

随着系统的演变，曾经被认为是复杂的对象就变成了基础对象，在这些对象的基础上构建更复杂的系统。而且，永远也不能够第一次就正确打造出这些基础对象，必须在上下文环境中使用它们，然后随着时间的推移不断地改进它们，因为我们对系统的真实行为了解得越来越多。

1.4 有组织和无组织的复杂性

发现共同抽象和机制极大地促进了我们对复杂系统的理解。例如，只要通过几分钟的指导，一名有经验的飞行员就能够进入一架以前从未飞过的多引擎飞机并安全地驾驶它。认识到所有飞机的共同属性，如舵、副翼和节流阀的功能之后，飞行员主要需要了解哪些特性是这架飞机所特有的。如果飞行员已经知道如何飞一架飞机，那么学会飞一架类似的飞机就会容易很多。

1.4.1 复杂系统的规范形式

这个例子表明，我们前面对“层次结构”这个术语的使用方式是很宽泛的。大部分有趣的系统不只包含单一的层次结构，相反，我们发现同一个复杂系统中通常表现出许多不同的层次结构。例如，研究飞行器时我们可以将它分解为推进系统、飞行控制系统等。这种分解代表了结构上或“组成部分（part of）”的层次结构。

另外，我们可以按一种完全正交的方式来分解这个系统。例如，涡轮引擎是一种具体的喷气引擎，Pratt and Whitney TF30 又是一种具体的涡轮引擎。换言之，喷气引擎代表了所有类型的喷气引擎的共同特性的抽象，涡轮引擎只是一种特殊类型的喷气引擎，它有一些特有的特征，可以和冲压喷气式引擎区分开来。

第二种层次结构代表了“是一种（is a）”的层次结构。根据我们的经验，从这两种观点来看系统都很重要，应既研究它的“是一种”层次结构也研究“组成部分”层次结构。我们把这些层次结构分别称为“类结构”和“对象结构”，原因将在下一章中解释[1]。

如果读者熟悉对象技术，我们要做一些澄清。在这个例子里，当我们提到类结构和对象结构时，并不是指在编写软件时创建的类和对象。我们指的是更高抽象层的类和对象，它们组成了复杂的系统，如喷气引擎、飞机骨架、不同类型的座位、自动导航子系统等。读者可以回顾前面关于复杂系统属性的讨论，什么是基础组件与观察者有关。

图 1-1 所示为系统的两个正交的层次结构：它的类结构和它的对象结构。每个层次结构都是分层的，在许多基础类和对象之上构建了许多抽象类和对象。选择哪些类或对象作为基础类或对象，这与要解决的问题有关。深入每个层次，又会看到另一层的复杂性。特别是在对象结构的各个部件中，同一抽象层上的对象之间存在着紧密的协作。

---

[1] 复杂软件系统还包含其他类型的层次结构。其中特别重要的是模块结构，它描述了系统物理组件之间的关系，以及处理层次结构，它描述了系统的动态组件之间的关系。

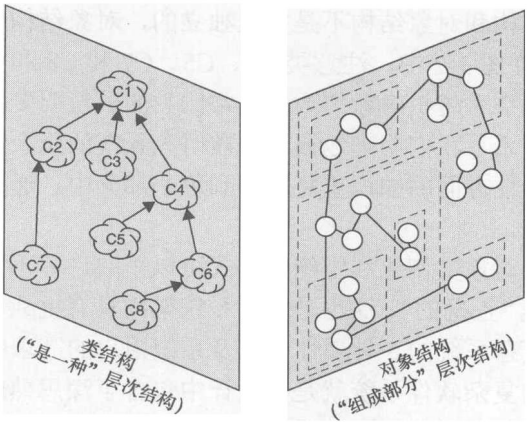


图 1-1 复杂系统的关键层次结构

将类结构和对象结构的概念与复杂系统的 5 种属性（层次结构、相对基础（如多层次的抽象）、分离关注、模式和稳定的中间形式）结合起来，我们发现，基本上所有的复杂系统都具有相同的（规范的）形式，如图 1-2 所示。我们将系统的类结构和对象结构统称为它的“架构”。

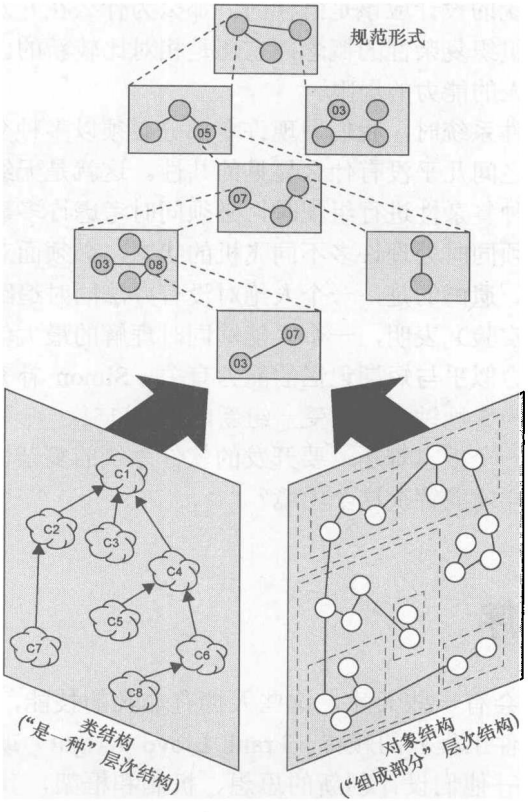


图 1-2 复杂系统的规范形式

还要注意的是，类结构和对象结构不是完全独立的，对象结构中的每个对象都代表了某个类的一个具体实例。（在图 1-2 中，注意类 C3、C5、C7 和 C8 和实例 03、05、07 和 08。）正如在图中所看到的，复杂系统中对象的数量通常比类的数量要多很多。通过展示“组成部分”和“是一种”层次关系，我们明确地揭示了被研究系统的冗余性。如果不揭示系统的类结构，我们就不得不将属性方面的知识重复地放到每个部分中。通过类结构，我们在一个地方记录了这些共同的属性。

在同样的类结构中，对象实例化和组织的方式可以采取许多不同的方式。没有一种特定的架构可以真正被认为是“正确的”。这也是使得系统架构具有挑战性的原因——在可能的许多种系统组件结构中、复杂系统的 5 种属性中以及系统用户的需求中寻找平衡。

经验表明，最成功的复杂软件系统就是在设计中包含了深思熟虑的类结构和对象结构，并具备了前一节所描述的复杂系统的 5 种属性。为了不让这个重要的结论被忽略掉，让我们说得再直接一些：如果不考虑这些因素，我们就不太可能准时地在预算范围之内交付满足需求的软件系统。

1.4.2 处理复杂性时人的能力的局限

既然我们知道复杂系统的设计应该是怎样的，那么为什么在开发它们时还是会遇到严重的问题呢？这涉及软件的组织复杂性的概念，它还是相对比较新的。但是，还存在另一个主宰因素：在处理复杂性时人的能力的局限。

在刚开始分析复杂软件系统时，我们发现许多部分必须以多种交错的方式进行交互，这些部分和它们的交互行为之间几乎没有什么显见的共性。这就是无组织的复杂性的例子。当我们通过设计的过程对这种复杂性进行组织时，必须同时考虑许多事情。例如，在一个航空交通控制系统中，我们必须同时处理许多不同飞机的状态，必须面对相当大的、交错的、有时是非确定性的状态空间。遗憾的是，一个人绝对没有办法同时追踪所有的细节。心理学家的一些实验（如 Miller 的实验）表明，一个人能够同时理解的最大信息数量是 7 个，上下浮动 2 个[14]。这种渠道能力似乎与短期记忆的能力有关。Simon 补充说明，处理速度也是一个限制因素：大脑需要大约 5 秒钟才能接受一组新的信息[15]。

因此，我们面对的是一个根本难题：要开发的软件系统的复杂性在增加，而我们处理复杂性的能力却有局限。怎样才能解决这个困境？

1.5 从混沌到有序

当然，我们中间总是会有一些天才，这些人拥有非凡的技能，一个人就能做一群普通开发者的工作，相当于软件工程界的赖特（Frank Lloyd Wright）或达芬奇。这些人就是我们寻找作为架构师的人——他们设计创新的思想、机制和框架，其他人可以用来作为其他应用或系统的架构基础。但是，“这个世界上天才很少，没有理由相信软件工程领域拥有大

量的天才”[2]。尽管我们中间确实有一些天才，但是在工业级软件开发的领域，我们不能够总是依赖个人的灵感引导我们前进。因此，必须考虑通过一些训练有素的方式来控制复杂性。

1.5.1 分解的作用

“控制复杂性的技巧我们从远古时代就知道了，即分而治之。”[16]在设计一个复杂系统时，重要的是将它分解为一些小而又小的部分，然后可以独立地处理每个部分。通过这种方式，我们适应了人类认知时的渠道能力局限：要理解某一层次的系统，只需要一次理解几个部分（而非所有部分）。实际上，正如 Parnas 所说的，通过分割系统的状态空间，聪明的分解直接针对软件系统内在的复杂性[17]。

1. 算法分解

对于自顶向下的结构化设计，我们中的大多数人都接受过正统的训练，所以我们将分解作为一种简单的算法分解，即系统中的每个模块代表了某个总体过程的一个主要步骤。图 1-3 是一个结构化设计的产品的例子，结构图展示了解决方案的不同功能模块之间的关系。这张结构图展示了一个程序的部分设计，即更新一个主控文件（master file）的内容。它是利用一个专家系统工具从一个数据流图中自动生成的，该工具包含了结构化设计的规则[18]。

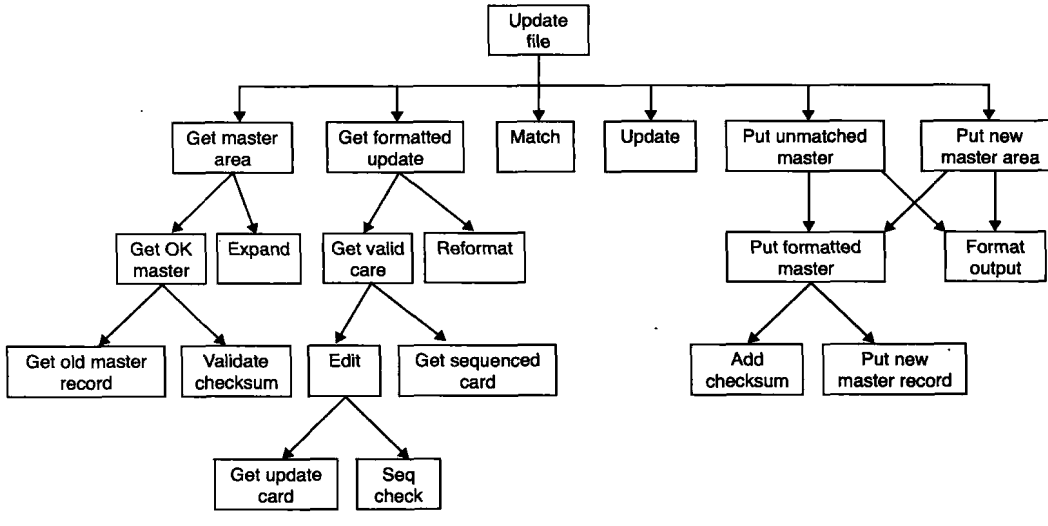


图 1-3 算法分解

2. 面向对象的分解

我们认为，对于同样的问题，还存在另一种可能的分解方式。如图 1-4 所示，我们根据问题领域中的关键抽象概念对系统进行了分解。我们没有将问题分解为“Get formatted Update（取得格式化的更新信息）”和“Add checksum（添加校验和）”这样的步骤，而是确定了“Master File”和“Checksum”这样的对象，这是直接从问题域的词汇表中得到的。

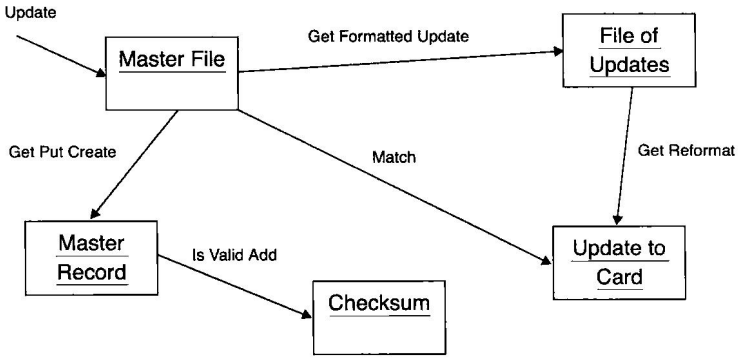


图 1-4 面向对象的分解

虽然两种设计解决的是相同的问题，但它们处理的方式相当不一样。在第二种分解中，我们把世界看成是一组自动化的代理，它们互相协作，执行某种高级的行为。所以“取得格式化的更新信息”不是存在于一个独立的算法中，而是与“File of Update（更新文件）”对象相关联的一个操作。调用这个方法创建了另一个对象，即“Update to Card（对卡的更新）”。按照这种方式，我们的解决方案中的每个对象都有它自己的独特行为，每个对象都是真实世界中的某个对象的模型。从这个角度来看，一个对象就是一个可以触摸的实体，展示了一些定义良好的行为。对象能做一些事情，我们通过发送消息要求它们做它们能做的事情。因为我们的分解基于对象而不是算法，所以称之为“面向对象的分解”。

3. 算法分解与面向对象分解

对复杂系统的分解，哪一种是正确的方法——按算法分解还是按对象分解？实际上，这个问题带有欺骗性，因为正确的答案是两种观点都有其各自的重要性。算法的观点强调了事件的顺序，面向对象的观点强调了一些代理，它们要么发出动作，要么是这些操作执行的对象。

分析和设计方法的分类

我们发现，区分“方法”和“方法学”这两个术语是有意义的。一种方法是一个有规定的过程，目的是生成一组模型，利用某种定义良好的表示法，描述被开发的软件系统的各个方面。另一种方法学是一组方法，适用于软件开发生命周期的各个阶段，它由过程、实践和某种一般的哲学统一起来。方法很重要，这有几个原因。首先，它们在复杂软件系统的开发中注入了纪律。其次，它们定义了一些产品，这些产品成为了开发团队中的成员进行沟通的载体。另外，方法定义了管理层测量进度和管理风险所需的里程碑。

随着软件系统复杂度的增长，方法也在演进。在计算机技术的早期，人们不会去写大型程序，因为那时计算机的能力非常有限。构建系统的主要限制条件是硬件：计算机的内存很小，程序必须与磁鼓这样的二级存储设备的大延迟搏斗，处理器的时钟周期也是以几百毫秒来计算的。在 20 世纪的 60 年代和 70 年代，计算机经济开始发生了极大

的变化，硬件的成本直线下降，同时计算的能力直线上升。因此，人们越来越希望对日益复杂的应用实现自动化，最终在经济上也可行了。作为重要的工具，高级程序设计语言出现了。这些语言改进了单个开发者及开发团队整体的生产效率，具有讽刺意味的是，这又迫使我们创建更为复杂的系统。

许多设计方法是在20世纪的60年代和70年代提出来的，其针对的是不断增加的复杂性。其中最有影响的方法是自顶向下的结构化设计，也被称为“组合设计”。这种方法直接受到传统的高级程序设计语言如FORTRAN和COBOL的影响。在这些语言中，基本的分解单元是子程序，这导致了程序具有树的形态，子程序通过调用其他子程序来完成工作。这就是自顶向下的结构化设计所采取的方法：设计者通过算法分解将大问题分解为较小的步骤。

自从20世纪的60年代和70年代以来，人们制造出了能力更强的计算机。结构化设计的价值没有改变，但正如Stein所说的，“当应用超过10万行代码时，结构化程序似乎就不行了”[19]。人们提出了许多设计方法，其中许多方法就是针对自顶向下的结构化设计的缺点提出来的。在Teledyne Brown Engineering的一次全面调查中，Peters[20]、Yau和Tsai[21]对比较有趣和成功的设计方法进行了分类整理。可能并不奇怪，这些方法中的大多数基本上都是类似主题的一些变奏。实际上，Sommerville指出，绝大多数方法都可以归为以下三类之一[23]：

- 自顶向下的结构化设计；
- 数据驱动设计；
- 面向对象设计。

Yourdon和Constantine[24]、Myers[25]和Page-Jones[26]等人的著作对自顶向下的结构化设计给出了例子和说明。这种方法的基础源自于Wirth [27, 28]及Dahl、Dijkstra、Hoare[29]的工作。Mills、Linger和Hevner [30]提出了结构化设计的一个重要的变化形式。这些不同的方法都采用了算法分解的方式。利用这种方法设计的软件可能比用其他方法设计的软件更多一些。但是，结构化设计不考虑数据抽象和信息隐藏的问题，它也没有提供足够的手段来处理并发。对于特别复杂的系统来说，结构化设计的可伸缩性不太好，而且这种方法与基于对象和面向对象的语言一起使用基本上不合适。

数据驱动设计是Jackson[31, 31]早期工作和Orr方法[33]的最好例证。在这种方法中，系统输入和输出之间的映射关系驱动着软件系统的结构。与结构化设计一起，数据驱动的设计已经成功地被应用于一些复杂的领域，特别是信息管理系统。这些系统涉及系统输入和输出之间的直接关系，但在考虑时间相关的事件方面要求不高。

面向对象分析的底层概念是设计者应该将系统建模为一组协作的对象，将单个对象作为类的实例，而类之间具有层次关系。面向对象的分析和设计直接反映了一些高级程序设计语言的结构，如Smalltalk、Object Pascal、C++、CommonLisp Object System(CLOS)、Ada、Eiffel、Python、Visual C#和Java。

但是，实际情况却是我们无法同时用两种方法来构建复杂系统，因为它们的观点是完全正交的[1]。开始分解系统时，我们必须要么从算法开始，要么从对象开始，然后利用得到的结构作为框架来表达其他的看法。

经验使我们首先应用面向对象的观点，因为这种方法更有助于组织软件系统的内在复杂性。它帮助我们描述复杂系统中有组织的复杂性，这些复杂系统包括计算机、行星、银河和大型社会团体等。第2章中将进一步讨论，与算法分解相比，面向对象分解具有一些非常重要的优点。面向对象分解通过复用共同的机制，得到一些较小的系统，从而提供了重要的表达经济性。面向对象系统在应对变化时也更有弹性，从而更能够随时间演变，因为它们的设计是基于稳定的中间状态的。实际上，面向对象分解极大地降低了构建复杂软件系统的风险，因为它们的思路是从我们有信心的、较小的系统开始增量式地演进。而且，通过帮助我们明智地决定对巨大的状态空间进行分离关注，面向对象的分解直接关注了软件的内在复杂性。

本书第3部分通过一些应用展示了这些好处，这些应用来自于一些不同的问题域。本小节的补充材料“分析和设计方法的分类”，进一步比较了面向对象的观点和较为传统的设计方法。

1.5.2 抽象的作用

我们在前面提到了Miller的实验，他从这些实验中得出结论，一个人同一时刻只能理解大约7个信息，上下浮动2个。这个数字似乎与信息的内容无关。正如Miller自己说的：“绝对判断的范围和短期记忆的范围对我们能够接收、处理和记住的信息量有着很强的限制。通过将输入组织为一些不同的维度，并形成一些片断序列，我们设法打破……这种信息瓶颈”[35]。用现在的术语来说，我们把这个过程叫做“分块”或“抽象”。

正如Wulf所描述的：“我们（人类）已经形成了一种异常强大的技术来对付复杂性。我们对它进行抽象。不能够全面掌握一个复杂的对象，我们就选择忽略它的非本质的细节，转而处理这个对象的一般化的、理想化的模型”[36]。例如，如果研究植物中光合作用的原理，我们可以关注叶子细胞中发生的化学反应，忽略掉其他的部分，如根和茎。我们仍然受到同时可以理解的事物数量的限制，但通过抽象，我们利用了信息的分块和不断增大的语义内容。当我们采用面向对象的观点来看世界时，尤其是这样，因为对象作为真实世界中实体的抽象，代表了特定的一块密集而内聚的信息。第2章将更详细地讨论抽象的意义。

1.5.3 层次结构的作用

另一种增加单块信息的语义内容的方法，是在复杂的软件系统中显式地组织类和对象层次结构。对象结构很重要，因为它展示了不同的对象之间如何通过一些交互模式进行协作，我们把这些交互模式称为“机制”。类结构也同样重要，因为它强调了系统中的公共结构行为。因此，不必去研究某片植物叶子上的每个光合作用细胞，只要研究一个这样的细胞就行了，

---

[1] Langdon指出，这种正交关系在很早的时候就有研究了。他说：“C.H.Waddington注意到双重观点可以追溯到古希腊时代。德谟克利特提出了一种被动的观点，他断言世界是由原子组成的。德谟克利特的观点将事物放在了考虑的中心。另一方面，主动观点的经典代言人是赫拉克利特，他强调处理的概念。”[34]

因为我们预期所有的其他细胞都会表现出相似的行为。虽然我们把一类对象的每个实例作为不同的实体，但是我们可以假定同类对象的所有其他实例都具有同样的行为。通过将对象分成具有相关抽象的组（例如，植物细胞与动物细胞），我们明确地区分了不同对象的共同属性和特有属性，这又进一步帮助我们掌握它们内在的复杂性[37]。

确定复杂软件系统中的层次关系通常并不容易，因为这要求在许多对象之中发现模式，每个对象都可能拥有大量复杂的行为。但当我们整理出这些层次结构之后，复杂系统的结构以及我们对它的理解都得到了极大的简化。第 3 章将详细讲解类和对象层次结构的实质，第 4 章将描述帮助我们确定这些模式的技术。

1.6 复杂系统的设计

每一种工程方法的实践，无论它是土木工程、机械工程、化学工程、电气工程还是软件工程，都涉及科学和艺术两方面的元素。正如 Petroski 雄辩地指出："一个新结构的设计概念既包括想象力的跳跃，也包括经验和知识的融合，就像艺术家在他的画布或纸上创作作品一样。当作为艺术家的工程师想出这个设计之后，必须由作为科学家的工程师采用科学方法进行分析，像其他科学家所使用的方法一样"[38]。类似地，Dijkstra 说："程序设计任务是应用抽象的大规模练习，因此需要正规数学家的能力和有能力的工程师的态度"[39]。

1.6.1 作为科学和艺术的工程

在设计一个全新的系统时，工程师的角色特别有挑战性。特别是在设计反应系统和命令控制系统时，常常需要针对一组完全独特的需求编写软件，运行在专门为这个系统配置的目标处理器上。在另一些情况下，如创建网络、研究人工智能的工具或信息管理系统，可能有定义良好的、稳定的目标环境，但是需求可能在一个或多个方面对软件技术形成压力。例如，可能需要创建一个更快的、容量更大的，或功能改进很快的系统。在所有这些情况下，我们都试图利用已经验证过的抽象和机制（用 Simon 的话说，就是"稳定的中间状态"）作为基础，在此之上构建新的复杂系统。在大量的可复用软件组件库中，软件工程师必须以一种创新的方式来组装这些部分，以满足明确和隐含的需求，就像画家或音乐家必须将他们的介质发挥到极致一样。

1.6.2 设计的含义

在每一种工程实践中，设计都是一种训练有素的方法，我们通过它来创造某个问题的解决方案，从而提供实践需求的途径。在软件工程中，Mostow 指出设计的目的是要构建如下的一个系统：

- 满足给定的（可能是非正式的）功能规格说明；
- 符合目标介质的限制；

- 满足隐含的和明确的性能及资源使用需求；
- 满足隐含的和明确的关于产品形式方面的设计限制条件；
- 满足对设计过程本身的限制条件，如时间、费用或进行设计可用的工具。

Stroustrup 指出，“设计的目的是创建一个干净的、相对简单的内部结构，有时候也被称为架构……一份设计是设计过程的最终产物”[41]。设计包括在一组竞争的需求之间进行平衡。设计的产品是一些模型，让我们能够阐明我们的结构，当需求冲突时进行折中，总之，为实现提供了一份蓝图。

1. 建模的重要性

建模在所有工程实践中都已得到广泛地接受，这主要是因为建模引证了分解、抽象和层次结构的原则[42]。设计中的每个模型都描述了被考虑的系统的某个方面。我们尽可能地在老模型的基础上构建新模型，因为我们对那些老模型已经建立起了信心。模型让我们有机会在受控制的条件下失败。我们在预期的情况和特殊的情况下评估每个模型，当它们没能按照我们的期望工作时，我们就修改它们。

我们发现，为了表达一个复杂系统的所有精妙之处，必须使用多种模型。例如，当设计一台个人计算机时，电子工程师必须考虑系统的组件级视图以及线路板的物理布局。这个组件视图构成了系统设计的逻辑视图，它帮助工程师思考组件间的协作关系。线路板的布局代表了这些组件的物理封装，它受到线路板尺寸、可用电源和组件种类等条件的限制。通过这个视图，工程师可以独立地思考散热和制造等方面的问题。线路板的设计者还必须考虑到在建系统的动态方面和静态方面。因此，电子工程师利用一些图示来展示单个组件之间的静态连接，也用一些时间图来展示这些组件随时间变化的行为。然后，工程师就可以利用示波器和数字分析设备来验证静态模型和动态模型的正确性。

2. 软件设计方法学的要素

显然，不存在魔法，没有“银弹”可以万无一失地让软件工程师从需求得到一个复杂系统的实现。实际上，关于复杂系统的设计，不存在所谓的指南手册。正如前面提到的复杂系统的5个属性，设计这样的系统需要增量和迭代的过程。

但是，好的设计方法仍然为开发过程带来了非常必要的实践方法。软件工程界已经发展出几十种不同的设计方法学，可以将这些方法学大致分成三类（参见补充材料“分析和设计方法的分类”）。除了它们的不同之处，所有这些方法都有一些共同的要素。具体来说，包括下列要素：

- 表示法　表达每个模型的语言；
- 过程　导致有序构建系统模型的过程；
- 工具　消除建模中的枯燥工作并强制实现模型本身的规则的工件，可以揭示错误和不一致性。

一个好的设计方法基于牢固的理论基础，同时又提供艺术创新的自由。

3. 面向对象开发的模型

是否存在“最好的”设计方法？对这个问题没有绝对的答案。它实际上只是前面问题的

另一种提法：什么是分解一个复杂系统的最佳方法？通过反复实践，我们发现，构建关注问题域中的“事物”的模型具有很大的价值，这形成了我们所谓的面向对象分解。

面向对象分析和设计的方法实现了面向对象分解。通过应用面向对象设计，我们创建出能够灵活适应变化的软件，并体现出表达的经济性。通过明智地分离它的状态空间，我们对软件正确性的信心被提升到了一个新高度。最终，降低了开发复杂软件系统所固有的风险。

本章中，我们举例介绍了利用面向对象分析和设计来掌握开发软件系统相关的复杂性。另外，我们还指出了应用这种方法所带来的一些基本好处。但是，在讨论面向对象设计的表示法和过程之前，我们必须研究面向对象开发所基于的原则，也就是抽象、封装、模块化、层次结构、类型、并发和持久。

1.7 小结

- 软件本质上是复杂的，软件系统的复杂性常常超出了人类智能的范围。
- 软件开发团队的任务就是制造出简单的假象。
- 复杂性常常以层次结构的形式表现出来，建立复杂系统的“是一种”和“组成部分”层次结构模型是有意义的。
- 复杂系统通常是从一些稳定的中间状态演进而来的。
- 人类的认识有一些基本的限制因素，我们可以通过分解、抽象和层次结构来克服这些限制。
- 复杂系统可以从事物或处理过程的角度来分析，采用面向对象的分解有一些令人感兴趣的理由。在这种方法中，将世界看作是一组有意义的对象进行协作，实现某种高级的行为。
- 面向对象分析和设计的方法实现了面向对象分解。面向对象的设计采用了一套表示法和过程来构造复杂软件系统，提供了丰富的模型，可以通过这些模型来阐明目标系统的不同方面。

第2章

对象模型

面向对象技术建立在很好的工程基础之上，它的要素统称为“开发对象模型”，或简称为“对象模型”。对象模型包括抽象、封装、模块化、层次结构、类型、并发和持久等原则。就它们本身来说，没有一项原则是新的。但重要的是，对象技术将这些要素以一种相互配合的方式结合起来了。

面向对象分析和设计在本质上与传统的结构化设计方法是不同的：它要求以一种不同的方式来思考分解，它得到的软件架构基本上超出了结构化设计的领域。

2.1 对象模型的演进

面向对象开发不会从无数使用早期技术而失败的软件项目的灰烬中自发地产生。它不是与早期方法断然决裂的。实际上，它是建立在以前技术的最佳思想之上的。本节将讨论我们行业中工具的演进，以帮助理解面向对象的基础和出现。

当回过头去看一看相对简单，但又多姿多彩的软件工程的历史时，肯定会注意到如下两个巨大的趋势：

- 关注点从小规模编程向大规模编程转变；
- 高级程序设计语言的演进。

最新的工业级软件系统与几年前它们的前辈相比，更大也更复杂。这种复杂性的增长推动了大量有用的软件工程应用研究，特别是在分解、抽象和层次结构等方面。开发表达能力更强的程序设计语言为这些进步提供了补充。

2.1.1 程序设计语言的换代

Wegner 根据语言的功能和产生的时间，将一些较为流行的高级语言进行了分类[2]。（这并不是一份所有程序设计语言的清单。）

- 第一代语言（1954～1958）

FORTRAN I	数学表达式
ALGOL 58	数学表达式
Flowmatic	数学表达式
IPL V	数学表达式

- 第二代语言（1959～1961）

FORTRAN II	子程序、单独编译
ALGOL 60	块结构、数据类型
COBOL	数据描述、文件处理
Lisp	列表处理、指针、垃圾收集

- 第三代语言（1962～1970）

PL/1	FORTRAN + ALGOL + COBOL
ALGOL 68	ALGOL 60 的严格继承者
Pascal	ALGOL 60 的简单继承者
Simula	类、数据抽象

- 代沟（1970～1980）

人们发明了许多不同的语言，但很少存活下来。但是，下面的语言值得一提。

C	高效、可执行程序小
FORTRAN 77	ANSI 标准

让我们扩展一下 Wegner 的分类。

- 面向对象兴盛（1980～1990，但幸存下来的语言不多）

Smalltalk 80	纯面向对象语言
C++	从 C 和 Simula 发展而来
Ada83	强类型，受到 Pascal 的很大影响
Eiffel	从 Ada 和 Simula 发展而来

- 框架的出现（1990～现在）

出现了许多语言活动、新版本和标准化工作，导致了程序设计框架。

Visual Basic	简化了 Windows 应用的图形用户界面（GUI）开发
Java	Oak 的后续版本，其设计意图是实现可移植
Python	面向对象的脚本语言
J2EE	基于 Java 的企业级计算框架
.NET	微软公司的面向对象框架
Visual C#	.NET 框架下 Java 的竞争者
Visual Basic .NET	针对微软.NET 框架的 Visual Basic

在这一系列的语言换代之中，每种语言支持的抽象机制发生了变化。第一代语言主要用于科学和工程应用，这个问题领域的词汇几乎全是数学。因此，开发了出像 FORTRAN I

这样的语言，让程序员能写出数学公式，从而不必面对汇编语言或机器语言中的一些复杂问题。所以，第一代高级程序设计语言标志着向问题空间靠近了一步，向底层计算机远离了一步。

在第二代语言中，重点是算法抽象。那时候，计算机变得越来越强大，计算机产业经济意味着更多的问题可以自动化，特别是在商业应用中。这时，关注的焦点主要在告诉计算机做什么：先读入这些个人记录，接下来进行排序，然后打印这份报告。同样，这一代的程序设计语言让我们向问题空间又靠近了一步，向底层计算机又远离了一步。

在20世纪60年代后期，特别是半导体和集成电路技术发明之后，计算机硬件的成本迅速下降，而处理能力几乎呈指数上升。这时可以解决更大的问题，但需要操作更多类型的数据。因此，像ALGOL 60和稍后的Pascal这样的第三代语言演进到支持数据抽象。这时，程序员可以描述相关数据的意义（它们的类型），并让程序设计语言强制确保这些设计决策。这一代高级程序设计语言再一次让我们向问题空间靠近了一步，向底层计算机远离了一步。

20世纪70年代开展了大量的程序设计语言研究活动，结果导致产生了数千种不同的程序设计语言和方言。在很大程度上，编写越来越大的程序的愿望凸显了早期语言的不足之处。因此，人们发展了许多新的语言机制来解决这些局限。极少语言幸存下来（最近的教科书中还提到Fred、Chaos或Tranquil等语言吗？），但是它们提出的许多要领被早期语言的继承者们吸收了。

我们最感兴趣的是所谓的“基于对象”和“面向对象”的语言。基于对象和面向对象的程序设计语言为软件的面向对象分解提供了最好的支持。这些语言的数量（以及原有语言的“对象化”变种的数量）在20世纪80年代和90年代早期大量增加。自1990年以来，在一些商业程序设计工具提供商的支持下，一些语言成为了主流OO语言（如Java和C++）。程序设计框架（如J2EE、.NET）通过提供组件和服务，简化了常见的、琐碎的编程任务，为程序员提供了很大的支持。它们的出现极大地提高了生产效率，展示了组件复用难以捉摸的未来。

2.1.2 第一代和第二代早期程序设计语言的拓扑结构

先来考虑一下每一代程序设计语言的结构。图2-1演示了第一代和第二代早期程序设计语言的结构。所谓“拓扑结构”（topology），指的是这种语言的基本物理构成单元，以及这些部分是如何连接的。从图中可以看到，像FORTRAN和COBOL这样的语言，所有应用的基本物理构成单元是子程序（对于使用COBOL的人来说，称之为段落）。

用这些语言编写的应用展现出相对较平的物理结构，只包含全局数据和子程序。图2-1中的箭头表明了子程序对不同数据的信赖关系。在设计时，设计者可以在逻辑上将不同类型的数据分开，但是在这些语言中没有任何机制来强制确保这些设计决策。程序中某个部分的错误可能给系统的其他部分带来毁灭性的连带影响，因为全局数据结构对于所有子程序都是可见的。

在对大型系统进行修改时，很难维持原有设计的完整性。而且，常常会引入混乱：即使在一段短时间的维护之后，这些语言编写的程序常常会包含子程序间的大量交叉耦合、对数据含义的假定及复杂的控制流，从而对整个系统的可靠性造成威胁，降低解决方案的整体清晰性。

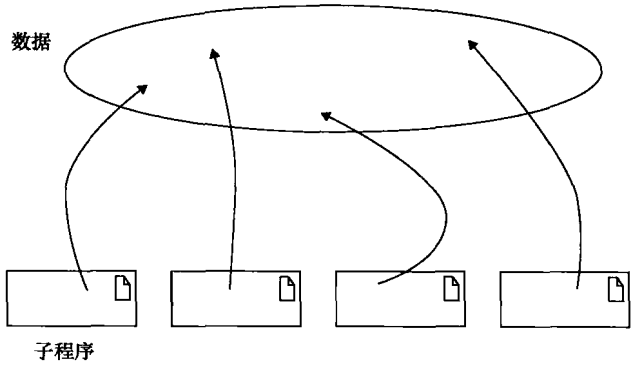


图 2-1 第一代和第二代早期程序设计语言的结构

2.1.3 第二代后期和第三代早期程序设计语言的结构

在 20 世纪 60 年代中期，程序被认为是问题和计算机之间重要的中间点[3]。“首次软件抽象，现在所谓的‘过程化’抽象，直接来自于这种实际的软件观点……子程序在 1950 年之前就发明了，但是作为一种抽象，那时候并没有被完全接受……相反，最初它们被看作是一种节省劳力的机制……但是很快，子程序就被认为是抽象程序功能的一种方式。”[4]

意识到子程序可以作为一种抽象机制，这产生了三个重要结果。首先，人们开始发明一些语言，支持各种参数传递机制。其次，奠定了结构化程序设计的基础，表明在语言上支持嵌套的子程序，并在控制结构和声明的可见性范围方面发展了一些理论。最后，出现了结构化设计方法，为试图构建大型系统的设计提供了指导，利用子程序作为基本构建块。因此，毫无悬念，如图 2-2 所示，第二代后期和第三代早期语言的结构基本上是前一代语言的主题变奏。这种结构关注早期语言的一些不足之处，具体来说就是需要对算法抽象有更强的控制。但是，它仍然未能解决大规模程序设计和数据设计的问题。

2.1.4 第三代后期程序设计语言的结构

从 FORTRAN II 开始，在大部分第三代后期程序设计语言中，出现了另一种重要的结构机制，并发展为对不断增长的大规模编程问题的关注。大规模编程项目意味着大型的开发团队，因此需要独立地开发同一个程序的不同部分。这种需求的解决方案是能够独立编译的模块，这在早期的概念中只是一种随意的数据和子程序的容器，如图 2-3 所示。模块很

少被看作是一种重要的抽象机制，在实践中，它们只是用于对最有可能同时改变的子程序分组。

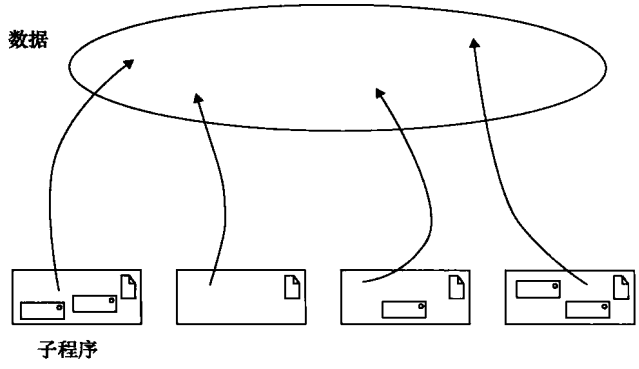


图 2-2 第二代后期和第三代早期程序设计语言的结构

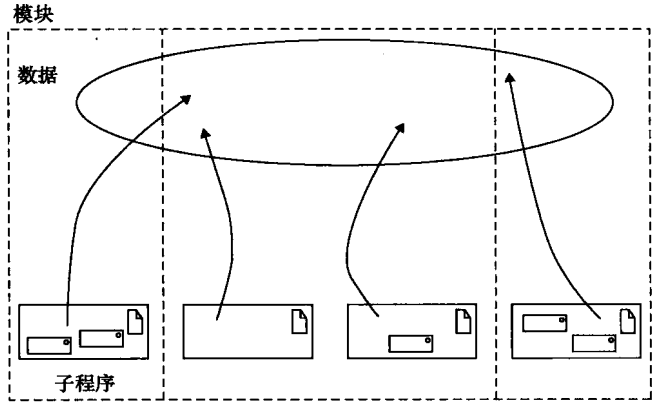


图 2-3 第三代后期程序设计语言的结构

这一代语言中的大多数虽然支持某些模块化结构，但是很少有规则要求模块间接口的语义一致性。为一个模块编写子程序的开发者可能假定它会通过三个不同的参数调用：一个浮点数、一个包含 10 个元素的数组和一个代表布尔标记的整型。在另一个模块中，对这个子程序的调用可能使用了不正确的参数，违反了这一假定：一个整数、一个包含 5 个元素的数组和一个负数。类似地，一个可能使用一块公共数据，把它当作自己的一样，而另一个模块可能违反这些假定，直接操作这块数据。不幸的是，因为这些语言中的大部分对数据抽象和强类型支持得都不太好，这样的错误只有在执行程序时才能被检测出来。

2.1.5 基于对象和面向对象的程序设计语言的结构

数据抽象对于掌握复杂性是很重要的。“通过过程可以实现的抽象在本质上很适合描

述抽象操作，但并不太适合描述抽象的对象。这是一个严重的缺陷，在许多应用中，要操作的数据对象的复杂性在很大程度上决定了问题的复杂性。”[5]这种认识有两个重要结果。首先，出现了数据驱动的方法，它为面向对象的语言提供了一种解决数据抽象问题的方法。其次，出现了关于类型概念的理论，这种理论最终在像 Pascal 这样的语言中得到了实现。

这些思想的自然成果首先出现在 Simula 语言中，经过改进，也导致了一些语言的出现，如 Smalltalk、Object Pascal、C++、Ada、Eiffel 和 Java。这些语言被称为基于对象的语言或面向对象的语言，原因稍后解释。图 2-4 展示了小型和中型应用中这种语言的结构。

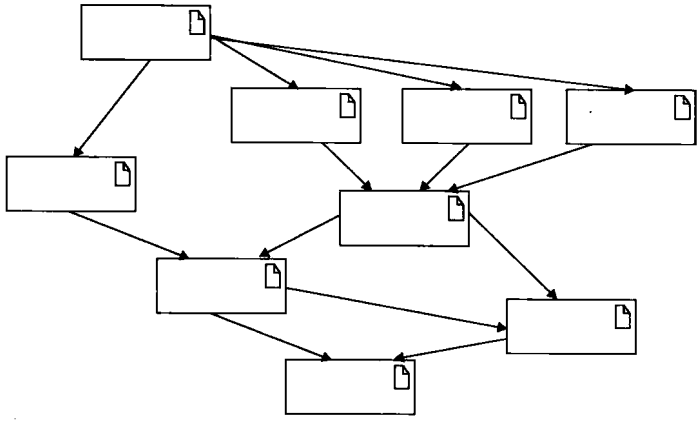


图 2-4 使用基于对象或面向对象编程语言的小型或中型应用的结构

这类语言的构建块是模块，它表现为逻辑上的一组类或对象，而不像早期语言那样是子程序。换言之，“如果过程和函数是动词，数据是名词，那么面向过程语言的程序就是围绕动词组织的，面向对象的程序就是围绕名词组织的”[6]。出于这个原因，小型或中型面向对象应用的物理结构表现为一个图，而不像面向算法的语言那样通常是一棵树。另外，基本上很少或没有全局数据。数据和操作被放在一个单元中，系统的基本逻辑构建块不再是算法，而是类或对象。

到目前为止，我们已经超越了“大型编程”（programming-in-the-large），必须面对“巨型编程”（programming-in-the-colossal）。对于非常复杂的系统，我们发现类、对象和模块提供了基本的抽象手段，但这还不够。幸运的是，对象模型在规模上可以扩展。在大型系统中，一组抽象可以构建在另一组抽象层之上。在任何一个抽象层上，都可以找到某些有意义的对象，它们可以协作实现更高层的行为。如果我们仔细看一组对象的实现，会看到另一组协作的抽象。这正是第 1 章中描述的复杂性的组成方式，图 2-5 展示了这种结构。

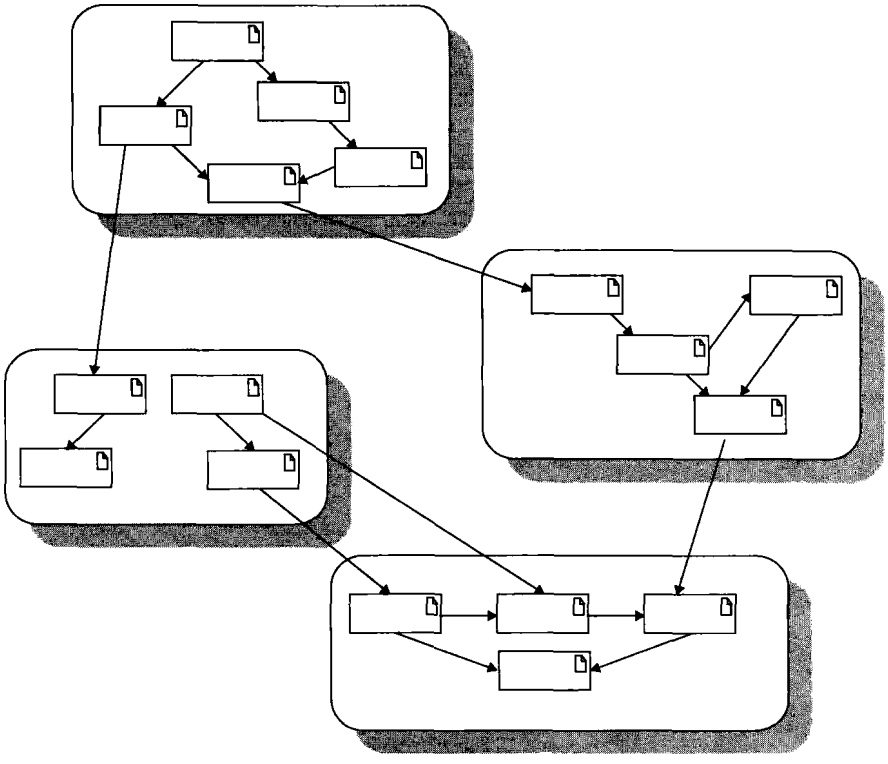


图 2-5 使用基于对象或面向对象编程语言的大型应用的结构

2.2 对象模型基础

结构化的设计方法指导开发者利用算法作为基本构建块来构建复杂系统。类似地，面向对象设计方法利用类和对象作为基本构建块，指导开发者探索基于对象和面向对象编程语言的表现力。

实际上，对象模型受到了一些因素的影响，而不只是面向对象程序设计。实际上，正如补充材料“基础—对象模型”中进一步讨论的，对象模型已被证实是计算机科学中的一个统一的概念，它不仅仅适用于程序设计语言，也适用于用户界面、数据库甚至计算机架构的设计。这种广泛适用的原因就是，面向对象帮助我们处理许多种不同系统中固有的复杂性。

因此，面向对象分析和设计代表了一种演进式的开发，而不是一种革命性的开发。它没有抛弃过去的优点，而是建立在已经证明的好方法的基础之上。遗憾的是，大多数程序员没有在 OOAD 上接受严格的培训。确实，许多优秀的工程师利用结构化设计技术开发并部署了无数有用的软件系统。然而，只利用算法分解在处理大量复杂性的情况时是有局限的，所以我们必须转向面向对象分解。而且，如果试图将 C++和 Java 这样的语言当作传统的面向算法

语言来使用，那么我们不仅会丧失语言提供的威力，而且结果还常常不如使用 C 和 Pascal 这样的老式语言。如果一个木匠不懂电，给他一个电钻，他也只会把它当作锤子来用。他最后可能会弄弯一些钉子，弄伤几根手指，因为电钻不是一把好锤子。

由于面向对象模型有许多不同的来源，所以它不幸地具有一些让人糊涂的技术。Smalltalk 程序员使用“方法”，C++程序员使用“虚成员函数”，CLOS 程序员使用“通用函数”。Object Pascal 程序员说“类型强制”，Ada 程序员则称之为“类型转换”，C#或 Java 程序员会使用“cast”。为了减少混淆，让我们来定义什么是面向对象，什么不是面向对象。

“面向对象”这个短语“已经被毫无顾虑地滥用了，就像‘故乡’、‘苹果派’、‘结构化程序设计’一样。”[17]我们都同意，对象是面向对象的核心。在前面一章中，我们将对象非正式地定义为一个可触摸的实体，它展示了某些定义良好的行为。Stefik 和 Bobrow 将对象定义为“包含了属性、过程和数据的实体，它们执行计算并保存局部的状态”[8]。将对象定义为实体在某种程度上有点问题，但这里的基本概念是对象统一了算法抽象和数据抽象的思想。Jones 进一步澄清了这个术语，指出“在对象模型中，重点在于灵活地刻画物理系统或抽象系统的组件，用一个程序系统来建模……对象具有某种‘完整性’，这种完整性不应违反，实际上也不能违反。对象只能够按照适合它的方式来改变状态、改变行为、实现操作或与其他对象发生联系。换言之，存在一些不变的特征，这些特征刻画了一个对象和它的行为。以一个电梯为例，刻画它的不变特征包括它只能在其竖井中上下运动……所有对电梯的模拟都必须包含这些不变特性，因为它们与电梯的概念是不可分割的整体”[32]。

基础—对象模型

Yonezawa 和 Tokoro 指出，“术语‘对象’在计算机科学的不同领域中独立地出现，几乎都是在 20 世纪 70 年代初期，它们出现时指的是不同的概念，但互相有关。所有这些概念的发明都是为了管理软件系统的复杂性，即对象代表了按模块分解的系统的组件，或者是知识表达的模块化单元”[9]。Levy 补充指出，下列事件促使了面向对象概念的演进：

- 计算机架构的进步，包括基于功能的系统以及对操作系统概念的硬件支持；
- 程序设计语言的进步，如 Simula、Smalltalk、CLU 和 Ada；
- 编程方法学的进步，包括模块化和信息隐藏[10]。

我们可以加上如下 3 点，它们也为对象模型的基础做出了贡献：

- 数据库模型的进步；
- 对人工智能的研究；
- 哲学和认知科学领域的进步。

对象的概念最先出现在 20 多年前的硬件领域中，始于基于描述符的架构（descriptor-based architecture）的发明，以及稍后的基于功能的架构（capability-based architecture）[11]。这些架构代表了与经典的冯·诺依曼架构的决裂，尝试拉近编程语言

的高级抽象和机器本身的低级抽象之间的距离[12]。据它的支持者说，这种架构的好处很多：更好的错误检测、改进的执行效率、更少的指令类型、更简单的编译，以及减少的存储需求。计算机也可以拥有面向对象的架构。

与面向对象架构密切相关的是面向对象操作系统。Dijkstra 在 THE 多程序系统（THE multiprogramming system）中的工作首次引入了将系统构建成分层状态机的概念[18]。其他的先锋面向对象操作系统包括 Plessey/System 250（为 Plessey 250 多处理器设计）、Hydra（为 CMU 的 C.mmp 设计）、CALTSS（为 CDC 6400 设计）、CAP（为 Cambridge CAP 计算机设计），UCLA Secure UNIX（为 PDP 11/45 和 11/70 设计）、StarOS（为 CMU 的 Cm*设计）、Medusa（同样为 CMU 的 Cm*设计）和 iMAX（为 Intel 432 设计）[19]。

对面向对象模型最重要的贡献也许来自于一些程序设计语言，我们称之为基于对象或面向对象的程序设计语言。类和对象的基本思想首次出现在程序设计语言 Simula 67 中。Flex 系统及后续的各种 Smalltalk 方言，如 Smalltalk-72、-74、-76 和最终现在使用的版本 Smalltalk-80，均采用了 Simula 的面向对象设计方式，很自然地将语言中的所有东西都变成了对象的实例。在 20 世纪 70 年代，像 Alphard、CLU、Euclid、Gypsy、Mesa 和 Modula 这样的语言被设计出来，它们支持后来兴起的数据抽象的要领。语言研究导致了将 Simula 和 Smalltalk 的概念嫁接到传统的高级程序设计语言上。面向对象概念和 C 的结合产生了 C++和 Objective C。然后出现了 Java，帮助程序员避免使用 C++中经常发生的编程错误。在 Pascal 中加入面向对象机制导致产生了 Object Pascal 语言、Eiffel 和 Ada。另外，许多 Lisp 的方言也包含了 Simula 和 Smalltalk 的面向对象特征。附录 A 将更详细地讨论一些这样的语言和其他程序设计语言的开发。

第一个正式指出按照层抽象来分解系统的重要性的人是 Dijkstra。Panas 后来引入了信息隐藏的思想[20]，在 20 世纪 70 年代，一些研究者在抽象数据类型机制的研究上走在了前面，最著名的有 Liskov and Zilles [21]、Guttag [22]和 Shaw [23]。Hoare 提出了关于类型和子类的理论。

虽然数据库技术的发展在某种程度上独立于软件工程，但它也为对象模型做出了贡献[25]，主要是通过实体—关系（ER）的方式进行数据建模[26]。ER 模型由 Chen[27]首先提出，在 ER 模型中，世界的模型由实体、实体的属性以及实体之间的关系构成。

在人工智能领域，知识表示方面的进步对理解面向对象的抽象也做出了贡献。在 1975 年，Minsky 首先提出了一个理论框架，将真实世界对象表示为观察到的图像以及自然语言认知系统[28]。从那时起，框架就在不同的人工智能系统中被用来作为架构的基础。

最后，哲学和认知科学对面向对象的发展也做出了贡献。世界可以被看作对象或过程的观点是希腊人的发明，在 17 世纪，笛卡尔说人们很自然地用面向对象的观点来看世界[29]。在 20 世纪，Rand 在她的客观主义认知论哲学中对这些主题进行了扩展。最近，Minsky 提出了一个人类智能的模型。在这个模型中，他认为意识是由一些无意识的代理构成的群体[31]。Minsky 指出，只有通过这些代理的协作，我们才能发现所谓的“智能”。

2.2.1 面向对象编程

那么，究竟什么是面向对象编程（OOP）？我们这样定义：

“面向对象编程是一种实现的方法，在这种方法中，程序被组织成许多组相互协作的对象，每个对象代表某个类的一个实例，而类则属于一个通过继承关系形成的层次结构。”

这个定义有三个要点：（1）利用对象作为面向对象编程的基本逻辑构建块（第 1 章中介绍的“组成部分”层次结构），而不是利用算法；（2）每个对象都是某个类的一个实例；（3）类与类之间可以通过继承关系联系在一起（第 1 章中介绍的“是一个”层次结构）。一个程序可能看起来像是面向对象的，但是如果不满足这三点之一，它就不是一个面向对象的程序。具体来说，没有继承的编程显然不是面向对象的，那只是利用抽象数据类型在编程。

根据这个定义，某些语言是面向对象的，而另一些语言就不是。Stroustrup 建议，“如果‘面向对象语言’这个术语有意义的话，它一定是意味着一种语言具有一些机制，能很好地支持面向对象风格的编程……如果语言提供的机制能够使得利用一种风格编程很方便，那么这种语言就很好地支持了这种风格。如果使用一种语言需要额外的努力或技能才能编写这样的程序，那它就不支持这种技术。在这种情况下，这种语言只能让开发者能用这些技术”[33]。从理论的角度来说，人们可以在 Pascal 甚至 COBOL 或汇编这样的非面向对象语言中模拟面向对象编程，但这样做非常得不偿失。所以 Cardelli 和 Wegner 说：

“当且仅当一种语言满足下列需求时，它才是面向对象的。

- 它支持对象，这些对象是具有命名的操作接口和隐藏的内部状态的数据抽象；
- 对象有相关的类型[类]；
- 类型[类]可以从超类型[超类]中继承属性。”[34]

一种语言支持继承，就意味着它可以表达类型之间的“是一种”关系。例如，红玫瑰是一种花，而花是一种植物。如果一种语言不提供对继承的直接支持，那么它就不是面向对象的。Cardelli 和 Wegner 将这种语言称为“基于对象（object-based）”的，而不是“面向对象（object-oriented）”的。在这个定义之下，Smalltalk、Object Pascal、C++、Eiffel、CLOS、C# 和 Java 都是面向对象的，Ada83 是基于对象的（后来在 Ada95 中加入了面向对象的支持）。但是，因为对象和类是这两种语言的基本元素，所以在基于对象和面向对象的语言中，都可以使用面向对象设计，而且也是非常推荐的。

2.2.2 面向对象设计

编程方法中的重点主要是正确有效地使用特定的语言机制，而设计方法的重点是正确有效地构造出复杂系统的结构。那么，究竟什么是面向对象设计（OOD）？我们建议予以下面的定义：

“面向对象设计是一种设计方法，包括面向对象分解的过程和一种表示法，这种表示法用于展现被设计系统的逻辑模型和物理模型、静态模型和动态模型。”

这个定义中有两个要点：（1）面向对象设计导致了面向对象分解，（2）面向对象设计使

用了不同的表示法来表达系统逻辑设计（类和对象结构）和物理设计（模块和处理架构）的不同模型，以及系统的静态和动态特征。

对面向对象分解的支持是面向对象设计与结构化设计的不同之处：前者利用类和对象抽象来构建逻辑系统结构，后者则利用算法抽象。我们用术语“面向对象设计”来指所有导致面向对象分解的方法。

2.2.3 面向对象分析

对象模型甚至对软件开发生命周期的更早阶段也产生影响。传统的结构化分析技术关注系统中的数据流，DeMarco [35]、Yourdon [36]、Gane 和 Sarson [37]的著作是其最好的代表，Ward 和 Mellor [38]以及 Hatley 和 Pirbhai [39]对其进行了实时扩展。面向对象分析（OOA）的重点在于构建真实世界的模型，利用面向对象的观点来看世界：

“面向对象分析是一种分析方法，这种方法利用从问题域的词汇表中找到的类和对象来分析需求。”

OOA、OOD 和 OOP 之间的关系如何？基本上，面向对象分析的结果可以作为开始面向对象设计的模型，面向对象设计的结果可以作为蓝图，利用面向对象编程方法最终实现一个系统。

2.3 对象模型要素

Jenkins 和 Glasgow 指出，“大部分程序员使用一种语言，并只使用一种编程风格。他们使用的编程方式是所用的语言强加给他们的。通常，他们没有机会换一种方式来思考问题，因此难以看到选择更适合手上问题的编程风格所带来的好处”[40]。Bobrow 和 Stefik 将编程风格定义为“一种组织程序的方式，基于某种编程概念模型和一种适合的语言，其目的是使得用这种风格编写的程序很清晰”[41]。他们进一步指出，存在 5 种主要的编程风格，这 5 种风格以及它们使用的抽象如下：

（1）面向过程　算法；
（2）面向对象　类和对象；
（3）面向逻辑　目标，通常以谓词演算的方式表示；
（4）面向规则　如果—那么规则；
（5）面向约束　不变的关系。

没有一种编程风格是最适合所有类型的应用的。例如，面向规则的编程可能最适合设计知识库，而面向过程的编程可能最适合设计计算密集的操作。根据我们的经验，面向对象风格最适合的应用范围最广，实际上，这种编程风格通常作为架构框架，被其他编程风格所使用。

每一种编程风格都是基于它自己的概念框架。对于所有面向对象的东西，概念框架就是

对象模型。这个模型有 4 个主要要素：

（1）抽象；

（2）封装；

（3）模块化；

（4）层次结构。

所谓“主要”，指的是如果一个模型不具备这些元素之一，就不是面向对象的。

对象模型有 3 个次要要素：

（1）类型；

（2）并发；

（3）持久。

所谓“次要”，指的是这些要素是对象模型的有用组成部分，但不是本质的。

如果没有这个概念框架，你可能在使用 Smalltalk、Object Pascal、C++、Eiffel 或 Ada 这样的语言编程，但是你的设计可能看起来像是 FORTRAN、Pascal 或 C 应用。更重要的是，你不太可能把握手上问题的复杂性。

2.3.1 抽象的意义

抽象是我们人类处理复杂性的基本方式。Dahl、Dijkstra 和 Hoare 指出，“抽象来自于对真实世界中特定对象、场景或处理的相似性的认知，并决定关注这些相似性而忽略不同之处”[42]。Shaw 将抽象定义为“对一个系统的一种简单的描述或指称，强调系统的某些细节或属性同时抑制另一些细节或属性。好的抽象强调了对读者或用户重要的细节，抑制了那些至少是暂时的非本质细节或枝节”[43]。Berzins、Gray 和 Naumann 建议，“只有当一个概念可以独立于最终使用和实现它的机制来描述、理解和分析时，我们才说这个概念是抽象的”[44]。结合这些不同的观点，我们将抽象定义如下：

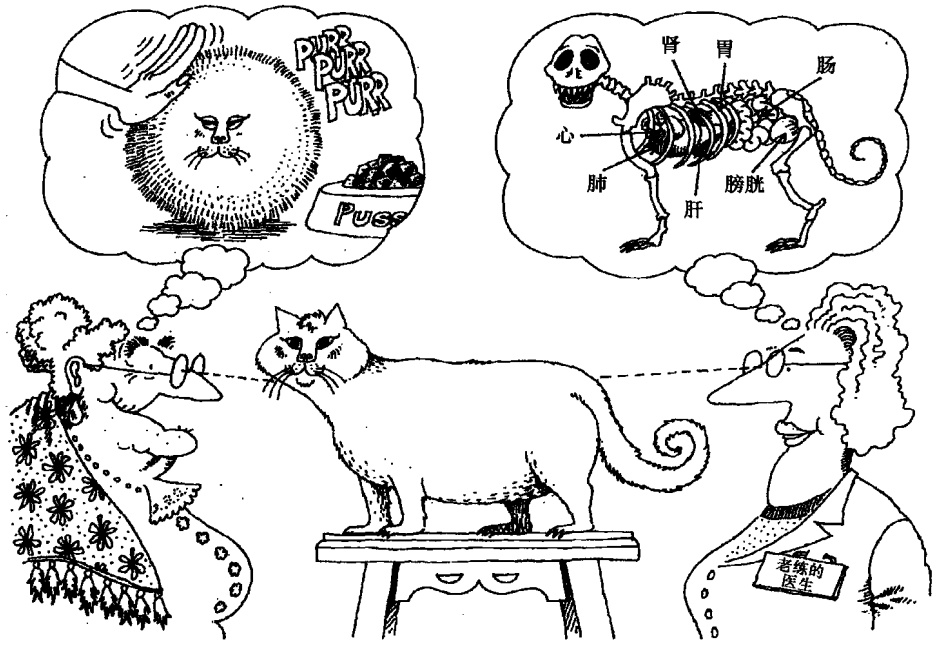
“抽象描述了一个对象的基本特征，可以将这个对象与所有其他类型的对象区分开来，因此提供了清晰定义的概念边界，它与观察者的视角有关。”

抽象关注一个对象的外部视图，所以可以用来分离对象的基本行为和它的实现。Abelson 和 Sussman 将这种行为/实现的分界称为抽象壁垒[45]，它是通过应用“最少承诺”原则来达成的。根据这个原则，对象的接口只提供它的基本行为，此外别无其他[46]。我们还想使用另一个原则，我们称之为“最少惊奇”原则，这个原则是指抽象捕捉了某个对象的全部行为，不多也不少，并且不提供抽象之外的惊奇效果或副作用。

对于给定的问题域决定一组正确的抽象，就是面向对象设计的核心问题。因为这个问题如此重要，所以我们将用第 4 章一整章的篇幅来讨论它。

“从那些准确地为问题域实体建模的对象到那些实际上没有什么理由存在的对象，存在着一系列的抽象。”[47]按最有用到最没有用的次序，这些抽象是：

- 实体抽象　一个对象，代表了问题域或解决方案域实体的一个有用的模型；
- 动作抽象　一个对象，提供了一组通用的操作，所有这些操作都执同类的功能；



抽象关注某个对象的基本特征，它与观察者的视角有关

- 虚拟机抽象　一个对象，集中了某种高层控制要用到的所有操作，或者这些操作将利用某种更低层的操作集；
- 偶然抽象　一个对象，封装了一组相互间没有关系的操作。

我们追求构建实体抽象，因为它们直接对应着给定问题域的词汇。

客户对象是使用其他对象（称为服务器对象）的资源的对象。可以通过考虑对象提供给其他对象的服务来总结一个对象的行为，以及它可能施加在其他对象上的操作。这种视角迫使我们集中关注对象的外部视图，导致了 Meyer 所谓的“编程契约模型”[48]：每个对象的外部视图定义了一份契约，其他对象可以依赖这份契约，而该对象则需要通过它的内部视图来实现这份契约（常常需要与其他对象协作）。这份契约因此建立了客户对象可以对服务器对象做出的所有假定。换言之，这份契约包含了对象的责任，即它的可靠的行为[49]。

单独来看，构成这份契约的每个操作都有一个唯一的签名，包含它所有的正式参数和返回值。我们把客户对象可以调用的整个操作集，以及这些操作合法的调用顺序，称为它的“协议”。协议表明了对象的动作和反应的方式，从而构成了抽象的完整静态和动态外部视图。

抽象思想的核心是不变性的概念。“不变量”（invariant）是某种布尔（真或假）条件，它的值必须保持不变。对于对象的每个操作，我们可以定义“前置条件”（precondition）（操作假定的不变量）和“后置条件”（postcondition）（操作满足的不变量）。违反一个不变量将破坏一个抽象相关的契约。如果违反了前置条件，这意味着客户没有完成它那部分的责任，因此服务器不能可靠地执行。类似地，如果违反了后置条件，这意味着服务器没有完成它那

部分的责任，所以客户不能再信任服务器的行为。出现异常表明某个不变量没有满足或不能满足。某些语言允许对象抛出异常，这样就可以中止处理，向其他对象报告问题，然后这些对象就可以捕捉异常并处理问题。

顺便提一下，术语“操作”、“方法”和“成员函数”是从三种不同的编程文化中发展而来的（分别是 Ada、Smalltalk 和 C++）。它们基本上指的是同样的东西，所以我们互换使用这些术语。

所有的抽象都有静态和动态的属性。例如，一个文件对象会占用特定存储设备上的一些空间，它有一个名字，也有内容。这些都是静态属性。这些属性的值是动态的，和对象的生命周期有关：一个文件对象可能变大或变小，它的名字可能改变，它的内容也可能改变。在面向过程风格的编程中，改变的动作和对象的动态值是所有程序的中心部分，当子程序被调用，语句被执行时，事情就发生了。在面向规则风格的编程中，当新的事件触发了规则，事情就发生了，它又可以进一步触发其他规则，如此等等。在面向对象风格的编程中，当我们操作一个对象时，事情就发生了（例如，向一个对象发出一个消息）。因此，调用一个对象的操作引发该对象的某种反应。我们可以有意义地执行一个对象上的哪些操作，以及该对象如何反应，构成了这个对象的全部行为。

抽象的例子

让我们通过一些例子来说明这些概念。关于如何发现给定问题的正确抽象，将在第 4 章讨论。

在一个溶液栽培的种植园中，作物是在营养液中栽培的，没有沙、碎石或其他土壤。维护正常的温室环境是一件精细的工作，这取决于栽培作物的种类和它的生长阶段。人们必须控制各种因素，如温度、湿度、光照、pH 值和营养液的浓度。在一个大型种植园中，拥有一个自动化的系统并不少见，这个系统持续监控并调整这些因素。简单地说，一个自动化园丁的目标是有效地种植作物，让多种作物健康地生长，并尽量减少人工干预。

这个问题的一个关键抽象是传感器。实际上，存在几种不同类型的传感器。任何影响产量的因素都必须被测量，所以必须有空气温度、水温、湿度、光照、pH 值和溶液浓度的传感器和其他一些传感器。从外部来看，温度传感器仅仅是一个对象，它知道如何测量某个具体位置的温度。什么是温度？它是一种数值，具有一定的范围和一定的精度，采用华氏、摄氏或开氏温标，只要选一种最适合我们问题的温标就行。什么是位置？它是种植园中某个可标识的地方，我们希望测量那里的温度。我们假定这样的位置不多。对于温度传感器来说，重要的不是它具体处于什么位置，而是它有一个位置，并有一个唯一的标识符，用来区分它和所有其他的传感器。现在我们可以问：温度传感器的责任是什么？我们的设计决策是，传感器负责知道某个位置的温度，并在询问时报告温度。更具体地说，一个客户可以对温度传感器执行哪些操作？我们的设计决策是，客户可以对它定标，并询问当前的温度。（参见图 2-6。注意，这种表示方法与 UML2.0 中的类的表示方法有点类似。第 5 章中将有准确的表示方法。）

到目前为止，我们描述的抽象都是被动的，某个客户对象必须操作一个空气温度传感器

对象，以确定它当前的温度。但是，存在另一种合理的抽象。它也许更适合，也许不太适合，这取决于我们可能做出的更大范围的系统设计决策。具体来说，让温度传感器不是成为被动的，而是成为主动的，这样它就不是由其他对象来激活，而是当它所在位置的温度改变了一定值时，激活其他对象。这种抽象与我们的第一种抽象几乎一样，只是它的责任有了一点小小的变化：传感器现在负责在温度变化时报告温度，而不只是在被询问的时候。这种抽象必须提供怎样的新操作呢？

这种抽象比第一种抽象要复杂一点（参见图 2-7）。这种抽象的客户可能调用一个操作来设定临界的温度范围。当传感器所在位置的温度上升或下降超过设定的范围时，它就负责报告。当函数被调用时，传感器提供它的位置和当前的温度，这样客户对象就有了足够的信息，可以根据情况做出反应。

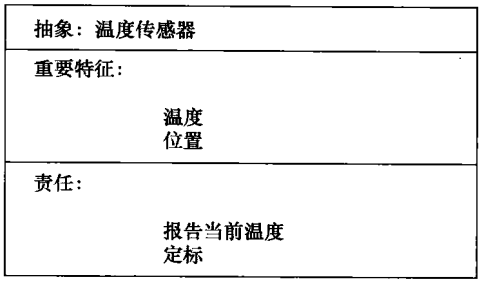


图 2-6 温度传感器的抽象

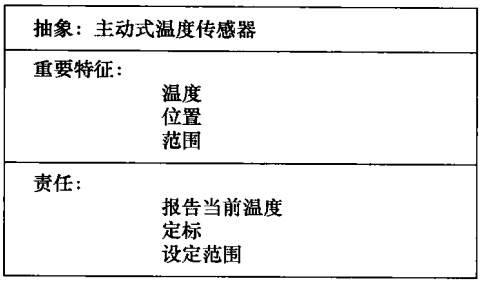


图 2-7 主动式温度传感器的抽象

主动式温度传感器如何执行它的责任，这是它的内部视图的功能，外部客户并不关心。这些东西成为这个类的秘密，由这个类的私有部分以及成员函数的定义共同实现。

让我们来考虑另一种抽象。对于每种作物，必须有一份培育计划，描述温度、光照、营养液和其他条件应该如何随时间的变化而变化。培育计划是一种合理的的实体抽象，因为它是问题域词汇的一部分。每种作物都有它自己的培育计划，但是所有作物的培育计划的形式都是一样的。

培育计划负责记录与培育作物有关的所有令人感兴趣的动作，对应到什么时间应该采取什么动作。例如，在某种作物生长的第 15 天，我们的培育计划可能是在 16 小时内将温度维持在 78°F，在这期间开灯 14 小时，然后在这一天的其他时间内将温度降到 65°F。我们也可能希望在这天的中间添加某种额外的营养液，并保持微酸的 pH 值。从培育计划外部的视角来看，客户必须能够建立起计划的细节，修改计划，并查询计划，如图 2-8 所示。（注意，抽象有可能随项目的生命周期而演进。随着细节的出现，“建立计划”这样的责任可能变成多项责任，如“设定温度”、“设定 pH 值”等。当收集到更多的客户需求知识，设计变成熟，考虑到实现的方式时，就会出现这种情况。）

我们的决定仍然是不要求培育计划自己执行它的计划：我们将这一点作为另一种抽象的责任（如一个计划控制器）。通过这种方式，我们在系统的不同逻辑部分之间创造了一种清楚的“分离关注”，这样就减小了单个抽象的概念规模。例如，可能有一个对象位于人机界面

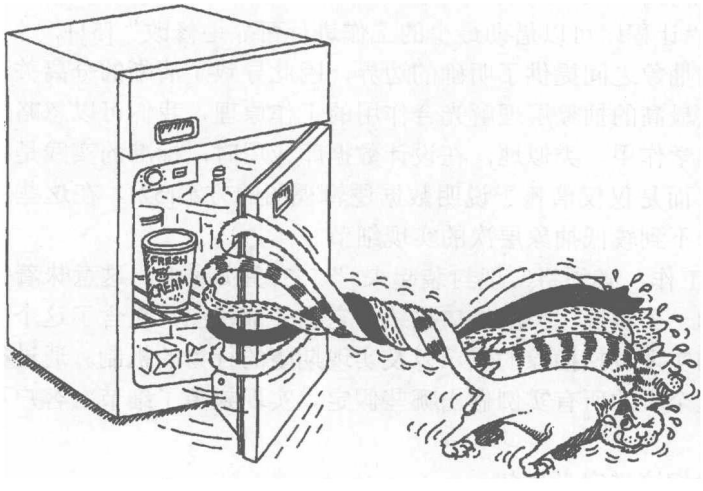
的边界上，将人的输入转变成计划。这个对象建立起培育计划的细节，所以必须能够改变培育计划对象的状态。必须有一个对象来执行培育计划，它必须能够读出特定时间的计划细节。

抽象：培育计划
重要特征： 名称
责任： 建立计划 修改计划 清除计划

相关的候选抽象：作物、条件、计划控制器

图 2-8 培育计划的抽象

这个例子指出，没有对象是孤立的，每个对象都与其他对象协作，实现某些行为[1]。这些对象之间如何协作的设计决策，定义了每种抽象的边界，从而也定义了每个对象的责任和协议。



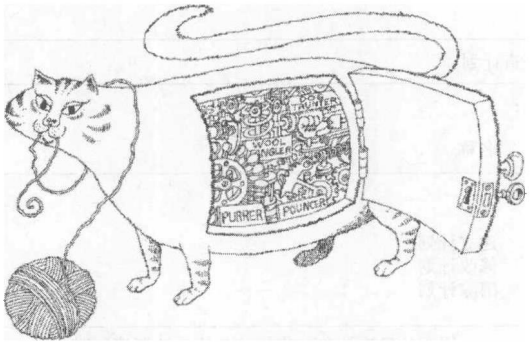
对象与其他对象协作，实现某种行为

2.3.2 封装的意义

虽然我们在前面曾将培育计划的抽象描述为一种时间/动作的映射，但是它的实现不一定必须是一张表或映射表这样的数据结构。实际上，选择哪种表示方式对培育计划的客户契约

[1] 换一种方式说，没有对象是一个孤岛（虽然岛屿可以抽象为一个对象），向诗人 John Donne 致歉。

并不重要，只要这种表示方式能实现这个契约就行。简单地说，对象的抽象应该优先于它的实现决定。当选择了一种实现之后，它就应该作为这种抽象后面的秘密，对绝大多数客户隐藏。



封装隐藏了对象实现的细节

抽象和封装是互补的概念：抽象关注对象的可以观察到的行为，而封装关注这种行为的实现。封装通常是通过信息隐藏来实现的（不只是数据隐藏）。信息隐藏是将那些不涉及对象本质特征的秘密都隐藏起来的过程。通常，对象的结构是隐藏的，其方法的实现也是隐藏的。“复杂系统的每一部分都不应该依赖于其他部分的内部细节。”[50]抽象“帮助人们思考他们做什么”，而封装“让程序可以借助最少的工作进行可靠地修改”[51]。

封装在不同的抽象之间提供了明确的边界，因此导致了清晰的分离关注。例如，再来看植物的例子。要从最高的抽象层理解光合作用的工作原理，我们可以忽略一些细节，如根的责任或细胞壁的化学作用。类似地，在设计数据库应用时，标准的实践是编写程序时不关心数据的物理表示，而是仅仅信赖于说明数据逻辑视图的方案[52]。在这些例子中，在一个抽象层次上的对象看不到较低抽象层次的实现细节。

“要让抽象能工作，必须将实现封装起来。”[53]在实践中，这意味着每个类必须有两个部分：一个接口和一个实现。类的接口描述了它的外部视图，包含了这个类所有实例的共同行为的抽象。类的实现包括抽象的表示以及实现期望的行为的机制。通过类的接口，我们能知道客户可以对这个类的所有实例做出哪些假定。实现封装了细节，客户不能对这些细节做出任何假定。

综上所述，我们这样定义封装：

“封装是一个过程，它分隔构成抽象的结构和行为的元素。封装的作用是分离抽象的概念接口及其实现。”

Britton 和 Parnas 将这些被封装的元素称为抽象的“秘密”。

封装的例子

为了说明封装的原理，让我们回到水培园艺系统。这个问题域中的另一个关键抽象是加热器。加热器是一种相当低层的抽象，因此我们可能决定，对这个对象可以执行的有意义的操作只有三种：打开它、关闭它、查看它是否在工作。

分离关注

我们没有让加热器负责维持一个固定的温度，而是把这个责任赋予了另一个对象（如加热控制器），它必须与一个温度传感器和一个加热器协作才能实现这个高级行为。把这个行为称为高级行为是因为，它是建立在温度传感器和加热器的简单语义基础之上的。它增加了某种新的语义，即“滞后现象”，这防止了加热器在温度接近边界条件时被快速打开和关闭。通过决定分离这部分责任，我们可以让每种抽象变得更为内聚。

关于加热器类，所有客户都要知道的是它提供的接口，即它在执行客户请求时会履行的责任（参见图 2-9）。

抽象：加热器
重要特征： 位置 状态
责任： 打开 关闭 提供状态信息

相关的候选抽象：加热控制器、温度传感器

图 2-9 加热器的抽象

来看看加热器的内部，我们会看到完全不同的情况。假设系统工程师决定将控制每个温室的计算机放在温室以外的某个地方（也许是为了避免恶劣的环境），并通过串行通信电缆将计算机连接到传感器和执行器上。加热器类的一种合理实现可能是利用一个电磁继电器来控制每个物理加热器，继电器由电缆发送的消息控制。例如，为了打开加热器，可能发出一条特殊的命令字符串，后面跟上一个数字，指明具体是哪个加热器，后面再跟上一个数字，表明是打开加热器。

假定出于某种原因，系统工程师决定使用内存映射 I/O 来代替串行通信电缆。我们不需要改变加热器的接口，但是实现会很不一样。客户根本不会看到任何变化，因为客户只能看到加热器接口。这就是封装的要点。实际上，客户不应该关心实现是怎样的，只要从加热器接收所需的服务就行。

接下来考虑培育计划类的实现。前面曾提到，培育计划本质上是一个时间/动作映射表。也许对这种抽象最合理的表示方式是一个时间/动作字典，利用一个开放的哈希表。不需要保存每个小时的动作，因为情况的变化没有这么快。可以只存放发生变化时的动作，让实现来推断两个时间点之间的情况。

在这种方式中，我们的实现封装了两个秘密：使用一个开放的哈希表（它明显是解决方案域中的词汇，不是问题域中的词汇）以及利用推断来减少存储需要（否则就要存放多得多

的时间/动作关系，来反映作物的整个生长季节）。这个抽象的客户不需要知道这些实现决定，因为它们实际上对这个类的外部可见的行为没有影响。

明智的封装使可能改变的设计决策局部化。随着系统的演进，它的开发者可能发现，在实际使用中，某种操作花的时间超过了可接受的范围，或者某些对象使用的空间超过了可用的空间。在这些情况下，对象的表示方法常常会改变，这样就可以采用更高效的算法，或者通过计算而不是存储某些数据来优化空间的使用。抽象的这种改变表示方法，同时又不影响其客户的能力就是封装的根本好处。

隐藏是一个相关的概念：在一个抽象层次隐藏起来的东西，在另一个抽象层次里可能代表了外部视图。对象的内部表示方法可能被揭示出来，但是绝大多数情况下，只有当这个抽象的创造者显式地暴露出实现，而且客户愿意接受由此而带来的额外的复杂性时，才会这样做。所以，封装不能阻止开发者做蠢事。正如 Stroustrup 所指出的，“隐藏是为了防止事故，而不是防止欺骗”[56]。当然，没有哪种程序设计语言防止人们看到一个类的实现，尽管操作系统可能拒绝你访问包含这个类实现的文件。

2.3.3 模块化的意义

“将一个程序分割到一些不同的组件中，这可以在某种程度上减少它的复杂性……虽然从这一点上来说，分割程序是有帮助的，但是分割程序的更大理由是它在程序内部创造了一些定义良好的、有文档描述的边界。这些边界，或者叫接口，对于理解程序是非常有价值的。”[57]在某些语言，如 Smalltalk 中，没有模块的概念，所以类就组成了分解的唯一物理单元。Java 有包的概念，包中包含类。在许多其他语言中，包括 Object Pascal、C++和 Ada，模块是一种独立的语言结构，确保了一组独立的设计决策。在这些语言中，类和对象构成了系统的逻辑结构，我们把这些抽象放入模块中，形成系统的物理架构。特别是对于较大型的应用来说，可能有成百上千个类，使用模块对管理复杂性有很大的帮助。



模块化将抽象打包成独立的单元

“模块化将程序划分为一些模块，这些模块可以独立地编译，但又与其他模块有联系。我们将使用 Parnas 的定义：‘模块之间的联系是模块相互之间所做出的假定’。”[58]大多数语言将模块作为一个独立的概念，它们也区分模块的接口和它的实现。因此，可以说模块化和封装是密不可分的。

对于一个给定的问题决定一组正确的模块，这和决定一组正确的抽象的难度几乎差不多。Zelkowitz 这样说肯定是正确的：“因为在设计阶段开始时我们可能不知道解决方案，分解为较小的模块可能相当困难。对于较老的应用（如写一个编译器），这个过程可能成为标准，但是对于新的应用（如防御系统或宇宙飞船的控制），这可能相当困难。”[59]

模块作为一种物理容器，我们在其中声明逻辑设计中的类和对象。这和电子工程师在设计计算机主板时的情况没有什么区别。NAND、NOR 和 NOR 等逻辑门可以用来构造必要的逻辑，但是这些门必须用标准集成电路的方式进行物理封装。由于缺少这样的标准软件部件，软件工程师拥有更大的自由度——就像电子工程师可以控制芯片厂一样。

对于很小的问题来说，开发者可能决定将所有的类和对象都声明在同一个包中。对于稍微有点实际意义的软件来说，更好的解决方案是将逻辑上相关的类和对象放在同一个模块中，只暴露出其他模块必须看到的元素。例如，考虑一个运行在一组分布式处理器上的应用，它使用消息机制来协调不同程序之间的动作。在一个大型系统，如命令与控制系统中，常常有几百甚至上千种这样的消息。一种很幼稚的策略可能是在各自的模块中定义每一种消息类。结果表明，这是一个非常差劲的决定。它不仅制造了文档噩梦，同时用户也很难找到他们需要的类。而且，当决定改变时，几百个模块都要修改或重新编译。这个例子说明，信息隐藏可能会造成相反的效果[60]。随意的模块化有时候比不实现模块化还要糟。

在传统的结构化设计中，模块化主要是考虑对子程序进行有意义地分组，利用耦合和内聚的判据。在面向对象的设计中，这个问题稍有不同。我们的任务是要决定类和对象的物理打包，这与子程序是明显不同的。

经验表明，有一些技术上和非技术上的指导方针可以帮助我们实现对类和对象的明智的模块化。Britton 和 Parnas 说：“分解为模块的总体目标是通过允许模块独立地设计和修改，从而减少软件的成本……每个模块的结构都应该足够简单，这样它就能被完全理解。应该能够在不知道其他模块的实现方法，并不会影响其他模块的行为的情况下，修改某个模块的实现。修改设计的容易程度应该能够满足需要变更的可能性。”[61]这些指导方针具有实践意义。在实践中，编译一个模块的成本相对来说是很小的：只有一个单元需要重新编译，然后重新链接应用。但是，重新编译模块接口的成本相对是较高的。特别是对于强类型的语言，开发者必须重新编译模块接口、模块实现及其他所有依赖该接口的模块。因此，对于很大型的程序来说（假定我们的环境不支持增量编译），对一个模块接口的改动可能导致长得多的编译时间。显然，开发经理不能忍受这种巨大的、“大爆炸式”的重新编译经常发生。出于这个原因，模块的接口应该尽可能的小，而又满足其他用到它的模块的需要。我们的风格是将尽可能多的东西隐藏到模块的实现中。与重写大量无关的接口代码相比，增量地从模块的实现到它的接口来改变声明要轻松很多。

开发者必须平衡两种竞争的技术考虑：封装抽象的愿望以及让其他模块看到某些抽象的需要。“可能独立变化的系统细节应该成为独立模块的秘密，模块之间存在的假定只能是那些不太可能变化的东西。每个数据结构对于一个模块来说都是私有的，它可能被这个模块中的一个或几个程序访问，但模块外的程序不能访问它。任何其他程序，如果需要保存在一个模块的数据结构中的信息，只能通过调用这个模块的程序获得。”[62]换言之，努力创造出高内聚（将逻辑上相关的抽象放在一起）、低耦合（减少模块间的依赖关系）的模块。从这个角度出发，我们可以这样定义模块化：

“模块化是一个系统的属性，这个系统被分解为一组高内聚、低耦合的模块。”

因此，抽象、封装和模块化的原则是相辅相成的。一个对象围绕单一的抽象提供了一个明确的边界，封装和模块化都围绕这种抽象提供了屏障。

另外有两个技术问题可能影响模块化的决定。首先，由于模块通常是软件的基本可分割单元，可以跨应用复用，所以开发者可能以方便复用的方式对类和对象进行打包。其次，许多编译器以分段的方式产生目标代码，每个模块生成一段。因此，对单个模块的规模可能有实际的限制。考虑到子程序调用的机制，模块中声明的位置可能在很大程度上影响引用变量的局部性，从而影响到虚存系统的分页行为。较差的局部性是指发生子程序跨段的调用，导致没有命中缓存并发生换页颠簸，最终降低了整个系统的执行速度。

还有一些竞争的非技术需求也可能影响模块化决定。通常，开发团队是根据模块来分配工作的，所以建立模块边界时要尽量减少开发组织中不同部分之间的接口。经验丰富的设计师通常负责模块的接口，经验较少的开发者完成模块的实现。从更大的范围来说，同样的情形也适用于合同分包的关系。我们可以对抽象进行打包，以便快速地稳定模块的接口，在不同公司之间达成一致意见。改变这样的接口通常涉及许多悲叹和愤怒（漫说还有许多纸面工作要做），所以这一原因通常导致保守设计的接口。谈到纸面工作，模块通常作为文档和配置管理的单元。有10个模块，也许就有人要做10倍的纸面工作，所以不幸的情况发生了，有时候文档方面的要求会影响模块设计的决定（通常是以最为消极的方式）。安全性也可能成为问题。大多数代码可能是非保密的，但最好将那些可能需要保密的代码放到一个独立的模块中。

对付这些不同的需求很困难，但不要忽略了最重要的一点：发现正确的类和对象，然后将它们放到不同的模块中，这基本上是独立的设计决定。类和对象的确定是系统逻辑设计的一部分，而模块的确定是系统物理设计的一部分。我们不能在物理设计之前完成所有逻辑设计，反之亦然。设计决策是以一种迭代的方式进行的。

模块化的例子

让我们来看看水培园艺系统中的模块化。假定我们决定使用一个商业产品工作站，让用户通过它来控制系统的操作。在这个工作站中，操作者可以创建新的培育计划，修改老的培育计划，跟踪当前执行的培育计划。由于在这里的关键抽象是培育计划，所以可能创建一个模块，目的是将所有与单个培育计划有关的类放在一起（如水果培育计划、谷物培育计划）。这些培育计划类的实现将包含在这个模块的实现中。我们可能还会定义一个模块，将所有与

用户界面功能相关的代码放在一起。

我们的设计可能还包含许多其他模块。最后，必须定义某个主程序，通过它来调用这个应用。在面向对象的设计中，定义这个主程序常常是最不重要的设计决策；而在传统的结构化设计中，这个主程序是根，是把所有东西聚在一起的基石。我们认为面向对象的方式更为自然，因为正如 Meyer 所说的，“将实际的软件系统描述为一组服务更为合适。用单个函数来定义这些系统通常是可行的，但这种做法显得太造作……真正的系统是没有顶的。”[63]

2.3.4 层次结构的意义

抽象是个好东西，但是除了那些太简单的应用之外，所有应用中都会包含许多不同的抽象，我们不能够一下子就理解它们。通过隐藏抽象的内部视图，封装有助于管理这种复杂性。但是，这还不够。一组抽象常常构成一个层次结构，通过在设计中确定这些层次结构，可以极大地简化对系统的理解。

我们将层次结构定义为：

“层次结构是抽象的一种分级或排序。”

在复杂系统中，最重要的两种层次结构是它的类结构（“是一种”层次结构）和对象结构（“组成部分”层次结构）。

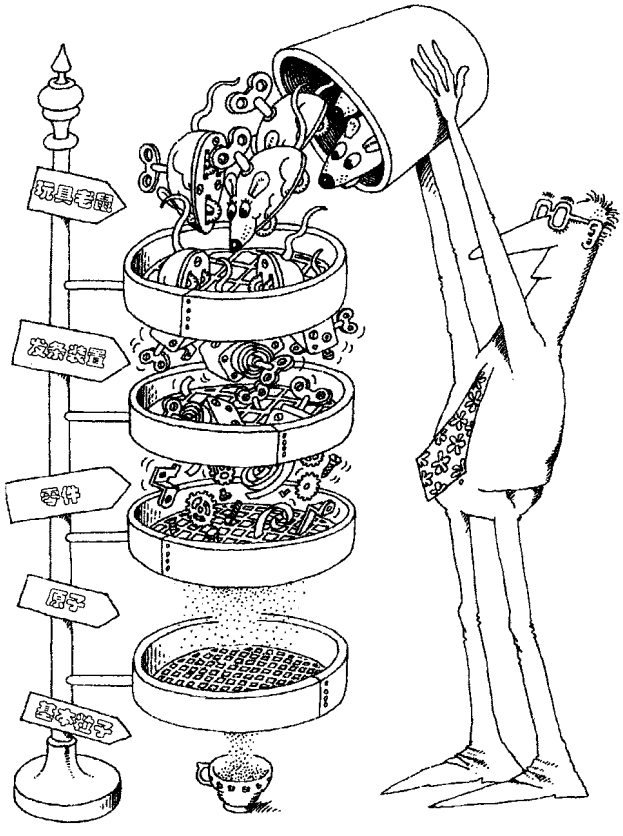
1. 层次结构的例子：单继承

继承是最重要的“是一种”层次结构，前面曾提到，它是面向对象系统的基本要素。继承基本上定义了类之间的关系，在这种关系中，一个类共享了一个或多个类（分别对应于单继承或多继承）中定义的结构或行为。继承因此代表了一种抽象的层次结构，在这个层次结构中，一个子类从一个或多个超类中继承。一般来说，子类会扩展或重新定义超类中的结构和行为。

从语义上说，继承表明了“是一种”关系。例如，熊“是一种”哺乳动物，房屋“是一种”有形资产，快速排序“是一种”具体的排序算法。继承因此实现了一种“一般/具体”的层次结构，其中子类将超类的一般结构和行为具体化。实际上，这也是继承的判据：如果 B 不是一种 A，那么 B 就不应该从 A 继承。

考虑在水培园艺系统中用到的不同类型的培育计划。前面我们描述了非常一般化的培育计划的抽象。然而，不同类型的作物需要特殊的培育计划。例如，对所有水果的培育计划一般比较相似，但与蔬菜或花卉作物的计划相比就有很大差别。由于抽象的这种分群特性，我们可以合理地定义一个标准水果培育计划，封装所有水果共同的行为，诸如何时授粉以及何时采摘等知识。可以断定，水果培育计划“是一种”培育计划。

在这种情况下，水果培育计划是更特殊的，培育计划是更一般的。谷物培育计划和蔬菜培育计划也是如此，即谷物培育计划“是一种”培育计划，蔬菜培育计划“是一种”培育计划。这里，培育计划是更一般的超类，其他的是特殊化的子类。



抽象构成了一个层次结构

当发展继承层次结构时，不同类中共同的结构和行为会被迁移到共同的超类中。这就是常把继承称为一般/特殊层次结构的原因。超类代表了一般化的抽象，子类代表了特殊的抽象，会添加、修改甚至隐藏来自超类的属性和方法。通过这种方式，继承让我们以一种经济的方式表达我们的抽象。实际上，忽略“是一种”层次结构将会导致膨胀的、不优雅的设计。“没有继承，每个类都会是一个独立的单元，每个都要从头开发。不同的类相互之间没有关系，因为每个类的开发者都根据他自己的选择来提供方法。所有跨类的一致性都源自于程序员的训练有素。继承让我们能够像比较新的概念和已经熟悉的概念那样定义新的软件，就像向新来的同事介绍某些概念一样。”[64]

在抽象、封装和层次关系中存在一种健康的压力。“数据抽象试图提供一个透明的边界，在这个边界之后，方法和状态是隐藏的。继承要求将这个接口开放到一定程度，允许不通过抽象来访问状态和方法。”[65]对于某一个类，通常有两种客户：调用这个类的实例的方法的对象以及从这个类继承的子类。Liskov 因此指出，通过继承，封装可以通过三种方式被打破：“子类可能访问其超类的实例变量，可以调用其超类的私有操作，或者直接引用其超类的超类”[66]。不同的程序设计语言在支持封闭和继承方面以不同的方式进行了折中。C++和 Java

提供了最大的灵活性。具体来说，类的接口可以有三个部分：私有部分，声明只能够由该类本身访问的成员；保护部分，声明可以由该类及其子类访问的成员；公有部分，可以被所有客户访问。

2. 层次结构的例子：多继承

前一个例子展示了单继承的应用：水果培育计划子类只有一个超类，即培育计划类。对于某些抽象，提供从多个超类的继承是有用的。例如，假定我们选择定义一个类来代表一种植物。对这个问题域的分析表明，花卉植物、水果和蔬菜具有一些特殊的属性，这些属性和我们的应用有关。例如，对于一种开花植物，预估它开花的时间和结籽的时间可能对我们很重要。类似地，对于所有水果和蔬菜，采摘的时间可能是抽象中的重要部分。为了体现我们的设计决策，一种办法是设计两个新类——一个花卉（Flower）类和一个水果蔬菜（FruitVegetable）类，它们都是植物（Plant）的子类。但是，如果需要对一种既开花，又结果的植物进行建模呢？例如，种花人常常采用苹果、樱桃和李子的花。对于这种抽象，需要设计第三个类，即花卉水果蔬菜类（FlowerFruitVegetable），它复制了来自花卉类和水果蔬菜类的信息。

所以，这种抽象更好的表达方式是利用多继承。首先，我们创建两个类分别表示花卉植物、水果蔬菜特有的属性。这两个类没有超类，它们是独立的。它们被称为“混入类”，因为它们将与其他类混合在一起，得到新的子类。

例如，我们可以定义一个玫瑰（Rose）类（参见图 2-10），它继承自植物（Plant）类和花卉混入（FlowerMixin）类。所以，子类 Rose 的实例可以既包含来自植物类的结构和行为，也包含来自花卉混入类的结构和行为。

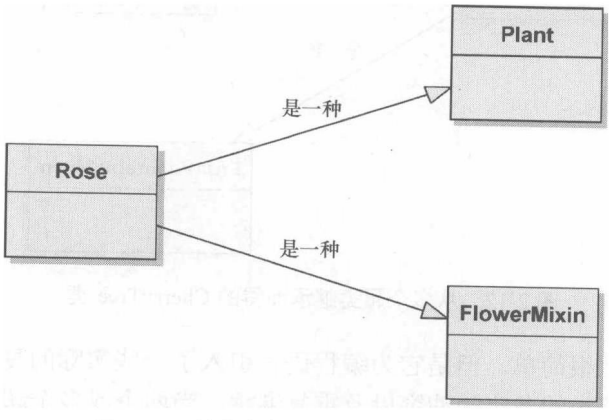


图 2-10 从多个超类继承而得的 Rose 类

类似地，胡萝卜（Carrot）类可以像图 2-11 所示的那样。在这两种情况下，都从两个超类继承而得到子类。

接下来，假定我们希望为樱桃树这样的植物定义一个类，它既有花也有果实。这可以像图 2-12 所示的那样。

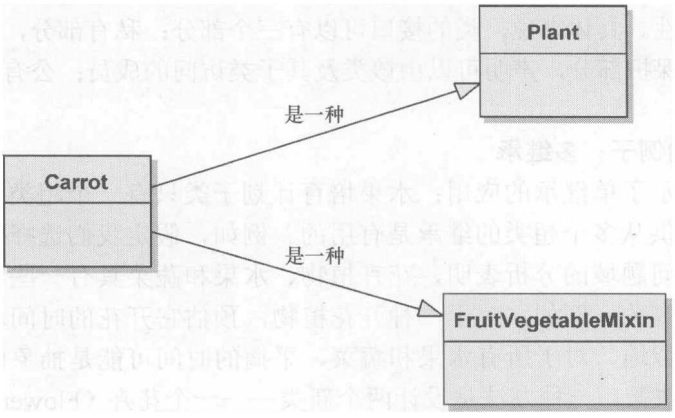


图 2-11 从多个超类继承而得的 Carrort 类

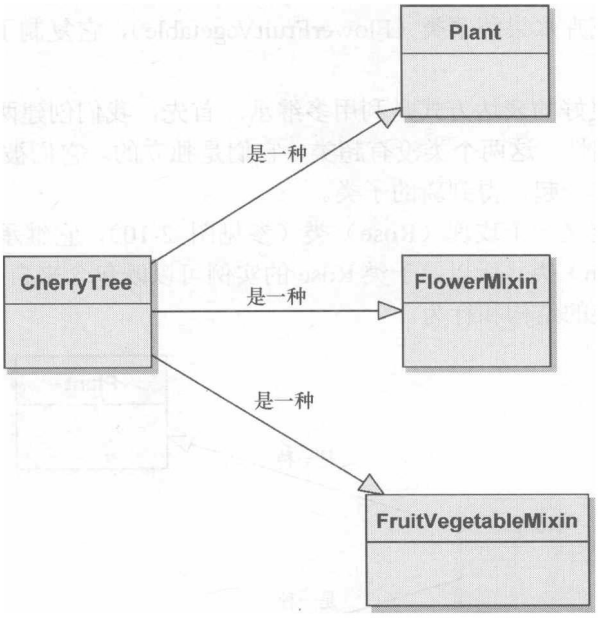


图 2-12 从多个超类继承而得的 CherryTree 类

多重继承在概念上很简单，但是它为编程语言引入了一些实际的复杂性。语言必须关注两个问题：来自不同超类的名字的冲突以及重复继承。当两个或多个超类提供了同样名字的属性或操作时，就会发生名字冲突的情况。

当两个或多个同辈超类具有共同的超类时，就会发生重复继承的情况。在这种情况下，继承的结构形成一个菱形，所以问题出现了，叶子类具有共同超类结构的一份拷贝还是多份拷贝？（参见图 2-13）。某些语言禁止重复继承，某些语言单方面地选择了一条路径，而另一些语言，如 C++，则允许由程序员来决定。在 C++中，虚基类用于说明一个共享的重复结

构，而非虚基类则导致子类中出现多个拷贝（通过显式的限定符来区分不同的拷贝）。

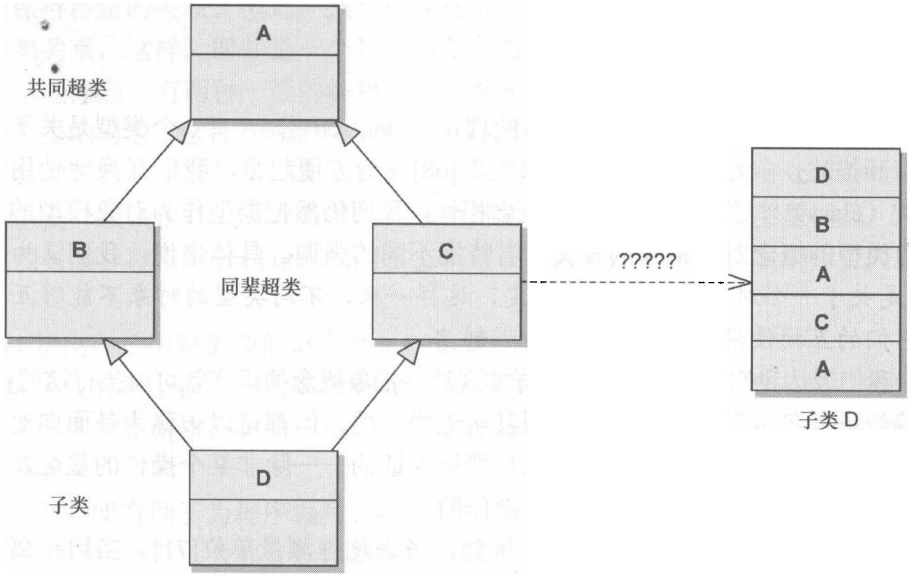


图 2-13 重复继承问题

多重继承常常被滥用。例如，棉花糖是一种糖，但显然不是一种棉花。同样，可以应用继承的判别测试：如果 B 不是一种 A，那么 B 就不应该从 A 继承。有可能的话，应该将错误的多继承结构缩减为一个超类加上其他类的聚合。

3. 层次结构的例子：聚合

“是一种”层次结构说明了一般/特殊关系，而“组成部分”层次结构则描述了聚合关系。例如，考虑种植园的抽象，可以认为种植园包含了一些植物和一份培育计划。换言之，植物是种植园的“组成部分”，培育计划也是种植园的“培育计划”。这种“组织部分”关系被称为聚合（aggregation）。

聚合不是面向对象开发或面向对象编程语言所特有的概念。实际上，只要是支持类似记录结构的语言都支持聚合。但是，将继承和聚合结合在一起的功能是强大的：聚合允许对逻辑结构进行物理分组，而继承允许这些共同的部分在不同的抽象中被复用。

在处理这样的层次结构时，常常提到抽象的层次，这个概念最早是由 Dijkstra[67]提出来的。对于“是一种”层次结构，高层的抽象是一般的，低层的抽象是具体的。因此，我们说花卉类的高层抽象是植物类。对于“组成部分”层次结构，一个类相对于组成它的实现的类来说，处于更高的层次。因此，种植园是植物类的更高层抽象。

聚合提出了所有权的问题。我们关于种植园的抽象允许不同的时间在种植园中培育不同的植物，但换一种植物不会改变种植园作为一个整体的特征，删除一个种植园也不会摧毁它的所有植物（它们可能被移植）。换言之，种植园和它的植物的生命周期是不相关的。与此不同的是，我们认为培育计划对象与种植园对象有着固有的关系，不能够独立存在。因此，当

创建一个种植园的实例时，也创建了一个培育计划实例，当删除种植园对象时，也会删除培育计划实例。

2.3.5 类型的意义

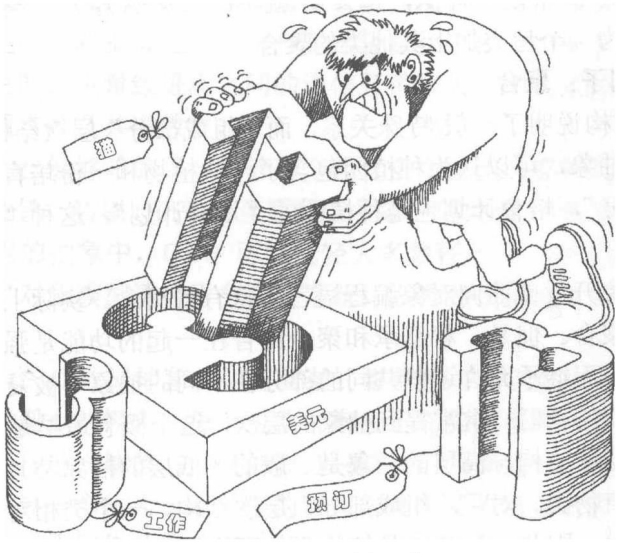
类型的概念主要来自于抽象数据类型的理论。Deutsch 指出，“一个类型是关于结构或行为属性的准确描述，一组实体共享这些属性”[68]。为方便起见，我们互换地使用术语类型（type）和类（class）[1]。虽然类型和类的概念相似，我们仍然把类型作为对象模型的一个独立要素，因为类型的概念对于抽象的含义有着特别不同的强调。具体来说，我们认为：

“类型是关于一个对象的类的强制规定，这样一来，不同类型的对象不能够互换使用，或者至少它们的互换使用受到非常严格的限制。”

类型让我们表达我们的抽象概念，这样实现这些抽象概念的语言就可以执行这些设计决策。

某种编程语言可以是强类型、弱类型甚至无类型的，但都可以被称为是面向对象的。例如，Eiffel 是强类型的，这意味着类型匹配是严格保证的——除非某个操作的签名在对象的类或超类中被定义过，否则是不能调用这个操作的。

类型匹配的概念是类型概念的核心。例如，考虑物理测量单位[71]。当用距离除以时间时，得到的值代表的是速度，而不是重量。类似地，用力的单位去除以温度是没有什么意义的，但是用力去除以质量是有意义的。这些都是强类型的例子，我们领域的一些规则，规定并执行某些合法的抽象组合。



强类型防止将抽象弄混

[1] 类型和类严格来说并不是一回事，某些语言区分这两个概念。例如，早期版本的 Trellis/Owl 语言允许对象既有一个类也有一个类型。在 Smalltalk 中，SmallInteger、LargeNegativeInteger 和 LargePositiveInteger 对象的类型是一样的，都是 Integer，虽然它们是不同的类[69]。但对于大多数人来说，区分类型和类的概念非常麻烦，也没有太多的价值。只要说类实现了类型就足够了。

强类型使得可以利用编程语言来执行某些设计决策，这样，当我们的系统的复杂度增长时，仍能保持特定的关联。但是，强类型也有不利的一面。从实践上来看，强类型引入了语义上的依赖关系，这样，即使是一个基类界面上的小改动，也需要重新编译所有的子类。

对于这些问题，有两种一般的解决办法。首先，可以使用一个类型安全的容器类，它只操作某一个类的对象。这种方法解决了第一个问题，即不同类型的对象被错误地混用。其次，可以使用某种运行时刻的类型标识，这解决了第二个问题，即知道当前处理的是哪一种类型的对象。但是一般来说，只有在有很好的理由时才会使用运行时刻的类型标识，因为它可能意味着削弱封装。稍后将会讨论，利用多态常常（但并非总是）可以缓解运行时类型标识的需要。

Tesler 指出，使用强类型的语言有一些重要的好处：

- 如果没有类型检查，大部分语言的程序可能在运行时以神秘的方式“崩溃”；
- 在大多数系统中，编辑—编译—调试循环相当繁琐，所以早期的错误检查是必不可少的；
- 类型声明有助于为程序编写文档；
- 如果声明类型，大部分编译器可以生成更有效率的目标代码。[72]

无类型的语言提供了更大的灵活性。但是，即使是无类型的语言，正如 Borning 和 Ingalls 所说的，“在大多数情况下，程序员实际上知道消息的参数需要哪种对象，以及会返回哪种对象”[73]。在实践中，强类型语言所提供的安全性常常能够补偿不使用无类型语言所带来的灵活性损失，特别在是大规模编程中。

类型的例子：静态类型和动态类型

类型的强与弱和类型的静态与动态的概念是完全不同的。类型的强与弱指的是类型一致性，而类型的静态与动态指的是名字与类型绑定的时间。静态类型（也被称为静态绑定或早期绑定）意味着所有变量和表达式的类型在编译时就固定了，动态类型（也被称为迟后绑定）意味着所有变量和表达式的类型直到运行时刻才知道。一种语言可以既是强类型的也是静态类型的（Ada），既是强类型的也支持动态类型（C++、Java），或者既是无类型的也支持动态类型（Smalltalk）。

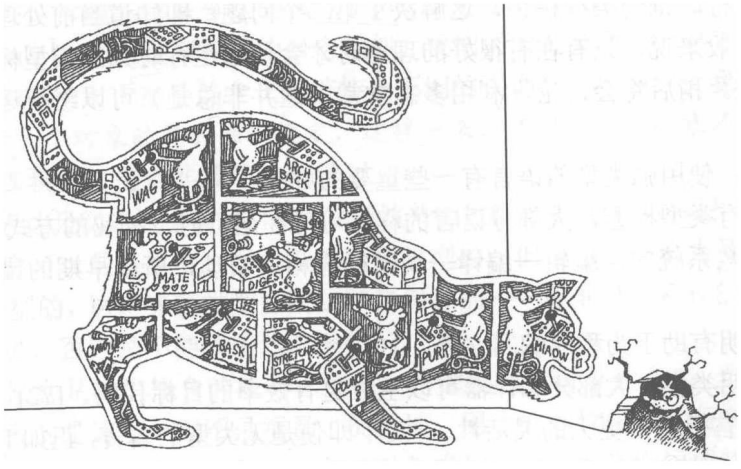
多态（polymorphism）是动态类型和继承互相作用时出现的一种情况。多态代表了类型理论中的一个概念，即一个名字（如一个变量声明）可以代表许多不同类的对象，这些类具有某个共同的超类。这个名字所代表的对象因此可以响应一组共同的操作[74]。与多态相对的是单态（monomorphism），这在强类型和静态类型的语言中都可以发现。

多态可能是面向对象语言中除了对抽象的支持以外最强大的功能，也正是它，区分了面向对象编程和较传统的抽象数据类型编程。在接下来的章节中可以看到，多态也是面向对象设计中的核心概念。

2.3.6 并发的意义

对于某些类型的问题，一个自动化的系统可能同时处理许多不同的事件。另外一些问题

则可能需要很多的计算，超出了单个处理器的能力。在这些情况下，很自然要考虑利用一组分布式的计算机来实现目标，或者利用多任务。每个程序至少都有一个控制线程，但涉及并发的系统可能有许多这样的线程：某些线程是短暂出现的，而另一些线程会伴随系统执行的整个生命周期。跨多个 CPU 执行的系统允许真正并发的控制线程，而运行在单 CPU 上的系统只能够实现模拟的并发控制线程，这通常是利用某种时间片算法来实现的。



并发允许不同的对象同时行动

我们也区分重量级并发和轻量级并发。重量级进程通常是由目标操作系统独立管理的，所以它有自己的地址空间。轻量级进程通常与其他轻量级进程一起处于单个操作系统进程之内，共享同样的地址空间。重量级进程之间的通信开销很大，涉及某种进程间通信技术；轻量级进程开销较小，通常涉及共享数据。

构建一个大型软件已经够难的了，设计一个包含多个控制线程的大型软件更是难得多，因为设计者必须考虑死锁、活锁、饥饿、互斥和竞争条件等问题。“在最高层次的抽象中，通过将并发隐藏在可复用的抽象中，OOP 可以减轻大多数程序员在并发问题上的负担。”[76] Black 等人因此建议，“对象模型适合于分布式系统，因为隐式地定义了发布和移动的单元以及实体的通信”[77]。

虽然面向对象编程关注数据抽象，但是封装、继承和并发关注了过程抽象和同步[78]。对象是统一这两种不同观点的概念：每个对象（来自于真实世界的一个抽象）都可以代表一个独立的控制线程（一种过程抽象）。这样的对象被称为“主动的”。在基于面向对象设计的系统中，我们可以将世界概念化为一组协作的对象，其中某些是主动的，因此作为独立活动的中心。出于这个概念，我们将并发定义如下：

“并发是一种属性，它区分了主动对象和非主动对象。”

并发的例子

考虑一个名为 ActiveTemperatureSensor 的传感器，它的行为要求定期测量当时的温度，然后通知客户与设定值相比，温度是否改变了一定的数值。我们不解释这个类是如何实现这

种行为的。实际情况是实现的秘密，但很清楚，需要某种形式的并发。

一般来说，在面向对象的设计中有三种并发方式。其一，并发是某种编程语言的内在特征，语言提供了并发和同步的机制。在这种情况下，可以创建一个主动对象，它与其他主动对象一起并发执行某些处理过程。

其二，可以使用一个类库来实现某种形式的轻量级进程。自然，这种实现是高度平台相关的，虽然这个库的接口相对来说也许是可移植的。在这种方式中，并发不是语言的内在特征（因此不会在非并发系统上增加任何负担），但通过这些标准类，并发看起来似乎是内在的特征。

其三，可以利用中断来实现并发的假象。当然，这要求我们具有某些底层硬件细节的知识。例如，在 ActiveTemprature Sensor 类的实现中，可能有一个硬件时钟，它定期中断应用。在中断发生的时候，所有这类传感器读取当时的温度，并根据需要调用回调函数。

不论采用哪种方法实现并发，都必须注意这样一个事实：当在一个系统中引入并发时，必须考虑主动对象之间、主动对象与纯粹串行执行的对象之间，如何同步它们的活动。例如，如果两个主动对象试图给第三个对象发送消息，必须确保使用了某种互斥手段，这样，被调用对象的状态才不会因两个主动对象试图同时更新它而被破坏。这是一起使用封装、抽象和并发时必须注意的要点。在并发的情况下，仅定义对象的方法是不够的，还必须确保这些方法的语义在多个控制线程的情况下仍然有效。

2.3.7 持久的意义

软件中的一个对象会占用一定量的空间，在一定的时间内存在。Atkinson 等人指出，对象的存在有一个区间——从计算对象表达式时产生的瞬时对象，到保存在数据库中、超过程序执行时间的对象。这个对象持久的谱系包括：

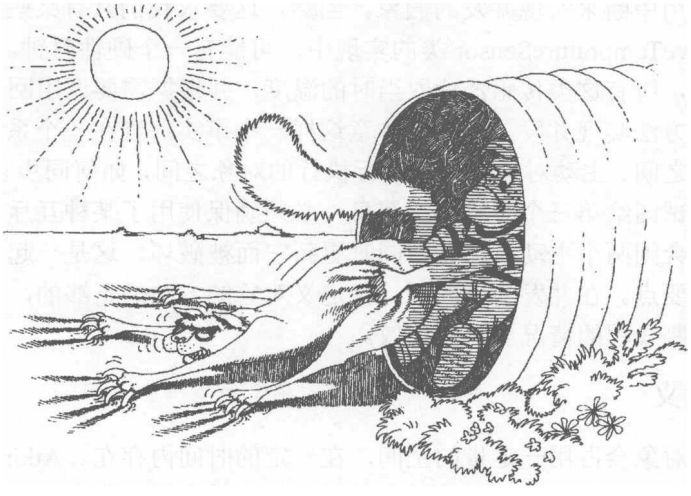
- 表达式计算的瞬时结果；
- 过程执行时的局部变量；
- 自有变量（像 ALGOL 60 中那样）、全局变量、堆中的值，它们的存在时间与它们的有效范围不同；
- 在程序执行之间存在的数据；
- 在程序的不同版本之间存在的数据；
- 比程序生命期长的数据[79]。

传统编程语言通常只关注前三种类型的对象持久，后三种类型的持久通常是数据库技术关注的领域。这导致了一些文化上的冲突，有时候形成了非常奇怪的架构：程序员会弄出比较随意的方案来存放对象，这些对象的状态在程序执行之间必须被保存，而数据库设计者错误地应用他们的技术来处理一些临时对象[80]。

关于 Atkinson 等人的“比程序生命期长的数据”，有一种有趣的变化，即在 Web 应用中，在整个事务执行过程中，应用可以不连接所使用的数据。在和数据源断开连接时，提供给客户应用或 Web 服务的数据可能发生怎样的变化？这两者之间的问题如何解决？类似微软的

ActiveX Data Object for .NET（ADO.NET）这样的框架被提了出来，以帮助解决这种分布式的、断开连接的情况。

统一对象和并发的概念导致了并发的面向对象程序设计语言。与此类似，在对象模型中引入持久的概念导致了面向对象的数据库。在实践中，这样的数据库是建立在已经证明的技术之上的，如序列化、排序、层次数据库、网状数据库或关系数据库模型。但是，它为程序员提供了一种面向对象接口的抽象，数据库查询和其他操作都是通过这个接口，以对象的方式完成的，这些对象的生命周期超过了单个程序的生命周期。这种统一极大地简化了某些类型应用的开发。具体来说，它允许我们对数据库和应用的非数据部分使用同样的设计方法。



持久跨越时间或空间保存了一个对象的状态和类

某些面向对象编程语言提供了对持久的直接支持。Java 提供了 Enterprise Java Beans（EJB）和 Java Data Object。Smalltalk 有一些协议，将对象序列化到存储设备，或者从存储设备中取出并恢复对象（对于子类，要重新定义序列化的方法）。但是，将对象序列化到普通文件中只是比较初级的持久解决方案，因为这不适合处理大量对象的情况。持久可以通过一些商业面向对象数据库来实现[81]。更典型的持久方法是为关系型数据库提供一层面向对象的皮肤。开发者可以自行定义对象—关系映射。但是，这也是一项很有挑战性的任务。有一些框架简单化了这项任务，如开放源代码的 Hibernate[85]。商业的对象—关系映射软件也是有的。如果已经在关系型数据库上投入了很多资金，将其替换掉会有很大风险或者代价昂贵，那么这种方法是最有吸引力的。

持久解决的不只是数据的生命周期问题。在面向对象的数据库中，不仅是对象的状态会被持久，对象的类也会超越单个应用而存在，这样，每个程序都会以同样的方式来解释这种保存的状态。随着数据库的增长，这显然使得保持数据库的完整性变得很有挑战性，特别是当必须改变一个对象的类时。

到目前为止，我们的讨论均与时间上的持久有关。在大多数系统中，一个对象被创建之

后会使用同样的物理地址，直到它不存在为止。但是，对于在一组分布式的处理器上执行的系统来说，有时候必须考虑跨空间的持久。在这样的系统中，考虑对象从一台计算机移动到另一台计算机是有意义的，它在不同计算机上甚至会有不同的表现形式。

综上所述，我们这样定义持久：

“持久是对象的一种属性，利用这种属性，对象跨越时间（例如，当对象的创建不存在了的时候，对象仍然存在）和空间（例如，对象的位置从它被创建的地址空间移开）而存在。”

2.4 应用对象模型

我们已经看到，对象模型从本质上与传统的结构化分析、结构化设计和结构化编程方法所使用的模型是很不一样的。这并不是说对象模型放弃了这些较老方法中好的原则和经验。相反，它引入了一些创新的元素，建立在这些更早的模型之上。因此，对象模型提供了一些其他模型没有提供的重要好处。最重要的是，使用对象模型能让我们构建的系统包含良好构造的复杂系统的 5 个属性。第 1 章中介绍了这 5 个属性：层次结构、相对的基础（如多个抽象层次）、分离关注、模式、稳定的中间状态。根据我们的经验，应用对象模型可以得到另外 5 个实际的好处。

2.4.1 对象模型的好处

首先，使用对象模型帮助我们探索基于对象和面向对象编程语言的表达能力。Stroustrup 指出，“如何最好地利用 C++这样的语言提供的好处，人们并不总是很清楚。将 C++作为一种带有一点数据抽象的、更好的 C 来使用，也能够在开发效率和代码质量方面得到很大改进。但是，在设计过程中利用类层次结构的好处，能够得到更进一步、更显著的改进。这常常被称为面向对象设计，这也是人们发现的使用 C++的最大好处”[82]。经验表明，如果不应用对象模型的这些要素，那么像 Smalltalk、C++、Java 这样的语言的更多强大的功能要么完全被忽略，要么被误用了。

其次，利用对象模式不仅鼓励软件的复用，而且鼓励整个设计的复用，这导致了可复用应用框架的产生[83]。我们发现，面向对象系统通常比等价的非面向对象实现更小。这不仅意味着编写更少的代码，而且软件复用度的提高也会反映到成本和开发进度上。但是，复用不会自动发生。如果复用不是项目的主要目标，那么它就不太可能实现。另外，复用设计可能在初次实现可复用的组件时花费更多。好消息是，初次的成本将在组件的后续使用中得到补偿。

再次，使用对象模型将得到构建在稳定的中间状态之上的系统，这样的系统更适合变化。这也意味着，这样的系统可以随时间而进化，而不是在对需求进行第一次主要修改时就抛弃原来的系统或者完全重新设计。

第 7 章将进一步探讨对象模型如何减少开发复杂系统所固有的风险。这第 4 项好处主要

是因为集成散布在生命周期的各个时刻，而不是作为一个大事件发生。对象模型设计明智的分离关注的原则也减少了开发的风险，增加了我们对设计正确性的信心。

最后，对象模型引起了人类认知的工作的兴趣。Robson 说，“许多不知道计算机如何工作的人会发现，面向对象系统的思想非常自然”[84]。

2.4.2 开放式问题

为了有效应用对象模型的诸要素，接下来必须关注如下一些开放式问题：

- 究竟什么是类和对象？
- 如何正确地确定与特定应用有关的类和对象？
- 怎样的表示法适合表示面向对象系统的设计？
- 怎样的过程可以导致结构良好的面向对象系统？
- 使用面向对象设计对管理层意味着什么？

这些问题是后面 5 章的主题。

2.5 小结

- 软件工程的成熟导致了面向对象分析、设计和编程方法的形成，所有这些技术都是为了解决大规模编程的问题。
- 有一些不同的编程模式：面向过程的、面向对象的、面向逻辑的、面向规则的、面向约束的。
- 抽象描述了一个对象的基本特征，可以将这个对象与所有其他类型的对象区分开来，因此提供了清晰定义的概念边界，它与观察者的视角有关。
- 封装是一个过程，它被分隔构成抽象的结构和行为的元素，封装的作用是分离抽象的概念接口及其实现。
- 层次结构是抽象的一种分级或排序。
- 类型是关于一个对象的类的强制规定，这样，不同类型的对象不能够互换使用，或者至少它们的互换使用受到非常严格的限制。
- 并发是一种属性，它区分了主动对象和非主动对象。
- 持久是对象的一种属性，利用这种属性，对象跨越时间和空间而存在。

类与对象

工程师和艺术家都必须非常熟悉他们那一行的材料。油画颜料和水彩、钢材和铝材、螺钉和钉子、对象和类——这些材料的功能都是类似的（例如，螺钉和钉子都起到固定作用），但每种材料又有自己特别的特点和用法。建筑师可能不知道最有效的敲钉子方法（这是木匠的专业技能），但建筑师必须懂得什么情况下要用钉子、螺钉、胶水或焊接。忽视这些基本考虑可能导致灾难性的后果。

我们使用面向对象的方法来分析和设计复杂的软件系统，基本构建块是类和对象。因为到目前为止只提供了这两种元素的非正式定义，所以在本章中，我们将详细讨论类和对象的本质以及它们的关系。在这个过程中，我们将提供一些关于得到高品质的抽象和机制和经验法则。

3.1 对象的本质

识别物理对象的能力是人类在早期就掌握的技能。一个色彩鲜艳的球会吸引婴儿的注意力，但是通常情况下，如果把球藏起来，小孩不会试图去寻找它。当对象离开了她的视野时，根据她的判断，这个对象就不存在了。一般直到接近一岁时，小孩才会建立起所谓“对象概念”，这种能力对于将来认知的发展相当重要。向一个一岁小孩展示一个球，然后再藏起来，她通常会寻找这个球，即使这个球不在视野之中。通过对象的概念，小孩意识到对象具有持久性和标识符，这与施加在对象上的操作无关[1]。

3.1.1 什么是对象，什么不是对象

第1章中，我们非正式地将对象定义为一个可以触摸的实体，展示了一些定义良好的行为。从人类认知的角度来看，对象可以是下列事物之一：

- 一个可以触摸或可以看见的东西；

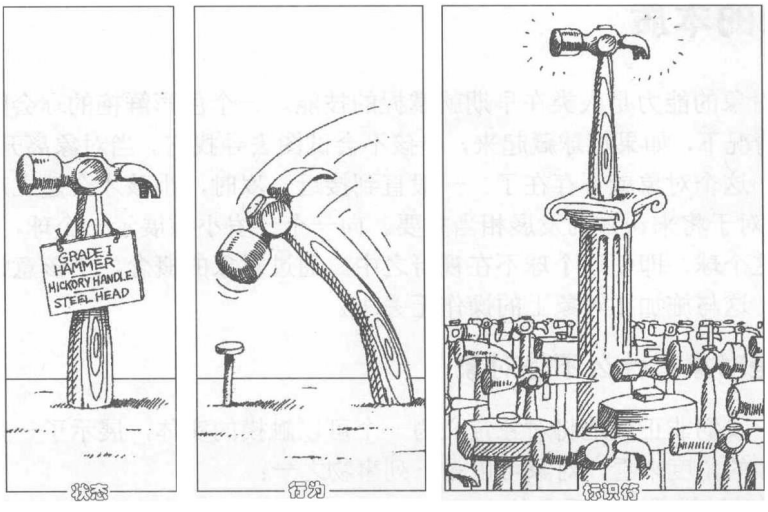
- 在智力上可以理解的东西；
- 可以指导思考或行动的东西。

我们对非正式的定义进行了补充，一个对象反映了某一部分的真实存在，因此它是在时间和空间中存在的某种东西。在软件中，“对象”这个术语首先正式出现在 Simula 语言中，对象通常存在于 Simula 程序中，用于模拟真实世界的某个方面[2]。

我们在软件开发过程中关注的对象不仅仅是真实世界中的对象。在设计过程中，我们创造出一些重要的对象，它们与其他对象的协作形成了一些机制，从而提供某种更高级的行为[3]。Jacobson 等人将“控制对象”定义为“将事件过程组织起来的对象，负责与其他对象的通信”[62]。这导致了 Smith 和 Tockey 的更明确的定义，他们认为，“一个对象代表了单个可识别的项、单元或实体，它可以是真实的，也可以是抽象的，在问题领域中承担定义良好的角色”[4]。

考虑一个制造工厂，它处理各种材料，制造多种产品，如自行车架和飞机机翼。制造工厂通常被划分成独立的车间：机械、化工、电子等。车间又被进一步划分成单元，每个单元中放着一组机器设备，如模压机、锻压机、车床等。在制造生产线上，我们可以看到装着原材料的桶。这些原材料用在一些化学处理过程中，制备一些复合材料，这些复合材料又经过成型，用于生产自行车架、飞机机翼和其他一些部件。前面提到的这些可触摸的东西都是对象。车床定义了一个明确的边界，将它与它处理的复合材料区分开来。自行车架定义了一个明确的边界，将它与生产自行车架的机器区分开来。

某些对象可能有明确的概念边界，但其代表的是不可触摸的事件或过程。例如，在制造工厂中的一个化学处理过程可能被作为一个对象，因为它具有明确的概念边界，通过一组有序的操作，在不同的时间与某些其他对象打交道，并展示出定义良好的行为。类似地，考虑一个对实心体建模的 CAD/CAM 系统。有一个立方体和一个球体相交，它们相交线是一条不规则的曲线。虽然它离开了球体或立方体就不存在，但这条线仍是一个对象，它有明确定义的概念边界。



一个对象具有状态，能展示某种定义良好的行为，且具有唯一的标识符

某些对象可能是可触摸的，但是物理边界不太清晰。像河流、雾、人群就属于这种类型的对象[1]。就像一个人拿着锤子就喜欢把世界上所有的东西看作是钉子一样，具有面向对象思想的开发者开始认为世界上所有的东西都是对象。这种观点有点幼稚，因为某些东西显然不是对象。例如，美和色彩这样的属性就不是对象，爱和恨这样的感情也不是对象。但是，这些东西有可能成为其他对象的属性。例如，我们可以说一个男人（一个对象）爱他的妻子（另一个对象），或者说某只猫（又一个对象）是灰色的。

因此，说对象是有明确定义的边界的东西是有意义的，但还不足以让我们能够区分不同的对象，也无法让我们判断这种抽象的品质。根据我们的经验，建议使用下面的定义：

“一个对象是一个具有状态、行为和标识符的实体。结构和行为类似的对象定义在他们共同的类中。“实例”和“对象”这两个术语可以互换使用。”

在接下来的小节中将会更详细地讨论状态、行为和标识符的概念。

3.1.2 状态

考虑一个分发软饮料的自动售货机。这类对象的一般行为是当某人在投币口塞进钱，并按下选择按钮时，机器就会提供一种饮料。如果用户先选择饮料再向投币口塞钱会怎样？大部分自动售货机什么也不会干，因为用户违反了它们基本的操作假定。换言之，用户没有注意到（先进行了选择）自动售货机处于一种状态（等待塞钱）。类似地，假设用户忽略了警示灯上说的“只收准确的零钱”，而放入了多余的钱。大多数机器不会为用户着想，它们会很高兴地吞下多余的钱。

在这些情况下，可以看到对象的行为如何受到它的历史的影响：人们操作一个对象的次序是重要的。这种行为依赖事件或依赖时间的原因是对象内部存在状态。例如，自动售货机的一个基本状态是用户塞入了一定数量的钱，但还没有进行选择。其他重要的属性包括可找的零钱数和软饮料的数量。

从这个例子中，我们可以得到下面的基本定义：

“对象的状态包括这个对象的所有属性（通常是静态的）以及每个属性当前的值（通常是动态的）。”

自动售货机的另一个属性是它可以接受钱币。这是一个静态（即固定）属性，意味着这是自动售货机的一项基本特征。与之相对的是，某一时刻它实际接受的金额代表着这个属性的动态值，它受到对机器操作次序的影响。当用户塞入钱币时，这个数量就增加了；当提供产品之后，这个数量就减少了。说这些值“通常是动态的”，是因为在某些情况下这些值是静态的。例如，一台自动售货机的流水号就是一个静态的属性和值。

一个属性是一种内在或独特的特征、特点、品质或特性，使一个对象区别于别的对象。例如，电梯的一个基本属性就是它只能够上下运动，而不能水平运动。属性通常是静态的，因为这样的特征是不可更改的，是对象的根本本质。说“通常是静态的”，是因为在某些情况

[1] 只有当抽象程度足够高时，才是这样的。对于一个在雾堤（fog bank）中行走的人来说，区分“我的雾”和“你的雾”通常是无用的。但是，考虑一张气象地图：旧金山的一个雾堤与伦敦的一个雾堤显然是不同的对象。

下，对象的属性会改变。例如，考虑一个能从环境中学习的机器人。它可能开始将一个出现的对象当成是一个固定的障碍物，后来却发现这个对象是一个可以打开的门。在这种情况下，随着知识的增长，机器人在构建世界的概念模型时创建的这个对象获得了新的属性。

所有的对象都有某种值，这个值可能是一个简单的数量，也可能代表另一个对象。例如，电梯的状态可能包括一个值 3，这表示电梯当前所处的楼层。在自动售货机的例子中，它的状态包含许多其他对象，例如一些软饮料。每个软饮料实际上都是不同的对象，它们的属性与自动售货机的属性不同（它们可以被喝掉，而自动售货机不可以），而且它们操作方式也完全不同。因此，我们可以区分对象和简单的值：像数字 3 这样简单的数量是“不受时间影响的、不可变化的、不可实例化的”，而对象是“有存在时间的、可以变化的、有状态的、可以实例化的，可以被创建、销毁和共享”[6]。

每个对象都有状态。这一事实意味着，每个对象都会在物理世界或计算机内存中占据一定的空间。

可以说，系统中所有的对象都封装了某种状态，系统中所有的状态都由对象所封装。封装一个对象的状态只是开始，这并不足以让我们刻画出在开发过程中发现的这种抽象的全部含义（请参考示例 3-1，它展示了一个简单的抽象如何演化）。出于这个原因，必须考虑对象的行为。

示例 3-1

考虑一个雇员记录的抽象。图 3-1 用统一建模语言的类表示法展示了这个抽象。（关于 UML 表示法的更多内容，请参见第 5 章。）

这个抽象的每一部分都代表了雇员抽象的一种属性。这个抽象不是一个对象，因为它没有代表某个特定的实例。当具体化时，例如我们有两个不同的对象——Tom 和 Kaitlyn，每个对象都在内存中占据了一定的空间（参见图 3-2）。

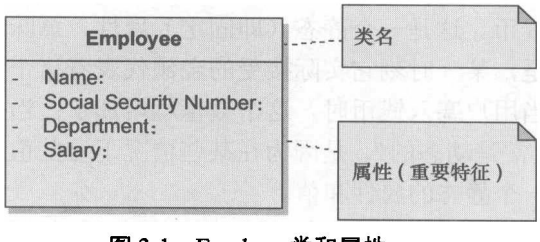


图 3-1 Emplyee 类和属性

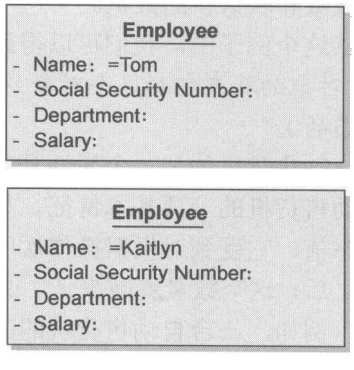


图 3-2 Employee 对象 Tom 和 Kaitlyn

这些对象不会与其他对象共享空间，虽然它们都有相同的属性。它们的状态具有共同的表现形式。

与暴露对象的状态相比，封装对象的状态是一项好的工程实践。例如，可以把这个抽象

（类）修改成如图 3-3 所示的样子。

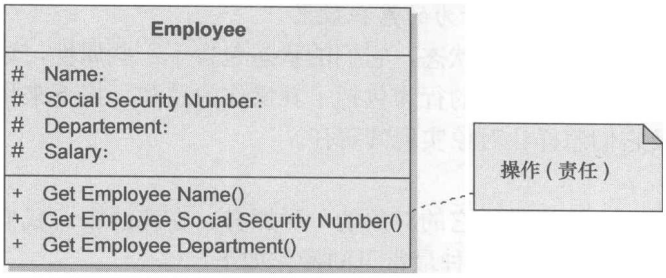


图 3-3 具有保护属性和公有操作的 Employee 类

这个抽象比前一个更复杂一些，但出于某些原因，它更好一些。具体一点说，它的内部表示形式被隐藏了（保护属性，用#表示），外部客户不能访问。如果修改它的表示形式，需要重新编译一些代码，但是从语义上说，外部客户不会受到这种修改的影响（换言之，原有的代码不会被破坏）。

同时，通过显式地声明客户可以对这个类的对象进行的某些操作（责任），我们记录了关于问题空间的某些决定。具体来说，我们让所有的客户（公有的，用+表示）有权取得雇员的 name（姓名）、social security number（社会保险号）、department（部门）。本章稍后将讨论可见性（即公有、保护、私有和包可见性）。

3.1.3 行为

没有对象是孤立存在的。对象与对象之间会相互操作。因此，我们可以这样说：

“行为是对象在状态改变和消息传递方面的动作和反应的方式。”

换言之，对象的行为代表了它外部可见的活动。

一个操作是某种动作，一个对象对另一个对象执行这个操作，目的是获得反应。例如，客户可能调用 append 和 pop 操作，分别使一个队列对象增长或缩减。客户也可能调用 length 操作，它返回一个值，表示队列对象的大小，但不会改变队列本身的状态。

在 Java 中，客户可以在一个对象上执行的操作通常被声明为方法。像 C++这样的语言来自于更过程化的早期语言，在这些语言中，我们说一个对象调用另一个对象的成员函数。在 Smalltalk 这样的纯面向对象的语言中，我们说一个对象向另一个对象传递一个消息。一般来说，一个消息就是一个对象执行了另一个对象的操作，虽然底层的分发机制是不一样的。方便起见，我们互换使用“操作”和“消息”这两个术语。

消息传递只是定义对象行为的一个方面，我们对行为的定义还指出，对象的行为会受到其状态的影响。考虑自动售货机的例子。我们可以调用某个操作选择商品，但自动售货的行为取决于它的状态。如果没有塞进足够购买所选商品的钱，自动售货机可能什么都不会做。如果我们提供了足够的零钱，自动售货机会收下零钱，然后给出我们选择的商品（然后改变它的状态）。因此可以说，一个对象的行为是它的状态以及施加在它上面的操作的函数。这个

副作用的概念让我们改进了状态的定义：

“一个对象的状态代表了它的行为的累积效果。”

大部分有意思的对象没有静态状态，它们的状态包含了一些属性，这些属性的值在对象活动时被修改并被查询。一个对象的行为包括了其操作的总和。接下来将讨论操作、它们与对象责任的关系以及它们怎样让对象实现其责任。

1. 操作

一个操作代表了一个类提供给它的对象的一种服务。在实践中，我们发现一个客户通常执行一个对象的5种操作[1]。其中三种最常见的操作如下：

- 修改操作　更改一个对象的状态的操作；
- 选择操作　访问一个对象的状态但并不更改这个状态的操作；
- 遍历操作　以一种定义良好的方式访问一个对象的所有部分的操作。

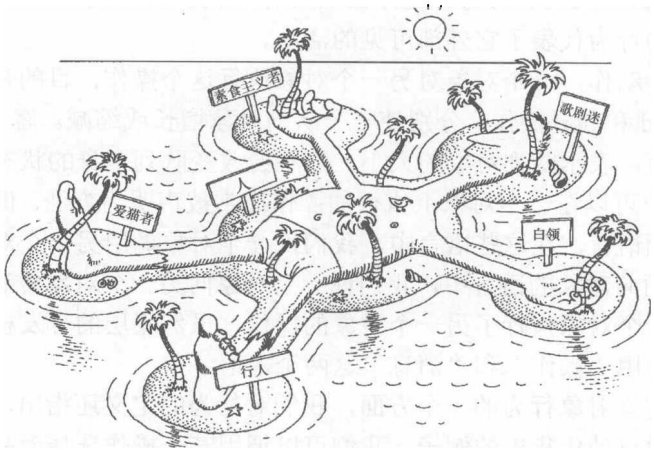
另外两种操作是公共的，它们体现了创建和销毁一个类的实例的需要：

- 构造操作　创建一个对象并初始化它的状态的操作；
- 析构操作　释放一个对象的状态并销毁对象本身的操作。

在C++中，构造操作和析构操作是作为类定义的一部分被定义的；但在Java中，只有构造操作，没有析构操作。在Smalltalk中，这样的操作通常是元类（即类的类）协议的一部分。

2. 角色和责任

一个对象的所有方法共同构成了它的协议。因此，一个对象的协议定义了对象允许的行为的封装，构成了这个对象完整的静态视图和动态视图。对于大多数有用的抽象来说，将这个较大的协议分成逻辑上的行为分组是有意义的。这些分组划分了对象的行为空间，表明了一个对象可以扮演的角色。角色是一个对象戴上的一个面具[8]，它定义了一种抽象与它的客户之间的契约。



对象可以扮演许多不同的角色

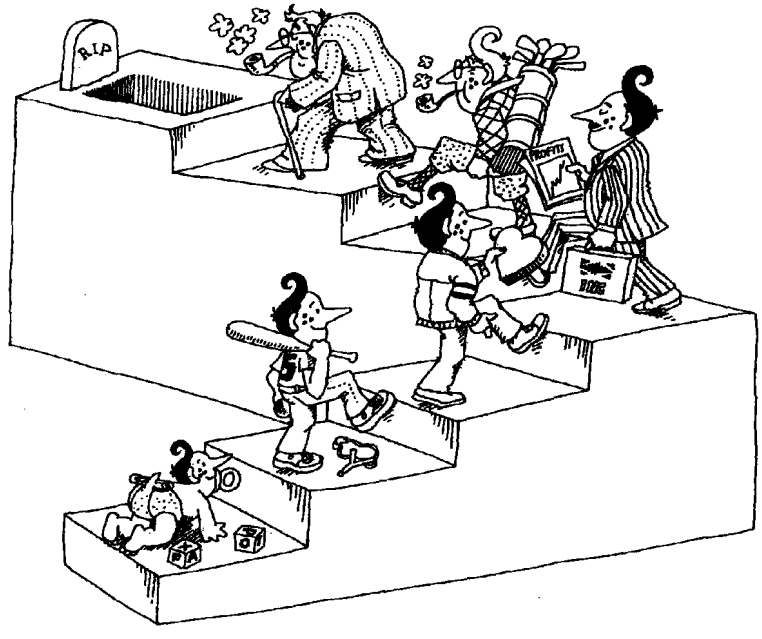
[1] Lippman建议了一种稍有不同的分类方式：管理函数、实现函数、辅助函数（所有的修改操作）和访问函数（与选择操作一样）[7]。

“责任意味着表达对象的一种目标以及它在系统中的位置。一个对象的责任是它为支持的所有契约提供的全部服务。”[9]换言之，可以说一个对象的状态和行为共同决定了这个对象可以扮演的角色，这又实现了这种抽象的责任。

实际上，大多数有意思的对象在它们的生命周期中扮演许多不同的角色。考虑下面的例子[10]。

- 一个银行账户可能扮演一个现金资产的角色，账户的所有者可以向它存钱或者从它取钱。但是，对于税务检查来说，这个账户可能扮演这样一个角色，即每年都必须报告它的股息。
- 对于一个交易者，一股股票代表了一个有价值的实体，可以买进或卖出。对于律师，同样的股票代表了一种法律手段，它包含了某种权利。
- 在一天中，同一个人可能扮演母亲、医生、园丁和电影评论员的角色。

对象扮演的角色是动态的，同时又是互斥的。在股票的例子中，它的角色有些重叠，但每个角色都固定地与和这股股票打交道的客户有关。在人的例子中，她的角色非常动态，不时地在转换。



对象在它们的生命周期中扮演着许多不同的角色

后面的章节中将讨论到，我们常常从检查对象扮演的不同角色开始分析问题。在设计时，我们精化这些角色，设计出特定的操作，实现每个角色的责任。

3. 对象像自动机

对象中存在状态，这意味着操作调用的次序非常重要。从而，导致了这样一种思想，即

每个对象就像一个微小的、独立的自动机[11]。实际上，对于某些对象来说，这种事件和操作的时间顺序非常普遍，所以我们可以很好地将这种对象的行为总结为一个等价的有限状态自动机。第5章中将展示一种层次结构的有限状态自动机的表示法，我们可以利用自动机来表达这些语义。

继续自动机的比喻，我们可以区分对象是主动的还是被动的。主动的对象有自己的控制线程，而被动的对象则没有。主动的对象通常是自动的，这意味着它们不需要由其他对象操作，就能表现出一些行为。而对于被动对象来说，只有当显式地操作它时，才会发生状态变化。从这个角度来看，我们系统中的主动对象是控制的中心。如果系统包含多个控制线程，通常会有多个主动对象。串行式的系统，通常只有一个主动对象，例如，有一个主要的对象负责管理事件循环，分发消息。在这样的架构中，所有其他的对象都是被动的，它们的行为最终是由那个主动对象发出的消息触发的。在其他类型的串行式系统架构中（如事务处理系统），没有明显的核心主动对象，所以控制一般分散在系统的被动对象中。

3.1.4 标识符

Khoshafina 和 Copeland 提出了这样的标识符定义："标识符是一个对象的属性，它区分这个对象与其他所有对象。"[12]

他们继续注解道，"大多数程序设计语言和数据库语言使用变量名称来区分临时对象，混淆了定址能力和标识符。大多数数据库系统使用标识符主键来区分持久对象，混淆了数据值和标识符。"不能够区分对象的名称和对象本身，这导致了面向对象编程中的许多错误。

示例 3-2 展示了维护创建的对象的标识符的重要性，并展示了标识符非常容易被丧失而无法恢复。

示例 3-2

考虑一个代表一个显示项的类。一个显示项是所有以 GUI 为中心的系统中的一种共同抽象：它是所有在某个窗口中显示的对象的基类，所以记录了所有这类对象公有的结构和行为。客户希望能够画出、选择并移动显示项，并查询它们被选择的位置和状态。每个显示项都有一个位置，由坐标 x 和 y 标出。

假定我们实例化了一些 DisplayItem 类，如图 3-4a 所示。具体来说，我们实例化这些类的操作在内存中占据了 4 个位置，它们的名称分别是 item1、item2、item3 和 item4。这里，item1 是一个独立的 DisplayItem 对象的名称，但其他 3 个名称是指向 DisplayItem 对象的指针。只有 item2 和 item3 真正指向了独立的 DisplayItem 对象（因为在它们的声明中，我们分配了新的 DisplayItem 对象），item4 有意地没有指向任何对象。而且，item2 和 item3 所指向的对象是匿名的，我们只能间接地通过它们的指针访问这些对象。

每个对象的唯一标识符（不一定是名称）是在对象的整个生命周期中都被保持的，即使它的状态改变时也是如此。这有点像禅宗的河流问题：即使河中流淌的水已经不一样了，明天的河流还是今天的河流吗？例如，让我们移动 item1。我们可以访问 item2 所指的对象，取

得它的位置，然后将 item1 移到同样的位置。

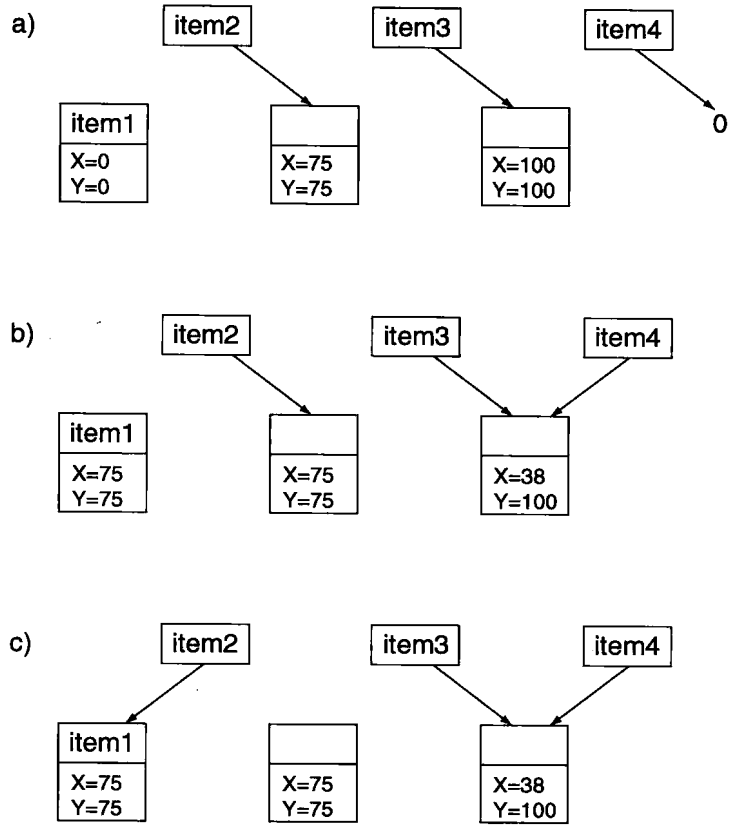


图 3-4 对象标识符

另外，如果让 item4 等于 item3，就可以利用 item4 来引用 item3 所指向的对象。利用 item4，可以将这个对象移动到一个新的位置，例如 x=38，y=100。图 3-4b 展示了这些结果。这里我们看到，item1 和 item2 所指的对象具有同样的位置状态，item4 和 item3 现在指向了同一个对象。请注意，我们说“item2 所指的对象”，而不说“对象 item2”。前一种方式更准确，虽然有时候我们互换这两种说法。

虽然 item1 和 item2 所指的对象具有相同的状态，但它们代表不同的对象。另外请注意，我们通过新的间接名称 item4 进行操作，改变了 item3 所指的对象的状态。这就是所谓的“结构共享”，意思是对某个对象可以用多种方式命名。换言之，一个对象有许多别名。结构共享是面向对象编程中许多问题的根源。如果没有意识到通过别名来操作一个对象的副作用，常常会导致内存泄漏、非法内存访问甚至更糟糕的、未预期的状态改变。例如，如果我们销毁了 item3 所指的对象，那么 item4 的指针值就没有意义了，这种情况称之为“悬空的引用”（dangling reference）。

考虑图 3-4c，它展示了修改 item2 指针的值，令它指向 item1 的结果。现在 item2 指向了

item1 对象。不幸的是，我们引入了内存泄漏：原来 item2 所指的对象不再有名称了，既没有直接名称，也没有间接名称，所以它的标识符被遗失了。在 Smalltalk 和 Java 这样的语言中，这样的对象会被垃圾收集，它们的空间会被自动回收；但在 C++这样的语言中，它们的存储空间不会被释放，直到创建它们的程序终止运行为止。尤其对于长时间运行的程序来说，这样的内存泄漏要么引起麻烦，要么带来灾难[1]。

3.2 对象之间的关系

一个对象本身是非常无趣的。对象通过与其他对象协作，为系统的行为做出贡献。“我们没有一个钻头研磨机（bit-grinding processor）来掠夺和打劫数据结构，但我们有许多行为良好的对象，它们有礼貌地互相询问，实现它们不同的愿望。”[13]例如，考虑一个飞机的对象结构，它被定义为“一组部件，它们都有落向地面的自然趋势，需要不断努力和管理才能力争取得成果”[14]。只有所有组成对象共同努力，飞机才能够飞行。

两个对象之间的关系包括了一个对另一个所做的假定，即包括可以执行哪些操作以及将导致怎样的行为。我们发现，在面向对象分析和设计中有两种对象关系特别有趣，它们是：

- 链接；
- 聚合。

3.2.1 链接

链接（link）这个术语来自 Rumbaugh 等人，他们将它定义为“两个对象之间物理上或概念上的联系”[16]。一个对象通过它与其他对象的链接，与其他对象进行协作。换言之，链接代表了具体的关联，通过这种关联，一个对象（客户）请求另一个对象（服务提供者）的服务，或者通过这种关联从一个对象导航到另一个对象。

图 3-5 展示了几种不同的链接。在这个图中，两个对象图标之间的连线代表这两个对象之间存在链接，意味着消息可以通过这个途径传递。消息用小箭头表示，它代表消息的方向，带有一个标签为消息本身命名。例如，图 3-5 中展示了一个简化的水流控制系统的一部分，这可能是在一个制造工厂中控制管道中水流的。可以看到，`FlowController` 对象有一个到 `Valve` 对象的链接。`Valve` 对象有一个 `DisplayPanel` 对象的链接，`DisplayPanel` 将显示它的状态。只有通过这些链接，一个对象才能向另一个对象发送消息。

在两个对象之间发送消息通常是单向的，虽然偶尔也会出现双向的情况。在我们的例子中，`FlowController` 对象调用了 `Valve` 对象上的操作（目的是改变它的设置）和

[1] 考虑控制卫星或心脏起搏器的软件发生内存泄漏的后果：重新启动离地球数百万公里远的卫星上的计算机是很不方便的；类似地，在心脏起搏器软件中，不可预测的自动垃圾收集有可能是致命的。出于这些原因，实时系统的开发者们通常避免在堆中不受限制的对象分配。

DisplayPanel 对象上的操作（目的是改变它显示的东西）。这种分离关注在结构良好的面向对象系统中是很常见的。另外请注意，虽然消息传递是由客户发起的（如 FlowController），指向服务提供者（如 Valve 对象），但是数据可以通过链接双向流动。例如，当 FlowController 调用 Valve 对象上的 adjust 操作时，数据（即要改变的设置）从客户流向了服务提供者。然后，如果 FlowController 调用了 Valve 对象上的另一个操作 isClosed，结果（即阀门是否处于完全关闭的位置）将从服务提供者被传回到客户。

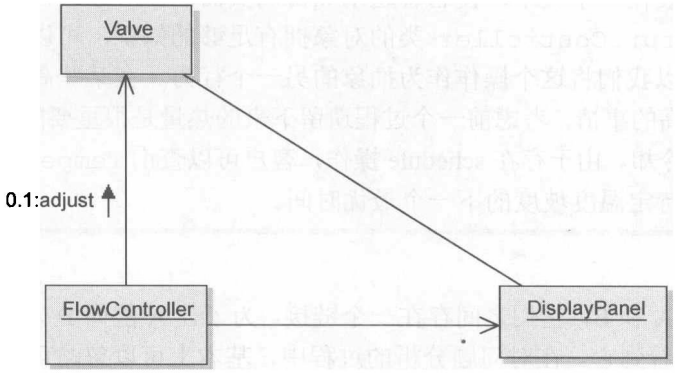


图 3-5 链接

作为链接的参与者，一个对象可能扮演以下三种角色之一：

- 控制器 这个对象可以操作其他对象，但不会被其他对象操作。在某些地方，“主动对象”和“控制器”这两个术语是互换使用的。
- 服务器 这个对象不操作其他对象，它只被其他对象操作。
- 代理 这个对象既可以操作其他对象，也可以被其他对象操作。创建代理通常是为了表示问题领域中的一个真实对象。

在图 3-5 所示的例子中，FlowController 是一个控制器对象，DisplayPanel 是一个服务器对象，Valve 是一个代理。示例 3-3 展示了这些责任如何恰当地被分配在一组协作的对象中。

示例 3–3

在许多不同类型的工业过程中，某些反应要求一定的温度坡度，即提升某种物质的温度，让它在某个温度上保持一段时间，然后将它冷却到环境温度。不同的过程需要不同的温度曲线：某些对象（如望远镜中的镜子）必须慢慢冷却，而另一些物质（如钢）必须迅速冷却。这种温度坡度具有足够的定义良好的行为，可以创建为一个类。因此我们提供了 TempratureRamp 类，它在概念上是一种时间/温度的映射关系（参见图 3-6）。

实际上，这种抽象的行为不只是简单的时间/温度映射关系。例如，可以设置一种温度坡度，要求在 60 分钟时达到 250°F（进入温度坡度 1 小时后），在 180 分钟时达到 150°F（进入温度坡度 3 小时后），然后我们想知道在 120 分钟时温度是多少。这需要线性插值，这就是我

们希望的这种抽象的另一个行为（即插值）。

有一个行为是显然不需要这种抽象具备的，即控制加热器来实现特定的温度坡度。我们喜欢更好的分离关注，让这种行为通过3个对象的协作来实现：一个温度坡度实例、一个加热器和一个温度控制器（参见图3-6）。process操作提供了这种抽象的主要行为，它的目标是利用指定位置的加热器实现指定的温度坡度。

对于我们的风格，有一点说明。第一眼看上去，似乎我们设计了一种抽象，其目的只是将一种功能分解包装在一个类中，使它看起来显得高贵而面向对象。但schedule操作表明并非如此。TemperatureController类的对象拥有足够的知识，可以决定何时应该安排哪一种具体控制，所以我们将这个操作作为抽象的另一个行为。在某些高能耗的工业过程中，加热物质是成本很高的事情，考虑前一个过程遗留下来的热量是很重要的，另外也要考虑未参与的加热器的正常冷却。由于存在schedule操作，客户可以查询TemperatureController对象，以确定处理特定温度坡度的下一个最优时间。

---

1. 可见性

考虑两个对象A和B，它们之间存在一个链接。为了让A能向B发送一条消息，B必须以某种方式让A能看到它。在对问题分析的过程中，基本上可以忽略可见性的问题，但当开始设计具体的实现时，就必须考虑跨越链接的可见性，因为，我们此时的考虑决定了链接两端对象的范围的访问。本章稍后将会详细讨论这一点。

2. 同步

当一个对象通过链接向另一个对象发送一条消息时，这两个对象就称为同步了。在完全串行式的应用中，这种同步通常是通过简单的方法调用来完成的。但是，在存在多控制线程的情况下，对象需要更复杂的消息传递机制来处理并发系统中可能出现的互斥问题。前面曾提到，主动对象拥有自己的控制线程，所以我们期望它们的语义在其他对象存在时仍然得到保证。但是，当一个主动对象与一个被动对象之间有链接时，我们必须选择以下3种同步方式之一：

- 顺序　只有在某一时刻只存在一个主动对象时，被动对象的语义才得到保证；
- 守卫　在多个控制线程的程序下，被动对象的语义也能保证，但主动的客户之间必须协作，以实现互斥访问；
- 并发　在多个控制线程的程序下，被动对象的语义也能保证，服务提供者保证互拆。

3.2.2 聚合

链接表明了一种端到端的关系或客户/服务提供者的关系，而聚合则表明了一种整体/部分层次结构，提供了从整体（也被称为聚合体）导航到它的部分的能力。例如，如图3-6所示，TemperatureController对象拥有到TemperatureRamp对象的链接，也有到Heater对象的链接。因此TemperatureController对象是整体，Heater是它的部分。聚合关系的表示方法将在第5章中进一步讨论。

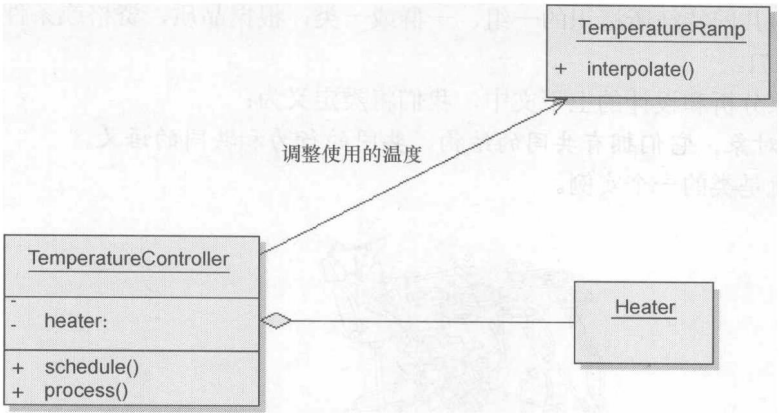


图 3-6 聚合

如果一个对象是另一个对象的一部分，就意味着它到它的聚合体有一个链接。通过这个链接，聚合体可以向它的部分发送消息。对于 `TemperatureController` 对象，可以找到它对应的 Heater。对于 Heater 这样的对象，当且仅当 Heater 的状态包括它的包装对象（也被称为它的容器）的知识时，才有可能导航到它的包装对象。

聚合可以代表物理上的包含，也可以不代表。例如，一架飞机由机翼、引擎、起落架等组成，这是物理上包含的例子。与此不同，股票持有人及其持有股票之间的关系则是不需要物理上包含的聚合关系。股票持有人拥有股票，但这些股票不是股票持有人的物理组成部分。这种整体/部分的关系更多的是概念上的，因此不太直接，不像构成飞机的各部分那样的物理聚合。

显然，在链接和聚合之间需要折中。有时候聚合更好，因为它将各个部分封装为整体的秘密；有时候链接更好，因为它们允许对象之间较松的耦合。明智的工程决定需要仔细权衡这两方面的因素。

3.3 类的本质

类的概念和对象的概念是紧密交织在一起的，因为我们在谈论一个对象时不得不提到它的类。但是，这两个术语之间存在着重要的差别。

3.3.1 什么是类，什么不是类

对象是存在于时间和空间中的具体实体，而类仅代表一种抽象，即一个对象的“本质”。因此，可以说 Mammal 类，它代表了所有哺乳动物的共同特征。要确定这个类中的某个具体的哺乳动物，则必须说“这个哺乳动物”或“那个哺乳动物”。

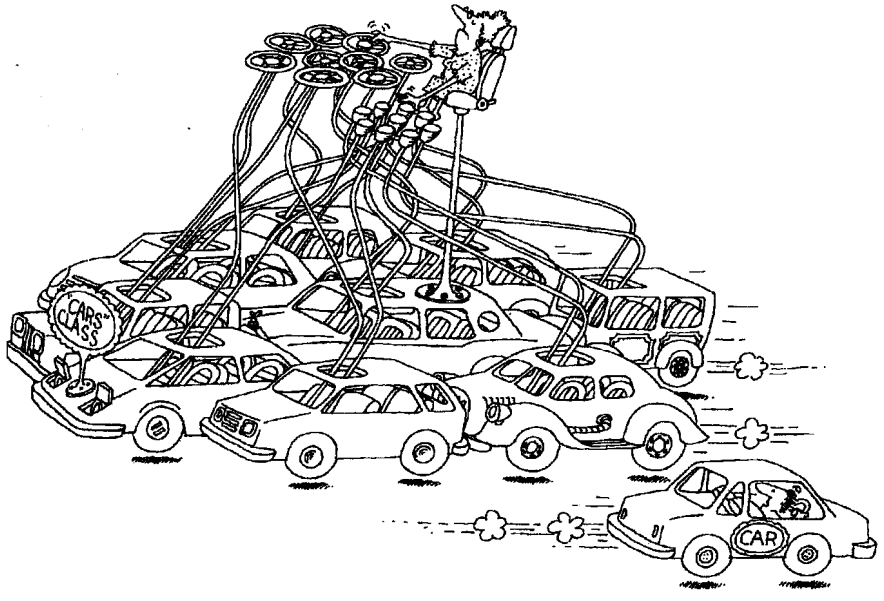
在日常用语中，Webster 的 *Third New International Dictionary* 将“class”定义为“由一些

共同特征或一项共同特征所标识的一组、一群或一类，根据品质、资格或条件进行的分组、区分或分级”[17]。

在面向对象分析和设计的上下文中，我们将类定义为：

“类是一组对象，它们拥有共同的结构、共同的行为和共同的语义。”

一个对象就是类的一个实例。



类是一组对象，它们拥有共同的结构和行为

什么不是一个类？一个对象不是一个类。没有共同结构和行为的对象不能够被划分为一类，因为根据定义，它们除了都是对象之外，没有别的共同点。

值得一提的是，类（正如大多数编程语言中所定义的）对于分解是必要的，但不是充分的。某些抽象非常复杂，所以不能够利用单个类定义的方式很方便地表达。例如，一种相当高层的抽象、一个 GUI 框架、一个数据库和整个库存系统在概念上都是独立的对象，但它们都不能被表示为一个单独的类[1]。相反，最好是将这些抽象表示为一组类，这些类的实例互相协作，提供我们所期望的结构和功能。Stroustrup 将这样的一组类称为一个组件[18]。

3.3.2 接口和实现

Meyer[19]和 Synder[20]都指出，编程在很大程度上是一种“制定契约”：一个较大问题的不同功能通过子契约被分配给不同的设计元素，被分解成较小的问题。没有别的情况比在

[1] 人们可能很想将这样的抽象用单个类来表达，但复用和修改的粒度太粗了。拥有一个“胖”接口是一种不好的实践，因为大多数客户只需要引用服务提供的一个小子集。而且，改变一个巨大接口的一部分就会影响到所有的客户，即使他们不关心改变的部分。嵌套类不会消除这些问题，它只是延缓了问题。

设计类时更能体现这种思想了。

一个单独的对象是一个具体实体，在整体系统中扮演某个角色，而类则记录了所有相关对象的共同结构和行为。因此，类起到的作用是在一种抽象和所有它的客户之间建立起协议。通过在类的接口中记录下这些决定，一种强类型的编程语言可以在编译时检查是否违反了这一协议。

这种编程即契约的观点可以区分一个类的外部视图和内部视图。一个类的接口提供了它的外部视图，因此强调了抽象，隐藏了它的结构和行为的秘密。这个接口主要由所有的操作声明构成，这些操作适用于这个类的所有对象，但它也可能包括其他类、常量、变量和异常的声明，因为这种抽象可能需要这些东西。与接口不同，类的实现是它的内部视图，它包含类行为的秘密。一个类的实现主要由类接口中定义的所有操作的实现构成。

可以进一步将类的接口分成以下 4 个部分：

- 公有　所有客户都可以访问的声明；
- 保护　只能由该类本身及其子类访问的声明；
- 私有　只能由该类本身访问的声明；
- 包　只能由同一个包中的类访问的声明。

这些类型的可见性的详细语义，可能根据实现的语言而有所不同。

可见性和友元

不同的编程语言提供了不同的公有、保护、私有和包可见性的组合，开发者可以进行选择，为类接口的每一部分建立具体的访问权限，从而控制客户可以看见什么，不能看见什么（即可见性）。

具体来说，C++允许开发者显式地声明这 4 种不同的可见性[1]。C++的友元机制允许一个类区分一些具有特权的类，让这些特权类能够看到这个类的保护部分和私有部分。友元打破了类的封装，所以在实际设计中必须慎重选择。Java 没有友元，但它具有某种类似的可见性类型，称为包可见性，在同一个包中的所有类都可以互相访问。除了友元之外，公有、保护和私有可见性在 Java 和 C++中是一样的。Ada 允许公有和私有可见性声明，但不支持保护可见性。在 Smalltalk 中，所有实例变量都是私有的，所有方法都是公有的。在 Object Pascal 中，字段和操作都是公有的，所以是未被封装的。

构成类的表示形式的常量和变量有各种叫法，这取决于我们使用的具体语言。例如，Smalltalk 使用的是术语“实例变量”（instance variable），Object Pascal 和 Java 使用的是术语“字段”（field），C++使用的是术语“数据成员”（data member）。我们将互换使用这些术语来表示类的这些部分，代表类实例的状态。

[1] C++的 struct 是一种特殊情况。struct 是这样一种类，它的所有元素都是公有的。

对象的状态必须在它对应的类中有某种表示形式，所以通常表示为常量或变量声明，作为类接口的保护或私有部分。通过这种方式，一个类的所有实例的共同表示形式被封装起来，对这种封装的修改不会在功能上影响任何外部客户。

3.3.3 类的生命周期

只要理解了一个简单类的公有操作的语义，就可以懂得这个类的行为。但对于某些更有趣的类来说（如移动 DisplayItem 类的一个实例，或者设置 TemperatureController 类的一个实例），就要涉及每个实例生命周期的不同阶段的不同操作。本章前面曾提到，这些类的实例就像一个小的自动机。由于这些实例具有共同的行为，所以可以通过类来描述这些共同的事件次序和时间次序语义。第 5 章中将讨论，我们可以利用有限状态自动机来描述某些有趣的类的这种动态行为。

3.4 类之间的关系

考虑这些对象的类之间的相似和不同——花、雏菊、红玫瑰、黄玫瑰、花瓣和瓢虫，我们可以得到下面的结论：

- 雏菊是一种花；
- 玫瑰是（另）一种花；
- 红玫瑰和黄玫瑰都是一种玫瑰；
- 花瓣是两种花的组成部分；
- 瓢虫会吃掉蚜虫等害虫，这些害虫会侵扰某些种类的花。

从这个简单的例子中我们可以得出结论，类像对象一样，也不是孤立存在的。对于一个特定的问题域，一些关键的抽象通常与各种有趣的方式联系在一起，形成了我们设计的类结构[21]。

出于某些原因，我们在两个类之间建立起关系。首先，一种类关系可能表明某种类型的共享。例如，雏菊和玫瑰都是花，这意味着它们都有色彩鲜艳的花瓣，散发出芳香。其次，一种类关系可能表明某种语义上的联系。因此，我们说红玫瑰和黄玫瑰的相似度要大于雏菊和玫瑰，雏菊和玫瑰的关系比花瓣和花的关系更密切。类似地，瓢虫和花之间存在一种共生关系：瓢虫保护花免遭害虫侵袭，花为瓢虫提供了食物来源。

总的来说，存在三种基本类型的类关系[22]。第一种关系是一般/特殊关系，表示“是一种”关系。例如，玫瑰是一种花，这意味着玫瑰是一种特殊的子类，而花是更一般的类。第二种关系是整体/部分，表示“组成部分”关系。因此，花瓣不是一种花，它是花的一个部分。第三种关系是关联，表示某种语义上的依赖关系，如果没有这层关系，这些类就毫无关系了，如瓢虫和花之间的关系。再如，玫瑰和蜡烛基本上是独立的类，但它们都是可以用来装饰餐桌的东西。

3.4.1 关联

在这些不同类型的类关系中，关联是最常见的，也是语义上最弱的。确定类之间的关联通常是分析和早期设计的活动。随着继续设计和实现，我们常常会细化这些较弱的关联，将它们变成某种更具体的类关系。

1. 语义上的依赖关系

如示例 3-4 所示，关联只代表一种语义上的依赖关系，它不表示这种依赖关系的方向（如果没有特别说明，关联意味着双向导航，如我们的例子所示），也不表示一个类与另一个类相关的具体形式（我们只能通过命名每个类在关系中扮演的角色来暗示这些语义）。但是，这些语义在分析问题时已经足够了，这时只需要确定这样的依赖关系。通过创建关联，用语义关系、它们的角色和它们的基数记录了参与关联的类。

示例 3-4

对于一种交通工具来说，两个关键抽象是交通工具和轮子。如图 3-7 所示，可以在这两个类之间显示一种简单的关联：Wheel 类和 Vehicle 类。（也许聚合关系更好。）这个关联隐含表明一种双向导航。对于一个 Wheel 的实例，应该能够定位代表它的 Vehicle 的对象；对于一个 Vehicle 的实例，应该能够定位所有的轮子。

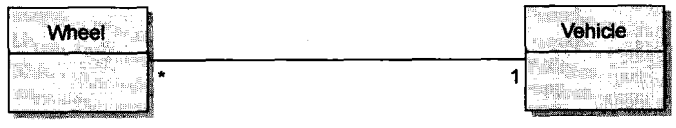


图 3-7 关联

这里展示了一对多的关联：每个 Wheel 实例与一个 Vehicle 有关，每个 Vehicle 实例可能与多个 Wheel 有关（用*号表示）。

---

2. 多重性

我们的例子引入了一种一对多的关联，这意味着对于每个 Vehicle 的实例，可能有 0 个（例如，船是一种交通工具，但没有轮子）或多个 Wheel 类的实例，对于每个 Wheel，只有一个 Vehicle。这说明了一种关联的多重性。在实践中，关联有以下三种常见的多重性：

- 一对一；
- 一对多；
- 多对多。

一对一的关系代表了一种非常有局限的关系。例如，在零售电话营销操作中，我们会发现 Sale 类和 CreditCardTransaction 类之间存在一对一的关系。每次销售都对应着一次信用卡事务，每次信用卡事务都对应一次销售。多对多的关系也很常见。例如，Customer 类的每个实例都可能与几个 SalesPerson 类的实例发起一次交易，每个销售人员都可能与许多不同的客户打交道。第 5 章将进一步讨论，这三种常见的多重性存在的一些变化情况。

3.4.2 继承

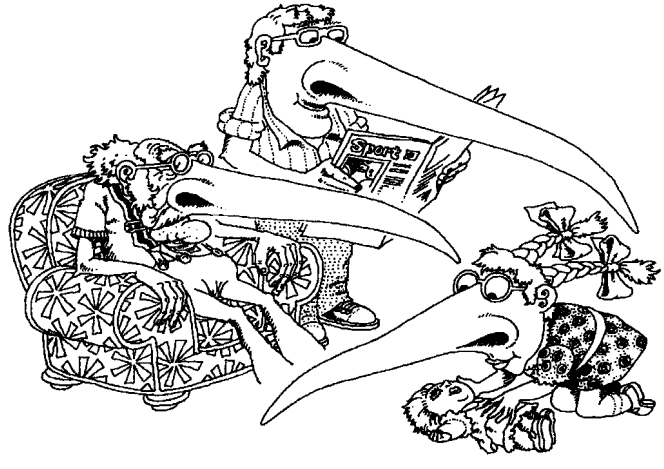
在这些具体的关系之中，继承也许是语义上最有趣的，它代表了一般/特殊关系。但是根据我们的经验，给定问题域的关键抽象之间存在着丰富的关系，继承对于表示所有这些关系是不够的。继承的一种替代方法是一种所谓“委托”的语言机制，通过这种机制，对象将它们的行为委托给一些相关的对象。

示例 3–5

在空间探测器被发射之后，它们向地面站发回重要子系统状态（如电力和推进子系统）以及不同传感器（如辐射传感器、质谱仪、摄像头、小陨石碰撞检测器等）的信息。这些被传回的信息被称为遥测数据。遥测数据一般以位流的方式被传输，它有一个头部，其中包含时间戳和信息类型的标识符，然后是几帧来自不同子系统和传感器的、经过处理的数据。这似乎就是几种不同数据的聚合。

这些重要的数据需要封装，否则没有办法防止客户修改像 timestamp（时间戳）或 currentPower（当前电能）这样的重要数据的值。类似地，这些数据的表示方式是已知的，所以，如果要改变这种表示方式（如加入一些新元素或改变原有的位对齐方式），所有客户都会受到影响。至少，我们需要重新编译所有对这个结构的引用。更重要的是，这样的修改可能会打破客户对这种表示形式所做的假定，导致程序逻辑受到破坏。

最后，假定我们的系统分析表明需要几百种不同类型的遥测数据，既包括前面提到的电子数据，也包括系统各处不同测试点的电压读数。我们会发现，声明这些额外的结构导致了大量的冗余，既重复了结构，也重复了共同的功能。



子类可以继承其超类的结构和行为

一种较好的做法是为每种遥测数据声明一个类。通过这种方式，可以将每个类的表示形式隐藏起来，将它的行为与它的数据关联起来。但是，这种方式仍然不能解决冗余的问题。

因此，更好的做法是构建一个类层次结构来反映我们的决策，在这个层次结构中，特殊化的类从一般化的类中继承结构和行为，如图 3-8 所示。

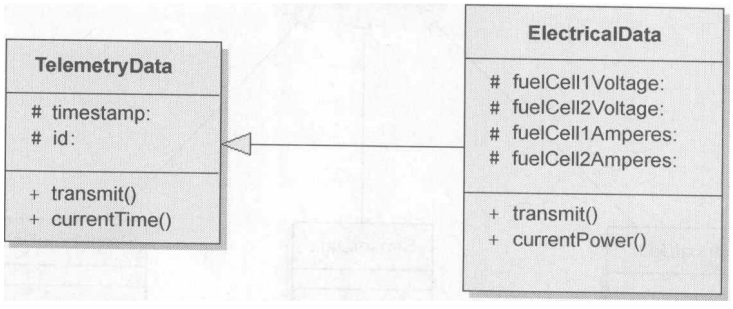


图 3-8 ElectricalData（电子数据）从超类 TelemetryData（遥测数据）中继承

对于 ElectricalData 类来说，它继承了 TelemetryData 类的结构和行为，但加上了它的结构（额外的电压数据），重新定义了它的行为（transmit 函数）来传输额外的数据，甚至可以添加它的行为（currentPower 函数，提供当前电能水平）。

1. 单继承

简单来说，继承是类之间的一种关系，在这种关系中，一个类共享了另一个类（单继承）或多个类（多继承）中定义的结构和行为。我们把提供给其他类继承的类称为“超类”。在示例 3-5 中，TelemetryData 是 ElectricalData 的超类。类似地，我们把从其他类继承的类称为“子类”。ElectricalData 是 TelemetryData 的子类。所以，继承在类之间定义了“是一种”关系，在这种关系中，子类从一个或多个超类中继承。这实际上是继承的判别测试。给定类 A 和类 B，如果 A 不是一种 B，那么 A 就不应该是 B 的子类。从这个意义上说，ElectricalData 是一种特殊类型的 TelemetryData。语言是否支持这种继承，决定了它是基于对象的语言还是面向对象的语言。

子类通常扩展或限制了超类中原有的结构和行为。扩展超类的子类被称为扩展继承。例如，子类 GuardedQueue 可能提供额外的操作，使这个类在多控制线程的情况下变得安全，从而扩展了超类 Queue 的行为。与此相对的是，限制超类的子类被称为限制继承。例如，子类 UnselectableDisplayItem 可能限制了超类 DisplayItem 的行为，禁止客户从视图中选择它的实例。在实践中，子类扩展了超类还是限制了超类并不总是很清楚。实际上，子类常常同时做这两件事。

图 3-9 展示了从超类 TelemetryData 中导出的单继承关系，每条有向线段代表一个“是一种”关系。例如，CameraData“是一种”SensorData，SensorData 又“是一种”TelemetryData。

这与语义网中的层次结构是等价的，语义网是认知科学和人工智能的研究者组织关于世界的知识的一种工具[25]。实际上，正如将在第 4 章中讨论的，在不同抽象之间设计一种合适的继承层次结构基本上是一种明智的分类工作。

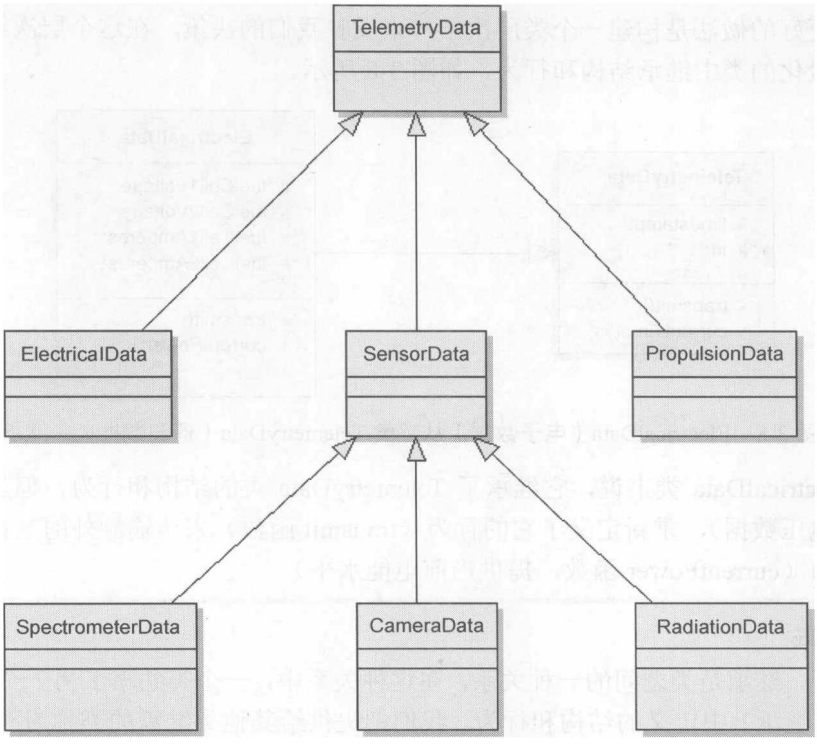


图 3-9 单继承

我们预计图 3-9 中的某些类会有实例，而另一些类会没有实例。例如，我们预计每一种最特殊的类（也被称为“叶子类”或“具体类”）都会有一些实例，如 ElectricalData 和 SpectrometerData。但是对于中间的、更一般的类来说，可能没有任何实例，如 SensorData 或 TelemetryData。没有实例的类被称为“抽象类”。抽象类在编写时就预期它的子类会添加结构和行为，通常是完成它未完成的方法实现。

在继承和封装之间存在着非常真实的压力。从大的方面来讲，利用继承暴露了被继承类的一些秘密。具体来说，这意味着要理解某个类的含义，就必须常常研究它的所有超类，有时还要研究超类的内部视图。

继承意味着子类继承了超类的结构。因此，在示例 3-5 中，ElectricalData 类的实例包含了超类的数据成员（如 id 和 timestamp），以及特殊类的数据成员（如 fuelCell1Voltage、fuelCell2Voltage、fuelCell1Amperes 和 fuelCell2Amperes）。

子类也从超类中继承行为。因此，ElectricalData 类的实例可以执行 currentTime 操作（从它的超类继承）、currentPower 操作（该类自己定义）及 transmit 操作（子类重新定义）。

2. 多态

对于 TelemetryData 类来说，成员函数 transmit 可能传送遥测数据流的标识符和它的时间戳。但是 ElectricalData 类的同一个函数可能调用 TelemetryData 的 transmit 函数并传送它的电

压和当前的值。

这种行为是由多态引起的。在泛化中，这样的操作被称为“多态的”。多态是类型理论中的一个概念，即一个名字可能代表许多不同类的实例，只要它们都有共同的超类。于是，由这个名字所代表的对象就能够以不同的方式对同一组操作做出反应。利用多态，一个操作可以被层次结构中的类以不同的方式实现。通过这种方式，子类可以扩展超类的能力，或者覆写父类的操作，就像示例 3-5 中 ElectricalData 所做的那样。

多态的概念首先是由 Strachey 提出的[29]，他谈到了一种初级的多态，指出像+这样的操作符可以被定义成含义不同的东西。我们将这个概念称为“重载”。在 C++中，开发者可以用同样的名字来定义函数，只要它们的调用可以通过函数签名来区别。函数签名由参数的个数和类型构成（C++与 Ada 不一样，在重载解析时是不考虑函数的返回值类型的）。与此不同的是，Java 不允许重载操作符。Strachey 也提到了参数多态，今天我们就称之为多态。

没有多态，开发者写的代码就必须包含大量的 case 或 switch 语句[1]。没有多态，我们就不能为各种遥测数据创建一个类层次结构，而不得不定义一个一体化的可变记录来包含所有与这类数据相关的属性。为了区分每一种变种，必须检查与该记录关联的标记。

要加入另一种遥测数据，必须修改这个可变记录，将它加到操作这条记录的每一个 case 语句中。这很容易出错，也增加了设计的不稳定性。

有了继承，就不需要一体化的记录，因为可以区分不同类型的抽象。Kanplan 和 Johnson 指出：“如果许多类使用相同的协议，多态就最有用”[30]。利用多态，我们就不需要大型的 case 语句，因为每个对象都知道自己的类型。

没有多态的继承是可能的，但它肯定用处不大。

多态和延迟绑定是分不开的。在出现多态时，方法和名字的绑定要在执行时确定。在 C++中，开发者可以控制成员函数使用早期绑定或迟后绑定。具体来说，如果方法被声明是 virtual 的，那就使用延迟绑定，这个函数就被认为是多态的；如果没有 virtual 声明，那么这个函数就使用早期绑定，可以在编译时解析。Java 不需要显式的 virtual 声明就执行迟后绑定。补充材料“调用一个方法”描述了一种实现是怎样选择某一个执行方法的。

调用一个方法

在传统的编程语言中，调用一个子程序完全是一种静态的活动。例如在 Pascal 中，如果一条语句调用了子程序 p，编译器通常会生成代码，创建一个新的调用栈，将正确的参数放入栈中，然后改变控制流，开始执行子程序 P 相关的代码。但是，在支持某种形式的多态的语言中，如 Smalltalk 和 C++，调用一个操作可能需要动态的活动，因为被操作的对象的类可能要到运行时刻才能知道。如果我们加入继承，事情就更有趣了。在有继承而没有多态的情况下，调用一个操作的语义基本上和简单的静态子程序调用是一

[1] 这实际上是多态的判别测试。如果存在 switch 语句，根据对象的类型来选择一个动作，这通常就是一个警告信号，表明开发者没有有效地应用多态行为。

样的，但在有多态的情况下，则必须使用更为复杂的技术。

考虑图 3-10 中的类层次结构，它展示了基类 DisplayItem 和三个子类，分别是 Circle、Triangle 和 Rectangle。Rectangle 也有一个子类，名为 SolidRectangle。在类 DisplayItem 中，假定定义了实例变量 theCenter（表示该显示项中心的坐标）以及下面的操作，作为我们早期的例子：

- draw　画出该项；
- move　移动该项；
- location　返回该项的位置。

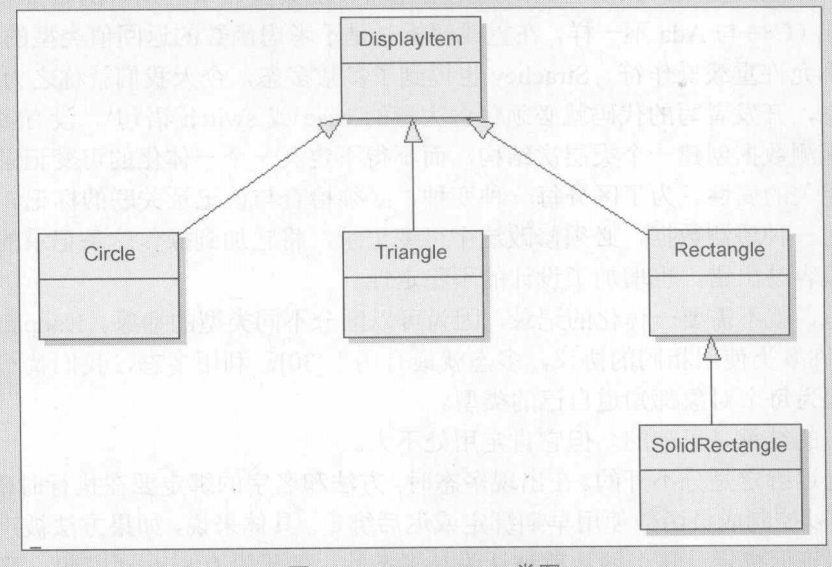


图 3-10　DisplayItem 类图

location 操作对于所有子类都是相同的，因此不需要重新定义，但我们预计 draw 操作和 move 操作会被重新定义，因为只有子类才知道如何画出和移动它们自己。

Circle 类必须包含实例变量 theRadius 和相应的操作来存取它的值。对于这个子类，重新定义的操作 draw 会画出以 theCenter 为圆心给定半径的圆。类似地，Rectangle 类必须包含实例变量 theHeight 和 theWidth 以及相应的操作来存取它们的值。对于这个子类，重新定义的操作 draw 会根据给定的高和宽画出矩形（也是以 theCenter 为中心）。子类 SolidRectangle 继承了 Rectangle 类的所有属性，但又重新定义了 draw 操作的行为。具体来说，SolidRectangle 类的 draw 实现首先调用了超类 Rectangle 中定义的 draw（画出矩形的外框），然后再填充这个矩形。调用 draw 操作要求实现多态行为。

现在，假定我们有一些客户对象希望画出所有的子类。在这种情况下，编译器不能够静态地生成代码来调用正确的 draw 操作，因为要到运行时刻才能知道被调用对象的类。让我们来考虑不同的面向对象语言是如何处理这种情况的。

因为 Smalltalk 是一种无类型的语言，所以方法选择是完全动态的。当客户向列表中的一项发出 draw 消息时，发生的事情如下：

- 该对象从它的类消息字典中查找该消息；
- 如果找到该消息，本地定义的方法代码就被调用；
- 如果找不到该消息，就继续在超类中查找该方法。

这个过程会沿着超类层次结构向上，直到找到该消息，或者直到到达最顶端的基类 Object 时也没找到该消息。在后一种情况下，Smalltalk 最终将发出 doesNotUnderstand 消息，表示出错。

这个算法的关键是消息字典，它是每个类的表示形式中的一部分，因此对于用户是不可见的。这个字典是在类创建时被创建的，包含了这个类的实例可以响应的所有方法。查找这个方法是耗时的，与简单的子程序调用相比，在 Smalltalk 中查找方法要花 1.5 倍的时间。所有产品品质的 Smalltalk 实现都通过提供缓冲的消息字典来优化方法选择，所以一般传递的消息可以实现快速的调用。缓冲通常能改进 20%～30%的性能[31]。

子类 SolidRectangle 中定义的 draw 操作产生了一种特殊情况。我们说它的 draw 实现首先调用了超类 Rectangle 中定义的 draw 操作。在 Smalltalk 中，用关键字 super 来指定一个超类方法。然后，当把 draw 消息传递给 super 时，Smalltalk 就会用前面提到的同样的方法选择算法，只是查找从这个对象的超类开始，而不是从它的类开始。

Deutsch 的研究表明，多态在 85%的情况下是不需要的，所以消息传递常常可以退化为简单的过程调用[32]。Duff 指出，在这种情况下，开发者常常隐含假定允许对象类的早期绑定[33]。遗憾的是，像 Smalltalk 这样的无类型的语言没有方便的方式可以告诉编译器这些隐含的假定。

像 C++这样的更强类型的语言确实让开发者声明这些信息。因为我们希望在可能的情况下避免方法选择，但又必须仍然允许多态选择的情况，所以在这些语言中调用一个方法与在 Smalltalk 中有些不同。

在 C++中，开发者可以将某个操作声明为 virtual，从而决定它是迟后绑定的。所有其他的方法都被认为是早期绑定的，因此编译器可以将方法调用静态解析为简单的子程序调用。

为了处理虚成员函数，大部分 C++实现都使用了 vtable 的概念。在对象创建时（也就是当对象的类被确定时），每个需要多态选择的对象都会定义一个 vtable。这个表通常包含一个虚函数指针列表。例如，如果我们创建了 Rectangle 类的一个对象，那么 vtable 中就有一项是为虚函数 draw 准备的，它指向最近的 draw 实现。例如，如果类 DisplayItem 包含了虚函数 Rotate，它在 Rectangle 类中没有重定义，那么表 Rotate 的 vtable 表项就会指向 DisplayItem 类中的 Rotate 实现。通过这种方式，运行时刻的查找就被省去了：对一个对象的虚成员函数的引用只是通过相应指针的一次间接引用，可以不必查找就立即调用到正确的代码[34]。

3. 多继承

在单继承中，每个子类都只有一个超类。但是，Vlisside 和 Linton 指出，虽然单继承非常有用，“但这常常迫使程序员从两个差不多有吸引力的类中选择一个来继承。这限制了预定义的类的实用性，常常需要重复代码。例如，没有办法派生出既是一个圆也是一张图画的图形，程序员必须从一个类派生，然后重新实现另一个类中的功能”[40]。

考虑人们如何组织不同的资产，如存款账户、房地产、股票和债券。存款账户和支票账户通常都是由银行管理的账户，所以我们可以将它们分类为银行账户，而银行账户是一种资产。股票和债券与银行账户的管理非常不同，所以我们可以将股票、债券、共同基金等分类为有价证券，而有价证券也是一种资产。

但是，还存在许多同样令人满意的方法，可以对存款账户、房地产、股票和债券进行分类。例如在某些情况下，区分出房地产和某些银行账户（在美国，可以根据一些限制条件由联邦存款保险公司保险）这样的可保险资产是有用的。另外，区分出那些返回股息或利息的资产也是有用的，如存款账户、支票账户及某些股票和债券。

遗憾的是，单继承的表达能力不足以刻画这样的网状结构，所以我们必须借助多继承[1]。图 3-11 展示了这样的类结构。这里我们看到，Security（有价证券）类是一种 Asset（资产），也是一种 InterestBearingItem（利息产生项）；类似地，BankAccount（银行账户）是一种 Asset，同时又是一种 InsurableItem（可保险项）和一种 InterestBearingItem。

设计一个合适的、涉及继承特别是多继承的类结构是一项困难的任务。这通常是一个增量和迭代的过程。当我们采用多继承时，会遇到两个问题：如何处理来自不同超类的名字冲突，以及如何处理重复的继承。

当两个或多个不同的超类对接口中的某些元素使用同样的名字时，就可能发生名字冲突，譬如同名的实例变量和方法。例如，假定 InsurableItem 类和 Asset 类都有一个名为 presentValue 的属性，表示该项资产的现值。由于 RealEstate 继承自这两个类，那么继承两个同样名字的操作是什么意思呢？这是多继承的关键困难：名字冲突可能导致多继承的子类的二义性行为。

有三种基本方法可用来解决这种冲突。首先，编程语言的语义可能认为这样的冲突是非法的，拒绝编译这样的类。其次，编程语言的语义可能认为不同类引入的相同名字指的是相同的属性。最后，编程语言的语义可能允许这种冲突，但需要所有对这个名字的引用都有完整的限定符，表明声明它的位置。

第二个问题是重复的继承，Meyer 是这样描述这个问题的：“由于多继承而引起的一个微妙的问题就是，如果一个类通过多个途径成为另一个类的祖先，那会发生什么情况。如果语言允许多继承，那么迟早会有人会写出 D 类有两个父类 B 和 C，而 B 和 C 又都以 A 作为父

[1] 这实际上是多继承的判别测试。如果我们遇到了类形成的网格，其中叶子类可以被分成不同的集合，代表正交的行为（如可以保险和产生利息）。这些集合相互重叠，那么这就表明，利用单继承的结构，我们无法在不打破某些叶子类的抽象的情况下，为它们分配应该具备的操作，从而得到相应的类。可以根据需要利用多继承来混合这些行为，从而解决这种情况。

类……或者其他的情况，使得 D 从 A 继承了两次（或更多）。这种情况被称为重复继承，必须正确地处理。”[41]例如，假定我们将 MutualFund 类（没有深思熟虑地）定义为 Stock 和 Bond 类的子类。这个类引入了对 Security 类的重复继承，Secirity 类既是 Stock 类的超类，也是 Bond 类的超类（参见图 3-11）。

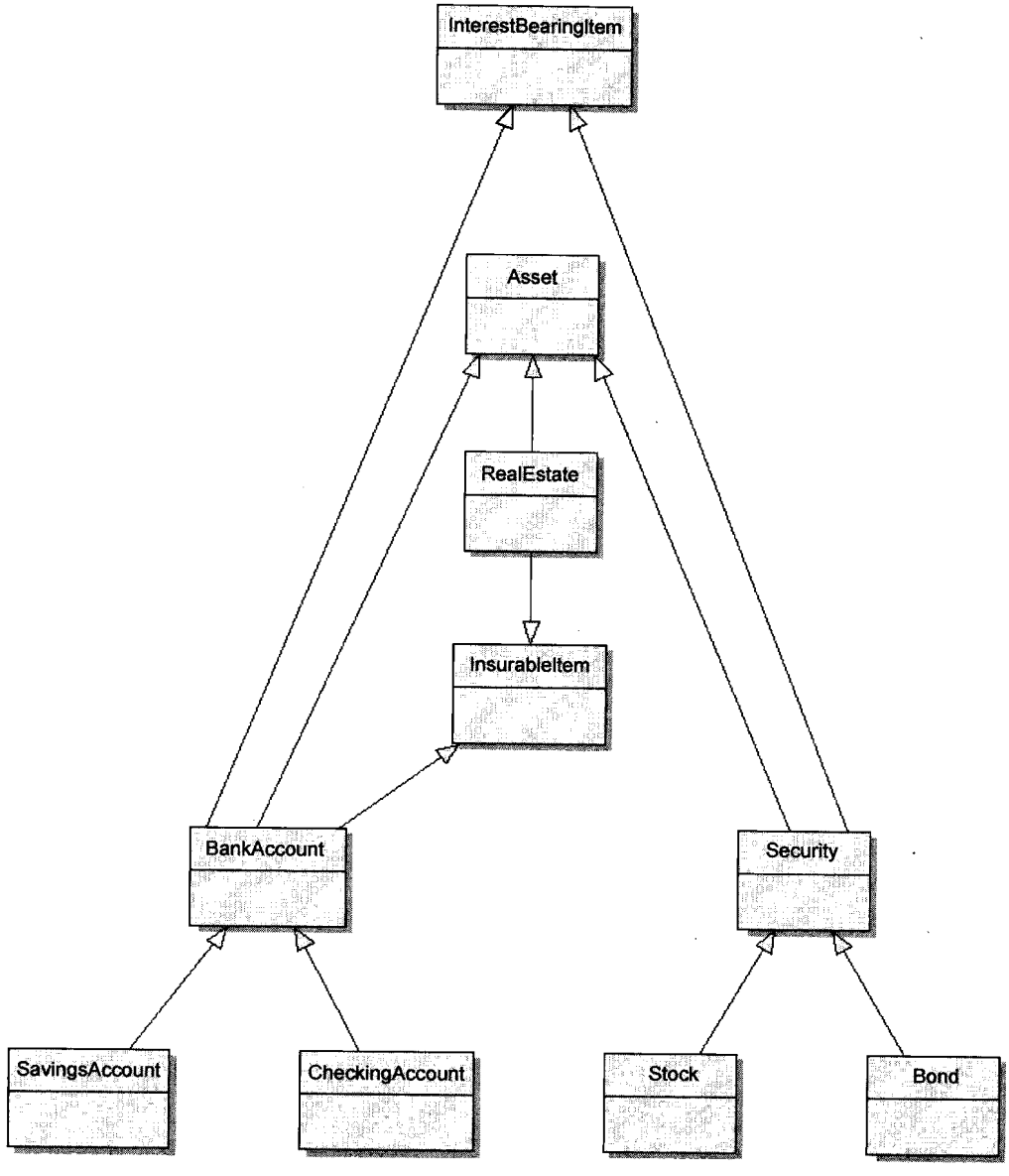


图 3-11 多继承

有几种不同的方法可以用来处理重复继承的问题。首先，可以将重复继承视为非法。其次，可以允许超类的重复，但要求使用完整的限定名来引用成员的具体拷贝。最后，可以将对同一个类的多次引用视为代表相同的类。不同的语言使用不同的方法来处理这个问题。

多重继承的存在促使了所谓的“混入类”（mixin）的出现。混入类来自于 Flavors 语言的编程文化：开发者可以组合（混入）小类来构建更复杂的类。“混入类在语法上等同于一个正常的类，但它的目的是不同的。这种类的目的只是……向其他的 flavor（类）添加功能，开发者永远也不能创建混入类的实例。”[44]在图 3-11 中，InsurableItem 和 IterestBearingItem 都是混入类。这些类都不能单独存在，它们被用于增强其他类的意义。因此，可以将混入类定义为一种包含单一、集中行为的类，通过继承，用于增强其他类的行为。混入的行为通常与被混入类的行为是完全正交的。如果一个类主要是通过继承从混入类派生而来，没有它自己的结构或行为，那么它就被称为聚合类（aggregate class）。

3.4.3 聚合

我们也需要聚合关系，它提供了类实例中的整体/部分关系。类之间的聚合关系与这些类的对象之间的聚合关系是并存的。

如图 3-12 所示，TemperatureController 类代表整体，Heater 类是它的部分之一。这完全对应于图 3-6 中这些类的实例之间的聚合关系。

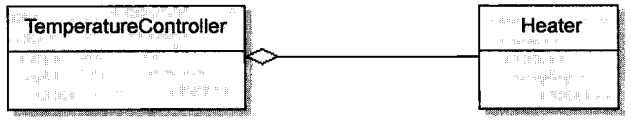


图 3-12 聚合

物理包容

在 TemperatureController 类的例子中，我们看到的是按值包容的聚合。这是一种物理包容，意味着 Heater 对象不会独立于包含它的 TemperatureController 而存在。这两个对象的生存期是紧密联系在一起的：当创建一个 TemperatureController 实例时，也会创建 Heater 类的实例；当销毁 TemperatureController 对象时，意味着也会销毁对应的 Heater 对象。

还有一种不这么直接的聚合，被称为“组合”（composition），它是按引用包容的。在这种情况下，TemperatureController 类仍然代表整体，Heater 类的实例仍然是它的一部分，虽然这一部分现在必须间接地访问。因此，这两个对象的生存期不像前面那么联系紧密，我们可以独立地创建和销毁每个类的实例。

聚合确定了整体部分的方向。例如，Heater 对象是 TemperatureController 对象的一部分，反之则不然。当然，正如在前面的例子中提到的，聚合不一定是物理上的包容。例如，虽然股票持有人拥有股票，但股票持有人并没有在物理上包容拥有的股票。相反，这些对象的生存期完全可以是独立的，虽然概念上的整体/部分关系仍然存在（每股股票都是股票持有者的一部分资产）。这种聚合的表示形式可以是非常间接的。

这仍是聚合，虽然它不是物理上的包容。总之，聚合的判别测试是：当且仅当两个对象之间存在整体/部分关系时，它们对应的类之间必然存在一种聚合关系。

多继承常常与聚合产生混淆。当考虑继承还是聚合时，要记得应用它们的判别测试。如果不能够肯定两个类之间存在“是一种”的关系，那么应该使用聚合或其他的关系来代替继承。

3.4.4 依赖关系

除了继承、聚合和关联之外，还有一类关系，被称为“依赖关系”。依赖关系表明，处于这种关系一端的元素以某种方式依赖于处于另一端的元素。这警告了设计者，如果其中一个元素发生了改变，可能会影响到另一个元素。存在许多种不同的依赖关系（完整列表请参见对象管理组织最新的 UML 规范[45]）。常常会在架构模型（一个系统组件或包依赖于另一个组件或包）或实现层（一个模块依赖于另一个模块）中看到依赖关系。

3.5 类与对象的互动

类和对象既是独立的概念，又密切相关。具体来说，每个对象都是某个类的一个实例，每个类都有 0 或多个实例。对于所有的应用，类几乎都是静态的，因此，它们的存在、语义和关系在执行程序之前都是确定的。类似地，绝大多数对象的类都是静态的，这意味着对象一旦被创建，它的类就确定了。与此形成鲜明对比的是，对象通常在应用的生存期中频繁地被创建和销毁。

3.5.1 类与对象的关系

例如，考虑一个空中交通管制系统的实现中的类和对象。其中一些比较重要的抽象包括飞机、飞行计划、跑道和空域（air space）。根据它们的定义，这些类和对象的含义是比较静态的。它们必须是静态的，否则开发者就不能够创建一个应用，这包含以下常识：飞机可以起飞、飞行和降落，两架飞机不应该同时占用相同的空间。与此不同，这些类的实例是动态的。以一种相当低的频率，人们会修建新的跑道，废弃老的跑道；稍频繁一点，新的飞行计划会被发布，老的飞行计划会被废弃；再频繁一些，新的飞机进入某个空域，老的飞机离开。

3.5.2 类与对象在分析和设计中的角色

在分析阶段和设计的早期阶段，开发者有两项主要任务：

- 从问题域的词汇表中确定出类；
- 创建一些结构，让多组对象一起工作，提供满足问题需求的行为。

我们将这样的类和对象统称为问题的“关键抽象”，把这些协作结构称为实现的“机制”。在开发这些的阶段中，开发者必须关注这些关键抽象和机制的外部视图。外部视图代表

了系统的逻辑框架，因此包含了系统的类结构和对象结构。在设计阶段的后期以及随后的实现阶段，开发者的任务发生了变化，其关注的焦点放在了这些抽象和机制的内部视图上，包括它们的物理实现。

3.6 创建高品质的类与对象

Ingalls 建议："系统应该利用一组最少的不会变化的部分进行构建，这些部分应该尽可能地通用，系统的所有部分应该被放入一个统一的框架。"[51]对于面向对象开发来说，这些部分就是构成系统关键抽象的类和对象，这个框架由系统的机制来提供。

根据我们的经验，类和对象的设计是一个增量、迭代的过程。坦白地说，除了那些最不重要的抽象，我们从来没有第一次就完全正确地定义一个类。对于最初的抽象，需要花一些时间来琢磨它粗糙的概念边界。当然，精化这些抽象是有代价的，包括系统的重新设计、系统设计的可理解性和系统设计结构的完整性等方面。因此，我们希望在一开始就尽量正确。

3.6.1 评判一种抽象的品质

人们怎样才能知道某个类或对象的设计是良好的？我们建议使用下面 5 个测量指标：

- 耦合；
- 内聚；
- 充分性；
- 完整性；
- 基础性。

耦合是来自于结构化设计的一个概念，但从字面上的解释来看，它也适合于面向对象设计。Stevens、Myers 和 Constantine 将耦合定义为"一个模块与另一个模块之间建立起的关联强度的测量。强耦合使系统变得复杂，因为如果模块与其他模块高度相关，它就难以独立地被理解、变化或修正。通过设计系统，使模块间的耦合降至最弱，可以降低复杂性。"[52] Page-Jones 给出了一个好的耦合的反例，他描述了一个模块化的立体声系统，其供电部分位于一个音箱之中[53]。

模块之间的耦合也适用于面向对象的分析和设计，但类和对象之间的耦合同样重要。然后，在耦合和继承的概念之间存在着矛盾关系，因为继承引入了严重的耦合。一方面，我们希望类之间的弱耦合，另一方面，继承（超类和子类之间的强耦合）又能帮助我们处理抽象之间的共性。

内聚的思想也来自结构化设计。简单地说，内聚测量了单个模块（对于面向对象来说，就是单个类或单个对象）内各个元素的联系程度。最不希望出现的内聚就是偶然性内聚，即将完全无关的抽象塞进同一个类或模块中。例如，考虑由狗和航天飞机的抽象组成的一个类，这两种抽象的行为基本上是无关的。最希望出现的内聚就是功能性内聚，即一个类或模式的

各元素一同工作，提供某种清晰界定的行为。因此，如果 Dog 类的语义包含了一只狗的行为——完全是狗，只有狗而没有其他，那么它就是功能性内聚。

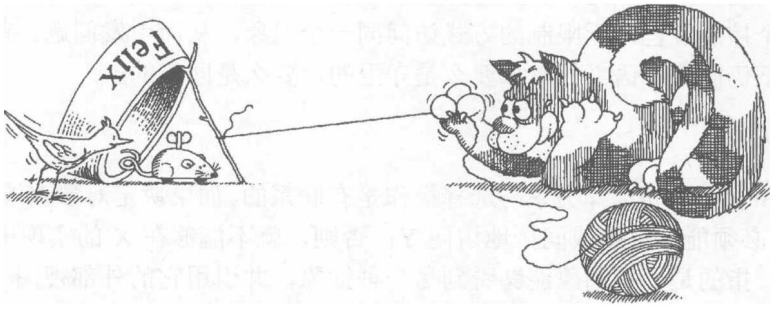
与耦合和内聚的概念相关，一个类或模块应该是充分的、完整的、简单的。所谓充分，指的是类或模块应该记录某个抽象足够多的特征，从而允许有意义的、有效的交互。否则，将使该组件变得无用。例如，如果我们设计 Set（集合）类，应该包含从集合中删除元素的操作，但如果忘记了加入元素的操作，我们的努力就徒劳了。在实践中，这种问题很早就会被发现。只要我们构建一个必须使用这种抽象的客户时，就会发现这样的缺陷。

所谓完整，指的是类或模块的接口记录了某个抽象全部有意义的特征。充分性意味着最小的接口，但一个完整的接口就意味着该接口包含了某个抽象的所有方向。因此完整的类或模块是足够通用的，可以被任何客户使用。完整性是一种主观判断，有可能做过头。为某个抽象提供全部有意义的操作会让用户不知所措，通常这也是不必要的，因为许多高级操作可以由低级操作组合得到。出于这个原因，我们也建议类和模块应该具有基础性。

基础性操作就是只有访问该抽象的底层表现形式才能够有效地实现的那些操作。因此，对一个集合添加一个元素是一项基础性操作，因为要实现这个 Add 操作，必须知道底层表现形式。另一方面，向集合添加 4 项元素的操作不是基础性操作，因为它可以通过更为基础性的 Add 操作来有效地实现，不必访问底层的表现形式。当然，有效与否也是一种主观的判断。如果一个操作只能通过访问底层表现形式来实现，那它无疑是基础性的。如果一个操作可以在已有的基础性操作之上实现，但这样做会消耗大量的计算资源，那么它也可以作为基础性操作的候选者。

3.6.2 选择操作

打造类或模块的接口是艰苦的工作。通常，我们先尝试设计类，然后，当我们和其他人创建客户时，我们发现需要扩展、修改并进一步精化这个接口。最后，我们可能发现一些操作的模式或抽象的模式，导致我们创建一些新的类或重新组织已有类之间的关系。



我们常常能够识别一些抽象模式、结构模式或行为模式

1. 功能语义

对于一个给定的类，我们的风格是让所有的操作保持基础性，这样每个操作都展示出小

的、定义良好的行为。我们把这样的方法称为“细致的（fine-grained）”。我们也倾向于分离方法，让它们相互之间不通信。通过这种方式，我们更容易构造一些子类，它们可以有意义地重新定义超类的行为。将一个行为提取为一个方法还是多个方法的决定有两个互相竞争的原因：将一个行为塞进一个方法中将导致更简单的接口，但方法会更大、更复杂；将一个行为分散到多个方法中将导致更复杂的接口，但方法会更简单。Meyer 指出：“好的设计者知道如何在在太多契约和太少契约之间平衡折中，太多契约导致片段化，太少契约导致无法管理的大模块。”[54]

在面向对象开发中，通常会整体设计一个类的所有方法，因为所有这些方法共同构成了这个抽象的完整协议。因此，对于某个期望的行为，我们必须决定将它放到哪个类中。Halbert 和 O'Brien 提出了下面的判断条件，需在做这样的决定时思考[55]：

- 可复用性　这个行为可以在多种上下文中使用吗？
- 复杂性　实现这个行为的难度有多大？
- 适用性　这个行为与打算放入的类型之间相关程度如何？
- 实现知识　这个行为的实现依赖于一个类型的内部细节吗？

我们通常选择声明一些有意义的操作，这样就可以针对一个对象执行它的类（或超类）中定义的方法。

2. 时间和空间语义

当确定存在某个操作，并定义了它的功能语言之后，必须决定它的时间语义和空间语义。这意味着我们必须决定它完成操作需要的时间以及存储空间。这样的决定通常是用最佳、平均和最差等术语来表达的，最差的情况规定了能够接受的上限。

前面也曾提到，当一个对象通过一个链接向另一个对象传递消息时，这两个对象必须以某种方式同步。在多控制线程的情况下，这意味着消息的传递比子程序调用要更复杂。在使用的大多数语言中，对象之间的同步不成问题，因为我们的程序只包含一个控制线程，这意味着所有的对象都是被依次访问的。这种情况下的消息传递很简单，因为它的语义基本上与简单的子程序调用相同。但是，在支持并发的语言中，我们必须关注更为复杂的消息传递形式，以避免两个控制线程以无限制的方式访问同一个对象，从而引发问题。前面曾提到，在多个控制线程下仍能保持语义的对象要么是守卫的，要么是同步的。

3.6.3 选择关系

在类之间和对象之间选择关系与选择操作是有联系的。如果决定对象 X 向对象 Y 发送消息 M，那么 X 必须能够直接或间接地访问 Y；否则，就不能够在 X 的实现中命名操作 M。所谓能够访问，指的是一种抽象能够看到另一种抽象，并引用它的外部视图中的资源。只有当它们的范围重叠，并且访问得到授权时（例如，类的私有部分只能被该类本身和它的友元访问），一种抽象才可以访问另一种抽象。因此，耦合是度量可访问程度的指标。

1. Demeter 法则

在选择对象间的关系时，有一项有用的方针，称为 Demeter 法则。它指出，“类的方法不

应该以任何方式依赖于任何类的结构，除了它自己类的当前（顶层）结构之外。而且，每个方法只能够对一个非常有限的类集的对象发出消息”[56]。应用这一法则的基本效果就是创建了一些松耦合的类，它们的实现秘密被封装了起来。这样的类是相当没有负担的，即为了理解一个类的意思，不需要理解许多其他类的细节。

在查看整个系统的类结构时，可能会发现它的继承关系宽而浅，或者窄而深，或者比较平衡。类结构宽而浅通常代表由独立的类构成的森林，它们之间可以混合或匹配[57]。类结构窄而深则表明各个类构成的树都与一个共同的祖先有关[58]。每种方式都有优点和不足。类的森林耦合更松，但它们可能没有体现出存在的共性。类的树体现了这种共性，所以单个类比森林中的类要小。但是要理解某个类时，通常需要理解它继承或用到的所有类的含义。类结构的正确形态与问题是高度相关的。

我们必须在继承、聚合、依赖关系之间进行类似的折中。例如，Car 类应该继承、包含，还是使用名为 Engine 和 Wheel 的类？在这个例子中，我们建议，聚合关系比继承关系更合适。Meyer 指出在类 A 和类 B 之间，“只有当 B 的每个实例都可以被看作 A 的实例时，才适合继承。当 B 的每个实例只是具有 A 的一个或多个属性时，适合使用客户关系”[59]。从另外一个角度来看，如果对象的行为超过了它的部分之和，在相关的类之间创建聚合关系而不是继承关系可能更好。

2. 机制和可见性

决定对象之间的关系主要是设计这些对象进行交互的机制。开发者要问的问题很简单：这些知识应该放在哪里？例如，在一份制造计划中，物料（称为批次）进入制造车间等待处理。当它们进入特定的车间时，我们必须通知车间的经理采取相应的行动。现在面临一个设计选择：一个批次的物料进入一个车间，这个操作是车间的操作，是批次的操作，还是两者的操作？如果我们决定它是车间的操作，那么车间就必须对批次可见。如果我们决定它是批次的操作，那么批次就必须对车间可见，因为必须知道它进了哪个车间。最后，如果我们认为它既是车间的操作，也是批次的操作，那么就必须使它们互相可见。我们还必须决定车间和经理之间的某种可见关系（而不是批次和经理），经理必须知道要管理的车间，或者车间必须知道它的经理。

3.6.4 选择实现

只有当我们使某个类或对象的外部视图稳定下来之后，才会转向它的内部视图。这个视图涉及两个不同的决定：为类或对象选择表示形式，以及将类或对象放入一个模块。

1. 表示形式

类或对象的表示形式几乎总是该抽象封装起来的秘密。这使得我们可以改变表示形式（例如，改变时间语义和空间语义），同时又不会违反客户对功能所做的任何假定。Wirth 指出：“表示形式的选择通常很困难，它不是由可用的方法唯一确定的。它总是必须注意到施加在数据上的操作。”[60]例如，有一个类的对象代表了一组飞行计划信息，我们需要优化它的表示形式以实现快速的查找，或者实现快速的插入和删除？我们不能同时优化两个方面，所

以我们的选择必须基于对这些对象的使用预期。有时候这不容易选择，我们最后得到了一些类，它们的接口是一样的，但它们的实现却非常不同，目的是提供不同的时间和空间特性。

在选择类的实现时，有一个更难的折中：计算对象状态的值，还是将它保存为一个字段。例如，假定我们有一个 Cone（圆锥）类，它包含一个 Volume（体积）方法。调用这个方法将返回这个对象的体积。作为这个类的表示形式的一部分，我们可能利用字段来保存圆锥的高和底面的半径。是否应该加一个字段来保存对象的体积？还是让 Volume 方法每次计算体积[61]？如果我们希望这个方法快速，应该将体积保存为一个字段；如果空间效率对我们更重要，我们就应该计算体积。哪种表示形式更好完全取决于具体的问题。不论哪种情况，我们都应该能够在不改变类的外部视图的情况下选择实现。实际上，我们甚至应该能够在类的客户不关心的情况下，改变这些表示形式。

2. 打包

类似的问题也适用于在模块中声明类和对象。可见性和信息隐藏这一对矛盾的需求通常引导我们决定在哪里声明类和对象。一般来说，我们追求构建功能内聚的、松耦合的模块。许多非技术因素会影响这些决定，如复用、安全及文档。像类和对象的设计一样，模块的设计也很重要，不应忽视。关于信息隐藏，Parnas、Clements 和 Weiss 指出，“应用这一原则并不总是很容易。它试图使软件在使用期的预期费用最小化，并要求设计者估计出变化的可能性。这样的估计基于过去的经验，通常要求应用领域的知识，并要求理解硬件和软件技术”[63]。

3.7 小结

- 对象具有状态、行为和标识符。
- 类似对象的结构和行为定义在它们共同的类中。
- 对象的状态包括对象所有的属性（通常是静态的）加上这些属性当前的值（通常是动态的）。
- 行为是对象在状态改变和消息传递方面的动作和反应。
- 标识符是对象的属性，它区分这个对象和其他对象。
- 类是具有共同的结构和行为的一组对象。
- 三种关系包括关联、继承和聚合。
- 关键抽象是来自于问题域词汇表的类和对象。
- 机制是一种结构，一组对象通过它互相协作，提供满足问题域的某种需求的行为。
- 抽象的品质可以通过它的耦合、内聚、充分性、完整性和基础性来度量。

第4章

分类

分类是组织知识的手段。在面向对象设计中，认识到事物间的相似性让我们能够将共性放在关键抽象和机制中，最终导致更小的应用和更简单的架构。不幸的是，没有实现分类的金光大道。对于那些习惯寻找菜谱答案的读者，我们明确地指出，确定类和对象没有简单的诀窍。没有所谓的“完美”类结构，也没有一组“正确”的对象。像所有工程学科一样，我们的设计选择是对许多竞争因素的折中。

幸运的是，在其他学科中存在着大量有关分类的历史经验。从一些更为经典的方法中，面向对象分析技术出现了，它提出了一些有用的、值得推荐的实践和经验法则，用于确定某个特定领域相关的类和对象。这些启发就是本章关注的重点。

4.1 正确分类的重要性

确定类和对象是面向对象分析和设计中具有挑战性的部分。经验表明，这种确定过程既涉及发现，也涉及发明。通过发现，我们逐渐从问题域的词汇表中识别出关键抽象和机制。通过发明，我们设计出泛化的抽象和一些新的机制，规定对象协作的方式。归根到底，发现和发明都是分类问题，而分类基本上是发现共性的问题。当进行分类时，我们寻找具有共同结构或表现出共同行为的一组事物。

明智的分类实际上也有好的科学的组成部分。Michalski 和 Stepp 指出：“科学中一个无所不在的问题就是对观察到的对象或情况进行有意义地分类。这样的分类促进了人们对观察到的情况的理解，而且有助于此后形成科学理论。”[2]同样的道理也适用于工程领域。在建筑设计和城市规划领域中，Alexander 指出，对于建筑师来说，“他的设计行为，不论是细微还是庞大而复杂，都完全是由当时在他头脑中的模式以及他将这些模式组合成一个新设计的能力所决定的”[3]。所以不奇怪，分类与面向对象设计的方方面面都是有关的。

分类帮助我们确定类之间的泛化、特化、聚合等层次结构。通过识别对象交互中的共同

模式，我们逐渐发明一些机制，成为实现的核心。分类也指导我们做出有关模块化的决定。可以选择将一些类和对象放在同一个模块或不同的模块中，这取决于我们对于这些类和对象所发现的共性。耦合和内聚也表明了一种类型的共性。在将处理过程分配到不同处理器上时，分类也起了作用。可将某些处理过程放在相同处理器或不同处理器上，这取决于分包、性能或可靠性的考虑。

4.1.1 分类的困难

在前一章中，我们将对象定义为有清楚定义的边界的事物。但是，将一个对象与其他对象分开来的边界常常非常模糊。例如，请看我们的腿。膝盖从哪里开始，到哪里结束？在识别人的谈话时，我们怎么知道某些声音会构成一个词，而不是前后的词的一部分？再考虑一下字处理系统的设计。字符是否构成一个类？或者，整个单词构成一个类是更好的选择？我们如何处理任意的、非连续的文本选择？另外，关于句子、段落或整篇文章，这些对象的类与我们的问题相关吗？

明智的分类很难，这并不是什么新闻。既然存在许多与面向对象设计中的问题类似的问题，那么让我们先看看两门其他学科中的分类问题——生物学和化学。

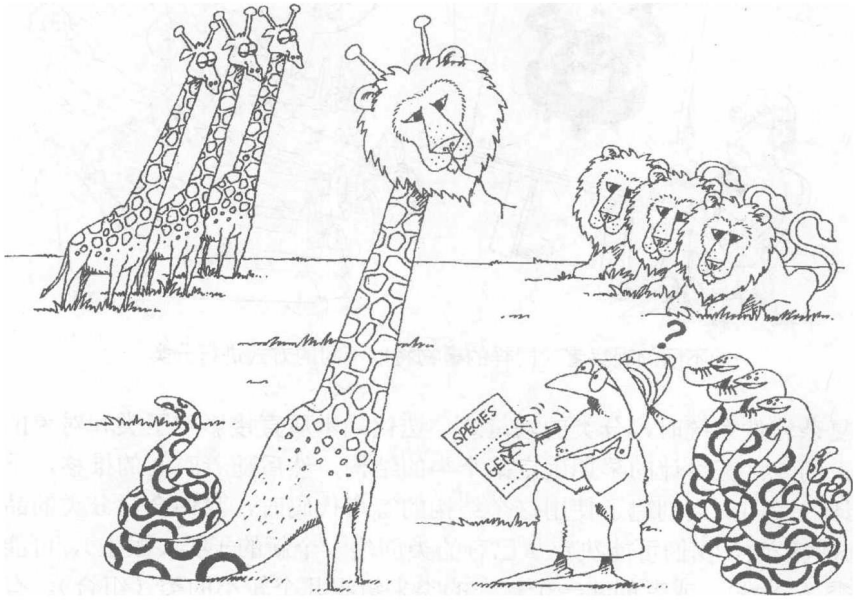
直到 18 世纪，社会上普遍的科学认识仍然是所有活着的有机体可以从最简单到最复杂进行分类，同时复杂度的测量又是相当主观的（不奇怪，人通常是被放在这个列表的最顶端的）。但在 18 世纪中期，瑞士植物学家 Carolus Linnaeus 提出了一种更详细的生物分类学，即根据他所谓的“属”和“种”。

一个世纪之后，达尔文提出自然选择是进化机制的理论，他指出今天的物种是从较早期的物种进化来的。达尔文的理论是建立在对物种的明智分类的基础之上的。达尔文自己曾说，自然学家“试图将种、属和科进行分类，形成所谓的自然系统。但这个系统的意义何在？某些作者只是将它看成是将相似的生物放在一起的一种方法，将它们与不相似的生物分开来”[4]。在现代生物学中，分类意味着“在推断的生物间自然关系的基础上，建立起分类系统的层次结构”[5]。在生物分类学中，最一般的分类方法是分成界、门、亚门、纲、目、科、属、种。

对于计算机科学家来说，生物学可能是一门难懂而成熟的学科，对于生物的分类有着定义良好的判别标准。实际上并非如此。“让人吃惊的是，科学家们更清楚银河中有多少星星，而不是地球上有多少物种。对于全球物种的估计在 200 万到 1 亿之间，最好的估计是在 1000 万个左右，只有 140 万个是有确切名称的。”[65]而且，对于同样生物应用不同判别标准会得到不同的结果。Martin 指出，“这都取决于想利用分类来做什么。如果想准确地反映这些物种基因上的联系，将得到一种答案。但如果想说明适应的程度，那就会得到另一种答案”[8]。这里的要点是，即使在十分严格的科学学科中，分类也高度取决于进行分类的原因。

我们可以在化学中看到类似的情况[9]。在很久以前，所有的物质都被看成是土、气、火和水的组合。从今天的标准来看（除非你是炼金术士），这并不是很好的分类方式。在 17 世纪中期，化学家 Robert Boyle 提出元素是化学的基本抽象，从元素可以构成更复杂的化合物。

只到一个世纪之后，在 1789 年，化学家 Lavoisier 才列出了第一个元素清单，包含大约 23 种元素，其中的某些后来被发现根本不是元素。此后，新的元素不断被发现，清单不断增长。最后，在 1869 年，化学家门捷列夫提出了周期律，为组织所有已知的元素提供了准确的判定条件，并且能够预言那些尚未发现的元素。周期律并不是元素分类故事的结束。在 20 世纪早期，具有类似化学性质但原子量不同的元素被发现了，这引入了同位素概念。

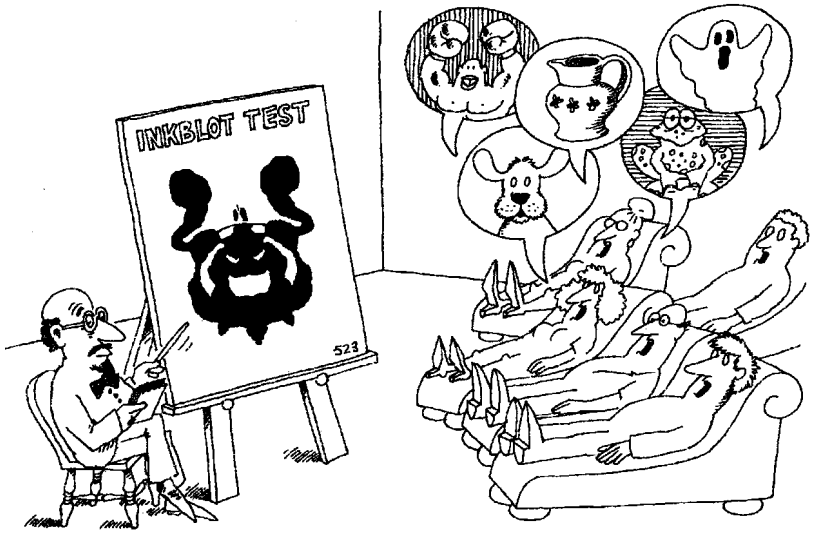


分类是我们组织知识的手段

这里的经验很简单：Descartes 指出，“发现一种秩序不是容易的任务……但是一旦秩序被发现，了解它是没有什么难度的”[10]。最好的软件设计看起来很简单，但是经验表明，需要很多艰苦的工作才能设计出简单的架构。

4.1.2 分类的增量和迭代本质

前面所说的并不是为缓慢的软件开发进度在辩解，虽然对于管理层或最终用户来说，软件工程师似乎需要几个世纪才能完成他们的工作。我们之所以说这些，是想指出明智的分类是很艰难的智力工作，最好是通过一个增量式、迭代式的过程来完成。Shaw 曾指出，在软件工程中，“单个抽象的形成常常遵循某个常见的模式。首先，问题以一种随意的方式得到解决。随着经验的积累，人们发现某些解决方式比另外一些解决方式效果更好，一些好方法在人们之间口口相传。最后，有用的解决方案得到了更系统的理解，人们对它们进行编码和分析。这使得人们能够形成支持自动化实现的模型，以及让这个解决方案更通用的理论。这反过来又导致了更复杂层次的实践，让我们能够处理更复杂的问题——我们对这些问题通常还是采取随意的方式，于是这个循环又开始了”[11]。



不同的观察者对同样的事物将以不同的方式进行分类

在设计复杂软件系统时，分类的增量式、迭代式本质直接影响了类和对象的层次结构。在实践中，人们常常在设计的早期确定某个类的结构，然后随着时间的推移，不断修改这个结构。在设计的后期，当创建了使用这个结构的客户代码时，我们会对分类的品质有更深的认识。基于这些经验，我们可能决定从已有的类创建一个新的子类（派生），可能将一个大类分为几个小类（分解），或者创建一个较大的类来组织几个较小的类（组合）。有时候，我们甚至可能发现以前没有意识到的共性，并设计出一个新类（抽象）[12]。

那么，为什么分类这么难？我们认为有两个重要的原因。首先，不存在所谓的“完美”分类，尽管某些分类肯定比另外一些更好。Coombs、Raiffa 和 Thrall 指出，“如果要将系统划分为对象系统，那么有多少科学家参与这项工作，就可能有多少种划分方法”[13]。任何分类都与进行分类的观察者的视角有关。其次，明智的分类要求大量的创造性思维。Birtwistle、Dahl、Myhrhaug 和 Nygard 指出，“有时候答案很明显，有时候它是个品味问题，另外一些时候，选择合适的组件是分析中的关键之处”[15]。这一事实让我们想起一个谜语：“为什么说激光像金鱼？……因为它们都不会吹口哨。”[16]只有创造性的思维才能在这种似乎无关的事物之间发现共性。

4.2 确定类和对象

从柏拉图时代以前开始，无数的哲学家、语言学家、认知科学家和数学家都考虑过分类的问题。显然，我们应该学习他们的经验，并将所学的东西应用到面向对象设计中去。

4.2.1 经典方法和现代方法

从历史上来看，存在三种一般的分类方式：

- 经典分类；
- 概念聚集；
- 原型理论[17]。

1. 经典分类

在经典的分类方法中，“所有具有某一个或某一组共同属性的实体构成了一个分类。这样的属性对于定义这个分类是必要的，也是充分的”[18]。例如，已婚人士构成了一个分类。一个人要么已婚，要么未婚，这个属性的值足以确定某个人属于哪一个人群。但是，高的人不会构成一个分类，除非我们可以一致同意以某个绝对标准来区分高矮这个属性。

经典分类法首先来自柏拉图，然后亚里士多德对植物和动物进行了分类，他使用的技术非常类似于现在小孩玩的“20 个问题”的游戏（它是动物、矿物，还是蔬菜？它有毛还是羽毛？它能飞吗？它有气味吗？）[20]。后来的哲学家采用了这种方法，最著名的有 Aquinas、Descartes 和 Locke。Aquinas 指出，“根据我们对事物所具有的属性和效用等本质的知识，我们可以对它进行命名”[21]。

经典的分类理论也反映了儿童发展的现代理论。皮亚杰指出，在 1 岁左右，儿童通常发展出物体存在的概念，此后不久，儿童获得了对这些物体进行分类的能力，首先使用像猫、狗、玩具等基本分类[22]。后来，儿童发现了更一般的分类（如动物）和更具体的分类（如猎兔犬）[23]。

总之，经典分类法利用相关的属性作为对象间相似性的判据。具体来说，人们可以根据某一属性是否存在，将对象划分到没有交集的集合中。Minsky 指出，“最有用的属性集合是其成员没有太多相互影响的集合。这解释了为什么下面的属性组合被广泛采用：尺寸、颜色、形状和物质。因为这些属性几乎相互之间没有影响，可以将它们任意组合，得到的对象或大或小、或红或绿、或木质或玻璃，形状上或球形或立方”[24]。一般来说，属性可以不只表示可以测量的特征，也可以包含观察到的行为。例如，鸟能飞而鱼不能飞这一事实可以作为一个属性，用于区分鹰和鲑鱼。

在特定情况下应该考虑哪些属性，这与领域是高度相关的。例如在汽车制造厂中，汽车的颜色对于库存控制可能是很重要的，但对于大城市里控制交通灯的软件来说，颜色就根本无关了。这就是为什么我们说分类没有绝对的标准，但是某个类结构可能比另一个类结构更适合某个应用。James 指出，“没有哪一种分类方法比其他的分类方法更能反映自然的真正结构或秩序。自然漠然地将自己奉献出来，我们可以按自己的意愿对存在的事物进行划分。某些分类可能比其他分类更重要，但这只是从我们感兴趣的角度来说，而不是因为它们更准确或更充分地反映了真实世界”[25]。

经典分类法渗入了许多当代的西方思想，但是正如前面提到的高个和矮个的例子那样，这种方法并非总是令人满意。Kosko 指出，“自然的分类会趋向混乱：大多数鸟会飞，但有些

鸟不会飞；椅子可以由木头、塑料或金属构成，几乎可以有任意条腿，这完全取决于它的设计者。对于任何自然的分类来说，在实践中似乎不可能得到一个属列表，能够排除所有不在这个分类中的个例并包含所有在这个分类中的个例”[26]。这是分类中真正的基本问题，概念聚集和原型理论试图解决这一问题。

2. 概念聚集

概念聚集是经典方式较为现代的变种，主要源自于对解释知识的表现方式的尝试。Stepp和Michalski指出，“在这种方法中，类（一些实体的聚集）的产生首先是形成类的概念描述，然后再根据这些描述对实体进行分类”[27]。例如，我们可以声明像“一首爱情歌曲”这样的概念。这个概念超越了一个属性，因为任何歌曲的“爱情歌曲特质”不是可以经验性地测量的东西。但是，如果我们确定某个歌曲更像是一首爱情歌曲，我们就会把它放到这个分类中。因此，概念聚集更多地代表了对象聚集的可能性。

概念聚集与模糊（多值）集理论是有密切关系的，在这种理论中，对象可以按不同的适合程度属于一个或多个分组。概念聚集通过关注“最适合”来进行绝对的分类判断。

分类的一个问题

图4-1包含了10个项，标签从A~J，每项代表了一列火车。每列火车包含一个火车头（在右边）及2~4个车厢，每个车厢形状不同，装载的货物也不同。在继续读下去之前，请花几分钟按照你认为有意义的方式将这些火车分成任意多个小组。例如，可以创建三个组：一个小组包含所有火车头的车轮都是黑色的火车，一个小组包含所有火车头的车轮都是白色的火车，另一个小组包含火车头的车轮有黑有白的火车。

这个问题来自于Stepp和Michalski在概念聚集方面的工作[19]。在实际生活中，没有“正确”的答案。在它们的实验中，被测试者提出了93种不同的分类方式。最常见的分类方式是根据火车的长度，得到了三个分组（有2节、3节和4节车厢的火车）。第二常见的分类方式是根据火车头轮子的颜色，像我们建议的那样。在93种分类方式中，大约40种方式是完全独特的。

我们通过这个例子来强调Stepp和Michalski的研究。大多数被测试者使用了两种最常见的分类方式，但是我们也遇到了一些相当有创造性的分类方式。例如，有一个被测试者将这些火车分为两组：一组火车的标签中只包含直线（A、E、F、H和I），另一组火车的标签中包含曲线。这确实是非线性思维的例子，有创造性，但并不奇怪。

完成了这个任务之后，让我们改变一下需求（这又和真实世界一样）。假定圆形代表有毒化学物质，矩形代表木材，所有其他形状代表乘客，再试试对这些火车进行分类，看看这些新知识如何影响你的分类。

在我们的被测试者中，火车的分组发生了显著的变化，大多数被测试者根据它们是否携带有毒物质来分类。我们从这个实验中得出结论，关于领域的知识越多，就越容易得到明智的分类。

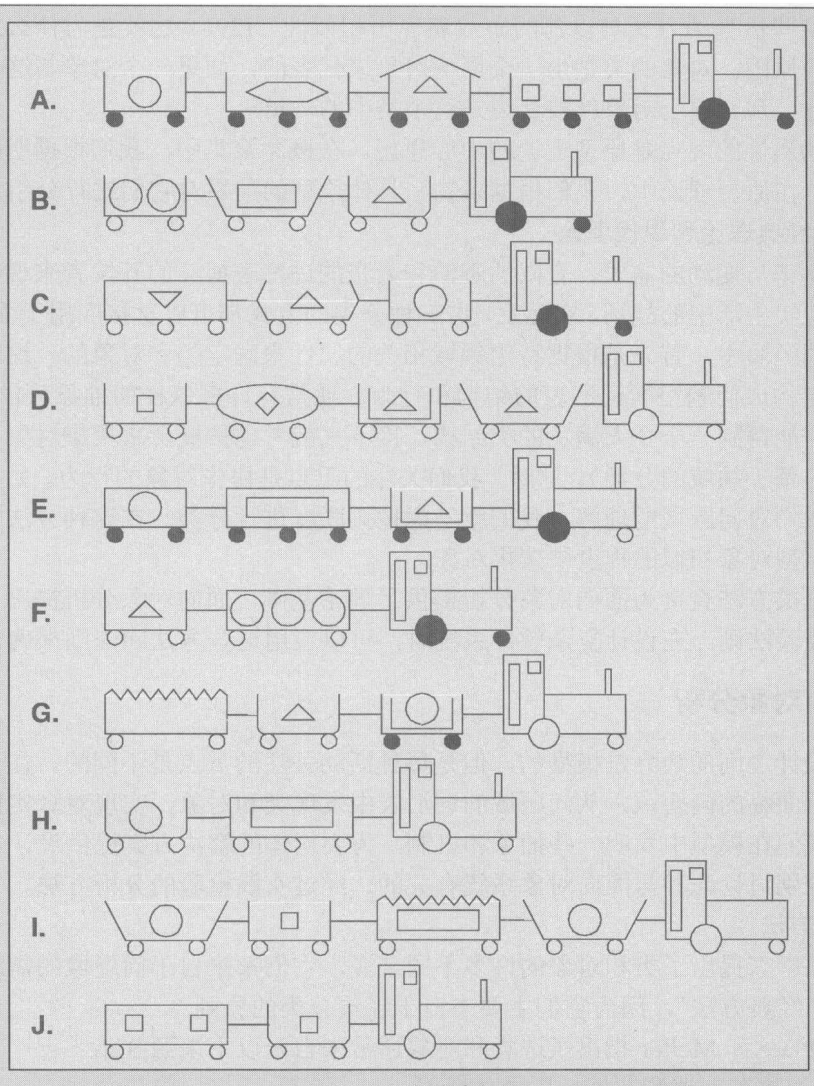


图 4-1 一个分类问题

3. 原型理论

在设计复杂软件系统时，经典分类和概念原型足以表达我们所需要的分类。但是，还有一些情况下，这些方法是不够的。这导致了原型理论，它主要来自于 Rosch 和她的同事在认知心理学方面的研究工作[28]。

有一些抽象既没有清晰界定的属性，也没有清楚的概念。Lakoff 解释了这个问题，“维特根斯坦指出，像游戏这样的分类不适合经典的方式，因为不存在所有游戏共有的属性……虽然没有一组属性是所有游戏共有的，但是游戏的分类是按照维特根斯坦所谓的家族相似性来组织的……维特根斯坦还指出游戏分类没有固定的边界。分类可以扩展，新的游戏被引入，

只要它们以某些恰当的方式与以前的游戏表现出相似性”[29]。这就是为什么这种方式被称为原型理论的原因：对象的类是由一个原型对象来代表的，如果一个对象与这个原型表现出重要的相似性，那么这个对象就被认为是这个类中的一员。

这种互动属性的观点是原型理论的中心思想。在概念聚集中，我们根据明确的概念对事物进行分组。在原型理论中，我们根据事物与具体原型的关系对它们进行分组。

4. 应用经典理论和现代理论

对于那些正与变动的需求、有限的资源和紧张的进度表搏斗的开发者来说，我们的讨论似乎远离了现实生活中的战场。实际上，这三种分类的方式都可以直接应用于面向对象分析。

在我们的经验中，首先是根据特定领域相关的属性来确定类和对象的。这时，我们关注的重点是确定构成问题空间词汇表的结构和行为。通常有许多这样的抽象可供选择[31]。如果这种方法不能得到一个令人满意的类结构，接下来就考虑按概念来聚集对象（或者按概念来优化最初的基于领域的分类）。这时，我们关注的重点是协作对象的行为。如果这两种方法都不能表达我们对问题域的理解，我们就考虑按关联度进行分类。在这种方法中，我们根据对象与某个原型对象相似的程度来聚集对象。

这三种分类方法直接为面向对象分析提供了理论基础，面向对象分析提出了一些实用的实践方法和经验法则。在设计复杂软件系统时，可以应用这些方法和法则来确定类和对象。

4.2.2 面向对象分析

分析和设计之间的边界是模糊的，但是每种活动关注的重点是不同的。在分析时，关注的重点是分析面临的问题域，从问题域的词汇表中发现类和对象，实现对真实世界的建模。在设计时，我们在模型中发明一些抽象和机制，为要构建的解决方案提供设计。

下面将分别讨论几种与面向对象系统有关的、经过实践检验的分析方法。

1. 经典方法

一些方法学家提出了类和对象的许多不同来源，它们都来自于问题域的需求。我们将这些方法称为“经典方法”，因为它们主要来自于经典分类的原则。

例如，Shlaer 和 Mellor 指出候选类和对象通常来自于以下来源[32]：

- 实物　汽车、遥测数据、压力传感器；
- 角色　母亲、教师、政治家；
- 事件　着陆、中断、请求；
- 交互　借贷、会议、相交。

从数据库建模的角度，Ross 提出了类似的列表[33]：

- 人　执行某项功能的人；
- 地点　人或物相关的区域；
- 物　实实在在的物理对象或对象的分组；
- 组织　正式组织起来的人、资源、机制和功能的集合，具有确定的任务，它的存在基本上与个人无关；

- 概念　不可触摸的原则或思想，用于组织或跟踪业务活动和沟通；
- 事件　发生的事情，通常是在某个日期和时间针对另外的事物发生的，或者是一个序列中的步骤。

Coad 和 Yourdon 提出了另一组可能对象的来源[34]：

- 结构　“是一种”或“组成部分”关系；
- 其他系统　与应用进行交互的外部系统；
- 设备　与应用进行交互的设备；
- 要记住的事件　必须记录的历史事件；
- 扮演的角色　用户在与应用的交互中扮演的不同角色；
- 位置　对应用来说有重要意义的物理位置、办公室和地点；
- 组织机构单位　用户属于的群体。

在最高的抽象层上，Coad 引入了主题领域的概念，它主是要与某个高层系统功能相关的类的逻辑分组。

2. 行为分析

这些经典方法关注问题领域中实实在在的事物，但面向对象分析的另一种思路是关注动态的行为，将这些行为作为类和对象的主要来源[1]。这类方法更像是概念聚集，根据一组展示出类似行为的对象来形成类。

例如，Wirfs-Brock、Wilkerson 和 Wiener 强调了责任。责任代表的是“对象维护的知识和可以执行的动作，其意义在于表达一个对象的目的以及它在系统中的位置。对象的责任是针对它支持的所有契约而提供的全部服务”[36]。在这种方式中，我们将具有共同责任的事物划分为一组，让超类包含一般的责任，让子类包含特殊的行为，从而构成类的层次结构。

Rubin 和 Goldberg 提出了一种方法，根据系统的功能来确定类和对象。他们指出，“我们所使用的方法强调了对系统中发生的事情的最基本的理解，它们是系统的行为。接下来我们将这些行为分配到系统的各个部分中，并试图理解谁发起了这些活动，谁参与了这些活动……扮演重要角色的发起者和参与者被确定为对象，我们针对这些角色，将行为责任分配到这些对象上”[37]。

Rubin 关于系统行为的概念与功能点的思想有着密切的关系。功能点是 Albrech 在 1979 年首次提出的，它是“作为最终用户的一项业务功能而定义的”[38]。功能点代表了某种输出、查询、输入、文件或接口。尽管这个定义的起源是信息系统，但功能点的思想却适用于所有类型的自动化系统——功能点是系统外部可见的、可测试的行为。

3. 领域分析

到目前为止，我们所讨论的原则通常适用于开发单个具体的应用。但是，领域分析试图确定在某个领域中所有应用都通用的类和对象，如病历跟踪、证券交易、编译器或导弹电子系统。如果在设计中对于已存在的关键抽象有点疑惑，领域分析师可以提供帮助，指出这些

[1] Shlaer 和 Mellor 扩展了他们早期的工作，也关注了行为。具体来说，他们研究了每个对象的生命周期，以此作为理解边界的手段[35]。

关键抽象在其他相关系统中已被证明有用。领域分析效果挺好，因为除了极特殊的情况之外，软件系统很少有真正独特的需求。

领域分析的思想首先是由 Neighbors 提出的。我们将领域分析定义为“确定对象、操作和关系的尝试，领域专家认为这些对象、操作和关系对这个领域很重要”[39]。Moore 和 Bailin 提出了下列领域分析的步骤：

- 咨询领域专家，构建一个通用的模型草稿；
- 检查领域中原有的系统，以一种通用的格式展示出这方面的理解；
- 咨询领域专家，确定系统间的相似和差异；
- 精化通用模型，以包含原有的系统。[40]

领域分析可以应用于许多类似的应用（垂直领域分析），也可以应用于同一应用的相关部分（水平领域分析）。例如，当我们开始设计一个新的病患监控系统时，理所当然应该调查已有系统的架构，理解以前使用了哪些抽象和机制，评估哪些对我们有用，哪些对我们没有用。类似地，账务系统必须提供许多不同类型的报表。通过将同一应用中的这些报表作为一个单独的领域来考虑，领域分析师可以让开发者理解服务于不同类型报表的关键抽象和机制。由此得到的类和对象反映了一组关键抽象和机制，这些抽象和机制是对具体报表生成问题的泛化，与单独分析和设计每个报告相比，这样得到的设计可能更简单。

到底谁是领域专家？通常，领域专家就是一个用户，如铁路系统中的一名列车工程师或调度员，或者医院中的一名护士或医生。领域专家通常不会是一名软件开发者，更常见的情况是，他只是非常熟悉某个问题的方方面面。领域专家说话时使用的是问题域的词汇。

某些经理对于让开发者和最终用户直接沟通可能存在一些顾虑（某些经理的做法更让人吃惊，他们甚至不让最终用户看到一名开发者）。对于极为复杂的系统来说，领域分析可能是一个正规的过程，用到多个领域专家和开发者的资源，并持续几个月的时间。并非所有的项目都需要这样正式的分析，特别是那些较小的项目。通常，清除设计问题要做的就是让一名领域专家和一名架构师或开发者进行简短的会晤。我们会很吃惊地看到，一点点领域知识都能够对明智的设计决策起到很大的作用。确实，我们发现在设计系统过程中举行多次这样的会议是很有用的。领域分析很少是一项独立的活动，如果我们更有意识地分析、设计，就能更好地关注领域分析。

4. 用例分析

孤立来看，经典分析、行为分析和领域分析的实践都非常依赖于分析师的个人经验。对于大多数开发项目来说，这是不可接受的，因为这样的过程既不是确定的，也不能预测其成功与否。

但是，有一种实践可以与前面三种方法结合使用，以一种有意义的方式来驱动分析的过程。这种实践就是用例分析，它是由 Jacobson 首先正式提出的。Jacobson 等人将用例定义为“一个执行者通过与系统进行对话的方式执行的一个行为上相关的事务序列，目的是为执行者提供某种可度量的价值”[41]。

简单来说，我们可以在需求分析时就应用用例分析，此时，最终用户、其他领域专家和开发团队列出系统操作的基本场景（不需要在一开始就细化这些场景，只要列出它们就行了）。

这些场景共同描述了这个应用的系统功能。分析随后研究每个场景，可能使用与影视行业的做法类似的故事板技术[42]。当分析团队分析每个场景时，他们必须确定参与该场景的对象、每个对象的责任以及这些对象与其他对象协作的方式，即每个对象调用其他对象的操作。通过这种方式，分析团队被迫在所有抽象之间形成清晰的分离关注。随着开发过程的继续，这些初始的场景会被扩充，作为次级系统行为的异常条件。这些次级场景引入了新的抽象，或者添加、修改、重新分配了原有抽象的责任。场景也被作为系统测试的基础。

5. CRC 卡

CRC 卡是作为一种简单而奇迹般有效的场景分析方式而出现的[1]。CRC 卡是 Beck 和 Cunningham 最先提出的，它是一种教授面向对象编程的工具[44]。CRC 卡已被证明是一种有用的开发工具，促进了开发者之间的头脑风暴，增进了沟通。CRC 卡就是一张 3×5 的索引卡片[2]，分析师在上面用铅笔写下类的名称（在卡的顶部）、它的责任（在卡的一半）以及它的协作者（在卡的另一半）。针对与场景有关的每个类，我们创建一张卡片。当团队成员分析该场景时，他们可能会为已有的类分配新的责任，将某些责任集中起来形成一个新的类，或者（更常见的情况）将一个类的责任分解到粒度更小的类中，也许会将这些责任放到其他的类上。

CRC 卡片可以被摆放在不同的位置，以代表协作的模式。从场景的动态语义上来看，卡片可以按一定顺序摆放，展示每个类的原型实例之间的消息流。从场景的静态语义上来看，卡片可以按一定顺序摆放，展示类之间的泛化/特化或聚合层次结构。

6. 非正式英语描述

对于经典的面向对象分析，有一种基本的替代方法，它是由 Abbott 首先提出来的。Abbot 建议写下问题的英语描述，然后划出名词和动词[45]。名词代表了候选对象，动词代表了这些对象上的候选操作。

Abbott 的方法是有用的，因为它很简单，并且迫使开发者研究问题空间的词汇表。但是，这并不是一种严格的方法，并且它在处理不太小的问题时的可伸缩性不好。人类的语言是一种非常不精确的表达方式，所以利用这种方法得到的对象和操作的品质将取决于作者的写作技巧。而且，名词可以动词化，动词也可以名词化，所以人们很容易弄乱候选列表，片面强调对象或操作。

7. 结构化分析

某些组织机构尝试使用结构化分析的产品作为面向对象设计的前端。这种技术吸引人的原因只是因为大量的分析师熟悉结构化分析，而且有一些计算机辅助软件工程（CASE）工具可以支持这些方法的自动化。从个人的观点来说，我们不鼓励将结构化分析作为面向对象设计的前端。

这种方法从系统的一个基本模型开始，由数据流图和结构化分析的其他产品来描述。这些图为问题提供了相当正式的模型。从这个模型出发，我们可以通过三种方式来确定问题域

---

[1] CRC 是 Class/Responsibilities/Collaborators 的缩写。

[2] 如果软件开发预算允许的话，就购买 5×7 的卡片。带划线的卡片不错，喷涂着色彩的卡片可以表明你是一位很酷的开发者。

中有意义的类和对象。

McMenamin 和 Palmer 提出从数据字典的分析开始，接下来分析模型的上下文背景图。他们说，“利用你的基本数据元素清单，考虑它们告诉了你什么信息或者它们描述了什么。例如，如果它们在句子中是形容词，那么它们修饰了什么名词？对这个问题的回答将形成候选对象的列表”[47]。这些候选对象通常来自于周围的环境、基本输入和输出以及这个系统所管理的产品、服务和其他资源。

另两种方式涉及对数据流图的分析。对于特定的数据流图（使用 Ward 和 Mellow[48]的术语定义），侯选对象可以从以下来源推出：

- 外部实体；
- 数据存储；
- 控制存储；
- 控制转换。

候选类有两个来源：

- 数据流；
- 控制流。

这就为我们留下了数据转换，我们要么将数据转换作为已有对象上的操作，要么将其作为新对象的行为，该新对象承担了这种转换的代理责任。

Seidewitz 和 Stark 提出了另一种技术，他们称之为“抽象分析”。抽象分析关注确定中心实体，它在本质上与结构化设计中的中心转换类似。他们说，“在结构化分析中，输入和输出数据被检查和跟踪，直到它们达到最高的抽象级别。在输入和输出之间的这些过程构成了中心转换。在抽象分析中，设计者所做的事是一样的，但也会检查中心转换，以确定哪些过程和状态代表了系统行为的最佳抽象模型”[49]。在某个数据流图中确定了中心实体之后，通过跟踪进入和离开中心实体的数据流，并对遇到的过程和状态进行分组，抽象分析继续确定所有的支持实体。在实践中，Seidewitz 和 Stark 发现抽象分析是一项很难成功应用的技术，作为替代，他们推荐使用面向对象的分析方法[50]。

我们必须非常强调这一点，即结构化设计通常是与结构化分析一起使用的，它完全与面向对象设计的原则不同。经验表明，如果开发者不能拒绝回到结构化设计的思维中，利用结构化分析作为面向对象设计的前端这种方法常常会失败。另一个很真实的危险就是，许多分析师趋向于用数据流图来反映设计，而不是反映问题域的基本模型。我们很难从一个明显基于算法分析的模型中创建一个面向对象的系统。这就是我们喜欢用面向对象分析作为面向对象设计的前端的原因：这降低了先验的算法表示法污染设计的风险。

如果必须使用结构化的分析作为前端，不管出于哪种光明正大的原因[1]，我们建议你在数据流图开始有点像设计而不是基本模型时，立即停止画数据流图。另外，当设计全面展开时，抛开结构化分析的产品也是一种很好的做法。要记住，包括数据流图在内的所有开发的产品，

[1] 政治斗争原因和历史原因显然不是光明正大的原因。

本身并不是终点。它们应该被看作是一些工具，帮助开发者理解问题及其实现方式。人们通常画出一张数据流图，然后发明一些机制来实现期望的行为。从实践上来说，正是设计的活动改变了开发者对问题的理解。因此，只有抽象层次足够高的结构化分析的产品才应该被保留下来。它们记录了问题的基本模型，可以从它们导出任意多种不同的设计。

4.3 关键抽象与机制

“关键抽象”是一个类或对象，它是问题域词汇表的一部分。确定这样的抽象的主要价值在于，它们给出了问题的边界，突出了系统中的事物——这些事物与我们的设计有关；同时，它们排除了系统之外的事物，这些事物是不必要的。

在前面一章中，我们使用了术语“机制”来描述对象协作的结构，这些对象协作结构提供了某种行为，满足了问题的一项需求。一个类的设计包含了单个对象如何行为的知识，而一种机制则是关于一组对象如何协作的设计决策。因此，机制代表了行为的模式。

接下来我们将讨论如何确定并精化这些关键抽象和机制。

4.3.1 确定关键抽象

确定关键抽象是与具体领域高度相关的。Goldberg 说：“正确选择对象当然取决于应用的目的，以及要操作的信息的粒度。”[51]

前面曾提到，确定关键抽象包含两个过程：发现和发明。通过发现过程，我们意识到领域专家所使用的抽象。如果领域专家提及它，那么这个抽象通常很重要[52]。通过发明过程，我们创造了新的类和对象，它们不一定是问题域的组成部分，但在设计或实现中也是很重要的。例如，使用 ATM 的客户提到账户、取款和存款，这些词是问题域词汇表的一部分。这种系统的开发者会使用这些抽象，但也必须引入新的抽象，如数据库、屏幕管理器、列表、队列等。这些关键抽象是具体设计的结果，不属于问题域。

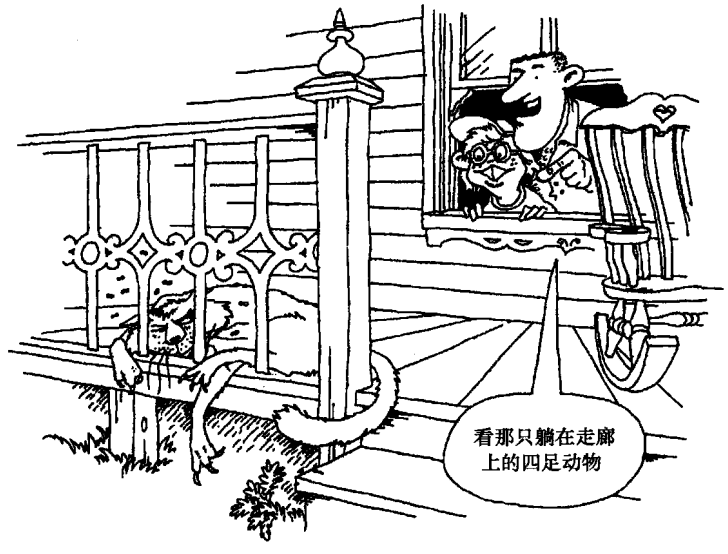
1. 精化关键抽象

当我们将某个关键抽象确定为候选者时，必须根据前一章中提出的测量指标来评估它。Stroustrup 指出，“通常这意味着程序员必须关注以下问题：这个类的对象是如何创建的？这个类的对象可以被复制或销毁吗？在这样的对象上可以执行什么操作？如果对这些问题没有好的答案，这个概念可能在开始时并不‘清晰’，也许最好是再仔细考虑一下这个问题和提出的解决方案，而不是立即开始针对问题进行‘编码’”[53]。

对于一个新的抽象，我们必须将它放在已经设计好的类和对象层次结构的上下文中。实际上，这既不是自顶向下的活动，也不是自底向上的活动。Halbert 和 O’Brien 指出，“当在一个类型层次结构中设计类时，并非总是从超类开始，然后创建子类。通常，会创建一些看起来不相似的类型，意识到它们是相关的，然后将它们的共同特点分离出来，放到一个或多个超类中……通常需要这样上上下下做几次才能得到一个完整和正确的程序设计”[54]。这

并不是乱来的借口，而是基于经验的一种判断，即面向对象设计是增量式、迭代式的。Stroustrup 也说过类似的话，他说，“最常见的类层次结构的组织方式是从两个类中提取公共部分放到一个新类中，或者将一个类分成两个新类”[55]。

将类和对象放到正确的抽象层次中是很难的。有时候我们可能发现一个通用的子类，并将它在类结构中上移，从而增加共享度，这被称为“类提升”[56]。类似地，我们可能发现一个类太一般化，因此从它派生出一个子类很困难，因为语义上的差距太大，这被称为“粒度规模冲突”[57]。不论是哪种情况，我们追求的都是确定高内聚、低耦合的抽象，这样就能缓解这两种情况。



类和对象应该处于正确的抽象层次——不太高也不太低

2. 命名关键抽象

正确地为事物命名（以便让名称能反映其语义）常常被许多开发者忽略，但这对于记录我们所描述的抽象的本质是很重要的。编写软件应该像英文散文那样认真，既要考虑到阅读者，也要考虑到计算机[58]。考虑一下我们在确定一个对象时可能需要的所有名称：对象本身的名称、它的类的名称、该类声明的模块的名称。将这些名称与数千个对象和数百个类相乘，你面临的是一个非常真实的问题。

我们给出以下建议：

- 对象应该用合适的名词词组来命名，如 theSensor 或简单的 shape；
- 类应该用常见的名词词组来命名，如 Sensor 或 Shape；
- 如果可能，选择的名称应该是领域专家使用和认识的名称；
- 修改操作应该用一个主动语态的动词词组来命名，如 draw 或 moveLeft；
- 选择操作应该表示出查询的意思，或者用 be 动词的形式来命名，如 extentOf 或

isOpen；

- 使用下划线和大写字母主要是个人的喜好。但不论采用哪种风格，至少在程序内要保持一致。

4.3.2 识别机制

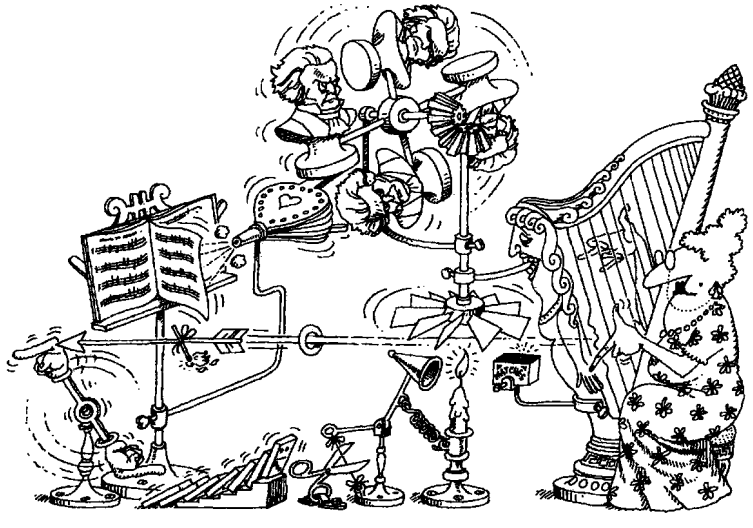
考虑一辆汽车的系统需求：踩下加速踏板将导致引擎转得更快，放掉加速踏板将导致引擎转得更慢。驾驶员完全不关心这到底是如何发生的。只要能够提供这种行为，我们可以采用任何机制，所以选择哪一种机制主要是设计时的选择。具体来说，下面几种设计都可以考虑。

- 一个机械连接装置，直接将加速踏板和燃油喷射器连接起来。
- 一个电子装置，连接加速踏板下的压力传感器和计算机，由计算机控制燃油喷射器（一种电线控制的机制）。
- 没有连接装置。油箱放在汽车的顶部，利用重力让燃油流向引擎。燃油的流速由油路上的一个夹子来调节，踩下加速踏板将放松夹子上的压力，导致燃油流得更快（一种低成本的机制）。

开发者从一种可选机制中选择哪一种通常是考虑其他因素的结果，如成本、可靠性、可生产性和安全性。

一个客户对象违反另一个对象的接口是不对的，同样，对象的行为超出某种机制打算提供的行为也是不能忍受的。实际上，如果踩下加速踏板却没有让引擎加速，而是打开了车灯，驾驶员会很吃惊的。

关键抽象反映了问题域的抽象，而机制是设计的灵魂。在设计过程中，开发者不仅必须考虑单个类的设计，还要考虑这些类的实例如何一起工作。同样，我们使用场景来驱动这个分析过程。



机制是对象协作提供某些高层行为的方式

当开发者决定采用某种协作模式之后，工作会被分解到许多对象上，即在相应的类上定义适当的方法。归根到底，单个类的协议包含了实现所有行为以及实现与其实例相关的所有机制所需要的全部操作。

因此，与类层次结构的设计一样，机制代表了战略设计决策。相比之下，单个类的接口设计更多的是一种战术决策。这些战略决策必须被明确表示出来，否则，我们将得到一些互相无法合作的类，所有类都努力完成自己的工作，很少考虑其他对象。最优雅、最精益、最快的程序会包含一些精心设计的机制。

1. 机制即模式

机制实际上是我们在构造良好的软件系统中发现的各种模式。在食物链的最底端，我们常用惯用法。惯用法是一种表示方式，它是某种编程语言或应用文化所特有的，代表了使用这种语言时被大家广泛接受的习惯[1]。例如，在 CLOS 中，没有程序员会在函数名或变量名中使用下划线，但在 Ada 中，这是很常见的[59]。学习一种编程语言的一部分工作就是要学习其惯用法，这通常是在程序员之间口口相传的。但是，Coplient 指出，惯用法在编写低层模式时是很重要的。他说，“许多常见的编程任务都有惯用的实现方式”，因此确定这样的惯用法让我们能够“利用 C++的结构来表达语言之外的功能，并制造出这就是语言的一部分的假象”[60]。

虽然惯用法是编程文化的一部分，但在食物链的高端，我们有框架。框架是一组类，为某个领域提供了一组服务。所以框架输出了一些类和机制，客户程序可以使用它们或者扩展它们。框架代表了大规模的复用。它们通常是商业供应商的产品，如微软公司的.NET 框架，或者是开放源代码的工作，如 Apache 软件基金会的 Struts 框架和 JUnit 框架（Erich Gamma 和 Kent Beck），以及许多其他框架。

2. 机制示例

考虑在图形用户界面中常常用到的画图机制。几个对象必须通过协作为用户展现一张图片：窗口、视图、被看到的模型以及知道何时（但不知道如何）显示这个模型的某个客户。客户首先告诉窗口画出它自己。由于它可能包含几个子视图，窗口接下来会告诉它的每个子视图画出自己。每个子视图会告诉它的模型画出自己，最终导致图像展现在用户面前。在这个机制中，模型完全与展示它的窗口和视图解除了耦合。这就是模型—视图—控制器模式（MVC 模式）。类似的机制几乎在所有面向对象的图形用户界面框架中都得到使用。

所以，机制代表了一种层次的复用，它高于单个类的复用。例如，MVC 模式在 Smalltalk 用户界面中得到广泛使用。MVC 模式又是建立在另一个模式的基础上的，即依赖模式。它被包含在 Smalltalk 的基本类 Model 的行为中，因此被许多 Smalltalk 类库所使用。

机制和模式的例子几乎可以在所有领域中找到。例如，在最高的抽象层次上，操作系统的结构可以根据调度程序所使用的机制来描述。在人工智能领域，人们使用各种机制来探索推理系统的设计。最常使用的模式是黑板机制，在这种机制中，不同的知识源分别更新黑板。

[1] 惯用法的一种确定的特征就是，忽略或违反这种惯用法马上就会导致一些社交方面的问题——会被贴上土人或外人的标签，不值得尊敬。

这样的机制没有中央控制，但对黑板的任何改动都会触发一个代理，去探索新的问题解决途径[63]。Coad 通过类似的方式确定了一些面向对象系统中的常见机制，包括时间关联的模式、事件日志的模式和广播的模式。在每一种情况下，这些机制的实现都不是单个的类，而是一组协作的类。

至此，我们结束了对分类以及面向对象设计基础概念的讨论。接下来的 3 章中将关注表示法、过程和实践。

4.4 小结

- 确定类和对象是面向对象分析和设计中的基本问题，确定过程包括发现和发明。
- 分类本质上是分组问题。
- 分类是一个增量、迭代的过程。它很难，其难处体现在给定的一组对象可以按几种同样正确的方式来分类。
- 分类的三种方式，包括经典分类（按属性分类）、概念聚集（按概念分类）和原型理论（按照与原型的关系来分类）。
- 场景是强大的面向对象分析工具，可以用于经典分析、行为分析、领域分析和用例分析。
- 关键抽象反映了问题域的词汇表，可以从问题域中发现，也可以作为设计的一部分而发明。
- 机制代表了战略上的设计决策，考虑的是许多不同类型的对象的协作活动。

方法

哪一种创新会走向成功的设计，哪一种会走向失败，这完全是不可预测的。每次设计新东西的机会，不论是桥梁、飞机，还是摩天大楼，工程师都面临着无数的选择。工程师可能决定尽可能多地从已有的设计中拷贝那些看起来不错特征，这些特征成功地承受了人和自然的考验。但是，他也可能决定对以前设计中的这些方面进行改进，这些改进似乎是人们期望的。

——Henry Petrosky
To Engineer Is Human

对于任何要在市场上取得成功的技术来说，某些事情一定要发生。需要积累相当数量的用户，他们证明成功地使用了该技术，而这将吸引其他人在这个技术领域进行投资。要积累相当数量的用户，共同的语言相当有用，这样就能够容易地教授、交换和传播这个技术领域的知识。

为了让这种技术在主流市场中有良好的表现，要传播的一种关键知识就是这种技术如何能够成功地使用或形成（即过程是怎样的），以及如何能够有效、高效地做到这一点。这就是本部分关注的重点：一种共同而标准的语言（统一建模语言）、一个过程以及面向对象分析和设计的实践。

第 5 章

表示法

画图的动作并不是分析或设计的要点。一张图示只是记录了系统行为的一种说明（对于分析来说）或一种架构的愿景与细节（对于设计来说）。如果遵循任何一种工程师的工作方式（软件、土木、机械、化工、建筑或其他），你很快就会意识到系统构想形成的唯一场所就是设计者的头脑。当这种设计随时间展开时，它常常被记录在一些高科技的媒质上，如同被记录白板、餐巾纸或信封的背面一样[1]。

5.1 统一建模语言

拥有一种定义良好的、富有表现力的表示法对于软件开发过程是很重要的。首先，标准的表示法让分析师或开发者能够描述一个场景，阐明一种架构，然后无二义地将这些决定告诉别人。画一张电路图，其晶体管的符号应基本上可以被世界上任何一个电子工程师看懂。类似地，如果一个在纽约的建筑师画出了一幢房屋的草图，在旧金山的建筑工人基本上能够毫无困难地理解哪里放门、哪里放窗、哪里放电器，只要图纸包含这些细节。其次，正如 Whitehead 在他的数学名著中所说的，“好的表示法消除了大脑的不必要工作，让大脑能够集中考虑更高级的问题”[2]。最后，一种富有表现力的表示法能够利用自动化的工具，消除关于这些决定的许多繁琐的一致性和正确性检查。美国国防科学委员会的一份报告指出：“软件开发是并且永远将是一种劳动力密集型的技术……虽然机器能够做一些辛苦而乏味的工作并让我们能够记录下我们心中的构想，但概念开发仍是典型的人类活动……软件开发中永远不会消失的部分就是打造概念结构，可以消失的部分是表达这些概念结构所需的工作量。”[3]

5.1.1 简单历史回顾

统一建模语言（UML）是分析、说明和设计软件系统的主要建模语言。随着面向对象编程语言在软件业中的出现，如第 2 章中所述，面向对象方法学也开始出现了。从 20 世纪 80

年代后期到90年代初期，出现了无数的方法学，之后人们对其进行了修改和提炼。这些方法学中的许多都在某些方面很强，在另一些方面较弱。这让方法学家们能够将其他方法学中有用的东西加到自己的方法学中。这反映了面向对象的实践者在真实世界中所做的事情。实践者可能主要采用某种方法学，如果出现了其他有用的思想，他们就会将这些思想融入每天的工作中。

在20世纪90年代中期，Booch、Rumbaugh和Jacobson加入了Rational软件公司，开始将他们各自的方法学融合在一起，创造出UML的第一个版本。然后，他们开始与其他方法学家和公司一起工作，向对象管理组织组织（OMG）提交一种标准的建模语言。OMG是一个协会，其目标是为计算机界创建标准并进行维护。在1997年11月，OMG采用了UML作为标准。自那时起，OMG就承担了UML的组织管理和继续开发的责任。

UML自成为标准后，有过一些版本。本书讨论的是UML 2.0。许多书籍都已经详细介绍了UML开发的历史，要了解更多的信息，请参见附录B。

5.1.2 模型与多重视图

许多其他行业（如电子、化工、建筑、音乐等）都有特有的表示法，表示人们创造出的制品。同样，人们用UML对构建的系统进行建模（即表示）。总的来说，所构建的UML模型将以一定的保真度展现要构建的真实系统。但是，不可能在一张大图上记录一个复杂软件系统的所有细节。UML有几种不同类型的图，每一种都提供了系统的某一种视图。Kleyn和Gringrich指出："开发者必须理解对象所涉及的结构和功能。开发者必须理解类对象的分类结构、使用的继承机制、对象独立的行为以及整体系统的动态行为。这个问题有点类似观看网球或足球这样的体育赛事——每个摄像机都揭示了动作的一个方面，这不能由一个摄像机独立完成。"[4]

例如，考虑由数百个类组成的应用。不可能制作一张图来展示所有这些类和它们之间的关系，实际上也不需要这样做。我们会使用几张类图，每张图都展示模型的一个视图。有一张图可能展示了某些关键类的继承关系，另一张图可能展示了某个类用到的所有类的传递闭包。当模型稳定下来的时候（即所谓的"稳定状态"），所有图和模型之间都保持语义上的一致。例如，如果在某一次交互中（我们在一张对象图中描述），对象A将消息M传递给对象B，那么M必须在B类中有直接或间接的定义。在对应用的类图中，类A和类B之间必须有相应的关系，这样，A类的实例实际上就可以在B类的实例上调用消息M。

在所有的图中，具有相同名称的所有实体都被认为是同一个模型项的引用。例如，如果类C出现在了两张不同的图中，它们指的都是同一个类C。这一规则对操作可能有例外，因为操作的名称可能被重载。

5.1.3 图分类

UML图可以分成两大类：结构图和行为图（参见图5-1）。这种分类方式与第1章中的复杂性讨论是一致的。系统的复杂性既来自于系统中元素的数量和组织（即结构），也来自于

这些元素协作完成其功能的方式（即行为）。

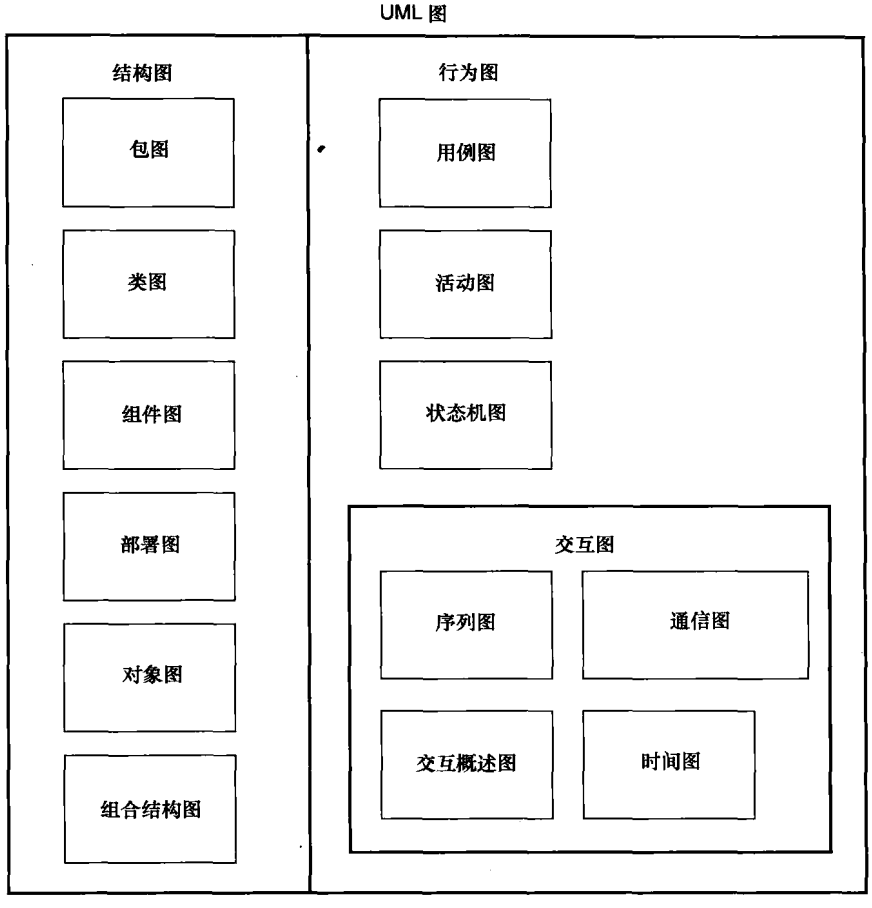


图 5-1 UML 图

1. 结构图

这些图用于展示系统中元素的静态结构。它们描述系统的架构组织、系统的物理元素、系统的运行时刻配置和业务中领域相关的元素等。UML 结构图包括：

- 包图；
- 类图；
- 组件图；
- 部署图；
- 对象图；
- 组合结构图。

结构图常常与行为图一起使用，描述系统的某个方面。每个类可能有一个相关的状态机图，描述该类的实例的事件驱动行为。类似地，与表示场景的对象图一起，我们可以提供一

张交互图，在评估时展示消息的时间或事件次序。

2. 行为图

到目前为止，我们介绍的图基本上都是静态的。但是，在所有的软件密集型系统中，事件都是动态发生的：对象被创建和销毁，对象之间按次序发出消息，而且在某些系统中，外部的事件会触发某些对象上的操作。不奇怪，在一张纸这样的静态媒质上描述动态行为是个难题，但基本上每个科学领域都会遇到这个问题。在面向对象开发中，我们利用下面一些图来描述问题的动态行为语义或它的实现：

- 用例图
- 活动图
- 状态机图
- 交互图
  - 序列图
  - 通信图
  - 交互概述图
  - 时间图

在本章的后面，我们将介绍构成这些图的 UML 元素。

5.1.4 在实践中使用图

UML 是一份详细的规范，但这并不意味着每次都要用到它的所有方面。实际上，这种表示法的一个子集足以表达大多数分析和设计问题中的语义。我们将在本章的表示法介绍中重点突出这个子集。那么，为什么需要这个子集之外的所有细节？答案很简单，这样的细节在表达某些重要的策略决定时是需要的（我们在本书的应用部分将展示这一点）。另外，有些细节存在于 UML 的基础结构中，工具提供商对它们感兴趣，这些细节有助于创建正向工程和逆向工程工具。这些内部的细节使得支持这种表示法的前端 CASE 工具能够与软件开发环境集成，集成环境关注的焦点在面向对象语言上。

Winberg 曾指出：在其他设计领域，例如建筑，草图是最常用到的图形手段，在创建性的设计工作完成之前，很少画出精确而详细的图纸。[5]要记住，表示法只是记录系统行为和架构分析的工具，表示法本身并不是目的。因此，应该只使用那些表达意思所必需的表示法元素，而不是其他的东西。正如过度定义需求很危险一样，过度定义问题的解决方案也是很危险的。例如，在建筑蓝图上，建筑师可能指出房间中电灯开关的一般位置，但是要到在房间装修之后，施工经理和业主完成了布线工作之后才能确定开关的准确位置。在建筑蓝图上指定电灯开关的准确三维位置是愚蠢的行为（当然，除非这种细节对于业主很重要，也许业主一家比平均身高矮很多或高很多）。因此，如果一个较小的软件系统的分析师、设计师和实现者的技术很好，而且已经建立起了密切的工作关系，较粗的草图就够了（虽然仍有必要为系统的维护者留下一份架构视图）。在实践中，很少会是这种情况。另一方面，如果系统很大，软件的部分很多，或者实现者的技术不是那么好，或者开发者被地理位置、时间或合约所分

隔，在开发的过程中就需要更多的细节。

5.1.5 概念模型、逻辑模型和物理模型

随着系统开发的推进和成熟，你的系统模式可能代表了不同的细节层次。概念模型记录了系统中存在（或将存在）的领域实体以及它们与系统中其他领域实体的关系。概念层的建模是利用业务领域的术语来完成的，应该是技术无关的。系统的逻辑视图利用了概念模型中创造的概念，建立起关键抽象和机制的意义，并确定系统的架构和整体设计。系统的物理模型描述了系统实现的具体软件和硬件构成。显然，物理模型是技术相关的。

对于某个项目来说，随着时间的推移，系统的设计将从概念成熟发展到逻辑成熟，最后到物理成熟。在开发生命周期的不同阶段，会使用不同的图。某些图只在项目开发生命周期的早期使用。根据创建的系统的类型，用法也会不一样。例如，一个券商的投资交易系统会使用更多的状态机图和时间图，而简单的支票本应用就不会这样。

对于某个项目来说，分析和设计的产物是通过这些模型来表示的。总地来说，这些不同模型的语义是丰富的：它们的表达能力足够强，让开发者能够记录系统分析和架构阐述时所有感兴趣的战略和战术决策；它们也足够完整，可以作为大多数面向对象语言实现的蓝图。

5.1.6 工具的角色

本章中介绍的表示法可以手工使用，虽然对于大型的应用来说，最好是使用自动化的工具来支持。对于任何表示法的自动化支持来说，工具能做的一件事就是使不好的设计者更快地创造出糟糕的设计。伟大的设计来自于伟大的设计者，而不是来自于伟大的工具。工具只是增强了个人的能力，让人们能够专注于分析或设计中真正有创造性的方面。因此，有一些事情工具可以做得很好，还有一些事情工具根本不能做。

工具可以提供一致性检查、约束条件检查、完整性检查和分析，它们可以帮助开发者以不受限制的方式浏览正在创建的分析和设计的产物。例如，当我们看一张组件图时，开发者可能想研究一下某一种机制。他可以利用工具来定位某个组件包含的所有类。在看描述一个场景的序列图时，开发者可能想看看继承的结构。如果这个场景包含一个主动对象，开发者可以利用一个工具来找到执行这个控制线程的处理器，然后查看这类的状态机在该处理器上的模拟执行过程。通过这些工具，开发者不必保存所有这些繁琐的细节，从而可以将关注的重点放在开发过程中有创造性的部分上。

从另一方面来说，工具不能告诉我们应该创建一个新的类，从而简化类的结构。这种工作需要人类的洞察力。可以考虑使用某种专家系统作为工具，但这要求：使用者既是面向对象开发的专家，也是问题域的专家；设计分类继承关系的能力以及大量的常识。我们不能指望近期会出现这样全知全能的工具，可以取代设计者，同时，我们又要创造真正的系统。

5.1.7 面向对象开发的产品

通常，系统分析将得到一组用例图和活动图（通过场景来表示系统的行为）、类图（表

示代理的角色和责任，这些代理提供了系统的行为）及交互图和状态机图（展示这些代理的事件次序行为）。类似地，系统的架构可能包括几组包图、类图、对象图、组件图和部署图，以及它们对应的动态视图。

这些图中包含了点到点的联系，允许我们从实现回溯到规格说明书，跟踪需求。从部署图开始，一个节点可能承载了一个工件，该工件实现了组件图中定义的一个组件。这个组件图可能包含一组类定义，这些类可以在相应的类图中找到。最后，单个类的定义指向了用例和需求，因为一般来说，这些类直接反映了问题空间的词汇表。

5.1.8 规模上的伸缩

我们发现，UML 既适用于只包含几十个类的小系统，也适合包含几千个类的大系统。后面两章中将会谈到，这种表示法特别适合增量、迭代的开发方式。开发者不会创建一个图之后就把它扔到一边，把它作为某种神圣、不变的工件。相反，这些图会随着设计过程而演进，新的设计决策不断出现，细节不断添加。

我们还发现，这种表示法基本上是语言无关的。它适用于大量的面向对象编程语言。

5.1.9 UML 的语法和语义

本章的其余部分将针对面向对象分析来设计和描述 UML 的语法和语义。利用第 2 章中介绍的溶液栽培种植园系统的问题，我们将提供这种表示法的一些小例子。本章不会解释 UML 图开发的过程，这是第 6 章的主题。

为了反映出它们之间的关系，我们按照通常可能的开发次序来介绍 UML 2.0 的图。我们相信，这比先介绍所有的结构图再介绍所有的行为图更好。具体地说，我们将以下面的次序来介绍这些图：

- 包图；
- 组件图；
- 部署图；
- 用例图；
- 活动图；
- 类图；
- 序列图；
- 交互概述图；
- 组合结构图；
- 状态机图；
- 时间图；
- 对象图；
- 通信图。

在本书的第 3 部分中，每一章都将介绍一个不同类型的应用，并将重点介绍在我们描述

的整体生命周期中，最适合这个应用的一组图。

5.1.10 UML 2.0 信息资源

UML 2.0 表示法具有相当的可扩展性和复杂性，OMG UML 2.0 规范的一篇评论明确地指出了这一点。有效地应用这一规范需要查看其他的资源，如本章的内容。如果读者在阅读了 OMG UML 2.0 规范和本章之后，需要进一步的解释或详细信息，我们推荐 *The Unified Modeling Language Reference Manual, Second Edition*。附录 B 列出了其他 UML 2.0 资源。

5.2 包图

在进行面向对象分析和设计时，需要组织开发过程的工件，从而清晰地展现出问题域的分析和相关的设计。具体的原因会有不同，但主要集中在可视模型本身的物理结构上，或者是希望通过多重视图来清晰地展现模型元素。组织 OOAD 工件的好处有以下几点[42]：

- 在复杂系统开发中提供清晰性和可理解性；
- 支持多用户使用的并发模型；
- 支持版本控制；
- 在多个层次提供抽象——从系统到组件中的类；
- 提供封装和包容，支持模块化。

实现这种组织的主要方式是利用 UML 包图，它提供了表现 UML 元素分组的能力。

包图的主要元素是包、它们的可见性和它们的依赖关系。

5.2.1 基本概念：包表示法

UML 的包是包图中使用的两种主要表示法之一。另一种是依赖关系，我们稍后会介绍。包的表示法是一个左上角带有标签的矩形。UML 2.0 规定，如果包中不包含 UML 元素，包的名称应该被放在矩形之内。如果它包含一些元素，名称就应该被放在标签之内。图 5-2 展示了特定工具实现的命名建议，它提供了 HydroponicsGardeningSystem 包的黑盒视图，没有显示其中包含的元素[45, 46]。

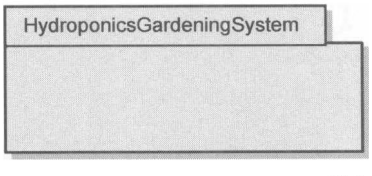


图 5-2 HydroponicsGardeningSystem 的包表示法

如果因为有一些元素存在，或者因为我们的关注，需要显示少量的元素，可以使用合适

的表示法（包、用例、类、组件等）在包内显示这些组成部分。图 5-3 也显示了 HydroponicsGardeningSystem 包，但它还包含了另外两个包。在左边的表示方法中，我们将 Planning 和 Greenhouse 包作为被物理包含的包在 HydroponicsGardeningSystem 包之内进行显示。右边显示了另一种包容关系的表示法[47, 48]。

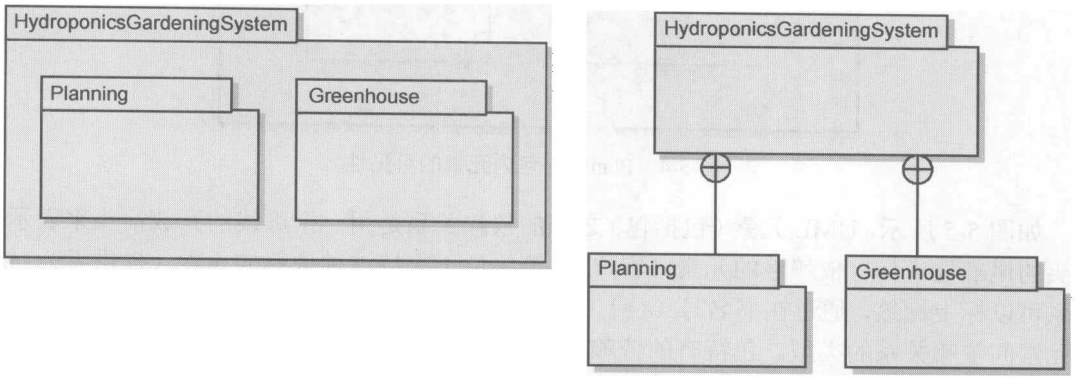


图 5-3 包容元素的包表示法

5.2.2 基本概念：元素的可见性

访问包中一组协作类（或者更一般地说，包中的任何元素）提供的服务取决于单个元素的可见性，包括内嵌的包。元素的可见性由包容的包所定义，可以是公有的或私有的，这既适用于被包容的元素也适用于导入的元素。导入元素的概念将在本节稍后介绍。

可见性是从包容的包的角度来定义的，包为它包含的元素提供了命名空间。因为包提供了命名空间，所以每个被包容的元素都有唯一的名称，至少在是同类型的其他元素范围内有唯一的名称。例如，这意味着同一个命名空间中没有两个类可以有相同的名称[49, 50]。我们将在讨论导入和访问时进一步讨论这个概念。

具有公有可见性的元素可以被认为是这个包的接口的一部分，因为这些元素可以被所有其他元素看见。具有私有可见性的元素不能够被包容的包之外的元素看见。下面提供了公有可见性和私有可见性的定义，括号中显示了相应的表示法：[51, 52]

- 公有（+） 对它所在的包（包括内嵌的包）以及外部的元素可见；
- 私有（-） 只对它所在的包和内嵌的包可见。

在一张图中，这种可见性表示法被放在元素名称的前面，如图 5-4 所示。GardeningPlan 类具有公有可见性，允许其他元素访问它，而 PlanAnalyst 类具有私有可见性。

5.2.3 基本概念：依赖关系

如果一个元素具有适当的可见性，允许对它进行访问，那么可以显示指向它的依赖关系，表示这种访问。这种依赖关系是包图中的另一种主要表示法，前面在讨论包表示法时曾提到这一点。依赖关系显示了一个元素依赖于另一个元素来实现它在系统中的责任。

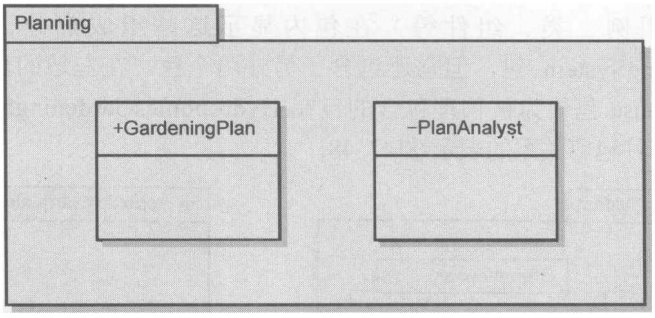


图 5-4 Planning 包内元素的可见性

如图 5-5 所示，UML 元素（包括包）之间的依赖关系是用一个虚线的开放箭头来表示的。箭头的尾部位于具有依赖性的元素（客户），箭头位于支持这种依赖的元素（提供者）。依赖关系可以标上标签，通过在书名号（《》）中包含依赖关系的类型（由一个关键词来表示），强调元素间依赖关系的类型。包特有的依赖关系包括导入、访问和合并，由于包容的元素之间的关系而导致的包间依赖关系包括跟踪、派生、精化、允许和使用[53]。

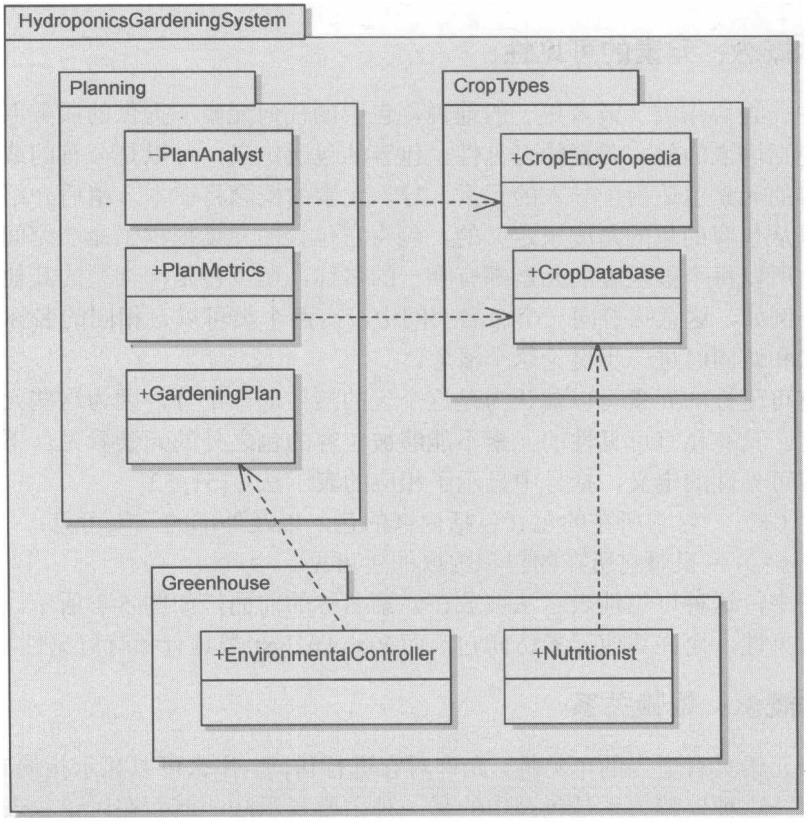


图 5-5 HydroponicsGardeningSystem 的依赖关系表示法

如果两个包之间存在多个包容元素依赖关系，这些依赖关系会聚合为包层面的依赖关系。包层面的依赖关系可以用一个关键词标签标出，放在书名号（《》）中，表示类型。但是，如果包含的依赖关系是不同类型的，包层面的依赖关系就不提供标签。图 5-6 展示了将图 5-5 所示依赖关系提升到包层面的情形。请注意，两个独立的元素依赖关系被聚合到了所在的包的层面上。

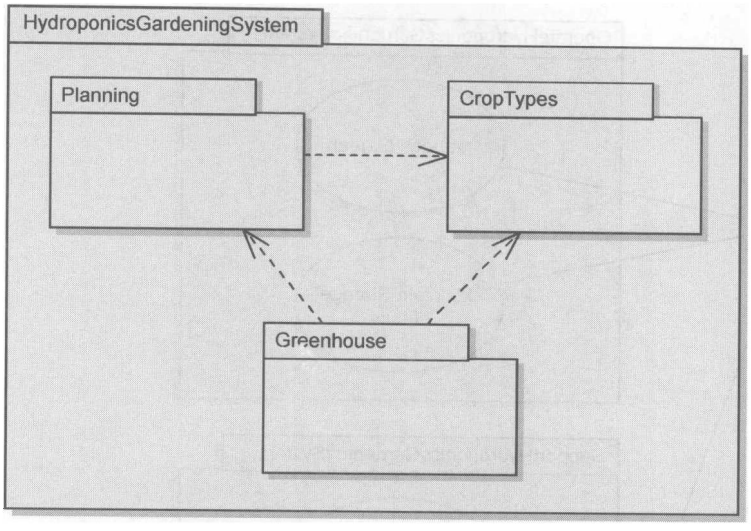


图 5-6 包容的元素依赖关系的聚合

5.2.4 基本概念：包图

到目前为止，我们讨论了可以作为包图的组成部分的元素：

- 包表示法；
- 元素可见性；
- 依赖关系。

包图是一种 UML 2.0 结构图，它包含作为主要被展现的 UML 元素的包，同时展示了包间的依赖关系。

但是，包表示法可以用于展示许多不同建模元素的结构和包容关系，例如类，前面在图 5-4 和图 5-5 中展示了这一点。它也可以用于不属于结构图的 UML 图。当我们提到包可以用来组织用例时，已经暗示了这一点。这样做的原因可能是在很大的系统中体现清晰性或划分工作。图 5-7 展示了一个例子，其中包被用来对 HydroponicsGardeningSystem 的用例进行分组，以便于两组具有不同专业知识的人（操作和支持）对它们提出规格说明[55]。本章稍后将深入讨论执行者和用例。

用包分组的元素通常应该在某种意义上是相关的。例如，它们是系统中的一个子系统，是系统某个方面的相关用例，或者是一些协作类，提供系统功能的一个子集[56]。

决定如何对元素进行分包的判定条件是什么？利用包来组织系统存在着许多不同的方

式，如按架构层、按子系统、按用户（针对用例），等等。好的分包是松耦合、高内聚的。也就是说，应该看到包内的元素有更多的交互，包间的元素有更少的交互。我们也应该尽量不要让泛化层次结构或聚合关系跨越包的边界。类似地，不要让用例的包含或扩展关系跨过包的边界[57]——本章稍后讨论了类和用例之后，这一点会更清楚。

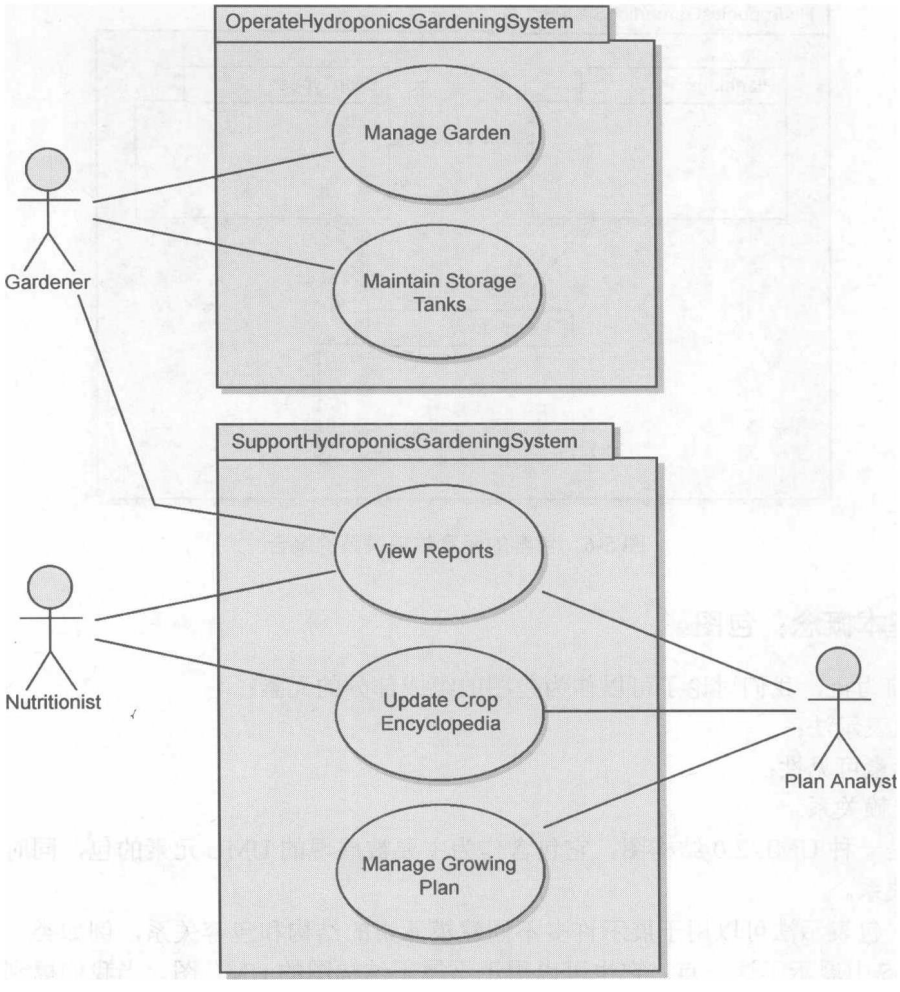


图 5-7 用于分割的包表示法

一个命名空间中包含的每个元素都可以通过“包名：元素名”这样的限定名称来引用。只要元素属于不同的命名空间（处于不同的包中），就可以有相同的名称[58, 59]。

5.2.5 高级概念：导入和访问

导入和访问实际上是同一个硬币的两面——导入是一种公有的包导入，而访问是一种私

有的包导入。这一点的真正意义在于：对于导入来说，其他可以看到导入包的元素也可以看到被导入的项；但对于访问来说，其他元素不能看到这些添加到导入包命名空间中的元素。这些被导入的项是私有的，它们在进行访问的包之外是不可见的[60、61]。

此时读者可能会问，为什么要进行包导入或包访问呢？这样做让我们能够通过限定名称引用其他命名空间的公有元素。执行导入的包将被导入元素的名称添加到了它的命名空间中。但是，如果相同类型的被导入元素刚好与已有的元素同名，那么它们就不会被添加到执行导入的包的命名空间中。类似地，如果从不同命名空间中导入的相同类型的元素具有相同的名称，它们也不会被添加到执行导入的包的命名空间中[62, 63]。

包元素的导入可以是广泛或集中的，即导入所有元素或者只导入选定的元素。回顾一下图 5-5，它展示了 PlanAnalyst 类以及对 CropEncyclopedia 类的依赖关系。因为 Planning 包没有导入 CropTypes 包，PlanAnalyst 必须使用限定名称 HydroponicsGardeningSystem::CropTypes::CropEncyclopedia 来指称 CropEncyclopedia 类。

为了表示 Planning 包导入了 CropTypes 包，我们显示了 Planning 到 CropTypes 的一个依赖关系，带有一个<<import>>标签，代表了公有的包导入，如图 5-8 所示。这意味着，PlanAnalyst 和 PlanMetrics 可以用无限定的名称来访问 CropEncyclopedia 类和 CropDatabase 类。对于 PlanMetrics 也是如此，因为它的命名空间（Plan 包）允许它访问外层包中的元素。

图 5-8 还显示了 Planning 包对 Plans 包执行了私有引入，依赖关系上显示了<<access>>标签。要让 PlanAnalyst 类使用无限定的名称来访问 GardeningPlan 和 PlanMetrics 类，这样做是必要的。由于访问依赖关系是私有的，所以虽然 Greenhouse 包引入了 Planning 包，但 Greenhouse 包中的元素（如 Gardener）仍不能够用无限定的名称来引用 GardeningPlan 和 PlanMetrics 类。而且，Greenhouse 包的元素根本不能看到 PlanAnalyst 类，因为它是私有可见的。

请看 Greenhouse 包的内部，Gardener 类必须使用 StorageTank 包的限定名称，因为它的命名空间没有导入这个包。例如，它必须使用 StorageTank::WaterTank 来引用 WaterTank 类。再进一步，让我们看看 EnvironmentalController 包中的元素。它们都具有私有可见性。这意味着它们在其命名空间之外是不可见的，即在 EnvironmentalController 包之外不可见。

综上所述，无限定的名称（常常被称为简单名称）是没有任何路径信息的名称，没有告诉我们它位于模型的什么位置。在包中，可以使用这种无限定的名称来访问以下元素[64, 65]：

- 包拥有的元素；
- 导入的元素；
- 外层包中的元素。

嵌套的包可以使用无限定的名称来引用外层包中的内容，而不论嵌套的层次有多少。但是，如果外层包中相同类型的元素具有和内层包中一样的名称，就必须使用限定名称。从外

层包的角度来看，访问的情形就完全不同——外层包需要导入它内嵌的包，然后才能使用无限定的名称来引用它们的元素[66, 67]。

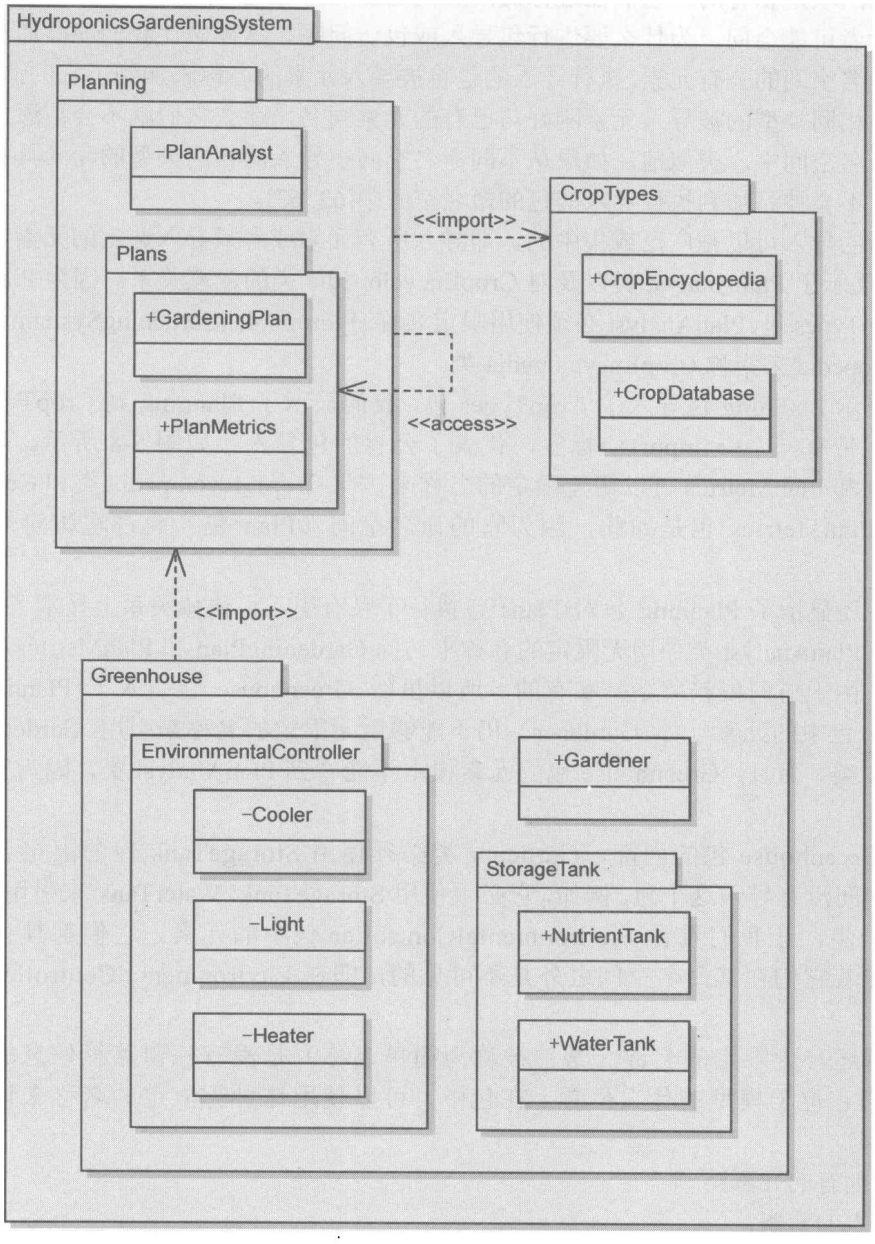


图 5-8 HydroponicsGardeningSystem 中的包导入

5.3 组件图

组件代表了一块可复用的软件，它提供了某种有意义的功能集。从最低的层面上来说，组件是一组类，它们本身是内聚的，与其他类的耦合相对比较松。系统中的每个类要么处于一个组件中，要么处于系统的最顶层。组件也可以包含其他组件。

组件是一种结构化的分类器（classifier），组件间的协作和内部结构可以利用组件图来表示。组件与组件之间通过定义良好的接口进行协作，从而提供系统的功能。组件本身也可以由一些协作的组件组成，提供它自己的功能。因此，我们可以用组件分层地解构一个系统，并表示它的逻辑架构。这种组件的逻辑视角是 UML 2.0 中新出现的。以前，组件被看作是在系统内部署的物理部件。现在，组件可以由部署在一个节点上的一个工件来实现[68, 69]。

组件图中的基本元素是组件、它们的接口和它们的实现。

5.3.1 基本概念：组件表示法

由于组件是一个有结构的分类器，所以它的详细装配可以通过一个组合结构来表示，这个结构包括部件、端口和连接器。图 5-9 展示了用于表示组件的表示法。它的名称，即 EnvironmentalControlSystem，是包含在分类器矩形框中的粗体字，使用开发团队所确定的某种命名惯例。另外，应该包含一种或两种组件标识：关键词标签<<component>>和分类器矩形框右上角显示的组件图标[70, 71]。

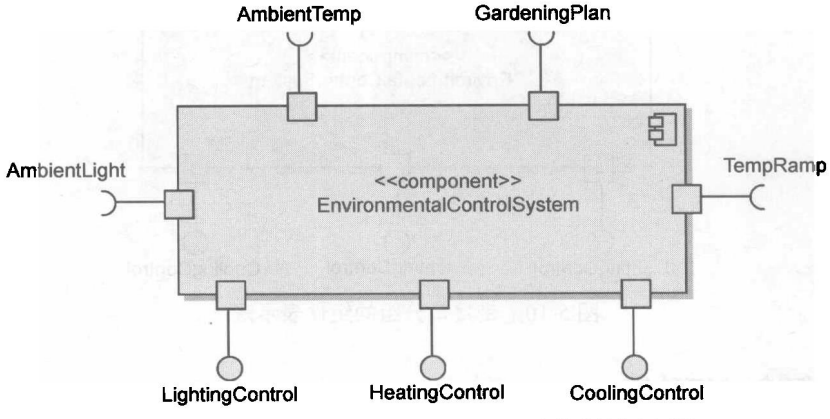


图 5-9 EnvironmentalControlSystem 中的组件表示法

在分类器矩形框的边界上，有 7 个端口，它们用小的正方形来表示。除非另有说明，端口将具有公有可见性。组件也可以拥有隐藏的端口，它们也是用同样的小正方形来表示的，但它们完全位于边界的内部。隐藏端口可以用于一些不公开提供的功能，如测试点等。组件通过端口与它的环境进行交互，端口为这个结构化的分类器提供了封装。这 7 个端口是没有

被命名的，但在需要阐明时应该命名，其格式为：端口名称: 端口类型。在命名端口时，端口类型是可选的[72, 73]。

对于图 5-9 中显示的端口来说，我们有与之相连的接口，这些接口定义了组件交互的细节。接口显示为小球和球窝的样子。提供的接口使用小球表示法，表明该组件向环境提供的功能，LightingControl 就是提供的接口的一个例子。要求的接口使用球窝表示法，表明该组件向环境要求的接口，AmbientTemp 就是要求的接口的一个例子[74, 75]。

这种 EnvironmentalControlSystem 的表现形式被认为是一种黑盒视图，因为我们只能看到组件在边界上要求的功能和提供的功能。我们不能够透视内部，看到那些被封装的组件或类，组件的功能实际上是由这些内部的组件或类提供的。

端口和接口之间不一定是一对一的关系，端口可以用来对接口分组，如图 5-10 所示。这样做可能是为了在复杂的图中提供清晰性，也可能是为了反映拥有一个端口的意图，某些类型的交互将通过这个端口进行。在图 5-10 中，环境光和温度的测量数据是在一个端口接收的。类似地，培育计划和温度坡度信息是由水培园艺系统的员工提供的，通过一个端口接收。在使用这种表示方法时请注意，不同接口的名称用逗号隔开。在这些复杂的端口上，也可以显示独立的接口，这样，一个端口上就包含两个要求的接口，分别是 AmbientLight 和 AmbientTemp。另一个端口上包含两个要求的端口，分别是 GardeningPlan 和 TempRamp[76, 77]。

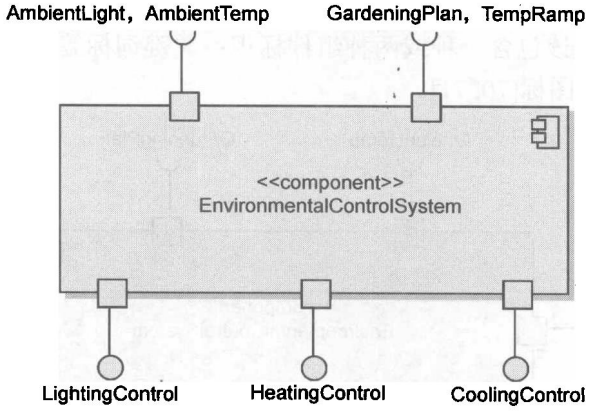


图 5-10 带接口分组的组件表示法

5.3.2 基本概念：组件图

在开发中，我们利用组件图来表达架构的逻辑分层和划分方式。组件图中展现了组件间的相互依赖关系，也就是它们通过定义良好的接口进行协作，从而提供系统的功能。图 5-11 展示了 EnvironmentalControlSystem 的组件图。这个白盒视图中展示了提供组件功能的 4 个被封装的组件：Environmental-Controller、LightingController、HeatingController 和 CoolingController [78, 79]。

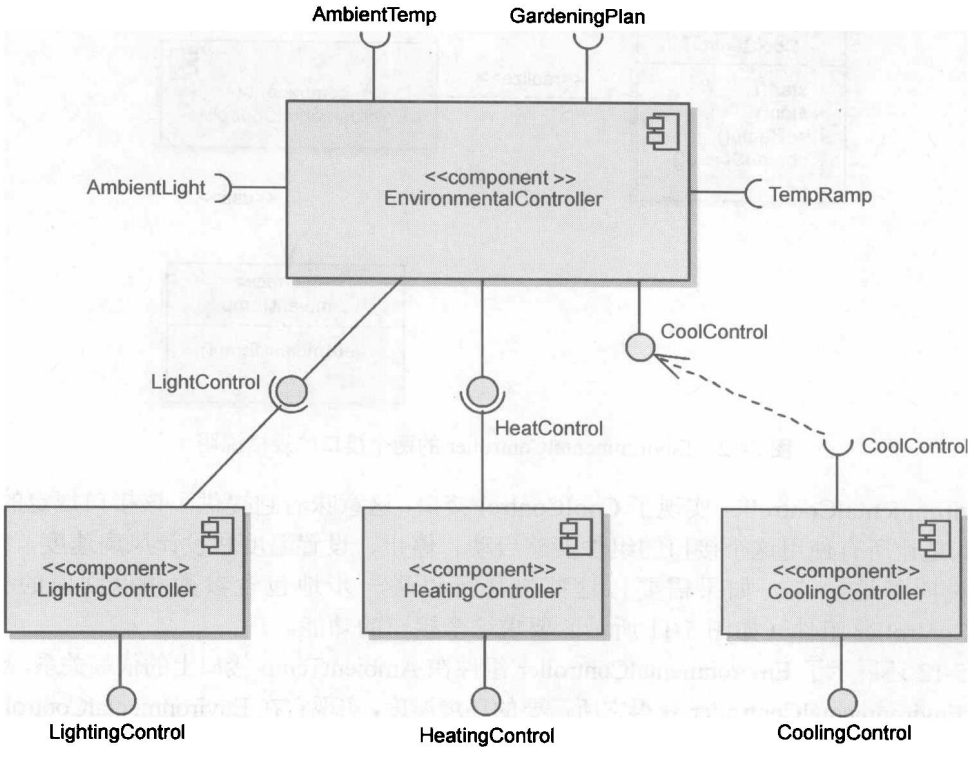


图 5-11 EnvironmentalControlSystem 的组件图

像图 5-9 一样，我们使用小球和球窝表示法来指定每个组件提供的接口和要求的接口。两个组件之间的接口被称为“组装连接器”，也被称为“接口连接器”。虽然组装连接器是通过小球和球窝表示法来表示的，但也可以使用直线来表示每个连接。然而，用直线连接给出的信息要少一些。EnvironmentalController 和 CollingController 之间的接口显示了一种依赖关系，这是另一种表示方式。这种依赖关系实际上是多余的，因为接口的名称是一样的：CoolControl[80, 81]。

在前面，我们提到了组件的复用本质。例如，只要另一个组件实现了 LightingController 要求的接口，它就可以取代 EnvironmentalControlSystem 中的 LightingController。组件的这种特性意味着可以更容易地根据需要升级系统。实际上，EnvironmentalControlSystem 的全部内容都可以被换掉，只要它包含的组件能够满足它要求的接口和提供的接口。

5.3.3 基本概念：组件接口

如果需要更详细地显示组件的接口，可以提供接口规格说明，如图 5-12 所示。在我们的例子中，规格说明只关注了 EnvironmentalController 的 7 个接口中的 2 个：CoolControl 和 AmbientTemp [82, 83]。

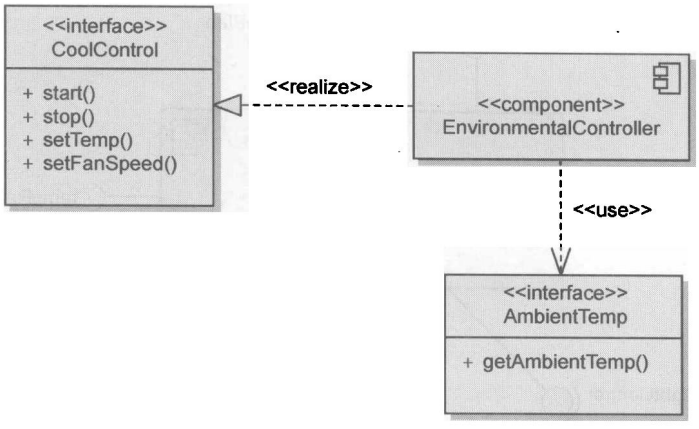


图 5-12 EnvironmentalController 的两个接口的规格说明

EnvironmentalController 实现了 CoolControl 接口，这意味着它提供了该接口规定的功能。这个功能是让所有使用这个接口的组件能够启动、停止、设置温度和设计风扇速度，它所包含操作说明了这一点。如果需要，这些操作可以进一步地包含参数和返回值的类型。CoolingController 组件（如图 5-11 所示）要求这个接口的功能。

图 5-12 还展示了 EnvironmentalController 组件在 AmbientTemp 接口上的依赖关系。通过这个接口，EnvironmentalController 获得它所需要的环境温度，实现它在 EnvironmentalControlSystem 组件内的责任。

图 5-13 中展示了 EnvironmentalController 的接口的另一种表示法。这里可以看到，在标题<<provided interfaces>>下面列出了 3 个提供的接口。对于图 5-12 中指定的 CoolControl 接

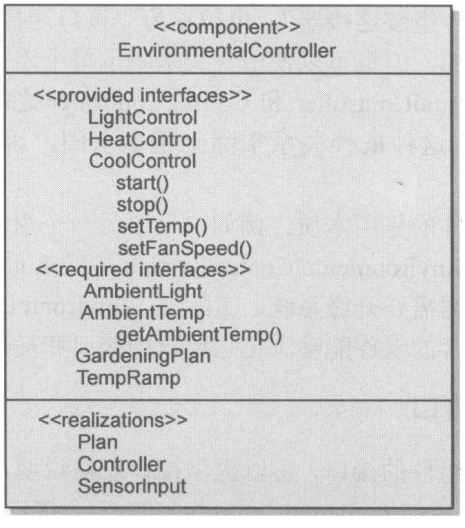


图 5-13 EnvironmentalController 的接口和实现的另一种表示法

口，我们提供了相关的操作。类似地，标题<<required interfaces>>下面列出了要求的接口，标题<<realizations>>下面列出了三个类[84, 85]。下面将讨论实现的概念。

5.3.4 基本概念：组件实现

图 5-13 指出，EnvironmentalController 组件是由 Plan、Controller 和 SensorInput 类实现的。这三个类提供了组件提供的接口所宣称的所有功能。但是，要做到这一点，需要组件要求的接口中指定的功能。

图 5-14 展示了 EnvironmentalController 组件以及 Plan、Controller 和 SensorInput 类之间的这种实现关系。这里，我们看到从每个类指向 EnvironmentalController 的实现依赖关系。同样是这一信息，也可以通过包含关系来表示，如图 5-15 所示，每个类都被物理包含在

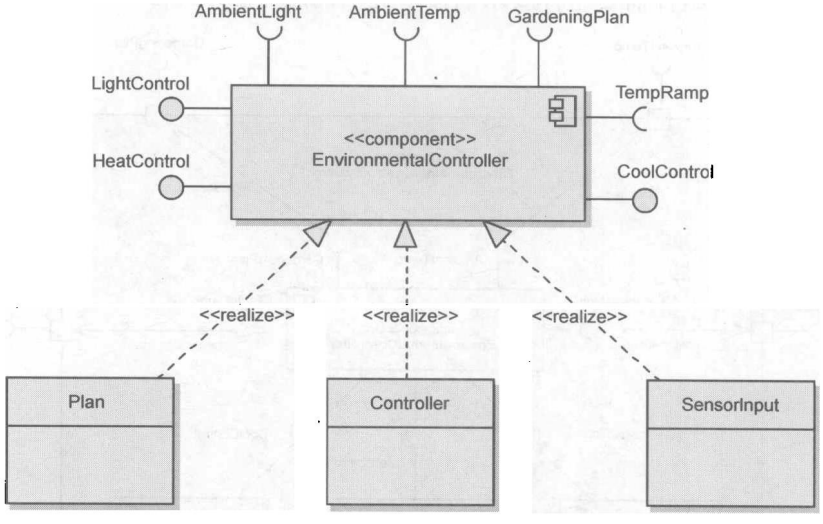


图 5-14 EnvironmentalController 的实现依赖关系

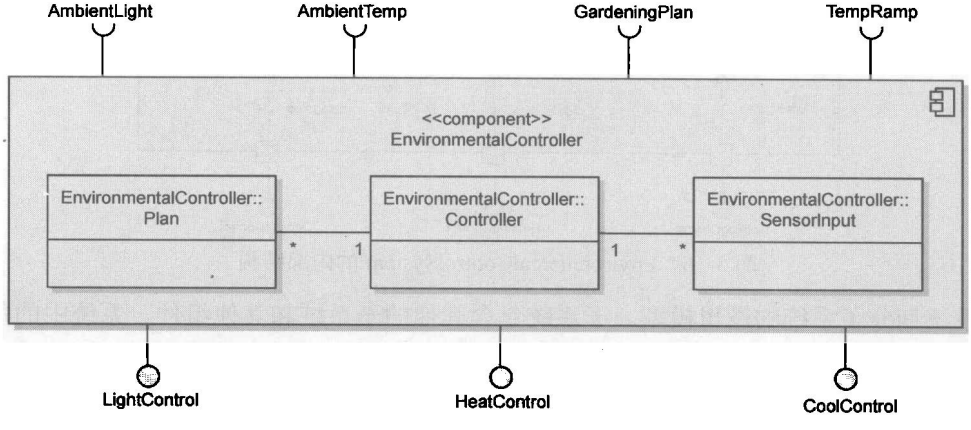


图 5-15 EnvironmentalController 实现的包容表示方式

EnvironmentalController 组件之中。这些内部类的命名惯例取决于建模工具。另外，请注意类之间的关联和多重性指定[88]。

5.3.5 高级概念：组件的内部结构

组件的内部结构可以通过内部结构图来展示。图 5-16 展示了 EnvironmentalController 子系统的一张内部结构图。在这个例子中，我们将关键词标签从<<component>>改为<<subsystem>>，因为它由 4 个具有某种复杂度且逻辑相关的组件组成。当然，这是一种主观的称谓，将标签保持为<<component>>也是可以的。另外，图 5-16 包含了我们还未遇到过的一种表示法，即<<delegate>>标签，它位于内部组件的接口和 EnvironmentalControlSystem 边界上的端口之间的连线上。这些连接提供了一种手段，表示哪个内部组件实现了提供的接口的责任，以及哪个内部组件需要组件要求的接口提供的服务[89, 90]。

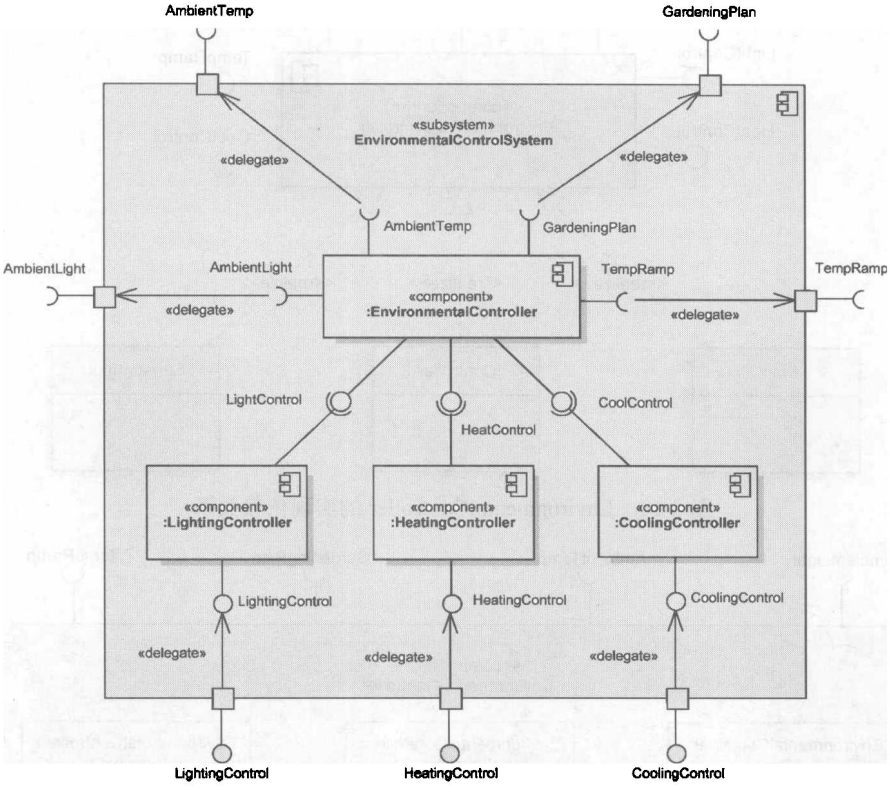


图 5-16 EnvironmentalControlSystem 的内部结构

子系统划分了系统的逻辑模型。子系统包含其他的子系统和其他组件。系统中的每个组件要么处于某个子系统中，要么处于系统的顶层。在实践中，大的系统会有一个顶层组件图，包含处于最高抽象层的子系统。开发者通过这个图来理解系统的总体逻辑架构。

这种表现形式比图 5-11 更能体现 EnvironmentalControlSystem 作为一个可复用组件（或子

系统）的特点。我们在边界上使用了端口，同时展示了实现接口“契约”的责任被代理给了一个或多个内部的部分。但是要记住，这些内部的部分可能需要来自 EnvironmentalControlSystem 组件的环境的服务，如培育计划，才能满足契约。

具体来说，EnvironmentalControlSystem 要求 GardeningPlan，后者规定了溶液种植园系统的环境需求（光照、加热和制冷）。这种对要求的接口的需要被代理给了一个未命名的端口，它与 GardeningPlan 接口相连。通过这种方式可以知道，如果我们想使用 EnvironmentalControlSystem 组件的服务，就必须向它提供培育计划。我们也知道，必须为它提供 AmbientLight、AmbientTemp 和 TempRamp 服务。

EnvironmentalControlSystem 的连接器提供了它与环境之间的通信联系，也为它的部件提供了内部通信的方式。在图 5-16 中，EnvironmentalControlSystem 视图中的新连接器类型是“委托连接器”，前面已提到过。通过这些连接器，EnvironmentalControlSystem 与环境沟通了它的责任（提供的接口）以及它的需求（要求的接口）。例如，:LightingController 组件透明地提供了 LightingControl 服务。EnvironmentalControlSystem 的用户不太可能拥有子系统的这种白盒视图[91, 92]。

5.4 部署图

部署图用于展示在系统的物理设计中，工件在节点上分布的情况。单张部署视图代表了一种系统工件结构的视图。在开发中，我们使用部署图来说明节点的物理集合，这些节点是系统执行的平台。

部署图有三个基本元素：工件、节点和它们的连接。

5.4.1 基本概念：工件表示法

工件是物理上存在的一件东西，它实现了一部分的软件设计。它通常是软件代码（可执行），但也可能是一个源文件、一份文档或与软件代码相关的其他文件。工件之间可以有关系，如依赖或组合[20, 21]。

工件的表示法包括一个带有工件名称的类矩形、关键词标签<<artifact>>及一个可选的图标。其图标看起来像一页折了右上角的纸。图 5-17 展示了 HeatingController.exe 工件，带有可选的图标。

工件的名称中包含扩展名.exe，表明这是可执行的（即软件代码）。HeatingController.exe 工件拥有一个到 HeatingController 组件的依赖关系，带有<<manifest>>标签。这意味着它是这个组件的物理实现，于是实现和设计就联系起来了。一个工件可以体现多个组件[22]。

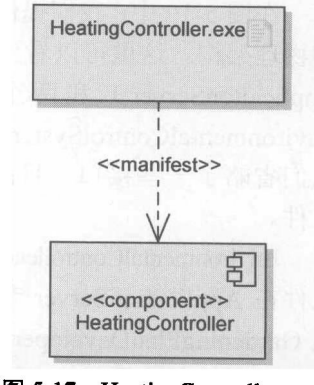


图 5-17 HeatingController.exe 的工件表示法

5.4.2 基本概念：节点表示法

节点是一种计算资源，通常包含存储和处理能力，工件部署在它上面执行。节点可以包含其他节点，以表示复杂的执行能力，这种情况是通过嵌套或利用组合关系来体现的。有两种类型的节点：设备和执行环境[23, 24]。

设备是一个提供了计算能力的硬件，如一台计算机、一个调制解调器或一个传感器。执行环境是软件，它用于部署特定类型的执行工件，如<<database>>和<<J2EE server>>。执行环境通常以一个设备作为宿主[25]。

图 5-18 展示了用于表示节点的三维立方体，即 PC 和 ApplicationServer 节点。这个图标可以带有一个符号，提供节点类型的额外可视化说明。对于节点的名称没有特别的限制，因为它们代表硬件实体，不是软件实体。

节点之间会通过消息和信号进行通信，我们用一条实线来表示通信路径。通常认为通信路径是双向的，如果某个连接是单向的，可以加上箭头来表示方向。每条通信路径可以包含一个可选的关键词标签，如<<http>>或<<tcp/ip>>，它提供了连接有关的信息。也可以为通信路径连接的节点指定多重性。

在图 5-18 中，PC 和 ApplicationServer 节点之间的通信是双向的。通信路径通常表示某种直接的硬件耦合，如 USB 电缆、以太网连接甚至是共享内存。但是，路径也可以表示不太直接的耦合，如卫星和地面的通信或移动电话通信。在我们的例子中，它表示利用 TCP/IP 协议的双向连接。我们规定一个或多个 PC 节点可以连接到 ApplicationServer 节点上。

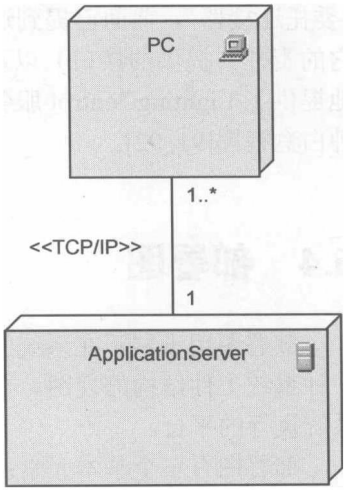


图 5-18 两个节点的表示法

5.4.3 基本概念：部署图

在图 5-19 中，我们提供了一个部署图的例子，它主要是溶液种植园系统的环境控制系统的物理架构。这里可以看到，我们的系统架构师决定将这部分系统分解为两个节点（PC 和 ApplicaitonServer）和两个设备（LightMeter 和 Themometer）。如果将这张部署图和 EnvironmentalControlSystem（图 5-11）的组件图进行比较，会看到它没有包含全部的接口。我们省略了一些接口，目的是使例子简单一些。另外，我们展示了每个工件只实现了一个组件。

EnvironmentalController.exe、LightingController.exe、HeatingController.exe 和 CoolingController.exe 工件在 ApplicationServer 节点上的部署是通过包容来表示的。另一种表示部署的方法是显示从 GardeningPlanDeveloper.exe 工件到 PC 节点的依赖关系，并带有<<deploy>>标签。第三种表示部署的方法是在节点图标内用文本的方式列出工件，这对于大型或更复杂的部署图尤其有用[26, 27]。

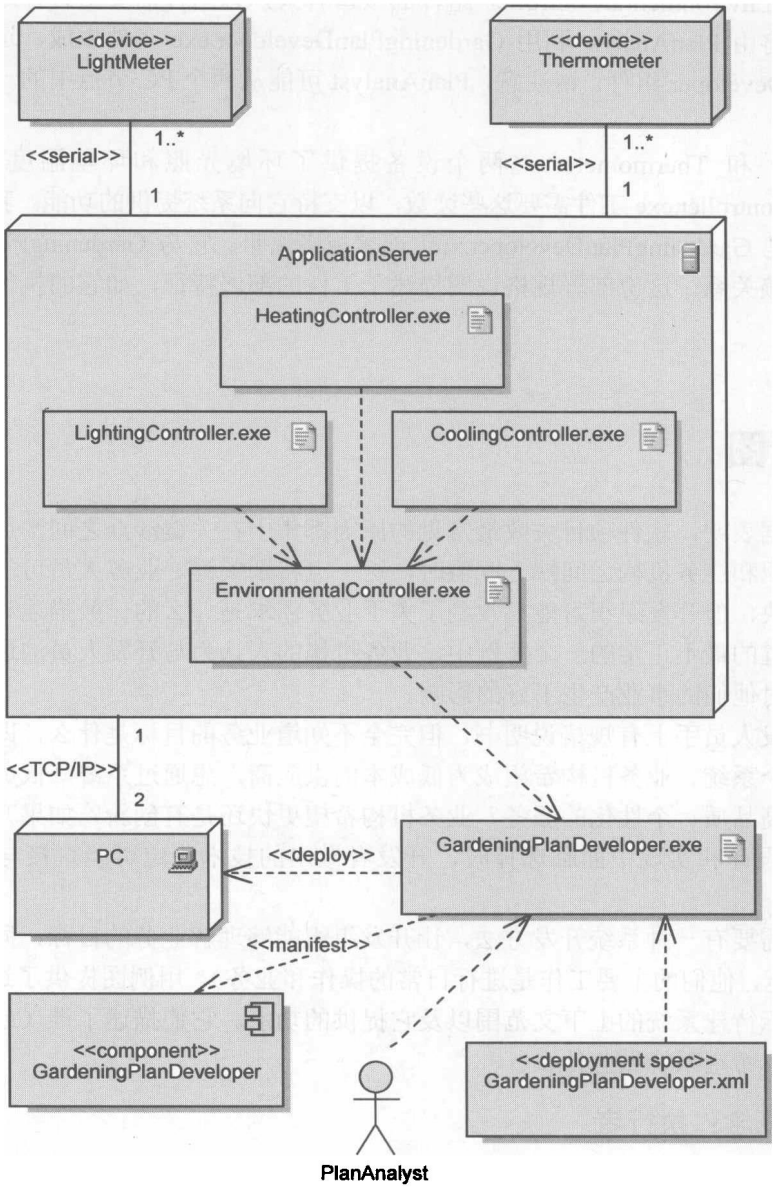


图 5-19 EnvironmentalControlSystem 的部署图

在图 5-19 中，有三个未命名的依赖关系，即从 LightingController.exe、HeatingController.exe、CoolingController.exe 工件到 EnvironmentalController.exe 工件的依赖关系。这些表示了它们所实现的组件之间的关系，而不是部署到节点上。

还有另一个从 EnvironmentalController.exe 工件到 GardeningPlanDeveloper.exe 工件的依

赖关系。这与 EnvironmentalController 组件的接口有关，该组件需要培育计划。这里我们看到，培育计划将由 PlanAnalyst 利用 GardeningPlanDeveloper.exe 工件生成，这个工件体现了 GardeningPlanDeveloper 组件。请注意，PlanAnalyst 可能从两个 PC 节点中的一个节点上完成这项任务。

LightMeter 和 Thermometer 这两个设备提供了环境光照和环境温度传感器读数，EnvironmentalController.exe 工件需要这些读数，以支持它向系统提供的功能。我们还没有讨论过的一个工件是 GardeningPlanDeveloper.xml 部署规格说明，它与 GardeningPlanDeveloper.exe 工件有一个依赖关系。这份部署规格说明描述了工件的部署特征，如它的执行和事务规格说明[28]。

5.5 用例图

多年的证据表明，软件项目失败最常见的原因都集中在关键涉众之间沟通不佳或缺少沟通。当开发组织和业务机构之间缺乏联盟时，这一点特别关键。业务人员可能知道他们有某个问题需要解决，但开发组织可能只收到了关于业务需求是什么的一般描述和一些具体的需求。在我们见过的最不正常的一个案例中，业务机构的人认为与开发人员沟通是一件丢人的事情，有可能对他们的事业产生不好的影响。

有时候开发人员手上有规格说明书，但完全不知道业务的目标是什么，即不知道为什么他们要构建这个系统。业务机构希望成为低成本的供应商，想通过系统降低成本吗？或者它的目标是提供高品质、个性化的服务？业务机构希望更快还是有创新？如果开发组织不理解业务目标，在设计和实现中面临选择时，开发者做出的技术决定可能直接与业务目标发生冲突。

我们非常需要有一种系统开发方法，让开发组织能够理解业务的目标，同时又不会麻烦业务人员（毕竟，他们的主要工作是进行日常的操作和业务）。用例图提供了这种能力。我们通过用例图展示待建系统的上下文范围以及它提供的功能。它们描述了谁（或什么）与系统交互，外部世界希望系统做些什么。

5.5.1 基本概念：执行者

执行者是与系统交互的实体，他们可以是人或其他系统。执行者位于他们使用的系统之外，用棍状的小人来表示。图 5-20 展示了前面讨论过的溶液种植园系统的两个执行者。



图 5-20 执行者

在考虑执行者时，一种方法是思考执行者所扮演的角色。在真实世界中，人们（系统）可能承担许多不同的角色。例如，一个人可以是一名销售人员、一名经理、一个父亲、一个艺术家等。

5.5.2 基本概念：用例

用例代表了执行者希望系统为他们做什么。图 5-21 展示了溶液种植园系统的一些用例，用一些椭圆表示。用例不仅仅是系统可以提供的功能，从“执行者的观点”来看，用例必须是一个“完整”的活动流程，它为执行者提供了“价值”。Jacobson 等人对用例的定义如下[16]：

“用例是通过某部分功能来使用系统的一种具体的方式……因此，用例是相关事务的一个具体序列，执行者和系统以对话的方式执行这些事务……从用户的观点来看，每个用例都是系统中一个完整序列的事件。”



图 5-21 用例

5.5.3 基本概念：用例图

为了展示哪个执行者使用哪个用例，可以利用基本的关联连接执行者和用例，创建一张用例图。这种关联用实线表示，如图 5-22 所示。

用例图中的关联表明哪个用户发起哪个用例。这里我们看到，只有执行者 Gardener 才能维护储水箱，但所有的执行者都可以查看报告。

1. 确定用例细节

那么，如何指定用例提供的功能细节？如何指定这个完整序列的事件？较好的方法是利用其他的 UML 模型（如活动图，本章稍后将讨论）和文本规格说明。在 UML 文献中，用例规格说明有许多不同的形式。大多数都包括以下信息：用例的名称、关于它的目的一段简短描述、乐观流程（即如果一切正常，用例中发生的事件流）、一个或多个更实际的流程（即事情没有按照愿望发生时的流程）。

2. 用例规格说明示例

下面来看看“维护储水箱”用例的例子。

用例规格说明

用例名称：维护储水箱

用例目的：本用例提供了维护储水箱中储水量的功能。本用例让执行者维护一组水箱和营养液箱。

乐观流程：

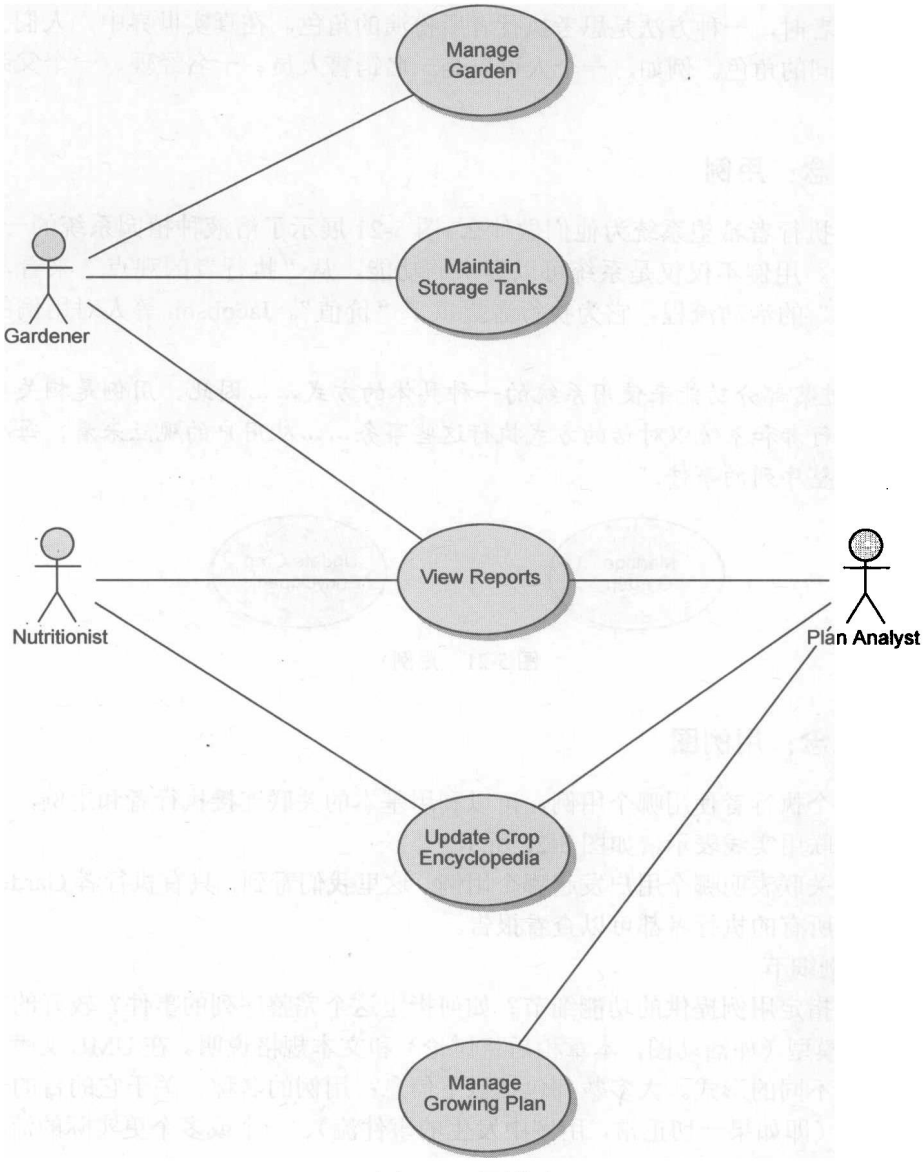


图 5-22 用例图

A. 执行者检查储水箱的当前容量。
B. 执行者决定水箱需要加注。
C. 水箱正常的水培园艺系统操作被执行者挂起。
D. 操作者选择水箱并设置加注量。
对于每个选定的水箱，执行步骤 E～G。
E. 如果水箱正在加热，系统停止加热器。

1．加热器达到安全温度。

F．系统加注水箱。

G．当水箱加注完成后，如果水箱是在加热的，系统启动加热器。

1．水箱达到操作温度。

H．执行者继续正常的水培园艺系统操作。

实际流程：

条件触发的可选流程如下。

条件 1：原料不够，不足以将水箱加注到执行者指定的容量。

D1．提醒执行者原料不够，不能达到设置的值。显示可用的原料。

D2．提示用户选择终止维护或重新设置加注容量。

D3．如果选择重新设置，执行步骤 D。

D4．如果选择终止，执行步骤 H。

D5．否则，执行步骤 D2。

条件 2：……

其他有用的信息也可以被加到规格说明中，如进入条件（在执行这个用例之前什么条件必须为真）、退出条件（在执行这个用例之后什么条件必须为真）、限制条件、假定等。例如，在我们的水培园艺系统中有一个限制条件，针对一种作物的营养液箱和水箱必须成对加注，在一次维护活动中完成。这是一个业务操作决定（限制条件），目的是防止破坏提供给这种作物的营养液和水的比例。

总之，用例规格说明不应该太长——它应该只有几页。如果规格说明非常长，就应该考虑一下这个用例做的事情是否太多了，有可能它实际上是多个用例。另外，出于实际的原因，也不能包含所有会触发可选流程的事情，而只要包含最重要、最关键的可选流程。不要包含所有可能的出错条件，诸如操作者输入的数据格式错误的情况（让用户界面去处理这类异常）。

5.5.4 高级概念：<<include>>和<<extend>>关系

有两种主要用于组织用例模型的关系——<<include>>和<<extend>>，它们既强大，又常被误用。这些关系用在用例之间。

1．<<include>>关系

在水培园艺系统的例子中，我们有一个名为“Update Crop Encyclopedia”（更新作物百科全书）的用例。在分析过程中，我们确定使用这个用例的执行者 Nutritionist 将先看到作物百科全书的内容，再更新它。这就是 Nutritionist 可以调用“View Reports”（查看报告）用例的原因。执行者 Gardener 在每次调用 Maintain Storage Tanks 时也是这样。两种执行者都不应该盲目地执行用例。因此，View Report 用例是其他两个用例需要的共同功能。这可以利用<<include>>关系在用例模型中描述，如图 5-23 所示。

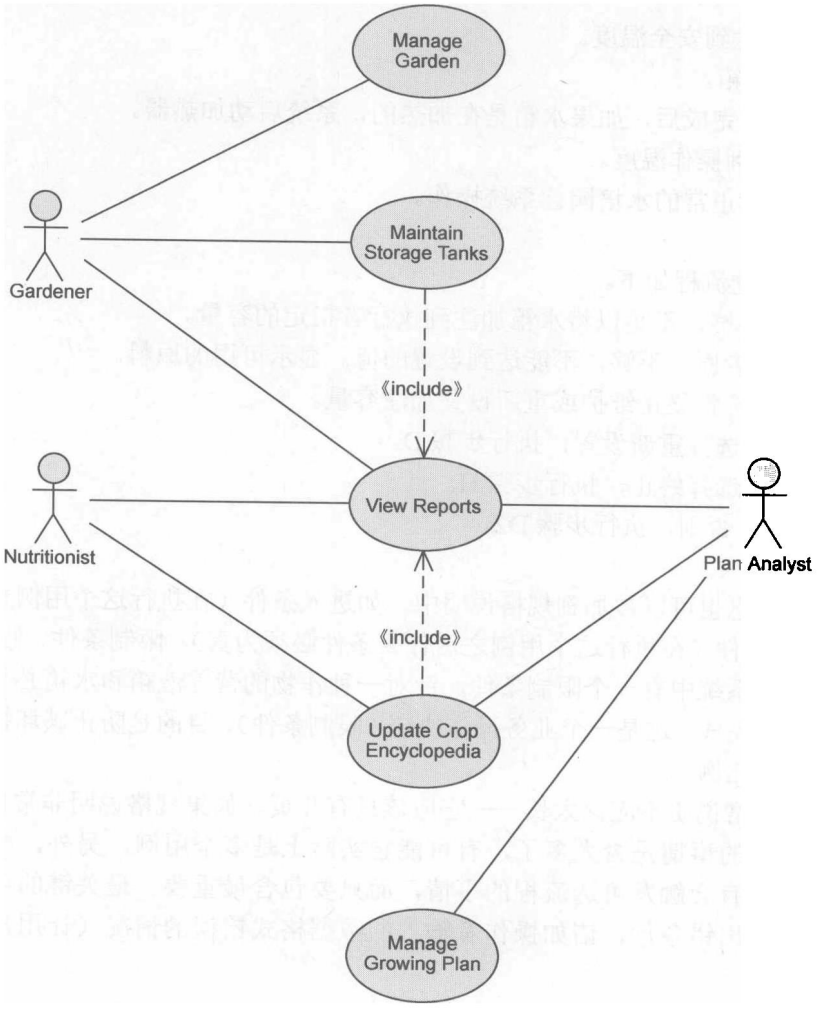


图 5-23 显示<<include>>关系的用例图

例如，这张图表明，Update Crop Encyclopedia 用例包含了 View Reports 用例。这意味着在执行 Update Crop Encyclopedia 时，View Reports 必须被执行。如果没有 View Reports，Update Crop Encyclopedia 就不能被看作是完整的。

当一个被包含的用例执行的时候，它在用例规格说明中就体现为一个包含点。包含点指明了在外层用例的什么位置执行被包含的用例。

2. <<extend>>关系

在开发用例时，可能会发现某些活动可以作为用例的一部分执行，但并不一定要运行它，用例也能成功。在我们的例子中，当执行者 Gardener 执行 Manage Garden 用例时，他可能想看一下某些报告，这可以通过 View Reports 用例来实现。但在执行 Manage Garden 时，View Reports 并不是必需的，Manage Garden 本身就是完整的。所以我们修改用例图，表明 View

Reports 用例扩展了 Manage Garden 用例，如图 5-24 所示。

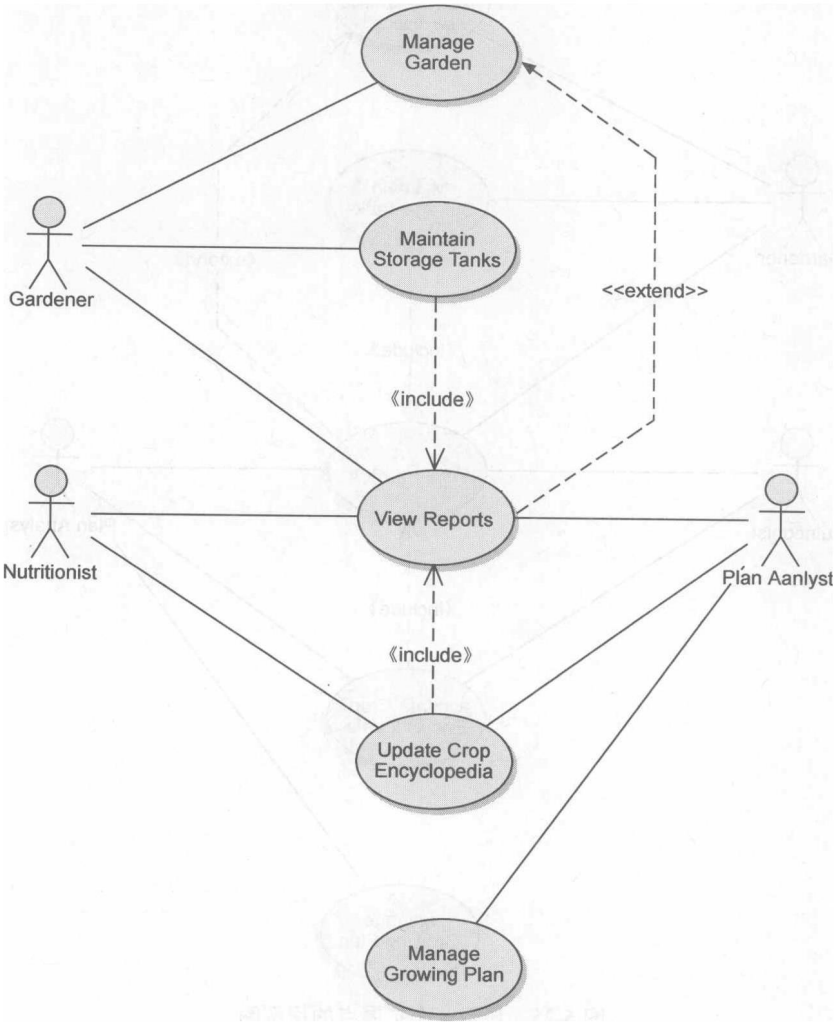


图 5-24 显示<<extend>>关系的用例图

当一个扩展的用例被执行的时候，它在用例规格说明中就体现为一个扩展点。扩展点指明了在外层用例的什么位置执行扩展的用例。请注意，扩展点也可以在用例图中显示，如图 5-25 所示。

3. <<include>>和<<extend>>关系的危险

虽然这两个概念在指定共同功能（<<include>>）和简化复杂用例流程（<<extend>>）时非常有用（我们前面进行了讨论），但是这些概念在用例建模时常常被误用。其主要的原因是有些人不太清楚<<include>>和<<extend>>之间的语义区别。关于这些区别，Maksimchuk 和 Naiburg 进行了准确地总结[17]，如表 5-1 所示。

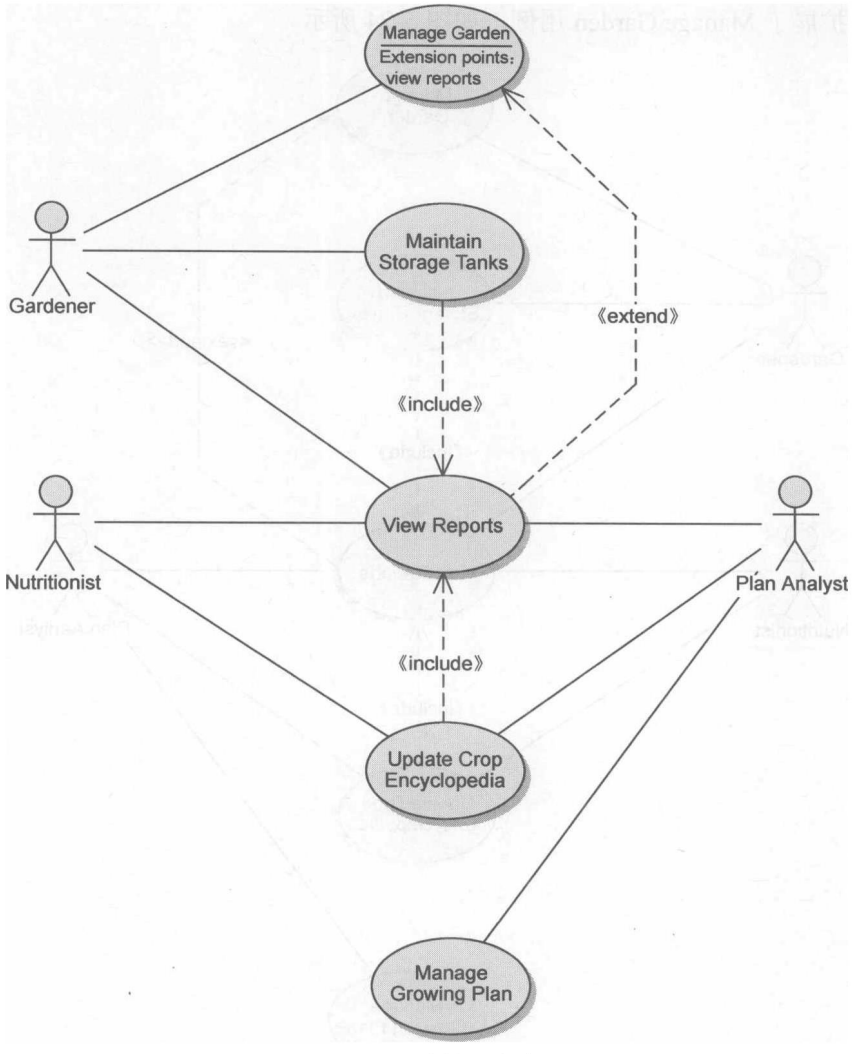


图 5-25 显示一个扩展点的用例图

表 5-1 <<include>>和<<extend>>关系之间的关键区别[a]

	包含的用例	扩展的用例
这个用例是可选的吗	否	是
没有这个用例，基本用例还完整吗	否	是
这个用例的执行是有条件的吗	否	是
这个用例改变了基本用例的行为吗	否	是

a. 复制得到了 Maksimchuk 和 Naiburg 的许可[17]。

对于这些关系，我们看到的另一个常见错误就是违反基本的用例原则。包含的用例和扩

展的用例仍然是用例，必须满足前面提到的用例原则。用例代表了一个“完整”的活动流程，它从“执行者的观点”反映了执行者希望系统做些什么，并为执行者提供了“价值”。如果坚定地坚持这些原则，还可以避免另一个错误：利用这些关系对用例进行“功能分解”。对于用例模型，这是我们看到的最普遍的问题，人们将用例变成越来越小的片段，利用<<include>>或<<extend>>将它们拼在一起。这个问题的根源在于结构化分析/结构化设计（SA/SD）的方法非常盛行的软件开发文化。这些方法将大的开发问题分解为较小的块，而这样做很快就会违反上一段中我们提到的用例原则，很快就会丧失前 4 章中提到的对象开发模型的好处。

5.5.5 高级概念：泛化

在第 3 章中曾介绍了泛化关系，它们也可以被用于组织用例。和类一样，用例也可以有一些共同的行为，其他用例（即子用例）可以通过添加步骤或精化其他方面来修改这些共同行为。例如，图 5-26 展示了购买不同门票的用例。

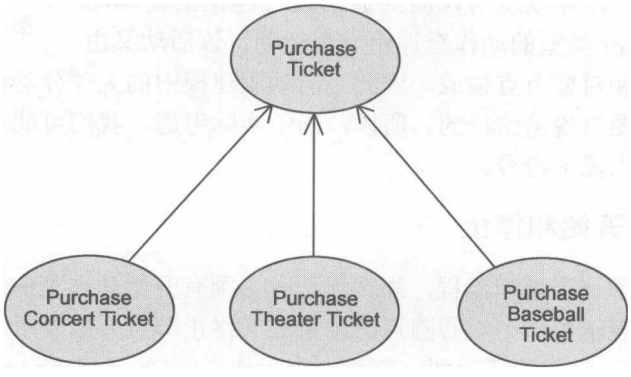


图 5-26 用例泛化

Purchase Ticket 包含了购买任何门票所需的基本步骤，而子用例针对具体门票的种类对 Purchase Ticket 进行了特化。

5.6 活动图

在前面一节中，我们用文本描述了用例的流程（用例规格说明）。文本描述有一些好处：它们容易被创建和修改（不需要复杂的工具），它们可以在任何地方被创建（只需要纸和笔），它们可以很容易地由业务人员和开发人员使用和分享，等等。但是，用文字描述的复杂用例、业务过程和算法可能难以理解。复杂流程的可视化表示就要好很多。我们可以用眼睛看到潜在的问题，一叠文本规格说明可能永远不能暴露这些种问题。

例如有一个项目，其中复杂的生产过程记录在大量的正式规格说明书上，一个公司负责

开发和维护这个项目。其他公司也会复查这些规格说明，它们是过程的实现者。即使经过了这些严格的控制，当基本的生产流程视图出现时，生产过程中多处走不通的环节很快就暴露出来了。

活动图提供了活动流程的可视化描述，可以是在系统、业务、工作流或其他过程中。这些图关注被执行的活动以及谁（或什么）负责执行这些活动。

活动图的元素包括动作节点、控制节点和对象节点。有三种类型的控制节点：初始和终止（终止节点有两个变例：活动终止和流程终止）、判断和合并、分叉和结合[1]。

5.6.1 基本概念：动作

在活动图中，动作是行为的基本单元。活动可以包含许多动作，这就是活动图所展现的东西。图 5-27 展示了水培园艺系统案例中执行的一个动作。

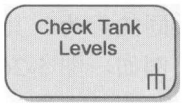


图 5-27 动作的表示法

注意动作表示法右下角的耙形符号，这表明它是一个 callBehavior 类型的动作。这是 UML 2 中预定义的动作之一，表示“为对象和链接操作、计算以及对象间的通信进行建模的简单动作”[101]。callBehavior 类型的动作会调用一个活动，该活动又由动作节点、控制节点和对象节点构成。因此，活动图建模中的大部分动作都会是这种类型，至少对于高层的活动图来说是这样的。所以，出于实际考虑，我们可能希望只在确实定义了要调用的活动时才使用耙形符号。

5.6.2 基本概念：开始和停止

既然活动图展示了一个处理流程，那么流程就必须有开始和结束的地方。活动流程的开始处（初始节点）被表示为一个实心圆点，结束处（终止节点）被表示为一个牛眼的样子。

图 5-28 展示了一个简单的活动图，它由一个动作——Check Tank Level 构成。

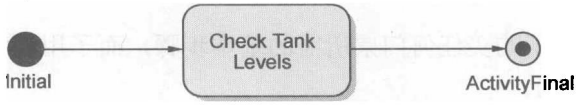


图 5-28 简单活动图的起始和终止节点

另一种类型的终止节点是流程终止节点，它由一个圆圈和一个“X”表示。流程终止节点用于停止一个流程，但不会停止整个活动，这将在合并节点的讨论中介绍。

5.6.3 基本概念：判断节点和合并节点

判断和合并节点控制了活动图中的流程。每种节点都由一个菱形来表示，带有进入和离开的箭头。判断节点具有一个进入流程和多个离开流程，其目的是将进入流程导向一个

[1] Conrad Bock 的 5 篇系列文章（*UML 2 Activity and Action Models*）详细地讨论了这个问题。（参见分类参考文献中的 L 部分。）

（且只有一个）离开流程。离开流程通常有一些警戒条件，以决定选择哪一条离开的路径。图 5-29 展示了警戒条件[all levels within tolerance]以及可选路径[else]。判断节点上没有等待或同步。

合并节点接受多个进入流程，并全部导向一个离开流程。合并节点上没有等待或同步。如图 5-30 所示，不论三个进入流程中的哪一个到达合并节点（显示为一个菱形），都会通过它被导向 Log System Event 动作。因此，多个事件都被记入日志。图 5-30 也展示了前面讨论过的流程终止节点。

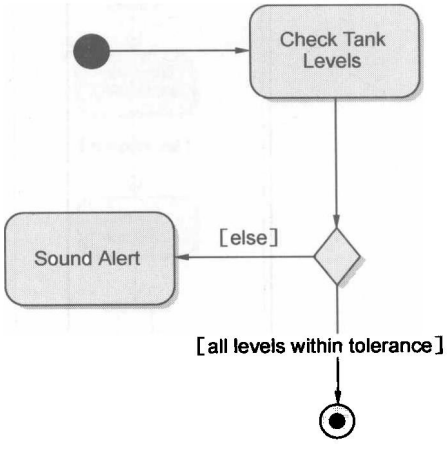


图 5-29 判断节点

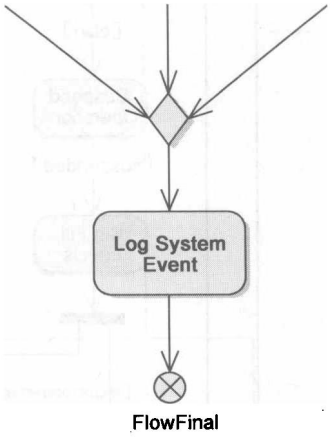


图 5-30 合并节点和流程终止节点

5.6.4 基本概念：分区

活动图中的元素可以利用分区来分组。分区的目的是说明执行具体活动的责任。在业务模型中，分区可以是一个业务单位、部门或组织机构；对于系统来说，分区可以是其他系统或子系统；在应用建模中，分区可以是应用中的对象。（但是，这种细粒度的交互通常会由序列图来表示，本章稍后将讨论。）每个分区都可以被命名，表示负责者。图 5-31 展示了水培园艺系统中，构成 Maintain Storage Tanks 用例的多个活动是如何被分为 Gardener、WaterTank 和 NutrientTank 等分区的。

5.6.5 高级概念：分叉、结合和并发

分叉节点和结合节点分别与判断节点和合并节点类似，关键的不同之处在于并发。分叉节点具有一个进入流程和多个离开流程，判断节点也是如此。不同之处在于，判断节点会选择一个离开流程，而进入分叉节点的一个流程会导致多个离开流程。所有离开流程是并发的。在图 5-31 中，来自 Set Fill Levels 动作的一个流程进入分叉节点，即第一根粗横线。此后，NutrientTank 流程（包含 Fill 动作）和 WaterTank 流程（包含 Disable Heating、Fill 和 Enable Heating 动作）是并行发生的。

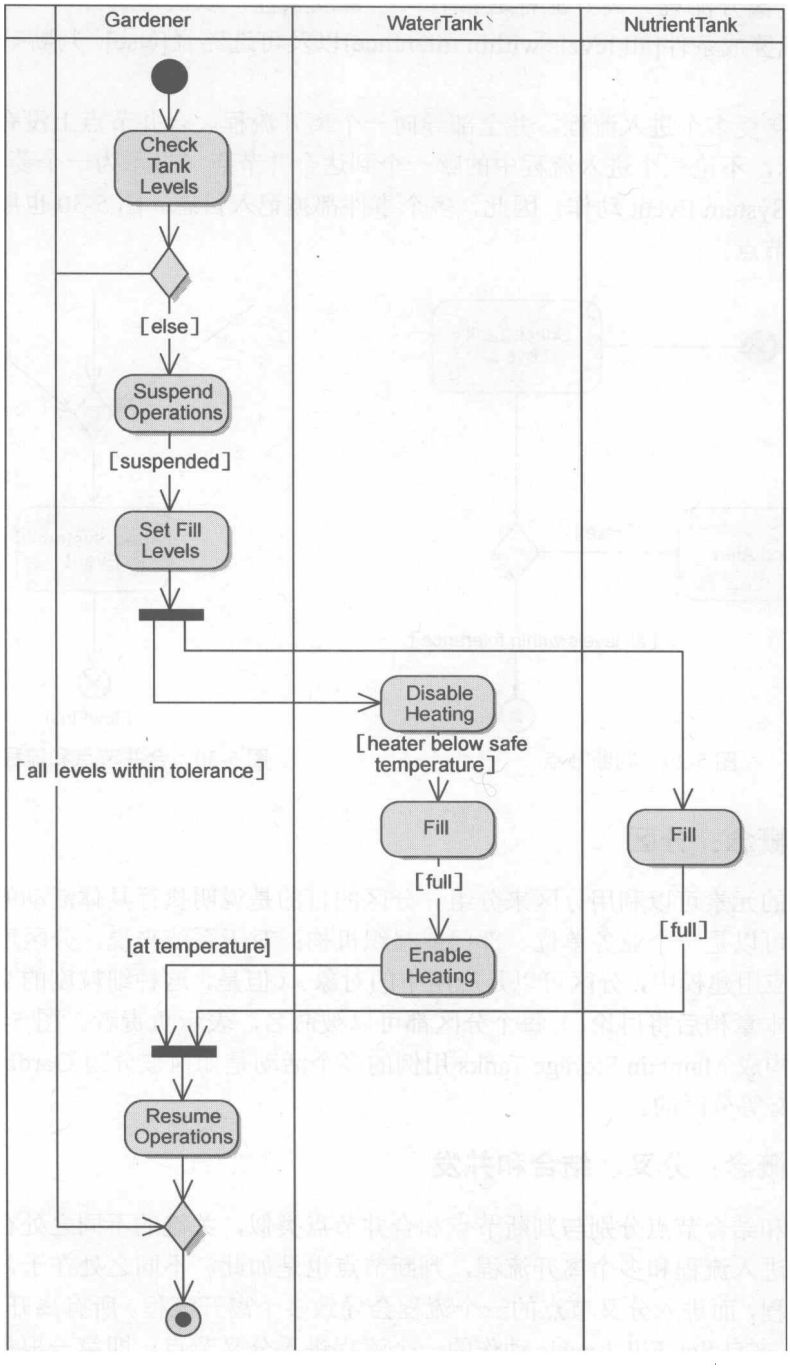


图 5-31 带有分区的活动图

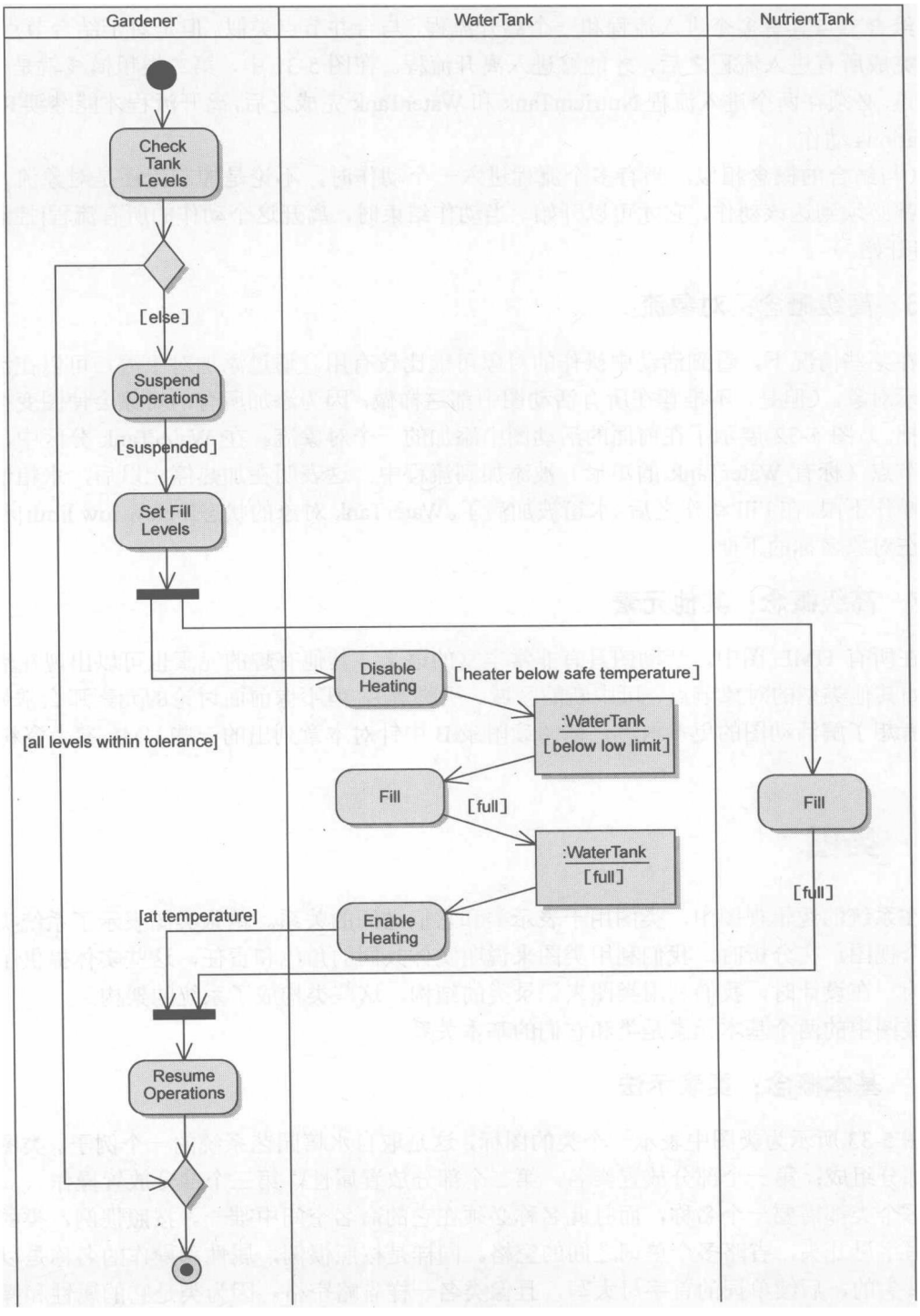


图 5-32 带有对象节点的活动图

结合节点具有多个进入流程和一个离开流程，与合并节点类似。但是对于结合节点来说，必须完成所有进入流程之后，才能够进入离开流程。在图 5-31 中，第二根粗横线就是一个结合节点。必须在两个进入流程 NutrientTank 和 WaterTank 完成之后，离开流程才能继续 Resume Operations 动作。

（与结合的概念相似，当有多个流程进入一个动作时，不论是控制流还是对象流，所有流程都必须到达该动作，它才可以开始。当动作结束时，离开这个动作的所有流程[控制和对象]将开始。）

5.6.6 高级概念：对象流

在某些情况下，看到活动中操作的对象可能比较有用。通过添加对象流，可以在活动图中显示对象。（但是，不推荐在所有活动图中都这样做，因为添加所有的对象会使图变得复杂而笨拙。）图 5-32 展示了在前面的活动图中添加的一个对象流。在 WaterTank 分区中，两个对象节点（标有 WaterTank 的矩形）被添加到流程中。这表明在加热停止以后，水箱低于了它的操作下限，在 Fill 动作之后，水箱被加满了。WaterTank 对象的状态[below low limit]和[full]显示在对象名称的下面。

5.6.7 高级概念：其他元素

在所有 UML 图中，活动图具有非常丰富的语义。其他有趣的元素也可以出现在活动图中（如其他类型的对象节点、可中断的区域、引脚等），但不像前面讨论的元素那么常见。如果读者想了解活动图的更多内容，请阅读附录 B 中针对本章列出的一些 UML 参考资料。

5.7 类图

在系统的逻辑视图中，类图用于表示类和它们之间的关系。单张类图表示了系统类结构的一个视图。在分析时，我们利用类图来说明实体共同的角色和责任，这些实体提供了系统的行为。在设计时，我们利用类图来记录类的结构，这些类构成了系统的架构。

类图中的两个基本元素是类和它们的基本关系。

5.7.1 基本概念：类表示法

图 5-33 所示为类图中表示一个类的图标，这是取自水培园艺系统的一个例子。类图标由三个部分组成：第一个部分放置类名，第二个部分放置属性，第三个部分放置操作。

每个类都需要一个名称，而且此名称必须在它的命名空间中唯一。按照惯例，类的名称以大写字母开头，省略多个单词之间的空格。同样是按照惯例，属性和操作的名称是以小写字母开头的，后续单词的首字母大写，且像类名一样省略空格。因为类是它的属性和操作的命名空间，属性名称在类的范围内必须无二义。根据所选择的实现语言的规则，操作名称也

必须如此。属性和操作规格说明的格式如下[29]。

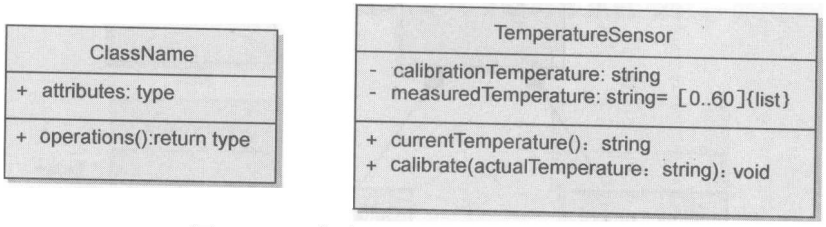


图 5-33 一般类图标和种植园系统的例子

- 属性规格说明格式：
  可见性 属性名称 ：类型 [多重性]= 默认值 {特性字符串}
- 操作规格说明格式：
  可见性 操作名称（参数名称 ：类型）：返回值 {特性字符串}

在后面的小节中，我们将讨论可见性的概念（公有可见性、私有可见性、保护可见性和包可见性）。对于属性和操作来说，类型是一个类或数据类型的名称。稍后也会讨论属性的多重性，现在请先看看图 5-33。在 TemperatureSensor 类中，measuredTemperature 属性的多重性是[0..60]，这表示一个包含 0～60 个温度测量值的数组。属性的默认值是在创建时给出的值，如果没有提供其他值的话。花括号中列出的特性字符串提供了额外的特性，例如 TemperatureSensor 类中 measuredTemperature 属性后面的{list}。在这个例子中，关键字 list 表示温度测量值是有序的，而且可能有重复值。这提供了一种手段来查看测量值的时间顺序，并允许测量的温度值出现重复。在操作的格式中，“参数名 ：类型”的组合根据需要可以重复，包含几个参数。

对于特定的类图，显示一个类的某些属性和操作是有用的。我们说“某些”是因为，对于凡是具有一点重要性的类，在一张类图中显示它的所有属性既不方便，也不必要。出于这个原因，我们显示的属性和操作是整个类规格说明的一个编辑过的视图。类的完整规格说明才是声明所有类成员的唯一位置。如果我们需要显示许多成员，可以放大类图标；如果我们决定不显示这样的成员，可以去掉分隔线，只显示类名。

作为表示法的一般原则，属性和操作这样的元素的语法可以根据所选的实现语言的语法来定制。这分离了不同语言的特性，简化了表示法。

第 3 章中曾提到，抽象类是不能创建实例的类。因为这样的类对于构建良好的类层次结构非常重要，所以有一种特殊的方式来表示抽象类，如图 5-34 所示。具体来说，用斜体来显示类名，表明只能为它的子类创建实例。类似地，为了表明一个操作是抽象的，用斜体来显示操作名称，这意味着这个操作可以在它的子类中以不同的方式实现。在水培园艺系统中，我们有一些食物含有特定的维生素和热量当量，但没有一种类型的食物被称为“食物项”。因此，FoodItem 类是抽象的。图 5-34 也展示了子类 Tomato，它代表一种在温室中生成的具体食物（可实例化）[30]。接下来将解释类之间关系的含义。

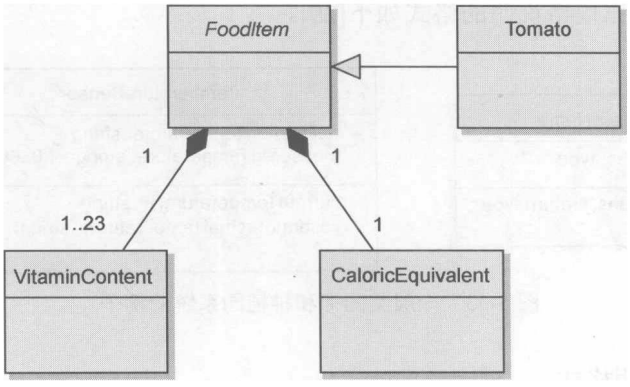


图 5-34 抽象类表示

5.7.2 基本概念：类关系

类很少是独立的，相反，它们会通过不同的方式与其他类协作。类之间的基本联系包括关联、泛化、聚合和组合，图 5-35 总结了它们的图标。每种关系都可以包含一个文本标签，记录关系的名称或者说明它的目的，关联端也可以有名称，但通常不会同时存在。

关联图标连接了两个类，体现了一种语义联系。关联通常用名词词组来标注，如图 5-35 中的 Analyzes，以说明关系的实质。类可能与它自己有关联（称为自关联），如 PlanAnalyst 类的实例之间的协作。注意，这里同时使用了关联端名称和关联名称，目的是提供清晰性。也有可能在同一对类之间存在多个关联。关联可以进一步通过多重性来修饰，第 3 章中曾介绍过，多重性的语法的例子如下：

- 精确 1 个；
- 数目不限（0 个或多个）；
- 0..* 0 个或多个；
- 1..* 1 个或多个；
- 0..1 0 个或 1 个；
- 3..7 指定范围（3～7，包含 3 和 7）。

多重性应用于关联的目标端，说明源类的每个实例与目标类实例的连接个数。除非显示说明，否则关系的多重性就是未指定的。通常最好是显示多重性，因为这样就不会引起误解。

其他三个基本类关系是对一般关联图标的精化。实际上，在开发过程中，这正是关系演变的一种趋势。我们先断定两个类之间存在语义上的联系，然后随着具体地确定它们之间联系的实质，常常会将关联精化为泛化、聚合和组合关系。

泛化图标表示一种泛化/特化关系（即第 3 章中描述的“是一种”关系），表现为带有封闭箭头的关联。箭头指向超类，关联的另一端是子类。图 5-35 中的 GrowingPlan 类是超类，它的子类是 FruitGrowingPlan。根据所选实现语言的规则，子类继承了超类的结构和行为。同样，根据这些规则一个类可以有一个（单继承）或多个（多继承）超类，超类间的名字冲

突也根据所选语言的规则来处理。另外，泛化关系不能有多重性指定。

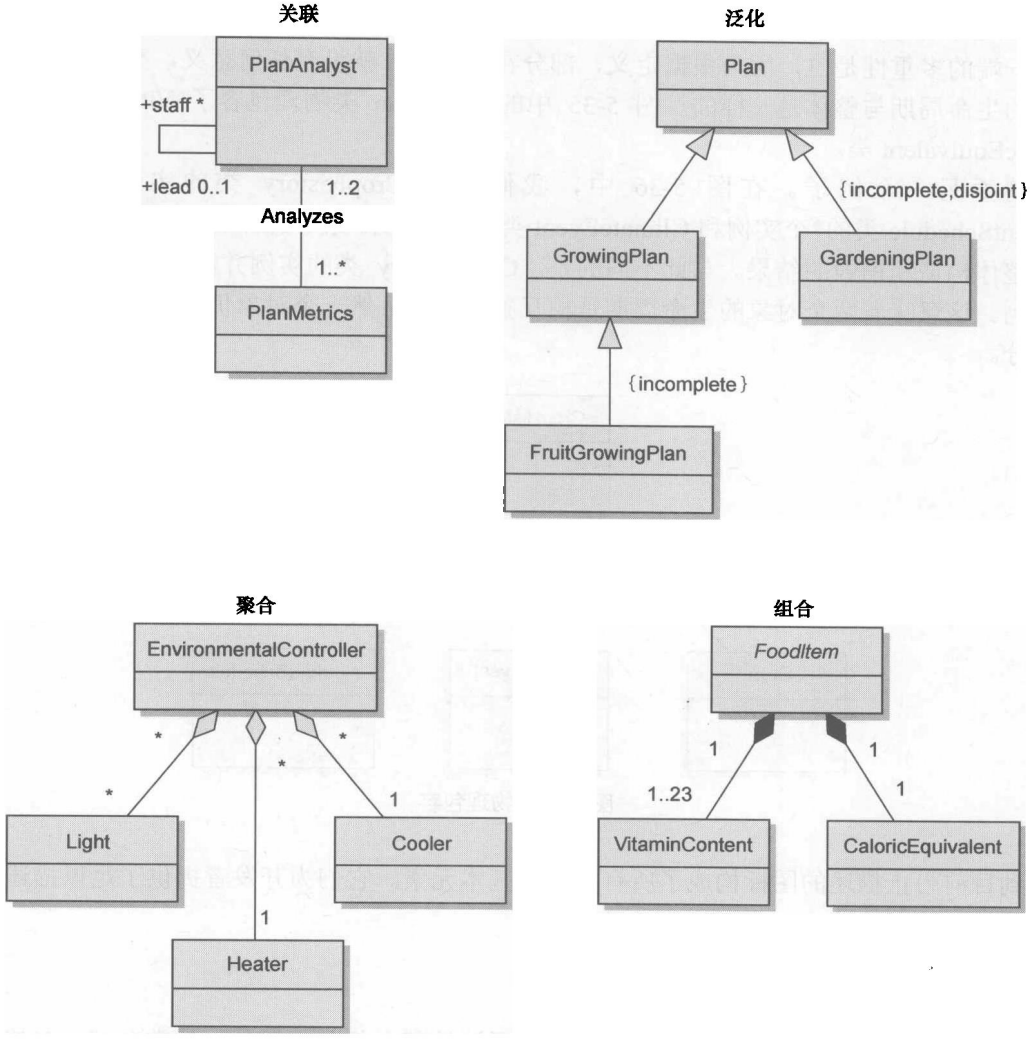


图 5-35 类关系图标

第 3 章中曾指出，关联体现了“组成部分”关系，它是一般的关联关系的一种约束形式。聚合图标表明一种整体/部分层次结构，也意味着能够从聚合体导航到它的部分。它表现为带有一个空心菱形的关联，菱形所在的一端是聚合体（整体）。另一端的类代表它的实例构成了聚合对象的部分。自聚合和循环聚合关系是可能的。这种整体/部分的层次关系并不意味着物理上的包容：一个专业协会有一些成员，但不表示协会拥有它的成员。在图 5-35 中我们看到，EnvironmentalController 类有 Light、Heater 和 Cooler 类作为它的部分。聚合关系末端的*（0 个或多个）多重性进一步突出了这不是物理包容关系。

选择聚合通常是分析或架构设计时的决定，选择组合（物理包容）通常是具体的、战术上的问题。区分物理包容是很重要的，因为在构建和销毁聚合体的部分时，它的语义会起作用。组合图标表示一种包容关系，表现为带有一个实心菱形的关联，菱形所在的一端是整体。在这一端的多重性是 1，因为根据定义，部分在整体之外就没有任何意义，整体拥有部分，部分的生命周期与整体是一样的。图 5-35 中的 FoodItem 类物理包容了 VitaminContent 和 CaloricEquivalent 类。

请看另一个例子。在图 5-36 中，我们看到 CropHistory 类的实例物理包容了 NutrientSchedule 类的*个实例和 ClimateEvent 类的*个实例。组合意味着这些部分的构造和销毁是整体构造和销毁的结果。与此不同的是，CropHistory 类的实例并没有物理上包容 Crop 的实例。这意味着两个对象的生命周期是相互独立的，虽然一个对象仍被看作是另一个对象的部分。

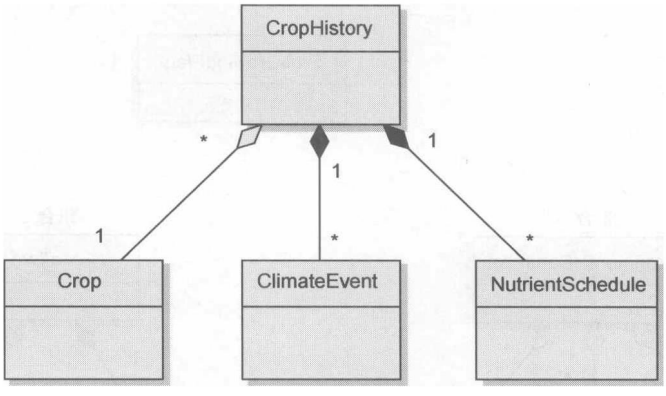


图 5-36 物理包容

到目前为止描述的图标构成了所有类图的基本元素。它们为开发者提供了足以描述系统类结构基础的表示法。

5.7.3 高级概念：模板（参数化）类

到目前为止，我们介绍的元素构成了类表示法的基本部分。但是，常常会有一些战略上和战术上的决定，需要利用基本表示法的扩展来记录。作为一般原则，我们需要坚持使用表示法的基本元素，只使用那些必要的高级概念来表示一些分析或设计的细节，而且前提是这些细节对于系统的展现和理解很重要。

某些面向对象编程语言，如 C++，提供了模板（参数化）类。模板类代表了一个类家族，它们的结构和行为与形式化的类参数是分别定义的。我们必须将这些形式化的参数映射到某一个具体的类（绑定的过程），然后才能得到这个类家族中的一个具体的类。所谓具体的类，指的是它能够拥有实例。

模板类与普通类有很大的不同，所以需要一种特殊的修饰。在图 5-37 所示的例子中，模

板类表现为一个简单的类，但在它的右上角有一个虚线框，其中包含了它的形式化参数。绑定的类也表现为一个简单的类。模板类和它的绑定类之间的绑定关系用一个虚线箭头来表示，箭头指向模板类，带有关键词<<bind>>。实际的参数与模板的形式参数绑定，它会单独显示，带有形如<Formal Parameter→Actual Parameter>的关键词。在图 5-37 中我们看到，PlanSet 类与 Set 模板类绑定，GardeningPlan 类是实参，代替了形参 Item。

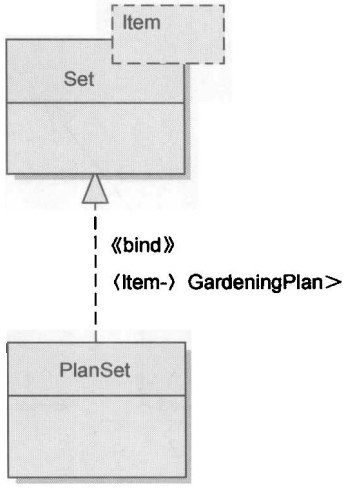


图 5-37 模板类和它绑定的类

模板类不能够拥有任何实例，也不能作为模板单独使用。绑定类定义了一种新类，与同一家族中的其他类都不同，这些类具有不同的实参。

5.7.4 高级概念：可见性

在 5.2 节中，我们从包中包含的元素能否在包外可见的角度讨论了可见性的概念。对于类中包含的元素，类也提供了封闭的命名空间。下面来看类关联、属性和操作的可见性。

所有有意思的面向对象编程语言都提供了类接口和类实现的清晰分离。第 3 章中曾提到，绝大多数面向对象编程语言允许开发者指定对接口的更细粒度的访问控制。例如在 C++中，成员可以是公有的（所有客户都能访问）、保护的（只有子类、友元或该类本身能访问）或私有的（只有该类本身或它的友元能访问）。在 C++中，某些元素可能是类实现的一部分，所以其即使对该类的友元也不可见，这被称为实现可见性。

可以指定利用以下的符号指定相应元素的可见性[43]：

- 公有可见性（+） 对能看到这个类的任何元素都可见；
- 保护可见性（#） 对这个类及其子类的其他元素可见；
- 私有可见性（-） 对这个类的其他元素可见；
- 包可见性（~） 对同一个包中的其他元素可见。

我们在关联端名称上加上这些可见性符号，说明关联的保见性，表示从源端到目标端的

访问。例如，在图 5-38 中可以看到，位于 CropDatabase 类和 GrainCrop 类之间关联端名称（database 和 crop）都是公有的。这意味着每个类都可以访问另一个类。与之不同，请看 GrainCrop 类和 GrainYieldPredictor 类之间的关联端名称的可见性，GrainCrop 对 GrainYieldPredictor 类是私有的。

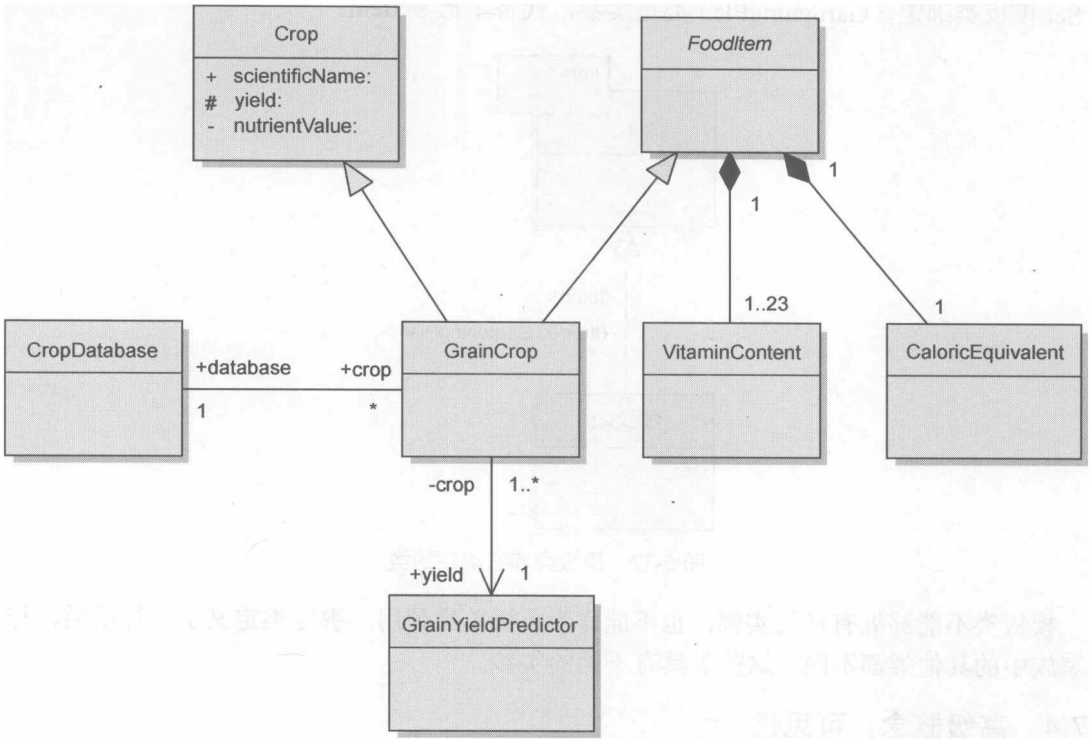


图 5-38 类可见性

这里要介绍的另一个高级概念是关联的方向性。在分析时，我们认为关联是分析类之间的双向逻辑连接。在设计时，我们将关注的焦点转到关联的导航性上。从 GrainCrop 类到 GrainYieldPredictor 类的单向关联通常意味着 GrainCrop 类的某些方法在实现时使用了 GrainYieldPredictor 类的服务。

这些可见性符号也适用于嵌套的实体，形式完全一样。具体来说，在类的图标中，我们可以在属性和操作名称前面加上可见性符号，说明属性和操作的可访问性。例如，在图 5-38 中，我们看到类 Crop 具有一个公有属性（scientificName）、一个保护属性（yield）及一个私有属性（nutrientValue）。

5.7.5 高级概念：关联端名称和限定符

在前面的章节中，我们描述了对象在与其他对象协作时所扮演的不同角色的重要性。简而言之，一种抽象的角色是它在给定的时刻向世界展现的面目。在类与类的关联中，角色说

明了类的目的或能力。如图 5-39 所示，这个角色用一个关联端名称来描述（UML 1 中的角色名称），该名称被放在扮演这个角色的类旁边。这里我们看到，PlanAnalyst 类和 Nutritionist 类都是 CropEncyclopedia 类的贡献者，这意味着它们都为百科全书添加信息。从在百科全书中寻找信息的角度来看，PlanAnalyst 类也是一个用户。在每一种情况下，客户扮演一种角色，这个角色确定了它与提供者之间的具体行为和协议。注意 PlanAnalyst 类的自关联：这里我们展示了这个类的多个实例之间可以互相协作，它们在协作时使用特定的协议，这种协议和它们与其他类（如 CropEncyclopedia）之间的行为是有区别的。

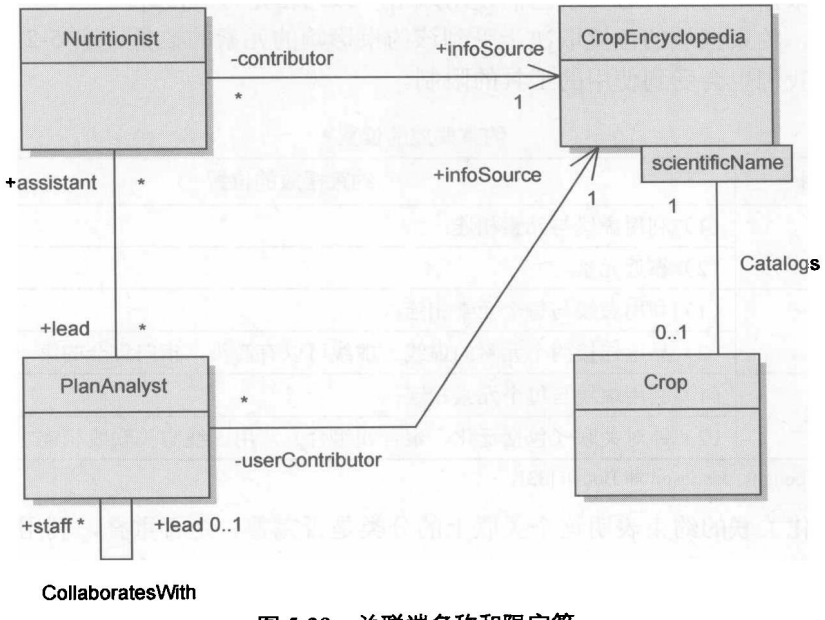


图 5-39 关联端名称和限定符

我们的例子也展示了 CropEncyclopedia 类和 Crop 类之间的关联，但使用了一种不同的修饰符号，这表示一个限定符，它显示为关联在 CropEncyclopedia 端的一个小矩形。限定符是一个属性，它的值唯一地确定了一个目标对象。

在这个例子中，CropEncyclopedia 类利用 scientificName 属性作为限定符来导航 CropEncyclopedia 的实例管理的集合项的每个元素。一般来说，限定符必须是对象的一个属性，它是关联目标端的聚合对象的一个部分。多重限定符是可能的，但限定符的值必须唯一。在没有限定符的情况下，关联的 Crop 一端的多重性会是 0 或更多（*）。

5.7.6 高级概念：约束

约束是必须保持的某些语义条件的表达式。换言之，约束是类或关系中的不变式，当系统处于稳定状态时必须保持。我们强调是“稳定状态”，因为可能存在系统改变的短暂瞬间（因此暂时处于内部不一致的状态），在这些瞬间无法保持系统的约束。只有当系统状态稳定时，约

束才是保证适用的。进入条件和退出条件就是约束的例子，它们在系统处于稳定状态，也就是在操作调用和完成的特定时间点是适用的[31]。

在设计系统的过程中，必须确保能够满足系统的约束。我们会在各种建模元素上应用约束，通常，任何元素都可以加上约束。我们使用一个包含表达式的约束修饰符号，将其放在花括号（{}）中，使其位于约束适用的类或关系附近。这种表达式可以用自然语言（文本）表示，也可以用更形式化的语言，如编程语言或 UML 对象约束语言（OCL）来表示。使用更形式化的语言的好处在于，某些工具提供了一些手段来验证约束是否得到满足。通常，开发团队用文本来指定约束，这要求它们使用风格一致的指定方式[32]。

在视图中，约束摆放的位置取决于受到该约束影响的元素的数目。表 5-2 提供了通用的指南。当然，我们也会受到使用的工具的限制。

表 5-2　　约束摆放的位置[a]

元素的数目	约束摆放的位置
1 个	（1）利用虚线与元素相连；
	（2）靠近元素
2 个	（1）利用虚线与每个元素相连；
	（2）靠近连接两个元素的虚线。虚线可以有箭头，指向集合的第一个位置
3 个或更多	（1）利用虚线与每个元素相连；
	（2）针对关联（包括泛化、聚合和组合），用虚线与关联线相连

a. 基于 Rambough、Jacobson 和 Booch [33]。

应用于泛化关联的约束表明这个关联上的分类是否完整、是否重叠，利用下面定义的 4 种约束[34]：

- Complete　超类的一个实例至少是子类的一个实例；
- Imcomplete　超类的一个实例可能不是子类的一个实例；
- Disjoint　不同分类之间没有共同实例；
- Overlapping　不同分类之间有共同实例。

回到图 5-35，来看看这些约束的例子。Plan 类的泛化上的{incomplete}约束表明，还可以有其他类型的计划，而不仅仅是培育计划和园艺计划。这意味着，Plan 类的实例可能既不是 GrowingPlan 类的实例，也不是 GardeningPlan 类的实例。{disjoint}约束表明，一份计划不可能既是培育计划，又是园艺计划，至少我们是以这种方式来定义计划的。也就是说，GrowingPlan 类的实例不可能是 GardeningPlan 类的实例。

在类图中，可以看到其他类型的约束，与类本身有关。这包括在类属性上的约束，如前面在图 5-33 中讨论过的，TemperatureSensor 类的 measuredTemperature 属性有{list}约束。这表明温度测量值是有序的，可能发生重复。

图 5-40 中的例子表明，我们可以将约束应用于单个类、整个关联以及关联中的执行者。在这个图中，我们看到 EnvironmentalController 类上的基数约束，表明系统中这个类的实例

不能超过 7 个。

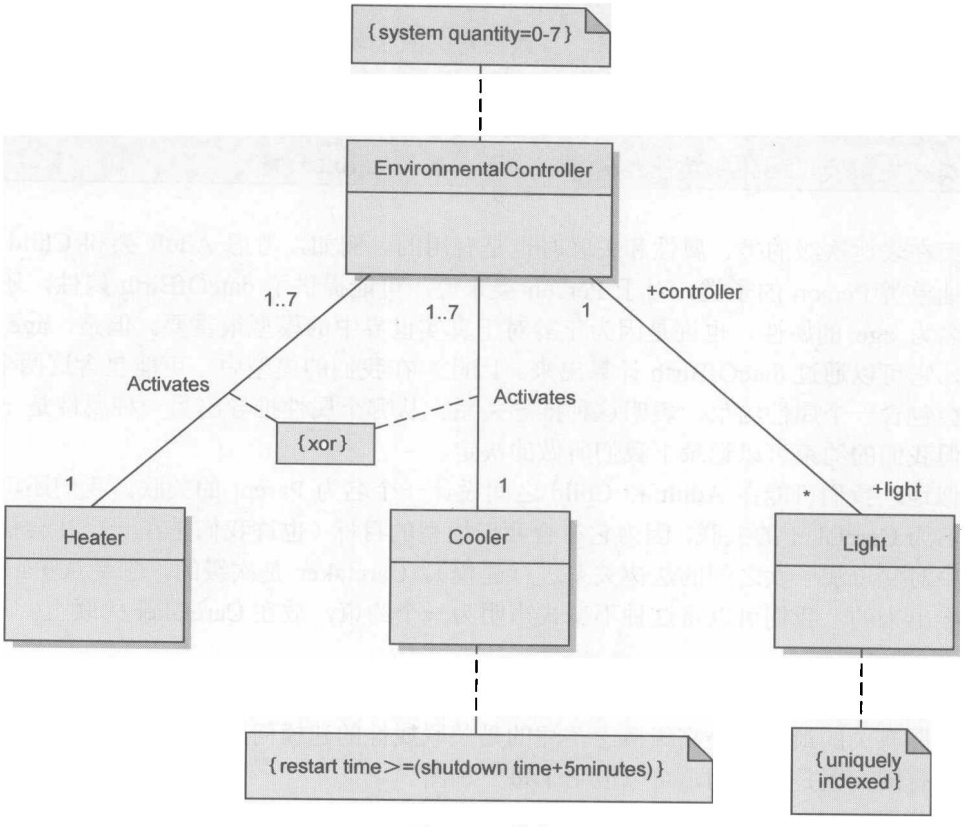


图 5-40 约束

Cooler 类上有另一种约束。这里我们看到冷却器操作的延迟说明——冷却器关闭后 5 分钟之内，不能重新启动。将这个约束加在 Cooler 类上是因为，我们将这一点作为类实例本身应该保持的不变式。

在图 5-40 中，我们也看到了两种不同类型的关联约束。在 EnvironmentalController 类和 Light 类之间的关联上，要求在这个关联中每个灯必须唯一地被编号。还有一个异或约束 {xor}，位于指向 Heater 类和 Cooler 类的控制器关联上。这表明了 EnvironmentalController 类上的一个不变式，即不能同时激活加热器和冷却器。将这个约束放在关联上而不是作为 Heater 类或 Cooler 类的约束，是因为这个不变式不能由加热器或冷却器自己来保持。

类关系的精化

前面，在讨论类关系图标时我们看到，泛化、聚合和组合是类之间更一般关系的精化。具体来说，当我们来看聚合时，图 5-35 展示的 EnvironmentalController 类是一个聚

合整体，Light、Heater 和 Cooler 类是它的部分。现在，在图 5-40 中我们看到，聚合被一般的关联关系所取代。发生了什么事情？图 5-40 可能表达了对这些类之间的关系的早期分析观点。在稍后的开发工作中，当我们对关系的确切实质进行战术上的决定时，我们会把这些关系精化为聚合。如图 5-35 所示，我们可能认为 EnvironmentalController 类更像是由 Light、Heater 和 Cooler 等类组成的子系统。我们可能需要在子系统中加入其他的类来提供真正的控制功能，例如，加入一个 Controller 类。

约束在表达次级的类、属性和关联时也是有用的。例如，考虑 Adult 类和 Child 类，它们都是抽象类 Person 的子类。对于 Person 类来说，可能提供了 dateOfBirth 属性，还可能包含一个名为 age 的属性，也许是因为年龄对于真实世界中的模型很重要。但是，age 属性是次级的，它可以通过 dateOfBirth 计算出来。因此，在我们的模型中，可能包含这两个属性，但是还会包含一个属性约束，表明这种推导关系。从哪个属性推导出另一种属性是一种战术决策，但我们的约束可以记录下我们所做的决定。

类似地，我们可能在 Adult 和 Child 之间设计一个名为 Parent 的关联。我们还可能设计另一个名为 Caretaker 的关联，因为它符合我们模型的目标（也许我们是在一个社会福利系统的分析中对父母和子女之间的法律关系进行建模）。Caretaker 是次级的，它是从 Parent 关联中被推导出来的，我们可以将这种不变式声明为一个约束，放在 Caretaker 关联上。

5.7.7 高级概念：关联类和注解

与类图有关的最后一个高级概念关注的是关联属性的建模问题，对这个具体问题的表示法对于所有图元素是通用的，可以应用于每一种图。

请考虑图 5-41 中的例子。Crop 和 Nutrient 之间的多对多关联表示每种作物依赖于 N 种营养，每种营养可以提供给 N 种不同的作物。NutrientSchedule 类实际上是这个多对多关系的一个属性。为了表明这个语义事实，我们从 Crop 到 Nutrient 的关联（带属性的关联）出发，画一条虚线，指向它的属性，即 NutrientSchedule 类（关联的属性）。每一个关联至多拥有一个这样的属性，它被称为“关联类”，其中关联的名称必须与作为属性的类名称匹配。

这种带属性的关联有一种泛化的表示方式。具体来说，在分析和设计时，存在大量似乎是随意的假定和决定，每个开发者都可能收集这些假定和决定。由于通常没有方便的地方来存放它们，只能记在开发者的脑子里，这些见解常常会丢失——这显然是一种不可靠的做法。因此，对图中的任意元素添加任意的注解是有用的，注解的文字记录了这些假定和决定。在图 5-41 中，有两个这样的注解。与 NutrientSchedule 类相连的注解告诉我们，它的实例预期是唯一的。另一个注解与 Nutrient 类相连，记录了我们预期 unitCost 属性将怎样填充。

对于这样的注解，我们使用特别的注解形状的图标，用虚线将它与它影响的元素连起来。取决于所用的工具，注解可能包含任何信息，包括纯文本、代码片段或对其他文档的引用。注解可能不与任何图中的元素相连，这表示它适用于整张图。

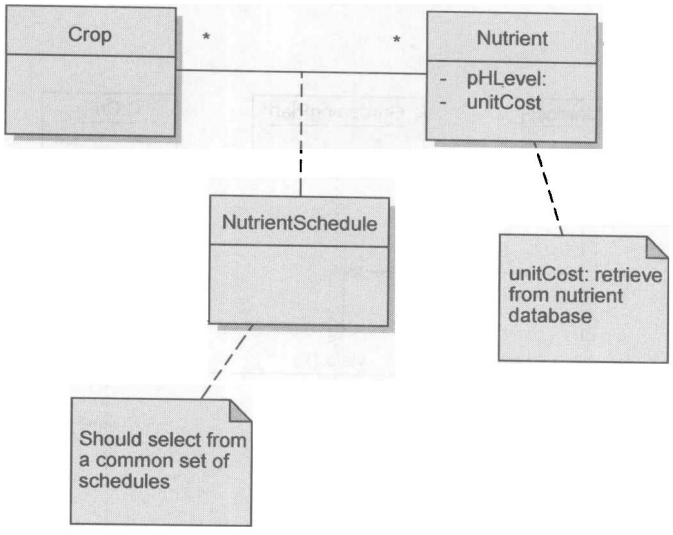


图 5-41 关联类和注解

5.8 序列图

作为一种通信图，序列图用于跟踪在同一个上下文环境中一个场景的执行。（通信图将在本章稍后讨论。）实际上，序列图在很大程度上就是通信图的另一种表现形式。

5.8.1 基本概念：对象与交互

在图 5-42 中提供了一个序列图，它重复展示了下面通信图的绝大部分语义。使用序列图的好处在于比较容易读出消息传递的相对次序。在开发生命周期的早期，在单个类的协议确定之前，序列图常常比对象图（在本章稍后讨论）更适合记录场景的语义。早期的序列图倾向于关注事件而不是操作，因为事件有助于确定被开发系统的边界。使用对象图的好处在于，它可以扩展到许多对象，包含复杂的调用。每种图都有自己的比较优势。

5.8.2 基本概念：生命线与消息

在序列图中，我们感兴趣的实体（与对象图中的对象一样）被水平地放在图的顶部。垂直的虚线称为“生命线”，画在每个对象下面，表示对象的存在。

消息（它可能表示事件或操作的调用）被画成水平的。消息图标的端点与垂直线相连，这些垂直线又与图顶部的实体相连。消息从发出者指向接收者。次序由垂直位置来表示，第一个消息出现在图的顶部，最后一个消息出现在图的底部。因此，就不需要顺序编号了。

消息的表示法（如线的类型和箭头的类型）说明了消息的类型，如图 5-43 所示。

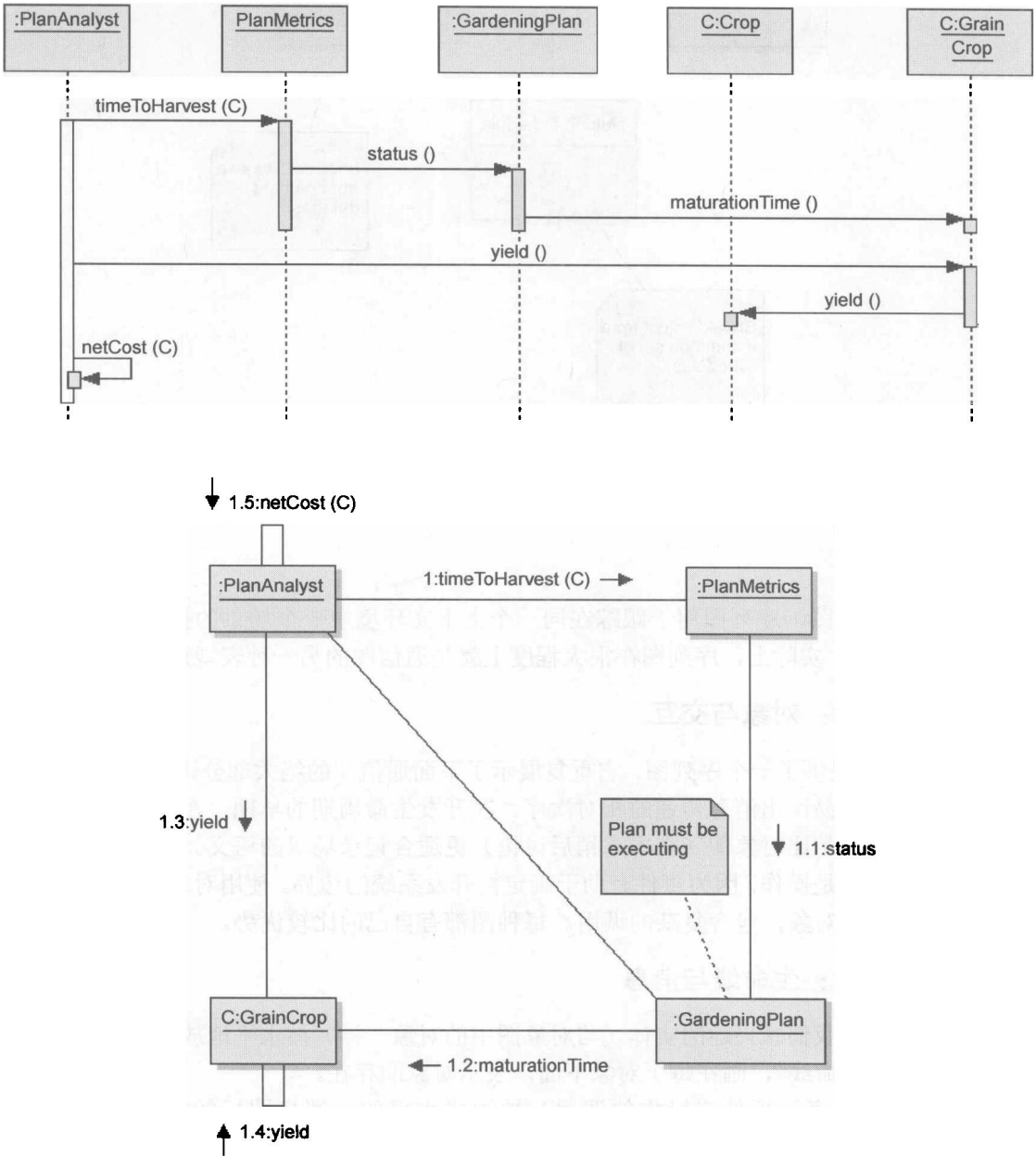


图 5-42 序列图（顶部）及其相关的通信图（底部）

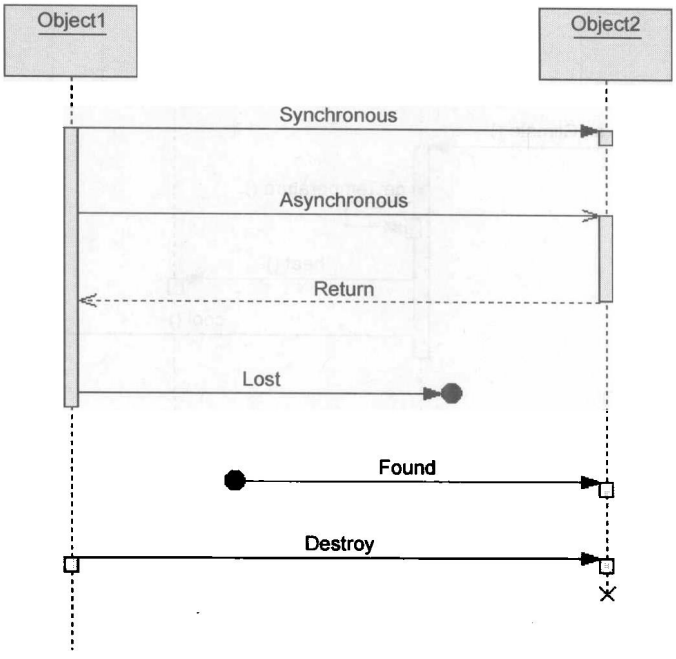


图 5-43 消息类型的表示法

同步消息（通常是一次操作调用）显示为一条带实心箭头的实线。异步消息是一条带开放箭头的实线。返回的消息是一条带开放箭头的虚线。丢失的消息（没有到达其目的地的消息）显示为一个同步消息，以端点（一个黑圆点）终止。发现的消息（发送者未知的消息）显示为一个同步消息，起点是一个端点符号。

5.8.3 高级概念：销毁事件

销毁事件表明何时一个对象被销毁，它表示为生命线末端的一个 X。请看图 5-43 中 Object2 生命线的例子。如果对象是一个组合对象，那么相关的对象也会被销毁。

序列图在概念上非常简单，但是可以加入其他元素，使它在某些复杂的交互模式中更具表现力。

5.8.4 高级概念：执行说明

简单的序列图可能没有说明消息传递时控制的焦点。例如，对象 A 向其他对象传递消息 X 和 Y，我们可能不清楚 X 和 Y 是来自 A 的独立消息，还是作为同一个封装消息 Z 中的一部分。如图 5-42 和图 5-43 所示，为了澄清这种情况，可以对序列图中从每个对象垂下的竖线进行修饰，加上一个盒子，表示控制流的焦点在这个对象上。例如，在图 5-44 中，我们看到 GardeningPlan 的匿名实例是最终的控制焦点，它的行为是执行一个气候计划并调用其他方法，这些方法又调用其他方法，最后将控制返回给 GardeningPlan 对象。

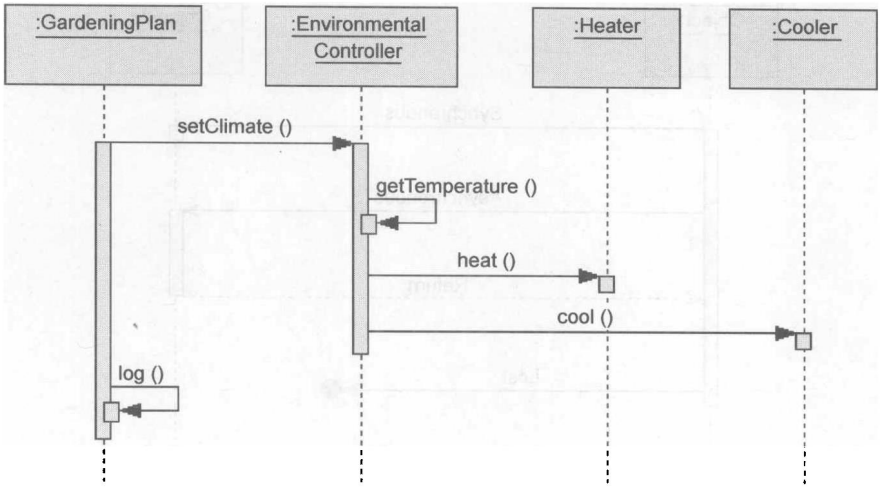


图 5-44 执行说明

脚 本

虽然不是 UML 的正式组成部分，但可能看到在序列图中使用描述性的文字。对于涉及条件或迭代的复杂场景来说，序列图可以通过使用脚本来增强。如图 5-45 中的例子所示，脚本可以写在序列图的左边，脚本的步骤与消息调用对齐。

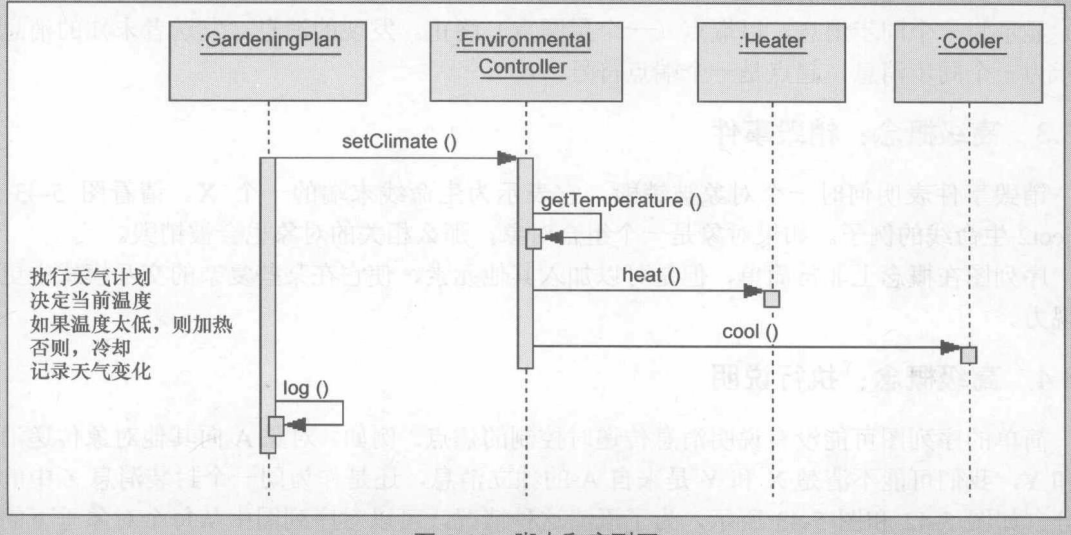


图 5-45 脚本和序列图

脚本可以用自由格式编写，也可以采用结构化的英语文本或者采用所选实现语言的语法。

5.8.5 高级概念：交互使用

UML 2.0 拥有各种结构来简单化复杂的序列图。首先要介绍的就是“交互使用”。交互使用是一种手段，用于说明在一个序列图中想复用别处定义的一个交互场景。如图 5-46 所示，它表现为一个带有 ref 标签的框。

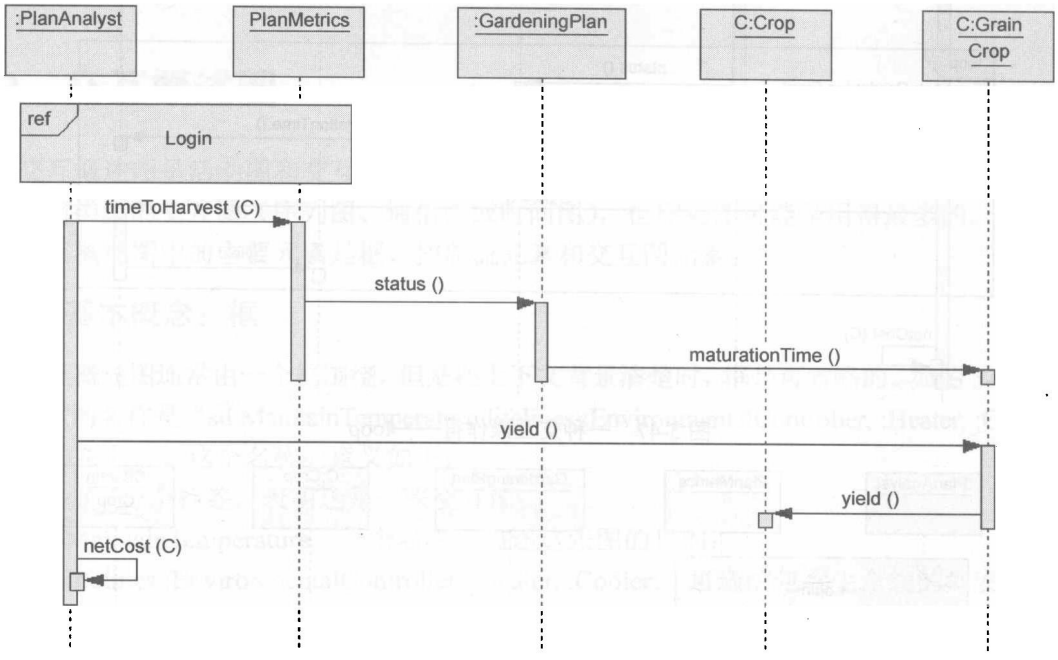


图 5-46 一种交互使用——Login

在这个例子中，我们修改了前面的序列图，引入了一个登录序列，要求在 PlanAnalyst 使用系统之前完成。这个带有 ref 标签的框表明，在这个序列中放置这个框的位置插入了 Login 序列（即拷贝了这个序列）。实际的登录序列会在另一张序列图中定义。

5.8.6 高级概念：控制结构

正如我们所看到的，序列片段可以用来简化序列图，也可以用来表示序列图中的流程控制结构。例如，图 5-47 展示了在序列图中引入一个循环。

请注意，这个框是以交互操作符 loop 命名的。在这个框内执行的序列是由条件[For each Gardening Plan]所控制的。

现在，让我们假定存在许多 GardeningPlan 对象，有些是被激活的，有些不是被激活的（过去的计划现在被保存起来，只是为了提供信息）。我们不希望对所有这些不被激活的计划执行循环，只希望对当前已被激活的计划执行循环。这可以通过 alt（“alternatives”的缩写）交互操作符来实现，如图 5-48 所示。

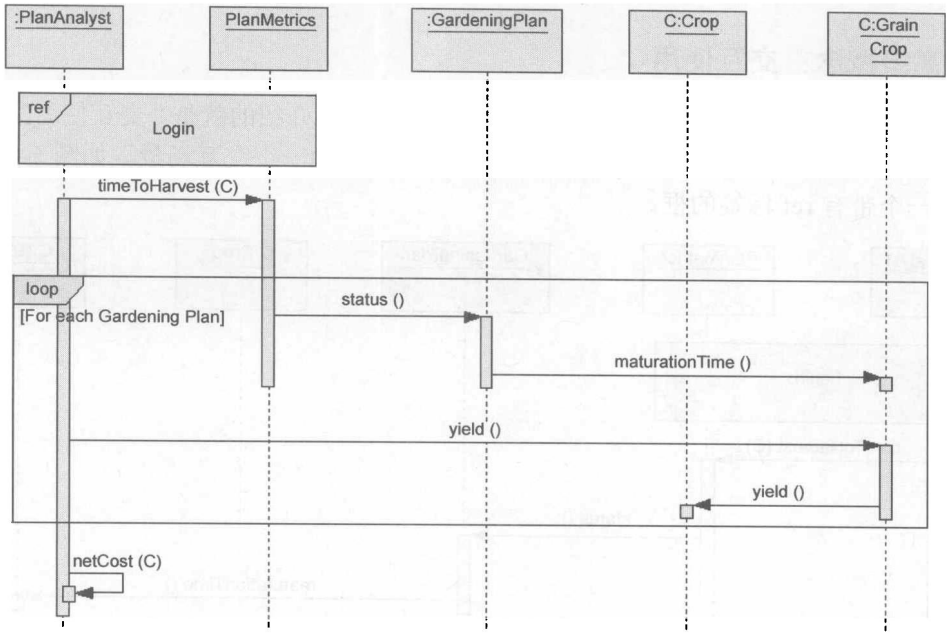


图 5-47 一种交互操作符——loop

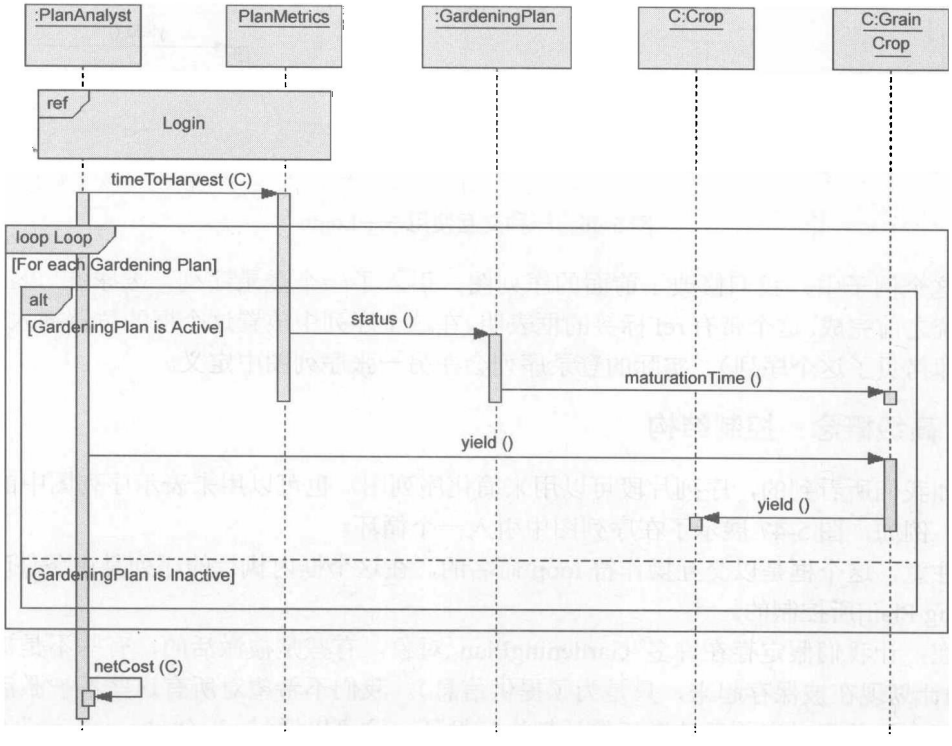


图 5-48 一种交互操作符——alt

在这个循环中，做了一个选择，由条件[GardeningPlan is Active]和[GardeningPlan is Inactive]所控制。这些条件选择执行序列的哪一部分。alt 框被分成两个区域，每个区域都有自己的条件。当条件为真时，框中这个区域的行为就得到执行。

还有一些其他交互操作符可以以这种方式使用，来控制序列图中的控制流。请参考附录 B 以了解所有其他的可选项。

5.9 交互概述图

交互概述图是活动图和交互图的结合，目的是概述交互图元素之间的控制流。虽然可以使用任何类型的交互图（序列图、通信图或时间图），但序列图可能是用得最多的。

交互概述图中的主要元素是框、控制流元素和交互图元素。

5.9.1 基本概念：框

交互概述图通常由一个框围绕，但是在上下文背景清楚时，框是可省略的。如图 5-49 所示，围绕的框的名称是“sd MaintainTemperature lifelines :EnvironmentalController, :Heater, :Cooler”，位于框的左上角。这个名称的意义如下：

- sd　一个标签，表明这是一张交互图；
- MaintainTemperature　一个名称，描述这张图的目的；
- lifelines :EnvironmentalController, :Heater, :Cooler;　可选的包含生命线的列表。

这个交互概述图包含了控制流和三个框，即 EvaluateTemperature、Heat 和 Cool，我们将在稍后讨论。

5.9.2 基本概念：控制流元素

交互概述图中的控制流是由活动图元素的组合来实现的，提供了可选路径和并行路径。可选路径控制是由判断节点的组合来实现的，控制流在判断节点选择合适的路径，对应的合并节点（如果需要）将所有可选路径重新会聚在一起。

这种组合在图 5-49 中出现了两次。第一次使用了一个判断节点，根据水培园艺系统的温度是否在边界之内来选择路径——在边界之内就不采取任何行动，不在边界之内就需要加热或制冷。我们使用了交互约束条件[lower bound <= temp <= upper bound]来选择合适的路径。第二次是组合使用判断节点和合并节点，控制是加热还是制冷，应用两个交互约束条件[temp < lower bound]和[temp > upper bound]。

并行路径的控制流是通过组合分叉节点和结合节点来实现的，分叉结点将控制流分成并行的路径，结合节点将并行的路径会聚在一起。关于并行路径，有一点很重要，即来自所有路径的控制必须到达结合节点，控制流才能继续下去。这要求我们确保，如果存在交互约束条件可能导致一条路径阻塞，一定要有另一条可选路径能让控制流继续下去。

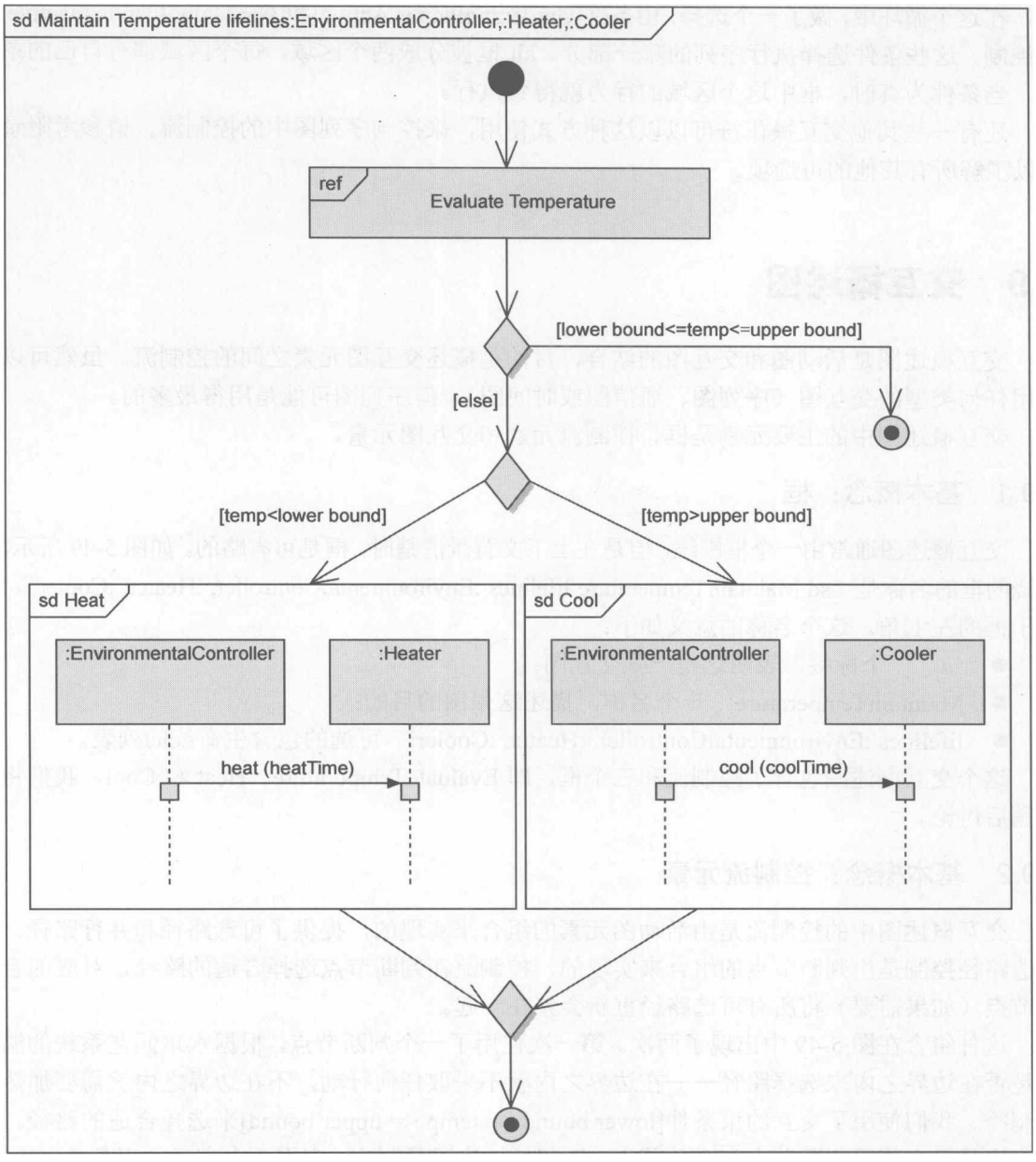


图 5-49 MaintainTemperature 的交互概述图

5.9.3 基本概念：交互图元素

交互概述图包含了两种类型的元素，目的是提供交互图的信息，它们分别是交互或交互使用。交互可以是任何类型的交互图，它提供了交互的内部细节。它们通常是序列图。它们

可以是匿名的或命名的，图 5-49 展示了 Heat 和 Cool 交互。

交互使用引用了一个交互图，而没有提供它的细节。图 5-49 中包含了一个例子，即 EvaluateTemperature 交互使用。EvaluateTemperature 的细节将展示如何管理如下一些考虑：

- 定期读取温度；
- 保护压缩机，在:Cooler 关闭后 5 分钟内不会重新启动；
- 基于每天的不同时间调整温度；
- 不同作用的温度变动范围。

5.10 组合结构图

组合结构图提供了一种手段来描述结构化类元及其内部结构的定义。这种内部结构是由部件及其相互连接构成的，它们都处于这个组合结构的命名空间之内。结构化类元可以嵌套，所以其中的部分可以是另一个结构化类元。除了表示组件之外，结构化类元也可以表示一个类。因此，组合结构图在设计时是很有用的，可以将类分解为组成部分，并对它们运行时刻的协作进行建模[36, 37]。

5.10.1 基本概念：组合结构的部分

图 5-50 展示了水培园艺系统的 WaterTank 的组合结构图。它的名称被放在顶部，具体的命名惯例应该由开发团队来确定。WaterTank 包含了 Heater 和 Tank 两部分，它们协作来实现 WaterTank 的功能，即提供适当加热的水供培植使用。

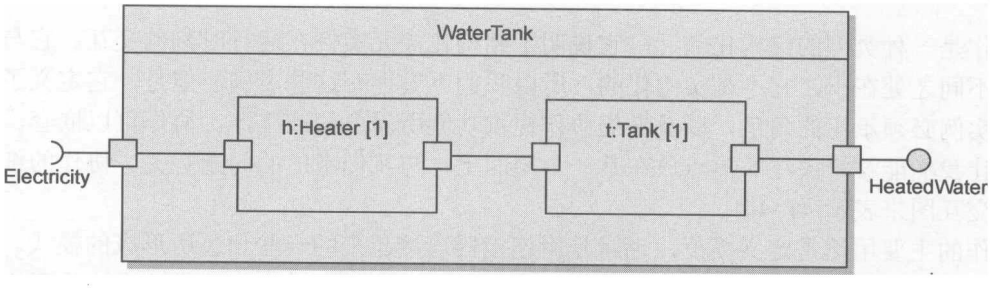


图 5-50 WaterTank 的组合结构图

组合结构中部分的名称采用的格式是“角色名称: 类名称[多重性]”，其中的角色名称定义了这个部分在这个组合结构中所扮演的角色。虽然显示多重性是可选的，但是我们在图 5-50 中明确了 WaterTank 包含一个 Heater 和一个 Tank。

5.10.2 基本概念：组合结构的部分与接口

组合结构及其部分通过端口与它们的外部环境实现接口，端口用某个部分或组合结构边

界上的一个小正方形来表示。在图 5-50 中我们看到，Heater 和 Tank 都有端口，它们通过这些端口进行交互，实现 WaterTank 的功能。另外，WaterTank 有一个端口，通过这个端口，它接收 Heater 所需的电力。它还有一个端口，从 Tank 向环境提供加热过的水。

利用端口进行所有的交互为结构化类元提供了封装。除非特别说明，这些端口具有公有可见性。隐藏端口的表示方法是将小正方形完全放在组合结构之内，只有一条边与边界接触。这些端口实现的功能可能是类似测试点这样的不公开提供的功能。端口的名称和多重性是可选的，但在需要时应该明确提供。端口名称的格式是“端口名称: 端口类型[多重性]”。在命名一个端口时，端口类型是可选的[38, 39]。

我们将接口与这些端口联接起来，这些接口定义了这个组合结构的交互细节。这些接口通常以小球和插座的表示法来表示。要求的接口表示为一个插座，表明需要从它的环境得到的服务。小球表示法则表明通过提供的接口所提供的服务。作为 WaterTank 的一部分，Heater 从水培园艺系统中获取电力，Tank 为培植提供加过热的水。

5.10.3 基本概念：组合结构连接器

组合结构图中的连接器提供了组合结构和它的环境之间的通信联系，也提供了内部部件之间的通信手段。在图 5-50 中，部件之间有三个连接器。连接组合结构边界的两个称为“委托连接器”，在 Heater 和 Tank 之间的称为“组装连接器”（也称为“接口连接器”）。这里，Heater 向 Tank 提供热量，以实现所需的服务。图 5-50 中使用了委托连接器的直线表示法来表示组装连接器，也可以使用小球和插座的表示法来表示这个连接。对连接器的命名是可选的，图 5-50 中的连接器没有被命名，因为这里的意思很清楚，不需要命名[40]。

5.10.4 高级概念：协作

协作是一种类型的结构化类元，它说明了结构化类元实例在运行时刻的交互。它与组合结构的不同之处在于，它不是实例化的，所以我们不会拥有这些实例。但是，它定义了结构化类元实例必须承担的角色，以及它们协作提供功能所需要的连接器。协作可以嵌套，抽象的概念让我们能够将关注的焦点放在某一个层面上，与我们考虑的问题有关。协作的细节可能通过交互图来表示[41, 44]。

协作的主要用途是定义模板，也就是用连接器连接起来的一些角色所形成的模式。在运行时刻，结构化类元实例会与这些角色绑定，这样它们就能通过合作提供这个协作所定义的功能。例如，图 5-51 展示的 TemperatureControl 协作定义了水培园艺系统中的一个控制温度的模式。在这个模式中，TemperatureController 利用了 TemperatureRamp，后者定义的支持一种作物所需的确切温度变化[93, 94]。

协作的名称显示在封装该协作的虚线椭圆内，可以用一条虚线（这里没有显示）与角色定义分开来。在这个协作中，我们定义了两个角色，即 TemperatureController 和 TemperatureRamp，它们由一个连接器相连。由于连接器没有被命名，它将由临时的运行时刻手段来实现，如一个属性或一个参数。如果它有名称，就会实现为一个实际关联的实例，即

一个链接[95, 96]。

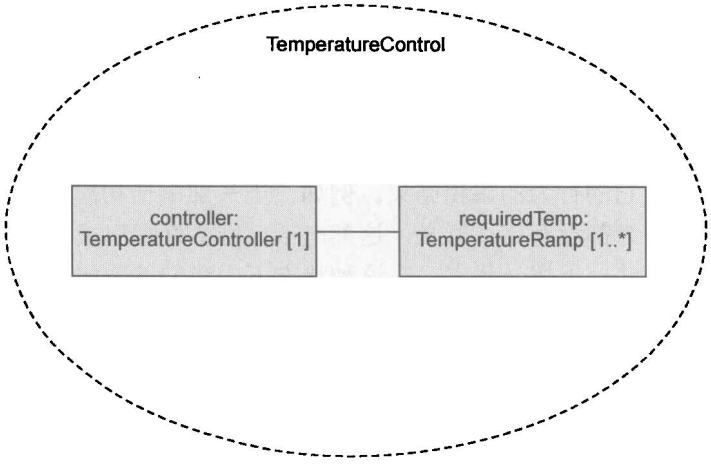


图 5-51 TemperatureControl 协作

角色带有名称和类型的标签，格式为“角色名称: 角色类型[多重性]”。角色名称描述了某个可以承担这一角色的类元实例，角色类型是对这个类元实例的约束。我们展示了 TemperatureController 和 TemperatureRamp 这两个角色的角色名称、角色类型和多重性[97, 98]。

一个角色定义了一些属性，这些属性是一个结构化类元参与这个协作所必须具备的。用接口来定义角色的类型意味着只要满足这个接口的实例就能够承担这个角色，而不论它的内部设计或实现如何。类元实例可以拥有超过某个角色所需要的功能，这样，它就能够在一个协作中同时承担多个角色，甚至同时在不同协作中承担多个角色[99, 100]。

5.11 状态机图

状态机在用到实时处理的行业中是众所周知的。状态机图被用来设计和理解时间关键的系统，在这种系统中，时间不正确的后果是严重的。医疗设备、财务交易系统、卫星命令和控制系统、武器系统等都是典型的例子，在这些系统中，状态机图可以在理解系统对关键事件的响应中起到重要的作用。

状态机图将行为表示为一系列的状态转换，由事件触发，并与可能发生的动作相关联。这样的状态机也被称为行为状态机。状态机图通常用于描述单个对象的行为。但是，它们也可以用于描述系统中更大元素的行为。(第 1 章中曾提到，在复杂系统中选择抽象的层次与观察者的意图有关。) 状态机图和活动图有亲戚关系。但是，状态图关注的是状态以及状态之间的转换，而不是活动的流程。

并非每个类都有重要的、事件次序相关的行为，所以我们只为那些表现出这种行为的类

提供状态机图。也可以利用状态机图来展示整个系统的事件次序相关的行为。在分析过程中，可以利用状态机图来说明系统的动态行为。在设计过程中，可以利用状态机图来记录单个类或几个协作类的动态行为。

状态机图的两个基本元素是状态和状态转换。

5.11.1 基本概念：初始状态、最终状态和简单状态

对象的状态代表了它的行为的累积结果。例如，当一部电话初次装好时，它处于空闲状态，这意味着以前的行为都没有什么意义，这部电话已准备好呼出或接收呼入。当我们拿起话筒时，话筒脱离底座并处于拨号状态。在这种状态下，我们不希望电话铃会响，我们希望能够与另一部电话旁的某个人建立通话。当话筒在底座上时，如果电话铃响了，我们拿起话筒，电话就处于接听状态，我们希望与呼入的人进行通话。

在任何给定的时间点，对象的状态包含了它的所有特性（通常是静态的），以及这些属性当前的值（通常是动态的）。所谓特性，指的是这个对象的所有属性和它与其他对象的关系。可以将单个对象状态的概念进行泛化，应用于这个对象的类，因为同一个类的所有实例都处于相同的状态空间，它包含了不确定的、有限可能性的一组状态。

当一个对象处于指定状态时，它可以做下面的事情：

- 执行一项活动；
- 等待一个事件；
- 完成一个条件；
- 做上面的几件事情或全部事情。

在每个状态机图中，必须只有一个默认的初始状态。我们从一个特殊符号画出一个无标签的转换指向这个初始状态，这个特殊符号显示为一个实心的圆。有时候，也需要设计一个结束状态。通常，与一个类或整个系统有关的状态机永远也不会到达结束状态，只是在包含这个状态机的对象被销毁时，它就不存在了。结束状态的表示方式是从它画出一个无标签的转换，指向一个特殊的符号，这个特殊符号显示为一个空心圆套着一个实心圆。初始状态和结束状态在技术上被称为“伪状态”。图 5-52 展示了水培园艺系统中一个时间间隔计时器的元素。对于简单的状态，状态的名称显示在代表状态的圆角矩形中。计时器的两个简单状态是 Initializing 和 Timming。简单状态没有子状态（将在下一小节中讨论子状态）。



图 5-52 简单状态的表示法

5.11.2 基本概念：转换与事件

状态之间的移动称为“转换”。在状态机图中，转换表示为状态之间的有向箭头。每个

状态转换连接两个状态。图 5-33 展示了从初始状态到 Initializing 状态、从 Initializing 状态到 Timing 状态以及从 Timing 到最终状态的转换。在状态之间移动被称为“触发一次转换”。状态可以有指向自己的状态转换，许多不同的状态转换指向同一个状态也是很常见的，但是这样的转换必须是唯一的，即任何时候都不会从一个状态触发多个状态转换。

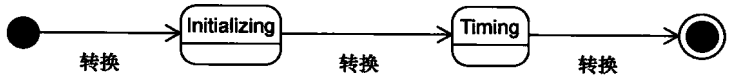


图 5-53 间隔计时器的转换

有多种不同的方式可以控制触发一次转换。没有标注的转换称为“完成转换”。这就意味着，当源状态完成时，这个转换就会被自动触发，然后进入目标状态。可以在图 5-53 中看到，Initializing 状态到 Timing 状态之间的转换就属于这种情况。

在其他情况下，必须发生某些事件才能触发转换，这样的事件被标注在转换上。事件是可能导致系统状态改变的某些发生的事情。例如，在水培园艺系统中，下列的事件对系统的行为将产生影响：

- 种植了一种新作物；
- 一种作物已经可以收割；
- 狂风暴雨天气导致温室温度下降；
- 一个制冷设备失效了；
- 时间推移。

前面 4 个事件可能触发某种动作，如开始或终止执行某一份种植计划，打开一个加热器，或者以声音的方式通知种植管理者。时间推移是另一个问题：虽然几秒钟或几分钟的流逝可能不会对我们的系统产生重大影响（已知作物的生长通常比这个时间要长很多），但过去几个小时或几天可能对系统是个信号——需要开灯或关灯，或者改变温室中的温度，以便人为地创造作物生长需要的一天。在图 5-54 中，我们画出了时间间隔计时器的状态图。在这里可以看到，计时器的执行可以通过 pause 事件进入 Paused 状态，然后通过 resume 事件继续计时。

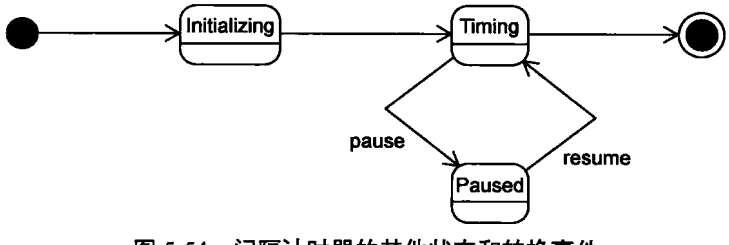


图 5-54 间隔计时器的其他状态和转换事件

虽然转换通常表示一个对象在状态之间变换，但它们也可以是递归的，表示从一个状态退出再重新进入同一个状态。

到目前为止介绍的 UML 元素包括了所有状态转换图的基本元素。它们共同提供了一种

表示法，能够描述简单的、平面的有限状态自动机，适合具有有限状态的应用。有些系统具有大量的状态，或者展现出特别复杂的事件次序有关的行为，涉及条件转换，或者转换依赖于前面进入过的状态。这样的系统就需要状态转换图的高级概念。下面将讨论这些概念。

5.11.3 高级概念：状态活动——入口活动、执行活动和出口活动

活动可能与状态关联起来。具体来说，可以指定在状态相关的某个时间点上执行某种活动：

- 在进入一个状态时执行一个活动；
- 在处于一个状态时执行一个活动；
- 在离开一个状态时执行一个活动。

图 5-55 展示了这个概念的一个例子。可以看到，在进入 Timing 状态时，启动了计时器（由一个箭头指向两条平行线的图标表示）；在退出这个状态时（由在两条平行线之间的一个箭头的图标表示），停止了计时器（请注意，这些图标是与工具有关的）。处于这个状态中时，测量时间的推移（由环形的箭头表示）。

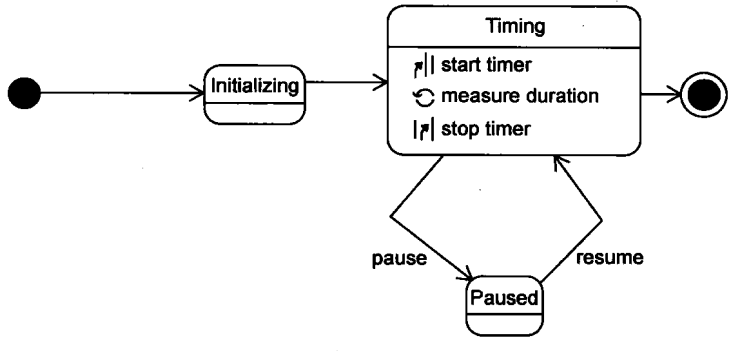


图 5-55 入口活动、执行活动和出口活动

5.11.4 高级概念：控制转换

前面曾提到，可以有一些方式对状态转换实现更好的控制，不仅仅是事件导致转换。可以通过指定条件来控制转换。这些条件起到了监护的作用，当事件发生时，条件将决定允许转换发生（如果条件为真）或不允许转换发生（如果条件为假）。

控制转换的另一种方法是利用效果。效果是一种行为（即活动、动作），它在指定事件发生时发生。因此，当转换事件发生时，转换会被触发，效果也同时发生。这些改进可以相互组合使用。

让我们对时间间隔计时器的例子进行扩展，说明效果的使用。水培园艺系统中发生的一个重要事件就制冷器失效。我们不能寄希望于这种事不会发生，我们需要将基本计时器变成一个时间间隔计时器，来测量制冷器运行的总时间。这样做的目的是在制冷器运行了一定的

时间之后，通知我们对它进行维护。我们希望定期的维护能防止制冷器失效。所以我们改进了状态机图，如图 5-56 所示。

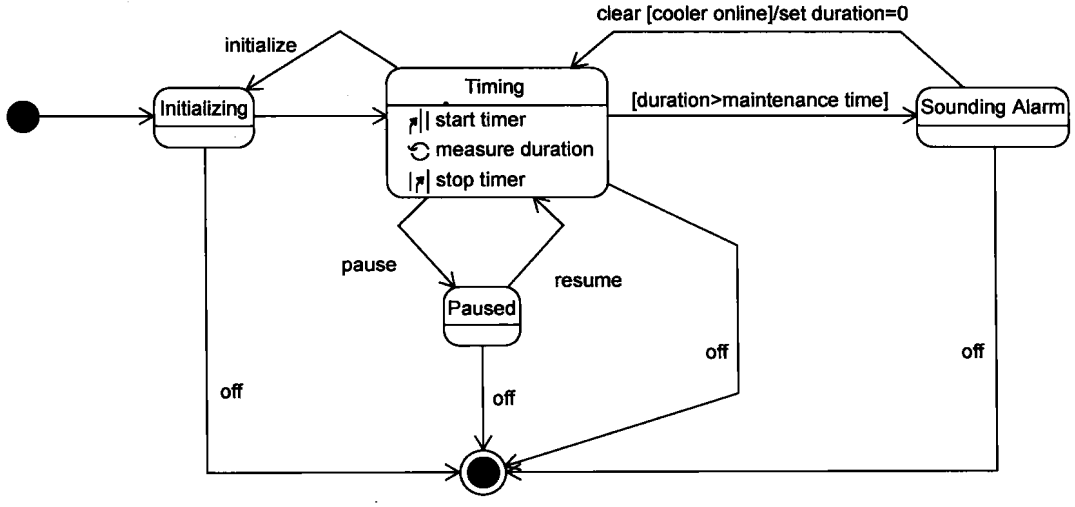


图 5-56 改进的间隔计时器状态机图

在这个图中，可以看到超时转换已经被一个条件所取代。这个条件规定当时间间隔超过维护时间时，计时器将转换到 Sounding Alarm 状态，警告需要维护。从 Sound Alarm 到 Timing 的转换展示了组合使用事件、条件和效果的情况。有一个 clear 事件，但它有一个警戒条件，即制冷器必须回到工作状态（然后计时才会继续）。这个条件被放在方括号中。如果 cooler 没有回到工作状态，这次转换就不会被触发。如果 cooler 回到了工作状态，效果会发生（持续时间被重置为 0，在斜杠后显示），转换也会被触发，计时重新开始。

在有条件的状态转换中，值计算的次序是很重要的。如果在事件 E 发生时，状态 S 在条件 C 下会发生转换 T，并产生效果 A，那么将以下列的次序进行：

- 事件 E 发生；
- 计算条件 C 的值；
- 如果 C 的值为真，那么 T 被触发，效果 A 被调用。

这意味着如果条件值计算的结果为假，状态转换就不会发生，直到下次事件再将发生时，再重新计算条件的值。计算值时的副作用或者执行一个退出活动时的副作用将不会影响到状态的转换。例如，假定事件 E 发生，条件 C 的计算结果为真，但是在执行退出活动时改变了一些东西，使 C 的值不再为真。这种情况下，状态转换仍会被触发。

5.11.5 高级概念：复合状态与嵌套状态

到这里为止，我们已经讨论了简单的状态和它们之间的转换。在更大、更复杂的系统中，状态机图会变得很大，纠缠在一起，变得不实用。嵌套状态的功能让状态图有了深度，这是状态图的一个主要功能，可以缓解复杂系统中状态和状态转换产生组合爆炸的情况。

在图 5-75 中，我们看到内嵌的状态 Timing、Sounding Alarm 和 Paused。这种嵌套用一个环绕的边界来表示，称为“区域”。这个封闭的边界被称为一个“复合状态”，所以我们现在有了一个复杂状态，名为 Operating，它包含内嵌的状态 Timing、Sounding Alarm 和 Paused。还要注意，这个图在视觉上进行了简化，只有一个转换从复合状态 Operating 指向最终状态。这意味着任何内嵌状态的 off 事件发生时，都会触发向最终状态的转换。

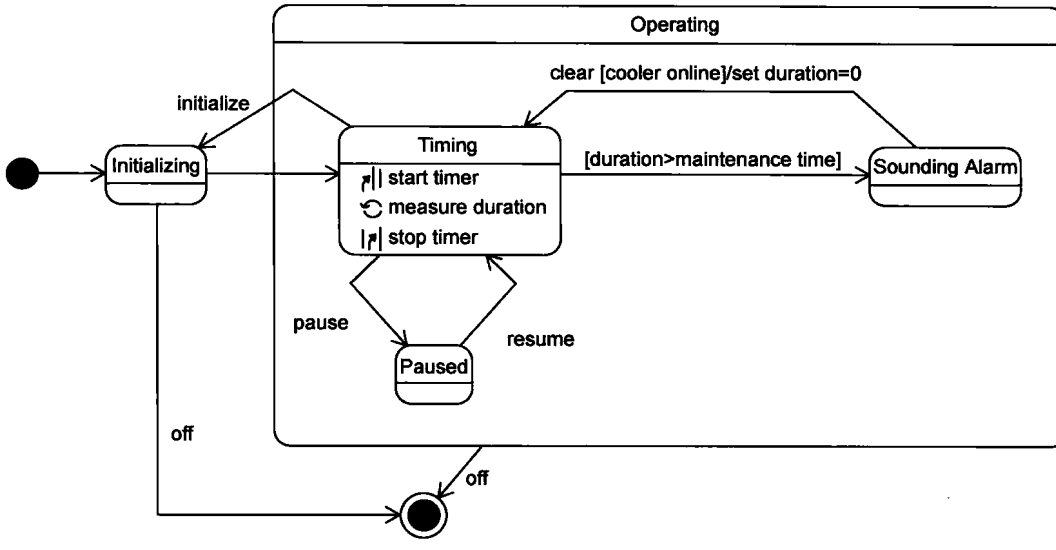


图 5-57 复合状态和嵌套状态

嵌套的深度可以是任意的，所以子状态也可以是复合状态，包含更低层的子状态。对于复合状态 Operating 和它的三个子状态来说，嵌套的语义意味着一种 XOR（异或）关系。如果系统处于 Operating 状态（复合状态），那么它必须处于三个子状态之一：Timing、Sounding Alarm 或 Paused。

在画嵌套的状态转换图时，为了简单起见，可以聚焦或不聚焦于某个状态。不聚焦时略去子状态的细节，聚焦时会显示子状态的细节。不聚焦时，图 5-7 会变成一个非常容易理解的状态图（参见图 5-58）。

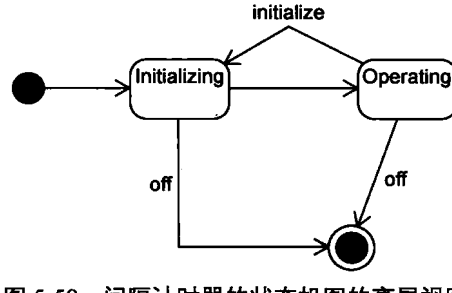


图 5-58 间隔计时器的状态机图的高层视图

5.11.6 高级概念：并发与控制

并发行为可以用状态机来描述，只要将复合状态用虚线分成两个或多个子区域就可以了。复合状态中的每个子区域表示并发发生的行为。图 5-59 展示了一个具有三个并发子区域的状态。

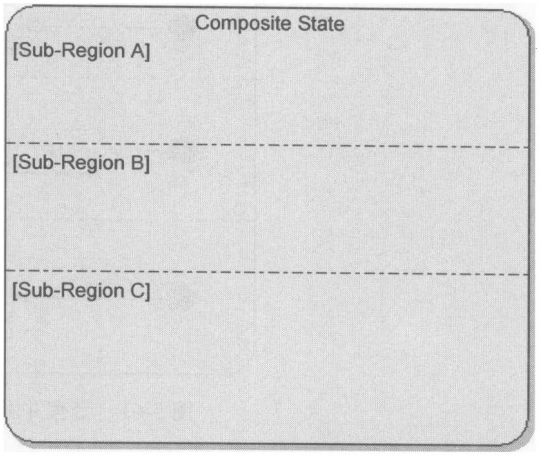


图 5-59 状态机图中的并发子区域

对于我们的例子来说，当制冷器到了需要维护的时间时，间隔计时器会发出声音警报。前面的图 5-56 中展示了这一点，即让状态机转换到 Sounding Alarm 状态。假设除了发出声音警报之，我们还希望记录下制冷器处于需要维护的状态多久了。图 5-60 中展示了这一点，用一个复合状态 Maintenance Overdue 来取代 Sounding Alarm 状态，它包含两个并发状态：Sounding Alarm 和 TimingMaintenance Overdue（它记录制冷器处于需要维护的状态的时间）。因此，当系统转换到 Maintenance Overdue 这个复合状态时，两个子状态都会开始并发地执行，即维护状态计时器开始计时，同时发出声音报警。（请注意，我们去掉了允许关闭报警而没有让制冷器重新工作的转换，参见图 5-56 中从 Sounding Alarm 状态发出的 off 转换。）这可以确保不会在没有进行维护的情况下关闭报警。

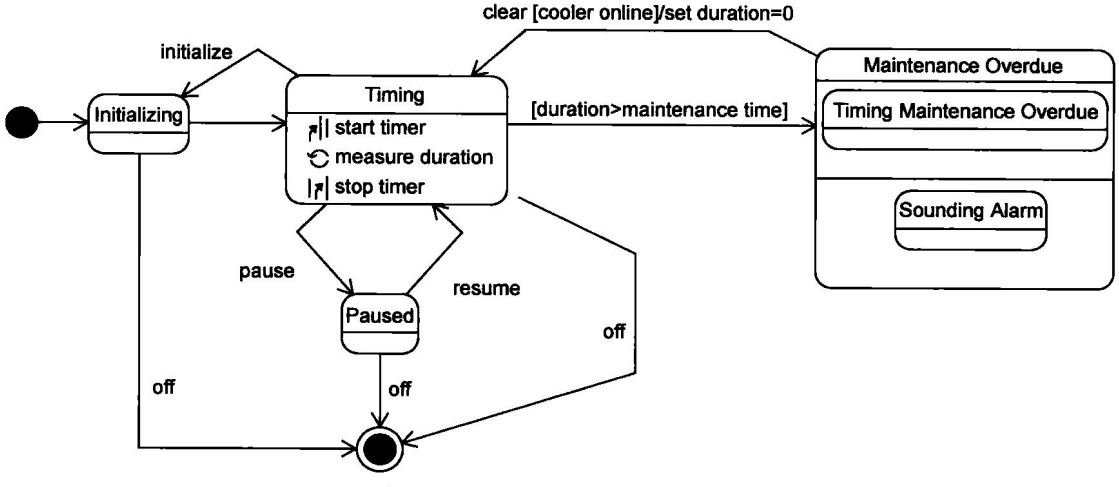


图 5-60 带有并发子区域的复合状态 Maintenance Overdue

利用 UML，可以用不同的方式来表示进出并发状态的控制流，每一种方式都有自己准确的含义。必须小心确保画出的控制流确实代表了你的意图。让我们通过一个一般的复合状态

来看看这一点，如图 5-61 所示。

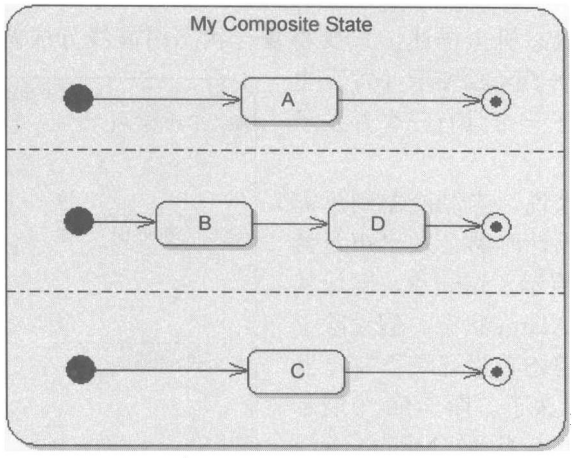


图 5-61 带有并发子区域的一般复合状态

可以通过不同的方式转换到一个复合状态。例如，可以有一个到复合状态的转换，如图 5-62 所示。在这种情况下，并发状态机将全部被激活，并发地开始工作。每个独立的子区域都会从这个区域中默认的初始状态开始（伪状态）。虽然不是必需，但我们建议明确地在子区域中显示初始状态，这样会更清晰。

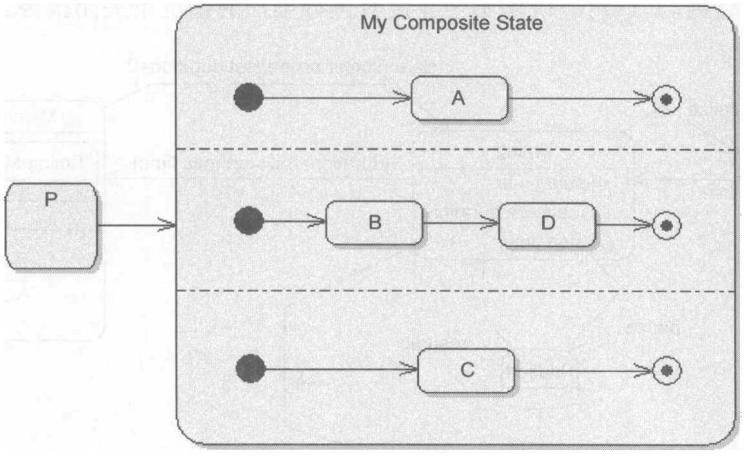


图 5-62 转换至一个复合状态

另一种进入复合状态的方式就是利用分叉。分叉节点实际上是另一种伪状态（像初始状态和最终状态一样），它将进入的转换分成两个或更多离开的转换。图 5-63 展示了这种方式下分叉的用法。这个单一的输入转换可能有一个事件或一个警戒条件，多个离开的转换则可能没有。在我们的例子中，分叉将控制流分开，复合状态中的并发执行将从子状态 A、B 和

C 开始，它们是从分叉离开的多个转换的目标子状态。

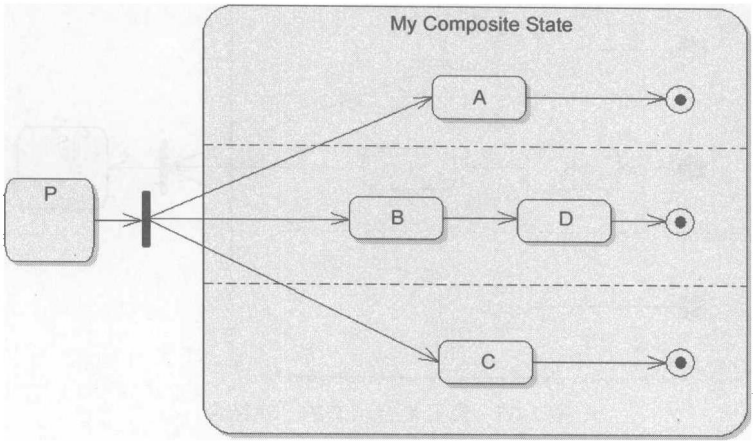


图 5-63 分叉至一个复合状态

在离开复合的并发状态时，可以采用类似的结构。图 5-64 展示了从复合状态到后续状态的转换。当前面的子状态（A、D 和 C）完成时，从复合状态发出的单一的转换就会被触发。

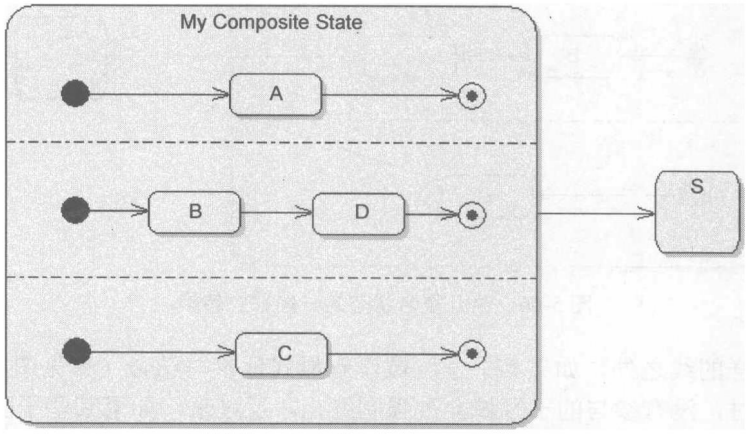


图 5-64 离开复合状态的一次完成转换

也可以利用一个结合节点（另一个伪状态）来执行类似的控制流合并。图 5-65 展示了来自一些并发子区域的转换指向一个竖线段，在这里它们合并成一个离开的转换。这个离开转换也可以有事件或警戒条件。在这种情况下，当所有的结合子状态（A、D 和 C）被激活，事件发生，而且条件满足时，到状态 S 的转换就会发生。

图 5-65 中显示的情况与图 5-66 中显示的情况是不等价的。在图 5-66 中，没有像使用结合节点时那样的控制流合并。在这种情况下，如果来自子状态 A、D 或 C 的转换中的任何一个被触发时，所有其他并发的子状态都会终止。

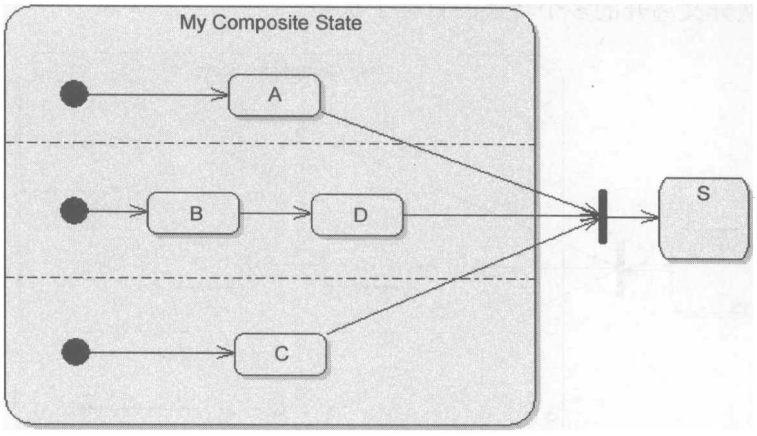


图 5-65 离开复合状态的一次结合

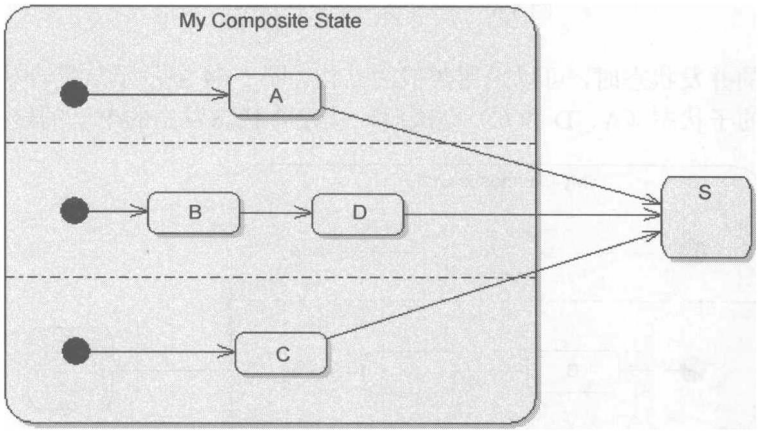


图 5-66 离开复合状态的一些独立转换

除了这些同样的线之外，如果某些子区域没有显式地参与结合（参见图 5-67），那么当结合转换被触发时，没有参与的子区域会被强制终止。反过来，如果某些子区域没有显式地参与分叉，那么没有参与的子区域会从它们初始的状态开始执行[18, 19]。

5.11.7 高级概念：子状态机状态

除了简单状态和复合状态之外，还有第三种主要的状态类型——子状态机状态。子状态机状态被用于简化状态机图，或者用于包含可以在不同图中复用的常见行为。例如，回过头去看图 5-56。让我们假定在维护时间超过时需要做更多的事情，而不仅仅是发出声音报警。假定需要另一个计时器来记录制冷器超过维护期的时间，并且系统需要记录温度、冷却剂的压力、开/关的周期、湿度，用图形的方式显示这些信息随时间的变化情况。这些新的记录需求可能导致一个相当复杂的状态机图。

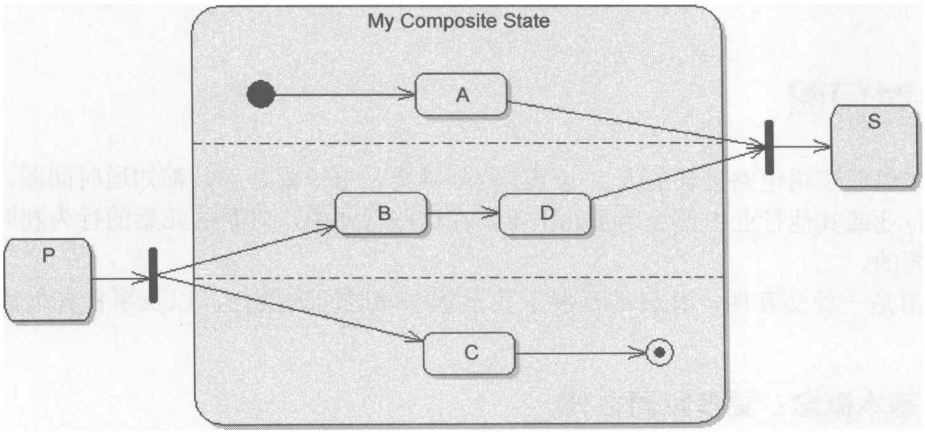


图 5-67 部分参与结合和分叉

为了让图形保持简单，可以用一个名为 Operating:Recording 的子状态机来代替 Sounding Alarm 子状态（参见图 5-68）。这个子状态机状态代表了“一个完全独立的状态机图”，它可以表示前面提到的所有详细的记录需求。通过这种方式，子状态机让我们能够将复杂的状态机图组织成为可以理解的结构。

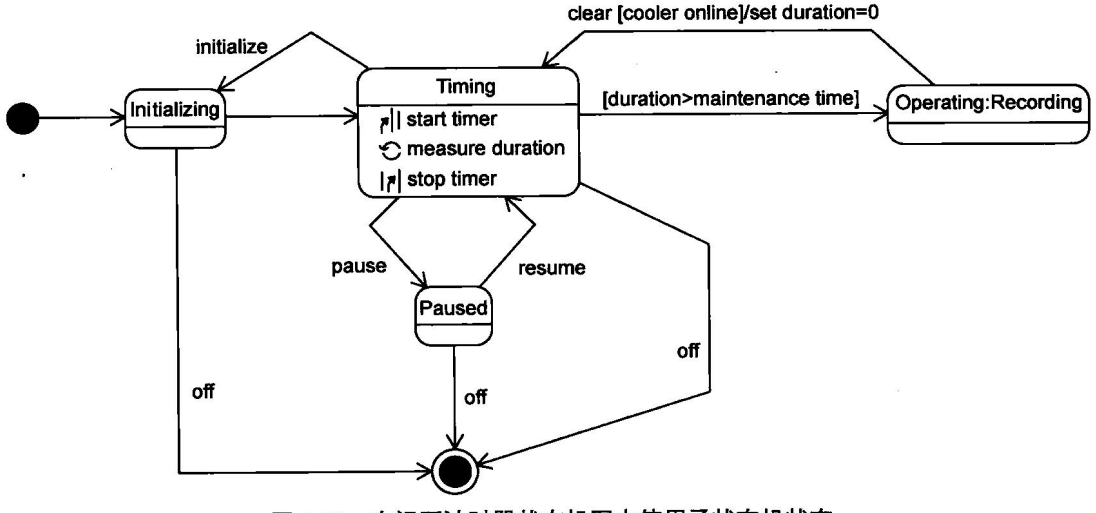


图 5-68 在间隔计时器状态机图中使用子状态机状态

5.11.8 高级概念：其他状态机图元素

在 UML 中，状态机图可能是语义最丰富的一种图，它们可以非常、非常复杂。除了之前讨论过的元素之外，还有许多元素可以用在状态机图中（如进入点和退出点、浅历史和深历史、协议状态机等）。如果读者需要或希望了解 UML 的点点滴滴，附录 B 中提供了一些有用的参考。

5.12 时间图

任何接触过逻辑电路设计的人、电子工程师甚至是电子爱好者，都知道时间图。类似的图在这一行业或其他行业中已经用了几十年，在这些行业中，对系统元素的行为和时间的理解是很重要的。

时间图是一种交互图，其目的是展示元素状态随时间的变化，以及事件如何改变这些状态。

5.12.1 基本概念：更多相同之处

时间图中的许多元素也出现在其他 UML 图中。它们可以有一条或多条生命线、一个或多个对象（或其他 UML 类元）、两个或多个状态、消息，等等。如果需要，请考虑前面的讨论，复习一下这些元素的语义。

5.12.2 基本概念：布局

时间图利用了 UML 的元素，以一种不同的组织方式呈现给用户。时间图的一般布局是从时间上追索与它并列的一个序列图（参见图 5-69）。

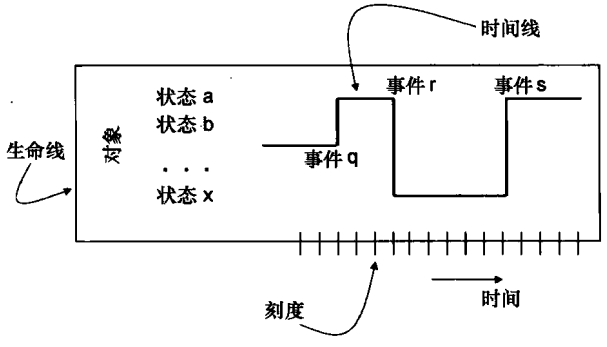


图 5-69 一般时间图

对于图中的每个对象，时间图有一条或多条生命线（看起来像水平的分区）。生命线的名称（即对象名称）显示在生命线内，对象可能的状态被列在生命线内。同时，生命线也显示了对象如何从一个状态转变到另一个状态。导致状态转变的事件显示在时间线旁。水平轴显示了时间，可能还有刻度，帮助读者更好地理解具体的时间。例如，图 5-70 展示了 Valve 对象的一个简单的时间图，在水培园艺系统中，这个对象受控于注满 WaterStorageTank 对象。

可以看到，Valve 有两个简单状态：open 和 closed。单独来看，这张时间图不能提供多少系统运行的知识。（实际上，可以认为这张时间图是不完整的。）阀门什么时候打开？什么时候关闭？什么事件导致这些状态改变？

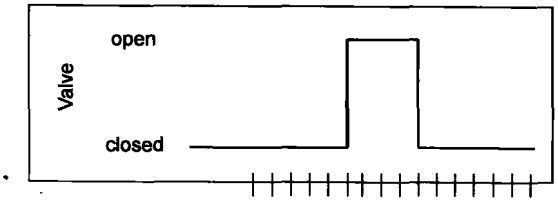


图 5-70 Valve 对象的时间图

5.12.3 基本概念：事件

导致状态改变的事件（或其他刺激物，如消息）显示在生命线内靠近对象时间线的地方。在图 5-71 中，我们加入了两个事件，即 TankLow 和 TankFull，它们导致了阀门的状态改变。

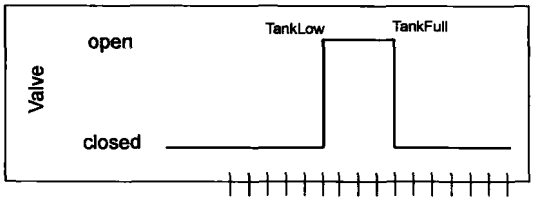


图 5-71 包含事件的 Valve 对象的时间图

5.12.4 基本概念：约束

在时间图中，约束可以用于说明约束或限制状态改变的条件。图 5-72 中展示了包含 Valve 和 Heater 两个对象的时间图，这张图展示了 Heater 的状态和 Valve 的状态之间的关系。

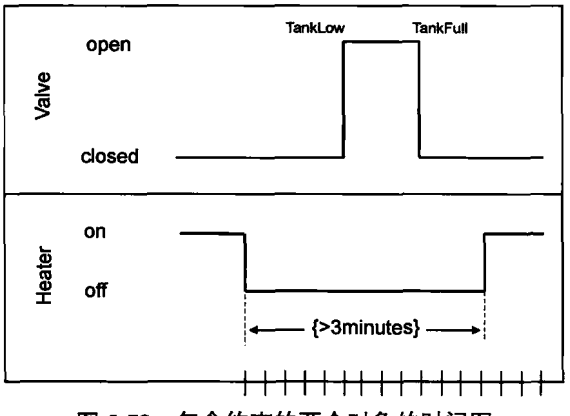


图 5-72 包含约束的两个对象的时间图

在这里可以看到，Heater 上有一个约束，限制了加热器多快可以重新打开。（这种类型的约束可能是为了防止快速或重复切换加热器的加热元件的开关状态，因为这会影响使用寿命。）该约束表明当加热器关闭之后，必须过 3 分钟才能够重新打开。

5.12.5 高级概念：另一种表示形式

当时间图有许多生命线，或者对象有许多状态时，除了像前面的图中那样使用生命线之外，还可以利用更紧凑的表示形式，如图 5-73 所示。状态显示在六边形内，状态改变发生在六边形之间。

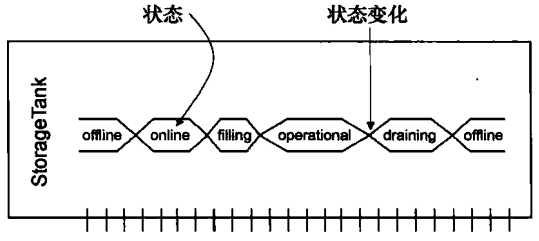


图 5-73 在时间图中使用紧凑表现形式代替时间线

在这里，状态的改变没有表示为时间线的抬升或下降，而是显示为沿着水平轴的变化。

5.12.6 高级概念：事件与消息

前面曾提到，不只是事件可以导致状态改变，像消息等其他刺激物也可以导致状态改变。那么，什么时候该用哪一种呢？这里的微妙区别在于，事件有具体发生的时间和空间，消息则不然。

例如，前面图 5-71 中显示的两个事件 TankLow 和 TankFull，是真实地发生在存储水箱中的。除了利用这些事件之外，还可以利用消息来打开或关闭阀门。假定种植管理员决定向储水箱加一些水，虽然水箱的当前容量并不低。在这种情况下，使用消息就比使用事件更好。图 5-74 展示了两条消息（openCmd 和 closeCmd）：命令阀门打开，然后加注水箱；然后关闭阀门，停止加注。

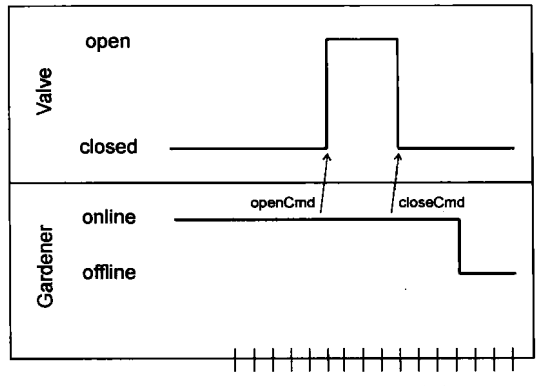


图 5-74 在时间图中使用消息代替事件

消息和事件都可以用在 UML 图中，目的是清楚地表达设计者的意图。但是在有疑问时，

最好的做法是检查有疑问的元素（如消息与事件）的语义，然后正确地使用它们。UML 图不只是草图，每个元素都有具体的意义和正确的用法。

5.13 对象图

对象图被用于说明系统的逻辑设计中存在的对象以及对象之间的关系。换言之，对象图代表了时间上的一张快照，记录了一组特定配置的对象上的瞬时事件流。因此，对象图是原型化的——每张对象图都代表了结构上的关系，这些关系可能发生在一组给定的类实例上。从这个意义上说，单张对象图代表了系统对象结构的一张视图。在分析过程中，对象图常被用于说明主要场景和次要场景的语义，提供对系统行为的跟踪。在设计过程中，对象图常被用于说明逻辑系统设计中各种机制的语义。不论在哪个开发阶段，对象图都能提供一些具体的例子，帮助实现关联类图的可视化。

对象图的两个基本元素是对象和它们的关系。

5.13.1 基本概念：对象

图 5-75 展示了在对象图中表示对象的图标。与类图相似，水平线将图标内的文字分成两个区域，一个区域代表对象的名称，另一个区域提供了可选的对象属性及值的视图。但在这里我们看到，工具的实现没有使用水平线将它完全分隔成两个区域。

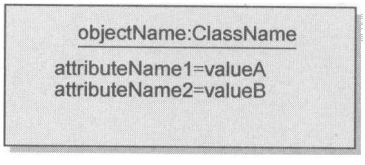


图 5-75 一般对象图标

对象的名称可以采用下面三种格式之一。

- objectName 只有对象名；
- :ClassName 只有类名；
- objectName :ClassName 对象名和类名。

所有的对象名称都加了下划线，目的是清楚地区分对象名称和类名称。如果没有指定一个对象的类，既没有用上面的语法显式地指定，也没有在对象的说明中隐式地指定，那么这个对象的类就被认为是匿名的。如果只指定了类名称，那么这个没有对象名的图标代表的就是一个明显的匿名对象。

对于某些对象来说，显示部分或全部属性可能会有用。说“某些”，是因为这些对象代表的只是对象结构的一个视图。这些属性必须引用这个对象的类或超类中定义的属性。语法中包含了指定每个属性的值的能力，如图 5-75 所示。我们没有展示类的属性，如操作，因为它们是这个类的所有实例共有的。

5.13.2 基本概念：对象关系

第 3 章中曾解释过，对象通过它们与其他对象的链接来进行交互，如图 5-76 所示。这张图是与图 5-39 的类图对应的对象图。链接是关联的一个实例，就像对象是类的实例一样。

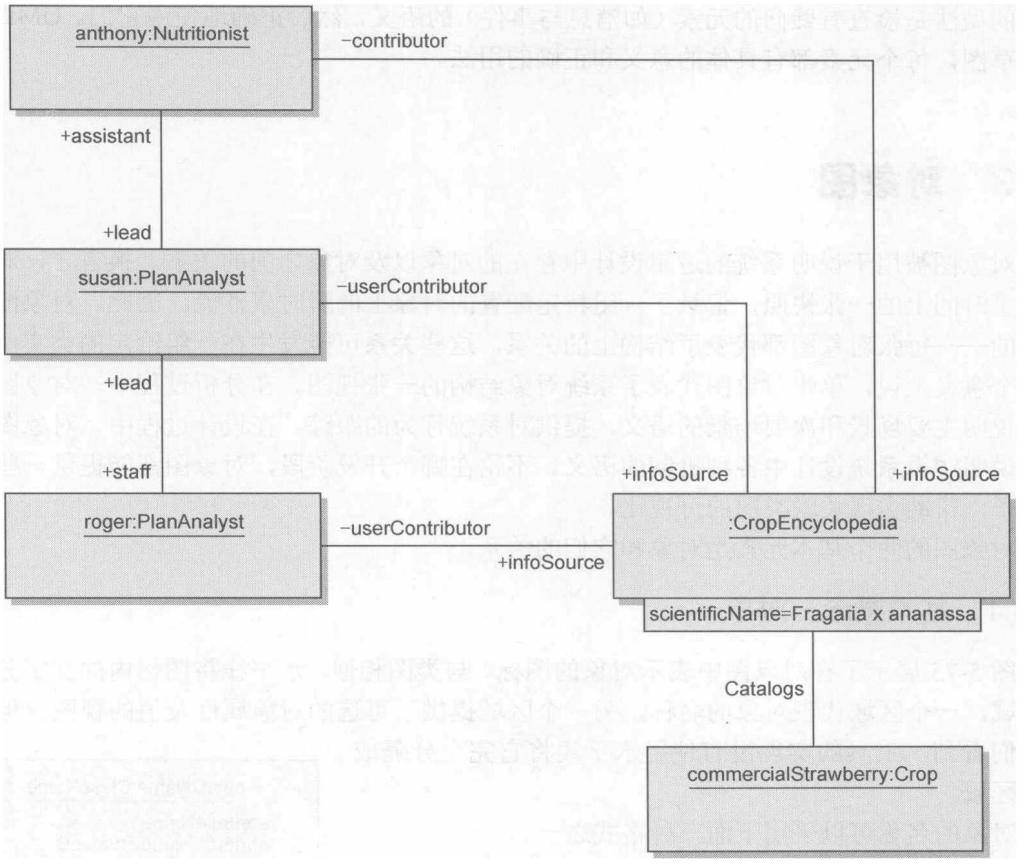


图 5-76 对象关系

当且仅当两个对象对应的类之间存在关联时，对象之间才可能存在链接。这种类关联可以通过任何方式实现，这意味着类的关系可以是简单的关联、泛化、聚合或组合。两个类中存在这种关联则表明，两个类的实例之间存在着一条通信路径（即链接），一个对象可以通过它向另一个对象发送消息。所有的类都隐式地有到自己的关联，因此对象可以向自己发送消息。

5.13.3 高级概念：端点名称和限定符

我们已经讨论了对象及其关系，它们构成了对象图中表示法的基本部分。但是，有一些特别棘手的开发问题需要比基本表示法更多的东西。我们曾在讨论类图的时候警告过，这里必须再次强调，只有在记录期望的场景语义必需时，才应使用这些高级功能。

在前面的小节中，我们指出类图中的关联可以用一个角色来修饰，说明一个类关联另一个类的目的或容量。对于某些对象图，重申两个对象间对应链接的这种角色也是有用的。通常，这种修饰有助于解释为什么一个对象操作另一个对象。图 5-76 提供了这种高级功能的一个例

子。这里我们看到，PlanAnalyst 对象（Susan）既用到了来自匿名 CropEncyclopedia 对象的信息，也向它贡献了信息。这样做时其扮演的角色是 userContributor，链接的端点名称说明了这一点。比较图 5-76 和图 5-39 时（在类图的讨论中），我们注意到有两个 PlanAnalyst 类的实例，一个是 Susan，角色是 lead，她与另一个实例 Roger 协作，Roger 的角色是 staff。Susan 在与 Anthony（一个 Nutritionist 对象）的关系中也扮演了 lead 的角色，Anthony 是协作 Susan 的。

在图 5-39 中，我们看到一种名为“限定符”的类修饰，它的值在关联的目标端唯一地确定了一个对象。具体来说，CropEncyclopedia 类利用 scientificName 作为限定符，来导航到 CropEncyclopedia 实例管理的一组条目中的具体一项。图 5-76 中的作物实例是 commercialStrawberry，它选择利用 Fragaria × ananassa[1]作为 scientificName 限定符。

利用和类图同样的表示方法，在对象图中使用的其他表示法还包括约束、关键词标签、导航和链接名称。

5.14 通信图

如果读者熟悉 UML 以前的版本，就会知道通信图在 UML 2.0 以前的版本中的名称——协作图。通信图是一种类型的交互图，它关注对象在参与具体的交互时，对象之间如何链接以及传递什么消息。

5.14.1 基本概念：对象、链接和消息

当且仅当两个对象对应的类之存在关联时，对象之间才可能存在链接。两个类中存在这种关联则表明，两个类的实例之间存在着一条通信路径（即链接），一个对象可以通过它向另一个对象发送消息。

如果对象 A 有一个链接到对象 B，A 就可以调用 B 中的类开放给 A 调用的所有操作，反之亦然。我们将调用操作的对象称为“客户”，将提供操作的对象称为“提供者”。一般来说，消息的发出者知道接收者，但接收者不一定知道发出者。

在稳定的状态下，系统的类结构和对象结构必须一致。如果我们展示了通过链接 L 调用了对象 B 上的操作 M，那么 B 的规格说明（或某个超类的规格说明）中必须包含操作 M。

图 5-77 展示了水培园艺系统的一张通信图的例子。这张图的意图是展示执行常见系统功能时的交互过程，即确定某种作物的预估净收割成本。

如图 5-77 所示，可以用一条或多条消息来修饰一个链接。通过一条指向目标对象的有向线段，可以说明消息的方向。操作调用是最常见的一种消息类型（另一种类型是信号）。操作调用可以包含实际的参数，这些参数与操作的签名相符，如图 5-77 中出现的 timeToHarvest 消息。

---

[1] Fragaria × ananassa 有一些常用的名称，包括 commercial strawberry、garden strawberry、cultivated strawberry 及 strawberry。

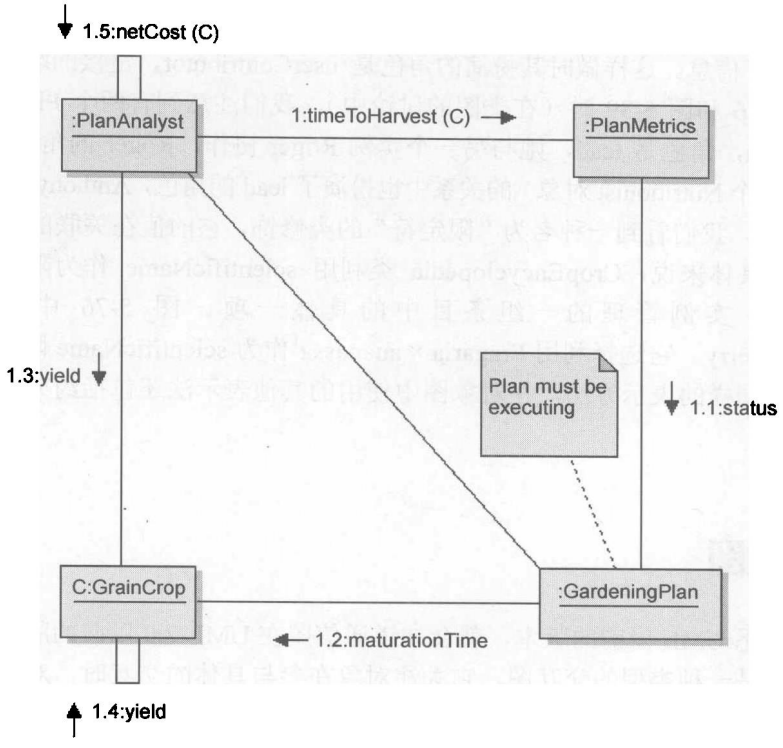


图 5-77 水培园艺系统的通信图

5.14.2 基本概念：顺序表达式

执行预估净收割成本这一系统功能需要几个不同对象的协作。为了显示明确的事件次序，我们在消息前面加上一个序号（从 1 开始）。这个顺序表达式说明了消息的相对次序，序号较小的消息在序号较大的消息之前被发送。图 5-77 中的序号就说明了这种消息次序。

利用嵌套的小数点数字的形式（如 4.1.5.2），可以展示某些消息是嵌套在下一次更高层的过程调用之内的。每个整数部分说明了在交互过程中嵌套的层次。同一层的整数部分说明消息的次序处于同一层次。在图 5-77 中，消息 1.3 跟在消息 1.2 后面，消息 1.2 跟在消息 1.1 后面，所有消息都是 timeToHarvest 调用（即消息 1）中嵌套的调用。

从图 5-77 中我们看到，这个场景的动作从某个 PlanAnalyst 对象调用 PlanMetrics 上的 timeToHarvest()操作开始。请注意，对象 C 是作为真实参数被传递给这个操作的。接下来，PlanMetrics 调用某个未命名的 GardeningPlan 对象上的 status()操作，图中包含了一个开发注解，说明必须检查指定的计划实际上在执行。GardeningPlan 对象接下来又调用了选定的 GrainCrop 对象上的 maturationTime()操作，要求得到作物预期成熟的时间。在这个选择符操作完成之后，控制回到 PlanAnalyst 对象，然后它调用 yield()操作，接着又导致了 C:GrainCrop 对象上的 yield()操作。控制再次回到 PlanAnalyst 对象上，它调用自己的 netCost()操作结束了

这个场景。(这个图也说明 PlanAnalyst 对象和 GardeningPlan 对象之间存在一个链接。虽然没有消息传递，但这个链接的存在突出了这两个对象之间的语义依赖关系。)

图 5-78 展示了和图 5-77 中一样的一组消息序列，但是消息的嵌套不同。这里，消息 1.1 和 1.2 嵌套在 timeToHarvest 消息（1）中，消息 2.1 嵌套在 yield 消息（2）中。其所提供的功能是相同的，但控制的结构不同。

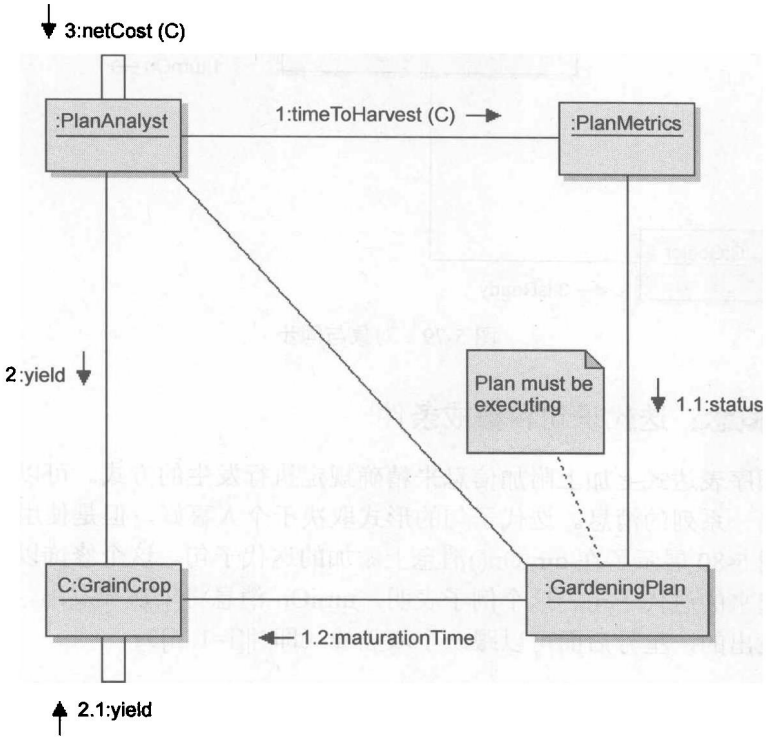


图 5-78 图 5-77 的另一种画法，显示了不同的次序编号

顺序表达式可以包含一个名称，说明当前的消息处于某个嵌套层次。例如，可以使用名称 a 和 b，消息 7.2a 和 7.2b 会在消息 7.2 被激活时并发发生。每个消息都有自己的控制流。

5.14.3 高级概念：消息与同步

尽管有点做作，图 5-79 中的例子展示了通信图中可能出现的不同类型的消息同步。消息 startup()是简单调用的例子，用带有实心箭头的有向线段表示，这说明它是一个同步消息。对于 startup()和 isReady()消息的情况，客户必须等待提供者完成消息的处理，然后控制流才能继续。

对于消息 turnOn()的情况，语义是不一样的。这是异步消息的一个例子，用开放的箭头表示。此时客户发出消息让提供者处理，提供者将消息加入待处理队列，然后客户就可以继续其他处理，而不必等待提供者。异步消息传递与中断处理是类似的。

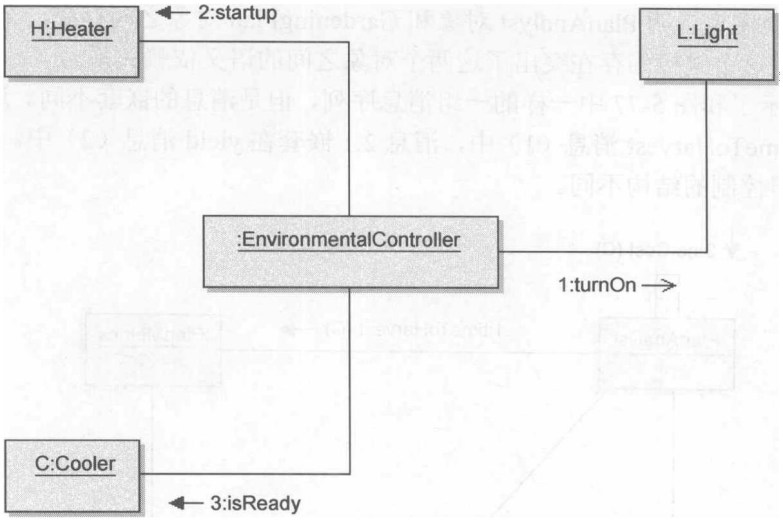


图 5-79 对象与同步

5.14.4 高级概念：迭代子句和警戒条件

还可以在顺序表达式上加上附加信息来精确规定执行发生的方式。可以加上一个迭代子句来说明会发出一系列的消息。迭代子句的形式取决于个人喜好，但是使用伪代码似乎是个不错的选择。图 5-80 展示了在 turnOn()消息上添加的迭代子句。这个修饰以星号开始，后面跟上方括号括起来的迭代子句。这个例子表明，turnOn 消息将依次被发出，从 1 到 n。如果消息是被并发发出的，星号后面可以跟一个双竖线（即*||[i=1..n]）。

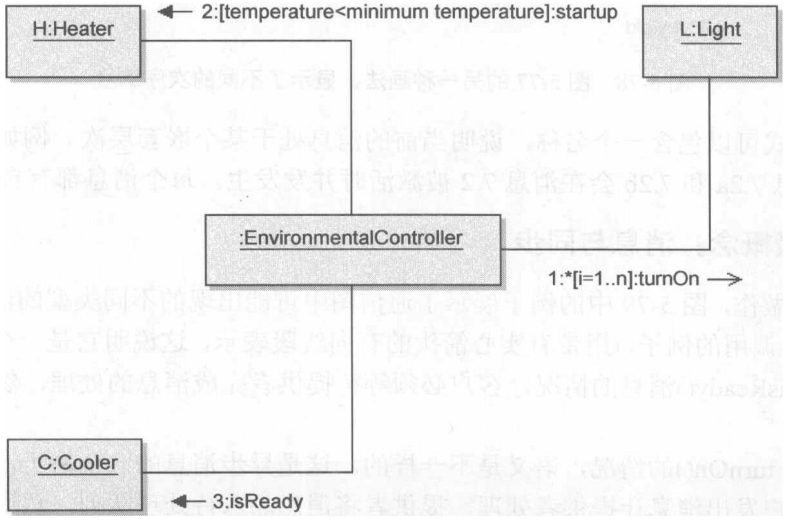


图 5-80 通信图中的迭代子句和警戒条件

警戒条件也可以用于修饰消息。表示法和迭代子句类似，但没有星号。警戒条件放在方括号内，如图 5-80 中 startup 消息上的警戒条件。这个条件表明，消息将在警戒条件为真时执行，在本例中就是当温度低于预期的最低温度时。警戒条件的表示方式取决于个人喜好。

5.15 小结

- 设计不是画一张图，图只是记录了设计。
- 在设计复杂系统时，重要的是从多个角度来看待设计：即它的概念模型、逻辑模型和物理模型，以及它的结构语义和行为语义。
- UML 包括 13 种图：包图、组件图、部署图、用例图、活动图、类图、序列图、交互概述图、组合结构图、状态机图、时间图、对象图和通信图。
- 包图提供了一种手段来组织开发过程中的工件，清晰地呈现问题空间的分析和相关的设计。分包的具体理由不同，但主要关注可视模型本身的物理结构，或者关注清楚地通过多重视图来展现模型元素。
- 组件图展示了组件的内部结构和它们与其他组件之间的依赖关系。这个图提供了组件的表示，这些组件通过定义良好的接口进行协作，提供系统的功能。
- 部署图展示了工件被分配到系统物理设计中的节点的情况。单张部署图代表了系统工件结构的一个视图。在开发过程中，使用部署图来说明一组物理节点的集合，它们是系统执行的平台。
- 用例图描述了待建系统的上下文环境和系统提供的功能。用例图描述了谁（或什么）与系统进行交互，它们展示了外部世界希望系统做些什么。
- 活动图提供了活动流程的可视化描述，可以从系统、业务、工作流或其他过程的角度进行描述。这些图关注执行的活动以及谁（或什么）负责执行这些活动。
- 类图展示了系统的逻辑设计中存在的类和类之间的关系。在分析时，我们利用类图来说明实体共同的角色和责任，这些实体提供了系统的行为。在设计时，我们利用类图来记录类的结构，这些类构成了系统的架构。
- 序列图跟踪了场景的执行，与对象图的上下文背景是一样的。在很大的程度上，序列图就是对象图的另一种表示形式。
- 交互概述图是活动图和交互图的组合，目的是提供元素间控制流程的概述。虽然可以使用任何类型的交互图，但最常用的是序列图。
- 组合结构图提供了一种方式，用以描述结构化类元及其内部结构。这种图在设计时也可以用来将类分解为它们的组成部分，并对各部分在运行时刻的协作进行建模。
- 状态机图用来设计和理解时间关键的系统。状态机图将行为表示为一系列的状态

转换，由事件触发，并与可能发生的动作相关联。这样的状态机也被称为行为状态机。

- 时间图是一种交互图，目的是展示元素状态随时间的变化，以及事件如何改变这些状态。
- 对象图展示了系统的逻辑设计中存在的对象以及对象之间的关系。单张对象图展示了系统对象结构的一张视图，通常用来表示一个场景。
- 通信图是一种交互图，它关注在参与具体的交互时，对象之间如何链接以及传递什么消息。

过程

业余软件工程师总是在寻找神奇的、轰动一时的方法或工具，这些方法和工具承诺用了它们就能让软件开发很轻松。专业软件工程师知道不存在这样的万能药，而业余软件工程师常常希望按照一本指南的步骤去做。专业软件工程师知道这样的开发方式通常会得到设计不当的产品，滋生进度谎言，开发者躲在后面，不愿为早期被误导的决策承担责任。业余软件工程师要么完全忽略文档，要么遵循一个文档驱动的过程，更关心这些文档产品带给用户的观感，而不太关心它们包含的实质内容。专业软件工程师承认创建某些文档很重要，但不会因为文档而放弃进行有意义的架构创新。

面向对象分析和设计的过程不能用一本指南来描述，但是它得到了很好的定义，为成熟的软件开发组织提供了可预测、可重复的过程。本章将详细讨论分析和设计过程（并从总体上讨论完整的软件开发过程），即我们认为每项开发设计活动的目的、产品、步骤和评判标准。

6.1 首要原则

关于过程的讨论从介绍一些首要原则——那些导致成功项目的特征开始。

6.1.1 成功项目的特征

成功的软件项目就是那些提交产物满足或超出客户预期的项目，而且开发过程符合时间上和费用上的要求，结果在面对变化和调整时有弹性。根据这个标准我们注意到，有一些特征[1]基本上是我们遇到的所有成功的面向对象系统所共有的，而且在失败的系统中显然是缺少的：

[1] 一些人可能会辩称，成功的项目有许多其他的特征，我们也同意。但是在本章，我们选择关注这两个特征，因为它们对面向对象的分析和设计过程有直接的影响。

- 存在很强的架构愿景；
- 应用了管理良好的迭代、增量式开发生命周期。

1. 很强的架构愿景

很强的架构愿景基本上是我们遇到的所有成功的面向对象系统所共有的特点。那么，什么是架构呢？IEEE 的“软件密集系统架构描述的推荐实践”（IEEE 1471）将架构定义为“系统的基本组织结构，包括它的组件、组件之间的相互关系、环境，以及设计和演进的指导原则”[42]。现在还有许多其他的架构定义，但大多数定义都表明，架构关注结构和行为，只关注重要的决定，可能符合某种架构风格，受到涉众和环境的影响，包含理性的决定[41]。

具有良好架构的系统是具有概念完整性的系统，正如 Brooks 所强调的，“概念完整性是系统设计中最重要的考虑”[1]。在某种程度上，系统的架构基本上与系统的最终用户无关。但是，对于构建可理解、可扩展、可重新组织、可维护和可测试的系统来说，拥有“清晰的内部结构”是很关键的[2]。只有对系统的架构有清晰的感觉，才可能发现共同的抽象和机制。研究这种共性最终将使得系统的构建变得更简单，从而使系统更小、更可靠。忽视架构愿景将导致烂泥式的软件。

没有正确的方法实现抽象分类，同样，也没有正确的方法设计给定系统的架构。对于任何应用领域，肯定会有一些极为愚蠢的方式，偶尔也会有一些非常优雅的方法，来设计解决方案的架构。那么，如何区分好的架构和差的架构呢？从本质上来说，好的架构应该是面向对象的，由组件构成。这不是说所有面向对象的架构都是好的，也不是说只有面向对象的架构才是好的。但是，正如在第 1 章和第 2 章中讨论的，应用面向对象分解的基本原则通常可以得到一些架构，这些架构展示了我们所期望的组织复杂性的特点。

好的软件架构通常具有以下一些共性。

- 它们由定义良好的抽象层构成，每个层代表了一种内聚的抽象，提供了定义良好的、受控的接口，并且建立在同样定义良好的、受控的低层抽象设施之上。
- 在每一层的接口和实现之间有清晰的分离关注，让我们能够改变一个层的实现而不会破坏它的客户对它的假定。
- 架构很简单：共同的行为是通过共同的抽象和共同的机制来实现的。

以这种方式组织的架构会降低复杂性，更为健壮和可靠。它们也促进了更有效的复用。

敏捷过程倾向于降低早期建立架构的重要性。相反，它们描述了简单设计、突发设计、重构和“偶然发现的”架构等概念[47]。在这样的过程中，架构随时间而演进。后面的小节中将讨论到，对于“追求理性的开发过程”来说，你所选择的方式取决于你的具体情况。在任何情况下，架构在生命周期的何时完成、如何完成都不会降低拥有架构愿景的重要性。没有这样的愿景，系统就较难随时间演进，也更难维护。

2. 迭代和增量式开发生命周期

在少数的应用领域中，要解决的问题可能已经定义得非常清楚，已经有了许多不同的实现。这时，我们有可能完全确定开发的过程：新系统的设计者已经理解了在这个领域中什么是重要的抽象，他们一般知道这样一个系统中预期行为的范围。在这样的过程中，创造性仍

然很重要，但是问题已经在很大程度上缩小了范围，因为系统的大多数重要决定已经确定。在这样的情况下，可能实现很高的生产率，因为大多数的开发风险已被消除[6]。对要解决的问题了解得越多，它就越容易被解决。

大多数的工业级软件问题不是这样的。大多数问题需要折中考虑一组特有的功能和性能需求，这项任务要求开发团队发挥全部的创造力。在这种情况下，不可能提供一个指南式的过程。软件开发和任何要求创造和创新的人类活动一样，需要一个迭代和增量式的过程，这个过程依赖于每个团队成员的经验、智慧和天分[1]。

迭代和增量式开发是以连续系列的发布版本（迭代）的方式开发系统的功能，完成度不断增加（增量）。发布版本可以是外部的（提供给客户），也可以是内部的（不提供给客户）。在一次迭代中选择实现哪些功能是由缓解项目风险来决定的，最重要的风险先处理。作为一次迭代的结果，得到的经验和产物将应用于下一次迭代。通过每次迭代，可以逐渐优化战略和战术决策，最后得到一个满足用户真实需求（常常没有明说）的解决方案。同时，这个方案又是简单、可靠、能适应变化的。

迭代和增量的方式是大多数现代软件开发方法的核心，包括像极限编程（XP）和 SCRUM 这样的方法。它非常适合面向对象的方法，在风险管理方面提供了一些优点。Gilb 恰如其分地指出："演进式[2]的提交是为了针对即将到来的不愉快的事实给我们一些早期的警告信号。"[15]

下面是迭代式开发方式的一些好处[45]。

- 允许需求变更，每次迭代关注一组具体的需求。
- 没有在项目结束时的"大爆炸式"的集成工作，每次迭代都包括这个版本中元素的集成，集成是渐进的、持续的。
- 风险尽早得到关注。早期的迭代缓解了关键的风险，允许在生命周期的更早时候确定新的风险，那时候更容易处理这些风险。
- 可以实现对产品的战术改变。为了针对竞争者的动作，可以对产品或产品的早期版本进行改变。
- 促进了复用。架构的关键组件实际上在早期建立，这样更容易确定可复用的组件以及利用已有组件的机会。
- 缺陷可以更早被发现并修正。每次迭代都执行测试，这样缺陷就可以更早被发现，并在下一次迭代中被修正，而不用等到项目的后期才发现，那时可能已经没有时间来修复缺陷（或者修复缺陷的影响太大）。
- 项目人员的工作更有效。迭代式开发鼓励团队成员在迭代中承担不同角色，这与管道式的组织方式不同。在管道式的组织方式中，分析师转给设计者，设计者转给程

[1] Booch 领导的"Day in the Life"经验性研究证实了这些观点。这个实验是在 2001 年 3 月 27 日（星期二）进行的，涉及 50 名来自世界各地的开发者。Booch 研究了一组开发者，观察他们做了些什么，如何使用工具。通过这项研究，他指出，"个人和团队必须面对高度的不确定性、模糊性和混乱，同时又要求创造性、可预测性和可重复性……开发是一项集体活动……团队生产率的重要性超过了个体程序员生产率的重要性"[40]。

[2] 在演进式的开发中，解决方案随着时间的推移而演进，而并非在项目开始就被定义并冻结。演进式开发非常适合迭代和增量式开发，因为每次迭代都可以利用前一次迭代的反馈信息，这为系统演进提供了一次机会。

序员，程序员转给测试者，如此等等。迭代式开发充分利用了团队成员的经验，消除了转手过程。

- 团队成员不断地学习。每次迭代都让团队成员有机会从过去的经验中学习（“实践创造完美”）。一次迭代中的问题可以在后面的迭代中解决。
- 开发过程可以被优化和改进。每次迭代都会对过程和组织方式中有效和无效的部分进行评估。这些评估的结果可以用于改进下一次迭代的过程。

我们已经讨论了区分成功项目的一些特点，接下来看看目前可用的一些过程的范围，以及实现理性过程的一些策略。

6.1.2 追求理性的开发过程

Parnas 和 Clements 曾说过，“我们永远也不会找到一个过程，让我们以一种完全理性的方式来设计软件”[9]，因为在开发过程中需要创造和创新。但是，他们接着说，“好消息是我们可以伪造这样一个过程……[因为]设计者需要指导，如果我们尝试遵循一个过程，而不是任意为之，我们就能够离理性的过程更近一些。当组织机构承担了很多软件项目时，拥有一个标准的过程是有好处的……如果我们在一个理想过程上达成一致意见，就更容易测量项目的进度。”前面曾提到，重要的是拥有管理良好的迭代增量式生命周期——管理良好是指过程可控并可测量，同时又不至于太严格，不会丧失鼓励创造和创新所需要的自由度。本小节中将讨论当今各种过程风格的范围，同时将在如何选择最符合项目和组织机构需要的过程方面提供一些建议。

在今天的软件开发社区中，有太多的软件开发过程可以选择——Rational 统一过程（RUP）、XP、SCRUM、Crystal 等。选择哪一种软件开发过程将对如何规划和开发软件项目产生深远的影响，甚至可能决定这些项目的成败。因此，这样的决定不能够马虎。好消息是，这样的选择不是非此即彼的。实际上，我们认为所有的软件开发过程都是处于一个过程连续光谱的某个位置，敏捷在一端，计划驱动在另一端[1]。某个具体过程在光谱中的位置要根据它的主要特点和总体策略来确定。

对于敏捷过程来说，其主要目标是在短时间内向客户提交满足他们需求的系统。过程只是实现目标的一种手段。因此，敏捷过程大致有以下特征：

- 轻量级、松散式、不太正式（只做绝对需要做的事情）；
- 依赖于团队成员的隐式知识（而不是有良好文档的过程）；
- 关注战术超过关注战略（不要为将来而构建，因为将来是不可知的）；
- 迭代式和增量式（在几个周期中提供系统的一些部分）；
- 严重依赖于客户的协作（客户积极参与需求确定和验证）；
- 自组织和管理（团队想出最好的工作方法）；
- 突发式而不是预先确定（在实际执行过程中让过程演进，而不是事先计划或确定）。

[1] Boehm 和 Turner 详细讨论了敏捷过程和计划驱动过程之间的差异[38]。

敏捷过程让软件开发团队免于遵循一组严格的步骤，使开发者能够将他们的创造力集中于被开发的系统上。

对于计划驱动的过程来说，除了在可接受的时间框架内向客户提交期望的系统之外，另一个重要的目标是定义和验证一个可预测、可重复的软件开发过程。这个过程不只是实现目标的一种手段，它本身就是一个目标。换言之，除了客户要求的系统之外，软件开发过程本身及其工件都是关键产物。因此，计划驱动的过程大致有以下特点：

- 较重量级、较正式（按照规定的活动得到文档齐全的工件）；
- 依赖于有良好文档的过程（而不是项目团队的隐式知识）；
- 关注战略超过关注战术（建立起强大的架构框架，可以包容将来的变化）；
- 依赖于客户的合约（制定合约并达成一致意见，合约描述了构建内容）；
- 受管理的、受控制的（遵守详细的计划，无论是团队内部还是团队之间都包括明确的里程碑和验证点）[1]；
- 事先定义然后持续改进（包括明确的过程改进流程和机制）。

存在一种常见的误解，即认为敏捷意味着没有过程而全是创造，过程驱动意味着全是过程而没有创造。这是对两种过程风格不正确的描述。敏捷并不意味着缺少过程。敏捷过程的目标是快速应对变化，既针对被开发的应用，也针对过程本身。根据定义，敏捷过程并不比计划驱动的过程更具有创造性和创新性。留出了原型制作时间的迭代增量式计划驱动过程也为创造性和创新性提供了充足的空间。

选择哪一种适合项目的软件开发过程，最好是通过一种两阶段的方式来确定。首先，确定你（你的组织和项目）在过程连续光谱中所处的位置，并选择一种过程风格，作为开发过程的总体指导框架。然后，定制并配置这个过程框架，让它包含来自其他过程风格的技术，这样得到的开发过程将在敏捷和计划驱动技术之间取得平衡，反映出你在过程连续光谱中所处的位置。例如，如果你在过程光谱中更接近敏捷这一端，那么你应该遵循的总体框架和战略就应该是敏捷的。然后，根据距离另一端有多远（计划驱动的一端），在过程中加入并优化一些计划驱动技术。另一点也很重要，即在生命周期的不同阶段，可以采用不同的过程风格，在生命周期的早期可能需要更多的敏捷，而在晚期则需要更多的精确。

为了弄清楚在过程连续光谱中所处的位置，需要用项目的特征和不同过程风格的特征进行比较，并选择最为匹配的过程风格。表 6-1 列出了一般与光谱两端的过程风格相关的项目特征。

表 6-1　　敏捷和计划驱动项目的特征

敏　捷	计 划 驱 动
项目小（5～10 人[a]）	项目大（超过 10 人）
有经验的团队，具有各种能力和技术	团队包括不同的能力和技术

[1] 计划驱动的过程不一定是迭代增量式的（一个项目可能应用严格的瀑布式计划驱动过程），它们也可以是迭代增量式的（我们建议应该如此）。

续表

敏 捷	计划驱动
团队成员是自我驱动的、独立的领导者及其他自己知道方向的人	团队是地理上分散或外包的
项目是内部项目，团队在相同的地理位置	项目具有战略重要性（例如，是一个企业首创的），范围跨越整个组织机构
系统是新的，充满了未知数	系统已经充分理解，具有熟悉的范围和功能
需求必须是已被发现的	需求相当稳定（变化的可能性小），能够事先确定
需求和环境是易变的，变化的可能性很大	系统大而复杂，具有关键安全需求或高可靠性需求
最终用户的环境是灵活的	项目涉众与开发团队之间的关系一般
与客户的关系密切，有很好的合作	存在外部合法性考虑（如合约、义务、针对具体行业标准的正式认证）
客户随时可以找到，全职投入且在同一地理位置	重点是强大的、量化的过程改进
在开发团队内部、开发团队之间、开发团队和顾客之间存在高度信任的环境	重视过程的定义和管理
需要快速价值和快速响应	重视过程的可预测性和稳定性

a. 关于敏捷项目团队规模的讨论，请参见 Boehm and Turner[38]。

虽然定义了过程，但和过程有关的工作并没有结束。随着开发生命周期中发现的问题，应该优化过程（理想情况是在每次迭代之后）。效果很好的过程活动应该保留，效果不好的过程活动应该去除。（然后，继续并重复这一过程。）根据过程执行的实际经验对过程进行持续改进应该是我们的目标。

本章后面的内容描述了一个软件开发过程的框架，它已为面向对象系统的构建进行了调整。这个软件开发过程是从两方面进行描述的——整体软件开发生命周期（宏观过程）和分析设计过程（微观过程）。关于宏观过程的讨论为微观过程设定了上下文，微观过程是本章真正的重点。

定义宏观过程和微观过程时关键的一点就是它们的关注点非常不同。宏观过程关注的是总体软件开发生命周期，而微观过程关注的是具体的分析和设计技术，即从需求到实现过程中使用的技术。生命周期风格（如瀑布式、迭代式、敏捷、计划驱动等）的选择将影响宏观过程，分析和设计技术（如结构化、面向对象等）的选择将影响微观过程。因此，不论选择敏捷还是计划驱动作为宏观过程，6.3 节中描述的面向对象分析和设计技术都将同样适用。

6.2 宏观过程：软件开发生命周期

宏观过程是总体的软件开发生命周期，它是微观过程的控制框架（本章稍后将描述微观过程）。它代表了整个开发团队的活动，因此，宏观过程规定了一些可测量的产物和活动，让

开发团队能够有意义地评估风险，并尽早对微观过程进行调整，以便更好地关注团队的分析设计活动。

我们前面曾提到，存在一个软件开发生命周期风格的连续光谱可供选择——从瀑布式到迭代式，从敏捷到计划驱动，以及它们之间的多种可能。对生命周期风格的选择将直接影响宏观过程的规模和表现（例如，开发阶段的定义和数量、推荐的迭代周期、平均的迭代数等）。

本节中将介绍一个计划驱动的宏观过程的例子，它直接支持我们在本章开始时讨论过的两种特征：很强的架构愿景和迭代式开发生命周期。我们描述的宏观过程是 RUP 生命周期[51]。我们以它为基础，比较其他可能的生命周期。关于不同敏捷方法的生命周期的比较，请参见 Larman[46]。

6.2.1 概述

宏观过程的目的是指导系统的整体开发，最终得到产品系统。宏观过程的范围从确定想法开始，直到实现该想法的第一个软件系统版本为止。

系统后续版本的开发，不论是为了演进还是为了维护，都通过执行另一次宏观过程来实现（另一个生命周期）。维护生命周期的规模和表现是最初的开发生命周期的变体。关于维护生命周期的更多信息，请参见补充材料“移交之后软件的演进和维护”。

本小节将关注迭代增量式的宏观过程。在这样一个过程中，宏观过程确定了系统将以演进式的方式，通过不断的优化，最后得到产品系统。这个过程的主要产物是一系列可执行的发布版本（迭代），体现了系统不断的优化（增量）。次要的产品包括系统行为原型[1]，目的是探索可选的设计或进一步分析系统功能中不明确的部分。次要的产品还包括用于记录设计决策和理由的文档[2]。

宏观软件开发过程可以从两个维度进行描述，即内容和时间——做什么和什么时候做。内容这一维描述了角色、任务和工作产品，也可以从科目或考虑的方面来描述，从逻辑上对内容进行分组。时间这一维描述了这个过程的生命周期，可以从里程碑、阶段和迭代的角度来描述。

移交之后软件的演进和维护

当一个软件系统被交付之后（移交之后），很可能需要对已部署的系统进行变更。这些变更可能是提供新的、改进的功能（演进），或是修复发现的缺陷（维护）。维护一个运营的系统而不破坏原有的功能，这是我们真正关注的问题。

关于已部署系统的成熟度，Lehman 和 Belady 提出了一些令人信服的观点。

[1] 关于系统行为原型的更多信息，请参见补充材料“在软件开发过程中制作原型”。

[2] 许多开发者受困于“不必要的文档”，因为他们没有从中感受到好处。但是，我们必须指出，文档的读者很少会是当前的开发团队，通常是团队以外的人（诸如集成者、数据库管理员、项目经理、运营支持团队、技术支持人员等）或将来加入团队的人。因此，只要文档将来会有人阅读，就应该创建。在开发时，文档应该随着系统一起演进，而不是被当作独立的里程碑，它应该是过程中自然的、半自动生成的产物。

- 真实世界中使用的程序必须进行变更，否则它在环境中的作用就会越来越小（不断变更法则）；
- 随着一个不断演进程序的变更，它的结构会变得更复杂，除非通过积极的工作来避免这一现象（复杂性增加原则）。

在敏捷开发中，对已部署的系统进行优化是尤其重要的。实际上，敏捷开发的一项关键原则就是“通过尽早交付和持续交付有价值的软件来满足客户”[35]。对于敏捷过程来说，已部署的系统会被不断重构，在简化结构的同时又不破坏原有的实现（例如，必须通过所有测试）。

要对已完成开发并已部署的软件系统进行变更，可以重新执行与原来软件开发生命周期类似的生命周期，维护生命周期的规模和表现方式（如需要哪些阶段，每个阶段需要几次迭代）取决于在这次发布版本中需要完成的工作。某些维护发布版本涉及简单的本地化修改，不包含架构的创新（例如，主要包括翻译阶段），但另一些维护发布版本需要考虑范围和业务价值，同时也要考虑架构和风险（例如，要包括“初始”阶段和“细化”阶段）。这样的维护发布版本被认为更重要，它们的生命周期更像是从头到尾的完整过程。关于维护生命周期的讨论，请参见 Kruchten[44]。

演进和维护生命周期的产品与最初发布生命周期的产品类似，但多了一份变更请求列表。在产品系统发布之前不久，系统的开发者和最终用户可能已经发现了一些改进或修改，他们希望在后续的产品发布版本中实现。出于业务上的原因，没有能够在初始的产品发布版本中实现这些改进或修改。另外，随着更多用户使用该系统，新的缺陷和使用模式会被发现，而品质保证团队没有预期到这些问题。（在以想不到的方式使用系统这方面，用户的创造性惊人。）变更请求列表用于收集缺陷和增强需求，这样，它们可以按优先级排列，在将来的版本中实现。

在移交后执行的活动与在系统开发时要求的活动是类似的。但是，除了一般的软件开发活动之外，移交后的发布计划还包括对变更请求排列优先级，评估它们对开发的影响和成本，将变更分配到某个发布版本中。同时，某些操作问题必须在 24 小时（或更短的时间）内解决。因此，必须在正常的发布机制之外交付补丁。配置和测试这些变更，并将它们集成到当前的开发版本中，可能是重要的项目活动。

6.2.2 宏观过程的内容维：科目

宏观过程包括下列科目，按相对次序来执行。

（1）需求　对于系统该做什么，建立并保持与客户和其他涉众的一致意见。定义系统的边界（划定界限）。

（2）分析与设计　将需求转化为系统设计，设计将作为在特定实现环境中进行实现的规格说明。这包括逐渐形成一个健壮的系统架构，建立起系统不同元素必须用到的共同机制。

（3）实现　实现、单元测试以及对设计进行集成，得到一个可执行的系统。

（4）测试　对实现进行测试，确保它实现了需求（即需求得到了正确地实现）。通过具体的展示来验证软件产品是否像设计的那样工作。

（5）部署　确保软件产品（包括已测试的实现）能被它的最终用户使用。

在生命周期中将执行下列科目：

- 项目管理　管理软件开发项目，包括计划、人员配置、监控项目以及管理风险；
- 配置和变更管理　标识出配置项，控制对这些配置项的变更，管理这些项的配置；
- 环境　提供软件开发环境，包括支持开发团队的过程和工具。

在软件开发过程中制作原型

制作原型在软件开发过程中有多种目的。在讨论制作原型的作用之前，让我们先定义什么是所谓的原型，具体来说，什么是行为原型。行为原型探索系统中某些独立的部分，如新算法、用户接口模型或数据库方案。它的目的是快速探索可选的设计，这样可以尽早解决某些风险，使产品的发布版本免受危害。从本质上来说，这样的原型是不完整的，只完成了起码的工作，在它们完成任务之后是要被抛弃的。

首次使用行为原型通常是在软件开发生命周期的早期，这时你正试图理解可以构建什么，可以利用什么技术来构建。对于每个重要的新系统来说，都应该进行一些概念验证，这是通过一些快速而粗糙的原型来实现的。显然，对于范围很大的应用来说（如对整个国家有重要意义或牵连多个国家的应用），原型工作本身就有很大的工作量。这样的原型工作是在预期之中的，实际上是受鼓励的。在概念验证时发现功能、性能、规模、复杂度等方面的假定有错误，比后来才发现要好得多。后来再抛弃既定的开发路线可能带来财务或社会灾难。

行为原型也可以在整个软件开发生命周期中使用，目的是更好地理解系统行为的语义。通常，开发团队会利用行为原型向最终用户展现用户界面的语义，以获取早期反馈意见，或者为实现某些机制进行性能上的折中。

任何编程方式都应该支持概念验证原型的开发，以便让开发团队更好地理解一种思想，揭示风险。但是，在很多情况下，因为存在相当丰富的面向对象应用框架，开发原型的速度常常比其他方式要快。我们也常常看到，概念验证原型是用一种语言开发（如Smalltalk），而最终产品是用另一种语言开发（如Java）。

原型不应该直接发展成为产品系统，除非有非常好的理由。为了满足短期时间表的方便显然不是一个好的理由：这样的决定代表了一种虚假的节约，有利于短期开发，但忽略了软件的拥有成本。在开发原型时，重要的是为每个原型建立起清楚的目标和完成标准。在完成时，决定采用一种方式将原型工作的结果集成到当前、后续的发布版本中。

图 6-1 展示了宏观过程科目的相对顺序和迭代实质。经过这些科目的每个循环构成了宏观过程的一次迭代。

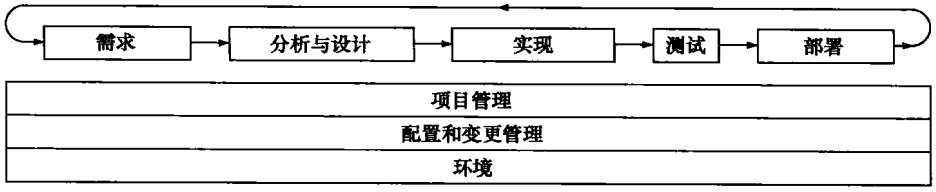


图 6-1 宏观开发过程科目

重要的是要强调，虽然这些科目一般是按顺序执行的（先是需求，然后是分析与设计，再然后是实现，等等），但宏观过程不一定是瀑布式的（也可以是）。在瀑布式宏观过程中，这些科目只经过一次——定义整个系统的需求，然后对整个系统进行分析和设计，等等。在迭代增量式宏观过程中，这些科目会经过多次，每次经过每个科目时执行的工作范围取决于你在整体开发过程中所处的位置。在我们讨论里程碑和阶段时，这一点会更清楚。

宏观过程的许多部分就是很好的软件管理实践，对于面向对象系统和非面向对象系统同样合适。其中包括一些基本实践，如需求管理、配置管理、测试和品质保证、代码走查以及编写文档。

讨论了宏观过程的内容维之后，接下来让我们关注时间维，这一维可以从里程碑、阶段和迭代的角度来描述。

6.2.3 宏观过程的时间维：里程碑和阶段

在迭代增量式的宏观过程中，一些科目是重复执行的。但是，迭代式开发过程不仅仅是一系列的迭代。必须有一个整体的框架，在这个框架下，迭代的执行反映了项目的战略计划，驱动着每次迭代的目标。这样的框架可以由一系列定义良好的里程碑来提供，通过执行一次或多次迭代来实现每个里程碑的目标[1]。在每个里程碑处将进行一次评估，确定目标是否实现。评估满意就让项目继续进入下一个阶段，去实现下一个里程碑。

里程碑确保迭代取得进展，最后收敛成一个解决方案，而不仅仅是不确定的迭代。它们不应该仅被看成是项目进度表上的一个日期，而应该被看成是质量或成熟度的关口，实现了这些里程碑就意味着项目已到达了某个成熟度，对演进的计划、规格说明和完成的解决方案有了进一步的理解。如果最初为某个里程碑设置的日期到了，但项目没有达到预期的成熟度或理解程度，那么里程碑日期就延迟了——这个日期是可变的，不是里程碑的评判标准。

图 6-2 展示了在一个迭代增量式宏观过程中，里程碑和迭代是如何结合的，也展示了里程碑描绘的各个阶段。

下面将详细描述每一个阶段。

1. 初始

本小节将介绍初始阶段的目标、活动、工作产品和里程碑。

[1] 瀑布式的宏观过程也有里程碑，但这些里程碑代表着整个系统的各个科目的完成（如需求完成、分析与设计完成等）。

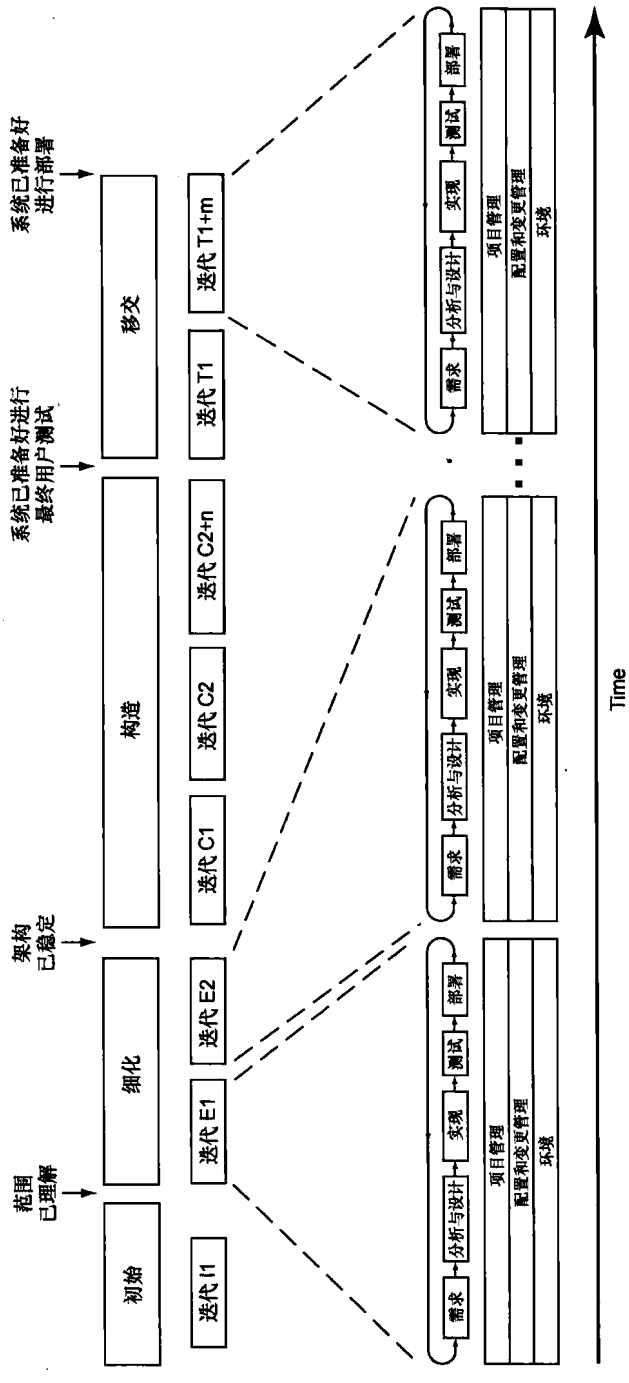


图 6-2 宏观过程里程碑、阶段和迭代

目标 初始阶段的目标是确保项目有价值并可行（范围和业务价值）。对于任何真正的新软件甚至是原有系统的新改写，有时候会在开发者、架构师、分析师或最终用户的头脑中，突然出现一个想法。这个想法可能代表一种新的业务冒险，在原有产品线中的一种新的补充产品，或者是已有软件系统的一组新功能。初始阶段的目的不是完全地确定这个想法，而是建立对这个想法的愿景并验证它的假定。即使是对一个已有系统的优化，初始阶段的工作也是有价值的。在这种情况下，初始阶段较为简单，但仍然关注于确保业务价值和技术可行性。

活动 在初始阶段，建立起系统的核心需求并排出优先级，对应该构建什么与顾客达成一致意见，并确保理解了与构建系统相关的主要风险，决定使用什么开发环境（包括过程和工具）。还记得前面讨论的选择最合适当前情况的过程吗？初始阶段就是做决定的时候，我们定制开发过程，选择支持该过程的工具。在初始阶段，没有什么活动是面向对象开发所特有的。

工作产品 初始阶段的主要工作产品是要构建的系统的愿景、行为原型、初始风险列表、确定的关键架构机制及开发环境。系统愿景提供了对待构建系统的清晰描述，包括它的范围、关键特征、对已有系统的影响与关系，以及必须考虑的所有限制条件。原型提供了概念验证，说明系统是可以构建的。风险列表确定了一些关键风险，必须在生命周期的早期缓解这些风险才能增加成功的可能性。架构机制定义了系统的一般能力，这种能力支持着基本的系统功能（如用户接口方式、错误检测和处理、持久、内存管理、进程间通信、事务管理和安全性等）。开发环境包括遵守的开发过程以及支持该过程的开发工具。

里程碑：范围已理解 如果清楚地理解了要构建的系统（系统的总体范围和关键需求），理解了这些需求的相对优先级以及构建该系统的重要业务原因，初始阶段就成功完成了。另外，客户和开发组织对系统的范围和总体交付时间表也达成一致意见。

2. 细化

本小节将介绍细化阶段的目标、活动、工作产品和里程碑。

目标 当已经理解待构建系统的范围并达成一致意见之后，我们的注意力就转向开发整体架构框架，这个架构框架将为后续的所有迭代奠定基础。我们的目标是，尽早确定架构的缺陷，建立起一些通用的策略，以得到一个更简单的架构。细化阶段是对架构进行探索、选择，并通过多次迭代逐渐形成架构。这种形成过程是通过缓和最大风险、实现最高优先级的需求和实现架构最重要的部分来驱动的。

活动 细化阶段包括制定架构方面的决定、建立架构框架、实现该框架、测试该框架，以及根据测试的结果优化该框架。架构的演进基本上就是努力满足一些相互竞争的约束条件，包括功能、时间和空间——我们总是受到最严格的约束条件的限制。例如，如果计算机的重量是一个关键因素（例如在设计航天器），那么我们就必须考虑单个内存芯片的重量，从而必须考虑这种重量影响到内存的容量，容量又限制了能够加载的程序的规模。放宽某个限制条件，其他的设计选择就成为可能；加强某个限制条件，某些设计就变得很难办。通过演进软件系统的架构，而不是以一种更僵化的方式来开发，就可以确定哪些约束条件是真正重要的，哪些限制条件只是幻觉。在细化阶段的早期，你通常知道得不多，不了解系统中哪里会成为

性能瓶颈。通过实际构建那些关键的架构部分，并通过测试来测量结果，开发团队就能更好地理解如何随时间的推移来优化架构。

工作产品 在细化阶段，架构是通过创建一系列可执行的架构发布版本来验证的，这些架构发布版本部分地满足了最终用户关键使用场景的语义（在架构上很重要的场景）。这些场景执行并测试了系统的主要部分以及这些部分的协作，并对已确定的风险领域进行探索。这些架构发布版本代表了整个架构中一个水平的切片，记录了所有重要系统部分的重要语义（但不是全部语义）。因此，细化阶段的结果不只是提供了一份架构文档，也包含了系统一些真正的发布版本，这些发布版本成为架构设计本身的实实在在的产物。架构发布版本应该是可执行的，这样就允许我们对架构进行精确地探测、研究和评估。这些架构发布版本将成为演进产品系统的基础。

里程碑：架构已稳定 如果已针对所有关键系统需求（包括功能需求和非功能需求）进行了验证（通过实际的测试和正式的复查），并且所有风险都得到适当缓和，从而能够预期系统开发完成的费用和时间进度，那么细化阶段就成功完成了。架构已稳定的一个关键标志就是关键架构接口和机制的变化率已大大降低，或者完全消除。对架构接口和机制的变化率的测量就是架构稳定性的首要评判标准[30]。局部变更在软件生命周期中是预期会发生的，但如果关键架构部分经常变动，则表明存在某些架构问题。这应该被看作是一个风险领域，说明细化阶段仍在继续。

3. 构造

本小节将介绍构造阶段的目标、活动、工作产品和里程碑。

目标 当架构稳定之后，我们关注的焦点就从理解问题和确定解决方案的关键部分转向可部署产品的开发。构建阶段就是从探索转向生产，其中，生产可以看作是“一个受控的方法学过程，将产品的品质提升到可以发布的水平”[24]。

活动 在构造阶段，我们完成系统的开发，以细化阶段得到的基线架构为基础。

工作产品 在构造阶段的迭代中，会得到一系列可执行的发布版本，满足剩下的最终用户场景的语义。随着这些发布版本在范围上逐渐增长，它们可以被精确地探测、研究和评估，并演进为产品系统。

里程碑：系统已准备好进行最终用户测试 如果发布版本的功能和品质足以部署给最终用户进行最终用户测试，构造阶段就成功地完成了。这个阶段的一些主要的好坏评价标准包括：发布版本的需求满足到什么程度，以及这些发布版本的品质。在这个阶段，品质的一个重要标志就是缺陷发现率。缺陷发现率是一个测量指标，说明检测出新的错误有多快[29]。通过在开发过程的早期对品质保证进行投入，我们就可能对每个发布版本建立起品质评判标准，管理团队可以利用这些指标来确定风险区域，并对开发团队进行一些修正。在每个发布版本之后，缺陷发现率通常会上升。缺陷发现率停滞不前通常表示存在未被发现的错误。不成比例的缺陷发现率通常表明架构还未稳定，或者某个发布版本中新的部分的设计或实现不正确。不论是哪种情况，系统都没准备好进行最终用户测试，这些测量指标应该作为调整后续发布版本的关注重点。

4. 移交

本小节将介绍移交阶段的目标、活动、工作产品和里程碑。

目标 在移交初始阶段中，确保软件被它的最终用户接受。

活动 在移交阶段，产品被提供给用户进行评估和测试（如 alpha 测试、beta 测试等）。然后开发团队汇集收到的反馈意见。移交阶段的重点是产品调优，处理配置、安装和易用性问题，处理早期使用者所提出的问题。支持文档在进行最后的开发，也包括所有相应的培训材料。所有产品相关的问题，如包装和市场宣传材料，也得到解决。然后，将对得到的产品进行验收测试。重要的是要注意到，即使在整个生命周期中都在进行测试，最终用户测试和最后的验收测试仍然很重要，因为这些测试确保了开发的产品在开发环境和目标安装环境下都实现了它的验收标准。

工作产品 移交阶段的工作产品包括包装好的产品、所有的支持文档、培训材料和市场宣传材料。

里程碑：系统已准备好进行部署 如果发布版本的功能和品质足以将产品提供给最终用户使用（系统通过了验收测试），移交阶段就成功地完成了。发布版本好坏的首要评判标准与构造阶段的指标类似，即报告的缺陷率的下降。但在这个阶段，是早期试用者在报告缺陷。

敏捷方法中的阶段

敏捷方法中也有阶段概念。在这份补充材料中，我们总结了一些敏捷方法中定义的阶段[48]。

XP 生命周期包括 5 个阶段。

（1）探索 确定可行性，理解第一个发布版本的关键“故事”，开发探索原型；

（2）计划 对第一个发布版本的日期和故事达成一致意见；

（3）迭代发布 通过一系列的迭代来实现并测试这些故事，优化迭代计划；

（4）产品化 准备支持材料（文档、培训、市场），并部署可操作的系统；

（5）维护 修复和扩展已部署的系统。

SCRUM 生命周期包括 4 个阶段。

（1）计划 建立愿景、设定期望值、确保资金、开发探索原型；

（2）筹划 为首次迭代准备优先级和计划，开发探索原型；

（3）开发 通过一系列的冲刺来实现需求，并优化迭代计划；

（4）发布 准备支持材料（文档、培训、市场），并部署可操作的系统。

可以看到，计划驱动和敏捷方法中定义的阶段是非常类似的。具体来说，我们介绍的所有方法都包含以下阶段：

- 建立愿景、可行性和范围；
- 发布和迭代计划；

- 实现和测试；
- 产品化和部署。

在介绍了一些宏观过程和里程碑的例子之后，接下来我们将关注的焦点转移到每个阶段中发生的事情——迭代。

6.2.4 宏观过程的时间维：迭代

如图 6-3 所示，在迭代式的宏观过程中，里程碑是通过执行一次或多次迭代来实现的，这些迭代可能包含部分或全部科目的活动。但是，根据这次迭代所处的阶段，在不同科目上所花的时间会有不同。如果迭代发生在初始阶段，我们会花更多的时间在需求上；如果迭代发生在细化阶段，我们会花更多的时间在分析和设计上（具体来说是架构）；如果迭代发生在构造阶段，我们会花更多的时间在实现和测试上；如此等等。当然，某些科目，如配置和变更管理、环境、项目管理，是在整个生命周期中都执行的活动。

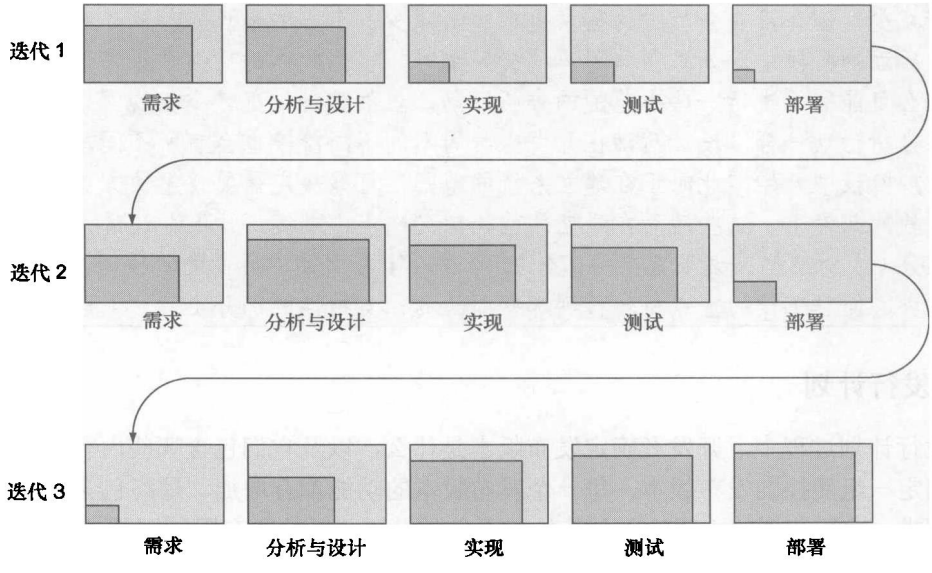


图 6-3 迭代关注焦点的转移

图 6-3 展示了项目关注的焦点是如何在一系列迭代中转移的。每个科目中矩形的大小代表了执行这个科目的活动所花的时间。关于分析和设计活动如何在迭代增量式生命周期中不断变化，请参见补充材料“分析与设计和迭代式开发”。

在每次迭代结束时，会进行事后分析，目的是从构建系统的状态、开发环境和团队的角度对这次迭代进行评估。每次迭代都应该被看成是调整项目路线的一次机会，要么调整后续迭代中实现的功能，要么优化环境，以改进那些效果不太好的方面。

迭代的概念在大多数软件开发方法中基本上是一样的。不同之处在于每次迭代推荐的时

间[46]：

- XP 推荐，如果可能，迭代的时间应该是一周或两周；
- SCRUM 规定所有的迭代（冲刺）应该是 30 天；
- RUP 推荐迭代的时间应该是 2～6 周。

前面曾提到，宏观过程的一项关键提交产物就是一系列的、逐渐演进的发布版本。因此，我们将在最后讨论发布版本计划，结束本节关于宏观过程的讨论。

分析与设计和迭代式开发

在迭代式开发生命周期中，分析和设计活动在开发生命周期的不同阶段会有所不同。

分析活动在生命周期的早期阶段（初始阶段和细化阶段）是很显著的，这时关注于建立软件的架构。在这些阶段，注意力集中在分析那些对架构有重要影响的需求上。在后面的阶段中，将完成实现，分析剩下的需求，但是这种分析不像在确定架构时那样广泛，因为大多数的主要系统部分都已经确定了。在后面的阶段中，在分析上所花的时间将逐渐减少，因为没有分析过的需求也在逐渐减少，工作的重点转移到了系统实现上。但是在后面的阶段中，如果在向用户移交系统时，根据收到的反馈意见需要引入需求变更，那么可能需要进行一些小范围的分析活动，虽然这种情况较少出现。

像分析活动一样，设计活动在开发生命周期的不同阶段也会有所不同。设计活动可以在早期阶段中开始，此时正在建立系统的范围，可能决定需要将解决方案建立在一些已有的软件部件上。这些活动在早期的架构定义迭代中继续，这时关注的是设计系统的主要部分（或对架构来说很重要的部分）。当进入生命周期中后面的阶段时，设计活动会逐渐减少，而且工作的重点将转移到那些周边的、支持性的部分上。

6.2.5 发行计划

在发行计划过程中，开发者确定发布版本是什么，以及它们包含哪些内容。发行计划的目的是确定一组受控的发布版本，每一个发布版本的功能都有增加，最终包含整个产品系统的全部需求。发行计划的主要输入是待构建系统的范围，以及所有限制因素（如费用、时间、品质）。在发布版本计划过程执行的活动包括建立项目的开发节奏，排列需求的优先次序，将需求分配到各次迭代中，确定迭代的发布版本是外部发布版本还是内部发布版本，最后制定详细的迭代计划。发行计划过程的结果是一份开发计划，它确定了一系列的发布版本、团队活动和风险评估。接下来将更仔细地讨论每一种发行计划活动。

在发行计划过程中，第一步就是建立项目的开发节奏——决定迭代的平均时间（即决定发布版本的时间间隔）。在决定认为单次迭代能完成多少工作时，迭代的时间是一个重要的因素。迭代发布日期应该间隔够长，以提供足够的开发时间，并允许发布版本与其他开发活动同步，例如文档编写和现场测试。对于从头到尾的开发时间在 6～12 个月的小项目来说，这可能意味着每 2～6 周发布一次；对于从头到尾的开发时间在 12～18 个月的中等规模的项目

来说，这可能意味着每 2～3 个月发布一次；对于需要更大开发工作量的、更复杂的项目来说，这可能意味着大约每 6 个月发布一次。更大的发布间隔是值得怀疑的，因为这样就不能强制实现微观的过程，可能隐藏有意或无意被忽略的风险。

知道迭代的时间将是多长之后，发行计划过程的下一步就是确定要交付的系统需求的优先级，包括功能需求和非功能需求。这些优先级将用于确定哪些需求被分配到哪些迭代中。

需求是根据一些因素来排列优先级的。这些因素可能包括以下几点：

- 对涉众的好处（例如，该需求对最终用户有多重要，或者它对于向项目出资人展示系统功能的一个一致的部分有多重要）；
- 对架构的影响和覆盖（例如，该需求是否涉及架构的关键方面，如访问数据库、与遗留系统集成，等等）；
- 关注该需求将缓解的风险（例如，该需求是否涉及对外部系统的访问，而该系统的接口并不是非常清楚）。

根据在开发生命周期中所处的位置，这些因素中的每一个可能具有不同的权重。例如，在细化阶段中被认为具有高优先级的部分与在构造阶段中被认为具有高优先级的部分是不同的（在细化阶段，架构重要性具有更大的权重）。重要的是要注意，排列需求优先级不是一次性的工作。它们的相对优先级应该在每次迭代时进行必要的评估和调整，其根据就是当前项目的状态、新的需求、新发现的风险以及对已有风险的缓解方案。由于多个因素将影响到优先级，确定需求的优先级最好是由一个团队来完成，这个团队的组成人员包括用户代表、领域专家、分析师、架构师、品质保证人员。

当需求的优先级被确定之后，需求就被分配到一系列的发布版本中，最高优先级的需求被分配到较早的迭代中。每次迭代都应该有一个计划好的效果，这个效果可以展示，有清晰的评估标准，可以用于评估这次迭代是否成功。迭代发布版本的内容是由迭代的范围来确定的，迭代的范围则是由迭代在软件生命周期中所处的位置来决定的（即迭代处于哪个阶段）。

每次迭代将得到一个发布版本，可以是内部的，也可以是外部的。最后的发布版本是一个外部发布版本，代表了产品系统。确定一个发布版本是内部的还是外部的取决于整体生命周期的阶段。在开发过程的早期，发布版本通常是内部的。主要的、可执行的发布版本由开发团队提交给品质保证人员，品质保证人员开始针对需求中建立的场景进行测试，然后收集关于这个发布版本的完整性、正确性和健壮性的信息。这种早期的信息收集有助于确定品质问题，这些品质问题可以在后续的发布版本开发中较早得到解决。在开发过程的后期，外部的发布版本更多，因为可执行的发布版将以受控的方式被提交给特定的最终用户（参与 alpha 测试和 beta 测试的客户）。所谓“受控的方式”，指的是开发团队小心地为每个发布版本设定期望值，并确定希望评估哪些方面。一般来说，开发团队会有更多的内部发布版本，只有少数可执行的发布版本会被提交给外部人员。内部发布版本代表了系统的一种持续集成，以及在某些关键系统部分存在着强制的结束。

请注意，创建一个发布版本的动作的成本是相对较高的（特别是外部发布版本），所以其他的一些因素，如时间、品质和范围，可能对发布版本的数量和时间间隔产生约束。在这

些情况下，如果强制以固定的费用、时间或范围来提交发布版本，品质可能就会打折扣。对于固定费用的合约来说，尤为如此。

发行计划的最后一项活动是制定详细的迭代计划。在迭代计划过程中，我们针对当前迭代制定详细的项目计划，并确定实现该发布版本所需的开发资源。与总体发行计划不同（它是事先制定的，确定了关键的里程碑、建议的迭代数目以及对它们的内容的较高层面的理解），详细的迭代计划是及时制定的（在迭代将开始时）。这让项目经理能够应对开发过程中不可避免的进度计划调整。“不合理的精确（在需求或计划中）已被证明是成功的显著而微妙的常见障碍。大多数时候，早期的精确只是不诚实的表现，它提供了某种更多品质提升的假象，而实际却不是这样。”[50]

在迭代开发中，发布计划是不断进行的，并且是风险驱动的。在每次迭代之后，剩下的开发计划应该重新检查，并根据需要进行调整。通常，这涉及重新调整需求的优先级，对日期的小调整，或者将功能从一次迭代移到另一次迭代。在生命周期中，应该进行定期的风险评估，并调整开发计划，先处理有风险的工作，这样风险就能被消除或减少，有助于开发团队管理将来的战略和战术上的折中。在开发过程的早期面对风险，这使我们更容易在后来进行实际的架构折中。

6.3 微观过程：分析与设计过程

在前面一节中讨论了整体软件开发过程（宏观过程）。本节中将讨论分析和设计过程（微观过程），看看哪些活动得到执行，产生了哪些工作产品。

6.3.1 概述

如图 6-4 所示，分析和设计过程是在整体软件开发过程的背景下进行的。宏观过程驱动了微观过程的范围，为微观过程提供了输入，并利用了微观过程的输出。具体来说，微观过程使用了宏观过程提供的需求（以及前面的微观过程迭代得到的分析和设计规格说明），产生了设计规格说明（最显著的是架构），然后在宏观过程中实现、测试并部署。

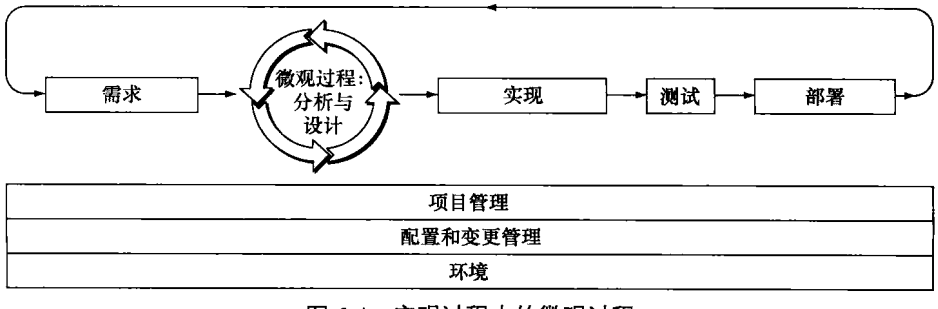


图 6-4 宏观过程中的微观过程

正如我们从两个维度（时间和内容）来描述宏观过程一样，我们也将从两个关键的维度来描述微观过程，即抽象层次和内容（活动和工作产品）。然后我们将讨论如何执行活动，以及得到的工作产品如何受到抽象层次的影响。

6.3.2 抽象层次

在微观过程中，传统的分析阶段和设计阶段被有意地模糊了，取而代之的是进行不同层次的抽象，形成一个连续光谱。分析利用系统需求作为输入，得到初始的解决方案；设计利用分析的结果作为输入，得到能够具体实现的规格说明。如果分析准确地展现了需求，并且是一致的，能够很好地作为设计的基础，那么我们就认为分析已经完成了。如果设计已经足够详细，能够进行实现和测试，那么我们就认为设计已经完成了。Mellor 等人指出，“分析的目标是提供对问题的描述。这种描述必须完整、一致、可读，并可以让不同的感兴趣的团体进行复查，可以针对现实进行测试”[16]。用我们的话来说，分析的目的是提供一个系统行为的模型。

分析关注的是行为，而不是形式。在分析时尝试对世界进行建模，从问题域的词汇表来确定元素，并描述它们的角色、责任和协作。在分析过程中，追求表示或实现问题是不恰当的。分析必须说明系统做什么，而不是系统如何做。只有出于暴露系统的行为的目的，而不是作为可测试的设计需求，在系统分析时有意说明“如何做”才是有用的。分析就是要更好地理解待解决的问题。分析是整体软件开发过程中的一个关键部分，如果执行得好，将导致更健壮、更好理解的设计，得到清晰的分离关注，在系统各元素间平衡地划分责任。

在设计时，创造一些元素，它们提供了分析元素所要求的行为。当得到了关于系统行为较为完整的模型时，就可以开始设计过程了。重要的是要避免不成熟的设计，即在分析还未结束之前就开始设计。同样重要的是避免延迟设计，即组织机构进行彻底的讨论，试图得到一个完美的分析模型，因此而无法实现（这种情况常常被称为分析麻痹症）。在分析过程中，不应该预期得到系统行为的彻底理解。实际上，在设计开始之前拿出完整的分析是不可能的，也是不值得追求的。构建系统的过程中会提出一些行为方面的问题，任何合理工作量的分析都不能有效地揭示这些问题。只要完成系统所有主要行为的分析，并稍稍涉及一些次要行为，确保没有遗漏基本的行为模式，这样就足够了。

因为架构在整体解决方案中占据着非常重要的位置，所以我们需要在开发架构时理解分离关注，同时在分析和设计时理解单个的组件。架构主要考虑的是系统各部分（即组件）之间的关系，以及它们的责任、接口和协作。相反，系统组件的分析和设计关注的是这些组件的内部，以及它们如何满足需求，这些需求是架构的分析和设计要求组件实现的。图 6-5 总结了分析与设计应该关注什么，既包括从架构的角度，也包括从组件的角度。

在整个开发生命周期中，分析与设计是在不同的抽象层次上完成的。抽象层次的数目不能够事先确定。这主要依赖于系统的规模。例如，实际上，有时可能会发现试图分析的组件太大了。可能要后退一步，对这个组件进行另一轮的架构分析，将它进一步划分成几个组件（或子组件），以便于管理或更好地分析。

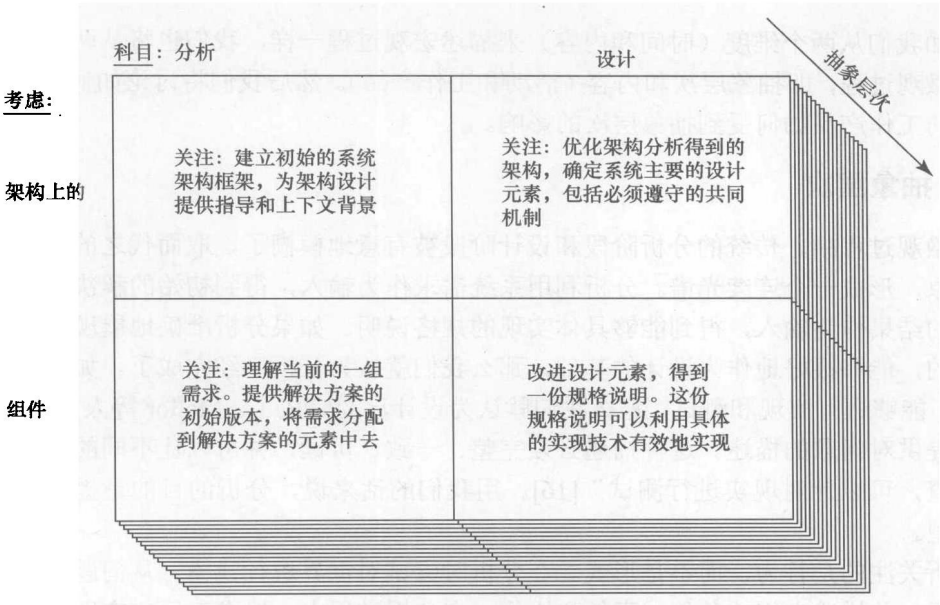


图 6-5 从不同视角来看分析与设计的不同关注点

接下来，让我们看看微观过程中执行的活动，以及得到的工作产品。

6.3.3 活动

微观过程包括下列一些活动，它们是针对一个具体的范围和一个具体层次的抽象来执行的：

- 确定元素[1] 发现（或发明）要处理的元素，确定面向对象的分解；
- 确定元素间的协作 描述已确定的元素之间如何协作，以提供系统的行为需求；
- 确定元素间的关系 确定元素之间的关系，以支持系统的协作；
- 确定元素的语义 建立已确定的元素的行为和属性，为下一层次的抽象准备元素。

微观过程的活动如图 6-6 所示。

虽然这些活动看起来是依次执行的，但实际上它们却是并行执行的。例如，可能在同一时间确定元素以及元素间的协作，也可能在确定元素间的协作时确定其行为和属性。这种能力来自于经验。请将微观过程的执行想象为针对当前的范围一遍遍地执行这些活动，但是，当你对这个过程更有经验时，执行的遍数可以很少。

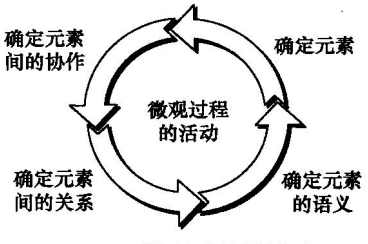


图 6-6 微观过程的活动

本章稍后将会更详细地讨论微观过程的每一项活动。接下让我们来看看微观过程的产品。

[1] 在本章全章中，我们都使用术语“元素”来指代在当前抽象层次要处理的“东西”。因此，一个元素可能是一个分析类、一个组件、一个设计类等。关于分析和设计中确定的元素的更多信息，请参见表 6-2。

6.3.4 产品

如你所料，微观过程的主要产物反映了不同的分析与设计考虑。

- “架构描述”描述了系统的架构，包括对通用机制的描述。该描述包括了分析/设计模型中对架构有重要意义的方面。
- “分析/设计模型”包括软件解决方案中的分析和设计元素、它们的组织方式以及实现。这些实现描述了系统的行为需求如何通过这些元素来实现。

对于分析/设计模型来说，选择怎样的细节层次来描述架构取决于待开发的系统以及所选择的开发过程的类型。用文档将架构记录下来之后，需要与开发团队进行沟通。毕竟，它描述了系统的架构愿景，其重要性 6.1.1 小节的“很强的架构愿景”中已经讨论过了。关于如何用文档来记录软件架构的建议，请参见补充材料“用文档记录软件架构”。

在微观过程中创建一个分析/设计模型有一些基本的好处。首先，维护一个分析/设计模型有助于建立一个共同一致的词汇表，在整个项目中使用。在开发过程中，分析/设计模型成为所有元素、它们的语义和关系的集中存放处。随着时间的推移，我们会添加新的元素，消除无关的元素，合并相似的元素，使分析/设计模型得到优化。通过这种方式，团队不断形成一种一致的语言表达方式。另外，拥有一个系统中元素的集成存放处不仅能确保这些元素的一致性，而且可以将其作为一种有效的载体，让我们能够以任意的方式查看项目中所有的元素。当开发团队的新成员必须很快了解已经在开发的解决方案时，这一点特别有用。分析/设计模型也让架构师能够全面了解一个项目，从而可能会发现一些共同之处，否则可能难以发现。你可能会猜想，使用 UML 来表示这个分析/设计模型将进一步增强这些好处。不但有“一张图相当于一千个字”的好处，而且基本上通过展示分析/设计模型来帮助揭示元素之间不一致的地方。（关于使用 UML 来表示面向对象分析与设计的更多信息，请参见第 5 章。）

选择是否维护独立的分析模型与设计模型，取决于待开发的系统以及所选择的开发过程的类型[1]。如果被开发的系统将存在数十年，有多个变体，或者为多个目标环境而设计，每个都有自己的架构，那么独立的分析模型可能会有用。在这种情况下，分析模型是作为不同设计模型（平台相关的）的一种抽象（平台无关的表示形式）被维护的。实际上，这就是“模型驱动架构（MDA）”的一项基本原则，MDA 是由“对象管理组织”支持的。维护一个独立的分析模型也可能是为了提供一个复杂系统的概念概述，但是，用良好的文档记录的架构也可以起到同样的作用。要在分析模型和设计模型之间保持高度的保真性，成本会很高。在确定是否需要一个独立的分析模型时要记住，需要额外的工作来确保分析模型和设计模型保持一致，在提供一个系统概念视图的独立模型的成本和好处之间，我们需要进行折中考虑。换一种方式，分析模型可以被看作是临时的工件，逐渐演进为设计模型（在这种情况下，分析模型被认为是“初始的”设计模型）。

[1] 关于软件开发过程的更多信息，请参见 6.1.2 小节。

用文档记录软件架构

用文档记录系统的架构对于架构本身和系统的涉众都有相当的价值。其价值不仅在于得到的文档，而且在于编写文档的过程本身。架构是设计的一部分，由许多涉众共享，而不仅仅是开发团队。部署设计者、网络设计者、应用支持和运营人员、帮助平台人员甚至项目经理都可能会阅读软件架构文档，但是，这些人即使会读，也很少会仔细地阅读软件设计说明。软件架构文档提供了极好的系统关键方面的概述，支持确认系统能够满足需求。用文档记录架构迫使你非常小心地考虑架构的不同方面。在这份补充材料中，我们对如何思考软件架构并用文档进行记录提供了一些建议。

我们前面曾讨论过，架构描述了一些关键的设计决策、规则、模式和一些约束，这些东西确定了一个框架，系统的设计和实现必须在这个框架内进行。软件架构涉及多个方面，所以架构描述也应该包括多个方面。根据 IEEE 1471 中的描述，架构师可以定义自己的视角和视图，用于沟通系统的架构。其中的视角描述了：

- 一个或多个系统模型以及这些模型的视图（投影）；
- 对这些视图感兴趣的涉众；
- 在视图中应该关注的涉众的考虑。

软件架构应该通过一组由视角确定的视图来表达，其中视角的作用是作为视图的指导。这些架构视图包括了那些从特定视角来看具有架构重要性的开发工件。

下面是一组简单的视图，可以用来描述软件架构。这组视图首先是由 Kruchten[43] 建议的，被称为 4+1 架构视图模型。

- 需求视图（也被称为用例视图）：需求视图描述了具有架构重要性的需求，既包括功能需求，也包括非功能需求。具有架构重要性的功能需求一般会驱动确定具有架构重要性的用例场景，这些用例场景将在软件生命周期的早期进行分析。具有架构重要性的非功能需求包括所有系统范围的架构品质（如可用性、弹性、性能、规模、可伸缩性、安全性、隐私性和易理解性）、经济上和技术上的约束（例如，利用已完成的产品、集成遗留系统、复用策略、要求的开发工具、团队结构和时间进度）以及法规上的约束（例如，遵守特定的标准和控制）。通常，正是这些非功能需求具有最大的架构重要性，它们驱动着架构机制的定义，这些架构机制通过逻辑视图被记录在文档中。
- 逻辑视图：逻辑视图包含了具有架构重要性的分析和设计元素、它们的关系以及它们组织成的组件、包和层，还有一些有选择的实现。它说明了这些具有架构重要性的元素如何协同工作，提供了需求视图中描述的那些具有架构重要性的用例场景。逻辑视图也描述了形成系统结构的关键机制和模式。
- 实现视图：实现视图描述了关键实现元素（可执行程序、目录）和它们的关系。这个视图很重要，因为实现的结构对当前的开发、配置管理、集成和测试具有主要的影响。

- 过程视图：过程视图描述了系统中的独立的控制线程和哪些逻辑元素参与了这些线程。
- 部署视图：部署视图描述了不同的系统节点（如计算机、路由器和虚拟机/容器），以及具有架构重要性的逻辑元素、实现元素或过程元素在这些节点上的分配。

这组视图之所以被称为“4+1”，是因为关于需求是否应该作为一种架构视图还存在着争论。在架构描述中包含需求视图是为了描述那些促使架构形成的需求，并表达架构的品质。因此，需求对于架构来说是很关键的，我们建议你找出那些具有架构重要性的需求，将它们作为架构描述的一部分，并在其他架构视图中跟踪这些需求。

Booch、Rumbaugh 和 Jacobson 指出，每种架构视图都可以独立存在，这样，不同的涉众就能够关注他们最关心的架构领域[39]。这 5 个架构视图相互之间也是有关系的（例如，部署视图中的节点包含了实现视图中的元素，实现视图中的元素又包括了逻辑和过程视图中的元素的物理实现）。

架构师应该自由地根据需求加入任意多的视图来描述软件的架构（如数据视图和用户体验视图），并除去不适用的视图。

许多其他的架构框架，不论是简单还是复杂，都具有使用视角和视图的共同特点。其中比较值得注意的有 Zachman framework [32]、Department of Defense Architecture Framework（DoDAF）[33]和 Federal Enterprise Architecture（FEA）[34]。

在某些情况下，根据项目和使用的过程，将所有的架构信息收集起来写成软件架构文档（SAD）可能是有意义的。SAD 成为了描述系统架构的主要工件，它包含了对所有具有架构重要性的工件的引用。如果有人想了解系统的架构，SAD 就是开始的地方。SAD 应该说明关键的架构考虑是如何处理的，所以最好按照刚才讨论的架构视图来组织。SAD 应该由整个团队进行复查，并随着架构的演进而更新。

6.3.5 微观过程与抽象层次

微观过程既适用于项目架构，也适用于应用工程，区别就在于所考虑的抽象层次。从架构的角度来看，微观过程提供了一个框架，使架构得到演进，并探索可选的各种设计方案；从工程的角度来看，微观过程提供了一种指导，帮助我们制定无数的战术决策，这些决定就是日常工作和架构调整的组成部分。

在微观过程活动中执行的工作的细节取决于当前的考虑（即架构或组件，请参考图 6-5）。下面针对前面定义的每一种考虑，列出了对微观过程活动重点的进一步描述。

- 在执行“架构分析”时，微观过程关注的是利用已有的参考架构或架构框架，创建一个初始的架构版本，并确定其他可以作为构建组件的资源。这包括系统的总体结构、它的关键抽象和它的机制。实际上，对于每个架构视图形成一种高层次的理解也是不错的。架构分析的结果被用于驱动架构设计。
- 在“架构设计”过程中，根据我们在架构分析中学到的东西，通过架构分析得到的

初始架构得到了改进。微观过程活动关注于改进已被识别出来的分析元素、设计元素以及它们的责任和交互。在这一个层次确定的设计元素代表了整个架构框架的构建组件，它们的关系确定了系统的整体结构。分析机制也得到改进，它利用了具体的技术成为设计机制。同时，架构上的并发和分布也得到了进一步的考虑。复用也很重要，加入已有设计元素（以及它们相关的实现）的机会和影响得到了评估。

- 在“组件分析”过程中，微观过程活动关注于确定分析元素及其责任和交互。这些分析元素代表了系统组件的第一次近似，然后被用于组件设计，以确定设计元素。重要的是要记住，微观过程分析活动的本质是提供组件的分析观点，应该避免试图在这个阶段设计组件，因为这些必需的额外考虑将在设计时进行。
- 在“组件设计”过程中，微观过程活动关注于改进组件的设计，从设计类的角度来定义组件，它们可以直接由选定的实现技术来实现。在详细设计中，继续改进设计类，找出它们的内容、行为和关系的细节。如果有了待实现的设计类的足够细节，这种改进就应该停止了。接下来就是实现，这是宏观过程的一部分。

虽然看起来微观过程似乎是一种清晰、全面、从高层抽象到低层抽象的过程，但事实却并非如此。

如前面的图 6-4 所示，你从来自宏观过程的一组需求（如用例、场景、功能点、用户故事以及附加的规格说明）开始你的微观过程[1]。然后开始执行一系列的微观过程迭代，每次迭代都将它的输入作为某种层次的抽象，并将这些输入实现为下一个层次的抽象。微观过程迭代的最终结果是最初需求的详细设计，这份设计将被返回给宏观过程，作为实现的依据。

在微观过程迭代中，在某一个时刻选择哪些元素进入较低层次的抽象取决于机会和风险。例如，在进行架构设计时，对于某个特定的范围，你可能针对你不太了解的一组元素“进一步深入”（即进行组件设计），目的是减少风险，然后再回来继续进行架构设计。下面仔细看一个例子，看看是否能说明我们的意思。

假设我们正处于宏观过程迭代的“细化”阶段，属于这次迭代范围的、具有架构重要性的需求已经准备好，要经历分析和设计过程（微观过程）。以下的场景描述了微观过程中可能发生什么。

（1）针对所有具有架构重要性的场景进行了架构分析。结果是一组具有架构重要性的分析元素。

（2）开始架构设计，利用所有的架构分析元素作为输入。在这次迭代中，发现一个设计元素没有得到很好的理解，所以“针对这个元素”进行了组件分析和设计，结果发现“在架构设计层”需要一些优化。这几次迭代的结果是一组具有架构重要性的设计元素。

（3）针对每个具有架构重要性的场景进行组件分析，利用架构设计元素作为输入。这些迭代的结果是一组设计元素，它们支持那些具有架构重要性的场景。

（4）针对组件分析得到的每个具有架构重要性的元素进行组件设计。

---

[1] 不论使用哪种类型的需求表示方式，重要的是需求确实通过文档，从用户的视角记录了系统需要做的事情，既包括功能需求，也包括非功能需求。本章中，我们使用“场景”来代指以用户为中心的需求视图。

（5）在更低的抽象层次上进行其他微观过程迭代（例如，从企业级到系统级、子系统级、组件级、子组件级等）。

这些迭代的结果是一组详细的设计元素，可以在宏观过程中实现。图 6-7 总结了宏观过程、微观过程和微观过程迭代之间的关系。

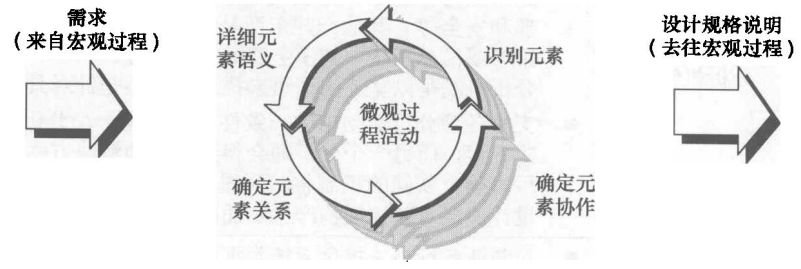


图 6-7 微观过程迭代

既然我们已经完成了对微观过程和抽象层次的讨论，就可以更详细地研究每项微观过程活动，并讨论执行了什么、得到了什么以及如何评估得到产品的品质。

6.3.6 识别元素

在对系统进行面向对象分解时，识别元素是一项关键活动。因此，这第一项微观过程活动的目的就是确定一些关键的元素，这些元素将在特定的抽象层次上描述解决方案。这里对“识别”的使用是比较宽松的。这项活动实际上是元素从一个抽象层向另一个抽象层的演化。在一个抽象层次上确定元素也包括了对前一个层次进行修改，这种修改将导致产生新的元素和不同的元素。在一个抽象层次上识别的元素将作为主要输入，用于确定下一个抽象层次上的元素。

在执行这项活动时，重要的是在确定元素和细化语义之间保持一种微妙平衡。确定元素应该只关注于确定元素，并在较高层次上对它们进行描述（简要的描述）。后续的微观过程活动将逐渐细化已确定的元素的语义。

1. 产品

这个微观过程活动的主要产品是分析/设计模型，它包含了在特定抽象层次上已确定的元素和它们的基本描述。表 6-2 总结了在不同的分析和设计活动中确定的元素。

表 6-2 在分析和设计活动中识别的元素

关注要点	确定的元素	目的和说明
架构分析	关键抽象	● 形成问题领域的词汇表。事先确定关键抽象将减少以后关键抽象概念定义冲突的可能性，即以后通过单项需求来确定元素的时候
	架构组成部分	● 表达系统中分离关注的领域，对分析元素进行分组（即组件）； ● 表达系统的高层逻辑组织。这些部分可以基于已有的架构框架。在一个分层的架构中，架构的部分就是层

续表

关注要点	确定的元素	目的和说明
架构分析	分析机制	● 表达在继续前进时所需的关键服务、基础设施和通用策略。 其中某些东西是基础性的，这意味着它们解决的是领域无关的问题，如内存管理、错误检测和处理、持久、进程间通信、事务管理和安全等。其他一些东西是领域相关的，如实时系统中的控制策略或信息系统中的事务和数据库管理。 分析机制是以宽泛的词语来描述的，不是针对具体实现的。 ● 支持各项分析活动中的一致性（即不是一个分析师会得到一个解决方案，而另一个分析师会得到不同的解决方案）。 尽早确定共同的机制将起到缓解风险的作用，即防止糟糕的战术设计决定对整体架构产生负面的影响
架构设计	具有架构重要性的设计元素	● 封装业务行为，提供系统数据的访问和管理。 ● 表达规格说明，这种规格说明能够利用特定的实现技术有效地被实现
	架构组成部分	● 细化原来在架构分析中确定的逻辑架构部分。 ● 对设计元素进行分组
	设计机制	● 将分析机制细化到特定的技术
组件分析	分析类[a]	● 表达初始的面向对象构成，这些对象形成了期望的解决方案，提供期望的行为。 ● 用内聚的职责描述独立的元素
组件设计	设计类	● 与架构设计的目标相同，只是这里我们工作在较低的抽象层次（即组件设计元素，而非架构设计元素）。这些设计元素被细化和描述到一定的详细程度，以便能够被实现

a. 某些人可能不同意在这里使用“类”这个词。但是，重要的不是它们叫什么名称，而是它们代表了什么。

2. 步骤

在第4章中，我们描述了具体的分类技术，目的是识别面向对象元素（即经典面向对象分析、行为分析、领域分析、用例分析、CRC 卡、有意义的英语描述和结构化分析）。正如在那一章中介绍的，识别面向对象元素通常包括两项活动：发现和发明。在分析过程中，识别多数是由发现来驱动的，而在设计过程中，发明起到了更大的作用。在分析过程中，设计者与领域专家合作，共同识别元素。他们必须善于发现抽象，能够研究问题域并发现有意义的分析元素。在设计过程中，架构师和设计者一起识别元素，他们必须善于从解决方案出发，创造出新的设计元素。

在设计过程中，某些在设计时识别的元素可能转化为实际的类，另一些元素可能成为某些抽象的属性或同义词。另外，在生命周期早期中确定的某些分析元素可能是错误的，但这不一定是坏事。在分析过程中，重要的是让这些决定可以在将来的开发过程中得到优化。在生命周期的早些时候遇到的实体事物和角色将一直进入到实现，因为它们对问题概念模型来说是非常基础的。随着对问题了解得更多，你可能会改变某些元素边界，重新定位责任，合

并类似的元素，并且常常会将较大的元素划分为一组互相协作的元素，从而形成解决方案中的某些机制。总之，分析元素常常是相当流动、可变的，它们可能有很大的变化，然后才在设计时被固定下来。

对于所有的抽象层来说，识别元素的方法总体上是一样的，不同的是从哪里开始（已经有了哪些抽象），关注的焦点是什么（具有架构重要性的元素或其他元素），以及要走多远（是否要深入某个设计元素，确定它由哪些元素组成）。例如，在进行架构设计时，可使用架构分析的结果作为起点，关注的是具有架构重要性的设计元素，也可能考虑这些具有架构重要性的元素由哪些元素组成，目的是确保已对每个元素的行为有了足够的了解，从而减少风险。在进行组件分析和设计时，可使用架构分析和设计的结果作为起点，识别出实现规格说明所需的其他设计元素，包括较细粒度的设计元素，它们组成了较粗粒度的元素（即提供某个组件行为的一些类）。

然后，识别元素重复递归地进行，目的是发明更多的细粒度的抽象来构造较高层的抽象，并发现已有抽象之间的共性，可利用这些共性来简化系统架构。在识别设计元素时，粒度最大的设计元素通常最先被识别出来，因为它们确定了系统的核心逻辑结构，而这些元素又由较小粒度的元素组成。但是，在实际工作中，也可以同时识别出处于不同粒度级别的设计元素，虽然这些元素之间存在着明显的顺序依赖关系（例如，只有当某个组件被识别出来以后，才能识别具体实现它的类）。

要得到细化的设计元素，以下的分析类是很好的候选者。

- 具有一组重要责任的分析类。
- 几组代表应该一起维护的信息的分析类。如果元素的信息应该一起维护，它们就应该属于同一个设计元素，涉及操作这些信息的责任也应该属于这个设计元素。
- 几组协作提供某个特定行为需求或相关行为需求的分析类（例如，参与相同或相关实现的分析类）。协作的几个元素应该属于同一个设计元素。
- 几组具有相同责任的分析类。类似的（或相关的）责任应该属于相同的设计元素。

除了查看分析元素作为设计元素的灵感之外，也可以通过应用选择的架构模式、设计模式和一般的设计原则，将分析元素细化为设计元素。一些模式的例子包括 IBM 的 eBusiness 模式[52]、架构模式[53]和设计模式[55]。一些设计原则的例子包括 Cheesman and Daniels[54] 描述的企业组件设计原则和 Herzum and Sims[56]描述的开发业务组件的最佳实践。

在识别元素时，研究处于类似抽象层次的类似系统总是聪明的做法。通过这种方式，可以从其他项目的经验中获益，这些项目也必须做出一些类似的开发决定。一般来说，在识别元素这一步，重要的是识别吸收（复用）已有元素的机会和影响，确保潜在可复用资产的适用环境与你的环境是一致的。

在架构分析过程中识别的逻辑部分通常是基于对特定架构模式的选择。随着设计元素的识别和分组，这些部分会在设计时被细化。某些部分划分指南包括将支持相同功能的元素放在一起。基于某个功能的其他功能被放在不同的部分中。在同一个抽象层中协作实现行为的功能应该被放在不同的部分中，它们代表一些对等的服务。这些决定具有战略上的重要意义。

在某些情况下，这种划分是自顶向下完成的，通过系统的全局视图，将它划分为代表主要系统服务的抽象，这些服务在逻辑上是内聚的，并且可能独立地发生变化。这种架构也可能自底向上，将语义上相近的一些类识别出来，放在一起。随着已有的设计部分逐渐增多，或者随着新的划分变得很明显，可以引入新的设计部分，或者重新组织已有的设计部分。这样的重构是敏捷过程的一项关键实践。

在架构分析过程中确定的机制被看成是共用策略和基础设施的占位符，所有的系统元素都需要它们的支持。这些分析机制是通过查看可能需要的关键服务来识别的，并以宽泛的术语加以描述。（关于如何识别机制，请参见第4章。）在架构设计过程中，你将决定这些分析机制如何设计和实现。因此，分析机制将细化为设计机制，关于它们的描述也变得更为详细。具体来说，设计机制是通过选定实现技术的具体功能来描述的。

如果元素是在不同的抽象层进行维护的（即分离的分析元素和设计元素），而不是某一个抽象层的元素平滑地过渡成为下一层的元素，那么从需求管理和变更管理的角度来说，保持不同层次抽象之间的可跟踪性是聪明的做法。建立并维护可跟踪性对于有效、准确地影响评估是很关键的。

3. 里程碑和评判标准

在特定的抽象层，针对特定的范围，如果得到了一组足够的抽象，一致地加以命名和描述，你就已经成功地完成了识别元素的微观过程活动。另一种好坏的评判就是在特定的抽象层，针对特定的范围，是否得到了足够稳定的分析/设计模型。换言之，在每次完成微观过程迭代之后，分析/设计模型没有大量的改动。例如，如果在项目生命周期的晚些时候才发现一些具有架构重要性的设计元素，则表明需求、分析或这部分设计存在缺陷。快速变化的分析设计模型是一个标识，要么表明开发团队还没有抓住重点，要么表明架构还有一些缺陷。随着开发的继续，可以通过跟踪协作的抽象之间的局部变化，来跟踪架构中较低层次的稳定性。

6.3.7 确定元素间的协作

第二种微观过程活动是确定元素间的协作，其目的是描述已被识别的元素如何协同工作，以提供系统的行为需求。在这个活动中，我们通过明智、可测量的责任分配，对已识别的元素进行细化。

1. 产品

这个微观过程活动的主要产品是一些实现，它们说明了已被识别的元素之间如何协作以提供某个范围内的行为需求。这些实现描述了一组行为需求是如何通过处于某个抽象层上的元素的彼此协作来实现的。实现反映了协作元素之间明确的责任分配，并在行为需求和软件解决方案之间架起了桥梁。实现最初是用分析元素来描述的，后来以设计元素来描述。

实现以及支持的元素责任通过分析/设计模型被记录在文档中。详细的程度和使用的表现形式取决于所面对的抽象层次。在分析过程中，可使用比较自由的方式来描述责任。通常使用一个短语或一句话就足够了，对于某项责任，如果需要更多的描述，那就过于复杂了，需要将其分成更小的部分。在设计过程中，可为每个元素创建规格说明，说明操作的名称，这

些操作构成了每个元素的协议。在详细设计过程中，这些操作以接口的方式正式地被记录下来，包含用选择的实现语言写下的方法签名。协作本身则可能利用简单的图示被记录在文档中，展示了元素之间的协作。UML 交互图（具体来说，包括序列图和通信图）是表现这种协作非常有效的方式。另外，某些元素的状态决定了它们与其他元素的协作。对于这些元素来说，可以利用状态机图来记录关键的状态转换。UML 状态机图对于表示这些状态机是非常有效的。关于说明元素语义的更多信息，请参见 6.3.9 小节。关于使用 UML 的更多信息，请参见第 5 章。

2. 步骤

对于将工作分配给已被识别的元素来说，分析行为需求是一项杰出的技术。下面的几个步骤描述了一种方法，确定了处于某个抽象层的一组元素的语义。

（1）分析行为，将责任分配给参与提供该行为的元素（即在前一个微观过程步骤中识别出来的那些元素）。请考虑异常行为和预期行为。如果某些元素的生命周期很重要或很关键，那就为它开发一个状态机。这一步的结果是得到行为的实现，说明参与的元素和它们的协作。

（2）在实现中提取模式，将这些模式用更抽象、更一般的实现方式来表达。

这种方法对于各个层次的抽象都同样有效，不论是在分析用例/场景所表达的系统行为（分析），或是在分析责任/接口所表达的组件行为（设计），还是在分析用文本方式表达的架构机制的行为（分析和设计）。

接下来，让我们更仔细地看看这些步骤。

行为分析 行为分析可以用来理解一组行为需求是如何通过解决方案中的元素来提供的。场景分析的主要产品是一组实现。实现可以通过用例分析（特别推荐）、行为分析或 CRC 卡等技术来得到，像第 4 章中介绍的那样。

在分析一个场景时，典型的事件序列可以总结如下。

（1）从行为需求中选择一个场景或一组场景作为考虑的对象。

（2）确定哪些元素与该场景有关。（元素本身可能已经在前面的微观过程活动中被识别出来了。）

（3）推演这个场景，将责任分配到每个元素，以完成期望的行为。根据需要，分配一些属性来表示结构化的元素，这些属性是实现特定责任所需要的。注意，在这一步中，重要的是关注行为，而不是结构。属性代表了有结构的元素，所以这是有危险的，特别是在分析的早期阶段。危险就是由于需要特定的属性，过早地绑定实现决定。只有当属性对于构建场景的概念模型很重要时，才应该在这里确定属性。

（4）随着场景分析的进行，对责任进行重新分配，以得到比较平衡的行为分配。只要有可能，就复用或适配已有的责任。将较大的责任分解成一些较小的责任是常见的动作，将较小的责任组合成较大的行为也是可能的，但出现的频率会低一点。对单个场景的分析可能导致不相似的责任被分配到相同的元素中。需要将这样的元素分解成多个元素，每个都包含一组一致、内聚的责任。

（5）在设计过程中，必须考虑这些实现的并发和分布。如果存在并发的可能，必须指明

参与者、代理和服务者以及它们之间的同步方式。在这个过程中，可能会发现需要在对象之间引入新的途径，以消除或合并未用到的、冗余的对象。

在分析一个场景时，可能会发现一个或多个元素的状态对于场景的发展具有重要的影响。在这种情况下，就需要花些时间仔细研究一下该元素可能经历的外部可见的状态改变，以确保场景的发展可以包容这些状态变化。使用状态机图是记录元素的关键状态和状态转换的一种准确的方式。

提取模式 这一步意识到了共性的重要性。当确定元素的语义时，必须对行为的模式保持敏感，这代表了复用的机会。

在提取模式时，典型的事件序列可能如下。

（1）针对在这个抽象层次的全部实现，寻找参与元素之间的交互模式。这样的协作可能代表了隐式的惯用法或机制，应该进行检查，以确保没有无必要的差异。重要的协作模式应该作为策略性的决定，明确地通过文档被记录下来，以便可以复用它们，而不需要重新发明。这项活动保持了架构愿景的完整性。

（2）针对这个抽象层生成的所有责任，寻找行为的模式。共同的角色和责任应该统一为共同元素的共同责任。

（3）如果工作在较低的抽象层次，随着具体的操作的确定，寻找操作签名中的模式。消除没有必要的差异，如果发现这样的操作签名重复出现，就引入共用的类。

如果发现了协作的模式，就用更抽象、更一般的实现来表示它们。

3. 里程碑和评判标准

如果你得到了一组一致的元素和责任，它们在某个抽象层次上，在一定的范围内提供了系统要求的功能行为，并且做到了在这些元素之间进行有意义的、平衡的责任分配，那么就成功地完成了确定元素协作这项微观过程活动。

作为这项活动的结果，你应该开发并验证了一些实现，这些实现代表了被考虑的范围的基本行为。所谓“基本行为”，指的是与应用的目标核心相关的行为。实现好坏的判别标准是完整性和简单性。每个实现都必须准确地反映了参与实现的单元元素的语义。一组好的实现应该包含所有主要的场景和一定比例的、感兴趣的次要场景。我们不指望也不追求对所有场景的实现，只考虑主要场景和一些次要场景就足够了。另外，一组好的实现也会发现行为模式，最后得到的解决方案结构体现了不同场景之间的共性。

对于单个的元素责任来说，要记住这项活动关注的是协作和确定“谁做什么”。在这个时候，记录下元素的责任就足够了。在较高的抽象层次上，可对责任进行有意义的说明。在较低的抽象层次上，可使用更精确的说明语义，但不要陷得太深，因为明确定义单个元素的语义是第 4 项微观过程活动的目的，这将在 6.3.9 小节中介绍。

下面列出了一些简单而有用的检查点，可以用于评估这项活动的结果。

- 元素应该具有平衡的责任。一个元素不应该“什么都做”。
- 元素应该具有一致的责任。如果元素的一些责任是无关联的，那么它应该被分成两个或多个元素。

- 应该没有两个元素具有相同或非常类似的责任。
- 为每个元素定义的责任应该支持该元素所参与的场景。
- 责任不简单或不清楚意味着给定的抽象没有很好地被定义。

至此，我们已经识别了元素，并确定了这些元素如何通过协作来提供需要的行为。接下来让我们把注意力放到元素之间的关系上，这些关系支持了它们的协作。

6.3.8 确定元素间的关系

第 3 项微观过程活动的目的是确定元素之间的关系，这些关系支持了前一项微观过程活动中定义的元素协作。确定元素的关系就形成了解决方案。具体来说，在架构层次的抽象上，关键元素和关键部分之间的关系确定了系统的总体结构，并为所有其他系统元素之间的关系奠定了基础。确定关系的目的是确定每个元素的边界，清楚地表达哪些元素之间互相协作。这项活动正式确定了元素之间的分离关注，这种分离关注最初是在确定元素协作时建立起来的。

1. 产品

这个微观过程活动的主要产品是在当前抽象层的元素之间的关系。确定好的关系被添加到演进中的分析/设计模型中。

尽管这些关系最终会以具体的形式表示(即通过编程语言表示)，我们仍建议先利用 UML 图或其他的图示方式可视化地对其进行表示。可视化的图示提供了更全面的架构视图，从而可以不受编程语言的限制来表达这些关系。这些图示帮助你想象和推理这些关系，这些关系所涉及的实体可能在概念上或物理上距离遥远。绘制这些图示有一个结果，即可能会发现以前未发现的交互模式（这可能是你希望使用的），也可以发现继承关系中设计得不合理的地方。

我们不追求，实际也不可能，得到一套面面俱到的图示，能表示出元素之间所有可以观察到的关系视图。相反，我们建议你关注感兴趣的那些视图。所谓的“感兴趣”，是指一组相关的元素，它们的关系表达了某种基本的架构决定或完成实现蓝图所需的某个细节。有一组图示可能应该考虑，它们与在前一个微观过程活动（确定元素协作）中得到的实现有关。这些图包含了参与实现的元素以及它们的关系，表达了实现的结构特点。

2. 步骤

一般来说，确定元素关系有两个步骤：

（1）识别关联，初步识别元素之间的语义联系；

（2）细化关联，成为语义上更丰富的关系（即聚合、依赖等）。

识别关联主要是分析和早期设计时的活动。在架构分析时，确定了高层架构部分之间的关系和关键抽象之间的关系。在架构设计时，通过这项活动来确定关键组件之间的关系，并将高层设计元素划分成设计部分。在组件分析时，通过这项活动来确定分析元素之间的关系(包括关联和某些重要的继承或聚合关系)。

在确定元素关联时，典型的事件序列可能如下。

（1）收集一组元素，它们处于某个抽象层，或者与特定的场景/实现有关系。

（2）考虑任意两个元素之间是否存在语义上的关系，如果存在，就建立一个关联。如果需要从一个元素导航到另一个元素，或者需要从一个元素调用某种行为，则都需要引入关联。如果两个元素必须彼此协作，它们之间就应该有关系。

（3）针对每个关联，如果角色没有和元素名称重复，就指定每个执行者的角色，并指定相关的多重性或其他类型的约束。只有当这些细节很明显时才包含它们，因为细化这些关系是下一步的工作。

（4）通过走查场景来验证你的决定，确保得到的关联对于提供参与场景的元素之间的导航和行为是必要和足够的。

第3章中曾解释，关联是语义上最弱的关系：它们仅表示某种一般的依赖关系。但是在分析和早期设计中，这通常就足够了，因为它记录了足够多的两个抽象之间关系的细节，同时又让我们避免做出不成熟的详细设计决策。

关联的细化既是分析活动，也是设计活动。在分析时，可能需要将某些关联演进为另一些关联，这些关联在语义上更精确、更具体，反映了对问题域不断加深的理解。在设计时，同样要转换一些关联，同时添加新的、具体的关系，以提供实现的蓝图。聚合、组合、依赖等关系是我们主要感兴趣的关系，包括它们的附加属性，如名称、角色、多重性等。

在细化元素关联时，典型的事件序列可能如下。

（1）寻找一组元素，它们已经被一组关联联系在一起（例如，参与某个实现的一组元素），并考虑每一个关系的语义，根据需要细化关系的类型。这种关系代表了另一个对象的一次简单使用吗？如果是这样，关联就应该被细化为一个依赖关系。这种关系代表了对应元素之间的一种结构关系吗？如果是这样，关联就应该被细化为一个聚合或组合关系。应该逐一检查每个关系，从而确定并记录下这些关系的实质。

（2）寻找元素之间的结构模式。如果发现了这种模式，就考虑创建新的元素来记录这种共用的结构，并通过继承（将这些类放到已有的继续结构中，或者创建新的继承结构）或聚合来引入这些新元素。

（3）寻找元素之间的行为模式。如果发现了这种模式，就考虑引入共用的参数化元素，执行共同的行为。

（4）考虑已有关联的导航性，如果可能，对导航性加以约束。如果不希望使用双向导航，就使用单向导航。

（5）随着开发的进行，引入一些细节，如关于角色、多重性等的说明。不用说明所有的细节，只要包含重要的分析或设计信息或者实现所必需的信息。

3. 里程碑和评判标准

如果你在某个抽象层次指定上了元素之间的关系，那么就成功地完成了确定元素关系这项微观过程活动。

在这个阶段要注意的一件事就是参与实现的元素之间关系的一致性。具体来说，对于每个实现，参与元素之间的关系以及元素之间要求的协作必须是一致的（如果存在协作，就必

须存在关系）。

好坏的评判标准包括内聚、耦合和完整性。当复查在这个活动中确定的关系时，所追求的是拥有在逻辑上高内聚和低耦合的元素。另外，希望确定给定抽象层次上所有重要的关系，这样在下一层抽象中就不需要引入新的重要关系，也不需要通过一些不自然的操作来使用已确定的那些关系。如果发现元素和关系在确定时不方便，那就表明还没有在元素之间设计出一组有意义的关系。

6.3.9 详细确定元素的语义

在此之前，我们主要关注的是元素如何合作。下面我们将自底向上仔细看看单元素的语义，以确保它们是一致的，并得到了很好的理解。

第 4 项微观过程活动的目的是理清某个抽象层面上已确定的元素的行为和属性，以确定在该元素参与的所有场景中，语义都是一致的，同时确保已为每个元素提供了足够的信息，让该元素能进入下一个抽象层。在这个活动中，元素的语义“在当前的抽象层次上”得到细化，加入了足够的细节，确保可以在下一个抽象层次上进行元素识别。例如在分析时，细化元素语义的目标是细化分析元素的语义，包含足够的信息，以便能够识别设计元素。在设计时，细化元素语义的目标是细化设计元素的语义，包含足够的细节，以便能够实现。

将这个活动作为微观过程的最后一个活动是有意为之的：微观过程首先关注的是元素之间的行为和协作，而将确定单个元素的详细语义尽可能地推到最后。这种策略避免了不成熟的决定，而不成熟的决定有可能让我们丧失得到更小、更简单架构的机会。同时，这种策略也让我们能够根据效率的需要自由地改变内部的表现形式，而且不会破坏原有的架构。微观过程的前 3 项活动关注的是元素的外界视图以及它们之间的协作，最后一项活动则关注单个的元素，清楚地确定每个元素的外部视图，并提供额外的细节，驱动内部视图的开发。

1. 产品

这个微观过程活动的主要产品是细化的分析/设计模型，其中包含了元素的详细语义。根据所面对的抽象层次，细节水平和表示水平都用于记录元素的语义。

在分析时，这项活动的结果是相对比较抽象的。你不太关心做出表示决定，相反，更关心发现新的抽象，向它们分配责任。在分析层详细确定语义可能包括：利用活动图的方式更详细地描述这些责任和总体流程。某些元素的责任受到事件的驱动，或者与状态顺序有关。对于这些元素，可利用状态机来描述每个元素协议的动态语义[1]。

在设计时，特别是在详细设计的后期，必须对表示做出具体的决定。随着开始细化单个元素的协议，你可能对具体的操作命名，但忽略它的完整操作签名。只要可行，就会提供每个操作的完整签名。在设计时，你可能还会指定应该使用某些算法。如果工作在较低的抽象层，随着开始与给定的实现语言进行绑定（如在详细设计时），详细的语义甚至可以包含伪代码或可执行的代码。当得到了正式的类接口之后，就可以开始利用编程工具来测试和确保实

[1] 协议规定了特定的操作需要以特定的顺序被调用。除了最简单的类之外，操作很少是独立的，每个操作都有一些先决条件要满足，这通常是通过调用其他操作来实现的。

现这些设计决策。在这一步将得到更正式的产品，其主要好处就是可以迫使开发者思考每个抽象协议的实际可行性。不能确定清楚的语义则标志着抽象本身有缺陷。

2. 步骤

详细的元素语义包括对结构和算法的选择，它们描述了元素的结构和行为。在细化元素语义时，典型的事件序列可能如下。

（1）列出该元素的角色和责任。从前面确定元素协作（第2项微观过程活动）得到的单个实现中，收集并确定这些结果。利用这些实现帮助确定参与元素的责任。通过在所有实现中查看所有调用该元素的协作，来确定该元素的责任。（元素的责任就是其他元素可以要求它做的所有事情。）

（2）更详细地描述每项责任。绘制描述整体流程的活动图或序列图，绘制状态机图来描述状态行为，等等。如果可能，为每项责任/操作推荐一种合适的算法。在设计时，考虑引入一些辅助的操作，将复杂的算法分解为不太复杂、可复用的部分。要折中考虑存储元素的状态还是计算元素的状态。

（3）在设计时，要考虑继承。选择合适的抽象类（或者创造新的抽象类，如果问题足够一般化的话），并根据需要调整继承关系。考虑打算赋予责任的那些元素。在理想的情况下，这可能只需要稍稍调整较低层元素的责任和协议。如果元素的语义不能够通过继承、实例化或委托来提供，请考虑在下一个抽象层次中提供一种合适的表示形式（也就是说，如果正处于设计层，这可能包括实现语言提供的原语）。要记住从该元素客户的角度来看操作，并为预期的使用方式选择最佳的表示形式，这一点很重要。但要注意，不可能针对所有使用方式都是最佳的。随着从后续的版本中得到更多的经验信息，就可以确定哪些元素的时间/空间效率不高，并对它们的实现进行局部的调整，而不必担心破坏客户程序对抽象所做的假定。

（4）随着每个元素的责任得到定义，考虑该元素为了实现其责任而必须具备的属性。

（5）在设计时，设计一组足够的操作，来实现这些责任。如果可能，尝试复用概念上类似的角色和责任。作为单个类来说，责任就是由该类的操作记录的；作为组件来说，责任是由该组件提供的服务来表现的，通过该组件接口的操作来记录。

- 逐一考虑每个操作，确保它是简单的。如果不是，就将它分解成更简单的操作，并提供出来。复合操作可以被保留在该元素中（如果操作足够通用，或者理由很充分），或者被移到一个公共的类中（特别是当这个操作有可能经常改变时）。分解操作可能让你找到更多的共性。
- 考虑构造、拷贝和析构的需求[13]。最好是对这些行为有通用的策略，而不是让每个类使用自己的习惯方式，除非这样做有很好的理由。
- 考虑完整性的需要。添加其他的一些简单操作，虽然它们不是目前的客户程序所需要的，但它们的存在使组件变得完整，所以可能会被将来的客户程序使用。由于我们认识到不可能有完美的完整性，所以更倾向于简单性，而不是完整性。

在细化元素的责任时，可能发现一些新的元素支持详细的责任描述。（例如，当更完整地描述一个元素的责任时，可能发现以前漏掉了一部分关键的信息，所以可能找到一些新的

元素来表示这部分信息。）用文档记录下这些元素和它们的责任，针对这些元素重复刚才描述的步骤。

当在设计过程中定义单个元素的语义时，元素之间的共性可能很明显，我们可能很想开始为这些元素定义非常精细的继承关系，以便反映共同的行为和共同的结构。但是，不要过早关注继承关系是很重要的一点：引入不成熟的继承关系常常导致类型完整性的丧失。继承的使用通常被认为是一项设计活动，因为这时对设计元素的语义有了更详细的了解，因此才能够更好地将它们放在继承层次之中。在设计时，遇到的类之间的共性可以通过泛化/特化层次结构来表达。当定义这种继承层次时，要注意平衡（继承层次不要太多也不要太少，不要太宽也不要太窄）。当结构模式或行为模式出现在这些类中时，重新组织继承层次，使共性最大化（但不以牺牲简单性为代价）。关于创建继承层次的更多考虑，请参考第 3 章。

在开发的早期阶段，在使用继承之前，对单个元素的语义文档是分开来的。但是，一旦有了继承层次，对元素语义的文档记录就必须显示操作在层次结构中的位置。在考虑与某个元素关联的操作时，重要的是决定把这个操作放在继承层次的哪一层上。某些操作可能被一组对等的类使用，那么就应该重构出一个共同的超类，可能是引入一个新的中间类。

在细化元素语义时，要确保“停留在当前的抽象层次上”。识别下一个抽象层次中的元素是在微观过程的下一次迭代的第一项活动中（或在宏观过程的实现阶段）完成的。

3. 里程碑和评判标准

如果对处于特定抽象层次的元素的语义有了更完整的理解（即提供了足够的细节，以便能够进入下一层次的抽象），并且为这些元素指定的语义在这个抽象层次上是一致的，那么就成功地完成了细化元素语义这项微观过程活动。

作为微观过程中的最后一项活动，其最终的目标是得到一组清晰的抽象，它们具有高内聚、低耦合的特点。

评估这项活动是否成功，要看单个元素的语义。作为这项活动的结果，应该对特定抽象层上的每个元素，得到一组相当充分、简单、完整的语义。应该为每个元素提供足够的细节，以便能够为下一个抽象层次识别元素。例如，在分析时，如果你得到了分析元素的责任和属性的描述，了解的东西足以让你进入设计，那么就成功地完成了这项活动。在设计时，如果你得到了更准确的语义（即操作和属性），足够进入实现和测试（即它们的结构和使用可以由选定的实现语言来实现），那么就成功地完成了这项活动。这不是说这些元素必须以生动的细节来表述，而只是说需要足够的信息，让有能力的实现者能够完成他们的工作。

这项活动结果好坏的主要评判标准是简单性。如果元素的语义是复杂、别扭或效率不高的，就表明该元素还缺少某些东西，或者选择了一种糟糕的表现方式。

这里我们完成了对微观过程活动的讨论，也完成了本章关于软件开发过程的讨论。我们希望你现在已经赞同把总体软件开发生命周期（宏观过程）和分析设计活动（微观过程）分开来考虑，并已经理解了这些过程如何相互支持。软件开发项目要想成功，很关键的一点是在宏观过程和微观过程的层面上，选择一个具体的开发过程，对它进行配置，使它满足项目的具体需要。

6.4 小结

- 成功的项目通常具有两个特点，即具有很强的架构愿景和管理良好的迭代、增量式开发生命周期。
- 架构描述了一些重要的决定，这些决定是针对结构和行为做出的，并且通常反映了一种架构风格。很强的架构愿景使系统的构造变得更简单，系统更健壮、更有弹性，并且能够有效地复用，同时更容易维护。
- 如果系统的功能是以一系列连续的发布版本（内部或外部）的方式提交的，完成度不断增加，这就是迭代和增量式开发，每个发布版本就是一次迭代。选择每个发布版本开发哪些功能是由缓和项目风险来决定的，最关键的风险最先考虑。迭代和增量的方式是绝大多数现代软件开发方法的核心，其中也包括敏捷方法，因为它是一种非常有效的风险和变更管理技术。
- 所有软件开发过程都处在一个过程连续光谱中的某个位置，敏捷方法和计划驱动方法分别处于这个连续光谱的两端。为某个项目选择正确的软件开发过程是由项目的特征（和组织的特征）决定的，并且涉及对开发过程的配置，以反映敏捷和计划过程之间的平衡，而这种平衡符合项目在连续谱中的位置。
- 本章中，是从两个方面来描述软件开发过程框架的——总体软件开发生命周期（宏观过程）以及分析和设计过程（微观过程）。对生命周期风格的选择（如瀑布式、迭代式、敏捷式、计划驱动式等）会影响宏观过程，对分析和设计技术的选择（如结构化、面向对象等）会影响微观过程。不论选择敏捷过程还是计划驱动过程，微观过程这一节中描述的面向对象分析和设计技术都可以非常适用。
- 微观过程的目的是以宏观过程提供的需求作为输入（也可能是前一次微观过程迭代得到的分析和设计规格说明），得到分析和设计规格说明，再反馈给宏观过程。最后，微观过程将得到规格说明，让宏观过程中的实现阶段去构建、测试和部署。
- 微观过程由 4 项关键活动组成（识别元素、确定元素协作、确定元素关系、细化元素语义）。微观过程的每次迭代都会针对某个抽象层上的一组行为需求，经历这些活动组成的迭代。基本的步骤和得到的产品对于所有的抽象层来说都是一样的，区别之处就在于细节的层次（更低的抽象层次将得到更详细的产品）。

实战

今天的软件开发是几十亿美元的、竞争激烈的、世界范围的业务，其范围从北美延伸至西欧和东欧，进入亚洲和太平洋区域。尽管有了支持面向对象开发主要功能的工具——需求管理、配置管理、设计、编码和测试，但依然有太多失败，如进度滞后、成本超支、功能没有被提供。单个开发项目就能损失成百上千万美元。一个不幸的例子是 FBI 的 Virtual Case File 系统。本来认为它是和恐怖主义战斗的重要工具，但经过 3 年的开发之后，2005 年 4 月，“当局放弃了这个 1.7 亿美元的项目，包括价值 1.05 亿美元的不可用代码”[1]。2006 年 3 月 16 日，FBI 又签了一个 3.05 亿美元的合同，开发 Sentinel 系统来代替 Virtual Case File 系统[2]。这只是太多失败的软件开发中的一个例子[3]。

事情的复杂性在于：设计软件不只是一个科学问题。考虑使用实体－关系建模来设计复杂数据库，这是面向对象设计的一个基础。正如 Hawryszkiewycz 所说，“虽然这听起来相当直白，但它确实涉及了一定的关于企业中不同对象的重要个人见解。结果是设计流程不是决定论的，不同设计人员可以产生同一企业的不同企业模型”[4]。

可以推论，不管多么精密的开发方法，不管有多么理由充足的理论基础，也不能忽略为真实世界设计系统的实践方面。

这意味着，我们必须考虑可靠的管理实践，如人员管理、发布管理和质量保证。对技术主义者来说，这些主题极度乏味；对于职业软件工程师来说，这些是成功建造复杂软件系统必须面对的现实。因此，本章将聚焦于面向对象开发实战[1]，并检验对象模型对不同管理实践的影响。

7.1 管理和计划

在一个迭代和增量生命周期的面前，拥有强有力的、积极主动管理和指导项目活动的项

[1] 韦氏新世界大学词典（*Webster's New World College Dictionary*）定义实战（pragmatic）为“关心确切的实践，而不是理论或投机；实践的”。

目领导是至关重要的。有太多的项目因为缺乏关注而走入歧途，而强有力的管理团队可以有效地缓解这个问题。

7.1.1 风险管理

归根结底，软件开发经理的职责就是管理非技术风险，而管理技术风险通常是项目架构师的职责。面向对象系统中的技术风险包括继承结构的选择（它在可用性和灵活性之间提供很好的协调），还包括机制的选择（它在简化系统架构时产生可接受的性能）。面向对象系统面临的非技术风险包括监督第三方厂商是否及时交付软件，管理顾客和开发团队之间的关系，以便在分析期间发现系统的真正需求。

正像在前一章提到的，面向对象开发的微观过程具有固有的不稳定性，需要采取积极的计划来强制结束。幸运的是，通过提供一系列有形产品（管理人员可以研究这些产品来确定项目是否健康，是否允许管理人员按需要重新配置团队资源），我们能够设计面向对象开发的宏观过程，使其可以走向结束。宏观过程的演化式开发方法意味着有机会在生命周期的早期识别出项目存在的问题，并在这些风险对项目的成功造成危害之前排除它们。

软件开发管理的许多基本实践，如任务计划和预排，都不受面向对象技术的影响。管理面向对象项目的不同之处在于，面向对象系统的任务调度和产品评审与非面向对象系统有一些差别。

7.1.2 任务计划

在任何大中型项目中，召开每周一次的团队会议来讨论已完成的和即将进行的工作是合适的。小频率的会议对促进团队成员之间的交流来说很有必要，但是过多的会议就会损害生产力，事实上这也是一个信号，表明项目失去了方向。面向对象软件开发需要每个开发人员都有大量日程外的时间可供支配，这样他们就可以思考、创新、开发，并可以在需要的时候随时与其他团队成员一起讨论细节性的技术问题。管理团队必须安排出这部分比较自由的时间。

团队会议为微调微观过程中的进度表和洞察隐现的风险提供了一个简单而有效的手段。这些会议可能会引起工作安排小的调整，以确保项目稳定进行：没有哪个项目可以承受它的任何一个开发人员空闲下来，而等待团队其他成员去稳定他们负责的那部分架构。面向对象系统尤其是这样，类和机制的设计遍及整个架构，如果某些关键类有变化就会导致开发的停滞。

就更大的范围来说，任务计划还包括调度宏观过程可提交的产品。在演化的各个发布版本之间，管理团队必须评估项目即将面临的和长远的风险，在必要时集中开发资源处理这些风险[1]，随后还要管理微观过程的下一个迭代，以便得到满足该发布版本所需用例场景要求的一个稳定系统。这个级别的任务计划常常因为制定时过于乐观而不能很好完成[5]。开发一开

[1] Gilb 说："如果你没有主动去攻击风险，风险就会主动来攻击你。"[6]

始被看作“一件简单的编程工作”，后来却延长到几个星期或几个月。当工作在系统某一部分的开发人员采用某些来自于系统其他部分的协议时，会遭受移交物不完整或不正确的危险，这时进度安排就被抛到了窗外。更危险的是，进度可能会因为出现不得不解决的性能问题，或者因为战术设计决策出现失误而受到致命的损害。

不受过于乐观的计划支配的关键是，开发团队所遵循的校准标准和他们的工具。通常，任务计划按如下方式进行。首先，管理团队将一个开发人员的工作重心引导到系统中一个特定部分，例如，设计一系列与关系数据库接口的类。开发人员考虑所做工作的范围，返回估计完成工作的时间，项目管理工作要依据这些因素来安排其他开发人员的活动。问题是这些估计并不都是可信的，因为它们通常代表着最好状况下的情况。一个开发人员可能认为某项工作一周内能做完，而另外一个开发人员则认为同样的工作需要一个月。在实际实施这项工作时，有可能要花费这两个人三个星期的时间。因为第一个人低估了工作（这是很多开发人员都会犯的错误），而第二个人高估了现实中可能会遇到的问题（这通常是因为他知道计划中预测的时间有别于实际工作中需要的时间，所以在其中加进大量不实用的活动）。为了使项目组对制定的计划有信心，管理团队必须对每一个开发人员可能遇到的附加因素作出计划。这并不是暗示管理人员不要相信他的开发人员，而是因为开发人员常常聚焦于技术问题，而不是计划问题。管理人员应该帮助他的开发人员学会制定有效的计划，通常这是只能通过实战经验才能获得的技巧。足够的训练和估算指南对于减少低效计划来说是必需的。

很明显，面向对象开发过程有助于建立这些标准的校准因素。迭代和增量的生命周期意味着项目的早期会建立很多个中间的里程碑，管理人员可以使用这些里程碑来收集开发人员用来制定和执行日程安排的跟踪记录的数据。随着演化的开发过程的进行，这意味着经过一段时间，管理人员会更好认识到开发团队中每个开发人员的真实生产率，而且开发人员也获得更加准确地评估自己工作的经验。同样的道理也适用于开发工具的选择。我们强调要尽早地交付架构方面的发布版本，面向对象开发过程也鼓励尽早地使用工具，这有利于及早发现这些开发工具的局限性，而不会在开发方式已经定型之后又要换用其他的开发工具。

7.1.3 开发评审

开发评审是每一个开发团队应该牢固树立的另一个实践。由于有了任务计划，软件开发评审的实施受面向对象技术的影响不大。然而，相对于非面向对象系统，要评审的东西就有分别了。

管理人员必须采取措施在过多和过少的开发评审之间取得平衡。在所有系统中（特别关键的系统除外），评审设计的每一个方面是不经济的。因此，管理人员应该利用团队珍贵的资源来评审系统中代表战略开发问题的那些方面。对于面向对象系统，建议对用例场景以及系统的架构做正式的评审，而对较小的战术问题进行更多非正式的评审。所选择的场景应该包括主要场景和有重要系统回应的备选场景。

正如在上一章所提到的，用例场景是面向对象开发分析阶段的主要产物，以系统提供的功能的形式，从用户的视角捕获系统的期望行为。用例场景的正式评审由团队的分析员（精

通用例开发)、领域专家及其他最终用户共同进行，其他开发人员在场，包括QA员工（测试员)。这种评审最好贯穿整个分析阶段，而不要等到分析阶段结束时才进行一个大规模的评审，到那时再发现问题就来不及做什么有意义的工作来重新分析了。经验表明，非开发人员也可以理解这些通过文本或活动图、序列图等图形展现的用例场景[1]。最后，这种评审还可以为系统的开发人员及用户建立一个通用的词汇表，让开发团队的其他成员在开发过程的早期就了解这些评审，这样可以使得他们直接面对系统的真实需求。

架构评审应该聚焦于系统的整体结构，包括它的类结构和机制。像用例场景的评审一样，架构评审应该贯穿项目开发的全过程，由项目的架构师和其他设计人员来领导。早期的评审注重于解决系统架构的问题，以后的评审可以聚焦于某个组件或具体的普遍机制。评审的中心目的是在生命周期早期确认设计的有效性。这样做也有助于沟通架构的愿景。评审的另外一个目的就是增加架构的可见度，以便创造机会，发现类的模式或对象的协作，用于以后简化架构。

非正式的评审应该根据开发的进展定期进行，通常包括对某个组件或低层机制的同级评审。这些评审的目的是验证这些战术决策，以及为高级开发人员指导初级开发人员提供一个工具。

7.2 人员配备

面向对象开发的人员配备和传统软件开发类似，区别是在开发的生命周期内这些资源的时间分布。例如，出于迭代和增量的本质，架构师和设计人员在过程中扮演着重要的角色。

7.2.1 资源配置

管理面向对象项目一个更值得高兴的方面是，在稳定状态下，通常会降低对资源的总需求量，而且相对于更传统的方法如单个瀑布，随时间的推移对资源的需求会有一些变动。在这里，有经验的团队的操作阶段会处于“稳定状态”。一般说来，组织承担的第一个面向对象项目将比非面向对象方法需要略多一些资源，主要是因为采纳任何新技术的过程都必然经过一条学习曲线。对象模型的基本资源优势并不会马上就体现出来，而是要等到第二甚至第三个项目，那时，开发团队更加熟练掌握面向对象分析设计——从架构到类设计，获取各种通用的抽象和机制，而管理团队也能更加自如地驾驭迭代和增量的开发过程。

就分析来说，资源需求在实施面向对象的方法时通常没有什么重大的改变。然而，因为面向对象过程把重点放在架构设计上，所以我们希望系统的架构师和其他设计人员在开发过程的早期加快工作，甚至有时候在分析的后期就催促他们开始架构方面的探索。在后续的增量中，需要的资源通常更少，主要的原因在于，正在进行的工作可以使早期架构设计或过去

[1] 我们在评审中曾遇到过用不同表示法的情况，如像天文学家、生物学家、气象学家、物理学家和银行家这样的非开发人员采用的表示法。

的增量中创建的通用抽象和机制发挥巨大的作用。测试也需要更少的资源，首要的原因是，在一个类或机制中增加新的功能应该主要通过修改一个结构来实现，而这样的结构在首次出现时我们知道它的行为是正确的。因此，测试应该在生命周期的早期进行，并且证明它自身是一个累计而非单独的活动。与传统的方式比较，集成通常需要更少的资源，主要是因为集成在整个开发生命周期中增量地发生，而不是出现在一次大的突发性事件中。因此，对于有经验的团队来说，在稳定的状态下，面向对象开发对人力资源的净需求要少于传统方式下的需求。此外，当我们考虑到面向对象软件的总体拥有成本时，总的生命周期成本通常会较小，因为最终产品会有更好的质量，同时也具有更好的修改余地。

7.2.2 开发团队角色

记住，软件开发归根结底是一项人为活动，这是很重要的。开发人员不是可以互换的部件，任何一个复杂系统的成功部署都需要一个专注的团队，里面的各位成员具有独特和不同的技能。

经验表明，与传统方法相比，面向对象开发过程需要对不同的技能有一个很细致的划分。我们发现在，一个面向对象项目中主要有以下三类角色：

- 项目架构师；
- 组件主管；
- 应用工程师。

项目架构师是梦想家，负责演化和维护系统架构。对于中小型系统来说，架构设计通常由一两个具有独特洞察力的个人负责。但对于大型项目来说，架构设计则通常由一个更大的团队负责。项目架构师并不一定是最高级的开发人员，但却应该是最有资格去制定战略决策的人，通常，选择他们是由于其具有丰富的建造类似系统的经验。正是因为这种特殊的经验，架构师凭直觉就能够判断出与给定领域有关的通用架构模式，以及什么样的性能问题可能出现在特定的架构变体中。除了分析和设计经验之外，架构师应该具有编程经验，并精通面向对象开发的表示法、过程和工具，因为他们必须最终根据类的聚集和对象的协作来展示架构愿景。

雇佣一个与本领域无关的架构师通常不会带来好的效果。打个比方，这就像是他骑上一匹白马闯进来，宣布某些架构愿景，然后挥鞭而去，让其他人来承受他作出的这些决策所带来的一系列后果。好得多的做法是：让一个架构师积极参与分析过程，并随后让这个架构师继续参与大部分的系统演化过程。这样一来，架构师会更加熟悉系统的实际需要，并且随着时间的推移，架构师将接受他的架构决策的结果。另外，通过由个人或小型团队负责架构的完整性，开发更有弹性架构的机会也增加了。

组件主管的工作是对项目进行最初的抽象。组件主管主要负责设计一个完整的组件。和项目架构师一起，每个组件主管必须设计、保护和商讨一个特定组件接口，然后指导它的实现。因此，组件主管是一个类聚集和与之相关的机制的最终拥有者，同时也负责在系统演化过程中对系统进行测试和发布。

组件主管必须精通面向对象开发的表示法和过程。通常，他们可能是比项目架构师更好的设计员和程序员，但是缺乏架构师所具有的丰厚经验。平均来说，组件主管约占整个开发团队的1/3～1/2。

应用工程师在项目中是次一级的开发人员，通常只完成一项或两项职责。某些应用工程师负责在组件主管的监督下实现一个组件。这个活动可以包括某些类设计，但通常还包括实现类并随后对团队内其他设计人员所设计的类和机制进行单元测试。然后，其他应用工程师负责使用这些由架构师和组件主管所设计的类，并组装这些类来完成系统的用例场景。在某种意义上，这些工程师负责使用类和架构机制所定义的领域特定语言来编写小的程序。完成这项工作的另一个方法是让应用工程师负责更多的类细节设计，但要保证组件主管有充分的监督和引导。

应用工程师熟悉面向对象开发的表示法和过程，但不必是这方面的专家；然而，应用工程师必须是非常优秀的程序员，能够理解所使用的编程语言的惯用语和特质。平均来说，一支开发团队中应该包含一半或更多的应用工程师。

技能的这种分解提出了大多数软件开发组织都要面对的人员配备问题，这些软件开发组织通常只有少数几个真正优秀的设计人员，其余更多的是缺乏经验的人员。这种人员配备方法的社会效益在于，可以给开发团队中那些没有经验的人员提供一条职业途径。明确地说，就是初级开发人员在更高级开发人员的引导下以一种近似师傅/学徒的关系工作。初级开发人员在使用已经设计好的类的过程中可以学到一些经验，逐渐地，他们学会自己设计高质量的类。这种安排也就意味着不是每一个开发人员都必须是抽象方面的专家，但是随着工作阅历的增加，技能可以逐渐增长。

在更大型的项目中，还需要许多其他不同的开发角色来完成工作。这里列出的一些角色（如系统管理员）并不关心面向对象技术的使用，虽然他们之中有些人（比如复用工程师）与对象模型有特别的关系。

- 项目经理　负责项目的交付、任务、资源和日程安排的动态管理。
- 分析员　负责演化和解释最终用户的需求，其必须是这个问题域内的专家，同时必须不脱离开发团队内的其他人员。
- 复用工程师　负责管理项目的类、组件和设计的仓储；通过参与评审和其他活动，积极寻找共同点，使它们可以被采用。在项目或整个组织内获得（例如，通过商业的库）、生产和改造通用的类和组件。
- 质量保证员　负责度量开发过程的产物。通常指导对所有原型和产品发布的系统级测试。
- 集成经理　负责组装已发布组件的兼容版本，以便组成一个可交付的发布版本，负责维护发布产品的配置。
- 文档编写员　负责产出产品及其架构的最终用户文档。
- 工具编制工程师　负责创建和改编方便生产项目移交物的软件工具。
- 系统管理员　负责管理项目使用的物理计算资源。

当然，并不是每一个项目都需要所有这些角色。对小型项目来说，同一个人可以承担多种角色；而对大型项目来说，可能每个角色就代表了一整个开发组织。甚至对于更大的项目，可能还会有更多的角色，如业务架构师、方法学家、配置管理主管和业务分析师。其中的一些，如方法学家，可能不只服务于一个项目[29]。

经验表明：与传统方法比较，面向对象开发使采用更小的开发团队成为可能。实际上，一个大概有 30～40 个开发人员的团队在一年的时间内开发数十万行高质量的代码并不是不可能。但是，我们认同 Boehm 的观点，他认为：“最好的结果发生在更少和更为优秀的人身上”[7]。不幸的是，如果为一个项目配备人员时使用了比传统方法所需人数更少的人员，可能会产生一定的阻力。这种方式破坏了一些经理建立自己王国的努力。其他管理人员喜欢藏在众多雇员的后面，因为有越多的人就代表越大的权力。另外，如果一个项目失败了，也会有更多的下属来分担过失。

仅仅因为项目采用最精密的设计方法或最先进的热门工具，并不意味着管理人员有权利放弃雇佣那些能够独立思考的设计人员的职责或是让项目靠“自动驾驶仪”支配。

7.3 发布版本管理

与面向对象开发相关的发布版本管理和传统软件开发类似；它们为开发过程提供了基石。开发团队在集成和测试已开发软件块（从类到组件，最终到整个软件系统）时，必须管理正在开发的系统的配置。

7.3.1 配置管理和版本控制

考虑一下开发人员个人的困境，他可能负责实现一个特定的组件。他必须有一个该组件的工作版本，即正在开发的版本。为了继续进一步的开发，至少所有的输入组件接口都必须是可用的。当这个工作版本状态变得稳定时，就可以将其发布给一个集成团队，集成团队负责为整个系统收集一组兼容的组件。最后，组件集合被冻结下来作为基线，并构成一个内部发布版本的一部分。然后，这个内部发布版本变成现在操作的发布版本，对于所有需要进一步精化实现的特定部分的、活动的开发人员，它是可见的。同时，开发人员个人还可以在其组件的更新版本上工作。这样，因为有了明确定义和妥善保护的组件接口，开发就有可能稳定地并行进行。

这个模型中蕴涵的思想就是：版本控制的首要单元是一簇类而不是单个的类。经验说明，管理类的版本是一种过细的粒度，因为没有一个类是孤立的。相反，更好的做法是建立相关的一组类的版本。这并不意味着不对类做版本控制，只是这不是首要焦点。实际上，这就意味着对组件做版本控制，因为一组类映射到组件。在软件系统内更高的级别，将对由多个低级别组件构成的子系统做版本控制。

在系统演化过程中的任意给定点，一个特定组件可能存在多个版本：一个是针对正在开

发的当前发布版本，一个是当前的内部发布版本，还有一个是最新的顾客发布版本。这加剧了对强有力的配置管理工具和版本控制工具的需求。

源代码不是唯一应该置于配置管理下的开发产物。同样的概念适用于面向对象开发的所有其他产物，如用例规格、可视化模型和软件架构文档。

7.3.2 集成

产业化项目要求开发程序族。在开发过程中的任意给定时间，都会有多个原型和生产发布版本，以及开发和测试的脚手架。很多时候，每一个开发人员都拥有他自己正开发的系统的可执行视图。正如前一章所解释的，面向对象开发迭代和增量过程的本质意味着不应该只有一次“大爆炸”式的集成事件（虽然处在麻烦中的项目可能会发生这样的事情）。相反，通常会有很多次较小的集成事件，每一次都标志着创建另外一个原型或架构发布版本。每一个这样的发布版本通常在本质上都是增量式的，从以前稳定的发布版本演化而来。正如 Davis 等人所说的，“当采用增量开发时，最初软件被有意地设计成满足较少的需求，但是用这种方式构造软件便于结合新的需求，从而达到更高的适用性”[9]。从系统最终用户的视角来看，一个宏观过程生成一系列可执行的发布版本，每一个都有新增的功能，最后演化成最终的产品系统。从组织内部人员的视角来看，实际上会构造很多发布版本，并且只有一些被冻结下来作为基线，以稳定重要的系统接口。这种策略倾向于降低开发风险，因为在开发过程的早期，它加速发现架构设计和性能问题的步伐。

对于更大型的项目来说，一个组织可以每隔 2、3 个月产生一个内部发布版本。对需要多得多的开发工作量的更复杂项目，这可能意味着根据项目的需要，每 6 个月左右才有一个发布版本。在稳定状态下，一个发布版本包括一组兼容的组件及相关文档。只要项目的主要组件足够稳定而且可以很好地协同工作来提供一些新级别的功能，就有可能建造一个发布版本。

7.3.3 测试

持续集成的原则同样适用于测试，测试也应该是在开发过程期间持续的活动。在面向对象架构的上下文中，测试应该至少包含以下三个维度。

（1）单元测试，包含单个类和机制的测试，由实现结构的应用工程师负责。

（2）组件测试，包含一个完整组件的集成测试，由组件主管负责，组件测试可被用作对组件的每个新发布版本的回归测试。注意，组件是泛指的，可以是小项目中的单个组件或组件的集合，在更大的项目中有时指一个子系统。

（3）系统测试，包括系统整体的集成测试，由质量保证团队负责。在集成团队组装新的发布版本时，系统测试通常也用作回归测试。

在每一个级别，测试应该聚焦于测试对象的外部行为；测试的第二个目的是把系统推向极限，以便了解系统在某些条件下如何发生故障。

7.4 复用

面向对象开发说得最多的一个好处就是复用。但要得到复用开发过程中很多工件这样的好处，需要管理上的承诺。

7.4.1 复用的元素

软件开发的任何工件都是可以被复用的，包括（需求和测试的）用例场景、设计、代码和文档。正如第 3 章中所述，类用作复用的首要语言工具：类可以被子类化，以特化或扩展基类。同样，如第 4 章中所述，我们可以以惯用法、机制和框架的形式复用类、对象和设计的模式。以组件的形式复用协作的类，通常能提供最大的好处。比起单个类的复用，框架复用和模式复用处在更高的抽象级别，因此也将提供更强大的支持（但同时也更难做到）。

引用不同复用级别的数字是危险和容易形成误导的[10]。在成功的项目中，我们会遇到高达 70%的复用率（意味着几乎 3/4 的软件原封不动地来自于其他来源），也会遇到低至 0%的复用率。复用的程度不应该被视为一个必须达到的定额，因为潜在的复用随领域不同有着多种表现形式，同时也受到很多非技术因素的影响，这其中包括进度的压力、分包商之间关系如何以及安全方面的考虑。

归根结底，在项目中任何数量的复用都比没有强，因为复用代表资源的节省，否则就要重复解决先前已经解决的问题。

7.4.2 建立复用制度

在一个项目甚至整个组织内，复用不会自发产生，它必须被制度化。这意味着必须积极寻找复用的机会和奖励复用。实际上，这就是把模式提取作为一个明确的活动包含在宏观过程中的原因。

要获得高效的复用程序最好是由专人来负责领导复用活动，同时让每个人都参与。复用活动包括识别共同的机会，通常是通过架构评审来发现的；复用活动还包括使用这些机会，通常是通过生产新的组件或改造已有的组件并在开发人员当中支持复用来实现的。这种方法需要给复用以明确的奖励，即使是很简单的奖励也可以很有效地鼓励复用。例如，对于作者或复用者来说，得到同行的认可通常是有用的。

除了复用开发资产之外，还可以购买商业类和组件来协助我们开发。但是，依然必须在架构的框架内部开发一个高效的设计，以使用这些商业库资产。这不只是把类和组件插在一起那么简单[30]。

归根结底，复用在短期内会消耗一定的资源，但是将获得长期的回报。只有那些用长远的眼光来看待软件开发，并且不仅仅为眼前的项目优化资源的组织，才能使复用活动最终取得成功。

7.5 质量保证和度量

软件质量保证包括“为整个软件产品使用的适应性提供证明的系统活动”[11]。质量保证试图针对一个软件系统的质量有多好给出可量化的度量。很多传统的度量可以直接用在面向对象系统上。

7.5.1 软件质量

Schulmeyer 和 McManus 将软件质量定义为“整个软件产品使用的适应性”[12]。软件质量不是自发产生的，它必须被设计进系统中。实际上，面向对象技术的使用并不能自动带来高质量的软件，使用面向对象分析设计技能和面向对象编程语言仍有可能设计和编写出低质量的软件。

这就是在面向对象开发过程中我们将重点放在软件架构上的原因。一个简单的、可修改的架构是所有高质量软件的关键，它的质量完全通过实施简单一致、支撑战略设计决策的战术设计决策得到。

正如前面所述，即使是在面向对象系统中，开发评审和其他几种检查方法也是很重要的，它们提供对软件质量的深刻了解。关于软件质量优劣的最重要可计量标准也许就是缺陷发现率。在系统的演化过程中，我们根据软件出现缺陷的严重程度和位置来跟踪缺陷，因此，针对时间来画图，缺陷发现率是错误发现快慢的度量。正如 Dobbins 所说的，“线的斜率比错误的实际数字更加重要”[13]。受控的项目有一条钟形曲线，缺陷发现率在测试期的中点附近达到顶点，然后开始下降直到很低的数值。失控的项目将有一条下降很缓慢或根本不下降的曲线。

面向对象开发的宏观过程进展顺利的原因之一，就是它允许尽早和持续地收集有关缺陷发现率的数据。对于每一个增量发布版本，我们都应该执行一个系统测试并画出缺陷发现率针对时间的图形。即使早期的发布版本只有很少的功能，我们还是希望看到一个健康项目的每一个发布版本都有一条钟形的曲线。

缺陷密度是另外一个有关的质量度量。度量每千行代码（KSLOC）出现多少缺陷是一种传统的方法，同样也适用于面向对象系统。在健康的项目中，缺陷密度倾向于“在大约 10000 行代码被检查后达到一个稳定值，并且无论后面有多少代码都保持基本不变”[14]。

在面向对象系统中我们还发现，根据每个类中出现的缺陷数量来衡量缺陷密度是很有用的。采用这种度量，80/20 规则是适用的：80%的软件缺陷将会在 20%的系统类中被发现[15]。

7.5.2 面向对象度量

英国物理学家开尔文（开尔文温标就是以他的名字命名）这样说过测量，“当你可以测量你要说的东西并将其表达成数字时，你了解了一些事情；但当你不能测量它，不能将其表

达成数字时，你的知识是贫乏和不尽人意的。这可能是知识的开始，但你几乎没有想到这是迈向科学的阶段”[1]。对于面向对象度量，我们关心的是阐明如何提供有意义的测量来支持软件系统的分析和设计。

两个类目的度量能帮助我们：过程度量和产品度量。过程度量，有时叫作项目度量，可以帮助管理团队评价所用的面向对象开发过程的进展。过程度量的例子包括花费的人-时数量、完成的工作量和项目花费的美元数量——把所有这些和计划相比较。我们也可以看一看更特定于面向对象开发的度量，例如 Lorenz 和 Kidd [16]所推荐的：

- 应用规模
  - 场景脚本的数目（NSS）
  - 关键类的数目（NKC）
  - 支持类的数目（NSC）
  - 子系统的数目（NOS）
- 人员规模
  - 每个类人-天数（PDC）
  - 每个开发人员类数（CPD）
- 调度
  - 主要迭代的数目（NMI）
  - 已完成合同的数目（NCC）

我们倾向于通过统计逻辑设计中的类或物理设计中已经完成和正在工作中的组件，来测量开发进度。正如之前章节中所描述的，另一个进度测量是关键接口的稳定性（即，变化得有多频繁）。一开始，所有关键抽象的接口每天甚至每小时都会变。随着时间推移，最重要的接口首先稳定，次重要的接口接着稳定，等等。接近开发生命周期末期，只有一些不重要的接口需要改变，因为大多数精力被放在使已经设计好的类和组件一起工作上。偶尔，一些关键接口上的改变是需要的，但这样的改变通常是向上兼容的。即使如此，也只会在仔细思考它们的影响后才做出这样的改变来。这些改变可以增量地引入产品系统，作为常见发布生命期的一部分。

在这里，我们的首要焦点是产品度量（有时叫作设计度量），以帮助开发团队评价他们的分析和设计工件，而不是过程度量。我们发现，适当的产品度量可以帮助架构师和组件主管评价设计的质量。例如，他们能够知道继承树的深度范围是否满足设计指南。随着时间推移，在不同项目上针对这些度量获取和分析适用的量化测量，将得到一个历史数据库，它可以用作当前项目上所分析的测量的比较点。

Chidamber 和 Kemerer 建议以下几条独立于语言的度量标准可以直接应用于面向对象系统：[17]

- 每个类的加权方法（WMC）；

[1] 据说开尔文勋爵（Lord William Thomson Kelvin）是 1891～1894 年期间在 *Popular Lectures and Addresses* 做出这个陈述的。他还假设“比空气重的飞行机器是不可能存在的。”我们只同意他的第一个陈述。

- 继承树的深度（DIT）；
- 孩子的数量（NOC）；
- 对象类间的耦合（CBO）；
- 类应答数（RFC）；
- 方法内聚缺乏度（LCOM）。

每个类的加权方法给出了个体类每个方法的复杂性的累加。如果所有方法的复杂性都一样，就变成测量每个类中方法的数量。在分配相对复杂的值给每个方法时，这个测量真地管用；然而，出于灵活性的考虑，Chidamber 和 Kemerer 没有提供定义这种复杂性的方法。通常来说，一个有更多方法的类比同等的类会更加复杂，更加针对特定应用，并且常常会有更多的缺陷。[17]。

继承树的深度和孩子的数量是类结构的形状和大小的度量标准。正如第 3 章中所描述的，结构良好的面向对象系统应该被搭建成一个类的“森林”，而不是一棵非常巨大的继承树。从主题类到最高级别父类测量继承树的深度，给出了针对继承功能的影响的测量。因此，由于继承的功能，更深的继承树增加了一个类的复杂性。

从继承树向下看，可以看到主题类的孩子数目。类的孩子越多，由于它产生的复用，它在软件系统设计中的影响越大[17]。

对象之间的耦合是它们到其他对象的连通性的测量，所以是类的依附集的测量。与传统的耦合测量一样，我们寻求设计松耦合对象，它们有更大的潜在复用能力。

类应答数测量它的实例可以执行来应答消息调用的方法。一般来说，比起同伴有更多可以被调用的方法的类，就更复杂。

方法内聚缺乏度测量类的抽象的统一性。一个类的方法间低内聚说明这是偶然或不适当的抽象。这样的一个类一般应该被重新抽象为多个类，或者把它的责任委托给其他已有的类[17]。

在关于软件质量工程的著作中，Kan 讨论了如何应用 Lorenz（之后由 Lorenz 和 Kidd 在 1994 年提出）以及 Chidamber 和 Kemerer 提出的产品度量的例子。11 个 Lorenz 设计度量包括若干面向对象设计指南和应用它们的经验规则。Kan 发现经验规则“非常管用。它们来自行业 OO 项目的经验，提供了一个比较和阐释的入口”[18]。

将 Chidamber 和 Kemerer 度量（CK 度量）应用于若干研究后，Kan 发现“在决定 CK 度量的偏好阈值之前，需要更多的实证研究”[18]。事实上，Chidamber 和 Kemerer 说，应该为每个开发点决定阈值。Kan 确实发现，“在实践使用中，该度量可以用来标注，让类获得特别的注意力”[18]。

最近，Kemerer 和 Darcy 提供了若干应用 CK 度量集的例子，并提供了关于它的实践应用的观察。从这些应用的研究中，他们做了若干关于面向对象度量的观察[19]：

- 这样的度量已经成功地应用于若干领域；
- 它们一致地展示和质量因素的关系（如成本、缺陷、复用和可维护性）；
- 一般有用的集合包括规模（WMC）、耦合（CBO 或 RFC）和内聚（LCOM）；

- 应该为局部的影响计算度量及所得出的预测之间的关系。

面向对象设计原则如何有助于提高软件质量，对此依然有不同意见。因此，如何构造适当的面向对象度量集合，依然有很多争论。我们相信，这里展现的度量提供了合理的测量集合，来帮助架构师和组件主管评价他们的面向对象设计的质量。

7.6 文档化

除了代码之外，还有对完整的软件系统的生命周期至关重要的开发工件。这些工件，如需求和设计，必须被文档化以支持开发过程及系统的运营和维护。

7.6.1 开发遗产

一个软件系统的开发不仅包括源代码的编写，某些开发产品为管理团队和用户提供了查看项目进展的各种方式。我们还试图为系统的最终维护人员留下分析和设计决策的遗产。面向对象分析设计的产物是可视的模型，以图的方式创建了系统的多种视图。这些视图包括用例图、活动图、类图、状态机图、序列图和组件图。通过合适的指南，我们可以一起使用这些图来向后跟踪系统的需求。用例图（及用例规格）展示了需求指定的高级别功能，活动图描述用例场景的细节，类图则代表关键抽象，这些抽象组成问题域的词汇，有复杂的状态相关行为的类用状态机图来检查，序列图展示了提供系统功能时对象的协作，组件图展示类到组件的映射。

7.6.2 文档化的内容

对一个系统架构和实现建立文档非常重要，但是产生这样的文档并不能推动开发过程：文档化是开发过程中一个基本但是具有辅助性质的产物。记住文档是活生生的产品是很重要的，应该让文档和项目发布版本的迭代和增量演化过程一起进行演化。和设计以及生成的代码一起，被交付的文档可作为大多数正式或非正式评审的基础。

必须对什么建立文档？显然，必须产生最终用户的文档，用来指导用户如何安装和操作每一个发布版本[1]。另外，必须产生分析文档，以通过用例场景捕获系统所需功能的语义。还必须生成架构和实现文档，用来与开发团队交流关于架构的愿景和细节，也用来保存相关的战略决策信息，使系统易于修改和随时间演化。

通常，系统架构和实现的基本文档（不一定是纸上的）应该包括以下内容：

- 描述高层系统架构的文档；
- 描述系统架构的关键抽象和机制的文档；
- 描述系统关键方面的相关行为的场景文档。

[1] 这是一个不成文的规定：对于提升个人生产率的软件来说，一个需要用户经常使用手册的系统是不友好的。面向对象用户界面应该专门设计，使它们的使用是直观和自我一致的，以便把对最终用户文档的需要减少到最低限度。

对面向对象系统来说，可能创建的最糟糕的文档就是逐个类对每个方法的语义作孤立的描述。这种方法会生成很多无用的文档，没有人会阅读或相信这些文档，同时，这种方法没有把更为重要的超越单个类的架构问题（即类和对象之间，特别是组件之间的协作）写入文档。好得多的做法是以 UML 图的形式将这些更高层次的结构纳入文档，随后把开发人员的注意力引向某些重要类的接口的战术细节。

7.7 工具

面向对象开发实践改变了开发团队在分析和设计期间所需要的工具。复杂面向对象系统的开发更是有天翻地覆的变化——尝试用最小的工具集来建造一个大型软件系统，相当于用石块和手工工具来建造一栋多层建筑。由于面向对象分析设计突出了关键抽象和机制，因此我们需要那些聚焦于更丰富语义的工具。另外，由面向对象开发的宏观过程所定义的各个发布版本的快速开发，需要那些可以在分析设计周期提供快速转换的工具。

选择规模合适的工具很重要。一个工具可以供一个开发人员编写小的、孤立的应用，而它并不一定适用于更复杂应用的产品发布版本。实际上，对于每一个工具来说，都有一个能力的限度，超出这一限度，它的优点就会被缺点和笨拙所压倒。

7.7.1 工具种类

我们已经认定三种适用于面向对象分析设计的首要工具。第一种是支持 UML 表示法的可视化建模工具。这种工具可以用在分析期间以捕获用例场景的语义，还可用在开发过程早期，以捕获战略和战术上的设计决策，维护对设计产品的控制，并协调一个团队的开发人员的设计活动。实际上，当设计演化进入产品实现阶段的时候，可视化建模工具的使用可以贯穿整个生命周期。在系统维护期间这种工具也是有用的。特别是，我们发现这个工具可以用于对面向对象系统很多有趣的方面进行逆向工程，至少可以产生系统类结构和组件架构。没有这个特性，设计人员可能会生成奇妙的可视化设计表示，但因为程序员只是醉心于实现而没有更新这些设计，所以只有当实现开始进行时才会发现这些图已经过时了。逆向工程使设计文档跟不上实际实现的可能性变得很小。

下一种是软件配置管理和版本控制工具，特别对于更大型的项目来说这是必需的。在整个软件开发生命周期内，这样的工具支持开发团队协作和共享资产。这些资产包括从用例图到类图序列图的所有分析和设计过程的工件，类图序列图提供架构和在架构内协作的组件的设计。正如之前提到的，这些组件是最佳的配置管理单元，特别是从复用的视角来看。

第三种对面向对象分析设计比较重要的工具是类库工具。很多语言都有预定义的类库，或者可以另外购买类库。当一个项目成熟时，类库随着领域特定可复用软件类和组件的增加而增加。类库的库存量变得巨大并不需要很长的时间，这会导致开发人员很难找到满足自己需要的类或组件。如果查找某个组件的可察觉成本（通常都会高估）高于从零开始创建这个

组件的可察觉成本（通常都会低估），所有复用的希望就都破灭了。因此，在开发时，至少要有一些最小的库管理工具是很重要的，这些工具允许设计人员根据不同的条件找到类和组件的位置，并将有用的类和组件添加到库中。

这三个工具经常集成起来，让开发团队无缝地使用它们的聚合能力。虽然集成开发环境（IDE）的首要功能是提供一个编程环境，但它也可以提供一个可视化建模、配置管理及版本控制和类库工具进行协作的基础。

7.7.2 组织上的意义

对强有力工具的这种需要，导致开发组织内部需要两种特殊的角色：复用工程师和工具编制工程师。其中，复用工程师的职责就是维护一个项目的类库。如果没有积极的努力，这样一个类库就会变成一片庞大的存放垃圾类的荒地，没有开发人员愿意去那里找东西。积极鼓励复用也经常是必要的，复用工程师通过从现在设计成果中提取有用的产品使这个过程变得容易。工具编制工程师的职责是创造领域特定的工具和裁剪现有的工具使之符合项目的需要。例如，一个项目可能需要通用的测试脚手架来测试用户界面的某个方面，或者可能需要定制的类浏览器。工具编制工程师最适合制作这些工具，通常这些工具来自于已在类库中的组件。这样的工具还可以用来做以后的开发工作。但是，在有集成工具套件的最佳情况下，工具编制工程师的角色就不需要了，系统管理员可以管理集成套件。

已经面临过人力资源贫乏的管理者可能会惋惜使用强有力的工具以及任用复用工程师和工具编制工程师是过于奢侈的。对于一些资源有限的项目来说，我们不否认存在这种现实。然而，在很多其他的项目中，我们发现这些活动可以以任何方式（通常采用一种特别的方式）进行。我们提倡在工具和人力上进行明确的投资以使这些特别活动更加集中和有效，这样做增加了整个开发工作的实际价值。

7.8 特殊主题

人们实践面向对象分析设计时，有若干主题的特别关注点。领域特定问题包括开发高效用户界面和集成遗留功能——从数据到整个系统。另一个特殊的关注点是如何高效地采纳面向对象技术。

7.8.1 领域特定问题

我们发现，确定的应用领域为特定的架构方面的考虑提供了充足的依据。一个高效用户界面的设计依然更多地偏向艺术而不是科学。对于这个领域，原型的使用是绝对必要的。来自最终用户的反馈必须尽早被收集并经常补充，以便评估用户交互中的姿态、错误行为以及其他交互范式。用例场景的生成也能够高效地驱动用户界面的分析。

一些应用还包括一个主要的数据库组件，其他应用可能需要集成其方案不可改变的数据

库，这通常是因为大量的数据已经被装入数据库（遗留数据的问题）。对这样的领域，可以直接用分离关注原则：最好是将对所有这样的数据库的访问封装在已经定义良好的接口类的界限之内。当面向对象分解和关系数据库技术混合时，这项原则特别重要。

也要考虑实时系统，在不同的上下文中，实时意味着不同的东西：以用户为中心的系统中，实时可能表示亚秒级的响应；而对于数据获取和控制的应用，可能表示亚微秒级的响应。要认识到，即使是对于硬实时系统，也不是系统的每一个组件都必须（或能够）被优化，这是很重要的。实际上，对于很多复杂系统来说，更大的风险是系统能否完成，而不是系统是否在其性能需求范围内执行。基于这个原因，我们要提防过早发生的优化。聚焦于产生简单的架构，发布版本的逐代演化将可以尽早地描述系统性能的瓶颈，以便有时间采取正确的行动。

我们说的遗留系统指的是这样的系统：在软件上投资巨大，无论从经济还是安全的角度，都不能放弃这笔投资。然而，这样的系统可能会有令人无法忍受的维护费用，随着时间的推移，它们需要被逐渐取代。幸运的是，对待遗留系统和对待数据库很相似：将对遗留系统功能的访问封装在定义良好的接口类的上下文之中，并随着时间的推移，移动面向对象架构的覆盖范围以便取代由遗留系统当前提供的某些功能。当然，以最终系统看起来会是什么样的架构愿景来开始是有必要的，所以逐步替代遗留系统不会以不一致的软件修补工作作为结束。

7.8.2 采纳面向对象技术

正如 Stix 和 Mosley 所报告的，“作为信息系统社群对对象技术专家的市场需要的反应，需要涉及很多认知问题。软件从业者面临两个主要的挑战——理解对象和理解如何设计。此外，已收集的证据表明，编程构造和设计是两个独立的技巧集合，它们必须并发地学习，以便高效实现和达到对象技术的好处”[20]。

如何开发这种面向对象设计的能力？我们推荐以下设想。

- 为开发人员和管理人员提供以下正式培训：
  - 统一建模语言；
  - 项目将要使用的面向对象分析和设计流程；
  - 项目将要使用的工具；
  - 项目将要使用的语言和库。
- 先在一个低风险项目中使用面向对象开发，让团队通过以下途径学习：
  - 使用有经验的 OOAD 顾问作为项目团队的导师；
  - 培养团队成员中的高手，让他们作为面向对象方法的导师，去其他项目播种。
- 向开发人员和管理人员展示设计良好的面向对象系统。

依照我们的经验，一个专业的开发人员需要几周来掌握一门新的编程语言的语法和语义。同一个开发人员可能要更多周的时间来了解类和对象的重要性和威力。然而，我们看到过掌握面向对象设计概念和应用的非常不同的情况。Maksimchuk 和 Naiburg 从他们称为培训陷阱的视角看问题：“一门编程语言可能是面向对象的，但学会一门面向对象语言不意味着你

已经学会了使用 UML 来做好的面向对象设计的概念”[21]。这个开发人员还需要大约 6 个月的经历，才能逐渐成长为一个胜任的类设计人员。这不一定是坏事，对于任何学科，都需要花费时间来掌握它的艺术。

我们发现，通过例子来学习通常是一种直接和高效的方法。一旦一个组织积累了一堆重要的用面向对象风格开发的应用，那么将新的开发人员和管理人员引入面向对象开发就会容易许多。开发人员可能从分析师做起，当他们更加熟练把握面向对象技能时就会逐渐成长到设计角色。或者，他们从设计人员开始，使用现有的结构良好的抽象。随着时间的推移，在更有经验的人的监督下，已经学习和使用过这些组件的开发人员，将得到足够的经验来开发一个有意义的对象模型的概念框架，并变成一个高效的设计人员。

7.9 面向对象开发的好处和风险

关于面向对象开发的好处，已经兜售了很多年，而且已经十分现实。但是，如果没有面向对象开发过程的成功应用，风险近在咫尺。

7.9.1 面向对象开发的好处

面向对象技术的采纳者通常因为两个原因而接纳这些实践。第一，他们寻求竞争优势，例如，缩短产品上市时间，增强产品灵活性，改善生产进度的预计。第二，这些项目可能有一些非常复杂，以至于似乎没有其他任何解决方案的问题。

在第 2 章中，我们建议使用对象模型来引导构造系统，这个系统应该包括构建良好的复杂系统的 5 个属性：层次结构、相对的基础（即多个级别的抽象）、分离关注、共同模式和稳定的中间形式。对象模型形成了面向对象开发的表示法和过程的概念框架，所以这些好处确实来自于这种方法本身。在第 2 章中，我们注意到来自对象模型（也就是来自面向对象开发）的下列特征的好处：

- 采用人类的认知来工作；
- 使系统更具有便于修改的灵活性；
- 鼓励软件组件的复用；
- 减少开发风险；
- 利用面向对象编程语言的表现力。

一些案例研究证实了这些发现。特别是，这些研究指出面向对象方法能够节省开发时间和缩小最终源代码的规模，一些案例比另一些要好[22, 23, 24]。

7.9.2 面向对象开发的风险

在面向对象开发的黑暗面，我们也发现了风险。Hantos 的一篇文章展示了对这些风险的一个创新的研究，其中将“Bertrand Meyer 的经典 OO 技术概念映射到 Barry Boehm 的十大方

法学无关的软件风险，以阐明潜在的受影响领域”[25]。从 Meyer 的工作[26]，Hantos 得出以下面向对象概念列表，并把它们映射到 Boehm 的风险[25]：

- 用一种方法定义架构和数据结构实例；
- 通过抽象和封装达到信息隐藏；
- 继承以组织相关的元素；
- 多态以自动适配所操作结构的类型来执行操作；
- 特化的分析和设计方法；
- 面向对象语言；
- 促进创建面向对象系统的环境；
- 按契约设计，一种强有力的技能，巧妙解决模块边界和接口问题；
- 内存管理，可以自动回收不使用的内存；
- 分布式对象，促进创建强有力的分布式系统；
- 对象数据库，超越关系数据库管理系统的数据类型限制。

对于映射的其他方面，他用 Boehm 的 10 大软件风险[27]以及一个来自 Boehm[28]的更新列表，得出以下 8 个风险[25]：

- 人员短缺；
- 不现实的进度、预算或过程；
- 商业现货产品、外部组件或遗留软件的短缺；
- 需求或用户界面的不匹配；
- 架构、性能或质量不过关；
- 持续的需求变更流；
- 外部执行任务不足；
- 曲解计算机科学。

Hantos 针对 8 个风险的每一个，以及他列出的面向对象概念如何增加或帮助缓解软件开发项目内部的特定风险，提供了详细的解释。

为了从一个简单的可视视角来看结果，Hantos 在一张映射图中概括了他的研究。我们看到，和用其他方法一样，Boehm 经典的软件开发风险在面向对象软件开发中也存在。Hantos 从正面展示了 Meyer 描述的若干面向对象开发概念可以帮助缓解软件风险。具体地说，“架构和实例”的概念帮助缓解“持续的需求变更流”和“外部执行任务不足”的风险。“抽象和封装”的概念也有助于缓解“持续的需求变更流”的风险[25]”。

如果我们回忆对这些面向对象概念的研究，就可以理解它们的风险缓解效应。需求变更，特别是贯穿一个开发项目的持续的需求变更，可能会造成混乱。但是，通过聚焦于适当的逻辑和物理的系统类及组件结构（架构），我们可以提供结构和行为的分区，以减少贯穿系统的需求变更波动效应。任何项目团队引起的系统开发缺陷都可用类似的手段分隔。

在为软件开发项目开发风险管理计划时，理解软件开发项目的潜在风险（以及面向对象概念如何潜在地增加或缓解它们）是重要的。

7.10 小结

- 一个复杂软件系统的成功开发和实施不仅仅包含产生代码。
- 软件开发管理的许多基本实践（如走查），都不受面向对象技术影响。
- 在稳定状态下，面向对象项目在开发期间资源方面的开销通常会降低。这些资源所扮演的角色与非面向对象系统有细微的不同。
- 在面向对象分析设计中，使用迭代的方法，永远都不应该只有一次“大爆炸”式的集成事件，发布版本的配置管理单元应该是组件，而不是单个类。
- 复用必须制度化才能成功。
- 缺陷发现率和缺陷密度是面向对象系统质量的有用测量标准。其他有用的测量标准包括不同过程和产品的度量。
- 文档决不应驱动开发过程。
- 与非面向对象系统开发相比，面向对象开发需要稍微不同的开发工具。
- 使用对象模型的组织需要思维方式上的转变。开发团队理解面向对象分析设计技能是关键的，面向对象软件开发不只是关于编程。
- 面向对象技术有很多好处，但也存在风险。好的风险管理有助于获得好处并最小化风险。

应用

要建立一种理论，就需要知道许多关于这个主题的基本现象。在计算理论方面，我们对这些知道得还不够，因而不能非常抽象地讲授这个主题。相反，我们应该多讲授一些我们现在能完全理解的特定例子，并希望我们能从中猜测并证明更多的一般原理。

——MARVIN MINSKY
Form and Content in Computer Science

方法是一个非常好的东西，但是从有经验的工程师的视角来看，如果我们曾设计的最优雅的表示法或过程无助于为现实世界建立系统，那么它也是完全无用的。本书前 7 章已经为本部分进行了一些铺垫，现在我们要将面向对象分析设计应用于软件系统的实际构造了。我们从不同的领域中选择一组应用，包括导航、指挥和控制、密码分析，数据采集和 Web 商业应用设计，每一个应用都涉及它们各自的一组问题。

本部分共有 5 章，我们将依次通过宏观过程的阶段，展示面向对象分析设计技能的应用。从先启出发，经过精化到构造。(移交的绝大部分内容超出了本书的范围。但是，我们将展示一些令人感兴趣的关于后移交的考虑。)即，每一章将主要强调宏观生命周期的一个特定部分，以及适用的分析和设计（即微过程）技能。我们相信，比起简单地聚焦于单个问题走遍面向对象分析设计的所有步骤，这是更令人感兴趣的方法。

本部分中的每一章不但聚焦于这里展示的开发的特定方面，也包括其他必要的方面，以提供上下文并使读者更好地理解这些章节的首要焦点：

- 第 8 章（基于卫星的导航） 聚焦于系统架构；
- 第 9 章（控制系统） 聚焦于系统需求；
- 第 10 章（密码分析） 聚焦于分析；
- 第 11 章（数据采集） 聚焦于分析到初步的设计；
- 第 12 章（Web 建模） 聚焦于详细设计和实现。

每一章自身可以被扩展为一整本书。因此，我们不能涉及每一个阶段、每一个活动和过程中的每一个步骤。但是，我们尽量涉及那些最令人感兴趣和重要的关键方面。

面向对象分析设计科目的关系和应该用什么图，并不是死板的或者有什么规定。某个图通常在一个阶段比另一个阶段出现得多。在项目生命周期的早期阶段，用例图更常见。一些图在真实项目中很少碰到。但是，正如你将在以下章节中看到的，某种类型的图的使用会贯穿整个项目生命周期。不同的是图所捕获的抽象级别。例如，在生命周期早期，组件图可能捕获非常大的、粗粒度的元素（如系统或子系统）。在生命周期后期，组件图可以用来捕获细粒度的实现元素（如可执行的软件）。贯穿本部分的所有章节，将看到抽象级别上的精化。

第8章

系统架构——基于卫星的导航

本书的前面部分展示了面向对象分析设计的原则和过程，还在第5章讨论了UML 2.0表示法。它们既被应用在最高级别的系统架构的开发中，也被应用在软件的开发中。对于系统架构来说，虽然比起开发类的结构和设计，我们关注理解系统需求并使用这些知识来把更大的系统分区成它的组成片段。但是，我们必须记住这个级别的关注点通常是十分抽象、宏观和影响巨大的，不涉及实现或技术细节。在设计架构时，如果我们理解这一点并采取正确的步骤，将更有可能创建一个生命很长的系统——更加可操作、可维护和可扩展。本来就应该是这样。

本章中将展示如何在逻辑上对所需功能分区，以开发假想的卫星导航系统（SNS）的系统架构。为了保持这个问题可管理，我们从简化的视角开发第一和第二个级别的架构，分别是定义组成片段和子系统。我们将展示一个有代表性的过程步骤和开发工件子集，而不是所有步骤和工件。从一个更完整的视角展示这些片段及其子系统的规格，很容易花上一整本书的篇幅。但是，我们展示的方法可以跨越一个架构级别（如片段或子系统）以及贯穿卫星导航系统架构的多个级别，这样应用得更完全。

我们选择这个领域，是因为比起只是作为案例问题而发明的简单系统来说，它在技术上复杂得多，而且非常有趣。今天，存在两个主要的基于卫星的导航系统，即美国的全球定位系统（GPS）和俄罗斯的全球导航卫星系统（GLONASS）。此外，第三个被称为Galileo的系统正在由欧盟开发。

8.1 先启

开发系统架构的第一个步骤实际上是系统工程步骤，而不是软件工程步骤，即使对于纯软件系统或绝大部分是软件的系统来说也是这样。系统工程被国际系统工程委员会（INCOSE）定义为“使得系统能够成功实现的跨学科方法和手段”[1]。INCOSE进一步定义

系统架构（这是我们在这里关注的焦点）为“布置元素和子系统，并给它们分配功能，以满足系统需求”[2]。

在这里关注的焦点是，通过定义问题的边界决定必须为顾客建造什么，决定使命用例，然后通过分析使命用例之一来决定一个系统用例子集。在这个过程中，我们从功能需求开发用例，并对非功能需求和约束归档。但在跳到需求分析之前，请先阅读一下补充材料“全球定位系统介绍”。

全球定位系统介绍

全球定位系统为任何拥有 GPS 接收器的人提供知道他在地球上位置的能力，而不管其位置在哪里，无论什么时间或天气[1]。GPS 卫星的轨道离地球 11000 海里，通过全世界的地面站可对其进行控制和监控。从第一颗 GPS 卫星在 1978 年升空，到 1994 年是第 24 颗，系统完成后，GPS 成为世界范围导航的福音[3]。

最早时，人们通过记住和认识日常生活中的路标来导航，后来，很多技术被开发出来，一直到今天的 GPS。在这期间，人们使用过地图和星图、罗盘、六分仪、记时计，以及现在的地面无线电导航系统，如 LORAN（长距离导航）[4]。

GPS 架构包括 3 个片段：控制、用户和空间。控制片段由 6 个地面站组成，主控制站位于科罗拉多的希尔（Schriever）空军基地。帮助很多人导航的接收器构成了用户片段，它接收来自 24 颗卫星的位置信息，包括空间片段的星座[5]。

GPS 接收器使用卫星数据广播时间和位置计算它们和卫星的距离。具体地说，“如果知道在空间上离卫星的确切距离，即可知道我们在想象中的球体表面的某个地方，该球体的半径等于我们到卫星半径的距离。如果知道离两个卫星的确切距离，即可知道我们位于两个球体相交线上的某处。如果再有来自第三、第四个卫星的测量，就可以发现我们的位置。GPS 接收器执行卫星范围测量，并得出它的位置”[6]。

全球定位系统有众多的应用，包括军事的和民用的。大多数人熟悉它的军事使用，军人用它在陆地上、海上和空中导航。它也被用在武器系统上，例如，为巡航导弹提供精确的实时导航来瞄准目标。但和很多人的生活相关的是民事应用。GPS 用于紧急服务，为需要的人们提供快速支持。它曾被用在英吉利海峡隧道的建造中，以保证来自英国和法国的不同挖掘团队在中间的精确位置相遇。它甚至被用于众多个人活动中，如驾驶、地理藏宝（geocaching）[2]和徒步旅游[7]。

8.1.1 卫星导航系统的需求

建造系统来帮助解决顾客问题的过程，开始于决定我们必须建造什么。第一步是阅读顾

[1] 以上介绍的信息来自 Aerospace Corporation 开发的 *GPS Primer—A Student Guide to the Global Positioning System*。更多的信息可参见 Aerospace Corporation 的 *Crosslink* 季刊 2002 年的 Summer 号，该杂志关注于卫星导航和 GPS。

[2] 全球 GPS Cache Hunt 的官方站点是 http://www.geocaching.com/。

客给我们的任何陈述问题或需要的文档。对于系统来说，我们已经得到了一份和高级别需求及约束相关联的愿景陈述。

愿景：

- 为顾客提供高效的和买得起的卫星导航系统服务。

功能需求：

- 提供 SNS 服务；
- 操作 SNS；
- 维护 SNS。

非功能需求：

- 保证充分服务的可靠性级别；
- 足够精度以支持当前和将来的用户需要；
- 关键系统能力上的功能冗余；
- 广泛的自动化以最小化操作成本；
- 容易维护以最小化维护成本；
- 可扩展以支持系统功能的升级；
- 服务生命期长，特别是空基（space-based）元素。

约束：

- 兼容国际标准；
- 最大化使用商用现货（COTS）硬件和软件。

显然，这是一个高度简化的需求陈述，但它确实提供了基于卫星的导航系统非常基本的规格。实践上，像这么大一个系统的详细需求只有在展示解决方案的活力，经过涉及众多领域专家和系统最终用户及客户参与的几百个人月的分析之后，才能得到。最终，一个大系统的需求可能包含数千页的文档（希望是可视的模型），不只阐明系统的通用行为，而且还包含错综复杂的细节，如用于人/机交互的屏幕布局。

8.1.2 定义问题的边界

虽然只是最小的需求和约束，但它们确实允许我们迈出设计卫星导航系统架构的重要的第一步——定义它的上下文，如图 8-1 所示。上下文图可让我们清晰理解环境内 SNS 必须提供的功能。执行者代表和系统交互的外部实体，包括人、提供服务的其他系统和现实环境。依赖箭头展示了外部实体依赖于 SNS，还是 SNS 依赖于它。

显而易见，User（用户）、Operator（操作员）和 Maintainer（维护人员）执行者依赖于 SNS，分别使用它的导航信息、操作它和维护它。虽然卫星导航系统有能力生成它自己的电源作为地面系统的备份，主电源服务还是由外部系统 ExternalPower（外部电源）执行者提供。类似地，ExternalCommunications（外部通信）执行者为 SNS 提供购买的通信服务，在某些情况下充当首要服务，在另一些情况下，作为内部提供的系统通信的备份。我们给这两个执行者的名字加上前缀“External（外部）”，以清晰地把它们和内部的系统电源

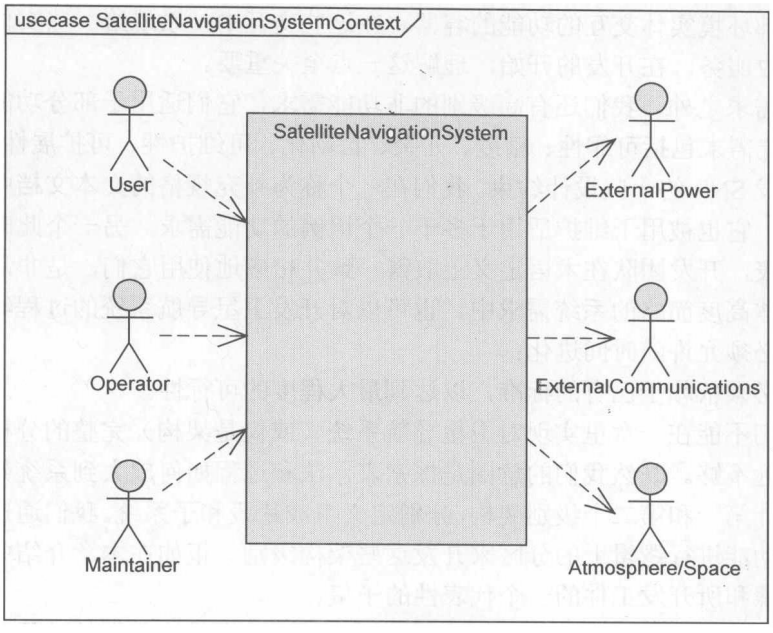


图 8-1 卫星导航系统上下文图

及通信服务分开。

剩下的执行者 Atmosphere/Space（大气/空间），可能看起来相当奇怪，直到我们把它看作卫星导航系统的地面和空基资产之间的通信的传输介质。因此，它是一个服务提供者。它的状态当然影响这些通信的质量。关于这个执行者，另一个方法是从约束“兼容国际标准”的视角来看。众多的国家和国际规范及条约管制着卫星传输，因此，我们有重要的理由来阐述这个执行者。

上下文图的一个关键点是系统的确切边界，即什么在系统里面，什么不在里面。有些人可能会质疑我们把 Operator 和 Maintainer 执行者放在 SatelliteNavigation 系统包的边界外面。通过这样做，可以从特定涉众即顾客的观点看问题，它们关注于给用户提供导航信息的系统。顾客关注的不是 SNS 运作于其内的更大的企业范围，不像 User 执行者，他可能把 Operator 和 Maintainer 看作是系统内部的。显然，这里的关键点是人的视角。例如，如果我们提供一个完整的一站式系统，包括操作和维护服务，我们就把 Operator 和 Maintainer 执行者放在 SatelliteNavigation 系统包边界里面。

我们已经看到，众多的上下文图表现变体，有些非常精致，有些非常简单。更精致的倾向于提供执行者和正在开发的系统之间两个方向流动信息的细节。如果系统在一个更成熟的环境内开发，可能是作为已有系统的代替品，这种类型的信息在开发周期的早期就已被了解，所以一些开发团队选择在这里表现它。

细节不重要，重要的是开发团队选择一种风格，对它归档，遵循它，以保证其清晰和容易理解。但是，我们更喜欢展示一张上下文图，因为它简单和清楚地传达了高级别概念：系

统是与它的外部环境实体交互的功能的容器。在这些交互中，系统给一些实体提供服务，从另一些实体接收服务。在开发的开始，理解这一点至关重要。

除了功能需求之外，我们还有高级别的非功能需求，它们适用于部分功能能力或整个系统。这些非功能需求包括可靠性、精度、冗余、自动化、可维护性、可扩展性和服务生命期。我们也看到开发 SNS 的一些设计约束。我们在一个称为补充规格的文本文档中维护非功能需求和设计约束，它也被用于维护适用于多于一个用例的功能需求。另一个此时必须开始的关键文档是词汇表，开发团队在术语定义上取得一致并相应地使用它们，是非常重要的。

即使从这些高度简略的系统需求中，也可以对开发卫星导航系统的过程给出两点意见：

（1）架构必须允许随时间进化；

（2）实现必须依赖于已有的标准，以达到最大程度的可行性。

显然，我们不能在一章里实现对卫星导航系统（或仅是架构）完整的分析或设计，即使是一本书也远远不够。既然我们的意图是探索表示法和过程如何放大到系统架构的开发，那么就聚焦于设计第一和第二个级别架构，分别定义组成片段和子系统。我们通过对 `Operator` 执行者使用的功能进行逻辑上的分区来开发这些架构级别。正如在本章介绍中所说的，我们只展示过程步骤和所开发工件的一个代表性的子集。

在审查愿景和需求之后，我们（架构团队）认识到，提供给我们的功能需求实际上是众多使命级别用例的容器（在 UML 中就是包），这些用例定义卫星导航系统必须提供的功能。这些使命用例包给我们提供 SNS 的高级别功能上下文，如图 8-2 所示。这些包包含展示 SNS

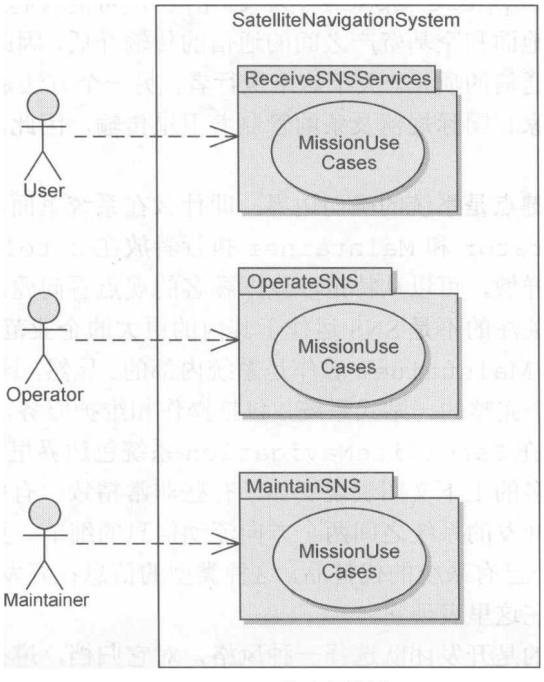


图 8-2 SNS 使命用例包

的用户、操作员和维护人员如何和要实现的系统交互以完成其使命的使命用例。既然我们在使用面向对象分析设计技能和 UML 2.0 表示法来执行一个系统工程而不是一个软件工程任务，那么对于如何使用图 8-2 展示的表示法，我们可能有点陌生。但是，我们相信它清晰展示了所需信息，并确保了可理解性。

8.1.3 决定使命用例

系统的愿景陈述相当开放：为我们的顾客提供高效和买得起的卫星导航系统服务。因此，架构师的任务是需要明智地修剪问题空间，使得问题可以解决。类似这样的问题容易遭受分析瘫痪，因此我们必须聚焦于提供最通用的导航服务，而不是试图使导航系统变成一个什么都有的大杂烩（结果是不能为任何人提供有用的功能）。我们将从开发 SNS 的使命用例开始。

像这样的大项目通常围绕一些小的、位于中心的负责建立总体系统架构的团队来组织，把实际的开发工作外包给其他公司或同一公司内的不同团队。甚至在分析期间，系统架构师通常会考虑一些划分实现元素的概念模型。基于我们建造、操作和维护基于卫星的系统的经验，我们相信最高级别的逻辑架构包括 4 个片段：地面、发射、卫星和用户。

有人可能争辩说，这是设计，不是分析，但我们反驳说一个人必须开始在某些点上约束设计空间。实际上，确定这个逻辑架构代表系统需求还是系统设计是困难的。抛开这个问题，这个开发阶段的系统架构主要是面向对象的。例如，架构展示了复杂的对象，如地面片段和卫星片段，每一个执行系统的主要功能。这正如在第 4 章讨论的，大系统中，在最高级别的抽象的对象倾向于沿着主要的系统功能线聚集。如何识别和精化这些对象，在分析期间和在设计期间所做的稍有不同。

甚至在有一个如图 8-3 所示的包图级别的概念架构之前，我们就可以开始分析，和领域专家一起工作，来清晰描述系统所需行为的首要使命用例。说“甚至在这之前”是因为，即使我们有 SNS 架构的一个表示法，也应该从黑盒视角开始我们的分析，这样就不会不必要地约束它的架构。即，首先分析所需功能来决定 SNS 的使命用例，而不是单个 SNS 片段的。然后，我们以卫星导航系统的白盒视角，分配这个用例功能到某个片段。

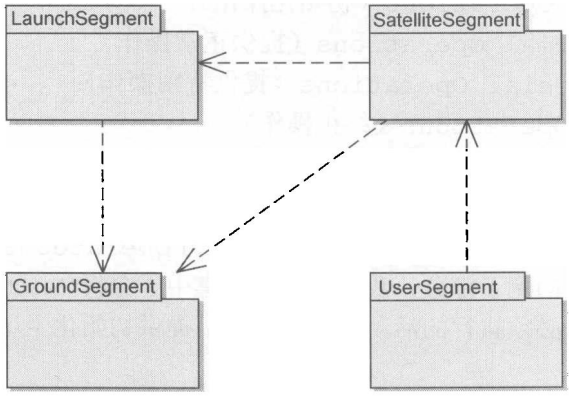


图 8-3 SNS 的逻辑架构

Rumbaugh、Jacobson 和 Booch 在他们所著的 *Unified Modeling Language Reference Manual* 中陈述，“活动图有助于理解系统的高级执行行为，而不涉及协作图所需的消息传送内部细节”[8][1]。因此，活动图是我们分析使命用例、阐述所期待的系统行为的首要工具。一些开发团队使用序列图或通信图来达到此目标，但我们相信，那些图更适合低级别架构层次的设计工作。在分析时，我们只关注于使命用例的成功场景，而把众多的备选场景留到以后。

术语成功场景（success scenario）读者可能不熟悉，众所周知的 ATM 例子帮我们解释了这一点。ATM 的一个用例是 `Withdraw Cash`（提取现金），这通常是我们在这些机器上所要做的。提取现金时，我们通过很多不同步骤和 ATM 交互：插卡，输入 PIN，选择取款，选择金额，等等。但这些步骤都没有体现最终目标（提取现金），因此它们本身不是真正的用例。`Withdraw Cash` 用例（和所有用例）包含很多不同的场景，每一个场景代表用例功能的一条路径。第一个是成功地提取现金，因此术语成功场景也叫作主要场景。备选或次要场景通常处理从主要场景分支的场合。例如，沿着主要场景 `Withdraw Cash` 往下走，到达选择所需金额这一点。ATM 回应所要求的金额超过了每天允许的金额，并请求我们选择另一个金额，我们照做，得到现金（虽然比我们最初想要的少）。这就是对一个次要场景的阐述。它跟随首要（成功）场景路径的若干步骤，然后偏离轨道去处理金额问题，然后跳回主要场景。

幸运的是，我们已经澄清了这个解释。但还要说一点，对于实时系统，如卫星导航系统，要明白它的很多功能是在次要场景中体现的。这些可能被认为是冰山在水面下的一部分，然而其对系统完整和安全的操作很关键。隐藏在幕后的次要场景通常得到关注较少，但它们可以损坏系统，就像冰山的水下部分能够把船撞沉。简言之，我们的分析必须包括次要场景。次要场景中包含的系统功能数量会有不同，但通常在这样的系统中是比较重要的。在这里我们不考虑它——但是，我们必须认识到真实的系统开发工作量。

现在，回到手边的任务，开发 `OperateSNS` 使命用例包的使命用例。基于对 SNS 总体操作的分析，定义 4 个相应的使命用例：

- `Initialize Operations`（初始化操作）；
- `Provide Normal Operations`（提供正常操作）；
- `Provide Special Operations`（提供特别操作）；
- `Terminate Operations`（终止操作）。

这些使命用例的规格主要依赖于开发团队的领域专家经验。除了过去的经验之外，模拟和原型经常是这个分析过程中有价值的工具。虽然我们的分析通常会使用活动图建模，就像在开发以下小节的系统用例时所做的。图 8-4 描绘了为 `OperateSNS` 使命用例包开发使命用例的分析结果。在本章的剩下部分，我们的工作将集中于分析 `Initialize Operations` 使命用例，以决定系统必须执行的活动，来为操作员提供初始化卫星导航系统操作的能力。

[1] 在 UML 2.0 中，协作图叫作通信图。

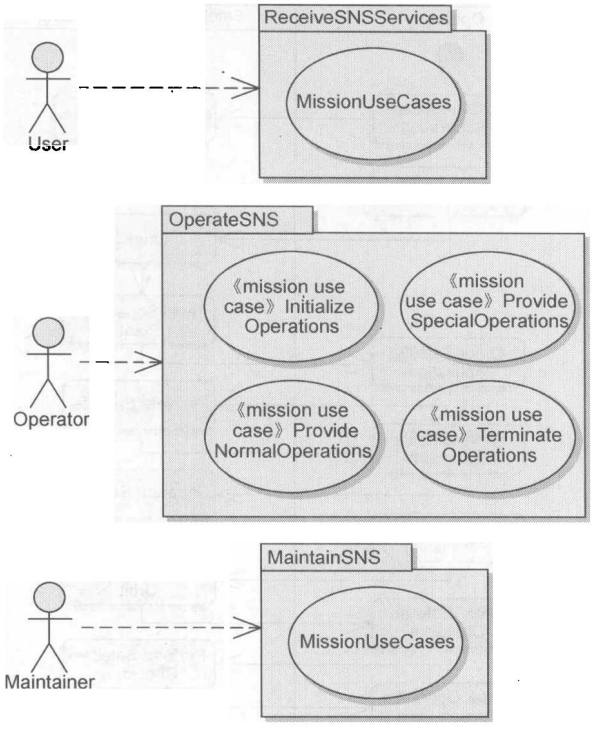


图 8-4 精化 OperateSNS 使命用例包

8.1.4 决定系统用例

正如之前阐述的，我们开发一张 Initialize Operations 使命用例功能的活动图来决定已被封装的系统用例。在开发这张活动图时，我不想使用表达组成 SNS 的片段所用的表示法（参见图 8-3）。这样做是因为不想因为预设手边问题可能的架构解决方案而约束我们对 SNS 操作的分析。我们聚焦于 SNS，把它当作一个黑盒，我们不能看到里面，只能看到它提供什么服务，而看不到它如何提供服务。在分析系统的高级别执行行为时，我们对跨越操作员和卫星导航系统之间边界的控制流感兴趣。

既然我们关心由 SNS 执行的活动，而不是通信或序列图里表达的消息传送，活动图是相对简单的。如果要定义整个 SNS 的系统活动，我们将为它的每个使命用例执行基于活动图的分析，以发现卫星导航系统必须执行以满足需求的众多活动。可以试想一下为了操作这样一个系统必须一天 24 小时执行的所有活动。但是，在这里我们聚焦于 Initialize Operations 使命用例，开发如图 8-5 所示的活动图。

从这个活动图，通过系统工程经验来判断，我们开发各自的系统用例列表。例如，决定把动作 Prepare for Launch（准备发射）和 Launch（发射）结合进一个系统用例 Launch Satellite（发射卫星）。剩下的动作体现重要的系统功能，因此我们决定每一个动作应该代表一个系统用例，得到 Initialize Operations 使命用例的系统用例，如表 8-1 所示。

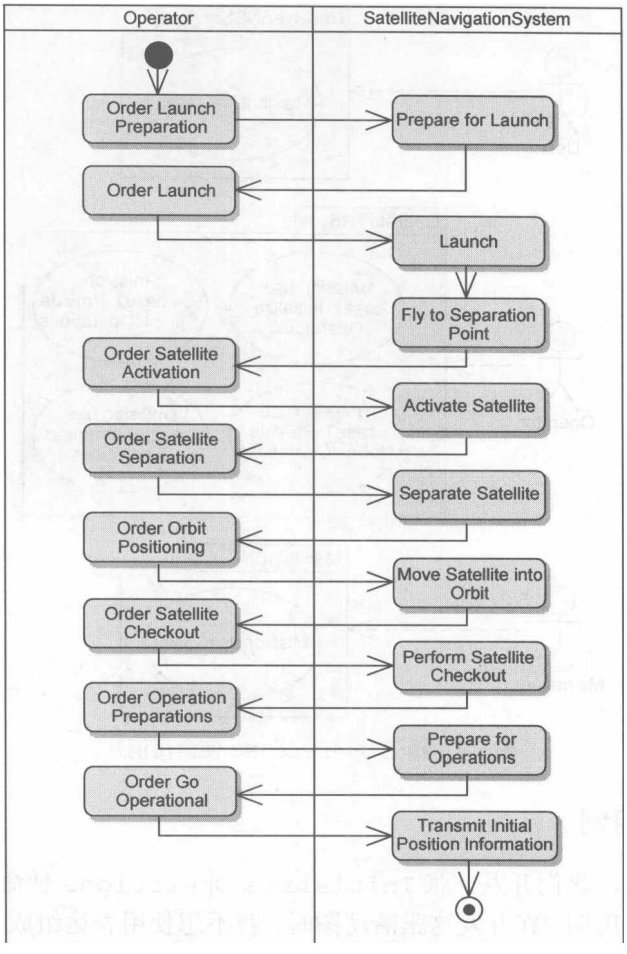


图 8-5 Initialize Operations 的黑盒活动图

表 8-1 Initialize Operations 系统用例

系统用例	用例描述
Launch Satellite	为发射准备发射器和它的卫星载荷，以及执行发射
Fly to Separation Point	发射器飞行到卫星载荷即将分离的点。这涉及多个发射器级别的使用和分离
Activate Satellite	执行卫星的激活，从发射器准备它的部署
Separate Satellite	从发射器部署卫星
Move Satellite into Orbit	使用卫星总线推进能力来定位卫星在正确的轨道平面内
Perform Satellite Checkout	执行轨道内检查卫星的能力
Prepare for Operations	在开始操作之前执行最后的准备
Transmit Initial Position Information	开始操作并传输初始位置信息给 SNS 用户

图 8-6 在一个更新过的用例图中展示了表 8-1 的系统用例。在这里，我们使用了 `InitializeOperations` 包来包含从 `Initialize Operations` 使命用例开发的系统用例。其他三个体现操作 SNS 的功能的使命用例，以带关键词标签`<<mission use case>>`的方式展示。可以发现，这个建模方法有用而且清楚；但是，每个开发团队需要决定和归档它所选择的技能。

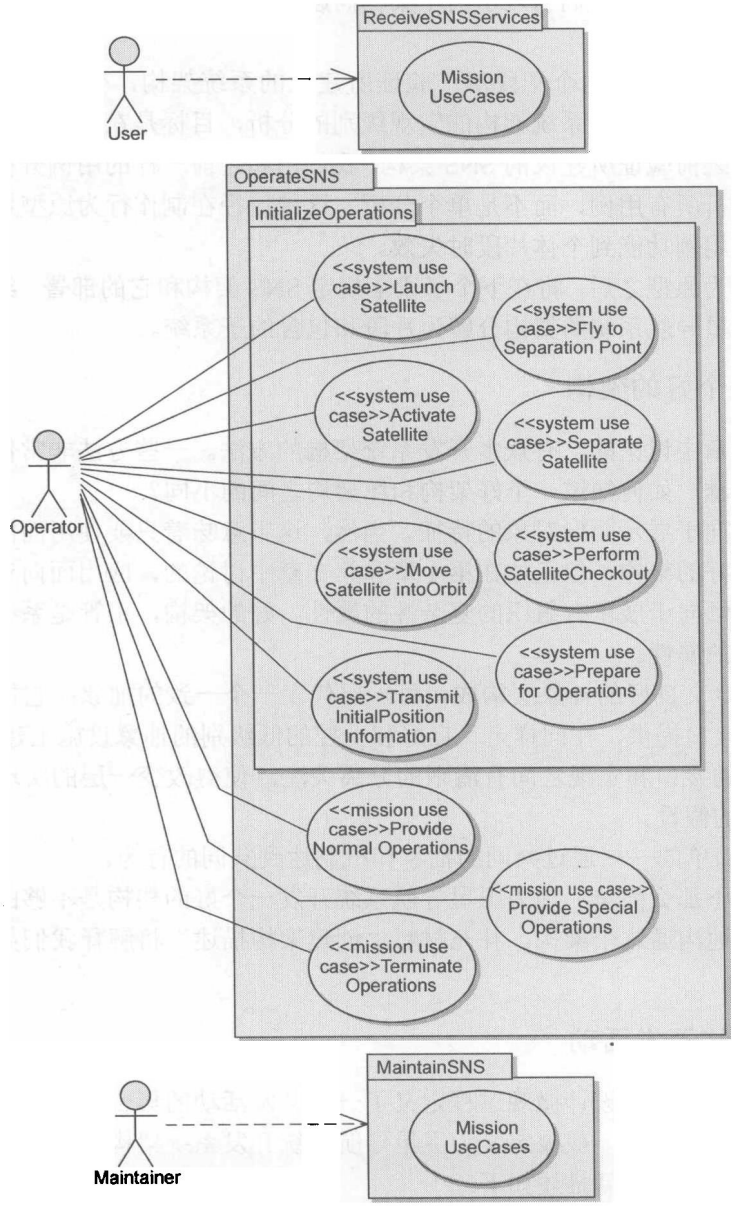


图 8-6 `Initialize Operations` 系统用例

8.2 精化

接下来讨论系统架构，该架构为实现之前段落开发的系统用例所包含的需求提供了基础。在头两个子段落中，我们引入了两个架构问题：开发一个好的架构时的关注点和开发一个架构时执行的活动。

这个讨论将引入下面两个子段落（验证所建议的系统架构，分配非功能需求和阐明接口），在那里我们执行 SNS 系统架构的宏观级别的分析，目标是在分析片段和阐明它们的协作子系统的架构之前验证所建议的 SNS 架构。我们用和之前一样的用例分析技能来开发系统用例，但同时分析所有用例，而不是单个分析。这种捷径在制作行为原型期间特别有效，但在确实分配系统用例功能到个体片段时失效。

完成制作行为原型之后，将在下个子段落规定 SNS 架构和它的部署。继续实际系统架构分析工作，将卫星导航系统的架构分解为片段和包含的子系统。

8.2.1 开发一个好的架构

正如在第 6 章中讨论的，有众多开发系统架构的方法。一些方法非常优雅，不幸地是，一些方法极其愚蠢。如何知道一个好架构和坏架构之间的不同？

好的架构倾向于展示面向对象的特征。当然，这不意味着只要使用面向对象技能，就一定能开发出一个好的架构。但正如在第 1 章和第 2 章中讨论的，应用面向对象分解背后的原则，产生的架构倾向于展示有组织的复杂性的属性。好的架构，不管是系统的还是软件的，通常有若干共同的属性。

- 它们由定义良好的抽象层构成，每一层代表一个一致的抽象；它们通过定义良好和受控的接口提供，在同样定义良好和受控的低级别的抽象设施上建造。
- 每一层的接口和实现之间有清晰的分离关注，使得改变一层的实现不会侵犯它的客户所做的假设。
- 架构是简单的——通过共同的抽象和机制达到共同的行为。

简单（或者不那么简单）地为卫星导航系统开发一个好的架构是不够的，我们必须和所有它的涉众高效地沟通这个架构。补充材料“创建架构描述”将解释我们如何着手开始这项任务。

8.2.2 定义架构开发活动

第 6 章展示的分析和设计微观过程定义了一个开发活动的集合，这些开发活动在系统内每个抽象级别执行。活动一般被定义为卫星导航系统开发系统架构必需的系统工程任务，这里从我们所聚焦的角度重新描述如下：

- 识别给定抽象级别的架构元素，进一步建立问题边界和开始面向对象分解；

- 识别元素的语义，即建立它们的行为和属性；
- 识别元素之间的关系以固化它们的边界和协作者；
- 指定元素接口，然后精化它们，准备下一个抽象级别的分析。

创建架构描述

在第 6 章的补充材料“归档软件架构”中，我们解释了归档系统的架构对架构师和其他系统涉众有何重要的价值。我们也讨论了 IEEE 1471-2000 标准，IEEE 推荐软件密集系统的架构描述实践，以及由 Kruchten 提出的 4+1 视图，这 5 个软件架构视图是用例视图、逻辑视图、实现视图、进程视图和部署视图。

虽然 IEEE1471-2000 标准聚焦于软件架构，Maier、Emery 和 Hilliard[9]认为“它一样适用于任何系统，因此适当使用它作为系统工程的一部分来描述系统架构”。此外，他们还说，“一个特别重要的 ANSI/IEEE 1471-2000 在系统工程中的应用可能是调和现在流行起来的巨大架构框架阵列”[10]。他们比较了若干更常用架构框架的观点：4+1、ISO RM-ODP、DoDAF 和 Zachman。

类似地，Kruchten 提出的 4+1 视图也适用于系统工程。Krikorian 发布了“增强版 4+1 视图”，他从系统工程的视角定义了 Kruchten 的视图，定义了合适的活动和工件[11]。

这个活动集合使得开发 SNS 架构时的首要关注点十分清晰，就是定义 SNS 架构的元素（片段和子系统）、它们的责任、它们的协作和它们的接口。这些提供给架构师一个框架，来演进架构和探索备选的设计。要记住的一点是，这些活动通常是并行执行的，而不是顺序的。例如，识别元素之间的关系可以帮助我们更好地建立它们的行为和属性。在以下小节中，我们将定义卫星导航系统的片段、它们提供的功能、它们彼此之间的协作和它们的接口。

8.2.3 验证所建议的系统架构

建议给予系统架构师机会来实验备选的系统分解，这样我们可以有充分的信心认为我们的总体设计决策是靠得住的。这可能涉及非常大规模的建模、模拟或制作原型。这些模型、模拟和原型会在这个系统成熟的过程中作为回归测试的工具被继续使用。

在本小节和下一个小节，我们将执行一个 SNS 系统架构的宏观分析，以便在继续进行之前，验证我们的假设和决策。要保证现在发现架构的任何问题，而不是以后。同样的道理，在开发生命周期早期，容纳需求变更要更为简单，需要的开销更低，架构变更也是如此。我们主要关注于 `Initialize Operations` 功能，因为在之前的开发中这曾经是一个有问题的区域。

这里须特别提醒一下：虽然我们在本小节和下一个小节使用的基本技能和开发系统真实架构时使用的非常相似，但在这里我们应用它们的方式是不同的。我们钻探得非常快，穿过若干架构级别，来研究一些更广的和 `Initialize Operations` 使命用例相关的系统关注点。我们在这里获得的模型不用作真实系统架构规格的一部分。

太常见的情况是，初始的架构决策没有被验证，因为架构团队没有想到利用这一步，或者因为开发往前推进的速度太快了。虽然情况是这样，团队经常立即分解系统用例来开发片段用例，片段用例被分派到片段架构团队。然后，片段架构团队重复该过程来定义子系统用例。最终，架构团队可能试图在整个架构级别上重新组织这些用例，以决定在每个架构级别是否每一件事能够融为一体。不幸的是，如果不是这样，就太迟了。在后面阶段的任何修正可能要困难得多，而且耗时耗钱。因此，在开发片段用例之前执行一个宏观分析是有利的。正是基于同样的理由，对片段架构团队来说，在开始分析时同样的过程也应该是必要的。

第一步是审查之前的工作结果，评价我们所站的地方，计划前方的道路。和领域专家一起，我们从功能和非功能视角评估 SNS 的逻辑架构（参见图 8-3）和 Initialize Operations 使命用例（参见图 8-5）的黑盒活动图。我们相信，我们已经正确地捕获了功能，然而还不肯定若干非功能需求。我们有在关键的系统能力上保证 SNS 功能冗余的需求。在系统架构的这个级别，我们列出片段的结构，而不是设计它们的内部架构。因此，在这里我们必须保证片段上的冗余。

为了满足这个功能冗余的需求，我们选择做出两个战略性的系统架构决策。首先，对于 SNS 地面片段的使命关键装置，我们有备份硬件。这个装置在可热切换模式下运行，同一时间内，首要和备份装置都是活动的。当出现导致首要装置不能执行任务的事件时，备份装置能够快速替换首要装置。其次，对于 SNS 发射片段，我们将使用同样的热切换装置方法来保证冗余功能。对于卫星片段，我们将利用这个事实：星座内的多个卫星是冗余的，还有在轨或准备发射的空闲卫星。这是除了每个卫星内的功能冗余之外，它的设计人员必须提供以满足 SNS 非功能需求的东西。对于用户片段，通过简单地提供整个用户片段的替代品来满足功能的冗余需求。就像卫星片段一样，这个片段的设计人员需要尽可能适当地关注内部的片段冗余。

从图 8-5 展示的 Initialize Operations 黑盒活动图开始，我们分配 SatelliteNavigation 系统分区展示的系统功能到一个或更多它的组成片段：地面、发射、卫星或用户。我们的目标是分配来自系统用例的片段用例到每个片段。这样，可以看到 SNS 功能由其片段的协作提供。如果适当地分配用例，个体片段将展示如下核心的面向对象原则。

- 抽象　相对于查看者的视角，片段提供了清晰定义的概念边界。
- 封装　片段分隔它们的子系统，子系统提供结构和行为。片段对于其他片段是黑盒。
- 模块化　片段被组织成一个内聚和松耦合的子系统的集合。
- 层次　展示一个排行或抽象次序的片段。

对 SNS 这样复杂的系统，要完成任何适当级别的细节，分析过程的这一部分可以轻易花掉几个月[1]。这是我们强烈建议首先通过尝试速成的概念证明（例如，建模、模拟或做原型），看看这部分的分析是否在正确的轨道上以验证架构决策的理由之一。架构团队不应该试图生成完整的用例列表（时间不够），但应该研究一定比例对架构有意义的用例。

[1] 但要小心分析瘫痪：如果系统分析周期需要的时间比业务的机会窗口长，那么所有跟随这条路的人就放弃希望吧，因为你最终会出局。

在浏览图 8-5 所展示的每个动作时，我们必须持续问自己很多问题。哪个或哪些片段应该负责某个动作？一个片段是否有足够知识来实施一个指向它的动作，或者它必须委托该行为?是否该片段试图做得太多？是否正在执行的动作在某种程度上并不相关？什么会出问题？这就是说，如果某个前置条件被违反，或者后置条件没有被满足，会怎么样?

这些问题和我们问自己是否在执行软件工程的问题如此相似，很有趣吧?还记得本章介绍里所说的吗？

本书之前展示的面向对象分析设计的原则和过程，以及第 5 章讨论的 UML 2.0 表示法，也适用于开发最高级别的系统架构和开发软件。

之前，我们执行一个 SNS 系统级别功能的黑盒分析。现在，我们聚焦于 SNS 内部的结构——组成这个系统的片段以及协作时每一个必须提供的功能，这样 SNS 才能满足它的需求。通过对之前表 8-1 列出的系统用例功能执行一个白盒分析，可以完成这项任务。深入 SNS，决定它如何提供所需的服务。

在继续进行之前，有些东西要注意。例如，“应该使用 Proteus-4 发射器”的约束当然会影响我们要做的功能责任分配。为了进一步贯彻这个思想，当卫星片段设计团队设计其片段的架构时，团队会被如“应该使用 Gamma II（B）卫星总线”这样的约束影响[1]。在补充材料“功能分配”中将进一步探索这一点。

功能分配

最终，我们必须转换系统需求为卫星导航系统硬件、软件和手工操作元素的需求，这样拥有不同技巧的不同组织可以并行地解决它们问题特定的部分。在这些工作期间，架构团队总是促进和维持系统的架构愿景。

在整个开发过程中，功能分配是系统架构师的关注焦点，因为分配可以在任意的抽象级别完成——从系统架构的最高到最低级别。以下列表提供了例子来阐明这个断言。

- 系统　可以分配整个卫星导航系统的功能给另一个开发组。例如，欧洲太空总署开发 Galileo 时可以追求共同开发的模式。
- 片段　片段级别分配的一个主要例子是外包整个发射工作。众多的公司提供这种类型的服务。当然，这意味着我们不会开发发射片段，而只是给分包商团队定义接口。
- 子系统[2]　可以展望 SNS 在这个级别的分配，包括在卫星片段开发中利用商业上可得的卫星总线子系统。
- 组件　这个级别位于 SNS 架构内，在这个级别，最有可能分配需求到硬件、软件或手工操作。例如，用户片段的用户界面子系统可能由两个组件组成，一个硬件（一个 LCD 屏幕）和一个软件（用于控制用户片段）。

[1] 卫星总线为机载载荷提供基础设施类型服务（如电源和通信），这些载荷提供特定的能力（例如，提供位置信息）。

[2] 这个时候讨论子系统和组件级别的 SNS 架构，可以使我们对未来的样子稍有了解。

之前，表 8-1 中列出了 8 个 SNS 系统用例，我们从图 8-5 SatelliteNavigation 系统分区中的动作开发出这些用例。如果是实际开发系统架构，而不是执行一个快速浏览分析，我们会为这些用例中的每一个开发各自的活动图，并在片段之间分配活动图动作。这样，它们可以展示出我们之前讨论的 4 个核心面向对象原则。但是，在这个宏观级别的分析中，为了有一个更宽的功能视角，我们在一张活动图中分析所有 8 个系统用例。这容易完成，因为我们没有打算在细节上挖得太深。我们关心的是片段上的用例功能分配比例。

有了手上的 4 个逻辑 SNS 片段（参见图 8-3）的实现，我们和领域专家一起开始我们的工作，分配动作中表示的功能。如果此时没有一个系统架构的表示法，我们可以分配功能到分区，起个通用的名字，如 SegmentOne、SegmentTwo 和 SegmentThree。在每个分区中分配动作，这样，每个片段在和其他片段协作以提供卫星导航系统功能（在这里是初始化操作）时，被定义为提供紧密相关的能力的专家。这个分配会在分析多个用例场景期间继续，以建立更完整的片段功能画面，所以支持为每个片段选择有意义的名称。

正如第 3 章中讨论的，这个方法和我们要设计良好的类的方法类似。在那一章里，我们说可以测量一个抽象的质量，并建议了 5 个度量，其中两个（耦合和内聚）是卫星导航系统架构关键抽象（即它的片段）的中心关注点。我们指定 SNS 片段是松耦合的。我们要它们“独立”，只有必需的最小数目的连接，来支持它们的协作以提供 SNS 功能。

内聚是另外的测量，通过它可以判断我们所选择的抽象的质量。内聚测量组成单个片段的元素之间的连接程度。最不受欢迎的内聚形式是偶然内聚——完全不相关的抽象被扔进一个 SNS 片段中，最受欢迎的内聚形式是功能内聚——片段的所有元素一起工作来提供一些有良好边界的行为。

通过浏览图 8-5 中所示的动作，可以看到如何到达图 8-7 所示的白盒活动图。Launch Satellite 系统用例包含两个动作：Prepare for Launch 和 Launch。相当明显，GroundSegment 和 LaunchSegment 应该提供这个能力。GroundSegment 需要执行准备来发射，也命令 LaunchSegment 准备。准备完成之后，Operator 命令 GroundSegment 发射，然后 GroundSegment 命令 LaunchSegment 发射。从 SNS 系统用例 Launch Satellite，我们获得了 GroundSegment 和 LaunchSegment 的若干动作。针对 Initialize Operations 包剩下的系统用例继续这个分析过程，来开发完整的活动图，如图 8-7 所示。

Initialize Operations 的白盒活动图只展示了 OperateSNS 使命用例包内包含的一部分功能的分析结果。剩下的都是通向这一点以及所有后续发生的活动的预备活动，包含在其他 3 个使命用例中：Provide Normal Operations、Provide Special Operations 和 Terminate Operations。但是，这些不是我们这次宏观分析的焦点。如果它们是，我们会重复我们的分析技能来阐述这个行为，提供一个关于片段如何共同操作来提供卫星导航系统的操作能力的更完整画面。

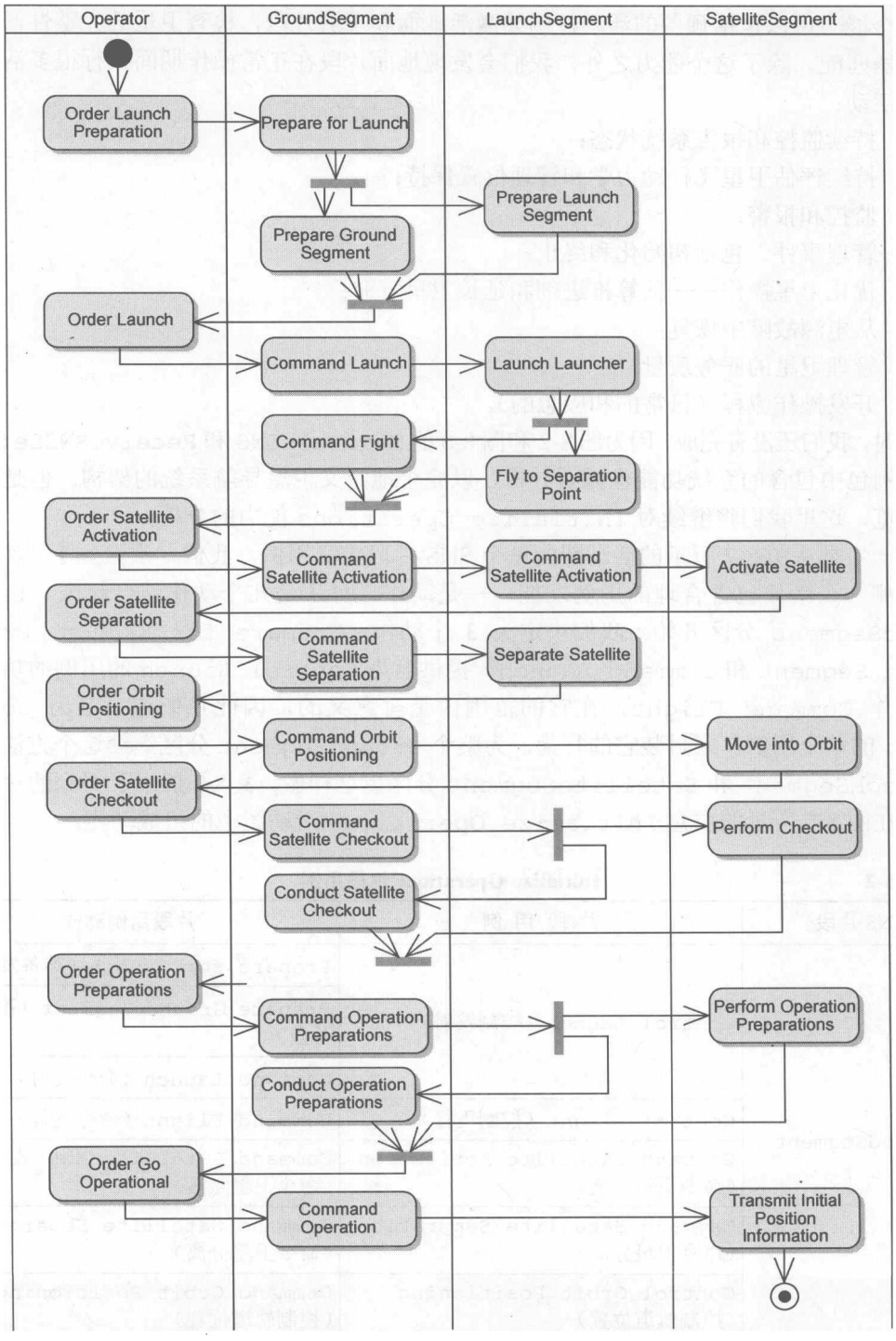


图 8-7 Initialize Operations 白盒活动图

这个能力可以包括预备的活动，例如激活地面和发射片段，检查卫星的完整性，把卫星和发射器匹配。除了这个能力之外，我们会发现地面片段在正常操作期间执行很多活动，包括以下活动：

- 持续监控和报告系统状态；
- 持续评估卫星飞行动力学和管理位置保持；
- 监控和报警；
- 管理事件，包括初始化和终止；
- 优化卫星操作——估算推进剂和延长卫星寿命；
- 从电源故障中恢复；
- 管理卫星的服务质量；
- 开发操作流程（日常的和应急的）。

此时，我们还没有完成，因为图 8-2 和图 8-6 的 MaintainSNS 和 ReceiveSNSServices 使命用例包中包含的系统功能也需要分析，以完全地定义卫星导航系统的架构。但是，为了快速浏览，这里我们将继续对 Initialize Operations 能力的分析。

下一步是从图 8-7 所示的活动图为每个 SNS 片段定义用例。我们一次聚焦于一个分区，并决定哪一个动作包含合理的用例功能——是动作自身还是几个动作结合起来。让我们从 GroundSegment 分区开始。我们决定头 3 个动作（Prepare for Launch、Prepare Ground Segment 和 Command Launch）提供名为 Control Launch 的用例的功能。下一个动作 Command Flight，在它的范围内是有意义的，因此我们定义名为 Control Flight 的单个用例，来封装它的行为。为整个 GroundSegment 分区继续这个方法，然后在 LaunchSegment 和 SatelliteSegment 分区也这样做。表 8-2 展示了得到的片段用例和它们在图 8-7 所示的白盒 Initialize Operations 活动图中的组成动作。

表 8-2　　Initialize Operations 片段用例

SNS 片段	片 段 用 例	片段用例动作
GroundSegment	Control Launch（控制发射）	Prepare for Launch（准备发射）
		Prepare Ground Segment（准备地面片段）
		Command Launch（命令发射）
	Control Flight（控制飞行）	Command Flight（命令飞行）
	Command Satellite Activation（命令卫星激活）	Command Satellite Activation（命令卫星激活）
	Command Satellite Separation（命令卫星分离）	Command Satellite Separation（命令卫星分离）
	Control Orbit Positioning（控制轨道位置）	Command Orbit Positioning（控制轨道位置）

续表

SNS 片段	片 段 用 例	片段用例动作
GroundSegment	Command Satellite Checkout (命令卫星检测)	Command Satellite Checkout (命令卫星检测)
		Conduct Satellite Checkout (命令卫星检测)
	Conduct Operation Preparations (实施操作准备)	Command Operation Preparations (命令操作准备)
		Conduct Operation Preparations (命令操作准备)
	Command Operation (命令操作)	Command Operation (命令操作)
LaunchSegment	Launch (发射)	Prepare Launch Segment (准备发射片段)
		Launch Launcher (发射发射器)
	Fly to Separation Point (飞到分离点)	Fly to Separation Point (飞到分离点)
	Command Satellite Activation (命令卫星激活)	Command.Satellite Activation (命令卫星激活)
	Separate Satellite (分离卫星)	Separate Satellite (分离卫星)
SatelliteSegment	Activate Satellite (激活卫星)	Activate Satellite (激活卫星)
	Maneuver to Orbit (机动到轨道)	Move into Orbit (移进轨道)
	Prepare for Operations (准备操作)	Perform Checkout (准备检测)
		Perform Operation Preparations (执行操作准备)
	Transmit Initial Position Information (传送初始位置信息)	Transmit Initial Position Information (传送初始位置信息)

8.2.4 分配非功能需求和阐明接口

在分析 Initialize Operations 功能期间，我们接收了一条附加的卫星导航系统非功能需求："从发射准备开始到卫星传输导航信息开始的时间应该少于 7 天。"为什么会有这样一条需求？也许我们的顾客不使用以在轨的备用卫星替换失灵的卫星的方法[1]，而更喜欢发射一颗代替的卫星，并让它在一个星期内运作。我们面对的任务是，在表 8-2 所示的用例以及任何附加用例之间分摊 7 天（168 小时）时间。例如，把卫星匹配到发射器，这样整个时间线跨度将少于 168 小时，正如我们的顾客所说的那样。

这个时候，一个合理的问题是："我们如何做这个？"这里有两个首要的问题——分配非功能需求和归档结果。分配适当比例的性能需求（168 小时）到每个片段用例非常依赖于

[1] 替换失灵的卫星可以通过使用之前发射到太空作为备用的卫星来实现。备用卫星待在太空中，就像一个坐在板凳上的体育运动员，等待着替补受伤的运动员。

领域专家的经验。除了使用领域专家和开发团队的经验之外，我们也使用其他技能，如模拟来决定备选分配方案的影响。对于我们的例子，168 小时中的 48 小时已经被分配到表 8-2 中展示的 Initialize Operations 片段用例。剩下的 120 小时被分配到所有预备活动，包括激活地面片段、激活发射片段、检查卫星完整性以及把卫星和发射器匹配。

第二个问题——如何归档结果，很大程度上依赖于团队使用的需求和可视化建模工具，当然，也依赖于开发过程。很多工具确实提供方法来归档结果，但信息通常在一个需求数据库中引用到可视化模型里的活动，或者被埋藏在活动属性框的 tab 下面。运行报表和执行统计分析是有用的，虽然它不提供我们喜欢的可视化表现，特别是在开发工作的这个级别。这里，我们特地选择用一张表（见表 8-3）来清晰地展示在片段用例之间分配 48 小时的结果。同样，这些技能也用于在所有最终阐述的片段用例之间分配其他非功能需求。

表 8-3　　Initialize Operationsa 发射时间分配

SNS 片段	片 段 用 例	已分配时间（小时:分）
GroundSegment	Control Launch	11:22
	Control Flight	0:17
	Command Satellite Activation	0:01
	Command Satellite Separation	0:01
	Control Orbit Positioning	0:05
	Command Satellite Checkout	16:30
	Conduct Operation Preparations	4:30
	Command Operation	0:01
LaunchSegment	Launch	11:30
	Fly to Separation Point	0:17
	Command Satellite Activation	0:01
	Separate Satellite	0:04
SatelliteSegment	Activate Satellite	0:03
	Maneuver to Orbit	13:45
	Prepare for Operations	21:29
	Transmit Initial Position Information	0:03[b]

a. 如果有真实世界中这些活动的经验，请原谅我们粗暴的分配。虽然所分配时间累加起来大约 80 小时，但实际花费的计时时间应该在 48 小时之内。这可能是因为很多动作并行的原因，如图 8-7 所示。

b. 这些 3 分钟表示一旦下命令后卫星系统开始传输位置信息所需时间。同样，40 分钟（48 小时中的）已经被分配给图 8-7 的 Operator 分区中展示的 8 个活动。

分配到片段用例的非功能需求处于架构层次中下一个更低的级别，使用和在片段级别同样的技能，在组成它的子系统用例之间分配。分配功能和非功能需求的技能可以从架构层次的一个级别到下一个级别递归地应用——从系统到片段，到它们的子系统，等等。

然后，可以询问由所给的设计约束间接引出的潜在需求：

- 兼容国际标准；
- 最大化使用 COTS 硬件和软件。

正如之前所讨论的，约束“兼容国际标准”驱动外部执行者 Atmosphere/Space 的规格。我们必须和规范无线电波使用的国家和国际机构打交道，来决定诸如可以与卫星片段通信的特定频率、传输位置信息的频率等。这意味着现在地面片段、发射片段（至少在飞行阶段期间）和卫星片段必须履行卫星导航系统的外部接口责任。指出这些问题是因为我们聚焦于功能能力，约束（和非功能需求）可能会在开发周期很晚的时候才被考虑，有时甚至被忽略。

外部接口的主题是还没有触及的另一个点，因为我们的焦点还是通过分析卫星导航系统的功能开发它的逻辑架构。开发和归档接口规格的技能应该类似于任何类型的系统或软件开发。卫星导航系统和图 8-1 所示的那些执行者——User、Operator、Maintainer、ExternalPower、ExternalCommunications 和 Atmosphere/Space 有接口。显然，在着手确定 User、Operator 和 Maintainer 执行者接口的规格之前，必须执行一些级别的功能分析。此外，人/机接口专家将是这项任务中关键的团队成员。与 ExternalPower 和 ExternalCommunications 执行者的接口可以很早就指定，因为标准已经说明了电源和通信如何供应。最后的外部接口，即与 Atmosphere/Space 执行者的接口，大部分由国家和国际上的行政管理机构通过管制卫星传输的规定和条约来指定。

8.2.5 规定系统架构及其部署

之前在图 8-3 中展示的 SNS 逻辑架构表示法已经经受了行为原型工作的测试。因此，图 8-3 代表第一个级别的 SNS 架构逻辑视图，就像图 8-8 所示的组件图一样。在 UML 2.0 中，组件图可以用于层层分解一个系统，所以在这里它代表卫星导航系统架构最高级别的抽象，即它的片段和它们的关系。图 8-8 阐明了表达片段之间接口的两种方法：球-球窝表示法（LaunchSupport 接口）和连接需求和供给接口的虚依赖线（PositionInformation 接口）。

回顾图 8-1，可以看到图 8-8 中所示的 SNS 接口没有考虑到三个系统执行者：ExternalPower、ExternalCommunications 和 Atmosphere/Space。这些执行者提供重要的服务给卫星导航系统，但它们不是在开发其逻辑架构时我们关注的焦点中心。

图 8-9 展示了图 8-8 表现的组件部署到系统架构节点的情况。这些组件是 SNS 的片段，表现为 UML 2.0 工件。这不是一个典型的表示法用法。通常，我们在处理节点上部署软件工件（如代码、源文件、文档或另一个和代码相关的东西）。但是，这张图清楚地表现了信息以及为系统工程使用 UML 2.0 表示法时的一些非标准的用法。Operator、Maintainer 和 User 执行者与卫星导航系统交互的接口被包含进它的片段，所以我们选择阐明这些依赖关系。

之前，我们决定提供功能冗余的方法是在 GroundSegment 和 LaunchSegment 以热切换模式为使命关键装置运行备份，即首要装置和备份装置同时是活跃的。开发 SNS 架构时，

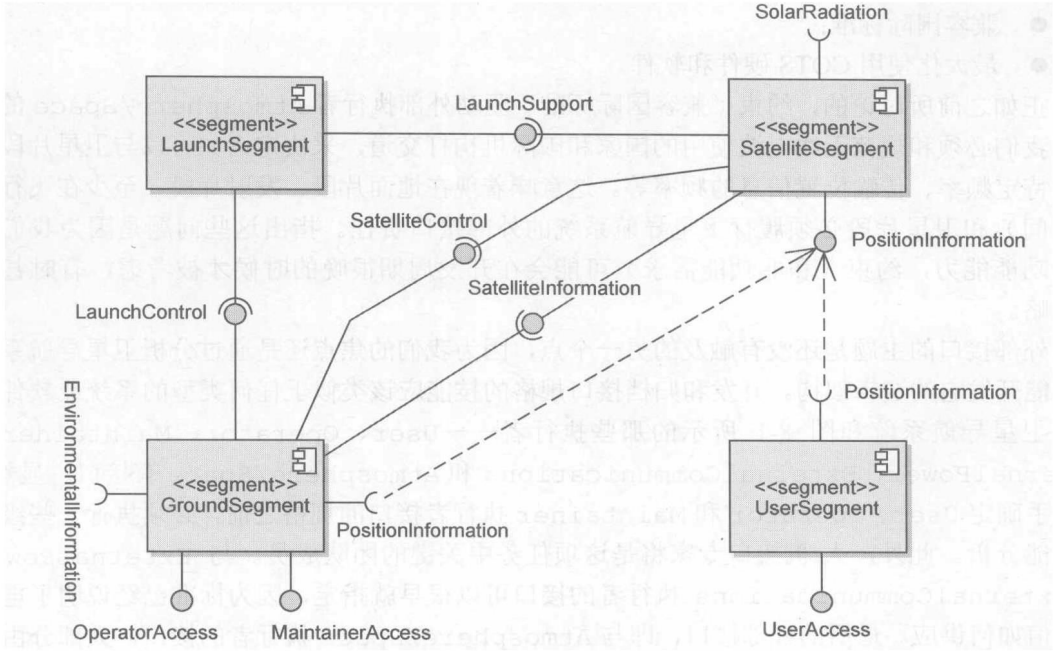

SolarRadiation
<>
LaunchSegment
LaunchSupport
<>
SatelliteSegment
SatelliteControl
PositionInformation
SatelliteInformation
LaunchControl
PositionInformation
EnvironmentalInformation
<>
GroundSegment
PositionInformation
<>
UserSegment
OperatorAccess
MaintainerAccess
UserAccess

图 8-8 卫星导航系统组件图

我们认识到满足功能冗余需求的更好方法是在两个地理上分散的站点上分布 GroundSegment，对于 LaunchSegment 也是如此。例如，这样可以避免因为自然灾害而完全丢失片段功能。这个设计决策通过 GroundSite 和 LaunchSite 节点之间多重性为 2 的通信关联表示。类似于原来建议的备份装置，我们现在有了备份站点，它随时待命，准备扮演首要站点的角色。

SNS 设计需要解释的另一个方面是 Satellite Constellation 节点和驻留在它上面的 SatelliteSegment 工件。在提供位置信息给 UserSegment 工件时，SatelliteConstellation 节点本质上是卫星导航系统的卫星和它们在太空中的位置的集合。SatelliteConstellation 节点给驻留在它上面的 SatelliteSegment 工件提供支持，如引力（一个我们忽略了的外部系统执行者?），以帮助保持卫星位于它们的适当轨道，以及帮助大气和外层空间为卫星提供通信介质。SatelliteSegment 工件上的多重性 {1..*}表示该星座至少有一个卫星。我们的顾客还没有决定卫星导航系统的覆盖面积[1]。当这个问题解决，我们将决定必需卫星的确切数目，来为 SNS 用户提供“高效和买得起的卫星导航系统服务”。

[1] 这种未作的决策肯定是程序风险(技术的和非技术的)的主要来源。为了帮助顾客敲定这个关键的需求，可以用模拟来决定覆盖所需的最优卫星数目。对这个主题感兴趣的读者，Aerospace Corporation 的 *Crosslink*（Summer 2002）（www.aero.org/publications/crosslink/summer2002/index.html）里有两篇密切相关的文章：*Orbit Determination and Satellite Navigation* 和 *Optimizing Performance Through Constellation Management*。

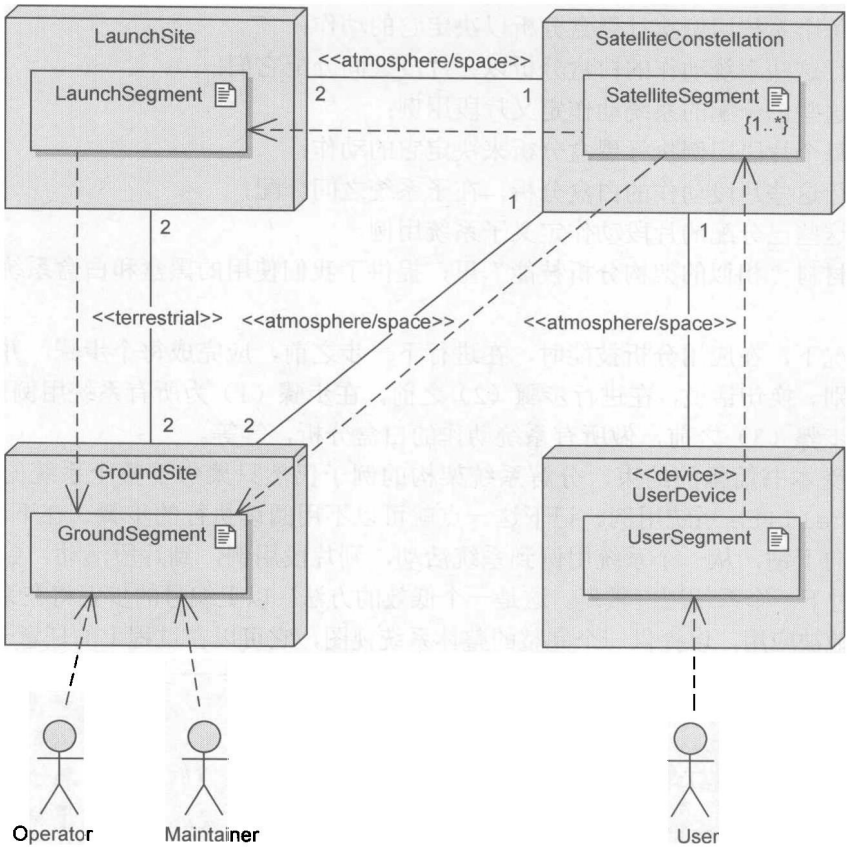


图 8-9 部署 SNS 片段

8.2.6 分解系统架构

对于 Initialize Operations 系统用例，现在已经验证了围绕卫星导航系统架构的假设和决策，可以进行它的片段以及它们的子系统的规格的工作。如果面临任何问题，我们早已按需要修改架构。在继续进行之前，对于我们的宏观分析结果，必须强调一个关键点——行为原型必须被抛弃，它已经完成了它想要达到的目标。同样的道理，原型代码不是可交付软件的基础，行为原型也不是卫星导航系统架构的基础。

之前图 8-3 所示的 SNS 逻辑架构图很有用，但不完整，因为这张图中的每个片段太大了，不能由小的开发人员团队开发。必须缩小每个片段，进一步分解它们为嵌套的子系统。这可通过应用同样的分析技能完成，这些技能曾经用于为 Initialize Operations 功能做卫星导航系统架构片段的原型（如图 8-8 中所示组件图描绘的），但这一次应用得更完整。这些技能在卫星导航系统所有抽象级别重复（从系统到片段，到它们的子系统，等等）为系统架构每一个级别的每个元素来决定用例。在我们做这件事情时，非功能需求在用例之间被分配到系统分解的每一个级别的每一个元素。为完整起见，我们的分析技能展示如下：

（1）为每个系统用例执行黑盒分析以决定它的动作；
（2）执行这些系统动作的白盒分析以在片段之间分配它们；
（3）从这些已分配的系统动作定义片段用例；
（4）为每个片段用例执行黑盒分析来决定它的动作；
（5）执行这些片段动作的白盒分析，在子系统之间分配；
（6）从这些已分配的片段动作定义子系统用例。

在补充材料“相似的架构分析技能”里，提供了我们使用的黑盒和白盒系统分析方法的视角。

正常情况下，在应用分析技能时，在进行下一步之前，应完成每个步骤，并遍及系统的整个架构级别。换句话说，在进行步骤（2）之前，在步骤（1）为所有系统用例做黑盒分析；然后在进行步骤（3）之前，做所有系统动作的白盒分析，等等。

但是，受本书篇幅的约束，分解系统架构的例子仍然只聚焦于整个系统的一部分——Launch Satellite 系统用例。记下这一点就可以不用阅读执行的步骤（在下面列出），并假设垂直地向下钻，从一个系统用例到系统活动，到片段用例，到片段活动，到子系统，等等。然后，对下一个系统用例重复。这是一个低效的方法。以上编号的步骤将在系统的每个架构级别水平地被应用，以提供一个完整的整体系统视图，它可以在过程上的任意一点被验证。

相似的架构分析技能

很多年来，系统工程师已经使用了和我们的描述非常相似的技能，来分析系统功能和分配功能到系统架构元素。这些黑盒和白盒分析技能已经被成功用于开发复杂系统的架构层次，如全球定位系统。

在 Jacobson 等人所著的 *The Object Advantage* 中，通过上级和下级用例的概念，展示了用例和它们在分析企业系统中的应用[12]。IBM Rational Unified Process for Systems Engineering（RUP SE）通过对 RUP 作系统工程的扩展，在本质上结合了这些方法。系统工程师已经高效地使用用例和黑盒/白盒分析概念很多年。

我们通过执行以下活动继续我们的分析（不在这里展示）。

- 为 Launch Satellite 系统用例执行黑盒分析来决定它的动作；
- 执行这些系统动作的白盒分析，以在片段之间分配它们；
- 从这些已分配的系统动作定义 GroundSegment 用例；
- 为 GroundSegment 的 Control Launch 用例执行黑盒分析来决定它的动作；
- 执行 GroundSegment 的 Control Launch 动作的白盒分析，以在它的子系统之间分配它们。

图 8-10 展示了这个分析的结果。我们看到，在它们协作来提供 GroundSegment 控制发射的功能时，每个 GroundSegment 子系统必须执行的动作。通过这样的分析，我们可以开发每个卫星导航系统片段的架构。下面将展示得到的片段架构。

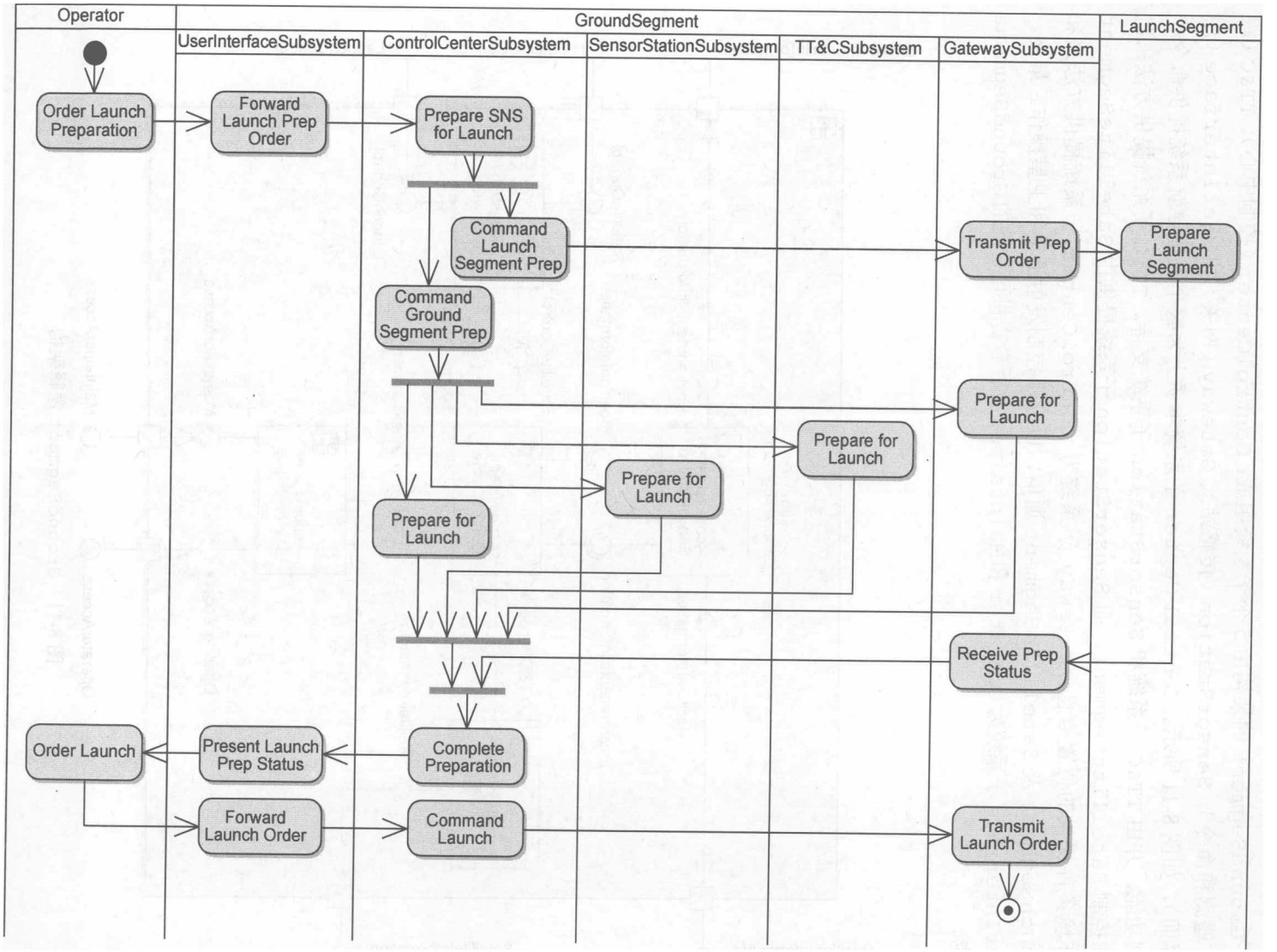


图 8-10 Control Launch 白盒活动图

GroundSegment 的架构由 5 个子系统组成：ControlCenter（控制中心）、TT&C（跟踪、遥测和命令），SensorStation（传感站）、Gateway（网关）和 UserInterface（用户界面），如图 8-11 所示。Control Center 子系统本质上为整个卫星导航系统提供命令和控制功能，并由 TT&C 子系统和 SensorStation 子系统支持。TT&C 子系统提供方法来监测和控制 SatelliteSegment，而 SensorStation 子系统提供由 SatelliteSegment 和环境条件提供的位置信息。Gateway 子系统为 ControlCenter 子系统提供方法来和 LaunchSegment 及 SatelliteSegment 通信，以分别控制发射活动和卫星操作。最终，UserInterface 子系统为 Operator 和 Maintainer 执行者提供可访问的 GroundSegment 功能。

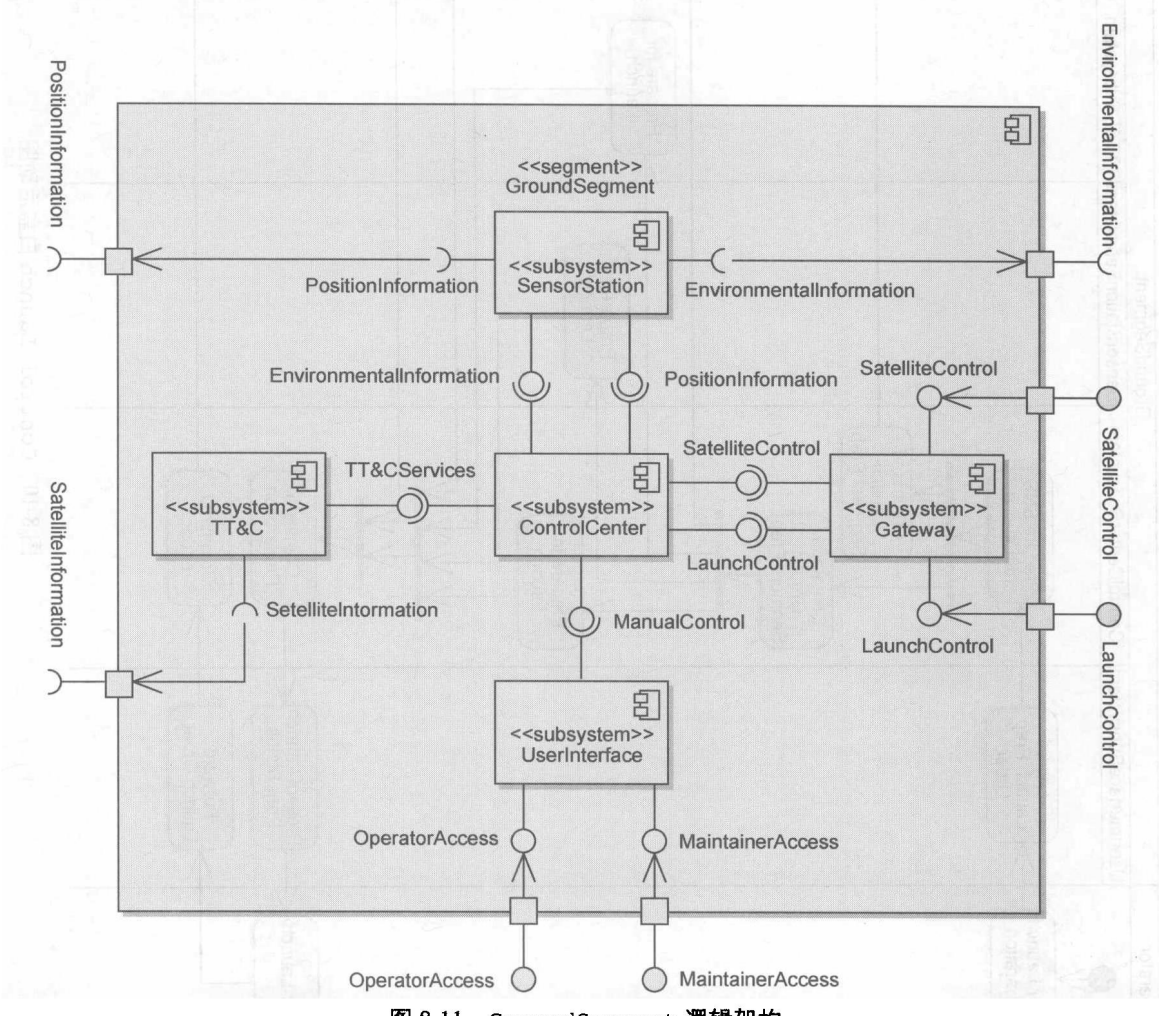

PositionInformation
EnvironmentalInformation
<>
GroundSegment
<<subsystem>>
SensorStation
PositionInformation
EnvironmentalInformation
EnvironmentalInformation
PositionInformation
SatelliteControl
SatelliteControl
TT&CServices
<<subsystem>>
TT&C
<<subsystem>>
ControlCenter
<<subsystem>>
Gateway
LaunchControl
SatelliteInformation
SetelliteIntormation
ManualControl
LaunchControl
LaunchControl
<<subsystem>>
UserInterface
OperatorAccess
MaintainerAccess
OperatorAccess
MaintainerAccess

图 8-11 GroundSegment 逻辑架构

图 8-12 展示了 LaunchSegment 的逻辑架构，由 3 个子系统组成：LaunchCenter（发射中心）、Launcher（发射器）和 Gateway（网关）。LaunchCenter 子系统为 LaunchSegment 提供命令和控制功能，和 GroundSegment 的 ControlCenter 子系统提供的类似。Launcher 子系统提供所有必需的能力来把 SatelliteSegment 放进它的初始轨道。在这里，Gateway 子系统就像在 GroundSegment 里一样，使 LaunchCenter 接收来自 GroundSegment 的发射控制支持，并提供发射支持给 Launcher。

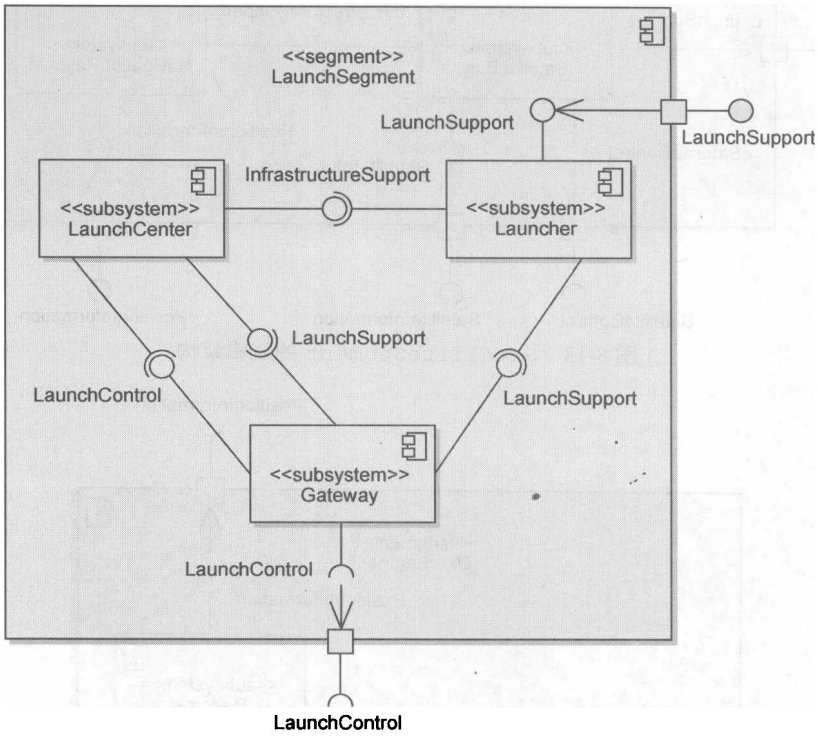

<>
LaunchSegment
LaunchSupport
LaunchSupport
InfrastructureSupport
<<subsystem>>
LaunchCenter
<<subsystem>>
Launcher
LaunchSupport
LaunchControl
LaunchSupport
<<subsystem>>
Gateway
LaunchControl
LaunchControl

图 8-12 LaunchSegment 的逻辑架构

SatelliteSegment 被分解成两个子系统，如图 8-13 所示。SatelliteBus 子系统为 NavigationPayload 子系统提供基础设施支持。在它的本体结构上，SatelliteBus 驻留了提供电源、姿态控制和推进的装置，来命名一些服务。这个装置使得 NavigationPayload 装置（包括一个高精度时钟和位置信号生成）提供位置信息给卫星导航系统用户成为可能。

UserSegment 也自然地被分解为若干子系统，如图 8-14 所示。Receiver 子系统从 SatelliteSegment 接收位置信息，并将其作为位置数据提供给 Processor 子系统；Processor 子系统把它转换为导航信息，由 UserInterface 子系统使用。UserInterface 为 User 提供方法，通过多种特化的用户界面（如按钮、触摸屏和声音警告）来访问和使用 UserSegment 导航服务。

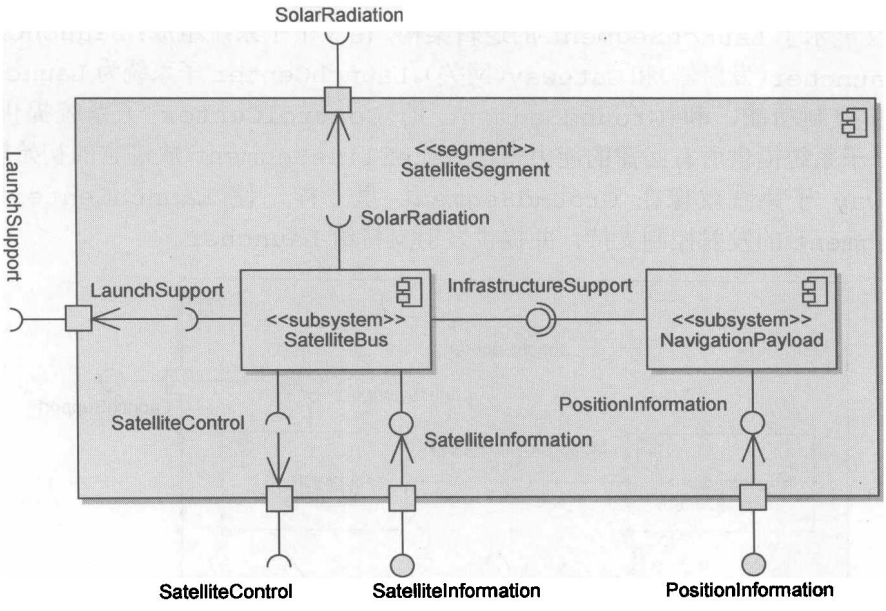

SolarRadiation
<>
SatelliteSegment
SolarRadiation
LaunchSupport
LaunchSupport
<<subsystem>>
SatelliteBus
InfrastructureSupport
<<subsystem>>
NavigationPayload
PositionInformation
SatelliteControl
SatelliteInformation
SatelliteControl
SatelliteInformation
PositionInformation

图 8-13 SatelliteSegment 的逻辑架构

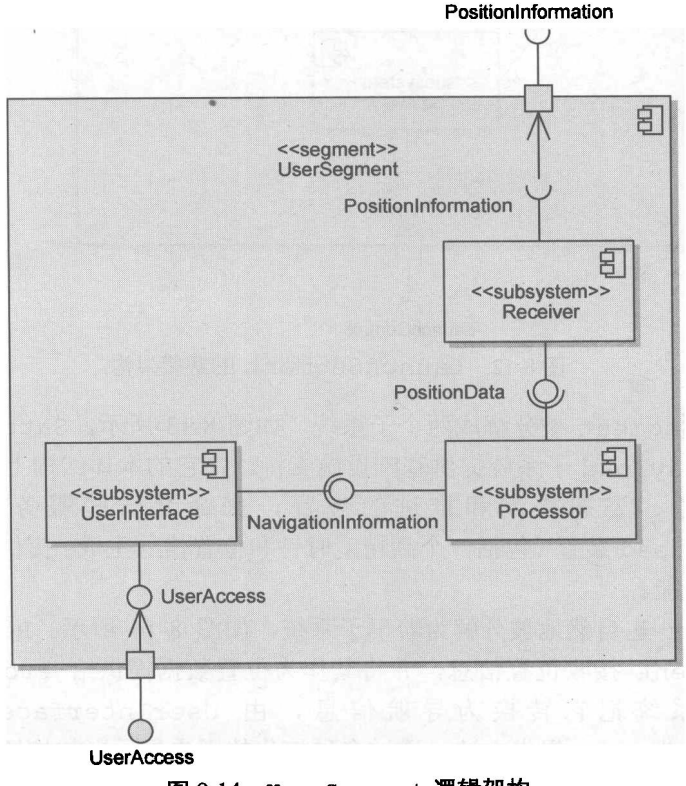

PositionInformation
<>
UserSegment
PositionInformation
<<subsystem>>
Receiver
PositionData
<<subsystem>>
UserInterface
NavigationInformation
<<subsystem>>
Processor
UserAccess
UserAccess

图 8-14 UserSegment 逻辑架构

这个设计带给我们 4 个顶级片段，每个包含若干子系统。我们已经分配系统功能给这些子系统，这些功能可以结合硬件、软件和手工操作来提供。在一些情况下，有经验的系统架构师很清楚这些分配。

正如在第 7 章中讨论的，这些片段和它们的子系统形成了工作分配单元，以及粗粒度的配置管理和版本控制单元。每个片段或子系统应该由一个组织、团队或人所拥有，但可能有更多人实现。所有者指导元素的详细设计和实现，并管理它相对于同一抽象级别其他元素的接口。因此，通过把非常复杂的问题依次分解为更小的单元，管理非常大的开发程序成为可能。

现在本章的目标已达到——展示了面向对象分析设计原则和过程以及 UML 2.0 表示法。它们可以被应用于开发最高级别的系统架构，也可以被应用于开发软件。

8.3 构造

正如在第 6 章指出的，在细化阶段的最后，应该开发系统的一个稳定架构。任何对系统架构的修改，作为构造阶段的活动的结果，可能限制于那些更低级别的架构元素，而不是我们关注过的卫星导航系统片段和子系统。依照本章的本意——展示开发 SNS 系统架构的方法，即通过逻辑上分区所需功能来定义组成片段和子系统，所以，在这个阶段，我们不展示任何架构开发活动。

8.4 后移交

卫星导航系统的原始非功能需求中，有两条导致我们要开发一个弹性的架构：可扩展性和长的服务寿命。除了很多其他方面之外，这个长的服务寿命限定了一个可扩展的设计以保证可靠地供给所要的功能。当有更多卫星导航系统用户以及当我们适配这个设计到新的实现时，它们将为已有的机制揭示新的、不曾预期的用法，强迫添加新的功能到系统。现在来研究，当我们添加新的功能和改变系统的目标硬件时，SNS 设计能多好地满足这些需求。

8.4.1 添加新的功能

让我们考虑附加一个需求，即使用传输自其他系统的位置信息的能力，如 GPS、GLONASS 和 Galileo。这会大大增加我们的系统在全世界的定位能力的有效性和精度。幸运的是，对卫星导航系统作最小的变更就可以容纳，因为它的影响和用户片段隔离。同样幸运的是这样一个事实：通过改变 `Receiver` 子系统和升级 `Processor` 子系统中的固件，已有的用户片段可以容易地升级来提供这个能力。因此，添加这个功能对已有的设计影响很小。这实际上在结构良好的面向对象系统是十分常见的：通过在已有机制上建造新的应用，可以

相当容易地应付系统重要的附加需求。

更激进的改变如何呢？假设顾客要引进支持搜索和救援（SAR）任务的能力，用我们的卫星导航系统接收遇险信标[1]。这个新的需求会如何影响我们的架构？分析之后，我们看到最大的影响是针对卫星片段，但对地面片段也有些影响。这不出乎意料。如果提前思考这个需求，我们会添加接收这些特定信号（也许一个信号范围）的能力，不会影响卫星片段设计，只影响使用功能的操作。否则，我们就需要给卫星片段添加一个提供这个能力的附加子系统。对于地面片段，影响只是能够中继接收遇险信标的信息到适当的管理机构，这可能会涉及对 ControlCenter 和 TT&C 子系统进行较小的软件或操作修改。

再说一次，这个大的变更将对总体 SNS 架构有最小的影响。不幸的是，对空基资产作出修改的困难，将强迫系统架构师非常深远地考虑系统的愿景。但即使在最坏的情况下，我们也能够在对已有的架构或功能影响最小的情况下，添加 SAR 能力到将来开发的卫星片段元素。

8.4.2 改变目标硬件

硬件技术依然以比生成软件的速度快得多的脚步往前走。此外，可能会有很多政治和历史原因导致我们在开发过程早期做出某些硬件和软件选择，而之后则可能会后悔[2]。出于这个原因，大系统的目标硬件比起软件来非常容易过时。

例如，在若干年的操作使用后，我们可能决定需要替换地面片段的整个 ControlCenter 子系统。这会如何影响我们已有的架构？如果在系统演进期间，我们保持子系统接口位于高的抽象级别，这个硬件变更对我们的软件影响最小。既然选择封装关于 ControlCenter 子系统具体情况的所有设计决策，没有其他子系统曾经被定义来依赖给定工作站的特定特征，例如，子系统封装所有这样的硬件秘密。这意味着在 ControlCenter 子系统中，工作站的行为是被隐藏的。因此，这个子系统扮演了一个抽象的防火墙，把所有其他客户从特定计算技术的复杂性屏蔽开来。

类似地，针对电信标准的激进变更也会影响我们的实现，但是这种影响是有限的。具体地说，我们的设计保证只有 Gateway 子系统知道网络通信。因此，即使变更发生在网络的基础，也绝不会影响任何高层次客户。Gateway 子系统把它们和真实世界的反复无常屏蔽开来。

我们所引进的变更都没有割裂已有架构的机理，这确实是架构良好的面向对象系统的终极标志。

---

[1] Galileo 程序从一开始就添加这种类型的能力，以辅助 Cospas-Sarsat 程序（www.cospas-sarsat.org/）达成在全世界支持 SAR 任务的使命。事实上，Galileo 程序相信它对这个工作的贡献会提供接近实时地在若干米内获取遇险信标和位置。

[2] 例如，我们的项目可能已经从一家第三方厂商选择一个特定硬件或软件产品，只是后来发现该产品没有存活到它所承诺的时间。甚至更糟，可能发现一个关键产品的唯一供应商倒闭了。对于这样的情况，项目经理通常作两个选择之一：推卸责任，或者选择另一个产品，希望系统的架构有足够弹性来容纳变更。虽然有时实施前一个选择已然非常令人满意，但使用面向对象分析设计可以帮助我们达成后一个选择。

控制系统——交通管理

软件开发经济学已经发展到这样的水平：比以往可实现更多种类的应用自动化——从控制汽车各种功能的嵌入式微计算机，到可大量削减苦工的产生动画电影的工具，到管理成百万消费者的交互式视频服务分布式系统。这些大型系统的显著特征是它们的极端复杂性。建造小规模实现的系统是可取的，但事实告诉我们，某些大型问题还是需要大规模的实现。对一些大型应用来说，经常需要由大型软件开发组织来完成开发，这些大型软件开发组织雇佣好几百名程序员，通过程序员之间的协作产生上百万行的代码以应付开发期间肯定会不断变化的需求。这些项目通常不只包含一个程序的开发。它们通常包含多个相互协作的程序，这些程序必须执行在一个包括多个相互之间以多种方式互连的计算机的分布式目标系统中。为降低开发风险，这样的项目通常需要有中心组织负责系统的架构和集成，其他工作可以转包给其他公司或其他内部组织完成。因此，整个开发团队并不是在一起的，通常在时间和空间上是分散的，这是由于在大型项目中经常有人员轮换。

满足于编写小型、单机、单用户和基于 Windows 的工具的开发人员，可能会发现与建立大量错综复杂应用相关联的问题如此之多，以至于他们认为即使尝试一下也是愚蠢的。然而，商业和科学世界的现状决定了建立复杂软件系统的必要性。实际上，在某些情况下，不去尝试一下是愚蠢的。难以想像使用手工系统来控制环绕大都市的空中交通、管理载人飞船的生命支撑系统或跨国银行的记账活动。要成功地将这些系统自动化，不仅要解决手头一些非常实际的问题，还会引出诸如低操作成本、高安全性、不断增加功能等一系列有形无形的利益问题。当然，最重要的是取得成功。建造复杂系统本身就是一项艰巨的任务，需要应用我们所知的最佳工程实践以及少数伟大设计者创造性的洞察力。

本章处理针对这样一个问题的开发。

9.1 先启

对大多数生活在美国的人来说，火车是很原始的代步工具。但在欧洲和亚洲大部分地区却完全相反，火车是交通网络的重要组成部分，每天，数万公里长的铁轨在城市内部以及跨越国界运送货物和旅客。公平地说，现在火车在美国国内仍是重要的、经济的货物运输方式。另外，随着大都市变得越来越拥挤，轻轨运输逐渐成为缓解交通拥挤和解决内燃发动机带来的污染问题的有吸引力的选择。

铁路依然还是一种行业，所以肯定是有利可图的。铁路公司必须巧妙地平衡省钱与安全的要求和提高运输效率与遵守列车时刻的压力。这些冲突需要列车交通管理的自动化解决方案，包括计算机化列车路由以及对列车系统所有元素进行监控。这样的自动化和半自动化列车系统如今在瑞士、英国、德国西部、法国、日本[1]、加拿大和美国已经存在。这些系统多是出于经济及社会方面的考虑：降低运营成本和更有效地利用资源，将提升的安全性作为副产品整合进来。

本节将通过阐述虚构的列车交通管理系统（TTMS）的需求，以及进一步描述所需功能的系统用例来开始我们的分析。

9.1.1 列车交通管理系统的需求

根据以往开发大型系统的经验，最初的需求陈述绝对不会是完整的，通常是含糊的，而且总是自相矛盾。出于这些原因，我们必须有意识地关注开发期间对不确定性的管理。因此，我们强烈建议特意允许这样一个系统的开发以增量和迭代的方式随着时间演化。正如第6章中所指出的，能使用户和开发人员更好地洞察到需求的恰当开发过程真的很重要——远远比没有现成的实现或原型而只是在纸上描述需求要好得多。另外，由于开发大型系统软件可能需要好几年，因此软件需求应该允许改变，以利用快速改变的硬件技术。[1]精制一个面向过时的硬件技术的优雅软件架构肯定是无益的。这就是为什么我们建议无论采用何种机制作为软件架构的一部分，都应该依据现行通信标准、图形标准、网络标准和传感器标准。对于真正领先的系统，有时率先使用新的硬件或软件技术是必要的，然而这会增加项目的风险，而这种项目通常已经包含很高的风险。在成功部署大型自动化应用的过程中，显然软件开发仍是具有最高技术风险的环节。我们的目标是将风险限制在可管理的级别，而不是进一步增加风险。

这是一个非常大、高度复杂的系统，在现实世界中不能用简单的需求来阐述。然而，对于本章，以下的需求足够用来做分析和设计工作。在真实世界中，像这样的一个问题容易遭受分析瘫痪，因为有数千条需求（有功能的、非功能的），还有上万条约束。显然，需要把我们的精力聚焦在最关键的元素上，在当前开发系统的操作上下文内为候选解决方案建立原型。

[1] 事实上，对于很多这种复杂度的系统，经常不得不处理很多不同种类的计算机。拥有考虑全面而稳定的架构会缓解开发中期更换硬件带来的很多风险，更换硬件在快速变化的硬件行业频繁发生。硬件产品层出不穷，因此重要的是管理系统的软件/硬件边界，这样在保持系统架构完整性的同时，还可引入新产品来减少系统的开销或改善系统的性能。

列车交通管理系统有两个主要功能：确定列车路线和监控列车系统。相关功能包括交通计划、故障预测、列车位置跟踪、交通状况监控、避免冲突和维护日志。从这些功能，我们定义了 8 个用例，如以所示：

- Route Train（确定列车路线） 建立定义特定列车旅行路线的列车计划；
- Plan Traffic（规划交通） 建立一个交通计划，在开发一个时间范围和地理区域内的列车计划时，该计划提供指南；
- Monitor Train Systems（监控列车系统） 监控机载列车系统正常运行；
- Predict Failure（预测失败） 执行列车系统状况的分析，以预测相对于列车计划失败的概率；
- Track Train Location（跟踪列车位置） 使用 TTMS 资源和 Navstar 全球定位系统（GPS）监控列车的位置；
- Monitor Traffic（监控交通） 监控一个地理区域内的所有列车交通；
- Avoid Collision（避免冲突） 提供自动和手动的方法来避免列车冲突；
- Log Maintenance（日志维护） 提供对列车执行日志维护的方法。

这些用例为列车交通管理系统建立了基本的功能需求，也就是说，告诉我们系统必须为用户做什么。此外，还有影响用例所阐述需求的非功能需求和约束，如下所示。

非功能需求：

- 安全运输旅客和货物；
- 支持列车加速到 250 英里/小时；
- 在 TTMS 边界与交通管理系统操作员互操作；
- 保证最大的复用和兼容现有设施；
- 提供一个持续可用级别达 99.99%的系统；
- 提供完整的 TTMS 能力功能冗余；
- 列车位置精确到 10 码；
- 列车速度精确到 1.5 英里/小时；
- 在 1.0 秒之内反馈到操作员；
- 有内在设计能力来维护和演进 TTMS。

约束：

- 符合国家标准，包括政府和行业的；
- 最大化使用商用现货（COTS）硬件和软件。

现在，已经定义了核心需求（至少在一个很高的级别），我们必须把注意力转到理解列车交通管理系统的用户上。我们发现有三种类型的人——Dispatcher（调度员）、TrainEngineer（列车工程师）和 Maintainer（维护人员）和系统交互。此外，列车交通管理系统和一个外部系统接口 Navstar GPS。这些执行者在 TTMS 内扮演以下角色：

- Dispatcher 建立列车行驶路线和跟踪单个列车的进展；
- TrainEngineer 监控列车的状况并操作列车；

- Maintainer 监控列车系统的状况并维护列车系统；
- Navstar GPS 提供地理位置服务来跟踪列车。

9.1.2 决定系统用例

图 9-1 展示了列车交通管理系统的用例图。在图中，我们看到每一个执行者使用的系统功能，也看到<<include>>和<<extend>>关系被用来组织若干用例之间的关系。用例 Monitor Train Systems 的功能被用例 Predict Failure 扩展。在监控系统的路径上，

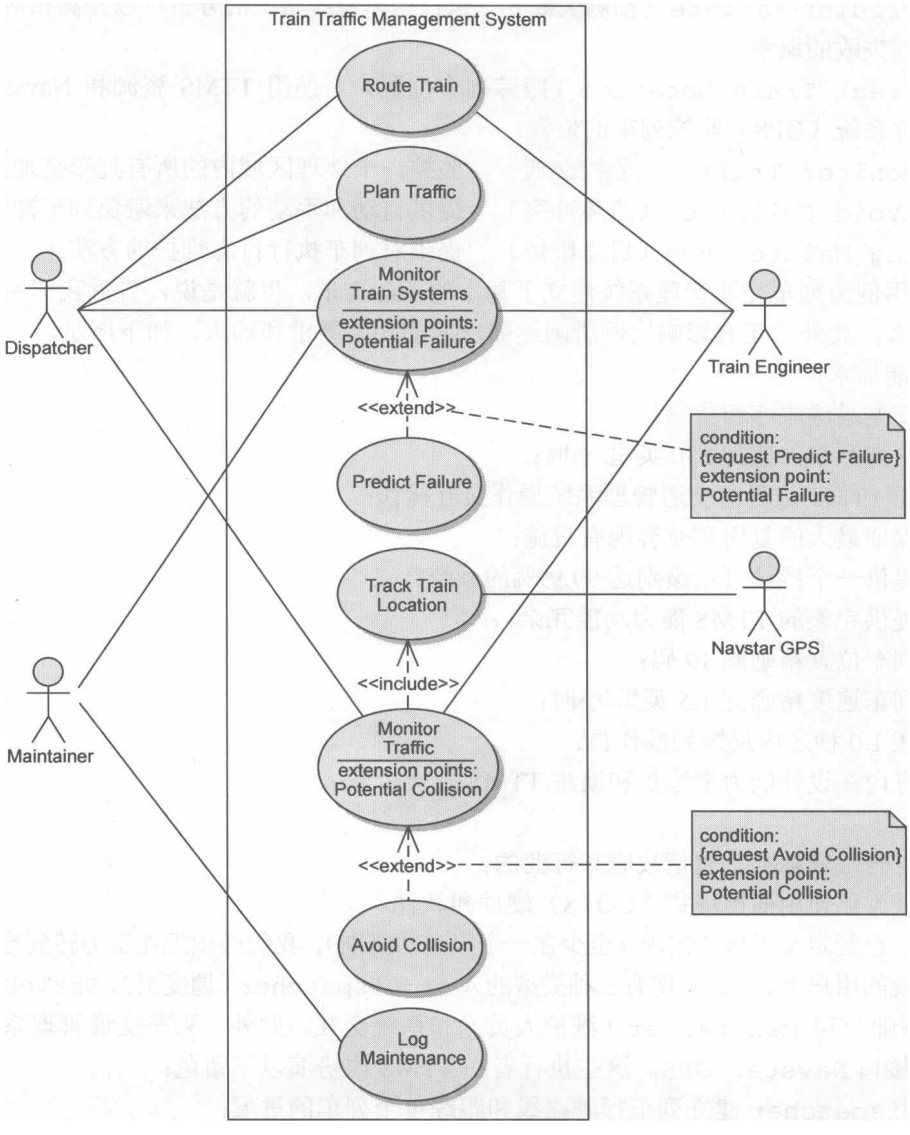


图 9-1 列车交通管理系统的用例图

操作异常或者被挂黄旗表示有问题待查的特定系统，可以请求一个失败预测分析(condition: {request Predict Failure})。这发生在 Potential Failure 扩展点。

Monitor Traffic 用例的功能也被 Avoid Collision 用例扩展了。当监控列车交通时，一个执行者可以选择让系统来辅助避免冲突——在 Potential Collision 扩展点。这种辅助可以支持手动和自动的干预。Monitor Traffic 总是包含 Track Train Location 用例的功能，以便拥有所有列车交通的精确画面。这将通过使用 TTMS 资源和 Navstar GPS 来达成。

可以通过叫作用例规格的文本文档来详述这些用例所提供功能的细节。在以下用例规格中，我们选择聚焦于两个主要的用例——Route Train 和 Monitor Train Systems。用例规格的格式是通用的，提供了设置信息以及主要场景和一个或多个备选场景。

应该说明的是，这些用例规格聚焦于系统用户和列车交通管理系统自身之间边界级别的交互。这个视角经常被称作黑盒视图，因为内部的系统功能外面是看不到的。当关注系统做什么，而不是怎么做时，可使用这个视图。

用例名：Route Train

用例目标：本用例的目标是建立一个列车计划，作为和特定列车路线以及一路上所发生的动作所有密切相关信息的仓储。

接触点：Katarina Bach

修改日期：9/5/06

前置条件：存在和正在开发的列车计划相关的时间范围和地理区域的交通计划。

后置条件：已开发特定列车的列车计划来详述它的旅行路线。

限制：在系统内，每个列车计划会有一个唯一的 ID。在特定时间帧内，资源不可以由多于一个列车计划提交利用。

假设：调度员可以查询和修改列车计划，列车工程师可以查询列车计划。

主要场景：

A. 列车交通管理系统（TTMS）向调度员展示选项列表。

B. 调度员选择开发新的列车计划。

C. TTMS 向调度员展示列车计划模板。

D. 调度员完成列车计划模板，提供关于机车 ID、列车工程师以及带时间路点的信息。

E. 调度员提交已完成的列车计划给 TTMS。

F. TTMS 分配一个唯一 ID 给列车计划并存储它。TTMS 使列车计划可以查询和修改。

G. 用例结束。

备选场景：

触发备选场景的条件如下。

条件 1：基于现有计划开发新的列车计划。

B1. 调度员选择基于现有计划开发新的列车计划。

B2. 调度员提供现有列车计划的搜索条件。

B3. TTMS 提供搜索结果给调度员。

B4. 调度员选择一个现有列车计划。

B5. 调度员完成列车计划。

B6. 继续主要场景步骤 E。

触发备选场景的条件如下。

条件 2：修改一个现有列车计划。

B1. 调度员选择修改一个现有列车计划。

B2. 调度员提供现有列车计划的搜索条件。

B3. TTMS 提供搜索结果给调度员。

B4. 调度员选择一个现有列车计划。

B5. 调度员修改列车计划。

B6. 调度员提交已修改的列车计划给 TTMS。

B7. TTMS 存储已修改的列车计划，使列车计划可以查询和修改。

B8. 用例结束。

用例名：Monitor Train Systems

用例目标：本用例的目标是监控机载列车系统适当地运作。

接触点：Katarina Bach

修改日期：9/5/06

前置条件：机车正在运行。

后置条件：已提供关于机载列车系统运行的信息。

限制：无。

假设：机车正在运作时，提供对机载列车系统的监控。除了通过视频显示之外，还要对系统问题提供声音和可见的指示。

主要场景：

A. 列车交通管理系统（TTMS）向列车工程师展示选项列表。

B. 列车工程师选择监控机载列车系统。

C. TTMS 向列车工程师展示列车系统的概要状态信息。

D. 列车工程师审查概要系统状态信息。

E. 用例结束。

备选场景：

触发备选场景的条件如下。

条件 1：请求系统的详细监控。

E1. 列车工程师选择执行有黄色状况系统的详细监控。

E2. TTMS 向列车工程师展示所选系统的详细系统状态信息。

E3．列车工程师审查详细系统状态信息。

E4．继续主要场景步骤 B。

扩展点——Potential Failure：

条件 2：请求系统的失败预测分析。

E3-1．列车工程师请求系统的失败预测分析。

E3-2．TTMS 为所选系统执行失败预测分析。

E3-3．TTMS 向列车工程师展示系统的失败预测分析。

E3-4．列车工程师审阅失败预测分析。

E3-5．列车工程师请求 TTMS 警告系统维护人员可能会失败。

E3-6．TTMS 警告系统的维护人员。

E3-7．维护人员请求失败预测分析的审阅。

E3-8．TTMS 向维护人员展示失败预测分析。

E3-9．维护人员审阅分析并决定黄色状况严重程度不够达到即时行动。

E3-10．维护人员请求 TTMS 告知列车工程师这个决定。

E3-11．TTMS 提供维护人员的决定给列车工程师。

E3-12．列车工程师选择执行所选系统的详细监控。

E3-13．继续备选场景步骤 E3。

即使列车交通管理系统的需求非常简化，我们还是没有完全阐述它们。它们还是有点模糊，这就像在开发真实世界的大型复杂系统时我们所面临的。正如之前讨论的，高效管理总是变化的需求对成功的开发过程很关键，成功的开发过程的定义就是及时地在预算内提供正确的功能。但不要认为我们的目标是阻止需求的变化，我们不能也不应该这样做。我们可以理解这一点，硬件技术的快速功能升级以及成本的急剧下降，为软件开发问题提供了更多的解决方案。看看今天的个人计算机上运行的难以置信的复杂软件，它们的处理器运行速度高达几个 GHz，还有成 G 的内存。

那么，我们如何容纳变化的需求，特别是涵盖若干年的开发时间范围？我们发现，在这样一个大型的自动化系统中，使用迭代和增量的开发过程是一个管理和需求变化相关的风险的关键方法。另一个方法是设计一个在整个开发中保持弹性的架构。还有一个方法是最大化使用 COTS 硬件和软件，作为 TTMS 的一个约束指引我们。在本章，我们的首要焦点是开发一个可以容纳变化的架构。

9.2 精化

现在我们将注意力转到开发列车交通管理系统的总体架构框架。从分析所需要的系统功能开始，得到 TTMS 架构的定义。然后，开始从系统工程转换到硬件和软件工程的科目。本节最后，我们将描述 TTMS 的关键抽象和机制。

9.2.1 分析系统功能

现在，已经阐述了列车交通管理系统的需求，我们的焦点转到系统的各部分如何集结起来提供所需功能。这个视角通常被称作白盒视图，因为从外面看到了系统内部的功能。我们用活动图来分析各种用例场景，开发下一个级别的细节。

让我们先看看图 9-2，图 9-2 分析了 Route Train 用例的主要场景。这个活动图比较直白，跟随了用例场景的路径。在图中我们看到，在 Dispatcher 创建一个新的 TrainPlan 对象时 Dispatcher 执行者和 RailOperationsControlSystem 的交互，我们指定 RailOperationsControlSystem 为 TTMS 首要的命令和控制中心。

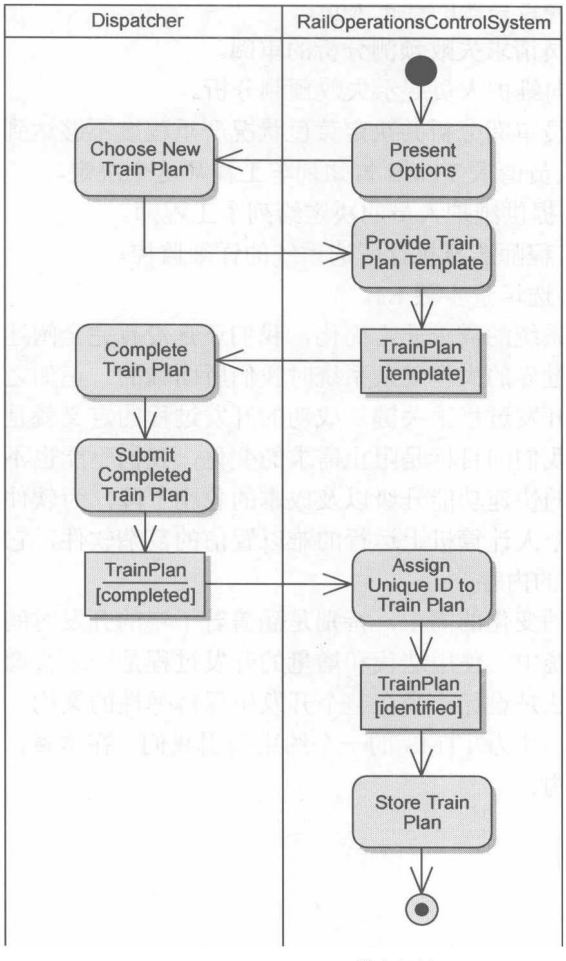


图 9-2 Route Train 的主要场景

当然，在决定 TTMS 的组成元素时，必须考虑功能和非功能的需求以及约束。但我们也有两个完全的技术关注点：封装抽象的欲望和使某个抽象对其他元素可见的需要。换句话说，

我们必须努力设计内聚（把逻辑相关的抽象组织在一起）和松耦合（最小化元素之间的依赖）的元素。因此，我们定义模块化作为被分解成内聚和松耦合元素集合的系统的一个特征。

和图 9-2 对比，图 9-3 的活动图更加复杂一些，因为我们已经展示了 Monitor Train Systems 用例的第一个备选场景，TrainEngineer 选择执行一个 LocomotiveAnalysisandReporting 系统的详细监控，该系统有一个黄色状况。我们看到，TTMS 提供这个能力的组成元素是 OnboardDisplay 系统、LocomotiveAnalysisandReporting 系统、Energy Management 系统和 DataManagement 单元。

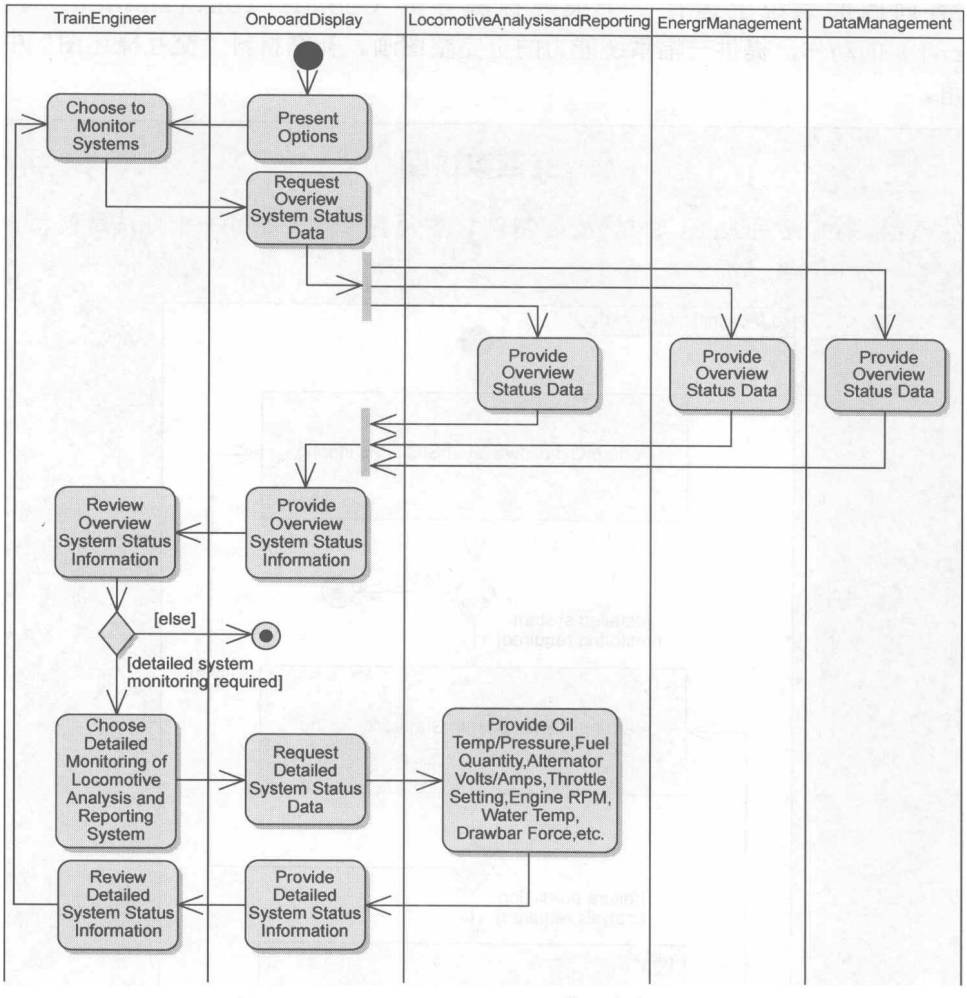


图 9-3 Monitor Train Systems 的一个备选场景

我们看到，OnboardDisplay 系统是 TrainEngineer 和 TTMS 之间的接口。它接收 TrainEngineer 的请求，以监控列车系统，然后从另外三个系统请求适当的数据。概要级别的状态信息被提供给 TrainEngineer 评审。此时，TrainEngineer 可以保

持在概要级别，结束主要场景。但是，在备选场景中，TrainEngineer 请求来自 LocomotiveAnalysisandReporting 系统的更详细评审，因为它展示了黄色状况，意味着某些类型的问题需要注意。作为响应，OnboardDisplay 系统检索来自展示系统的详细数据。在评审这个信息之后，TrainEngineer 返回，监控概要级别的系统状态信息。

图 9-3 的活动图表现一个（备选）场景或两个（首要和备选）分离的场景，这和项目习惯有关。之前描述的第二个备选场景详细说明了 Predict Failure 用例对 Monitor Train Systems 用例功能的扩展。这个场景可以被附加到图 9-3，通过详细描述 TrainEngineer 请求在有问题的系统上运行一个失败预测分析（condition:{request Predict Failure}）的动作，提供一幅系统能力的更完整图画。补充材料“交互概述图”中将展示这个视角。

交互概述图

描绘 Monitor Train Systems 用例首要和备选场景的另一个方法是使用一张交互概述图，如图 9-4 所示。

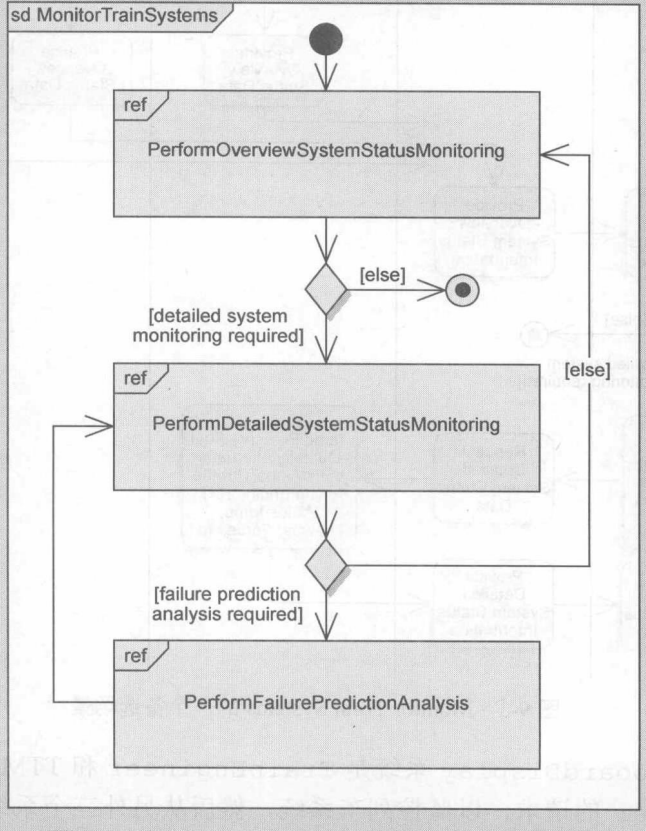


图 9-4 Monitor Train Systems 的交互概述图

名副其实，这张图展示了完整的 Monitor Train Systems 用例功能的高级别概览。这张交互概述图展示了交互发生之间的流，由左上角带 ref 的三个框表示。在引用到交互图的地方，我们可以展示真实的交互，以提供每个场景的细节：Perform Overview System Status Monitoring、Perform Detailed System Status Monitoring 和 Perform Prediction Analysis。

交互概述图可以使用任意类型的交互来展示细节——序列、通信、时间或另一个交互概述。正如我们看到的，这张图可以用来映射从一个交互到另一个的流，这在描述长而复杂的交互时很有用。

9.2.2 定义 TTMS 架构

通过对所有用例场景所需的功能作更为全面的分析，也考虑到非功能需求和约束的影响，我们得到一张列车交通管理系统主要元素的块图，如图 9-5 所示[2]。机车分析和报告系

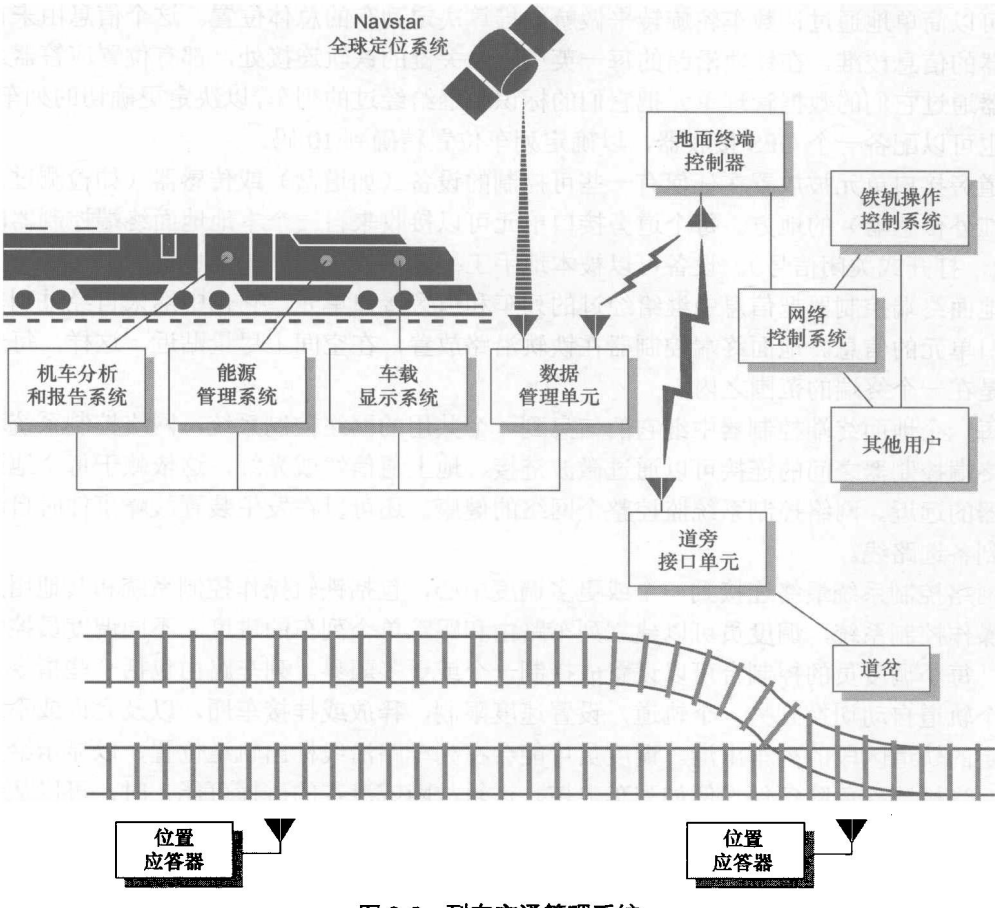


图 9-5 列车交通管理系统

统包括若干数字模拟的传感器，来监控机车状况，包括油温、油压、燃料量、发电机伏特和安培、节流设置、发动机RPM、水温和挂钩牵引力。传感器值通过机载显示系统展示给列车工程师，并在网络上展示给别处的调度员和维护人员。无论何时某个传感器的值溢出正常的操作范围，预警或告警条件会被登记。传感器值日志被维护，以支持维护和燃料管理。

能量管理系统实时建议列车工程师使用最高效的节流和制动设置。这个系统的输入包括轨面和坡度、速度限制、进度、列车负载和可用能源。从这些信息，系统可以决定高效利用燃料的节流和制动设置，设置要符合所要的进度和安全要求。所建议的节流和制动设置、轨面和坡度以及列车位置和速度可以显示在车载显示系统上。

车载显示系统提供列车工程师的人/机接口。来自机车分析和报告系统、能量管理系统和数据管理单元的信息在这里显示。有软键可以让工程师选择不同的显示。

数据管理单元作为所有车载系统和网络剩下部分的通信网关，它和所有列车、调度员和其他用户相连。

列车位置跟踪由网络上的两个设备完成——位置应答器和Navstar GPS。机车分析和报告系统可以简单地通过计数车轮旋转来做航位推算决定列车的总体位置。这个信息由来自位置应答器的信息校准，在铁轨沿路的每一英里以及关键的铁轨连接处，都有位置应答器。这些应答器通过它们的数据管理单元把它们的标识中继给经过的列车，以决定更确切的列车位置。列车也可以配备一个GPS接收器，以确定列车位置精确到10码。

道旁接口单元被放置在任何有一些可控制的设备（如道岔）或传感器（如检测过热轮轴承的红外传感器）的地方。每个道旁接口单元可以接收来自一个本地地面终端控制器的命令（例如，打开或关闭信号）。设备可以被本地手工控制接管。每个单元也可以报告它的当前设置。地面终端控制器把信息中继给经过的列车和道旁接口单元，它也中继来自经过列车和道旁接口单元的信息。地面终端控制器在铁轨沿路放置，在空间上尽量贴近，这样，每一个列车总是在一个终端的范围之内。

每一个地面终端控制器中继它的信息到一个共用的网络控制系统。网络控制系统和每个地面终端控制器之间的连接可以通过微波链接、地上通信线或光纤，这依赖于每个地面终端控制器的远近。网络控制系统监控整个网络的健康，还可以在发生装置故障事件时自动路由信息到备选路线。

网络控制系统最终连接到一个或更多调度中心，包括铁轨操作控制系统和其他用户。在铁轨操作控制系统，调度员可以建立列车路由和跟踪单个列车的进度。不同调度员控制不同疆界，每个调度员的控制台可以设置成控制一个或更多疆界。列车路由包括一些指令，实现从一个轨道自动切换到另一个轨道，设置速度限制，释放或挂接车厢，以及允许或不允许一个特定的轨道区段的列车出港。调度员可能会在列车路沿线标出轨道位置，以显示给列车工程师。当检测到危险条件（例如列车失控、铁轨故障或潜在的碰撞可能）时，可以从铁轨操作控制系统（调度员手动或自动）停止列车。调度员也可以调用列车工程师个人得到的任何信息，以及发送移动权限、道旁设备设置和计划修订。

很明显，轨道布局和道旁装置会随着时间推移而变化；此外，列车的数量和它们的路由

每天都会变化。因此列车交通管理系统必须被设计成允许容纳新的传感器、网络和处理器技术。我们的非功能需求有内设的能力来维护和演进 TTMS，所以非常明显，我们必须设计一个架构，它有随着时间进化的弹性。此外，我们的约束告诉我们，系统必须依赖于国家标准（政府和行业），最大化地使用 COTS 硬件和软件。

9.2.3 从系统工程到硬件和软件工程

在分析阐述列车交通管理系统首要功能需求的用例场景时，到目前为止，我们的开发执行的是系统工程，而不是硬件或软件工程活动。从这个分析，我们能够指定块图的主要元素，来定义一个候选的 TTMS 系统架构。当我们继续开发时，更低级别硬件和软件元素的架构将基于系统架构师心里可能有的概念而进化。最终，我们决定由硬件、软件或手工操作履行的系统功能的比例。此时，开发变得更像是系统、硬件、软件和操作工程团队之间的合作。

图 9-5 的块图展现了一个使用面向对象方法开发的候选架构，可以清楚看到架构的组件本质。我们看到，TTMS 元素在执行系统主要功能时展示出内聚和松耦合。在进一步分析系统的功能时，可以继续使用活动图（特别是和领域专家一起工作时），活动图提供了每个系统元素在系统场景中协作时所作工作的清晰视角。在进行到更低级别的架构层次时，需要更特定的交互细节。因此，我们更有可能使用序列图、类图和原型来检查所需的系统行为。

图 9-6 提供一张序列图，捕获了一个自动处理每日列车次序的简单场景，比起之前在图 9-2 展示的活动图，序列图提供了更多列车交通管理系统内部工作的细节。我们假设这个场景实质上开始于图 9-2 的结果，新的列车计划已经被创建。在这里我们只看到发生的主要事件以及系统元素的交互。在后面的开发中，我们必须开始归档元素细节，例如属性定义、操作签名和关联规格。

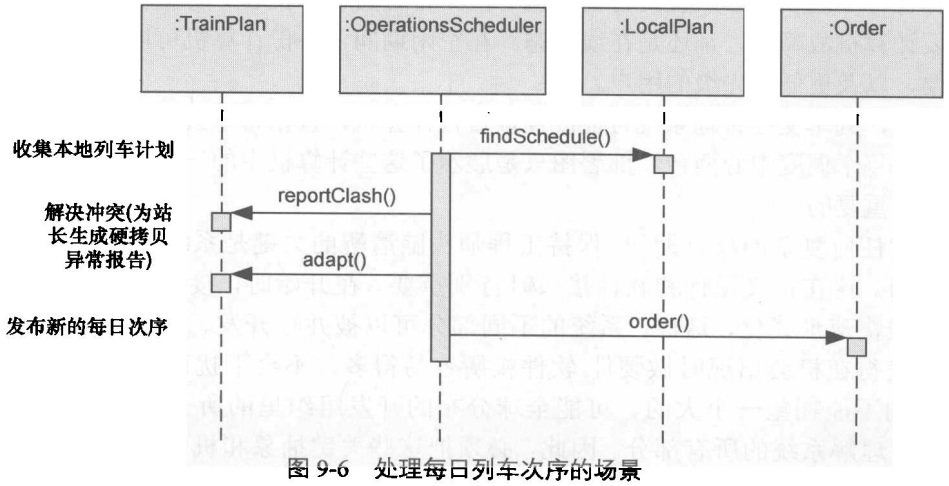


图 9-6 处理每日列车次序的场景

在完成TTMS功能的系统工程分析（穿过它的架构级别）之后，必须分配系统需求到硬件、软件甚至手工操作的元素。说“甚至手工操作的”，是因为我们太经常认为每件事情能够而且应该是自动的。

当然，这是一个典型的目标，但我们必须理解，一些功能，特别是安全关键的区域，应该体现（以及在某些情况下必须通过法律）人性化设计。在一些情况下，分配需求到硬件或软件相当明显。例如，对于描述列车计划日程来说，软件是合适的实现方式。对于车载显示系统和铁轨操作控制中心的显示，可能使用现成的终端或工作站。这些分配决策由很多标准来驱动，包括复用问题、现有商业产品以及系统架构师的经验和偏好。当选择现有商业产品，例如系统中的很多传感器时，我们把那些元素设计决策分配给了生产厂商的工程师。一般来说，虽然在需要最多弹性的地方，我们会倾向于软件，但在性能至关重要的地方，我们会选择硬件。

对于问题的目标，我们假设系统架构师已经选择了一个初始的硬件架构。不必认为这个选择是不可逆的，但至少它给我们一个分配软件需求的开始点。当进行分析和设计时，我们需要自由折中硬件和软件：可能后面会决定，为了满足某些需求，需要额外的硬件，或者某个功能通过软件执行比硬件要好。

图9-7阐明了列车交通管理系统的目标部署硬件，使用了部署图表示法。这个硬件架构平行于之前图9-5所示的系统块图。具体地说，每个列车有一台计算机，包含机车分析和报告系统、能量管理系统、车载显示系统和数据管理单元。每个位置应答器连接到一个发送器，通过发送器，消息可以被发送到经过的列车；没有计算机和位置应答器关联。另一方面，每个道旁设备（包括一个道旁接口单元和它的道岔）的集合由一台道旁计算机控制，该计算机可能通过它的发送器和接收器与经过的列车或地面终端控制器通信。正如之前讨论的，发送器和接收器之间的通信可能通过微波链接、地上通信线或光纤完成。每个地面终端控制器最终连接到一个局域网，每个调度中心一个（包含铁轨操作控制系统）。因为需要不间断的服务，我们选择在每个调度中心放置两台计算机——一台首要计算机和一台备份计算机，期望无论何时首要计算机故障时，都还是在线状态。在空闲期间，备份计算机可以用来服务其他的计算的需要，以及更低优先级的用户。

运作时，列车交通管理系统可能涉及数百台计算机，包括每个列车一台，每个道旁接口单元一台和每个调度中心两台。部署图只是展示了这些计算机中的一些，因为类似计算机的配置完全是重复的。

在开发任何复杂的项目期间，保持工程师头脑清醒的关键是系统关键元素之间坚实而显式的接口。这在定义硬件和软件接口时特别重要。在开始时，接口可以松散地定义，但它们必须被快速形式化，这样，系统的不同部分可以被并行开发、测试和发布。定义良好的接口也使得在机会出现时做硬件/软件权衡容易得多，不会干扰已经完成的系统部分。此外，我们不能期望一个大的、可能全球分布的开发组织里的所有开发人员有一个完整的视图，并理解系统的所有部分。因此，必须把这些关键抽象和机制的规格留给最好的架构师。

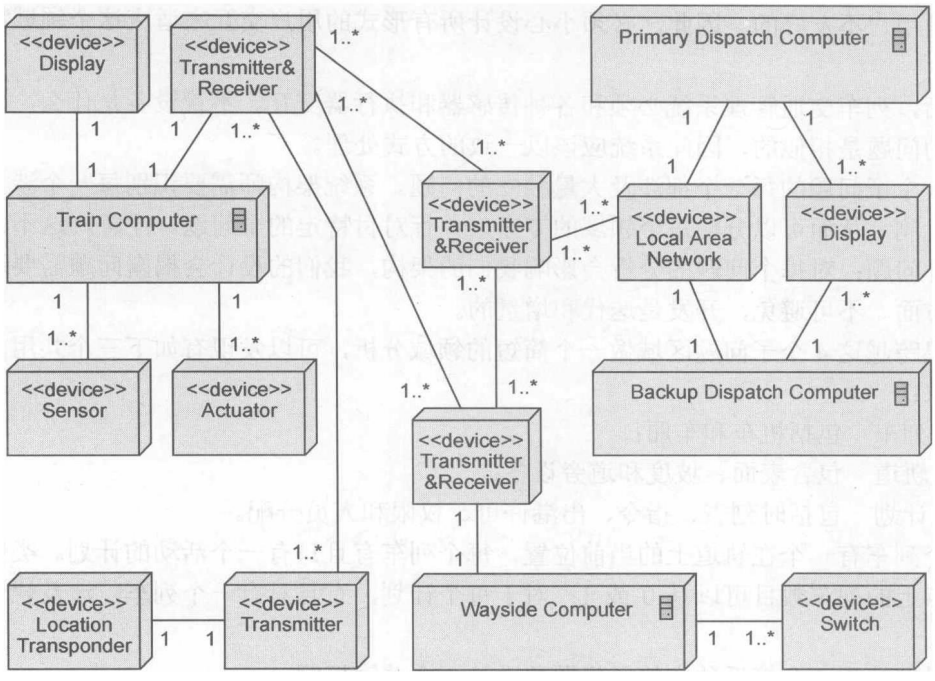


图 9-7 列车交通管理系统的部署图

9.2.4 关键抽象和机制

研究列车交通管理系统的需求可知，实际上有 4 个不同的子问题要解决：

- 网络；
- 数据库；
- 人/机接口；
- 实时的模拟和数字设备控制。

在涉及最大的开发风险时，如何来识别这些问题？

把这个系统绑到一起的线是一个分布式的通信网络。消息通过无线电，从应答器被传送到列车，以及在列车和地面终端控制器之间、列车和道旁接口单元之间、地面终端控制器和道旁接口单元之间被传送。消息也必须在调度中心和单个地面终端控制器之间被传送。整个系统的安全操作依赖于及时可靠地传输和接收消息。

另外，这个系统必须同时跟踪很多不同列车的当前位置和计划路线。即使存在来自整个网络的并发更新和查询，也必须保持这个信息最新而且自我一致。这基本上是一个分布式数据库问题。

人/机接口工程提出一个不同的问题集合。具体地说，这个系统的用户主要是列车工程师和调度员，他们不需要精于使用计算机。专业软件工程师可以接受操作系统如 UNIX 或 Windows 的用户界面，但对于像列车交通管理系统这样的应用的最终用户来说，这样的界面

被认为是用户不友好的。因此，必须小心设计所有形式的用户交互来适应这个领域特定的用户群。

最后，列车交通管理系统必须和各种传感器和执行器交互。不管设备是什么，感应和控制环境的问题是相似的，因此系统应该以一致的方式处理。

这4个子问题的每一个都涉及大量独立的问题。系统架构师需要识别每一个涉及的关键抽象和机制，从而可以分配每个领域的专家来并行对付特定的子问题。注意，这不是一个分析或设计问题：对每个问题的分析会影响我们的架构，我们的设计会揭露问题需要进一步分析的新方面。不可避免，开发是迭代和增量的。

如果跨越这4个子问题区域做一个简短的领域分析，可以发现有如下三个共用的高级别关键抽象：

- 列车　包括机车和车厢；
- 轨道　包含表面、坡度和道旁设备；
- 计划　包括时刻表、指令、出港许可、权限和人员分配。

每个列车有一个在轨道上的当前位置，每个列车有且只有一个活动的计划。类似地，每个轨道点上的列车数目可以是0或1；对于每个计划，有且只有一个列车，涉及轨道上的很多点。

可以继续为4个接近独立的子问题各设计一个关键机制：

- 消息传送；
- 列车日程计划；
- 显示信息；
- 传感器数据采集。

这4个机制形成了系统的灵魂。它们代表通向我们已经识别为最高开发风险区域的道路。因此，在这里部署最好的系统架构师是关键的，他们实验候选方法，并最终定下一个框架。初级开发人员可以从这个框架构造系统的剩余部分。

9.3 构造

架构设计涉及建立系统的中心类结构，以及赋予这些类共同协作的规格。尽早聚焦于这些机制直接攻击了系统中风险最高的元素，具体地捕获系统架构师的愿景。最终，这个阶段的产品作为类和协作的框架，最后的系统的其他功能元素可以在其上建造。

在本节中，我们将从检查这个系统的4个关键机制的语义开始——包括消息传送、列车日程计划、显示信息和传感器数据采集。这引起针对发布管理的讨论，发布管理支持我们的迭代和增量开发过程。本节将通过分析如何开发一个支持TTMS子系统规格的系统架构，得出结论。

9.3.1 消息传送

消息，我们的意思不是指面向对象编程语言里面的方法调用，而是指问题域词汇表里的概念，它处在更高级别的抽象。例如，典型的列车交通管理系统消息包括像这样的信号：激活道旁设备，指示列车路过特定位置和调度员下达给列车工程师的命令。一般来说，这些种类的消息会在 TTMS 内的两个不同级别传送：

- 计算机和设备之间；
- 计算机之间。

我们的兴趣在第二个级别的消息传送。因为我们的问题涉及地理上分布式的通信网络，所以必须考虑诸如噪声、装置故障和安全性等问题。

可以先通过检查每对通信中的计算机来识别这些消息，如之前的部署图所示（参见前面图 9-7）。对于每个对，我们必须问如下三个问题。

（1）每台计算机管理什么信息？

（2）一台计算机到另一台应该传递什么信息？

（3）这个信息应该在什么抽象级别？

这些问题没有确定的答案。我们还是必须使用迭代的方法，直到定义出满意的正确消息，这样，系统里没有通信瓶颈（也许因为路径上有太多消息，或者消息太大或太小）。

在这个级别的设计，绝对关键的是聚焦于这些消息的实质，而不是形式。常看到系统架构师一开始为消息选择“比特级别”的表达方式，过早地选择这样一个低级别的表达方式的真正问题是，依赖于特定表达方式的每一个客户肯定会改变和受到干扰。而且，在设计流程的此时，我们不能充分知道这些消息如何用来做关于时间和空间高效表达方式的明智决策。

通过聚焦于这些消息的实质，我们尽量把焦点放在每个消息类的外部视图。换句话说，必须决定每个消息的角色和责任，以及在每个消息上可以有意义执行的操作。

图 9-8 中的类图捕获关于列车交通管理系统中一些最重要消息的设计决策。注意，所有消息最终都是泛化抽象类 `Message` 的实例，该类包含所有消息共有的行为。三个更低级别的类代表消息的主要类目，即 `Train StatusMessage`、`TrainPlanMessage` 和 `WaysideDeviceMessage`。这些类每一个都被进一步特化。实际上，我们最后的设计可能包括大量这样的特化类，那时这些中间类的存在甚至变得更加重要；没有它们，我们最终会得到很多不相关（因此难于维护）的组件——代表每个不同的特化类。正如我们的设计所揭示的，我们可能发现其他重要的消息分组，并因此发现其他中间类。幸运的是，以这种方式重新组织类层次通常对最终使用基类的客户有最小的语义影响。

作为架构设计的一部分，尽早稳定关键消息类的接口是明智的。为了制定所有这样的类的角色和责任，可以从这个层次更令人感兴趣的基类的领域分析开始。

一旦设计了更重要的消息的接口，就可以以这些类为基础编写程序，来模拟消息流的创建和接收。我们可以把这些程序用作临时的脚手架，来测试系统的不同部分，包括在开发期间和它们要接口的部分完成之前。

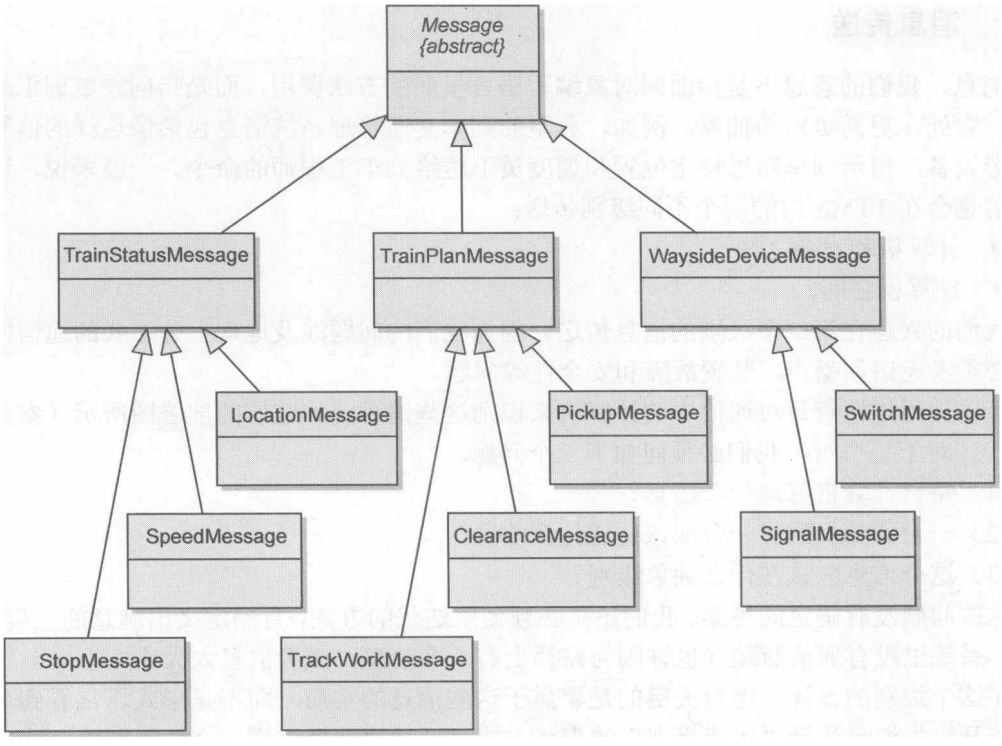


图 9-8 Message 类图

图 9-8 中的类图无疑是不完整的。实践上，我们发现可以先识别最重要的消息，并在揭露共同形式更少的通信时让所有其他消息演进。使用一个面向对象架构允许我们增量添加这些消息，而不会瓦解已有的系统设计，因为这样的改变一般是向上兼容的。

一旦我们满意这个类结构，就可以开始设计消息传递机制本身。对于这个机制，这里有两个相互竞争的目标：它必须提供可靠的消息送达，还要在足够高的抽象级别进行，这样客户不需要担心消息送达如何发生。这样的一个消息传递机制允许它的客户对消息如何发送和接收做简化的假设。

图 9-9 提供了一个捕获消息传递机制设计的场景。正如这张图所表明的，为了发送一条消息，客户首先创建一条新的消息 m，然后广播它到其节点的消息管理器，消息管理器的责任是把消息放进队列，以便最后的传输。注意，我们的设计使用 4 个主动对象（拥有它们自己的控制线程），在对象表示法中多加一条的竖线表示：Client、messageMgr:Queue、messageMgr':Queue 和 Receiver。还要注意，消息管理器接收要广播的消息作为参数，然后使用 Transporter 对象的服务来把消息减小为它的标准形式，并在整个网络广播。

正如图 9-9 所说明的，我们选择让它成为一个异步的操作（用开叉的箭头表示），因为我们不想让客户等待消息通过无线电链接被发送，这需要时间来编码、解码，也许因为噪声需要重传。最终，网络另一边的一些 Listener 对象接收这个消息，以标准形式将它展示给它的

节点的消息管理器。消息管理器接下来创建一条并行的消息，并将其放入队列。接收器可以在它的消息管理器队列头部阻塞，以等待下一条消息到达，该消息将作为操作 nextMessage() 的参数移交，这是一个同步操作。

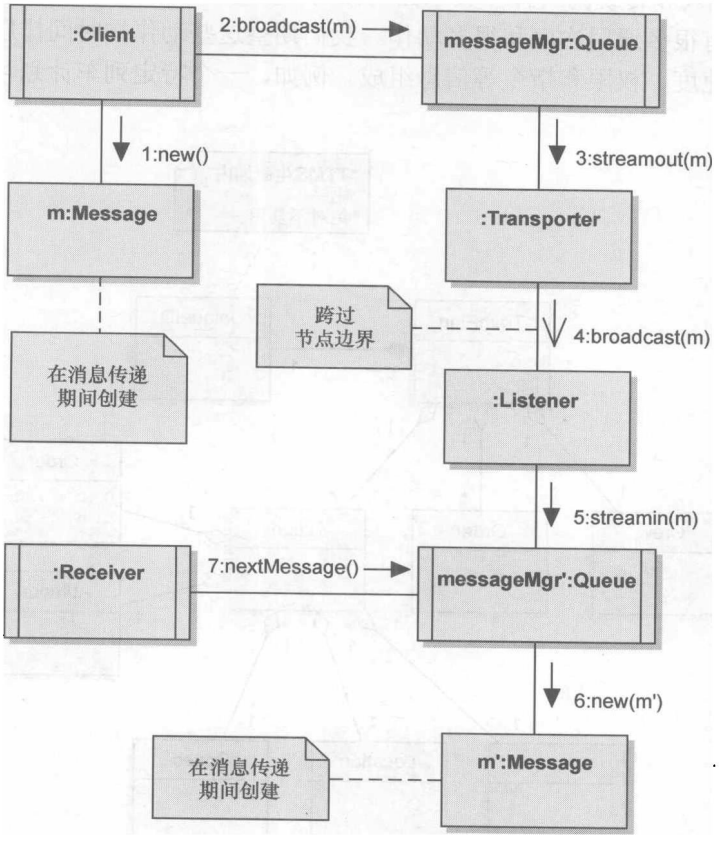


图 9-9 消息传递机制

我们的设计把消息管理器放在 ISO 开放系统互连（OSI）网络模型[3]的应用层。这允许所有发送消息的客户和接收消息的客户在最高抽象级别操作，即以应用特定消息的形式。

我们期望这个机制的最后实现更复杂一些。例如，可能要添加加密和解密行为，并引进代码来检测和修正错误，这样，当存在噪声或装置故障时，仍然可以保证可靠的通信。

9.3.2 列车日程计划

正如之前说明的，列车计划概念是列车交通管理系统运作的中心。每个列车有且只有一个活动的计划，每个计划只被分配到一个列车，在轨道上可能会涉及很多不同次序和位置。

第一步是确切决定哪些部件构成列车计划。为此，需要考虑一个计划的所有潜在客户以及我们期望它们每一个如何使用该计划。例如，一些客户可能允许创建计划，另一些可能允

许修改计划，还有一些可能只允许阅读计划。在这个意义上，一个列车计划作为所有和特定列车的路线以及一路上发生的动作（如挂载或卸除车厢）密切相关信息的仓储。

图 9-10 演示了我们关于 TrainPlan 类结构的战略决策。我们用一张类图来展示组成一个列车计划的部件（和传统的实体—关系图所做的很类似）。可以看到每个列车计划有且只有一个职员，可能有很多通用次序和很多动作。我们期望这些动作按时间排序，每个动作由例如时间、位置、速度、权限和指令等信息组成。例如，一个特定列车计划可能由表 9-1 所示动作组成。

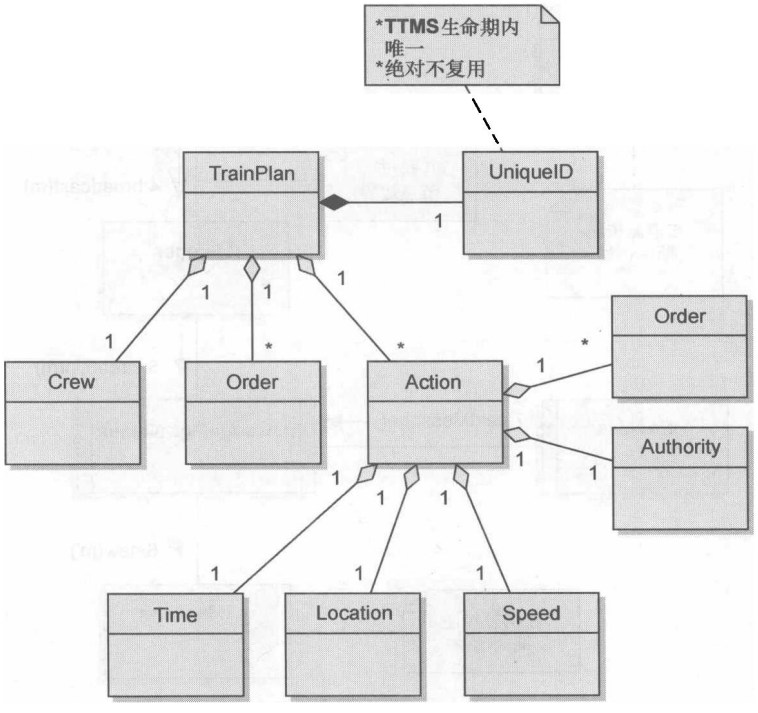


图 9-10 TrainPlan 类图

正如图 9-10 中所表明的，TrainPlan 类有一个 UniqueId，它的目标是提供唯一识别每个 TrainPlan 实例的数字。因为这里的信息复杂，这张图中的类也可以被考虑成类的属性，那就是真正的孤立类。例如，UniqueID 类不只是一个识别数字，它包含各种满足严格的国家和国际规定所必需的各种属性和操作。另一个例子是职员的工作场所有限制——他们可能只在某个位置工作，或者在特定时间某个位置必须保持速度限制。

正如对 Message 类和它的子类所做的，我们可以在开发过程早期设计列车计划最重要的元素；在我们对各种客户应用计划时，它的细节会随着时间进化。

同一时间内，我们可能会有过多的活动和不活动的列车计划。这个事实带来了之前谈过的数据库问题。图 9-10 中的类图可以作为这个数据库的逻辑方案的概要。因此，下一个问题很简单，列车计划放在哪里？

在更完美的世界里，没有通信噪声或延迟和无限的计算资源，我们的解决方案是把所有列车计划放在单个中央数据库里。这个方法保证每个列车计划只会产生一个实例。但是真实世界和这相差甚远，所以这个解决方案不实际。

必须考虑通信延迟，我们没有无限的处理器周期。因此，不得不从列车上访问位于调度中心的计划根本不能满足我们的实时和近实时需求。

但是，我们可以在我们的软件里创造单个中央数据库的错觉。基本上，我们的解决方案是有一个列车计划数据库位于调度中心计算机，还有单个计划的拷贝分布在网络上各个站点。然后，出于效率的考虑，每个列车计算机可以保留一份它当前计划的拷贝。因此，车载软件查询这个计划的延迟可以忽略。如果有调度员动作或列车工程师决策（可能性较小）导致的计划改变，我们的软件必须能够保证该计划的所有拷贝能及时更新。

表 9-1　　列车计划可能包含的动作

时　间	位　置	速　度	权　限	次　序
0800	Pueblo	As posted	See yardmaster	Depart yard
1100	Colorado Springs	40 mph		Set out 30 cars
1300	Denver	45 mph		Set out 20 cars
1600	Pueblo	As posted		Return to yard

这个场景的展现方式是我们的列车日程计划机制的一个功能，如图 9-11 所示。每个列车计划的首要版本驻留在调度中心的一个中央数据库，有零或多个镜像拷贝分散在网络。无论

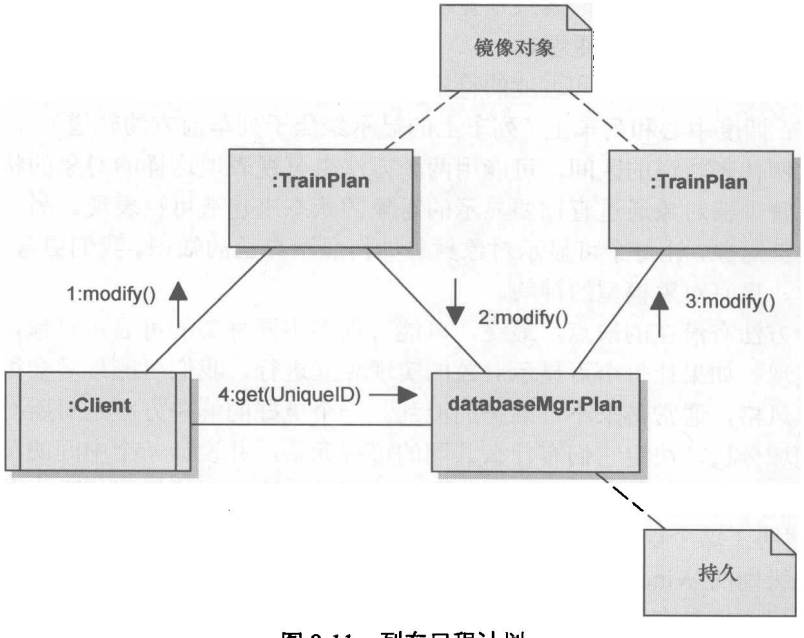


图 9-11　列车日程计划

何时一些客户请求一列特定列车计划的拷贝（通过操作 get()，把 UniqueId 值作为一个参数调用），首要版本都将作为参数被克隆和移交给客户，拷贝网络位置被记录进数据库。现在，假设列车上的一个客户需要对特定计划做改变，也许是列车工程师的一些动作导致的。最终，这个客户会调用列车计划拷贝上的操作，修改它的状态。这些操作也会发送消息给中央数据库，以同样的方式修改计划首要版本的状态。因为我们记录列车计划的每个拷贝在网络上的位置，所以也可以广播消息到中央仓储，中央仓储强迫所有剩下的拷贝的状态做相应的更新。为了保证变更在网络上一致，可以使用记录锁定机制，这样，直到所有拷贝和首要版本更新之前，变更不会被提交。

如果调度中心的一些客户引发该变更——也许由一些调度员的动作引起，这个机制同样适用。首先，计划的首要版本会被更新，然后对所有拷贝的变更会通过同样的机制广播到整个网络。在任何一种情况下，我们到底如何广播这些变更？答案是使用之前设计的消息传递机制。具体地说，需要把一些新的列车计划消息添加到我们的设计，然后在这个更低级别的消息传递机制上建造我们的列车计划机制。

在调度计算机上使用购买的现货数据库管理系统，以满足任何数据库备份、恢复、审计跟踪和安全性的需求。

9.3.3 显示信息

使用现货技术满足数据库需要，可以帮助我们聚焦于问题的领域特定部分。通过使用标准图形设施，可以满足我们的显示需要。使用现货图形软件，高效地提升了系统的抽象级别，这样，开发人员不需要担心在像素级别操纵可显示对象的可视表现。封装关于如何可视表现各种对象的设计决策，依然是重要的。

例如，考虑显示轨道表面和坡度的特定区间。我们的需求限定这样一个显示可以出现在两个不同地方：调度中心和列车上（列车上的显示聚焦于列车前方的轨道）。假设我们有一些类，它们的实例代表轨道的区间，可能用两个方法来可视表现这样的对象的状态。首先，可能有一些显示管理器对象通过查询要显示的对象的状态来建造可视表现。另一种方法是，可以不要这个外部对象，让每个可显示对象封装如何显示自己的知识。我们更喜欢第二种方法，因为它更简单，更有对象模型的神韵。

但是这个方法有潜在的缺点。最终，可能有很多不同种类的可显示对象，每个由不同组的开发人员实现。如果让每个可显示对象的实现独立进行，我们可能最后会得到冗余代码，有不同的实现风格，通常带来不可维护的混乱。一个更好的解决方案是对所有种类的可显示对象做一个领域分析，决定它们有什么共同的可视元素，并设计一个中间的类工具集合，为这些共同的图像元素提供显示例程。这些类工具可以依次在更低级别的现货图形包上建造。

图 9-12 阐明了这个设计，展示了共享共同类工具的所有可显示对象的实现。这些工具依次建造在更低级别的 Windows 接口上，这些接口对所有高层次类隐藏。实战上，例如 Windows API 这样的接口不容易在单个类里表达。因此，我们的图简化了一些。更有可能的情况是，我们的实现需要一套和 Windows API 及列车显示工具对等的类工具。

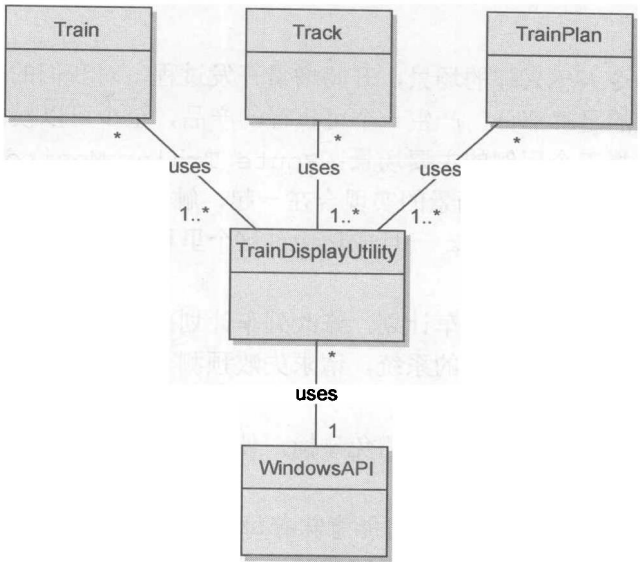


图 9-12 用于显示的类工具

这个方法的主要优势是它限制了任何硬件/软件权衡导致的更低级别变化的影响。例如，如果我们发现需要用更强有力的设备替换我们的显示硬件，只需要重新实现 `TrainDisplayUtility` 类中的例程。没有这个例程集合，低级别变化就要求修改每一个可显示对象的实现。

9.3.4 传感器数据采集

正如我们的需求所建议的，列车交通管理系统包括很多不同种类的传感器。例如，每个列车上监控油温、燃料量、节流设置、水温、牵引负荷等的传感器。类似地，一些道旁设备上的活动传感器报告道岔和信号的当前位置等其他事情。各种传感器返回值的种类都是不同的，但不同传感器数据的处理大体相同。此外，大多数传感器必须周期性取样。如果值在某个范围内，没有特别的事情发生，只是通知一些客户新的值；如果这个值超过当前预设限制，可能要警告一个不同的客户；最终，如果这个值大大超出它的上限，可能需要响起一些警报，并通知另一个客户采取极端动作（例如，当机车油压掉到危险的水平时）。

为每一种传感器复制这个行为不只让人厌烦和容易出错，而且通常导致冗余代码。除非我们发掘共性，否则不同的开发人员最终会开发出多个解决方案来解决同一问题，导致大量出现差别甚微的传感器机制，系统更难维护。因此，非常值得对所有周期性的、非离散的传感器做一个领域分析，这样我们可以为所有种类的传感器开发一个通用的传感器机制。可以使用一个包含一个传感器类层次的架构，以及一个基于框架的、周期性从这些传感器获取数据的机制。

9.3.5 发布版本管理

既然使用增量开发方法，我们将研究发布版本管理技能的使用和进一步分析系统架构及

其子系统规格。

先选择一小部分令人感兴趣的场景，开始增量开发过程，对我们的架构做一个垂直的切片。然后，实现系统的足够部分，产生一个可执行的产品，至少可以模拟这些场景的执行。

例如，可以只选择三个用例的主要场景：Route Train、Monitor Train Systems 和 Monitor Traffic。这三个场景的实现合在一起，触及了几乎每一个关键的架构接口，因此强迫我们验证我们的战略假设。一旦成功通过这个里程碑，就可以按照以下序列生成一个新的发布流：

（1）基于已有列车计划创建列车计划，修改列车计划；

（2）请求详细监控有黄色状况的系统，请求失败预测分析，请求维护人员审查失败预测分析；

（3）手工避免碰撞，请求自动协助避免碰撞，使用 TTMS 资源或 Navstar GPS 跟踪列车交通。

对于 12～18 个月的开发周期，这可能意味着每 3 个月产生一个相当稳定的发布版本，其中每个建造在其他版本的功能之上。当完成时，我们已经覆盖系统中的每一个场景。

这个策略的成功关键是风险管理，对于每个发布版本，我们识别最高开发风险并直接攻击它们。对于控制系统应用，这意味着引进测试尽早积极控制（这样我们可以足够早地识别任何系统控制漏洞，对它们做一些事情）。正如发布序列所说明的，这也意味着为每个发布版本广泛选择场景，跨越系统的功能元素。这样，不会因为我们的分析中未预见到的鸿沟而造成盲点。

9.3.6 系统架构

非常大的系统的软件设计通常应在目标硬件完成之前开始。软件设计常常比硬件设计需要长得多的时间，在任何情况下，必须针对整个过程中的每件事情作权衡。这暗示软件中的硬件依赖必须被最大限度地隔离，这样，在缺少稳定的目标环境时，软件设计可以进行。这也暗示着在头脑中软件必须以可替换子系统的理念设计。在一个命令和控制系统，如列车交通管理系统中，我们希望在开发系统的软件期间利用已成熟的新硬件技术。

还必须尽早对系统的软件有一个明智的物理分解，这样分包商可以在系统的不同部分并行工作。大系统的物理分解经常受很多非技术原因的驱使，其中最重要的也许是分配工作到独立的开发人员团队。分包商的关系通常在复杂系统生命期的早期建立，这时通常没有足够信息来对合适子系统分解作出坚实的技术决策。

如何选择合适的子系统分解？最高级别的对象通常围绕功能线聚集。再说一次，这和对象模型并非正交的。因为术语功能（functional）在这里的意思不是指体现简单输入/输出映射的算法抽象。我们是在谈论代表对外可见和可测试行为的场景，这些场景由对象逻辑集合的动作协作产生。因此，首先识别的最高级别抽象和机制是一个好的候选，可以围绕它组织我们的子系统。我们可以首先断言存在这样的子系统，然后随时间演进它们的接口。

图 9-13 所示组件图表达了我们关于列车交通管理系统顶级系统架构的设计决策。这里有

一个分层架构，包含了我们之前识别的 4 个子问题的功能，即网络、数据库、人/机接口和实时设备控制。

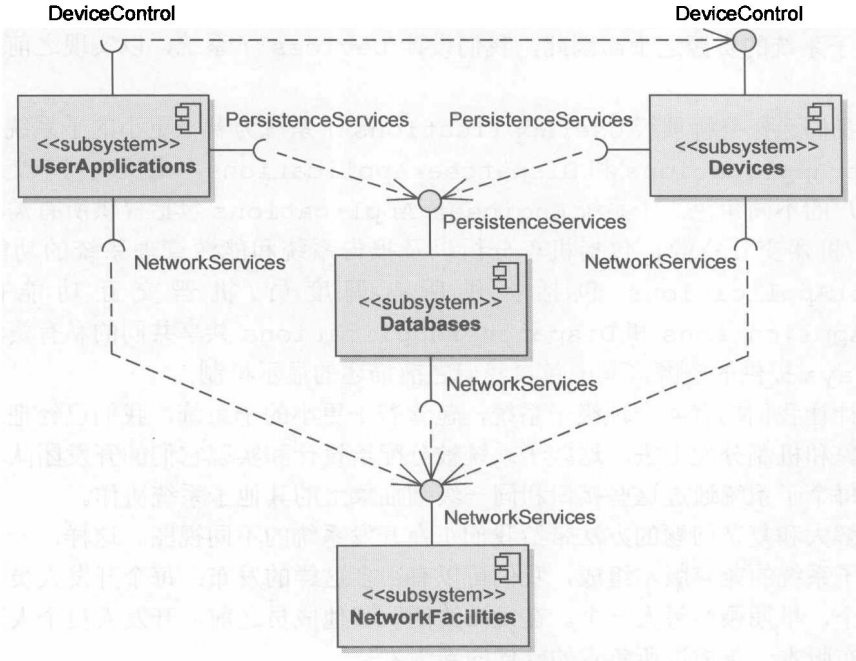


图 9-13 列车交通管理系统的顶级组件图

9.3.7 子系统规格

如果我们聚焦于这些子系统中的任意一个外部视图，会发现它有对象的所有特征。它有唯一、静态的标识，它包含很多状态，展示非常复杂的行为。子系统是其他子系统和最终类的仓储，因此，它们最好由通过供给接口输出的资源刻画，例如，图 9-13 所示的 NetworkFacilities 子系统提供的 NetworkServices。

图 9-13 所示的组件图只是 TTMS 子系统架构规格的一个开始点。这些顶级子系统必须进一步被分解到多个架构级别的嵌套子系统。例如 NetworkFacilities 子系统，我们把它分解为两个其他子系统：一个私有的 RadioCommunication 子系统和一个公开的 Messages 子系统。私有的子系统隐藏物理无线电设备的软件控制细节，而公开的子系统提供之前设计的消息传递机制功能。

名为 Databases 的子系统建造在 NetworkFacilities 子系统的资源之上，并实现之前创建的列车计划机制。我们选择进一步分解这个子系统为两个公开的子系统，代表系统中的主要数据库元素。我们把这些嵌套的子系统分别命名为 TrainPlanDatabase 和 TrackDatabase。我们也期望有一个私有的子系统 DatabaseManager，它的目标是提供两个领域特定数据库的所有公共服务。

在 Devices 子系统中，我们选择把和所有路旁设备相关的软件分组到一个子系统，和所有车载机车执行器和传感器相关联的软件分组到另一个子系统。这两个子系统对于 Devices 子系统的客户来说都是可访问的，他们都建造在 TrainPlanDatabase 和 Messages 子系统的资源之上。因此，我们设计 Devices 子系统，以实现之前描述的传感器机制。

最终，我们选择分解顶级 UserApplications 子系统为若干更小的子系统，包括子系统 EngineerApplications 和 DispatcherApplications，来反映列车交通管理系统两个主要用户的不同角色。子系统 Engineer Applications 包括提供所有需求中指定的列车工程师/机器交互资源，包括机车分析以及报告系统和能源管理系统的功能。子系统 DispatcherApplications 包括提供所有调度员/机器交互功能的软件。EngineerApplications 和 DispatcherApplications 共享共同的私有资源，如从子系统 Displays 提供的资源，其中包含我们之前描述的显示机制。

这个设计让我们拥有 4 个顶级子系统，包含若干更小的子系统，我们已经把之前发现的所有关键抽象和机制分配上去。这些子系统被分配给设计和实现它们的开发团队，保持已定义的接口，每个子系统通过这些接口和同一级别抽象上的其他子系统协作。

这个分解大和复杂问题的方法带给我们正在开发系统的不同视图。这样，一个系统发布版本由每个子系统的兼容版本组成，我们可以有很多这样的发布：每个开发人员一个，质量保证团队一个，早期顾客每人一个。在发布给团队其他成员之前，开发人员个人可以创建自己的稳定发布版本，集成进所负责的软件的新版本。

做这个工作的关键是小心设计子系统接口。一旦工程化，这些接口必须被严格保护。如何决定每个子系统的外部视图？把每个子系统看作一个对象。因此，问同样的问题，就像问一个更原生的对象：这个对象体现什么状态？客户可以有意义地执行哪些操作？什么操作需要其他对象？

例如，考虑子系统 TrainPlanDatabase。它建造于其他 3 个子系统之上（Messages、TrainDatabase 和 TrackDatabase），有若干重要的客户，即 4 个子系统——WaysideDevices、LocomotiveDevices、EngineerApplications 和 Dispatcher Applications。TrainPlanDatabase 包含相对直白的状态，具体地说，即所有列车计划的状态。当然，前提是这个子系统必须支持分布式的列车计划机制行为。从外部看，客户看到单个的数据库，但从里面看，我们知道这个数据库实际上是分布的，因此必须在 Messages 子系统的消息传递机制之上构造。

TrainPlanDatabase 提供什么服务？就是所有有用的数据库操作：添加记录、删除记录、修改记录和查询记录。我们最终会捕获所有这些设计决策，以类的形式补充这个子系统。类提供所有这些操作的声明。

在设计的这个阶段，继续每个子系统的设计流程。再说一次，我们不指望这些接口一开始就恰好正确，必须允许它们随着时间进化。高兴的是，对于更小的对象，我们的经验表明，如果之前很好地以面向对象的方式刻画每个子系统的行为，那么需要对这些接口所做的大多

数变更是向上兼容的。

9.4 后移交

为了得到一个安全和有用的控制系统，工作需要一直进行。这不是说我们绝不会到达稳定系统的那个点——事实上，每一个移交我们都必须达到稳定。然而，现实是，对于处于业务中心的系统，例如铁路运输，在业务规则变化时，硬件和软件必须适应；否则，我们的系统会变成一项负债，而不是有竞争力的资产。虽然，对列车交通管理系统这样的系统，变化的主要风险是技术上的，但也有政治和社会风险。拥有一个有弹性的面向对象架构，开发组织至少为公司提供了很多自由度，使其能够敏捷地适应改变着的调控环境和市场。

对于列车交通管理系统，我们可以展望一条明显的附加需求，即薪水处理。具体地说，假设我们的分析展示了列车公司薪水处理当前由一款不再生产的硬件支持，我们有很大的风险失去薪水处理能力，因为单个严重的硬件故障会使会计系统永远瘫痪。因此，我们可能选择在列车交通管理系统中集成薪水处理。这两个看起来不相关的问题能如何共存不难构想，我们可以简单地把它们看作分离的应用，薪水处理作为背景活动运行。

进一步的调查展示，集成薪水处理确实能获得巨大的价值。可以回忆之前的讨论，列车计划包含关于人员分配的信息。因此，我们有可能跟踪真实的 vs. 计划的人员分配，这样，可以计算工作时、加班数量等。通过直接取得这个信息，薪水计算会更精确，当然，也更及时。

添加这个功能会怎样影响已有的设计？非常小。我们的方法是，在 UserApplications 子系统内再添加一个子系统，代表薪水处理功能。在架构中的这个位置，这样一个子系统对它所基于的所有重要机制都是可见的。实际上，这在结构良好的面向对象系统中十分通用：通过在已有的机制上建造新的应用，系统相当容易地对付明显的附加需求。

一个甚至更显著的变化是，通过建造一个调度员助理，把专家系统技术注入系统，该助理可以提供适当的交通路由和紧急响应建议。这将会如何影响系统架构？

影响非常小。我们的解决方案是在子系统 TrainPlanDatabase 和 DispatcherApplications 之间添加一个新的子系统。因为这个专家系统包含的知识库平行于 TrainPlanDatabase 的内容。此外，子系统 DispatcherApplications 是这个专家系统的唯一客户。我们需要开发一些新的机制来建立一种把建议展示给最终用户的方式。例如，可以使用一个黑板架构。

如果实现得好的话，架构的一个迷人特征就是它们将达成相当大量的关键功能以及适应性。换句话说，如果我们已经选择正确的元素功能和结构，会发现，用户很快会发现各种方法，以设计人员绝对想像不到或不希望的方式演进系统功能。当发现客户使用我们的系统的模式时，通过正式地使它们成为架构的一部分来编纂这些模式，是非常有意义的。设计良好的架构的一个迹象是，可以通过复用已有的机制引入这些新的模式，因此保持了它的设计完整性。

第 10 章

人工智能——密码分析

具有感知能力的生物向我们展示了一个庞大、复杂的行为集合，这些行为包括意识，还有我们知之甚少的原理机制。例如，想一想需要在某城市办一些差事时，你如何解决安排路线的问题。再想想，当你穿过一间昏暗灯光的房间时，怎样才能看清物体的轮廓而不至于跌倒。更进一步，在聚会上当许多人同时说话时，你怎样才能集中于其中的一次对话。这些问题还没有简捷的求解算法。最优路径规划是一个 NP 完全（NP-complete）问题。在黑暗地带导航涉及对模糊而不完备的可视输入（非常学术的说法）的理解。从众多的声源中鉴别出一个说话的人需要听者从噪音中辨别有意义的数据，然后从剩余的杂音中滤除不想要的会话。

人工智能领域的研究者已经着手研究这些问题以及相似的问题，以便提高我们对人类认知过程的理解。该领域的活动通常涉及模仿人类行为某些方面的智能系统的构造。Erman、Lark 和 Hayes-Roth 指出：

“在许多属性上，智能系统与传统系统不同：

- 它们追求的目标随时间而变化；
- 它们结合、使用和维护知识；
- 它们利用多种不同的特别的子系统，这些子系统包含一些经过选择的方法；
- 它们同用户和其他系统进行智能交互；
- 它们分配自己的资源和注意力。”[1]

任何一个特征都足以使得智能系统的构建成为一项非常困难的任务。当认识到正在为不同领域开发的智能系统影响到我们的生命财产安全时，例如医疗诊断或飞机航线，这个任务就变得更加迫切。因为我们必须设计这些系统，以保证没有任何危险发生——人工智能包含的不是一般意义上的知识。

虽然这个领域有时被热情过度的出版物过分推销，但是人工智能的研究的确给我们一些合理、实际的想法。在它们中间，我们注意到知识表示的方法，以及智能系统的通用问题解决架构的演化，包括基于规则的专家系统与黑板模型[2]。在本章中，我们将转向一个破解密

文的智能系统的设计，该系统使用与人类求解同样问题所用方法平行的黑板框架。正如我们将要看到的，面向对象开发非常适用于这个领域。

10.1 先启

我们的问题是一个密码分析问题，即把密文转换回明文的过程。在大多数通用形式下，解读密文是一个很难处理的问题，甚至需要挑战最精巧的技能。幸运的是，我们的问题相对简单，因为我们把范围限制为经过单次置换加密的密文。

10.1.1 密码分析需求

密码学“包含把难于理解的数据转变为非指定组合的各种方法”[3]。使用密码算法，消息（明文）可以被转化为密码（密文），然后再由密文转换回明文。

一种最基本的密码算法叫做置换加密，自从罗马时代起，该方法就一直被使用。使用这种加密方法，明码字母表的每一个字母都被映射到一个不同的字母。例如，可以将每一个字母转换为它的下一个字母：A 变为 B，B 变为 C，Z 绕回变为 A，等等。因此，明文

```
CLOS is an object-oriented programming language
```

可以被加密成密文

```
DMPT jt bo pckfdu-psjfoufe qsphsbnnjoh mbohvbhf
```

更经常的情况是，字母置换被打乱。例如，A 变成 G，B 变成 J，等等。例如，考虑如下密文：

```
PDG TBCER CQ TCK AL S NGELCH QZBBR SBAJG
```

提示：字母 C 代表明文字母 O。

仅仅使用一次置换来加密明文消息是相当简化的假设，然而，解码得到的密文在算法上并不是一项轻松的任务。解码有时需要实验，容许错误，通过假设特定的置换，估计它们的含义。例如，以密文中一两个字母的单词开始，假设它们代表普通的单词，如 I 和 a，或者 it、in、is、of、or 和 on。通过替换在密文中其他地方出现的这些字母，可以发现一些解码其他单词的暗示。例如，如果有一个以 o 开始的三字母的单词，可以推理这个单词可能是 one、our 或 off。

也可以用拼写和文法知识来破解置换加密。例如，双写字母的出现不可能代表序列 qq。类似地，可以尝试把以字母 g 结尾的单词扩展为后缀 ing。在更高的级别的抽象上，可以假设单词序列 it is 比 if is 更有可能出现。另外，一个典型的句子结构应当包括一个名词和一个动词。因此，假如已经识别出一个动词却没有执行者或代理，那么可能要开始搜寻形容词和名词。

有时我们可能不得不重做。例如，可能已经假设某个两字母的单词是 or，但是对于字母 r 的置换在其他单词中引起矛盾或者走进死胡同，我们就不得不尝试单词 of 或 on，然后撤消

先前的假定和基于它的置换。

这就得出了问题的主题需求：设计一个系统，给出一份密文，假设它仅仅经过一次置换加密，把它转换为原始的明文。

10.1.2 定义问题的边界

作为分析的一部分，让我们浏览一下求解一份简单密文的场景。花几分钟解决下面的问题，你求解的时候，记录下你是如何做的（事先阅读资料是不公平的）。

```
Q AZWS DSSC KAS DXZNN DASNN
```

提示：字母 W 代表明文 V。

尝试穷举搜索没有任何意义。假如明文字母表仅仅由 26 个英文大写字母组成，那么有 26!（大约 4.03×10^{26}）种可能的组合。因此，我们必须尝试一些技术而不是采取鲁莽的方式。一个可选的技术是基于对句子、单词、字母结构的知识做一个假设，然后自然地得到结果。一旦不能再往下走，就在第一个假设的基础上再选择一个更有希望的假设，如此等等。每个后续的假设会带我们逐渐逼近一个解决方案。如果发现进行不下去，或者得到一个与先前假设矛盾的结论，就必须返回并改变先前的假设。

下面是我们的解决方案，在每一步展示了结果。

（1）根据提示，可以直接把 W 置换为 V。

```
Q AZVS DSSC KAS DXZNN DASNN
```

（2）第 1 个单词较短，因此它可能是 A 或 I，假定它是 A。

```
A AZVS DSSC KAS DXZNN DASNN
```

（3）第 3 个单词需要一个元音，它可能是一个双字母。它可能既不是 II，也不是 UU，因为已经用了一个 A，所以它不能是 AA。因此，尝试 EE。

```
A AZVE DEEC KAE DXZNN DAENN
```

（4）第 4 个单词有 3 个字母，且以 E 结尾，它可能是 THE。

```
A HZVE DEEC THE DXZNN DHENN
```

（5）第 2 个单词需要一个元音，但是仅有一个 I、O 或 U（A 和 E 已经用过），只有 I 能给出有意义的单词。

```
A HIVE DEEC THE DXINN DHENN
```

（6）仅有几个 4 个字母的单词有 EE，包括 DEER、BEER 和 SEEN。语法知识暗示第 3 个单词应当是一个动词，因此，选择 SEEN。

```
A HIVE SEEN THE SXINN SHENN
```

（7）这个句子没有任何意义（hives cannot see），因此可能在以上过程中做了一个错误的假设。问题可能出在第 2 个单词的元音上，因此考虑修正初始的假设。

```
I HAVE SEEN THE SXANN SHENN
```

（8）现在对付最后一个单词。双字母不能是 SS（已经用了一个 S，并且 SHESS 没有任何意义），但是 LL 构成一个有意义的单词。

```
I HAVE SEEN THE SXALL SHELL
```

(9)第5个词是名词短语的一部分，因此它可能是一个形容词(例如，STALL不予考虑)。搜索适合模式S?ALL的单词，得到SMALL。

I HAVE SEEN THE SMALL SHELL

因此，我们得到解答。

对于这个问题的解决过程，可以给出以下简评。

- 应用了许多不同的知识源，比如语法、拼写和元音。
- 把假设记录在一个中央位置，然后把知识源应用到这些假设上，推测它们的结果。
- 进行推理，多少靠点运气。我们有时用从一般到特殊的规则推理（如果单词由3个字母组成并且以E结尾，它可能是THE），有时从特殊到一般推理（?EE?可能是DEER、BEER或SEEN，但是既然这个词必须是动词而不是名词，所以仅SEEN满足假设）。

通过观察这些问题求解，可以识别一些关键抽象。关键抽象是开始建立初始架构框架时，解决方案的分析元素。以上三点识别出多个知识源、一个放置假设或假定的中央位置和一个有条件控制问题求解的控制组件。

前面描述的问题求解的方法被称为黑板模型（blackboard model）。黑板模型在1962年由Newell首次提出，后来，Reddy和Erman把它合并到Hearsay和Hearsay II项目中——这两个项目处理语音识别问题[4]，黑板模型被证明在这个领域用处很大。不久，这个框架成功被应用到其他领域，包括信号解释、三维分子结构建模、图像识别和规划[5]。在陈述性知识的表示方面，黑板框架被证明有特殊价值，与其他方法[6]相比有更高的空间与时间效率。

黑板框架是一个可以作为问题求解算法的分析结果来应用的架构模式。这个框架可以用类和描述类的实例如何协作的机制来表达。

10.1.3 黑板框架的架构

Englemore和Morgan把黑板模型比作一组人解决一个智力拼图问题：

“设想在一个有一块大黑板的房间里，有一组人围着黑板，每人手上拿着特大的拼图块。首先，人们自愿将他们的那些拼块放在黑板（假设黑板可粘贴）最有“希望”的位置。小组中的每一个人都会看看自己的拼块是否适合黑板上已有的块。拿着合适拼块的人会走上黑板，更新以前的安排方式。更新使得其他拼块依次排列，其他人拿着他们的块走到黑板，放上拼块。一个人拥有的拼块多少无关紧要。在完全沉默的情况下他们就可以完成拼图，也就是说，不需要直接沟通。每个人自主活动，了解什么时候他的拼块用得上。对人们走上黑板拼图的顺序不做规定，他们相互之间的合作由黑板上解的状态来协调。如果观察完成任务的过程，可以发现求解是逐步达到的（每次拼一块），而且是凭机会（出现增加一块的机会）而不是系统化地从某个地方开始（例如，从左上角开始，试拼每一块）的”[7]。

如图10-1所表明的，黑板框架由三个元素组成：一块黑板、多个知识源和一个协调这些知识源的控制器[8]。注意下面的陈述如何描述从问题空间识别出来的关键抽象。根据Nii的解释，“黑板模型的目的是保持知识源产生和需要的计算数据与解状态数据。黑板由解空间的

对象组成。黑板上的对象按照层次被组织成多个分析级别。对象以及它们的属性定义解空间的词汇表”[9]。

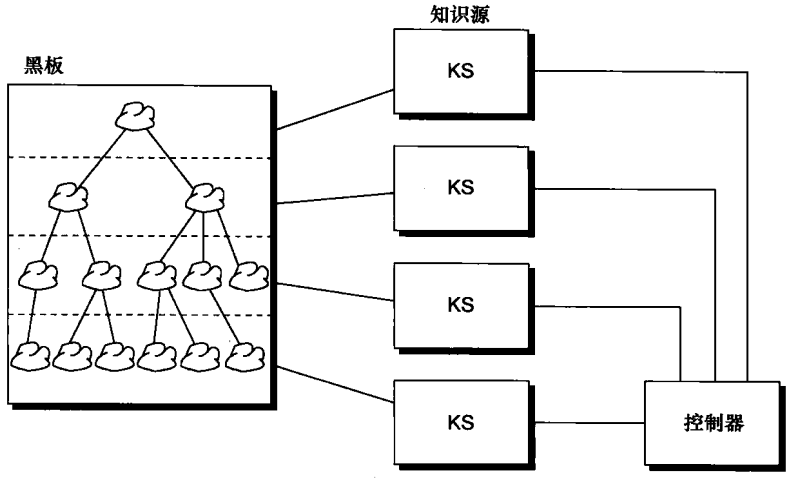


图 10-1 一个黑板框架

Englemore 和 Morgan 解释道：“解决一个问题所需要的领域知识被分成许多分离和独立的知识源。每一个知识源的目标就是贡献会导致问题解的信息。一个知识源使用黑板上的一组当前信息，并且通过加入它的特殊知识来更新黑板。知识源以过程、规则集或逻辑断言的形式表现。”[10]

知识源（简写为 KS）与特定领域相关。在语音识别系统中，知识源可能包括能对音素、语素、单词和句子推理的代理。在图像识别系统中，知识源包括可识别一些简单图像元素（如相似纹理的边缘和区域）的代理，以及表示每一个场景中感兴趣对象（如房屋、道路、田野、汽车和人）的高层抽象。一般说来，知识源与黑板上的对象的层次结构类似。进一步地，每一个知识源使用在一个级别上的对象作为它的输人，然后产生或修改另一级别上的对象作为输出。例如，在一个语音识别系统中，包含单词知识的知识源可以查看音素流（在低的抽象级别上）组成一个新的单词（在更高的抽象级别上）。另外，包含句子结构知识的知识源可能假设需要一个动词（在高级抽象级别上）；通过过滤一系列可能的单词（在较低的级别上），知识源可以验证这个假设。

这两种推理的方法分别代表前向链与后向链。前向链包括从具体断言到概括断言的推理，后向链从一个假设开始，然后从用已知的断言验证假设。知识源依赖环境，它可能被选择激活，用到前向链或后向链中，这就是为什么我们说黑板模型中的控制是凭机会的。

知识源通常包含两个要素，即前置条件和动作。知识源的前置条件代表知识源感兴趣的黑板的状态。例如，图像识别系统中一个知识源的前置条件可能是发现一个相对线形区域的图像元素(可能代表一条路)。触发一个前置条件导致知识源把注意力集中到黑板的这个部分，然后知识源通过执行自己的规则与过程知识采取行动。

在这种环境下，轮询是没有必要的：当一个知识源认为有一些感兴趣的东西值得贡献时，它将通知黑板控制器。形象地说，这就像每个知识源举手表示它有一些有用的事情要做，控制器从这些迫切的知识源中选择最有希望的一个。

10.1.4 知识源的分析

让我们回到具体的问题，考虑对求解有贡献的知识源。最好的策略是与这个领域的专家坐在一起，记录此人用于求解该领域问题的启思录。这种做法在大多数知识工程应用中较为典型。对于目前的问题，这种做法可能涉及求解许多密码，并需要记录下沿途的思考过程。

分析表明有 13 个知识源与此相关，它们和所包含的知识如下：

- 常用前缀（Common prefixes） 以 `re`、`anti`、`un` 等开头的常用单词；
- 常用后缀（Common suffixes） 以 `ly`、`ing`、`es` 和 `ed` 等结尾的常用单词；
- 辅音字母（Consonants） 非元音字母；
- 直接置换（Direct substitution） 给出提示作为问题陈述一部分；
- 双字母（Double letters） 常见双字母，如 `tt`、`ll` 和 `ss`；
- 字母出现频率（Letter frequency） 每一个字母出现的频率；
- 合法字符串（Legal strings） 合法与非法的字母组合，如 `qu` 和 `zg`；
- 模式匹配（Pattern matching） 匹配一特定字母模式的单词；
- 句子结构（Sentence structure） 语法，包括名词短语和动词短语的含义；
- 短单词（Small words） 可能匹配 1 个字母、2 个字母、3 个字母和 4 个字母单词；
- 破解（Solved） 问题是否被破解，或者是否不可能再继续进程；
- 元音字母（Vowels） 非辅音字母；
- 单词结构（Word structure） 元音的位置及名词、动词、形容词、副词、冠词、连词等的常见结构。

从面向对象的视角看，13 个知识源中的每一个都代表一个架构中的候选类：每个实例体现某些状态（它的知识），每个实例展示某个类的具体行为（后缀知识源可以对那些被怀疑有常见结尾的单词做出反应），并且，每个实例具有唯一标识（短单词知识源独立于模型匹配知识源而存在）。

也可以按层次排列这些知识源。具体地说，一些知识源作用于句子，一些知识源作用于字母，还有一些知识源作用于连续的字母组，最低层的知识源作用于单个字母。实际上，这个层次反映的是可能出现在黑板上的对象：句子、单词、字母串与字母。

10.2 精化

现在我们准备使用已经描述的黑板框架来设计一个密码分析问题的解决方案。这是一个经典的大范围重用的例子，因为我们可以重用一个已被验证的架构模式作为设计的基础。

黑板框架的架构表明：在我们的系统中，一块黑板、几个知识源与一个控制器是最高级别对象的一部分。下一步的任务是识别领域特定的类和对象，这些类和对象是一般关键抽象的具体化。

10.2.1 黑板对象

黑板是一个不同级别抽象的精化结构。该抽象作为对象被捕获，以层次的方式出现在黑板结构中。层次对象结构和不同级别的知识源抽象是平行的。知识源使用黑板作为输入数据、局部求解、替代选择、最后解答和控制信息的全局来源。

为了开始设计黑板的层次结构，需识别以下三个类：

- `Sentence` 完整的密文；
- `Word` 密文中的单词；
- `CipherLetter` 单词的单个字母。

知识源也必须分享每个知识源所做假设的知识，因而应当包括下面的黑板对象类：

- `Assumption` 知识源所做的假设。

最后，重要的是知道字母表中哪些明文和密文字母已经被用在知识源所做的假设中。因此，包括如下的类：

- `Alphabet` 明文字母表、密文字母表和两者之间的映射。

这 5 个类中是否有公共的东西？回答是肯定的。每一个类代表一些可能被放置于黑板上的对象，这些对象的属性将它们区别于知识源和控制器。因而，创建 `BlackboardObject` 类，作为可能出现在黑板上的每一个对象的超类。图 10-2 展示了我们对 `Blackboard` 抽象的初步设计。

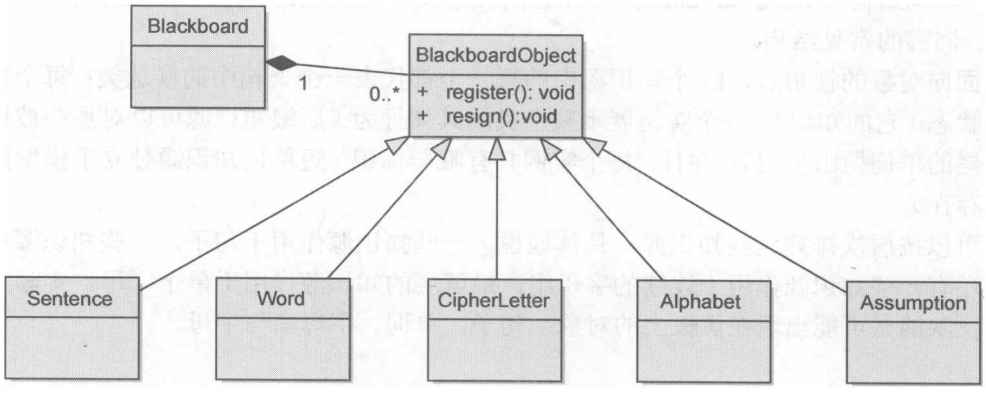


图 10-2 初步的 `Blackboard` 类图设计

从 `BlackboardObject` 的外部视图来看这个类，可以定义两个有用的操作：

- `register` 增加对象到黑板；
- `resign` 从黑板移除对象。

为什么定义 register 和 resign 作为对 BlackboardObject 实例的操作，而不是作为对 Blackboard 自身的操作呢？这和告诉对象将自己绘在一个窗口里没什么不同。决定将这种操作放在何处的石蕊测试是类自身是否有足够的知识或责任执行该操作。Register 和 resign 确实就是这种情况。BlackboardObject 是唯一有这种详细知识的抽象：知道如何将自己与 Blackboard 贴上或分离。（虽然，它当然确实需要和 BlackboardObject 协作）。事实上，每一个 BlackboardObject 在被贴到黑板时，应当自我意识到，它是这个抽象的一个重要责任。因为只有这样，它才能开始参与，凭机会解决 Blackboard 上的问题。

10.2.2 依赖和认定

单个句子、单词和密文字母有另一个共同的东西：每个都有依赖于它们的知识源。一个给定的知识源可能对这些对象中的一个或多个表现出兴趣，因此，一个句子、单词或密文字母必须维持对每一个这种知识源的引用，以便当关于该对象的假设改变时，能够通知相应的知识源一些有趣的事情已经发生。为了提供这个机制，引入一个简单的抽象类：Dependent。

为了设计 Dependent 类，我们包含一个对象，代表一个知识源集合：

- references 知识源的集合。

此外，为该类定义以下操作：

- add 增加一个到知识源的引用；
- remove 移除一个到知识源的引用；
- numberOfDependents 返回依赖的数目；
- notify 广播每个依赖的一个操作。

操作 notify 有被动迭代器的语义，这意味着当调用它的时候，可以提供一个希望在集合中每个依赖上进行的操作。

依赖是一个可以与其他类“混合”的独立属性。例如，一个 CipherLetter 既是一个 BlackboardObject，也是一个 Dependent，因此我们可以合并这两个抽象来达到想要的行为。以这种方式使用一个抽象类，提升了架构中的可重用性和关注分离。

CipherLetter 和 Alphabet 有另一个共同的属性：这两个类的实例可以有关于它们自己的假设（同时记住，一个 Assumption 对象也是一种 BlackboardObject）。例如，某个知识源可能假设密文字母 K 代表明文字母 P。随着逐步接近求解结果，我们可能作出不可改变的断言——G 代表 J。因此，需要包括一个维持相关联对象的假设和断言的类。我们把这个类识别为 Affirmation。

在我们的架构中，仅对 CipherLetter 和 Alphabet 中的单个字母做认定。正如早先的场景所暗示的，密文字母代表可以做陈述的单个字母，字母表由许多字母组成，对每一个字母可能有不同的陈述。定义 Affirmation 为一个独立的类，从而捕捉这两个不同类的共同行为。

我们为 Affirmation 的实例定义以下操作：

- make 做一个陈述；

- retract 收回一个陈述；
- ciphertext 给定一个明文字母，返回它的密文等价物；
- plaintext 给定一个密文字母，返回它的明文等价物。

进一步的分析表明，我们应当清楚地区分由一个陈述所扮演的两个角色：假设，代表在密文字母与明文等价物之间的临时映射关系；断言，一个持久的映射，这意味着映射被定义，因此是不可变的。解密的过程中，知识源将做许多假设，随着逐渐靠近最终结果，这些映射最终变为断言。对这些变化的角色建模，我们需要重新定义先前标识的类 Assumption，同时引入一个新的名为 Assertion 的子类，这两个类的实例均被类 Affirmation 的实例管理，同时被放在黑板上。从完成操作 make 和 retract 的签名开始，把一个 Assumption 或 Assertio 参数包括在内，然后，增加以下选择器。

- isPlainLetterAsserted 选择器：明文字母被定义了吗？
- isCipherLetterAsserted 选择器：密文字母被定义了吗？
- plainLetterHasAssumption 选择器：有一个关于明文字母的假设吗？
- cipherLetterHasAssumption 选择器：有一个关于密文字母的假设吗？

Assumption 对象是一种 BlackboardObject，因为它们代表对所有知识源感兴趣的陈述。需要声明成员对象来代表以下属性。

- target 对其进行假设的黑板对象；
- creator 创建假设的知识源；
- reason 知识源进行假设的理由；
- plainLetter 对其进行假设的明文字母；
- cipherLetter 明文字母的假定值。

对每一个这些属性的需要，主要来源于一个假设的本质：特定的知识源关于明文/密文映射作出一个假设，这样做是基于某个原因(通常因为触发了某些规则)。对第一个成员 target 的需要不太显著。之所以包括它，是因为涉及回溯问题。如果不得不逆转一个假设，必须通知所有最初对它作出假设的黑板对象，以便它们可以警告所依赖的（经由依赖机制）知识源，它们的意义已经改变。

接下来，声明 Assumption 的子类，将其命名为 Assertion。类 Assumption 和 Assertion 共享以下操作：

- isRetractable 选择器：映射是临时的吗？

对于谓词 isRetractable，所有的 Assumption 对象都回答为真，但所有 Assertion 对象都回答为假。另外，一旦作出一个断言，它既不能被重新陈述，也不能被收回。

图 10-3 提供了一张类图，展示了 Dependent 和 Affirmation 类之间的协作。要特别注意每一个抽象在不同的关联中所扮演的角色。例如，一个 KnowledgeSource 是一个 Assumption 的创建者，也是一个 CipherLetter 的 referencer。与一个抽象相比，一个角色代表针对世界的一个不同的视图，所以我们期望看到知识源与假设之间以及知识源与字母之间有着不同的协议。

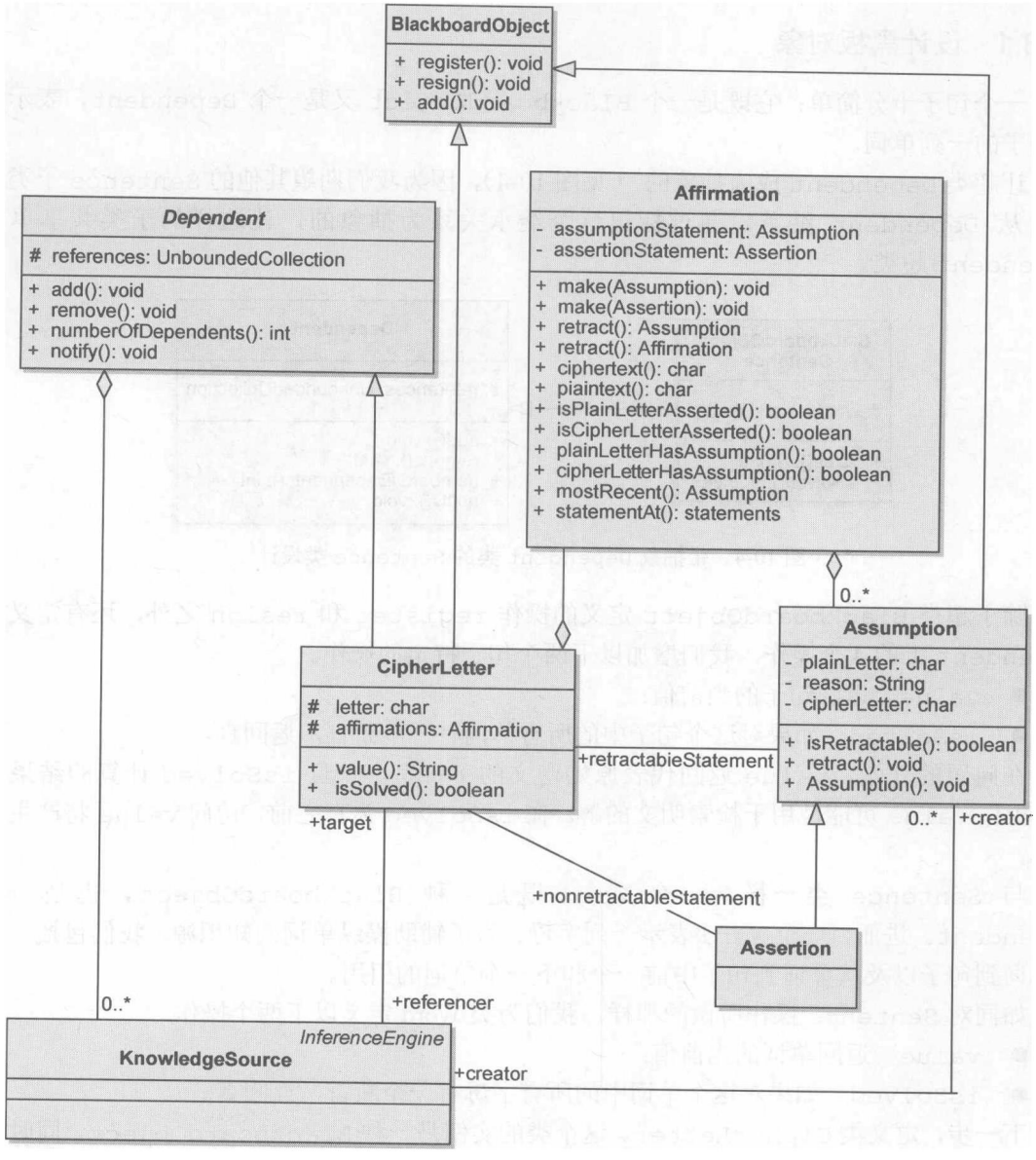


图 10-3 Dependency 和 Affirmation 类

10.3 构造

在 Alphabet 类之后，让我们通过做一小点隔离的类设计，继续设计 Sentence、Word 和 CipherLetter 类。

10.3.1 设计黑板对象

一个句子十分简单：它既是一个 BlackboardObject 又是一个 Dependent，表示组成句子的一列单词。

让超类 Dependent 成为抽象的[1]（见图 10-4），因为我们期望其他的 Sentence 子类也尽量从 Dependent 继承。通过标记这个继承关系为抽象的，让这样的子类共享单个 Dependent 超类。

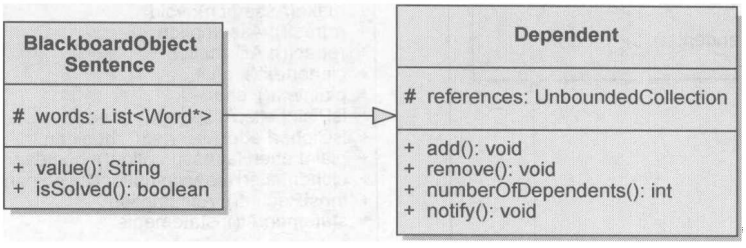


图 10-4 带抽象 Dependent 类的 Sentence 类设计

除了超类 BlackboardObject 定义的操作 register 和 resign 之外，还有定义在 Dependent 上的 4 个操作，我们增加以下两个句子特定的操作。

- value 返回句子的当前值；
- isSolved 如果对这个句子中的所有单词有一个断言，返回真。

在问题的开始，value 返回代表原始密文的字符串。一旦 isSolved 计算的结果为真，操作 value 可能被用于检索明文的解。在 isSolved 为真之前，访问 value 将产生部分解。

与 Sentence 类一样，一个 Word 既是一种 BlackboardObject，也是一种 Dependent。进而，一个 Word 表示一列字母。为了辅助操纵单词的知识源，我们包括一个从单词到句子以及从单词到句子中前一个和下一个单词的引用。

如同对 Sentence 操作所做的那样，我们为类 Word 定义以下两个操作。

- value 返回单词的当前值；
- isSolved 如果对这个单词中的所有字母有一个断言，返回真。

下一步，定义类 CipherLetter。这个类的实例是一种 BlackboardObject，同时也是一种 Dependent。除了继承的行为之外，每个 CipherLetter 对象都有一个值（如密文字母 H）和一个关于它相应明文字母的假设和断言的集合。可以用类 Affirmation 来收集这些陈述。图 10-5 所示为添加了 CipherLetter 和 Word 设计之后的架构框架。

注意：我们包括选择器 value 和 isSolved，这与 Sentence 和 Word 的设计类似。最后，还要为 CipherLetter 的客户提供操作，在安全方式下存取它的假设和断言。

[1] 在 UML2.0 中，抽象类用斜体的类名表示。也可以在属性列表中放上{abstract}关键字。

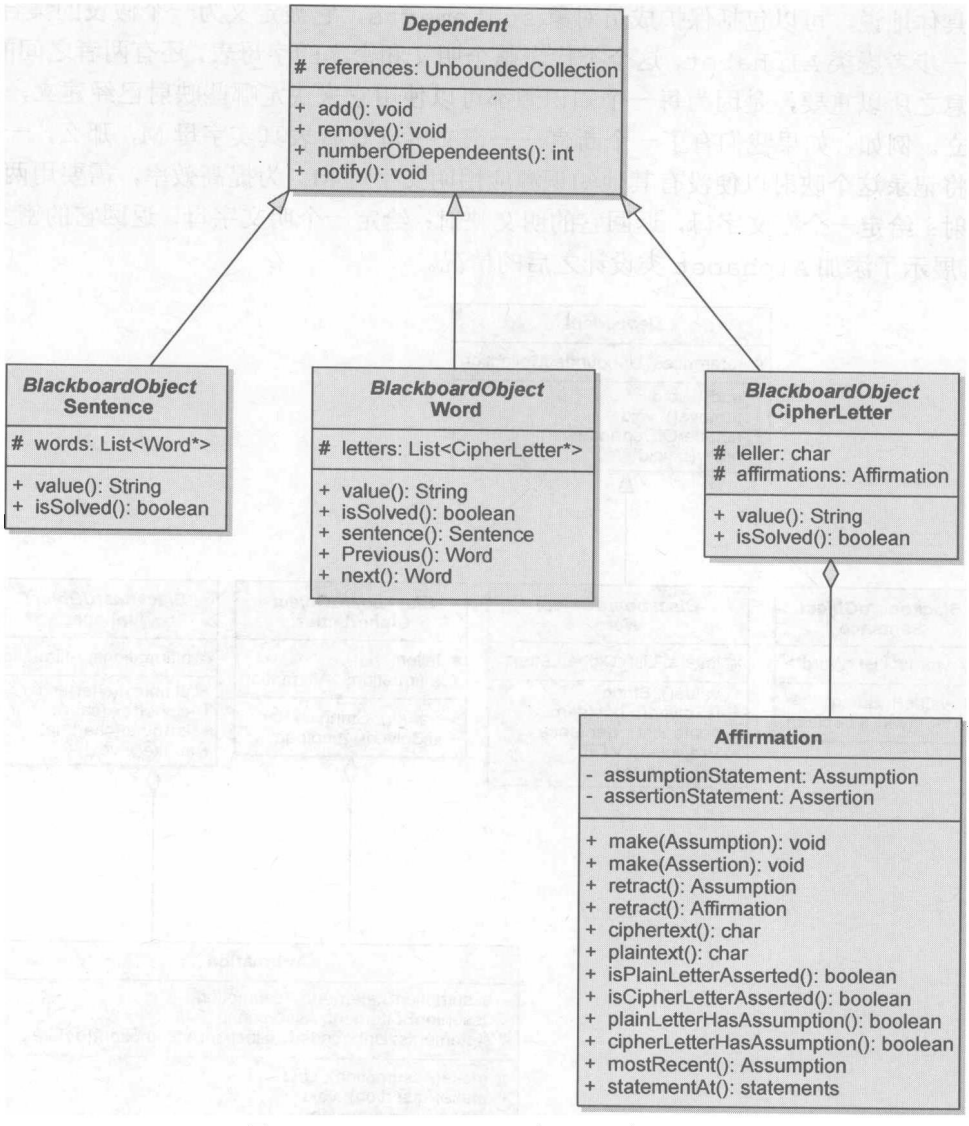


图 10-5 CipherLetter 和 Word 类的设计

关于成员对象认定（affirmation）有一个注释：我们希望这是一个假设和断言的集合，根据创建时间排序，集合中最近的陈述代表当前的假设和断言。选择保持所有的假设历史是因为，我们允许知识源可以看到之前被拒绝的假设，这样它们可以从之前的错误中学习。这个决定影响 Affirmation 类的设计决策，我们对其添加以下操作：

- mostRecent 选择器：返回最近的假设或断言；
- statementAt 选择器：返回第 n 个陈述。

现在已经重定义了它的行为，下一步，关于 Affirmation 类，可以做一个合理的实现

决定。具体地说，可以包括保护成员对象 statements，它被定义为一个假设的集合。

下一步考虑类 Alphabet。这个类代表整个明文和密文的字母表，还有两者之间的映射。这个信息之所以重要，是因为每一个知识源都可以使用它来决定哪些映射已经建立，哪些还有待建立。例如，如果我们有了一个断言——密文字母 C 代表真实字母 M，那么，一个字母表对象将记录这个映射以便没有其他知识源应用明文字母 M。为提高效率，需要用两种方式查询映射：给定一个密文字母，返回它的明文映射；给定一个明文字母，返回它的密文映射。图 10-6 展示了添加 Alphabet 类设计之后的情况。

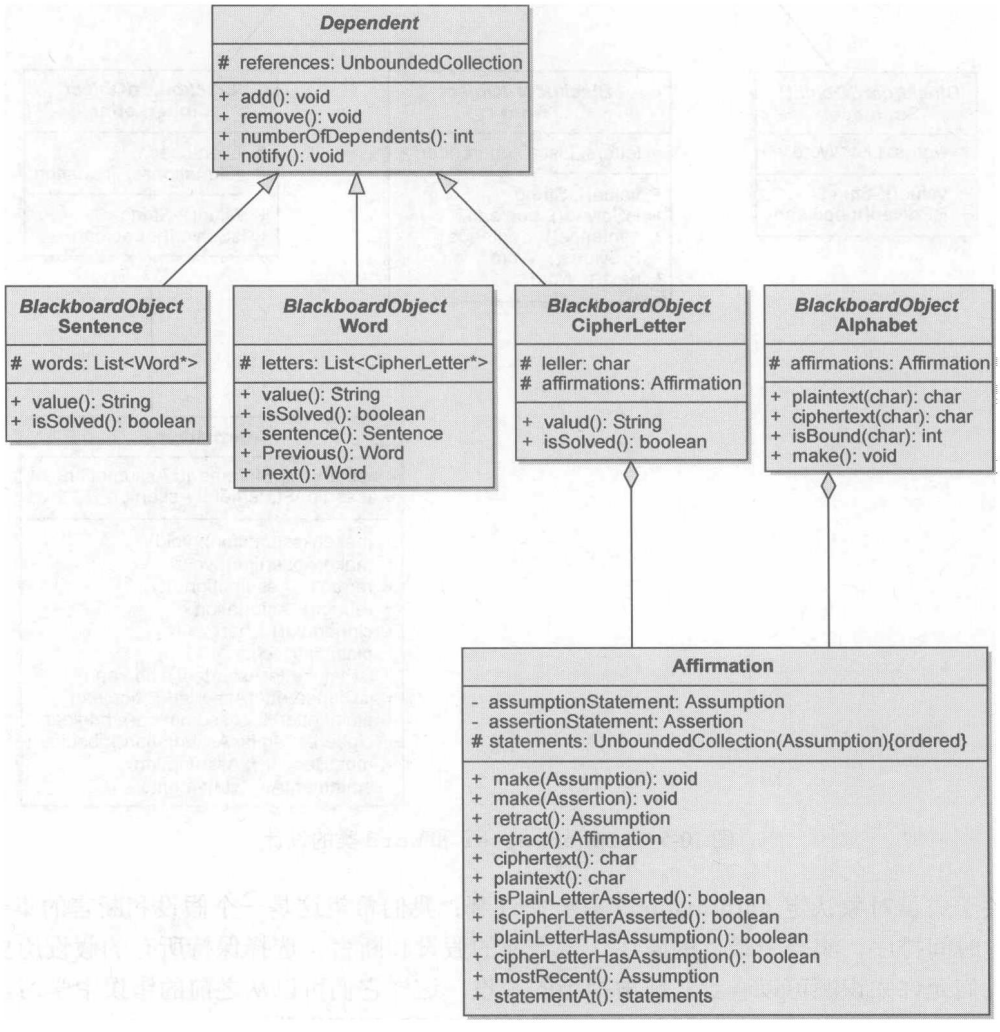


图 10-6 Alphabet 类的设计

就像 CipherLetter 类一样，也包括一个保护成员对象 affirmations，并提供适当的操作来访问它的状态。

现在，我们准备定义 Blackboard 类。这个类有收集 BlackboardObject 类及其子类实例的简单责任。这样，可以设计 Blackboard 为 DynamicCollection 实例的一种类型。我们选择从类 DynamicCollection 继承，而不是包含它的一个实例，因为 Blackboard 传递我们对继承的测试——Blackboard 的确是一种集合。

Blackboard 类提供如 add 和 remove 这样的操作，它从 Collection 类继承这些操作。我们的设计包括 5 个黑板特定的操作：

- reset　擦干净黑板；
- assertProblem　在黑板上放上初始问题；
- connect　将知识源附加到黑板上；
- isSolved　如果句子被破解，返回真；
- retrieveSolution　返回解出的明文句子。

针对第二个操作，需要在黑板与它的知识源之间建立一个依赖。

图 10-7 中归纳了一个与 Blackboard 协作的类的设计。在这个图中，注意，Blackboard

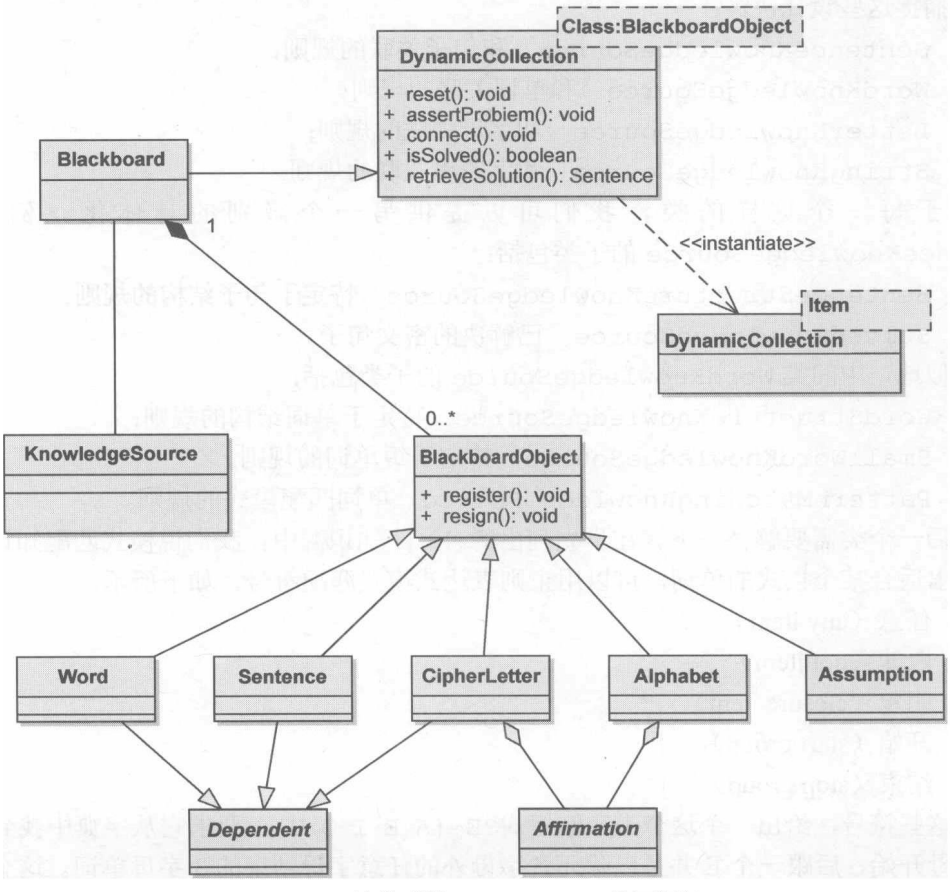


图 10-7　精化后的 Blackboard 类图设计

类可以从模板类 DynamicCollection 实例化得到，也可以从它继承而来。图中也清楚地展示了为什么把类 Dependent 作为抽象类是一个好的设计决定。具体地说，Dependent 代表一个仅包含部分 BlackboardObject 子类的行为。通过使 Dependent 成为抽象的，我们增加了复用它的机会。

10.3.2 设计知识源

在前一节，我们识别了 13 个与此问题相关的知识源。正如为 Blackboard 对象所做的，可以设计包含这些知识源的类结构，从而把所有的公共特征提升到更高的抽象类。

1. 设计特定知识源

假设存在一个叫作 KnowledgeSource 的抽象类，它的目的更像类 BlackboardObject。比起将 13 个知识源的每一个作为一个更通用类的直接子类来，首先进行一个领域分析是有用的，看看是否存在成簇的知识源。实际上，有这样一些组：一些知识源操作于整个句子，一些操作于整个单词，一些操作于相邻的字母串，还有一些操作于单个字母。可以通过创建以下子类捕捉这些设计决策：

- SentenceKnowledgeSource 和句子关联的规则；
- WordKnowledgeSource 和单词关联的规则；
- LetterKnowledgeSource 和字母关联的规则；
- StringKnowledgeSource 和字符串关联的规则。

对于每一个这样的类，我们可以提供另一个级别的具体化。例如，类 SentenceKnowledgeSource 的子类包括：

- SentenceStructureKnowledgeSource 特定于句子结构的规则；
- SolvedKnowledgeSource 已解决的密文句子。

类似地，中间类 WordKnowledgeSource 的子类包括：

- WordStructureKnowledgeSource 特定于单词结构的规则；
- SmallWordKnowledgeSource 特定于短单词的规则；
- PatternMatchingKnowledgeSource 单词匹配模式的规则。

最后一个类需要解释一下。在早先列出的 13 个知识源中，我们说模式匹配知识源的目的是找出适合某个模式的单词。可以用正则表达式模式匹配符号，如下所示。

- 任意（any item） ?
- 取非（not item） ~
- 闭包（closure item） *
- 开始（start group） {
- 结束（stop group） }

用这些符号，给出一个这个类实例模式?E~{A E I O U}，要求它从字典中找到所有以任意字母开始、后跟一个 E 并且以除元音字母外的任意字母结束的 3 字母单词。这个类的所有实例分享一个单词字典，并且每个实例都有它自己的正则表达式模式匹配代理。在设计中，

这个类的细节行为对于我们并不重要，因此，我们推迟发现它剩下的接口和实现。

接下来，可以声明类 String KnowledgeSource 的以下子类：

- CommonPrefixKnowledgeSource 特定于前缀的规则；
- CommonSuffixKnowledgeSource 特定于后缀的规则；
- DoubleLetterKnowledgeSource 双字母的规则，如 oo、ll 等；
- LegalStringKnowledgeSource 特定于成为合法字符串的规则。

最后，引入类 LetterKnowledgeSource 的以下子类：

- DirectSubstitutionKnowledgeSource 特定于字母置换的规则；
- VowelKnowledgeSource 特定于元音的规则；
- ConsonantKnowledgeSource 特定于辅音的规则；
- LetterFrequencyKnowledgeSource 特定于字母频率的规则。

图 10-8 展示了 KnowledgeSource 的层次结构。

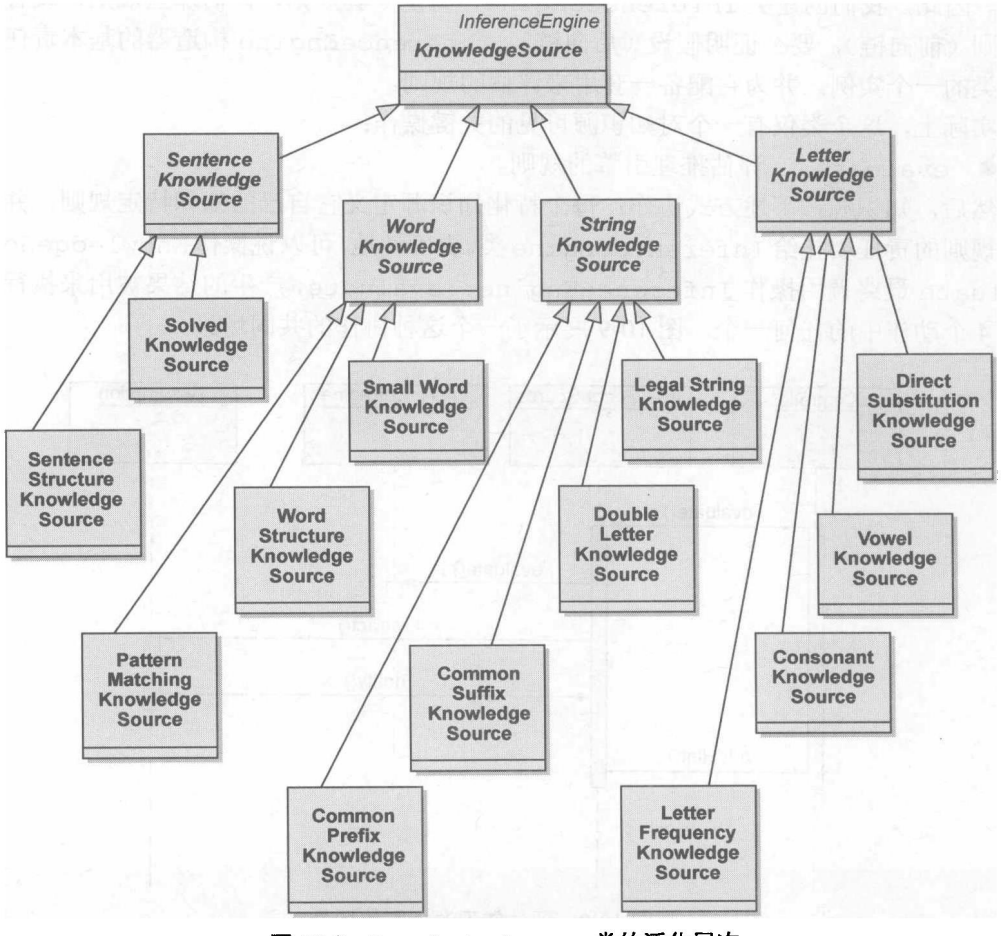


图 10-8 KnowledgeSource 类的泛化层次

2. 泛化知识源

分析表明，仅有两个主要的操作应用到所有这些特化类：

- `reset` 重新开始知识源；
- `evaluate` 估计黑板的状态。

接口简单的原因是，知识源是相对自治的实体：把一个知识源指向一个感兴趣的 `Blackboard` 对象，然后告诉它根据当前 `Blackboard` 的全局状态来评估它的规则。作为规则评估的一部分，一个给定知识源可能做以下任何一件事情。

- 为置换加密提出一个假设；
- 在先前的假设中发现矛盾，并使矛盾的假设被收回；
- 为置换加密提出断言；
- 告诉控制器有某些令人感兴趣的知识要贡献。

这些是所有独立于特定种类知识源的通用动作。进一步泛化，这些动作代表推理引擎的行为。因此，我们创建类 `InferenceEngine`，给定一套规则，评估那些规则，或者产生新的规则（前向链），要么证明假设（后向链）。`InferenceEngine` 构造器的基本责任是构造这个类的一个实例，并为它配备一套用来评估的规则。

实际上，这个类仅有一个对知识源可见的关键操作：

- `evaluate` 评估推理引擎的规则。

然后，知识源按下述方式协作：每个特化知识源定义它自己的知识特定规则，并将评估这些规则的责任委托给 `InferenceEngine` 类。更确切地，可以说操作 `KnowledgeSource::evaluate` 最终调用操作 `InferenceEngine::evaluate`，产生的结果被用来执行先前列出的 4 个动作中的任何一个。图 10-9 展示了一个这种协作的共同场景。

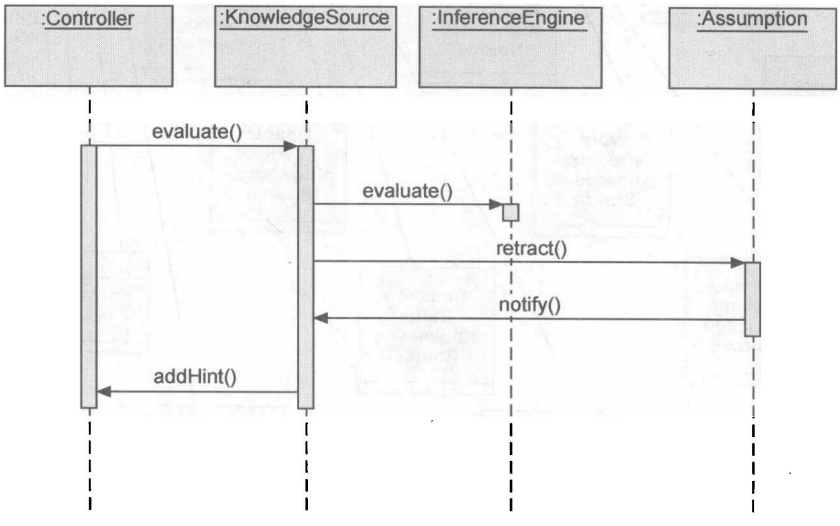


图 10-9 评估知识源规则的场景

序列图展示了这个场景中的以下步骤：

（1）为动作选择一个 KnowledgeSource；

（2）针对 Blackboard 的状态评估 KnowledgeSource；

（3）做某些动作，例如，撤回一个 Assumption；

（4）通知所有依赖的 KnowledgeSource 对象假设已经撤回；

（5）告诉 Controller，KnowledgeSource 有一个新的提示来解决黑板问题。

一条规则究竟是指什么？使用一种认识常见后缀模式的模式匹配算法，如*I??，一条规则可以表达为常见的后缀知识源。给定一个与正则表达式模式*I??匹配的字母串，候选的后缀可能包括 ING、IES 和 IED。

就它的类结构而言，可以说一个知识源就是一种推理引擎。除此之外，每个知识源必须与一个黑板对象有某些关联，因为在那里它发现要操作的对象。最后，每个知识源必须与一个控制器有关联，通过发送求解暗示来协作；反过来，控制器可以一次接一次地触发知识源。图 10-10 展示了这些设计决定。

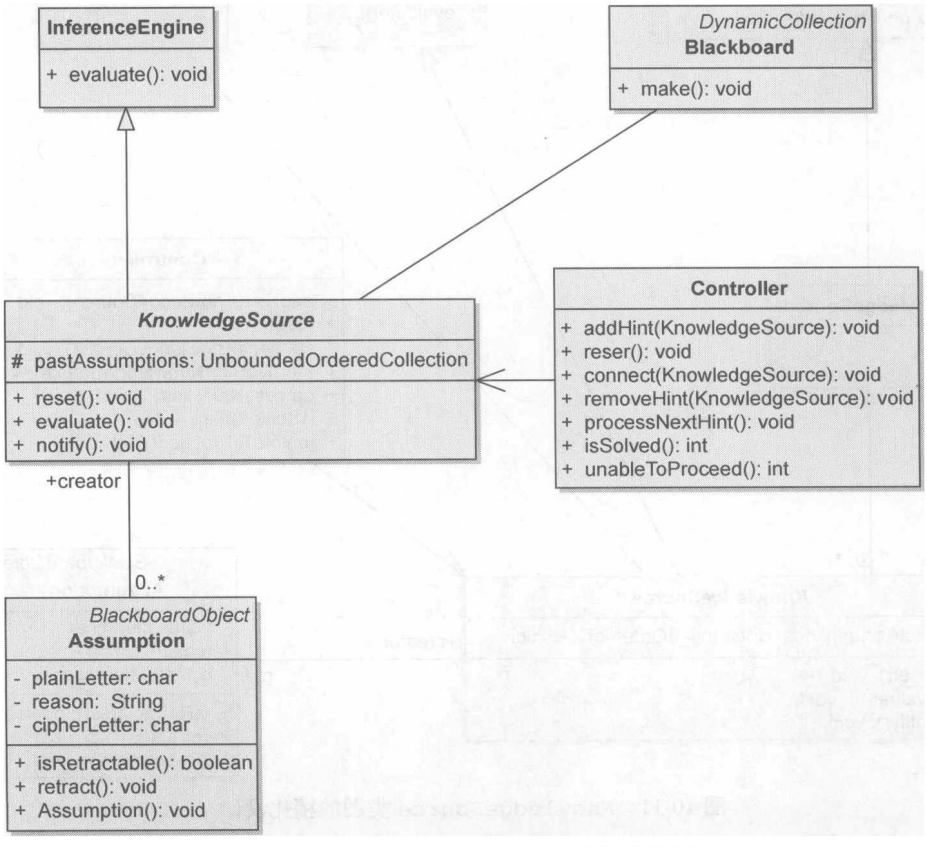


图 10-10 KnowledgeSource 的初步设计

我们还引入了集合 pastAssumptions，以便知识源能够跟踪已经做过的所有假设和断言，从而从错误中学习。

类 Blackboard 的实例被用作 Blackboard 对象的仓储。出于类似的原因，对于一个特定的问题，我们需要一个 KnowledgeSources 类，来表示整个知识源集合。

这个类的职责之一是当创建类 KnowledgeSources 的一个实例时，也创建 13 个单独的 KnowledgeSource 对象。可以在这个类的实例上执行以下三个操作：

- restart 重启知识源；
- startKnowledgeSource 给出一个特定知识源的初始条件；
- connect 把知识源附加到黑板或控制器。

根据这些设计决定，图 10-11 提供了 KnowledgeSource 类结构的精化设计。

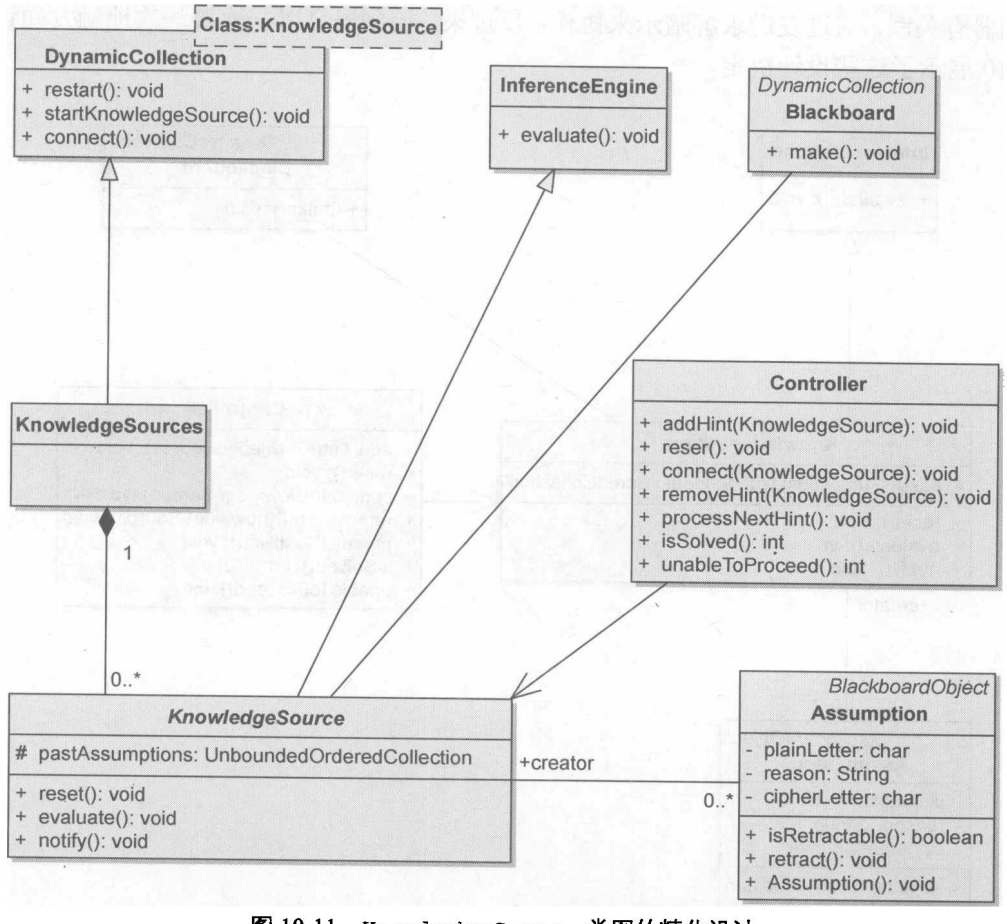


图 10-11 KnowledgeSource 类图的精化设计

10.3.3 设计控制器

考虑一下控制器与单个知识源是如何交互的。在解密的每一阶段，一个特定的知识源可能发现有一些有用的东西值得贡献，它给控制器一个提示。相反，知识源可能决定它早期的提示不再适用，从而将其移除。一旦所有的知识源有一个机会，控制器就选择最有希望的提示，通过调用 evaluate 操作激活合适的知识源。

控制器如何决定激活哪一个知识源？我们可以设计一些合适的规则：

- Assertion 优先级比 Assumption 高；
- SolvedKnowledgeSource 提供最有用的提示；
- PatternMatchingKnowledgeSource 提供比 SentenceStructureKnowledgeSource 更高优先级的提示。

控制器扮演着一个代理的角色，它负责协调在黑板上进行操作的各个知识源。

控制器必须关联到它的知识源。通过这个知识源，控制器可以访问相应命名的类 KnowledgeSources。另外，控制器必须有一个根据优先级规则排序的提示集作为它的属性。按照这种方式，控制器可以容易地选择激活带有最感兴趣提示的知识源。

进入一个略为孤立的类的设计，为 Controller 类提供以下操作：

- reset 重启控制器；
- addHint 增加一个知识源提示；
- removeHint 移除一个知识源提示；
- processNextHint 评估下一个最高优先级的提示；
- isSolved 选择器：如果问题获得解决，返回真；
- unableToProceed 选择器：如果知识源被阻塞，返回真；
- connect 把控制器附加到知识源。

在某种意义上，控制器受从各知识源中收到的提示驱动。这样，有限状态机非常适合于捕捉这个类的动态行为。

例如，考虑图 10-12 展示的状态转换图。这里可以看到，控制器可能是以下 5 个主要状态之一：Initializing、Selecting、Evaluating、Stuck 和 Solved。控制器最有兴趣的活动发生在 Selecting 和 Evaluating 状态之间。选择时，控制器自然地从状态 Creating Strategy 转换到 Processing Hint，最终到 Selecting KS 状态。如果一个知识源事实上被选中，那么控制器将转换到 Evaluating 状态，在其中它首先处于 Updating Blackboard 状态。如果增加对象，控制器将转换到 Connecting 状态；如果假设被撤回，控制器将转换到 Backtracking 状态，这时它也通知所有的依赖者。

如果不能继续进行，控制器就无条件地转换到 Stuck 状态；如果发现一个已求解的黑板问题，就转换到 Solved 状态。

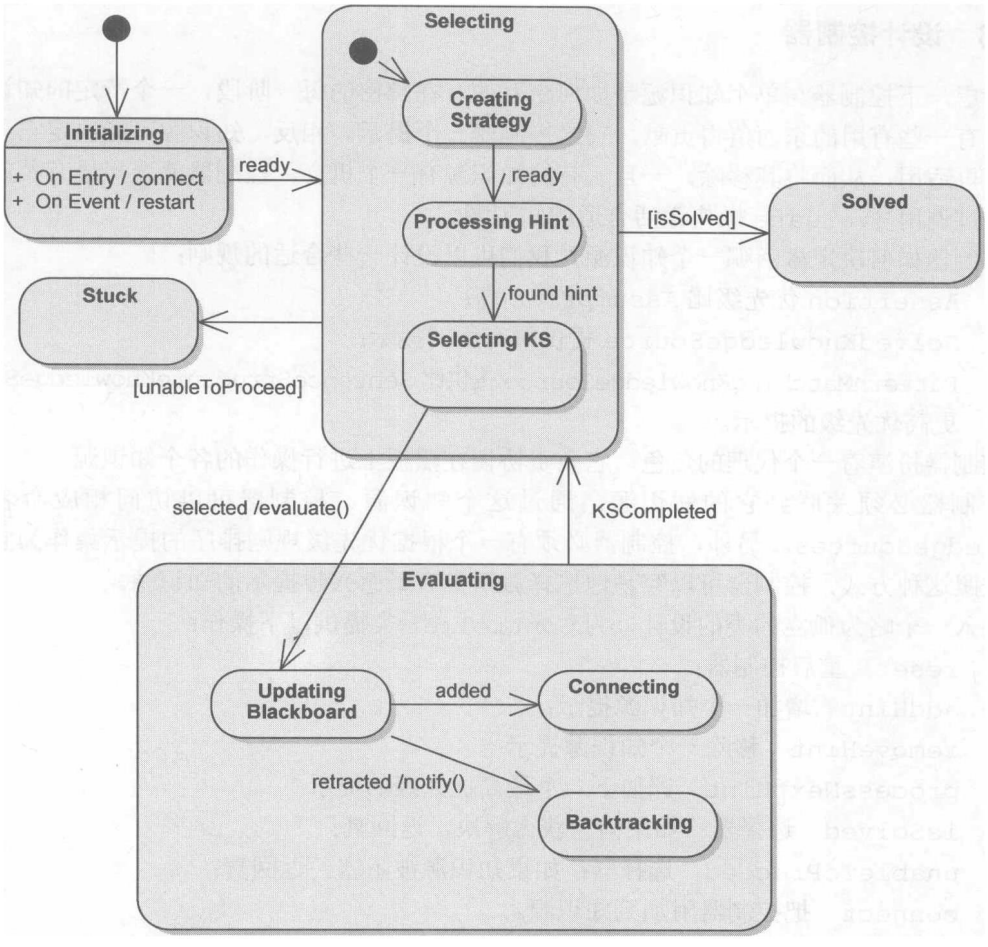


图 10-12 控制器状态机

10.3.4 集成黑板框架

至此，我们已经定义了领域里的关键抽象，可以继续将它们放在一起构成一个完整的应用。我们先实现和测试整个架构的一个垂直切片，然后，逐步地完成这个系统，一次一个机制。

1. 集成顶层对象

图 10-13 是捕获系统中系统顶层对象设计的组合结构图，平行于之前在图 10-1 展示的通用黑板框架结构。在图 10-13 中，我们用一种与展示嵌套类同样简洁的风格，通过 `theBlackboard` 集合展示黑板对象的物理包含，通过 `theKnowledgeSources` 集合展示知识源的物理包含。

在这个图中，我们引入一个叫作 `Cryptographer` 的新类的实例，这个类用作一个包含

黑板、知识源与控制器的聚合。以这种方式，我们的应用可以提供这个类的若干实例，因而，可能有几个黑板同时运行。

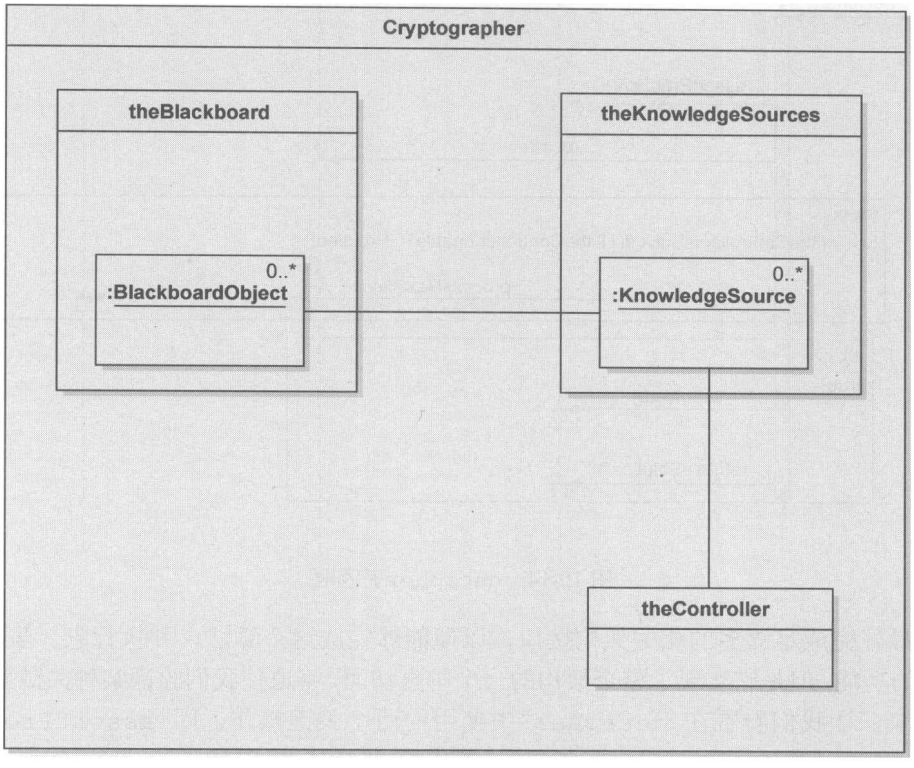


图 10-13 密码分析的组合结构图

为 Cryptographer 类定义如下两个主要的操作：

- reset 重新启动黑板；
- decipher 求解给定的密文。

我们要求这个类的构造器所具有的行为是，在黑板与它的知识源之间以及知识源与控制器之间建立依赖。reset 方法是相似的，因为它仅仅重置这些连接，并将黑板、知识源与控制器返回到一个稳定的初始状态。

虽然这里不展示 decipher 操作的具体细节，但它的签名包括一个字符串，通过该字符串我们提供待解的密文。以这种方式，主程序的根变得相当简单，这在精心设计的面向对象系统中是常见的。

并不意外，操作 decipher 的实现稍微复杂一些。基本上，首先调用 assertProblem 操作在黑板上设置问题。下一步，必须启动知识源，让它们注意到这个新的问题。最后，必须进行循环，在每一个新的过程中告诉控制器处理下一个提示，直到问题获得解决或者所有知识源不能继续处理为止。图 10-14 使用一张序列图展示了控制流。

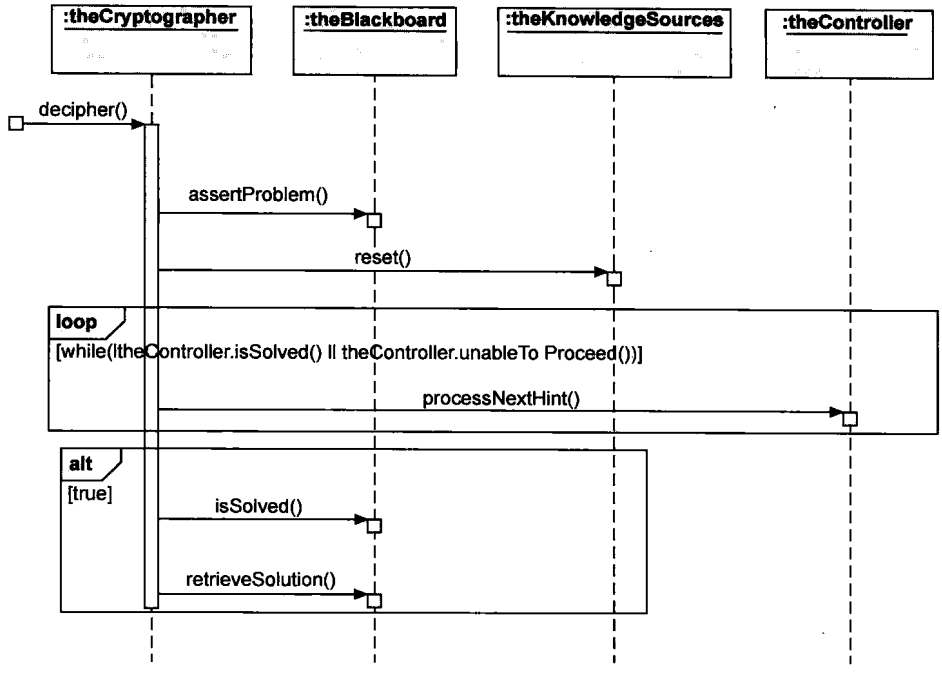


图 10-14 decipher 序列图

建议最好完成足够多的相关架构接口，以便能够完成这个算法，并执行它。虽然此时它只有最小的功能，但是它实现了整个架构的一个垂直切面，将迫使我们验证某种关键架构决策。

接下来，让我们看看在 decipher 中使用的两个关键操作，即 assertProblem 和 retrieveSolution。assertProblem 操作特别令人感兴趣，因为它必须产生整个 Blackboard 对象的集合。以简单伪代码脚本的形式描述算法如下。

```
trim all leading and trailing blanks from the string
return if the resulting string is empty
create a sentence object
add the sentence to the Blackboard
create a word object (this will be the leftmost word in the sentence)
add the word to the blackboard
add the word to the sentence
for each character in the string, from left to right
  if the character is a space
    make the current word the previous word
      create a word object
      add the word to the blackboard
      add the word to the sentence
    else
      create a cipher-letter object
      add the letter to the blackboard
      add the letter to the word
```

设计的目的只是为实现提供蓝图。这个脚本提供了一个足够详细的算法，因此我们不必展示它的全部实现。

操作 retrieveSolution 简单得多，我们仅仅返回黑板上句子的值。在 isSolved 计算为真之前调用 retrieveSolution 将产生部分解。

2. 实现假设机制

此时，我们已经实现允许设置和检索 Blackboard 对象值的机制。下一个主要功能点包括进行关于 Blackboard 对象的假设的机制。这是一个特别有意义的话题，因为假设是动态的（意味着在构造一个解的过程中，它们被例行地创建和销毁），它们的创建或撤销引发控制器事件。

图 10-15 展示了一个知识源陈述一个假设的主要场景。正如这张通信图所示，一旦 KnowledgeSource 创建一个 Assumption，它就通知 Blackboard，接着 Blackboard 为它的 Alphabet 做这个 Assumption，同时为每个 BlackboardObject 应用这个 Assumption。使用依赖机制，受影响的 BlackboardObject 进而可以通知它所依赖的 KnowledgeSource。

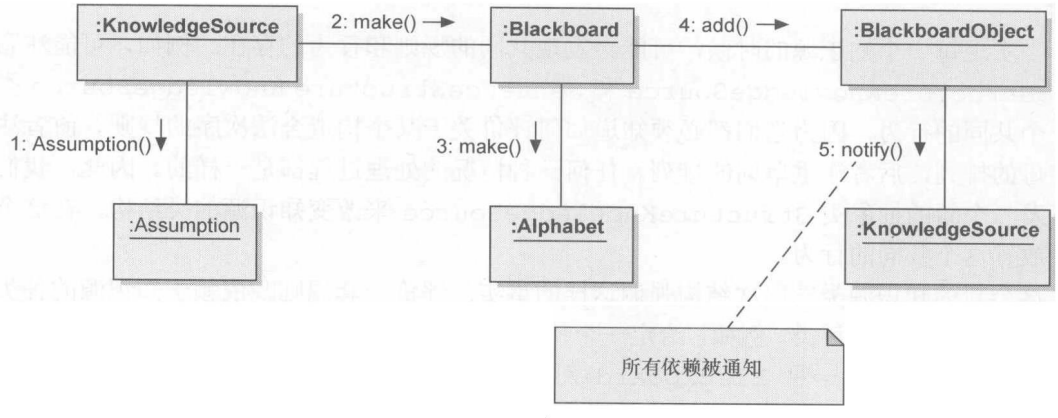


图 10-15 假设机制

在最简单的实现中，撤回一个假设就是简单地撤销这个机制的工作。例如，要撤回关于一个密文字母的假设，我们只要弹出它的假设集合，直至包括正在撤回的假设。以这种方式，给定的假设及所有建于其上的假设都被撤销。

更精密的机制也是可能的。例如，假定做如下的假设：某个单字母单词就是字母 I（假设我们需要一个元音）。稍后，我们可能做另一个假设：某个双字母单词是 NN（假设我们需一个辅音）。如果发现必须撤回第一个假设，可能不必撤消第二个假设。这种方法需要给 Assumption 类添加一个新行为，以便它能够跟踪什么假设依赖其他假设。在这个系统的演化过程中，我们能合理地推迟这种改进，因为增加这个行为没有架构上的影响。

10.3.5 添加新的知识源

既然黑板框架的关键抽象已经就位，并且一旦陈述和撤回假设的机制起作用，那么下一

步就是实现 InferenceEngine 类，因为所有的知识源都依赖它。正如之前提到的，这个类仅有一个真正有意义的操作，即 evaluate。在这里我们不展示它的细节，因为这个特定的方法没有揭示新的重要设计问题。

一旦我们确信推理引擎正常工作，就可以逐步增加每一个知识源。之所以强调使用增量过程，有以下两个原因。

（1）对于给定的知识源，我们并不清楚什么规则是真正重要的，直到将它们应用于实际问题。

（2）调试知识源容易得多，前提是实现和测试更小的相关规则集合，而不是试图一次性测试它们。

根本上来讲，实现每个知识源主要是一个知识工程问题。对于一个给定的知识源，必须与专家（可能是密码专家）协商以决定什么规则是有意义的。当测试每一个知识源的时候，我们的分析可能揭示某些规则是无用的，另一些规则要么太具体，要么太笼统，或许还有一些规则丢失了。然后，我们可以选择改变给定知识源的规则，甚至增加新的知识源。

当实现每一个知识源的时候，可能会发现共同的规则和行为的存在。例如，可能注意到 WordStructureKnowledgeSource 和 SentenceStructure KnowledgeSource 类共享一个共同的行为，因为它们都必须知道如何评价关于某个构造合法次序的规则。前者注重于字母的排列，后者注重单词的排列。任何一种情况的处理过程都是一样的。因此，我们通过开发一个新的抽象类 StructureKnowledgeSource 来改变知识源的类结构，在这个新类中放置这个共同的行为。

这个新的知识源类的层次结构强调这样的事实：评价一套规则既依赖于知识源的种类，也依赖于黑板对象的种类。例如，给定一个特定的知识源，它可能在一种 Blackboard 对象上使用前向链，在另一种 Blackboard 对象上使用后向链。进一步地，给定一个特定的 Blackboard 对象，如何评价它将依赖于应用哪一个知识源。

10.4 后移交

本节中，我们将考虑提升密码分析系统的功能，并观察我们的设计是如何影响这种改变的。

10.4.1 系统增强

在任何智能系统中，知道问题的最终答案都很重要，但是知道系统如何求解问题同样重要。因此，我们希望我们的应用能够自省：它应当跟踪什么时候知识源被激活，做了什么假设以及为什么，如此等等，以便今后我们可以向它提问。例如，为什么做一个假设，如何到达另一个假设，以及什么时候一个特定知识源被激活。

为增加这个新的功能，需要做两件事情。首先，必须设计一种跟踪控制器与每个知识源执行工作的机制；其次，必须修改适当的操作，以便它们记录这个信息。基本上，该设计要求知识源、控制器把它们所做的事情登记在某个中央仓储中。

让我们从开发支持这个机制所需的类开始。首先，可以定义类 Action，它用来记录一个特定知识源和控制器所做的事情。图 10-16 展示了将 Action 类放到架构设计中时的设计。

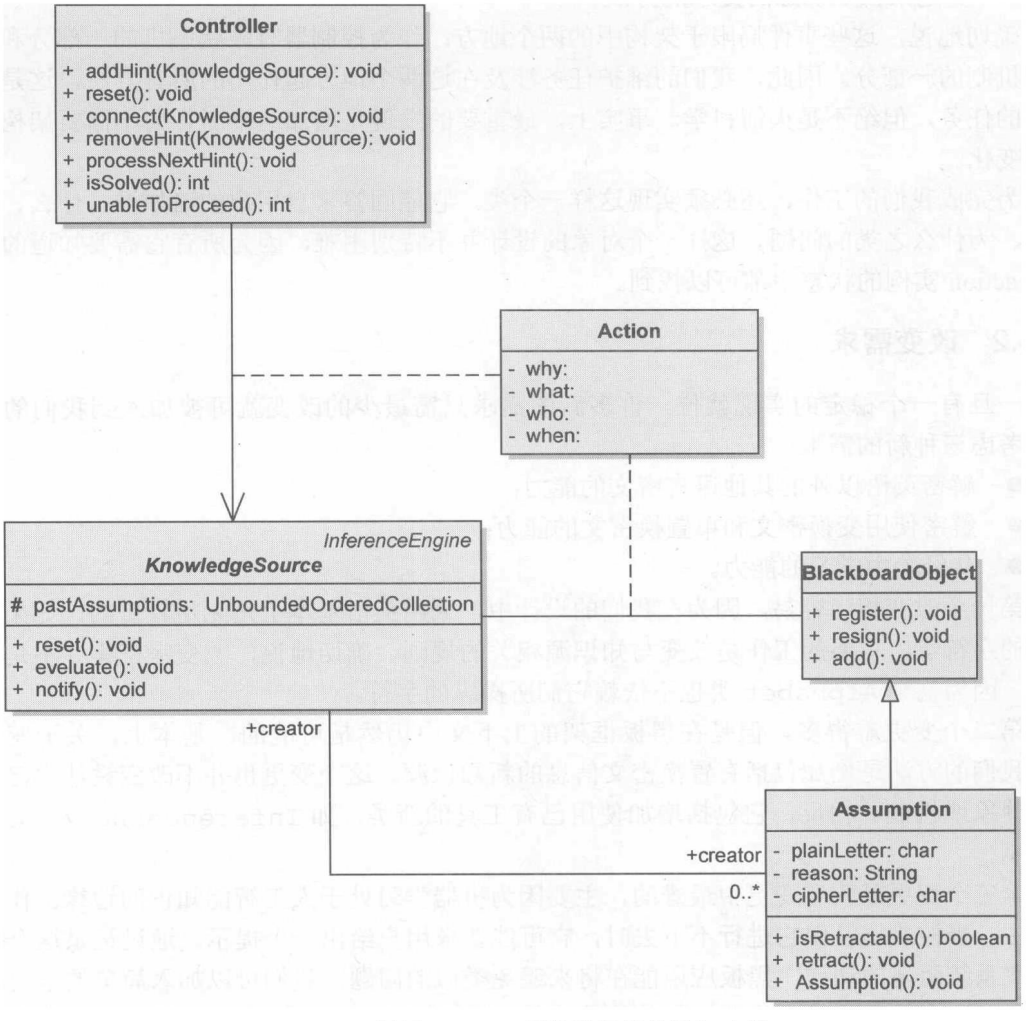


图 10-16 通过 Action 类设计提供的附加功能

例如，如果控制器选择一个特殊的知识源来激活，它将创建这个类的一个实例，为自己设置参数 who，为知识源设置参数 what，并且为某些解释（可能包括提示的当前优先级）设置 why 参数以及发生时间 when。

任务的第一部分完成了，而接下来的第二部分也相当容易。考虑一下在我们的应用中，

哪里发生了重要的事件。实践证明，有5种主要的操作受到影响：

- 陈述一个假设的方法；
- 撤回一个假设的方法；
- 激活一个知识源的方法；
- 引起规则被评价的方法；
- 登记来自知识源的提示的方法。

确切地说，这些事件局限于架构中的两个地方：作为控制器有限状态机的一部分和作为假设机制的一部分。因此，我们的维护任务涉及在这两个地方起作用的所有方法，这是一项枯燥的任务，但绝不是火箭科学。事实上，最重要的发现是增加这个新行为不需要架构发生显著变化。

为完成我们的工作，还必须实现这样一个类，它能回答来自用户的诸如谁、什么、什么时间、为什么之类的问题。这样一个对象的设计并不特别困难，因为所有它需要知道的信息从类action实例的状态中都可以找到。

10.4.2 改变需求

一旦有一个稳定的实现就位，许多新的需求只需最少的改变就可被加入到我们的设计中。考虑三种新的需求：

- 解密英语以外的其他语言密文的能力；
- 解密使用变换密文和单置换密文的能力；
- 从经验中学习的能力。

第一个改变相当容易，因为在我们的设计中，采用英语基本上是无足轻重的。如果使用同样的字符集，主要的工作是改变与知识源相关的规则。确切地说，改变字符集并不是那么困难，因为甚至`Alphabet`类也不依赖它们所操纵的字符。

第二个变更难得多，但是在黑板框架的上下文中仍然是可能的。基本上，关于变换加密，我们的方法是增加包括有置换密文信息的新知识源。这个变更也并不改变设计中已有的关键抽象或机制。相反，它包括增加使用已有工具的新类，如`InferenceEngine`类和假设机制。

第三个变更是所有变更中最难的，主要因为机器学习处于人工智能知识的边缘。作为一种方法，当控制器发现它进行不下去时，它可能要求用户给出一个提示。通过记录这个提示与导致系统阻塞的动作，黑板应用能在将来避免类似的问题。我们可以加入简单的学习机制而不用对现有的类作重大修改。与其他的改变一样，这个改变可以在已有的工具上构建。

数据采集——气象监测站

对于很多科学系统，自动收集数据通常使用传感器或设备实现。这种数据采集通常涉及处理信号和波形来获得所要的信息。数据采集系统的组件包括适当的传感器，用来转换任意被测量的参数为电子信号，电子信号由数据采集硬件获取。所开发的控制软件解释信号，以便分析和显示。

使用面向对象技能来设计一个数据采集系统，允许我们把测量及收集数据的硬件和分析信息的应用隔离。可以定义一个健壮的架构，允许添加或替换传感器和设备，而不会扰乱控制应用的架构。应用接口作为盖在硬件上的皮肤，允许将测量设备从处理信息的应用隔离。在本章的案例中，我们提供了一个数据采集系统的例子——一个气象监测系统。气象监测系统使用传感器和设备测量气象条件，然后进行分析和显示。这个例子阐明了一个实时控制处理应用的面向对象解决方案，该应用提供了一个隔离硬件和应用的可复用组件架构。

11.1 先启

我们的气象监测系统是一个简单的应用，仅仅包含少数的类。乍一看，面向对象的新手们可能会被引导到采用一种固有的非面向对象的方式（例如，通过考虑数据流和不同的输入/输出映射）来解决这个问题。然而，正如我们将要看到的那样，即使是一个像这样小的系统也会很好地借鉴面向对象架构，并显露出一些面向对象开发过程的基本原则。

11.1.1 气象监测站需求

本系统应提供对各种气象条件的自动监测。具体地说，它必须测量：

- 风速和风向；
- 温度；

- 气压；
- 湿度。

系统也应提供以下导出的测量数据：

- 风冷度；
- 露点温度；
- 温度趋势；
- 气压趋势。

系统应有一个决定当前时间和日期的方法，以便它能够报告过去 24 小时内 4 种主要测量数据的最高值和最低值。

系统应当有一个显示屏不断显示所有 8 个主要数据和导出数据，同时显示当前的时间和日期。用户可以通过键盘来指挥系统显示任意一个主要测量数据在 24 小时内的最高值和最低值，以及出现这些值的时间。

系统应该允许用户根据已知值来校正传感器，并允许设置当前的时间和日期。

11.1.2 定义问题的边界

分析时首先要考虑的是软件运行的硬件平台。这是系统分析的固有问题，涉及到制造能力和成本问题，这些问题远远超出本书的讨论范围。为了限定问题，以便暴露软件分析设计问题，我们做以下策略性的假定。

- 处理器（即 CPU）采用 PC 或手持设备式的；
- 时间和日期由一个时钟提供；
- 通过远端的传感器来测量温度、气压和湿度；
- 用一个带有风向标（能感知 16 个方向中任一方向的风）和一些风杯（推动对回转进行计数的计数器）的标柱来测量风向和风速；
- 通过键盘提供用户输入；
- 显示器是一个现货 LCD 图形设备；
- 计算机每 1/60 秒有一次定时器中断。

图 11-1 提供了一个部署图来说明这个硬件平台。

在这个问题上，我们已经选择抛弃一些硬件，这样就可以更好地聚焦在系统软件上。显然，去掉一些硬件（如去掉一些用户输入和图形设备的硬件）就可能需要较多的软件，但在这个特定应用中，改变硬件/软件的界限对我们的面向对象架构来说，在很大程度上是无足轻重的。确实，面向对象系统的特征之一就是倾向于用问题空间的词汇说话，从而描绘出一个与问题的关键实体的抽象相并行的虚拟机。改变系统硬件的细节仅仅影响对系统底层的抽象。

通过围绕每一个这样的接口包装一个类，硬件接口的细节很容易从软件抽象隔离。例如，可以设计出一个简单的类来访问当前的时间日期。首先对这个隔离类进行分析，考虑这个抽

象应当扮演的角色和承担的责任[1]。这样，我们就可以决定，这个类负责跟踪当前的时间和日期，包括时、分、秒、月、日和年。我们的分析可能会决定将这些责任转变为两个服务，分别表示为操作 currentTime 和 currentDate。操作 currentTime 返回以下格式的字符串。

```
13:56:42
```

表示当前的时、分和秒。操作 currentDate 返回以下格式的字符串：

```
6-10-93
```

表示当前的月、日和年。

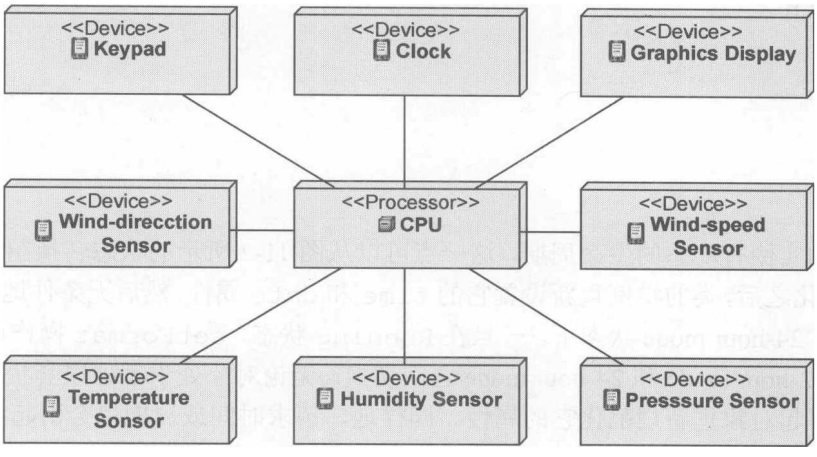


图 11-1 气象监测系统部署图

进一步分析可以得出一个更加完善的抽象，允许客户选择 12 小时制或 24 小时制的时间格式，我们可以为这种抽象提供一个另外的更改操作 setFormat。

通过从公开客户的视角来指定这个抽象的行为，我们将接口和实现做了清晰的分离。基本的思想是对每一个类建立外部视图，就好像已经完全控制了它下面的平台，然后把类的实现作为通向它内部视图的桥梁。这样，在系统硬件/软件边界处的类的实现就将抽象的外部视图同它下面的平台衔接在一起，下面的平台是受系统决策约束的，而系统的决策并不掌握在软件工程师的手中。当然，抽象的内外视图之间的鸿沟并非大得需要一个厚重而低效的实现来粘合它们。

因此，时间和日期类的责任必须包括设定日期和时间。完成这个责任需要新的服务集来设置时间和日期，我们通过以下操作提供：setHour、setMinute、setSecond、setDay、setMonth 和 setYear。

下面总结一下时间/日期类的抽象。

类名：TimeDate

[1] 实际上，不应该一开始就去设计一个新类，而应该首先去寻找已经存在的满足要求的类。一个时间和日期类当然是复用的很好候补。但在本章，我们假定找不到这样的类。

责任：跟踪当前的时间日期。

操作：

currentTime

currentDate

setFormat

setHour

setMinute

setSecond

setMonth

setDay

setYear

属性：

time

date

这个类的实例有动态的生命周期，这一点可以从图 11-2 所示的状态转换图中看出来。可以看到，初始化之后，类的实例重新设置它的 time 和 date 属性，然后无条件地进入 Running 状态，运行在 24-hour mode 状态下。一旦在 Running 状态，setFormat 操作可以将对象的运行模式在 12-hour mode 和 24-hour mode 之间切换。无论对象处于哪种嵌套状态，设置时间和日期都会引起对象重新规范化它的属性。同样地，请求时间或日期也会引起对象计算一个新的字符串值。

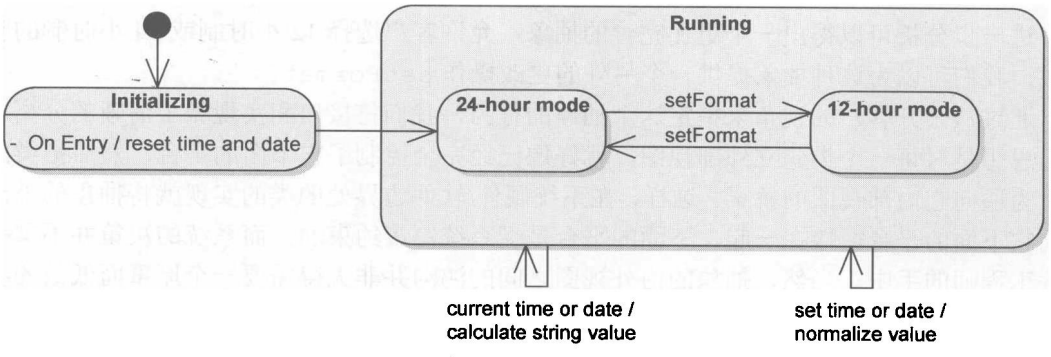


图 11-2 类 TimeDate 的生命周期

我们已经足够详细地说明了这个抽象的行为，所以能够将它以场景的形式提供给分析期间可能发现的其他客户使用。在考虑这些场景之前，说明一下本系统中其他有形对象的行为。

在本系统中，类 Temperature Sensor 相当于一个硬件温度传感器的模拟。孤立的类分析可以得到这个抽象初步的外部视图。

类名：Temperature Sensor
责任：跟踪当前温度。
操作：
currentTemperature
setLowTemperature
setHighTemperature
属性：
temperature

操作 currentTemperature 是自解释的。另外两个操作直接来源于我们的需求，促使我们提供校正每个传感器的机制。现在，假定每个温度传感器值用一个定点数表示，它的低点和高点可以校正到适合已知的实际值，在这两点之间用简单的线性内插法将中间的数字转换为实际的温度，如图 11-3 所示。

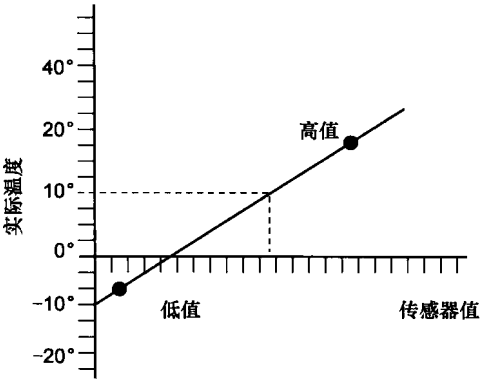


图 11-3 Temperature 传感器校正

细心的读者可能会纳闷，系统中已经有了实际的温度传感器，为什么还要抽象出这个类？系统中确实有温度传感器，但是预料到要复用这个抽象，我们选择捕获它为一个类，这样可以降低它与这一个特定系统的耦合。实际上，特定系统中温度传感器的数目对于我们的架构来说基本上是无足轻重的，通过设计一个类，可以使得这个系统家族的其他成员的程序能够简单地操作任意数目的传感器。

可以在下列规格说明中展示关于气压传感器的抽象。

类名：Pressure Sensor
责任：跟踪当前气压。
操作：
currentPressure
setLowPressure
setHighPressure
属性：
pressure

回顾一下系统需求，可以发现在这个类和前面的类 Temperature Sensor 中，遗漏了一个重要的行为。具体地说，需求迫使我们提供报告温度和气压趋势的手段。目前（因为我们在做分析，而不是设计）我们仅需要聚焦于行为的本性上，最重要的是，聚焦于提取什么抽象来负责该行为。

对 Temperature Sensor 和 Pressure Sensor 这两个类，可以用-1 和 1 之间的浮

点数来表达趋势，这些数字表示某个时间区间上若干数值的一条拟合直线的斜率。[1]因此，我们在这些类中增加以下的责任和其相应的操作。

责任：报告温度或压力趋势，表示为给定时间区间上过去值的拟合直线的斜率。

操作：Trend

因为这个行为是 Temperature Sensor 和 Pressure Sensor 类共有的，所以我们的分析建议创建一个公共的超类 Trend Sensor，来负责提供这个共同行为。

为了全面，应该指出，在我们的分析中也可以选择另外一种观点看这个事情。过去的决策是把这个共同行为作为传感器类本身的一个责任。也可以决定把这种行为作为某个外部代理的一部分工作，由它定期查询特定的传感器，然后计算出趋势，但是因为这样做比较复杂，我们拒绝采用。原来对 Temperature Sensor 和 Pressure Sensor 类所做的规格说明，首先假定每一个抽象有足够的知识来执行这种趋势报告的行为，通过把责任结合起来（以超类的形式），最后可以得到一个简单和概念上内聚的抽象。

可以在下列规格说明中展示关于湿度传感器的抽象。

类名：Humidity Sensor

责任：跟踪当前湿度，表示为饱和度百分比，范围是 0%～100%。

操作：

currentHumidity

setLowHumidity

setHighHumidity

属性：

humidity

Humidity Sensor 没有计算其趋势的责任，因而它不是 Trend Sensor 的子类。

回顾一下系统需求可以发现，一些行为是类 Temperature Sensor、Pressure Sensor 和 Humidity Sensor 共有的。特别是，我们的需求要求提供一种方式来报告过去 24 小时内每一个这些传感器的最高值和最低值。

现在不去考虑如何实现这个责任，那是设计的问题，而不是分析的问题。然而，因为这个行为是三个传感器类共有的，所以我们建议创建一个公共的超类 Historical Sensor，负责提供这个公共的行为。

类名：Historical Sensor

责任：报告过去 24 小时内的最高和最低值。

操作：

highValue

lowValue

timeOfHighValue

---

[1] 数值 0 意味着温度或气压是稳定的，0.1 表示适度上升，−0.3 表示快速下降；接近−1 或 1 的值表示环境的大变动，它超过了系统期望正确处理的场景范围。

timeOfLowValue

Humidity Sensor 是 Historical Sensor 的直接子类。Trend Sensor 是一个中间抽象类，它搭起 Historical Sensor 抽象和具体类 Temperature Sensor 及 Pressure Sensor 之间的桥梁。

可以在以下规格说明中展示关于风速传感器的抽象。

类名：WindSpeed Sensor

责任：跟踪当前风速。

操作：

currentSpeed

setLowSpeed

setHighSpeed

属性：

speed

我们的需求说明：不能够直接探测出当前的风速。必须这样来计算风速：将标柱上风杯的旋转次数除以计数间隔，然后乘以与特定的标柱装置对应的比例值。不用说，这种计算方法是这个类的一个秘密，客户也不关心 currentSpeed 是怎样计算的，只要这个操作能计算出有意义的值，能履行它的契约就可以了。

对上面 4 个具体类（Temperature Sensor、Pressure Sensor、Humidity Sensor 和 WindSpeed Sensor）做快速的领域分析，可以揭示另一个共同的行为：这些类都可以根据两个已知的数据点，用线性内插法来校正自己。我们不是将这个行为复制到 4 个类中，而是创建一个更高一级的超类 Calibrating Sensor 来负责这个行为，它的规格说明如下。

类名：Calibrating Sensor

责任：给定两个已知数据点，提供线性插值的值。

操作：

currentValue

setHighValue

setLowValue

Calibrating Sensor 是 Historical Sensor 的直接超类[1]。

最后一个具体传感器——风向传感器有点不同，它既不需要校正也不需要报告历史趋势。可以在下列规格说明中展示这个实体的抽象。

类名：WindDirection Sensor

责任：跟踪当前风向，表达为罗盘图上的点。

操作：currentDirection

属性：direction

[1] 这个层次关系通过了我们的继承石蕊测试：Temperature Sensor 是类 Trend Sensor 的一种，Trend Sensor 又是 Historical Sensor 的一种，而 Historical Sensor 是 Calibrating Sensor 的一种。

为了统一传感器抽象，我们创建抽象基类 Sensor，它是类 WindDirection Sensor 和类 Calibrating Sensor 的直接超类。图 11-4 说明了这个完整的层次结构。

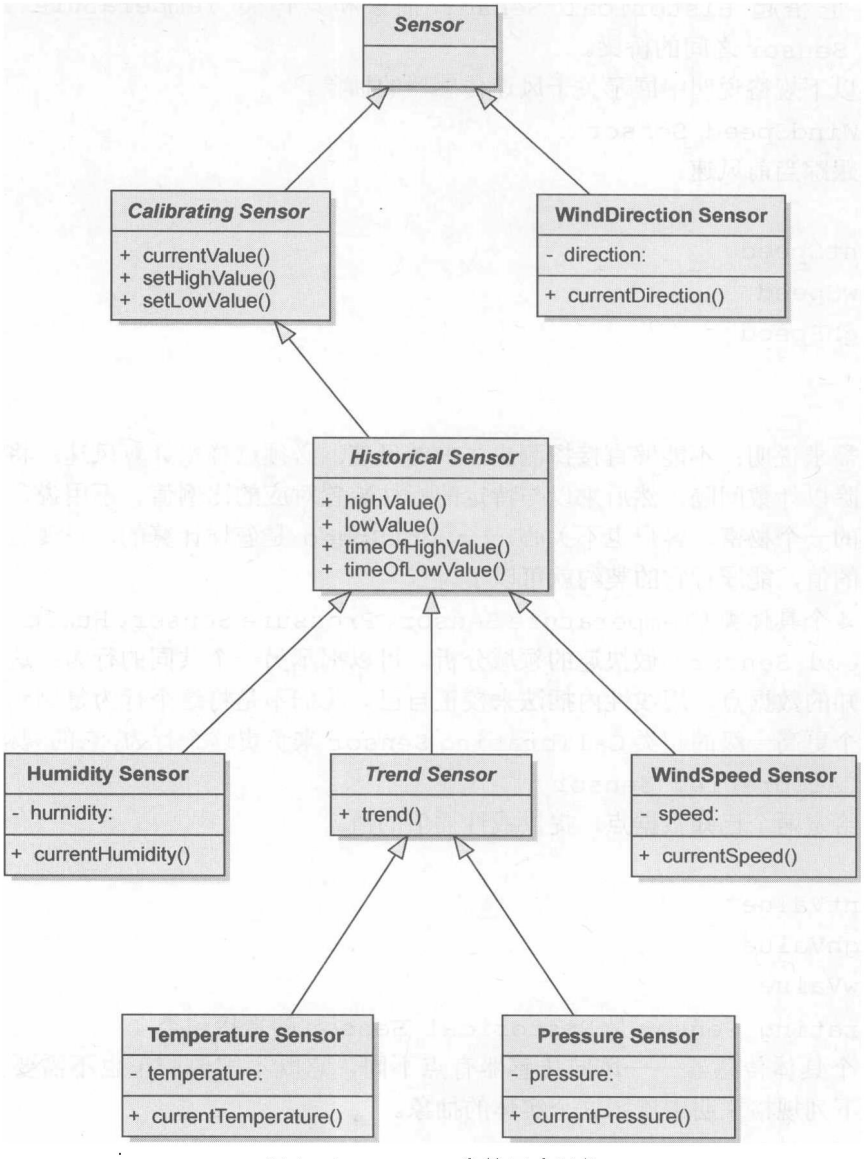


图 11-4 Sensor 类的层次结构

虽然用于用户输入的小键盘抽象不是传感器类层次的一部分，但是它有一个简单的规格说明。

类名：Keypad

责任：跟踪最近一次用户输入。

操作：lastKeyPress

属性：key

值得注意的是，这个类对任何特定键的含义一无所知，它的实例仅仅知道几个键中的一个被按下。我们把解释这些键的含义的责任委托给不同的类，我们将确定什么时候把这些具体的边界类应用到场景中。

对 LCD Device 类的抽象可以将软件同可能使用的特定硬件隔离。为了将软件同可能使用的特定图形硬件解耦，分析促使我们去做气象监测系统的一些通用显示的原型，然后决定需要的界面。

图 11-5 提供了这样一个原型。在这个原型中，省略了对系统需求中的风冷度和露点以及诸如如何显示主要测量数据在过去 24 小时内的最高值或最低值的细节。但是，出现了某些模式：仅仅需要显示文本（以两种不同的大小和两种不同的风格）、圆和线（粗细不同）。此外，我们注意到，显示的一些元素是静态的（如 TEMP 标签），另外一些元素是动态的（如风向）。我们选择通过软件来显示这些静态和动态元素。这样，LCD 自身就不需要特别的标签，从而减轻硬件的负担，但同时也会稍微增加软件的工作量。

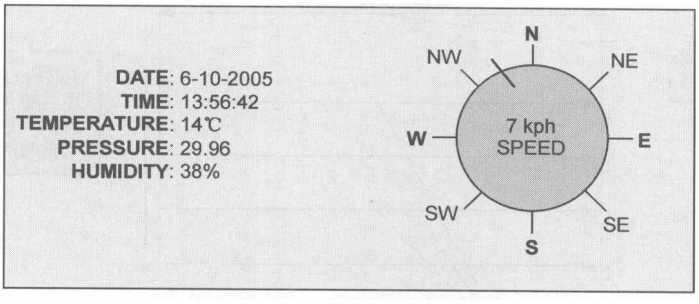


图 11-5 气象监测系统显示面板

可以将这些需求转换成以下类规格说明。

类名：LCD Device

责任：管理 LCD 设备，为显示某些图形元素提供服务。

操作：

drawText

drawLine

drawCircle

setTextSize

setTextStyle

setPenSize

正像类 Keypad 一样，类 LCD Device 并不知道它所操纵的元素的含义。该类的实例仅仅知道怎样显示文字和直线，而不知道这些图形代表什么。这种关注分离留给我们松耦合

的抽象(这正是我们想要的),但是它需要一个代理来负责未加工的传感器数据到显示的转换。我们推迟创建这个新的抽象，直到研究这个系统的一些应用场景之后。

最后一个需要考虑的边界类是关于定时器的。我们将做一个简化的假定，每个系统中有且只有一个定时器，它每隔 1/60 秒向计算机发出中断，调用一个中断服务例程。这个细节特别蹩脚，如果能够对其他的软件抽象隐藏这个实现细节，那是最好的。我们可以设计一个具有回调函数的类，它仅仅输出静态成员（所以要求系统中确实要有一个定时器）。

图 11-6 提供了一个序列图来说明这个抽象的一个用例。从图中可以看出定时器如何和它的客户协作：首先，客户提供一个回调函数，然后每隔 1/60 秒定时器调用这个函数。在这种方式中，客户不必知道怎样去截取定时事件，定时器也不必知道当这样一个定时事件出现时该怎么做。这个协议对客户要求的主要责任比较简单，客户必须在 1/60 秒之内执行完其回调函数，否则定时器将错过一个事件。

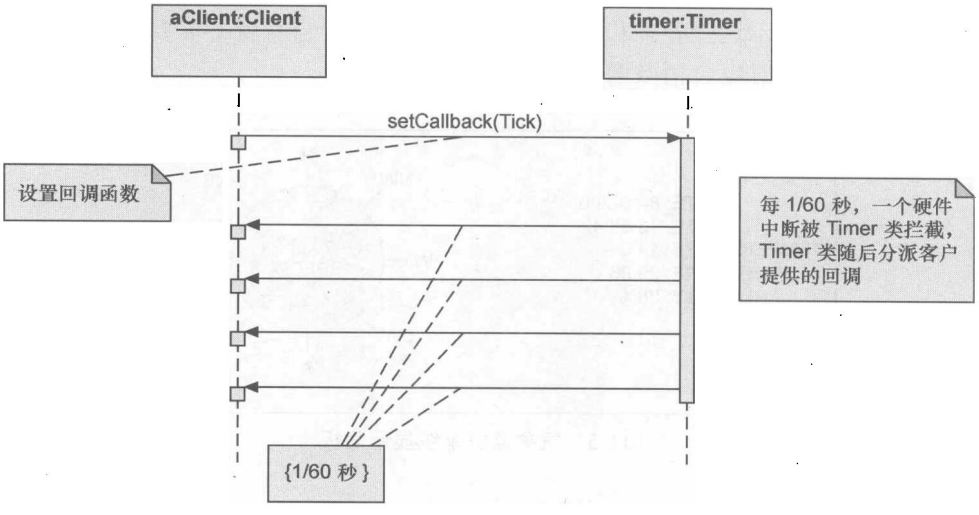


图 11-6 Timer 交互图

由于要截取定时事件，Timer 类应当是一个主动抽象，这就意味着它处于控制线程的根部。下面是这个类抽象的规格说明。

类名：Timer

责任：截取定时事件，相应分派回调函数。

操作：setCallback()

11.1.3 场景

我们已经在系统边界处建立了抽象，现在通过研究几个使用场景来继续分析。首先列出一些主要用例（参见图 11-7），这些用例是从系统客户的观点来看的。

- 监测基本的气象测量数据，包括风速、风向、温度、气压和湿度；

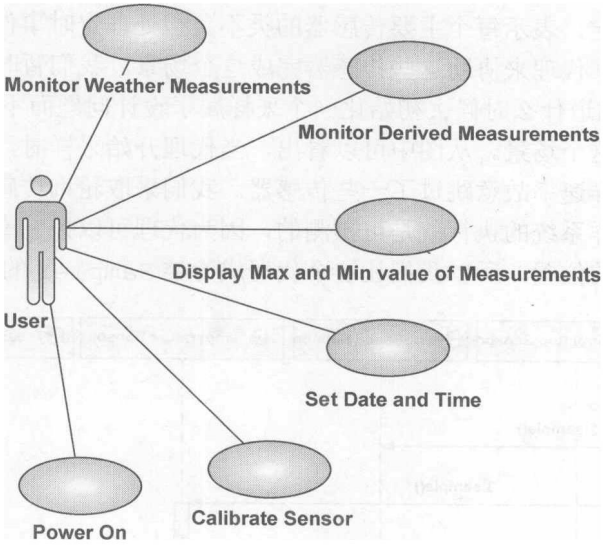


图 11-7 气象监测系统的主要用例

- 监测导出的测量数据，包括风冷度、露点、温度趋势和气压趋势；
- 显示选定测量数据的最高值和最低值；
- 设置时间和日期；
- 校正选定的传感器；
- 启动系统。

再增加两个次要用例：

- 电源故障；
- 传感器故障。

11.2 精化

为了阐明系统的行为（但不是设计），让我们考察以下这些场景。

11.2.1 气象监测系统用例

监测基本的气象测量数据是气象监测系统的首要功能点。其中一个系统约束是：不可能在 1 秒内测量 60 次以上。幸运的是，大多数感兴趣的气象条件改变得极慢。通过分析，我们提出了以下采样速率，这些速率能够充分地捕获气象状况的改变：

- 风向：每 0.1 秒
- 风速：每 0.5 秒
- 温度、气压和湿度：每 5 分钟

早先我们已经决定，表示每个主要传感器的类不负担处理定时事件的责任。因此我们的分析需要设计一个外部代理来协助这些传感器完成这个场景。我们暂时推迟对代理的行为进行规格说明（它如何知道什么时候去初始化一个采样属于设计问题而不是分析问题）。图 11-8 所示的交互图阐述了这个场景。从图中可以看出，当代理开始采样时，它依次查询每一个传感器，但为了降低采样速率故意跳过了一些传感器。我们采取轮询传感器而不是让传感器作为一个控制线程，这样系统的执行就是可预测的，因为代理可以控制事件流。这个名字也反映了它在系统行为中的位置，所以我们让这个代理成为类 Sampler 的一个实例。

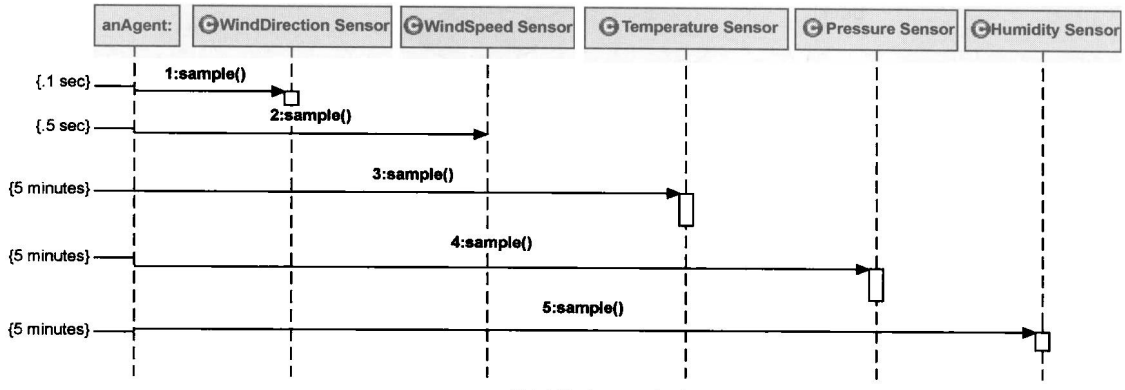


图 11-8 用于监测基本测量数据的场景

必须通过询问交互图的对象中哪一个对象负责将采样值显示在类 LCD Device 的一个实例上，来继续这个场景。最终，我们有两个选择：要么让每一个传感器负责显示它自己（类 MVC 的架构中使用的公共模式），要么创建一个分离的对象负责这个行为。对这个特定的问题，我们选择后者，因为它将所有关于显示布局的设计决策封装到一个类中[1]。

这样，就可把下面的类规格说明加进分析产物中。

类名：Display Manager

责任：管理 LCD 设备上项目的布局。

操作：

drawStaticItems

displayTime

displayDate

displayTemperature

displayHumidity

displayPressure

[1] 这里的主要问题是我们在什么地方显示每一项，而不是每一项看起来怎么样。因为这是一个可能改变的决策，所以最好把在 LCD 设备何处显示每一项的所有知识封装在一个类中。因此，改变我们关于前端面板布局的假设，只需要设计一个类而不是多个类。

```
displayWindChill
displayDewPoint
displayWindSpeed
displayWindDirection
displayHighLow
```

操作 drawStaticItems 用来绘制显示的不变部分，比如用来显示风向的罗盘。我们也让操作displayTemperature和操作displayPressure负责显示它们相应的趋势(因此，当我们转到实现时，必须为这些操作提供一个合适的签名)。

图 11-9 提供了一个类图来说明这些抽象必须协作完成这个场景。同时，图中也显示了每一个抽象在与其他类关联时所扮演的角色。

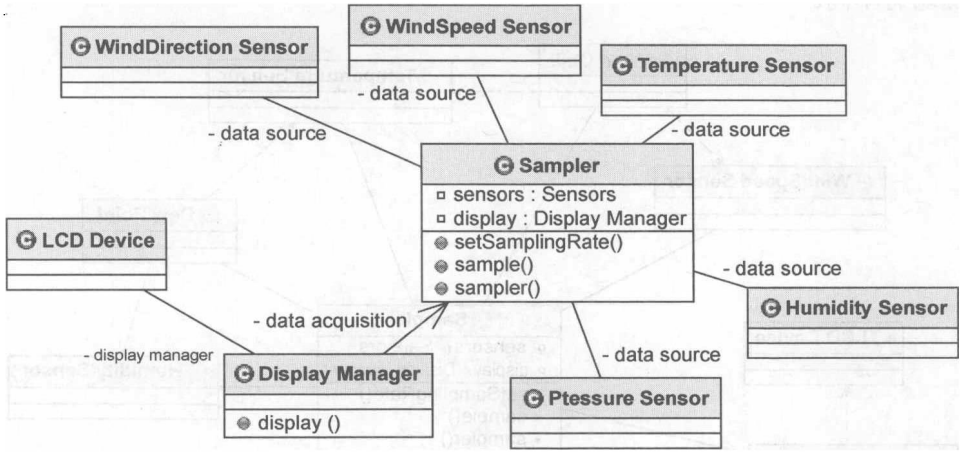


图 11-9 Sampler 和 Display Manager 类

决定在系统中包含一个 Display Manager 类，有一个重要的附带作用[1]。具体来说，就是使软件国际化（也就是使软件适用于不同的国家和语言）容易得多，因为怎样命名元素和显示元素的知识（如 TEMP 和 WIND）是这个类的秘密的一部分。

国际化要求我们考虑隐含的需求：系统应当以摄氏还是华氏显示温度？类似地，系统应当以公里每小时（k/h）还是英里每小时（m/h）显示风速？最终，我们的软件不应受到限制。因为要为最终用户提供灵活性，所以必须在 Temperature Sensor 和 WindSpeed Sensor 类中增加一个操作 setMode。也必须给这些类增加一个新的责任，使它们的实例在一个已知的稳定状态构造自己。最后，必须相应地修改操作 Display Manager::drawStaticItems 的签名，以便在改变测量数据的单位时，显示管理器能够在需要时更新前端面板。

这个发现促使我们为分析中的考虑增加一个场景，即：

[1] 这是一个分析决策还是设计决策？问题可以从两个方向来讨论，虽然在不得不移交的产品软件面前，这样的讨论主要是学术上的。如果一个决策能够提升我们对系统所需行为的理解，另外还引导我们得到一个优雅的架构，我们真地不介意它叫什么。

- 设置温度和风速的测量单位

我们将推迟考虑这个场景，直到研究处理用户交互的其他用例之后。

可以通过已经建立的 Temperature Sensor 和 Pressure Sensor 类的协议，来监测导出的温度和压力趋势的测量数据。但是，为了完成所有的导出测量数据场景，需要创建两个新类——Wind Chill 和 Dew Point 来负责计算它们各自的值。这两个抽象都不代表传感器，不指代系统中的任何有形设备。它们各自作为代理与其他两个类协作完成各自的责任。具体地说，Wind Chill 与 Temperature Sensor 和 WindSpeed Sensor 协作，Dew Point 与 Temperature Sensor 和 Humidity Sensor 协作。依次地，Wind Chill 和 Dew Point 就和 Sampler 协作，所使用的机制与 Sampler 用来监测所有主要气象测量数据的机制相同。图 11-10 说明了这个场景中涉及的类。基本上，这个类图与图 11-9 展示的系统视图略有不同。

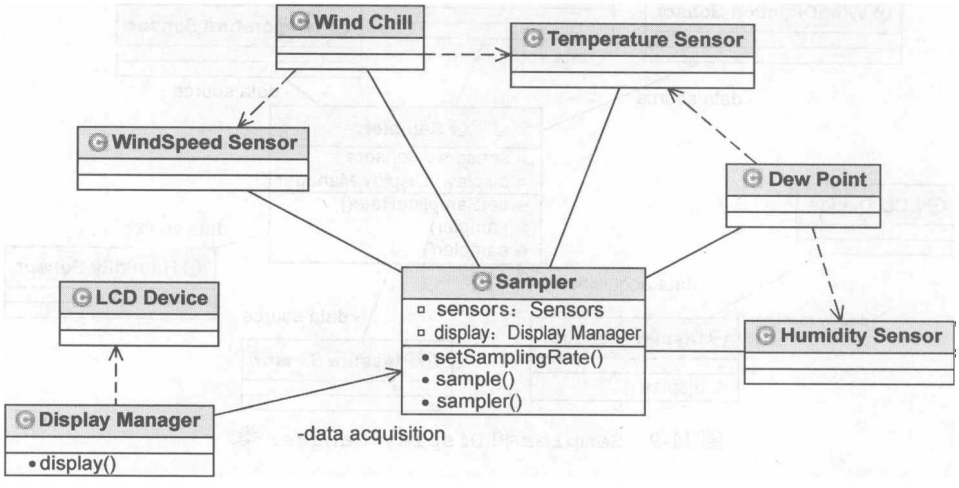


图 11-10 导出测量数据的类

为什么将 Wind Chill 和 Dew Point 定义为类，而不是通过一个简单的非成员函数来完成它们的计算？答案是这种情况通过了我们的面向对象抽象石蕊测试：Wind Chill 和 Dew Point 的实例提供某种行为（即计算它们各自的值），并封装一些状态（每一个必须维持与两个不同具体传感器中的一个特定实例之间的关联），它们各自都有唯一的标识（每一个特定的风速传感器/温度传感器关联必须有它自己的 Wind Chill 对象）。通过“对象化”这些貌似算法的抽象，可以得出一个更加可重用的架构：Wind Chill 和 Dew Point 可以从这个特定应用中被提升，因为对于客户它们呈现出清晰的契约，相对于其他抽象，每一个都提供清晰的分离关注。

下一步要考虑的是涉及用户与气象监测系统交互的不同场景。决定用户与像这样的嵌入式控制器进行交互的恰当动作就像设计一个图形用户界面一样，更是一门艺术。设计这样的用户界面完整方案超出了本书的范围，但是为软件分析人员提供的基本信息是原型化工作，

而且这确实是基本的，它有助于减轻设计用户界面的风险。而且，根据面向对象架构来实现我们的决策，使得改变这些用户界面决策要相对容易一些，不用破坏设计的机理。

考虑一些可能的用户交互用例场景。

用例名：Display Max and Min Value of Measurements

叙述：这个用例显示所选测量数据的最高值和最低值。

基本流：

（1）当用户按下 SELECT 键时，用例开始；

（2）系统显示 SELECTING；

（3）用户按下 WIND、TEMP、PRESSURE 或 HUMIDITY 键中的任何一个，其他按键（除 RUN 外）被忽略；

（4）系统闪烁相应的标签；

（5）用户按下 UP 或 DOWN 键来分别选择显示 24 小时中的最高值或最低值，其他的按键（除 RUN 外）被忽略；

（6）系统显示所选值，同时显示该值出现时的时间；

（7）控制返回步骤（3）或步骤（5）。

注意：用户可以按下 RUN 键来提交或放弃操作，此时，正在闪烁的信息、选择的值和 SELECTING 信息将消失。

这个场景引导我们在 Display Manager 类中增加两个操作——flashLabel（它根据适当的操作变量来识别标签闪烁或停止闪烁）和 displayMode（在 LCD 设备上显示文本消息）。

设置时间和日期的场景与上面的场景相似。

用例名：Set Date and Time

叙述：这个用例设置日期和时间。

基本流：

（1）当用户按下 SELECT 键时，用例开始；

（2）系统显示 SELECTING；

（3）用户按下 TIME 或 DATE 键中的任一个，其他按键（除 RUN 和上面场景的步骤（3）所列出的键外）被忽略；

（4）系统闪烁相应的标签，同时闪烁选择项的第一个字段（即时间的小时字段和日期的月份字段）；

（5）用户按下 LEFT 或 RIGHT 键来选择另外的字段（选择可以来回移动），用户按下 UP 或 DOWN 键来升高或降低被选中的字段的值；

（6）控制返回步骤（3）或步骤（5）。

注意：用户可以按下 RUN 键来提交或放弃操作，此时正在闪烁的信息和 SELECTING 消息消失，时间和日期被重置。

使用以下用户动作的相关模式来校正特定的传感器。

用例名：Calibrate Sensor

叙述：这个用例用于校正传感器。

基本流：

（1）当用户按下 CALIBRATE 键时，用例开始；

（2）系统显示 CALIBRATING；

（3）用户按下 WIND、TEMP、PRESSURE 或 HUMIDITY 键中的任何一个，其他按键（除 RUN 外）被忽略；

（4）系统闪烁相应的标签；

（5）用户按下 UP 或 DOWN 键来选择高校正点或低校正点；

（6）显示器闪烁相应值；

（7）用户按下 UP 或 DOWN 键来调整选中的值；

（8）控制返回步骤（3）或步骤（5）。

注意：用户可以按下 RUN 键来提交或放弃操作，此时正在闪烁的信息和 CALIBRATING 消息消失，校正功能被重置。

校正时，类 Sampler 的实例必须被告知停止采样所选项，否则错误的信息将被显示给用户。因此，这个场景需要我们在类 Sampler 中引入两个新的操作 inhibitSample 和 resumeSample，它们都有一个指定特定测量的签名。

最后一个涉及用户界面的主要场景和设置测量单位如下。

用例名：Set Unit of Measurement

叙述：这个用例用于设置温度和风速的测量单位。

基本流：

（1）当用户按下 MODE 键时，用例开始；

（2）系统显示 MODE；

（3）用户按下 WIND、TEMP 键中的任何一个，其他按键（除 RUN 外）被忽略；

（4）系统闪烁相应的标签；

（5）用户按下 UP 或 DOWN 键来切换当前的测量单位；

（6）系统更新选中项的测量单位；

（7）控制返回步骤（3）或步骤（5）。

注意：用户可以按下 RUN 键来提交或放弃操作，此时正在闪烁的信息和 MODE 消息消失，测量项的当前单位被设置。

通过研究这些场景，可以确定面板上按钮的排列（这是一个系统决策），如图 11-11 所示。

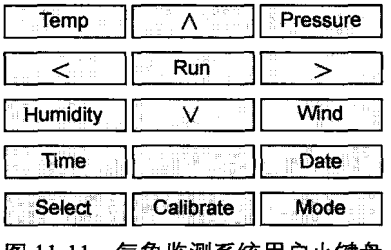


图 11-11 气象监测系统用户小键盘

这些用户界面的每一个都涉及某些形式的模态或有序事件的行为，很适合用状态转换图来表达。因为这些场景耦合得如此紧密，所以我们设计一个新类 InputManager 来负责完成下面契约性的规格。

类名：InputManager

责任：管理和分派用户输入。

操作：processKeyPress

唯一的操作 processKeyPress 激活存活于这个类实例之后的状态机。

如图 11-12 所示，这个类的最外层状态机图包括 4 个状态：Running、Calibrating、Selecting 和 Mode。这些状态直接对应于前述的场景，根据 Running 状态时截获的第一个按键来转换各自的状态，当最后的按键是 Run 时返回 Running 状态。每次进入 Running 时，显示板上的消息被清除。

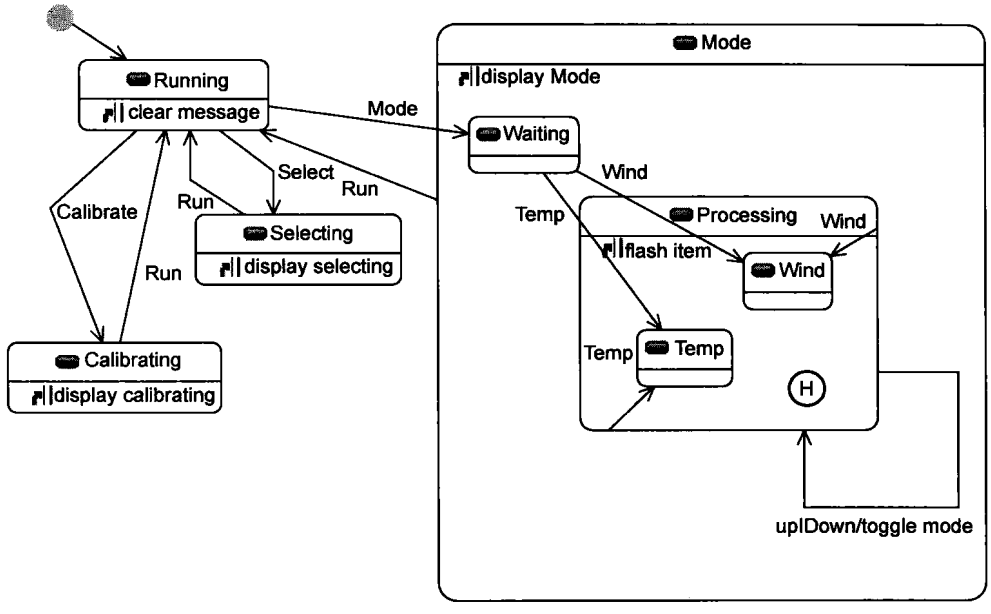


图 11-12 InputManager 状态机图

图 11-12 通过展开 Mode 状态来显示我们如何采取比较形式化的方式来表达场景的动态语义。当进入这个状态时，进入动作（entry action）是在显示板上显示适当的消息。首先是在 Waiting 状态，此时如果截获用户的 TEMP 或 WIND 按键事件，则转出 Waiting 状态进入一个嵌套的状态 Processing，如果截获的是 Run 键，则转回最外层的 Running 状态。每次进入 Processing 状态时，闪烁适当的项；随后进入这个状态时，进入以前进入过的嵌套状态 Temp 或 Wind。

在 Temp 或 Wind 状态，我们可以截获 5 个按键中的任何一个：Up 或 Down（切换相应的模式）、TEMP 或 WIND（重新进入适当的嵌套状态）或者 RUN（脱离外部的 Mode 状态）。

相似地，Selecting 和 Calibrating 状态也可以被展开，以揭示更多的嵌套状态。这里不给出展开的状态机图，因为对于目前的问题，它没有揭示出特别有用的信息[1]。

[1] 当然，对于一个生产产品，一个完整的分析将完成这个状态转换图的展示。在此，我们推迟完成这个任务，因为它是乏味的。事实上，它所揭示的关于正在构造系统的信息都是我们已经知道的。

最后一个主要场景涉及系统启动，它要求我们以一个有序的方式来激活所有的对象，而且要确保每个对象从一个稳定的初始状态开始。下面是针对这个场景作出分析后编写的脚本。

用例名：Power On

叙述：启动系统。

基本流：

（1）当电源接通时，用例开始；

（2）构造每一个传感器。有历史数据的传感器清除其历史数据，趋势传感器准备好它们的斜率计算算法；

（3）初始化用户输入缓冲区，无用按键（由噪音引起）被抛弃；

（4）绘制静态的显示元素；

（5）初始化采样过程。

后置条件：每一个主要测量数据的过去高/低值被设置成首次采样的值和时间。

温度和气压趋势斜率为0。

`InputManager` 处于 Running 状态。

注意，上面脚本中使用后置条件来说明这个场景完成后期望的系统状态。我们将会看到，系统中没有代理来完成这个场景。这个行为是由许多对象协作产生的，每一个对象都被赋予责任，将自己带到一个稳定的初始状态。

这样就完成了对气象监测系统主要场景的研究。为了彻底地完成分析，需要进入不同的次要场景。然而，此时已经发现了足够多的系统功能点，我们想继续进行架构设计，这样就可以开始验证我们的战略决策。

每一个软件系统都需要有一种简单而强有力的组织哲学（把它想成描述系统架构精彩片段的软件等价物），气象监测系统也不例外。开发过程的下一步是清楚地表达这个架构框架，这样就可以有一个稳定的基础来演化系统的功能点。

11.2.2 架构框架

在数据采集和过程控制领域，有许多可以遵循的架构模式，其中两个最普遍的模式是自动执行者同步和基于时间帧的处理。

在第一种模式中，我们的架构包括许多相对独立的对象，它们中的每一个都充当一个控制线程。例如，可以创建若干个新的基于原始硬件/软件抽象的传感器对象，每个对象负责自己的采样，然后向处理这些采样的某个中央代理报告。这个架构有它的优点；如果有一个分布式系统，必须从许多远端收集样本，那么这是唯一有意义的框架。这种架构也允许进行更多的局部采样过程优化（每一个采样执行者都有知识来使自己适应条件的改变，如果条件许可，也许通过提高或降低采样速率来达到这个目的）。

然而，这个架构模式总体上不怎么适合于硬实时系统。在硬实时系统中，必须可以完全预测事件发生的时间。虽然气象监测系统不是硬实时系统，但是它确实需要少量可预测和有序的行为。出于这个原因，我们转向另外一种基于时间帧的处理模式。

如图 11-13 所示，这个模型是基于时间的，它将时间分成若干帧（通常是固定的长度），帧又可以更进一步被分成子帧，每个子帧包含一些功能行为。从一个帧到另外一个帧的活动可能不同。例如，可以每隔 10 个帧进行一次风向采样，每隔 30 个帧进行一次风速采样[1]。这种架构模式的主要优点是能够更严格地控制事件的顺序。

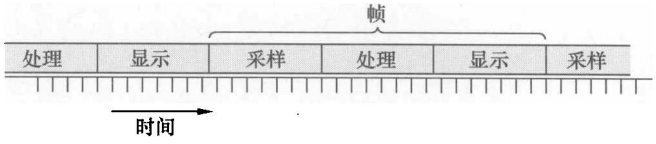


图 11-13 时间帧处理

图 11-14 提供了一张类图，表达了气象监测系统的架构。从图中可以看到我们在早期分析时发现的大多数类，主要的不同点是，现在是展示所有关键抽象之间是怎样协作的。如同产品系统类图中常见的，我们不（也不能）展示每个类和每个关系。例如，忽略关于所有传感器的类层次关系。

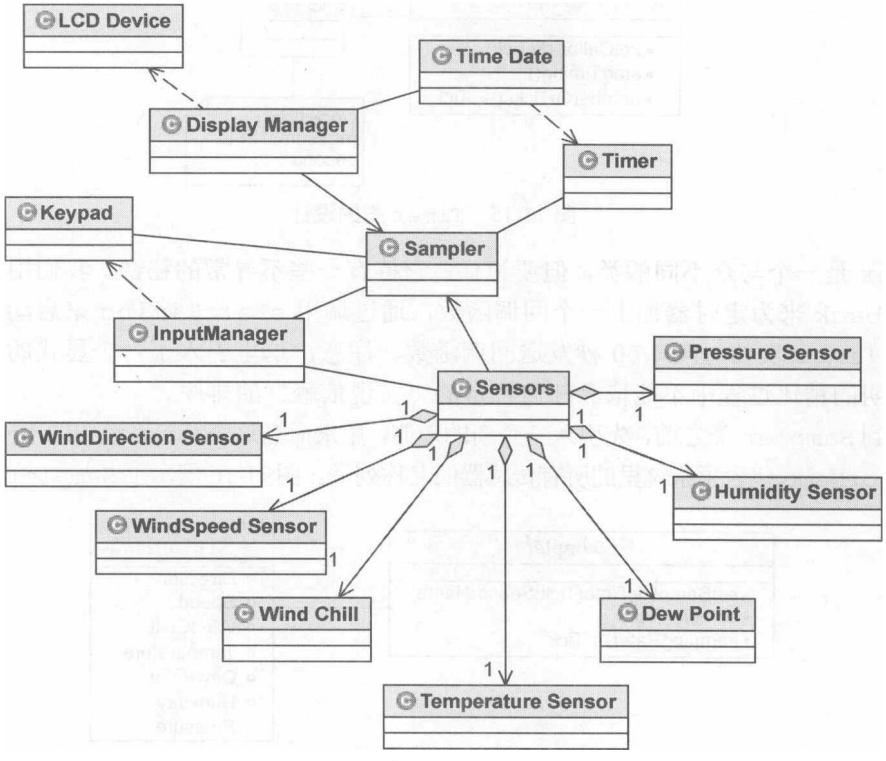


图 11-14 气象监测系统的架构

[1] 例如，如果每一帧被指定为 1/60 秒，则 30 帧就表示 0.5 秒。

在这个架构中，我们创建了一个名叫 Sensors 的新类，它的职责是作为系统中所有物理传感器的集合。由于在系统中至少有两个其他代理（Sampler 和 InputManager）必须与整个传感器集合关联，把传感器塞进一个容器类中可以让我们将系统中的传感器作为一个逻辑整体来对待。

11.3 构造

这个架构的主要行为是由 Sampler 和 Timer 类协作完成的，因此在架构设计期间具体化这些类的原型是明智的做法，这样就可以验证我们的设想。

11.3.1 帧机制

我们从精化类 Timer 的接口开始介绍，该类分派回调函数。图 11-15 展示了类设计。

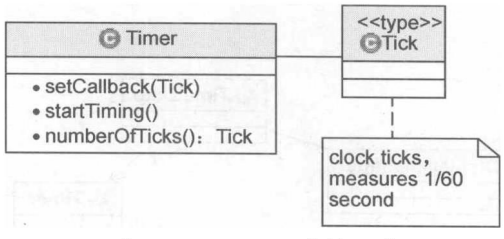


图 11-15 Timer 类的设计

Timer 是一个与众不同的类，但要记住，它持有一些不寻常的秘密。我们用第一个操作 setCallback 来为定时器加上一个回调函数，通过调用 startTiming 来启动定时器的行为，此后 Timer 实体每隔 1/60 秒发送回调函数。注意，这里引入了一个显式的启动操作，因为在声明的精化过程中不能依靠任何特定的“实现依赖”的排序。

在转到 Sampler 类之前，先引入一个新的声明，用来命名这个特定系统中的不同传感器。枚举类 SensorName 包含了系统里的所有传感器的名称列举。图 11-16 展示了 Sampler 类的接口。

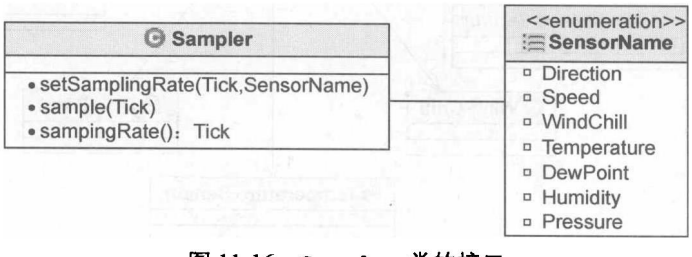


图 11-16 Sampler 类的接口

在引入修改器 setSamplingRate 及其选择器 samplingRate 后，用户就可以动态改变采样对象的行为。

为了将 Timer 和 Sampler 类粘在一起，我们需要少许的 C++粘合代码。首先声明一个 Sampler 类的实例和一个非成员函数，代码如下。

```
Sampler sampler;

void acquire(Tick t)
{
Sampler.sample(t);
}
```

现在我们可以写一个主函数片段，它仅仅将回调函数加到定时器并启动采样过程代码如下。

```
main() {

  Timer::setCallback(acquire);
  Timer::startTiming();

  while(1) {
    ;
  }

  return 0;

}
```

这是一个相当典型的面向对象系统的主程序。它比较短（因为实际的工作被委托给系统中的关键对象），包括一个分派循环（在这里，循环体没有做任何事情，因为没有后台处理需要完成）[1]。

继续讨论系统架构，下一步为 Sensors 类提供一个接口（见图 11-17）。我们假定存在不同的具体传感器类。

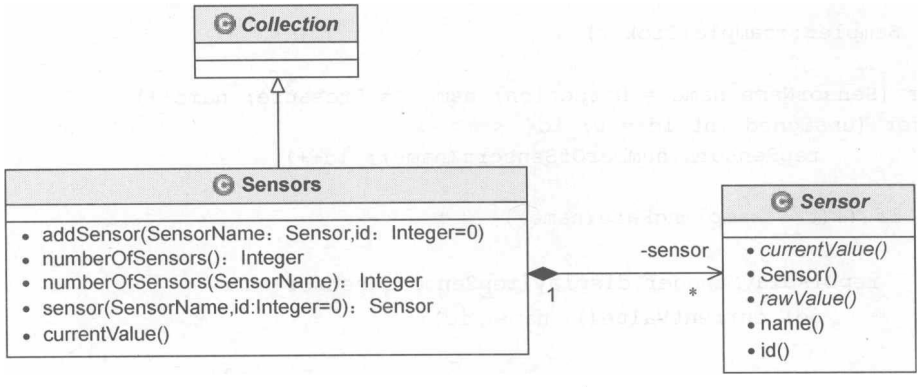


图 11-17 Sensors 类的接口

[1] 还有另一种公用的架构模式：在循环体内截获外部或内部的事件，然后将它们分派给合适的代理。

从根本上说，这是一个集合类型的类，因此，我们使类 Sensors 成为基础类 Collection 的一个子类[1]。我们将 Collection 定义为一个受保护的超类，因为不想将它的大部分操作暴露给 Sensors 类的客户。Sensors 的声明仅提供少量的操作，因为我们的问题受到严格限制，传感器只能被增加而不能从集合中被删除。

我们已经创建了一个泛化的传感器集合类，它能够容纳同一种传感器的多个实例，每一个实例可以用唯一的 ID 来区分——这些 ID 从 0 开始。

在 Sampler 类里指定与 Sensors 和 Display Manager 类的关联，并修改 Sampler 类的一个实例的声明如图 11-18 所示。

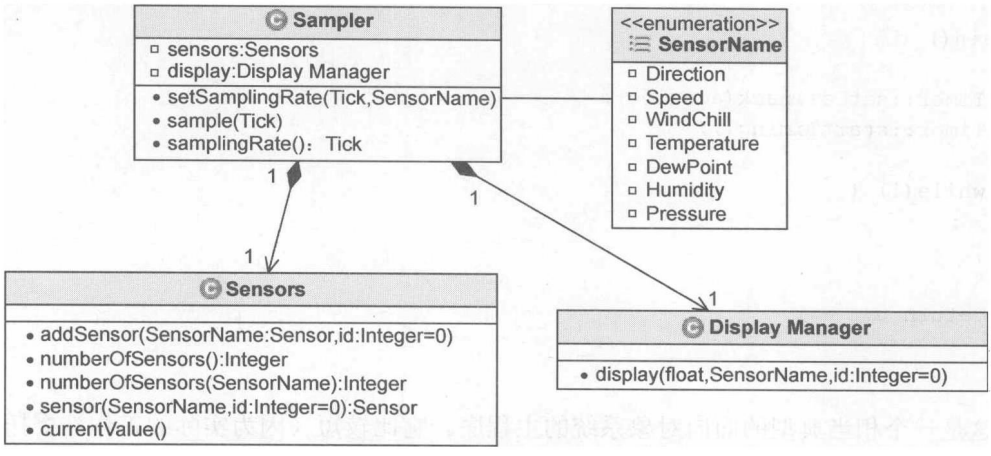


图 11-18 Sampler 类的设计

Sampler 对象的构造将这个代理与系统所使用的具体传感器集合以及特定的显示管理器相连接。

现在，可以实现 Sampler 类的关键操作 sample，代码如下。

```
void Sampler::sample(Tick t)
{
  for (SensorName name = Direction; name <= Pressure; name++)
    for (unsigned int id = 0; id  <
            repSensors.numberOfSensors(name); id++)

      if (!(t % samplingRate(name)))

        repDisplayManager.display(repSensors.sensor(name,
            id).currentValue(), name, id);
}
```

这个成员函数的动作是遍历每一种传感器，然后，遍历该种传感器中的每个唯一的传感

[1] Collection 类是一个抽象的超类，它为语言库所供应的项目集合提供共同操作。

器。对于遇到的每一个传感器，sample 检查是否到了对它的值采样的时候；如果是，则访问集合中的传感器，取走它的当前值，将这个值提交给与 Sampler 实例相关联的显示管理器[1]。

这个操作的语义依赖于操作 currentValue（该操作是为基类 Sensor 而定义的）的多态行为。这个操作也依赖于为类 Display Manager 而定义的 display 操作。

现在我们已经精化了架构的这个元素，图 11-19 所示为突出显示这个框架机制的新类图。

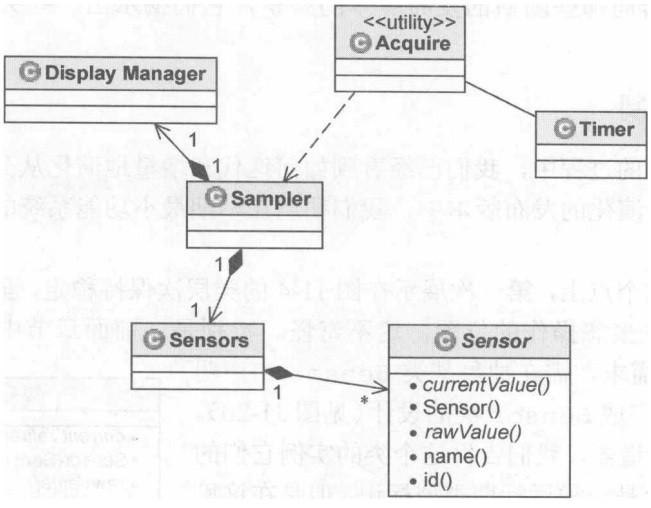


图 11-19 框架机制

我们已经通过分析几个场景验证了我们的架构，现在可以继续介绍增量开发系统的功能点。

11.3.2 发布计划

首先，我们提出一个发布版本序列，它们中每一个都是建立在以前的版本之上：

- 开发一个具有最小功能的发布版本，它仅监测一个传感器；
- 完成传感器的层次结构；
- 完成负责管理显示的各个类；
- 完成负责管理用户界面的各个类。

我们可以按任何次序提供这些发布版本，但是我们选择这种从最高风险到最低风险的方式，这样可以强迫我们的开发过程首先直接攻击困难的问题。

开发最小功能发布版本要求对架构进行纵向的切片，实现几乎每一个关键抽象的小部分。这种做法致力于解决项目中的高风险，即，是否有正确的抽象，以及这些抽象是否有正

[1] 另外一种方法是，让每一个传感器提供一个成员函数来返回它的采样速率，提供另外一个成员函数在 LCD 上绘制传感器。尽管这种设计将一些责任转移给了传感器类，但它使得 Sampler 类的实现变得简单和更易扩展。

确的角色和责任。

这种做法也给我们一些早期的反馈，因为我们可以操纵一个可执行系统。这种强迫早些得到结果的方法有许多技术和社会的效益。在技术方面，它强迫我们开始将系统中的硬件与软件部分衔接在一起，这样能够尽早发现任何的不匹配；在社会方面，它允许我们从真实用户的视角得到对系统的外观和感觉方面的早期反馈。

因为完成这个发布版本很大程度上是一种战术实现方式，所以我们将不再暴露更多它的结构。现在，我们转向那些随后的发布版本的要素，它们揭示出一些关于开发过程的有用知识。

11.3.3 传感器机制

在构造系统架构的过程中，我们已经看到如何迭代和增量地演化从分析时就开始的传感器类的抽象。在这个演化的发布版本中，我们期望在早期最小功能系统的基础上完成这个类层次的细节。

在开发周期的这个点上，第一次展示在图 11-4 的类层次保持稳定，虽然为了提取更大的共性，必须调整某些多态操作的位置，这不奇怪。特别是，前面章节中指出过，出于操作 `currentValue` 的需求，需在抽象基类 `Sensor` 中声明多态操作。我们可以完成 `Sensor` 类的设计（见图 11-20）。

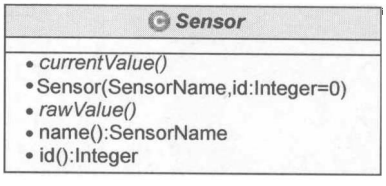


图 11-20 `Sensor` 类的设计

注意，通过类构造器，我们告诉这个类的实例它们的名字和 ID。这实质上是一种运行期类型标识，但是在这里必须提供这个信息，因为按照需求，每个传感器实例必须有一个到特定接口的映射。通过使这个接口成为一个传感器名字和 ID 的函数，可以隐藏这个映射的秘密。

既然已经增加了这个新责任，就可以返回去简化 `DisplayManager::display` 的签名，使其只有一个参数（即对 `Sensor` 对象的引用）。可以去除这个函数的其他参数，因为 `Display Manager` 现在可以询问 `Sensor` 对象来获得它的名字和 ID。

做这个改变是明智的，因为它简化某些交叉类接口。实际上，如果不能跟上像这样的小改变，那么我们的软件架构最终将变得很糟糕，其中协作类之间采用的协议将会变得不一致。

直接子类 `Calibrating Sensor` 的声明建立在 `Sensor` 基类之上（见图 11-21）。

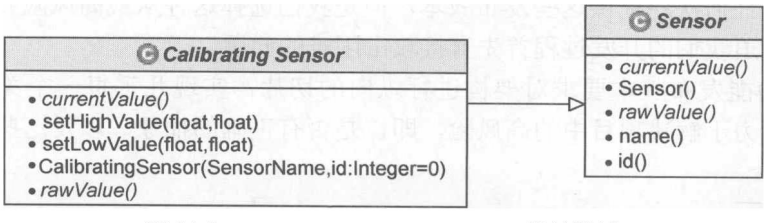


图 11-21 `Calibrating Sensor` 类的设计

Calibrating Sensor 引入两个新的操作——setHighValue 和 setLowValue，并实现前面定义的函数 currentValue。

下面考虑子类 Historical Sensor 的声明，它基于类 Calibrating Sensor（见图 11-22）。

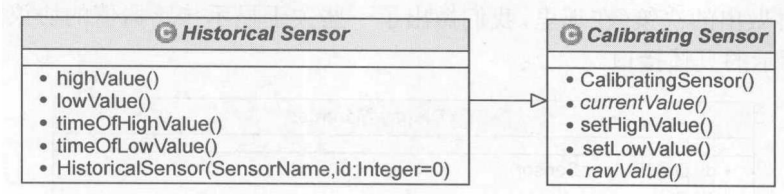


图 11-22 Historical Sensor 类的设计

Historical Sensor 有 4 个操作，在得到高低值时，它们的实现需要与 TimeDate 类协作。注意，Historical Sensor 仍然是一个抽象类，因为我们还没有完成对抽象函数 rawValue 的定义，我们推迟把它定义为一个具体的子类责任。

Trend Sensor 继承自 Historical Sensor，并增加了一个新责任（见图 11-23）。

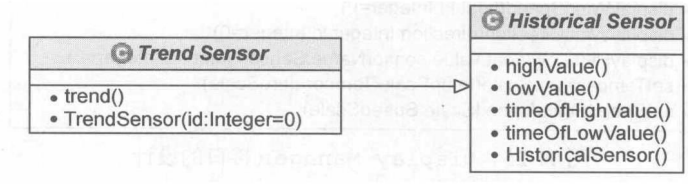


图 11-23 Trend Sensor 类的设计

Trend Sensor 引入了一个新函数。同一些中间类增加的新操作一样，将 trend 声明为具体的，因为我们不希望子类改变它们的行为。

最后，得出具体的子类，如 Temperature Sensor（见图 11-24）。

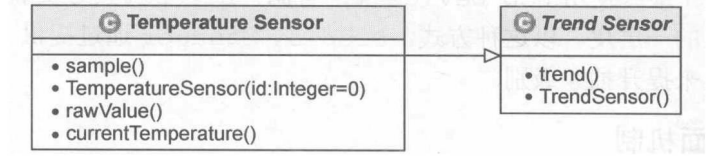


图 11-24 Temperature Sensor 类的设计

注意，这个类的构造器的签名与它的超类稍微不同。简单地说，是因为在这个抽象级别上，我们知道类的确切名称。同时注意，我们根据前面的分析引入操作 currentTemperature，这个操作与多态函数 currentValue 在语义上是相同的，但我们仍然选择都定义它们，因为操作 currentTemperature 更为类型安全。

一旦成功完成这个层次所有类的实现而且将它们集成到之前的发布版本中，我们就可以设计下一级的系统功能。

11.3.4 显示机制

下一个发布版本的实现将完成类 DisplayManager 和 LCD Device 的功能。实际上，这不需要新的设计工作，只需要对某些函数的签名和语义做出一些战术决策。结合在分析第一个架构原型时做出的决策(在那里，我们做出了一些关于显示传感器值的协议的重要决策)，得到图 11-25 所示的具体接口。

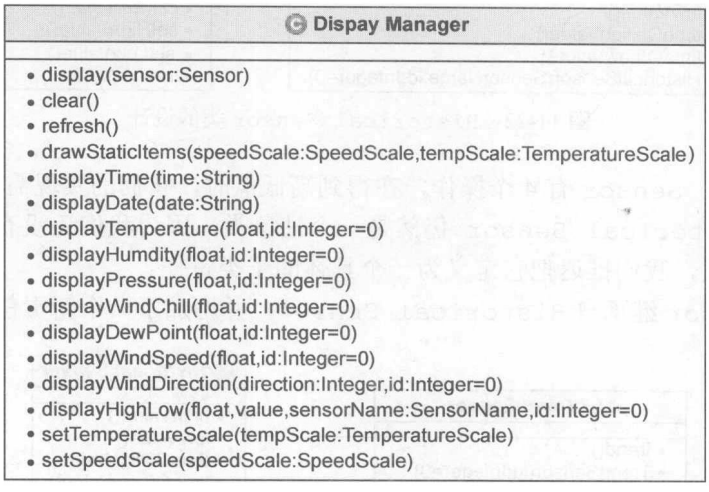


图 11-25 Display Manager 接口的设计

这些操作都不是抽象的，因为我们不希望有任何子类。

注意，这个类有几个原生操作（如 displayTime 和 refresh），也有组合操作 display，该组合操作的出现大大简化了必须与 Display Manager 的实例交互的客户的动作。

DisplayManager 最终使用 LCD Device 类的资源，LCD Device 类就像之前描述过的，作为底层硬件之上的一层皮。以这种方式，Display Manager 通过提供一个更直接说明问题空间本质的协议来提升抽象级别。

11.3.5 用户界面机制

最后一个主要发布版本的焦点是类 Keypad 和 InputManager 的战术设计和实现。与 LCD Device 类相似，Keypad 类也是作为底层硬件之上的一层皮，这样就减轻了 InputManager 直接与硬件对话的讨厌细节。解耦这两个抽象也使得在不破坏架构稳定性的情况下更换物理输入设备更加容易。

我们以一个声明作为开始，这个声明的作用是用问题空间的词汇来命名物理键。一个枚举类 Key 的定义如图 11-26 所示。

我们用 k 作为前缀来避免同 SensorName 中定义的名字发生冲突。接下来，可以捕获

Keypad 类的抽象，如图 11-27 所示。



图 11-26 Key 枚举类的设计

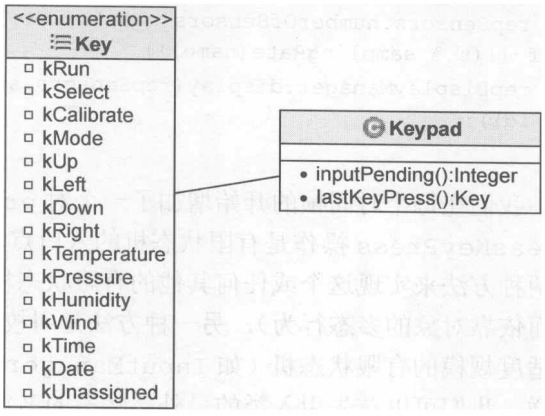


图 11-27 Keypad 类的设计

这个类的协议是从前面的分析得来的。我们增加了操作 inputPending，这样，当存在尚未被处理的用户输入时，客户就可以查询。

类 InputManager 有一个类似的稀疏接口，如图 11-28 所示。

正如我们将看到的，这个类大多数有用的工作是在它的有限状态机的实现中完成的。

正如图 11-14 中所说明的那样，Sampler、InputManager 和 Keypad 类的实例协作响应用户输入。为了集成这三个抽象，必须巧妙地修改 Sampler 类的接口，使其包含一个新的对象 repInputManager（见图 11-29）。

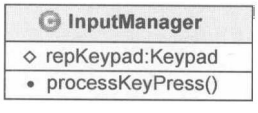


图 11-28 InputManager 接口的设计

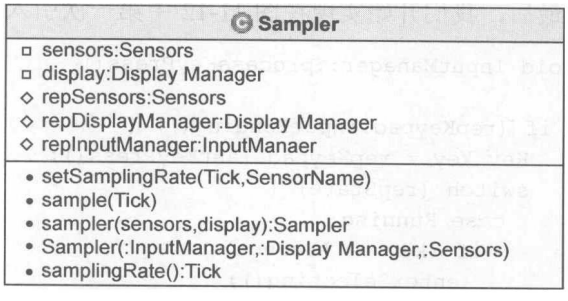


图 11-29 Sampler 接口修改后的设计

通过这个设计决策，在构造 Sampler 的实例时，在 Sensors、Display Manager 和 InputManager 类的实例之间建立一个关联。这样设计确保了 Sampler 的实例总是有一个传感器集合、一个显示管理器和一个输入管理器。

我们还必须增量地修改函数 Sampler::sample 的实现，代码如下。

```
void Sampler::sample(Tick t)
{
  repInputManager.processKeyPress();
```

```
    for (SensorName name = Direction; name <= Pressure; name++)
      for (unsigned int id = 0; id <
          repSensors.numberOfSensors(name); id++)
        if (!(t % samplingRate(name)))
          repDisplayManager.display(repSensors.sensor(name,
          id));
}
```

这里，我们在每个时间帧的开始增加了一个对 processKeyPress 的调用。

processKeyPress 操作是有限状态机的入口点，有限状态机驱动这个类的实例。最终，可以采用两种方法来实现这个或任何其他的有限状态机：一种方法是明确地把状态表示为对象（并因而依靠对象的多态行为），另一种方法是用枚举文字来表示每个不同的状态。

对于适度规模的有限状态机（如 InputManager 类的有限状态机），用后一种方法就足够了。这样，我们可以首先引入类的最外部状态的名字（见图 11-30）。

下一步，引入一些受保护的辅助函数（见图 11-31）。

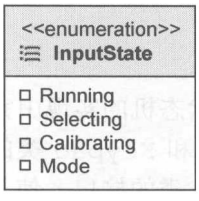


图 11-30 InputState 枚举类的设计

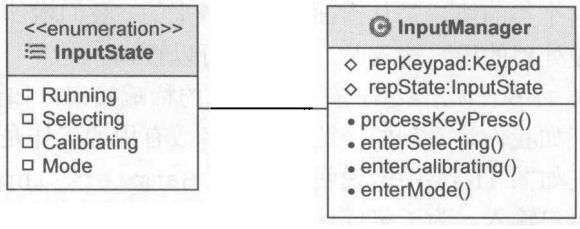


图 11-31 带 InputState 的 InputManager 类设计

最后，我们开始实现在图 11-12 中第一次引入的状态转换，代码如下。

```
void InputManager::processKeyPress()
{
  if (repKeypad.inputPending()) {
    Key key = repKeypad.lastKeyPress();
    switch (repState) {
      case Running:
        if (key == kSelect)
          enterSelecting();
        else if (key == kCalibrate)
          enterCalibrating();
        else if (key == kMode)
          enterMode();
        break;
      case Selecting:
          ...
        break;
      case Calibrating:
          ...
        break;
```

```
        case Mode:
            ...
            break;
        }
    }
}
```

这个函数与其关联辅助函数的实现对应于图 11-12 所示的状态转换图。

11.4 后移交

这个基本的气象监测系统的完全实现具有适度的规模，仅包含大约 20 个类。然而，对于任何真正使用中的软件，改变是不可避免的。来看一下两个增强功能对这个系统架构的影响。

现在我们的系统提供了监测许多有用气象状况的功能，但是可能很快就会发现，用户还想测量降雨量。增加一个雨量测量器会产生什么影响?

幸运的是，不需要根本地改变我们的架构，而仅仅需要扩展它。使用图 11-14 所示的系统架构视图作为基线来实现这个新特性，必须做以下事情：

- 创建一个新类 RainFall Sensor，将它插入到传感器类层次结构的合适位置(RainFall Sensor 是 Historical Sensor 的一种)；
- 更新枚举 SensorName；
- 更新 Display Manager，使其知道怎样显示这个传感器的值；
- 更新 InputManager，使其知道怎样计算新定义的键 RainFall；
- 正确地在系统的 Sensors 集合中增加这个类的实例。

我们必须处理其他一些需要嫁接到这个新抽象中的一些小的战术问题，但最终，不需要破坏系统的架构或关键机制。

再来考虑另一种完全不同的功能。假定我们希望有能力下载一天的气象状况记录到远端的计算机，为了实现这个特性，需要作出下面的改变：

- 创建一个新类 SerialPort，其负责管理一个用于串行通信的端口；
- 创建一个新类 Report Manager，其负责收集下载需要的信息。基本上，这个类必须使用集合类 Sensors 的资源以及与 Sensors 关联的具体传感器的资源；
- 修改 Sampler::sample 的实现，使其周期性地为串行端口服务。

作出这种改变不需要搞乱现存的架构，而是要重用然后提升现有的机制，这是一个设计完善的面向对象系统的标志。

第 12 章

Web 应用——休假跟踪系统

今天，对于很多业务来说，工人的独立性不断增强。工人在多个项目之间分配他们的时间，并向多个项目经理报告，这已经是常事。结果是，经理和他们的工人们之间的非正式交互少了，他们发现管理工人的休假时间越来越困难。因此，在本章探索的例子中，我们虚构的公司决定为经理和雇员开发和部署一个弹性的休假时间管理应用，让他们用来管理休假时间。

对于一个大企业组织，实现这样或类似的功能，决不是孤立地做出决策。所提议的这个系统不可能是这个组织曾经创建的第一个系统，所要的功能必须在所有已有系统的上下文中考虑（也许还要考虑其他被提议的系统）。结果是，很多架构决策已经是事先做好的，在我们的例子里，决定把这个功能作为一个 Web 应用移交，不只是能够与任务合拍，也可以把它考虑成组织现有 intranet 的一个扩展，这样就为它的用户提供了一个方便和自然的进入点。

以下章节将概要地讨论这个系统中对架构有重要意义的和令人感兴趣的方面。由于本书篇幅的限制，我们只表达那些对分享通用的感觉和理解应用来说必须具备的方面和内容。在真实世界中，精确和完整地讨论这样一个系统，需要多得多的内容、模型和例子。这里所覆盖的内容是，如何把面向对象的关键或中心原则应用到这种类型的 Web 应用开发，并讨论若干 Web 应用开发过程特有的有趣问题。

12.1 先启

系统的需求展示为愿景文档概要、关键特性、用例模型、关键用例规格和对架构有显著意义的需求条目。

12.1.1 需求

很容易概括这个项目的愿景：

休假跟踪系统（VTS）将为雇员提供管理自己休假、病假和事假时间的能力，而不必成为一个公司政策专家或本部门休假政策专家。

这个系统的最重要目标是为雇员个人提供能力和责任来管理他们和公司的雇用协议的这个特定方面。这个欲望的潜在动机包括需要梳理人力资源（HR）部的功能，最小化与核心业务无关的活动的管理，给雇员一种授权的感觉。只有系统开发得易用、直观和智能，这些目标才会被满足。因此，可以简单表述一个占支配地位的设计目标：

系统必须容易使用。

读到这样一个含糊的需求，即使是团队中商业软件开发经验最少的开发人员，也会眼珠直转。这些东西太直观和主观，决不应该被记录成官方需求，对吧？并不一定是这样。清楚的是，这简单的一行字虽不是硬性和可跟踪的需求，但它确实表达了所要系统的一个诚实的特性。它是如此重要，以至于除非至少看起来开发人员已经满足它，并最终被最终用户接受，否则已移交的应用就会失败。

像这样含糊的高级别特性，有时需要成为官方需求集的一部分，它们不可以客观地通过测试来验证和跟踪，而是用来支持和纠正其他更具体的需求或设计决策。作为一个例子，考虑对于一个 Web 应用的需求，在一些点上需要最终用户提交给系统一些格式化的文本（如粗体、斜体、列表、段落等）。架构师有两个潜在的解决方案。他们可以要求最终用户把特别的代码放进文本中，就像很多流行的公告牌系统今天所做的，也可以使用定制的或商业上可得的 Java applet 来提供 WYSIWYG 的格式化。这样一个 applet 的技术复杂性和风险是显著的，然而最终用户体验的优势也同样清晰。

如果首要的系统特性或设计目标之一是容易使用，使用这个 applet 是合理的。但是，如果容易使用只是简单的“想要”，而不是关键的需求，引进潜在复杂性和风险的 applet 就不合理。

这个应用的主要目标是改善这个组织的内部业务流程，至少是减少用来管理休假时间请求的时间。过去，所有休假时间不得不由一位直接经理批准，然后在被认可之前，由一位人力资源部职员检查。有时这个手工过程可能要好几天。一个自动系统会加快这个过程，最多需要一名直接经理手工批准（一些高级别雇员可以不需要经理批准）。

这个系统有节省时间和金钱的潜力，大多数发生在人力资源部，本质上是把个体的时间请求过程拿出来，将其替换为一个基于规则的验证系统。HR 人员依然负责输入和更新系统中的雇员休假数据，但是，他们不再是每个时间请求的请求和验证链上的一个链接。

系统将提供以下关键特性：

- 实现一个弹性的基于规则的系统，以验证和确认休假时间请求；
- 经理批准（可选）；
- 提供对之前日历年的请求的访问，允许请求可以在将来一年半内补偿；
- 使用 E-mail 通知来请求经理批准并通知雇员请求状态变更；
- 使用已有的硬件和中间件；
- 作为已有 intranet 门户系统的一个扩展来实现，并使用门户的单点登录机制做所有

授权；

- 保持所有事务的活动日志；
- 使 HR 和系统管理人员能够覆盖所有规则限制的动作，并对覆盖记录日志；
- 允许经理直接奖励事假时间（系统设置限额）；
- 为其他内部系统提供一个 Web 服务接口，以查询任何给定雇员的休假请求总览；
- 和人力资源部遗留系统接口，以检索所需雇员信息和变更。

12.1.2 用例模型

顶级用例模型包含 4 个执行者和 8 个用例，如图 12-1 所示。用例的粒度是相当粗的。例如，用例 `Manage Time` 描述由 `Employee` 调用的功能，包括查看、创建和取消休假时间请求。用例不是描述单个功能需求，而是以场景的形式描述给调用它的执行者提供显著价值的某些东西。例如，只是能够查看休假时间请求的价值很小，但能够管理自己的休假时间，则确实提供了显著的价值。

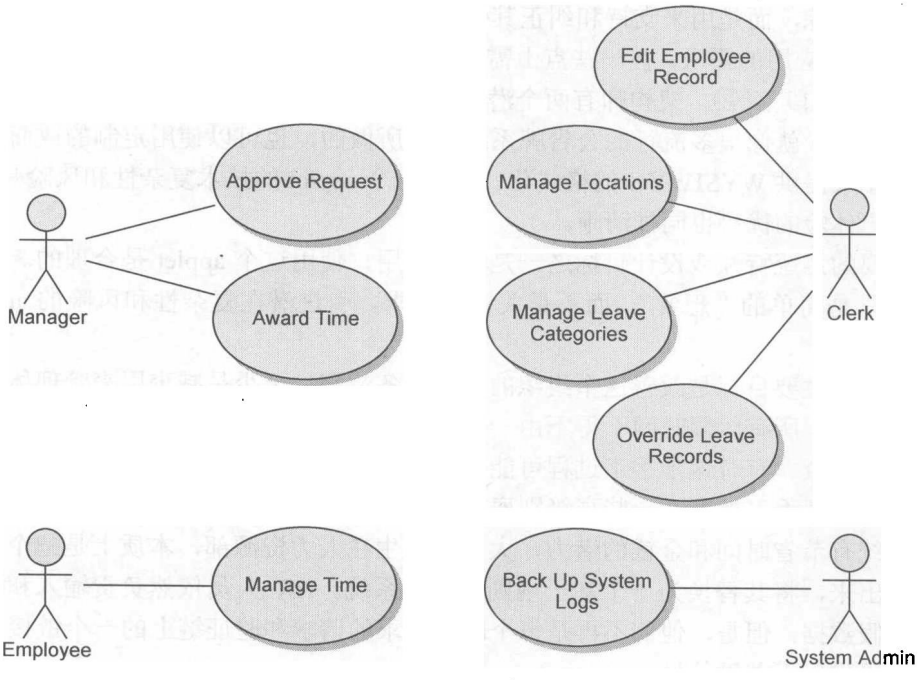


图 12-1 顶级用例模型

关于 Web 为中心的系统的用例，一个令人感兴趣的观察是它们倾向于非常严格地以刺激和响应的形式表达，即，一个用例场景通常表达成一个执行者动作和即时系统响应的列表。系统响应与真实和个性化命名的 Web 页面或屏幕之间通常也会有很密切的关系。这和非 Web 应用形成对比，对于非 Web 应用，活动场景的内容更多聚焦于交换的信息和使用姿态，而不是离散的用户界面单元，如 Web 为中心系统中的 Web 页面。

系统包含以下执行者。

- Employee 这个系统的主要用户。雇员使用这个系统来管理他休假时间。
- Manager 有所有管辖雇员的能力和目标的雇员，但有附加的责任，即批准直接下属休假请求。在系统设置的某个限额的允许下，经理可以嘉奖下属补偿时间（comp time）。
- Clerk 人力资源部成员，他有足够的权力查看雇员的个人数据，并负责保证雇员在所有 HR 系统中的信息维持最新和正确。一个 HR 职员可以添加或移除系统中的几乎任何记录。在真实世界中，HR 职员可能是也可能不是雇员；但是，如果他们是雇员，将使用两个分离的登录 ID 来管理这两个不同的角色。
- System Admin 负责系统技术资源（如 Web 服务器、数据库）平滑运行，并收集和归档所有日志文件的角色。

主要用例如下：

- Manage Time 描述雇员如何请求并查看休假时间请求；
- Approve Request 描述经理对下属的休假时间请求如何反应；
- Award Time 描述经理如何嘉奖下属额外的休假时间（补偿时间）；
- Edit Employee Record 描述 HR 职员如何编辑系统中的雇员信息，包括设置所有休假时间允许和经理可以嘉奖的最多时间；
- Manage Locations 描述 HR 职员如何管理位置记录及其规则；
- Manage Leave Categories 描述 HR 职员如何管理休假类目及其规则；
- Override Leave Records 描述 HR 职员如何覆盖由系统中的规则作出的任何对休假时间请求的拒绝；
- Back Up System Logs 描述系统管理员如何备份系统日志。

12.2 精化

有时候，不总是清楚分析何时开始或者先启阶段的收集和理解需求何时结束，这也是迭代开发过程如此流行以及瀑布过程经常被质疑的原因。但是，重要的是，先描述和讨论最重要的、对架构有显著意义的用例。不需要所有细节都完整，但在可以精化特定用例之前，应该涉及对架构有显著意义的用例。

在理想化的世界里，系统的分析应该独立于实现架构。但是，现实是实现架构的先验知识可能会影响分析的形态。

当正在开发的应用是一个 Web 应用时，特别会发生这种情况。因为在大多数情况下，决定采纳一个 Web 为中心的架构在先启阶段就被知道了。在这个案例中，解决方案将最终作为一个 Web 为中心的应用移交，这出现在详细需求中。例如，Manage Time 用例（稍后讨论）主要流程中的陈述，特别提示“雇员应该能够访问一份可视的日历，来帮助选择和比较所选日期”是基于“大多数 Web 浏览器没有通用日历控件”的知识。在一个本地 Windows 客户

应用的规格中，这样一个能力的假设可能不会被显式提出，因为这样的控件在基于 Windows 的本地客户应用中普遍被使用。这个案例的用例作者显式识别了一个系统需要用户友好的区域。更微妙和普遍的指示是，在用例书写过程中知道的架构知识，常常会引用“导航到 Web 页面”和“提交”信息，概念紧紧和 Web 为中心的架构链接在一起。

可惜，没有一致的方法来量化地表达一个武断的架构。这里，我们将使用 4+1 架构视图模型[3]，就像第 6 章中描述的。

以下对 Web 应用架构的简短描述，不意味着对 Web 为中心的架构的完整讨论；但是这里确实覆盖了足够显著的方面，来帮助解释之后作出的设计决策。

12.2.1 部署视图

Web 应用是客户/服务器应用的一种特化，至少有两个主节点——服务器和客户浏览器。服务器是一个网络上已知地址的节点，倾听特定端口（通常是端口 80）的 HTTP 请求。在用户的指示下，客户浏览器应用作出请求，请求服务器上 HTML 格式化的资源。服务器很可能会并行运行很多服务，包括其他 Web 应用。服务器可能是一个数据库服务器、一个应用服务器等。在图 12-2 中，`Client` 和 `Server` 节点被清楚标识。

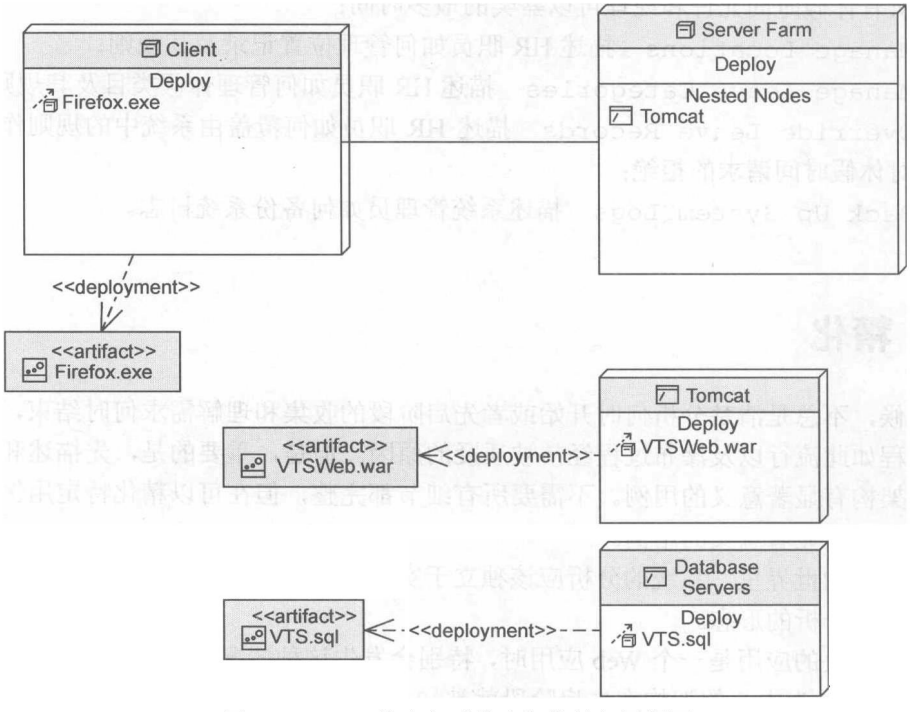


图 12-2 Web 为中心系统中组件的部署视图

在关键组件的部署视图中，执行环境 `Tomcat` 和 `Cloudscape` 被处理成服务器的嵌套节点。`Tomcat` 节点是一个基于 Java 环境的 Web 应用执行环境。`Tomcat` 节点自身在这里被

展示，部署了工件 VTSWeb.war——一个 Web 应用归档文件。Cloudscape 执行环境是一个能够执行 SQL 文件的数据库服务器，这里展示它部署了一个工件叫作 VTS.sql。

图 12-2 中的 Client 节点部署了 Firefox.exe 工件，这是一个可执行的 HTML 浏览器应用。Client 节点和 Server 节点有一个通信链接，这个通信链接可以是从拨号 ISP 到宽带或无线的任何东西。需要关注的重要和逻辑方面是，客户通过这个通信链接与嵌套 Web 和数据库服务器通信。

Web 为中心的系统的表达是过分简化的。在更大的系统中，所有节点、组件、服务器和通信链接的拓扑结构的部署图是复杂的，需要好的可视化和归档系统，如 UML。图 12-3 所示为一张抽象的部署图，展示一个智能路由器如何可以被用于一个 Web 服务器 farm，以及整个应用集合如何可以从应用服务器运行。这样的策略细节可以十分复杂，但通过添加附加处理节点缩放一个应用的通用策略，在 Web 应用当然是可行的。

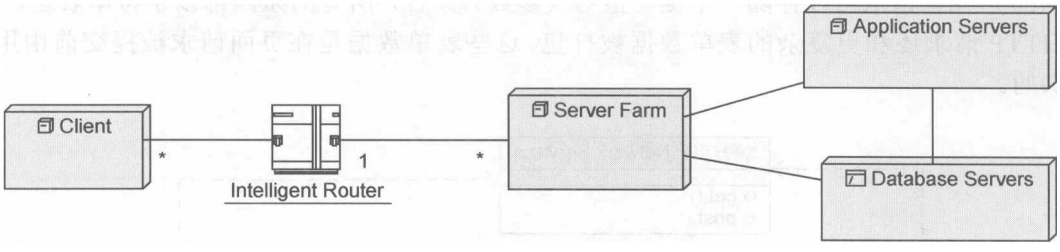


图 12-3 有并行和冗余处理节点的 Web 应用服务器 Farm 部署图

12.2.2 逻辑视图

典型的 Web 应用至少有 4 个逻辑组件：运行在客户上的浏览器、Web 或应用容器、分离的应用逻辑自身的组件和数据库服务器组件。在图 12-4 中，Firefox.exe 组件是常见的多平台浏览器，Tomcat 是流行的基于 Java 服务器页面（JSP）规格的 Web 容器。VTSWeb 组件代表业务应用，Cloudscape 是一个简单的便携式数据库服务器。这张图最重要的逻辑点是，客户浏览器绝对不会直接和数据库接触，甚至不和业务应用组件接触，对服务器端资源的所有访问都由 Web 容器做中介。这使得 Web 容器成为考虑安全性需求时的一个重要组件。

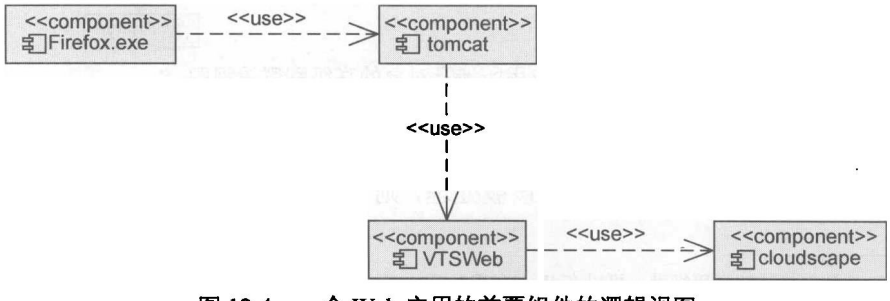


图 12-4 一个 Web 应用的首要组件的逻辑视图

Web 服务器被设计成倾听某个端口——通常是端口 80，并响应 GET 和 POST 请求。GET 和 POST 是在 HTTP 协议中定义的命令。基本上，它们是从 Web 服务器请求信息的两个简单的方式。当用户提供的数据或文件被发送到服务器时，POST 方法更可扩展一些，更常被使用。使用哪一个命令的决定嵌在 HTML 格式化页面中，由浏览器呈现。通常浏览器请求返回的信息是一个 HTML 格式化文档，可以被浏览器可视地呈现；而服务器可以返回任何形式的流数据，让客户负责保存或委托另一个本地安装的应用处理信息。

图 12-5 展示了代表应用的客户和服务器层中的元素的逻辑类。在客户上，一个浏览器负责用简单的 HTTP GET 或 POST 命令请求 Web 页面（或者更通用地说，资源）。这些服务可以由操作系统提供或者在浏览器自身实现（浏览器调用更低级别的网络 API）。Web 容器持续倾听某个端口（如端口 80）是否有进入的 HTTP 请求。一个 HTTP 请求被浏览器打包并发送给 Web 容器。它包含一个资源标识符，指向一个特定的 Web 页面或引用一个 Web 应用。请求可以伴随一个键—值对（参数）集合，所有的东西都以字符串表达。[1]一些 HTTP 请求还和更复杂的表单数据被打包，这些表单数据是在页面请求被提交前由用户提供的。

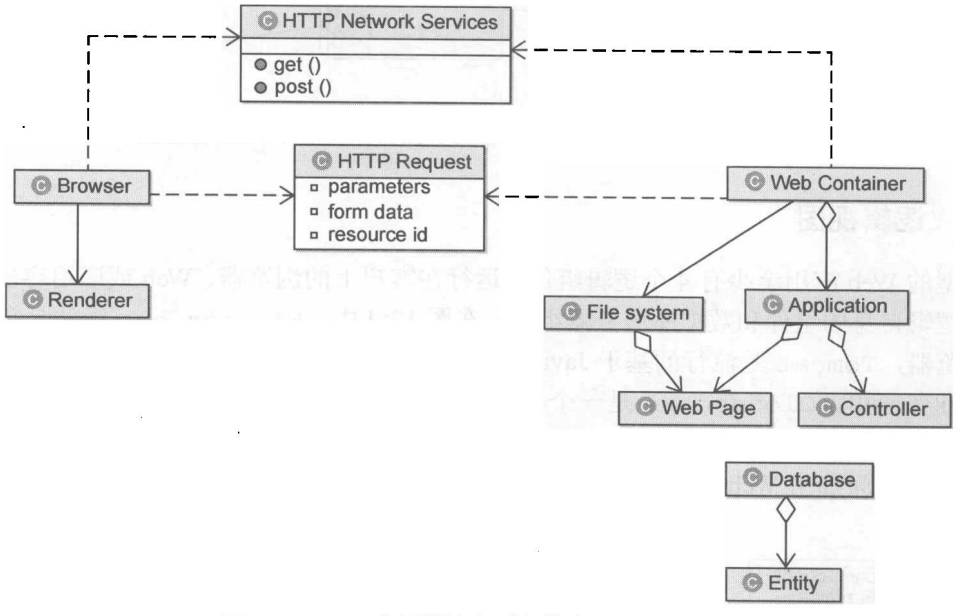


图 12-5 Web 应用所涉及对象的高级别逻辑视图

当 Web 容器接收资源请求时，它必须首先决定要调用的文件或应用资源。容器调用适当的资源，如果资源表明它是动态的且应该被处理，则 Web 容器将执行它。在处理期间，HTTP

[1] 关于参数和 Post 数据格式的琐碎细节，可以在 World Wide Web Consortium 网站找到（www.w3.org）。幸运的是，大多数这些细节由我们选择实现应用的 Web 应用框架管理。

请求检查信息（参数和表单数据），应用执行必需的业务逻辑。通常这种逻辑执行在一个分离的应用服务器（如一个 EJB 容器）上，它可能会访问另一个数据库服务器。

逻辑上，一个 Web 应用由 Web 页面、控制器和实体组成（分析模型中三个基本的构造型）。静态 Web 页面（没有处理）可以只驻留在一个文件系统中，而包含业务逻辑处理的 Web 页面必须在容器上下文中被加载和执行。控制器对象经常被嵌进组件中，持久实体由数据库管理。

12.2.3 进程视图

典型的 Web 应用至少涉及两个进程，但通常会更多。除了在处理 HTTP 请求期间之外，客户和服务器异步操作。附加的数据库、认证和消息服务器经常是典型的 Web 应用的一部分，它们可以共存在单个服务器节点上或者分布到多个节点。Web 应用的过程布局和任何客户/服务器架构一样有同样的弹性，只有一个不同的需求——客户节点必须运行某些形式的 Web 浏览器客户软件，以初始化和服务器端应用的通信。

正如之前提到的，理解 Web 应用架构最重要的是它们工作在无连接模式，即客户和服务器的连接时间决不会长过一个 GET 或 POST 请求过程。一旦浏览器请求一项服务器资源（即 HTML 格式化 Web 页面），服务器回应了该资源，客户和服务器之间的连接就断了。图 12-6 展示了客户向服务器（Tomcat）做简单的 GET 请求。服务器根据请求决定调用什么 Web 应用和资源。应用中的一个 Web 页面被实例化或调用，相当于业务逻辑执行的主触发器。在处理 GET 或 POST 请求的过程期间，Web 应用中几乎所有的业务逻辑都被调用。当逻辑完成时，应用负责准备一个响应页面（即场景中的下一个 Web 页面）。重要的是认识到，一旦 Web 资源的请求被履行，应用将停止为特定客户工作。

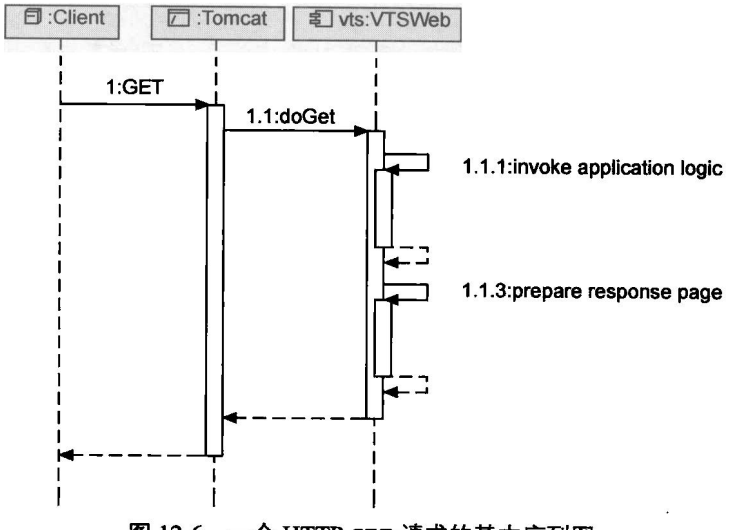


图 12-6 一个 HTTP GET 请求的基本序列图

客户状态管理

Web应用面对的最令人感兴趣的问题之一是，如何在无连接环境中管理服务器上的客户状态。既然客户和服务器之间做出的每一个请求和响应都由一个新的和唯一的连接完成，那么服务器跟踪任何一个特定客户的请求序列将是困难的。

对很多应用来说，管理状态是重要的，因为单个用例场景经常涉及在很多不同Web页面间导航。如果没有状态管理机制，将不得不为每个新的Web页面不停地提供所有之前输入过的信息。即使对于最简单的应用，这也是让人厌烦的。想像一下，每一次访问时，不得不从头重输购物车内容，或者当检查基于Web的e-mail时，为被访问的每一个屏幕输入用户名和密码。

这个问题最通用的实现解决方案——原来由World Wide Web Consortium（W3C）提议，是HTTP状态管理机制或更流行的cookie。Cookie是Web服务器可以请求Web浏览器保持的一小块数据，随后在每一次浏览器请求来自服务器的HTTP资源时返回。通常，cookie的数据量很小，在100～1000字节之间。Web应用框架这样管理客户状态：在浏览器第一次和应用交互时，每次生成一个唯一的标识符，把这个ID放进浏览器的cookie。这个ID用作服务器上映射的一个键，包含该特定客户的所有状态信息。

URL重定向是管理客户状态的另一个方法。在这个方法中，每一个超链接和表单把ID放在URL的末尾，提交给服务器请求下一页面。该ID扮演和cookie一样的角色，但作为一个URL的参数，它可以用在显式关闭了cookie特性的浏览器。这个方法的缺点是，Web站点的每一个页面必须是动态的，用户不能在用例场景中间漂移到应用之外，以免客户和服务器之间的特定会话断开。

管理客户状态的另外两个方法是串行化整个状态到URL参数或cookie。由于各种原因，这些方法并不常用，至少这些方法天生缺少安全性，因为客户的状态被通过网络发回浏览器，而在之前的方法中，状态只是在服务器管理。当所有状态值都被放在cookie中，也会受到4K大小的限制，每个域最多有20个cookie。所有状态数据必须被编码成简单的文本（没有空白、分号等）。这也意味着不能容易地用会话状态中的高层次对象捕获客户状态。

这意味着，服务器应用如何可以跟踪特定客户的请求历史，并因此跟踪客户比较内部的状态，一点也不明显。例如，如果没有利用某个HTTP特性或者实现某个架构机制，对于一个服务器应用来说，即使对一个简单的购物车或一个特定用户走过一个多步骤的向导的状态，管理起来也会很困难。幸运的是，大多数Web应用环境提供很多有用的工具和机制来管理客户状态。

这个架构的另一个含义是，服务器不知道在一些业务流程的中间，用户是否抛弃了应用。有时一个远程用户从网络断开连接，因此不能完成已经开始的特定业务过程，这完全是有可能的。在更经典的客户/服务器应用中，服务器可以接收一个通知说客户过早断开连接；

但在一个 Web 应用中，当用户断开连接或简单地决定关掉浏览器时，没有通知被发送给服务器。

因此，Web 应用设计必须非常在意 Web 页面请求之间应该打开和访问什么资源。例如，Web 应用设计的一个根本规则是，决不在一个页面打开一个事务，在另一个页面关闭它。来自单个客户的 Web 页面请求之间的时间通常在秒级，随时会突然停止。在这种级别管理事务和锁肯定会使应用服务器垮掉。

12.2.4 实现视图

图 12-7 展示了一个休假跟踪系统的实现模型概览，并描述了系统的层和包。在这张图中，主要 Web 层组件 `VTSWeb.war`，包含 Web 页面工件（JSP 和 HTML 文件）以及 `web` 和 `sdo` 包中的一个 `Java` 类集合。这个代码用于处理和调用业务层中的 `EJB` 逻辑。`com.acme.vts` 包在后台，作为所有 `Java` 组件的主要命名空间。部署视图更多描述层中组件部署的细节，逻辑视图更多描述组件本质和责任的细节。

12.2.5 用例视图

通过实现基本的刺激/响应用例，部署、逻辑、处理和实现视图被绑在一起。客户向服务器做出一个 HTTP 请求，请求一个 Web 页面。服务器检查请求，并决定需要加载和执行什么应用或资源。一些资源会导致业务逻辑处理，另外一些则简单地显示静态的数据。当处理完成时，应用负责做出一个响应，通常以一个 HTMLWeb 页面的形式在之前 Web 页面的位置呈现。这个响应页面包含新的信息和用户调用或请求的选项。通过组装所有这些 Web 页面的集合，每一个页面将特化以显示和接受作为应用的一部分的信息，可以实现一个完整的业务过程。

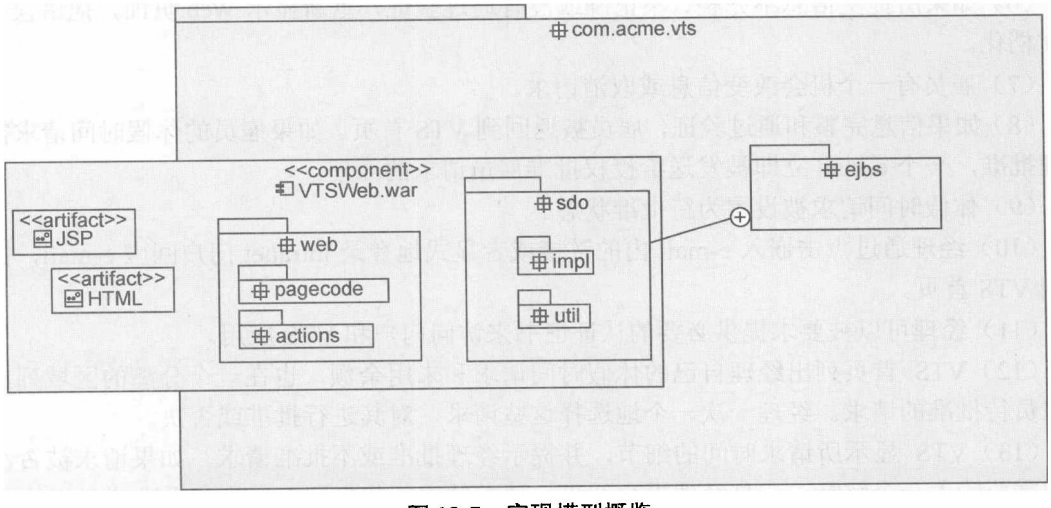


图 12-7 实现模型概览

在本章中，我们聚焦于一个在架构上有代表性的用例——Manage Time，这个用例被调用最频繁以及被系统的所有执行者查看最多。所以，高效实现这个用例和保证它满足所有的总体设计目标，包括易用特性，是关键的。

用例规格可以使用很多不同模板。本章中使用的格式展示了流程中更健壮的步骤，不使用 Web 应用可能创建的刺激/响应风格流。这种用例规格风格也适用于存在复杂或健壮的备选流的情况。最终使用的风格或格式由你和你的组织所要求的规范级别而定。

以下例子给出 Manage Time 用例规格的概览。

用例名：Manage Time

执行者：Employee

目标：雇员希望提交一个新的休假时间请求。

前置条件：雇员已经由门户框架认证并被识别为一个公司雇员，有管理他自己休假时间的权限。

主流程：

（1）雇员从 intranet 门户选择一个到 VTS 的链接，用例开始。

（2）VTS 使用雇员的证书查询所有雇员的休假时间请求的当前状态以及未用余额。将显示从之前 6 个月到将来 18 个月的信息。

（3）雇员要求创建一个新的请求。雇员选择一个有正余额的休假时间类目。

（4）VTS 提示雇员请求休假的日期和时间。雇员应该可以通过一个可视日历来帮助选择和比较所选日期。

（5）雇员选择所要的日期和每日小时数（例如，4 小时可能表明这是一个半天的休假时间请求）。雇员输入简短的标题和描述（长度不超过一个段落），这样，经理将有更多信息来决定是否批准这个请求。输入所有信息后，雇员提交请求。

（6）如果所提交信息不完整、不正确或没有通过验证，重新显示 Web 页面，把错误高亮和文档化。

（7）雇员有一个机会改变信息或取消请求。

（8）如果信息完整和通过验证，雇员被返回到 VTS 首页。如果雇员的休假时间请求需要经理批准，一个 e-mail 立即被发送给授权批准雇员请求的经理。

（9）休假时间请求被设置为待批准状态。

（10）经理通过点击嵌入 e-mail 内的链接或者显式地登录 intranet 门户回应 e-mail，并导航到 VTS 首页。

（11）经理可以被要求提供必要的认证证书来访问门户和 VTS 应用。

（12）VTS 首页列出经理自己的休假时间请求和未用余额，也在一个分离的区域列出下属雇员待批准的请求。经理一次一个地选择这些请求，对其进行批准或否决。

（13）VTS 显示所请求时间的细节，并提示经理批准或不批准请求。如果请求被否决，经理需要输入一个解释。一旦经理提交结果，请求的内部状态即改变为已批准或已拒绝。

（14）不管一个请求被批准还是拒绝，一个 E-mail 通知将立即被发送给做出请求的雇员。

经理的屏幕返回到 VTS 首页，经理可以批准其他待批准请求，作出自己的请求，或者简单地离开 VTS 应用。

备选流：Withdraw Request

目标：雇员要求撤回一个待批准的休假时间请求。

前置条件：雇员已经做出一个休假时间请求，该请求还在等待一个被授权经理的批准或否决。参见主流程的前置条件。

（1）雇员通过 intranet 门户应用导航到 VTS 首页，门户标识和认证雇员有使用 VTS 的必需权限。

（2）VTS 首页包含一个概览，包括休假时间请求、每种时间类目的未用余额，以及之前 6 个月到将来 18 个月之间所有活动休假时间请求的当前状态。

（3）雇员选择要撤回的休假时间请求，该请求当前处于待批准状态。

（4）VTS 提示雇员确认撤回之前提交的休假时间请求的请求。

（5）雇员确认想要撤回，该请求从经理的待批准列表中被移除。

（6）系统发送一封通知 E-mail 给经理。

（7）系统更新请求状态为已撤回。

备选流：Cancel Approved Request

目标：雇员要求取消一个已批准的休假时间请求。

前置条件：雇员有一个休假时间请求已经被批准，并进行了一些最近（之前 5 个业务日）或将来的时间调度。参见主流程前置条件。

（1）雇员通过 intranet 门户应用导航到 VTS 首页，门户标识和认证雇员有使用 VTS 的必需权限。

（2）VTS 首页包含一个概览，包括休假时间请求、每种时间类目的未用余额，以及之前 6 个月到将来 18 个月之间所有活动休假时间请求的当前状态。

（3）雇员选择要取消的休假时间请求，该请求时间在将来（或最近），请求已经被批准。

（4）如果该请求是在将来，雇员被提示确认此次取消；如果该请求在前面几天，雇员被提示确认取消和提供一个简短的解释。如果雇员认可取消，并提供所需信息，一个 E-mail 通知被发送给经理，该请求的状态变为已取消，用来做出该请求的时间额度被还给雇员。雇员也可以中止该取消，对当前请求不造成影响。

（5）雇员被返回 VTS 首页。概览被更新以反映雇员对待批准休假时间请求所做出的改变。

备选流：Edit Pending Request

目标：雇员要求编辑待批准的请求的描述或标题。

前置条件：一个雇员已经做出一个休假时间请求，该请求还在等待一个被授权经理的批准或否决。参见主流程的前置条件。

（1）雇员通过 intranet 门户应用导航到 VTS 首页，门户标识和认证雇员有使用 VTS 的必需权限。

（2）VTS 首页包含一个概览，包括休假时间请求、每种时间类目的未用余额，以及之前

6 个月到将来 18 个月之间所有活动休假时间请求的当前状态。

（3）雇员选择要编辑的请求，该请求正在等待批准。

（4）VTS 显示一个可编辑的请求视图。雇员可以改变标题、评论或日期，也可以选择删除或撤回这个请求。

（5）雇员改变请求信息，并提交改变给系统。

（6）如果雇员撤回请求，VTS 在撤回请求之前提示确认；如果改变只是针对信息，改变被接受，屏幕返回到 VTS 首页；如果改变的信息有错误或问题，VTS 重新显示正在编辑的页面以及高亮并解释所有问题。

这个用例描述包含较多信息，包括关于对 VTS 的建议以及期望典型的雇员如何使用。很大程度上，这是一份从 Employee 执行者的观点得到的系统功能描述。实际上，这正是用例要做的。不幸的是，这个描述甚至不足以开始分析，还需要非功能需求和更多领域信息。非功能需求通常包括环境、性能、可扩展性、安全性等方面的需求。领域知识可以以离散需求的形式存在，例如：

- 所有雇员一天工作 8 小时；
- 每个雇员的休假时间请求受限于每个雇员的首要工作地点，除了公司政策的总体限制之外；
- 休假时间请求验证规则由 HR 部门定义和拥有。

这些类型的需求或知识可以被嵌在用例规格中，它们也可以被捕获作为离散的需求条目。这种类型的领域知识也可以单独写成文档，被共同引用。例如，验证一个休假时间请求的详细政策和规则是公司的雇员手册的一部分，很可能可以通过各种来源得到（如 intranet、表单、文档、新雇员入职简介等）。一个项目的需求集简单地引用这些已有的文档，而不是试图复制它们。

12.3 构造

就像大多数类型的软件应用一样，分析一个潜在的 Web 应用时，最重要的任务是系统实体和加工的识别。一个应用的实体和加工代表业务领域里的概念，最理想的情况是独立于架构，但不一定要这样。在一个 Web 应用中，一个关键的工件集合是导航图和 Web 页面定义。粗略地说，这些元素对应于经典的分析构造型：实体、控制器和边界。我们将从一个 Web 为中心的模型开始——用户体验（UX）模型。

12.3.1 用户体验模型

UX 模型[1, 2]是在一个足够抽象的级别捕获 Web 应用用户界面元素的例子，以便表达系统 Web 页面之间具体的导航图，而忽略风格（字体、字号、颜色等）和其他用户界面特定的元素——这些元素最好在后面的过程被开发。图 12-8 展示了一个描述用于实现我们的首要用

例的屏幕的 UX 模型的高级别片段。在这个模型和图中有两个关键构造型——<<Screen>> <<Form>>。<<Screen>>构造型的类代表一个显示给最终用户的用户界面的完整单元，或者粗略地说，一个 Web 页面。一些包含 HTML 表单元素的屏幕，用来将用户输入的信息提交回服务器。这些构造型的类总是由一个屏幕包含，在提交时会导航到系统中的另一个屏幕。单向关联关系表明屏幕的导航路径。

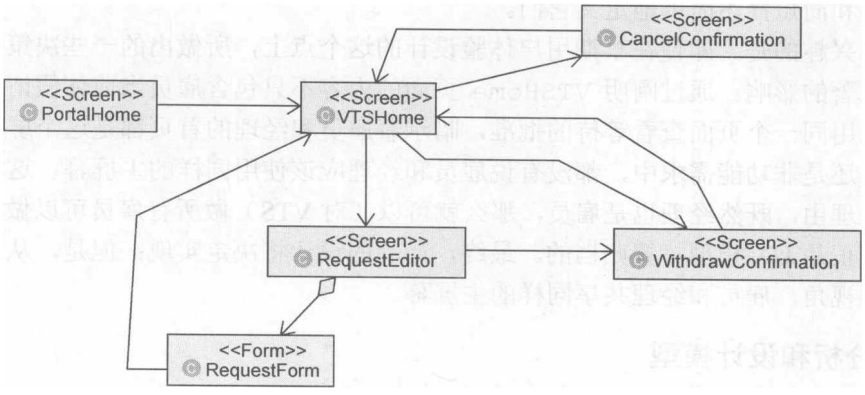


图 12-8 一个高级别用户体验模型

对于理解系统屏幕的导航流，图 12-8 是最有用的，但它提供屏幕的内容很少。包含屏幕内容的更详细的图在这个抽象级别也是有用的。图 12-9 展示了 VTSHome 屏幕的内容。<<Screen>>构造型类的属性表明离散的数据值，通常是字符串或容易呈现成 HTML 的简单类型。employee name 和 current date 属性通常用在头部或屏幕顶端。message 属性用来在动作发生之后显示一个信息或错误消息。例如，在雇员完成新的休假时间请求之后，返回到主屏幕，该消息可能说一些“你的请求需要你的经理批准，已经 E-mail 通知他”之类的话。

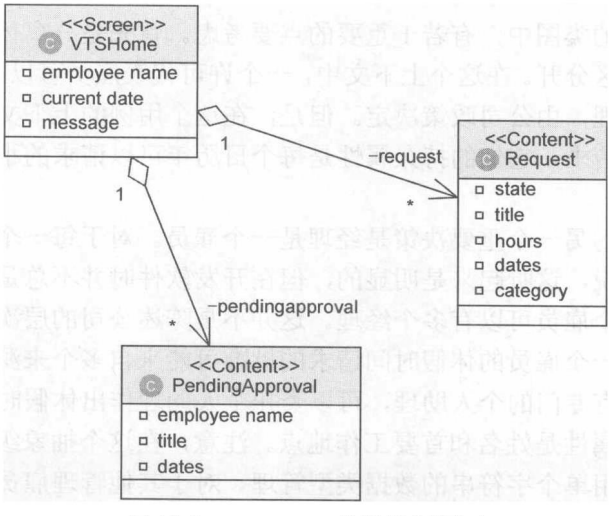


图 12-9 VTSHome 类的详细视图

屏幕中的内容经常是多值和复杂的。为了准确表达这种类型的内容，我们定义另一个类构造型——<<Content>>，将其应用于标识凝聚在一起的一捆信息的类。内容条目经常用作列表中的一行。在图 12-9 中，类 Request 和 PendingApproval 被建模为内容条目，被包含在主屏幕中。屏幕可能显示每个类的多个实例。Content 类定义的属性类似于屏幕的属性，容易以 HTML 的方式呈现。在这个高级别的模型中，担心实际的数据类型不重要，重要的是用名字和简短描述简单地定义它们。

令人感兴趣的是，即使在早期用户体验设计的这个点上，所做出的一些决策对最后的设计也会有显著的影响。通过阐明 VTSHome 页面的内容不只包含雇员当前休假请求的概要，而且经理也用同一个页面查看等待的批准，暗示着雇员和经理的首页都是这个屏幕。无论是用例规格中还是非功能需求中，都没有说雇员和经理应该使用同样的主屏幕。这样一个假设并不是没有理由，既然经理也是雇员，那么就可以（对 VTS）做所有雇员可以做的事情。像这样的决策正是 UX 模型打算归档的。最终，最后的设计将决定实现；但是，从逻辑（和用户体验）的视角，雇员和经理共享同样的主屏幕。

12.3.2 分析和设计模型

在分析中，要捕获系统的实体和加工。既然知道这个系统将实现为一个 Web 应用，就可以少关心一些边界，因为它们在 UX 模型中被探索过了。分析模型是识别解空间元素的第一个尝试，它使用领域的词汇表。这意味着大多数分析模型的元素名字对应于需求描述和领域及最终用户识别的事物和活动。

容易的开始方法是对用例规格和相关需求文档做一个简单的名词－动词分析。重要的名词通常代表模型中的类，动作短语（动词）倾向于代表那些类的操作，属性和关系代表模型中所捕获类的自然的内在属性。图 12-10 表达了一个开始点，聚焦于我们的首要用例中表达的主要领域类和概念。

在图 12-10 所示的类图中，有若干重要的点要考虑。首先是分离休假时间请求的想法，把它和休假时间许可区分开。在这个上下文中，一个许可代表雇员可以抽出来请事假的时间。许可由人力资源部管理，由公司政策决定。但是，在这个用例的上下文中，它们的首要责任是创建新的休假时间请求，它们的持久属性是每个日历年可以请求的小时数以及许可何时过期（如果有的话）。

图 12-10 中捕获的另一个重要决策是经理是一个雇员。对于每一个熟悉这些通用业务术语的流行用法的人来说，这听起来是明显的，但在开发软件时并不总是能做这样的假设。类图更进了一步，说一个雇员可以有多个经理。这并不是陈述公司的层次和组织，在这个上下文中，它意味着对于一个雇员的休假时间请求的批准可能来自多个来源。例如，在一些公司中，排位高的主管会有专门的个人助理，可以委托个人助理作出休假时间决策。

雇员的其他固有属性是姓名和首要工作地点。注意，在这个抽象级别和所需的用法，一个雇员的姓名只需要用单个字符串的数据类型管理。对于其他管理雇员信息的应用，很可能需要头衔、姓、名和中间名的分离的值，但在这个应用中，这些区分不需要，因为雇员姓名

唯一被使用的地方是在主屏幕显示。这是个普遍规则：除非问题绝对需要一个复杂的解决方案，否则不要主动提出。

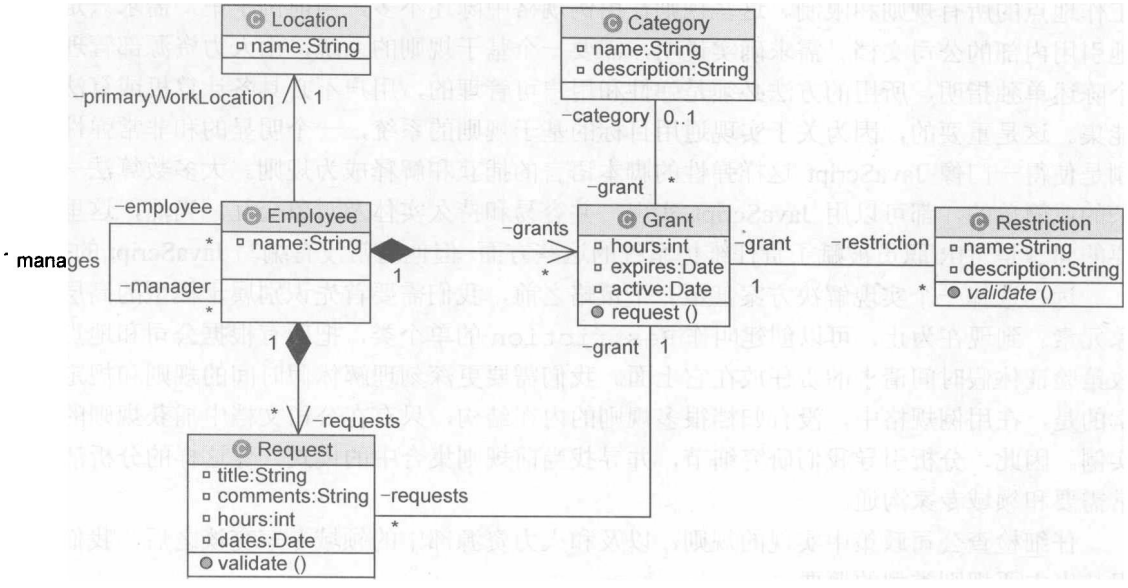


图 12-10 支持首要用例的分析模型类图

对大多数分析模型来说，另一个令人感兴趣的重要方面是缺少标识符定义。在有持久实体的几乎每一个系统实现中，都存在唯一 ID 的概念。例如，大多数公司分配一个雇员 ID 给每个雇员。这个 ID 唯一地定义特定雇员，如果雇员离开，然后返回到公司，这个 ID 经常会被重复使用。因为姓名的唯一性无法保证，所以确实需要 ID。然而在我们的模型中，没有 ID 属性的定义。这是因为在大多数分析模型中，可以安全地假设 ID 和其他用架构来实现分析所必需的属性可以按需要添加。像这样的信息，除非有非常特定的业务需要，在分析期间没有必要指定。例如，假设我们的系统有另一个执行者叫做 `Auditor`，它负责浏览所有休假时间的请求和检查是否遵守规定。该执行者确实需要雇员 ID 来交叉引用其他雇员数据。在这个案例中，显式指定雇员的这个属性是重要的。

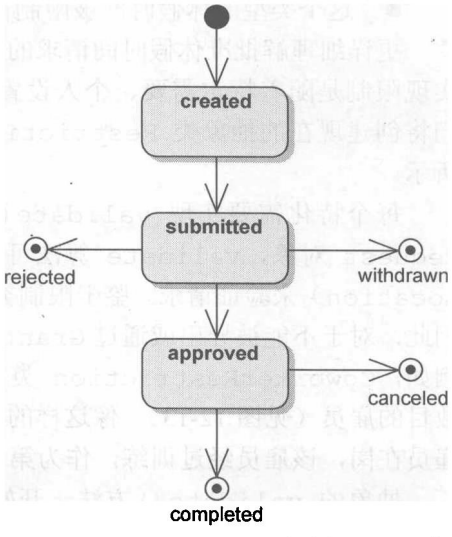


图 12-11 `Request` 类的状态机

模型中一些令人感兴趣的元素可以用状态机精化。例如，`Request` 类有一个定义好的状态机，如图 12-11 所示。就状态机而言，虽然，不是那么令人感兴趣，但它是有意义的。它告诉我们，对于一个 `Request` 来说，状态很重要而且已经被很好地定义。在图 12-11 中，可以看到 3 个过渡状态和 4 个最后的

状态定义。通过这些状态的路径非常简单，在该图中没有被标出来。

分析最困难的部分是创建相关元素，以保证任何所请求的休假时间通过关于公司和首要工作地点的所有规则和限制。这些规则在用例规格中陈述不多，当前例子中，需求只是简单地引用内部的公司文档。需求确实说明，需要一个基于规则的系统，由人力资源部管理。这个陈述单独指明，所用的方法必须是弹性和用户可管理的，用户不必具备计算机或算法的技能集。这是重要的，因为关于实现通用目标的基于规则的系统，一个明显的和非常弹性的机制是使得一门像 JavaScript 这样弹性的脚本语言的捕获和解释成为规则。大多数算法——复杂的或简单的，都可以用 JavaScript 实现，并容易和持久实体及对象交互。当然，这里的主要的问题是，HR 职员被赋予责任维护系统的这些方面，但他通常没有编写 JavaScript 的技能。

远在选择一个实现解决方案甚至一个策略之前，我们需要首先识别属于模型的高层次抽象元素。到现在为止，可以创建叫作 Restriction 的单个类，把所有根据公司和地点特定政策验证休假时间请求的责任放在它上面。我们需要更深刻理解休假时间的规则和规定。不幸的是，在用例规格中，没有归档很多规则的内在结构，只有在公司文档中捕获规则的特定实例。因此，分析引导我们研究细节，并寻找当前规则集合中的模式。像这样的分析活动经常需要和领域专家沟通。

仔细检查公司政策中实现的规则，以及和人力资源部中的领域专家交谈之后，我们可以组装出主要规则类型的概要：

- 对于 Y 类型的许可，一个雇员不能申请连续超过 X 天的离开时间；
- 当直接和公司或地点特定的假日相邻时，类型 X 的休假时间不能被接受；
- 类型 X 的休假时间被限制在 Y 小时每星期或每月；
- 当只有 X 数量的雇员调度从事来自雇员列表 Y 工作时，休假时间不可以被许可；
- 在这些日期 X，休假时间不可以被许可；
- 这个类型的休假时间被限制在星期中的某一天{M，T，W，Th，F，Sat，Sun}。

更详细理解批准休假时间请求的潜在规则后，我们的模型会有一些重要的改变。显然，实现限制是随着特定需要、个人设置信息而变化的。在试图制定一个解决方案策略时，我们将创建现在的抽象类 Restriction 的特化，系统中的每个规则类型一个，如图 12-12 所示。

每个特化需要实现 validate()，这个方法接受一个 Request 对象作为参数。从 Request 对象，validate 算法可以导航到大多数它需要的信息（Employee、Grant、Location）来验证请求。鉴于限制类型的多变本质，已有的信息并不一定都足以作出验证。因此，对于不能被导出或通过 Grant 对象导航的对象，每个特化会管理一个属性或关系集。例如，CoworkerRestriction 类需要管理合作者列表，以及识别允许调度来工作的最小数目的雇员（见图 12-13）。像这样的规则经常和安全问题关联（例如，必须总是有至少一个雇员在岗，该雇员经过训练，作为第一后援）。

抽象的 validate()方法一开始只是返回一个简单的布尔值，表明请求通过还是未通过。但如果靠近一些看需求，将会看到用户需要被通知，给他一个解释，是否他试图提交一

个无效的休假时间请求。既然验证规则现在被封装在每个 Restriction 特化中，看起来，这些类提供一个机制来把这个信息返回给任何调用过程是适当的。同样，既然可以发生多于一次违规，那么验证的结果应该作为一个集合来访问。这些需求的结果引导我们创建一个新的类 ValidationResult（见图 12-14），它由 validate() 方法返回。

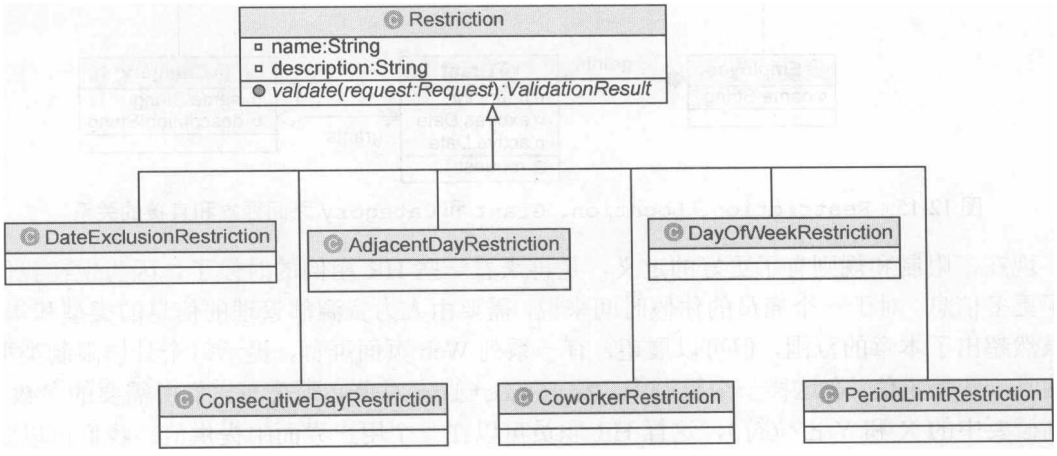


图 12-12 Restriction 类层次

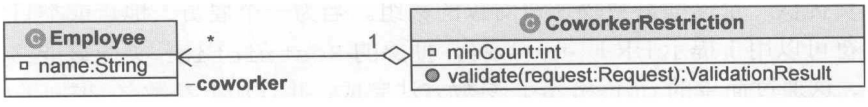


图 12-13 CoworkerRestriction 类和属性

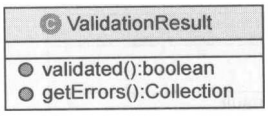


图 12-14 ValidationResult 类

ValidationResult 类确实十分简单。它定义两个方法：validated() 返回一个简单的布尔值，表明已验证的请求；getErrors() 返回一个字符串消息的集合，每个对应于对请求检测出的问题。在分析级别，保持类这样简单就够了，让设计活动解决实现细节。这里的要点是，必须有可能存在一个过程来验证休假时间请求，使得每个错误结果都是可得的，并提供一个简单的方法来决定有效性。

另一个对这里演进的规则系统的观察是，限制不可以和特定 Grant 或 Employee 对象关联，而是被广泛应用于一个 Location 或 grants 的 Category。这样，我们早先的分析模型必须改变（见图 12-15）。建立这些关系使得人力资源部应用一个广泛的共同规则类更加容易，不必为每个雇员复制它们。显然，在验证期间可以导航到这个信息，设计变化的动机是基于使得系统更容易使用的强烈欲望，在这种情况下，对于 HR 人员来说，易用性会显著改善。

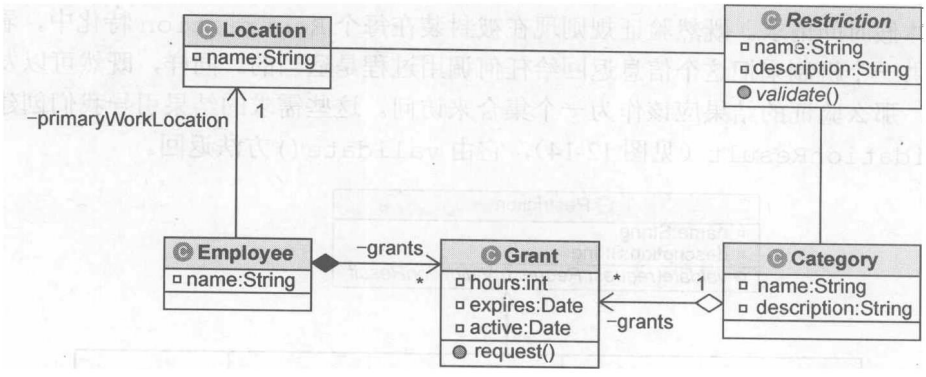


图 12-15 Restriction、Location、Grant 和 Category 之间独立和直接的关系

现在，限制和规则有了更好的定义，是再来看一些 HR 用例的时候了，因为现在我们知道了更多信息。对于一个雇员的休假时间来说，需要由人力资源部管理的信息的类型和本质。这虽然超出了本章的范围，但可以展望，有一系列 Web 页面屏幕，提示每个具体限制类型需要的唯一和特定值。在这样一个机制中，Restriction 对象必须公布它们所需要的参数（即规则概要中的 X 和 Y 占位符），这样 HR 职员可以在一个用户界面中提供值。我们可以定义一个辅助类 RestrictionParameterDescriptor 来封装每个这些值。Restriction 的具体实现负责提供一个这些参数描述符对象的数组。当为一个雇员、地点或类目定义新的限制时，描述符可以用于提示 HR 职员。最终，抽象的 Restriction 类应该能够接受和提供个体参数值，这通过简单的 put 和 get 参数方法完成，并且加上参数名。图 12-16 阐明了这两个类。

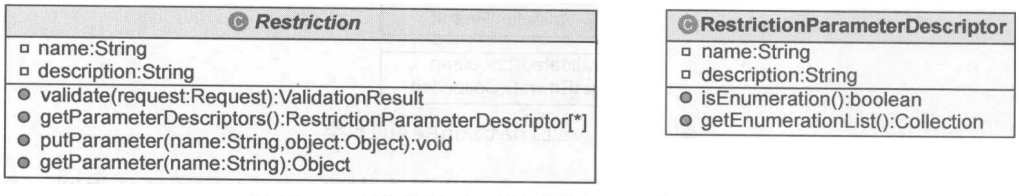


图 12-16 使限制更容易通过编程和用户界面管理

在分析期间的各种点上，对模型中出现的想法做测试是有用的。但是，既然我们仍然独立于一个可以为之书写代码的真实具体的架构，就必须更抽象一点来分析我们的模型。可以通过在模型元素上执行思考实验来做到这一点，具体地说，试图适度理解特定场景如何由模型中归档的结构和行为来执行的细节。处理限制的模型区域是一个好的例子，需要用真实的数据来验证，至少典型和最重要的用例可以这样实现。

UML 提供了一对有用的工具：对象图和通信图。在这里，我们的任务是评估模型验证新的休假时间请求的能力。要保证行为能够根据需求，并使用模型中已定义的类和对象来完成。在这个思考实验中，我们用这些图来理解工作在 North Factory、名字叫 Jim 的雇员的一个新请求如何被执行。

图 12-17 展示了虚构的雇员 Jim 的一个对象集的对象图，该雇员要求使用作为按时完成工作的额外奖励的休假时间许可。该图表明，通过依赖，可以访问或导航其他对象实例。实例有现实生活中的名字，和场景相适应，后面跟着一个冒号以及对象的类或类型。这张图用几个随意的矩形来注解，有助于强调和澄清三个限制来源：Grant、Location 和 Category。

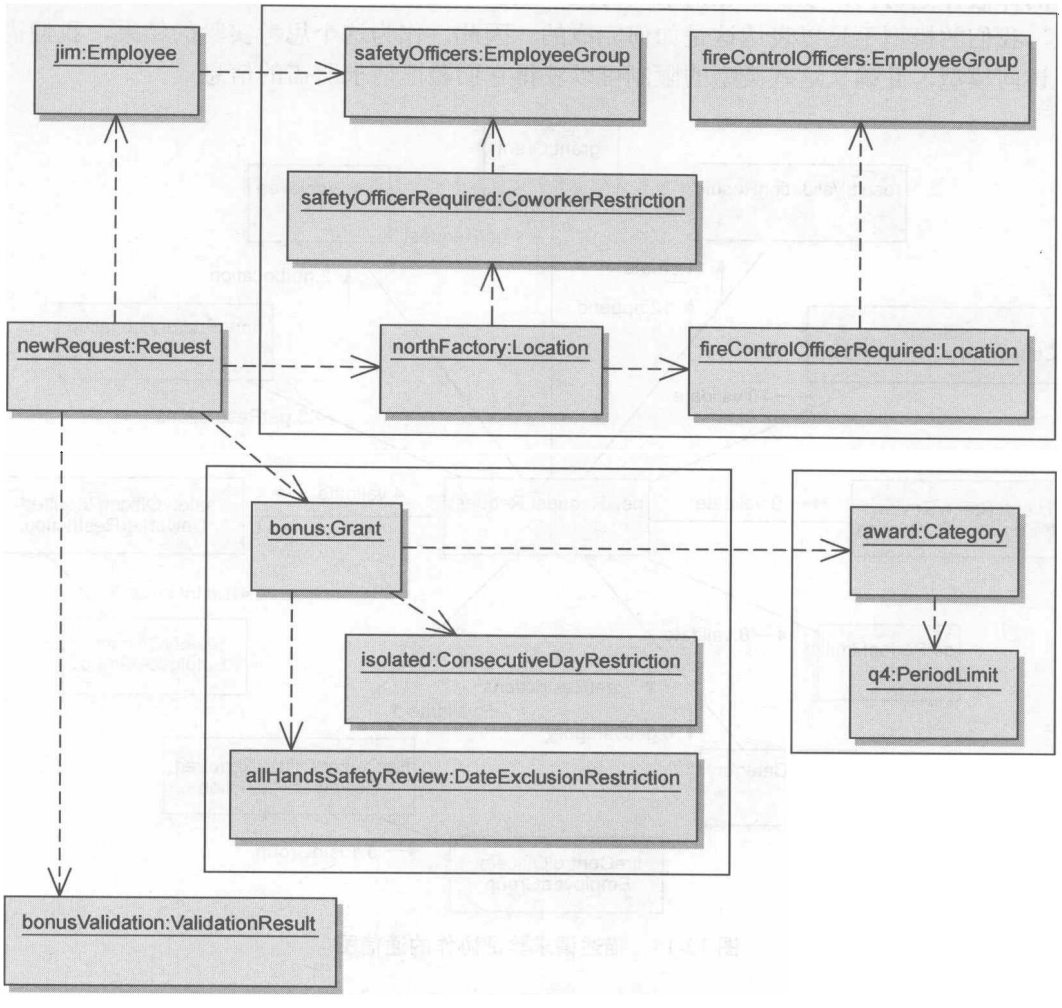


图 12-17 描述请求验证协作的对象图

在图 12-18 中，我们用一张通信图来为 Request 验证检查通信路径。线代表对象实例之间的通信路径，而不是像类图中那些结构上的关系。沿着编了号的消息遍历通信路径，检查这个协作的行为。行为的主要协调者是 Request 实例，它在这个场景中做了大多数工作。通用流是为 Location、Category 等获取并验证各种限制，通过调用每个限制上的 validate()操作来达到。如果一个限制没有通过验证，它的消息被附加到验证结果实例。

一个要跟随的令人感兴趣的流是 CoworkerRestriction 的流。为了验证，CoworkerRestriction 不仅需要知道它代表哪一个 EmployeeGroup，也要知道要为哪一个 Employee 检查规章制度。快速浏览 validate() 的签名可以发现，Request 对象作为参数被传入，从它可以导航到 Employee。但是，这些限制也需要能够被小组中的所有其他雇员的休假计划访问，以保证在请求的时间，至少小组中的一个雇员被调度在位置上。到此为止，我们的模型不足以表达这是如何完成的。因此，作为这个思考实验的结果，我们需要重新访问模型，并确认这些类型的限制可以获得它们验证请求所需的信息。

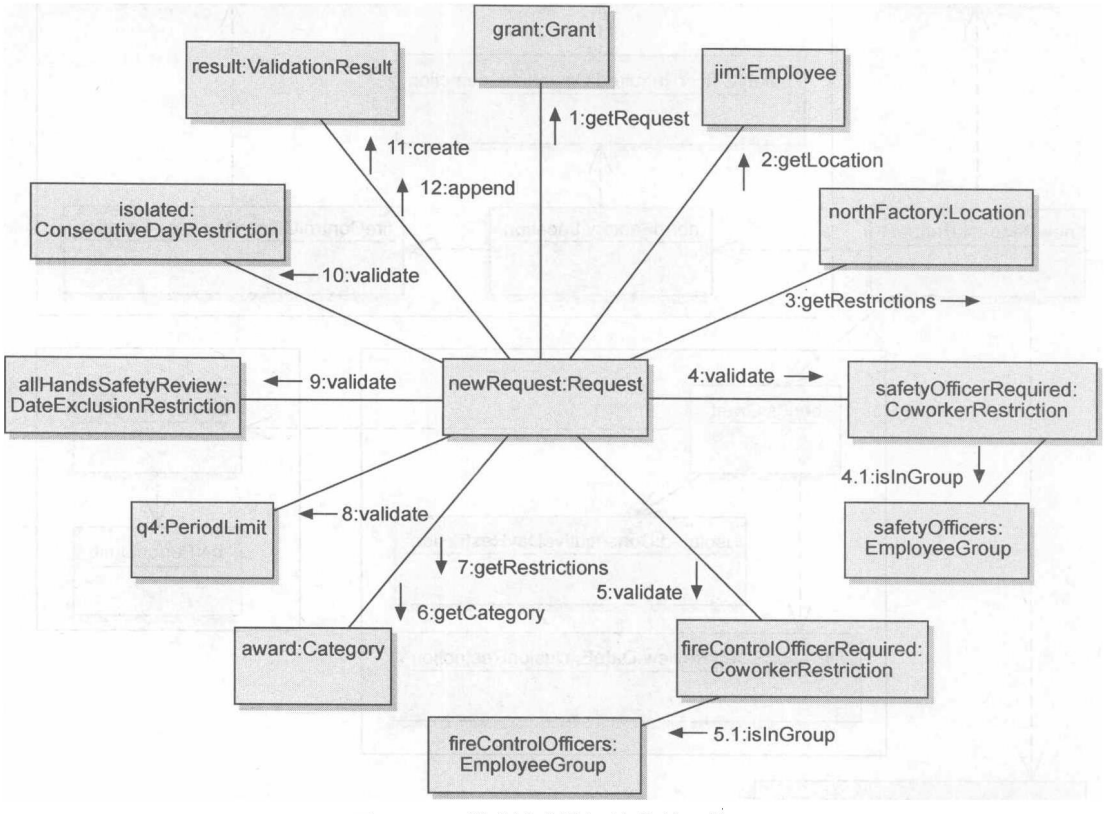


图 12-18 描述请求验证协作的通信图

正如第 6 章中讨论的，分析的开始和结束不是一个里程碑事件。分析活动经常贯穿整个开发生命周期，但绝大部分集中在至少理解了大多数需求的时候。当分析模型足够成熟时，就是开始设计的时候了。很明显，这是一个含糊的判断，但这是大多数项目的现实。决定何时开始设计活动的准则通常是非常本地化的，例如，和项目团队及组织的历史或架构的成熟度有关。

现在我们从分析转换到最终可以作为软件执行的设计。一些设计模型和可执行的模型（或实现模型）是等同的，而其他一些就足够抽象，实现时仍然需要某些技巧和工作，Web

应用通常是这些的混合。大多数业务层元素（实体和控制器）的映射十分接近于系统中所得的代码，但是，大多数表现模型在准备成为最后的系统之前，实现时需要大量的工作。

在本章讨论的 VTS 设计里，没有空间来覆盖 Web 为中心的应用所有令人感兴趣的设计方面和问题，即使是简短的概述。相反，我们只讨论一些最重要的设计点，特别是那些 Web 应用架构特有的。

除了基本的 Web 架构组件之外，这个项目的架构师决定使用以下组件和技术：

- 客户层　任何支持 HTML 3.2 的浏览器；
- 表现层　JSP（Java Server Pages）、Servlet 和 JSF（Java Server Faces）；
- 业务层　EJB（企业 Java Beans）2.x（具体地说，用于实体的容器管理持久性（CMP）bean、用于控制器的会话 bean）和 SDO（服务数据对象）；
- 安全性　中央认证服务（CAS）[1]。

决定在总体架构中使用基于 Java 的产品，不是因为基于 Java 的解决方案比起其他任何已有的技术能更好地满足所述需求，而是因为它们除了适合这个任务之外，碰巧更好地匹配了组织的当前技巧和工具集。这个组织过去已经成功地用核心的 J2EE 技术建造了若干基于 Java 的企业应用。开发人员熟悉这些技术，他们的工作站中已经有了建造和测试这些类型的应用所需的工具。就这样的情形而言，没有令人信服的原因让架构师切换整个技术框架。只有两个原因使得架构师考虑这样一个显著的改变。

（1）目前的经验展示出技术上有严重的问题，或者没有能力满足开发团队、维护团队和最终用户的需要。

（2）技术自身不再被支持，或者进化得很厉害，看起来就像新的技术。

在客户和表现层，首要的设计目标是实现一个直观的用户界面，该界面响应快速而且容易导航。界面不应该依赖于任何特定浏览器版本或特性，或者说得更准确一些，这个应用应该适合所有支持标准 HTML 3.2 的浏览器。即使编写本书时的 HTML 当前版本是 4.01，支持版本 3.2（及后续版本）的决策的驱动因素并不是哪一种浏览器容易得到或者流行，而是所需的最小功能。已经决定实现这个应用的所有必需特性能够用更老的 HTML 规格完成，因此，既然没有真正的需要来要求新的浏览器和排除旧版本，应用可以支持更广泛的客户配置选择。

除了核心的表现技术 JSP 和 Servlet 之外，架构师还决定使用 JSF 组件来协助开发用户界面。JSF 是一个 API 和定制的 tag 库，它包含很多组件，这些组件在 HTML 界面中显示表单，接受和验证输入，协助页面导航。在这里，一个值得注意的兴趣点是，决定不使用 Apache Struts 框架。Struts 是一个更老的模型—视图—控制器（MVC）框架，在 JSP 应用中依然非常流行。Struts 和 JSF 看起来配合得非常好，即使它们的一些功能有重叠。虽然 Struts 确实提供一些 JSF 的功能，它主要的强项是控制器特性。当一个 Web 应用实现长的（即很多 Web 页面）业务流程时，Struts 特别有用。我们的应用的业务流程相对较短，Struts 框架带来的价值不够显著，反而带来需要实现额外层的复杂性。

---

[1] 更多信息，参见 www.ja-sig.org/products/cas/。

在绝大多数(即使不是所有)的应用中，逻辑是在业务层执行的，它运行在一个J2EE Web和EJB容器中。决定Web和EJB容器是否运行在同一台机器上，是否需要群集或其他这样的配置信息，不是一个设计考虑，除非已经有条件做出这样的决策。选择基于J2EE的架构的一个优势是，系统开始部署时，可以晚一点做出这样的部署选择，或者可以按需改变。在设计和实现活动期间，要记住的重要事情是，不要创建或编码任何东西来妨碍J2EE容器迁移或部署组件的正常能力。这样的开发实践经常出现在开发团队必须遵循的编码或设计指南中。

决定使用CMP机制来存储所有持久数据。有很多理由使用这个机制，也有很多理由不使用这个机制。其他持久化选项包括使用POJO（Plain Old Java Object）和关系/对象持久化层，如Hibernate、Castor或Apache ObJectRelationalBridge（OJB)。这些选项通常更轻量、更有效率，特别是在单个节点运行应用时。既然这个应用的性能需要没有足够显著到值得用另一个技术或额外的层，所以我们决定使用已有容器内建的CMP机制来持久化。另外，使用SDO来管理表现和业务层之间的数据流，可使得管理实体数据更容易。

这个应用的架构不容易被抛弃的一点是使用CAS来实现单点登录。单点登录的思路是，在同样的浏览器中，一个最终用户只需要提供一次认证证书来访问整个范围相关或不相关的Web应用。这早在创建愿景时已经决定——VTS是公司已有的intranet门户系统一个扩展，而系统当前使用CAS来做认证管理。因此，新的VTS应用也不得不使用CAS来识别和认证最终用户。

12.3.3 实体

设计和实现实体的最重要部分是识别它们。在任何给定的分析模型中，任何有属性的对象，甚至一些没有任何已定义属性的对象，都可以被设计为实体。这里的窍门是，只选择模型中那些真正需要设计为持久实体的类。一些分析类，即使有了已定义的属性，可能只是瞬时类或值类。例如，`ValidationResult`类只用作`validate`方法的返回值。验证不被记日志或者记录在这个系统中，因此`ValidationResult`类不需要持久。当一个请求被验证，一个这个类的实例被返回给调用者，以检查验证结果。完成时，结果不再需要或被引用。因此，`ValidationResult`类可以保持为简单的POJO。

VTS将使用CMP bean来管理它的所有持久实体。从第一次被引入以来，CMP已经走了很长一段路。一开始有很多问题，特别是在关系和性能上。但是，更新的规格和改善过后的EJB容器已经解决了很多问题，使得CMP成为管理实体的优选机制。

当前，已经围绕CMP bean开发了若干设计模式。一般推荐只使用CMP bean来定义本地接口。一个本地接口只能被同一容器里的另一个bean访问。比起通过一个支持跨节点通信的基础设施，通过本地连接bean来整编数据要高效得多。这些对象由一个外观对象访问，该外观对象有一个到CMP的本地连接，也公布一个远程接口，由其他会话bean和远程客户访问。

SDO用于整编进出表现层的数据，这些对象由业务层中的外观对象创建和管理。SDO是两个大的J2EE容器厂商BEA和IBM最近合作的结晶，代表处理EJB应用数据某些最佳

实践的普遍认同。

对于 CMP bean 来说，另一个重要的设计模式是，它们应该被设计成带有最小的业务级别行为（如果有）。事实上，最容易把 CMP 的定义想成包裹在数据库表外面的薄层。所有业务级别行为被放在会话 bean 外观对象中，会话 bean 外观对象负责统筹各种 CMP bean 中的状态，以完成一个业务行为单元。这对如何设计和实现分析实体有直接影响。

看一看 Category 类，可以开始理解所有这些设计级别的模式和惯例在代码中如何实现。这个类相对简单，因为它只定义两个持久属性，加上一个到 Grant 类实例的关系。幸运的是，在分析级别，Category 不定义任何有意义的行为。如果它定义了，需要把它从实体移除，放在一个外观对象里。在这里，我们以一个非常简单的实体开始。

设计实体用<<Entity bean>>构造型类表达。这些类定义它们的持久属性，主键属性被进一步构造型化。一个<<Entity bean>>类的方法被隔离成本地和远程（虽然根据我们的设计实践，在这个例子中 Category 没有定义任何远程方法），以及实例和类级别。这其中的每一个和用于定义它们的真实接口相对应。在 Category 类中，我们需要两个接口——CategoryLocal 和 CategoryLocalHome 来定义本地方法，本地方法在 bean 实例以及负责创建或查找实例的 home 或工厂对象上调用。图 12-19 展示了一张有到 Grant 实体[1]的关系的 Category 实体 bean 的类 UML 图[2]。

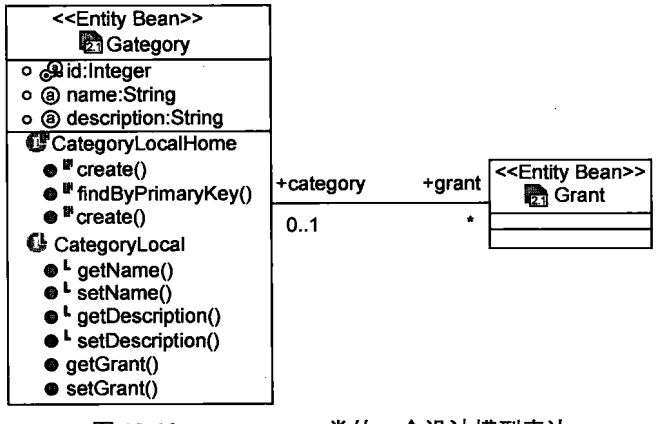


图 12-19 Category 类的一个设计模型表达

通过处理每个实体的单个模型元素，可以更容易理解它们的关系。但是，在实现中，任何给定的实体事实上由很多分离和不同的类、接口或配置文件实现。例如，Category 类需要两个分离的接口（CategoryLocal 和 CategoryLocalHome）和一个实现类（CategoryBean），和其他 bean 一起被打包进一个 Java 归档（JAR）文件，形成了 EJB 部署描述符文件和数据库中表的定义（图 12-20）。要认识到的重要事情是，当从分析向设计和

[1] 既然设计模型的目标平台是 J2EE 架构，使用平台特定的表示法是适当的。

[2] 这张图是一个实体 bean 的定制视图，在一个 J2EE 系统中，包括实现类及其接口和一些配置信息。同样，使用 getter 和 setter（提供访问对象属性的方法）不是一个面向对象传统，而是使用 Java 和 J2EE 框架的一个限定。

实现迁移时，通常有模型元素大的扇出。一个分析元素经常映射到很多实现元素，所有元素都需要协同。代码清单 12-1 展示了包含和 Category 实体相关的元素的 EJB 部署描述符片段。

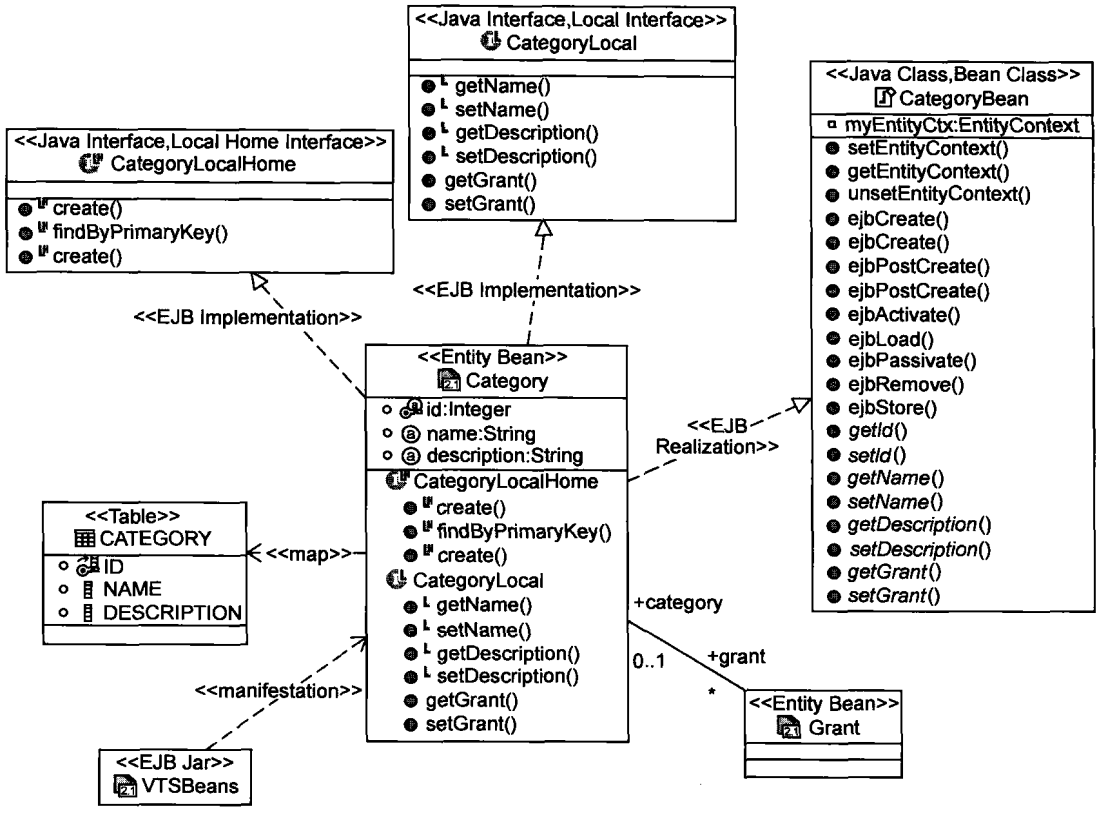


图 12-20 Category 类的设计模型元素

代码清单 12-1 包含和 Category 实体相关的元素的 EJB 部署描述符片段

```
<ejb-jar id="ejb-jar_ID" version="2.1"
xmlns="http://javA.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://javA.sun.com/xml/ns/j2ee
http://javA.sun.com/xml/ns/j2ee/ejb-jar_2_1.xsd">
  <display-name>VTSEJB</display-name>
  ...
    <entity id="Category">
      <ejb-name>Category</ejb-name>
      <local-home>com.acme.vts.CategoryLocalHome</local-home>
      <local>com.acme.vts.CategoryLocal</local>
      <ejb-class>com.acme.vts.CategoryBean</ejb-class>
      <persistence-type>Container</persistence-type>
      <prim-key-class>javA.lang.Integer</prim-key-class>
```

```
    <reentrant>false</reentrant>
  <cmp-version>2.x</cmp-version>
  <abstract-schema-name>Category</abstract-schema-name>
  <cmp-field id="CMPAttribute_1107874632507">
    <field-name>id</field-name>
  </cmp-field>
  <cmp-field id="CMPAttribute_1107874633048">
    <field-name>name</field-name>
  </cmp-field>
  <cmp-field id="CMPAttribute_1107874633068">
    <field-name>description</field-name>
  </cmp-field>
  <primkey-field>id</primkey-field>
  <query>
    <description></description>
    <query-method>
      <method-name>findAll</method-name>
      <method-params/>
    </query-method>
    <ejb-ql>select object(o) from Category o</ejb-ql>
  </query>
  </entity>
...
</ejb-jar>
```

跟随设计指南，我们为 Category 类定义一个外观（façade）[1]对象。会话外观的目标是简化接口，来做业务实体的共同任务。这经常涉及若干对象中方法的协同。会话外观提供一个更简单的接口，来查找、创建、修改和删除实体。幸运的是，我们的开发 IDE 很容易自动为实体 bean 创建会话外观。所得的外观对象叫作 CategoryFacade，就像实体 bean 的表达一样，它在远程和本地级别、实例和工厂级别都定义功能。在模型中，加上构造型<<facade>>的依赖表明设计模型所生成外观对象和设计模型实体表达之间的语义连接。外观类提供很多工具方法来创建、查找、更新和移除 Category 实体。实质上，它作为这个类的实例的单个源管理器。图 12-21 阐明了这样一个外观对象。

1. 服务数据对象

这个特定外观实现的另一个特征是使用新兴的标准——SDO。简单地说，SDO 是一种以从数据源断接的方式访问和操纵数据的机制，它提供了一种有用的方法来整编系统中的数据。SDO 使用断接数据图，使得客户可以检索来自于一个源的数据图，操纵它，然后把改变应用于原始数据源。

使用 SDO 让我们得到更一致的 bean 数据接口，遍及整个应用。在生成外观类时，也生成相应的 SDO 对象集。每个被创建的 SDO 对象定义两个接口：一个根对象（给数据图）和一个实体接口。（根对象代表数据图概念上的根，参见图 12-22。）

[1] 参见 Core J2EE Patterns [4]里描述的会话外观模式。

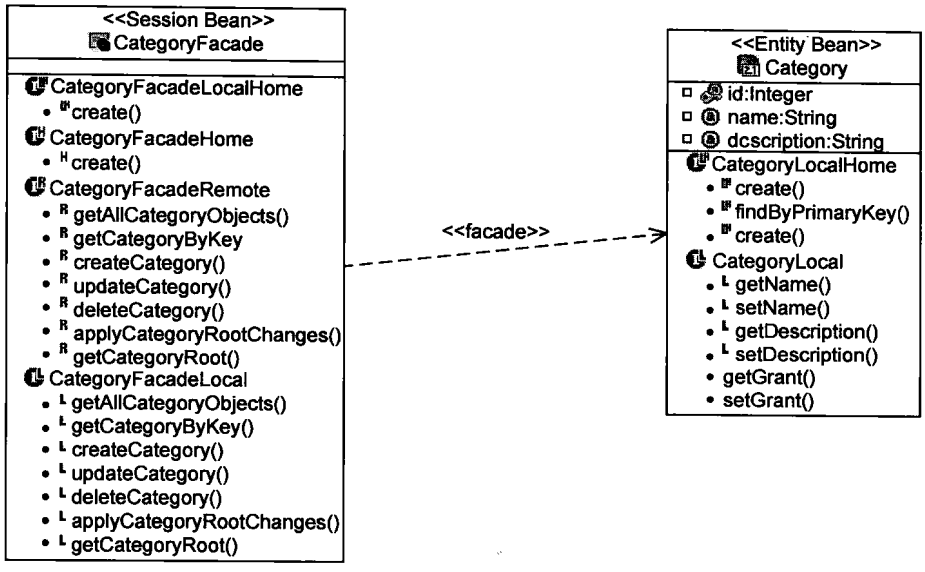


图 12-21 外观对象管理对实体 bean 的访问

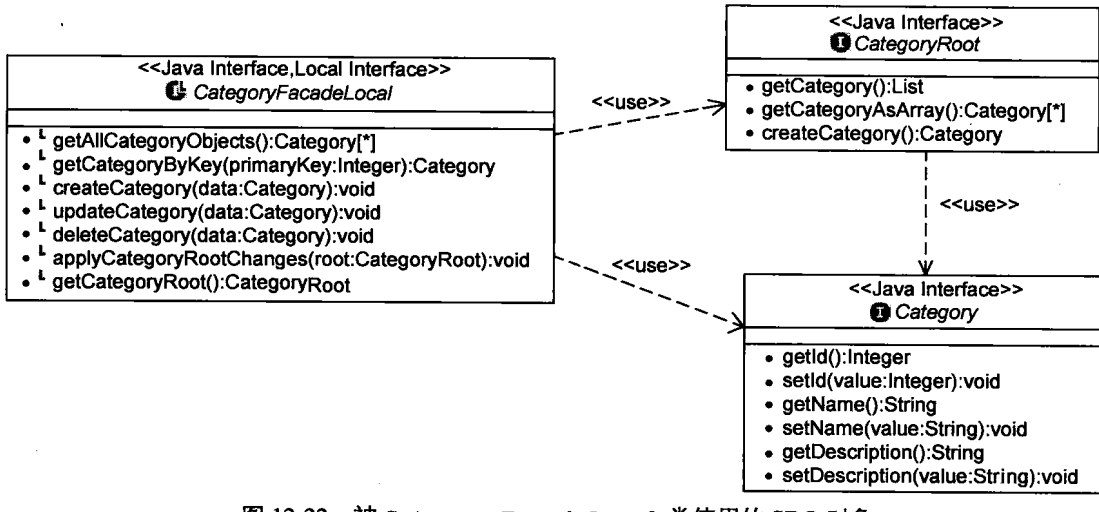


图 12-22 被 `CategoryFacadeLocal` 类使用的 SDO 对象

外观对象使用 SDO 对象作为它创建、检索、更新和删除（CRUD）方法的主要参数。它也提供一个方法来更新对整个 SDO 对象图所作的任何改变。这个外观的典型用法是，被另一个会话 bean 作为业务逻辑控制器使用，也许是履行 Web 页面的请求。让我们看一个样例场景：需要更新和创建休假时间请求类目。这个特定任务的责任落在 HR 职员角色上。这样，一个业务逻辑控制器需要能够获得所有已有的类目（通常只有一打左右）[1]，把它列在一个

[1] 在外观类中，随意使用 getAllObjects()方法会导致系统在真实情况下测试时出现意想不到的严重性能问题。想像一下，把任何中等规模业务的所有顾客地址集合放进一个数组的性能开销。

Web 页面上。它允许用户选择其中一个，进行编辑或移除。业务逻辑控制器可以容易地删除一个类目以及到 SDO 的引用或它的主键值。改变一个类目的值时，SDO 中的值做类似改变，然后告诉外观使用它的本地值来更新数据库中的持久实体。

2. 主键生成

所有对象—关系数据库系统的一个普遍问题是实体 bean 主键的生成。像社会安全号码或通用产品码（UPC）这样的自然主键值的获得，高度依赖于领域，不能总是假设其会存在。这意味着应用必须定义和实现一个策略，为没有自然键属性的新实体创建主键。在我们的系统中，实体 `Grant`、`Restriction` 和 `Request` 就是没有可以安全用作主键值的单个属性的业务对象。

在传统的数据库系统中，一个键通过若干属性和外键值的组合得来；但是，使用单个整数或字符串原生类型的主键值，大多数 EJB 设计会更高效。这意味着需要有一个为实体创建主键的策略。理想情况下，同样的策略和机制可以用于所有需要生成键值的实体类型。

在这个应用中，我们选择序列块（Sequence Blocks）策略生成主键，该策略在 Marinescu [5]中描述。这个策略精化了一个非常简单的方法，在本质上，该方法创建一个新的 CMP 实体类型，负责管理下一个可用作特定类型实体的键的可得整数值。这个方法的首要问题是性能，特别是在经常创建新实体的系统中。用一个保留大块潜在键值的会话 bean 管理对实体的访问，可以解决这些性能问题。这样，对持有键的 CMP 实体的访问稍微减少，减少的程度依赖于会话 bean 保留的键的长度。

在我们的应用加入这个策略，需要创建另一个实体和会话 bean，由其他实体外观对象作本地访问（见图 12-23）。`SequenceEntity bean` 只有两个属性：序列名（即需要一个生成主键的实体类型名称）和下一个可以安全用作主键的可得整数值。因为这个策略在一个时间里保留了成块的键值，在确实已使用的整数值序列中，完全可能有空白。所以，使用这个策略的应用不应做任何关于主键值次序或分布的假设。

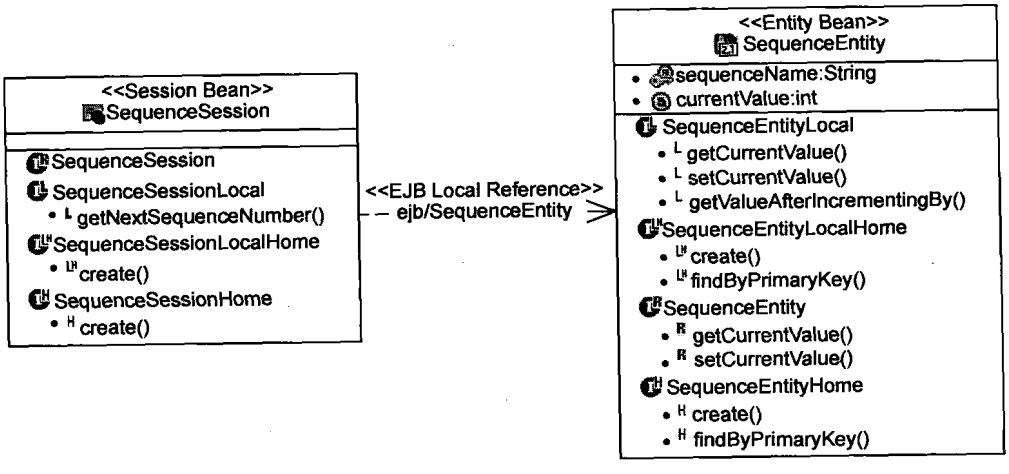


图 12-23 用于生成主键的实体和会话 Bean

当需要创建实体的一个全新实例时，键生成器被其他外观对象使用。这通过更新外观对象上的 createCategory()方法完成，来检查一个进入的 Category SDO 是否有一个空的 id 字段。如果 id 字段是空的，外观负责为这个类目生成一个新的键。这个过程如果失败，会导致抛出一个异常。

获得新的主键值很简单，访问Sequence Session bean，然后调用getNextSequenceNumber()方法，以实体名作为参数。createCategory 方法的代码如代码清单 12-2 所示。doApplyChanges()方法是一个 SDO 框架实现方法，一般说来，源代码是看不到的。

代码清单 12-2 createCategory 方法代码

```
public void createCategory(Category data)
            throws CreateException {
   try {
     if( data.getId() == null ) {
       InitialContext ctx = new InitialContext();
       SequenceSessionLocalHome home =
            (SequenceSessionLocalHome)
             ctx.lookup("java:comp/env/ejb/SequenceSession");
       SequenceSessionLocal sequence = home.create();
       int id = sequence.getNextSequenceNumber("Category");
       data.setId( new Integer(id) );
     }
     doApplyChanges(data);
   } catch (Exception ex) {
     throw new CreateException(
         "System error while creating \"Category\".", ex);
   }
 }
```

更新所有外观的 create 方法，并使用 SequenceSession 和 SequenceEntitybean，可以创建新的实体实例，而不用担心获得或计算所需的每个实例的主键（见图 12-24）。

3. finder

EJB 实体设计中另一个重要的考虑是 finder 方法的识别。这些方法负责搜索匹配给定条件集合的特定实体实例。条件可以很简单，如所有休假时间请求类目的实例（通常会得到不超过一打实例的列表）。finder 也可以实现更复杂的搜索，例如，查找在之前一年的 3 月 1 日和 6 月 23 日之间，特定事假类目的事假请求被拒绝的所有雇员。不管细节，关于 finder，重要的一点是匹配实体的过滤发生在服务器上，理想情况下，在数据库服务器上。通常，这是过滤大的实体集合最有效率的方法。唯一的另外选择是获得一个实体的所有实例，在它被潜在地通过网络传送和在另一个过程重新实例化之后，遍历所有实例并执行匹配测试。但这不是过滤大集合的最有效方法。

finder 背后的思想和 SQL SELECT 语句紧密相关。EJB finder 以另一种语言表达——EJB 查询语言（EJB QL），它和 SQL 非常相似，但不完全相同。例如，以下 EJB QL 语句返回没

有任何限制条件的所有 Grant 实体实例。

```
select object(o) from Grant o where o.restrictions is not empty
```

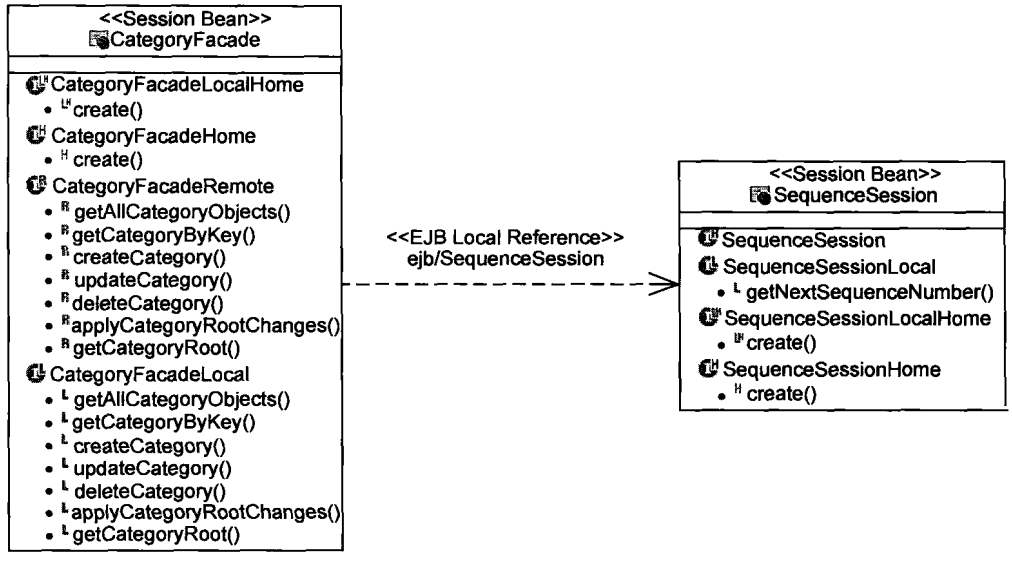


图 12-24 CategoryFacade 会话 bean 使用 SequenceSession Bean 创建新的类目

finder 也可以被参数化。在下一例子中，finder 查询返回排除所给日期的所有 DateExclusionRestriction 实例。在查询中，问号后跟着参数列表中的参数索引，以引用参数。

```
select object(o) from DateExclusionRestriction o where o.date is ?1
```

最终，这些查询被捕获，被放进一个主 EJB 配置文件 ejb-jar.xml，并和在 home 接口声明的 Java 方法关联。

12.3.4 控制器

即使在 Web 应用中，对于业务逻辑控制的概念也有很多方法。很多 Web 为中心的应用使用模型—视图—控制器（MVC）方法。当前在 J2EE Web 应用中实现严格 MVC 的最流行框架是 Apache Struts 框架。这个框架提供一个小的运行时组件及一套 JSP tag 库，来实现用户界面，并使得协调控制框架更加容易。MVC 设计模式是一个被广泛使用和阐释的模式。即使在 Web 为中心的应用的上下文中，也有数不清的方法来实现基本的模式。也许，在基于 Java 的 Web 应用中，Struts 设计和实现是这个模式第一个被广泛接受的实现。

在表现层中实现一个严格 MVC 范型的想法听起来是合理的，但进一步靠近观察系统级用例后，发现没有清楚的原因表明这样的控制是必需的。在绝大多数用例场景中，所需功能只是比基本的实体 CRUD 操作多一些。MVC 范型适合统筹和协调复杂的业务操作，部分简单应用不需要。

这个系统的架构师决定不使用一个明显的 MVC 范型，如 Struts 所提供的，而是依赖于

作为 Web 页面的一部分提供的内在协同和控制。对一些架构师来说，决定不采用一个 MVC 和/或 Struts 控制框架是令人惊讶的，因为很多人认为 MVC 对所有以 Web 为中心的应用都很关键。但是，对于成功发布应用没有即刻需要的，或者系统已有文档或将来计划没有明显需要的中间件、组件甚至战略，我们认为不值得冒险去使用[1]。

做出这个决策之后，所有控制构造要么放在业务层的 EJB 会话对象，要么放在服务器层的 bean，要么作为 tag 嵌进 Web 页面。

12.3.5 Web 页面和用户界面

实际上，Web 页面的设计分离成两个主要关注点：页面本身——它们的超链接和表单字段，以及它们的布局。对于第一个关注点，系统架构师和信息架构师协作决定确切需要什么样的概念页面，页面里面应该有什么，如何导航以完成系统的业务目标。Web 页面通常作为 JSP 页面实现，混合了表现数据和到业务处理 bean 的关系。但是，第二个关注点，全部与美学和用户理解相关。Web 页面布局定义信息出现在页面何处，而不是什么信息应该在那里。它聚焦于页面的组织，这样，最终用户才可以最有效地理解和操作页面所展示的东西。本章剩下的部分将关注第一个关注点，因为第二个关注点更多是关于艺术和可视创造力，而不是传统的分析和设计。

网页设计以 UX 模型开始。UX 模型通常是描述 Web 页面名称、内容和链接的一个非常好的起始候选。一般说来，UX<<screen>>类映射到单个 JSP 页面。当页面里没有动态内容时，纯 HTML Web 页面可以是 Web 应用的一部分；但是，即使在这些情况下，也是使用 JSP 页面更好，因为一些客户端状态管理解决方案需要动态重写所有 URL 超链接，这只能在动态页面里做（参见本章前面部分的补充材料“客户状态管理”）。

依赖于所使用的 Web 工具，UX 模型通常可以容易地转换成站点布局或设计模型（见图 12-25）。<<screen>>元素映射到 JSP 页面，关联映射到导航链接，可以是简单超链接或用户提供附加数据的表单。

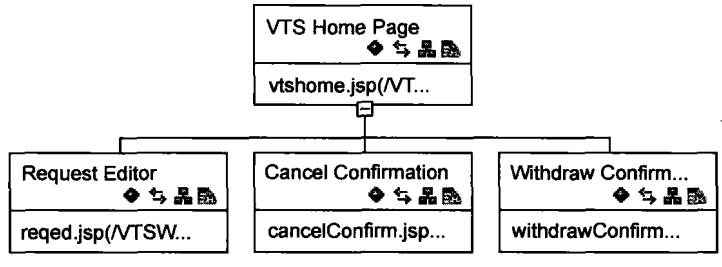


图 12-25 设计工具表达站点布局

在设计表现层时，有两个主要的考虑：如何填充页面的动态内容，以及在页面迁移时如何调用业务逻辑处理。这个应用的架构是 J2EE，架构师已经指定使用 JSF 来帮助集成带有这

[1] 同样的原因，一些人可能纳闷为什么决定采用 EJB 来持久化，还会争论说有更简单和高效的持久化策略。就像很多实际开发项目中所做的决策一样，作为决策的基础，经常要把技术上的长处以及组织的历史和经验作同等考虑。

两个任务的页面构造。JSF 帮助简化动态 Web 页面的构造。它定义了很多定制的 tag，这些 tag 嵌进 JSP 页面中的 HTML，用于调用业务对象上的方法和提取数据以放进 HTML 元素中，最终呈现在客户屏幕上。除了 JSF tag 之外，Java 服务器页面标准模板库（JSTL）也提供很多有用的 tag 来处理 JSP 页面中的数据。

1. 填充动态内容

用动态内容填充一个 Web 页面，需要一个表现层中的 Java 会话 bean 和业务层中的会话 bean 之间的连接。这应该被定义为一个远程引用，以便在不同节点上潜在分离各层。为了填充 VTS 首页，除了其他信息之外，还需要请求的当前雇员概要。这个概要来自一个 Employee CMP bean 和它相关的 Request CMP bean。既然对 CMP bean 的访问由生成的外观对象管理，表现层中的会话 bean 需要一个到外观对象的远程引用。外观将返回 SDO 对象给表现层会话 bean，表现层会话 bean 依次处理它们，以产生一个可直接在 JSP 中使用的简单 Java 对象。

在图 12-26 中，EmployeeSummarySession 类是一个在表现层执行的会话 bean。它的首要工作是连接到 EmployeeFacade，并创建 Employee 对象的请求概要。这个会话 bean 也收集和提供即将用于 VTS 首页的其他雇员特定信息。EmployeeSummary 类和它的公开内嵌类 RequestSummary 是 POJO，通过 JSTL 和 JSF tag 直接用于 JSP 页面。VTS 首页的部分 JSP 源代码如代码清单 12-3 所示。

代码清单 12-3 VTS 首页 JSP 源代码

```
<HTML>
<HEAD>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f"%>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h"%>
<%@ page language="java" contentType="text/html;
charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
<jsp:useBean id="vtsHome" class="vtsweb.actions.VtsHomeAction"/>
...
<TITLE>VTS Home Page</TITLE>
</HEAD>
<f:view>
  <BODY>
  ...
  <P>Request Summary</P>

  <c:forEach items="${vtsHome.employeeSummary}">
    <c:url var="delUrl" value="faces/reqed">
      <c:param name="id" value="${leaveRequest.id}" />
      <c:param name="action" value="delete" />
    </c:url>
    <c:url var="edUrl" value="faces/reqed">
      <c:param name="id" value="${leaveRequest.id}" />
      <c:param name="action" value="edit" />
```

```
      </c:url>
      <tr>
        <td>${leaveRequest.date}</td>
        <td>${leaveRequest.type}</td>
        <td>${leaveRequest.status}</td>
        <td><a href="${delUrl}">delete</a></td>
        <td><a href="${edUrl}">edit</a></td>
      </tr>

    </c:forEach>

    <P><BR>Outstanding Grants</P>
    ...
    </BODY>
  </f:view>
  </HTML>
```

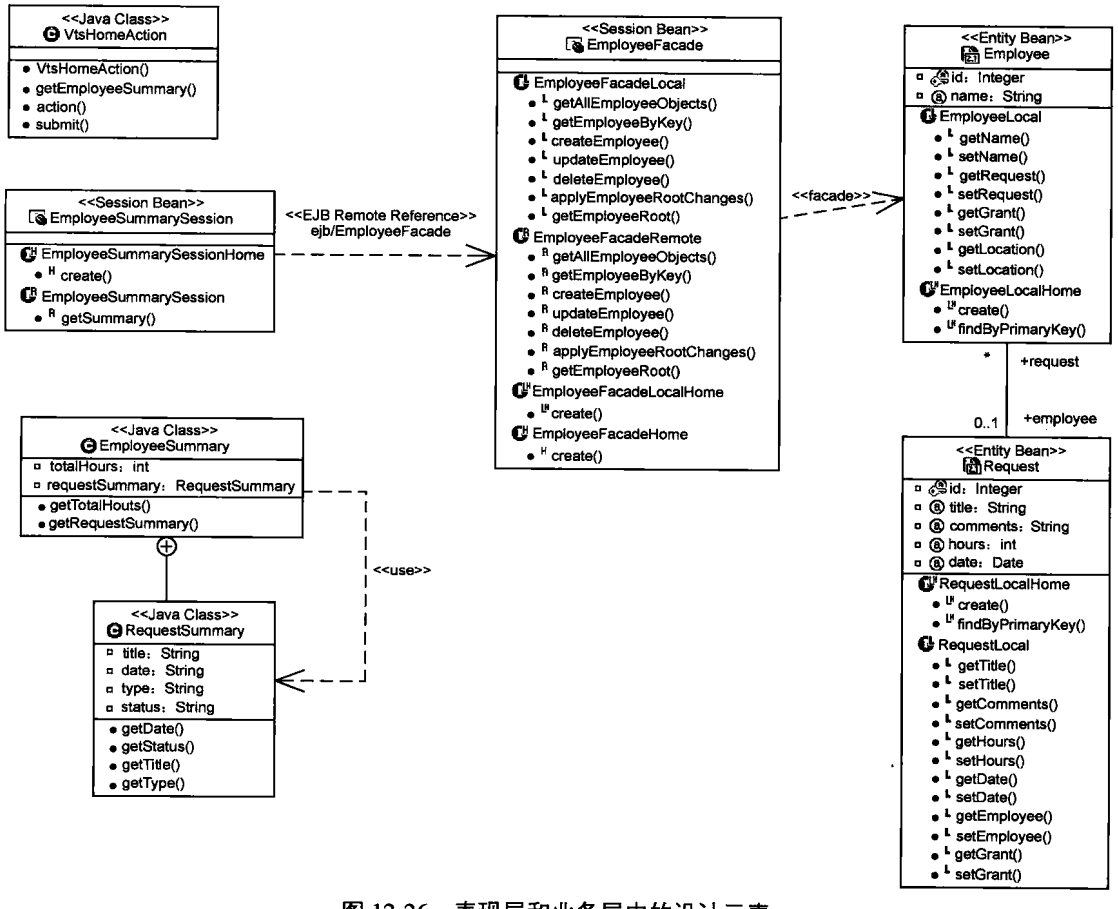


图 12-26 表现层和业务层中的设计元素

像我们的应用中所有的 JSP 页面一样，VTS 首页使用若干不同 tag 库中的很多定制 tag。三个最常用的 tag 集合是：核心 JSF tag、JSF HTML tag 和 JSTL tag。在 JSP 源代码的开始处，包含了导入的、相关的前缀标识符。

JSP tag 引用表现层 bean，表现层 bean 用于所有业务状态访问，并作为页面中动作的本地控制器。该 bean 实现为一个 POJO，和本地引用 vtsHome 一起，在 JSP 上下文中被引用。

```
<jsp:useBean id="vtsHome" class=
   "vtsweb.actions.VtsHomeAction"/>
```

这个 bean 可以用在 JSF 和 JSTL 定制 tag 中，提取业务数据，调用业务方法。在我们的首页例子中，JSTL forEach tag 遍历 employeeSummary bean 集合，该集合包含 RequestSummary 对象实例，每个代表一个特定休假时间请求。

一般来说，用一个简单的 bean 对象和一个特定页面配对来隔离 JSP 代码和来自真实 EJB 的逻辑，是一个好的实践。这个对象负责组织和提供可以放进 JSP 的所有离散数据值。这样，有创造力的团队成员负责页面的最后布局更加容易，他们不必处理有时令人困惑的管理 EJB 的技术需求。

2. 调用业务逻辑

Web 应用的工作主要在页面迁移期间完成。逻辑由用户查看或导航到下一页面的请求触发。下一页面可以是，也可以不是用户所请求的真实页面。依赖于业务逻辑的结果，基于当前客户的状态，所要的页面可以切换到不同的页面。例如，如果一个用户提交一个休假时间请求，但忘记填写所需字段，系统不会如用户期望的那样返回到首页，而是返回到编辑页面，最有可能是显示一条告警消息，指示需要的信息。

在一个 JSF 应用中，导航路径由一个名为 faces-config.xml 的配置文件定义。另外，这个文件还定义可以在 JSP 页面（如 VtsHomeAction）中引用的受托管 bean。基于结果值，导航规则定义从一个页面到另一个页面允许的迁移。这个结果值在其中一个受托管 bean 中计算，在被返回时，faces 框架使用它来决定加载和构造哪一个页面作为响应。代码清单 12-4 展示了 faces-config.xml 中的导航规则。

代码清单 12-4 faces-config.xml 中的导航规则

```
<navigation-rule>
  <from-view-id>/reqed.jsp</from-view-id>
  <navigation-case>
    <from-outcome>failure</from-outcome>
    <to-view-id>/reqed.jsp</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>success</from-outcome>
    <to-view-id>/vtshome.jsp</to-view-id>
  </navigation-case>
```

```
</navigation-rule>
```

代码清单 12-4 中的规则说明，当 reqed.jsp 上的按钮或超链接组件被激活时，如果按钮或超链接组件 tag 引用的结果是 success，应用将从 reqed.jsp 页面导航到 vtshome.jsp 页面；否则，应用返回到 reqed.jsp。

超链接或表单按钮被放进起源页面，使用 JSF HTML tag。在下面的命令按钮例子中，单击按钮会导致 vtsHomeAction 实例的 submit()方法被调用，因为 action 属性引用 VtsHomeAction 后台 bean 的 submit 方法。submit()方法执行一些处理，返回一个逻辑结果，该结果被传送到缺省的导航处理器，导航处理器根据一个在配置文件里定义的导航规则集合匹配结果。

```
<h:commandButton value="Submit Changes"
action="#{vtsHome.submit}"/>
```

这个设计的影响是，所有表现层逻辑在托管或后台 bean 中被表达，这些 bean 在 Web 服务器中被执行。通过创建正常的到业务层中 EJB 的远程 EJB 引用来完成它们的任务，业务层可以潜在地运行在系统不同节点上。

12.4 移交和后移交

Web 为中心的架构，特别是基于 Internet 的，除了常见功能和通用性能之外，还有它们自己特别的实现和测试关注点。本质上，Web 应用存在于一个异构环境中，环境很容易遭受改变，无法假设客户的硬件和软件配置。为了在设计期间帮助解决这些问题，在我们的 VTS 应用中，没有故意使用特别的浏览器特定技术。

实现一个 Web 应用，绝大部分工作是服务器端软件的开发，几乎所有写成典型 Web 应用的软件都在服务器端执行。只有用了定制的 JavaScript，或者开发特化的 applet 或客户端控件，开发人员才真正需要关心应用可能面临的每一个客户硬件和软件配置的细节。

即使选择的技术得到最流行的浏览器的最新版本的官方支持，也不能保证它们在跨越浏览器和客户配置时表现一样。这样的技术和相关问题如下。

- Java scripting　并不是所有浏览器都以完全一样的方式实现客户端脚本。一些浏览器用新的专有特性扩展了脚本语言，其他一些阐释规格稍微不同，或者有不受官方规格管制的不寻常的副作用。
- 样式表　样式表和其他布局特定的功能依赖于可得的字体和屏幕尺寸。仅因为使用样式表，并不意味着你的页面会以同样的方式呈现在所有客户配置上。
- 框　自从引入以来，框的实现就是一个问题。滚动、改变尺寸和定位问题经常折磨应用中框的使用。如果一个应用使用了框，一定要完全测试它们，不只是针对所有

目标客户配置，也针对涉及被测试应用的使用场景。

- 带宽　我们的开发工具变得更加精密，在 Web 页面中结合进精致的特性变得更加容易，由于大量的 HTML 和/或大量客户必须执行的脚本，一些最终用户就会经历极其慢的页面迁移。如果正在开发的应用要在一个 Internet 上匿名的用户基础上部署，必须特别小心地为低带宽和弱客户配置测试和设计页面。

因为技术随着时间进化，所以这个问题集也一直在改变。

面向对象编程语言

面向对象技术的使用不只限于特定语言，它的适用范围波及一个宽广的基于对象和面向对象的编程语言光谱。虽然分析和设计很重要，但是不能忽略编码的细节，因为我们的软件架构最终必须表达成一些编程语言。实际上，正如 Wulf 建议的，一门编程语言服务于三个目标[1]：

- 它是一个设计工具；
- 它是一个人类消费的工具；
- 它是一个指示计算机的工具。

本附录为可能不熟悉本书提及的某一门面向对象编程语言的读者准备。在这里，我们提供很多更重要的和流行的语言的概要描述，还有共同的例子，以此为基础比较三门最流行的面向对象编程语言（Smalltalk、C++和 Java）的语法、语义和惯用法。

A.1 语言进化

当前，有超过 2500 种不同的高级编程语言[2]。之所以能看到如此多不同的语言，是因为每一门都由它认知的问题域的特定需求形成。此外，每一门新的语言的存在使得开发人员可以解决越来越复杂的问题。每探索一个应用，语言设计人员就学习到新的经验教训，这些课程改变他们关于什么是重要语言什么不是的基本假设。语言的进化也严重受计算理论进步的影响，导致对语句、模块、抽象数据类型和过程的语义形式上的理解。

正如第 2 章中讨论的，不同编程语言支持不同的抽象：数学、算法、数据或面向对象。编程语言的最新进展是受到对象模型影响的结果。正如第 2 章中讨论过的，一门语言如果直接支持数据抽象和类，就被认为是基于对象的。一门面向对象语言是基于对象再加上提供继承和多态支持。

每一门当代基于对象和面向对象编程语言的共同祖先是 Simula，Simula 由 Dahl、Myrhurg

和 Nygard[3]在 20 世纪 60 年代开发。Simula 建造于 ALGOL 的思想基础之上，但添加了封装和继承的概念。也许更重要的是，作为一门描述系统和开发仿真的语言 Simula 引入了编写映射问题域词汇程序的科目。

图 A-1 来源于 Éric Lévénez 关于计算机语言历史的 Web 站点[4]，它展示了大多数有影响力和被广泛使用的基于对象和面向对象编程语言的谱系。盒子的长度表明通用领域中该语言的开发或显著使用活动，箭头表明它们的开发中所受的主要影响。

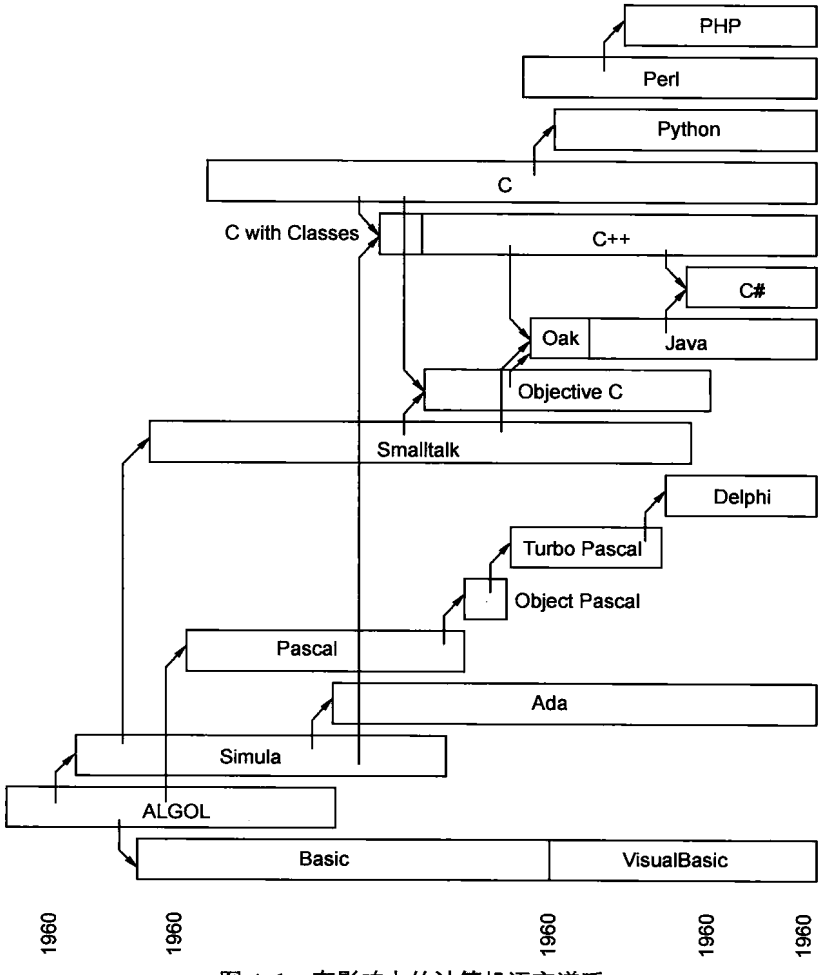


图 A-1 有影响力的计算机语言谱系

决定任何给定编程语言（甚至是语言家族）使用的流行度或是一项让人畏缩的任务，可能会挑起语言粉丝的激情，还会出现许多高级和业余统计学家。虽然单单流行度本身不意味着编程语言的质量，但确实暗示普遍使用和广泛传播的程度。

TIOBE 编程社群索引（TIOBE Programming Community Index，TCP-Index）[5]是决定给

定编程语言流行度的新颖方法，这种方法通过检查 Internet Web 页面和新闻组搜索来得出结论。通过为每个目标语言使用简单的查询，得到代表兴趣的相对页面百分比，把它们列表显示并和之前几个月及几年的检查结果比较。得到的结果是给定语言的当前流行度有趣的指示器。本书写作之时的最新结果如表 A-1 所示。正如你看到的，头 10 名语言基本上都是面向对象语言，过程语言中只有 C 保持在前列。

在以下每个部分，我们都将简短地讨论一门表现突出的面向对象语言。代码的例子基于一个简单的窗体形状问题。在这个问题中，定义类来表达可以在显示器上绘制的形状的类型，图 A-2 展示了所涉及类的通用 UML 模型。

表 A-1　　基于 Internet 决定当前流行的编程语言

名 次	编 程 语 言	评 分
1	C	19.37%
2	Java	18.57%
3	Perl	10.37%
4	C++	9.72%
5	PHP	7.97%
6	(Visual) Basic	6.78%
7	Delphi/Kylix	2.89%
8	Python	2.80%
9	C#	2.78%
10	SQL	2.65%
11	JavaScript	1.39%
12	COBOL	1.38%
13	IDL	1.23%
14	SAS	1.09%
15	Lisp	0.86%
16	Fortran	0.82%
17	MATLAB	0.78%
18	Ada	0.68%
19	Pascal	0.49%
20	AWK	0.48%

以下每一个例子试图展示如何在语言中表达关键面向对象特性，如类、继承和多态。我们不试图为给定平台优化所述任务。

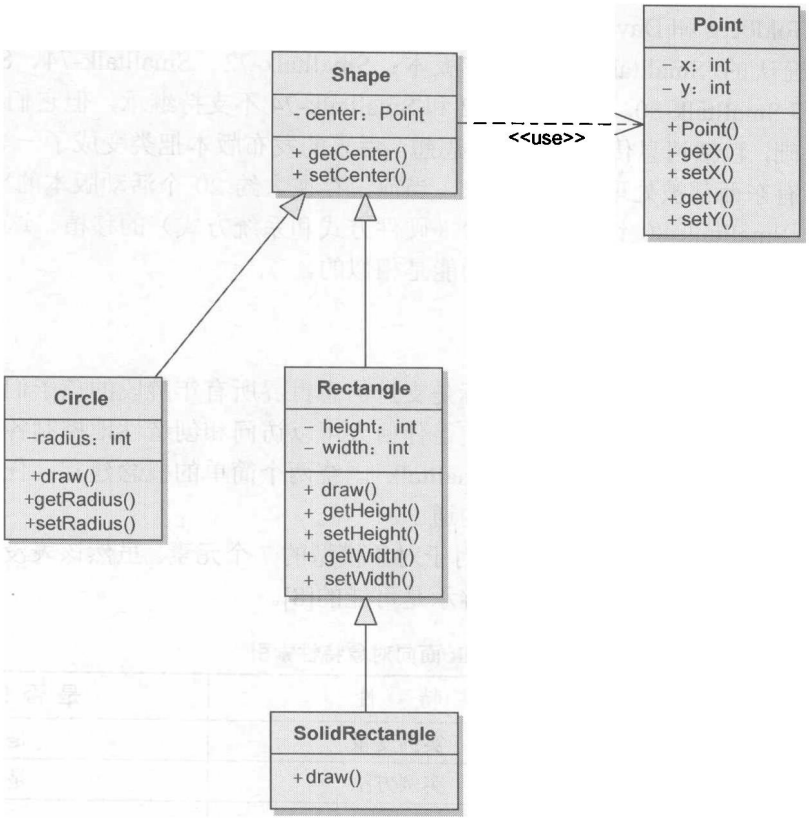


图 A-2 每个例子所使用的通用 UML 类图

A.2 Smalltalk

Smalltalk 由 Xerox Palo Alto 研究中心学习研究小组成员作为 Dynabook 的软件元素而创建，Dynabook 是 Alan Kay 提出的一个构想性项目。虽然 Smalltalk 也从 FLEX 语言和 Seymore Papert 与 Wallace Feurzeig 的工作中吸取了某些想法，但 Simula 对它影响最大。Smalltalk 代表一门语言，也代表一种软件开发环境。它是一门纯面向对象编程语言，其中所有东西都被看作是对象，甚至整数也是类。在 Simula 之后，Smalltalk 也许是最重要的面向对象编程语言，因为它的概念不仅影响到几乎每一门后来的面向对象编程语言的设计（即使 Smalltalk 自身不再流行），还影响到图形用户界面的视感，如 Macintosh 用户界面、Windows、Motif、KDE 和 Gnome，所有这些现在已经获得广泛认可。

Smalltalk 的演化经历了十多年的努力，是团队协作的结果。在大多数 Smalltalk 开发时间内，Dan Ingalls 是主架构师，还有 Peter Deutsch、Glenn Krasner 和 Kim McCall 的协作和支持，Smalltalk 环境的元素由 James Althoff、Robert Flegal、Ted Kaehler、Diana Merry 和 Steve Putz

开发，Adele Goldberg 和 David Robson 担任 Smalltalk 项目的记录人。

有 5 个确认的 Smalltalk 早期发布版本：Smalltalk-72、Smalltalk-74、Smalltalk-76、Smalltalk-78 和 Smalltalk-80。Smalltalk-72 和 Smalltalk-74 不支持继承，但它们确实支持大多数语言概念基础，包括消息传送和多态的思想。后来的发布版本把类变成了一等公民，最终，在环境里，所有东西都被处理成一个对象。当前，存在大约 20 个活动版本的 Smalltalk[6]，其中大多数是 Smalltalk-80 针对特定平台（硬件方式和系统方式）的移植。这些版本中，除了用户界面通常差别很大，类库和总体功能是相似的。

A.2.1 概述

Ingalls 宣称："Smalltalk 项目的目标是支持信息世界所有年龄段的孩子们。它的挑战在于识别和驾驭足够简单而有力的隐喻，使得任何人可以访问和创造性地控制各种信息，从数字到文本，从声音到图像。"[7]为此，Smalltalk 围绕两个简单的概念建立：任何事物都被处理成一个对象，对象之间通过传递消息沟通。

表 A-2 概括了 Smalltalk 的特性，相对于对象模型的 7 个元素。虽然该表没有直接说明多继承，但通过重定义某些原生方法，多继承是可能的[8]。

表 A-2　　Smalltalk 面向对象特性索引

对象模型元素	特　性	是否包含
抽象	实例变量	是
	实例方法	是
	类变量	是
	类方法	是
封装	变量的	私有
	方法的	公开
模块化	模块的种类	无
层次	继承	单
	泛型（？）单元	否
	元类	是
类型化	强类型化	否
	多态	是（单）
并发	多任务	间接（通过类）
持久化	持久对象	否

A.2.2 例子

考虑这样一个问题：有一个不同结构形状的列表，其中的每一个特定形状对象可能是圆、矩形或实心矩形。

Smalltalk 有一个巨大的类库，它已经包含圆和矩形类，因此在这个语言中，我们的解决方案几乎是微不足道的，这展示了复用的重要性。但是，为了比较，让我们假设只有画线和弧的原生类。

定义类 AShape，代码如下。

```
Object subclass: #AShape
     instanceVariableNames: 'theCenter'
     classVariableNames: ''
     poolDictionaries: ''
     category: 'Appendix'

initialize
     "Initialize the shape"

     thisCenter := Point new
setCenter: aPoint
     "Set the center of the shape"
     theCenter := aPoint
center
     "Return the center of the shape"
     ^theCenter
draw
     "Draw the shape"
     self subclassResponsibility
```

接下来，定义子类 ACircle，代码如下。

```
AShape subclass #ACircle
     instanceVariableNames: 'theRadius'
     classVariableNames: ''
     poolDictionaries: ''
     category: 'Appendix'

setRadius: anInteger
     "Set the radius of the circle"
     theRadius := anInteger radius
     "Return the radius of the circle"
     ^theRadius
draw
     "Draw the circle"
     | anArc index |
     anArc := Arc new.
     Index := 1.
      [index <= 4]
            whileTrue:
                  [anArc
                        center: theCenter
```

```
                    radius: theRadius
                    quadrant : index.
                    anArc display.
                    Index := index + 1]
```

接下来，子类ARectangle 可以定义如下。

```
AShape subclass: #ARectangle
     instanceVariableNames: 'theHeight theWidth'
     classVariableNames: ''
     poolDictionaries: ''
     category: 'Appendix'
draw
     "Draw the rectangle"
     | aLine upperLeftCorner |
     aLine := Line new.
     upperLeftCorner := theCenter x - (theWidth / 2) @
(theCenter y - (theHeight / 2)).
     aLine beginPoint: upperLeftCorner
     aLine endPoint: upperLeftCorner x + theWidth @
upperLeftCorner y.
     aLine display.
     aLine beginPoint: aLine endpoint.
     aLine endpoint: upperLeftCorner x + theWidth @
(upperLeftCorner y + theHeight).
     aLine display.
     aLine beginPoint: aLine endPoint.
     aLine endPoint: upperLeftCorner x @
(upperLeftCorner y + theHeight).
     aLine display.
     aLine beginPoint: aLine endPoint.
     aLine endPoint: upperLeftCorner.
     aLine display.

setHeight: anInteger
     "Set the height of the rectangle"

     theHeight := anInteger

setWidth: anInteger
     "Set the width of the rectangle"

     theWidth := anInteger

height
     "Return the height of the rectangle"

     ^theHeight
```

```
width
     "Return the width of the rectangle"

     ^theWidth
```

最后，我们也可以定义子类 ASolidRectangle。

```
ARectangle subclass #ASolidRectangle
     instanceVariableNames: ''
     classVariableNames: ''
     poolDictionaries: ''
     category: 'Appendix'

draw
     "Draw the rectangle"
     | upperLeftCorner lowerRightCorner |
     super draw.
     upperLeftCorner := theCenter x - (theWidth quo: 2) + 1 @
(theCenter y - (theHeight quo: 2) + 1).
     lowerRightCorner := upperLeftCorner x + theWidth - 1 @
(upperLeftCorner y + theHeight - 1).
     Display
          fill: (upperLeftCorner corner: lowerRightCorner)
          mask: Form gray
```

A.2.3 参考文献

最新的参考文献是 ANSI Smalltalk 标准：*Programming Language Smalltalk* [20]。其他重要的参考文献包括 Goldberg 与 Robson 的 *Smalltalk-80: The Language* [9], Goldberg 的 *Smalltalk-80: The Interactive Program ming Environment* [10]，以及 Krasner 的 *Smalltalk-80: Bits of History, Words of Advice* [11]。LaLonde 与 Pugh [12]对 Smalltalk-80 作了深入探索，包括类库和应用开发。

A.3 C++

C++由 AT&T 贝尔实验室的 Bjarne Stroustrup 设计。C++的前身是一门叫作“带类的 C”的语言，也是由 Stroustrup 在 1980 年开发的。带类的 C 受 C 和 Simula 语言的影响很大。C++很大程度上是一个 C 的超集。但是，在某种意义上，C++是更好的 C，因为它提供类型检查、重载函数和很多其他改进。但是最重要的是，C++给 C 添加了面向对象编程特性。

早期的 C++翻译器技术包括一个 C 预处理器的使用，叫做 cfront。因为这个翻译器释放出中间表达形式的 C 代码，使得十分快捷地把 C++虚拟移值到每一个 UNIX 架构成为可能。现在，几乎对于每一种指令集架构，商业上都有 C++翻译器和本地编译器。

C++语言有几个主要发布版本。版本 1.0 和它的精简发布版本给 C 添加了基本的面向对象编程语言特性，如单继承和多态，以及类型检查和重载。版本 2.0 发布于 1989 年，它以较大的用户社群的广泛经验为基础，比之前的版本在很多地方有了改善（例如，引进多继承）。版本 3.0 发布于 1990 年，它引进了模板（参数化类）和异常处理。ANSI X3J16 C++委员会已经采纳了命名空间控制（和我们的类目表示法一致）和运行时类型识别的建议。除了 ANSI 之外，C++还被 BSI（英国标准学会）、DIN（德国国家标准组织）和 ISO（国际标准组织）及其他组织标准化。ISO 标准在 1997 年最终完成和被采纳，在 1998 年 8 月被认可。ISO 是 C++标准目前的维护者。

A.3.1 概述

Stroustrup 指出："C++的设计初衷是让作者和他的朋友不必用汇编、C 或各种现代高级语言编程。它的主要目标是使程序员更容易和更愉快地编写好的程序。从来没有 C++纸面设计，设计、写文档和实现是同时进行的。"[13]C++纠正了 C 中的很多缺陷，给语言添加了对类、类型检查、重载、自由存储管理、常量类型、引用、内联函数、派生类和虚函数的支持[14]。

相对于对象模型的 7 个元素，我们在表 A-3 中概括了 C++的特性。

表 A-3　　C++面向对象特性索引

对象模型元素	特　性	是否包含
抽象	实例变量	是
	实例方法	是
	类变量	是
	类方法	是
封装	变量的	公开、保护、私有
	方法的	公开、保护、私有
模块化	模块种类	文件
层次	继承	多
	泛型（？）单元	是
	元类	否
类型化	强类型化	是
	多态	是（单）
并发	多任务	间接（通过类）
持久化	持久对象	否

A.3.2 例子

让我们再次实现形状问题。C++的通用风格是把每个类的外部视图都放进头文件中。因

此，我们可以编写以下代码。

```
Struct Point {
     int x;
     int y;
};

class Shape {
public:
     Shape();
     void setCenter(Point p);
     virtual void draw() = 0;
     Point center() const;
private:
     Point theCenter;
};

class Circle : public Shape {
public:
     Circle();
     void setRadius(int r);
     virtual void draw();
     int radius() const;
private:
     int theRadius;
};

class Rectangle : public Shape {
public:
     Rectangle();
     void setHeight(int h);
     void setWidth(int w);
     virtual void Draw();
     int height() const;
     int width() const;
private:
     int theHeight;
     int theWidth;
};

class SolidRectangle : public Rectangle {
public:
     virtual void draw();
};
```

C++的定义不包括类库。为了达到我们的目标，我们假设存在通用图形显示（例如，X Windows 或 Microsoft Windows）的编程接口，以及可以链接到全局对象 Display、Window 和 GraphicsContext。然后，在一个分离的文件中完成以上方法，代码如下。

```
Shape::Shape() {
      theCenter.x = 0;
      theCenter.y = 0;
};

void Shape::setCenter(Point p) {
     theCenter = p;
}

Point Shape::center() const {
      Return theCenter;
}

Circle::Circle() : theRadius(0) { }

void Circle::setRadius(int r) {
      theRadius = r;
}

void Circle::draw() {
      int X = (center().x - theRadius);
      int Y = (center().y - theRadius);
      XDrawArc(Display, Window, GraphicsContext,
X, Y, (theRadius * 2),
(theRadius * 2), 0, (360 * 64));
};

int Circle::radius() const {
      return theRadius;
}

Rectangle::Rectangle() : theHeight(0), theWidth(0) { }

void Rectangle::setHeight(int h) {
      theHeight = h;
}

void Rectangle::setWidth(int w) {
      theWidth = w;
}

void Rectangle::draw() {
      int X = (center().x - (theWidth / 2));
      int Y = (center().y - (theHeight / 2));
      XDrawRectangle(Display, Window, GraphicsContext,
X, Y, theWidth, theHeight);
};
```

```
int Rectangle::height() const {
      return theHeight;
};

int Rectangle::width() const {
      return theWidth;
};

void SolidRectangle::draw() {
      Rectangle::draw();
      int X = (center().x - (width() / 2));
      int Y = (center().y - (height() / 2));
      gc oldGraphicsContext - GraphicsContext;
      XSetForeground(Display, GraphicsContext, Gray);
      XDrawFilled (Display, Window, Graphics, X, Y,
width(), height());
      GraphicsContext = oldGraphicsContext;

};
```

A.3.3 参考文献

最流行的C++参考文献一直是Ellis和Stroustrup的*The Annotated C++ Refer ence Manual* [15]，通常被称作“ARM”。最近的参考文献是Stroustrup的*The C++ Standard: Incorporating Technical Corrigendum No. 1* [16]。Stroustrup在其他著作中深入剖析了这门语言以及它在面向对象设计上下文中的使用 [17, 18]。

A.4 Java

在Sun Microsystem一个不为人注意的小组里，James Gosling和其他一些人觉得C++对于他们所要做的事情来说太困难了，所以创建了一门小的编程语言，叫做Oak。

Java的创始人本来是在建造数字控制消费者设备（如娱乐平台和微波炉）的高级软件。在把这个技术带入数字有线电视市场的努力失败后，他们认识到，新近崛起的商业Internet和他们所拥有的技术是个绝配。他们把该语言重新命名为 Java，开始把它作为一门通用目标的编程语言、一门容易在整个 Internet 普及并在 HTML 浏览器上下文中执行的语言来推销它。

Java的巨大突破出现在1995年5月，Sun Microsystems和Netscape宣布Java技术将被整合进当时Internet上最常用的浏览器Netscape Navigator，这高效地创建了一个这个技术潜在用户的巨大市场。

虽然早期的市场工作聚焦于Java的Internet适应性和平台独立性，但真正让Java变成受

人喜爱的面向对象语言的，是它的服务器端使用。Java 的图形性能总是受到批评，今天，我们可以发现，大多数 Java 开发人员是在服务器端（即非基于图形组件）使用 Java，以移交 Web 页面和业务层功能。

即使 Java 面世仅仅 10 多年，但不管从哪个角度来衡量，都可以看到它的惊人成长。成长不仅来自语言的进化，而且更显著的是，它的相关技术，如企业 Java Bean（EJB）、Java 服务器页面（JSP）和 Java 2 Micro Edition（J2ME）。Java 技术和标准继续快速而有组织地成长。在 1998 年，Sun 创建了 Java Community Process（JCP），来管理 Java 技术规格、参考实现和测试套件的开发和修订。这个过程概述了社群对 Java 规格请求（JSR）的评估和反应，JSR 针对的是新的规格或已有规格的重要修改。当前，大约有 100 个规格的工作在进行中。

Sun Microsystems 继续管理 Java 语言及其相关技术的总体开发。

A.4.1 概述

Java 不只是一种编程语言，像 Smalltalk 一样，它更是一个开发环境和运行环境。Java 技术包括使用虚拟机和通用中间字节码。也和 Smalltalk 类似，它包括一个丰富的、可扩展的类库。因为它的移值性和丰富的标准功能集，Java 技术和 Internet 如此匹配。

Java 语言自身有意做成和 C++相似。Sun 的团队开始时使用 C++，但发现问题太多，决定创建一门更合适的新语言绕过这些问题。

我们要建造一个系统，使用它不需要很多深奥的培训就能够容易地编程，并且能促进今天的标准实践。大多数程序员这些日子里使用 C，大多数程序员使用 C++来做面向对象编程。因此，即使我们发现 C++不合适，我们也将 Java 设计得尽可能接近于 C++，这样使得系统更加可理解。[19]

Java 忽略了 C++中发现的很多特性，例如多继承和操作符重载。Java 最有用的特性是自动垃圾收集。Java 开发人员不要求执行自己的内存管理。相反，他们可以创建新的对象实例，并可以保证当所有对它们的引用都被移除时，内存会在某些点上被回收。虽然没有 C++那么高效，这个垃圾收集确实消灭了一个非常常见的编程错误来源。

Java 在所有可行的地方试图增强类型安全性。但是，因为使用它丰富的类库，特别是它的集合类，像 C++那样严格的类型检查是不切实际的。最新的规格——Java 2 的版本 1.5，确实包括一些新东西，叫作泛型（JSR 14）。Java 泛型，也被称为参数化类型，类似于 C++模板，使得 Java 开发人员可以创建类型安全的集合。

相对于对象模型的 7 个元素，我们在表 A-4 中概括了 Java 的特性。

表 A-4　　Java 面向对象特性索引

对象模型元素	特　性	是否包含
抽象	实例变量	是
	实例方法	是

续表

对象模型元素	特　性	是否包含
抽象	类变量	是
	类方法	是
封装	变量的	公开、保护、私有、包
	方法的	公开、保护、私有、包
模块化	模块种类	文件
层次	继承	单
	泛型单元	是
	元类	否
类型化	强类型化	否
	多态	是（单）
并发	多任务	是
持久化	持久对象	否

A.4.2 例子

为 Java 重新创建我们的例子，为每个已定义的类创建一个分离的类文件。首先，创建 Point 类，代码如下。

```
package test;
public class Point {

      private int x = 0;
      private int y = 0;

      public Point(int x, int y) {
            this.x = x;
            this.y = y;
      }

      public int getX() {
            return x;
      }

      public void setX(int x) {
            this.x = x;
      }

      public int getY() {
            return y;
      }
```

```
        public void setY(int y) {
            this.y = y;
    }
}
```

下一步，定义 Shape 类，代码如下。

```
package test;

public class Shape {

    private Point center;

    public Point getCenter() {
        return center;
    }

    public void setCenter(Point center) {
        this.center = center;
    }
}
```

然后是创建 Circle 类，代码如下。

```
package test;
import java.awt.Graphics;

public class Circle extends Shape {
    private int radius = 0;

    public void draw(Graphics g) {
        int x = (getCenter().getX() - radius);
        int y = (getCenter().getY() - radius);
        int d = radius * 2;
        g.drawOval(x, y, d, d);
    }

    public int getRadius() {
        return radius;
    }

    public void setRadius(int radius) {
        this.radius = radius;
    }
}
```

然后，创建 Rectangle 类代码如下。

```
package test;
```

```
import java.awt.Graphics;

public class Rectangle extends Shape {

      private int height = 0;
      private int width = 0;

      public void draw(Graphics g) {
            int x = (getCenter().getX() - width/2);
            int y = (getCenter().getY() - height/2);
            g.drawRect(x, y, width, height);
      }

      public int getHeight() {
            return height;
      }

      public void setHeight(int height) {
            this.height = height;
      }

      public int getWidth() {
            return width;
      }

      public void setWidth(int width) {
            this.width = width;
      }
}
```

最后是创建 SolidRectangle 类，代码如下。

```
package test;
import java.awt.Color;
import java.awt.Graphics;

public class SolidRectangle extends Rectangle {

      public void draw(Graphics g) {
            super.draw(g);
            int width = getWidth()-1;
            int height = getHeight()-1;
            int x = (getCenter().getX() - width/2);
            int y = (getCenter().getY() - height/2);
            g.setColor(Color.GRAY);
            g.fillRect(x, y, width, height);
      }
}
```

A.4.3 参考文献

Sun Microsystems Press（SMP）是 Sun Microsystems、Prentice Hall 和 Addison-Wesley 合作的机构。SMP 是一个关于 Java 技术所有方面和级别的书籍的优秀的、非常详细的来源。既然 Java 的成长是受 Internet 推动的，也许 Internet 才是 Java 信息最好和最新的来源。Ken Arnold、James Gosling 和 David Holmes 负责制作了 *The Java Programming Language, Third Edition*——一本 Java 语言的完整介绍，而且深入剖析了 Java 的基本设计目标。

进一步阅读

本附录通过参考“分类书目”中列出的条目，指引读者对每一章作进一步阅读。参考文献采取以下形式：[<标签> <年>]。例如，Brooks [H 1975] 指他 1975 年的书 *The Mythical Man-Month* 中书目的 H 部（软件工程）。

第 1 章

Brooks 的两部经典著作[H 1975，H 1987]清楚地描述了和开发复杂软件系统相关的挑战。Glass [H 1982]、负责采办的国防部副部长办公室（国防科学委员会特别工作组报告）[H 1987]和美国国防部[H 1982]提供了关于当前软件实践的进一步信息。关于软件故障的本质和原因的实证研究可参见 van Genuchten [H 1991]、Guindon 等[H 1987]和 Jones [H 1992]。

Simon 的两项工作[A 1962，A 1982]是关于复杂系统架构奠基参考文献的，Courtois [A 1985]把这些想法应用在软件领域。Alexander 的奠基著作 [I 1979]提供了建立物理结构架构的崭新方法。Peter [I 1986]与 Petroski [I 1985]分别研究了社会与物理系统上下文中的复杂性。类似地，Allen 与 Starr [A 1982]研究了很多领域中的分层系统。Flood 与 Carson [A 1988] 从系统科学理论的角度做了复杂性的形式研究。Waldrop [A 1992]描述新兴的复杂性科学以及研究了复杂的自适应系统、突发行为和自组织。Miller [A 1956]的报告提供了人类认知基本限制因素的实证。

软件工程主题有很多优秀的参考文献。Ross、Goodenough 与 Irvine [H 1980]和 Zelkowitz [H 1978]是两篇经典的论文，概括了软件工程的本质元素。该主题的外延工作包括 Jensen 与 Tonies [H 1979]、Sommerville [H 1989]、Vick 与 Ramamoorthy [H 1984]、Wegner [H 1980]、Pressman [H 1992]、Oman 与 Lewis [H 1990]、Berzins 与 Luqi [H 1991]和 Ng and Yeh [H 1990]。通用软件工程的其他相关论文可以参看 Yourdon [H 1979]和 Freeman 与 Wasserman [H 1983]。Graham [F 1991]和 Berard [H 1993]展示了面向对象软件工程的广泛应用。

Gleick[I 1987]提供了一份非常有可读性的混沌科学介绍。

第2章

对象模型的概念由Jones [F 1979]和Williams [F 1986]第一次引入。Kay的博士论文[F 1969]建立了之后的面向对象编程工作追随的方向。

Shaw[J 1984]提供了对关于高级编程语言中抽象机制的优秀概述。抽象的理论基础可以参见Liskov与Guttag [H 1986]、Guttag [J 1980]和Hilfinger [J 1982]。Parnas [F 1979]是信息隐藏的奠基著作。由Pattee [J 1973]编辑的工作讨论了层次的意义和重要性。

面向对象应用的案例研究可以参见Taylor [H 1990，C 1992]、Berard [H 1993]、Love [C 1993]和Pinson与Weiner [C 1990]。

涉及所有面向对象技术主题的优秀论文集可以参见Peterson [G 1987]、Schriver与Wegner [G 1987]和Khoshafian与Abnous [E 1990]。若干年度的面向对象技术会议的会议记录也是优秀的材料来源。

负责建立对象技术标准的组织包括对象管理组织和ANSI X3J7 委员会。

C++的首要参考文献是Ellis与Stroustrup[G 1990]，其他有用的参考文献包括Stroustrup [G 2000]、Lippman、LaJoie与Moo [G 2005]和Coplien [G 1992]。

关于NET框架基础的一个好的介绍是Conrad等人的 *Introducing .NET* [G 2001]。

第3章

MacLennan [G 1987] 讨论了值和对象之间的区别。Meyer [J 1987]的工作提出了按契约编程的思想。

很多工作和类层次的主题相关，特别强调继承和多态方法。Albano[G 1983]、Allen与Starr [A 1982]、Brachman [J 1983]、Hailpern与Nguyen[G 1987]和Wegner与Zdonik [J 1988]的工作为所有重要的概念和问题提供了一个优秀理论基础。Cook与Palsberg [J 1989]和Touretzky[G 1986]提供了继承语义的形式处理。Wirth [J 1987]提出记录类型扩展的相关方法，并将其使用在Oberon语言中。Ingalls [G 1986] 提供了关于多个多态主题的有用讨论。Grogono [G 1989]研究了多态和类型检查的相互作用，Ponder与Buch [G 1992]警告了不受约束的多态的危险。Meyer[G 1988]和Halbert与O'Brien [G 1988]提供了高效使用继承的实践指南。LaLonde和Pugh [J 1985]研究了如何教人高效使用特化和泛化的问题。

Rubin与Goldberg [B 1992]和Wirfs-Brock、Wilkerson与Wiener [F 1990]进一步详细探讨了一个抽象的角色和责任的本质。Coad [F 1991]考虑了类设计良好程度的测量。

Meyer [G 1986]研究了在Eiffel语言中体现时的泛型和继承之间的关系。Stroustrup [G 1988a] 在C++中提出了参数化类型机制。

使用exemplar，基于类的层次的一个备选方案是委托。Stein [G 1987]详细研究了这个方法。

第4章

分类的问题是永恒的问题。柏拉图在《政治家》中介绍了经典的分类方法，有相似属性

的对象被分组。在《范畴》中，亚里士多德挑选这个主题分析类和对象之间的不同。若干世纪后，阿奎那在 *Summa Theologica* 中，笛卡尔在 *Rules for the Direction of the Mind* 中，思考了分类的哲学。当代客观主义（objectivist）哲学家包括 Rand [I 1979]。

Lakoff 与 Johnson [I 1980]和 Goldstein 与 Alger [C 1992]讨论了客观主义另一种关于世界的视角。

分类本质上是一种人类的技巧。Piaget 首先研究了童年早期开发获得分类能力的理论，Maier [A 1969]作了概括。Lefrancois [A 1977]提供了关于这些想法可读性非常好的介绍，并提供了一个关于孩子获得对象概念的优秀演讲。

认知科学家在很细的细节上探索了分类问题。Newell 与 Simon[A 1972]提供了一个关于人类分类技巧材料很棒的来源。更普遍的信息可以参见 Simon [A 1982]、Hofstadter [I 1979]、Siegler 与 Richards [A 1982]和 Stillings 等[A 1987]。Lakoff [A 1987]，一个语言学家，针对不同人类语言如何演化以处理分类问题以及意识所揭示的东西，提供了他的见解。Minksy [A 1986]以关于意识结构的理论开始，从相反的方向研究了这个主题。

Michalski 与 Stepp [A 1983，A 1986]、Peckham 与 Maryanski [J 1988]和 Sowa [A 1984]详细描述了概念聚类，一种通过分类来表达知识的方法。Prieto-Diaz 与 Arango [A 1991]的完整论文集描述了领域分析——一种通过检查问题域词汇表查找关键抽象和机制的方法。Iscoe [B 1988]对这个领域作出了若干重要的贡献。更多信息可以参见 Iscoe、Browne 与 Werth [B 1989]，以及 Moore 与 Bailin [B 1988]和 Arango [B 1989]。

智能分类经常需要以创新的方法来看世界，这些技巧是可以学习的（或者，至少被鼓励的）。VonOech [I 1990]阐述了一些通向创造力的路径。Coad [A 1993]开发了一个棋盘游戏（对象游戏（Object Game））用来培养识别类和对象的技巧。

在软件系统模式分类方面做的很多工作，兴起了惯用法、机制和框架的分类学。令人感兴趣的参考文献包括 Gamma 等人的著作[1995]、Coplien [G 1992]、Coad [A 1992]、Johnson [A 1992]、Shaw [A 1989，A 1990，A 1991]和 Wirfs-Brock [C 1991]。 Alexander 影响深远的著作[I 1979]把模式应用到建筑架构和城市规划领域。

数学家试图为分类设计实证方法，导致了测量理论的诞生。Stevens [A 1946]和 Coombs、Raiffa 与 Thrall [A 1954]提供了这个主题的奠基著作。

北美分类学会每年公布两次期刊，包括各种关于分类问题的论文。

第 5 章

对象管理组织已经制作了 UML 超结构（Superstructure）规格——版本 2.0，它是本书的基础，引导我们学习这个新版表示法的使用。可以在 UML 资源页面 www.uml.org[L 2004]获得。

正如之前说过的，UML 2.0 的特性非常丰富和完整。但是，在日常生活中，可能你只会使用表示法的一部分。如果你对探索 UML 不经常使用的部分感兴趣，有很多书特别聚焦于表示法的细节。建议阅读 Rumbaugh、Jacobson 与 Booch [L 2005]和 Booch、Rumbaugh 与

Jacobson [L 2005]，这是关于这个主题的权威讨论。

Blaha 与 Rumbaugh [L 2005] 用 UML 2.0 表示法更新了他们 1991 年的面向对象建模和设计的经典著作。

Bennett、Skelton 与 Lunn [L 2005]给出了针对 UML 2.0 表示法的实践指南，还给出了例子、练习和测验题。这是该书 2001 版本的一个有价值的更新，适合 UML 或 UML 2.0 的初学者。

Fowler [L 2003] 提供了 UML 2.0 典型使用的简明表达。

第 6 章

Booch [F 1982]第一个归档了本章中所描述过程的一个早期形式。之后，Berard [F 1986]细化了这个工作。相关方法包括 Seidewitz 与 Stark [F 1986a, F 1986b, F 1987, F 1988]的 GOOD（通用面向对象设计）、Kerth [F 1988]的 MOOD（多视图面向对象设计）和 CRI、CISI Ingenierie 与 Matra 为欧洲空间站[F 1987]所做的 HOOD（分层面向对象设计）。一个附加的相关工作是 Stroustrup [G 1991]，实质上提出了类似的过程。

除了之前第 2 章引用的工作之外，很多其他的方法学家提出了特定的面向对象开发过程，为文献目录提供了一个巨大的参考文献集合。一些更有趣的贡献来自 Kruchten [F 2003]、Alabios [F 1988]、Boyd[F 1987]、Buhr [F 1984]、Cherry [F 1987, F 1990]、deChampeaux 与 Balzer 等[F 1992]、deChampeaux 与 Lea 及 Faure [F 1992]、Felsinger [F 1987a, F 1987b]、Firesmith [F 1986, F 1993]、Hines 与 Unger [G 1986]、Jacobson [F 1985]、Jamsa [F 1984]、Kadie [F 1986]与 Masiero 及 Germano [F 1988]、Nielsen [F 1988]、Nies [F 1986]、Rajlich 与 Silva [F 1987]和 Shumate [F 1987]。

各种面向对象开发过程的比较可以参见 Arnold 等[F 1991]、Boehm-Davis 与 Ross [H 1984]、deChampeaux [B 1991a，B 1991b]、Cribbs 与 Moon 及 Roe [F 1992]、Fowler [F 1992]、Kelly [F 1986]、Mannino [F 1987]、Song [F 1992]和 Webster [F 1988]。Brookman [F 1991]和 Fichman 与 Kemerer [F 1992] 提供了结构化和面向对象方法的比较。

软件过程的实证研究可以参见 Curtis、Kellner 与 Over [H 1992]和软件过程工作室[H 1988]。另一个令人感兴趣的参考文献是 Guindon、Krasner 与 Curtis [H 1987]，它研究了开发过程早期开发人员所用的探索过程。Rechtin [H 1992] 为必须驱动开发过程的软件架构师提供了实践指南。

关于如何仿制一个成熟的过程，经典的参考文献是 Parnas 与 Clements [H 1986]。

第 7 章

van Genuchten [H 1991]和 Jones [H 1992]研究了常见的软件风险。Abdel-Hamid 与 Madnick [H 1991]研究了开发团队的动力学。

Gilb [H 1988]和 Charette [H 1989]是有用的软件工程管理实践参考文献。对于在开发期间真正进行的事情所做的实际研究（实战把理论赶出了窗口），可参见 Glass [H 1982]、Lammers

[H 1986]和 Humphrey [H 1989]。DeMarco 与 Lister [H 1987]、Yourdon [H 1989b]、Rettig [H 1990]和 Thomsett [H 1990]向开发经理推荐了很多东西。

Schulmeyer 与 McManus [H 1992]提供了一份关于软件质量保证的优秀通用参考文献。Chidamber 与 Kemerer [H 1991] 和 Walsh [H 1992, 1993]研究了面向对象系统上下文中的质量保证和度量。Kemerer 与 Darcy [H 2005]提供了应用 Chidamber Kemerer（CK）度量集的例子，并观察了它们的实践应用。

Kan [H 2002] 使用从面向对象项目中学到的教训和特定度量提供了一份软件质量工程和度量的全面参考文献。Lorenz 与 Kidd [H 1994]和 Henderson-Sellers [H 1996]提供了被高度认可的面向对象度量文献。

关于个人和组织如何迁移到对象模型的建议，在 Goldberg [C 1978]、Goldberg 与 Kay [G 1977]和 Kempf [G 1987]中进行了描述。

Hantos [H 2005] 通过映射来自 Boehm [H 1989, H 2002]的软件风险信息到来自 Meyer [H 1995]的面向对象概念，提供了一个面向对象开发风险的创新视角。

第 8 章

Aerospace 公司的出版物 *GPS Primer—A Student Guide to the Global Positioning System* 提供了一个针对全球定位系统的介绍[C 2003]。

欧盟委员会能源运输总署有一个全面的 Web 站点来讨论 Galileo——欧盟卫星导航系统[C 2006]。

IEEE-Std-1471-2000 提供了一个软件密集型系统的架构描述框架，以及对描述的内容的定义 [D 2000]。

在题为“*ANSI/IEEE 1471 and Systems Engineering*”的论文中，Maier、Emery 与 Hilliard [D 2004] 展示了在描述系统架构的系统工程活动期间使用这个规格的基本理由。

Kruchten [D 1995]介绍了架构的 4+1 视图模型，使用 5 个视图描述软件：逻辑视图、进程视图、物理视图、开发视图和用例视图。

在题为“*Introduction to Object-Oriented Systems Engineering*”的论文的第 1 和第 2 部分中，Krikorian[D 2003]提出了将 Kruchten 的 4+1 视图应用到系统工程，并展示了为此定义的扩展 4+1 视图。

IEEE-Std-12207 包括 3 个部分：12207.0、12207.1 和 12207.2。IEEE-Std-12207.0-1996 [H 1996] 提供了可用于国家和国际业务的软件实践基础。它澄清、修改并增加了 ISO/IEC 12207: 1995。IEEE-Std-12207.1-1997 [H 1997a] 给出了关于记录生命周期数据的指南。IEEE-Std-12207.2-1997 [H 1997b]给出了关于实现 IEEE-Std-12207.0-1996 过程的指南。

IEEE-Std-1220-2005 [H 2005]提供了一个方法，来应用和管理产品开发使用的系统工程过程的跨学科任务。

国际系统工程学会（INCOSE）制作了系统工程师所执行活动的全面指南[I 2006]。

第9章

列车交通管理系统的一些设计点，基于Murphy [C 1988]所描述的高级列车控制系统。

实际上，消息翻译和验证发生在所有命令和控制系统中。Plinta、Lee和Rissman [C 1989]是针对该问题的一篇优秀文章，提供了以一种类型安全的方式跨过分布式系统中的处理器传递消息的设计机制。

第10章

在架构模式的上下文中，Shaw [A 1991]讨论了黑板框架以及其他种类的应用框架。

Englemore与Morgan [C 1988]全面讨论了黑板系统，包括它们的进化、理论、设计和应用。除了其他主题，还有两个面向对象黑板系统的描述：BB1——来自Stanford，以及BLOB——为英国国防部开发。关于黑板系统，其他有用的信息来源可以参见Hayes-Roth [J 1985]和Nii [J 1986]。

关于基于规则的系统中前向链和后向链的详细讨论可以参见Barr与Feigenbaum [J 1981]、Brachman与Levesque [J 1985]、Hayes-Roth与Waterman及Lenat [J 1983]和Winston and Horn [G 1989]。

Meyer与Matyas [I 1982]研究了各种密码的优点和缺点，以及破解它们的算法方法。

Corkill [D 1991]和Hunt [D 2002]讨论了黑板应用框架的考虑、好处和缺点。

第11章

Hansen [H 1977]、Ben-Ari [H 1982]和Holt等[H 1978]详细讨论了进程同步、死锁、活锁和竞争条件的问题。Mellichamp [H 1983]、Glass [H 1983]和Foster [H 1981]提供了开发实时应用问题的通用参考文献。Lorin [H 1972]把并发看作硬件和软件的相互作用。

第12章

Garrett [M 2002] 讨论了高级别用户相关的设计问题。Constantine与Lockwood [M 1999]也讨论了用户相关的设计问题，并强调了用例和建模方法。

Monson-Haefel [M 2001]提供了对EJB设计和编程迷人的介绍。

Geary与Horstmann [M 2004]讨论了实现Web应用可复用组件的上下文中的用户界面设计。

Cavaness [M 2004]讨论了Struts框架和模型—视图—控制器设计模式。

注解

前言

Mills, H. 1985. *DPMA and Human Productivity*. Houston, TX: Data Processing Management Association.

第 1 部分：概念

Wagner, J. 1986. *The Search for Signs of Intelligent Life in the Universe*. New York, NY: Harper and Row, p. 202. By permission of ICM, Inc.

第 1 章：复杂性

[1] Brooks, F. April 1987. No Silver Bullet: Essence and Accidents of Software Engineering. *IEEE Computer* vol. 20(4), p. 12.

[2] Peters, L. 1981. *Software Design*. New York, NY: Yourdon Press, p. 22.

[3] Brooks. No Silver Bullet, p. 11.

[4] Parnas, D. July 1985. *Software Aspects of Strategic Defense Systems*. Victoria, Canada: University of Victoria, Report DCS-47-IR.

[5] Peter, L. 1986. *The Peter Pyramid*. New York, NY: William Morrow, p. 153.

[6] Waldrop, M. 1992. *Complexity: The Emerging Science at the Edge of Order and Chaos*. New York, NY: Simon and Schuster.

[7] Courtois, P. June 1985. On Time and Space Decomposition of Complex Structures. *Communications of the ACM* vol. 28(6), p. 596.

[8] Simon, H. 1982. *The Sciences of the Artificial*. Cambridge, MA: The MIT Press, p. 218.

[9] Rechtin, E. October 1992. The Art of Systems Architecting. *IEEE Spectrum* vol. 29(10), p. 66.

[10] Simon. *Sciences*, p. 217.

[11] Ibid., p. 221.

[12] Ibid., p. 209.

[13] Gall, J. 1986. *Systemantics: How Systems Really Work and How They Fail.* Second Edition. Ann Arbor, MI: The General Systemantics Press, p. 65.

[14] Miller, G. March 1956. The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information. *The Psychological Review* vol. 63(2), p. 86.

[15] Simon. *Sciences*, p. 81.

[16] Dijkstra, E. 1979. Programming Considered as a Human Activity. In *Classics in Software Engineering.* New York, NY: Yourdon Press, p. 5.

[17] Parnas, D. December 1985. Software Aspects of Strategic Defense Systems. *Communications of the ACM* vol. 28(12), p. 1328.

[18] Tsai, J., and Ridge, J. November 1988. Intelligent Support for Specifications Transformation. *IEEE Software* vol. 5(6), p. 34.

[19] Stein, J. March 1988. Object-Oriented Programming and Database Design. *Dr. Dobb's Journal of Software Tools for the Professional Programmer* no. 137, p. 18.

[20] Peters. *Software Design.*

[21] Yau, S., and Tsai, J. June 1986. A Survey of Software Design Techniques. *IEEE Transactions on Software Engineering* vol. SE-12(6).

[22] Teledyne Brown Engineering. October 1987. *Software Methodology Catalog*, Report MC87-COMM/ADP-0036. Tinton Falls, NJ.

[23] Sommerville, I. 1985. *Software Engineering.* Second Edition. Wokingham, England: Addison-Wesley, p. 68.

[24] Yourdon, E., and Constantine, L. 1979. *Structured Design.* Englewood Cliffs, NJ: Prentice-Hall.

[25] Myers, G. 1978. *Composite/Structured Design.* New York, NY: Van Nostrand Reinhold.

[26] Page-Jones, M. 1988. *The Practical Guide to Structured Systems Design.* Englewood Cliffs, NJ: Yourdon Press.

[27] Wirth, N. January 1983. Program Development by Stepwise Refinement. *Communications of the ACM* vol. 26(1).

[28] Wirth, N. 1986. *Algorithms and Data Structures.* Englewood Cliffs, NJ: Prentice-Hall.

[29] Dahl, O., Dijkstra, E., and Hoare, C. A. R. 1972. Structured Programming. London, England: Academic Press.

[30] Mills, H., Linger, R., and Hevner, A. 1986. *Principles of Information System Design and Analysis.* Orlando, FL: Academic Press.

[31] Jackson, M. 1975. *Principles of Program Design.* Orlando, FL: Academic Press.

[32] Jackson, M. 1983. *System Development.* Englewood Cliffs, NJ: Prentice- Hall.

[33] Orr, K. 1971. *Structured Systems Development*. New York, NY: Yourdon Press.

[34] Langdon, G. 1982. *Computer Design*. San Jose, CA: Computeach Press, p. 6.

[35] Miller. Magical Number, p. 95.

[36] Shaw, M. (ed.). 1981. *ALPHARD: Form and Content*. New York, NY: Springer-Verlag, p. 6.

[37] Goldberg, A. 1984. *Smalltalk-80: The Interactive Programming Environment*. Reading, MA: Addison-Wesley, p. 80.

[38] Petroski, H. 1985. *To Engineer Is Human*. New York, NY: St. Martin's Press, p. 40.

[39] Dijkstra, E. January 1993. *American Programmer* vol. 6(1).

[40] Mostow, J. Spring 1985. Toward Better Models of the Design Process. *AI Magazine* vol. 6(1), p. 44.

[41] Stroustrup, B. 1991. *The C+ Programming Language*. Second Edition. Reading, MA: Addison-Wesley, p. 366.

[42] Eastman, N. 1984. Software Engineering and Technology. *Technical Directions* vol. 10(1), p. 5.

[43] Brooks. No Silver Bullet, p. 10.

第 2 章：对象模型

[2] Wegner, P. June 1981. *The Ada Programming Language and Environment*. Unpublished draft.

[3] Abbott, R. August 1987. Knowledge Abstraction. *Communications of the ACM* vol. 30(8), p. 664.

[4] Ibid., p. 664.

[5] Shankar, K. 1984. Data Design: Types, Structures, and Abstractions. *Handbook of Software Engineering*. New York, NY: Van Nostrand Reinhold, p. 253.

[6] *Macintosh MacApp 1.1.1 Programmer's Reference*. 1986. Cupertino, CA: Apple Computer, p. 2.

[7] Bhaskar, K. October 1983. How Object-Oriented Is Your System? *SIGPLAN Notices* vol. 18(10), p. 8.

[8] Stefik, M., and Bobrow, D. Winter 1986. Object-Oriented Programming: Themes and Variations. *AI Magazine* vol. 6(4), p. 41.

[9] Yonezawa, A., and Tokoro, M. 1987. Object-Oriented Concurrent Programming: An Introduction. In *Object-Oriented Concurrent Programming*. Cambridge, MA: The MIT Press, p. 2.

[10] Levy, H. 1984. *Capability-Based Computer Systems*. Bedford, MA: Digital Press, p. 13.

[11] Ramamoorthy, C., and Sheu, P. Fall 1988. Object-Oriented Systems. *IEEE Expert* vol. 3(3), p. 14.

[12] Myers, G. 1982. *Advances in Computer Architecture*. Second Edition. New York, NY: John Wiley and Sons, p. 58.

[18] Dijkstra, E. May 1968. The Structure of the "THE" Multiprogramming System. *Communications of the ACM* vol. 11(5).

[19] Pashtan, A. 1982. Object-Oriented Operating Systems: An Emerging Design Methodology. *Proceedings of the ACM '82 Conference*. ACM.

[20] Parnas, D. 1979. On the Criteria to Be Used in Decomposing Systems into Modules. In *Classics in Software Engineering*. New York, NY: Yourdon Press.

[21] Liskov, B., and Zilles, S. 1977. An Introduction to Formal Specifications of Data Abstractions. In *Current Trends in Programming Methodology: Software Specification and Design* vol. 1. Englewood Cliffs, NJ: Prentice-Hall.

[22] Guttag, J. 1980. Abstract Data Types and the Development of Data Structures. In *Programming Language Design*. New York, NY: Computer Society Press.

[23] Shaw, M. October 1984. Abstraction Techniques in Modern Programming Languages. *IEEE Software* vol. 1(4), p. 10.

[24] Nygaard, K., and Dahl, O-J. 1981. The Development of the Simula Languages. In *History of Programming Languages*. New York, NY: Academic Press, p. 460.

[25] Atkinson, M., and Buneman, P. June 1987. Types and Persistence in Database Programming Languages. *ACM Computing Surveys* vol. 19(2), p. 105.

[26] Rumbaugh, J. April 1988. Relational Database Design Using an Object- Oriented Methodology. *Communications of the ACM* vol. 31(4), p. 415.

[27] Chen, P. March 1976. The Entity-Relationship Model—Toward a Unified View of Data. *ACM Transactions on Database Systems* vol. 1(1).

[28] Barr, A., and Feigenbaum, E. 1981. *The Handbook of Artificial Intelligence*. Vol. 1. Los Altos, CA: William Kaufmann, p. 216.

[29] Stillings, N., Feinstein, M., Garfield, J., Rissland, E., Rosenbaum, D., Weisler, S., and Baker-Ward, L. 1987. *Cognitive Science: An Introduction*. Cambridge, MA: The MIT Press, p. 305.

[30] Rand, Ayn. 1979. *Introduction to Objectivist Epistemology*. New York, NY: New American Library.

[31] Minsky, M. 1986. *The Society of Mind*. New York, NY: Simon and Schuster.

[32] Jones, A. 1979. The Object Model: A Conceptual Tool for Structuring Software. In *Operating Systems*. New York, NY: Springer-Verlag, p. 8.

[33] Stroustrup, B. May 1988. What Is Object-Oriented Programming? *IEEE Software* vol. 5(3), p. 10.

[34] Cardelli, L., and Wegner, P. December 1985. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys* vol. 17(4), p. 481.

[35] DeMarco, T. 1979. *Structured Analysis and System Specification*. Englewood Cliffs, NJ: Prentice-Hall.

[36] Yourdon, E. 1989. *Modern Structured Analysis*. Englewood Cliffs, NJ: Prentice-Hall.

[37] Gane, C., and Sarson, T. 1979. *Structured Systems Analysis*. Englewood Cliffs, NJ: Prentice-

Hall.

[38] Ward, P., and Mellor, S. 1985. *Structured Development for Real-Time Systems*. Englewood Cliffs, NJ: Yourdon Press.

[39] Hatley, D., and Pirbhai, I. 1988. *Strategies for Real-Time System Specification*. New York, NY: Dorset House.

[40] Jenkins, M., and Glasgow, J. January 1986. Programming Styles in Nial. *IEEE Software* vol. 3(1), p. 48.

[41] Bobrow, D., and Stefik, M. February 1986. Perspectives on Artificial Intelligence Programming. *Science* vol. 231, p. 951.

[42] Dahl, O., Dijkstra, E., and Hoare, C. A. R. 1972. *Structured Programming*. London, England: Academic Press, p. 83.

[43] Shaw. Abstraction Techniques.

[44] Berzins, V., Gray, M., and Naumann, D. May 1986. Abstraction-Based Software Development. *Communications of the ACM* vol. 29(5), p. 403.

[45] Abelson, H., and Sussman, G. 1985. *Structure and Interpretation of Computer Programs*. Cambridge, MA: The MIT Press, p. 126.

[46] Ibid., p. 132.

[47] Seidewitz, E., and Stark, M. 1986. Towards a General Object-Oriented Software Development Methodology. *Proceedings of the First International Conference on Ada Programming Language Applications for the NASA Space Station*. NASA Lyndon B. Johnson Space Center, TX: NASA, p. D.4.6.4.

[48] Meyer, B. 1988. *Object-Oriented Software Construction*. New York, NY: Prentice Hall.

[49] Wirfs-Brock, R., and Wilkerson, B. October 1989. Object-Oriented Design: A Responsibility-Driven Approach. *SIGPLAN Notices* vol. 24(10).

[50] Ingalls, D. The Smalltalk-76 Programming System Design and Implementation. *Proceedings of the Fifth Annual ACM Symposium on Principles of Programming Languages*. ACM, p. 9.

[51] Gannon, J., Hamlet, R., and Mills, H. July 1987. Theory of Modules. *IEEE Transactions on Software Engineering* vol. SE-13(7), p. 820.

[52] Date, C. 1986. *Relational Database: Selected Writings*. Reading, MA: Addison-Wesley, p. 180.

[53] Liskov, B. May 1988. Data Abstraction and Hierarchy. *SIGPLAN Notices* vol. 23(5), p. 19.

[54] Britton, K., and Parnas, D. December 8, 1981. *A-7E Software Module Guide*. Washington, DC: Naval Research Laboratory, Report 4702, p. 24.

[56] Stroustrup, B. 1988. Private communication.

[57] Myers, G. 1978. *Composite/Structured Design*. New York, NY: Van Nostrand Reinhold, p. 21.

[58] Liskov, B. 1980. A Design Methodology for Reliable Software Systems. In *Tutorial on Software Design Techniques*. Third Edition. New York, NY: IEEE Computer Society, p. 66.

[59] Zelkowitz, M. June 1978. Perspectives on Software Engineering. *ACM Computing Surveys* vol. 10(2), p. 20.

[60] Parnas, D., Clements, P., and Weiss, D. March 1985. The Modular Structure of Complex Systems. *IEEE Transactions on Software Engineering* vol. SE-11(3), p. 260.

[61] Britton and Parnas. *A-7E Software*, p. 2.

[62] Parnas, D., Clements, P., and Weiss, D. 1983. Enhancing Reusability with Information Hiding. *Proceedings of the Workshop on Reusability in Programming,* Stratford, CT: ITT Programming, p. 241.

[63] Meyer. *Object-Oriented Software Construction*, p. 47.

[64] Cox, B. 1986. *Object-Oriented Programming: An Evolutionary Approach.* Reading, MA: Addison-Wesley, p. 69.

[65] Danforth, S., and Tomlinson, C. March 1988. Type Theories and Object-Oriented Programming. *ACM Computing Surveys* vol. 20(1), p. 34.

[66] Liskov. Data Abstraction and Hierarchy, p. 23.

[67] As quoted in Liskov. Design Methodology, p. 67.

[68] Zilles, S. 1984. Types, Algebras, and Modeling. In *On Conceptual Modeling: Perspectives from Artificial Intelligence, Databases, and Programming Languages.* New York, NY: Springer-Verlag, p. 442.

[69] Borning, A., and Ingalls, D. 1982. *A Type Declaration and Inference System for Smalltalk.* Palo Alto, CA: Xerox Palo Alto Research Center, p. 134.

[71] Stroustrup, B. 1992. Private communication.

[72] Tesler, L. August 1981. The Smalltalk Environment. *Byte* vol. 6(8), p. 142.

[73] Borning and Ingalls. *Type Declaration*, p. 133.

[74] Thomas, D. March 1989. What's in an Object? *Byte* vol. 14(3), p. 232.

[76] Lim, J., and Johnson, R. April 1989. The Heart of Object-Oriented Concurrent Programming. *SIGPLAN Notices* vol. 24(4), p. 165.

[77] Black, A., Hutchinson, N., Jul, E., Levy, H., and Carter, L. July 1986. *Distribution and Abstract Types in Emerald.* Report 86-02-04. Seattle, WA: University of Washington, p. 3.

[78] Proceedings of the ACM SIGPLAN Workshop on Object-Based Concurrent Programming. April 1989. *SIGPLAN Notices* vol. 24(4), p. 1.

[79] Atkinson, M., Bailey, P., Chisholm, K., Cockshott, P., and Morrison, R. 1983. An Approach to Persistent Programming. *The Computer Journal* vol. 26(4), p. 360.

[80] Khoshafian, S., and Copeland, G. November 1986. Object Identity. *SIGPLAN Notices* vol. 21(11), p. 409.

[81] *Vbase Technical Overview*. September 1987. Billerica, MA: Ontologic, p. 4.

[82] Stroustrup, B. November 1987. Possible Directions for C++. *Proceedings of the USENIX C++*

Workshop. Santa Fe, NM, p. 14.

[83] Meyer. *Object-Oriented Software Construction*, pp. 30-31.

[84] Robson, D. August 1981. Object-Oriented Software Systems. *Byte* vol. 6(8), p. 74.

[85] See www.hibernate.org for more information and downloads.

第 3 章：类与对象

[1] Lefrancois, G. 1977. *Of Children: An Introduction to Child Development*. Second Edition. Belmont, CA: Wadsworth, p. 244-246.

[2] Nygaard, K., and Dahl, O-J. 1981. The Development of the Simula Languages. In *History of Programming Languages*. New York, NY: Academic Press, p. 462.

[3] Halbert, D., and O'Brien, P. September 1988. Using Types and Inheritance in Object-Oriented Programming. *IEEE Software* vol. 4(5), p. 73.

[4] Smith, M., and Tockey, S. 1988. *An Integrated Approach to Software Requirements Definition Using Objects*. Seattle, WA: Boeing Commercial Airplane Support Division, p. 132.

[6] MacLennan, B. December 1982. Values and Objects in Programming Languages. *SIGPLAN Notices* vol. 17(12), p. 78.

[7] Lippman, S. 1989. *C++ Primer*. Reading, MA: Addison-Wesley, p. 185.

[8] Adams, S. 1993. Private communication.

[9] Wirfs-Brock, R., Wilkerson, B., and Wiener, L. 1990. *Designing Object- Oriented Software*. Englewood Cliffs, NJ: Prentice Hall, p. 61.

[10] Rubin, K. 1993. Private communication.

[11] *Macintosh MacApp 1.1.1 Programmer's Reference*. 1986. Cupertino, CA: Apple Computer, p. 4.

[12] Khoshafian, S., and Copeland, G. November 1986. Object Identity. *SIGPLAN Notices* vol. 21(11), p. 406.

[13] Ingalls, D. 1981. Design Principles Behind Smalltalk. *Byte* vol. 6(8), p. 290.

[14] Gall, J. 1986. *Systemantics: How Systems Really Work and How They Fail*. Second Edition. Ann Arbor, MI: The General Systemantics Press, p. 158.

[16] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W. 1991. *Object-Oriented Modeling and Design*. Englewood Cliffs, NJ: Prentice-Hall, p. 459.

[17] *Webster's Third New International Dictionary of the English Language,* unabridged. 1986. Chicago, IL: Merriam-Webster.

[18] Stroustrup, B. 1991. *The C+ Programming Language*. Second Edition. Reading, MA: Addison-Wesley, p. 422.

[19] Meyer, B. 1987. *Programming as Contracting*. Report TR-EI-12/CO. Goleta, CA: Interactive Software Engineering.

[20] Snyder, A. November 1986. Encapsulation and Inheritance in Object- Oriented Programming

Languages. *SIGPLAN Notices* vol. 21(11).

[21] LaLonde, W. April 1989. Designing Families of Data Types Using Exemplars. *ACM Transactions on Programming Languages and Systems* vol. 11(2), p. 214.

[22] Rumbaugh, J. April 1988. Relational Database Design Using an Object- Oriented Methodology. *Communications of the ACM* vol. 31(4), p. 417.

[25] Brachman, R. October 1983. What IS-A Is and Isn't: An Analysis of Taxonomic Links in Semantic Networks. *IEEE Computer* vol. 16(10), p. 30.

[29] As quoted in Harland, D., Szyplewski, M., and Wainwright, J. October 1985. An Alternative View of Polymorphism. *SIGPLAN Notices* vol. 20(10).

[30] Kaplan, S., and Johnson, R. July 21, 1986. *Designing and Implementing for Reuse*. Urbana, IL: University of Illinois, Department of Computer Science, p. 8.

[31] Deutsch, P. 1983. Efficient Implementation of the Smalltalk-80 System. In *Proceedings of the 11th Annual ACM Symposium on the Principles of Programming Languages*, p. 300.

[32] Ibid., p. 299.

[33] Duff, C. August 1986. Designing an Efficient Language. *Byte* vol. 11(8), p. 216.

[34] Stroustrup, B. 1988. Private communication.

[40] Vlissides, J., and Linton, M. 1988. Applying Object-Oriented Design to Structured Graphics. *Proceedings of USENIX C++ Conference*. Berkeley, CA: USENIX Association, p. 93.

[41] Meyer, B. 1988. *Object-Oriented Software Construction*. New York, NY: Prentice Hall, p. 274.

[44] Hendler, J. October 1986. Enhancement for Multiple Inheritance. *SIGPLAN Notices* vol. 21(10), p. 100.

[45] Object Management Group's Web site: www.omg.org/.

[51] Ingalls, D. August 1981. Design Principles Behind Smalltalk. *Byte* vol. 6(8), p. 286.

[52] Stevens, W., Myers, G., and Constantine, L. 1979. Structured Design. In *Classics in Software Engineering*. New York, NY: Yourdon Press, p. 209.

[53] Page-Jones, M. 1988. *The Practical Guide to Structured Systems Design*. Englewood Cliffs, NJ: Yourdon Press, p. 59.

[54] Meyer. *Programming as Contracting*, p. 4.

[55] Halbert, D., and O'Brien, P. September 1988. Using Types and Inheritance in Object-Oriented Programming. *IEEE Software* vol. 4(5), p. 74.

[56] Sakkinen, M. December 1988. Comments on "the Law of Demeter" and C++. *SIGPLAN Notices* vol. 23(12), p. 38.

[57] Lea, D. August 12, 1988. *User's Guide to GNU C++ Library*. Cambridge, MA: Free Software Foundation, p. 12.

[58] Ibid.

[59] Meyer. *Object-Oriented Software Construction*, p. 332.

[60] Wirth, N. 1986. *Algorithms and Data Structures*. Englewood Cliffs, NJ: Prentice-Hall, p. 37.

[61] Keene, S. 1989. *Object-Oriented Programming in Common Lisp*. Reading, MA: Addison-Wesley, p. 68.

[62] Jacobson, I., Christerson, M., Jonsson, P., and Övergaard, G. 1992. *Object- Oriented Software Engineering: A Use Case Driven Approach*. Reading, MA: Addison-Wesley, p. 186.

[63] Parnas, D., Clements, P., and Weiss, D. 1989. Enhancing Reusability with Information Hiding. In *Software Reusability*. New York, NY: ACM Press, p. 143.

第 4 章：分类

[2] Michalski, R., and Stepp, R. 1983. Learning from Observation: Conceptual Clustering. In *Machine Learning: An Artificial Intelligence Approach*. Palo Alto, CA: Tioga, p. 332.

[3] Alexander, C. 1979. *The Timeless Way of Building*. New York, NY: Oxford University Press, p. 203.

[4] Darwin, C. 1984. *The Origin of Species. Vol. 49 of Great Books of the Western World*. Chicago, IL: Encyclopedia Britannica, p. 207.

[5] *The New Encyclopedia Britannica*. 1985. Chicago, IL: Encyclopedia Britannica, vol. 3, p. 356.

[8] As quoted in Lewin, R. November 4, 1988. Family Relationships Are a Biological Conundrum. *Science* vol. 242, p. 671.

[9] *The New Encyclopedia Britannica* vol. 3, p. 156.

[10] Descartes, R. 1984. *Rules for the Direction of the Mind. Vol. 31 of Great Books of the Western World*. Chicago, IL: Encyclopedia Britannica, p. 32.

[11] Shaw, M. May 1989. Larger Scale Systems Require Higher-Level Abstractions. *SIGSOFT Engineering Notes* vol. 14(3), p. 143.

[12] Goldstein, T. May 1989. The Object-Oriented Programmer. *The C++ Report* vol. 1(5).

[13] Coombs, C., Raiffa, H., and Thrall, R. 1954. Some Views on Mathematical Models and Measurement Theory. *Psychological Review* vol. 61(2), p. 132.

[15] Birtwistle, G., Dahl, O-J., Myhrhaug, B., and Nygard, K. 1979. *Simula begin*. Lund, Sweden: Studentlitteratur, p. 23.

[16] Heinlein, R. 1966. *The Moon Is a Harsh Mistress*. New York, NY: The Berkeley Publishing Group, p. 11.

[17] Sowa, J. 1984. *Conceptual Structures: Information Processing in Mind and Machine*. Reading, MA: Addison-Wesley, p. 16.

[18] Lakoff, G. 1987. *Women, Fire, and Dangerous Things: What Categories Reveal About the Mind*. Chicago, IL: The University of Chicago Press, p. 161.

[19] Stepp, R., and Michalski, R. February 1986. Conceptual Clustering of Structured Objects: A Goal-Oriented Approach. *Artificial Intelligence* vol. 28(1), p. 53.

[20] Wegner, P. 1987. The Object-Oriented Classification Paradigm. In *Research Directions in Object-Oriented Programming*. Cambridge, MA: The MIT Press, p. 480.

[21] Aquinas, T. 1984. *Summa Theologica. Vol. 19 of Great Books of the Western World*. Chicago, IL: Encyclopedia Britannica, p. 71.

[22] Maier, H. 1969. *Three Theories of Child Development: The Contributions of Erik H. Erickson, Jean Piaget, and Robert R. Sears, and Their Applications*. New York, NY: Harper and Row, p. 111.

[23] Lakoff. *Women, Fire, and Dangerous Things*, p. 32.

[24] Minsky, M. 1986. *The Society of Mind*. New York, NY: Simon and Schuster, p. 199.

[25] *The Great Ideas: A Syntopicon of Great Books of the Western World*. 1984. *Vol. 1 of Great Books of the Western World*. Chicago, IL: Encyclopedia Britannica, p. 293.

[26] Kosko, B. 1992. *Neural Networks and Fuzzy Systems*. Englewood Cliffs, NJ: Prentice-Hall.

[27] Stepp and Michalski. Conceptual Clustering, p. 44.

[28] Lakoff. *Women, Fire, and Dangerous Things*, p. 7.

[29] Ibid., p. 16.

[30] Lakoff, G., and Johnson, M. 1980. *Metaphors We Live By*. Chicago, IL: The University of Chicago Press, p. 122.

[31] Meyer, B. 1988. Private communication.

[32] Shlaer, S., and Mellor, S. 1988. *Object-Oriented Systems Analysis: Modeling the World in Data*. Englewood Cliffs, NJ: Yourdon Press, p. 15.

[33] Ross, R. 1987. *Entity Modeling: Techniques and Application*. Boston, MA: Database Research Group, p. 9.

[34] Coad, P., and Yourdon, E. 1990. *Object-Oriented Analysis*. Englewood Cliffs, NJ: Prentice-Hall, p. 62.

[35] Shlaer, S., and Mellor, S. 1992. *Object Lifecycles: Modeling the World in States*. Englewood Cliffs, NJ: Yourdon Press.

[36] Wirfs-Brock, R., Wilkerson, B., and Wiener, L. 1990. *Designing Object- Oriented Software*. Englewood Cliffs, NJ: Prentice Hall, p. 61.

[37] Rubin, K., and Goldberg, A. September 1992. Object Behavior Analysis. *Communications of the ACM* vol. 35(9), p. 48.

[38] Dreger, B. 1989. *Function Point Analysis*. Englewood Cliffs, NJ: Prentice Hall, p. 4.

[39] Arango, G. May 1989. Domain Analysis: From Art Form to Engineering Discipline. *SIGSOFT Engineering Notes* vol. 14(3), p. 153.

[40] Moore, J., and Bailin, S. 1988. *Position Paper on Domain Analysis*. Laurel, MD: CTA, p. 2.

[41] Jacobson, I., Ericsson, M., and Jacobson, A. 1994. *The Object Advantage: Business Process Reengineering with Object Technology*. Wokingham, England: Addison-Wesley.

[42] Zahniseer, R. July/August 1990. Building Software in Groups. *American Programmer* vol. 3(7-8).

[44] Beck, K., and Cunningham, W. October 1989. A Laboratory for Teaching Object-Oriented Thinking. *SIGPLAN Notices* vol. 24(10).

[45] Abbott, R. November 1983. Program Design by Informal English Descriptions. *Communications of the ACM* vol. 26(11).

[47] McMenamin, S., and Palmer, J. 1984. *Essential Systems Analysis*. New York, NY: Yourdon Press, p. 267.

[48] Ward, P., and Mellor, S. 1985. *Structured Development for Real-Time Systems*. Englewood Cliffs, NJ: Yourdon Press.

[49] Seidewitz, E., and Stark, M. August 1986. *General Object-Oriented Software Development*. Report SEL-86-002. Greenbelt, MD: NASA Goddard Space Flight Center, pp. 5-2.

[50] Seidewitz, E. 1990. Private communication.

[51] Goldberg, A. 1984. *Smalltalk-80: The Interactive Programming Environment*. Reading, MA: Addison-Wesley, p. 77.

[52] Thomas, D. May/June 1989. In Search of an Object-Oriented Development Process. *Journal of Object-Oriented Programming* vol. 2(1), p. 61.

[53] Stroustrup, B. 1986. *The C++ Programming Language*. Reading, MA: Addison-Wesley, p. 7.

[54] Halbert, D., and O'Brien, P. September 1988. Using Types and Inheritance in Object-Oriented Programming. *IEEE Software* vol. 4(5), p. 75.

[55] Stefik, M., and Bobrow, D. Winter 1986. Object-Oriented Programming: Themes and Variations. *AI Magazine* vol. 6(4), p. 60.

[56] Stroustrup, B. 1991. *The C+ Programming Language*. Second Edition. Reading, MA: Addison-Wesley, p. 377.

[57] Stefik and Bobrow. Object-Oriented Programming, p. 58.

[58] Lins, C. 1989. A First Look at Literate Programming. In *Structured Programming*.

[59] Gabriel, R. 1990. Private communication.

[60] Coplien, J. 1992. *Advanced C++ Programming Styles and Idioms*. Reading, MA: Addison-Wesley.

[61] Adams, S. July 1986. MetaMethods: The MVC Paradigm. In *HOOPLA: Hooray for Object-Oriented Programming Languages*. Everette, WA: Object-Oriented Programming for Smalltalk Applications Developers Association vol. 1(4), p. 6.

[63] Englemore, R., and Morgan, T. 1988. *Blackboard Systems*. Wokingham, England: Addison-Wesley, p. v.

[64] Coad, P. September 1992. Object-Oriented Patterns. *Communications of the ACM* vol. 35(9).

[65] World Resources Institute Web site. Accessed Jan. 2006. How Many Species Are There?.

http://pubs.wri.org/pubs_content_ text.cfm?ContentID=535.

第 2 部分：方法

Petroski, H. 1985. *To Engineer Is Human*. New York, NY: St. Martin's Press, p. 73.

第 5 章：表示法

[1] Shear, D. December 8, 1988. CASE Shows Promise, but Confusion Still Exists. *EDN* vol. 33(25), p. 168.

[2] Whitehead, A. 1958. *An Introduction to Mathematics*. New York, NY: Oxford University Press.

[3] Defense Science Board. September 1987. *Report of the Defense Science Board Task Force on Military Software*. Washington, DC: Office of the Undersecretary of Defense for Acquisition, p. 8.

[4] Kleyn, M., and Gingrich, P. September 1988. GraphTrace—Understanding Object-Oriented Systems Using Concurrently Animated Views. *SIGPLAN Notices* vol. 23(11), p. 192.

[5] Weinberg, G. 1988. *Rethinking Systems Analysis and Design*. New York, NY: Dorset House, p. 157.

[16] Jacobson, I., Christerson, M., Jonsson, P., and Övergaard, G. 1992. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Reading, MA: Addison-Wesley.

[17] Maksimchuk, R., and Naiburg, E. 2005. *UML for Mere Mortals*. Boston, MA: Addison-Wesley.

[18] Blaha, M., and Rumbaugh, J. 2005. *Object-Oriented Modeling and Design with UML*. Second Edition. Upper Saddle River, NJ: Prentice Hall, pp. 116-118.

[19] Rumbaugh, J., Jacobson, I., and Booch, G. 2005. *The Unified Modeling Language Reference Manual*. Second Edition. Boston, MA: Addison- Wesley, pp. 248-253.

[20] Ibid., pp. 40, 109, 171.

[21] Object Management Group. 2004. *UML 2.0 Superstructure Specification*. Needham, MA: Object Management Group, pp. 206-208.

[22] Rumbaugh, Jacobson, and Booch. *The Unified Modeling Language Reference Manual*, pp. 40, 171-172.

[23] Ibid., pp. 109, 479.

[24] Object Management Group. *UML 2.0 Superstructure Specification*, p. 222.

[25] Rumbaugh, Jacobson, and Booch. *The Unified Modeling Language Reference Manual*, p. 312.

[26] Ibid., p. 313.

[27] Object Management Group. *UML 2.0 Superstructure Specification*, p. 219.

[28] Ibid., pp. 213-214.

[29] Rumbaugh, Jacobson, and Booch. *The Unified Modeling Language Reference Manual*, pp.

187, 498.
[30] Ibid., pp. 129-133.
[31] Ibid., p. 285.
[32] Ibid., pp. 285-288.
[33] Ibid., pp. 285-287.
[34] Ibid., pp. 374-375.
[35] Ibid., p. 429.
[36] Ibid., p. 623.
[37] Object Management Group. *UML 2.0 Superstructure Specification*, p. 174.
[38] Rumbaugh, Jacobson, and Booch. *The Unified Modeling Language Reference Manual*, pp. 523-525.
[39] Object Management Group. *UML 2.0 Superstructure Specification*, pp. 189-190.
[40] Rumbaugh, Jacobson, and Booch. *The Unified Modeling Language Reference Manual*, pp. 282-283.
[41] Object Management Group. *UML 2.0 Superstructure Specification*, pp. 169, 176-181.
[42] Rumbaugh, Jacobson, and Booch. *The Unified Modeling Language Reference Manual*, pp. 42-43, 111-112, 504-507.
[43] Ibid., p. 679.
[44] Ibid., pp. 71-73, 227-231.
[45] Ibid., pp. 506-507.
[46] Object Management Group. *UML 2.0 Superstructure Specification*, pp. 169, 111-112.
[47] Ibid., pp. 111-112.
[48] Rumbaugh, Jacobson, and Booch. *The Unified Modeling Language Reference Manual*, pp. 506-507.
[49] Ibid., pp. 113, 505-506, 678-680.
[50] Object Management Group. *UML 2.0 Superstructure Specification*, pp. 109-111.
[51] Ibid., pp. 109-111.
[52] Rumbaugh, Jacobson, and Booch. *The Unified Modeling Language Reference Manual*, pp. 113, 505-506, 678-680.
[53] Ibid., pp. 112-113, 384, 505-506.
[54] Ibid., pp. 112-113, 505-506.
[55] Ibid., pp. 42, 111-112, 507.
[56] Ibid., pp. 42, 111-112, 507.
[57] Ibid., pp. 42, 111-112, 507.
[58] Ibid., pp. 42, 383-384, 505.
[59] Object Management Group. *UML 2.0 Superstructure Specification*, p. 111.

[60] Ibid., pp. 113-114.

[61] Rumbaugh, Jacobson, and Booch. *The Unified Modeling Language Reference Manual*, pp. 383-387, 505.

[62] Ibid., pp. 383-387, 505.

[63] Object Management Group. *UML 2.0 Superstructure Specification*, pp. 113-114.

[64] Ibid., p. 111.

[65] Rumbaugh, Jacobson, and Booch. *The Unified Modeling Language Reference Manual*, pp. 384-385, 505.

[66] Ibid., pp. 383-387, 505.

[67] Object Management Group. *UML 2.0 Superstructure Specification*, pp. 113-114.

[68] Rumbaugh, Jacobson, and Booch. *The Unified Modeling Language Reference Manual*, pp. 31, 73, 253, 255.

[69] Object Management Group. *UML 2.0 Superstructure Specification*, pp. 147, 150, 153.

[70] Ibid., p. 153.

[71] Rumbaugh, Jacobson, and Booch. *The Unified Modeling Language Reference Manual*, pp. 31, 74, 256.

[72] Ibid., pp. 74, 254, 257, 523-525.

[73] Object Management Group. *UML 2.0 Superstructure Specification*, pp. 150-151.

[74] Ibid., pp. 150-151, 154.

[75] Rumbaugh, Jacobson, and Booch. *The Unified Modeling Language Reference Manual*, pp. 31, 74, 254, 256, 416.

[76] Ibid., pp. 254, 257.

[77] Object Management Group. *UML 2.0 Superstructure Specification*, p. 162.

[78] Ibid., pp. 147, 150, 153.

[79] Rumbaugh, Jacobson, and Booch. *The Unified Modeling Language Reference Manual*, pp. 73-74, 253-254.

[80] Ibid., pp. 31-33, 257-258, 282-283.

[81] Object Management Group. *UML 2.0 Superstructure Specification*, pp. 150, 156, 160.

[82] Ibid., pp. 154-155.

[83] Rumbaugh, Jacobson, and Booch. *The Unified Modeling Language Reference Manual*, pp. 256-257, 415-417.

[84] Ibid., pp. 256-257, 418.

[85] Object Management Group. *UML 2.0 Superstructure Specification*, p. 155.

[86] Ibid., p. 155.

[87] Rumbaugh, Jacobson, and Booch. *The Unified Modeling Language Reference Manual*, pp. 256-257, 418.

[88] Object Management Group. *UML 2.0 Superstructure Specification*, pp. 155-156.

[89] Ibid., p. 150.

[90] Rumbaugh, Jacobson, and Booch. *The Unified Modeling Language Reference Manual*, p. 31.

[91] Ibid., pp. 258, 283.

[92] Object Management Group. *UML 2.0 Superstructure Specification*, pp. 160-162.

[93] Ibid., pp. 169, 176-181.

[94] Rumbaugh, Jacobson, and Booch. *The Unified Modeling Language Reference Manual*, pp. 71-73, 227-231.

[95] Ibid.

[96] Object Management Group. *UML 2.0 Superstructure Specification*, pp. 169, 176-181.

[97] Ibid.

[98] Rumbaugh, Jacobson, and Booch. *The Unified Modeling Language Reference Manual*, pp. 71-73, 227-231.

[99] Ibid.

[100] Object Management Group. *UML 2.0 Superstructure Specification*, pp. 169, 176-181.

[101] Rumbaugh, Jacobson, and Booch. *The Unified Modeling Language Reference Manual*, p. 98.

第 6 章：过程

[1] Brooks, F. 1975. *The Mythical Man-Month*. Reading, MA: Addison- Wesley, p. 42.

[2] Stroustrup, B. 1991. *The C+ Programming Language*. Second Edition. Reading, MA: Addison-Wesley.

[6] Jones, C. September 1984. Reusability in Programming: A Survey of the State of the Art. *IEEE Transactions on Software Engineering* vol. SE-10(5).

[9] Parnas, D., and Clements, P. 1986. A Rational Design Process: How and Why to Fake It. *IEEE Transactions on Software Engineering* vol. SE-12(2).

[13] Stroustrup. *The C+ Programming Language*, p. 373.

[15] Gilb, T. 1988. *Principles of Software Engineering Management*. Reading, MA: Addison-Wesley, p. 92.

[16] Mellor, S., Hecht, A., Tryon, D., and Hywari, W. September 1988. Object- Oriented Analysis: Theory and Practice, Course Notes. In *Object-Oriented Programming Systems, Languages, and Applications.* San Diego, CA: OOPSLA'88, p. 1.3.

[24] Andert, G. 1992. Private communication.

[29] Walsh, J. *Preliminary Defect Data from the Iterative Development of a Large C++ Program*. Vancouver, Canada: OOPSLA'92.

[30] Chmura, L., Norcio, A., and Wicinski, T. July 1990. Evaluating Software Design Processes by Analyzing Change Date Over Time. *IEEE Transactions on Software Engineering* vol. 16(7).

[31] As quoted in Summerville, I. 1989. *Software Engineering*. Third Edition. Wokingham, England: Addison-Wesley, p. 546.

[32] The Zachman Institute for Framework Advancement: www.zifa.com/.

[33] Department of Defense Architecture Framework Working Group. Februrary 9, 2004. *DoD Architecture Framework, Version 1.0. Volume 1: Definitions and Guidelines*. Accessed in January 2007 at www.dod.mil/cio-nii/docs/DoDAF_v1_Volume_I.pdf.

[34] For more information on the Federal Enterprise Architecture, visit www.whitehouse.gov/omb/egov/a-1-fea.html.

[35] Agile Alliance Web site. Accessed in January 2007 at http://agilemanifesto.org/principles.html.

[38] Boehm, B. W., and Turner, R. 2004. *Balancing Agility and Discipline: A Guide for the Perplexed*. Boston, MA: Addison-Wesley, 2004.

[39] Booch, G., Rumbaugh, J., and Jacobson, I. 1999. *The Unified Modeling Language User's Guide*. Reading, MA: Addison-Wesley, 1999.

[40] Booch, G., and Brown, A. W. October 28, 2002. *Collaborative Development Environments*. Rational Software Corporation.

[41] Eeles, P. February 15, 2006. What Is a Software Architecture? Accessed in January 2007 at www-128.ibm.com/developerworks/rational/library/feb06/eeles/.

[42] IEEE Computer Society. 2000. IEEE Recommended Practice for Architectural Description of Software Intensive Systems. IEEE Std. 1471-2000.

[43] Kruchten, P. November 1995. The 4+1 View Model of Architecture. *IEEE Software* 12(6), pp. 42-50.

[44] Kruchten, P. August 2001. Software Maintenance Cycles with the RUP. *Rational Edge*. Accessed in January 2007 at www.ibm.com/developerworks/rational/library/content/RationalEdge/aug01/SoftwareMaintenanceCycleswiththeRUPAug01.pdf.

[45] Kroll, P., and Kruchten, P. 2003. *The Rational Unified Process Made Easy*. Boston, MA: Addison-Wesley.

[46] Larman, C. 2004. *Agile and Iterative Development*. Boston, MA: Addison-Wesley.

[47] Martin, R. C. 2002. *Agile Software Development, Principles, Patterns, and Practices*. Upper Saddle River, NJ: Prentice Hall.

[48] For more information about MDA, see the Object Management Group's related Web site, www.omg.org/mda.

[51] The Rational Unified Process product produced by IBM Rational, 2006.

[52] Adams, J., Koushik, S., Vasudeva, G., and Galambos, G. 2001. *Patterns for e-Business: A Strategy for Reuse*. Double Oak, TX: IBM Press.

[53] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. 1996. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. New York, NY: John Wiley & Sons.

[54] Cheesman, J., and Daniels, J. 2001. *UML Components: A Simple Process for Specifying Component-Based Software*. Boston, MA: Addison-Wesley.

[55] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley.

[56] Herzum, P., and Sims, O. 2000. *Business Component Factory: A Comprehensive Overview of Component-Based Development for the Enterprise*. New York, NY: Wiley.

第 7 章：实践

[1] Goldstein, H. September 2005. Who Killed the Virtual Case File? *IEEE Spectrum* vol. 42, p. 24.

[2] Miller, J. March 20, 2006. FBI Awards Lockheed $305 Million Sentinel Contract. *Government Computer News*. Accessed in January 2007 at www.gcn.com/online/vol1_no1/40145-1.html.

[3] Charette, R. September 2005. Why Software Fails. *IEEE Spectrum* vol. 42, p. 42.

[4] Hawryszkiewycz, I. 1984. *Database Analysis and Design*. Chicago, IL: Science Research Associates, p. 115.

[5] van Genuchten, M. June 1991. Why Is Software Late? An Empirical Study of Reasons for Delay in Software Development. *IEEE Transactions on Software Engineering* vol. 17(6), p. 589.

[6] Gilb, T. 1988. *Principles of Software Engineering Management*. Reading, MA: Addison-Wesley, p. 73.

[7] As quoted in Zelkowitz, M. June 1978. Perspectives on Software Engineering. *ACM Computing Surveys* vol. 10(2), p. 204.

[8] Showalter, J. 1989. Private communication.

[9] Davis, A., Bersoff, E., and Comer, E. October 1988. A Strategy for Comparing Alternative Software Development Life Cycle Models. *IEEE Transactions on Software Engineering* vol. 14(10), p. 1456.

[10] Goldberg, A. 1993. Private communication.

[11] Schulmeyer, G., and McManus, J. 1992. *Handbook of Software Quality Assurance*. Second Edition. New York, NY: Van Nostrand Reinhold, p. 7.

[12] Schulmeyer and McManus. *Handbook of Software Quality Assurance*, p. 5.

[13] Schulmeyer and McManus. *Handbook of Software Quality Assurance*, p. 184.

[14] Schulmeyer and McManus. *Handbook of Software Quality Assurance*, p. 169.

[15] Walsh, J. *Preliminary Defect Data from the Iterative Development of a Large C++ Program*. Vancouver, Canada: OOPSLA'92.

[16] Lorenz, M., and Kidd, J. 1994. *Object-Oriented Software Metrics*. Upper Saddle River, NJ: Prentice Hall.

[17] Chidamber, S., and Kemerer, C. 1993. *A Metrics Suite for Object-Oriented Design*. Cambridge,

MA: MIT Sloan School of Management.

[18] Kan, S. H. 2002. *Metrics and Models in Software Quality Engineering*. Second Edition. Boston, MA: Addison-Wesley.

[19] Kemerer, C., and Darcy, D. November/December 2005. OO Metrics in Practice. *IEEE Software*, p. 17.

[20] Stix, A., and Mosley, P. H. 2002. Cognitive Complexities Confronting Software Developers Utilizing Object Technology. *The Proceedings of ISECON 2002* vol. 19, §342a.

[21] Maksimchuk, R. A., and Naiburg, E. J. 2004. *UML for Mere Mortals*. Boston, MA: Addison-Wesley, p. 208.

[22] Schmucker, K. 1986. *Object-Oriented Programming for the Macintosh*. Hasbrouk Heights, NJ: Hayden, p. 11.

[23] Taylor, D. 1992. *Object-Oriented Information Systems*. New York, NY: John Wiley and Sons.

[24] Pinson, L., and Wiener, R. 1990. *Applications of Object-Oriented Programming*. Reading, MA: Addison-Wesley.

[25] Hantos, P. February 2005. Inherent Risks in Object-Oriented Development. *CrossTalk, The Journal of Defense Software Engineering*, pp. 13-16; Fig. 2. Accessed in January 2007 at www.stsc.hill.af.mil.

[26] Meyer, B. 1995. *Object Success: A Manager's Guide to Object-Oriented Technology and Its Impact on the Corporation*. Upper Saddle River, NJ: Prentice Hall.

[27] Boehm, B. 1989. *Software Risk Management (Tutorial)*. New York, NY: IEEE Computer Society Press.

[28] Boehm, B. October 2002. Software Risk Management: Overview and Recent Developments. *17th International Forum on COCOMO and Software Cost Modeling*. Los Angeles, CA: University of Southern California Center for Software Engineering.

[29] Lyons, B. 2005. Private communication.

[30] Ibid.

第 3 部分：应用

Minsky, M. April 1970. Form and Content in Computer Science. *Journal of the Association for Computing Machinery* vol. 17(2), p. 197.

第 8 章：系统架构——基于卫星的导航

[1] International Council on Systems Engineering (INCOSE). June 2006. *Systems Engineering Handbook*. INCOSE-TP-2003-016-02, Version 3. Seattle, WA: INCOSE, app. 8.

[2] INCOSE. June 2004. *Systems Engineering Handbook*. INCOSE-TP-2003-016-02, Version 2a. Seattle, WA: INCOSE, p. 293.

[3] The Aerospace Corporation. 2003. *GPS Primer—A Student Guide to the Global Positioning System*, p. 2. Accessed in January 2007 at www.aero.org/education/primers/gps/GPS-Primer.pdf.
[4] Ibid., pp. 2-3.
[5] Ibid., p. 4.
[6] Ibid., p. 5.
[7] Ibid., p. 6.
[8] Rumbaugh, J., Jacobson, I., and Booch, G. 2005. *The Unified Modeling Language Reference Manual*. Second Edition. Boston, MA: Addison- Wesley, p. 37.
[9] Maier, M. W., Emery, D., and Hilliard, R. 2004. ANSI/IEEE 1471 and Systems Engineering. *Systems Engineering* vol. 7(3), p. 257.
[10] Ibid., p. 269.
[11] Krikorian, H. F. March/April 2003. Introduction to Object-Oriented Systems Engineering, Part 1. *IT Professional*, p. 40.
[12] Jacobson, I., Ericsson, M., and Jacobson, A. 1994. *The Object Advantage: Business Process Reengineering with Object Technology*. Wokingham, England: Addison-Wesley, pp. 319-337.

第 9 章：控制系统——交通管理

[1] Murphy, E. December 1988. All Aboard for Solid State. *IEEE Spectrum* vol. 25(13), p. 42.
[2] *Rockwell Advanced Railroad Electronic Systems*. 1989. Cedar Rapids, IA: Rockwell International.
[3] Tanenbaum, A. 1981. *Computer Networks*. Englewood Cliffs, NJ: Prentice-Hall.

第 10 章：人工智能——密码分析

[1] Erman, L., Lark, J., and Hayes-Roth, F. December 1988. ABE: An Environment for Engineering Intelligent Systems. *IEEE Transactions on Software Engineering* vol. 14(12), p. 1758.
[2] Shaw, M. 1991. *Heterogeneous Design Idioms for Software Architecture*. Pittsburgh, PA: Carnegie Mellon University.
[3] Meyer, C., and Matyas, S. M. 1982. *Cryptography*. New York, NY: John Wiley and Sons, p. 1.
[4] Nii, P. Summer 1986. Blackboard Systems: The Blackboard Model of Problem Solving and the Evolution of Blackboard Architectures. *AI Magazine* vol. 7(2), p. 46.
[5] Englemore, R., and Morgan, T. 1988. *Blackboard Systems*. Wokingham, England: Addison-Wesley, p. 16.
[6] Ibid., p. 19.
[7] Ibid., p. 6.
[8] Ibid., p. 12.
[9] Nii. Blackboard Systems, p. 43.
[10] Englemore and Morgan. *Blackboard Systems*, p. 11.

第 12 章：Web 应用——休假跟踪系统

[1] Conallen, J. 2003. *Building Web Applications with UML*. Second Edition. Boston, MA: Addison-Wesley.

[2] Eeles, P., Houston, K. A., and Kozaczynski, W. 2003. *Building J2EE Applications with the Rational Unified Process*. Boston, MA: Addison- Wesley.

[3] Kruchten, P. November 1995. The 4+1 View Model of Architecture. *IEEE Software* 12(6), pp. 42-50.

[4] Alur, D., Crupi, J., and Malks, D. 2003. *Core J2EE Patterns: Best Practices and Design Strategies*. Second Edition. Upper Saddle River, NJ: Prentice Hall.

[5] Marinescu, F. 2002. *EJB Design Patterns*. New York, NY: John Wiley.

附录 A：面向对象编程语言

[1] Wulf, W. January 1980. Trends in the Design and Implementation of Programming Languages. *IEEE Computer* vol. 13(1), p. 15.

[2] The Language List, a compendium of programming languages, originating in the Usenet news group comp.lang.misc, currently maintained by Bill Kinnersley. http://people.ku.edu/~nkinners/LangList/Extras/langlist.htm.

[3] Birtwistle, G., Dahl, O-J., Myhrhaug, B., and Nygard, K. 1979. *Simula begin*. Lund, Sweden: Studentlitteratur.

[4] Éric Lévénez's Web site on computer language history: www.levenez.com/lang/.

[5] The TIOBE Programming Community Index: www.tiobe.com/tiobe_index/index.htm.

[6] The Smalltalk Web site: www.smalltalk.org/smalltalk/history.html.

[7] Ingalls, D. The Smalltalk-76 Programming System Design and Implementation. *Proceedings of the Fifth Annual ACM Symposium on Principals of Programming Languages*, ACM, p. 9.

[8] Borning, A., and Ingalls, D. 1982. Multiple Inheritance in Smalltalk-80. *Proceedings of the National Conference on Artificial Intelligence*. Menlo Park, CA: AAAI.

[9] Goldberg, A., and Robson, D. 1989. *Smalltalk-80: The Language*. Reading, MA: Addison-Wesley.

[10] Goldberg, A. 1984. *Smalltalk-80: The Interactive Programming Environment*. Reading, MA: Addison-Wesley.

[11] Krasner, G. 1983. *Smalltalk-80: Bits of History, Words of Advice*. Reading, MA: Addison-Wesley.

[12] LaLonde, W., and Pugh, J. 1990. *Inside Smalltalk, Volumes 1 and 2*. Englewood Cliffs, NJ: Prentice Hall.

[13] Stroustrup, B. 2000. *The C+ Programming Language*. Special Third Edition. Boston, MA:

Addison-Wesley, p. 4.

[14] Gorlen, K. 1989. An Introduction to C++. In *Unix System V AT&T C++ Language System, Release 2.0 Selected Readings*. Murray Hill, NJ: AT&T Bell Laboratories, p. 2-1.

[15] Ellis, M., and Stroustrup, B. 1990. *The Annotated C++ Reference Manual*. Reading, MA: Addison-Wesley.

[16] Stroustrup, B. 2003. *The C++ Standard: Incorporating Technical Corrigendum No. 1*. Second Edition. New York, NY: John Wiley & Sons.

[17] Stroustrup, B. 1991. *The C++ Programming Language*. Second Edition. Reading, MA: Addison-Wesley.

[18] Stroustrup, B. 1994. *The Design and Evolution of C++*. Reading, MA: Addison-Wesley.

[19] The Java Language: An Overview. Accessed in January 2007 at http://java.sun.com/docs/overviews/java/java-overview-1.html.

[20] International Committee for Information Technology Standards. January 1998. *Programming Language Smalltalk*. Document Number ANSI/INCITS 319-1998.

术语表

A

abstract class（抽象类） 没有实例的类。编写一个抽象类，是希望它的具体子类给它添加结构和行为，通常通过实现它的抽象操作来达到该目的。

abstract operation（抽象操作） 由一个抽象类声明但不实现的操作。

abstraction（抽象） 对象的本质特征，把它从所有其他种类的对象区分出来，这样，相对于查看者的视角，可以提供清晰定义的概念边界，聚焦于对象本质特征的过程。抽象是对象模型的基础元素之一。

access control（访问控制） 控制访问模型元素，例如，包的内容或类的结构或行为的机制。参见 visibility（可见性）。

action node（活动节点） 活动包含的三种节点类型之一，它定义活动的行为步骤。UML 2 定义所有可得类型的动作，包括那些引发活动或操作的行为的动作。

active object（主动对象） 拥有自己的控制线程的对象。

activity（活动） 包含活动节点、控制节点和对象节点的行为规格。

actor（执行者） 执行者定义外部实体在和系统交互时所扮演的角色。

aggregation（聚合） 一个对象由一个或更多其他对象组成的整体/部分关系，每个组成对象被认为是整体的一部分。这种关系是包含的弱形式，整体和部分的生命期是独立的。

algorithmic decomposition（算法分解） 把系统分解成部分的过程，其中每一个代表一些大过程中的小步骤。应用结构化设计方法导致算法分解，它的焦点是系统内的控制流。

architectural mechanism（架构机制） 与基本系统功能交互或支持基本系统功能的通用系统能力的表达。

architecture（架构） 系统的组件及其关系的逻辑和物理结构，由开发期间应用的所有战略和战术的设计决策炼成。

association（关联） 说明两个类之间的语义连接的关系。

attribute（属性） 类的一部分，它的值形成了类的状态定义。类的属性共同构造了它的结构。

B

base class（基类） 类结构中最泛化的类，大多数应用有很多这样的根类。一些语言定义一个原生的基类，作为所有类的终极超类。

behavior（行为） 对象如何行动和反应，表现为状态改变和消息传送。它是对象的外部可见和可测试的活动。

behavioral prototype（行为原型） 探索系统的一些元素的例子，例如，架构的一个方面、一个新的算法、一个用户界面模型或一个数据库纲要。它的目标是快速探索设计备选方案，这样，风险的区域可以早点被解决，而不会危及生产的发布版本。

binding（绑定） 表示类和一个名称（如变量声明）的关联。

C

cardinality（基数） 类可能有的实例数目，参与类关系的实例数目。

class（类） 一个共享相同结构和行为的对象集合。术语类和类型通常（但不总是）是可互换的。类在概念上和类型稍有不同，前者强调结构和行为的分类。

class diagram（类图） 面向对象设计表示法的一部分，用于在系统的逻辑设计中展示类和它们之间的关系的存在。一张类图可能代表系统的所有或部分类结构。

class structure（类结构） 一张图，其顶点代表类，弧代表这些类之间的关系。系统的类结构由一组类图代表。

client（客户） 使用另一个对象的服务的对象，可以是在其上操作，也可以是引用它的状态。

collaboration（协作） 若干模型元素共同操作来提供一些高层次行为的过程。

component（组件） 类的逻辑集合，这些类协作来提供一套服务，服务通过组件所提供的接口来提供。组件请求的服务通过需求接口来要求。组件也可以由其他组件组成，如果需要，可以嵌套到任何一个级别。

composition（组合） 一个对象由一个或更多其他对象组成的整体/部分关系，每一个被看作整体的一个部分。这种关系是聚合的强形式，整体和部分的生命期是互相依赖的。

concrete class（具体类） 实现完整因此可以有实例的类。

concurrency（并发） 把主动对象和非主动对象区分开的特征。

concurrent object（并发对象） 一个主动对象。在存在多线程控制的情况下，它的语义得到保证。

constraint（约束） 必须保持的一些语义条件的表达。

constructor（构造器） 创建对象和/或初始化它的状态的操作。

control node（控制节点） 活动包含的三种节点类型之一。它提供开始、停止和活动顺序执行的动作。

container class（容器类） 一个类，它的实例是其他对象的集合。容器类可以表示同构集合

（集合中的所有对象属于同一个类）或异构集合（集合中的对象可能属于不同的类，虽然它们都共享一个共同的超类）。容器类最经常被定义为参数化类，用一些参数指定被包含对象的类。

CRC cards（CRC 卡） 类/责任/协作者，一个关于系统中关键抽象和机制简单头脑风暴工具。

D

data dictionary（数据字典） 列举系统中所有类的全面的仓储。

delegation（委托） 一个对象的动作转发一个操作到另一个对象，代表第一个对象执行。

destructor（析构器） 释放对象状态和/或销毁对象自身的操作。

device（设备） 没有计算资源的硬件。

dynamic binding（动态绑定） 不进行名称/类关联的绑定，直到执行时名称说明的对象被创建。

E

encapsulation（封装） 分隔构造它的结构和行为的抽象元素的过程。封装分离了抽象及其实现的契约接口。

event（事件） 一些可能导致系统状态改变的发生。

F

field（字段） 对象状态的部件的仓储，对象的字段共同构成了它的结构。术语字段、实例变量、成员对象和槽是可互换的。

forward engineering（正向工程） 从逻辑或物理模型生成可执行代码。

framework（框架） 为特定领域提供一套服务的类集合。这样，框架导出很多个别类和机制，客户可以使用或者适配它们。

free subprogram（自由子程序） 作为一个对象或者相同或不同类的对象上的非原生操作的过程或函数。自由子程序是不属于对象的方法的任何子程序。

friend（友元） 一个类或操作，它的实现可能引用另一个类的私有部分，该类可以单独扩展，提供朋友关系。

function（函数） 作为一些对象的行为结果的输入/输出映射。

function point（功能点） 在需求分析语境下，单个外部可见和可测试的活动。

G

generic function（泛型函数） 对象的一个操作。类的泛型函数可以在子类中被重定义，因此对于给定对象，它通过在因继承层次而相关的各个类中声明的一个方法集来实现。术语泛型函数和虚函数通常是可互换的。

guard（警戒） 应用于一个事件的布尔表达式。如果为真，表达式允许事件导致系统的状态改变。

H

hierarchy（层次） 抽象的排行或排序。复杂系统中两个最常用的层次包括它的类结构（包括“是一个”层次）和它的对象结构（包括“……的一部分”层次）。层次也可以在复杂系统的组件和部署架构中被发现。

I

identity（标识） 将对象和所有其他对象区分开的本质。

implementation（实现） 类或对象的内部视图，包括其行为的秘密。

information hiding（信息隐藏） 隐藏无助于对象本质特征的所有秘密的过程。通常，对象的结构和它的方法实现是被隐藏的。

inheritance（继承） 类之间的关系，在这种关系中，一个类共享在一个（单继承）或多个（多继承）其他类定义的结构或行为。继承定义一个类之间的“是一个”层次，子类从一个或更多泛化的超类继承。子类通常通过增加或重定义已有的结构和行为特化它的超类。

instance（实例） 可以对其做事情的一些东西。一个实例有状态、行为和标识。相似实例的结构和行为在它们共同的类中被定义。术语实例和对象是可互换的。

instance variable（实例变量） 对象状态的部件的仓储。对象的实例变量共同构成了它的结构。

instantiation（实例化） 填充模板类或参数化类，产生一个可以创建实例的类的过程。

interface（接口） 例如类、对象、组件或组合结构的外部视图，强调它隐藏结构和行为的秘密的抽象。

invariant（不变式） 一些必须保持为真的条件的布尔表达式。

iterator（迭代器） 允许对象的部分被访问的操作。

K

key abstraction（关键抽象） 形成问题域词汇表的一部分的类或对象。

L

layer（层） 位于同一级别的抽象的组件集合。

level of abstraction（抽象级别） 类结构、对象结构、组件架构或部署架构中抽象的相对排行。以“……的一部分”的层次形式，如果一个给定抽象建于其他抽象之上，比起其他抽象来，它是更高级别的抽象；以“是一个”的层次形式，高级别抽象是泛化的，低级别抽象是特化的。

link（链接） 在两个对象之间，一个关联的实例。

M

mechanism（机制） 一种结构，在那里，对象协作起来提供一些行为，以满足问题的需求。

member function（成员函数） 对象上的操作，作为类的声明的一部分来定义；所有成员函数都是操作，但不是所有操作都是成员函数。术语成员函数和方法通常是可互换的。在一些语言中，成员函数单独存在，并可以被子类重定义；而在其他语言中，成员函数不可以被重定义，而是作为泛型函数或虚函数（两者都可以在子类中被重定义）实现的一部分。

member object（成员对象） 对象状态的部件的仓储。对象的成员对象共同构成了它的结构。术语字段、实例变量、成员对象和槽是可互换的。

message（消息） 一个对象在另一个对象上执行的操作。术语消息、方法和操作通常是可互换的。

metaclass（元类） 类的类。实例是类本身的类。

method（方法） 对象上的操作，作为类的声明的一部分来定义；所有方法都是操作，但不是所有操作都是方法。术语消息、方法和操作通常是可互换的。在一些语言中，方法单独存在并可以被子类重定义；而在其他语言中，方法不可以被重定义，而是作为泛型函数或虚函数（两者都可以在子类中被重定义）实现的一部分。

modularity（模块化） 系统被分解成一系列内聚和松耦合的组件的特征。

monomorphism（单态） 类型理论中的概念，一个名称（如一个变量声明）只表示同一个类的对象。

O

object（对象） 可以对其做事情的一些东西。一个对象有状态、行为和标识。相似对象的结构和行为在它们共同的类中被定义。术语实例和对象是可互换的。

object diagram（对象图） 面向对象设计表示法的一部分，用于在系统的逻辑设计中展示对象和它们之间的关系的存在。一张对象图可能代表系统的所有或部分对象结构，并主要阐明逻辑设计中的机制的语义。单张对象图代表某一时刻瞬时事件或对象配置的快照。

object model（对象模型） 形成面向对象设计基础的原则集，强调抽象、封装、模块化、层次、类型化、并发和持久化原则的软件工程范型。

object node（对象节点） 活动包含的三种节点类型之一。它定义在活动的动作之间流动的数据。

object structure（对象结构） 一张图，图的顶点代表对象，弧代表对象之间的关系。系统的对象结构由一系列对象图代表。

object-based programming（基于对象的编程） 一种编程方法。程序被组织为相互协作的对象集合，每一个对象代表一些类型的一个实例；它的类型是类型层次的所有成员，该类型层次通过继承关系以外的关系来联合。在这样的程序中，类型一般被看作静态的，而对象通常有动态得多的本质，或多或少被静态绑定和单态的存在所约束。

object-oriented analysis（面向对象分析） 一种分析方法。从类和对象的视角研究需求，类和对象可以从问题域词汇表中被发现。

object-oriented decomposition（面向对象分解） 把系统分解为部分的过程，每一个部分代表来自问题域的一些类或对象。应用面向对象设计方法导致面向对象分解，我们把世界看

作对象的集合，对象与另一个对象共同操作，以达到一些所要的功能。

object-oriented design（面向对象设计） 一种设计方法，包括面向对象分解的过程以及描绘逻辑和物理上当前设计系统的静态和动态模型的表示法。具体地，这个表示法的例子包括类图、对象图、组件图和部署图。

object-oriented programming（面向对象编程） 一种编程方法。程序被组织为相互协作的对象集合，每一个代表一些类型的一个实例，它的类型是类型层次的所有成员，该类型层次通过继承关系来联合。在这样的程序中，类一般被看作静态的，而对象通常有动态得多的本质，动态绑定和多态的存在加剧了这种动态。

operation（操作） 为了引起反应，一个对象在另一个对象上执行的一些工作。一个特定对象上的所有操作可以是自由子程序和成员函数或方法。术语消息、方法和操作通常是可互换的。

P

parameterized class（参数化类） 作为其他类模板的类，模板可以被其他类、对象和/或操作参数化。在可以创建实例之前，参数化类必须被实例化（填入它的参数）。参数化类通常用作容器类。术语模板类和参数化类是可互换的。

partition（分区） 形成给定级别抽象的一部分的组件或子系统，也是活动图的结构划分，通常叫做泳道。

passive object（被动对象） 没有包含自己的控制线程的对象。

persistence（持久化） 对象的一个特征。这个特征的存在使得对象超越了时间（即，在它的创建者停止存在之后，对象继续存在）和/或空间（即，对象的位置从被它创建的地址空间移走）。

polymorphism（多态） 类型理论中的一个概念，一个名称（如一个变量声明）可以表示很多不同类的对象，这些类和一些共同超类相关。因此，这个名称表示的任何对象可以以不同的方式响应一些共同的操作集合。

postcondition（后置条件） 由一个操作满足的不变式。

precondition（前置条件） 由一个操作假设的不变式。

private（私有） 形成类或对象的接口的一部分的声明。声明为私有意味着对其他任何类或对象不可见。

process（进程） 单个控制线程的激活。

processor（处理器） 有计算资源的硬件。

protected（保护） 形成类或对象的接口的一部分的声明。声明为保护意味着对除了子类之外的其他任何类或对象不可见。

protocol（协议） 对象可能动作或反应的方式，构成对象的完整静态和动态外部视图。对象的协议定义了对象可允许行为的封皮。

public（公开） 形成类或对象的接口的一部分的声明。声明为公开意味着对所有对它有可见性的其他类或对象可见。

Q

qualifier（限定符） 一个属性，它的值唯一地标识单个目标对象。

R

reactive system（反应性系统） 一个事件驱动的系统。反应性系统的行为不是简单的输入/输出映射。

real-time system（实时系统） 本质的过程必须符合某些关键的时间限制的系统。硬实时系统必须是确定的，错过最后期限可能会导致灾难性的结果。

reference architecture（参考架构） 一个预定义的架构模式或模式集合，可能部分或完全地为在特定业务和技术语境以及使它们能够使用的支持工件中使用而被实例化、设计和证明。

responsibility（责任） 一个对象持有的一些负责任的行为。责任表示强制对象提供某个行为。

reverse engineering（逆向工程） 从可执行代码产出逻辑或物理模型。

role（角色） 一个类或对象参与与另一个类和对象的关系时的目标或能力。对象的角色表示在某个时间点定义好的行为集合选择，角色是对象在给定时刻展现给世界的面貌。

S

scenario（场景） 引起一些系统行为的事件概述。

sequential object（顺序对象） 只有在单个控制线程的存在下语义才能保证的被动对象。

server（服务器） 不操作其他对象只被其他对象操作的对象，提供某种服务的对象。

service（服务） 由系统给定部分提供的行为。

signature（签名） 一个操作的形式参数和返回类型的完整剖面。

slot（槽） 对象状态的部件的仓储。对象的槽共同构成了它的结构。术语字段、实例变量、成员对象和槽是可互换的。

state（状态） 一个对象行为的累积结果，一个对象可以存在的可能条件之一，由区别于其他数量的限定数量刻画。在给定的时间点，一个对象状态包含所有（通常是静态的）对象属性加上每个这些属性的当前（通常是动态的）值。

state machine diagram（状态机图） 面向对象设计表示法的一部分，用于展示给定类的状态空间、导致从一个状态转换到另一个的事件和状态改变导致的动作。

state space（状态空间） 对象所有可能的状态的列举。对象的状态空间包含无限的状态，但只有有限数量是可能的（虽然不总是想要或期望的）状态。

static binding（静态绑定） 在名称被声明时（编译时），创建名称指代的对象之前，做名称/类关联的绑定。

strategic design decision（战略设计决策） 影响遍及架构的设计决策。

strongly typed（强类型化） 编程语言的一个特征，保证所有表达式类型一致。

structure（结构） 对象状态的具体表现。一个对象不和其他任何对象共享它的状态，虽然

同一个类的所有对象确实共享它们状态的同一个表现。

structured design（结构化设计） 一种设计方法，包含算法分解的过程。

subclass（子类） 继承自一个或更多类（称为直接超类）的类。

subsystem（子系统） 组件的集合，其中一些对其他子系统可见，另一些是隐藏的。

superclass（超类） 被另一个类（称为它的直接子类）继承的类。

synchronization（同步） 操作的并发语义。一个操作可能是简单（只涉及一个控制线程）、同步（两个进程交会）、阻断（仅当第二个进程已经在等待时，一个进程可能和另一个交会）、超时（一个进程可以和另一个交会，但会只等待第二个进程指定的时间）或异步（两个进程独立操作）的。

T

tactical design decision（战术设计决策） 影响局部架构的设计决策。

template class（模板类） 作为其他类的模板的类，模板可以被其他类、对象和/或操作参数化。在对象被创建之前，模板类必须被实例化（填入它的参数）。模板类通常用作容器类。术语模板类和参数化类是可互换的。

thread of control（控制线程） 单个进程。控制线程的开始是一个系统内所发生的独立动态动作的根。一个给定系统可能同时有很多控制线程，其中一些可以动态地存在和停止存在。跨越多个处理器执行的系统允许真正并发的控制线程，运行在单个处理器上的系统只能营造并发控制线程的错觉。

transition（转换） 从一个状态到另一个状态。

type（类型） 定义一个对象可以具有的允许值的域和可以在对象上执行的操作。术语类和类型通常（但不总是）是可互换的。类型是稍微不同于类的概念，前者强调遵从共同协议的重要性。

typing（类型化） 强制一个对象的类，以防止不同类型的对象被互换，最多允许它们只能以很严格的方式互换。

U

Unified Modeling Language，UML（统一建模语言） 一种语言（表示法），用来实施面向对象分析设计，以建模开发关注的方面。

V

virtual function（虚函数） 对象的一个操作，虚函数可以被子类重定义。因此，对于给定对象，它实现在各个类中声明的方法集合，这些类通过它们的继承层次相关。术语泛型函数和虚函数通常是可互换的。

visibility（可见性） 一个抽象看见另一个并引用它的外部视图中的资源的能力。仅当它们的范围重叠时，抽象对另一个抽象可见。导出控制可以进一步限制对可见抽象的访问。可见性的例子包括 public、private、protected 和 package。

分类书目

分类书目被划分成若干部分，标签为 A～M。附录 B 的参考文献引用条目以[<标签><年>]的方式出现在分类书目中。例如，Brooks [H 1975]是指他 1975 年的书 *The Mythical Man-Month* 书目的 H 部分（软件工程）。

A. 分类

Allen, T., and Starr, T. 1982. *Hierarchy: Perspectives for Ecological Complexity*. Chicago, IL: The University of Chicago Press.

Aquinas, T. *Summa Theologica*. Vol. 19 of Great Books of the Western World. Chicago, IL: Encyclopedia Britannica.

Aristotle. *Categories*. Vol. 8 of Great Books of the Western World. Chicago, IL: Encyclopedia Britannica.

Bateson, G. 1979. *Mind and Nature: A Necessary Unity*. New York, NY: Bantam Books.

Brachman, R., McGuinness, D., Patel-Schneider, P., and Resnick, L. *Living with Classic. Principles of Semantic Networks*. San Mateo, CA: Morgan Kaufman Publishers.

Bulman, D. January 1991. Refining Candidate Objects. *Computer Language* vol. 8(1).

Cant, S., Jeffery, D., and Henderson-Sellers, B. October 1991. *A Conceptual Model of Cognitive Complexity of Elements of the Programming Process*. New South Wales, Australia: University of New South Wales.

Classification Society of North America. *Journal of Classification*. New York, NY: Springer-Verlag.

Coad, P. September 1992. Object-Oriented Patterns. *Communications of the ACM* vol. 35(9).

———. 1993. *The Object Game*. Austin, TX: Object International.

Coombs, C., Raiffa, H., and Thrall, R. 1954. Some Views on Mathematical Models and Measurement Theory. *Psychological Review* vol. 61(2).

Courtois, P. June 1985. On Time and Space Decomposition of Complex Structures. *Communications of the ACM* vol. 28(6).

Cunningham, W., and Beck, K. July/August 1989. Constructing Abstractions for Object-Oriented Abstractions. *Journal of Object-Oriented Programming* vol. 2(2).

Darwin, C. *The Origin of Species*. Vol. 49 of Great Books of the Western World. Chicago, IL: Encyclopedia Britannica.

Descartes, R. *Rules for the Direction of the Mind*. Vol. 31 of Great Books of the Western World. Chicago, IL: Encyclopedia Britannica.

Flood, R., and Carson, E. 1988. *Dealing with Complexity*. New York, NY: Plenum Press.

Gould, S. June 1992. We Are All Monkey's Uncles. *Natural History*.

Johnson, R. 1992. *Documenting Frameworks Using Patterns*. Vancouver, Canada: OOPSLA'92.

Lakoff, G. 1987. *Women, Fire, and Dangerous Things: What Categories Reveal about the Mind*. Chicago, IL: The University of Chicago Press.

Lefrancois, G. 1977. *Of Children: An Introduction to Child Development*. Second Edition. Belmont, CA: Wadsworth.

Lewin, R. November 4, 1988. Family Relationships Are a Biological Conundrum. *Science* vol. 242.

Maccoby, M. December 1991. The Innovative Mind at Work. *IEEE Spectrum* vol. 28(12).

Maier, H. 1969. *Three Theories of Child Development: The Contributions of Erik H. Erickson, Jean Piaget, and Robert R. Sears, and Their Applications*. New York, NY: Harper and Row.

May, R. September 16, 1988. How Many Species Are There on Earth? *Science* vol. 241.

Michalski, R., and Stepp, R. 1983. Learning from Observation: Conceptual Clustering. In *Machine Learning: An Artificial Intelligence Approach*, ed. R. Michalski, J. Carbonell, and T. Mitchell. Palo Alto, CA: Tioga.

Miller, G. March 1956. The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information. *The Psychological Review* vol. 63(2).

Minsky, M. April 1970. Form and Content in Computer Science. *Journal of the ACM* vol. 17(2).

———. 1986. *The Society of Mind*. New York, NY: Simon and Schuster.

Moldovan, D., and Wu, C. December 1988. A Hierarchical Knowledge-Based System for Airplane Classification. *IEEE Transactions on Software Engineering* vol. 14(12).

Newell, A. 1990. *Unified Theories of Cognition*. Cambridge, MA: Harvard University Press.

Newell, A., and Simon, H. 1972. *Human Problem Solving*. Englewood Cliffs, NJ: Prentice Hall.

Papert, S. 1980. *Mindstorms: Children, Computers, and Powerful Ideas*. New York, NY: Basic Books.

Plato. *Statesman*. Vol. 7 of Great Books of the Western World. Chicago, IL: Encyclopedia Britannica.

Prieto-Diaz, R., and Arango, G. 1991. *Domain Analysis and Software Systems Modeling*. Las

Alamitos, CA: Computer Society Press of the IEEE.

Shaw, M. 1989. Larger Scale Systems Require Higher-Level Abstractions. *Proceedings of the Fifth International Workshop on Software Specification and Design*. IEEE Computer Society.

———. 1990. Elements of a Design Language for Software Architecture. Pittsburgh, PA: Carnegie Mellon University.

———. 1991. *Heterogeneous Design Idioms for Software Architecture*. Pittsburgh, PA: Carnegie Mellon University.

Siegler, R., and Richards, D. 1982. The Development of Intelligence. In *Handbook of Human Intelligence*, ed. R. Sternberg. Cambridge, United Kingdom: Cambridge University Press.

Simon, H. 1962. The Architecture of Complexity. *Proceedings of the American Philosophical Society* vol. 106.

———. 1982. *The Sciences of the Artificial*. Cambridge, MA: The MIT Press. Sowa, J. 1984. *Conceptual Structures: Information Processing in Mind and Machine*. Reading, MA: Addison-Wesley.

———. 1991. *Principles of Semantic Networks*. San Mateo, CA: Morgan Kaufman Publishers.

Stepp, R., and Michalski, R. 1986. Conceptual Clustering of Structured Objects: A Goal-Oriented Approach. *Artificial Intelligence* vol. 28(1).

Stevens, S. June 1946. On the Theory of Scales of Measurement. *Science* vol. 103(2684).

Stillings, N., Feinstein, M., Garfield, J., Rissland, E., Rosenbaum, D., Weisler, S., and Baker-Ward, L. 1987. *Cognitive Science: An Introduction*. Cambridge, MA: The MIT Press.

Waldrop, M. 1992. *Complexity: The Emerging Science at the Edge of Order and Chaos*. New York, NY: Simon and Schuster.

B. 面向对象分析

Arango, G. May 1989. Domain Analysis: From Art Form to Engineering Discipline. *SIGSOFT Engineering Notes* vol. 14(3).

Bailin, S. 1988. *Remarks on Object-Oriented Requirements Specification*. Laurel, MD: Computer Technology Associates.

Bailin, S., and Moore, J. 1987. *An Object-Oriented Specification Method for Ada*. Laurel, MD: Computer Technology Associates.

Barbier, F. May 1992. Object-Oriented Analysis of Systems through Their Dynamical Aspects. *Journal of Object-Oriented Programming* vol. 5(2).

Borgida, A., Mylogoulos, J., and Wong, H. 1984. Generalization/Specialization as a Basis for Software Specification. In *On Conceptual Modeling: Perspectives from Artificial Intelligence, Databases, and Programming Languages*, ed. M. Brodie, J. Mylopoulos, and J. Schmidt. New York, NY: Springer-Verlag.

Cernosek, G., Monterio, E., and Pribyl, W. 1987. *An Entity-Relationship Approach to Software Requirements Analysis for Object-Based Development*. Houston, TX: McDonnell Douglas Astronautics.

Coad, P. Summer 1989. OOA: Object-Oriented Analysis. *American Programmer* vol. 2(7–8).

———. April 1990. *New Advances in Object-Oriented Analysis*. Austin, TX: Object International.

Coad, P., and Yourdon, E. 1991. *Object-Oriented Analysis*. Second Edition. Englewood Cliffs, NJ: Yourdon Press.

Dahl, O-J. 1987. Object-Oriented Specifications. In *Research Directions in Object-Oriented Programming*, ed. B. Schriver and P. Wegner. Cambridge, MA: The MIT Press.

deChampeaux, D. April 1991a. *A Comparative Study of Object-Oriented Analysis Methods*. Palo Alto, CA: Hewlett-Packard Laboratories.

———. April 1991b. *Object-Oriented Analysis and Top-Down Software Development*. Palo Alto, CA: Hewlett Packard Laboratories.

DeMarco, T. 1979. *Structured Analysis and System Specification*. Englewood Cliffs, NJ: Prentice Hall.

Embley, D., Kurtz, B., and Woodfield, S. 1992. *Object-Oriented Systems Analysis: A Model-Driven Approach*. Englewood Cliffs, NJ: Yourdon Press.

EVB Software Engineering. 1989. *Object-Oriented Requirements Analysis*. Frederick, MD.

Gane, C., and Sarson, T. 1979. *Structured Systems Analysis*. Englewood Cliffs, NJ: Prentice Hall.

Hatley, D., and Pirbhai, I. 1988. *Strategies for Real-Time System Specification*. New York, NY: Dorset House.

Ho, D., and Parry, T. July 1991. *The Hewlett-Packard Method of Object-Oriented Analysis*. Palo Alto, CA: Hewlett-Packard Laboratories.

Iscoe, N. 1988. *Domain Models for Program Specification and Generation*. Austin, TX: University of Texas.

Iscoe, N., Browne, J., and Werth, J. 1989. *Modeling Domain Knowledge: An Object-Oriented Approach to Program Specification and Generation*. Austin, TX: The University of Texas.

Lang, N. January 1993. Shlaer-Mellor Object-Oriented Analysis Rules. *Software Engineering Notes* vol. 18(1).

Marca, D., and McGowan, C. 1988. *SADT™: Structured Analysis and Design Technique*. New York, NY: McGraw-Hill.

Martin, J., and Odell, J. 1992. *Object-Oriented Analysis and Design*. Englewood Cliffs, NJ: Prentice Hall.

McMenamin, S., and Palmer, J. 1984. *Essential Systems Analysis*. New York, NY: Yourdon Press.

Mellor, S., Hecht, A., Tryon, D., and Hywari, W. September 1988. *Object-Oriented Analysis: Theory and Practice, Course Notes*. San Diego, CA: OOPSLA'88.

Moore, J., and Bailin, S. 1988. *Position Paper on Domain Analysis*. Laurel, MD: Computer Technology Associates.

Page-Jones, M., and Weiss, S. Summer 1989. Synthesis: An Object-Oriented Analysis and Design Method. *American Programmer* vol. 2(7–8).

Rubin, K., and Goldberg, A. September 1992. Object Behavior Analysis. *Communications of the ACM* vol. 35(9).

Saeki, M., Horai, H., and Enomoto, H. May 1989. Software Development Process from Natural Language Specification. *Proceedings of the 11th International Conference on Software Engineering*. New York, NY: Computer Society Press of the IEEE.

Shemer, I. June 1987. Systems Analysis: A Systemic Analysis of a Conceptual Model. *Communications of the ACM* vol. 30(6).

Shlaer, S., and Mellor, S. 1988. *Object-Oriented Systems Analysis: Modeling the World in Data*. Englewood Cliffs, NJ: Yourdon Press.

———. July 1989. An Object-Oriented Approach to Domain Analysis. *Software Engineering Notes* vol. 14(5).

———. Summer 1989. Understanding Object-Oriented Analysis. *American Programmer* vol. 2(7–8).

———. 1992. *Object Lifecycles: Modeling the World in States*. Englewood Cliffs, NJ: Yourdon Press.

Stoecklin, S., Adams, E., and Smith, S. 1987. *Object-Oriented Analysis*. Tallahassee, FL: East Tennessee State University.

Sully, P. Summer 1989. Structured Analysis: Scaffolding for Object-Oriented Development. *American Programmer* vol. 2(7–8).

Tsai, J., and Ridge, J. November 1988. Intelligent Support for Specifications Transformation. *IEEE Software* vol. 5(6).

Veryard, R. 1984. *Pragmatic Data Analysis*. Oxford, England: Blackwell Scientific Publications.

Ward, P. March 1989. How to Integrate Object Orientation with Structured Analysis and Design. *IEEE Software* vol. 6(2).

Weinberg, G. 1988. *Rethinking Systems Analysis and Design*. New York, NY: Dorset House.

C. 面向对象应用

Abdali, K., Cherry, G., and Soiffer, N. November 1986. A Smalltalk System for Algebraic Manipulation. *SIGPLAN Notices* vol. 21(11).

Abdel-Hamid, T., and Madnick, S. December 1989. Lessons Learned from Modeling the Dynamics of Software Development. *Communications of the ACM* vol. 32(12).

The Aerospace Corporation. 2003. *GPS Primer—A Student Guide to the Global Positioning System*.

www.aero.org/education/primers/gps/GPS-Primer.pdf.

Almes, G., and Holman, C. September 1987. Edmas: An Object-Oriented, Locally Distributed Mail System. *IEEE Transactions on Software Engineering* vol. SE-13(9).

Anderson, D. November 1986. Experience with Flamingo: A Distributed, Object-Oriented User Interface System. *SIGPLAN Notices* vol. 21(11).

Archer, J., and Devlin, M. 1987. *Rational's Experience Using Ada for Very Large Systems*. Mountain View, CA: Rational.

Bagrodia, R., Chandy, M., and Misra, J. June 1987. A Message-Based Approach to Discrete-Event Simulation. *IEEE Transactions on Software Engineering* vol. SE-13(6).

Barry, B. October 1989. Prototyping a Real-Time Embedded System in Smalltalk. *SIGPLAN Notices* vol. 24(10).

Barry, B., Altoft, J., Thomas, D., and Wilson, M. October 1987. Using Objects to Design and Build Radar ESM Systems. *SIGPLAN Notices* vol. 22(12).

Basili, V., Caldiera, G., and Cantone, G. January 1992. A Reference Architecture for the Component Factory. *ACM Transactions on Software Engineering and Methodology* vol. 1(1).

Batory, D., and O'Malley, S. October 1992. The Design and Implementation of Hierarchical Software Systems with Reusable Components. *ACM Transactions on Software Engineering and Methodology* vol. 1(4).

Bezivin, J. October 1987. Some Experiments in Object-Oriented Simulation. *SIGPLAN Notices* vol. 22(12).

Bhaskar, K., and Peckol, J. November 1986. Virtual Instruments: Object-Oriented Program Synthesis. *SIGPLAN Notices* vol. 21(11).

Bihair, T., and Gopinath, P. December 1992. Object-Oriented Real-Time Systems: Concepts and Examples. *IEEE Computer* vol. 25(12).

Bjornerstedt, A., and Britts, S. September 1988. AVANCE: An Object Management System. *SIGPLAN Notices* vol. 23(11).

Bobrow, D., and Stefik, M. February 1986. Perspectives on Artificial Intelligence Programming. *Science* vol. 231.

Boltuck-Pasquier, J., Grossman, E., and Collaud, G. August 1988. Prototyping an Interactive Electronic Book System Using an Object-Oriented Approach. *Proceedings of ECOOP'88: European Conference on Object-Oriented Programming*. New York, NY: Springer-Verlag.

Bonar, J., Cunningham R., and Schultz, J. November 1986. An Object-Oriented Architecture of Intelligent Tutoring Systems. *SIGPLAN Notices* vol. 21(11).

Booch, G. 1987. *Software Components with Ada: Structures, Tools, and Subsystems*. Menlo Park, CA: Benjamin/Cummings.

Borning, A. October 1981. The Programming Language Aspects of ThingLab, a Constraint-

Oriented Simulation Laboratory. *ACM Transactions on Programming Languages and Systems* vol. 3(4).

Bowman, W., and Flegal, B. August 1981. ToolBox: A Smalltalk Illustration System. *Byte* vol. 6(8).

Britcher, R., and Craig, J. May 1986. Using Modern Design Practices to Upgrade Aging Software Systems. *IEEE Software* vol. 3(3).

Britton, K., and Parnas, D. December 8, 1981. *A-7E Software Module Guide*. Report 4702. Washington, DC: Naval Research Laboratory.

Brooks, R. 1987. *A Hardware Retargetable Distributed Layered Architecture for Mobile Robot Control*. Cambridge, MA: MIT Artificial Intelligence Laboratory.

Brooks, R., and Flynn, A. June 1989. *Fast, Cheap, and Out of Control: A Robot Invasion of the Solar System*. Cambridge, MA: MIT Artificial Intelligence Laboratory.

Bruck, D. 1988. Modeling of Control Systems with C++ and PHIGS. *Proceedings of USENIX C++ Conference*. Berkeley, CA: USENIX Association.

Budd, T. January 1989. The Design of an Object-Oriented Command Interpreter. *Software—Practice and Experience* vol. 19(1).

C++ Booch Components Class Catalog. 1992. Santa Clara, CA: Rational.

Call, L., Cohrs, D., and Miller, B. October 1987. CLAM—An Open System for Graphical User Interfaces. *SIGPLAN Notices* vol. 22(12).

Campbell, R., Islam, N., and Madany, P. 1992. *The Design of an Object-Oriented Operating System: A Case Study of Choices*. Vancouver, Canada: OOPSLA'92.

Caplinger, M. October 1987. An Information System Based on Distributed Objects. *SIGPLAN Notices* vol. 22(12).

Cargill, T. November 1986. Pi: A Case Study in Object-Oriented Programming. *SIGPLAN Notices* vol. 21(11).

Carroll, M. September 1990. *Building Reusable C++ Components*. Murray Hills, NJ: AT&T Bell Laboratories.

Cmelik, R., and Genani, N. May 1988. Dimensional Analysis with C++. *IEEE Software* vol. 5(3).

Coggins, J. September 1990. *Design and Management of C+ Libraries*. Chapel Hill, NC: University of North Carolina.

Cointe, P., Briot, J., and Serpette, B. 1987. The Formes System: A Musical Application of Object-Oriented Concurrent Programming. In *Object-Oriented Concurrent Programming*, ed. Yonezawa and M. Tokoro. Cambridge, MA: The MIT Press.

Collins, D. 1990. *What Is an Object-Oriented User Interface?* Thornwood, NY: IBM Systems Research Education Center.

Comeau, G. March 1991. C++ in the Real World: Interviews with C++ Application Developers. *The C++ Report* vol. 3(3).

Coplien, J. September 1991. Experience with CRC Cards in AT&T. *The C++ Report* vol. 3(8).

Coutaz, J. September 1985. Abstractions for User Interface Design. *IEEE Computer* vol. 18(9).

Custer, H. 1993. *Inside Windows NT*. Redmond, WA: Microsoft Press.

Dasgupta, P. November 1986. A Probe-Based Monitoring Scheme for an Object-Oriented Operating System. *SIGPLAN Notices* vol. 21(11).

Davidson, C., and Moseley, R. 1987. *An Object-Oriented Real-Time Knowledge-Based System*. Albuquerque, NM: Applied Methods.

Davis, J., and Morgan, T. January 1993. Object-Oriented Development at Brooklyn Union Gas. *IEEE Software* vol. 10(1).

deChampeaux, D., Anderson, A., Lerman, D., Gasperina, M., Feldhousen, E., Glei, M., Fulton, F., Groh, C., Houston, D., Monroe, C., Raj, R., and Shultheis, D. October 1991. *Case Study of Object-Oriented Software Development*. Palo Alto, CA: Hewlett-Packard Laboratories.

Dietrich, W., Nackman, L., and Gracer, F. October 1989. Saving a Legacy with Objects. *SIGPLAN Notices* vol. 24(10).

Dijkstra, E. May 1968. The Structure of the "THE" Multiprogramming System. *Communications of the ACM* vol. 11(5).

Durand, G., Benkiran, A., Durel, C., Nga, H., and Tag, M. 1988. *Distributed Mail Service in CSE System*. Paris, France: Synergie Informatique et Development.

Englemore, R., and Morgan, T. 1988. *Blackboard Systems*. Wokingham, England: Addison-Wesley.

Epstein, D., and LaLonde, W. September 1988. A Smalltalk Window System Based on Constraints. *SIGPLAN Notices* vol. 23(11).

The European Commission, Directorate-General for Energy and Transportation. 2006. Galileo—The European Satellite Navigation System. http://ec.europa.eu/dgs/energy_transport/galileo/index_en.htm.

Ewing, J. November 1986. An Object-Oriented Operating System Interface. *SIGPLAN Notices* vol. 21(11).

Fenton, J., and Beck, K. October 1989. Playground: An Object-Oriented Simulation System with Agent Rules for Children of All Ages. *SIGPLAN Notices* vol. 24(10).

Fischer, G. 1987. *An Object-Oriented Construction and Tool Kit for Human-Computer Communication*. Boulder, CO: University of Colorado Department of Computer Science and Institute of Cognitive Science.

Foley, J., and van Dam, A. 1982. *Fundamentals of Interactive Computer Graphics*. Reading, MA: Addison-Wesley.

Frankowski, E. 1986. *Advantages of the Object Paradigm for Prototyping*. Golden Valley, MN: Honeywell.

Freburger, K. October 1987. RAPID: Prototyping Control Panel Interfaces. *SIGPLAN Notices* vol. 22(12).

Freitas, M., Moreira, A., and Guerreiro, P. July/August 1990. Object-Oriented Requirements Analysis in an Ada Project. *Ada Letters* vol. X(6).

Funk, D. 1986. Applying Ada to Beech Starship Avionics. *Proceedings of the First International Conference on Ada Programming Language Applications for the NASA Space Station*. Houston, TX: NASA Lyndon B. Johnson Space Center.

Garrett, N., and Smith, K. November 1986. Building a Timeline Editor from Prefab Parts: The Architecture of an Object-Oriented Application. *SIGPLAN Notices* vol. 21(11).

Goldberg, A. 1978. *Smalltalk in the Classroom*. Palo Alto, CA: Xerox Palo Alto Research Center.

Goldberg, A., and Rubin, K. October 1990. Taming Object-Oriented Technology. *Computer Language* vol. 7(10).

Goldstein, N., and Alger, J. 1992. *Developing Object-Oriented Software for the Macintosh*. Reading, MA: Addison-Wesley.

Gorlen, K. December 1987. An Object-Oriented Class Library for C++ Programs. *Software—Practice and Experience* vol. 17(12).

Gray, L. 1987. *Transferring Object-Oriented Design Techniques into Use: AWIS Experience*. Fairfax, VA: TRW Federal Systems Group.

Grimshaw, A., and Liu, J. October 1987. Mentat: An Object-Oriented Macro Data Flow System. *SIGPLAN Notices* vol. 22(12).

Grossman, M., and Ege, R. October 1987. Logical Composition of Object-Oriented Interfaces. *SIGPLAN Notices* vol. 22(12).

Gutfreund, S. October 1987. ManiplIcons in ThinkerToy. *SIGPLAN Notices* vol. 22(12).

Gwinn, J. February 1992. Object-Oriented Programs in Realtime. *SIGPLAN Notices* vol. 27(2).

Harrison, W., Shilling, J., and Sweeney, P. October 1989. Good News, Bad News: Experience Building a Software Development Environment Using the Object-Oriented Paradigm. *SIGPLAN Notices* vol. 24(10).

Hekmatpour, A., Orailoglu, A., and Chau, P. April 1991. Hierarchical Modeling of the VLSI Design Process. *IEEE Expert* vol. 6(2).

Hollowell, G. November 1991. Leading the U.S. Semiconductor Manufacturing Industry Toward an Object-Oriented Technology Standard. *Hotline on Object-Oriented Technology* vol. 3(1).

Ingalls, D., Wallace, S., Chow, Y., Ludolph, F., and Doyle, K. September 1988. Fabrik: A Visual Programming Environment. *SIGPLAN Notices* vol. 23(11).

Jacky, J., and Kalet, I. November 1986. An Object-Oriented Approach to a Large Scientific Application. *SIGPLAN Notices* vol. 21(11).

Jacobson, I. January 1993. Is Object Technology Software's Industrial Platform? *IEEE Software* vol. 10(1).

Jerrell, M. October 1989. Function Minimization and Automatic Differentiation Using C++.

SIGPLAN Notices vol. 24(10).

Johnson, R., and Foote, B. June/July 1988. Designing Reusable Classes. *Journal of Object-Oriented Programming* vol. 1(2).

Jones, M., and Rashid, R. November 1986. Mach and Matchmaker: Kernel and Language Support for Object-Oriented Distributed Systems. *SIGPLAN Notices* vol. 21(11).

Jurgen, R. May 1991. Smart Cars and Highways Go Global. *IEEE Spectrum* vol. 28(5).

Kamath, Y., and Smith, J. November/December 1992. Experiences in C++ and O-O Design. *Journal of Object-Oriented Programming* vol. 5(7).

Kay, A., and Goldberg, A. March 1977. Personal Dynamic Media. *IEEE Computer*.

Kerr, R., and Percival, D. October 1987. Use of Object-Oriented Programming in a Time Series Analysis System. *SIGPLAN Notices* vol. 22(12).

Kiyooka, G. December 1992. Object-Oriented DLLs. *Byte* vol. 17(14).

Kozaczynski, W., and Kuntzmann-Combelles, A. January 1993. What It Takes to Make O-O Work. *IEEE Software* vol. 10(1).

Krueger, C. June 1992. Software Reuse. *ACM Computing Surveys* vol. 24(2).

Kuhl, F. 1988. *Object-Oriented Design for a Workstation for Air Traffic Control*. McLean, VA: The MITRE Corporation.

LaPolla, M. 1988. *On the Classification of Object-Oriented Design: The Object-Oriented Design of the AirLand Battle Management Menu System*. Austin, TX: Lockheed Software Technology Center.

Lea, D. 1988a. *User's Guide to GNU C++ Library*. Cambridge, MA: Free Software Foundation.

———. 1988b. The GNU C++ Library. *Proceedings of USENIX C++ Conference*. Berkeley, CA: USENIX Association.

Leathers, B. July 1990. Cognos and Eiffel: A Cautionary Tale. *Hotline on Object-Oriented Technology* vol. 1(9).

Ledbetter, L., and Cox, B. June 1985. Software-ICs. *Byte* vol. 10(6).

Lee, K., and Rissman, M. February 1989. *An Object-Oriented Solution Example: A Flight Simulator Electrical System*. Pittsburgh, PA: Software Engineering Institute.

Lee, K., Rissman, M., D'Ippolito, R., Plinta, C., and Van Scoy, R. December 1987. *An OOD Paradigm for Flight Simulators*. Report CMU/SEI-87-TR-43. Pittsburgh, PA: Software Engineering Institute.

Levy, P. 1987. *Implementing Systems Software in Ada*. Mountain View, CA: Rational.

Lewis, J., Henry, S., Kafura, D., and Shulman, R. July/August 1992. On the Relationship Between the Object-Oriented Paradigm and Software Reuse: An Empirical Investigation. *Journal of Object-Oriented Programming* vol. 5(4).

Linton, M., Vlissides, J., and Calder, P. February 1989. Composing User Interfaces with InterViews.

IEEE Computer vol. 22(2).

Liu, L., and Horowitz, E. February 1989. Object Database Support for a Software Project Management Environment. *SIGPLAN Notices* vol. 24(2).

Locke, D., and Goodenough, J. 1988. *A Practical Application of the Ceiling Protocol in a Real-Time System*. Report CMU/SEI-88-SR-3. Pittsburgh, PA: Software Engineering Institute.

Love, T. 1993. *Object Lessons*. New York, NY: SIGS Publications.

Lu, Cary. December 1992. Objects for End Users. *Byte* vol. 17(14).

Madany, P., Leyens, D., Russo, V., and Campbell, R. 1988. A C++ Class Hierarchy for Building UNIX-like File Systems. *Proceedings of USENIX C++ Conference*. Berkeley, CA: USENIX Association.

Madduri, H., Raeuchle, T., and Silverman, J. 1987. *Object-Oriented Programming for Fault-Tolerant Distributed Systems*. Golden Valley, MN: Honeywell Computer Science Center.

Maloney, J., Borning, A., and Freeman-Benson, B. October 1989. Constraint Technology for User Interface Construction in ThingLab II. *SIGPLAN Notices* vol. 24(10).

McDonald, J. October 1989. Object-Oriented Programming for Linear Algebra. *SIGPLAN Notices* vol. 24(10).

Mentor's Lessons in the School of Hard Knocks. January 25, 1993. *Busines Week.*

Meyrowitz, N. November 1986. Intermedia: The Architecture and Construction of an Object-Oriented Hypermedia System and Applications Framework. *SIGPLAN Notices* vol. 21(11).

Miller, M., Cunningham, H., Lee, C., and Vegdahl, S. November 1986. The Application Accelerator Illustration System. *SIGPLAN Notices* vol. 21(11).

Mohan, L., and Kashyap, R. May 1988. An Object-Oriented Knowledge Representation for Spatial Information. *IEEE Transactions on Software Engineering* vol. 14(5).

Morgan, T., and Davis, J. March 1991. Large-Scale Object Systems Development. *Hotline on Object-Oriented Technology* vol. 2(5).

Mraz, R. December 1986. *Performance Evaluation of Parallel Branch and Bound Search with the Intel iPSE Hypercube Computer*. Wright-Patterson Air Force Base, OH: Air Force Institute of Technology.

Muller, H., Rose, J., Kempf, J., and Stansbury, T. October 1989. The Use of Multimethods and Method Combination in a CLOS-Based Window Interface. *SIGPLAN Notices* vol. 24(10).

Murphy, E. December 1988. All Aboard for Solid State. *IEEE Spectrum* vol. 25(13).

Nerson, J. September 1992. Applying Object-Oriented Analysis and Design. *Communications of the ACM* vol. 35(9).

NeXT Embraces a New Way of Programming. November 25, 1988. *Science* vol. 242.

Orden, E. 1987. Application Talk. *HOOPLA: Hooray for Object-Oriented Programming Languages* vol. 1(1). Everette, WA: Object-Oriented Programming for Smalltalk Application Developers

Association.

Orfali, R., and Harkey, D. 1992. *Client/Server Programming with OS/2*. New York, NY: Van Nostrand Reinhold.

Oshima, M., and Shirai, Y. July 1983. Object Recognition Using Three-Dimensional Information. *IEEE Transactions on Pattern Analysis and Machine Intelligence* vol. 5(4).

Page, T., Berson, S., Cheng, W., and Muntz, R. October 1989. An Object-Oriented Modeling Environment. *SIGPLAN Notices* vol. 24(10).

Pashtan, A. 1982. Object-Oriented Operating Systems: An Emerging Design Methodology. *Proceedings of the ACM '82 Conference*. New York, NY: Association of Computing Machinery.

Piersol, K. November 1986. Object-Oriented Spreadsheets: The Analytic Spreadsheet Package. *SIGPLAN Notices* vol. 21(11).

Pinson, L., and Wiener, R. 1990. *Applications of Object-Oriented Programming*. Reading, MA: Addison-Wesley.

Pittman, M. January 1993. Lessons Learned in Managing Object-Oriented Development. *IEEE Software* vol. 10(1).

Plinta, C., Lee, K., and Rissman, M. March 29, 1989. A Model Solution for C3I: Message Translation and Validation. Pittsburgh, PA: Software Engineering Institute.

Pope, S. April/May 1988. Building Smalltalk-80-Based Computer Music Tools. *Journal of Object-Oriented Programming* vol. 1(1).

Raghavan, R. 1990. *Taming Windows 3.0 and DOS Using C++*. Lake Oswego, OR: Wyatt Software.

Rockwell International. 1989. *Rockwell Advanced Railroad Electronic Systems*. Cedar Rapids, IA.

Rombach, D. March 1990. Design Measurement: Some Lessons Learned. *IEEE Software* vol. 7(2).

Rubin, K., Jones, P., Mitchell, C., and Goldstein, T. September 1988. A Smalltalk Implementation of an Intelligent Operator's Associate. *SIGPLAN Notices* vol. 23(11).

Rubin, R., Walker, J., and Golin, E. October 1990. Early Experience with the Visual Programmer's WorkBench. *IEEE Transactions on Software Engineering* vol. 16(10).

Ruspini, E., and Fraley, R. 1983. ID: An Intelligent Information Dictionary System. In *Entity-Relationship Approach to Software Engineering*, ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.

Russo, V., Johnston, G., and Campbell, R. September 1988. Process Management and Exception Handling in Multiprocessor Operating Systems Using Object-Oriented Design Techniques. *SIGPLAN Notices* vol. 23(11).

Sampson, J., and Womble, B. 1988. *SEND: Simulation Environment for Network Design*. Dallas, TX: Southern Methodist University.

Santori, M. August 1990. An Instrument That Isn't Really. *IEEE Spectrum* vol. 27(8).

Scaletti, C., and Johnson, R. September 1988. An Interactive Environment for Object-Oriented Music Composition and Sound Synthesis. *SIGPLAN Notices* vol. 23(11).

Schindler, J., and Joy, S. February 1992. *An Introduction to Object Technology at Liberty Mutual.* Liberty Mutual Information Systems Research and Development.

Schoen, E., Smith, R., and Buchanan, B. December 1988. Design of Knowledge-Based Systems with a Knowledge-Based Assistant. *IEEE Transactions on Software Engineering* vol. 14(12).

Schulert, A., and Erf, K. 1988. Open Dialogue: Using an Extensible Retained Object Workspace to Support a UIMS. *Proceedings of USENIX C++ Conference.* Berkeley, CA: USENIX Association.

Scott, R., Reddy, P., Edwards, R., and Campbell, D. 1988. GPIO: Extensible Objects for Electronic Design. *Proceedings of USENIX C++ Conference.* Berkeley, CA: USENIX Association.

Smith, R., Barth, P., and Young, R. 1987. A Substrate for Object-Oriented Interface Design. *Research Directions in Object-Oriented Programming.* Cambridge, MA: The MIT Press.

Smith, R., Dinitz, R., and Barth, P. November 1986. Impulse-86: A Substrate for Object-Oriented Interface Design. *SIGPLAN Notices* vol. 21(11).

Sneed, H., and Gawron, W. 1983. The Use of the Entity/Relationship Model as a Schema for Organizing the Data Processing Activities at the Bavarian Motor Works. In *Entity-Relationship Approach to Software Engineering*, ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.

Snodgrass, R. 1987. An Object-Oriented Command Language. In *Object Oriented Computing: Implementations* vol. 2, ed. G. Peterson. New York, NY: Computer Society Press of the IEEE.

Software Made Simple. September 30, 1991. *Business Week.*

Sridhar, S. September 1988. Configuring Stand-Alone Smalltalk-80 Applications. *SIGPLAN Notices* vol. 23(11).

Stadel, M. January 1991. Object-Oriented Programming Techniques to Replace Software Components on the Fly in a Running Program. *SIGPLAN Notice* vol. 26(1).

Stevens, A. 1992. *C++ Database Development.* New York, NY: MIS Press.

Stokes, R. 1988. Prototyping Database Applications with a Hybrid of C++ and 4GL. *Proceedings of USENIX C++ Conference.* Berkeley, CA: USENIX Association.

Szcur, M., and Miller, P. September 1988. Transportable Applications Environment (TAE) PLUS: Experiences in Objectively Modernizing a User Interface Environment. *SIGPLAN Notices* vol. 23(11).

Szekely, P., and Myers, B. September 1988. A User Interface Toolkit Based on Graphical Objects and Constraints. *SIGPLAN Notices* vol. 23(11).

Tanner, J. April 1, 1986. *Fault Tree Analysis in an Object-Oriented Environment.* Mountain View, CA: IntelliCorp.

Taylor, D. 1992. *Object-Oriented Information Systems.* New York, NY: John Wiley and Sons.

Temte, M. November/December 1984. Object-Oriented Design and Ballistics Software. *Ada Letters* vol. 4(3).

Tripathi, A., and Aksit, M. November/December 1988. Communication, Scheduling, and Resource Management in SINA. *Journal of Object-Oriented Programming* vol. 1(4).

Tripathi, A., Ghonami, A., and Schmitz, T. 1987. Object Management in the NEXUS Distributed Operating System. *Proceedings of the Thirty-second IEEE Computer Society International Conference*. New York, NY: Computer Society Press of the IEEE.

Ursprung, P., and Zehnder, C. 1983. HIQUEL: An Interactive Query Language to Define and Use Hierarchies. In *Entity-Relationship Approach to Software Engineering*, ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.

van der Meulen, P. October 1987. INSIST: Interactive Simulation in Smalltalk. *SIGPLAN Notices* vol. 22(12).

Vernon, V. September/October 1989. The Forest for the Trees. *Programmer's Journal* vol. 7(5).

Vilot, M. Fall 1990. Using Object-Oriented Design and C++. *The C++ Journal* vol. 1(1).

Vines, D., and King, T. 1987. *Experiences in Building a Prototype Object-Oriented Framework in Ada*. Minneapolis, MN: Honeywell.

Vlissides, J., and Linton, M. 1988. Applying Object-Oriented Design to Structured Graphics. *Proceedings of USENIX C++ Conference*. Berkeley, CA: USENIX Association.

Volz, R., Mudge, T., and Gal, D. 1987. Using Ada as a Programming Language for Robot-Based Manufacturing Cells. In *Object-Oriented Computing: Concepts* vol. 1, ed. G. Peterson. New York, NY: Computer Society Press of the IEEE.

Walther, S., and Peskin, R. October 1989. Strategies for Scientific Prototyping in Smalltalk. *SIGPLAN Notices* vol. 24(10).

Wasserman, A., and Pircher, P. January 1991. Object-Oriented Structured Design and C++. *Computer Language* vol. 8(1).

Weinand, A., Gamma, E., and Marty, R. September 1988. ET++—An Object-Oriented Application Framework in C++. *SIGPLAN Notices* vol. 23(11).

Welch, B. July/August 1991. Securities Objects—The Complexity. *Object Magazine* vol. 1(2).

White, S. October 1986. Panel Problem: Software Controller for an Oil Hot Water Heating System. *Proceedings of COMPSAC*. New York, NY: Computer Society Press of the IEEE.

Wirfs-Brock, R. September 1988. An Integrated Color Smalltalk-80 System. *SIGPLAN Notices* vol. 23(11).

———. October 1991. Object-Oriented Frameworks. *American Programmer* vol. 4(10).

WOSA Extensions for Financial Services. December 1992. Banking Systems Vendor Council.

Wu, P. January 1992. An Object-Oriented Specification for a Compiler. *SIGPLAN Notices* vol. 27(1).

Yoshida, N., and Hino, K. September 1988. An Object-Oriented Framework of Pattern Recognition. *SIGPLAN Notices* vol. 23(11).

Yoshida, T., and Tokoro, M. March 31, 1986. *Distributed Queueing Network Simulation: An Application of a Concurrent Object-Oriented Language*. Yokohama, Japan: Keio University.

Young, R. October 1987. An Object-Oriented Framework for Interactive Data Graphics. *SIGPLAN Notices* vol. 22(12).

D. 面向对象架构

Athas, W., and Seitz, C. August 1988. Multicomputers: Message-Passing Concurrent Computers. *IEEE Computer* vol. 21(8).

Corkill, D. D. September 1991. Blackboard Systems. *AI Expert* vol. 6(9), pp. 40–47.

Dahlby, S., Henry, G., Reynolds, D., and Taylor, P. 1982. The IBM System/38: A High Level Machine. In *Computer Structures: Principles and Examples*, ed. G. Bell and A. Newell. New York, NY: McGraw-Hill.

Dally, W., and Kajiya, J. March 1985. An Object-Oriented Architecture. *SIGARCH Newsletter* vol. 13(3).

Fabry, R. 1987. Capability-Based Addressing. In *Object-Oriented Computing: Implementations* vol. 2, ed. G. Peterson. New York, NY: Computer Society Press of the IEEE.

Flynn, M. October 1980. Directions and Issues in Architecture and Language. *IEEE Computer* vol. 13(10).

Harland, D., and Beloff, B. December 1986. Microcoding an Object-Oriented Instruction Set. *Computer Architecture News* vol. 14(5).

Hillis, D. 1985. *The Connection Machine*. Cambridge, MA: The MIT Press.

Hunt, J. May 2002. Blackboard Architectures. JayDee Technology Ltd 27. www.agent.ai/doc/upload/200402/hunt02_1.pdf.

IEEE-Std-1471-2000. 2000. *IEEE Recommended Practice for Architectural Description of Software-Intensive Systems*. New York, NY: IEEE.

Iliffe, J. 1982. *Advanced Computer Design*. London, England: Prentice Hall International.

Intel. 1981. *iAPX 432 Object Primer*. Santa Clara, CA.

Ishikawa, Y., and Tokoro, M. March 1984. The Design of an Object-Oriented Architecture. *SIGARCH Newsletter* vol. 12(3).

Kavi, K., and Chen, D. 1987. Architectural Support for Object-Oriented Languages. *Proceedings of the Thirty-second IEEE Computer Society International Conference*. New York, NY: Computer Society Press of the IEEE.

Krikorian, H. F. March/April 2003. Introduction to Object-Oriented Systems Engineering, Part 1. *IT Professional*, p. 40.

Kruchten, P. 1995. The 4+1 View Model of Architecture. *IEEE Software* 12(6), pp. 42–50.

Lahtinen, P. September/October 1982. A Machine Architecture for Ada. *Ada Letters* vol. 2(2).

Lampson, B., and Pier, K. January 1981. A Processor for a High-Performance Personal Computer. In *The Dorado: A High Performance Personal Computer*. Report CSL-81-1. Palo Alto, CA: Xerox Palo Alto Research Center.

Langdon, G. 1982. *Computer Design*. San Jose, CA: Computeach Press.

Levy, H. 1984. *Capability-Based Computer Systems*. Bedford, MA: Digital Press.

Lewis, D., Galloway, D., Francis, R., and Thomson, B. November 1986. Swamp: A Fast Processor for Smalltalk-80. *SIGPLAN Notices* vol. 21(11).

Maier, M. W., Emery, D., and Hilliard, R. 2004. ANSI/IEEE 1471 and Systems Engineering. *Systems Engineering* vol. 7(3), pp. 257–270.

Mashburn, H. 1982. The C.mmp/Hydra Project: An Architectural Overview. In *Computer Structures: Principles and Examples*, ed. G. Bell and A. Newell. New York, NY: McGraw-Hill.

Myers, G. 1982. *Advances in Computer Architecture*. Second Edition. New York, NY: John Wiley and Sons.

Rattner, J. 1982. Hardware/Software Cooperation in the iAPX-432. *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*. New York, NY: Association of Computing Machinery.

Rose, J. September 1988. Fast Dispatch Mechanisms for Stock Hardware. *SIGPLAN Notices* vol. 23(11).

Samples, D., Ungar, D., and Hilfinger, P. November 1986. SOAR: Smalltalk without Bytecodes. *SIGPLAN Notices* vol. 21(11).

Shaw, M., and Garlan, D. 1996. *Software Architecture, Perspectives on an Emerging Discipline*. Upper Saddle River, NJ: Prentice Hall.

Soltis, R., and Hoffman, R. 1987. Design Considerations for the IBM System/38. In *Object-Oriented Computing: Implementations* vol. 2, ed. G. Peterson. New York, NY: Computer Society Press of the IEEE.

Thacker, C., McCreight, E., Lampson, B., Sproull, R., and Boggs, D. August 1979. *Alto: A Personal Computer*, Report CSL-79-11. Palo Alto, CA: Xerox Palo Alto Research Center.

Ungar, D. 1987. *The Design and Evaluation of a High-Performance Smalltalk System*. Cambridge, MA: The MIT Press.

Ungar, D., and Patterson, D. January 1987. What Price Smalltalk? *IEEE Computer* vol. 20(1).

Ungar, D., Blau, R., Foley, P., Samples, D., and Patterson, D. March 1984. Architecture of SOAR: Smalltalk on a RISC. *SIGARCH Newsletter* vol. 12(3).

Wah, B., and Li, G. April 1986. Survey on Special Purpose Computer Architectures for AI. *SIGART Newsletter*, no. 96.

Wulf, W. January 1980. Trends in the Design and Implementation of Programming Languages. *IEEE Computer* vol. 13(1).

Wulf, W., Levin, R., and Harbison, S. 1981. *HYDRA/C.mmp: An Experimental Computer System*. New York, NY: McGraw-Hill.

E. 面向对象数据库

Alford, M. 1983. Derivation of Element-Relation-Attribute Database Requirements by Decomposition of System Functions. In *Entity-Relationship Approach to Software Engineering*, ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.

Andleigh, P., and Gretzinger, M. 1992. *Distributed Object-Oriented Data-Systems Design*. Englewood Cliffs, NJ: Prentice Hall.

Atkinson, M., Bailey, P., Chisholm, K., Cockshott, P., and Morrison, R. 1983. An Approach to Persistent Programming. *The Computer Journal* vol. 26(4).

Atkinson, M., and Buneman, P. June 1987. Types and Persistence in Database Programming Languages. *ACM Computing Surveys* vol. 19(2).

Atkinson, M., and Morrison, R. October 1985. Procedures as Persistent Data Objects. *ACM Transactions on Programming Languages and Systems* vol. 7(4).

Atwood, T. February 1991. Object-Oriented Databases. *IEEE Spectrum* vol. 28(2).

Bachman, C. 1983. The Structuring Capabilities of the Molecular Data Model. In *Entity-Relationship Approach to Software Engineering*, ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.

Batini, C., and Lenzerini, M. 1983. A Methodology for Data Schema Integration in the Entity-Relationship Model. In *Entity-Relationship Approach to Software Engineering*, ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.

Beech, D. 1987. Groundwork for an Object Database Model. In *Research Directions in Object-Oriented Programming*, ed. B. Schriver and P. Wegner. Cambridge, MA: The MIT Press.

———. September 1988. Intensional Concepts in an Object Database Model. *SIGPLAN Notices* vol. 23(11).

Bertino, E. 1983. Distributed Database Design Using the Entity-Relationship Model. In *Entity-Relationship Approach to Software Engineering*, ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.

Blackwell, P., Jajodia, S., and Ng, P. 1983. A View of Database Management Systems as Abstract Data Types. In *Entity-Relationship Approach to Software Engineering*, ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.

Bloom, T. October 1987. Issues in the Design of Object-Oriented Database Programming Languages. *SIGPLAN Notices* vol. 22(12).

Bobrow, D., Fogelsong, D., and Miller, M. 1987. Definition Groups: Making Sources into First-Class Objects. In *Research Directions in Object-Oriented Programming*, ed. B. Schriver and P. Wegner. Cambridge, MA: The MIT Press.

Brathwaite, K. 1983. An Implementation of a Data Dictionary to Support Databases Designed Using the Entity-Relationship (E-R) Approach. In *Entity-Relationship Approach to Software Engineering*, ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.

Breazeal, J., Blattner, M., and Burton, H. March 28, 1986. *Data Standardization Through the Use of Data Abstraction*. Livermore, CA: Lawrence Livermore National Laboratory.

Brodie, M. 1984. On the Development of Data Models. In *On Conceptual Modeling: Perspectives from Artificial Intelligence, Databases, and Programming Languages*, ed. M. Brodie, J. Mylopoulos, and J. Schmidt. New York, NY: Springer-Verlag.

Brodie, M., and Ridjanovic, D. 1984. On the Design and Specification of Database Transactions. In *On Conceptual Modeling: Perspectives from Artificial Intelligence, Databases, and Programming Languages*, ed. M. Brodie, J. Mylopoulos, and J. Schmidt. New York, NY: Springer-Verlag.

Butterworth, P., Otis, A., and Stein, J. October 1991. The GemStone Object Database Management System. *Communications of the ACM* vol. 34(10).

Carlson, C., and Arora, A. 1983. UPM: A Formal Tool for Expressing Database Update Semantics. In *Entity-Relationship Approach to Software Engineering*, ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.

Casanova, M. 1983. Designing Entity-Relationship Schemes for Conventional Information Systems. In *Entity-Relationship Approach to Software Engineering*, ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.

Cattell, R. May 1983. *Design and Implementation of a Relationship-Entity-Datum Data Model*, Report CSL-83-4. Palo Alto, CA: Xerox Palo Alto Research Center.

———. 1991. *Object Data Management*. Reading, MA: Addison-Wesley. Chen, P. March 1976. The Entity-Relationship Model—Toward a Unified View of Data. *ACM Transactions on Database Systems* vol. 1(1).

———. 1983. ER—A Historical Perspective and Future Directions. In *Entity-Relationship Approach to Software Engineering*, ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.

Claybrook, B., Claybrook, A., and Williams, J. January 1985. Defining Database Views as Data Abstractions. *IEEE Transactions on Software Engineering* vol. SE-11(1).

D'Cunha, A., and Radhakrishnan, T. 1983. Applications of E-R Concepts to Data Administration. In *Entity-Relationship Approach to Software Engineering*, ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.

Date, C. 1981, 1983. *An Introduction to Database Systems*. Reading, MA: Addison-Wesley.

———. 1986. *Relational Database: Selected Writings*. Reading, MA: Addison-Wesley.

———. 1987. *The Guide to the SQL Standard*. Reading, MA: Addison-Wesley.

Duhl, J., and Damon, C. September 1988. A Performance Comparison of Object and Relational Databases Using the Sun Benchmark. *SIGPLAN Notices* vol. 23(11).

Harland, D., and Beloff, B. April 1987. OBJEKT—A Persistent Object Store with an Integrated Garbage Collector. *SIGPLAN Notices* vol. 22(4).

Hawryszkiewycz, I. 1984. *Database Analysis and Design*. Chicago, IL: Science Research Associates.

Higa, K., Morrison, M., Morrison, J., and Sheng, O. June 1992. An Object-Oriented Methodology for Knowledge Base/Database Coupling. *Communications of the ACM* vol. 35(6).

Hull, R., and King, R. September 1987. Semantic Database Modeling: Survey, Applications, and Research Issues. *ACM Computing Surveys* vol. 19(3).

Jajodia, S., Ng, P., and Springsteel, F. 1983. On Universal and Representative Instances for Inconsistent Databases. In *Entity-Relationship Approach to Software Engineering*, ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.

Ketabchi, M., and Berzins, V. January 1988. Mathematical Model of Composite Objects and Its Application for Organizing Engineering Databases. *IEEE Transactions on Software Engineering* vol. 14(1).

Ketabchi, M., and Wiens, R. 1987. Implementation of Persistent Multi-User Object-Oriented Systems. *Proceedings of the Thirty-second IEEE Computer Society International Conference*. New York, NY: Computer Society Press of the IEEE.

Khoshafian, S., and Abnous, R. 1990. *Object-Orientation: Concepts, Languages, Databases, User Interfaces*. New York, NY: John Wiley and Sons.

Kim, W., and Lochovsky, K. 1989. *Object-Oriented Concepts, Databases, and Applications*. Reading, MA: Addison-Wesley.

Kim, W., Ballou, N., Chou, H., Garze, J., Woelk, D., and Banerjee, J. September 1988. Integrating an Object-Oriented Programming System with a Database System. *SIGPLAN Notices* vol. 23(11).

Kim, W., Banerjee, J., Chou, H., Garza, J., and Woelk, D. October 1987. Composite Object Support in an Object-Oriented Database System. *SIGPLAN Notices* vol. 22(12).

Kung, C. Object Subclass Hierarchy in SQL: A Simple Approach. *Communications of the ACM* vol. 33(7).

Laenens, E., and Vermeir, D. August 1988. An Overview of OOPS+, an Object-Oriented Database Programming Language. *Proceedings of ECOOP'88: European Conference on Object-Oriented Programming*. New York, NY: Springer-Verlag.

Lamb, C., Landis, G., Orenstein, J., and Weinreb, D. October 1991. The ObjectStore Database System. *Communications of the ACM* vol. 34(10).

Larson, J., and Dwyer, P. 1983. Defining External Schemas for an Entity-Relationship Database. In *Entity-Relationship Approach to Software Engineering*, ed. C. Davis et al. Amsterdam, The

Netherlands: Elsevier Science.

Maier, D., and Stein, J. 1987. Development and Implementation of an Object-Oriented DBMS. In *Research Directions in Object-Oriented Programming*, ed. B. Schriver and P. Wegner. Cambridge, MA: The MIT Press.

Margrave, G., Lusk, E., and Overbeek, R. 1983. Tools for the Creation of IMS Database Designs from Entity-Relationship Diagrams. In *Entity-Relationship Approach to Software Engineering*, ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.

Mark, L., and Poussopoulos, N. 1983. Integration of Data, Schema, and Meta-Schema in the Context of Self-Documenting Data Models. In *Entity-Relationship Approach to Software Engineering*, ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.

Markowitz, V., and Makowsky, J. August 1990. Identifying Extended Entity-Relationship Object Structures in Relational Schemas. *IEEE Transactions on Software Engineering* vol. 16(8).

Marti, R. 1983. Integrating Database and Program Descriptions Using an ER Data Dictionary. In *Entity-Relationship Approach to Software Engineering*, ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.

Merrow, T., and Laursen, J. October 1987. A Pragmatic System for Shared Persistent Objects. *SIGPLAN Notices* vol. 22(12).

Mitchell, J., and Wegbreit, B. 1977. Schemes: A High-Level Data Structuring Concept. In *Current Trends in Programming Methodology: Data Structuring* vol. 4, ed. R. Yeh. Englewood Cliffs, NJ; Prentice Hall.

Morrison, R., Atkinson, M., Brown, A., and Dearle, A. April 1988. Bindings in Persistent Programming Languages. *SIGPLAN Notices* vol. 23(4).

Moss, E., Herlihy, M., and Zdonik, S. September 1988. *Object-Oriented Databases, Course Notes*. San Diego, CA: OOPSLA'88.

Moss, J. August 1992. Working with Persistent Objects: To Swizzle or Not to Swizzle. *IEEE Transactions on Software Engineering* vol. 18(8).

Nastos, M. January 1988. Databases, Etc. *HOOPLA: Hooray for Object-Oriented Programming Languages* vol. 1(2). Everette, WA: Object Oriented Programming for Smalltalk Application Developers Association.

Navathe, S., and Cheng, A. 1983. A Methodology for Database Schema Mapping from Extended Entity Relationship Models into the Hierarchical Model. In *Entity-Relationship Approach to Software Engineering*, ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.

Ontologic. 1987. *Vbase Technical Overview*. Billerica, MA.

Oracle. 1989. *Oracle for Macintosh: References, Version 1.1*. Belmont, CA.

Penny, J., and Stein, J. October 1987. Class Modification in the GemStone Object-Oriented DBMS. *SIGPLAN Notices* vol. 22(12).

Peterson, R. 1987. Object-Oriented Database Design. In *Object-Oriented Computing: Implementations* vol. 2, ed. G. Peterson. New York, NY: Computer Society Press of the IEEE.

Premerlani, W., Blaha, M., Rumbaugh, J., and Varwig, T. November 1990. An Object-Oriented Relational Database. *Communications of the ACM* vol. 33(11).

Sakai, H. 1983. Entity-Relationship Approach to Logical Database Design. In *Entity-Relationship Approach to Software Engineering*, ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.

Skarra, A., and Zdonik, S. 1987. Type Evolution in an Object-Oriented Database. In *Research Directions in Object-Oriented Programming*, ed. B. Schriver and P. Wegner. Cambridge, MA: The MIT Press.

Skarra, A., and Zdonik, S. November 1986. The Management of Changing Types in an Object-Oriented Database. *SIGPLAN Notices* vol. 21(11).

Smith, D., and Smith, J. 1980. Conceptual Database Design. In *Tutorial on Software Design Techniques*, Third Edition, ed. P. Freeman and A. Wasserman. New York, NY: Computer Society Press of the IEEE.

Smith, J., and Smith, D. Database Abstractions: Aggregation and Generalization. *ACM Transactions on Database Systems* vol. 2(2).

Smith, K., and Zdonik, S. October 1987. Intermedia: A Case Study of the Differences Between Relational and Object-Oriented Database Systems. *SIGPLAN Notices* vol. 22(12).

Stein, J. March 1988. Object-Oriented Programming and Database Design. *Dr. Dobb's Journal* vol. 13(3).

Teorey, T., Yang, D., and Fry, J. June 1986. A Logical Design Methodology for Relational Databases Using the Extended Entity-Relationship Model. *ACM Computing Surveys* vol. 18(2).

Thuraisingham, M. October 1989. Mandatory Security in Object-Oriented Database Systems. *SIGPLAN Notices* vol. 24(10).

Veloso, P., and Furtado, A. 1983. View Constructs for the Specification and Design of External Schemas. In *Entity-Relationship Approach to Software Engineering*, ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.

Wiebe, D. November 1986. A Distributed Repository for Immutable Persistent Objects. *SIGPLAN Notices* vol. 21(11).

Wiederhold, G. December 1986. Views, Objects, and Databases. *IEEE Computer* vol. 19(12).

Wile, D., and Allard, D. May 1982. Worlds: An Organizing Structure for Object-Bases. *SIGPLAN Notices* vol. 19(5).

Zdonik, S., and Maier, D. 1990. *Readings in Object-Oriented Database Systems*. San Mateo, CA: Morgan Kaufmann.

Zhang, Z., and Mendelzon, A. 1983. A Graphical Query Language for Entity-Relationship Databases.

In *Entity-Relationship Approach to Software Engineering*, ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.

F. 面向对象设计

Abbott, R. November 1983. Program Design by Informal English Descriptions. *Communications of the ACM* vol. 26(11).

———. August 1987. Knowledge Abstraction. *Communications of the ACM* vol. 30(8).

Ackroyd, M., and Daum, D. 1991. Graphical Notation for Object-Oriented Design and Programming. *Journal of Object-Oriented Programming* vol. 3(5).

Alabios, B. September 1988. Transformation of Data Flow Analysis Models to Object-Oriented Design. *SIGPLAN Notices* vol. 23(11).

Arnold, P., Bodoff, S., Coleman, D., Gilchrist, H., and Hayes, F. June 1991. *An Evaluation of Five Object-Oriented Development Methods*. Bristol, England: Hewlett-Packard Laboratories.

Bear, S., Allen, P., Coleman, D., and Hayes, F. Graphical Specification of Object-Oriented Systems. *Object-Oriented Programming Systems, Languages, and Applications*. Ottawa, Canada: OOPSLA'90.

Beck, K., and Cunningham, W. October 1989. A Laboratory for Teaching Object-Oriented Thinking. *SIGPLAN Notices* vol. 24(10).

Berard, E. 1986. *An Object-Oriented Design Handbook*. Rockville, MD: EVB Software Engineering.

Berzins, V., Gray, M., and Naumann, D. May 1986. Abstraction-Based Software Development. *Communications of the ACM* vol. 29(5).

Blaha, M. April 1988. Relational Database Design Using an Object-Oriented Methodology. *Communications of the ACM* vol. 31(4).

Booch, G. September 1981. Describing Software Design in Ada. *SIGPLAN Notices* vol. 16(9).

———. March/April 1982. Object-Oriented Design. *Ada Letters* vol. 1(3).

———. February 1986. Object-Oriented Development. *IEEE Transactions on Software Engineering* vol. 12(2).

———. 1987. *On the Concepts of Object-Oriented Design*. Denver, CO: Rational.

———. Summer 1989. What Is and What Isn't Object-Oriented Design. *American Programmer* vol. 2(7–8).

Booch, G., and Vilot, M. Object-Oriented Design. *The C++ Report*.

Booch, G., Jacobson, I., and Kerth, N. September 1988. *Specification and Design Methodologies in Support of Object-Oriented Programming, Course Notes*. San Diego, CA: OOPSLA'88.

Bowles, A. November/December 1991. Evolution vs. Revolution: Should Structured Methods Be Objectified? *Object Magazine* vol. 1(4).

Boyd, S. July/August 1987. Object-Oriented Design and PAMELA™. *Ada Letters* vol. 7(4).

Bril, R., deBunje, T., and Ouvry, A. October 1991. *Development of SCORE: Towards the*

Industrialization of an Object-Oriented Method Using the Formal Design Language COLD-1 as Notation. Eindhoven, The Netherlands: Philips Research Laboratories.

Brookman, D. November/December 1991. SA/SD versus OOD. *Ada Letters* vol. XI(9).

Bruno, G., and Balsamo, A. November 1986. Petri Net-Based Object-Oriented Modeling of Distributed Systems. *SIGPLAN Notices* vol. 21(11).

Buhr, R. 1984. *System Design with Ada*. Englewood Cliffs, NJ: Prentice Hall.

———. August 22, 1988. *Machine Charts for Visual Prototyping in System Design*. SCE Report 88-2. Ottawa, Canada: Carleton University.

———. September 14, 1988. *Visual Prototyping in System Design*. SCE Report 88-14. Ottawa, Canada: Carleton University.

———. 1989. *System Design with Machine Charts: A CAD Approach with Ada Examples*. Englewood Cliffs, NJ: Prentice Hall.

Buhr, R., Karam, G., Hayes, C., and Woodside, M. March 1989. Software CAD: A Revolutionay Approach. *IEEE Transactions on Software Engineering* vol. 15(3).

Bulman, D. August 1989. An Object-Based Development Model. *Computer Language* vol. 6(8).

Cherry, G. 1987. *PAMELA 2: An Ada-Based Object-Oriented Design Method*. Reston, VA: Thought** Tools.

———. 1990. *Software Construction by Object-Oriented Pictures*. Canandaigua, NY: Thought**Tools.

Clark, R. June 1987. Designing Concurrent Objects. *Ada Letters* vol. 7(6).

Coad, P. September 1991. OOD Criteria. *Journal of Object-Oriented Programming* vol.(5).

Coleman, D., Hayes, F., and Bear, S. December 1990. *Introducing Objectcharts or How to Use Statecharts in Object-Oriented Design*. Bristol, England: Hewlett-Packard Laboratories.

Comer, E. July 1989. *Ada Box Structure Methodology Handbook*. Melbourne, FL: Software Productivity Solutions.

Constantine, L. Summer 1989. Object-Oriented and Structured Methods: Towards Integration. *American Programmer* vol. 2(7–8).

CRI, CISI Ingenierie, and Matra. June 20, 1987. *HOOD: Hierarchical Object-Oriented Design*. Paris, France.

Cribbs, J., Moon, S., and Roe, C. 1992. *An Evaluation of Object-Oriented Analysis and Design Methodologies*. Raleigh, NC: Alcatel Network Systems.

Cunningham, W., and Beck, K. November 1986. A Diagram for Object-Oriented Programs. *SIGPLAN Notices* vol. 21(11).

Davis, N., Irving, M., and Lee, J. 1988. *The Evolution of Object-Oriented Design from Concept to Method*. Surrey, United Kingdom: Logica Space and Defence Systems Limited.

Dean, H. May 1991. Object-Oriented Design Using Message Flow Decomposition. *Journal of*

Object-Oriented Programming vol. 4(2).

deChampeaux, D., Balzer, B., Bulman, D., Culver-Lozo, K., Jacobson, I., and Mellor, S. 1992. *The Object-Oriented Software Development Process*. Vancouver, Canada: OOPSLA'92.

deChampeaux, D., Lea, D., and Faure, P. 1992. *The Process of Object-Oriented Design*. Vancouver, Canada: OOPSLA'92.

Edwards, J., and Henderson-Sellers, B. November 1991. *A Graphical Notation for Object-Oriented Analysis and Design*. New South Wales, Australia: University of New South Wales.

Felsinger, R. 1987a. *Integrating Object-Oriented Design, Structured Analysis/Structured Design, and Ada for Real-Time Systems*. Mt. Pleasant, SC.

———. 1987b. *Object-Oriented Design, Course Notes*. Torrance, CA: Data Processing Management Association.

Fichman, R., and Kemerer, C. October 1992. Object-Oriented and Conventional Analysis and Design Methodologies. *IEEE Computer* vol. 25(10).

Firesmith, D. May 6, 1986. *Object-Oriented Development*. Fort Wayne, IN: Magnavox Electronic Systems Co.

———. 1993. *Object-Oriented Requirements Analysis and Logical Design*. New York, NY: John Wiley and Sons.

Fowler, M. 1992. *A Comparison of Object-Oriented Analysis and Design Methods*. Vancouver, Canada: OOPSLA'92.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. 1993. *A Catalog of Object-Oriented Design Patterns*. Cupertino, CA: Taligent.

———. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley.

Gane, C. Summer 1989. Object-Oriented Data/Process Modeling. *American Programmer* vol. 2(7–8).

Giddings, R. May 1984. Accommodating Uncertainty in Software Design. *Communications of the ACM* vol. 27(5).

Gomaa, H. September 1984. A Software Design Method for Real-Time Systems. *Communications of the ACM* vol. 27(9).

Gossain, S., and Anderson, B. *An Iterative Design Model for Reusable Objects*. Ottawa, Canada: OOPSLA'90.

Gouda, M., Han, Y., Jensen, E., Johnson, W., and Kain, R. November 1977. Towards a Methodology of Distributed Computer System Design. *Sixth Texas Conference on Computing Systems*. New York, NY: Association of Computing Machinery.

Graham, I. 1991. *Object-Oriented Methods*. Wokingham, England: Addison-Wesley.

Grosch, J. December 1983. Type Derivation Graphs—A Way to Visualize the Type Building

Possibilities of Programming Languages. *SIGPLAN Notices* vol. 18(12).

Harel, D. 1987. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming* vol. 8.

———. May 1988. On Visual Formalisms. *Communications of the ACM* vol. 31(5).

Henderson-Sellers, B. 1992. *A Book of Object-Oriented Knowledge*. Englewood Cliffs, NJ: Prentice Hall.

Inwood, C. 1992. Analysis versus Design: Is There a Difference? *The C++ Journal* vol. 2(1).

Jackson, M. Summer 1989. Object-Oriented Software. *American Programmer* vol. 2(7–8).

Jacobson, I. August 1985. *Concepts for Modeling Large Real-Time Systems*. Academic dissertation. Stockholm, Sweden: Royal Institute of Technology, Department of Computer Science.

———. October 1987. Object-Oriented Development in an Industrial Environment. *SIGPLAN Notices* vol. 22(12).

Jacobson, I., Christerson, M., Jonsson, P., and Overgaard, G. 1992. *Object-Oriented Software Engineering*. Wokingham, England: Addison-Wesley.

Jamsa, K. January 1984. Object-Oriented Design vs. Structured Design—A Student's Perspective. *Software Engineering Notes* vol. 9(1).

Johnson, R., and Russo, V. May 1991. *Reusing Object-Oriented Designs*. Urbana, IL: University of Illinois.

Jones, A. 1979. The Object Model: A Conceptual Tool for Structuring Software. In *Operating Systems*, ed. R. Bayer et al. New York, NY: Springer-Verlag.

Kadie, C. 1986. *Refinement Through Classes: A Development Methodology for Object-Oriented Languages*. Urbana, IL: University of Illinois.

Kaplan, S., and Johnson, R. July 21, 1986. *Designing and Implementing for Reuse*. Urbana, IL: University of Illinois, Department of Computer Science.

Kay, A. August 1969. *The Reactive Engine*. Salt Lake City, UT: The University of Utah, Department of Computer Science.

Kelly, J. 1986. A Comparison of Four Design Methods for Real-Time Systems. *Proceedings of the Ninth International Conference on Software Engineering*. New York, NY: Computer Society Press of the IEEE.

Kent, W. 1983. Fact-Based Data Analysis and Design. In *Entity-Relationship Approach to Software Engineering*, ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.

Kerth, N. L. 1988. MOOD: A Methodology for Structured Object-Oriented Design. Tutorial presented at OOPSLA'88, San Diego, CA.

Kim, J., and Lerch, J. 1992. *Towards a Model of Cognitive Process in Logical Design: Comparing Object-Oriented and Traditional Functional Decomposition Software Methodologies*. Pittsburgh, PA: Carnegie Mellon University.

Kruchten, P. 2003. *The Rational Unified Process: An Introduction.* Third Edition. Boston, MA: Addison-Wesley.

Ladden, R. July 1988. A Survey of Issues to Be Considered in the Development of an Object-Oriented Development Methodology for Ada. *Software Engineering Notes* vol. 13(3).

Lieberherr, K., and Riel, A. October 1989. Contributions to Teaching Object-Oriented Design and Programming. *SIGPLAN Notices* vol. 24(10).

Liskov, B. 1980. A Design Methodology for Reliable Software Systems. In *Tutorial on Software Design Techniques*, Third Edition, ed. P. Freeman and A. Wasserman. New York, NY: Computer Society Press of the IEEE.

Lorenz, M. 1993. *Object-Oriented Software Development.* Englewood Cliffs, NJ: Prentice Hall.

Manniño, P. April 1987. A Presentation and Comparison of Four Information System Development Methodologies. *Software Engineering Notes* vol. 12(2).

Martin, B. 1993. *Designing Object-Oriented C++ Applications Using the Booch Method.* Englewood Cliffs, NJ: Prentice Hall.

Masiero, P., and Germano, F. July 1988. JSD as an Object-Oriented Design Method. *Software Engineering Notes* vol. 13(3).

Meyer, B. 1988. *Object-Oriented Software Construction.* Englewood Cliffs, NJ: Prentice Hall.

Meyer, B. March 1987. Reusability: The Case for Object-Oriented Design. *IEEE Software* vol. 4(2).

———. 1989. From Structured Programming to Object-Oriented Design: The Road to Eiffel. *Structured Programming* vol. 10(1).

Mills, H. June 1988. Stepwise Refinement and Verification in Box-Structured Systems. *IEEE Computer* vol. 21(6).

Mills, H., Linger, R., and Hevner, A. 1986. *Principles of Information System Design and Analysis.* Orlando, FL: Academic Press.

Minkowitz, C., and Henderson, P. March 1987. *Object-Oriented Programming of Discrete Event Simulation Using Petri Nets.* Stirling, Scotland: University of Stirling.

Monarchi, D., and Puhr, G. September 1992. A Research Typology for Object-Oriented Analysis and Design. *Communications of the ACM* vol. 35(9).

Mostow, J. Spring 1985. Toward Better Models of the Design Process. *AI Magazine* vol. 6(1).

Moulin, B. 1983. The Use of EPAS/IPSO Approach for Integrating Entity Relationship Concepts and Software Engineering Techniques. In *Entity-Relationship Approach to Software Engineering*, ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.

Mullin, M. 1989. *Object-Oriented Program Design with Examples in C++.* Reading, MA: Addison-Wesley.

Nielsen, K., and Shumate, K. August 1987. Designing Large Real-Time Systems with Ada.

Communications of the ACM vol. 30(8).

Nielsen, K. March 1988. *An Object-Oriented Design Methodology for Real-Time Systems in Ada.* San Diego, CA: Hughes Aircraft Company.

Nies, S. 1986. The Ada Object-Oriented Approach. *Proceedings of the First International Conference on Ada Programming Language Applications for the NASA Space Station.* Houston, TX: NASA Lyndon B. Johnson Space Center.

Ossher, H. 1987. A Mechanism for Specifying the Structure of Large, Layered, Systems. In *Research Directions in Object-Oriented Programming*, ed. B. Schriver and P. Wegner. Cambridge, MA: The MIT Press.

Page-Jones, M., Constantine, L., and Weiss, S. October 1990. Modeling Object-Oriented Systems: The Uniform Object Notation. *Computer Language* vol. 7(10).

Parnas, D. 1979. On the Criteria to Be Used in Decomposing Systems into Modules. *Classics in Software Engineering*, ed. E. Yourdon. New York, NY: Yourdon Press.

Parnas, D., Clements, P., and Weiss, D. March 1985. The Modular Structure of Complex Systems. *IEEE Transactions on Software Engineering* vol. SE-11(3).

Pasik, A., and Schor, M. January 1984. Object-Centered Representation and Reasoning. *SIGART Newsletter* no. 87.

Rajlich, V., and Silva, J. 1987. *Two Object-Oriented Decomposition Methods.* Detroit, MI: Wayne State University.

Ramamoorthy, C., and Sheu, P. Fall 1988. Object-Oriented Systems. *IEEE Expert* vol. 3(3).

Reenskaug, T. August 1981. User-Oriented Descriptions of Smalltalk Systems. *Byte* vol. 6(8).

Reiss, S. 1987. An Object-Oriented Framework for Conceptual Programming. In *Research Directions in Object-Oriented Programming*, ed. B. Schriver and P. Wegner. Cambridge, MA: The MIT Press.

Richter, C. August 1986. An Assessment of Structured Analysis and Structured Design. *Software Engineering Notes* vol. 11(4).

Rine, D. October 1987. A Common Error in the Object Structure of Object-Oriented Methods. *Software Engineering Notes* vol. 12(4).

Rosenberg, D., and Jennett, P. July 1992. Object-Oriented Analysis and Design Methods. *Frameworks* vol. 6(4).

Ross, R. 1987. *Entity Modeling: Techniques and Application.* Boston, MA: Database Research Group.

Rosson, M., and Gold, E. October 1989. Problem-Solution Mapping in Object-Oriented Design. *SIGPLAN Notices* vol. 24(10).

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

Sahraoui, A. 1987. *Towards a Design Approach Methodology Combining OOP and Petri Nets for*

Software Production. Toulouse, France: Laboratoire d'Automatique et d'analyses des systemes du C.N.R.S.

Sakai, H. 1983. A Method for Entity-Relationship Behavior Modeling. In *Entity-Relationship Approach to Software Engineering*, ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.

Seidewitz, E. May 1985. *Object Diagrams*. Greenbelt, MD: NASA Goddard Space Flight Center.

Seidewitz, E., and Stark, M. 1986a. Towards a General Object-Oriented Software Development Methodology. *Proceedings of the First International Conference on Ada Programming Language Applications for the NASA Space Station*. Houston, TX: NASA Lyndon B. Johnson Space Center.

———. August 1986b. *General Object-Oriented Software Development*. Report SEL-86-002. Greenbelt, MD: NASA Goddard Space Flight Center.

———. July/August 1987. Towards a General Object-Oriented Design Methodology. *Ada Letters* vol. 7(4).

———. 1988. *An Introduction to General Object-Oriented Software Development*. Rockville, MD: Millennium Systems.

Shilling, J., and Sweeney, P. October 1989. Three Steps to Views: Extending the Object-Oriented Paradigm. *SIGPLAN Notices* vol. 24(10).

Shlaer, S., Mellor, S., and Hywari, W. 1990. *OODLE: A Language-Independent Notation for Object-Oriented Design*. Berkeley, CA: Project Technology, California.

Shumate, K. 1987. *Layered Virtual Machine/Object-Oriented Design*. San Diego, CA: Hughes Aircraft Company.

Smith, M., and Tockey, S. 1988. *An Integrated Approach to Software Requirements Definition Using Objects*. Seattle, WA: Boeing Commercial Airplane Support Division.

Solsi, S., and Jones, E. March/April 1991. Simple Yet Complete Heuristics for Transforming Data Flow Diagrams into Booch Style Diagrams. *Ada Letters* vol. XI(2).

Song, X. May 1992. *Comparing Software Design Methodologies Through Process Modeling*. Irvine, CA: University of California.

Stark, M. April 1986. *Abstraction Analysis: From Structured Analysis to Object-Oriented Design*. Greenbelt, MD: NASA Goddard Space Flight Center.

Strom, R. October 1986. A Comparison of the Object-Oriented and Process Paradigms. *SIGPLAN Notices* vol. 21(10).

Teledyne Brown Engineering. October 1987. *Software Methodology Catalog*, Report MC87-COMM/ADP-0036. Tinton Falls, NJ.

The Fusion Object-Oriented Analysis and Design Method. May 1992. Bristol, England: Hewlett Packard Laboratories.

Thomas, D. May/June 1989. In Search of an Object-Oriented Development Process. *Journal of*

Object-Oriented Programming vol. 2(1).

Wahl, S. December 13, 1988. Introduction to Object-Oriented Software. *C++ Tutorial Program of the USENIX Conference*. Denver, CO: USENIX Association.

Walters, N. July/August 1991. An Ada Object-Based Analysis and Design Approach. *Ada Letters* vol. XI(5).

Wasserman, T., Pircher, P., and Muller, R. December 1988. *An Object-Oriented Structured Design Method for Code Generation*. San Francisco, CA: Interactive Development Environments.

———. Summer 1989. Concepts of Object-Oriented Structured Design. *American Programmer* vol. 2(7–8).

———. March 1990. The Object-Oriented Structured Design Notation for Software Design Representation. *IEEE Computer* vol. 23(3).

Webster, D. December 1988. Mapping the Design Information Representation Terrain. *IEEE Spectrum* vol. 21(12).

Williams, L. 1986. *The Object Model in Software Engineering*. Boulder, CO: Software Engineering Research.

Wirfs-Brock, R., and Wilkerson, B. October 1989. Object-Oriented Design: A Responsibility-Driven Approach. *SIGPLAN Notices* vol. 24(10).

Wirfs-Brock, R., Wilkerson, B., and Wiener, L. 1990. *Designing Object-Oriented Software*. Englewood Cliffs, NJ: Prentice Hall.

Xong, X., and Osterweil, L. June 1992. *A Detailed Objective Comparison and Integration of Two Object-Oriented Design Methodologies*. Irvine, CA: University of California.

Yau, S., and Tsai, J. June 1986. A Survey of Software Design Techniques. *IEEE Transactions on Software Engineering* vol. SE-12(6).

Zachman, J. 1987. A Framework for Information Systems Architecture. *IBM Systems Journal* vol. 26(3).

Zimmerman, R. 1983. Phases, Methods, and Tools—A Triad of System Development. In *Entity-Relationship Approach to Software Engineering*, ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.

G. 面向对象编程

Ada and C++: Business Case Analysis. July 1991. Washington, DC: Deputy Assistant Secretary of the Air Force.

Adams, S. July 1986. MetaMethods: The MVC Paradigm. *HOOPLA: Hooray for Object-Oriented Programming Languages* vol. 1(4). Everette, WA: Object-Oriented Programming for Smalltalk Applications Developers Association.

Agha, G. October 1986. An Overview of Actor Languages. *SIGPLAN Notices* vol. 21(10).

———. 1988. *Actors: A Model of Concurrent Computation in Distributed Systems*. Cambridge, MA: The MIT Press.

Agha, G., and Hewitt, C. 1987. Actors: A Conceptual Foundation for Concurrent Object-Oriented Programming. In *Research Directions in Object-Oriented Programming*, ed. B. Schriver and P. Wegner. Cambridge, MA: The MIT Press.

Aksit, M., and Tripathi, A. September 1988. Data Abstraction Mechanisms in Sina/st. *SIGPLAN Notices* vol. 23(11).

Albano, A. June 1983. Type Hierarchies and Semantic Data Models. *SIGPLAN Notices* vol. 18(6).

Almes, G., Black, A., Lazowska, E., and Noe, J. January 1985. The Eden System: A Technical Review. *IEEE Transactions on Software Engineering* vol. SE-11(1).

Alpert, S., Woyak, S., Shrobe, H., and Arowood, L. December 1990. Object-Oriented Programming in AI. *IEEE Expert* vol. 5(6).

Althoff, J. August 1981. Building Data Structures in the Smalltalk-80 System. *Byte* vol. 6(8).

Ambler, A. 1980. Gypsy: A Language for Specification and Implementation of Verifiable Programs. In *Programming Language Design*, ed. A. Wasserman. New York, NY: Computer Society Press.

America, P. 1987. POOL-T: A Parallel Object-Oriented Language. In *Object-Oriented Concurrent Programming*, ed. Yonezawa and M. Tokoro. Cambridge, MA: The MIT Press.

Apple Computer. 1989. *MacApp: The Expandable Macintosh Application*, version 2.0B9. Cupertino, CA.

———. *Macintosh Programmer's Workshop Pascal 3.0 Reference*. Cupertino, CA. AT&T Bell Laboratories. 1989. *UNIX System V ATT C++ Language System, Release 2.0 Library Manual*. Murray Hill, NJ.

———. *UNIX System V ATT C++ Language System, Release 2.0 Product Reference Manual*. Murray Hill, NJ.

———. *UNIX System V ATT C++ Language System, Release 2.0 Release Notes*. Murray Hill, NJ.

———. *UNIX System V ATT C++ Language System, Release 2.0 Selected Readings*. Murray Hill, NJ.

Attardi, G. 1987. Concurrent Strategy Execution in Omega. In *Object-Oriented Concurrent Programming*, ed. Yonezawa and M. Tokoro. Cambridge, MA: The MIT Press.

Bach, I. November/December 1982. On the Type Concept of Ada. *Ada Letters* vol. II(3).

Badrinath, B., and Ramamritham, K. May 1988. Synchronizing Transactions on Objects. *IEEE Transactions on Computers* vol. 37(5).

Ballard, M., Maier, D., and Wirfs-Brock, A. November 1986. QUICKTALK: A Smalltalk-80 Dialect for Defining Primitive Methods. *SIGPLAN Notices* vol. 21(11).

Beaudet, P., and Jenkins, M. June 1988. Simulating the Object-Oriented Paradigm in Nial. *SIGPLAN Notices* vol. 23(6).

Bennett, J. October 1987. The Design and Implementation of Distributed Smalltalk. *SIGPLAN Notices* vol. 22(12).

Bergin, J., and Greenfield, S. March 1988. What Does Modula-2 Need to Fully Support Object-Oriented Programming? *SIGPLAN Notices* vol. 23(3).

Bhaskar, K. October 1983. How Object-Oriented Is Your System? *SIGPLAN Notices* vol. 18(10).

Birman, K., Joseph, T., Raeuchle, T., and Abbadi, A. June 1985. Implementing Fault-Tolerant Distributed Objects. *IEEE Transactions on Software Engineering* vol. SE-11(6).

Birtwistle, G., Dahl, O-J., Myhrhaug, B., and Nygard, K. 1979. *Simula begin*. Lund, Sweden: Studentlitteratur.

Black, A., Hutchinson, N., Jul, E., and Levy, H. November 1986. Object Structure in the Emerald System. *SIGPLAN Notices* vol. 21(11).

Black, A., Hutchinson, N., Jul, E., Levy, H., and Carter, L. July 1986. *Distribution and Abstract Types in Emerald*. Report 86-02-04. Seattle, WA: University of Washington.

Blaschek, G. 1989. Implementation of Objects in Modula-2. *Structured Programming* vol. 10(3).

Blaschek, G., Pomberger, G., and Stritzinger, A. 1989. A Comparison of Object-Oriented Programming Languages. *Structured Programming* vol. 10(4).

Block, F., and Chan, N. October 1989. An Extended Frame Language. *SIGPLAN Notices* vol. 24(10).

Bobrow, D. November 1984. *If Prolog Is the Answer, What Is the Question?* Palo Alto, CA: Xerox Palo Alto Research Center.

———. 1985. An Overview of KRL, a Knowledge Representation Language. In *Readings in Knowledge Representation*, ed. R. Brachman and H. Levesque. Los Altos, CA: Morgan Kaufmann.

Bobrow, D., DeMichiel, L., Gabriel, R., Keene, S., Kiczales, G., and Moon, D. September 1988. Common Lisp Object System Specification X3J13 Document 88-002R. *SIGPLAN Notices* vol. 23.

Bobrow, D., Kahn, K., Kiczales, G., Masinter, L., Stefik, M., and Zdybel, F. August 1985. *COMMONLOOPS: Merging Common Lisp and Object-Oriented Programming*, Report ISL-85-8. Palo Alto, CA: Xerox Palo Alto Research Center, Intelligent Systems Laboratory.

Borgida, A. January 1985. Features of Languages for the Development of Information Systems at the Conceptual Level. *IEEE Software* vol. 2(1).

———. October 1986. Exceptions in Object-Oriented Languages. *SIGPLAN Notices* vol. 21(10).

Borning, A., and Ingalls, D. 1982a. A Type Declaration and Inference System for Smalltalk. Palo Alto, CA: Xerox Palo Alto Research Center.

———. 1982b. Multiple Inheritance in Smalltalk-80. *Proceedings of the National Conference on Artificial Intelligence*. Menlo Park, CA: AAAI.

Bos, J. September 1987. PCOL—A Protocol-Constrained Object Language. *SIGPLAN Notices* vol.

22(9).

Briot, J., and Cointe, P. October 1989. Programming with Explicit Metaclasses in Smalltalk. *SIGPLAN Notices* vol. 24(10).

Buzzard, G., and Mudge, T. 1987. Object-Based Computing and the Ada Programming Language. In *Object-Oriented Computing: Concepts* vol. 1, ed. G. Peterson. New York, NY: Computer Society Press of the IEEE.

Canning, P., Cook, W., Hill, W., and Olthoff, W. October 1989. Interfaces for Strongly Typed Object-Oriented Programming. *SIGPLAN Notices* vol. 24(10).

Caudill, P., and Wirfs-Brock, A. November 1986. A Third Generation Smalltalk-80 Implementation. *SIGPLAN Notices* vol. 21(11).

Chambers, C., Ungar, D., and Lee, E. October 1989. An Efficient Implementation of Self, a Dynamically Typed Object-Oriented Language Based on Prototypes. *SIGPLAN Notices* vol. 24(10).

Chang, S. 1990. *Visual Languages and Visual Programming*. New York, NY: Plenum Press.

Chin, R., and Chanson, S. March 1991. Distributed Object-Based Programming Systems. *ACM Computing Surveys* vol. 23(1).

Clark, K. December 1988. PARLOG and Its Application. *IEEE Transactions on Software Engineering* vol. 14(12).

Cleaveland, C. 1980. Programming Languages Considered as Abstract Data Types. *Communications of the ACM.*

Coad, P., and Nicola, J. 1993. *Object-Oriented Programming*. Englewood Cliffs, NJ: Yourdon Press.

Cointe, P. October 1987. Metaclasses Are First Class: The ObjVlisp Model. *SIGPLAN Notices* vol. 22(12).

Connor, R., Dearle, A., Morrison, R., and Brown, A. October 1989. An Object Addressing Mechanism for Statically Typed Languages with Multiple Inheritance. *SIGPLAN Notices* vol. 24(10).

Conrad, J., Dengler, P., Francis, B., Glynn, J., Harvey, B., Hollis, B., Ramachandran, R., Schenken, J., Short, S., and Ullman, C. 2001. *Introducing .NET*. Birmingham, United Kingdom: Wrox Press.

Conroy, T., and Pelegri-Llopart, E. 1983. An Assessment of Method-Lookup Caches for Smalltalk-80 Implementations. In *Smalltalk-80: Bits of History, Words of Advice*, ed. G. Krasner. Reading, MA: Addison-Wesley.

Coplien, J. 1992. *Advanced C++ Programming Styles and Idioms*. Reading, MA: Addison-Wesley.

Corradi, A., and Leonardi, L. December 1988. The Role of Opaque Types in Building Abstractions. *SIGPLAN Notices* vol. 23(12).

Cox, B. January 1983. The Object-Oriented Pre-compiler. *SIGPLAN Notices* vol. 18(1).

———. October/November 1983. Object-Oriented Programming in C. *Unix Review*.

———. January 1984. Message/Object Programming: An Evolutionary Change in Programming Technology. *IEEE Software* vol. 1(1).

———. February/March 1984. Object-Oriented Programming: A Power Tool for Software Craftsmen. *Unix Review*.

———. 1986. *Object-Oriented Programming: An Evolutionary Approach*. Reading, MA: Addison-Wesley.

Cox, B., and Hunt, B. August 1986. Objects, Icons, and Software-ICs. *Byte* vol. 11(8).

Cox, P., and Pietrzykowski, T. March 1989. *Prograph: A Pictorial View of Object-Oriented Programming*. Nova Scotia, Canada: Technical University of Nova Scotia.

deJong, P. October 1986. Compilation into Actors. *SIGPLAN Notices* vol. 21(10).

Deutsch, P. August 1981. Building Control Structures in the Smalltalk-80 System. *Byte* vol. 6(8).

———. 1983. Efficient Implementation of the Smalltalk-80 System. *Proceedings of the 11th Annual ACM Symposium on the Principles of Programming Languages*. New York, NY: Association of Computing Machinery.

Dewhurst, S., and Stark, K. 1989. *Programming in C++*. Englewood Cliffs, NJ: Prentice Hall.

Diederich, J., and Milton, J. May 1987. Experimental Prototyping in Smalltalk. *IEEE Software* vol. 4(3).

Dixon, R., McKee, T., Schweizer, P., and Vaughn, M. October 1989. A Fast Method Dispatcher for Compiled Languages with Multiple Inheritance. *SIGPLAN Notices* vol. 24(10).

Dony, C. August 1988. An Object-Oriented Exception Handling System for an Object-Oriented Language. *Proceedings of ECOOP'88: European Conference on Object-Oriented Programming*. New York, NY: Springer-Verlag.

Duff, C. August 1986. Designing an Efficient Language. *Byte* vol. 11(8).

Dussud, P. October 1989. TICLOS: An Implementation of CLOS for the Explorer Family. *SIGPLAN Notices* vol. 24(10).

Eccles, J. 1988. Porting from Common Lisp with Flavors to C++. *Proceedings of USENIX C++ Conference*. Berkeley, CA: USENIX Association.

Edelson, D. September 1987. How Objective Mechanisms Facilitate the Development of Large Software Systems in Three Programming Languages. *SIGPLAN Notices* vol. 22(9).

Ellis, M., and Stroustrup, B. 1990. *The Annotated C++ Reference Manual*. Reading, MA: Addison-Wesley.

Endres, T. May 1985. Clascal—An Object-Oriented Pascal. *Computer Language* vol. 2(5).

Entsminger, G. 1990. *The Tao of Objects*. Redwood City, CA: M & T Books.

Filman, R. October 1987. Retrofitting Objects. *SIGPLAN Notices* vol. 22(12).

Finzer, W., and Gould, L. June 1984. Programming by Rehearsal. *Byte* vol. 9(6).

Foote, B., and Johnson, R. October 1989. Reflective Facilities in Smalltalk-80. *SIGPLAN Notices* vol. 24(10).

Freeman-Benson, B. October 1989. A Module Mechanism for Constraints in Smalltalk. *SIGPLAN Notices* vol. 24(10).

Fukunaga, K., and Jirose, S. November 1986. An Experience with a Prolog-Based Object-Oriented Language. *SIGPLAN Notices* vol. 21(11).

Gabriel, R., White, J., and Bobrow, D. September 1991. CLOS Integrating Object-Oriented and Functional Programming. *Communications of the ACM* vol. 34(9).

Goldberg, A. August 1981. Introducing the Smalltalk-80 System. *Byte* vol. 6(8).

———. September 1988. Programmer as Reader. *IEEE Software* vol. 4(5).

Goldberg, A., and Kay, A. March 1976. *Smalltalk-72 Instruction Manual.* Palo Alto, CA: Xerox Palo Alto Research Center.

———. 1977. *Methods for Teaching the Programming Language Smalltalk.* Report SSL 77-2. Palo Alto, CA: Xerox Palo Alto Research Center.

Goldberg, A., and Pope, S. Summer 1989. Object-Oriented Programming Is Not Enough. *American Programmer* vol. 2(7–8).

Goldberg, A., and Robson, D. 1983. *Smalltalk-80: The Language and Its Implementation.* Reading, MA: Addison-Wesley.

———. 1989. *Smalltalk-80: The Language.* Reading, MA: Addison-Wesley.

Goldberg, A., and Ross, J. August 1981. Is the Smalltalk-80 System for Children? *Byte* vol. 6(8).

Goldstein, T. May 1989. The Object-Oriented Programmer. *The C++ Report* vol. 1(5).

Gonsalves, G., and Silvestri, A. December 1986. Programming in Smalltalk-80: Observations and Remarks from the Newly Initiated. *SIGPLAN Notices* vol. 21(12).

Gorlen, K. 1989. An Introduction to C++. In *UNIX System V ATT C++ Language System, Release 2.0 Selected Readings.* Murray Hill, NJ: ATT Bell Laboratories.

Gorlen, K., Orlow, S., and Plexico, P. 1990. *Data Abstraction and Object-Oriented Programming in C++.* New York, NY: John Wiley and Sons.

Gougen, J., and Meseguer, J. 1987. Unifying Functional, Object-Oriented, and Relational Programming with Logical Semantics. In *Research Directions in Object-Oriented Programming,* ed. B. Schriver and P. Wegner. Cambridge, MA: The MIT Press.

Graube, N. August 1988. Reflexive Architecture: From ObjVLisp to CLOS. *Proceedings of ECOOP'88: European Conference on Object-Oriented Programming.* New York, NY: Springer-Verlag.

Grogono, P. November 1989. Polymorphism and Type Checking in Object-Oriented Languages. *SIGPLAN Notices* vol. 24(11).

———. 1991. Issues in the Design of an Object-Oriented Programming Language. *Structured*

Programming vol. 12(1).

Hagmann, R. 1983. Preferred Classes: A Proposal for Faster Smalltalk-80 Execution. In *Smalltalk-80: Bits of History, Words of Advice*, ed. G. Krasner. Reading, MA: Addison-Wesley.

Hailpern, B., and Nguyen, V. 1987. A Model for Object-Based Inheritance. In *Research Directions in Object-Oriented Programming*, ed. B. Schriver and P. Wegner. Cambridge, MA: The MIT Press.

Halbert, D., and O'Brien, P. September 1988. Using Types and Inheritance in Object-Oriented Programming. *IEEE Software* vol. 4(5).

Halstead, R. 1987. Object Management on Distributed Systems. In *Object-Oriented Computing: Implementations* vol. 2, ed. G. Peterson. New York, NY: Computer Society Press of the IEEE.

Harland, D., Szyplewski, M., and Wainwright, J. October 1985. An Alternative View of Polymorphism. *SIGPLAN Notices* vol. 20(10).

Hendler, J. October 1986. Enhancement for Multiple Inheritance. *SIGPLAN Notices* vol. 21(10).

Hines, T., and Unger, E. 1986. *Conceptual Object-Oriented Programming*. Manhattan, KS: Kansas State University.

Ingalls, D. 1978. The Smalltalk-76 Programming System Design and Implementation. *Proceedings of the Fifth Annual ACM Symposium on Principles of Programming Languages*. New York, NY: Association of Computing Machinery.

———. August 1981a. Design Principles Behind Smalltalk. *Byte* vol. 6(8).

———. August 1981b. The Smalltalk Graphics Kernel. *Byte* vol. 6(8).

———. 1983. The Evolution of the Smalltalk Virtual Machine. In *Smalltalk-80: Bits of History, Words of Advice*, ed. G. Krasner. Reading, MA: Addison-Wesley.

———. November 1986. A Simple Technique for Handling Multiple Polymorphism. *SIGPLAN Notices* vol. 21(11).

Ishikawa, Y., and Tokoro, M. 1987. Orient84/K: An Object-Oriented Concurrent Programming Language for Knowledge Representation. In *Object-Oriented Concurrent Programming*, ed. Yonezawa and M. Tokoro. Cambridge, MA: The MIT Press.

Jackson, M. May 1988. Objects and Other Subjects. *SIGPLAN Notices* vol. 23(5).

Jacky, J., and Kalet, I. September 1987. An Object-Oriented Programming Discipline for Standard Pascal. *Communications of the ACM* vol. 30(9).

Jacobson, I. November 1986. Language Support for Changeable, Large, Real-Time Systems. *SIGPLAN Notices* vol. 21(11).

Jeffery, D. February 1989. Object-Oriented Programming in ANSI C. *Computer Language*.

Jenkins, M., and Glasgow, J. January 1986. Programming Styles in Nial. *IEEE Software* vol. 3(1).

Johnson, R. November 1986. Type-Checking Smalltalk. *SIGPLAN Notices* vol. 21(11).

Johnson, R., Graver, J., and Zurawski, L. September 1988. TS: An Optimizing Compiler for Smalltalk.

SIGPLAN Notices vol. 23(11).

Kaehler, T. November 1986. Virtual Memory on a Narrow Machine for an Object-Oriented Language. *SIGPLAN Notices* vol. 21(11).

Kaehler, T., and Patterson, D. 1986. *A Taste of Smalltalk*. New York, NY: W. W. Norton.

———. August 1986. A Small Taste of Smalltalk. *Byte* vol. 11(8).

Kahn, K., Tribble, E., Miller, M., and Bobrow, D. November 1986. Objects in Concurrent Logic Programming Languages. *SIGPLAN Notices* vol. 21(11).

———. 1987. Vulcan: Logical Concurrent Objects. In *Research Directions in Object-Oriented Programming*, ed. B. Schriver and P. Wegner. Cambridge, MA: The MIT Press.

Kaiser, G., and Garlan, D. October 1987. MELDing Data Flow and Object-Oriented Programming. *SIGPLAN Notices* vol. 22(12).

Kalme, C. March 27, 1986. *Object-Oriented Programming: A Rule-Based Perspective*. Los Angeles, CA: Inference Corporation.

Kay, A. *New Directions for Novice Programming in the 1980s*. Palo Alto, CA: Xerox Palo Alto Research Center.

Keene, S. 1989. *Object-Oriented Programming in Common Lisp*. Reading, MA: Addison-Wesley.

Kelly, K., Rischer, R., Pleasant, M., Steiner, D., McGrew, C., Rowe, J., and Rubin, M. March 30, 1986. *Textual Representations of Object-Oriented Programs for Future Programmers*. Palo Alto, CA: Xerox AI Systems.

Kempf, R. October 1987. Teaching Object-Oriented Programming with the KEE System. *SIGPLAN Notices* vol. 22(12).

Kempf, J., Harris, W., D'Souza, R., and Snyder, A. October 1987. Experience with CommonLoops. *SIGPLAN Notices* vol. 22(12).

Khoshafian, S., and Copeland, G. November 1986. Object Identity. *SIGPLAN Notices* vol. 21(11).

Kiczales, G., Rivieres, J., and Bobrow, D. 1991. *The Art of the Metaobject Protocol*. Cambridge, MA: The MIT Press.

Kilian, M. April 1987. *An Overview of the Trellis/Owl Compiler*. Hudson, MA: Digital Equipment Corporation.

Kimminau, D., and Seagren, M. 1987. *Comparison of Two Prototype Developments Using Object-Based Programming*. Naperville, IL: AT&T Bell Laboratories.

Knowledge Systems Corporation. 1987. *PluggableGauges Version 1.0 User Manual*. Cary, NC.

Knudsen, J., and Madsen, O. August 1988. Teaching Object-Oriented Programming Is More Than Teaching Object-Oriented Programming Languages. *Proceedings of ECOOP'88: European Conference on Object-Oriented Programming*. New York, NY: Springer-Verlag.

Knudsen, J. August 1988. Name Collision in Multiple Classification Hierarchies. *Proceedings of ECOOP'88: European Conference on Object-Oriented Programming*. New York, NY:

Springer-Verlag.

Korson, T., and McGregor, J. September 1990. Understanding Object-Oriented: A Unifying Paradigm. *Communications of the ACM* vol. 33(9).

Koshmann, T., and Evens, M. July 1988. Bridging the Gap Between Object-Oriented and Logic Programming. *IEEE Software* vol. 5(4).

Koskimies, K., and Paakki, J. July 1987. TOOLS: A Unifying Approach to Object-Oriented Language Interpretation. *SIGPLAN Notices* vol. 22(7).

Krasner, G. August 1981. The Smalltalk-80 Virtual Machine. *Byte* vol. 6(8).

———, ed. 1983. *Smalltalk-80: Bits of History, Words of Advice*. Reading, MA: Addison-Wesley.

Krasner, G., and Pope, S. August/September 1988. A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. *Journal of Object-Oriented Programming* vol. 1(3).

Kristensen, B., Madsen, O., Moller-Pedersen, B., and Nygaard, K. 1987. The BETA Programming Language. In *Research Directions in Object-Oriented Programming*, ed. B. Schriver and P. Wegner. Cambridge, MA: The MIT Press.

LaLonde, W. April 1989. Designing Families of Data Types Using Exemplars. *ACM Transactions on Programming Languages and Systems* vol. 11(2).

LaLonde, W., and Pugh, J. 1990. *Inside Smalltalk* vol. 1 and 2. Englewood Cliffs, NJ: Prentice Hall.

LaLonde, W., Thomas, D., and Pugh, J. November 1986. An Examplar Based Smalltalk. *SIGPLAN Notices* vol. 21(11).

Lang, K., and Peralmutter, B. November 1986. Oaklisp: An Object-Oriented Scheme with First Class Types. *SIGPLAN Notices* vol. 21(11).

Laursen, J., and Atkinson, R. October 1987. Opus: A Smalltalk Production System. *SIGPLAN Notices* vol. 22(12).

Lieberherr, K., and Holland, I. March 1989. Formulations and Benefits of the Law of Demeter. *SIGPLAN Notices* vol. 24(3).

———. September 1989. Assuring Good Style for Object-Oriented Programs. *IEEE Software* vol. 6(5).

Lieberherr, K., Holland, I., Lee, G., and Riel, A. June 1988. An Objective Sense of Style. *IEEE Computer* vol. 21(6).

Lieberman, H. November 1986. Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems. *SIGPLAN Notices* vol. 21(11).

———. 1987. Concurrent Object-Oriented Programming in Act 1. In *Object-Oriented Concurrent Programming*, ed. A. Yonezawa and M. Tokoro. Cambridge, MA: The MIT Press.

Lieberman, H., Stein, L., and Ungar, D. May 1988. Of Types and Prototypes: The Treaty of Orlando. *SIGPLAN Notices* vol. 23(5).

Lim, J., and Johnson, R. April 1989. The Heart of Object-Oriented Concurrent Programming. *SIGPLAN Notices* vol. 24(4).

Linowes, J. August 1988. It's an Attitude. *Byte* vol. 13(8).

Lippman, S., Lajoie, J., and Moo, B. 2005. *C++ Primer*. Fourth Edition. Boston, MA: Addison-Wesley.

Liskov, B., Atkinson, R., Bloom, T., Moss, E., Schaffert, C., Scheifler, R., and Snyder, R. 1981. *CLU Reference Manual*. New York, NY: Springer-Verlag.

Liskov, B., Snyder, A., Atkinson, R., and Schaffert, C. 1980. Abstraction Mechanisms in CLU. In *Programming Language Design*, ed. A. Wasserman. New York, NY: Computer Society Press.

Liu, C. March 1991. On the Object-Orientedness of C++. *SIGPLAN Notices* vol. 26(3).

Lujun, S., and Zhongxiu. August 1987. An Object-Oriented Programming Language for Developing Distributed Software. *SIGPLAN Notices* vol. 22(8).

MacLennan, B. 1987. Values and Objects in Programming Languages. In *Object-Oriented Computing: Concepts* vol. 1, ed. G. Peterson. New York, NY: Computer Society Press of the IEEE.

Madsen, O. 1987. Block Structure and Object-Oriented Languages. In *Research Directions in Object-Oriented Programming*, ed. B. Schriver and P. Wegner. Cambridge, MA: The MIT Press.

Madsen, O., and Moller-Pedersen, B. August 1988. What Object-Oriented Programming May Be—And What It Does Not Have to Be. *Proceedings of ECOOP'88: European Conference on Object-Oriented Programming*. New York, NY: Springer-Verlag.

———. October 1989. Virtual Classes: A Powerful Mechanism in Object-Oriented Programming. *SIGPLAN Notices* vol. 24(10).

Manci, D. 1990. *Use of Metrics to Evaluate C++*. Liberty Corner, NJ: AT&T Bell Laboratories.

Mannino, M., Choi, I., and Batory, D. November 1990. The Object-Oriented Functional Data Language. *IEEE Transactions on Software Engineering* vol. 16(11).

Marcus, R. November 1985. Generalized Inheritance. *SIGPLAN Notices* vol. 20(11).

Markowitz, V., and Raz, Y. 1983. Eroll: An Entity-Relationship, Role-Oriented Query Language. In *Entity-Relationship Approach to Software Engineering*, ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.

Masini, G., Napoli, A., Colnet, D., Leonard, D., and Tompre, K. 1991. *Object-Oriented Languages*. London, England: Academic Press.

Mellender, F. October 1988. An Integration of Logic and Object-Oriented Programming. *SIGPLAN Notices* vol. 23(10).

Methfessel, R. April 1987. Implementing an Access and Object-Oriented Paradigm in a Language That Supports Neither. *SIGPLAN Notices* vol. 22(4).

Meyer, B. November 1986. Genericity versus Inheritance. *SIGPLAN Notices* vol. 21(11).

———. February 1987. Eiffel: Programming for Reusability and Extendability. *SIGPLAN Notices* vol. 22(2).

———. November/December 1988. Harnessing Multiple Inheritance. *Journal of Object-Oriented Programming* vol. 1(4).

Micallef, J. April/May 1988. Encapsulation, Reusability, and Extensibility in Object-Oriented Programming Languages. *Journal of Object-Oriented Programming* vol. 1(1).

Microsoft C++ Tutorial. 1992. Redmond, WA: Microsoft Corporation.

Microsoft Windows Guide to Programming. 1992. Redmond, WA: Microsoft Corporation.

Minsky, N., and Rozenshtein, D. October 1987. A Law-Based Approach to Object-Oriented Programming. *SIGPLAN Notices* vol. 22(12).

———. October 1989. Controllable Delegation: An Exercise in Law-Governed Systems. *SIGPLAN Notices* vol. 24(10).

Miranda, E. October 1987. BrouHaHa—A Portable Smalltalk Interpreter. *SIGPLAN Notices* vol. 22(12).

Mittal, S., Bobrow, D., and Kahn, K. November 1986. Virtual Copies: At the Boundary Between Classes and Instances. *SIGPLAN Notices* vol. 21(11).

Moon, D. November 1986. Object-Oriented Programming with Flavors. *SIGPLAN Notices* vol. 21(11).

Morrison, R., Dearle, A., Connor, R., and Brown, A. July 1991. An Ad Hoc Approach to the Implementation of Polymorphism. *ACM Transactions on Programming Languages and Systems* vol. 13(3).

Mossenbock, H., and Templ, J. 1989. Object Oberon—A Modest Object-Oriented Language. *Structured Programming* vol. 10(4).

Mudge, T. March 1985. Object-Based Computing and the Ada Language. *IEEE Computer* vol. 18(3).

Murray, R. 1990. *C++ Tactics*. Liberty Corner, NJ: AT&T Bell Laboratories.

Nelson, M. October 1991. Concurrency and Object-Oriented Programming. *SIGPLAN Notices* vol. 26(10).

Nierstrasz, O. October 1987. Active Objects in Hybrid. *SIGPLAN Notices* vol. 22(12).

Novak, G. June 1983. Data Abstraction in GLISP. *SIGPLAN Notices* vol. 18(6).

———. Fall 1983. GLISP: A Lisp-Based Programming System with Data Abstraction. *AI Magazine* vol. 4(3).

Nygaard, K. October 1986. Basic Concepts in Object-Oriented Programming. *SIGPLAN Notices* vol. 21(10).

Nygaard, K., and Dahl, O-J. 1981. The Development of the Simula Languages. In *History of Programming Languages*, ed. R. Wexelblat. New York, NY: Academic Press.

O'Brien, P. November 15, 1985. *Trellis Object-Based Environment: Language Tutorial*. Hudson, MA: Digital Equipment Corporation.

O'Grady, F. July/August 1990. Is There Life After COBOL? *American Programmer* vol. 3(7–8).

Object-Oriented Programming Workshop. October 1986. *SIGPLAN Notices* vol. 21(10).

Olthoff, W. 1986. *Augmentation of Object-Oriented Programming by Concepts of Abstract Data Type Theory: The ModPascal Experience*. Kaiserslautern, West Germany: University of Kaiserslautern.

Osterbye, K. June/July 1988. Active Objects: An Access-Oriented Framework for Object-Oriented Languages. *Journal of Object-Oriented Programming* vol. 1(2).

Paepcke, A. October 1989. PCLOS: A Critical Review. *SIGPLAN Notices* vol. 24(10).

Parc Place Systems. 1988. *The Smalltalk-80 Programming System Version VI 2.3*. Palo Alto, CA.

Pascoe, G. August 1986. Elements of Object-Oriented Programming. *Byte* vol. 11(8).

———. November 1986. Encapsulators: A New Software Paradigm in Smalltalk-80. *SIGPLAN Notices* vol. 21(11).

Perez, E. September/October 1988. Simulating Inheritance with Ada. *Ada Letters* vol. 8(7).

Peterson, G., ed. 1987. *Object-Oriented Computing Concepts*. New York, NY: Computer Society Press of the IEEE.

Pinson, L., and Wiener, R. 1988. *An Introduction to Object-Oriented Programming and Smalltalk*. Reading, MA: Addison-Wesley.

Pohl, I. 1989. *C++ for C Programmers*. Redwood City, CA: Benjamin/Cummings.

Pokkunuri, B. November 1989. Object-Oriented Programming. *SIGPLAN Notices* vol. 24(11).

Ponder, C., and Bush, B. June 1992. Polymorphism Considered Harmful. *SIGPLAN Notices* vol. 27(6).

Pountain, D. August 1986. Object-Oriented FORTH. *Byte* vol. 11(8).

Proceedings of ECOOP'88: European Conference on Object-Oriented Programming. August 1988. New York, NY: Springer-Verlag.

Proceedings of OOPSLA'86: Object-Oriented Programming Systems, Languages, and Applications. November 1986. *SIGPLAN Notices* vol. 21(11).

Proceedings of OOPSLA'87: Object-Oriented Programming Systems, Languages, and Applications. October 1987. *SIGPLAN Notices* vol. 22(12).

Proceedings of OOPSLA'88: Object-Oriented Programming Systems, Languages, and Applications. September 1988. *SIGPLAN Notices* vol. 23(11).

Proceedings of OOPSLA'89: Object-Oriented Programming Systems, Languages, and Applications. October 1989. *SIGPLAN Notices* vol. 24(10).

Proceedings of OOPSLA'90. Object-Oriented Programming Systems, Languages, and Applications. October 1990. *SIGPLAN Notices* vol. 25(10).

Proceedings of OOPSLA'91. Object-Oriented Programming Systems, Languages, and Applications.

November 1991. *SIGPLAN Notices* vol. 26(11).

Proceedings of OOPSLA'92. Object-Oriented Programming Systems, Languages, and Applications. October 1992. *SIGPLAN Notices* vol. 27(10).

Proceedings of the ACM SIGPLAN Workshop on Object-Based Concurrent Programming. April 1989. *SIGPLAN Notices* vol. 24(4).

Proceedings of the USENIX Association C++ Workshop. November 1987. Berkeley, CA: USENIX Association.

Proceedings of the Workshop on Data Abstraction, Databases, and Conceptual Modeling. 1980. *SIGPLAN Notices* vol. 16(1).

Pugh, J. March 1984. Actors—The Stage Is Set. *SIGPLAN Notices* vol. 19(3).

Rathke, C. 1986. *ObjTalk: Repräsentation von Wissen in einer objektorientierten Sprache.* Stuttgart, West Germany: Institut für Informatik der Universität Stuttgart.

Rentsch, T. September 1982. Object-Oriented Programming. *SIGPLAN Notices* vol. 17(12).

Rettig, M., Morgan, T., Jacobs, J., and Wimberly, D. January 1989. Object-Oriented Programming in AI. *AI Expert.*

Robson, D. August 1981. Object-Oriented Software Systems. *Byte* vol. 6(8).

Rumbaugh, J. October 1987. Relations as Semantic Constructs in an Object-Oriented Language. *SIGPLAN Notices* vol. 22(12).

Russo, V., and Kaplan, S. 1988. A C++ Interpreter for Scheme. *Proceedings of USENIX C++ Conference.* Berkeley, CA: USENIX Association.

Sakkinen, M. August 1988. On the Darker Side of C++. *Proceedings of ECOOP'88: European Conference on Object-Oriented Programming.* New York, NY: Springer-Verlag.

———. December 1988. Comments on "the Law of Demeter" and C++. *SIGPLAN Notices* vol. 23(12).

Saltzer, J. 1979. Naming and Binding of Objects. In *Operating Systems*, ed. R. Bayer et al. New York, NY: Springer-Verlag.

Sandberg, D. November 1986. An Alternative to Subclassing. *SIGPLAN Notices* vol. 21(11).

———. October 1988. Smalltalk and Exploratory Programming. *SIGPLAN Notices* vol. 23(10).

Saunders, J. March/April 1989. A Survey of Object-Oriented Programming Languages. *Journal of Object-Oriented Programming* vol. 1(6).

Schaffert, C., Cooper, T., and Wilpolt, C. November 25, 1985. *Trellis Object-Based Environment: Language Reference Manual.* Hudson, MA: Digital Equipment Corporation.

Schaffert, C., Cooper, T., Bullis, B., Kilian, M., and Wilpolt, C. November 1986. An Introduction to Trellis/Owl. *SIGPLAN Notices* vol. 21(11).

Schmucker, K. 1986a. MacApp: An Application Framework. *Byte* vol. 11(8).

———. 1986b. Object-Oriented Languages for the Macintosh. *Byte* vol. 11(8).

———. 1986c. *Object-Oriented Programming for the Macintosh*. Hasbrouk Heights, NJ: Hayden.

Schriver, B., and Wegner, P., eds. 1987. *Research Directions in Object-Oriented Programming*. Cambridge, MA: The MIT Press.

Seidewitz, E. October 1987. Object-Oriented Programming in Smalltalk and Ada. *SIGPLAN Notices* vol. 22(12).

———. March/April 1992. Object-Oriented Programming with Mixins in Ada. *Ada Letters* vol. XII(2).

Shafer, D. 1988. *Hyper Talk Programming*. Indianapolis, IN: Hayden.

Shah, A., Rumbaugh, J., Hamel, J., and Borsari, R. October 1989. DSM: An Object-Relationship Modeling Language. *SIGPLAN Notices* vol. 24(10).

Shammas, N. October 1988. Smalltalk a la C. *Byte* vol. 13(10).

Shan, Y. October 1989. An Event-Driven Model-View-Controller Framework for Smalltalk. *SIGPLAN Notices* vol. 24(10).

Shapiro, J. 1991. *A C++ Toolkit*. Englewood Cliffs, NJ: Prentice Hall.

Shaw, M. 1981. *ALPHARD: Form and Content*. New York, NY: Springer-Verlag.

Shibayama, E. September 1988. How to Invent Distributed Implementation Schemes of an Object-Based Concurrent Language—A Transformational Approach. *SIGPLAN Notices* vol. 23(11).

Shibayama, E., and Yonezawa, A. 1987. Distributed Computing in ABCL/1. In *Object-Oriented Concurrent Programming*, ed. Yonezawa and M. Tokoro. Cambridge, MA: The MIT Press.

Shopiro, J. December 13, 1988. Programming Techniques with C++. *C++ Tutorial Program of the USENIX Conference*. Denver, CO: USENIX Association.

———. December 1989. An Example of Multiple Inheritance in C++: A Model of the Iostream Library. *SIGPLAN Notices* vol. 24(12).

Simonian, R., and Crone, M. November/December 1988. InnovAda: True Object-Oriented Programming in Ada. *Journal of Object-Oriented Programming* vol. 1(4).

Snyder, A. February 1985. *Object-Oriented Programming for Common Lisp*. Report ATC-85-1. Palo Alto, CA: Hewlett-Packard.

———. November 1986. Encapsulation and Inheritance in Object-Oriented Programming Languages. *SIGPLAN Notices* vol. 21(11).

———. 1987. Inheritance and the Development of Encapsulated Software Components. In *Research Directions in Object-Oriented Programming*, ed. B. Schriver and P. Wegner. Cambridge, MA: The MIT Press.

———. January 1993. The Essence of Objects: Concepts and Terms. *IEEE Software* vol. 10(1).

Software Productivity Solutions. 1988. *Classical-Ada User Manual*. Melbourne, FL.

Stankovic, J. April 1982. Software Communication Mechanisms: Procedure Calls versus Messages.

IEEE Computer vol. 15(4).

Stefik, M., and Bobrow, D. Winter 1986. Object-Oriented Programming: Themes and Variations. *AI Magazine* vol. 6(4).

Stefik, M., Bobrow, D., Mittal, S., and Conway, L. Fall 1983. Knowledge Programming in Loops. *AI Magazine* vol. 4(3).

Stein, L. October 1987. Delegation Is Inheritance. *SIGPLAN Notices* vol. 22(12).

Stroustrup, B. January 1982. Classes: An Abstract Data Type Facility for the C Language. *SIGPLAN Notices* vol. 17(1).

———. October 1986. An Overview of C++. *SIGPLAN Notices* vol. 21(10).

———. 1987. The Evolution of C++. *Proceedings of the USENIX C++ Workshop*. Santa Fe, NM: USENIX Association.

———. November 1987. Possible Directions for C++. *Proceedings of the USENIX C++ Workshop*. Santa Fe, NM: USENIX Association.

———. 1988a. Parameterized Types for C++. *Proceedings of USENIX C++ Conference*. Berkeley, CA: USENIX Association.

———. May 1988b. What Is Object-Oriented Programming? *IEEE Software* vol. 5(3).

———. August 1988c. A Better C? *Byte* vol. 13(8).

———. 1991. *The C+ Programming Language*. Second Edition. Reading, MA: Addison-Wesley.

———. 2000. *The C++ Programming Language*. Special Third Edition. Boston, MA: Addison-Wesley.

Suzuki, N. 1981. Inferring Types in Smalltalk. In *Proceedings of the Eighth Annual Symposium of ACM Principles of Programming Languages*. New York, NY: Association of Computing Machinery.

Suzuki, N., and Terada, M. 1983. Creating Efficient Systems for Object-Oriented Languages. *Proceedings of the 11th Annual ACM Symposium on the Principles of Programming Languages*. New York, NY: Association of Computing Machinery.

Symposium on Actor Languages. October 1980. *Creative Computing*.

Tektronix. 1988. Modular Smalltalk.

Tesler, L. August 1986. Programming Experiences. *Byte* vol. 11(8).

The Smalltalk-80 System. August 1981. *Byte* vol. 6(8).

Thomas, D. March 1989. What's in an Object? *Byte* vol. 14(3).

Tieman, M. May 1, 1988. *User's Guide to GNU C++*. Cambridge, MA: Free Software Foundation.

Tokoro, M., and Ishikawa, Y. October 1986. Concurrent Programming in Orient84/K: An Object-Oriented Knowledge Representation Language. *SIGPLAN Notices* vol. 21(10).

Touati, H. May 1987. Is Ada an Object-Oriented Programming Language? *SIGPLAN Notices* vol. 22(5).

Touretzky, D. 1986. *The Mathematics of Inheritance Systems*. Los Altos, CA: Morgan Kaufman Publishers.

Tripathi, A., and Berge, E. An Implementation of the Object-Oriented Concurrent Programming Language SINA. *Software—Practice and Experience* vol. 19(3).

U.S. Department of Defense. February 1983. *Reference Manual for the Ada Programming Language*. Washington, DC: Ada Joint Program Office.

Ungar, D. September 1988. Are Classes Obsolete? *SIGPLAN Notices* vol. 23(11).

Ungar, D., and Smith, R. October 1987. Self: The Power of Simplicity. *SIGPLAN Notices* vol. 22(12).

van den Bos, J., and Laffra, C. October 1989. PROCOL: A Parallel Object Language with Protocols. *SIGPLAN Notices* vol. 24(10).

Vaucher, J., Lapalme, G., and Malenfant, J. August 1988. SCOOP: Structured Concurrent Object-Oriented Prolog. *Proceedings of ECOOP'88: European Conference on Object-Oriented Programming*. New York, NY: Springer-Verlag.

Warren, S., and Abbe, D. May 1980. Presenting Rosetta Smalltalk. *Datamation*.

Watanabe, T., and Yonezawa, A. September 1988. Reflection in an Object-Oriented Concurrent Language. *SIGPLAN Notices* vol. 23(11).

Wegner, P. October 1987. Dimensions of Object-Based Language Design. *SIGPLAN Notices* vol. 22(12).

———. January 1988. Workshop on Object-Oriented Programming at ECOOP 1987. *SIGPLAN Notices* vol. 23(1).

———. August 1990. Concepts and Paradigms of Object-Oriented Programming. *OOPS Messenger* vol. 1(1).

———. October 1992. Dimensions of Object-Oriented Modeling. *IEEE Computer* vol. 25(10).

Wiener, R. June 1987. Object-Oriented Programming in C++ —A Case Study. *SIGPLAN Notices* vol. 22(6).

Williams, G. Summer 1989. Designing the Future: The Power of Object-Oriented Programming. *American Programmer* vol. 2(7–8).

Wilson, R. November 1, 1987. Object-Oriented Languages Reorient Programming Techniques. *Computer Design* vol. 26(20).

Winblad, A., Edwards, S., and King, D. 1990. *Object-Oriented Software*. Reading, MA: Addison-Wesley.

Winston, P., and Horn, B. 1989. *Lisp*. Third Edition. Reading, MA: Addison-Wesley.

Wirfs-Brock, R., and Wilkerson, B. September 1988. An Overview of Modular Smalltalk. *SIGPLAN Notices* vol. 23(11).

Wirth, N. June 1987. Extensions of Record Types. *SIGCSE Bulletin* vol. 19(2).

———. July 1988a. From Modula to Oberon. *Software—Practice and Experience* vol. 18(7).

———. July 1988b. The Programming Language Oberon. *Software—Practice and Experience* vol. 18(7).

Wolf, W. September 1989. A Practical Comparison of Two Object-Oriented Languages. *IEEE Software* vol. 6(5).

Yokote, Y., and Tokoro, M. November 1986. The Design and Implementation of Concurrent Smalltalk. *SIGPLAN Notices* vol. 21(11).

———. October 1987. Experience and Evolution of Concurrent Smalltalk. *SIGPLAN Notices* vol. 22(12).

Yonezawa, A., and Tokoro, M., eds. 1987. *Object-Oriented Concurrent Programming*. Cambridge, MA: The MIT Press.

Yonezawa, A., Briot, J., and Shibayama, E. November 1986. Object-Oriented Concurrent Programming in ABCL/1. *SIGPLAN Notices* vol. 21(11).

Yonezawa, A., Shibayama, E., Takada, T., and Honda, Y. 1987. Modeling and Programming in an Object-Oriented Concurrent Language ABCL/1. In *Object-Oriented Concurrent Programming*, ed. Yonezawa and M. Tokoro. Cambridge, MA: The MIT Press.

Yourdon, E. February 1990. Object-Oriented COBOL. *American Programmer* vol. 3(2).

———. January 1992. Modeling Magic. *American Programmer* vol. 5(1).

Zave, P. September 1989. A Compositional Approach to Multiparadigm Programming. *IEEE Software* vol. 6(5).

H. 软件工程

Abdel-Hamid, T., and Madnick, S. 1991. *Software Project Dynamics*. Englewood Cliffs, NJ: Prentice Hall.

Abelson, H., and Sussman, G. 1985. *Structure and Interpretation of Computer Programs*. Cambridge, MA: The MIT Press.

Andrews, D., and Leventhal, N. 1993. *FUSION: Integrating IE, CASE, and JAD: A Handbook for Reengineering the Systems Organization*. Englewood Cliffs, NJ: Yourdon Press.

Appleton, D. January 15, 1986. Very Large Projects. *Datamation*.

Aron, J. 1974a. *The Program Development Process: The Individual Programmer*. Vol. 1. Reading, MA: Addison-Wesley.

———. 1974b. *The Program Development Process: The Programming Team*. Vol. 2. Reading, MA: Addison-Wesley.

Babich, W. 1986. *Software Configuration Management*. Reading, MA: Addison-Wesley.

Ben-Ari, M. 1982. *Principles of Concurrent Programming*. Englewood Cliffs, NJ: Prentice Hall.

Berard, E. 1993. *Essays on Object-Oriented Software Engineering*. Englewood Cliffs, NJ: Prentice

Hall.

Berson, A. 1992. Client/Server Architecture. New York, NY: McGraw-Hill.

Berzins, V., and Luqi. 1991. *Software Engineering with Abstractions*. Reading, MA: Addison-Wesley.

Biggerstaff, T., and Perlis, A. 1989. *Software Reusability*. New York, NY: ACM Press.

Bisant, D., and Lyle, J. October 1989. A Two-Person Inspection Method to Improve Programming Productivity. *IEEE Transactions on Software Engineering* vol. 15(10).

Bischofberger, W., and Keller, R. 1989. Enhancing the Software Life Cycle by Prototyping. *Structured Programming*.

Bloom, P. April 1993. Trends in Client-Server/Cooperative Processing Application Development Tools. *American Programmer*.

Boar, B. 1984. *Application Prototyping*. New York, NY: John Wiley and Sons.

Boehm, B. August 1986. A Spiral Model of Software Development and Enhancement. *Software Engineering Notes* vol. 11(4).

———. September 1992. Risk Control. *American Programmer* vol. 5(7).

Boehm, B. 1989. *IEEE Tutorial on Software Risk Management*. IEEE Computer Society Press.

———. October 2002. Software Risk Management: Overview and Recent Developments. *17th International Forum on COCOMO and Software Cost Modeling*. Los Angeles, CA.

Boehm, B., and Papaccio, P. 1988. Understanding and Controlling Software Costs. *IEEE Transactions on Software Engineering* vol. 4(10).

Boehm-Davis, D., and Ross, L. October 1984. *Approaches to Structuring the Software Development Process*. Report GEC/DIS/TR-84-B1V-1. Arlington, VA: General Electric.

Booch, G. 1986. *Software Engineering with Ada*. Menlo Park, CA: Benjamin/Cummings.

Brooks, F. 1975. *The Mythical Man-Month*. Reading, MA: Addison-Wesley.

———. April 1987. No Silver Bullet: Essence and Accidents of Software Engineering. *IEEE Computer* vol. 20(4).

Charette, R. 1989. *Software Engineering Risk Analysis and Management*. New York, NY: McGraw-Hill.

Chidamber, S., and Kemerer, C. 1991. *Towards a Metrics Suite for Object-Oriented Design*. Phoenix, Arizona: OOPSLA'91.

———. 1993. *A Metrics Suite for Object-Oriented Design*. Cambridge, MA: MIT Sloan School of Management.

Chmura, L., Norcio, A., and Wicinski, T. July 1990. Evaluating Software Design Processes by Analyzing Change Date Over Time. *IEEE Transactions on Software Engineering* vol. 16(7).

Corkill, D. D. September 1991. Blackboard Systems. *AI Expert* vol. 6(9), pp. 40–47.

Cox, B. November 1990. Planning the Software Industrial Revolution. *IEEE Software* vol. 7(6).

Curtis, B. May 17, 1989. . . .*But You Have to Understand, This Isn't the Way We Develop Software at Our Company*. MCC Technical Report Number STP-203-89. Austin, TX: Microelectronics and Computer Technology Corporation.

Curtis, B., Kellner, M., and Over, J. September 1992. Process Modeling. *Communications of the ACM* vol. 35(9).

Dahl, O., Dijkstra, E., and Hoare, C. A. R. 1972. *Structured Programming*. London, England: Academic Press.

Davis, A. 1990. *Software Requirements: Analysis and Specification*. Englewood Cliffs, NJ: Prentice Hall.

Davis, A., Bersoff, E., and Comer, E. October 1988. A Strategy for Comparing Alternative Software Development Life Cycle Models. *IEEE Transactions on Software Engineering* vol. 14(10).

Davis, C., Jajodia, S., Ng, P., and Yeh, R., eds. 1983. *Entity-Relationship Approach to Software Engineering*. Amsterdam, The Netherlands: Elsevier Science.

DeMarco, T., and Lister, T. 1987. *Peopleware*. New York, NY: Dorset House.

DeRemer, F., and Kron, H. 1980. Programming-in-the-Large versus Programming-in-the-Small. In *Tutorial on Software Design Techniques*, Third Edition, ed. P. Freeman and A. Wasserman. New York, NY: Computer Society Press of the IEEE.

Dewire, D. 1992. *Client/Server Computing*. New York, NY: McGraw-Hill.

Dijkstra, E. 1979. Programming Considered as a Human Activity. In *Classics in Software Engineering*, ed. E. Yourdon. New York, NY: Yourdon Press.

———. 1982. *Selected Writings on Computing: A Personal Perspective*. New York, NY: Springer-Verlag.

Dowson, M. August 1986. The Structure of the Software Process. *Software Engineering Notes* vol. 11(4).

Dowson, M., Nejmeh, B., and Riddle, W. February 1990. *Software Engineering Practices in Europe, Japan, and the U.S.* Boulder, CO: Software Design and Analysis.

Dreger, B. 1989. *Function Point Analysis*. Englewood Cliffs, NJ: Prentice Hall.

Eastman, N. 1984. Software Engineering and Technology. *Technical Directions* vol. 10(1). Bethesda, MD: IBM Federal Systems Division.

Fagan, M. June 1976. *Design and Code Inspections and Process Control in the Development of Programs*. IBM-TR-00.73.

Foster, C. 1981. *Real-Time Programming*. Reading, MA: Addison-Wesley.

Freedman, D. February 1992. The Devil Is in the Details: Everything Important Must Be Reviewed. *American Programmer* vol. 5(2).

Freeman, P. 1975. *Software Systems Principles*. Chicago, IL: Science Research Associates.

Freeman, P., and Wasserman, A., eds. 1983. *Tutorial on Software Design Techniques*. Fourth

Edition. New York, NY: Computer Society Press of the IEEE.

Gehani, N., and McGettrick, A. 1986. *Software Specification Techniques*. Reading, MA: Addison-Wesley.

Gilb, T. 1988. *Principles of Software Engineering Management*. Reading, MA: Addison-Wesley.

Glass, R. 1982. *Modern Programming Practices: A Report from Industry*. Englewood Cliffs, NJ: Prentice Hall.

———. 1983. *Real-Time Software*. Englewood Cliffs, NJ: Prentice Hall.

———. 1991. *Software Conflict*. Englewood Cliffs, NJ: Yourdon Press.

Goldberg, A., and Rubin, K. 1992. *Tutorial on Object-Oriented Project Management*. Vancouver, Canada: OOPSLA'92.

Guengerich, S. 1992. *Downsizing Information Systems*. Carmel, IN: Sams.

Guindon, R., Krasner, H., and Curtis, B. 1987. *Breakdowns and Processes During the Early Activities of Software Design by Professionals. Empirical Studies of Programmers, Second Workshop*. Norwood, NJ: Ablex Publishing Company.

Guttman, M., and Matthews, J. November/December 1992. Managing a Large Project. *Object Magazine* vol. 2(4).

Hansen, P. 1977. *The Architecture of Concurrent Programs*. Englewood Cliffs, NJ: Prentice Hall.

Hantos, P. 2005. Inherent Risks in Object-Oriented Development. U.S. Air Force Software Technology Support Center (STSC): *CrossTalk, The Journal of Defense Software Engineering*.

Henderson-Sellers, B. 1996. *Object-Oriented Metrics: Measures of Complexity*. Upper Saddle River, NJ: Prentice Hall.

Henderson-Sellers, B., and Edwards, J. September 1990. The Object-Oriented Systems Lifecycle. *Communications of the ACM* vol. 33(9).

Hoare, C. April 1984. Programming: Sorcery or Science? *IEEE Software* vol. 1(2).

Holt, R., Lazowska, E., Graham, G., and Scott, M. 1978. *Structured Concurrent Programming*. Reading, MA: Addison-Wesley.

Humphrey, W. 1988. Characterizing the Software Development Process: A Maturity Framework. *IEEE Software* vol. 5(2).

———. 1989. *Managing the Software Process*. Reading, MA: Addison-Wesley.

Hunt, J. May 2002. Blackboard Architectures. JayDee Technology Ltd 27. www.agent.ai/doc/upload/200402/hunt02_1.pdf.

IEEE-Std-1220-2005. 2005. *IEEE Standard for Application and Management of the Systems Engineering Process*. New York, NY: IEEE.

IEEE-Std-12207.0-1996. 1996. *Standard for Information Technology—Software Life Cycle Processes*. New York, NY: IEEE.

IEEE-Std-12207.1-1997. 1997a. *Standard for Information Technology—Software Life Cycle Processes—*

Life Cycle Data. New York, NY: IEEE.

IEEE-Std-12207.2-1997. 1997b. *Standard for Information Technology—Software Life Cycle Processes—Implementation Considerations*. New York, NY: IEEE.

Jackson, M. 1975. *Principles of Program Design*. Orlando, FL: Academic Press.

———. 1983. *System Development*. Englewood Cliffs, NJ: Prentice Hall.

Jensen, R., and Tonies, C. 1979. *Software Engineering*. Englewood Cliffs, NJ: Prentice Hall.

Jones, C. September 1984. Reusability in Programming: A Survey of the State of the Art. *IEEE Transactions on Software Engineering* vol. SE-10(5).

———. September 1992. Risky Business: The Most Common Software Risks. *American Programmer* vol. 5(7).

Kan, S. H. 2002. *Metrics and Models in Software Quality Engineering*. Second Edition. Boston, MA: Addison-Wesley.

Karam, G., and Casselman, R. February 1993. A Cataloging Framework for Software Development Methods. *IEEE Computer*.

Kemerer, C., and Darcy, D. November/December 2005. OO Metrics in Practice. *IEEE Software* vol. 22(6), p. 17.

Kishida, K., Teramoto, M., Torri, K., and Urano, Y. September 1988. Quality Assurance Technology in Japan. *IEEE Software* vol. 4(5).

Lammers, S. 1986. *Programmers at Work*. Redmond, WA: Microsoft Press.

Laranjeira, L. May 1990. Software Size Estimation of Object-Oriented Systems. *IEEE Transactions on Software Engineering* vol. 16(5).

Ledgard, H. Summer 1985. Programmers: The Amateur vs. the Professional. *Abacus* vol. 2(4).

Lejter, M., Myers, S., and Reiss, S. December 1992. Support for Maintaining Object-Oriented Programs. *IEEE Transactions on Software Engineering* vol. 18(12).

Linger, R., and Mills, H. 1977. On the Development of Large Reliable Programs. In *Current Trends in Programming Methodology: Software Specification and Design* vol. 1, ed. R. Yeh. Englewood Cliffs, NJ: Prentice Hall.

Linger, R., Mills, H., and Witt, B. 1979. *Structured Programming: Theory and Practice*. Reading, MA: Addison-Wesley.

Liskov, B., and Guttag, J. 1986. *Abstraction and Specification in Program Development*. Cambridge, MA: The MIT Press.

Lorenz, M., and Kidd, J. 1994. *Object-Oriented Software Metrics*. Englewood Cliffs, NJ: Prentice Hall.

Lorin, H. 1972. *Parallelism in Hardware and Software*. Englewood Cliffs, NJ: Prentice Hall.

Luqi. August 1990. A Graph Model for Software Evolution. *IEEE Transactions on Software Engineering* vol. 16(8).

———. May 1990. Software Evolution Through Rapid Prototyping. *IEEE Computer* vol. 22(5).

Martin, J., and McClure, C. 1988. *Structured Techniques: The Basis for CASE*. Englewood Cliffs, NJ: Prentice Hall.

Matsubara, T. July/August 1990. Bringing Up Software Designers. *American Programmer* vol. 3(7–8).

McCabe, T., and Butler, C. December 1989. Design Complexity Measurement and Testing. *Communications of the ACM* vol. 32(12).

Mellichamp, D. 1983. *Real-Time Computing*. New York, NY: Van Nostrand Reinhold.

Meyer, B. 1995. *Object Success: A Manager's Guide to Object-Oriented Technology and Its Impact on the Corporation*. Upper Saddle River, NJ: Prentice Hall.

Mills, H. November 1986. Structured Programming: Retrospect and Prospect. *IEEE Software* vol. 3(6).

Mills, J. July 1985. A Pragmatic View of the System Architect. *Communications of the ACM* vol. 28(7).

Mimno, P. April 1993. Cilent-Server Computing. *American Programmer*.

Mullin, M. 1990. *Rapid Prototyping for Object-Oriented Systems*. Reading, MA: Addison-Wesley.

Munck, R. 1985. Toward Large Software Systems That Work. *Proceedings of the AIAA/ACM/NASA/IEEE Computers in Aerospace V Conference*. Menlo Park, CA: AIAA.

Myers, G. 1978. *Composite/Structured Design*. New York, NY: Van Nostrand Reinhold.

Newport, J. April 28, 1986. A Growing Gap in Software. *Fortune*.

Ng, P., and Yeh, R. 1990. *Modern Software Engineering*. New York, NY: Van Nostrand Reinhold.

Office of the Under Secretary of Defense for Acquisition. September 1987. *Report of the Defense Science Board Task Force on Military Software*. Washington, DC.

The Official Handbook of Mascot, Version 3.1. June 1987. London, England: Crown Copyright.

Oman, P., and Lewis, T. 1990. *Milestones in Software Evolution*. Los Alamitos, CA: Computer Society Press of the IEEE.

Orr, K. 1971. *Structured Systems Development*. New York, NY: Yourdon Press.

Page-Jones, M. 1988. *The Practical Guide to Structured Systems Design*. Englewood Cliffs, NJ: Yourdon Press.

Parnas, D. December 1985. Software Aspects of Strategic Defense Systems. *Communications of the ACM* vol. 28(12).

———. July 1985a. Why Conventional Software Development Does Not Produce Reliable Programs. *Software Aspects of Strategic Defense Systems*. Report DCS-47-IR. Victoria, Canada: University of Victoria.

———. July 1985b. Why Software Is Unreliable. *Software Aspects of Strategic Defense Systems*. Report DCS-47-IR. Victoria, Canada: University of Victoria.

Parnas, D., and Clements, P. 1986. A Rational Design Process: How and Why to Fake It. *IEEE Transactions on Software Engineering* vol. SE-12(2).

Peters, L. 1981. *Software Design*. New York, NY: Yourdon Press.

Pressman, R. 1988. *Making Software Happen*. Englewood Cliffs, NJ: Prentice Hall.

———. 1992. *Software Engineering: A Practitioner's Approach*. Third Edition. New York, NY: McGraw-Hill.

Rakos, J. 1990. *Software Project Management for Small to Medium Sized Projects*. Englewood Cliffs, NJ: Prentice Hall.

Ramamoorthy, C., Garg, V., and Prakask, A. July 1986. Programming in the Large. *IEEE Transactions on Software Engineering* vol. SE-12(7).

Rechtin, E. October 1992. The Art of Systems Architecting. *IEEE Spectrum* vol. 29(10).

Rettig, M. October 1990. Software Teams. *Communications of the ACM* vol. 33(10).

Ross, D., Goodenough, J., and Irvine, C. 1980. Software Engineering: Process, Principles, and Goals. *Tutorial on Software Design Techniques*, Third Edition, ed. P. Freeman and A. Wasserman. New York, NY: Computer Society Press of the IEEE.

Rubinstein, R., and Hersh, H. 1984. *The Human Factor*. Burlington, MA: Digital Press.

Schulmeyer, G., and McManus, J. 1992. *Handbook of Software Quality Assurance*. Second Edition. New York, NY: Van Nostrand Reinhold.

Shaw, M. November 1990. Prospects for an Engineering Discipline of Software. *IEEE Software* vol. 7(6).

Smith, M., and Robson, D. June 1992. A Framework for Testing Object-Oriented Programs. *Journal of Object-Oriented Programming* vol. 5(3).

Software Process Workshop. May 1988. *SIGSOFT Software Engineering Notes* vol. 14(4).

Sommerville, I. 1989. *Software Engineering*. Third Edition. Wokingham, England: Addison- Wesley.

Song, X., and Osterweil, L. 1993. *Executing an Iterative Design Process*. Irvine, CA: University of California.

Spector, A., and Gifford, D. April 1986. Computer Science Perspective of Bridge Design. *Communications of the ACM* vol. 29(4).

Stevens, W., Myers, G., and Constantine, L. 1979. Structured Design. In *Classics in Software Engineering*, ed. E. Yourdon. New York, NY: Yourdon Press.

Symons, C. 1988. Function Point Analysis: Difficulties and Improvements. *IEEE Transactions on Software Engineering* vol. 14(1).

Taylor, D. 1990. *Object-Oriented Technology: A Manager's Guide*. Alameda, CA: Servio Corporation.

The Software Trap: Automate—Or Else. May 9, 1988. *Business Week*.

Thomsett, R. July/August 1990. Effective Project Teams. *American Programmer* vol. 3(7–8).

———. June 1991. Managing Superlarge Projects: A Contingency Approach. *American Programmer* vol. 4(6).

U.S. Department of Defense. July 30, 1982. *Report of the DoD Joint Service Task Force on Software Problems*. Washington, DC.

van Genuchten, M. June 1991. Why Is Software Late? An Empirical Study of Reasons for Delay in Software Development. *IEEE Transactions on Software Engineering* vol. 17(6).

Vick, C., and Ramamoorthy, C. 1984. *Software Engineering*. New York, NY: Van Nostrand Reinhold.

Vonk, R. 1990. *Prototyping*. Englewood Cliffs, NJ: Prentice Hall.

Walsh, J. 1992. *Preliminary Defect Data from the Iterative Development of a Large C++ Program*. Vancouver, Canada: OOPSLA'92.

———. January 1993. *Software Quality in an Iterative Object-Oriented Development Paradigm*. Santa Clara, CA: Rational.

Ward, M. 1990. *Software That Works*. San Diego, CA: Academic Press.

Ward, P., and Mellor, S. 1985. *Structured Development for Real-Time Systems: Introduction and Tools*. Englewood Cliffs, NJ: Yourdon Press.

Wegner, P. 1980. *Research Directions in Software Technology*. Cambridge, MA: The MIT Press.

———. July 1984. Capital-Intensive Software Technology. *IEEE Software* vol. 1(3).

Weinberg, G. 1988. *Understanding the Professional Programmer*. New York, NY: Dorset House Publishing.

Weinberg, G., and Freedman, D. 1990. *Handbook of Walkthroughs, Inspections, and Technical Reviews*. New York, NY: Dorset House.

Whitten, N. 1990. *Managing Software Development Projects*. New York, NY: John Wiley and Sons.

Wilde, N., and Huitt, R. December 1992. Maintenance Support for Object-Oriented Programs. *IEEE Transactions on Software Engineering* vol. 18(12).

Wilde, N., Matthews, P., and Huitt, R. January 1993. Maintaining Object-Oriented Software. *IEEE Software* vol. 10(1).

Wirth, N. 1986. *Algorithms and Data Structures*. Englewood Cliffs, NJ: Prentice Hall.

Workshop on Software Configuration Management. November 1989. *SIGSOFT Software Engineering Notes* vol. 17(7).

Yamaura, T. January 1992. Standing Naked in the Snow. *American Programmer* vol. 5(1).

Yeh, R., ed. 1977. *Current Trends in Programming Methodology: Software Specification and Design*. Englewood Cliffs, NJ: Prentice Hall.

Yourdon, E. 1975. *Techniques of Program Structure and Design*. Englewood Cliffs, NJ: Prentice Hall.

———, ed. 1979. *Classics in Software Engineering*. New York, NY: Yourdon Press.

———. 1989a. *Modern Structured Analysis*. Englewood Cliffs, NJ: Prentice Hall.

———. 1989b. *Structured Walkthroughs*. Englewood Cliffs, NJ: Prentice Hall.

———. August 1989c. The Year of the Object. *Computer Language* vol. 6(8).

———. Summer 1989d. Object-Oriented Observations. *American Programmer* vol. 2(7–8).

Yourdon, E., and Constantine, L. 1979. *Structured Design*. Englewood Cliffs, NJ: Prentice Hall.

Zahniseer, R. July/August 1990. Building Software in Groups. *American Programmer* vol. 3(7–8).

Zave, P. February 1984. The Operational versus the Conventional Approach to Software Development. *Communications of the ACM* vol. 27(2).

Zelkowitz, M. June 1978. Perspectives on Software Engineering. *ACM Computing Surveys* vol. 10(2).

I. 特别参考文献

Alexander, C. 1979. *The Timeless Way of Building*. New York, NY: Oxford University Press.

DeGrace, P., and Stahl, L. 1990. *Wicked Problems, Righteous Solutions*. Englewood Cliffs, NJ: Yourdon Press.

Fukuyama, F. 1992. *The End of History and the Last Man*. New York, NY: The Free Press.

Gall, J. 1986. *Systemantics: How Systems Really Work and How They Fail*. Second Edition. Ann Arbor, MI: The General Systemantics Press.

Gleick, J. 1987. *Chaos*. New York, NY: Penguin Books.

Heckbert, P. 1988. Ray Tracing Jell-O Brand Gelatin. *Communications of the ACM* vol. 31(2).

Heinlein, R. 1966. *The Moon Is a Harsh Mistress*. New York, NY: The Berkeley Publishing Group.

Hofstadter, D. 1979. *Gödel, Escher, Bach: An Eternal Golden Braid*. New York, NY: Vintage Books.

International Council on Systems Engineering (INCOSE). 2006. *INCOSE Systems Engineering Handbook*. Version 3. INCOSE-TP-2003-002-03.

Inside Macintosh vol. 1-5. 1988. Reading, MA: Addison-Wesley.

Kawasaki, G. 1990. *The Macintosh Way*. Glenview, IL: Scott, Foresman and Company.

Lakoff, G., and Johnson, M. 1980. *Metaphors We Live By*. Chicago, IL: The University of Chicago Press.

Lammers, S. 1986. *Programmers at Work*. Bellevue, WA: Microsoft Press.

Meyer, C., and Matyas, S. M. 1982. *Cryptography*. New York, NY: John Wiley and Sons.

Parker, T. 1983. *Rules of Thumb*. Boston, MA: Houghton Mifflin Company.

Peter, L. 1986. *The Peter Pyramid*. New York, NY: William Morrow.

Petroski, H. 1985. *To Engineer Is Human*. New York, NY: St. Martin's Press.

Rand, Ayn. 1979. *Introduction to Objectivist Epistemology*. New York, NY: New American Library.

Reti, L. 1988. *The Unknown Leonard*. New York, NY: Abradale Press.

Sears, F., Zemansky, M., and Young., H. 1987. *University Physics*. Seventh Edition. Reading, MA: Addison-Wesley.

U.S. Department of Commerce/National Institute of Standards and Technology. October 25, 1999. *Data Encryption Standard (DES), Federal Information Processing Standards Publication 46-3*.

vonOech, R. 1990. *A Whack on the Side of the Head*. New York, NY: Warner Books.

Wagner, J. 1986. *The Search for Signs of Intelligent Life in the Universe*. New York, NY: Harper and Row.

Whitehead, A. 1958. *An Introduction to Mathematics*. New York, NY: Oxford University Press.

J. 理论

Aho, A., Hopcroft, J., and Ullman, J. 1974. *The Design and Analysis of Computer Programs*. Reading, MA: Addison-Wesley.

Almarode, J. October 1989. Rule-Based Delegation for Prototypes. *SIGPLAN Notices* vol. 24(10).

Appelbe, W., and Ravn, A. April 1984. Encapsulation Constructs in Systems Programming Languages. *ACM Transactions on Programming Languages and Systems* vol. 6(2).

Averill, E. April 1982. Theory of Design and Its Relationship to Capacity Measurement. *Proceedings of the Fourth Annual International Conference on Computer Capacity Management*. San Francisco, CA: Association of Computing Machinery.

Barr, A., and Feigenbaum, E. 1981. *The Handbook of Artificial Intelligence*. Los Altos, CA: William Kaufmann.

Bastani, F., and Iyengar, S. March 1987. The Effect of Data Structures on the Logical Complexity of Programs. *Communications of the ACM* vol. 30(3).

Bastani, F., Hilal, W., and Sitharama, S. October 1987. Efficient Abstract Data Type Components for Distributed and Parallel Systems. *IEEE Computer* vol. 20(10).

Belkhouche, B., and Urban, J. May 1986. Direct Implementation of Abstract Data Types from Abstract Specifications. *IEEE Transactions on Software Engineering* vol. SE-12(5).

Bensley, E., Brando, T., and Prelle, M. September 1988. An Execution Model for Distributed Object-Oriented Computation. *SIGPLAN Notices* vol. 23(11).

Berztiss, A. 1980. Data Abstraction, Controlled Iteration, and Communicating Processes. *Communications of the ACM*.

Bishop, J. 1986. *Data Abstraction in Programming Languages*. Wokingham, England: Addison-Wesley.

Boehm, H., Demers, A., and Donahue, J. October 1980. *An Informal Description of Russell*. Technical Report TR 80-430. Ithaca, NY: Cornell University.

Borning, A., Duisberg, R., Freeman-Benson, B., Kramer, A., and Woolf, M. October 1987. Constraint Hierarchies. *SIGPLAN Notices* vol. 22(12).

Boute, R. January 1988. Systems Semantics: Principles, Applications, and Implementation. *ACM Transactions on Programming Languages and Systems* vol. 10(1).

Brachman, R. October 1983. What Is-a Is and Isn't: An Analysis of Taxonomic Links in Semantic Networks. *IEEE Computer* vol. 16(10).

Brachman, R., and Levesque, H., eds. 1985. *Readings in Knowledge Representation*. Los Altos, CA: Morgan Kaufmann.

Brooks, R. April 1987. *Intelligence without Representation*. Cambridge, MA: MIT Artificial Intelligence Laboratory.

Bruce, K., and Wegner, P. October 1986. An Algebraic Model of Subtypes in Object-Oriented Languages. *SIGPLAN Notices* vol. 21(10).

Card, S., Moran, T., and Newell, A. 1983. *The Psychology of Human-Computer Interaction*. Hillsdale, NJ: Lawrence Erlbaum Associates.

Cardelli, L., and Wegner, P. December 1985. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys* vol. 17(4).

Claybrook, B., and Wyckof, M. 1980. Module: An Encapsulation Mechanism for Specifying and Implementing Abstract Data Types. *Communications of the ACM*.

Cline, A., and Rich, E. December 1983. *Building and Evaluating Abstract Data Types*. Report TR-83-26. Austin, TX: University of Texas, Department of Computer Sciences.

Cohen, A. January 1984. Data Abstraction, Data Encapsulation, and Object-Oriented Programming. *SIGPLAN Notices* vol. 19(1).

Cohen, N. November/December 1985. Tasks as Abstraction Mechanisms. *Ada Letters* vol. 5(3–6).

Cohen, P., and Loiselle, C. August 1988. Beyond ISA: Structures for Plausible Inference in Semantic Nets. *Proceedings of the Seventh National Conference on Artificial Intelligence*. Saint Paul, MN: American Association for Artificial Intelligence.

Collins, W. 1992. *Data Structures: An Object-Oriented Approach*. Reading, MA: Addison-Wesley.

Cook, W., and Palsberg, J. October 1989. A Denotational Semantics of Inheritance and Its Correctness. *SIGPLAN Notices* vol. 24(10).

Courtois, P., Heymans, F., and Parnas, D. October 1971. Concurrent Control with "Readers" and "Writers." *Communications of the ACM* vol. 14(10).

Danforth, S., and Tomlinson, C. March 1988. Type Theories and Object-Oriented Programming. *ACM Computing Surveys* vol. 20(1).

Demers, A., Donahue, J., and Skinner, G. Data Types as Values: Polymorphism, Type-Checking, Encapsulation. *Proceedings of the Fifth Annual ACM Symposium on Principles of Programming Languages*. New York, NY: Association of Computing Machinery.

Dennis, J., and Van Horn, E. March 1966. Programming Semantics for Multiprogrammed Computations. *Communications of the ACM* vol. 9(3).

Donahue, J., and Demers, A. July 1985. Data Types Are Values. *ACM Transactions on Programming Languages and Systems* vol. 7(3).

Eckart, J. April 1987. Iteration and Abstract Data Types. *SIGPLAN Notices* vol. 22(4).

Embley, D., and Woodfield, S. 1988. Assessing the Quality of Abstract Data Types Written in Ada. *Proceedings of the 10th International Conference on Software Engineering*. New York, NY: Computer Society Press of the IEEE.

Ferber, J. October 1989. Computational Reflection in Class-Based Object-Oriented Languages. *SIGPLAN Notices* vol. 24(10).

Fisher, J., and Gipson, D. November 1992. In Search of Elegance. *Computer Language* vol. 9(11).

Gannon, J., Hamlet, R., and Mills, H. July 1987. Theory of Modules. *IEEE Transactions on Software Engineering* vol. SE-13(7).

Gannon, J., McMullin, P., and Hamlet, R. July 1981. Data Abstraction Implementation, Specification, and Testing. *ACM Transactions on Programming Languages and Systems* vol. 3(3).

Gardner, M. May/June 1984. When to Use Private Types. *Ada Letters* vol. 3(6).

Goguen, J., Thatcher, J., and Wagner, E. 1977. An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types. In *Current Trends in Programming Methodology: Data Structuring* vol. 4, ed. R. Yeh. Englewood Cliffs, NJ: Prentice Hall.

Goldberg, D. 1989. *Genetic Algorithms*. Reading, MA: Addison-Wesley.

Graube, N. October 1989. Metaclass Compatibility. *SIGPLAN Notices* vol. 24(10).

Gries, D., and Prins, J. July 1985. A New Notion of Encapsulation. *SIGPLAN Notices* vol. 20(7).

Grogono, P., and Bennett, A. November 1989. Polymorphism and Type Checking in Object-Oriented Languages. *SIGPLAN Notices* vol. 24(11).

Guttag, J. 1980. Abstract Data Types and the Development of Data Structures. In *Programming Language Design*, ed. A. Wasserman. New York, NY: Computer Society Press of the IEEE.

Hammons, C., and Dobbs, P. May/June 1985. Coupling, Cohesion, and Package Unity in Ada. *Ada Letters* vol. 4(6).

Harel, D., and Kahana, C. October 1992. On Statecharts with Overlapping. *ACM Transactions on Software Engineering and Methodology* vol. 1(4).

Harel, D., Lachover, H., Naamad, A., Pnueli, A., Politi, M., Sherman, R., Shtull-Trauring, S., and Trakhtenbrot, M. April 1990. STATEMATE: A Working Environment for the Development of Complex Reactive Systems. *IEEE Transactions on Software Engineering* vol. 16(4).

Harrison, G., and Liu, D. July/August 1986. Generic Implementations Via Analogies in the Ada Programming Language. *Ada Letters* vol. 6(4).

Hayes, P. 1981. The Logic of Frames. In *Readings in Artificial Intelligence*, ed. B. Webber and N. Nilsson. Palo Alto, CA: Tioga.

Hayes-Roth, F. July 1985. A Blackboard Architecture for Control. *Artificial Intelligence* vol. 26(3).

Hayes-Roth, F., Waterman, D., and Lenat, D. 1983. *Building Expert Systems*. Reading, MA: Addison-Wesley.

Haynes, C., and Friedman, D. October 1987. Embedding Continuations in Procedural Objects. *ACM Transactions on Programming Languages and Systems* vol. 9(4).

Henderson, P. February 1986. Functional Programming, Formal Specification, and Rapid Prototyping. *IEEE Transactions on Software Engineering* vol. SE-12(2).

Herlihy, M., and Liskov, B. October 1982. A Value Transmission Method for Abstract Data Types. *ACM Transactions on Programming Languages and Systems* vol. 4(4).

Hesselink, W. January 1988. A Mathematical Approach to Nondeterminism in Data Types. *ACM Transactions on Programming Languages and Systems* vol. 10(1).

Hibbard, P., Hisgen, A., Rosenbers, J., Shaw, M., and Sherman, M. 1981. *Studies in Ada Style*. New York, NY: Springer-Verlag.

Hilfinger, P. 1982. *Abstraction Mechanisms and Language Design*. Cambridge, MA: The MIT Press.

Hoare, C. October 1974. Monitors: An Operating System Structuring Concept. *Communications of the ACM* vol. 17(10).

Hoare, C. 1985. *Communicating Sequential Processes*. Englewood Cliffs, NJ: Prentice Hall.

Hogg, J., and Weiser, S. October 1987. OTM: Applying Objects to Tasks. *SIGPLAN Notices* vol. 22(12).

Jajodia, S., and Ng. P. 1983. On Representation of Relational Structures by Entity-Relationship Diagrams. In *Entity-Relationship Approach to Software Engineering*, ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.

Johnson, C. 1986. Some Design Constraints Required for the Assembly of Software Components: The Incorporation of Atomic Abstract Types into Generically Structured Abstract Types. *Proceedings of the First International Conference on Ada Programming Language Applications for the NASA Space Station*. Houston, TX: NASA Lyndon B. Johnson Space Center.

Kernighan, B., and Plauger, P. 1981. *Software Tools in Pascal*. Reading, MA: Addison-Wesley.

Knight, B. 1983. A Mathematical Basis for Entity Analysis. In *Entity-Relationship Approach to Software Engineering*, ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.

Knuth, D. 1973. *The Art of Computer Programming* vol. 1-3. Reading, MA: Addison-Wesley.

Kosko, B. 1992. *Neural Networks and Fuzzy Systems*. Englewood Cliffs, NJ: Prentice Hall.

LaLonde, W., and Pugh, J. August 1985. Specialization, Generalization, and Inheritance: Teaching Objectives Beyond Data Structures and Data Types. *SIGPLAN Notices* vol. 20(8).

Leeson, J., and Spear, M. March 1987. Type-Independent Modules: The Preferred Approach to Generic ADTs in Modula-2. *SIGPLAN Notices* vol. 22(3).

Lenzerini, M., and Santucci, G. 1983. Cardinality Constraints in the Entity-Relationship Model. In

Entity-Relationship Approach to Software Engineering, ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.

Levesque, H. July 1984. Foundations of a Functional Approach to Knowledge Representation. *Artificial Intelligence* vol. 23(2).

Lindgreen, P. 1983. Entity Sets and Their Description. In *Entity-Relationship Approach to Software Engineering*, ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.

Lins, C. 1989. A First Look at Literate Programming. *Structured Programming.*

Liskov, B. 1980. Programming with Abstract Data Types. In *Programming Language Design*, ed. A. Wasserman. New York, NY: Computer Society Press of the IEEE.

———. May 1988. Data Abstraction and Hierarchy. *SIGPLAN Notices* vol. 23(5).

Liskov, B., and Scheifler, R. July 1983. Guardians and Actions: Linguistic Support for Robust, Distributed Programs. *ACM Transactions on Programming Languages and Systems* vol. 5(3).

Liskov, B., and Zilles, S. 1977. An Introduction to Formal Specifications of Data Abstractions. In *Current Trends in Programming Methodology: Software Specification and Design* vol. 1, ed. R. Yeh. Englewood Cliffs, NJ: Prentice Hall.

Lowry, M., and McCartney, R.. 1991. *Automating Software Design.* Cambridge, MA: The MIT Press.

Lucco, S. October 1987. Parallel Programming in a Virtual Object Space. *SIGPLAN Notices* vol. 22(12).

Maes, P. October 1987. Concepts and Experiments in Computational Reflection. *SIGPLAN Notices* vol. 22(12).

Mark, L. 1983. What Is the Binary Relationship Approach? In *Entity-Relationship Approach to Software Engineering*, ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.

Markowitz, V., and Raz, Y. 1983. A Modified Relational Algebra and Its Use in an Entity-Relationship Environment. In *Entity-Relationship Approach to Software Engineering*, ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.

Matsuoka, S., and Kawai, S. September 1988. Using Tuple Space Communication in Distributed Object-Oriented Languages. *SIGPLAN Notices* vol. 23(11).

McAllester, D., and Zabih, F. November 1986. Boolean Classes. *SIGPLAN Notices* vol. 21(11).

McCullough, P. October 1987. Transparent Forwarding: First Steps. *SIGPLAN Notices* vol. 22(12).

Merlin, P., and Bochmann, G. January 1983. On the Construction of Submodule Specifications and Communication Protocols. *ACM Transactions on Programming Languages and Systems* vol. 5(1)

Meyer, B. 1987. *Programming as Contracting.* Report TR-EI-12/CO. Goleta, CA: Interactive Software Engineering.

———. October 1992. Applying "Design by Contract." *IEEE Computer* vol. 25(10).

Minoura, T., and Iyengar, S. January 1989. Data and Time Abstraction Techniques for Multilevel Concurrent Systems. *IEEE Transactions on Software Engineering* vol. 15(1).

Murata, T. 1984. Modeling and Analysis of Concurrent Systems. In *Software Engineering*, ed. C. Vick and C. Ramamoorthy. New York, NY: Van Nostrand Reinhold.

Mylopoulos, J., and Levesque, H. 1984. An Overview of Knowledge Representation. In *On Conceptual Modeling: Perspectives from Artificial Intelligence, Databases, and Programming Languages*, ed. M. Brodie, J. Mylopoulos, and J. Schmidt. New York, NY: Springer-Verlag.

Nakano, R. 1983. Integrity Checking in a Logic-Oriented ER Model. In *Entity-Relationship Approach to Software Engineering*, ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.

Newton, M., and Watkins, J. November/December 1988. The Combination of Logic and Objects for Knowledge Representation. *Journal of Object-Oriented Programming* vol. 1(4).

Nii, P. Summer 1986. Blackboard Systems: The Blackboard Model of Problem Solving and the Evolution of Blackboard Architectures. *AI Magazine* vol. 7(2).

Ohori, A., and Buneman, P. October 1989. Static Type Inference for Parametric Classes. *SIGPLAN Notices* vol. 24(10).

Pagan, F. 1981. *Formal Specification of Programming Languages*. Englewood Cliffs, NJ: Prentice Hall.

Parent, C., and Spaccapieta, S. July 1985. An Algebra for a General Entity-Relationship Model. *IEEE Transactions on Software Engineering* vol. SE-11(7).

Parnas, D. 1977. The Influence of Software Structure on Reliability. In *Current Trends in Programming Methodology: Software Specification and Design* vol. 1, ed. R. Yeh. Englewood Cliffs, NJ: Prentice Hall.

———. 1980. Designing Software for Ease of Extension and Contraction. In *Tutorial on Software Design Techniques*, Third Edition, ed. P. Freeman and A. Wasserman. New York, NY: Computer Society Press of the IEEE.

Parnas, D., Clements, P., and Weiss, D. 1983. Enhancing Reusability with Information Hiding. *Proceedings of the Workshop on Reusability in Programming*, Stratford, CT: ITT Programming.

Pattee, H. 1973. *Hierarchy Theory*. New York, NY: George Braziller.

Peckham, J., and Maryanski, F. September 1988. Semantic Data Models. *ACM Computing Surveys* vol. 20(3).

Pedersen, C. October 1989. Extending Ordinary Inheritance Schemes to Include Generalization. *SIGPLAN Notices* vol. 24(10).

Peterson, J. September 1977. Petri Nets. *Computing Surveys* vol. 9(3).

Reed, D. September 1978. Naming and Synchronization in a Decentralized Computer System. Cambridge, MA: The MIT Press.

Rich, C., and Wills, L. January 1990. Recognizing a Program's Design: A Graph-Parsing Approach. *IEEE Software* vol. 7(1).

Robinson, L., and Levitt, K. 1977. Proof Techniques for Hierarchically Structured Programs. In *Current Trends in Programming Methodology: Program Validation* vol. 2, ed. R. Yeh. Englewood Cliffs, NJ: Prentice Hall.

Ross, D. July/August 1986. Classifying Ada Packages. *Ada Letters* vol. 6(4).

Ruane, L. January 1984. Abstract Data Types in Assembly Language Programming. *SIGPLAN Notices* vol. 19(1).

Rumbaugh, J. September 1988. Controlling Propagation of Operations Using Attributes on Relations. *SIGPLAN Notices* vol. 23(11).

Sedgewick, R. 1983. *Algorithms*. Reading, MA: Addison-Wesley.

Shankar, K. 1984. Data Design: Types, Structures, and Abstractions. In *Software Engineering*, ed. C. Vick and C. Ramamoorthy. New York, NY: Van Nostrand Reinhold.

Shaw, M. 1984. The Impact of Modeling and Abstraction Concerns on Modern Programming Languages. *On Conceptual Modeling: Perspectives from Artificial Intelligence, Databases, and Programming Languages*, ed. M. Brodie, J. Mylopoulos, and J. Schmidt. New York, NY: Springer-Verlag.

———. October 1984. Abstraction Techniques in Modern Programming Languages. *IEEE Software* vol. 1(4).

———. May 1989. Larger Scale Systems Require Higher-Level Abstractions. *SIGSOFT Engineering Notes* vol. 14(3).

Shaw, M., Feldman, G., Fitzgerald, R., Hilfinger, P., Kimura, I., London, R., Rosenberg, J., and Wulf, W. 1981. Validating the Utility of Abstraction Techniques. In *ALPHARD: Form and Content*, ed. M. Shaw. New York, NY: Springer-Verlag.

Shaw, M., Wulf, W., and London, R. 1981. Abstraction and Verification in ALPHARD: Iteration and Generators. In *ALPHARD: Form and Content*, ed. M. Shaw. New York, NY: Springer-Verlag.

Sherman, M., Hisgen, A., and Rosenberg, J. 1982. A Methodology for Programming Abstract Data Types in Ada. *Proceedings of the AdaTEC Conference on Ada*. New York, NY: Association of Computing Machinery.

Siegel, J. April 1988. Twisty Little Passages. *HOOPLA: Hooray for Object-Oriented Programming Languages* vol. 1(3). Everette, WA: Object Oriented Programming for Smalltalk Application Developers Association.

Stefik, M., Bobrow, D., and Kahn, K. January 1986. Integrating Access-Oriented Programming into a Multiparadigm Environment. *IEEE Software* vol. 3(1).

Strom, R., and Yemini, S. January 1986. Typestate: A Programming Language Concept for

Enhancing Software Reliability. *IEEE Transactions on Software Engineering* vol. SE-12(1).

Stubbs, D., and Webre, N. 1985. *Data Structures with Abstract Data Types and Pascal*. Monterey, CA: Brooks/Cole.

Swaine, M. June 1988. Programming Paradigms. *Dr. Dobb's Journal of Software Tools* no. 140.

Tabourier, Y. 1983. Further Development of the Occurrences Structure Concept: The EROS Approach. In *Entity-Relationship Approach to Software Engineering*, ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.

Tanenbaum, A. 1981. *Computer Networks*. Englewood Cliffs, NJ: Prentice Hall.

Tenenbaum, A., and Augenstein, M. 1981. *Data Structures Using Pascal*. Englewood Cliffs, NJ: Prentice Hall.

Throelli, L. October 1987. Modules and Type Checking in PL/LL. *SIGPLAN Notices* vol. 22(12).

Tomlinson, C., and Singh, V. October 1989. Inheritance and Synchronization with Enabled-sets. *SIGPLAN Notices* vol. 24(10).

Toy, W. 1984. Hardware/Software Tradeoffs. In *Software Engineering*, ed. C. Vick and C. Ramamoorthy. New York, NY: Van Nostrand Reinhold.

Vegdahl, S. November 1986. Moving Structures between Smalltalk Images. *SIGPLAN Notices* vol. 21(11).

Walters, N. October 1992. Using Harel Statecharts to Model Object-Oriented Behavior. *SIGSOFT Notices* vol. 17(4).

Wasserman, A. 1980. Introduction to Data Types. In *Programming Language Design*, ed. A. Wasserman. New York, NY: Computer Society Press of the IEEE.

Weber, H., and Ehrig, H. July 1986. Specification of Modular Systems. *IEEE Transactions on Software Engineering* vol. SE-12(7).

Wegner, P. June 6, 1981. *The Ada Programming Language and Environment*. Unpublished draft.

———. 1987a. On the Unification of Data and Program Abstraction in Ada. In *Object-Oriented Computing: Concepts* vol. 1, ed. G. Peterson. New York, NY: Computer Society Press of the IEEE.

———. 1987b. The Object-Oriented Classification Paradigm. In *Research Directions in Object-Oriented Programming*, ed. B. Schriver and P. Wegner. Cambridge, MA: The MIT Press.

Wegner, P., and Zdonik, S. August 1988. Inheritance as an Incremental Modification Mechanism or What Like Is and Isn't Like. *Proceedings of ECOOP'88: European Conference on Object-Oriented Programming*. New York, NY: Springer-Verlag.

Weihl, W., and Liskov, B. April 1985. Implementation of Resilient, Atomic Data Types. *ACM Transactions on Programming Languages and Systems* vol. 7(2)

Weinberg, G. 1971. *The Psychology of Computer Programming*. New York, NY: Van Nostrand Reinhold.

Weller, D., and York, B. May 1984. A Relational Representation of an Abstract Type System. *IEEE Transactions on Software Engineering* vol. SE-10(3).

White, J. July 1983. On the Multiple Implementation of Abstract Data Types within a Computation. *IEEE Transactions on Software Engineering* vol. SE-9(4).

Wirth, N. December 1974. On the Composition of Well-Structured Programs. *Computing Surveys* vol. 6(4).

———. January 1983. Program Development by Stepwise Refinement. *Communications of the ACM* vol. 26(1).

———. 1986. *Algorithms and Data Structures*. Second Edition. Englewood Cliffs, NJ: Prentice Hall.

———. April 1988. Type Extensions. *ACM Transactions on Programming Languages and Systems* vol. 10(2).

Wolf, A., Clarke, L., and Wileden, J. April 1988. A Model of Visibility Control. *IEEE Transactions on Software Engineering* vol. 14(4).

Woods, W. October 1983. What's Important about Knowledge Representation? *IEEE Computer* vol. 16(10).

Zilles, S. 1984. Types, Algebras, and Modeling. In *On Conceptual Modeling: Perspectives from Artificial Intelligence, Databases, and Programming Languages*, ed. M. Brodie, J. Mylopoulos, and J. Schmidt. New York, NY: Springer-Verlag.

Zippel, R. June 1983. Capsules. *SIGPLAN Notices* vol. 18(6).

K. 工具和环境

Andrews, T., and Harris, C. 1987. *Combining Language and Database Advances in an Object-Oriented Development Environment*. Billerica, MA: Ontologic.

Corradi, A., and Leonardi, L. 1986. *An Environment Based on Parallel Objects*. Bologna, Italy: Universita di Bologna.

Deutsch, P., and Taft, E. June 1980. *Requirements for an Experimental Programming Environment*. Report CSL-80-10. Palo Alto, CA: Xerox Palo Alto Research Center.

Diederich, J., and Milton, J. October 1987. An Object-Oriented Design System Shell. *SIGPLAN Notices* vol. 22(12).

Durant, D., Carlson, G., and Yao, P. 1987. *Programmer's Guide to Windows*. Berkeley, CA: Sybex.

Erman, L., Lark, J., and Hayes-Roth, F. December 1988. ABE: An Environment for Engineering Intelligent Systems. *IEEE Transactions on Software Engineering* vol. 14(12).

Ferrel, P., and Meyer, R. October 1989. Vamp: The Aldus Application Framework. *SIGPLAN Notices* vol. 24(10).

Fischer, H., and Martin, D. 1987. *Integrating Ada Design Graphics into the Ada Software*

Development Process. Encino, CA: Mark V Business Systems.

Goldberg, A. 1984a. *Smalltalk-80: The Interactive Programming Environment*. Reading, MA: Addison-Wesley.

———. 1984b. The Influence of an Object-Oriented Language on the Programming Environment. In *Interactive Programming Environments*, ed. B. Barstow. New York, NY: McGraw-Hill.

Goldstein, I., and Bobrow, D. March 1981. *An Experimental Description-Based Programming Environment*. Report CSL-81-3. Palo Alto, CA: Xerox Palo Alto Research Center.

Gorlen, K. May 1986. *Object-Oriented Program Support*. Bethesda, MD: National Institute of Health.

Hecht, A., and Simmons, A. 1986. Integrating Automated Structured Analysis and Design with Ada Programming Support Environments. *Proceedings of the First International Conference on Ada Programming Language Applications for the NASA Space Station*. Houston, TX: NASA Lyndon B. Johnson Space Center.

Hedin, G., and Magnusson B. August 1988. The Mjolner Environment: Direct Interaction with Abstractions. *Proceedings of ECOOP'88: European Conference on Object-Oriented Programming*. New York, NY: Springer-Verlag.

Hudson, S., and King, R. June 1988. The Cactic Project: Database Support for Software Environments. *IEEE Transactions on Software Engineering* vol. 14(6).

International Business Machines. April 1988. *Operating System/2 Seminar Proceedings, IBM OS/2 Standard Edition Version 1.1, IBM Operating System/2 Update, Presentation Manager*. Boca Raton, FL.

Kant, E. 1987. *Interactive Problem Solving with a Task Configuration and Control System*. Ridgefield, CT: Schlumberger-Doll Research.

Kleyn, M., and Gingrich, P. September 1988. GraphTrace—Understanding Object-Oriented Systems Using Concurrently Animated Views. *SIGPLAN Notices* vol. 23(11).

Laff, M., and Hailpern, B. July 1985. SW-2—An Object-Based Programming Environment. *SIGPLAN Notices* vol. 20(7).

MacLenna, B. July 1985. A Simple Software Environment Based on Objects and Relations. *SIGPLAN Notices* vol. 20(7).

Marques, J., and Guedes, P. October 1989. Extending the Operating System to Support an Object-Oriented Environment. *SIGPLAN Notices* vol. 24(10).

Minsky, N., and Rozenshtein, D. February 1988. A Software Development Environment for Law-Governed Systems. *SIGPLAN Notices* vol. 24(2).

Moreau, D., and Dominick, W. 1987. *Object-Oriented Graphical Information Systems: Research Plan and Evaluation Metrics*. Lafayette, LA: University of Southwestern Louisiana, Center for Advanced Computer Studies.

Nakata, S., and Yamazak, G. 1983. ISMOS: A System Based on the E-R Model and Its Application to Database-Oriented Tool Generation. In *Entity-Relationship Approach to Software Engineering*, ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.

Nye, A. 1989. *Xlib Programming Manual for Version 11*. Newton, MA: O'Reilly and Associates.

O'Brien, P., Halbert, D., and Kilian, M. October 1987. The Trellis Programming Environment. *SIGPLAN Notices* vol. 22(12).

Open Look Graphical User Interface Functional Specification. 1990. Reading, MA: Addison-Wesley.

OSF/Motif Style Guide, Version 1.0. 1989. Cambridge, MA: Open Software Foundation.

Penedo, M., Ploedereder, E., and Thomas, I. February 1988. Object Management Issues for Software Engineering Environments. *SIGPLAN Notices* vol. 24(2).

Reenskaug, T., and Skaar, A. October 1989. An Environment for Literate Smalltalk Programming. *SIGPLAN Notices* vol. 24(10).

Rosenplatt, W., Wileden, J., and Wolf, A. October 1989. OROS: Toward a Type Model for Software Development Environments. *SIGPLAN Notices* vol. 24(10).

Russo, V., and Campbell, R. October 1989. Virtual Memory and Backing Storage Management in Multiprocessor Operating Systems Using Object-Oriented Design Techniques. *SIGPLAN Notices* vol. 24(10).

Scheifler, R., and Gettys, J. 1986. The X Window System. *ACM Transactions on Graphics* vol. 63.

Schwan, K., and Matthews, J. July 1986. Graphical Views of Parallel Programs. *Software Engineering Notes* vol. 11(3).

Shear, D. December 8, 1988. CASE Shows Promise but Confusion Still Exists. *EDN* vol. 33(25).

Sun Microsystems. March 29, 1987. *NeWS Technical Overview*. Mountain View, CA.

Tarumi, H., Agusa, K., and Ohno, Y. 1988. A Programming Environment Supporting Reuse of Object-Oriented Software. In *Proceedings of the 10th International Conference on Software Engineering*. New York, NY: Computer Society Press of the IEEE.

Taylor, R., Belz, F., Clarke, L., Osterweil, L., Selby, R., Wileden, J., Wolf, A., and Young, M. February 1988. Foundations for the Arcadia Environment. *SIGPLAN Notices* vol. 24(2).

Tesler, L. August 1981. The Smalltalk Environment. *Byte* vol. 6(8).

Vines, D., and King, T. 1988. *Gaia: An Object-Oriented Framework for an Ada Environment*. Minneapolis, MN: Honeywell.

Weinand, A., Gamma, E., and Marty, R. 1989. Design and Implementation of ET++, a Seamless Object-Oriented Application Framework. *Structured Programming* vol. 10(2).

Wiorkowski, G., and Kull, D. 1988. *DB2 Design and Development Guide*. Reading, MA: Addison-Wesley.

L. 统一建模语言

Bennett, S., Skelton, J., and Lunn, K. 2005. *Schaum's Outline of UML*. Second Edition. United Kingdom: McGraw-Hill.

Blaha, M., and Rumbaugh, J. 2005. *Object-Oriented Modeling and Design with UML*. Second Edition. Upper Saddle River, NJ: Prentice Hall.

Bock, C. 2003. UML 2 Activity and Action Models. *Journal of Object Technology* vol. 2(4), pp. 43–53. www.jot.fm/issues/issue_2003_07/column3.

———. 2003. UML 2 Activity and Action Models, Part 2: Actions. *Journal of Object Technology* vol. 2(5), pp. 41–56. www.jot.fm/issues/issue_2003_09/column4.

———. 2003. UML 2 Activity and Action Models, Part 3: Control Nodes. *Journal of Object Technology* vol. 2(6), pp. 7–23. www.jot.fm/issues/issue_2003_11/column1.

———. 2004. UML 2 Activity and Action Models, Part 4: Object Nodes. *Journal of Object Technology* vol. 3(1), pp. 27–41. www.jot.fm/issues/issue_2004_01/column3.

———. 2004. UML 2 Activity and Action Models, Part 5: Partitions. *Journal of Object Technology* vol. 3(7), pp. 37–56. www.jot.fm/issues/issue_2004_07/column4.

Booch, G., Rumbaugh, J., and Jacobson, I. 2005. *The Unified Modeling Language User Guide*. Second Edition. Boston, MA: Addison-Wesley.

Fowler, M. 2003. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Third Edition. Boston, MA: Addison-Wesley.

Maksimchuk, R., and Naiburg, E. 2005. *UML for Mere Mortals*. Boston, MA: Addison-Wesley.

Naiburg, E., and Maksimchuk, R. 2001. *UML for Database Design*. Boston, MA: Addison-Wesley.

Rumbaugh, J., Jacobson, I., and Booch, G. 2005. *The Unified Modeling Language Reference Manual*. Second Edition. Boston, MA: Addison-Wesley.

Object Management Group. 2004. *UML Superstructure Specification, v2.0*. www.uml.org/.

M. 基于 Web 建模

Cavaness, C. 2004. *Programming Jakarta Struts*. Second Edition. Indianapolis, IN: O'Reilly.

Constantine, L. L., and Lockwood, L. A. D. 1999. *Software for Use: A Practical Guide to the Models and Methods of Usage-Centered Design*. Reading, MA: Addison-Wesley.

Garrett, J. J. 2002. *The Elements of User Experience*. Boston, MA: New Riders Press.

Geary, D., and Horstmann, C. 2004. *Core JavaServer Faces*. Upper Saddle River, NJ: Prentice Hall.

Monson-Haefel, R. 2001. *Enterprise JavaBeans*. Third Edition. Indianapolis, IN: O'Reilly.