Broadview
www.broadview.com.cn

Pearson

Jolt大奖精选丛书

面向对象分析与设计

第3版
修订版

Object-Oriented Analysis and Design with Applications (Third Edition)

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

王海鹏 潘加宇 译

中国工信出版集团

電子工業出版社

Broadview®
www.broadview.com.cn

Pearson

Jolt大奖精选丛书

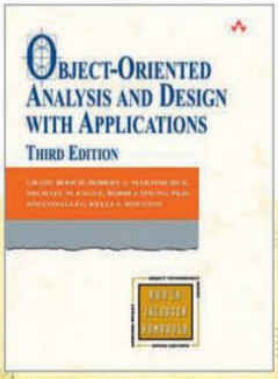
面向对象分析与设计

第3版
修订版

Object-Oriented Analysis and Design with Applications (Third Edition)

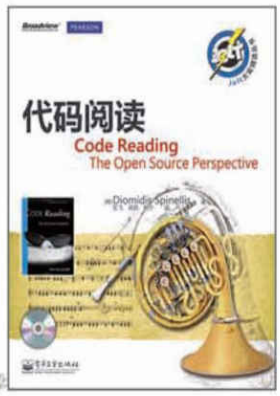
[美] Grady Booch
Robert A. Maksimchuk
Michael W. Engle
Bobbi J. Young,PH.D.
Jim Conallen
Kelli A. Houston
著

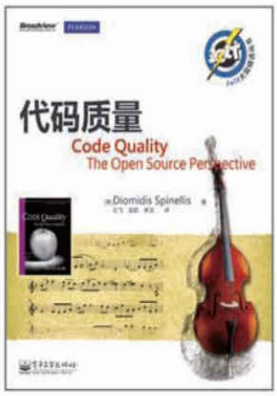
王海鹏 潘加宇 译

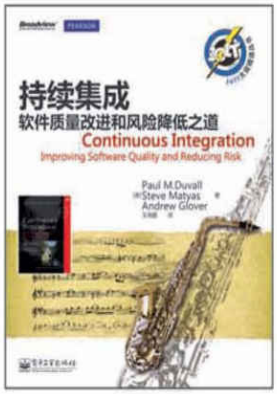


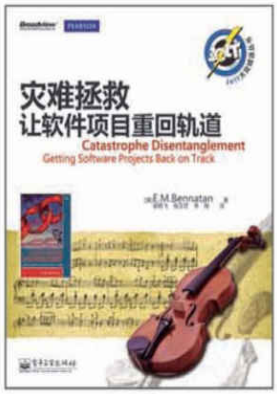
中国工信出版集团

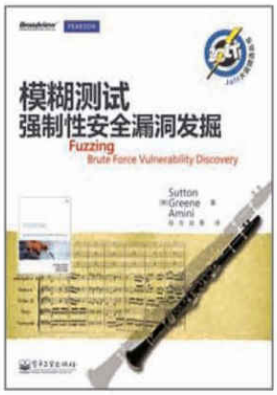
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

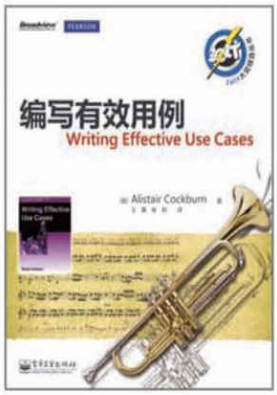

代码阅读
Code Reading
The Open Source Perspective
Diomidis Spinellis


代码质量
Code Quality
The Open Source Perspective
Diomidis Spinellis


持续集成
软件质量改进和风险降低之道
Continuous Integration
Improving Software Quality and Reducing Risk
Paul M.Duvall
Steve Matyas
Andrew Glover


灾难拯救
让软件项目重回轨道
Catastrophe Disentanglement
Getting Software Projects Back on Track
E.M.Bennatan


模糊测试
强制性安全漏洞发掘
Fuzzing
Brute Force Vulnerability Discovery
Sutton
Greene
Amini


编写有效用例
Writing Effective Use Cases
Alistair Cockburn

版权信息

COPYRIGHT

书名：面向对象分析与设计：第3版：修订版

作者：布奇（Booch,G.）等

译者：王海鹏，潘加宇

出版社：电子工业出版社

出版时间：2016年5月

ISBN：9787121286667

字数：850千字

内容简介

Jolt大奖素有“软件业之奥斯卡”的美称，本丛书精选自Jolt历届获奖图书，以植根于开发实践中的独到工程思想与杰出方法论为主要甄选方向。本书第1版和第2版分别于1991年和1993年荣获Jolt震撼奖。

本书是UML创始人Grady Booch的代表作之一。

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交付阶段，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。书中介绍的概念都基于牢固的理论基础。同时，作者十分注重实效，基于其丰富的经验，面向软件工程实践者的实际需要，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；运用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。

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。本书非常适合实际系统和软件的开发者、系统分析师或架构师、项目经理阅读，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。

出版说明

经久不息的回荡

今时的读书人，不复有无书可读之苦，却时有品种繁多而无从择优之惑，甚而专业度颇高的技术书领域，亦日趋遭逢乱花迷眼的境地。此时，若得觅权威书评，抑或有公信力的排行榜，可按图索骥，大大增加选中好书的命中率。然而，如此良助，不可多得，纵观中外也唯见一枝独秀——素有“软件业奥斯卡”之美誉的Jolt奖！

震撼世界者为谁

在计算设备已经成为企业生产和日常生活之必备工具的今天，专业和大众用户对于软件的功能、性能和用户体验的要求都在不断提高。在这样的背景下，如何能够发挥出软件开发的最高效率和最大效能，已经是摆在每一个从业者面前的重大课题，而这也正是Jolt大奖横空出世的初衷及坚持数年的宗旨。

Jolt大奖历时20余年，在图书及软件业知名度极高，广受推崇。奖如其名，为引领计算机科学与工程发展主流，Jolt坚持将每年的奖项只颁给那些给整个IT业界带来震撼结果的图书、工具、产品及理念等，因一流的眼光及超高的专业度而得以闻名遐迩，声名远播。

除图书外，Jolt针对软件产品设有诸多奖项分类，如配置管理、协作工具、数据库引擎/数据库工具、设计工具/建模、开发环境、企业工具、库/框架、移动开发工具等。但图书历来是Jolt大奖中最受瞩目且传播最广的一个奖项分支。Jolt曾设有通用类图书、技术类图书等分类，每个分类又设有“卓越奖”（Jolt Award，一般为一个）和“生产力奖”（Productivity Award，一般为2或3个）。获奖技术图书一经公布，即打上经典烙印，可谓一举“震撼全世界”（赞助商Jolt可乐的广告词）。

作为计算机技术图书的后进，我们总在追问——是谁在震撼世界，是谁在照亮明天？Jolt大奖恰似摆在眼前的橱窗，让我们可以近距离观看潮流在舞蹈，倾听震撼在轰鸣！

朝花夕拾为哪般

Jolt像是一年一度的承诺，在茫茫书海中为我们淘砺出一批批经得起岁月冲刷的杰作，头顶桂冠的佳作也因而得以一批批引进中国，为国人开阔了眼界，滋补了技术养分。然而，或因技术差距造就的生不逢时、水土不服，或因翻译、制作的不如人意，抑或是疏于宣传等诸多原因，这些经典著作在国内出版后，尽管不乏如获至宝的拥趸，却仍不为诸多人所知，从而与大量本应从中获益的读者擦肩而过。既然这生生错失的遗憾本不该发生，则更不应延续。为此，我们邀国外出版同行、国内技术专家一道，踏上朝花夕拾之路，竭力为广大读者筛选出历久弥新、震撼依旧的Jolt图书精品。

Jolt获奖图书皆由业界专家一致评出，并得到软件从业人员的高度认可，虽然这些书今天读来，不再能看到20世纪史诗时代那般日新月异的理论突破，以及依赖于高深繁复的科学研究所取得的系统化成果，更多是在日复一日的开发实践中总结和提炼出来的工程思想和方法论。重新选材之所以有所弃取，从Jolt多年来的评奖规律中可窥端倪——

一万小时真理见

凡是在工程思想领域取得革命性、颠覆性突破的图书，就被归于“震撼”获奖分类。比如，从基于过程的程序设计模型过渡到面向对象的全新模型，就是软件开发思想上的一次带来巨大震撼的革命；再比如，打破传统的瀑布模型而转向持续集成的软件交付模型，这也是一场业界的重大思想转变。像这样的重大思想突破，可以说是数年甚至数十年一遇的，而荣获Jolt大奖的图书中更为常见的，则是基于最佳实践的“生产效率”获奖者。获得此类殊荣的图书，都是作者们从平凡的、重复的，甚至用一般人的眼光看来不怎么起眼的日常开发实践中，以独具的慧眼、过人的耐心和大胆的创新，闯开一条不平常道路的心血与经验总结。

这些图书所涉及的主题，都是普通的软件开发人员每天要面对的工作——代码阅读、撰写测试用例、修复软件问题……但就是这样貌似平淡无奇的工作，是否能每一天、每一个项目都做好，着实拉开了软件开发人员素质的差距，也决定了软件企业开发出来的产品和服务的质量。我们中国有一句古话，叫作熟能生巧；某位著名企业家也说过一句家喻户晓的名言：“把简单的事千百万次地做好，就是不简单的。”这些朴素而实际的真理，同样也是本套丛书最能彰显的所谓程序员精神。它建立在脚踏实地的实践基础之上，也充满了对于自由和创新的向往。

名作可堪比名曲

就不因岁月流逝而褪色来说，与这些Jolt名作相媲美者，只有那些百年响彻、震撼古今的经典名曲。希望本丛书带给大家的每部著作，也如百听不厌的乐曲，掩卷良久方余音绕梁，真知存心。仔细想来，软件开发与古典音乐岂非有异曲同工之妙？既是人类心智索问精确科学的探究，亦是寻觅美学享受的追求。工程是艺术的根基，而艺术是工程的极致。衷心地希望各位读者能够认真阅读本丛书的本本珍品，并切实地用于自己的日常工作中，在充分享受大师魅力的同时，为中国的软件事业谱写更多、更震撼的乐章。

谨以本书献给**Jan**：

我的朋友，我的爱人，我的妻子。

——Grady

重看面向对象

软件的本质是什么？从不同的角度来看，会有不同的答案。有人认为是程序加文档，有人认为是人机交互，有人认为是增删改查，有人认为是抽象模型，而我认为是算法。

计算机科学的基石是图灵机抽象：一个输入集合，一个输出集合，一个内部状态集合，一个计算规则集合。这个抽象十分强大，我们甚至可以认为一头奶牛也是一个图灵机：吃的是草，挤的是奶。

一个表达式也是一个图灵机，其中的操作数是输入，求值的结果是输出。一个函数也是图灵机，参数是输出，返回值是输出。编程或设计软件就是在通用图灵机的基础上，设计一个具体的图灵机。我们设计软件接受怎样的输入，设计软件内部的状态，设计表示计算规则的代码，设计软件的输出。

近年来逐渐流行的函数式编程，就是建立在这个抽象的基础上。而且函数式编程的思想由来已久，可以追溯到最古老的高级语言之一：Lisp。这种思想非常简单：给定一个输入集合，经过函数的处理，给出一个输出集合。由此也导出了Map-Reduce等流行的架构设计模式：一个计算集群仍然是一个图灵机。

纯粹的函数有一点不足，它没有内部状态。可以说，它是简化了的图灵机。但在有些时候，我们确实需要内部状态。根据内部状态的不同，对于同样的输入，可能给出不一样的输出。于是便有了闭包的概念，它是一个函数加上相关的上下文环境状态。这样，我们可以毫无困难地构建任何具体的图灵机（好吧，正确实现一个算法还是比较困难的）。

闭包可以看成是拥有内部状态的函数，这就相当于一个简单的对象，它只有一个方法。反过来，对象可以看成是几个闭包，它们共享了内部状态。所以有人说：闭包是懒人的对象，对象是懒人的闭包。因此，函数式编程和面向对象思想，在底层基础上是一致的。

面向对象思想的历史和函数式一样久远。实际上，它们都是我们在设计算法时的一种抽象。只有利用抽象概念，才能实现人与人

之间的沟通。“你想吃苹果吗？”这里的“苹果”就是一个抽象概念，它隐藏了苹果实现的许多细节。人的大脑喜欢工作在一组抽象概念上。名词是结构或存在的抽象，动词是行为或过程的抽象。

我们在设计算法时，既需要函数抽象，也需要对象抽象。今天，面向对象和函数式编程的思想在各种编程语言中融合，可以说是殊途同归。

抽象是强大的工具，但用得不好，也会产生不良的后果。最重要的问题，就是创建太多不必要的抽象。毕竟，抽象只是我们脑中的概念，我们可以创造出任何概念。比如上帝和各种鬼神，直到科学家说，在科学的系统里不需要假设存在一个上帝。面向对象在这方面遇到的问题比较多。举例来说，一个Java的Hello world程序，就要涉及好几个概念，直接导致程序的代码比较长。而在函数式编程中，这通常只是一次函数调用。又比如，在一个使用Struts、Spring、Hibernate构建的Java Web应用程序中，处理一个Get请求的调用栈，可能是长长的一串。数据在不同的概念抽象之间反复倒手，白白浪费了计算资源。

任何两种观点都是互补的。面向对象思想在过去的软件开发中取得了辉煌的成绩。函数式编程让我们能从另一个角度审视面向对象，更进一步体会面向对象抽象的强大，也发现面向对象中一些误用的地方。如无必要，勿增实体。也许我们不需要假设以太的存在，就能解释光在真空中的传播。

多年后重读这本书，促使我重新思考，需要利用哪些抽象来设计我的算法。

这些年来，这本书让我受益良多，再次向大家郑重推荐。

王海鹏

2016年1月5日

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上的化身叫作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上进行的。

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交付阶段，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

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。这是可以看到的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，“为什么事情总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相比有四项主要区别，如下所示。

1．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。

2．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。

3．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。

4．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。它的具体目标如下。

- 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篇：概念、方法和应用，其中穿插了大量的补充材料。

概念

第1篇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篇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篇提供了一组（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材料。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

对于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，则可以在第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协作，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——Camy、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，他们的爱和支持给我了真正的灵感。

——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接受咨询。

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博士是一名咨询师，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的志愿者和大学讲师。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。

第1篇　概念

Isaac Newton（艾萨克·牛顿）爵士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

医生、土木工程师和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，并导致陈述的需求中存在缺陷。我们常常把这种情况称为软件危机，但老实说，问题持续了这么长时间，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，丧失了许多机会。没有足够好的开发者来创建用户需要的所有新软件，而且在任何组织机构中，相当一部分开发者必须经常维护或照看老的软件。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地增强个人的能力，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。这些部分又可以进一步分解，ALU可以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组成，如NAND门、反相器等。

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一组设备，相互协作，为更高的抽象层提供服务。我们选择某个抽象层来满足特定的需求。例如，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。植物是复杂的多细胞生物，通过各种植物器官系统之间的协作，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），每种结构都有不同的、特有的结构。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。

给定抽象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，但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，存在着清晰的关注点分离。

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，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，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。天文学家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。

我们再次发现，一种非凡的共性以普适机制的方式，统一了这种宏大的层次结构。具体来说，宇宙中似乎只存在四种相互作用力：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。这些大多数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、修复或扩展它们的功能。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“工业级”的开发挑战，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应用程序的创建，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四个原因：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。

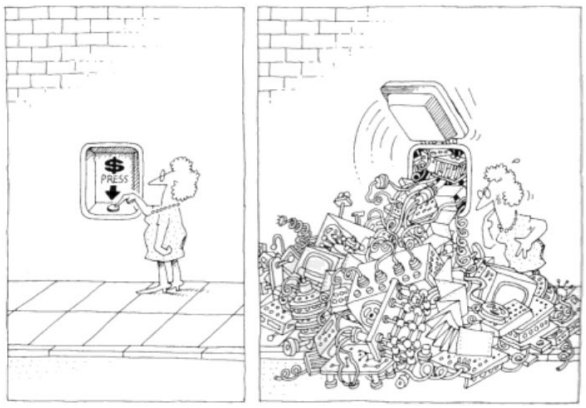
1．问题域的复杂性

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所说的任意复杂性的原因之一。

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，即让用户与大量的、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。我们追求通过发明一些聪明、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，即便是一个系统需求的数量，有时候也无法逃避，这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，因此更难协调，特别是

当开发团队的地理位置分散的时候，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

一家建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。我们试图以关注点分离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，穷尽所有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，对于系统的正确性，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，我们认为所有复杂系统都存在5个共同的属性。

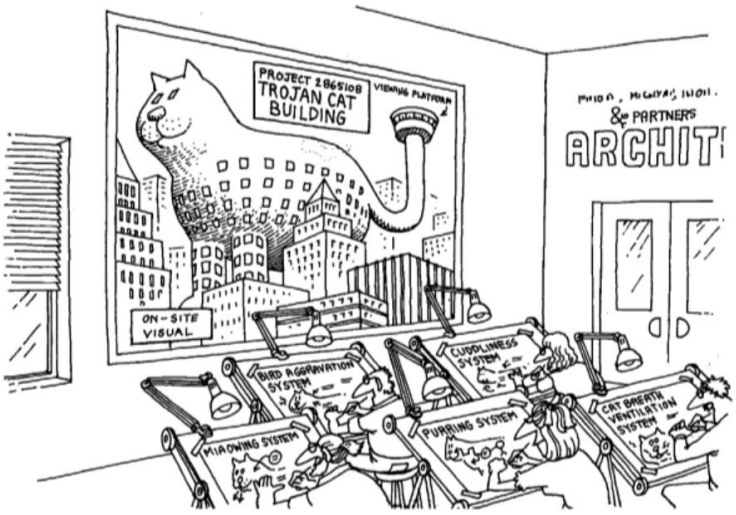
1.3.1 层次结构

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子系统，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及其层次结构的函数，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]。



复杂系统的架构是它所有的组件及其层次结构的函数

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。

1.3.3 关注点分离

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，他又称它们是“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“关注点分离”，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。”[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驾驶一架飞机，那么学会驾驶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，应该既研究它的“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”，原因将在第2章中解释[2]。

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。

如图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。

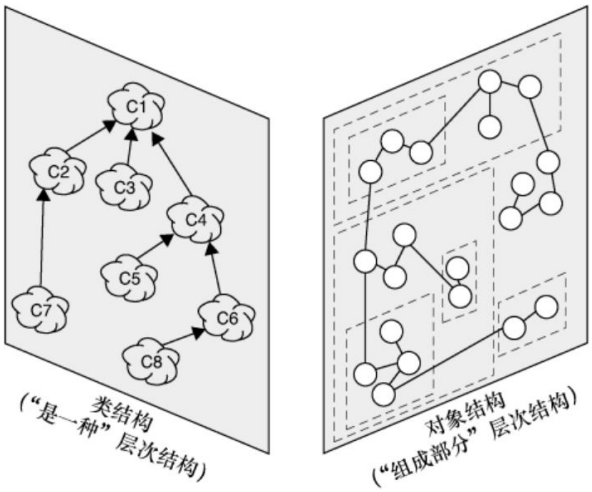


图1-1　复杂系统的关键层次结构

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）、关注点分离、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”。

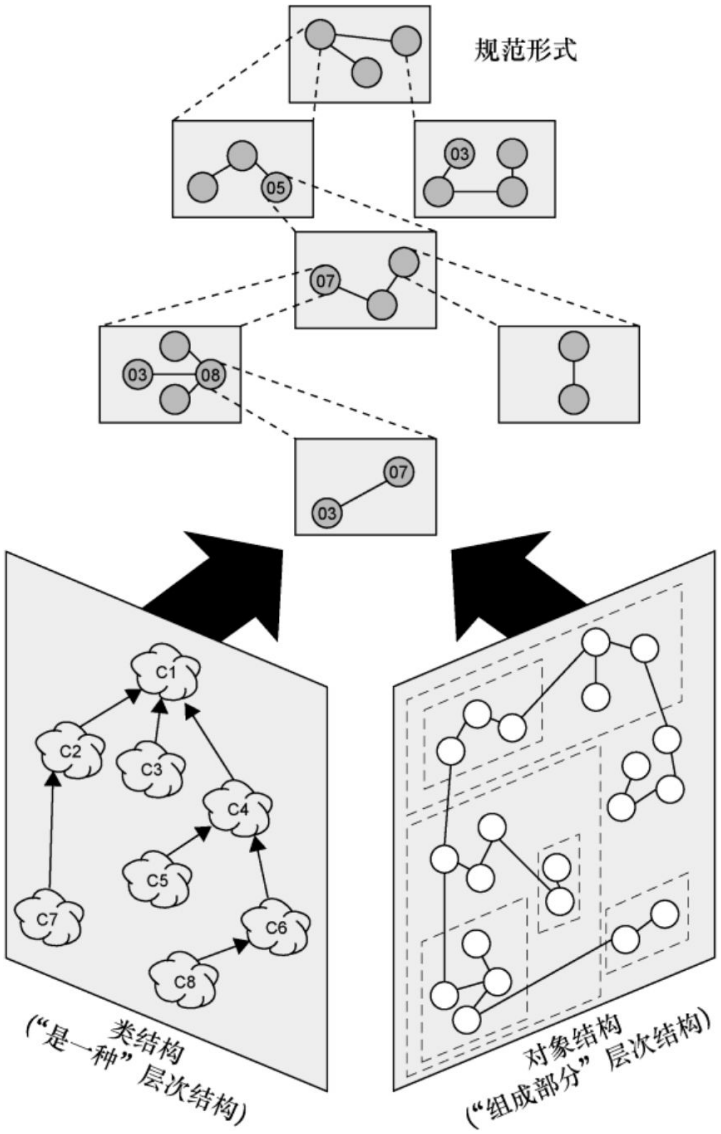


图1-2　复杂系统的规范形式

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”。这也是系统架构具有挑战性的原因——在可能的多种系统组件结构、复杂系统的5种属性及系统用户的需求之中寻找平衡。

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 人在处理复杂性时的能力局限

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，即人在处理复杂性时的能力局限。

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莱特（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，聪明的分解直接解决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]。

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。

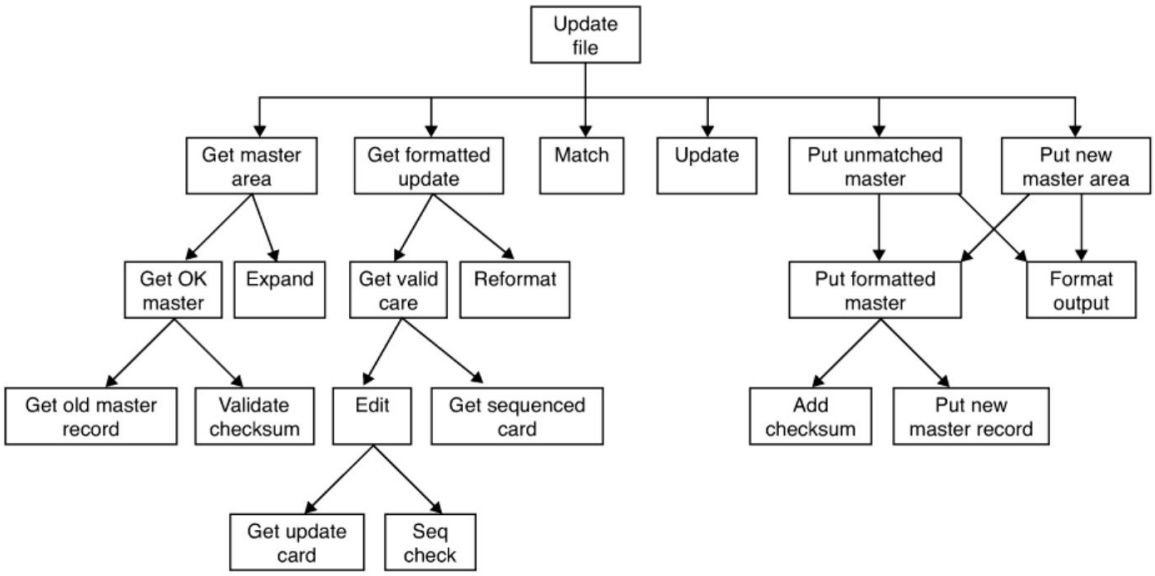


图1-3　算法分解

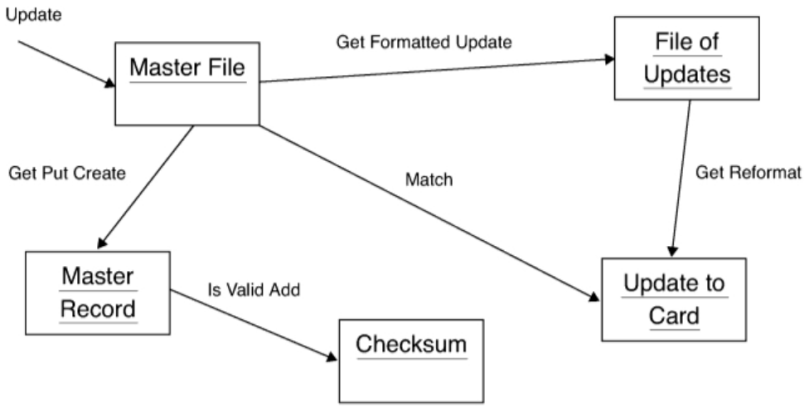


图1-4　面向对象的分解

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 Updates（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。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，方法也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60年代和70年代提出来的，它们要解决的问题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的影响，如FORTRAN和COBOL。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, 32]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。正如结构化设计一样，数据驱动的设计已经成功地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++、Common Lisp 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。[3]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关注点分离，面向对象的分解直接关注了软件的内在复杂性。

本书第三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。本节补充材料“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，并形成一些片段序列，我们设法打破……这种信息瓶颈”[35]。用现在的术语来说，我们把这个过程叫作“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。如果不能够全面掌握一个复杂的对象，我们就选择忽略非本质的细节，转而处理这个对象的一般化的、理想化的模型”[36]。例如，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，我们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，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曾雄辩地指出：“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，因此既需要正规数学家的能力，也需要称职工程师的态度”[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。特别是在设计响应式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，如创建框架、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，从而提供实现需求的途径。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；
- 满足隐含和明确的性能及资源使用需求；
- 满足隐含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。这个组件级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需的实践方法。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”）。除去了它们的不同之处，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：分解一个复杂系统的最佳方法是什么？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。通过明智地分离软件的状态空间，我们对软件正确性的信心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。

---

[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存在更深层次的有序，以一种名为“吸引子”的模式存在。

[2]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。

[3]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作为关注的中心。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]

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。

毫无疑问，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

面向对象开发不是从无数使用早期技术和失败的软件项目的灰烬中自发产生的。它不是与早期方法断然决裂的。实际上，它建立在以前技术的最佳思想之上。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，开发出了像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。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？），但是它们引入的许多概念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，展示了组件复用这个容易被忘记的承诺。

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依赖关系。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，并且常常会引入混乱：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。

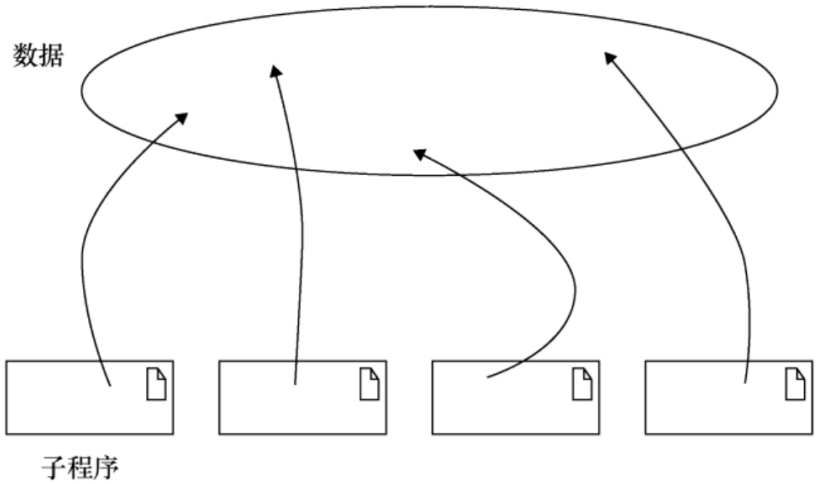


图2-1　第一代和第二代早期程序设计语言的结构

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年之前就被发明了，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。第三，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。

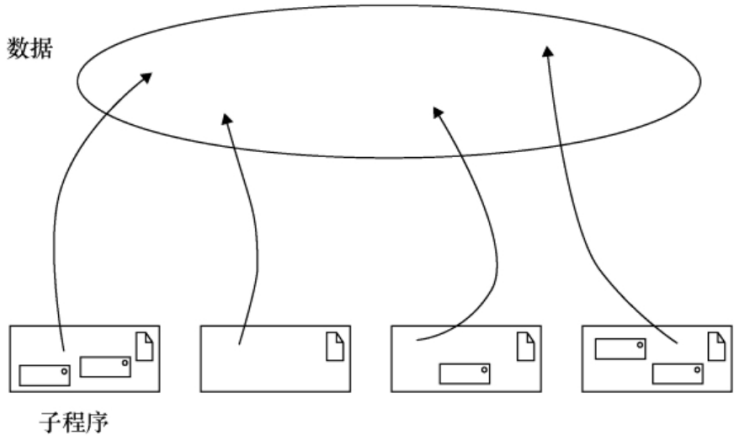


图2-2　第二代后期和第三代早期程序设计语言的结构

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。

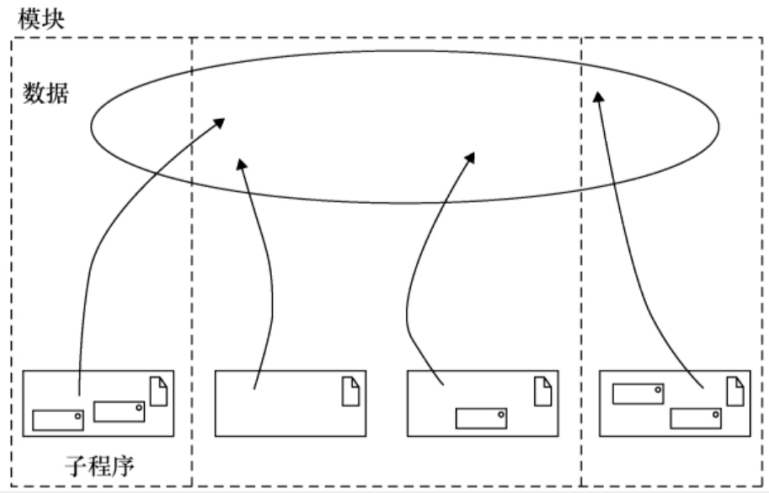


图2-3　第三代后期程序设计语言的结构

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，一个模块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的支持都不太好，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

数据抽象对于把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。

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。

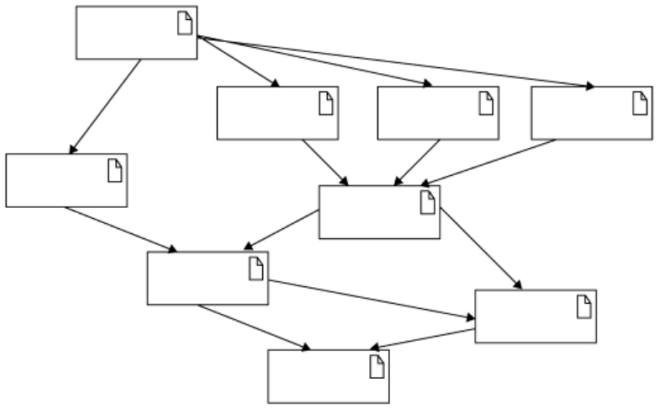


图2-4　使用基于对象或面向对象编程语言的小型或中型应用的结构

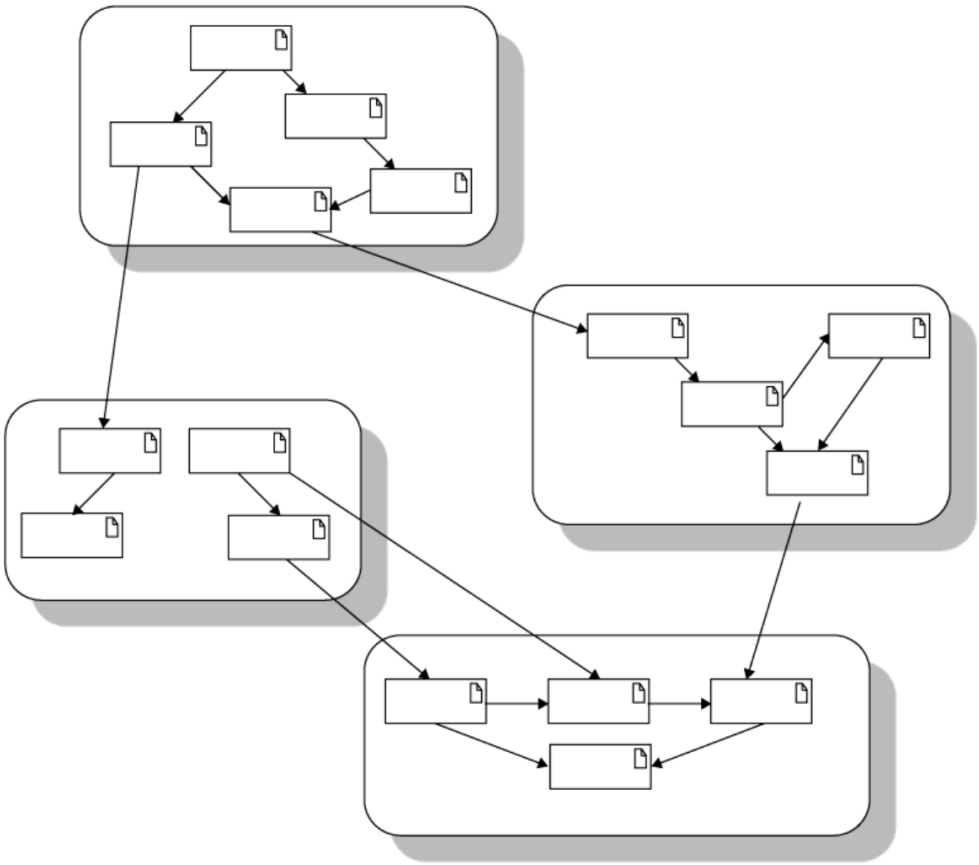


图2-5　使用基于对象或面向对象编程语言的大型应用的结构

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，而是基于已经证明的好方法。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术语。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。在第1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征，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]。

我们可以加上如下三点，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台处理器设计）、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、Smalltalk-74、Smalltalk-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。Parnas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成为了架构的基础。

最后，哲学和认知科学对面向对象的发展也做出了贡献。世界可以被看作是对象或过程，这两种观点是希腊人的发明，在17世纪，笛卡儿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，这种语言只是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）？我们建议采用下面的定义：

> “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。

支持面向对象分解，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会产生影响。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，它基于某种编程概念模型和一种适合的语言，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。这个模型有四个主要要素：

（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。

对象模型有三个次要要素：

（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。

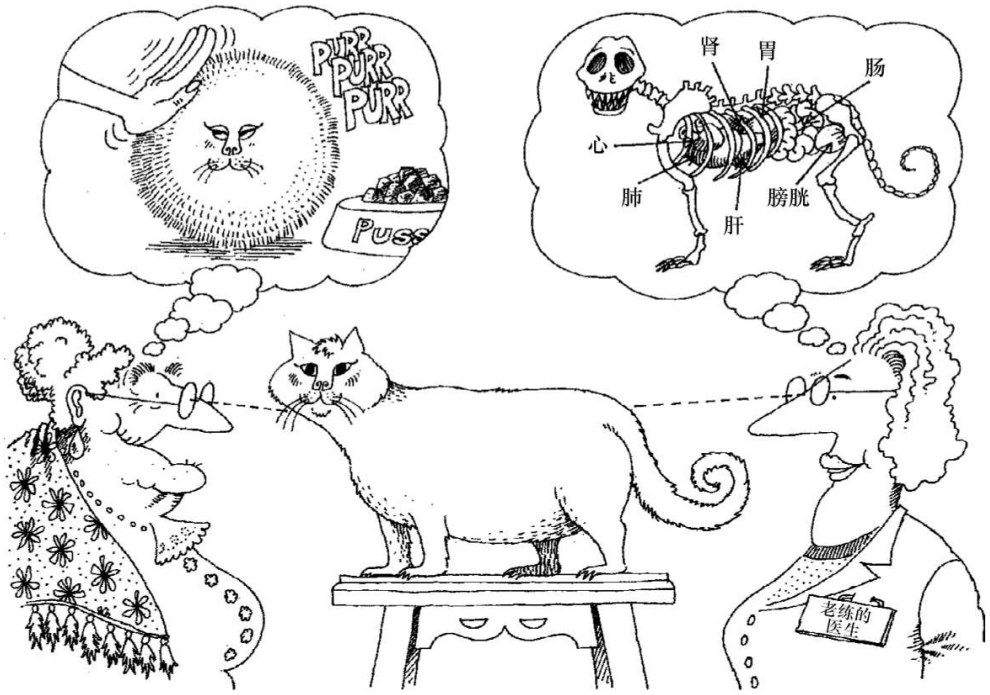
如果没有这个概念框架，你可能在使用Smalltalk、Object Pascal、C++、Eiffel或Ada这样的语言编程，但是你的设计可能看起来像是FORTRAN、Pascal或C应用。更重要的是，你不太可能把握手上问题的复杂性。

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。



抽象关注某个对象的基本特征，它与观察者的视角有关

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，所有这些操作都执行同类的功能；
- 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职责，即它的可靠的行为。

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，就意味着客户没有完成它那部分的职责，因此服务器不能可靠地执行。类似地，如果违反了后置条件，就意味着服务器没有完成它那部分的职责，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：温度传感器的职责是什么？我们的设计决策是，传感器负责知道某个位置的温度，并在被询问时报告温度。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UML 2.0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，只是它的职责有了一点小小

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。

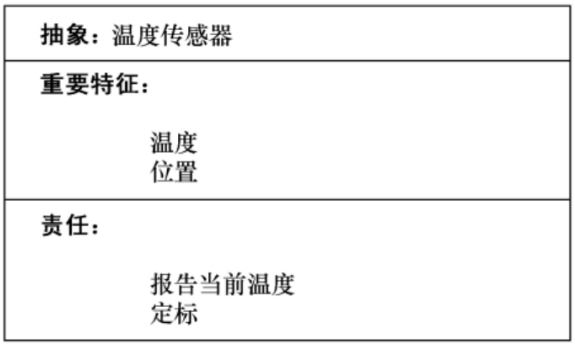


图2-6　温度传感器的抽象

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职责，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”这样的职责可能变成多项职责，如“设定温度”、“设定pH值”等。当收集到更多的客户需求知识，设计变得成熟，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职责（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“关注点分离”，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。

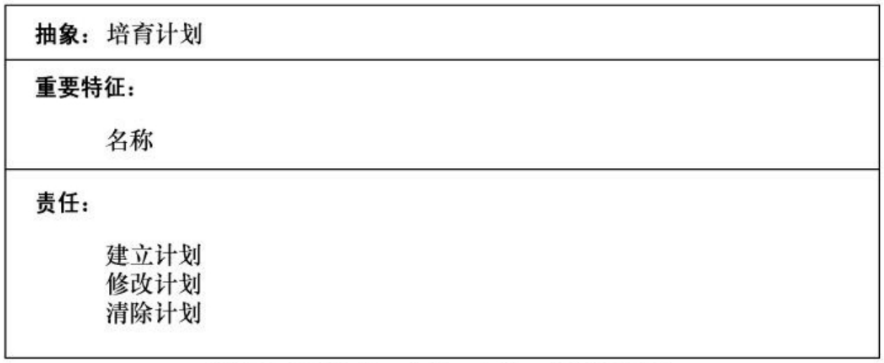
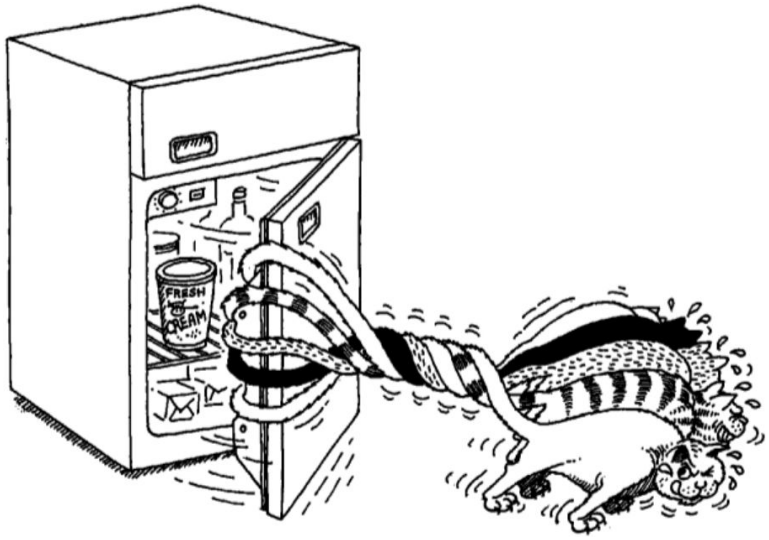


图2-8　培育计划的抽象

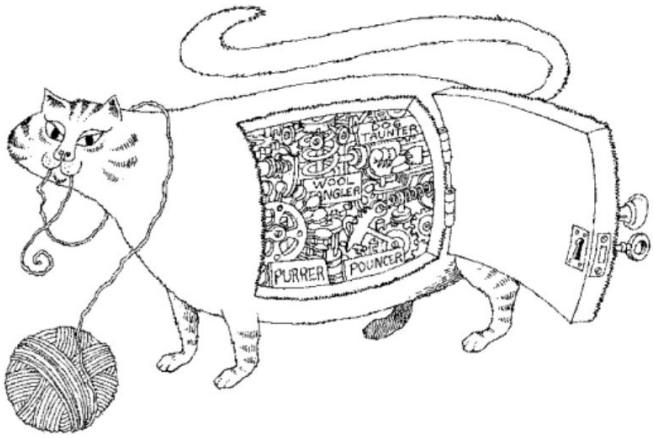
这个例子指出，没有对象是孤立的，每个对象都与其他对象协作，实现某些行为。[1]这些对象之间如何协作的设计决策，定义了每种抽象的边界，从而也定义了每个对象的职责和协议。



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：抽象关注的是对象可以观察到的行为，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的修改”[51]。

封装在不同的抽象之间提供了明确的边界，因此导致了清晰的关注点分离。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，如根的职责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，而是仅仅依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。

关注点分离

我们没有让加热器负责维持一个固定的温度，而是把这个职责赋予了另一个对象（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，它基于温度传感器和加热器的简单语义基础之上。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职责，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职责（参见图2-9）。

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。

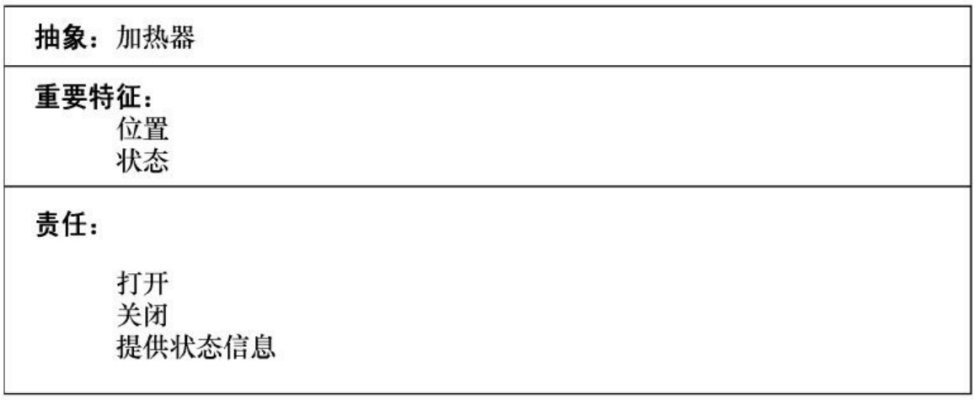


图2-9　加热器的抽象

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，利用一个开放的散列表。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：使用一个开放的散列表（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外部可见的行为没有影响。

明智的封装让可能改变的设计决策局部化。随着系统的演进，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。抽象让我们既能改变表示方法，同时又不影响其客户，这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，对于理解程序非常有价值。”[57]某些语言，如Smalltalk，没有模块的概念，所以类就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。

模块化将抽象打包成独立的单元

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NOT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。一个大型系统，如命令与控制系统，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。它不仅造成了文档噩梦，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的分组，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，增量地将声明从模块的实现移到它的接口要轻松得多、稳定得多。

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引起许多悲叹和愤怒（别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，可能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

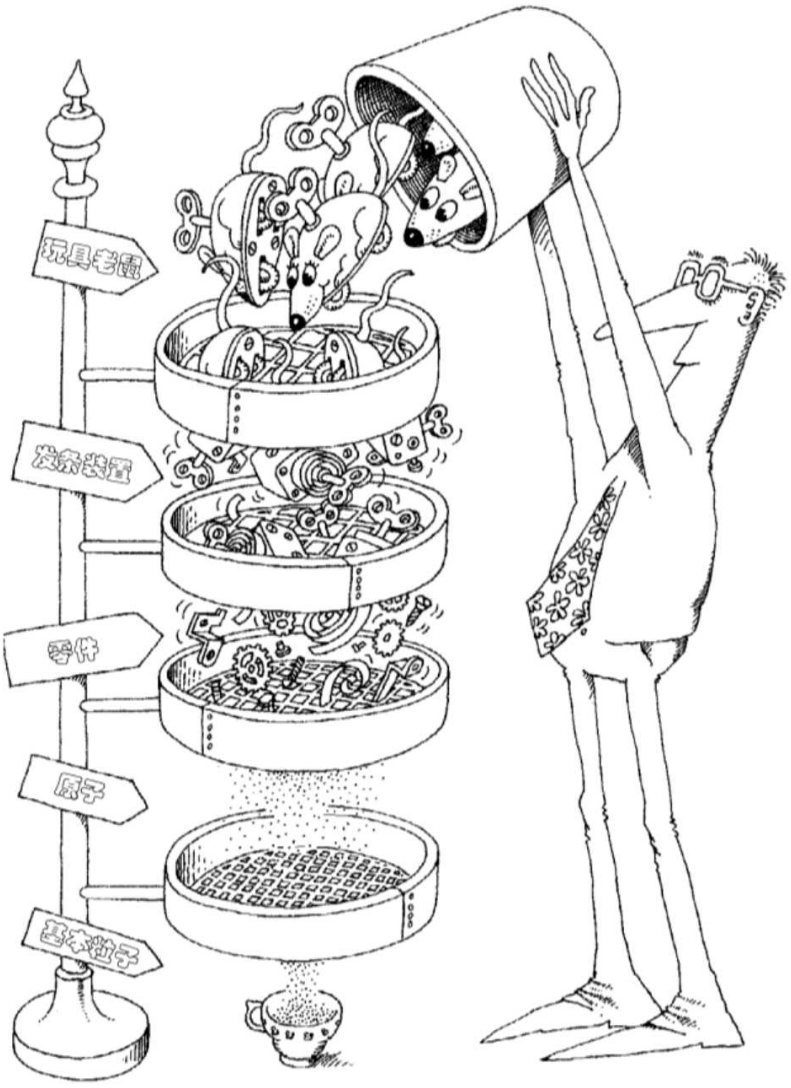
继承是最重要的“是一种”层次结构，前面曾提到，它是面向对象系统的基本要素。继承基本上定义了类之间的关系，在这种关系中，一个类共享了一个或多个类（分别对应于单继承或多继承）中定义的结构或行为。继承因此代表了一种抽象的层次结构，在这个层次结构中，一个子类从一个或多个超类中继承。一般来说，子类会扩展或重新定义超类中的结构和行为。

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。

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。通过比较新的概念和已经熟悉的概念，继承让我们能够定义新的软件，这就像对新来的同事介绍某些概念一样。”[64]



抽象构成了一个层次结构

在抽象、封装和层次关系之间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：调用该类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]。在支持封装和继承方面，不同的程序设计语言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，可以让所有客户访问。

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，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）类和一个果蔬（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，即花卉果蔬类（FlowerFruitVegetable），它复制了来自花卉类和果蔬类的信息。

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、果蔬特有的属性。这两个类没有超类，它们是独立的。它们被称为“混入类（mixin class）”，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。

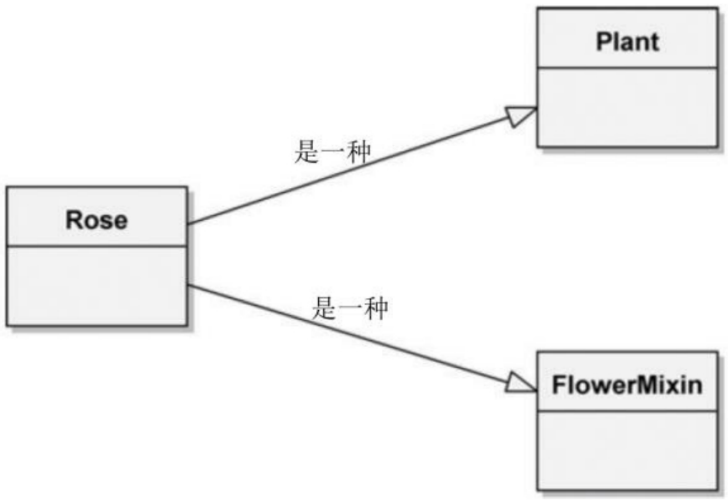


图2-10　从多个超类继承而得的Rose类

类似地，胡萝卜（Carrot）类可以如图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。

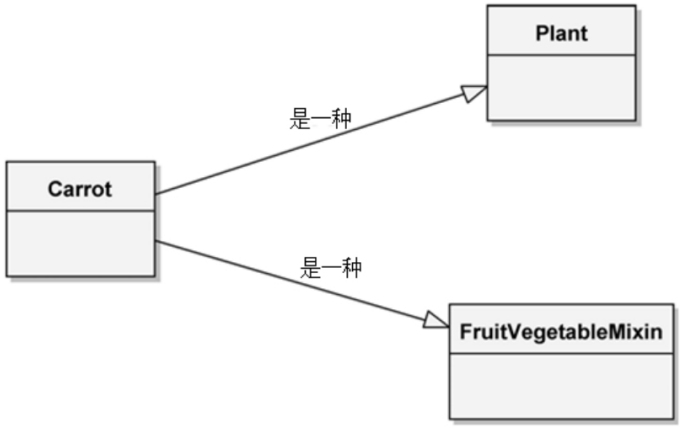


图2-11　从多个超类继承而得的Carrort类

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，如图2-12所示的那样。

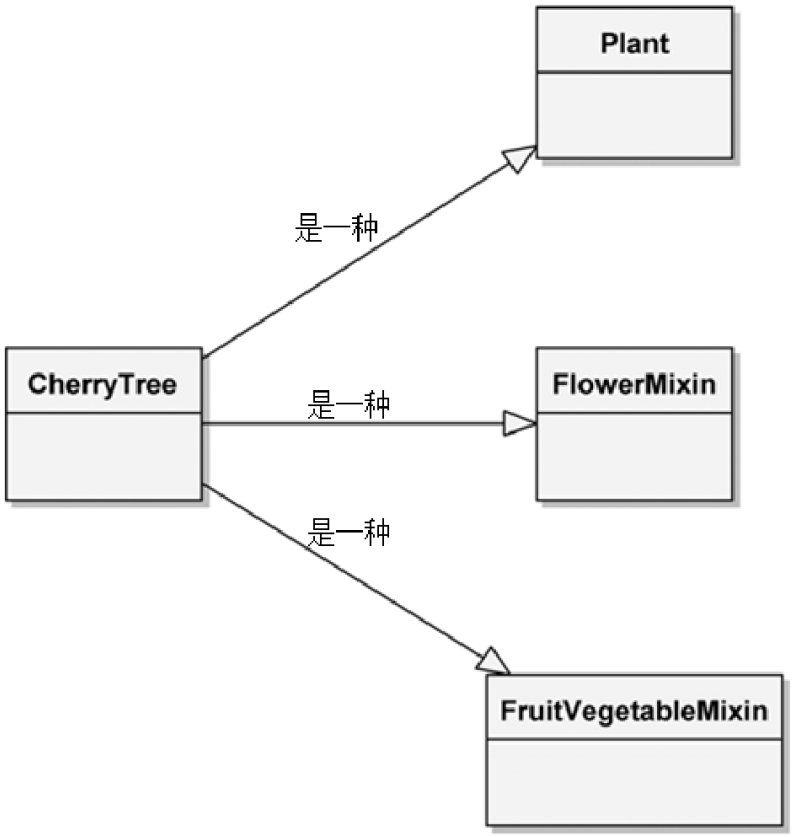


图2-12　从多个超类继承而得的CherryTree类

多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副本还是多份副本？（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副本（通过显式的限定符来区分不同的副本）。

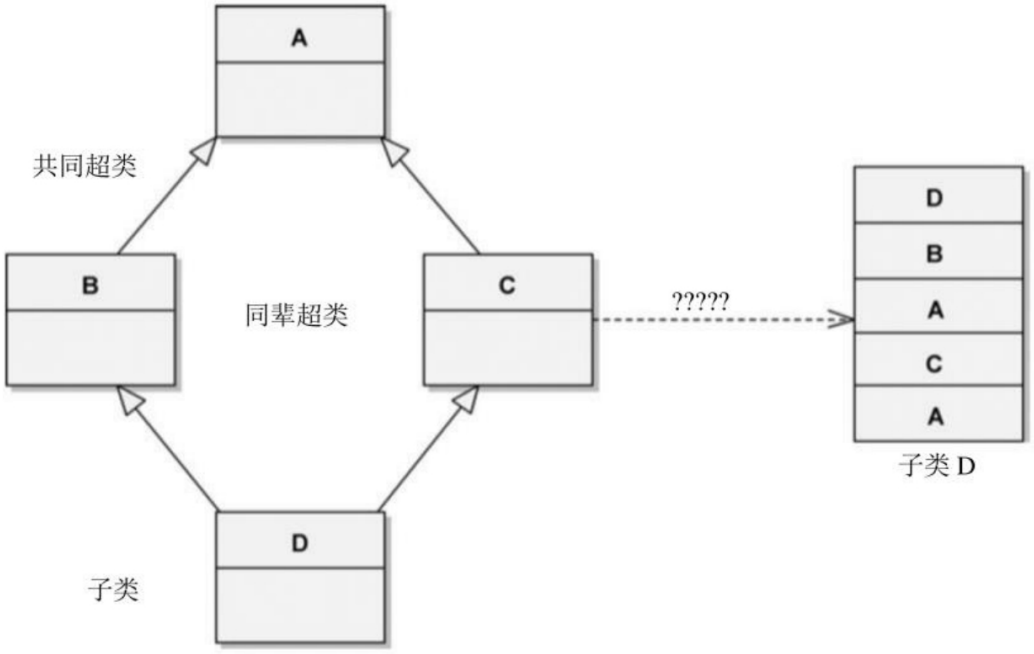


图2-13　重复继承问题

多继承常常被滥用。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“组成部分”关系称为聚合（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

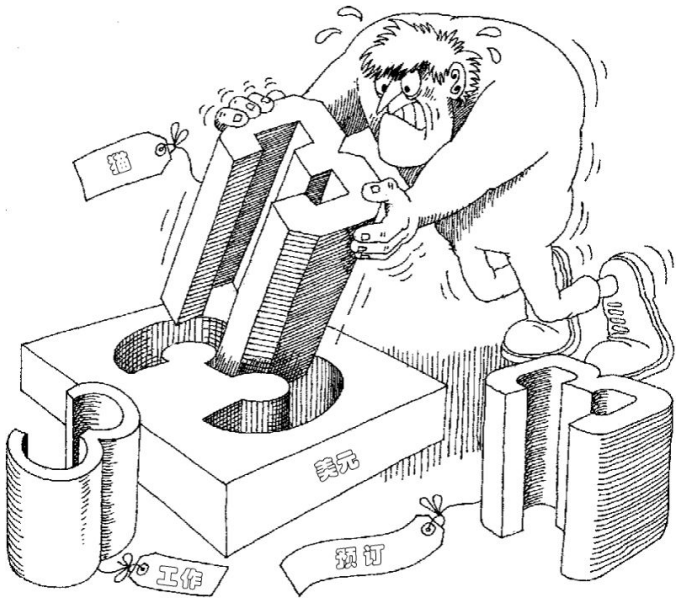
类型的概念主要来自于抽象数据类型的理论。Deutsch指出，“一个类型是关于结构或行为属性的准确描述，一组实体共享这些属性”[68]。为方便起见，我们互换地使用术语类型（type）和类（class）。[2]虽然类型和类的概念相似，我们仍然把类型作为对象模型的一个独立要素，因为类型的概念对于抽象的含义有着特别不同的强调。具体来说，我们认为：

> “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，其中我们的领域规则规定并强制保证某些合法的抽象组合。



强类型防止将抽象弄混

强类型让我们可以利用编程语言来强制保证某些设计决策，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，即使是基类接口上的一个小改动，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—调试循环相当烦琐，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，特别是在大规模编程中。

类型的例子：静态类型和动态类型

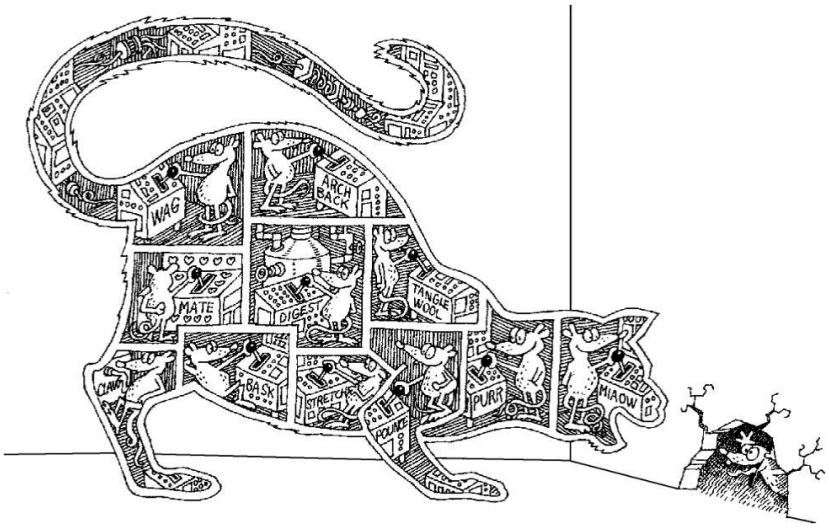
类型的强与弱和类型的静态与动态的概念是完全不同的。类型的强与弱指的是类型一致性，而类型的静态与动态指的是名字与类型绑定的时间。静态类型（也称为静态绑定或早期绑定）意味着所有变量和表达式的类型在编译时就固定了，动态类型（也称为延迟绑定）意味着所有变量和表达式的类型直到运行时刻才知道。一种语言可以既是强类型的也是静态类型的（Ada），既是强类型的也支持动态类型（C++、Java），或者既是无类型的也支持动态类型（Smalltalk）。

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。

第三，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ActiveTemperatureSensor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。这是同时使用封装、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：

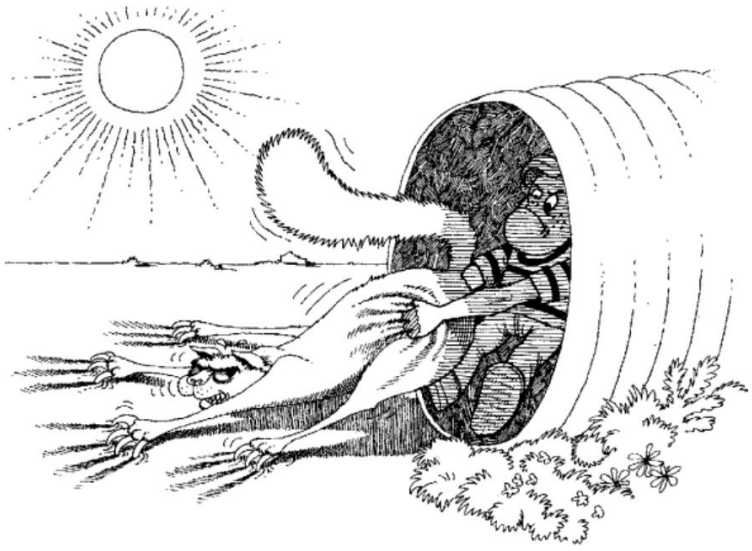
- 表达式计算的瞬时结果；
- 过程执行时的局部变量；
- 自有变量（像ALGOL 60中那样）、全局变量、堆中的值，它们的存在时间与它们的有效范围不同；
- 在程序执行之间存在的数据；
- 在程序的不同版本之间存在的数据；
- 比程序生命期长的数据[79]。

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，如果必须改变一个对象的类，更是如此。

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，当对象的创建者不存在了的时候，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）、关注点分离、模式、稳定的中间状态。根据我们的经验，应用对象模型还可以得到另外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，利用对象模型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。

第三，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关注点分离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，对象模型引起了对人类认知工作的兴趣。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。

---

[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。

[2]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。

第3章　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经验法则，以打造高品质的抽象和机制。

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的“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。

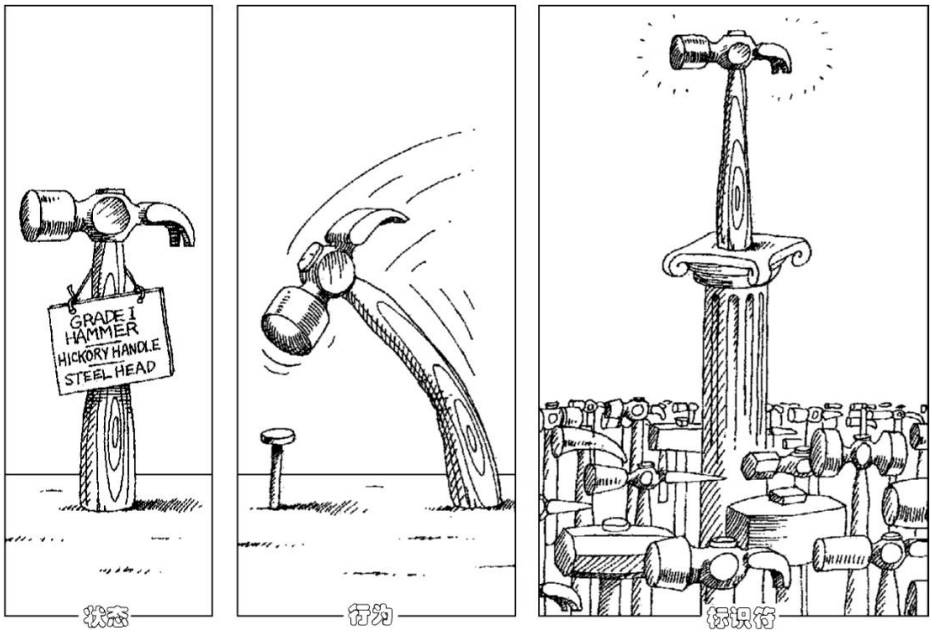
- 一个可以触摸或可以看见的东西；
- 在智力上可以理解的东西；
- 可以指导思考或行动的东西。

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，它们的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，展示某种定义良好的行为，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：

> “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它们共同的类中。‘实例’和‘对象’这两个术语可以互换使用。”

在接下来的3.1.2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在于对象内部存在状态。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。

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，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（UML）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。当在具体化时，

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）。

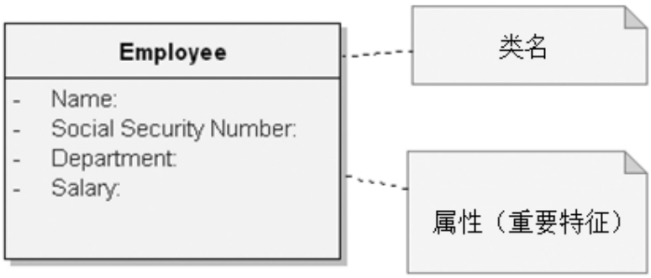


图3-1　Employee类和属性

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。

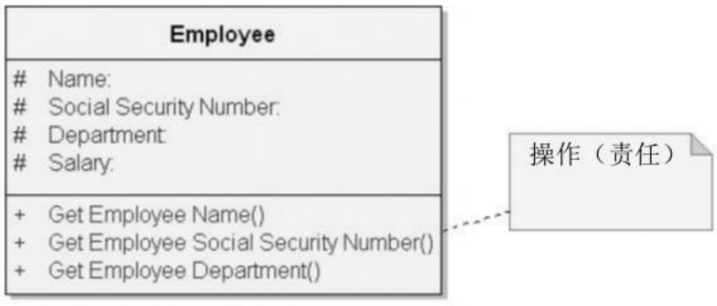


图3-3　具有保护属性和公有操作的Employee类

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。如果要修改它的表示形式，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（职责），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。

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、它们与对象职责的关系以及它们怎样让对象实现其职责。

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五种操作。[2]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。

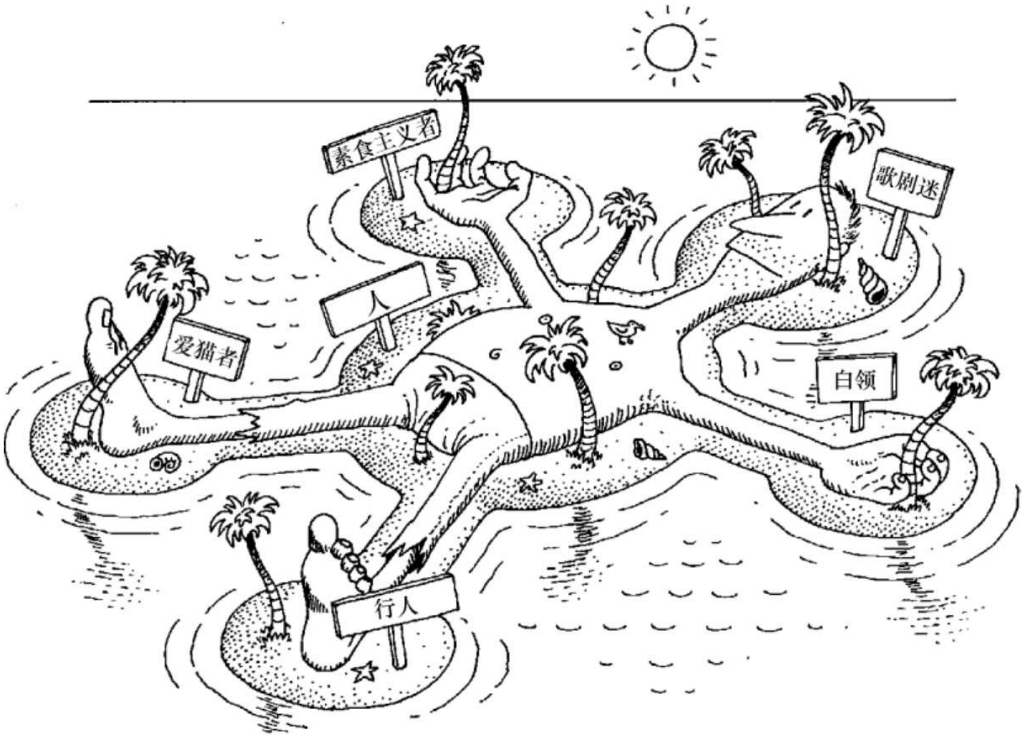
另外两种操作是公共的，它们体现了创建和销毁一个类的实例的需要。

- 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．角色和职责

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

“职责意味着表达对象的一种目标以及它在系统中的位置。一个对象的职责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职责。

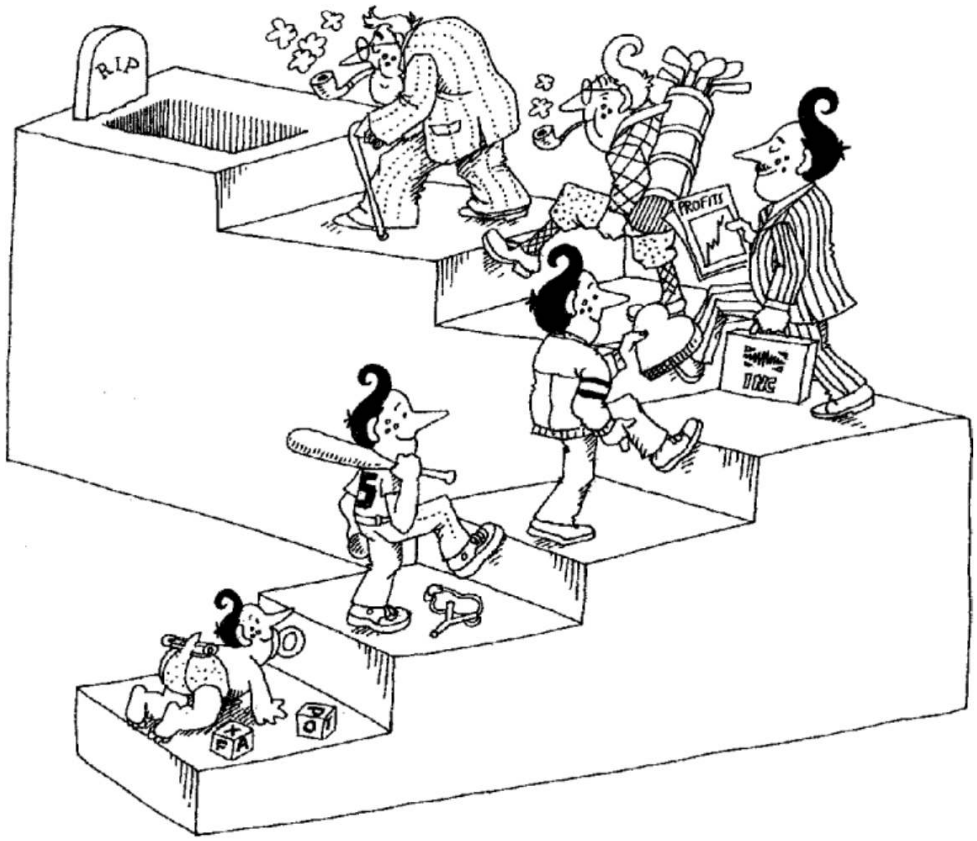
实际上，大多数有意思的对象在它们的生命周期中扮演许多不同的角色。考虑下面的例子[10]。

■ 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许多不同的角色

在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，我们细化这些角色，设计出特定的操作，实现每个角色的职责。

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，只有在显式地操作它时，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

Khoshafian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丧失而无法恢复。

示例3-2

考虑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四个位置，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，但其他三个名称是指向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。

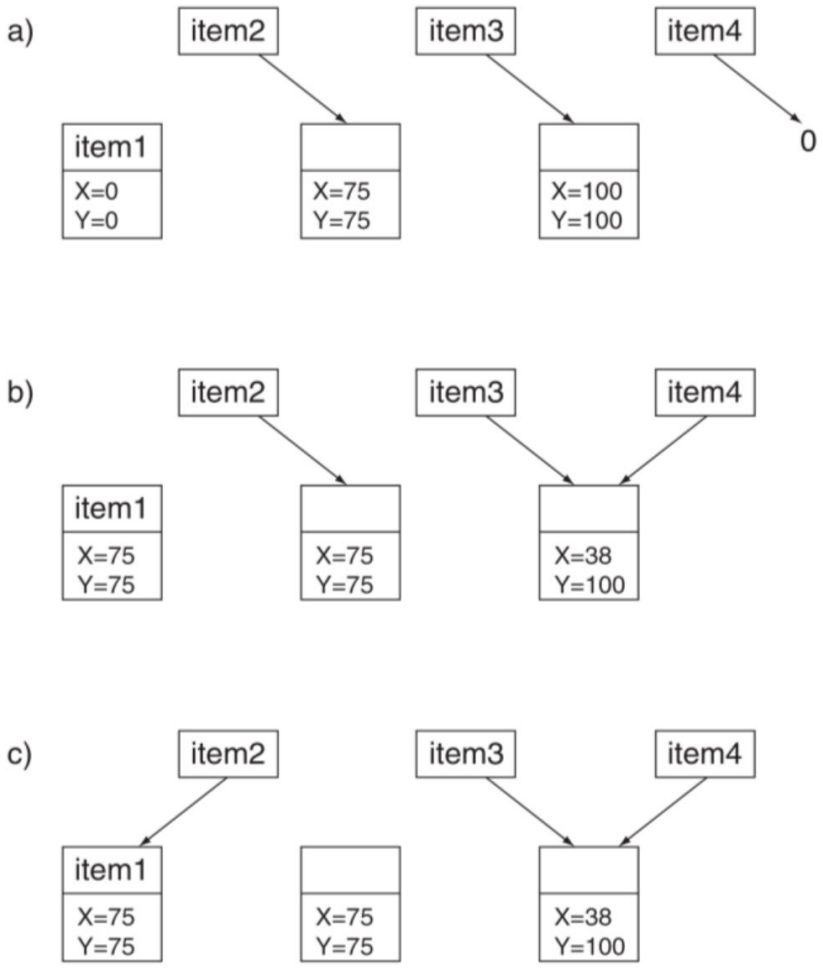


图3-4　对象标识符

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使用这两种说法。

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，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。[3]

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水流的。可以看到，FlowController对象（水流控制器）有一个到Valve对象（阀门）的链接。Valve对象有一个到DisplayPanel对象（显示面板）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DisplayPanel对象上的操作（目的是改变它显示的东西）。这种关注点分离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。

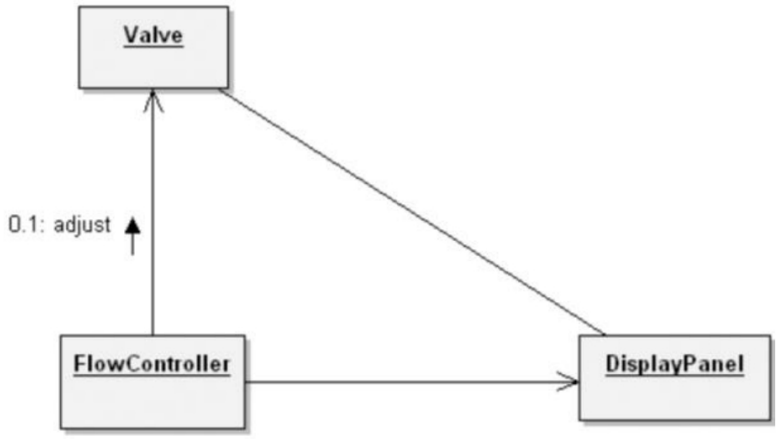


图3-5　链接

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域中的一个真实对象。

在如图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展示了这些职责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TemperatureRamp类，它在概念上是一种时间/温度的映射关系（参见图3-6）。

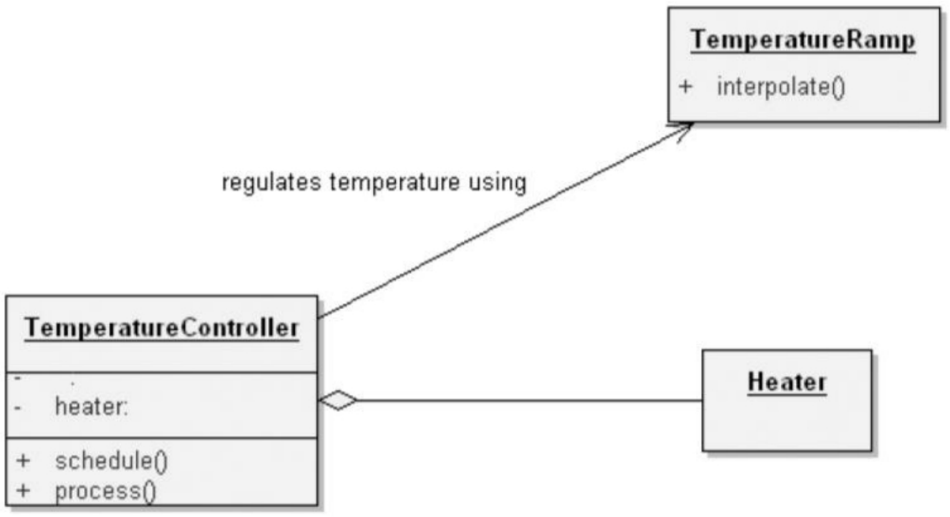


图3-6　聚合

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）。

有一个行为显然不是这种抽象需要具备的，即控制加热器来实现特定的温度坡度。我们喜欢更好的关注点分离，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，我们必须选择以下三种同步方式之一。

- 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，服务提供者保证互斥。

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。

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，这两个术语之间又存在着重要的差别。

3.3.1 什么是类，什么不是类

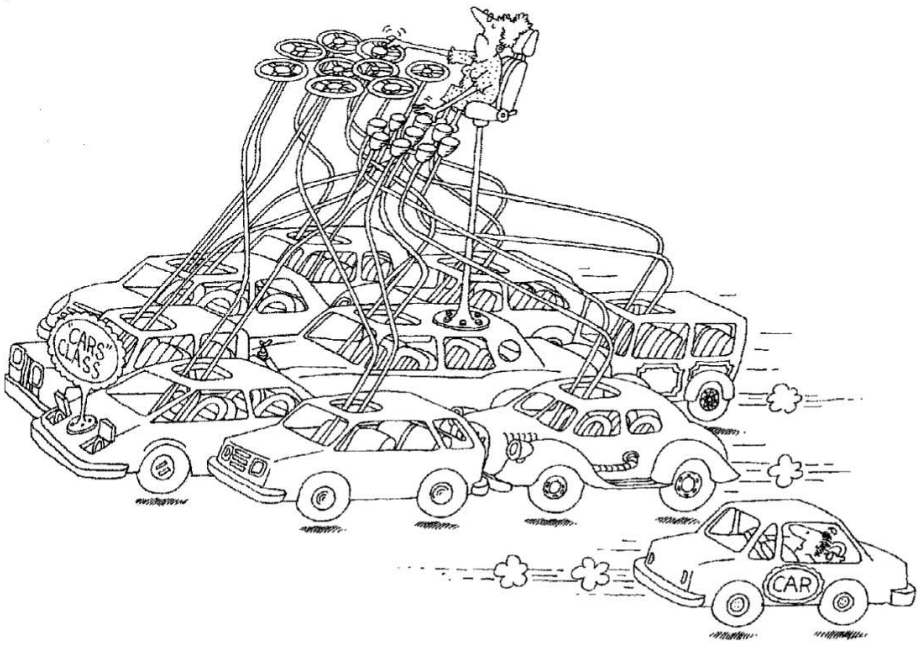
对象是存在于时间和空间中的具体实体，而类仅代表一种抽象，即一个对象的“本质”。因此，可以说Mammal类，它代表了所有哺乳动物的共同特征。要确定这个类中的某个具体的哺乳动物，则必须说“这个哺乳动物”或“那个哺乳动物”。

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，但它们都不能表示为一个单独的类。[4]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Snyder[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，从而被分解成较小的问题。没有别的情况比在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四个部分。

- 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四种不同的可见性。[5] 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。

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，对这种表示形式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。

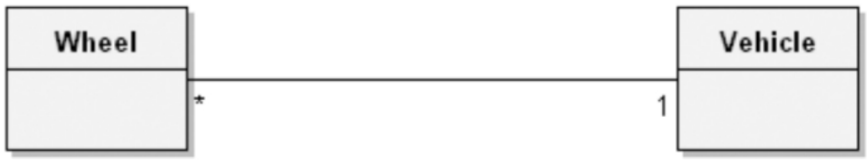


图3-7　关联

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。

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。

子类可以继承其超类的结构和行为

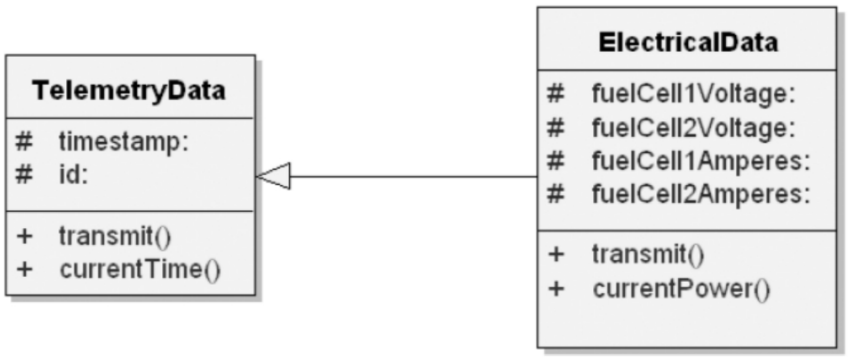


图3-8　ElectricalData（用电数据）从超类TelemetryData（遥测数据）中继承

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。

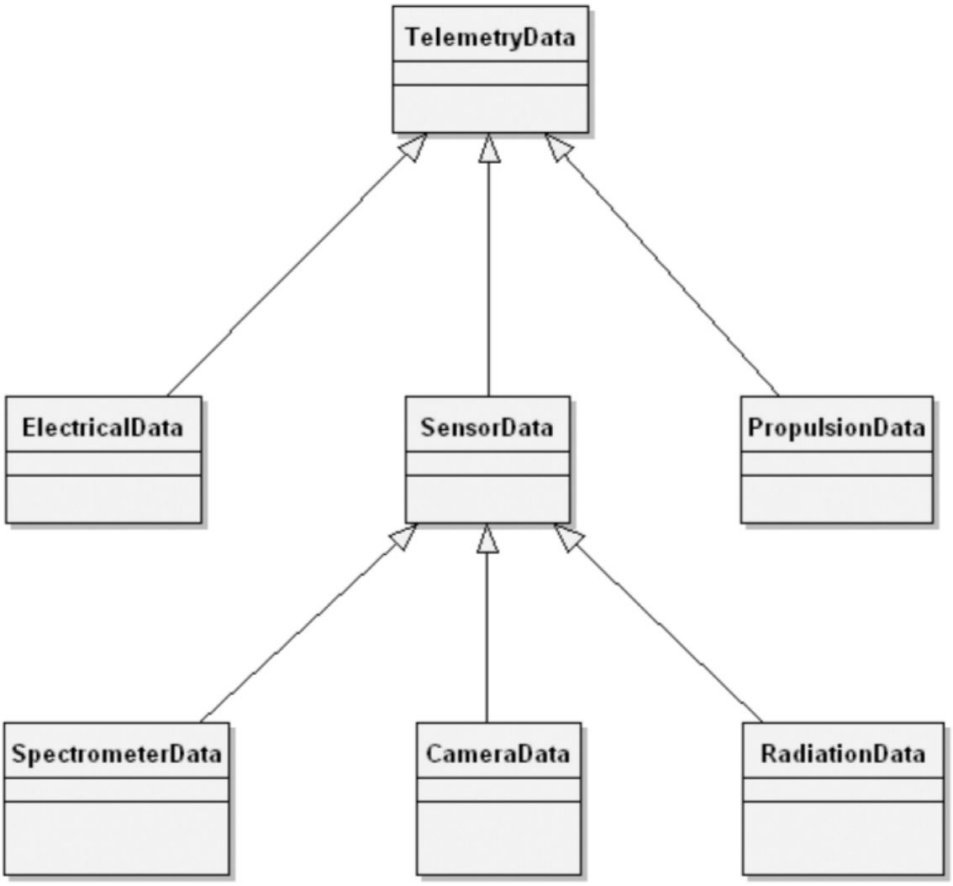


图3-9　单继承

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，在重载解析时不考虑函数的返回值类型）。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。[6]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。Kaplan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延迟绑定。具体来说，如果将方法声明为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声明就执行延迟绑定。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3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。

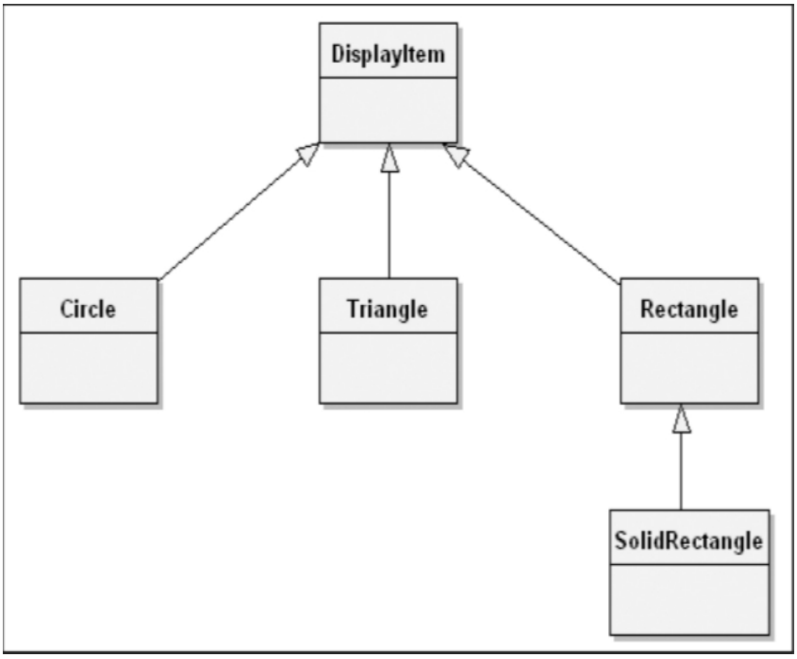


图3-10　DisplayItem类图

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，直至找到该消息，或者直至到达最顶端的基类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，从而决定它是延迟绑定的。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，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，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，Vlissides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、支票账户以及某些股票和债券。

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。[7]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。第三，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，那么迟早会有人写出D类有两个父类B和C，而B和C又都以A作为父类……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）。

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。第三，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。

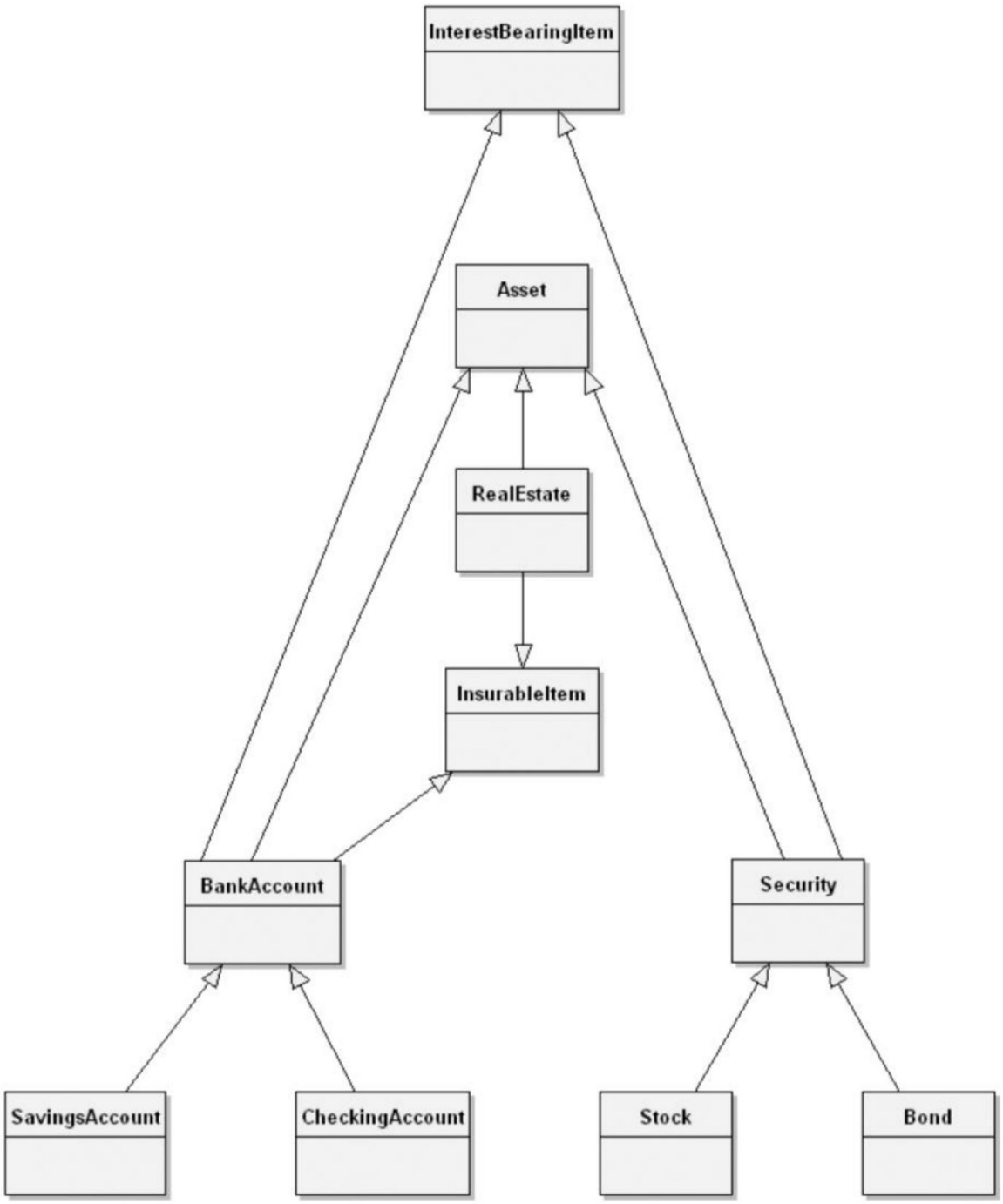


图3-11　多继承

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InterestBearingItem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。

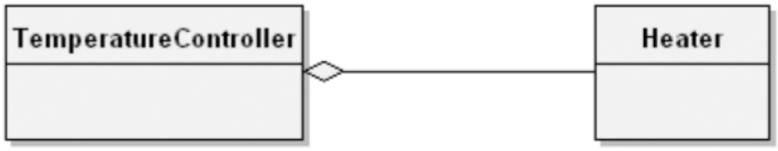


图3-12 聚合

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，细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？我们建议使用下面五个测量指标。

- 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。然而，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，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。

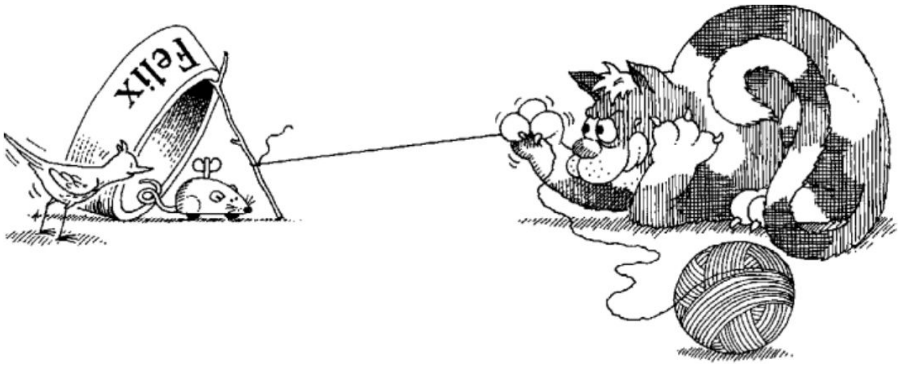
所谓完整，指的是类或模块的接口记录了某个抽象全部有意义的特征。充分性意味着最小的接口，但一个完整的接口就意味着该接口包含了某个抽象的所有方向。因此完整的类或模块是足够通用的，可以被任何客户使用。完整性是一种主观判断，有可能做过头。为某个抽象提供全部有意义的操作会让用户不知所措，通常这也是不必要的，因为许多高级操作可以由低级操作组合得到。出于这个原因，我们也建议类和模块应该具有基础性。

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、修改并进一步细化这个接口。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“细粒度的（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。决定将一个行为提取为一个方法还是多个方法，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，需要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

在确定存在某个操作，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，有一条有用的指导原则，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成本最小化，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。

---

[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。

[2]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]。

[3]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地分配对象。

[4]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。

[5]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。

[6]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。

[7]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。

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。既然存在许多类似面向对象设计中的问题，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。

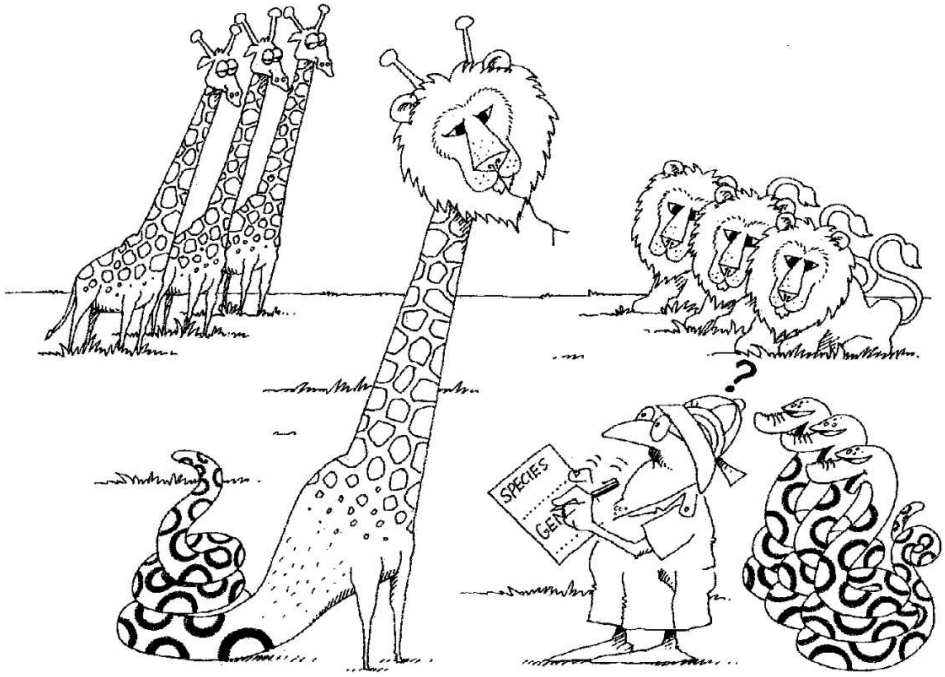
直到18世纪，社会上普遍的科学认识仍然是所有活着的有机体可以从最简单到最复杂进行分类，同时复杂度的测量又是相当主观的（不奇怪，人通常被放在这个列表的最顶端）。但在18世纪中期，瑞士植物学家Carolus Linnaeus提出了一种更详细的生物分类学，即根据他所谓的“属”和“种”。

一个世纪之后，达尔文提出了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。达尔文的理论以对物种的明智分类为基础。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。但如果想说明适应性的程度，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，分类的结果在很大程度上也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，化学家拉瓦锡才列出了第一个元素清单，包含大约23种元素，后来人们发现其中的某些根本不是元素。此后，新的元素不断发现，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，人们发现了具有类似化学性质但原子量不同的元素，这引入了同位素概念。



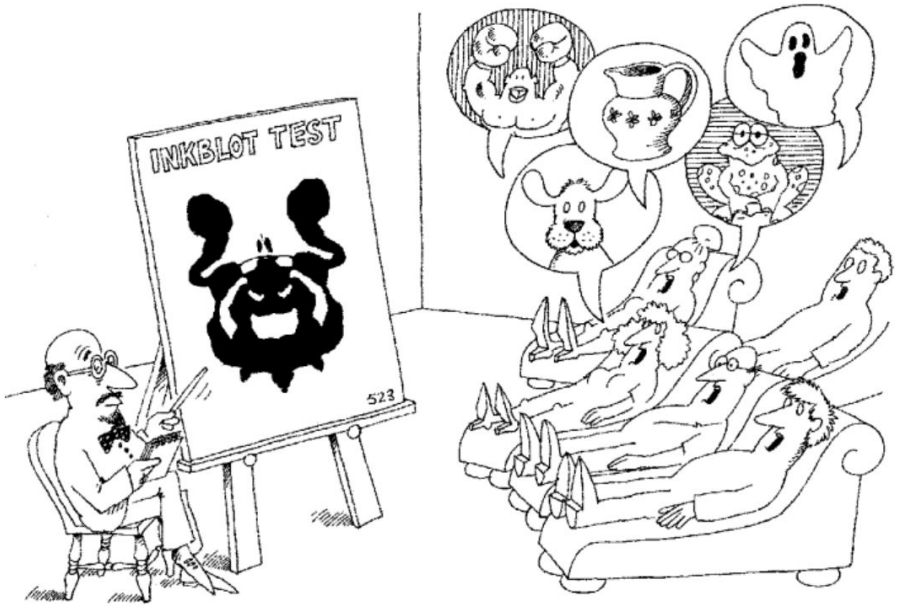
分类是我们组织知识的手段

这里的经验很简单：Descartes指出，“发现一种秩序不是容易的任务……但是一旦秩序被人发现，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得到了广泛采用：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性列表，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。

一个分类问题

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，测试参与者提出了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。大多数测试参与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，有一个测试参与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。

在我们的测试参与者中，火车的分组发生了显著的变化，大多数测试参与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。

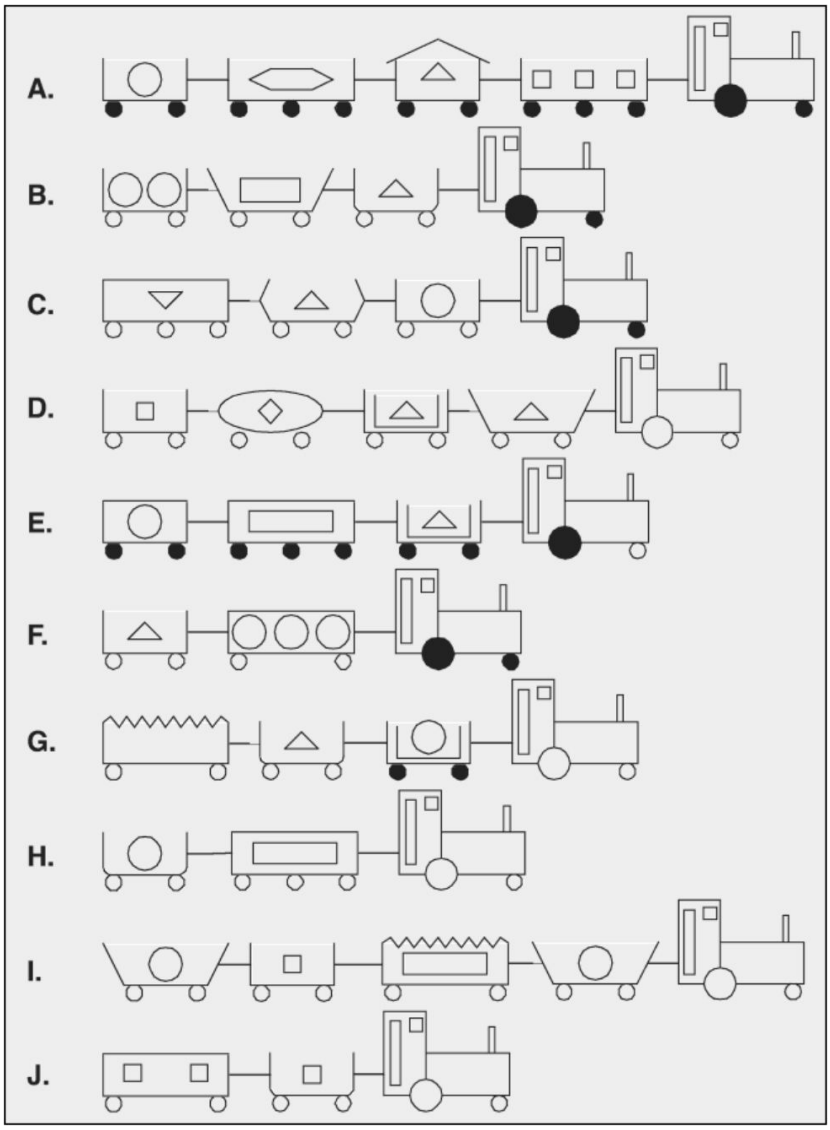


图4-1　一个分类问题

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计划搏斗的开发者来说，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强调了职责。职责代表的是“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。对象的职责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，我们将具有共同职责的事物划分为一组，让超类包含一般的职责，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，将行为职责分配到这些对象上”[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，指出这些关键抽象在其他相关系统中已证明有用。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；
- 细化通用模型，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。领域分析不太会是一项独立的活动，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]。分析团队分析每个场景时，必须确定参与该场景的对象、每个对象的职责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关注点分离。随着开发过程的继续，这些初始的场景将得到扩充，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职责。场景也将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。[2] 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[3]，分析师在上面用铅笔写下类的名称（在卡的顶部）、它的职责（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职责，将某些职责集中起来形成一个新的类，或者（更常见的情况）将一个类的职责分解到粒度更小的类中，也许会将这些职责放到其他的类上。

CRC卡片可以摆放在不同的位置，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。Abbott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侯选操作。

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，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]。这些侯选对象通常来自于周围的环境、基本输入输出，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Mellor[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职责。

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，对输入和输出数据进行检查和跟踪，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[4]，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，本身并不是终点。它们应该被看成是一些工具，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细化这些关键抽象和机制。

4.3.1 确定关键抽象

确定关键抽象与具体领域极为相关。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。

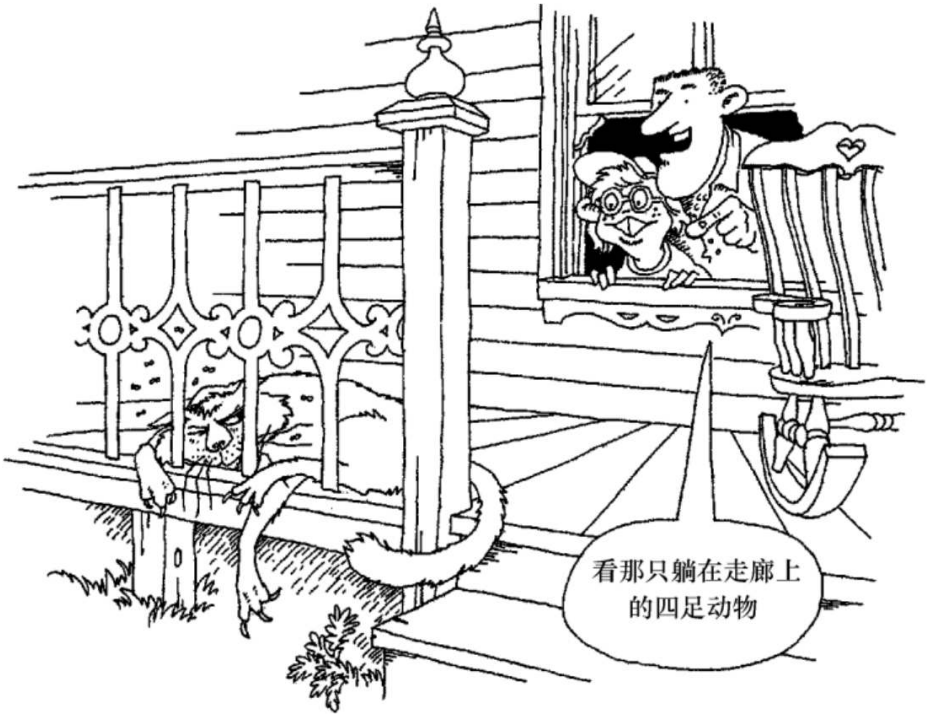
1．细化关键抽象

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，这称为“类提升”[56]。类似地，我们可能发现某个类太一般化，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

许多开发者常常忽略了为事物正确地命名（以便让名称能反映其语义），但这对于记录我们所描述的抽象的本质是很重要的。编写软件应该像写英文散文那样认真，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。
- 使用下画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：

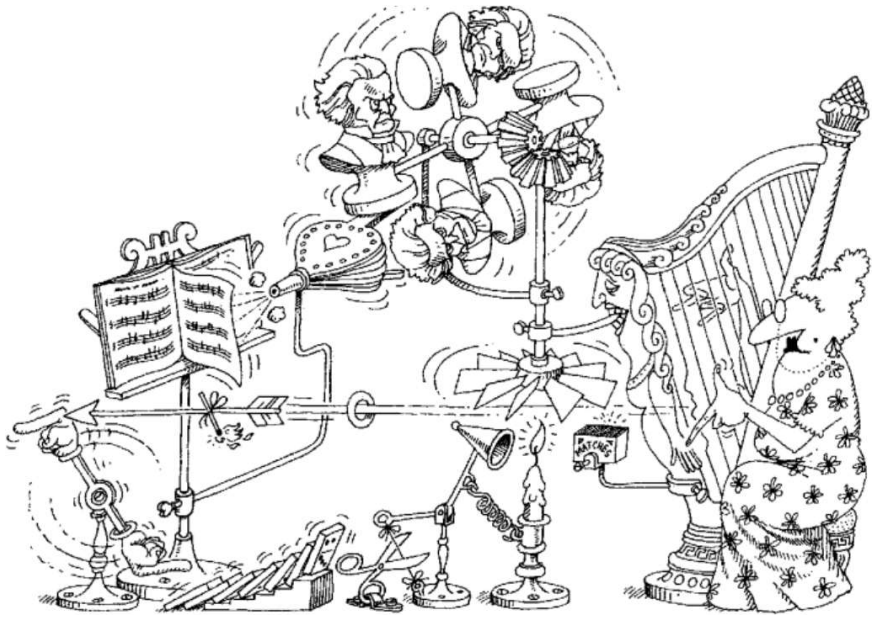
- 一个机械连接装置，直接将加速踏板和燃油喷射器连接起来。
- 一个电子装置，连接加速踏板下的压力传感器和计算机，由计算机控制燃油喷射器（一种电线控制的机制）。

■ 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，我们有惯用法。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大家广泛接受的习惯。[5]例如，在CLOS中，没有程序员会在函数名或变量名中使用下画线，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，Coplien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、被查看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对象。客户对象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。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。接下来的三章中将关注表示法、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。

---

[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]。

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

[3]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。

[4]政治斗争原因和历史原因显然不是光明正大的原因。

[5]惯用法有一种确定的特征，即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。

第2篇　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，不论是设计桥梁、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复制那些看起来不错的特征，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 Petroski
>
> *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，需要传播的一种关键知识，即这种技术如何能够成功地使用或形成（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，如同被记录在白板、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，正如怀特海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]。第三，一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烦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：“软件开发现在是，而且永远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，之后人们对它们进行了修改和提炼。其中许多方法学都在某些方面很强，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。实践者们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（OMG）提交一种标准的建模语言。OMG是一个协会，其宗旨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职责。

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种视图。Kleyn和Gingrich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，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，若类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）。

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

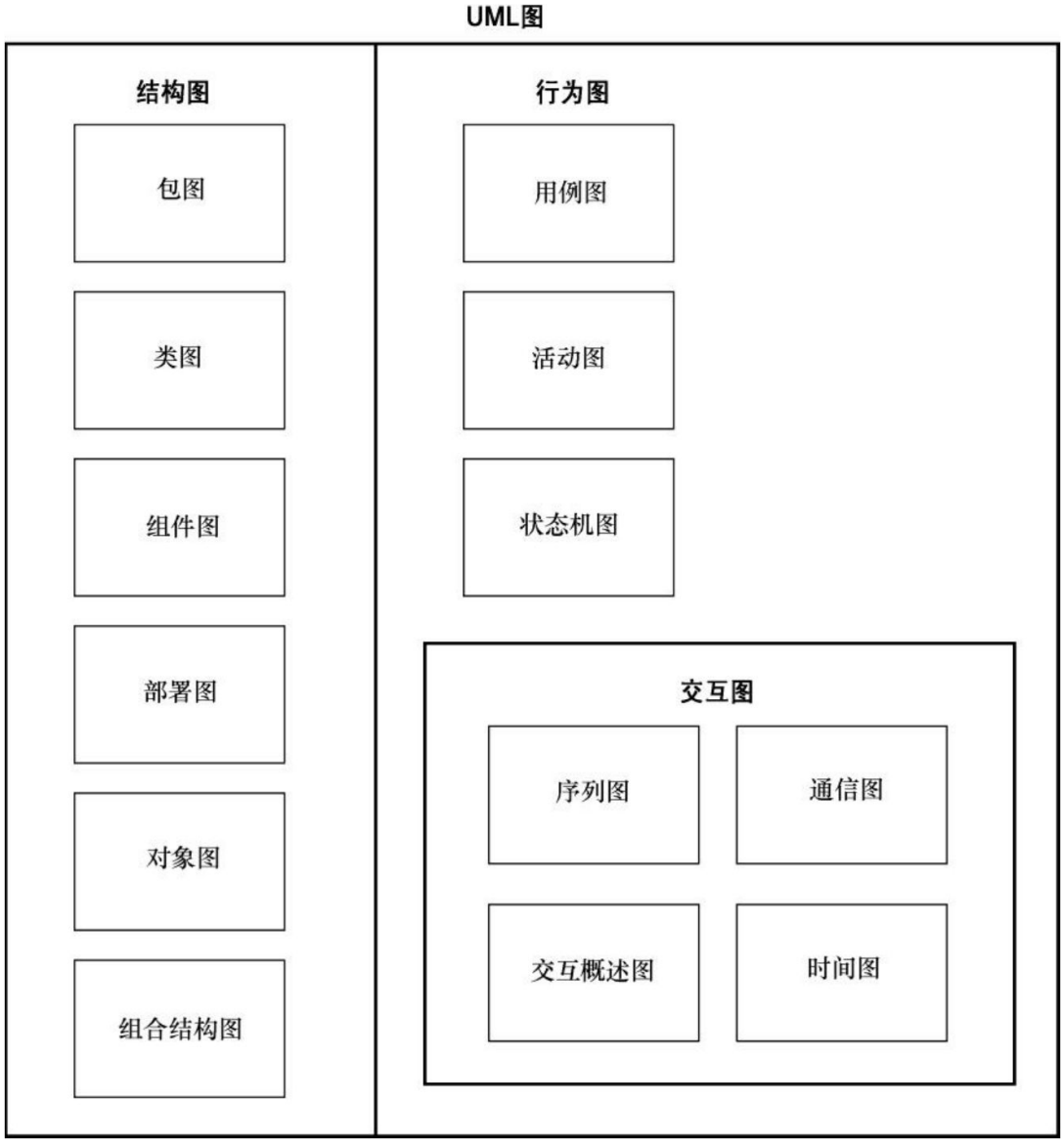


图5-1　UML图

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，我们可以提供一张交互图，与表示场景的对象图一起，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。

温伯格曾指出：“在其他设计领域，如建筑，草图是最常用到的图形手段，在创造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。总的来说，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，然而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，工具能让不好的设计者更快地创造出糟糕的设计。卓越的设计来自于卓越的设计者，而不是来自于卓越的工具。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种机制。他可以利用工具来定位某个组件包含的所有类。在查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。利用这些工具，开发者不必记住所有烦琐的细节，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，以便简化类的结构。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：（1）使用者既是面向对象开发的专家，也是问题域的专家；（2）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（表示代理的角色和职责，这些代理提供了系统的行为）以及交互图和状态机图（展示这些代理的事件次序行为）。类似的，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，目的是追踪需求。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。开发者不会在创建一个图之后就退下，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

本章余下的部分将针对面向对象分析和设计，来描述UML的语法和语义。利用第2章中介绍的溶液种植园系统的问题，我们将提供这种表示法的一些小例子。本章不会解释得到UML图的开发过程，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

在本书的第三部分，每一章都将介绍一个不同类型的应用，并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，或希望通过多重视图，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；
- 在多个层次上提供抽象——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，及其可见性和依赖关系。

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]。

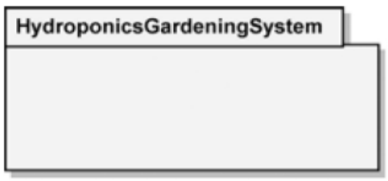


图5-2　HydroponicsGardeningSystem的包表示法

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]。

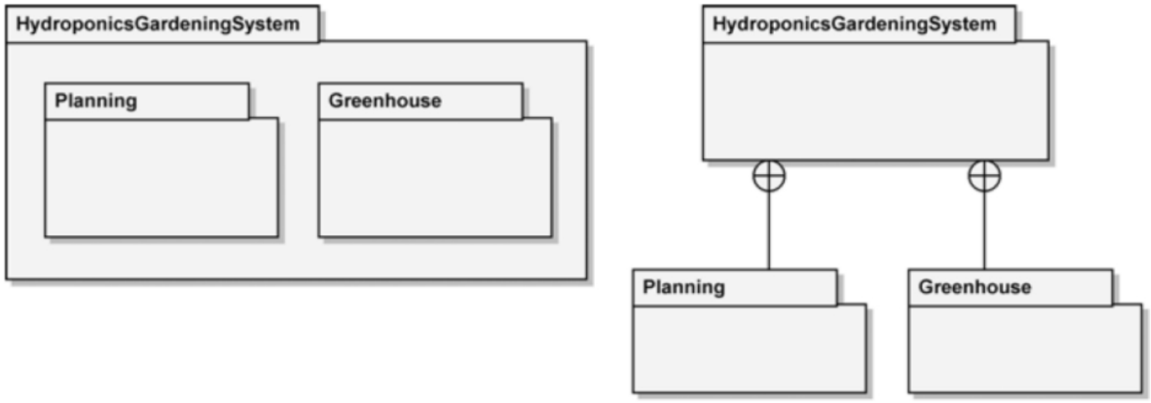


图5-3　包容元素的包表示法

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所处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。

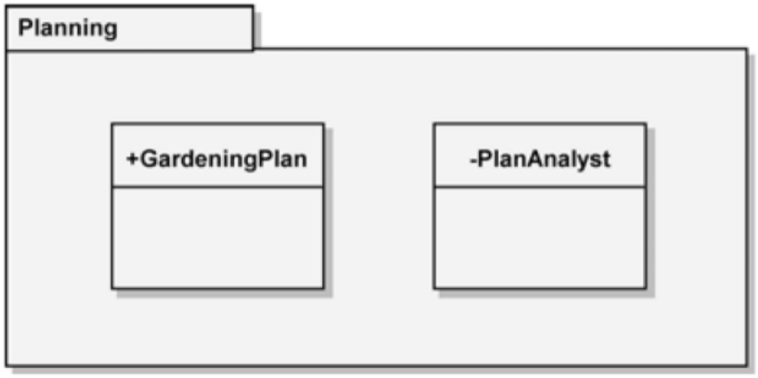


图5-4　Planning包内元素的可见性

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，前面在讨论包的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职责。

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，通过在符号（«»）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、细化、允许和使用[53]。

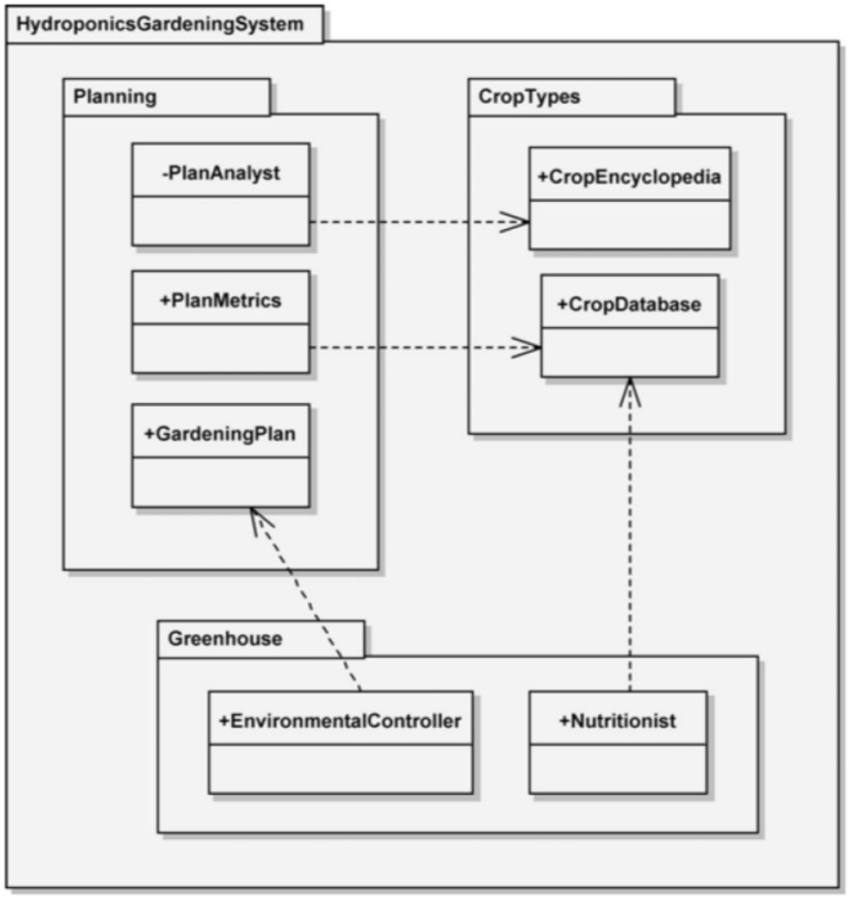


图5-5　HydroponicsGardeningSystem的依赖关系表示法

如果两个包中的多个元素之间存在依赖关系，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，放在符号（«»）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展示了将如图5-5所示的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。

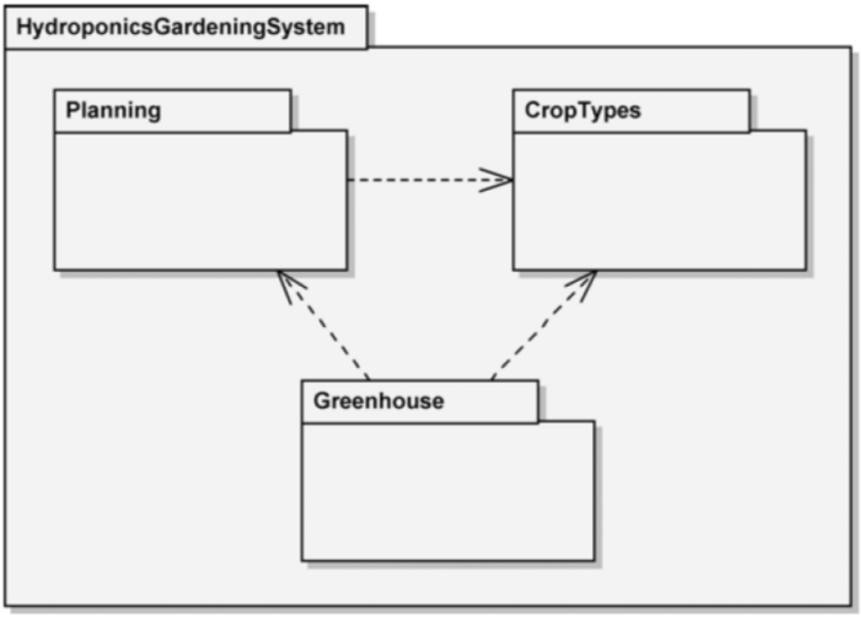


图5-6　包容的元素依赖关系的聚合

5.2.4　基本概念：包图

到目前为止，我们讨论了可以在包图中出现的元素：

- 包表示法
- 元素可见性
- 依赖关系

包图是一种UML 2.0结构图，它以包作为主要展现的UML元素，同时展示了包间的依赖关系。

但是，包的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5-5中展示了这一点。它也可以用于那些不属于结构图的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。这样做可能是为了在很大的系统中体现清晰性，或对工作进行划分。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。

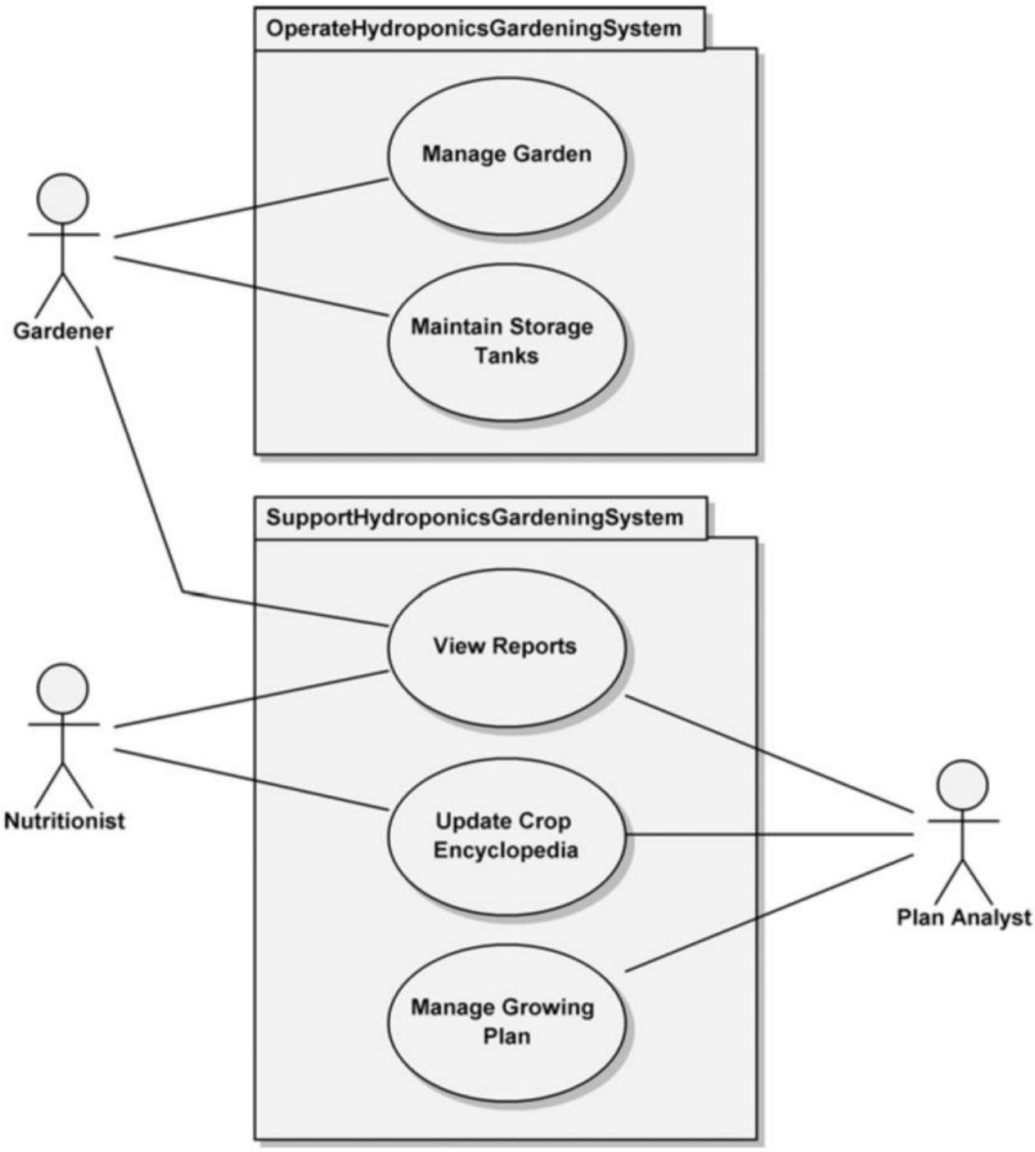


图5-7　用于分割包的表示法

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。

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一枚硬币的两面——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（Plans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]。

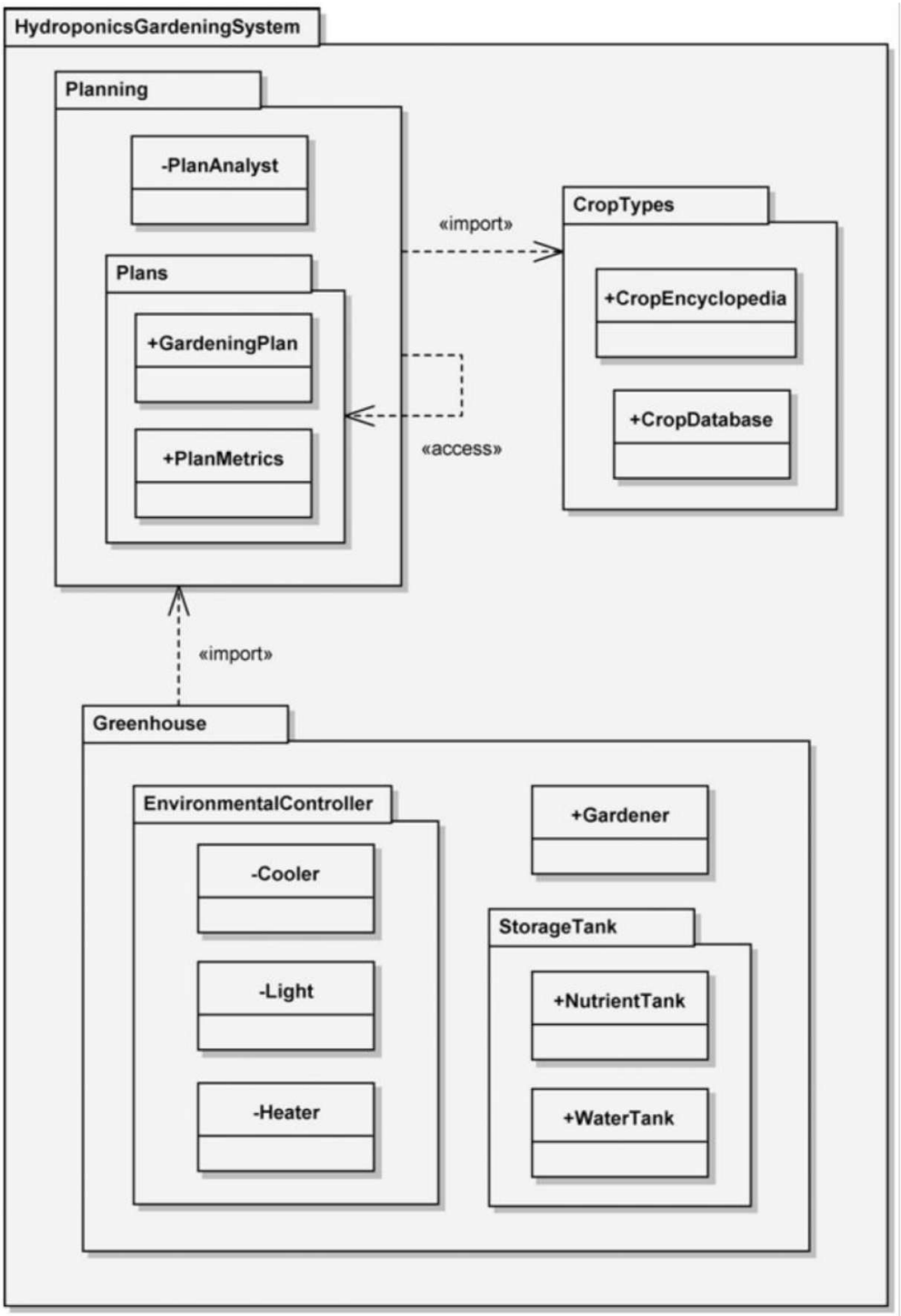


图5-8　HydroponicsGardeningSystem中的包导入

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散。系统中的每个类要么处于一个组件中，要么处于系统的顶层。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]。

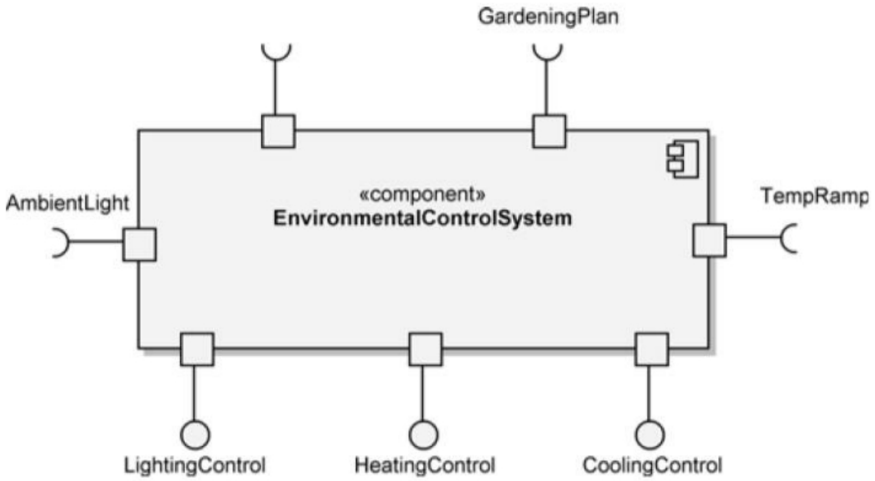


图5-9　EnvironmentalControlSystem中的组件表示法

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示意图，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溶液种植园系统的员工提供的，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]。

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

件功能的四个封装的组件：Environmental- Controller、LightingController、HeatingController和CoolingController [78, 79]。

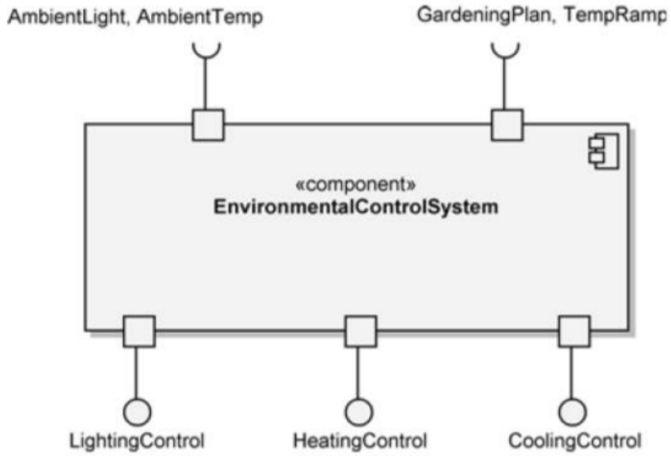


图5-10　带接口分组的组件表示法

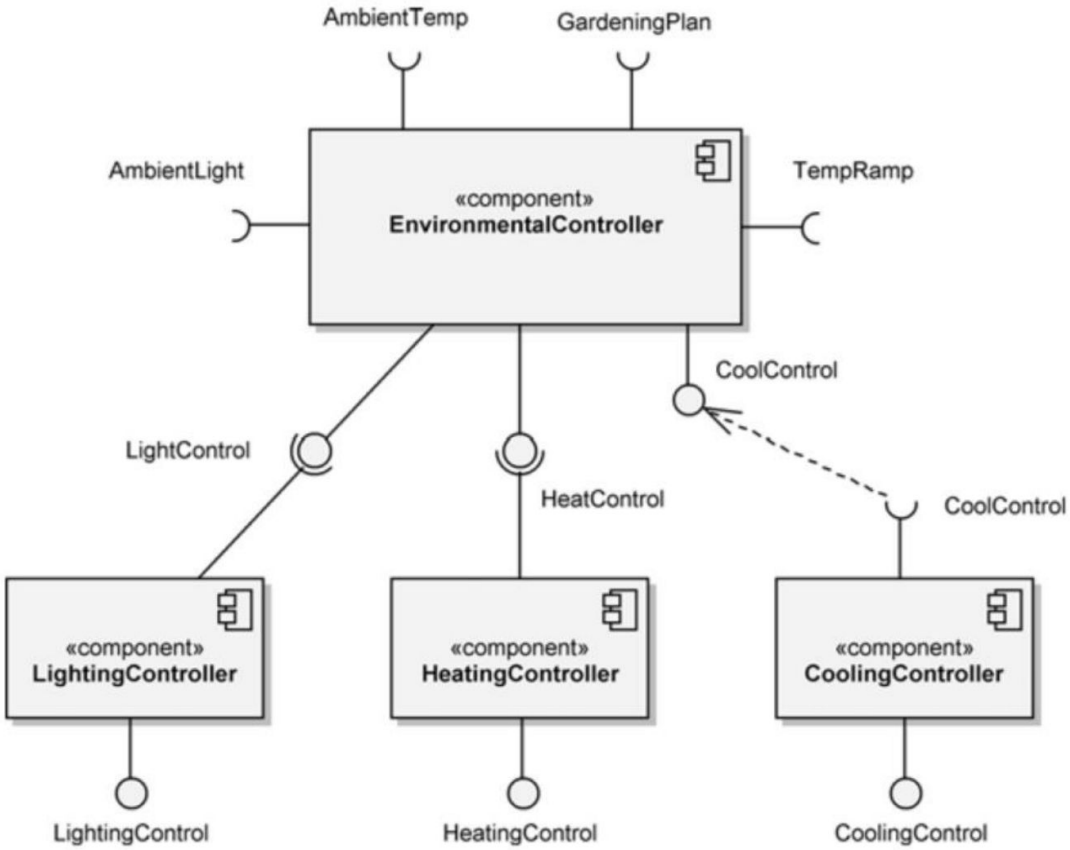


图5-11　EnvironmentalControlSystem的组件图

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CoolingController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个接口中的两个：CoolControl和AmbientTemp [82, 83]。

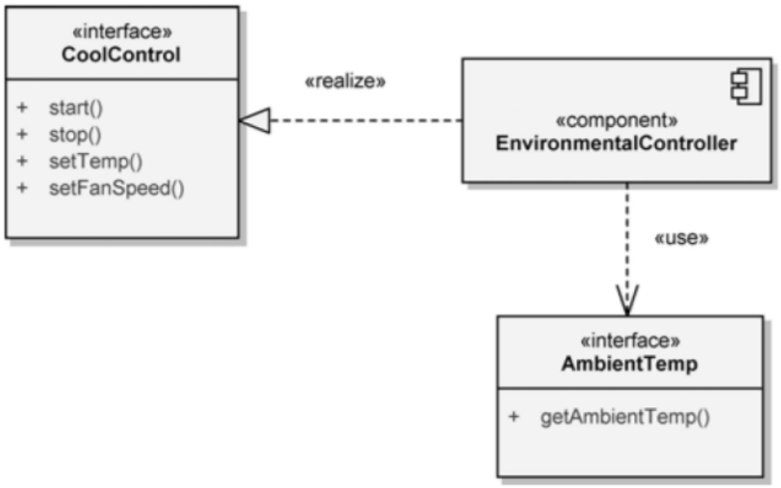


图5-12 EnvironmentalController的两个接口的规格说明

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、设置温度和设置风扇速度，它所包含的操作说明了这一点。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组件内的职责。

在图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»下面列出了三个提供的接口。对于图5-12中指定的CoolControl接口，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]。下一小节将讨论实现的概念。

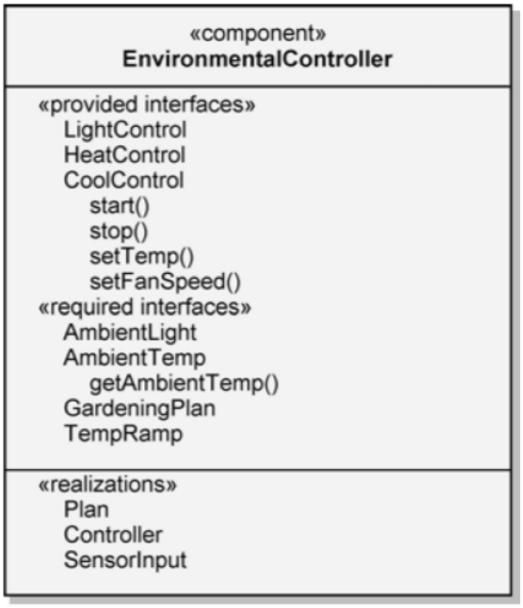


图5-13　EnvironmentalController的接口和实现的另一种表示法

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类来实现的。这三个类提供了组件接口宣称的所有功能。但是，要做到这一点，需要组件要求的接口所指定的功能。

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]。

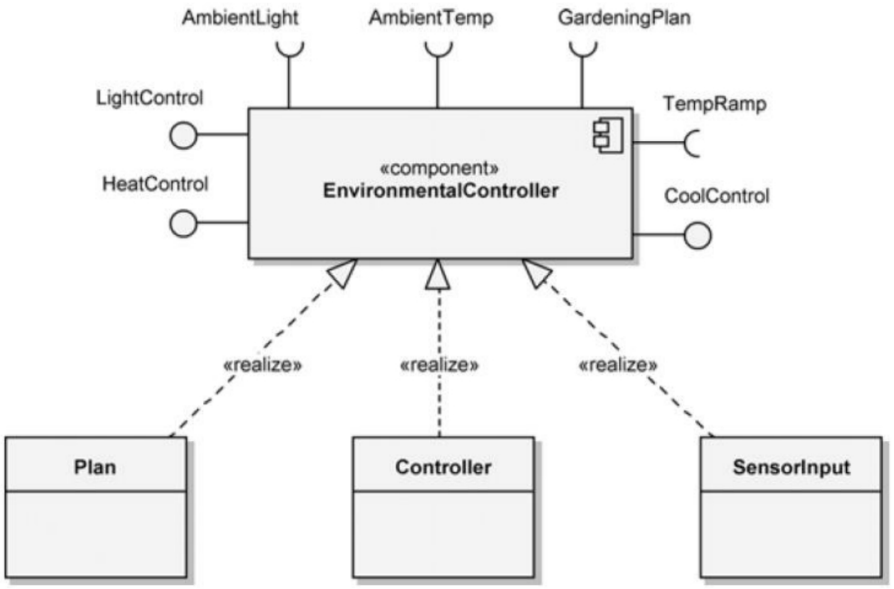


图5-14　EnvironmentalController的实现依赖关系

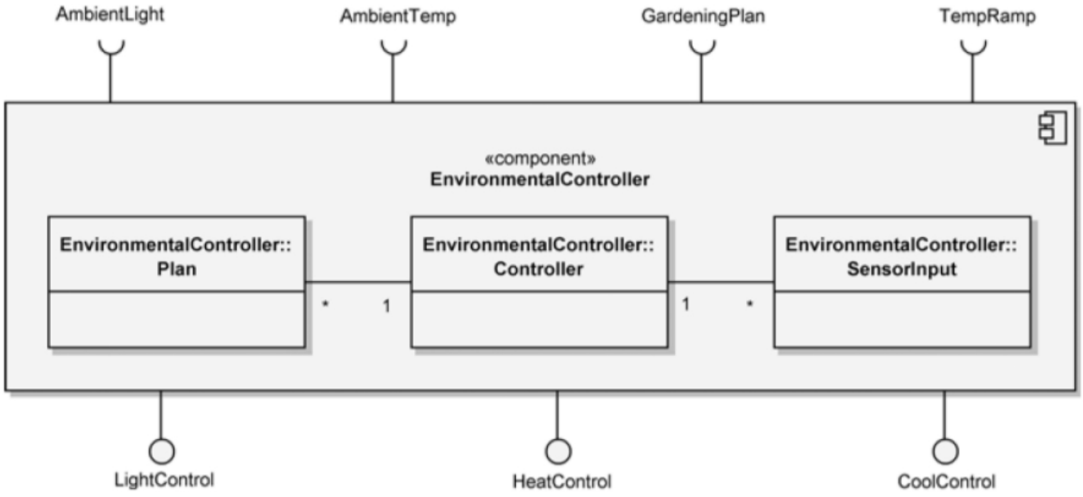


图5-15　EnvironmentalController实现的包容表示方式

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EnvironmentalControlSystem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»，因为它由四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职责，以及哪个内部组件需要组件要求的接口提供的服务[89, 90]。

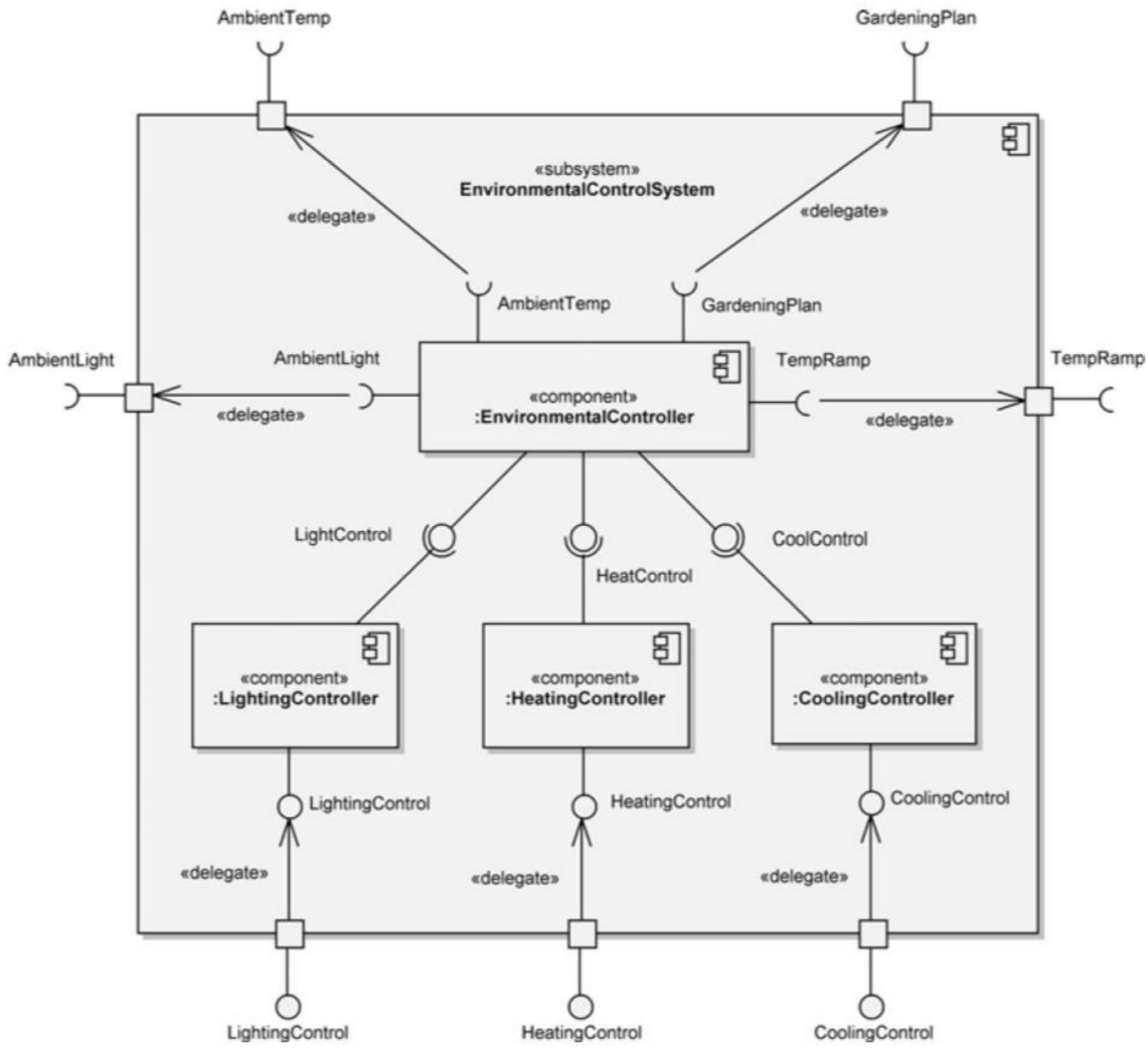


图5-16　EnvironmentalControlSystem的内部结构

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”的职责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、加热和冷却）。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与环境沟通了它的职责（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。

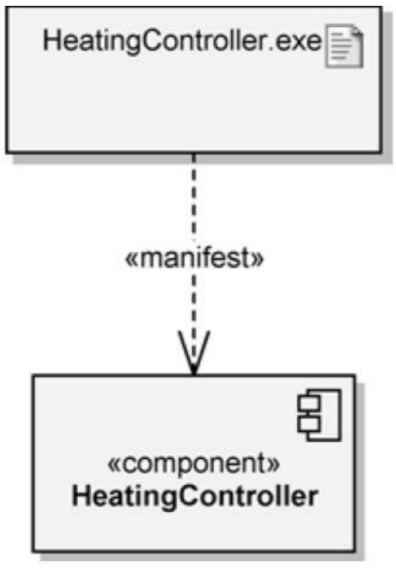


图5-17　HeatingController.exe的工件表示法

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]。

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。

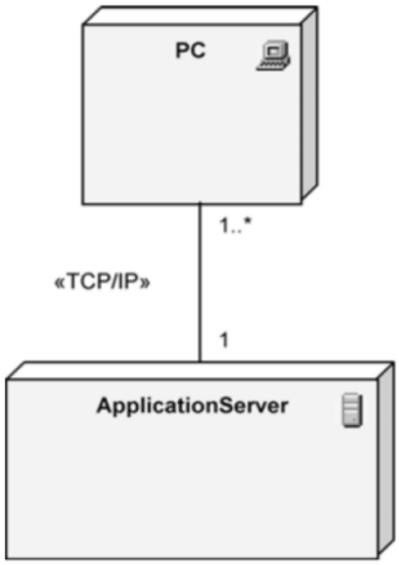


图5-18　两个节点的表示法

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 Thermometer）。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]。

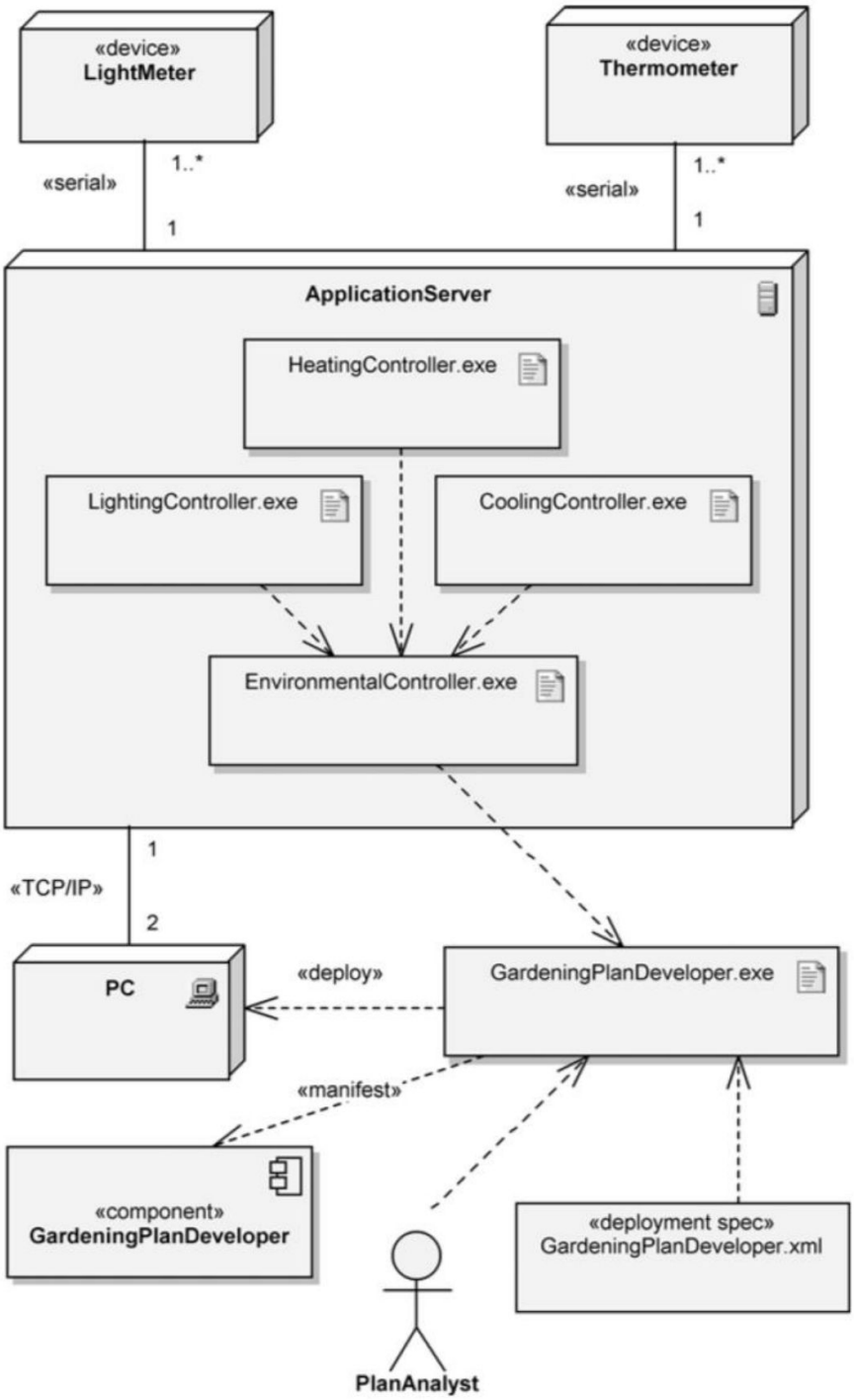


图5-19　EnvironmentalControlSystem的部署图

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，不知道他们为什么要构建这个系统。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、一位父亲、一位艺术家等。

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。

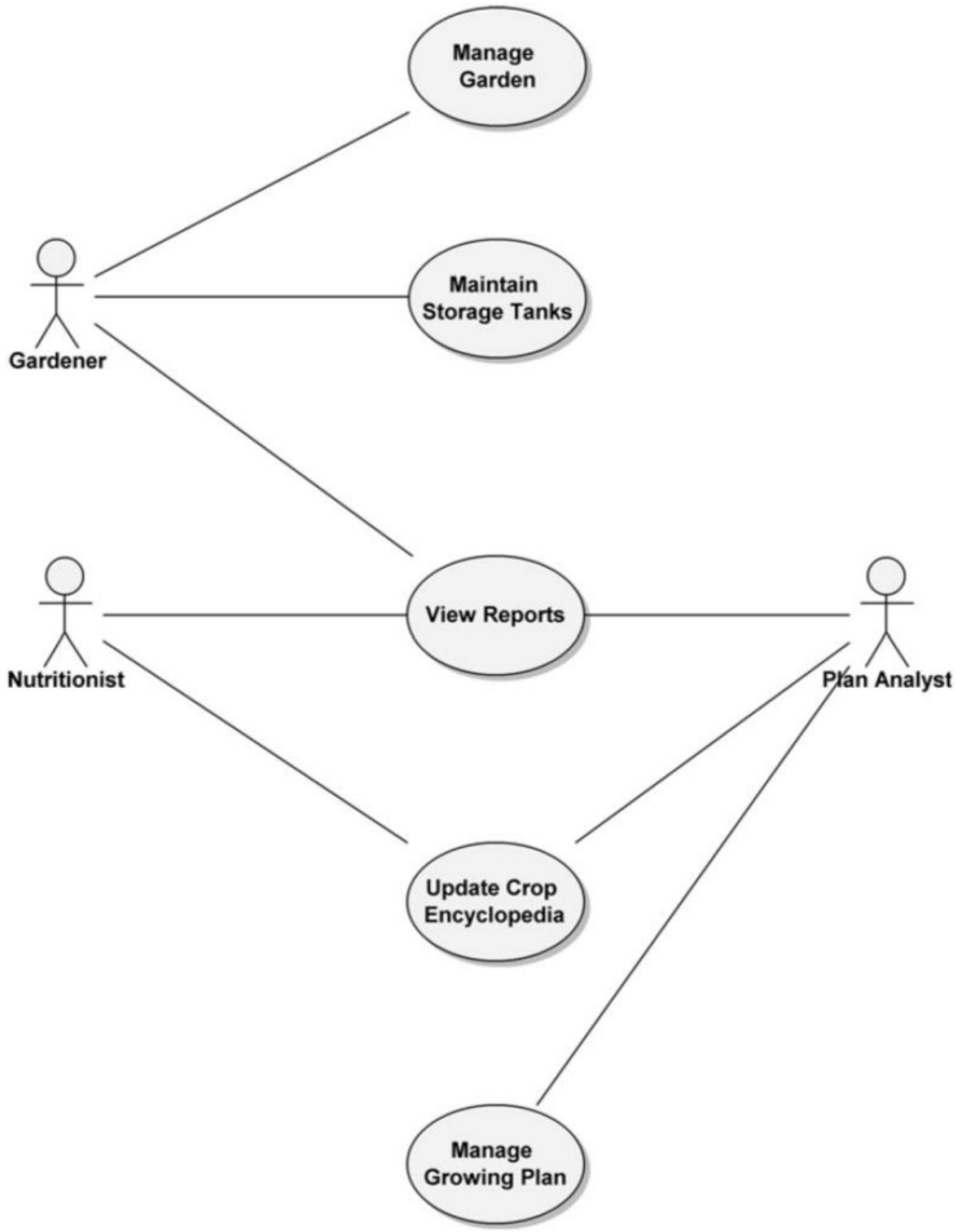


图5-22　用例图

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：

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．水箱正常的溶液种植园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。

- 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。

- 水箱达到操作温度。

H．执行者继续正常的溶液种植园系统操作。

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加入规格说明中，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，在我们的溶液种植园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。

总的来说，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

在溶液种植园系统的例子中，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。

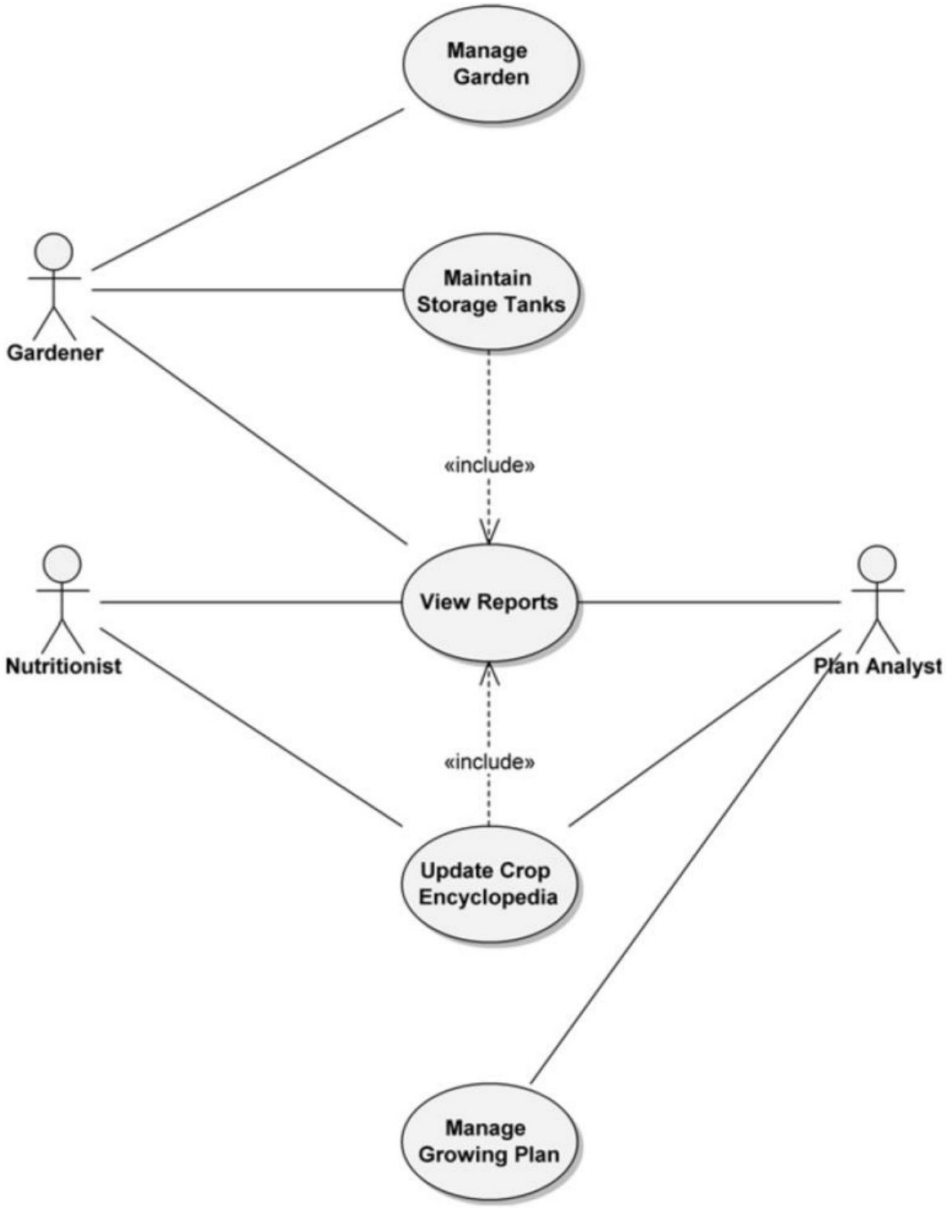


图5-23　显示«include»关系的用例图

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。

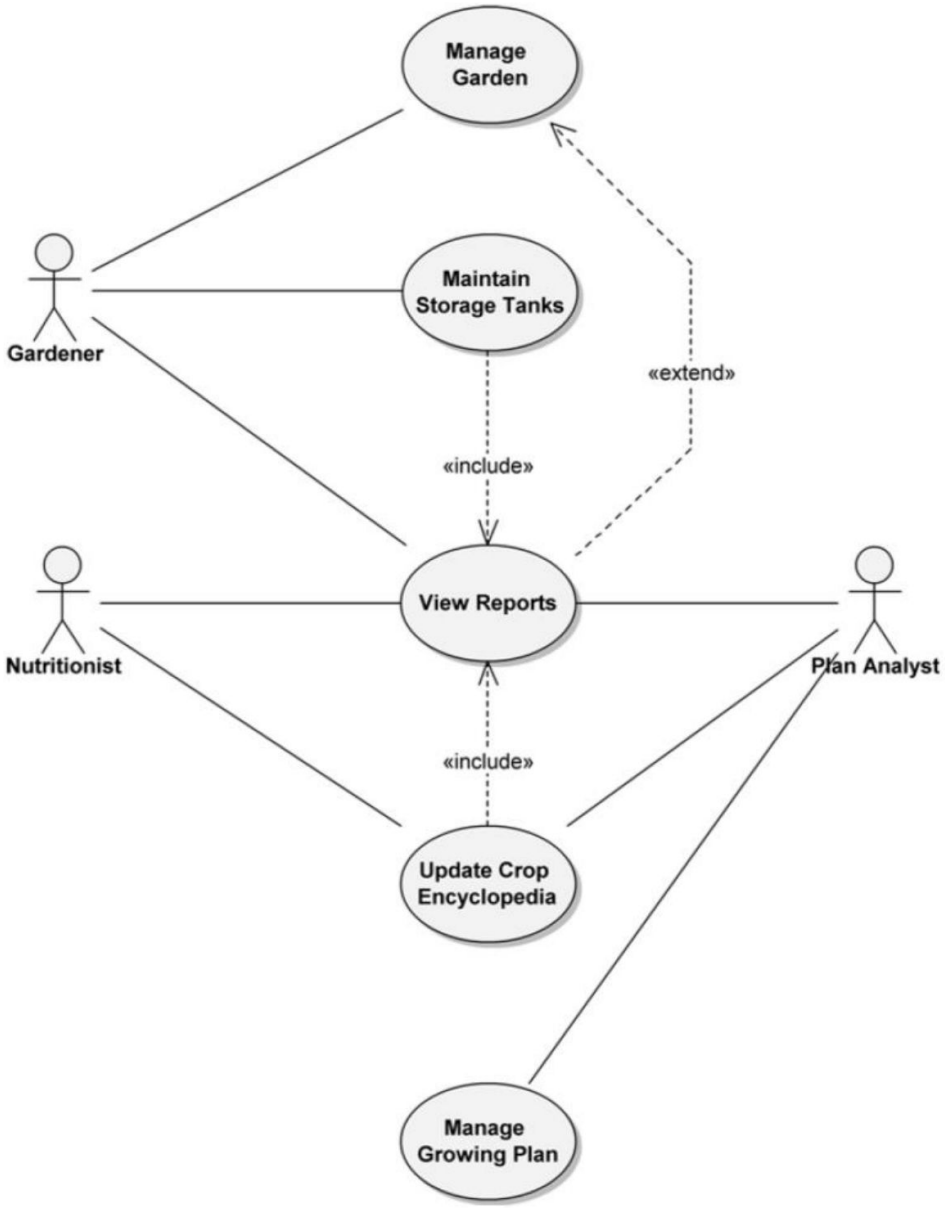


图5-24　显示«extend»关系的用例图

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。

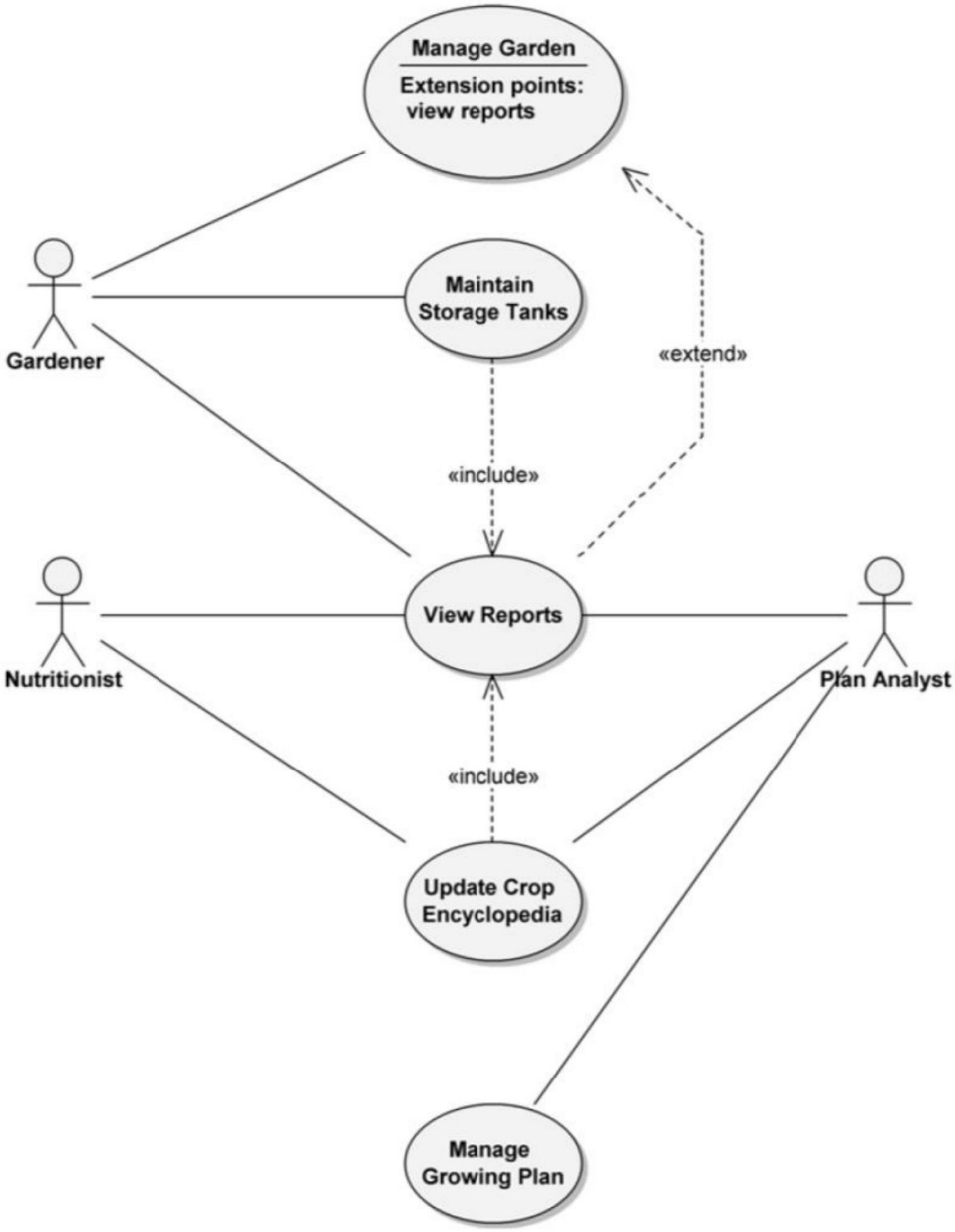


图5-25　显示一个扩展点的用例图

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进行了准确的总结[17]，如表5-1所示。

表5-1　«include»和«extend»关系之间的关键区别[a]

	«include»用例	«extend»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。这样做很快就会违反上一段中我们提到的用例原则，丧失前四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细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。

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。

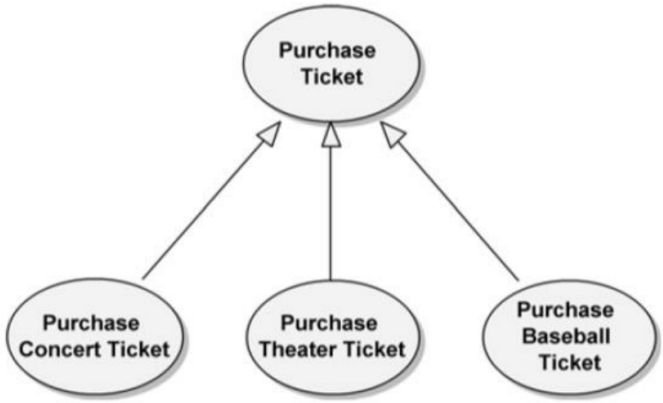


图5-26　用例泛化

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：它们容易创建和修改（不需要复杂的工具），它们可以在任何地方创建（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出这些问题。

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，他们是过程的实现者。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展示了溶液种植园系统案例中执行的一个动作。

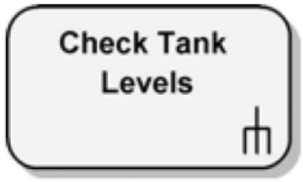


图5-27　动作的表示法

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，它由一个动作构成，即Check Tank Level。

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（且只有一个）离开流程。离开流程通常有一些约束条件，以决定选择哪一条离开的路径。图5-29展示了约束条件[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。

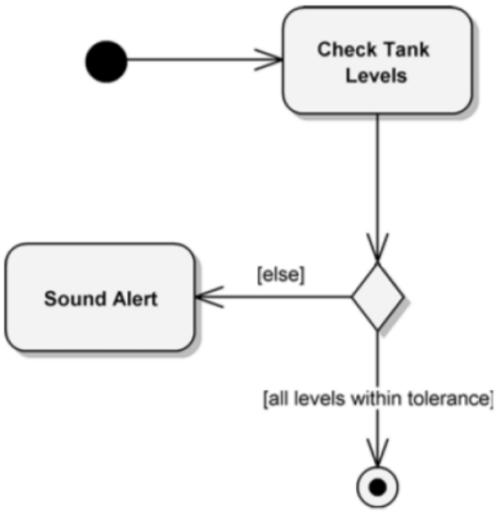


图5-29　判断节点

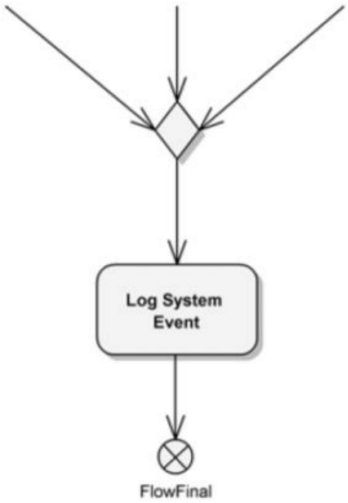


图5-30　合并节点和流程终止节点

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职责。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展示了溶液种植园系统中，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，即第一条粗横线。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。

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，第二条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。

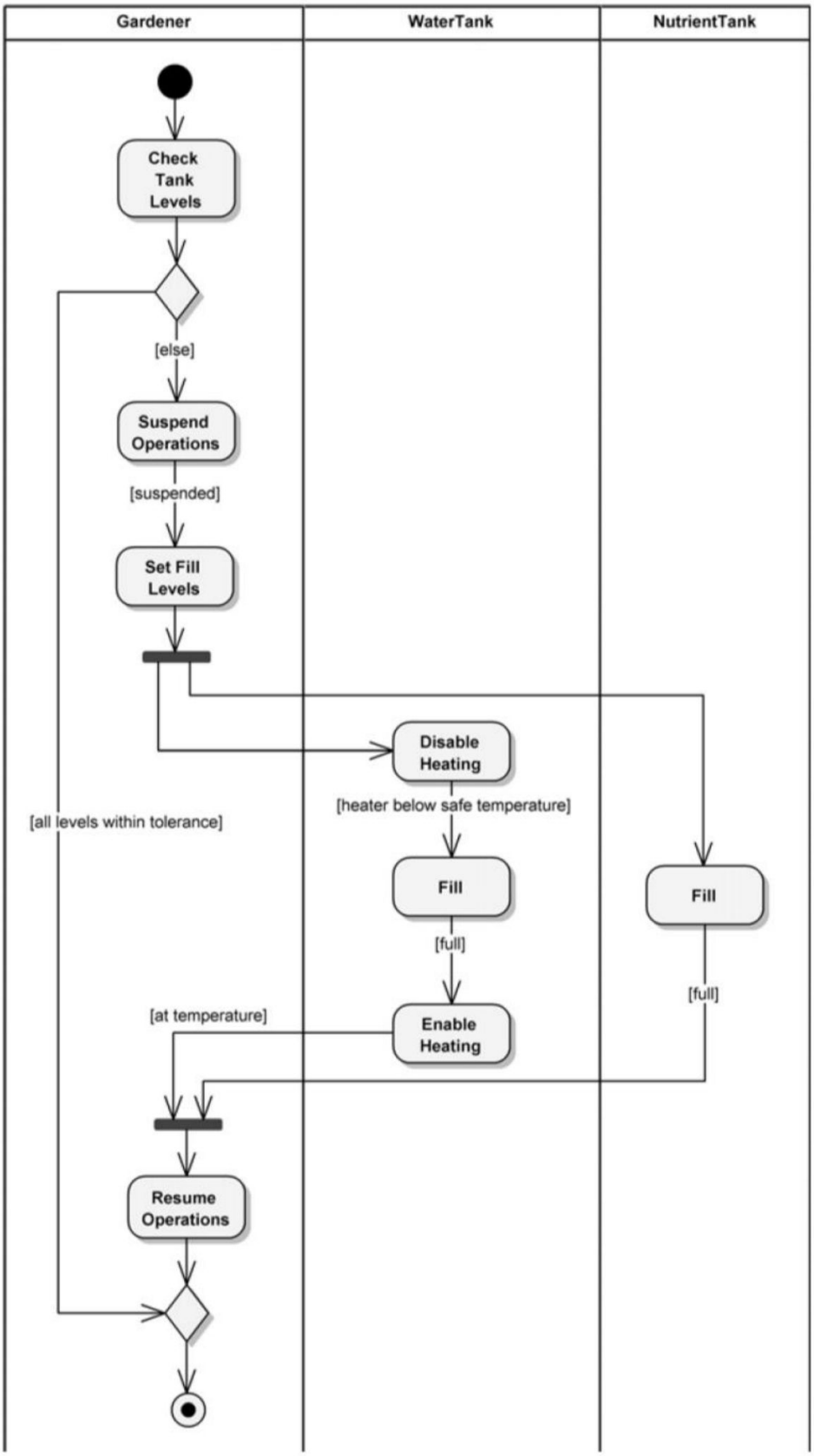


图5-31　带有分区的活动图

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职责，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。

类图中有两个基本元素，即类和它们的基本关系。

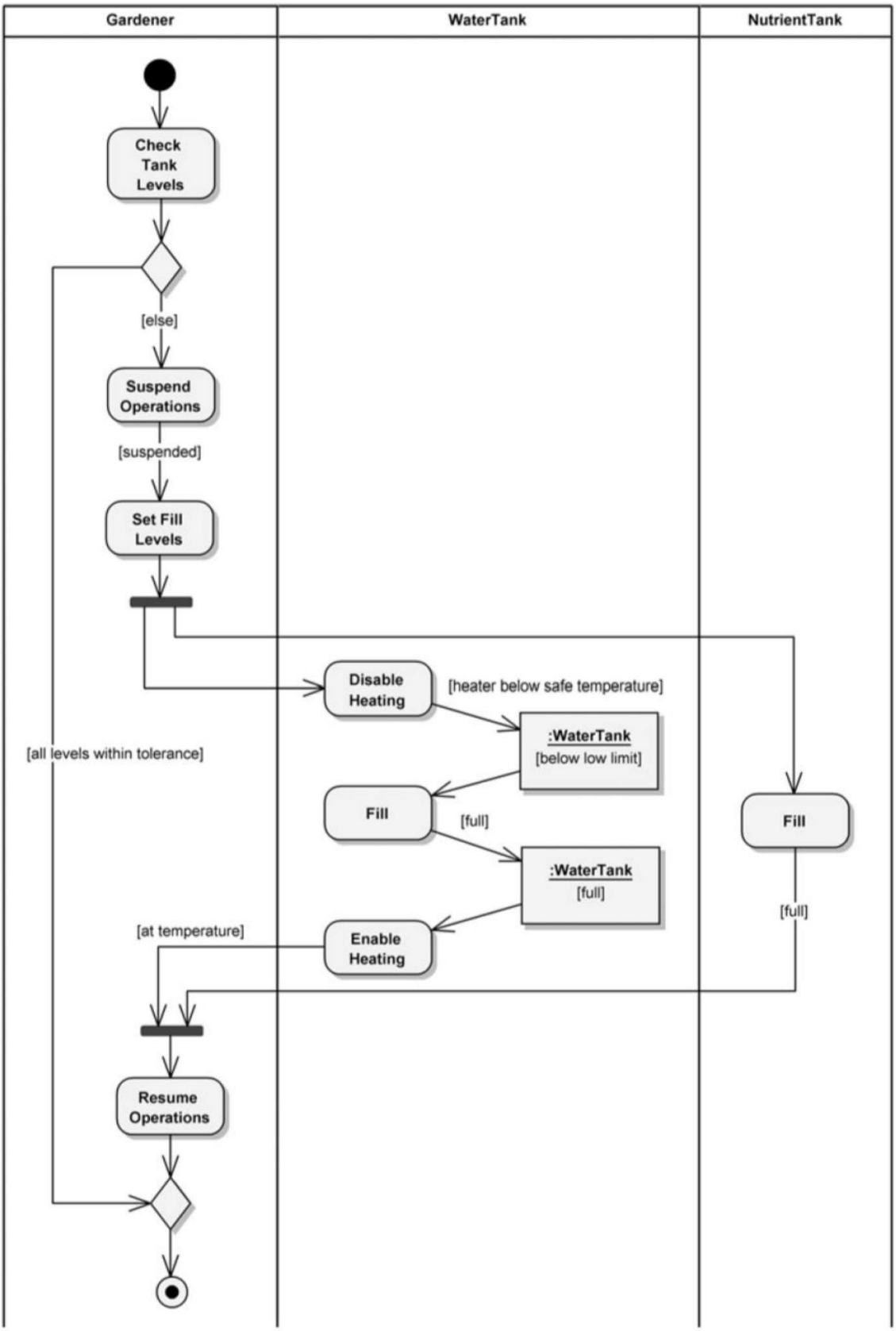


图5-32　带有对象节点的活动图

5.7.1　基本概念：类表示法

如图5-33所示为类图中表示一个类的图标，这是取自溶液种植园系统的一个例子。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]。

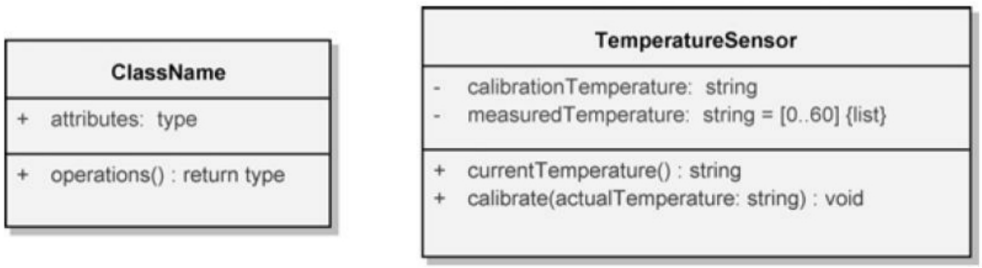


图5-33　一般类图标和种植园系统的例子

■　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]，这表示了一个包含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，就是用斜体来显示类名，表明只能为它的子类创建实例。类似的，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。在溶液种植园系统中，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。

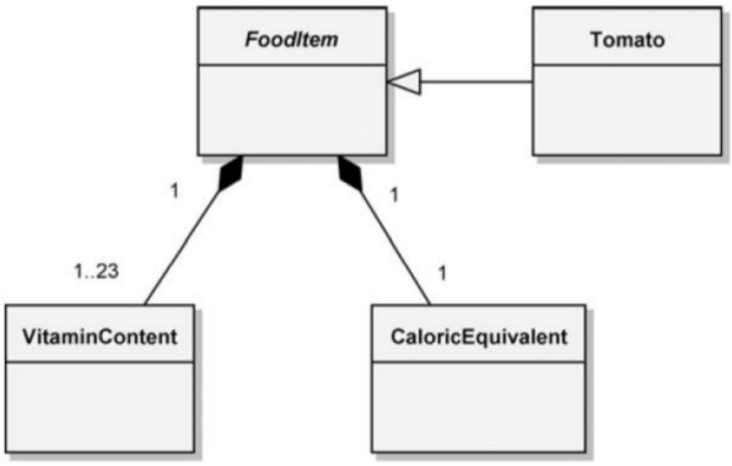


图5-34　抽象类表示

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细化。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，常常会将关联细化为泛化、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。

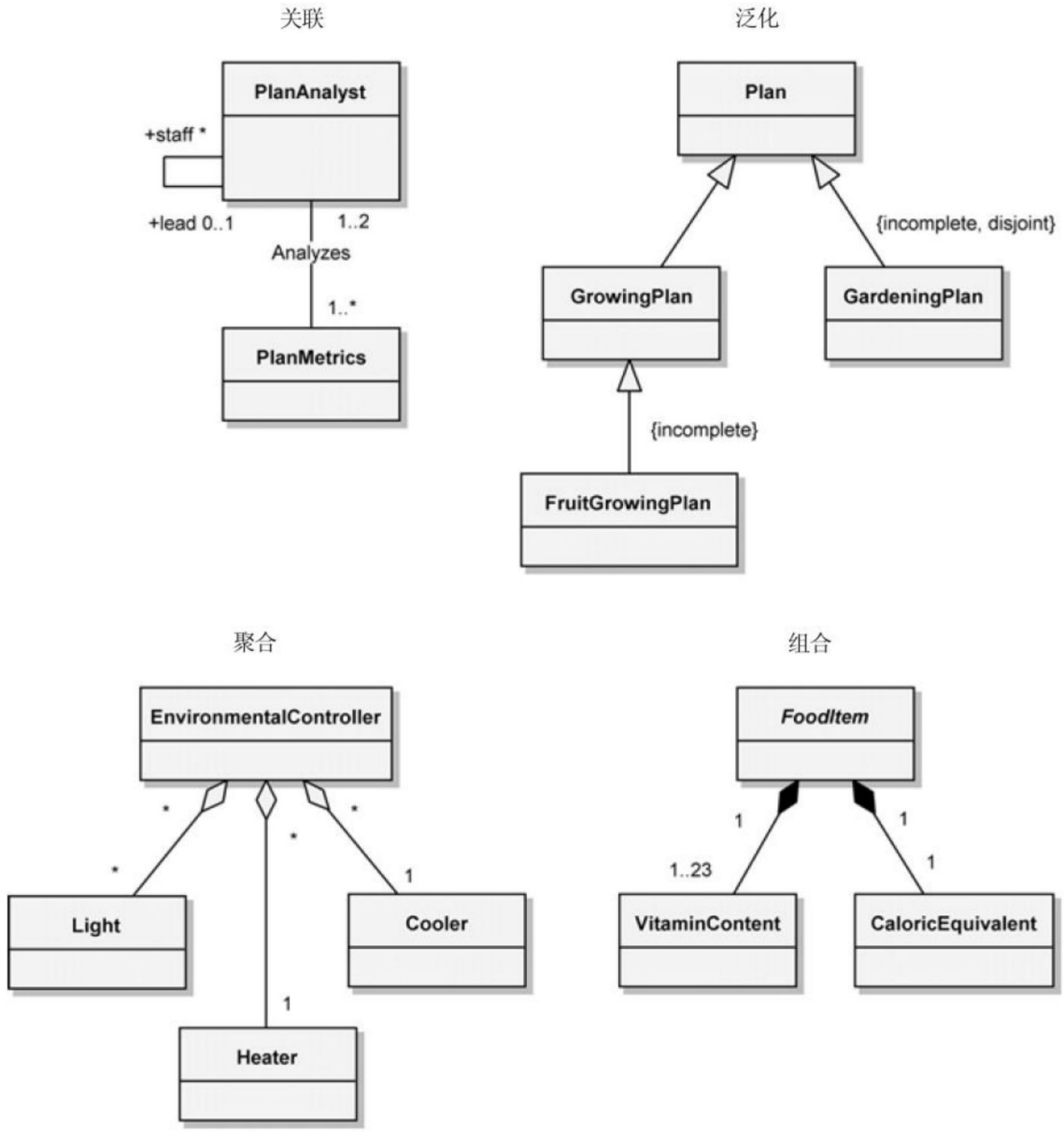


图5-35　类关系图标

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。从图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。

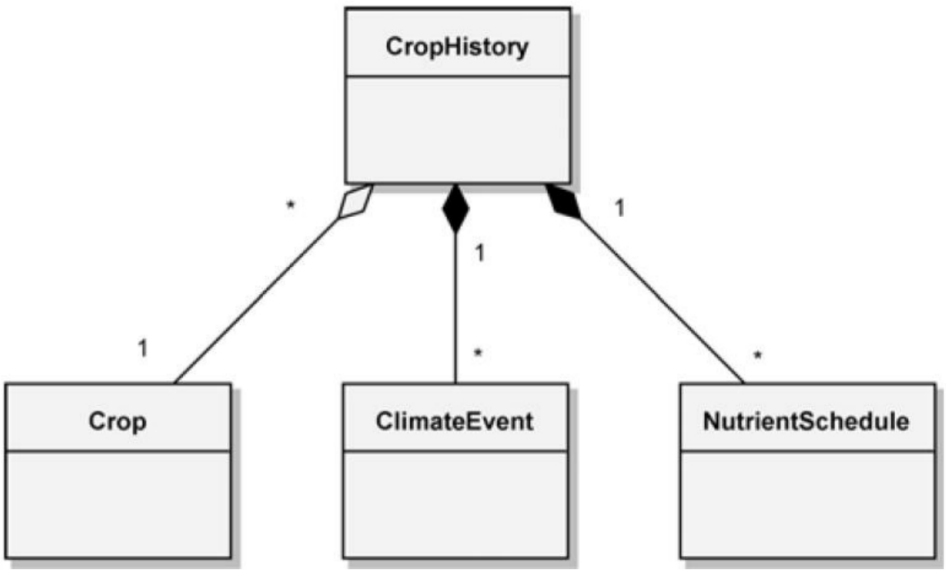


图5-36　物理包容

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。

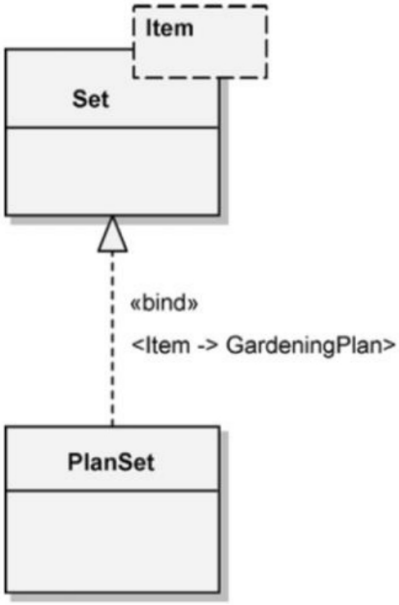


图5-37　模板类和它绑定的类

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，说明关联的可见性，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。

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）。

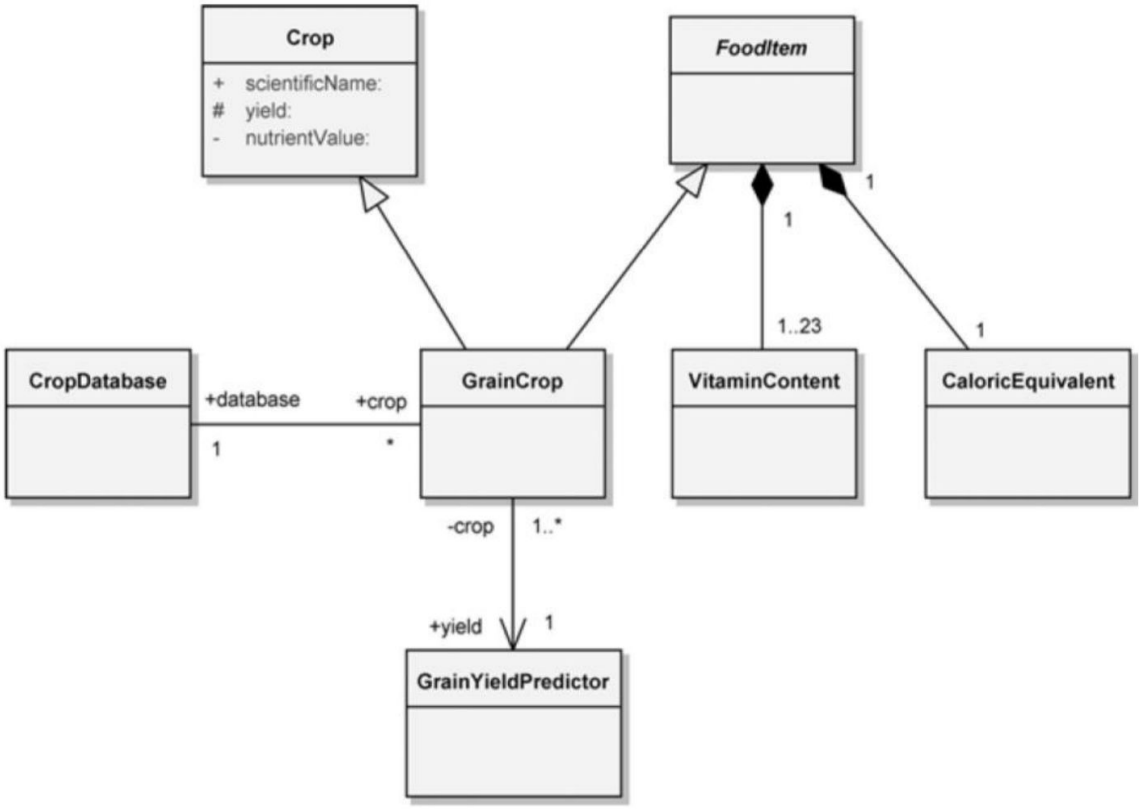


图5-38　类可见性

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。

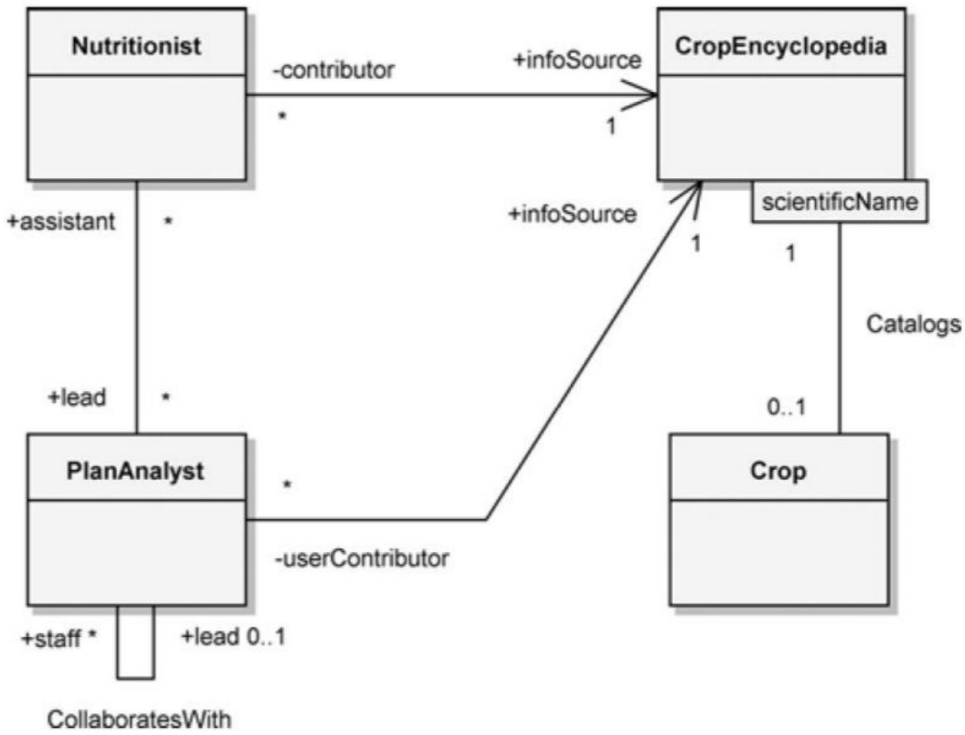


图5-39　关联端名称和限定符

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时是适用的，即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Rumbaugh、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。
- Incomplete：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。

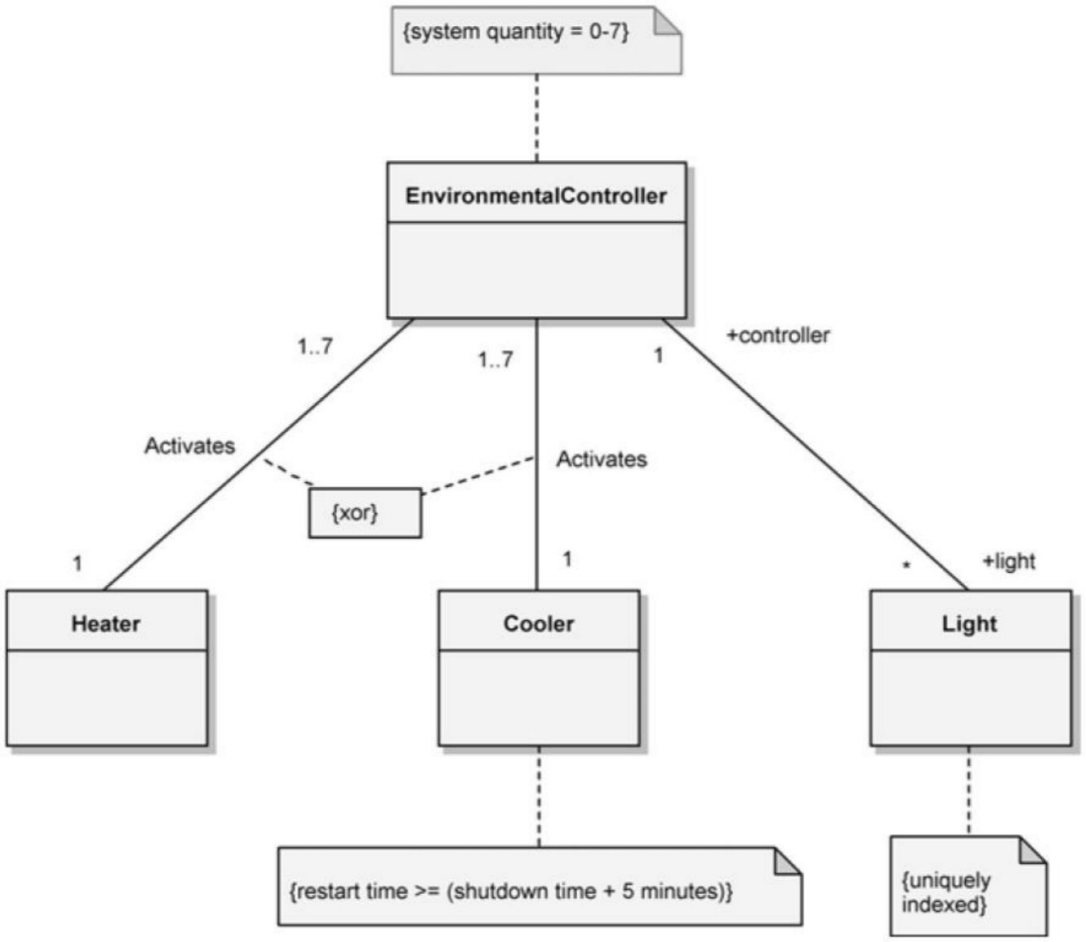


图5-40　约束

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Environmental- Controller类和Light类之间的关联上，要求在这个关联中每个灯必须有唯一编号。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。

类关系的细化

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细化。具体来说，当我们来看聚合时，图5-35展示的Environmental-Controller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，我们会把这些关系细化为聚合。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。

这种带属性的关联具有一般性。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。

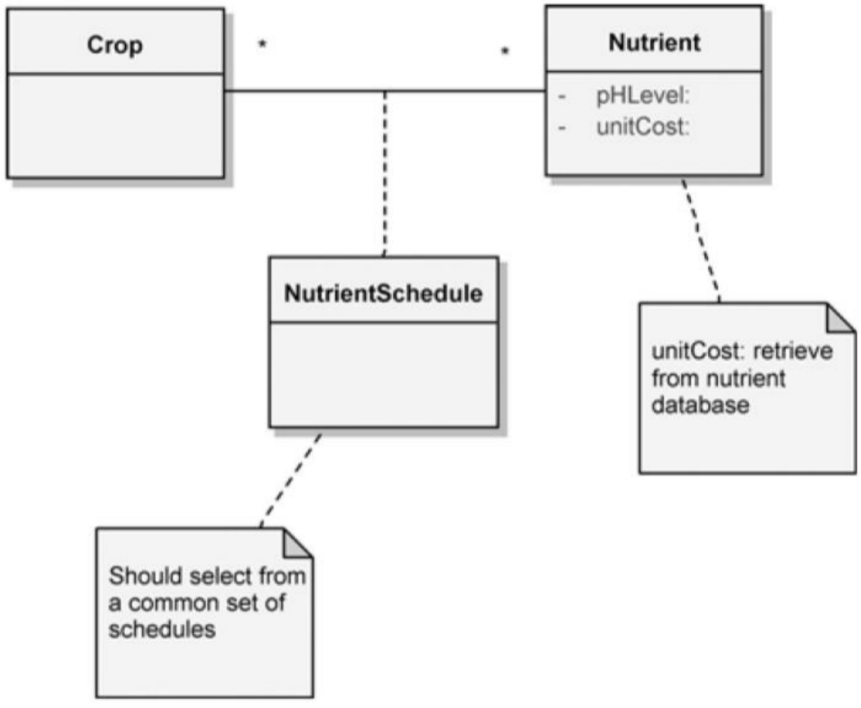


图5-41　关联类和注解

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。

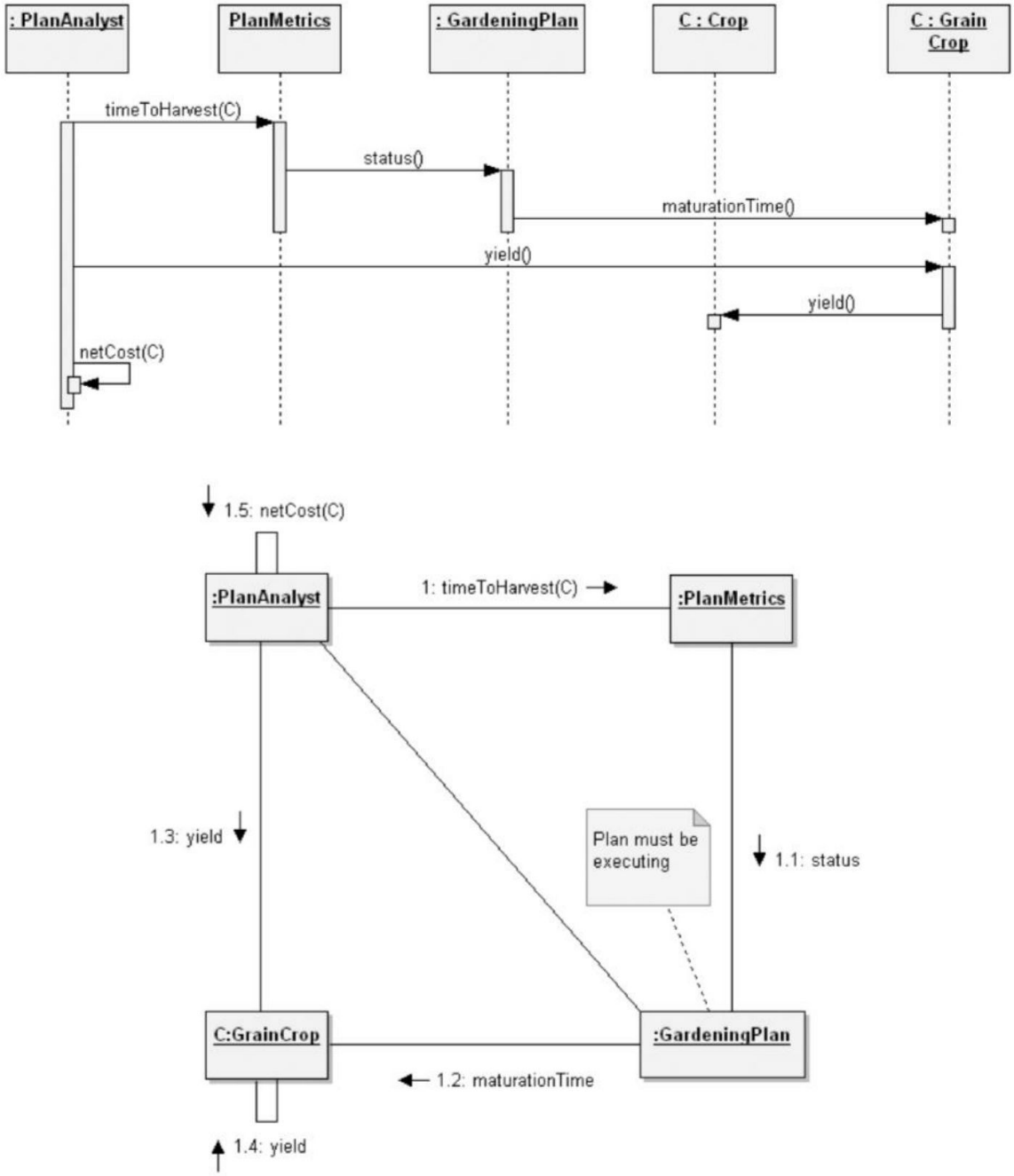


图5-42　序列图（顶部）及其相关的通信图（底部）

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。

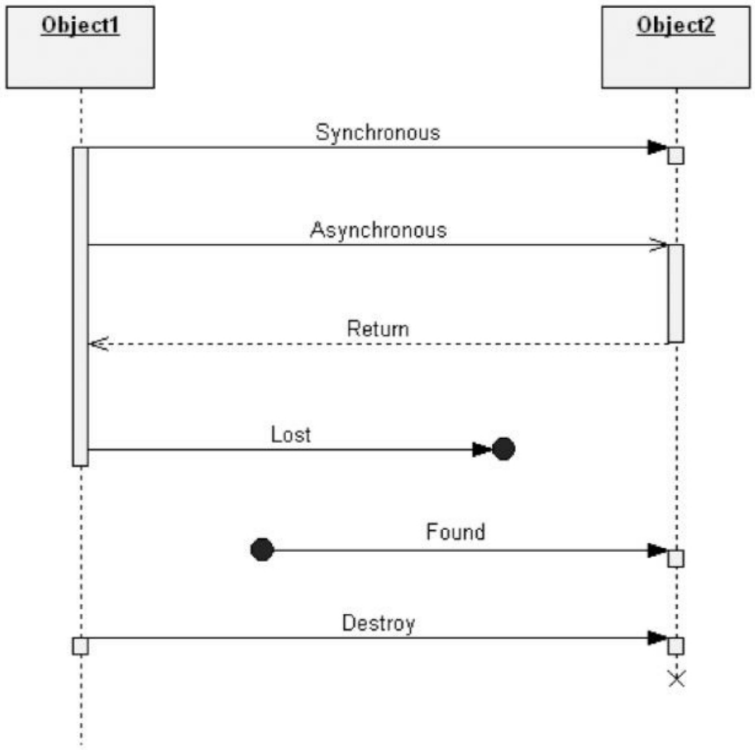


图5-43　消息类型的表示法

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。

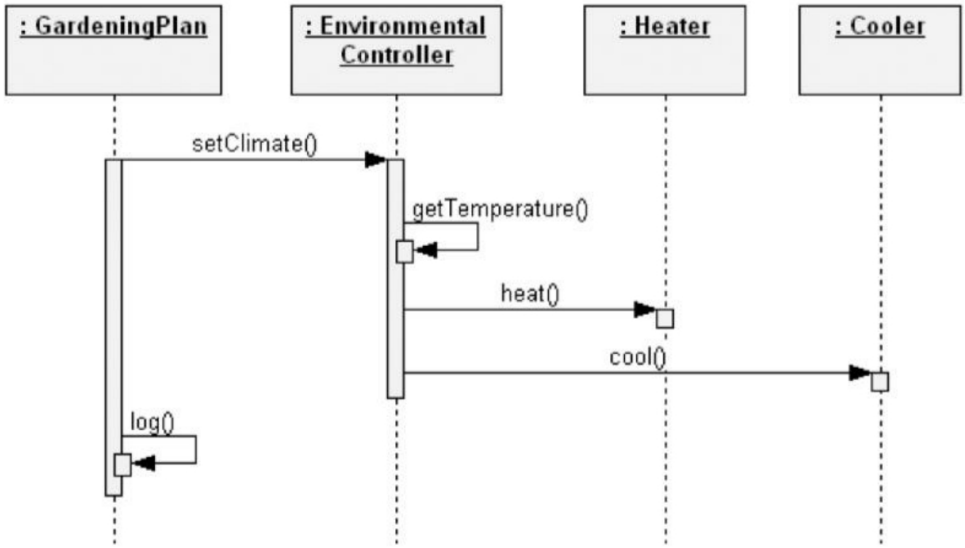


图5-44　执行说明

脚本

虽然不是UML 2.0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。

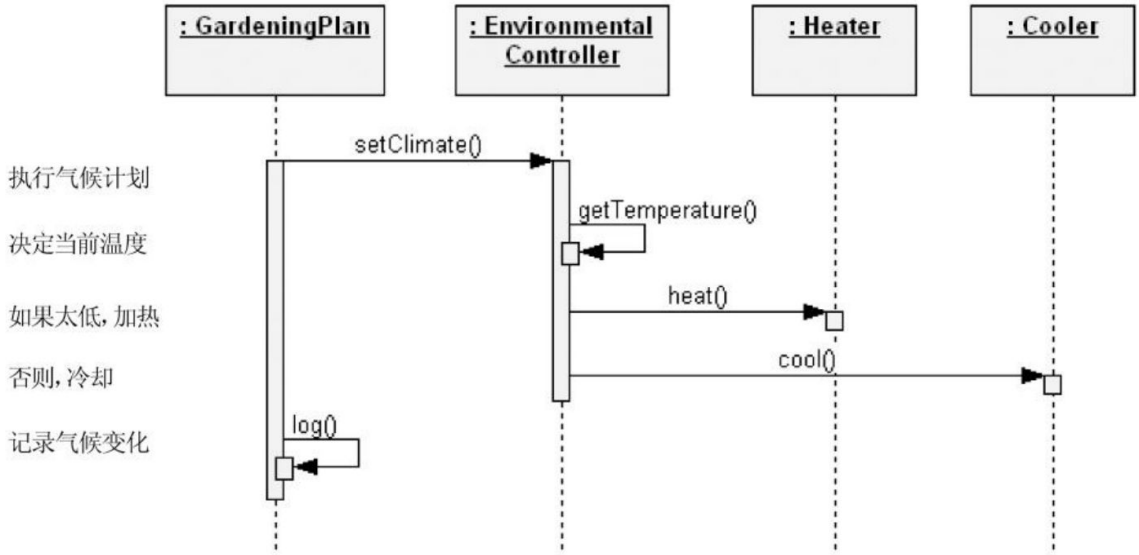


图5-45　脚本和序列图

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。

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（即复制了这个序列）。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。

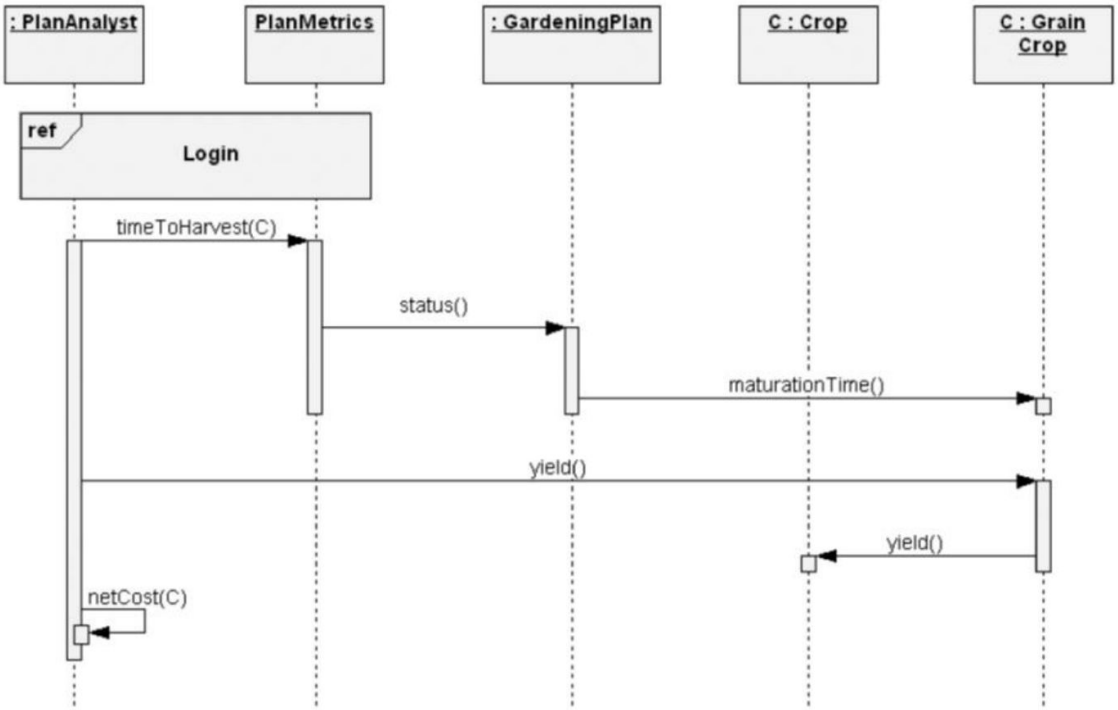


图5-46　一种交互使用——Login

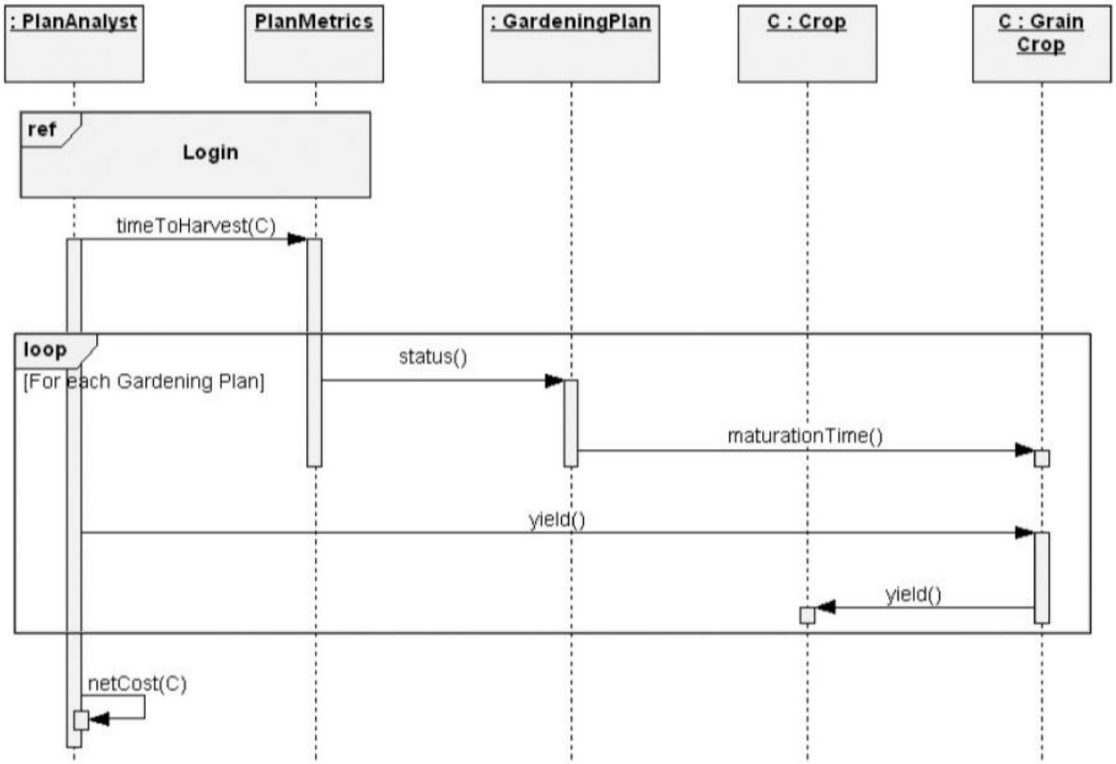


图5-47　一种交互操作符——loop

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，有些是激活的，有些是非激活的（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非激活的计划执行循环，只希望对当前已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。

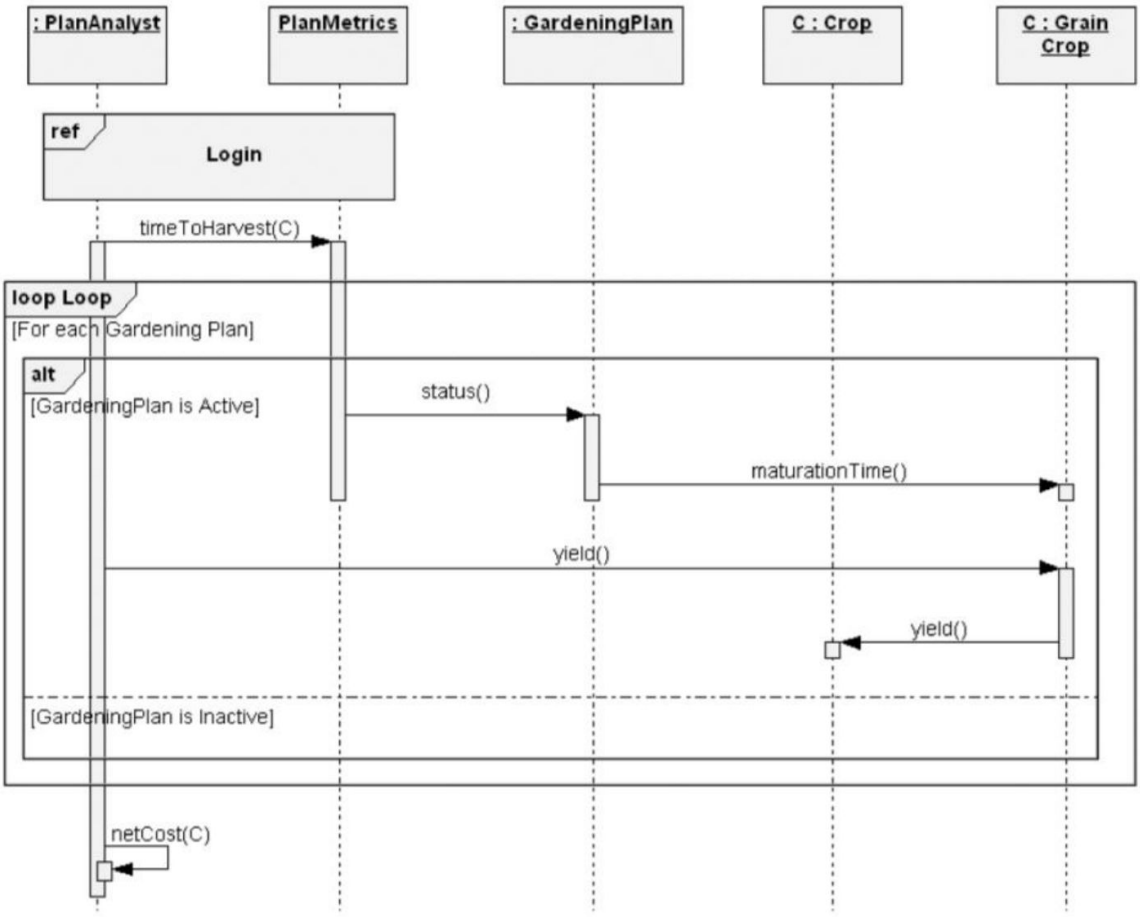


图5-48　一种交互操作符——alt

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按这种方式使用，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 :Environmental- Controller, :Heater, :Cooler”，位于框的左上角。这个名称的意义如下。

- sd：标签，表明这是一张交互图。
- MaintainTemperature：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，根据溶液种植园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。

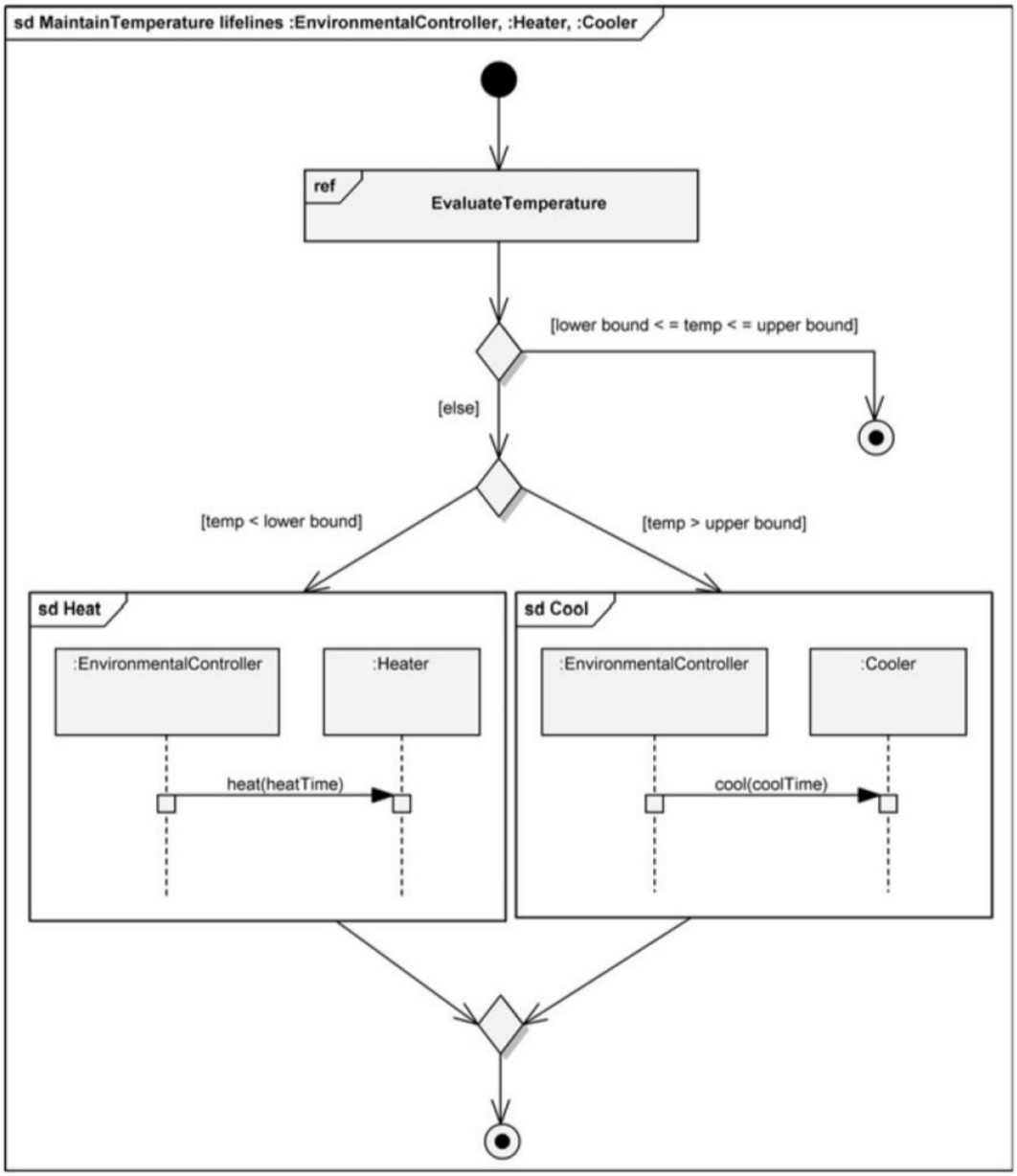


图5-49　MaintainTemperature的交互概述图

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，并对它们在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展示了溶液种植园系统的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。

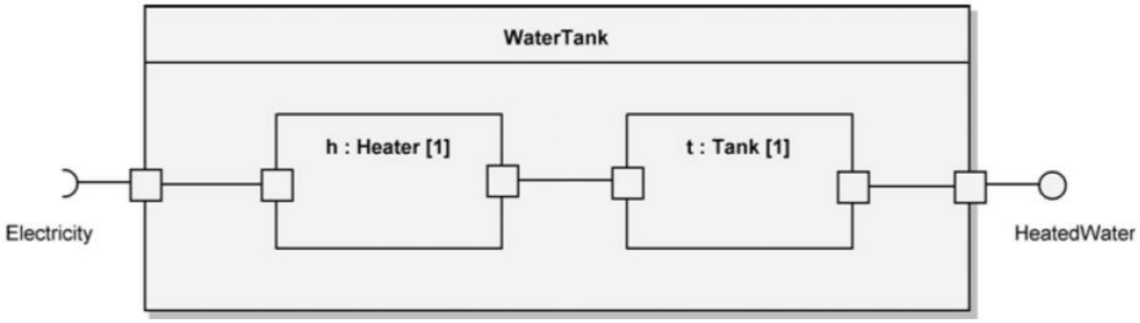


图5-50 WaterTank的组合结构图

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，这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连接起来，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从溶液种植园系统中获取电力，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为组合结构及其环境提供了通信联系，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“接口接连器”）。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协作定义了溶液种植园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]。

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]。

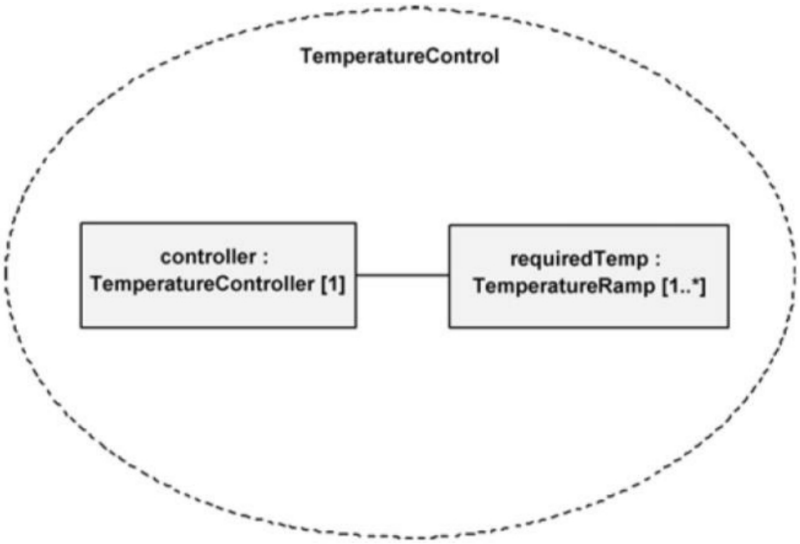


图5-51　TemperatureControl协作

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，这个状态空间包含了不确定的、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展示了溶液种植园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Timing。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”。这就是意味着，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，在溶液种植园系统中，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。

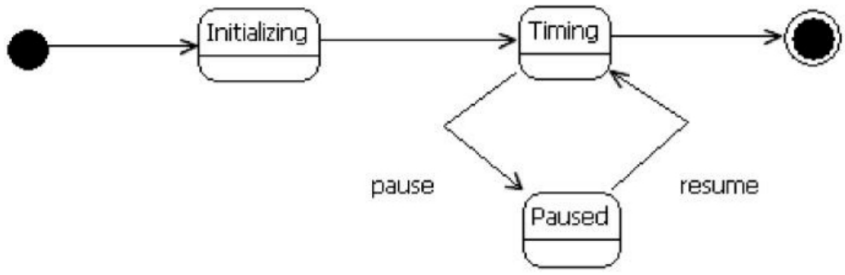


图5-54　间隔计时器的其他状态和转换事件

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）。

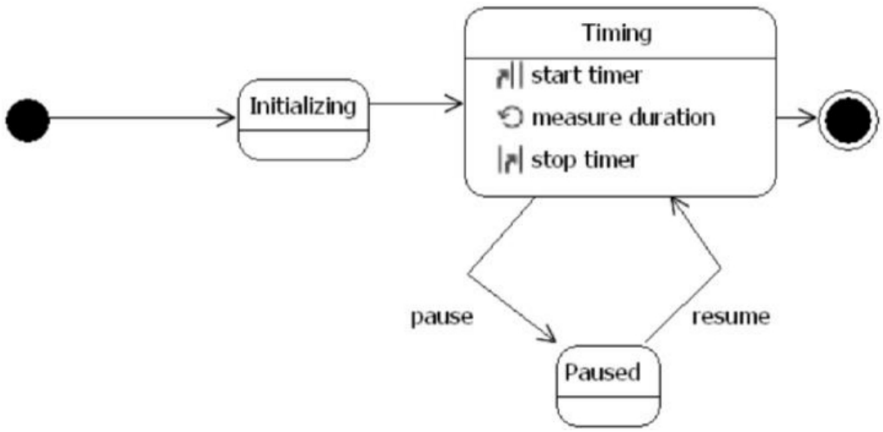


图5-55 入口活动、执行活动和出口活动

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。溶液种植园系统中可能发生的一个重要事件就是制冷器失效。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。

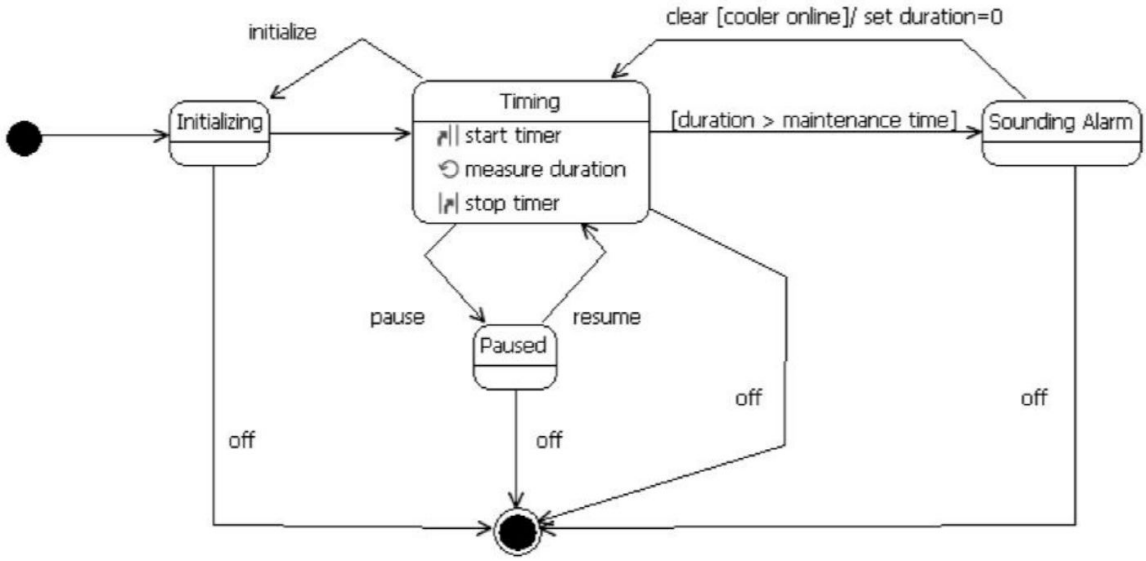


图5-56　改进的间隔计时器状态机图

在图5-56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Sounding 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，但它有一个约束条件，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。如果制冷器没有回到工作状态，这次转换就不会被触发。如果制冷器回到了工作状态，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。在这种情况下，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5-57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合状态，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。

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5-57会变成一个非常容易理解的状态图（参见图5-58）。

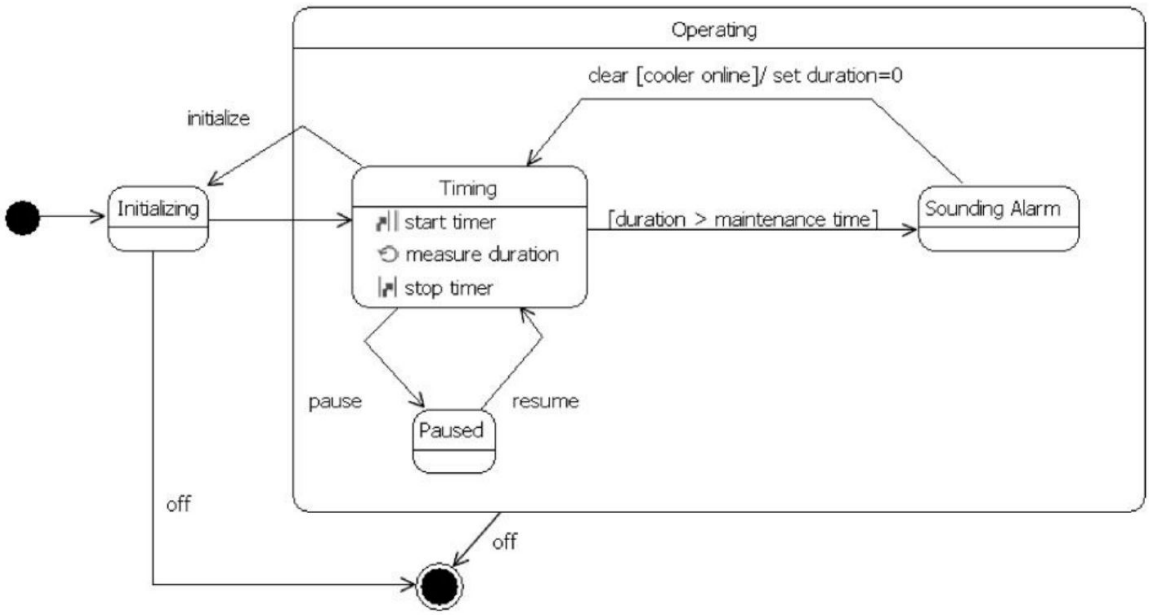


图5-57　复合状态和嵌套状态

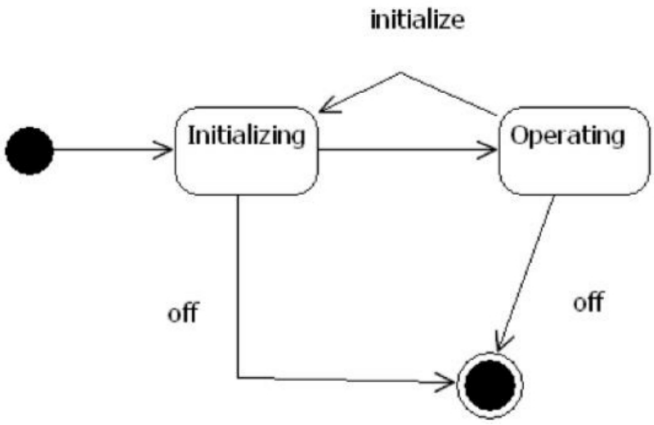


图5-58　间隔计时器的状态机图的高层视图

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。

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外，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Timing Maintenance 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。

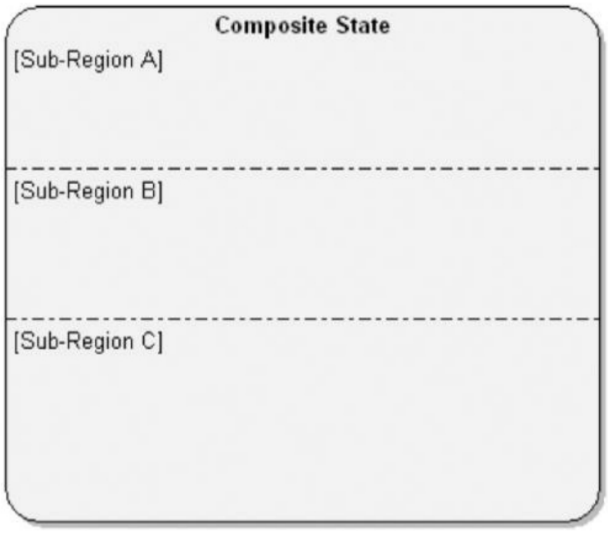


图5-59　状态机图中的并发子区域

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。

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。

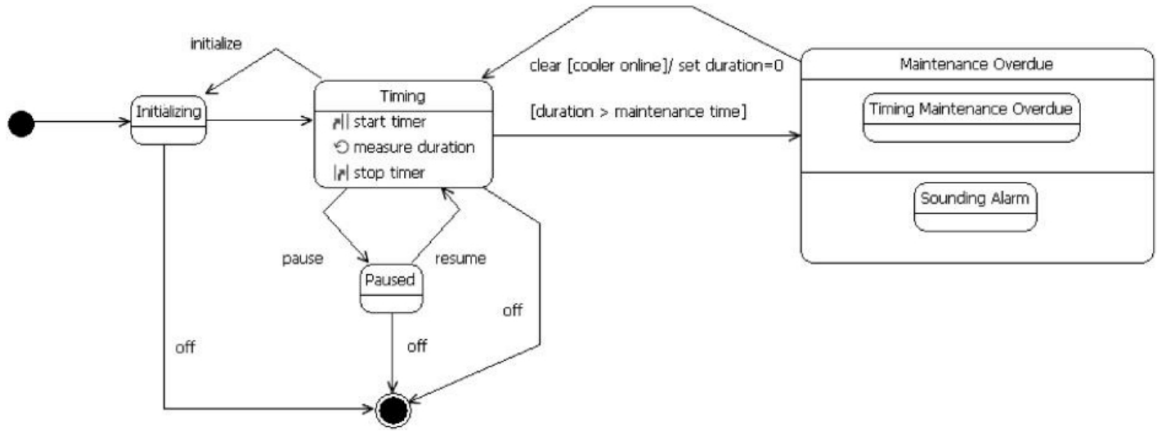


图5-60　带有并发子区域的复合状态Maintenance Overdue

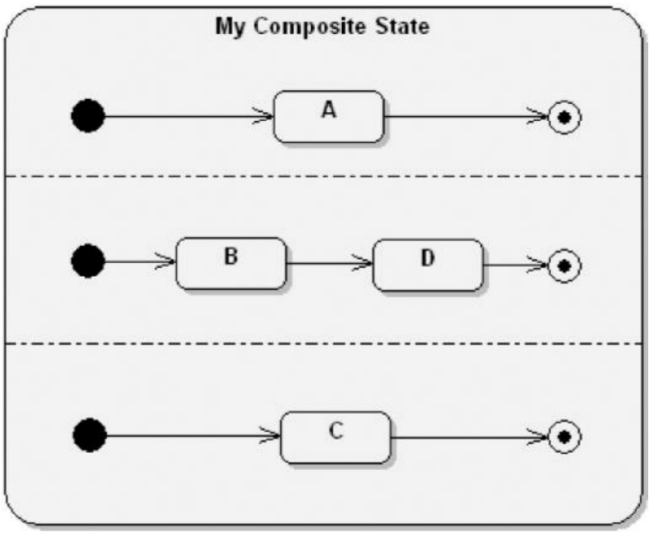


图5-61　带有并发子区域的一般复合状态

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约束条件，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。

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。

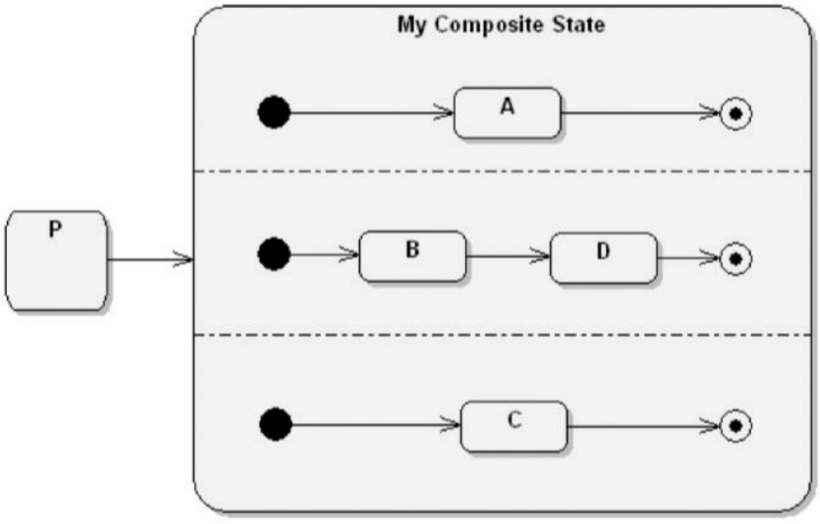


图5-62　转换至一个复合状态

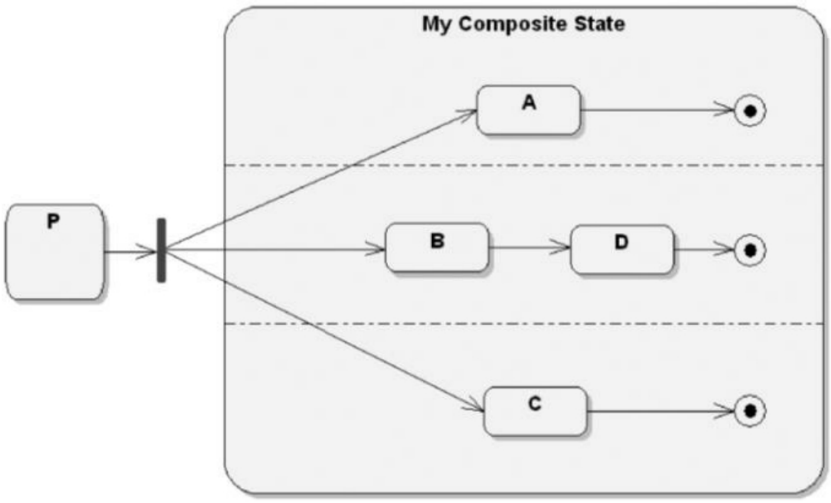


图5-63　分叉至一个复合状态

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约束条件。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。

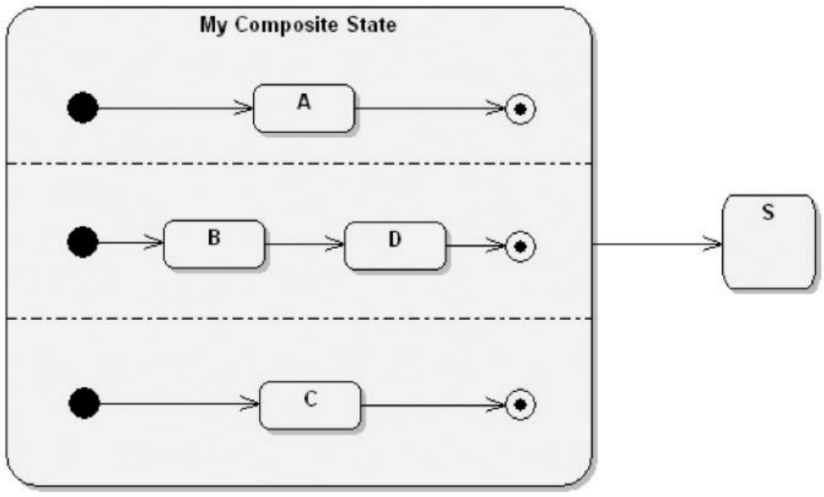


图5-64　离开复合状态的一次完成转换

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。

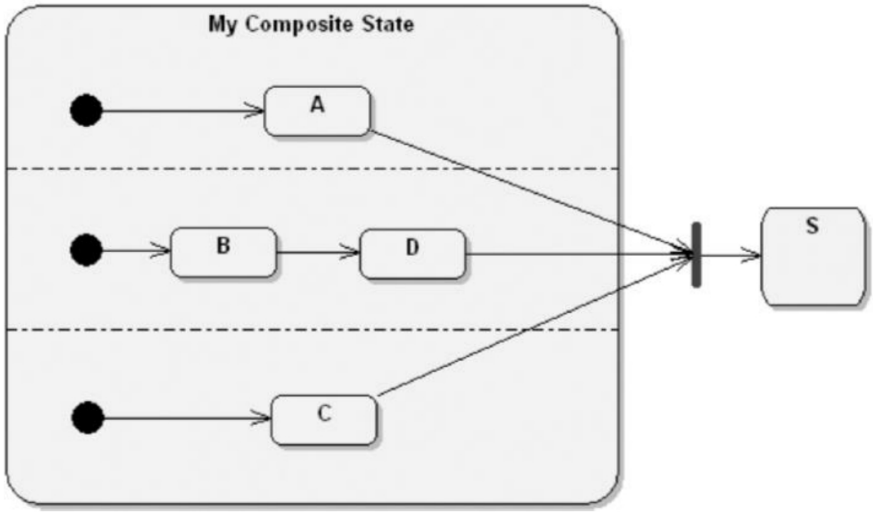


图5-65　离开复合状态的一次结合

沿着这些同样的线，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]。

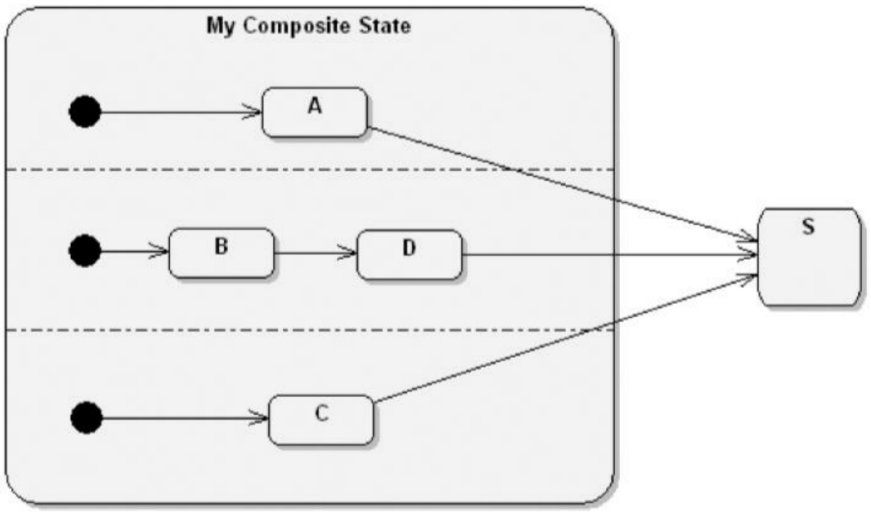


图5-66　离开复合状态的一些独立转换

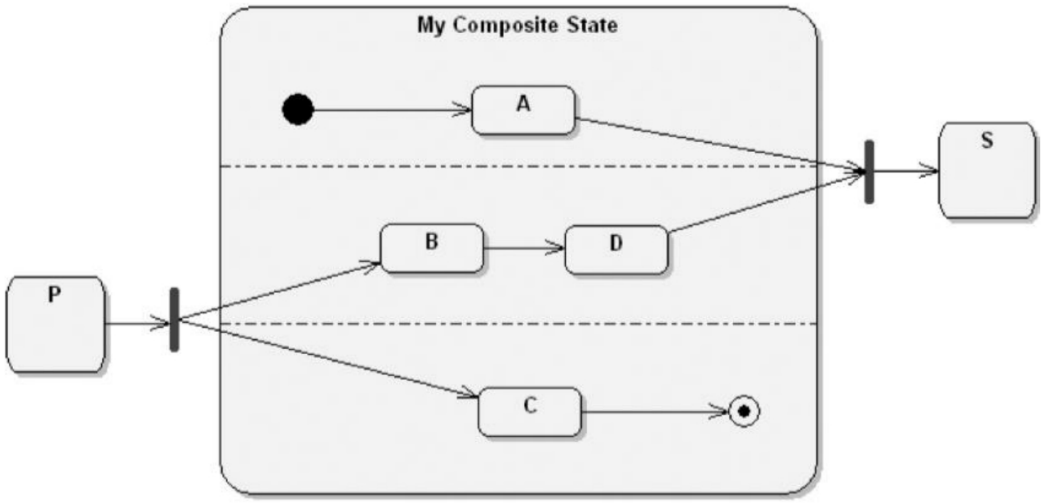


图5-67　部分参与结合和分叉

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。

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。

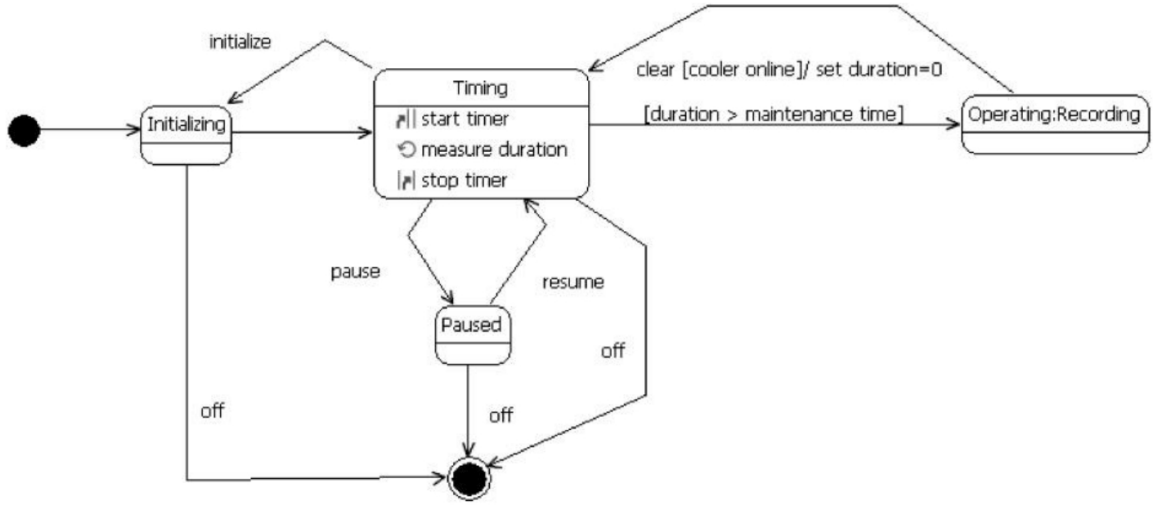


图5-68　在间隔计时器状态机图中使用子状态机状态

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溯与它并列的一个序列图（参见图5-69）。

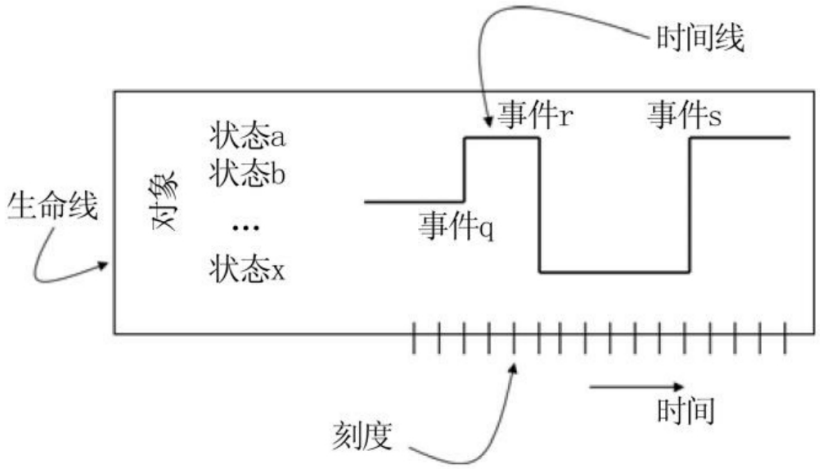


图5-69 一般时间图

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（阀门）对象的一个简单的时间图，在溶液种植园系统中，这个对象受到控制，以注满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？

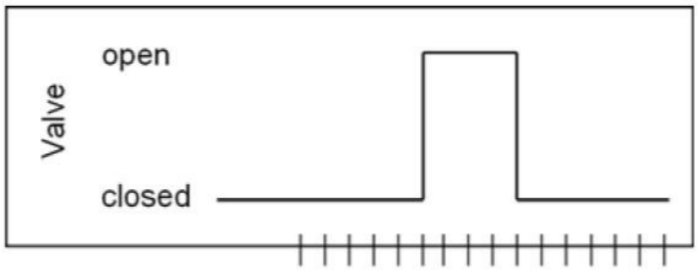


图5-70　Valve对象的时间图

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。

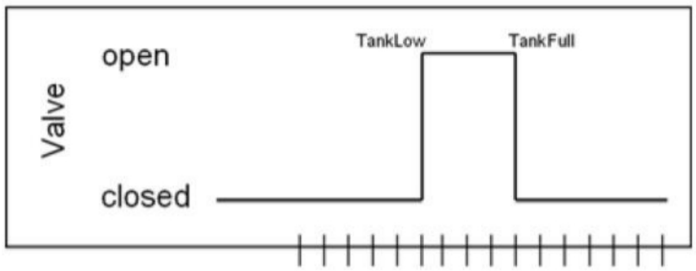


图5-71　包含事件的Valve对象的时间图

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。

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。

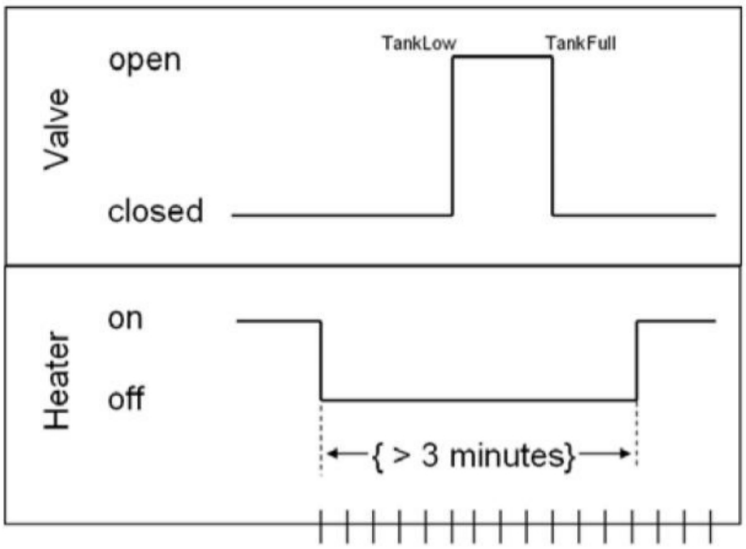


图5-72　包含约束的两个对象的时间图

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。

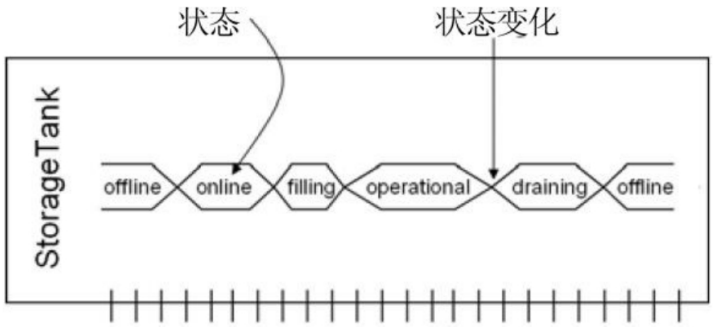


图5-73　在时间图中使用紧凑表现形式代替时间线

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。

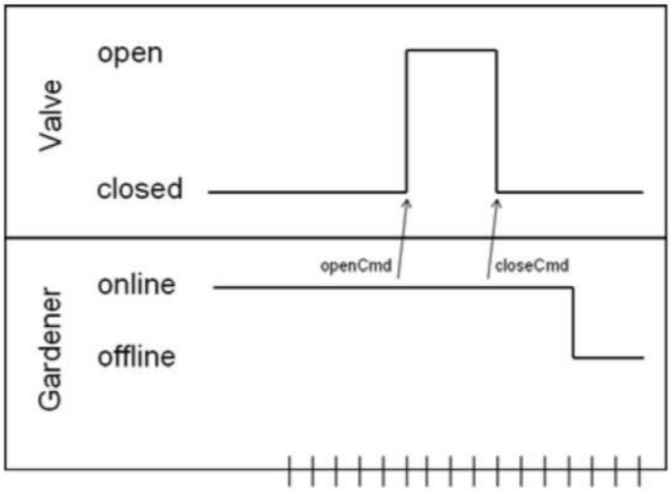


图5-74　在时间图中使用消息代替事件

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。

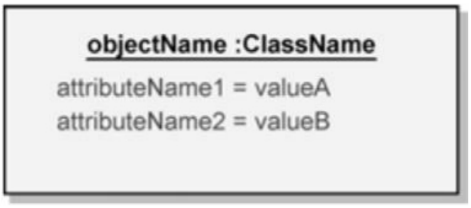


图5-75 一般对象图标

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画线，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。

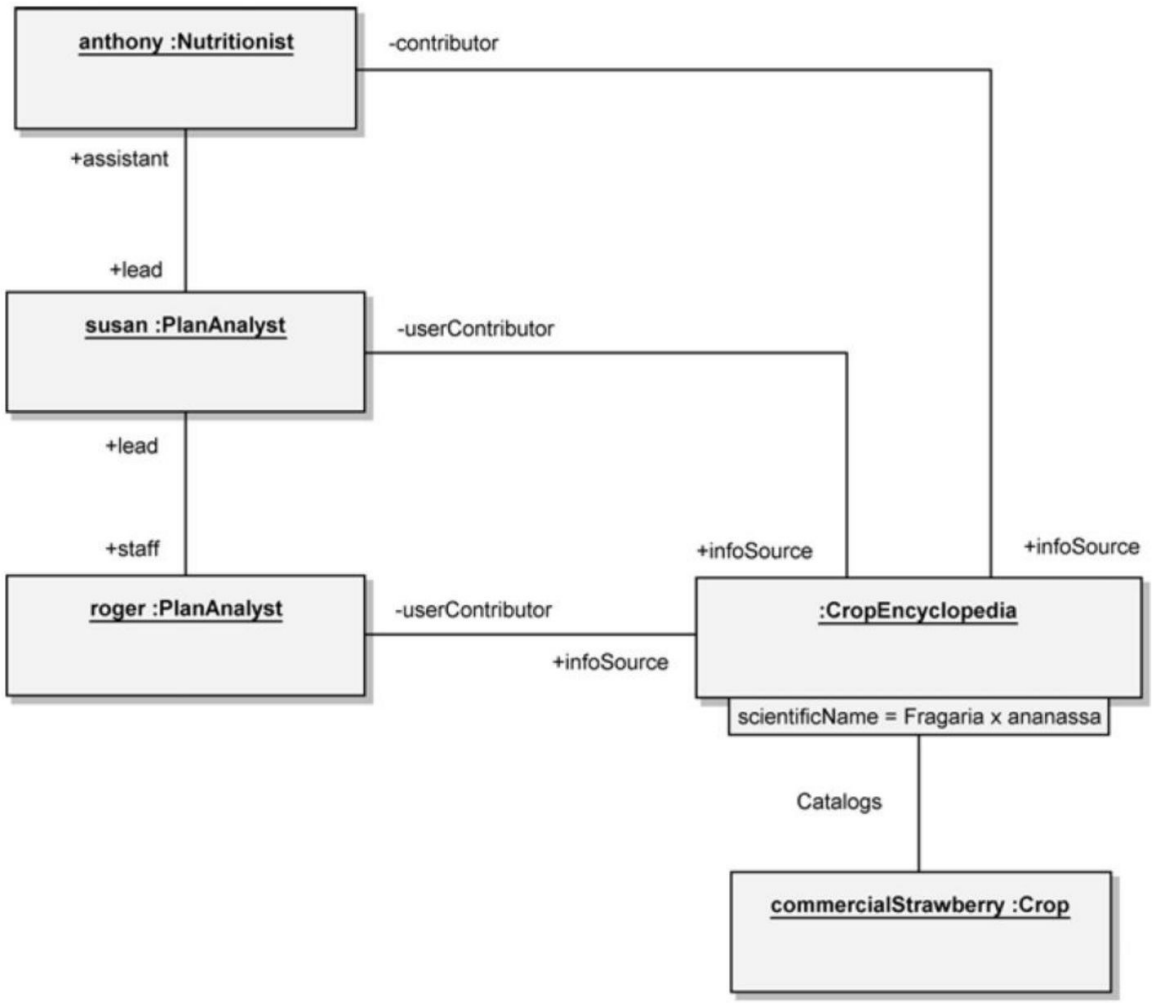


图5-76　对象关系

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。在这里我们看到，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[2]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展示了溶液种植园系统的一张通信图的例子。这张图的意思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。

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。

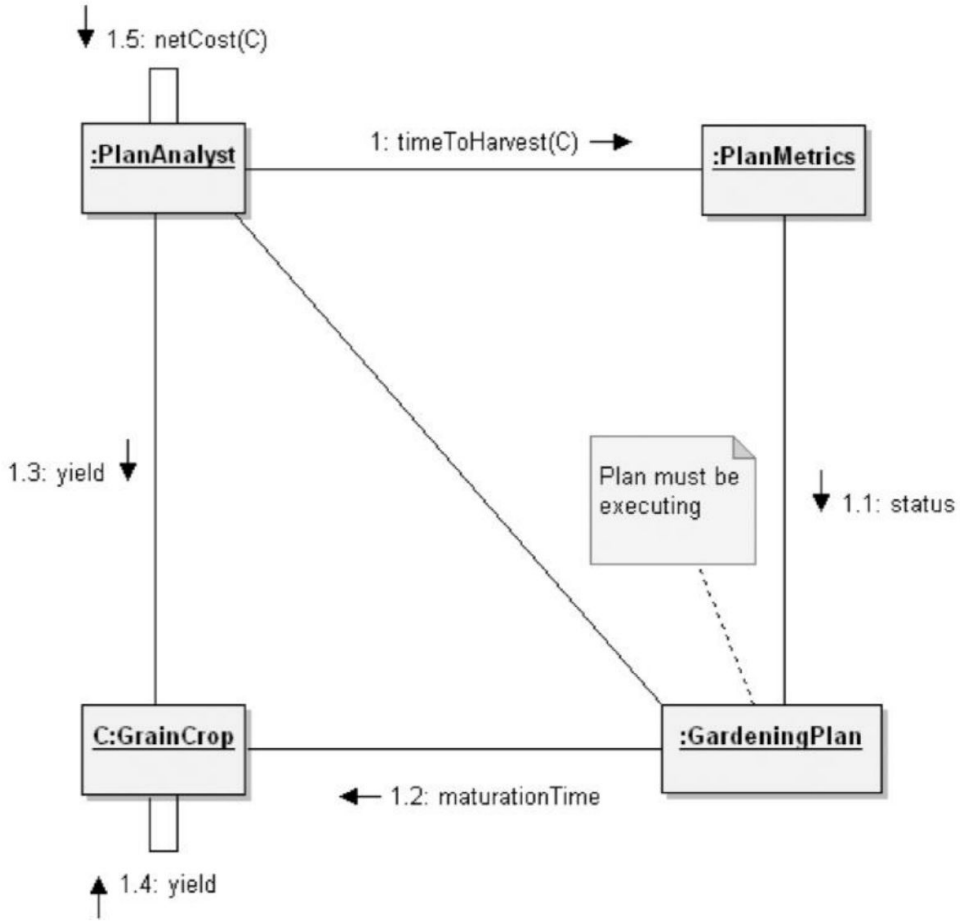


图5-77　溶液种植园系统的通信图

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。

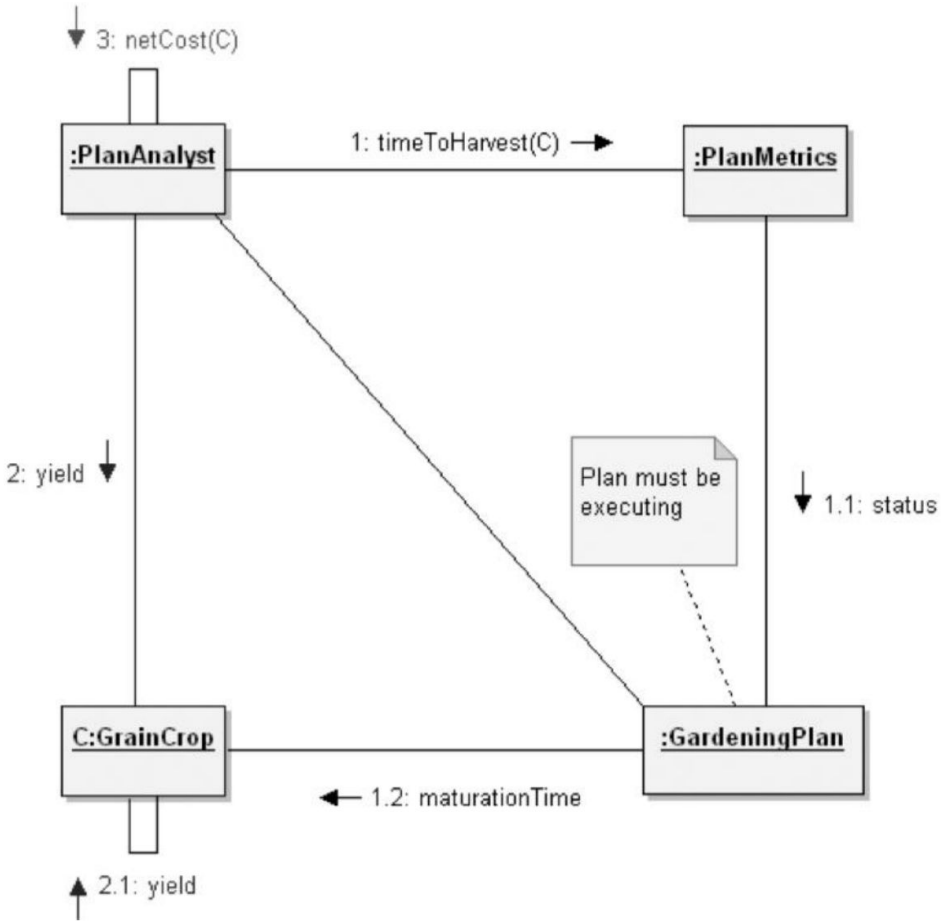


图5-78　图5-77的另一种画法，显示了不同的次序编号

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

尽管有点造作，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。

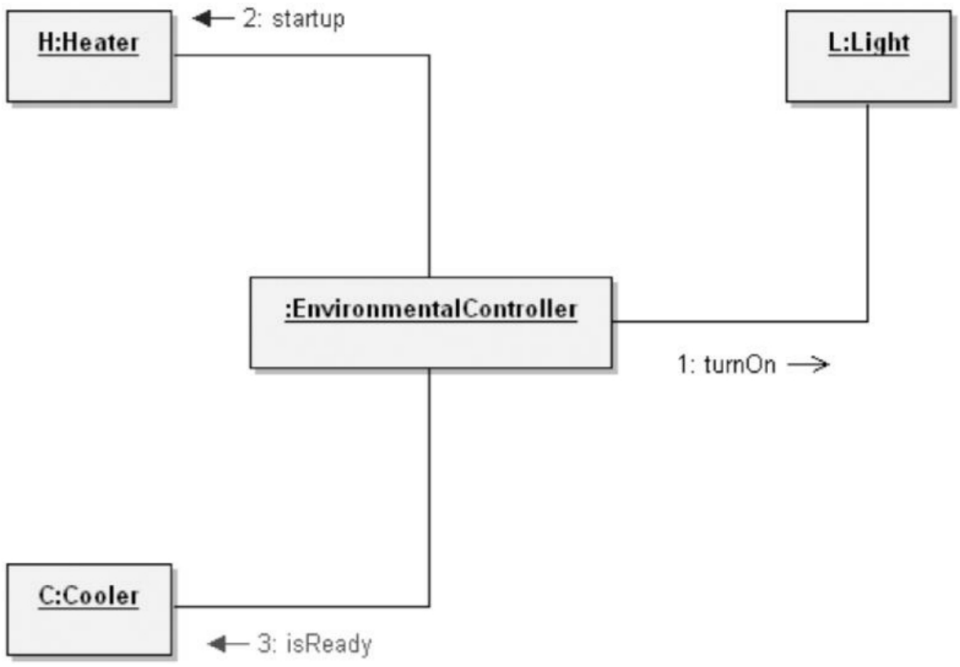


图5-79　对象与同步

5.14.4　高级概念：迭代子句和约束条件

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]）。

约束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。约束条件放在方括号内，如图5-80中startup消息上的约束条件。这个条件表明，消息将在约束条件为真时执行，在本例中就是当温度低于预期的最低温度时。约束条件的表示方式取决于个人喜好。

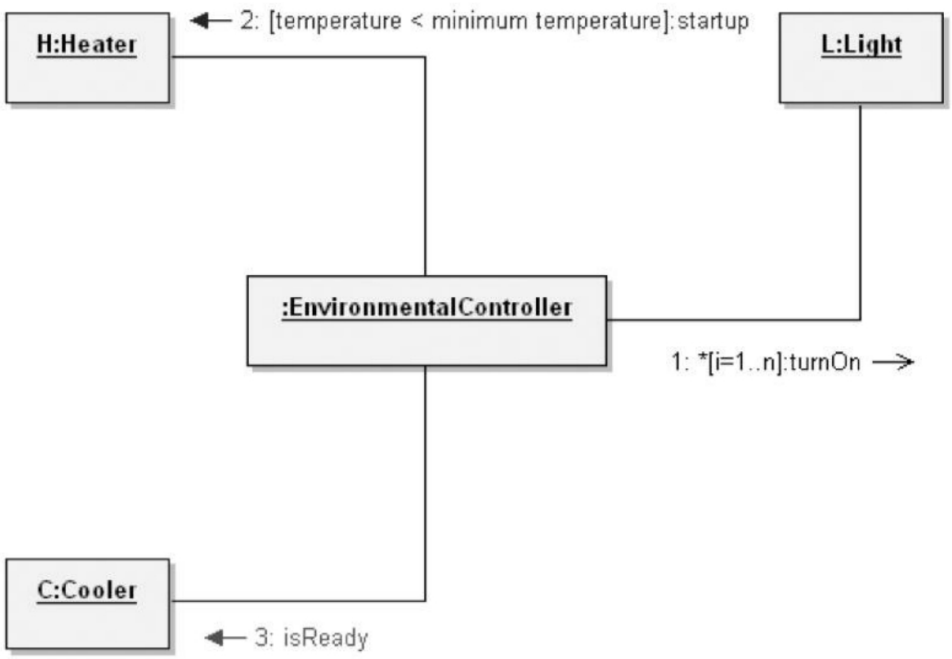


图5-80　通信图中的迭代子句和约束条件

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职责，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。

---

[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。）

[2]Fragaria × ananassa有一些常用的名称，包括commercial strawberry、garden strawberry、cultivated strawberry及plain strawberry。

第6章　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，即那些导致成功项目的特征开始。

6.1.1 成功项目的特征

成功的软件项目是指：提交产物达到或超出客户预期，开发过程符合时间和费用的要求，并且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，我们遇到的所有成功的面向对象系统基本上都有一些特征[1]，而失败的系统中显然没有：

- 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，这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”是非常关键的[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。

实现抽象分类没有正确的方法，同样，设计给定系统的架构也没有正确的方法。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关注点分离，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，也更为健壮和可靠。它们还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、浮现式设计、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。不论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，系统就较难以随时间演进，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。[2]

迭代和增量式开发是以连续系列的发布版本（迭代）的方式来开发系统的功能，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，取决于对项目风险的缓解，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[3]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是在早期建立的，这样更容易确定可复用的组件和利用已有组件的机会。
- 可以更早发现缺陷并修正。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。

- 开发过程可以得到优化和改进。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的迭代过程。

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。本小节将讨论当今各种过程风格的范围，并针对如何选择最符合项目和组织机构需要的过程，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。[4]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）；
- 浮现式而不是预先确定（在实际执行过程中让过程演进，而不是事先计划或确定）。

敏捷过程让软件开发团队不必遵循一组严格的步骤，使开发者能够将他们的创造力集中在被开发的系统上。

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）；[5]
- 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，最好是通过两个步骤来确定。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，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，需要将项目的特征与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
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，这应该是我们的目标。

本章后面的内容描述了一个软件开发过程的框架，它已针对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。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情形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“交付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[6]，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。[7]

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。

交付之后软件的演进和维护

当一个软件系统交付之后（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。

- 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）。

通过重新执行与原来软件开发生命周期类似的生命周期，可以为已部署的软件系统开发变更并交付，不过维护生命周期的规模和情形（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，主要包括交付阶段），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。

在交付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，交付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，设计将作为特定实现环境下实现的规格说明。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（即需求得到了正确的实现）。通过具体的展示来验证软件产品是否像设计那样工作。

（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打算抛弃的。

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关系方面的灾难。

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，就构成了宏观过程的一次迭代。

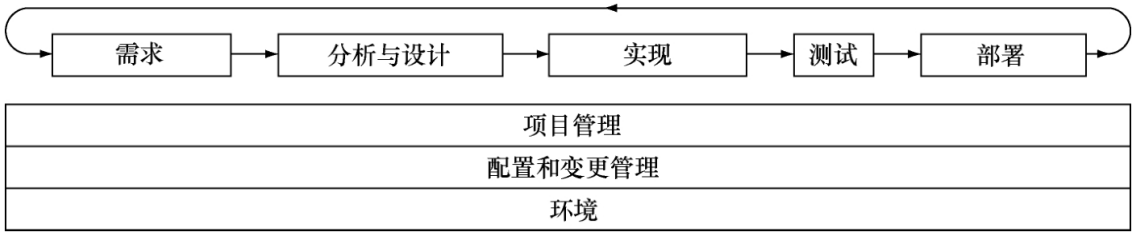


图6-1　宏观开发过程科目

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。

宏观过程的许多要素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。[8]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。

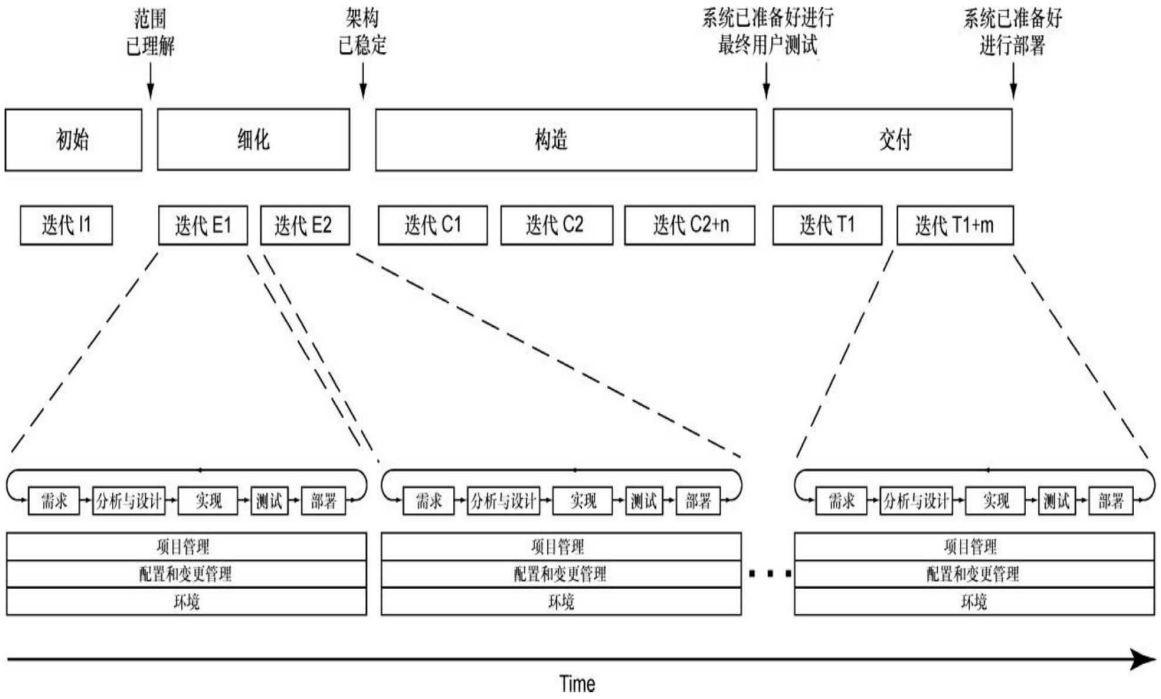


图6-2　宏观过程里程碑、阶段和迭代

目标　初始阶段的目标是确保项目有价值并且可行（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，有时候在开发者、架构师、分析师或最终用户的头脑中会突然出现一个想法。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，但仍然专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和开发环境。系统愿景提供了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　在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，在设计航天器时，计算机的重量是一个关键因素，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。放宽某个约束条件，其他设计选择就成为可能；加强某个约束条件，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，哪些约束条件只是幻觉。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着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的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，也包含了系统的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．交付

本小节将介绍交付阶段的目标、活动、工作产品和里程碑。

目标　在交付阶段中，确保软件被它的最终用户接受。

活动　在交付阶段，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。交付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。支持文档也将进行最后的开发，还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　交付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），交付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”。

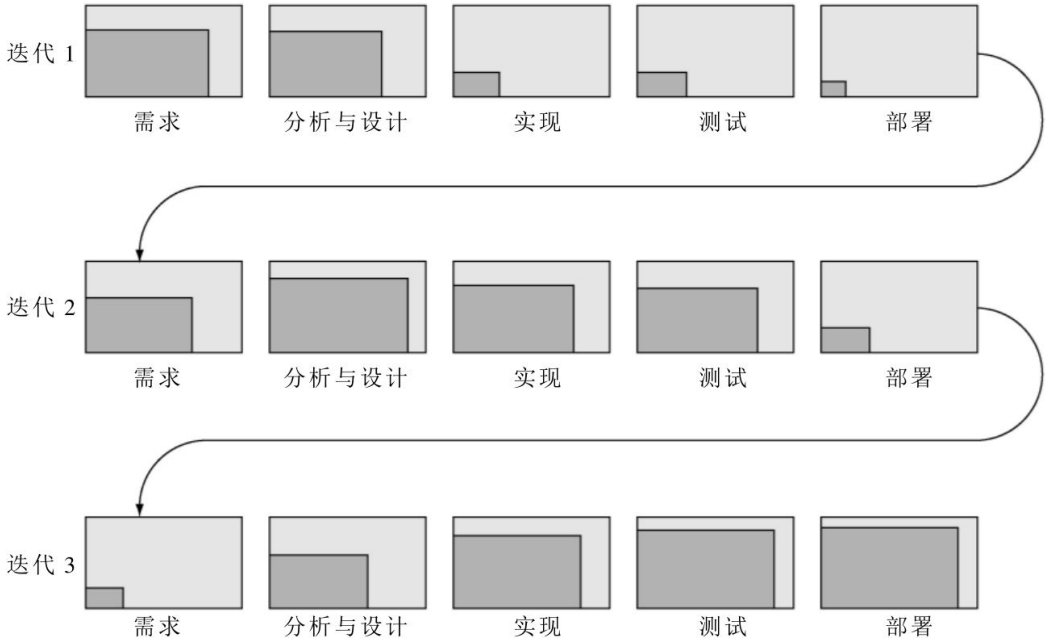


图6-3　迭代关注焦点的转移

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，这时我们关注的是建立软件的架构。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，如果在向用户交付系统时，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，这时我们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）。在决定单次迭代预期能完成多少工作时，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，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。我们利用这些优先级，来确定哪些需求将分配到哪些迭代中。

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某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，架构的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、架构师和品质保证人员。

在优先级确定之后，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。要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一些品质问题，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，并且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，其成本是相对较高的（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，尤其是这样。

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）已被证明是成功的常见障碍，它显著而微妙。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在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

前面一节讨论了整体软件开发过程（宏观过程）。本节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。

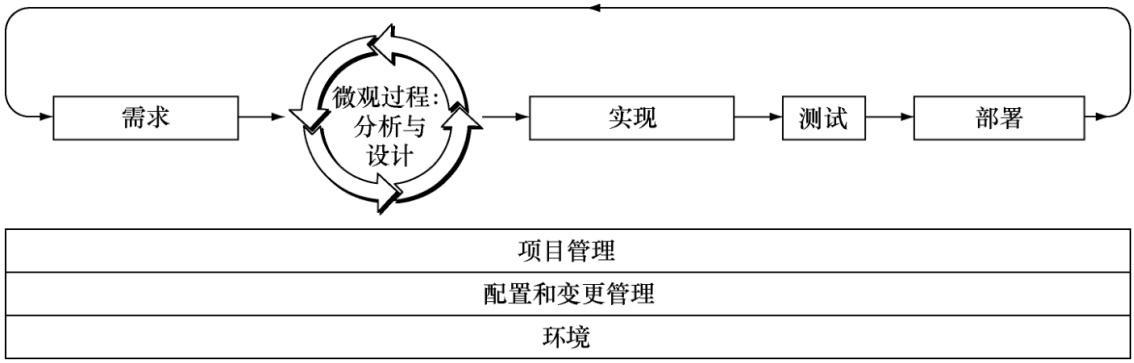


图6-4 宏观过程中的微观过程

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、职责和协作。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，得到清晰的关注点分离，在系统各元素间平衡地划分职责。

在设计时，你会创造一些元素，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关注点分离，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，以及它们的职责、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。

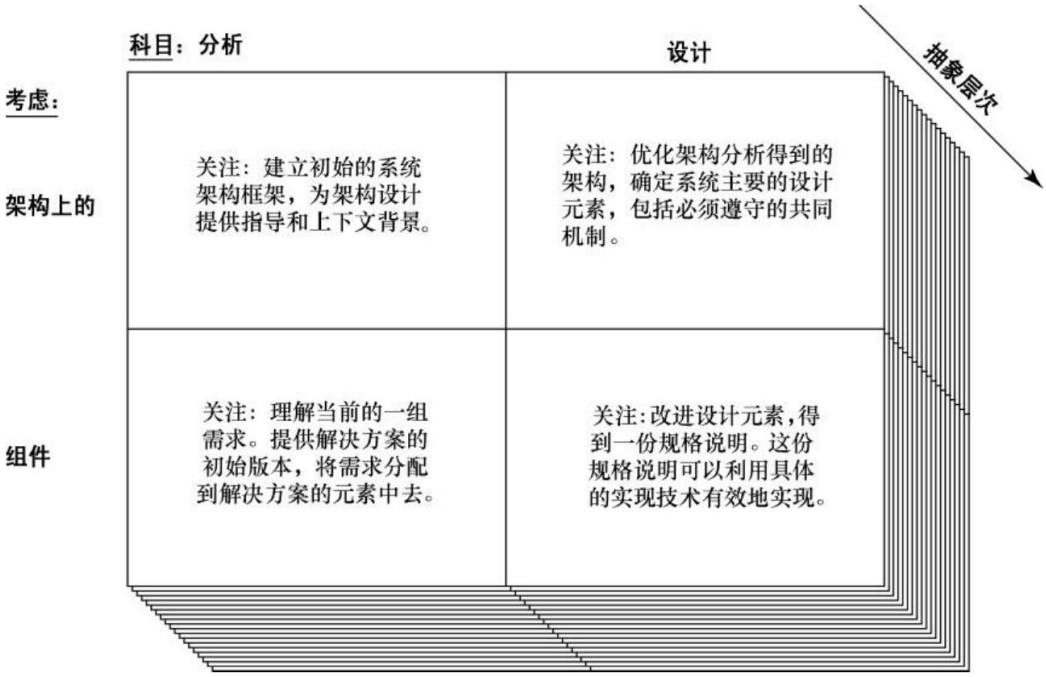


图6-5　从不同视角来看分析与设计的不同关注点

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：[9]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。

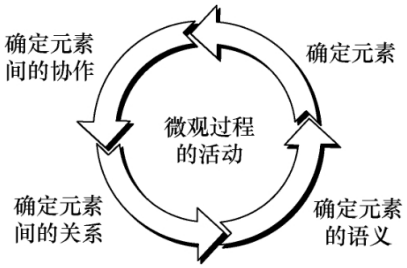


图6-6　微观过程的活动

虽然这些活动看起来是依次执行的，但实际上它们是并行执行的。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。

本章稍后将会更详细地讨论微观过程的每一项活动。下面让我们来看看微观过程的产品。

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，其重要性在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及其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。如你所料，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“一图胜千言”的好处，而且让分析/设计模型可视化也有助于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。[10]如果待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）。

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，这些人即使会阅读软件设计说明，也不太会仔细阅读。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。

- 进程视图：进程视图描述了系统中的独立的控制线索和哪些逻辑元素参与了这些线索。

- 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非常关键的，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，这些决策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：

■ 在进行“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，通过架构分析所产生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职责和交互。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职责和交互。这些分析元素代表了系统组件的第一次近似，然后应用于组件设计，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。[11]

然后你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。

（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。

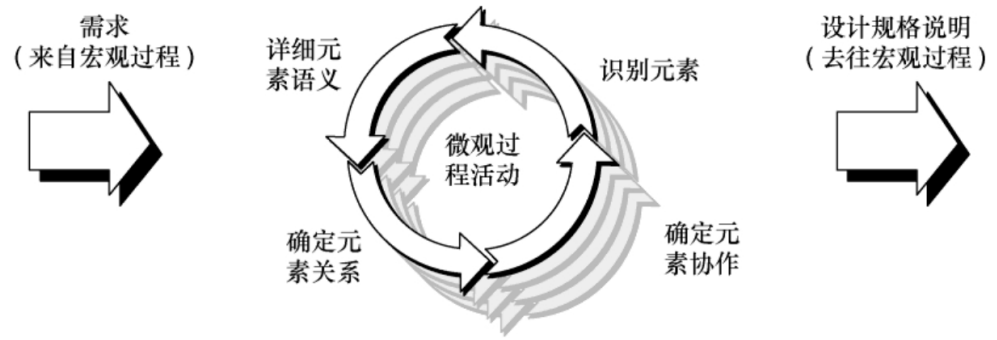


图6-7　微观过程迭代

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。确定元素时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 与晚确定关键抽象（在通过单项需求来确定元素时）相比，早确定关键抽象将减少概念定义冲突的可能性
	架构组成部分	■ 表达系统中关注点分离的领域，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与领域无关的问题，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策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 ■ 用内聚的职责来描述独立的元素
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，重新定位职责，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。

- 具有一组重要职责的分析类。
- 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职责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。
- 几组具有相同职责的分析类。类似的（或相关的）职责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追踪性是聪明的做法。建立并维护可追踪性对于有效、准确地影响评估是非常关键的。

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，若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，我们通过明智的、可测量的职责分配，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如何通过处于某个抽象层上的元素的彼此协作来实现。实现反映了协作元素之间明确的职责分配，并在行为需求和软件解决方案之间架起了桥梁。实现最初是以分析元素描述的，后来以设计元素描述。

实现以及支持的元素职责通过分析/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职责。通常使用一个短语或一句话就足够了，对于某项职责，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，将职责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，用更抽象、更一般的实现方式来表达这些模式。

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），或是在分析职责/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　利用行为分析，我们可以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识别出来了。）

（3）推演这个场景，将职责分配到每个元素，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职责所需要的。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，对职责进行重新分配，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职责。将较大的职责分解成一些较小的职责是常见的动作，将较小的职责组合成较大的

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职责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、内聚的职责。

（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及状态转换的一种准确方式。

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职责，寻找行为的模式。共同的角色和职责应该统一为共同元素的共同职责。

（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职责，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、平衡的职责分配，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。

对于单个的元素职责来说，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，记录下元素的职责就足够了。在较高的抽象层次上，可对职责进行有意义的说明。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职责。一个元素不应该“什么都做”。
- 元素应该具有一致的职责。如果元素的一些职责是无关联的，那么它应该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职责。
- 为每个元素定义的职责应该支持该元素所参与的场景。
- 职责不简单或不清楚意味着给定的抽象没有得到很好的定义。

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关注点分离，这种关注点分离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正在演进的分析/设计模型之中。

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进行可视化表示。可视化的图示提供了更全面的架构视图，这样在表达这些关系时，就能够不受编程语言的限制。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，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，它们已经通过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承结构中，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，只需包含重要的分析或设计信息，或实现所必需的信息。

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。

在这个阶段要注意一件事，即参与实现的元素关系的一致性。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。在复查在这个活动中确定的关系时，你追求的是拥有逻辑上高内聚和低耦合的元素。另外，你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而为之的：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机会，不能得到更小、更简单的架构。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部视图以及元素之间的协作，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。细节水平和记录元素语义的表示方式，都取决于你所面对的抽象层次。

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，向它们分配职责。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职责和总体流程。某些元素的职责受到事件的驱动，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。[12]

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，你可能为具体的操作命名，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，你可能还会指明应该使用某些算法。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。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职责。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职责。通过在所有实现中查看所有调用该元素的协作，来确定该元素的职责。（元素的职责就是其他元素可以要求它做的所有事情。）

（2）更详细地描述每项职责。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，为每项职责/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。考虑打算赋予职责的那些元素。在理想的情况下，这可能只需要稍稍调整较低层元素的职责和协议。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）。要记住从该元素的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）在为每个元素定义职责时，考虑该元素为了实现其职责而必须具备的属性。

（5）在设计时，设计一组足够的操作，来实现这些职责。如果可能，尝试复用概念上类似的角色和职责。作为单个类来说，职责就是由该类的操作记录的；作为组件来说，职责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、复制和析构的需求[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。

在细化元素的职责时，你可能发现一些新的元素支持详细的职责描述。（例如，当更完整地描述一个元素的职责时，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职责，针对这些元素重复刚才描述的步骤。

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：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职责和属性的描述，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是如何相互支持的。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解项目风险来决定的，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光谱中的位置。

■ 本章从两个方面来描述软件开发过程框架，即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）。

[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。

[2]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]。

[3]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。

[4]Boem和Turner详细讨论了敏捷过程和计划驱动过程之间的差异[38]。

[5]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）。

[6]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”。

[7]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。

[8]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）。

[9]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。

[10]关于软件开发过程的更多信息，请参见6.1.2小节。

[11]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。

[12]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。

第7章　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、功能缺失等。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，设计软件不是一门精确的科学。请考虑使用实体-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，“虽然这听起来相当简单，但它确实包含了一些关于企业中不同对象重要性的个人见解。结果导致设计流程没有确定性，对于同一企业，不同设计人员可以得到不同的企业模型”[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，我们都不能忽略真实世界设计系统的实践问题。

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不仅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、顾客和开发团队之间的关系管理（以便在分析期间发现系统的真正需求），以及诸如此类的问题。

正如前一章提到的，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，如任务计划和走查，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复查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都是合适的。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进度计划外的时间可供支配，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的小调整，以确保项目稳定进行：没有哪个项目可以承受某个开发人员空闲下来，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风险和长期的风险，并在必要时集中开发资源来处理这些风险[2]，随后还要管理下一个微观过程迭代，以便得到一个稳定系统，满足该发布版本计划要实现的用例场景。这个级别的任务计划常常在制定时过于乐观，所以不能很好地完成[5]。开始时开发被看作“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对来自于系统其他部分的某些协议做出假定，随后却意外地发现交付了不完整或不正确的类时，进度计划就被抛到了窗外。更危险的是，可能因为出现了必须解决的性能问题，或者因为战术设计决策出现失误，而导致进度受到致命的破坏。

要想免受过于乐观的计划之苦，关键是校准开发团队及其工具，这是一个持续的过程。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做出计划。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，通常这是只有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，管理人员可以通过这些里程碑来收集每个开发人员的跟踪记录数据，以制定和实现进度计划。随着演进式开发的推进，这意味着经过一段时间，管理人员会更好地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开发工具。

7.1.3 开发复查

开发复查是一项充分发展的实践，每一个开发团队都应该采用。由于有了任务计划，软件开发复查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，要复查的东西就有分别了。

管理人员必须采取措施，在过多和过少的开发复查之间取得平衡。在所有系统中（安全攸关的系统除外），复查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宝贵的资源，复查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架构进行正式复查，而对较小的战术问题进行更多的非正式复查。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。

正如在上一章中提到的，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。用例场景的正式复查由团队的分析员（精通用例开发）、领域专家及其他最终用户共同进行，其他开发人员也在场，包括QA员工（测试员）。这种复查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大规模的复查，如果到那时再发现问题，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。[3]最后，这种复查还可以为系统的开发人员及用户建立一个共同的词汇表。开发团队的其他成员目睹这些复查，可以让他们在开发过程的早期就直接面对系统的真实需求。

架构复查应该聚焦于系统的整体结构，包括它的类结构和机制。像用例场景的复查一样，架构复查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。早期的复查注重于扫清系统架构的问题，以后的复查可以聚焦于某个组件或具体的、无处不在的机制。复查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愿景。复查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，便于以后简化架构。

非正式的复查应该定期进行，与开发的进度相符，通常包括对某个组件或低层机制的同级复查。这些复查的目的是验证这些战术决策，它们的第二个目的是为高级开发人员指导初级开发人员提供一种机制。

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，区别在于开发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

管理面向对象项目有一个更值得高兴的方面，即与单个瀑布这样的更传统的方法相比，在稳定的状态下，会减少对总体资源的需求，并在资源的使用时间方面会有一些变化。这里的关键短语是“在稳定的状态下”，同时团队必须是有经验的。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历一条学习曲线。对象模型的基本资源优势并不会马上体现出来，而是要等到第二甚至第三个项目，那时，开发团队将更加熟练地掌握面向对象分析设计（从架构设计到类设计），熟悉如何获得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让他们开始架构方面的探索。在后续的增量中，需要的资源通常较少，其主要原因在于，正在进行的工作可以使早期架构设计，过去的增量中创建的通用抽象和机制也可以发挥巨大的作用。测试需要的资源也更少，首要的原因在于，为类或机制中增加新的功能主要通过修改结构来实现，而我们知道这样的结构原来的行为是正确的。因此，测试通常在生命周期中更早的时候开始，并且清楚地表明它是一种累积式的活动，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，而不是以一种“大爆炸”的方式出现。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。而且，

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，因为最终产品的质量会好得多，所以也更适应变化。

7.2.2 开发团队角色

记住，软件开发归根结底是人的行为，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且不同的技能。

经验表明，与传统方法相比，面向对象开发过程对技能的划分应该稍有不同。我们发现在面向对象项目的开发团队中，以下三类角色是核心：

- 项目架构师
- 组件主管
- 应用工程师

项目架构师是梦想家，负责系统架构的演进和维护。对于中小型系统来说，架构设计通常由一个或两个特别有洞察力的人负责。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，选择他们是由于他们具有建造类似系统的丰富经验。正是因为这种经验，架构师凭直觉就知道给定领域相关的通用架构模式，也知道特定的架构变体可能出现什么样的性能问题。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，因为他们最终必须通过一些类和对象的协作来表达他们的架构愿景。

雇用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，让其他人来承受他做出的这些决策所带来的一系列后果。更好的做法是让一个架构师积极参与分析过程，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，架构师将承受他的架构决策所带来的影响。另外，通过让一个人或小型开发者团队负责架构的完整性，更有可能得到比较有弹性的架构。

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。每个组件主管必须和项目架构师一起，设计、辩护和谈判具体组件的接口，然后指导其实现。因此，组件主管是一组类和相关机制的最终拥有者，同时也负责在系统演进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，但是他们缺乏架构师所具有的丰富经验。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，通常承担一项或两项职责。某些应用工程师负责在组件主管的管理下实现一个组件。这个活动可能包括某些类设计，但通常包括实现类，然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。接下来，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，通过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种方法是让应用工程师负责更多的类细节设计，但要保证组件主管提供充分的监督和引导。

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点。一般来说，一支开发团队中应该包含一半或更多的应用工程师。

这种技能分解解决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、资源和进度计划的动态管理。

■ 分析员　　负责演进和解释最终用户的需求，必须是这个问题域的专家，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、组件和设计的版本库；通过参与复查和其他活动，积极寻找共同点，使它们可以被采用；获取（例如，通过商业的库）、生产和改造通用的类和组件，在项目或整个组织内复用。

■ 质量保证员　　负责测量开发过程的产物。通常指导对所有原型和产品发布版的系统级测试。

■ 集成经理　　负责组装已发布组件的兼容版本，以便形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　　负责编写产品及其架构的最终用户文档。

■ 工具编制工程师　负责创建和调整软件工具，以利于项目交付物的生成。

■ 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个开发组织。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，面向对象开发使得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非不可能。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。

即使项目采用了最精密的设计方法或最先进的热门工具，也不意味着管理人员有权利放弃职责，不去雇用那些能够独立思考的设计人员，或是让项目以“自动驾驶”的方式运行。

7.3 发布版本管理

面向对象开发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部件（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

请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。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细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。

这个模型中蕴含的思想就是：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。这并不是意味着不对类做版本控制，只是这不是首要焦点。实际上，这就意味着对组件进行版本控制，因为一组类映射到组件。在软件系统内更高的层次上，将对多个低级别组件构成的子系统进行版本控制。

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。这就更需要强有力的配置管理工具和版本控制工具。

源代码不是唯一应该纳入配置管理的开发产物。同样的概念适用于面向对象开发的所有其他产物，如用例规格说明、可视化模型和软件架构文档。

7.3.2 集成

工业级的项目要求开发程序家族。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，以及开发和测试的辅助工具和代码。每一个开发人员通常都拥有他自己正在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进而来。正如Davis等人所说的，“在采用增量开发时，最初的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，从而达到更高的适应性”[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。这种策略往往会降低开发风险，因为它更容易在开发过程的早期发现架构设计和性能的问题。

对于中等规模的项目来说，一个组织可以每隔2、3个月产生一个内部发布版本。更复杂的项目需要多得多的开发工作量，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，就有可能构建一个发布版本。

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。组件测试可以当成是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，系统测试通常也用做回归测试。

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，以便了解系统在特定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，但要享受到这个好处，实现开发过程中许多工件的复用，还需要管理上的承诺。

7.4.1 复用的元素

软件开发中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，类是首要的语言复用手段：类可以派生子类，从而特化或扩展基类。同样，如第4章中所述，我们可以通过惯用法、机制和框架的形式，实现类、对象和设计模式的复用。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）。

引用复用水平的数字很危险，容易产生误导[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，因为根据领域不同，潜在的复用有着多种表现形式，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，复用总比没有复用要强，因为复用代表了资源的节省，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，这就是将模式提取作为一个明确的活动包含在微观过程中的原因。

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。复用活动包括识别通用化的机会，这通常是通过架构复查来发现的；复用活动还包括利用这些机会，这通常是

通过生产新的组件或改造已有的组件，并在开发人员中支持复用来实现的。这种方法需要给予复用明确的奖励，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。

除了开发可复用的资产之外，还可以通过购买商业类和组件库来协助我们开发。但是，我们仍然必须在架构的框架之内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放在一起那么简单[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。组织只有以长远的眼光来看待软件开发，而不仅仅是为眼前的项目优化资源，才能使复用活动最终取得成功。

7.5 质量保证和测量指标

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]。质量保证的目标是对一个软件系统的质量有多好给出可量化的测量。很多传统的测量指标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。

这就是我们在面向对象开发过程中将重点放在软件架构上的原因。一个简单的、适应变化的架构是所有高质量软件的关键，它的质量完全来自于简单一致、支持战略设计的战术设计决策。

正如前面所述，即使是在面向对象系统中，开发复查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。关于软件质量的优劣，最重要的量化指标可能就是缺陷发现率。在系统的演进过程中，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，按时间分布的缺陷发现率就说明了错误发现的快慢。正如Dobbins所说的，“线的斜率比错误的实际数字更重要”[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，就是它允许我们尽早地、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，并画出缺陷发现率按时间分布的图形。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。

缺陷密度是另一个相关的质量测量指标。测量每千行代码（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，缺陷密度通常“在大约检查了10000行代码后达到一个稳定值，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。采用这种测量指标，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 面向对象测量指标

英国物理学家开尔文（开尔文温标就是以他的名字命名的）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，你就了解了一些事情；但当你不能测量它，不能将其表达成数字时，你的知识就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已是迈向科学的阶段”。[4]对于面向对象测量指标，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。

两大类测量指标能帮助我们：过程测量指标和产品测量指标。过程测量指标，有时称为项目测量指标，可以帮助管理团队评价采用面向对象开发过程的进展。过程测量指标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——这些都与计划进行比较。我们也可以看一看面向对象开发特有的一些测量指标，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
  - 每个类的人-天数（PDC）

- 每个开发人员的类数（CPD）
- 调度
- 主要迭代的数目（NMI）
- 已完成合同的数目（NCC）

我们测量开发进度的一般方式，是统计已完成的、能工作的逻辑设计类或物理设计组件。正如之前章节中所描述的，另一个进度测量指标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。随着时间的推移，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，因为大多数精力用于让已经设计好的类和组件一起工作。有时候，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，也只会在仔细考虑了它们的影响之后才做出这样的改变。这些改变可以增量地引入产品系统，作为平常的发布生命期的一部分。

在这里，我们的首要焦点不是过程测量指标，而是产品测量指标（有时称为设计测量指标），这些测量指标帮助开发团队评价他们的分析和设计工作。我们发现，适当的产品测量指标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测量指标获取和分析适用的量化测量数据，将得到一个历史数据库，可以和当前项目所分析的测量数据进行比较。

Chidamber和Kemerer建议，以下几条独立于语言的测量指标可以直接应用于面向对象系统。[17]

- 每个类的加权方法（WMC）
- 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单个类中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。

如果我们为每个方法指定相对复杂的值，这个测量指标确实管用；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，方法太多的类比方法少的类更加复杂，更加针对特定应用，并且常常包含更多的缺陷。[17]

继承树的深度和孩子的数量是类结构的形状和规模的测量指标。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。继承树的深度就是从主题类到最高层父类的层数，它提供了一个测量指标，从继承功能的角度说明了对主题类的影响。因此，由于类继承的功能，更深的继承树增加了该类的复杂性。

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，由于它导致的复用，它在软件系统设计中的影响也越大[17]。

对象之间的耦合是它们到其他对象的连通性的测量指标，因此是类的依附集的测量指标。与传统的耦合测量指标一样，我们追求设计松耦合对象，它们有更大的复用潜力。

类应答数是类方法的测量指标，它表明类的实例在应答一条消息调用时执行的方法。一般来说，如果某个类可调用的方法比同级别的类多很多，它就更复杂。

方法内聚缺乏度是类抽象的统一性的测量指标。一个类的方法间的内聚程度低，说明这是偶然或不适当的抽象。这样的类一般应该重新抽象为多个类，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（后来由Lorenz 和Kidd在1994年提出）以及Chidamber和Kemerer提出的产品测量指标的例子。Lorenz的11个设计测量指标包括若干面向对象设计指南，以及应用它们的经验法则。Kan发现经验法则“非常管用。它们来自产业级OO项目的经验，提供了比较和解释的入口”[18]。

将Chidamber和Kemerer测量指标（CK测量指标）应用于若干研究后，Kan发现“在决定CK测量指标的偏好阈值之前，还需要更多的实证研究”[18]。事实上，Chidamber和Kemerer说，应该为每个开发地点确定阈值。Kan确实发现，“在实际使用中，该测量指标可以标出偏离较远的类，从而引起特别的注意”[18]。

最近，Kemerer和Darcy提供了若干应用CK测量指标集的例子，并提供了关于它的实践应用的观察。从这些应用的研究中，他们对面向对象测量指标给出了若干评语[19]。

- 这样的测量指标已经成功地应用于若干领域。
- 它们一致地展示了与质量因素的关系（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测量指标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，人们对此仍有不同意见。因此，对于如何构造适当的面向对象测量指标集，依然有很多争论。我们相信，这里展现的测量指标提供了合理的测量指标集合，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，还有一些开发工件对软件系统的整个生命周期也是至关重要的。这些工件，如需求和设计，必须形成文档，以支持开发过程及系统的运营维护。

7.6.1 开发遗产

软件系统的开发不仅仅包括源代码的编写。某些开发产品提供了一些方式，让管理团队和用户查看项目的进展。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一些可视的模型，以图的形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。这些图与适当的指南一起，让我们能够向后跟踪系统的需求。用例图（及用例规格）展示了需求所规定的高层功能。活动图描述用例场景的细节。类图则代表关键抽象，这些抽象组成了问题域的词汇。有些类的复杂行为与状态有关，就用状态机图来表示。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

为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产物，但却是次要的。文档也是活生生的产品，记住这一点是很重要的，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。交付的文档与设计及生成的代码一起，成为大多数正式或非正式复查的基础。

哪些东西必须文档化？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[5]。另外，必须产生分析文档，目的是通过用例场景记录下系统所需功能的语义。我们还必须生成架构和实现文档，目的是与开发团队交流关于架构的愿景和细节，同时保存相关的战略决策信息，使系统易于修改，易于随时间演进。

通常，系统架构和实现的基本文档（不一定是纸质的）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。

对面向对象系统来说，最糟糕的文档就是为每个类、每个方法的语义提供孤立的描述。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，相当于用石雕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，所以我们需要那些聚焦于更丰富语义的工具。另外，面向对象开发的宏观过程确定了各个发布版本的快速开发，所以我们需要有工具支持分析设计周期的快速转换。

工具良好的伸缩性是很重要的。一个工具可以支持一个开发人员编写小的、孤立的应用，但它并不一定适用于更复杂应用的产品发布版本。实际上，每个工具都有能力的限度，超出这一限度，它的优点就会被缺点和不方便所压倒。

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主要工具。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，保持对设计产品的控制，并协调一个团队的开发人员的设计活动。实际上，随着设计演进到生产实现，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对面向对象系统中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。如果没有这个特性，设计人员可能会生成奇妙的可视化设计表示，但在实现开始进行时，就会发现这些图已经过时了，因为程序员只是醉心于实现而没有更新这些设计。逆向工程使设计文档跟不上实际实现的可能性变得很小。

接下来，我们需要一种软件配置管理和版本控制工具，较大型的项目尤其如此。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。这些资产包括所有分析和设计过程的工件，从用例图到类图、序列图，它们描述了架构设计。这些资产也

包括在架构内协作的组件。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。

我们发现，面向对象分析设计中第三种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。随着一个项目逐渐成熟，类库成长为领域特定的可复用软件类，组件也随着时间的推移而增加。用不了多长时间，类库就会变得巨大，这会导致开发人员很难找到满足自己需要的类或组件。如果感觉查找某个组件的成本（通常会高估），要高于感觉从零开始创建这个组件的成本（通常会低估），所有复用的希望就都破灭了。因此，重要的是至少要有某种最小的类库工具，让设计者根据不同的条件来定位类和组件，并将开发出来的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基础平台，让可视化建模、配置管理和版本控制以及类库工具进行协作。

7.7.2 组织上的意义

由于需要强有力的工具，导致了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项目的类库。如果没有积极的努力，这样的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，复用工程师从现有的设计成果中提取有用的产品，使复用过程变得容易。工具编制工程师的职责是创造领域特定的工具，同时裁剪现有的工具，使之符合项目的需要。例如，一个项目可能需要通用的测试支持工具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打造这些工具，通常这些工具来自类库中已有的组件。这样的工具也可以用于以后的开发工作。但最好是有集成工具套件，这样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。

如果管理者已经面临人力资源匮乏，可能会叹惜使用强大的工具过于奢侈，任用复用工程师和工具编制工程师也是如此。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，我们发现这些活动总是以某种方式在进行，通常是以自发的方式。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，有一些特别关注的主题。领域特定问题包括高效用户界面的开发和遗留功能的集成，遗留功能包括从数据到整个系统。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，某些应用领域有充分的理由采用特定的架构。高效用户界面的设计就属于这样的应用领域，它依然更多地偏向艺术而不是科学。对于这个领域，使用原型是绝对必要的。来自最终用户的反馈必须尽早收集并经常补充，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一些数据库，其中的表结构不可改变，这通常是因为数据库中已有大量的数据（遗留数据的问题）。对这样的领域，可以直接用关注点分离原则：对所有这样的数据库的访问最好限制在定义良好的接口类之内。如果面向对象分解和关系数据库技术混用，这项原则就特别重要。

另外，请考虑实时系统。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。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）优化，认识到这一点很重要的。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暴露系统性能的瓶颈，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，取代目前由遗留系统提供的某些功能。当然，要点是从一个架构愿景开始，它表明最终系统看起来会是什么样子，这样逐步替代遗留系统就不会最后变成不一致的软件补丁。

7.8.2 采纳面向对象技术

正如Stix和Mosley所报告的，“随着信息系统社群顺从市场的面向对象需求，许多认知问题需要关注……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，它们必须同时学习，以便高效地实现并达到对象技术的好处”[20]。

如何培养这种面向对象设计的能力？我们推荐以下思路。

- 为开发人员和管理人员提供以下正式培训：
  - 统一建模语言。
  - 项目准备使用的面向对象分析和设计流程。
  - 项目准备使用的工具。
  - 项目准备使用的语言和库。
- 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：
  - 聘请有经验的OOAD顾问作为项目团队的导师。
  - 在团队成员中培养专家，让他们作为面向对象方法的现场指导者，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。

根据我们的经验，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，并不意味着你已经理解了如何利用UML完成面

向对象的好设计”[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，那么在面向对象开发中引入新的开发人员和管理人员就会容易许多。开发人员可能从分析师做起，当他们更熟练地把握面向对象技能时，就会逐渐成长为设计师。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，在更有经验的人的指导下，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，已经宣传了很多年，而且已经十分现实。但是，如果没有很好地应用面向对象开发过程，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，改善生产进度的预估；第二，这些项目可能有一些非常复杂的问题，所以似乎没有其他任何解决方案。

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）、关注点分离、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。
- 使系统在变化面前更具有弹性。
- 鼓励了软件组件的复用。
- 减少了开发风险。
- 利用了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，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。
- 通过继承来组织相关的元素。
- 通过多态来执行操作，自动适配到所操作的结构类型。
- 特别的分析和设计方法。
- 面向对象语言。
- 有利于创建面向对象系统的环境。
- 按契约设计，即一种强有力的技能，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，有利于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。

在映射的另一面，他参考了Boehm的10大软件风险[27]以及来自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。
- 商业上架销售产品、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、性能或质量的不足。
- 不断的需求变更。
- 外部执行任务不足。
- 过度追求计算机科学。

Hantos详细解释了这8个风险，并解释了在软件开发项目中，他列出的面向对象概念如何增加或帮助缓解特定的风险。

为了让结果简单可视化，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“不断的需求变更”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“不断的需求变更”的风险[25]”。

如果我们回忆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），我们可以分隔结构和行为，以减少贯穿系统的需求变更波动效应。项目团队引起的所有系统开发缺陷都通过类似的手段分隔。

在为软件开发项目制定风险管理计划时，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风险）是很重要的。

7.10 小结

- 一个复杂软件系统的成功开发和实施不只是产生代码。
- 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稍有不同。
- 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有用的面向对象系统质量测量标准。其他有用的测量标准包括不同过程测量指标和产品测量指标。
- 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。关键是开发团队要理解面向对象分析设计技术，面向对象软件开发不只是编程。
- 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，同时使风险最小化。

---

[1]韦氏新世界大学词典（Webster’s New World College Dictionary）将“实战（pragmatic）”定义为“关心确切的实践，而不是理论或投机；实践的”。

[2]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]

[3]我们在复查中曾遇到过各种非开发人员使用这种表示法，如天文学家、生物学家、气象学家、物理学家和银行家等。

[4]据说开尔文勋爵（LordWilliamThomsonKelvin）是1891—1894年期间在*PopularLectures 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。

[5]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，让它们的使用直观且一致，以便将用户对最终用户文档的需要减少到最低限度。

第3篇　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，每一个应用都涉及到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。从初始出发，经过细化到构造。（交付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交付的考虑。）也就是说，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，以提供上下文背景，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和应该用什么图的关系，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。不同的是图所记录的抽象层次。例如，在生命周期早期，组件图可能记录非常大的、粗粒度的元素（如系统或子系统）。在生命周期后期，组件图可以用来记录细粒度的实现元素（如可执行的软件）。贯穿本部分的所有章节，将看到抽象层次上的细化。

第8章　系统架构——基于卫星的导航

本书前面的部分展示的面向对象分析设计原则和过程，以及在第5章讨论的UML 2.0表示法，既适用于最高层次的系统架构开发，也适用于软件开发。与开发类的结构和设计不同，系统架构关注的是理解系统需求，并利用这些知识来划分较大的系统。但是，我们必须记住这个层次的关注点通常十分抽象、宏观、影响巨大，且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寿命很长的系统——更加可操作、可维护和可扩展。本来就应该是这样的。

本章将展示如何在逻辑上对所需功能进行划分，来开发假想的卫星导航系统（SNS）的系统架构。为了让这个问题可管理，我们从简化的视角开发第一和第二层次的架构，分别定义组成部分和子系统。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。如果要从一个更完整的视角展示这些部分及其子系统的规格说明，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层次（如部分或子系统），贯穿卫星导航系统架构的多个层次，完整地应用。

我们选择这个领域，是因为与仅作为案例问题而发明的简单系统相比，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，欧盟正在开发第三个系统，名为Galileo。

8.1 初始

开发系统架构首先要进行的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“安排元素和子系统，并给它们分配功能，以满足系统需求”[2]。

我们此时关注的焦点是确定必须为顾客建造什么，即定义问题的边界，确定任务用例，然后通过分析任务用例之一来确定一个系统用例子集。在这个过程中，我们从功能需求中形成用例，并用文档记录下非功能需求和约束。但在进入需求分析之前，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

全球定位系统让任何拥有GPS接收器的人知道他在地球上的位置，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监控。第一颗GPS卫星在1978年升空，到1994年系统完成时的第24颗，GPS成了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，后来取得了许多技术进展，一直到今天的GPS。在此期间，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个部分：控制、用户和空间。控制部分由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部分，它接收来自24颗卫星的位置信息，这24颗卫星构成了空间部分的星群[5]。

GPS接收器利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，“如果我们知道离太空中一颗卫星的确切距离，就可以知道我们处于想象中的球体表面的某个位置，该球体的半径等于我们到卫星的距离。如果知道离两颗卫星的确切距离，即可知道我们位于两个球体相交线上的某个位置。如果再有来自第三颗、第四颗卫星的测量，就可以发现我们的位置。GPS接收器处理卫星范围内的测量数据，并得出它的位置”[6]。

全球定位系统有大量的应用，包括军事的和民用的。大多数人熟悉它的军事使用，军方用它在陆地上、海上和空中导航。它也用在武器系统上，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。它甚至用于大量个人活动中，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，这个过程从决定我们必须建造什么开始。第一步是阅读顾客提供给我们的所有陈述问题或需求的文档。对于系统来说，我们已经得到了一份有关高层次需求和约束的愿景陈述。

愿景：

- 为顾客提供高效的、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；
- 提供足够精度以支持当前和未来的用户需要；
- 提供关键系统能力的功能冗余；
- 广泛的自动化，使操作成本最小化；
- 容易维护，使维护成本最小化；
- 可扩展，支持系统功能的升级；
- 服务生命期长，特别是太空部分的元件。

约束：

- 兼容国际标准；
- 充分利用商用现货（COTS）硬件和软件。

显然，这是一个高度简化的需求陈述，但它确实为基于卫星的导航系统提供了非常基本的规格说明。在实践中，像这么大一个系统，只有在展示了解决方案的可行性，经过几百个人月的分析，在众多领域专家和系统最终用户及客户参与之后，才能得到它的详细需求。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），不仅阐明了系统的一般行为，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

虽然只是最少的需求和约束，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地理解SNS在环境中必须提供的功能。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。依赖箭头展示了是外部实体依赖于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）等执行者依赖于SNS，分别使用它的导航信息、对它进行操作和维护。虽然卫星导航系统有能力提供它自己的电力作为地面系统的备份，但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，ExternalCommunica tions（外部通信）执行者为SNS提供购买的通信服务，有时候作为首要服务，有时候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）”，以便清晰地把它们和内部的系统电源及通信服务分开。

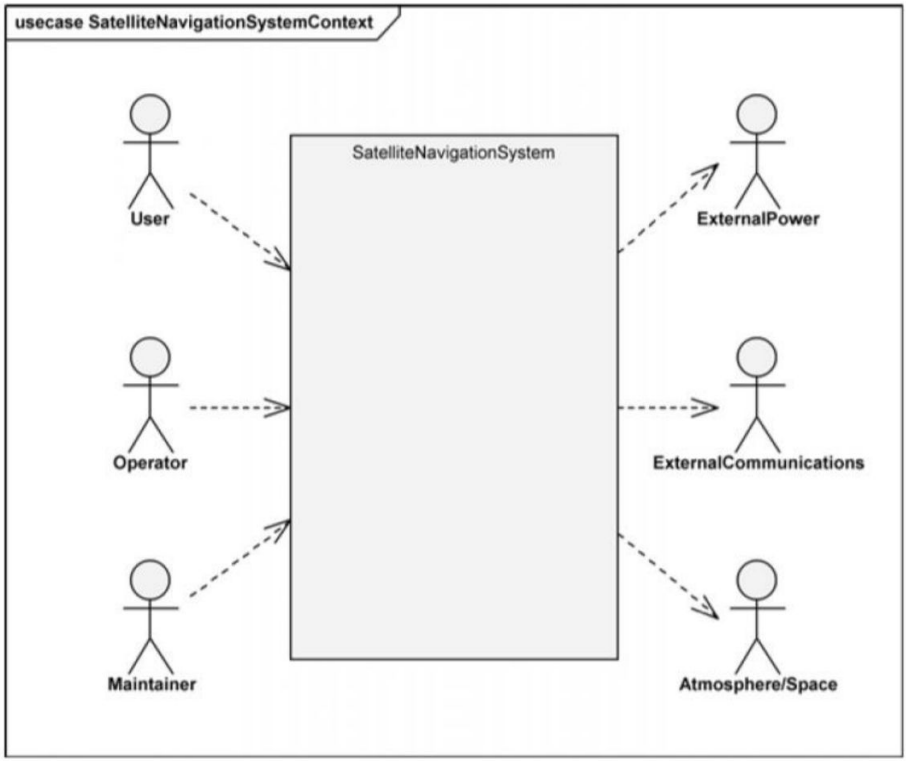


图8-1　卫星导航系统上下文图

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部分和太空部分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，另一种方法是从约束“兼容国际标准”的视角来看。许多国家的、国际的法规及条约管制着卫星传输，因此，我们有重要的理由来详细说明这个执行者。

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。这样，我们可以从特定涉众（即顾客）的观点来看问题，他们关注的是为用户提供导航信息的系统。顾客关注的不是运营SNS的庞大的企业，这与User执行者不同。User执行者可能把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，我们要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SatelliteNavigationSystem包的边界之内。

我们曾看到过许多表示上下文图的变体，有些非常精致，有些非常简单。比较精致的通常提供执行者和正在开发的系统之间双向流动信息的细节。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替代品，那么这种类型的信息在开发周期的早期就已了解，所以一些开发团队选择在此时表现它。

细节不太重要，重要的是开发团队选择一种风格，形成文档，遵循它，以保证其清晰和容易理解。但我们更喜欢展示一张上下文图，因为它简单、清楚地传达了高层次的概念，即系统是功能的容器，与外部环境中的实体实现交互。在这些交互中，系统为一些实体提供服务，从另一些实体接收服务。在开发开始时，理解这一点是至关重要的。

除了功能需求，我们还有高层次的非功能需求，它们适用于部分功能或整个系统。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。我们在一个名为“补充规格说明”的文本文档中维护非功能需求和设计约束，它也用于维护适用于多于一个用例的功能需求。还有一个关键文档必须从此时开始维护，即词汇表。开发团队在术语定义上取得一致，然后照此使用，这是非常重要的。

即使从这些高度概括的系统需求中，也可以对开发卫星导航系统的过程给出两点意见：

（1）架构必须能够随时间演进；

（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个和第二个层次的架构，分别定义组成部分和子系统。通过对Operator执行者使用的功能进行逻辑上的划分，我们设计出这两个层次的架构。正如在本章的介绍中所说的，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任务级用例的容器（在UML中就是包），这些用例将定义卫星导航系统必须提供的功能。这些任务用例包为我们提供了SNS高层次功能的上下文，如图8-2所示。这些包中包含了一些任务用例，展示了SNS的用户、操作员和维护人员如何与系统进行交互，以完成其任务。由于我们是在利用面向对象分析设计技术和UML 2.0表示法来执行一个系统工程任务，而不是一个软件工程任务，所以我们在图8-2中使用的表示法可能让你觉得有点陌生。但是，我们相信它清晰地展示了所需信息，并确保了可理解性。

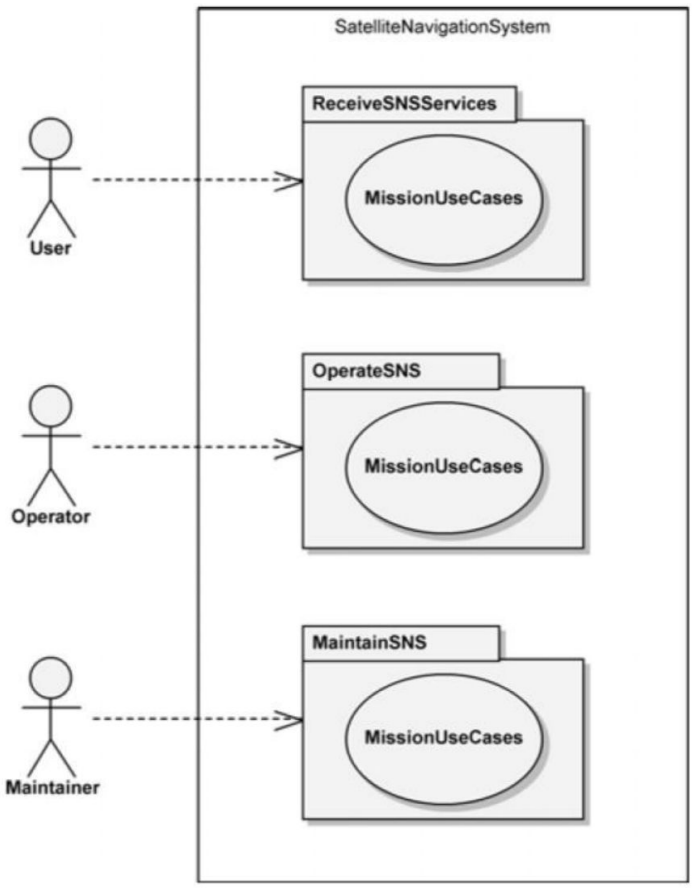


图8-2　SNS任务用例包

8.1.3 确定任务用例

系统的愿景陈述相当开放：“为顾客提供高效的、买得起的卫星导航系统服务”。因此，架构师需要明智地精简问题空间，使问题可以解决。类似这样的问题容易遭受过度分析之苦，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的任务用例开始。

像这样的大项目，通常围绕某个位于中心的小团队来组织，他们负责建立总体系统架构，并将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，系统架构师也常有一些概念模型，对实现元素进行划分。基于建造、操作和维护基于卫星的系统的经验，我们相信最高层次的逻辑架构包括4个部分：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从某些时间点开始约束设计空间。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，该架构展示了一些复杂的对象，如地面部分和卫星部分，每一个对象将执行系统的一个主要功能。这正如在第4章中讨论的：大系统中，最高层次的抽象对象通常沿着主要的系统功能线聚集。如何识别和细化这些对象，在分析阶段和设计阶段基本上没有区别。

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主要任务用例。说“甚至在这之前”是因为，即使我们有了SNS架构的概念，也应该从黑盒视角开始我们的分析，这样就不会给架构带来不必要的约束。也就是说，首先分析所需的功能，以决定SNS的任务用例，而不是针对单个SNS部分。然后，我们以白盒的方式来看卫星导航系统，将这个用例功能分配到某个部分。

Rumbaugh、Jacobson和Booch在他们所著的*Unified Modeling Language Reference Manual*中指出，“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]。[3]因此，活动图是我们分析任务用例、阐述预期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，我们只关注任务用例的成功场景，而把众多的备选场景留到以后。

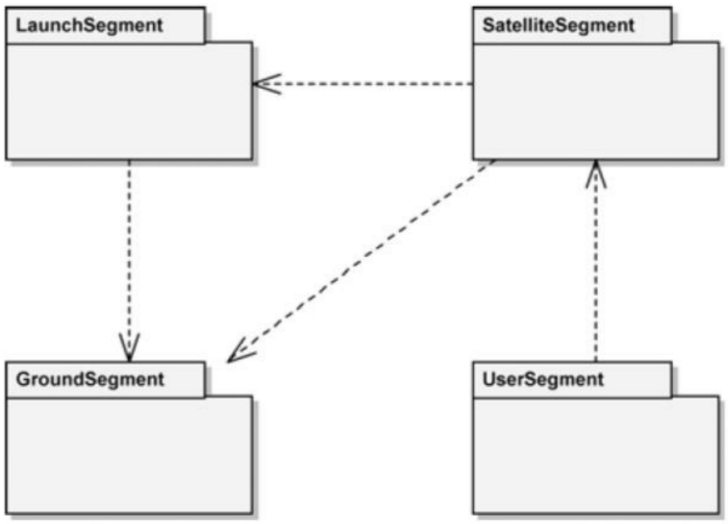


图8-3　SNS的逻辑架构

读者可能不熟悉“成功场景（success scenario）”这个术语，众所周知的ATM例子帮我们解释了这个术语。ATM的一个用例是Withdraw Cash（提取现金），这通常就是我们在这些机器上想做的事。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（以及所有用例）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，“成功场景”也称为“主要场景”。备选或次要场景通常处理从主要场景分支的情况。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说明。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，之后再跳回主要场景。

我们希望这个解释已经澄清了概念，而不是让你更糊涂。但还要说一点，对于实时系统，如卫星导航系统，要明白它的许多功能是在次要场景中体现的。这些功能可认为是冰山在水面下的一部分，然而它们对系统操作的完整性和安全性非常关键。隐藏在幕后的次要场景通常得到的关注较少，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，我们在实际的系统开发工作中必须考虑。

现在，回到手上的任务，开发OperateSNS任务用例包中的任务用例。基于对SNS总体操作的分析，我们确定了四个相应的任务用例：

- 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）

这些任务用例的规格说明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，模拟和原型通常是这个分析过程中有价值的工具。但是，我们的分析通常会用活动图来建模，在开发下一小节的系统用例时，我们正是这样做的。图8-4描绘了开发OperateSNS任务用例包中的任务用例时，我们的分析结果。在本章剩下的部分，我们的工作将集中于分析Initialize Operations任务用例，以确定系统必须执行哪些活动，才能让操作员能够初始化卫星导航系统的操作。

8.1.4 确定系统用例

正如之前所说的，我们为Initialize Operations任务用例的功能绘制了一张活动图，以确定这个封装的系统用例。在绘制这张活动图时，我们没有试图利用SNS的组成部分的概念（参见之前的图8-3）。这样做有一个理由：我们不希望预设手上问题可能的架构解决方案，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，因此只能看到它提供什么服务，而看不到它如何提供服务。在分析系统的高层执行行为时，我们对操作员和卫星导航系统之间跨越边界的控制流感兴趣。

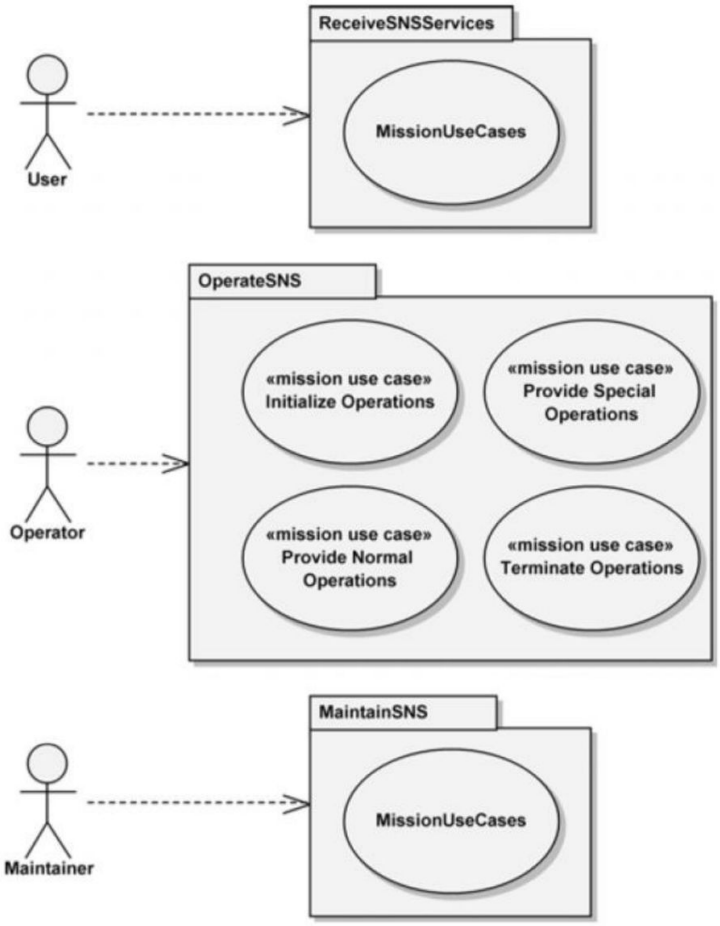


图8-4　细化OperateSNS任务用例包

既然我们关心SNS执行的活动，而不是通信或序列图里表达的消息传送，那么活动图就是比较简单的。如果要定义整个SNS的系统活动，我们就要基于活动图，对它的每个任务用例进行分析，以发现卫星导航系统为了满足需求而必须执行的众多活动。请试想一下，为了操作这样一个系统，每天24小时必须执行的所有活动。但是，这里我们关注的是Initialize Operations任务用例，图8-5是我们为它绘制的活动图。

根据这个活动图，通过系统工程经验来判断，我们得到了各自的系统用例列表。例如，我们决定把Prepare for Launch（准备发射）和Launch（反射）这两个动作放进系统用例Launch Satellite（发射卫星）之中。剩下的动作体现了重要的系统功能，因此我们决定每一个动作应该代表一个系统用例，从而得到了Initialize Operations任务用例的系统用例，如表8-1所示。

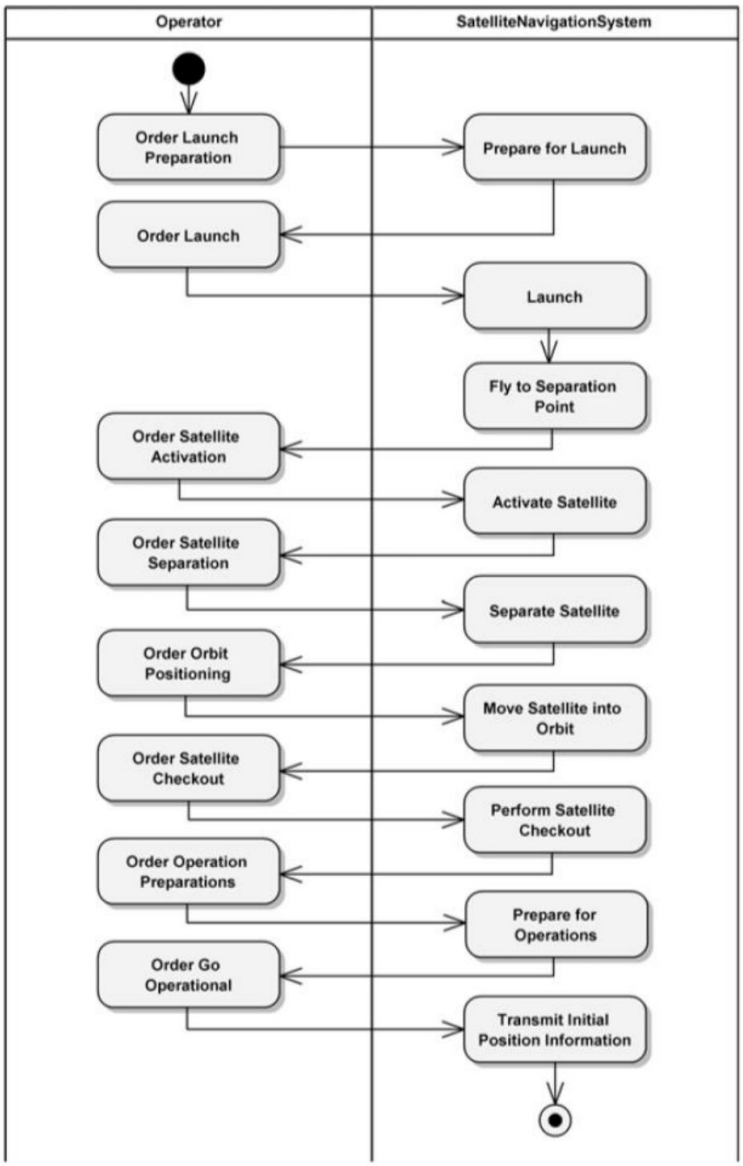


图8-5　Initialize Operations的黑盒活动图

表8-1　Initialize　Operations的系统用例

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，并执行发射
Fly to Separation Point	发射器飞行到卫星载荷即将分离的点。这涉及多级发射器的使用和分离
Activate Satellite	激活卫星，准备从发射器部署它
Separate Satellite	从发射器部署卫星
Move Satellite into Orbit	利用卫星总线推进能力将卫星定位在正确的轨道平面内
Perform Satellite Checkout	对卫星的能力执行在轨检查
Prepare for Operations	在开始操作之前执行最后的准备
Transmit Initial Position Information	开始操作并向 SNS 用户传输初始位置信息

图8-6是更新过的用例图，展示了表8-1中的系统用例。在这里，我们使用了InitializeOperations包来容纳从Initialize Operations任务用例得

到的系统用例。其他三个任务用例包含了操作SNS的功能，带有关键词标签«mission use case»。我们认为这种建模方法有用而且清楚，但是，每个开发团队都需要确定所选择的技术，并用文档记录下来。

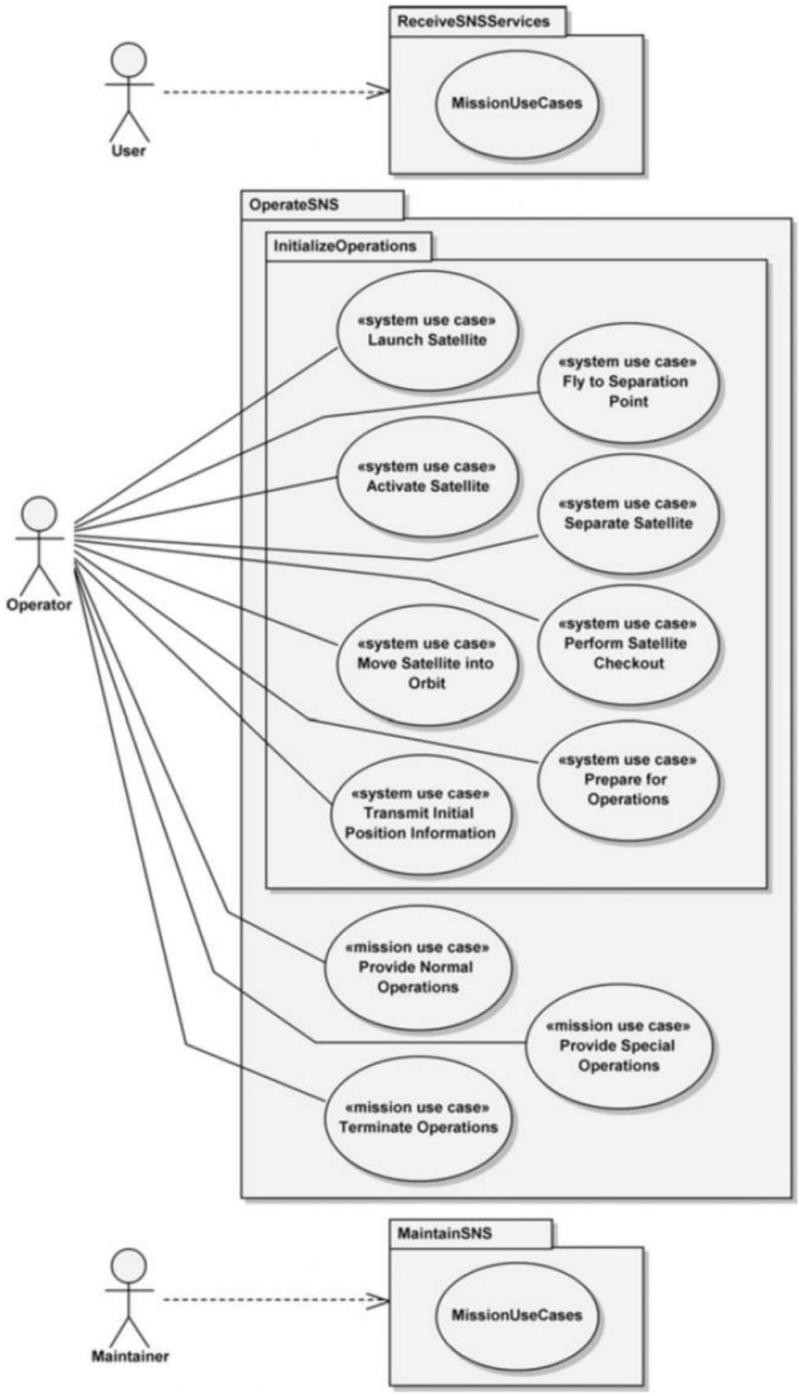


图8-6　Initialize Operations的系统用例

8.2 细化

接下来讨论系统架构，它奠定了一个基础，以便实现前一节得到的系统用例中所包含的需求。在头两个子小节中，我们引入了两个架构问题：开发好架构时的关注点，以及开发架构时执行的活动。

这个讨论将我们带入下面两个子小节（验证所建议的系统架构，分配非功能需求和确定接口），在那里我们执行SNS系统架构的宏观层分析。我们的目标是在分析部分并确定它们的协作子系统的架构之前，验证所建议的SNS架构。我们利用前面一样的用例分析技术来开发系统用例，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时特别有效，但实际将系统用例功能分配到每个部分时，这是无效的。

完成制作行为原型之后，我们将在下一子小节中规定SNS的架构及其部署。然后，我们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它的部分，以及部分所包含的子系统。

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，不幸的是，另一些方法却极其愚蠢。如何知道好架构和糟糕架构之间的不同？

好的架构通常表现出面向对象的特征。当然，这不意味着只要使用面向对象技术，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，得到的架构通常表现出有组织的复杂性的特点，这正是我们希望的。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，通过定义良好的、受控的接口来提供功能，在同样定义良好的、受控的低层抽象设施上建造。
- 每一层的接口和实现之间有清晰的关注点分离，使得改变一层的实现不会违反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，我们必须和它的所有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组开发活动，它们在系统内的每个抽象层次上执行。这些活动从总体上确定了为卫星导航系统开发系统架构所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。

- 识别给定抽象层次的架构元素，目的是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，然后细化它们，准备下一个抽象层次的分析。

创建架构描述

在第6章的补充材料“用文档记录软件架构”中，我们解释了用文档记录系统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，即IEEE推荐的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，但Maier、Emery和Hilliard[9]认为“它一样适用于所有系统，因此适合作为系统工程的一部分来描述系统架构”。此外，他们还说，“ANSI/IEEE 1471-2000在系统工程中可能有一个特别重要的应用，即调和现在流行起来的各种架构框架”[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]。

这组活动使得开发SNS架构时的首要关注点十分清晰，即定义SNS架构的元素（部分和子系统）、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，而不是顺序执行的。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。在接下来的子小节中，我们将定义卫星导航系统的部分、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

我们建议让系统架构师有机会实验备选的系统分解方式，这样我们就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继续使用。

在这个子小节和下一个子小节，我们将完成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。我们希望确保现在就发现架构的所有问题，而不是以后。在开发生命周期早期，容纳需求变更比较简单，代价比较低，同样的道理，架构变更也是如此。我们主要关注Initialize Operations的功能，因为在之前的开发中，这曾经是一个有问题的区域。

这里必须特别提醒一下：虽然我们在本小节和下一个小节使用的基本技术与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非常不同的。我们关注的是快速钻过若干架构层次，目的是针对Initialize Operations任务用例，研究更多的系统关注点。我们在这里获得的模型不会作为真实系统架构规格说明的一部分。

初始的架构决策在很多时候都没有经过验证，因为架构团队没有想到这一步的效用，或者因为开发往前推进的速度太快了。一般的情况是，团队常常立即开始分解系统用例，得到开发部分用例，再将部分用例分派给部分架构团队。然后，部分架构团队重复该过程来定义子系统用例。最后，架构团队可能试图穿过各个架构层次重新组织这些用例，以确定是否在每个架构层次上，每件事都能够融为一体。不幸的是，如果不能融为一体，就太迟了。在这么晚的阶段，任何修正可能都要困难得多，而且耗时耗钱。因此，在开发部分用例之前执行宏观分析是有好处的。正是基于同样的理由，对部分架构团队来说，在开始分析时，采用同样的过程也应该是必要的。

第一步是复查之前的工作结果，评估我们所处的位置，计划前面的道路。我们和领域专家一起，从功能和非功能的角度来评估SNS的逻辑架构（参见之前的图8-3），以及Initialize Operations任务用例（参见之前的图8-5）的黑盒活动图。我们相信，我们已经正确地捕获了功能，但对一些非功能需求还不能确定。我们需要在关键的系统能力上保证SNS的功能冗余。在系统架构的这个层次，我们列出部分之间的结构关系，但不设计部分的内部架构。因此，在这里我们必须确保跨越各部分的冗余。

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定。首先，对于SNS地面部分的关键任务设备，我们有备份硬件。这个设备将在热切换模式下运行，在同一时刻，首要和备份设备

都是活动的。如果出现导致首要设备不能执行任务的事件，备份设备就能够快速取代首要设备。其次，对于SNS发射部分，我们将使用同样的热切换设备方法来保证冗余功能。对于卫星部分，我们将利用这个事实：星群内的多个卫星是冗余的，还有在轨或准备发射的空闲卫星。除此之外，设计人员必须在每个卫星内部提供功能冗余，以满足SNS的非功能需求。对于用户部分，通过简单地提供整个用户部分的替代品来满足功能冗余的需求。就像卫星部分一样，这个部分的设计人员需要尽可能适当地关注内部的部分冗余。

从之前图8-5展示的Initialize Operations的黑盒活动图开始，我们将SatelliteNavigationSystem分区展示的系统功能分配到一个或多个组成部分中：地面、发射、卫星或用户。我们的目标是将部分用例分配到每个部分，这些部分用例来自系统用例。这样，我们可以看到SNS功能由其部分通过协作来提供。如果适当地分配用例，单个部分将展示如下的面向对象核心原则。

- 抽象：相对于查看者的视角，部分提供了清晰定义的概念边界。
- 封装：部分隔离它们的子系统，子系统提供结构和行为。部分对于其他部分是黑盒。
- 模块化：部分被组织成一个高内聚、松耦合的子系统的集合。
- 层次：部分展示了分级或排序的抽象。

对SNS这样复杂的系统，要完成任何合理级别的细节，这部分的分析过程很容易就会花掉几个月。[4]这也是一个理由，导致我们强烈建议先验证架构决定，即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。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的、对架构有意义的用例。

在浏览图8-5所展示的每个动作时，我们必须不断问自己一些问题。哪个或哪些部分应该负责某个动作？某个部分是否有足够知识来实施分配给它的动作，还是必须委托该行为?该部分是否试图做太多的事情?它正在执行的动作是否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?

这些问题和我们在执行软件工程时问自己的问题如此相似，很有趣吧?还记得本章介绍里所说的吗？

“本书前面的部分展示的面向对象分析设计原则和过程，以及在第5章讨论的UML 2.0表示法，既适用于最高层次的系统架构开发，也适用于软件开发。”

之前，我们对SNS系统级别功能进行了黑盒分析。现在，我们聚焦于SNS内部的结构——组成这个系统的部分以及协作时每个必须提供的功能，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进行白盒分析，我们可以完成这项任务。我们深入SNS，确定它如何提供所需的服务。

在继续进行之前，有些地方要注意。例如，“应该使用Proteus-4发射器”的约束当然会影响我们的功能责任分配。沿着这个思路进一步考虑，当卫星部分的设计团队在设计这一部分的架构时，会受到一些约束的影响，如“应该使用Gamma II（B）卫星总线”。[5]在补充材料“功能分配”中，我们将进一步讨论这一点。

功能分配

最终，我们必须将系统需求转换为卫星导航系统硬件、软件和手工操作等方面的需求，这样拥有不同技能的不同组织可以并行地解决特定部分的问题。在这些工作期间，架构团队总是在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层次上完成——从系统架构的最高层到最低层。以下列表提供了例子来阐明这个断言。

- 系统：我们可以将整个卫星导航系统的功能分配给另一个开发组。例如，可以寻求与开发Galileo的欧洲太空总署合作，共同开发。
- 部分：在部分层次上分配的一个主要例子就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部分，而只是与分包商团队一起定义发射部分的接口。
- 子系统[6]：我们可以想象SNS在这个层次上的分配，包括在卫星部分的开发中利用商业上可采购的卫星总线子系统。
- 组件：这个层次位于SNS架构内，在这个层次，最有可能将需求分配到硬件、软件或手工操作。例如，用户部分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（用于控制用户部分）。

之前，表8-1中列出了8个SNS系统用例，我们从图8-5的SatelliteNavigation- System分区的动作中开发出这些用例。如果是实际开发系统架构，而不是进行快速浏览分析，我们会为每个用例开发各自的活动图，并在部分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，因为我们不打算在细节上挖得太深。我们关心的是部分上的用例功能的分配比例。

有了4个逻辑SNS部分（参见之前的图8-3）的实现，我们就开始和领域专家一起工作，分配动作中表示的功能。如果此时没有系统架构的概念，我们可能将功能分配到一些分区，并给分区取个通用的名字，如SegmentOne、SegmentTwo和SegmentThree。然后我们在每个分区中分配动作，就好像每个部分都是一个专家，提供一些紧密相关的能力，与其他部分协作，提供卫星导航系统的功能（在这里是初始化操作）。这种分配会在分析多个用例场景期间继续进行，以建立各部分更完整的功能画面，从而为每个部分选择有意义的名称提供支持。

正如第3章中讨论的，这个方法类似于我们设计良好的类的方法。在那一章里，我们说可以测量一个抽象的质量，并建议了5个测量指标，其中两个（耦合和内聚）是卫星导航系统架构关键抽象（即它的部分）的中心关注点。我们规定了SNS的部分，让它们是松耦合的。我们希望它们“独立”，只有必需的最小数目的连接，以支持它们的协作提供SNS的功能。

内聚是另一个测量指标，通过它也可以判断我们所选择的抽象的质量。内聚测量组成单个部分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，即完全不相关的抽象被扔进一个SNS部分之中。最受欢迎的内聚形式是功能内聚，即部分的所有元素一起工作，提供一些有良好边界的行为。

通过浏览如图8-5中所示的动作，可以看到如何得到如图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的系统用例Launch Satellite，我们获得了GroundSegment和LaunchSegment的若干动作。针对Initialize Operations包中余下的系统用例，我们继续这个分析过程，形成完整的活动图，如图8-7所示。

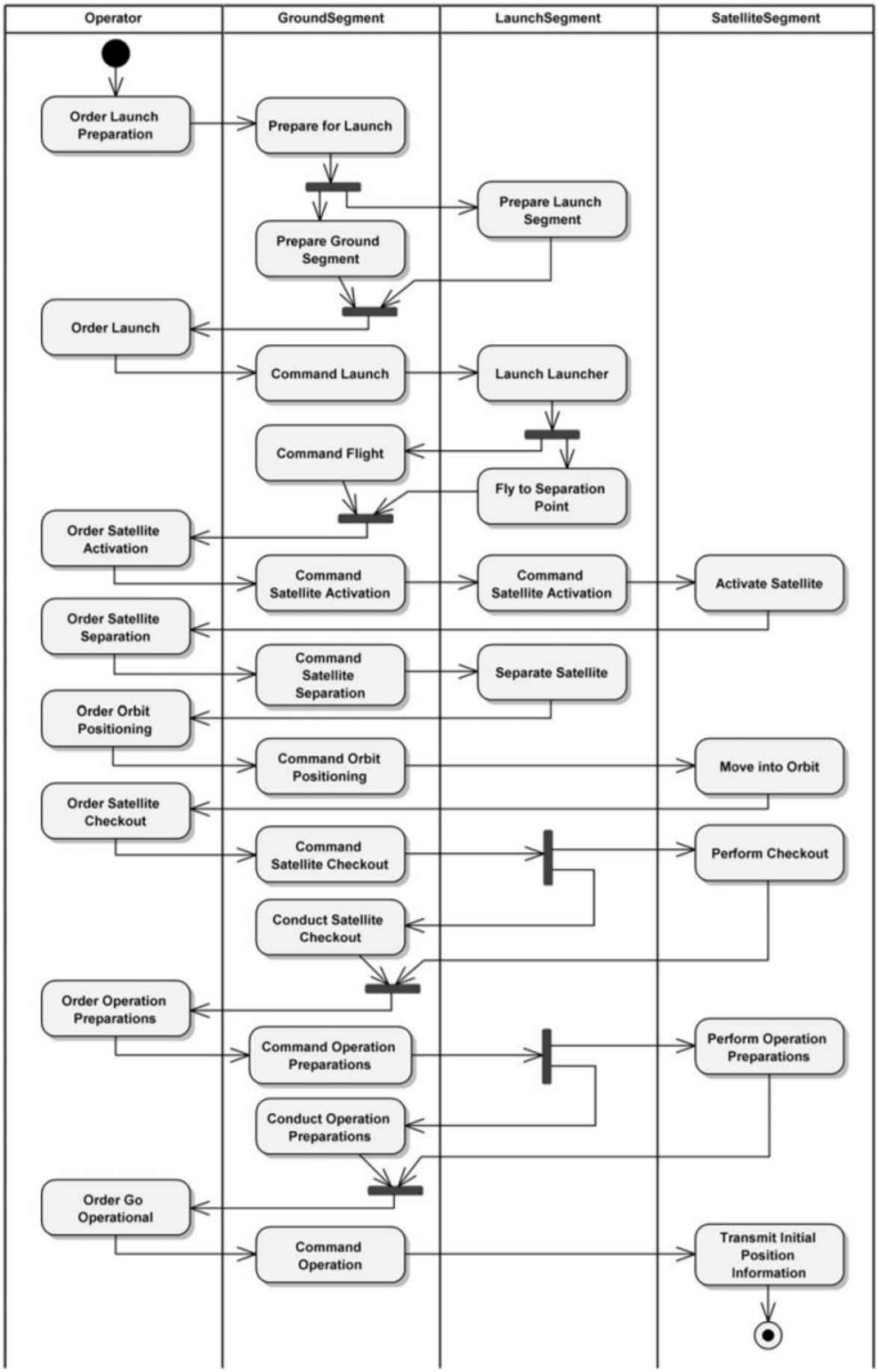


图8-7　Initialize Operations的白盒活动图

Initialize Operations的白盒活动图只展示了OperateSNS任务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的预备活动

以及所有后续发生的活动，它们包含在其他3个任务用例中：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术来明确这个行为，提供一个更完整的画面，说明部分如何共同操作，从而提供卫星导航系统的操作能力。

这种能力将包括预备的活动，例如，激活地面和发射部分，检查卫星的完整性，匹配卫星和发射器。除了这种能力之外，我们会发现地面部分在正常操作期间执行很多活动，包括以下一些：

- 持续监控和报告系统状态；
- 持续评估卫星飞行动力学并管理位置保持；
- 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）。

我们不应该在此时完成，因为图8-2和8-6的MaintainSNS和ReceiveSNSServices任务用例包中包含的系统功能也需要分析，才能完全地确定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的能力进行分析。

下一步是根据如图8-7所示的活动图，为每个SNS部分定义用例。我们一次聚焦于一个分区，并决定哪些动作包含合理的用例功能，是动作独立包含还是几个动作结合起来。让我们从GroundSegment分区开始。我们决定前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，在它的范围内是很重要的，因此我们定义名为Control Flight的单个用例，来封装它的行为。我们继续对整个GroundSegment分区执行这个方法，然后对LaunchSegment和SatelliteSegment分区也这样做。表8-2展示了针对图8-7的Initialize Operations白盒活动图，得到的部分用例及其组成动作。

表8-2　Initialize Operations部分用例

SNS 部分	部分用例	部分用例动作
GroundSegment	Control Launch（控制发射）	Prepare for Launch（准备发射）
		Prepare Ground Segment（准备地面部分）
		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）
	Command Satellite Checkout（命令卫星检测）	Command Satellite Checkout（命令卫星检测）
		Conduct Satellite Checkout（实施卫星检测）
	Conduct Operation Preparations（实施操作准备）	Command Operation Preparations（命令操作准备）
		Conduct Operation Preparations（实施操作准备）
	Command Operation（命令操作）	Command Operation（命令操作）
LaunchSegment	Launch（发射）	Prepare Launch Segment（准备发射部分）
		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 分配非功能需求和确定接口

在分析Initialize Operations的功能时，我们收到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？也许我们的顾客不希望用在轨的备用卫星替换失灵的卫星，[7]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，将如表8-2所示的用例以及所有附加用例分配到7天（168小时）时间里，如将卫星匹配到发射器，这样整个时间线跨度少于168小时，正如我们的顾客所要求的那样。

这个时候，一个合理的问题是：“我们如何做？”这里有两个主要问题：分配非功能需求和用文档记录结果。将性能需求（168小时）按适当比例分配到每个部分用例，非常依赖于领域专家的经验。除了使用领域专家和开发团队的经验之外，我们也使用其他技术，如通过模拟来确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部分用例。剩下的120小时被分配到所有预备活动，包括激活地面部分、激活发射部分、检查卫星完整性以及匹配卫星和发射器。

第二个问题是如何用文档记录结果，这在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记录结果，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，或者藏在活动属性框的tab下面。虽然这对运行报表和执行统计分析是有用的，但它不是我们喜欢的可视化表现方式，特别是在开发工作的这个层面。这里，我们特地选择用一张表（表8-3）来清晰地展示在部分用例之间分配48小时的结果。最终说明在其他非功能需求在部分用例之间的分配时，我们也使用同样的技术。

表8-3　Initialize Operations[a]发射时间分配

SNS 部分	部 分 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的时间。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。

分配到部分用例的这些非功能需求，在架构层次中下一个较低的层次，被分配到构成它的各个子系统用例中，使用的技术和在部分层次上一样。分配功能和非功能需求的技术可以从架构层次的一个层次到下一个层次递归地应用——从系统到部分，到它们的子系统，等等。

然后，我们可以讨论给定的设计约束间接引出的潜在需求：

- 兼容国际标准；
- 充分利用商用现货（COTS）硬件和软件。

正如之前所讨论的，“兼容国际标准”这一约束决定了外部执行者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部分通信的特定频率、传输位置信息的频率等。这意味着现在地面部分、发射部分（至少在飞行阶段期间）和卫星部分必须履行卫星导航系统的外部接口责任。我们指出

这些问题，是因为在关注功能时，约束（和非功能需求）可能会在开发周期中太晚的时候才被考虑，有时甚至被忽略。

外部接口问题是另一个还没有真正触及的方面，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。开发和用文档记录接口规格的技术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所示的那些执行者有接口，即User、Operator、Maintainer、ExternalPower、External- Communications和Atmosphere/Space。显然，在着手确定User、Operator和Maintainer执行者接口的规格之前，必须进行一些层次上的功能分析。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和机构通过管制卫星传输的法规和条约来指定。

8.2.5 规定系统架构及其部署

之前在图8-3中展示的SNS逻辑架构的概念经受住了行为原型工作的测试。因此，图8-3代表SNS架构的第一层逻辑视图，就如图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层次的抽象，即它的部分和它们的关系。图8-8阐明了表达部分之间接口的两种方法：小球和球窝表示法（LaunchSupport接口）、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中的SNS接口没有考虑到3个系统执行者：ExternalPower、ExternalCommunications和Atmosphere/Space。这些执行者为卫星导航系统提供重要的服务，但在开发逻辑架构时，它们不是我们关注的中心。

图8-9展示了图8-8中的组件部署到系统架构节点的情况。这些组件是SNS的部分，表现为UML 2.0工件。我们承认，这不是这种表示法的典型用法。通常，我们在处理节点上部署软件工件（如代码、源文件、文档或另一个与代码相关的东西）。但是，这张图清楚地展现了信息。在系统工程中使用UML 2.0表示法时，一些非标准的用法是不可避免的。Operator、Maintainer和User等执行者与卫星导航系统之间实现交互的接口包含在它的各部分中，所以我们选择通过依赖来阐明这些关系。

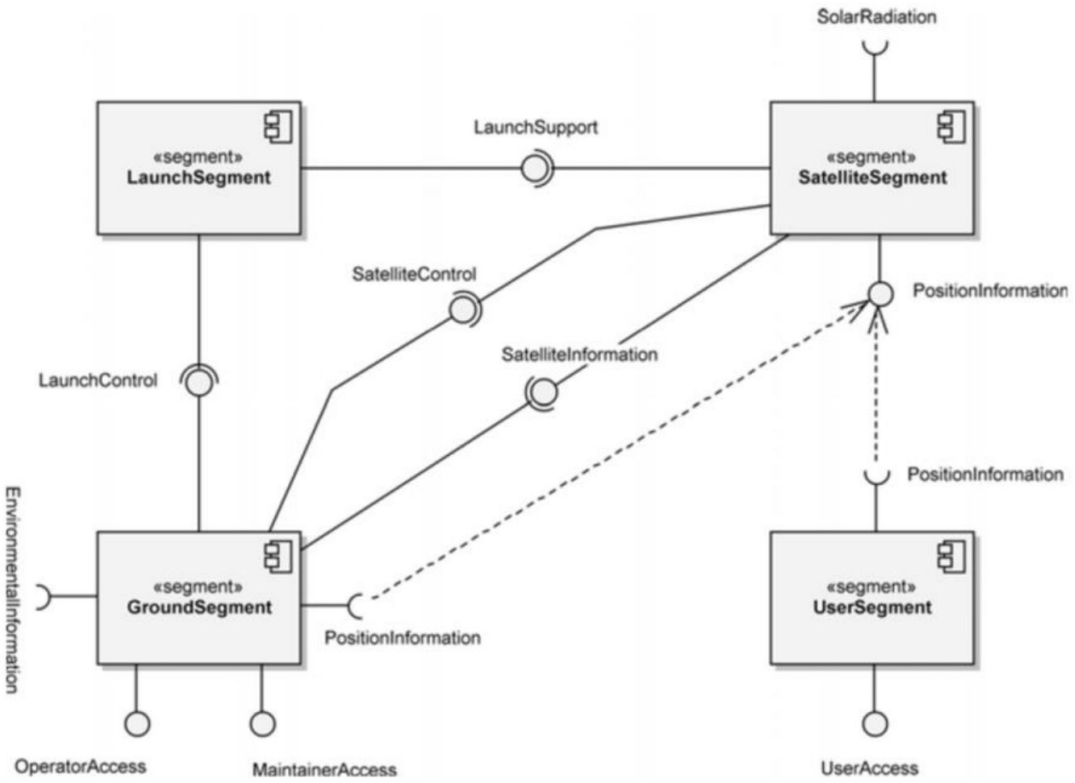


图8-8　卫星导航系统组件图

之前，我们确定了提供功能冗余的方法，即在GroundSegment和LaunchSegment上通过热切换模式，为任务关键的设备运行备份，首要设备和备份设备同时是活跃的。开发SNS架构时，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放置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丧失部分的功能。这个设计决策表示为GroundSite和LaunchSite节点之间多重性为2的通信关联。类似于原来建议的备份设备，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。SatelliteConstellation节点本质上是卫星导航系统的一组卫星和它们在太空中的位置的集合，它们向UserSegment工件提供位置信息。SatelliteConstellation节点为驻留在它上面的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?），以帮助保持卫星处于适当的轨道。它也提供了大气和外层空间，作为卫星通信的介质。SatelliteSegment工件上的多重性{1..*}表示该星群至少有一个卫星。我们的顾客还没有确定卫星导航系统的覆盖面积。[8]如果解决了这个问题，我们将确定卫星必需的确切数目，以便为SNS用户提供“高效的、买得起的卫星导航系统服务”。

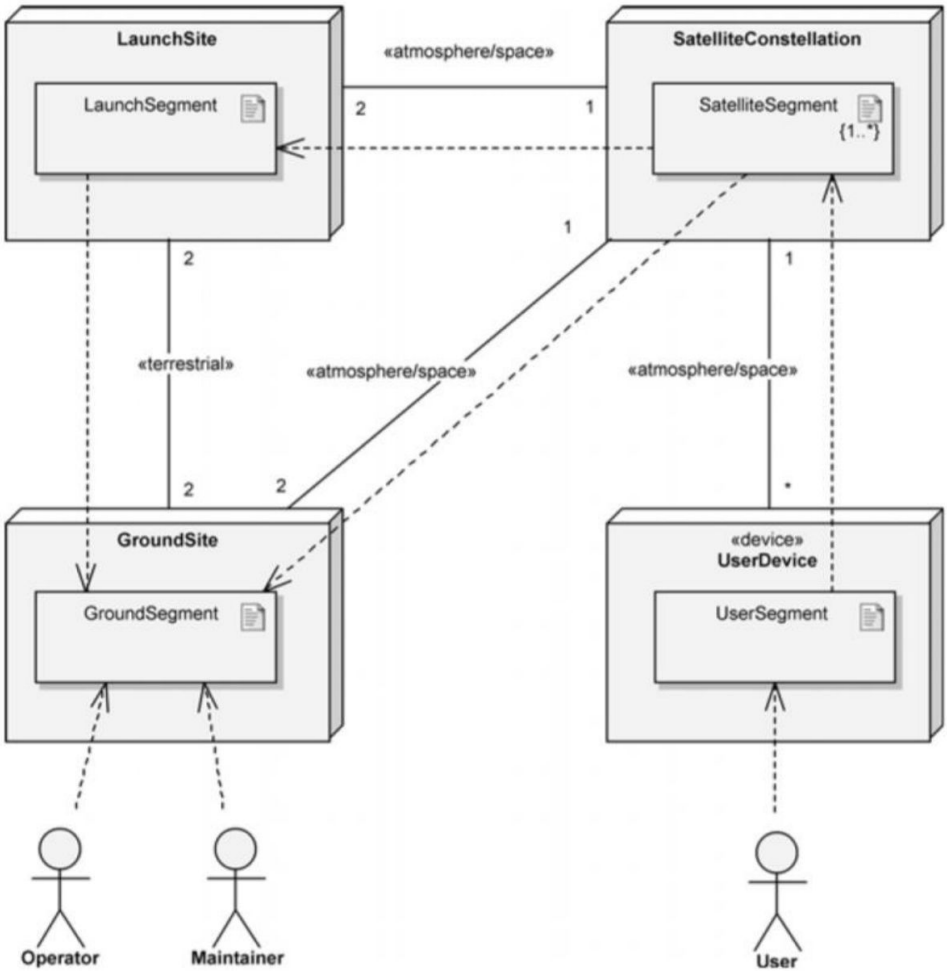


图8-9　部署SNS部分

8.2.6　分解系统架构

既然我们已经针对Initialize Operations系统用例，验证了卫星导航系统架构相关的假设和决策，接下来就可以进行它的各部分及其子系统的规格说明了。如果之前我们曾面临任何问题，应该早已根据需要修改了架构。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。

之前如图8-3所示的SNS逻辑架构图很有用，但不完整，因为这张图中的各部分太大了，不能由小的开发人员团队开发。必须进入到各个部分，进一步将它们分解为嵌套的子系统。这可以通过应用同样的分析技术完成。在针对Initialize Operations功能为卫星导航系统架构的各部分制作原型时（如图8-8组件图所描绘的），我们曾使用过这些技术，但这一次应用得更完整。这些技术在卫星导航系统所有的抽象层

次上重复（从系统到部分、到它们的子系统，等等），为系统架构中每一个层次上的每个元素确定用例。在我们做这件事情时，非功能需求在各个用例上分配，定位到系统分解的每一个层次、每一个元素上。为完整起见，我们的分析技术展示如下。

（1）对每个系统用例执行黑盒分析，以决定它的动作。

（2）对这些系统动作执行白盒分析，将它们分配到各个部分上。

（3）通过这些已分配的系统动作来确定部分用例。

（4）对每个部分用例执行黑盒分析，以决定它的动作。

（5）对这些部分动作执行白盒分析，将它们分配到各个子系统上。

（6）通过这些已分配的部分动作来确定子系统用例。

我们在补充材料“相似的架构分析技术”里，提供了使用黑盒和白盒系统分析方法的观点。

正常情况下，在应用分析技术时，我们将完成每一步，跨越系统的整个架构层次，然后再进行下一步。换句话说，在进行步骤（2）之前，在步骤（1）中对所有系统用例进行黑盒分析；然后在进行步骤（3）之前，对所有系统动作进行白盒分析，等等。

但是，由于本书篇幅的限制，分解系统架构的例子仍然只聚焦于整个系统的一部分，即Launch Satellite系统用例。我们特别说明这一点，这样你就不会在阅读了我们的执行步骤（在下面列出）之后，误认为应该垂直地向下钻探，从一个系统用例到系统活动、到部分用例、到部分活动、到子系统，等等。然后再对下一个系统用例重复。那样做将是一个低效的方法。以上编号的步骤将在系统的每个架构层次上水平地应用，以提供一个完整的整体系统视图，它可以在过程中的任意时刻进行验证。

相似的架构分析技术

多年以来，系统工程师使用了和我们的描述非常相似的技术，来分析系统功能，并将功能分配到系统架构的各元素上。这些黑盒和白盒分析技术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*一书中，通过上级和下级用例的概念，展示了用例及其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进行系统工程的扩展，在本质上结合了这些方法。多年以来，系统工程师一直在高效地使用用例和黑盒/白盒分析概念。

我们通过执行以下活动，继续我们的分析（不在这里展示）。

- 针对Launch Satellite系统用例进行黑盒分析，以决定它的动作；
- 对这些系统动作进行白盒分析，将它们分配到各个部分上；
- 通过这些已分配的系统动作来确定GroundSegment的部分用例；
- 针对GroundSegment的Control Launch用例进行黑盒分析，以决定它的动作；
- 对GroundSegment的Control Launch动作进行白盒分析，将它们分配到各个子系统上。

图8-10展示了这个分析的结果。我们可以看到，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部分的架构。下面将展示得到的部分架构。

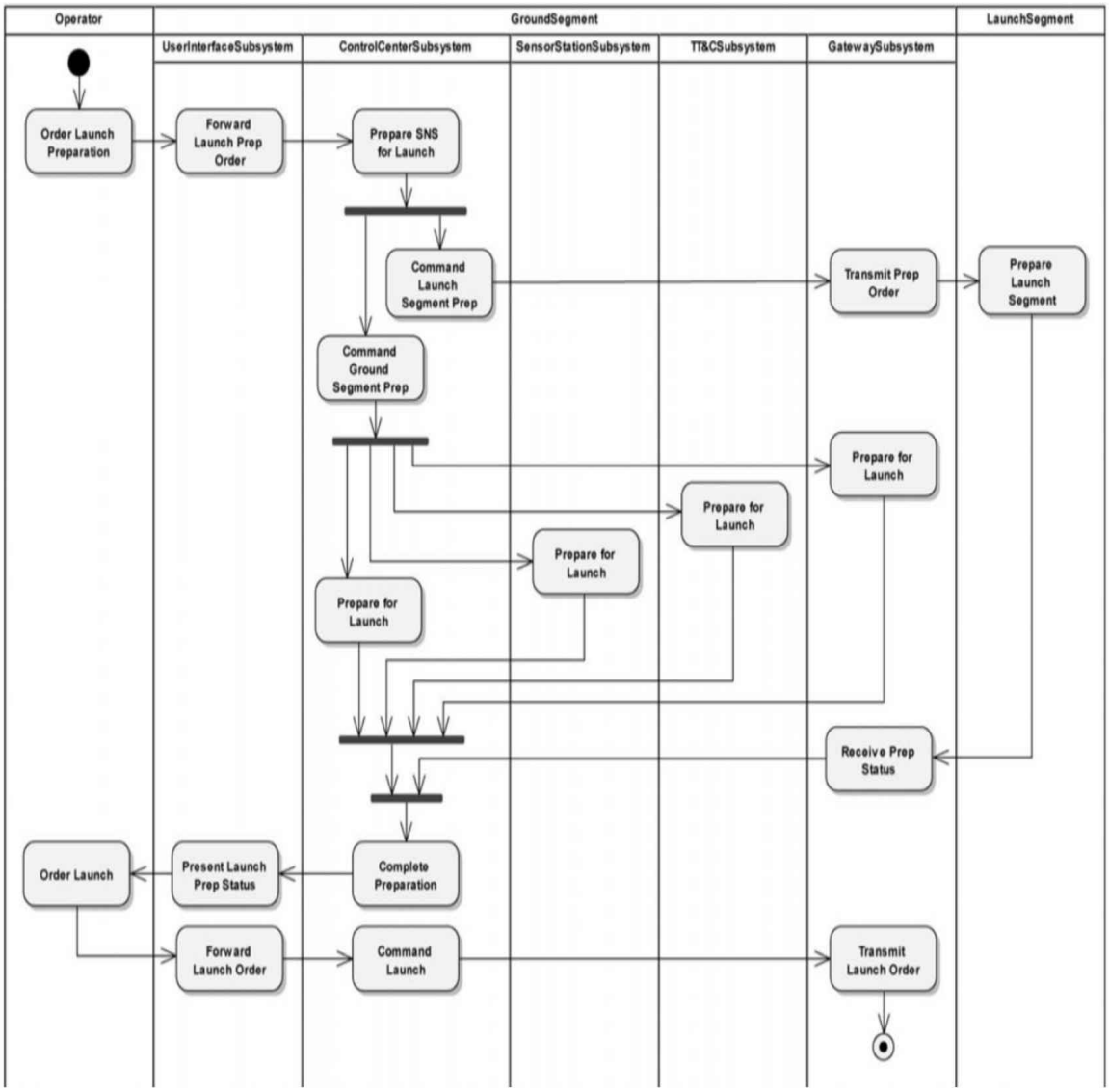


图8-10　Control Launch白盒活动图

如图8-11所示，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）。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，以便与LaunchSegment及SatelliteSegment通信，分别控制发射活动和卫星操作。最后，UserInterface子系统让Operator和Maintainer执行者能够访问GroundSegment的功能。

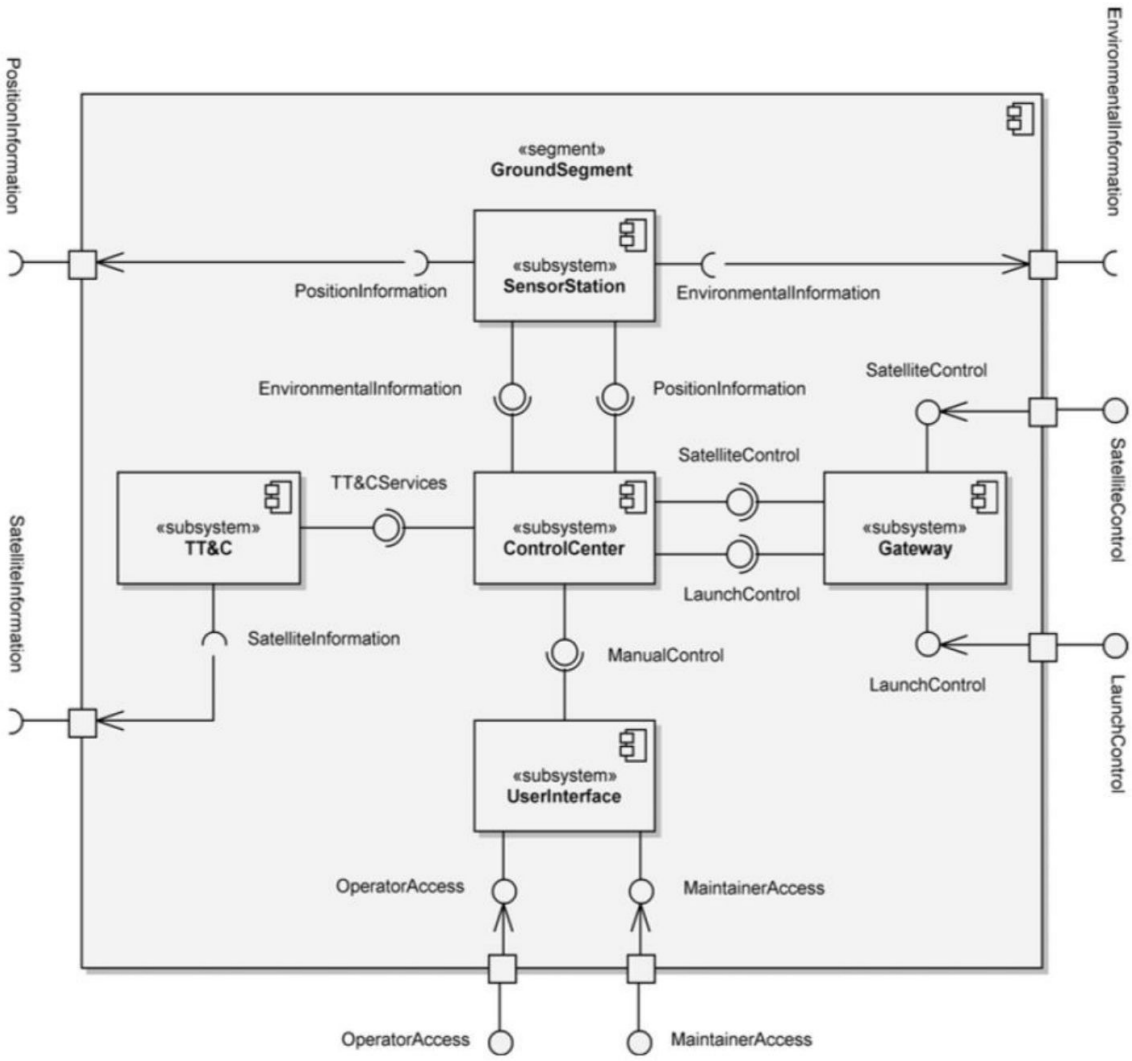


图8-11　GroundSegment逻辑架构

图8-12展示了LaunchSegment的逻辑架构，由三个子系统组成：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，让SatelliteSegment进入它的初始轨道。在这里，Gateway子系统就像在GroundSegment里一样，让LaunchCenter接收来自GroundSegment的发射控制支持，并向Launcher提供发射支持。

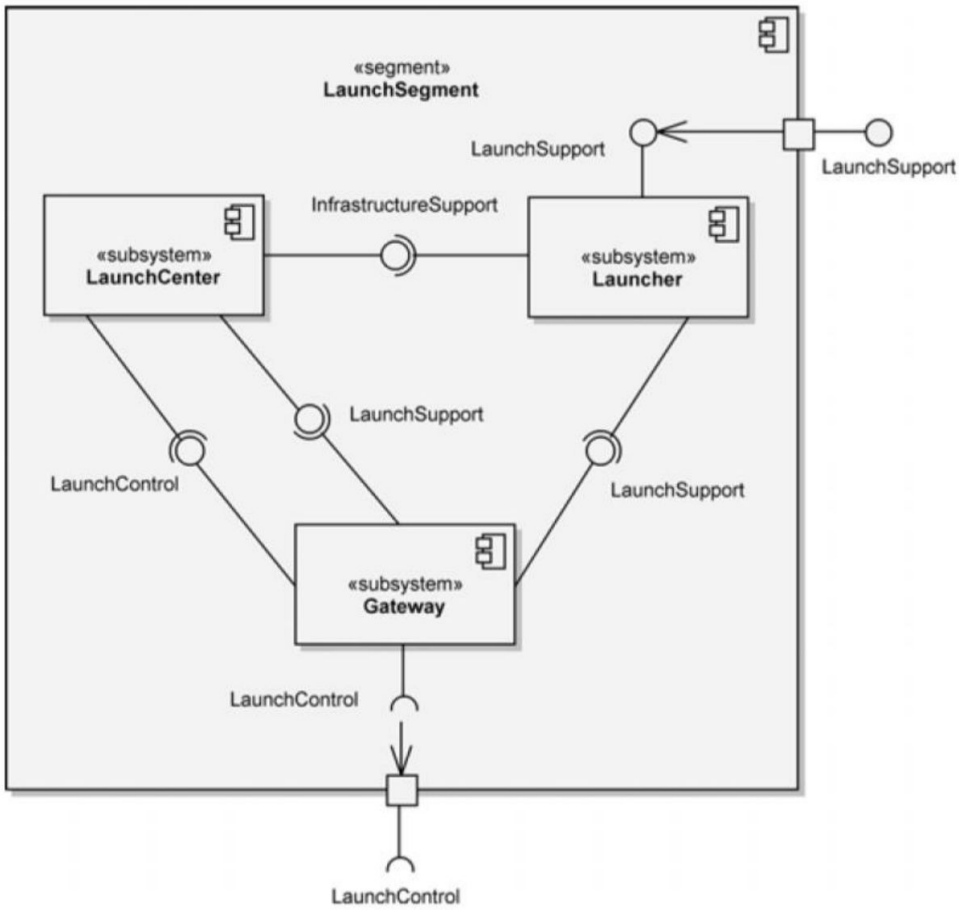


图8-12　LaunchSegment的逻辑架构

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上挂接的设备提供了电源、姿态控制和推进等诸多服务。这个设备让NavigationPayload设备（包括一个高精度时钟和位置信号生成）能够向卫星导航系统用户提供位置信息。

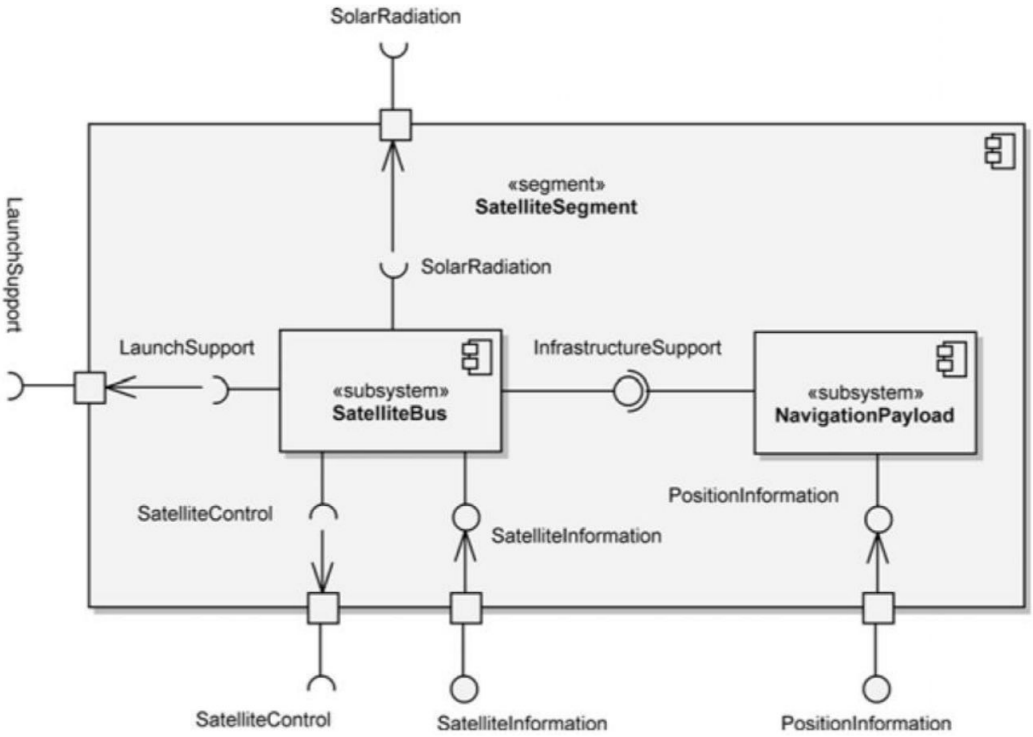


图8-13　SatelliteSegment的逻辑架构

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，通过多种特殊的用户界面（如按钮、触摸屏和声音警告）来访问和使用UserSegment的导航服务。

这个设计带给我们4个顶层的部分，每个部分包含若干子系统。我们已经将系统功能分配给了这些子系统，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，这些部分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。每个部分或子系统应该由一个组织、团队或个人所拥有，但可能由更多人实现。拥有者指导元素的详细设计和实现，并管理它与同一抽象层次上其他元素的接口。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。

现在我们已经实现了本章的目标——我们展示了面向对象分析设计的原则和过程以及UML 2.0表示法。它们既可以应用于开发最高层次的系统架构，也可以应用于软件开发。

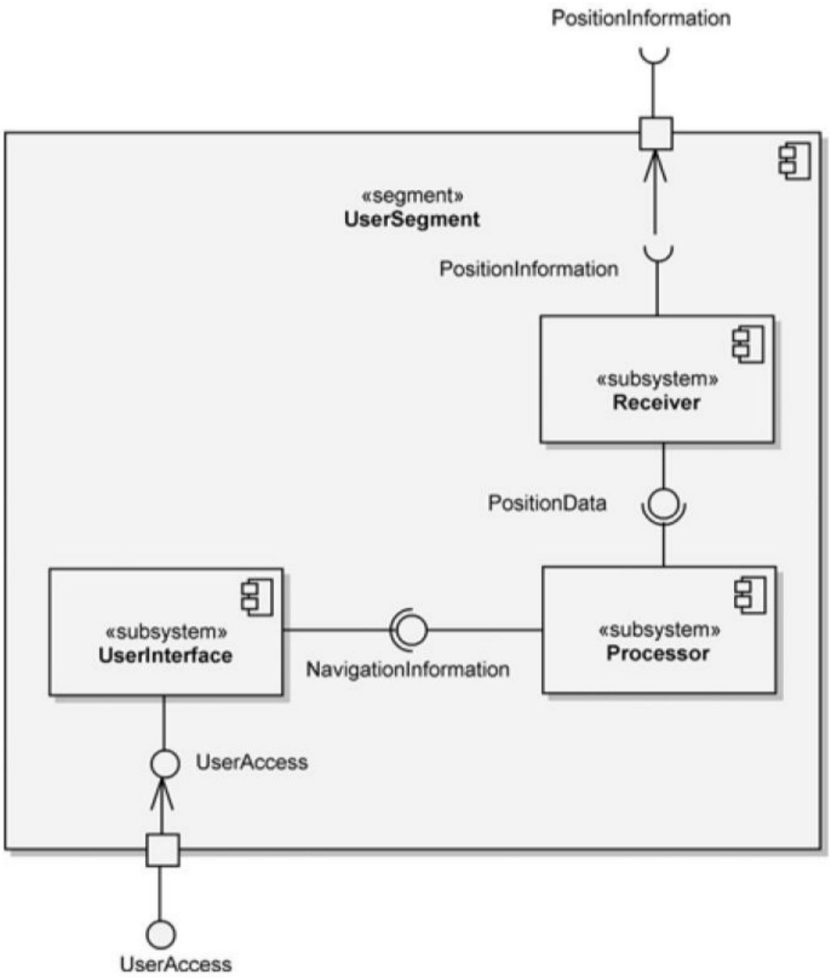


图8-14　UserSegment逻辑架构

8.3 构造

正如在第6章中指出的，在细化阶段的最后，应该开发一个系统的稳定架构。因构造阶段的活动而导致的所有对系统架构的修改，可能限制在那些较低层次的架构元素上，而不是我们关注过的卫星导航系统的部分和子系统。根据本章的意图（即展示开发SNS系统架构的方法，通过逻辑上划分所需功能来定义组成部分和子系统），我们在这个阶段不展示任何架构开发活动。

8.4 交付之后

在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：可扩展性和长服务寿命。与很多其他方面一起，这个长服务寿命的需求主导了一个可扩展的设计，目的是确保可靠地提供需要的功能。在有更多卫星导航系统用户时，当我们将这个设计调整到新实现的时候，他们将发现已有机制有一些未曾预期过的新用法，产生了为系统添加新功能的压力。现在我们来研究，在添加新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，即使用来自其他系统的位置信息的能力，如GPS、GLONASS和Galileo。这将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，对卫星导航系统做最小的变更就可以实现，因为它的影响仅限于用户部分。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，已有的用户部分可以容易地通过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中是十分常见的：通过在已有机制上建造新的应用，可以相当容易地应对系统重要的附加需求。

更激进的改变会怎样呢？假设顾客要引入支持搜索和救援（SAR）任务的能力，用我们的卫星导航系统接收遇险信标。[9]这个新的需求会对我们的架构产生怎样的影响？分析之后，我们看到最大的影响是针对卫星部分，但对地面部分也有些影响。这不意外。如果提前思考这个需求，我们就会添加接收这些特定信号（也许一个信号范围）的能力，不会影响卫星部分的设计，只影响使用功能的操作。否则，我们就需要给卫星部分添加另一个子系统，来提供这个能力。对于地面部分的影响，只是能够将接收到遇险信标的信息中继到适当的管理机构，这可能会涉及对ControlCenter和TT&C子系统进行较小的软件修改或操作修改。

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，对太空部分资产进行修改非常困难，这将迫使系统架构师对系统愿景深谋远虑。但即使在最坏的情况下，我们也能够在将来为正在

开发的卫星部分元素添加SAR能力，同时使对已有的架构或功能影响保持最小。

8.4.2 改变目标硬件

硬件技术的发展仍然比生成软件的速度更快。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，而之后又可能会后悔。[10]出于这个原因，大系统的目标硬件比起软件来说非常容易过时。

例如，在若干年的操作使用后，我们可能决定需要替换地面部分的整个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，我们让子系统接口处于高的抽象层次，那么这种硬件变更就会对我们的软件影响最小。因为我们选择了封装关于ControlCenter子系统具体情况的所有设计决策，所以没有其他子系统曾定义为依赖给定工作站的特定特征。子系统封装了所有这样的硬件秘密。这意味着在ControlCenter子系统中，工作站的行为是隐藏的。因此，这个子系统扮演了一个抽象的防火墙，对所有其他客户屏蔽了特定计算技术的复杂性。

类似地，针对电信标准的激进变更也会影响我们的实现，但是这种影响也是有限的。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，即使网络发生了根本的改变，也绝不会影响任何高层次客户。Gateway子系统对它们屏蔽了真实世界的反复无常。

我们所引入的变更都没有破坏已有架构的结构，这确实是架构良好的面向对象系统的终极标志。

---

[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的站点是www.geocaching.com/。

[3]在UML 2.0中，协作图称为通信图。

[4]但要小心过度分析：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。

[5]卫星总线为星上的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）。

[6]这个时候讨论子系统和组件层次的SNS架构，可以使我们对未来的样子稍有了解。

[7]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。

[8]这种未做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*。

[9]Galileo程序从一开始就添加了这种类型的能力，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。

[10]例如，我们的项目可能已经从一家第三方厂商选择了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，项目经理通常会做两个选择——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。

第9章　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：各种自动化的应用比以前要多得多——从控制汽车各种功能的嵌入式微计算机，到大量消除乏味工作的制作动画电影的工具，再到向数百万消费者提供交互式视频服务的管理系统。这些大型系统的显著特征是极端复杂。建造实现规模小的系统是可取的，但事实告诉我们，某些大型问题还是需要大规模的实现。一些大型应用常常需要由大型软件开发组织来完成。这些大型软件开发组织雇用好几百名程序员，他们必须相互协作，针对一组需求产生上百万行的代码，而这些需求在开发期间肯定会不断变化。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，这些程序必须在一个分布式的目标系统上执行，该系统由多个计算机通过各种方式连接而成。为降低开发风险，这样的项目通常需要有一个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、单用户和基于窗口的工具的开发人员，可能会发现建立大型应用的相关问题错综复杂，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。试想，通过手工系统来控制大都市周围的空中交通，管理载人飞船的生命支撑系统或跨国银行的记账活动。成功地将这些系统自动化不仅能解决手头一些非常实际的问题，还能带来一系列有形的和无形的好处，如更低的操作成本、更高的安全性、不断增加的功能等。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，也需要一些卓越设计者富有创造性的洞察力。

本章处理针对这样一个问题的开发。

9.1 初始

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或跨越国界运送着货物和旅客。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，轻轨运输逐渐成为一种有吸引力的选择，以缓解交通拥挤和解决内燃机带来的污染问题。

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表的压力。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，并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。这些系统大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，同时也提升了安全性。

在本节中，我们通过确定需求，并确定进一步描述所需功能的系统用例，对虚构的列车交通管理系统（TTMS）进行分析。

9.1.1 列车交通管理系统的需求

根据我们开发大型系统的经验，最初的需求陈述总是不完整的，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，我们强烈建议这样一个系统的开发应该特意做出安排，以便采用增量和迭代的方式随着时间演化。正如第6章中所指出的，恰当的开发过程能让用户和开发人员更好地洞察需求，这真的很重要——远远好过没有现成的实现或原型，只在纸上描述需求。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]针对一种在系统实施时肯定过时的硬件技术来打造优雅的软件架构，这肯定是无益的。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，但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，显然软件开发仍是技术风险最

高的环节。我们的目标是将风险控制在可管理的级别，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进行分析和设计工作。在真实世界中，像这样的问题容易遭受分析麻痹症之苦，因为有数千条功能需求和非功能需求，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环境中为候选解决方案建立原型。

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、冲突避免和日志维护。从这些功能，我们定义了8个用例，如下所示。

- Route Train（确定列车路线）：建立列车计划，确定特定列车的行车路线。
- Plan Traffic（规划交通）：建立交通计划，为列车计划提供时间框架和地理区域方面的指导。
- Monitor Train Systems（监控列车系统）：监控车载列车系统正常运行。
- Predict Failure（预测失败）：对列车系统状况进行分析，以预测相对于列车计划失败的概率。
- Track Train Location（跟踪列车位置）：利用TTMS资源和Navstar全球定位系统（GPS）来监控列车的位置。
- 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。
- 保证最大程度地复用和兼容现有设施。
- 提供一个可用性水平达到99.99% 的系统。
- 为TTMS的功能提供完整的功能冗余。
- 列车位置精确到10码。
- 列车速度精确到1.5英里/小时。
- 对操作员的输入在1.0秒之内做出反应。
- 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。
- 最大程度利用商用现货（COTS）硬件和软件。

现在，已经定义了核心需求（至少在一个很高的层次），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和系统交互：Dispatcher（调度员）、TrainEngineer（列车工程师）和Maintainer（维护人员）。此外，还有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。

- 调度员建立列车行驶路线和跟踪单个列车的进展。
- 列车工程师监控列车的状况并操作列车。
- 维护人员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，也看到我们采用了«include»和«extend»关系来组织一些用例之间的关系。用例Monitor Train Systems的功能被用例Predict Failure扩展。在监控系统的过程中，对某个运行异常的系统，或者被挂黄旗、表示有问题待查的系

统，可以请求进行失败预测分析（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。在监控列车交通时，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来实现。

我们可以通过名为用例规格说明的文本文档，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。用例规格说明的格式是通用的，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，这些用例规格说明聚焦于系统用户和列车交通管理系统之间的边界级交互。这个视角经常被称为黑盒视图，因为内部的系统功能外面是看不到的。在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，作为所有相关信息的存放处，记录特定列车路线以及一路上所发生的动作。

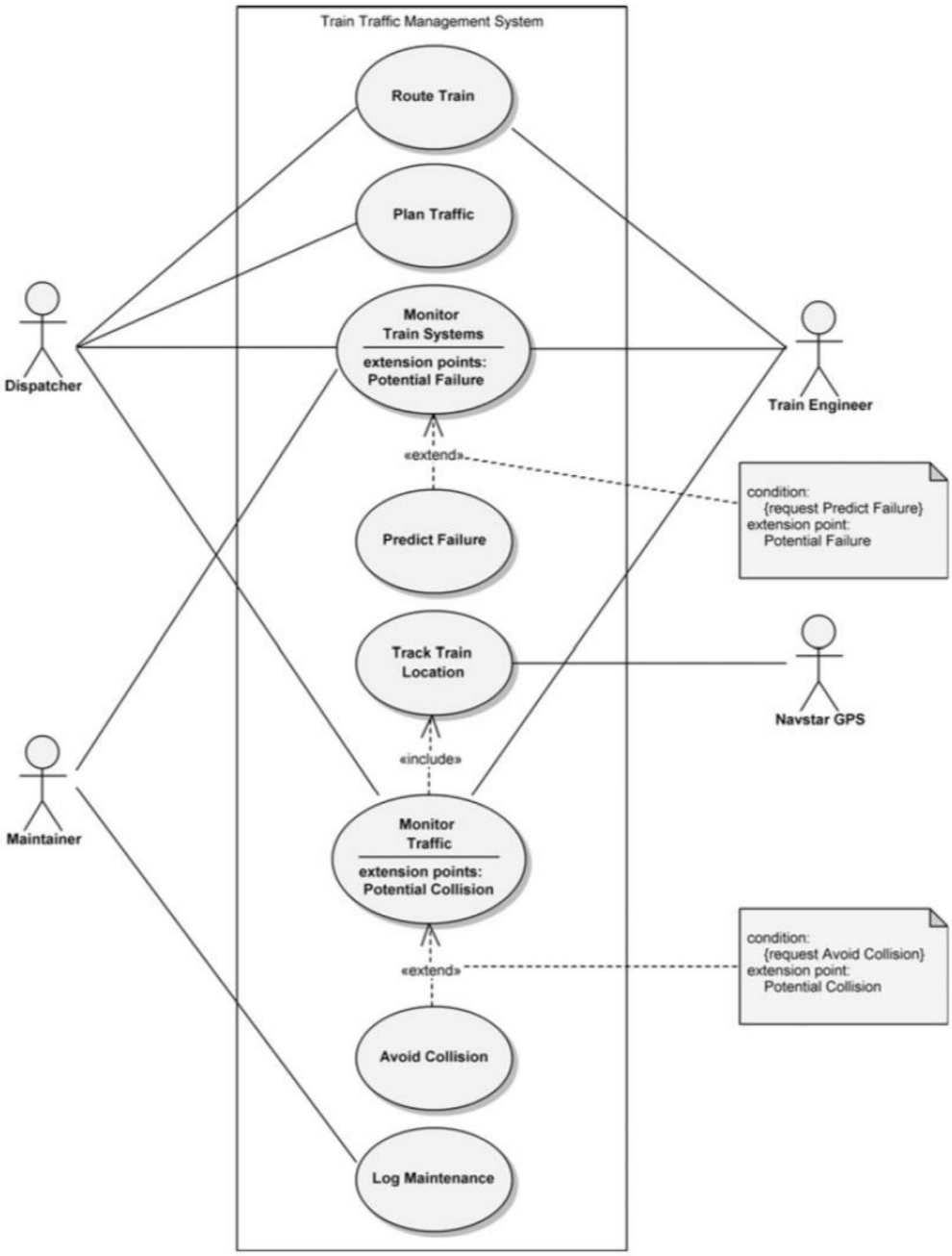


图9-1　列车交通管理系统的用例图

接触点：KatarinaBach

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。在特定时间片内，资源不可以托付给一个以上的列车计划使用。

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的路点的信息。

E．调度员提交已完成的列车计划给TTMS。

F．TTMS为列车计划分配一个唯一ID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车载列车系统正常地运作。

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：已提供关于车载列车系统运行的信息。

限制：无。

假设：机车正在运作时，提供对车载列车系统的监控。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车载列车系统。

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，并决定黄色状况的严重程度不足以马上采取行动。

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。

虽然列车交通管理系统的需求已经大大简化，但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情况。正如之前讨论的，高效的管理不断变化的需求对成功的开发过程来说非常关键，成功的开发过程的定义就是在预算内及时地提供正确的功能。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，还有几个GB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管理需求变化相关风险的关键方法。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程度地使用COTS硬件和软件，这作为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 细化

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上。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

既然已经确定了列车交通管理系统的需求，我们的焦点就转到系统的各部分如何集合起来提供所需功能。这个视角通常被称为白盒视图，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，开发下一个层次的细节。

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，遵从了用例场景的路径。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。

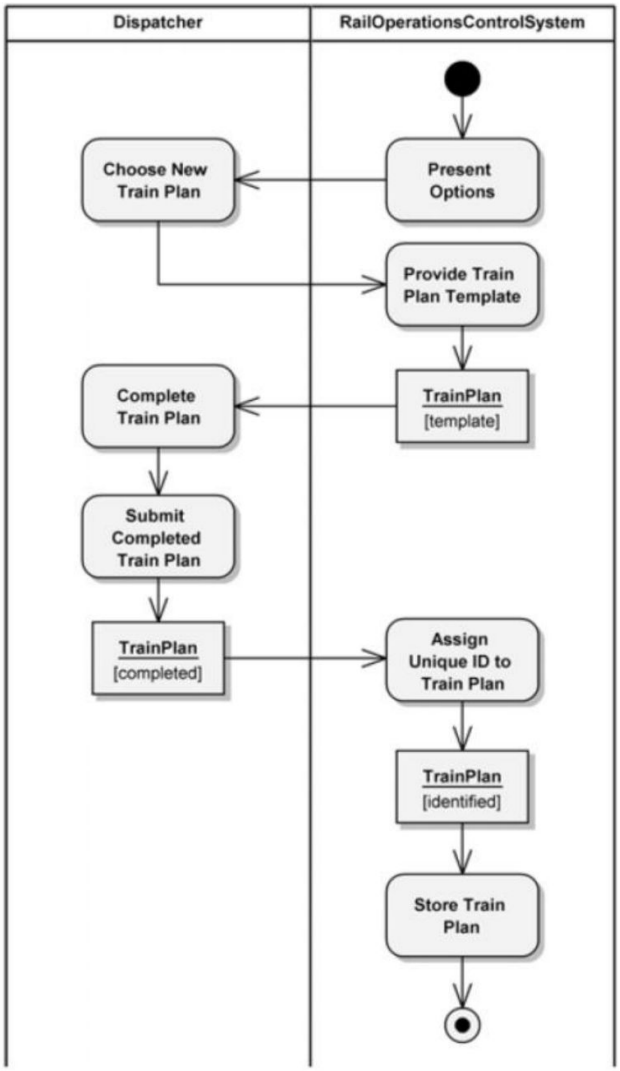


图9-2　Route Train的主要场景

当然，在决定TTMS的组成元素时，必须考虑功能需求、非功能需求和约束。但我们也有两个竞争的技术关注点：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，我们必须努力设计高内聚（把逻辑相关的抽象组织在一起）、松耦合（最小化元素之间的依赖）的元素。因此，我们确定将模块化作为系统的特征，从而将系统分解为一组高内聚、松耦合的元素。

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Locomotive- AnalysisandReporting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、LocomotiveAnalysisand- Reporting系统、EnergyManagement系统和DataManagement单元。

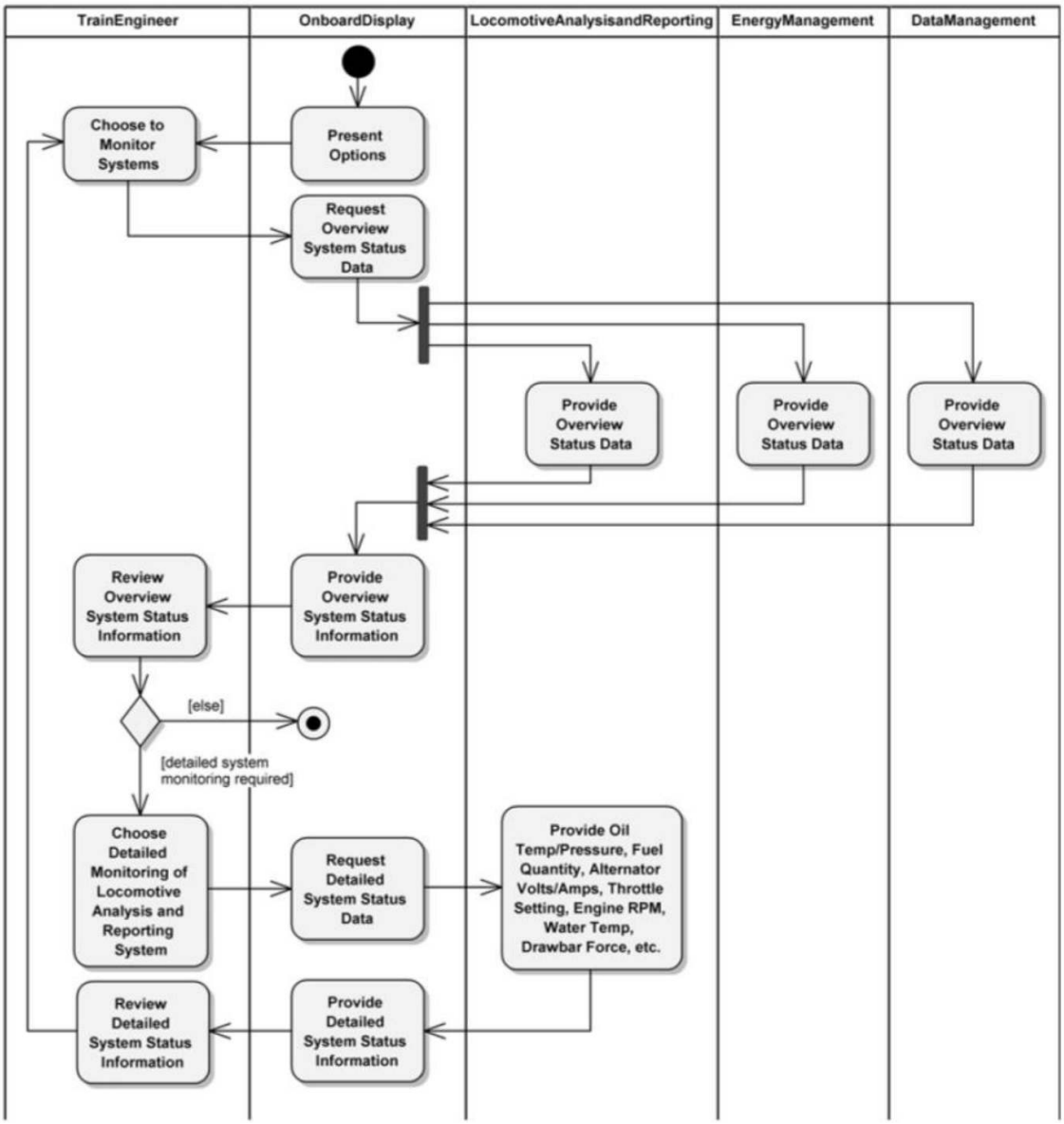


图9-3　Monitor Train Systems的一个备选场景

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复查。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系统的更详细复查，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。在复查这个信息之后，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）场景还是两个（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，为系统能力提供一幅更完整的图画。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。

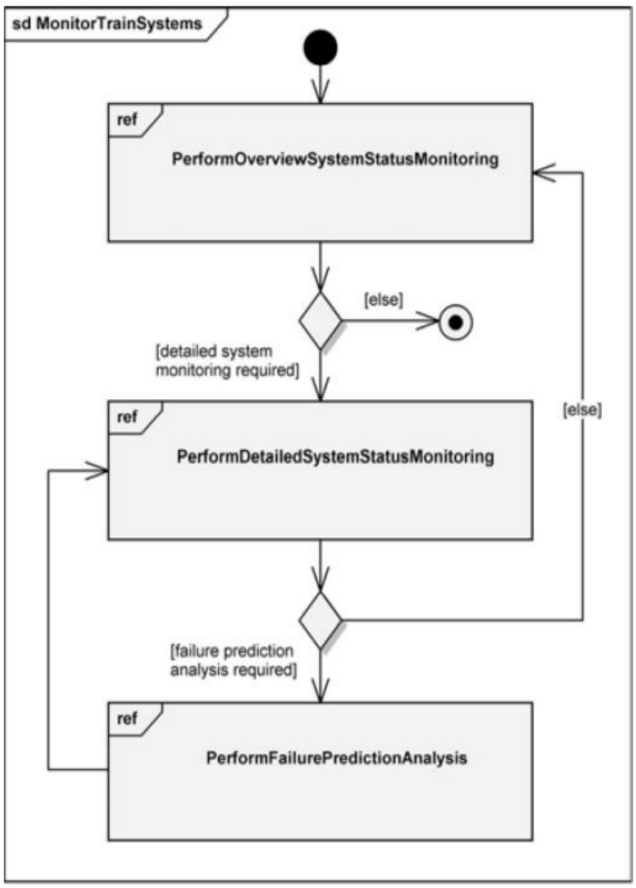


图9-4　Monitor Train Systems的交互概述图

名副其实，这张图展示了完整的Monitor Train Systems用例功能的高层次概览。这张交互概述图展示了交互发生之间的流程，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 Failure 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，这张图可以用来规划从一个交互到另一个交互的流程，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进行更为全面的分析，同时考虑到非功能需求和约束的影响，我们得到了一张列车交通管理系统主要元素的块图，如图9-5所示[2]。机车分析和报告系统包括若干数字传感器和模拟传感器，来监控机车状况，包括油温、油压、燃料量、发电机的伏特和安培、节流设置、发动机RPM、水温和挂钩牵引力。传感器的值通过车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，如果某个传感器的值超出正常的操作范围，预警或告警的情况会被登记。传感器的值保存在日志中，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，以符合时刻表和安全的要求。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。

车载显示系统为列车工程师提供人机接口。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。有软件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其余部分的通信网关，该网络让所有列车、调度员和其他用户相连。

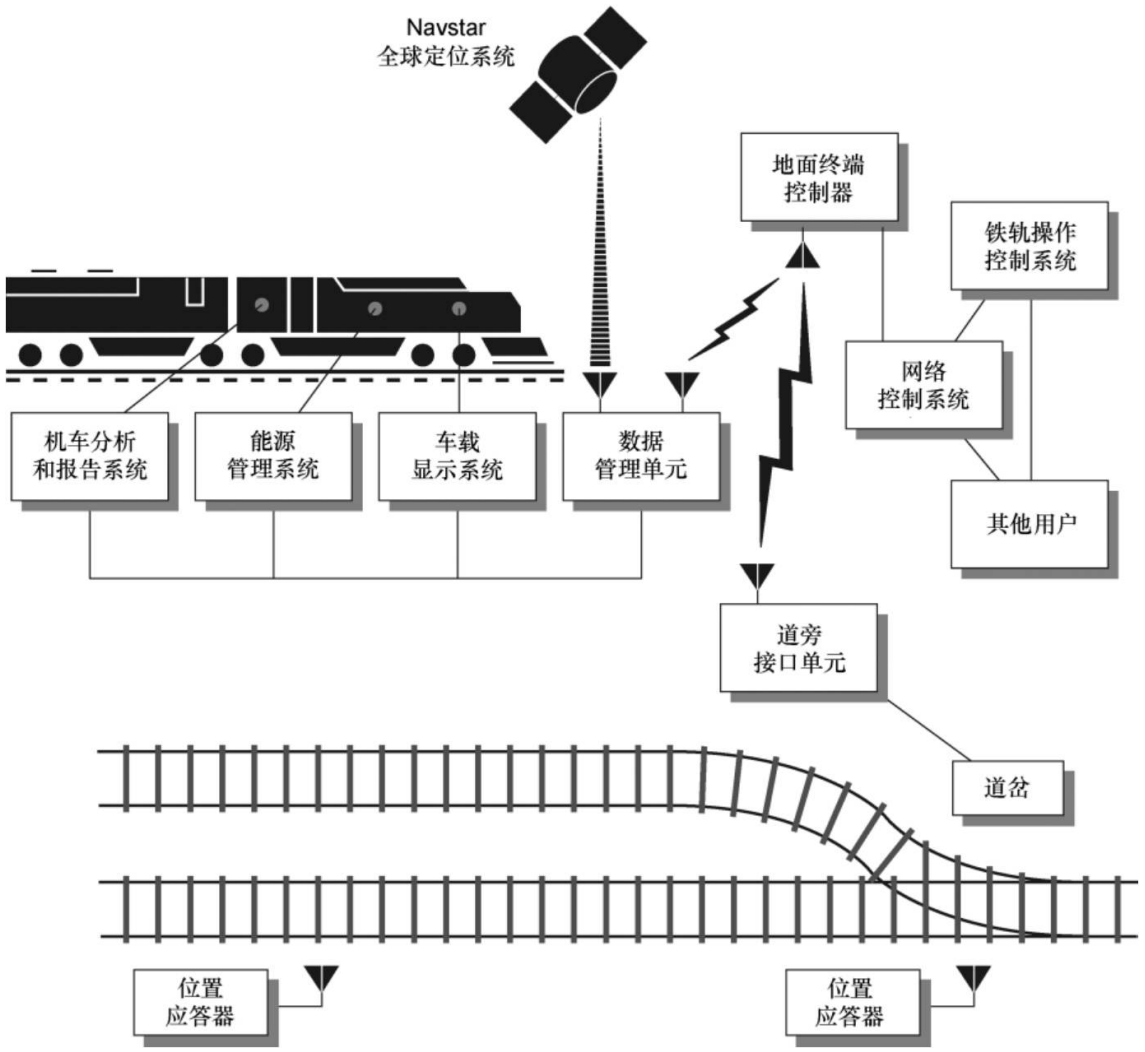


图9-5　列车交通管理系统

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对车轮旋转进行计数，来确定列车的大体位置。这个信息由来自位置应答器的信息校准，在铁轨沿线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传递给经过的列车，以确定更确切的列车位置。列车也可以配备一个GPS接收器，通过它可以使列车的位置精确到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（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传递给经过的列车和道旁接口单元，它也传递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将它的信息传递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可以通过微波链接、地上通信线或光纤连接，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，自动将信息路由到备选路线。

网络控制系统最终连接到一个或多个调度中心，该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中，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多个疆界。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线路沿线标出轨道位置，显示给列车工程师。当检测到危险条件（如列车失控、铁轨故障或潜在的碰撞可能）时，可以通过铁轨操作控制系统（调度员手动或自动）停止列车。调度员也可以调用每个列车工程师能够得到的所有信息，并发送移动权限、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线每天都会变化。因此，列车交通管理系统的设计必须允许添加新的传感器、网络和处理器技术。我们的非功能需求要求有内设的能力来维护和演进TTMS，所以非常明显，我们必须设计一个有弹性的架构，可以随着时间进化。此外，约束告诉我们，系统必须依赖于国家标准（政府和行业），最大程度地使用COTS硬件和软件。

9.2.3 从系统工程到硬件和软件工程

到目前为止，在针对列车交通管理系统分析用例场景并确定首要的功能需求时，我们执行的是系统工程，而不是硬件或软件工程活动。通过这个分析，我们能够指定块图的主要元素，以确定候选的TTMS系统架构。当我们继续开发时，较低层次的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、软件或手工操作来履行的系统功能的比例。那时，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的块图展现了一个利用面向对象方法开发的候选架构，这样我们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高内聚和松耦合。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）。活动图提供了一个清晰的视角，说明了每个系统元素在系统场景中协作时所做的工作。在进行到架构层级的更低层次时，需要更具体的交互细节。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提供了一张序列图，记录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，序列图提供了列车交通管理系统内部工作的更多细节。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，我们必须开始用文档记录元素的细节，如属性定义、操作签名和关联规格。

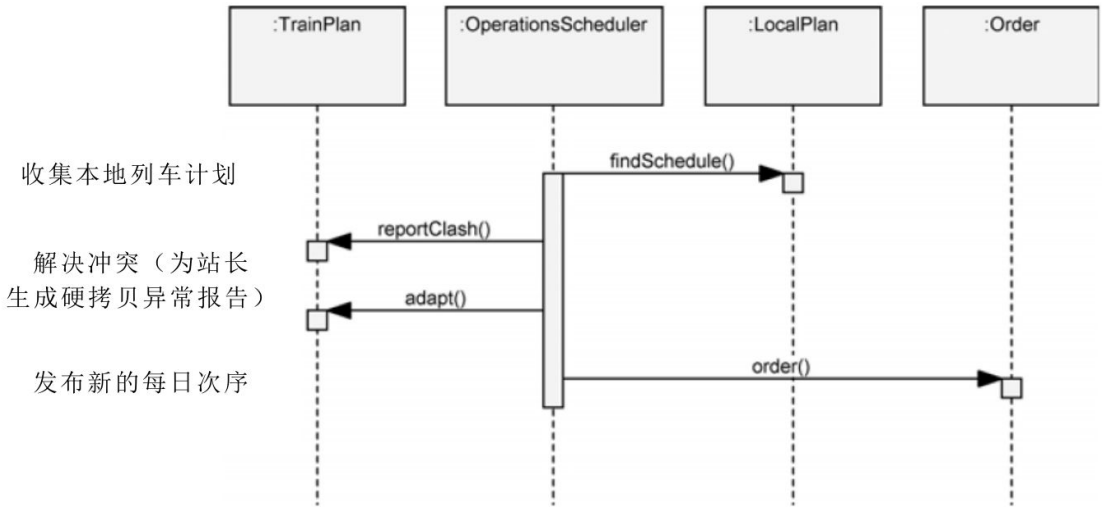


图9-6　处理每日列车次序的场景

在完成TTMS功能的系统工程分析（穿过它的架构层次）之后，必须将系统需求分配到硬件、软件甚至手工操作的元素上。说“甚至手工操作的”，是因为我们常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，特别是在安全关键的领域，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，将需求分配到硬件或软件是相当明显的。例如，对于描述列车时刻表来说，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。在选择现有商业产品（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。但是一般来说，在需要最多弹性的地方，我们会倾向于软件，而在性能至关重要的地方，我们会选择硬件。

考虑到问题的目的，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推翻的，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如图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，每组道旁设备（包括一个道旁接口单元和它的道岔）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路、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，每个调度中心设一个（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，我们期望无论何时，如果首要计算机发生故障，备份计算机都将在线。在空闲期间，备份计算机可以用来服务其他低优先级用户的计算的需求。

在运营时，列车交通管理系统可能涉及数百台计算机，包括每个列车有一台，每个道旁接口单元有一台和每个调度中心有两台。部署图只展示了这些计算机中的一部分，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，要保持头脑清醒，关键是在系统的关键元素之间打造良好而明确的接口。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，但它们必须快速形式化，这样，系统的不同部分可以并行开发、测试和发布。在机会出现时，定义良好的接口也使硬件/软件的折中容易得多，不会干扰已经完成的系统部分。此外，我们不能期望在一个大型的、也许是全球分布的开发组织里，所有开发人员都有一个完整的视图，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说明工作留给最好的架构师。

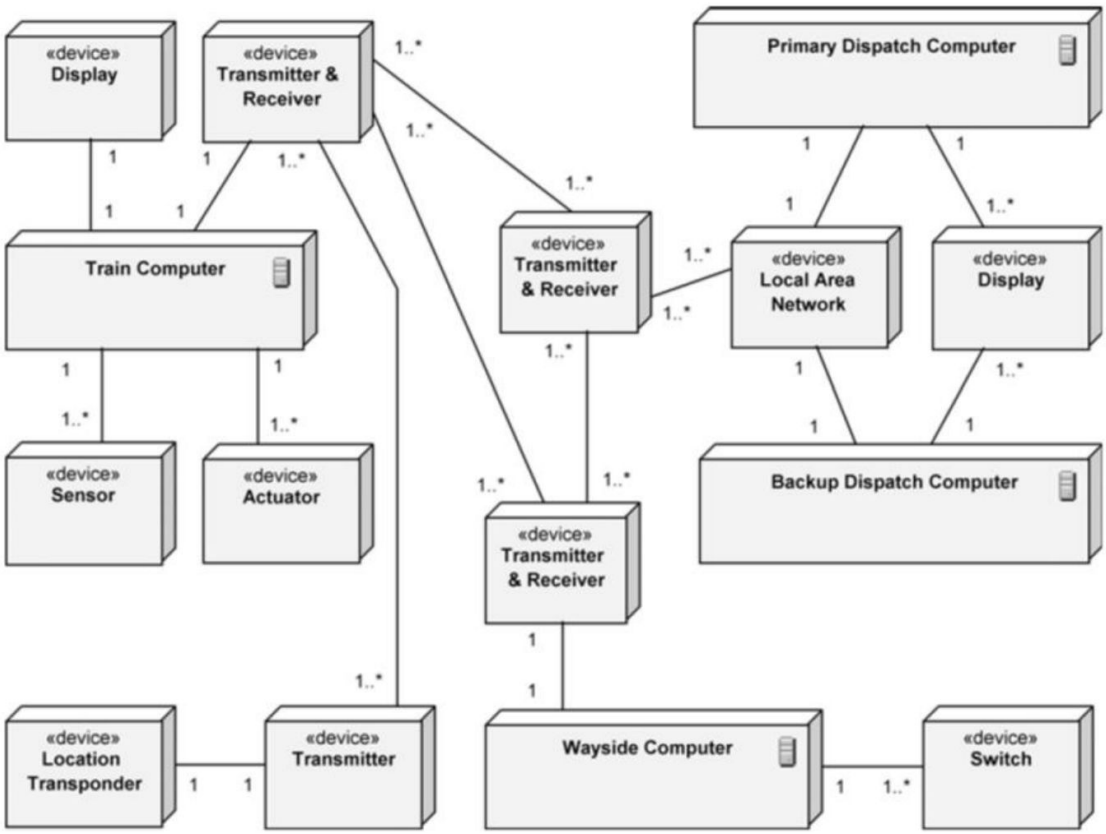


图9-7　列车交通管理系统的部署图

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，实际上有四个不同的子问题要解决：

- 网络
- 数据库
- 人/机接口
- 实时控制模拟设备和数字设备

在涉及最大的开发风险时，如何来识别这些问题？

这个系统的整合靠的是一个分布式的通信网络。消息通过无线电，从应答器传送到列车，并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传送。消息也必须在调度中心和单个地面终端控制器之间传送。整个系统的安全运营依赖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。即使有来自网络各处的并发更新和查询，我们也必须保持这个信息最新而且一致。这基本上是一个分布式数据库问题。

人机接口工程提出一组不同的问题。具体地说，这个系统的用户主要是列车工程师和调度员，他们不需要精通计算机的使用。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种应用的最终用户来说，这样的界面被认为是对用户不友好的。因此，必须小心设计所有形式的用户交互，以适应这个领域特定的用户群。

最后，列车交通管理系统必须与各种传感器和执行器（actuator）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。

这四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问题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应对特定的子问题。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。

如果跨越这四个子问题域做一个简短的领域分析，可以发现有如下三个共用的高层次关键抽象。

- 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。

每列列车有一个在轨道上的当前位置，每列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，有且只有一列列车，涉及轨道上的很多点。

可以继续为四个接近独立的子问题各设计一个关键机制：

- 消息传送
- 列车时刻表计划
- 显示信息
- 传感器数据采集

这四个机制成为系统的灵魂。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，在这里安排最好的系统架构师是非常重

要的，他们实验各种候选方法，并最终定下一个框架。初级开发人员可以利用这个框架构造系统的剩余部分。

9.3 构造

架构设计包括建立系统的中心类结构，并为这些类的共同协作提供规格说明。尽早聚焦于这些机制直接攻击了系统中风险最高的元素，明确地记录了系统架构师的愿景。从根本上说，这个阶段的产品将作为类和协作的框架，最终系统的其他功能元素将在这个框架上建造。

在本节中，我们将从检查这个系统的四个关键机制的语义开始，即消息传送、列车时刻表计划、显示信息和传感器数据采集。这引发了关于发布管理的讨论，发布管理支持我们的迭代和增量开发过程。本节最后将分析如何开发一个系统架构，以支持TTMS的子系统规格说明。

9.3.1 消息传送

所谓消息，我们的意思不是面向对象编程语言中的方法调用，而是问题域词汇表里的概念，它处于更高的抽象层次。例如，典型的列车交通管理系统消息包括如这样的信号：激活道旁设备。该信号表明列车经过特定的位置，以及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内的两个不同的层次上传送：

- 计算机和设备之间
- 计算机之间

我们的兴趣在第二个层次的消息传送上。因为我们的问题涉及地理上分布式的通信网络，我们必须考虑诸如噪声、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（参见前面的图9-7）。对于每一对通信，我们必须问以下三个问题：

（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层次？

这些问题没有确定的解决方案。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，或者消息量太大或太小）。

在这个层次的设计，绝对关键的是聚焦于这些消息的实质，而不是形式。我们常常看到一些系统架构师，一开始就为消息选择“比特层次”的表达方式。过早选择这样低层次的表达方式有一个真正的问题，即它肯定会改变，从而导致依赖于特定表达方式的每一个客户都会受到干扰。而且，在设计过程的这个阶段，我们不能充分了解这些消息的使用方式，所以不能明智地确定具有时间和空间效率的表达方式。

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上。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地执行的操作。

图9-8中的类图记录了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。三个更低层次的类代表消息的主要类目，即TrainStatusMessage、TrainPlanMessage和WaysideDeviceMessage。每个类都进一步特化。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，对最终使用基类的客户来说，以这种方式重新组织类层次通常语义影响最小。

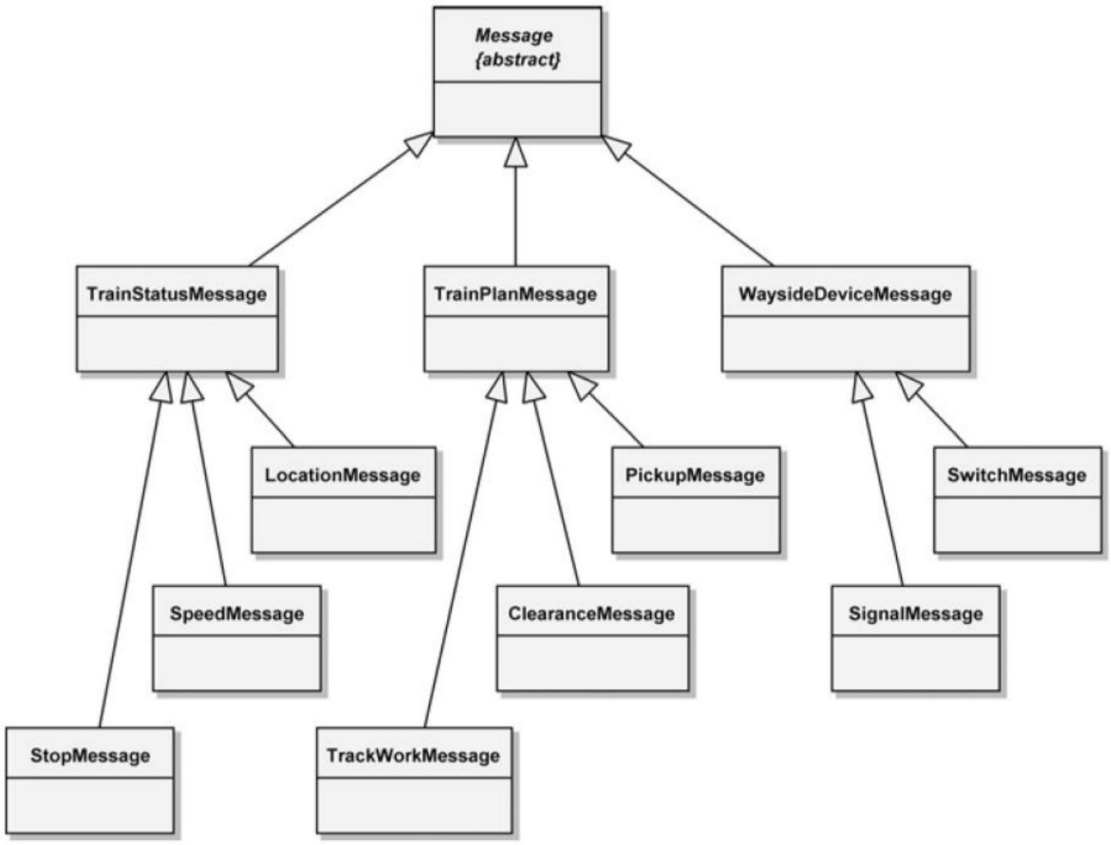


图9-8　Message类图

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。为了形式化所有这样的类的角色和责任，我们可以对这个层次结构中更令人感兴趣的基类，开始领域分析。

一旦设计了一些更重要的消息接口，就可以基于这些类来编写程序，模拟消息流的创建和接收。在与之接口的部分开发期间或开发之前，我们可以将这些程序作为临时的辅助性代码，来测试系统的不同部分。

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，并随着不太通用的通信形式的不断揭示，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式地添加这些消息，而不会破坏已有的系统设计，因为这样的改变一般是向上兼容的。

当我们对这个类结构感到满意时，就可以开始设计消息传递机制本身。机制的设计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层次进行，这样客户不需要担心消息送达如何发生。这样的消息传递机制允许它的客户对消息的发送和接收方式做出简化的假设。

图9-9提供了一个场景，记录了消息传递机制的设计。如图所示，为了发送一条消息，客户首先创建一条新的消息m，然后通过广播传递给它的节点的消息管理器，消息管理器的责任是将消息放进队列，以便最后的传输。注意，我们的设计使用四个主动对象（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，然后利用Transporter对象的服务，将消息简化为它的标准形式，并在整个网络广播。

正如这幅图所说明的，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()的参数交付，这是一个同步操作。

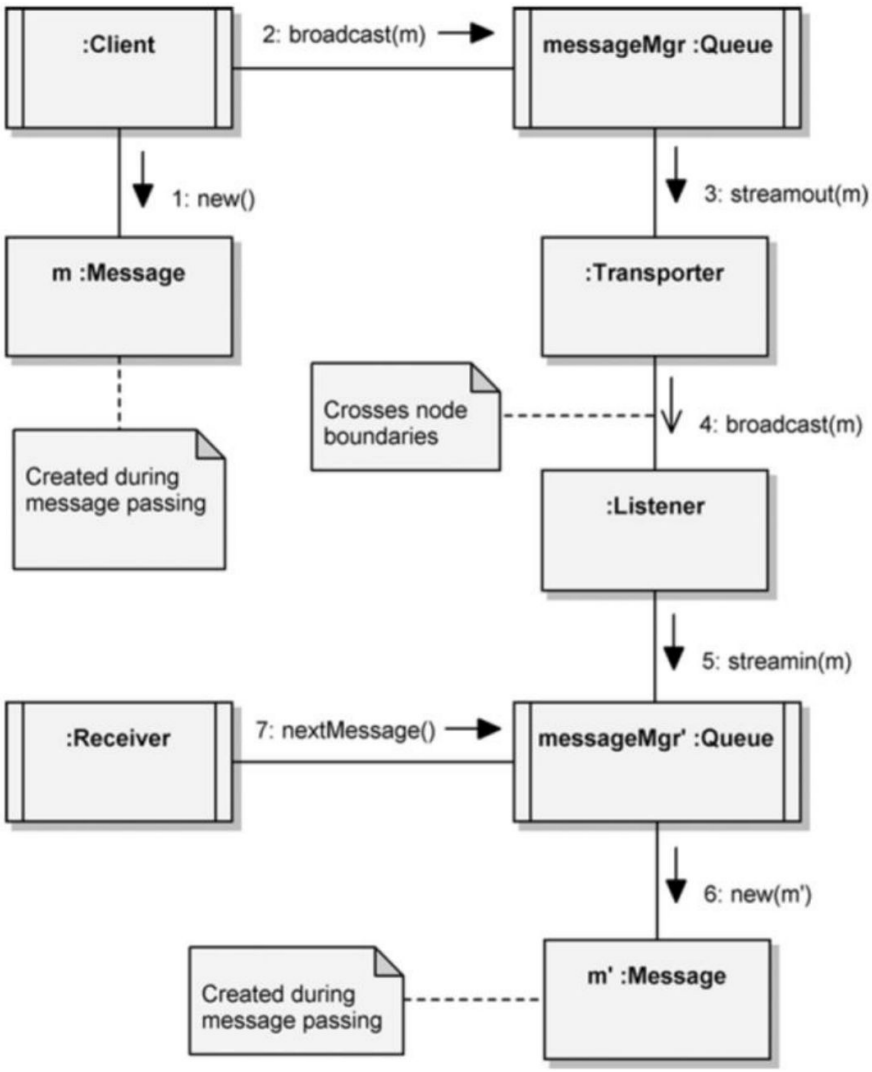


图9-9　消息传递机制

我们的设计将消息管理器放在ISO开放系统互连（OSI）网络模型[3]的应用层。这让所有发送消息的客户和接收消息的客户在最高抽象层次上操作，即针对具体应用的消息。

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　列车时刻表计划

正如之前指出的，列车计划的概念是列车交通管理系统运作的中心。每列列车有且只有一个活动的计划，每个计划只分配到一列列车，在轨道上可能会涉及很多不同的次序和位置。

第一步是确定到底哪些部件构成了列车计划。为此，需要考虑一个计划的所有潜在客户，并考虑每一个客户预期将如何使用该计划。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。从这个意义上说，列车计划是信息的存放处，对于特定列车路线及一路上发生的动作（如挂载或卸除车厢），保存相关的所有信息。

图9-10记录了我们关于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）。因此，我们可以看到每个列车计划有且只有一个班组，可能有很多一般命令和很多动作。我们期望这些动作按时间排序，每个动作由一些信息组成，如时间、位置、速度、权限和指令等。例如，一个特定列车计划可能由表9-1中所示的动作组成。

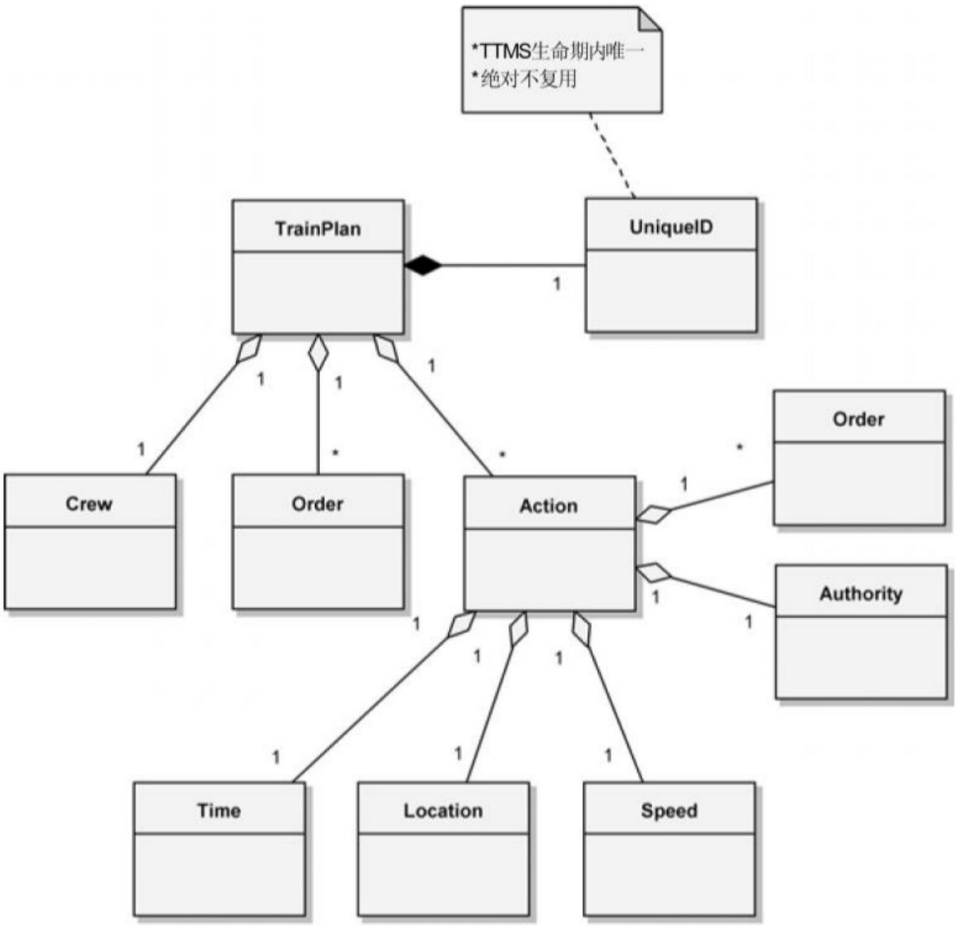


图9-10　TrainPlan类图

如图9-10所示，TrainPlan类有一个UniqueId，其目的是提供唯一编号，识别每个TrainPlan实例。因为这里的信息复杂，这张图中的类在别的情况下可以考虑作为一个类的属性，但在这里是真正独立的类。例如，UniqueID类不只是一个识别编号，它包含了各种属性和操

作，以满足严格的国家规定和国际规定。另一个例子是班组的工作要受到限制：他们只能在特定位置工作，或者在特定时间、特定位置必须保持限速。

正如Message类和它的子类那样，我们可以在开发过程早期设计列车计划最重要的元素；在我们实际对计划的各种客户应用计划时，它的细节将随时间演进。

同一时间内，我们可能会有太多的活动或不活动的列车计划，这让我们必须面对之前谈到过的数据库问题。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，有无限的计算资源，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。但是真实世界与此相差甚远，所以这个解决方案不实际。我们必须考虑通信延迟，我们也没有无限的处理器周期。因此，如果必须从列车上访问位于调度中心的计划，就根本不能满足实时和接近实时的需求。

但是，我们可以在我们的软件里创造单个中央数据库的幻象。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的各个站点。然后，出于效率的考虑，每列列车的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

如图9-11所示，这个场景展现的是列车时刻表计划机制的一个功能。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中，有零或多个镜像拷贝分散在网络各处。无论何时，如果某个客户请求一个特定列车计划的拷贝（通过操作get()，调用时以UniqueId作为一个参数），首要版本都将被克隆，并作为参数传递给客户，该拷贝的

网络位置将记录在数据库中。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。因为我们记录了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数据库，中央数据库强迫其余所有拷贝的状态进行相应的更新。为了保证变更在网络上一致，可以使用记录锁定机制，这样，除非所有拷贝和首要版本已经更新，否则变更不会被提交。

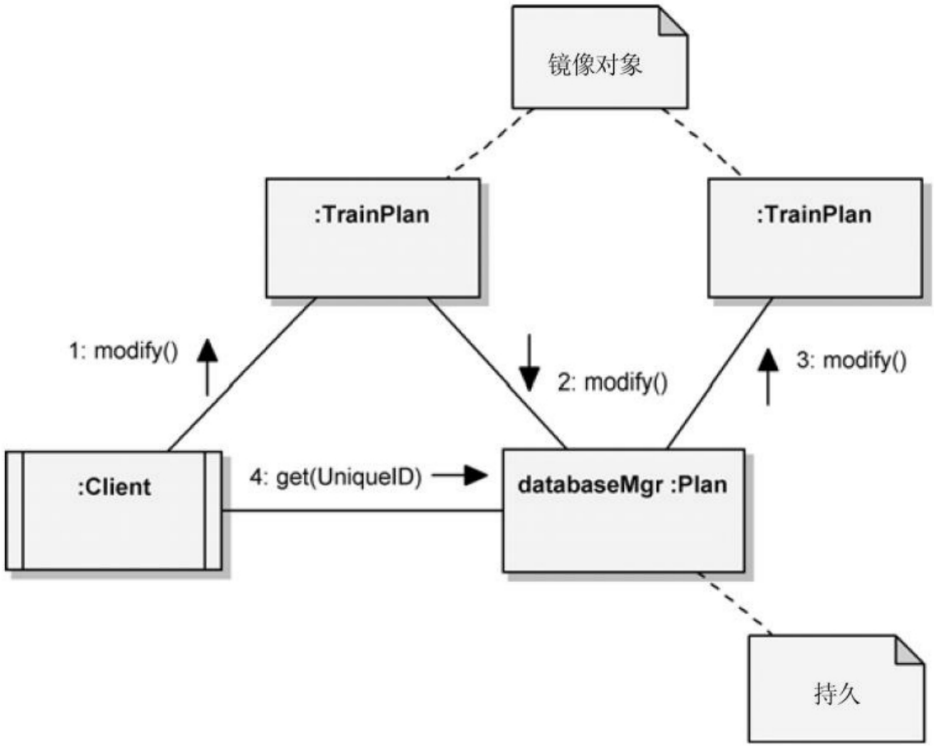


图9-11　列车时刻表计划

如果调度中心的一些客户引发该变更，这也许是由一些调度员的动作引起的，那么这个机制同样适用。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。在这两种情况下，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之中，然后在这个更低层次的消息传递机制上构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，以便满足所有关于数据库备份、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，有助于我们关注问题的领域特定部分。通过使用标准图形设施，可以在显示需求方面实现类似的好处。使用现货图形软件，高效地提升了系统的抽象层次，这样，开发人员不需要担心在像素层面上操纵可显示对象的可视展现。对如何显示各种对象的设计决策进行封装，这依然是重要的。

例如，考虑显示特定区间的轨道特征描述和等级。需求表明，这样显示可以出现在两个不同地方：调度中心和列车上（列车上的显示只关注列车前方的轨道）。假设我们有一些类，它们的实例代表轨道的区间，我们可以用两种方式来展现这样的对象的状态。在第一种方式中，我们可以有一些显示管理对象，通过查询待显示对象的状态来构造可视的展现。另一种方式是不要这个外部对象，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。最后，可能有很多不同种类的可显示对象，每个对象由不同组的开发人员实现。如果让每个可显示对象独立实现，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。更好的解决方案是对各种可显示对象进行领域分析，确定它们有什么共同的可视元素，并设计一组中间类工具，为这些共同的图像元素提供显示例程。这些类工具的构建可以基于较低层次的现货图形包。

图9-12阐明了这个设计，展示了所有可显示对象的实现共享了通用的类工具。这些工具的构建基于较低层次的Windows接口，这些接口对所有高层次类隐藏。实践中，像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，我们的实现需要为Windows API提供一套类工具，就像列车显示工具一样。

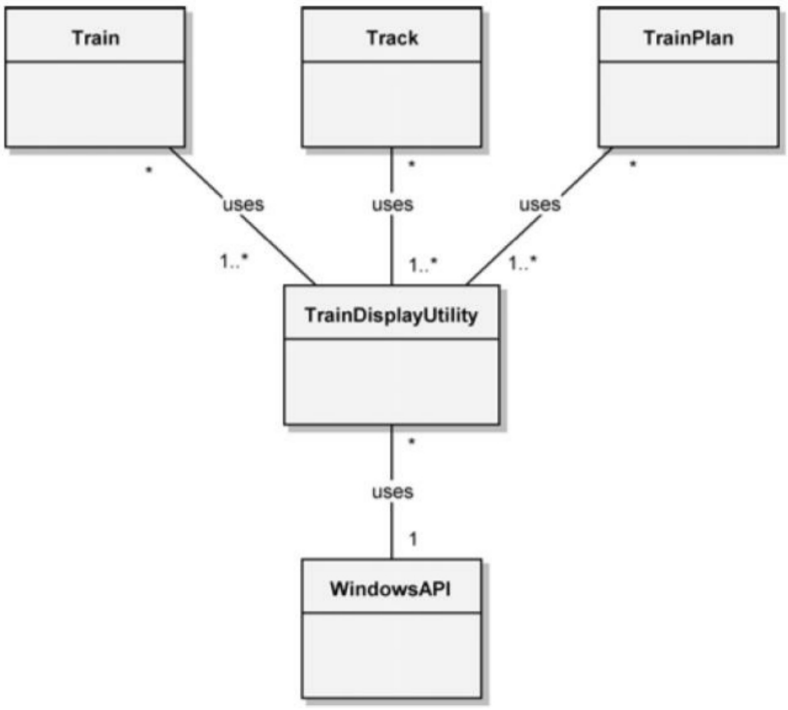


图9-12　用于显示的类工具

这个方法有一个主要优势，它限制了因硬件/软件折中而导致的所有较低层次变化的影响。例如，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，低层次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

需求表明，列车交通管理系统包括很多不同种类的传感器。例如，每列列车有监控油温、燃料量、节流设置、水温、牵引负荷等传感器。类似地，一些道旁设备有活动传感器，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；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，当机车油压降到危险的水平时）。

为每一种传感器复制这个行为不仅让人厌烦和容易出错，而且通常会导致冗余的代码。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，导致大量出现细微差别的传

感器机制，系统更难维护。因此，对所有周期性的、非离散的传感器进行领域分析是非常值得的，这样我们可以为各种传感器开发一个通用的传感器机制。我们可以采用包含传感器类层次的架构，并采用基于帧的机制，周期性地从这些传感器获取数据。

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术的使用，并进一步分析系统架构及其子系统规格说明。

我们的增量开发过程先是从选择少量令人感兴趣的场景开始，对我们的架构做一个垂直的切片。然后，实现足够的系统部分，得到一个可执行的产品，至少可以模拟这些场景的执行。

例如，我们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，要求我们触及几乎每一个关键的架构接口，从而迫使我们验证我们的战略假设。一旦成功通过了这个里程碑，就可以按照以下顺序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来追踪列车交通。

对于12～18个月的开发周期，这可能意味着每三个月产生一个相当稳定的发布版本，每个版本都基于其他版本的功能。在完成时，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，我们识别出最高开发风险并直接攻击它们。对于这样的控制系统应用，意味着尽早引入测试进行积极控制（这样我们可以足够早地识别所有系统控制漏洞，并对它们进行修复）。发布序列表明，这也意味着为每个发布版本广泛地选择场景，跨越系统的诸多功能元素，这样就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

对于非常大型的系统来说，软件设计通常必须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，在所有情况下，整个过程中都必须进行软硬件的折中。这暗示着软件对硬件的依赖必须最大限度地隔离，这样，在缺少稳定的目标环境时，软件设计也可以进行。这也暗示着软件在设计时，必须有可替换的子系统的理念。在列车交通管理系统这样的命令和控制系统中，我们希望利用在开发系统的软件期间成熟起来的新硬件技术。

我们还必须尽早对系统的软件进行明智的物理分解，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到许多非技术原因的驱使，其中最重要的也许是将工作分配到独立的开发者团队。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做出良好的技术决策，进行合适的子系统分解。

如何选择合适的子系统分解？最高层次的对象通常围绕着功能线聚集。再说一次，这与对象模型并不是正交的。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。我们是在谈论一些场景，它们从外部可见，有可测试的行为，由对象逻辑集合的动作协作产生。因此，首先识别的最高层次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，然后随着时间推移来演进它们的接口。

如图9-13所示的组件图展示了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，包含了我们之前确定的四个子问题的功能，即网络、数据库、人机接口和实时设备控制。

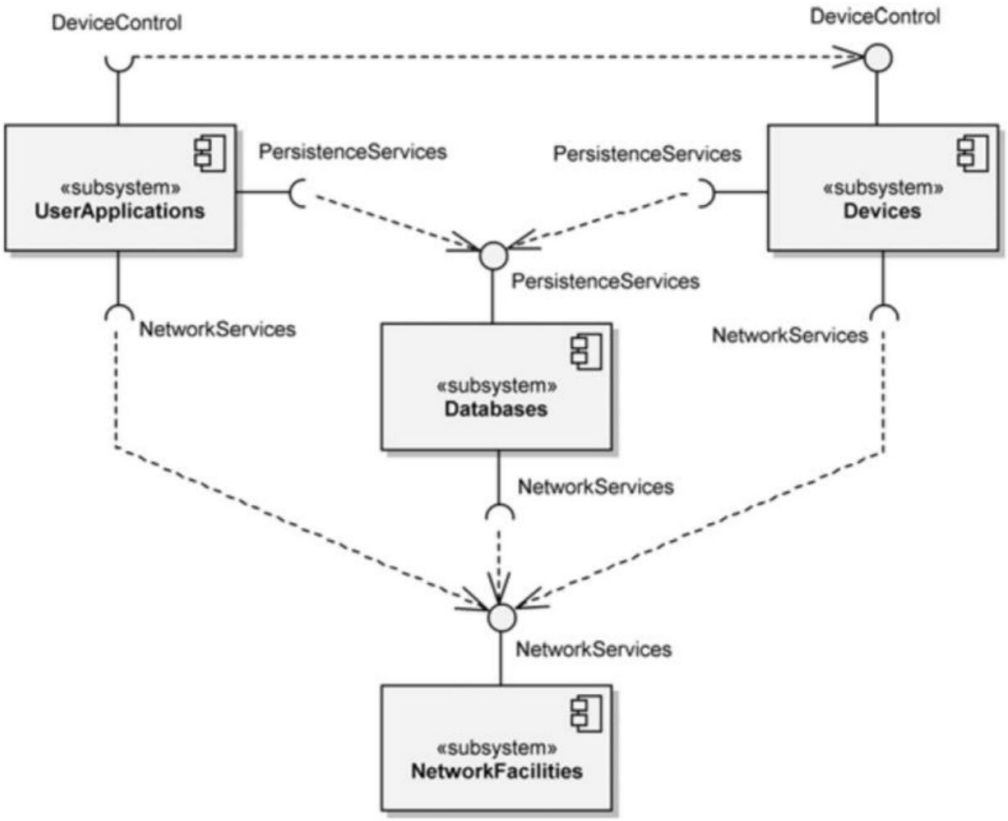


图9-13　列车交通管理系统的顶级组件图

9.3.7　子系统规格说明

如果我们聚焦于这些子系统中任意一个的外部视图，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存放处，因此，最好由它们提供的接口所输出的资源来刻画，如图9-13所示的NetworkFacilities子系统提供的NetworkServices。

如图9-13所示的组件图只是TTMS子系统架构规格说明的起点。这些顶级子系统必须进一步分解为多个架构层次的嵌套子系统。例如，NetworkFacilities子系统，我们将它分解为另外两个子系统：一个私有的RadioCommunication子系统和一个公有的Messages子系统。私有的子系统隐藏了物理无线电设备的软件控制细节，而公有的子系统提供了之前设计的消息传递机制功能。

名为Databases的子系统基于NetworkFacilities子系统的资源构建，实现了之前设计的列车计划机制。我们选择将这个子系统进一步分解为两个公开的子系统，代表系统中的主要数据库元素。这两个嵌套的子系统分别命名为TrainPlanDatabase和TrackDatabase。我们也希

望有一个私有的子系统DatabaseManager，它的目标是为两个领域特定的数据库提供所有的公共服务。

在Devices子系统中，我们选择将所有与路旁设备相关的软件分组到一个子系统，将所有与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，它们的构建都基于TrainPlanDatabase和Messages子系统的资源。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，我们选择将顶级UserApplications子系统分解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EngineerApplications包含了一些资源，这些资源提供了需求规定的列车工程师和机器之间的所有交互，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包括的软件提供了调度员和机器之间所有的交互功能。EngineerApplications和DispatcherApplications共享共同的私有资源，如子系统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了四个顶级子系统，其中包含若干更小的子系统，我们已经将之前发现的所有关键抽象和机制分配到这些子系统中。这些子系统被分配给设计和实现它们的开发团队，同时保持已定义的接口，让每个子系统通过这些接口与同一抽象层次上的其他子系统协作。

这种分解大型复杂问题的方法，为我们正在开发的系统提供了不同的视图。这样，一个系统发布版本就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版：每个开发人员有一个，质量保证团队有一个，也许还有一个提供给早期顾客使用。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，将他们负责的软件的新版本集成进去。

这种方式成功的关键是小心设计子系统接口。一旦工程化，这些接口必须严格保护。如何决定每个子系统的外部视图？我们将每个子系统看成一个对象。因此，问同样的问题，就像问一个更原生的对象：这个对象包含什么状态？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。它的构建基于其他三个子系统（Messages、TrainDatabase和TrackDatabase），它有若干重要的

客户，即四个子系统：WaysideDevices、LocomotiveDevices、EngineerApplications和Dispatcher Applications。TrainPlanDatabase包含相对简单的状态，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的行为。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。我们最终会通过类的形式记录所有这些设计决策，这些设计决策构成了这个子系统。类提供所有这些操作的声明。

在设计的这个阶段，我们将继续每个子系统的设计过程。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演进。有幸的是，对于较小的对象，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都是向上兼容的。

9.4 交付之后

一个安全而有用的控制系统是一项持续的工作。这不是说我们绝不会到达稳定系统的那个点——事实上，每一交付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，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大的自由度，使其能够敏捷地适应不断变化的法规环境和市场。

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，即工资单处理。具体地说，假设我们的分析表明，铁路公司工资单处理目前由一款已经停产的硬件支持，我们有很大的风险丧失工资单处理的能力，因为一个严重的硬件故障就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工资单处理。这两个看起来不相关的问题如何共存？这不难构想，我们可以简单地把它们看成分离的应用，工资单处理作为背景活动运行。

进一步的调查表明，集成工资单处理确实能获得巨大的价值。你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，我们有可能追踪真实的人员分配和计划的人员分配，这样，可以计算工作时间、加班数量等。通过直接取得这个信息，工资单计算会更精确，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，代表工资单处理功能。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，系统能够相当容易地处理重要的附加需求。

可能有一个更重要的变更，即通过建造一个调度员助理，把专家系统技术注入我们的系统，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Dispatcher Applications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，建立一种将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，架构的一个迷人特征就是它们将实现相当大量的关键功能以及适应性。换句话说，如果我们已经选择了正确的元素功能和结构，就会发现，用户很快会找到各种方法，以设计人员绝对想象不到或不希望的方式演进系统功能。如果我们在客户对系统的使用中发现了模式，那么将这些模式编写出来，让它们正式成为架构的一部分，是非常有意义的。设计良好的架构有一个特点，可以通过复用已有的机制引入这些新的模式，从而保持它的设计完整性。

---

[1]事实上，对于很多这种复杂的系统，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。

第10章　人工智能——密码分析

具有感知能力的生物展示出一个庞大、复杂的行为集合，这些行为源自于意识，通过我们知之甚少的机制产生。例如，想一想需要在某城市办事时，如何解决路线安排的问题。再想想，当你穿过一间灯光昏暗的房间时，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（完全是字面意义上的）视觉输入的理解。从众多的声源中鉴别出一个说话的人，需要听者从噪声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：

> “在许多特征上，智能系统与传统系统不同，这些特征并非总是存在，
>
> - 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，这些子系统包含各种经过选择的方法。
> - 它们与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让智能系统的构建成为一项非常困难的任务。正在为不同领域开发的智能系统将影响我们的生命财产安全，如医疗诊断或飞机航线。当我们认识到这一点时，这个任务就变得更加迫切了。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宣传，但是人工智能的研究的确给了我们一些合理、实际的想法。我们从中发现知识表示的各种方法，以及智能系统的通用问题解决架构的演化方式，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，该系统使用黑板框架，与人类求解同样问题所

用的方法差不多。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 初始

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，甚至需要挑战最尖端的技术。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“包含一些数据表示的方法，使未授权方难以理解”[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作置换加密，自从罗马时代起，该方法就一直被采用。使用这种加密方法，明码字母表的每一个字母都被映射为一个不同的字母。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，如下明文：

```
CLOS is an object-oriented programming language
```

可以被加密成如下密文：

```
DMPT jt bo pckfdu-psjfoufe qsphsbnnjoh mbohvbhf
```

更常用的情况是，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，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。在更高的层次的抽象上，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，然后撤销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，给定一份密文，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，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就不予考虑）。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），有时从特殊到一般进行推理（?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以及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，这个框架被成功应用到其他领域，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，空间与时间效率更高。

黑板框架是一个架构模式，我们对问题求解算法的分析结果表明，可以应用这个框架。这个框架可以通过类和机制来表示，其中机制描述类的实例之间如何协作。

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，其他人拿着他们的拼块走到黑板，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的（例如，从左上角开始，试拼每一块）”[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层次。对象以及它们的属性定义解空间的词汇表”[9]。

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]

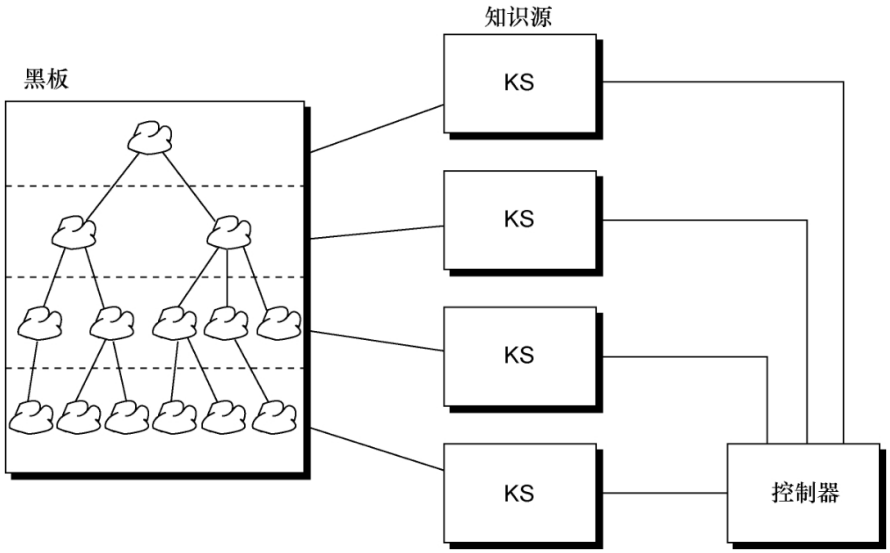


图10-1　一个黑板框架

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。进一步的，每一个知识源使用在一个层次上的对象作为它的输入，然后产生或修改另一层次上的对象作为输出。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（在低的抽象层次上）组成一个新的单词（在更高的抽象层次上）。另外，包含句子结构知识的知识源可能假设需要一个动词（在高级抽象层次上），通过过滤一系列可能的单词（在较低的层次上），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，记录此人求解该领域问题时的启发式思考过程。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，这种做法可能涉及尝试求解一些密码，并记录下整个思考过程。

分析表明有13个知识源与此相关，这些知识源及它们包含的知识如下：

- 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），符合一个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 细化

现在我们准备使用前面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复用的例子，因为我们可以复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层次对象的一部分。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

黑板是一个不同层次抽象的细化结构。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层次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

这五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，这些对象的属性使它们有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。

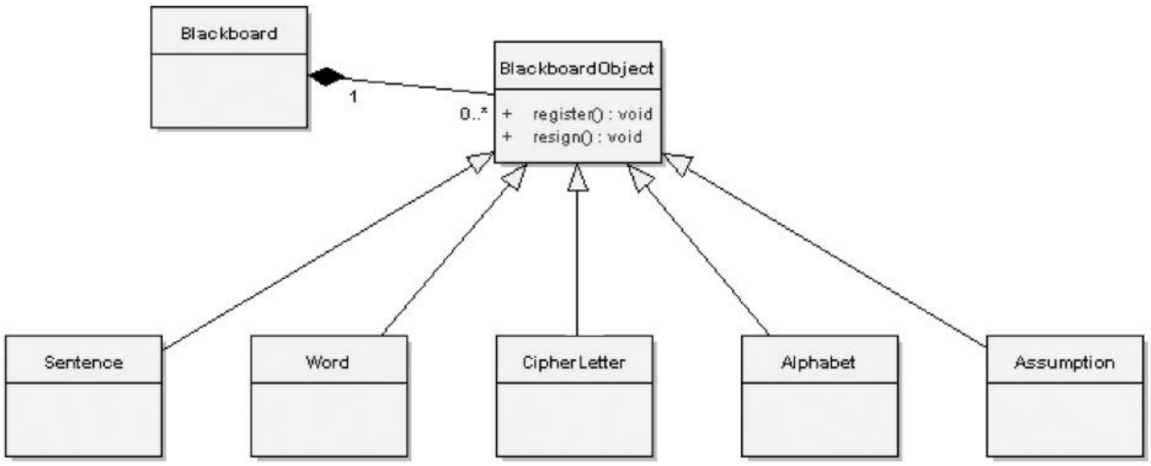


图10-2　初步的Blackboard类图设计

从BlackboardObject的外部视图来看这个类，可以定义两个有用的操作：

- register　　在黑板上增加对象
- resign　　从黑板上移除对象

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判别测试在于，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，它肯定需要和BlackboardObject协作）。事实上，每一个BlackboardObject在被贴到黑板时，应当自己意识到这一点，这是该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，可以提供一个希望的操作，在集合中的每个依赖上执行。

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，提升了架构中的可复用性和关注分离。

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Blackboard- Object）。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，我们可能做出不可改变的断言——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。将Affirmation定义为一个独立的类，从而记录下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，一个不变的映射，这意味着映射已确定，因此是不可变的。解密的过程中，知识源将做出许多假设，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Assertion参数包括在内，然后，增加以下选择器方法。

- 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 假设所针对的黑板对象
- creator 创建假设的知识源
- reason 知识源进行假设的理由
- plainLetter 假设所针对的明文字母
- cipherLetter 假设的密文字母

之所以需要这些属性，主要来源于一个假设的本质：特定的知识源对明文/密文映射做出一个假设，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做出假设的黑板对象，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，一旦做出一个断言，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creator，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，所以我们会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。

10.3 构造

在Alphabet类之后，让我们通过做一些隔离的类设计，继续设计Sentence、Word和CipherLetter类。

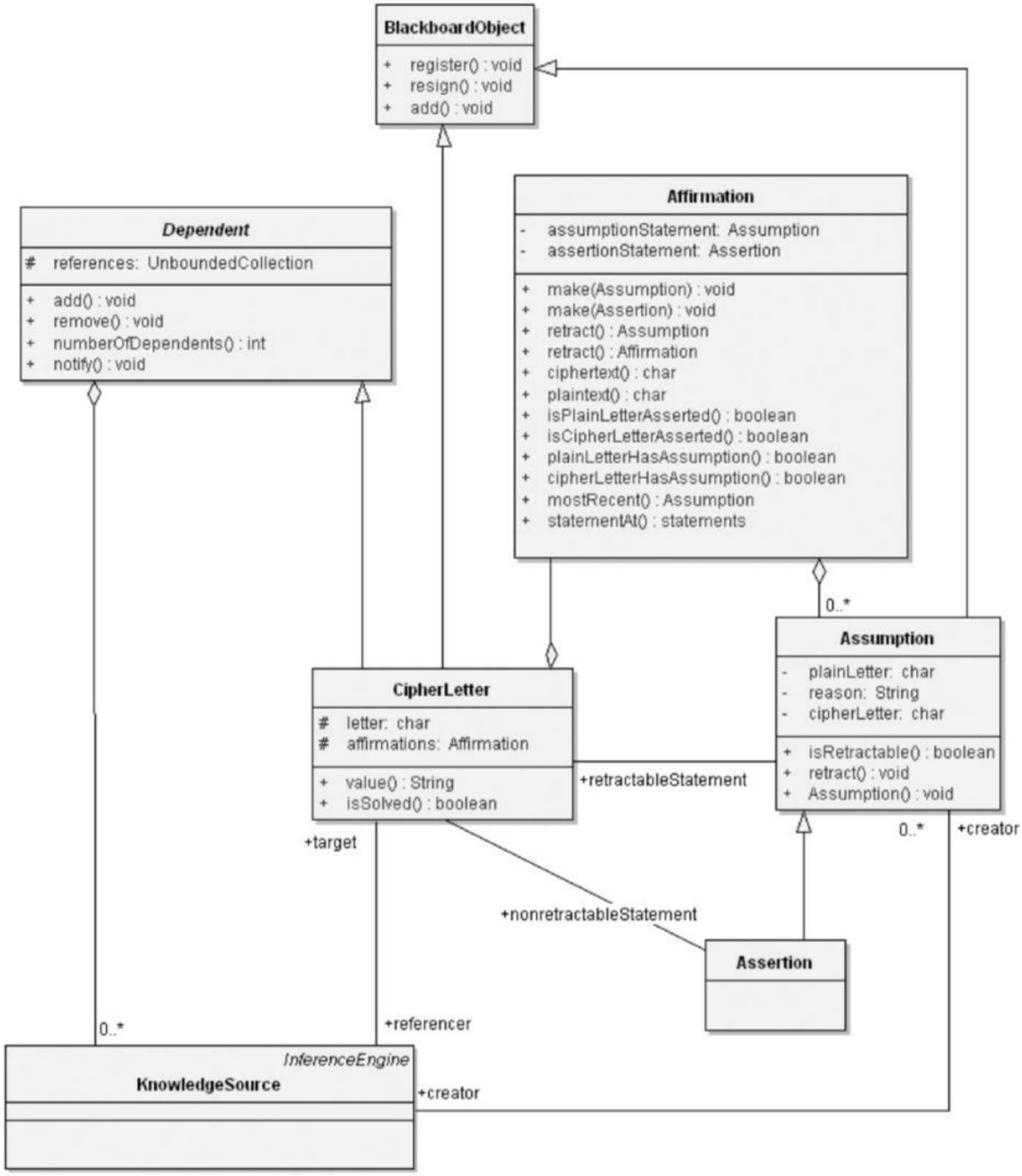


图10-3 Dependency和Affirmation类

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]（如图10-4所示），因为我们预期会有其他的Sentence子类从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。

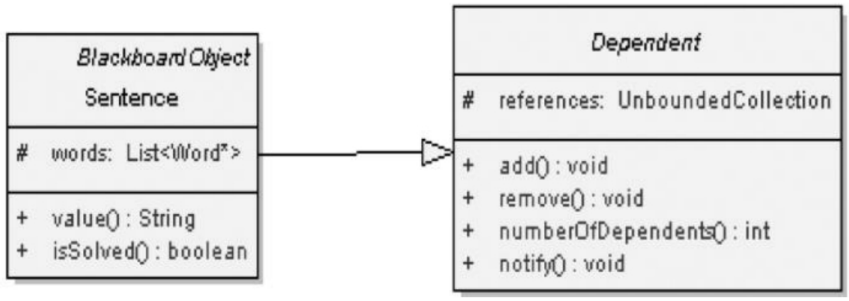


图10-4　带抽象Dependent类的Sentence类设计

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有的操作：

- value　　返回句子的当前值。
- isSolved　如果对这个句子中的所有单词都有一个断言，则返回真。

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将得到部分解。

与Sentence类一样，一个Word既是一种BlackboardObject，也是一种Dependent。而且，一个Word表示一列字母。为了辅助操纵单词的知识源，我们包含一个从单词到句子的引用，同时也包含从该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，则返回真。

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。如图10-5所示为添加了CipherLetter和Word设计之后的架构框架。

请注意，我们包含了选择器方法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，以安全方式访问它的假设和断言。

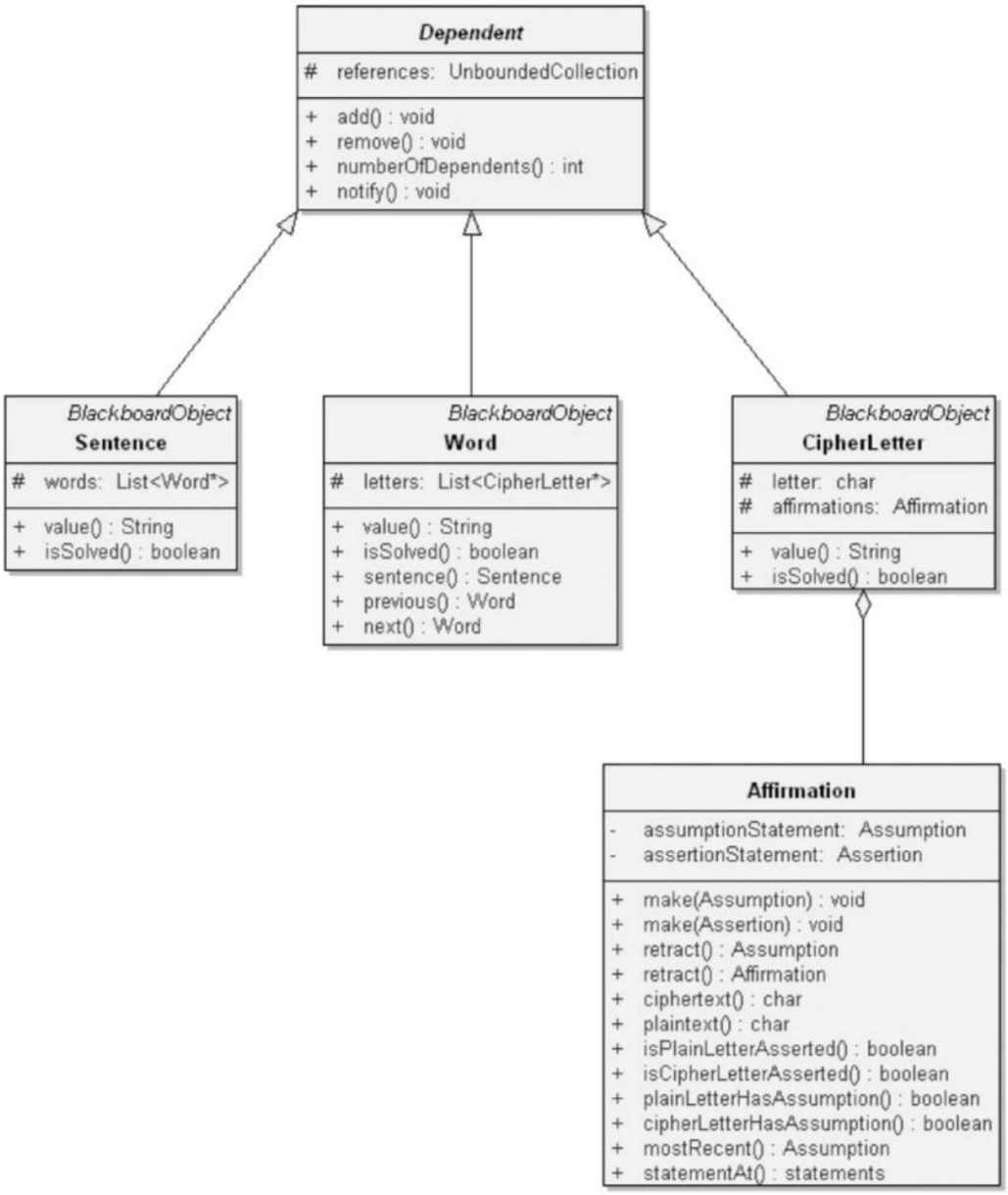


图10-5　CipherLetter和Word类的设计

关于成员对象affirmations有一点说明：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。这个决定影响了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

既然我们已经重定义了它的行为，下一步就可以对Affirmation类做出一个合理的实现决定。具体地说，可以包含保护成员对象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，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。

就像CipherLetter类一样，也包含一个保护成员对象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。这个类的职责很简单，即收集BlackboardObject类及其子类的实例。这样，可以将Blackboard设计为DynamicCollection的一种实例。我们选择从类DynamicCollection继承，而不是包含它的一个实例，因为Blackboard通过了我们对继承的测试：Blackboard的确是一种集合。

Blackboard类提供了诸如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个黑板特有的操作：

- reset　　擦干净黑板。

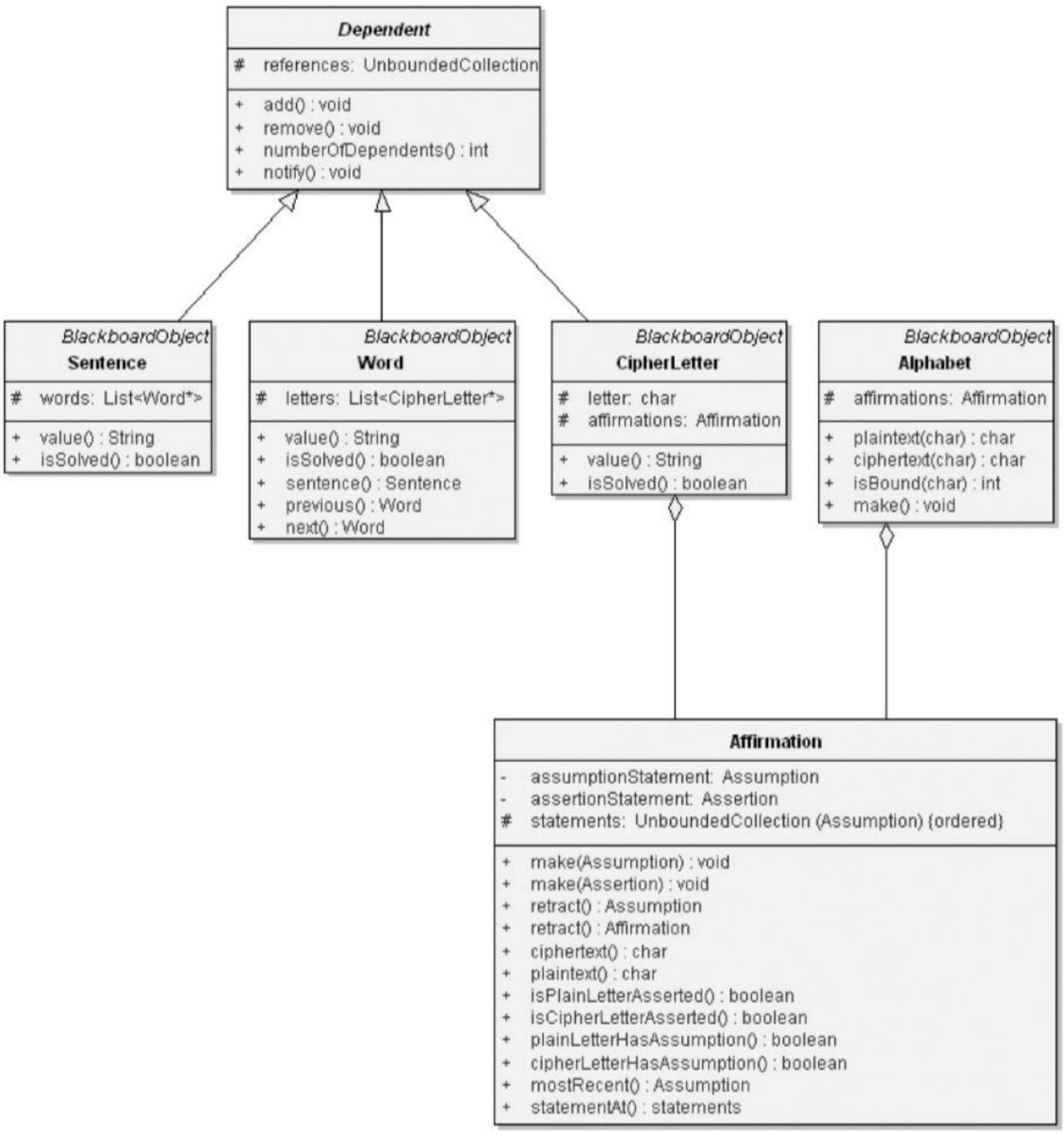


图10-6　Alphabet类的设计

- 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。

要在黑板与它的知识源之间建立依赖关系，就需要第二个操作。

在图10-7中，我们总结了与Blackboard协作的类的设计。请注意，在这张图中，Blackboard类既是从模板类DynamicCollection实例化而来，也是从它继承而来。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

表的行为仅出现在BlackboardObject的一部分子类中。通过使Dependent成为抽象的，我们增加了复用它的机会。

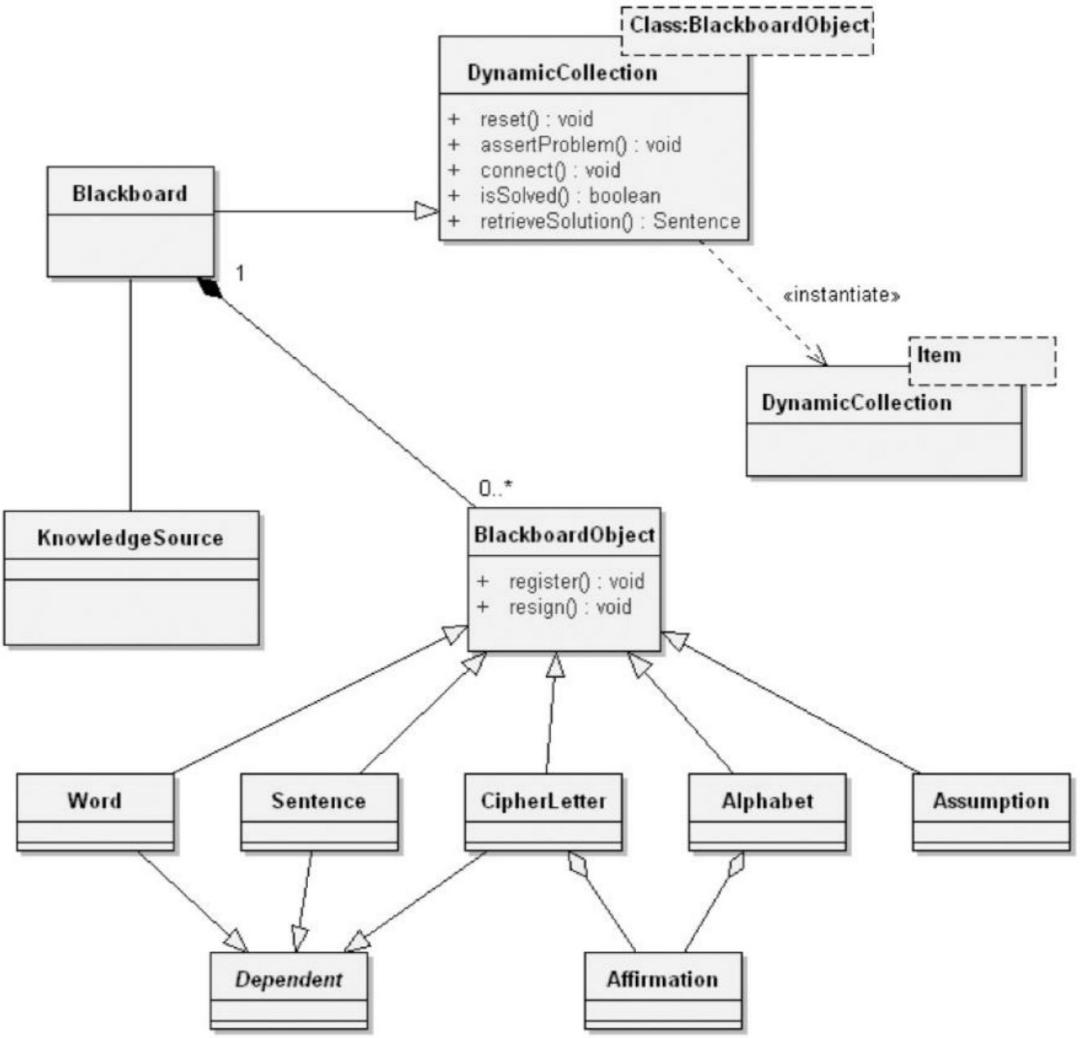


图10-7　细化后的Blackboard类图设计

10.3.2　设计知识源

在前一节，我们确定了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中。

1．设计特定知识源

假设存在一个名为KnowledgeSource的抽象类，它的目的更像BlackboardObject类。我们没有将13个知识源的每一个作为这个更通用的类的直接子类，而是首先进行领域分析，看看是否存在成簇的知识源，这样做是有用的。实际上，存在这样一些分组：一些知识源作用

于整个句子，一些作用于整个单词，一些作用于相邻的字母串，还有一些作用于单个字母。可以通过创建以下子类来记录这些设计决策：

- SentenceKnowledgeSource　　句子相关的规则
- WordKnowledgeSource　　单词相关的规则
- LetterKnowledgeSource　　字母相关的规则
- StringKnowledgeSource　　字符串相关的规则

对于每个类，我们可以提供另一个层次的规格说明。例如，类SentenceKnowledgeSource的子类包括：

- SentenceStructureKnowledgeSource　针对句子结构的规则
- SolvedKnowledgeSource　　已解决的密文句子

类似地，中间类WordKnowledgeSource的子类包括：

- WordStructureKnowledgeSource　　针对单词结构的规则
- SmallWordKnowledgeSource　　针对短单词的规则
- 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　　针对前缀的规则

- CommonSuffixKnowledgeSource 针对后缀的规则
- DoubleLetterKnowledgeSource 双字母的规则，如oo、ll等
- LegalStringKnowledgeSource 针对那些字符串合法的规则

最后，引入类LetterKnowledgeSource的以下子类：

- DirectSubstitutionKnowledgeSource 针对字母置换的规则
- VowelKnowledgeSource 针对元音的规则
- ConsonantKnowledgeSource 针对辅音的规则
- LetterFrequencyKnowledgeSource 针对字母频率的规则

图10-8展示了KnowledgeSource的层次结构。

2．泛化知识源

分析表明，仅仅两个主要的操作适用于所有这些特化类：

- reset 重新开始知识源。
- evaluate 评估黑板的状态。

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。

- 为置换密码提出假设。

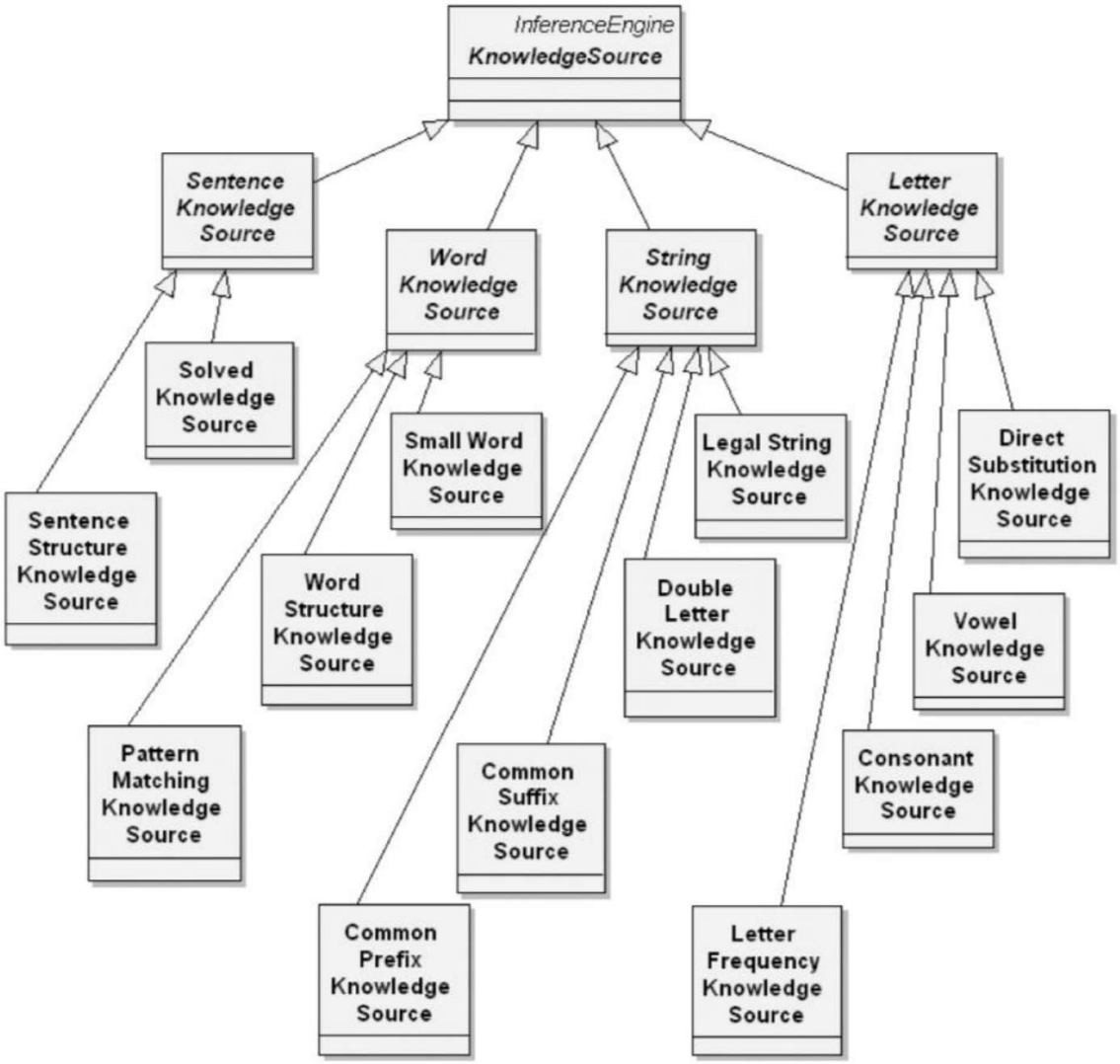


图10-8　KnowledgeSource类的泛化层次

- 在先前的假设中发现矛盾，并使矛盾的假设被收回。
- 为置换密码提出断言。
- 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，这些动作代表了一个推理引擎的行为。因此，我们创建InferenceEngine类，给定一套规则，评估那些规则，或者产生新的规则（前向链）/证明假设（后向链）。InferenceEngine构造方法的基本职责是构造这个类的一个实例，并为它配备一套用于评估的规则。

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之前列出的四个动作中的一个。图10-9展示了一个这种协作的共同场景。

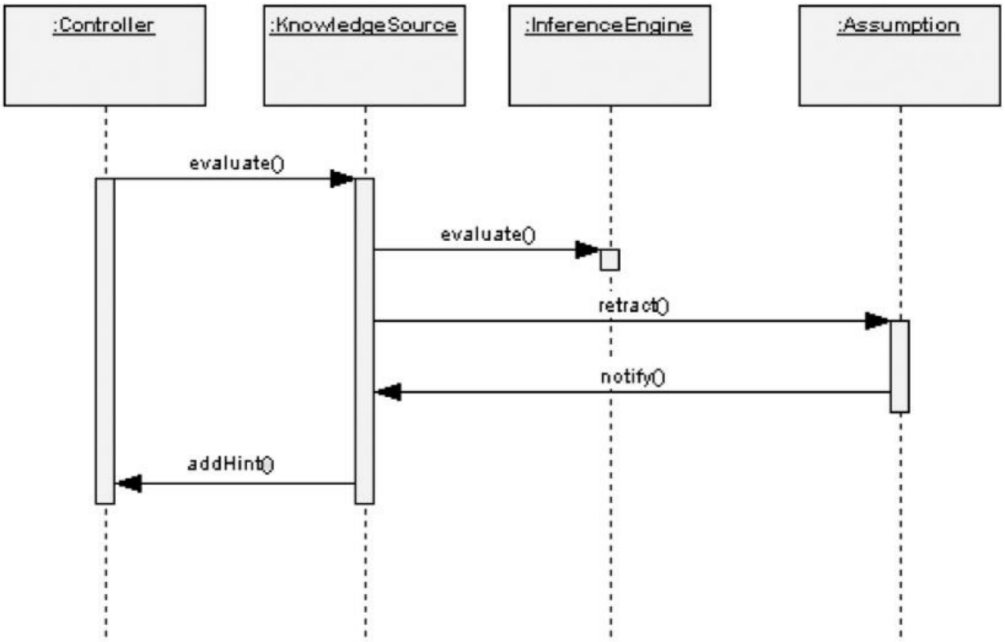


图10-9　评估知识源规则的场景

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？一条规则可能针对的是常见的后缀知识源，利用一种模式匹配算法（如*I??），来识别常见的后缀模式。

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，因为它在那里找到要操作的对象。最后，每个知识源必须与一个控制器有关联，通过发送解的提示来协作；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。

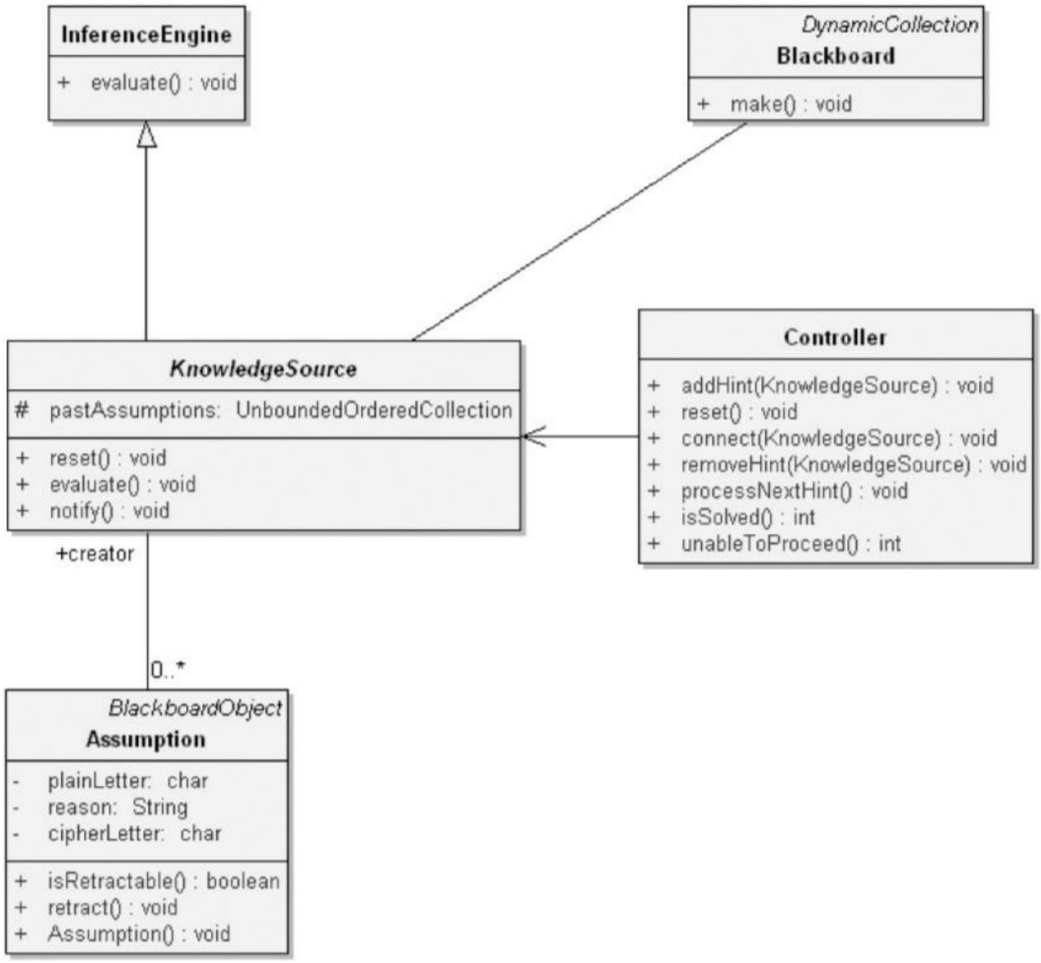


图10-10　KnowledgeSource的初步设计

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，目的是从错误中学习。

类Blackboard的实例被用作BlackboardObject的存放处。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类结构的细化设计。

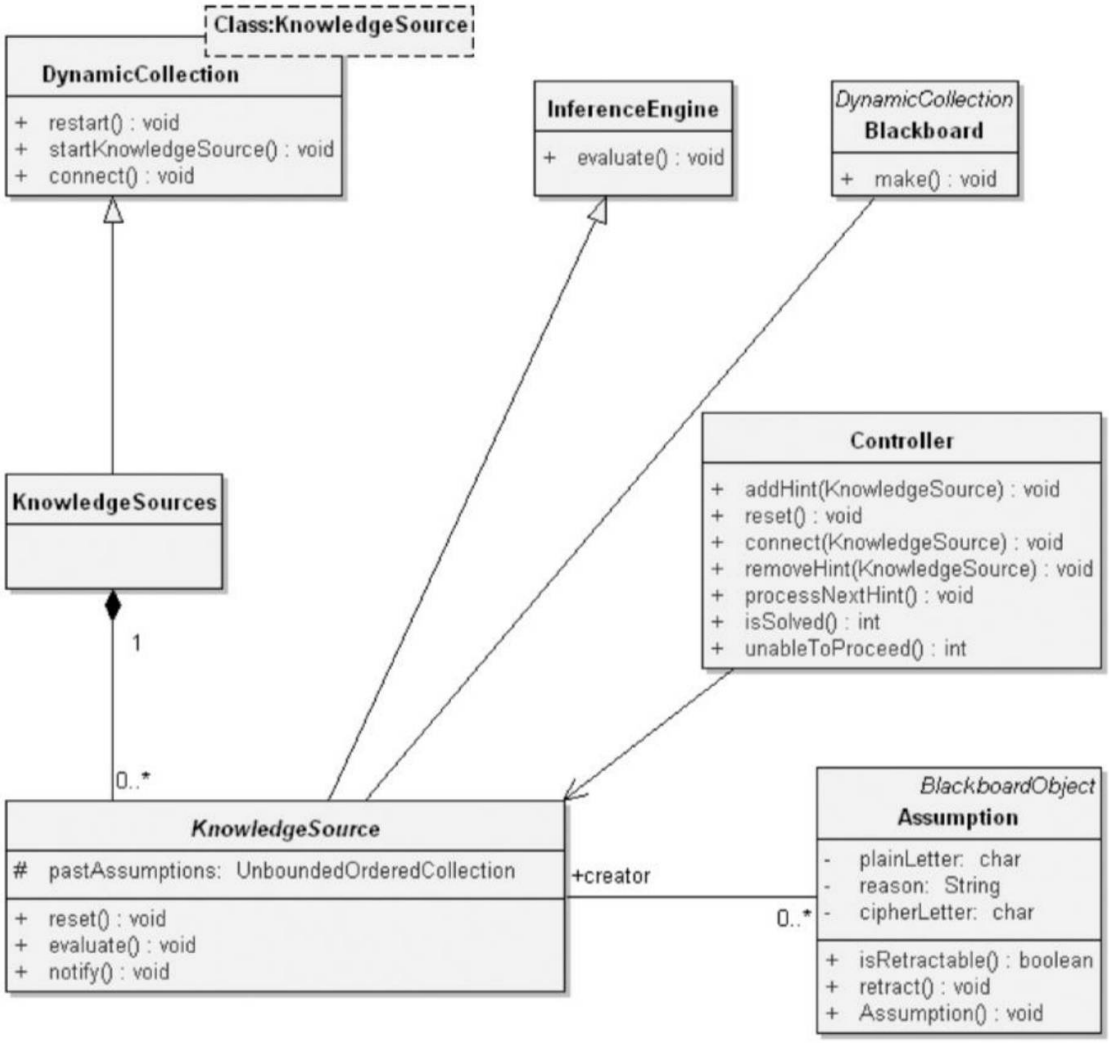


图10-11　KnowledgeSource类图的细化设计

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。一旦所有的知识源都有一个机会，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SentenceStructureKnowledge- Source更高优先级的提示。

控制器扮演着一个代理的角色，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，通过这个关联，控制器可以访问相应命名的类KnowledgeSources。另外，控制器必须将提示集作为它的一个属性，根据上述优先级规则进行排序。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。

进入更为孤立的类设计，我们为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 将控制器附加到知识源。

从某种意义上说，控制器受到从各知识源接收的提示的驱动。所以，有限状态机非常适合于描述这个类的动态行为。

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。

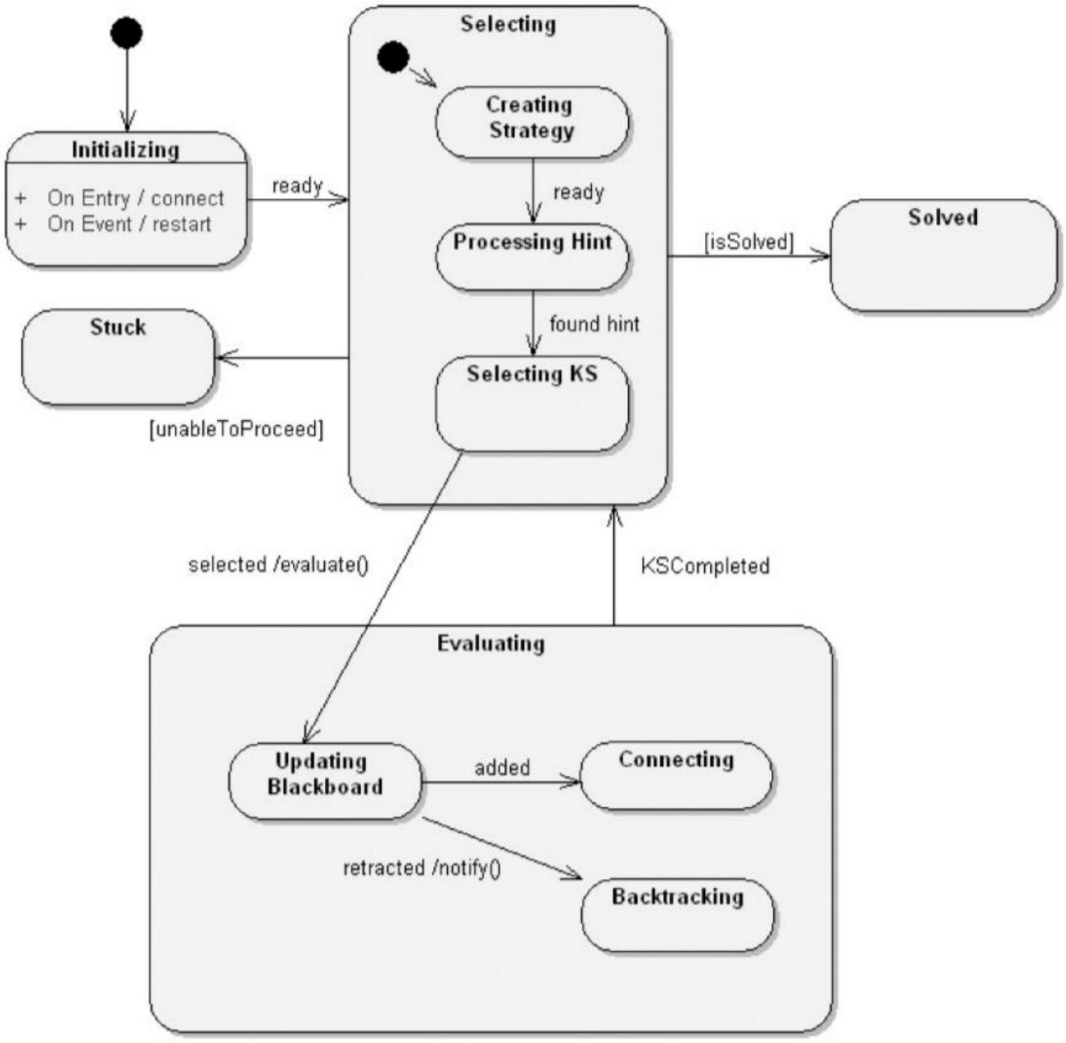


图10-12　控制器状态机

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。

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是描述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中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，展示了theBlackboard集合对黑板对象的物理包含，以及theKnowledgeSources集合对知识源的物理包含。

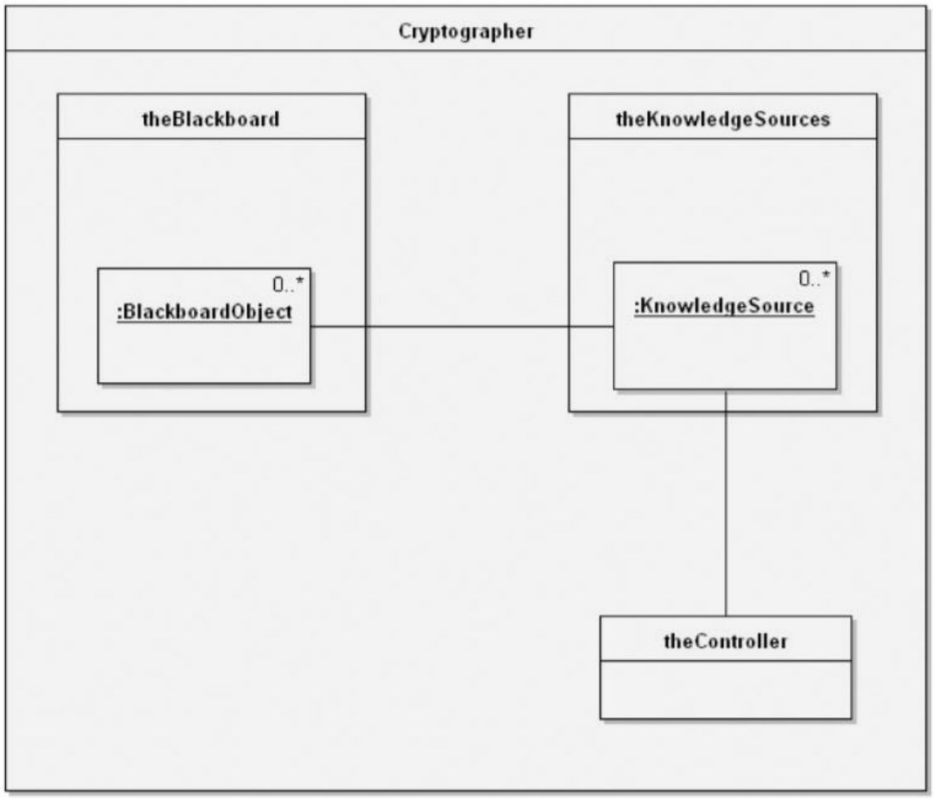


图10-13　密码分析的组合结构图

在这个图中，我们引入一个名为Cryptographer的新类的实例，这个类用做一个包含黑板、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。

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。

作为这个类的构造方法的一部分，我们要求它具有这样的行为，即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，但它的签名中包含一个字符串，我们通过该字符串提供待解的密文。按照这种方式，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。

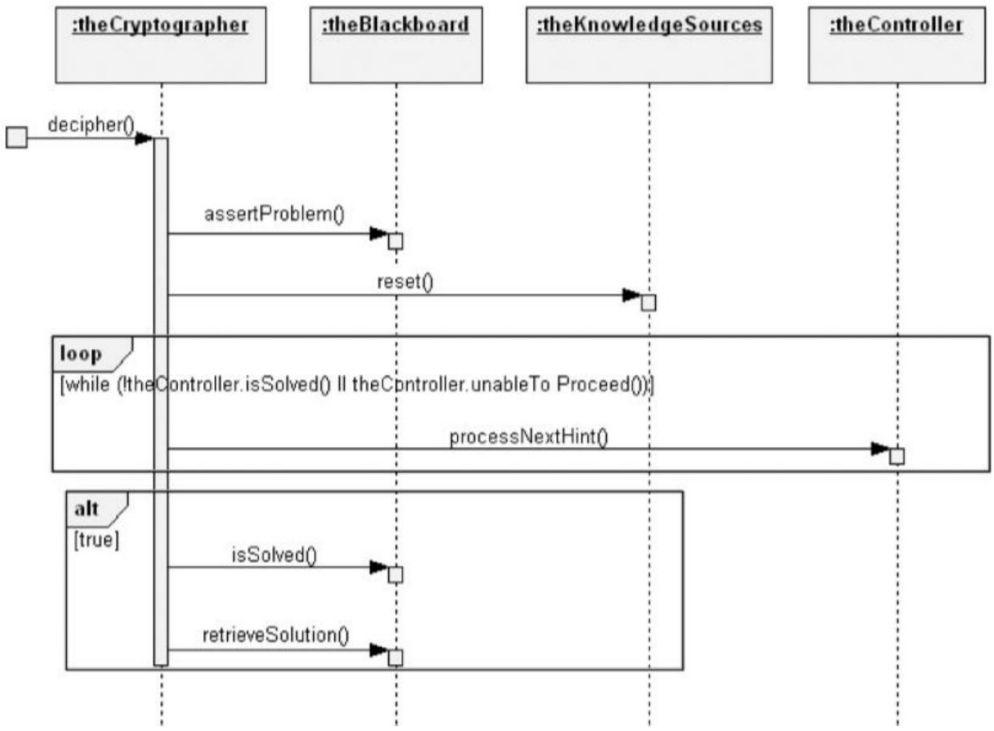


图10-14　decipher序列图

我们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。

retrieveSolution操作要简单得多，我们仅仅返回黑板上句子的值。在isSolved的结果为真之前调用retrieveSolution，将得到部分解。

2．实现假设机制

此时，我们已经实现了一些机制，让我们能够设置和检索Blackboard对象的值。下一个主要功能点是对Blackboard对象做出假设的机制。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展示了知识源陈述假设的主要场景。正如这张通信图所示，一旦KnowledgeSource创建了一个Assumption，它就通知Blackboard，接着Blackboard对它的Alphabet做出这个Assumption，同时对每个BlackboardObject应用这个Assumption。通过依赖机制，受影响的BlackboardObject进而可以通知它所依赖的KnowledgeSource。

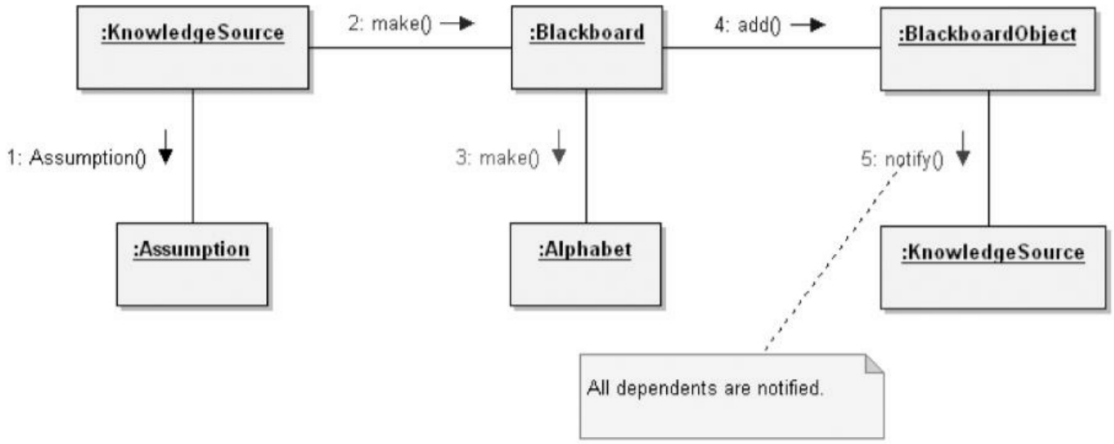


图10-15　假设机制

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，直至包括打算撤回的假设。通过这种方式，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，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，可能不必撤销第二个假设。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，因为增加这个行为不会对架构产生影响。

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，并且陈述和撤回假设的机制也已经生效，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）如果我们实现并测试较小的相关规则集合，而不是试图一次性地测试所有规则，那么调试知识源就要容易得多。

从根本上说，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，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，或许还有一些规则缺失了。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。

在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StructureKnowledge- Source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 交付之后

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存放处中。

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。

例如，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，有五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。

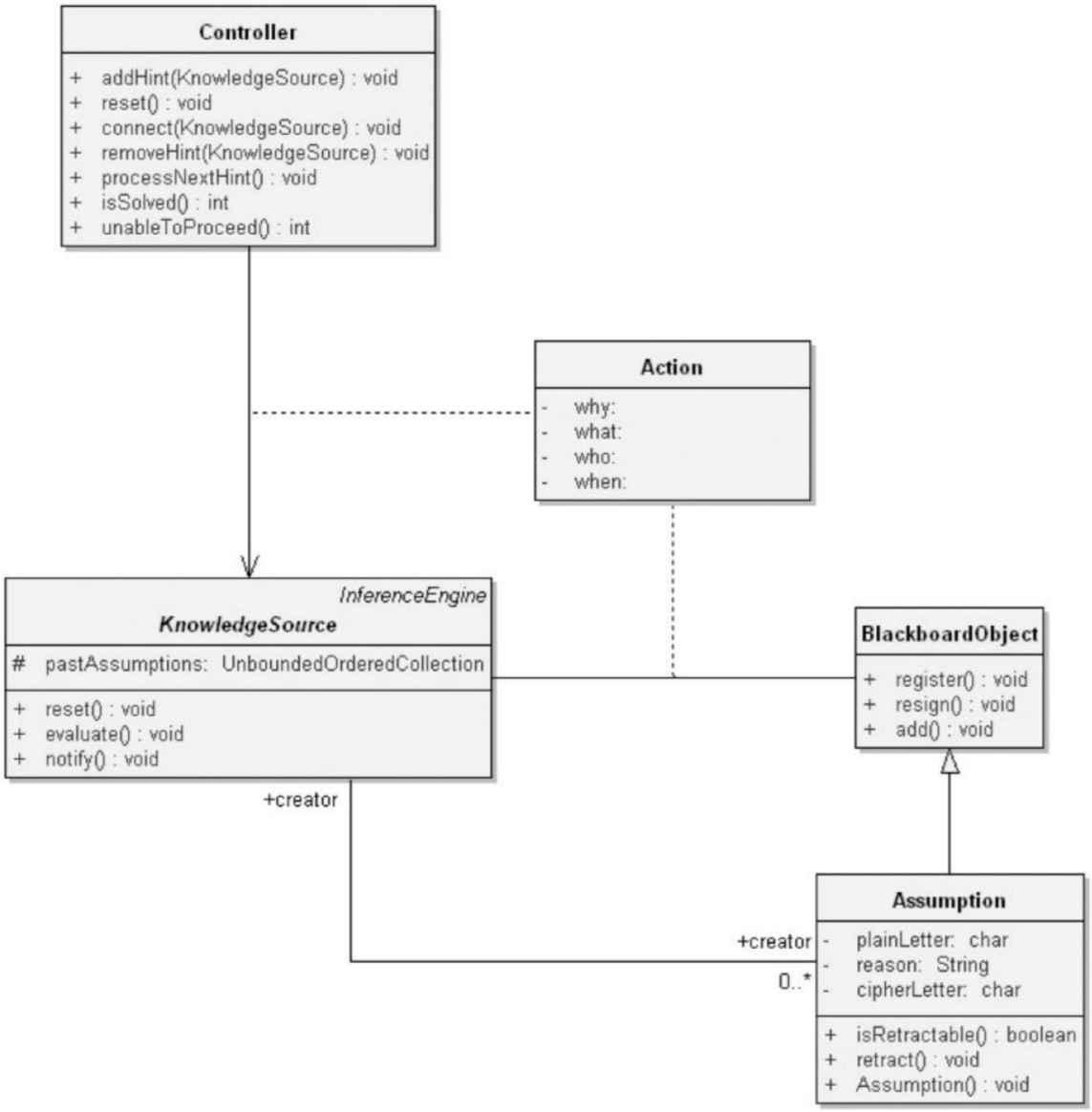


图10-16　通过Action类设计提供的附加功能

- 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，这项任务有点累人，但没有火箭科学那么可怕。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各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。

第一个变更相当容易，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，它会增加一些新类，使用已有的设施，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前沿。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，而不用对现有的类进行重大修改。与其他的改变一样，这个改变可以在已有的设施上构建。

---

[1]在UML 2.0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。

第11章　数据采集——气象监测站

对于很多科学系统，自动收集数据通常利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合适的传感器，将测量的参数转换为电子信号，再由数据采集硬件采集。所开发的控制软件对信号进行解释，以便分析和显示。

使用面向对象技术来设计一个数据采集系统，让我们能够将测量及收集数据的硬件与分析信息的应用进行隔离。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。应用接口就像覆盖在硬件上的皮肤，可以将测量设备与处理信息的应用实现隔离。在本章的案例中，我们提供了一个数据采集系统的例子，即一个气象监测系统。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 初始

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，面向对象的新手可能很想采用一种本质上非面向对象的方式来解决这个问题，即考虑数据流和不同的输入/输出之间的映射。然而，正如我们将要看到的那样，即使是像这样小的一个系统，也可以很好地借鉴面向对象架构，并在开发过程中展示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小时内四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。用户可以通过小键盘来指挥系统，让它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的值来校正传感器，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。为了限定问题边界，以便展示软件分析设计问题，我们做以下战略性的假定：

- 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。
- 通过远程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（推动计数器对回转进行计数）的标柱来测量风向和风速。
- 通过小键盘提供用户输入。
- 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职责。[1]这样，我们就可以决定，这个类负责追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职责转变为两个服务，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：

表示当前的月、日和年。

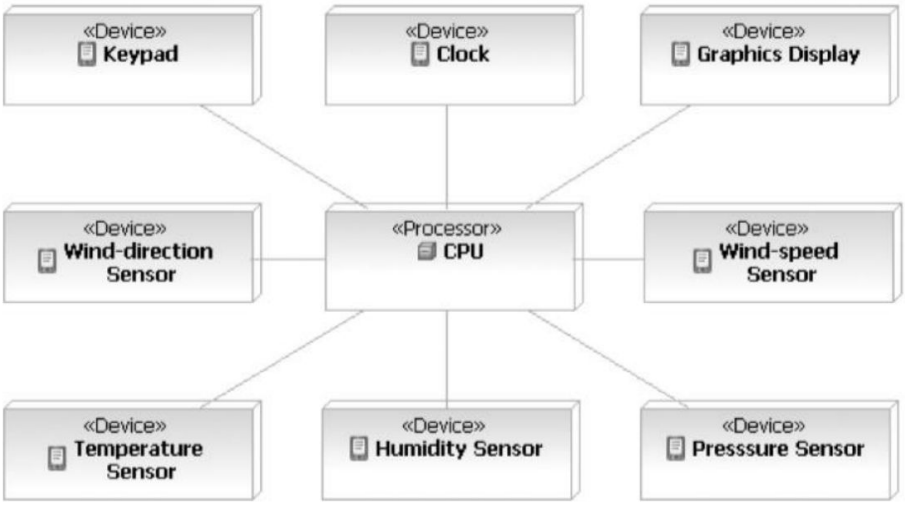


图11-1　气象监测系统部署图

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，然后将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，其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跨度一定不能太大，不需要一个厚重而低效的实现来粘合它们。

因此，时间和日期类的职责必须包括设定日期和时间。完成这个职责需要一组新的服务，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

职责：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，这一点可以从如图11-2所示的状态转换图中看出。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。

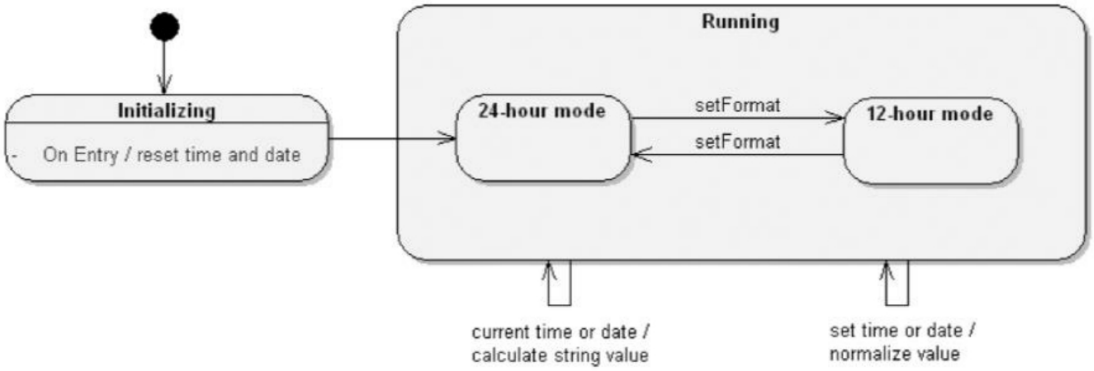


图11-2　类TimeDate的生命周期

我们已经足够详细地说明了这个抽象的行为，所以能够将它提供给其他客户的场景使用，这些客户是在分析期间可能发现的。在考虑这些场景之前，我们先说明一下本系统中其他实体对象的行为。

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

职责：跟踪当前温度。

操作：

currentTemperature

setLowTemperature

setHighTemperature

属性：

temperature

操作currentTemperature的含义不言自明。另外两个操作直接来源于我们的需求，该需求要求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符合已知的实际值，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。

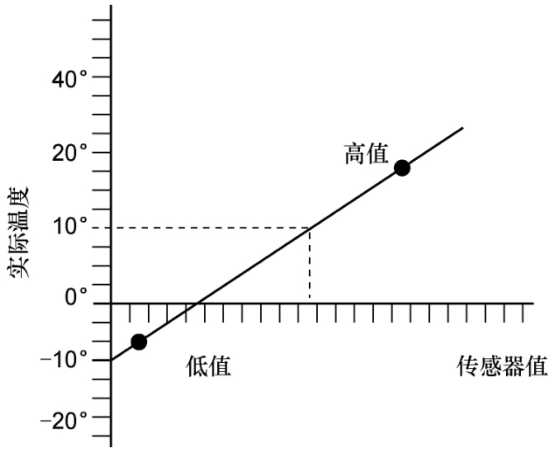


图11-3　Temperature传感器校正

细心的读者可能会纳闷，既然需求表明系统中只有一个温度传感器，为什么还要抽象出这个类？确实如此，但是预料到要复用这个抽象，我们选择将它设计为一个类，这样可以解除它与这个特定系统的耦合。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，可以让这类系统的其他程序简单地操作任意数目的传感器。

我们可以在下列规格说明中表示气压传感器的抽象。

类名：Pressure Sensor

职责：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，需求要求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性质，最重要的是，决定什么抽象应该负责该行为。

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。[2]因此，我们在这些类中增加以下的职责和其相应的操作。

职责：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。

为了全面起见，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。前面的决策是把这个共同行为作为传感器类本身的一个职责。我们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，没有必要，所以我们拒绝采用。我们原来对Temperature Sensor和Pressure Sensor类所做的规格说明表明，每个抽象都有足够的知识来执行这种趋势报告的行为，通过将职责合并（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。

湿度传感器的抽象可以表示为下面的规格说明。

类名：Humidity Sensor

职责：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没有计算其趋势的职责，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职责，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

职责：报告过去24小时内的最高和最低值。

操作：

highValue

lowValue

timeOfHighValue

timeOfLowValue

Humidity Sensor是Historical Sensor的直接子类，Trend Sensor也是。Trend Sensor是一个中间抽象类，它在Historical Sensor抽象与具体类Temperature Sensor及Pressure Sensor之间搭起桥梁。

风速传感器的抽象可以表示为下面的规格说明。

类名：WindSpeed Sensor

职责：跟踪当前风速。

操作：

currentSpeed

setLowSpeed

setHighSpeed

属性：

speed

我们的需求表明，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。

对上面四个具体类（Temperature Sensor、Pressure Sensor、Humidity Sensor和WindSpeed Sensor）进行快速的领域分析，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四个类中，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

职责：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。[3]

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。这个实体的抽象可以表示为下面的规格说明。

类名：WindDirection Sensor

职责：跟踪当前风向，表示为罗盘图上的点。

操作：currentDirection

属性：direction

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。

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。

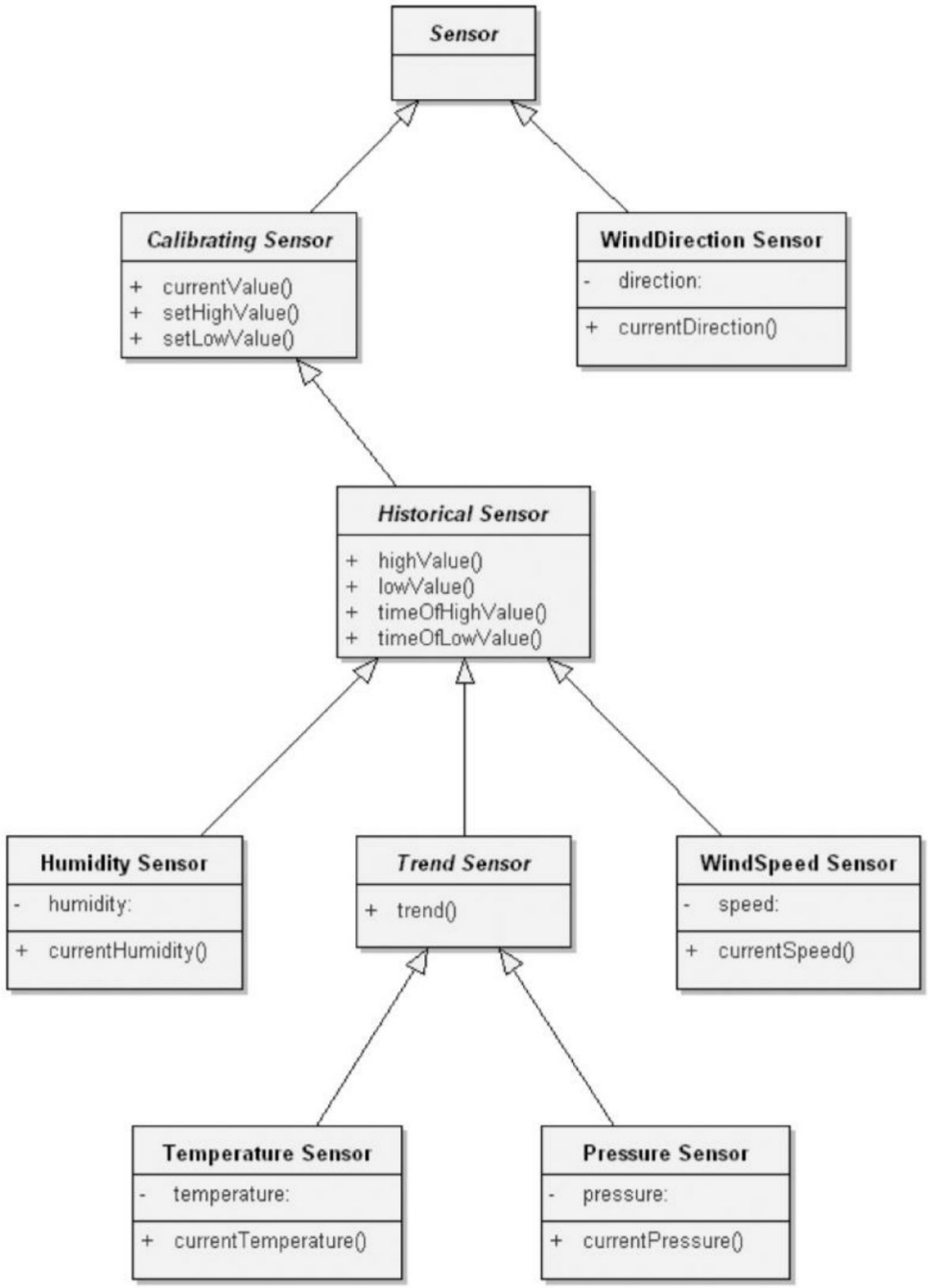


图11-4　Sensor类的层次结构

类名：Keypad

职责：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。我们将解释这些键的含义的职责委

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。

LCD Device类的抽象可以将软件与可能使用的特定硬件隔离。为了解除软件与可能使用的特定图形硬件之间的耦合，分析促使我们去做气象监测系统的一些常用显示画面的原型，然后确定界面需求。

图11-5提供了这样一个原型。在这个原型中，我们省略了风冷度和露点的显示需求，也省略一些细节，例如，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。但是，出现了某些模式：我们只需要显示文本（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，我们还注意到，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。

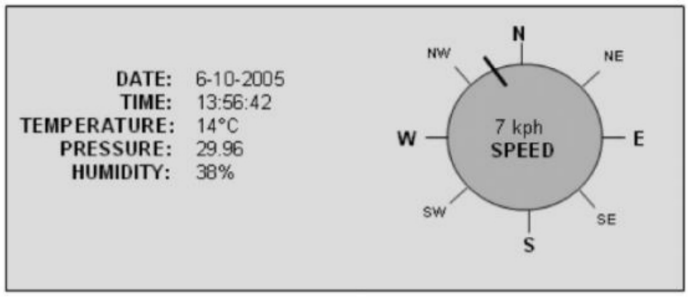


图11-5　气象监测系统显示面板

可以将这些需求转换成以下的类规格说明。

类名：LCD Device

职责：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。这种关注点分离留给我们松耦合的抽象（这正是我们想要

的），但是这需要我们找到一个代理，负责协调原始的传感器和显示器。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。我们将做出一个简化的假定，每个系统中有且只有一个定时器，它每间隔1/60秒向计算机发出中断，调用一个中断服务例程。这个细节特别令人讨厌，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。我们可以设计一个类，它使用回调函数，并且仅仅提供静态成员（这样就限制了系统中只有一个定时器）。

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与它的客户协作：首先，客户提供一个回调函数，然后每间隔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职责比较简单，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。

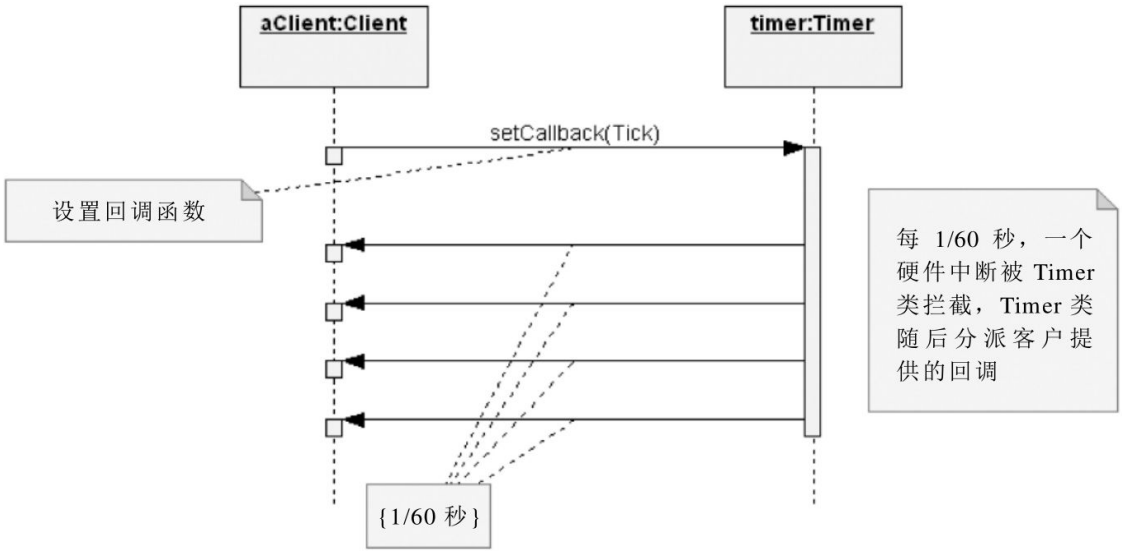


图11-6　Timer交互图

由于要截取定时事件，Timer类应当是一个主动的抽象，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

职责：截取定时事件，并相应地调用一个回调函数。

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。

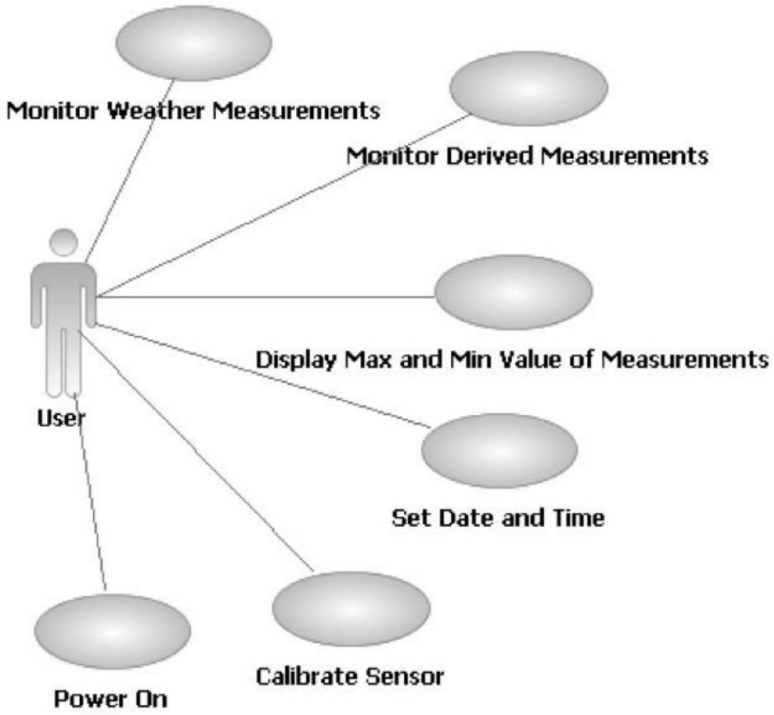


图11-7 气象监测系统的主要用例

- 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 细化

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的改变要慢得多。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职责。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，这属于设计问题而不是分析问题）。如图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。

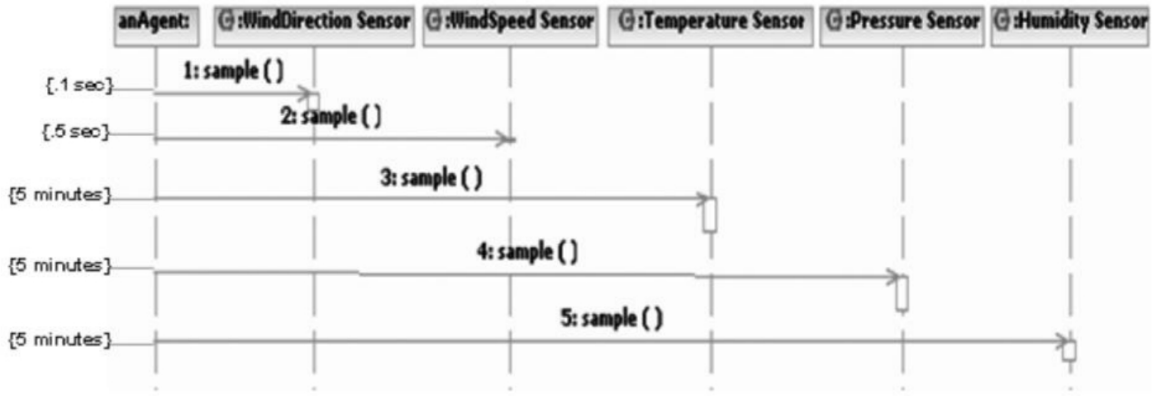


图11-8　用于监测基本测量数据的场景

要继续这个场景，我们必须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。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架构中常用的模式），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。[4]这样，就可把下面的类规格说明加入分析产物中。

类名：Display Manager

职责：管理LCD设备上显示项的布局。

操作：

drawStaticItems
displayTime
displayDate
displayTemperature
displayHumidity
displayPressure
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。我们也假定操作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，展示了协作完成这个场景所必需的抽象。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。

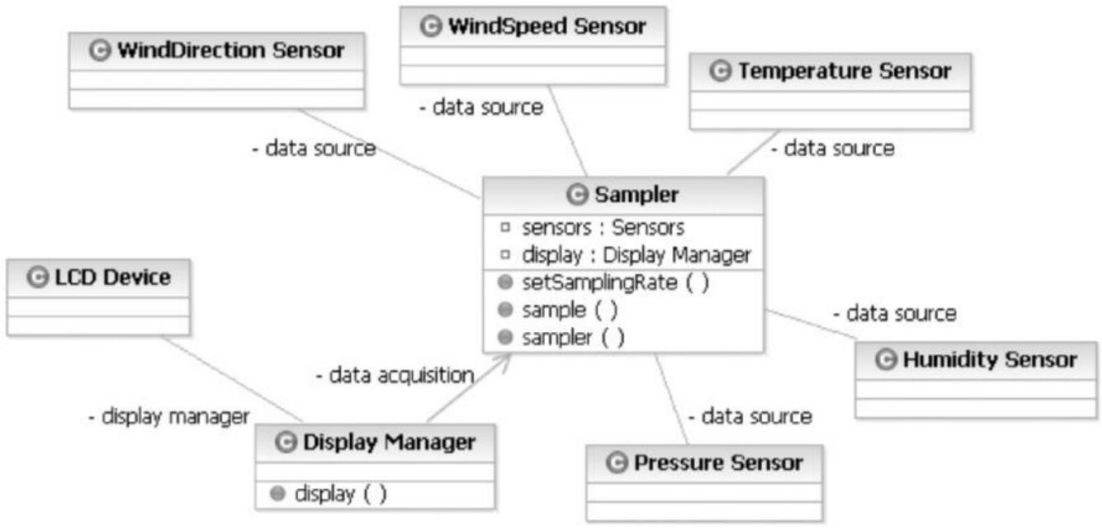


图11-9　Sampler和Display Manager类

决定在系统中包含一个Display Manager类，有一个重要的附带作用。[5]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职责，使它们的实例在构造自己时处于一个已知的稳定状态。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：

■　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。

我们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，即Wind Chill和Dew Point，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职责。具体地说，Wind Chill与Temperature Sensor和WindSpeed Sensor协作，Dew Point与Temperature Sensor和Humidity

Sensor协作。而Wind Chill和Dew Point与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。

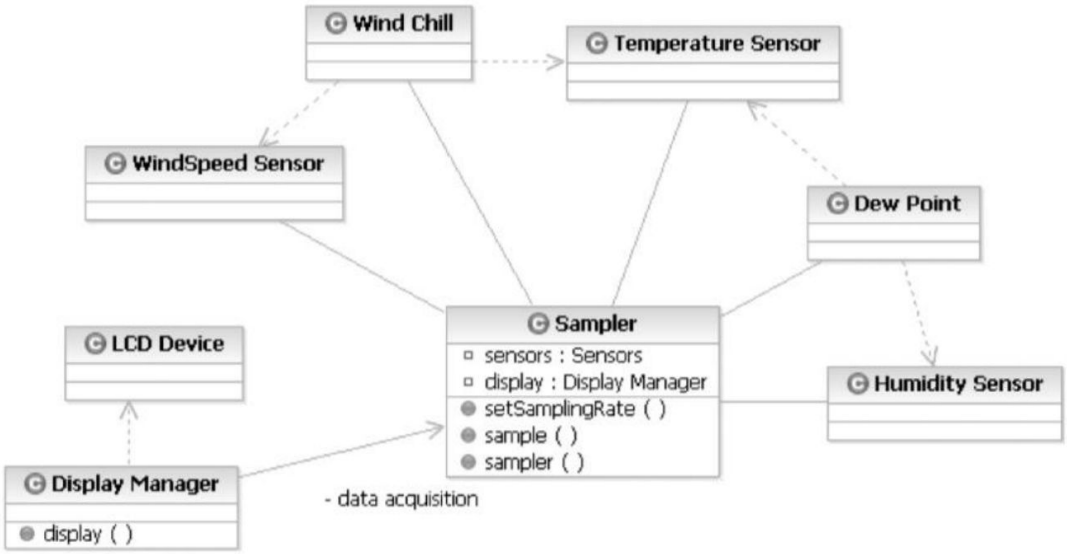


图11-10　导出测量数据的类

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的判别测试：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提取出来，因为对于客户它们呈现出清晰的契约，相对于其他抽象，它们各自提供清晰的关注点分离。

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。与设计一个图形用户界面一样，决定用户与这样的嵌入式控制器进行交互的恰当动作，也是一门艺术。全面描述如何设计这样的用户界面超出了本书的范围，但这里提供给软件分析人员的基本信息是原型有效，并且确实从根本上有助于缓解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，不必破坏设计的结构。

请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

描述：这个用例显示所选测量数据的最高值和最低值。

基本步骤：

（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（它在LCD设备上显示一条文本消息）。

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

描述：这个用例设置日期和时间。

基本步骤：

（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

描述：这个用例用于校正传感器。

基本步骤：

（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是关于设置测量单位的。

用例名：Set Unit of Measurement

描述：这个用例用于设置温度和风速的测量单位。

基本步骤：

（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，可以确定小键盘上按键的排列（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

这些用户界面场景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

职责：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。

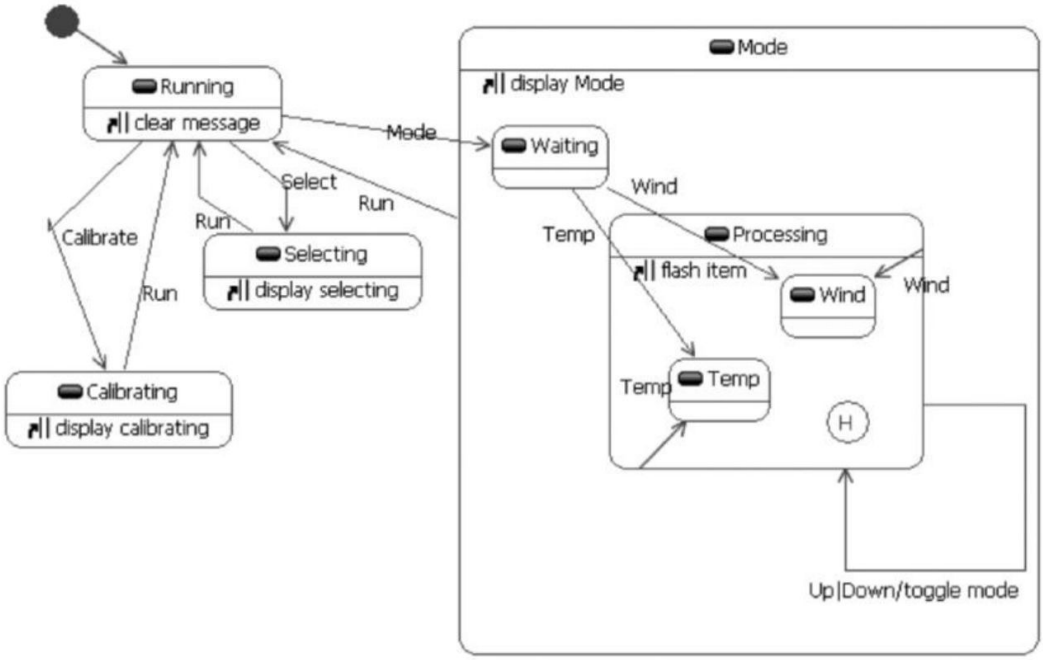


图11-12　InputManager状态机图

我们展开了Mode状态，以展示如何更正规地表达场景的动态语义。当进入这个状态时，进入动作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）。

类似地，Selecting和Calibrating状态也可以展开，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，不会揭示出特别有用的信息。[6]

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做出分析后编写的脚本。

用例名：Power On

描述：启动系统。

基本步骤：

（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职责，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（可以把它想象成描述系统架构的精彩片段（sound bite），不过针对的是软件），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，也许可以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。[7] 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。

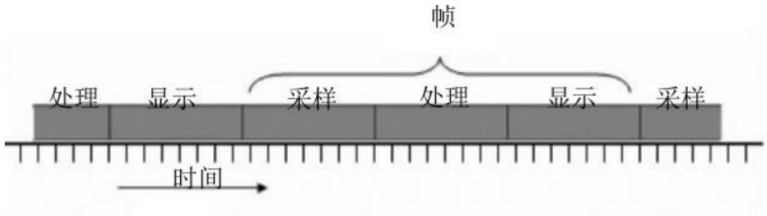


图11-13　时间帧处理

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。

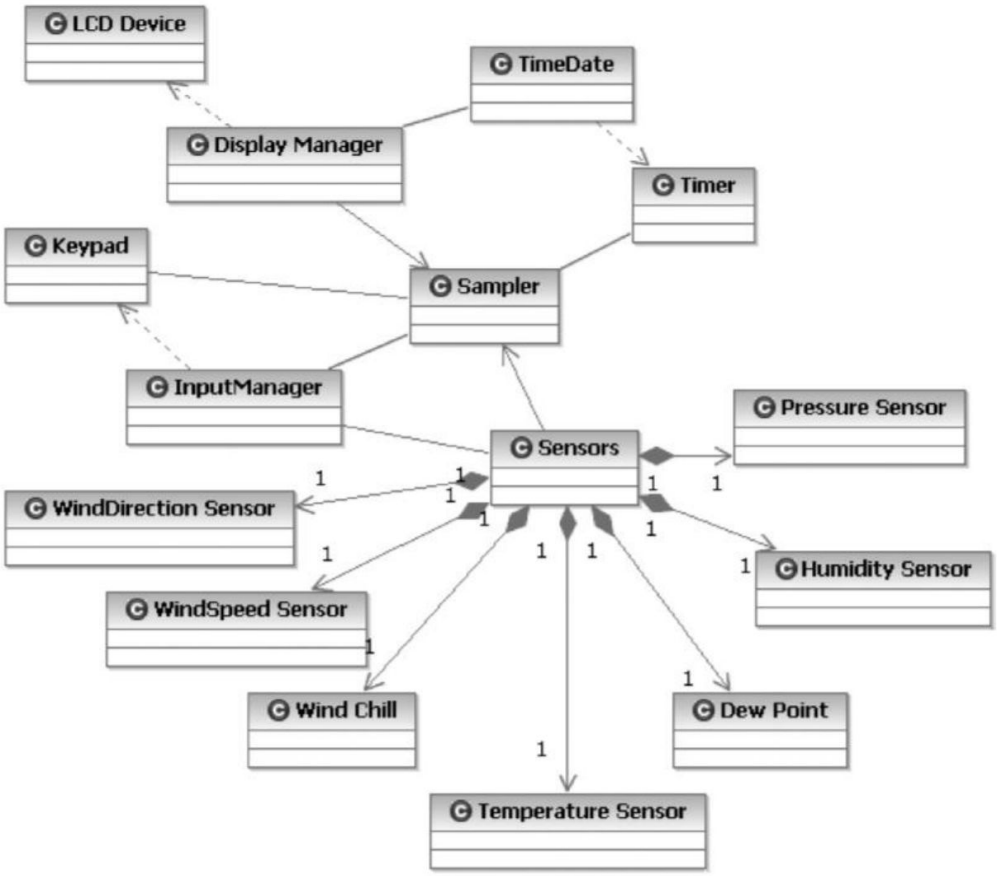


图11-14　气象监测系统的架构

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

我们从细化类Timer的接口开始介绍，该类调用回调函数。图11-15展示了类设计。

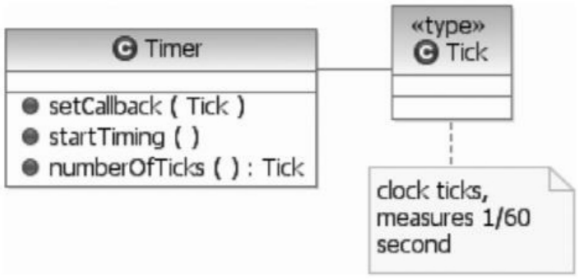


图11-15 Timer类的设计

Timer是一个与众不同的类，但要记住，它拥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秒调用该回调函数。注意，这里引入了一个显式的启动操作，因为在声明的细化过程中，我们不能依靠任何特定的“依赖于实现”的次序。

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。

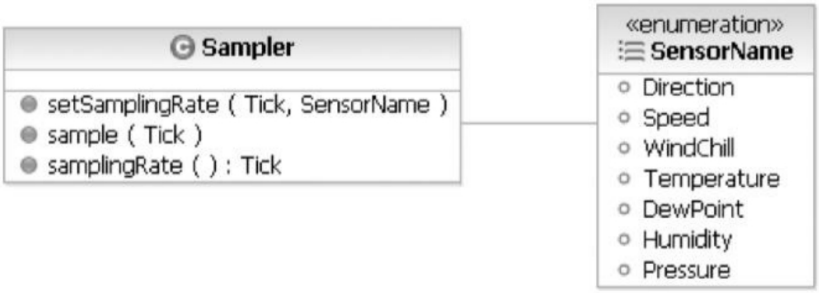


图11-16 Sampler类的接口

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）。[8]

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。

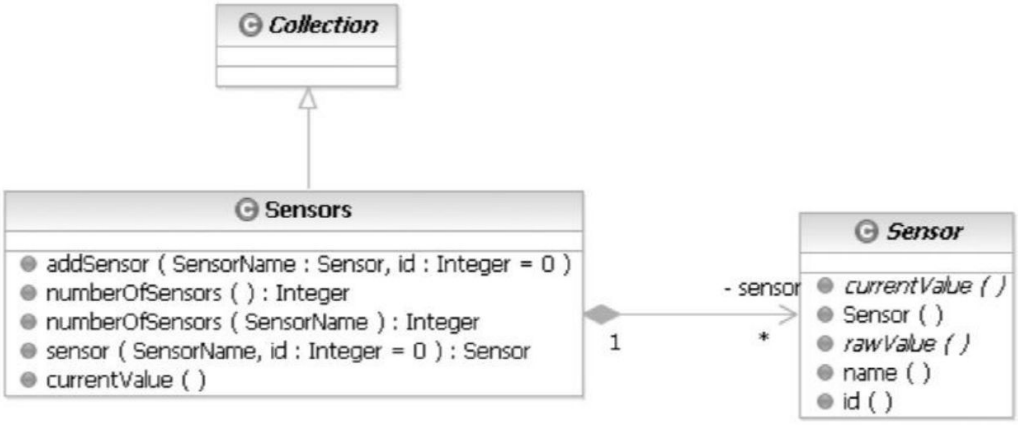


图11-17　Sensors类的接口

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。[9]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，传感器只能增加而不能从集合中删除。

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。

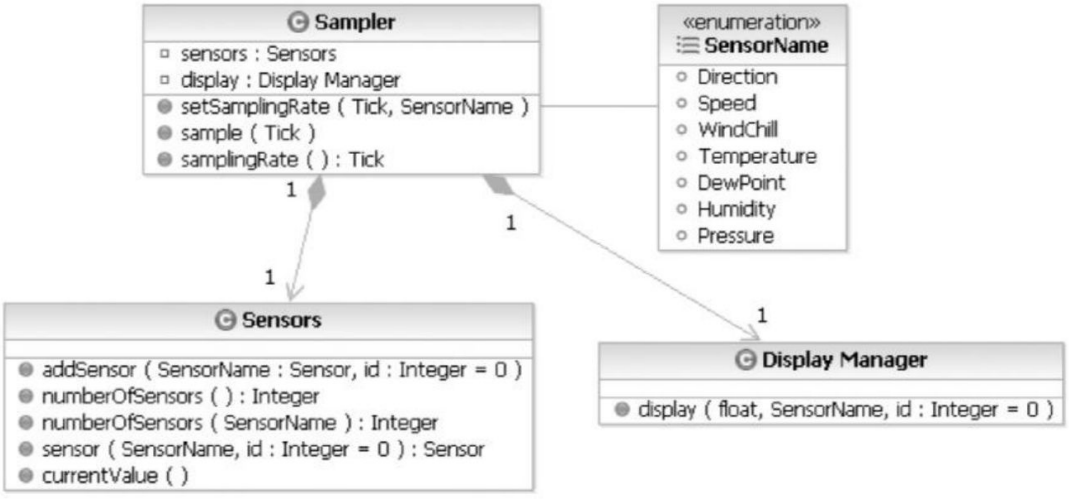


图11-18　Sampler类的设计

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。[10]

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）。这个操作也依赖于为类Display Manager而定义的display操作。

现在我们已经细化了架构的这个元素，图11-19是突出显示这个框架机制的新类图。

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。

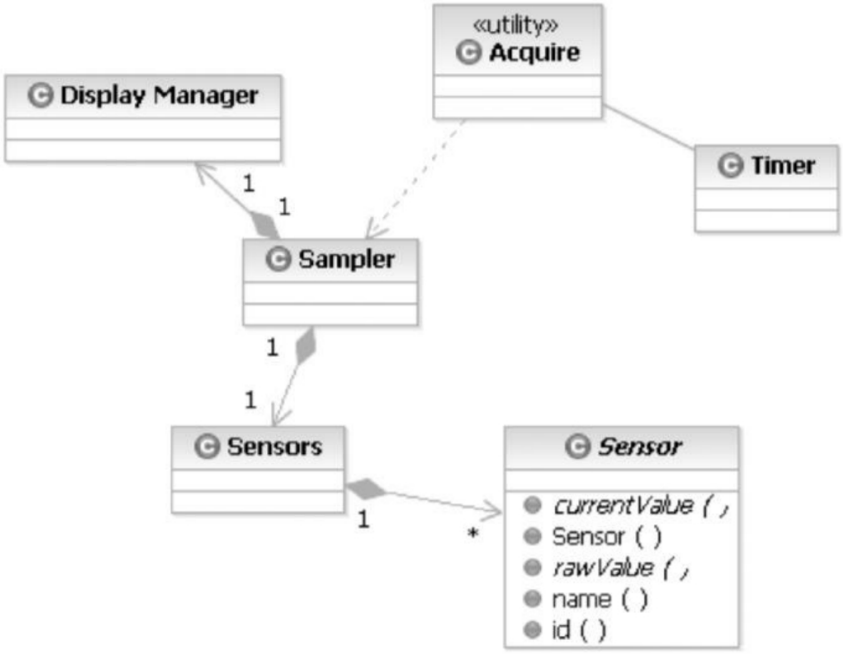


图11-19　框架机制

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，这样可以迫使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确的角色和职责。这种做法也带给我们一些早期的反馈，因为我们可以操作一个可执行系统。这种强迫早些得到结果的方法有许多技术和社会方面的效益。在技术方面，它迫使我们开始将系统中的硬件与软件部分衔接在一起，这样就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，得到对系统的观感的早期反馈。

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，我们转向那些后续发布版本的要素，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。

在开发周期的这个时候，第一次展示在图11-4的类层次仍然保持稳定，但是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）。

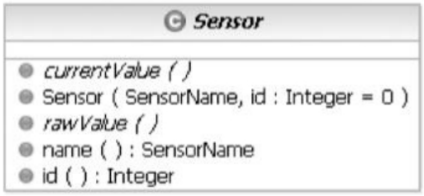


图11-20　Sensor类的设计

注意，通过类构造器，我们让这个类的实例知道它们的名字和ID。这实质上是一种运行时类型标识，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职责，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，因为它简化了某些跨类接口。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）。

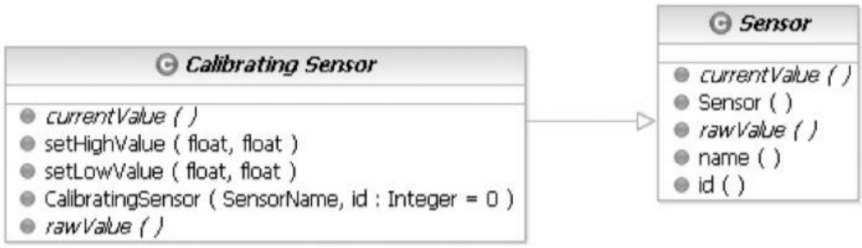


图11-21　Calibrating Sensor类的设计

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）。

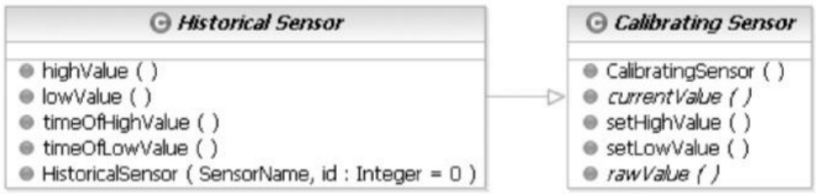


图11-22　Historical Sensor类的设计

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职责。

Trend Sensor继承自Historical Sensor，并增加了一个新职责（见图11-23）。

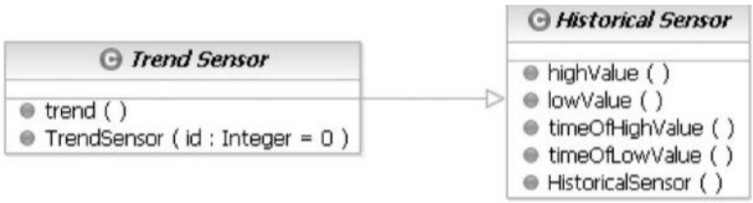


图11-23　Trend Sensor类的设计

Trend Sensor引入了一个新函数。正如一些中间类增加的新操作那样，我们将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）。

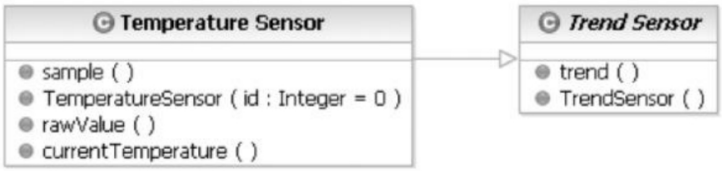


图11-24　Temperature Sensor类的设计

注意，这个类的构造器的签名与它的超类稍有不同。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即可。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），得到如图11-25所示的具体接口。

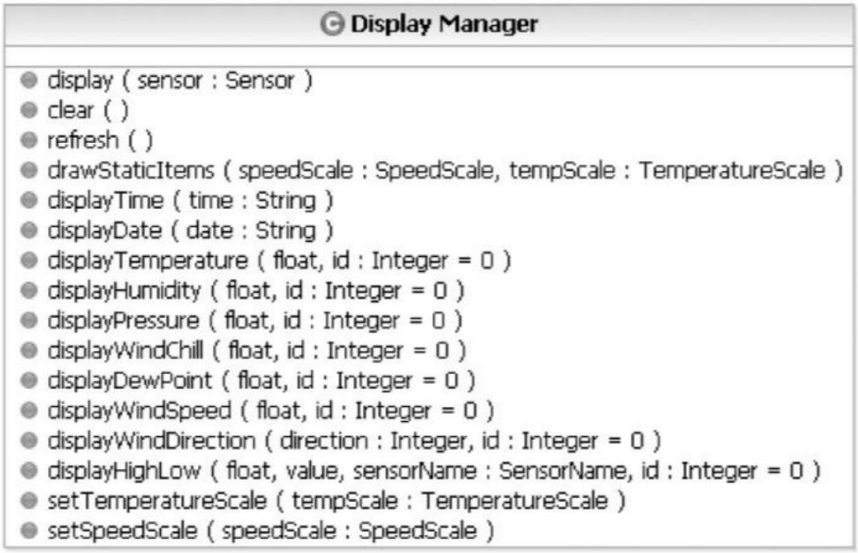


图11-25　Display Manager接口的设计

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。

Display Manager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肤。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肤，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。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。

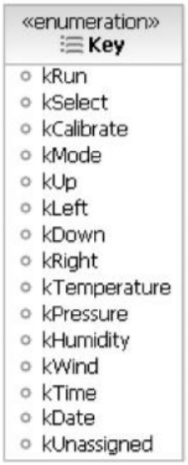


图11-26 Key枚举类的设计

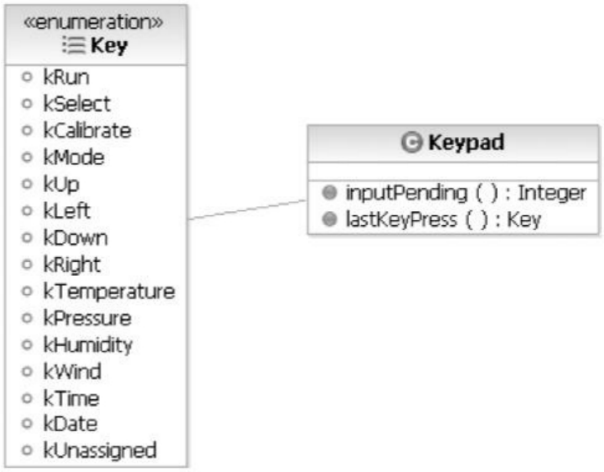


图11-27　Keypad类的设计

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有一个类似的简单接口，如图11-28所示。

正如我们将看到的，这个类大多数有趣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

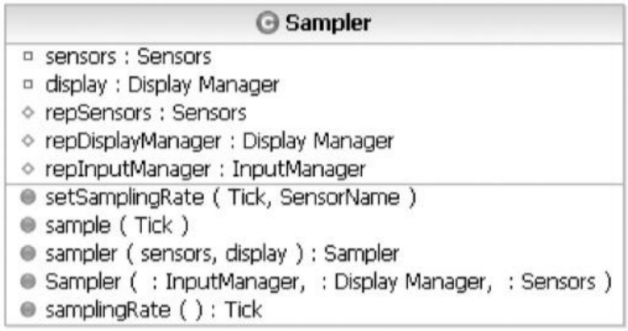


图11-29　Sampler接口修改后的设计

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）。

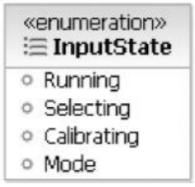


图11-30　InputState枚举类的设计

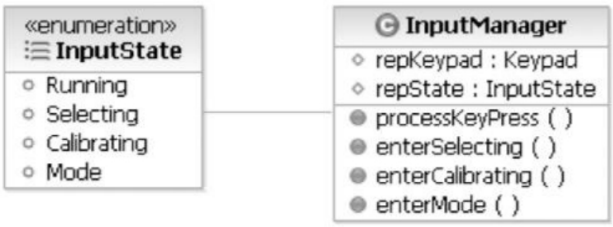


图11-31　带InputState的InputManager类设计

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如图11-12所示的状态转换图。

11.4 交付之后

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。使用如图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。假定我们希望能够将一天的气象状况记录下载到远端的计算机，为了实现这个特性，需要做出下面的改变。

- 创建一个新类SerialPort，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。

做出这种改变不需要破坏原有的架构，而是复用并增强原有的机制，这是一个设计完善的面向对象系统的标志。

---

[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选者。但在本章，我们假定找不到这样的类。

[2]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−l或l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。

[3]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。

[4]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，只需要修改一个类而不是多个类。

[5]这是一个分析决策还是设计决策？两种答案都有其道理，虽然在不得不交付的产品软件面前，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，我们实际上并不介意它叫什么。

[6]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。

[7]例如，若每一帧被指定为1/60秒，则30帧就表示0.5秒。

[8]还有另一种常用的架构模式：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。

[9]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。

[10]另一种方法是，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职责转移给了传感器类，但它使得Sampler类的实现变得简单和更易扩展。

第12章　Web应用——休假跟踪系统

今天，对于很多业务来说，员工的独立性不断增强。员工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，经理和他们的员工们之间的非正式交互少了，他们发现管理员工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并部署一个弹性的休假时间管理应用，用于管理他们的休假时间。

对于一个大企业组织，实现这样或类似的功能，绝不是孤立的决策。提议的这个系统不可能是这个组织创建的第一个系统，所要求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）。结果导致很多架构决策已经是事先做好的。在我们的例子里，我们决定将这个功能作为一个Web应用交付，这不仅是因为Web应用很适合这项任务，还因为它是组织原有内部网络的扩展，这样就为它的用户提供了一个方便和自然的进入点。

在接下来的几节中，我们将概要地讨论这个系统的架构要点和令人感兴趣的方面。由于本书篇幅的限制，我们只表达一些必要的方面和内容，目的是分享对该应用的一般感觉和理解。在真实世界中，精确并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，我们介绍了面向对象的关键核心原则，并将这些原则应用于这种类型的Web应用开发，同时我们也讨论了Web应用开发过程中特有的若干有趣问题。

12.1 初始

系统的需求表示为：一份概要性的愿景文档、关键特性、用例模型、关键用例规格说明，以及对架构有重要意义的需求条目。

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，同时雇员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，是让雇员个人有能力、有责任来管理他们和公司的雇用协议中有关休假的方面。这个需求的潜在动机包括需要精简人力资源（HR）部的功能，最小化与核心业务无关的活动的管理，并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。这些东西太一般、太主观，绝不应该记录为正式需求，对吧？并不一定是这样。很清楚，这行简单陈述虽然不是硬性的、可跟踪的需求，但它确实表达了期望系统的一个真实特性。它非常重要，至少看起来开发人员要满足它，并基本上被最终用户接受，否则交付的应用就会失败。

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，请考虑对于一个Web应用的需求，有时候需要最终用户向系统提交一些格式化的文本（如粗体、斜体、列表、段落等）。架构师有两个可能的解决方案。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）。

这个系统有可能大幅节省人力资源部的时间和金钱，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）。
- 允许访问前一年的请求，允许请求在将来一年半内补偿。
- 利用E-mail通知来请求经理批准并向雇员通知请求状态变更。
- 利用已有的硬件和中间件。
- 作为已有内部网络门户系统的一个扩展来实现，并利用门户的单点登录机制完成所有认证。
- 保持所有事务的活动日志。
- 使HR和系统管理人员能够覆写所有规则限制的动作，并对覆写记录日志。
- 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，以查询任一雇员的休假请求汇总。
- 与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了一些步骤，这些步骤为调用它的执行者提供显著的价值。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。

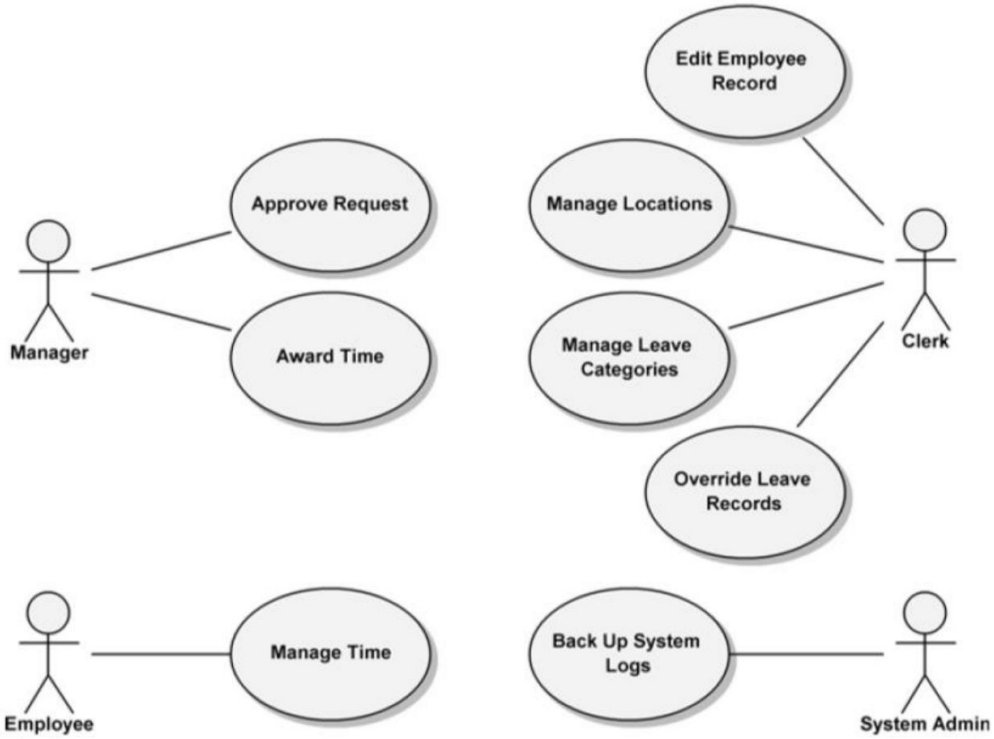


图12-1　顶级用例模型

关于以Web为中心的系统的用例，有一个令人感兴趣的发现，即它们通常非常严格地以刺激和响应的形式表达。也就是说，一个用例场景通常表达成一个执行者动作和系统即时响应的列表。系统响应与实际的、单个命名的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操作动作，而不是离散的用户界面单元，诸如以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的休假时间。

■　Manager：拥有所有一般雇员的能力和目标，但有附加的责任，即批准直接下属的休假请求。根据系统设置的特定限额，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。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）的平滑运行，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职员如何覆写由系统中的规则做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 细化

有时候不太清楚分析何时开始、初始阶段的需求收集和理解何时结束。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，但在能够细化特定用例之前，应该提出对架构有重要意义的用例。

在理想化的世界里，系统的分析应该独立于实现架构。但现实却是，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为中心的架构在初始阶段就知道了。在这个案例中，解决方案将最终作为一个以Web为中心的应用交付，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显式地提出，因为这样的控件在基于Windows的本地客户应用中普遍被使用。这个案例中的用例作者显式地确定了系统需要用户友好的一个区域。在用例书写过程中知道的架构知识，这还有更微妙和普遍的迹象，即常常会提到“导航到Web页面”和“提交”信息，这些概念和以Web为中心的架构紧紧地联系在一起。

可惜，没有一致公认的方法来量化地描述任意一种架构。这里，我们将使用4+1架构视图模型[3]，如第6章所述。

以下对Web应用架构的简短描述，并非对以Web为中心的架构的完整讨论。但是我们确实介绍了一些足够重要的方面，来帮助解释之后做出的设计决策。

12.2.1 部署视图

Web应用是客户/服务器应用的一种特例，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，监听特定端口（通常是端口80）的HTTP请求。在用户的指示下，客户浏览器应用提出请求，请求服务器上HTML格式的资源。服务器很可能会同

时运行很多服务，包括其他Web应用，也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。

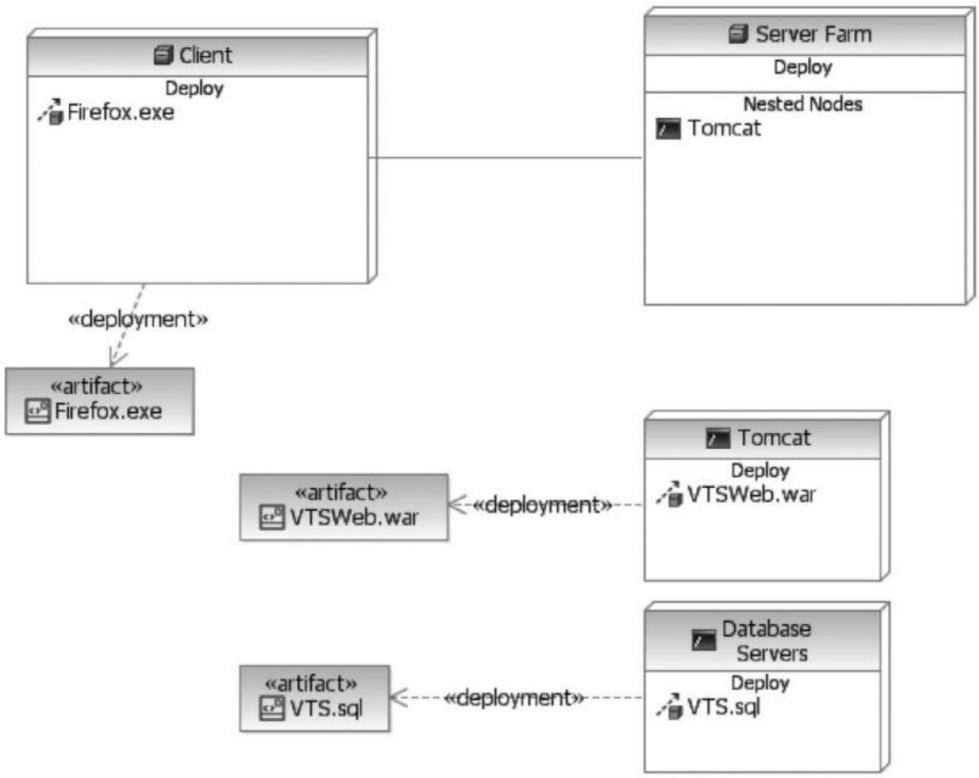


图12-2　以Web为中心系统中组件的部署视图

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。这里显示Tomcat节点上部署了工件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，这里显示它部署了一个工件，名为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方式。需要关注一个重要的逻辑方面，即客户通过这个通信链接与嵌套Web和数据库服务器通信。

这种以Web为中心的系统的表示方式是简单化的。在更大的系统中，展示所有节点、组件、服务器和通信链接的拓扑结构的部署图是复杂的，需要好的可视化和文档化系统，如UML。如图12-3所示为一张抽象的部署图，展示了一个智能路由器如何用于一个Web服务器集群，以及整个应用集合如何在多个应用服务器上运行。这样的策略细节可以十分复杂，但实现应用可伸缩性的通用策略是添加额外的处理节点，这对于Web应用当然是可行的。

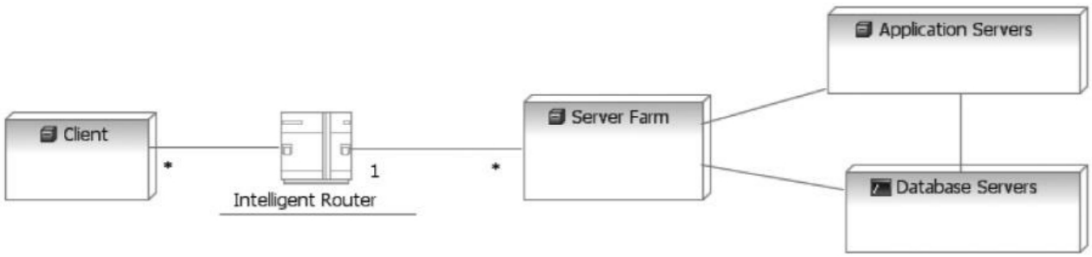


图12-3　有并行和冗余处理节点的Web应用服务器Farm部署图

12.2.2　逻辑视图

典型的Web应用至少有四个逻辑组件：运行在客户上的浏览器、Web或应用容器、针对应用逻辑本身的分离的组件，以及数据库服务器组件。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）规范的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容器提供中介。这使得Web容器成为考虑安全性需求时的一个重要组件。

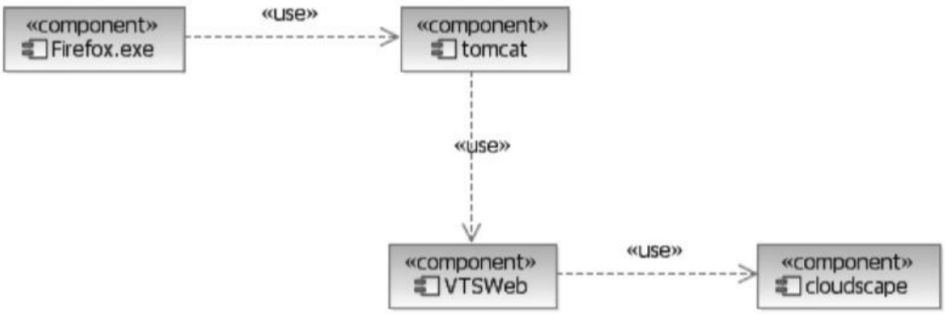


图12-4　一个Web应用的首要组件的逻辑视图

Web服务器被设计成监听某个端口（通常是80端口），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来处理信息。

图12-5展示了代表应用的客户机和服务器层中的元素的逻辑类。在客户机上，一个浏览器负责用简单的HTTP GET或POST命令请求Web页面（或者更笼统地说，资源）。这些服务可以由操作系统提供

或由浏览器自己实现（浏览器调用更低级别的网络API）。Web容器持续监听某个端口（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。请求可以伴随一组键-值对（参数），所有的东西都以字符串表达。[1]一些HTTP请求还打包了更复杂的表单数据，这些表单数据是在页面请求提交前由用户提供的。

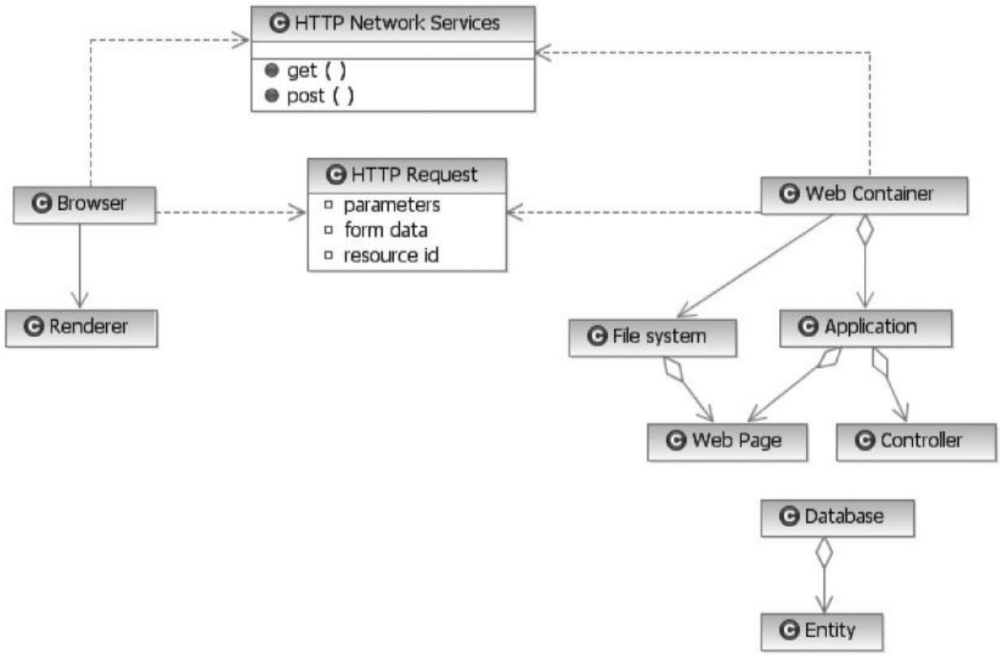


图12-5　Web应用所涉及对象的高级别逻辑视图

当Web容器接收到资源请求时，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容器执行它。在处理期间，HTTP请求检查信息（参数和表单数据），应用执行必要的业务逻辑。通常这种逻辑在一个分离的应用服务器（如一个EJB容器）上执行，应用服务器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加载和执行。控制器对象经常嵌入组件中，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，通常会更多。除了在处理HTTP请求期间，客户和服务器的操作是异步的。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，或分布到多个节点上。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，即客户节点必须运行某种形式的Web浏览器客户软件，发起与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理解它们工作在无连接模式下，即客户和服务器的连接时间绝不会长过一个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）发出简单的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资源的请求得到满足，应用将停止为特定客户工作。

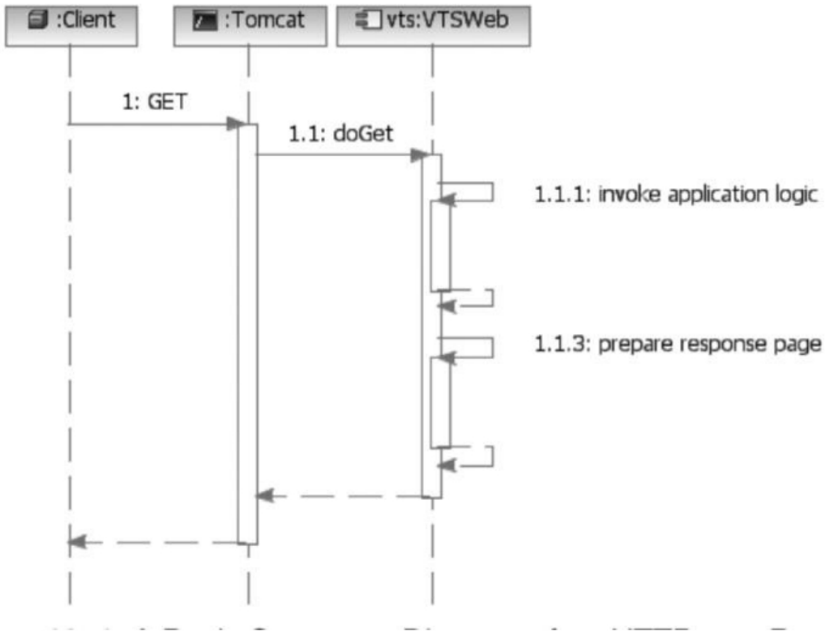


图12-6　一个HTTP GET请求的基本序列图

客户状态管理

Web应用面对的最令人感兴趣的问题之一，就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。请想象一下，每一次访问时，不得不重新输入购物车内容，或者当检查基于Web的E-mail时，为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，即众所周知的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资源时发送。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作为服务器上映射表的一个键，指向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。这个方法有一个缺点，即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将整个状态序列化，成为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，状态只是在服务器上管理。如果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）。这也意味着不能够容易地利用会话状态中的高层次对象来记录客户状态。

这意味着，服务器应用如何能够追踪特定客户的请求历史，并追踪该客户的相关内部状态，一点也不明显。例如，没有利用某个HTTP特性或者实现某种架构机制，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多步骤向导的状态，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。

这个架构还有另一层隐含的意思：服务器不知道，在一些业务流程的中间用户是否放弃了应用。有时一个远程用户的网络连接断开了，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，服务器可以收到通知，表明客户过早断开了连接。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，没有通知发送给服务器。

因此，Web应用设计必须非常小心，注意在Web页面请求之间打开和访问了什么资源。例如，Web应用设计有一条根本规则：绝不在一个页面打开一个事务，而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，并且随时会突然停止。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包中的一组Java类。这些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。部署视图更详细地描述了组件在各层中的部署，逻辑视图更详细地描述了组件本质和责任。

12.2.5 用例视图

通过实现基本的刺激/响应用例，部署、逻辑、进程和实现视图绑在了一起。客户向服务器发起一个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，应用负责生成一个响应，通常以一个HTML Web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一些选项，供用户调用或请求。通过组装所有这些Web页面，让每一个页面专门显示并接受应用的一部分信息，就可以实现一个完整的业务过程。

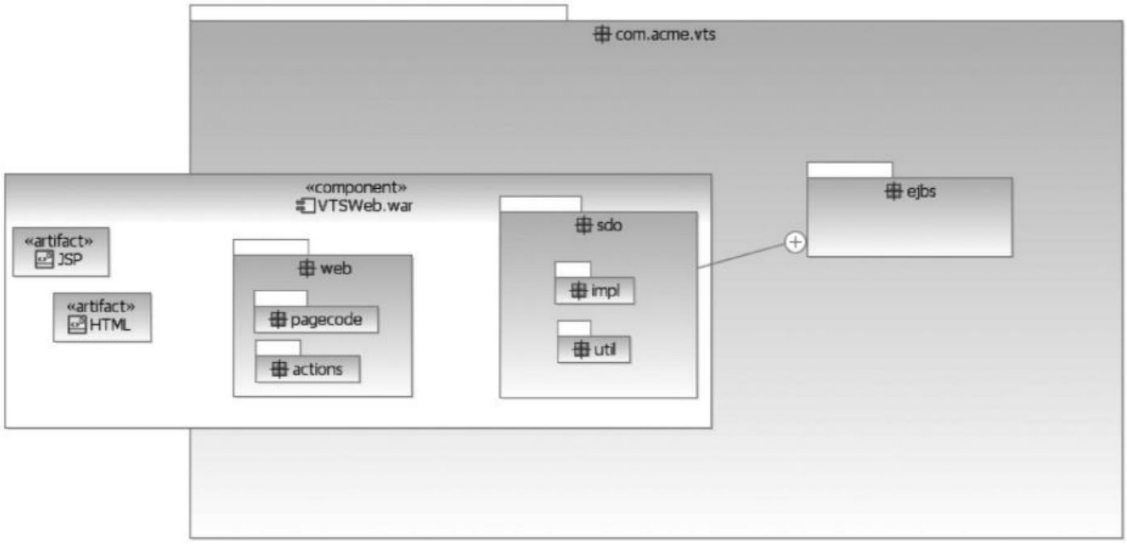


图12-7　实现模型概览

在本章中，我们聚焦于一个在架构上有代表性的用例：Manage Time，这个用例的调用最频繁，也是系统中所有执行者查看最多的用例。所以，高效地实现这个用例，确保满足所有的总体设计目标（包括易用性），是非常关键的。

用例规格说明可以使用许多不同的模板。本章中使用的格式展示了流程中更健壮的步骤，避免使用Web应用可能创建的刺激/响应风格

的流程。这种用例规格说明风格也适用于存在复杂或健壮的备选流程的情况。最终使用的风格或格式，取决于你和组织机构所要求的正式程度。

以下例子给出Manage Time用例规格说明的概览。

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，有管理自己休假时间的权限。

主流程：

（1）雇员从内部网络门户选择一个到VTS的链接，用例开始。

（2）VTS利用雇员的证书，查询所有雇员的休假时间请求的当前状态以及未用的余额。显示包括从之前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，错误高亮显示，并给出文档。

（7）雇员有一个机会更改信息或取消请求。

（8）如果信息完整并通过验证，雇员将返回到VTS首页。如果雇员的休假时间请求需要经理批准，系统立即向有权批准雇员请求的经理发送一封电子邮件。

（9）休假时间请求被设置为待批准状态。

（10）经理通过单击嵌入电子邮件内的链接或者显式地登录内部网络门户回应电子邮件，并导航到VTS首页。

（11）经理可能需要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，同时也在一个分离的区域中列出下属雇员待批准的请求。经理每次选择一个请求，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，系统立即向提出请求的雇员发送一封电子邮件通知。经理的屏幕返回到VTS首页，经理可以批准其他待批准请求，提出自己的请求，或者简单地离开VTS应用。

备选流程：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：雇员已经提出一个休假时间请求，该请求还在等待授权经理的批准或否决。参见主流程的前置条件。

（1）雇员通过内部网络门户应用导航到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）系统发送一封通知邮件给经理。

（7）系统更新请求状态为已撤回。

备选流程：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）雇员通过内部网络门户应用导航到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，系统向经理发送一封邮件通知，该请求的状态变为已取消，该请求包含的时间额度还给雇员。雇员也可以中止这次取消，对当前请求不造成影响。

（5）雇员被返回VTS首页。概览将更新，以反映雇员对待批准休假时间请求所做出的更改。

备选流程：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：雇员已经提出一个休假时间请求，该请求还在等待授权经理的批准或否决。参见主流程的前置条件。

（1）雇员通过内部网络门户应用导航到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，高亮显示并解释所有问题。

这个用例描述包含较多信息，包括对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的领域信息。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。领域知识可能以离散需求的形式存在，例如：

- 所有雇员一天工作8小时。
- 除了公司政策的总体限制之外，每个雇员的休假时间请求还受限于该雇员的首要工作地点。
- 休假时间请求验证规则由HR部门定义和负责。

这些类型的需求或知识可以嵌在用例规格说明中，也可以作为离散的需求条目记录下来。这种类型的领域知识还可以单独写成文档，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雇员手册的一部分，很可能可以通过各种来源得到（如内部网络、表单、文档、新雇员入职简介等）。一个项目的需求集可以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，最重要的任务是确定系统的实体和过程。一个应用的实体和过程代表了业务领域里的概念，最理想的情况是独立于架构的，但也不一定如此。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。我们将从一个以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]是例子，它在一个足够抽象的层次上记录了Web应用的用户界面元素，以便表达系统Web页面之间具体的导航图，同时忽略了风格（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中开发。图12-8展示了一个UX模型的高层次片段，描述了我们实现首要用例的一些屏幕。这个模型和图中有两个关键构造型，即«Screen»和«Form»。带有«Screen»构造型的类代表了用户界面的一个完整单元，它将显示给最终用户，或者粗略地说，就是一个Web页面。一些屏幕包含了HTML表单元素，用于将用户输入的信息提交给服务器。带有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。

对于理解系统各屏幕的导航流程，图12-8是最有用的，但它基本上没有提到屏幕的内容。在这个抽象层次上，包含屏幕内容的更详细的图也是有用的。图12-9展示了VTSHome屏幕的内容。带有«Screen»构造型的类，其属性代表了一些离散的数据值，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。

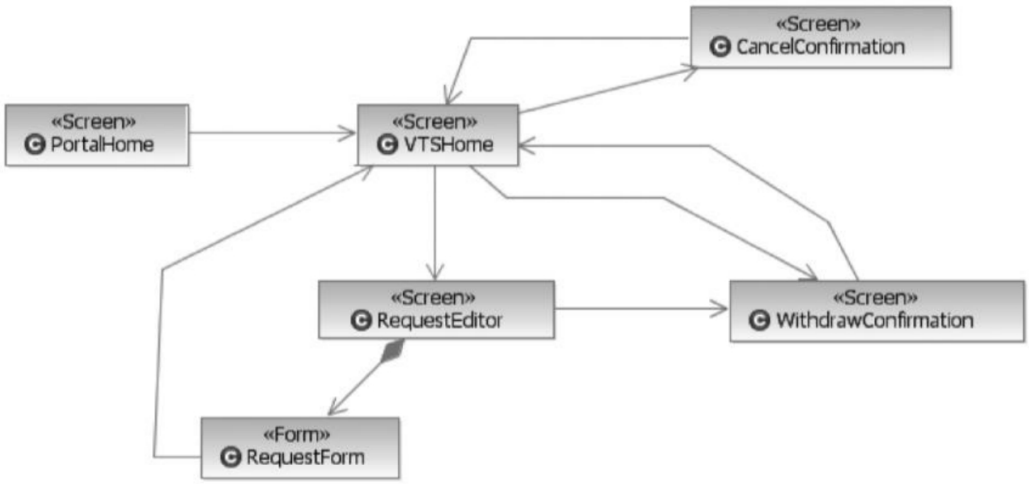


图12-8　一个高层次的用户体验模型

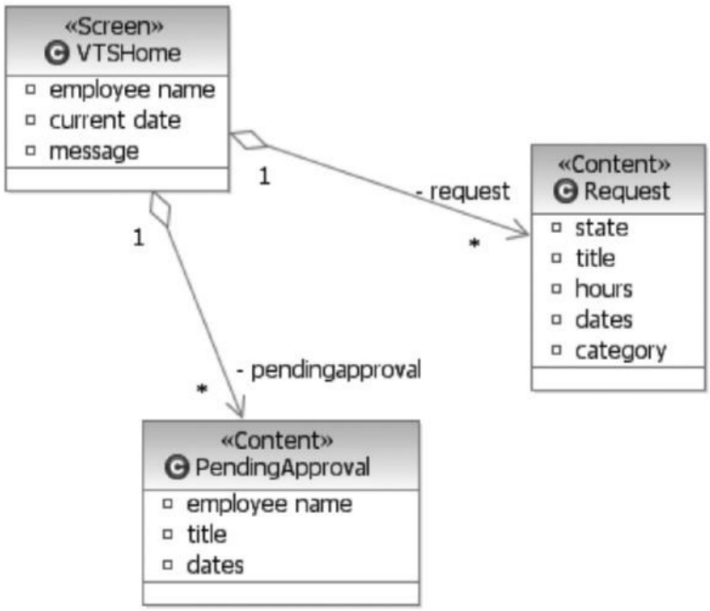


图12-9　VTSHome类的详细视图

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，即«Content»，带有该构造型的类用于标识一组内聚的信息。内容条目经常用做列表中的一行。在图12-9中，类Request和PendingApproval被建模为内容条目，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。在这个高层次的模型中，不需要担心实际的数据类型，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时，所做出的一些决策对最后的设计也会有重要的影响。通过指定VTSHome页面的内容不仅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，我们就暗示了雇员和经理的首页都是这个屏幕。无论是用例

规格说明还是非功能需求，都没有说雇员和经理应该使用同样的主屏幕。这样一个假设并非没有理由，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模型打算记录下来的。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，要记录系统的实体和过程。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中描述的事物和活动，也是领域及最终用户可以识别的。

有一种容易的开始方法，即对用例规格说明和相关的需求文档进行简单的名词-动词分析。重要的名词通常代表模型中的类，动作短语（动词）常常代表那些类的操作。属性和关系代表类的自然、内在的属性，记录在模型中。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。

在如图12-10所示的类图中，有若干重点需要考虑。首先一个想法，就是将休假时间请求（Request）和休假时间许可（Grant）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中记录了另一个重要决策，即经理是一个雇员。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出这样的假设。类图更进了一步，说明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源。例如，在一些公司中，职位高的主管会有专门的个人助理，可以委托个人助理做出休假时间决策。

雇员的其他固有属性是姓名和首要工作地点。注意，在这个抽象层次和要求的用法中，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，很可能需要将头衔、姓、名和中间名分开来，但在这个应用中，这些区分不需要，因为雇员姓名

唯一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。

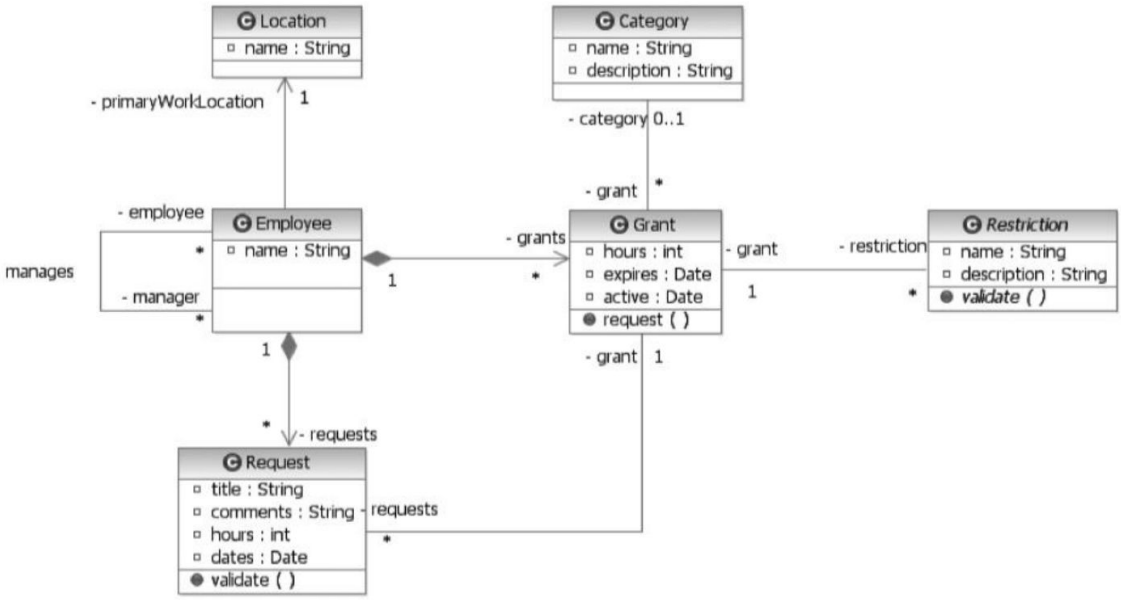


图12-10　支持首要用例的分析模型类图

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时必需的属性可以根据需要添加。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作Auditor，它负责浏览所有休假时间的请求，并检查是否遵守规定。该执行者确实需要雇员ID来交叉引用其他雇员数据。在这种情况下，显式指定雇员的这个属性是重要的。

模型中一些令人感兴趣的元素可以用状态机细化。例如，Request类有一个定义好的状态机，如图12-11所示。状态机本身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，可以看到三个过渡状态和四个最后的状态定义。通过这些状态的路径非常简单，在该图中没有标出来。

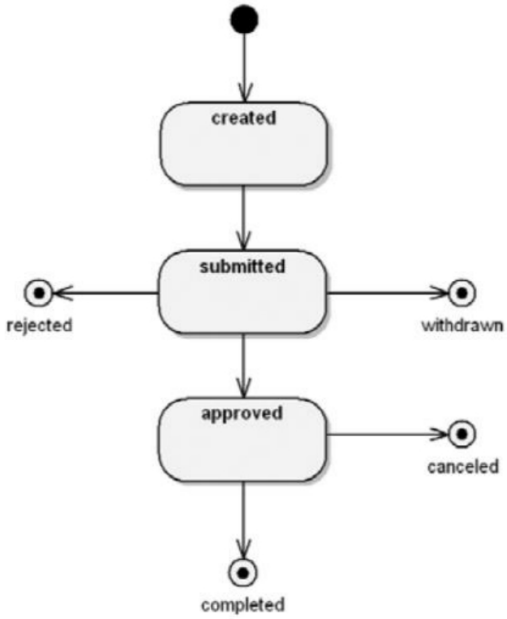


图12-11　Request类的状态机

分析最困难的部分是创建一些相关元素，以保证所有请求的休假时间通过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说明中陈述不多，在当前例子中，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。这个陈述就表明，所用的方法必须有弹性，可由用户管理，而且用户不必具备计算机或算法方面的技能。这一点很重要，因为对于实现通用目标的基于规则的系统来说，有一种明显的、非常有弹性的机制，就是允许记录并解释一门有弹性的脚本语言，如JavaScript，用它来描述规则。大多数算法，不论复杂或简单，都可以用JavaScript实现，并且容易和持久实体及对象进行交互。当然，这里的主要问题在于，HR职员被赋予责任维护系统的这些方面，但通常没有编写JavaScript的技能。

远在选择一个实现解决方案甚至一个策略之前，我们需要先识别属于该模型的高层次抽象元素。现在我们可以创建名为Restriction的单个类，将所有根据与公司和地点相关的政策来验证休假时间请求的责任放在它上面。我们需要更深刻地理解休假时间的规则和规定。不幸的是，在用例规格说明中，没有记录很多规则的内在结构，只有公司文档中才记录了规则的具体实例。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。像这样的分析活动常常需要和领域专家沟通。

仔细检查公司政策中实现的规则，并和人力资源部中的领域专家交谈之后，我们可以整理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。

■ 如果与公司或地点特定的假日直接相邻，类型*X*的休假时间不能接受。

■ 类型*X*的休假时间限制为每星期*Y*小时或每月*Y*小时。

■ 如果在*Y*个雇员中只有*X*个雇员安排在工作，休假时间不能许可。

■ 在某些日期*X*，休假时间不能许可。

■ 这个类型的休假时间限制在一星期中的某一天{周一，周二，周三，周四，周五，周六，周日}。

更详细地理解批准休假时间请求可能的规则之后，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，我们将针对系统中的每个规则类型，创建现有抽象类Restriction的一个特化，如图12-12所示。

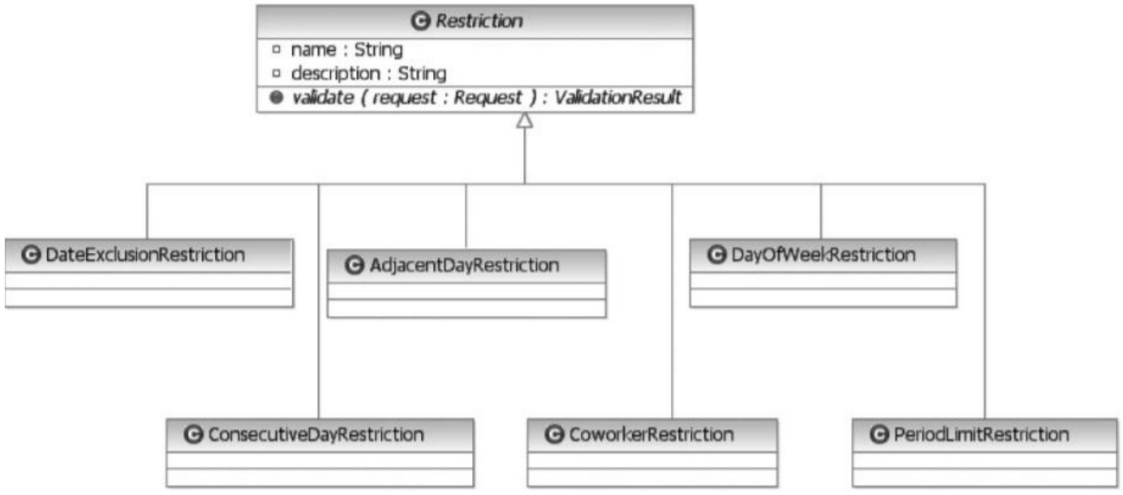


图12-12　Restriction类层次

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足以完成验证。因此，对于不能通过Grant对象导出或导航的对象，每个特化会管理一组属性或关系。例如，CoworkerRestriction类需要管理合作者列表，并确定允许安排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，可以作为第一后援）。

图12-13　CoworkerRestriction类和属性

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。但如果更仔细地查看需求，会看到用户需要收到通知，在试图提交一个无效的休假时间请求时得到一个解释。既然验证规则现在被封装在每个Restriction特化中，那么合适的做法似乎就是让这些类提供一种机制，将这个信息返回给所有调用过程。同样，既然可以发生多次违规，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。

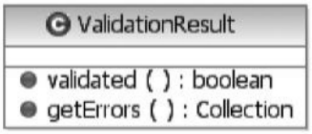


图12-14　ValidationResult类

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。在分析时，让类保持这样简单就够了。实现细节让设计活动解决。这里的要点是，必须有一个过程可以验证休假时间请求，让每个错误结果都是可得的，并提供一种简单的方法来决定有效性。

对这里演进的规则系统还有另一项观察，即限制不能与特定Grant或Employee对象关联，而是广泛地应用于一个Location或Category的许可。这样，我们早先的分析模型必须改变（见图12-15）。建立这些关系让人力资源部更容易应用一大类共同规则，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，但设计变更的动机却是基于让系统更容易使用的强烈欲望，在这种情况下， HR人员的易用性会显著改善。

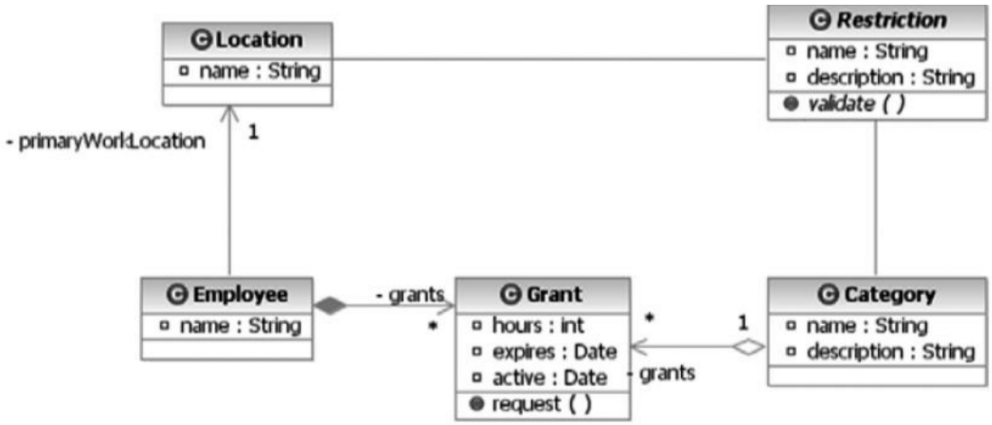


图12-15　Restriction、Location、Grant和Category之间独立和直接的关系

既然限制和规则有了更好的定义，再来就是看一些HR用例的时候了，因为现在我们对于信息的类型和本质知道得更多，这些信息需要由人力资源部管理，目的是管理雇员的休假时间。虽然超出了本章的范围，但可以展望，将有一系列Web页面屏幕，提示每个具体限制类型需要的特有的、具体的值。在这样的机制中，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单个参数的值，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展示了这两个类。

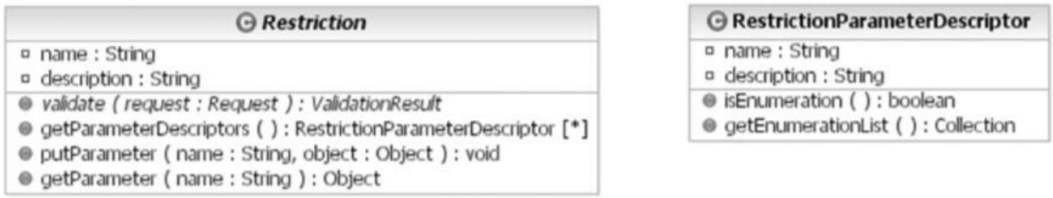


图12-16　使限制更容易通过编程和用户界面管理

在分析期间的不同时候，对加入模型中的想法进行测试是有用的。但是，既然我们还没有得到一个可以为之书写代码的真实具体的架构，就必须对模型进行比较抽象的分析。可以通过在模型元素上进行思想实验来做到这一点，具体地说，就是针对模式中记录的结构和行为，试图以适度的细节来理解特定的场景。我们的模型中处理限制的部分是一个好例子，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。我们希望确保能够根

据需求，并利用模型中已定义的类和对象来完成行为。在这个思想实验中，我们用下面的图来理解工作在North Factory、名字叫Jim的雇员的一个新请求如何被执行。

图12-17展示了针对虚构的雇员Jim的一组对象的对象图，该雇员要求使用休假时间许可，这是对他按时完成工作的奖励。该图表明，通过依赖，可以访问或导航到其他对象实例。实例有真实的名字，和场景相适应，后面跟着一个冒号以及对象的类或类型。这张图标注了三个自由形式的矩形，有助于强调和澄清三个限制来源：Grant、Location和Category。

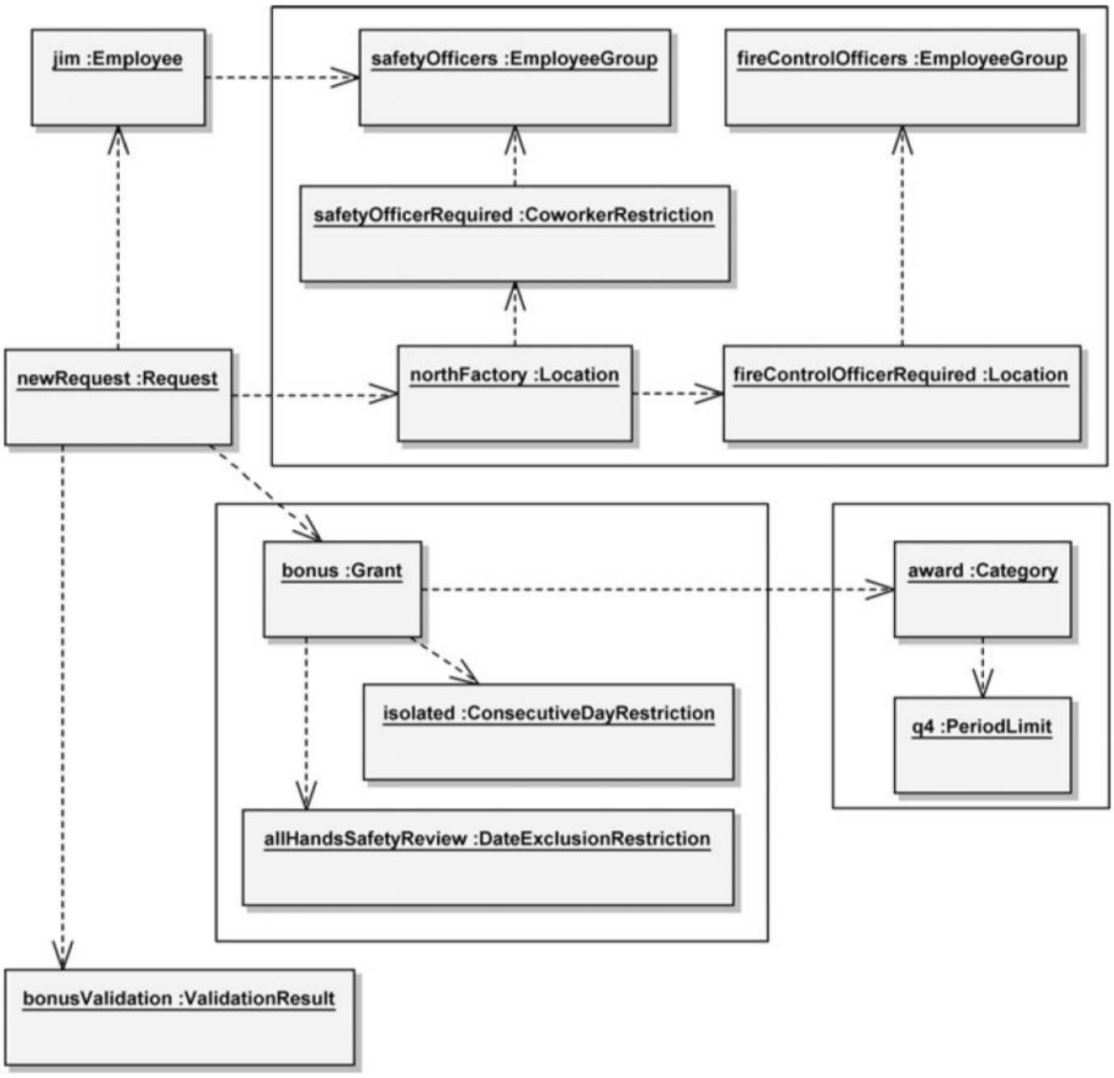


图12-17　描述请求验证协作的对象图

在图12-18中，我们利用一张通信图来考查Request验证的通信路径。连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。我们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，它在这个场景中完成了大多数

工作。通用流是针对Location、Category等，获取并验证各种限制，通过调用每个限制上的validate()操作来完成。如果一个限制没有通过验证，它的消息就被附加到验证结果实例中。

CoworkerRestriction流程是一个要遵守的有趣的流程。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，这些限制也需要访问小组中所有其他雇员的休假计划，以保证在请求的时间，至少小组中的一个雇员安排在位置上。到此为止，我们的模型不足以表达这是如何完成的。因此，作为这个思想实验的结果，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。

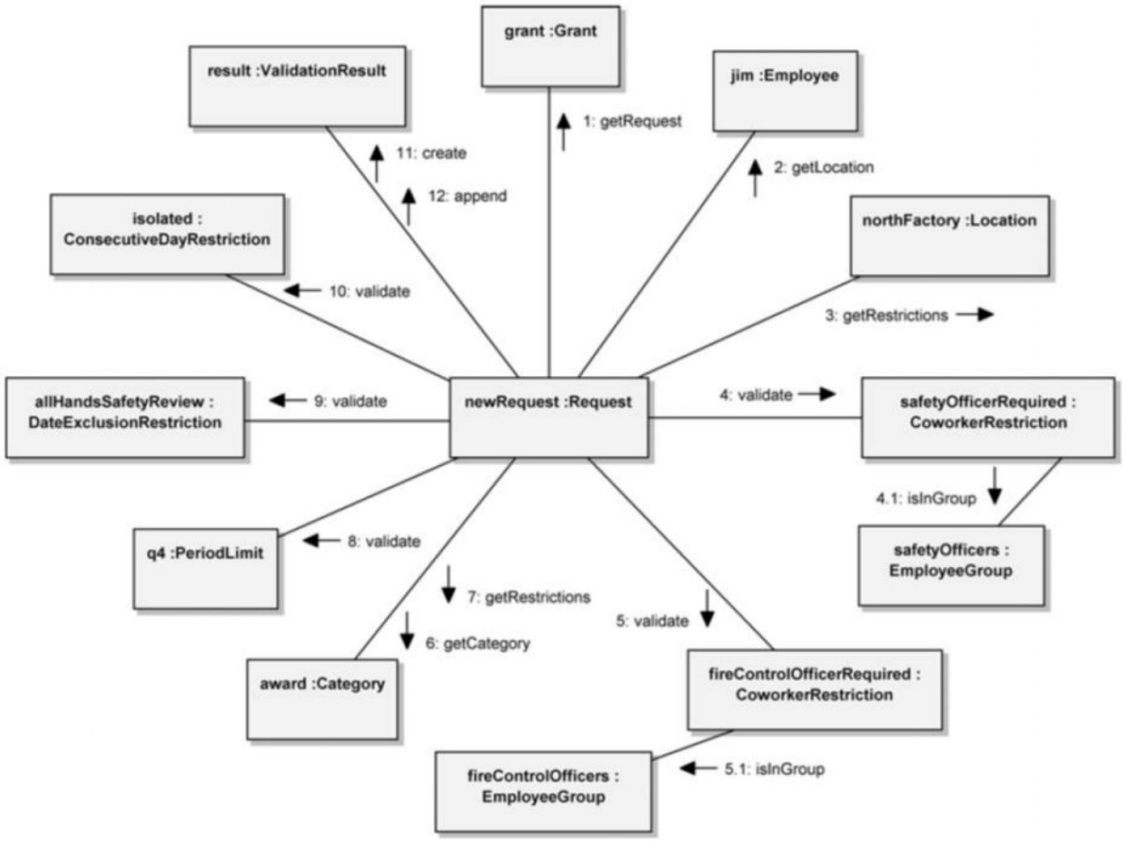


图12-18　描述请求验证协作的通信图

正如第6章中所讨论的，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，这是一个模糊的判断，但这是大多数项目的现实。决定何时开始设计活动的判据通常取决于具体的情况，例如，项目团队及组织的历史或架构的成熟度。

现在我们将分析转换为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，而另一些设计模型相当抽象，实现时仍然需要某些技能和工作。Web应用通常包含了各种情况。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。但大多数展现模型在实现时需要进行大量的工作，才能为最后的系统做好准备。

在本章讨论的VTS设计里，没有篇幅来讨论以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。
- 展现层：JSP（Java Server Pages）、Servlet和JSF（Java Server Faces）。
- 业务层：EJB（企业Java Beans）2.x（具体地说，容器管理持久性（CMP）用于实体，会话bean用于控制器）和SDO（服务数据对象）。
- 安全性：中央认证服务（CAS）[2]。

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，恰巧更好地匹配了组织机构的当前技能和工具集。这个组织机构过去已经成功地应用核心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。只有两个原因会让架构师考虑这种重要的改变。

（1）目前的经验表明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）技术本身不再被支持，或者进化得很厉害，看起来就像新的技术。

在客户端和展现层，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。我们已经确定，实现这个应用所需的全部特性都能够用更老的HTML规范完成。因此，既然不是真地需要新的浏览器并排除旧版本，该应用就可以支持更广泛的客户端配置选择。

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。这里要提到一点有趣之处，即决定不使用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，但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。

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，是否需要集群或其他这样的配置信息，并不是设计时的考虑，除非已经有条件做出这样的决策。选择基于J2EE的架构有一个优势，就是在系统开始部署时，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需要记住一件重要的事情：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编码或设计指南中，开发团队必须遵循。

我们决定使用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。

这个应用的架构中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是公司已有的内部网络门户系统一个扩

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

设计和实现实体时，最重要的工作是识别它们。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（transient）类或值类。例如，ValidationResult类只用做validate方法的返回值。验证在这个系统中不记入日志或保存，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来整合数据要高效得多。这些对象由一个外观（façade）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用于整合进出展现层的数据，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，代表了在处理EJB应用数据时，对某些最佳实践的普遍认同。

对于CMP bean来说，还有另一个重要的设计模式，即它们应该设计成带有最小的业务层行为（如果有的话）。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。所有业务层行为被放在会话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层的模式和惯例在代码中如何实现。这个类相对简单，因为它只定义了两个持久属性，加上一个到Grant类实例的关系。幸运的是，在分析层，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»类的方法被分离成本地方法和远程方法（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），同时也分成实例方法和类方法。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[3]的关系的Category实体bean的类UML图[4]。

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。但在实现中，任何给定的实体事实上由很多分离的、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），与其他bean一起打包进一个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（如图12-20所示）。要认识到的重要事情是，当从分析向设计和实现迁移时，通常模型元素会大量增长。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展示了一段EJB部署描述符，其中包含与Category实体相关的元素。

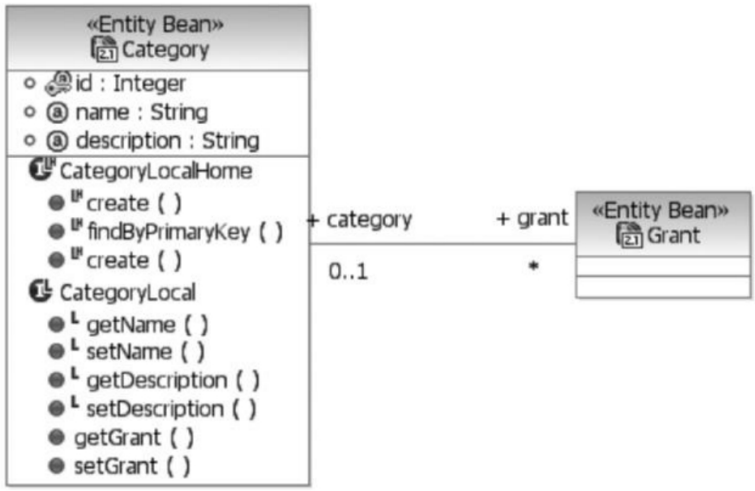


图12-19　Category类的一个设计模型表达

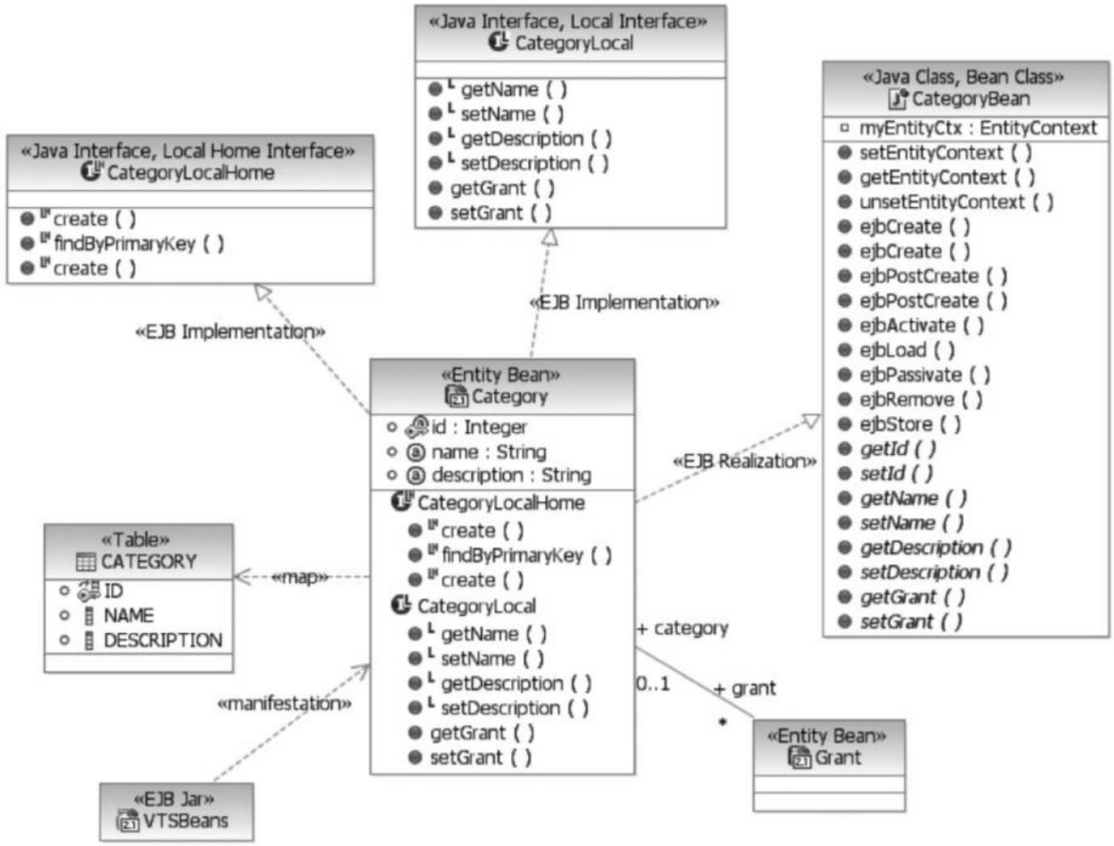


图12-20　Category类的设计模型元素

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>
```

按照设计指南，我们为Category类定义一个外观（façade）[5]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。得到的外观对象叫作CategoryFacade，就像实体bean的展现一样，它定义了远程和本地的、实例和静态工厂的功能。在模型中，加上构造型«façade»的依赖关系表明了设计模型所生成外观对象和设计模型实体表示之间的语义连接。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，它相当于这个类的实例的单一来源管理器。图12-21展示了这样一个外观对象。

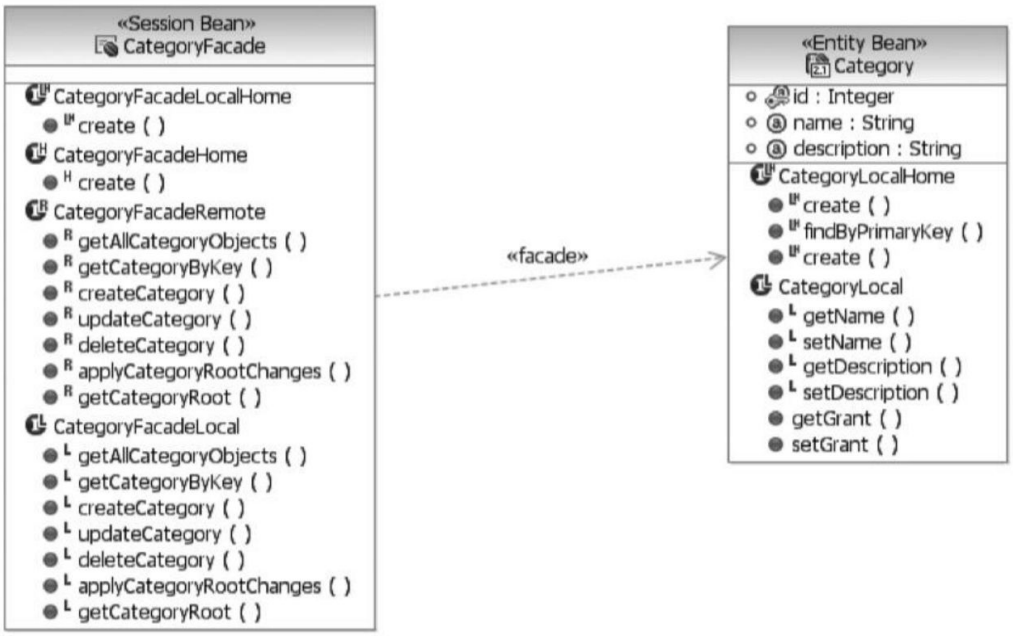


图12-21　外观对象管理对实体bean的访问

1．服务数据对象

这个特定外观实现的另一个特征是使用了新兴的标准，即SDO。简单地说，SDO是一种不连接数据源而访问和操纵数据的机制，它提供了一种有用的方法来整合系统中的数据。SDO使用非连接的数据图（data graph），让客户能够从一个数据源取得数据图，操纵它，然后将改变应用于原始数据源。

使用SDO让我们在整个应用中得到更一致的bean数据接口。在生成外观类时，也生成相应的SDO对象集。每个被创建的SDO对象定义了两个接口：一个根对象（针对数据图）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。）

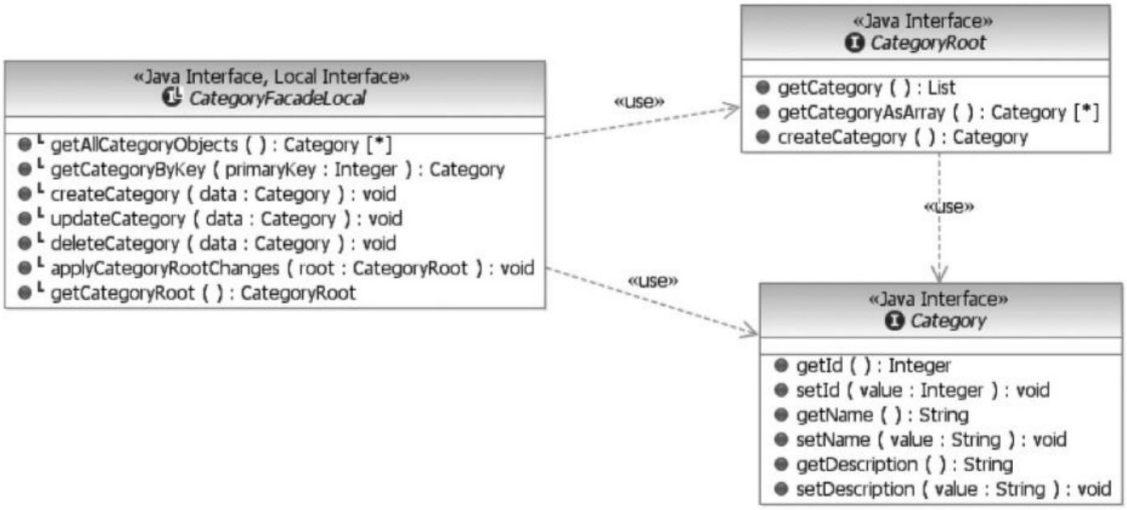


图12-22　CategoryFacadeLocal类使用的SDO对象

外观对象利用SDO对象作为它创建、检索、更新和删除（CRUD）方法的主要参数。它也提供了一个方法，更新对整个SDO对象图所做的任何改变。这个外观对象的典型用法是由另一个会话bean使用，作为业务逻辑控制器，也许代表一个Web页面完成一项请求。让我们看一个样例场景：需要更新或创建休假时间请求类目。这个特定任务的职责落在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）[6]，将它们列在一个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利用对SDO的引用或主键值，容易地删除一个类目。改变一个类目的值时，SDO中的值进行类似的改变，然后告诉外观对象，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-关系数据库系统有一个普遍的问题，即实体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，是否能得到取决于领域，我们不能假定总是会存在。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就是这样的例子，没有单个属性可以安全地作为主键值。

在传统的数据库系统中，键可以通过若干属性和外键值的组合而得。但是，使用整数或字符串这样的原生类型作为单个的主键值，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。这个策略细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，负责管理可用做特定类型实体的键的下一个可用整数值。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来管理对实体的访问，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保留的键的规模。

在我们的应用中加入这个策略，需要创建另一个实体和会话bean，由其他实体外观对象进行本地访问（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）和下一个可以安全地作为主键的整数值。因为这个策略在一个时间里保留了成块的键值，在实际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对主键值次序或分布做任何假设。

如果需要创建实体的一个全新实例，另一个外观对象就会使用键生成器。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。

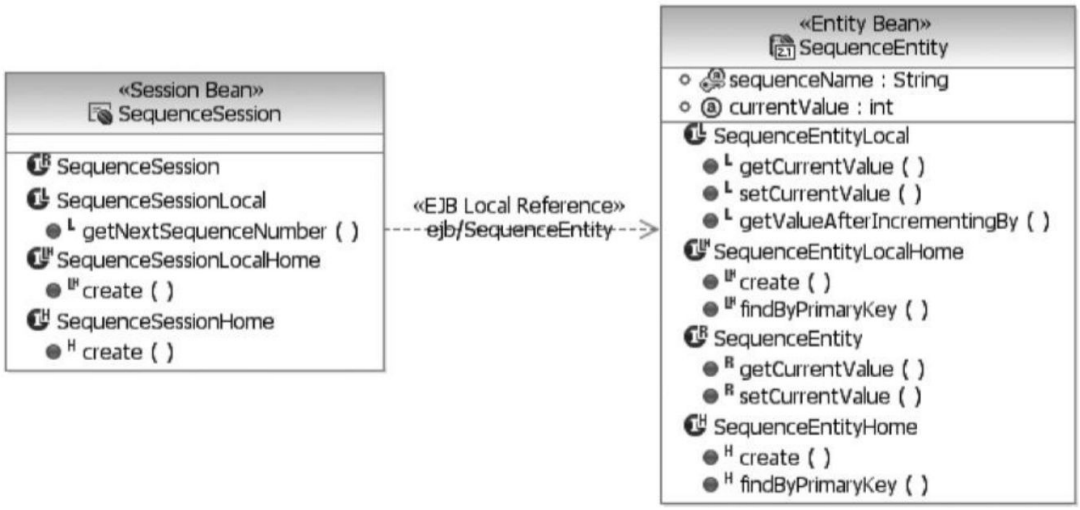


图12-23　用于生成主键的实体和会话Bean

获得新的主键值很简单，访问SequenceSession bean，然后调用getNext- SequenceNumber()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SequenceEntity bean，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）。

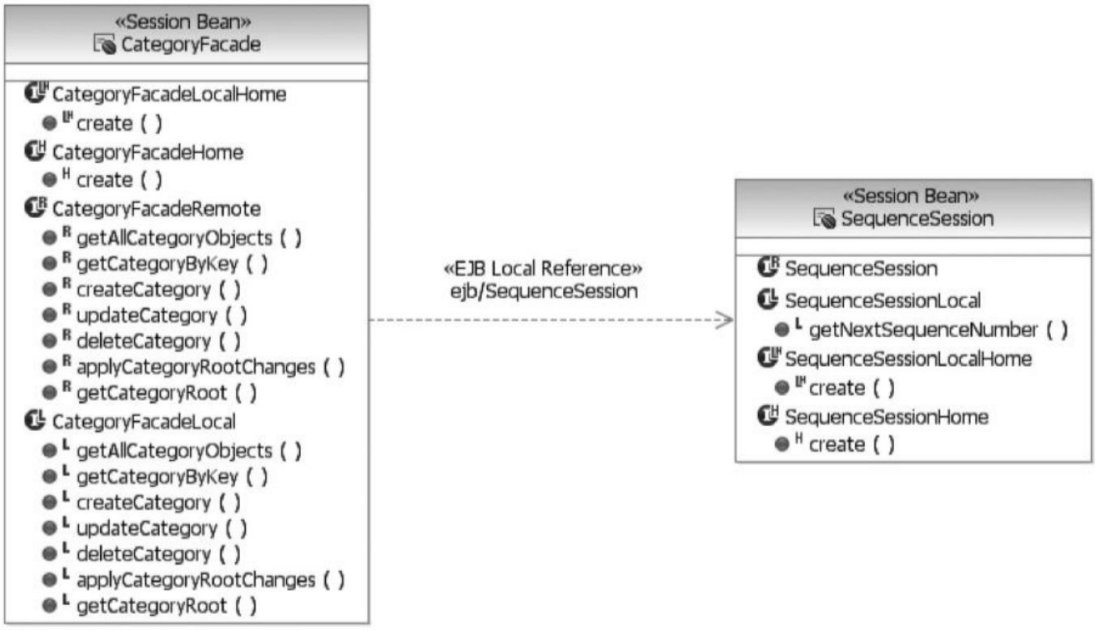


图12-24　CategoryFacade会话bean使用SequenceSession Bean创建新的类目

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，是在数据库服务器上。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，在可能通过网络传送并在另一个进程中重新实例化之后，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，即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
```

finder也可以参数化。在下面的例子中，finder查询将返回所有DateExclusionRestriction实例，这些实例排除了给定的日期。查询中引用的参数表示为一个问号，后面跟着在参数列表中的参数索引。

```
select object(o) from DateExclusionRestriction o where o.date
is ?1
```

最终，这些查询被记录下来，被放进一个主EJB配置文件ejb-jar.xml，并与home接口中声明的Java方法进行关联。

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。很多以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实现的模式。即使在以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设计和实现是该模式的第一个被广泛接受的实现。

在展现层中实现一个严格MVC模式的想法似乎很合理，但仔细研究系统级用例后发现，不清楚哪里需要这样的控制。在绝大多数用例

场景中，所需功能只是比基本的实体CRUD操作多一点。MVC模式适合统筹和协调复杂的业务操作，它不是我们的简单应用所需要的。

这个系统的架构师决定不使用一个明显的MVC模式（如Struts所提供的），而是依赖于Web页面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为中心的应用都非常关键。但是如果中间件、组件或策略对于成功发布应用并非必需，或者在已有的文档或系统将来的计划中没有明确出现，我们就认为不值得承担使用的风险。[7]

做出这个决策之后，所有控制结构要么放在业务层的EJB会话对象中，要么放在服务器层的bean中，要么作为tag嵌入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：（1）页面本身，它们的超链接和表单字段；（2）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到底需要怎样的概念页面，页面里面应该有什么，如何导航以完成系统的业务目标。Web页面通常实现为JSP页面，其中混合了展现数据和与业务处理bean的关系。但是，第二个关注点完全是美学和用户理解的问题。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形式，以便让最终用户最有效地理解页面展示的内容，并完成工作。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的是关于艺术和视觉创造性，而不是传统的分析和设计。

网页设计以UX模型开始。UX模型通常是一个非常好的起点，描述了最初侯选的Web页面名称、内容和链接。一般说来，UX的«screen»类映射到单个JSP页面。如果页面里没有动态内容，纯HTML Web页面也可以是Web应用的一部分。然而，即使在这些情况下，最好也是使用JSP页面，因为一些客户端状态管理解决方案需要动态地重写所有URL超链接，这只能在动态页面里实现（参见本章前面部分的补充材料“客户状态管理”）。

基于所使用的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。

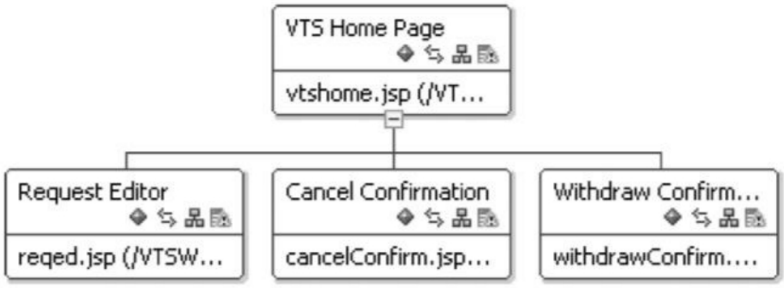


图12-25　设计工具表达站点布局

在设计展现层时，有两个主要的考虑：如何填充页面的动态内容，以及在页面转换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来帮助集成页面，并实现这两个任务。JSF有助于简化动态Web页面的构造。它定义了很多定制的tag，这些tag与HTML一起嵌入JSP页面中，用于调用业务对象上的方法并提取数据以放进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）也提供了许多有用的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，需要一个展现层中的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，以便有可能将不同的层部署到不同的节点上。为了填充VTS首页，除了其他信息之外，还需要当前雇员的请求概要。这个概要来自一个Employee CMP bean和与它相关的Request CMP bean。既然对CMP bean的访问由生成的外观对象管理，展现层中的会话bean需要一个到外观对象的远程引用。外观将返回SDO对象给展现层会话bean，展现层会话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类是一个在展现层执行的会话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也收集并提供即将用于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.VtsHome-
Action"/>
...
<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>
```

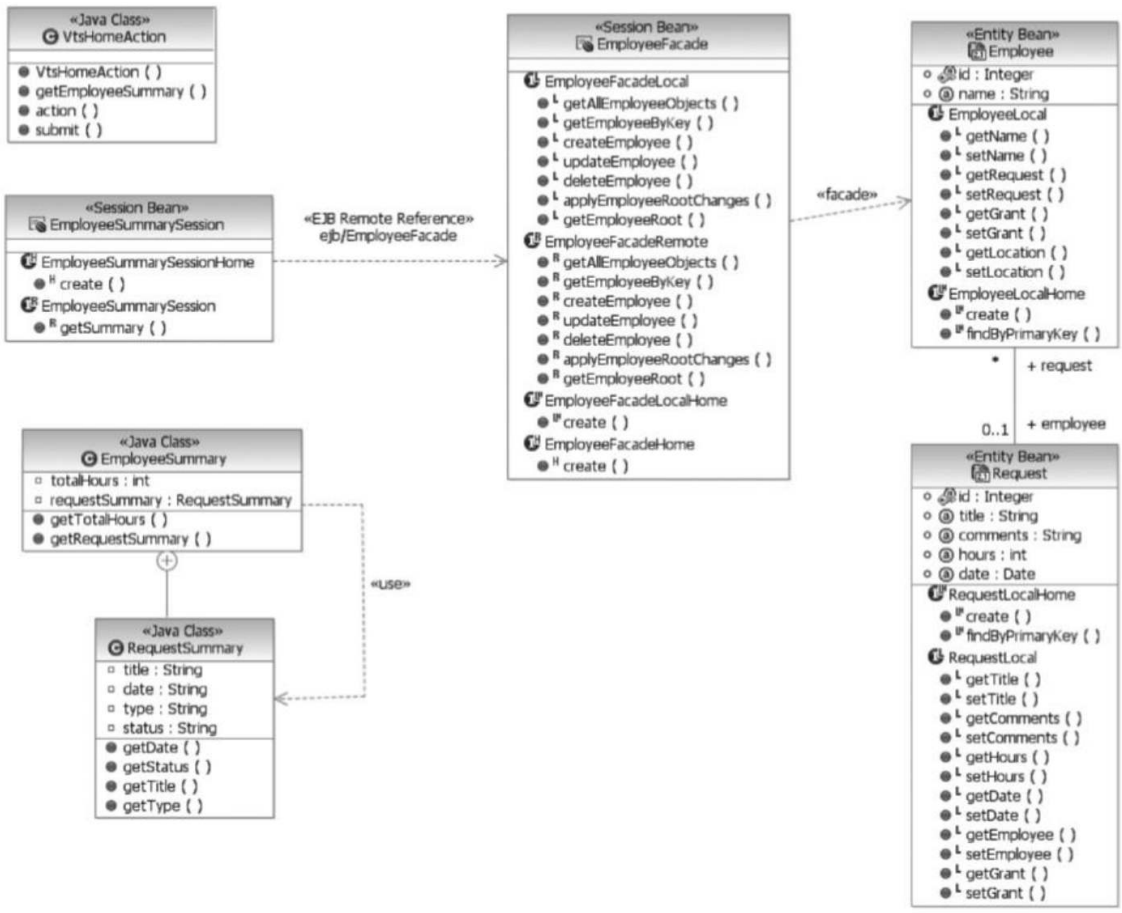


图12-26　展现层和业务层中的设计元素

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，包含了导入并关联的前缀标识符。

JSP tag引用展现层bean，展现层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的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，这是一个好的实践。该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转换期间完成。逻辑由用户查看或导航到下一页面的请求触发。下一页面可以是用户实际请求的页面，也可以不是。依赖于业务逻辑的结果，基于当前客户的状态，期望的页面可能切换到不同的页面。例如，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。导航规则定义从一个页面到另一个页面的可能转换，转换基于一个结果值。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并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。

发起页面中包含了超链接或表单按钮，并使用了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

被传送到默认的导航处理器，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}"/>
```

这个设计导致所有展现层逻辑在托管或后台bean中表达，这些bean在Web服务器中执行。通过创建标准的对业务层中EJB的远程EJB引用来完成它们的任务，业务层可能运行在系统的不同节点上。

12.4 交付和交付之后

以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，还有它们自己特殊的实现和测试考虑。本质上，Web应用存在于一个异构环境中，环境很容易改变，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，有意没有使用特定浏览器才有的特殊技术。

实现一个Web应用，绝大部分工作是服务器端软件的开发，在典型的Web应用中，几乎所有编写的软件都在服务器端执行。只有使用了定制的JavaScript，或者开发专门的applet或客户端控件时，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。一些浏览器通过新的专有特性扩展了脚本语言，另一些对规范的解释稍有不同，或者有不受官方规范管制的不寻常的副作用。

- 样式表：样式表和其他布局相关的功能依赖于可获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（Frame）：自从引入以来，框的实现就是一个问题。在应用中使用框时，滚动、改变尺寸和定位问题经常变得很麻烦。如果一个应用使用了框，一定要完全测试它们，不仅针对所有目标客户配置，还要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，一些最终用户就会遭遇极慢的页面转换。如果正在开发的应用将在Internet部署，并针对匿名的用户群，就必须特别小心，针对低带宽和弱客户配置进行页面测试和设计。

当技术随着时间推移而进化时，这些问题也一直是挑战。

---

[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。

[2]更多信息，参见www.ja-sig.org/products/cas/。

[3]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。

[4]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）不是面向对象传统，而是使用Java和J2EE框架的一个限定。

[5]参见Core J2EE Patterns[4]中描述的会话外观模式。

[6]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。想象一下，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。

[7]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做同等考虑。

附录A　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的[1]。

- 它是设计工具。
- 它是供人类使用的工具。
- 它是提供计算机指令的工具。

某些读者可能不熟悉本书提及的某一门面向对象编程语言，本附录就是为这些读者准备的。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

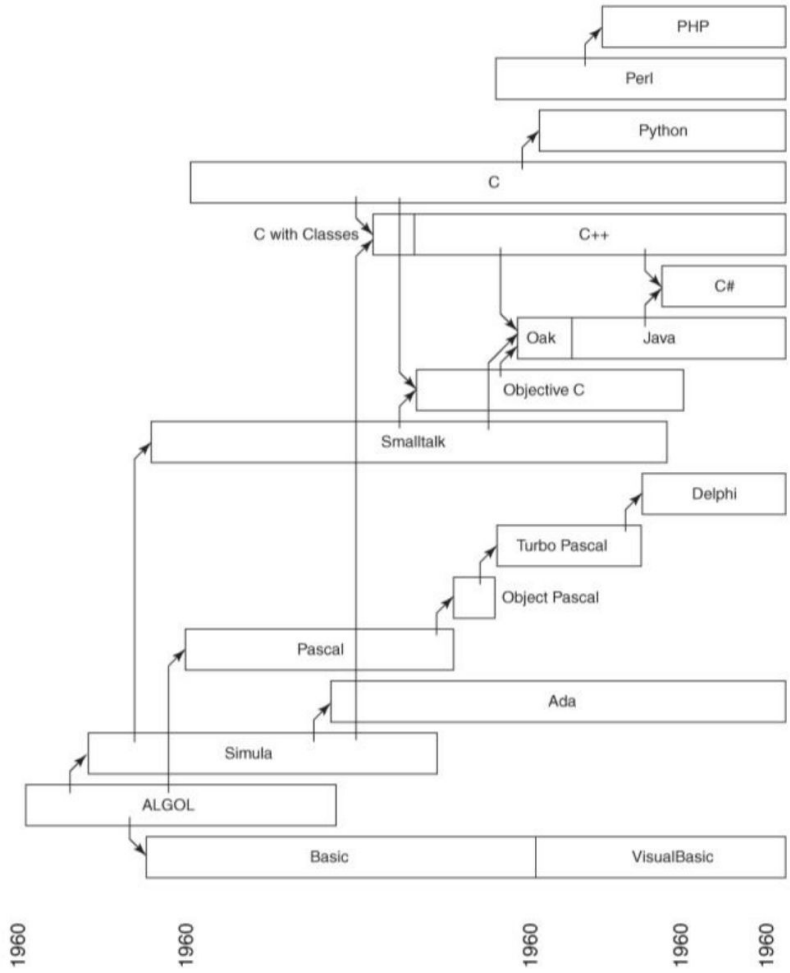
当前，有超过2500种不同的高级编程语言[2]。我们之所以能看到如此多不同的语言，是因为每一门语言都由它认知的问题域的特定需求形成。此外，每一门新的语言都让开发者能够转向越来越复杂的问题。通过每个以前未探索过的应用，语言设计者学习到新的经验，这些经验改变了他们对语言中什么重要、什么不重要的基本假设。计算理论的进步对语言进化的影响也很大，导致人们规范地理解了语句、模块、抽象数据类型和过程的语义。

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。我们在第2章中还讨论过，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。面向对象语言是基于对象的，同时支持继承和多态。

所有当代基于对象和面向对象编程语言的共同祖先都是Simula，Simula由Dahl、Myrhurg和Nygard[3]在20世纪60年代开发。Simula基于ALGOL的思想，但添加了封装和继承的概念。也许更重要的是，

Simula作为一门描述系统和开发仿真的语言，引入了编写程序的学科，将词汇表映射到程序的问题域。

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的、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。箭头表明它们的发展中的主要影响。



图A-1　有影响力的计算机语言谱系

确定某种编程语言（甚至是语言家族）使用的流行度或是一项让人畏惧的任务，可能会挑起语言粉丝的情绪，还会引发许多高级和业余统计学家的审查。虽然单单流行度本身并不意味着编程语言的质量，但确实暗示了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]是确定某种编程语言流行度的新颖方法，这种方法通过检查因特网Web页面和新闻组的搜索结果来得出结论。通过对每种目标语言进行简单的查询，得到代表兴趣的相对页面百分比，把它们进

行列表显示，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。

在以下每个小节，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，我们定义类来表示形状的类型，这些形状可以在显示器上绘制。图A-2展示了所涉及类的通用UML模型。

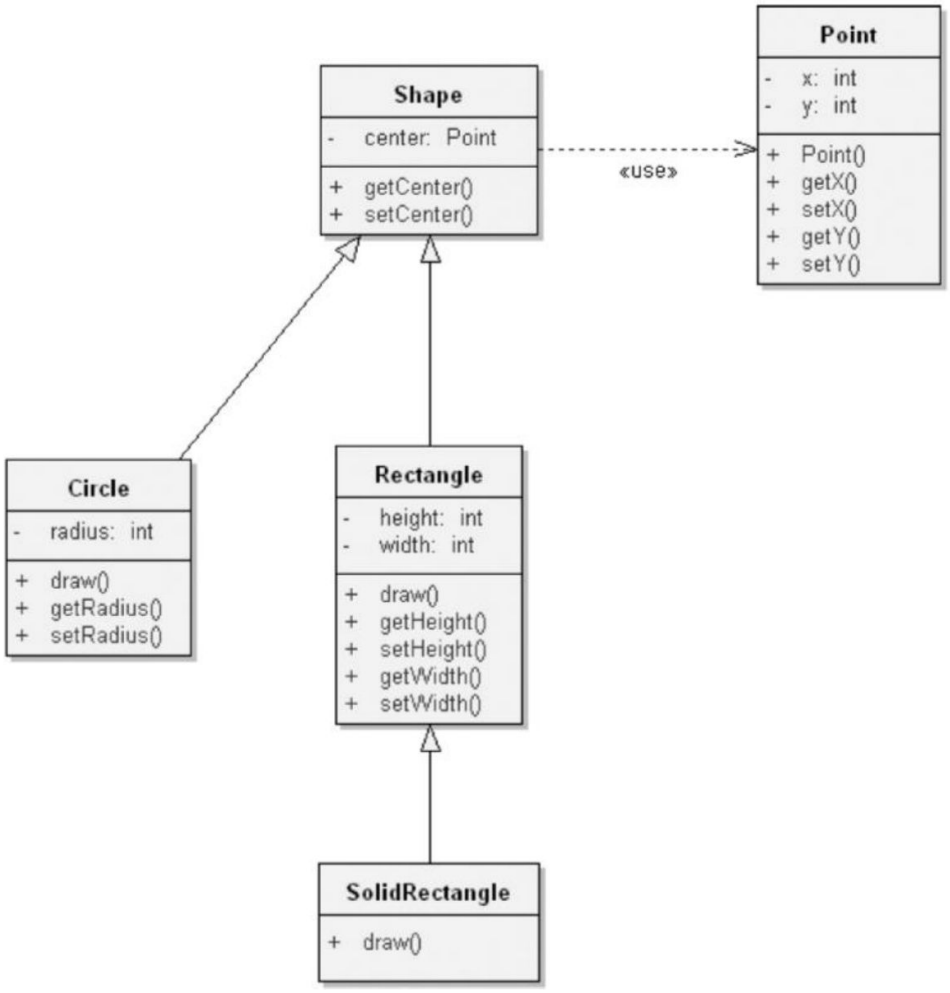
表A-1　基于Internet确定当前流行的编程语言

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%

续表

名　次	编 程 语 言	评　分
15	Lisp	0.86%
16	Fortran	0.82%
17	MATLAB	0.78%
18	Ada	0.68%
19	Pascal	0.49%
20	AWK	0.48%

以下每一个例子都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。我们不试图针对给定的平台优化所述任务。



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，从而将环境中的所有东西都作为对象来处理。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。在这些版本中，虽然用户界面通常截然不同，但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。

我们从定义类AShape开始，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 Programming Environment*[10]，以及Krasner的*Smalltalk-80: Bits of History，Words of Advice*[11]。LaLonde与Pugh [12]对Smalltalk-80做了深入探索，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C的超集。但在某种意义上，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，叫作cfront。因为这个翻译器释放出中间表达形式的C代码，使得C++事实上可以很快地移植到每一种UNIX架构上。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为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（和我们的类目概念一致）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-3C++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（如X Windows或Microsoft Windows）的编程接口，以及可以链接的全局对象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++ Reference Manual*[15]，通常被称为“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 Microsystems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，叫作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，这实际上为这门技术创建了一个潜在用户众多的巨大市场。

虽然早期的市场工作聚焦于Java的Internet适应性和平台无关性，但真正让Java变成受人喜爱的面向对象语言，是因为它在服务器端的使用。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，以交付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，更显著的是，它的相关技术，如企业Java Beans（EJBs）、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技术规范、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规范请求（JSR）的评估和反应，JSR针对的是新的规范或已有规范的重要修改。当前，大约有100个规范的工作在进行中。

Sun Microsystems持续管理着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。因为它的可移植性和丰富的标准功能集，Java技术和Internet匹配得非常好。

Java语言本身有意保持和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++来进行面向对象编程。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省去了C++中的很多特性，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。最新的规范——Java 2的版本1.5，确实包括一些新东西，称为泛型（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-4Java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
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

针对Java重新创建我们的例子，就需要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;
	}
}
```

接着创建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。

附录B　进一步阅读

本附录通过参考分类书目中列出的条目，指引读者对每一章做进一步阅读。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*，在书目的H部分（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]、负责采购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的奠基性参考文献，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]描述了新兴的复杂性科学，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。该主题的扩展工作包括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]最先引入。Kay的博士论文[F 1969]为之后的许多面向对象编程工作奠定了方向。

Shaw[J 1984]针对高级编程语言中的抽象机制进行了很好的总结。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]编辑的著作讨论了层次结构的意义和重要性。

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]。一些年度的面向对象技术会议的会议纪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，并在Oberon语言中使用。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]研究了教授高效地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。

基于类的层次结构有的一种替代方案，即利用“模范（exemplar）”实现委托。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，笛卡儿在*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]做了概括。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，即通过检查问题域词汇表来发现关键抽象和机制的方法。Iscoe [B 1988]对这个领域做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

对象管理组织已经推出了UML超结构（Superstructure）规范2.0版，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）。另一项相关的工作是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，文献目录针对这个方面提供了大量的参考文献。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]。另一份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]向开发经理提出了一些建议。

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] 利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] 将Boehm [H 1989，H 2002]的软件风险信息映射到Meyer [H 1995]的面向对象概念，对面向对象开发的风险提供了一个创新的视角。

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部分和第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包括三个部分：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、修改并扩充了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，还描述了两个面向对象黑板系统：来自Stanford的BB1，以及为英国国防部开发的BLOB。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，强调用例和建模方法。

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篇：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. *Communicationsof 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. *Communicationsof 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, p. 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篇：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., p. 383-387, 505.

[63] Object Management Group. *UML 2.0 Superstructure Specification*, p. 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 ComprehensiveOverview of Component-Based Development for the Enterprise*. New York, NY: Wiley.

第7章：实战

[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*, p. 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篇：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 2.0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的简单头脑风暴工具。

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，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，一个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。具体来说，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，通常叫作泳道。

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）：从可执行代码产生出逻辑或物理模型。

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。

博文视点致力于专业计算机图书的出版，我们欢迎优秀译者翻译经典外版书籍，欢迎优秀作者投稿。

联系人：符隆美

E-mail：fulm@phei.com.cn

Broadview®
www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版

面向对象分析与设计

第3版
修订版

Object-Oriented Analysis and Design with Applications (Third Edition)

本书长期以来一直是面向对象技术十分重要的参考书，面向对象技术经过演进，已经成为了主流软件开发的事实标准。在大家高度期待的第 3 版中，读者可以学习如何利用统一建模语言（UML）2.0 来实现面向对象方法。

本书的作者包括 UML 创始人之一的 Grady Booch，他们总结了各种丰富的经验，提出了对象开发的改进方法，以解决系统和软件开发者所面临的复杂问题。通过大量的例子，他们说明了基本概念、解释了方法，同时展示了各行各业的成功应用，包括系统架构、数据获取、密码学、控制系统和 Web 应用。读者还可以得到一些重要问题的实用建议，包括分类、实现策略，以及具有成本效益的项目管理。

本书涵盖内容

- 关于 UML 2.0 的大量介绍（实际上是一本“书中之书”），从表示法的最基本元素到高级元素，并强调了关键的改变
- 极大地强调了对建模的关注（这也是读者迫切要求的），包括五章内容，每章强调了总体开发生命周期中的一个特定阶段
- 推理复杂系统的新方法，包括利用 OOAD 和 UML，对系统架构进行新的处理
- 对于对象模型中广泛误解的元素，仔细检查了它们的概念基础，诸如抽象、封装、模块化和层次结构等
- 建议如何分配开发团队的资源，以及如何管理开发复杂软件系统相关的风险
- 附录讨论了主要的面向对象编程语言，如 Java 和 C++

关于作者

Grady Booch 是 IBM 院士，也是六本面向对象编程畅销书的作者。他是世界知名的 OO 技术发起人和 UML 创始人之一。

Bobert A. Maksimchuk 是 Unisys Chief Technology Office 的一名研究主管，关注新出现的建模技术。他是 *UML for Database Design* 和 *UML for Mere Mortals* 的合著者。

Michael W. Engle 是洛克希德马丁公司的首席工程师。他有丰富的技术和管理经验，从项目启动到运营支持，涵盖了完整的系统开发生命周期。作为系统架构师，Mike 在复杂系统开发中采用了面向对象技术。

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。

Jim Conallen 是 IBM Rational 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。

Kelli A. 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。

上架建议：软件工程

ISBN 978-7-121-28666-7

9 787121 286667

定价：109.00元

Pearson
www.pearson.com

策划编辑：张春雨　符隆美
责任编辑：刘　舫