面向对象技术实践丛书

Object-Oriented Analysis & Design

面向对象的分析与设计

Andrew Haigh 著　等译

机械工业出版社
China Machine Press

目　　录

译者序

前言

第一部分　什么是面向对象

第1章　面向对象简介 …… 1

1.1 结构化技术和面向对象技术的比较 …… 3

1.2 什么是面向对象 …… 4

1.2.1 面向对象技术是如何与用户关联的 …… 5

1.2.2 面向对象技术的其他优势 …… 5

1.2.3 面向对象技术的一些弱势 …… 7

1.3 什么是对象 …… 7

1.3.1 识别对象 …… 8

1.3.2 属性 …… 12

1.3.3 方法 …… 13

1.3.4 对象状态 …… 14

1.3.5 类 …… 14

1.4 面向对象基础 …… 15

1.5 继承 …… 17

1.6 重定义 …… 21

1.7 文档 …… 21

1.7.1 类的描述 …… 21

1.7.2 图的使用 …… 22

1.7.3 继承 …… 23

1.7.4 编码规范 …… 23

1.8 小结 …… 24

第二部分　分　　析

第2章　分析 …… 25

2.1 预分析 …… 26

2.2 当一个对象不成为对象时 …… 27

2.2.1 公共汽车站问题域的实例 …… 28

2.2.2 桌子问题域的实例 …… 29

2.2.3 问题域小结 …… 31

2.3 使用用例分析 …… 31

2.3.1 用例图 …… 31

2.3.2 一个简单用例的例子 …… 32

2.3.3 一个用例模板 …… 32

2.3.4 一个用例实例 …… 33

2.3.5 写好用例的七个要点 …… 34

2.4 记录分析 …… 36

2.4.1 分析文档：类的静态特性 …… 36

2.4.2 分析文档：类的动态特性 …… 38

2.4.3 分析文档：系统的静态特性 …… 39

2.4.4 分析文档：系统的动态特性 …… 45

2.5 小结 …… 50

第三部分　设　　计

第3章　设计方案 …… 51

3.1 抽象类 …… 52

3.2 应用程序编程接口 …… 53

3.2.1 API结构出现以前 …… 53

3.2.2 为什么使用API结构 …… 54

3.2.3 从API类中派生 …… 54

3.2.4 使用API类 …… 54

3.2.5 Java原始接口 …… 55

3.3 模板 …… 57

3.3.1 何时使用模板而不使用继承 …… 57

3.3.2 在C++中实现的模板样本 …… 58

3.4 好的设计——原则和度量标准 …… 60

3.4.1 认识设计中“毒瘤”产生的原因 …… 60

3.4.2 面向对象的类的设计原则 …… 61

3.4.3 设计的度量标准 …… 64

3.5 全局对象 …… 71

3.6 确定实现方法 …… 73

3.7 虚方法 …… 74

3.8 复制构造函数 …… 76

3.8.1 表层复制构造函数 …… 76

3.8.2 深层复制构造函数 …… 76

3.9 关联的实现 …… 77
3.9.1 双向关联 …… 77
3.9.2 单向关联 …… 77
3.10 小结 …… 78
第4章 需要避免的设计方案 …… 79
4.1 过程对象 …… 79
4.1.1 过程的变化 …… 81
4.1.2 差异处理 …… 82
4.1.3 增加新的订单类型 …… 82
4.1.4 将控制过程放在对象内部 …… 82
4.2 责任的委托 …… 82
4.2.1 实例1——确定某人的年龄 …… 83
4.2.2 实例2——过滤数据 …… 84
4.3 方法责任 …… 85
4.3.1 实例1——买一台烤面包炉 …… 85
4.3.2 实例2——显示运动队的信息 …… 86
4.3.3 实例3——更新联盟中各运动队列表 …… 87
4.3.4 实例4——对联盟中各运动队排序 …… 88
4.3.5 方法的回顾 …… 89
4.4 C++中的友元结构 …… 90
4.4.1 访问级别 …… 90
4.4.2 友元是如何影响访问级别的 …… 90
4.4.3 使用友元结构 …… 91
4.4.4 对友元结构的评价 …… 92
4.5 多重继承 …… 92
4.5.1 从WorkingStudent派生一个类 …… 94
4.5.2 重新定义被继承的name方法 …… 95
4.5.3 多重继承菱形 …… 96
4.5.4 多重继承的替代方法 …… 98
4.6 继承的不当使用 …… 99
4.7 小结 …… 102
第5章 高级设计技术 …… 103
5.1 高级API结构 …… 103
5.1.1 什么是高级API结构 …… 104
5.1.2 如何克服缺点 …… 104
5.2 线程 …… 105
5.2.1 资源同步 …… 106
5.2.2 Java同步的问题 …… 106
5.2.3 资源的死锁 …… 110
5.3 Model/View/Controller机制 …… 113
5.3.1 中心MVC Controller方案 …… 114
5.3.2 线程方案 …… 116
5.3.3 被动反应式方案 …… 118
5.3.4 Java方案 …… 118
5.4 暴露接口方案 …… 125
5.5 引用计数 …… 127
5.5.1 通过继承实现引用计数 …… 127
5.5.2 通过关联实现引用计数 …… 128
5.5.3 多线程应用程序 …… 129
5.6 小结 …… 130

第四部分 编 程

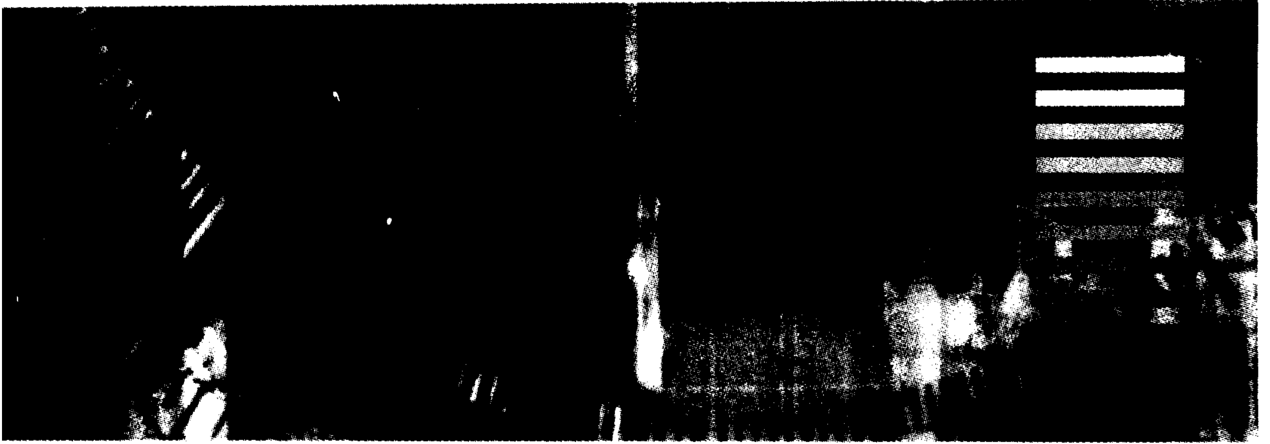
第6章 测试 …… 132
6.1 测试装备 …… 132
6.2 关于构造方法和析构方法的测试 …… 133
6.3 方法测试 …… 134
6.3.1 if-then-else …… 135
6.3.2 for循环 …… 135
6.3.3 while循环 …… 136
6.3.4 switch语句 …… 137
6.3.5 try-catch …… 137
6.3.6 函数调用 …… 138
6.3.7 测试单个方法的例子 …… 138
6.4 类测试 …… 139
6.5 整体测试 …… 147
6.6 图形用户界面测试 …… 153
6.6.1 基本窗口测试 …… 153
6.6.2 使用菜单 …… 156
6.7 强度测试 …… 160
6.8 系统测试 …… 160
6.9 规模测试 …… 163
6.10 回归测试 …… 163
6.11 小结 …… 163
第7章 调试 …… 164
7.1 使用调试工具前的准备 …… 165
7.2 启动调试工具 …… 165
7.2.1 首先启动调试工具 …… 165
7.2.2 将调试工具联上运行中的

应用程序 …… 165
7.2.3 使用调试工具和核心文件 …… 166
7.3 调试工具的子命令 …… 166
7.3.1 使应用程序停止 …… 167
7.3.2 运行应用程序 …… 167
7.3.3 检查应用程序 …… 168
7.3.4 检查数据 …… 169
7.3.5 确定逐行控制 …… 170
7.3.6 检查多线程应用程序 …… 170
7.3.7 别名 …… 173
7.4 调试实例 …… 173
7.4.1 实例代码 …… 173
7.4.2 使用调试工具 …… 175
7.5 小结 …… 186
第8章 移植 …… 187
8.1 移植到新的操作系统 …… 187
8.1.1 Microsoft Visual C++ 中的线程支持 …… 188
8.1.2 UNIX 中的线程支持 …… 189
8.1.3 Java 中的线程支持 …… 191
8.2 移植到新的硬件平台 …… 192
8.2.1 支持 Endianism …… 192
8.2.2 32 位和 64 位机器的比较 …… 195
8.3 移植到新的语言 …… 195
8.3.1 国际化和本地化 …… 195
8.3.2 应用程序国际化时需要考虑的问题 …… 196
8.3.3 单字节和双字节字符集 …… 199
8.3.4 宽字符串 …… 200
8.3.5 Unicode …… 200
8.4 将消息中的字符串本地化 …… 201
8.4.1 创建消息目录—— UNIX …… 201
8.4.2 资源文件—— Microsoft …… 205
8.5 开发国际化应用程序 …… 205
8.5.1 策划一个国际化应用程序 …… 205
8.5.2 确定接受哪些数据 …… 206
8.5.3 编写代码 …… 206
8.5.4 设计用户界面 …… 206
8.5.5 测试应用程序 …… 206
8.6 设计用户界面 …… 207
8.6.1 创建应用程序消息文本 …… 207
8.6.2 设计菜单和表单 …… 207
8.6.3 使用图标和位图 …… 207
8.7 建立可被移植的应用程序的目标结构 …… 208
8.8 小结 …… 208
第9章 应用程序生命周期 …… 209
9.1 写出源代码的文档 …… 209
9.1.1 一般的注释 …… 209
9.1.2 C++ 文件的文档 …… 210
9.1.3 C++ 头文件的语法 …… 211
9.1.4 Java 文件的文档 …… 212
9.1.5 源代码语句的安排 …… 213
9.2 组织项目的目录结构 …… 215
9.3 使用 make 工具 …… 216
9.3.1 选项 …… 217
9.3.2 操作数 …… 217
9.3.3 读取 makefile 和环境 …… 218
9.3.4 makefile 目标项 …… 218
9.3.5 特殊字符 …… 218
9.3.6 特殊功能目标 …… 219
9.3.7 后缀替换宏引用 …… 219
9.3.8 makefile 的例子 …… 222
9.3.9 可移植 makefile 的例子 …… 222
9.3.10 创建依赖条件 …… 227
9.4 使用源代码管理控制工具 …… 227
9.4.1 源代码管理控制系统 …… 227
9.4.2 SCCS 的例子 …… 231
9.5 错误报告 …… 238
9.6 改进需求 …… 238
9.7 修改记录 …… 238
9.8 回归测试 …… 238
9.9 小结 …… 239

第五部分 实例学习

第10章 实例学习1——一个模拟的公司 …… 242
10.1 项目需求 …… 242
10.2 用例 …… 242
10.2.1 用例模板的翻版 …… 243
10.2.2 Use Case #1——贷款申请 …… 243

10.2.3 Use Case #2——购置机器设备…… 245
10.2.4 Use Case #3——生产运营………… 245
10.2.5 Use Case #4——处理公司的账务 …… 247
10.2.6 Use Case #5——显示公司的详细信息 …… 248
10.3 分析文档——类的静态特性 …… 248
10.3.1 类图 …… 249
10.3.2 CRC卡片 …… 250
10.3.3 脚本 …… 250
10.4 分析文档——类的动态特性 …… 253
10.5 分析文档——系统的静态特性 …… 255
10.5.1 类关系图 …… 255
10.5.2 协作图 …… 258
10.6 分析文档——系统的动态特性 …… 262
10.6.1 活动图 …… 263
10.6.2 序列脚本 …… 267
10.6.3 序列图 …… 272
10.7 小结 …… 274
第11章 实例学习2——开发一个多线程机场管理模拟程序 …… 275
11.1 一次只有一架飞机 …… 275
11.2 一个停机位入口同时有两架飞机 …… 280
11.2.1 降落过程 …… 280
11.2.2 起飞过程 …… 281
11.2.3 修改后的降落/起飞过程 …… 282
11.2.4 修改后的Java代码 …… 283
11.3 一个停机位入口同时有三架飞机 …… 289
11.3.1 降落过程 …… 289
11.3.2 起飞过程 …… 290
11.3.3 修改后的降落过程 …… 291
11.3.4 修改后的Java代码 …… 291
11.4 更多的飞机——再增加一些停机位入口 …… 293
11.5 飞机在机场活动的整个生存周期 …… 295
11.6 最终的解决方案 …… 305
11.6.1 机场细节信息窗口 …… 305
11.6.2 Java代码 …… 306
11.7 小结 …… 306
附录 “哲学家”源代码 …… 307



第一部分　什么是面向对象

目标：

- ▶ 面向对象简介
- ▶ 学习面向对象的基础知识

第1章　面向对象简介

本章将讨论以下内容：

- 什么是对象
- 识别对象
- 类、属性和方法
- 面向对象的基础知识
- 将发现的一切形成文档

自从有计算机程序以来，程序员就一直在内存和外存容量的苛刻限制下“艰苦”劳作。尽管如此，程序员还是创造了许多令人惊奇的工程软件。他们能够利用最少的计算机资源来编制大多数的程序软件。在这样的程序中，除了必需的功能模块外，没有什么装饰性的或可有可无的东西。

后来，高性能的计算机越来越普及，它们拥有较多的内外存空间，编程也发展到一个较高的层次，不再对任一细节都斤斤计较。渐渐地，对程序员来说，有一点越来越清晰，那就是：要采用结构化的分析和设计技术来理顺编程中的混乱状况。这些结构化的技术具有革命性的作用，它们允许程序员用形式化的方法来编写应用程序。一个软件计划可顺利地被客户认可并最终被测试验证。遗憾的是，结构化技术引起一个问题，它们不允许最后完成的应用程序具有太大的灵活性，除非考虑到将来需要重写应用程序的重要部分。而面向对象技术就能够提供所需的灵活性。我们可以用家居来做个类比。

一般来说，我们之所以想到买房子，是因为感觉到目前的居室空间不够了。在目前的居室设计下，再也没有足够的空间来满足家庭成员的日常活动了。人们对于房子的需求会发生变化，就像一个企业对现有软件的需求也会改变一样。像对待房子一样，对于不能满足需求的软件，一般有两种解决方法：

- 你可能摒弃旧软件而购买新软件，当然，这需要再培训。遗憾的是，旧软件没有转卖的价值，另外，编一个新软件一般会比建造一座新房子更费时间。
- 另一个廉价的选择是在现有软件的基础上做一个扩展件。你可以用最低的培训代价获得新增的功能。

旧软件（或性能不佳、设计不好的软件）就像乡村中的一所旧房子。房子可以“生长”以满足主人不断变化的需求，如在已有房子的基础上增加更多的房间。新的扩展部分可附加在原结构上实现特有的目标。但是，扩展也要有空间才能实现，所以，从一个房间得到另一个房间也成为一种冒险。一些软件与这些房子类似：增加的新特性并不改进设计，而只是增强现有的功能。软件的特色多多，但它如同一场梦魇，任何人都试图改进它的功能和性能。改变软件的功能就像改变旧房子中的房间一样困难，改进的费用可能超过房子本身的价值。

对软件的解决方法要比重新设计改造旧房子（尤其要保持一定的空间次序）容易实施。对软件来说，你只需将错误废弃并从错误中学习，争取下一次创造出更好的软件。

一个现代建筑要考虑到人们如何居住、如何享受建筑提供的功能，那么，采用面向对象理念的程序员会考虑到如何让相关的信息同步变化、提供编程灵活性，如何使设计能够适应可能的变化。尽管结构化的方法论不像面向对象方法那样具有严格的学习曲线，但它不能带来适应现代商业变化的编程灵活性，也不能经受住“重复不断”的变化。

1.1 结构化技术和面向对象技术的比较

对那些使用早期的结构化方法的编程人员来说，掌握面向对象的编程技术可能会有一些精神上的压力。但是面向对象理念的优点远远超过学习过程中的困难。以下是一个结构化设计和一个面向对象设计的对比实例。

面向对象的技术出现以前，一切都是围绕结构化体系模型设计的。这种模型背后的推动力量就是“过程”，它无所不在。这种理念特别流行，因为模型化后的一切就已经是过程了。图1-1展示了一个大大简化了的订单处理过程。

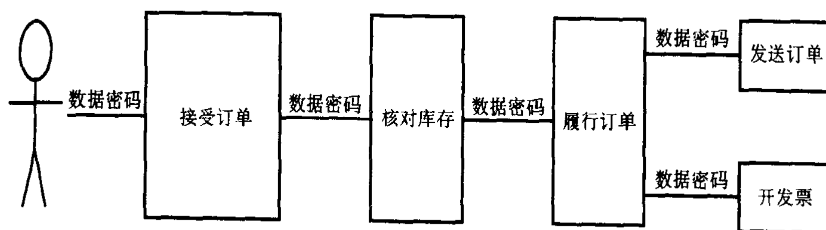


图1-1 一个订单处理的简单例子

图1-1中并没有显示这样的事实：每一个过程都必须与数据库交互。通过使用数据库，每个持有密码的过程都可跟踪客户的订单。通常，对信息仓库的交互接口会提供一整套普遍意义的信息处理方法——例如，setSomething（设置访问信息的密码或某些属性值）。考虑到一些比较特别的问题，最后的接口会比应用程序使用的大得多。

相反，面向对象技术的驱动力是信息——它在系统中到处“流动”。相关的信息都“绑”为“一束”，每个新的订单都是系统中的一“束”新的信息，这就像是在数据库表中增加一行记录。

另外一个变化是，尽管不像外部过程那样易于直接处理信息本身，但每一“束”信息都有一套提供接口的方法，以允许其他“束”按照可控制的方式来处理信息。这如同数据库本身具有处理信息的方法一样，知道如何处理信息。

至此，我们就有了“对象”的基础，即信息束，它包含以可控制的方式处理信息的方法。

二者之间最后一个不同之处在于，如果一个不同类型的订单进入系统，面向对象系统更能适应这种变化。系统只需从现有订单派生出一个新订单，改变其中的数量和类型数据，并根据情况采用适当的方法。这就是面向对象系统的运作方式，图1-2展示了在面向对象系统中的订单处理的工作模式。

正如你看出的一样，这就是在系统中“流动”的对象。在此，过程的概念被对象的状态来标识——也就是说，原来那些被描述为“核对”的过程，现在被描述为“一个正在根据自己的

信息执行核对方法的对象”。旧的过程的完成也标志着一个新过程的开始。而在面向对象系统中，一个对象本身可以标明并改变自己的状态，如，从“正在核对的”到“核对完成的”。

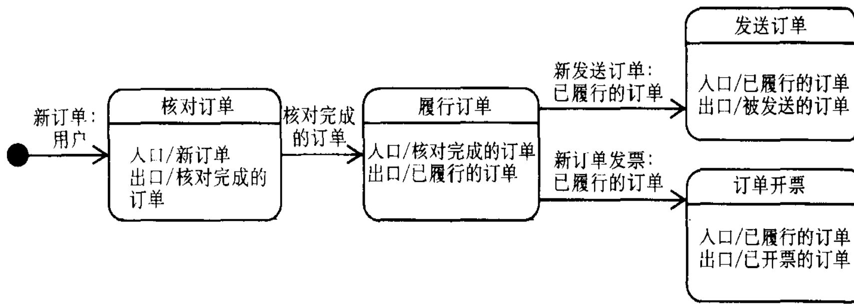


图 1-2　面向对象系统中的订单处理的工作模式

1.2　什么是面向对象

使用面向对象技术可使编程者全面理解问题模型的环境。对问题中的各个组件进行分别标识并细化各组件之间的关系，就可达此目的。

我们以一个空中交通控制系统（一个空中交通管理员将飞机控制权移交给下一段飞行路线的管理员）为例，见图 1-3，用面向对象技术，可以较容易地形成问题的抽象模型：

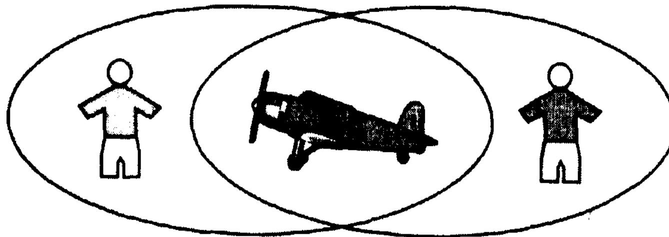


图 1-3　空中交通控制系统

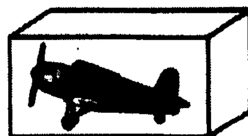
在这个例子中，第一个管理员只需向第二个管理员传达足够的信息，以便他能够确定飞机的位置。所提供的信息必须足以识别飞机（即飞机发出的信号），并要提供通信使用的频率。使用面向对象技术，这些事务场景可以准确地模型化，如图 1-4 所示。

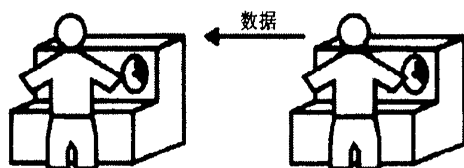
但是，如果第二个管理员需要的只是某一部分信息，如飞行高度和信号，利用结构化技术就能够满足要求，所以，在理想情况下，面向对象技术的优势不见得会显示出来。

现在，请考虑这样一种情况：管理员需要确认全体机组人员的某些细节，如机组人员是否满员。如果整个问题已经使用面向对象技术模型了，那么处理这个需求的功能就被包含在对象（在本例中，对象就是飞机）中。对象就像最初的数据项集，允许实现任何需求。在图 1-5 中，我们在飞机外画一个矩形盒，表明它是一个对象。

使用面向对象技术，飞机对象本身包含机组人员的信息，它自己能够确定目前的机组人员数量是否符合正常情况并做出适当的答复。

对结构化系统来说，传达给空中交通管理员的最初信息只是信息而已。任何待澄清的问题都需要由代表飞机概念的系统来处理，见图 1-6。要回答管理员的问题，系统需要确定飞机的类型，再确定这种飞机需要多少机组人员，最后与目前机上机组人员的实际数量相比较。





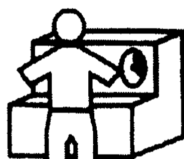


图 1-4 空中交通控制的面向对象技术

图 1-5 作为对象的飞机

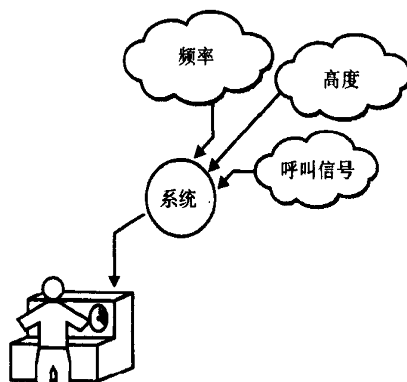


图 1-6 代表飞机概念的系统

看一看更为复杂的情况（这常常会引起系统的重新设计），如飞机有多种用途，进而可能有多种配置，如货机、客机或外交专机。对于每一种配置，“正常机组人员”的概念都是不同的。

空中交通管理员可以直接询问飞机信息，而不必从一些信息片段去推出另一些信息片段。也就是说，面向对象技术可以提供更符合客观现实的模型，系统中各对象之间的交互活动是符合客观事务的本来面貌的。

1.2.1 面向对象技术是如何与用户关联的

从以上的例子可以看出，对象无所不在。另外还可以看出，对涉及到的人来说，面向对象的理念是自发的、源自天然的。所以，面向对象技术被设计得更遵循思维的自然方式。也就是说，用到的术语、定义、符号应该与每个人相关，包括用户（客户）。

基于面向对象的分析和设计技术的这些特点，用户可以从始至终参与到系统的分析中，包括找出系统中的对象、定义对象等。用户也能参与系统的设计，因为在讨论对象之间的交互作用时，大家都使用同样的术语、概念。用户还可以参加系统文档的创建、编写，因为他们理解已有对象的明确含义。他们还会在已有对象交互作用的层面上理解各个设计阶段，并最终跟踪对象和系统的设计，一直到实现阶段。

1.2.2 面向对象技术的其他优势

使用面向对象技术，还可获得其他一些优势，如下面谈到的源代码复用、源代码的可维护性、创建已有的对象和使用纯粹的面向对象的语言。

1. 复用

一旦系统被完成，不但设计和实现的方法成为一笔财富，创建的对象也将随项目一起继续

生存下来。例如，某项目中开发了一个支持图形接口的库，在未来的系统中，我们就不需要再重新开发这样的接口库，而只需调出以前的接口库就可以了，因为它是可以复用的。

复用是采用面向对象的方法和语言的一个推动力。这种信念逐渐变成了强迫性的，以致于人们经常将复用等同于面向对象。

复用有助于降低未来项目的成本，缩短未来项目的开发时间。从表面上看，复用似乎是一件不太难做的事情，但它常常不能按计划进行，因为几乎没人能写出非常符合未来需要的代码，所以总是需要重做一些工作。

虽然面向对象技术允许软件复用，但使用面向对象技术设计的软件不必非被复用不可。其实，软件的复用早在面向对象技术出现之前就存在了。有些公司向其他公司出售事务处理软件库，这也就是允许软件复用的一种体现。

不管用什么方式看待面向对象技术，复用都不是自动实现的。代码只有满足下面的条件才能被复用：

- 已有的代码必须有一个可管理的存储库（就像图书馆管理图书的方式一样，要有类似于图书管理员的人员，对提交到库中的代码检查它的可复用性）。
- 代码必须有完整有效的文档。
- 代码应该容易使用，也就是说它的对外接口要自然、支持较多的使用方式。比如表达日期的类 date，如果它只支持“MM/DD/YY”这样的日期格式，被复用的价值就不太大；如果它还支持另外三种日期格式“MM/DD/YYYY”、“DD/MM/YY”和“DD/MM/YYYY”，它就更容易被复用。
- 代码的存储库要采用适当的手段加以宣传，让相关人员易于获取。如在 Internet 上发布，或用 SDK（Software Development Kit）的方式定期更新版本。
- 代码的存储库要易于访问。不容易访问的库将使用户望而生畏，减少代码复用的机会。
- 鼓励使用者反馈意见。如果代码不能满足使用者的需求，就会出现反馈意见。可以改进现有代码，也可以再编写新的代码（当然新代码要加入代码存储库），以满足使用者的合理需求。
- 鼓励使用代码存储库，鼓励向存储库贡献新的思想或代码。

要在两种成本之间掌握好平衡，即自己实现可复用代码并管理存储库的成本与购买同样功能的具有技术支持和使用许可的代码库的成本。

2. 可维护性

维护任何源代码的最大障碍之一是要极力去理解两个问题：每一个组件在做什么、与系统中其他组件有什么关系。

从必要性上讲，前者应该说是一个技术问题。如果假定：因为某人是技术专家，他或她就可以比较容易地理解其他程序员编写的代码，那就太简单化了。比如，举一个例子，他们在代码中看到一点以前从没遇到的细微差别，那就需要花相当时间去理解这些差别可能引起的所有隐含效果。在这种情况下，文档和注释可能就没什么价值了。

问题的第二个方面关心代码与系统的其余部分如何关联。一段代码可能就是一个技术杰作，它可能具有很高的编码质量，有丰富的文档和有用的注释，但如果不能提供它为什么存

在、它能处理哪些事务或者如何使用它等线索，那么，这段代码与一个黑盒子没什么两样。

问题的第二个方面与问题域直接相关。只有充分理解问题域，一段段代码才能被放入系统的关联网中。面向对象应用程序的一个关键特征是可以向一个问题域的专家咨询有关对象之间的关系，因为它们与问题域的实质内容相关。

从以下例子看出，我们需要问题域专家的帮助：

> 一个程序员小组被要求维护一个国际象棋比赛的程序。这个系统有很多复杂的算法。除此之外，系统中还设计了很多复杂的对弈策略，需要请一位象棋大师才能确认系统是否真的在按设计的方式运行。

3. 在现有对象的基础上构建应用程序

面向对象技术的令人愉悦的一面是一切都是模块化的、有模板的。开始时，先要收集需要努力实现的系统中的各种对象，这些对象相互关联，形成系统的基本架构。接下来，设计过程将构造一个更大的模块——你的应用程序，该模块具有确定的接口，可被更大的应用程序使用。最初的能够实现一些常规处理的简单对象最终突然变成了复杂的应用系统的一部分。

4. 纯语言

分析首先要注重当前系统的需求，但好的设计要能够允许系统“生长”。面向对象的分析和设计技术是概念性的——即，在系统中发现的对象可用多种方式实现。倘若分析和设计完全遵从面向对象的原则，使用者就可以用任何编程语言实现应用程序。显然，使用面向对象的语言更能得到完美的结果，而使用其他语言将使实现过程变得更困难，因为这些语言可能不支持面向对象的某些基本特点。

比如，像 Visual Basic 这样的编程语言是基于对象的，但它不支持面向对象的所有特征。这并不意味着不能使用这种语言，只是在使用这种语言时，设计实际解决方案的过程会更困难。

1.2.3 面向对象技术的一些弱势

没有一种简单的面向对象方法论能满足任一类型的系统和项目的要求，所以，我们的一个诀窍就是以一种方法论为基础，必要时再参考其他可用的技术。虽然本书遵从 UML（通用建模语言，Unified Modeling Language）标准，但也用到了其他的技术，如将在第 2 章谈到的 CRC 卡片（CRC Card）和脚本。

但是，现有的分析和设计工具比较昂贵，而且不允许用户引进另外的表示法和技术。在这种形势下，分析和设计人员常常采用图形程序包或桌面出版程序包，正如我写本书采用的策略。

1.3 什么是对象

对象是一个真实的或抽象的元素项，它包含信息（即描述对象的属性）和用于处理对象的方法。任何对象都可包含其他对象，这些对象又可包含其他对象，直到系统中最基本的对象被揭示出来。

例如，小轿车可被看成一个对象，它包含许多组件，其中之一就是发动机。发动机可被看成一个对象，它也包含其他对象。至于对象要细化到哪一级，则取决于系统的需求。

1.3.1　识别对象

识别对象时，可以采用以下几个有效的技术：

- ▶ 仔细阅读功能说明书并在所有名词下划线。这有助于用潜在的类覆盖对系统的分析结果。我之所以用“潜在的类”这个词，是因为，比如，若“money”被识别为支付工资的应用程序中的一个对象，这时，用一个数表示“money”更容易。但是，在另外的情形下，“money”可能变成一个需要指定货币单位、不只是一个数量的类。这种“情形”常被称之为问题域，本书将在第2章中讨论。

一些必要的表示法的快速入门

每一个对象都有一个类（即对象的定义）与之对应，类将在本章随后的部分中讨论，但现在，我们引入本书中使用的一些表示法。这些方法基于UML Version 1.4。

- ▶ UML中，类的表示法为ClassName。
- ▶ UML中，一个类的非特定的对象表示为:ClassName，其中的冒号“:”用于界定非特定对象。
- ▶ 一个非特定类的对象表示为ObjectName:，注意，类是非特定的。

- ▶ 寻找实在的事物即那些与系统中的其他对象交互的事物，如：
- ▶ 人员（管理者、雇员、家庭、朋友）
- ▶ 地点（家、工作地点、度假目的地）
- ▶ 文件（购物单、法律协议、出生证或结婚证）
- ▶ 寻找对象之间的关系。

1. 实例：寻找一个家庭中的对象

一个**:Male**（男）对象和一个**:Female**（女）对象形成一个通常被描述为“婚姻”的关系。婚姻状况可以是**:Male**对象和**:Female**对象的属性。**:Male**对象和**:Female**对象中包含互相指向对方的句柄，如图1-7所示。

图1-7　一个:Male对象和一个:Female对象的结合

一般需要有一个法定记录来维持婚姻关系。这种情况下，关系“married”（“已婚”）被表示为一个对象**:Marriage**，如图1-8所示。

如果child（孩子）出生了，那么孩子就与夫妇一起组成了一个Family（家庭），一种新的关系对象出现了，见图1-9。

但是上述模型只表示了家庭成员，并没有表明他们与家庭的关系。UML中有一个用于此目的的表示法，即链，它可用来强调特殊的关系。链的表示方式见图1-10。

2. 另一个识别对象的实例

下一个例子涉及一组公司职员，其中每一位都扮演公司的不同角色。从分析的角度看，这

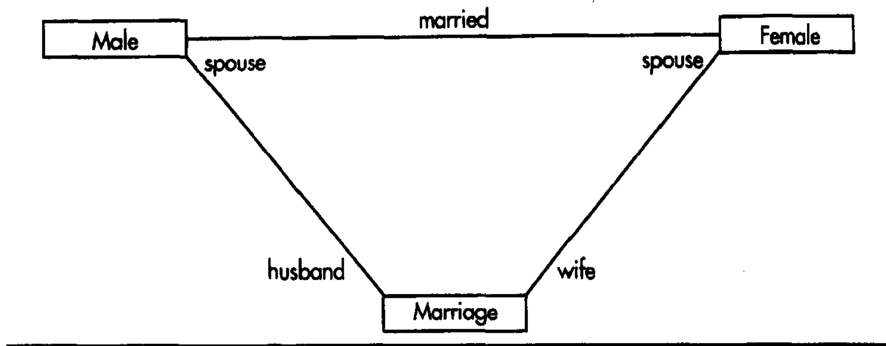


图 1-8 一个:Marriage关系对象

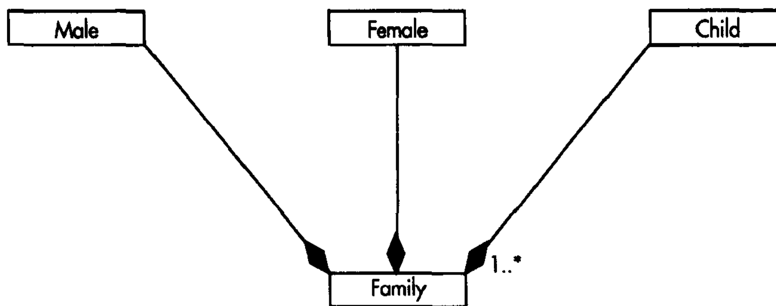


图 1-9 组成家庭的新的关系对象

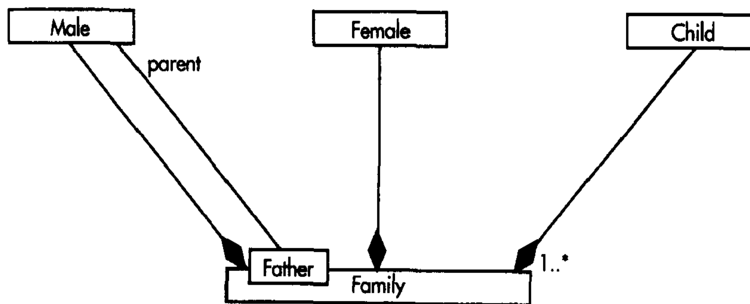
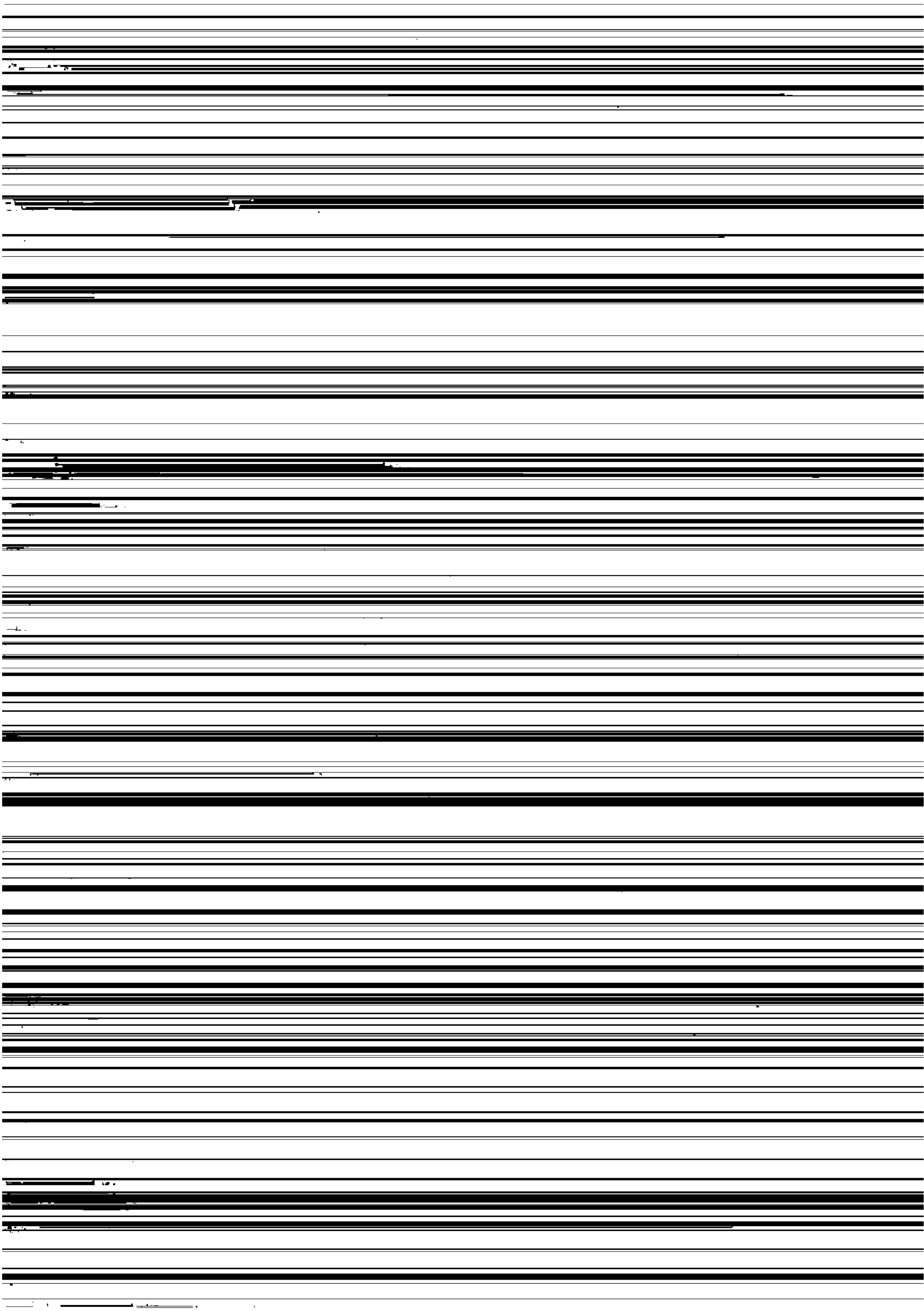


图 1-10 链的表示方式

个问题的实质是如何识别并表示每一位职员。

- ▶ 公司的每一位职员如图 1-11 所示。图中，对职员没有任何说明，如职务、所在项目名称、工作职责或任何其他特殊关系等。
- ▶ 一方面，对大多数公司来说都有许多项目组，职员都属于一些项目组，如图 1-12 所示。身着横条上衣的职员属于会计部门，身着白色上衣的职员属于公司网站制作部门，而身着竖条上衣的职员则属于公司的仓库管理部门。
- ▶ 另一方面，在所有的项目组中，存在三个独立的团队。团队中的职员如图 1-13 所示。第一个团队（身着斜条纹裤子）负责开发，第二个团队（着白色裤子）负责品质保证，而最后一个团队（着斜格裤子）负责技术支持。



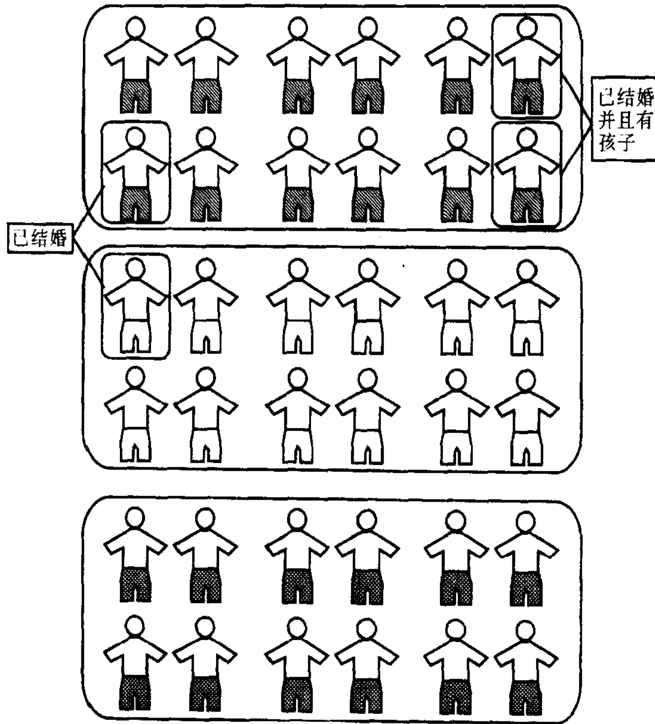


图 1-15 相互之间有特殊关系的职员

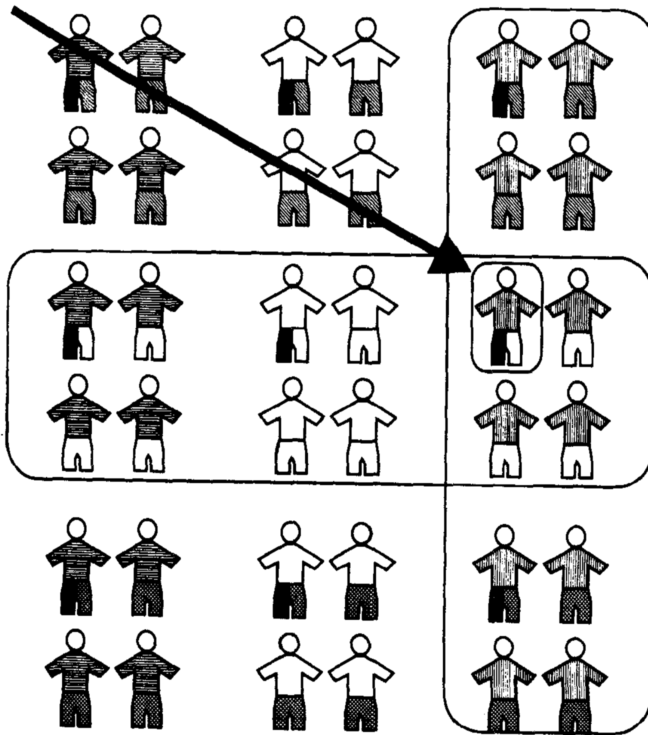


图 1-16 特殊显示的职员容易被描述

- 一位职员
- 一个团队的成员——即品质保证团队的成员
- 一个项目组的成员——即仓库管理部门的成员
- 换个说法，以上任一角色或所有角色

总之，可能被考虑和应该被考虑的任何事情都可以成为项目中的对象。只有对所有情况进行了全面考虑，才能对系统做出合理的分析。

在项目的分析阶段，可能会将具有同样特点的对象合并或删除，但每一个决定都依据项目的具体环境或特点而定。

对于如何描述图1-16中特殊显示的职员的问题有多种答案，但每一个答案都依据项目的具体情形而定。其中一个答案将在第3章多重继承部分讨论。

1.3.2 属性

属性就是对象的一个特征并在系统的具体环境中有其对应的数值。回头看图1-7。婚姻状况可作为**:Male**对象和**:Female**对象的属性描述，见图1-17。

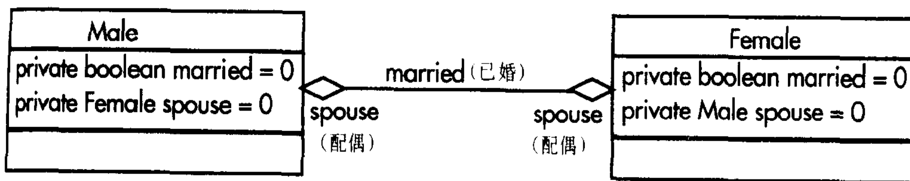


图1-17　对象的属性描述

回头看图1-8。**:Marriage**对象属性可能是什么时候结婚、在什么地方结婚、哪个**:Male**对象和哪个**:Female**对象结婚，如图1-18所示。

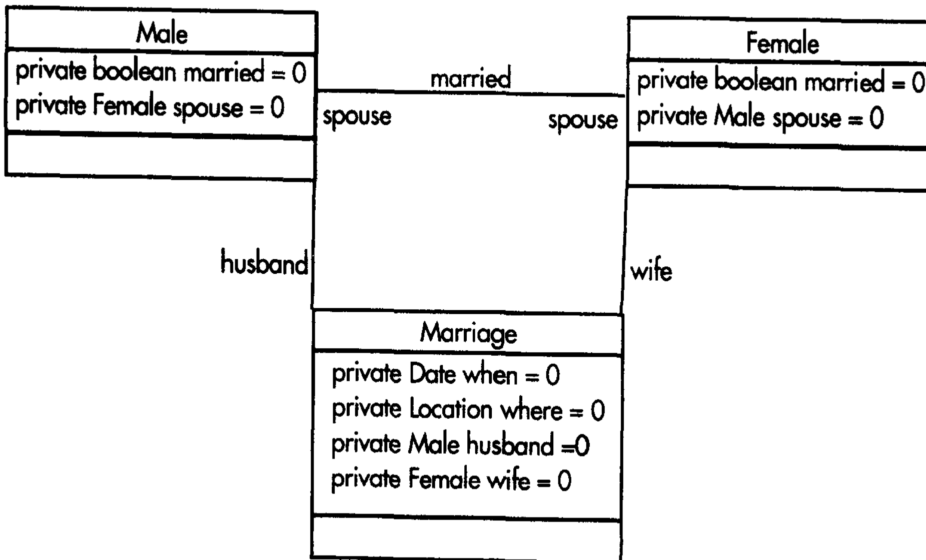


图1-18　:Marriage对象的属性描述

还有一个包含属性的对象的例子是MyCar:。MyCar:的属性有多个，它们是制造商、型号、车龄、颜色、发动机大小等。

当描述MyCar:时，我需要知道我的听众需要的信息。如果听众只是想知道我能否送他到机场，他并不关心MyCar:的颜色。另一方面，如果MyCar:被人偷去，警察很可能非常想知道它的颜色。所以，你就明白了，MyCar:有多个属性，每个属性有一个数值：

MyCar: Attributes: make, model, engineSize, color

再举一个例子，以说明属性值是依赖于具体情形的。假设有两座小房子，它们毗邻而建，共用一面墙。天长日久，其中一座小房子的主人也把另一座房子买下了。他们推倒了公用的墙，把两座小房子合并为一座大房子。这样就存在三张房屋图纸：原有的两座小房子的图纸和新的大房子的图纸。

从新房子的角度来讲，原有的图纸对目前的住户没有实际意义，但是，从保持历史记录的角度来讲，这些信息却有很高的价值，因为它们展示了地产的变迁。

1.3.3 方法

方法（Method）是一个对象允许其他对象与之交互的方式。给一个对象定义的方法被写为并被称为“接口”。接口用于规定如下几个方面的内容：

- 对象支持什么样的方法
- 方法是如何被使用的
- 还需要哪些附加信息
- 预计会发生什么事件（这个方法做什么）
- 向调用者返回什么结果（如果有的话）

对象之间通过发送消息进行交互，消息激活已公布的方法，而方法则代表发送方对接收方实施一个功能。“激活一个方法”指发送者等待接收者处理这个方法并返回，而“发送一个消息”指发送者可以不等待。这样可允许并行动作的发生。

每个消息都必须与一个公布的接口相匹配。消息的格式向每个方法提供了被称之为签名的东西。这个方法签名是编译器必需的，具有如下的形式：

<方法的名称><附加信息或参数>：<返回的数据类型>

对象可能公布一个与另一个方法同名但不同参数的接口。这表明，这些方法具有同样的功能，但使用不同的输入。方法签名会向开发工具和运行系统提供足够的信息，来确定调用哪一个方法。

现在可以向前例中的MyCar:加入方法了，如下所示：

MyCar:	Attributes:	make, model, enginSize, color
	Methods:	startCar (), driveCar (), changeWheel ()

对象Me:提供一个方法 *IsItYourBirthday* ()。例如，向Me:询问：今天是不是我的生日？可用如下的写法：

```
result=Me.IsItYourBirthday ( "TODAY" );
```

如果今天是我的生日，结果将返回“YES”，否则，返回“NO”。

方法的重载

如果一个方法的名称在同一范围内由于不同的原因被使用，这个方法就被重载了。表 1-1 中显示的方法 *Age* 就是这样一个例子——它在三种不同的语言中被重载。选择使用哪一个方法是根据采用的方法签名确定的。返回的数据类型和给定的输入参数将被检查，以确定使用正确的方法。

表 1-1 方法的重载示例

	C++	SMALLTALK	Java
get	int *Age* (void);	*Age* ^Nage	int *Age* ()
put	*Age* (int newVal);	Age: *val* Nage = : val	*Age* (int newVal)

1.3.4 对象状态

对象状态是一个对象在生命周期过程中的一个状态。比如，一个汽车轮胎要么是充满气的，要么是没气的，也就是说，汽车轮胎有两个状态。一个对象状态可能是一个方法的返回值，或者可被用于在方法中表示一个对象。复合状态是由多个简单状态组成的状态。从概念上说，一个对象会在某状态维持一个小的时间片段，但是，从语义学上说，状态既可包括转眼飞逝的瞬时状态，也可包括非瞬时的过渡期状态。

状态图是图形化的相互关联的规则的集合。它可用来将一个对象正在进行的活动模型化。状态图是 UML 的一部分，将在第 2 章中讨论。

向前例中的**MyCar:**加入状态，描述如表 1-2 所示。

表 1-2 MyCar:的属性、方法和状态

MyCar:	**Attributes:**	**make, model, engineSize, color**
	Methods:	startCar (), driveCar (), changeWheel ()
	State:	stopped, running, flatTire

1.3.5 类

寻找到了一个对象后，第一个任务就是记录下来。这可通过使用类即对象定义来实现。类类似于使用手册，它告诉使用者：使用这样的对象可以做些什么，并列出可使用的方法、这些方法需要的参数、以及返回的结果。

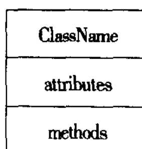
正如前面提到的，一个对象的定义包括：

- 属性
- 方法
- 状态

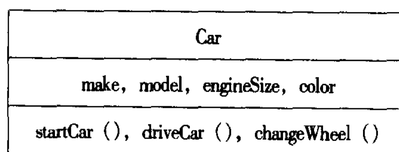
定义了若干对象后，可能出现这样一种情况：它们的属性和方法均一致。这时，可以定义一个类来描述这些对象的集合。再引用前面的例子，公司中的一个人可被定义为该公司的一名职员，我们可以定义公司职员类，这样，公司中的每个人都是一个实例，也就是类定义 **Employee** 的一个对象。

接下来再看，**MyCar:Car**的属性取决于我选择的汽车，除非我选择了法拉利（Ferrai），因为法拉利一般都是红色。这样颜色不再是一个对象的属性，而成为所有法拉利的属性，它成为一个类（所有的法拉利）属性，而不再是实例的属性。

一个对象的特性可以总结在类定义中，UML 表示法中对一个类的表示如下：

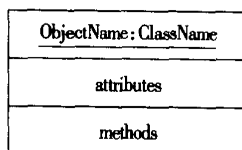


对于类 **Car**，它的属性以及可用的方法表示如下：

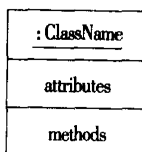


类支配可使用的方法，而实现则支配方法是如何工作的。另外，**MyCar:Car**是类 **Car** 的一个具名的实例（即对象）。

对象的表示法显示如下。实例化的对象的基础——类——被表示为包含在对象中。将类表示作为对象表示的一部分，有助于明确对象的根源，因为它明明白白地将对象源自的类表示出来了。在后面的讨论中，这个表示法将被进一步扩展，将继承也包括进来。



至此，以前介绍的对象终于被认识了。也就是说，它们是具名的对象的例子，如**MyCar:**。在 UML 中有一个专门的表示法，用来表示尚未命名的对象实例，即匿名对象，如下所示：



1.4 面向对象基础

数据抽象和封装允许系统中的信息以良好定义的实体的形式存在，继承提供了一种在相关

的类中共享属性和方法的机制，多态性和重定义则具有这样的能力：无需重新编写类的处理代码，就可交换不同类的对象。

除了引入面向对象的概念之外，本节也将强调清晰而准确的文档的必要性。

数据抽象和封装

数据抽象是指从丰富的信息相关的数据中的提取。将相关的数据保存在一起以便处理，这一点很重要。同样重要的是从特别的细节中提取普遍意义（或平常）的数据。在一个人事处理系统中，使用“人们”要比使用那些特定的人名如“Tom”、“Dick”或“Harriet”容易实现。

封装是将相关的数据隐藏在接口方法中。这些方法允许访问数据，并对数据进行处理。登录窗口就是一个封装的例子，它可以防止不速之客访问你的计算机资源。登录窗口就是操作系统提供的隐藏你的计算机资源的一个接口方法。

数据抽象和封装的前提如下：

- 使用者不知道或不应该知道对象是如何实现的。
- 使用者不应知道对象实际上的复杂或简单程度。
- 使用者只能使用被提供的方法。这些方法给出实施操作的必要的接口。

实质上，一个对象也不过是封装和数据抽象罢了，见图 1-19。

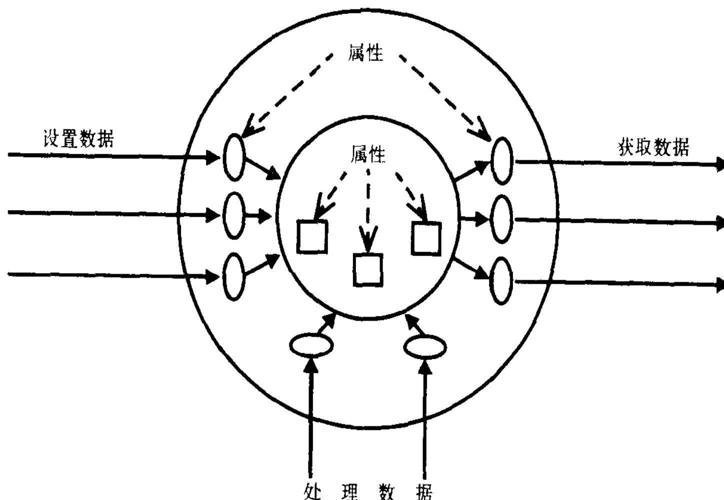


图 1-19　数据抽象和封装

图 1-19 说明了对象的属性是怎样通过方法被访问的。方法有多种类型，它包括向属性赋值的方法、获取属性值的方法以及以某种方式处理属性并返回一个合成结果的方法。

例如，在对象<u>ME:</u>中有一个属性 dateOfBirth，可以直接访问方法<u>ME:</u>getdateOfBirth（）来得到出生日期。

方法<u>ME:</u>getAge（）返回<u>ME:</u>的年龄，可以通过访问特定的年龄属性得到年龄，也可以用属性 dateOfBirth 与当前日期相比较来确定年龄（按年还是按月取决于使用的实现手段）。本例中的年龄是一个可被衍生的元素，它可从其他属性计算得到。UML 表示法中，在属性名前加一个斜杠（/），表示它是可衍生属性。

下面又是一个说明数据抽象和封装工作模式的例子。假定每周你都到同一个商店去买日常用品，你知道如果走到通道1再回到通道2，就能拿到要买的东西并能正好装满购物篮。

在一个不变的世界里，面向对象技术没有太多优势。面向对象技术提供一个媒介，你不需再提着购物篮，穿越一个个通道亲自去取货物。在这个购物的例子中，媒介就是商店的店员，他将替你取货物。用编程术语来说，媒介就是一个接口，即一套被定义的、允许使用者与对象交互的方法。

面向对象真正的优势在一个变化的世界中更能显现。如果商店的布局改变了，店员仍能替你找到货物，你无需关心新布局的任何情况。与此类似，尽管对象的结构发生了变化，其接口仍能工作。对结构调整的对象来说，需要改变的是接口实现功能的方法，而不是使用的接口方式，这样就可以减小对象变化对整个系统的影响。

媒介的另一个好处是：如果一个店员休假了或生病了，另一个店员可以代替他的工作。用编程术语来说，一个对象可以被系统中另外一个类型相似的对象代替。

现有的系统继续发挥作用，正如所有从旧系统中继承的遗传特点会继续有效一样，但也应考虑设计新的系统，以包容新的特色，新的对象也可能被引入系统。培训新的店员，让他们学会使用商店里的改进设施，学会通过电话或传真接收订单，但是对来店中直接递上订单的传统购物者的支持还是要保留的。

总之，面向对象提供了稳定不变的接口，尽管对象的结构可能被调整，甚至整个对象会被同类型的其他对象替代。这种机制给系统提供适应新变化、扩展新特点的能力。

1.5 继承

定义在面向对象技术中的“继承”是将多个类的共同特征抽象为一个更普遍的类，这个类成为这些特殊的类（或子类）的父类。

虽然属性和方法实际上是在一个继承等级结构的不同层次上实现的，对那些其类构成继承等级结构的一部分的对象来说，它的实现出现在对象本身的局部。也就是说，与使用局部方法不同，使用继承方法并没有限制方法的实现地点。当维护代码时，这就引起问题，因为方法实现的地点是不明确的。本章“文档”一节将涉及这个主题。

继承等级结构的根类是不从其他任何类派生的类。在图1-20中，根类就是Person类。

1. 继承Ⅰ：派生继承

图1-20的例子中，类**Person**被确定为类**Employee**的父类，类**Employee**被确定为类**Secretary**和类**Manager**的父类。

类**Person**包含整个继承等级结构中的共同属性和方法（见表1-3），位于等级图的最顶层。

类**Employee**包含与一个更特殊的**Person**类型相关的属性和方法（见表1-4）。

类**Secretary**是类**Employee**的一个特例（见表1-5）。

类**Manager**也是类**Employee**的一个特例（见表1-6）。

图1-20的图形表示法显示了上述例子中的继承等级关系。任何两级都构成一种关系。有多种描述这种关系的方式，见表1-7。

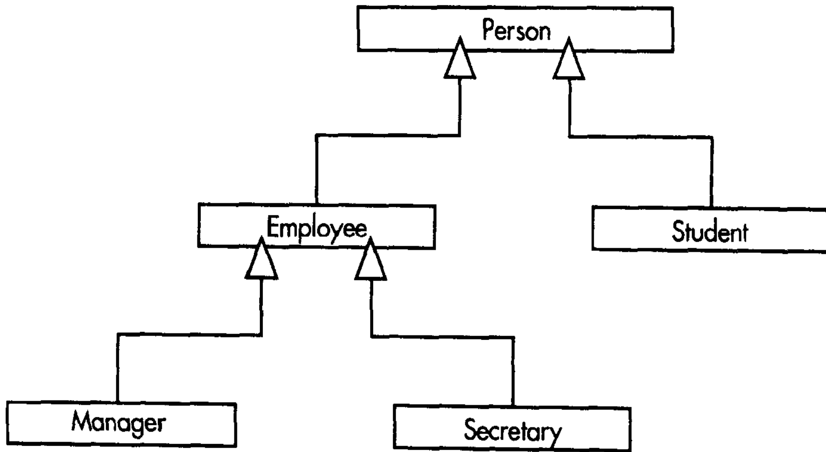


图 1-20 一个派生继承的继承等级结构

表 1-3 类 Person 的属性和方法

ClassName	Person
Attributes	name, address, age
Methods	askName (), changeAddress (), askAddress (), askAge ()

表 1-4 类 Employee 的属性和方法

ClassName	Employee
Attributes	employeeNumber, jobTitle, phoneNumber
Methods	setEmployeeNumber (), promoteEmployee ()

表 1-5 类 Secretary 的属性和方法

ClassName	Secretary
Attributes	groupManager, secretarialAndAdministrativeSkills
Methods	askToBookTravel (), askToArrangeMeeting ()

表 1-6 类 Manager 的属性和方法

ClassName	Manager
Attributes	groupMembers, groupResponsibilities
Methods	askToAssessGroupMembers (), planProjects ()

表 1-7 继承关系的描述

Person	关系	Employee	关系	Person
Person	是 ... 的父类	Employee	是 ... 的子类	Person
Person	从 ... 继承	Employee	从 ... 派生	Person
Person	是 ... 的上级类	Employee	是 ... 的下级类	Person

2. 继承Ⅱ：抽象继承

前一小节讨论了如何使类特殊化，即从父类派生子类，本小节将讨论如何从具有类似属性和方法的类中标识出父类，即找到并形成父类。

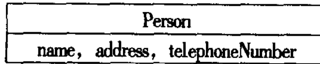
构建属性和方法表后，就可以看出哪些类具有相同的特征。这些特征可被提取出来，并组

织成一个新的类，即父类。这有助于减少因在不同的类中实现相同功能的方法而引起的重复劳动。我们从两个独立的类 **Lecturer** 和 **Student** 开始，它们的特征表示如下：

Student
name，address，telephoneNumber
coursesAttended ()

Lecturer
name，address，telephoneNumber
coursesGiven ()

属性 *name*，*telephoneNumber* 和 *address* 被提取出来，定义一个新类 **Person** 如下：



Person
name，address，telephoneNumber

新提取出来的类 **Person** 形成继承等级结构图的根，与类 **Lecturer** 和 **Student** 相连，如图 1-21 所示。

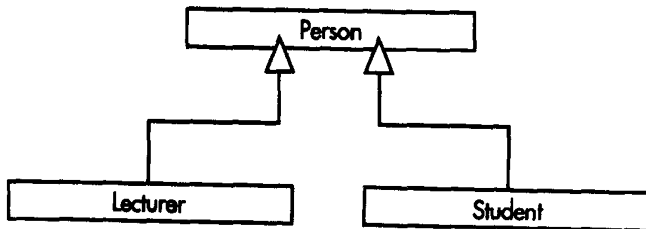


图 1-21　类 Person 作为继承等级结构的根

在这个例子中，称类 **Person** 被类 **Lecturer** 和 **Student** 继承，称类 **Lecturer** 和 **Student** 派生于类 **Person**。

图 1-22 表示了类 **Lecturer** 和 **Student** 是如何继承描述在类 **Person** 中的属性的。如前所述，从使用者的角度看，继承的特征和自有的特征并没有差别，使用继承的特征并不比使用自有特征需要更多的编程劳动。

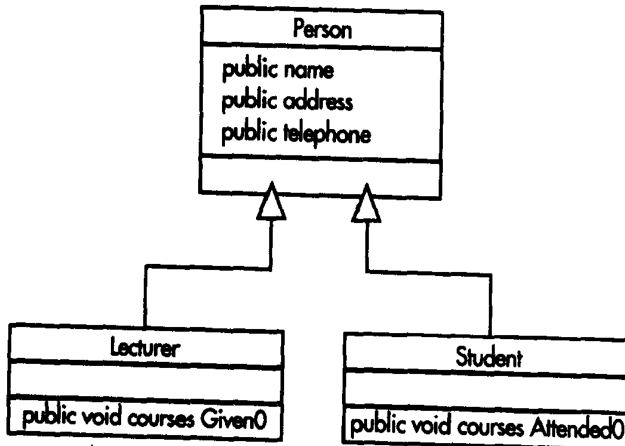


图 1-22　类继承中的属性继承

多态性

术语多态性（Polymorphism）有两种可能的定义。多态性的典型定义关注不同的类支持同

一个方法。一个经典的例子——至少是经常被提及的例子——是一个 **Shape** 类和两个派生类：**Triangle** 和 **Circle**。类 **Shape** 定义了方法 drawSelf ()，而类 **Circle**（一个派生类）重新定义了该方法，如图 1-23 所示。

使用了继承机制后，如果定义了一个指向 **Shape** 类的指针，这个指针也可以指向任何从 **Shape** 派生的类，这允许你将一个 **Circle** 对象赋给一个 **Shape** 类的指针。

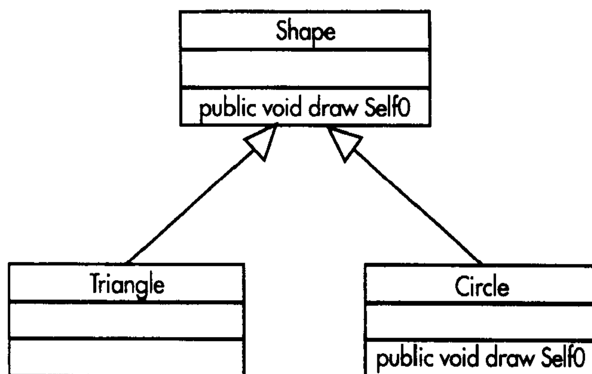


图 1-23 子类重定义父类中的方法

```
Shape *shape;
shape = new Circle ();

/*
** This will call the drawSelf () method of the Circle class as it
** re-implements the method described in the Shape class
*/
shape.drawSelf ();
```

多态性表明：虽然 **Shape** 指针指向的不是一个 **Shape** 对象，它也会调用适当的 drawSelf ()方法。本例中，如果类 **Circle** 重新实现了 drawSelf () 方法，则这个重新实现的方法将被调用。这是由方法重定义完成的，将在下一节中讨论。

多态性的另一个定义是一个对象支持多个接口，正如本例和下一个例子所示（见图 1-24）。

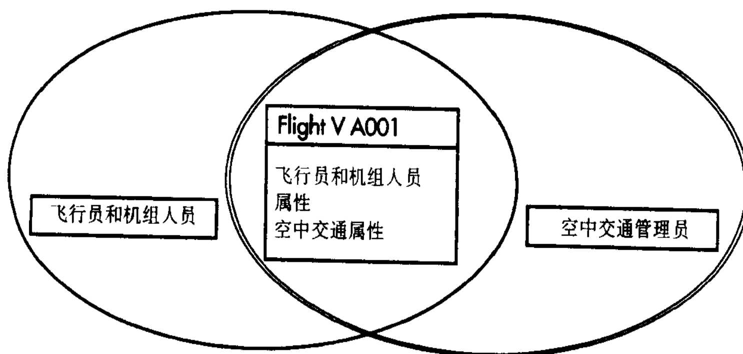


图 1-24 一个对象支持多个接口

多态性的这个定义允许对象Flight VA001:存在于不止一个领域。对乘客来说，这个对象是旅行的载体；对飞机驾驶员来说，它是被操作的对象；对于空中交通管理员来说，它是一个需要指导并在指定的机场起飞和降落的对象。当然不论在哪一种情况下，被引用的对象都是Flight VA001:，只是根据所在的情形不同，对不同的人有不同的意义。

1.6 重定义

重定义（Overriding）是一种机制，它允许子类为父类中已经实现的方法提供另一个实现。图 1-25 显示了一个重定义的例子。

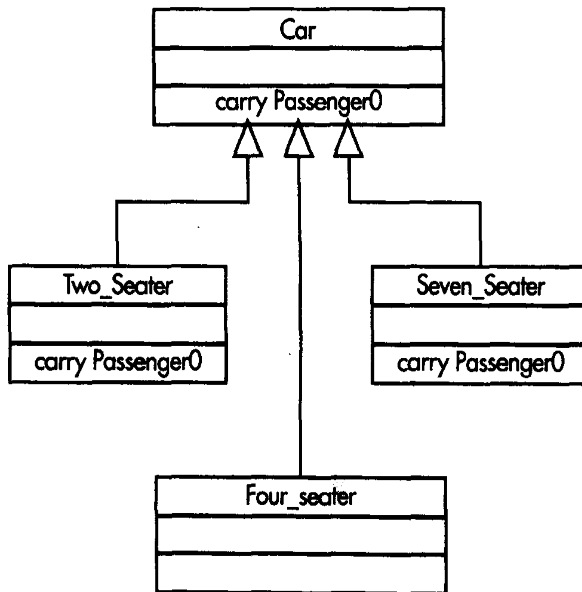


图 1-25 重定义基类中的方法

本例中，父类 **Car** 被三个子类 **Two _ Seater**、**Four _ Seater** 和 **Seven _ Seater** 继承。父类 **Car** 支持方法 *carryPassengers* ()，但这个方法的实现只是泛指。正因为如此，三个子类中有两个重新实现或者说重定义了这个继承的方法 carryPassengers ()。

当一个类的实例**Sports Car:Two Seater**被问及能乘坐几名乘客时，它将回答“1 名”，而一个类的实例**Mini-van:Seven Seater**将回答“6 名”。当一个类的实例**Car:Four Seater**被问及同样的问题时，它把回答留给从其基类 **Car** 中继承的方法处理，然后回答“2 名”。

1.7 文档

对需要维护面向对象编程代码的任何人来说，缺乏文档是最头痛的事。应用程序中类的数量可能是两位数甚至三位数，如果没有文档，这些类之间的关系就完全不清楚。实际上，甚至它们在应用程序中存在的理由也成为疑问了。

1.7.1 类的描述

类的每一个细节都要被描述清楚。可以将下面几个问题作为参考准则，以便写出你的类的

完整文档。

1. 为什么存在这个类

完全有必要说明类在整个系统设计中扮演的角色。要说明这个类之所以存在，是基于所做的系统分析的结果、所使用的设计方法的结果，还是因为一些特别的实现的要求。

2. 这个类与其他类之间有什么相关吗

这个类与系统中的其他类如何交互？这个类中的哪些事务处理是系统必需的？将这些处理放在同一个类中的理由是什么？这个类是否控制其他类或被其他类控制？

3. 这个类的属性是什么？它们各自描述什么

重要的是：知道为什么一个类要用某些属性来描述，这些属性的数据类型是什么，以及各属性在类中扮演什么角色。属性的声明方式也有助于定义类之间的关系。例如，在 C++ 中，指针可以表示与其他对象之间的松散结合，因为它可以被赋为空（表示不指向任何对象），而非指针数据表达了一种紧密结合，对象必须包含至少一个其他类的对象。

4. 这个类的方法是什么

每个类都定义了一套方法。有些方法用于向属性赋值；另一些方法用于获取数值。有些方法是共用的，而其他一些方法则只能为类本身所用。重要的是每个方法都要有完整的文档，文档的内容如下：

- 文档应该包括：这个方法是做什么的——从类的角度讲，方法担当哪些任务。
- 文档应该说明：方法所需的入口参数的数据类型是什么——如，不要试图把一个日期传给一个需要时间参数的方法，这注定是要失败的。
- 文档还应该说明：方法的返回值的数据类型，以及返回值的意义——例如，一个方法可能返回一个整数，但 0 可能代表成功而 -1 可能代表失败。
- 在什么情况下，方法会失败？这意味着：测试类的极限是接受标准的一部分。

最后一项往往得不到足够的重视。很多情况下，已写好（或将要写）的类发布后面临惨痛的失败，原因是类的作者按照他们考虑到的最大极限来测试类，遗憾的是，这些极限从未被告知类的使用者，他们对最大极限有不同的理解。

例如，有一个容器类，在一般容纳 10 个条目时可以正常工作，它的测试极限是容纳 20 个条目，但使用者在试图容纳 100 个条目时，遇到了灾难性的失败。应该告诉使用者测试极限，并且应给出提示：如果超出极限会有怎样的后果。类的实现中也应该有安全检查机制，遇到不安全因素时向使用者发出警告，并避免对象失败。

1.7.2 图的使用

另一个易被忽略的方面是图的使用。决不能低估表达清晰的图的作用。展现在视觉中的类继承等级结构图是那么地鲜明，使人马上能掌握要旨。面向对象的每一个方面都可用独立的图展示出来，这样可以大大减少由于在一张图中放入过多信息而引起的杂乱。再细致一点，可以把一个对象放在图的中心，有关的其他类呈放射状排列在其周围。关于这一个方面，第 2 章会进一步探讨。

1.7.3 继承

遗憾的是，前两个小节中讲到的类的描述和图的使用很少能清晰地强调继承的方法和数据变量。为此，一个图形化的类浏览器就显得很有价值。

图 1-20 的例子表示了一个小型继承关系图，它有三个层次。含有太多层次的继承关系图不太实用，并且无法在一个单页中全部显示。

1.7.4 编码规范

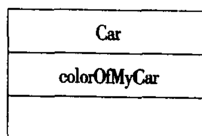
在编写文档的过程中，编码规范常常被忽视。但是，好的编码规范的一个重要的好处是：当阅读源码时，可以很容易区分出哪些数据变量和方法是类自有的，哪些是从父类中继承的。

属性和方法的表示法

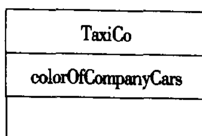
以下内容介绍 UML v1.4 中使用的表示属性和方法的表示法，这部分内容还包括临时变量和继承方法的表示法，其中有些内容 UML 标准尚未涉及。要介绍的内容有：

- 实例属性
- 类属性
- 临时变量
- 方法的表示法
- 继承的方法

实例属性　如果以小轿车的颜色作为例子，可以看出，小轿车的颜色多种多样，所以定义一个小轿车的类时，有必要分配一个指定轿车颜色的属性，以便确定其颜色的惟一性。这个属性要被定义为实例属性，因为它的取值会因每辆轿车而不同。描述实例属性使用的表示法是：第一个字母小写，而随后的每一个单词的首字母大写。如，*colorOfMyCar* 是一个私有的实例属性。对这个属性来说，每辆轿车都有自己的惟一取值。



类属性　相反地，当建立一个出租汽车公司的模型时，你可能为公司使用的小轿车定义一个类，它们有相同的颜色属性。这样，你能保证属于这个出租汽车公司的所有的小轿车能返回同一个颜色属性值。这类属性被称为类属性。它超出于类的实例而存在。实际上，就是没有任何实例，它也有属性值，就像一个新的出租汽车公司在买第一辆车前会决定车的统一标志。描述一个类属性的表示法的例子是 *colorOfCompanyCars*，一个用于描述出租汽车公司汽车的集体颜色的私有属性。



临时变量 表示法中所有其他的变量都是临时变量。它们用于保存调用方法的返回值或计算结果。描述临时变量的表示法如*colorOfMyCar*，与通常的属性类似，只是随后有一个下划线“_”。

方法的表示法 在项目的设计阶段，需要用表示法区别所使用的方法是类自有（也可能是重定义的），还是继承的。

图 1-25 的关于继承的例子中，用下面的表示法，方法 *carryPassengers* （）表示如下：

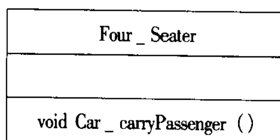
从这个表示法中，系统的设计者和开发者能确定有多少方法需要编码，再根据这个信息，确定工程的工期。

任何类方法都像类属性那样，用下划线标识。

继承的方法 继承的方法的表示法不是 UML 的一部分。我之所以包括了这种表示法，是因为有时它对编写类的文档有用。被继承的方法有一个前缀，该前缀与最初定义该方法的类名一致。描述一个继承方法的表示法如：

+ *Car _ carryPassengers ()*

下面显示的是一个继承方法，它也表达了方法是从那个类继承的。



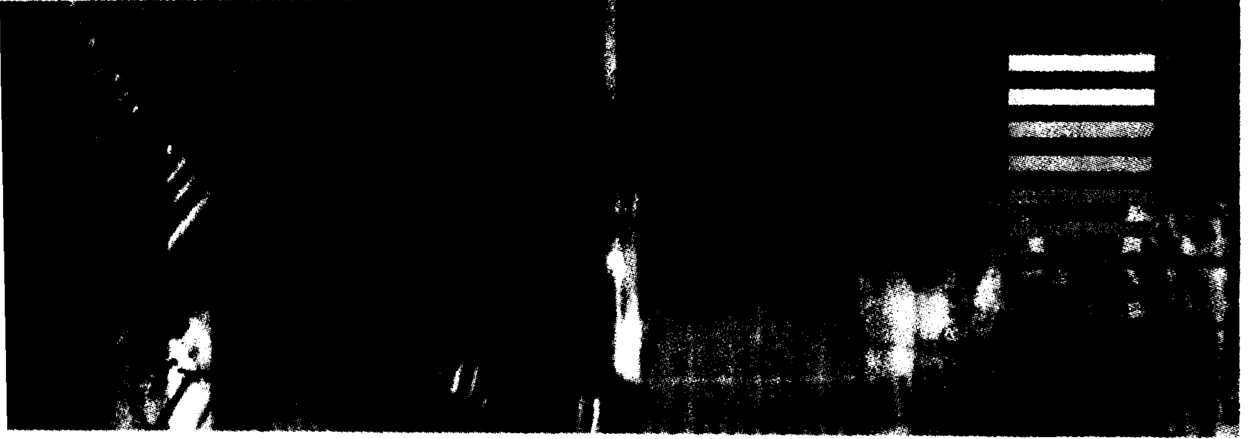
1.8 小结

本章引入了对象，并讨论了寻找和发现对象的方法。也谈到了对象的特征，它的属性、方法和状态。还有一节讨论了类和对象的设计图或使用手册。

本章还引入了面向对象技术的基础知识：

- ▶ 数据抽象和封装
- ▶ 继承
- ▶ 重定义
- ▶ 多态性

最后，本章讨论了文档化的有关问题。



第二部分 分 析

目标：

- 学习如何收集信息
- 描述 UML v1.4 支持的基本分析文档类型

第2章　分　　析

本章将讨论以下内容：

- ▶ 预分析、信息收集任务
- ▶ 确定对象到底由什么决定
- ▶ “用例”方法
- ▶ 如何使用由 UML v1.4 标准定义的分析记录方法

本章将向读者介绍面向对象分析的各个阶段。我们从讨论分析的真实意义开始，然后探讨如何将分析的范围控制在问题域的方方面面，接下来涉及的是使用“用例”分析来确定系统的高级需求，最后讨论的是分析记录的各个方面，包括类和系统的动态和静态特性。

2.1　预分析

“分析是很关键的，通过分析能够在当前可用资源的条件下，识别和定义值得解决的商业问题。如果没有分析，那是很危险的，极有可能设计和编制出一个不能满足用户需求的系统，这是因为，用户的需求没有被充分地识别和定义。”

“如果你不能准确地知道用户需要什么，系统就不能交付使用。”

—— *Basic Systems Analysis*, *Alan Daniels and Don Yeates*

问题是，分析员如何保证能够准确理解用户的需求？下面的技术可用于发现用户的需求。

1. 面谈

面谈非常有利于分析员尽快了解真实情况。在分析阶段，目前的或计划的过程中涉及到的每一个人都应被问及。这样能够保证过程的每一个细微之处均可被了解。过程的实际运行方式很可能与人们想像的运行方式不太相同。例如，当你问一位经理时，他可能告诉你商业过程以一种特定的方式运行，但当你问具体做这项工作的人员时，他可能告诉你还有另外一个步骤。这个步骤可能算不了什么——他可能只是复制了一份文档，以便在文件柜中妥善保存，或是复制一份文档，送到其他部门存档。不管这个步骤是什么，对经理而言，它都是透明的，所以他不能告诉你这方面的信息，因为他不是这方面信息的最具发言权的人。

面谈的另一个重要作用是迅速核查事实。用一个简短的会谈澄清事实总比细查相关的文档简便。

2. 使用问卷调查

若要大范围多来源地收集信息，使用问卷调查就是一种很实用的方式，这可以避免收集信息时制定与每个人的面谈计划。在进行计划中的面谈前发放问卷调查也是有好处的，因为它们允许被访问者安排收集数据的时间，从而节约面谈的时间。

虽然问卷调查有利于以最小的实际代价、从多个来源获取大量信息，但对问卷调查中的问题必须注意，它们必须经过仔细的设计，以获得准确而中肯的信息，因为从问卷中很容易得出歧义的信息，如下面的问题：

问题：完成过程 A 需要多少时间？

回答：几个小时，但周四用的时间较长。

这样的答案可能导致这样的结论：周四的工作负荷比较重。但考虑到其他指标不支持这样的假说，这个结论看起来有点奇怪。进一步调查后，发现多花费的时间与工作量没有关系：周四，一个重要的职员迟到了。他的迟到不可避免地歪曲了问题的原始答案。

3. 观察

除了面谈和问卷调查，还有另外一个很好的信息来源，那就是现场观察已在运行的系统。操作员可能对自己的工作习以为常了，操作过程几乎已成为本能了，当被问及系统的运行过程时，他们可能会忘记提及一些关键步骤。

4. 文件记录和表示法

虽然用以上几个步骤获取信息是非常重要的，但如果获得的信息不能被准确记录，那还是毫无意义，所以记录信息这个阶段才是最重要的。简单地记录信息是不够的，记录信息必须达到这样的程度：保持最大的信息量和获取信息时的具体环境。下面是先前周四有职员迟到的例子，其中包括具体情形和扩展信息：

过程 A 通常需要三个小时，但是周四的汇报晚两个小时。具体的情况是：由于等待一位每周四汇报工作的职员，这个过程被延误了。如果这个过程很重要，可以考虑重新安排任务。

记录下来的信息是调研过程输出的惟一实质性的东西，它们之所以重要是基于以下几个方面：

- **分析**：格式化的文档和注释能够强调系统的重要特征。
- **沟通**：标准化的文档和注释有利于项目组以明确的方式相互沟通。
- **清晰**：提供技术词典能够保证术语的含义始终如一，不易被误解。
- **培训**：使用格式化的文档和注释可使新来的人员在较短的时间内发挥作用。
- **管理**：在系统能够实际运行之前，形成的文档为项目组的所有成员提供一种协调一致的手段。
- **安全**：好的文档就像设计图纸。如果系统遭到破坏甚至被摧毁，设计图纸能使修复和重建尽快地进行，完成后的系统很有可能像新建的一样。同样，计算机系统的准确的文档对于快速有效地解决问题非常有价值。本书中，关于面向对象分析使用的文档将在本章后几节讨论。使用的表示法主要是 UML v1.4，还有其他一些表示法。

2.2 当一个对象不成为对象时

面向对象分析的一个最困难的任务是史别出什么是类、食么不是类。在第 1 章中，有一个工资单应用程序的例子，其中的“money”（工资数额）被识别为一个对象。在这种情况下，一

个数值就可以方便地表示“money”。但是在另一种情况下，“money”可能不只是一个数量，还需要确定其货币类型，因而成为一个类。需要一个特定的类吗？或者说，一个数字是否就足以表示它？要回答这个问题，必须明白：重要的是，对你和你试图解决的问题来说，对象是什么。

与识别有关的另一个问题是识别和定义问题域。问题域是被研究的问题的范围。整个业务或系统中有哪些需要被考虑并建立起问题的模型？当识别对象时，很容易迷失在问题域的外面、不适当地将分析复杂化。最好还是用一些实例来说明问题域，这些例子中也包含一些技巧，可以在识别对象时使用。这里将给出两个例子。

- 第一个例子关注在公共汽车站等候公交车的人与一站接一站驾驶公交车的人的不同需求。
- 第二个例子表明不同的人对什么是“桌子”这个问题的三种不同理解。

2.2.1　公共汽车站问题域的实例

公共汽车站和公共汽车线路是如何相关联的？本例将给出三种不同的观点。

1. 公共汽车线路的模型

图2-1显示了一条公共汽车线路（BusRoute）的模型。每一条线路由一组公共汽车站（BusStop）组成。这组车站并不是一个散乱的集合体，因为设置每一条新线路都要确定车站的排列次序。

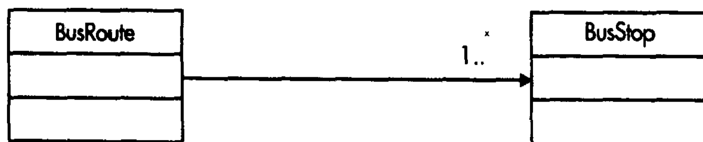


图2-1　公共汽车线路的模型

如果你是一个打算在车站候车的乘客，这个模型几乎没有什么用。除了知道你所处的这个车站应该至少属于一条线路外，并没有其他指示，如车站属于哪些线路，什么时候会有车，等等。这是因为**:BusRoute**与**:BusStop**之间的关系是单向的，**:BusRoute**知道关于**:BusStop**的信息，而**:BusStop**不知道**:BusRoute**的信息。

2. 公共汽车站的模型

对一位站在车站候车的乘客来说，图2-2的解决方案是一个更好的模型，这里有经过这个车站的公共汽车线路的有关信息。但是，**:BusRoute**与**:BusStop**之间的关系现在反过来了，**:BusStop**知道关于**:BusRoute**的信息，而**:BusRoute**则不知道**:BusStop**的信息。结果是，没有需要的线路信息——线路从哪儿开始、又在哪儿结束，或者，整个旅程需要多长时间、正点到达下一站的时间。

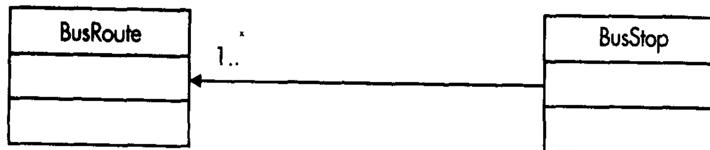


图2-2　公共汽车站的模型

接下来的问题是，公共汽车可能永远也不会到达一个**:BusStop**。原因是：尽管乘客知道他们是在一个**:BusRoute**（因为每个**:BusStop**都知道**:BusRoute**的信息），但一个**:BusRoute**上的公共汽车司机并不知道属于**:BusRoute**的一部分的**:BusStop**。没有这些信息，公共汽车司机不知道按怎样的路线行车，甚至根本没有驶离车站。

3. 双方都顾及的模型

下面的解决方案对两方都有利（见图2-3)。类**TimeTable**访问所有的公共汽车线路和所有的公共汽车站，类**BusRoute**和类**BusStop**也都能访问类**TimeTable**。

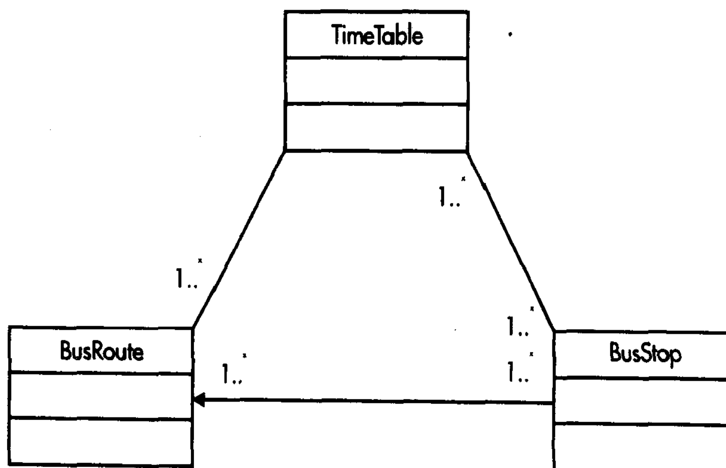


图2-3 线路和站均顾及的模型

一个**:TimeTable**可能是每个类中的一个真实的实例（如每个公共汽车站的一本实实在在的时刻表手册），也可能是对某个单个实例的访问，如一个中央数据库。每个**:BusRoute**可以访问**:TimeTable**，使用**:TimeTable**能够确定途经的**:BusStop**。每个**:BusStop**可以访问**:TimeTable**，使用**:TimeTable**能够确定**:BusRoute**和公共汽车的时刻表。

2.2.2 桌子问题域的实例

你可能会想，一个桌子应该不会给任何人带来什么问题吧？不管怎样，一个桌子不过是若干部件的组合。可是，不同的人会以不同的视角看待桌子，如同下面的例子。

1. 木工的看法

木工要花很长的时间来加工桌子的部件。他可能加工了一些部件又扔掉了一些有瑕疵的部件。最终，他总算做好了所有部件，并将它们安装在一起，完成了整个产品。一旦安装完成，木工就把所有这些部件的组合整体描述为一张桌子。对于木工来说，在所有的部件被安装完成之前，桌子什么也不是，重要的是桌子的部件，如桌面和桌腿，见图2-4。

图中的“空心菱形”在表示法中用来指示整体和部件之间的松散结合关系。如果桌子被解体，留下的部件还可以被木工重新使用。

2. 搬运公司对于桌子的看法

搬运公司的一个代表来到家具公司安排桌子的运输。就桌子的做工手艺略加评论之后，他

就转入对他有实际利害关系的问题。对于搬运公司的代表来说，桌子是一个具有长、宽、高和重量属性的对象（见图2-5）。桌子的审美价值不在他们的考虑之列，组成桌子的部件也同样不在考虑之列。

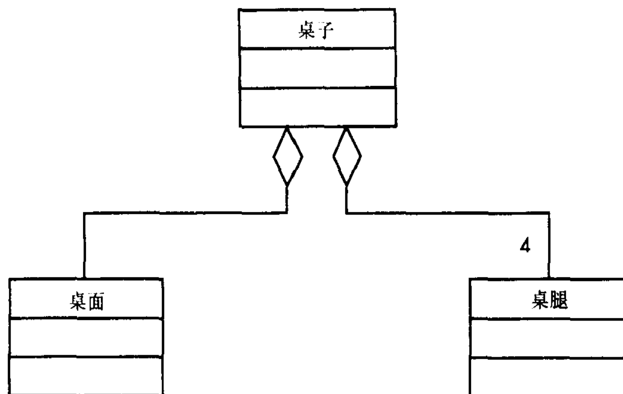


图2-4　木工对桌子的看法

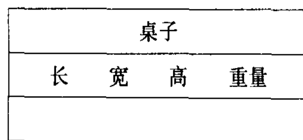


图2-5　搬运公司考虑的桌子属性

3. 桌子的主人对于桌子的看法

当桌子被运到并打开包装后，桌子的主人看到了桌子的真实面目。仔细检查过桌子并表示满意后，桌子的主人将桌子移动到理想的位置。虽然木工可能基于一个目的建造了桌子，但桌子的主人将做出最后的评判。对主人来说桌子的一切都是重要的：它的尺寸、座位空间以及外观等等。可是，与木工不同，木工可以替换或扔掉有瑕疵的部件，而桌子的主人要么接受桌子的任何瑕疵，要么更换整张桌子。所以对主人来说，桌子不只是所有部件的总和，并且如果它的任何部分被损坏，桌子也就被损坏了。

图2-6中“实心的菱形”在表示法中指示整体和部件之间的紧密联系。

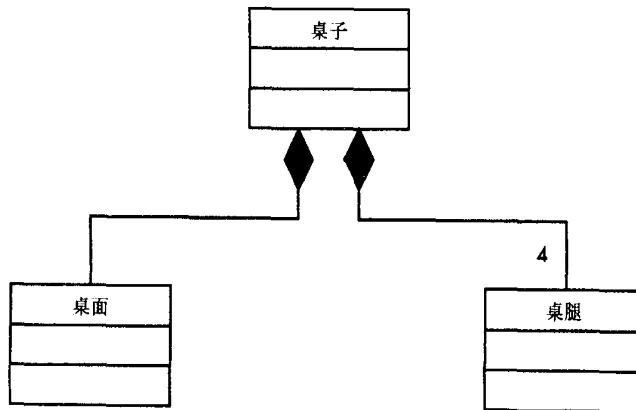


图2-6　主人对于桌子的看法

4. 桌子实例总结

注意，桌子的主人和木工使用的表示法是有差别的。主人看到桌子腿，但认为它们只能作为桌子的一部分而存在。桌子是由部件组成的，但一条桌腿本身并没有意义。木工则将桌腿看做独立存在的物体，并用“桌子”这个名称作为描述最后完成的集合体的简称。

2.2.3 问题域小结

在以上例子中，重要的是理解问题域。项目要为哪些人解决问题？他们是怎样理解系统中的对象的？如果有必要，描绘出对象的生存周期（如一张桌子从木工、搬运公司、分销商、零售商到购买者），系统设计者需要能够无缝地将对象从一种认识映射到另一种认识。第5章的“高级API方案”一节将描述一个可能的解决方法。

2.3 使用用例分析

用例（use case）用于在一个高级层次定义系统的功能，而不是描述参与系统活动的类和操作的细枝末节。用例允许用户描述系统的重要需求。

用例是客户和开发者之间沟通活动的不可分割的一部分，允许各方识别重要的外部实体（如用户和数据库），这些外部实体就是“参与者”。知道了这些参与者，你就可以讨论它们与系统之间的交互，包括可能的任何变化或异常。最后，用例让你创建一个共享的技术词典，并提供关于该问题域的一个有价值的统一认识。

一个用例可被其他用例引用，即可能存在嵌套的用例。

2.3.1 用例图

用例图说明一个系统中存在于参与者和用例之间的关系。用例说明系统的可见的外部表现。参与者在用例图中用于表示与系统交互的外部实体，如一个用户或一个数据库。

一个用例图可以使用以下的符号：

- 椭圆：代表用例。
- 条形：代表参与者。
- 用例四周的矩形：代表正在设计的系统。
- 参与者和用例之间的实线：代表关联。
- 用例之间有单向箭头的虚线：代表关联，如图2-7所示。include意指虚线起点处的用例要使用虚线箭头末端的用例去完成它的任务。extend意指箭头处的用例可被扩展，扩展的方法是：用包括在扩展用例中的动作去替换一个步骤。

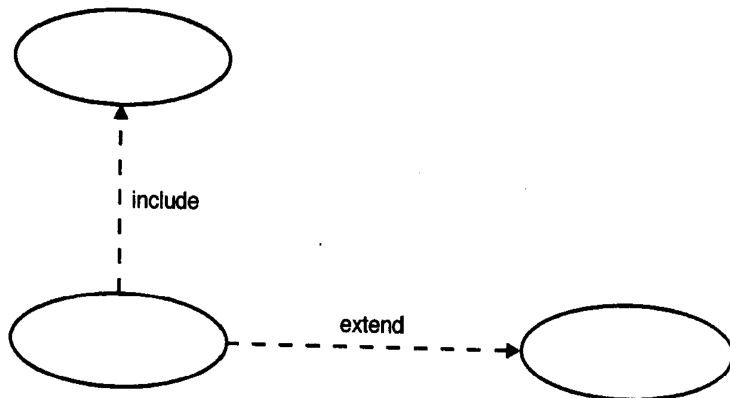


图2-7 用例之间的关联符号

2.3.2 一个简单用例的例子

图 2-8 中的简单例子表示工作人员是怎样为一名学生注册一门培训课程的。

虽然有这样一个图形表示法，但再保持用例的一个书面记录会更好，因为它允许保存其他附加信息，包括用例的完整描述。

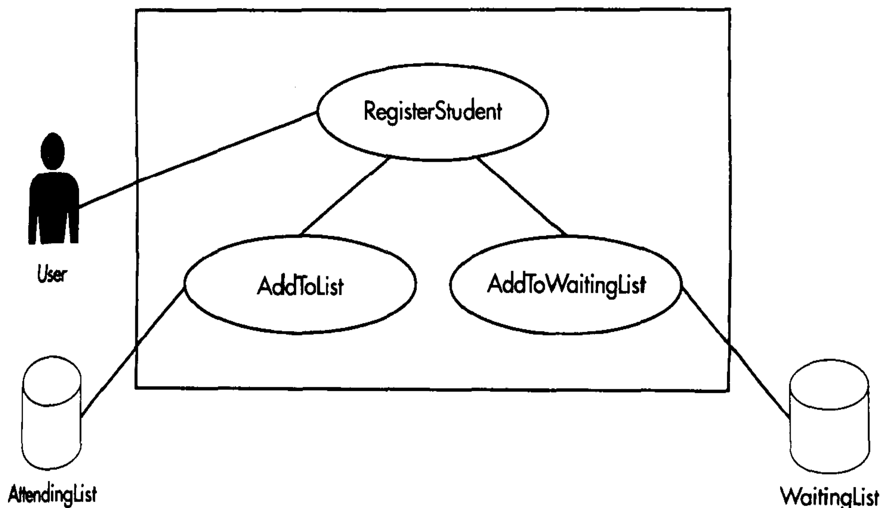


图 2-8 一个简单的用例例子

2.3.3 一个用例模板

以下是用例模板的每一部分的描述。每个用例都被给定一个惟一的编号和参考名称（或简称），如“UC #1 registerStudent”。

用例编号	< 简称 >
描述	该用例的详尽的解释

以下四个项目从外部交互的角度描述用例是怎样开始和完成的：

前提	系统需要在什么条件下，用例才可以运行
触发	什么条件可以使用例开始
成功	当用例完成时，系统应处于什么状态
中止	如果用例被中止时，会有什么情况发生

参与者这部分描述用例涉及的主要外部实体。一个参与者可能是系统的一个用户或系统使用的一个数据库。这部分也描述可能涉及的其他外部实体：

参与者	主要的	谁担任主要角色?
	从属的	谁担任次要角色?

过程这部分描述一个正常的、成功运行的用例的每一个详细步骤——如，某工作人员成功

注册了一个学生的某个培训课程：

过程　　步骤（step #）　　简称<动作>　　描述

变更部分描述同样能到达用例成功结局的替代步骤——如，工作人员可能决定注册另一门课程，而不是原来指定的那门课程：

变更　　步骤#　　<动作>　　描述

异常（Exceptions）部分描述用例正常执行过程中可能发生的任何出错的事件——如，学生不能被系统识别，必须被登记：

异常　　步骤#　　<动作>或用例　　描述

其他有用的信息

以下列出的是能够为用例提供更全面的描述的补充信息。

- **优先级**：该用例的优先级是什么？
- **持续时间**：该用例预计需要多长时间？
- **频度**：该用例发生的频繁程度如何？
- **交互界面**：系统与参与者之间有三种可能的交互界面：
  - **交互式**：每个客户与系统都有对话。
  - **静态的**：该用例依赖于不变的信息，如培训课程的教学大纲和课程表。
  - **程序表**：该用例依赖于动态特性的信息，如电影院将要上映的影片列表。
- **不定的因素**：是否有其他事件影响用例，如，较多的学生意味着较多的课程。
- **交付日期**：系统必须在什么时候完成？
- **相关的用例**：是否有任何其他用例与当前的用例相关？如，可能会有一个确认课程的用例。

2.3.4 一个用例实例

尽管一个用例模板有利于用例的规范化、模型化，并抓住问题本质，但它可能太累赘，并有歧义，其原因之一是没有使用 Ivor Jacobson 为用例设计的技术。

以下是由图 2-8 例子衍生的一个用例实例，它太累赘并有歧义：

Use Case #1	课程人数已满
描述	Smith 先生试图为 Eric 预订一门培训课程，但发现该课程已被订满了

前提	该课程必须存在。Eric 必须注册，以便被加入到课程中
触发	Smith 先生向 Tim 打电话，要求接受 Eric 为课程的学员
成功	Smith 先生应被告知 Eric 正在学习这门课程，或被告知 Eric 在等待名单中的位置
中止	对听课学生名单 AttendingList 或等待学生名单 WaitingList 所做的任何添加都应被删除

参与者	主要的	Mr.Smith，Tim，AttendingList
	从属的	WaitingList

过程	步骤（Step #）	简称<动作>	描 述
	1	ClientContact	Smith 先生打电话给 Tim
	2	QueryStudent	Eric 是注册的学生吗
	3	RegisterStudent	给 Eric 登记这门课程
	4	CheckTitle	确认登记的课程无误
	5	CheckDateTime	确认登记的课程的日期和时间
	6	CheckLocation	确认登记的课程的地点
	7	CheckCapacity	检查 AttendingList 中的空余名额
	8	PromptUser	AttendingList 名额已满，询问 Smith 先生是否将 Eric 加入 WaitingList
	9	AddToList	将 Eric 加入 WaitingList

2.3.5 写好用例的七个要点

上一节的用例实例不是一个好例子，它既不太通用也不太特定。下面是整理用例的七个要点，它们基于 Mark Ratjens 的一本著作（发表在 http://www.class.com.au/newsletr/ 97sep/usecases.htm）。

步骤 1：给用例一个恰当的名称

命名的指导性原则是：用例应该记载一个具有普遍意义的情况。上一节实例中的情况是一个过程，该过程描述一个委托人向一个系统用户打电话，系统用户按要求将一名学生加入到一门课程的学生名单中。因此，这个用例的命名可以是"Registering for a course"（"课程登记"）。

步骤 2：建立易于识别的风格

用参与者的任务来标识参与者，并在整个用例模型中使用一致的命名。尽量使用不同的字体或风格（如斜体）将参与者的名与用例的其他部分区别开来——如，"Smith 先生"变成"*client*"，"Tim"变成"*User*"，"Eric"变成"*student*"。

步骤 3：去掉注释性或次要的描述

要去掉参与者与系统交互之外的注释性事件的描述（如委托人询问是否 Eric 已经作为学生注册过）。正确的交互应是委托人询问学生是否注册过，交互应是特定的。

步骤 4：同时创建本质的和详尽的用例

至少应该存在一种版本的用例：特意剔除"参与者/系统"交互过程中的特殊成分的用例。这样的用例被称为"本质用例"（essential use case）。术语"本质用例"的发明人 Larry Constantine说："由于'essential use case'代表了抽象的、简化了的'用户/系统'交互活动的结构，它有利于更好地捕捉用户的基本意图和用户交互的目的。"从用例中去掉非关键的细节，以便关注于关键的需求——例如，用一个动作去替换每个课程细节的确认。

另一种版本的用例也可能会存在，它指定接口的特别之处，每个特别之处代表一种可能的选项。不过，使用有效的原型演示会更好。

步骤 5：建立域模型

常用的用例模型化实践通常会掩盖或忽略域模型的地位。Jacobson 认为，域模型的作用是

支持用例模型。域模型描述在应用程序环境中有直接对应者的对象，并且描述系统必须知道的对象。将域中的对象形成文档，作为需求的一部分，这有利于简化用例模型。如有可能，引用域对象定义就可简化用例。上一节的例子中，当委托人为多个学生登记同一门课程时，他的学费就会有个折扣。这个结论已经超出了这个用例的范围，应被推迟到域对象中讨论。

步骤 6：多种智能的步骤

采用一种系统地编号的方法来描述用例。用例模板中定义的基本过程应该用于描述被定义的过程的成功运行情况。随着交互活动进行，还应该有一些变通的情况——例如，委托人除通过电话外还可以通过传真和 e-mail 联系——这些变通的情况应在一个单独的部分中标注。还有，如果处理过程的某个环节可能出现异常情况，也应该在一个单独的部分中描述，如，课程的名额已满，异常处理就将学生放入等待列表中。

步骤 7：使用主动语态

为避免任何疑惑或误解的可能，在用例模型中，要尽可能使用主动语态。描述参与者和系统的动作要特别明确。

使用以上推荐的七个准则，图 2-8 例子的用例修改如下：

Use Case #1	课程登记
描述	系统用户为委托人的学生预订培训课程
前提	课程必须存在，将被加入的学生必须存在
触发	委托人打电话要求允许一名学生被现有的一门课程接受
成功	委托人应被告知是否该学生正上这门课，或该学生在等待列表中的位置
中止	对 AttendingList 或 WaitingList 所做的任何增补都应被清除

参与者	主要的	系统用户和 AttendingList
	从属的	WaitingList

过程	步骤（Step #）	简称<动作>	描 述
	1	ClientContact	委托人打电话
	2	RegisterStudent	给学生登记课程
	3	ConfirmCourse	与委托人确认课程细节：日期、时间、地点。根据操作的复杂程度，这可能变成它自己的用例
	4	CheckCapacity	检查 AttendingList 中是否有空余名额
	5	AddToList	将学生加入 AttendingList
	6	CheckDiscount	如果委托人为多名学生预订了一门课程，这些学生可能被考虑给予学费折扣

变更	步骤（Step #）	<动作>	描述
	1	ClientContact	传真
	1	ClientContact	E-mail

异常	步骤（Step #）	<动作>或 UseCase	描　述
	4a	ListIsFull	提示系统用户询问委托人，是否要将学生放入等待列表中
	4b	AddToList	仅当委托人同意，才将学生放入 WaitingList 中
	4c	"AnotherCourse"（这是另一个用例的名称。	与 AnotherCourse（另开一门课程）的接口，如果在等待列表中有足够的学生，将再开一门课程）

2.4 记录分析

分析的目的是收集系统的信息，并以一个有效的方式将它呈现出来。本节将关注进行分析时使用的文档，将涉及四个方面的内容，每个方面都有它自己的文档：

- ▶ 分析类的静态方面的特性
- ▶ 分析类的动态方面的特性
- ▶ 分析系统的静态方面的特性
- ▶ 分析系统的动态方面的特性

2.4.1 分析文档：类的静态特性

下面这部分讨论的文档将提供类的静态特性的信息和类定义的细节，还将讨论类之间的交互活动。涉及的文档如下：

- ▶ 描述类本身的类图。
- ▶ CRC（Class：Responsibilities：Collaborators）卡片：开始定义类的地位或作用。
- ▶ 脚本：描述类与类之间的交互活动。

1. 类图

类图全面描述一个类中存在的属性和方法，参见下面有关类 Loan 和类 CashAccount 两个表中的内容。由于类的描述是在分析过程中逐渐成熟的，所以这些类图文档将被不断地修改，直到分析完成，形成稳定的类描述。

类图中使用的表示法如下：

+	公共的属性/方法（public attribute/method）
#	保护型的属性/方法（protected attribute/method）
-	私有的属性/方法（private attribute/method）
/	派生的属性/方法（derived attribute/method）
$	基于类的属性/方法（class-based attribute/method）

以下两个类图摘自第 10 章。

类名	*Loan*	描　述
属性	*- duration*，*- repayment*	两个私有属性

（续）

类名	*Loan*	描 述
操作	+ *Loan*（*amount*：*double*，*duration*：*integer*）：*Loan*	一个与类名相同、返回该类的一个对象的公共方法被称为构造函数（constructor）。构造函数在需要创建新对象时被调用。大多数构造函数不需要参数，但本例中，如果不知道一笔贷款（Loan）的数量和期限，这笔贷款就没什么意义了
	+ *showRepayment*（*void*）：*double*	一个公共方法，它没有参数，但返回一个 double 型数据
	+ *adjustDuration*（*void*）：*void*	一个公共方法，它既没有参数，也不返回数值

类名	*CashAccount*	描述
属性	- *balance*	一个私有属性
操作	+ *CachAccount*（*void*）：*CashAccount*	一个无参数的公共型构造函数
	+ *credit*（*double*）：*void*	
	+ *balance*（*void*）：*double*	
	+ *debit*（*double*）：*void*	
	+ *setBalance*（*double*）：*void*	
	+ *adjustMonth*（*void*）：*void*	

2. CRC 卡片（非 UML）

在分析过程中，CRC（Class：Responsibilities：Collaborators）卡片与类图联合使用，用于记录和说明被识别的类。CRC 卡片是参照标准的名信片设计构思的，参见下表的显示。与“Class（类）”对应的是类的名称（即“Loan”）。在“Responsibilities（职责）”部分，记录类支持的任务或职责。“Collaborators（协作类）”部分包含使用该类的类或被该类使用的类。

下面就是两个 CRC 卡片的实例，它们摘自第 10 章。

Class:	Loan
Responsibilities:	管理一笔贷款的细节
Collaborators:	Display，CompanyDetails，CashAccount

Class:	CashAccount
Responsibilities:	管理公司的现金
Collaborators:	Loan，CompanyDetails，Machine，Display

一张 CRC 卡片应该能够显示一个类的所有信息。如果一张 CRC 卡片不足以显示完所有信息，通常是因为类太复杂了，应该拆分为若干简单一些的类。

使用 CRC 卡片的一个有用的附带效应是解决分析阶段的僵局。一个小组中，每人用一个 CRC 卡片记录系统中每个类的职责，并解决类之间发生的交互活动。在分析过程的这个环节

上，CRC卡片被用作助记工具，即，CRC卡片就是用于记录类的存在，并没有更多的意义。

3. **脚本（非UML）**

脚本用于表示类之间的所有可能的交互活动，以便为其他文档作准备。脚本被分解为若干表格书写，每个表格可反映两个类之间发生的交互活动。表2-1说明了**:User**是如何与应用程序中的其他类交互的。

表2-1 :User的交互活动

调用者	方 法	被调用者	结 果	Use Case #
:User	*requestLoan ( )*	**:Display**	选择贷款选项	1
:User	*PurchaseMachine ( )*	**:Display**	选择新机器选项	2
:User	*nextTurn ( )*	**:Display**	选择下一轮	3
:User	*displayDetails ( )*	**:Display**	选择显示细节选项	5

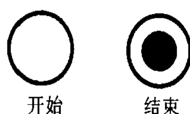
2.4.2 分析文档：类的动态特性

用来表达类的动态特性的文档就是状态图。类的动态特性表明一个类的对象在其生存周期中怎样回应来自各方面的刺激。

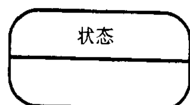
1. **状态图**

状态图用于表明一个对象怎样从内部对不同的方法作出反应。状态图使用的表示法如下。

状态图的开始和结束：



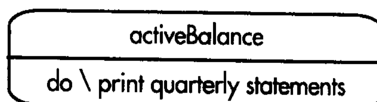
状态符号 下面是一个简单的状态符号：



2. **带动作的状态符号**

对象处于该状态时，执行的内部动作或活动的列表

下面是一个状态符号的例子：



简单的状态转换 一个简单的状态转换表示两个状态之间的关系。要使转换出现，标明转换的事件必须已经出现。转换事件的表示法为：

<事件名称>（参数列表）[保证条件] /动作

如，debit（amount：double）[balance < -10000] /finish，表示如下：

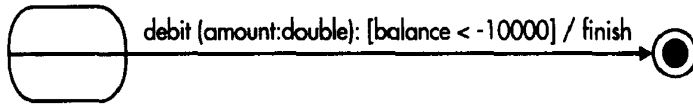
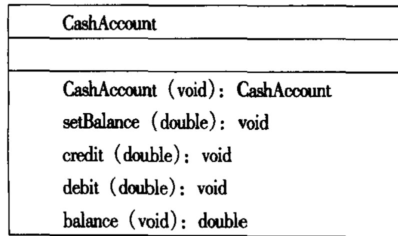


图 2-9 所示的是摘自第 10 章的一个状态图的例子。

类 **CashAccount** 定义五个基本操作：



一个 **:CashAccount** 对象从余额为 0 开始。图 2-9 的状态图说明了任何 **:CashAccount** 对象在其生存周期中可预期的状态变化。

其他实用的表示法允许采用分支或执行线程，如图 2-10 所示。

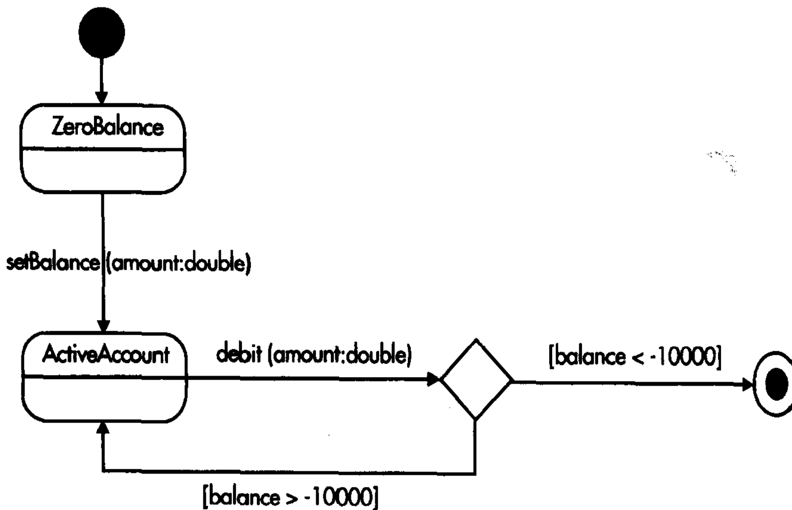


图 2-9 一个 **:CashAccount** 对象的状态图

2.4.3 分析文档：系统的静态特性

本小节中论及的文档提供关于系统的信息，表明类之间的相互作用。类关系图用于表达类之间的关系。协作图用于表达系统中的对象之间的相互作用。

1. 类关系图

发现了系统中的类并用 CRC 卡片和类图形成类的文档后，下一步就是将类之间存在的多个关系形成文档。使用的类关系图可表达如下几个方面的关系：

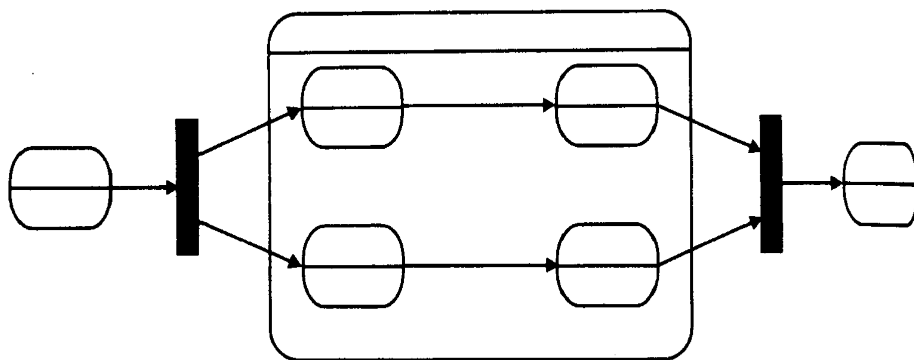


图 2-10　采用分支或执行线程的表示法

- 简单关联：表达典型的关联。
- 有向关联：表达单向的关系。
- 关联类：表达从一个关联中派生的类。
- 多维关联：表达两个以上的类之间的关联。
- 集合关联：表达一个类是其他类的集合。
- 组合关联：表达一个类是由其他类组成的。
- 继承：表达由其他类继承或从其他类派生的类。

所有的关系图都使用一致的表示法；继承关系图表示法将在本章最后一部分“继承”一节中讨论。关系表示法解释如下：

- 限定项放在关联线的各端点处，用于指示该关联中每个类的角色和作用。每个限定项表示为一个小菱形，放在类和关联线的端之间。如，一个公司可能有许多员工，如果限定项是员工数，则该关系正好变为一个。
- 多样性用于表示关系的复杂性，如 1-to-0..1、1-to-1、1-to-many 或其他的任意组合。
- 关联名称是赋予关联的一个名字，这样可以通过名称来引用该关联。
- 导引箭头可用于指示关联的方向。默认时，关联线两端都没有箭头。

简单关联　关系图的首要任务是用于记载类是怎样互相连接、允许它们互相交互并在系统中履行各自的职责的。对于本章前面一部分描述的公交线路的例子，图 2-11 是它的一个全面记载的关系图。

有向关联　这是对象之间的单向关系。只有有向关联的源（即客户端）知道目标（即服务端）的信息。关系的目标能接收消息，但不知其来源。在图 2-12 的例子中，一个**:Polygon**对象知道**:Point**对象的消息，但反过来就不行了。

关联类　这是一种既有关联特性又有类特性的模型。关联类可被看做一个具有类特性的关联，也可以看做一个具有关联特性的类。参见例图 2-13。

多维关联　这是一种三个或更多类之间存在的关联。从各个类的角度看，关联的每个实例是源自各个类的一个 n 维数值。参见例图 2-14。

集合关联　它是关联的一种特殊形式，表明一个类包含对其他类的引用。这种关联被称为集合（整体）和组件（局部）之间的整体/局部关系。这种形式的关系允许一个类与其他类共

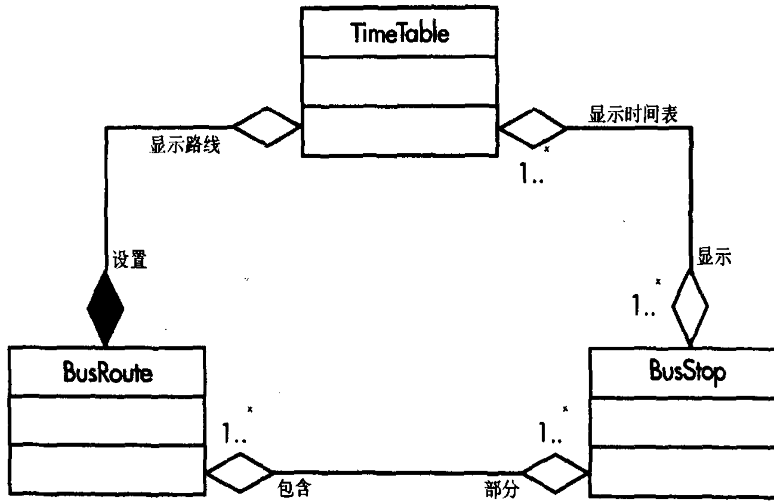


图 2-11 公交线路模型的类关系图

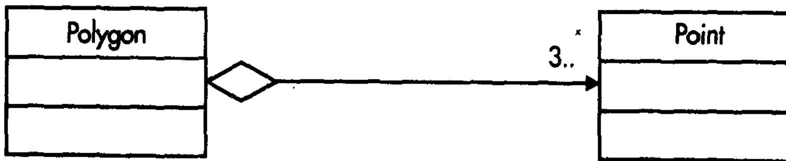


图 2-12 有向关联的例子

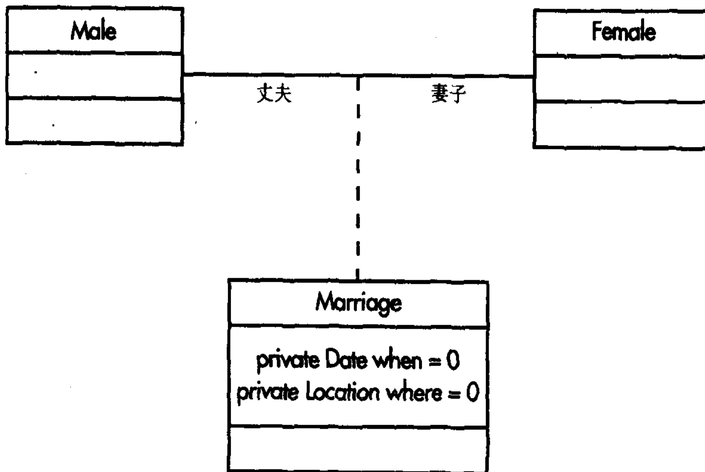


图 2-13 关联类

享其组件。本章前面部分“木工对于桌子的看法”一节中的问题域例子就是属于集合关联，其关系图如图 2-15 所示。

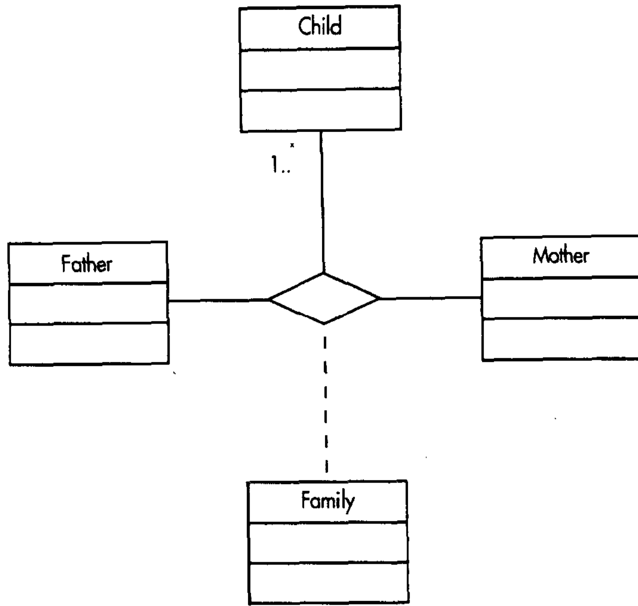


图 2-14　多维关联的例子

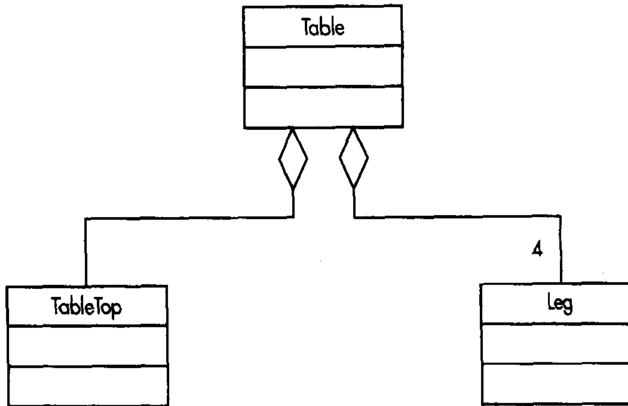
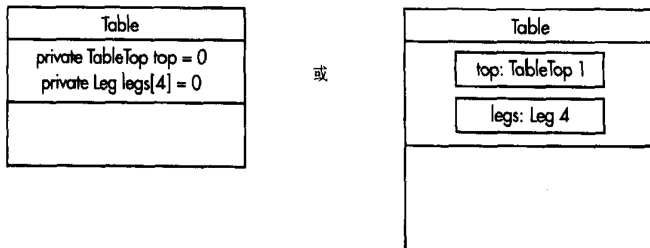


图 2-15　集合关联的例子

组合关联　组合关联是一种强调所有权的集合关联，在这种关联中，整体的生成周期决定组件的生存周期。“桌子的主人对于桌子的看法”一节中的问题域例子也属于组合关联，其关系图如图 2-16 所示。

在 UML v1.4 中，对于上述的关系图，还有另外两种 UML 表示法。第一种是类图，第二种是图形化类图。



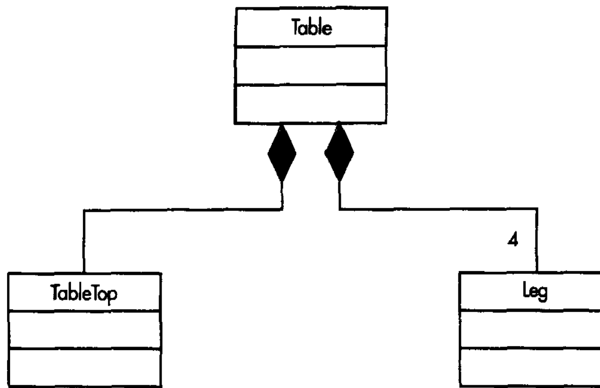


图 2-16 组合关联的例子

继承 正如第 1 章所说，继承有两种形式。第一种是派生继承，关注从单个父类中派生出子类，第二种是抽象继承，关注收集共同的属性和方法创建父类。图 2-17 再次显示其表示法。

在 UML 中也存在一种替代的表示法，如图 2-18 所示。

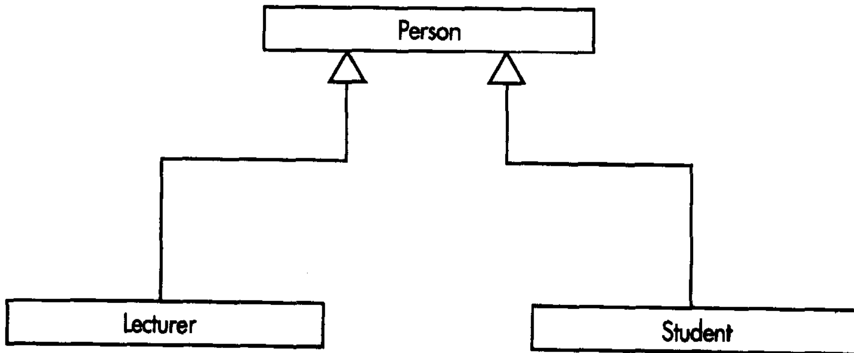


图 2-17 抽象继承的表示法

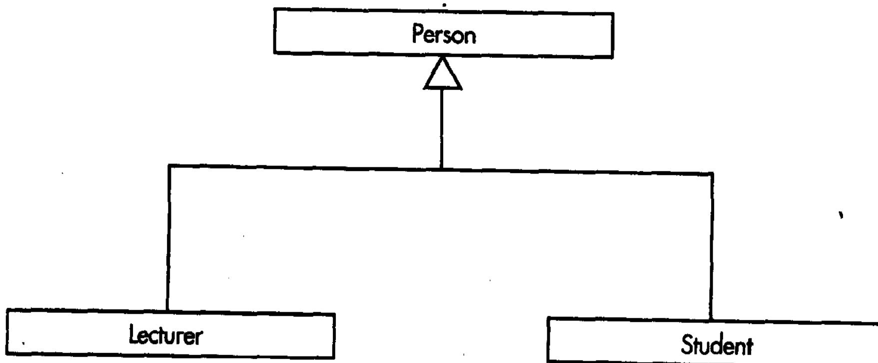


图 2-18 UML 中抽象继承的表示法

2. 协作图

协作图用于在给定的情形下，表示对象以及对象之间的关系。例如，对每一个用例图，将有一个协作图。协作图可以存在于两个层次。

第一个层次称为实例层次。这个层次的协作图表达对象、对象之间的联系，以及用于执行

所要求的协作的方法。图 2-19 就是一个例子。

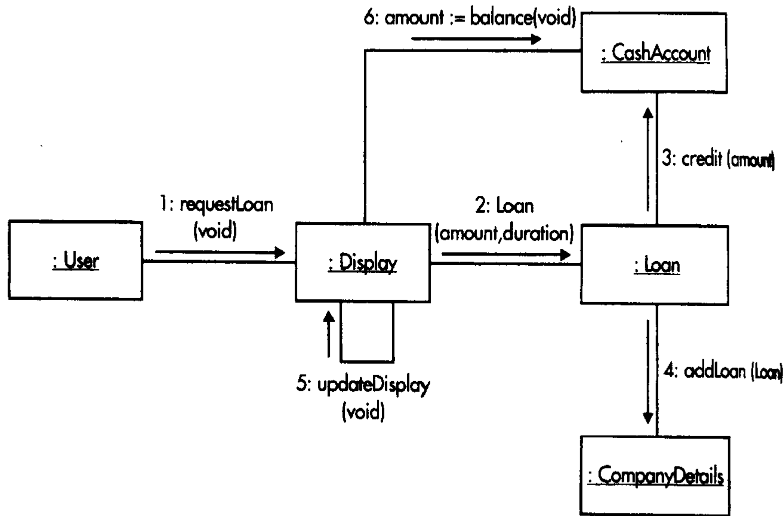


图 2-19 一个实例层次协作图的例子

实例层次协作图的特征列举如下：

▶ 对象被表示成条形或矩形。

▶ 对象一般以如下的格式标识：

<instance name> : <class name>，即：<实例名> : <类名>。

▶ 允许出现一个 "/role"，它只用来指定类。

▶ 方法被标识为：1：<method name>（argument），即：1：<方法名>（参数）。

▶ 方法的重复执行被标识为：1.1 * [i=1..n]：<method>（i）。

第二个层次被称为详述层次。这个层次的协作图表达对象担当的角色以及对象之间存在的关联。图 2-20 就是一个例子。

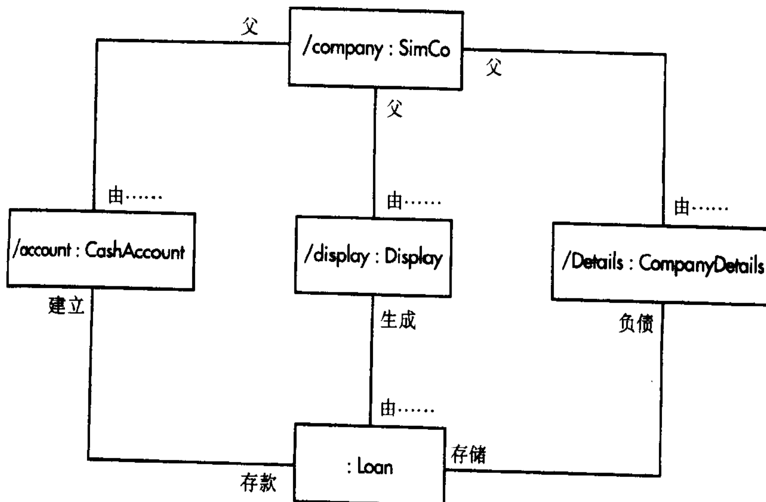


图 2-20 一个详述层次协作图的例子

详述层次协作图的表示法如下：

- 每条线都是一个关联，应被标识为关联。
- 每一个矩形包含一个特定的类担当的角色，以“/role：class”的形式标识。

2.4.4 分析文档：系统的动态特性

本节中讨论的文档有活动图和序列脚本与序列图（sequence script and diagram）。

1. 活动图

与描述对象的内部状态的状态图相对应，活动图用于描述系统的状态。一个活动图，如图2-21所示，表达完成一个任务时涉及的步骤。

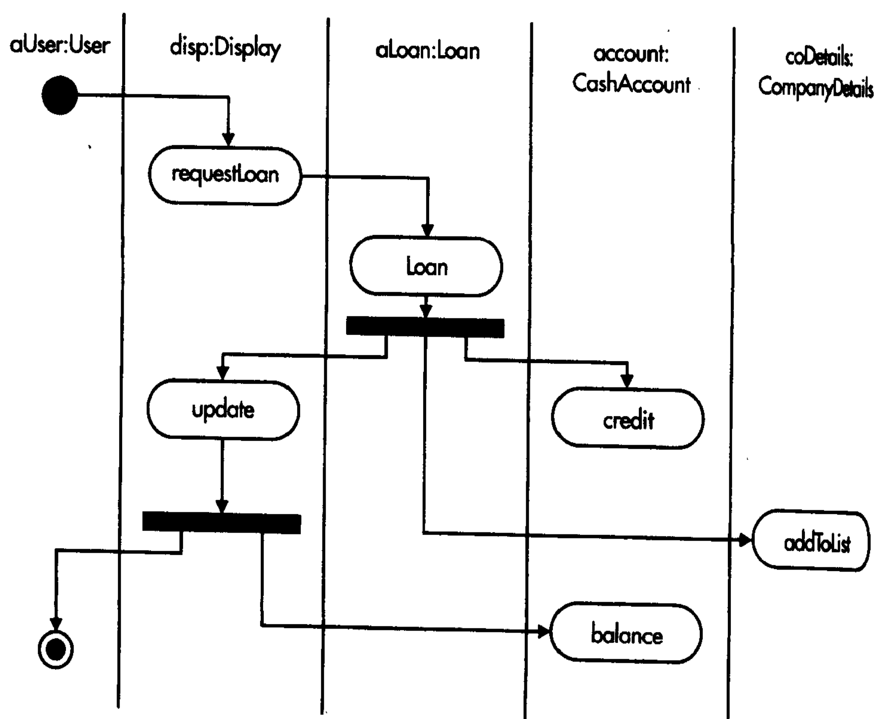


图2-21 一个活动图的例子

活动图中使用的表示法说明如下：

- 活动图被分为若干逻辑栏或“泳道”。每个泳道指示执行所分配的动作需要的一些职责。动作可能被一个对象执行，也可能被一组协同工作的对象共同执行。
- 实心圆表示整个活动的开始。
- 一个动作态代表一个不可被中断的可执行的运算。它在活动图中被表示为一个包含动作名称的胶囊状“菱形”。
- 转换代表从一个动作态到另一个动作态的变化。在活动图中，转换被表示为一条有单向箭头的线，箭头指向给定对象的新的动作态。
- 分支表示由一条路径变为两条或更多条新路径的分岔点。分支由一个菱形表示，每个

输出分支（即新路径）上都有一个用方括号“[ ]”括起的保证条件，表示引导应用程序按该分支运行的判据。

▶ 内有一个小实心圆的圆圈表示整个活动的结束。

一个附加的表示法解释如下：

▶ 粗黑同步条表示控制流是怎样分岔和合并的，如图2-22所示。

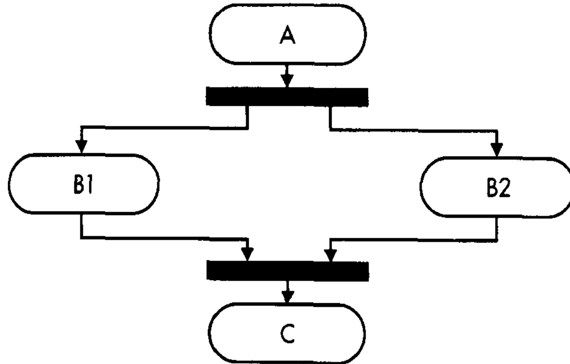


图2-22　控制流的分岔与合并

▶ 信号可用于允许资源共享。如果一个资源正被使用，一个过程可能等待，直至收到资源被释放的信号，如图2-23所示。信号发出端的符号是由一个凸五边形表示，它看起来像在一个矩形的一边贴上一个三角形。信号接收端的符号是由一个凹五边形表示，它看起来像在一个矩形的一边砍出一个三角形缺口。

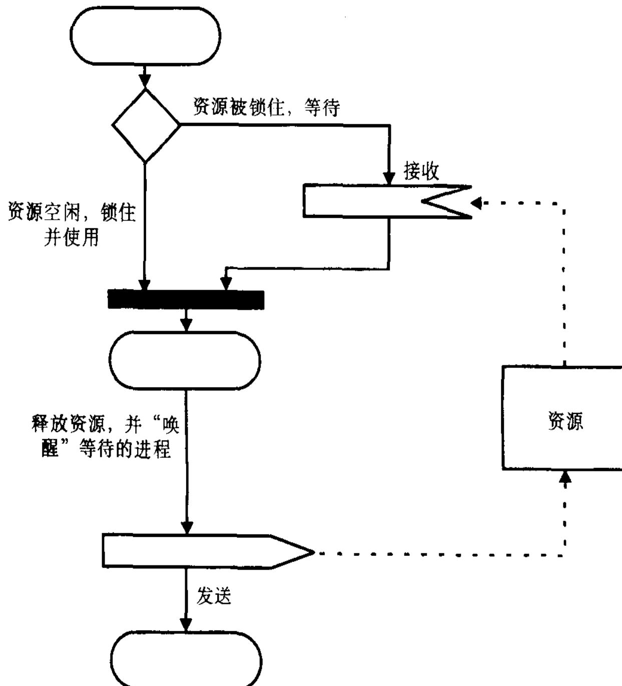


图2-23　用于允许资源共享的信号

2. 序列脚本和序列图

序列脚本和序列图用于表示按出现顺序排列的对象交互活动。特别是，它们表明了参与交互活动的对象和对象之间交换消息的顺序。

序列脚本表示法 序列脚本是序列图的替代方法。记录系统的静态特性时，脚本被形成，以保证类之间的每一个交互活动均被记录。序列脚本拓展了那些表示法，用于以表格的形式表示序列图。

静态脚本表示如下：

源	方法	目的

动态序列脚本表示如下：

Ref #	源	方法	目的	Next Ref #

“Ref #”和“Next Ref #”域显示系统的流。动作的序列由尾字符不同的序号表示，如“1”到“1.a”到“1.b”到“1.c”，用表格的形式则显示如下：

Ref #	源	方法	目的	Next Ref #
1				1.a
1.a				1.b
1.b				1.c

要表示过程的内部行为，可将其放入花括号对“{}”，如{return void}或{performAction}：

Ref #	源	方法	目的	Next Ref #
1	User	Save	Display	1.a
1.a	Display	saveData (data: datatype)	Database	1.b
1.b	Database	{doSomething}	Database	1.c
1.c	Database	{return}	Display	1.d
1.d	Display	{end}	—	—

系统的运行一般不会像“由步骤a到步骤b到步骤c”这样简单，所以在这些情况下，需要使用一些其他附加的表示法。

其中第一个表示法用于表示由用户引导（user-directed）作出的抉择。用户被提示提供一些输入：

2.a	Display	“enter amount”	User	2.b

每一个可能的输入共享同一个“Ref #”，在上例中，“Ref #”均为“2.b”。但是一个输入会引起不同的后果，因此，需要修改索引值（即“Next Ref #”），以反映这样的事实：

2.b	<amount < = 0>	2.1
2.b	<amount > 0>	2.2

第二个附加的表示法用于表达由应用程序引导（application-directed）进行的选择——如，调用一个方法，从数据库中返回一个指定的对象：

3.a	User	Find (void)	Database	3.b
3.b		[! found]		3.1
3.b		[found]		3.2

总之，使用的表示法如下：

{}	用于描述进程在当时的内部行为
“”	提示用户做出选择，如“接受”或“拒绝”
< >	表示用户引导的分支（即由对提示的不同应答引起的分支）
[]	表示应用程序引导的分支（即由某操作的返回结果引起的分支）

下面的序列脚本实例来自第10章，相应的序列图见图2-24。

Ref #	源	方法	目的	Next Ref #
1	User	RequestLoan	Display	1.a
1.a	Display	“Enter amount”	User	1.b
1.b		<amount <= 0>		1.1
1.b		<amount > 0>		1.2
1.1	Display	{end}		
1.2	Display	“Enter duration of loan” User		1.2.a
1.2.a		<duration <= 0>		1.2.1
1.2.a		<duration > 0>		1.2.2
1.2.1	Display	{end}		
1.2.2	Display	“Confirm Loan”	User	1.2.2.a
1.2.2.a		<Reject>		1.2.2.1
1.2.2.a		<Accept>		1.2.2.2
1.2.2.1	Display	{end}		
1.2.2.2	Display	Loan (amount, duration) Loan		1.2.2.2.a
1.2.2.2.a	Loan	Credit (amount)	CashAccount	1.2.2.2.b

（续）

Ref #	源	方法	目的	Next Ref #
1.2.2.2.b	CashAccount	{return}	Loan	1.2.2.2.c
1.2.2.2.c	Loan	AddLoan (self)	Company Details	1.2.2.2.d
1.2.2.2.d	Company Details	{return}	Loan	1.2.2.2.e
1.2.2.2.e	Loan	{return}	Display	1.2.2.2.f
1.2.2.2.f	Display	UpdateCash (void)	Display	1.2.2.2.g
1.2.2.2.g	Display	balance (void)	CashAccount	1.2.2.2.h
1.2.2.2.h	CashAccount	{return}	Display	1.2.2.2.i
1.2.2.2.i	Display	{end}		

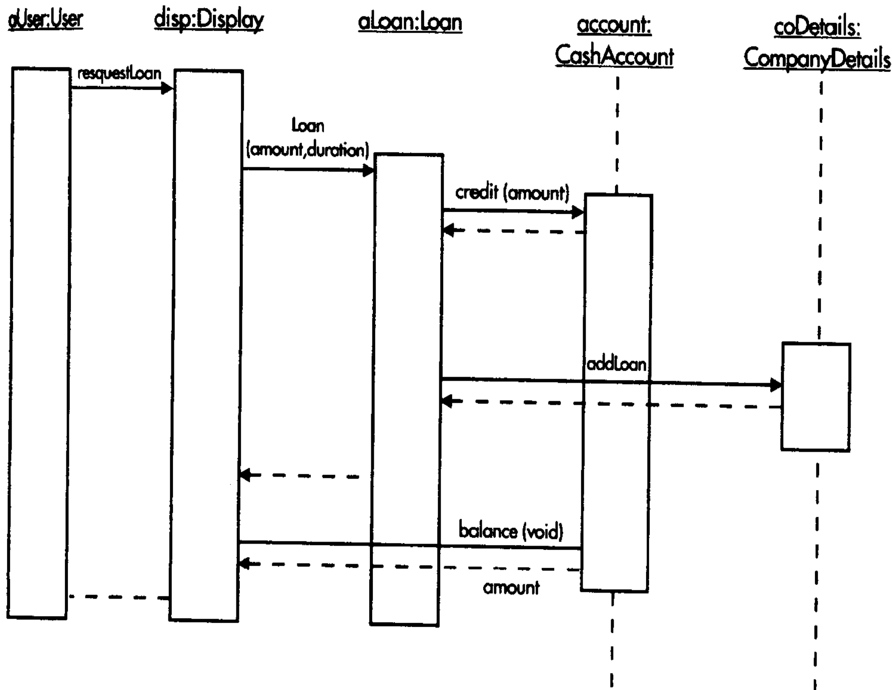


图 2-24　SimCo 例子的序列图

序列图　序列图使用的表示法由四个主要元素组成：

- 给定的交互活动中涉及的对象在序列图的顶部开始横向排列。
- 每个对象都有相关的“生命线”。如果对象在图中表示的交互活动之前就已经存在，那么它的生命线是一条竖直的虚线。一旦该对象成为交互活动的一个活动的部分，它的生命线转为一个细而高的矩形。
- 如果对象是在交互活动中创建的，它的生命线始于创建之处——即，它不是从序列图的顶部开始的。类似地，如果一个对象是在交互活动中被清除的，在清除之处，对象的细而高的生命线上有一个大的“X”穿过。

- 对象对（pair of objects）之间传递的消息被连接到相应的活动矩形的边界。这些消息是按照发出的时间顺序自上而下排放，第一个消息位于序列图的顶部，最后一个消息位于序列图的底部。发出的消息被画为实线，返回的相应消息被画为虚线。

其他附加的表示法还有：

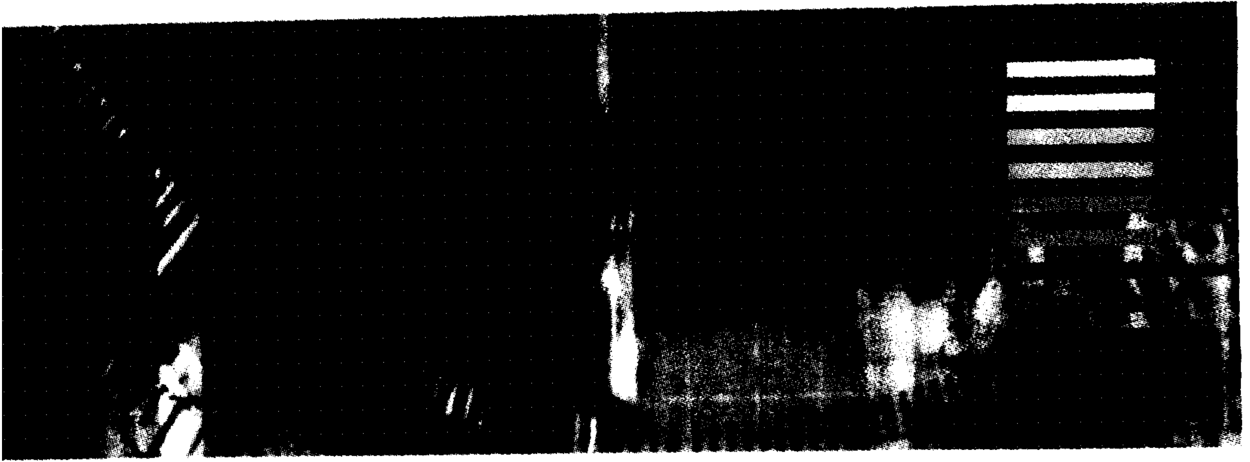
- 自引用 用一个附带的小矩形表示，上有曲线箭头离开和折回到同一个生命线。
- 分支 用若干发自生命线同一点的线表示，每条线上有显示在方括号中的保证条件，它们决定将选择哪条路径。

图 2-24 的序列图表示了一个用户通过**disp:Display**对象申请贷款的情形。用户和**disp:Display**对象之间进行对话（未画出），确认贷款的数额和期限。当细节被确认之后，一个**:Loan**对象被创建。这个**:Loan**对象被记入**account:CashAccount**对象，并被加入由**coDetails:CompanyDetails**对象控制的贷款列表。随后，控制权返回到**disp:Display**对象，该对象向**account:CashAccount**对象请求更新账户余额，然后控制权返回到用户，以便进行下一次循环。

2.5 小结

本章综合介绍了 UML v1.4 表示法，这些表示法是进行系统分析时需要用到的。本章虽没有对各种表示法的每一个方面都加以详述，但介绍的内容已经足以很好地使用这些表示法了。

用这些表示法的知识武装之后，我们就可以在下面几章探索各种设计准则了。



第三部分 设 计

目标：

- ▶ 学习良好设计的基本层面
- ▶ 理解度量设计为什么重要
- ▶ 学习如何度量设计
- ▶ 关注经典的不好的设计并给出替代方案
- ▶ 引入一些高级的设计结构

第3章 设计方案

本章将讨论以下内容：

- ▶学习抽象和接口设计方法
- ▶学习模板和继承之间的区别
- ▶学习设计的原则
- ▶理解好的设计的度量标准
- ▶理解设计手段对设计本身的影响
- ▶理解不同类型的复制构造方法

软件设计者的任务就是创建一个功能可行且易于维护的设计。一旦有了这样一个设计，实现小组（编码小组）必须从最初的开发阶段到最后的应用程序自始至终地贯彻该设计，不得进行任何修改。

如果在能否更改最初的设计这一点上开始妥协——无论是打算修改问题还是只增强产品性能——每个更改都像一个“毒瘤”。对多数“毒瘤”来说，若能够尽快发现，“病人”，也就是设计，可以被拯救。外科手术不是必须的，只需要求每一个更改都必须维护原始设计的纯粹性。

如果设计中的“毒瘤”不能被检查管理，设计也就逐渐变得不可行了。渐渐地，即便是做一个最小的改变也颇费周折，最后每一个人都认为必须推倒重来。理想的解决方法是从一开始就避免“毒瘤”的产生。

3.1 抽象类

抽象类是一种设计结构，它允许设计者创建一个永远不会有实例对象的类。正常情况下，每一个类都会有实例对象。为避免抽象类出现实例对象，正常类和抽象类的区别被定义在类的实现中。抽象类中有一些方法没有相应的实现：这些方法被称为抽象方法。这意味着抽象类的定义是不完整的，它也因此永远不能创建实例对象。

使用抽象类有几个原因。原因之一是：通过在等级结构图中占据一个虽没有实例对象但仍需要描述的位置，它有助于建立一个逻辑的继承等级结构。下面就是一个抽象类的例子（见图3-1)，该类属于等级结构图的一部分。虽然 **Shape** 类永不会创建对象，它仍然在等级图中占有

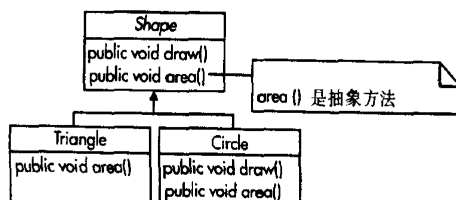


图 3-1 抽象类的例子

一个位置。在 UML v1.4 表示法中，抽象类的名称以斜体表示，抽象方法也表示为斜体。

在继承等级图中有一个位置，其好处体现在使用集合时。在设计中不要使用一个:Triangle对象的集合和另一个:Circle对象的集合，取而代之的是在设计中包含一个:Shape对象的集合，它含有:Triangle对象和:Circle对象。这样做是有好处的，如果目前的集合中再包括由类 Square（另一个从 Shape 类派生的类）创建的对象时，就不需要另外修改源代码了。

使用抽象类的另一个原因将在下一节“应用程序编程接口（API）”中被探讨。为确保一个类不会创建对象，大多数面向对象语言提供用以表明类是抽象类的关键字。

在 Java 语言中，使用关键字“abstract”定义一个抽象类，如：

```
public abstract class <classname>
{
    public abstract <returnType> <methodName> (<arguments>);
}
```

在 C++ 语言中，如果一个类的任何一个方法被声明为“纯虚的（pure virtual）”，该类就被描述为“是抽象的”。一个纯虚的方法使用纯指定符“=0”，如：

```
public class <className>
{
    virtual <returnType> <methodName> (<arguments>) = 0;
}
```

3.2 应用程序编程接口

在第 1 章中，当方法被开始引入时，它们被共同地描述为类的接口。对正常的继承来说，不仅类的接口被继承，实现接口的方法也被继承。这里，应用程序编程接口（API）被定义为一个只允许其接口被继承的类。一个 API 类通常用于将若干其他类的接口集在同一个类中。注意到 API 类只包含接口，而从不会直接创建对象是很重要的。API 类因而就是一个抽象类。其中声明为“抽象”的方法被派生类继承为纯接口（interface - only）方法。

为将 API 类与其他类联系起来，它们必须形成一个等级结构，并把 API 类作为根类。但是与继承等级结构不同，子类不继承属性和方法，取而代之的是，它们可以提供包含在接口中的方法的实现。所有的接口方法都被定义在 API 类中，每一个子类以自己的方式实现这些方法。

3.2.1 API 结构出现以前

在 API 结构出现之前，如果一个对象X:要与构建同一接口的来自其他类的对象进行交互，对象X:需要有一个特定的指向其他类的引用。

下面的例子中，一个小型机场的空中交通管理员想知道使用该机场的大多数飞机的详细资料（见图 3-2）。

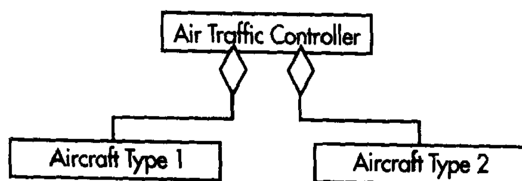


图 3-2 空中交通管理的关系图

事实上，他可能认出每一架飞机，认识它们的飞行员，叫得出它们的姓名。管理员知道Eric有一架呼叫信号为“XY123Z”的飞机，他想知道Eric的飞机目前的飞行高度：

```
Eric = new TwinJet (callsign = "XY123Z");
altitude = Eric.altitude ();
```

显然，使用机场的飞机类型越多，空中交通管理员的任务就越困难。要管理员认识每一种类型的飞机是很不现实的，因此，当一架新型的飞机不期而至时，会发生什么情况呢？管理员仅仅会因为以前从没见过这种飞机就拒绝让它着陆吗？

3.2.2　为什么使用API结构

在上面的例子中，情况是这样的，当新型的飞机到来时，空中交通管理员假定飞机遵循一些基本准则，有一个基本接口，被标识为‘plane’，是飞机的默认通用分类。

如果所有的飞机都被标识为‘plane’，只要不妨碍执行飞行任务，管理员是不会太关心的。总之，不管飞机实际上是什么类型，它们在雷达显示屏上的小点是一样大的。因此，‘plane’成为所有飞机的API类。只要每一种飞机对每一个‘plane’的交互活动都有一种回应，‘plane’API类就足够了。

3.2.3　从API类中派生

正如本章开始时提到的，一个API类是一个继承等级结构的“根”。飞机的继承等级结构图显示如图3-3所示。不同的类**AircraftType1**和**AircraftType2**继承了定义在类**AirTrafficControllerAircraft**中的所有属性和方法。

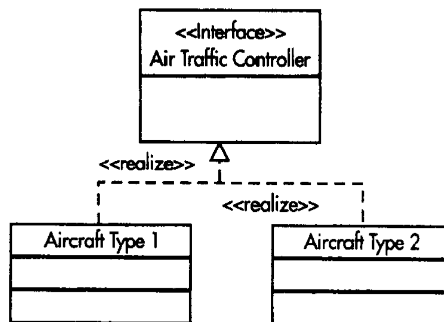


图3-3　飞机的继承等级结构图

3.2.4　使用API类

UML v1.4中，将一个类与API类联系起来的表示法是这样显示的：继承等级结构图被替换为一个API类表示法的组合。类**AircraftType1**不再被显示为类**AirTrafficController Aircraft**的派生类，而是被显示成与一个代表API类的圆相连接。

使用方法图显示如图 3-4 所示。该图中显示了一个附加的类 **AirTraffic Controller**，它使用被定义在 API 类中的属性和方法与派生类进行交互。

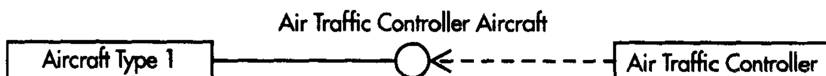


图 3-4 方法图图例

与类 **AircraftType1** 相同的喷气式飞机实现了被描述为‘plane’的接口（与类 **AirTrafficControllerAircraft** 相同），并被空中交通管理员使用。完全的解释性的使用方法图显示如图 3-5 所示。

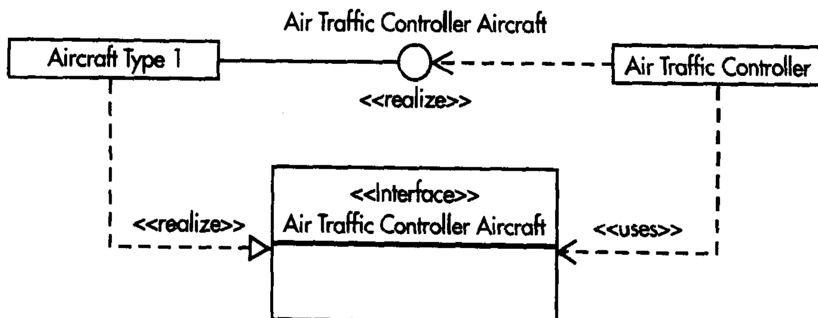


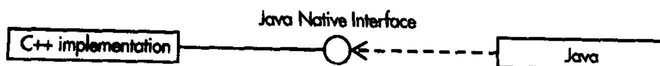
图 3-5 完全的方法图图例

注意，在任何情况下，用户，即空中交通管理员，要被设计为与 **AirTrafficControllerAircraft** 对话，而不是与各种具体类型的飞机对话。这是为了保证他们不偏离既定的接口。

3.2.5 Java 原始接口

如果用户直接从 Java 中使用系统的原始特性，API 类也可被用到。这就是 Java 原始接口 (Java Native Interface，JNI)。它允许用户使用 Java 的方式访问系统，但实际的实现是在一种编程语言如 C++ 中进行的。Java 接口类被编译，然后被处理，产生一个头文件（.h），以便在 C++ 程序中使用。随后，写出实现所需特性的 C++ 类，该类被编译并形成一个共享库，被 Java 程序使用。

这种模型与 API 模型如出一辙，因为 Java 类提供接口而不提供隐藏在 C++ 库中的实现。重要的是要注意到：尽管 C++ 库中的实现改变了，Java 程序不必重新构建。只有接口改变了，程序才需要重新构建，正如任何其他的 Java 类改变了一样。



一个关于JNI的例子描述如下：

1. Dinner.java

```
..
public class Dinner
{
    static
    {
        System.loadLibrary("phil_java");
    }

..
    public native void StartUp();
}
```

创建文件Dinner.java后，下一步是用‘javac’编译该文件。‘javac’是Java编译器，可在Java开发工具包中获得：

```
javac Dinner.java
```

我们编译了文件‘Dinner.class’后，使用‘javah’可获得头文件。‘javah’产生描述指定类的C语言头文件，这些C语言文件提供实现原始方法必需的信息：

```
javah -jni Dinner
```

产生的头文件显示如下。

2. Dinner.h

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class Dinner */

#ifndef _Included_Dinner
#define _Included_Dinner
#ifdef __cplusplus
extern "C" {
#endif
/* Inaccessible static: threadGroup */
/*
 * Class:     Dinner
 * Method:    StartUp
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_Dinner_StartUp
  (JNIEnv *, jobject);

#ifdef __cplusplus
}
#endif
#endif
```

使用产生的头文件，原始代码就可被写出，摘录如下。

3.phils.cpp

```
# include "Dinner.h"
..
JNIEXPORT void JNICALL
Java_Dinner_StartUp( JNIEnv *env, jobject obj )
{
..
}
```

现在，原始代码应该被编译，并被构建到一个共享库中，共享库的名称与最初在 Dinner.java文件中指定的一致。本例中，共享库的名称在 Solaris 平台上是‘libphil_java.so’，如果使用 Microsoft 平台，共享库是‘phil_java.dll’。

3.3 模板

模板是一种产生通用类的机制。通用类定义一套操作某种数据类型的方法，当该类的对象被创建时，这种数据类型被指定。通用类指定处理一种抽象数据类型的逻辑。

集合类就是一个好例子。区分集合类的惟一的条件是集合类创建的对象。一个通用集合类会使用与其他集合类相同的方法，但是，通用集合类包含的对象的类型在以后定义。当从一个通用类创建对象时，数据类型被指定。在 C++ 中，像通用类这样的术语被称为模板。

模板集合类的定义与普通的集合类一样，但有以下几点例外：

- ▶被包含的对象由‘T’定义。
- ▶类定义的前缀为‘template <class T>’，这是给编译器的指令。

UML v1.4 中关于模板的表示法如图 3-6 所示。一个名为 **Set** 的模板类用于创建另一个名为 **Students** 的类，基于的数据类型为类 **Student**。

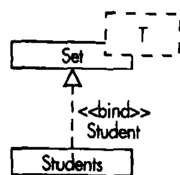


图 3-6 模板的表示法示例

在 C++ 中，按照下面的方法创建模板，并给定使用的对象类型：

```
RealCollection    < collectStudents>        myCollectionOfStudents;
```

3.3.1 何时使用模板而不使用继承

下面两个例子说明了如何在使用模板和使用继承之间做出选择。

- ▶例子 1 要求提供上学的不同的学生的集合。其中，每个集合将只包含学习某特定课程的学生，如，一个学习物理的学生的集合和另一个学习历史的学生的集合。被支持的方法只有创建、析构、增加和删除。
- ▶例子 2 要求提供上学的不同的学生的集合。但与例子 1 不同，集合被设计为包含不同年龄组的学生，如，11~12 岁、12~13 岁、13~14 岁等等。被支持的方法为创建、删除，以及被教授的课程、被教授的方式。

这两个例子在描述上看起来比较类似，但它们导致两种完全不同的设计。

类的行为与保存在集合中的对象的类型是相互关联的，如，集合中的方法的实现是否需要每个类有不同的实现。例如，在例1中，集合中的方法与类的不同类型无关，但在例2中，至少有一个方法是不同的：尤其是课程的教授方式。

所以，对例1，设计可以通过基于模板的集合来实现，而对于例2，设计使用继承来实现。

3.3.2　在C++中实现的模板样本

以下是任何类都需要的两个文件，一个头文件和一个实现文件。它们表示了一个‘Set’是怎样被编写为一个模板的。

1. 类头文件

```
#ifndef SET_H_
#define SET_H_

enum Status { OK, FULL, EMPTY, ERROR };

template <class T>
class Set
{
    public:
        // Constructor
        Set ( int numElements );

        // Destructor
        ~Set ();

        // Container Methods
        Status add ( T *value );
        Status read ( int index, T ** value );
        Status removeIndex ( int index );
        Status removeValue ( T *value );
        Status resizeSet ( int newSize );

        // Information methods
        Status getTotalSize ( int *size );
        Status getCurrentSize ( int *size );

    private:
        // The container
            T    **base_;

            int    totalSize_;
            int    currentSize_;
    };
    #endif // SET_H_
```

2. 类实现文件

```
#include <stdlib.h>
#include "Set.h"

template <class T>
Set<T>::Set ( int numElements )
{
    // create a container whose size is given as a parameter
    // to this method
}

template <class T>
Set<T>::~Set ()
{
    // delete the elements created by the constructor
}

template <class T>
Status Set<T>::add ( T *value )
{
    // check that there is still space in the container for the new item
    // if there is enough room, add the value
    // if there is not enough room, call resetSet to increase the size
}

template <class T>
Status Set<T>::read ( int index, T **value )
{
    // check that the required index is within the current range
    // if the index is within range, return the value
}

template <class T>
Status Set<T>::removeIndex ( int index )
{
    // check that the required index is within the current range
    // if the index is within range, delete the value
    // then reset the container to eliminate the deleted value
}

template <class T>
Status Set<T>::removeValue ( T *value )
{
    // try to determine if the value is in the set
    // if the value is not in the set return an error
    // if the value is in the set, delete the value
    // then reset the container to eliminate the deleted value
}

template <class T>
Status Set<T>::resizeSet ( int newSize )
{
```

```
    // increase the size of the allocated space
}

template <class T>
Status Set<T>::getTotalSize ( int *size )
{
    *size = totalSize_;
    return OK;
}

template <class T>
Status Set<T>::getCurrentSize ( int *size )
{
    *size = currentSize_;
    return OK;
}
```

3. 创建一组整数

```
Set <int>    setOfInt;
```

3.4 好的设计——原则和度量标准

设计的原则和度量标准这一议题基于 Robert C. Martin（来自 ObjectMentor）和 Dr. Linda Rosenburg（来自 NASA 的 Software Assurance Technology Center）的若干著作，以及一项研究的成果。[①]

3.4.1 认识设计中“毒瘤”产生的原因

本节讨论设计中“毒瘤”的不同方面。毒瘤的四个被认识的症状是：僵化（rigidity）、脆弱（fragility）、不可移植性（immobility）、粘性（viscosity）。

- **僵化** 僵化一词用到软件上，是指软件难于修改，包括简单的修改。出现这样的问题是由于软件模块之间的耦合过于紧密，任何变化都会像涟漪一般引起依赖模块的其他变化。
- **脆弱** 与僵化密切相关的另一个症状是脆弱。脆弱用于软件时，是指软件的每一次变化会涉及到很多地方。脆弱的另一方面的表现是，软件在一个区域的变化会涉及到看起来与变化区域没有概念上的关联的其他区域。
- **不可移植性** 不可移植性是指应用程序在从其他项目或本项目的其他部分复用类方面的无能。这个问题是由那些试图节约时间和人力的工程师提出的。软件工程师找到了一个类，其功能和特点与他们所需要的类似。但是，往往是这样：这个类还有一些不需要的功能，或者类处理的问题稍有区别。进行一番分析后，工程师发现从这个类中分离出所需的功能需要太多的工作量，复用只是一个不可行的建议。结果是，干脆重写这个类，

① 参考：Robert C Martin，Design principles and design patterns，http：www.objectmentor.com；
Dr.Linda Rosenburg，Software Quality Metrics for Object Oriented System Environments，satc.gsfc.nasa.gov.

而不再复用了。

- ▶**粘性** 粘性是一个描述实施保持设计的更改的难易程度的词语。如果实现保持设计的方法比较困难，就说明该设计的粘性高。做错事情容易，但做正确的事情就难了。以上四个症状中，任何一个都是不良设计体系的证据标志，任何表现出这些症状的应用程序都会遭受将死于“毒瘤”的设计的折磨。一个重要的问题是，是什么导致了这些烂根情况的发生?

需求的变化

正如每个设计者和开发者所想到的，设计“毒瘤”的通常的原因源于最初的设计没能估计到的新的需求。当新的开发者，即不属于原先的开发小组的开发者要做一些更改时，另外的问题就出现了。这些新的开发者不太欣赏原有的设计，所以他们的更改可能在不知不觉中与原有的设计相冲突。如果一个设计由于需求变化的压力而失败，那么问题在于设计本身。不管怎样，都需要寻求一种方法，使得设计具有变化的弹性，免受设计“毒瘤”的伤害。

3.4.2 面向对象的类的设计原则

每当类之间的依赖关系改变时，这些变化都直接或间接地引起前面提到的四个症状。为了有利于管理类依赖关系改变引起的后果，下面列出几个被证明是行之有效的基本原则。本节讨论的原则有：开闭原则（Open Closed Principle，OCP）、雷斯科夫替换原则（Liskov Substitution Principle，LSP）、依赖性转换原则（Dependency Inversion Principle，DIP）。这些原则以及其他的原则在 Robert C. Martin 的著作中都有深入的探讨。

1. 开闭原则

开闭原则（OCP）的定义表明：一个模块在扩展性方面应该是开放的，在更改性方面应该是封闭的。

在所有的面向对象设计原则中，开闭原则是最重要的。简而言之，写出的类应该可以被扩展，类的功能可以变化，而不需改变类的源代码。

实现开闭原则有若干技术，这些技术均建立在抽象之上。本节将阐明几项这样的技术。

动态多态性（Dynamic Polymorphism） 下面的例子显示：每当需要新的功能以支持新的调制解调器（“猫”，Modem）时，函数 LogOn 都必须被修改。问题更严重的是，每一种新类型的“猫”都将引起对Modem::Type 枚举类型的修改，从而使得现有的、依赖于该枚举类型的编码不得不被重新编译。

```
Listing for LogOn (which must be modified to be extended)
struct Modem
{
    enum Type {hayes, courrier, newModem) type;
};
struct Hayes
{
    Modem::Type type;
    // Hayes related stuff
```

```
};
struct Courrier
{
    Modem::Type type;
    // Courrier related stuff
};
struct NewModem
{
    Modem::Type type;
    // NewModem related stuff
};
void LogOn(Modem& m, string& pno, string& user, string& pw)
{
    if (m.type == Modem::hayes)
    {
        DialHayes((Hayes&)m, pno);
    }
    else if (m.type == Modem::courrier)
    {
        DialCourrier((Courrier&)m, pno);
    }
    else if (m.type == Modem::newModem)
    {
        DialNewModem ((NewModem&)m, pno)
    }
}
```

上述简单例子的编码表明，问题源于 if/else 或 switch 语句的使用。这些语句的问题是，每当需要修改时，if/else 或 switch 语句均需要更新，以选择使用正确的函数。当需要增加功能时，每个选择语句都必须被查寻到并被修改。

如果一个程序员试图表现自己的才能，在一些假设的基础上对代码进行优化，就会存在一个实际的问题。本例中，Hayes 和 Courirer 调制解调器的功能完全相同，所以可能有类似于以下代码的优化：

```
if (modem.type == Modem::newModem)
{
    SendNewModem((NewModem&)modem, c);
}
else
{
    SendHayes((Hayes&)modem, c);
}
```

如果 if/else 或 switch 语句没有被充分地处理，如果 Hayes 和 Courirer 调制解调器的“发送(send)”功能有了区别，这个优化将会导致各种各样的复杂问题。开发者将花费相当长的时间确定应用程序无法正确运行的原因，他们原以为调用的函数肯定是正确的，付出了辛苦，却从

没想到代码是被“优化”过的。

一个关于如何实现 OCP 的例子如图 3-7 所示，它是使用接口构造的。现在，LogOn 函数依赖于 Modem 接口。这样，增加新的“猫”就不会引起 LogOn 函数的变化。这个例子显示了一个可被扩展的类，当增加新的“猫”时，类本身不会被修改。

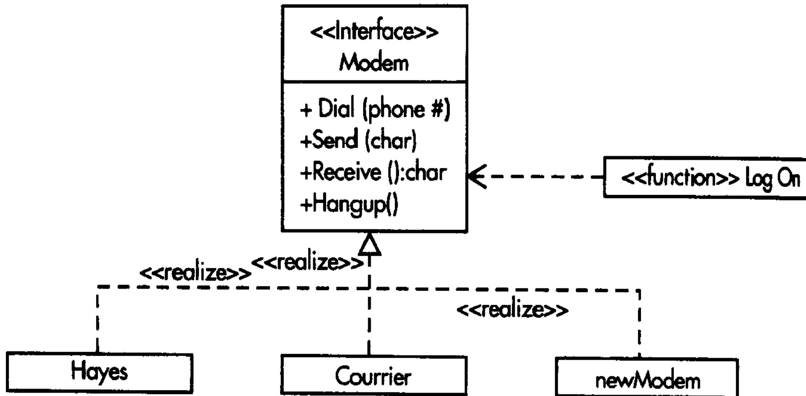


图 3-7 使用接口实现 OCP

新的 Modem 接口的代码显示如下：

```
LogOn has been closed for modification
class Modem
{
    public:
        virtual void Dial(const string& pno) = 0;
        virtual void Send(char) = 0;
        virtual char Recv() = 0;
        virtual void Hangup() = 0;
};
void LogOn(Modem& m, string& pno, string& user, string& pw)
{
    m.Dial(pno);
    // etc.
}
```

静态多态性（Static Polymorphism） 模板是另一项遵循 OCP 的技术。下面的代码显示了这种技术是如何实现的。LogOn 函数可被扩展到很多不同类型的“猫”，并不需要更改代码。

```
LogOn is closed for modification through static polymorphism
template <typename MODEM>
void LogOn(MODEM& m, string& pno, string& user, string& pw)
{
   m.Dial(pno);
   // you get the idea.
}
```

OCP 小结：虽然 OCP 难以完全实现，但即便是部分地遵循 OCP，也可使应用程序的结构得到巨大的改进。

2. 依赖性转换原则

依赖性转换原则（DIP）的定义表明，设计应该依赖于抽象特征，不应依赖于具体细节。抽象特征定义了系统的接口。本章前几节中谈到的应用程序编程接口（API）就是抽象特征的一个例子。

依赖于抽象特征 这个原则隐含的意义很简单。设计中的每个依赖关系都应该指向一个接口，或一个抽象类，正如图3-7所示，每个方法都引用接口类Modem，而不是直接引用一个具体的“猫”类。

依赖关系不应该指向一个具体的类。这个限制可能看起来有一点残酷，但还是应该尽可能地遵循这个原则。原因之一是接口在设计过程的早期就可被达成共识，因而变化也较具体类要小。

另外，正如OCP中谈到的那样，接口和抽象类的设计是很灵活的。

缓和的力量 DIP背后的推动力是试图减小对易变类的依赖性。DIP的一个假设就是任何具体的事物都是易变的。当设计和应用程序处于早期阶段时，这个假设多数情况下都是成立的。当然，任何假设都存在异常情况。例如，标准C语言库中的头文件（.h）部分是非常具体的，但它们根本不变。因此，依赖这些头文件是没有害处的。类似的异常情况还可能发生在来自其他项目或外部公司的类或类包上，因为它们不大可能变化，也不大可能将易变性引入你的设计中。

不管怎样，正因为有些事情被认为是不变的，除了不使用抽象接口外，没有替代的方法。例如，如果强迫你使用Unicode字符，一个使用标准字符串的应用程序就将面临问题了。

说明：Unicode是一种16位字符编码体系，它被设计用来支持现代世界的多种书面文本的显示和其他方面的特性。

对象创建 对于DIP这种思想，存在一个问题，即：当应用程序需要创建类的实例时，因为实例只能从具体的类创建，因此，创建实例依赖于存在一个具体的类，而这样的类却是设计时应尽量避免的。

由于实例的创建可能发生在设计体系的任何地方，脱离具体类和对于具体类的依赖关系似乎是根本不可能的。但是，对这个问题有一个解决方法：实现一个“factory”。这个解决方法对本书来说太深了，将留给高一级的读物。

3.4.3 设计的度量标准

下面将讨论衡量设计质量的六个度量标准。每个标准均与面向对象系统特别相关。表3-1中，每个标准都与适用的面向对象结构一同显示。

表3-1 衡量设计质量的六个度量标准

度量标准名称	描 述	面向对象结构
WMC	类方法数	类/方法
RFC	类应答数	类/消息
LCOM	方法关联性缺乏度	类/关联
CBO	对象之间的耦合度	耦合
DIT	继承树深度	继承
NOC	子类数量	继承

1. 度量评价准则

对于传统的设计方法（即结构化的设计方法），处理设计结构和（或）数据结构的度量标准是相互区别、各自独立的。而对于面向对象设计，其度量标准必须处理对象，每个对象是属性和方法的结合体。因此，面向对象度量准则应该评价以下几个方面：

▶效率：设计的实现是否高效？

▶复杂性：设计中的体系结构的复杂度如何？

▶清晰度/可用性：设计是否易于理解和易于使用？

▶可复用性/特殊性：设计是可复用的还是应用程序特有的？

▶可测试性/可维护性：设计是否支持测试功能？

任何一个度量标准都必须至少在上述五个方面之一进行有效的度量。下面对表3-1中的每个标准都将给出一个描述性说明、一个使用方法的例子和对得到的结果的分析。用这些标准衡量体系的主要结构，这些结构若不能被适当地建立，将对设计和代码质量属性有负面的影响。

2. 实例设计

面向对象设计要求另外一种不同的思维方式。本例如图3-8所示，它的开发是为了揭示面向对象的概念和如何应用以上提到的度量标准。Shape是所有几何图形的集合，是一个超类。

Shape有一个画特定图形的方法和一个计算周长的方法。从Shape出发，我们可以开发两个子类：三角形和四边形，这两个类之所以成为类，是因为它们也有子类。

一个三角形有三条边、三个角、一条底边和一个高。当创建三角形时，还有一个约束，即三角形三角之和为180度。增加一个新的方法用于计算三角形的面积，它的计算方式是1/2×(底边长×高)。三角形有三个子类：scalene（不等边三角形）、equilateral（等边三角形）和isosceles（等腰三角形）。

不等边三角形不引入新的属性和方法，但它却在创建方法（构造函数）中增加一个约束，即：任意两边均不等长。

等腰三角形也不引入新的属性和方法，但它也在创建方法（构造函数）中增加一个约束，即两条边被设为等长。底（base）被设为第三条边长的一半，高被设为［边长^2 - 底^2］的平方根。㊀

等边三角形也不引入新的属性和方法，但它也在创建方法（构造函数）中增加一个约束，即所有边都等长，三个角都为60度。底被设为边长的一半，高被设为（sin60°×边长）。㊀

一个四边形（quadrilateral）有四条边和四个角，创建四边形时，有一个约束，即四个角之和为360度。类quadrilateral有两个子类：trapezium（梯形）和rectangle（矩形）。

梯形引入一个新的属性，“高（height）”，该属性由一个新的方法“area”使用。方法“area”中，用相互平行的两条边的长与高相乘。创建方法中有一个约束：有两条边相互平行。

矩形不引入新的属性和方法。创建方法中有一个约束，即边1和边3等长，边2和边4等长，并且所有四个角均相等（为90度）。有两条边相互平行。方法“area”被修改为边1与边2相乘。rectangle只有一个子类：square（正方形）。

㊀ 按这样的定义，三角形的面积＝底×高。——编者注

Square不引入新的属性和方法。创建方法中有一个约束。方法“area”被修改为边 1 的平方。完整的类继承等级图如图 3-8 所示。

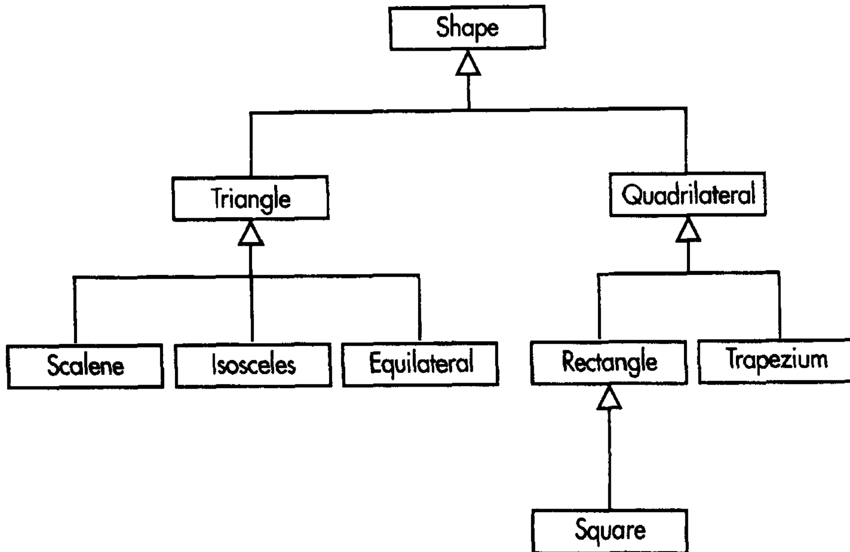


图 3-8　Shape 的继承等级结构

3. 基于类的度量标准

以下描述的前三个度量标准根据类中的方法来衡量类的复杂度：消息和关联。

类方法数（Weighted Methods per Class）　对 WMC 标准有两种解释。第一种被称为方法复杂度，本书将不予探讨；第二种解释是指在一个类中实现的方法的数量。表 3-2 显示了从图 3-8 的等级结构中收集的信息，表明了各个类的 WMC 值。

表 3-2　Shape 继承等级结构中的类与类方法数

类	类方法数
Shape	2
Triangle	2
Scalene triangle	1
Equilateral triangle	1
Isosceles triangle	1
Quadrilateral	1
Trapezium	2
Rectangle	2
Square	2

经验证明这样的结论：一个类中的方法的数量越大，类的影响就越大。含有大量方法的类

的影响范围，从所需的测试工作量到可能只为特定的应用程序所用。方法众多的类会限制复用的可能性。WMC标准可用于度量可用性和可复用性。

类应答数（Response For a Class） 类应答数（RFC）度量标准就是类中存在的、可被发送到类的对象的消息激活的方法的数量，它包括类等级结构中所有的可访问的方法。表3-3再次使用了图3-8中的等级结构图中的信息，表示了各类的RFC值。

表3-3 Shape继承等级结构中的类与类应答数

类	类应答数
Shape	2
Triangle	4
Scalene triangle	5
Equilateral triangle	5
Isosceles triangle	5
Quadrilateral	3
Trapezium	5
Rectangle	5
Square	7

类中可被消息激活的方法的数量越大，类的复杂性就越大。如果大量的方法可因应答消息被激活，类的测试和调试就变得复杂起来，因为对测试人员来说，需要更深层次的理解。

RFC标准用于评价系统设计、可用性和可测试性。

方法关联性缺乏度（Lack of Cohesion Of Methods） 方法关联性缺乏度（LCOM）度量标准通过实例变量或属性衡量方法的相似程度。任何方法分离性的度量都有助于识别类设计中的瑕疵。

关联性是类中的方法的相互关联，共同协作，提供完整周密的行为的程度，所以关联性指设计各部分的内在一致性。关联性以被封装在对象中的数据为中心，关注方法是如何与数据交互，完成整个行为的。方法的相似程度是类关联性的一个重要方面，我们的目标是实现最大的关联度。适应性好和可复用的程序具有耦合性低，关联性高的特点。

有效的面向对象设计可最大限度地增加关联性，因为它能提高封装性能。方法关联性缺乏度标准就是针对关联性的。

度量关联性至少有两种不同的方法：

- ▶对类中的每一个数据项，计算使用该数据项的方法在方法中所占的百分比，然后对这些百分比平均，再从100%中减去这个平均百分比。得到的百分数越低，意味着类中的数据和方法的关联性越高。
- ▶如果方法对相同的属性操作，那么，它们的相似性较大。在由方法使用的属性集合的交集中，对产生的不连通集合的数量进行计数，这是另一种计算关联性的方法。

表3-4还是使用图3-8中的等级结构图中的信息，表示了各个类的LCOM值。

表3-4　Shape继承等级结构中的类与方法关联性缺乏度

类	LCOM值
Shape	100%
Triangle	37.5%
Scalene triangle	37.5%
Equilateral triangle	37.5%
Isosceles triangle	37.5%
Quadrilateral	22⅓%
Trapezium	43.75%
Rectangle	43.75%
Square	43.75%

关联度高表明类的划分合理，缺乏关联性或关联性低会增加系统的复杂性，因而会增加开发过程中的错误概率。低关联性的类可被划分为两个或多个关联性高的子类。这个标准用于评价设计实现和可复用性。

对象类之间的耦合度（Coupling Between Object Classes）　对象类之间的耦合度（CBO）度量是与一个类耦合的其他类的数量。它是通过计算与一个类所依赖的继承等级结构明确不相关的类的数量而获得的。

耦合是衡量一个实体与另一个实体之间所建立的联系的强度的尺度。类（或对象）的耦合表现在三个方面：

▶当消息在对象之间传递时，对象是耦合的。

▶当类之间相互交互时，它们是耦合的，即在一个类中声明的方法使用另一个类中的方法。

▶继承性在父类和子类之间引入了重要而紧密的耦合关系。

在这里考虑继承等级结构中的耦合关系没有意义。CBO度量标准是根据类之间的交互活动而不是根据继承关系计算出来的。

过分的耦合对模块设计有害，会妨碍代码复用。类越独立，就越容易被其他应用程序复用。耦合度的数值越大，对其他设计部分变化的敏感性越高，因而维护也就越困难。强耦合关系会使系统复杂化，因为如果其中的模块与其他模块关联，就难于理解、改变和纠错。系统设计过程中，若能保证模块间的耦合尽可能达到最弱，就会减低系统的复杂性，这样就可以提高设计的模块化程度和封装性。

CBO度量设计实现和可复用性。

继承树深度（Depth of Inheritance Tree）　继承树深度（DIT）定义为，继承等级结构图中的一个类的深度是从该类的节点到继承树的根的最大长度，通过祖先类的数量来度量。

面向对象系统中另一个设计抽象方式是使用继承。继承是类之间的一种关系，能让编程者复用以前定义的对象，包括变量和操作符。通过降低操作和操作符的数量，继承降低了复杂

性，但这种对象抽象方式可能造成维护和设计的困难。用于度量继承关系的两个标准是继承等级结构的深度和宽度。

表 3-5 是从图 3-8 的等级结构中收集的信息，显示了各个类的 DIT 值。

表 3-5 Shape 继承等级结构中的类与继承树深度

类	继承树深度值
Shape	0
Triangle	1
Scalene triangle	2
Equilateral triangle	2
Isosceles triangle	2
Quadrilateral	1
Trapezium	2
Rectangle	2
Square	3

一个类在等级结构中的位置越深，它继承的方法的数量就可能越多，从而就越难于预测它的行为。较深的继承树会增大设计复杂度，因为会涉及较多的方法和类，但继承的方法被复用的潜力也较大。一个支持 DIT 的度量标准是继承方法数（Number of Methods Inherited，NMI）。

DIT 标准主要度量复用性，但也与可理解性和可测试性有关。

子类数量（Number Of Children） 子类数量（NOC）指继承等级结构中直接从属于一个类的子类的数量（见表 3-6），它是一个类在设计和系统中的潜在影响力的指示器。

表 3-6 Shape 继承等级结构中的类与子类数量

类	子类数量
Shape	2
Triangle	3
Scalene triangle	0
Equilateral triangle	0
Isosceles triangle	0
Quadrilateral	2
Trapezium	1
Rectangle	1
Square	0

该表仍然是收集图 3-8 中的等级结构中的信息，表示了各个类的 NOC 值。

子类的数量越大，对父类抽象不当的可能性也就越大，因此很可能是一个子类错用的案例。但是，子类的数量越大，复用程度也越高，因为继承就是一种复用。如果一个类有大量的子类，可能需要对类的方法做更多的测试，这样就拉长了测试时间。因此，NOC 主要度量可

测试性和设计。

4. 度量标准小结

除了评估软件质量方面的性质外，软件度量标准还应该满足某些理论上的准则。这些准则是根据适用该标准的面向对象结构来制定的。

- ▶特别性：不能每个类的某项度量标准值都相等，如果相等，该项标准就失去了度量的价值。
- ▶非惟一性（等值观念）：两个类可能含有相等的度量标准值（即两个类具有相同的复杂性）。
- ▶设计细节很重要：即使两个类被设计实现同样的功能，在确定类的度量标准值时，设计细节仍很重要。
- ▶单调性：两个类组合后的度量标准值应不小于任一个组件类的度量标准值。
- ▶交互活动的差异：某两个类之间的交互活动可能与其他两个类之间的交互活动不同，各种组合具有不同的复杂度值。
- ▶交互活动增加复杂度：当两个类组合时，类之间的交互可能会使复杂性的度量值增大。

表3-7根据面向对象设计的关键概念：方法、类（关联性）、耦合和继承，总结了各种度量标准。虽然存在一些建议的门槛值，但几乎没有哪个应用程序的数据能支持或说明它们。因此，表3-7给出度量标准的一个大概的解释（如，较大的数值表明应用程序依赖性，即非通用性）。

表3-7 面向对象的度量标准

度量标准	面向对象特点	度量的理念	度量方法	解　释
WMC 类方法数	类/方法	复杂性 可用性 可复用性	1. #类中实现的方法的数量 2. 各方法的复杂性之和	较大=>通过继承对子类的潜在影响越大；具有应用程序依赖性
RFC 类应答数	类/方法	设计可用性 可测试性	#因回应消息而被激活的方法数	较大=>复杂性高，降低可理解度；测试和调试更复杂
LCOM 方法关联性缺乏度	类/关联	设计 可复用性	类中由属性决定的方法间的相似性	低=>类中的关联性好 高=>类中的关联性差——将类拆分
CBO 对象之间的耦合度	耦合	设计 可复用性	#明确的非继承相关的类的数量	高=>设计低劣；难于理解；降低复用性；增加维护成本
DIT 继承树深度	继承	可复用性 可理解性 可测试性	从类节点到根的最大长度	较高=>更复杂；可重用性好
NOC 子类数量	继承	设计	#直接子类的数量	较高=>可复用；设计不好，增加测试量。过度地关注一个类

3.5 全局对象

全局对象是一个类的实例，它具有应用程序范围，即，是一个可被系统中任何地方的其他对象访问的对象。很多情况下，需要类的单个的或数量受控制的实例，例如，一个公司的独个的 CEO（首席执行官）或足球比赛场上的独个的裁判或冰球比赛中的两位裁判。

创建一个 CEO 或裁判的类是很容易的，问题在于限制创建的实例的数量。解决方法似乎是显而易见的，让类跟踪存在的实例的数量，不允许存在的实例数超过预定的数值。

如果一个类具有单个实例是全局对象的惟一目的，那么倒是可以考虑使用类方法。类方法确实允许从系统的任何地方访问一个对象，但它们不具有使用实例的灵活性。类方法的一些不利之处在于：

▶类方法不能被子类重载，而实例方法可被重载。

▶实例方法可被子类重载，因而具有功能上的灵活性。

▶有一个能控制自身实例数量的类，可提供较类方法更大的灵活性（灵活性正是类方法所缺乏的）。

实现

在 Smalltalk 中，使用了 80 个共享池。任何一个希望拥有更多听众的对象被安排在一个共享池中，其他对象就可以访问共享池，以便找到他们需要的对象。这个例子将“MyObject”设为共享池“Smalltalk”的一部分。

如，Smalltalk at：# MyObject put：nil

与系统中其他的大多数对象不同，全局对象直到第一次被使用时，才有必要被创建。在全局对象的类定义中定义单个类方法，用于获取对全局对象的访问权。

类方法首次被调用时，它创建全局对象，并返回对象的句柄。对类方法的随后调用中，全局对象的句柄被返回。

全局对象的表示法如图 3-9 所示。

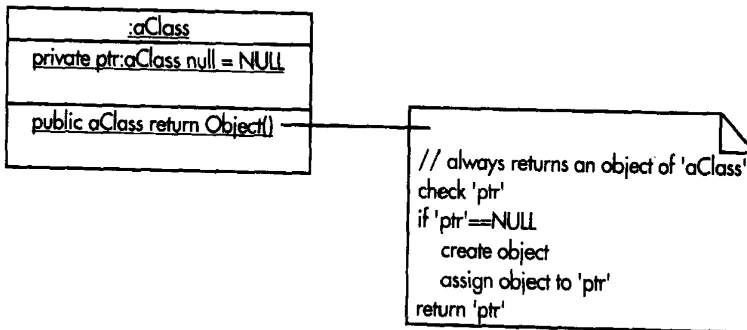


图 3-9 全局对象的表示法

要访问全局对象中的任何类方法以外的其他方法，可使用如下的访问方法：

```
answer = GlobalObjectClassDefinition::ReturnObject ().question();
```

下面的例子说明了如何在 Java 程序中实现一个全局对象。

1.GlobalObj.java

```
public class GlobalObj
{
    /*
    ** This is the reference to the single instance of this class
    */
    private static GlobalObj           globalObj;

    public GlobalObj ()
    {
    /*
        ** Assign this object to the variable
        */
        globalObj = this;
        :
    }

    public static void main (String[] args)
    {
        /*
        ** Create the main application object
        */
        GlobalObj go = new GlobalObj ();
    }

    public static GlobalObj GetThis ()
    {
        /*
        ** When requested, return the reference to the object
        */
        return globalObj;
    }
}
```

下面的代码说明了如何使用Java全局对象：

```
SimCo    parent;
parent = (SimCo)SimCo.GetThis();
```

下面的例子说明了如何在C++程序中实现一个全局对象。

2.Global.h

```
#ifndef GLOBAL_H_
#define GLOBAL_H_

class Global
{
   public:
```

```
        // static / class method that returns a handle to the global
        // object
        static    Global    &GetGlobal ();
        char*     name();

    private:
        // the global variable used to hold the handle to the global
        // object
        static    Global    *myGlobal_;
};
#endif // GLOBAL_H_
```

3.Global.cpp

```
#include "Global.h"
// initialisation of the global variable
Global    *Global::myGlobal_ = 0;

// the static / class method that will return a handle to the global
// object
Global&
GetGlobal ()
{
    // if this is the first time that this method has been used then
    // the value
    // of the global variable will be NULL
    if (myGlobal_ == NULL)
    {
        // if this is the case, then create the global object and set
    // the global
    // variable to hold the handle to the global object
    myGlobal_ = new Global;
}

// return the value of the global variable,
// i.e. the handle of the global object
return myGlobal_;
}
```

下面的代码说明了如何使用C++全局对象：

```
name = Global::GetGlobal().name();
```

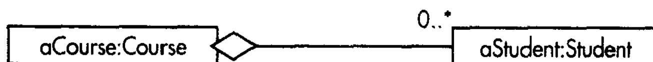
3.6 确定实现方法

本例说明，在设计中必须考虑不同客户的需求（实现语言的选择）。例如，以下就是三个

设计，它们实现基本相同的功能，但方式却很不相同。

1. 使用一组 Student 对象

本设计中，一个 **Course** 类被设计成包含一组 **Student** 对象：

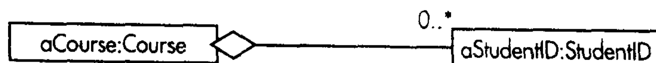


这个设计能使**:Course**对象通过调用一个方法直接访问**:Student**对象，因为所有的**:Student**对象都被包含在**:Course**对象中，是**:Course**对象的局部对象。这个设计的优点在于允许直接访问任何**:Student**对象。

这个设计的问题是（尤其是如果使用C++语言实现）：**:Course**对象中含有一组**:Student**对象的副本。这意味着如果任何一个**:Student**对象被更新了，则**:Course**对象需要更新它的内部的**:Student**对象副本。有些编程语言如 Java 不存在这个问题，因为它们允许从应用程序中的不同地点直接访问对象，而不需要对象以副本的形式存在。

2. 使用一组 Student 标识（ID）

本设计中，**Course** 类包含一组 Student 标识（*StudentID*）。每个 *StudentID* 是惟一的。



:Course对象通过**:StudentID**对象只能直接访问 Student 的标识。

本设计的问题是，在获取系统进一步的信息之前，还要有相当多的准备工作：

(1) 需要找到适当的**:StudentID**。

(2) 需要使用**:StudentID**对象找到相应的**:Student**对象。

(3) 要求**:Student**对象，返回需要的信息。

本设计的优点是只在本地保存任一**:Student**对象的最少信息。这样就允许**:Student**对象脱离**:Course**对象被独立更新。本设计的劣势就是前面提到的从一个指定**:Student**对象获取信息前的种种准备。

3. 使用一组 Student 对象的引用

C++支持本设计中使用的指针，其中，对象本身可被一个引用取代。这样的设计有许多优点：

- ▶具有第一个设计方案所能提供的所有优点，通过引用直接访问对象。
- ▶对象的任何变化不要求对**:Course**对象的更新。

本设计的最大的缺点是，并非所有编程语言都支持引用。

3.7 虚方法

这是 C++的一个独特性质，但若不能恰当地理解，就可能带来问题。当在设计中使用继

承时，重要的是知道何时使用关键字“**virtual**”。关键字“**virtual**”在 C++ 中允许派生类重定义其父类的方法。由于并非每一个方法都需要被重定义，所以不是每一个类都需要与这个关键字联系在一起。下面的例子说明了当一个特殊的方法——析构方法，没有被声明为“**virtual**”时，将出现什么情况。

一个葡萄酒生产监视应用程序中，一个类的定义类似于如下的代码：

```
class WineBottle
{
    public:
        WineBottle ( void ) { numWineBottles++; }
        ~WineBottle ( void ) { numWineBottles--; }
        int getNumWineBottles ( void ) { return numWineBottles; }

    private:
        static int    numWineBottles;
};
```

类变量要按 C++ 的要求被初始化：

```
int WineBottle::numWineBottles = 0;
```

对于大多数的基本应用程序来说，这个类是可以满足需要的，但现在，设计者想要跟踪葡萄酒现有的种类。可以从第一个类 WineBottle 派生一个新类，以便 WhiteWineBottles（白葡萄酒的瓶数）的数量和 WineBottle（总的葡萄酒的瓶数）的总数均被保存。

```
class WhiteWineBottle
{
    public:
        WhiteWineBottle ( void ) { numWhiteWineBottles++; }
        ~WhiteWineBottle ( void ) { numWhiteWineBottles--; }
        int getNumWhiteWineBottles ( void )
        { return numWhiteWineBottles; }

    private:
        static int    numWhiteWineBottles;
};
```

另外，也要初始化类变量：

```
int WhiteWineBottle::numWhiteWineBottles = 0;
```

可以假定，在应用程序的某处，使用“new”创建一个“WhiteWineBottle”对象，稍后使用“delete”将其消除。

```
WineBottle *bottlePtr = new WhiteWineBottle;
...
delete bottlePtr;
```

根据类的定义，两个类的析构函数与相应的构造函数的效果相反。但是，这里有一个错

误：当用“delete”清除 bottlePtr 时，它将不调用 WhiteWineBottle 的析构函数。问题出在析构函数的定义方式。当“delete”操作被激活时，编译器要做出一个选择：应该调用类。WineBottle 的析构函数还是调用类 WhiteWineBottle 的析构函数呢？由于它不能作出决定，它就使用指针变量的析构函数，即 WineBottle 类的析构函数。

对这个问题的解决方法是将析构函数定义为“virtual”。通过把析构函数声明为虚函数，编译器将像处理其他虚函数一样，决定调用的析构函数，因此，被指向的对象的析构函数将被激活，即 WhiteWineBottle 的析构函数被调用。

3.8 复制构造函数

一个复制构造函数用于产生一个传递进来的参数对象的副本。例如，下面的代码从已有的 String 对象创建一个 String 对象的副本。

```
String (const String &object);
```

有两种形式的复制构造函数：表层复制和深层复制。

3.8.1 表层复制构造函数

表层复制构造函数（如图 3-10 所示）将复制对象但不复制它指向的任何数据。

这种复制机制存在一个问题：当原始对象和复制对象被删除时，如图 3-11 所示，其中一个对象怎样知道另一个对象还存在呢？对这个问题，解决方法将在 5.5 节中讨论。

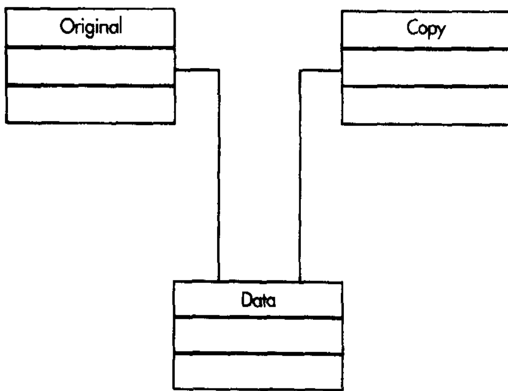


图 3-10　表层复制构造函数

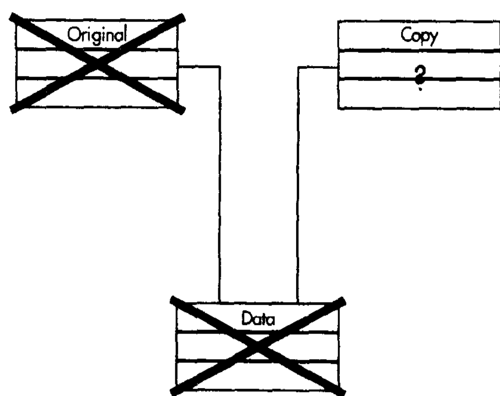


图 3-11　当原始对象和复制对象被删除时

3.8.2 深层复制构造函数

深层复制构造函数（如图 3-12 所示）不但复制对象，并且复制对象指向的一切。

不幸的是，如果大量的对象由复制产生，这种深层复制也存在问题，即内存消耗。这个问题的解决方法与表层复制的解决方法一样，将在 5.5 节中讨论。

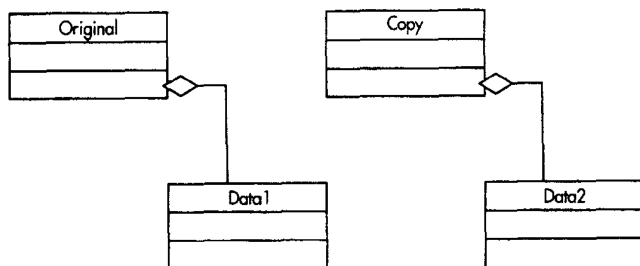


图 3-12 深层复制构造函数

3.9 关联的实现

存在两种形式的关联：一种是双向关联，另一种是单向关联。

3.9.1 双向关联

从图 3-13 的例子可以看出，有两种截然不同的关联：第一种是从“公司”到“雇员”，公司雇用雇员；第二种是从“雇员”到“公司”，雇员被公司雇用。

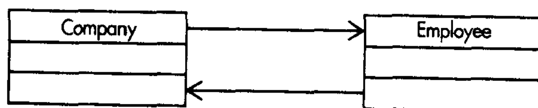


图 3-13 双向关联的例子

这些关联存在于“雇员”，是表示他的公司的属性，也存在于“公司”，是表示其雇员的属性。

由于关联存在于两个方向，当发生变化时，两方均要刷新，这一点很重要。

除了设置与关联对应的属性之外，还有一种表示关联的替代的方法：定义一个类，表达并把握这个关联。这个新类在具体的环境中表示如图 3-14 所示。

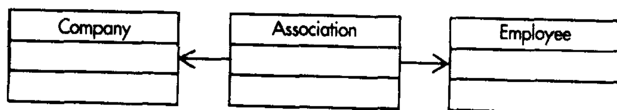


图 3-14 定义一个关联类

3.9.2 单向关联

单向关联表示如图 3-15 所示，每个“雇员”为“公司”完成一项任务，而每个任务不只属于某个雇员。

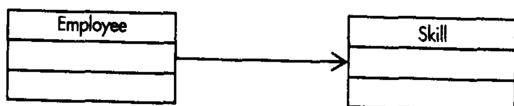


图 3-15 单向关联的例子

3.10 小结

本章介绍了最常用的设计手段，即：

- ▶抽象类
- ▶应用程序编程接口
- ▶模板
- ▶好的设计——原则和度量标准
- ▶全局对象
- ▶确定实现方法
- ▶虚方法
- ▶复制构造函数
- ▶关联的实现

下一章将讨论应该避免的设计手段。

第 4 章　需要避免的设计方案

本章将讨论以下内容：

- 非面向对象的过程对象
- 理解为什么纯粹派不欣赏某些 C++ 结构
- 学习关于委托和方法责任
- 说明有关继承的错误使用

本章涉及一些我认为不适当的设计结构。我称之为“不适当的设计”，是指那些不符合面向对象范例的结构、那些设计未被全面深思熟虑就使用的结构或者采用的捷径。本章中将讨论的不适当的设计结构是：过程对象、委托责任、方法责任、友元结构、多重继承和谬用的继承。每个设计结构将被深入讨论，并联系实例，还会给出替代的方案。

4.1　过程对象

第 1 章中，我给出了一个关于如何将一个结构化系统转为一个面向对象系统的例子。原始的结构化系统表示如图 4-1 所示。

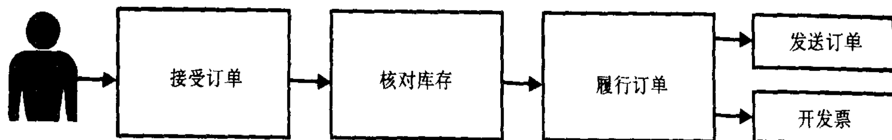


图 4-1　结构化的订单处理系统

而希望的面向对象系统类似于图 4-2。

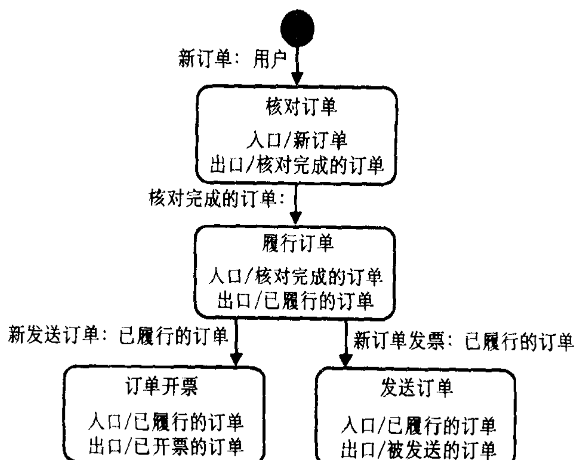


图 4-2　面向对象的订单处理系统

遗憾的是，对原始系统的一般反应是创建一个过程对象。过程对象是以对象的名义体现上述系统中的所有活动的对象。过程对象设计和面向对象设计的主要区别是：过程对象被设计为，以旧过程惯用的方式推动数据在系统中“流动”，而在面向对象设计中，对象对消息做出反应，自身在系统中以预定的方式运行。上述系统的过程对象如图 4-3 所示。

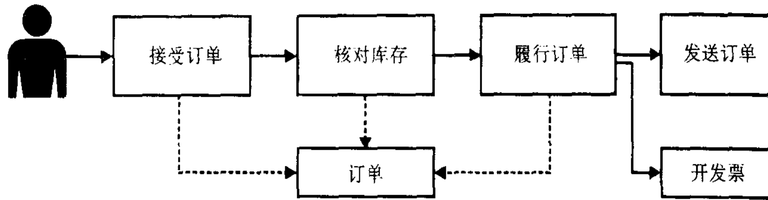


图 4-3 过程对象的订单处理系统

过程对象就是结构化系统的化身，所以也带有最初的结构化设计的所有问题：

▶过程变化时，大量的编码不得不被修改。

▶依据预定的准则使用不同的处理订单的方法（也被称为差异处理，differential processing），效率不太高。

▶增加一种新的订单类型需要进行大量的重编码，甚至要完全重写。

▶order 对象处于‘process’对象的外部控制之下。

问题是，“这些问题能够避免吗?”答案当然是 YES。众所周知，对象有属性和方法。第 1 章引入了对象具有状态的概念。确实，UML v1.4 为此提供了状态图形式的表示法，见第 2 章中的描述。

技巧是通过使对象自身具有相互对话的能力，将这些貌似单纯的对象转为具有目标性的对象。最初的系统遵循“step - by - step”的宗旨，从一个过程到另一个过程的每个转换都以信号为先导。如，从“核对订单”到“正在履行的订单”转换时，需要这样的信号：订单需要的所有货项都在库存中。

要确定预订的货项是否在库存中，需要“核对订单”过程对象向货栈对象（warehouse object）查询这些货项是否存在。一旦所有需要的货项都被在货栈中成功定位，就可以产生一个信号：这个订单可以被履行了，如图 4-4 所示。

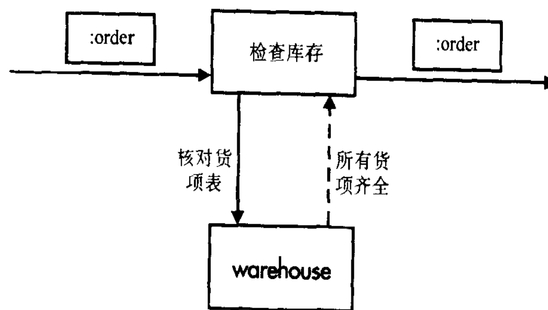


图 4-4 以信号为先导的过程转换

解决方案如下：

(1) 将“step - by - step”的架构转为状态图。

(2) 让"order"对象本身与"warehouse"对象一起检查它的需求。

(3) 对状态图初始化，以便"order"对象通过过程的状态而非过程对象转换。

使用这些解决步骤，将"Check Order"从最初的系统设计转换为状态图和协作图片段，表示如图4-5所示。

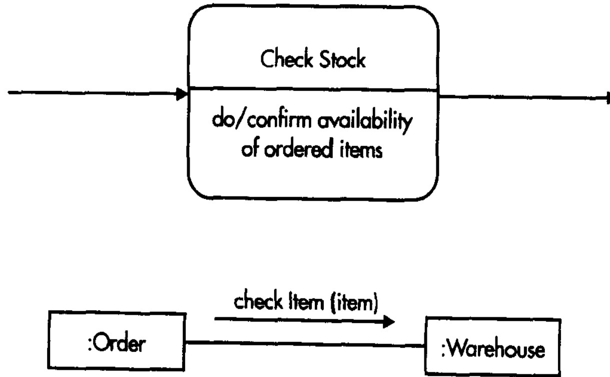


图4-5 "Check Order"的状态图和协作图片段

确定了"order"对象将利用状态图来驾驭自身的命运之后，那么，这样的方法如何解决以前的设计中的问题呢?

4.1.1 过程的变化

遗憾的是，要支持任何过程变化，需要进行一些代码的修改。有一个好消息：代码修改被限制在对象的状态图中，如图4-6所示。任何其他的变化都被限制在需要支持该变化的对象中。

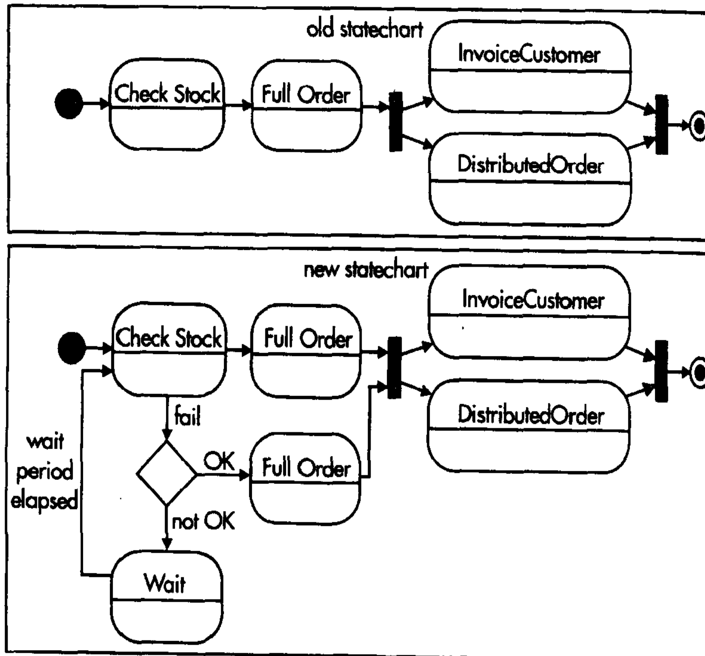


图4-6 过程的变化被反映在状态图中

4.1.2 差异处理

使用状态图允许不同的对象遵从不同的途径，而不需做太大的改变。惟一需要改变代码的是那些需要支持设计变化的对象。表示用新对象“invoice preferred customer”替代通用对象“invoice customer”的状态图如图4-7所示。

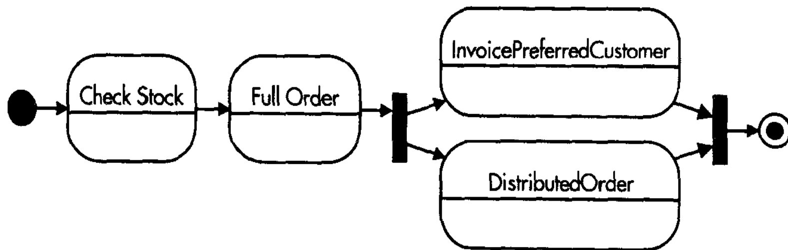


图4-7 状态图显示差异处理

4.1.3 增加新的订单类型

任何新的订单类型都能以最小的代价加入到系统。新订单类型需要从已有的订单类型派生，并被加入到“order”的创建方法中，以便能够创建新增类型的订单对象。

4.1.4 将控制过程放在对象内部

将状态图全部放在内部，订单图“order”表示如图4-8所示。

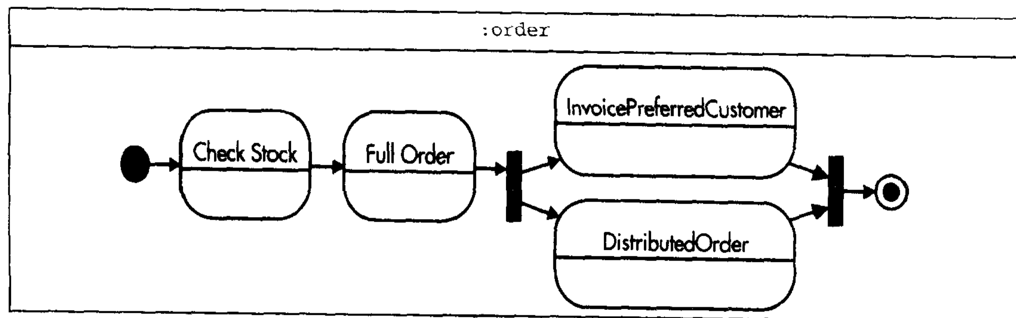


图4-8 将控制过程放在对象内部

4.2 责任的委托

对系统进行分析时，那些支持系统所需要的对象被识别出来。从这些对象，你可以写出类，记载对象将要支持的属性和方法。遗憾的是，只找到对象支持的方法是不够的。

在面向对象设计的过程中，一个老生常谈的错误是，当设计系统时，根据对象被识别的状态来使用对象。像图4-9那样简单地用一个对象代替一个数据项不是一个好的设计。

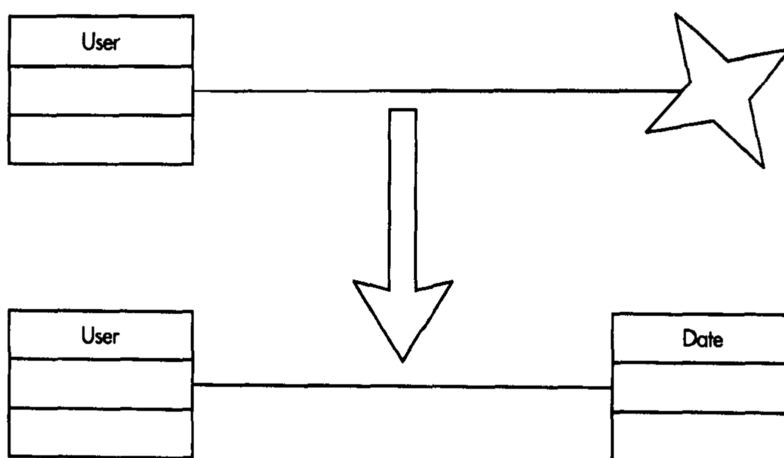


图 4-9 表明数据是如何被转为数据对象的图示

本节讨论委托的使用如何能够改进系统的设计。第一个实例显示了确定某人年龄的一个低效率的方案，第二个实例表现过滤数据的错误方式。这个例子先介绍不好的面向对象的过滤系统是如何设计的，再给出一个高效率的面向对象的设计，最后加上一个最灵活的面向对象的最终设计。

4.2.1 实例 1——确定某人的年龄

正如以前提到的，常见的错误是简单地用一个数据对象取代数据。这样做之后，另一个接踵而来的错误是像处理数据一样处理这些数据对象。

我的意思可以用图 4-10 来表达。对象**: User**向对象**: Person**请求他们的“出生日期”，获得出生日期后，对象**: User**向对象**: Calendar**请求当前日期。有了这两条信息，对象**: User**可以计算出对象**: Person**的年龄。

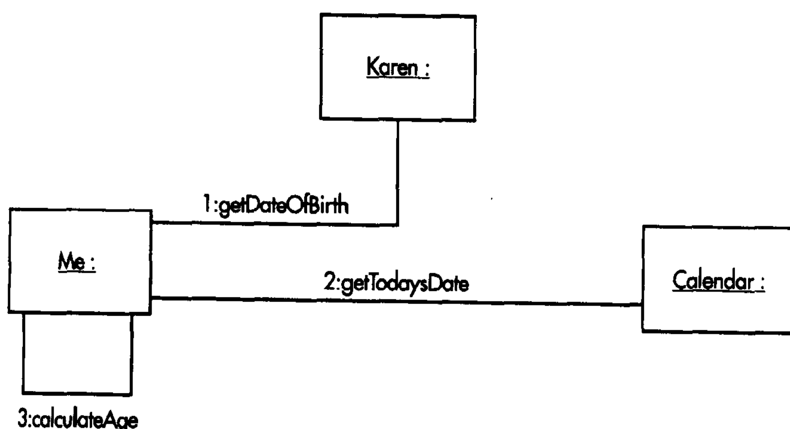


图 4-10 确定年龄的方案一

如果不让“person”去做这项工作，就存在一个实际问题：你必须知道如何处理请求得到的出生日期数据结果：

- 20 March 1963
- March 20 1963
- 20/03/63
- 20 - 03 - 63
- 03/20/63
- 03 - 20 - 63

以上各种数据格式不同但意义相同。另外，从"person"得到的结果也可能与"calendar"要求的参数格式不同。

对这个问题的实际解决方法是避免参与所有的出生日期数据格式的处理，尤其是当所有的需求只是确定 person 的年龄时。要实现这个方法，让对象 **: User**向对象 **: Person**请求年龄，然后再让对象 **: Person**向对象 **: Calendar**请求当前的日期，以便计算他们的年龄，表示如图 4-11 所示。

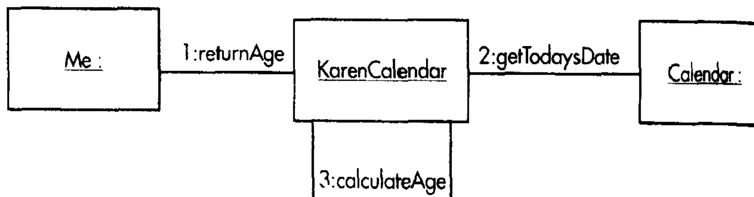


图 4-11　确定年龄的方案二

4.2.2　实例 2——过滤数据

本例说明了三个设计方案，第一个设计表现了以传送数据对象的方式进行数据过滤的方案，第二个设计表现了数据对象激活过滤器的方案，而在第三个设计中，数据对象具有可变的过滤器。

1. 直接传送数据对象

这种设计方式体现了对面向对象设计的一种理解，即认识到一切都是对象；但遗憾的是，这种理解就到此为止了。面向对象的精神不是把一切都转为对象，而是提供一种灵活而可维护的设计。

图 4-12 的设计表示了一个 **: Process**对象将一个 **: Data**对象传递给一个 **: Filter**对象。在面向对象设计中，一般不将一个对象传递到其他对象进行处理。让设计将一个 **: Process**对象传递给一个 **: Data**对象，说明这是一个面向对象出现以前的设计方式。

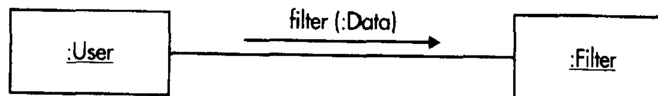


图 4-12　直接传送数据对象

2. 数据对象激活过滤器

对于面向对象设计，对象要发送消息。因此，不是由 **: User**对象向 **: Filter**对象传递一个 **: Data**对象，而是由 **: User**对象向 **: Data**对象发送一个消息进行数据过滤。当这个消息到达 **: Data** 对象时，**: Data**对象决定什么内容被过滤，并使用内部的 **: Filter**对象实施过滤功能，见图 4-13。

图4-13 数据对象激活过滤器

3. 最灵活的过滤系统设计

前一个设计表示了 **:Data** 对象激活 **:Filter** 对象，以实施需要的功能。接下来的设计中，**:Data** 对象可以在运行时决定使用某一个 **:Filter** 对象。这是通过设计一个过滤器接口、然后写一个 **:Filter** 对象实现该接口来达到此目的的。在这种模式下，过滤器对象能够改变，以实现不同的过滤功能，见图4-14。

图4-14 设计一个过滤器接口

4.3 方法责任

上一节讨论了如何使用委托来改进系统的设计，本节进一步延展委托这个主题，将关注的焦点放在如下问题：方法在哪里执行最好？

本节中，我们设计一个典型的系统，让对象可以请求到信息，再根据信息计算出结果。这是一个具有真实意义的例子，能反映出设计为什么是不合理的。

4.3.1 实例1——买一台烤面包炉

你走进一家商场，要求看看正在出售的烤面包炉。然后，你选中了想要的烤面包炉，要求购买。售货员消失了几分钟后，抱着装有烤面包炉的一个盒子回来了。这个盒子里面正是你要买的烤面包炉，你就把它抱回家了。

到家后，你大吃一惊，因为你发现盒子里面只装有烤面包炉的各种部件，还有关于如何安装烤面包炉的说明书。如果你自己装配了烤面包炉而且毫无怨言，那么正好说明你宽恕了现实中的事物，也宽恕了编程世界中的不好的设计方法。

但是，你很可能抱怨。你可能带着这一盒子零件回到商场，要求他们为你装配烤面包炉。但是，与买烤面包炉不同，你发现在你的软件设计中，有一些对象能给你基本的信息，让你实现进一步的任务，而这对你却是可以接受的。为什么呢？

现在，商场售货员处于这样一个位置：他们不只售卖产品，顾客还希望他们能够装配产品。事实上，顾客希望他们能够装配商场出售的所有型号的烤面包炉。另外，商场还需要保证，烤面包炉不仅可以正常工作，还应该安全可靠。售货员和商场现在都要抱怨了。于是成立了一个委员会，来决定谁应该装配烤面包炉。

委员会的最后决定是：设计和制造烤面包炉的公司应该负责装配烤面包炉。这个决定的理由是，公司已经设计了烤面包炉，他们知道如何装配。不止这些，他们还负责烤面包炉的测试，向最终用户提供质量保证。这样能让商场自由地售卖烤面包炉，而无后顾之忧，你也能安

心回家，拥有一台装配完成的、经过测试的、质量有保证的烤面包炉。

随后的例子有助于进一步说明这一点。

4.3.2 实例2——显示运动队的信息

本例进行了多种设计，它是关于一项竞技运动的联盟对象**:League**从运动队对象**:Team**获取信息并发布到一个显示设备的问题。我们先对三种设计逐一描述，再分别对它们进行讨论和评价。(League是由各个运动队组成的联盟，运动队之间每赛季进行联赛，以决定它们在联盟中的名次，如足球联赛。对象**:League**代表联盟，对象**:Team**代表各运动队。——译者注)

设计1

图4-15显示了一个**:League**对象在向显示设备发出信息之前，向**:Team**对象发出多次请求，获取需要的所有信息的情形。

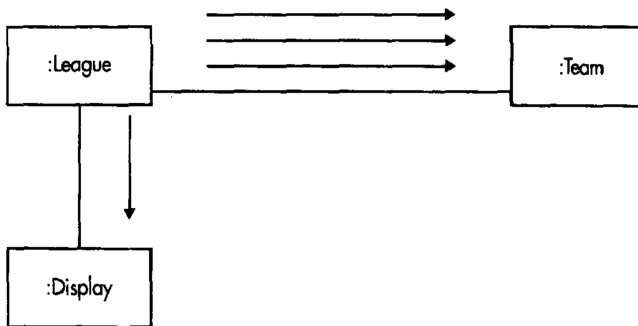


图4-15　**:League**对象负责信息搜集

设计2

图4-16显示，一个**:League**对象只向**:Team**对象发出一次获取信息的请求，将收集信息的责任留给**:Team**对象。**:Team**对象仍需要将收集的信息返回给**:League**对象，以便被传送到显示设备。

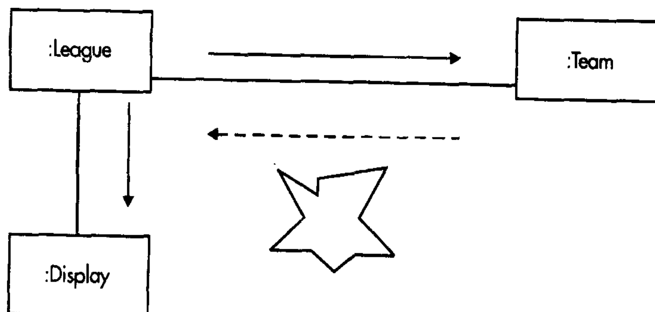


图4-16　**:Team**对象负责信息收集

设计3

最后的设计形式如图4-17所示。这个设计是基于设计2的，它不再将信息从**:Team**对象返回给**:League**对象，而是由**:Team**对象直接向显示设备发送信息。

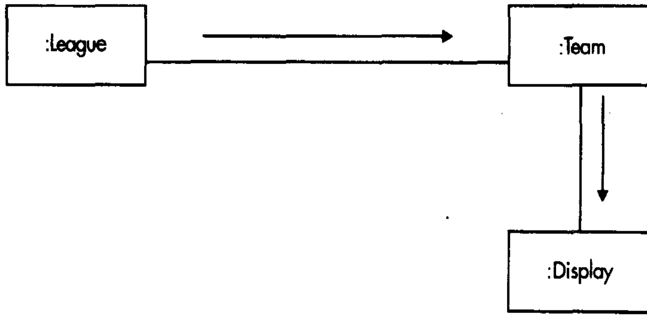


图 4-17 :Team对象直接向显示设备发送信息

设计 1 采用的设计方法比原来的非面向对象设计好不了多少。选择设计 2 还是设计 3 有点不太好决断。

于是，问题就来了，如：

▶ :League对象发布:Team对象的信息时要增加一些它自己的信息（如标识和注释）吗？
▶ 显示每一个:Team对象的信息都用同一个显示设备合理吗？

从这些问题就引出如下的论点：

▶ 使用设计 1，:League对象必须知道:Team对象中存在哪些可被请求的属性。
▶ 如果:Team对象中属性的数量或类型改变了，:League对象也需要被修改。
▶ 设计 3 支持多处理（multi－processing），因为:League对象可以向所有:Team对象广播一个“Display”消息，然后转移到其他事务。
▶ 设计 3 还允许改变 Team 的类型，而不需要随之修改:League对象。如，:League对象可以用一个通用的 Team 类来实现，这样允许它使用从 Team 类派生的类的对象。通过这种方式，:League对象能支持不同类型的 Team。

4.3.3 实例 3——更新联盟中各运动队列表

当:League对象希望更新每个:Team对象时，同样会出现类似于前一个例子的一系列问题。当新的信息可用时，:League对象需要用新信息更新每个:Team对象。

设计 1

:League对象更新每个:Team对象的每个属性，如图 4-18 所示。:League对象需要知道哪个

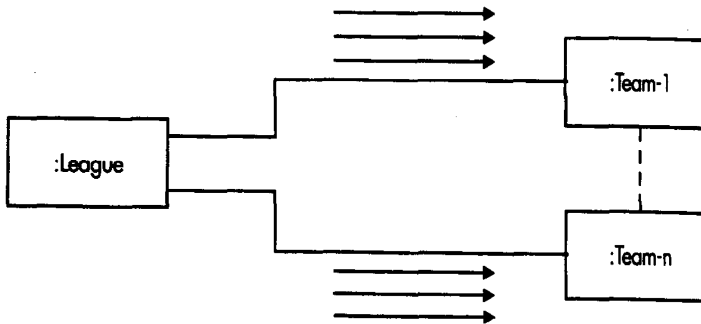


图 4-18 :League对象更新:Team对象的每个属性

属性可以被更新，因为它提供的信息可能超过:Team对象目前可以支持的范围。

设计 2

:League对象将信息打包后用单个消息发送到每个:Team对象，如图 4-19 所示。如果信息过多，每个:Team对象将忽略多余的信息。

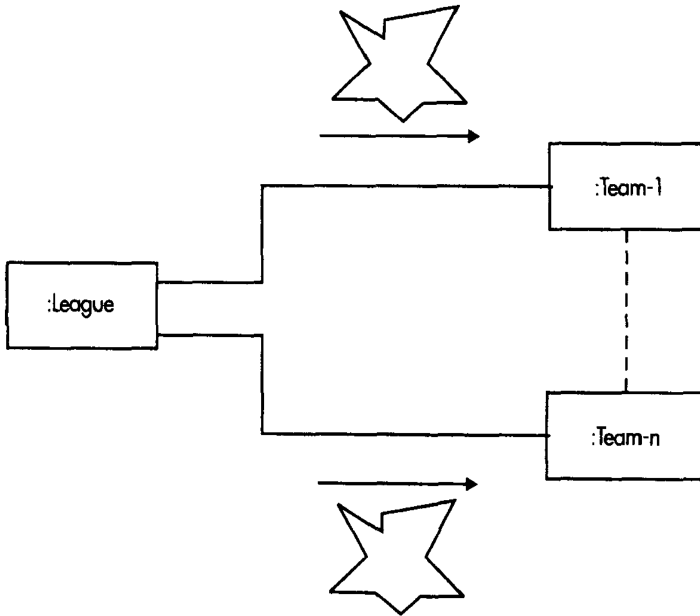


图 4-19 :League对象以单个消息发送信息

4.3.4 实例 4——对联盟中各运动队排序

使用以前的配置，:League对象可以访问一组:Team对象。我们讨论的下一个问题是，:League对象如何根据每个:Team对象内部的某些属性值对各运动队进行排序。

对于这个问题，一般的反应是用下面描述的第一个设计，但是这样做可能是错误的。

设计 1

:League对象发出一系列请求，从联盟运动队列表中相邻的运动队对象获得信息，如图 4-20所示。随后，:League对象比较得到的信息，根据一些内部规则确定两个运动队对象的相对

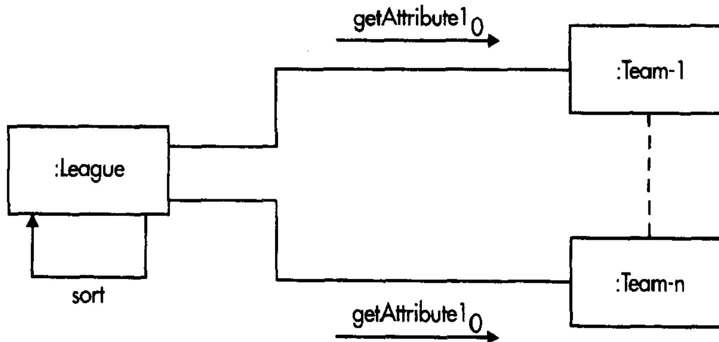


图 4-20 :League对象收集信息以排序

位置，并在联盟运动队列表中改变它们的位置。

但是，**Team**类和**League**类中属性数量和类型的任何变化都会引起必要的修改以反映这些变化。这些变化不只局限于收集到的信息，还会涉及到为区别两个运动队对象进行的测试。

设计2

前一个设计的问题在于出现修改的可能的次数。不仅如此，**:League**对象还失去了支持多种类型运动队的能力，因为每一种运动队类型要求按不同的属性排序。

例如：

- ▶足球队赢得比赛或踢平都可得分，排序就按这样得到的分数进行。
- ▶棒球队也根据它们的得分记录排序，但它们的得分只来自赢得的比赛，因为棒球比赛不会出现平局。

考虑到上述因素，一个排序过程要完成两件事：

(1) 按照一些规则，确定如何分出两个运动队对象的次序，如哪一个队赢得的比赛较多。

(2) 如果比较的结果与目前的运动队的次序不符，要调整运动队的次序。

该问题的解决方案如图4-21所示。第一个**:Team**对象被传给第二个**:Team**对象，然后这第一个**:Team**对象进行一个属性的比较，确定两个队中哪一个次序更靠前。确定了哪个队次序在前之后，第一个**:Team**对象可以返回结果，**:League**对象使用这个结果在联盟运动队列表中调整两个队的次序。

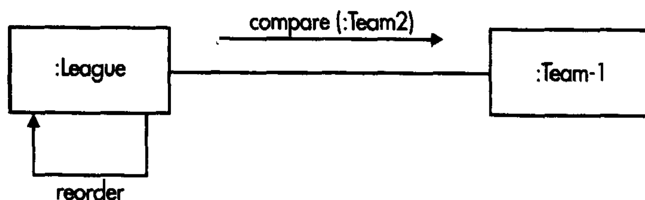


图4-21 由**:Team**对象进行属性比较

例如，如果比较的结果产生这样的事实：第一个**:Team**对象应该放在第二个**:Team**对象之前，则第一个**:Team**对象返回值“TRUE”；如果结果次序相反，则返回“FALSE”。根据返回的结果，**:League**对象就可将运动队在列表中重新排序。

转嫁实施比较责任的优点在于：

- ▶ **:League**对象不必再收集运动队的信息。
- ▶运动队自身就可以决定什么是有效的测试准则。
- ▶对于不同类型的运动队，只要他们具有可供比较的属性，**:League**对象都可以支持。

4.3.5 方法的回顾

本节表明：重要的不仅是由分析决定需要的是哪个方法、属性应放在什么地方，同样重要的还有决定工作的职责在哪里实施。

前几个例子表明，类可以为**:League**对象定义一个方法，如“UpdateEntries”，但实际的属性更新被留给每一个**:Team**对象。**:Team**对象被赋予这样的职责：代表**:League**对象完成具体的

更新操作。

这种机制的最主要的结果是，:League对象只需知道很少关于运动队的信息。这样也就允许:League对象支持多种类型的运动队。这种机制不仅将工作职责放在了最适当的地方，而且提升了对象的灵活性。

4.4　C++中的友元结构

本节首先讨论与所有面向对象编程语言有关的可见层面，即属性和方法，接下来引入友元概念，并讨论友元对这些可见层面的影响方式。

4.4.1　访问级别

每一种面向对象编程语言都提供三个访问属性和方法的级别，这三个级别如下：

private	声明为 private 的所有属性和方法只能被类本身的成员函数使用
protected	声明为 protected 的所有属性和方法只能被类本身和该类的派生类的成员函数使用
public	声明为 public 的所有属性和方法可被任何函数使用

4.4.2　友元是如何影响访问级别的

友元打破了这些访问规则，允许对其他对象的无限制的访问。使用了友元以后，三种访问级别改变如下：

private	所有被声明为 private 的属性和方法只能被类本身的成员函数和声明为该类的友元使用
protected	所有被声明为 protected 的属性和方法只能被类本身和该类的派生类的成员函数、该类的友元、以及该类的派生类的友元使用
public	声明为 public 的所有属性和方法可被任何函数使用

例如，类 **A** 将类 **B** 作为友元，那么，任何:B对象都被允许无限制地访问:A对象的属性和方法。对上面的例子来说，允许特殊访问权的意义是明显的。类 **A** 是一个秘密代理的定义，而类 **B** 是一个代理控制者的类定义。

‘askName’的一个公共的用法只能揭示代理的表面名称，而代理控制者对象能够使用‘realName’方法。不使用友元的解决方案将在第 5 章中讨论。

在 C++中，友元机制的负面作用是它允许的访问超出被保护的级别，甚至达到这样一种程度：拥有友元的类中没有安全的属性和方法。

适用于友元的新的访问规则表示如下：

```
class A
{
   friend class B;
   private:
       int a;
};

class B
{
```

```
    friend class C;
    void function (A* aPtr)
    {
        aPtr->a++; // this will work as B is a friend of A
    }
};

class C
{
    void function (A* aPtr)
    {
        aPtr->a++; // error: C is not a friend of A
        //despite being a friend of a friend
    }
};

class D : public B
{
    void function (A* aPtr)
    {
        p->aPtr++; // error: D is not a friend of A
        // despite being derived from a friend

    }
};
```

以上各类之间的关系如图4-22所示。

由于类**B**是类**A**的一个直接友元，它的函数可以访问类**A**的私有（private）属性“a”。

类**C**是类**B**的一个友元，也就是类**A**的一个间接友元，但一个间接友元不允许访问类**A**的私有属性“a”。

类**D**是从类**B**派生的，因此它是类**A**的一个派生友元。但一个派生友元也不允许访问类**A**的私有属性“a”。

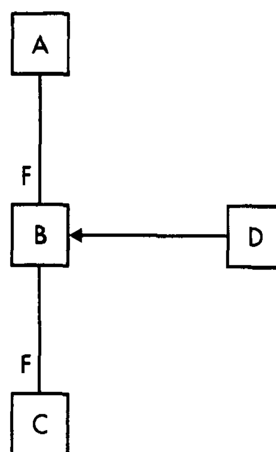


图4-22 类关系的图示

4.4.3 使用友元结构

以下说明了三种使用友元结构的方式。在一个对象类定义中，友元可按以下方式使用：

1. 从一个C语言过程中访问

这种友元结构的表示法是

friend < return type > methodName

任何类定义中包含上述友元声明的类，它的任何属性和方法都可被指定的方法（即methodName）访问。这个方法不属于某个类，而是一个典型的标准C过程，它需要同时访问若干类的内部数据和方法。这种友元机制用于减少数据封装过程的需要，也提高了性能，但打破了面向对象的原则。

例如：

friend void accessNoProblem ()

这个声明可能出现在若干类的声明中。方法 accessNoProblem 对这个类中的一切都有访问权。

2. 从一个类的方法中访问

这种友元结构的表示法是

friend <return type> className:: methodName ()

任何类定义中包含上述友元声明的类，它的任何属性和方法都可被友元声明中指定类的指定方法访问。

如：

friend void seemsOK:: limitedAccess ()

任何 **seemsOK**对象都可通过方法 *limitedAccess* 获得对包含该友元声明的类中的所有属性和方法的访问权。

3. 从一个类中访问

这种友元结构的表示法是

friend class className

任何类定义中包含上述友元声明的类，它的任何属性和方法都可被从指定类（即 className）创建的任何对象的任何方法访问。

如：

friend class realCloseFriend

任何 **realCloseFriend**对象均可通过它的任一方法，获得对包含该友元声明的类的属性和方法的访问权，因为它们被赋予了无限制的访问权。

4.4.4 对友元结构的评价

总之，友元结构用于获得对从约定的类定义创建的对象的属性和方法的附加或特别的访问权。如果给予友元的访问级别与给予派生类或子类的访问级别相同，那么，我认为这种友元机制还有一些价值。但给予的访问级别超出一个对象中被保护的部分，允许访问一个对象的私有部分。我相信这种访问级别应该为非常特别的友元或关系密切的成员保留。

4.5 多重继承

多重继承，正如其名称的意思，是定义一个具有多个父类的类。多重继承用于将多个类的特性组合为一个类。有几种面向对象编程语言不支持多重继承，因而也不被纯粹派认为是真正的面向对象结构。

本节的讨论包括以下内容：

- 为什么设计者认为他们需要使用多重继承
- 与多重继承有关的问题是，内存是如何映射的，以及，当编译器试图确定使用哪个方法时，冲突是怎样出现的

▶设计多重继承的替代方法：设计者试图将哪些东西模型化

多重继承的典型用法表示如图 4-23 所示。

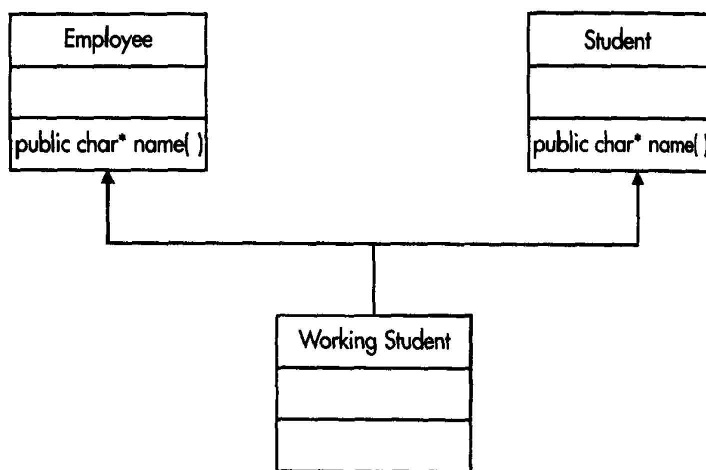


图 4-23 多重继承的典型用法

下面的例子表示了三个类，两个父类和一个子类，子类是从两个父类派生的。三个类的代码如下：

```
class Employee
{
   public:
       virtual char* name ();
   private:
       char* name;
};

class Student
{
   public:
       virtual char* name ();
};

class WorkingStudent  : public Employee,
                        public Student
{
   // this class does not declare it's own name method
};
```

使用前面描述的多重继承，可写出下面的代码：

```
WorkingStudent*    wsPtr = new WorkingStudent;

wsPtr->name ();    // error!
```

这样会引起一个错误，因为两个父类都声明了一个名为 name 的方法，所以，当一个派生

类的:**WorkingStudent**对象试图使用继承方法 name 时，系统不知道选择哪一个 name 方法。

解决方法是显式地指明使用的 name 方法，如：

```
wsPtr->Employee::name ();
wsPtr->Student::name ();
```

4.5.1 从 WorkingStudent 派生一个类

作为一个转折，我们从 **WorkingStudent** 派生一个类，命名为 **DayReleaseStudent**。这个 **DayReleaseStudent** 类的声明中，重定义了 name 方法。新的继承等级结构如图 4-24 所示。

图 4-24　从 WorkingStudent 派生一个类

新类的代码如下：

```
class DayRelease
{
    public:
        virtual char* name ();
};
```

新类及其被重定义的方法的使用如下：

```
DayRelease*    drPtr = new DayRelease;
```

但是，还存在问题：

```
drPtr->Employee::name ();        // works as before
drPtr->Student::name ();         // works as before
drPtr->name ();                  // error! this is still ambiguous
```

激活本对象的内部方法 name 的惟一手段如下：

```
drPtr->DayRelease::name ();
```

4.5.2 重新定义被继承的 name 方法

设计者很有可能希望派生的 **WorkingStudent** 类重新定义继承的两个版本的 name 方法。遗憾的是，这是不可能的，因为一个类只允许有一个名为 name 的方法。对于这个问题的解决方法如图 4-25 所示。

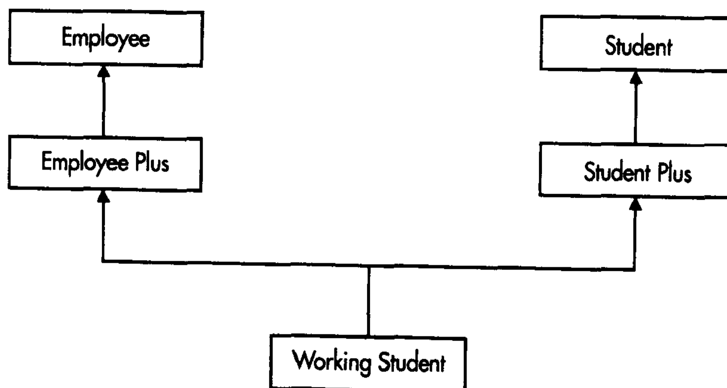


图 4-25 从临时类派生出 **WorkingStudent**

解决方法包括，增加两个临时类，然后从这两个新类派生出类 **WorkingStudent**。

```
class EmployeePlus : public Employee
{
   public:
       virtual char* employeeName () = 0;
       virtual char* name () { return employeeName (); }
};

class StudentPlus : public Student

{
   public:
       virtual char* studentName () = 0;
       virtual char* name () { return studentName (); }
};

class WorkingStudent : public EmployeePlus,
                       public StudentPlus
{
    public:
        virtual char* employeeName ();
        virtual char* studentName ();
};
```

两个新的派生类都重新命名了它们继承的 name 方法。新命名的方法被定义为纯虚方法（Virtual method），这样，派生类 **WorkingStudent** 就被迫提供方法的实现，从而达到方法被重定

义的目的。

要做的最后一件事是将原来的方法的名称与新的方法的名称联系起来，实现的手段是，由原来的方法激活在派生类中实现的方法。

一个使用的例子如下：

```
WorkingStudent*     wsPtr = new WorkingStudent;

Employee*           ePtr = wsPtr;
Student*            sPtr = wsPtr;

ePtr->name ();
// rather than calling the original Employee::name method
// the following will call the redefined method WorkingStudent::employeeName

sPtr->name ();
// rather than calling then original Student::name method
// the following will call the redefined method WorkingStudent::studentName
```

4.5.3 多重继承菱形

多重继承菱形如图 4-26 所示。当两个类从同一个类派生，又被另一个类多重继承时，就会产生菱形的继承结构。图 4-26 中，类 **Employee** 和类 **Student** 都是从类 **Person** 派生的。

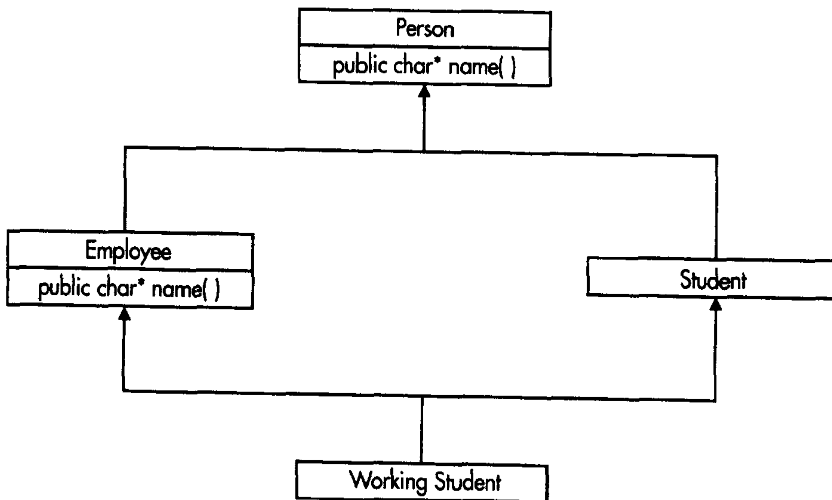


图 4-26 一个基本的多重继承菱形结构

这种菱形结构有一些特殊的性质，我们现在就予以讨论。第一点与内存的使用有关，而第二点又是与派生的方法有关。

1. 映射内存

当一个类从另外一个类派生时，这两个类的定义在内存中共存。这对菱形继承结构来说，会出现一个问题，因为从表面上看，类 **Person** 似乎在内存中存在两份（如图 4-27 所示）。遗憾的是，事实也确实如此：

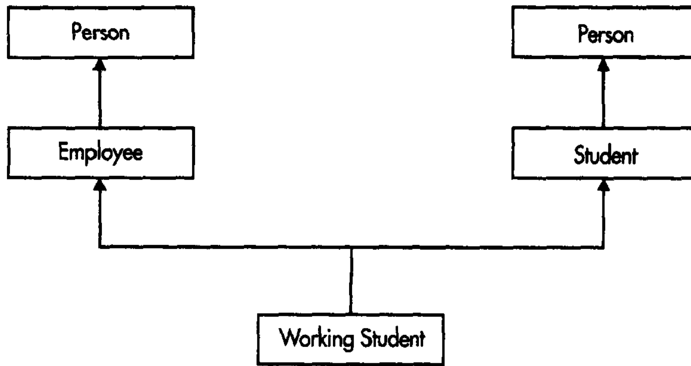


图 4-27 类 Person 在内存中存在两份

对这个内存问题，有一个解决办法，即：让类 **Employee** 和类 **Student** 均将 **Person** 作为虚基类（Virtual base class），从类 **Person** 派生：

```
class Person
{
    public:
        virtual char* name ();
    private:
        char* name;
}

class Employee : virtual public Person
{
...
}

class Student : virtual public Person
{
...
}

class WorkingStudent : public Employee,
                       public Student
{
...
}
```

这种虚基类的机制意味着，对每一个 **:WorkingStudent**对象，内存中只有一份 Person 类存在，而不是上面谈到的两份。

这种机制不足的一面是速度慢，因为为了防止内存的重复，与类 Person 有关的内存是通过指针被访问的，如图 4-28 所示。

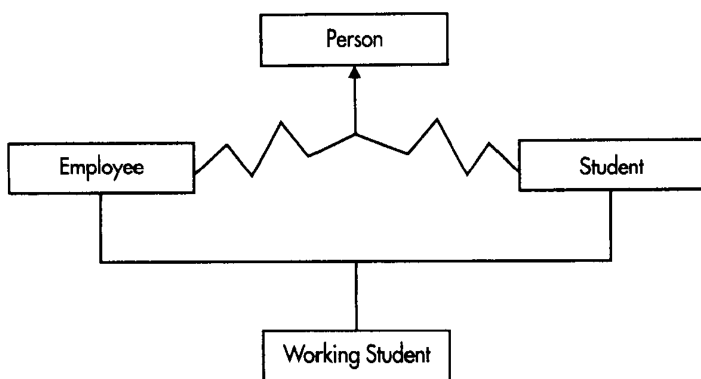


图 4-28 通过指针访问 Person 的内存

2. 被继承的方法

在图 4-26 所示的菱形结构中，假设类 **Person** 中声明了一个名为 name 的方法。现在假设类 **Employee** 重定义了方法 name，而类 **Student** 没有重定义该方法。像前面一样，类 **WorkingStudent** 是从类 **Employee** 和类 **Student** 派生的。

那么，当方法 name 被激活时，会发生什么情况呢？name 方法是直接从类 **Employee** 中继承，还是通过类 **Student** 从基类 **Person** 中继承？

乍一看去，就像以前谈到的方法继承的例子，答案似乎是有二义性的，但是，你错了。

答案比问题本身更严重：它完全依赖于类 **Person** 是怎样被类 **Employee** 和类 **Student** 继承的：

▶如果类 **Person** 是类 **EmployeeOR** 类 **Student** 的非虚基类，则答案是有歧义的。

▶但是，如果类 **PersonBOTH** 是两个类 **EmployeeANDStudent** 的虚基类，那么，答案就非常明确，从类 **Employee** 继承的 name 方法将被使用。来自类 **Employee** 的 name 方法比类 **Person** 中的原始 name 方法具有更高的优先级。

4.5.4 多重继承的替代方法

那么，既然你对多重继承已经彻底幻灭，有没有一个替代方法呢？所幸的是，对多重继承确实有一个替代方法。不仅解决了二义性的问题，而且还使系统的意图更清晰。

下面的例子仍然基于前面的 **WorkingStudent** 的问题。第一个例子是关于一名参加工作的学生，而第二个例子是关于一名参加学习的雇员。

1. 方案 1——工作的学生

类 **WorkingStudent** 是从类 **Student** 派生的，如图 4-29 所示。

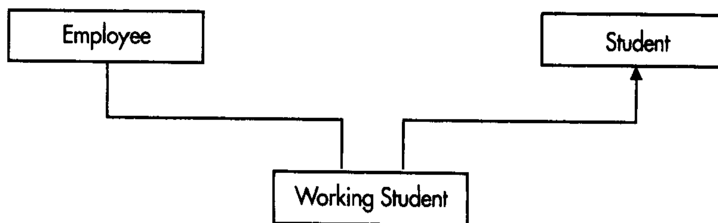


图 4-29 **WorkingStudent** 从类 **Stdent** 派生

除了完成作为学生的任务，一个**:WorkingStudent**还需要完成作为雇员的任务。类 **WorkingStudent** 包含一个**:Employee**对象的引用，该对象是**:Student**对象在开始工作时需要的。类 **WorkingStudent** 对象中将定义方法，以便使用**:Employee**对象的引用激活其方法。

例子：**:WorkingStudent**。*WhenIsPayDay*（）被用于向**:Employee**对象传递需要处理的要求。

分析阶段将决定类 **Employee** 中被类 **WorkingStudent** 支持的方法的数量，也就是说，一个**:WorkingStudent**对象不可能被问及养老金的安排或他们对公司小汽车的选择。

2. 方案 2——在学的雇员

StudyingEmployee 类是从 Employee 类派生的（见图 4-30）。

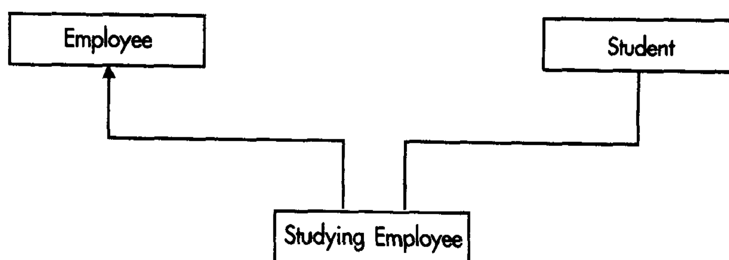


图 4-30　**StudyingEmployee** 类从 **Employee** 类派生

除了能够完成作为雇员的任务，一个**:StudyingEmployee**对象还需要完成作为学生的任务。类 **StudyingEmployee** 包含一个**:Student**对象的引用，该对象是**:StudyingEmployee**对象开始学习时需要的。类 **StudyingEmployee** 定义了方法，以便使用**:Student**对象的引用激活其方法。

例子：**:StudyingEmployee**。WhenDoesSchoolStart（）被用于向**:Student**对象传递需要处理的要求。

4.6 继承的不当使用

下面显示的例子（见图 4-31）是一个继承等级结构，但它背后有一个逻辑问题，因为两个派生的类型都是数据类型。

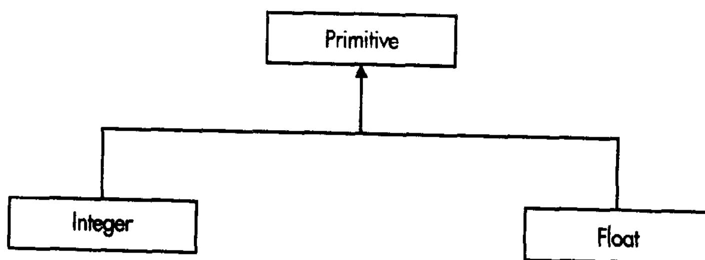


图 4-31　一个继承等级结构

在任何类 **Primitive** 的实例可被使用的地方，根据继承的概念，派生类的实例也可被使用。实际上，在这些派生类中，存在一个逻辑上的能力，即：它们可以互相替换。

但是，这里也存在一个问题，甚至一些身价颇高的顾问也掉进去了。

正如以前提到的，有了一个可互换组件的继承等级结构，可降低代码开发的成本，因为它

们共享常用的代码。如，可以设计接口，只有一个方法去执行每个任务。这些方法将从类**Primitive**或它的派生类创建的对象作为参数。接口的代码如下：

```
class Interface
{
    public:
        void method1 (Primitive p);
        void method2 (Primitive p1, Primitive p2);
}
```

现在请注意，当继承结构扩展为如图4-32所示时，会发生什么现象。

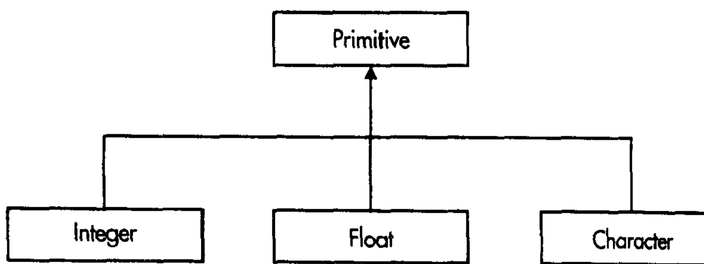


图4-32　经扩展的继承等级结构

与前一个例子不同，这个继承等级结构不能按照前一个例子的方式提供可变换的组件。你可能问这样的问题："如果method1的一个参数是一个 **Character**对象，它被执行时会发生什么情况呢?"

答案应该很简单。遗憾的是，这个问题与顾问让项目的底层开发小组如何实现对新的派生类型的支持有关。开发小组还没有被允许实现一个正确的设计，相反，顾问告诉他们要对他们已经定义的接口做一些调整。

他们被迫进行的修改是采用运行时检查的形式（见图4-33）。

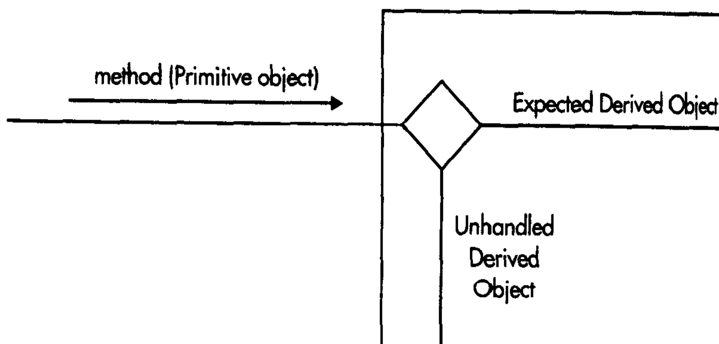


图4-33　采用以运行时检查的形式

感觉到应验了这个似乎容易的指示，顾问将继承等级结构扩展，包括进了很多不同的类型（见图4-34）。

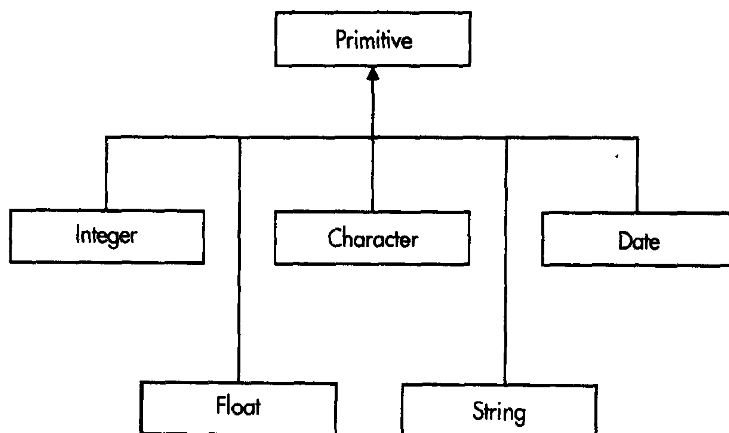


图 4-34　继续扩展的继承等级结构

虽然底层开发小组尽了最大努力，错误还是发生了。在某一点，我们花费数天时间试图使用这些底层类去实现一个特有的功能，但几乎毫无进展，只好谦卑地寻求帮助。经过一番调查研究之后，我们被告知问题出在我们的编程错误上，用行话说就是："你使用类的方法不正确。"

几分钟的瞠目结舌之后，我们中的一位总算回过神来，能问出一个简单问题了："如果这是一个公开的接口，我们对它的使用方法都可汇编成书，怎么会是我们在不正确地使用它呢?"即刻的回应就是："在向这些方法传递参数时，你只能传递它们支持的类型。"

正是这一点使我们认识到。虽然底层开发小组是 C++ 的专家，但它不是一个有幸具有面向对象技巧的小组。我们对他们的多次提醒都没有什么效果。一个月之内，我们中的大多数人都离开了这个项目，因为我们都能够看到，这个项目的问题将变得更加严重。

实际的解决方法

尽管传递的参数可能会有悖于继承等级，但实际的问题在于接口和顾问对于最大代码复用率的渴求，因此，方法就尽可能通用，并使用基于类 **Primitive** 的参数。

然而，稍微思考一下并进行一点小小的努力，就可以使接口正确地工作。解决方法就是将参数的类型检查任务从运行提前至编译。毕竟，编译器是非常善于确定方法在调用时是否使用了正确的参数类型。接口可能被编码如下：

```
class Interface
{
    public:
        void method1 (Integer i) { oldMethod1 (i); }
        void method1 (Character c) { oldMethod1 (c); }

        void method2 (Integer i, Float f)
                     { oldMethod2 (i, f); }
        void method2 (Integer i, String str)
                     { oldMethod2 (i, str); }

    private:
        void oldMethod1 (Primitive p);
```

```
        void oldMethod2 (Primitive p1, Primitive p2);
};
```

使用这个新的接口，它永不会被用错，因为共用的方法支持特殊类型而不支持通用的类 **Primitive**，如下所示：

```
Integer i = 0;
Character c = 'a';
Float fl = 2.1;
Interface* If = new Interface;

If->method1 (i)         // this will compile
If->method1 (fl)        // this will not compile

If->method2 (i, fl)     // this will compile
If->method2 (c, fl)     // this will not compile
```

现在，由编译器来确定参数类型是否与调用的某特定方法相符合，而不是等到程序运行时再看参数检查是否正确。

另外，与以前不同，对类 **Primitive** 继承等级结构的进一步扩展都对接口设计没有影响，除非再增加适当的方法。

4.7 小结

本章讨论了如下的设计结构：

▶过程对象

▶委托责任

▶方法责任

▶友元

▶多重继承

▶错用的继承

下一章将讨论高级设计的主题。

第 5 章　高级设计技术

本章将讨论以下内容：

- ▶ 学习如何使用高级应用程序编程接口
- ▶ 了解如何在一个应用程序中使用线程
- ▶ 学习 Model/View/Controller 设计方法
- ▶ 学习如何更多地暴露已定义的接口
- ▶ 了解引用计数及其使用方法

本章将设计方法引导到一个新的层次。本章所讨论的方法不会在每一个项目中都被使用，但对于那些比较复杂的项目，它们的作用是无价的。本章讨论的设计方法如下：

- ▶ 高级应用程序编程接口（API），它是标准 API 结构的一个扩展。
- ▶ 线程，用于设计同时做多件事情的应用程序。
- ▶ Model/View/Controller，Model 是应用程序的驱动核心，Controller 指引 Model 的方向，View 用于监视 Model 中的变化。
- ▶ 暴露接口，这种方法允许设计者以分立的步骤暴露类的接口。
- ▶ 引用计数，这种结构用于避免由于表层复制构造函数引起的内存问题。当拷贝被多个任务共享时，就会出现内存问题。在第 4 章中曾引入了表层复制构造函数的定义。

5.1　高级 API 结构

API 结构允许写出一个支持特殊接口的类，而这个接口是若干特征类似的类支持的。下面的图示为第 3 章中讨论的 API 结构使用的表示法的翻版。

一般来说，这个结构还是挺有效的，除非基于接口类的对象之间相互转换。例如，下面的图 5-1 表达了一个对象——人，它是为了代表婴儿（Baby）而被创建的。当婴儿长大时，这个

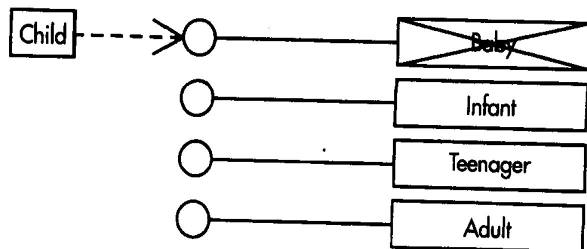
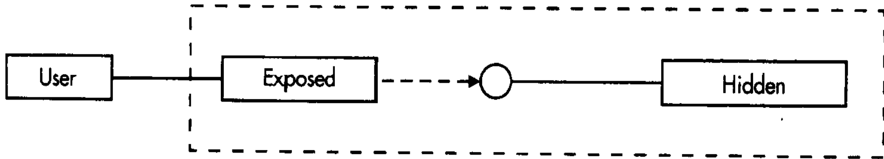


图 5-1　对象转换的例子

对象被一个幼儿（Infant）对象代替，然后是一个少年对象（Teenager），最后是一个成人对象（Adult）。在每一次转换时，都需要创建一个新对象，去代替已有的对象。任何对于以前对象的引用（如指向这些对象的指针）都需要被更新，以便使用新的对象。如果无需更新对于对象的引用，就可实现一个人的年龄阶段转换，那将是比较有利的。

5.1.1 什么是高级 API 结构

高级 API 结构的不同之处在于，它提供一个中心引用，来隐藏潜在的变化，下面的插图表明了这种新结构的表示法。



虽然 API 方法允许相似的对象之间相互转换，但它强加了一顶“帽子”。在下面的 API 例子中，如果一个可变的对象被另一个对象代替，**:User**对象将做这项工作。被**:User**对象持有的引用需要被设置，以使用新的可变的对象。如果一个人的年龄阶段的转换不干扰外部对象就可实现，这将是有利的。为了达此目的，你可以增加另外一层抽象。本例的图示如图 5-2 所示。

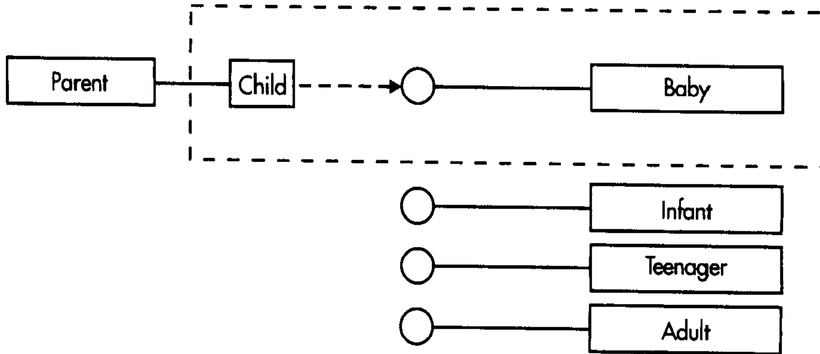


图 5-2　增加另外一层抽象

这个新的层隔离了对象的变化，不再涉及到外部的对象。当这个**:User**对象被其他**:User**对象结合时，这种隔离尤其重要。使用 API 结构，当对象经历一个从婴儿到幼儿的转变时，每个外部对象都需要被涉及到。每个对该对象的引用都必须改变。

使用高级 API 结构，当与接口相关的对象改变时，每个外部对象维护它自身对于可改变的对象的引用。这种结构的优点是：它在减少变化的影响的同时，能够使系统“生长”。

这种结构的缺点是：所有被隐藏的类定义具有不同的接口。与类 **Child** 和类 **Adult** 的定义相比，类 **Baby** 定义了一个更有限的方法集。这意味着，实现的接口不能支持所有这些类。

5.1.2 如何克服缺点

如果你让所有被隐藏的类的定义支持一个特殊接口时，这种方案的缺点就能够被克服。要解决定义不同接口的派生类定义的问题，可使用下面两种解决方法之一：

▶ 在被暴露的类和接口类定义中，尽量写出每一个方法。显然，这不是一个有价值的解决方法，因为，为了跟踪新的方法，被暴露的类和接口类定义都需要被更新。不仅如此，如果一个方法要被另一个派生类定义支持，每个被隐藏的类定义都需要提供方法的实现。

```
baby:changeNappy ()
adult:changeNappy ()

baby:whereDoYouWork ()
adult:whereDoYouWork ()
```

▶ 另一个替代方法是提供一个由大多数或所有派生类支持的方法组成的有限子集。被暴露的类和接口类定义也支持这些方法。

目前的问题是如何支持其余的方法。解决方式是增加一个支持一个关键字和一些参数的方法：

```
answer = aPerson:question ( "changeNappy" );
answer = aPerson:question ( "whereDoYouWork" );
```

aPerson:对象（**PersonEnvelope**类定义）将question传递到被封入的派生类对象（Baby，Child或Adult)，所有这些对象都重载了定义在**PersonLetter**类定义中的方法question。如果某个派生类对象不支持question，则会给出一个“NotSupported”的答复，否则，就会返回恰当的答复。

5.2 线程

线程允许一个应用程序在继续执行主任务的同时执行第二个任务。使用线程的一个优点是，应用程序能更好地利用多CPU计算机的潜能，因为应用程序的每一个线程都能够在它本身的CPU上执行。

使用线程的应用程序软件的一个例子是“哲学家”（Philosophers)。哲学家应用程序的工作模型描述如下：

“有10位哲学家，他们的生活有两部分：吃饭和思考。在一个圆桌旁，每位哲学家都有自己的位置，圆桌的中央是一个盛米饭的大碗。吃米饭需要两支（一双）筷子，但提供的筷子只有10支（5双)，即平均每两位哲学家有一双筷子。哲学家只能拿起紧邻他左边和右边的筷子。所有哲学家在身体结构方面是等同的，即吃饭，然后思考，两者交替进行。”

以上问题的一个解决方案是，针对一位哲学家写代码。代码完成后，下一步是创建若干线程，每个线程都运行已完成的哲学家代码。关于哲学家之间共享筷子的问题将在下一节讨论。

从本质上讲，线程有两种形式：

▶ **分离的（Detached）** 指一个线程独立于主程序而运行。在这个线程完成之前，主程序可能退出。这种形式的线程的表示法如图5-3所示。

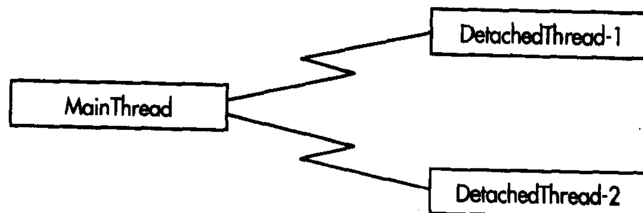


图5-3 分离的线程表示法

▶ **结合的（Joinable）** 当你希望主程序能够等待线程完成时，可以使用这种风格的线程。

这种形式的线程的表示法如图5-4所示。

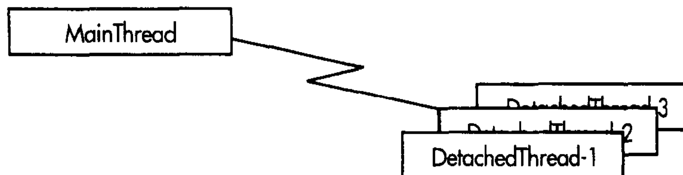


图5-4 结合的线程表示法

5.2.1 资源同步

线程很可能共享资源，因此它们也很有可能同时试图使用这些资源。这被称为竞赛条件（Race Condition），因为每个线程都试图在另一个线程之前访问资源。为防止这种情况的发生，代码可以被同步，在同一时间只允许一个线程访问资源。

在C++中，这种同步是通过使用相互排斥锁（mutually exclusive lock或*mutex*）实现的。只有一个线程可以持有排斥锁，不允许其他线程持有。在Java中，这种同步是通过使用关键字“Synchronized”实现的。两种语言的实现方式如下。

1. C++中的同步

```
lock_mutex (named_mutex)
    synchronized code
unlock_mutex (named_mutex)
```

2. Java中的同步

```
synchronized (lock variable)
{
    synchronized code
}
```

可是，有时候，能够掌握某个资源的控制权，并维持一段延长的时间显得比较重要。在这种情况下，获取排斥锁（mutex）的代码与释放排斥锁的代码相距较远。C++支持这种长时间的资源控制，而实质上，具体的实现方式与以前相似。

```
// inside first method
lock_mutex (named_mutex)
// inside second method
unlock_mutex (named_mutex)
```

遗憾的是，Java不支持这种机制，因为“同步”方案被限制只能在单个方法之内有效。

5.2.2 Java同步的问题

在C++的实现中，如果因为排斥锁已经被锁定而引起获取排斥锁失败，则调用排斥锁的

线程将停止处理，等待排斥锁被释放。为了在Java中支持全面的资源同步功能，有必要提供一种阻塞机制（Blocking mechanism），它类似于C++中的排斥锁机制。下面是两套使用方法的实例。第一套实例中，每一个排斥锁就是一个布尔变量，而第二套实例则使用一个排斥锁链表。

第一套使用方法——每个排斥锁使用单一的布尔变量

对于前面的哲学家的问题，筷子就可以用一个布尔变量数组实现：

```
private static Boolean    chopsticks = new Boolean [<number of philosophers>];
```

下面的方法是请求排斥锁的方法：

```
public void Psem (int pos)
{
   boolean    flag;

   System.out.println (threadName + " RequestLock");
```

代码被同步（一次一个线程）：

```
synchronized (Dinner.chopstickLock [pos])
{
```

排斥锁的当前值被形成记录：

```
flag = Dinner.chopstickLock [pos].booleanValue ();
```

如果结果是‘false’（排斥锁是可用的），将数值设为‘true’（获得排斥锁），然后返回：

```
if (flag == false)
{
    Dinner.chopstickLock [pos] = new Boolean (true);
    System.out.println (threadName + " AcquiredLock");
    return;
    }
}
```

否则，如果标志flag仍然为‘true’（排斥锁被其他人持有）：

```
while (flag == true)
{
```

进行代码同步：

```
synchronized (Dinner.chopstickLock [pos])
{
```

并记录排斥锁的值：

```
flag = Dinner.chopstickLock [pos].booleanValue ();
```

若结果为‘false’（排斥锁可用），设置其值为‘true’（获取），然后返回

```
        if (flag == false)
        {
            Dinner.chopstickLock [pos] = new Boolean (true);
            System.out.println (threadName + " AcquiredLock");
            return;
        }
    }
```

否则，等待0.2秒，并重复

```
                try {wait (200);}
                catch (InterruptedException e) {}
        }
    }
```

下面这个方法是释放排斥锁的方法：

```
public void Vsem (int pos)
{
    System.out.println (threadName + " Vsem " + pos);
```

代码被同步（一次一个线程）：

```
synchronized (Dinner.chopstickLock [pos])
{
```

布尔值被设为'false'（释放）：

```
        Dinner.chopstickLock [pos] = new Boolean (false);
    }
}
```

第二套使用方法——每个排斥锁使用一个链表

lockList是使用一个Vector实现的，一个Vector就是一个对象数组，它提供索引机制，并在必要时动态更新数组的大小。

```
private static Vector    lockList = new Vector ();
```

下面的方法用于请求排斥锁，如果该方法失败了，它使得线程阻塞，直到线程获得排斥锁，或者线程被唤醒（Unblocked）。

```
public void RequestLock ()
{
   Boolean    empty;
```

首先，对lockList的使用进行同步，以保证只有一个线程拥有访问权；然后，设置一个标志，表明链表是否为空；最后，将线程本身加入链表——这是重要的一步。

```
synchronized (lockList)
{
    empty = lockList.isEmpty ();
    lockList.addElement (this);
}
```

标志 empty 反映等待排斥锁的线程的数量:

- 如果 'empty == false',表明已经有一个线程在使用排斥锁,本线程必须等待。
- 如果 'empty ! = false',那么排斥锁是可用的,函数立即返回。
- 如果当线程获得排斥锁时,它还没有将自身加入链表,另外一个线程将看到一个空的链表,它将不再等待。我们可使用第二个标志表示一个被锁定的排斥锁,但以将线程加入链表这种方式也一样能很好地达到目的。

```
if (empty == false)
{
    try {this.wait ();}
    catch (InterruptedException e) {}
}
```

如果链表是空的,或本线程已经被唤醒,本方法退出。唤醒线程可使用方法 ReleaseLock:

```
}
```

当前活动的线程想释放排斥锁时,就需调用下面的方法,该方法找到第一个正在等待的线程(如果有的话),然后唤醒它。

```
public void ReleaseLock ()
{
    Thread    peek;
```

对 lockList 的使用进行同步,以保证只有一个线程拥有访问权:

```
synchronized (lockList)
{
```

从链表中清除线程本身,在链表中只保留等待的线程:

```
lockList.removeElement (this);
```

如果链表不为空,表明还有正在等待使用排斥锁的线程:

```
if (lockList.isEmpty () == false)
{
    try
    {
```

得到链表中的第一个线程:

```
        peek = (Thread)lockList.firstElement ();
```

同步这个线程,以保证没有其他元素试图使用它:

```
            synchronized (peek)
            {
```

唤醒这个线程：

```
                peek.notify ();
            }
        }
        catch (NoSuchElementException e) {}
    }
  }
}
```

这些方法可用于提供对资源的锁定，并提供C++中支持的阻塞特性。下面是这些方法的一个应用：

```
public void run()
{
    System.out.println ("Starting " + threadName);
    RequestLock ();
    System.out.println ("Doing something " + threadName);
    ReleaseLock ();
}
```

5.2.3　资源的死锁

当两个或更多线程已经取走其他线程需要继续使用的资源排斥锁时，死锁（deadlock）就出现了。例如，有两个线程，要完成各自的活动，每个线程都需要两个排斥锁。如果第一个线程获得了第一个排斥锁，而第二个线程获得了第二个排斥锁，就导致系统进入死锁状态，因为这两个线程都无法得到两个排斥锁，无法继续运行。

线程的经典例子“哲学家”不仅用于展示线程的工作方式，还是一个关于死锁的例子。“哲学家”程序背后的假设是，若干哲学家走到一个圆桌旁并坐下来用餐。

每位哲学家想要用餐，都必须拿到一双筷子（两支），如图5-5所示，其中一支筷子与他左边相邻的哲学家共享，另外一支筷子与他右边相邻的哲学家共享。一位哲学家吃完一口饭后，为避免其他哲学家饥饿，他放下一双筷子，思考一段时间，等待再次拿到一双筷子，吃下一口。当每一位哲学家都拿起他右边的一支筷子，使得与他右边相邻的哲学家拿不到左手边的一支筷子时，就出现死锁了（见图5-6）。

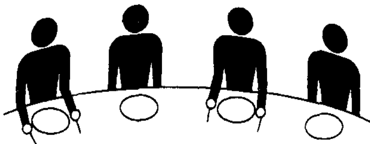


图5-5　哲学家用餐的前提

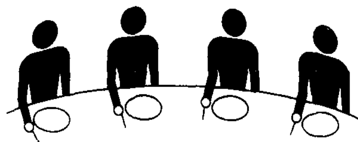


图5-6　死锁出现了

“哲学家”程序的典型的实现依赖于在圆桌旁坐下并依次开始工作的哲学家，因为第一位哲学家总比最后一位哲学家先走到圆桌旁。

这种理论可以正常工作，除非出现了一个随机因素。哲学家两次用餐之间用于思考的时间片断可能不同，或者如果他们没有拿到某一支筷子，用于思考的时间可能又不同了。

试图锁定一个排斥锁

那么，怎样解决上述问题呢？C++有一种机制，允许一个线程尝试并锁定一个排斥锁，或者说，针对上面的例子，尝试并拿到一支筷子。这当然很好，但它的作用在哪里呢？如果某位哲学家拿到了他右边的一支筷子，他就尽力去拿左边的那支筷子。如果他拿不到左边的那支筷子，控制权被返回到这位哲学家线程，慷慨大方的他可以释放已经拿到的右边的筷子，这样就允许他右边的哲学家有机会用餐。

```
// The argument passed in, is the position of the philosopher at the table
public int Psem_try (int pos)
{
```

首先，在特定的地点同步筷子的使用。如果某支筷子在桌子上（booleanValue为false），则拿起这支筷子，将booleanValue置为true，并返回数值0——表明得到这支筷子。否则，返回数值1——表明筷子正在被使用，哲学家必须等待：

```
synchronized (Dinner.chopstickLock [pos])
{
    if (Dinner.chopstickLock [pos].booleanValue () == false)
    {
        Dinner.chopstickLock [pos] = new Boolean (true);
        System.out.println (threadName + " AcquiredLock");
        return 0;
    }
    else
        {
            System.out.println (threadName + " WaitingForLock");
            return 1;
        }
    }
}
```

现在，这些方法可用于锁定资源，并同时支持防止死锁的功能。如下的代码体现了它们的用法：

```
// Each philosopher is tasked taking 5 bites from the food in front
of them for (int i = 0; i < 5; i++)
{
```

哲学家获取右手边的筷子：

```
Psem (rightChopstick);
```

如果使用方法‘Psem_try’获取左边的筷子的请求被拒绝，则释放右边的筷子。沉思20

毫秒，再获取右边的筷子：

```
while (Psem_try (leftChopstick) == 1)
{
    Vsem (rightChopstick);
    try {sleep (20);}
    catch (InterruptedException e) {}
    Psem (rightChopstick);
}
```

两支筷子都获得后，就可用餐了。用餐时间是20毫秒，然后释放两支筷子：

```
System.out.println ("All chopsticks acquired " + threadName);
try {sleep (20);}
catch (InterruptedException e) {}
Vsem (leftChopstick);
Vsem (rightChopstick);
```

沉思100毫秒后，再准备下一次用餐：

```
    try {sleep (100);}
    catch (InterruptedException e) {}
}
```

使用其他的方法套件也会出现死锁。当测试我写的实例学习2的Java代码（即第11章的多线程机场管理应用程序）时，我注意到虽然一架飞机刚离开了机场，并释放了跑道排斥锁，另一架等待着的飞机并没有降落，还有一架等待着的飞机也没有起飞。发生这种情况的原因是另外一种形式的死锁——飞机都不知道那条跑道已经被释放。仔细检查代码可能使两架飞机迅速遍历等待链表，并确定链表是不空的。离开机场的飞机释放跑道排斥锁并试图唤醒等待的飞机，问题是两架等待的飞机都没有将自身加入等待链表，因此，释放跑道的信息无法通知到这两架飞机。这样，请求跑道的两架飞机陷入睡眠状态，而且永远都无法被唤醒。

如下是需要对‘RequestLock’的代码所做的修改：

```
while (empty == false)
{
```

以前，线程的等待时间是不确定的，现在让它等待2秒钟：

```
try {plane.sleep (2000);}
catch (InterruptedException e) {}
```

当线程醒来时，检查等待链表中的第一个元素：

```
try
{
    peek = (Plane)lockList.firstElement ();
```

如果第一个元素是线程本身，标志置为true，并退出While循环：

```
        if (peek == plane)
            empty = true;
    }
    catch (NoSuchElementException e) {}
}
```

5.3 Model/View/Controller 机制

Model/View/Controller（MVC）是一种机制，其中:Model对象能够改变自己的状态，让其他:View对象监视这种状态的变化，以执行一个动作。状态变化的发生独立于:View对象。:Model对象也无需知道自己正在被监视。反映MVC模型本质的一个例子是图5-7中显示的广播电台。电台是:Model对象，而收听电台节目的人（Listener）是:View对象。电台不知道谁在收听节目，这正是:Model对象的运作方式。:Controller对象是音乐、新闻或天气预报的节目制作者（Producer）。对于:Model对象，惟一的原则是，当状态发生变化时，它应该产生事件消息。:View对象感兴趣的正是这些消息。

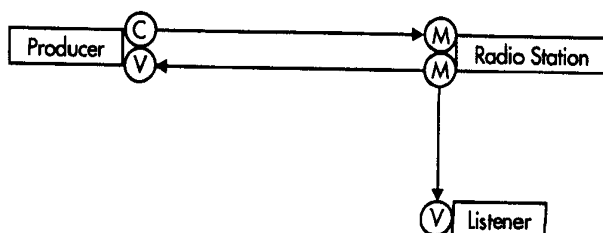


图5-7 MVC图使用的表示法

在图5-8中，:MyStreet对象是设计用于模拟一条当地街道的。模拟的时间段在00:01到23:59之间，出现的一些重要事件是‘日出（SunRise）’和‘日落（SunSet）’。每当这些事件出现时，:MyStreet对象就“广播”一条对应的事件消息。

正如下面的描述所示，:StreetLight对象和:PostalWorker对象已经注册过，对特定的消息感兴趣。:StreetLight对象在收到‘SunRise’事件消息时自动关闭，而在收到‘SunSet’事件消息时自动打开。:PostalWorker对象在收到‘SunRise’事件消息时就开始递送邮件。

在图5-8所示的例子中，可能还有另外一个:View对象，如，一个:HomeOwner对象。它不监视:MyStreet对象是否发出事件消息，而是监视:PostalWorker对象是否发出MailDelivered（邮件已经送到）事件消息，以便及时收集新到达的邮件。这种情况下，:PostalWorker对象将担当两种角色，对:MyStreet对象来说，它是一个View，而对:HomeOwner对象来说，它是一个Model。

:Controller对象向:Model对象发送控制消息。对于模拟当地街道的例子，典型的控制消息如下：

- StopSimulation
- StartSimulation
- SunRise
- SunSet
- ResetSimulation

▶ StartSimulationFromTime

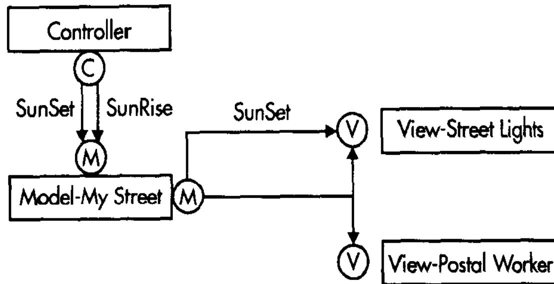


图 5-8 MyStreet 的 MVC 模型

另外，作为 Model 的**:MyStreet**对象可以产生新的事件消息:Reset 和 TimeNow，**:View**对象可使用这些消息校正自己的状态。

若从白天的中午开始模拟过程，就意味着**:StreetLight**对象需要始于“关闭”的状态，这样，它就需要注册 TimeNow 事件，以便对该事件消息做出反应。

正如**:PostalWorker**对象担当两种角色一样，Controller 可以注册为**:MyStreet**对象的一个 View。这样，Controller 能够保证 Model 功能正确。

5.3.1 中心 MVC Controller 方案

这是第一个方案，它基于一个中心 MVC controller。这个中心 MVC controller 保存若干相关信息的链表。中心 MVC controller 保存的链表有如下几种：

▶ 控制 model 中的方法的控制消息的链表

▶ model 的事件消息链表

▶ view 对象和 model 事件消息的链表

链表 1——控制消息

这个链表包含那些将 controller 对象消息与 model 对象和对消息做出反应的 model 对象的方法联系起来的项目，表示如下：

model 通过对象注册，表明将处理某些控制对象的消息。为了向链表中加入或从链表中删除某些项目，中心 MVC controller 支持如下的方法：

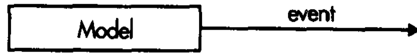
```
ControlRegister (self, controlMessage, model, model_method)
ControlUnregister (self, controlMessage, model)
```

当 controller 对象要对 model 对象中的方法起作用时，它产生特定的事件。要产生一个事件，需在中心 MVC controller 端调用一个方法，该方法确定是否有注册的 model 对象，然后激活特定的 model 对象的方法。中心 MVC controller 的方法如下：

```
ControlRaise (self, controlMessage)
```

链表2——Model事件消息

这个链表包含记载model对象能够产生的事件的项目，表示如下：



为了向链表中加入或从链表中删除某些项目，中心MVC controller支持如下的方法：

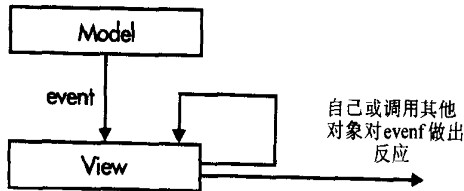
```
ModelRegister (self, event)
ModelUnregister (self, event)
```

当model对象要对一个动作反应、产生一个事件时，需在中心MVC controller端调用一个方法。中心MVC controller对象的方法如下：

```
MethodRaise (self, event)
```

链表3——View对象

这个链表包含记载View对象对model对象产生的事件的反应方式的项目，表示如下：



为了向链表中加入或从链表中删除项目，中心MVC controller对象支持如下的方法：

```
ViewRegister (self, self_method, model, event)
ViewUnregister (self, model, event)
```

当一个特定的model对象产生特定的事件时，中心MVC controller对象检查链表中的相关的View对象，并调用适当的方法。

可能的问题

当试图处理事件链表时，可能出现一些问题，它们是：

- 当在链表中进行增加、搜索或删除操作时要小心谨慎，因为其他对象可能在同一时间访问链表。重要的是，要锁定这个链表资源，以保证在任何时候都只有一个使用者可以访问链表。
- 在control/model链表中，要小心避免循环依赖关系的出现，如图5-9所示。

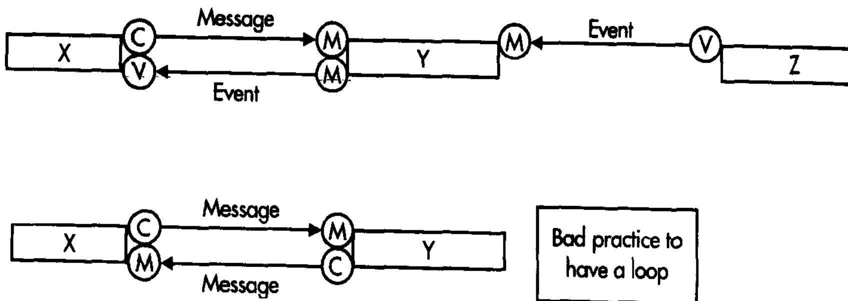


图5-9 循环依赖关系

5.3.2　线程方案

每个 model 对象都创建一个链表，以接受对某个特定事件感兴趣的 view 对象。MVC 的特点在于 model 对象的事件是异步的。为了帮助系统支持这种特点，每个 view 对象都创建一个向 model 对象注册的线程，然后，由这个线程处理发生的事件，如图 5-10 所示。

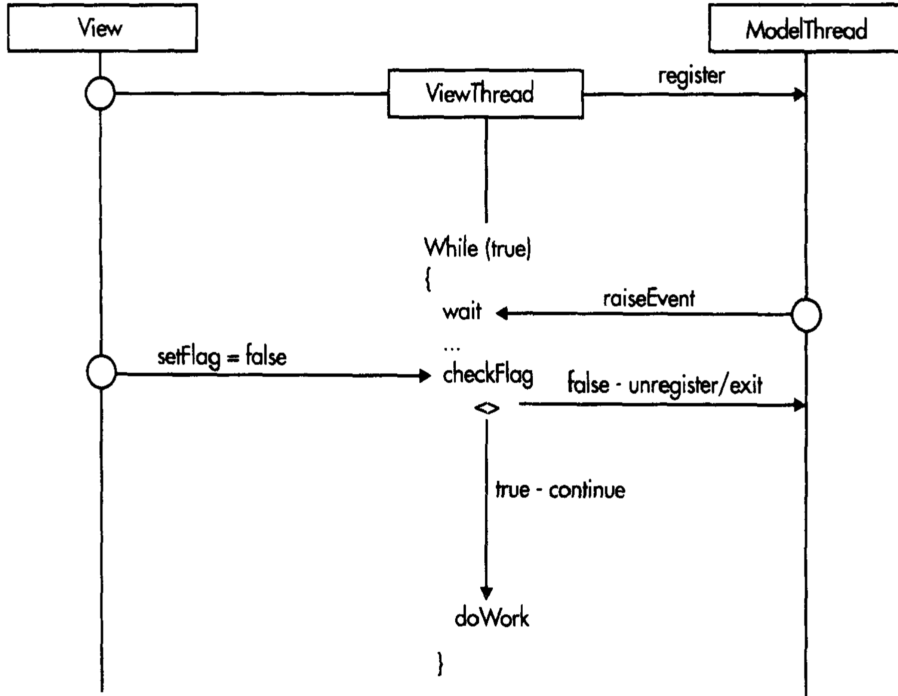


图 5-10　线程方案

1. 注册以处理事件

:View对象创建一个用于监视 **:Model**对象发生的事件的线程，即监视者线程（viewer thread），这个线程激活一个**:Model**对象端的注册方法，该方法将线程加入一个链表，这个链表是由对事件感兴趣的监视者组成的（见图 5-11）。

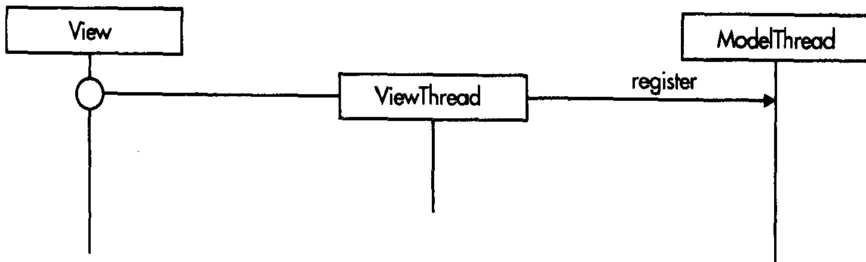


图 5-11　注册以处理事件

2. 等待事件

监视者线程拥有一个规定的程序循环。循环开始时，先将线程置于睡眠状态。当线程被唤醒时，它检查一个标志，看在睡眠时是否失去注册资格。如果标志还是有效的，线程则处理被注册的

方法，处理结束后，线程回到睡眠状态，并再次开始循环。如果标志无效，线程则终止，见图5-12。

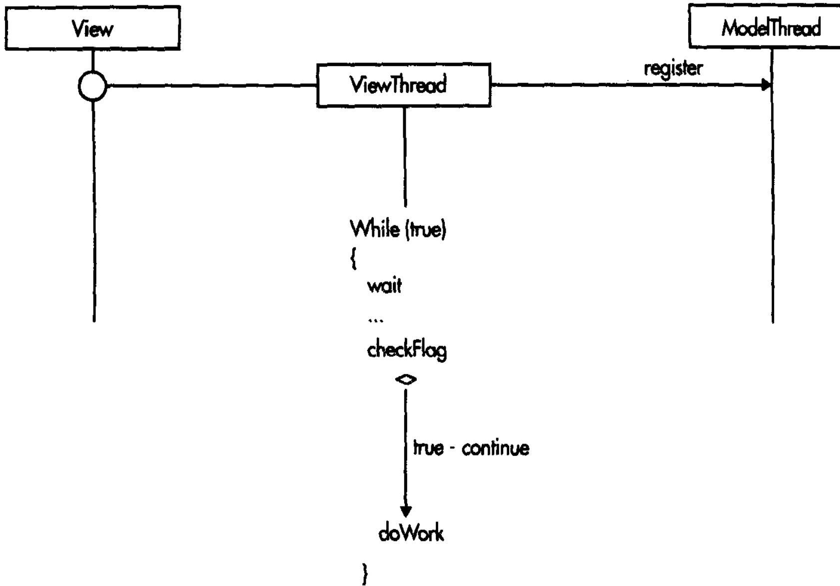


图5-12 等待事件

3. 产生事件

当:Model对象要产生一个事件时，它找到适当的事件链表，并拷贝该链表，这是为了避免修改正在被处理的链表。然后，它根据这个拷贝的链表，向表中的每一个线程发送一个唤醒的消息，见图5-13。

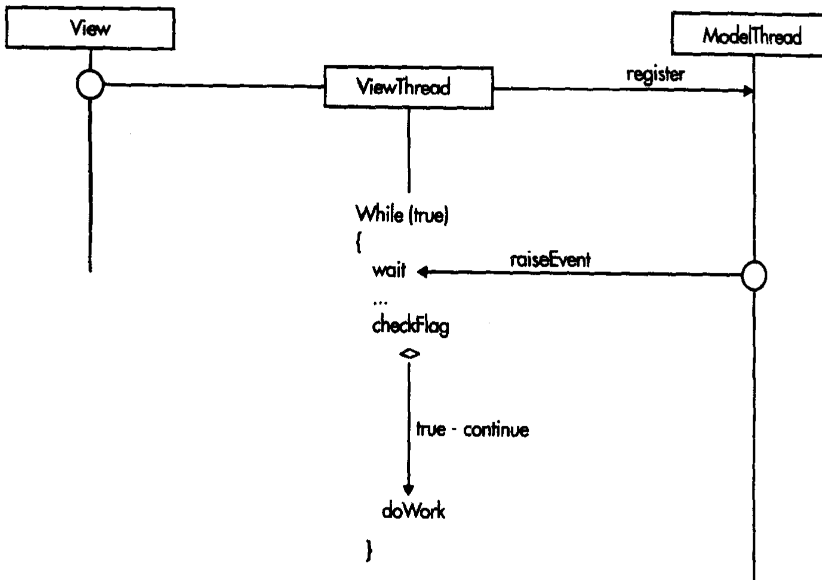


图5-13 产生事件

4. 失去监视事件的资格

当主:View对象线程不再希望监视:Model对象变化的消息时，它将线程标志置为false，这

样，下一次事件线程被唤醒时，它检查这个标志，退出循环，失去对:Model对象的监视资格，然后终止，见图5-14。

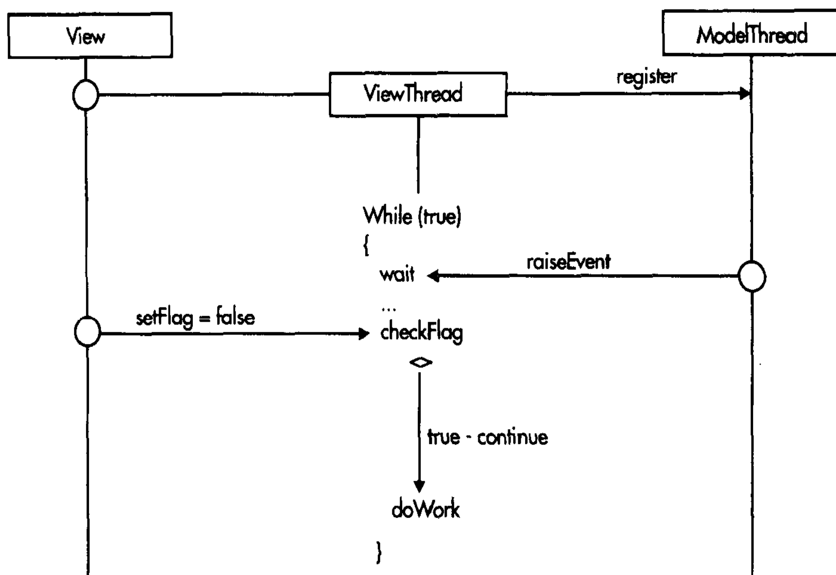
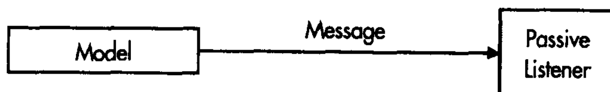


图5-14 失去监视事件资格

5.3.3 被动反应式方案

被动反应式方案不使用独立的线程去管理:View对象。与前面的方案不同，本方案中的:View对象只执行对model对象产生的事件的反应动作。



5.3.4 Java方案

这种方案使用Java编程语言提供的Observer接口和**Observable**类。

Observer接口能够使任何对象成为一个:View对象。Observable类用于创建:Model对象。这两个类自动处理MVC设计的消息通报函数(Notification function)。它们提供这样一种机制:view能自动被告知model中的变化。

下面的例子中有两个窗口（见图5-15)，其中一个窗口包含一个文字框，另一个窗口包含一个滚动条。例子中的model是一个从类**Observable**派生的类，并由主应用程序类创建。这个例子表明了当滚动条的滚动筏移动时，文字框中的数值是如何变化的，或者当文字框中的数值改变时，滚动筏的位置是怎样变化的。

1. Model

作为**Model**类的部分代码显示如下。类定义了一个属性，用于保存可监视的信息。:Model对象不直接改变信息的值，它提供一个方法，这个方法接受将被赋予这个属性的新的数值。当属性值改变时，:Model对象本身激活*setChanged*（）方法，表明有一个变化发生了，

再激活 *notifyObservers*（）方法，向<u>:View</u>对象告知这个变化的发生。

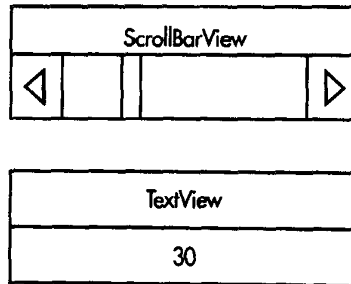


图 5-15 滚动条窗口和文字框窗口

```
import java.util.Observable;

public class Model extends Observable
{
    // set a default value
    private int value = 0;

    public Model (int newValue)
    {
        // set an initial value
        this.value = newValue;
    }

    public void setValue (int newValue)
    {
        // change the value
        this.value = newValue;

        // set the value changed flag
        // then notify all view objects
        setChanged();
        notifyObservers();
    }
}
```

2. 监视者

通过Observer接口的实现，可以创建监视另一个对象的变化的对象。Observer接口要求在新类中提供一个 *update*（）方法。每当<u>:Model</u>对象改变状态、并调用其 *notifyObservers*（）方法通知<u>:View</u>对象时，*update*（）方法即被调用。<u>:View</u>对象应据此确定新的数值，并适当调整它的监视情况。

作为View类的部分代码显示如下：

```
import java.util.Observer;
import java.util.Observable;

public class View implements Observer
{
```

```
    private Model model;

    public View(Model model)
    {
        this.model = model;
    }

    public void update (Observable obs, Object obj)
    {
        // if the object sending the notifyObservers is the
        // expected model
        if (obs == model)
        {
            // display the new value of the model attribute
            System.out.println (model.getCurrentValue());
        }
    }
}
```

3. 注册一个 View

创建一个**:Model**对象和若干个**:View**对象后，接下来是在 model 中注册这些 view。**Observable** 类使用 *addObserver* （）方法将一个 view 加入它内部的**:View**对象链表，以备向链表中的对象通知自身的变化。下面的代码显示了怎样使用 *addObserver* （）方法将**:View**对象 TextView 和 ScrollBarView 加入链表。

```
import java.util.Observer;
import java.util.Observable;

public class MVC
{
    public MVC()
    {
        // this model is created with an initial value
        // and minimum and maximum values
        Model model = new ObservableValue (50, 0, 100);
        TextView tv = new TextView (model);
        ScrollBarView sbv = new ScrollBarView (model);

        // add the view objects to the model
        model.addObserver(tv);
        model.addObserver(sbv);
    }

    public static void main(String [] args)
    {
        MVC m = new MVC();
    }
}
```

4. 实现控制

图 5-16 表明了例子的当前状态。

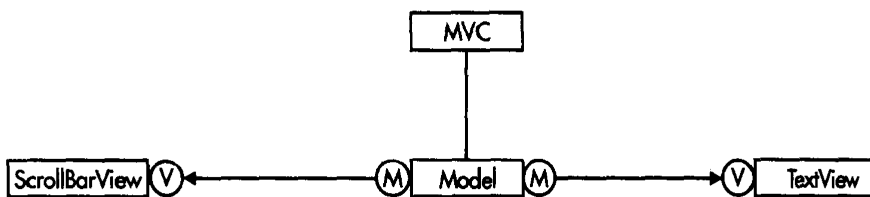


图 5-16 例子的当前状态

这个例子依据**:Model**对象中的数值，移动滚动筏到适当位置，并在文字框中显示适当的数字。**:Model**对象已经有一个 *setValue*（）方法，只需使用就可以了。

当一个**:View**对象中的变化被反映在**:Model**对象中时，每个现有的**:View**对象都可被转化为 controller。当用户移动滚动筏时，新的位置信息被设置在 model 中，这个新数值被反映在文字框中。当用户改变文字框中的数值时，新的数值信息被设置在 model 中，这个新数值被反映在滚动条中，即：将滚动筏移动到一个适当的位置。

5. 结局

图 5-17 表明了这个完整的例子，随后有 **Model** 类、**ScrollBarView** 类和 **TextView** 类的代码。

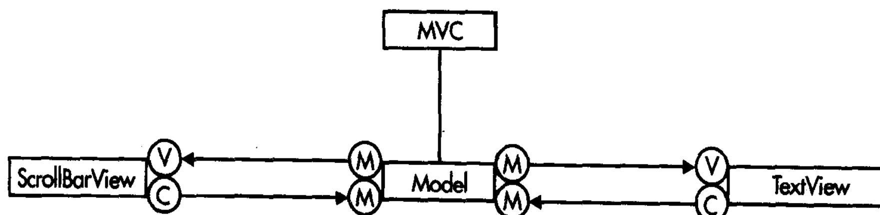


图 5-17 完整的例子

6. Model

```
import java.util.Observable;

public class Model extends Observable
{
    private int modelValue = 0;
    private int minValue = 0;
    private int maxValue = 0;

    public Model(int value, int min, int max)
    {
        this.modelValue = value;
        this.minValue = min;
        this.maxValue = max;
    }

    // this method sets the model value
    // and then notifies viewing objects
```

```
    public void setValue(int value)
    {
        this.modelValue = value;
        setChanged();
        notifyObservers();
    }

    // this method is used by the view objects to update themselves
    public int getCurrentValue()
    {
        return modelValue;
    }

    // this method returns the minimum value
    public int getMinimumValue()
    {
        return minValue;
    }

    // this method returns the maximum value
    public int getMaximumValue()
    {
        return maxValue;
    }
}
```

7. ScrollBarView

```
import java.awt.*;
import java.util.Observer;
import java.util.Observable;

public class ScrollBarView extends Frame implements Observer
{
    private Model model;

    private Scrollbar sb = null;

    public ScrollObserver(Model model)
    {
        super("Scroll Observer Tool");

        this.model = model;

        setLayout(new GridLayout(0, 1));
        sb = new Scrollbar(Scrollbar.HORIZONTAL,
                           model.getCurrentValue(), 10,
                           model.getMinimumValue(),
                           model.getMaximumValue());
```

```
        add(sb);
        pack();
        show();
    }
    public boolean handleEvent(Event evt)
    {
        // if the window is being destroyed,
        // unregister the view object then return
        if (evt.id == Event.WINDOW_DESTROY)
        {
            model.deleteObserver(this);
            dispose();
            return true;
        }
        else
        {
            if ((evt.id == Event.SCROLL_LINE_UP)
             || (evt.id == Event.SCROLL_LINE_DOWN)
             || (evt.id == Event.SCROLL_PAGE_UP)
             || (evt.id == Event.SCROLL_PAGE_DOWN)
             || (evt.id == Event.SCROLL_ABSOLUTE))
            {
                // change the value in the model to reflect
                // the position of the scrollbar
                model.setValue(sb.getCurrentValue());
                return true;
            }
        }

        return super.handleEvent(evt);
    }

    // this method uses the current value of the model
    // to position the scrollbar
    public void update(Observable obs, Object obj)
    {
        if (obs == model)
        {
                sb.setValue(model.getCurrentValue());
            }
        }
    }
```

8. TextView

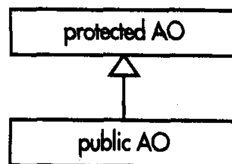
```
import java.awt.*;
import java.util.Observer;
import java.util.Observable;
```

```
public class TextView extends Frame implements Observer
{
    private Model model;
    private TextField txtf;

    private int minimum = 0;
    private int maximum = 0;
    public TextObserver (Model model)
    {
        super("Text Observer Tool");

        this.model = model;

        setLayout(new GridLayout(0, 1));

        minimum = model.getMinimumValue();
        maximum = model.getMaximumValue();

        txtf = new TextField(
               String.valueOf(model.getCurrentValue()));
        add(txtf);
        pack();
        show();
    }

    public boolean action(Event evt, Object obj)
    {
        if (evt.target == txtf)
        {
            int num = 0;
            try
            {
                num = Integer.parseInt(tf.getText());
            }
            catch (NumberFormatException nfe)
            {
                num = 0;
            }

            // if the num is less than the minimum
            // set it to minimum
            if (num < minimum)
                num = minimum;

            // if the num is greater than the maximum
            // set it to maximum
            if (num > maximum)
                num = maximum;
```

```
                model.setValue(num);
                return true;
            }
            return false;
        }

        public boolean handleEvent(Event evt)
        {
            // if the window is being destroyed,
            // unregister the view object then return
            if (evt.id == Event.WINDOW_DESTROY)
            {
                model.deleteObserver(this);
                dispose();
                return true;
            }

            return super.handleEvent(evt);
        }

        // this method uses the current value of the model
        // to set the text in the text field
        public void update(Observable obs, Object obj)
      {
            if (obs == model)
            {
                txtf.setText(String.valueOf(model.getCurrentValue()));
            }
        }
    }
```

5.4 暴露接口方案

本方案背后的前提是，派生类暴露更多基类的接口。这种方案的表示法如下。



以下的代码说明了如何使用这种方案：

```
class Parent
{
    public:
        virtual void method1 ();

    protected:
        virtual void method2 ();
}
```

```
class Child : public Parent
{
    public:
        virtual void method2 ();
}
```

下面是一个例子，说明了如何使用暴露接口方案来代替 C++友元结构。原始的设计如图 5-18 所示，它使用友元结构，设计一个 **SecretAgent** 类，该类将方法 *realName* 隐藏为一个私有方法，提供一个伪的方法 *name*，这样就能隐藏**:SecretAgent**实例的真实名称。**SecretAgent** 类将 **Controller** 类作为它的一个友元，允许**:Controller**实例访问隐藏的 *realName* 方法。

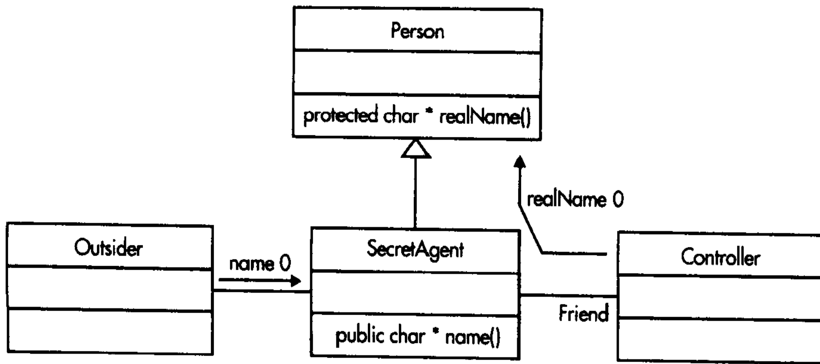


图 5-18　使用友元结构

新的设计如图 5-19 所示，使用暴露接口方案来处理 **SecretAgent** 类。由于设计使用了继承的

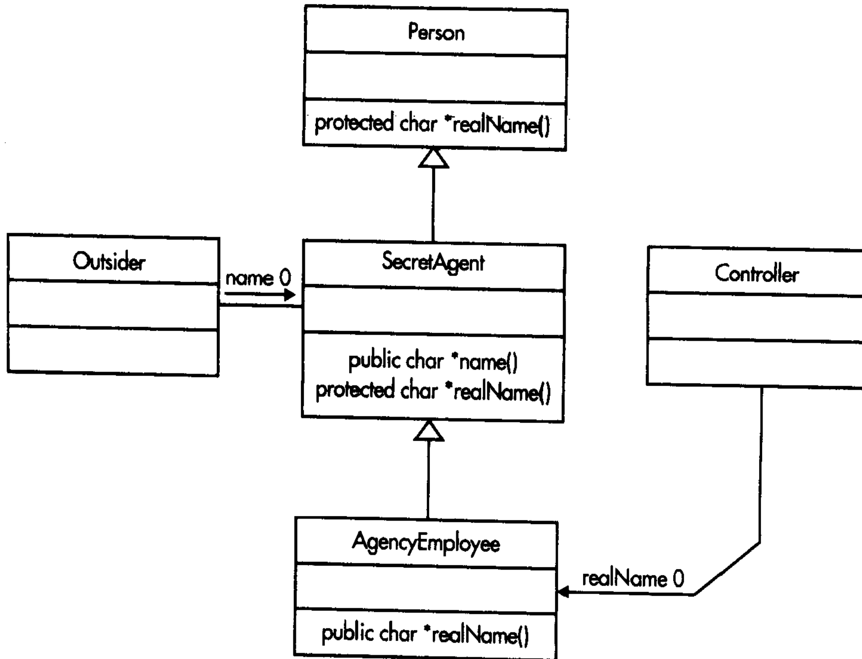


图 5-19　使用暴露接口方案

方法，*realName* 方法被声明为 protected 型的方法。要提供对这个方法的访问，从类 **SecretAgent** 派生一个新的类 **AgencyEmployee**。这个 **AgencyEmployee** 类是一个特别的类，被 Controller 用于访问适当的方法。

这个方案的惟一的问题是太真实了。一个敌类也能从 **SecretAgent** 类派生一个类，以访问 *realName* 方法。不管怎样，对于真正的秘密代理人，它们首先应被怀疑。

5.5 引用计数

在前些章节关于表层复制构造函数的讨论中，曾提到过一个问题：当原始的对象被析构后，复制的对象将不能再访问被它引用的数据，如图 5-20 所示。

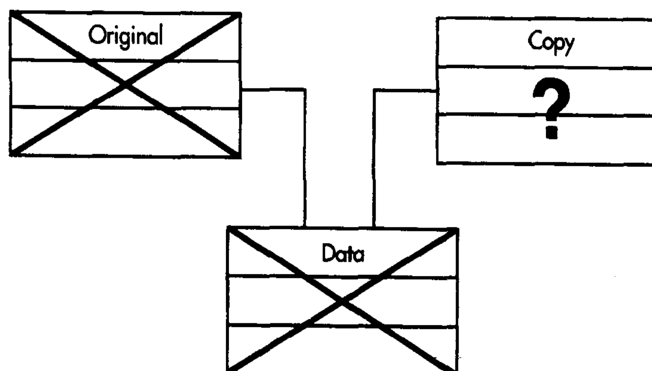


图 5-20 复制的对象失去数据访问权

这个问题的解决方法是对引用数据的对象进行计数，如图 5-21 所示。这个方法被称为引用计数（Reference counting)。引用计数的宗旨是对每一个新的引用，如拷贝对象，计数增加 1，当引用被清除时，计数减 1。如果计数到达零，则数据被释放。

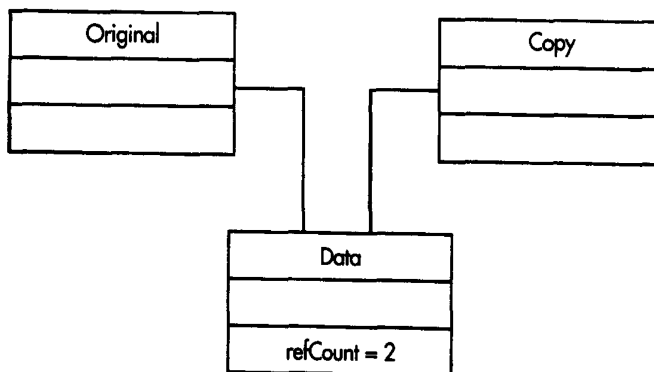


图 5-21 引用计数

实现引用计数有两种方法。第一种实现方法是通过继承进行的，而第二种方法是通过关联进行的。

5.5.1 通过继承实现引用计数

在这种实现方法中，定义一个类来管理引用计数。这意味着每个希望具有引用计数功能的类

都应该从该类派生，如图5-22所示。这个类声明一个属性（即计数值）和两个方法。第一个方法对计数值加1，第二个方法对计数值减1。不管怎样，如果计数值等于0，对象就可以清除了。

```
class RefCount
{
   public:
      incRef() { count++; }
      decRef() { --count; if (count == 0) delete this; }
   private:
      int count;
};
```

类RefCount可被这样使用：

```
class A: public RefCount
{
   // A is now able to support reference counting.
};
```

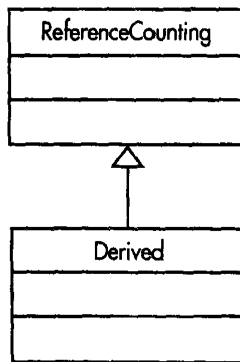


图5-22　通过继承实现引用计数

5.5.2 通过关联实现引用计数

在这种实现方法中，仍然有一个管理引用计数的类，但这个类的使用是通过关联而不是通过继承实现的，如图5-23所示。类的减小计数的方法（即方法decRef）也有所不同：该方法不

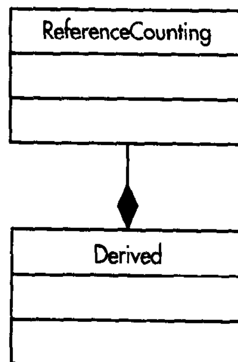


图5-23　通过关联实现引用计数

再控制数据对象的清除，类必须自己去清除。

新的引用计数类和与该类关联的方式的例子表示如下：

```
class RefCount
{
    public:
        incRef() { count++; }
        decRef() { --count; }
    private:
        int count;
};
class String
{
    public:
        void incRef() { refCount.increment(); }
        void decRef()
            { if (refCount.decrement() == 0) { delete (this); } }

    private:
        char    *buffer;
        RefCount refCount;    // reference count
};
```

使用引用计数对象来实现一个复制构造函数的例子如下：

```
String::operator = (const String& str)
{
   char *temp = buffer;
   // if the incoming string buffer is not empty, increment the count
   if (str.buffer)
       str.incRef();

   // overwrite the existing buffer with the incoming string buffer
   buffer = str.buffer;

   // if the old string buffer was not empty decrement the count
   if (temp)
       temp.decRef();

   // return a pointer to this object
   return *this;
}
```

5.5.3 多线程应用程序

至于选择哪一种实现方法，并没有一个正确的答案。不过，通过关联实现引用计数确实考虑到了支持所谓原子整型（atomic integer）自然数据类型的操作系统的特性。如果应用程序被设计为多线程系统，这种数据类型尤其重要。问题在于，即使已经有线程正在访问引用计数，

在任何时候，任何线程都有可能试图访问这个引用计数。为了避免这样的冲突，原子整型提供一个锁，以保证在指定的时间只有一个线程能修改引用计数的数值。还要注意的是，上面的复制构造函数的代码中，作为参数传进来的:String对象在所有操作进行之前，其引用计数就已经被增加了，这样能避免清除将被复制的对象。

对于不支持自然原子整型数据类型的操作系统，下面的代码表明了如何使用排斥锁创建引用计数。

```
# include <pthread.h>

class RefCount
{
    public:
        RefCount ();
        ~RefCount();
    // lock the mutex, save the incremented value,
    // then unlock the mutex and return the saved value
    int incRef()
    {
        pthread_mutex_lock (&m_mutex);
        int rval = ++m_value;
        pthread_mutex_unlock (&m_mutex);
        return rval;
    }

    // lock the mutex, save the decremented value,
    // then unlock the mutex and return the saved value
    int decRef()
    {
        pthread_mutex_lock (&m_mutex);
        int rval = --m_value;
        pthread_mutex_unlock (&m_mutex);
        return rval;
    }

private:
    pthread_mutex_t     refMutex;
    int                 count;
};
```

5.6 小结

本章是集中讨论纯粹的面向对象设计的最后一章。下一章将讨论应用程序被实现以后存在的问题，随后的几章将涉及测试和调试，还有其他主题。

第四部分　编　　程

目标：

- ▶ 理解测试涉及的各个层次
- ▶ 理解调试应用程序使用的技术
- ▶ 学习三种最常用的调试工具（DBX、GDB 和 JDB）的基本命令
- ▶ 学习移植——多操作系统支持和多语言支持
- ▶ 理解应用程序的生命周期

第6章 测　　试

本章将讨论以下内容：

- ▶ 学习测试各方法的重要性
- ▶ 理解如何测试完整的类
- ▶ 理解强度测试与规模测试的差别
- ▶ 理解回归测试的重要性

测试的重要性怎么强调都不为过。测试的目的是为了保证产品的质量，也可以用于保证产品不发生退步（例如，以前正常的功能被破坏了）。最后，测试也能帮助软件移植小组保证他们所完成产品的版本的每一点都像原始版本一样好。

测试有许多层次，每个层次在产品的生存周期内都有其特殊的位置。本章所讨论的测试涵盖了从测试每一行代码到测试整个系统的所有领域。

所有测试都必须有计划。如果一个特定的测试没有计划，那么，真正实施的测试就不可能覆盖该应用程序的所有重要部分。

在本章后面的关于测试的例子中，利用了 SimCo 案例研究中的 Java 代码。

6.1 测试装备

测试装备是一种环境，软件的各组件可以被放入并被测试，如果被测试的类不与任何其他的类交互，那么，测试装备就由一个主程序和被测试的类组成。然而，如果被测试的类确实与其他的类有交互活动，那么，测试装备就由主程序、被测试的类及替代其他与之交互的类的哑类组成。对被测试的每一个新类或子系统，则有可能需要一个新的主程序。

测试装备的概念在于为一个或多个类创建一个实例，再通过一系列的方法调用来“锻炼”这些类。对每一个测试，重要的是它必须被书面写出，是可重现的、自包含的，并且是专门特定的。

如果利用一个随机数生成器来模拟不可预知的行为，那么，可重现性就尤为重要。当测试单元的逻辑流程时，不要将随机数生成器的种子（即生成数的起始值）设置为依赖于某一真实的随机事件（如当前时间)，那样就会真的给出一个不可预知的行为，一旦检测出问题，测试不可能被重现。所以，从这个意义上说，必须给随机数生成器一个固定的起点。一次测试成功后，改变这个起点，重复多次测试，逐步确信测试是成功的。

测试应是自包含的，这样测试就可以在封闭的情况下被任何人进行实施。当要求某人改正一个问题时，自包含性就显得很重要了，因为他需要多次重新运行测试以隔离出问题。最后，当问题被改正后，还需要重新运行该测试，确认问题已被改正。

测试还应是非常专门的，以保证达到测试一个功能的特定部分的目的。如果一个测试的内容过于粗泛，当问题被检测出时，则就需要写出更多足够专门的测试，以识别出该问题。

为了有助于确信每一行代码都被执行，可以在程序的关键点使用打印语句或输出语句。重要的是输出语句要放在适当的位置，并且，当不再需要它们时能被关掉。这可能显得有点多余，但如果输出语句不能被正确地移去，可能会带来问题。系统环境变量可作为一个开关，这样的开关伴随一个日志文件名。系统环境变量将在讨论“类测试”时引入。

下面是关于所需的各个测试层次的描述。

6.2 关于构造方法和析构方法的测试

在类中的任何东西被测试之前，重要的是搞清楚能否从该类创建实例。创建了实例之后，通过测试，确信当实例被清除之后，它不会在系统中留下残余（即所谓的内存泄漏）。

一个测试的例子

被测试的 Java 代码源自 CashAccount.java。

1. TestHarness.java

```
/*
** Name: TestHarness
** This class is the main class in the test harness
*/
import java.util.*;


public class TestHarness
{
   /*
   ** define a variable of the desired class
   */
   public         CashAccount      cash;

   public TestHarness ()
   {
      System.out.println ("Pre-new CashAccount");


      /* Create an object of class 'CashAccount' */
      cash = new CashAccount();
      System.out.println ("Post-new CashAccount");
   }


   public static void main (String[] args)
   {
       TestHarness th = new TestHarness ();
   }
}
```

2. CashAccount.java

```
/*
** Name: CashAccount
** This class has been written to provide support for the bank
** of the company simulated in SimCo.
*/
import java.lang.*;
import java.text.*;
import java.util.*;

/*
** Name: CashAccount
** This method is the constructor for the class
** It assigns zeros to the balance array of the company
**
** Input: none
** Output: none
*/
public class CashAccount
{
    public CashAccount ()
    {
       System.out.println ("New CashAccount instance created");
    }
}
```

3. 预期的输出

```
Pre-new CashAccount
New CashAccount instance created
Post-new CashAccount
```

6.3 方法测试

该层次的测试是最彻底的，在某种意义上说，也是最冗长乏味的。尽管如此，由于在一些场合该层次的测试是基本的要求，所以认真研究如何进行这种测试是一个聪明的选择。

类中的每一个方法都要在孤立的状态下被测试，测试的思路是以各种能想像得到的方式运行该方法。测试的目标是，在一系列的测试中，该方法的任意一行代码都必须至少被运行一次。不仅如此，代码中每一种分支的组合都必须被运行。这种形式的测试称为逻辑测试，因为它测试代码中的逻辑流程。

在输出中，重要的是改写方法中需执行打印语句的地方。下面有几个关于在何处使用打印语句的例子，并描述了每个结构的工作方式：

- if-then-else
- for 循环
- while 循环

- switch 语句
- try-catch
- 函数调用

6.3.1 if-then-else

if-then-else 结构根据测试条件取值（TRUE 或 FALSE），在两个选项中选择一个。

```
if (test condition)
{
   print ("rephrase the test condition - TRUE")
   code to be executed
}
else
{
   print ("rephrase the test condition - FALSE")
   code to be executed
}
```

该结构也可以被嵌套，例如：

```
if (<today is a workday>)
{
    print ("<today is a workday> - TRUE")
    code to be executed
}
else
{
    print ("<today is a workday> - FALSE")
    if (<today is a Saturday>)
    {
        print ("<today is a Saturday> - TRUE")
        code to be executed
    }
    else
    {
        print ("<today is a Saturday> - FALSE")
        code to be executed
    }
}
```

6.3.2 for 循环

for 循环结构具有这样一种能力，即：将一个动作执行预定的次数。尽管每一次循环正常时可运行到结束，但通过使用另外的结构，有可能改变 for 循环的运行方式。例如，使用 Continue 语句可强制提前结束当前循环，使用 Break 语句可强制提前退出 for 循环结构。

```
print ("rephrase for loop criteria - BEFORE")
for (for loop criteria 'initialization;condition;increment')
{
    print ("current value of loop variable")
    code to be executed

    <test condition>
    {
        print ("break out of the loop -\
                current value of loop variable")
        break
    }
    <test condition>
    {
        print ("continue with next loop -\
                current value of loop variable")
        continue
    }

    print ("end of loop")
}
// Because the loop could have been exited other than
// the loop condition having been met
// restate the current value of the loop count
print ("rephrase for loop criteria - AFTER")
```

6.3.3 while 循环

while 循环结构具有这样一种能力，即：将一个动作执行不确定的次数。当测试条件取值为 TRUE 时循环继续。每一次循环一般都会正常完成，但它也与 for 循环结构一样，可以使用另外的结构改变循环的运行方式。例如，可以使用 continue 语句强制提前结束当前的循环，可以使用 break 语句强制提前退出循环结构。

```
print ("rephrase while loop criteria - BEFORE")
while (test condition)
{
    print ("rephrase the loop condition")
    code to be executed

    <test condition>
    {
        print ("break out of the loop")
        break
    }

    <test condition>
    {
```

```
        print ("continue with next loop")
        continue
    }

    print ("end of loop - rephrase the loop condition")

}
// Because the loop could have been exited other than
// a change in test condition result, restate the current
// value of the test condition
print ("rephrase while loop criteria - AFTER")
```

6.3.4 switch语句

switch结构在多重选项中提供了一个选择。有一种测试条件，它既不是TRUE也不是FALSE，而是映射一个数值。每一个映射的数值都在某个‘case (value)’语句中有一个捕捉点。‘default’语句则用于捕捉不能被任何‘case’语句捕捉的数值。

```
print ("switch condition - BEFORE")
switch (switch condition)
    case (value):
        print ("rephrase switch condition - switch value")
        code to be executed
    default:
        print ("rephrase switch condition - DEFAULT")
        code (if any) to be executed
print ("switch condition - AFTER")
```

6.3.5 try-catch

try-catch结构的好处在于，它允许一段代码有局部的异常处理。当出现了异常时，该结构找到设计的处理该情况的catch语句，解决程序遇到的问题（必要时，可返回到主程序）。然而，最典型的用法是，将catch语句设置为捕捉异常的手段，这些异常来自与本地代码完全无关的其他地方。例如，可以在主程序中设置catch语句，以在程序退出之前处理一般的异常。从测试的观点看，这并不太有用，因为无法确知异常的根源，特别是当有多处可能产生这种异常时。所以，很重要的一点是，在异常出现之前放置打印语句。

```
try
{
    print ("Inside 'try' #n")
    code to be executed
        // whenever an exception is about to be generated
        // it is important to document it.
        print ("Just before a throw of exception 'x'")
        throw new exception_type (argument)
}
```

```
catch (exception_type and argument)
{
    print ("Inside 'catch' #n - exception 'x'")
    code to be executed
}
```

附加的 Java 程序

```
finally
{
    // This code can be reached because:
    // 1: Normal by reaching the end of the try-block
    // 2: After an exception that has been caught
    // 3: After an exception that has not been caught
    // 4: Any other reason leaving the try-block
    //    break, continue or return statement
    print ("Inside 'Finally' #n")
}
```

6.3.6 函数调用

与通过循环或 if-then-else 结构跟踪程序进程同样重要的是，跟踪方法本身是如何被调用、从哪儿被调用的。尤其是当一个方法可以从不同的地方被调用时，这一点显得格外重要，因为它可用于监视应用程序的流程。如果一个方法被重复调用，它可能成为程序运行瓶颈的焦点，所以监视系统的流程是很重要的。

```
print ("'current method' - 'next method' - BEFORE")
next_method (...)
print ("'current method' - 'next method' - AFTER")
```

6.3.7 测试单个方法的例子

下面被测试的方法是 CashAccount.java 中的 setBalance ()。

1. 加入 TestHarness.java 中的代码

```
/* previous code that constructed the 'cash' object */

System.out.println ("Pre-CashAccount:setBalance-2000.0");
cash.setBalance (2000.00);
System.out.println ("Post-CashAccount:setBalance-2000.0");

/* end of TestHarness constructor */
```

2. 加入 CashAccount.java 中的代码

```
/*
** Name: setBalance
** This method sets the balance of the current month
```

```
**
** Input: amount to be set
** Output: none
*/
public void setBalance (double money)
{
    System.out.println ("CashAccount:setBalance-" + money);
    balance = money;
    System.out.println
        ("CashAccount:setBalance-balance=" + balance);
}
```

3. 预期的输出

```
Pre-new CashAccount
New CashAccount instance created
New CashAccount-balance=0.0
Post-new CashAccount
Pre-CashAccount:setBalance-2000.0
CashAccount:setBalance-2000.0
CashAccount:setBalance-balance=2000.0
Post-CashAccount:setBalance-2000.0
```

6.4 类测试

当成功地完成了所有方法的测试之后，用户现在处于这样一个位置：即他/她可以保证每个方法的实现是正确的。接着往下进行的一组测试将关注一个类在生存期测试中的反应。这些测试是对一个类的整体测试，它们确定，在一个典型的系统方案中，类被创建时的反应和它的每一个方法被调用时的反应。测试应该确定，对任何方法的调用是否会对其他方法产生不希望的边缘效应或后果。如，某些方法只能以特定的次序被调用，例如必须先初始化环境，然后才能干些什么。如果存在这种情况，仅利用测试装备个别地调用方法，问题是显示不出来的，但它确实是会引起问题的因素。因此，设计者或程序员必须确保防止有问题的方法调用组合，或它们可通过适当的消息被处理。

现在测试装备需要被修改，以便从准备好的脚本中运行。每个脚本通过一系列方案去运行类，目的是遍历所有的方法调用的组合。

利用脚本测试的例子

下面的脚本被设计用于测试方法调用组合中的一种，这些方法调用是为类 **CashAccount** 而制定的。

1. 重写 TestHarness.java

可在 TestHarness.java 中添加两段。第一段允许它开/关输出语句，第二段允许测试装备处理脚本文件。可以给输出语句指定一个数，使得不同的代码段都能在不同的时候允许自己的输出语句。另外，可以指定一个文件名以接受在此之前已直接显示给用户的消息。在 C++ 中也

可使用系统环境变量，利用 getenv 命令访问它们。

让我们从添加 SimCo 中到处使用的全局变量开始。第一个变量设置数组的大小，第二个变量用于引用上个月（Last month）。

```
public final int nummonths = 6;
public final int lastmonth = 5;
```

不是在类 **TestHarness** 中使 *traceOut* 方法作为 static 方法，而是建立一个将它引用为实例方法的能力。这样可使得代码易于处理。

```
private static TestHarness      testH;
private static int              traceN;
private static String           traceF;
private static PrintStream      log;

private void traceOut (int number, String mesg)
{
    /*
    ** If the system environment trace level matches that of the
    ** output statement, output the message
    */
    if (number == traceN)
    {
        log.println (mesg);
    }
}

public TestHarness ()
{
    ...
    /* needs to be placed inside the constructor for TestHarness */
    testH = this;
    ...
}

public static void main (String[] args)
{
    /*
    ** Read the trace level system environment variable,
    ** if the value is not set, return "0"
    */
    String traceNstr = System.getProperty ("TRACEN", "0");
    /* Parse the returned value to return the integer value */
    traceN = Integer.parseInt (traceNstr);
    /*
    ** Read the file system environment variable,
    ** if the value is not set,
    ** use "test.out" to create a file in the current directory
```

```
    */
    traceF = System.getProperty ("TRACEF", "test.out");

    /* if the trace level is not the default open the file */
    if (traceN != 0)
    {
        try
        {
            log = new PrintStream
                (new FileOutputStream (traceF), true);
        }
        catch (FileNotFoundException e) { System.err.println (e); }
    }
    TestHarness th = new TestHarness ();
}

public static TestHarness GetThis ()
{
    return testH;
}
```

利用下面的代码设置应用程序的环境变量：

```
java -DTRACEN=1 -DTRACEF=Test1.dat TestHarness
```

接下来的变量用于读入脚本，并将它解析，以备使用。

String	line	脚本中的行被读入该变量
StringTokenizer	t	该变量将行解析为独立的标记
int	classNum	类编号，允许使用 switch 语句
String	className	被测试的类名
int	methodNum	方法编号，允许使用 switch 语句
String	method	被测试的方法名
String	args	在脚本文件中指定的字符串参数
int	intVal	当需要使用时，它就是“int”
Double	Dval	为从字符串中提取一个 double 数值，首先必须将它变成一个 Double
double	dVal	当需要使用时，它就是“double”

文件的打开必须被包在一个处理 IOException 的 try-catch 结构内：

```
try
{
  /* open the script file */
  BufferedReader script =
  new BufferedReader (new FileReader("script.dat"));

  /* close the script file */
  script.close ();
}
catch (IOException e)
{ System.err.println (e); }
```

脚本文件的读入是在一个永不结束的 for 循环中进行的。该循环仅在以下的情况下终止：存在读文件错、到达文件末尾、或遇到预先定义的记号。预先定义的记号是位于行首的单个的“:”。

```
for (;;)
{
    /* read a line and break if it is empty */
    line = script.readLine();
    if (line == null) break;

    /*
    ** count the number of tokens there are on the line
    ** if the line consists of a solitary ':', break
    */
    t = new StringTokenizer (line, ":");
    if (t.countTokens () == 0) break;

    /* parse each line */
}
```

类序号是从行内提取的第一个标记。将它提取为一个数，是为了能够把它用在 switch 语句中。如果类序号等于零，则该行为注释行。

```
/* if the classNum == 0, it is a comment, ignore it */
classNum = Integer.parseInt (t.nextToken ());
if (classNum == 0)
    continue;
```

随后可从该行提取下面的几则信息：首先是类名，它是一个字符串；其次是被测试的方法的序号，它被提取为一个数，以便能用在 switch 语句中；接下来的是被测试的方法名，它是字符串；最后是一个字符串，它包含一个由逗号分隔的参量列表。

```
/* process the remainder of the line */
className = t.nextToken ();
methodNum = Integer.parseInt (t.nextToken ());
routine = t.nextToken ();
args = t.nextToken ();
```

在这个脚本例子中，被测试的类是 **CashAccount** 类，有 6 个可能被测试的方法。

```
case 1: cash = new CashAccount (); break;
case 2:
{
    /* balanceByIndex takes an int argument */
    intVal = Integer.parseInt(args);
    cash.balanceByIndex (intVal);
    break;
}
```

```
case 3:
{
    /* credit takes a double argument */
    DVal = Double.valueOf (args);
    dVal = DVal.doubleValue ();
    cash.credit (dVal);
    break;
}
case 4:
{
    /* debit takes a double argument */
    DVal = Double.valueOf (args);
    dVal = DVal.doubleValue ();
    cash.debit (dVal);
    break;
}
case 5:
{
    /* setBalance takes a double argument */
    DVal = Double.valueOf (args);
    dVal = DVal.doubleValue ();
    cash.setBalance (dVal);
    break;
}
case 6: cash.adjustMonth(); break;
```

在生成的各段输出之间，插入一个空行，这样做仅是为了使输出阅读起来方便一些。

```
/* print a separator between line of output */
traceOut (1, " ");
```

2. CashAccount.java

为了能引用声明为 **TestHarness** 类的一部分的全局变量，需要创建一个 **TestHarness** 类的对象。

```
TestHarness    parent;
```

关于当前月与上月收支平衡的信息保存在一个 double 型的数组中：

```
double   balance [];
```

类 **CashAccount** 的构造函数获得对 **TestHarness** 的引用，以创建一个 double 型数组，然后将数组的各数值赋为零：

```
public CashAccount ()
{
    int    i;

    /*
```

```
    ** Get the reference of the test harness
    */
   parent = (TestHarness)TestHarness.GetThis();

   parent.traceOut (1, "New CashAccount instance created");

   balance = new double [parent.nummonths];
   parent.traceOut (1, "Pre-for loop:i < parent.nummonths=" +
       parent.nummonths);
   for (i = 0; i < parent.nummonths; i++)
   {
       parent.traceOut (1, "start:i < parent.nummonths:i=" + i);
       balance [i] = 0;
       parent.traceOut (1, "balance [" + i + "]=" + balance [i]);
       parent.traceOut (1, "end:i < parent.nummonths");
   }
   parent.traceOut
       (1, "Post-for loop:i=" + i + " < parent.nummonths");
}
```

方法 *balanceByIndex* 返回数组中的某个指定的余额：

```
public double balanceByIndex (int index)
{
    parent.traceOut (1, "CashAccount:balanceByIndex-balance [" +
        index + "]=" + balance [index]);
    return balance [index];
}
```

方法 *credit* 将指定的数额加到当前月的余额上：

```
public void credit (double money)
{
    parent.traceOut (1, "CashAccount:credit-balance=" +
        balance [parent.lastmonth] + " money " + money);
    balance [parent.lastmonth] = balance [parent.lastmonth] + money;
    parent.traceOut (1, "CashAccount:credit-balance=" +
        balance [parent.lastmonth]);
}
```

方法 *debit* 从当前月的余额中减去指定的数额，如下所示。

```
public void debit (double money)
{
    parent.traceOut (1, "CashAccount:debit-balance=" +
        balance [parent.lastmonth] + " money " + money);
    balance [parent.lastmonth] = balance [parent.lastmonth] - money;
    parent.traceOut (1, "CashAccount:debit-balance=" +
        balance [parent.lastmonth]);
}
```

方法 *setBalance* 将当前月的余额设置为指定数值：

```
public void setBalance (double money)
{
    parent.traceOut (1, "CashAccount:setBalance-" + money);
    balance [parent.lastmonth] = money;
    parent.traceOut (1, "CashAccount:setBalance-balance=" +
        balance [parent.lastmonth]);
}
```

方法 *adjustMonth* 将保存在数组中的各月余额向前置换一个月。最早的月被清除，而当前月的余额则变成上月的余额：

```
public void adjustMonth ()
{
    int    i;

    parent.traceOut (1, "Pre-for loop:i < parent.lastmonth=" +
        parent.lastmonth);
    for (i = 0; i < parent.lastmonth; i++)
    {
        parent.traceOut (1, "start:i < parent.lastmonth:i=" + i);
        balance [i] = balance [i + 1];
        parent.traceOut (1, "balance [" + i + "]=" + balance [i]);
        parent.traceOut (1, "end:i < parent.lastmonth");
    }
    parent.traceOut
        (1, "Post-for loop:i=" + i + " < parent.lastmonth");
}
```

3. 脚本文件——预期的输出

在下面的表中，左边是传给测试装备的脚本，右边是预期的输出：

0: This script is designed to test the	New CashAccount instance created
0: creation of a CashAccount object	Pre-for loop: i < parent.nummonths = 6
1: CashAccount: 1: new: -:	start: i < parent.nummonths: i = 0
	balance [0] = 0.0
	end: i < parent.nummonths
	start: i < parent.nummonths: i = 1
	balance [1] = 0.0
	end: i < parent.nummonths
	start: i < parent.nummonths: i = 2
	balance [2] = 0.0
	end: i < parent.nummonths
	start: i < parent.nummonths: i = 3
	balance [3] = 0.0
	end: i < parent.nummonths
	start: i < parent.nummonths: i = 4

	balance [4] =0.0 end: i < parent.nummonths start: i < parent.nummonths: i=5 balance [5] =0.0 end: i < parent.nummonths Post-for loop: i=6 < parent.nummonths
0: Then to check that it is full of zeros 1: CashAccount: 2: balanceByIndex: 0: 1: CashAccount: 2: balanceByIndex: 1: 1: CashAccount: 2: balanceByIndex: 2: 1: CashAccount: 2: balanceByIndex: 3: 1: CashAccount: 2: balanceByIndex: 4: 1: CashAccount: 2: balanceByIndex: 5:	 CashAccount: balanceByIndex-balance [0] =0.0 CashAccount: balanceByIndex-balance [1] =0.0 CashAccount: balanceByIndex-balance [2] =0.0 CashAccount: balanceByIndex-balance [3] =0.0 CashAccount: balanceByIndex-balance [4] =0.0 CashAccount: balanceByIndex-balance [5] =0.0
0: To set the balance of the current month 1: CashAccount: 5: setBalance: 2000.0: 0: Rotate the months 1: CashAccount: 6: adjustMonth: -:	CashAccount: setBalance-2000.0 CashAccount: setBalance-balance=2000.0 Pre-for loop: i < parent.lastmonth=5 start: i < parent.lastmonth: i=0 balance [0] =0.0 end: i < parent.lastmonth start: i < parent.lastmonth: i=1 balance [1] =0.0 end: i < parent.lastmonth start: i < parent.lastmonth: i=2 balance [2] =0.0 end: i < parent.lastmonth start: i < parent.lastmonth: i=3 balance [3] =0.0 end: i < parent.lastmonth start: i < parent.lastmonth: i=4 balance [4] =2000.0 end: i < parent.lastmonth Post-for loop: i=5 < parent.lastmonth
0: Credit the balance of the current month 0: with 360.0 1: CashAccount: 3: credit: 360.0:	CashAccount: credit-balance=2000.0 money 360.0 CashAccount: credit-balance=2360.0
0: Rotate the months 1: CashAccount: 6: adjustMonth: -:	Pre-for loop: i < parent.lastmonth=5 start: i < parent.lastmonth: i=0 balance [0] =0.0 end: i < parent.lastmonth start: i < parent.lastmonth: i=1 balance [1] =0.0 end: i < parent.lastmonth start: i < parent.lastmonth: i=2

	balance [2] =0.0 end: i < parent.lastmonth start: i < parent.lastmonth: i=3 balance [3] =2000.0 end: i < parent.lastmonth start: i < parent.lastmonth: i=4 balance [4] =2360.0 end: i < parent.lastmonth Post-for loop: i=5 < parent.lastmonth
0: Debit the balance of the current month 0: with 340. 0 1: CashAccount: 4: debit: 340. 0: 0: Rotate the months 1: CashAccount: 6: adjustMonth: -:	CashAccount: debit - balance=2360. 0 money 340. 0 CashAccount: debit - balance=2020. 0 Pre - for loop: i < parent. lastmonth=5 start: i < parent. lastmonth: i=0 balance [0] =0. 0 end: i < parent. lastmonth start: i < parent. lastmonth: i=1 balance [1] =0. 0 end: i < parent. lastmonth start: i < parent. lastmonth: i=2 balance [2] =2000. 0 end: i < parent. lastmonth start: i < parent. lastmonth: i=3 balance [3] =2360. 0 end: i < parent. lastmonth start: i < parent. lastmonth: i=4 balance [4] =2020. 0 end: i < parent. lastmonth Post - for loop: i=5 < parent. lastmonth
0: Re-display the array of monthly balances	
1: CashAccount: 2: balanceByIndex: 0:	CashAccount: balanceByIndex-balance [0] =0.0
1: CashAccount: 2: balanceByIndex: 1:	CashAccount: balanceByIndex-balance [1] =0.0
1: CashAccount: 2: balanceByIndex: 2:	CashAccount: balanceByIndex-balance [2] =2000.0
1: CashAccount: 2: balanceByIndex: 3:	CashAccount: balanceByIndex-balance [3] =2360.0
1: CashAccount: 2: balanceByIndex: 4:	CashAccount: balanceByIndex-balance [4] =2020.0
1: CashAccount: 2: balanceByIndex: 5:	CashAccount: balanceByIndex-balance [5] =2020.0
:	< end of script >

6.5 整体测试

当一个类的方法被组合测试时，可能会发现一些问题。类似地，设计整体测试是为了保证当多个类被组合使用时，它们能像期望的那样发挥作用。正是在这个阶段，测试装备中使用的骨架可被先前已成功通过测试的类替代。利用真实类进行测试得出的结果与相对于骨架进行测试得出的结果应该相匹配。如果测试结果有差异，显然就存在问题。

有可能骨架本身就不能正确地摹拟真实的类，如果出现这种场合，则需重写骨架，同时，那些与之相关的测试都必须重新进行，因为它们已经失效了。

整体测试的例子

为了能进行整体测试，必须对测试装备的源代码进行修改，以支持多个类。可以通过改变处理脚本文件的方式以支持多个类。每个类通过它自己的方法被处理。

在下面的例子中，加入了两个新的类。被运行的脚本将创建一个**:CashAccount**对象、一个**:Factory**对象，然后创建一个新的**:Machine**对象，并将其加入**:Factory**。

1. 修改后的 TestHarness.java

```
public        Factory        factory;

public TestHarness ()
{
...
    /* Process each class in its own method */
    switch (classNum)
    {
        case 1: processCashAccount (methodNum, args); break;
        case 2: processFactory (methodNum, args); break;
        case 3: processMachine (methodNum, args); break;
    }
...
}
```

这是一个基于方法的新类的例子：

```
private void processFactory (int methodNum, String args)
{
    int             intVal;
    Double          DVal;
    double          dVal;
    int             retI;
    switch (methodNum)
{
    case 1: factory = new Factory (); break;
    /*
    ** factory methods 2,3 and 4 are not tested by this script
    */
    case 5:
    {
        /*
        ** howManyMachines returns the number of
        ** machines already owned
        */
        retI = factory.howManyMachines ();
        parent.traceOut
```

```
                (1, "processFactory:howManyMachines=" + retI);
            break;
        }
    }
}
```

2. Factory.java

```
import java.lang.*;
import java.text.*;
import java.util.*;

public class Factory
{
   TestHarness    parent;

   /*
   ** A Factory keeps a record of how many machines it currently
   ** contains it also keeps a record of how much stock it had at
   ** the start of a manufacturing cycle 'instock' and at the end
   ** of the run 'endstock'
   */
   Vector         machines;
   public int     instock [];
   public int     endstock [];

   public Factory ()
   {
      int    i;

      parent.traceOut (1, "New Factory");
      parent = (TestHarness)TestHarness.GetThis();
```

创建对象:

```
machines = new Vector();
instock = new int [parent.nummonths];
endstock = new int [parent.nummonths];
```

初始化数组 instock 和 endstock:

```
parent.traceOut
    (1, "Factory:Pre-for loop:i < parent.lastmonth=" +
    parent.lastmonth);
for (i = 0; i < parent.nummonths; i++)
{
    parent.traceOut (1, "start:i < parent.nummonths:i=" + i);
        instock [i] = 0;
    parent.traceOut (1, "instock [" + i + "]=0");
        endstock [i] = 0;
    parent.traceOut (1, "endstock [" + i + "]=0");
```

```
        parent.traceOut (1, "end:i < parent.nummonths");
    }
    parent.traceOut (1, "Factory:Post-for loop:i=" + i + " <
        parent.nummonths");
}
```

将新建的**:Machine**对象加入向量 machines:

```
public void addMachine (Machine newMC)
{
    parent.traceOut (1, "Factory:addMachine");
    machines.add (newMC);
}
```

返回 machine 对象向量中的机器数:

```
        public int howManyMachines ()
        {
            int result = machines.size ();
            parent.traceOut
            (1, "Factory:howManyMachines:result=" + result);
        return result;
    }
}
```

3. Machine.java

```
import java.lang.*;
import java.text.*;
import java.util.*;

public class Machine
{
```

适用于每个**:Machine**对象的默认信息:

```
private static final double cost = 2000;
private static final int output = 10;
private static final double overhead = 400;
private static final double rawcost = 220;
```

每当创建一个新的**:Machine**对象时，将**:Machine**对象的费用从**:CashAccount**对象中减去。然后，将这个新的**:Machine**对象加入**:Factory**对象目录:

```
public Machine ()
{
    parent.traceOut (1, "New Machine");
    TestHarness parent = TestHarness.GetThis();
    parent.traceOut (1, "Machine:Pre-cash.debit (" + cost + ")");
    parent.cash.debit (cost);
    parent.traceOut (1, "Machine:Pre-addMachine");
    parent.factory.addMachine (this);
}
```

返回静态的一台机器的成本值：

```
static public double mccost ()
{
    parent.traceOut (1, "Machine:mccost:return cost");
    return cost;
}
```

返回静态的一台机器的生产值。这是一台机器在一个生产周期中能生产出的产品单元的最大数量：

```
static public int mcoutput ()
{
    parent.traceOut (1, "Machine:mcoutput:return output");
    return output;
}
```

返回静态的一台机器的日常费用值。这是每一台机器的日常费用，用于计算生产周期中的费用：

```
static public double mcoverhead ()
{
    parent.traceOut (1, "Machine:mcoverhead:return overhead");
    return overhead;
}
```

返回静态的原材料的费用值，用于计算生产周期中的费用：

```
static public double mcrawcost ()
{
    parent.traceOut (1, "Machine:mcrawcost:return rawcost");
    return rawcost;
}
}
```

4. 脚本文件——预期的输出

在下面的表中，左边显示的是传给测试装备的脚本，右边显示的是预期的输出：

脚本	预期的输出
0: Create a CashAccount object 1: CashAccount: 1: new: -:	New CashAccount instance created Pre-for loop: i < parent.nummonths = 6 start: i < parent.nummonths: i = 0 balance [0] = 0.0 end: i < parent.nummonths start: i < parent.nummonths: i = 1 balance [1] = 0.0 end: i < parent.nummonths start: i < parent.nummonths: i = 2 balance [2] = 0.0

	end: i < parent.nummonths start: i < parent.nummonths: i = 3 balance [3] = 0.0 end: i < parent.nummonths start: i < parent.nummonths: i = 4 balance [4] = 0.0 end: i < parent.nummonths start: i < parent.nummonths: i = 5 balance [5] = 0.0 end: i < parent.nummonths Post-for loop: i = 6 < parent.nummonths
0: To set the balance of the current month 1: CashAccount: 5: setBalance: 2000.0:	CashAccount: setBalance-2000.0 CashAccount: setBalance-balance = 2000.0
0: Create a Factory object 2: Factory: 1: new: -:	New Factory Factory: Pre-for loop: i < parent.lastmonth = 5 start: i < parent.nummonths: i = 0 instock [0] = 0 endstock [0] = 0 end: i < parent.nummonths start: i < parent.nummonths: i = 1 instock [1] = 0 endstock [1] = 0 end: i < parent.nummonths start: i < parent.nummonths: i = 2 instock [2] = 0 endstock [2] = 0 end: i < parent.nummonths start: i < parent.nummonths: i = 3 instock [3] = 0 endstock [3] = 0 end: i < parent.nummonths start: i < parent.nummonths: i = 4 instock [4] = 0 endstock [4] = 0 end: i < parent.nummonths start: i < parent.nummonths: i = 5 instock [5] = 0 endstock [5] = 0 end: i < parent.nummonths Factory: Post-for loop: i = 6 < parent.nummonths
0: Display the last monthly balance 1: CashAccount: 2: balanceByIndex: 5:	CashAccount: balanceByIndex-balance [5] = 2000.0

0: Obtain the static information 3: Machine: 2: mccost: - 3: Machine: 3: mcoutput: -: 3: Machine: 4: mcoverhead: -: 3: Machine: 5: mcrawcost: -:	 Machine: mccost: return cost processMachine: mccost = 2000.0 Machine: mcoutput: return output processMachine: mcoutput = 10 Machine: mcoverhead: return overhead processMachine: mcoverhead = 400.0 Machine: mcrawcost: return rawcost processMachine: mcrawcost = 220.0
0: Return the number of machines already purchased 2: Factory: 5: howManyMachines: -: 0: Purchase a machine 3: Machine: 1: new: -:	Factory: howManyMachines: result = 0 processFactory: howManyMachines = 0 New Machine Machine: Pre-cash.debit (2000.0) CashAccount: debit-balance = 2000.0 money 2000.0 CashAccount: debit-balance = 0.0 Machine: Pre-addMachine Factory: addMachine
0: Re-display the last monthly balance 1: CashAccount: 2: balanceByIndex: 5:	CashAccount: balanceByIndex-balance [5] = 0.0
0: Return the number of machines now purchased 2: Factory: 5: howManyMachines: -:	Factory: howManyMachines: result = 1 processFactory: howManyMachines = 1
:	< end of script >

6.6 图形用户界面测试

前面的测试集中在应用程序的功能方面的测试。接下来的测试将集中在图形用户界面(GUI)的视觉与感觉方面的测试。当确定了GUI视觉和感觉无误后，再下一阶段的测试就用这新的GUI取代以前的测试装备。

下面的两个例子覆盖了基本窗口的测试内容。基本窗口通过一个菜单选项与应用程序进行最低程度的交互。全面的交互将在本章后面的“系统测试”部分介绍。

6.6.1 基本窗口测试

设计该测试用于确认GUI看上去符合编程者的要求，并且窗口能被关闭。完整的窗口如图6-1所示。

1. SimCo.java

```
public class SimCo
{
public static SimCo          simco;
public        Display        disp;
```

有6个月的记录被保留：

```
public final int nummonths = 6;
```

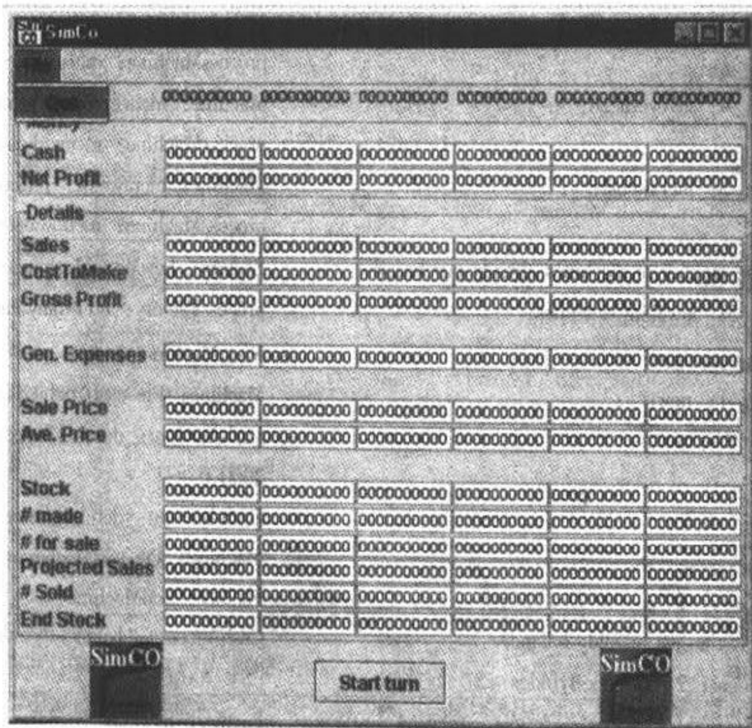


图6-1　完整的SimCo窗口

创建**:Display**对象的主类方法如下：

```
public SimCo ()
{
    simco = this;
    disp = new Display("SimCo");
}
```

下面是启动应用程序的方法：

```
public static void main(String[] args)
{
    SimCo co = new SimCo ();
}
```

全局对象方法如第5章所描述，它允许**:SimCo**对象可从任何地方被引用，代码如下：

```
public static SimCo GetThis ()
    {
        return simco;
    }
}
```

2. Display.java

```
public class Display    extends Jframe
                        implements ActionListener
{
    SimCo          parent;
```

声明菜单选项：

```
JMenuBar       menuBar;
JMenu          file;
JMenuItem      miQuit;
```

其余的类变量声明有：

- ▶ 声明数据组。
- ▶ 声明文本域数组，每个数组有6个元素。
- ▶ 声明图标。

下面是类 Display 的主方法：

```
public Display(String title)
{
    super (title);
```

窗口监听者将被 Quit 按钮激活：

```
addWindowListener(new WindowAdapter ()
{
    public void windowClosing(WindowEvent we) { System.exit (0); }
});
```

从主类的**:SimCo**对象获得月数：

```
count = SimCo.GetThis().nummonths;
```

创建 menuBar 并将其加到应用程序框架（frame）：

```
menuBar = new JMenuBar ();
setJMenuBar (menuBar);
```

创建菜单按钮并将它加到菜单条：

```
file = new JMenu ("File");
menuBar.add (file);
```

创建一个菜单的下拉项，加上一个动作监听者（action listener），加上一个动作关键词，然后将动作关键词加到菜单按钮：

```
miQuit = new JMenuItem ("Quit");
miQuit.addActionListener (this);
miQuit.setActionCommand ("QUIT");
file.add (miQuit);
```

主显示随 JPanel 的创建启动，JPanel 包含其余的可视元素：

```
months = new JPanel ();
months.setLayout(new GridBagLayout());
months.setBorder(empty);
```

创建下列各项：

▶ 标签条，并且给它定位。

▶ 与月份相联系的文本域。

▶ 包含有现金和利润的货币组（money panel）。

▶ 包含有其余数据的细目组（details panel）。

▶ 图标按钮，并将这些按钮置于“Start turn”按钮周围。

将所有各项组合在一起，利用窗格目录（content pane）的 BorderLayout 设置窗口图标。

下面是 Quit 菜单按钮的动作方法的实现：

```
    public void actionPerformed (ActionEvent e)
    {
        String str = e.getActionCommand ();

        if (str.compareTo (new String ("QUIT")) == 0)
        {
            System.exit (0);
        }
    }
}
```

6.6.2 使用菜单

本节展示实现购置一台机器设备所需的代码。其功能与整体测试的例子中的功能完全相同，我们将在后面给出测试输出的比较。

新的用户窗口如图 6-2 所示。

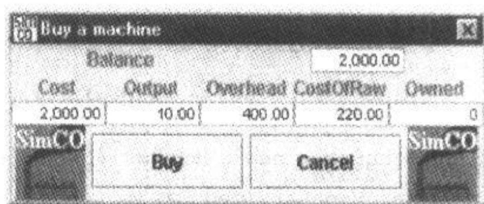


图 6-2 购置机器（Buy a machine）窗口

1. 修改后的 SimCo.java

作为类声明的一部分，加入对两个新对象的支持，

```
public      CashAccount     cash;
public      Factory         factory;
```

数组的长度是 6（nummonths），最后一个元素是 # 5：

```
public final int lastmonth = 5;
```

在主类的方法中，创建**cash：CashAccount**对象和**factory：Factory**对象，将 cash：CashAccount 初始化为 2000.00：

```
cash = new CashAccount();
cash.setBalance (2000.00);
factory = new Factory ();
```

2. 修改后的 Display.java

加入两个新的菜单选项 admin 和 miMachine 进行类声明，代码如下：

```
Jmenu        file, admin;
JmenuItem    miQuit, miMachine;
```

加入两个方法，用来格式化文本串，以便填入一个 10 个字符宽的文本域中：

```
DecimalFormat ten_two_f = new DecimalFormat("###,##0.00");
DecimalFormat ten_d = new DecimalFormat("##,###,###");
```

修改主类的字符串以支持新的菜单选项：

```
public Display(String title)
{
    ...
    parent = SimCo.GetThis();
    count = parent.nummonths;
    ...
    admin = new JMenu ("Admin");
    menuBar.add (admin);
    ...
    miMachine = new JMenuItem ("Machine");
    miMachine.addActionListener (this);
    miMachine.setActionCommand ("MACHINE");
    admin.add (miMachine);
    ...
}
```

下面是更新 cash 文本域的方法的实现代码：

```
public void updateCash ()
{
    cashV [parent.lastmonth].setText
        (ten_two_f.format
            (parent.cash.balanceByIndex (parent.lastmonth)));
}
```

修改该方法以支持“Machine”菜单项：

```
public void actionPerformed (ActionEvent e)
{
   String str = e.getActionCommand ();

   if (str.compareTo (new String ("QUIT")) == 0)
   {
```

```
            System.exit (0);
        }
        else if (str.compareTo (new String ("MACHINE")) == 0)
        {
            purchaseMachine ();
        }
}
```

加入响应“Machine”菜单按钮被按下的新方法：

```
public void purchaseMachine ()
{
    final JDialog machine = new JDialog (this, "Buy a machine",
        true);
```

创建窗口内容和动作按钮：

```
JButton buyB = new JButton ("Buy");
buyB.addActionListener (new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        Machine newMC = new Machine ();
        machine.hide ();
        updateCash ();
    }
});
```

制定一个“Cancel”按钮，当无购置行为时，用它关闭窗口：

```
JButton cancelB = new JButton ("Cancel");
cancelB.addActionListener (new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        parent.traceOut (1, "purchaseMachine:cancelled");
        machine.hide ();
    }
});
```

载入窗口时提供有效信息：

```
cashV.setText (ten_two_f.format
    (parent.cash.balanceByIndex (parent.lastmonth)));
mccostV.setText (ten_two_f.format (Machine.mccost ()));
mcoutV.setText (ten_two_f.format (Machine.mcoutput ()));
mcohV.setText (ten_two_f.format (Machine.mcoverhead ()));
mcrawV.setText (ten_two_f.format (Machine.mcrawcost ()));
ownedV.setText (ten_d.format (parent.factory.howManyMachines ()));
```

3. 测试输出的比较

相似的测试的响应结构见下表。左边展示的是激活用户界面的脚本的文字输出，右边展示的是对用户交互的响应。

整体输出	GUI菜单测试输出
…	…
Machine：mccost：return cost	Machine：mccost：return cost
ProcessMachine：mccost = 2000.0	
(TestHarness output)	
Machine：mcoutput：return output	Machine：mcoutput：return output
ProcessMachine：mcoutput = 10	
(TestHarness output)	
Machine：mcoverhead：return overhead	Machine：mcoverhead：return overhead
ProcessMachine：mcoverhead = 400.0	
(TestHarness output)	
Machine：mcrawcost：return rawcost	Machine：mcrawcost：return rawcost
ProcessMachine：mcrawcost = 220.0	
(TestHarness output)	
Factory：howManyMachines：result = 0	Factory：howManyMachines：result = 0
ProcessFactory：howManyMachines = 0	
(TestHarness output)	
	CashAccount：balanceByIndex-balance [5] = 2000.0
	(action method checking the cash balance)
	Machine：mccost：return cost
	(action method checking machine cost)
New Machine	New Machine
Machine：Pre-cash.debit (2000.0)	Machine：Pre-cash.debit (2000.0)
CashAccount：debit-balance = 2000.0 money 2000.0	CashAccount：debit-balance = 2000.0 money 2000.0
CashAccount：debit-balance = 0.0	CashAccount：debit-balance = 0.0
Machine：Pre-addMachine	Machine：Pre-addMachine
Factory：addMachine	Factory：addMachine
CashAccount：balanceByIndex-balance [5] = 0.0	CashAccount：balanceByIndex-balance [5] = 0.0
Factory：howManyMachines：result = 1	
(Script output)	
ProcessFactory：howManyMachines = 1	
(TestHarness output)	

4. GUI中的出错处理

通过检查花费在新机器上的资金的数额，你可以在“Machine”窗口加入出错处理。

```
buyB.addActionListener (new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        /*
        ** check the amount of money available against the cost
        ** of a new machine
        */
        if (parent.cash.balanceByIndex (parent.lastmonth) <
            Machine.mccost ())
        {
            parent.traceOut (1, "purchaseMachine:not enough money");
            JOptionPane.showMessageDialog
                (Display.this, "Not enough money");
        }
        else
        {
            /*
            ** create a new machine, hide this window and
            ** update the cash text field to reflect the new balance
            */
            Machine newMC = new Machine ();
            machine.hide ();
            updateCash ();
        }
    }
}
```

6.7 强度测试

可以也应该在测试的任何阶段做强度测试。它的基本假设是，将每一组件进行大量测试。

举例说，在强度测试中反复创建和清除一个复合对象。如果一个简单的创建-清除循环会产生1个字节的内存泄露，这个内存泄露几乎是感知不出来的。但是，在一个单项测试中，这个创建-清除循环被重复成千上万次后，则内存泄露就会达到一个能辨别出的量。

强度测试应集中在应用程序中那些资源被创建和清除的地方，并且每个测试都集中于某个特性。例如，如果一个应用程序管理它自己的资源，无论是内存、网络连接还是其他资源，强度测试用来显露那些系统不能最佳利用这些资源、有时甚至产生灾难性结果的场合。这些问题可以显示出系统管理的算法有缺陷。

6.8 系统测试

再下一步的测试就是移去测试装备，测试整个系统，最终，整体的结果可在应用程序的主窗口中可见，它显示在图6-3中。在这些单元中，较近的月份的信息处在较靠右边的列中。在

内容为 Months（月份）信息的行中，前5个单元的内容为“…”，第6个的内容为“0”，这表明应用程序在第0月启动，Cash单元被初始化为2000.00。其他惟一含有数值的单元就剩下一件商品的平均卖出价。

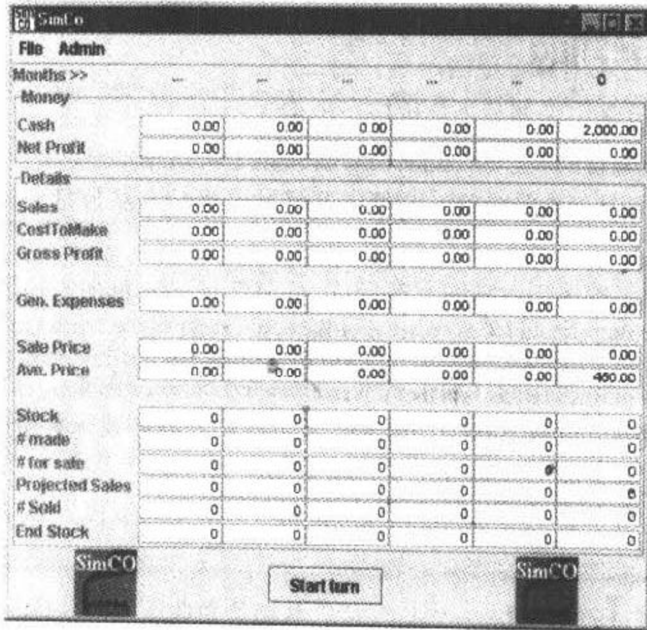


图6-3　在第0月启动的SimCo主窗口

用户为了启动生产周期，他需要更多的资金。为了得到更多的资金，可用Loan菜单按钮要求贷款，然后出现下一个窗口（见图6-4），其内容如下：

（1）用户输入申请的贷款数额及打算还贷的年限。本例中的贷款数额为3000，还贷年限为3年。

（2）用户可按OK按钮确认借贷，或按Cancel按钮取消申请。

（3）如果按下了OK按钮，则对应当前月的Cash单元中的数字将增加到5000，表示贷款已加入。

现在用户就有足够的资金了，他们现在需要做的是购置一台机器设备以制造供出售的产品。为此，用户可利用Machine菜单按钮，将会出现如图6-5所示的窗口。用户有两种选择：购置一台设备（按下Buy按钮）或取消该窗口（按下Cancel按钮）。如果选择的是购置设备，则当前月的Cash单元中的数字将减至3000。

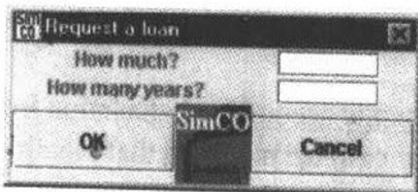


图6-4　要求贷款窗口

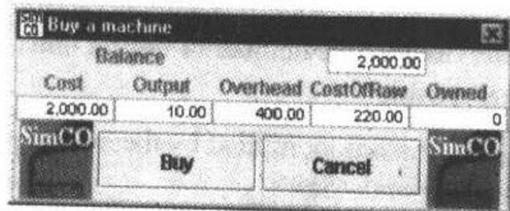


图6-5　购置机器窗口

拥有足够的资金和设备之后，用户就可以通过按下“Start turn”按钮启动一个生产周期。当

该按钮被按下后，“Next turn”窗口出现，如图6-6所示。

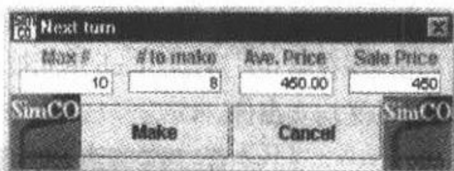


图6-6　下一轮生产周期窗口

窗口中显示了可以被生产出的产品的最大数量（该数字是基于所拥有的设备和各台机器的潜在产量算出的）和每件产品的平均出厂价。本窗口还包含两个文本输入域，第一个用来输入用户希望的产量，第二个是每件产品的售价。售价是用来调节平均价格，平均价格用来决定销售量，所以用户应尽可能地使售价接近平均价格。

下面的列表给出的是月度会计核算，它覆盖从可被销售的产品数量、到通过月度的净盈亏来调整的现金结余。

- 上一个生产周期未售出的产品库存数量（“Stock”单元）再加上当前生产周期的产品数量（“# made”单元），等于当前生产周期的产品待售数量（“# for sale”单元）
- 基于售价和平均价格所作的计划销量（“Projected Sales”单元），用于指示销售趋势。
- 从产品的待售数量（“# for sale”单元）减去已售产品的实际数量（“# sold”单元），给出下一个销售周期的库存量（“# End Stock”单元）
- 已售产品的数量（“# sold”单元）乘以售价（“# Sale Price”单元）给出当月的总销售收入（“# Sales”单元）
- 产品的制造成本显示在“CostToMake”栏，从销售总收入（“Sales”栏）中减去制造成本，给出当月的毛利润（“Gross Profit”栏）
- 显示在“Gen. Expenses”栏中的是总支出（借贷成本加上总的管理成本），从毛利润（“Gross Profit”栏）中减去总支出，给出当月的纯利润（“Net Profit”栏）
- 将纯利润加入上月的现金量（Cash-month 0），给出本月的现金量（Cash-month 1）。

当一个生产周期结束后，应用程序的主窗口显示如图6-7所示。

SimCo

File　Admin

Months >>				...	0	1
Money						
Cash	0.00	0.00	0.00	0.00	3,000.00	2,429.08
Net Profit	0.00	0.00	0.00	0.00	0.00	-570.92
Details						
Sales	0.00	0.00	0.00	0.00	0.00	2,700.00
CostToMake	0.00	0.00	0.00	0.00	0.00	1,760.00
Gross Profit	0.00	0.00	0.00	0.00	0.00	940.00
Gen. Expenses	0.00	0.00	0.00	0.00	0.00	1,510.92
Sale Price	0.00	0.00	0.00	0.00	0.00	460.00
Ave. Price	0.00	0.00	0.00	0.00	460.00	462.00
Stock	0	0	0	0	0	0
# made	0	0	0	0	0	8
# for sale	0	0	0	0	0	8
Projected Sales	0	0	0	0	8	5
# Sold	0	0	0	0	0	6
End Stock	0	0	0	0	0	2

SimCO　Start turn　SimCO

图6-7　经过一个生产周期后的SimCo主窗口

6.9 规模测试

与强度测试相似，规模测试（Scalability Testing）也将测试其最大潜力。这个阶段的测试是在系统的最终版本上进行的。这些测试包括将尽可能多的用户连接到系统上（多个用户同时访问一个网站或数据库)。这些测试也可以是通过使系统负载尽可能多执行活动来进行。例如一个股票市场交易系统，通过并发测试确定系统可以在线处理多少个股票市场交易而不致使系统瘫痪或与用户断开。

6.10 回归测试

回归测试是对产品的新版本连续地重复旧版本曾进行过的相同测试，期待能出现相同的结果。如果结果有异，这表明新版本引入了以前未曾有的错误，或是代码发生退步，错误被重新引入。

在改进产品质量的过程中，回归测试是十分重要的。另外。应用程序系统的开发者也应该采取回归测试，他们必须保证对系统所进行的任何工作，无论是修改已发现出的缺陷，或是加入新的功能，都不能对原应用程序系统造成任何损害。

对产品添加任何新的功能时，必须在原有的回归测试基础上进行新的回归测试，这一点是极为重要的。如果在产品的第六个版本中只进行相对于第一个版本功能的回归测试，则这个测试几乎是毫无用处的，因为有五个版本的功能没有被测试，对它们中的任何一个进行回归测试都可能会不合格，直到移交质量保证部门才会被察觉。

6.11 小结

本章涵盖了有关应用的测试中的许多不同层次，但还不仅限于此。本章首先引进了作为测试一个应用程序的手段的测试装备，它可以用来遍历一个应用程序中每一个对象中的所有方法。在分别测试了每一个方法之后，测试装备可以将类作为一个整体进行测试。测试装备的最后用途是对类的组合进行整体测试。再下一阶段的测试就是针对用户界面，测试用户界面如何同应用程序进行交互。在本章的结尾，讨论了强度测试、系统测试、并发测试和回归测试。

第7章　调　　试

本章将讨论以下内容：

- 解释使用调试工具的目的
- 展示三种普遍使用的调试工具（DBK，GDB和JDB）的基本命令
- 给出两个调试工具的使用例子

调试工具是设计用于帮助程序员了解应用程序是如何工作的、以及如何最好地修改任何运行时的错误（bug）。调试工具提供一个测试环境，在这个环境中应用程序可以在用户的控制下运行。它还允许用户在应用程序运行过程中检查变量取值的变化。

正常情况下，如同在测试过程中所做的，在源代码中加入输出语句就足以确定一个应用程序是如何工作的。输出语句能够确定一段代码是否被走到，或者如果该段代码被执行，生成的结果是什么。

然而，如果应用程序非常复杂，或者已经执行了一段时间，输出的潜在规模十分巨大，那么，这样的做法就不太可取，除非有特别的理由。一个可能的理由是：程序员想检查该应用程序在不同的平台上，或是大范围地进行了重新编码后，它能否以同样的方式执行。输出结果可以被存档，并在以后被用来作回归测试。

不使用输出语句的其他原因有：

- 加入新的输出语句可能要求应用程序被重建，那将会是很耗时的。
- 输出语句仅能告诉用户什么代码已被执行，但不能说明它为什么被执行。有时候了解代码为什么不被执行比了解什么代码已被执行更重要。

本章参考三个最常用的调试工具，重点讨论一些常用的命令。调试工具通常都与一个编译器打包在一起。例如，Java编译器javac有一个对应的调试工具jdb。作为参考的调试工具有：dbx，一个典型的UNIX调试工具；gdb，随LINUX发行、与GNU工具套件一起被提供的调试工具；以及jdb，一个Java调试工具。本章的最后给出一个用C++写成的例子，在调试这个例子时，使用了两种调试工具：dbx和gdb。

任何一个调试工具都具有如下的基本功能：

- 提供对一个运行中的应用程序的内部运行机制的访问。
- 允许程序员单步跟踪应用程序以确定它的流程。
- 允许检查一个应用程序的方法调用栈。方法调用栈能显示从主程序到当前方法的调用层次。这将在本章中稍后的“检查应用程序”中讨论。
- 允许检查一个应用程序的破坏文件（核心文件）。在UNIX环境中，核心文件是一个应用程序由于一个服务器错误导致应用程序终止的结果。它包含显示应用程序崩溃时的

状态的信息，包括方法调用栈。

- 允许程序员检查应用程序中的变量的值。
- 允许程序员检查与应用程序相关的内存的内容。

7.1 使用调试工具前的准备

传给调试工具的应用程序文件是一个由调试工具编译的目标（可执行）文件。当编译应用程序时，使用“-g”（生成符号表）命令通知编译器在应用程序中装入调试信息。

7.2 启动调试工具

调试应用程序，可以使用下面的三种方式：

- 启动调试工具，然后在调试工具的环境中启动应用程序。如果试图在问题发生之前确定它的产生原因，可以使用这种方式。
- 在应用程序运行（或似乎死机）时，将其与调试工具联在一起。例如，一个已运行了几个小时的应用程序看上去像停止执行了一样，这种方式可以用来确定应用程序的当前状况。
- 如果应用程序已经崩溃，并且生成了一个核心文件，那么，调试工具可被用于检查崩溃的起因。

7.2.1 首先启动调试工具

当试图确定应用程序中问题的起因，或试图跟踪应用程序的逻辑流程时，可以与应用程序一起启动调试工具。

dbx	dbx <应用程序名>
gdb	gdb <应用程序名>
jdb	jdb <应用程序名>

7.2.2 将调试工具联上运行中的应用程序

有时候，应用程序只是在调试工具以外运行时才会显出它的问题。对于这种现象的解释是：当应用程序在调试器中运行时，调试器降低应用程序的运行速度，因为它需要实时地跟踪程序的执行，这可能掩盖一些对时间敏感的问题。在这种情况下，可以在问题即将发生前或已发生后，通过将应用程序与调试器联在一起来调试这些问题。尽管这可能不允许你完全跟踪问题的原因，但能给你问题发生时的足够信息，如变量值和调用栈等。在将调试器工具与应用程序联在一起之前，你需要具有向应用程序发送“**kill**”命令的权限。“**kill**”命令用于向应用程序发送信号并将其中断。

下面给出的步骤概括了如何将调试器联到一个正在运行的应用程序：

(1) 使用“**ps**”命令确定进程的 ID。

(2) 如果你有“kill”应用程序的权限，那么，你可以将上述的进程 ID 作为参数来运行调试工具。对于不同的调试器，表 7-1 给出了需要执行的具体指令。

(3) 然后，调试器中断进程，确定目标文件的全名，读入符号信息，并提示输入下面的命令（表7-1）：

表7-1 将调试工具联上运行中的应用程序的命令

调试工具	命 令
AIX - dbx	dbx -a <进程ID>
Solaris - dbx	dbx - <进程ID>
gdb	gdb <应用程序名>
	(gdb) attach <进程ID>

在Java中也可以这样做，但会稍稍复杂一些，在此不再讨论。

人为暂停一个应用程序

有时，一个应用程序是被另一个应用程序启动的，例如，一个应用程序被Internet浏览器启动。这种应用程序的启动和运行将不受用户的干预，所以要想将它联上调试工具非常困难。这里给出的技巧可以人为暂停一个应用程序，以便利用ps命令进行定位，并联上调试工具。为了暂停一个应用程序，可以在应用程序中插入一段能产生循环的代码。当联上调试工具后，该段代码可以被找到，且循环变量的值（在下面的例子中为'xxx'）可被改变为任何可以中断循环的值。在下面的例子中，因为变量'xxx'的值为1，所以代码会循环，将'xxx'的值改变为0，则将中断循环。

```
// the following lines are added to pause the application
int xxx=1;
while (xxx == 1)
{
   sleep(5);
}
// pause code ends here
```

7.2.3 使用调试工具和核心文件

如果一个应用程序崩溃了，一般地它会创建一个核心文件。该文件非常有用，因为它包含了关于应用程序崩溃时正在做什么的信息。它通过显示应用程序调用栈以及在不同层次的栈的变量值展示出这些信息。它在调试工具dbx和gdb中的用法见表7-2。

7.3 调试工具的子命令

下面列出的命令是从所有可用命令中选出的，是使用最广泛的命令。如需更多的信息和帮

表7-2 使用调试工具和核心文件的用法

调试工具	命 令
dbx	dbx <应用程序名> core
gdb	gdb <应用程序名> core

注：Java调试工具不支持核心文件的使用。

助，可查阅系统帮助文件，或启动调试工具后键入'help'。Help命令在本章介绍的三种调试

工具中都是可用的。Help 命令显示一个列表，它包含调试工具支持的命令及对应的简单说明。

每个调试工具都有下面列出的五大类的命令：

- ▶ 使应用程序停止
- ▶ 运行应用程序
- ▶ 检查应用程序
- ▶ 检查数据
- ▶ 逐行控制

7.3.1 使应用程序停止

断点用来控制应用程序的执行。它们可用于在某些执行点上中断应用程序，以便检查一些相关的变量值和内存的内容。

1. 设置断点

在调试器中，断点可以设置在行号或设置在一个方法的第一条指令，例如：

gdb	在指定行或指定方法设置断点
	break <filename>: 54
	在指定文件的第 54 行设置断点。文件名默认时，就是当前文件
	break <className>.<methodName>
	在指定类的指定方法的第一行设置断点
dbx，jdb	**stop at <className>: 22**
	在源文件（包含指定的类）的第 22 行的第一条指令设置断点
	(dbx) **stop in <className>::<methodName>**
	(jdb) **stop in <className>.<methodName>**
	在指定类的指定方法的第一行设置断点

2. 清除断点

dbx	**status** 列出所有断点
	clear 清除位于指定的源文件行的断点
	delete <argument> 清除由参数指定的编号的断点。断点的编号可由
	status 命令显示
gdb	**clear** 清除位于指定行或函数的断点
	delete 清除一些断点或自动显示的表达式
jdb	**clear <classname>.<method>**
	clear <classname>: <line #>
	clear 列出当前所有的断点

3. 另外的断点命令

gdb	**disable** 禁止一些断点
gdb	**enable** 允许一些断点

7.3.2 运行应用程序

下面给出的命令用于在调试器内执行或终止应用程序。

1. 退出调试工具

dbx, gdb, jdb	**quit** 终止调试工具
gdb	**kill** 终止被调试的应用程序的执行

2. 运行应用程序

dbx, gdb, jdb	**run** 在启动了调试器和设置了必要的断点后，可以用该命令来启动被调试的应用程序。该命令可伴随传给应用程序的参数 **run numPhils = 10** 该命令将参数"numPhils = 10"传给应用程序。应用程序会解析该参数以决定该做什么

3. 重新运行应用程序

dbx	**rerun** 用先前的参数开始应用程序的执行
gdb	**run** 用先前的参数开始应用程序的执行

7.3.3 检查应用程序

下面的这些命令用于检查应用程序的流程。

1. 打印栈

gdb	**backtrace/bt** 打印所有的栈桢
dbx	**where** 显示活动的过程和函数的列表
jdb	**where** 不带参数，打印当前线程的栈 **where all** 打印当前线程组中所有线程的栈 **where <threadindex>** 打印指定线程的栈
	如果当前线程被挂起（通过一个事件，如断点，或通过挂起命令），可以用 **Print** 命令和 **dump** 命令显示局部变量和域（field）。用 **up** 命令和 **down** 命令选择当前的栈桢

2. 向下移动栈

dbx, gdb, jdb　　**down** 选择并打印被当前栈桢调用的栈桢。向下移动应用程序栈指针，直到被当前方法调用的方法

3. 向上移动栈

dbx, gdb, jdb　　**up** 选择并打印调用当前栈桢的栈桢。向上移动应用程序栈指针，直到调用当前方法的方法

4. 显示源代码

(dbx)	**list** 从发出请求的行开始，显示 10 行。如果当前行是 36，则 list 显示 36 - 45 行 再次键入 list 将显示 46 - 54 行 **list 36，50** 列出行 36 - 50
(gdb)	**list** 以当前行为中心，显示 10 行。如当前行是 36，则 list 显示行 31 - 40 再次键入 list 将显示 37 - 46 行 **list 36，50** 列出行 36 - 50

（续）

(jdb)	**list** 以当前行为中心，显示10行。如当前行是19，键入**list**将显示行15-24 再次键入**list**将重新显示15-24行 **list 30** 显示行26-35 list总是返回显示当前行。所以如果当前行是19，则再次键入**list**将显示行15-24

7.3.4 检查数据

下面的命令用于检查正在运行的应用程序中的数据。

1. 倒出变量的内容

dbx	**dump** 显示指定过程中的变量名称及其值
jdb	**dump** 对于原型（primitive）值，该命令等同于**print**命令。对于对象，它将打印定义在对象中的每个域的当前值。包括静态域和实例域

2. 打印变量的内容

说明：要想显示局部变量，包含该变量的类必须带“-g”选项进行编译。

gdb	**print** 打印表达式EXP的值 **inspect** 等同于**print**命令 **output** 与**print**命令相似，但不提交值的历史记录，也不打印新行
dbx，jdb	**print** 显示对象及原型值。对于原型的变量或域，打印实际值。对于对象，打印一份简短描述 （dbx，jdb）print也可用来运行一个过程并打印返回代码 （jdb）想得到关于一个对象的更多的信息，可参见下面的dump命令
jdb	**printf** C风格的printf格式串

3. 改变变量的值

下面的命令可用于在应用程序运行的过程中修改一个变量的当前值。它的用处体现在以下的两个方面：

- 看看如果给变量不同的值，将会有什么情况发生。
- 为了改变循环中的条件，使得应用程序可以继续。该技术用于启动应用程序，然后在应用程序上附入一个调试工具。

dbx	**assign** <**var**> = <**expr**> 计算表达式‘expr’的值并将结果赋给变量‘var’
gdb	**set** <**var**> = <**expr**> 计算表达式‘expr’的值并将结果赋给变量‘var’
jdb	**set** <**var**> = <**expr**> 计算表达式‘expr’的值并将结果赋给变量‘var’

4. 显示数据类型

dbx，gdb　　**whatis var** 打印表达式‘var’的数据类型。

5. 检查内存

所用的格式为

- **10c** 意即显示 10 字节的内存。'c' 也是表明以字符格式显示内存的内容。
- **10x**（**dbx**）意即显示 10 个字（20 个字节）的内存——参见下面的例子。'x' 表明以十六进制格式显示内存。
- **10x**（**gdb**）意即显示 10 个字（40 个字节）的内存——参见下面的例子。

AIX - dbx	0x2ff2led8 /10c
Solaris - dbx	examine 0x2ff2led8/10c
gdb	x/10c 0x2ff2led8

7.3.5 确定逐行控制

可以用断点来控制大规模的应用程序流程。下面的命令以逐行中断为基础，确定应用程序流程。

1. 从断点处继续

dbx，gdb，jdb	**continue** 在断点、异常或单步执行后，继续执行被调试的应用程序，直到应用程序结束或遇到下一个断点。

2. 移到下一个语句

dbx，gdb，jdb	**next** 该命令将程序的运行前进到当前栈桢的下一行（跳过）。

3. 单步执行通过应用程序

dbx，gdb，jdb	**step** 该命令将程序的运行前进到下一行，无论它是在当前栈桢中或是一个被调用的方法（跳入）。

4. 结束方法

下面的这些命令继续执行应用程序，直到当前的方法返回到调用它的方法。

AIX -（dbx）	**Return**
Solaris -（dbx）	**Step up**
（jdb）	**Step up**
（gdb）	**Finish**

下面这个命令强制立即退出 gdb 方法：

gdb	**Return** 使选中的栈桢返回到它的调用者。

7.3.6 检查多线程应用程序

在某些操作系统中，单个程序可以有一个以上的执行线程。线程的精确语义对不同的操作系统是不同的。每个线程执行应用程序的一部分中属于该线程本身的副本。线程可以共享资源，但它们有自己的执行栈和可能的私有内存，以保证它们能够相互独立地运行。有些线程当需要必要的资源时就能马上获得，所以它们的执行比那些不得不等待的线程更快。

每个调试工具都提供了调试多线程应用程序的手段。调试时，调试工具在同一时刻只在一个线程上有效，其他线程不被挂起，调试工具只能在同一时刻聚焦于一个线程。这个被聚焦的线程被称为当前线程，所以，调试命令是相对于当前线程显示信息的。如果在另外一个线程遇到一个断点，调试工具自动将焦点改变到那个线程。

1. dbx 的线程支持

```
(dbx) thread
current thread ($thread) is t@1
(dbx) help thread
thread (command)
thread                      # Display current thread
thread <tid>                # Switch to thread <tid>.
```

在下面的变化中，省去的 <tid> 隐指当前线程。

```
thread -info [ <tid> ]      # Print everything known about the given
                              thread.
thread -hide [ <tid> ]      # Hide the given (or current) thread.
                              It will not show up in the generic
                              'threads' listing.
thread -unhide [ <tid> ]    # Unhide the given (or current). thread.
thread -unhide all          # Unhide all threads.
thread -suspend <tid>       # Keep the given thread from ever
                             running. A suspended thread shows up
                             with an 'S' in the threads list.
thread -resume <tid>        # Undo the effect of '-suspend'.
thread -blocks [ <tid> ]    # List all locks held by the given thread
                             which are blocking other threads.
thread -blockedby [ <tid> ] # Show which synchronization object the
                             given thread is blocked by, if any.
(dbx) threads
*>    t@1  a l@1  ?()   breakpoint             in main()
      t@2  b l@2  ?()   running                in _signotifywait()
      t@3  b l@3  ?()   running                in _lwp_sema_wait()
      t@4         ?()   sleep on (unknown)     in _swtch()
(dbx) help threads
threads (command)
threads                     # Print the list of all known threads.
threads -all                # Print threads normally not printed
                              (zombies).
threads -mode all|filter    # Controls whether 'threads' prints all
                              threads or filters them by default.
threads -mode auto|manual   # Under the GUI, enables automatic
                              updating of the thread listing.
threads -mode               # Echo the current modes.
```

2. jdb 的线程支持

选择一个线程作为当前线程。许多 jdb 的命令基于当前线程的设置。线程是由上述 **threads**

命令描述的线程索引（index）指定的。

```
threads [threadgroup]      -- list threads
thread <thread id>         -- set default thread
suspend [thread id(s)]     -- suspend threads (default: all)
resume [thread id(s)]      -- resume threads (default: all)
where [thread id] | all    -- dump a thread's stack
wherei [thread id] | all   -- dump a thread's stack, with pc info
kill <thread> <expr>       -- kill a thread with the given exception
                              object
interrupt <thread>         -- interrupt a thread
threadgroups               -- list threadgroups
threadgroup <name>         -- set current threadgroup
```

3. gdb 的线程支持

在所有线程或在某个特定的线程设置断点：

break <line-spec> - break <file>: 54 或 break <file>.<method>

---

break <line-spec> thread <thread # >

4. 例子

(1) 将循环代码加入应用程序。

(2) 在一个窗口中，运行应用程序。

(3) 在另一个窗口中，键入 **ps-efw**，显示活动的进程。

(4) 在同一个窗口中，用下面的命令启动调试工具。

```
$> gdb 'which <application>'
GNU gdb 4.18
Copyright 1998 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public
License, and you are welcome to change it and/or distribute
copies of it under certain conditions.
Type 'show copying' to see the conditions.
There is absolutely no warranty for GDB.  Type 'show warranty'
for details.
This GDB was configured as "i386-redhat-linux"...
(gdb)
```

(5) 将调试工具联上应用程序的进程的序号，该序号使用 <application> 的最顶层线程，如，29 397：

```
(gdb) attach 29397
Attaching to program: <application> 29397
Reading symbols from
<library>...done.
0x40aeb54e in __select () from /lib/libc.so.6
(gdb)
```

(6) 键入 **info threads**，确定使用哪一个线程。

(7) 设置断点：

```
(gdb) break <file>:<line #> thread <thread #>
Breakpoint 1 at 0x405ad791: file <file>, line <line #>.
```

说明：指定的行是加入到应用程序的循环代码的第一个语句：while (xxx = = 1)。

(8) 键入 **continue**：

```
(gdb) c
Continuing.
[Switching to Thread <internal thread #>]

Breakpoint 1, <method name> (<arguments>)
 at <file>:<line #>
<line #>     while (xxx == 1)
```

(9) 现在，你可以跳出循环，继续逐步运行应用程序：

```
(gdb) set xxx=0
(gdb) n
<the next line in the code after the looping code>
```

7.3.7 别名

别名是一个命令的快捷方式。这样的别名只在 dbx 工具中可用。

AIX - dbx	**alias l list** 使字符‘l’作为命令 list 的别名
Solaris - dbx	**alias p = print** 使字符‘p’作为命令 print 的别名

这些命令也可被放入一个 .dbxinit (AIX) 或 .dbxrc (Solaris) 文件中，dbx 每次运行时，这个文件被载入。

7.4 调试实例

在这个例子中，使用者可以亲身经历调试工具 dbx 和 gdb 支持的许多命令。当 dbx 工具的使用方法出现差别时，AIX 和 Solaris 两种形式均被给出。

这个例子得自第 8 章，它用于展示有关不同硬件体系结构的问题。

7.4.1 实例代码

以下为本例中使用的代码：

```
/*
**endian.cpp
**This application is used to test endism.
**
** Expected results:
**       Big Endian         Little Endian
**       12 34 56 78        78 56 34 12
**       12 34 56 78        34 12 78 56
**       41 42 43 00        41 42 43 00
*/
```

```
#include <stdio.h>
#include <string.h>

/*
** This structure allocates 3 variables each of 4 bytes
** Each variable is a different type, this allows the example
** to show how data is stored in memory
*/
struct endian {
    long l;
    short s[2];
    char c[4];
};

endian  temp;

/*
** This method is used to print the four bytes of each variable
** passed to it
*/
void printValue (char* ptr)
{
    int    i;

    for (i = 0; i < 4; i++)
    {
        printf ("%x ", *ptr++);
    }
    printf ("\n");
}

int main(int argc, char* argv[])
{
    char    *ptr;

/*
** These are the values that will be stored in memory
*/
   temp.l = 305419896;                          /* 0x12345678 */
   temp.s [0] = 4660;                           /* 0x1234 */
   temp.s [1] = 22136;                          /* 0x5678 */
   strcpy (temp.c, "ABC");

/*
** Set the pointer to the long variable
** and then print the 4 bytes of memory
*/
   ptr = (char *)&temp.l;
   printValue (ptr);
```

```
/*
** Set the pointer to the short array variable
** and then print the 4 bytes of memory
*/
   ptr = (char *)&temp.s;
   printValue (ptr);

/*
** Set the pointer to the char array variable
** and then print the 4 bytes of memory
*/
   ptr = (char *)&temp.c;
   printValue (ptr);

   return 0;
}
```

7.4.2 使用调试工具

这一部分显示将在第 8 章中讨论的代码实例 endian。我们对这个实例使用两种调试工具 dbx 和 gdb。每一步及其显示给用户的输出都被记录下来。

步骤 1：启动调试工具：

```
$ > dbx endian
```

```
$ gdb endian
```

设置断点，使得应用程序能在‘main’的第一个语句停止：

```
(dbx) stop in main
(2) stop in main
```

```
(gdb) break main
Breakpoint 1 at 0x80487ce: file endian.cpp, line 48.
```

运行应用程序：

```
(dbx) run
Running: endian
(process id 21096)
stopped in main at line 48 in file "endian.cpp"
48        temp.l = 305419896;        /* 0x12345678 */
```

```
(gdb) run
```

```
Starting program: /home/jasmine/endian/endian
Breakpoint 1, main (argc = 1, argv = 0xbffffaa4) at endian.cpp: 48
48        temp.l = 305419896;          /* 0x12345678 */
```

当应用程序在‘main’中停止时，列出这个断点附近的代码：

```
(dbx) list
48        temp.l = 305419896;          /* 0x12345678 */
49        temp.s [0] = 4660;           /* 0x1234 */
50        temp.s [1] = 22136;          /* 0x5678 */
51        strcpy (temp.c, "ABC");
52
53        /*
54         ** Set the pointer to the long variable
55         ** and then print the 4 bytes of memory
56         */
57        ptr = (char *) &temp.l;
```

```
(gdb) list
43        char *ptr;
44
45        /*
46         ** These are the values that will be stored in memory
47         */
48        temp.l = 305419896;          /* 0x12345678 */
49        temp.s [0] = 4660;           /* 0x1234 */
50        temp.s [1] = 22136;          /* 0x5678 */
51        strcpy (temp.c, "ABC");
52
```

列出更多的应用程序代码：

```
(dbx) l
58        printValue (ptr);
59
60        /*
61         ** Set the pointer to the short array variable
62         ** and then print the 4 bytes of memory
```

```
63        */
64        ptr = (char *) &temp.s;
65        printValue (ptr);
66
```

```
(gdb) l
53        /*
54        ** Set the pointer to the long variable
55        ** and then print the 4 bytes of memory
56        */
57        ptr = (char *) &temp.l;
58        printValue (ptr);
59
60        /*
61        ** Set the pointer to the short array variable
62        ** and then print the 4 bytes of memory
```

单步执行到下一个语句，即执行第48行的代码，显示第49行的代码：

```
(dbx) next
stopped in main at line 49 in file endian.cpp
49        temp.s [0] = 4660;        /* 0x1234 */
```

```
(gdb) next
49        temp.s [0] = 4660;        /* 0x1234 */
```

第48行向long型变量temp.l赋值。打印这个数值：

```
(dbx) print temp.l
temp.l = 305419896
```

```
(gdb) print temp.l
$1 = 305419896
```

要求调试工具确定temp.l的类型，以确认这个long型变量被正确处理：

```
(dbx) whatis temp.l
long l;
```

```
(gdb) whatis temp.l
```

```
type = long int
```

单步执行到下一个语句，即执行第49行的代码，显示第50行的代码：

```
(dbx) n
stopped in main at line 50 in file "endian.cpp"
50        temp.s [1] = 22136;        /* 0x5678 */
```

```
(gdb) n
50        temp.s [1] = 22136;        /* 0x5678 */
```

打印对整个变量的赋值。第49行向short型数组变量temp.s的第一个元素赋值；第二个元素显示为0：

```
(dbx) print temp.s
temp.s = (4660, 0)
```

```
(gdb) print temp.s
$2 = {4660, 0}
```

单步执行到下一个语句，即执行第50行的代码，显示第51行的代码：

```
(dbx) n
stopped in main at line 51 in file "endian.cpp"
51        strcpy (temp.c, "ABC");
```

```
(gdb) n
51        strcpy (temp.c, "ABC");
```

打印对整个变量的赋值。第50行向short型数组变量temp.s的第二个元素赋值：

```
(dbx) print temp.s
temp.s = (4660, 22136)
```

```
(gdb) print temp.s
$3 = {4660, 22136}
```

要求调试工具确定temp.s的类型，以确认这个short型变量被正确处理：

```
(dbx) whatis temp.s
```

```
short s [2];
```

```
(gdb) whatis temp.s
type = short int [2]
```

单步执行到下一个语句，即执行第51行的代码，显示第57行的代码：

```
(dbx) n
stopped in main at line 57 in file "endian.cpp"
57        ptr = (char * ) &temp.l;
```

```
(gdb) n
57        ptr = (char * ) &temp.l;
```

第51行向char型数组变量temp.c赋值，打印这个赋值：

```
(dbx) print temp.c
temp.c = "ABC"
```

```
(gdb) print temp.c
$4 = "ABC"
```

要求调试工具确定temp.c的类型，以确认这个char型数组被正确处理：

```
(dbx) whatis temp.c
char c [4];
```

```
(gdb) whatis temp.c
type = char [4]
```

打印对结构temp的赋值：

```
(dbx) print temp
temp = {
    l = 305419896
    s = (4660, 22136)
    c = "ABC"
}
```

```
(gdb) print temp
```

```
$5 = {l = 305419896, s = (4660, 22136), c = "ABC"}
```

打印结构 temp 的内存地址：

```
(dbx) print &temp
&temp = 0x20fe8
```

```
(gdb) print &temp
$6 = (endian *) 0x8049a10
```

检查这个地址的内存：

```
AIX - (dbx) 0x20000910/
0x20000910: 1234
```

```
Solaris - (dbx) examine 0x20fe8
0x00020fe8: temp: 0x1234
```

```
(gdb) x 0x8049a10
0x8049a10 <temp>: 0x12345678
```

再使用一种格式检查这个地址：dbx 使用双字节显示十六进制数，对于结构中的三个 4 字节变量，需要显示 6 组数值；而 gdb 使用 4 字节显示，因此只需要显示 3 个字：

```
AIX - (dbx) 0x20000910/6x
0x20000910: 1234 5678 1234 5678 4142 4300
```

```
Solaris - (dbx) examine 0x20fe8/6x
0x00020fe8: temp: 0x1234 0x5678 0x1234 0x5678 0x4142 0x4300
```

```
(gdb) x/3x 0x8049a10
0x8049a10 <temp>: 0x12345678 0x56781234 0x00434241
```

单步执行到下一个语句，即执行第 57 行的代码，显示第 58 行的代码：

```
(dbx) n
stopped in main at line 58 in file "endian.cpp"
58   printValue(ptr);
```

```
(gdb) n
58        printValue(ptr);
```

调试进入方法‘printValue’；注意，gdb显示传递到方法中的参数：

```
(dbx) s
stopped in printValue at line 34 in file "endian.cpp"
34        for (i = 0; i < 4; i++)
```

```
(gdb) s
printValue (ptr=0x8049a10 "xV4 \ 0224 \ 022xVABC") at endian.cpp: 34
34        for (i = 0; i < 4; i++)
```

显示应用程序的调用栈：

```
(dbx) where
=> [1] printValue (ptr = 0x20fe8 "^R4Vx^R4VxABC"), line 34 in "endian.cpp"
[2] main (argc = 1, argv = 0xeffffb44), line 58 in "endian.cpp"
```

```
(gdb) where
#0 printValue (ptr = 0x8049a10 "xV4 \ 0224 \ 022xVABC") at endian.cpp: 34
#1 0x804880c in main (argc = 1, argv = 0xbffffaa4) at endian.cpp: 58
```

向上移动应用程序调用栈，将显示激活当前方法的方法：

```
(dbx) up
Current function is main
58  printValue (ptr);
```

```
(gdb) up
#1 0x804880c in main (argc = 1, argv = 0xbffffaa4) at endian.cpp: 58
58        printValue (ptr);
```

再显示应用程序调用栈，注意，dbx显示正被显示的栈：

```
(dbx) where
  [1] printValue (ptr = 0x20fe8 "^R4Vx^R4VxABC"), line 34 in "endian.cpp"
=> [2] main (argc = 1, argv = 0xeffffb44), line 58 in "endian.cpp"
```

```
(gdb) where
#0 printValue (ptr = 0x8049a10 "xV4 \ 0224 \ 022xVABC") at endian.cpp: 34
#1 0x804880c in main (argc = 1, argv = 0xbffff9d4) at endian.cpp: 58
```

回到栈中的当前方法：

```
(dbx) down
Current function is printValue
  34      for (i = 0; i < 4; i++)
```

```
(gdb) down
#0 printValue (ptr = 0x8049a10 "xV4\0224\022xVABC") at endian.cpp: 34
34        for (i = 0; i < 4; i++)
```

再显示应用程序调用栈：

```
(dbx) where
=> [1] printValue (ptr = 0x20fe8 "^R4Vx^R4VxABC"), line 34 in "endian.cpp"
   [2] main (argc = 1, argv = 0xeffffb44), line 58 in "endian.cpp"
```

```
(gdb) where
#0 printValue (ptr = 0x8049a10 "xV4\0224\022xVABC") at endian.cpp: 34
#1 0x804880c in main (argc = 1, argv = 0xbffff9d4) at endian.cpp: 58
```

单步执行到下一个语句，即执行第34行的代码，显示第36行的代码：

```
(dbx) n
stopped in printValue at line 36 in file "endian.cpp"
  36      printf ( "%x", *ptr++);
```

```
(gdb) n
36        printf ( "%x", *ptr++);
```

再执行两次循环，注意，dbx不在条件语句的关闭扩号处停止，而gdb却停在此处：

```
(dbx) n
stopped in printValue at line 36 in file "endian.cpp"
  36      printf ( "%x", *ptr++);
(dbx) n
stopped in printValue at line 36 in file "endian.cpp"
  36      printf ( "%x", *ptr++);
```

```
(gdb) n
```

```
37      }
(gdb) n
36      printf ( "%x", * ptr++ );
(gdb) n
37      }
(gdb) n
36      printf ( "%x", * ptr++ );
(gdb) n
37      }
```

打印循环条件变量 i 的当前值：

```
(dbx) pi
i = 2
```

```
(gdb) pI
$ 7=2
```

保持单步执行应用程序，直到第38行：

```
(dbx) n
stopped in printValue at line 36 in file "endian.cpp"
  36      printf ( "%x", * ptr++ );
(dbx) n
stopped in printValue at line 38 in file "endian.cpp"
  38      printf ( "\n");
```

```
(gdb) n
36      printf ( "%x", * ptr++ );
(gdb) n
37      }
(gdb) n
38      printf ( "\n");
```

单步执行到 printValue 方法结束：

```
(dbx) n
12 34 56 78
```

```
stopped in printValue at line 39 in file "endian.cpp"
39        }
```

```
(gdb) n
39        }
```

跳出该方法，回到上一个方法：

```
(dbx) n
stopped in main at line 64 in file "endian.cpp"
  64      ptr = (char *) &temp.s;
```

```
(gdb) n
main (argc=1 , argv=0xbffffaa4) at endian.cpp: 64
64        ptr = (char *) &temp.s;
```

再显示应用程序的调用栈：

```
(dbx) where
=> [1] main (argc = 1, argv = 0xeffffb44), line 64 in "endian.cpp"
(gdb) where
#0 main (argc = 1, argv = 0xbffffaa4) at endian.cpp: 64
```

单步执行到下一个语句，即执行第64行的代码，显示第65行的代码：

```
(dbx) n
stopped in main at line 65 in file "endian.cpp"
  65      printValue (ptr);
```

```
(gdb) n
65        printValue (ptr);
```

调试进入方法‘printValue’；注意，gdb显示传递到方法中的参数：

```
(dbx) s
stopped in printValue at line 34 in file "endian.cpp"
  34      for (i = 0; i < 4; i++)
```

```
(gdb) s
printValue (ptr=0x8049a14 "4 \ 022xVABC") at endian.cpp: 34
```

```
34        for (i = 0; i < 4; i++)
```

从方法返回（对 dbx 是结束方法的执行，对 gdb 则是立即返回）；注意 dbx 是从方法返回并移动到代码的下一行，而 gdb 则返回到调用方法的行。

```
AIX - (dbx) return
12 34 56 78
stopped in main at line 71 in file "endian.cpp" ($ t1)
  71      ptr = (char *) &temp.c;
```

```
Solaris - (dbx) step up
12 34 56 78
printValue returns
stopped in main at line 71 in file "endian.cpp"
  71      ptr = (char *) &temp.c;
```

```
(gdb) return
Make printValue (char *) return now? (y or n) #0 0x804881f in main (argc=1,
argv=0xbffffaa4) at endian.cpp: 65
65        printValue (ptr);
(gdb) n
71        ptr = (char *) &temp.c;
```

调试到下一个语句，即执行第 71 行的代码并显示第 72 行的代码：

```
(dbx) n
stopped in main at line 72 in file "endian.cpp"
  72      printValue (ptr);
```

```
(gdb) n
72        printValue (ptr);
```

调试进入方法 printValue：

```
(dbx) s
stopped in printValue at line 34 in file "endian.cpp"
  34      for (i = 0; i < 4; i++)
```

```
(gdb) s
printValue (ptr=0x8049a18 "ABC") at endian.cpp: 34
```

```
34        for (i = 0; i < 4; i++)
```

返回并结束方法（对 gdb 也是结束方法的执行）：

```
(dbx) step up
41 42 43 0
printValue returns
stopped in main at line 74 in file "endian.cpp"
74        return 0;
```

```
(gdb) finish
Run till exit from #0 printValue (ptr = 0x8049a18 "ABC") at endian.cpp: 34
41 42 43 0
0x8048832 in main (argc = 1, argv = 0xbffff9d4) at endian.cpp: 72
72        printValue (ptr);
(gdb) n
74        return 0;
```

结束应用程序：

```
(dbx) c
execution completed, exit code is 0
```

```
(gdb) c
Continuing.
78 56 34 12
Program exited normally.
```

退出调试工具：

```
(dbx) quit
```

```
(gdb) quit
```

7.5 小结

本章描述了三种不同的调试工具的最常用的命令。涉及的命令可分成五类：使应用程序停止、运行应用程序、检查应用程序、检查数据、提供给用户的逐行控制调试进程。另外，还论及了调试多线程应用程序。最后以一个由调试工具 dbx 和 gdb 调试的应用程序的例子结束了本章。

第8章　移　　植

本章将讨论以下内容:

- 理解所有的操作系统并不相同
- 学习 Endianism
- 学习如何开发面向国际市场的应用程序

通常，应用程序都是仅为一个硬件平台和一种操作系统而编写的。它们使用某一种编程语言设计，并支持某一种自然语言。然而，通过移植，你的应用程序可以在不同的硬件和（或）软件环境中工作。尽管移植并不是与面向对象分析和设计特别相关，但它是一个你在为其他人编写应用程序之初就应考虑的问题。

当然，有些平台比其他平台更易于移植。在软件生产中移植并不是很容易的工作，通常它都是由一个专门从事移植工作的职业团队来完成。

本章感兴趣的主题是移植到一个新的硬件平台，移植到一种新的操作系统，以及移植到一种新的自然语言（国际化和当地化）。有些人可能会争辩说国际化和当地化根本算不上是移植，但我仍然有理由认为它是移植，就像支持一个新的硬件一样，支持一种新的自然语言（口头或书面）是一种对源代码和用户界面改动最少的移植。

应该注意的是，上面提及的每一个主题：硬件、操作系统及自然语言，它们分别影响应用程序的不同部分。当解决一个具体的问题时，每一个主题都有对它的特别要求，每一个主题都将在本章中被讨论。

8.1　移植到新的操作系统

如果要将你的应用程序移植到另一个操作系统，请（仔细）阅读随软件而附的使用手册。如果你要用的编程结构正巧提到它仅针对该特定的平台，这里给你一个提示：不要使用它，使用一个可移植的并且能跨多个操作系统工作的编程结构。

当写一个 C++ 应用程序时，移植工程师可以利用条件编译指令帮助写出能跨不同操作系统的代码。这些指令可在任何的 C 或 C++ 编程书籍中查到，列表如下：

- # define 和 # undef
- # if、# elif、# else 和 # endif
- # ifdef、# ifndef、# else 和 # endif
- # if defined、# if ! defined、# else 和 # endif

操作系统之间的差别之一是基本数据类型。Microsoft Visual C++®定义了许多特殊的数据类型。它们包括 DWORD，LPCSTR 和 TCHAR。代码可以被同道的 Microsoft Visual C++ 用户很快地理解，但这些数据类型在任何其他的操作系统中都不存在。

幸运的是这些数据类型的绝大多数在实际的C++中都能找到它的等价数据类型，但事情也并不总是如此：

- DWORD = 32位无符号整数
- LPSTR = 32位指向字符串的指针
- TCHAR = The _ TCHAR：在TCHAR.H中条件定义的数据类型。如果在你的构造中定义了符号_ UNICODE，则_ TCHAR被定义为wchar _ t；否则，对于单字节和MBCS构造，_ TCHAR被定义为char（wchar _ t是基本的Unicode宽字符数据类型，是与8位有符号char对应的16位字符）。

另外一个出现问题的方面是系统方法（函数）的使用。下面的例子表现了访问时间的同一系统方法的不同版本。编译指令的使用表现了如何实现可用于多个操作系统中的一个函数。

```
/*
** The 'time.h' header file can be found in one of two places
** either '.../include/time.h'
** or '.../include/sys/time.h'
*/
#if defined(INCLUDE_SYS_TIME)
#include<sys/time.h>
#else
#include <time.h>
#endif

void getTimeOfDay (int time_secs, int time_msecs)
{
   struct timeval  t;
   struct timezone tz;

#if defined(TIMEONLY_GETTIMEOFDAY)
   gettimeofday(&t);
#elif defined(TIME_AND_TZ_GETTIMEOFDAY)
   gettimeofday(&t, &tz);
#endif

    time_secs = t.tv_sec;
    time_msecs = t.tv_usec;
}
```

下面的这些例子体现了两个支持线程的方法Psem和Vsem分别在Microsoft Visual C++、UNIX C++和Java环境中的实现。UNIX版本包含文件‘PM.h’，它使用一个头文件来表明如何使用宏（macro）来支持System V线程和POSIX线程，以实现对不同线程模型的支持。

8.1.1 Microsoft Visual C++中的线程支持

```
// Psem - block while trying to obtain a specific mutex
//     mutex - the ID of the mutex being requested
//     seq - used to identify where in the program this routine
//           was called from
```

```
void Psem (HANDLE mutex, int seq)
{
    DWORD stat;
    stat = WaitForSingleObject( mutex, INFINITE );
    if(stat == WAIT_FAILED)
    {
      printf ("Error lock - %d = %d\n", seq, stat);
      exit(1);
    }
}

// Vsem - release a specific mutex
//     mutex - the ID of the mutex being released
//     seq - used to identify where in the program this routine
//           was called from
void Vsem (HANDLE mutex, int seq)
{
    DWORD stat;
    stat = ReleaseMutex ( mutex );
    if(stat == 0)
    {
      printf ("Error unlock - %d = %d\n", seq, stat);
      exit(1);
    }
}
```

8.1.2 UNIX中的线程支持

以下摘自头文件‘PM.h’，这个头文件包含常用的线程数据类型和系统过程中的与平台无关的宏：

```
/*
** Porting Threads
** Macros that distinguish between the two most popular threading
** models on UNIX, System V threads and POSIX threads
**
** PM_ - PortableMutex
**
** Name: PT_MUTEX - Simplest synchronization object
**
** Description:
**    When operating system semaphores are used,
**    this is the simplest form of synchronization object.
*/
# ifdef SYS_V_THREADS
# define  PM_t                    mutex_t
# define  PM_init (mutexptr)      mutex_init(mutexptr,NULL,NULL)
# define  PM_destroy (sptr)       mutex_destroy(sptr)
# define  PM_lock (sptr)          mutex_lock(sptr)
# define  PM_trylock (sptr)       mutex_trylock(sptr)
# define  PM_unlock (sptr)        mutex_unlock(sptr)
```

```
# endif /* SYS_V_THREADS /
# ifdef POSIX_THREADS
# define  PM_t                    pthread_mutex_t
# define  PM_init (mutexptr)      pthread_mutex_init(mutexptr,NULL)
# define  PM_destroy (sptr)       pthread_mutex_destroy(sptr)
# define  PM_lock (sptr)          pthread_mutex_lock(sptr)
# define  PM_trylock (sptr)       pthread_mutex_trylock(sptr)
# define  PM_unlock (sptr)        pthread_mutex_unlock(sptr)
# endif /* POSIX_THREADS */
```

这里是两个线程方法的代码示意：

```
// Psem - block while trying to obtain a specific mutex
//     mutex - the ID of the mutex being requested
//     seq - used to identify where in the program this routine
//           was called from
void Psem (PM_t *mutex, int seq)
{
    int stat;
    // request a lock on a specific mutex
    stat = PM_lock (mutex);
    if (stat != 0)
    {
        // if the request fails, report error to the user
        // do not try and recover
        cout << "Error PM_lock - " << seq << " = " << stat << endl;
       exit(1);
    }
}

// Vsem - release a specific mutex
//     mutex - the ID of the mutex being released
//     seq - used to identify where in the program this routine
//           was called from
void Vsem (PM_t *mutex, int seq)
{
    int stat;
    // release the lock on a specific mutex
    stat = PM_unlock (mutex);
    if (stat != 0)
    {
    // if the release fails, report error to the user
    // do not try and recover
    cout << "Error PM_unlock - " << seq << " = " << stat << endl;
        exit(1);
    }
}
```

8.1.3 Java中的线程支持

```
// Psem - block while trying to obtain a specific mutex
//     pos - the number of the mutex being requested
public void Psem (int pos)
{
    boolean    flag;

System.out.println (threadName + " RequestLock");
synchronized (Dinner.chopstickLock [pos])
{
flag = Dinner.chopstickLock [pos].booleanValue ();
if (flag == false)
        {
            Dinner.chopstickLock [pos] = new Boolean (true);
            System.out.println (threadName + " AcquiredLock ");
            return;
        }
    }

    while (flag == true)
    {
        synchronized (Dinner.chopstickLock [pos])
        {
            flag = Dinner.chopstickLock [pos].booleanValue ();
            if (flag == false)
            {
                Dinner.chopstickLock [pos] = new Boolean (true);
                System.out.println (threadName + " AcquiredLock");
                return;
            }
        }
        try {sleep (20);}
        catch (InterruptedException e) {}
    }
}

// Vsem - release a specific mutex
//    mutex - the number of the mutex being released
public void Vsem (int pos)
{
   System.out.println (threadName + " Vsem " + pos);
   synchronized (Dinner.chopstickLock [pos])
   {
      Dinner.chopstickLock [pos] = new Boolean (false);
   }
}
```

8.2 移植到新的硬件平台

在过去，实际上只有两种硬件平台，使用 Intel x86 处理器的机器和不使用这种处理器的机器。所以，在相当长一段时间内，一个程序员只需关注这两种平台之间的相互移植，这种关注称为“endianism”，它与数据在内存中的存放方式有关。

但是，最近几年，两个阵营都出现了新的 64 位模式。以前，一切都是 32 位的，也就是说整型（integer）和长整型（long）都占据 32 位的内存，不幸的是，这也意味着程序员经常将这两种数据类型互用。

8.2.1 支持 Endianism

下面是一些关于 endianism 的历史渊源。

使用名称“Big Endian”和“Little Endian”，是与经典儿童读物《格列佛游记》中的流血冲突的恰当类比。冲突的双方是虚构的两个岛屿，Lilliput 和 Blefescu，冲突的起因是打碎鸡蛋时应敲击哪一端（大的一端还是小的一端）的问题。在《格列佛游记》中，Lilliput 人喜欢敲击鸡蛋的小端，而 Blefescu 人喜欢敲击鸡蛋的大端，他们为此争战不休。在我们的情况下，问题与一个多字节数据类型的结束位（最高位或最低位，most significant or least significant）有关。

属于 Big - Endian 阵营（最高位存储在前）的有 Java VM 计算机、Java 二进制文件格式、IBM 360 以及相关的大型机、Motorola 68K 和最大型机。

Blefescu 人（Big - Endians 阵营）主张整型的存储方式应该是最高位首先存储。当使用调试器检查内存的内容时，Big - Endian 更容易理解。

属于 Little - Endian 阵营（最低位存储在前）的是 Intel 8080、8086、80286、奔腾（Pentium）系列，以及在 Apple II 中流行的 AMD 6502。

Lilliput 人（Little - Endians 阵营）认为，将低位存储在前面更自然，因为当你手动算术运算时，你是从最低位开始，逐渐向最高位行进的。当使用调试器检查内存的内容时，Little - Endian 就带来疑问了。

对于 Big - Endian 顺序来说，一个多字节数据类型的地址是它的最高位字节（它的“大端”）的地址，而对于 Little - Endian 顺序来说，一个多字节数据类型的地址是它的最低位字节（它的“小端”）的地址。对于在高级编程语言中声明的结构，内存中字节的顺序依赖于字节顺序和具体的数据类型，如在下面的 C 结构中。

要写出无论在 Big - Endian 还是 Little - Endian 目标环境中都能够正确编译并运行的代码，程序员必须遵循一个基本原则：剔除与字节顺序有关的代码或者将这些代码括入 # ifdef/else 语句中。每当用于保存数据的数据类型与用于获取数据的数据类型不一致时，就会使用与字节顺序有关的代码。下面的例子有一点令人疑惑，但它显示了数据是如何使用指定的数据类型保存，然后使用字符类型来访问它。这个程序说明了在一个 Little - Endian 机器中，字在内存中是如何存放的：即 4 个字节倒置、两个半字双双倒置，以及整字正放。

```
/*
** endian.cpp
** This application is used to test endism.
**
** Expected results:
```

```
**      Big Endian          Little Endian
**      12 34 56 78         78 56 34 12
**      12 34 56 78         34 12 78 56
**      41 42 43 00         41 42 43 00
*/
#include <stdio.h>
#include <string.h>

/*
** This structure allocates 3 variables each of 4 bytes
** Each variable is a different type, this allows the example
** to show how data is stored in memory
*/
struct endian {
   long l;                       /* 1 * 4-byte variable */
   short s[2];                   /* 2 * 2-byte variables */
   char c[4];                    /* 4 * 1-byte variables */
};

endian  temp;

/*
** This method is used to print the four bytes of each variable
** passed to it
*/
void printValue (char* ptr)
{
   int    i;

   for (i = 0; i < 4; i++)
   {
      printf ("%x ", *ptr++);
   }
   printf ("\n");
}
int main(int argc, char* argv[])
{
    char    *ptr;

/*
** These are the values that will be stored in memory
*/
    temp.l = 305419896;          /* 0x12345678 -> 12-34-56-78 */
    temp.s [0] = 4660;           /* 0x1234 -> 12-34 */
    temp.s [1] = 22136;          /* 0x5678 -> 56-78 */
    strcpy (temp.c, "ABC");      /* 'A'-'B'-'C'-<end of string> */

/*
```

```
** Set the pointer to the long variable
** and then print the 4 bytes of memory
*/
    ptr = (char *)&temp.l;
    printValue (ptr);

/*
** Set the pointer to the short array variable
** and then print the 4 bytes of memory
*/
    ptr = (char *)&temp.s;
    printValue (ptr);

/*
** Set the pointer to the char array variable
** and then print the 4 bytes of memory
*/
    ptr = (char *)&temp.c;
    printValue (ptr);

    return 0;
}
```

当使用 Microsoft Visual C++ 调试器检查内存时，看起来一切 OK。

```
[-]temp
 |-- a               0x12345678
 |-- [-]b
 |    |-- [0x0]      0x1234
 |    |-- [0x1]      0x5678
 |-- [-]c
      |-- [0x0]      0x41       'A'
      |-- [0x1]      0x42       'B'
      |-- [0x2]      0x43       'C'
      |-- [0x3]      0x00       ''
```

然而，当这个程序运行时，得到的结果是

```
78 56 34 12
34 12 78 56
41 42 43 00
```

当使用 UNIX C++ 构造这个程序并使用 UNIX 系统下的 dbx 工具检查内存时：

```
$ dbx endian
dbx > stop at 30
dbx > print &temp
0xeffffad8
dbx > examine 0xeffffad8 /6x
0xeffffad8: 0x1234 0x5678 0x1234 0x5678 0x4142 0x4300
```

程序运行时，得到的结果是

```
12 34 56 78
12 34 56 78
41 42 43 00
```

当使用 Linux 系统中的 g++ 构造这个程序并使用 Linux 系统下的 gdb 工具检查内存时：

```
$ gdb endian
gdb > break 30
gdb > print &temp
$1=(endian*)0xbffffb8c
gdb > x/3x 0xbffffb8c
0xbffffb8c: 0x12345678 0x56781234 0x00434241
```

程序在 Linux 系统中运行时，得到的结果是

```
78 56 34 12
34 12 78 56
41 42 43 00
```

8.2.2 32 位和 64 位机器的比较

目前，人们使用的大多数机器是 32 位的机器。这是因为它们使用 32 位来访问内存。32 位允许程序访问 4G 的地址空间。整数和长整数数据类型的字节大小都被设为 32 位。由于寄存器的大小不同，所以数据的传递是 32 或 64 位，进而影响不同机器的数据结构的对齐方式。另外，在 64 位机器上，地址指针与整型的字节大小不同，将它保存在内存中时，二者不可混用。如果目前有 64 位的长整型数据类型、而整型仍保留为 32 位的机器，就意味着程序员不得不特别谨慎，以免混淆数据类型。

新的 64 位机器将长整型数据类型设置为 64 位，而整型仍保留为 32 位。64 位的机器允许程序员访问的地址空间为 1.8×10^{19}。

8.3 移植到新的语言

本节讨论指定应用程序运行环境的重要性。涉及的主题有指定地点，单字节、多字节和 Unicode 存储系统的区别，以及消息目录。

国际化被表示为‘i18n’，它代表‘i’，跨越随后的 18 个字符，直到结尾处的‘n’。类似地，本地化可被表示为‘l10n’，它代表‘l’，跨越随后的 10 个字符，直到结尾处的‘n’。

8.3.1 国际化和本地化

软件的国际化，指不必处理程序的可执行代码，就可开发一个面向特定用户群体的软件的过程。

软件的本地化指改编一个软件产品，使之适用于一个特殊的地区，或供特定书面语言的用户使用的过程。本地化包括消息的翻译、图标或图像的更改等。好的 i18n 设计可以大大简化

本地化过程。

软件应该被设计为具有全球开发能力，尽管它是完全本地化的。好的 i18n 项目将需要本地化的资料保存为独立于代码的一种格式。将你的应用程序国际化应该包括下列内容：

- 实现应用程序，以保证它能够在多数地区工作。
- 使应用程序可配置，即只需简单地替换消息/资源文件，就可实现本地化。
- 同时支持不同地区。
- 允许非技术人员（如翻译小组）不需打开源代码就可访问有关信息。
- 支持通用特性，使得开发一个可移植到不同平台的应用程序成为可能。

虽然 i18n 不要求支持 Unicode，但它确实能大大简化 i18n 项目的开发和维护。一些以前非常困难的任务现在正是由于它，变得非常简单了。

8.3.2 应用程序国际化时需要考虑的问题

要将你的应用程序国际化，有一些需要考虑的基本问题。它们有助于开发不依赖于潜在的最终用户的语言、国家和文化的软件。这样会得到一个可被立即翻译到多种国家或地区的应用程序。将程序国际化的三个关键步骤是：

(1) 分离功能与形式，即分离应用程序的工作方式与它的表现形式。

(2) 避免做文化方面的假设。

(3) 提供对多个 locale 的支持。

说明：虽然看起来简单，第一步的关注焦点是将静态信息（如图像、窗口布局）从程序代码中分离出来，第二步的关注焦点是保证程序在运行过程中产生的文本（如错误消息）能够以正确的语言、适当的格式呈现在目标用户面前。

另外，还有处理输入和输出的问题，如何处理来自各种输入源（从键盘到声音）的输入，然后如何将它们正确回显给用户。这些事情对英语来说相对简单，但对其他一些语言来说却非常具有挑战性。

1. 分离功能与形式

要设计好的应用程序软件，需要将实现用户界面的程序代码与实现基础功能的程序代码分离开来。这样做的好处之一是应用程序能够被移植到其他支持不同的用户界面风格的平台，只有用户界面部分的代码需要修改。

可以认为，用户界面包含用户能够看得见的条目。这些条目包括（但不限于）如下各项：

- 向用户显示的消息
- 包含文字或图标的按钮
- 用户能看到的、引导应用程序运行的菜单命令
- 对话框布局、图标和颜色

描述用户界面的程序代码应该保存在一起，但与描述应用程序功能的程序代码保持分离。

2. 避免文化方面的假设

每当程序员编写一段代码时，他们通常会做一系列假设。其中一个假设可能是软件产品的使用者都懂英语，而另外一个假设可能是用户使用的输入设备（键盘）都是一样的。单是这样

的两个假设，就自动限制了产品超越当前环境的可能性。虽然英语在许多国家广为使用，但它通常是第二语言，不是母语，不能被完全理解，尤其是当程序员使用一些方言时。

例如，当在美国和英国驾驶汽车时，‘STOP’意味着停止；但是，在一个交叉路口避让主路上的车辆时，美国人和英国人一个使用‘YIELD’，另一个则使用‘GIVE WAY’。程序员所做的任何文化方面的假设都会使移植用户界面的代码更为困难，甚至不得不重写基础代码。

3. 提供对多个 locale 的支持

应用程序使用 locale 来确定应用程序的某些特性将如何体现。调用 setlocale（）可设置进程或获取进程的设置。由环境变量控制的特性列出如下。

LANG 这个变量确定当不存在 LC_ALL 和其他 LC_（LC_COLLATE，LC_CTYPE，LC_MESSAGES，LC_MONETARY，LC_NUMERIC，LC_TIME）环境变量时，语言、当地习惯和字符编码集的 locale 分类。应用程序可用这个变量确定错误信息和操作指示的编程语言、对照表的顺序、日期的格式及其他。

LC_ALL 这个变量确定所有 locale 分类的数值。在所有以 LC_开头的环境变量（LC_COLLATE，LC_CTYPE，LC_MESSAGES，LC_MONETARY，LC_NUMERIC，LC_TIME）和环境变量 LANG 中，LC_ALL 环境变量值的优先级是最高的。

LC_COLLATE——对照表 这个变量确定字符对照表的 locale 分类。它决定常规表达方式和排序的对照表信息，包括多种工具中的等价类和多字符对照元素以及 strcoll（）和 strxfrm（）函数。所有的语言（甚至那些使用相同字母的语言）不必具有相同的字母顺序的概念。更重要的，当假定字母顺序与字符集的字符码值的数字顺序一致时，要特别小心。实际上，‘a’与‘A’截然不同，‘b’与‘B’也截然不同。在字符集中，它们有不同的字符码值。这意味着你不能用像 strcmp（）这样的位方式的词汇比较手段对用户可见的列表进行排序。strcmp（）变量不能用于位方式的比较。这是因为比较的结果是“A < B < a < b”。

并非所有的语言都认为同样的字符是等价的，或者改变一个字符的大小写总是一对一的映射。在确定两个字符串是否相同时，重音的差别、某些字符的存在或缺席、甚至拼写的差别可能无关紧要。

当检查字符是否属于特定的类别时，不要特别地列出你所关心的字符，不能假定它们在编码体系中有特定顺序。如，/A-Za-z/不能代表多数欧洲语言中所有的字母。在很多系统中，/0-9/也不能表示所有的数字，这包括使用 C 接口函数（如 isupper（）和 islower（））。

LC_CTYPE——字符处理 这个变量确定字符处理函数（如 *tolower*（），*toupper*（），*isdigit*（），*isalpha*（），*mbtowc*（），*wctomb*（））的 locale 分类。这个环境变量确定作为字符的文本数据的字节顺序的解释（如，单字节或多字节字符）、字符的分类（如，字母，数字和图形）和字符类的行为。这个变量的其他语义（如果有的话）是与实现相关的。

LC_MESSAGE 这个变量确定 gencat、catopen、catgets 和 catclose 使用的消息分类的位置。在做出诸如怎样将单个的文本片段粘合在一起形成完整的句子（如生成错误信息）的假设时要小心。当信息被翻译到一种新的语言时，词组的顺序可能会改变。这个变量的其他语义（如果有的话）是与实现相关的。设置 LC_MESSAGES 应该影响那些格式未被指定的诊断和通知信息的语言和文化习惯。

说明：可能存在这种情况，即句子的一部分依赖于句子的其他部分而变化（最常见的例子是对于一个数字之后的名词，选择单数形式还是复数形式）。

LC_MONETARY 这个变量确定货币单位如何显示。数字表示法也可能根据度量单位和货币数值的不同而变化。国与国之间的货币是可能不同的。一个很好的例子是$ 1000 dollar 的表示法。这可能是美国的美元或加拿大的元。根据 locale 的不同，美元可显示为 USD，而加拿大元可显示为 CAD。有时显示的数字量可能变化，数字本身也可能变化。

```
%%%%%%%%%%%%
LC_MONETARY
%%%%%%%%%%%%
int_curr_symbol    ""
currency_symbol    ""
mon_decimal_point  ""
mon_thousands_sep  ""
mon_grouping       ""
positive_sign      ""
negative_sign      ""
```

这个变量的其他语义（如果有的话）是与实现相关的。

LC_NUMERIC 这个变量确定 locale 分类，该分类决定不同工具中的数字格式化信息（如，千位分隔符和基数字符）、printf（）和 scanf（）中的格式化 I/O 操作，以及 strtod（）中的字符串转换函数。数字和日期在不同的语言中的表示方法并不相同。尤其是，不要实现将数字转换为字符串的函数，不要调用像 sprintf（）这样的低级系统接口，它们产生的结果不会因语言的变化而变化。

```
%%%%%%%%%%%%
    LC_NUMERIC
%%%%%%%%%%%%
    decimal_point     "<period>"
    thousands_sep     ""
    grouping          ""
```

这个变量的其他语义（如果有的话）是与实现相关的。

LC_TIME——日期和时间 这个变量确定日期和时间格式信息的 locale 分类。它影响 strftime（）中的时间函数的行为。这个变量的其他语义（如果有的话）是与实现相关的。计算时间有多种计时系统，如年和月的长度、哪一天是一周的第一天、月和年的可接受的数值范围（用 DateFormat）、你所在的时区（用 TimeZone）或何时开始夏令时等。

以下是如何使用 locale 变量的例子。请求系统打印出当前各种 locale 的设置。一切都是操作系统的默认设置：

```
$ locale
LANG=en_US
LC_CTYPE="C"
LC_NUMERIC="C"
LC_TIME="C"
LC_COLLATE="C"
```

```
LC_MONETARY="C"
LC_MESSAGES="C"
LC_ALL=
```

将语言 locale 设置为 British English，再显示各种 locale 的设置：

```
$ setenv LANG en_UK
$ locale
LANG=en_UK
LC_CTYPE="en_UK"
LC_NUMERIC="en_UK"
LC_TIME="en_UK"
LC_COLLATE="en_UK"
LC_MONETARY="en_UK"
LC_MESSAGES="en_UK"
LC_ALL=
```

改变对照表 locale 的设置：

```
$ setenv LC_CTYPE "en_US"
$ locale
LANG=en_UK
LC_CTYPE="en_US"
LC_NUMERIC="en_UK"
LC_TIME="en_UK"
LC_COLLATE="en_UK"
LC_MONETARY="en_UK"
LC_MESSAGES="en_UK"
LC_ALL=
```

恢复到系统默认设置：

```
$ setenv LANG en_US
LANG=en_US
LC_CTYPE="C"
LC_NUMERIC="C"
LC_TIME="C"
LC_COLLATE="C"
LC_MONETARY="C"
LC_MESSAGES="C"
LC_ALL=
```

8.3.3 单字节和双字节字符集

单字节字符集中，用一个字节或 8 位来存储字符。这种字符集可最多存储 256 个字符。ASCII 字符集就属于单字节字符集。双字节字符集中，用两个字节或 16 位来存储字符。这种字符集可存储多达 2^{16} 个字符。

双字节字符集的特殊问题

创建双字节字符集（double-byte character set，缩写为 DBCS）的目的是为了处理使用表意

字符的东亚语言，这些语言需要的字符总数多于 ANSI 支持的 256 个。DBCS 中，访问字符使用 16 位表示法，即使用两个字节。16 位表示法可表示 65 536 个字符，东亚语言中定义的字符数远少于此。例如，日语字符集中目前定义了 10 000 多个字符。

在使用 DBCS 的地区——包括中国、日本和韩国——单字节和双字节字符都被包括在字符集中。这些地区使用的单字节字符与当地的国家标准一致，并与 ASCII 字符集基本一致。这些单字节字符集（SBCS）中的某些码位被设计为 DBCS 字符的首字节（lead byte）。由一个首字节和一个尾字节组成的连续字节对表示一个双字节字符。用作首字节的编码范围由各地区决定。

说明：DBCS 是一个与 Unicode 不同的字符集。

当开发支持 DBCS 的应用程序时，你应该考虑如下几方面：

- Unicode、ANSI 和 DBCS 之间的区别
- DBCS 排序和字符串比较
- DBCS 字符串处理函数
- DBCS 字符串转换
- 在 DBCS 环境中，如何正确地显示和打印字符
- 如何处理包含双字节字符的文件
- DBCS 标识符
- 支持 DBCS 的事件
- 如何调用 Windows API

提示：开发支持 DBCS 的应用程序是一个很好的实践的机会，无论该应用程序是否会在使用 DBCS 的地区运行。这种实践有助于你开发一个灵活的、易于移植的、真正国际化的应用程序。

8.3.4 宽字符串

宽字符（wchar_t）通常是一个两字节的多语言字符编码。不过，有些操作系统，如 Solaris，将 wchar_t 的长度定义为 4 字节。按照 Unicode 标准，现代计算机范畴内使用的任何字符，包括科技符号和特殊印刷字符，都可表示为一个宽字符。因为每个宽字符都被表示为固定的 16 位大小，所以，使用宽字符可以简化国际化字符集的编程。

一个宽字符字符串被表示为一个 wchar_t [ ] 数组，并由一个 wchar_t * 指针指示。在 ASCII 字符前使用前缀字符 L 就可将任何 ASCII 字符表示为一个宽字符。如，L'\0'就是宽字符的字符串结束符 NULL。类似地，在 ASCII 字符串前使用前缀字符 L 就可将任何 ASCII 字符串表示为一个宽字符字符串，如，L"Hello"。

通常，宽字符比多字节字符占用的内存空间更多，但处理速度快。另外，多字节编码同时只能支持一个地区的编码，而 Unicode 编码可同时表示世界上的所有字符。

8.3.5 Unicode

Unicode 是 16 位编码字符集，它用一个统一的形式（即，不是各国家标准的重复形式）包括了世界上所有常用的字母字符集和表意字符集。在应用程序中，Unicode 字符是使用‘wchar

_t’数据类型来处理的。以下是关于Unicode需要考虑的几个要点：

- ▶ 在目前的版本中，Unicode标准包含源自24种支持脚本的30 000多个独立的编码字符。这些字符覆盖了美洲、欧洲、中东、非洲、印度、亚洲和太平洋地区的主要书面语言。
- ▶ Unicode是Unicode Consortium的商标，进一步的信息可访问http：//unicode.org/。有几种编程语言内部使用Unicode，Visual Basic和Java就是两种这样的编程语言。
- ▶ Unicode允许一个程序在编程环境中，将所有的文本数据标准化为单个编码方案。编码转换只出现在输入和输出端。当文本在编程环境中时，对文本的操作被简化，因为你不需考虑（或跟踪）一个特定文本的编码。
- ▶ Unicode支持多语言数据，因为它覆盖世界上的所有语言的编码。你无需用特定的编码去标识数据片段，以支持适当的字符，你还能够在单个文本片段中混合多种语言。ASCII和Latin-1字符可被映射到Unicode字符。

8.4 将消息中的字符串本地化

以下讨论使用消息目录来将字符串本地化的问题，随后是关于Microsoft™编程语言中的资源文件的问题。Java本地化和国际化工具不在此讨论，但我相信，通过阅读这个工具自带的文档，你就可以自己学会。

8.4.1 创建消息目录——UNIX

消息目录是一种在多种语言中提供一致的消息的方法。然后这些消息可被任何程序访问。程序根据用户的locale设置，做出使用哪一个消息目录的决定。

消息目录文件始于程序员创建的源文件，包含应用程序需要的消息文本。这些消息文本可被用户界面使用或提供其他信息。

消息源文件被创建之后，它们被转换为消息目录。然后，这些目录被应用程序用于获取并显示消息文本。翻译消息源文件中的文本不需要改变或重编译应用程序。

以下是关于创建、转换和使用消息文件的步骤：

- ▶ 创建一个消息源文件
- ▶ 创建一个消息目录——UNIX工具使用gencat
- ▶ 用应用程序显示消息文本——C语言方法使用的是catopen、catgets和catclose

1. 创建一个消息源文件

要创建消息文本源文件，可以用任何文本编辑器打开一个文件，输入消息标识值（ID）。在一些消息目录工具创建的消息文本源文件中，标识值可能不是数值而是符号。这些消息目录工具不属于本书的范围，在此不予讨论。最后，输入消息文本。下面是一个消息源文件的例子：

```
1 message-text          $ (This message is numbered)
2 message-text          $ (This message is numbered)
OUTMSG message-text     $ (This message has a symbolic identifier \
                          called OUTMSG)
4 message-text          $ (This message is numbered)
```

创建消息源文件时，需要遵守一些基本规则：

- ▶ 消息 ID 值或标识与消息文本之间必须有一个空白字符。
- ▶ 在单个消息文本集中，消息 ID 值必须是递增的，但不必连续。数值 0 不是一个有效的消息 ID 值。

在消息源文件中加注释 你可以在消息源文件中除消息文本外的任何地方加注释。注释的格式是在$（dollar 符）后留至少一个空格或 Tab 符，然后是注释文本。下面就是一个注释的例子：

```
$ This is a comment.
```

注释不会出现在根据消息源文件产生的消息目录中。每当感觉有必要加注释时，就应该马上加；注释从不会显得多余。加上注释，能够保证程序员知道消息文本可以被恰当地用在应用程序的哪些地方。可以另外加上一些注释，以便翻译者知道消息文本被使用时的上下文环境。一个简单的短语用在两种不同的环境中，就可能有两种不同的翻译方法。最后这一点是很重要的：不要重载消息。如果一个消息不大可能被翻译为不同消息，那么，原始的消息应该出现多次。应该使用注释标识变量（如%s，%c，%d）代表的意义。为清晰起见，注释应该紧邻它指示的消息。

消息文本续写到下一行 跟在消息 ID 值后的空白符的所有文本，直到该行结束，都属于消息文本的内容。转义字符‘\’用于将消息文本续写到下一行。‘\’必须是一行的最后一个字符，正如下面的例子：

```
5 This is the text associated with \
message number 5.
```

上例的结果就是下面的单行消息文本：

```
This is the text associated with message number 5.
```

在消息文本中加入特殊字符 使用转义字符‘\’，可将其他的特殊字符插入到消息文本。这些特殊字符列出如下：

\n	新行开始字符
\t	水平 Tab 字符
\v	竖直 Tab 字符
\b	退格（backspace）字符
\r	行结束字符
\f	form-feed 字符
\\	字符“\”
\ddd	与由有效的八进制数字 ddd 表示的八进制数值相联系的单字节字符。最多可由三位八进制数指定 如果在字符串中使用八进制字节字符，必须将该字符的三位八进制数写完整，以避免与后面的字符混淆。如，字符$的八进制数值为 44，要显示$ 2.00，使用 \0442.00，而不使用 \442.00，否则，2 将被解析为八进制数值的一部分

（续）

\ xdd	与由两位有效的十六进制数字 dd 表示的十六进制数值相联系的单字节字符。最多可由两位十六进制数指定。你必须包括前面的 0，以避免解析错误（见 \ ddd 中的说明）
\ xdddd	与由四位有效的十六进制数字 dddd 表示的十六进制数值相联系的双字节字符。最多可由四位十六进制数指定。你必须包括前面的 0，以避免解析错误（见 \ ddd 中的说明）

具有信息量的消息文本 重要的是，消息文本要向用户提供有用的信息。只告诉用户出现了错误是不够的。用户应该被告知错误是怎样出现的、他们怎样才可能改正这个错误。以下是两个不同的消息文本：

```
Unhelpful Message:     Value out of range
Informative Message:   The range for the value is between 1 and 99.
```

消息“Value out of range”对用户没有太大的用处；而消息“The range for the value is between 1 and 99.”将涉及的数据的取值范围准确地告诉了用户。

消息源文件代码的例子显示如下：

```
$ This is a message source file sample.
$ This is a set of messages.
1 The specified file does not have read permission on\n
2 The %1$s file and the %2$s file are same\n
3 Hello world!\n
$ This is another set of messages
11 fieldef: Cannot open %1$s \n
12 Hello world\n
```

2. 创建消息目录

程序员创建包含应用程序消息的消息源文件，并将它转为一个消息目录。将消息源文件翻译为其他语言，然后转为消息目录，这个过程不需要改变或重编译应用程序。要创建一个消息目录，用系统命令 **gencat** 处理你的完整的消息源文件。命令 **gencat** 用于处理一个包含消息 ID 数值和相关的文本的消息源文件。下面的例子使用消息源文件‘messages.msg’中的信息，产生消息目录文件‘messages.cat’：

```
gencat messages.cat messages.msg
```

如果目录文件‘messages.cat’已经存在，命令 gencat 根据消息源文件中的语句修改目录文件。如果目录文件不存在，命令 **gencat** 创建该文件。

可以指定任意多个消息文本源文件。这些文件按照你指定的顺序被处理。每个后续的源文件依次修改目录文件。如果没有指定源文件，命令 **gencat** 从标准输入接受消息源文件数据。

3. 在应用程序中显示消息

要在应用程序中访问消息目录，必须包含下面几项：

▶ 头文件 limits.h 和 nl _ types.h

▶ 初始化 locale 环境
▶ 打开目录文件
▶ 读出一条消息文本
▶ 显示一条消息文本
▶ 关闭目录文件

以下的方法提供了在一个应用程序中显示消息目录中的消息的必要手段：

setlocale	设置 locale。通过调用子过程 **setlocale**，为选择的消息目录语言设置环境变量 LC_ALL 或 LC_MESSAGES
catopen	打开一个指定的消息目录并返回一个目录描述符，用于从目录中获取信息
catgets	调用 **catopen** 子过程成功后，从目录中获取一条信息文本
catclose	关闭指定的消息目录

下面的 C 程序'hello'，显示了如何使用 **catopen** 方法打开消息目录'hello.cat'，通过 **catgets** 方法从目录中获得信息，利用 **printf** 方法显示信息，通过 **catclose** 方法关闭目录。

```
/* program: hello */
#include <nl_types.h>
#include <locale.h>
nl_catd catd;
main()
{
   /*  initialize the locale  */
   setlocale (LC_ALL, "");
   /* open the catalog */
   catd=catopen("hello.cat",NL_CAT_LOCALE);
printf(catgets(catd,1,1,"Hello World!"));
catclose(catd);                    /* close the catalog */
exit(0);
}
```

在上面的例子中，方法 **catopen** 通过文件名指示消息目录'hello.cat'。环境变量 **NLSPATH** 定义了消息目录的地点。如果消息目录被方法 **catopen** 成功打开，方法 **catgets** 返回一个指向'hello.cat'目录中的特定消息的指针。如果消息目录没被找到，或目录中不存在消息，方法 **catgets** 返回默认的字符串'Hello World!'。

认识环境变量 NLSPATH　环境变量 **NLSPATH** 指定寻找消息目录时搜索的路径。当方法 **catopen** 被调用以便定位并打开一个消息目录时，它根据指定的顺序搜索这些路径。如果消息目录没被找到，获取消息的方法返回程序提供的默认的消息。关于 **NLSPATH** 的默认路径，见 **/etc/environment** 文件。

获取程序提供的默认消息　如果需要的消息由于某种原因未能得到，所有的获取消息的方法均返回程序提供的默认消息文本。程序提供的默认消息通常是简单的单行消息文本，它不包含消息代号。愿意看到这些默认消息的用户可以将 **LC_MESSAGE** 目录设置为 C locale，或者不设置环境变量 **NLSPATH**。当环境变量 **LC_ALL**、**LC_MESSAGE** 或 **LANG** 均未被设置时，

LC _ MESSAGE 目录就默认为 C locale。

8.4.2 资源文件——Microsoft

Microsoft 提供了资源文件，它可被编译到应用程序执行时使用的动态连接库中。将本地化的字符串包含到应用程序中有两种方式。第一种是将所有的字符串包含在一个资源文件中，第二种则需要多个资源文件，二者均说明如下。

1. 单个资源文件

在开发环境中，打开一个资源文件，并插入一个字符串表。默认时，这个字符串表被用于保存本国语言的消息文本字符串。然后，拷贝这个字符串表，并改变与它相联系的语言。改变语言允许将新字符串表中的消息文本翻译为相应的语言。这些新的消息文本可被应用程序使用与原来的字符串表一致的 ID 引用，应用程序仅仅需要改变它所使用的语言。

一个可能的字符串表显示如下：

```
String Table
   |
   +---[abc] String Table
   +---[abc] String Table [English (U.K.)]
   +---[abc] String Table [French (France)]
```

2. 多资源文件

与前一个例子的不同之处是将三个字符串表中的每一个都保存在一个独立的资源文件中。使用哪一个资源文件是在编译时而不是在运行时决定的。决定的方法类似于如下代码：

```
#ifdef _FRENCH
#include "resourceFile_f.rc
#elif _ENGLISH_UK
#include "resourceFile_uk.rc
#else // default USA
#include "resourceFile_us.rc
#endif
```

8.5 开发国际化应用程序

要支持全球市场，应用程序必须被设计成在国际和国内同样有效。本节描述如何使用国际化的特点，为选择的地区生产应用程序，包括策划一个国际化的应用程序并设计它的界面。还讨论一些相关的问题，如，输入国际化数据、在国际化应用程序中进行数据排序、处理双字节字符集等。

8.5.1 策划一个国际化应用程序

要准备开发一个国际化应用程序，通常涉及三个步骤：建立数据、编写代码和设计用户界面。但是，在经历这些步骤之前，你需要考虑下面的问题：

▶ 什么样的数据是可接受的？

▶ 你如何编写一个国际化的应用程序?

▶ 当设计用户界面时你应该考虑哪些方面?

下面将谈到这些问题，并提出在你准备应用程序之前需要考虑的其他问题。

提示：*如果在最初时就将应用程序设计为一个国际化应用程序，而不是修改一个应用程序，使之符合以后的国际化用途，你能够减少开发国际化应用程序的代价，并能更快地将其推向市场。*

8.5.2 确定接受哪些数据

要决定哪些数据是可接受的，首先要考虑使用应用程序的地区。地区决定数据的文化内容，以及准备数据使用的语言。

另外，语言影响准备数据使用的code page（如，简体中文的code page为936——译者注）。code page是计算机用于适当显示数据的字符集，通常处理国际字符。国际字符包括那些含有发音符号的字符。发音符号被置于字母的上、下或中部，以标志与不带发音符号的字母的发音区别。最常用的发音符号是抑音（如à中`）、锐音（如á中的´）、长音（如 â 中的 ^ ）、变音（如 ä 中的 ¨ ）、环音（如 å 中的 ° ）和叉音（如 ø 中的/），，它们均与元音连用。

有些语言，如中文、韩文和日文，使用双字节字符集 DBCS 表示它们的数据。如果你的应用程序可能在这些环境下运行，你可能需要使用特殊的字符串处理函数和特殊的排序标准，以保证应用程序能恰当地工作。

8.5.3 编写代码

一个完整的应用程序由用户界面组件和程序功能组件组成。用户界面包括图形、文本字符串和与地区有关的设置，如日期、货币、数值格式和分隔符。程序功能组件含有所有地区共用的代码，包括处理用户界面中使用的字符串和图形的代码。设计你的应用程序时，要使用户界面组件与程序功能组件保持分离，因为相互独立的组件可使应用程序的本地化和维护更加容易。如，因为组件是独立的，当需要定位界面元素时，你就不必浏览所有的源代码了。

8.5.4 设计用户界面

用户界面中使用的菜单、表单、控件、工具条和位图都必须适合你为之设计应用程序的地区。例如，如果你为德国和法国的用户设计应用程序，对话框必须足够大，以保证对话框中的指令被翻译为德语和法语后，仍能够恰当地显示。另外，图标和位图中使用的图像必须符合地区文化，以保证它们在目标地区能被充分理解。

8.5.5 测试应用程序

要测试一个国际化的应用程序，你需要检查你为之设计应用程序的国家或地区以及从属语言。测试涉及检查应用程序的数据和用户界面，以保证它们的日期和时间、数值格式、货币、列表分隔符和度量单位均符合该地区的标准。

8.6 设计用户界面

当将一个应用程序本地化时，由于文本可能会增加，所以当设计下面的用户界面时要小心些。

- ▶ 应用程序的消息文本
- ▶ 菜单和表单
- ▶ 图标和位图

8.6.1 创建应用程序消息文本

当创建一个应用程序的消息文本时，英语文本字符串的长度通常小于其他语言中的等价字符串。下表显示了翻译后的字符串的平均增长率（与原始的英文字符串的长度相比）。

英文长度（以字符数表示）	本地化字符串的增长率
1~4	200%
5~10	180%
11~20	160%
21~30	140%
31~50	120%
50以上	110%

8.6.2 设计菜单和表单

正如前面的消息文本中提到的，在菜单和表单中显示本地化文本可能不合适，除非在设计时考虑到本地化后的文本长度的变化。在设计时给本地化后的文本预留一些空间，这样，本地化工程师就无需再浪费时间去调整控件的大小，也不需要重新设计界面。

8.6.3 使用图标和位图

如果图标和位图形式的图像被适当使用，它们能够成为应用程序用户界面的一个重要的部分。遗憾的是，使用图标和位图传达的语义比使用文字语言表达的意思更含糊。“一图绘千言”这样的说法是正确的，除非图也需要被翻译。使用图标和位图时，你应该遵从下面这些方针：

- ▶ 使用可被广泛识别的图像。如，使用信封代表邮件，而不要使用邮箱，因为它不是一个通用符号。
- ▶ 适当地使用颜色和模式。如，某些颜色组合可能具有地方性的重要意义。另外，也要理解色盲者的需要。交叉路口的交通信号灯有时闪动以指示优先权：红表示停止以保证通畅的路面，黄意味着警告。色盲的人们遇到这些信号时总是停止，因为他们不能分辨出红和黄。
- ▶ 位图中尽可能不要使用文字，因为翻译后文字的长度可能是一个问题，它会出现在界面的很多地方。

- 不要使用不可在多种文化之间翻译的任何东西。通常，这包括土语、俚语和诙谐语。
- 使用工具提示帮助解释图标的意义，工具提示的优点在于能够根据要显示的文本，自动扩展提示窗口的大小。
- 显示男人和女人时，要保证你绘出的图像与使用该应用程序的地区的文化取向一致。如果你不能确信一个图标或位图是否合适，要咨询你为之设计应用程序的地区的某些人。

8.7 建立可被移植的应用程序的目录结构

这个目录结构允许程序员将应用程序中的可移植部分从应用程序的其他部分分离出来。第9章讨论如何将这个目录结构整合为一个完整的项目目录结构。这个目录结构分布于应用程序的需要移植的组件中。'i18n' 组件有自己的目录，独立于因操作系统不同而变化的组件。

```
porting
    |-- exposed porting header files
    |-- package
        |-- private header files specific to package
        |-- component
            |-- source files directories
        |-- object files directory
        |-- library directory
```

例子

```
porting
    |-- exposed porting header files
    |-- messages
        |-- private header files specific to package
        |-- Unix message source files
            |-- source files directories
        |-- Microsoft resource files
        |-- object files directory
        |-- library directory
```

8.8 小结

本章涉及软件移植的基本层面。讨论到的主题包括以下几个方面：

- 移植到新的操作系统
- 移植到新的硬件平台
- 移植到新的语言 'i18n' 和 'l10n'
- 将消息中的字符串本地化
- 设计国际化应用程序
- 设计用户界面

最后，介绍了一个支持移植组件的应用程序的目录结构。

第 9 章　应用程序生命周期

本章将讨论以下内容：

- 学习书写代码文档的一个标准
- 理解一个结构化的目录布局的必要性
- 学习 make 工具的使用
- 学习源代码管理控制工具
- 学习如何记录应用程序源代码以外的其他方面

一个应用程序生命周期的三个阶段是分析、设计和实现。本章重点在最后一个阶段，即实现阶段。虽然我们不讨论使用哪一种编程语言，但要讨论增进可维护性的一些技术，包括维护应用程序源代码的两种普遍使用的工具：make 工具和代码管理控制工具。

本章中，我们还要讨论四种技术，它们有利于跟踪错误报告、改进需求、修改记录和回归测试。当管理大型项目时，这些技术会给你帮助。

9.1　写出源代码的文档

前面几章中，文档的主题是在项目这一级讨论的，即写出分析和设计的文档。本节讨论代码实现的文档，和一些增进代码可读性的技术。

写好代码的文档很重要，原因有两个：

- 如果人们理解一个类是被设计用于完成什么任务的，他们能更好地决定何时在自己的项目中复用该类。不准确的或含糊的文档妨碍类复用，因为人们不会复用那些他们不理解的类。
- 如果文档能够清晰地表明类的功能、哪一个方法实现特别的功能、以及它们是怎样实现的，那么，类的修改和支持就会更容易。如果没有描述目前任务的实现方式的文档，试图扩展一个方法的功能几乎是不可能的。

9.1.1　一般的注释

注释应该用于解释一个方法中的重要步骤，如，解释为什么代码以一种特别的方式实现。用一个注释告诉读者一个变量要加 1 了，简直是浪费时间，除非给出加 1 的原因（下面的例子中，使用了 C 语言风格的注释，这并没有什么特别的理由）。

```
/* comment */
```

或

```
/*
** comment
** another comment
*/
```

代码中的注释的例子如下：

```
/* comment */
code
```

或

```
code                                    /* comment */
```

或

```
if (test == 0)
{
   code                                /* comment */
}
else
{
   code                                /* comment */
}
```

9.1.2 C++文件的文档

本节说明怎样写出一个C++源文件的文档。文档内容包括以下几个方面：

- 一个引导头，解释源文件的作用，包括修改的历史记录
- 使用了哪些外部头文件
- 使用了哪些内部头文件

然后，每个方法都有它自己的解释性的文档。

```
/*
** Name of file
** What classes does this file provide implementations for
** What are the responsibilities of these classes
** Who created the file
** Change History
**     dd-mmm-yyyy (who made the change)
**         what is this change
*/

/*
** System include files
** include files from outside of the project environment
*/

/*
** Local include files
** Done in this order so that local include files do not try to
```

```
** define something that could be declared by the system include
** files include files from the project environment
*/

/*
** Method Header
** Method Name
** Purpose of method
**     Describe any algorithms used in this method
** Input
**     method takes no arguments
**     Arg1 - purpose and range of variable value
** Output/Return
**     method is void
**     return value 1 - when it succeeds
**     return value 2 - when it fails
** Change History
**     dd-mmm-yyyy (who made the change)
**         what is this change
*/
<return type>
method name (data_type argument_name, ...)
{
    /*
    ** local variables
    */
}
```

9.1.3 C++头文件的语法

所有的头文件都应该以一个条件编译标志开始，如下面所示。当编译器第一次包含这个头文件时，条件编译标志_ HEADER _ FILE _ NAME _ H被检测，如果该标志尚未被定义，编译器就继续包含这个头文件。头文件的第二行定义了这个标志，因此，如果编译器试图再次包含这个头文件，就会失败。这样，当编译一个源代码文件时，同一个头文件就不会被多次包含。对每一个被编译的源文件，条件编译标志被假定为未定义，直到一个头文件定义了该标志。重要的是，每一个头文件使用的条件编译标志的名字应该不相同。在标志的名字中使用头文件的文件名，就可解决这个问题。如果不能保证不同的头文件使用不同的条件编译标志名称，一个头文件就可能中止其他头文件的处理，因为检测标志时会失败。

```
#ifndef __HEADER_FILE_NAME_H
#define __HEADER_FILE_NAME_H

/*
** Who created the file
** Change History
**     dd-mmm-yyyy (who made the change)
**         what is this change
*/
/*
** forward declaration of classes
```

```
*/
class <class name>;
 /*
 ** What is the super class of this class (if any)
 */
public class <classname>
 {
    public:
        /*
        **  document the methods of this class that are for public
        **  consumption
        */
    protected:
        /*
        **  document the methods of this class that are for use
        **  by this and derived classes
        */
    private:
        /*
        **  document the methods of this class that are for use
        **  by this class only
        */
        /*
        **  For each variable (the scope if hopefully private)
        **      what is the variable used for
        **      what is it data type
        **      is the variable a class or instance variable
        */
};

#endif /* __HEADER_FILE_NAME_H
```

9.1.4 Java 文件的文档

Java 源文件的文档与 C++ 源文件中使用的文档基本相似。

```
/*
**  Name of file
**  What are the responsibilities of this class
**  Who created the file
**  Change History
**      dd-mmm-yyyy (who made the change)
**          what is this change
*/


/*
**  import files
*/

/*
```

```
**  The class definition statement.
**  Does this class extend another class?
**  Does this class implement any interfaces?
*/
public class <classname>

/*
**  For each variable:
**      what is the variable used for
**      what is its scope: public, private or protected
**      what is it data type
**      is the variable a class or instance variable
*/

/*
** Method Name
** Purpose of method
**     Describe any algorithms used in this method
** Input
**     method takes no arguments
**     Arg1 - purpose and range of variable value
** Output/Return
**     method is void
**     return value 1 - when it succeeds
**     return value 2 - when it fails
** Change History
**     dd-mmm-yyyy (who made the change)
**         what is this change
*/
<scope><return type>
method name (data_type argument_name, ...)
{
    /*
    ** local variables
    */
}
```

9.1.5 源代码语句的安排

以下是几个显示如何书写源代码的例子。

1. 条件

当条件列在 if - then - else 语句、while 循环或 for 循环语句中时，一定要将条件与一个可知的数值比较。语句：

```
if (method_call () == some_value)
```

就比下面这个依赖于一个方法的返回结果的方式好：

```
if (method_call ())
```

因为对前一个语句来说，如果 method_call 被修改，它可以依赖于一个可能变化的数值。

2. if-then-else

语句：

```
if (condition)
{
    <code statements>
}
else
{
    <code statements>
}
```

与下面的格式相比，当检查花括号的匹配关系时，更适合人眼视觉的习惯：

```
if (condition) {
    <code statements>
}
else {
    <code statements>
}
```

3. for 循环和 while 循环

与 if-then-else 语句一样，将花括号放在独立的一行的行首：

```
for (<initialization>; <condition_is_false>; <increment>)
{
    <code statements>
}
```

和

```
while (<condition_is_true>)
{
    <code statements>
}
```

如果 for 循环或 while 循环的使用仅是为了利用它们的副作用（如寻找字符串的结尾指针），可以写为无循环体的形式：

```
for (;*dst++ != *src++; )
    ;    /* void body */
```

和

```
while (*dest++ != *src++)
    ;    /* void body */
```

9.2 组织项目的目录结构

以下是一个例子，它说明了如何组织一个项目的目录结构，以方便管理和控制。尤其是当一个项目包含多个相对独立的组件时，若将它们放在各自的目录中更利于使用，所以更应当重视项目的目录结构。

每个项目都含有下列组件：

- 构造应用程序的工具和控制文件。
- 可以被移植的部分。
- 其他组件的公用部分。
- 应用程序的中枢或核心。
- 应用程序的前端或外部接口。
- 保存应用程序已构造的组件的构造目录结构。
- 一个模拟应用程序安装过程的虚拟安装目录。它与构造目录类似，但可能包括配置信息。
- 最后是一个包含所有测试程序的目录。

```
project
  |-- tools
      |-- makefiles
      |-- build scripts
  |-- porting
      |-- exposed porting header files
      |-- package
          |-- private header files specific to package
          |-- source files directories
              |-- component_1
              |-- component_2
          |-- object files directory
          |-- library directory
  |-- common
      |-- exposed common header files
      |-- package
          |-- private header files specific to package
          |-- source files directories
              |-- component_1
              |-- component_2
          |-- object files directory
          |-- library directory
  |-- back-end
      |-- exposed back-end header files
      |-- package
          |-- private header files specific to package
          |-- source files directories
```

```
            |-- component_1
            |-- component_2
        |-- object files directory
        |-- library directory
|-- front-end
    |-- exposed porting header files
    |-- package
        |-- private header files specific to package
        |-- source files directories
            |-- component_1
            |-- component_2
        |-- object files directory
        |-- executables directory
|-- buildArea
    |-- executabls
    |-- libraries
    |-- message_catalogs / resource files
|-- dummy_installation_area
    |-- executabls
    |-- libraries
    |-- message_catalogs / resource files
    |-- other_files (configuration_files, header_files
                    - if needed)
|-- test_suite
    |-- test1
        |-- project
            |-- test1 source files
            |-- test1 object files
            |-- test1 library - if needed
            |-- test1 executable - if needed
```

9.3　使用 make 工具

make 工具用于保持一系列文件（即目标）的一致更新。该工具使用一个被称为 makefile 的文件来包含处理指令。指令描述怎样从源创建一个目标以及使用的命令。每个目标的源本身也可能是一个目标，可能有自己的创建指令。

make 工具以递归的方式检查每个目标和它的源。如果处理完所有的源文件后，发现目标文件丢失了或比任何一个源文件都旧，make 工具就重新构造它。

要构造一个给定的目标，make 工具执行一个命令列表，该列表称为规则（*rule*）。这个规则可能被显式地列在目标的 makefile 项中，或者由 make 工具隐含支持。

如果在命令行中没有指定目标，make 工具使用 makefile 中定义的第一个目标，这通常被定义为‘all’。

9.3.1 选项

支持的选项如下：

-f *makefile*	make工具使用的文件的文件名通常是makefile。但，‘-f’允许其他文件名作为使用的文件的名称，如： make -f <name> 使用描述文件*makefile*。‘-’作为*makefile*参数，指示标准输入。*makefile*的内容，如果存在的话，重载标准的隐含规则和预定义的宏。当多于一个的‘-f *makefile*’对出现时，make工具使用这些文件的级联，级联的顺序与文件出现的顺序相同
-n	非执行模式。打印命令，但不执行它们。以@开头的偶数行被打印。但是，如果一个命令行包含对宏$（MAKE）的引用，该行总是被执行。当处于POSIX模式时，以“+”开头的行被执行
-s	Silent模式。执行命令行之前不打印命令行。与特定功能目标.SILENT：相同

说明：如果工作目录中有一个名为makefile的文件，make则使用该文件。但是，如果有一个较新的SCCS历史文件（SCCS/s.makefile），make尝试获取并使用它的最新的版本。

缺少上述文件时，如果工作目录中存在一个名为Makefile的文件，make则尝试使用该文件。如果有一个较新的SCCS历史文件（SCCS/s.Makefile），make尝试获取并使用它的最新的版本。当没有指定makefile时，POSIX模式的/usr/ccs/bin/make和/usr/xpg4/bin/make则按顺序尝试下面的文件。

./makefile，./Makefile

s.makefile，SCCS/s.makefile

s.Makefile，SCCS/s.Makefile

还存在其他选项，但它们已超出本书的范围。

9.3.2 操作数

make工具可被用于指定的目标，如：

```
make target
```

这种使用方法的一个例子是“make clean”，用于清除对象、库和可执行文件，以备重建一切项目。

提示：宏定义。该定义使用指定的宏重载makefile文件或环境中所有的常规定义。但是，这个定义还能被条件宏赋值重载。

```
make macro=value
```

这种使用方法的一个例子是“make DEBUG=1”。这个宏与本章后面“Makefiles实例”一节中的“OS_SPECIFIC.defs”中的适当指令一起，可被用于激活调试标志‘-g’的编译操作，编译时加上供调试工具使用的信息。

9.3.3 读取 makefile 和环境

make 工具阅读命令行中的其他选项，这些选项也会起作用。接着，make 读入默认的 makefile，它通常包含预定义的宏、隐含规则的目标项，以及其他规则，如获取 SCCS 文件的规则。

然后，make 工具从环境引入变量，并将它们作为定义的宏。因为 make 使用遇到的最新的定义，所以，一个在 makefile 中的宏定义通常重载环境中的一个同名的变量。

再后，make 工具读入你使用 -f 指定的所有 makefile，或者其中的一个 makefile 或 Makefile，按照“选项”部分描述的方式。

make 工具最后读入命令行参数中提供的所有宏定义。这些定义重载 makefile 和环境中的宏，当然只在该 make 命令本身重载，不会影响以后的 make 命令。

9.3.4 makefile 目标项

makefile 中的目标项格式如下：

```
## the '->' is used to indicate a tab character
target... [:|::] [dependency] ... [; command] ...
   ->   [command]
```

第一行包含一个目标的名称，或一个由空格分开、以冒号（:）结束的目标名称的列表。若给出一个目标列表，则相当于对每一个目标有一个同样格式的独立项。冒号之后，通常是一个依赖条件或一个依赖条件列表。构造目标之前，make 工具先检查这个列表。依赖条件列表可能以一个分号（;）结束，然后跟一个 Bourne Shell 命令。目标项的随后几行以一个 TAB 键开始，并包含 Bourne Shell 命令。这些命令组成构造目标的规则。

Shell 命令可能使用转义符号（\）延伸到下一行，如 \n。延伸的行也必须以一个 TAB 键开始。

要重新构造一个目标，make 工具展开宏，去掉最初的 TAB 键，要么直接执行命令（如果它不含有 shell 元字符），要么将每个命令行传到一个 Bourne shell 命令执行。

首行若不以一个 TAB 键或‘#’开始，则标志着另一个目标或宏定义的开始。

9.3.5 特殊字符

表 9-1 简要列出 makefile 中的字符的意义。

表 9-1　makefile 字符表

特殊字符	描　述
全局的	
#	开始一个注释。注释在下一个换行符处结束。如果“#”在一个命令行的 TAB 键之后，该行被传送到 shell（shell 仍然将“#”当作一个注释的开始）
include *filename*	如果 *include* 是一行的头 7 个字符，并且后面有一个空格或 TAB 键，则随后的字符串被当作一个插入在该行的文件名。这种格式允许嵌套，嵌套的级数不超过 16。如果 *filename* 是一个宏引用，则该宏被展开

（续）

特殊字符	描 述
目标和依赖关系	
:	目标列表的结束符。冒号之后的部分被加入目标或目标列表的依赖条件列表 例子 clean:src_clean
%	模式匹配通配符的元字符。与shell通配符“*”一样，在条件宏定义中，或在一个模式替换宏引用中，“%”可用于匹配目标名称或依赖条件中的任意字符串（由零个或多个字符组成）。注意，在一个目标、一个依赖条件名称，或一个模式替换宏引用中，“%”只能出现一次 例子 %.o:%.c
宏	
=	宏定义。该字符左边的那个单词是宏的名称，右边的那些单词组成宏的值。开头和结尾处的空白字符被从值中剔除。“=”字符之后隐含有一个单词隔断
$	宏引用。随后的字符、括号或花括号中的字符串被解释为一个宏引用：make展开这个引用（包括$），用该宏的值来替换它
( ) 或 { }	宏引用名称的界定符。$后面的、由括号或花括号括起来的单词被当作将要引用的宏的名称。若没有界定符，make只将它后面的第一个字符作为宏的名称
$$	对dollar符宏的一个引用，该宏的值为字符$。这种方式用于向shell传送以$开头的变量表达式、引用将被shell展开的环境变量，或者在一个目标的依赖条件列表中延迟动态宏的处理，直到该目标被实际处理
\$	转义的dollar符。在规则中被解释为一个字母的dollar符
+=	当代替“=”使用时，在一个宏定义中加上一个字符串（与“=”不同，必须用空白符括起来）
规则	
@	如果第一个非TAB键的字符是一个@，make在执行该命令行之前不打印它。这个字符不被传送到shell
?	规避命令依赖检查。以该字符开头的命令行将不进行命令依赖检查
!	强制命令依赖检查。命令依赖检查将对命令行强制进行，如果不使用“!”，这些命令行的检查可能被禁止。对那些包含“?”动态宏引用的命令行（如，“$?”）来说，命令依赖检查通常被禁止

当“@”、“?”或“!”的任意组合出现在TAB键后的第一个字符位置时，三者所有的规则都适用。不会向shell传送任何内容。

9.3.6 特殊功能目标

下面的目标名称在makefile中使用时，将执行特殊的功能：

.INIT:	如果在makefile中定义，这个目标和它的依赖关系将在处理其他任何目标之前被构造
.SILENT:	无声运行。当这个目标出现在makefile中时，make在执行命令之前不显示它。当在POSIX模式中使用时，它后面可能跟随一些目标的名称，只有这些目标将被无声执行

9.3.7 后缀替换宏引用

在宏中可以进行替换，形式如下：

$（name：string1 = string2）

一个替换的例子如下：

```
SRC = x.c y.c z.c
OBJ = $(SRC:.c=.o)
# which gives x.o y.o and z.o
```

这里，string1要么是一个后缀，要么是一个在宏定义中将被替换的单词，string2是替换后的后缀或单词。宏的值中的单词之间以空格、TAB键，或转义换行字符隔开。

1. 动态宏

有几个动态维护的宏，在规则中用作缩写比较好。这些宏的引用被列于表9-2。如果你要定义它们，make就重载原有的定义。

表9-2　动态宏

名　称	说　明
$*	当前目标的基本名称，被衍生出，与一个隐含的规则共同使用
$<	依赖文件的名称，被衍生出，与一个隐含的规则共同使用
$@	当前目标的名称，这是惟一一个在依赖条件列表中使用时值被严格确定的动态宏（此时它的形式为"$$@".）

$?	比目标更新的依赖条件列表。对于包含该宏的行，命令依赖检查自动被禁止，就如同命令有一个前缀"?"。请见表9-1"特殊字符"部分中关于"?"的描述。你可以使用命令行前缀"!"来强制进行命令依赖检查
	例子：
	`print : *.c`
	`  -> $(PRINT) $?`
	`  -> touch print`
	变量"$?"用于只打印那些被改变过的文件

在一个依赖条件列表中引用动态宏"$@"时，需要在引用之前再加一个字符"$"（即，"$$@"）。因为make像对隐含规则一样对"$<"和"$*"赋值（按照后缀列表和目录内容），当与显式的目标项一起使用时可能不安全。

2. 隐含规则

如果makefile中不存在某个目标的对应项，make则尝试确定该目标的类别（如果有的话），并将规则应用于该类别。一个隐含规则描述如何从相关的依赖文件中构造属于一个给定类别的任何目标。目标类别的确定可根据一个模式或一个后缀，以及目标将由此构造的相关的依赖文件（具有相同的基本名称）。除了预定义的一套隐含规则外，make还允许你通过模式或后缀定义你自己的隐含规则。

3. 执行命令

命令行一次被执行一条，每一条的执行都是通过它自身的过程或shell。shell命令（如常用的*cd*）在makefile中非转义换行情况下是无效的。命令行在执行之前被打印出来（宏已经被展开）。如果命令以"@"开头，或makefile中有一个".SILENT:"目标项，或者make运行时有"-s"选项，就不打印执行的命令行了。虽然make的选项"-n"指定只打印命令行而不执

行，但包含宏“$（MAKE)”的行还是被执行，包含特殊字符“@”的行会被打印。“-t”选项只更新一个文件的修改日期，而不执行任何规则。如果源文件被多人维护，这种方式可能是危险的。

4. Bourne Shell 方案

如果控制结构是有分支的，要使用Bourne Shell，即使用如下形式的命令行：

```
if expression ; \
then command ; \
... ; \
else command ; \
... ; \
fi
```

虽然有7个输入行之多，但转义换行字符保证make将所有这些行当作一个（Shell）命令行。

如果控制结构是循环，要使用Bourne Shell，即使用如下形式的命令行：

```
for var in list ; \
do command; \
... ; \
done
```

要引用一个Shell变量，需使用双dollar符（$$），这样能够避免make展开dollar符。

5. 诊断

表9-3给出了诊断信息及说明。

表9-3 诊断信息及说明

诊断信息	描述
Don't know how to make target *target*	makefile中没有与 *target* 对应的目标项，而且make中没有适用的隐含规则（后缀列表中没有具有后缀的依赖文件，或者目标的后缀不在后缀列表中）
*** *target* removed	构造目标 *target* 时make被中断。这时，make不是将部分完成的新于依赖条件的目标文件留下，而是将名为 *target* 的文件删除
*** *target* not removed	构造目标 *target* 时make被中断，且 *target* 在目录中不存在
*** *target* could not be removed, *reason*	构造目标 *target* 时make被中断，且 *target* 由于指定的原因（*reason*）不能被删除
Read of include file *file* failed	在指定包含路径中的makefile未被找到，或者不可访问
Loop detected when expanding macro value *macro*	在宏定义中发现对正在定义的宏的引用
*** Error code *n*	先前的Shell命令返回一个非零的错误代码
Conditional macro conflict encountered	只在使用‘-d’选项时显示这个错误信息，它表明当前正被处理的两个或更多的并行目标依赖于一个目标，而这个目标由于条件宏的原因，对每个依赖它的目标来说构造的结果不同。由于这个目标不可能同时满足这些依赖关系，就引起冲突

9.3.8 makefile 的例子

下面这个例子说明，pgm依赖于两个文件a.o和b.o，而这两个文件又依赖于各自的源文件（a.c和b.c）以及一个共用的文件incl.h。

1. 简单的显式 makefile

```
pgm: a.o b.o
$(LINK.c) -o $@ a.o b.o
a.o: incl.h a.c
cc -c a.c
b.o: incl.h b.c
cc -c b.c
```

2. 简单的隐含 makefile

下面的makefile使用隐含规则来表达同样的依赖关系：

```
pgm: a.o b.o
cc a.o b.o -o pgm
a.o b.o: incl.h
```

9.3.9 可移植 makefile 的例子

以下的makefile例子体现一些通用的定义、特定操作系统的定义、通用的规则和特定操作系统的规则，然后是应用程序目录结构中每个层次的makefile，包括库和可执行文件的不同的makefile。“ - > ”用于指示继续前一行需要的8字符TAB键。

1. genericMakefile.defs

```
## Setup the macro to include to commonly used values
INCLUDE             = -I$(PROJECT_SRC)/$(part)/hdr
DEFINES             =

## Add extra header files if the package is only 'pack1' or 'pack2'
## The test below searches for the package name in the
## comma separated list, which consists of 'pack1' and 'pack2'.
## If the result is positive, the package name does not equal the
## first argument of the 'ifneq', which by default is an empty string
## and so the commands are executed
ifneq (,$(findstring $(package), pack1, pack2))
    CUSTOM_INCLUDE     = \
   ->   -I$(PROJECT_SRC)/$(part)/$(another_package)/hdr
    CUSTOM_DEFINES     = -DANSI
    CUSTOM_LIBRARIES   = -l<package1lib>
endif

## Setup aliases for libraries
## example for the shared library 'libpackage1'
```

```
## and the static library 'libpackage2'
package1name       = libpackage1.$(SH_lib_EXT)
package1lib        = $(PROJECT_BUILD_LIB)/$(package1name)

package2name       = libpackage2.$(SH_lib_EXT)
package2lib        = $(PROJECT_BUILD_LIB)/$(package2name)

include $(PROJECT_MAKE_DIR)/${OS_SPECIFIC}.defs
```

2. OS _ SPECIFIC.defs

```
PROJECT_SOURCE     = <some directory>
PROJECT_BUILD      = $(PROJECT_SOURCE)/build
PROJECT_BUILD_BIN  = $(PROJECT_BUILD)/bin
PROJECT_BUILD_LIB  = $(PROJECT_BUILD)/lib
PROJECT_MAKE_DIR   = $(PROJECT_BUILD)/makefiles

ST_lib_EXT         = a
SH_lib_EXT         = so

OBJ_EXT            = o

#
# Platform dependant include dirs
#
INCLUDE            += -I/usr/include -I/usr/ucbinclude -I.

ifdef DEBUG
OPTFLAGS           += -g
CPP_OPTFLAGS       += -g
else
OPTFLAGS           += -xO5
CPP_OPTFLAGS       += -xO5
endif

ifneq (,$(findstring $(package), package1))
DEFINES            += -D<operating_specific_define>
endif

## library archive command
AR                 = /usr/ccs/bin/ar
## compilers
CC                 = <compiler location>/cc
CPP                = <compiler location>/CC
## copy files
CP                 = /bin/cp
## directory manipulation
MKDIR              = /bin/mkdir -p
RMDIR              = /bin/rm -rf
```

```
## remove files
RM                  = /bin/rm -rf
## Compiler flags
cflags              += ...
CCFLAGS             += ...


## AIX specific flag to force characters to be signed
CCFLAGS             += -qchars=signed
cflags              += -qchars=signed


CC.options          = $(OPTFLAGS) $(INCLUDE) $(DEFINES) $(CFLAGS)
CPP.options         = $(CPP_OPTFLAGS) $(INCLUDE) $(DEFINES) $(CFLAGS)
## Compiler command
CC.comp             = $(CC) -c $(CC.options)
CPP.comp            = $(CPP) -c $(CPP.options)

## Library flags
LDFLAGS             = ...
## Building library command
CC.link             = $(CC) $(OPTFLAGS) $(LDFLAGS)

## Standard libraries = usulally system libraries
STDLIBS             = ... -lpthread -lm
```

3. genericMakefile.rules

```
all-packages:
   ->   for package in boo $(packages); do \
   ->       if [ -d "$$package" ]; then \
   ->       ->   (cd $$package && \
   ->       ->   $(MAKE) all && exit 1);\
   ->       fi;\
   ->   done; exit 0;

all-clean:
   ->   for package in boo $(packages); do \
   ->       if [ -d "$$package" ]; then \
   ->       ->   (cd $$package;\
   ->       ->   ($(MAKE) clean || exit 1));\
   ->       fi;\
   ->   done; exit 0;

src-clean:
   ->   (cd $(PROJECT_SOURCE)/$(part)/$(package); \
   ->   for objfile in boo $(OBJ_FILES); do \
   ->       if [ -f "$$objfile" ]; then \
   ->       ->   $(RM) $$objfile; \
```

```
->        fi; \
->    done;)

include $(PROJECT_MAKE_DIR)/${OS_SPECIFIC}.rules
```

4. OS_SPECIFIC.rules

```
## the rules the build to a library can either be placed here if it
## has operating specific needs
```

5. Makefile 部分

```
##
##  Makefile for top of tree part e.g. back
##
##  History:
##      <date> <author>
##          <description>
##

export part     = <part>

include $(PROJECT_MAKE_DIR)/genericMakefile.defs

packages = package_1 package_2

all:    all-components
clean:  all-clean

include $(PROJECT_MAKE_DIR)/genericMakefile.rules
```

6. 库文件 Makefile

```
##
##  makefile for the <part>/<package> library
##
##  History:
##      <date> <author>
##          <description>
##
part                = <part>
package             = <package>

include $(PROJECT_MAKE_DIR)/genericMakefile.defs

SRC_FILES           = \
    src/<component>/<source_file_1> \
    src/<component>/<source_file_2>

OBJ_FILES           = \
```

```
    obj/<object_file_1> \
    obj/<object_file_2>

OBJDIR              = ./obj
BINDIR              = ./bin
LIBDIR              = ./lib
INCDIR              = ./hdr
SRCDIR              = ./src

INCLUDE             += $(CUSTOM_INCLUDE)

LIBS                += $(CUSTOM_LIBS) $(STDLIBS)

all : $(OBJDIR) $(package1lib)
clean: src-clean

$(package1lib): $(OBJ_FILES)
   ->    $(CPP.link) -o $(LIBDIR)/$(package1name) $(LDFLAGS) \
   ->        $(OBJ_FILES) $(LIBS)
   ->    $(CP) $(LIBDIR)/$(package1name) $(package1lib)

obj/%.o: src/<component>/%.c
   ->    $(CC.comp) $(CCFLAGS) $(DEFINE) $(INCLUDE) $< -o $@

include $(PROJECT_MAKE_DIR)/genericMakefile.rules
```

7. 可执行文件 Makefile

```
##
## makefile for the <part>/<package> executable
##
## History:
##     <date> <author>
##         <description>
##

part                = <part>
package             = <package>

include $(PROJECT_MAKE_DIR)/genericMakefile.def

SRC_FILES           = \
    src/<component>/<source_file_1> \
    src/<component>/<source_file_2> \
    src/<component>/<source_file_3> \
    src/<component>/<source_file_4>

OBJ_FILES           = \
    obj/<object_file_1> \
```

```
    obj/<object_file_2> \
    obj/<object_file_3> \
    obj/<object_file_4>

OBJDIR              = ./obj
BINDIR              = ./bin
LIBDIR              = ./lib
INCDIR              = ./hdr
SRCDIR              = ./src

INCLUDE             += $(CUSTOM_INCLUDE)

LIBS                += $(CUSTOM_LIBS) $(STDLIBS)

all : $(OBJDIR) <executable1>
clean:  src-clean

$(PROJECT_BUILD_BIN)/<executable1>: $(OBJ_FILES)
   ->   $(CC.comp) -o $(BINDIR)<executable1> $(OBJ_FILES) \
   ->       -L$(PROJECT_BUILD_LIB) $(LIBS)
   ->   $(CP) $(BINDIR)<executable1> $(PROJECT_BUILD_BIN)

obj/<component_name>.$(OBJ_EXT): src/<component_name>/<source_file_1>
   ->   $(CC) $(CCFLAGS) $(DEFINE) $(INCLUDE) \
   ->       src/<component_name>/<source_file_1>

include $(PROJECT_MAKE_DIR)/genericMakefile.rules
```

9.3.10 创建依赖条件

当一个文件的依赖条件被创建时，makefile中用户命令之后的行显示如下。有一个语句声明将被遵循的依赖条件。obj文件显示在左边，它依赖的文件在右边。只要右边的文件被更新，就会导致左边的文件被重新构造。

```
# DO NOT DELETE THIS LINE - make depend depends on it.
obj/<object_file_1>: $(PROJECT_SRC)/$(<part>)/$(<package>)/hdr/<header_file>.h
```

9.4 使用源代码管理控制工具

源代码管理控制是一个有用的工具，因为它允许程序员跟踪对应用程序的源代码的修改记录。一个源代码管理控制系统用于记载一个文件的不同版本之间的区别，以及解释为何进行修改的相关注释。

这里讨论的源代码管理控制系统是一个在UNIX平台上使用的系统，名为SCCS。

9.4.1 源代码管理控制系统

本节包括源代码管理控制系统（SCCS）的描述、可被使用的各种命令，以及最后的一个

显示如何使用SCCS的简单例子。

1. **描述**

命令 *sccs* 是 SCCS 的各种工具程序的一个综合的、直接的前端命令。*sccs* 将指定的子命令用到指定的文件的相关历史文件中。

一个 SCCS 历史文件的名称是在工作文件的文件名前加一个前缀“s.”得到的。*sccs* 命令通常期望这些 s. 文件保存在一个 SCCS 子目录中。因此，当你给 *sccs* 提供一个 *file* 参数时，它通常将子命令应用于 SCCS 子目录下一个名为 s.*file* 的文件。如果 *file* 是一个带路径的文件名，*sccs* 就在该文件的父目录的 SCCS 子目录下查找历史文件。如果 *file* 是一个目录名，*sccs* 就将子命令应用于该目录下的每一个 s.*file* 文件。因此，命令

```
example% sccs get program.c
```

将把子命令 get 应用到一个名为

```
SCCS/s.program.c
```

的历史文件，而命令

```
example% sccs get SCCS
```

将把子命令 get 应用到 SCCS 子目录下的每一个 s. 文件。

命令 *sccs* 本身的选项必须出现在子命令参数之前。指定子命令的选项必须出现在子命令参数之后。这些选项因每一个子命令而不同，将与这些子命令本身一起说明（见下一节中关于子命令的解释）。

2. **操作数**

支持的操作数如下：

- ▶ **子命令**　一个 SCCS 工具名或列在“用法”列表中的其中一个拟工具名。
- ▶ **选项**　传送到子命令的一个选项或选项参数。
- ▶ **操作数**　传送到子命令的一个操作数。

3. **用法**

表 9-4 的 *SCCS* 子命令中的大部分激活保存在/usr/ccs/bin 中的程序。这些子命令中的大部分接受另外的参数，这些参数记录在子命令激活的工具程序的参考页中。

表 9-4　*sccs* 的子命令

命　令	描　述
check [- b]	检查那些当前正在编辑的文件。与 info 和 tell 一样，但返回一个退出代码，而不是产生一个文件的列表。如果一个文件正在被编辑，check 返回一个非零的退出状态码 - b 忽略分支
clean [- b]	清除当前目录中能够从一个 SCCS 历史恢复的所有的一切。不清除那些正被编辑的文件 - b 不管是否正在被编辑，不检查分支 如果分支版本保存在同一个目录，子命令 clean - b 是危险的

(续)

命 令	描 述
create	创建（初始化）历史文件。子命令 *create* 进行以下的步骤： 1. 将原始的源文件在当前目录下重新命名为，program.c 2. 在 SCCS 子目录中创建名为 s.program.c 的历史文件 3. 对 program.c 进行一个‘sccs get’操作，获取一个只读的最初版本的副本
deledit [- s] [- y [*comment*]]	等同于一个‘sccs delta’操作后再加上一个‘sccs edit’操作。deledit 将文件检入，再将文件检出，但不改变文件的当前工作副本 - s 沉默。不报告检入数据和统计信息 - y [*comment*] 提供检入版本的注释信息。如果省略 - y 选项，系统将向用户提示输入注释。*comment* 为空意味着检入版本中的注释域为空
delget [- s] [- y [*comment*]]	进行一个‘sccs delta’操作后，再进行一个‘sccs get’操作，检入一个版本并获取新版本的只读副本。关于 - s 和 - y 选项的描述，请见子命令 *deledit*。*sccs* 命令对参数列表中指定的所有文件进行 delta 操作，然后对所有这些文件进行一个 get 操作。如果在 delta 操作过程中出现错误，get 操作就不进行了
delta [- s] [- y [*comment*]]	将对源文件的修改检入（check in）。记录文件被检出（check out）以来引入的每一行的变化。有效用户的 I.D. 必须与检出该文件的人的 I.D. 一致。参考 sccs - delta (1)。关于 - s 和 - y 的描述，请见子命令 deledit
diffs [- c*date - time*] [- r *sid*]	比较被检出用于编辑的一个文件的当前工作副本与 SCCS 历史中的一个版本之间的差别。默认时使用检入的最新的版本 - c*date - time* 用指定的日期和时间以前检入的最新版本比较。*date - time* 子命令的形式为：*yy* [*mm* [*dd* [*hh* [*mm*] *ss*]]]]]。被省略的单位默认为它们的最大可能值，即 - c7502 与 - c750228235959等同 - r*sid* 用与指定的版本号对应的版本进行比较
edit	获取文件的一个版本用以被编辑。命令 *sccs* edit 抽取该文件的一个你可写的版本，并在 SCCS 子目录下创建一个 p.file，作为对该文件历史的一个锁定，以保证其他任何人都不能对该版本检入或检出 I.D. 关键字被以非展开的方式获取 子命令 *edit* 与 get 接受同样的选项，如下
enter	与 create 类似，但省略最后的‘sccs get’操作。如果历史文件被初始化后，要立即进行一个‘sccs edit’操作，可使用 enter
get [- ek] [- c*date - time*] [- r [*sid*]]	从 SCCS 历史中获取一个版本。默认时，得到一个最新版本的只读工作副本，I.D. 关键字为展开的形式 - e 取出文件用于编辑 - k 禁止展开 I.D. 关键字。使用 - e 即意味着使用 - k
help *message - code* | *sccs - command*	帮助信息命令 提供关于 SCCS 诊断的更多信息。当你提供由 SCCS 显示出的诊断信息代号时，*help* 子命令显示该错误的简短解释。如果你提供一个 SCCS 命令的名称时，它显示出有关的使用方法介绍。help 命令也能提供关键字的帮助信息。参见 sccs - help (1)
info [- b]	显示当前被编辑的文件的列表，包括被检出的版本号、将被检入的版本号、持有编辑权限的用户名，和文件被检出的日期和时间 - b 忽略分支

（续）

命　令	描　述
print	打印每个指定文件的所有历史操作
prs [-el] [-c*date* - *time*] [-r*sid*]	详细显示一个 s.file 的版本列表或其他内容 -e 显示在指定版本之前创建的那些版本的信息，其中用 -r 或 -c 描述指定版本 -l 显示在指定版本之后创建的那些版本的信息，其中用 -r 或 -c 描述指定版本
sact	显示一个 SCCS 文件的编辑活动状态。参见 sccs - sact (1)
sccsdiff -r*old* - *sid* -r*new* - *sid*	使用 diff 比较与指定 SID 对应的两个版本
tell [-b]	显示当前处于检出状态的文件的列表，每个文件占一行。 -b 忽略分支
unedit or unget	"撤销（undo）"上一次的编辑或'get -e'，并使文件的当前工作副本恢复到以前的状态。unedit 撤销所有自文件被检出以来所做的一切未被检入的修改

4. 环境

关于影响 *sccs* 运行的环境变量，见 environ（5）中的描述：LC_CTYPE、LC_MESSAGES 和 NLSPATH。

如果变量以一个斜杠开始（绝对路径名），SCCS 在该变量指定的目录下查找 SCCS 历史文件。如果 PROJECTDIR 不以一个斜杠开始，表明它得自一个用户名，那么，SCCS 在该用户的基础目录的 src 或 source 子目录下查找历史文件。如果找到这样的目录，它就使用这个目录；否则，就将变量值作为一个相对路径名。

5. SCCS——文件

以下是与 SCCS 工具有关的文件和目录列表：

- SCCS *SCCS* 子目录
- SCCS/d. file　表示差别的临时目录
- SCCS/p. file　检出版本的锁（许可）文件
- SCCS/q. file　临时文件
- SCCS/s. file　SCCS 历史文件
- SCCS/x. file　s. file 的临时副本
- SCCS/z. file　临时锁文件
- /usr/ccs/bin/ *　SCCS 工具程序

6. *SCCS* 关键字

如果没有 -e 和 -k 选项，get 子命令会展开表 9-5 中的关键字，即用获取的源文件的文本中指定的值来替代它们。

表 9-5　*sccs* 关键字和值

关键字	值
%A%	I.D. 行的快捷表示法，含有其他方面的数据（1）：%Z% %Y% %M% %I% %Z%
%B%	SID 分支组件

（续）

关键字	值
%C%	当前行序号。它是为了用于标识程序输出的信息，如“this shouldn’t have happened”类的错误。它不是为了用于表示每一行的序号
%D%	当前日期：*mm*/*dd*/*yy*
%E%	适用的版本的最新日期被创建：*mm*/*dd*/*yy*
%F%	SCCS s.file 的名称
%C%	适用的版本的最新日期被创建：*mm*/*dd*/*yy*
%H%	当前日期：*mm*/*dd*/*yy*
%I%	被获取的版本的 SID：%R%.%L%.%B%.%S%
%L%	SID 级别组件
%M%	模块名称：去掉前缀的 s.file 名称
%P%	s.file 的完整名称
%Q%	s.file 中的 q 标志的值
%R%	SID 发行组件
%S%	SID 顺序组件
%T%	当前时间：*hh*:*mm*:*ss*
%U%	适用的版本的最新时间被创建：*hh*:*mm*:*ss*
%W%	I.D. 行的快捷表示法，含有其他方面的数据：%Z% %M% %I%
%Y%	模块类型：s.file 中 t 标志的值
%Z%	4字符字符串：‘@（#）’，由其他识别

9.4.2 SCCS 的例子

创建源安全文件。这个文件有79行，版本为1.1：

```
>> sccs create endian.cpp
endian.cpp:
No id keywords (cm7)
1.1
79 lines
No id keywords (cm7)
```

打印目录：

```
>> ls
,endian.cpp   SCCS          endian        endian.cpp
```

检查是否有被打开、正在编辑的文件：

```
>> sccs check
```

检查目录中的文件与SCCS控制之下的文件之间的差别：

```
>> sccs diffs endian.cpp

------- endian.cpp -------
```

将文件检出，以被编辑。以前的版本为1.1，新版本为1.2：

```
>> sccs edit endian.cpp
1.1
new delta 1.2
79 lines
```

检查能被编辑的文件：

```
>> ls -l
total 36
-rw-r--r--   1 jasmine  jasii       941 Feb  1 15:20 ,endian.cpp
drwxrwxr-x   2 jasmine  jasii       512 Feb 16 14:25 SCCS
-rwxr-xr-x   1 jasmine  jasii     11052 Feb  1 15:20 endian
-rw-r--r--   1 jasmine  jasii       941 Feb 16 14:25 endian.cpp
```

检查SCCS目录：

```
>> ls SCCS
p.endian.cpp  s.endian.cpp
```

撤销文件，以保证从目录中删除它：

```
>> sccs unget endian.cpp
1.2
```

再检查该目录：

```
>> ls -l
total 34
-rw-r--r--   1 jasmine  jasii       941 Feb  1 15:20 ,endian.cpp
drwxrwxr-x   2 jasmine  jasii       512 Feb 16 14:26 SCCS
-rwxr-xr-x   1 jasmine  jasii     11052 Feb  1 15:20 endian
```

打印文件和一些修订历史：

```
>> sccs print endian.cpp
SCCS/s.endian.cpp:

D 1.1 01/02/16 14:23:23 jasmine 1 0     00079/00000/00000
MRs:
COMMENTS:
date and time created 01/02/16 14:23:23 by jasmine

1.1      /*
1.1      ** endian.cpp
1.1      ** This application is used to test endism.
```

```
1.1     **
1.1     ** Expected results:
1.1     **      Big Endian          Little Endian
1.1     **      12 34 56 78         78 56 34 12
1.1     **      12 34 56 78         34 12 78 56
1.1     **      41 42 43 00         41 42 43 00
1.1     */
...
1.1     int main(int argc, char* argv[])
1.1     {
...
1.1     /*
1.1     ** Set the pointer to the long variable
1.1     ** and then print the 4 bytes of memory
1.1     */
1.1         ptr = (char *)&temp.l;
1.1
1.1     .   printValue (ptr);
1.1
1.1     /*
1.1     ** Set the pointer to the short array variable
1.1     ** and then print the 4 bytes of memory
1.1     */
1.1         ptr = (char *)&temp.s;
1.1
1.1         printValue (ptr);
1.1
1.1     /*
1.1     ** Set the pointer to the char array variable
1.1     ** and then print the 4 bytes of memory
1.1     */
1.1         ptr = (char *)&temp.c;
1.1
1.1         printValue (ptr);
1.1
1.1         return 0;
1.1     }
```

将文件检出，以被编辑：

```
>> sccs edit endian.cpp
```

编辑该文件：

```
>> vi endian.cpp
```

显示文件的编辑活动：

```
>> sccs sact endian.cpp
1.1 1.2 jasmine 01/02/16 14:28:24
```

检查目录中的该文件与SCCS控制下的该文件之间的差别：

```
>> sccs diffs endian.cpp

------- endian.cpp -------
58d57
<
66d64
<
74d71
<
```

显示被编辑的文件：

```
>> sccs tell
endian.cpp
```

显示被编辑的文件的更多的历史记录：

```
>> sccs info
  endian.cpp: being edited: 1.1 1.2 jasmine 01/02/16 14:28:24
```

撤销文件（清除对文件进行的所有的编辑）：

```
>> sccs unedit endian.cpp
  endian.cpp: removed
1.1
79 lines
No id keywords (cm7)
```

再显示任何被编辑的文件的历史：

```
>> sccs info
Nothing being edited
```

再次检出文件以被编辑。以前的版本是1.1，新的版本是1.2：

```
>> sccs edit endian.cpp
1.1
new delta 1.2
79 lines
```

提交进行的改变，然后获取该文件（只读）：

```
>> sccs delget endian.cpp
comments? this change removes the blank lines between the assignments
statements and the method invocation
No id keywords (cm7)
1.2
0 inserted
3 deleted
76 unchanged
```

```
1.2
76 lines
No id keywords (cm7)
```

打印文件和一些修订历史：

```
>> sccs print endian.cpp
SCCS/s.endian.cpp:

D 1.2 01/02/16 14:38:56 jasmine 2 1     00000/00003/00076
MRs:
COMMENTS:
this change removes the blank lines between the assignment statements
and the method invocation

D 1.1 01/02/16 14:23:23 jasmine 1 0     00079/00000/00000
MRs:
COMMENTS:
date and time created 01/02/16 14:23:23 by jasmine

1.1     /*
1.1     ** endian.cpp
1.1     ** This application is used to test endism.
1.1     **
...
1.1         ptr = (char *)&temp.l;
1.1         printValue (ptr);
...
1.1         ptr = (char *)&temp.s;
1.1         printValue (ptr);
...
1.1         ptr = (char *)&temp.c;
1.1         printValue (ptr);
1.1
1.1         return 0;
1.1     }
```

再次检出文件以被编辑：

```
>> sccs edit endian.cpp
sccs edit endian.cpp
1.2
new delta 1.3
76 lines
```

编辑该文件：

```
>> vi endian.cpp
```

提交进行的改变，然后获取该文件（只读）：

```
>> sccs delget endian.cpp
comments? put blank lines back
No id keywords (cm7)
1.3
3 inserted
0 deleted
76 unchanged
1.3
79 lines
No id keywords (cm7)
```

打印这个文件，以及一些修订历史：

```
>> sccs print endian.cpp
SCCS/s.endian.cpp:

D 1.3 01/02/16 14:41:38 jasmine 3 2     00003/00000/00076
MRs:
COMMENTS:
put blank lines back

D 1.2 01/02/16 14:38:56 jasmine 2 1     00000/00003/00076
MRs:
COMMENTS:
this change removes the blank lines between the assignment statements
and the method invocation

D 1.1 01/02/16 14:23:23 jasmine 1 0     00079/00000/00000
MRs:
COMMENTS:
date and time created 01/02/16 14:23:23 by jasmine

1.1     /*
1.1     ** endian.cpp
1.1     ** This application is used to test endism.
1.1     **
...
1.1         ptr = (char *)&temp.l;
1.3
1.1         printValue (ptr);
...
1.1         ptr = (char *)&temp.s;
1.3
1.1         printValue (ptr);
...
1.1         ptr = (char *)&temp.c;
1.3
1.1         printValue (ptr);
1.1
```

```
1.1          return 0;
1.1      }
```

显示这个文件的第一个版本与最后一个版本之间的差别：

```
>> sccs sccsdiff -r 1.1 -r 1.3 endian.cpp
SCCS/s.endian.cpp: No differences
```

显示这个文件的第一个版本与第二个版本之间的差别：

```
>> sccs sccsdiff -r 1.1 -r 1.2 endian.cpp
58d57
<
66d64
<
74d71
<
```

显示这个文件的第二个版本与第三个版本之间的差别：

```
>> sccs sccsdiff -r 1.2 -r 1.3 endian.cpp
57a58
>
64a66
>
71a74
>
```

显示这个文件的历史：

```
>> sccs prs endian.cpp
SCCS/s.endian.cpp:

D 1.3 01/02/16 14:41:38 jasmine 3 2      00003/00000/00076
MRs:
COMMENTS:
put blank lines back

D 1.2 01/02/16 14:38:56 jasmine 2 1      00000/00003/00076
MRs:
COMMENTS:
this change removes the blank lines between the assignments
statements and the method invocation

D 1.1 01/02/16 14:23:23 jasmine 1 0      00079/00000/00000
MRs:
COMMENTS:
date and time created 01/02/16 14:23:23 by jasmine
```

9.5 错误报告

错误是指一个导致功能失败的事件，也就是说无法按照计划实施工作进程。当你向应用程序开发团队报告一个错误时，需要提供某些信息。每个新的错误报告需要有一个惟一的错误代号。

错误报告中需要的信息列举如下：

- **错误日期**　报告该错误的日期。
- **错误报告人**　报告该错误的人的姓名和修改错误后被通知的人的姓名。
- **错误优先级**　该错误被关注的优先级。优先级越高，表示错误报告人认为该错误越重要。
- **错误描述**　该错误的简述和全部描述，以便错误报告阅读者能得知错误的所有内容。
- **错误重现步骤**　重现该错误的操作步骤的描述，以及有助于重现该错误使用的任何脚本。

9.6 改进需求

改进是一个增加新功能或扩展现有功能的请求，需要的信息如下：

- 一个惟一的代号，以便该需求可被惟一地索引。
- 请求改进的人员的姓名。
- 改进的描述：改进的目的，以及改进为何是必需的。

9.7 修改记录

这种技术也可被用于记录已经实现的请求，或记录对现有错误的修正。记录的信息如下：

- 与这个修改有关的错误或改进的惟一的代号。
- 描述修改是如何实现的。
- 修改的日期和修改人员姓名。
- 描述与原始的错误或改进需求有关的修改是如何测试的。
- 描述这个修改的影响。它需要局部的组件重新构造吗？或者，它需要项目中的其他部分重新构造吗？
- 这个修改影响到的所有组件的列表，用于确定这个修改的影响。
- 实现这个修改时改变的文件的列表。
- 最后，所有的变化的列表，逐个文件、逐行地列出。这个列表应该显示出源代码中增加、改变和删除的代码行。

9.8 回归测试

每当对源码进行修改时，提供一个回归测试是必要的。回归测试提供一种机制，以保证当源码改变时，以前正常工作的功能依然正常。需要的信息如下：

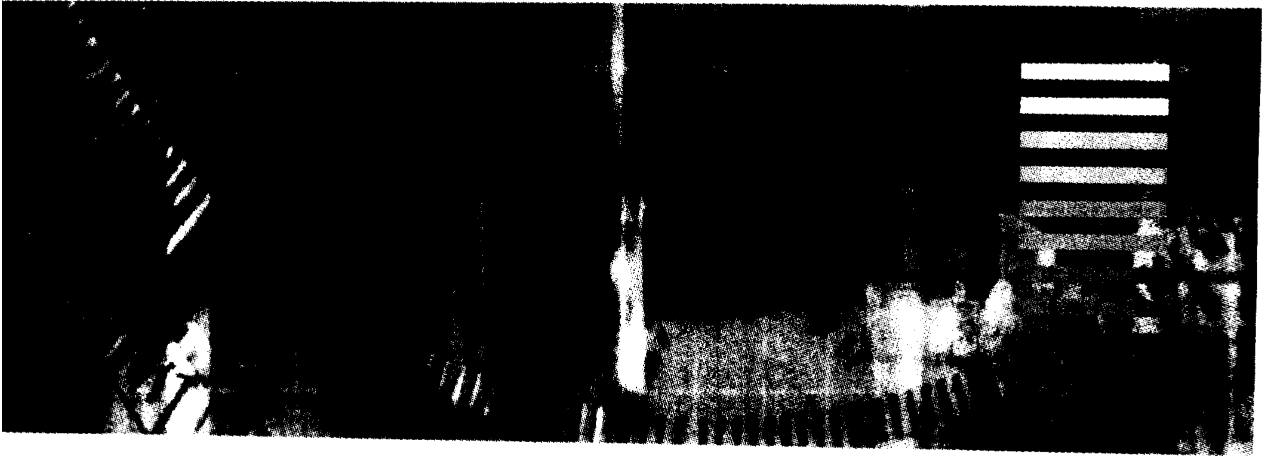
- 一个惟一的测试代号

- 测试人员的姓名
- 创建测试的日期
- 被测试的功能
- 与该测试有关的修改记录的代号
- 实现测试采用的步骤和与测试有关的任何脚本。测试步骤包括应用程序环境的配置，以及应用程序本身，包括装载需要的数据。

9.9 小结

本章向你引见了如下内容：

- 书写源代码文档的一个标准
- 一个结构化的项目目录布局，它可以将应用程序的逻辑部件分离开来
- make 工具，用于构造应用程序
- 源代码管理控制工具，它对管理重要的应用程序的源代码非常必要
- 关于错误报告、改进需求、代码修改和回归测试的记录和文档



第五部分 实 例 学 习

目标：

- 通过一个简单的应用程序实例，理解分析文档的使用
- 理解开发多线程应用程序时遇到的问题及其解决方法

第10章　实例学习1——一个模拟的公司

本章将讨论以下内容：

- 分析游戏 SimCo
- 根据 SimCo 的具体情形演示第2章中的每一种分析文档

这是一个假想的小型制造公司的应用程序的例子，它演示了对应用程序分析和设计的过程中产生的各种文档。这个应用程序名为 SimCo，它代表了被模拟的公司。

10.1　项目需求

这个应用程序模拟一个小型的制造公司。设计出的应用程序将使得用户能够获得贷款、购置机器，并通过一系列的月度生产运营，监视公司的绩效表现。

作为这个学习实例的一部分，将创建下列的分析图：

(1) 产生系统的用例。

(2) 产生最初识别出的类的类图。

(3) 创建用于体现每个类与系统的关系的 CRC 卡片。

(4) 使用类关系图显示类之间的相互关系。

(5) 使用活动图来描述每个用例的“流向”。

(6) 使用脚本记录每个用例中类实例之间的交互关系。

(7) 从脚本中得出序列图和协作图。

(8) 使用状态图来描绘生命周期较长的对象的模型。

10.2　用例

针对我们的企业应用程序，我已经识别出五个用例：

- **提出贷款申请**　这个假想的公司可能需要一笔贷款，用来购置更多的机器设备，或者只是用作流通资金。
- **购置机器设备**　要想做点事情，公司必须购置生产商品的机器设备。
- **生产运营**　生产运营过程就是用户决定生产多少商品、以什么样的价格出售商品的过程。可以向用户提供计划销售额和售出商品的可预期的平均价格的指示信号。
- **计算出公司的细节信息**　公司的账目要每月更新一次。这些账目从销量、日常开支（如员工工资、房租）等得出毛利，再计算出公司的纯利。另外，像存货清单、销售额细节这样的详细信息也要更新。

▶ **显示公司的详细信息** 向用户显示公司当月或前五个月的交易活动的一览信息。

10.2.1 用例模板的翻版

这个用例模板最初是在第 2 章引入的。用例模板用于细化应用程序的高级层面，见表 10-1。

表 10-1 用例模板

部分		描述
用例序号		<简短的命名>
描述		该用例的详细解释
前提		要使这个用例能够工作，系统需要处于什么样的条件下。如，商店要卖东西必须首先开张
触发条件		是什么导致这个用例开始工作？如，顾客需要商品，并进入商店
成功		用例完成后，系统处于什么样的状态？如，顾客拥有了所有需要的商品，并感到很愉快。货币保存在出纳机中，有人重新上货，等待下一位顾客
中止		如果用例被放弃了，会发生哪些情况？如，如果顾客放下了购物篮，没买任何东西就离开了商店，就需要有人看到这些，并将货物放回到货架上
参与者	主要的	谁起主导作用？如，顾客和收款员？
	从属的	谁起次要作用？如，店员？

正如第 2 章中提到的，用例模板中的以下几个部分分别表现正常处理过程、正常处理过程中的任何变化部分以及可能出现的异常情况。

过程	步骤（Step #）	简短的命名<action>	描述
变更	步骤（Step #）	<action>	描述
异常	步骤（Step #）	<action>或 use case	描述

下面还有一些信息，可对每个用例提供较全面的描述。

- ▶ 这个用例的优先级怎么样？
- ▶ 用例预期运行多长时间？
- ▶ 用例发生的频度怎样？
- ▶ 与参与者之间的交互界面：
  - ▶ **交互的** 如每个顾客与商店都需要“对话”。
  - ▶ **静态的** 如被扫描的商品的价格。大多数商品的价格是静态的。
  - ▶ **定期的** 如货物上架是由专门的工人在职员班次轮换时进行的。
- ▶ **不定的因素** 很多情况可能影响用例，如更多的顾客、更多的职员，或更大的商店。
- ▶ **交付日期** 何时系统必须完成？
- ▶ **相关的用例** 如，处理货物上架的用例和收集顾客丢下的购物篮的用例。

10.2.2 Use Case #1——贷款申请

这个假想的公司可能需要一笔贷款，用来购置更多的机器设备，或者只是用作流通资金。图 10-1 表示了获得贷款的用例，随后的表 10-2 表明了这个用例的模板。

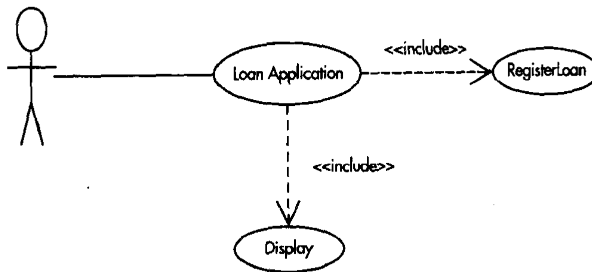


图 10-1 贷款申请

表 10-2 Use case #1 模板

Use case #1	贷 款 申 请		
描述	向系统请求贷款，然后输入贷款的数量和偿还的月度期限。并将贷款打入公司的现金账目		
前提	没有前提条件		
触发条件	用户需要选择这个选项		
成功	创建一笔贷款，并记入公司的账目。借得的款项归入公司的现金账户		
中止	不创建贷款，款项不归入公司的现金账户		
参与者	主要的	用户	
	从属的	无	
过程	Step #	Short Name < action >	描述
	1	PromptAmount	提示用户输入贷款的数量
	2	CheckAmount	检查申请的贷款数量大于零
	3	PromptDuration	提示用户输入贷款的期限
	4	CheckDuration	检查输入的贷款期限大于零
	5	PromptConfirm	提示用户确认贷款的细节。详细列出总的贷款数量，包括利率和还款计划
	6	ConfirmLoan	告诉用户贷款已经被确认
	7	CreditMoney	将款项划入公司的现金账户
	8	RegisterLoan	将贷款期限和还款计划记入公司的账目
变更	Step #	< action >	描述
	1	UseGUI	不使用文本式的交互方式，而使用图形界面（GUI）显示一个对话框
异常	Step #	< action > 或 use case	描述
	2a	BadAmount	如果贷款数量小于或等于零，可用于表示不想再继续申请贷款
	2b	Exit	退出贷款申请
	4a	BadDuration	如果贷款期限小于或等于零，可用于表示不想再继续申请贷款
	4b	Exit	退出贷款申请
	6a	Decline	谢绝贷款
	6b	Exit	退出贷款申请

10.2.3 Use Case #2——购置机器设备

公司为了生存，必须购置机器设备，以生产商品。图 10-2 表示了购置设备的用例，表10-3 表明了这个用例的模板。

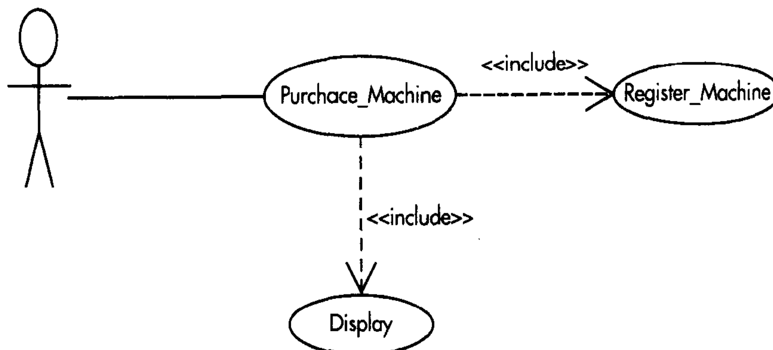


图 10-2 购置设备

表 10-3 Use case #2 模板

Use case #2	购　置　设　备		
描述	向系统请求一台设备，用于生产商品		
前提	公司的现金账户中需要有足够的款项，而不至于负债		
触发条件	用户需要选择这个选项		
成功	买到一台设备，并在公司登记。从公司的现金账户中划去花费的款项		
中止	不买设备。不从公司的现金账户中划去款项		
参与者	主要的	用户	
	从属的	无	
过程	Step #	Short Name <action>	描述
	1	DisplayDetails	显示设备的细节信息
	2	PromptPurchase	提示用户购置设备
	3	ConfirmPurchase	告诉用户购置交易已经被确认
	4	DebitMoney	从公司的现金账户中划去花费的款项
	5	RegisterMachine	在工厂中登记购置的设备
变更	Step #	<action>	描述
	1	UseGUI	不使用文本式的交互方式，而使用图形界面（GUI）显示一个对话框
异常	Step #	<action>或 use case	描述
	3a	Decline	谢绝购置设备
	3b	Exit	退出购置设备

10.2.4 Use Case #3——生产运营

在生产运营过程中，用户决定生产多少商品、以什么样的价格出售商品。可以向用户提供计划销售额和售出商品的可预期的平均价格（Average Price）的指示信号。图 10-3 表示了生产运营的 Use case，表 10-4 表明了这个 Use case 的模板。

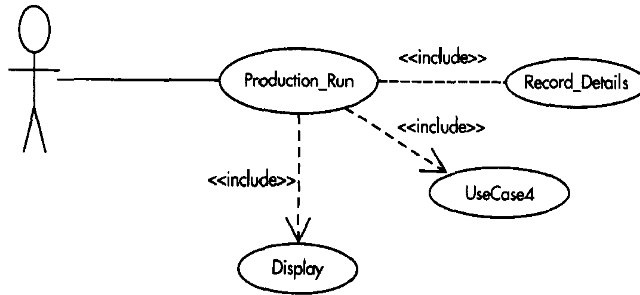


图 10-3 生产运营

表 10-4 Use case #3 模板

Use case #3	生 产 运 营		
描述	生产商品，制定销售价格，并将商品售出		
前提	生产商品的机器设备必须存在		
触发条件	用户需要选择这个选项		
成功	商品被生产出来并被售出		
中止	一切原封不动		
参与者	主要的	用户	
	从属的	无	
过程	Step #	Short Name <action>	描述
	1	ShowProjectedSales	向用户显示生产的商品的计划销售额
	2	NumberToMake	提示用户输入将要生产的商品的数量
	3	CheckAmount	检查数值是大于零还是等于零，因为零意味将售出现有的库存
	4	CheckMax	检查将被生产的商品数量不超过现有设备的生产能力
	5	ShowAvePrice	显示商品的平均市场价格
	6	SalePrice	提示用户输入商品的售价
	7	CheckPrice	检查输入的价格大于零
	8	NumberForSale	计算现有库存的商品的数量
	9	NumberSold	计算售出的商品的数量
	10	RecordSold	记录售出的商品的数量
	11	RecordUnsold	记录未售出的商品的数量，因为这个数量就是下个月的库存
	12	AdjustMarket	调整下个月的计划销售额和平均价格
	13	RecordSales	记录销售额
	14	CalcCostToMake	计算原材料的成本
	15	PayCostToMake	从现金账户中减去原材料的成本
	16	CalcGrossProfit	计算毛利：销售额减去生产成本
变更（Variations）	Step #	<action>	描述
	1	UseGUI	不使用文本式的交互方式，而使用图形界面（GUI）显示一个对话框

（续）

Use case #3	生 产 运 营		
异常（Exceptions）	Step #	< action > 或 use case	描述
	3a	BadAmount	如果数量小于零，可用于表示不想再继续生产运营
	3b	Exit	退出生产运营
	4a	TooMany	如果输入的数量大于设备生产能力，则提示用户重新输入
	4b	Exit	退出生产运营
	6a	BadPrice	如果数量小于零或等于零，可用于表示不想再继续生产运营
	6b	Exit	退出生产运营

可将下面的这些信息加入这个用例，以利于描述这个用例。

- 这个用例的优先级怎么样？优先级为次高。
- 用例发生的频度怎样？用户每进行一轮生产，就需要执行一次。
- 与参与者之间的交互界面：
  - **交互的** 即，用户确定生产的商品的数量和销售价格。
  - **静态的** 即，工厂的生产能力是静态的。
  - **定期的** 即，一旦本用例完成，就要运行“计算出公司的细节信息”（即 Use Case #4）。
- 相关的用例有哪些？“计算出公司的细节信息”（即 Use Case #4）和“显示公司的详细信息”（即 Use Case #5）。

10.2.5 Use Case #4——处理公司的账务

这个公司的账目要每月更新一次。这些账目从销量、日常开支（如员工工资、房租）等得出毛利（Gross Profit），再计算出公司的纯利（Net Profit）。另外，像存货清单、销售额细节这样的详细信息也要更新。图 10-4 表示了处理公司账务的用例，表 10-5 表明了这个用例的模板。

图 10-4 处理公司账务

表 10-5 Use case #4 模板

Use case #4	计算出公司的细节信息		
描述	在每个生产运营月度的结尾，计算出公司的账务		
前提	没有前提条件		
触发条件	生产运营月度的结束		
成功	公司账目被更新		
中止	不做任何更新		
参与者	主要的	生产运营	
	从属的	无	
过程	Step #	Short Name < action >	描述
	1	GetLoan	从现有的贷款计算出月还款额度
	2	GetOtherExpenses	计算其他的月度开支

（续）

Use case #4	计算出公司的细节信息		
	3	CalcExpenses	计算总的月度开支
	4	GetGrossProfit	从销售额账目得到毛利
	5	CalcNetProfit	从毛利中减去开支（得到纯利）
	6	GetCash	在现金账户中结算余额
	7	AddNetProfit	将纯利加到现金余额中
	8	RecordBalance	记录新的现金余额
	9	AdjustLoan	调整贷款偿还数量

下面是附加的信息：

相关的用例有哪些？“生产运营”（即 Use Case #3）。

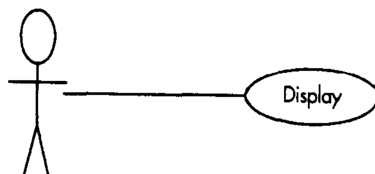


图 10-5 公司详细信息显示

10.2.6 Use Case #5——显示公司的详细信息

向用户显示公司当月和前五个月的交易活动一览信息。图 10-5 表示了显示公司的详细信息的用例，表 10-6 表明了这个用例的模板。

表 10-6 Use case #5 模板

Use case #5	显示公司的详细信息		
描述	向用户显示公司的全面细节信息。以表格的形式显示信息，表中的一列代表一个月		
前提	没有前提条件		
触发条件	用户需要选择这个选项		
成功	显示公司的所有细节信息		
中止	N/A		
参与者	主要的	用户	
	从属的	无	
过程	Step #	Short Name < action >	描述
	1	ShowMonth	
	2	ShowBalance	显示一个月的现金余额
	3	ShowNetProfit	显示一个月的纯利
	4	ShowSales	显示一个月的销售额
	5	ShowCost	显示一个月的生产成本
	6	ShowGrossProfit	显示销售额中的毛利
	7	ShowExpenses	显示开支费用
	8	ShowAvePrice	显示商品的平均市场价格
	9	ShowSalePrice	显示用户设定的价格
	10	ShowInstock	显示库存的商品数量
	11	ShowMade	显示生产的商品数量
	12	ShowForSale	显示可供销售的商品数量
	13	ShowProjectedSales	显示商品的计划销售额
	14	ShowSold	显示售出的商品数量
	15	ShowEndStock	显示未售出的商品数量

10.3 分析文档——类的静态特性

下面几种文档提供有关类的信息。它们也记录这些类怎样与系统进行交互。这些文档是：

▶ 类图
▶ CRC 卡片
▶ 脚本

10.3.1 类图

从上一节描述的五个用例中，浏览一下用例的部件，可以确定应用程序中包含九个类，见图 10-6。

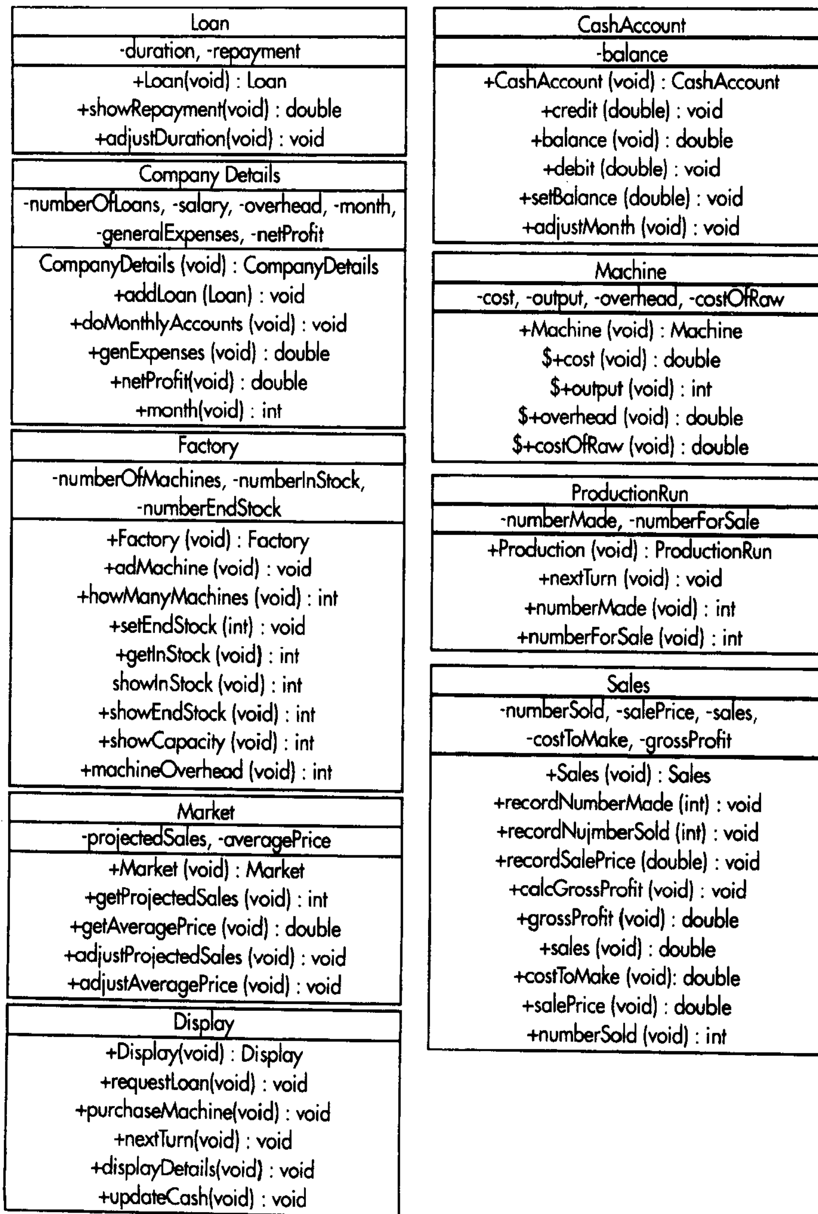


图 10-6 应用程序的九个类的类图

10.3.2 CRC卡片

确定了应用程序中的类之后，表10-7中就是CRC卡片，我将用它记录类的计划的职责和类之间的协作关系。

表10-7 CRC卡片

Class:	Loan
职责:	管理贷款的细节
协作类:	Display, CompanyDetails, CashAccount
Class:	CashAccount
职责:	管理公司的现金
协作类:	Loan, CompanyDetails, Machine, Display
Class:	CompanyDetails
职责:	管理公司的细节信息
协作类:	Display, Loan, CashAccount, Sales, Factory
Class:	Machine
职责:	管理设备的细节信息
协作类:	Display, Factory, CashAccount
Class:	Factory
职责:	管理工厂子系统
协作类:	Display, ProductionRun, Machine, CompanyDetails
Class:	ProductionRun
职责:	管理生产运营
协作类:	Display, Factory, Market, Sales
Class:	Market
职责:	管理影响销售额的外部因素
协作类:	Display, ProductionRun
Class:	Sales
职责:	管理销售额账务
协作类:	Display, ProductionRun, CompanyDetails
Class:	Display
职责:	管理用户界面
协作类:	CashAccount, CompanyDetails, Factory, Market, Sales, ProductionRun, Machine

10.3.3 脚本

每一个脚本记录一个类与系统中其他类之间的交互方式。

1. 类User

类 User 是驱动应用程序的外部实体，见表 10-8。

2. 类 Display

类 Display 将用户提出的请求传给系统中的其他类，见表 10-9。

3. 类 Loan

类 Loan 驱动两个外部交互活动，将贷款额加入公司的现金账户，并增加公司的贷款的总数，见表 10-10。

4. 类 Machine

类 Machine 驱动两个外部交互活动，从公司的现金账户中减去设备的花费，并增加公司的设备总数，见表 10-11。

5. 类 ProductionRun

类 ProductionRun 驱动多个外部交互活动，其中每一个都与计算商品生产信息有关，见表 10-12。

表 10-8 类 User

调用者	方 法	被调用者	结 果	User Case #
: User	*RequestLoan* ()	: Display	选择贷款选项	1
: User	*PurchaseMachine* ()	: Display	选择新设备选项	2
: User	*NextTurn* ()	: Display	选择下一个循环	3
: User	*DisplayDetails* ()	: Display	选择显示细节选项	5

表 10-9 类 Display

调用者	方 法	被调用者	结 果	User Case #
: Display	*Loan* ()	: Loan	创建一笔贷款	1
: Display	*Balance* ()	: CashAccount	显示最近更新的余额	1, 2, 5
: Display	*Machine*:: *mccost* ()	: Machine	显示一台设备的费用	2
: Display	*Machine*:: *mcoutput* ()	: Machine	显示一台设备的生产量	2
: Display	*Machine*:: *mcoverhead* ()	: Machine	显示一台设备的日常费用	2
: Display	*Machine*:: *mcrawcost* ()	: Machine	显示原料的费用	2
: Display	*Machine* ()	: Machine	创建一台新设备	2
: Display	*HowManyMachines* ()	: Factory	显示已经拥有的设备的数量	2
: Display	*Capacity* ()	: Factory	获得工厂中的设备的生产能力	3
: Display	*ShowInstock* ()	: Factory	显示上月底没有售出的商品的数量	5
: Display	*ShowEndStock* ()	: Factory	显示仍未售出的商品的数量	5
: Display	*NextTurn* ()	: ProductionRun	开始一个新的生产运营月度	5
: Display	*NumberMade* ()	: ProductionRun	显示当月生产的商品的数量	5
: Display	*NumberForSale* ()	: ProductionRun	显示待售的商品总数量	5

（续）

调用者	方法	被调用者	结果	User Case #
: Display	*month* ()	: CompanyDetails	显示当月	5
: Display	*NetProfit* ()	: CompanyDetails	显示从销售额中减去所有花费的纯利	5
: Display	*GeneralExpenses* ()	: CompanyDetails	显示其他的费用，如工资	5
: Display	*sales* ()	: Sales	显示销售额	5
: Display	*CostToMake* ()	: Sales	显示生产商品时原料的费用	5
: Display	*GrossProfit* ()	: Sales	显示从销售额中减去成本的毛利	5
: Display	*SalePrice* ()	: Sales	显示待售商品的价格	5
: Display	*numberSold* ()	: Sales	显示当月售出的商品的数量	5
: Display	*averagePrice* ()	: Market	显示待售商品的市场平均价格	5
: Display	*projectedSales* ()	: Market	显示商品的计划销售额	5

表 10-10 类 Loan

调用者	方法	被调用者	结果	User Case #
: Loan	*credit* ()	: CashAccount	增加公司的现金账户的资金量	1
: Loan	*addLoan* ()	: CompanyDetails	在公司细节信息中登记这笔贷款	1

表 10-11 类 Machine

调用者	方法	被调用者	结果	User Case #
: Machine	*debit* ()	: CashAccount	从公司的现金账户中减去设备的花费	2
: Machine	*addMachine* ()	: Factory	增加公司的设备总数	2

表 10-12 类 ProductionRun

调用者	方法	被调用者	结果	User Case #
: ProductionRun	*recordNumberMade* ()	: Sales	保存生产的商品的数量	3
: ProductionRun	*recordSalePrice* ()	: Sales	保存商品的销售价格	3
: ProductionRun	*recordNumberSold* ()	: Sales	保存当月售出的商品的数量	3
: ProductionRun	*calcGrossProfit* ()	: Sales	从销售额和生产成本计算毛利	3
: ProductionRun	*getInstock* ()	: Factory	获得上月底未售出的商品的数量	3
: ProductionRun	*setEndStock* ()	: Factory	设置本月末售出的商品的数量	3
: ProductionRun	*projectedSales* ()	: Market	获得生产的商品的计划销售额	3
: ProductionRun	*averagePrice* ()	: Market	获得生产的商品的平均价格	3
: ProductionRun	*adjustAvePrice* ()	: Market	根据随机因素调整平均价格	3
: ProductionRun	*adjustProjSales* ()	: Market	根据随机因素调整计划销售额	3

6. 类 CompanyDetails

类 CompanyDetails 驱动有关月底账目计算的信息，见表 10-13。

表 10-13 类 CompanyDetails

调用者	方 法	被调用者	结 果	User Case #
: CompanyDetails	*machineOverhead* ()	: Factory	获得一台设备运行的日常开支	4
: CompanyDetails	*adjustDuration* ()	: Loan	将所有贷款的期限减少一个月	4
: CompanyDetails	*showRepayment* ()	: Loan	获得本月的贷款偿还额	4
: CompanyDetails	*grossProfit* ()	: Sales	从销售额中得到毛利	4
: CompanyDetails	*adjustMonth* ()	: CashAccount	从前一个月得到余额	4
: CompanyDetails	*credit* ()	: CashAccount	记录现金余额的盈亏	4

7. 类 Factory

类 Factory 负责回答设备日常开支的问题，见表 10-14。

表 10-14 类 Factory

调用者	方 法	被调用者	结 果	User Case #
: Factory	*mcoverhead* ()	: Machine	返回一台设备运行的日常费用	4

10.4 分析文档——类的动态特性

状态图就是记录类的动态特性的分析文档。

状态图

每个类都用单独的状态图表示。该类的对象一旦被创建，其状态保持不变，直到应用程序结束。

类 Loan 见图 10-7。

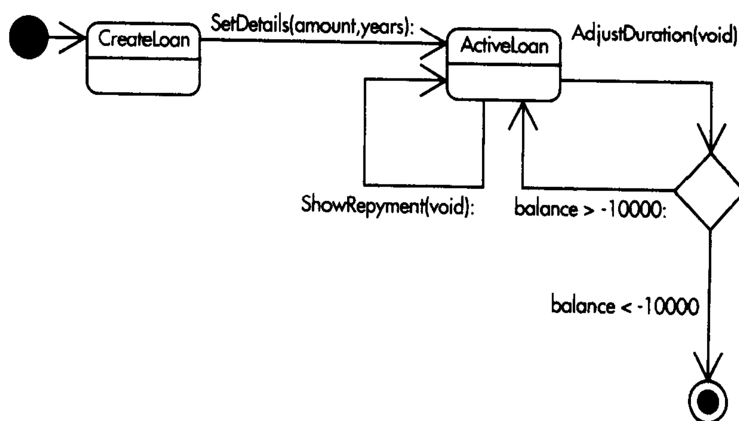


图 10-7 类 Loan 的状态图

类 CashAccount 见图 10-8。

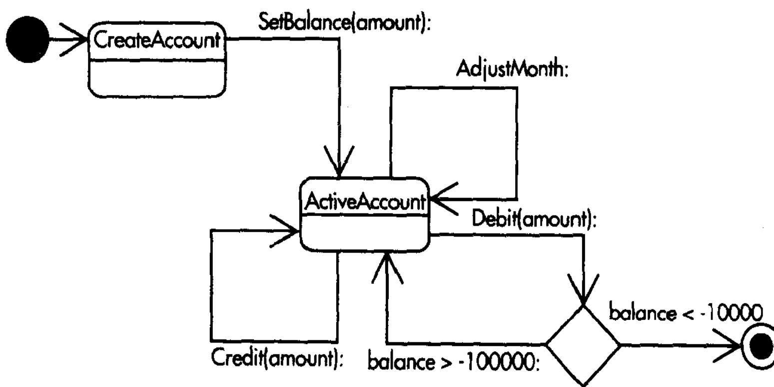


图 10-8　类 CashAccount 的状态图

类 ProductionRun 见图 10-9。

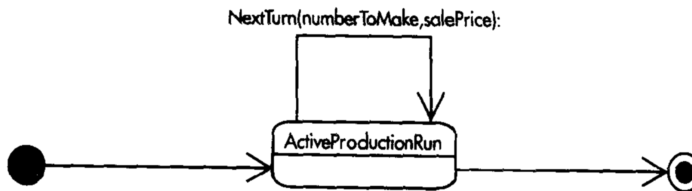


图 10-9　类 ProductionRun 的状态图

类 Market 见图 10-10。

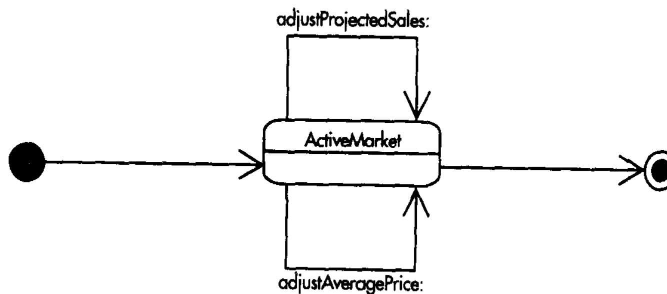


图 10-10　类 Market 的状态图

类 Sales 见图 10-11。

类 CompanyDetails 见图 10-12。

类 Machine 见图 10-13。

类 Factory 见图 10-14。

类 Display 见图 10-15。

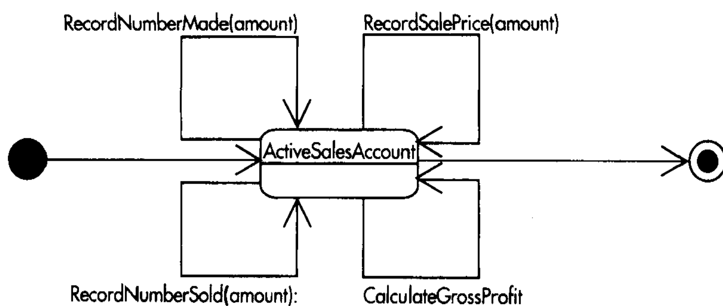


图 10-11　类 Sales 的状态图

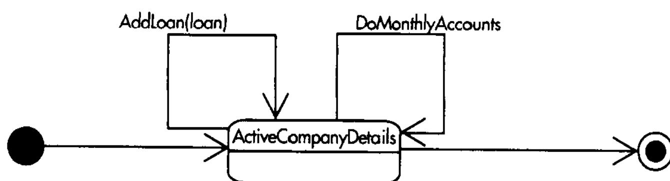


图 10-12　类 CompanyDetails 的状态图

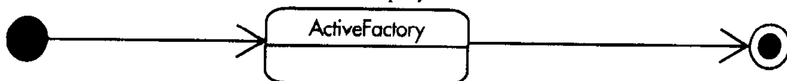


图 10-13　类 Machine 的状态图

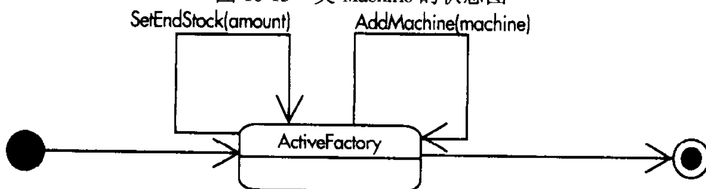


图 10-14　类 Factory 的状态图

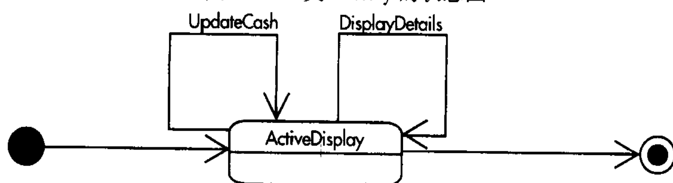


图 10-15　类 Display 的状态图

10.5　分析文档——系统的静态特性

本节的文档提供系统的静态方面的信息，也就是说，下面的文档表明了类是如何与特定的刺激交互、如何做出反应的，而不表明系统是如何随时间而变化的。这些文档有如下两类：

- 类关系图
- 协作图，实例层次和详述层次

10.5.1　类关系图

整个应用程序本身和每个 use case 都有一个类关系图。整个应用程序的类关系图如图 10-16

所示。

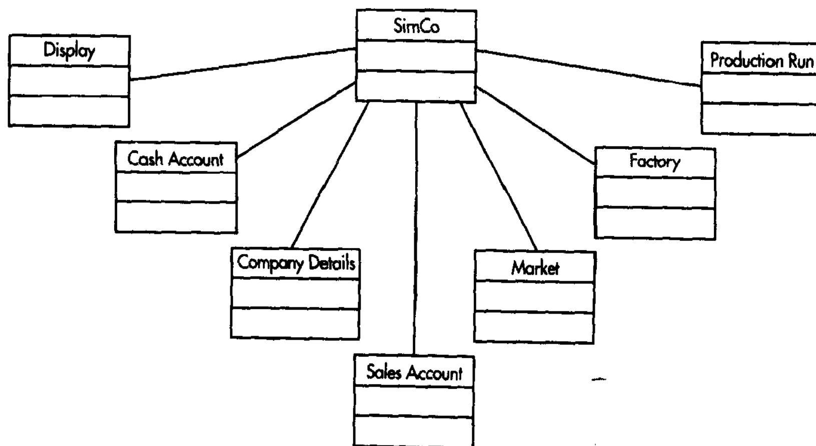


图 10-16 整个应用程序的类关系图

Use case #1——贷款申请的类关系图如图 10-17 所示。

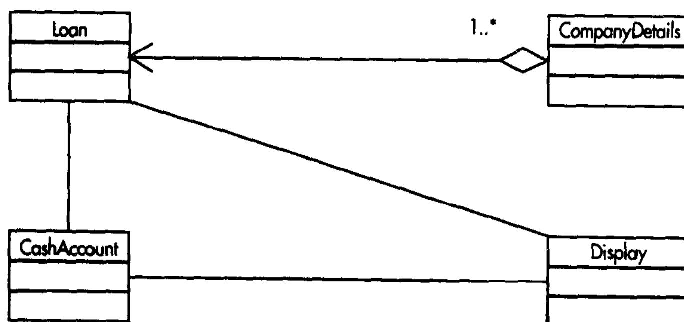


图 10-17 Use case #1——贷款申请的类关系图

Use case #2——购置机器设备的类关系图如图 10-18 所示。

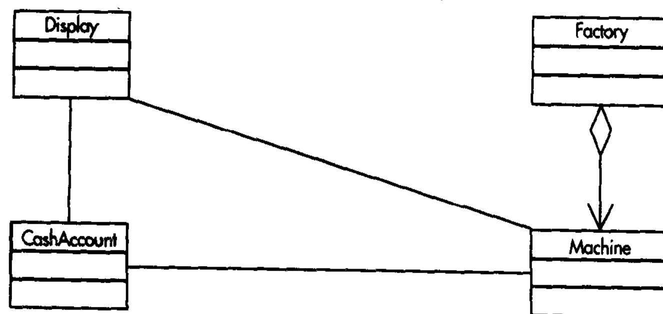


图 10-18 Use case #2——购置机器设备的类关系图

Use case #3——生产运营的类关系图如图 10-19 所示。

Use Case #4——处理公司账务的类关系图如图 10-20 所示。

Use Case #5——显示公司详细信息的类关系图如图 10-21 所示。

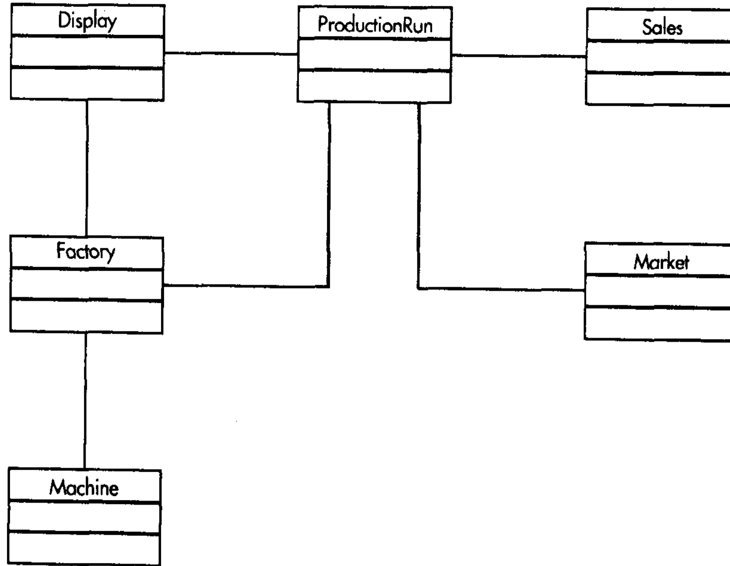


图 10-19　Use case #3——生产运营的类关系图

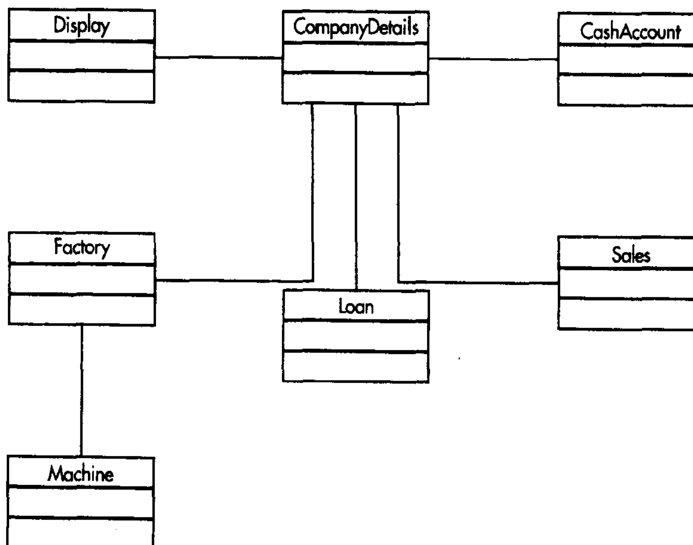


图 10-20　Use case #4——处理公司账务的类关系图

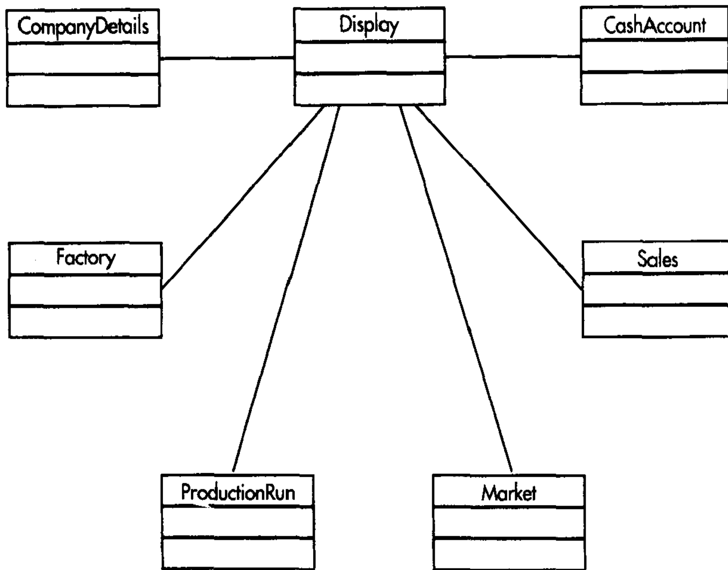


图 10-21 Use case # 5 ——显示公司详细信息的类关系图

10.5.2 协作图

整个应用程序和每个用例均有两个协作图，第一个是实例层次协作图，第二个是详述层次协作图。

整个应用程序的实例层次协作图如图 10-22 所示。

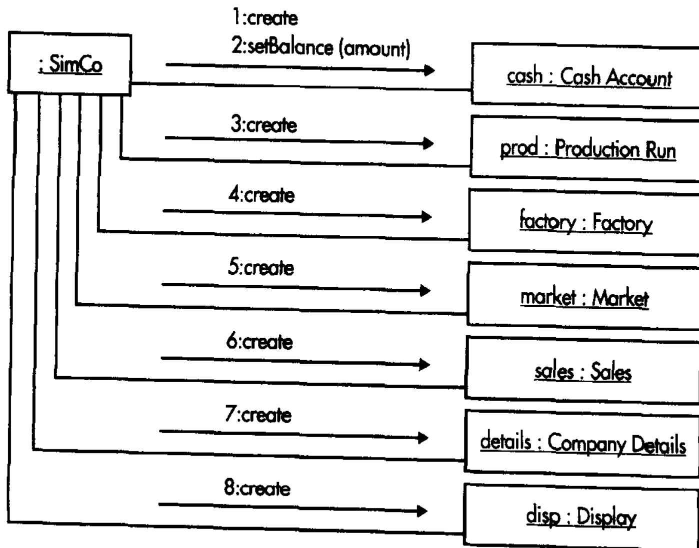


图 10-22 整个应用程序的实例层次协作图

整个应用程序的详述层次协作图如图 10-23 所示。

Use case # 1 ——贷款申请的实例层次协作图如图 10-24 所示。

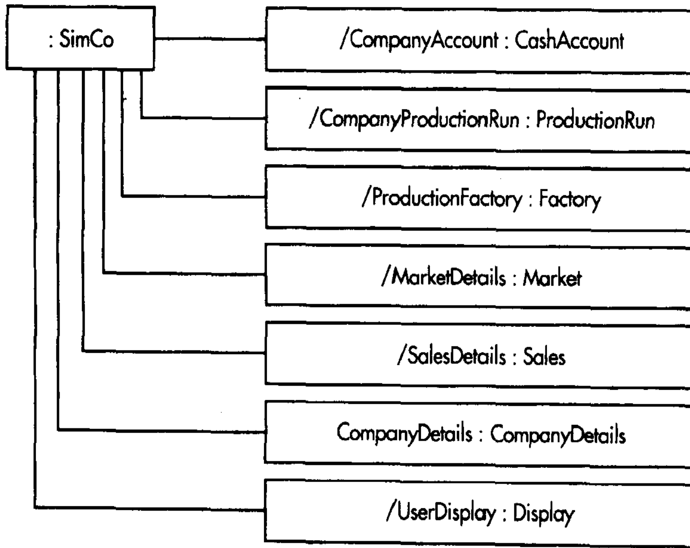


图 10-23 整个应用程序的详述层次协作图

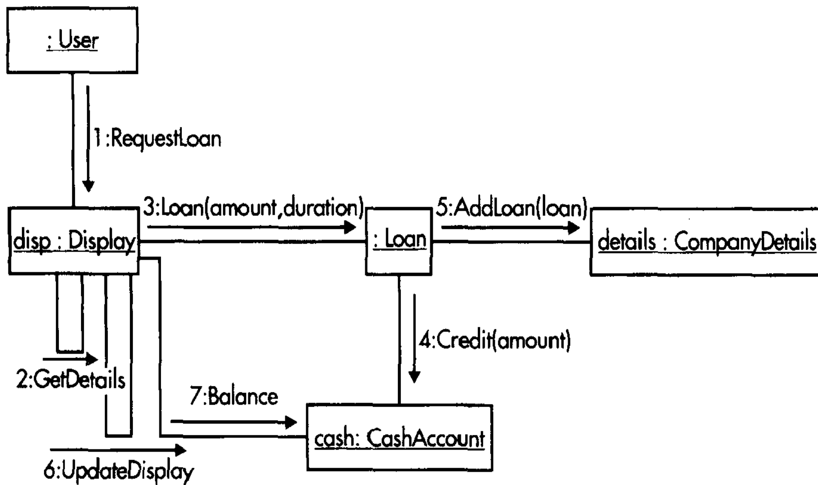


图 10-24 Use case # 1 ——贷款申请的实例层次协作图

Use case #1——贷款申请的详述层次协作图如图 10-25 所示。

Use case #2——购置机器设备的实例层次协作图如图 10-26 所示。

Use case #2——购置机器设备的详述层次协作图如图 10-27 所示。

Use case #3——生产运营的实例层次协作图如图 10-28 所示。

Use case #3——生产运营的详述层次协作图如图 10-29 所示。

Use case #4——处理公司账务的实例层次协作图如图 10-30 所示。

Use case #4——处理公司账务的详述层次协作图如图 10-31 所示。

Use case #5——显示公司详细信息的实例层次协作图如图 10-32 所示。

Use case #5——显示公司详细信息的详述层次协作图如图 10-33 所示。

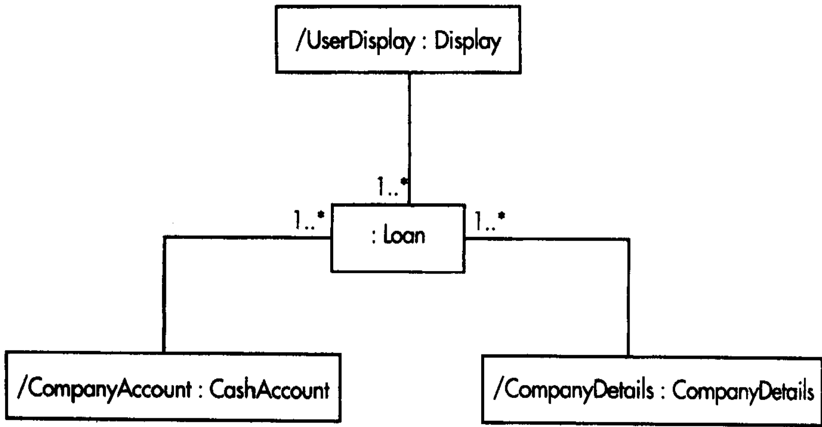


图 10-25 Use case # 1——贷款申请的详述层次协作图

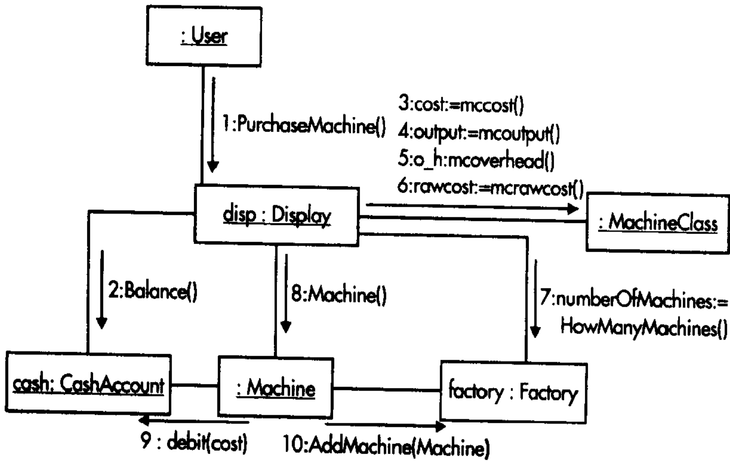


图 10-26 Use case # 2——购置机器设备的实例层次协作图

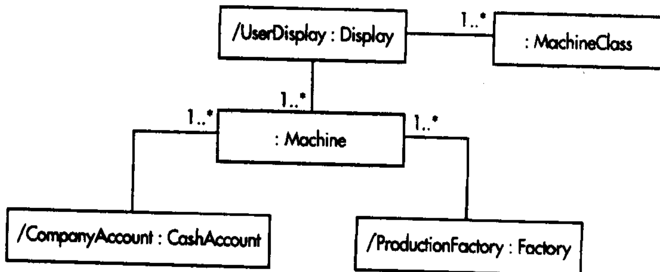


图 10-27 Use case # 2——购置机器设备的详述层次协作图

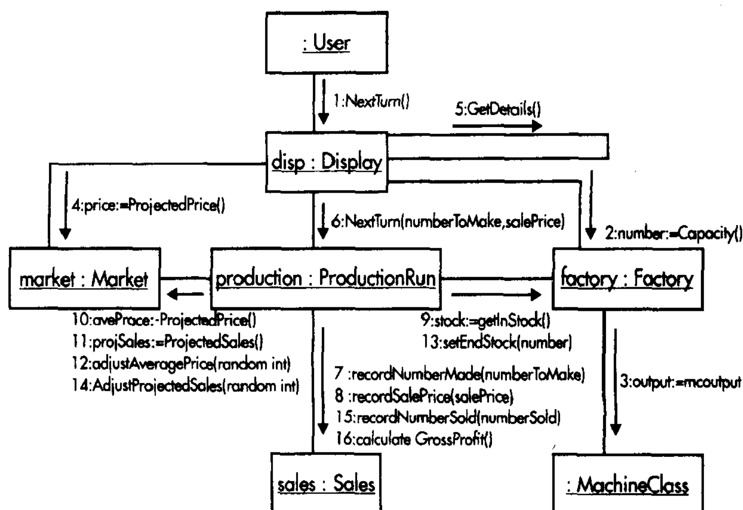


图 10-28 Use case # 3——生产运营的实例层次协作图

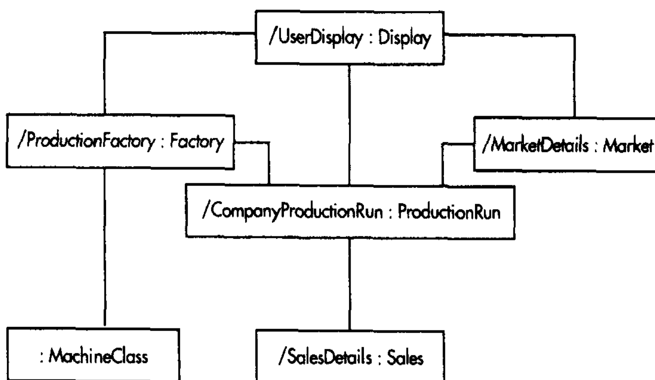


图 10-29 Use case # 3——生产运营的详述层次协作图

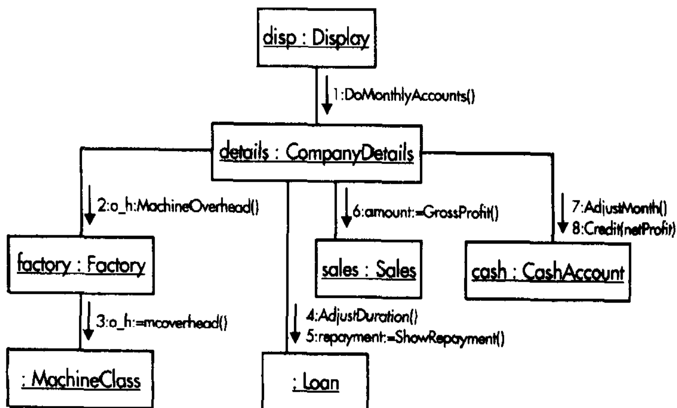


图 10-30 Use case # 4——处理公司账务的实例层次协作图

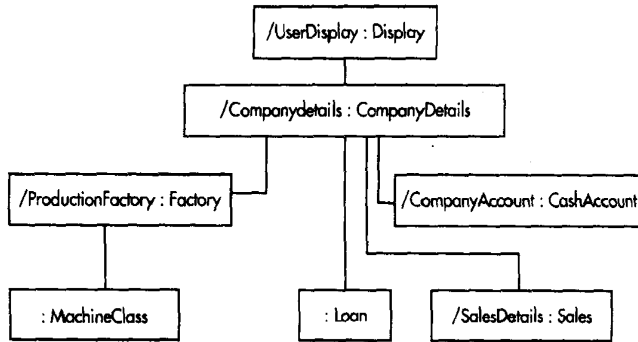


图 10-31　Use case # 4——处理公司账务的详述层次关系图

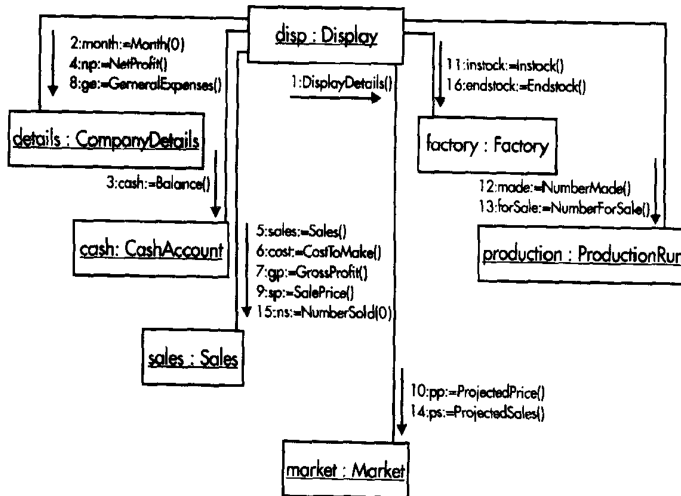


图 10-32　Use case # 5——显示公司详细信息的详述层次协作图

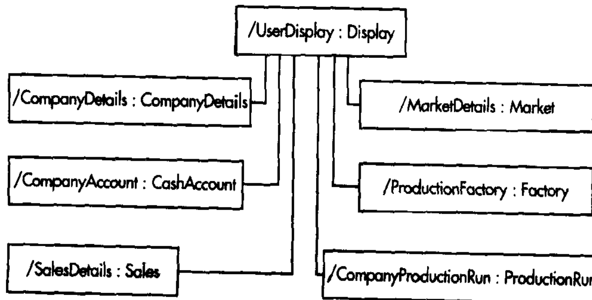


图 10-33　Use case # 5——显示公司详细信息的详述层次协作图

10.6　分析文档——系统的动态特性

本节的文档描述系统随时间迁移而表现出的动作。

活动图：这些图显示用例从开始到结束的发展情况。

序列脚本和序列图：序列脚本和序列图描述在对象之间的交互活动中使用的方法，以及这

些方法被使用的顺序。

10.6.1 活动图

每个用例都有一个活动图，分别见图 10-34 ~ 图 10-38。

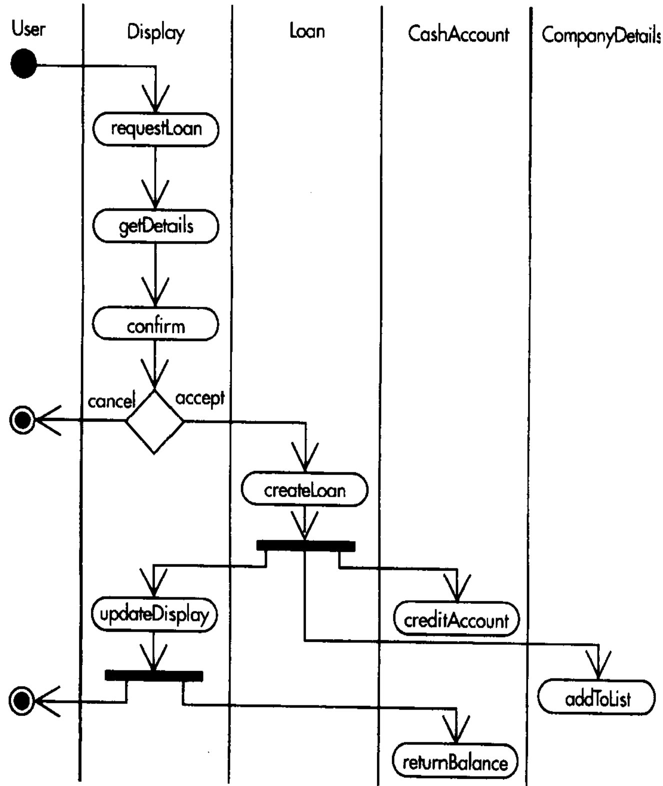


图 10-34 Use case # 1——贷款申请的活动图

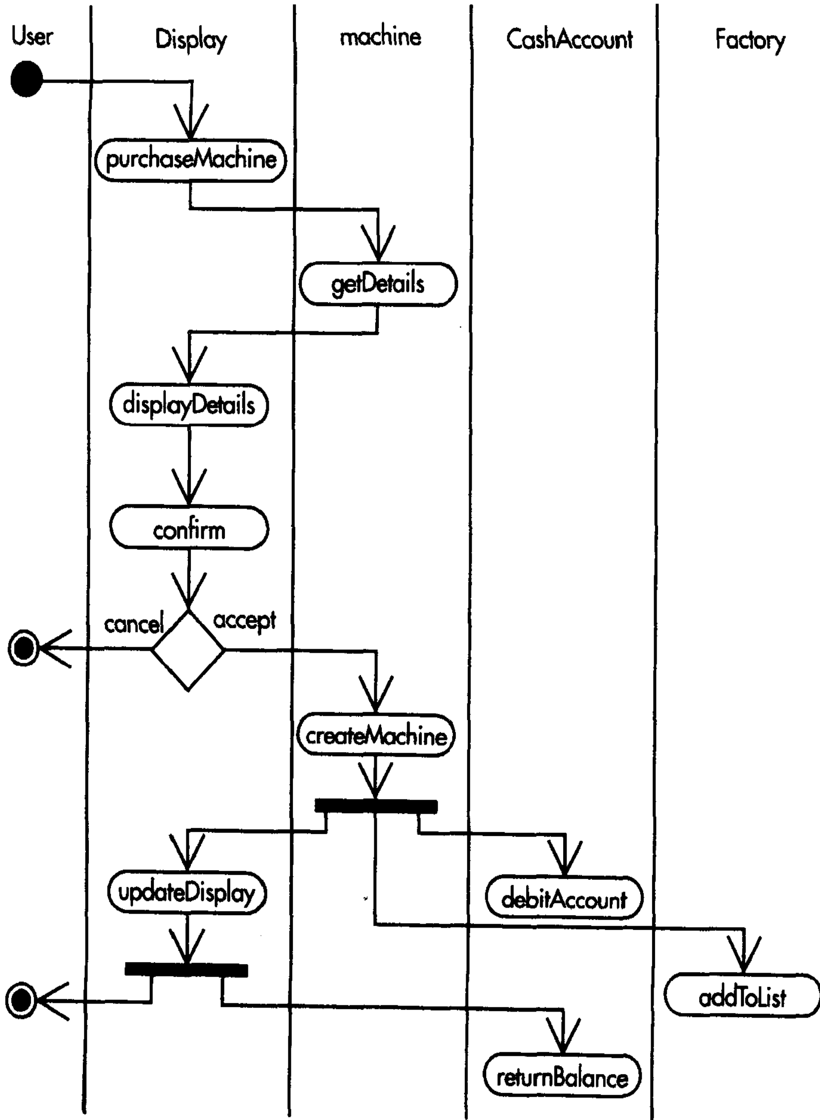


图 10-35 Use case #2——购置机器设备的活动图

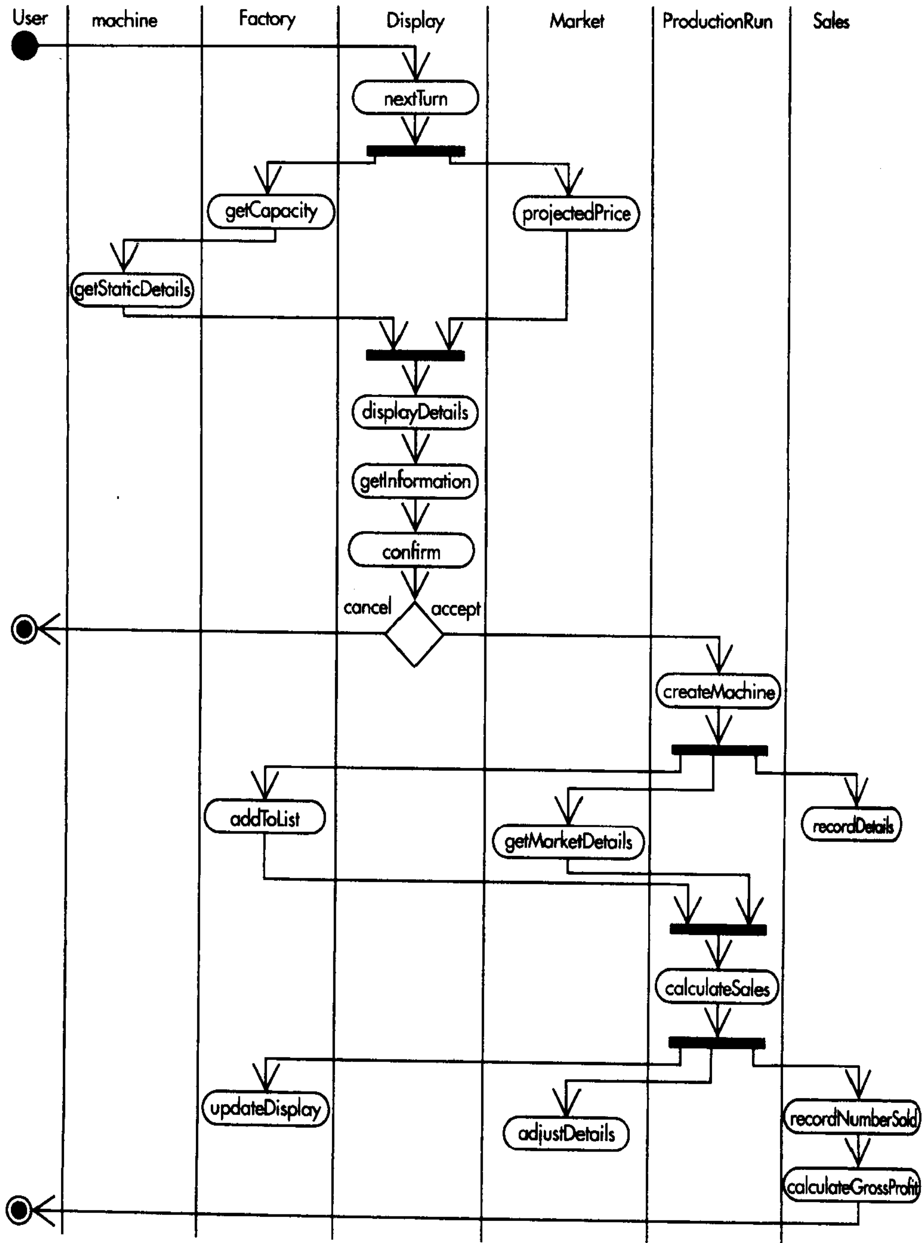


图 10-36　Use case #3——生产运营的活动图

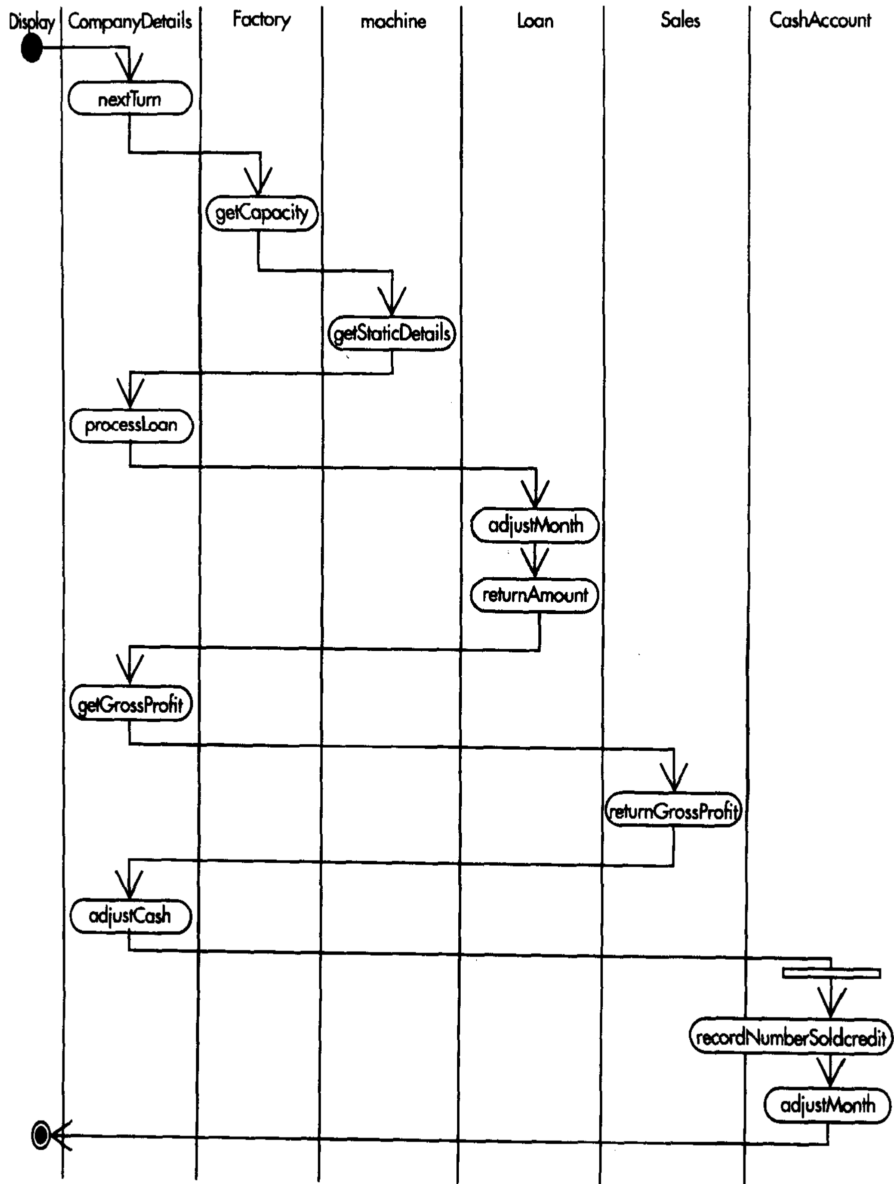


图 10-37 Use case #4——处理公司账务的活动图

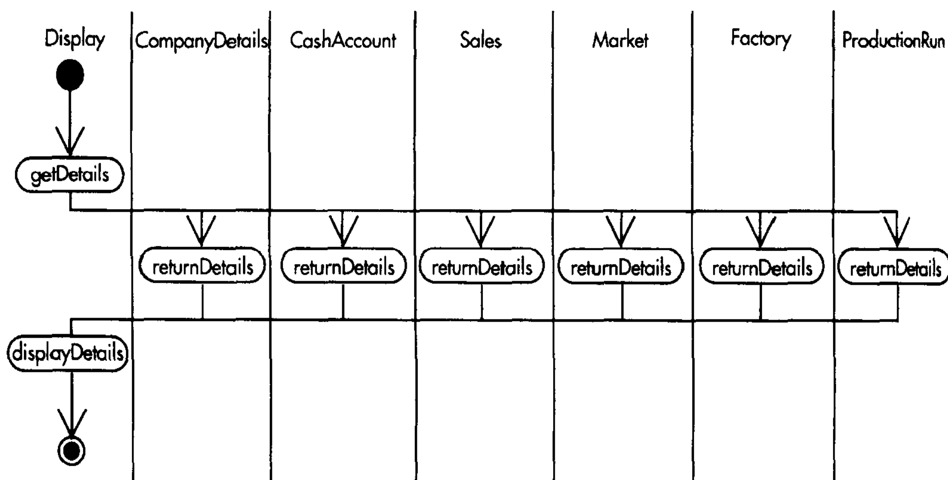


图 10-38　Use case #5——显示公司详细信息的活动图

10.6.2 序列脚本

表10-15～表10-19表示了前面描述的每个用例的脚本。每个脚本将用户开始与系统交互一直到结束的整个过程展示给读者。这些脚本的语法结构请见第2章。这里使用的语法不属于UML v1.4标准的一部分，但是，我发现它不仅十分具有描述性，而且在展示系统运行的细节方面很有用，易于被与项目有关的每一个人理解。

表 10-15　UC #1——贷款申请

Ref #	源	方法	目的	Next Ref #
1	User	requestLoan (void)	Display	1.a
1.a	Display	"Enter amount	User	1.b
1.b	User	<amount < = 0>	Display	1.1
1.b	User	<amount > 0>	Display	1.2
1.1	Display	{end}		
1.2	Display	"Enter duration of loan"	User	1.2.a
1.2.a	User	<duration < = 0>	Display	1.2.1
1.2.a	User	<duration > 0>	Display	1.2.2
1.2.1	Display	{end}		
1.2.2	Display	"Confirm Loan"	User	1.2.2.a
1.2.2.a	User	<Reject>	Display	1.2.2.1
1.2.2.a	User	<Accept>	Display	1.2.2.2
1.2.2.1	Display	{end}		
1.2.2.2	Display	Loan (amount, duration)	Loan	1.2.2.2.a
1.2.2.2.a	Loan	credit (amount)	CashAccount	1.2.2.2.b
1.2.2.2.b	CashAccount	{return void}	Loan	1.2.2.2.c

（续）

Ref #	源	方法	目的	Next Ref #
1.2.2.2.c	Loan	addLoan (self)	CompanyDetails	1.2.2.2.d
1.2.2.2.d	CompanyDetails	{return void}	Loan	1.2.2.2.e
1.2.2.2.e	Loan	{return void}	Display	1.2.2.2.f
1.2.2.2.f	Display	updateCash (void)	Display	1.2.2.2.g
1.2.2.2.g	Display	balance (void)	CashAccount	1.2.2.2.h
1.2.2.2.h	CashAccount	{return double}	Display	1.2.2.2.i
1.2.2.2.i	Display	{end}		

表 10-16 UC #2——购置设备

Ref #	源	方法	目的	Next Ref #
2	User	purchaseMachine (void)	Display	2.a
2.a	Display	balance (void)	CashAccount	2.b
2.b	CashAccount	{return double}	Display	2.c
2.c	Display	Machine:: mccost (void)	Machine	2.d
2.d	Machine	{return double}	Display	2.e
2.e	Display	Machine:: mcoutput (void)	Machine	2.f
2.f	Machine	{return int}	Display	2.g
2.g	Display	Machine:: mcoverhead (void)	Machine	2.h
2.h	Machine	{return double}	Display	2.i
2.i	Display	Machine:: mcrawcost (void)	Machine	2.j
2.j	Machine	{return double}	Display	2.k
2.k	Display	howManyMachines (void)	Factory	2.1
2.1	Factory	{return int}	Display	2.m
2.m	Display	"Confirm Purchase"	User	2.n
2.n	User	< Reject >	Display	2.1
2.n	User	< Accept >	Display	2.2
2.1	Display	{end}		
2.2	Display	Machine (void)	Machine	2.2.a
2.2.a	Machine	debit (cost)	CashAccount	2.2.b
2.2.b	CashAccount	{return void}	Machine	2.2.c
2.2.c	Machine	addMachine (this)	Factory	2.2.d
2.2.d	Factory	{return void}	Machine	2.2.e
2.2.e	Machine	{return void}	Display	2.2.f
2.2.f	Display	updateCash (void)	Display	2.2.g
2.2.g	Display	balance (void)	CashAccount	2.2.h
2.2.h	CashAccount	{return double}	Display	2.2.i
2.2.i	Display	{end}		

表10-17 UC #3——生产运营

Ref #	源	方法	目的	Next Ref #
3	User	nextTurn (void)	Display	3.a
3.a	Display	capacity (void)	Factory	3.b
3.b	Factory	{return int}	Display	3.c
3.c	Display	projectedPrice (void)	Market	3.d
3.d	Market	{return double}	Display	3.e
3.e	Display	"Enter number to make"	User	3.f
3.f	User	< amount < 0 >	Display	3.1
3.f	User	< amount = > 0 >	Display	3.2
3.1	Display	{end}		
3.2	Display	"Enter sale price"	User	3.2.a
3.2.a	User	< price < = 0 >	Display	3.2.1
3.2.a	User	< price > 0 >	Display	3.2.2
3.2.1	Display	{end}		
3.2.2	Display	"Confirm next turn"	User	3.2.2.a
3.2.2.a	User	< Reject >	Display	3.2.2.1
3.2.2.a	User	< Accept >	Display	3.2.2.2
3.2.2.1	Display	{end}		
3.2.2.2	Display	nextTurn (numberToMake, salePrice)	ProductionRun	3.2.2.2.a
3.2.2.2.a	ProductionRun	recordNumberMade (int)	Sales	3.2.2.2.b
3.2.2.2.b	Sales	{return void}	ProductionRun	3.2.2.2.c
3.2.2.2.c	ProductionRun	recordSalePrice (double)	Sales	3.2.2.2.d
3.2.2.2.d	Sales	{return void}	ProductionRun	3.2.2.2.e
3.2.2.2.e	ProductionRun	getInstock (void)	Factory	3.2.2.2.f
3.2.2.2.f	Factory	{return int}	ProductionRun	3.2.2.2.g
3.2.2.2.g	ProductionRun	projectedPrice (void)	Market	3.2.2.2.h
3.2.2.2.h	Market	{return double}	ProductionRun	3.2.2.2.i
3.2.2.2.i	ProductionRun	projectedSales (void)	Market	3.2.2.2.j
3.2.2.2.j	Market	{return int}	ProductionRun	3.2.2.2.k
3.2.2.2.k	ProductionRun	adjustAvePrice (double)	Market	3.2.2.2.1

（续）

Ref #	源	方法	目的	Next Ref #
3.2.2.2.1	Market	{return void}	ProductionRun	3.2.2.2.m
3.2.2.2.m	ProductionRun	setEndStock (int)	Factory	3.2.2.2.n
3.2.2.2.n	Factory	{return void}	ProductionRun	3.2.2.2.o
3.2.2.2.o	ProductionRun	adjustProjSales (double)	Market	3.2.2.2.p
3.2.2.2.p	Market	{return void}	ProductionRun	3.2.2.2.q
3.2.2.2.q	ProductionRun	recordNumberSold (int)	Sales	3.2.2.2.r
3.2.2.2.r	Sales	{return void}	ProductionRun	3.2.2.2.s
3.2.2.2.s	ProductionRun	calcGrossProfit (void)	Sales	3.2.2.2.t
3.2.2.2.t	Sales	{return void}	ProductionRun	3.2.2.2.u
3.2.2.2.u	ProductionRun	{return void}	Display	3.2.2.2.v
3.2.2.2.v	Display	{end}		

表 10-18　UC #4——处理公司账务

Ref #	源	方法	目的	Next Ref #
4	Display	doMonthlyAccounts (void)	CompanyDetails	4.a
4.a	CompanyDetails	machineOverhead (void)	Factory	4.b
4.b	Factory	{return double}	CompanyDetails	4.c
4.c	CompanyDetails	adjustDuration (void)	Loan	4.d
4.d	Loan	{return void}	CompanyDetails	4.e
4.e	CompanyDetails	showRepayment (void)	Loan	4.f
4.f	Loan	{return void}	CompanyDetails	4.g
4.g	CompanyDetails	grossProfit (int)	Sales	4.h
4.h	Sales	{return double}	CompanyDetails	4.i
4.i	CompanyDetails	adjustMonth (void)	CashAccount	4.j
4.j	CashAccount	{return void}	CompanyDetails	4.k
4.k	CompanyDetails	credit (netProfit)	CashAccount	4.1
4.1	CashAccount	{return void}	CompanyDetails	4.m
4.m	CompanyDetails	{return void}	Display	4.n

表 10-19　UC #5——显示公司详细信息

Ref #	源	方法	目的	Next Ref #
5	User	displayDetails (void)	Display	5.a
5.a	Display	month (void)	CompanyDetails	5.b
5.b	CompanyDetails	{return int}	Display	5.c
5.c	Display	balance (void)	CashAccount	5.d
5.d	CashAccount	{return double}	Display	5.e
5.e	Display	netProfit (void)	CompanyDetails	5.f
5.f	CompanyDetails	{return double}	Display	5.g
5.g	Display	sales (void)	Sales	5.h
5.h	Sales	{return double}	Display	5.i
5.i	Display	costToMake (void)	Sales	5.j
5.j	Sales	{return double}	Display	5.k
5.k	Display	grossProfit (void)	Sales	5.1
5.1	Sales	{return double}	Display	5.m
5.m	Display	generalExpenses (void)	CompanyDetails	5.n
5.n	CompanyDetails	{return double}	Display	5.o
5.o	Display	salePrice (void)	Sales	5.p
5.p	Market	{return double}	Display	5.q
5.q	Display	projectedPrice (void)	Market	5.r
5.r	Sales	{return double}	Display	5.s
5.s	Display	showInstock (void)	Factory	5.t
5.t	Factory	{return int}	Display	5.u
5.u	Display	numberMade (void)	ProductionRun	5.v
5.v	ProductionRun	{return int}	Display	5.w
5.w	Display	numberForSale (void)	ProductionRun	5.x
5.x	ProductionRun	{return int}	Display	5.y
5.y	Display	projectedSales (void)	Market	5.z
5.z	Market	{return int}	Display	5.aa
5.aa	Display	numberSold (void)	Sales	5.bb
5.bb	Sales	{return double}	Display	5.cc
5.cc	Display	showEndStock (void)	Factory	5.dd
5.dd	Factory	{return int}	Display	5.ee
5.ee	Display	{end}		

10.6.3　序列图

每个用例有一个序列图，见图 10-39 ~ 图 10 – 43。

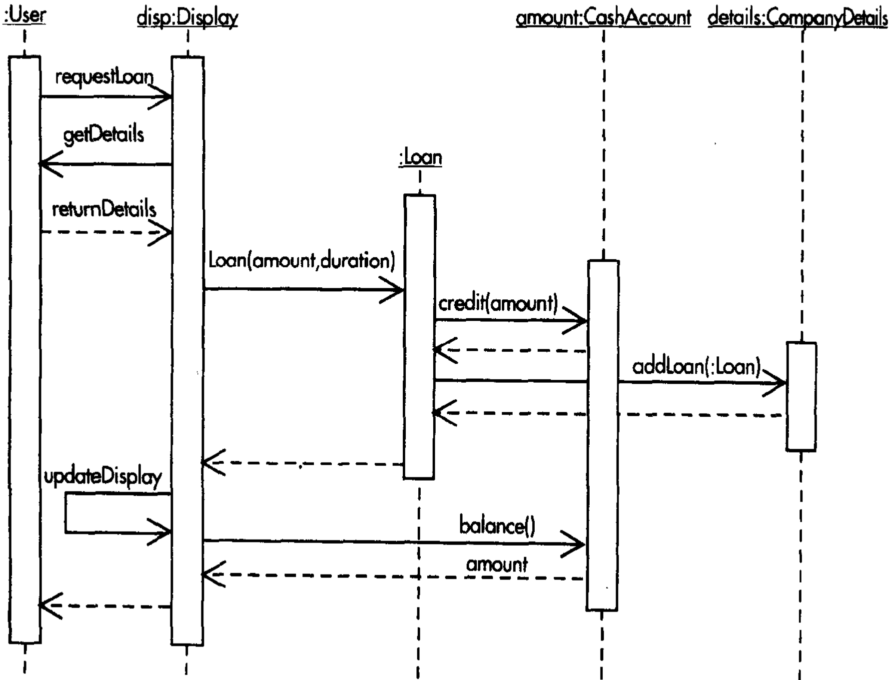


图 10-39　Use case #1——贷款申请的序列图

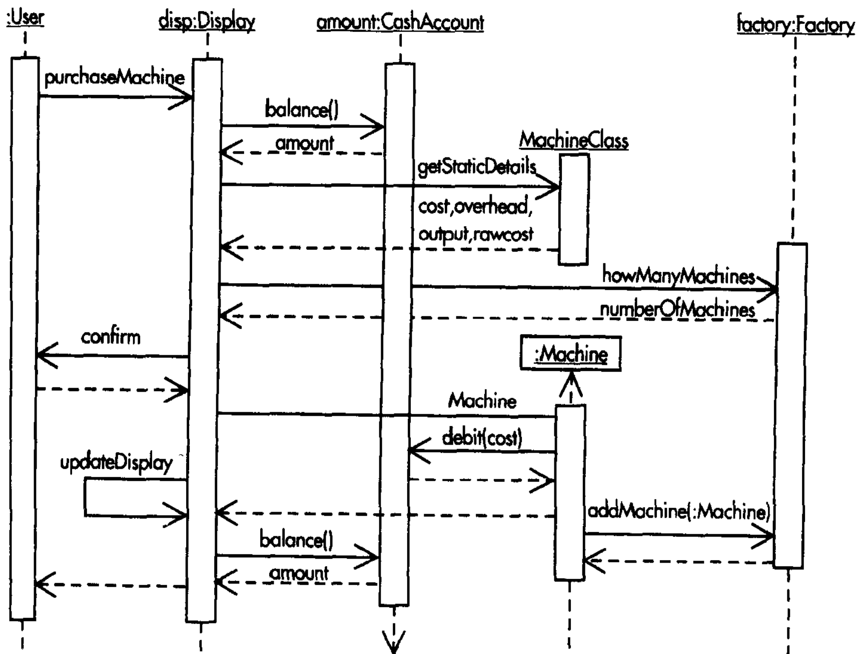


图 10-40　Use case #2——购置设备的序列图

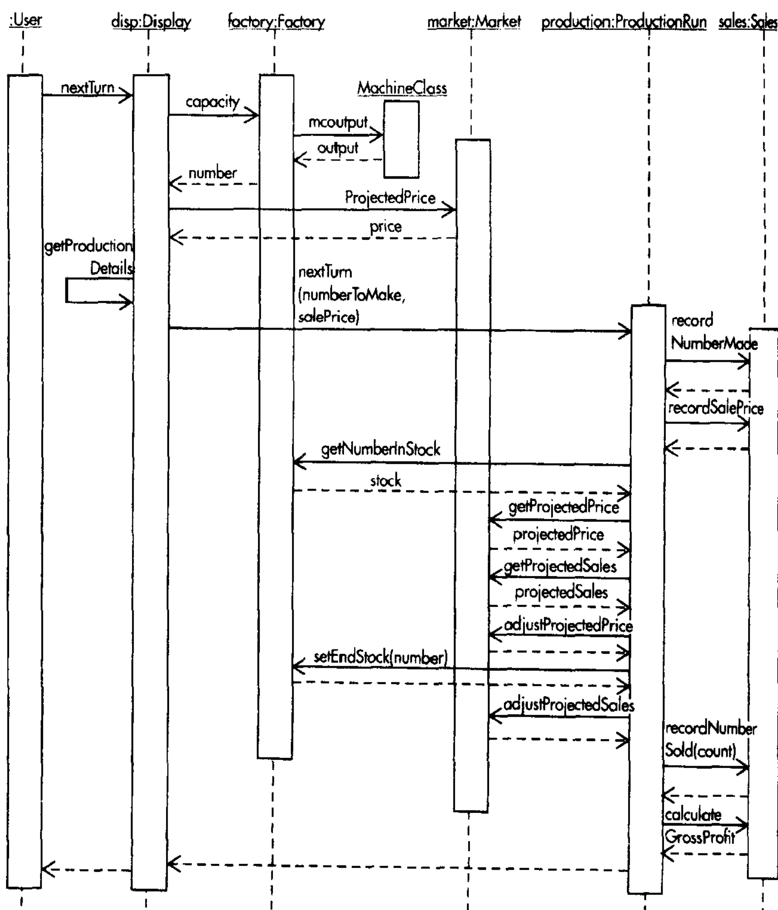


图 10-41　Use case #3——生产运营的序列图

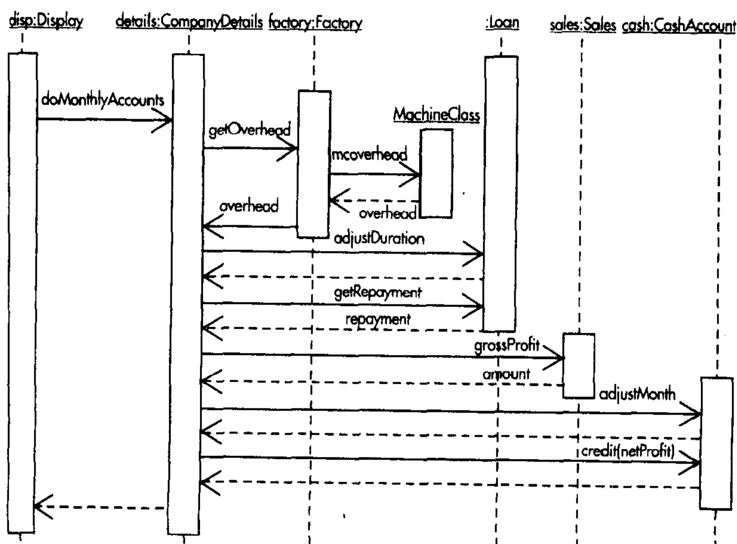


图 10-42　Use case #4——处理公司账务的序列图

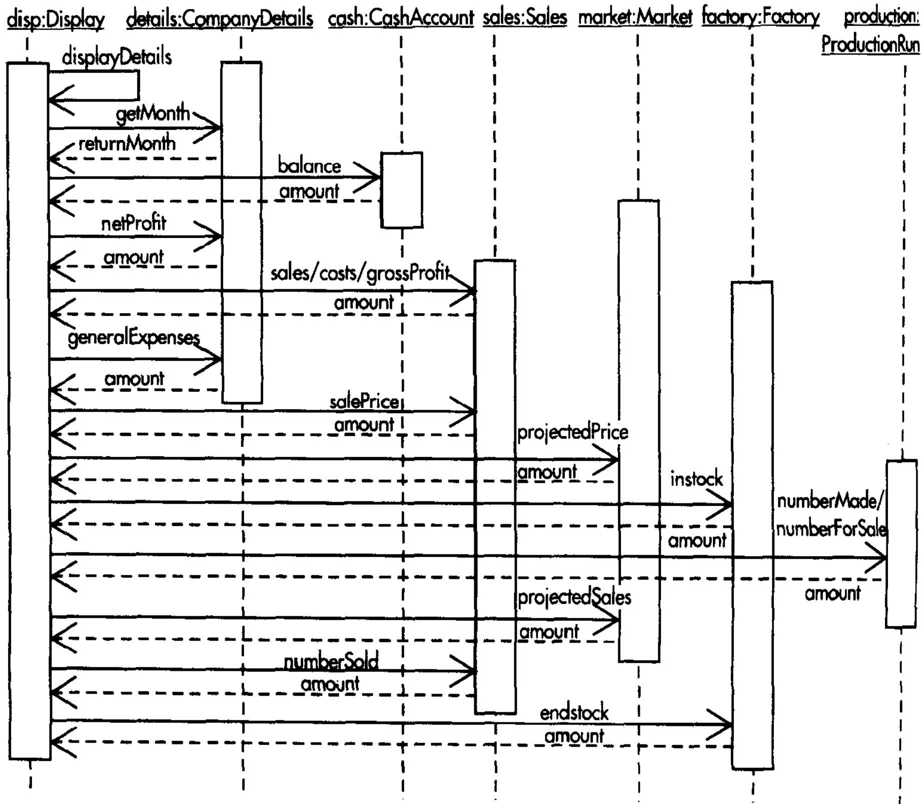


图 10-43　Use case #5——显示公司详细信息的序列图

10.7 小结

本章带你身临其境地“游历”了一遍应用程序分析阶段使用的文档。每一种文档都是在第2章中引入的，而在本章体现在一个完整的实例中。本章还说明了在分析阶段的一个领域中发现的信息如何使用在另一个领域中。例如，静态文档描述类之间的交互接口，而动态文档中则表明了这些接口的实际使用方式。本章中使用的每一个文档，表面上看是涉及同样的信息，但却是以不同的方式，因此，提供不同视角。所以，我建议在分析阶段使用所有的文档类型，以保证提供完整的覆盖范围和充分的可理解性。

第 11 章　实例学习 2——开发一个多线程机场管理模拟程序

本章将讨论以下内容：

▶ 学习如何开发一个多线程应用程序

▶ 理解与线程间交互有关的问题

通过本章的机场管理模拟程序的个案分析和学习，可以使我们了解线程模型应用程序的开发。本章使用线程技术，将一个小的、每天一架飞机的机场应用软件扩展为一个有实际意义的东西。最初程序中，机场只有一条跑道、一个停机位和一个入口，而最终的应用程序模型将允许很多飞机起降，同时避免撞机或死锁。

11.1　一次只有一架飞机

本部分详细说明目前机场是怎样管理使用其设施的飞机的。

▶ 降落过程：飞机使用跑道，降落，然后滑行到停机位，如图 11-1 所示。

▶ 起飞过程：飞机滑行到跑道，然后起飞，见图 11-2。

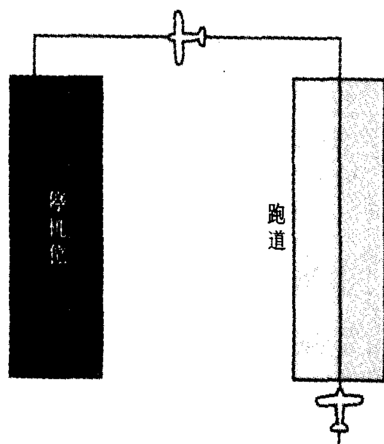


图 11-1　降落过程

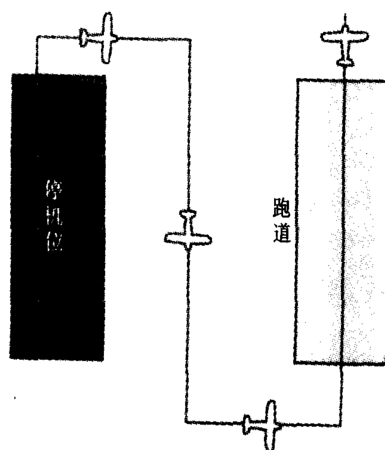


图 11-2　起飞过程

Java 代码

以下是用于模拟机场管理的 Java 应用程序的基本概貌。

1. Airport.java

```
/*
**  Name: Airport.java
**  This is the main class for the airport simulation.
**  It creates one thread to simulate a single aircraft usage of
**  the airport.
*/
```

几乎所有编写的Java程序都包含这些基本的引用（Import）文件：

```
import java.awt.*;
import java.io.*;
import java.lang.*;
import java.util.*;
```

类 **Airport** 是机场模拟应用程序的主类：

```
public class Airport
{
```

机场图（airport map）是机场的图形化表示，它显示跑道、停机位，以及在机场中不断移动的飞机：

```
public AirportMap          airportMap;
```

threadGroup用于管理线程，可以将线程分组管理，也可以管理单个独立的线程。下面的语句声明了threadGroup。虽然在本例中，threadGroup不是必需的，但为了以后的版本扩展的余地，还是将它包括进来了。

```
protected ThreadGroup     threadGroup;

public Airport()
{
```

以下的代码创建并显示AirportMap。AirportMap是类 **Canvas** 的扩展，因此需要一个框架(Frame)，以便在其中显示。AirportMap的代码将在Airport的代码之后给出：

```
Frame         mapFrame;
Panel         panel;

mapFrame = new Frame ("Airport Map");
airportMap = new AirportMap ();
panel = new Panel ();
panel.setLayout (new BorderLayout ());
panel.add ("Center", airportMap);

mapFrame.add ("Center", panel);
mapFrame.setSize (100, 150);
mapFrame.show ();
```

接下来，我们需要初始化threadGroup：

```
threadGroup = new ThreadGroup("Airports");
```

尽管目前不必有一个表示飞机的数组，但以后有用：

```
Plane planes [] = new Plane [1];
```

每一个线程都有一个 *run* 方法。这个 *run* 方法可由线程的创建者调用，或者，就像本例，每个飞机线程本身将激活 *run* 方法：

```
planes [0] = new Plane (threadGroup, 0, this);
```

机场应用程序使用线程构造 'join' 等待代表飞机的线程，以保证只要有一架飞机存在，应用程序就不会退出：

```
        try
        {
            planes [0].join();
        }
        catch (InterruptedException e) {}

        System.exit (0);
    }

    /*
    ** The main method of the application.
    ** Start the Airport
    */
    public static void main(String[] args)
    {
        Airport a = new Airport ();
    }
}
```

2. AirportMap.java

```
/*
**  Name: AirportMap.java
*/
import java.awt.*;
public class AirportMap extends Canvas
{
    public AirportMap ()
    {
    /*
    ** Create the airport map
    */
    }
```

方法名称： LandPlane

该方法用于表示来机场降落并滑行到停机位入口的飞机。

入口参数：指向飞机的一个句柄

输出：无

```
public void LandPlane (Plane plane)
{
    /*
    ** Display the aircraft flying north as it uses the runway to
    ** land. Finally show the aircraft moving west and then south
    ** towards the terminal
    */
}
```

方法名称：Takeoff

该方法用于表示从停机位入口滑行到跑道然后起飞的飞机。

入口参数：指向飞机的一个句柄

输出：无

```
    public void Takeoff (Plane plane)
    {
        /*
        ** Display the aircraft moving away from the terminal, down
        ** the map as it taxis to the start of the runway, then
        ** finally north as it flies away from the airport
        */
    }
}
```

3. Plane.java

```
/*
**  Name: Plane.java
**  This class is the aircraft, it dictates how the aircraft will
**  approach and use the airport
*/
import java.awt.*;
import java.io.*;
import java.lang.*;
import java.util.*;
public class Plane extends Thread
{
```

下面这行代码是应用程序用于标识线程的线程名称：

```
String          threadName;
```

下面的变量指向 Airport 实例本身的一个引用，可用于获取应用程序中的其他参量或部件：

```
Airport         airport;
```

我们使用下面的变量来表示飞机在停机位入口处停留的时间：

```
static int              timeAtGate;
```

接下来，我们声明用于在 AirportMap 中显示飞机的变量。

下面是类 Plane（飞机）的主构造函数，它设置线程名称，并创建飞机的图像。它还启动线程的运行。

```
public Plane
    (ThreadGroup threadGroup, int name, Airport theAirport)
{
    /*
    ** Create our server thread with a name.
    */
    super(threadGroup, "Plane-" + name);

    /*
    ** Save the reference to the Airport instance
    */
    airport = theAirport;

    /*
    ** Set the time at the terminal for this aircraft,
    ** to be 3 seconds
    */
    timeAtGate = 3000;

    /*
    ** Create the images and draw the aircraft
    */
    createImages ();

    /*
    ** name the thread
    */
    threadName = new String ("Plane-" + name);

    /*
    ** Start the thread
    */
    this.start();
}

public void createImages ()
{
    /*
    ** Create the polygon objects and draw the aircraft
    */
```

下面的插图是表示飞机对象的多边形：

```
}

/*
**  Run
*/
    public void run()
    {
        System.out.println ("Waiting to land " + threadName);
        airport.airportMap.LandPlane (this);

        System.out.println ("At gate " + threadName);
        /* Spend some time at the gate */
        try
        {
            sleep (timeAtGate);
        }
        catch (InterruptedException e) {}

        System.out.println ("Waiting to takeoff " + threadName);
        airport.airportMap.Takeoff (this);

        System.out.println ("END " + threadName);
    }
}
```

11.2 一个停机位入口同时有两架飞机

现在，将机场管理模型改变一下：有两架飞机试图在同一时间使用该机场，当然，机场还是只有一个停机位。

11.2.1 降落过程

两架飞机在机场中的降落过程描述如下：

- **第一架飞机的降落**：与上一节中的描述类似，飞机使用跑道，降落，然后滑行到停机位。
- **第二架飞机的降落**：第二架飞机与第一架飞机的降落过程完全一致。

1. 问题 #1——新闻简报："两架飞机在停机位相撞"

问题在于第二架飞机与第一架飞机的操作过程完全一致，即使假定停机位是空闲的，它们也可能在滑行时盲目相撞（见图 11-3）。那么，如何避免撞机事故呢？

2. 解决方法 #1

停机位必须提供一个排斥锁（mutex），一个在同一时间内只能被一架飞机持有的锁。这样，当第二架飞机试图获取排斥锁时，它将被告知：停机位中已经有一架飞机，它必须在新建的紧靠跑道的降落区内等待。图 11-4 中，排斥锁显示为一把挂锁，而降落区显示为一个内有斜十字的圆：

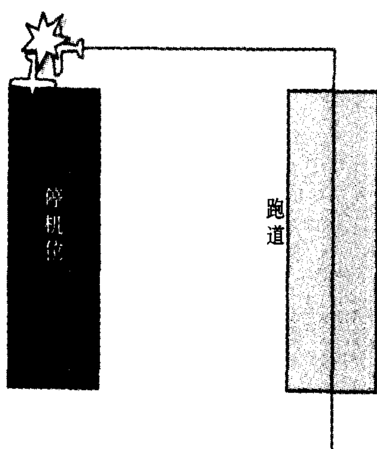


图 11-3 两架飞机在停机位相撞

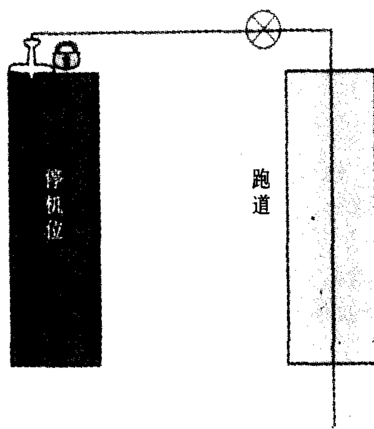


图 11-4 停机位提供一个排斥锁

3. 修改后的降落过程

任何飞机都使用跑道，降落，然后移动到降落区。一旦进入降落区，飞机就请求停机位的排斥锁。如果排斥锁是空闲的，飞机就滑行到停机位。如果排斥锁不空闲，它就等待排斥锁变为空闲。

11.2.2 起飞过程

两架飞机从机场起飞的过程描述如下：

- **第一架飞机的起飞**：与上一节中的描述类似，飞机滑行到跑道，然后起飞。
- **第二架飞机的起飞**：第二架飞机与第一架飞机的起飞过程完全一致。

1. 问题#2——新闻简报："两架飞机在跑道相撞"

当第一架飞机正准备起飞时，如果第二架飞机不在地面上，就出现问题了。如第一架飞机正准备起飞时，第二架飞机却正准备降落。那么，如何避免撞机事故呢？见图 11-5。

2. 解决方法#2

跑道必须提供一个排斥锁（见图 11-6），一个在同一时间内只能被一架飞机持有的锁。这样，当任何一架飞机试图获取排斥锁时，它将被告知：已经有一架飞机正使用跑道，它必须等待，见图 11-6。

3. 修改后的起飞过程

飞机请求使用跑道，若被允许使用，飞机就起飞。

4. 问题#3——新闻简报："机场无动静"

虽然使用排斥锁可以避免撞机事故的发生，但如果飞机不释放排斥锁，就会引起死锁（见图 11-7）。那么，如何避免死锁的发生呢？

5. 解决方法#3

当飞机确实不再需要跑道排斥锁时，确保释放跑道排斥锁（见图 11-8）。另外，当确信飞机使用完停机位排斥锁后，确保释放停机位排斥锁。

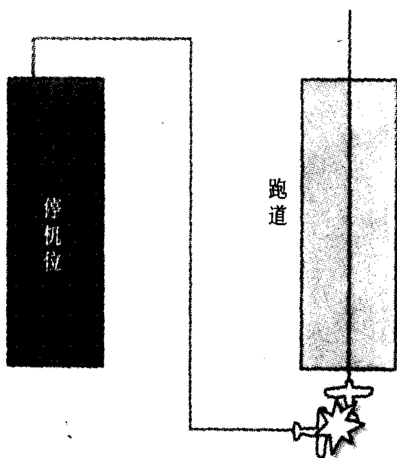


图 11-5 两架飞机在跑道相撞

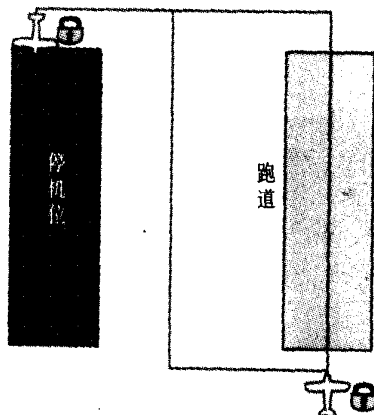


图 11-6 跑道也提供排斥锁

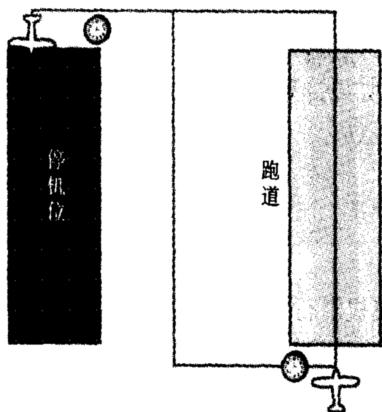


图 11-7 陷于停顿的机场

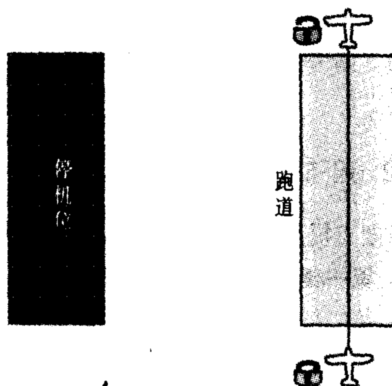


图 11-8 离开跑道的飞机释放排斥锁

11.2.3 修改后的降落/起飞过程

下面列出修改后的降落和起飞过程。

1. 降落过程

下面列出了飞机安全降落必须遵守的顺序步骤。

(1) 飞机请求跑道排斥锁。

(2) 飞机获得跑道排斥锁。

(3) 飞机降落。

(4) 飞机移动到降落区。

(5) 飞机释放跑道排斥锁。

(6) 飞机请求停机位排斥锁。

(7) 飞机获得停机位排斥锁。

(8) 飞机滑行到停机位。

2. 起飞过程

下面列出了飞机安全起飞必须遵守的顺序步骤。

(1) 飞机请求跑道排斥锁。

(2) 飞机获得跑道排斥锁。

(3) 飞机滑行到跑道。

(4) 飞机起飞。

(5) 飞机释放停机位排斥锁。

(6) 飞机释放跑道排斥锁。

11.2.4　修改后的 Java 代码

新的代码反映了飞机起降时需要支持的解决方法 #1、#2 和 #3。

1. Airport.java

```
public class Airport
{
    < code as before >
```

为了解决目前遇到的三个问题，需要声明排斥锁变量。本例中，排斥锁是通过向量（Vector）链表的形式实现的。这种方法允许每个线程等待排斥锁变为空闲。当一个线程释放排斥锁时，它从链表的表头去掉自身，并通知链表中的下一个线程（目前已成为链表中的第一个元素），它现在已经得到排斥锁了。链表中的第一个元素为当前活动的线程，而所有其他线程为等待的线程。

```
/* Solution #2 - runway mutex */
private static Vector             runwayList = new Vector();
/* Solution #1 - terminal gate mutex */
private static Vector             gatesList = new Vector ();

public Airport()
{
    /*
    ** Create and display the AirportMap
    */

    < code as before >

    /*
    ** In this example the array is constructed to hold two
    ** threads
    */
    Plane planes [] = new Plane [2];
    /*
    ** Create the threads
```

```
        */
        for (int j = 0; j < 2; j++)
        {
            planes [j] = new Plane (threadGroup, j, this);
```

与实际情况一样，并非所有的飞机都在同一时刻到达机场。下面的这个延迟在相邻的飞机之间增加 5 秒钟的时间间隔。

```
        synchronized (this)
        {
            try
            {
                wait (5000);
            }
            catch (InterruptedException e) {}
        }
    }
```

正如以前提到的，为了避免飞机线程一开始运行，应用程序就退出，强迫它在此等待线程运行完毕。

```
        try
        {
            for (int j = 0; j < 2; j++)
            {
                planes [j].join();
            }
        }
        catch (InterruptedException e) {}

        System.exit (0);
    }

    /* Start the Airport */
    public static void main(String[] args)
    {
        Airport a = new Airport ();
    }
```

这些方法提供了对问题 # 1 和问题 # 2 的解决方法，它们使用跑道排斥锁和停机位入口排斥锁。它们请求排斥锁，如果请求失败，就引起线程阻塞或挂起。如果线程得到排斥锁，则设置锁的数值。当线程释放锁时，则重置锁的数值。可以提供一个通用的方法，它将线程和锁作为变量。

```
    public void RequestRunwayLock (Plane plane)
    {
        RequestLock (plane, runwayList);
    }
```

```
public void RequestGatesLock (Plane plane)
{
    RequestLock (plane, gatesList);
}
```

通用的方法 *RequestLock* 将调用的线程作为变量，以便能够得知哪一个线程在请求排斥锁，还将请求的排斥锁的类型作为变量。

对参数 lockList 进行同步，以保证只有一个线程可以访问它。设置一个标志，表明链表是否为空，然后将当前的飞机线程加入链表。如果链表中含有一个元素，则表明锁正在被使用。如果链表中含有一个以上的元素，则表明锁正在被使用，而且还有其他线程正在等待使用该锁。如果链表为空，则表明排斥锁是空闲的，因为它没被使用，而且没有其他线程在等待使用它。

```
private void RequestLock (Plane plane, Vector lockList)
{
    Boolean   empty;
    synchronized (lockList)
    {
        empty = lockList.isEmpty ();
        lockList.addElement (plane);
    }

    /*
    ** if the list is not empty, emulate C++ and wait
    */
    if (empty == false)
    {
        try {plane.wait ();}
        catch (InterruptedException e) {}
    }
    /*
    ** This method will exit if the list is empty or this
    ** thread has been woken up (unblocked)
    */
}
```

当当前活动线程不再需要排斥锁时，这些方法被调用。它们调用一个通用的方法，该方法将线程和排斥锁作为变量。

```
public void ReleaseRunwayLock (Plane plane)
{
    ReleaseLock (plane, runwayList);
}
```

```
public void ReleaseGatesLock (Plane plane)
{
    ReleaseLock (plane, gatesList);
}
```

通用的方法 *ReleaseLock* 将调用的线程作为变量，以便能够得知哪一个线程将释放排斥锁，还将释放的排斥锁作为变量。

对参数 lockList 进行同步，以保证只有一个线程可以访问它。从链表中清除当前的飞机线程，以保证链表中只包含等待使用锁的线程。如果链表为空，则返回。如果链表不为空，检查第一个元素，并通知它现在已经得到排斥锁了。

```
private void ReleaseLock (Plane plane, Vector lockList)
{
    /*
    ** This variable is assigned to the extracted list element.
    */
    Plane    peek;
    synchronized (lockList)
    {
        /*
        ** remove current aircraft thread from the list
        */
        lockList.removeElement (plane);
        /*
        ** If the list is not empty it means that there is
        ** another waiting to use this lock
        */
        if (lockList.isEmpty () == false)
        {
            try
            {
                /*
                ** Get the first element on the list
                */
                peek = (Plane)lockList.firstElement ();
                /*
                ** synchronize the thread, so that nothing else
                ** is trying to use it
                */
                synchronized (peek)
                {
                    /* Wake up the thread */
                    peek.notify ();
                }
```

```
                }
                catch (NoSuchElementException e) {}
            }
        }
    }

}
```

2. AirportMap.java

```
public class AirportMap extends Canvas
{
    < code as before >
```

下面是一个新的 *LandPlane* 方法，因为如果停机位忙时，降落的飞机需要滑行到降落区。

```
public void LandPlane (Plane plane)
{
    /*
    ** Display the aircraft flying north as it uses the runway
    ** to land, then moves slightly to the west to the landing
    ** area
    */
}
```

下面是一个新的方法：*TaxiToGate*。它将飞机从降落区滑行到停机位入口。

```
public void TaxiToGate (Plane plane)
{
    /*
    ** Display the aircraft moving west from the landing area and
    ** south towards the terminal gate
    */
}
public void Takeoff (Plane plane)
{
    /*
    ** Display the aircraft moving away from the terminal, down
    ** the map as it taxis to the start of the runway, then
    ** finally north as it flies away from the airport
    */
}
}
```

3. Plane.java

```
public class Plane extends Thread
{
        < code as before >
/*
```

```
**  Run
*/
    public void run()
    {
        System.out.println ("Waiting to land " + threadName);
```

降落过程（1，2）——请求并获取跑道排斥锁：

```
/* Solution #2 - runway mutex */
airport.RequestRunwayLock (this);
System.out.println ("Got use of runway " + threadName);
```

降落过程（3，4）——飞机降落并移动到降落区：

```
System.out.println ("Landing " + threadName);
airport.airportMap.LandPlane (this);
System.out.println ("Landed " + threadName);
```

降落过程（5）——释放跑道排斥锁：

```
/* Solution #3 - release mutex when no longer needed */
airport.ReleaseRunwayLock (this);
```

降落过程（6，7）——请求并获取停机位入口排斥锁：

```
/* Solution #1 - terminal gate mutex */
airport.RequestGatesLock (this);
System.out.println ("Got a gate " + threadName);
```

降落过程（8）——滑行到停机位入口：

```
airport.airportMap.TaxiToGate (this);
System.out.println ("At gate " + threadName);
```

在入口消耗一段时间：

```
try
{
    sleep (timeAtGate);
}
catch (InterruptedException e) {}
System.out.println ("Waiting to leave " + threadName);
```

起飞过程（1，2）——请求并获取跑道排斥锁：

```
/* Solution #2 - runway mutex */
airport.RequestRunwayLock (this);
System.out.println ("Got runway for take-off " + threadName);
```

起飞过程（3，4）——飞机滑行到跑道，然后起飞：

```
airport.airportMap.Takeoff (this);
```

起飞过程（5，6）——飞机释放所有的排斥锁：

```
        /* Solution #3 - release mutex when no longer needed */
        airport.ReleaseGatesLock (this);
        airport.ReleaseRunwayLock (this);

        System.out.println ("END " + threadName);
    }
}
```

11.3 一个停机位入口同时有三架飞机

现在，再将机场管理模型改变一下：有三架飞机试图在同一时间使用该机场，当然，机场还是只有一个停机位。

11.3.1 降落过程

飞机在机场中的降落过程描述如下：

▶ **第一架飞机的降落**：下面列出了第一架飞机安全降落必须遵守的顺序步骤。

1. 第一架飞机请求跑道排斥锁。
2. 该飞机获得跑道排斥锁。
3. 该飞机降落。
4. 该飞机释放跑道排斥锁。
5. 该飞机请求停机位排斥锁。
6. 该飞机获得停机位排斥锁。
7. 该飞机滑行到停机位。

▶ **第二架飞机的降落**：下面列出了第二架飞机安全降落必须遵守的顺序步骤。

1. 第二架飞机请求跑道排斥锁。
2. 该飞机获得跑道排斥锁。
3. 该飞机降落。
4. 该飞机释放跑道排斥锁。
5. 该飞机请求停机位排斥锁。
6. 该飞机等待。

1. 问题 # 4—新闻简报："飞机在跑道尽头相撞"

第二架飞机降落后，释放了跑道排斥锁。第三架飞机把它看成是可以降落的信号。不幸的是，第二架飞机正停在跑道尽头等待停机位变为空闲（见图 11-9)。你怎样避免飞机相撞呢？

2. 解决方法 # 4

解决方法是提供一个跑道尽头的排斥锁（见图 11-10)。这个排斥锁用于指示降落区正在使用，其他飞机不可降落。一旦一架飞机已经到达停机位，它必须释放降落区排斥锁，以允许其他飞机使用。

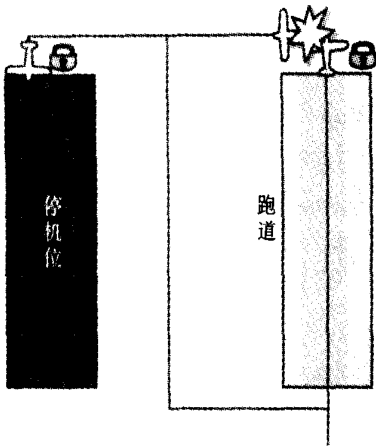


图 11-9 飞机在跑道尽头相撞

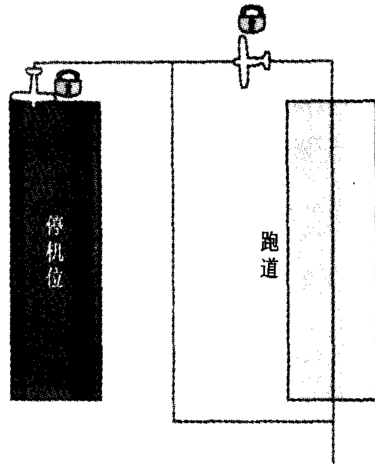


图 11-10 提供跑道尽头的排斥锁

3. 问题 #5——新闻简报："机场无动静"

这是另外一个死锁问题。进入机场空域的飞机已经得到跑道排斥锁，正等待获得降落区排斥锁，但降落区内有另一架飞机在等待获得停机位排斥锁，而停机位内还有一架飞机在等待获得跑道排斥锁——连环死锁（见图 11-11）。你怎样避免这样的死锁呢？

4. 解决方法 #5

进入机场空域的飞机需要在获取跑道排斥锁之前先得到降落区排斥锁（见图 11-12），确认它是否能够降落。这样能保证飞机降落后，能移动到降落区并释放跑道排斥锁。一旦进入机场空域的飞机降落并移动到降落区，就应该释放跑道排斥锁，允许跑道再次用于起飞。

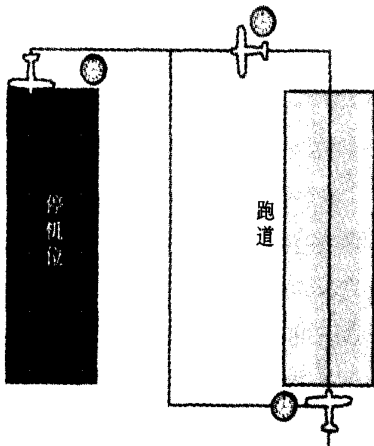


图 11-11 陷于停顿的机场

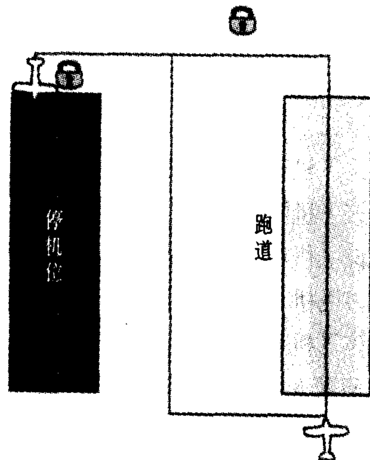


图 11-12 必须先得到降落区排斥锁

11.3.2 起飞过程

起飞过程与以前描述的一样。

11.3.3 修改后的降落过程

下面列出了这些飞机安全降落必须遵守的顺序步骤。

(1)飞机请求降落区排斥锁。

(2)飞机得到降落区排斥锁。

(3)飞机请求跑道排斥锁。

(4)飞机获得跑道排斥锁。

(5)飞机降落,并移动到降落区。

(6)飞机释放跑道排斥锁。

(7)飞机请求停机位排斥锁。

(8)飞机获得停机位排斥锁。

(9)飞机释放降落区排斥锁。

(10)飞机滑行到停机位。

11.3.4 修改后的 Java 代码

新的代码反映了支持解决方法 #1、#2、#3、#4 和 #5 所需的修改。

1. Airport.java

```
public class Airport
{
```

为了解决问题 #4, 需要声明另外一个排斥锁变量。这个排斥锁也是通过向量链表的形式实现的。

```
< code as before >
/* Solution #4 - landing area mutex */
private static Vector            landingList = new Vector ();
/* Solution #2 - runway mutex */
private static Vector            runwayList = new Vector ();
/* Solution #1 - terminal gate mutex */
private static Vector            gatesList = new Vector ();

public Airport()
{
    < code as before accept now using three threads>
}

/* Start the Airport */
public static void main(String[] args)
{
    Airport a = new Airport ();
}
```

附加的方法解决问题 #4, 它们使用降落区排斥锁。

```
    public void RequestLandingLock (Plane plane)
    {
        RequestLock (plane, landingList);
    }

    public void ReleaseLandingLock (Plane plane)
    {
        ReleaseLock (plane, landingList);
    }

    < other mutex methods as before >
}
```

2. AirportMap.java

这部分代码没有变化。

3. Plane.java

```
public class Plane extends Thread
{
    < code as before >
/*
**  Run
*/
    public void run()
    {
        System.out.println ("Waiting to land " + threadName);
```

降落过程（1，2）——请求并获取降落区排斥锁：

```
/* Solution #4 - landing area mutex */
/* Solution #5 - acquire landing area before runaway */
airport.RequestLandingLock (this);
System.out.println ("Got somewhere to land " + threadName);
```

降落过程（3，4）——请求并获取跑道排斥锁：

```
/* Solution #2 - runway mutex */
airport.RequestRunwayLock (this);
System.out.println ("Got use of runway " + threadName);
System.out.println ("Landing " + threadName);
```

降落过程（5）——飞机降落并移动到降落区：

```
airport.airportMap.LandPlane (this);
System.out.println ("Landed " + threadName);
```

降落过程（6）——释放跑道排斥锁：

```
/* Solution #3 - release mutex when no longer needed */
airport.ReleaseRunwayLock (this);
```

降落过程（7，8）——请求并获取停机位入口排斥锁：

```
/* Solution #1 - terminal gate mutex */
airport.RequestGatesLock (this);
System.out.println ("Got a gate " + threadName);
```

降落过程（9）——释放降落区排斥锁。

```
/* Solution #4 - landing area mutex */
airport.ReleaseLandingLock (this);
```

降落过程（10）——滑行到停机位入口：

```
airport.airportMap.TaxiToGate (this);
System.out.println ("At gate " + threadName );
```

在入口消耗一段时间：

```
try
{
    sleep (timeAtGate);
}
catch (InterruptedException e) {}
System.out.println ("Waiting to leave " + threadName);
```

起飞过程（1，2）——请求并获取跑道排斥锁：

```
/* Solution #2 - runway mutex */
airport.RequestRunwayLock ();
System.out.println ("Got runway for take-off " + threadName);
```

起飞过程（3，4）——飞机滑行到跑道，然后起飞：

```
airport.airportMap.Takeoff (this);
```

起飞过程（5，6）——飞机释放所有的排斥锁：

```
            /* Solution #3 - release mutex when no longer needed */
            airport.ReleaseGatesLock (this);
            airport.ReleaseRunwayLock (this);

            System.out.println ("END " + threadName);
        }
}
```

11.4 更多的飞机——再增加一些停机位入口

预期机场的交通频度会大幅度上升，所以要增建一些停机位入口。现在，停机位共有四个入口。

降落过程

降落过程没有变化。

1. 问题 # 6 ——新闻简报："飞机在滑行时相撞"

问题是这样的：一架飞机可能正向停机位及其入口滑行，而另一架飞机可能正滑向跑道准备起飞（见图 11-13)。你怎样避免飞机相撞呢?

2. 解决方法 # 6

解决方法是提供一个滑行的排斥锁（见图 11-14)。在同一时刻，这个排斥锁只允许一架飞机在机场内任意地点滑行。

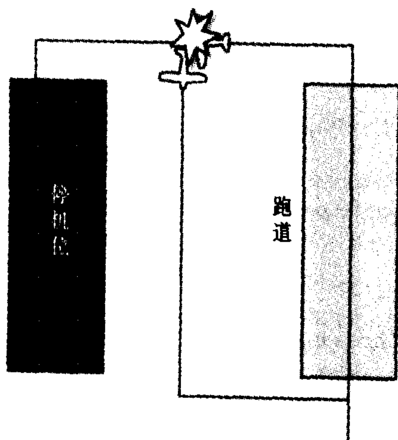


图 11-13 飞机在滑行时相撞

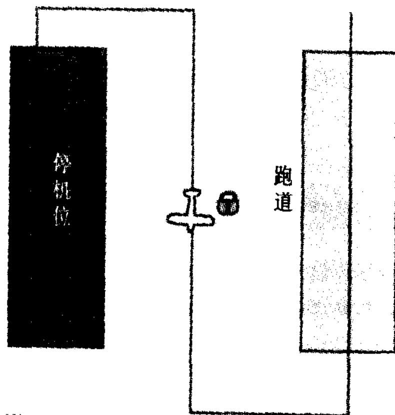


图 11-14 提供滑行的排斥锁

3. 问题 # 7 ——新闻简报："机场寂静无声，等待飞机离开"

问题是这样的：一架飞机要起飞，它必须得到跑道排斥锁和滑行排斥锁。在飞机滑向跑道并起飞的过程中，机场进入停滞和等待的状态。跑道排斥锁和滑行排斥锁是机场中最重要、使用最广泛的排斥锁。如果一架飞机同时持有这两个锁，其他任何动作都无法进行下去了（见图 11-15)。

4. 解决方法 # 7

解决方法是提供另一个等待区，或称为起飞区（Takeoff area)。这次它将被用于起飞。一旦飞机进入起飞区，它能够释放滑行排斥锁，然后请求跑道排斥锁（见图 11-16)。

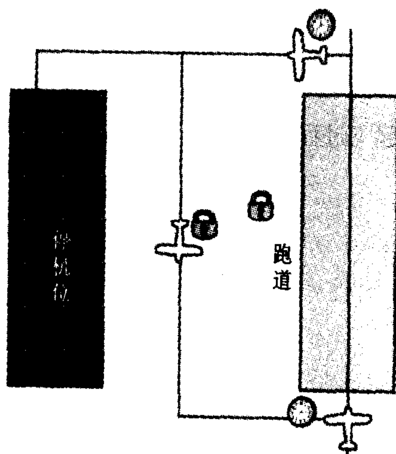


图 11-15 一架飞机同时持有跑道和滑行排斥锁

5. 问题 # 8 ——新闻简报："飞机在试图同时起飞时相撞"

这个问题与问题 # 4 ——多架飞机降落类似。一架飞机一到达起飞区并释放了滑行排斥

锁，另一架飞机就认为这是一个可以滑向跑道起飞的信号（见图 11-17）。

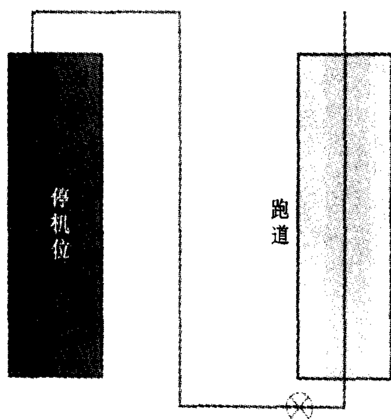


图 11-16 提供起飞区

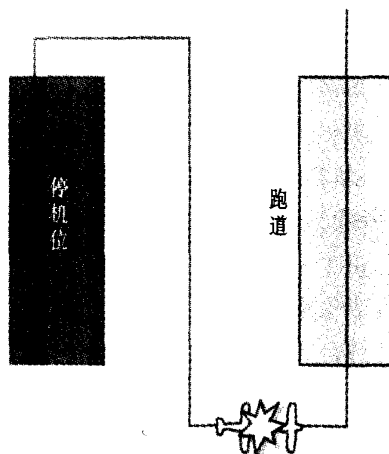


图 11-17 同时起飞时相撞

6. 解决方法 #8

与解决方法 #4 类似，提供一个排斥锁，以保证一架飞机只有在持有这个锁时，才能向起飞区滑行（见图 11-18），例如，当起飞区中没有其他飞机时。

与解决方法 #5 类似，为避免死锁，飞机需要在试图得到滑行锁之前先得到起飞区排斥锁（见图 11-19）。

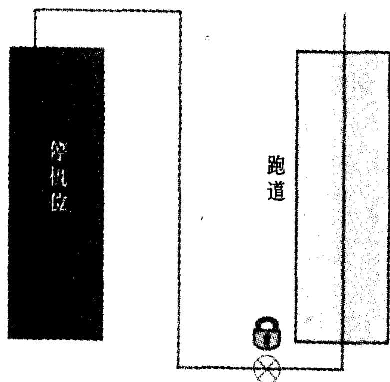


图 11-18 提供起飞区排斥锁

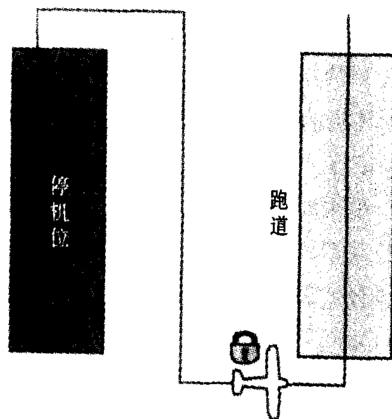


图 11-19 必须先得到起飞区排斥锁

11.5 飞机在机场活动的整个生存周期

图 11-20 说明了排斥锁的位置和它们控制的区域。

▶ **飞机的降落**：下面列出了飞机安全降落必须遵守的顺序步骤。

(1) 请求降落区排斥锁。

(2) 获得降落区排斥锁。

(3) 请求跑道排斥锁。

(4) 获得跑道排斥锁。

(5) <降落并移动到降落区>。

(6) 释放跑道排斥锁。

(7) 请求一个停机位入口排斥锁。

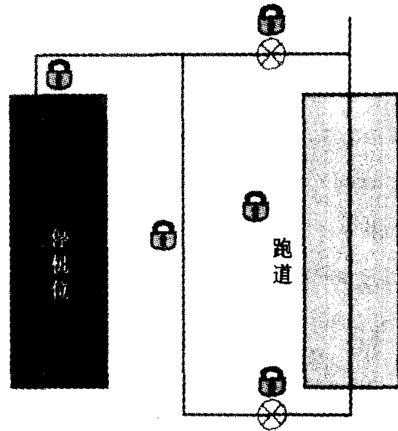


图 11-20 各排斥锁的位置和它们控制的区域

▶ **飞机移动到停机位入口**：下面列出了飞机安全移动到停机位入口必须遵守的顺序步骤。

(1) 获得一个停机位入口排斥锁。

(2) 请求滑行排斥锁。

(3) 得到滑行排斥锁。

(4) 释放降落区排斥锁。

(5) <滑行到分配的停机位入口>。

(6) 释放滑行排斥锁。

▶ **飞机准备好起飞**：下面列出了飞机准备好起飞必须遵守的顺序步骤。

(1) 请求起飞区排斥锁。

(2) 获得起飞区排斥锁。

(3) 请求滑行排斥锁。

(4) 获得滑行排斥锁。

(5) <滑行到起飞区>。

(6) 释放停机位入口排斥锁。

(7) 释放滑行排斥锁。

▶ **飞机的起飞**：下面列出了飞机起飞必须遵守的顺序步骤。

(1) 请求跑道排斥锁。

(2) 获得跑道排斥锁。

(3) 释放起飞区排斥锁。

(4) <起飞>。

(5) 释放跑道排斥锁。

下面的图 11-21 表明了拥有 4 个停机位入口、可供多架飞机使用的机场的情形。

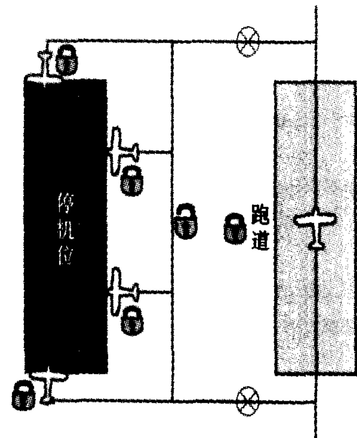


图 11-21 四个停机位入口的机场

修改后的 Java 代码

下面的新代码反映了支持解决方法 # 6、# 7 和 # 8 所需的修改。

1. Airport.java

```
static Vector    gatesLock = new Vector ()
static Integer   gate1 = new Integer (1);
static Integer   gate2 = new Integer (2);
```

```
static Integer   gate3 = new Integer (3);
static Integer   gate4 = new Integer (4);

/* Solution #6 - taxiing area mutex */
static Vector    taxiList = new Vector ();

/* Solution #8 - holding area mutex */
static Vector    holdingList = new Vector ();
```

对 ‘gatesLock’ 向量同步，使得 4 个停机位入口均可被加入其中。

```
synchronized (gatesLock)
{
    gatesLock.addElement (gate1);
    gatesLock.addElement (gate2);
    gatesLock.addElement (gate3);
    gatesLock.addElement (gate4);
}
```

以前，请求排斥锁的代码是这样实现的：

```
public void RequestGatesLock (Plane plane)
{
    RequestLock (plane, gatesList);
}
```

以前的方法 ‘RequestGatesLock’ 只考虑了一个停机位入口。这个方法的实现已被重新设计，可处理 4 个停机位入口：

```
public Integer RequestGatesLock (Plane plane)
{
    boolean    empty;

    Integer    atGate;

    synchronized (gatesLock)
    {
        empty = gatesLock.isEmpty ();

        synchronized (gatesList)
        {
            gatesList.addElement (plane);
        }
    }
    if (empty == true)
    {
        try {plane.wait ();}
            catch (InterruptedException e) {}
    }
```

```
        synchronized (gatesLock)
        {
            atGate = (Integer)gatesLock.firstElement ();
            gatesLock.removeElement (atGate);

            synchronized (gatesList)
            {

            gatesList.removeElement (plane);
        }
    }

    return (atGate);
}
```

下面是其他“请求”方法的实现：

```
public void RequestTaxiLock (Plane plane)
{
    RequestLock (plane, taxiList);
}

public void RequestHoldingLock (Plane plane)
{
    RequestLock (plane, holdingList);
}
```

以前，释放排斥锁的代码是这样实现的：

```
/* old code */
public void ReleaseGatesLock (Plane plane)
{
    ReleaseLock (plane, gatesList);
}
```

以前的方法‘ReleaseGatesLock’只考虑了一个停机位入口。这个方法的实现已被重新设计，可处理4个停机位入口：

```
public void ReleaseGatesLock (Integer atGate)
{
    Plane    peek;

    synchronized (Airport.gatesLock)
    {
        Airport.gatesLock.addElement (atGate);

        synchronized (Airport.gatesList)
        {
            if (Airport.gatesList.isEmpty () == false)
```

```
        {
            try
            {
                peek = (Plane)Airport.gatesList.firstElement ();
                synchronized (peek)
                {
                    peek.notify ();
                }
            }
            catch (NoSuchElementException e) {}
        }
    }
  }
}
```

下面是其他“释放”方法的实现：

```
/* new code */
public void ReleaseTaxiLock (Plane plane)
{
    ReleaseLock (plane, taxiList);
}

public void ReleaseHoldingLock (Plane plane)
{
    ReleaseLock (plane, holdingList);
}
```

2. AirportMap.java

```
/*
**  TaxiToGateOne
*/
public void TaxiToGateOne (Plane plane)
{
    /*
    ** Display the aircraft moving west from the landing area and
    ** south towards the terminal
    ** Gate 1 is at the northern end of the terminal
    */
}
/*
**  TaxiToGateTwo
*/
public void TaxiToGateTwo (Plane plane)
{
    /*
    ** Display the aircraft moving west from the landing area and
```

```
    ** south towards the terminal
    ** Gate 2 is the topmost gate on the eastern side of the terminal
    */
}

/*
**  TaxiToGateThree
*/
public void TaxiToGateThree (Plane plane)
{
    /*
    ** Display the aircraft moving west from the landing area and
    ** south towards the terminal Gate 3 is the bottommost gate on
    ** the eastern side of the terminal
    */

}

/*
**  TaxiToGateFour
*/
public void TaxiToGateFour (Plane plane)
{
    /*
    ** Display the aircraft moving west from the landing area and
    ** south towards the terminal
    ** Gate 2 is at the southern end of the terminal
    */
}

/*
**  TaxiFromGateOne
*/
public void TaxiFromGateOne (Plane plane)
{
    /*
    ** Display the aircraft moving up and around the terminal,
    ** towards the holding area just to the west of the southern
    ** end of the runway
    */
}

/*
**  TaxiFromGateTwo
*/
public void TaxiFromGateTwo (Plane plane)
{
    /*
```

```
    ** Display the aircraft moving away from the terminal, and south
    ** towards the holding area just to the west of the southern end
    ** of the runway
    */
}

/*
**  TaxiFromGateThree
*/
public void TaxiFromGateThree (Plane plane)
{
    /*
    ** Display the aircraft moving away from the terminal, and south
    ** towards the holding area just to the west of the southern end
    ** of the runway
    */
}

/*
**  TaxiFromGateFour
*/
public void TaxiFromGateFour (Plane plane)
{
    /*
    ** Display the aircraft moving away from the terminal, towards
    ** the holding area just to the west of the southern end of the
    ** runway
    */
}

/*
** Method Name: Takeoff
** This method is used to show an aircraft taxiing from the
** terminal gate onto the runway and then taking off
**
** Input: A handle to the aircraft to use
** Output: none
*/
public void TakeOff (Plane plane)
{
    /*
    ** Display the aircraft moving east away from the holding area
    ** to the start of the runway, then finally north as it flies
    ** away from the airport
    */
}
```

3. Plane.java

```
public class Plane extends Thread
{
        < code as before >
/*
**  Run
*/
public void TaxiToGate (int gateNo)
{
    if (gateNo == 1)
    {
        airport.airportMap.TaxiToGateOne (this);
    }
    else if (gateNo == 2)
    {
        airport.airportMap.TaxiToGateTwo (this);
    }
    else if (gateNo == 3)
    {
        airport.airportMap.TaxiToGateThree (this);
    }
    else
    {
        airport.airportMap.TaxiToGateFour (this);
    }
}
/*
**
*/
public void TaxiFromGate (int gateNo)
{
    if (gateNo == 1)
    {
        airport.airportMap.TaxiFromGateOne (this);
    }
    else if (gateNo == 2)
    {
        airport.airportMap.TaxiFromGateTwo (this);
    }
    else if (gateNo == 3)
    {
        airport.airportMap.TaxiFromGateThree (this);
    }
    else
    {
        airport.airportMap.TaxiFromGateFour (this);
    }
}
```

```
/*
**  Run
*/
public void run()
{
    Integer    atGate;

    System.out.println ("Waiting to land " + threadName);
```

降落过程（1，2）——请求并获取降落区排斥锁：

```
/* Solution #4 - landing area mutex */
/* Solution #5 - acquire landing area before runaway */
airport.RequestLandingLock (this);
System.out.println ("Got somewhere to land " + threadName);
```

降落过程（3，4）——请求并获取跑道排斥锁：

```
/* Solution #2 - runway mutex */
airport.RequestRunwayLock (this);
System.out.println ("Got use of runway " + threadName);
System.out.println ("Landing " + threadName);
```

降落过程（5）——飞机降落并移动到降落区：

```
airport.airportMap.LandPlane (this);
System.out.println ("Landed " + threadName);
```

降落过程（6）——释放跑道排斥锁：

```
/* Solution #3 - release mutex when no longer needed */
airport.ReleaseRunwayLock (this);
```

降落过程（7）——请求停机位入口排斥锁：
移动到停机位入口过程（1）——得到停机位入口排斥锁：

```
/* Solution #1 - terminal gate mutex */
atGate = airport.RequestGatesLock (this);
System.out.println ("Got a gate " + threadName);
```

移动到停机位入口过程（2）——请求并得到滑行排斥锁：

```
/* Solution #6 - taxiing mutex */
airport.RequestTaxiLock (this);
System.out.println ("Got use of taxiway " + threadName + "\n");
System.out.println ("Taxiing " + threadName + "\n");
```

移动到停机位入口过程（4）——释放降落区排斥锁：

```
/* Solution #4 - landing area mutex */
airport.ReleaseLandingLock (this);
```

移动到停机位入口过程（5）——滑行到指定的停机位入口：

```
TaxiToGate (atGate.intValue ());
System.out.println ("At gate " + threadName + "\n");
```

移动到停机位入口过程（6）——释放滑行排斥锁：

```
/* Solution #5 - release lock to avoid deadlock*/
airport.ReleaseTaxiLock (this);
```

在入口消耗一段时间：

```
try
{
    sleep (timeAtGate);
}
catch (InterruptedException e) {}
System.out.println ("Waiting to leave " + threadName + "\n");
```

准备好起飞过程（1，2）——请求并得到起飞区排斥锁：

```
/* Solution #5 - holding lock */
airport.RequestHoldingLock (this);
System.out.println
    ("Got holding area lock " + threadName + "\n");
```

准备好起飞过程（3，4）——请求并得到滑行排斥锁：

```
/* Solution #8 - the order in which mutexes are acquired */
/* Solution #6 - taxiing lock */
airport.RequestTaxiLock (this);
System.out.println ("Got use of taxiway " + threadName + "\n");
System.out.println ("Taxiing " + threadName + "\n");
```

准备好起飞过程（5）——从停机位入口滑行到起飞区：

```
TaxiFromGate (atGate.intValue ());
System.out.println ("At holding area " + threadName + "\n");
```

准备好起飞过程（6）——释放停机位入口排斥锁：

```
/* Solution #3 - release mutex when no longer needed */
airport.ReleaseGatesLock (atGate);
```

准备好起飞过程（7）——释放滑行排斥锁：

```
/* Solution #5 - taxiing lock */
airport.ReleaseTaxiLock (this);
System.out.println ("At holding area " + threadName + "\n");
```

起飞过程（1，2）——请求并得到跑道排斥锁：

```
/* Solution #2 - runway lock */
airport.RequestRunwayLock (this);
System.out.println ("Got runway " + threadName + "\n");
```

起飞过程（3）——释放起飞区排斥锁：

```
airport.ReleaseHoldingLock (this);
```

起飞过程（4）——飞机起飞：

```
System.out.println ("Taking off " + threadName + "\n");
airport.airportMap.TakeOff (this);
```

起飞过程（5）——释放跑道排斥锁：

```
    /* Solution #3 - release mutex when no longer needed */
    airport.ReleaseRunwayLock (this);

    System.out.println ("END " + threadName + "\n");
}
```

11.6 最终的解决方案

最终的解决方案中，另外增加了 3 个类：

- **AirportDetails**：作为显示每架飞机细节信息的窗口。窗口显示的细节包括：哪架飞机正在请求、得到和释放哪些排斥锁，还有，每架飞机得到某个排斥锁后，都在做什么。
- **MediumPlane** 和 **SmallPlane**：增加这两个派生类是为了支持多种类型的飞机。SmallPlane 的尺寸小一半，因此它在停机位入口等待的时间也少一半。

11.6.1 机场细节信息窗口

下面的图 11-22 显示了所有的飞机在接近、使用机场，和从机场起飞时处于不同状态时的信息。

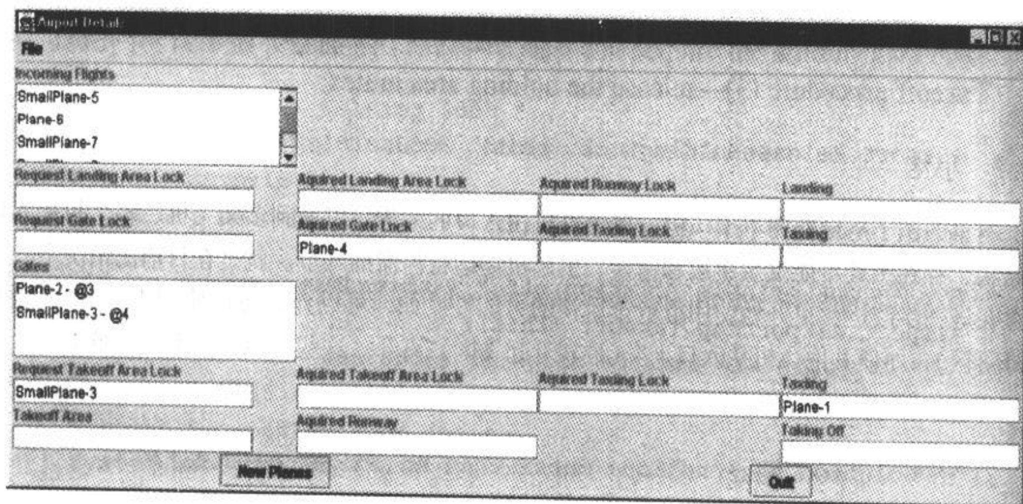


图 11-22　机场细节信息窗口

11.6.2 Java代码

下面的新代码反映了为支持 **AirportDetails** 类和两个飞机派生类所作的改动。

Airport.java

```
public class Airport
{
    public AirportDetails    airportDetails;
    ...
    private Random generator =
        new Random(System.currentTimeMillis());
...
    public Airport()
    {
        airportDetails = new AirportDetails ("Airport Details");
        ...
        Plane planes [] = new Plane [10];
        /* Create the threads */
        for (int j = 0; j < 10; j++)
        {
            if (generator.nextInt (2) < 1)
            {
                planes [j] = new Plane (threadGroup, j, this);
            }
            else
            {
                planes [j] = new Plane (threadGroup, j, this);
            }
        }
        ...
    }
}
```

11.7 小结

本章的例子从最初的单架飞机、单个停机位入口的结构，发展到多架飞机、多个机场入口的结构，存在多处可能发生事故和错误、引起机场运行停滞的环节。就是在目前的情况下，仍然存在进一步改进、更好地使用滑行排斥锁的可能性。如，可以允许飞机使用入口 #1（最靠近降落区），而使其他飞机从入口 #4（最靠近起飞区）离开。这些就要留到下一次解决了。

附录 “哲学家”源代码

以下是前面几章中提到的“哲学家”应用程序的一种实现，本程序是使用 Visual C++® 实现的。

Windows Visual C++®

phils.cc

```
/*
**  Name: phils.cc - Dining Philosophers
**
** Description:
**  There are 10 philosophers who spend their lives either eating or
**  thinking.
**  Each philosopher has his own place at a circular table, in the
**  center of which is a large bowl of rice.  To eat rice requires
**  two chopsticks, but only 10 chopsticks are provided, one between
**  each pair of philosophers. The only chopsticks a philosopher can
**  pick up are those on his immediate right and left.
**  Each philosopher is identical in structure, alternately eating
**  then thinking. The problem is to simulate the behaviour of the
**  philosophers while avoiding deadlock (the request by a philosopher
**  for a chopstick can never be granted) and indefinite postponement
**  (the request by a philosopher is continually denied), known
**  colloquially as "starvation."
**
**  Created by : Andrew Haigh
**  Change history
**      20-mar-2001 (Andrew Haigh)
**          created
*/

/*
**  system include files
*/
#include <windows.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <conio.h>
#include <process.h>

/*
**  local variables

*/
HANDLE  hChopsticksMutex [numPhilosophers];
HANDLE  hSyncEvent;
int     ThreadNr;                    /* Number of threads started */
```

```
/*
**  local defines
*/
#define numPhilosophers 10    // number of philosophers
#define numBites        5     // number of bites to take

/*
** getrandom returns a random number between min and max,
** which must be in integer range.
*/
#define getrandom( min, max ) ((rand() % (int) \
                  (((max) + 1) - (min))) + (min))

/*
**  forward declaration of methods
*/
void main( void );                          /* Thread 1: main */
void PhilosopherProc (int * MyID);          /* Threads 2 to n: display */

/*
**  Psem
**  Purpose of method
**      This method will lock the mutex and return
**      If the mutex is not free, it will wait until it is
**  Input
**      mutex - handle of the mutex
**      seq - number to be printed in the message
**  Output/Return
**      method is void
**  Change History
**      20-mar-2001 (Andrew Haigh)
**          created
*/
void
Psem (HANDLE mutex, int seq)
{
    DWORD stat;
    stat = WaitForSingleObject( mutex, INFINITE );
    if(stat == WAIT_FAILED)
    {
      printf ("Error lock - %d = %d\n", seq, stat);
      exit(1);
    }
}

/*
**  Psem_try
**  Purpose of method
**      This method will try to lock the mutex
**      If the mutex if free, acquire it and return
**      If the mutex is already locked it will return WAIT_TIMEOUT
**  Input
**      mutex - handle of the mutex
**      time - period to wait for the mutex to become free
**      seq - number to be printed in the message
```

```
**  Output/Return
**      return 0 - when it succeeds
**      return WAIT_TIMEOUT - when it timed-out
**      exit 1 - when it fails
**  Change History
**      20-Mar-2001 (Andrew Haigh)
**          created
*/
DWORD
Psem_try (HANDLE mutex, DWORD time, int seq)
{
    DWORD stat;
    stat = WaitForSingleObject( mutex, time );
    if(stat == WAIT_TIMEOUT)
    {
        return WAIT_TIMEOUT;
    }
    else if (stat == WAIT_FAILED)
    {
        printf ("Error trylock - %d = %d\n", seq, stat);
        exit(1);
    }
    else
        return 0;
}

/*
**  Vsem
**  Purpose of method
**      Release the mutex
**  Input
**      mutex - handle of the mutex
**      seq - number to be printed in the message
**  Output/Return
**      method is void
**  Change History
**      20-mar-2001 (Andrew Haigh)
**          created
*/
void
Vsem (HANDLE mutex, int seq)
{
    DWORD stat;
    stat = ReleaseMutex ( mutex );
    if(stat == 0)
    {
        printf ("Error unlock - %d = %d\n", seq, stat);
        exit(1);
    }
}

/*
```

```
**  main
**  Purpose of method
**      This method controls the program,
**      it creates the mutexes, it then creates the threads,
**      starts the threads and then waits for them to finish
**  Input
**      argc - number of command line arguments
**      argv - the command line arguments
**  Output/Return
**      method is void
**  Change History
**      20-mar-2001 (Andrew Haigh)
**          created
*/
void
main(int argc, char **argv)                         /* Thread One */
{
    int    i;
    int    *buffer;

    /* Create the mutexes and reset thread count. */
    hSyncEvent = CreateEvent (NULL,TRUE, FALSE, "SyncCond");
    for (i = 0; i < numPhilosophers; i ++)
        hChopsticksMutex [i] = CreateMutex (NULL, FALSE, NULL);

    ThreadNr = 10;

    /*
    ** Start waiting for keyboard input to dispatch threads or exit
     */
     for (i = 0; i < numPhilosophers; i++)
     {
         buffer = calloc (2, sizeof (int));
         *buffer = i;
         _beginthread (PhilosopherProc, 0, buffer);
     }
     SetEvent (hSyncEvent);

     /*
     **  wait to be signalled that the threads have finished
     */
     do
     {
         Sleep (2000L);
         Psem (hSyncEvent, 99);
         ResetEvent (hSyncEvent);
     }
     while (ThreadNr != 0);

     /*
     ** All threads done. Clean up handles
     */
```

```
    for (i = 0; i < numPhilosophers; i ++)
    {
        CloseHandle (hChopsticksMutex [i]);
    }
    CloseHandle (hSyncEvent);
}

/*
**  PhilosopherProc
**  Purpose of method
**      This is the process run by each philosopher thread
**  Input
**      MyId - unique thread identifier
**  Output/Return
**      method is void
**  Change History
**      20-mar-2001 (Andrew Haigh)
**          created
*/
void
PhilosopherProc( int *MyID )
{
    int     I, nextstick;
Psem (hSyncEvent, 0);

nextstick = (*MyID+1)%numPhilosophers;

for (i = 0; i < numBites; i++)
{
    printf ("Philosopher %d thinking\n", *MyID);
    Sleep (0);  // yield

    // hungry, so picks up chopsticks
    Psem (hChopsticksMutex[*MyID], 3);

    while (Psem_try (hChopsticksMutex[nextstick], 0, 4) == 1)
    {
        printf ("Philosopher %d waiting\n", *MyID);
        Vsem (hChopsticksMutex[*MyID], 3);
        Sleep (getrandom (1, 100));
        Psem (hChopsticksMutex[*MyID], 3);
    }

    // eating
    printf ("Philosopher %d eating bite %d\n", *MyID, i);
    Sleep (100L);  // yield

    Vsem (hChopsticksMutex[(*MyID+1)%numPhilosophers], 4);
    Vsem (hChopsticksMutex[*MyID], 3);

    Sleep (getrandom (500, 1000));  // yield
}
```

```
    printf ("Thread %d ending\n", *MyID);
    ThreadNr--;
    SetEvent (hSyncEvent);
    return;
}
```

UNIX C++

以下是“哲学家”应用程序在 Unix 系统中的一种实现，它适用于 Unix C++环境和 Unix Posix 线程。

phils.cpp

```
/*
**  Name: phils.cc - Dining Philosophers
**
**  Description:
**  There are 10 philosophers who spend their lives either eating or
**  thinking.
**  Each philosopher has his own place at a circular table, in the
**  center of which is a large bowl of rice.  To eat rice requires
**  two chopsticks, but only 10 chopsticks are provided, one between
**  each pair of philosophers. The only chopsticks a philosopher can
**  pick up are those on his immediate right and left.
**  Each philosopher is identical in structure, alternately eating
**  then thinking. The problem is to simulate the behaviour of the
**  philosophers while avoiding deadlock (the request by a philosopher
**  for a chopstick can never be granted) and indefinite postponement
**  (the request by a philosopher is continually denied), known
**  colloquially as "starvation."
**
**  Created by : Andrew Haigh
**  Change history
**      20-mar-2001 (Andrew Haigh)
**          created
*/

/*
**  system include files
*/
# include <stdlib.h>
# include <iostream.h>
# include <sys/errno.h>
# include <unistd.h>

/*
** local include files
*/
# define  POSIX_THREADS
# include <PortableThreads.h>

/*
**  local variables
```

```
*/
const int  numPhilosophers = 10;    // number of philosophers
const int  numBites = 5;            // number of bites to take

// Setup the chopsticks as mutexs
PM_t       chopsticks[numPhilosophers];

// Provide a mutex to the resource 'threadCount'
int        threadCount = 0;
PM_t       threadCountMutex;

// Provide a condition variable and Boolean to allow the threads
// to have a synchronized start, and a mutex to setup the condition
PM_t       syncCondMutex;
int        syncBit;
PC_t       syncCond;

/*
**  Psem
**  Purpose of method
**      This method will lock the mutex and return
**      If the mutex is not free, it will wait until it is
**  Input
**      mutex - handle of the mutex
**      seq - number to be printed in the message
**  Output/Return
**      method is void
**  Change History
**      20-mar-2001 (Andrew Haigh)
**          created
*/
void
Psem (PM_t *mutex, int seq)
{
    int stat;
    // request a lock on a specific mutex
    stat = PM_lock (mutex);
    if(stat)
    {
        // if the request fails, report error to the user
        // do not try and recover
        cout << "Error PM_lock - " << seq << " = " << stat << endl;
       exit(1);
    }
}

/*
**  Psem_try
**  Purpose of method
**      This method will try to lock the mutex
**      If the mutex if free, acquire it and return
**      If the mutex is already locked it will return WAIT_TIMEOUT
```

```
**  Input
**      mutex - handle of the mutex
**      seq - number to be printed in the message
**  Output/Return
**      return 0 - when it succeeds
**      return 1 - when the mutex is not free
**      exit 1 - when it fails
**  Change History
**      20-Mar-2001 (Andrew Haigh)
**          created
*/
int
Psem_try (PM_t *mutex, int seq)
{
    int stat;
    // try and get the lock on the mutex
    stat = PM_trylock (mutex);
    if(stat == EBUSY)
    {
        return 1;
    }
    else if (stat != 0)
    {
        cout << "Error PM_lock - " << seq << " = " << stat << endl;
        exit(1);
    }
    else
        return 0;
}

/*
**  Vsem
**  Purpose of method
**      Release the mutex
**  Input
**      mutex - handle of the mutex
**      seq - number to be printed in the message
**  Output/Return
**      method is void
**  Change History
**      20-mar-2001 (Andrew Haigh)
**          created
*/
void
Vsem (PM_t *mutex, int seq)
{
    // release the lock on a specific mutex
    if (PM_unlock (mutex))
    {
        // if the release fails, report error to the user
        // do not try and recover
```

```
        cout << "Error PM_unlock - " << seq << " = " << stat << endl
        exit(1);
    }
}

/*
**  PhilosopherThread
**  Purpose of method
**      This is the process run by each philosopher thread
**      Each philosopher tries to pick up both chopsticks
**      When they have both chopsticks, they eat and return the
**      chopsticks and wait/think. This is repeated 'numBites' times
**  Input
**  Output/Return
**      method is void *
**      return value 0 - when it succeeds
**  Change History
**      20-mar-2001 (Andrew Haigh)
**          created
*/
void *
PhilosopherThread()
{
    int thrid, stat;
    int nextstick;
    PT_t self = PT_get_thread_id();

    Psem (&threadCountMutex, 1);

    thrid = threadCount;
    threadCount++;
    cout << "Thread " << thrid << " entry point self=" << self
        << " addr = "<< &chopsticks << endl;

    Vsem (&threadCountMutex, 1);

    cout << "Local thread id = " << thrid << endl;

    // wait on a condition broadcast to start thread
    Psem (&syncCondMutex, 2);

    while (!syncBit)
    {
        stat = pthread_cond_wait(&syncCond, &syncCondMutex);
        if(stat)
        {
            cout << "Error pthread wait1 = " << stat << endl;
            exit(1);
        }
    }

    cout << "Thread " << thrid <<
```

```
        " released and beginning run." << endl;

    Vsem (&syncCondMutex, 2);

    for ( int bites = 1; bites <= numBites; bites++ )
    {
        // thinking
        cout << "Philosopher " << thrid << " thinking" << endl;
        PT_yield();

        // hungry, so picks up chopsticks
        Psem (&chopsticks[thrid], 3);

        Psem_try (thrid, &chopsticks[thrid],
            &chopsticks[(thrid+1)%numPhilosophers], 4);

        // eating
        cout << "Philosopher " << thrid << " eating" <<
            " bite " << bites << endl;
        PT_yield();

        // done eating
        Vsem (&chopsticks[(thrid+1)%numPhilosophers], 4);

        PT_yield();

        Vsem (&chopsticks[thrid], 3);
    }
    cout << "Thread " << thrid << " ending self=" << self << endl;
    return 0;
}

/*
**  main
**  Purpose of method
**      This method controls the program,
**      it creates the mutexes, it then creates the threads,
**      starts the threads and then waits for them to finish
**  Input
**      argc - number of command line arguments
**      argv - the command line arguments
**  Output/Return
**      return 0 - is successful finish
**      exit (1) - is a failure
**  Change History
**      20-mar-2001 (Andrew Haigh)
**          created
*/
int
main (int argc, char **argv)
{
    PT_t phils[numPhilosophers];
```

```
int i, stat;
void *ret;

/*
** initialize the thread counter and the chopsticks
*/
stat = PM_init(&threadCountMutex,NULL);
Psem (&threadCountMutex, 0);
for ( i=0; i < numPhilosophers; i++ )
{
    cout << "Init mutex no.=" << i << " addr=" <<
        &chopsticks[i] << endl;
    stat = PM_init(&chopsticks[i],NULL);
    if(stat)
    {
        cout << "Error PT_init:1 = " << stat << endl;
        exit(1);
    }
}

/*
** initialize the synchronization flag
** that starts the threads running
*/
syncBit = 0;
stat = PM_init(&syncCondMutex, NULL);
if(stat)
{
    cout << "Error PT_init:2 = " << stat << endl;
    exit(1);
}
stat = PC_init(&syncCond, NULL);
if(stat)
{
    cout << "Error PC_init = " << stat << endl;
    exit(1);
}

    // Create the philosopher threads
    for ( i=0; i < numPhilosophers; i++ )
    {
        // create the philosopher threads
        PT_create(NULL, NULL, (void *(*)(void*))PhilosopherThread,
            NULL, &phils[i], PTHREAD_CREATE_JOINABLE, &stat);
    }
    cout << numPhilosophers << " threads created." << endl;

    Vsem (&threadCountMutex, 6);

    // now wake them all up at once
    Psem (&syncCondMutex, 7);

    syncBit = 1;
```

```
    stat = PC_broadcast(&syncCond);
    if(stat)
    {
        cout << "Error PC_broadcast = " << stat << endl;
        exit(1);
    }
    cout << "Startup condition signaled." << endl;

    Vsem (&syncCondMutex, 7);

    // wait for all threads to finish execution
    for ( i = 0; i < numPhilosophers; i++ )
    {
        stat = PT_join( phils[i], &ret );
        if(stat)
        {
            cout << "Error PT_join" << i <<"  = " << stat << endl;
            exit(1);
        }
    }

    cout << "Normal shutdown" <<endl;

    return 1;
}
```

Java

这些 Java 源文件中，第一个是控制文件，用于管理包含 Java 线程运行代码的其他文件。

Dinner.java

```
/*
**  Name: Dinner.java - Dining Philosophers
**
**  Description:
**  There are 10 philosophers who spend their lives either eating or
**  thinking.
**  Each philosopher has his own place at a circular table, in the
**  center of which is a large bowl of rice.  To eat rice requires
**  two chopsticks, but only 10 chopsticks are provided, one between
**  each pair of philosophers. The only chopsticks a philosopher can
**  pick up are those on his immediate right and left.
**  Each philosopher is identical in structure, alternately eating
**  then thinking. The problem is to simulate the behaviour of the
**  philosophers while avoiding deadlock (the request by a philosopher
**  for a chopstick can never be granted) and indefinite postponement
**  (the request by a philosopher is continually denied), known
**  colloquially as "starvation."
**
**  Created by : Andrew Haigh
**  Change history
**      20-mar-2001 (Andrew Haigh)
**          created
*/
```

```
import java.io.*;
import java.lang.*;
import java.util.*;

public class Dinner
{
    protected ThreadGroup    threadGroup;
    static Boolean           chopstickLock [] = new Boolean [10];
    static int               numPhils = 5;

    /*
    **  Dinner
    **  Purpose of method
    **      This method controls the program,
    **      it creates the mutexes, it then creates the threads,
    **      starts the threads and then waits for them to finish
    **  Input
    **  Output/Return
    **      method is the class constructor
    **  Change History
**      20-mar-2001 (Andrew Haigh)
**          created
*/
public Dinner()
{
    // Create the threadgroup.
    threadGroup = new ThreadGroup("Dinner");

    synchronized (chopstickLock)
    {
        for (int j = 0; j < numPhils; j++)
        {
            chopstickLock [j] = new Boolean (false);
        }
    }

    Phils phils [] = new Phils [numPhils];

    try
    {
        // Start the thread
        for (int j = 0; j < numPhils; j++)
        {
            phils [j] = new Phils (threadGroup, j, this);
            phils [j].start ();
        }

        for (int j = 0; j < numPhils; j++)
        {
            phils [j].join ();
        }
```

```
        }
        catch(InterruptedException e){}
    }

    // Start the application
    public static void main(String[] args)
    {
        Dinner a = new Dinner ();
    }

}
```

Phils.java

```
/*
**  Name: Phils.java - Philosophers thread
**
**  Description:
**      This is the process run by each philosopher thread
**      Each philosopher tries to pick up both chopsticks
**      When they have both chopsticks, they eat and return the
**      chopsticks and wait/think. This is repeated 'numBites' times
*/

import java.io.*;
import java.lang.*;
import java.util.*;

public class Phils extends Thread
{
    String    threadName;
    int       thrid;
    Dinner    dinner;

    /*
    **  Phils
    **      Each philosopher tries to pick up both chopsticks
    **      When they have both chopsticks, they eat and return the
    **      chopsticks and wait/think. This is repeated 'numBites' times
    **  Input
    **      threadGroup - group to which the thread belongs, a group
    **                    allows to be manipulated as a unit
    **      name - unique thread identifier
    **      parent - handle back to the thread owner
    **  Output/Return
    **      method is the class constructor
    **  Change History
    **      20-mar-2001 (Andrew Haigh)
    **          created
    */
    public Phils (ThreadGroup threadGroup, int name, Dinner parent)
    {
        // Create our server thread with a name.
```

```
        super(threadGroup, "Philosopher-" + name);

        dinner = parent;

        threadName = new String ("Philosopher-" + name);
        thrid = name;
    }

    /*
    **  Psem
**  Purpose of method
**      This method will lock the mutex and return
**      If the mutex is not free, it will wait until it is
**  Input
**      pos - the number of the mutex
**  Output/Return
**      method is void
**  Change History
**      20-mar-2001 (Andrew Haigh)
**          created
*/
public void Psem (int pos)
{
    boolean    flag;

    /*
    ** lock the mutex variable for single thread access
    */
    System.out.println (threadName + " RequestLock");
    synchronized (Dinner.chopstickLock [pos])
    {
        /*
        ** if the mutex is free acquire it and return
        */
        flag = Dinner.chopstickLock [pos].booleanValue ();
        if (flag == false)
        {
            Dinner.chopstickLock [pos] = new Boolean (true);
            System.out.println (threadName + " AcquiredLock");
            return;
        }
    }

    /*
    ** the mutex was already locked
    ** loop and keep testing the mutex availability
    ** until it becomes free
    */
    while (flag == true)
    {
        synchronized (Dinner.chopstickLock [pos])
        {
```

```
            flag = Dinner.chopstickLock [pos].booleanValue ();
            if (flag == false)
            {
                Dinner.chopstickLock [pos] = new Boolean (true);
                System.out.println (threadName + " AcquiredLock");
                  return;
              }
          }
          try {sleep (20);}
          catch (InterruptedException e) {}
      }
  }

  /*
  **  Psem_try
  **  Purpose of method
  **      This method will try to lock the mutex
  **      If the mutex if free, acquire it and return
  **      If the mutex is already locked it will return WAIT_TIMEOUT
  **  Input
  **      pos - the number of the mutex
  **  Output/Return
  **      return value 0 - when it succeeds
  **      return value 1 - when it fails
  **  Change History
  **      20-Mar-2001 (Andrew Haigh)
  **          created
  */
  public int Psem_try (int pos)
  {
      System.out.println (threadName + " Psem_try " + pos);
      synchronized (Dinner.chopstickLock [pos])
      {
          if (Dinner.chopstickLock [pos].booleanValue () == false)
          {
              Dinner.chopstickLock [pos] = new Boolean (true);
              System.out.println (threadName + " AcquiredLock");
              return 0;
          }
          else
          {
              System.out.println (threadName + " Lock - waiting");
              return 1;
          }
      }
  }

  /*
  **  Vsem
  **  Purpose of method
  **      Release the mutex
  **  Input
```

```
**      pos - the number of the mutex
**  Output/Return
**      method is void
**  Change History
**      20-mar-2001 (Andrew Haigh)
**          created
*/
public void Vsem (int pos)
{
    System.out.println (threadName + " Vsem " + pos);
    synchronized (Dinner.chopstickLock [pos])
    {
        Dinner.chopstickLock [pos] = new Boolean (false);
    }
}

/*
**  Run
**  Purpose of method
**      This is the process run by each philosopher thread
**      Each philosopher tries to pick up both chopsticks
**      When they have both chopsticks, they eat and return the
**      chopsticks and wait/think. This is repeated 'numBites' times
**  Input
**  Output/Return
**      method is void *
**  Change History
**      20-mar-2001 (Andrew Haigh)
**          created
*/
public void run()
{
    int     next = thrid + 1;

    if (next == Dinner.numPhils)
        next = 0;

    System.out.println ("Go " + threadName);

    for (int i = 0; i < 5; i++)
    {
        Psem (thrid);

        while (Psem_try (next) == 1)
        {
            Vsem (thrid);
            try {sleep (20);}

          catch (InterruptedException e) {}
          Psem (thrid);
      }
```

```
            System.out.println ("All chopsticks acquired" +
                threadName);

            try {sleep (20);}
            catch (InterruptedException e) {}

            Vsem (next);
            Vsem (thrid);

            try {sleep (20);}
            catch (InterruptedException e) {}
        }

        System.out.println ("END " + threadName);
    }
}
```

Java JNI（C++）

下面的文件与前面使用 Java 和 Unix C++ 编写的“哲学家”程序类似。惟一的区别是，C++代码不是通过 main 过程调用的，而是由 Java 代码调用。在 C++代码线程开始之前，需要对线程环境初始化，所以有过程“StartUp”和“ShutDown”。

Dinner.java

```
/*
**  Name: Dinner.java - Dining Philosophers
**
**  Description:
**  There are 10 philosophers who spend their lives either eating or
**  thinking.
**  Each philosopher has his own place at a circular table, in the
**  center of which is a large bowl of rice.  To eat rice requires
**  two chopsticks, but only 10 chopsticks are provided, one between
**  each pair of philosophers. The only chopsticks a philosopher can
**  pick up are those on his immediate right and left.
**  Each philosopher is identical in structure, alternately eating
**  then thinking. The problem is to simulate the behaviour of the
**  philosophers while avoiding deadlock (the request by a philosopher
**  for a chopstick can never be granted) and indefinite postponement
**  (the request by a philosopher is continually denied), known
**  colloquially as "starvation."
**
**  Created by : Andrew Haigh
**  Change history
**      20-mar-2001 (Andrew Haigh)
**          created
*/
import java.io.*;
import java.lang.*;
import java.util.*;

public class Dinner
```

```
{
    protected static ThreadGroup    threadGroup;

    /*
    **  Purpose of method
    **      load the JNI libraries
    **  Input
    **  Output/Return
    **  Change History
    **      20-mar-2001 (haian02)
    **          created
    */
    static
    {
        System.loadLibrary("phil_java");
    }

    /*
    **  main
    **  Purpose of method
    **      This method controls the program,
    **      it creates the mutexes, it then creates the threads,
    **      starts the threads and then waits for them to finish
    **      It uses two JNI methods to initialize and cleanup the JNI code
    **  Input
    **      args - command line arguments
    **  Output/Return
    **      method is void
    **  Change History
    **      20-mar-2001 (Andrew Haigh)
    **          created
    */
    public static void main(String args[])
    {
        int    numPhils = 1;
        Dinner bigMeal = new Dinner();
        bigMeal.StartUp();
        // Create the threadgroup.
        threadGroup = new ThreadGroup("Philosophers");

        Phils [] phils = new Phils [numPhils];

        try
        {
            for (int j = 0; j < numPhils; j++)
            {
                System.out.println("new philosopher");
                phils [j] = new Phils(threadGroup, j);
                phils [j].start ();
            }

            for (int j = 0; j < numPhils; j++)
```

```
                {
                    phils [j].join ();
                }
            }
            catch(InterruptedException e) {}

            bigMeal.ShutDown();
        }

        public native void StartUp();
        public native void ShutDown();
    }
```

Dinner.h

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class Dinner */

#ifndef _Included_Dinner
#define _Included_Dinner
#ifdef __cplusplus
extern "C" {
#endif
/* Inaccessible static: threadGroup */
/*
 * Class:     Dinner
 * Method:    ShutDown
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_Dinner_ShutDown
  (JNIEnv *, jobject);

/*
 * Class:     Dinner
 * Method:    StartUp
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_Dinner_StartUp
  (JNIEnv *, jobject);

#ifdef __cplusplus
}
#endif
#endif
```

Phils.java

```
/*
**  Name: Phils.java - Philosophers thread
**
**  Description:
```

```
**      This is the process run by each philosopher thread
**      It calls the JNI code to execute the philosophers actions
*/
import java.io.*;
import java.lang.*;
import java.util.*;

public class Phils extends Thread
{
    String    threadName;

    /*
    **  Phils
    **  Purpose of method
    **      Adds the thread to the thread group and names the thread
    **  Input
    **      threadGroup - handle of the mutex
    **      name - unique thread identifier
    **  Output/Return
        **      method is the class constructor
        **  Change History
        **      20-mar-2001 (Andrew Haigh)
        **          created
        */
        public Phils (ThreadGroup threadGroup, int name)
        {
            // Create our server thread with a name.
            super(threadGroup, "Philosopher-" + name);

            threadName = new String ("Java-" + name);
        }

        /*
        **  run
        **  Purpose of method
        **      Call the JNI method
        **  Input
        **  Output/Return
        **      method is void
        **  Change History
        **      20-mar-2001 (Andrew Haigh)
        **          created
        */
        public void run()
        {
            System.out.println("start Jphil " + threadName);
            this.eatDinner(threadName);
            System.out.println("end Jphil " + threadName);
        }

        public native void eatDinner(String ThrdName);
    }
```

Phils.h

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class Phils */

#ifndef _Included_Phils
#define _Included_Phils
#ifdef __cplusplus
extern "C" {
#endif
/* Inaccessible static: threadInitNumber */
/* Inaccessible static: stopThreadPermission */
#undef Phils_MIN_PRIORITY
#define Phils_MIN_PRIORITY 1L
#undef Phils_NORM_PRIORITY
#define Phils_NORM_PRIORITY 5L
#undef Phils_MAX_PRIORITY
#define Phils_MAX_PRIORITY 10L
/*
 * Class:     Phils
 * Method:    eatDinner
 * Signature: (Ljava/lang/String;)V
 */
JNIEXPORT void JNICALL Java_Phils_eatDinner
  (JNIEnv *, jobject, jstring);

#ifdef __cplusplus
}
#endif
#endif
```

Phils.cpp

```
/*
** Name: phils.cpp - Dining Philosophers
**
** Description:
**  There are 10 philosophers who spend their lives either eating or
**  thinking.
**  Each philosopher has his own place at a circular table, in the
**  center of which is a large bowl of rice.  To eat rice requires
**  two chopsticks, but only 10 chopsticks are provided, one between
**  each pair of philosophers. The only chopsticks a philosopher can
**  pick up are those on his immediate right and left.
**  Each philosopher is identical in structure, alternately eating
**  then thinking. The problem is to simulate the behaviour of the
**  philosophers while avoiding deadlock (the request by a philosopher
**  for a chopstick can never be granted) and indefinite postponement
**  (the request by a philosopher is continually denied), known
**  colloquially as "starvation."
*/
```

```
/*
**  system include files
*/
# include <stdlib.h>
# include <iostream.h>
# include <sys/errno.h>

/*
**  local include files
*/
# include "Dinner.h"
# include "Phils.h"
# define  POSIX_THREADS
# include <PortableThreads.h>
# include <unistd.h>

/*
**  local variables
*/
const int  numPhilosophers = 10;     // number of philosophers
const int  numBites = 5;             // number of bites to take

// Setup the chopsticks as mutexs
PM_t       chopsticks[numPhilosophers];

// Provide a mutex to the resource 'threadCount'
int        threadCount = 0;
PM_t       threadCountMutex;

// Provide a condition variable and Boolean to allow the threads
// to have a synchronized start, and a mutex to setup the condition
PM_t       syncCondMutex;
int        syncBit;
PC_t       syncCond;

/*
**  Java_Dinner_StartUp
**  Purpose of method
**      Initialize the JNI philosophers environment
**  Input
**      *env - java environment
**      obj - java object
**  Output/Return
**      method is void
**  Change History
**      20-mar-2001 (Andrew Haigh)
**          created
*/
JNIEXPORT void JNICALL
Java_Dinner_StartUp( JNIEnv *env, jobject obj )
{
    int i, stat;
```

```
    printf ("in Phils_Startup\n");

    stat = PM_init(&threadCountMutex,NULL);
    stat = PM_lock(&threadCountMutex);
    for ( i=0; i < numPhilosophers; i++ )
    {
        cout << "Init mutex no.=" << i << " addr=" <<
            &chopsticks[i] << endl;
        stat = PM_init(&chopsticks[i%numPhilosophers],NULL);
        if(stat)
        {
            printf ("Error PT_init:1 = %d\n", stat);
            exit(1);
        }
    }

    syncBit = 0;
    stat = PM_init(&syncCondMutex, NULL);
    if(stat)
    {
        printf ("Error PM_init:2 = %d\n", stat);
        exit(1);
    }
    stat = PC_init(&syncCond, NULL);
    if(stat)
    {
        printf ("Error PC_init:3 = %d\n", stat);
        exit(1);
    }
    printf ("in Phils_Startup\n");
}

/*
**  Java_Dinner_Shutdown
**  Purpose of method
**      Release the JNI philosophers environment
**  Input
**      *env - java environment
**      obj - java object
**  Output/Return
**      method is void
**  Change History
**      20-mar-2001 (Andrew Haigh)
**          created
*/
JNIEXPORT void JNICALL
Java_Dinner_ShutDown( JNIEnv *env, jobject obj )
{
    int i, stat;
    printf ("in Phils_Shutdown\n");

    stat = PM_destroy(&threadCountMutex);
```

```
    for ( i=0; i < numPhilosophers; i++ )
    {
        stat = PM_destroy(&chopsticks[i%numPhilosophers]);
    }

    stat = PM_destroy(&syncCondMutex, NULL);
    stat = PC_destroy(&syncCond, NULL);

    printf ("in Phils_Shutdown\n");
}

/*
**  Psem
**  Purpose of method
**      This method will lock the mutex and return
**      If the mutex is not free, it will wait until it is
**  Input
**      mutex - handle of the mutex
**      seq - number to be printed in the message
**  Output/Return
**      method is void
**      return value 1 - when it succeeds
**      return value 2 - when it fails
**  Change History
**      20-mar-2001 (Andrew Haigh)
**          created
*/
void Psem (PM_t *mutex, int seq)
{
    int stat;
    // request a lock on a specific mutex
    stat = PM_lock (mutex);
    if(stat)
    {
        // if the request fails, report error to the user
        // do not try and recover
       cout << "Error PM_lock - " << seq << " = " << stat << endl;
       exit(1);
    }
}

/*

**  Psem_try
**  Purpose of method
**      This method will try to lock the mutex
**      If the mutex if free, acquire it and return
**      If the mutex is already locked it will return WAIT_TIMEOUT
**  Input
**      mutex - handle of the mutex
**      seq - number to be printed in the message
**  Output/Return
```

```
**      return 0 - when it succeeds
**      return WAIT_TIMEOUT - when it timed-out
**      exit 1 - when it fails
**   Change History
**      20-Mar-2001 (Andrew Haigh)
**          created
*/
int
Psem_try (PM_t *mutex, int seq)
{
    int stat;
    // try and get the lock on the mutex
    stat = PM_trylock (mutex);
    if(stat == EBUSY)
    {
        return 1;
    }
    else if (stat != 0)
    {
        cout << "Error PM_lock - " << seq << " = " << stat << endl;
        exit(1);
    }
    else
        return 0;
}

/*
**   Vsem
**   Purpose of method
**      Release the mutex
**   Input
**      mutex - handle of the mutex
**      seq - number to be printed in the message
**   Output/Return
**      method is void
**   Change History
**      20-mar-2001 (Andrew Haigh)
**          created
*/
void
Vsem (PM_t *mutex, int seq)
{
    // release the lock on a specific mutex
    if (PM_unlock (mutex))
    {
        // if the release fails, report error to the user
        // do not try and recover
        cout << "Error PM_unlock - " << seq << " = " << stat << endl;
        exit(1);
    }
}
```

```
/*
**  PhilosopherThread
**  Purpose of method
**      This is the process run by each philosopher thread
**      Each philosopher tries to pick up both chopsticks
**      When they have both chopsticks, they eat and return the
**      chopsticks and wait/think. This is repeated 'numBites' times
**  Input
**  Output/Return
**      method is void *
**      return value 0 - when it succeeds
**  Change History
**      20-mar-2001 (Andrew Haigh)
**          created
*/
void *PhilosopherThread()
{
    int thrid, stat;
    int nextstick;
    PT_t self = PT_get_thread_id();

    Psem (&threadCountMutex, 1);

    thrid = threadCount;
    nextstick = (thrid+1)%numPhilosophers;
    threadCount++;
    cout << "Thread " << thrid << " entry point self=" << self
        << " addr = "<< &chopsticks << endl;

    Vsem (&threadCountMutex, 1);

    cout << "Local thread id = " << thrid << endl;
// wait on a condition broadcast to start thread
Psem (&syncCondMutex, 2);

while (!syncBit)
{
    stat = pthread_cond_wait(&syncCond, &syncCondMutex);
    if(stat)
    {
        cout << "Error pthread wait1 = " << stat << endl;
        exit(1);
    }
}

cout << "Thread " << thrid <<
    " released and beginning run." << endl;

Vsem (&syncCondMutex, 2);

for ( int bites = 1; bites <= numBites; bites++ )
{
```

```
        // thinking
        cout << "Philosopher " << thrid << " thinking" << endl;
        PT_yield();

        // hungry, so picks up chopsticks
        Psem (&chopsticks[thrid], 3);

        while (Psem_try (&chopsticks[nextstick], 4) == 1)
        {
            Vsem (&chopsticks [third], 3);
            PT_yield ();
            Psem (&chopsticks [third], 3);
        }

        // eating
        cout << "Philosopher " << thrid << " eating" <<
            " bite " << bites << endl;
        PT_yield();

        // done eating
        Vsem (&chopsticks[(thrid+1)%numPhilosophers], 4);

        PT_yield();

        Vsem (&chopsticks[thrid], 3);
    }
    cout << "Thread " << thrid << " ending self=" << self << endl;
    return 0;
}

/*
**  Java_Phils_eatDinner
**  Purpose of method
**      This is the process run by each philosopher thread
**      Each philosopher tries to pick up both chopsticks
**      When they have both chopsticks, they eat and return the
**      chopsticks and wait/think. This is repeated 'numBites' times
**  Input
**      *env - java environment
**      obj - java object
**  Output/Return
**      method is void
**  Change History
**      20-mar-2001 (Andrew Haigh)
**          created
*/
JNIEXPORT void JNICALL
Java_Phils_eatDinner(JNIEnv *env, jobject obj )
{
    PT_t    phils[numPhilosophers];

    int i, stat;
```

```
    void    *ret;

    // Create the philosopher threads
    for ( i=0; i < numPhilosophers; i++ )
    {
        // create the philosopher threads
        PT_create(NULL, NULL, (void *(*)(void*))PhilosopherThread,
            NULL, &phils[i], PTHREAD_CREATE_JOINABLE, &stat);
    }
    cout << numPhilosophers << " threads created." << endl;

    Vsem (&threadCountMutex, 6);

    // now wake them all up at once
    Psem (&syncCondMutex, 7);

    syncBit = 1;
    stat = PC_broadcast(&syncCond);
    if(stat)
    {
        cout << "Error PC_broadcast = " << stat << endl;
        exit(1);

      }
      cout << "Startup condition signaled." << endl;

      Vsem (&syncCondMutex, 7);

      // wait for all threads to finish execution
      for ( i = 0; i < numPhilosophers; i++ )
      {
          stat =  PT_join( phils[i], &ret );
          if(stat)
          {
              cout << "Error PT_join" << i <<"  = " << stat << endl;
              exit(1);
          }
      }

      cout << "Normal shutdown" <<endl;

      return 1;
}
```