

# UML 基础与 Rose 建模案例

郑潮 吴建 编著

人 民 邮 电 出 版 社

## 图书在版编目 ( CIP ) 数据

UML 基础与 Rose 建模案例 / 吴建, 郑潮, 汪杰编著.

—北京: 人民邮电出版社, 2004.10

ISBN 7 - 115 - 12711 - 5

. U... . 吴... 郑... 汪... . 面向对象语言, UML—程序设计  
. TP312

中国版本图书馆 CIP 数据核字 ( 2004 ) 第 109158 号

## 内 容 提 要

本书介绍了用 UML ( 统一建模语言 ) 进行软件建模的基础知识以及 Rational Rose 工具的使用方法, 其中, 前 8 章是基础部分, 对软件工程思想、UML 的相关概念、Rational Rose 工具以及 RUP 软件过程等进行了详细的介绍; 后 3 章是案例部分, 通过 3 个综合实例, 对 UML 建模 ( 以 Rose 为实现工具 ) 的全过程进行了剖析; 最后的附录中给出了 UML 中常用的术语、标准元素和元模型, 便于读者查询。

本书是一本基础与实例紧密结合的 UML 书籍, 可以作为相关软件设计与开发人员的学习指导用书, 也可以作为高等院校相关专业的教材。

### UML 基础与 Rose 建模案例

- 
- ◆ 编 著 吴 建 郑 潮 汪 杰  
责任编辑 汤 倩
  - ◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街 14 号  
邮编 100061 电子函件 315@ptpress.com.cn  
网址 <http://www.ptpress.com.cn>  
读者热线: 010-67132692  
北京密云春雷印刷厂印刷  
新华书店总店北京发行所经销
  - ◆ 开本: 787×1092 1/16  
印张: 18.75  
字数: 449 千字 2004 年 10 月第 1 版  
印数: 1 - 5 000 册 2004 年 10 月北京第 1 次印刷

ISBN 7-115-12711-5/TP · 4263

---

定价: 29.00 元

本书如有印装质量问题, 请与本社联系 电话: ( 010 ) 67129223

# 前言

面向对象的建模语言出现在 20 世纪的七八十年代，随着编程语言的多样化以及软件产品在更多领域的应用，当时的软件工程学者开始分析与设计新的软件方法论。在这期间出现了超过 50 种的面向对象方法，对于这些不同符号体系的开发方法，软件设计人员和程序员往往很难找到完全适合他们实际开发的建模语言，而且这也妨碍了不同公司，甚至是不同项目开发组间的交流与经验共享。因此，有必要确立一款标准统一的，能被绝大部分软件开发和设计人员认可的建模语言——UML 应运而生。1997 年 11 月 17 日，UML1.1 被 OMG（对象管理组织）采纳，正式成为一款定义明确、功能强大、受到软件行业普遍认可的、可适用于广泛领域的建模语言。

现实中，一个运作管理体系合理的软件企业不仅仅是发布软件可执行的代码，它应该在开发的过程中产生用于控制、评测和交流的中间制品。例如，需求分析报告、软件结构体系、设计报告、源代码、项目计划、测试计划、测试报告和发布产品等。这些制品无论是对软件产品的开发过程还是对软件产品发布后的维护以及改良都起着至关重要的作用。在 UML 中，定义了明确的、标准的、统一的描述手段来表达这些制品，进而描绘系统的基本蓝图，如业务过程、系统中的类、数据库模式和可复用的软件构件等。

如今，UML 已经成为面向对象软件系统分析设计的必备工具，也是广大软件系统设计人员、开发人员、项目管理员、系统工程师和分析员必须掌握的基础知识。

本书有以下突出特点：

- 在讲解基础知识的过程中，结合了大量实例，不同于众多纯理论的 UML 教材的风格，更利于读者的理解和接受。
- 将 Rational Rose 的操作贯穿于整本书的理论中，并在最后给出了 3 个综合性案例，体现了本书较强的实践性，便于读者对案例进行改进，利用 Rose 工具创建自己的 UML 模型。

本书前 8 章是基础部分，主要介绍了 UML 的基础知识，准确地、完整地描述了 UML 的相关概念，便于读者掌握 UML 知识体系；后 3 章是案例部分，通过 3 个综合实例，对 UML 建模（以 Rose 为实现工具）的全过程进行了剖析；最后的附录中给出了 UML 中常用的术语、标准元素和元模型，便于读者查询。本书可以作为相关软件设计与开发人员的学习指导用书，也可以作为高等院校相关专业的教材。

本书由郑潮、吴建、汪杰编写，参与本书写作的还有厉蒋、赵斯思、李功等人。在编写过程中，我们力求精益求精，但难免存在一些不足之处，如果读者使用本书时遇到问题，可以发 E-mail 到 [tangqian@ptpress.com.cn](mailto:tangqian@ptpress.com.cn) 与我们联系。

编 者

2004 年 9 月

# 目 录

第 1 章	软件工程与 UML 概述 .....	1
1.1	软件工程概述 .....	1
1.1.1	软件工程的提出 .....	1
1.1.2	软件工程的 5 个阶段 .....	1
1.2	UML 语言概述 .....	2
1.2.1	UML 的历史 .....	2
1.2.2	UML 包含的内容 .....	3
1.2.3	UML 的定义 .....	5
1.2.4	UML 的应用领域 .....	6
第 2 章	Rational Rose 简介 .....	8
2.1	建模概论 .....	8
2.2	Rational Rose 的安装 .....	8
2.2.1	安装前的准备 .....	8
2.2.2	安装的步骤 .....	9
2.3	Rational Rose 使用 .....	11
2.3.1	Rational Rose 主界面 .....	12
2.3.2	用 Rational Rose 建模 .....	16
2.3.3	设置全局选项 .....	18
2.3.4	框图设计 .....	19
2.3.5	双向工程 .....	23
第 3 章	UML 语言初览 .....	27
3.1	概述 .....	27
3.2	UML 中的事物 .....	27
3.2.1	结构事物 ( Structure Things ) .....	27
3.2.2	行为事物 ( Behavior Things ) .....	29
3.2.3	组织事物 ( Grouping Things ) .....	30
3.2.4	辅助事物 ( Annotation Things ) .....	30
3.3	UML 中的关系 .....	30
3.3.1	关联关系 ( Association ) .....	30
3.3.2	依赖关系 ( Dependency ) .....	31
3.3.3	泛化关系 ( Generalization ) .....	31
3.3.4	实现关系 ( Realization ) .....	31
3.4	UML 中的视图 .....	32
3.5	UML 中的图 .....	33

3.5.1	静态图 .....	33
3.5.2	动态图 .....	35
第 4 章	静态视图 .....	38
4.1	概述 .....	38
4.2	类与关系 .....	38
4.2.1	类 .....	38
4.2.2	关系 .....	41
4.3	类图 .....	48
4.3.1	类图的概念和内容 .....	49
4.3.2	类图的用途 .....	49
4.3.3	类图建模技术 .....	50
4.4	对象图 .....	52
4.4.1	对象图的概念和内容 .....	52
4.4.2	对象图建模 .....	53
4.5	包图 .....	53
4.5.1	包的名字 .....	54
4.5.2	包拥有的元素 .....	54
4.5.3	包的可见性 .....	55
4.5.4	引入与输出 .....	55
4.5.5	泛化关系 .....	56
4.5.6	标准元素 .....	56
4.5.7	包建模技术 .....	56
4.6	实例——图书馆管理系统中的静态视图 .....	57
4.6.1	建立对象图步骤 .....	57
4.6.2	对象的生成 .....	58
4.6.3	用 Rose 绘制对象图 .....	58
第 5 章	用例视图 .....	63
5.1	概述 .....	63
5.2	参与者 ( Actor ) .....	63
5.3	用例 ( Use Case ) .....	64
5.3.1	用例的概念 .....	64
5.3.2	识别用例 .....	65
5.3.3	用例与事件流 .....	67
5.3.4	用例间的关系 .....	67
5.4	用例图建模技术 .....	69
5.4.1	对语境建模 .....	69
5.4.2	对需求建模 .....	70
5.5	实例——图书馆管理系统中的用例视图 .....	70
5.5.1	确定系统涉及的内容 .....	70
5.5.2	确定系统参与者 .....	71

5.5.3 确定系统用例 .....	71
5.5.4 用 Rational Rose 来绘制用例图 .....	71
<b>第 6 章 动态视图 .....</b>	<b>78</b>
6.1 时序图 ( Sequence Diagram ) .....	78
6.1.1 时序图的概念和内容 .....	78
6.1.2 时序图的用途 .....	79
6.1.3 时序图的建模技术 .....	80
6.2 协作图 ( Collaboration Diagram ) .....	81
6.2.1 协作图的概念和内容 .....	81
6.2.2 协作图的用途 .....	82
6.2.3 协作图的建模技术 .....	82
6.2.4 协作图与时序图的互换 .....	83
6.3 状态图 ( Statechart Diagram ) .....	84
6.3.1 状态图的概念和内容 .....	84
6.3.2 状态图的用途 .....	89
6.3.3 状态图的建模技术 .....	89
6.4 活动图 ( Activity Diagram ) .....	91
6.4.1 活动图的概念和内容 .....	91
6.4.2 活动图的用途 .....	97
6.4.3 活动图的建模技术 .....	97
6.5 实例——图书馆管理系统的动态视图 .....	99
6.5.1 各种动态视图的区别 .....	99
6.5.2 用 Rose 绘制状态图 .....	100
6.5.3 用 Rose 绘制活动图 .....	103
6.5.4 用 Rose 绘制时序图 .....	108
6.5.5 用 Rose 绘制协作图 .....	111
<b>第 7 章 UML 实现与部署 .....</b>	<b>114</b>
7.1 组件图 ( Component Diagrams ) .....	114
7.1.1 组件图的概念和内容 .....	114
7.1.2 组件 .....	114
7.1.3 接口 .....	115
7.1.4 关系 .....	116
7.1.5 补充图标 .....	117
7.1.6 组件图建模技术 .....	118
7.2 配置图 ( Deployment Diagrams ) .....	120
7.2.1 配置图的概念和内容 .....	120
7.2.2 节点 .....	120
7.2.3 组件 .....	121
7.2.4 关系 .....	122
7.2.5 配置图建模技术 .....	122

7.3	实例——图书馆管理系统的组件图与配置图 .....	124
7.3.1	绘制组件图与配置图的步骤 .....	124
7.3.2	用 Rose 绘制组件图 .....	125
7.3.3	用 Rose 绘制配置图 .....	127
第 8 章	UML 与统一开发过程 .....	130
8.1	软件过程历史概述 .....	130
8.1.1	软件开发过程简介 .....	130
8.1.2	当前流行的软件过程 .....	130
8.2	RUP 简介 .....	131
8.2.1	什么是 RUP 过程 .....	131
8.2.2	RUP 的特点 .....	131
8.2.3	RUP 的十大要素 .....	134
8.3	统一开发过程核心工作流 .....	138
8.3.1	需求捕获工作流 .....	139
8.3.2	分析工作流 .....	143
8.3.3	设计工作流 .....	146
8.3.4	实现工作流 .....	150
8.3.5	测试工作流 .....	154
8.4	RUP 统一过程案例 .....	159
8.4.1	简介 .....	159
8.4.2	要求 .....	160
8.4.3	创意设计大纲 .....	161
8.4.4	导航图 .....	161
8.4.5	创意设计比选方案 .....	161
8.4.6	Web 设计元素 .....	162
8.4.7	初始 Web 用户接口原型 .....	162
8.4.8	UI 指南 .....	163
8.4.9	Web 用户接口总体原型 .....	163
8.4.10	总体导航图 .....	163
第 9 章	图书馆管理系统 .....	165
9.1	需求分析 .....	165
9.1.1	系统总体功能需求 .....	165
9.1.2	基本数据维护功能需求 .....	166
9.1.3	基本业务功能需求 .....	167
9.1.4	数据库维护功能 .....	169
9.1.5	查询功能需求 .....	169
9.1.6	安全使用管理功能需求 .....	170
9.1.7	帮助功能需求 .....	170
9.2	UML 系统建模 .....	171
9.2.1	用例的建立 .....	171

9.2.2	时序图与协作图的生成 .....	174
9.2.3	状态图的生成 .....	175
9.2.4	活动图的生成 .....	176
9.3	类与接口 .....	177
9.3.1	类图的生成 .....	177
9.3.2	包图的生成 .....	182
9.3.3	组件图的生成 .....	183
9.4	系统部署 .....	185
第 10 章	ATM 自动取款机系统 .....	186
10.1	系统概述 .....	186
10.2	需求分析 .....	186
10.2.1	系统总体功能需求 .....	187
10.2.2	读卡机模块需求 .....	188
10.2.3	键盘输入模块需求 .....	189
10.2.4	IC 认证模块需求 .....	189
10.2.5	显示模块需求 .....	190
10.2.6	吐钱机模块需求 .....	190
10.2.7	打印报表模块需求 .....	191
10.2.8	监视模块需求 .....	191
10.2.9	数据库模块需求 .....	192
10.3	系统用例模型 .....	192
10.3.1	角色的确定 .....	192
10.3.2	创建用例 .....	193
10.3.3	创建角色用例关系图 .....	194
10.4	系统动态模型 .....	196
10.4.1	创建活动图 .....	196
10.4.2	时序图 .....	197
10.4.3	协作图 .....	199
10.5	创建系统包图 .....	199
10.5.1	ATM 系统包图 .....	200
10.5.2	Hardware 包内的类 .....	200
10.5.3	Logic 包内的类 .....	201
10.6	系统类模型 .....	201
10.6.1	Logical 视图 .....	201
10.6.2	类图 .....	202
10.6.3	状态图 .....	204
10.7	系统部署 .....	205
10.7.1	组件图 .....	205
10.7.2	配置图 .....	207



第 11 章 大型仓库信息管理系统开发 .....	209
11.1 系统概述.....	209
11.2 需求分析.....	209
11.2.1 系统总体功能需求 .....	209
11.2.2 用户登录 .....	210
11.2.3 仓库管理 .....	211
11.2.4 业务查询 .....	214
11.2.5 系统设置 .....	216
11.3 系统用例模型.....	217
11.3.1 角色的确定 .....	217
11.3.2 创建用例 .....	218
11.3.3 创建角色用例关系图 .....	219
11.4 系统动态模型.....	222
11.4.1 活动图 .....	222
11.4.2 时序图 .....	223
11.4.3 协作图 .....	225
11.5 创建系统包图.....	227
11.5.1 仓库管理系统包图 .....	227
11.5.2 人员信息 (peopleinformatoin) 包内的类 .....	227
11.5.3 事务包 (business) 包内的类 .....	228
11.5.4 接口包 (interfaces) 包内的类 .....	228
11.6 系统类模型.....	229
11.6.1 Logical 视图 .....	229
11.6.2 类图 .....	230
11.7 系统部署.....	232
11.7.1 组件图 .....	233
11.7.2 配置图 .....	234
附录 A.....	236
A.1 术语.....	236
A.1.1 范围 .....	236
A.1.2 部分术语 .....	236
A.2 标准元素.....	277
A.3 元模型.....	285
A.3.1 简介 .....	285
A.3.2 背景 .....	285
A.3.3 元元模型 .....	287
参考文献.....	288

# 第 1 章 软件工程与 UML 概述

本章对软件工程和 UML 进行简要的介绍，共分 3 小节，每小节介绍一个主题：软件工程概述、UML 语言概述和 UML 的现状与未来。通过对本章的阅读，读者可以对什么是软件工程、什么是 UML，以及为什么要使用 UML 等问题有一个清楚的认识。

## 1.1 软件工程概述

### 1.1.1 软件工程的提出

在 20 世纪 60 年代计算机技术发展初期，程序设计是少数聪明人干的事。他们的智力与技能超群，编写的程序既能控制计算机，又不易被别人理解和使用。那个时期，人们随心所欲的编程，结果产生了一系列问题：程序质量低下、错误频出、进度延误、费用剧增……这些问题导致了“软件危机”。

在 1968 年，一群程序员、计算机科学家与工业界人士聚集在一起共商对策。通过借鉴传统工业的成功作法，他们主张通过工程化的方法开发软件来解决软件危机，并冠以“软件工程”这一术语。30 余年来，尽管软件的一些毛病仍然无法根治，但软件的发展速度却超过了任何传统工业，并未出现真正的软件危机，这的确是前人的先见之明。如今软件工程成了一门学科。

### 1.1.2 软件工程的 5 个阶段

软件开发是一套关于软件开发各阶段的定义、任务和作用的，建立在理论上的一门工程学科。它对解决软件危机，指导人们利用科学和有效的方法来开发软件，提高及保证软件开发的效率和质量起到了一定的作用。

经典的软件工程思想将软件开发分成以下 5 个阶段：需求分析（Requirements Capture）阶段、系统分析与设计（System Analysis and Design）阶段、系统实现（Implementation）阶段、测试（Testing）阶段和维护（Maintenance）阶段。

#### （1）需求分析（Requirements Capture）阶段

需求分析阶段是通常所说的开始阶段，但实际上，真正意义上的开始阶段要做的是选择合适的项目——立项阶段。其实，软件工程中的许多关于思想的描述都是通俗易懂的。立项阶段，顾名思义，就是从若干个可以选择的项目中选择一个最适合自己的项目的阶段。这个选择的过程是至关重要的，因为它将直接决定整个软件开发过程的成败。通常情况下，要考虑几个主要的因素：经济因素（经济成本、受益等）、技术因素（可行性、技术成本等）和管理因素（人员管理、资金运作等）。

在立项之后,真正进入了软件开发阶段(当然,这里所说的是广义的软件开发,狭义的软件开发生通常指的是编码)。需求分析是整个开发过程的基础,也直接影响着后面的几个阶段的进展。纵观软件开发从早期纯粹的程序设计到软件工程思想的萌发产生和发展的全过程,不难发现,需求分析的工作量在不断增加,其地位也随之不断提升。这一点可以从需求分析在整个开发过程中所占的比例(无论是时间、人力,还是资金方面)地不断提高上看出。

#### (2) 系统分析与设计(System Analysis and Design)阶段

系统分析与设计包括分析和设计两个阶段,而这两个阶段是相辅相成、不可分割的。通常情况下,这一阶段是在系统分析员的领导下完成的,系统分析员不仅要有深厚的计算机硬件与软件的专业知识,还要对相关业务有一定的了解。系统分析通常是与需求分析同时进行,而系统设计一般是在系统分析之后进行的。

#### (3) 实现(Implementation)阶段

系统实现阶段也就是通常所说的编码(Coding)阶段,在软件工程思想出现之前,这基本上就是软件开发的全部内容

#### (4) 测试(Testing)阶段

测试阶段的主要任务是通过各种测试思想、方法和工具,使软件的 Bug 降到最低。微软(Microsoft)宣称他们采用零 Bug 发布的思想确保软件的质量,也就是说只有当测试阶段达到没有 Bug 时他们才将产品发布。测试是一项很复杂的工程。

#### (5) 维护(Maintenance)阶段

在软件工程思想出现之前,这一阶段是令所有与之相关的角色(包括客户和发方)头疼的。可以说,软件工程思想很大程度上是为了解决软件维护的问题而提出的。因为,在软件工程的3大目的——软件的可维护性、软件的可复用性和软件开发的自动化中,可维护性就是其中之一,而且软件的可维护性是复用性和开发自动化的基础。在软件工程思想得到迅速发展的今天,虽然软件的可维护性有了很大的提高,但目前软件开发中所面临的最大的问题仍是维护问题。每年都有许多软件公司因为无法承担对其产品的高昂的维护成本而宣布破产。

值得注意的是,软件工程主要讲述软件开发的道理,基本上是软件实践者的成功经验和失败教训的总结。软件工程的观念、方法、策略和规范都是朴实无华的,一般人都能领会,关键在于运用。不可以把软件工程方法看成是诸葛亮的锦囊妙计——在出了问题后才打开看看,而应该事先掌握,预料将要出现的问题,控制每个实践环节,防患于未然。

## 1.2 UML 语言概述

### 1.2.1 UML 的历史

面向对象的分析与设计(OOA&D)方法的发展在20世纪80年代末至90年代中出现了一个小高潮,UML是这个高潮的产物。它不仅统一了Booch、Rumbaugh和Jacobson的表示方法,而且对其做了进一步的发展,并最终统一为大众所接受的标准建模语言。

公认的面向对象建模语言出现于20世纪70年代中期。从1989年~1994年,其数量从不到10种增加到了50多种。在众多的建模语言中,语言的创造者努力宣传自己的产品,并在实践中不断完善。但是,使用面向对象方法的用户并不了解不同建模语言的优缺点及相互之间的差异,因而很难根据应用特点选择合适的建模语言,于是爆发了一场“方法大战”。20世纪90年代中期,一批新方法出现了,其中最引人注目的是Booch 1993、OOSE和OMT-2等。

Booch是面向对象方法最早的倡导者之一,他提出了面向对象软件工程的概念。1991年,他将以前面向Ada的工作扩展到整个面向对象设计领域。Booch 1993比较适合于系统的设计和构造。Rumbaugh等人提出了面向对象的建模技术(OMT)方法,采用了面向对象的概念,并引入各种独立于语言的表示符。这种方法用对象模型、动态模型、功能模型和用例模型,共同完成对整个系统的建模,所定义的概念和符号可用于软件开发的分析、设计和实现的全过程,软件开发人员不必在开发过程的不同阶段进行概念和符号的转换。OMT-2特别适用于分析和描述以数据为中心的信息系统。Jacobson于1994年提出了OOSE方法,其最大特点是面向用例(Use Case),并在用例的描述中引入了外部角色的概念。用例的概念是精确描述需求的重要武器,但用例贯穿于整个开发过程,包括对系统的测试和验证。OOSE比较适合支持商业工程和需求分析。此外,还有Coad/Yourdon方法,即著名的OOA/OOD,它是最早的面向对象的分析和设计方法之一,该方法简单、易学,适合于面向对象技术的初学者使用,但由于该方法在处理能力方面的局限,目前已很少使用。

概括起来,首先,面对众多的建模语言,用户由于没有能力区别不同语言之间的差别,因此很难找到一种比较适合其应用特点的语言;其次,众多的建模语言实际上各有千秋;第三,虽然不同的建模语言大多雷同,但仍存在某些细微的差别,极大地妨碍了用户之间的交流。因此在客观上,有必要在精心比较不同的建模语言优缺点及总结面向对象技术应用实践的基础上,组织联合设计小组,根据应用需求,取其精华,去其糟粕,求同存异,统一建模语言。

1994年10月,Grady Booch和Jim Rumbaugh首先将Booch 93和OMT-2统一起来,并于1995年10月发布了第一个公开版本,称之为统一方法UM 0.8(Unified Method)。1995年秋,OOSE的创始人Jacobson加盟到这一工作中。经过Booch、Rumbaugh和Jacobson 3人的共同努力,于1996年6月和10月分别发布了两个新的版本,即UML 0.9和UML 0.91,并将UM重新命名为UML(Unified Modeling Language)。UML的开发者倡议并成立了UML成员协会,以完善、加强和促进UML的定义工作。当时的成员有DEC、HP、I-Logix、Itellicorp、IBM、ICON Computing、MCI Systemhouse、Microsoft、Oracle、Rational Software、TI以及Unisys。UML成员协会对UML 1.0及UML 1.1的定义和发布起了重要的促进作用。

### 1.2.2 UML 包含的内容

首先,UML融合了Booch、OMT和OOSE方法中的基本概念,而且这些基本概念与其他面向对象技术中的基本概念大多相同,因而,UML必然成为这些方法以及其他方法的使用者乐于采用的一种简单一致的建模语言。其次,UML不是上述方法的简单汇合,而是在这些方法的基础上广泛征求意见,集众家之长,几经修改而完成的,UML扩展了现有方法的应用范围。第三,UML是标准的建模语言,而不是标准的开发过程。

作为一种建模语言,UML的定义包括UML语义和UML表示法两个部分。

### (1) UML 语义

描述基于 UML 的精确元模型定义。元模型为 UML 的所有元素在语法和语义上提供了简单、一致和通用的定义性说明,使开发者能在语义上取得一致,消除了因人而异的表达方法所造成的影响。此外 UML 还支持对元模型的扩展定义。

### (2) UML 表示法

定义 UML 符号的表示法,为开发者或开发工具使用这些图形符号和文本语法为系统建模提供了标准。这些图形符号和文字所表达的是应用级的模型,在语义上它是 UML 元模型的实例。

标准建模语言 UML 的重要内容可以由下列 5 类图来定义。

第 1 类是用例图( Use Case Diagram ),从用户角度描述系统功能,并指出各功能的操作者。

第 2 类是静态图 ( Static Diagram ),包括类图、对象图和包图。其中类图描述系统中类的静态结构。不仅定义系统中的类,表示类之间的联系(如关联、依赖和聚合等),也包括类的内部结构(类的属性和操作)。类图描述的是一种静态关系,在系统的整个生命周期中都是有效的。对象图是类图的实例,使用与类图几乎完全相同的标识。它们的不同点在于对象图显示类的多个对象实例,而不是实际的类,一个对象图是类图的一个实例。由于对象存在生命周期,因此对象图只能在系统某一段时间内存在。包由包或类组成,表示包与包之间的关系。包图用于描述系统的分层结构。

第 3 类是行为图 ( Behavior Diagram ),描述系统的动态模型和组成对象间的交互关系,包括状态图和活动图。其中状态图描述类的对象所有可能的状态以及事件发生时状态的转移条件。通常,状态图是对类图的补充。在实际上并不需要为所有的类画状态图,只需为那些有多个状态且其行为受外界环境的影响并且发生改变的类画状态图。而活动图描述满足用例要求所要进行的活动以及活动间的约束关系,有利于识别并行活动。

第 4 类是交互图 ( Interactive Diagram ),描述对象间的交互关系,包括时序图和合作图。其中,时序图显示对象之间的动态合作关系,它强调对象之间消息发送的顺序,同时显示对象之间的交互;合作图描述对象间的协作关系,合作图跟时序图相似,显示对象间的动态合作关系。除显示信息交换外,合作图还显示对象以及它们之间的关系。如果强调时间和顺序,则使用时序图;如果强调上下级关系,则选择合作图。这两种图合称为交互图。

第 5 类是实现图 ( Implementation Diagram ),包括组件图和配置图。其中组件图描述代码部件的物理结构及各组件之间的依赖关系。一个组件可能是一个资源代码组件、一个二进制组件或一个可执行组件。它包含逻辑类或实现类的有关信息。组件图有助于分析和理解部件之间的相互影响程度。配置图定义系统中软硬件的物理体系结构。它可以显示实际的计算机和设备(用节点表示)以及它们之间的连接关系,也可显示连接的类型及部件之间的依赖性。在节点内部,放置可执行部件和对象,以显示节点与可执行软件单元的对应关系。

从应用的角度看,当采用面向对象技术设计系统时,首先是描述需求;其次根据需求建立系统的静态模型,以构造系统的结构;第 3 步是描述系统的行为。其中在第 1 步与第 2 步中所建立的模型都是静态的,包括用例图、类图(包含包)、对象图、组件图和配置图等 5 个图形,是标准建模语言 UML 的静态建模机制。其中第 3 步中所建立的模型或者可以执行,或者表示执行时的时序状态或交互关系。它包括状态图、活动图、时序图和合作图等 4 个图形,是标准建模语言 UML 的动态建模机制。因此,标准建模语言 UML 的主要内容也可以归纳为静态建

模机制和动态建模机制两大类。

### 1.2.3 UML 的定义

UML ( Unified Modeling Language , 统一建模语言 ), 是一种面向对象的建模语言。它的主要作用是帮助用户对软件系统进行面向对象的描述和建模(建模是通过将用户的业务需求映射为代码,保证代码满足这些需求,并能方便地回溯需求的过程),它可以描述这个软件开发过程从需求分析直到实现和测试的全过程。UML 通过建立各种类、类之间的关联、类/对象怎样相互配合实现系统的动态行为等成分(这些都称为模型元素)来组建整个模型。UML 提供了各种图形,比如用例图、类图、时序图、协作图和状态图等,来把这些模型元素及其关系可视化,让人们可以清楚容易地理解模型。可以从多个视角来考察模型,从而更加全面地了解模型,这样同一个模型元素可能会出现在多个图中,对应多个图形元素。

#### 1. UML 的组成

UML 由视图 ( View )、图 ( Diagram )、模型元素 ( Model Element ) 和通用机制 ( General Mechanism ) 等几个部分组成。

视图 ( View ) 是表达系统的某一方面特征的 UML 建模元素的子集,由多个图构成,是在某一个抽象层上,对系统的抽象表示。

图 ( Diagram ) 是模型元素集的图形表示,通常是由弧 ( 关系 ) 和顶点 ( 其他模型元素 ) 相互连接构成的。

模型元素 ( Model Element ) 代表面向对象中的类、对象、消息和关系等概念,是构成图的最基本的常用概念。

通用机制 ( General Mechanism ) 用于表示其他信息,比如注释、模型元素的语义等。另外,UML 还提供扩展机制 ( Extension Mechanism ),使 UML 语言能够适应一个特殊的方法 ( 或过程 ),或扩充至一个组织或用户。

UML 是用来描述模型的,用模型来描述系统的结构或静态特征,以及行为或动态特征。从不同的视角为系统构架建模,形成系统的不同视图。

(1) 用例视图 ( Use Case View ),强调从用户的角度看到的或需要的系统功能,是被称为参与者的外部用户所能观察到的系统功能的模型图。

(2) 逻辑视图 ( Logical View ),展现系统的静态或结构组成及特征,也称为结构模型视图 ( Structural Model View ) 或静态视图 ( Static View )。

(3) 并发视图 ( Concurrent View ),体现了系统的动态或行为特征,也称为行为模型视图 ( Behavioral Model View ) 或动态视图 ( Dynamic View )。

(4) 组件视图 ( Component View ),体现了系统实现的结构和行为特征,也称为实现模型视图 ( Implementation Model View )。

(5) 配置视图 ( Deployment View ),体现了系统实现环境的结构和行为特征,也称为环境模型视图 ( Environment Model View ) 或物理视图 ( Physical View )。

视图是由图组成的,UML 提供 9 种不同的图:

(1) 用例图 ( Use Case Diagram ),描述系统功能;

(2) 类图 ( Class Diagram ),描述系统的静态结构;

- (3) 对象图 (Object Diagram), 描述系统在某个时刻的静态结构;
- (4) 时序图 (Sequence Diagram), 按时间顺序描述系统元素间的交互;
- (5) 协作图 (Collaboration Diagram), 按照时间和空间顺序描述系统元素间的交互和它们之间的关系;
- (6) 状态图 (State Diagram), 描述了系统元素的状态条件和响应;
- (7) 活动图 (Activity Diagram), 描述了系统元素的活动;
- (8) 组件图 (Component Diagram), 描述了实现系统的元素的组织;
- (9) 配置图 (Deployment Diagram), 描述了环境元素的配置, 并把实现系统的元素映射到配置上。

根据它们在不同架构视图的应用, 可以把 9 种图分成:

- (1) 用户模型视图, 用例图;
- (2) 结构模型视图, 类图和对象图;
- (3) 行为模型视图, 时序图、协作图、状态图和活动图 (动态图);
- (4) 实现模型视图, 组件图;
- (5) 环境模型视图, 配置图。

## 2. UML 的建模机制

UML 有两套建模机制: 静态建模机制和动态建模机制。静态建模机制包括用例图、类图、对象图、包、组件图和配置图。动态建模机制包括消息、状态图、时序图、协作图、活动图。

对于本节中的诸多概念, 读者暂时只需了解即可, 在后面的章节中将会结合实例进行详细的介绍。

### 1.2.4 UML 的应用领域

UML 的目标是以面向对象图的方式来描述任何类型的系统。其中最常用的是建立软件系统的模型, 但它同样可以用于描述非软件领域的系统, 如机械系统、企业机构或业务过程, 以及处理复杂数据的信息系统、具有实时要求的工业系统或工业过程等。

总之, UML 是一个通用的标准建模语言, 可以对任何具有静态结构和动态行为的系统进行建模。此外, UML 适用于系统开发过程中从需求规格描述到系统完成后测试的不同阶段。在需求分析阶段, 可以用用例来捕获用户需求。通过用例建模, 描述对系统感兴趣的外部角色及其对系统 (用例) 的功能要求。分析阶段主要关心问题域中的主要概念 (如抽象、类和对象等) 和机制, 需要识别这些类以及它们相互间的关系, 并用 UML 类图来描述。为实现用例, 类之间需要协作, 这可以用 UML 动态模型来描述。在分析阶段, 只对问题域的对象 (现实世界的概念) 建模, 而不考虑定义软件系统中技术细节的类 (如处理用户接口、数据库、通信和并行性等问题的类)。这些技术细节将在设计阶段引入, 因此设计阶段为构造阶段提供更详细的规格说明。

编程 (构造) 是一个独立的阶段, 其任务是用面向对象编程语言将来自设计阶段的类转换成实际的代码。在用 UML 建立分析和设计模型时, 应尽量避免考虑把模型转换成某种特定的编程语言。因为在早期阶段, 模型仅仅是理解和分析系统结构的工具, 过早考虑编码问题十分不利于建立简单、正确的模型。

UML 模型还可作为测试阶段的依据。系统通常需要经过单元测试、集成测试、系统测试和验收测试。不同的测试小组使用不同的 UML 图作为测试依据：单元测试使用类图和类规格说明；集成测试使用部件图和协作图；系统测试使用用例图来验证系统的行为；验收测试由用户进行，以验证系统测试的结果是否满足在分析阶段确定的需求。



## 第 2 章 Rational Rose 简介

### 2.1 建模概论

无论何种复杂程度的工程项目,设计都是从建模开始的,设计者通过创建模型和设计蓝图来描述系统的结构。比如,电子工程设计人员使用惯用标记和示意图进行复杂系统的最初设计,会计总是在表格上规划公司的财务蓝图,而行政管理人员则常使用组织流图这种可视化的方式来描述所管理的部门。

建模意义重大,“分而治之”是一个古老而有效的概念。可以想象,把特别复杂而困难的问题细化分解之后,一次只是设法解决其中一个,事情就变得容易多了。模型的作用就是使复杂的信息关联简单易懂,它使使用者容易洞察复杂的原始数据背后的规律,并能有效地将系统需求映射到软件结构上去。

Rose 是美国的 Rational 公司的面向对象建模工具,利用这个工具,可以建立用 UML 描述的软件系统的模型,而且可以自动生成和维护 C++、Java、VB 和 Oracle 等语言和系统的代码。Rational Rose 包括了统一建模语言(UML),OOSE 及 OMT。其中统一建模语言(UML)由 Rational 公司 3 位世界级面向对象技术专家 Grady Booch、Ivar Jacobson 和 Jim Rumbaugh 通过对早期面向对象研究和设计方法的进一步扩展而得来的,它为可视化建模软件奠定了坚实的理论基础。

### 2.2 Rational Rose 的安装

#### 2.2.1 安装前的准备

(1)安装 Rose 需要 Windows 2000/Windows XP 及其以上版本,并且如果是 Windows 2000 则要确认已经安装了 Server Pack 2。

(2)安装 Rose,必须先得到 Rose 的安装包。建议购买 Rational 公司的正版软件,Rational 现已被 IBM 收购,读者可以下载 Rose 的试用版。

目前 Rational Rose 的最新版本为 2003,接下来的章节将介绍 Rose 2003 的安装和使用。如果读者现在使用的是 Rose 2002 也没关系,基本的操作是一样的。

## 2.2.2 安装的步骤

(1) 双击启动 Rational Rose 2003 的安装程序，进入安装向导界面，如图 2-1 所示。

(2) 单击“下一步”按钮，进入如图 2-2 所示对话框，让用户选择要安装的产品。这里选择第 2 项即“Rational Rose Enterprise Edition”。

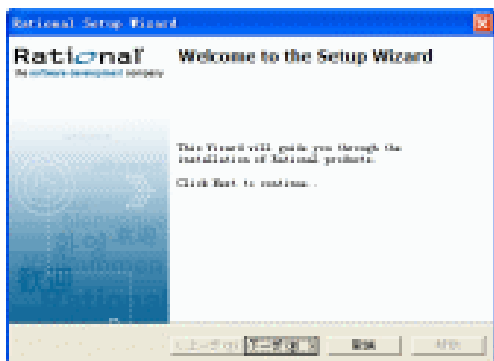


图 2-1 Rational Rose 2003 安装向导



图 2-2 选择安装的产品

(3) 单击“下一步”按钮，进入如图 2-3 所示界面。在图 2-3 中选择“Desktop installation from CD image”选项，表示创建一个本地的应用程序而不是网络的。

(4) 继续单击“下一步”按钮，进入安装向导界面，如图 2-4 所示。

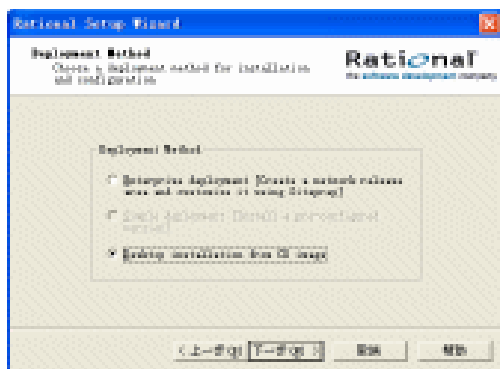


图 2-3 选择安装方式



图 2-4 安装向导说明

(5) 单击“Next”按钮，进入产品声明界面，如图 2-5 所示。

(6) 继续单击“Next”按钮，进入协议许可界面，如图 2-6 所示。选中“I accept the terms in the license agreement”单选按钮即可。

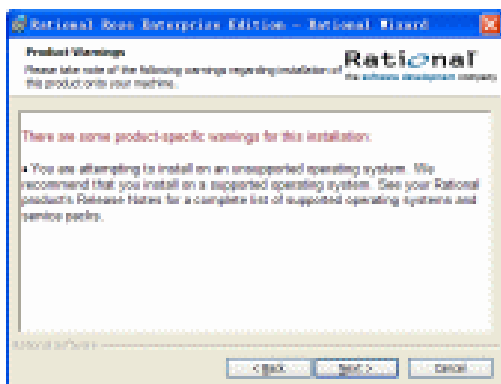


图 2-5 产品声明

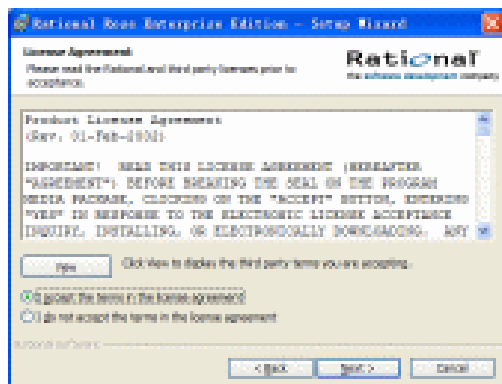


图 2-6 版权声明

(7) 继续单击“Next”按钮，进入安装路径设置界面，如图 2-7 所示。可以单击“Change”按钮选择安装路径。

注意：完全安装需要 1.5GB 左右的磁盘空间。

(8) 路径设置完毕后，单击“Next”按钮，进入自定义安装设置界面，用户可以根据实际需要进行选择，如图 2-8 所示。

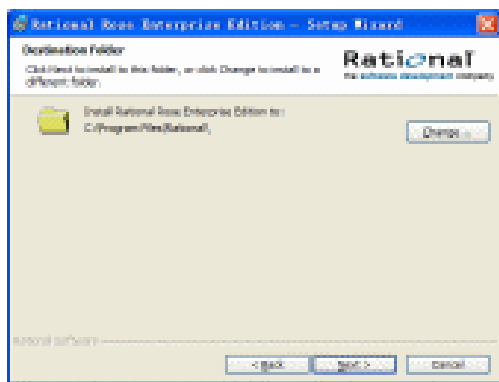


图 2-7 设置安装路径

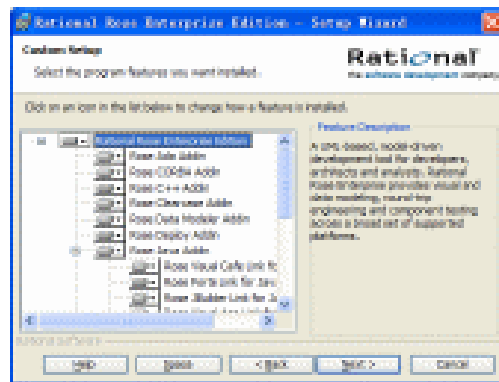


图 2-8 自定义安装选项

(9) 继续单击“Next”按钮，进入开始安装界面，如图 2-9 所示。

(10) 单击“Install”按钮，开始拷贝文件，如图 2-10 所示。

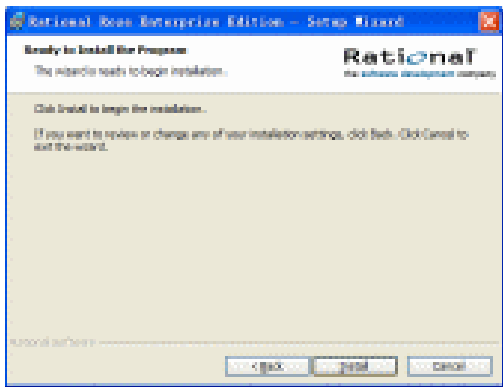


图 2-9 开始安装界面

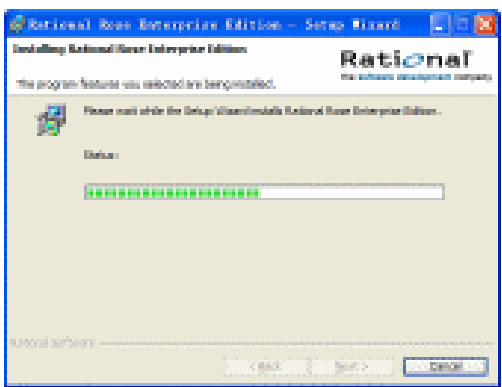


图 2-10 拷贝文件

(11) 系统安装完毕，完成界面如图 2-11 所示。

(12) 单击“Finish”按钮后，会弹出注册对话框，要求用户对软件进行注册，如图 2-12 所示。

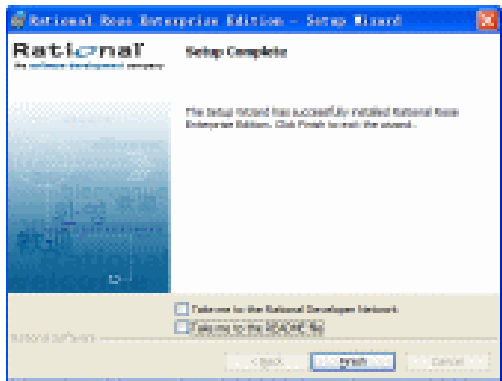


图 2-11 安装完成界面

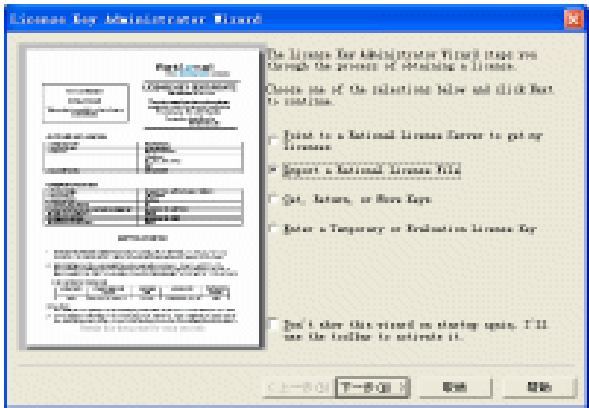


图 2-12 软件注册

注意：可以有多种方法进行注册，建议购买正版软件，如果是试用版本，则不用注册。

## 2.3 Rational Rose 使用

Rational Rose 是菜单驱动式的应用程序，可以通过工具栏使用其常用工具。它的界面分为 3 个部分——Browser 窗口、Diagram 窗口和 Document 窗口。Browser 窗口用来浏览、创建、

删除和修改模型中的模型元素；Diagram 窗口用来显示和创作模型的各种图；而 Document 窗口则用来显示和书写各个模型元素的文档注释。

### 2.3.1 Rational Rose 主界面

启动 Rational Rose 2003，出现如图 2-13 所示的启动画面。



图 2-13 启动界面

启动界面消失后，进入到 Rational Rose 2003 的主界面，首先弹出如图 2-14 所示的对话框。这个对话框用来设置本次启动的初始动作，分为 New（新建模型）、Existing（打开现有模型）和 Recent（最近打开模型）3 个选项卡。

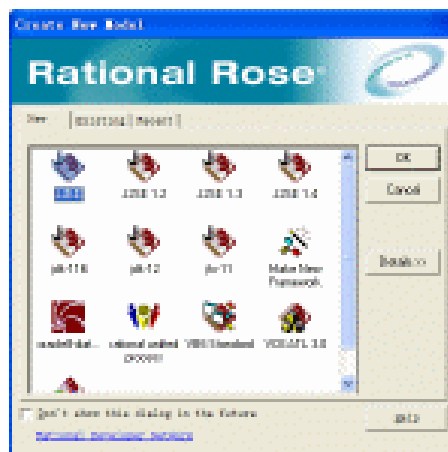


图 2-14 “新建模型”选项卡

第 1 个选项卡是 New，用来选择新建模型时采用的模板，目前 2003 版所支持的模板有 J2EE（Java 2 Enterprise Edition，Java 第二版规范企业级版），J2SE（Java 2 Standard Edition，Java 第二版规范标准版）的 1.2、1.3 和 1.4 版，JDK（Java Development Kit，Java 开发工具包）的 1.16 版和 1.2 版，JFC（Java Fundamental Classes，Java 基础类库）的 1.1 版，Oracle8-datatype（Oracle8 的数据类型），Rational Unified Process（即 RUP，Rational 统一过程），VB6 Standard（VB6 标准程序），VC6 ATL（VC6 Active Templates Library，VC6 活动模板库）3.0 版，以及 VC6 MFC（VC6 Microsoft Fundamental Classes，VC6 基础类库）的 3.0 版。

第 2 个选项卡是 Existing，如图 2-15 所示。  
第 3 个选项卡是 Recent，如图 2-16 所示。

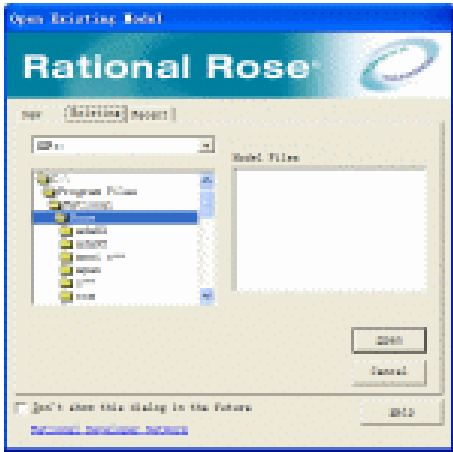


图 2-15 “打开现有模型”选项卡

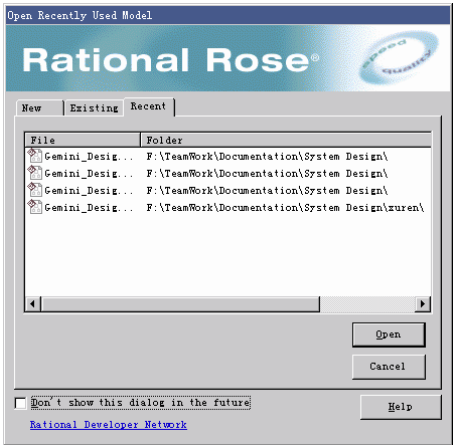


图 2-16 “最近打开模型”选项卡

由于暂时不需要任何模板，只需要新建一个空白的模型，所以直接单击“Cancel（取消）”按钮，这样，就显示出了 Rational Rose 的主界面，如图 2-17 所示。

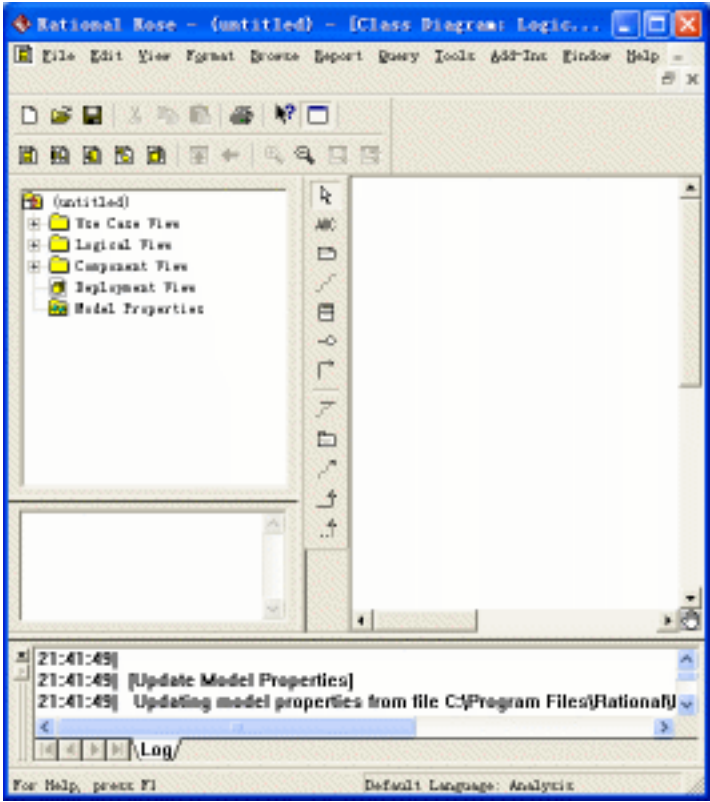


图 2-17 Rational Rose 的主界面

可以看到，Rational Rose 的主界面由标题栏、菜单栏、工具栏、工作区和状态栏组成。默认的工作区又分为 3 个部分，左方是树形视图和文档，右方是主要的编辑区，而下方则是动作记录。以下对各组成部分做简单说明。

标题栏用来显示当前正在编辑的模型名称，由于此时的空模型刚刚新建，还没有被保存，故而标题栏上显示为 untitled，如图 2-18 所示。



图 2-18 标题栏

菜单栏包含了所有可以进行的操作，有 File（文件）、Edit（编辑）、View（视图）、Format（格式）、Browse（浏览）、Report（报告）、Query（查询）、Tools（工具）、Add-Ins（插件）、Window（窗口）和 Help（帮助），如图 2-19 所示。



图 2-19 菜单栏

工具栏包含了最常用的一些操作，当然，用户也可以自行添加或删除工具栏中的按钮。默认的情况下，工具栏从左到右依次分为 7 组，如图 2-20 所示。



图 2-20 默认工具栏

第 1 组有 3 个按钮，分别对应 File（文件）菜单中的 New（新建模型）、Open（打开现有模型）和 Save（保存模型）菜单项，如图 2-21 所示。

第 2 组也有 3 个按钮，分别对应 Edit（编辑）菜单中的 Cut（剪切）、Copy（复制）和 Paste（粘贴）3 个菜单项，如图 2-22 所示。



图 2-21 工具栏第 1 组



图 2-22 工具栏第 2 组

第 3 组只有一个按钮，对应的是 Print（打印）功能，如图 2-23 所示。

第 4 组有两个按钮，分别是 Help（帮助）和 View Doc（显示文档），如图 2-24 所示。



图 2-23 工具栏第 3 组

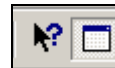


图 2-24 工具栏第 4 组

第 5 组是共有 5 个按钮，分别代表 Browse Class Diagram（浏览类图）、Browse Interaction Diagram（浏览交互图）、Browse Component Diagram（浏览组件图）、Browse State Machine

Diagram (浏览状态机图) 和 Browse Deployment Diagram (浏览实施图), 从不同的方面描述模型, 如图 2-25 所示。



图 2-25 工具栏第 5 组

第 6 组是浏览工具栏, 使用上面的按钮可以在图之间切换。第 1 个按钮是 Browse Parent (浏览父图), 即浏览在层次结构上高于当前图一层的图, 而第 2 个是 Browse Previous Diagram (浏览前一张图), 如图 2-26 所示。

最后一组是显示工具栏, 使用此工具栏的按钮可以使显示的图形按需要放大或缩小。从左到右分别是 Zoom In (放大)、Zoom Out (缩小)、Fit In Window (和显示窗口一样大), 以及 Undo Fit In Window (恢复原来大小), 如图 2-27 所示。



图 2-26 工具栏第 6 组



图 2-27 工具栏第 7 组

工作区分成 3 部分: 左边的部分是树形视图和文档区, 其中上面是当前项目模型的树形视图, 下面是对应的文档区, 每选中树形视图的某个对象 (类、关系或图), 下面的文档区就会显示其对应的文档名称, 如 2-28 所示。

主要编辑区如图 2-29 所示。在主要编辑区中, 可以打开模型中的任意一张图, 并利用左边的工具栏对图进行浏览和修改。



图 2-28 树形视图和文档区

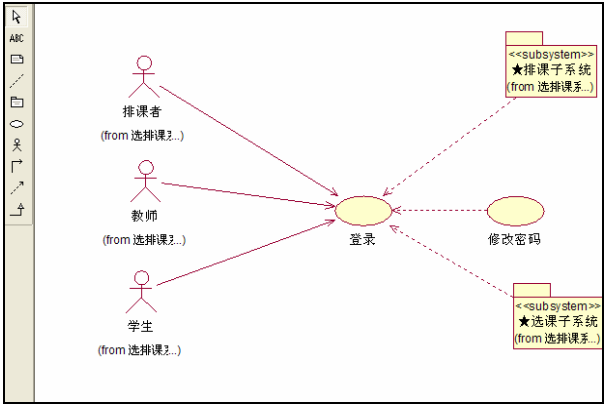


图 2-29 主要编辑区

动作记录区, 在这里记录了对模型所做的所有重要动作, 如图 2-30 所示。



图 2-30 动作记录区



状态栏显示了一些提示和当前所用的语言，如图 2-31 所示。



图 2-31 状态栏

以上分区域介绍了 Rational Rose 的默认界面，详细的界面说明，将在后续章节中结合实际操作做具体说明。

### 2.3.2 用 Rational Rose 建模

#### 1. 创建模型

Rose 模型文件的扩展名是.mdl，要创建模型，需要完成下列步骤：

- (1) 从菜单栏选择“File New”，或单击标准工具栏中的“New”按钮；
- (2) 弹出如图 2-14 所示的对话框，选择要用到的框架，单击“OK”按钮，或者“Cancel”按钮（表示不使用框架）。

如果选择使用框架，Rose 自动装入这个框架的默认包、类和组件。框架提供了每个包中的类和接口，各有相应属性和操作。通过创建框架，可以收集类与组件，作为基础设计和建立多个系统。如果单击“Cancel”按钮，则创建一个空项目，用户需要从头开始对系统建模。

#### 2. 保存模型

Rational Rose 的保存，类似于其他应用程序。可以通过菜单或者工具栏来实现。

- (1) 保存模型：通过选择菜单“File Save”或者工具栏的“Save”按钮，来保存系统建模。
- (2) 保存日志：激活日志窗口（见图 2-30），通过菜单“File Save Log As”来保存，或者右键单击日志窗口，在弹出的菜单中选择“Save Log As”命令来保存。

#### 3. 发布模型

可以把 Rose 建立的模型发布到 Web，使得其他人都能够浏览模型。

- (1) 选择菜单命令的“Tools Web Publisher”，在弹出的如图 2-32 所示的对话框中选择要发布的模型视图和包。

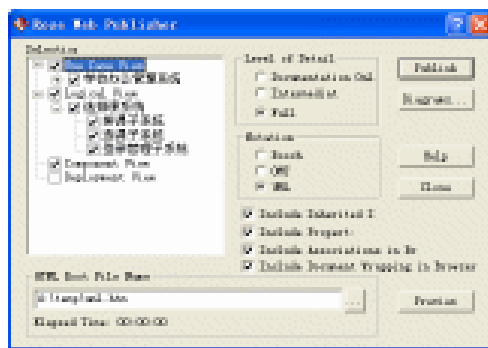


图 2-32 发表界面

- (2) 设定细节内容 (Level of Detail 单选框)。
- (3) 选择是否发布属性, 关联等内容 (见图 2-32 的若干检查框)。
- (3) 输入发表模型的根文件名 (在 HTML Root File Name 文本框中输入)。
- (4) 选择框图的文件格式 (单击 “Diagrams...” 按钮), 如图 2-33 所示。



图 2-33 选择文件格式界面

- (5) 单击 “Publish (发布)” 按钮发布模型。
- (6) 进入发布后的根目录 (笔者设置的目录为 d:\temp), 可以看到发布后的所有文件如图 2-34 所示。

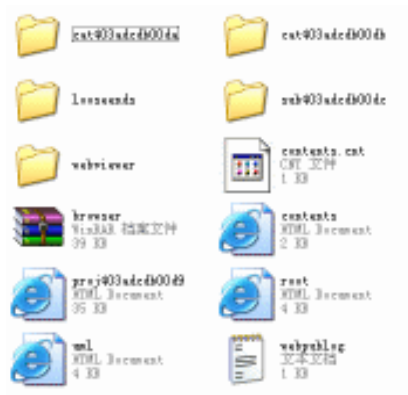


图 2-34 发布后的文件

- (7) 单击 uml.htm 文件, 可以查看整个系统的建模内容, 而不需要通过 Rational Rose 来看, 通过浏览器就可以了, 如图 2-35 所示。

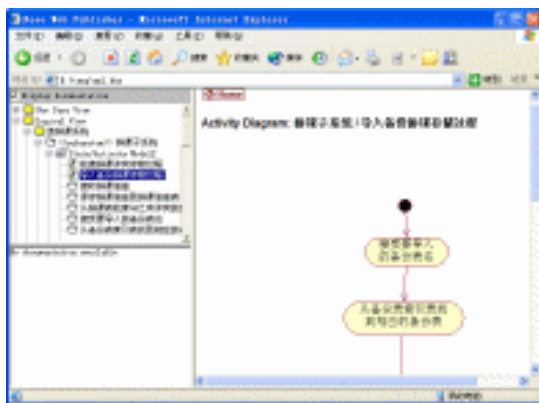


图 2-35 发布后的模型

### 2.3.3 设置全局选项

全局选项可以通过菜单“Tools Options”进行设置，如图 2-36 所示。

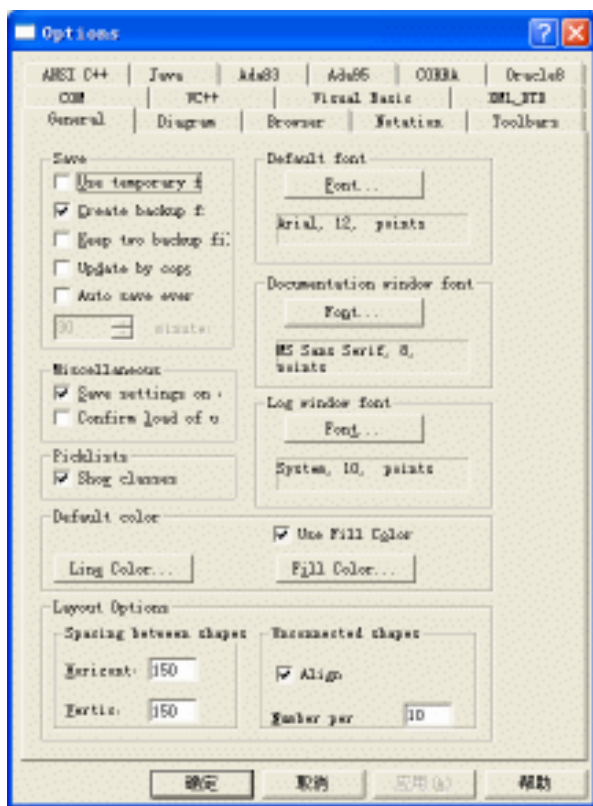


图 2-36 属性界面

#### 1. 设置字体

在图 2-37 所示的对话框中单击“Font...”按钮（根据不同的对象单击不同的“Font...”按钮）弹出如图 2-37 所示的对话框，可以设置字体。

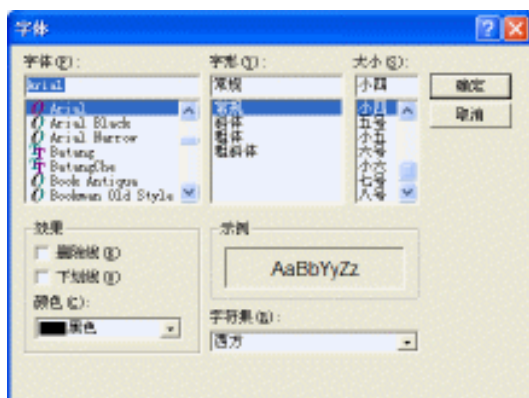


图 2-37 字体选择界面

## 2. 设置颜色

如果要改变对象颜色，可以单击“Line Color”按钮或者“Fill Color”按钮进行颜色选择，如图 2-38 所示。



图 2-38 颜色选择界面

### 2.3.4 框图设计

框图设计是 Rose 使用的主要部分，这里通过 Use Case（用例图）做简要的介绍，具体设计步骤读者可参考后续章节的内容。

#### 1. 创建 Use Case（用例图）

- (1) 右键单击浏览器中的 Use Case View；
- (2) 选择弹出的菜单中的“New Use Case Diagram”，如图 2-39 所示；

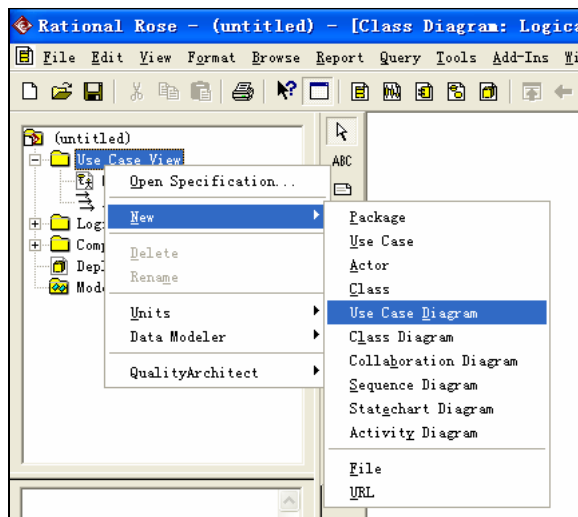


图 2-39 用例新建界面

- (3) 输入框图的名称；
- (4) 双击新创建的框图将其打开。

## 2. 打开 Use Case 图

- (1) 从浏览器中的视图中选择；
- (2) 双击 Use Case 框图打开。

或者通过菜单实现：

- (1) 选择菜单“Browse Use Case Diagram”，在弹出窗口中进行选择，如图 2-40 所示。

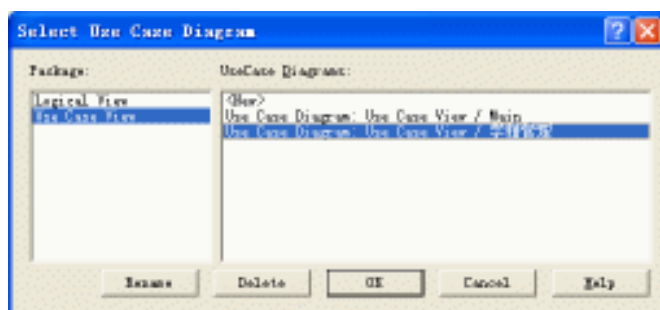


图 2-40 选择 UseCase

- (2) 在包列表 (Package) 中选择框图所在的包；
- (3) 在 Use Case 列表框中选择所要打开的框图；
- (4) 单击“OK”按钮打开。

## 3. 删除框图

- (1) 在浏览器中右键单击框图；
- (2) 在弹出菜单中选择“Delete”并确定，如图 2-41 所示。

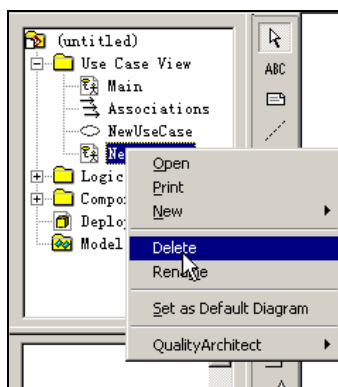


图 2-41 用例选择界面

## 4. Use Case 设计

新建一个 Use Case 框图以后，可以在如图 2-42 所示的设计页面中设计 Use Case 图。图中左边部分是 Use Case 的工具栏，右边部分是进行图形化建模的面板。

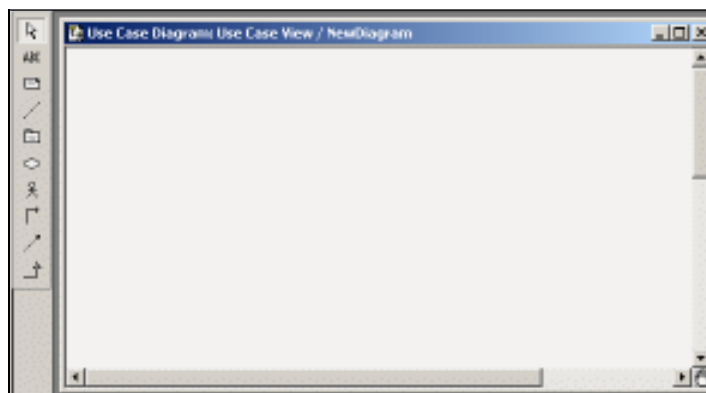


图 2-42 用例绘制界面

对于工具栏, 可以进行定制。如果看不到需要的工具按钮, 可以通过菜单“View Toolbars Configure...”进行设置, 如图 2-43 所示, 这里不仅可以对用例图, 也可以对其他图的快捷工具栏内容进行定制。

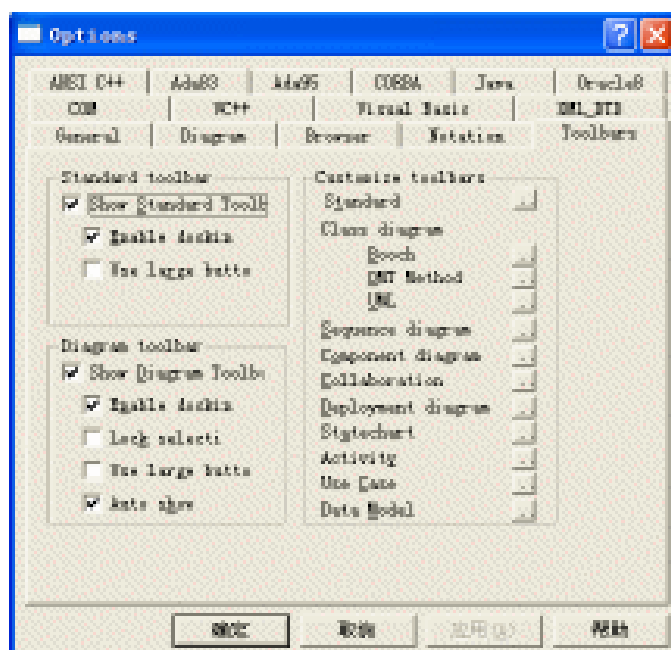


图 2-43 快捷工具设置界面

在快捷工具栏上直接单击右键, 在右键菜单中选择“Customize”, 可以进入如图 2-44 所示的自定义工具栏界面。



图 2-44 工具栏定制界面

接下来可以进行 Use Case 的设计工作，像通常使用的图形设计界面一样，工具栏上的元素可以随意拖放到设计窗口中去，如图 2-45 所示。

注意：每次窗口中删除的元素，其实没有被真正删除，必须在浏览器窗口中通过右键菜单才能删除相应元素，如图 2-46 所示。

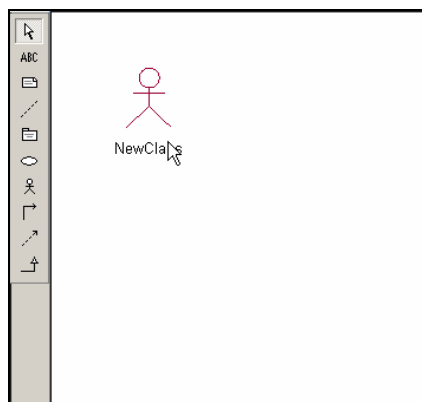


图 2-45 用例角色新建界面

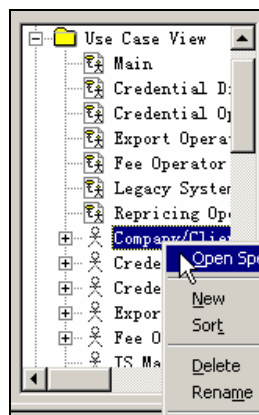


图 2-46 元素删除界面

如图 2-47 所示是一个用 Rational Rose 画出的 Use Case 图的实例。

注意：以上各种符号的含义在后续章节中都会有详尽的介绍，这里只需要了解即可。

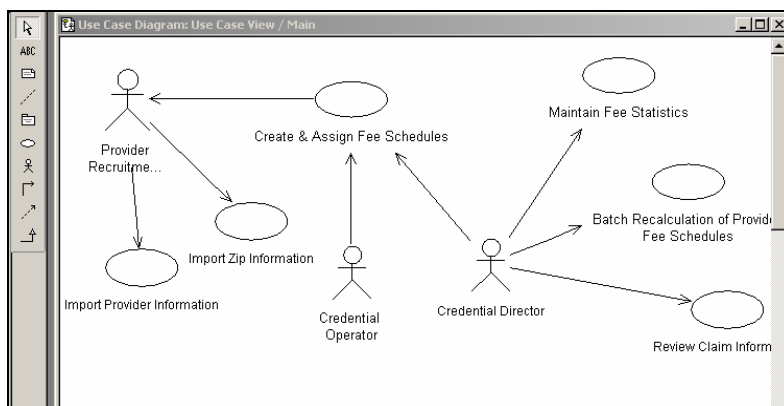


图 2-47 用例图实例

### 2.3.5 双向工程

Rose 提供双向工程的功能。双向工程指的是生成代码（正向工程）和逆向转出工程（反向工程）。生成代码是指根据选择开发应用程序的语言生成对应的程序代码，其步骤为：检查模型 创建组件 建立组件与类的映射 设置代码生成属性 选择类、组件和包 生成代码。逆向转出工程是指根据选择开发应用程序的语言生成对应的程序的代码，其步骤与生成代码相反。

#### 1. 生成代码（正向工程）

##### （1）检查模型

从菜单中选择“Tools Check Model”，从日志窗口中观察错误日志，如果有错误，信息将显示在日志窗口中，如图 2-48 所示。

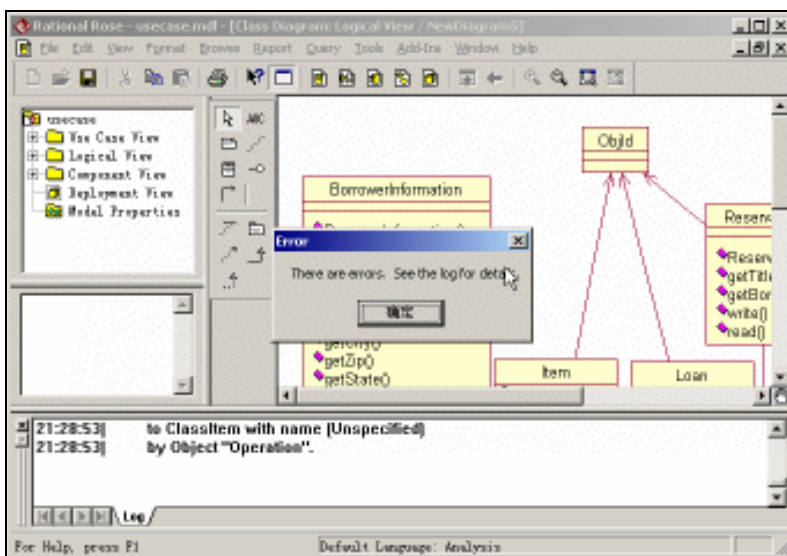


图 2-48 检查模型界面

如果要发现访问问题，即寻找不同包中的两个类之间存在关系时发生的问题，从菜单选择“Report Show Access Violations”访问窗口将显示访问问题，如图 2-49 所示。



图 2-49 显示访问问题

进行语言的独立检查，可以选择“Tools Java Syntax Check”进行语法的检查，如图 2-50 所示。也可以查其他语言的语法，把 Java 替代成相应的语言即可。



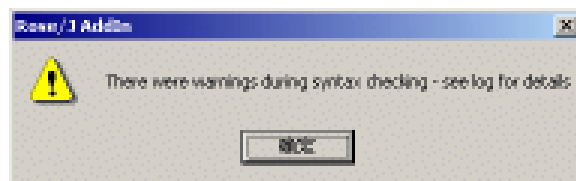


图 2-50 检查语法界面

### (2) 创建组件

组件有多种类型，比如源代码、执行文件、运行库、ActiveX 组件和其他小程序。组件框图上可以加进组件之间的依赖型。单击框图工具栏中的组件图标，可以加入新组件。具体可参考第 7 章中组件图的创建。

### (3) 建立组件与类的映射

单击组件框图中的组件，从菜单选择“Open Specification”，如图 2-51 所示，弹出如图 2-52 所示的对话框，选择“Realizes”选项卡，单击相应的类，并从菜单选择“Assign”，则在浏览器 Logical 的类名中将显示组件名。

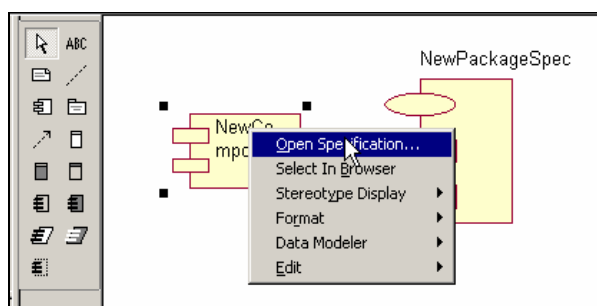


图 2-51 建立映射界面 1

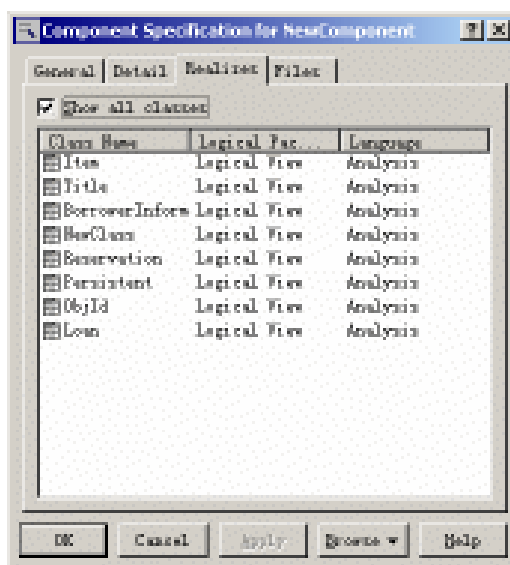


图 2-52 建立映射界面 2

## (4) 设置代码生成属性

类、组件都可以设置代码生成属性，一般使用 Rose 默认的设置方式。要设定浏览代码的生成属性，选择菜单“Tools Options”，弹出如图 2-53 所示的对话框，选择相应的标签就可以设定相应的值。

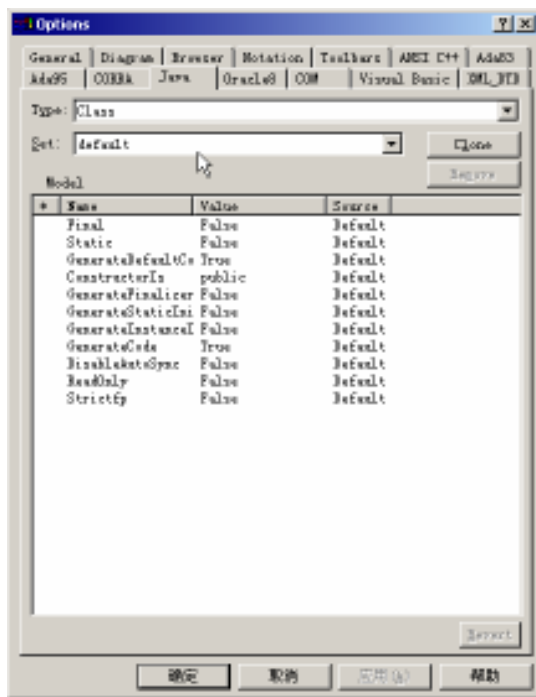


图 2-53 设置属性界面

## (5) 选择类、组件和包

代码生成时，一次可以生成一个类、组件或者包，可以用鼠标单击激活相应类、组建或者包进行选择。

## (6) 生成代码

选择菜单“Add-Ins Add-In Manager”，弹出如图 2-54 所示的对话框，在对话框中通过勾选显示或隐藏各种语言生成菜单。然后选择菜单“Tools Java/J2EE Generation Code”生成代码。

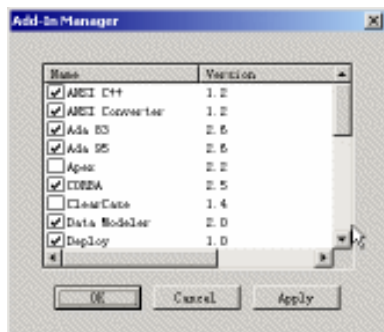


图 2-54 语言选择界面

## 2. 逆向转出工程（反向工程）

利用收集到的元素信息，Rose 创建或者更新对象模型，这些元素包括 Classes、Attributes、Operations、Relationships、Packages 和 Components。逆向转出工程代码过程之后，代码中的组件也会显示出来。其步骤如下：

（1）装入相应框架；

（2）选择菜单“Tools Java/J2EE Reverse Engineer...”，出现如图 2-55 所示的对话框，然后单击“Reverse”按钮即可。

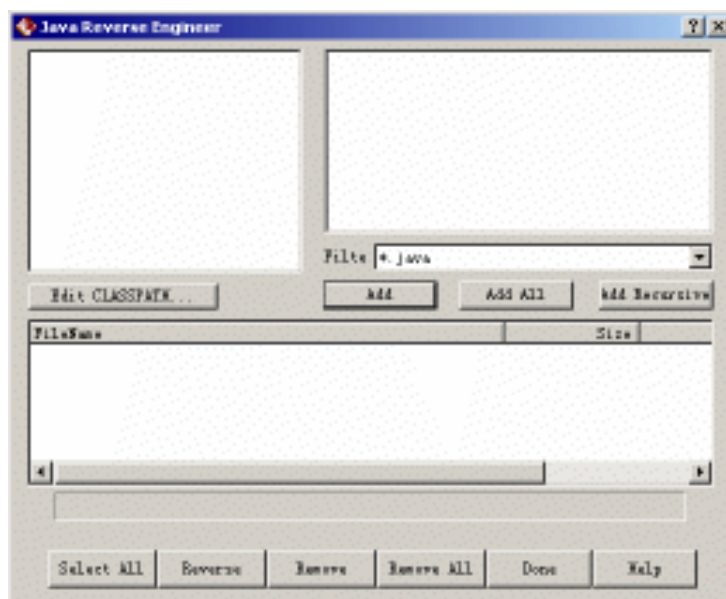


图 2-55 选择逆向转出工程代码

# 第 3 章 UML 语言初览

## 3.1 概述

UML 用来描述模型的内容有 3 种，分别是事物（Things）、关系（Relationships）和图（Diagrams），而这 3 种内容下面又有具体的划分，如图 3-1 所示。

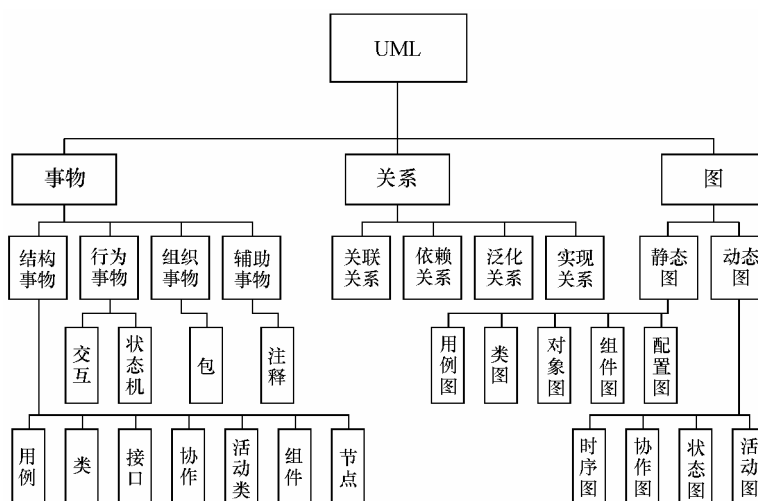


图 3-1 UML 的内容结构

以下各节对该内容结构的各个概念进行详细介绍。

## 3.2 UML 中的事物

从图 3-1 中可以看到，UML 中的事物包括结构事物、行为事物、组织事物和辅助事物（也称注释事物）。

### 3.2.1 结构事物（Structure Things）

结构事物主要包括 7 种，分别是类、接口、协作、用例、活动类、组件和节点。

#### （1）类（Class）

类是具有相同属性、相同方法、相同语义和相同关系的一组对象的集合。在 UML 图中，

类通常用一个矩形来表示，如图 3-2 所示。

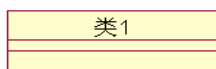


图 3-2 类

#### (2) 接口 (Interface)

接口是指类或组件所提供的、可以完成特定功能的一组操作的集合，换句话说，接口描述了类或组件的对外的、可见的动作。通常，一个类实现一个或多个接口。在 UML 图中，接口通常用一个圆形来表示，如图 3-3 所示。



图 3-3 接口

#### (3) 协作 (Collaboration)

协作定义了交互的操作，表示一些角色和其他元素一起工作，提供一些合作的动作。在 UML 图中，协作通常用一个虚线椭圆来表示，如图 3-4 所示。



图 3-4 协作

#### (4) 用例 (Use Case)

用例定义了系统执行的一组操作，对特定的用户产生可以观察的结果。在 UML 图中，用例通常用一个实线椭圆来表示，如图 3-5 所示。



图 3-5 用例

#### (5) 活动类 (Active Class)

活动类是对拥有线程并可发起控制活动的对象（往往称为主动对象）的抽象。在 UML 图中，活动类的表示方法和普通类的表示方法相似，也是使用一个矩形，只是最外面的边框使用粗线，如图 3-6 所示。

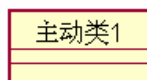


图 3-6 活动类

#### (6) 组件 (Component)

组件是物理上可替换的，实现了一个或多个接口的系统元素。在 UML 图中，组件的表示方法比较复杂，如图 3-7 所示。

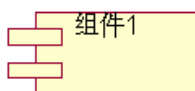


图 3-7 组件

#### (7) 节点 (Node)

节点是一个物理元素，它在运行时存在，代表一个可计算的资源，比如一台数据库服务器。在 UML 图中，节点使用一个立方体来表示，如图 3-8 所示。



图 3-8 节点

### 3.2.2 行为事物 (Behavior Things)

行为事物主要有两种：交互和状态机。

#### (1) 交互 (Interaction)

在 UML 图中，交互的消息通常画成带箭头的直线，如图 3-9 所示。



图 3-9 交互

#### (2) 状态机 (State Machine)

状态机是对象的一个或多个状态的集合。在 UML 图中，状态机通常用一个圆角矩形来表示，如图 3-10 所示。

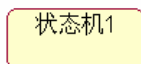


图 3-10 状态机

### 3.2.3 组织事物 (Grouping Things)

组织事物是 UML 模型中负责分组的部分，可以把它看作一个个的盒子，每个盒子里面的对象关系相对复杂，而盒子与盒子之间的关系相对简单。组织事物只有一种，称为包(Package)。

包是一种有组织地将一系列元素分组的机制。包与组件的最大区别在于，包纯粹是一种概念上的东西，仅仅存在于开发阶段结束之前，而组件是一种物理的元素，存在于运行时。在 UML 图中，包通常表示为一个类似文件夹的符号，如图 3-11 所示。

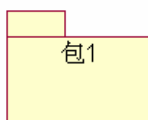


图 3-11 包

### 3.2.4 辅助事物 (Annotation Things)

辅助事物也称注释事物，属于这一类的只有注释 (Annotation)。

注释就是 UML 模型的解释部分。在 UML 图中，一般表示为折起一角的矩形，如图 3-12 所示。

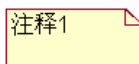


图 3-12 注释

## 3.3 UML 中的关系

UML 中的关系 (Relationships) 主要包括 4 种：关联关系、依赖关系、泛化关系和实现关系。

### 3.3.1 关联关系 (Association)

关联关系是一种结构化的关系，指一种对象和另一种对象有联系。给定关联的两个类，可以从其中的一个类的对象访问到另一个类的相关对象。在 UML 图中，关联关系用一条实线表

示，如图 3-13 所示。

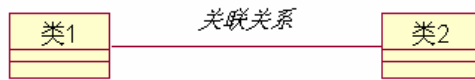


图 3-13 关联关系

另外，关联可以有方向，表示该关联在某方向被使用。只在一个方向上存在的关联，称作单向关联( Unidirectional Association )，在两个方向上都存在的关联，称作双向关联( Bidirectional Association )。

### 3.3.2 依赖关系 (Dependency)

对于两个对象 X、Y，如果对象 X 发生变化，可能会引起对另一个对象 Y 的变化，则称 Y 依赖于 X。在 UML 图中，依赖关系用一条带有箭头的虚线来表示，如图 3-14 所示。



图 3-14 依赖关系

### 3.3.3 泛化关系 (Generalization)

UML 中的泛化关系定义了一般元素和特殊元素之间的分类关系，与 C++ 及 Java 中的继承关系有些类似。在 UML 图中，泛化关系用一条带有空心箭头的实线来表示，如图 3-15 所示。



图 3-15 泛化关系

### 3.3.4 实现关系 (Realization)

实现关系将一种模型元素（如类）与另一种模型元素（如接口）连接起来，其中接口只是行为的说明而不是结构或者实现。真正的实现由前一个模型元素来完成。在 UML 图中，实现关系一般用一条带有空心箭头的虚线来表示，如图 3-16 所示。





图 3-16 实现关系

以上讲述了 UML 中的 4 种关系，除了需要注意各个关系的区别与联系以外，还要了解对关系的修饰。最常见的，对关系可以做两种修饰。

第 1 种是命名，即可以为关系取名，如图 3-17 所示。

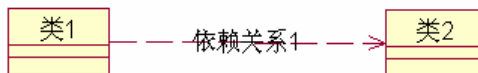


图 3-17 关系命名

第 2 种是数字，可以表示不同对应情况的关系，比如一对多、多对一、一对一和多对多等，如图 3-18 所示。



图 3-18 关系数字

### 3.4 UML 中的视图

UML 中的各种组件和概念之间没有明显的划分界限，但为方便起见，用视图来划分这些概念和组件。视图只是表达系统某一方面特征的 UML 建模组件的子集。视图的划分带有一定的随意性，但希望这种看法仅仅是直觉上的。在每一类视图中使用一种或两种特定的图来可视化地表示视图中的各种概念。

在最上一层，视图被划分成 3 个视图域：结构分类、动态行为和模型管理。

结构分类描述了系统中的结构成员及其相互关系。类元包括类、用例、组件和节点。类元为研究系统动态行为奠定了基础。类元视图包括静态视图、用例视图、实现视图和配置视图。

动态行为描述了系统随时间变化的行为。行为用从静态视图中抽取的系统的瞬间值的变化来描述。动态行为视图包括状态视图、活动视图和交互视图。

模型管理说明了模型的分层组织结构。包是模型的基本组织单元。特殊的包还包括模型和子系统。模型管理视图跨越了其他视图，并根据系统开发和配置组织这些视图。

UML 还包括多种具有扩展能力的组件，这些扩展能力有限但很有用。这些组件包括约束、构造型和标记值，它们适用于所有的视图元素。

表 3-1 所示列出了 UML 的视图和视图所包括的图以及与每种图有关的主要概念。不能把这张表看成是一套死板的规则，应将其视为对 UML 常规使用方法的指导，因为 UML 允许使

用混合视图。

表 3-1 UML 视图和图

主要的域	视图	图	主要概念
结构	静态视图	类图	类、关联、泛化、依赖关系、实现、接口
	用例视图	用例图	用例、参与者、关联、扩展、包括、用例泛化
	实现视图	组件图	组件、接口、依赖关系、实现
	配置视图	配置图	节点、构件、依赖关系、位置
动态	状态视图	状态图	状态、事件、转换、动作、
	活动视图	活动图	状态、活动、完成转换、分叉、结合
	交互视图	时序图	交互、对象、消息、激活
		协作图	协作、交互、协作角色、消息
模型管理	模型管理视图	类图	包、子系统、模型
可扩展性	所有	所有	约束、构造型、标记值

## 3.5 UML 中的图

UML 中的图有 9 种，主要分为两类：静态图和动态图。

### 3.5.1 静态图

UML 中有 5 种静态图：用例图、类图、对象图、组件图和配置图。

#### (1) 用例图 (Use Case Diagram)

用例图展现了一组用例、参与者以及它们间的关系。可以用用例图描述系统的静态使用情况。在对系统行为组织和建模方面，用例图是相当重要的。用例图的例子如图 3-19 所示。

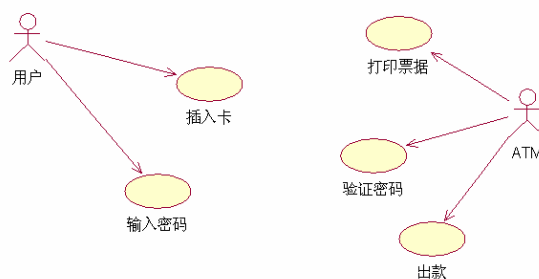


图 3-19 用例图举例

注释：

小人形状的用户和 ATM 是参与者；

椭圆形状的插入卡、输入密码、打印票据、验证密码及出款是用例；

这些概念在后续章节会有详细的阐述。

## (2) 类图 (Class Diagram)

类图展示了一组类、接口和协作及它们间的关系,在建模中所建立的最常见的图就是类图。用类图说明系统的静态设计视图,包含主动类的类图——专注于系统的静态进程视图。系统可有多类图,单个类图仅表达了系统的一个方面。一般在高层给出类的主要职责,在低层给出类的属性和操作。类图的例子如图 3-20 所示。

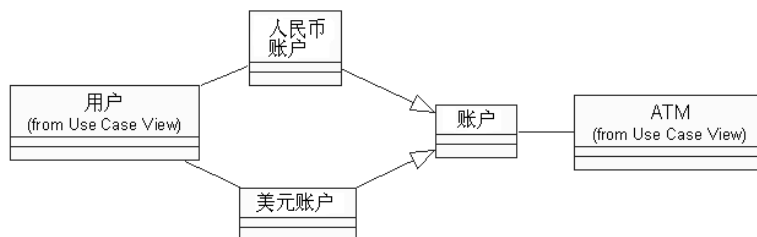


图 3-20 类图举例

注释：

图中反映了 5 个类之间的关联关系；

人民币账户类和美元账户类从账户类继承；

账户与 ATM 相关联；

用户与两种账户类相关联。

## (3) 对象图 (Object Diagram)

对象图展示了一组对象及它们间的关系。用对象图说明类图中所反应的事物实例的数据结构和静态快照。对象图表达了系统的静态设计视图或静态过程视图,除了现实和原型方面的因素外,它与类图作用是相同的。

## (4) 组件图 (Component Diagram)

组件图,又称构件图,展现了一组组件之间的组织和依赖,用于对原代码、可执行的发布、物理数据库和可调整的系统建模。组件图的例子如图 3-21 所示。

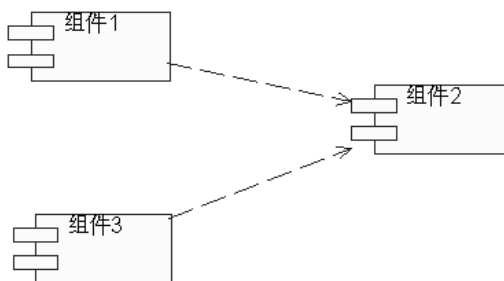


图 3-21 组件图举例

注释：图中有 3 个组件，组件 1 与组件 3 和组件 2 存在着依赖关系。

#### (5) 配置图 (Deployment Diagram)

配置图展现了对运行时处理节点以及其中组件的部署。它描述系统硬件的物理拓扑结构（包括网络布局和组件在网络上的位置），以及在此结构上执行的软件（即运行时软组件在节点中的分布情况）。用配置图说明系统结构的静态配置视图，即说明分布、交付和安装的物理系统。配置图的例子如图 3-22 所示。

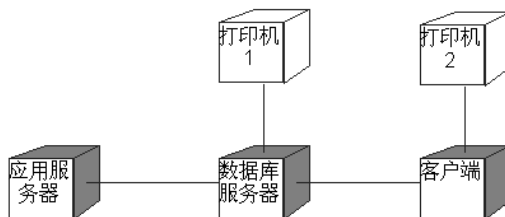


图 3-22 配置图举例

注释：图中有 3 个处理机与和两个设备，相互之间是关联的关系。

### 3.5.2 动态图

动态图有 4 种，分别是：时序图、协作图、状态图和活动图。

#### (1) 时序图 (Sequence Diagram)

时序图展现了一组对象和由这组对象收发的消息，用于按时间顺序对控制流建模。用时序图说明系统的动态视图。时序图的例子如图 3-23 所示。

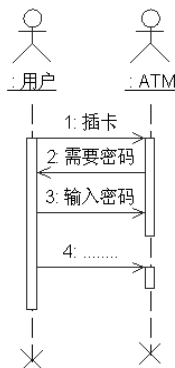


图 3-23 时序图举例

注释：该图反映了用户与 ATM 的交互过程：用户把卡插入 ATM 中，ATM 向用户发出需要密码指令，用户再把密码提供给 ATM.....

#### (2) 协作图 (Collaboration Diagram)

协作图展现了一组对象间的连接以及这组对象收发的消息。它强调收发消息对象的组织结构，按组织结构对控制流建模。协作图的例子如图 3-24 所示。

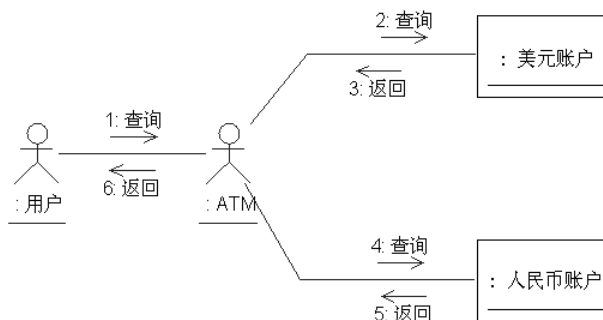


图 3-24 协作图举例

注释：用户向 ATM 提出查询要求，ATM 根据用户提供的信息，选择对于美元账户或者人民币账户的查询路径，并返回信息给用户。

### (3) 状态图 (Statechart Diagram)

状态图展示了一个特定对象的所有可能状态以及由于各种事件的发生而引起的状态间的转移。一个状态图描述了一个状态机，用状态图说明系统的动态视图。状态图对于接口、类或协作的行为建模尤为重要，可用它描述用例实例的生命周期。状态图的例子如图 3-25 所示。

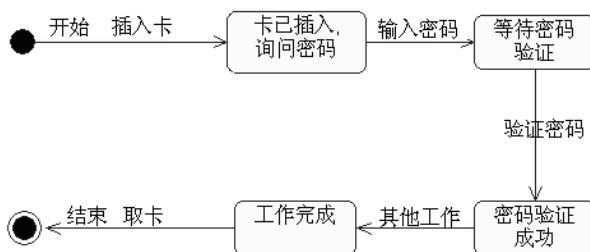


图 3-25 状态图举例

注释：

标有“开始”和“结束”的是开始状态和结束状态；

整个状态的转换过程如下：从开始状态进入插入卡状态，经过输入密码操作，进入等待密码验证状态，然后进行密码验证，之后进入密码验证成功状态，再进行其他工作，最后工作完成取卡，进入到状态结束。

### (4) 活动图 (Activity Diagram)

活动图显示了系统中从一个活动到另一个活动的流程。活动图显示了一些活动，强调的是对象之间的流程控制。活动图的例子如图 3-26 所示。

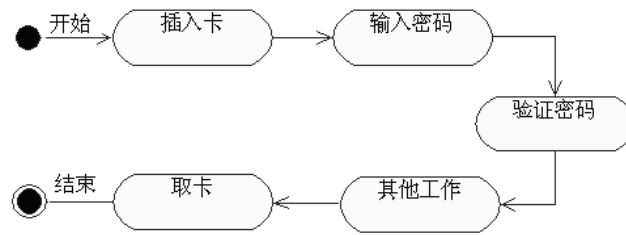


图 3-26 活动图举例

注释：

标有“开始”和“结束”的是开始状态和结束状态。

活动图以活动作为节点，从开始状态起步，进行插入卡活动，然后输入密码，系统验证密码，并进行其他工作，最后用户取卡，接着活动结束。

# 第 4 章 静态视图

## 4.1 概述

静态视图 (Static Diagram) 是建立其他视图的基础。静态视图用于对应用领域中的概念以及系统实现有关的内部概念建模,它将行为实体描述成离散的模型元素,但不描述与时间有关的系统行为。静态视图将系统中的行为实体看作是将被类所指定、拥有并使用的物体,这些实体的动态行为在其他视图(交互视图和状态机视图)中进行描述。

静态视图包括类图,对象图和包图。其中类图描述系统中类的静态结构。它不仅定义系统中的类,表示类之间的联系,如关联、依赖、聚合等,还包括类的内部结构(类的属性和操作)。类图描述的是一种静态关系,在系统的整个生命周期都是有效的。通过分析用例和问题域,就可以得到相关的类,然后再把逻辑上相关的类封装成包。这样可以很好体现出系统的分层结构,使人们对系统层次关系一目了然。对象图是类图的实例,几乎有与类图完全相同的标识。他们的不同点在于对象图显示类图的多个对象实例,而不是实际的类。一个对象图是类图的一个实例。由于对象存在生命周期,因此对象图只能在系统某一时间存在。包由包或类构成,表示包与包之间的关系。包图用于描述系统的分层结构。

## 4.2 类与关系

### 4.2.1 类

类是任何面向对象系统中最重要构造块。类是一种重要的分类器 (Classifier),用来描述结构和行为特性的机制,它包括类、接口、数据类型、信号、组件、节点、用例和子系统。

类是对一组具有相同属性、操作、关系和语义的对象的描述。这些对象包括现实世界中的软件事物和硬件事物,甚至也可以包括纯粹概念性的事物,它们是类的实例。一个类可以实现一个或多个接口。结构良好的类具有清晰的边界,并成为系统中职责均衡分布的一部分。

类在 UML 中由专门的图符表达,是分成 3 个分隔区的矩形。其中顶端的分隔区为类的名字,中间的分隔区存放类的属性、属性的类型和值(在 UML 符号表示中给出类的初始值),第 3 个分隔区放操作、操作的参数表和返回类型,如图 4-1 所示。



图 4-1 类的 UML 符号

在给出类的 UML 表示时，可以根据建模的实际情况来选择隐藏属性区或操作区，或者两者都隐藏。如图 4-2 和图 4-3 所示表示了图书馆中书的类，但图 4-3 中左侧类的操作分隔区为空，这并不代表没有操作，只是因为未显示，右侧类则隐藏了属性分隔区。

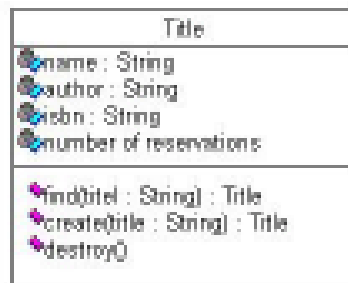


图 4-2 类

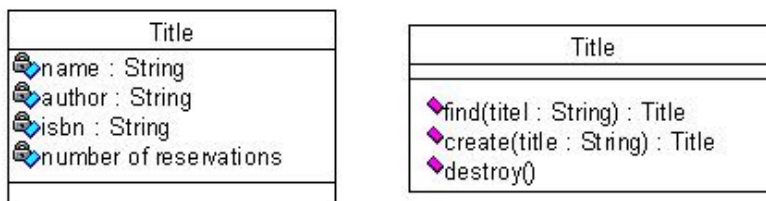


图 4-3 类的省略形式

#### (1) 名称 (Name)

类的名称是每个类所必需的构成，用于和其他类相区分。名称 (name) 是一个文本串，可分为简单名称和路径名称。单独的名称即不包含冒号的字符串叫做简单名 (single name)，用类所在的包的名称作为前缀的类名叫做路径名 (path name)，如图 4-4 所示。

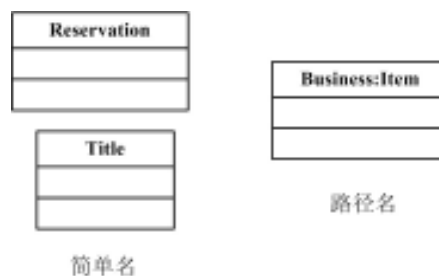


图 4-4 类的简单名和路径名



## (2) 属性 (Attribute)

类的属性是类的一个组成部分,它描述了类在软件系统中代表的事物所具备的特性。类可以有任意数目的属性,也可以没有属性。属性描述了正在建模的事物的一些特性,这些特性是所有对象所共有的。例如对学生来建模,每个学生都有名字、专业、籍贯和出生年月,这些都可以作为学生类的属性。

在 UML 中类属性的语法为:

```
[可见性] 属性名 [ : 类型 ] [= 初始值] [{ 属性字符串 }]
```

其中[ ]中的部分是可选的。类中属性的可见性主要包括 public、private 和 protected 3 种,它们分别用“+”、“-”和“#”来表示。

根据定义,类的属性首先是类的一部分并且每个属性都必须有一个名字以区别于类的其他属性,通常情况下属性名由描述所属类的特性的短名词或名词短语构成(通常以小写字母开头)。类的属性还有取值范围,因此还需为属性指定数据类型。例如布尔类型的属性可以取两个值 TRUE 和 FALSE。当一个类的属性被完整的定义后,它的任何一个对象的状态都由这些属性的特定值所决定。

## (3) 操作 (Operation)

类的操作是对类的对象所能做的事务的抽象。它相当于一个服务的实现,该服务可以由类的任何对象请求以影响其行为。一个类可以有任何数量的操作或者根本没有操作。类的操作必须有一个名字,可以有参数表,可以有返回值。根据定义,类的操作所提供的服务可以分为两类,一类是操作的结果引起对象状态的变化,状态的改变也包括相应动态行为的发生;另一类是为服务的请求者提供返回值。

在 UML 中类操作的语法为:

```
[可见性] 操作名 [ ( 参数表 ) ] [ : 返回类型 ] [{ 属性字符串 }]
```

实际建模中,操作名是用来描述所属类的行为的短动词或动词短语(通常以小写字母开头)。如果是抽象操作,则用斜体字表示。

## (4) 职责 (Responsibility)

职责是类或者其他元素的契约或义务。当创建一个类时,就声明这个类的所有对象具有相同种类的状态和相同种类的行为。在较高层次上,这些相应的属性和操作正是要完成类的职责和特性。类的职责是自由形式的文本,它可以写成一个短语、一个句子或是一段短文。在图形上,把职责列在类图底部的分隔栏中。举例来说,ATM 管理系统中,CheckUser 负责检验用户,并决定是否给该用户办理信用卡,如图 4-5 所示。

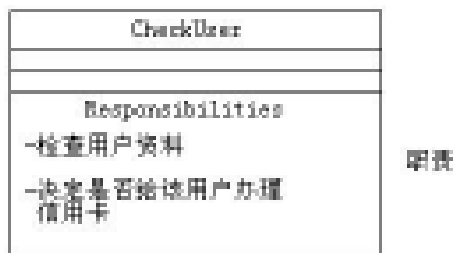


图 4-5 类的职责

### 4.2.2 关系

抽象过程中,你会发现很少有类是独立存在的,大多数的类以某些方式彼此协作。如果离开了这些类之间的关系,那么类模型仅仅只是一些代表领域词汇的杂乱矩形方框。因此,在进行系统建模时,不仅要抽象出形成系统词汇的事物,还必须对这些事物间的关系进行建模。

关系 (Relationship) 是事物间的联系。在类的关系中,最常用的 4 种分别为:依赖 (Dependency), 它表示类之间的使用关系;泛化 (Generalization), 它表示类之间的一般和特殊的关系;关联 (Association), 它表示对象之间的结构关系;实现 (Realization), 它是规格说明和其实现之间的关系。

#### 1. 依赖

依赖是两个元素之间的关系,对一个元素 (提供者) 的改变可能会影响或提供消息给其他元素 (客户)。也就是说:客户以某种方式依赖于提供者。在实际的建模中,类元之间的依赖关系表示某一类元以某种方法依赖于其他类元。

从语义上理解,关联、实现和泛化都是依赖关系,但因为他们有更特别的语义,所以在 UML 中被分离出来作为独立的关系。

在图形上, UML 把依赖描述成一条有方向的虚线,指向被依赖的对象,如图 4-6 所示。

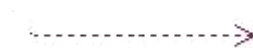


图 4-6 依赖关系

UML 建模过程中,常用依赖指明一个类把另一个类作为它的操作的特征标记中的参数。当被使用的类发生变化时,那么另一个类的操作也会受到影响,因为这个被使用类此时已经有了不用的接口和行为。举例来说,类 TV 中的方法 change 使用了类 channel 的对象作为参数。因此在类 TV 和类 channel 之间存在着依赖关系。显然,当类 channel 发生变化时 (电视频道改变), 类 TV 的行为也发生了相应的变化,如图 4-7 所示。

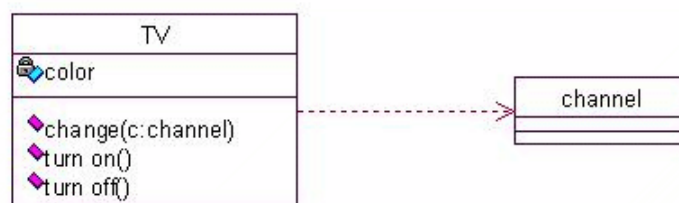


图 4-7 依赖关系

UML 定义了 4 种基本依赖类型。它们分别是使用 (Usage) 依赖、抽象 (Abstraction) 依赖、授权 (Permission) 依赖和绑定 (Binding) 依赖。在定义依赖关系时,要用到两个概念——客户和提供者。客户是指依赖关系起始的模型元素,提供者是指依赖关系箭头所指的模型元素。

### (1) 使用依赖

所有的使用依赖是非常直接的，它通常表示客户使用提供者提供的服务以实现它的行为。以下给出 5 种使用依赖定义的，应用于依赖关系的原型。

`<<use>>`

`<<use>>` 依赖是类最常用的依赖。它声明使用一个模型元素需要用到已存在的另一个模型元素，这样才能正确实现使用者的功能（包括了调用、实例化、参数和发送）。

在实际建模中，有 3 种情况下产生使用依赖：当客户类的操作需要提供者类的参数；客户类的操作返回提供者类的值以及客户类的操作在实现中使用提供者类的对象。以图 4-7 所示的模型为例（TV 类为客户、channel 类为提供者），它们间的关系应该符合上述的第 3 种情况，是使用依赖。

`<<call>>`

`<<call>>` 是操作间的依赖，它声明了一个类调用其他类的操作的方法。这种类型的依赖在 UML 建模中不被广泛使用，它适用于更深的建模层次。同样，现在也很少有 CASE 工具支持操作之间的依赖。

`<<parameter>>`

`<<parameter>>` 是操作和类之间的依赖，它描述的是一个操作和它的参数之间的关系。这种依赖方式在实际中也较少被使用。

`<<send>>`

`<<send>>` 描述的是信号发送者和信号接收者之间的关系，它规定客户把信号发送到非指定的目标。

`<<instantiate>>`

`<<instantiate>>` 是指一个类的方法创建了另一个类的实例声明，它规定客户创建目标元素的实例。

### (2) 抽象依赖

抽象依赖建模表示客户和提供者之间的关系，它依赖于在不同抽象层次上的事物。以下给出 3 种抽象依赖定义的，应用于依赖关系的原型。

`<<trace>>`

`<<trace>>` 声明不同模型中的元素之间存在的一些连接。例如提供者可以是类的分析视图，客户则可以是更详细的设计视图，系统分析师可以用 `<<trace>>` 来描述它们之间的关系。

`<<refine>>`

`<<refine>>` 声明具有不同语义层次上的元素之间的映射。抽象依赖中的 `<<trace>>` 可以用来描述不同模型中的元素间的连接关系，`<<refine>>` 则用于相同模型中元素间的依赖。例如在分析阶段遇到一个类 Student，在设计时这个类细化成更具体的类 Student。

`<<derive>>`

`<<derive>>` 声明一个类可以从另一个类导出。当想要表示一个事物能从另一事物派生而来时就使用这个依赖构造型。

### (3) 授权依赖

授权依赖表达一个事物访问另一事物的能力。提供者可以规定客户的权限，这是提供者控制和限制对其内容访问的方法。以下给出 3 种授权依赖定义的应用于依赖关系的原型。

<<access>>

<<access>>是包间的依赖，它描述允许一个包访问另一个包的内容。<<access>>允许一个包（客户）引用另一个包（提供者）内的元素，但客户包必须使用路径名称。

<<import>>

<<import>>是与<<access>>概念相似的依赖，它允许一个包访问另一个包的内容并为被访问包的组成部分增加别名。<<import>>将提供者的命名空间整合到客户的命名空间，但当客户包中的元素与提供者中的元素同名时，会产生冲突。在这种情况下，可以使用路径名或增加别名来解决冲突。

<<friend>>

<<friend>>允许一个元素访问另一个元素，不管被访问的元素是否可见，这大大地便利了客户类访问提供者的私有成员。但并不是所有的计算机语言都支持<<friend>>依赖，C++允许类间的<<friend>>依赖，而Java和C#则不支持。

#### （4）绑定依赖

绑定依赖是较高级的依赖类型，它用于绑定模板以创建新的模型元素。

<<bind>>规定了客户用给定的实际参数实例化提供者模板。

## 2. 泛化

泛化是一般事物（称为超类或父类）和该事物的较为特殊的种类（称为子类）之间的关系，子类继承父类的属性和操作，除此之外通常子类还添加新的属性和操作，或者修改了父类的某些操作。泛化意味着子类的对象可以用在父类的对象可能出现的地方，但反过来则不成立。例如电视可以分为彩色电视和黑白电视，电视也可以分为CRT电视、液晶电视、背投电视和等离子电视。这些都是泛化关系，只是观察事物的角度不一样。更简单的来说，泛化关系描述了类之间的“is a kind of”（属于……的一种）的关系。

在图形上，泛化用从子类指向父类的空心三角形箭头表示（如图4-8所示），多个泛化关系可以用箭头线表示的树形来表示，每一个分支指向一个子类。



图4-8 泛化关系的UML符号表示

如图4-9所示，类BookTitle（图书名）和类MagazineTitle（杂志名）是类Title（题名）的子类。因此，有空心三角形箭头从类MagazineTitle和类BookTitle指向类Title。很显然类MagazineTitle和类BookTitle继承了类Title的某些属性，还添加了属于自己的某些新的属性。

泛化有两个主要的用途：

第一是用来定义以下的情况：当一个变量被声明表示承载某个给定类的值时，可使用类的实例作为值，这正是由Barbara Liskov提出的可替代性原则。这种原则表明无论何时祖先被声明，则后代的一个实例可以被使用。例如，父类电视机被声明，那么一个液晶电视的对象就是一个合法的值。泛化使得多态操作成为了可能，即操作的实现是由它们所使用的对象的类，而不是由调用者确定的。多态的意思是“多种形态”，多态操作是具有多种实现的操作。如图4-10描述了一个多态模型，假设有类Canvas，它维护Shape的集合。

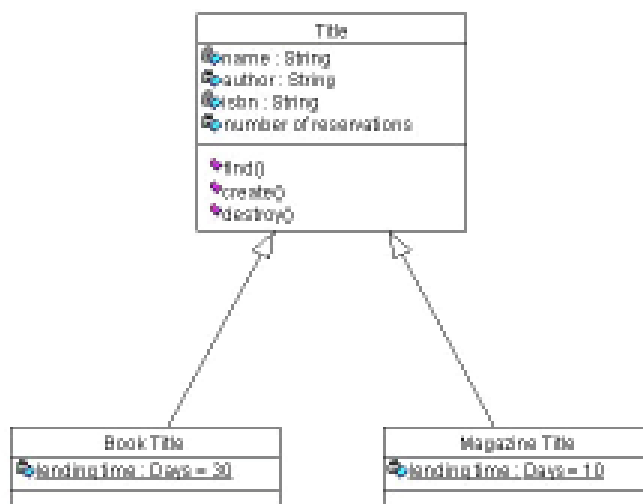


图 4-9 泛化关系

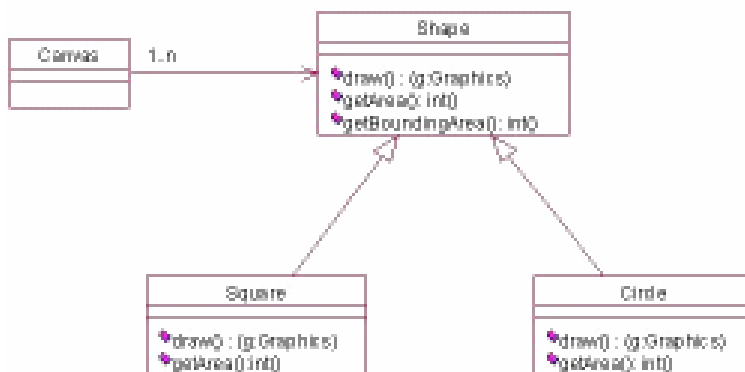


图 4-10 多态示例

第二，共享祖先所定义成分的前提下，允许它自身定义增加的描述，这被称作继承。继承是一种机制，通过该机制类对象的描述，从类及其祖先的声明部分聚集起来。对类元而言，没有具有相同特征标记的属性会被多次声明，无论直接的或继承的，否则将发生冲突，且模型形式错误。也就是说，祖先声明过的属性不能被后代再次声明，但如果类的接口一致（具有同样的参数、约束和含义），操作可在多个类中声明。如只有一个父类，则是单继承，如图 4-9 所示，如果一个类有多个父类，并且从每一个父类那里都得到继承信息，则称之为多重继承。多重继承的实例如图 4-11 所示。

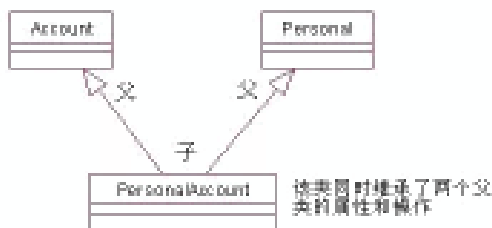


图 4-11 多重继承

### 3. 关联

关联是一种结构关系，它指明一个事物的对象与另一个事物的对象间的联系。也就是说，如果两事物间存在链接，这些事物的类间必定存在着关联关系，因为链接是关联的实例，就如同对象是类的实例一样。举例来说，学生在大学里学习，大学又包括许多的学院，显然在学生、学院和大学间存在着某种链接。在 UML 建模设计类图时，就可以在学生（Student）、学院（Institute）和大学（University）3 个类间之间建立关联关系。

在图形上，关联用一条连接相同类或不同类的实线表示（如图 4-12 所示）。要表示结构的关系时就使用关联。

图 4-12 关联关系的 UML 符号表示

除了关联的基本形式外，还有 4 种应用于关联的修饰，它们分别是名称、角色、多重性和聚合。

#### （1）名称

关联可以有一个名称，用于描述该关系的性质（如图 4-13 所示）。此关联名称应该是动词短语，因为它表明源对象正在目标对象上执行动作。名称也可以前缀或后缀一个指引阅读方向的实心三角形箭头，为的是消除名称含义上可能存在的歧义。但关联的名称并不是必需的，当要明确的给关联提供角色名或当一个模型存在许多关联且要对这些关联进行查阅和区别时，才要给出关联名。



图 4-13 关联的名称

#### （2）角色

当一个类处于关联的某一端时，该类就在这个关系中扮演一个特定的角色。具体来说，角色就是关联关系中一个类对另一个类所表现的职责。当它们由这个关系的实例所连接时，角色

名称应该是名词或者是名词短语。如图 4-14 所示,在这对关系中 Student 类将扮演 Learner(学习者)的角色, University 类将扮演 Teacher(教学者)的角色,它们是彼此相关联的。

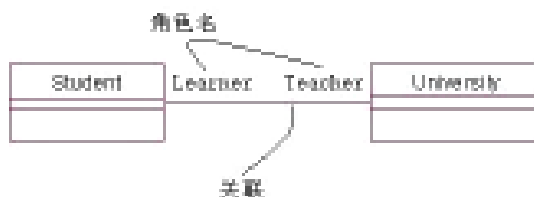


图 4-14 关联的角色

### (3) 多重性

约束是 UML 三大扩展机制之一。多重性是其中的第一种约束,也是目前使用最广泛的约束。在实际建模过程中,在关联实例中说明两个类间存在多少个相互连接是很重要的。这里所指的“多少”就是关联角色的多重性。

多重性被表示为用点分隔的区间,每个区间的格式为:minimum..maximum,其中 minimum 和 maximum 是 Int 型的整数。多重性语法的一些示例如表 4-1 所示。

表 4-1 多重语法的示例

修 饰	语 义
0..1	0 或 1
1	恰为 1
0..* 等同于 0..n	0 或更多
1..* 等同于 1..n	1 或更多
* 等同于 n	0 或更多
1..6	1 ~ 6

Student(学生)与 University(大学)间的多重性关系如图 4-15 所示。

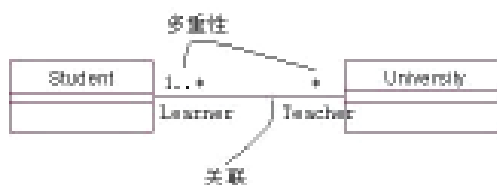


图 4-15 关联的多重性

### (4) 聚合关系 (Aggregation Relationship)

聚合关系是一种特殊的关联关系,它表示类间的关系是整体与部分的关系。更简单的说,

关联关系中一个类描述了一个较大的事物，它由较小的事物组成，这种关系就是聚合，它描述了“has-a”的关系，即整体对象拥有部分对象。

在 UML 中，聚合关系用空心的菱形头的实线表示，如图 4-16 所示，Institute（学院）是 University（大学）的组成部分，因此，在类 University 和类 Institute 之间的关系是聚合关系。

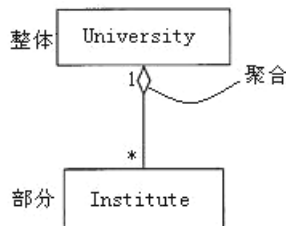


图 4-16 聚合关系

#### （5）组成关系（Composition Relationship）

聚集表示部分与整体关系的关联，组成是更强形式的关联，整体有管理部分的特有的职责并且它们有一致是生命期。可以这么说，组成是另一种形态的聚合，它在聚合的基础上添加了更精确的一些语义。

在 UML 中，聚合关系用实心的菱形头的实线表示，组成关系示例如图 4-17 所示。

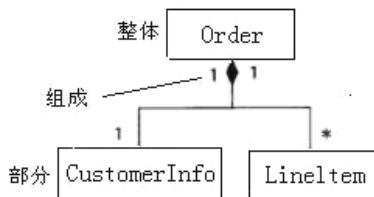


图 4-17 组成关系

#### （6）导航性（Navigation）

导航性表示可从源类的任何对象到目标类的一个或多个对象遍历。也就是说给定源类的一个对象，可以达到目标类的所有对象。可以在关联关系上加上箭头表示导航方向。只在一个方向上可以导航的关联称为单向关联（Uni-directional Association），用一条带箭头的实线表示；在两个方向上都可以导航的关联称为双向关联（Bi-directional Association），用一条没有箭头的实线表示。

好的面向对象分析与设计的目标之一就是降低类间的耦合度，系统分析师可以使用导航性来实现。通过类 Order 和 Product 间的单向关联，能够简单的从对象 Order 到对象 Product 导航。对象 Product 不知道它属于哪个 Order，因此没有到 Order 的耦合。具体的模型示例如图 4-18 所示。



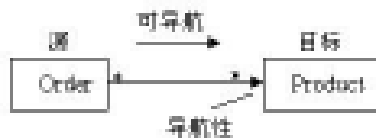


图 4-18 导航性

#### 4. 实现

实现是规格说明和其实现间的关系。它表示不继承结构而只继承行为。大多数情况下，实现关系用来规定接口和实现接口的类或组件之间的关系。接口是能够让用户重用系统一组操作集的 UML 组件。一个接口可以被多个类或组件实现，一个类或组件也可以有多个接口。

可以在两种情况下使用实现关系：第一，在接口与实现该接口的类间；第二，在用例以及实现该用例的协作间。

在 UML 中，实现关系用一个带空心三角形的箭头来表示，箭头方向指向接口。例如，计算机键盘保证自己的部分行为能够“实现”打字员的行为，也就是说它们间存在着实现关系。具体模型示例如图 4-19 所示。

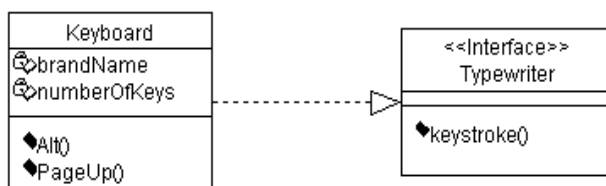


图 4-19 实现关系

实现关系还有一种省略表示法：将接口表示为一个小圆圈并和实现它的类用一条线相连。如图 4-20 所示。

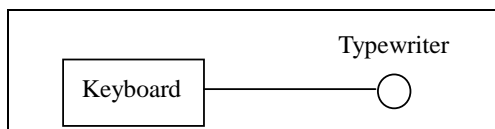


图 4-20 实现关系省略表示法

### 4.3 类图

在对一个软件系统进行设计和建模的时候，通常是从构造系统的基本词汇开始，包括构造这些基本词汇的基本属性和行为。然后要考虑的是这些基本词汇之间的关系，因为在任何系统中孤立的元素是很少出现的。这样系统分析师就能从结构上对所设计的系统有清晰的认识。比如构造汽车，首先确定像车厢、车轮和发动机等的基本词汇，分析它们的属性（如车厢的材质、颜色等）和行为（如发动机的运转等），然后考虑这些词汇间的关系。

系统分析师将上述的行为可视化后，就是通常所说的类图。类图是面向对象系统建模中最常用的图，它是定义其他图的基础，在类图的基础上，状态图、协作图、组件图和配置图等进一步描述系统的其他方面的特性。

### 4.3.1 类图的概念和内容

类图 (Class Diagram) 是描述类、接口、协作以及它们之间关系的图。它是系统中静态视图的一部分，静态视图可以包括许多的类图。静态视图用于为软件系统进行结构建模，它构造系统的词汇和关系，而结构模型的可视化就是通过类图来实现的。

类图所包括的内容：类；接口；协作；依赖、泛化、实现和关联关系。

像 UML 建模中的其他图一样，类图也可以包含注解和约束。类图中还可以含有包或子系统，它们使模型元素聚集成更大的模块。类图的内容如图 4-21 所示。

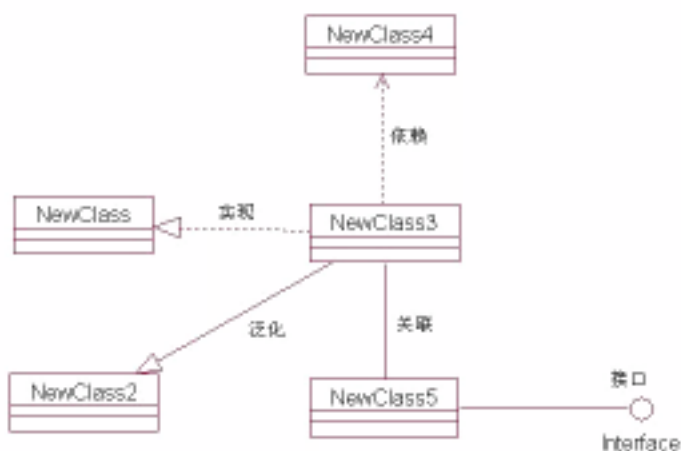


图 4-21 类图的内容

### 4.3.2 类图的用途

类图是系统静态视图的一部分，它主要用来描述软件系统的静态结构。该视图主要支持系统的功能需求，也就是系统要提供给最终用户的服务。当系统分析师以支持软件系统的功能需求为目的设计静态视图时，通常以下述 3 种方法之一使用类图。

#### (1) 对系统的词汇建模

在前面已经提到，用 UML 构建系统通常是从构造系统的基本词汇开始，用于描述系统的边界，也就是说用来决定哪些抽象是要建模系统中的一部分，哪些抽象是处于要建模系统之外。这是非常重要的一项工作，因为系统最基本的元素在这里被确定。系统分析师可以用类图描述抽象和它们的职责。

#### (2) 对简单协作建模

现实世界中的事物大多都是相互联系、相互影响的，将这些事物抽象成类后，情况也是如

此。所要构造的软件系统中的类很少有孤立存在的，它们总是和其他类协同工作，以实现强于单个类的语义。因此，在抽象了系统词汇后，系统分析师还必须将这些词汇中是事物协同工作的方式可视化和详述。

### （3）对逻辑数据库模式建模、

在设计一个数据库时，通常使用数据库模式来描述数据库的概念设计。数据库模式建模是对数据库概念设计的蓝本，可以使用类图对这些数据库的模式进行建模。

## 4.3.3 类图建模技术

### 1. 对简单协作建模

协同是软件系统的动态交互在软件系统静态视图上的映射。协同的静态结构是通过类图表达出来的。在对类图的简单协同建模时，不仅要描述类的职责、结构和服务，还要强调类间的关系。

在协同建模时，要遵循的策略包括以下几个方面。

（1）识别要模拟的机制。一个机制描述了被建模的部分系统的一些功能和行为，这些功能和行为是由类、接口等元素交互作用产生的。

（2）对每种机制，识别参与协作的类、接口和其他协作，并识别它们间的关系。

（3）通过协作的脚本，发现建模的模型是否有被遗漏和有语义错误，并更正错误。

（4）得出相应类的对象，并确定具体的属性和操作。

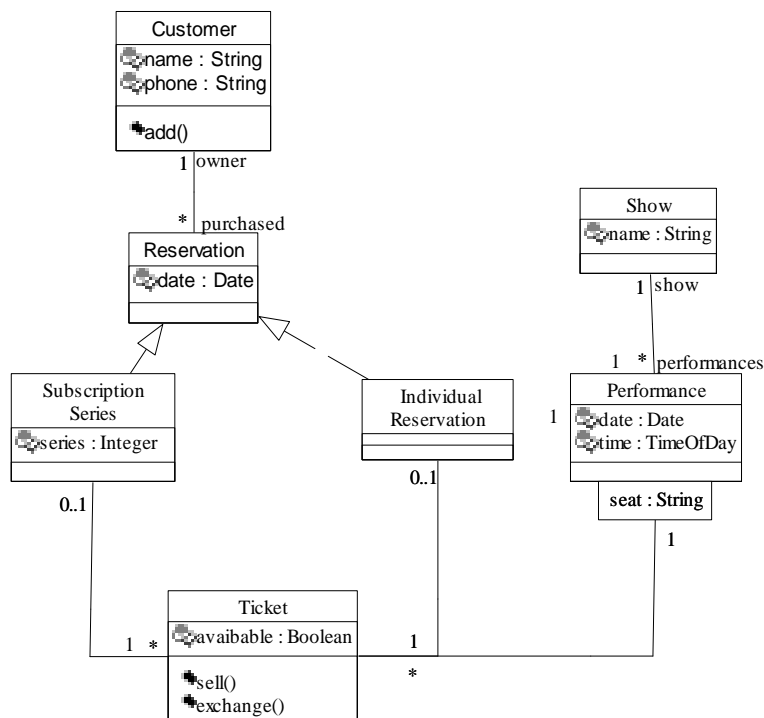


图 4-22 简单协作建模

如图 4-22 所示，该类图是一个订票系统的类图之一。该类图中包括有类 Customer（顾客）、Reservation（订票）、Individual Reservation（个人订票）、Subscription Series（订套票）、Ticket

(票)、Performance (出席) 和 Show (演出)。类 Individual Reservation 和类 Subscription Series 都继承父类 Reservation 的属性和一部分操作。类 Customer 和类 Reservation 是一对多的关系，即一个顾客能够预定许多场演出的票。类 Ticket 和类 Performances、类 Performances 和类 Show 都是多对一的关系，这个系统还包括许多类，它们没有出现在图 4-22 中，这些类以及它们间的关系可以通过其他类图来描述。

## 2. 对数据库模式建模

在对软件系统进行建模时，不仅要定义系统的动态行为，还需要为动态行为所操作的数据指定相应的格式。这些数据通常被永久的保持在数据库中，以便以后检索，就是所谓的永久存储对象。

传统的逻辑数据库建模工具“实体 - 关系 (E-R)”图只针对数据，而 UML 的类图还允许对行为建模。在物理模型中，类图的操作一般转换成数据库的触发器或存储过程。因此，UML 很适合对逻辑数据库和物理数据库建模。

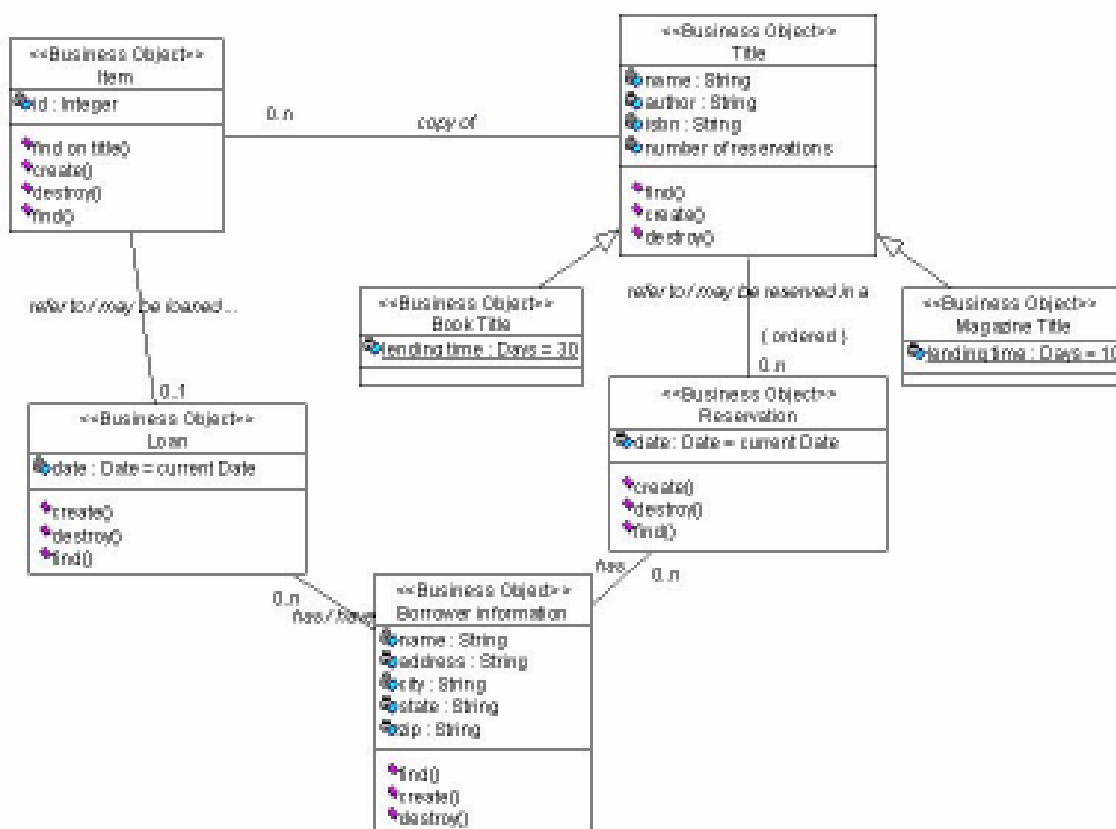


图 4-23 对数据库建模

在为数据库建模时，要遵循的策略包括以下几个方面。

- (1) 在系统中确定的类，它的状态必须超过其应用系统生命周期。
- (2) 创建包含这些类的类图，并把它们标记成永久的 (persistent)。
- (3) 展开这些类的结构信息，即详细的描述属性的细节，并注重关联和构造类的基数。

(4) 观察系统中的公共模式 (如循环关联、一对一关联等), 它们往往使物理数据库设计复杂化。如果有必要, 需要创建简化逻辑结构的中间抽象。

(5) 考虑这些类的行为, 扩充那些对于数据存储和数据完整性很重要的操作。

(6) 如果可能, 用工具把逻辑设计换成物理设计。

如图 4-23 所示描述了某图书馆信息系统的一组类。这些类都被标记成永久的 (persistent), 这是因为这些类的实例需要被数据库永久性的保持。类图还给出了类的属性和操作以及它们间的关系。例如, 类 Magazine Title 和类 Book Title 继承父类 Title 的属性和一些操作。从类图中, 读者可以很容易的发现类间的关联关系。比如, 每一个 Borrower Information (借阅人信息) 都可以对应于 0 个或多个 Loan (借书信息) 和 Reservation (预定信息), 而每个 Loan 和 Reservation 只能对于 1 个 Borrower Information, 图中其他类之间的关联关系与此类似。该类图描述的是数据库逻辑结构。

## 4.4 对象图

在 UML 中, 类图描述的是系统的静态结构和关系, 而交互图描述系统的动态特性。在跟踪系统的交互过程时, 往往会涉及到系统交互过程的某一瞬间交互对象的状态, 但系统类图和类图都没有对此进行描述。于是, 在 UML 里就用对象图来描述参与一个交互的各对象在交互过程中某一时刻的状态。

在一个复杂的系统中, 出错时所涉及的对象可能会处于一个具有众多类的关系网中。分析这样的情况可能会很复杂, 因此系统测试员需要为出错时刻系统各对象的状态建立对象图, 这将大大的便利分析错误, 解决问题。

### 4.4.1 对象图的概念和内容

在 UML 中, 对象图 (Object Diagram) 表示在某一时刻一组对象以及它们之间关系的图。对象图可以被看作是类图在系统某一时刻的实例。在图形上, 对象图由节点以及连接这些节点的连线组成, 节点可以是对象也可以是类, 连线表示对象间的关系。对象图模型如图 4-24 所示。

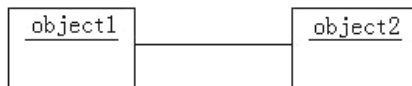


图 4-24 对象图

对象图除了描述对象以及对象间的连接关系外, 还可包含标注和约束。如果有必要强调与对象相关类的定义, 还可以把类描绘到对象图上。当系统的交互情况非常复杂时, 对象图还可包含模型包和子系统。

和类图一样可以使用对象图对系统的静态设计或静态进程视图建模, 但对象图更侧重于现实或原型实例, 这种视图主要支持系统的功能需求, 也就是说, 系统提供给其最终用户的服务。

对象图描述了静态的数据结构。

#### 4.4.2 对象图建模

对象图主要用来描述类的实例在特定时刻的状态。它可以是类的实例也可以是交互图的静态部分。对于复杂的数据结构，对象图也非常有用。

对于组件图和实施图来说，UML 可以直接对它们建模，组件图和实施图上分别可以包含部件或结点的实例。如果这两张图上只包含实例，而不包含任何消息，那么也可以把它们看成是特殊的对象图。

对象图的建模过程：

- (1) 确定参与交互的各对象的类，可以参照相应的类图和交互图；
- (2) 确定类间的关系，如依赖、泛化、关联和实现；
- (3) 针对交互在某特定时刻各对象的状态，使用对象图为这些对象建模；
- (4) 建模时，系统分析师要根据建模的目标，绘制对象的关键状态和关键对象之间的连接关系。

如图 4-25 所示显示了针对某公司建模的一组对象。该图描述了该公司的部门分组情况。c 是类 Company 的对象，这个对象与 d1, d2, d3 连接，d1, d2, d3, d4 都是类 Department 的对象，它们具有不同的属性值，即有不同的名字。d1 和 d4 连接，d4 是 d1 的一个实例。

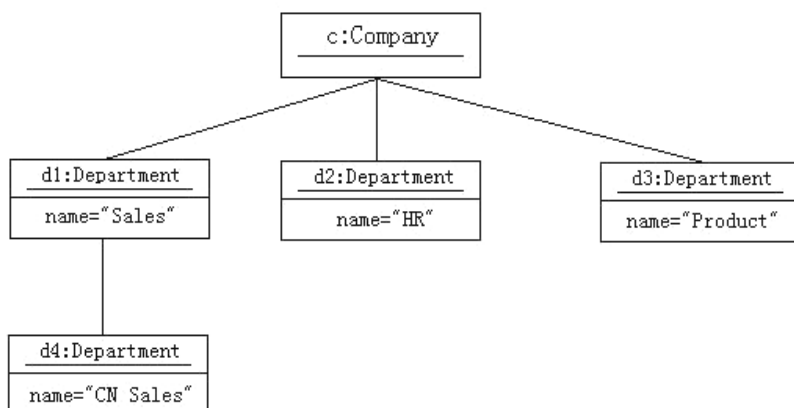


图 4-25 对象图

## 4.5 包图

包图由包和包之间的联系构成，它是维护和控制系统总体结构的重要建模工具。

当对大型系统进行建模时，经常需要处理大量的类、接口、组件、节点和图，这时就有必要将这些元素进行分组，即把那些语义相近并倾向于一起变化的元素组织起来加入同一包，这样方便理解和处理整个模型。同时也便于轻松地控制这些元素的可见性，使一些元素在包外可见，一些元素是隐藏在包内的。设计良好的包是高内聚、低耦合的，并且对其内容的访问具有

严密的控制。

#### 4.5.1 包的名字

和其他建模的元素一样，每个包都必须有一个区别于其他包的名字。模型包的名字是一个字符串，它可分为简单名（simple name）和路径名（path name）。简单名是指包仅含一个简单的名称，路径名是指以包位于的外围包的名字作为前缀的包名。

图形上，包是带有标签的文件夹，如图 4-26 所示。



图 4-26 包的名字

#### 4.5.2 包拥有的元素

包是对模型元素进行分组的机制，它把模型元素划分成若干个子集。包可以拥有 UML 中的其他元素，包括类、接口、组件、节点、协作、用例和图，包甚至还可以包含其他包。

包的作用不仅仅是为模型元素分组。它还对所拥有的模型元素构成一个命名空间，这就意味着一个模型包的各个同类建模元素不能具有相同的名字，不同模型包的各个建模元素能具有相同的名字，因为它们代表不同的建模元素。在同一包内，不同种类的模型元素能够具有相同的名字，但可能会带来不必要的麻烦，不推荐这样做。

如图 4-27 所示，可以用文字或者图形的方式来显示包的内容。

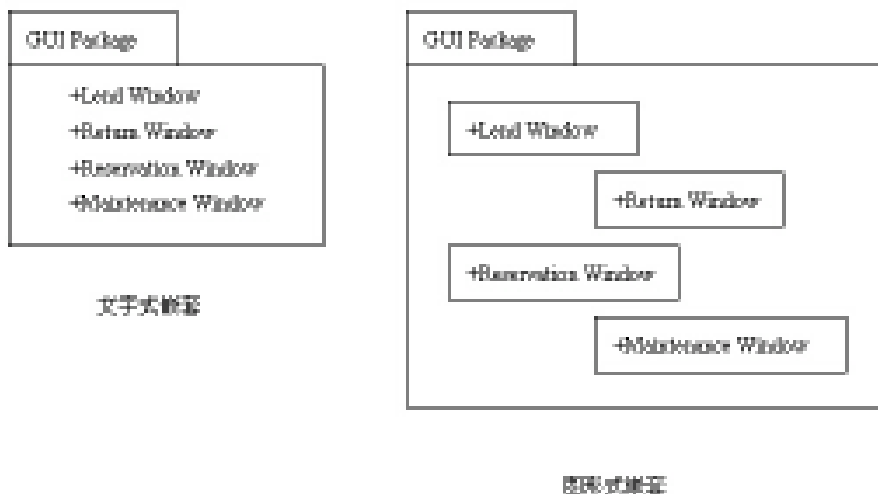


图 4-27 包拥有的元素

### 4.5.3 包的可见性

包在软件模型中不可能是孤立存在的，包内的模型元素必然会和外部的类存在某些联系。而好的软件模型中各个包间应该做到高内聚、低耦合，为了能做到这一点，应该对包内的元素加以控制，使得某些元素能被外界访问，包内其他的元素对外界不可见。这就是所谓的包内元素可见性控制。

包的可见性用来控制包外界的元素对包内元素的可访问权限，这一点和类的可见性类似。可见性可以分成 3 种。

(1) 公有访问 (public)：包内的模型元素可以被任何引入此包的其他包的内含元素访问。公有访问用前缀于内含元素名字的加号 (+) 表示。

(2) 保护访问 (protected)：表示此元素能被该模型包在继承关系上后继模式包的内含元素访问。保护访问用前缀于内含元素名字的 # 号 (#) 表示。

(3) 私有访问 (private)：表示此元素可以被属于用一包的内含元素访问。私有访问用前缀于内含元素名字的减号 (-) 表示。

如图 4-27 中的包内含的 4 个元素都是公有访问的，引入 GUI Package 的包都能看见这 4 元素。从包外看，Lend Window 的限定全名应该是 GUI Package :: Lend Window。

### 4.5.4 引入与输出

在 UML 里，引入一个包中的元素可以单向的访问另一个包中的元素。引入 (import) 关系用构造型的 import 来修饰。包中具有公有访问权限的内含元素称为输出 (export)。

如图 4-28 所示，包 GUI 有两个公共类 Window 和 Form，类 EvenHandler 是受保护的，不能被包输出。包 Policies 引入包 GUI，因此 GUI 中的类 window 和 Form 对于包 Policies 中的元素是可见的，受保护类 EvenHandler 对于包 Policies 中的元素是不可见的。本例中，包 Client 引入包 Policies，包 Policies 引入包 GUI，但并不意味着包 Client 引入包 GUI。如果包 Client



想访问包 GUI，包 Client 必须显示的引入包 GUI。由于包 Server 没有引入包 GUI、Policies 和 Client，因此它们之间不能相互访问。

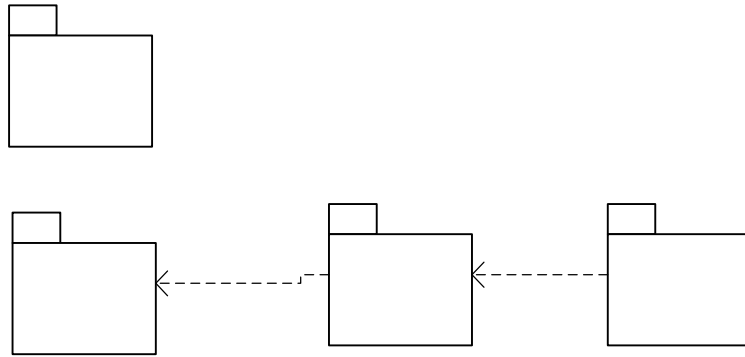


图 4-28 引入和输出

#### 4.5.5 泛化关系

和类间的泛化关系类似，包间也存在着泛化关系。包间的泛化关系也像类那样遵循替代原则，特殊包可以应用到一般包被使用的任何地方。包间还存在另一种关系：引入和访问依赖，用于在一个包引入另一个包输出的元素。

如图 4-29 所示，包 GUI 包含两个公共类 Window 和 Frame，一个受保护类 EventHandler。如类的继承那样，包能继承一些元素也可能替换或修改一些元素。例如特殊包 ClientGUI 继承了一般包 GUI 的公共类 Window 和受保护类 EventHandler，覆盖了类 Frame，并添加了新的元素 Toolbar。



图 4-29 包间的泛化关系

#### 4.5.6 标准元素

UML 的扩充机制同样适用于包，可以使用标记值来增加包的新特性，用构造型来描述包的新种类。UML 定义了 5 种构造型来为其标准扩充，分别是虚包 (facade)、框架 (framework)、桩 (stub)、子系统 (subsystem) 和系统 (system)。

##### (1) 虚包

虚包是包的一种扩充，它只拥有对其他包内元素的引用，本身不包括任何定义模型元素。

##### (2) 框架

框架是一个主要由样式 (pattern) 组成的包。

Server  
Package

GUI  
Package

## (3) 桩

桩描述一个作为另一个包的公共内容代理的包。

## (4) 子系统

子系统代表系统模型中一个独立的组成部分。子系统代表系统中一个语义内聚的元素的集合，可以用接口来指定与外界的联系和其外部行为特征。

## (5) 系统

系统代表当前模型描述的整个软件系统。

#### 4.5.7 包建模技术

当为较复杂的系统建模时，使用包是非常有效的建模方法。包将建模元素按语义分组，从而使得复杂的系统模型能够被构造、表达、理解和管理。

包在很多方面与类相似，但是在对大系统模型时要特别注意区别包与类。类是对问题领域或解决方案的事物的抽象，包是把这些事物组织成模型的一种机制。包可以没有标识，因为它没有实例，在运行系统中不可见；类必须有标识，它有实例，类的实例（对象）是运行系统的组成元素。

建立包图的具体做法如下。

(1) 分析系统模型元素（通常是对象类），把概念上或语义上相近的模型元素纳入一个包。

(2) 对于每一个包，标出其模型元素的可视性（公共、保护或私用）。

(3) 确定包与包之间的依赖联系，特别是输入依赖。

(4) 确定包与包之间的泛化联系，确定包元素的多态性与重载。

(5) 绘制包图。

(6) 包图精化。

如图 4-30 所示显示了图书馆信息系统的包图。整个系统分为了两个包 Business Package 和 GUI Package。其中 Business Package 包括了系统中的事务类，比如书名、借阅信息和预定信息等，GUI Package 提供了系统的用户接口类，比如借书、归还、预定和续借窗体。这种划分方式使系统结构清晰并且更容易被理解。

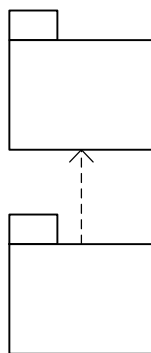


图 4-30 图书馆信息系统的包图

## 4.6 实例——图书馆管理系统中的静态视图

建立系统的静态视图的过程是对系统领域问题及其解决方案的分析和设计的过程。静态视图设计的主要内容是类图的建立，也就是找出系统中类与类之间的联系，并加以分析，最后用图形表示出来。

以“图书馆管理系统”为例来建立相应的静态视图，图书管理系统的建模背景在第9章中有详细的说明，这里说明静态视图部分相关内容的操作实现。

### 4.6.1 建立对象图步骤

建立对象图的步骤如下：

- (1) 研究分析问题领域，确定系统的需求；
- (2) 发现对象和对象类，明确类的属性和操作；
- (3) 发现类之间的静态联系，一般与特殊关系，部分和整体关系，研究类之间的继承性和多态性；
- (4) 设计类与联系；
- (5) 绘制对象类图并编制相应的说明。

从分析问题领域来涉及对象与类是比较常规的面向对象的系统分析方法，UML 采用 Rational 统一过程的 Use Case 驱动的分析方法，从业务领域得到参与者与用例，建立业务模型，读者可参考第5章的相关内容。

### 4.6.2 对象的生成

整个图书管理系统的类数目众多，这里不一一分析，以图书管理系统的读者与书籍信息、借阅信息和预留信息等为例来说明对象图的建立过程。

读者与书籍信息是图书管理系统的基本信息，是系统必需的部分。

#### (1) 读者类的基本属性

- 名字
- 邮编
- 地址
- 城市
- 省份
- 借书
- 预留书籍

#### (2) 书籍类的基本属性

- 书名
- 作者
- 序列号

## ■ 类型

## 4.6.3 用 Rose 绘制对象图

首先,在 Rose 开发环境中树形列表的“Logical View”文件夹(其实是包)图标上单击鼠标右键,在弹出的快捷菜单中,最常用的功能是“Open Specification(打开属性说明)”和“New(新建 UML 元素)”。其中,“Open Specification”可以打开当前选定对象的属性和说明,并对其具体的修改和更新。而“New”则可以新建包括 Class(类)、Class Utility、Use Case(用例)、Interface(接口)、Package(包)、Class Diagram(类图)、Use Case Diagram(用例图)、Collaboration Diagram(协作图)、Sequence Diagram(时序图)、State Chart Diagram(状态图)和 Activity Diagram(活动图),这里选择 Class Diagram(类图),如图 4-31 所示。

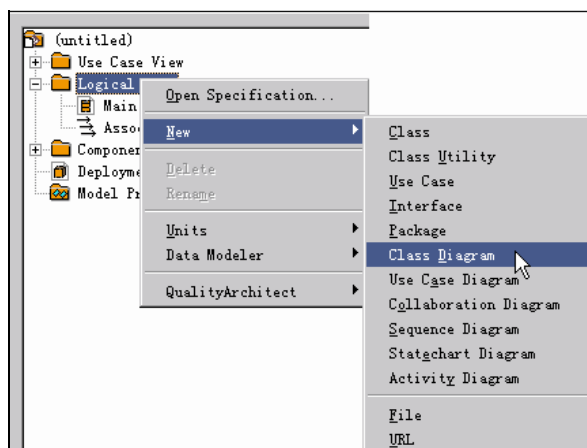


图 4-31 新建类图

这样,一个类图就新建完成了,如图 4-32 所示,修改类图的名称为 Class Diagram。双击打开类图,界面上部中间的编辑工具栏将有所变化,如图 4-33 所示。

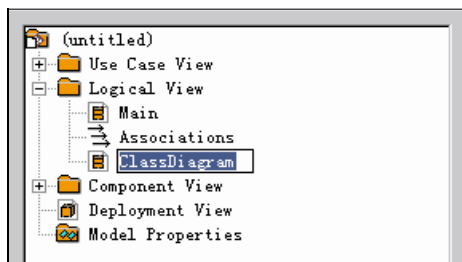


图 4-32 修改类图名称

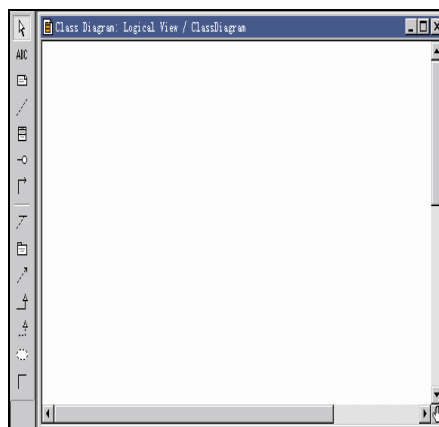


图 4-33 编辑区域

编辑工具栏上从上到下依次是：Selection Tool（选择工具）、Text Box（文本框）、Note（注解）、Anchor Note to Item（注解和元素的连线，是虚线）、Class（类）、Interface（接口）、Unidirectional Association（有方向的关联关系）、Association Class（关联类）、Package（包）、Dependency or Instantiate（依赖关系或实例关系）、Generalization（泛化关系）、Realize（实现关系）、Use-Case Realization（用例实现）和 Association（无方向的关联关系）。

不过并不是只能加入这些符号，默认的编辑工具栏只提供了一些最常用的符号列表，而用户可以对这个列表进行个性化设置，按照自己的需要来定制工具栏很简单，只需要在这个编辑工具栏上单击右键，在弹出的快捷菜单中选择 Customize（自定义）菜单项，如图 4-34 所示。

这样就会弹出如图 4-35 所示的对话框，在这个对话框里，有两个列表框，通过对这两个列表框中元素的移动，可以定制当前的编辑工具栏。

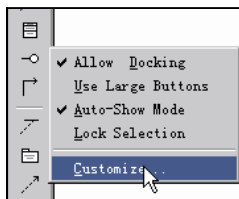


图 4-34 定制编辑工具栏



图 4-35 定制编辑工具栏对话框

单击工具栏上的类图标，单击设计框的空白处，在设计框中加入一个类，同时修改相应的类名称为 BrowserInformation，如图 4-36 和图 4-37 所示。

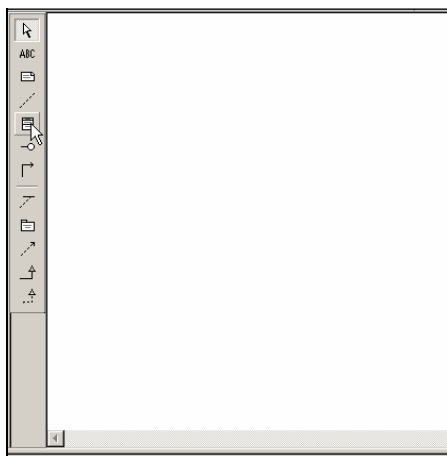


图 4-36 画类图过程示意图 1

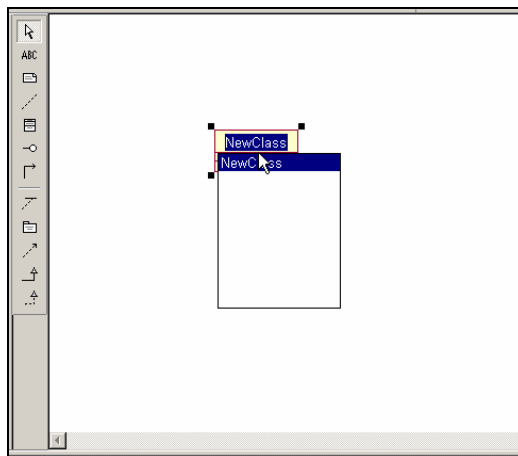


图 4-37 画类图过程示意图 2

在生成的类上点击鼠标右键，在弹出的如图 4-38 所示的菜单中选择“New Attribute”，加入类的属性，单击“New Operation”，加入类的方法。双击属性或者操作名，可以进行直接的修改。

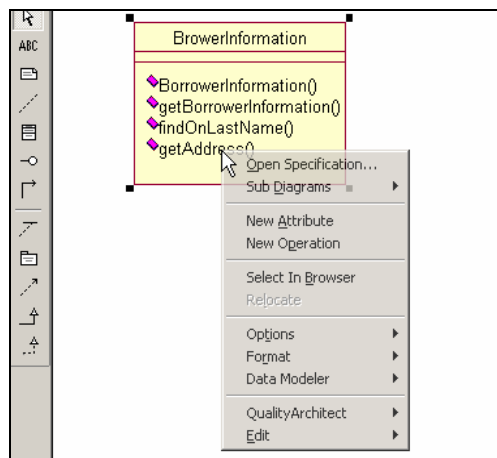


图 4-38 定制类的属性

单击“Open Specification”菜单，将弹出如图 4-39 所示的对话框，通过这个对话框可以设置这个类的基本信息。

Name (名称): 类的名称

Type(类型): 类的类型, 包括 ParamerizedClass、InstantiatedClass、ClassUtility、Paramerized ClassUtility 和 InstantiatedClassUtility 6 种类型。

Stereotype (模板类型): 模板的类型共有 11 种。

Export Control (输出控制): 包括 Public、Private、Protected 和 Implementation 4 种。

Document (说明): 描述类图的文档

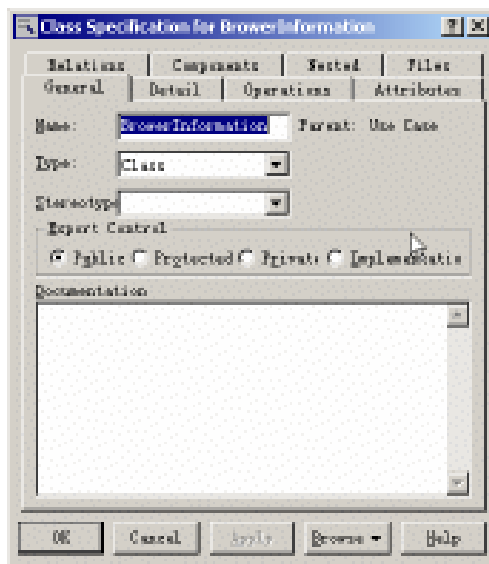



图 4-39 类属性的设置

用同样的方法创建 Persistent 类，然后单击工具栏上的  图标，如图 4-40 所示，接着单击 BorrowerInformation 类，并延伸到 Persistent，如图 4-41 所示，表明相互的继承关系。

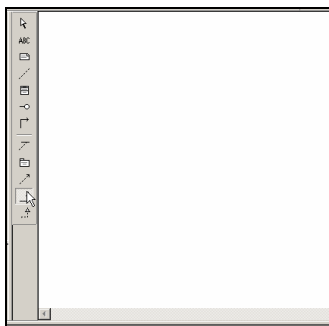


图 4-40 建立类之间的关联示意图 1

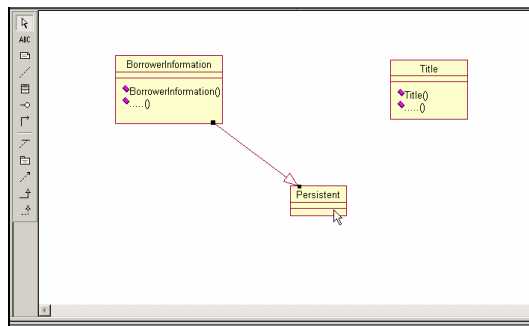


图 4-41 建立类之间的关联示意图 2

用同样的方法，可以创建 Title、ObjId、Item 和 Loan 等类并建立连接，最终的结果如图 4-42 所示。

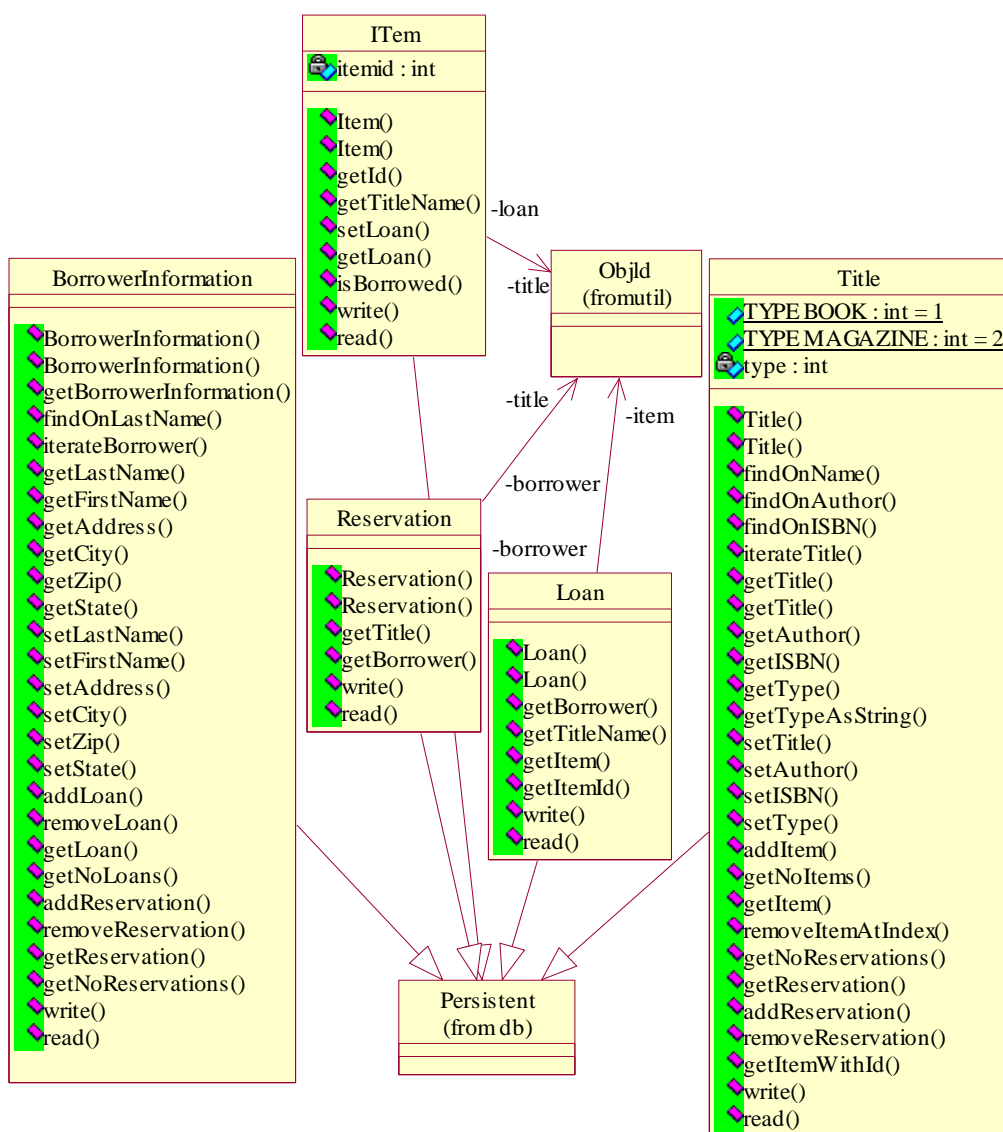


图 4-42 图书馆管理系统的类图



# 第 5 章 用例视图

## 5.1 概述

画好用例图 ( Use Case Diagrams ) 是由软件需求到最终实现的第一步，在 UML 中用例图用于对系统、子系统或类的行为的可视化，以便使系统的用户更容易理解这些元素的用途，也便利软件开发人员最终实现这些元素。

实际中，当软件的用户开始定制某软件产品时，最先考虑的一定是该软件产品功能的合理性、使用的方便程度和软件的用户界面等特性。软件产品的价值通常就是通过这些外部特性动态的体现给用户，对于这些用户而言，系统是怎样被实现的、系统的内部结构如何不是他们所关心的内容。而 UML 的用例视图就是软件产品外部特性描述的视图。用例视图从用户的角度而不是开发者的角度来描述对软件产品的需求，分析产品所需的功能和动态行为。因此对整个软件开发过程而言，用例图是至关重要的，它的正确与否直接影响到用户对最终产品的满意程度。

UML 中的用例图描述了一组用例、参与者以及它们之间的关系，因此用例图包括以下 3 方面内容：

- ( 1 ) 用例 ( Use Case )；
- ( 2 ) 参与者 ( Actor )；
- ( 3 ) 依赖、泛化以及关联关系。

和其他图一样，用例图也可以包含注解和约束。用例图还可以包含包，用于将模型中的元素组合成更大的模块。有时，还可以把用例的实例引入到图中。用例图模型如图 5-1 所示。参与者用人形图标表示，用例用椭圆形符号表示，连线描述它们之间的关系。



图 5-1 用例图

## 5.2 参与者 ( Actor )

参与者 ( Actor ) 是系统外部的一个实体 ( 可以是任何的事物或人 )，它以某种方式参与了

用例的执行过程。参与者通过向系统输入或请求系统输入某些事件来触发系统的执行。参与者由他们参与用例时所担当的角色来表示。

在获取用例前要先确定系统的参与者，可以根据以下的一些问题来寻找系统的参与者。

- (1) 谁或什么使用该系统；
- (2) 交互中，它们扮演什么角色；
- (3) 谁安装系统；
- (4) 谁启动和关闭系统；
- (5) 谁维护系统；
- (6) 与该系统交互的是什么系统；
- (7) 谁从系统获取信息；
- (8) 谁提供信息给系统；
- (9) 有什么事发生在固定事件。

在建模参与者过程中，记住以下要点。

- (1) 参与者对于系统而言总是外部的，因此它们在你的控制之外。
- (2) 参与者直接同系统交互，这可以帮助定义系统边界。
- (3) 参与者表示人和事物与系统发生交互时所扮演的角色，而不是特定的人或特定的事物。
- (4) 一个人或事物在与系统发生交互时，可以同时或不同时扮演多个角色。例如，某研究生担任某教授的助教，同职业的角度看，他扮演了两个角色——学生和助教。
- (5) 每一个参与者需要有一个具有业务一样的名字，在建模中，不推荐使用诸如 NewActor 这样的名字。
- (6) 每个参与者必须有简短的描述，从业务角度描述参与者是什么。
- (7) 像类一样，参与者可以具有分栏，表示参与者属性和它可接受的事件。一般情况下，这种分栏使用的并不多，很少显示在用例图中。

## 5.3 用例 (Use Case)

### 5.3.1 用例的概念

用例是一个叙述型的文档，用来描述参与者 (Actor) 使用系统完成某个事件时的事情发生顺序。用例是系统的使用过程，更确切的说，用例不是需求或者功能的规格说明，但用例也展示和体现出了其所描述的过程中的需求情况。

图形上用例用一个椭圆来表示，用例的名字可以书写在椭圆的内部或下方。用例的 UML 图标如图 5-2 所示。



图 5-2 用例的名字

每个用例都必须有一个惟一的名称以区别于其他用例。用例的名称是一个字符串，它包括简单名（simple）和路径名（path name）。图 5-2 所示左边的用例使用的是简单名。用例的路径名是在用例名前加上它所属包的名称。图 5-2 所示右边的用例使用的是路径名，用例 Maintenance（续借）是属于事务包（Business）的。

### 5.3.2 识别用例

在本章开始已经说明了用例图对整个系统建模过程的重要性，在绘制系统用例图前，还有很多工作要做。系统分析者必须分析系统的参与者和用例，它们分别描述了“谁来做？”和“做什么？”这两个问题。

识别用例最好的办法就是从分析系统的参与者开始，考虑每个参与者是怎样使用系统。使用这种策略的过程中可能会找出一个新的参与者，这对完善整个系统建模很有帮助。用例建模的过程就是迭代和逐步精华的过程，系统分析师从用例的名称开始，然后开始添加用例细节信息。这些信息由初始简短描述组成，它们被精华成完整的规格说明。

在识别用例的过程中，通过以下几个问题可以帮助识别用例：

- （1）特定参与者希望系统提供什么功能；
- （2）系统是否存储和检索信息，如果是，这个行为由哪个参与者触发；
- （3）当系统改变状态时，通知参与者吗；
- （4）存在影响系统的外部事件吗；
- （5）是那个参与者通知系统这些事件。

如图 5-3 所示描述了某仓库管理信息系统的用例模型。通过与系统用户的沟通，需求分析师可以把该软件系统要实现的功能归结为以下几个问题：

- （1）购买的商品入库；
- （2）将积压的商品退给供应商；
- （3）将商品移送到销售部门；
- （4）销售部门将商品移送到仓库；
- （5）管理员盘点仓库；
- （6）供应商提供各种货物；
- （7）用户查询销售部门的营销记录；
- （8）用户查询仓库中的所有变动记录。

通过上述的这些问题，需求分析师可以把本系统所涉及的操作归结为：仓库信息的管理、维护，以及各种信息的分析查询 3 个方面。根据这些分析的结果，可以创建以下参与者。

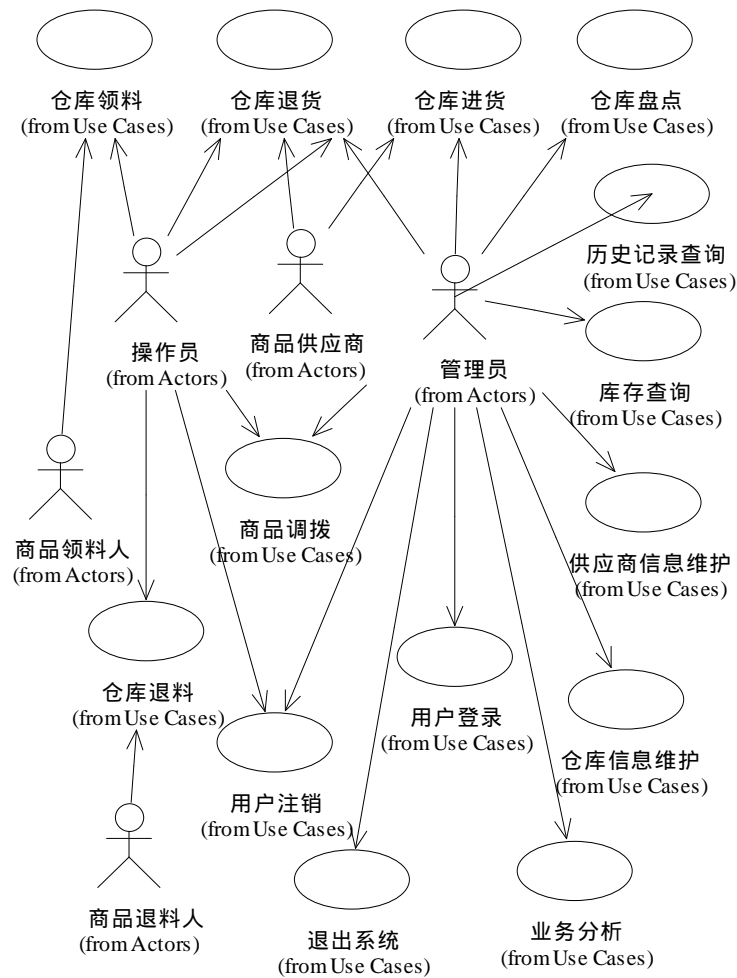


图 5-3 仓库信息系统的用例图

- 操作员
- 管理员
- 供应商
- 商品领料人
- 商品退料人

根据上述的参与者，可以建立如下用例：

- 仓库进货
- 仓库退货
- 仓库领料
- 仓库退料
- 商品调拨
- 仓库盘点
- 库存查询
- 业务分析

- 仓库历史记录查询
- 供应商信息维护
- 仓库信息维护
- 用户登录
- 用户注销
- 退出系统

### 5.3.3 用例与事件流

用例分析处于系统的需求分析阶段，这个阶段应该尽量避免考虑系统实现的细节问题。也就是说，用例描述的是一个系统做什么，而不是怎么做。

可以通过一个清晰的、易被用户理解的时间流来说明一个用例的行为。这个事件流包括用例何时开始和结束，用例何时和参与者交互，什么对象被交互以及该行为的基本流和可选流。

例如，图 5-3 所示的仓库管理信息系统中，用例“用户登录”可采用以下方法。

主事件流：参与者管理员或操作员输入自己的密码时，用例开始。输入的密码被提交后，服务器判断密码是否正确。如果正确，用户成功登录，系统根据用户的类型（管理员或操作员）为其分配相应的权限。

异常事件流：用户密码错误，不能登录，用例重新开始。

异常事件流：在密码提交前，用户清除输入密码，重新填写。

### 5.3.4 用例间的关系

用例除了与其参与者发生关联外，还可以参与系统中的多个关系，这些关系包括：泛化关系、包含关系和扩充关系。应用这些关系是为了抽取出系统的公共行为和变种。

#### 1. 泛化关系（Generalization）

用例间的泛化关系和类间的泛化关系类似，即在用例泛化中，子用例表示父用例的特殊形式。子用例从父用例处继承行为和属性，还可以添加行为或覆盖、改变已继承的行为。当系统中具有一个或多个用例是较一般用例的特化时，就使用用例泛化。

在图形上，用例间的泛化关系用带空心箭头的实线表示，箭头的方向由子用例指向父用例。

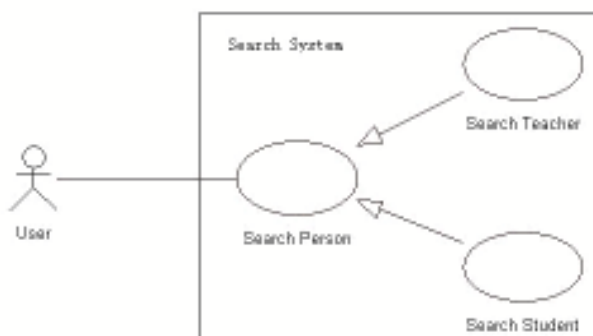


图 5-4 用例间的泛化关系

如图 5-4 所示是某学校信息系统用例图的部分内容。用户“Search Person”负责在学校范围内查找符合用户输入条件的人员信息。该用例有两个子用例“Search Teacher”和“Search Student”。这两个用例都继承了父用例的行为，并添加了自己的行为。它们在查找过程中加入了属于自己的查询范围。

## 2. 包含关系 (Include)

包含关系把几个用例的公共步骤分离成一个单独的被包含用例。用例间的包含关系允许包含提供者用例的行为到用户用例的事件中。把包含用例称为客户用例，被包含用例称为提供者用例，包含用例提供功能给客户用例。

要使用包含关系，就必须在客户用例中说明提供者用例行为被包含的详细位置。这一点和功能调用有点类似，事实上，它们在某种程度上具有相似的语义。

如图 5-5 所示是某学校信息系统用例图的部分内容。用户请求系统做的事件（例如修改个人信息、查看个人信息和删除个人信息）都涉及到找到某个特定的人。如果每次都必须编写事件序列，那么用例会变得很复杂。这里就可以使用包含关系，使用用例 SearchPerson 被用例 ChangePersonDetails、ViewPersonDetails 和 DeletePersonDetails 所包含，这样就能避免许多重复的动作。

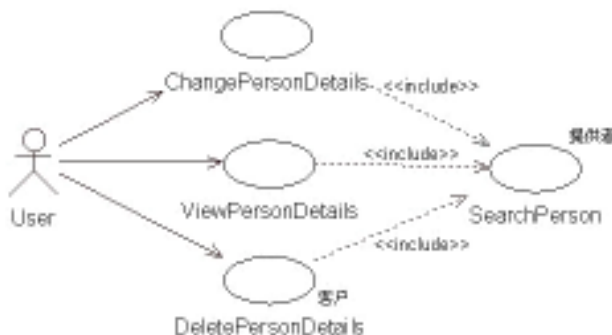


图 5-5 用例间的包含关系

## 3. 扩展关系 (Extend)

扩展关系是把新行为插入到已有用例的方法。基础用例提供了一组扩展点 (Extension points)，在这些扩展点中可以添加新的行为，而扩展用例提供了一组插入片段，这些片段能够被插入到基础用例的扩展点。

基础用例不必知道扩展用例的任何细节，它仅为其提供扩展点。事实上，基础用例没有扩展也是完整的，这一点与包含关系有所不同。一个用例可能有多个扩展点，每个扩展点也可以出现多次。但在一般情况下，基础用例的执行不会涉及扩展用例的行为，如果特定条件发生，扩展用例的行为才被执行，然后流继续。

扩展关系为处理异常或构建灵活系统框架提供了一种有效的方法。

如图 5-6 所示是图书馆信息系统用例图的部分内容。基础用例是 ReturnBook (还书)。如果一切顺利图书可以被归还，但如果借阅人所借图书超期，按规定就要交纳一定数额的罚金，这时就不能执行用例提供的常规动作。如果更改用例 ReturnBook，势必会增加系统的复杂性。

因此可以在用例 ReturnBook 中增加扩充点，特定条件是 OverdueBook（超期），如果满足特定条件，将执行扩展用例 IssueFine（交纳罚金），这样显然能使系统更易被理解。



图 5-6 用例间的扩展关系

## 5.4 用例图建模技术

### 5.4.1 对语境建模

在 UML 建模过程中，可以使用用例图对系统的语境进行建模，强调系统外部的参与者。系统语境是由处于系统外部并且与系统进行交互的事物所构成。语境定义了系统存在的环境。对系统语境建模可以参考如下方法。

(1) 得出需要从系统中得到帮助的组，执行系统功能必需的组，与外界进行交互的组，以及执行某些辅助功能的组，并由此来识别系统外部的参与者。

(2) 将类似的参与者组织成泛化的关系。

(3) 如需加深理解，可以为参与者提供构造型。

(4) 说明用例图中参与者和用例间的通信路径。

例如，图 5-7 所示显示了某大学图书馆信息系统的语境，它强调系统外部的参与者。其中借书人（Borrower）可以泛化成两类：学生（Student）和老师（Teacher）。这样划分是完全有必要的，因为大学中图书馆对老师读者和学生读者的待遇有所不同，老师一般可以借更多的书、享受更长的借阅时间。用例图中还涉及到了图书馆管理员，也参与了借书和还书的过程。

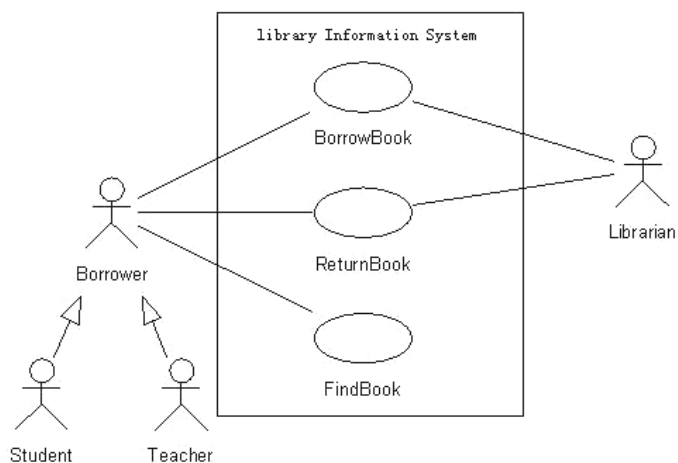


图 5-7 用例图语境建模

### 5.4.2 对需求建模

软件需求就是根据用户对产品功能的期望,提出产品外部功能的描述。所要做的工作是获取系统的需求,归纳系统所要实现的功能,使最终的软件产品最大限度的贴近用户的要求。一般要考虑系统做什么(what),而尽可能的不去考虑怎么做(how)。UML 用例图可以表达和管理系统大多数的功能需求。

对系统功能建模可以参考如下方法:

- (1) 识别系统外部的参与者,从而建立系统的语境;
- (2) 考虑每一个参与者期望的行为或需要系统提供的行为;
- (3) 把公共行为命名为用例;
- (4) 确定供其他用例使用的用例和扩展其他用例的用例;
- (5) 在用例图中对这些用例、参与者和它们间的关系建模;
- (6) 用描述非功能需求的注释修饰用例图。

功能需求实例可以参照图 5-3 以及其分析过程。

## 5.5 实例——图书馆管理系统中的用例视图

下面以比较常见的图书馆管理系统为例,设计一个图书管理系统的用例图。图书管理系统的建模背景将在第 9 章中有详细的说明。

### 5.5.1 确定系统涉及的内容

图书管理系统用于对书籍的借阅以及对读者信息进行统一的管理,凡是有关这些操作的内容都属于系统的范围,比如读者要借书、还书、预留书籍,工作人员查看读者信息,查看书籍信息,等等。



### 5.5.2 确定系统参与者

对参与者的确定，需要分析系统涉及的问题领域，明确系统运行的主要任务。根据图书管理系统的需求分析，可以得到如下任务：

- 读者要借书籍
- 读者要还书籍
- 读者要预留书籍
- 读者要撤销预留书籍
- 工作人员根据读者要求提供服务
- 工作人员进行查询，修改信息

这个用例图的参与者严格上有两个，一个是图书馆工作人员，一个读者，而实际系统使用的主要操作者是图书馆工作人员，读者没有操作系统的权限，只是向工作人员提供请求服务的信息。

### 5.5.3 确定系统用例

Use Case 是参与者与系统在交互过程中所需要完成的事务。一个完整的需求分析，要求必须找出所有的用例。这里分析最主要的 3 个部分：读者请求服务的用例图；工作人员维护读者信息、书籍信息的用例图；工作人员登录查询信息的用例图，分别如下所示。

#### (1) 读者请求服务的用例图

- 还书
- 借书
- 预留书籍
- 取消预留

#### (2) 工作人员维护读者信息、书籍信息的用例图

- 增加书目
- 删除书目
- 增加书籍
- 删除或更新书籍信息
- 增加读者
- 删除或更新读者信息

#### (3) 工作人员登录查询信息的用例图

- 登录
- 查看书籍信息
- 查看读者信息

### 5.5.4 用 Rational Rose 来绘制用例图

首先，在 Rose 的树形列表中的 Use Case 包的图标上单击鼠标右键，在弹出的快捷菜单中，

最常用的功能是“Open Specification”(打开属性说明)和“New”(新建 UML 元素)。其中,“Open Specification”可以打开当前选定对象的属性和说明,并对其进行具体的修改和更新。而“New”则可以新建包括 Package(包)、Use Case(用例)、Class(类)、Actor(角色)、Use Case Diagram(用例图)、Class Diagram(类图)、Collaboration Diagram(协作图)、Sequence Diagram(时序图)、State Chart Diagram(状态图)和 Activity Diagram(活动图),这里选择 Use Case Diagram(用例图),如图 5-8 所示。

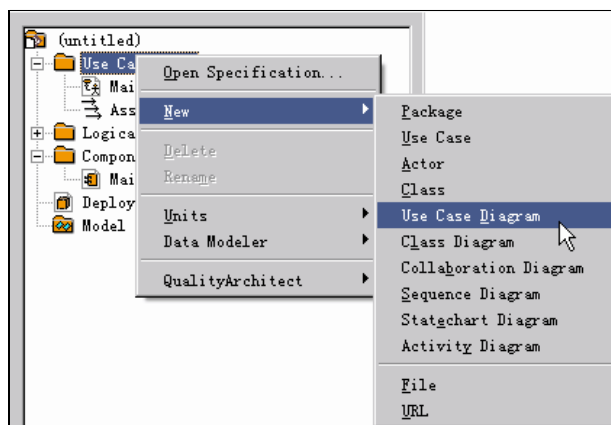


图 5-8 新建用例图

这样,一个用例图就新建完成了,如图 5-9 所示,修改用例图的名称为 UseCaseDiagram。双击打开此图,编辑工具栏又有所变化,如图 5-10 所示。

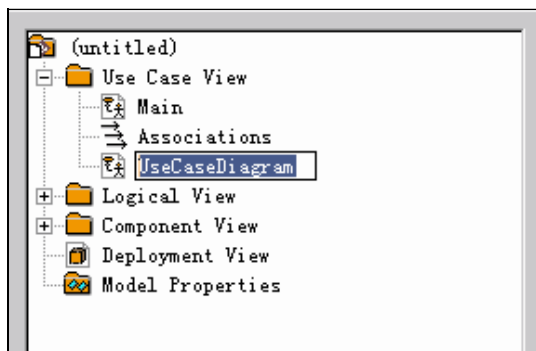


图 5-9 修改用例图名称



图 5-10 编辑工具栏

编辑工具栏是可以自己设定的,选择菜单“Views Toolbars Configure...”,在窗口中选择 Use Case 可以自定义用例图的工具栏(如图 5-11 所示)内容,读者可以根据自己的需要来设定,默认工具栏中是最常用的内容。



图 5-11 自定义工具栏

在视图界面上绘制相应的用例图，如图 5-12 所示。

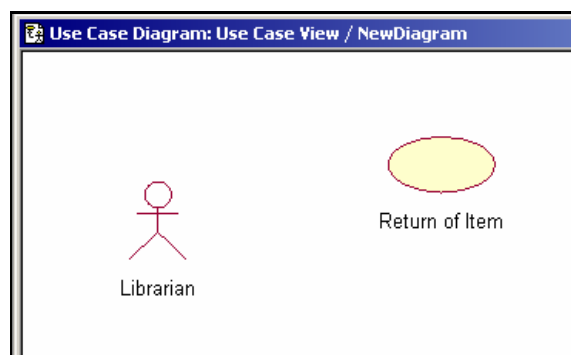


图 5-12 用例图绘制过程示意图 1

在用例上单击鼠标右键，在弹出菜单中选择“Open Specification...”，可以对用例的属性进行设定，如图 5-13 所示。

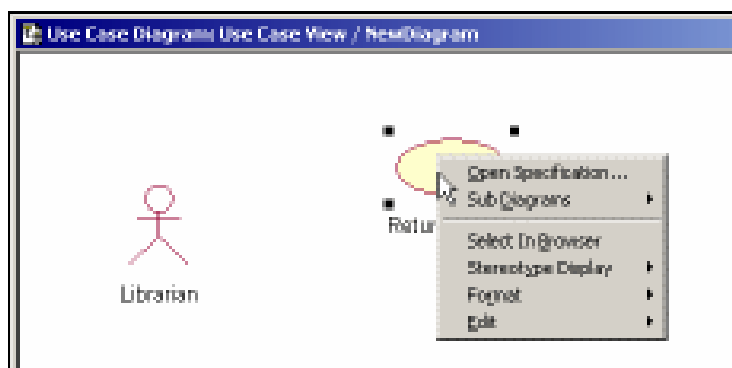


图 5-13 用例图绘制过程示意图 2

根据属性设置的各选项（如图 5-14 所示），读者可以添上需要设定的内容，包括名称、类型、层次和说明部分等。

名称：用例的名称。


类型：用例的类型，包括 business use case、business use-case realization 和 use-case realization 3 种类型，最常用的是 business use case。

层次：用例的分层，越是底层着眼点越小，越接近计算机解决问题的水平，反之更抽象。

说明：描述用例图的文档。



图 5-14 属性设置标签

设定参与者与用例的联系箭头，点击工具栏上的 ，如图 5-15 所示。

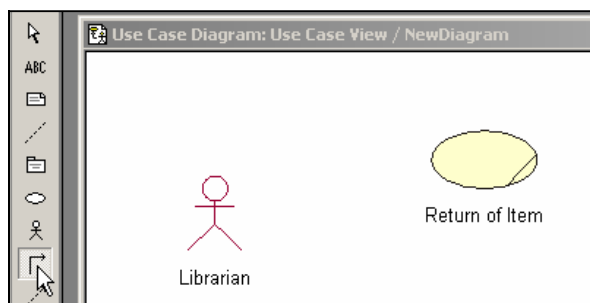


图 5-15 用例图绘制过程示意图 3

连接参与者与用例，读者可以参考用例属性的设置来设定箭头的属性，如图 5-16 所示。

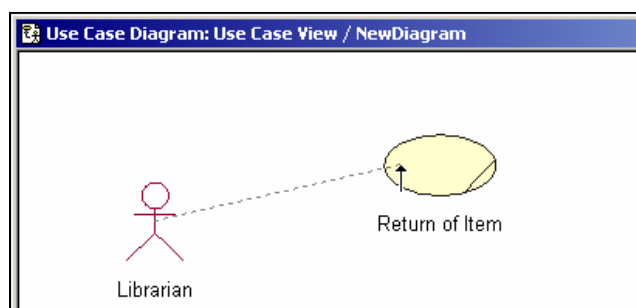


图 5-16 用例图绘制过程示意图 4

根据图书管理系统的参与者，以及用例分析，相应可以得到 3 张用例图，如图 5-17、图 5-18 和图 5-19 所示。

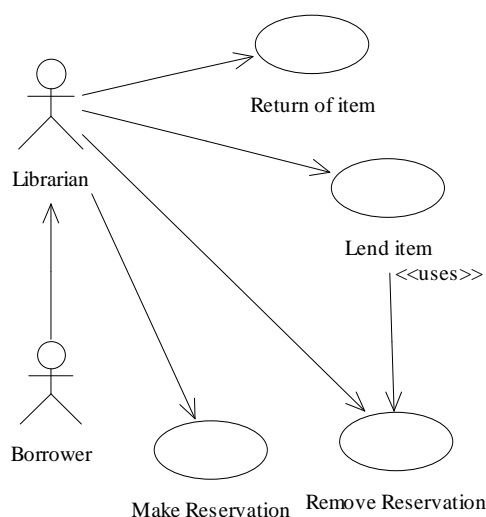


图 5-17 读者得到服务的用例图

注释：

Return of Item：还书用例。

Lend Item：借书用例。

Make Reservation：删除预留书籍用例。

Remove Reservation：删除预留书籍用例。

服务用例图包括 4 个用例，其中服务的信息是从借书者传递给管理员的，而借书跟删除预留之间存在<<uses>>的关系。

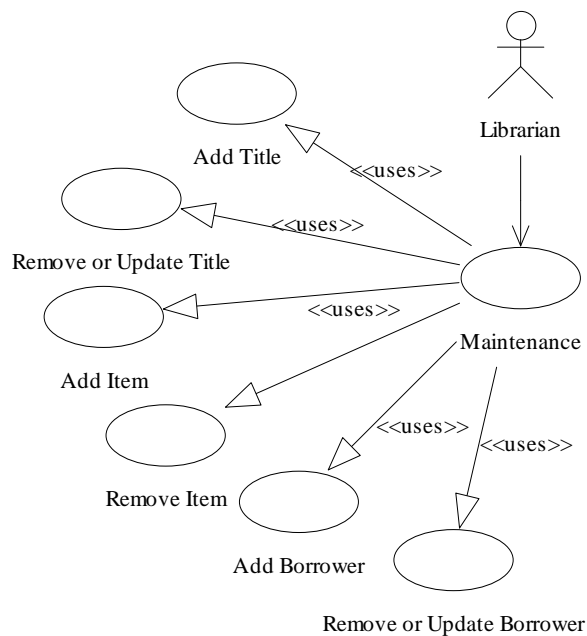


图 5-18 管理员维护读者/书籍信息的用例

注释：

Add Title：增加书籍用例。

Remove or Update Title：删除或者更新书籍信息用例。

Add Item：增加数目信息用例。


Remove Item：删除数目信息用例。

Add Borrower：增加读者信息用例。

Remove or Update Borrower：删除或者更新读者信息用例。

Maintenance：维护用例。

Librarian (图书馆工作人员)：参与者。

在上述的维护用例中有 6 个子内容，用 “” 来说明它们之间的关系，表示用例的泛化。

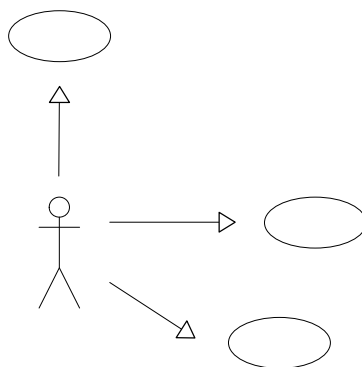


图 5-19 管理员登录，查询用例

注释：

Login：登录用例。

Seek Title：查询书籍用例。

Seek Borrower：查询读者信息用例。

---

注意：构造用例图时，不需要关心系统的实现问题，每一个用例的内部实现细节不是本阶段要考虑的问题。用例视图的最终目的是说明系统对于用户来说是什么样子的，因此，能否正确反映用户的需求是用例视图的关键所在。

---

## 第 6 章 动态视图

在建好系统静态模型的基础上,接下来需要分析和设计系统的动态结构,并且建立相应的动态模型。动态模型描述了系统随时间变化的行为,这些行为是用从静态视图中抽取的系统的瞬间值的变化来描述的。在 UML 的表现上,动态模型主要是建立系统的交互图和行为图。交互图包括时序图和协作图;行为图则包括状态图和活动图。

时序图用来显示对象之间的关系,并强调对象之间消息的时间顺序,同时显示对象之间的交互。协作图主要用来描述对象间的交互关系。状态图通过对类对象的生存周期建立模型来描述对象随时间变化的动态行为。活动图是一种特殊形式的状态机,用于对计算流程和 workflows 建模。

### 6.1 时序图 (Sequence Diagram)

#### 6.1.1 时序图的概念和内容

交互图 (Interaction Diagram) 描述了一个交互,它由一组对象和它们之间的关系组成,并且还包括在对象间传递的信息。时序图 (Sequence Diagram) 是强调消息时间顺序的交互图。时序图描述类系统中类和类之间的交互,它将这些交互建模成消息交换,也就是说,时序图描述了类以及类间相互交换以完成期望行为的消息。

UML 中,图形上参与交互的各对象在时序图的顶端水平排列,每一个对象的底端都绘制了一条垂直虚线,当一个对象向另一个对象发送消息时,此消息开始于发送对象底部的虚线,终止于接收对象底部的虚线,这些消息用箭头表示,水平放置,沿垂直方向排列,在垂直方向上,越靠近顶端的消息越早被发送。当对象收到消息后,此对象把消息当作执行某种动作的命令。因此,可以这样理解,时序图向 UML 用户提供了事件流随时间推移的、清晰的和可视化的轨迹,如图 6-1 所示。

时序图中包括如下元素:类角色、生命线、激活期和消息。

##### (1) 类角色 (Class Role)

类角色代表时序图中的对象在交互中所扮演的角色,如图 6-1 所示位于时序图顶部的对象代表类角色。类角色一般代表实际的对象。

##### (2) 生命线 (Lifeline)

生命线代表时序图中的对象在一段时期内的存在。如图 6-1 所示,每个对象底部中心都有一条垂直的虚线,这就是对象的生命线,对象间的消息存在于两条虚线间。



### (3) 激活期 (Activation)

激活期代表时序图中的对象执行一项操作的时期。如图 6-1 所示每条生命线上的窄的矩形代表活动期。激活期可以被理解成 C 语言语义中一对花括号 “{}” 中的内容。

### (4) 消息 (Message)

消息是定义交互和协作中交换信息的类,用于对实体间的通信内容建模。信息用于在实体间传递信息,允许实体请求其他的服务,类角色通过发送和接收信息进行通信。

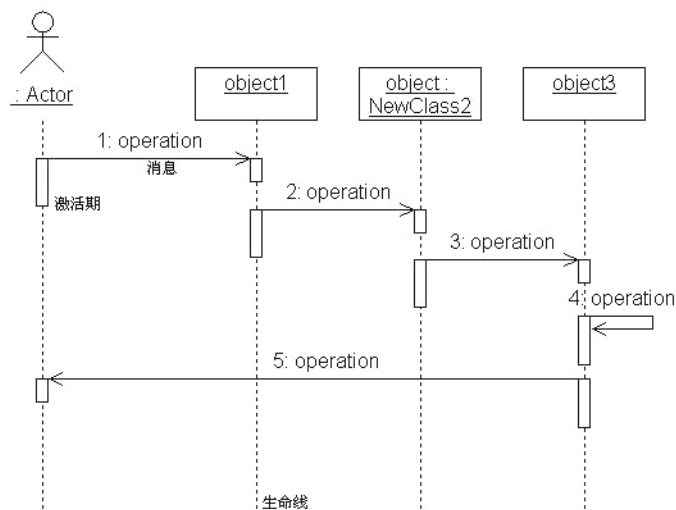


图 6-1 时序图

## 6.1.2 时序图的用途

时序图强调按时间展开的消息传送,这在一个用例脚本的语境中对动态行为的可视化非常有效。

UML 的交互图是用于对系统的动态方面的建模,交互图又可分为时序图和协作图。虽然在本章开始提及了时序图用于描述对象之间消息的时间顺序,协作图用于描述对象间的交互关系,但读者可能问这两者在特性上有什么不同,以致它们的用途有所差别。以下是时序图有别于协作图的特性。

### (1) 时序图有生命线

生命线表示一个对象在一段时期内的存在,正是因为这个特性,使时序图适合对象之间消息的时间顺序。一般情况下,对象的生命线从图的顶部画到底部,这表示对象存在于交互的整个过程,但对象也可以在交互中创建和撤销,它的生命线从接收到“create”消息开始到接收到“destroy”消息结束。这一点是协作图所不具备的。

### (2) 时序图有激活期

激活期代表一个对象直接或间接的执行一个动作的时间,激活矩形的高度代表激活持续时间。时序图的这个特性可视化地描述了对象执行一项操作的时间,显然这个特性使系统间对象的交互更容易被理解。这也是协作图所不能提供的。

### 6.1.3 时序图的建模技术

对系统动态行为建模，当强调按时间展开信息的传送时，一般使用时序图。但一个单独的时序图只能显示一个控制流。一般来说，一个完整的控制流肯定是复杂的，因此可以新建许多交互图（包括若干时序图和交互图），一些图是主要的，另一些图用来描述可选择的路径和一些例外，再用一个包对它们进行统一的管理。这样就可以用一些交互图来描述一个冗大复杂的控制流。

使用时序图对系统建模时，可以遵循如下策略。

- (1) 设置交互的语境，这些语境可以是系统、子系统、操作、类、用例和协作的一个脚本。
- (2) 通过识别对象在交互中扮演的角色，根据对象的重要性，将其从左向右的方向放在时序图中。
- (3) 设置每个对象的生命线。一般情况下，对象存在于交互的整个过程，但它也可以在交互过程中创建和撤销。
- (4) 从引发某个交互的信息开始，在生命线之间按从上向下的顺序画出随后的消息。
- (5) 设置对象的激活期，这可以可视化实际计算发生时的时间点、可视化消息的嵌套。
- (6) 如果需要设置时间或空间的约束，可以为每个消息附上合适的时间和空间约束。
- (7) 给某控制流的每个消息附上前置或后置条件，这可以更详细地说明这个控制流。

如图 6-2 所示的时序图描述了某信用卡客户使用 ATM 提款的过程。

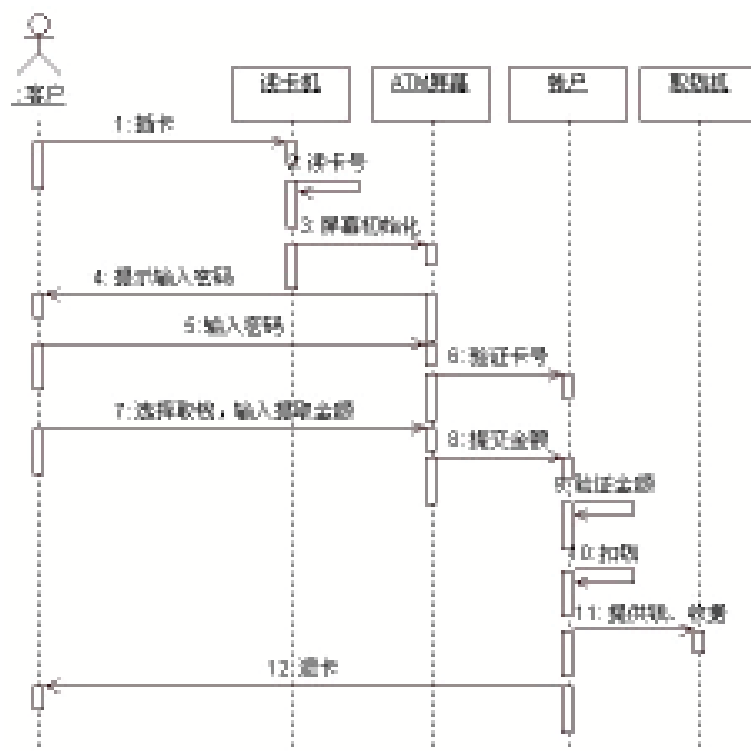


图 6-2 ATM 取钱过程时序图

这张时序图描述涉及了4个对象：客户、读卡机、ATM 屏幕、客户的账户和取钱机。取钱动作从用户将卡插入读卡机开始，读卡机读卡号，打开张三的账目对象，并初始化屏幕。屏幕提示输入用户密码，张三输入其密码，然后屏幕验证密码与账户对象，发出相符合的信息。屏幕向张三提供选项，张三选择取钱，并在屏幕的提示下输入提取金额。ATM 机开始验证用户账户金额，验证通过后在其账户扣取相应金额并提供现金，最后是退卡。

## 6.2 协作图 ( Collaboration Diagram )

### 6.2.1 协作图的概念和内容

协作图是动态视图的另一种表现形式，它强调参加交互的各对象的组织。协作图只对相互间有交互作用的对象和这些对象间的关系建模，而忽略了其他对象和关联。协作图可以被视为对象图的扩展，但它除了展现出对象间的关联外，还显示出对象间的消息传递。

UML 中，图形上交互图的对象用矩形表示，矩形内是此对象的名字，连接用对象间相连的直线表示，连线可以有名字，它标于表示连接的直线上。如果对象间的连接有消息传递，则把消息的图标沿直线方向绘制，消息的箭头指向接受消息的对象。由于从图形上绘制的协作图无法表达对象间消息发送的顺序，因此需要在消息上保留对应时序图的消息顺序号，如图 6-3 所示。

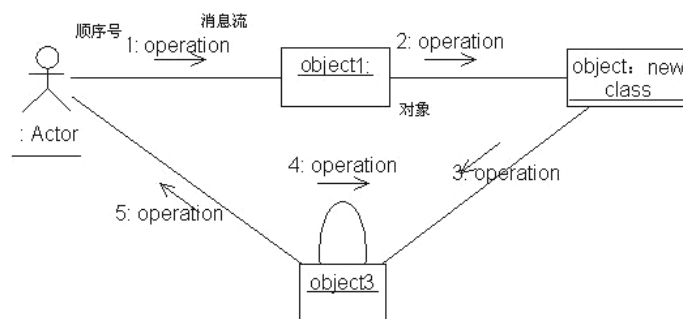


图 6-3 协作图

时序图中包括如下元素：类角色、关联角色和消息流。

#### (1) 类角色 ( Class Role )

类角色代表协作图中对象在交互中所扮演的角色。如图 6-3 所示，矩形中的对象代表类角色。类角色的代表参与交互的对象，它的命名方式和对象的命名方式一样。

#### (2) 关联角色 ( Association Role )

关联角色代表协作图中连接在交互中所扮演的角色。如图 6-3 所示，连接（即连线或路径）代表关联角色。

#### (3) 消息流 ( Message Flow )

消息流代表协作图中对象间通过链接发送的消息。如图 6-3 所示，类角色之间的箭头表明在对象间交换的消息流，消息由一个对象发出由消息所指的对象接收，链接用于传输或实现消

息的传递。消息流上标有消息的序列号和类角色间发送的消息。一条消息会触发接受对象中的一项操作。

### 6.2.2 协作图的用途

如果按组织对控制流建模,应该选择使用协作图。协作图强调交互中实例间的结构关系以及所传送的消息。协作图对复杂的迭代和分支的可视化以及对多并发控制流的可视化要比时序图好。

时序图和协作图都可用于对系统动态方面的建模,而协作图更强调参加交互的各对象的组织。以下是协作图有别于时序图的两点特性。

#### (1) 协作图有路径

为了说明一个对象如何与另一个对象链接,可以在链的末路上附上一个路径构造型。例如构造型<<local>>,它表示指定对象对发送者而言是局部的。

#### (2) 协作图有顺序号

为了描述交互过程中消息的时间顺序,需要给消息添加顺序号。顺序号是消息的一个数字前缀,它是一个整数,由1开始递增,每个消息都必须有惟一的顺序号。可以通过点表示法代表控制的嵌套关系,也就是说在激活期1中,消息1.1是嵌套在消息1中的第一个消息,它在消息1.2之前,消息1.2是嵌套在消息1中的第2个消息,它在消息1.3之前。嵌套可以有任意深度。与时序图相比,协作图能显示更为复杂的分支。

### 6.2.3 协作图的建模技术

对系统动态行为建模,当按组织对控制流建模时,一般使用协作图。像时序图一样,一个单独的协作图只能显示一个控制流。当要描述系统的复杂的控制流时,可以新建许多协作图,一些图是主要的,另一些图用来描述可选择的路径和一些例外,再用一个包对它们进行统一的管理。这样可以使描述有合理明确的结构。

使用协作图对系统建模时,可以遵循如下策略。

(1) 设置交互的语境,语境可以是系统、子系统、操作、类、用例或用例的脚本。

(2) 通过识别对象在交互中所扮演的角色,开始绘制协作图,把这些对象作为图的顶点放在协作图中。

(3) 在识别了协作图对象后,为每个对象设置初始值。如果某对象的属性值、标记值、状态或角色在交互期发生变化,则在图中放置一个复制对象,并用变化后的值更新它,然后通过构造型<<become>>或<<copy>>的消息将两者连接。

(4) 设置了对对象的初始值后,根据对象间的关系开始确定对象间链接。一般先确定关联的链接,因为这是最主要的,它代表了结构的链接。然后需要确定的是其他的链接,用合适的路径构造型修饰它们,这表达了对象间是如何互相联系的。

(5) 从引起交互的消息开始,按消息的顺序,把随后的消息附到适当的链接上,这描述了对象间的消息传递,可以用带小数点的编号来表达嵌套。

(6) 如果需要说明时间或空间的约束,可以用适当的时间或空间约束来修饰每个消息。

(7)在建模中,如果想更详细地描述这个控制流,可以为交互过程中的每个消息都附上前置条件和后置条件。

如图 6-4 所示的协作图描述了某连锁企业对其分店的管理。

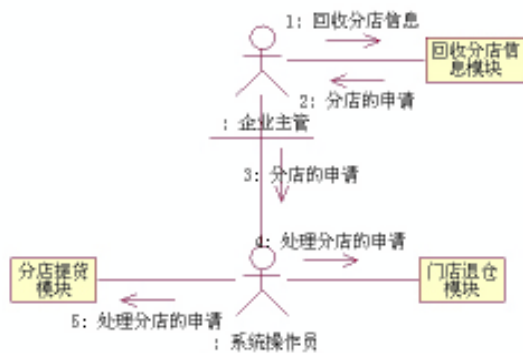


图 6-4 分店管理协作图

管理过程从企业主管开始,他向回收分店信息模块发送回收分店信息的信息,该模块在收到此消息后,回复给企业主管分店的申请。该申请可以是公司提取分店库存不足的货物,也可以是推给公司分店库存过量的货物。企业主管接收到消息后将消息提交给系统操作员。操作员进行相应的操作来处理分店的申请。

#### 6.2.4 协作图与时序图的互换

协作图 and 时序图都是表示对象间的交互作用,只是它们侧重点有所不同。时序图描述了交互过程中的时间顺序,但没有明确的表达对象间的关系,协作图描述了对对象间的关系,但时间顺序必须从序列号获得。协作图 and 时序图都来自 UML 元模型的相同信息,因此它们的语义是等价的,它们可以从一种形式的图转换成另一种形式的图,而不丢失任何信息。

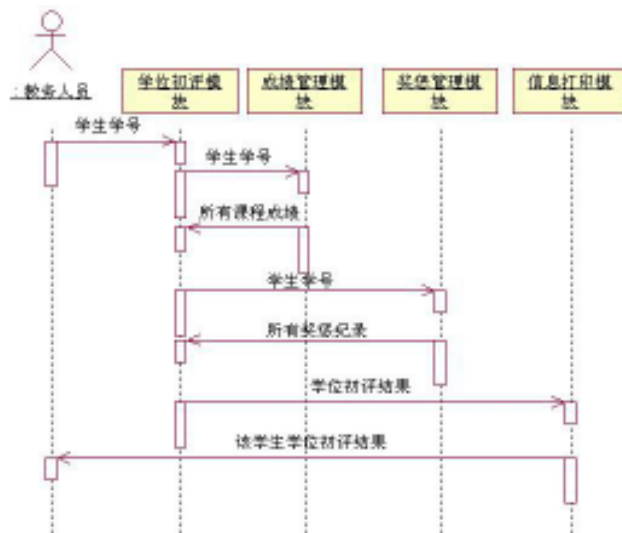


图 6-5 毕业管理时序图

图 6-5 所示是学生信息系统中毕业管理的时序图，它可以转换成图 6-6 所示的协作图。两者所描述的控制流相同，只是所强调的内容有所不同。学生学位评审的流程如下：教务人员将需评审的学生的学号输入学位初评模块，学位初评模块会查询相应学生的所有成绩和奖惩记录来作为学位评定的依据。学位初评模块将初评的结果打印，学位初评打印稿被提交给教务人员，控制流结束。

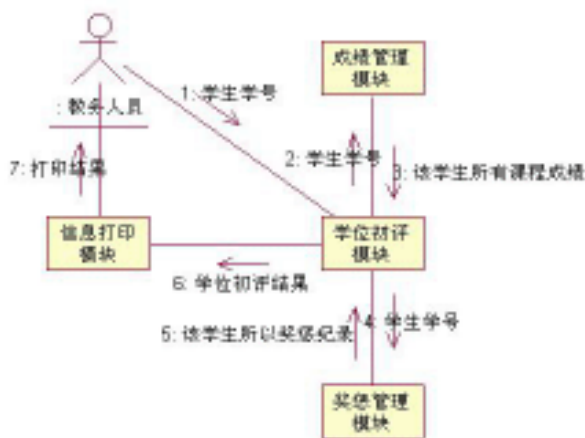


图 6-6 毕业管理协作图

## 6.3 状态图 (Statechart Diagram)

在系统分析员对某对象建模时，最自然的方法并不是着眼于从活动到活动的控制流，而是着眼于从状态到状态的控制流。例如，按下电灯的开关，电灯改变了它的状态；拉上卧室的窗帘，卧室里亮度的状态由亮变暗，等等。系统中对象状态的变化是最容易被发现和理解的，因此在 UML 中，可以使用状态图展现对象状态的变化。

### 6.3.1 状态图的概念和内容

状态图是 UML 中对系统动态方面建模的图之一。状态图是通过类对象的生命周期建立模型来描述对象随时间变化的动态行为。状态图显示了一个状态机，它基本上是一个状态机中的元素的一个投影，这也就意味着状态图包括状态机的所有特性。

状态图是一种特殊种类的图形，它拥有所有其他图一样的公共特性，即名称和投影在一个模型上的图形。状态图和其他图的区别在于它的内容。状态图通常包括如下内容。

#### (1) 状态

状态定义对象在其生命周期中的条件或状况，在此期间，对象满足某些条件，执行某些操作或等待某些事件。状态用于对实体在其生命中状况建模。

## (2) 转换

转换包括事件和动作。事件是发生在时间空间上的一点值得注意的事情。动作是原子性的，它通常表示一个简短的计算处理过程（如赋值操作或算术计算）。

在 UML 中，图形上每一个状态图都有一个初始状态（实心圆），用来表示状态机的开始，还有一个终止状态（半实心圆），用来表示状态机的终止，其他的状态用一个圆角的矩形表示。转换表示状态间可能的路径，用箭头表示，事件写在由它们触发引起的转换上。具体状态图模型如图 6-7 所示。

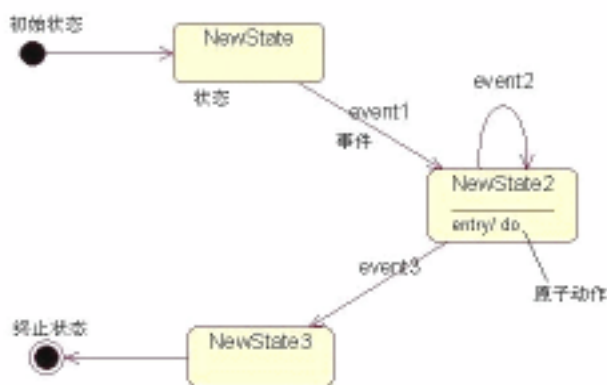


图 6-7 状态图

### 1. 状态机 (State Machine)

在 UML 里，状态机用于对具有事件驱动的特性的动态行为建模。事件驱动的动态行为是对象当前时刻的动态行为，将取决于当前的事件输入和此对象在以前时刻的动态行为的结果。

状态机是展示状态与状态转换的图。通常一个状态机依附于一个类，并且描述这个类的实例对接收到的事物的反应。对于这个类而言，一般只需要一个状态机就能够对该类中所有对象对事件响应而引起的状态迁徙建模。虽然有时，对象对时间的响应而产生内部事件，但在典型情况下，事件都是由其他对象作为消息传递的。状态机也可依附于操作、用例和协作并描述它们的执行过程。

状态机是一个对象的局部视图，它将一个对象与其外部世界分离开来并独立考查其行为。状态机可以精确地描述对象行为，这有助于系统建模人员理解例如用户接口、设备控制器这样的控制机，但它并不适合表达系统执行操作，如果需要描述系统内对象行为产生的影响，使用交互视图更为适合。总而言之，可以使用状态机对类、用例、子系统或是整个系统的动态行为建模。

可以有两种方法来可视化状态机：如果强调从活动到活动的控制流，一般使用活动图（将在 6.4 节中做具体的介绍）；如果强调对象的潜在状态和这些状态间的转换，一般使用状态图。

### 2. 状态 (State)

状态是状态机的重要组成部分，它描述了状态机所在对象动态行为的执行所产生的结果。这里的结果一般是指能影响此对象对后续事件响应的结果。状态用于对对象在其生命中的状况建模，在这些状况下状态可以满足某些条件、执行某些操作或等待某些事件。

图形上，使用一个圆角矩形表示一个状态（可参照图 6-7）。一个完整的状态有 5 个组成部分。

#### （1）名字（name）

状态的名字由一个字符串构成，用以识别不同的状态。状态可以是匿名的，即没有名字。状态名一般放置在状态图符的顶部。

#### （2）入口/出口动作（entry/exit action）

入口/出口动作表示进入/退出这个状态所执行的动作。入口动作的语法是 entry/执行的动作；出口动作的语法是 exit/执行的动作。这里所指的动作可以是原子动作，也可以是动作序列（action sequence）。

#### （3）内部转换（Internal Transition）

内部转换是不会引起状态变化的转换，此转换的触发不会导致状态的入口/出口动作被执行。定义内部转换的原因是有时候入口/出口动作显得是多余的。例如：某状态的入口/出口分别是打开/关闭某文件，但如果用户仅仅是想更改该文件的文件名，那么，这里所定义的入口/出口动作显得多余，这时就可以使用内部转换，而不触发入口/出口动作的执行。

在图形表示上，由于内部转换不引起状态的转变，因此它的文字标识被附加在表示状态的圆角矩形内部，而不使用箭头进行图形标识。

内部变迁的语法是：事件/动作表达式

#### （4）延迟事件（Deferred Event）

延迟事件该状态下暂不处理，但将推迟到该对象的另一个状态下事件处理队列。也就是所延迟事件是事件的一个列表，此列表内的事件当前状态下不会处理，在系统进入其他状态时再处理。

具有某些动态行为的对象在运行过程中，某个状态下，总会有一些事件被处理，而另一些事件不能被处理。但对于这个对象来说，有些不能被处理的事件是不可以被忽略的，它们会以队列的方式被缓存起来，等待系统在合适的状态下在处理它们。对于这些被延迟的事件，可以使用状态的延迟事件来建模。

#### （5）子状态（Substate）

在复杂的应用中，当状态机处于某特定的状态时，状态机所在的对象在此刻的行为还可以用一个状态机来描述，也就是说，一个状态内部还包括其他状态。在 UML 里，子状态被定义成状态的嵌套结构，即包含在某状态内部的状态。

比如图书馆信息系统的图书查询，目前绝大多数的信息系统都采用 C/S（客户机/服务器）结构，客户机提供用户查询的接口，而图书数据是保存在服务器上。当某用户在客户机上输入查询条件，客户机的查询状态又可以分成 3 个子状态：发送查询信息给主机、等待主机回复和显示查询结果。

在 UML 里，包含子状态的状态被称为复合状态（Composite State），不包含子状态的状态被称为简单状态（Simple State）。子状态以两种形式出现：顺序子状态和并发子状态。

##### 顺序子状态（Sequential Substate）

如果一个复合状态的子状态对应的对象在其生命期内任何时刻都只能处于一个子状态，即不会有多个子状态同时发生的情况，这个子状态被称为顺序子状态。

当状态机通过转换从某状态转入复合状态时，此转换的目的可能是这个复合状态本身，也



可能是复合状态的子状态。如果是前者,状态机所指的对象首先执行复合状态的入口动作,然后子状态进入初始状态并以此为起点开始运行。如果此转换的目的是复合状态的子状态,复合状态的入口动作首先被执行,然后复合状态的内嵌状态机以此转换的目标子状态为起点开始运行。顺序子状态模型如图 6-8 所示。

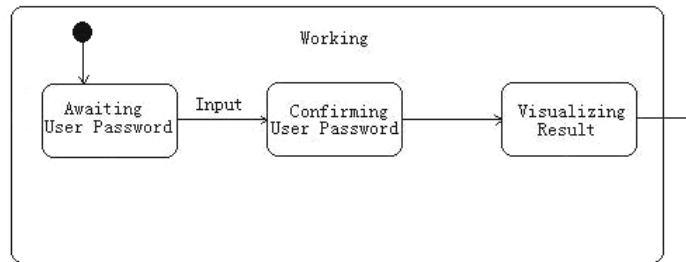


图 6-8 顺序子状态

图 6-8 所示描述的是确认用户密码状态下的 3 个子状态,显然它们是一个接一个发生的。

Awaiting User Password: 等待用户输入密码;

Confirming User Password: 确认用户密码;

Visualizing Result: 显示确认结果。

如果复合状态的内嵌状态机执行过程被中断,如果不指定,下次再进入此复合状态的内嵌状态机将从初始状态开始运行。如果希望下一次内嵌状态机从上次中断点开始运行,可以在建模时设置一个特殊的状态——历史状态 (History state),它可以记录复合状态转出时的正在运行的子状态。

#### 并发子状态 (Concurrent Substate)

如果复合状态内部只有顺序子状态,那么这个复合状态机只有一个内嵌状态机。但有时可能需要在复合状态中有两个或多个并发执行的子状态机。这时,称复合状态的子状态为并发子状态。

顺序子状态与并发子状态的区别在于后者在同一层次给出两个或多个顺序子状态,对象处于同一层次中来自每个并发子状态的一个时序状态中。当一个转换所到的组合状态被分解成多个并发子状态组合时,控制就分成与并发子状态对应的控制流。在两种情况下控制流会汇合成一个:第一,当一个复合状态转出的转移被激发时,所有的内嵌状态机的运行被打断,控制流会汇合成一个,对象的状态从复合状态转出;第二,每个内嵌状态机都运行到终止状态。这时,所有的内嵌状态机的运行被打断,但对象还处于复合状态。

并发子状态的模型如图 6-9 所示。

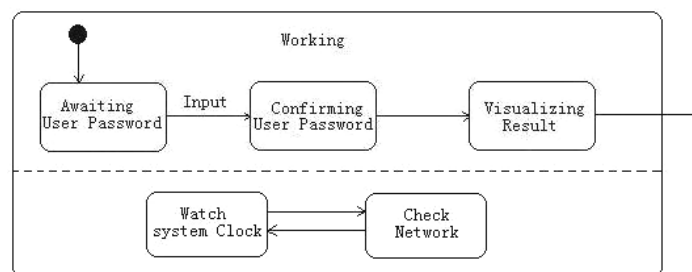


图 6-9 并发子状态

系统在 Working 状态时，并不是仅仅做密码确认的工作，它还要监视系统时间（Watch system Clock），并且定时检查网络连接，以防止网络发生无响应等错误，因为用户的信息都存储在远程的主机上。这也提供了系统运行的稳定性。

### 3. 转换

状态图通过对对象的状态以及状态间的转换建模来展现系统动态行为。

转换是状态间的关联。它们用于对一个实体的不同状态间的关系建模。当某实体在第一个状态中执行一定的动作，并在某个特定事情发生并且某个特定的条件满足时进入下一个状态。在 UML 里，转换由 5 个部分组成，它们分别是：源状态、目标状态、触发事件、监护条件和动作。

#### （1）源状态（Source State）

转换描述的是状态机所在的对象的状态的变化（状态图是可视化状态机的一种方式）。转换使对象从某个状态转换到另一个状态。那么在转换被激发之前，对象所处的状态就是转换的源状态。源状态就是被转换影响的状态。某对象处于源状态，当它接收到触发事件或满足监护条件，就会激活一个转换。

一个转换可以有多个源状态，这表示状态机所在对象中的多个控制流在转换发生时汇合成一个控制流。在 UML 中，多源状态的转换通常使用活动图表示。

#### （2）目标状态（Target State）

转换使对象从一个状态转换到另一个状态。转换完成后，对象状态发生了变化，这时对象所处的状态就是转换的目标状态。目标是转换完成后活动的状态。在图形上，转换的源状态位于表示转换的箭头的起始位置。转换的目标状态位于表示转换的箭头所指的那个状态。在这里要特别注意把源状态、目标状态的概念与状态图的初始状态、终止状态的概念区别开来。

同样，一个转换可以有多个目标状态，这表示状态机所在的对象在转换被激活的时刻一个控制流分解为多个控制流。在 UML 中，多目标状态的转换通常使用活动图表示。

#### （3）触发事件（Trigger Event）

状态机描述了对对象的具有事件驱动的动态行为。在这些动态行为中，对象动作的执行、状态的改变都是以特定事件的发生为前提的。转换的触发事件就是引起转变的事件。这里所指的事件可以是信号、调用、时间段或状态的一个改变。一个信号或调用可以带有参数，参数值可以由监护条件和动作的表达式的转换得到。

在 UML 中还可能有无触发转换，它不需要事件触发，一般当它的源状态已经完成它的活动时，无触发转换为被触发。

#### （4）监护条件（Guard Condition）

转换可能具有一个监护条件。监护条件是一个方括号括起来的布尔表达式，它被放在触发事件的后面。监护条件可以引用对象的属性值和触发事件的参数。当一个触发事件被触发时，布尔表达式被赋值。如果值是“真”，则触发事件使转换有效。如果值是“假”，则不会引起转换。

监护条件只在引起转换的触发事件发生时被赋值一次，如果此转换被重新触发，监护条件会被重新赋值。

#### （5）动作（Action）

当转变被激活时，它对应的动作被执行。动作是一个可执行的原子计算，它可以包括操作调用、另一个对象的创建或撤销、向一个对象发送信号。动作也可以是一个动作序列，即包括一序列的简单动作。动作或动作序列的执行不会被同时发生的其他动作所影响。

根据 UML 的概念，动作的执行时间是非常短的，与外界的时间相比几乎可以忽略，因此在动作执行过程中不允许被中断，这点正好与活动相反，活动是可以被其他事件中断的。在某动作执行时，一般新进的事件会被安排在一个等待队列里。

### 6.3.2 状态图的用途

状态图用于对系统的动态方面建模 动态方面指出现在系统体系结构中任一对象按事件排序的行为，其中这些对象可以是类、接口、构件和节点。当使用状态图对系统建模时，可以在类、用例、子系统或整个系统的语境中使用状态图。

前面曾经提到，状态机是展示状态与状态转换的图，可以使用状态图和活动图这两种方法来可视化状态机。这就涉及到一个问题，在对用例、类或系统建模时，在什么样的情况下使用状态图呢？根据状态图在 UML 中的定义，对反应型对象建模一般使用状态图。反应型对象是指一个为状态图提供语境的对象。反映型对象通常具有如下特点：

- (1) 响应外部事件，即来自对象语境外的事件；
- (2) 具有清晰的生命期，可以被建模为状态、迁徙和事件的演化；
- (3) 当前行为和过去行为存在着依赖关系；
- (4) 在对某事件做出反映后，它又会变回空闲状态，等待下一个事件。

虽然状态图和活动图都可以对系统的动态方面建模，但它们建模的目的有本质的区别。活动图更强调对有几个对象参与的活动过程建模，而状态图更强调对单个反应型对象建模。

在 UML 建模过程中，状态图是非常必要的，它能帮助系统开发人员理解系统中对象的行为。而类图 and 对象图只能展现系统的静态层次和关联，并不能表达系统的行为。一幅结构清晰的状态图详细描述了对象行为，这大大的帮助了开发人员构造出符合用户需求的系统。

### 6.3.3 状态图的建模技术

使用状态图一般是对系统中反映型对象建模 特别是对类、用例和系统的实例的行为建模。在对这些反映型对象建模时，要描述 3 个方面内容：对象可能处于的稳定状态，触发状态转变的事件，对象状态改变时发生的动作。这也是状态图要表达的主要内容。

稳定状态代表对象能在一段事件内被识别。当事件发生时，对象从一个状态转换到另一个状态，这些事件可以是外部的也可能是内部自身的转换。在事件或对状态的变化过程中，对象是通过执行一个动作来做出响应的。

在使用状态图对系统反映型对象建模时，可以参照以下步骤进行：

- (1) 识别一个要对其生命周期进行描述的参与行为的类；
- (2) 对状态建模，即确定对象可能存在的状态；
- (3) 对事件建模，即确定对象可能存在的状态；
- (4) 对动作建模，即确定当转变被激活时，相应被执行的动作；

(5) 对建模结果进行精化和细化。

如图 6-10 所示是手机的状态图。当手机开机时,它处于空闲状态(idle),当用户开始使用电话呼叫某人(call someone)时,手机进入拨号状态(dialing)。如果呼叫成功,即电话接通(connected),手机就处于通话状态(working);如果呼叫不成功(can't connect),例如对方线路问题、关机和拒接等,这时手机停止呼叫,重新进入空闲状态(idle)。手机在空闲状态(idle)下被呼叫(be called),手机进入响铃状态(ringing)。如果用户接听电话(pick up),手机就处于通话状态(working);如果用户未做出任何反映(haven't acts),可能他没有听见铃声,手机一直处于响铃状态(ringing);如果用户拒接来电(refused),手机回到空闲状态(idle)。

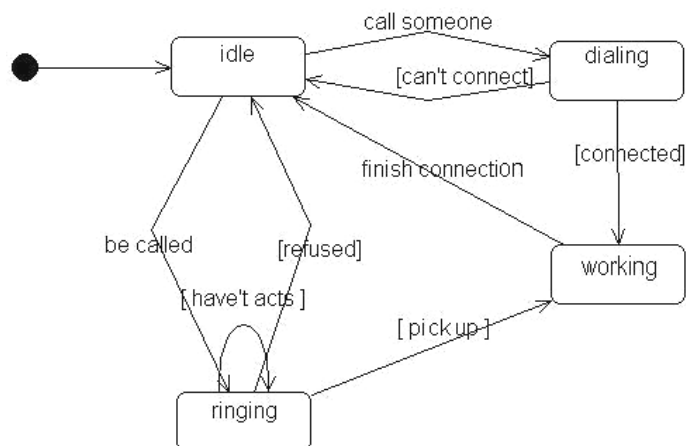


图 6-10 手机状态图

以下是一段 Java 源代码,如图 6-11 所示是其对应的状态图。

```

...
int i=0;
int sum=0;
...
public int count() {
    switch(state) {
        case working :
            if(i<10) {
                state=working;
                i=i+1;
                sum=sum+i;
            }else
                state=complete ;
            break;
        case complete :
            system.out.println ("compute complete");
    }
    return sum;
}

```

以上这段源码的功能是从 1~9 的数字累加。当 count 事件发生时,对象状态进入 working, i 和 sum 的初始值都是 0。i 不断加 1,当 i=10 时,跳出循环,对象的状态变成 complete,这

时 sum 的值应该是 1 ~ 9 的数字之和。在显示 compute complete 后，进入结束状态。

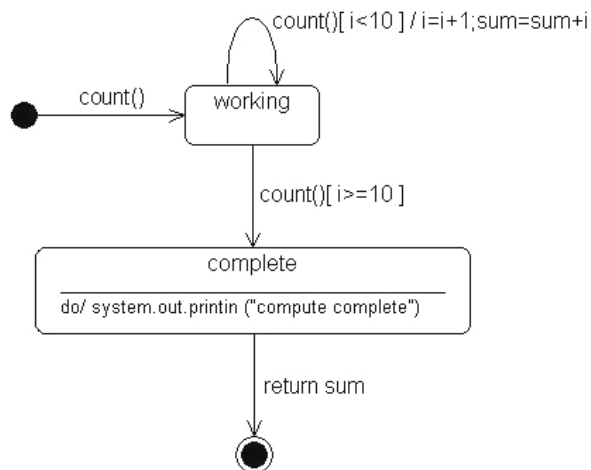


图 6-11 count()函数状态图

## 6.4 活动图 ( Activity Diagram )

一般学习过 C 语言或其他程序设计语言的读者一定接触过流程图，因为流程图清晰的表达了程序的每一个步骤序列、过程、判定点和分支。程序流程图无论对编程者自身或是阅读程序的人都是极好的文档资料。对于程序员，一般都推荐他们使用流程图做可视化描述工具来描述问解决方案。在 UML 里，活动图本质上就是流程图，它描述系统的活动、判定点和分支等，因此它对开发人员来说是一种重要的工具。

### 6.4.1 活动图的概念和内容

活动图是 UML 中描述系统动态行为的图之一，它用于展现参与行为的类的活动或动作。活动是在状态机中一个非原子的执行，它由一系列的动作组成，动作由可执行的原子计算组成，这些计算能够使系统的状态发生变化或返回一个值。

状态机是展示状态与状态转换的图。通常一个状态机依附于一个类，并且描述这个类的实例对接收到的事物的反应。状态机有两种可视化方式，分别为状态图和活动图。活动图被设计用于描述一个过程或操作的工作步骤，从这方面理解，它可以算是状态的一种扩展方式。状态图描述一个对象的状态以及状态改变，而活动图除了描述对象状态之外，更突出了它的活动。

UML 中，图形上活动图里的活动用圆角矩形表示，但这里的圆角矩形比状态图窄一些，看上去更接近椭圆。一个活动结束自动引发下一个活动，则两个活动之间用带箭头的连线相连接，连线的箭头指向下一个活动。和状态图相同，活动图的起点也是用实心圆表示，终点用半

实心圆表示。状态图中还可以包括判定、分叉和联结。活动图模型如图 6-12 所示。

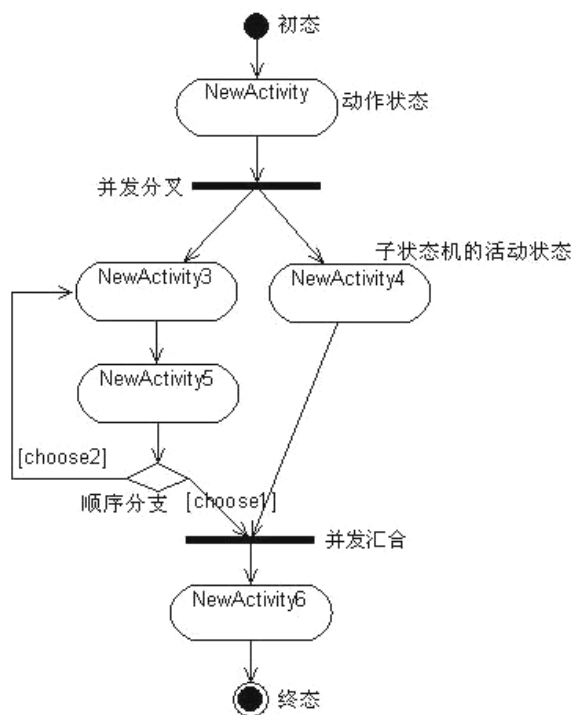


图 6-12 活动图

### 1. 动作状态

活动图包括动作状态和活动状态。对象的动作状态是活动图最小单位的构造块，表示原子动作。在 UML 里，动作状态是以执行指定动作，并在此动作完成后通过完成变迁转向另一个状态而设置的状态。这里所指的动作有 3 个特点：原子性的，即不能被分解成更小的部分；不可中断的，即一旦开始就必须运行到结束；瞬时的，即动作状态所占用的处理时间通常是极短的，甚至是可以被忽略的。

动作状态表示状态的入口动作。入口动作是在状态被激活的时候执行的动作，在活动状态机中，动作状态所对应的动作就是此状态的入口动作。

在 UML 中，动作状态使用带圆端的方框表示，如图 6-13 所示。动作状态所表达的动作就写在此圆端方框内。建模人员可以使用文本串来描述动作，它应该是动词或者是动词短语，因为动作状态表示某些行为。



图 6-13 动作状态

动作状态是一定具有入口动作和至少一条引出迁移的 UML 符号。动作状态和先前所介绍的状态图中的状态具有不同的图标。动作状态被绘制成带圆端的方框，而状态被绘制成带圆角的矩形。因此，无论从概念上还是从表达方式上，动作状态和状态都有所不同，请读者要注意

两者之间的区别。

## 2. 活动状态

对象的活动状态可以被理解成一个组合,它的控制流由其他活动状态或动作状态组成。因此活动状态的特点是:它可以被分解成其他子活动或动作状态,它能够被中断,占有有限的事件。

活动状态内部的活动可以用另一个状态机描述。从程序设计的角度来理解,活动状态是软件对象实现过程中的一个子过程。如果某活动状态是只包括一个动作的活动状态,那它就是动作状态,因此动作状态是活动状态是一个特例。

在 UML 中,动作状态和活动状态的图标没有什么区别,都是圆端的方框。只是活动状态可以有附加的部分,如可以指定入口动作、出口动作、状态动作以及内嵌状态机。

## 3. 转移

当一个动作状态或活动状态结束时,该状态就会转换到下一个状态,这就是无触发转移或称为自动转移。无触发转移实际上是没有任何特定的事件触发的转移,即当状态结束工作时就自动的发生转移。

活动图开始于初始状态,然后自动转移到第一个动作状态,一旦该状态所说明的工作结束,控制就会不加延迟的转换到下一个动作或活动状态,并以此不断重复,直到遇到一个通知状态为止。现实中,一般的控制流都有初始状态和终止状态,除非某对象开始后就不会停止。与状态图相同,活动图的初始状态也是用一个实心球表示,终止状态是用一个半实心球表示。具体的转移模式如图 6-14 所示。

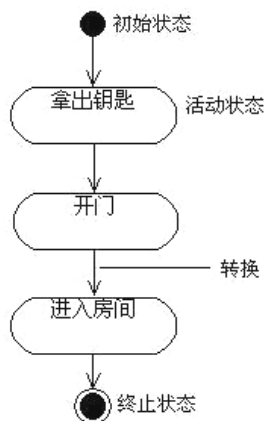


图 6-14 转移

## 4. 分支 (Branch)

在软件系统的流程图中,分支十分常见,它描述了软件对象在不同的判断结果下所执行的不同动作。在 UML 中,活动图也提供了描述这种程序结构的建模元素,这被称为分支。分支是状态机的一个建模元素,它表示一个触发事件在不同的触发条件下引起多个不同的转移。

活动图中的分支用一个菱形表示。分支可以有一个进入转换和两个或多个输出转换。在每条输出转换上都有监护条件表达式(即一个布尔表达式)保护,当且仅当监护表达式的值为真

时，该输出路径才有效。在所有的输出转换中，其监护条件不能重叠，而且它们应该覆盖所有的可能性。例如， $i>1$  和  $i>2$  这两个分支被认为是重叠； $i>1$  和  $i<1$  这两个分支没有覆盖所有的可能性，当  $i=1$  时，控制流可能被冻结，即无法选择适当的输出路径。为了方便起见，可以使用关键字 `else` 来标记一个离去转换，它表示其他监护条件都不为真时执行的路径。这样就不会出现监护条件没有覆盖所有的可能性的错误。

在活动图中引入了分支后，就可以用它来描述其他程序结构了。例如以下是一段 C 语言的循环语句，其对应的活动图如图 6-15 所示。

```
for(i:=1;i<10,i++)
{
    count(i);
}
```

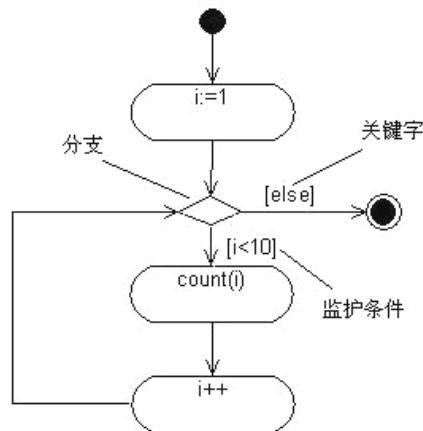


图 6-15 循环

## 5. 分叉和汇合

在建模过程中，可能会遇到对象在运行时存在两个或多个并发运行的控制流。在 UML 中，可以使用分叉把路径分成两个或多个并发流，然后使用结合，同步这些并发流。

一个分叉表示把一个控制流分解成两个或多个的并发运行控制流，也就是说分叉可以有一个输入转换和两个或多个输出转换，每个转换都是独立的控制流。从概念上说，分叉的每一个控制流都是并发的，但实际中，这些流可以是真正的并发，也可以是时序或交替的。

汇合代表两个或多个并发控制流同步发生。当所有的控制流都到达汇合点后，控制才继续向下进行。一个汇合可以有两个或多个转换和一个输入输出转换。

图形上，分叉和汇合都使用同步条表示。同步条是一条粗的水平线。如图 6-16 所示是关于学生参加考试的活动图。从初始状态开始，然后转换到活动状态“进入考场”，接下来自动迁移到分支，这产生两个并发工作流，“检查证件”和“对号入座”。在检查完证件后，进入活动状态“发考卷”，只有当“发考卷”和“对号入座”都完成时，转换汇合到“开始答题”。



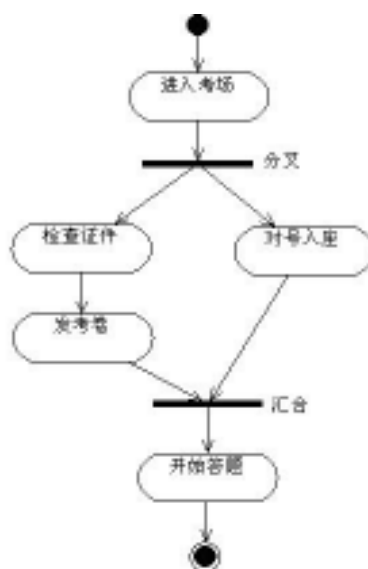


图 6-16 分叉与汇合

## 6. 泳道

泳道将活动图的活动状态分组，每一组表示负责那些活动的业务组织。在活动图里泳道区分了活动的不同职责，在泳道活动图中，每一个活动都只能明确的属于一个泳道。从语义上，泳道可以被理解为一个模型包。

泳道可以用于建模某些复杂的活动图。这时，每一个泳道可以对应于一个协同，其中活动可以由一个或多个相互连接的类的对象实现。

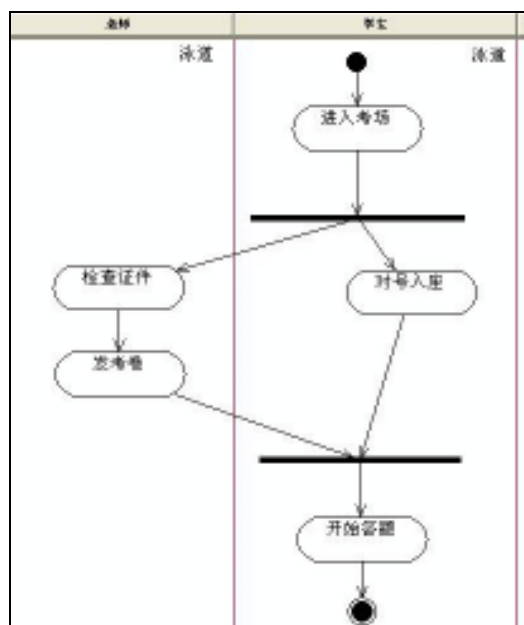


图 6-17 泳道图

在 UML 中，泳道是活动图中的一些垂直展现，把它的邻居隔开，泳道之间可以有转换。活动图中的每个泳道必须有惟一的名称以区别于其他泳道。如图 6-17 所示是学生参加考试模型的泳道活动图。

## 7. 对象流

活动图一般是对系统进行需求分析，描述系统的动态行为，这些工作处于软件开发的早期阶段。当软件开发进入建造期后，就需要考虑动态的行为实现。这时，就可以在活动图中使用对象流。

用活动图描述某个对象时，可以把所涉及的对象放置在活动图上，并用一个依赖将这些对象连接到对它们进行创建、撤销和修改的活动转换上。这种依赖关系和对象的应用被称为对象流。对象流是动作和对象间的关联。对象流可用于对下列关系建模：动作状态对对象的使用以及动作状态对对象的影响。

在 UML 中，图形上，使用矩形表示对象，矩形内是该对象的名称，名称下面的方括号中命名此对象的状态，还可以在对象名的下面加一个分隔栏表示对象的属性值。对象和动作之间使用带箭头的虚线连接带箭头的虚线表示对象流，如图 6-18 所示。

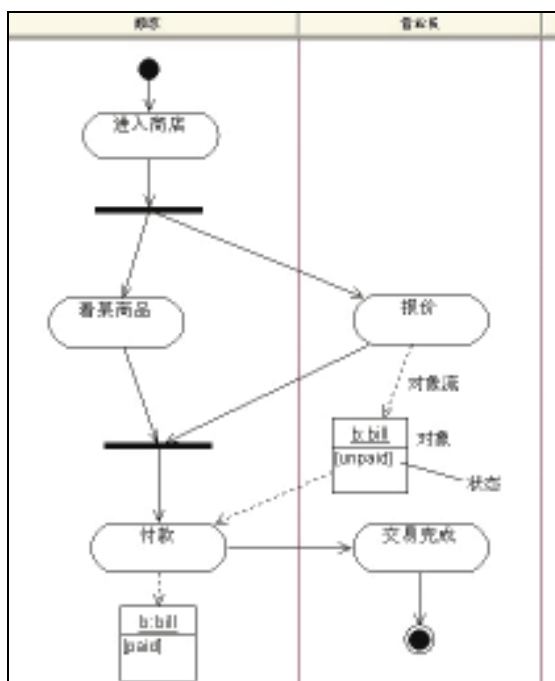


图 6-18 对象流

图 6-18 所示的活动图描述了一个顾客进入商店购买商品的工作流。对象 bill 表示所购买相应商品所对应的账单。当顾客在看商品时，bill 处于 unpaid 的状态，顾客购买后，bill 的状态变成了 paid，这是 UML 中除类的状态图外表达对象状态改变的另一种方法。

### 6.4.2 活动图的用途

活动图用于对系统的动态行为建模。它是状态机的一种可视化形式,另一种可视化形式是状态图。活动图描述了从活动到活动的流,活动是状态机中进行的非原子操作。活动图实际上是状态图的特殊形式,它的每个状态都具有入口动作,用以说明进入该状态发生的操作。

在对一个系统建模时,通常有两种使用活动图的方式:

(1) 为工作流建模

对工作流建模强调与系统进行交互的对象所观察到的活动。工作流一般处于系统的边界,用于可视化、详述、构造和文档化开发系统所涉及的业务流程。

(2) 为对象的操作建模

活动图本质上就是流程图,它描述系统的活动、判定点和分支等部分。因此,在 UML 中,可以把活动图作为流程图来使用,用于对系统的操作建模。

### 6.4.3 活动图的建模技术

在系统建模过程中,活动图能够被附加到任何建模元素,以描述其行为,这些元素包括用例、类、接口、组件、节点、协作、操作和方法。现实中的软件系统一般都包含了许多类,以及复杂的业务过程,这里所指的业务过程就是所谓的工作流。可以用活动图来对这些工作流建模,以便重点描述这些工作流。系统分析师还可以用活动图对操作建模,用以重点描述系统的流程。

无论在建模过程中活动图的重点是什么,它都是描述系统的动态行为。在建模过程中,读者可以参照以下步骤进行:

- (1) 识别要对其工作流进行描述的类;
- (2) 对动态状态建模;
- (3) 对动作流建模;
- (4) 对对象流建模;
- (5) 对建模结果进行精化和细化。

如图 6-18 和图 6-19 所示,这些活动图是对工作流建模,工作流通常是一个商业过程。图 6-19 描述的是用户使用手机接听和拨打电话的过程。具体的工作流细节请读者参照 6.3.3 小节中的说明。

虽然可以使用活动图对每一个操作建立流程图(即为对象的操作建模),但实际应用中却很少这么做。因为使用编程语言来表达更为便捷和直接。只有当操作行为非常复杂时才用活动图来描述操作的内容,因为这时通过阅读代码可能很难理解相应的操作过程。

如图 6-20 所示的活动图为对象的操作建模,其重点描述了求 Fibonacci 数列第  $n$  个数的 fib 函数的流程。Fibonacci 数列以 0 和 1 开头,以后的每一个数都是前两个数之和。

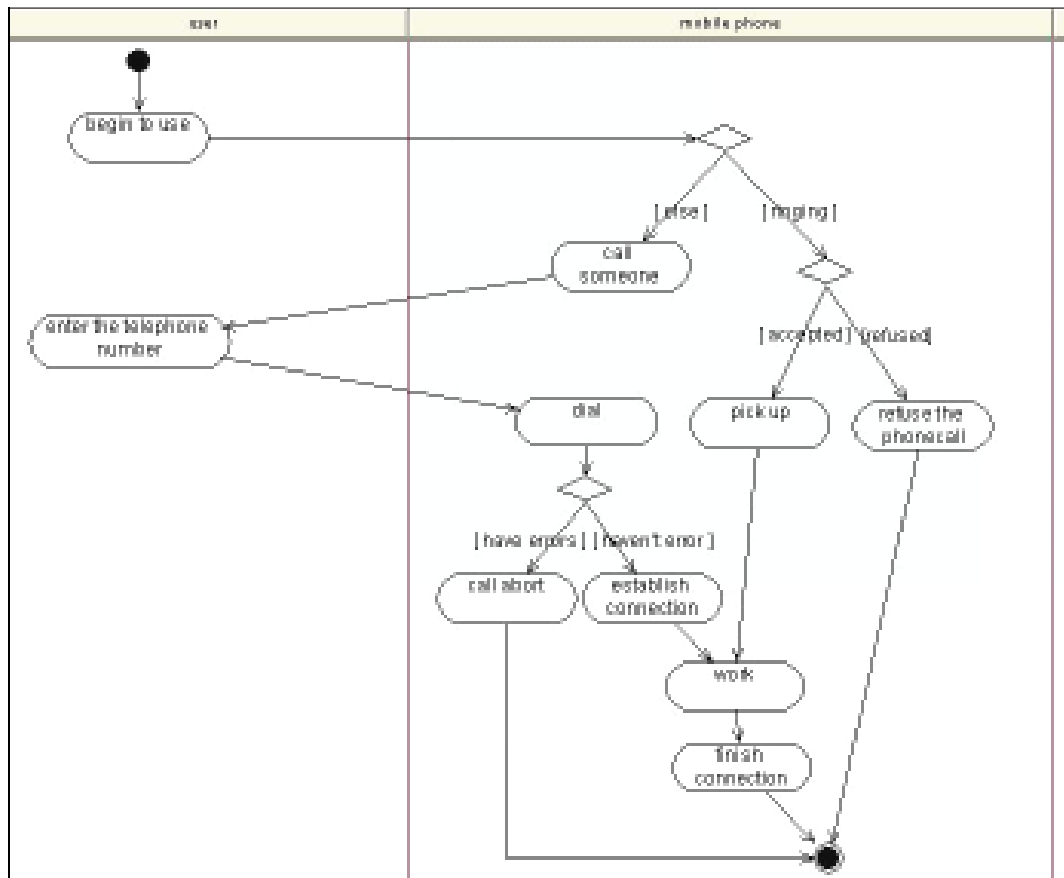


图 6-19 活动图

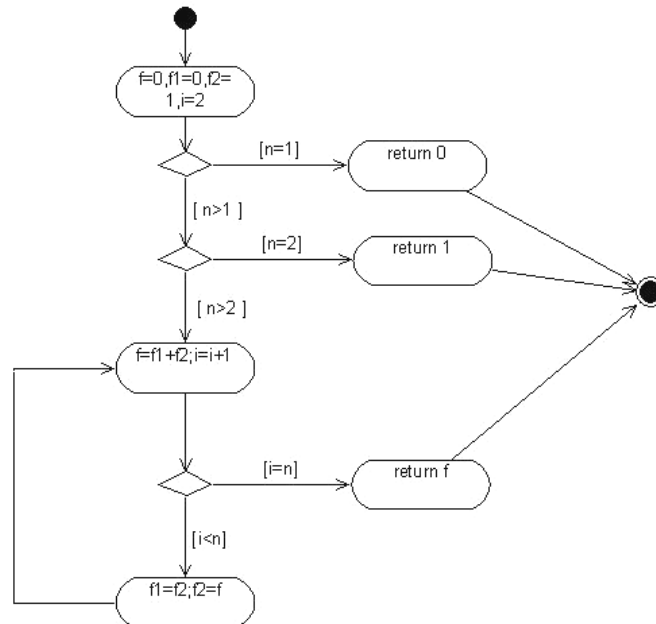


图 6-20 Fib 函数活动图

Fibonacci 数列 0、1、1、2、3、5、8、13、21.....的 C 语言实现源码。

```
#include<stdio.h>
long fibonacci (long);
main()
{
    long result,number;
    printf("enter an integer:");
    scanf ("%ld",&number);
    result=fib (number);
    printf("fib (%ld)=%ld\n",number,result);
    return 0;
}
long fib (long n)
{
    long f=0,f1=0,f2=1;
    if(n==0||n==1)
        return n;
    else
    {
        for( i=2; i<=n; i++ )
        {
            f=f1+f2;
            f1=f2;
            f2=f;
        }
        return f;
    }
}
```

## 6.5 实例——图书馆管理系统的动态视图

下面以图书馆管理系统为实例，说明如何绘制实际应用系统的动态视图，图书馆管理系统的建模背景在第9章中有详细的说明。

### 6.5.1 各种动态视图的区别

UML 的动态视图包括交互图，状态图和活动图等。

交互图、状态图和活动图都是为了说明系统行为模型而建立的，各自侧重点不同。区别在于：

- (1) 状态图是为一个对象的生命期间的情况建立模型；
- (2) 交互图（时序图与协作图）表示若干对象在一起工作完成某项服务；
- (3) 活动图描述活动的序列，建立活动间控制流的模型。

### 6.5.2 用 Rose 绘制状态图

状态图的创建过程如下：在树形列表中的 Logical View 包的图标上单击鼠标右键，在弹出的快捷菜单中，选择“New（新建）”菜单项，在弹出的子菜单中，选择“Statechart Diagram（状态图）”，如图 6-21 所示。

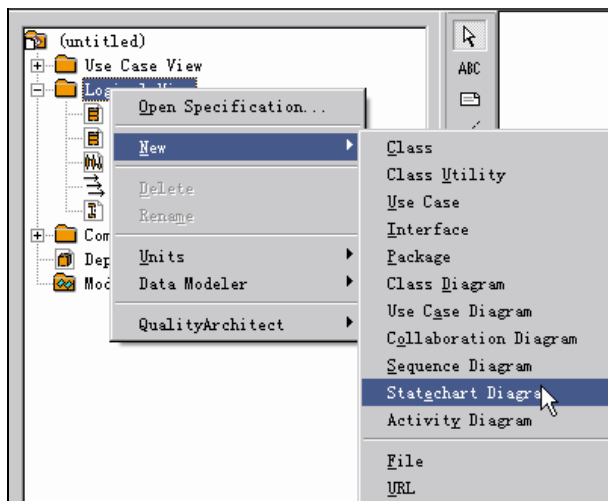


图 6-21 新建状态图

一个新的状态图新建完成，如图 6-22 所示，修改状态图名为 StatechartDiagram。

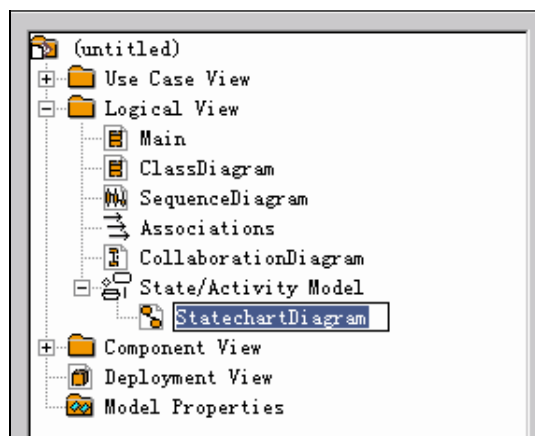


图 6-22 修改状态图名称

创建一个状态图后，Rational Rose 建立一个名为“State/Activity Model”的包，然后在其中创建状态图。双击打开状态图，编辑工具栏也有所不同，如图 6-23 所示。变化仍然从第 5 个图标开始。第 5 个是 State（状态），第 6 个是 Start State（开始状态），一张图中只能出现一个，第 7 个是 End State（结束状态），可以出现多个，第 8 个是 State Transition（从一个状态到另一状态的转换），最后一个是 Transition to Self（向本状态转换）。


每一个状态图都至少有一个开始状态和一个结束状态，单击工具栏上  标签，再单击编辑框，加入一个开始状态，如图 6-24 和图 6-25 所示。



图 6-23 编辑工具栏图



图 6-24 状态图绘制过程示意图 1

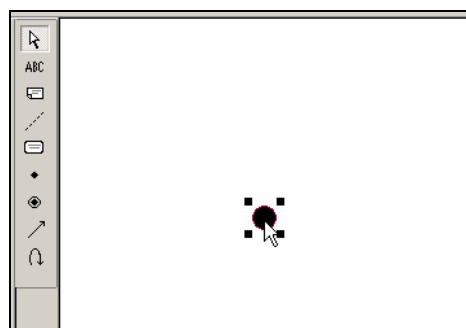



图 6-25 状态图绘制过程示意图 2

继续在工具栏上单击  标签，然后单击编辑框的空白处，自动生成状态图，输入相应的状态名字，如图 6-26 和图 6-27 所示。

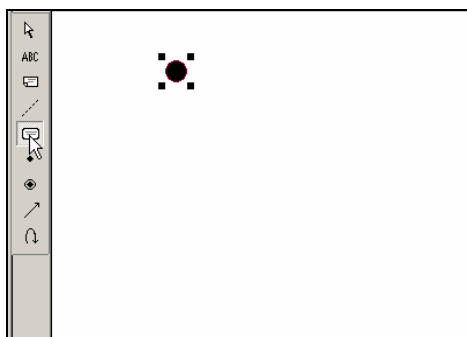


图 6-26 状态图绘制过程示意图 3

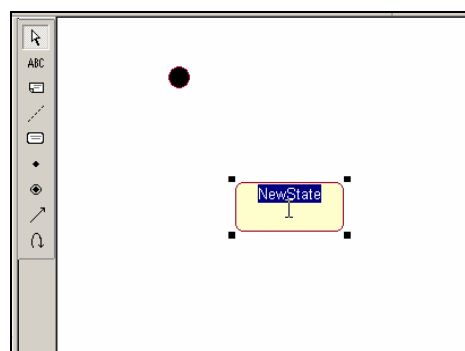



图 6-27 状态图绘制过程示意图 4

然后单击工具栏的  标签，用来进行状态连接，编辑状态的流程，先单击开始状态，然后拖动到下一个 Login（登录）状态，如图 6-28 和图 6-29 所示。

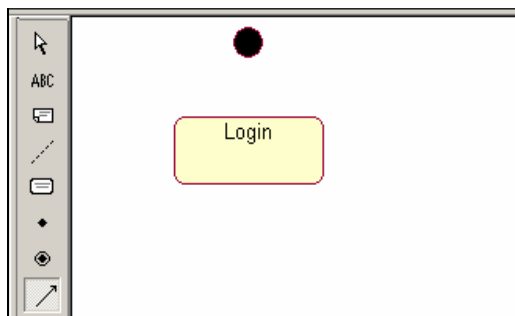


图 6-28 状态图绘制过程示意图 5

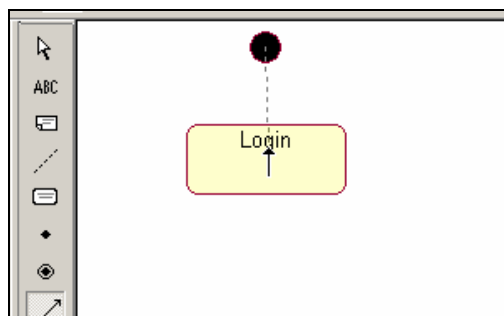



图 6-29 状态图绘制过程示意图 6

每一个状态图都有一个结束状态 ，操作类似于初始状态，单击编辑框空白处，出现结束标记图，如图 6-30 所示。

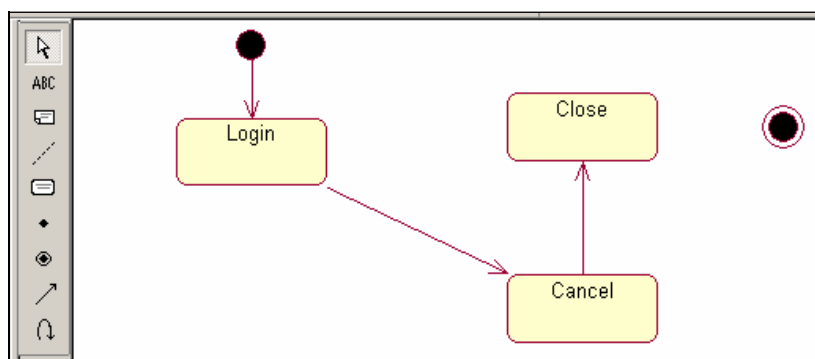


图 6-30 状态图绘制过程示意图 7

类似地，建立其他状态，然后根据状态流程进行连线，最后得到如图 6-31 所示的状态图。

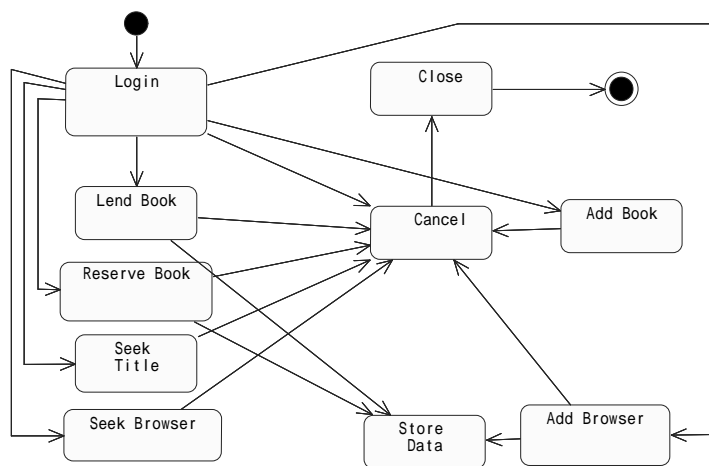


图 6-31 图书馆管理系统的状态图



注释：

Login：登录状态。

Lend Book：借阅书籍状态。

Reserve Book：预留书籍状态。

Seek Title：查询书籍信息状态。

Seek Borrower：查找读者信息状态

Store Data：存储数据状态，进行系统的数据操作，确认存储保留信息。

Add Borrower：增加读者状态。

Add Book：增加书籍状态。

### 6.5.3 用 Rose 绘制活动图

活动图的创建过程和状态图相似。要创建与当前状态图相同意义的活动图，最直接的办法就是在状态图所在的“State/Activity Model”包的图标上单击鼠标右键，在弹出的快捷菜单中选择“New（新建）”菜单项，并在弹出的子菜单中选择“Activity Diagram”即可，如图 6-32 所示。

一个与当前状态图相同意义的活动图就新建完成，如图 6-33 所示，修改活动图名称为 ActivityDiagram。

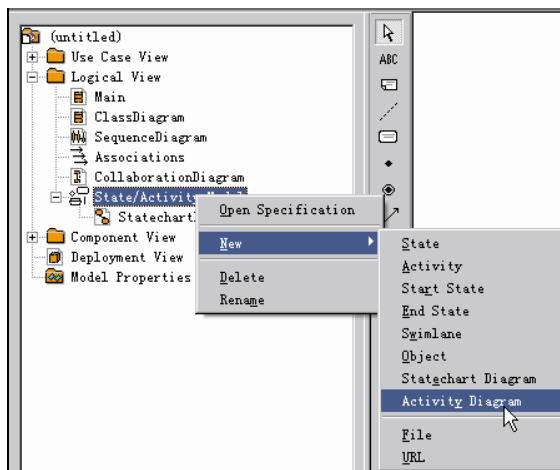


图 6-32 新建活动图

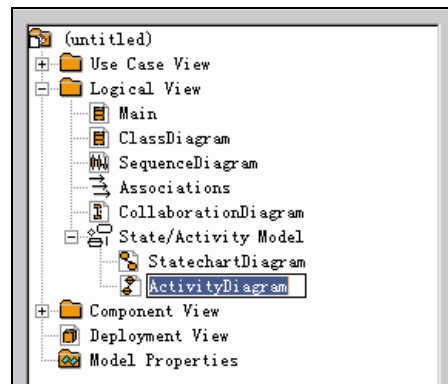


图 6-33 修改活动图名称

双击打开活动图，可以看到快捷菜单相对于状态图时多了一些图标。第 6 个图标是 Activity（活动），第 11 个是 Horizontal Synchronization（水平同步），第 12 个是 Vertical Synchronization（垂直同步），第 13 个是 Decision（分支），最后一个是 Swimlane（泳道），如图 6-34 所示。

活动图类似于状态图，需要有开始状态和结束状态，具体绘制方法与状态图类似，这里不再赘述，直接给出有起始状态和终止状态的活动图，如图 6-35 所示。



图 6-34 编辑工具栏

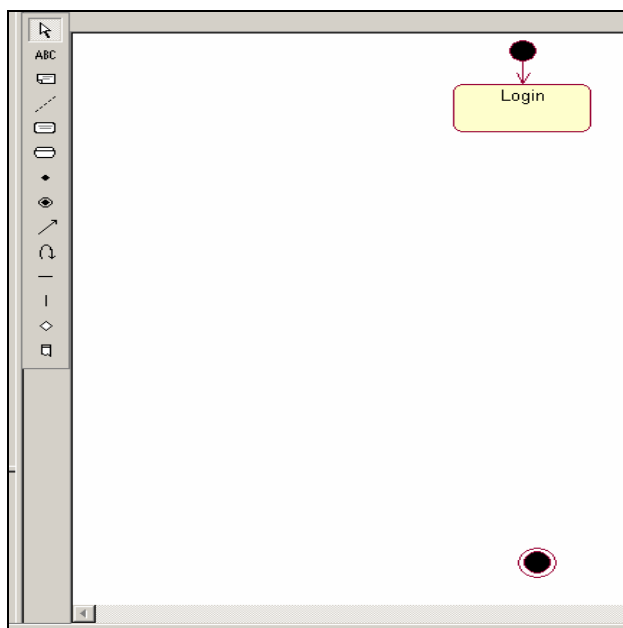



图 6-35 活动图绘制过程示意图 1

点击表示同步的  标签，然后单击编辑框的空白，出现同步标志，如图 6-36 和图 6-37 所示。

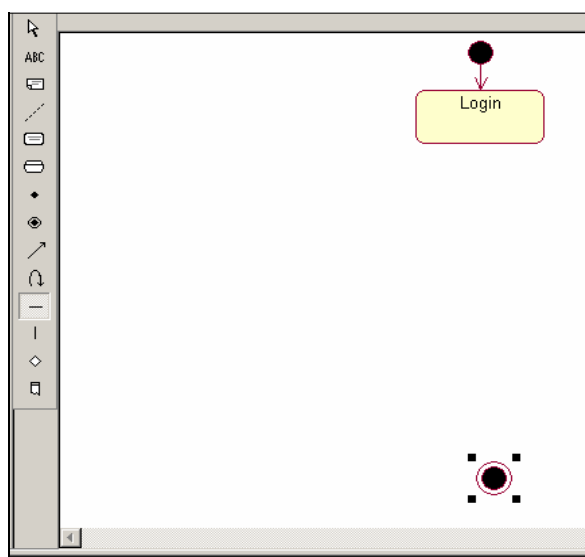


图 6-36 活动图绘制过程示意图 2

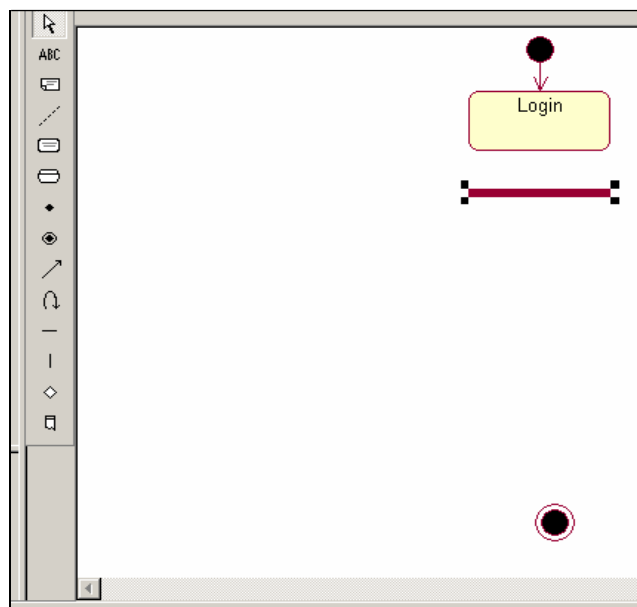



图 6-37 活动图绘制过程示意图 3

点击连接标签, 然后在编辑框中连接 Login 与同步标志图。用同样的方法连接同步标志图与借书等各状态, 如图 6-38 所示。

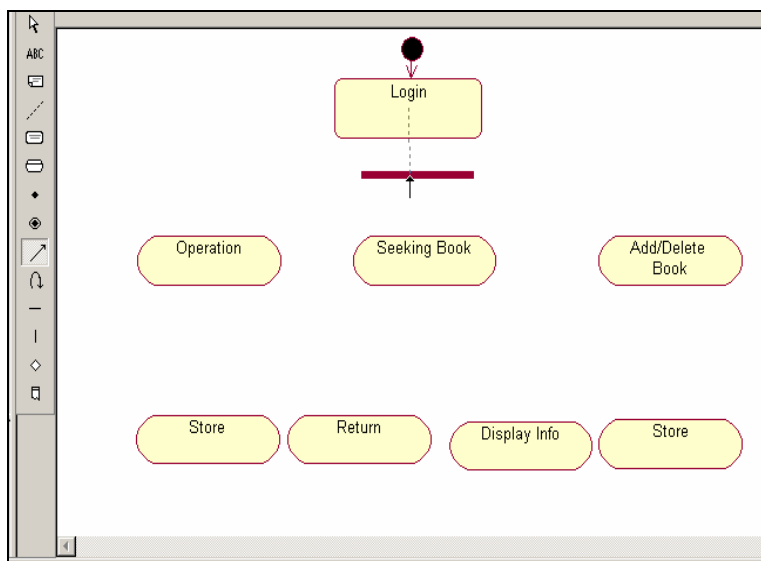


图 6-38 活动图绘制过程示意图四

为了保证连接的美观, 连线可以是直线也可以是折线, 如图 6-39 所示。

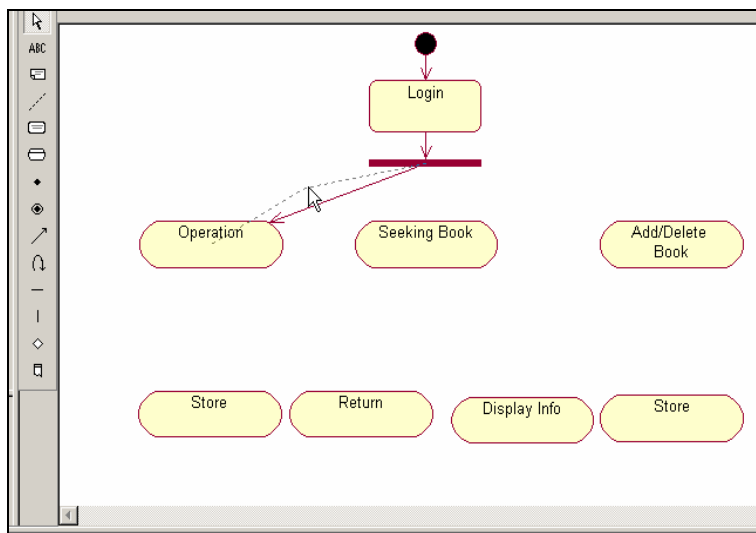


图 6-39 活动图绘制过程示意图 5

然后单击分支  标签，在编辑框的空白处加入这个分支标签，如图 6-40 和图 6-41 所示。

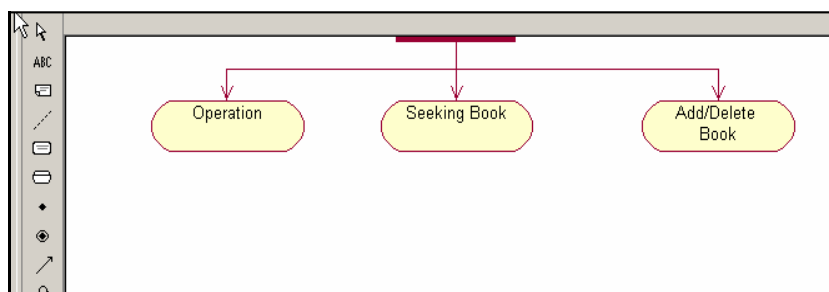


图 6-40 活动图绘制过程示意图 6

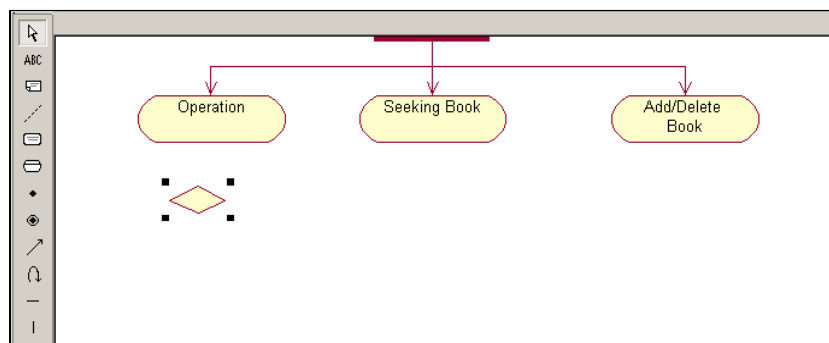



图 6-41 活动图绘制过程示意图 7

接着用  来连接，方法与上面的连接操作类似，用带箭头的连线描述各个状态之间的关联。在该例中，根据书籍的有无来区分两种状态，一种状态是借出书，保存数据信息；另一种是不借书，而返回信息说明没有书籍，如图 6-42 所示。

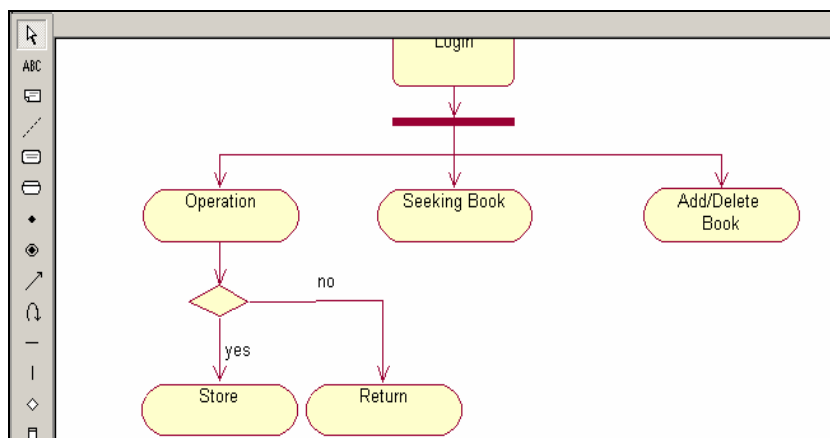


图 6-42 活动图绘制过程示意图 8

最后连接所有可能的活动图，得到如图 6-43 所示的活动图。预留书籍、查询读者和增加书籍等分别与借书、查询书籍和增加读者的活动图的绘制过程相似。

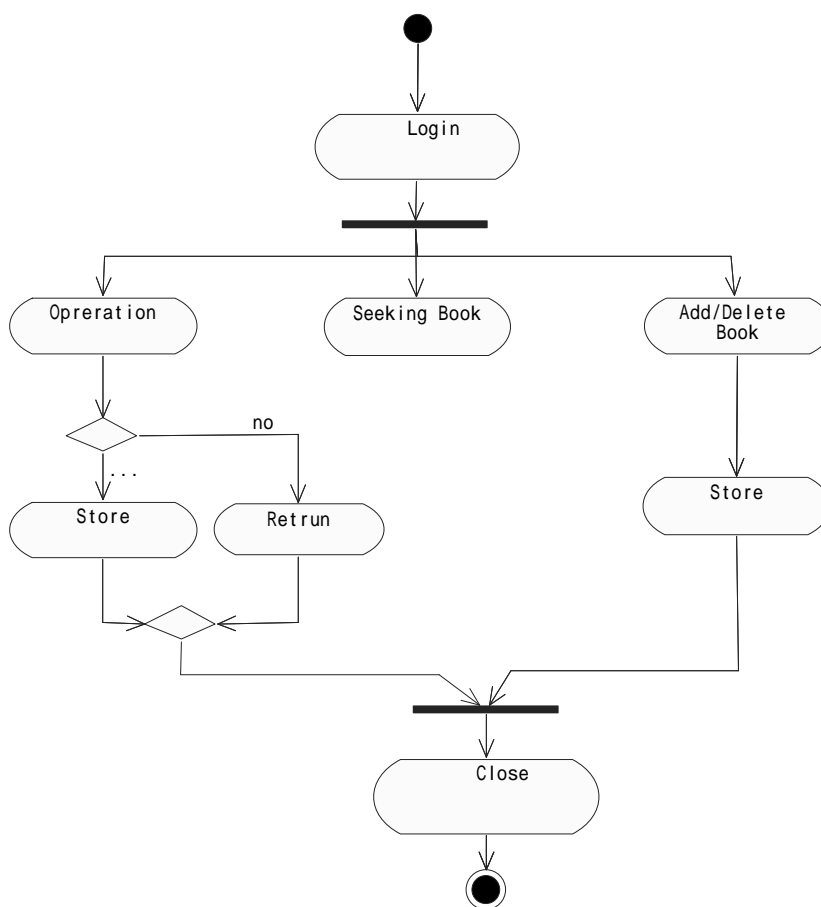


图 6-43 图书管理系统的活动图

注释：

Login：登录状态活动。

Operation：基本业务操作活动。

Seeking Book：查询书籍活动。

Add/Delete Book：增加/删除书籍活动。

Store：存储信息活动。

Return：取消操作活动，回到操作前的状态。

为了图的清楚，简略了类图的一部分内容。

#### 6.5.4 用 Rose 绘制时序图

时序图的创建过程如下：在树形列表中的 Logical View 包的图标上单击鼠标右键，在弹出的快捷菜单中，选择“New（新建）”菜单项，在弹出的子菜单中，选择“Sequence Diagram（时序图）”，如图 6-44 所示。

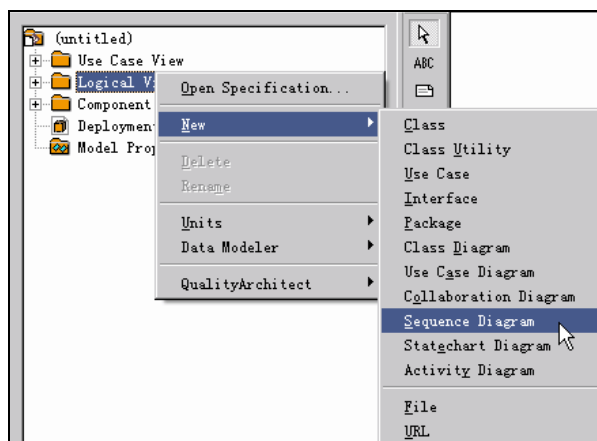


图 6-44 新建时序图

一个新的时序图就创建完成，修改时序图名称为 SequenceDiagram，如图 6-45 所示。

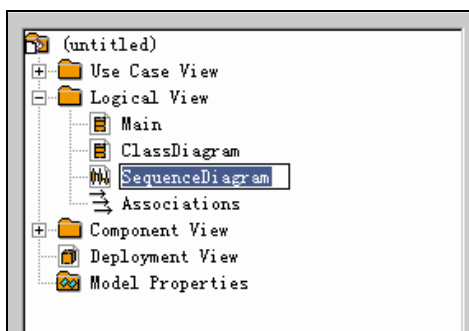


图 6-45 修改时序图名称

双击打开此图，编辑工具栏也会作相应的变化，如图 6-46 所示。编辑工具栏中第 5~9 个图标是新出现的，其中，第 5 个是 Object（对象），即时序图中消息的发送方和接收方；第六个是 Object Message（对象消息），即时序图中发送的消息；第 7 个是 Message to Self（自发自收消息），即时序图中发送给自己的消息；第 8 个是 Return Message（返回消息），即时序图中接收到消息后发回的回应消息；第 9 个是 Destruction Marker（消亡标志），即对象消亡的标志。

接下来开始绘制系统的时序图，单击工具栏  标签，然后单击编辑框，建立一个 Object，如图 6-47 所示。



图 6-46 编辑工具栏

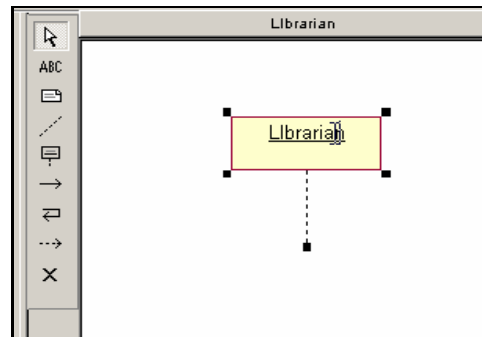


图 6-47 时序图绘制过程示意图 1

双击 Librarian 图标，弹出如图 6-48 所示的对话框，其中各项的说明如下。

Name：Object 的名称。

Class：可以关联相应的类，这里关联 Librarian，图也相应变成 Use Case 中的 .

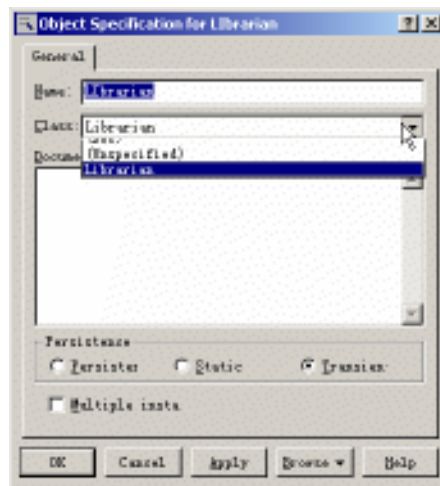


图 6-48 时序图中的“对象属性”对话框

设置好属性，单击“OK”按钮后，相应的类和名称会用冒号隔开，如图 6-49 所示。

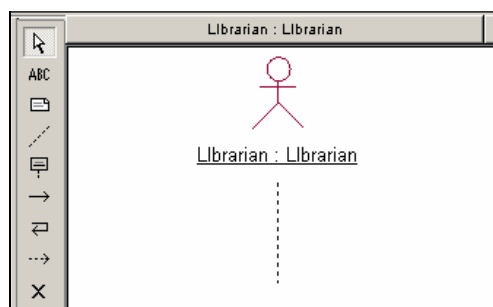



图 6-49 时序图绘制过程示意图 2

用同样的方法，再新建一个 Object ( User Login )。单击  标签，连接图中的两个 Object (对象)，注意连接的位置在垂直的虚线与虚线之间，而不是图标本身，如图 6-50 所示。

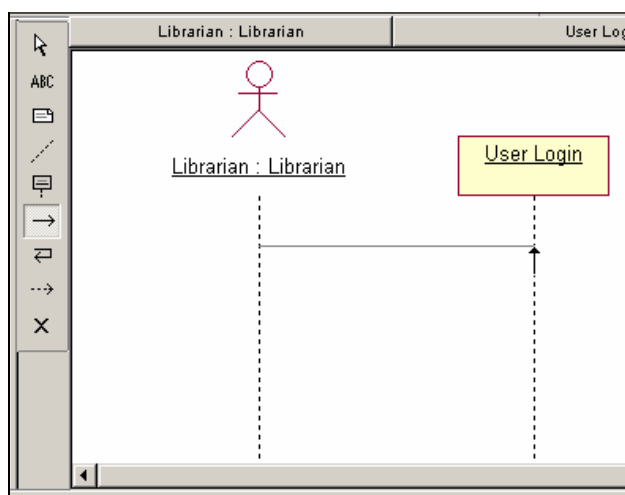


图 6-50 时序图绘制过程示意图 3

释放鼠标后得到如图 6-51 所示界面，这是时序图的最基础部分。

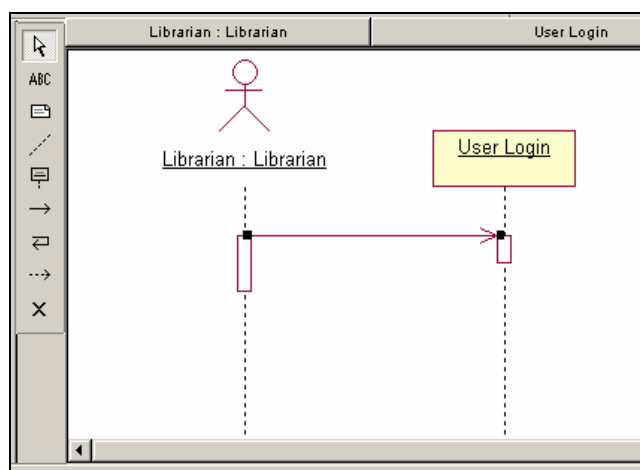


图 6-51 时序图绘制过程示意图 4



最后可以得到图书馆工作人员使用的时序图，如图 6-52 所示。

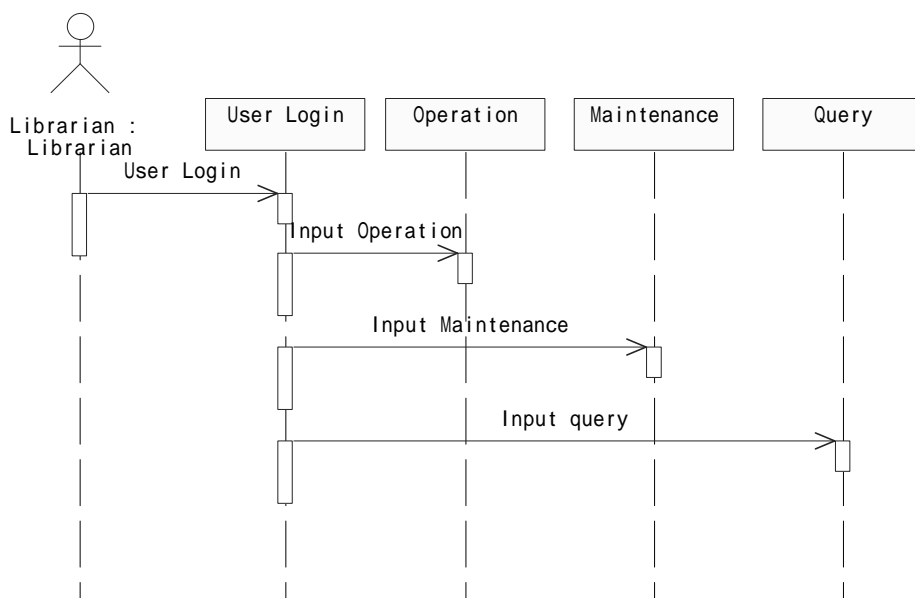


图 6-52 图书馆管理系统中的工作人员时序图

注释：

Librarian：管理人员，进行各种系统的操作。

User Login：用户登录，需要管理员输入登录验证信息。

Operation：基本业务，管理员输入必要的业务处理要求。

Maintenance：维护，管理员进行信息的维护。


Query：查询，管理员进行信息的查询。

### 6.5.5 用 Rose 绘制协作图

协作图的创建过程和时序图相似：在树形列表中的 Logical View 包的图标上单击鼠标右键，在弹出的快捷菜单中，选择“New(新建)”菜单项，在弹出的子菜单中，选择“Collaboration Diagram”(协作图)，如图 6-53 所示。

协作图建好后，修改协作图名成为 Collaboration Diagram，如图 6-54 所示。

双击打开此图，可以看到，编辑工具栏（如图 6-55 所示）的第 5 个及其之后的按钮也会有所变化。第 5 个按钮是 Object(对象)；第 6 个是 Object Instance(对象实例)；第 7 个是 Object Link(对象或对象实例之间的连接)；第 8 个是 Link to Self(与本身的联系)；第 9~12 个是消息传递的方向和类型了。

绘制图书馆工作人员使用系统的协作图的过程如下：新建一个 Object，使这个 Object 关联 Librarian 类，变成  图；接着，新建 Login 对象。在协作图里面对象之间的关系代表了协作工作的方向，连接的同时还要用相应的箭头来表明关系。

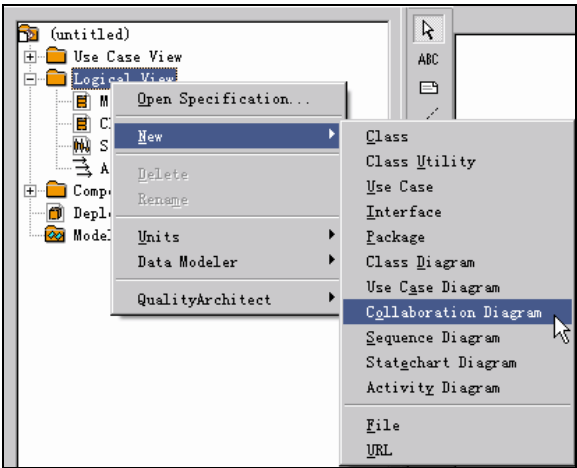


图 6-53 新建协作图

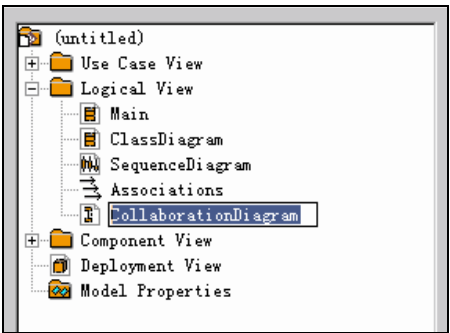


图 6-54 修改协作图名称


点击  标签，然后在编辑框中点击 Librarian 类，拖动到 Login，如图 6-56 所示。



图 6-55 编辑工具栏图

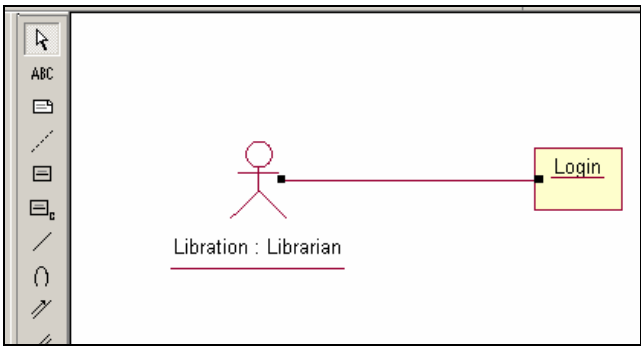



图 6-56 协作图绘制过程示意图 1

然后还需要在连线上画上相应的箭头关系，表明协作工作的方向。点击  标签，在相应的箭头上单击，出现方向标志，同时可键入方向名称，如图 6-57 和图 6-58 所示。

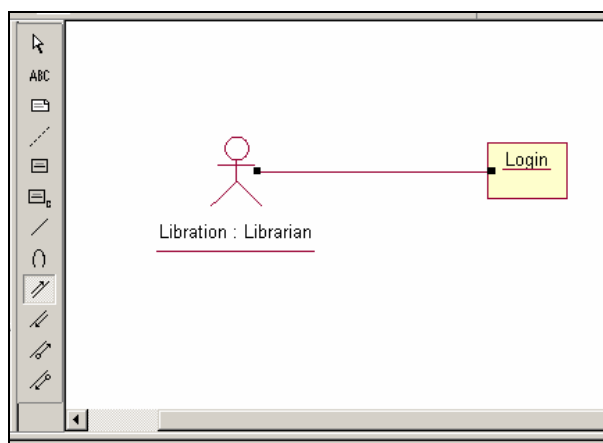


图 6-57 协作图绘制过程示意图 2

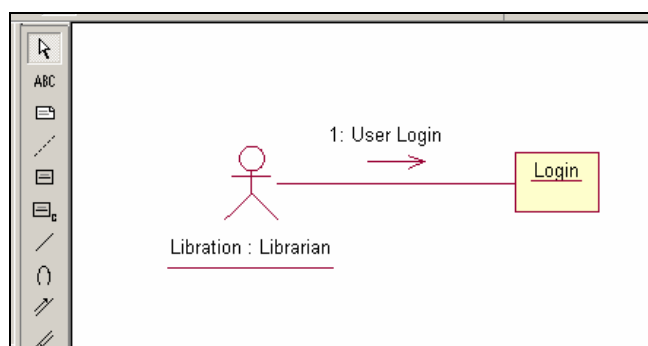


图 6-58 协作图绘制过程示意图 3

类似地，可以得到图书馆工作人员使用的协作图，如图 6-59 所示。

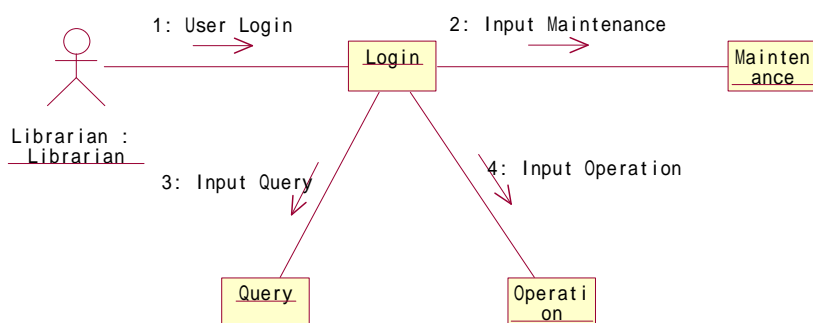


图 6-59 图书馆管理系统中的协作图

# 第 7 章 UML 实现与部署

## 7.1 组件图 (Component Diagrams)

对软件建模的过程中,可以使用用例图来推断系统希望的行为;使用类图来描述系统中的词汇;使用交互视图、状态图和活动图来说明这些词汇中的事物如何相互协作来完成某些行为。在这一切完成之后,开始人员需要把这些逻辑设计图转化成实际的事物,如可执行文件、库、表、文件和文档。在此过程中,有时必须新建某些组件,有时则可以复用已有的组件。在 UML 中,使用组件图来可视化物理组件以及它们间的关系,并描述其构造细节。

### 7.1.1 组件图的概念和内容

组件图是对面向对象系统的物理方面建模时使用的两种图之一,另一种图是配置图。组件图描述软件组件以及组件之间的关系,组件本身是代码的物理模块,组件图则显示了代码的结构。在 UML 中,每一个组件图只是系统实现视图的一个图形表示,也就是说任何一个组件图不能描述系统实现视图的所有方面,当系统中的组件组合起来,这时就能表示系统完整的实现视图,而其中的一个组件图只表示实现视图的一部分。

组件图中可以包括包和子系统,它们可以将系统中的模型元素组织成更大的组块。有时,当系统有需要可视化一个基于组件的实例时,还需要在组件图中加入实例。

以下是在系统建模过程中建立组件图的用途:

- (1) 组件图能帮助客户理解最终的系统结构;
- (2) 组件图使开发工作有一个明确的目标;
- (3) 组件图有利于帮助工作组的其他人员理解系统,例如,编写文档和帮助的人员不直接参与系统的分析和设计,然而它们对系统的理解直接影响到系统文档的质量,而组件图是帮助他们理解系统有力的工具;
- (4) 使用组件图有利于软件系统的组件重用。

组件图中通常包括:组件、接口和关系。

在接下来的几节将就这些元素做详细的介绍。

### 7.1.2 组件

组件定义开发时和运行时的物理对象的类。组件是系统中可替换的物理部件,它包装了实现而且遵从并统一提供一组接口的实现。组件常用于对可分配的物理单元建模,这些物理单元包含模型元素,并具有身份标识和明确定义的接口。

组件一般表示实际存在的、物理的物件，它具有很广泛的定义，以下的一些内容都可以被认为是组件：程序源代码、子系统、动态链接库、ActiveX 控件、JavaBean、Java Servlet、Java Server Page。这些组件一般都包含很多类并实现许多接口。

在 UML 中，图形上组件使用左侧带有两个突出小矩形的矩形表示，如图 7-1 所示。

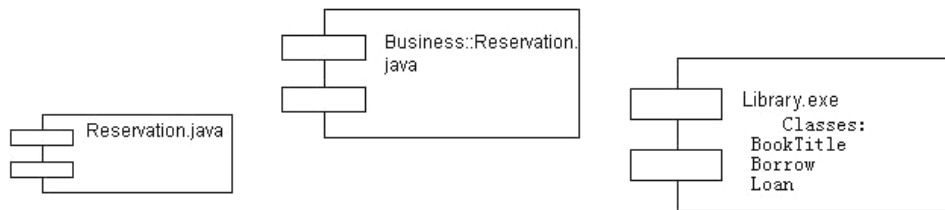


图 7-1 组件图

组件的名字位于组件图标的内部，组件名是一个文本串（如图 7-1 左侧图所示）。如果组件被某包所包含，可以在它的组件名前加上它所在包的名字（如图 7-1 中间图所示），Reservation.java 组件是属于事务包（Business）的。图 7-1 右侧的图还增加了一些表达组件的细节信息，它在图标中添加了实施该组件所需要的类。

在对软件系统建模的过程中，一般存在以下 3 种类型的组件。

（1）配置组件（Deployment Component）：配置组件是形成可执行文件的基础。例如动态链接库（DLL）、二进制可执行体（Executable）、ActiveX 控件和 JavaBeans。

（2）工作产品组件（Work Product Component）：工作产品组件是配置组件的来源，例如数据文件和程序源代码。

（3）执行组件（Execution Component）：执行组件是最终可运行系统产生的运行结果。

例如，很多读者都玩过 Windows 的扫雷游戏。当你点击扫雷游戏的图标开始游戏的时候，该图标所对应的 winmine.exe 就是配置组件。在扫雷开始后会打开存储用户信息的数据文件，用于保持以前的最好成绩，这些都是工作产品组件。游戏结束后，系统会把相应的成绩更新到用户数据文件，这时又可以算是执行组件。

### 7.1.3 接口

接口是一个类提供给另一个类的一组操作。如果一组类和一个父类之间没有继承关系，但这些类的行为可能包括同样的一些操作，这些操作具有同样的型构，不同的类之间就可以使用接口来重用这些操作。

组件可以通过其他组件的接口，使用其他组件中定义的一些操作。组件的接口又可以分为两种类型。

（1）导出接口（export interface）：导出接口由提供操作的组件提供。

（2）导入接口（import interface）：访问服务的组件使用导入接口。

前面提到，绘制组件图的用途之一就是有利于软件系统的组件重用。而使用接口则是组件重用的重要方法。系统开发人员可以在另一个系统中使用一个已有的组件，只要新系统能使用组件的接口访问新组件。他们还可以使用新的组件替换已有的组件，只要新的组件和被替换组

件接口标准一致。这在实际软件领域已经有了广泛的使用,例如许多软件的升级补丁就是使用接口一致的新组件替换旧组件。

在 UML 中,图形上接口使用一个小圆圈来表示,接口和组件之间如果使用实线连接,表示实现关系,如图 7-2 所示。

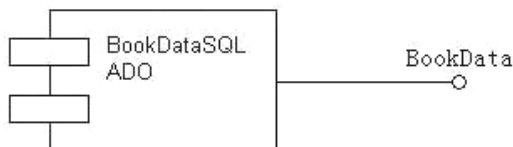


图 7-2 组件和接口 (实现关系)

如果组件和接口间使用虚线箭头连接,则表示依赖关系,也就是组件和它的导入接口间的关系,如图 7-3 所示。组件 BookDataSQL ADO 是负责连接数据库,用于读取图书信息的组件,它实现了接口 BookData,组件 BookTitleData 是处理书名信息的组件,它的信息来源于图书数据库。因此组件 BookTitleData 依赖于接口 BookData。组件 BookTitleData 通过接口享受了组件 BookDataSQLADO 所提供的服务。

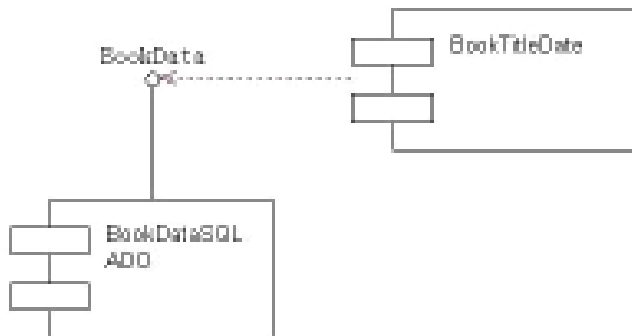


图 7-3 组件和接口 (依赖关系)

#### 7.1.4 关系

组件图中可以包括以下关系:依赖、泛化、关联和实现。从概念上理解,组件图可以算作一种特殊的类图,它重点描述系统的组件以及它们间的关系。

组件图中的依赖关系使用虚线箭头表示。具有依赖关系的组件有以下一些性质:客户端组件依赖于提供者组件;提供者组件在开发时存在,但运行时不需要存在,如图 7-4 所示。

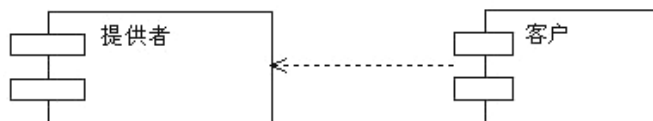


图 7-4 依赖关系

实现关系使用实线表示。实现关系多用于组件和接口之间。组件可以实现接口，这只是一种简单的说法，实际上是组件中的类实现了接口。组件和接口间实现关系的模型如图 7-5 所示，具体的实例参见图 7-3。

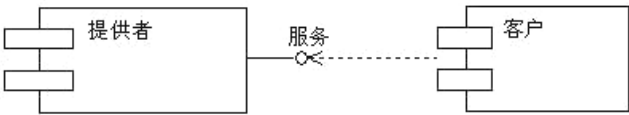


图 7-5 实现关系

7.1.5 补充图标

组件定义非常广泛，例如程序源代码、子系统、动态链接库、ActiveX 控件、JavaBean 等都可以被认为是组件。在实际建模过程中，如果仅仅使用一个图标表示组件可能会有所不便，因此在一些建模工具里都为不同类型定义了特别的图标，这便于系统设计师建模，也便于其他人员理解。

下面以 Rational Rose 为例，介绍不同类型组件的图标表示。

(1) 子程序规范和子程序体

以下两个图标分别表示子程序的显示规范和实现体。子程序一般是一组子程序集名。子程序中不包括类定义，如图 7-6 所示。

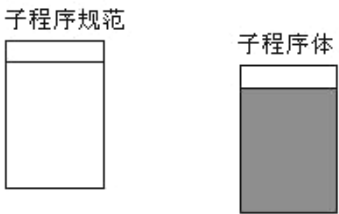


图 7-6 子程序规范和子程序体

(2) 主程序

如图 7-7 所示表示主程序。主程序是包含程序根的文件。例如，在 Delphi 中，主程序文件包含应用程序对象。



图 7-7 主程序

### (3) 包规范和包体

UML 中，包是类的实现。包规范表示程序中的头文件，包含类的函数原型信息。在 C++ 中，包规范为.h 文件。包体包含类操作代码，在 C++ 中，包体为.cpp 文件。包规范和包体的表示如图 7-8 所示。

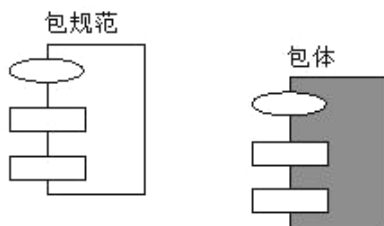


图 7-8 包规范和包体

### (4) 任务规范和任务体

如图 7-9 所示表示具有独立控制线程的包。可执行文件通常用具有.exe 扩展的任务规范来表示。

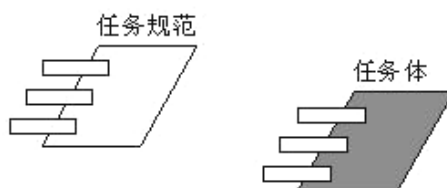


图 7-9 任务规范和任务体

### (5) 数据库

如图 7-10 所示表示数据库。数据库可能包含一个和几个结构。



图 7-10 数据库

## 7.1.6 组件图建模技术

组件图用于对系统的实现视图建模。组件图描述软件组件及组件之间的关系，组件本身是代码的物理模块，组件图则显示了代码的结构。组件是逻辑架构中定义的概念和功能（类、对象以及它们的关系和协作）在物理架构中的实现。

在实际建模过程中，读者可以参照以下步骤进行：

- (1) 对系统中的组件建模；
- (2) 定义相应组件提供的接口；



- (3) 对它们间的关系建模；
- (4) 对建模的结果进行精华和细化。

如图 7-11 所示描述了一组从图书馆信息系统中提取的数据库表。图中有一个数据库 library.db，它包括 5 个表，分别为读者信息表 (borrower information)、图书信息表 (book information)、借书信息表 (borrow)、预定信息表 (reservation) 和还书信息表 (return)。在组件图中，用原型为表的组件来表示这些表。如果有必要，还可以详述这些表的内容。

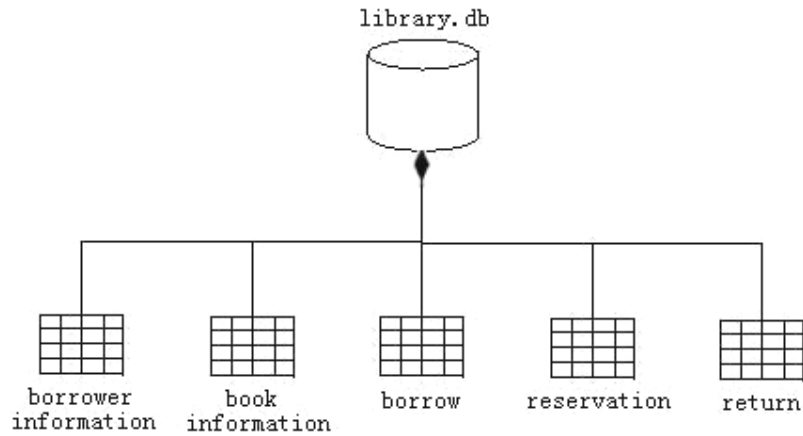


图 7-11 对物理数据库建模

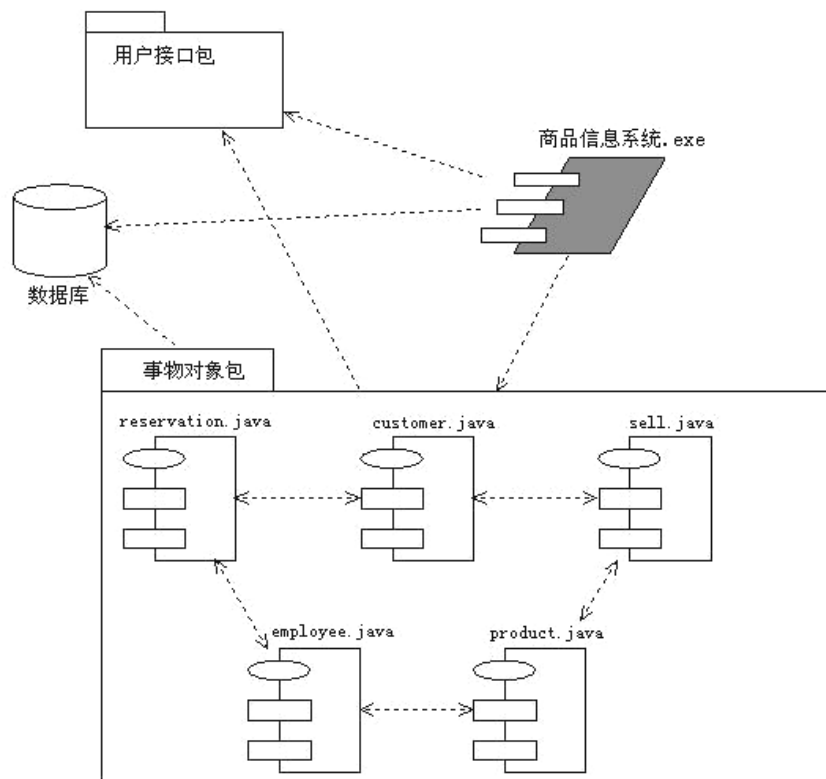


图 7-12 商品信息系统组件图

图 7-12 所示是描述商品信息系统的组件图的综合实例。用户接口包负责用户的交互和图形的显示、打印。数据库负责存储商品信息、顾客信息和交易信息等。业务对象包执行系统的业务逻辑，它是完成系统各项功能的中间环节。其中业务对象包中包括类 reservation、类 customer、类 sell、类 employee 和类 product 的头文件，双向箭头对应双向关联表示相互依赖。

## 7.2 配置图 (Deployment Diagrams)

当一个软件开发组开发并实施某软件项目时，这个开发组可能需要软件设计人员和系统开发人员。在分工方面，软件设计人员主要负责软件的构造和实施，即根据用户的需求开发出符合用于要求的软件产品。仅仅做到这一点是不够的，开发小组还需要系统开发人员，他们负责系统的硬件和软件两个方面，并保证开发出的软件产品能够在合适的硬件系统上运行。UML 主要是为构造软件提供便利，但它也可以用于设计系统硬件。本小节介绍的配置图就是用于描述软件执行所需的处理器和设备的拓扑结构。

### 7.2.1 配置图的概念和内容

配置图是对面向对象系统的物理方面建模时使用的两种图之一，另一种图是组件图。配置图显示了运行软件系统的物理硬件，以及如何将软件部署到硬件上。也就是说，这些图描述了执行处理过程的系统资源元素的配置情况以及软件到这些资源元素的映射。

配置图中可以包括包和子系统，它们可以将系统中的模型元素组织成更大的组块。有时，当系统需要可视化硬件拓扑结构的一个实例时，还需要在配置图中加入实例。配置图中还可以包含组件，这些组件都必须存在于配置图中的节点上。

配置图描述了运行系统的硬件拓扑。在实际使用中，配置图常被用于模拟系统的静态配置视图。系统的静态配置视图主要包括构成物理系统的组成部分的分布和安装。

配置图中通常包括：节点、组件和关系。

### 7.2.2 节点

节点是定义运行时的物理对象的类，它一般用于对执行处理或计算的资源建模。节点通常具有如下两方面内容：能力（如基本内存、计算能力和二级存储器）和位置（在所有必需的地理位置上均可得到）。在建模过程中，可以把节点分成两种类型。

（1）处理器（Processor）：能够执行软件构件、具有计算能力的节点。

（2）设备（Device）：没有计算能力的节点，通常是通过其接口为外界提供某种服务，例如打印机、扫描仪等都是设备。

在 UML 中，图形上节点使用一个三维立方体来表示，如图 7-13 所示。

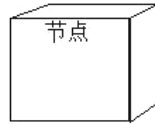


图 7-13 节点

节点的名字位于节点图标的内部，节点名是一个文本串。如果节点被某包所包含，可以在它的组件名前加上所在包的名字（如图 7-14 左侧图标所示），节点 Printer（打印机）是属于 OutPutDevice（输出设备包）的。节点的立方体还可以划分出多个区域，每个区域中可以添加一些细节的信息（如图 7-14 右侧图标所示），例如在该节点上运行的软件或者该节点的功能等。根据附加信息可知，节点 Server 的功能是打印服务器。

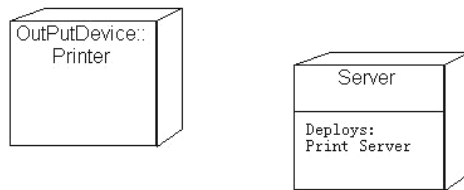


图 7-14 节点的名字

### 7.2.3 组件

配置图中还可以包含组件。这里所指的组件就是 7.1.2 小节中介绍的组件图中的基本元素，它是系统中可替换的物理部件，并包装提供某些服务的接口。

可将组件包含在节点符号中，表示它们处在同一个节点上，并且在同一个节点上执行。从节点类型可以画一条带有 `<<support>>` 的相关性的虚线箭头指向运行时的组件类型，说明该节点支持指定组件（如图 7-15 左侧图形所示）。当一个节点类型支持一个组件类型时，在该节点类型实例上执行它所支持的组件的实例是允许的。

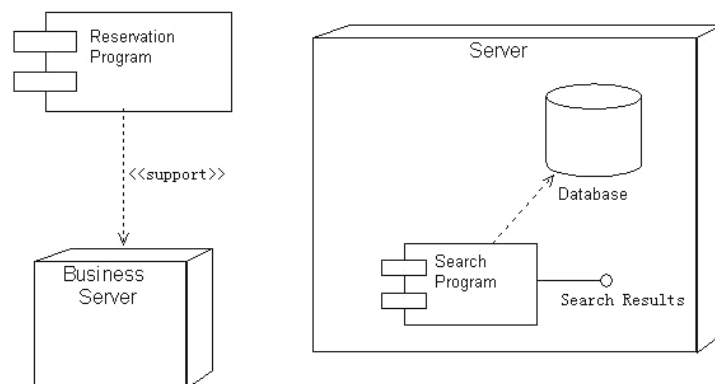


图 7-15 节点与组件

可以通过虚线箭头将不同组件连接在一起，表示它们之间的依赖关系（配置图中，只显示运行时的组件）。如图 7-15 右侧图形所示，这意味着，一个组件使用另一个组件中的服务。

#### 7.2.4 关系

组件图中通常包括依赖关系和关联关系。从概念上理解，配置图也是一种类图，其描述了系统中的节点以及节点间的关系。

配置图中的依赖关系使用虚线箭头表示，它通常用在配置图的组件和组件之间。读者可以参考图 7-15 所示的实例。

关联关系常用于对节点间的通信路径或连接进行建模。关联用一条直线表示，说明在节点间存在某类通信路径，节点通过这条通信路径交换对象或发送信息，模型如图 7-16 所示。

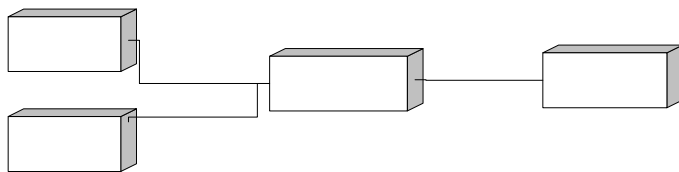


图 7-16 关联关系

#### 7.2.5 配置图建模技术

配置图用于对系统的实现视图建模。绘制这些视图主要是为了描述系统中各个物理组成部分的分布、提交和安装过程。

在实际应用中，并不是每一个软件开发项目都必须绘制配置图。如果项目开发组所开发的软件系统只需要运行于一台计算机并且只需使用此计算机上已经由操作系统管理的标准设备（比如键盘、鼠标和显示器等），这种情况下就没有必要绘制配置图了。另一方面，如果项目开发组所开发的软件系统需要使用操作系统管理以外的设备（例如数码相机和路由器等）或者系统中的设备分布在多个处理器上，这时就有必要绘制配置图，以帮助开发人员理解系统中软件和硬件的映射关系。

绘制系统配置图，可以参照以下步骤进行：

- （1）对系统中的节点建模；
- （2）对节点间的关系建模；
- （3）对系统中的节点建模，这些组件来自组件图；

- (4) 对组件间的关系建模；
- (5) 对建模的结果进行精华和细化。

如图 7-17 所示是家用计算机系统的配置图。配置图中包括电脑主机和一些外围设备，这些设备包括 Monitor（显示器）、KeyBoard（键盘）、Mouse（鼠标）、Modem（调制解调器）。除了这些计算机的外围设备，配置图中还包括了拨号上网的 ISP（网络服务提供商）。

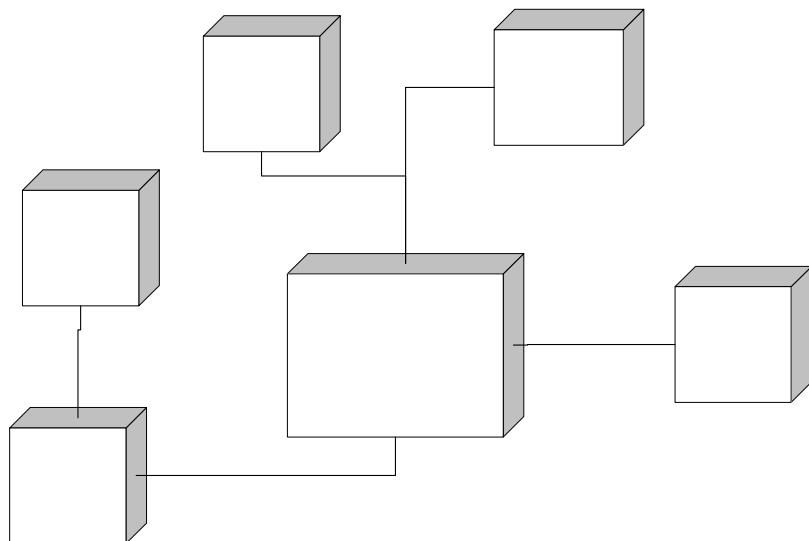


图 7-17 家用计算机系统配置图

如图 7-18 所示是一个完全分布式系统的配置图。图中包括了两个客户机，分别用 client1 和 client2 表示，它们与 regional server（地区服务器）相连，地区服务器作为 country server（国家服务器）的前端，它们也相连。在图中，Internet 被表示为原型为<<network>>的节点。在这个分布式系统中，存在多个地区服务器和国家服务器，这些服务器间也是彼此相连的，但是配置图中没有绘制出来。

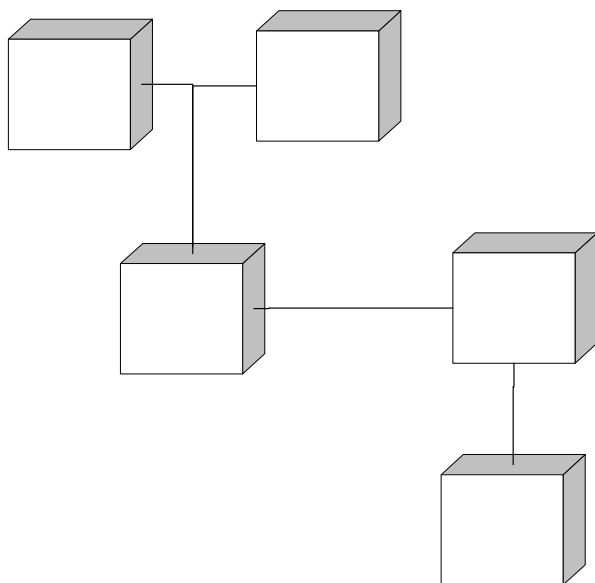


图 7-18 分布式系统配置图

如图 7-19 所示是一个在线交易系统的配置图。图中包括了两个客户机，是访问该在线交易系统的客户。客户机与 Web 服务器相连，客户通过访问 Web 服务器获取商品信息。Web 服务器在获得订单后把消息提交给应用程序服务器，应用程序服务器处理消息并把结果储存在数据库服务器中。

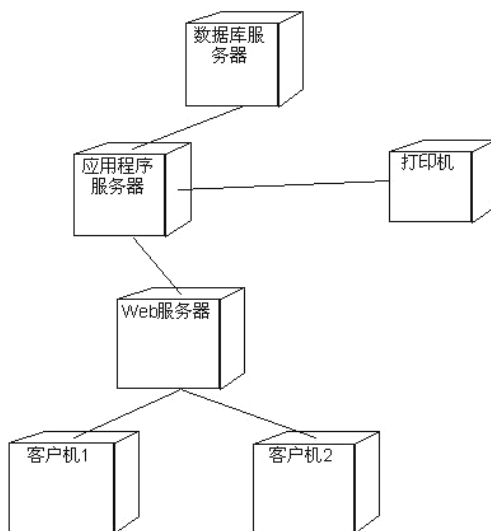


图 7-19 在线交易系统配置图

## 7.3 实例——图书馆管理系统的组件图与配置图

本节以图书馆管理系统为实例，说明如何绘制实际应用系统的组件图与配置图。图书管理系统的建模背景在第 9 章中有详细的说明。

### 7.3.1 绘制组件图与配置图的步骤

绘制组件图步骤如下：

- (1) 确定组件；
- (2) 给组件加上必要的构造型；
- (3) 确定组件的联系；
- (4) 绘制组件图。

绘制配置图步骤如下：

- (1) 确定节点；
- (2) 加上构造型；
- (3) 确定节点的联系；
- (4) 绘制配置图。

7.3.2 用 Rose 绘制组件图

组件图的创建过程和前面几章中提到的图的创建过程相似，只是创建的包要选择 Component View，在树形列表中的 Component View 包的图标上单击鼠标右键，在弹出的快捷菜单中，选择“New（新建）”菜单项，在弹出的子菜单中，选择“Component Diagram（组件图）”，如图 7-20 所示。

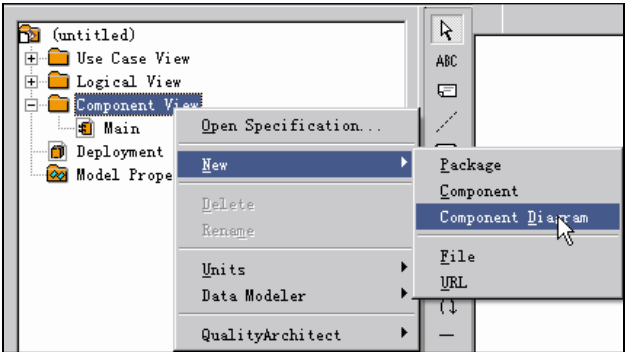


图 7-20 新建组件图

一个新的组件图就创建好了，如图 7-21 所示，将组件图的名称改为 ComponentDiagram。

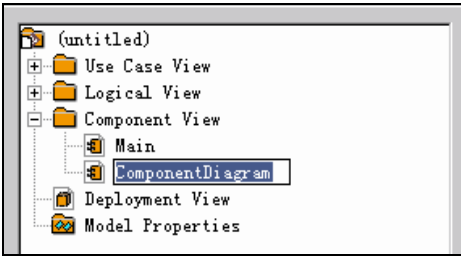


图 7-21 修改组件图名称

双击打开组件图，可以看到从第 5 个图标开始，编辑工具栏增添了新的内容，如图 7-22 所示。



图 7-22 编辑工具栏

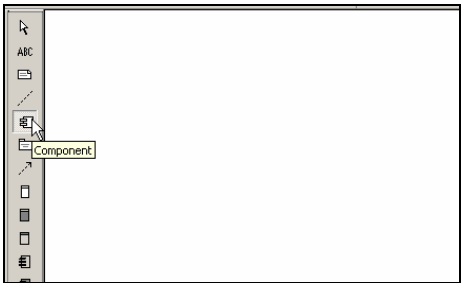



图 7-23 组件图绘制过程示意图 1

点击工具栏 标签，然后在编辑框的空白处单击加入一个组件，可以自行输入相应的组件名称，如图 7-23 和图 7-24 所示。

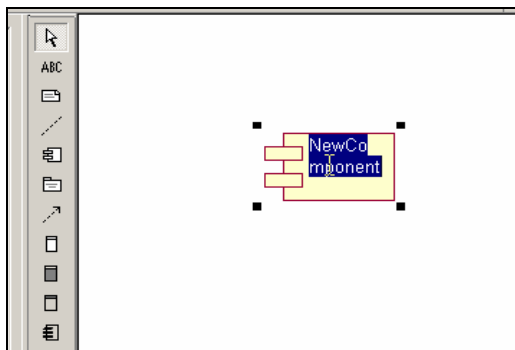


图 7-24 组件图绘制过程示意图 2

用同样的方法，加入其他的组件，如图 7-25 所示。

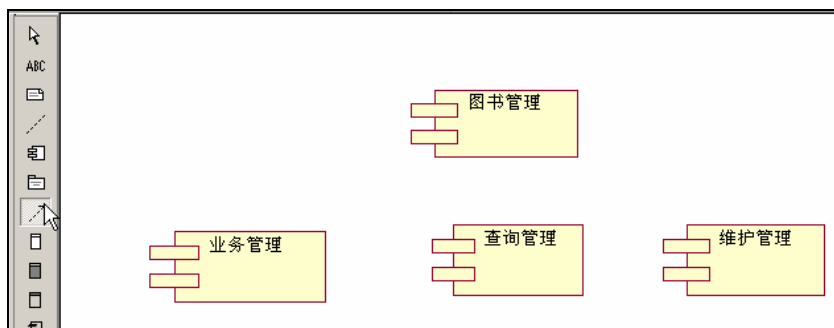



图 7-25 组件图绘制过程示意图 3

点击工具栏的连接线 标签，然后连接“图书管理”与“业务管理”，如图 7-26 所示。

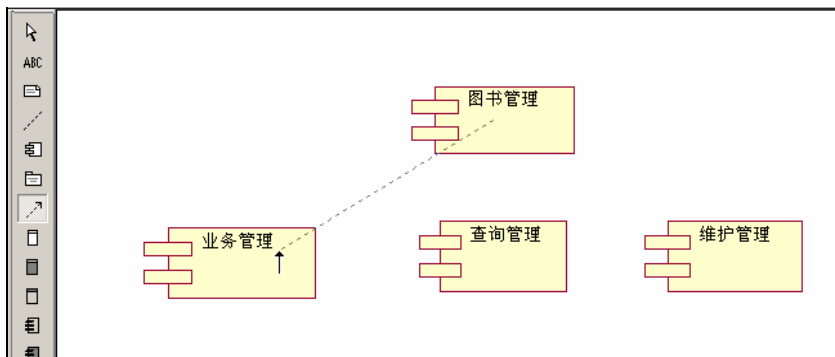


图 7-26 组件图绘制过程示意图 4



最后得到如图 7-27 所示的组件图。

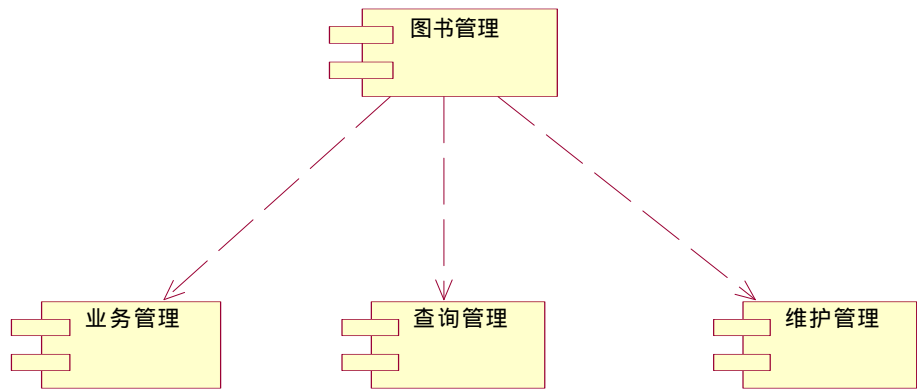


图 7-27 图书馆管理系统组件图

7.3.3 用 Rose 绘制配置图

配置图其实并不需要创建，因为模型里面已经建好了配置图，如图 7-28 所示。

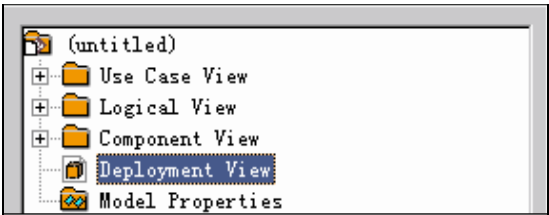


图 7-28 配置图

双击打开，可以看到编辑工具栏，如图 7-29 所示。其中的第 5 个图标是 Processor（处理器，表示处理单元），第 6 个是 Connection（连接），第 7 个是 Device（设备）。

现在对 9 种图的创建都做了一个简单的说明，将模型保存，只需要选择“File”（文件）菜单中的“Save”（保存）菜单项即可，如图 7-30 所示。



图 7-29 编辑工具栏

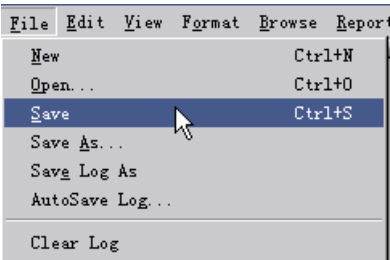


图 7-30 保存模型

在弹出的窗口中选择合适的路径，如图 7-31 所示。

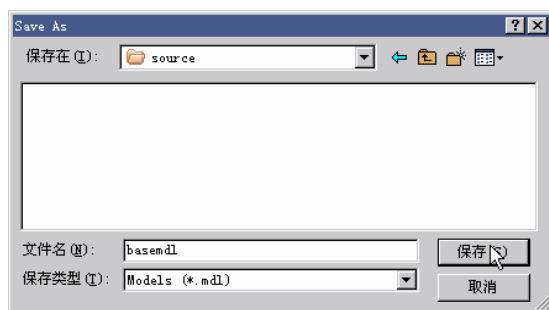


图 7-31 选择合适的路径


接下来，要向这些图中添加元素。单击工具栏的设备标签, 然后在编辑框的空白处单击，出现设备图，接着输入相应的名称，如图 7-32 和图 7-33 所示。



图 7-32 配置图绘制过程示意图 1

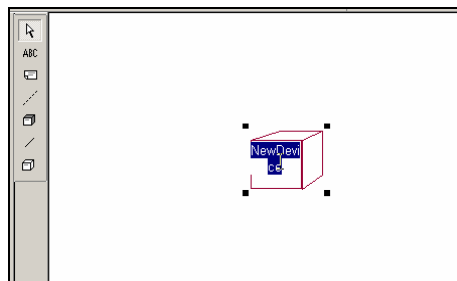


图 7-33 配置图绘制过程示意图 2

同样的，加入多个设备图，如图 7-34 所示。

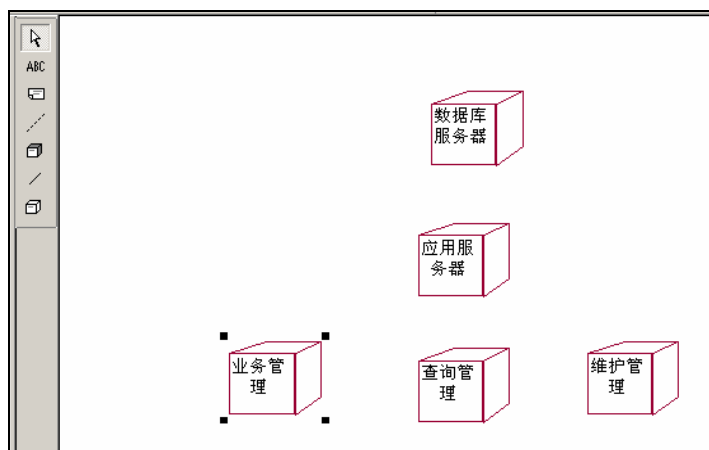



图 7-34 配置图绘制过程示意图 3

点击工具栏的连接标签, 然后连接数据库与应用服务器, 如图 7-35 和图 7-36 所示。

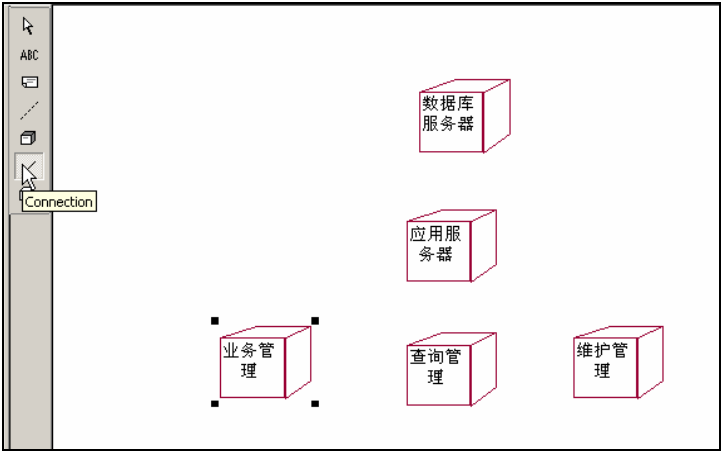


图 7-35 配置图绘制过程示意图 4

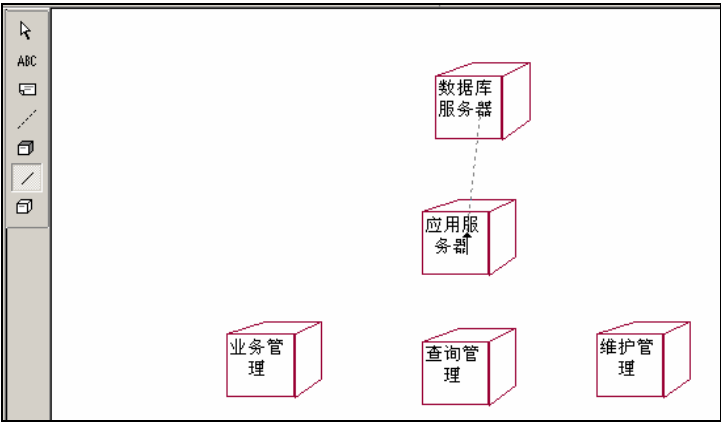


图 7-36 配置图绘制过程示意图 5

最后，可以得到如图 7-37 所示的配置图。

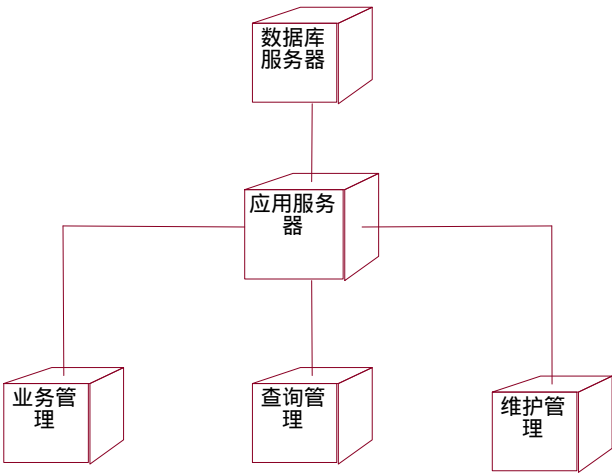


图 7-37 图书管理系统的配置图

# 第 8 章 UML 与统一开发过程

## 8.1 软件过程历史概述

### 8.1.1 软件开发过程简介

什么是软件过程？概括地讲，软件过程是指实施于软件开发和维护中的阶段、方法、技术、实践及相关产物（计划、文档、模型、代码、测试用例和手册等）的集合。行之有效的软件过程可以提高开发软件组织的生产效率、提高软件质量、降低成本并减少风险。

那么，软件过程对于软件企业来说有什么重要意义呢？行之有效的软件过程可以提高软件企业的开发效率。首先，通过理解软件开发的基本原则有助于对软件开发过程中一些重要的问题做出明智的决定；其次，可以促进开发工作的标准化、促进项目小组之间的可重用性和一致性；第三，它提供了一个可以使软件企业引进行业内先进开发技术的机会，这些技术包括代码检测、配置管理、变更控制以及体系结构建模等等。有效地软件过程还有助于改进软件企业的软件维护和技术支持等工作。首先，它定义了如何管理软件的变更并将这些变更的维护工作适当地分配到软件的未来版本中，这样使整个变更的过程无缝地进行；其次，它定义了如何将软件产品平稳地过渡到运行实施和技术支持阶段，以及如何有效地开展这些工作。

### 8.1.2 当前流行的软件过程

对于软件企业而言，有必要采用某种业界认可的软件过程，或是利用新的技术改进自身已经存在软件过程。因为现在的软件规模越来越大，复杂程度越来越高，在软件的开发和维护过程中缺乏有效管理和控制，这对于一个软件企业的成功是非常不利的。而且不仅仅是软件变得更复杂，现在的软件企业也通常需要同时进行多个软件的开发，需要对项目进行有效地管理。当今需要的软件应该是交互性的、国际化的、用户友好的、高处理效率的和高可靠性的系统，这就要求软件企业提高产品的质量并且最大可能地实现软件复用，以较低的成本和较高的效率完成工作。行之有效的软件过程为实现这些目标提供了基础。目前，行业内有多种成熟的软件过程可供借鉴，比较具有代表性、采用较广泛的软件过程主要包括以下几种：

- (1) Rational Unified Process (RUP)
- (2) OPEN Process
- (3) Object-Oriented Software Process (OOSP)
- (4) Extreme Programming (XP)
- (5) Catalysis
- (6) Dynamic System Development Method (DSDM)

## 8.2 RUP 简介

### 8.2.1 什么是 RUP 过程

Rational Unified Process (以下简称 RUP) 是一套软件工程方法, 主要由 Ivar Jacobson 的 The Objectory Approach 和 The Rational Approach 发展而来。同时, 它又是文档化的软件工程产品, 所有 RUP 的实施细节及方法导引均以 Web 文档的方式集成在一张光盘上, 由 Rational 公司开发、维护并销售, 当前版本是 5.0.RUP, 是一套软件工程方法的框架, 软件开发者可根据自身的实际情况, 以及项目规模对 RUP 进行裁剪和修改, 以制定出合乎需要的软件工程过程。

RUP 吸收了多种开发模型的优点, 具有很好的可操作性和实用性。一经推出, 凭借 Booch、Ivar Jacobson 以及 Rumbagh 在业界的领导地位以及与统一建模语言的良好集成、多种 CASE 工具的支持, 以及不断的升级与维护, 迅速得到业界广泛的认同, 越来越多的组织以它作为软件开发模型框架。

### 8.2.2 RUP 的特点

#### 1. RUP 的二维开发模型

RUP 可以用二维坐标来描述。横轴通过时间组织, 是过程展开的生命周期特征, 体现开发过程的动态结构; 纵轴以内容来组织, 是自然的逻辑活动, 体现开发过程的静态结构, 如图 8-1 所示。

注意: 后面使用到该图的时候, 给出了中文标注, 以方便读者理解。

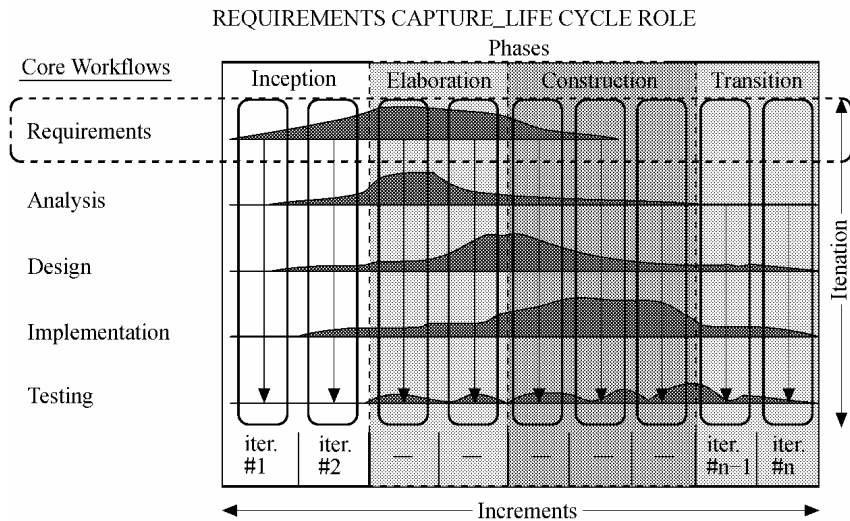


图 8-1 RUP 的二维开发模型

### (1) 瀑布模型

```

graph TD
    A[问题定义] --> B[可行性研究]
    B --> C[需求分析]
    C --> D[总体设计]
    D --> E[详细设计]
    E --> F[编程实现]
    F --> G[测试与运行]
    G --> H[维护]
  
```

瀑布模型的优点在于规定了一个易于实施的标准流程,使软件的开发不再是漫无目的的编码,做完一件事之后应该做什么,都由这个流程规定好,不会出现混乱的安排,这比依靠程序员的“个人技艺”开发软件要好得多。但是也必须看到,瀑布模型也有它与生俱来的缺陷。

瀑布模型的主要缺点是:第一,在软件开发的初始阶段就指明软件系统的全部需求是困难的,甚至是不切实际的。而瀑布模型在需求分析阶段要求客户和系统分析员必须确定系统的完整需求才能开展后续的工作。第二,需求确定后,用户和软件项目负责人要等较长的一段时间才能得到软件的最初版本。如果用户对这个软件提出较大的修改意见,那么整个项目将会蒙受

巨大的人力、财力和时间方面的损失。

因此，原始的瀑布模型在使用上具有一定的局限性。

## (2) 改进的瀑布模型

改进的瀑布模型一般被描述成可以回溯的瀑布模型，如图 8-3 所示。

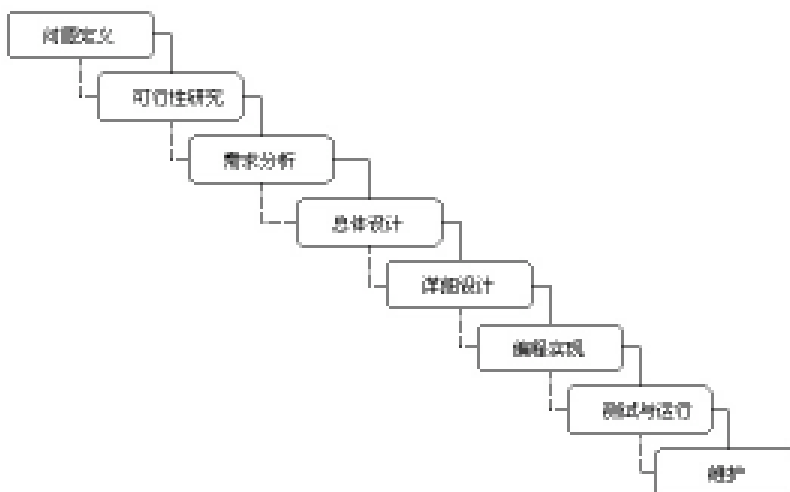


图 8-3 改进的瀑布模型

可以看到，在每一个过程进行的时候，都可以回溯到上一个过程，对本过程中发生的问题和错误进行修改，当然，回溯不仅限于一步，图 8-3 所示的情况只是最简单的一种。在实际的操作中，流程可以回溯任意多的步骤。

改进的瀑布模型解决了不能回溯的问题，避免了在软件完成后才发现与用户需求不符的情况的发生，使瀑布模型的可行性大大增强，但随之而来的新问题是：当回溯太多的时候，瀑布模型标准的优点荡然无存。因为团队不知道下一次回溯会在什么时候，也不知道下一个步骤是否需要回溯。如果项目经理的个人能力不够强，很容易造成混乱。

这时，迭代式开发的思想应运而生。

## 3. RUP 的迭代开发模型

RUP 中的每个阶段可以进一步分解为迭代。一个迭代是一个完整的开发循环，产生一个可执行的产品版本，是最终产品的一个子集，它增量式地发展，从一个迭代过程到另一个迭代过程，直到成为最终的系统。

传统上的项目组织是顺序通过每个工作流，每个工作流只有一次，也就是我们熟悉的瀑布生命周期（见图 8-4）。这样做的结果是到实现末期产品完成并开始测试，在分析、设计和实现阶段所遗留的隐藏问题会大量出现，项目可能要停止并开始一个漫长的错误修正周期。

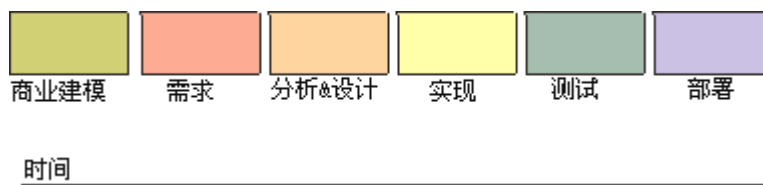


图 8-4 瀑布模型

一种更灵活，风险更小的方法是多次通过不同的开发工作流，这样可以更好的理解需求，构造一个健壮的体系结构，并最终交付一系列逐步完成的版本，这叫做一个迭代生命周期。在工作流中的每一次顺序的通过称为一次迭代。软件生命周期是迭代的连续，通过它，软件是增量开发的。一次迭代包括了生成一个可执行版本的开发活动，还有使用这个版本所必需的其他辅助成分，如版本描述、用户文档等。因此开发迭代在某种意义上是在所有工作流中的一次完整的经过，这些工作流至少包括：需求工作流、分析和设计工作流、实现工作流、测试工作流，其本身就像一个小型的瀑布项目，如图 8-5 所示。

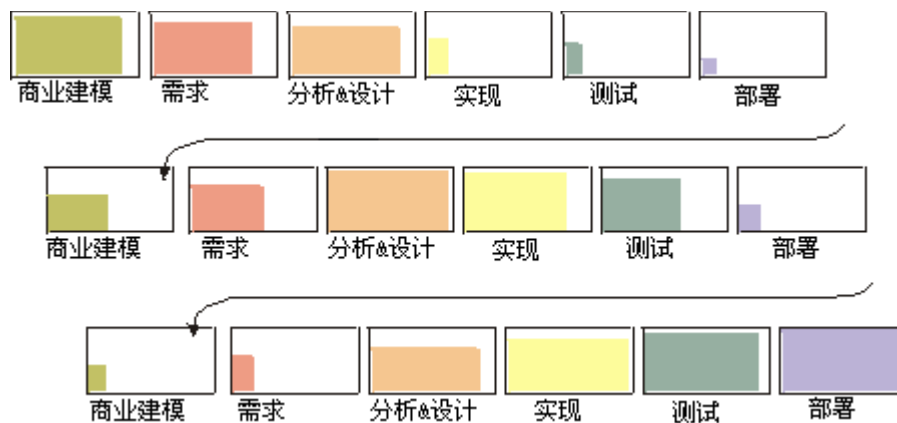


图 8-5 RUP 的迭代模型

与传统的瀑布模型相比较，迭代过程具有以下优点：

- (1) 降低了在一个增量上的开支风险。如果开发人员重复某个迭代，那么损失只是本次开发有误的迭代的花费。
- (2) 降低了产品无法按照既定进度进入市场的风险。通过在开发早期就确定风险，可以尽早解决问题而不至于在开发后期匆匆忙忙。
- (3) 加快了整个开发工作的进度。因为开发人员清楚问题的焦点所在，他们的工作会更有效率。

由于用户的需求并不能在一开始就做出完全的界定，它们通常是在后续阶段中不断细化的。因此，迭代过程这种模式使适应需求的变化会更容易。

### 8.2.3 RUP 的十大要素

#### 1. 开发前景

“有一个清晰的前景是开发满足真正需求的产品关键。”

前景抓住了 RUP 需求流程的要点：分析问题、理解需求、定义系统，当需求变化时的管理需求。前景给更详细的技术需求提供了一个高层的、有时候是合同式的基础。正像这个术语隐含的意思那样，它是软件项目的一个清晰的、通常是高层的视图，能被过程中任何决策者或者实施者借用。它捕获了非常高层的需求和设计约束，让读者能理解将要开发的系统。它还提供了项目审批流程的输入，因此就与商业理由密切相关。最后，由于前景构成了“项目是什么？”



和“为什么要进行这个项目？”，所以可以把前景作为验证将来决策的方式之一。

对前景的陈述应该能回答以下问题，而且这些问题还可以分成更小、更详细的问题：

- (1) 关键术语是什么（词汇表）；
- (2) 我们尝试解决的问题是什么（问题陈述）；
- (3) 用户是谁，他们的需求是什么；
- (4) 产品的特性是什么；
- (5) 功能性需求是什么（用例）；
- (6) 非功能性需求是什么；
- (7) 设计约束是什么。

## 2. 达成计划

“产品的质量只会和产品的计划一样好。”

在 RUP 中，软件开发计划（SDP）综合了管理项目所需的各种信息，也许会包括一些在开始阶段开发的单独的内容。SDP 必须在整个项目中被维护和更新。

SDP 定义了项目时间表（包括项目计划和迭代计划）和资源需求（资源和工具），可以根据项目进度表来跟踪项目进展。同时也指导了其他过程内容（process components）的计划：项目组织、需求管理计划、配置管理计划、问题解决计划、QA 计划、测试计划、评估计划以及产品验收计划。

在较简单的项目中，对这些计划的陈述可能只有一两句话。比如，配置管理计划可以简单的这样陈述：每天结束时，项目目录的内容将会被压缩成 ZIP 包，拷贝到一个 ZIP 磁盘中，加上日期和版本标签，放到中央档案柜中。

软件开发计划的格式远远没有计划活动本身以及驱动这些活动的思想重要。正如 Dwight D.Eisenhower 所说：“plan 什么也不是，planning 才是一切。”

## 3. 标识和减小风险

RUP 的要点之一是在项目早期就标识并处理最大的风险。项目组标识的每一个风险都应该有一个相应的缓解或解决计划。风险列表应该既作为项目活动的计划工具，又作为确定迭代的基础。

## 4. 分配和跟踪任务

有一点在任何项目中都是重要的，即连续的分析来源于正在进行的活动和进化的产品的客观数据。在 RUP 中，定期的项目状态评估提供了讲述、交流和解决管理问题、技术问题以及项目风险的机制。团队一旦发现了这些障碍物，就把所有这些问题都指定一个负责人，并指定解决日期。进度应该定期跟踪，如有必要，更新应该被发布。

这些项目“快照”突出了需要引起管理注意的问题。随着时间的变化（虽然周期可能会变化），定期的评估使经理能捕获项目的历史，并且消除任何限制进度的障碍或瓶颈。

## 5. 检查商业理由

商业理由从商业的角度提供了必要的信息，以决定一个项目是否值得投资。商业理由还可以帮助开发一个实现项目前景所需的经济计划。它提供了进行项目的理由，并建立经济约束。当项目继续时，分析人员用商业理由来正确的估算投资回报率（ROI，即 Return on Investment）。

商业理由应该给项目创建一个简短但是引人注目的理由，而不是深入研究问题的细节，以使所有项目成员容易理解和记住它。在关键里程碑处，经理应该回顾商业理由，计算实际的花费、预计的回报，决定项目是否继续进行。

#### 6. 设计组件构架

在 RUP 中，软件系统的构架是指一个系统关键部件的组织或结构，部件之间通过接口交互，而部件是由一些更小的部件和接口组成的，即主要的部分是什么？它们又是怎样结合在一起的？

RUP 提供了一种设计、开发、验证构架的系统的方法。在分析和设计流程中包括以下步骤：定义候选构架、精化构架、分析行为（用例分析）和设计组件。

要陈述和讨论软件构架，必须先创建一个构架表示方式，以便描述构架的重要方面。在 RUP 中，构架表示由软件构架文档捕获，它给构架提供了多个视图。每个视图都描述了某一组用户所关心的正在进行的系统的某个方面。用户有设计人员、经理、系统工程师和系统管理员等。这个文档使系统构架师和其他项目组成员能就与构架相关的重大决策进行有效的交流。

#### 7. 对产品进行增量式的构建和测试

在 RUP 中实现和测试流程的要点是在整个项目生命周期中增量的编码、构建和测试系统组件，在开始之后每个迭代结束时生成可执行版本。在精化阶段后期，已经有了一个可用于评估的构架原型；如有必要，可以包括一个用户界面原型。然后，在构建阶段的每次迭代中，组件不断的被集成到可执行、经过测试的版本中，不断地向最终产品进化。动态及时的配置管理和复审活动也是这个基本过程元素的关键。

#### 8. 验证和评价结果

顾名思义，RUP 的迭代评估捕获了迭代的结果。评估决定了迭代满足评价标准的程度，还包括学到的教训和实施的过程改进。

根据项目的规模和风险以及迭代的特点，评估可以是对演示及其结果的一条简单的记录，也可能是一个完整的、正式的测试复审记录。

此处的关键是既关注过程问题又关注产品问题。越早发现问题，就越没有问题。

#### 9. 管理和控制变化

RUP 的配置和变更管理流程的要点是当变化发生时管理和控制项目的规模，并且贯穿整个生命周期。其目的是考虑所有的用户需求，尽可能的满足，同时仍能及时的交付合格的产品。

用户拿到产品的第一个原型后（往往在这之前就会要求变更），他们会要求变更。重要的是，变更的提出和管理过程始终保持一致。

在 RUP 中，变更请求通常用于记录和跟踪缺陷和增强功能的要求，或者对产品提出任何其他类型的变更请求。变更请求提供了相应的手段来评估一个变更的潜在影响，同时记录就这些变更所做出的决策。他们也帮助确保所有的项目组成员都能理解变更的潜在影响。

#### 10. 提供用户支持

在 RUP 中，部署流程的要点是包装和交付产品，同时交付有助于最终用户学习、使用和维护产品的任何必要的材料。

项目组至少要给用户提供一个用户指南（也许是通过联机帮助的方式提供），可能还有一

个安装指南和版本发布说明。

根据产品的复杂度，用户也许还需要相应的培训材料。最后，通过一个材料清单（BOM 表，即 Bill of Materials）清楚地记录应该和产品一起交付那些材料。

## 11. 十大要素的应用

### （1）对于非常小的项目

首先，在一个非常小的、没有经验的项目组（才学了 RUP）中，使用 RUP 和 Rational 开发工具来构造一个简单的产品，可以参考以上十大要素，以使项目组不被 RUP 的细节和 Rational Suites 的功能压垮。

实际上，即使没有任何自动化工具也可以实施十大要素。管理一个小项目，一个项目笔记本，就是一个非常好的起点，可以把它分成 10 个部分，每一部分专用于十大要素中的一个要素。

### （2）对于增长的项目

当然，当一个项目的规模和复杂度增长时，应用十大要素的简单方法很快就变得不可操作，而对自动化工具的需求就变得比较明显了。项目的领导者一般刚开始时应用十大要素和 RUP 的“最佳实践”，需要时再逐步增加支持工具，而不是一下子就尝试使用全套 Rational Suites。

### （3）成熟的项目团队

对成熟的项目团队而言，可能已经在采用某种软件过程和使用 CASE 工具，十大要素可以提供一种快速评估方法，用来评估关键过程元素的平衡性，标识它们并确定改进的优先级。

### （4）对于所有的项目

各个项目都不太一样，有些项目似乎并不真正需要所有的要素。在这些情况下，重要的是考虑：如果团队忽视某个要素后会发生什么问题。举例说明如下。

- 没有前景——你会迷失方向，走很多弯路，把力气浪费在毫无结果的努力上。
- 没有计划——你将无法跟踪进度。
- 没有风险列表——你的项目会陷入“专注于错误的问题”的危险里面，可能一下子被一个没有检测的地雷击倒，并为此付出 5 个月的代价。
- 没有问题列表——没有定期的问题分析和解决，小问题会演变成大问题。
- 没有商业理由——你在冒浪费时间和金钱的风险。项目最终要么超支，要么被取消。
- 没有构架——在出现交流、同步和数据存取问题时，你可能无法处理。
- 没有产品（原型）——你将不能有效的测试，并且会失去客户的信任。
- 没有评估——你将没有办法掌握实际情况与项目目标、预算和最后期限之间的距离。
- 没有变更请求——你将无法估计变更的潜在影响，无法就互相冲突的需求确定优先级，无法在实施变更时通知整个项目组。
- 没有用户支持——用户将不能最有效的使用产品，技术支持人员也会淹没在大量支持请求中。

## 8.3 统一开发过程核心 workflow

迭代式开发不仅仅解决了瀑布模型不可回溯的缺点，也保留了瀑布模型规则化、流程化的优点。而 Rational 公司提供的统一流程 RUP ( Rational Unified Process , Rational 统一过程 ) 就是以迭代式开发为基础的，如图 8-6 所示。

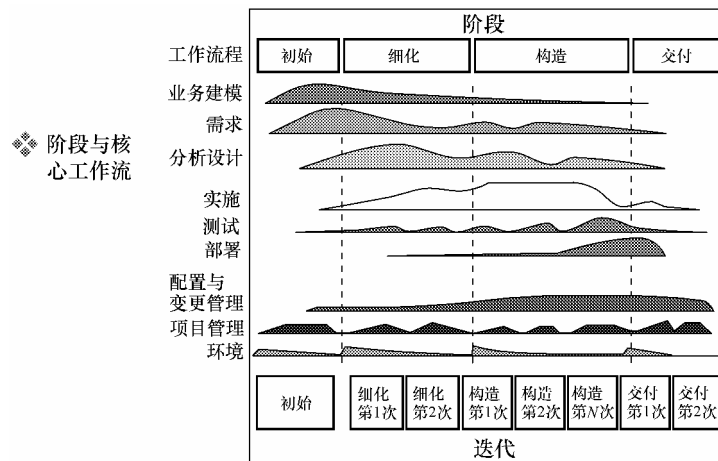


图 8-6 Rational 统一过程

可以看到，RUP 中包括初始（也称为先启）、细化（精化）、构造（构建）和交付（产品化）4 个阶段，以及业务建模、需求、分析设计、实施（实现）、测试、部署、配置与变更管理、项目管理、环境 9 个核心工作流程。每个阶段都是由一次或多次迭代所组成。下面对 RUP 的 4 个阶段做一些详细的介绍。

### （1）初始

初始也称为初始的目标，是“获得项目的基础”。初始阶段的主要人员是项目经理和系统设计师，他们所要完成的任务包括对系统的可行性分析，即软件系统的实现技术和经济方面的可行性；创建基本需求以有助于界定系统范围；识别软件系统的关键任务。

初始的焦点是需求和分析工作流。

### （2）精化

精化阶段的主要目标是创建可执行构件基线；精化风险评估；定义质量属性；捕获大部分的系统功能需求用例；为构造阶段创建详细计划。精化阶段并不是要创建真正可执行的系统，也不是要创建原型，这个阶段只需要展现用户所期望系统。精化是开发过程最重要的阶段，因为在以后的阶段是以精化的结果为基础的。

精化的焦点是需求、分析和设计工作流。

### （3）构建

构建的主要目标是完成所有的需求、分析和设计。精化阶段的制品将演化成最终系统，构建的主要问题是维护系统框架的完整性。开发人员应该避免由于软件编码问题而造成最终系统低质量、高维护成本的情况的发生。

构建的焦点是实现工作流。

#### (4) 交付

交付是完整的系统部署到用户所处的环境。交付阶段的目标包括修复系统缺陷；为用户环境准备新软件；如果出现不可预见问题则修改软件；创建用户使用手册和系统文档；提供用户咨询。

交付的焦点是实现和测试 workflow。

下面将围绕制品、工作人员和 workflow 3 个方面，针对几个关键流程中所应用到的 UML 过程进行说明。

### 8.3.1 需求捕获 workflow

软件需求是指用户对目标软件系统在功能、行为、性能和设计约束等方面的期望。需求捕获就是通过对对应问题的理解和分析，确立问题涉及的信息、功能和系统行为、将用户需求精确化、完全化。如图 8-7 所示，需求的焦点主要在初始和精化阶段，在精化阶段后期，需求捕获的工作量大幅下降。

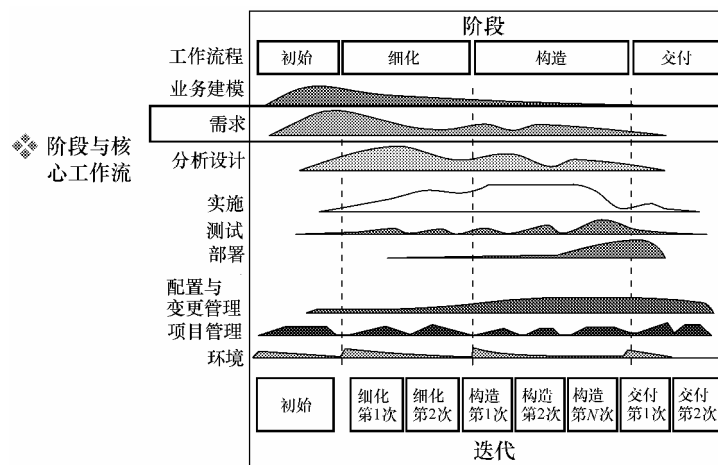


图 8-7 需求捕获 workflow

#### 1. 制品

在需求捕获 workflow，主要的 UML 制品包括用例模型、参与者、用例、构架描述、术语表 and 用户界面原型。

##### ■ 用例模型（Use Case Model）

用例模型主要包括系统参与者、用例以及它们之间的关系。用例模型可以被视为软件开发人员和软件客户之间的“桥梁”，这座“桥梁”可以帮助软件开发人员和客户在需求方面达到共识。

UML 允许用各种图来展现用例模型，这些图从不同的角度来表示参与者和用例。

##### ■ 参与者（Actor）

用户模型主要用于描述系统能为各种类型的用户做些什么，这些用户作为相关工作单元部分直接与系统进行交互，它们可以是人类用户或其他系统，如图 8-8 所示。



图 8-8 参与者

### ■ 用例 (Use Case)

用例定义了系统所提供的功能和行为单元。可以认为,参与者使用系统的每种方式都可以表示为一个用例,如图 8-9 所示。



图 8-9 用例

在 UML 术语中,一个用例往往被认为是一个类元,也就是说,它具有操作 (Java 里称为方法) 和属性。因此,用例可以由顺序图和协作图来详细描述。

### ■ 构架描述

构架描述阐述了对构架来说重要的和关键性功能的用例,它包括用例模型的构架视图。相应的用例实现包括在分析和设计的模型的框架视图中。

### ■ 术语表 (Glossary)

每个业务领域具有自己独特的语言,需求分析的目的在于理解和捕获这些语言。术语表提供了主要业务术语和定义字典。术语表有利于开发人员之间就各种概念和观点的定义达成一致,从而降低了由于开发人员间的理解差异而造成错误发生的可能性。

### ■ 用户界面原型

用户界面原型可以在需求捕获期间由客户理解和确定参与者和系统之间的交互,这不仅有助于开发更好的用户界面,而且有助于更好地理解用例。

## 2. 工作人员

参与需求捕获阶段的工作人员有系统分析人员、用例描述人员、用户界面设计人员和构架设计师。

### ■ 系统分析人员 (System Analyst)

系统分析人员在需求捕获阶段作为建模的协调者和领导者负责界定系统,确定参与者和用例,并确保用例模型是完整的、一致的。在需求捕获阶段,系统分析人员主要负责用例模型、参与者和术语表 3 个制品。

---

注意:系统分析人员的职责相对宏观,尽管系统分析人员负责确定软件系统的用例模型和参与者,但并不对每个单独的用例负责(这些工作通常由专门的用例描述人员完成)。

---

### ■ 用例描述人员 (Use Case Specifier)

捕获需求是软件开发过程中极为重要的阶段,它直接影响到用户对最终的软件产品的满意度。这个过程一般需要多人共同完成,系统分析人员以及其他工作人员相互协助,来对一个或多个用例进行详细描述,这些工作人员称为用例描述人员。

### ■ 用户界面设计人员 (User Interface Designer)

用户界面设计人员负责对用户界面进行可视化定型。

### ■ 构架设计师 (Architect)

构架设计师也参与需求 workflow，这有利于描述用例模型的构架视图。

### 3. 工作流

需求捕获的工作流主要包括 5 个活动：确定参与者和用例、区分用例的优先级、详细描述一个用例、构造用户界面原型以及构造用例模型。

#### ■ 确定参与者和用例

确定参与者和用例的目的是从环境中界定系统；概述哪些参与者将与系统进行交互，以及他们将从系统中得到哪些功能（用例）；捕获和定义术语表中的公用术语，这是对系统功能进行详细说明的基础，如图 8-10 所示。

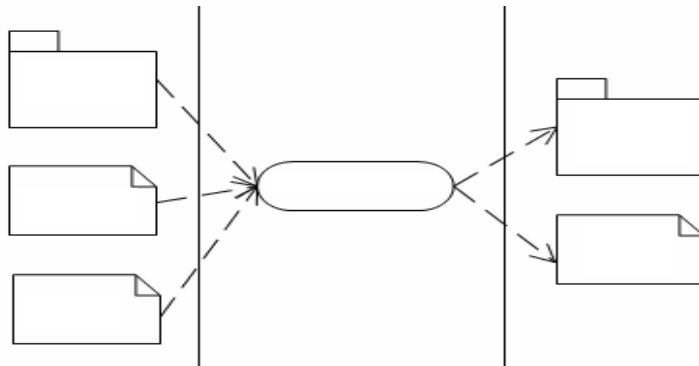


图 8-10 确定参与者和用例

确定参与者和用例的过程通常包括 4 个步骤：确定参与者；确定用例；简要描述每个用例；整体上描述用例模型。实现中，这些步骤通常是并发执行的，系统分析人员和用例描述人员没有必要顺序的执行。

#### ■ 区分用例优先级

区分用例优先级是为了决定用例模型中哪些用例需要在早期的迭代中进行开发（包括分析、设计和实现等），以及哪些用例可以在随后的迭代中进行开发，如图 8-11 所示。

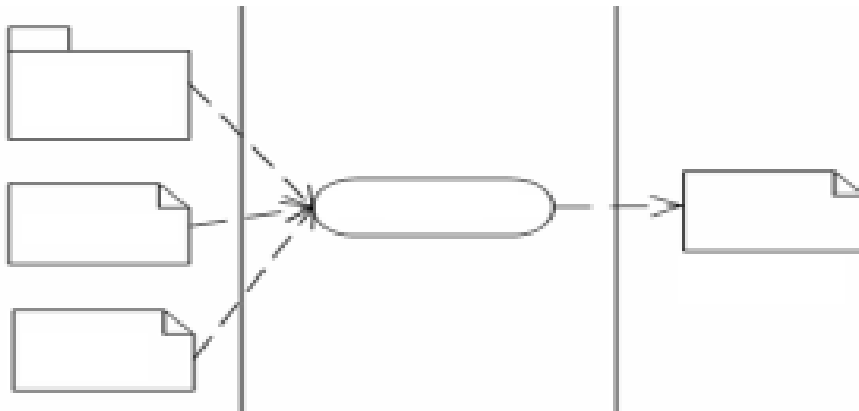


图 8-11 区分用例优先级

■ 详细描述一个用例

详细描述用例的主要目的是为了详细描述事件流。这个活动包括构造用例说明、确定用例说明中包括的内容、对用例说明进行形式化描述 3 个步骤，最终的结果是以图或文字表示的某用例的详细说明，如图 8-12 所示。

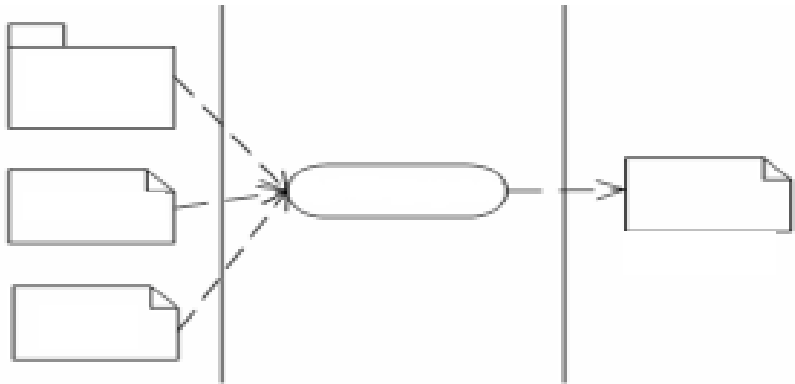


图 8-12 详细描述一个用例

■ 构造用户界面原型

在系统分析人员建立起用例模型，确定了用户是谁以及他们要用系统做什么后。接下来的工作就是要着手设计用户界面。这个活动由逻辑用户界面设计、实际用户界面设计和构造原型两部分组成，最终的结果是生成一个用户界面简图和用户界面原型，如图 8-13 所示。

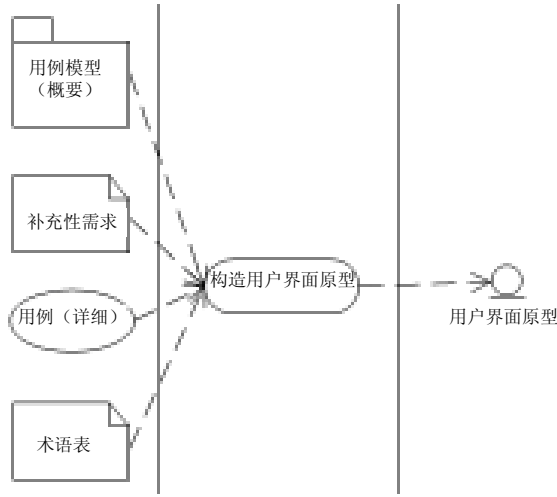


图 8-13 构造用户界面原型

■ 构造用例模型

构造用例模型的主要目的是提取可以被更具体的用例说明所使用的通用的和共享的用例功能说明；提取可以扩展更具体的用例说明的补充的或可选的用例功能说明。这个活动由确定

术语表  
用例模型  
(概要)

补充性需求

术语表



共享的功能性说明、确定补充和可选的功能说明、确定用例之间的其他关系 3 部分组成。

在完成了确定系统用例和参与者之后，系统分析人员可以重新构造用例的完整集合，以使模型更易于理解 and 处理，如图 8-14 所示。

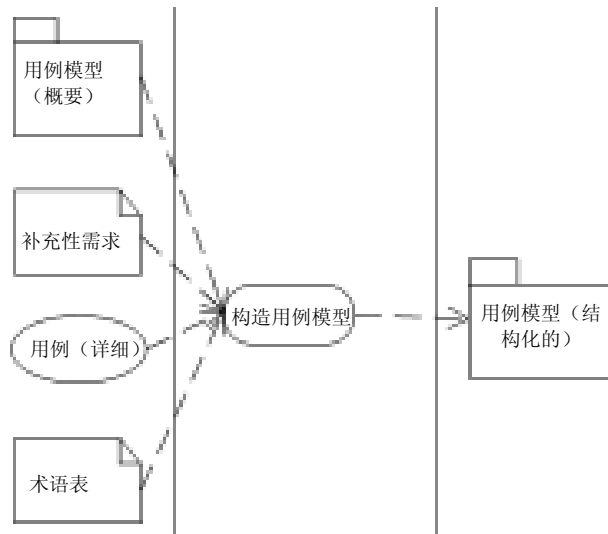


图 8-14 构造用例模型

### 8.3.2 分析 workflow

从图 8-15 可知，分析的主要工作开始于初始阶段的结尾，和需求一样是精化阶段的主要焦点。精化阶段的大部分活动是捕获需求，分析工作于需求捕获在很大程度上重叠，实际上，这两种活动是相辅相成的，在对系统进行需求捕获的同时往往会加入一些分析。

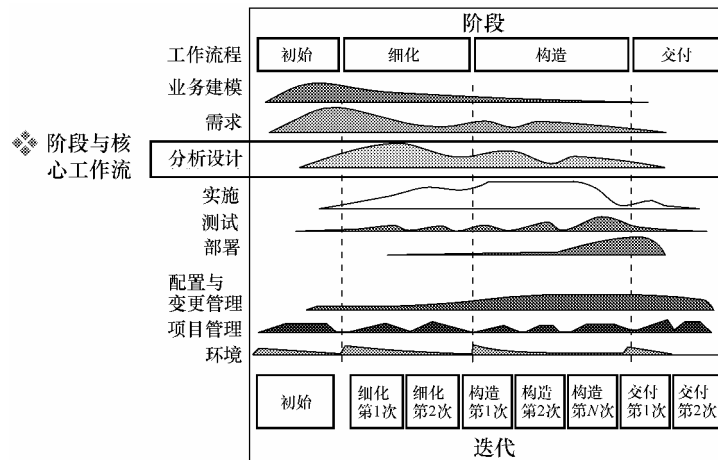


图 8-15 分析 workflow

## 1. 制品

在分析工作流程期间，主要的 UML 制品包括分析模型、分析类、用例实现（分析）、分析包和构架描述。

### ■ 分析模型

分析模型由代表该模型顶层包的分析系统来表示。

### ■ 分析类

分析类代表问题域中的简捷抽象，它映射到现实世界。分析类应该以清晰的、无歧义的方式映射到现实世界，如产品或账户。一般来说，分析类总能符合 3 种基本构造型中的一种：边界类、控制类和实体类。每一种构造型都有具体的语义。

**边界类：**用于建立系统与其参与者之间交互的模型。这种交互通常包括接收来自用户和外部系统的信息与请求，以及将信息与请求提交到用户和外部系统。

**实体类：**用于对长效且持久的信息建模，主要是对诸如个体、实际对象或实际事件的某些现象或概念的信息及相关行为建模。

**控制类：**代表协调、排序、事务处理以及对其他对象的控制，经常用于封装与某个具体用例有关控制。控制类还可以用来表示业务逻辑之类与系统存储的任何具体的长效信息都没有关系的对象。

### ■ 用例实现（分析）

在找出分析类后，分析的关键就是找出用例实现。用例实现由一组类所组成，这些类实现了用例中所说明的行为。分析类图是用例实现的关键部分，一组类是相关的，这些类的实类能够协作以实现由一个或多个用例所说明的行为。

### ■ 分析包

分析包提供了一种以可管理分块的方式对分析模型的制品进行组织的方法。分析包中可以包含用例、分析类、用例实现和其他分析包。分析包反映了元素真正的语义分组，而不仅是一些逻辑框架的假想视图。

### ■ 构架模型

构架描述阐述了对构架重要的制品，它包括分析模型的构架视图。

## 2. 工作人员

在分析工作流程期间，所参与的工作人员有构架设计师、用例工程师和构件工程师。

### ■ 构架设计师

构架设计师对分析模型的完整性负责，确保分析模型作为一个整体的正确性、一致性和易读性。在分析 workflow 中，构架设计师负责分析模型和构架描述两个制品。他不需要对分析模型中各种制品的持续开发和维护负责，这些是相关的用例工程师和构建工程师的职责。

### ■ 用例工程师

用例工程师负责一个或几个用例实现的完整性，确保它们实现了各自的需求。用例工程师不仅要完成用例的分析，还要完成用例的设计。

在分析 workflow 中，用例工程师主要负责用例实现（分析）这一个制品。

### ■ 构件工程师

构件工程师定义并维护一个或多个分析类，确保每个分析类能实现来自他所参与的各个用

例实现的需求；维护一个或多个包的完整性。

在分析工作流中，构件工程师负责分析类和分析包两个制品。

### 3. 工作流

分析工作流主要包括 4 个活动：构架分析、分析用例、分析类和分析包。

#### ■ 构架分析

构架分析的目的在于通过确定分析包、鲜见的分析类和共用的特殊需求来概述分析模型和构架，如图 8-16 所示。

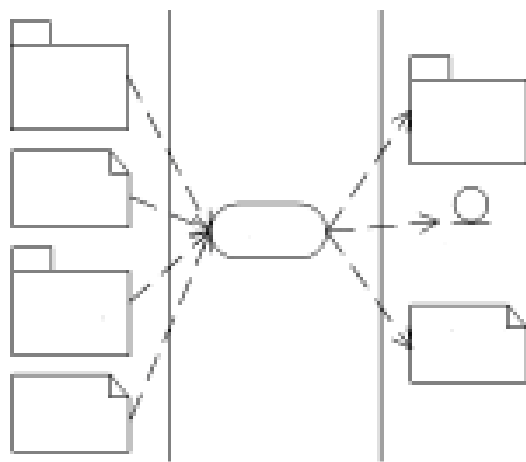


图 8-16 构架分析

#### ■ 分析用例

分析用例的目的在于：确定分析类；将用例的行为分配给相交互的分析对象；以及捕获用例实现中的特定需求。具体的输入和结果如图 8-17 所示。

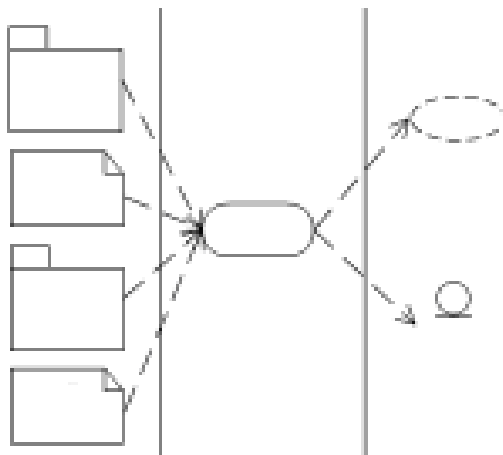


图 8-17 分析用例

### ■ 分析类

分析类的目的在于：依据分析类在用例实现中的角色来确定和维护它的职责；确定和维护分析类的属性及其关系；捕获对该分析类实现的特殊需求。具体的输入和结果如图 8-18 所示。

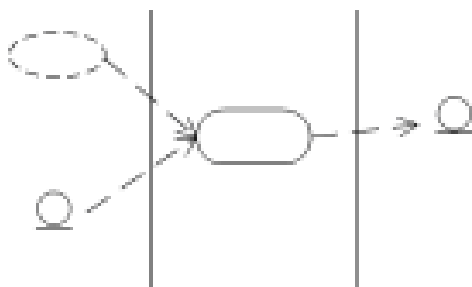


图 8-18 分析类

### ■ 分析包

分析包的目的在于：确保该分析包尽可能与其他分析包无关；确保该分析包能够实现一些领域内用例的目的；以及能够对未来变化的影响进行估计。

一般来说，分析包活动一般做法是：定义和维护该包与其他包的依赖；确保包中包含恰当的类；限制对其他包的依赖。

具体的输入和结果如图 8-19 所示。

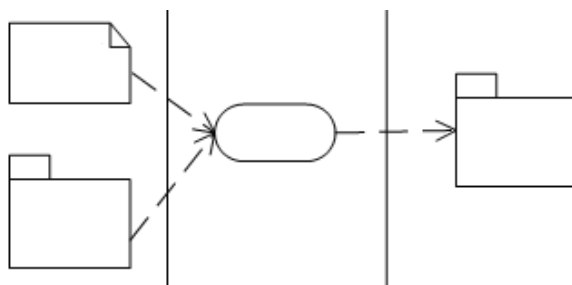


图 8-19 分析包

## 8.3.3 设计 workflow

设计 workflow 的主要工作是位于精化阶段的最后部分和构造阶段的开始部分的主要建模活动（见图 8-15）。系统建模最初的焦点是需求和分析，在分析活动逐步完善后，建模的焦点开始转向设计。正如图 8-15 所示，分析和设计可以是同时进行的，因此在这张图中它们合并为一个 workflow。

### 1. 制品

在设计 workflow 期间，包括设计模型、设计类、用例实现（设计）、设计子系统、接口、构架描述（设计模型）、实施模型和构架描述（实施模型）。

### ■ 设计模型

设计模型是一个用于描述用例物理实现的对象模型。设计模型作为系统实现的抽象用于系统实现中活动的基本输入。设计模型由设计系统来表示，设计系统为模型的高层子系统。

### ■ 设计类

设计类是已经完成了规格说明并能够被实现的类。设计类来源于问题域和解域。在分析阶段，系统分析师着重于确定系统需要的行为，较少考虑如何去实现这些行为；在设计阶段，构架设计师必须准确的说明类的属性集合以及分析类指定的操作转化成一个或多个方法的完整集合。

设计良好的分析类应该包括如下基本特征：完整的和充分的、原始的、高内聚、低耦合。设计类需要被完整的说明，因为它将会被交给程序员编写成实际的源代码，或使用 CASE 工具直接生成代码。

### ■ 用例实现 - 设计

用例实现 - 设计是实现用例的实际对象和设计类的协作，即在设计模型内的协作关系，以设计类及对象为基础，描述了一个特定用例的实现和执行。

用例实现由设计交互图、包含参与设计类的类图两部分组成，这是精化分析交互图和类图以显示设计制品的过程。

### ■ 设计子系统

设计子系统提供了一种将设计模型中的制品组织成为更易于管理的功能块的方法，它主要用于分离设计、代表粗颗粒度的构件、打包遗留系统。一个设计子系统可以包含设计类、用例实现、接口和其他的子系统。

### ■ 接口

接口用于详细说明由设计类和子系统所提供的操作。接口的关键思想在于通过类或子系统将功能规格说明同它的实现分离。

接口的每个操作必须包括完整的操作签名；操作语义；可选的构造型、可选的约束和标记值的集合。接口不能包括属性、操作实现、从接口到其他物质的导航。

### ■ 配置图

在设计时产生初步的配置图，用来表明软件系统是如何分布在物理的计算节点上的。从图 8-4 可以了解到，部署的大部分发生在实施阶段。

## 2. 工作人员

参与设计工作流的工作人员包括构架设计师、用例工程师和构件工程师。

### ■ 构架设计师

在设计工作流中，构架设计师的主要工作是保持设计和实施模型的完整性，并保证模型整体上不会发生错误且容易被其他开发人员理解。

在实际开发中，架构工程师必须对设计模型和实施模型的架构负责，但他不需要关心设计模型中的各种制品的继续开发以及维护。

### ■ 用例工程师

用例工程师的工作是保证系统中用例实现(设计)的完整性，以确保它们实现特定的需求。这里所说的完整性是指用例实现的图形以及文字必须易读且符合系统用途。

### ■ 构件工程师

构件工程师的工作是定义和维护设计类的操作、方法、属性、关系以及实现性需求，以保证每个设计类实现特定的需求。构建工程师往往还需要负责子系统的完整性以及该子系统所包含的模型元素。

### 3. 工作流

设计工作流中，主要包括 4 种活动：构架设计、设计一个用例、设计一个类和设计一个子系统。

#### ■ 构架设计

构架设计是设计阶段首要进行的活动，主要目的是通过节点及其网络配置、子系统及其接口，以及对构架有重要意义的设计类（如主动类）的识别，来勾划设计和实施模型及其构架。具体的输入与产出如图 8-20 所示。

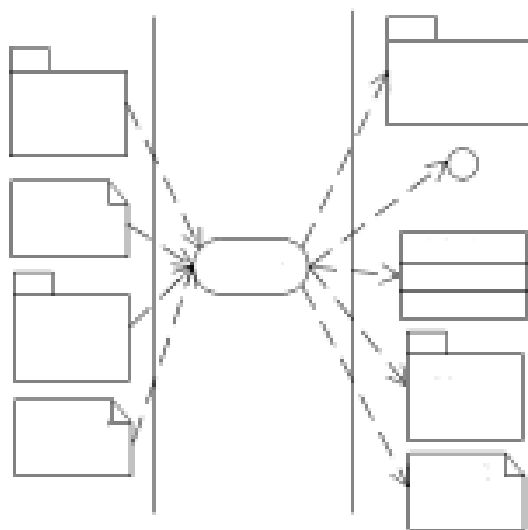


图 8-20 构架设计

#### ■ 设计一个用例

设计一个用例主要过程包括 4 个部分：识别设计类或子系统，实类需要去执行用例的事件流；把用例的行为分布到有交互作用的设计对象或所参与的子系统；定义对设计对象或子系统及其接口的操作需求；为用例捕获实现性需求。具体的输入和产出如图 8-21 所示。

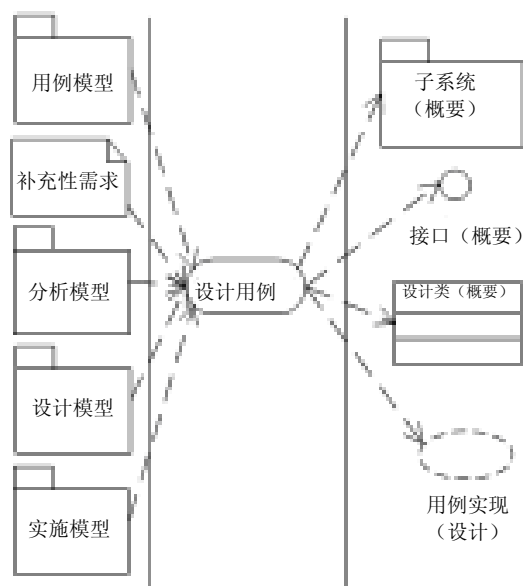


图 8-21 设计一个用例

#### ■ 设计一个类

设计一个类，当然是要创建一个设计类，这个设计类能够实现其在用例实现中以及非功能性需求中所要求的角色。

设计类要具有一些方面的内容：操作，属性，所参与的关系，实现操作的方法，强制状态，对任何通用设计机制的依赖，与实现相关的需求，以及需要提供的任何接口的正确实现。

具体的输入和产出如图 8-22 所示。

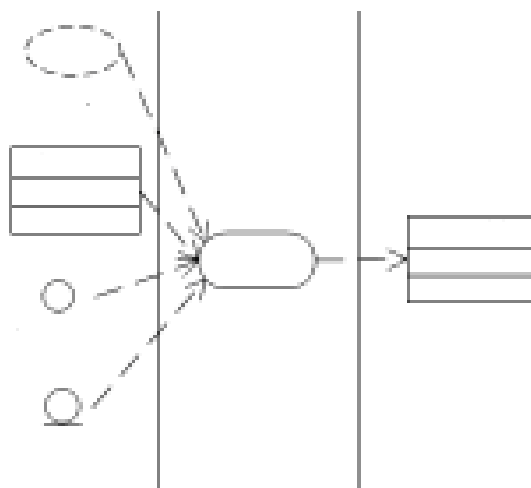


图 8-22 设计一个类

■ 设计一个子系统

设计一个子系统由 3 个目的：为了确保该子系统尽可能的独立于别的子系统或它们的接口；为了确保该子系统提供正确的接口；为了确保该子系统实现其目标，即提供其接口所定义操作的正确实现。具体的输入和产出如图 8-23 所示。

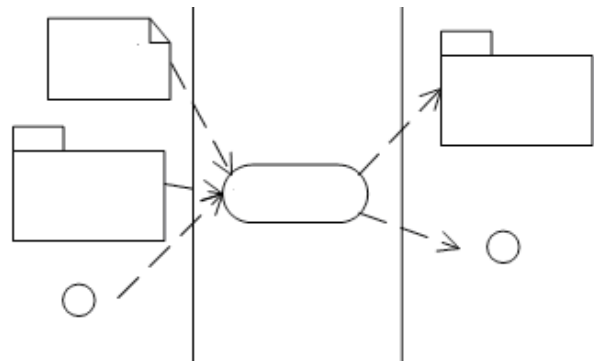


图 8-23 设计一个子系统

8.3.4 实现 workflow

实现（也称之为实施）是把设计模型转换成可执行代码的过程。从系统分析师或系统设计师的角度看，实现 workflow 的重点就是完成软件系统的可执行代码。

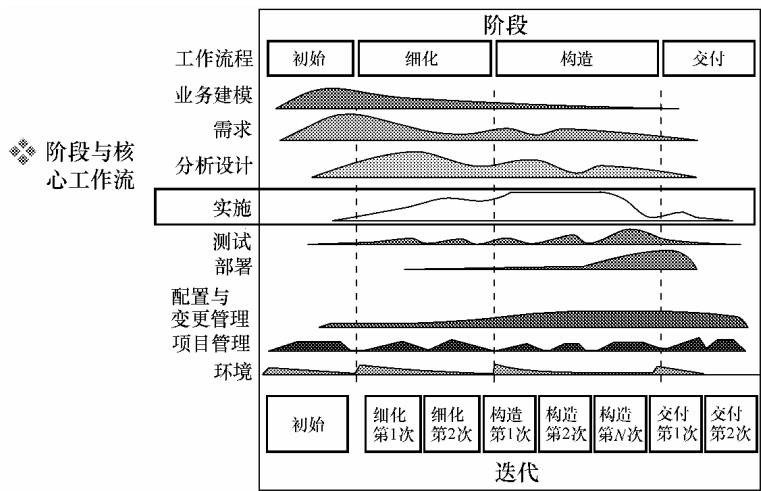


图 8-24 实现 workflow

如图 8-24 所示，实现 workflow 是构建阶段的焦点。在这里需要指出，生成系统的实现模型仅仅是实现 workflow 的副产品，开发人员应该着重于开发系统的代码。事实上，CASE 工具，例如 Rational Rose，都允许对源代码进行逆向工程从而得到实现模型。



## 1. 制品

在实现工作流中，主要有 6 种制品：实现模型、构件、实现子系统、接口、构架描述（实现模型）和集成构造计划。

### ■ 实现模型

实现模型是一个包含构建和接口的实现子系统的层次结构。实现模型描述如何使用源代码文件、可执行体等构件来实现设计模型中的元素，还描述构件是如何组织起来的，以及这些构件之间是的依赖关系。

### ■ 构件

常见构件（或称构造型）主要包括以下几种：

<<EXE>>，表示一个可以在节点上运行的程序；

<<Database>>，表示一个数据库；

<<Application>>，表示一个应用程序；

<<Document>>，表示一个文档。

构件可以被认为是系统中可替换的物理部件，它包装了实现而且遵从并提供一组接口的实现。一般来说，实现模型中的构件是跟踪依赖于设计模型中的某个类的。

### ■ 实现子系统

实现子系统提供一种把实现模型的制品组织成更易于管理的功能块的方法。一个子系统可以包含构件或接口，并可以实现并提供接口。

在设计工作流中提及了设计子系统，它主要用于管理设计模型中的制品。设计子系统与相应的实现子系统间存在着一对一的跟踪关系。

### ■ 接口

实现工作流必须按正确的方式实现一个接口所定义的全部操作。提供接口的实现子系统也必须包含提供该接口的构件。

### ■ 构架描述（实现模型）

构架描述（实现模型）阐述了对构架有重要意义的制品。构架模型中的下列制品对构架有重要意义：实现模型分解成子系统、子系统接口及它们之间的依赖关系的分解，以及关键的构件。

### ■ 集成构造计划

考虑到增量的构造方式，在每一步增量中所要解决的集成问题就很少，每一步增量的结果称为“构造”，它是系统的一个可执行版本，包括了部分或全部的系统功能。集成构造计划就是对每一步增量工作和工作结果的描述。

## 2. 工作人员

参与实现工作流的工作人员有构架设计师、构件工程师和系统集成人员。

### ■ 构架设计师

在实现工作流中，构架设计师的主要工作是保持实现模型的完整性，并保证模型整体不会发生错误且容易被其他开发人员理解。

在实际开发中，架构工程师必须对实现模型架构以及可执行体与节点间的映射负责，但他不需要关心实现模型中的各种制品的继续开发以及维护。

### ■ 构件工程师

构件工程师定义和维护一个或多个文件构件的源代码,以确保每个构件能够正确实现其功能。由于实现子系统与设计子系统是一一对应的,构建工程师往往还需要保证实现子系统的内容的正确性。

### ■ 系统集成人员

系统集成工作不属于构件工程师的工作范围,这项工作由专人(系统集成人员)负责。系统集成人员的工作包括规划在每次迭代中所需的构造序列,并对每个构造的各部分已经实现的进行集成。规划的结果是一个集成构造计划。

## 3. workflow

在实现 workflow 中,包括一系列活动:构架实现、系统集成、实现一个子系统、实现一个类和执行单元测试。

### ■ 构架实现

构架实现的主要流程为:识别对构架有重要意义的构件,例如可执行构件;在相关的网络配置中将构件映射到节点上。

构架实现由构架设计师负责,主要的输入和制品如图 8-25 所示。

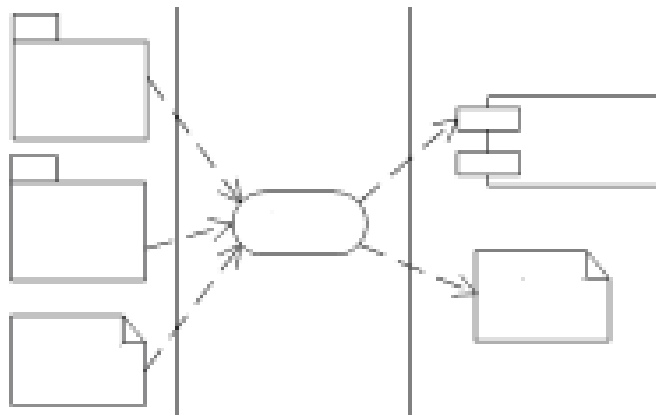


图 8-25 构架实现

### ■ 系统集成

系统集成的主要流程为:创建集成构造计划,描述迭代中所需的构造和对每个构造的需求;在进行集成测试前集成每个构造。

系统集成由系统集成人员负责,主要的输入和制品如图 8-26 所示。

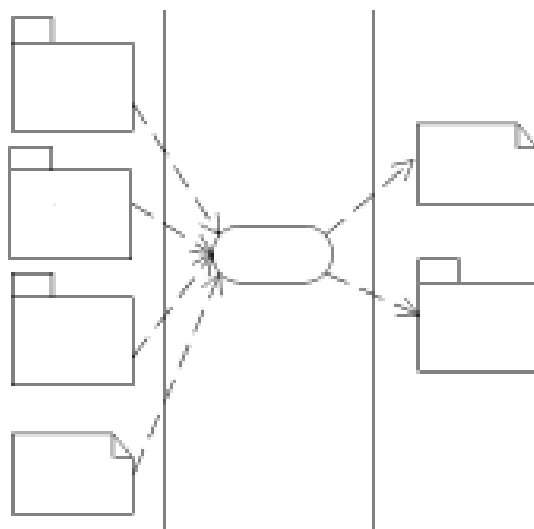


图 8-26 系统集成

#### ■ 实现一个子系统

实现一个子系统的目的是确保一个子系统履行它在每个构造中的角色，由构件工程师负责，主要的输入和制品如图 8-27 所示。

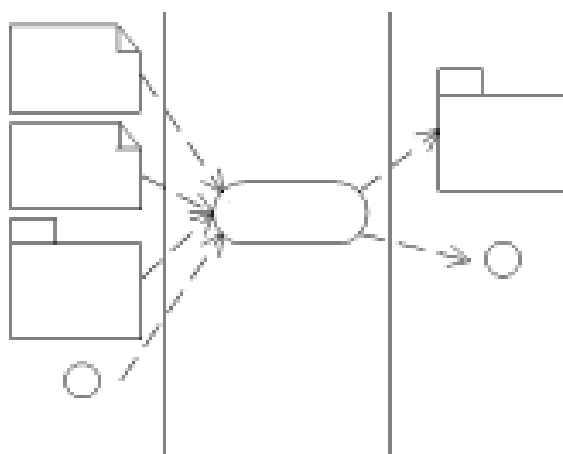


图 8-27 实现一个子系统

#### ■ 实现一个类

实现一个类是为了在一个文件构件中实现一个设计类，主要流程为：勾画出将包含源代码的文件构件；从设计类及其所参与的关系中生成源代码；按照方法实现设计类的操作；确保构件提供与设计类相同的接口。

实现一个类由构件工程师负责，主要的输入和制品如图 8-28 所示。

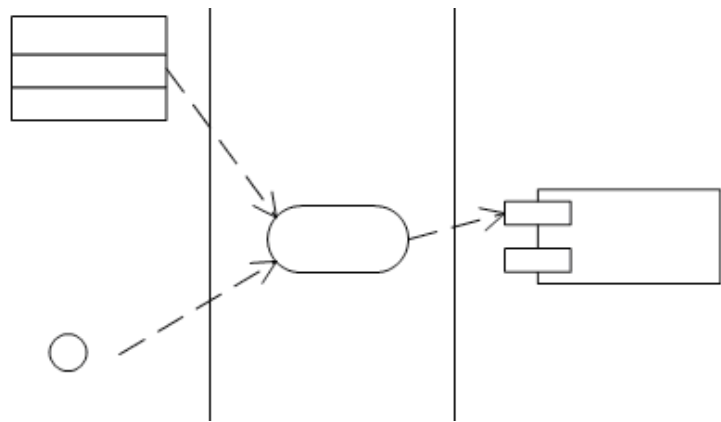


图 8-28 实现一个类

■ 执行单元测试

执行单元测试是为了把已实现的构件作为个体单元进行测试，由构件工程师负责，主要的设计类（实现）输入和制品如图 8-29 所示。

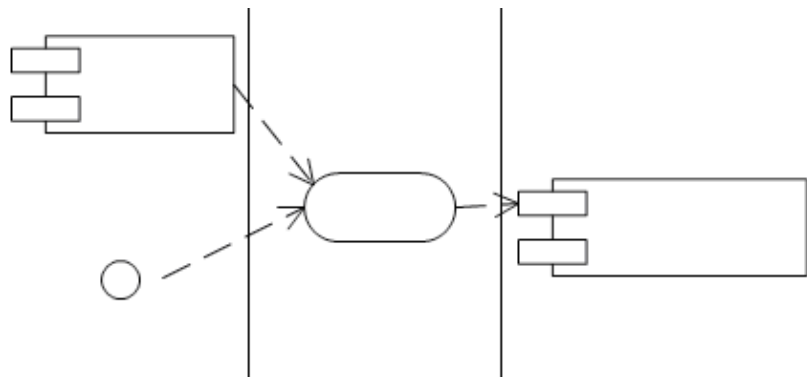


图 8-29 执行单元测试

8.3.5 测试 workflow

在完成需求捕获、分析、设计和实现等阶段的开发后，得到了源代码，这时就必须开始寻找软件产品中可能存在的错误与缺陷。如果不能及时发现这些错误，软件产品很可能不能使用甚至造成巨大的损失。测试是一项相当主要的工作，其工作量约占开发总工作量的 40% 以上，对于某些有特殊安全要求的软件产品，测试的成本甚至是开发成本的 3 ~ 5 倍。

由图 8-30 可知，测试 workflow 贯穿于软件开发的整个过程。它开始于软件开发的初始阶段，而细化阶段和构造阶段是测试的焦点。

在开始介绍测试 workflow 之前，注意一点：测试是为了找出程序中的错误与权限，而不能证明程序无错。

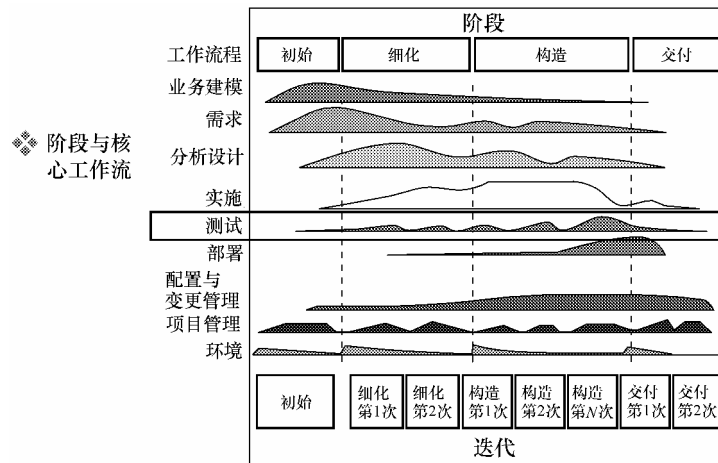


图 8-30 测试工作流

### 1. 制品

测试工作流中，包括 7 个制品：测试模型、测试用例、测试规程、测试构件、制定测试计划、缺陷和评估测试。

#### ■ 测试模型

测试模型是测试用例、测试规格以及测试构件的集合。它主要描述通过集成测试和系统测试对实现模型中的可执行构件进行测试的方法。测试模型中存在着包，它用于管理那些将在测试中使用的测试用例、测试规格以及测试构件。

#### ■ 测试用例

测试用例详细说明了用输入或结构去测试什么样的内容，以及在何种条件下进行测试。

#### ■ 测试规程

测试规程说明了怎样执行一个或几个测试用例或者其中的一部分。一个测试用例可由一个测试规则说明，也可对不同的测试用例重用同样的测试规则。

#### ■ 测试构建

测试构件自动执行一个或几个测试规程或者其中的一部分。它通常使用脚本语言或编程语言来开发。

#### ■ 测试计划

测试计划详细规定了测试策略、资源和进度安排。

#### ■ 缺陷

缺陷及系统异常。测试的目的就是在软件交付前尽可能多的找出并更正系统的缺陷。

#### ■ 评估测试

评估测试是对测试工作各项结构的评估。

### 2. 工作人员

参与测试工作流的工作人员主要有 4 类：测试设计人员、构件工程师、集成测试人员以及系统测试人员。

#### ■ 测试设计人员

测试设计人员是测试工作流的“领导者”。他的工作包括：决定测试的目标和测试进度；选择测试用例及相应的测试规则；对测试完成后的集成和系统测试进行评估。测试设计人员并不需要直接参与实际测试工作。

■ 构件工程师

构件工程师负责测试软件，以便自动执行一些测试规程。

■ 集成测试人员

集成测试人员直接参与到系统的测试工作，他的主要工作是对实现工作流中产生的每一个构造进行集成测试。

■ 系统测试人员

系统测试人员也直接参与到系统的测试工作，他的主要工作是完成对作为一个完整迭代的结构的构造进行所需的系统测试工作。

### 3. 工作流

在测试工作流中，包括 6 种活动：制定测试计划、设计测试、实现测试、执行集成测试、执行系统测试和评估测试。

■ 制定测试计划

制定测试计划主要包括：描述测试策略；估计测试工作所需的人力以及系统资源等；制定测试工作的进度。

制定测试计划由测试工程师来负责，主要的输入和制品如图 8-31 所示。

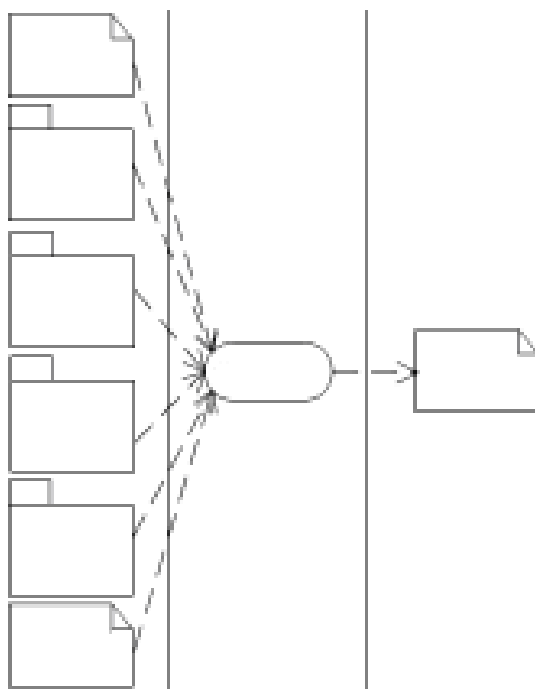


图 8-31 制定测试计划

### ■ 设计测试

测试设计主要包括：识别并描述每个构造的测试用例；识别并构造用于详细说明如何进行测试的测试规程。

设计测试由测试工程师来负责，主要的输入和制品如图 8-32 所示。

### ■ 实现测试

实现测试的目的是为了尽可能的建立测试构件以使测试规程自动化，由构件工程师负责，主要的输入和制品如图 8-33 所示。

### ■ 执行集成测试

执行在迭代内创建的每个构造所需要的集成测试，并捕获其测试结果，由集成测试人员负责，主要的输入和制品如图 8-34 所示。

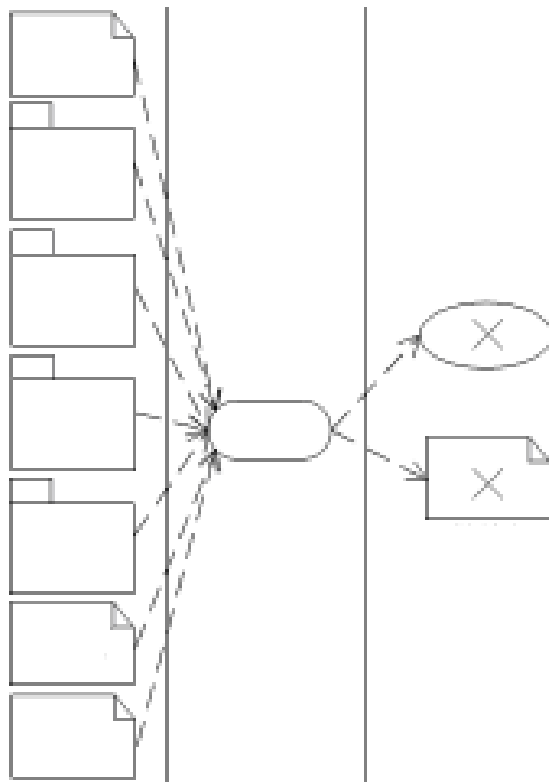


图 8-32 设计测试

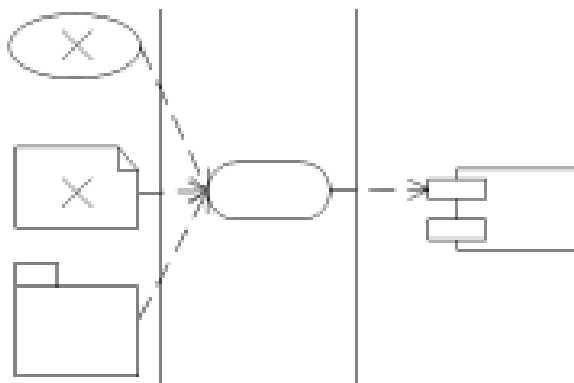


图 8-33 实现测试

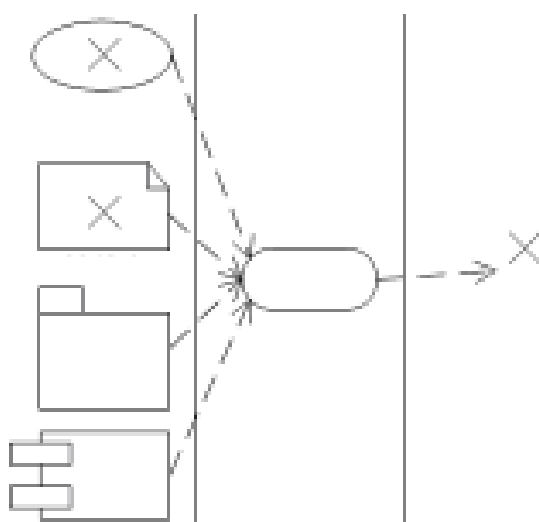


图 8-34 执行集成测试

#### ■ 执行系统测试

系统测试的目的是为了执行在每一次迭代中需要的系统测试,并且捕获其测试结果,由系统测试人员负责,主要的输入和制品如图 8-35 所示。

#### ■ 评估测试

评估测试的目的是为了对一次迭代内的测试工作做出评估,由测试工程师来负责,主要的输入和制品如图 8-36 所示。



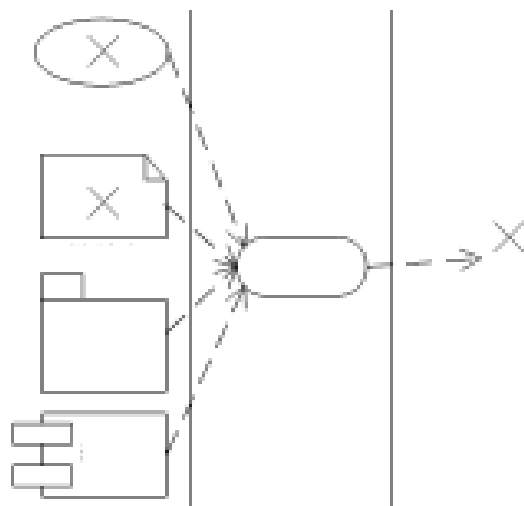


图 8-35 执行系统测试

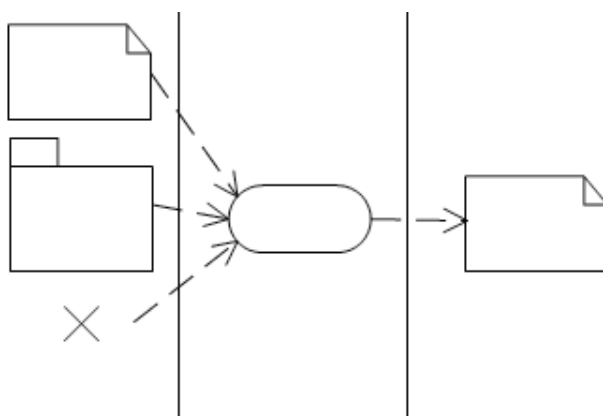


图 8-36 评估测试

## 8.4 RUP 统一过程案例

本节描述了怎样使用 Rational 公司提出的 RUP( Rational Unified Process ,Rational 统一过程 )理论来开发 Web 应用。本节特别关注于软件生命周期的前端部分 ,以及如何集成使用 RUP 理论的创意设计过程和软件工程过程。

### 8.4.1 简介

近几年,出现了一种新一代的软件模式,即经常谈到的 E-Commerce、E-Business 等。这些 Web 应用软件使客户和商业公司具备了在线商务处理、供应链集成、动态满足客户要求以

及个性化服务等能力。这些系统具有鲁棒的、可伸缩的和可扩展的架构，可以提升性能并且能处理大量的不可预知的商务交易负载。这种架构的特点是多层的、封装了商务逻辑、集成了多种遗留下来的异构系统。本节主要讨论如何建立这种类型的 Web 应用软件，并称这种类型的软件系统为 Web 解决方案（Web Solution）。

开发 Web 解决方案与开发其他应用软件有许多相似点，但也有许多不同之处。对于 Web 开发人员来说，挑战不仅仅在于要掌握新的软件技术，还必须使用一种能与一类新型用户达成一致意见的开发过程。

这里关注创意设计过程的重要性的同时，也希望能指出其他过程的重要性，例如架构、配置管理和测试。例如，对架构关注不够或缺少早期的架构运行测试，将可能会导致软件架构很脆弱，并且不能适应所需承受的处理负载。在本文末尾主要讨论了一些较好的实践方法，这些方法对于成功地实施基于 Web 的项目是必须的。

在新的由 Web 应用软件构建的市场环境中，客户并不只是购买软件，然后安装在计算机上，还要判断出 Web 界面是否吸引人。要构建一个成功的 Web 解决方案，关键之一就是要使 Web 应用程序有引人注目的外观。所以在软件开发的各个过程中，都有必要集成创意设计过程。

用例提供了项目相关人员如用户、经理、艺术指导、构架师和程序员等相互进行联系的通用语言，任何人都可以使用这种语言与项目取得沟通，并描述 Web 应用将做些什么。它允许人们用商业解决方案方面的语言进行交谈，每个人都根据用例确定的规则来表达。

#### 8.4.2 要求

构建 Web 解决方案与开发其他应用软件相比，要涉及到更多的人员。这些人员通常包括商业执行官、市场人员、创意设计人员、客户支持人员和技术研发团队等。拥有一种能够促进这些人员相互交流的过程是成功的关键。

成功的 Web 解决方案起始于引人注目的视觉外观，并且要确保一开始就和项目的目标保持一致。视觉表现应符合以下目标：必须与要解决的问题相适应；定义系统的边界；描述系统最重要的特征。

一旦视觉外观取得一致意见，将要召开交流促进会，以进一步定义系统的用户，以及系统应该为这些用户提供的服务。这些用户和服务将使用统一建模语言（UML）进行记录归档。在 UML 语言中，用户被称为活动者，服务被称为用例。这种将需求文档化的方法可以使用户说清楚他们到底需要什么服务，也能使开发人员完整而精确地验证用户需求。用例还能为开发过程提供一个前后一致的思路。

不要做重复的事情，应从过去的项目中挖掘出可重用的用例来使用。在开发系统用例的同时，还应在增补规格书，加入非功能性需求的描述，一些通用的术语也应该加入到词汇表中。增补规格书包含了一些对通常需求（对整个系统而言）的描述，如可用性、稳定性、性能、安全性、Web 集群和可支持性等。词汇表保证了项目成员对重要的概念保持统一的理解。

### 8.4.3 创意设计大纲

开发 Web 解决方案需要更加重视用户接口的创意设计。在定义了活动者和用例的同时，就已经得到了初步的用户接口方案。更进一步的用户接口方案在 Web 开发团体中通常被称做创意设计大纲。创意设计大纲定义了如下内容：

- (1) 网站的总体风格（例如这个网站是权威性的，或是轻松幽默的，或是服务性的）；
- (2) 用户将以什么样的方式来访问这个网站（例如他们的连接速度如何）；
- (3) 用户将使用什么浏览器；
- (4) 网站是否使用了框架结构；
- (5) 网站在色彩的使用上是否有限制；
- (6) 如果有用，需要定义一个色彩标准指南（包含网站的 Logo 和总体色调的标准）；
- (7) 需要什么样的动态效果点缀（例如鼠标翻滚效果、动画、滚动新闻和多媒体等）。

### 8.4.4 导航图

导航图是 Web 解决方案中的一种用层次树方式显示的视图，能够描述网站的用户如何访问该网站。这种层次图通常被称作网站地图，但是选择“导航图”这种称法，因为“网站地图”这个短语有多个含义。导航图的每一层显示出到达该层对应的屏幕（页面）需要点击多少次。通常，用户总是希望只需在起始页（通常被称作主页）上点击一次，即可到达目标 Web 页面。

在项目的早期确定网站导航图，可以提供一种有效的沟通媒介，便于项目的设计人员和项目开发团队之间交流。用户也可以更好地想象浏览该 Web 应用的情况，创意设计人员可以更好地了解网站导航模式。用例模型用于描述系统向最终用户提供怎样的服务，因此导航图是自然地从小模型进化而来的。

导航图是“用例故事板”技术的变种，“用例故事板”在 Rational 统一过程的“用户接口建模”行为中进行了定义。要开发导航图，首先应为每一个用例定义一个主窗口或主页面。在这个阶段，还不能确切地知道每一页看起来会是什么样子，甚至不能确切知道到底会有哪些具体的页面。所以需要关注于定义“逻辑页面”。这些逻辑页面是可选的，将随着用户接口的发展完善被采纳或被抛弃。逻辑页面用 UML 的构件——边界类来分析描述。

随后在设计和实现阶段，将用 HTML 页面或其他可视元素来替代 UML 的描述。一旦逻辑页面被定义了，导航图着眼于描述用户如何从一个逻辑页面浏览到另一个，同时描述了逻辑页面提供的主要特性。对大型系统而言，人们通常会为每一个活动者定义一个导航图。为了更深入地挖掘细节，每一个用例也需要定义同类的视图，以便定义那些用户在执行用例时将会浏览到的更多的页面（边界类）。当这些页面（边界类）被定义时，同时也描述了它们所处理的信息内容。

### 8.4.5 创意设计比选方案

创意设计比选方案（Creative Design Comps）向用户提供了 Web 解决方案的一些可选的视觉外观设计方案，以便推进创意设计过程，最终得到一个确实吸引人的视觉设计。

“比选方案”包含了一些网站的视觉外观模型。这些模型通常是一些有代表性的“平面”图形，由许多描绘浏览窗口外观的浏览视窗图组成。

Rational 统一过程提供了一个活动，即用户接口原型建模，它提供了一个收集用户接口反馈意见的通用方法。在 Web 开发方面，把这个活动稍微扩充了一下，提出了创意设计比选方案的概念。要创建比选方案，应选择一个最重要的用例，然后开发出许多可选的视觉外观设计方案（例如至少 10 个以上）。从这个方案集中选出 3 个最有希望的设计提交给客户。通常在项目的最终的 Web 设计方案达成一致意见以前会有 3 次迭代。这是一个创意设计和迭代的过程。

一旦这个过程达成一致意见并终止后，创意设计比选方案将进一步开发成具有实际功能的用户接口原型。非功能性的比选方案仅仅是第一代原型，用于征求那些关注网站视觉外观特性的用户的意见。

#### 8.4.6 Web 设计元素

Web 设计元素 (Web Design Elements) 指的是那些被组合到一个网站各个页面中的零散的图形图像。保证网站上用户接口的一致性对于保证网站的可用性来说是必须的。网站应该给用户前后一致的浏览体验。为了做到这点，项目开发过程中必须在网站中统一使用一整套标准的图形组件。这些图形组件应该在项目起始之初就设计好，还应设计如何使用这些图形组件的指南，以便项目组的全体成员都明白何时以及如何使用这些组件。

例如，Web 设计元素可以包含像导航图标和页面背景等图形元素。在整个网站中重复使用高质量的标准图形元素可以保证网站的一致性，并缩短网站投用前所需时间，减少开发费用，同时通过使用一套更高质量的图形也可以提高网站的开发质量。

Rational 统一过程在“ workflow 细节：分析行为”中区分了用例和组件。随后在“ workflow 细节：设计组件”中描述了这些组件的精确细节、实现和单元测试过程。

Web 设计元素是与最初的 Web 用户接口原型一起创建的。创意设计比选方案中可以挑选加工出用于 Web 用户接口原型的 Web 设计元素，并可以从中确定最终的用户接口原型，同时 Web 设计元素也被确定下来。

#### 8.4.7 初始 Web 用户接口原型

创意设计比选方案最终发展成用户接口原型。用户接口原型的外观基于最终确定的创意设计比选方案，并在 Rational 统一过程的“活动：原型化用户接口”阶段被创建，设计用户接口原型时使用了上面定义的 Web 设计元素。初始 Web 用户接口原型通常只支持了一小部分系统功能，这个原型是基于最重要和最典型的用例来创建的。

初始 Web 用户接口原型的开发能促进用户和设计者的交流，更好地沟通对网站的外观和给人的感觉的看法，同时网站相关的功能也被开发出来。在投入主要资金用于开发用户接口和网站功能之前，尽早获得用户对网站的看法是很有必要的。

### 8.4.8 UI 指南

Web 用户接口原型完成后，就可以编写用户接口设计的详细指南。该设计风格指南将指定何时以及如何使用 Web 设计元素、色彩配置、字体、层叠样式表，以及确定导航单元将怎样起作用 and 放置在哪里。UI 指南在“活动：开发用户接口指南”中被定义。

### 8.4.9 Web 用户接口总体原型

Web 用户接口总体原型（Full Web UI Prototype）对初始用户接口原型进行了扩展，使其包含了所有的用例。现在原型将能演示所有可浏览的页面和网站上的所有可视元素。根据系统的后端功能开发情况，页面中使用了真实或虚拟的数据。

尽管人们希望在项目的每一个建设迭代过程中，作为 Web 用户接口总体原型组成部分的页面都能够被精心开发，但 Web 用户接口总体原型开发的目的主要在于就用户接口的范围和详细性质达成一致。

### 8.4.10 总体导航图

在完成 Web 用户接口总体原型的开发后，应该创建总体导航图。该导航图基于最初的网站导航图和 Web 解决方案中所有已定义的用例。该导航图应该包含 Web 用户接口原型中所有已定义的页面（屏幕）。

许多曾在互联网淘金热中建立的公司陷入了一个误区，他们认为在“Web 时代”搞开发不需要实行健全的软件工程规范。公司要建立新颖的 Web 解决方案需要关注用户接口是否吸引人，但同时也要建立健壮的系统架构。Web 用户没有耐心浏览设计糟糕的网站，瞬间就可能失去一个潜在用户。

下面讨论 Rational 统一过程所描述的 6 个最具实践意义的软件开发理论，同时也讨论怎样在 Web 解决方案的开发中使用的这些理论。

#### 1. 迭代开发

迭代开发基于不断地发现、发明和实现，它强调在开发周期的早期辨明项目的风险，以便能够一致地、及时地、富有效率地管理和克服这些风险。这种可控的迭代开发过程可以使 Web 应用的发布周期更短、更快。

#### 2. 管理需求

需求管理是一种系统的方法，用于提取、组织、沟通和管理软件敏感系统或应用的不断变化需求。由于 Web 应用的需求经常随市场而变更，因此跟踪市场的发展并在项目开发周期中跟踪这些变化是非常有必要的。

#### 3. 利用组件架构

架构从组件、组件集成方式和组件间交互作用的机制和模式等方面描述了应用软件的结构。

鉴于 Web 应用软件必须是开放的和可扩展的，并且在当前基础上可以变更，因此使用组件架构是至关重要的。组件架构易于扩充，并能适应正在进行的变化，可以最大限度地重用以

前开发的组件和第三方开发的组件。

#### 4. 可视化建模

模型可以帮助理解和弄清楚问题以及解决方案，模型是对实体的简化，用于理解复杂的系统。Web 架构是多层的和分布式的，在设计、开发和配置时比较复杂。管理这些复杂的需求需要建立慎重考虑的和清晰的架构和设计。可视化建模符号如 UML 提供了表示系统架构和设计的机制。

#### 5. 检验质量

质量关注过程和产品。无论是中间过程还是最终产品，都应有高质量。Web 产品主要用于公用场合，失败的代价会很高。在性能和稳定性不好的情况下，公众就会不再注意这个 Web 站点。Web 应用的发布周期确定后，尽早地、经常地和自动化地对其进行测试是很重要的。

#### 6. 控制变更

Web 应用包含许多对象和组件，它们由许多人创建和编辑，经常并行被开发出来。在这个持续开发的工作和环境中，控制变更是必须的。多个版本和配置的并发管理需要在整个开发周期内进行严格的配置和变更管理。

## 第9章 图书馆管理系统

当今社会已经进入信息化时代,图书馆中传统的手工管理方式已经远远不能满足与日俱增的书籍管理需要,为了提高工作效率,降低运行成本,方便读者使用,图书管理系统必须引入信息化管理方式。本章将详细分析一个图书馆管理系统的实现过程。从需求分析,架构设计,系统建模,详细设计,代码实现逐步展开分析,整个过程按照软件设计实际流程进行。

### 9.1 需求分析

需求分析是软件设计的第一步,是整个软件成功实现的基础,只有真正做好需求分析,才能真正了解客户的需要,以指导好下一步的工作,整个软件的实施是建立在需求所分析出的各项功能上的。接下来就针对图书管理系统的总体需求做一个分析,可能一些图书馆管理系统有特殊的需求,读者在实际开发过程中,应根据实际情况在此系统的基础上做进一步的工作。

#### 9.1.1 系统总体功能需求

首先需要对图书管理系统进行详细地了解和分析,一个功能完全满足基本需要的系统,必须包括以下的几个模块。

##### (1) 基本数据维护功能模块

在这个功能模块中,提供了使用者录入、修改并进行维护基本数据的途径。比如在这个模块中可以输入读者的信息、书籍的各项信息,也可以对这些信息做修改、更新。

##### (2) 基本业务功能模块

这个功能模块主要实现对读者利用图书馆借还书籍的管理,比如根据读者的借用书籍要求更新图书馆书籍数据库系统,如果书籍已经借出,可以进行预留操作。用户每次还书也要进行数据库记录的各项更新。这个模块是整个图书管理系统的关键部分。

##### (3) 数据库管理功能模块

在图书管理部门,对所有的图书信息以及会员信息都要进行统一管理,书籍的借出等情况也都要进行详细的登记,以便能对整个图书管理部门的运作有全面的了解,并根据实际情况补充书源不足的部分。

##### (4) 信息查询功能模块

在对图书管理信息系统进行全面信息化管理的过程中,查询是一项非常重要的功能。比如用户借书信息,借还日期的信息,书籍库源情况信息,预留信息等,这项功能能使用户得到即时书籍信息,方便用户根据实际情况选择业务方式。

### (5) 安全/使用管理功能模块

这是任何一个信息管理系统都需要的部分,图书管理系统的操作也只能由专人进行,只有图书管理部门的工作人员才能拥有权限,特别是图书的借出状况,如果没有安全管理部分,后果难以想象,可能每次登录都需要用户身份的验证。

### (6) 帮助功能模块

这也是软件不可缺少的部分,为了方便使用者使用软件,应该有一个详细的帮助模块。图书管理系统的功能需求图如图 9-1 所示。

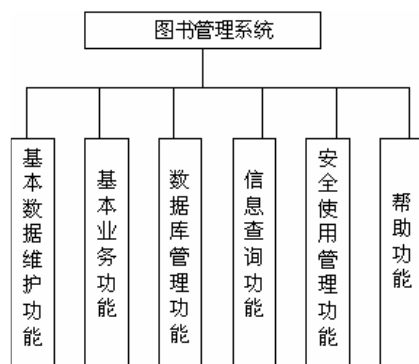


图 9-1 系统总体功能需求框图

## 9.1.2 基本数据维护功能需求

基本数据维护模块的具体功能如图 9-2 所示。基本维护信息包括 :书籍信息和用户信息等。

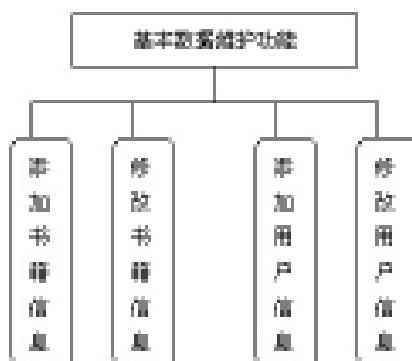


图 9-2 基本数据维护模块功能需求框图

(1) 书籍信息的增加,如图 9-3 所示。

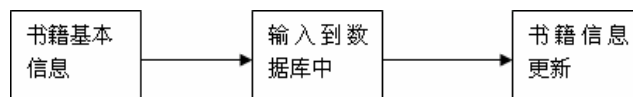


图 9-3 增加书籍信息



这个功能实现把书籍信息输入到数据库中。书籍信息包括名称、作者、ISBN、类别、预留信息和书项。

(2) 书籍信息的修改,如图 9-4 所示。这个功能实现把书籍信息修改后输入到数据库中。

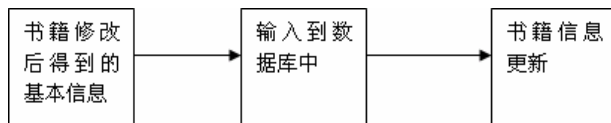


图 9-4 修改书籍信息

前置条件: 已存在书籍。

(3) 用户信息的增加,如图 9-5 所示。这个功能实现把读者信息输入到数据库中。读者信息包括姓名、地址、城市、区号、借书条目和预留条目信息。

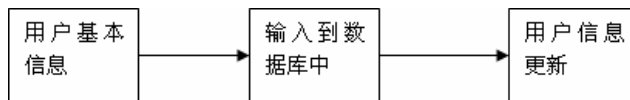


图 9-5 增加用户信息

(4) 用户信息的修改,如图 9-6 所示。这个功能实现把读者信息修改后输入到数据库中。

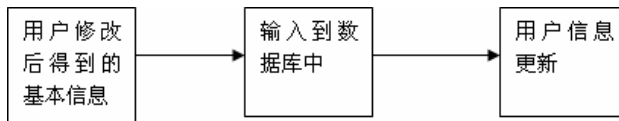


图 9-6 修改用户信息

前置条件: 已存在该读者。

### 9.1.3 基本业务功能需求

基本业务功能需求模块的具体功能如图 9-7 所示。这些功能需求基本上包括了读者要求提供的业务。

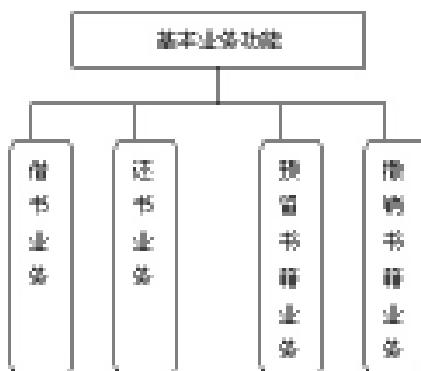


图 9-7 基本业务功能需求框图

(1) 借书功能需求,如图 9-8 所示。该功能实现的是将读者要求借出的书籍信息加入到数据库中,然后更新数据库,借出的时间段内,该书不能再借出。借出信息包括书目和借出者等。

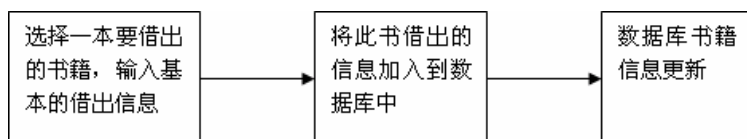


图 9-8 借书功能

(2) 还书功能需求,如图 9-9 所示。这个功能实现将读者要还的书籍的信息以及读者的姓名,重新将书籍状态设置为可用。同时更新数据库。输入信息包括书名、作者、借出的书目和借书读者的姓名。

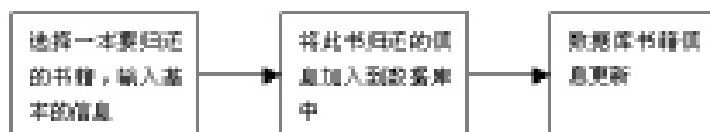


图 9-9 还书功能

(3) 预留书籍功能需求,如图 9-10 所示。这个功能实现将读者的预留书籍要求输入到书籍信息中,便于借出书籍归还时能够及时把书借给预留读者。预留信息包括书名、作者和要求预留书籍的读者姓名。

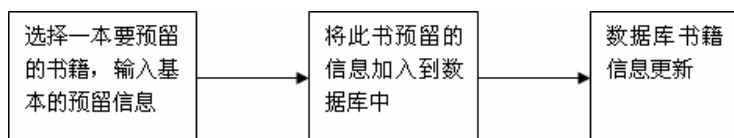


图 9-10 预留书籍功能

前置条件：相应书籍已经全部借出。

(4) 撤销预留功能需求,如图 9-11 所示。该功能实现根据读者的要求撤销预留功能,改变当前书籍的预留状态,更新数据库,以便其他读者能进行书籍的预留或者借出。

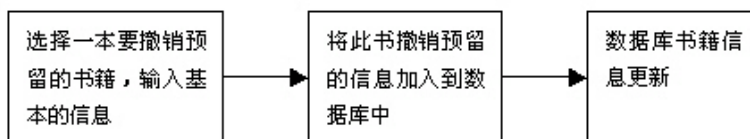


图 9-11 撤销预留功能

前置条件：预留书籍状态已经开启。

### 9.1.4 数据库维护功能

数据库维护功能是信息管理系统不可或缺的部分，其需求如图 9-12 所示。

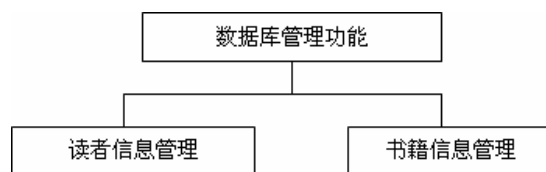


图 9-12 数据库维护功能需求

(1) 读者信息管理功能需求，如图 9-13 所示。该功能实现对读者信息的统一管理，可以增加和修改读者信息。

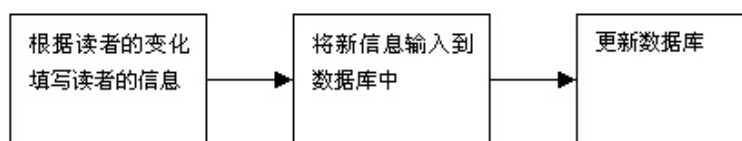


图 9-13 读者信息管理功能

(2) 书籍信息管理功能需求，如图 9-14 所示。该功能实现对书籍信息的统一管理，可以增加或者修改书籍的信息。

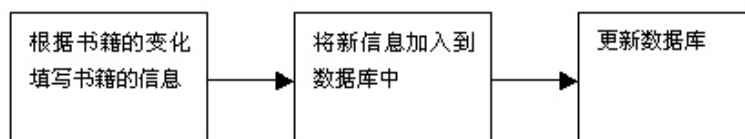


图 9-14 书籍信息管理功能

### 9.1.5 查询功能需求

这个模块的具体功能如图 9-15 所示。



图 9-15 查询功能需求框图

(1) 读者信息查询需求, 如图 9-16 所示。该功能实现查询读者的现有信息, 除了读者的个人信息外, 还有读者的预留书籍, 以及已经借出的书籍信息。

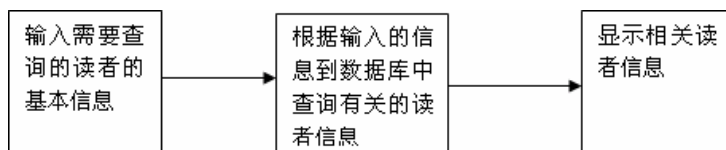


图 9-16 账单查询功能

(2) 书籍信息查询需求, 如图 9-17 所示。

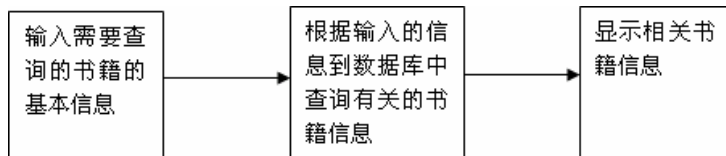


图 9-17 书籍查询功能

### 9.1.6 安全使用管理功能需求

这个模块的具体功能如图 9-18 所示。安全的管理功能包括用户名以及密码验证的管理。

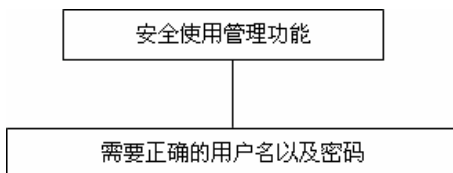


图 9-18 安全使用管理功能需求框图

### 9.1.7 帮助功能需求

这个模块的具体功能如图 9-19 所示。帮助功能包括软件总体说明和软件使用说明。



图 9-19 帮助功能需求框图

## 9.2 UML 系统建模

使用 UML 进行系统建模，就是使用面向对象的方法来分析系统，然后用可视化的模型将信息用标准的图形直观地显示出来，以此建立面向对象的系统模型。本例主要使用 UML 在分析、设计、实现过程中相应的视图来进行系统开发的分析，以帮助开发人员了解系统功能与系统流程。

### 9.2.1 用例的建立

分析阶段最重要的是用例视图的建立，用例视图强调用户希望得到的功能，它是成为参与者的外部用户所能观察到的系统功能的模型图。通过用户视图，使用者应该明确软件后续设计阶段所要完成的任务，整个软件直到实现的过程都是围绕需求阶段的用户例来进行的。

#### (1) 角色 (Actors) 的确定

角色是与系统有交互作用的人或事务，通常代表着一个系统的使用者，或者需要与系统打交道的人或事务。

在图书管理系统中有如下事务：

- 读者要借书籍
- 读者要还书籍
- 读者要预留书籍
- 读者要撤销预留书籍
- 工作人员根据读者要求提供服务
- 工作人员进行查询，修改信息

对于读者来说，所要求的服务都是直接传递给工作人员的，真正的系统的使用者是工作人员，读者只是跟工作人员打交道，不涉及到系统本身。而整个工作流程中，只出现两个角色，那就是读者和工作人员。

使用 Rational Rose 来建立角色，如图 9-20 所示。

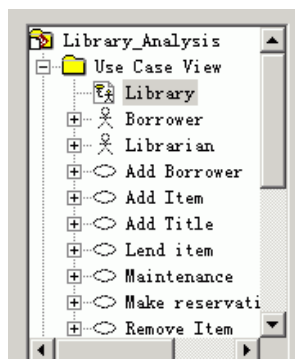


图 9-20 Rose 中角色的建立

## (2) 创建用例

用例是系统跟用户的交互，是系统提供的功能块。使用用例与传统的方法不同，将项目分解成使用用例是面向对象的过程而不是面向实现的过程，用例关注的是系统外的用户，有助于让开发人员了解最重要的部分——满足用户需求和期望。通过用例，用户也能清楚看到系统提供的功能。在本系统中，通过分析得到以下用例：

- 借出书籍
- 返还书籍
- 预留书籍
- 撤销预留书籍
- 增加书籍
- 修改更新书籍
- 增加书目
- 删除书目
- 增加读者
- 修改更新读者
- 查询书籍信息
- 查询读者信息
- 登录验证
- 得到帮助信息

图书管理系统中读者得到服务的用例图、工作人员维护读者及图书信息的用例图和工作人员登录及查询用例图分别如图 9-21、图 9-22 和图 9-23 所示。

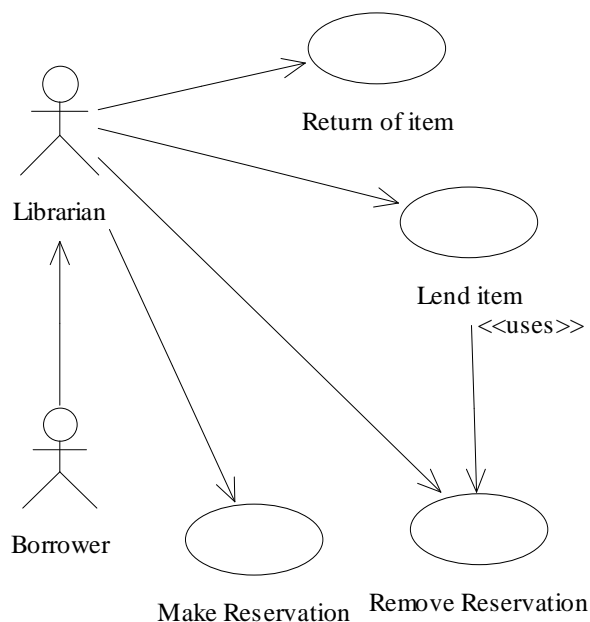


图 9-21 读者得到服务的用例图

注释：

Return of Item：还书用例。

Lend Item：借书用例。

Remove Reservation：删除预留书籍用例。

Make Reservation：预留书籍用例。

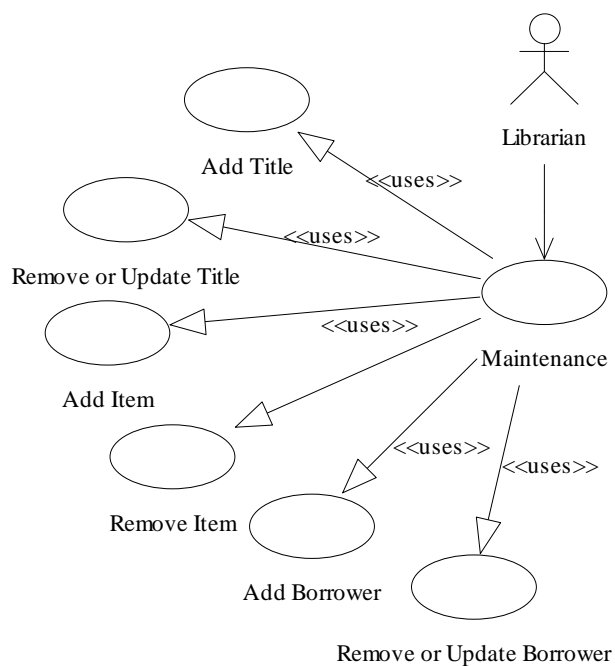


图 9-22 工作人员维护读者及图书信息的用例图

注释：

Add Title：增加书籍用例。

Remove or Update Title：删除或者更新书籍信息用例。

Add Item：增加数目信息用例。

Remove Item：删除数目信息用例。

Add Borrower：增加读者信息用例。

Remove or Update Borrower：删除或者更新读者信息用例。

Maintenance：维护用例。

Librarian（图书馆工作人员）：参与者。

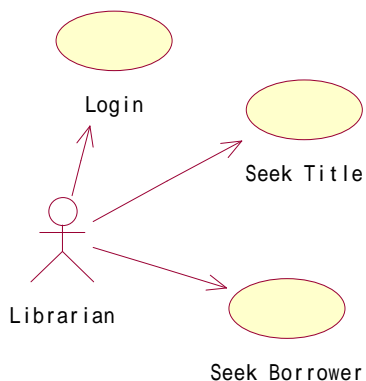


图 9-23 工作人员登录及查询的用例图

注释：

Login：登录用例。

Seek Title：查询书籍用例。

Seek Borrower：查询读者信息用例。

### 9.2.2 时序图与协作图的生成

表现系统流程以及系统元素之间的交互关系可以用两种视图：时序图 (Sequence Diagram) 与协作图 (Collaboration Diagram)。

时序图的功能是按时间顺序描述系统元素间的交互，协作图的功能按照时间和空间顺序描述系统元素间的交互和它们之间的关系。

(1) 工作人员使用系统的时序图如图 9-24 所示。

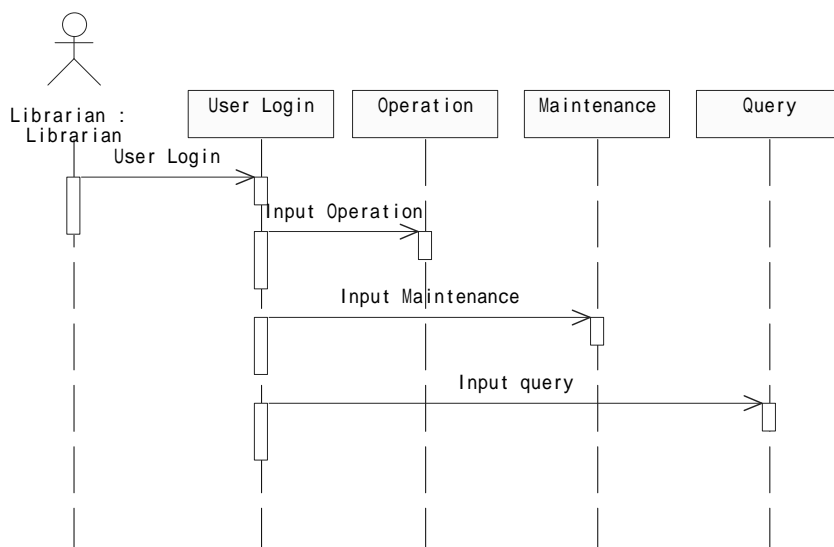


图 9-24 工作人员操作时序图



注释：

Librarian：管理人员，进行各种系统的操作。

User Login：用户登录，需要管理员输入登录必须的验证信息。

Operation：基本业务，管理员输入必要的业务处理要求。

Maintenance：维护，管理员进行信息的维护。

Query：查询，管理员进行信息的查询。

(2) 工作人员使用系统的协作图如图 9-25 所示。

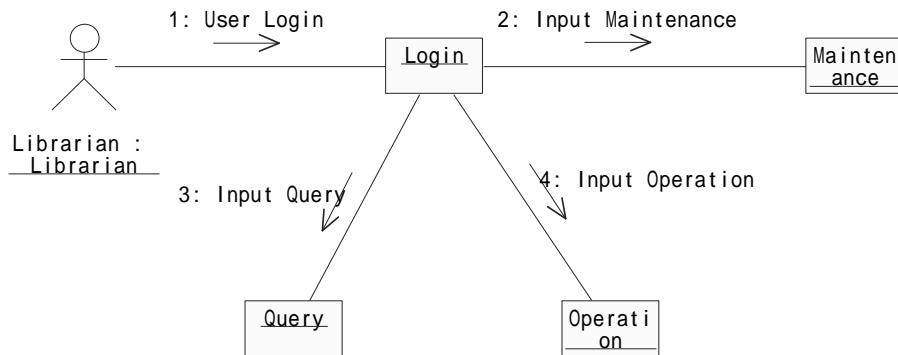


图 9-25 工作人员操作协作图

### 9.2.3 状态图的生成

工作人员使用系统的状态图如图 9-26 所示。

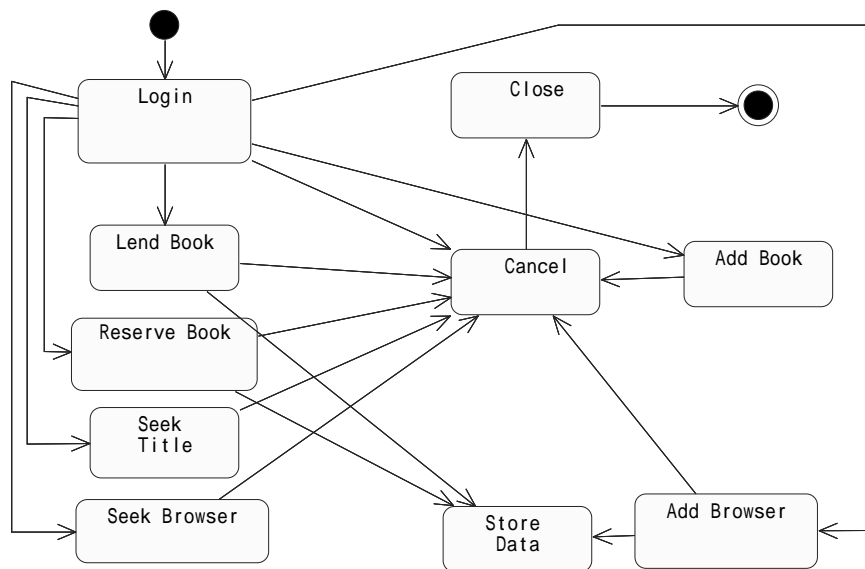


图 9-26 工作人员操作状态图

注释：

Login：登录状态。

Lend Book：借阅书籍状态。

Reserve Book：预留书籍状态。

Seek Title：查询书籍信息状态。

Seek Borrower：查找读者信息状态。

Store Data：储存数据状态，进行完系统的数据操作，确认以存储保留信息。

Add Borrower：增加读者状态。

Add Book：增加书籍状态。

#### 9.2.4 活动图的生成

工作人员使用系统的活动图如图 9-27 所示。

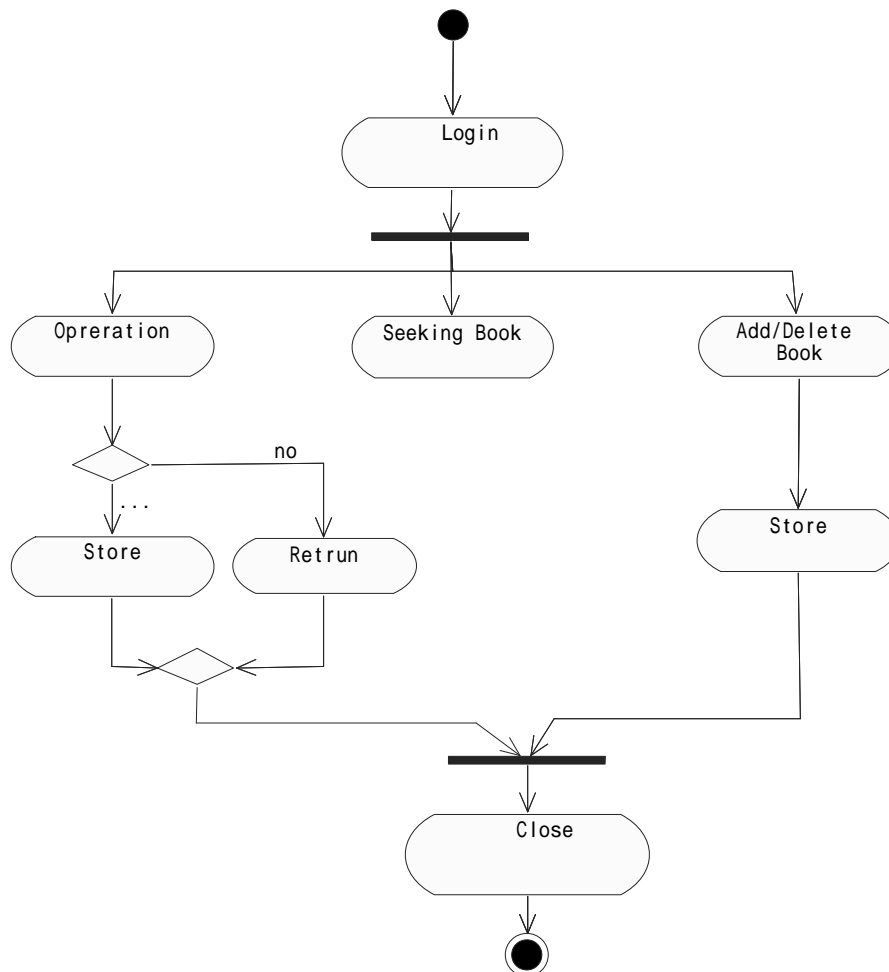


图 9-27 工作人员操作活动图

注释：

Login：登录状态活动。

Operation：基本业务操作活动。

Seeking Book：查询书籍活动。

Add/Delete Book：增加/删除书籍活动。

Store：存储信息活动。

Return：取消操作活动，回到操作前状态的。

为了图的清楚，简略了类图的一部分内容。

## 9.3 类与接口

类图是系统设计核心的部分，其中 Persistent 表示数据库部分，所有与存储信息有关的类都以此为父类，明确基本的类以及相互的关系有助于使用者进一步工作，这也是进入编码阶段重要的基础。

### 9.3.1 类图的生成

图书馆管理系统的商业对象类图、数据库类图和效用类图分别如图 9-28 ~ 图 9-30 所示。

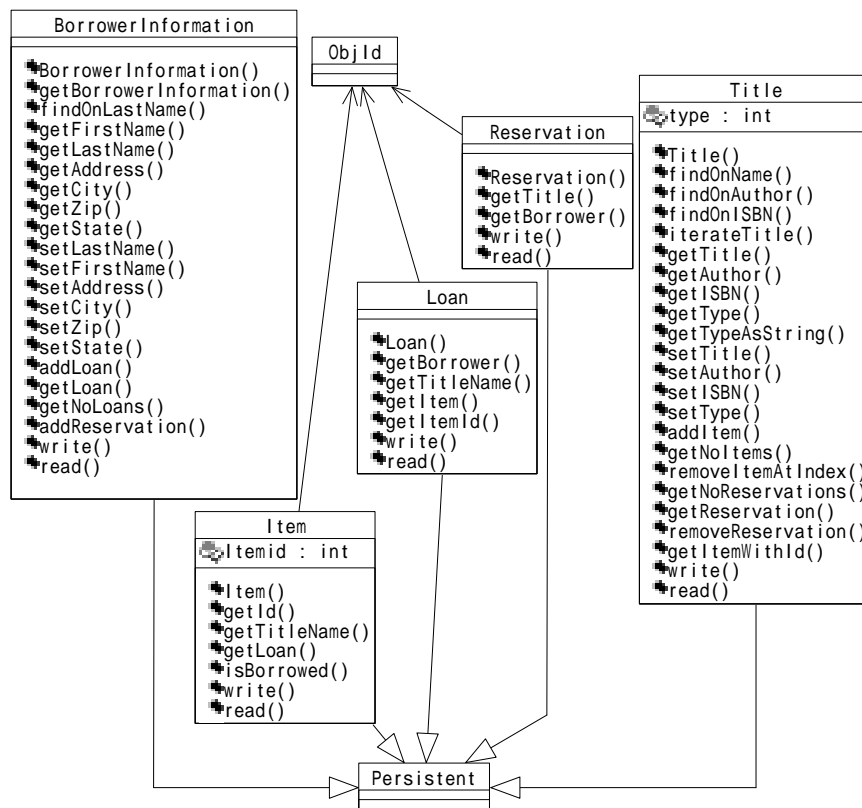


图 9-28 商业对象类图

商业对象类图：主要包括读者信息、书籍信息和借阅信息等几个类。商业对象类图的结构如图 9-28 所示，这里含有整个系统类的基础信息部分。

商业对象类图中各个类的说明见表 9-1。

表 9-1 商业对象类图说明

类 名	中文类名	说 明
BorrowerInformation	读者信息类	包括读者个人信息和借阅信息
Title	书籍类	包括书籍信息和借阅信息
Item	书项类	包括书项信息和借阅信息
Loan	借书类	包括借书关联信息
Objid	对象父类	所有信息类的基本父类
Reservation	预留类	包括预留书籍的关联信息
Persistent	固定类	包括数据库关联的共有父类信息

数据库类图：便于数据库的操作，定义的包含数据库基本信息的父类，包括存储、删除、更新和读写等内容。

效用类图：包括对象信息的一些共有内容的基础类。

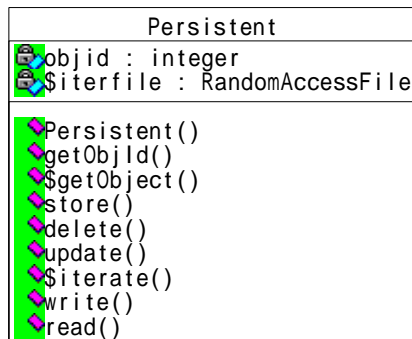


图 9-29 数据库类图

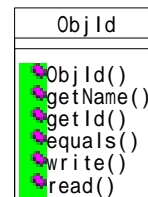


图 9-30 效用类图

下面介绍的是 UI(用户界面)的类图部分，因为相应类图比较复杂，没有完整列出。Base 类图、Function 类图、Information 类图和 Maintenance 类图分别如图 9-31 ~ 图 9-34 所示。

Base 类图：这部分类图是系统运行界面的基础部分，包括系统的启动类 StartClass、主窗口类，以及一些帮助和退出对话框。

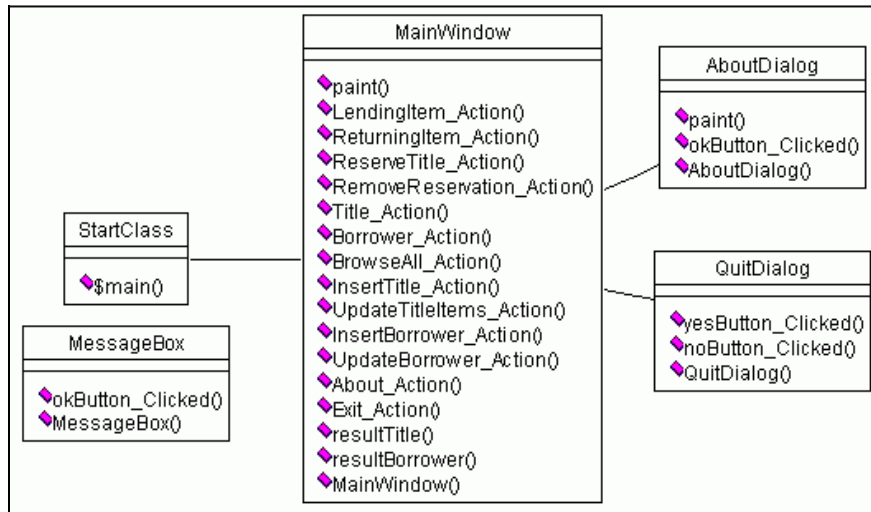


图 9-31 Base 类图

Base 类图中各个类的说明见表 9-2。

表 9-2 Base 类图说明

类 名	中文类名	说 明
MainWindow	主窗口类	系统主界面的主要信息
AboutDialog	帮助对话框类	系统帮助信息的对话界面
QuitDialog	退出对话框类	系统退出时与用户的交流界面
StartClass	启动类	系统启动类
MessageBox	消息框类	系统消息显示界面

Function 类图：这部分类图是系统运行界面的基本功能部分，包括借书、还书、预订书籍和取消预订的界面。

Function 类图中各个类的说明见表 9-3。

表 9-3 Function 类图说明

类 名	中文类名	说 明
MainWindow	主窗口类	系统主界面的主要信息
LendItemFrame	借书框架类	借书操作的界面框架
ReturnItemFrame	还书框架类	还书操作的界面框架

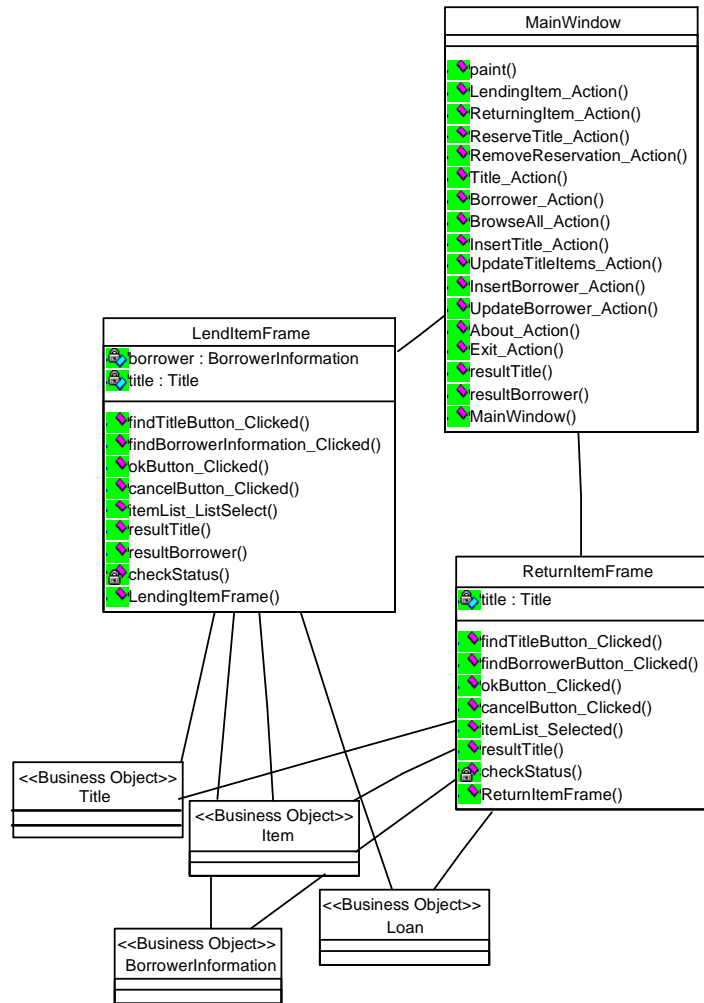


图 9-32 Function 类图

Information 类图：这部分类图是系统运行界面的信息显示部分，包括书籍信息，读者信息的显示，查询各种信息的对话框等界面。

Information 类图中各个类的说明见表 9-4。

表 9-4 Information 类图说明

类 名	中文类名	说 明
MainWindow	主窗口类	系统主界面的主要信息
TitleInfoWindow	书籍信息窗口类	查询书籍所有信息的显示窗口
FindTitleDialog	查询书籍对话框类	查询书籍时显示必要选择项的对话框
BorrowerInfoWindow	读者信息窗口类	查询读者所有信息的显示窗口
FindBorrowerDialog	查询读者对话框类	查询书籍时显示必要选择项的对话框

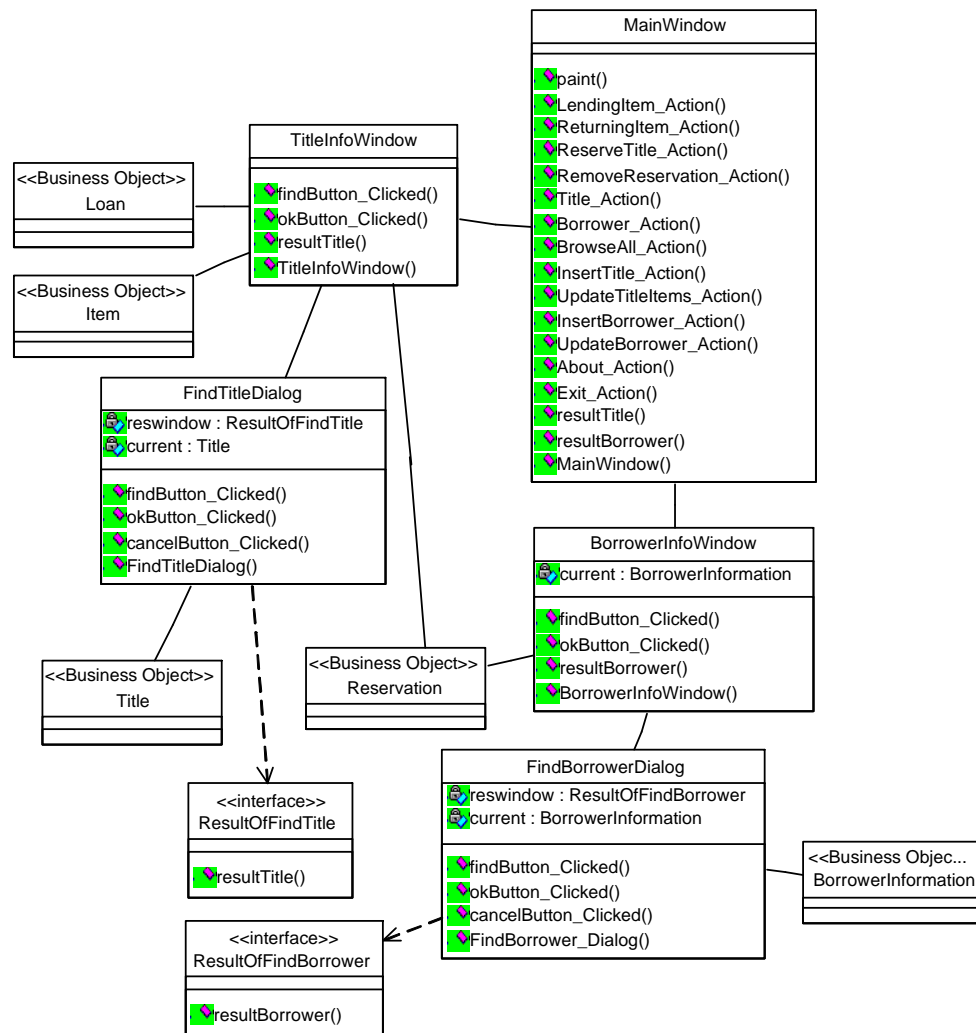


图 9-33 Information 类图

Maintenance 类图：这部分类图是系统运行界面的维护部分，包括读者信息，书籍信息的修改、更新和查询的内容。

Maintenance 类图中各个类的说明见表 9-5。

表 9-5

Maintenance 类图说明

类名	中文类名	说明
MainWindow	主窗口类	系统主界面的主要信息
TitleFrame	书籍框架类	维护书籍信息的框架
BorrowerFrame	读者框架类	维护读者信息的框架

续表

类名	中文类名	说明
FindBorrowerDialog	查询读者对话框类	查询需要维护的读者信息的对话框
FindTitleDialog	查询书籍对话框类	查询需要维护的书籍信息的对话框
UpdateBorrowerFrame	更新读者框架类	更新读者信息操作的基本框架
UpdateTitleFrame	更新书籍框架类	更新书籍信息操作的基本框架

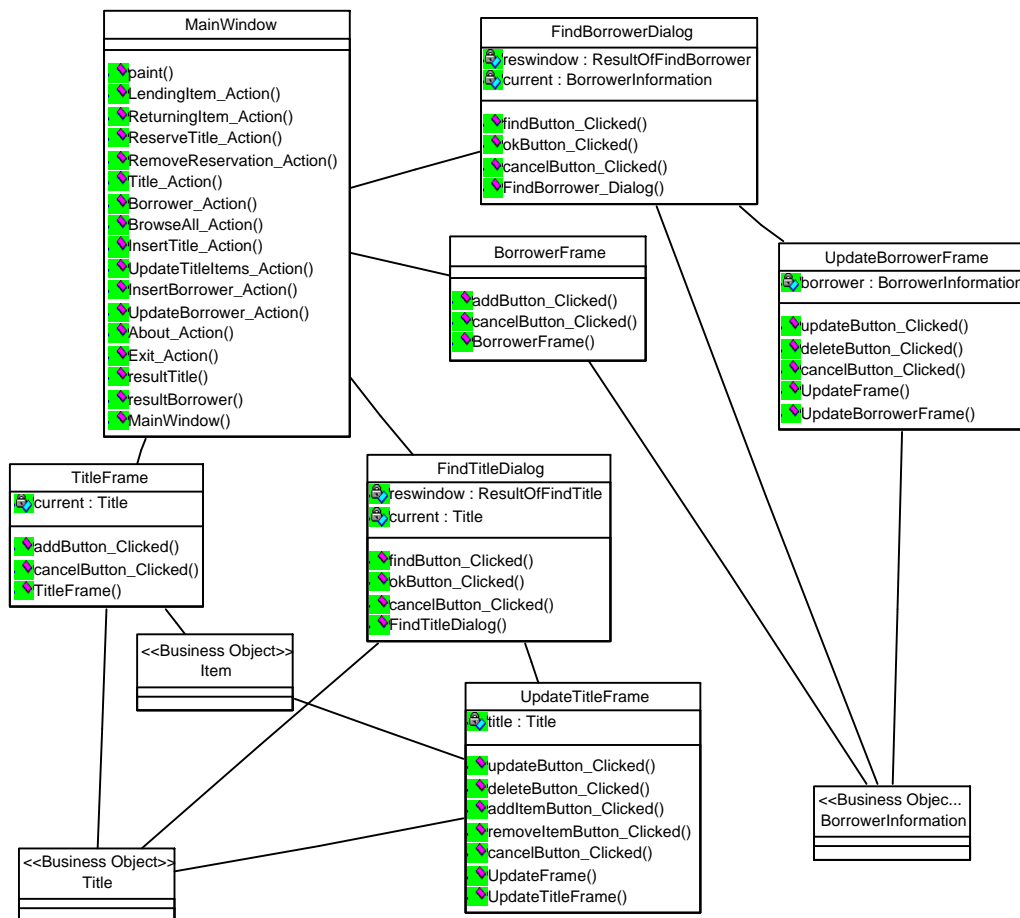


图 9-34 Maintenance 类图

### 9.3.2 包图的生成

包图主要显示类的包以及这些包之间的依赖关系。有时还显示包和包之间的继承关系和组成关系。系统包图显示了系统中不同包之间的相互连接关系。包括了 UI 包、商业对象(Business



Object) 包、效用 (Utility) 包和数据库 (Database) 包。图书管理系统的包图如图 9-35 所示。

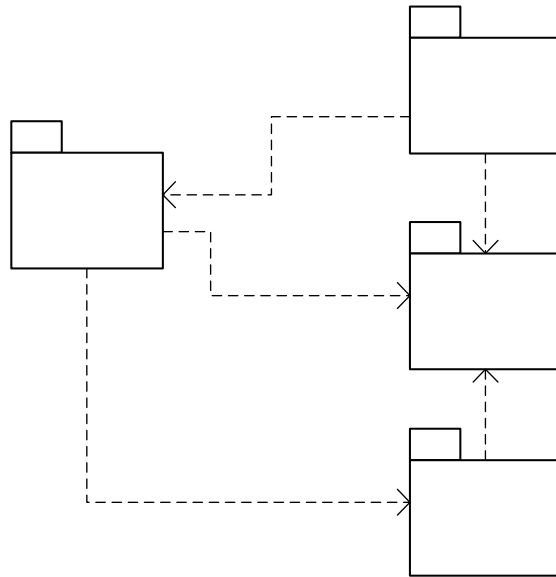


图 9-35 包图

### 9.3.3 组件图的生成

系统组件图：系统包含 4 个类包：UI 包、商业对象包、效用包和数据库包，以及一个启动程序组件 StartClass.java。图书管理系统的组件图如图 9-36 所示。

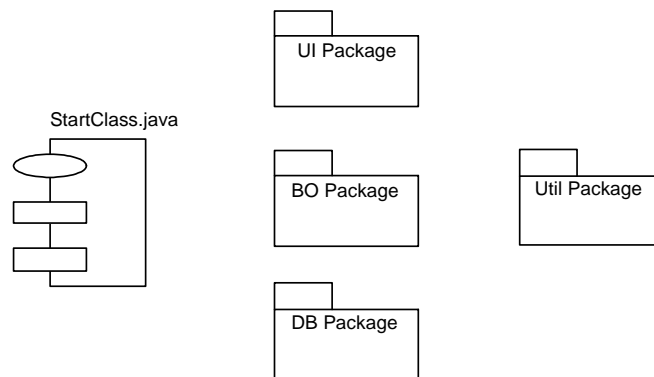


图 9-36 系统组件图

商业对象组件图：其中包含 5 个组件部分，如图 9-37 所示。

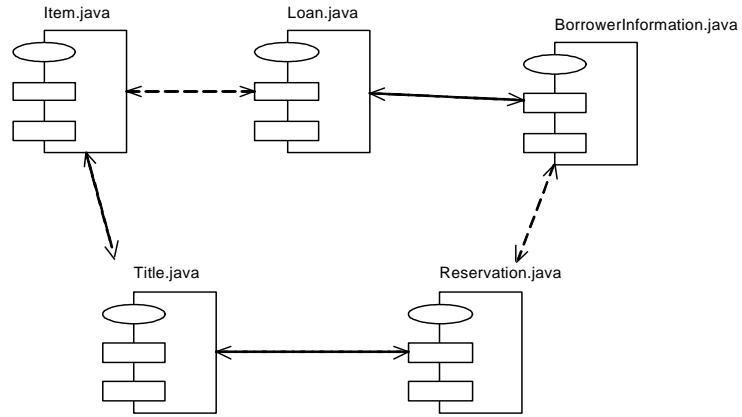


图 9-37 商业对象包的组件图

UI 包组件图：包含所有 UI 界面的组件。如图 9-38 所示。

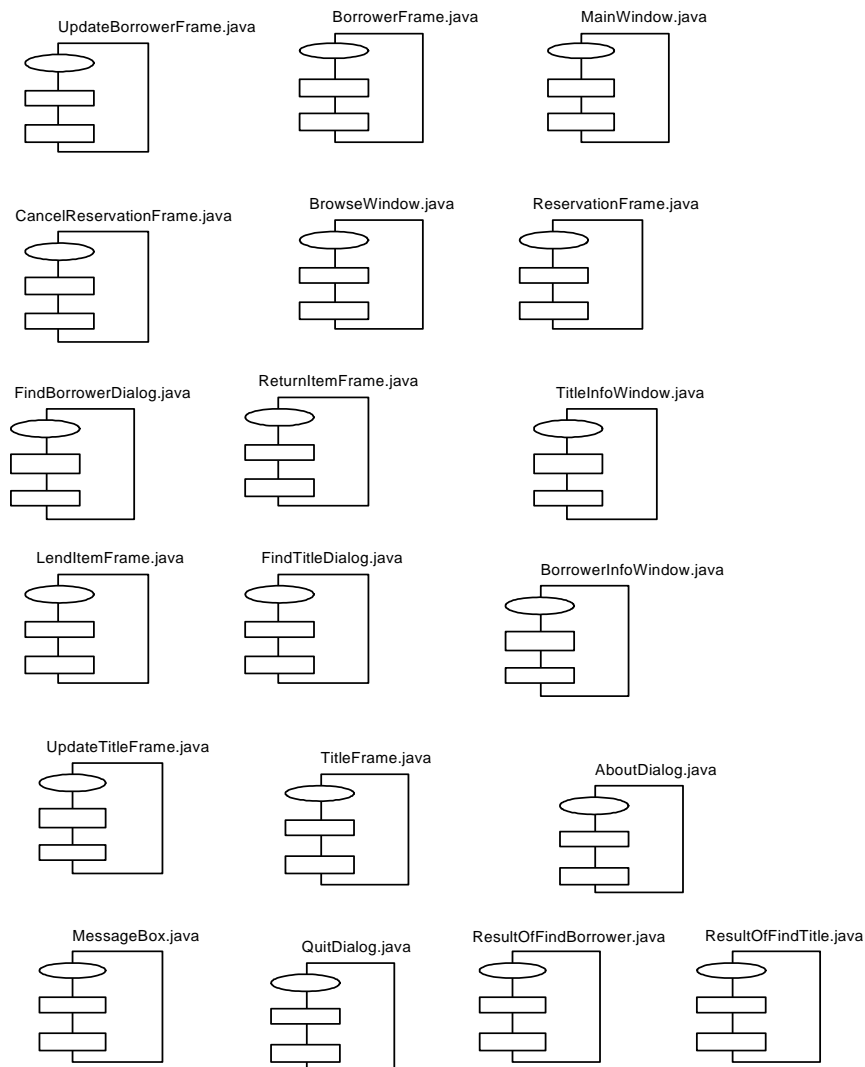


图 9-38 UI 包的组件图

## 9.4 系统部署

图书管理系统的展开图（配置图），如图 9-39 所示。系统由 5 个节点组成，应用服务器负责整个系统的运行总体协调工作，数据库服务器负责数据库的管理。业务管理、查询管理以及维护管理分管各个模块的内容。

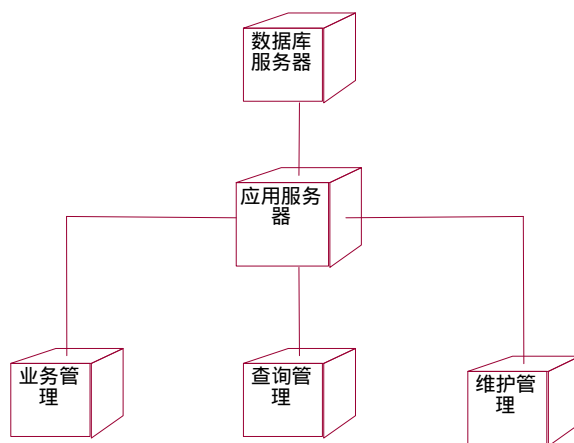


图 9-39 图书管理系统的展开图

# 第 10 章 ATM 自动取款机系统

## 10.1 系统概述

ATM ( Automatic Teller Machine ) 自动取款机，是由计算机控制的自动出纳系统。它主要服务于活期储蓄，是实现客户自我服务的先进电子化设备。在我国，基本上所有银行系统都有自己的 ATM 系统。庞大的系统上运行着无数的金钱交易，因此安全性要求极高。现在很多城市的不同银行间的 ATM 交易都通过一个银行联合会来统一标准，简称为银联。正是因为它，人们才可以跨行取款。整个 ATM 取款机的业务大致分为 4 块：查询余额、取款、存款和更改密码。本章主要学习如何对 ATM 取款机进行建模。

因为 ATM 技术的高安全性和高速度，所以在我国的发展十分迅速，尤其在银行领域。截至 1999 年，我国宽带网已初具规模，建立起了全国公用宽带业务网 ( CHINAFRN / ATM )，已建成完整的电子商务安全认证体系。公用宽带业务网骨干网连通全国 31 个省市，20 多个省已建成了省内宽带网，其中有 17 个省的多媒体宽带网已投入了业务使用，并提供 ATM 等高速业务。

由于 ATM 技术能适应高带宽应用的需求，能同时传送多种数据信息，具有良好的可扩展性，高强度的安全性等优点，因此在计算机网络领域有着巨大的应用。几乎所有的银行系统都是采用 ATM 技术，主要用在各省市之间交换数据和 ATM 终端。每天银行都有大量的报表数据要通过计算机处理然后汇总到数据库中，ATM 技术的产生为此过程的安全性提供了强大的支持。

在本书介绍的建模基础上，读者可以根据需要来扩展自己的系统，以实现具体的功能。因为 ATM 机具有的自动取款功能和无人值守的特点，使得利用 ATM 机犯罪的案件和纠纷不断增加。为了防止 ATM 取款机被抢劫，可以安装智能摄像头来通知 ATM 停止交易，从而避免损失。为此，增加了意外情况管理模块来扩充 ATM 系统的功能，使得整个系统更强大。

## 10.2 需求分析

在开始制定需求前，先介绍一种简单的确定需求的方法：CRC 模型法。

CRC( Class Responsibility Collaborator )卡建模是一种简单且有效的面向对象的分析技术。在一个 OO ( 面向对象 ) 开发项目中，包括用户、系统分析员和开发者在建模和设计过程中经常应用 CRC 卡建模，使整个开发团队普遍的理解形成一致。它由 3 部分组成类 ( Class ) 职责 ( Responsibility ) 和协作 ( Collaborator )，如图 10-1 所示。

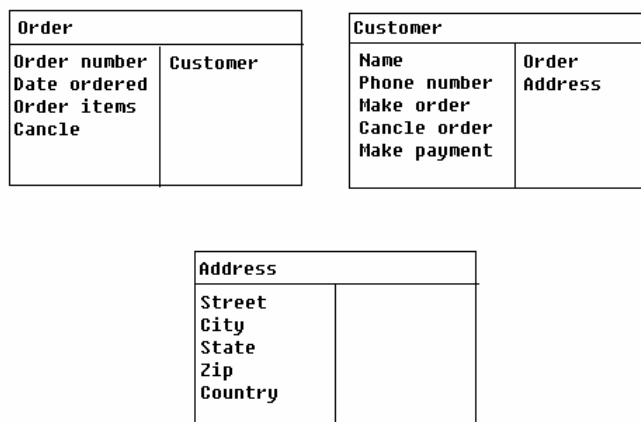


图 10-1 CRC 卡的样式

一个类代表许多类似的对象，而对象是系统模型化中关注的事物，可以是一个人、地方、事情、或任何对系统有重要性的概念。类名一般列在 CRC 卡的顶部。

职责是类需要知道或做的任何事物。这些职责是类自身所知的知识，或类在执行时所需的知识。

协作是指为获取消息，或协助执行活动的其他类。在特定情形下，与指定的类按一个设想共同完成一个（或许多）步骤。协作的类顺着 CRC 卡的右边排列。

### 10.2.1 系统总体功能需求

需求分析是成功实施一个管理系统的基础，只有弄清楚客户的需求，才能真正开发出满足客户需要的信息系统，也才能够真正让整个系统发挥其相应的作用。本节以比较高的层次来对整个 ATM 系统进行需求分析。

ATM 系统是一个复杂的软件控制硬件的系统，了解外部设备如何协调工作是整个建模的基础。这是以具体的业务为出发点对它进行建模，一个功能完全的 ATM 系统必须包括以下的几个模块。

（1）读卡机模块 在这个功能模块中，允许客户使用银行卡插入读卡机。读卡机来识别卡的种类并在显示器上提示输入密码。

（2）键盘输入模块 在这个功能模块中，客户可以输入密码和取款金额，并选择要进行的事务。通常在这个键盘上只设置数字键和选择键，目的是方便客户使用。只有在这个功能模块中，需要客户的交互。

（3）IC 认证模块 这个功能模块主要用于鉴别卡的真伪。基于 IC 卡的安全授权系统，完全可以从技术上严格保证卡的惟一性与防伪性，使基于数字形式的电子政务和电子商务安全运转，其目的就是网络安全。

（4）显示模块 在这个功能模块中，显示一切与客户有关的信息，包括一切交互时所需的提示，确认等信息。

（5）吐钱机模块 在这个功能模块中，吐钱机按照客户的需求，选择合适面值的钞票给客

户，是比较关键的一步。

(6) 打印报表模块 在这个功能模块中，是提供给客户一张取款凭据。客户可以选择打印与不打印。主要信息是卡号和金额等。

(7) 监视器模块 在这个功能模块中，为防止以外事件产生而设置摄像头，以保证户外交易的安全性。银行有权调查取款记录。

系统功能需求可用图 10-2 简要表示。

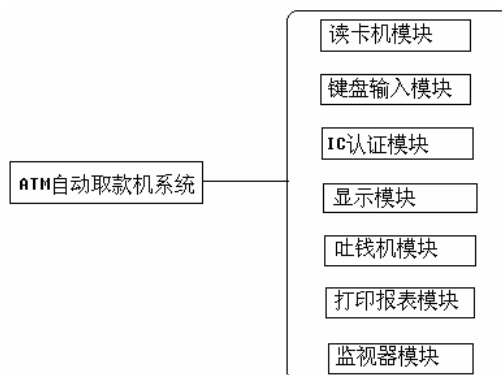


图 10-2 ATM 系统功能需求

### 10.2.2 读卡机模块需求

(1) 规格说明 读取客户插入的银行卡，如图 10-3 所示。

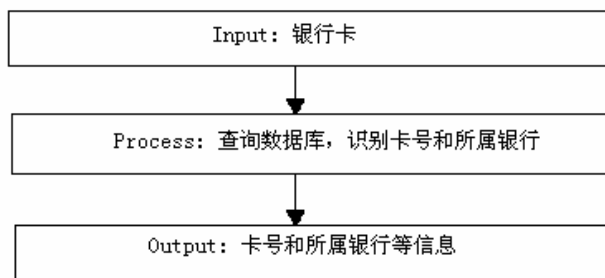


图 10-3 读卡机模块

(2) 引言 为了识别银行卡的类型。

(3) 输入 插入银行卡。

(4) 处理 读取卡号等信息，查找数据库中找到相关的信息，识别卡的类型并记录卡号和所属银行等信息。

(5) 输出 输出卡号及所属银行等信息。

### 10.2.3 键盘输入模块需求

(1) 规格说明 接受来自客户的输入，如图 10-4 所示。

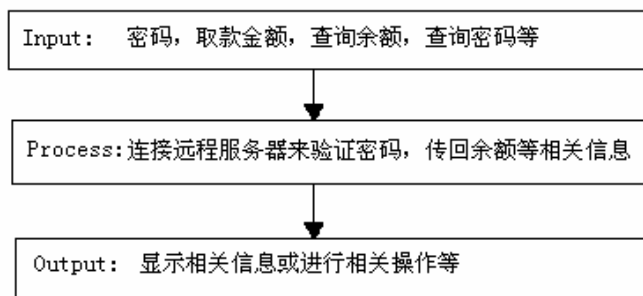


图 10-4 键盘输入模块

(2) 引言 客户通过键盘与机器交互。

(3) 输入 输入密码、输入取款金额、查询余额、查询密码、打印报表和取卡等。

(4) 处理 连接远程服务器来验证密码，传回余额等相关信息。

(5) 输出 验证密码正确则显示正确提示信息，查询余额操作并显示余额，如有取款命令则提示吐卡机工作，如需打印报表则提示打印机工作等。

### 10.2.4 IC 认证模块需求

(1) 规格说明 验证银行卡的真伪，如图 10-5 所示。

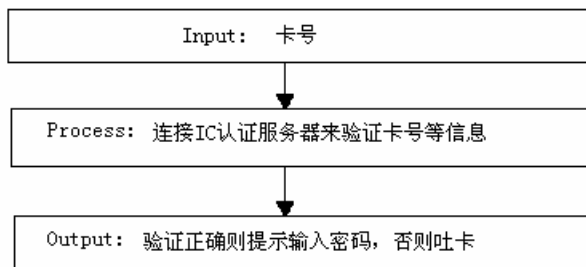


图 10-5 IC 认证模块

(2) 引言 通过 IC 验证来正确识别有效的银行卡。

(3) 输入 输入卡号。

(4) 处理 连接 IC 认证服务器来验证卡号等信息。

(5) 输出 验证正确则提示输入密码，否则吐卡。

### 10.2.5 显示模块需求

(1) 规格说明 显示信息给客户，如图 10-6 所示。

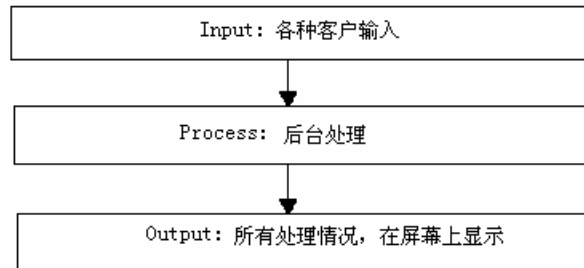


图 10-6 显示模块

(2) 引言 显示信息，提示进行有效操作。

(3) 输入 各种客户输入。

(4) 处理 后台处理。

(5) 输出 所有处理情况在屏幕上显示。

### 10.2.6 吐钱机模块需求

(1) 规格说明 提供现金给客户，如图 10-7 所示。

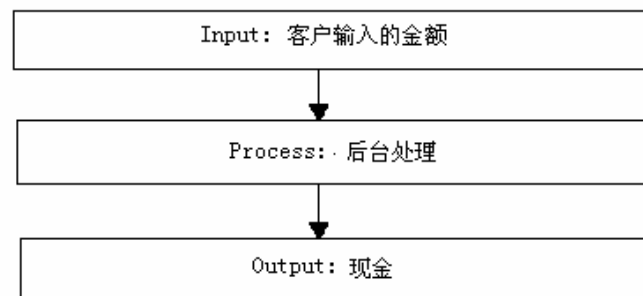


图 10-7 吐钱机模块

(2) 引言 根据客户输入的金额，以现金的形式给出，有限额。

(3) 输入 客户输入金额。

(4) 处理 后台处理。

(5) 输出 输出现金。



### 10.2.7 打印报表模块需求

(1) 规格说明 提供报表给客户，如图 10-8 所示。

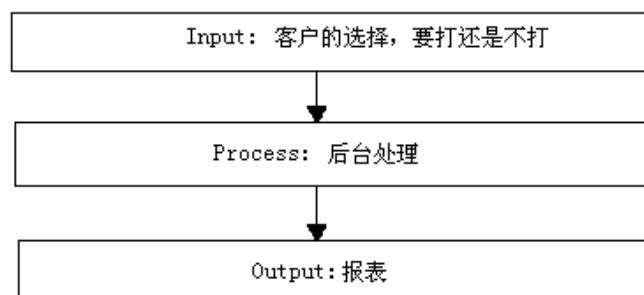


图 10-8 打印报表模块

(2) 引言 根据客户的选择来决定是否要打印报表。

(3) 输入 客户选择是否打印。

(4) 处理 后台处理。

(5) 输出 输出报表。

### 10.2.8 监视模块需求

(1) 规格说明 监视客户在取款机前的操作，如图 10-9 所示。

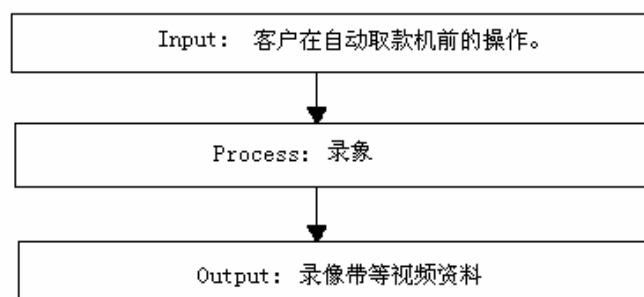


图 10-9 打印报表模块

(2) 引言 保障银行及客户的利益。

(3) 输入 客户在自动取款机前的操作。

(4) 处理 录像。

(5) 输出 录像带等视频资料。

### 10.2.9 数据库模块需求

(1) 规格说明 查询客户银行卡记录，如图 10-10 所示。

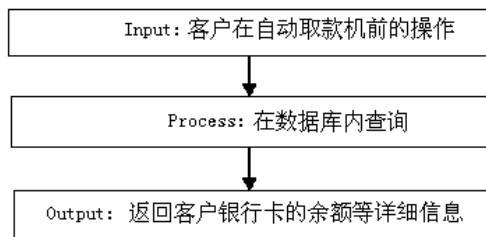


图 10-10 数据库模块

(2) 引言 ATM 客户端连接远程数据库读取数据。

(3) 输入 客户在自动取款机前的操作。

(4) 处理 在数据库内查询。

(5) 输出 返回客户银行卡的余额等详细信息。

## 10.3 系统用例模型

用例视图 (Use Case View) 强调从用户的角度看到的或需要的系统功能，是被称为参与者的外部用户所能观察到的系统功能的模型图。下面引导读者熟悉建模的顺序，掌握 UML 建模的一些基本方法，领会面向对象设计方法的实质。

建立用例视图分为以下几个步骤：角色的确定；创建用例；创建角色 - 用例关系图

### 10.3.1 角色的确定

角色不是系统的一部分，是与系统有交互作用的人或事物。通常情况下这代表了一个系统的使用者或外部通信的目标。

首先考察 ATM 系统需要为哪些人服务。可以归纳出如下：

- 客户使用 ATM 系统进行现金交易；
- 银行官员更改 ATM 的设置，放置现金，维护机器等
- 信用系统作为外部的角色参与整个交易过程

ATM 作为一个独立的系统，与客户、银行官员和信用系统这 3 个角色产生了交互。这里信用系统作为外部内容也是个角色。

所以可以创建角色：

- 客户；
- 银行官员；
- 信用系统。

使用 Rational Rose 的 Use Case View 中建立角色的视图，图 10-11 所示。



图 10-11 在用例视图中创建角色

### 10.3.2 创建用例

用例模型是系统和角色之间的对话，它表现系统提供的功能，即系统给操作者提供什么样的使用操作。

在创建用例时，碰到的一个问题是用例中的描述程度，即需要多大（或多小）的用例？没有惟一的、完全正确的答案。可以认为，第一规则应该是要用例典型地描绘了系统功能中从开始到结束的大部分作用。

用例是角色启动的，基于这样的考虑，ATM 系统根据业务流程大致可以分为以下的几个用例：

- 客户取钱；
- 客户存钱；
- 客户查询余额；
- 客户转账；
- 客户更改密码；
- 客户通过信用系统付款；
- 银行官员改变密码；
- 银行官员为 ATM 添加现金；
- 银行官员维护 ATM 硬件；
- 信用启动来自客户的付款。

使用 Rational Rose 的 Use Case View 中建立的用例如图 10-12 所示。



图 10-12 在用例视图中创建用例

### 10.3.3 创建角色用例关系图

在角色和用例之间存在关联关系,这种类型的关联关系通常涉及到角色和用例之间的通信关联关系。用户、项目管理员、分析人员、开发人员、质量保证工程师和任何对系统感兴趣的人都可以浏览这个框图,了解系统的功能。这个关系图直观地显示了 ATM 系统使用用例与角色间的交互。客户的用例关系图如图 10-13 所示。

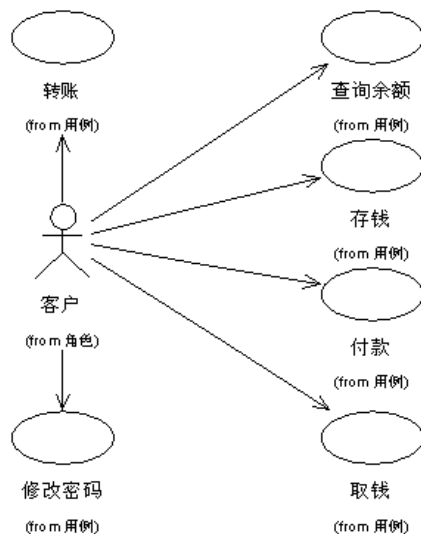


图 10-13 客户的用例关系图

图中一共有 6 个用例:转账、查询余额、存钱、付款、取钱和修改密码。在这个关系图中,着重展示了客户能在 ATM 取款机前的业务用例。这与真实的情况基本相符,都描述了客户能

干什么。箭头表示谁发起的这个动作，此处表明全部是客户发起的。

银行官员的用例关系图如图 10-14 所示。

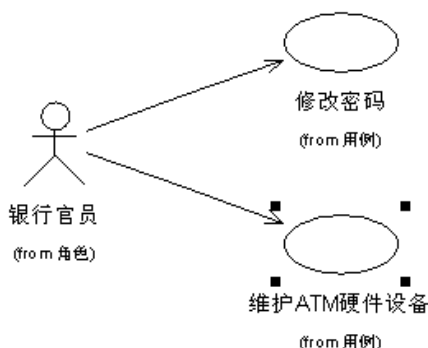


图 10-14 银行官员的用例关系图

银行官员的业务用例相对来说就比较少，只有两个，因为他不是客户，没什么操作要执行。银行官员的主要任务就是管理维护 ATM 自动取款机，保持它们正确的运行，维护硬件设备主要是添加现金，给机器上油，重新启动系统，关闭系统，把客户被吞的卡取出等操作。

信用系统的用例关系图如图 10-15 所示。

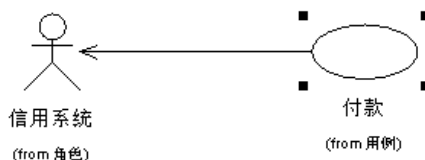


图 10-15 信用系统的用例关系图

信用系统的用例在只有一个，因为跟客户联系相对较少，所以在这里不详细介绍了。

以上 3 个模型分别为 3 个角色而建立，下面给出整个系统的用例关系图如图 10-16 所示。

在这里可以清晰地看到整个 ATM 系统的业务框架，并且可以从这个框图获取大量信息。

通过这个案例，读者可以知道谁与系统进行交互。通过查阅使用案例与角色，可以了解项目的具体范围，这样有助于寻找缺少的功能。例如还需要增加客户透支的功能，这时只要增加一个用例“透支”即可。

通常，一个系统要创建几个用例框图，高层框图在 Rational Rose 中称为主框图，显示用例包（组）。其他框图显示一组用例与角色。创建多少用例框图和取什么名字完全可以自行决定。框图中应该有足够的信息，又不至于太拥挤。

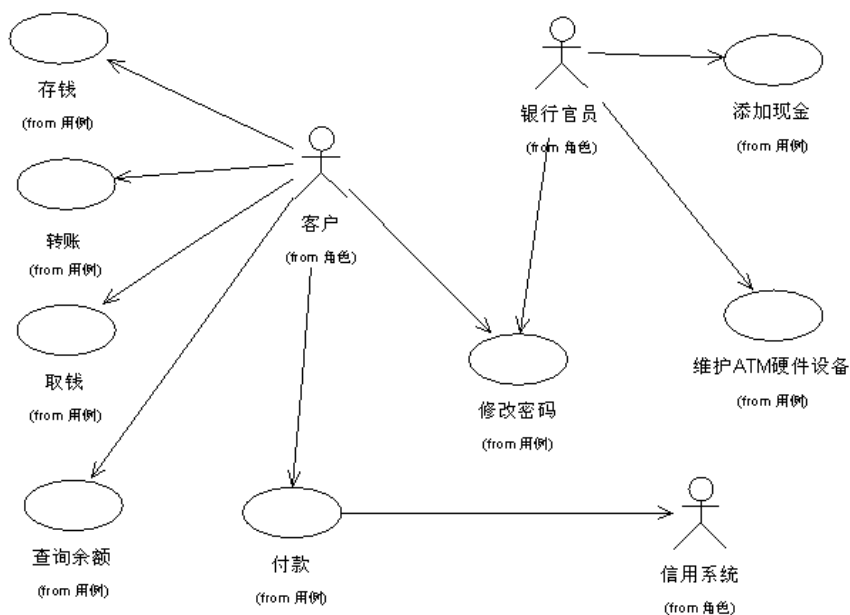


图 10-16 整个系统的用例关系图

## 10.4 系统动态模型

动态模型包括许多框图：活动图、时序图和协作图等。创建这些框图目的是为了更好地了解业务流程。这些框图是对用例图的补充。

### 10.4.1 创建活动图

活动图展示了系统中的功能流，可以在业务模型中显示业务 workflow；可以在收集需求时显示一个使用案例的事件流。这些框图定义 workflow 从那里开始，到哪里结束，workflow 中发生哪些活动，这些活动按什么顺序发生。活动是 workflow 期间完成的任务。

活动图可以分为垂直泳道，每个泳道表示 workflow 中不同的角色。由一个泳道中的活动，就可以知道这个角色的责任。通过不同泳道中活动之间的过渡，就可以了解谁要与谁进行通信。这些信息在建模或理解业务过程时非常的重要。

如图 10-17 所示，客户服务代表，信用部经理还有客户三者发生了相互的关系。首先客户服务代表收集信息，并建立客户账号，也就是 Account[Initializing]。然后由信用部经理检查信用历史，分两种情况，符合和不符合。符合就批准账号申请，不符合就拒绝账号申请，分别对应为 Account[Denied]和 Account[Approved]。账号被批准后，传给客户，最后客户领到银行卡，也就是 Account[Open]。

经过这样的可视化建模，读者可以比较清楚地知道整个开户过程的业务流程。了解了这些知识后，就可以为将来的细化做准备。

图 10-17 所示演示了活动图的例子。框图中的活动用导角矩形表示，这是工作流期间发生的步骤。工作流影响的对象用方框表示。开始状态表示工作流开始，结束状态表示工作流结束，决策点用菱形表示。

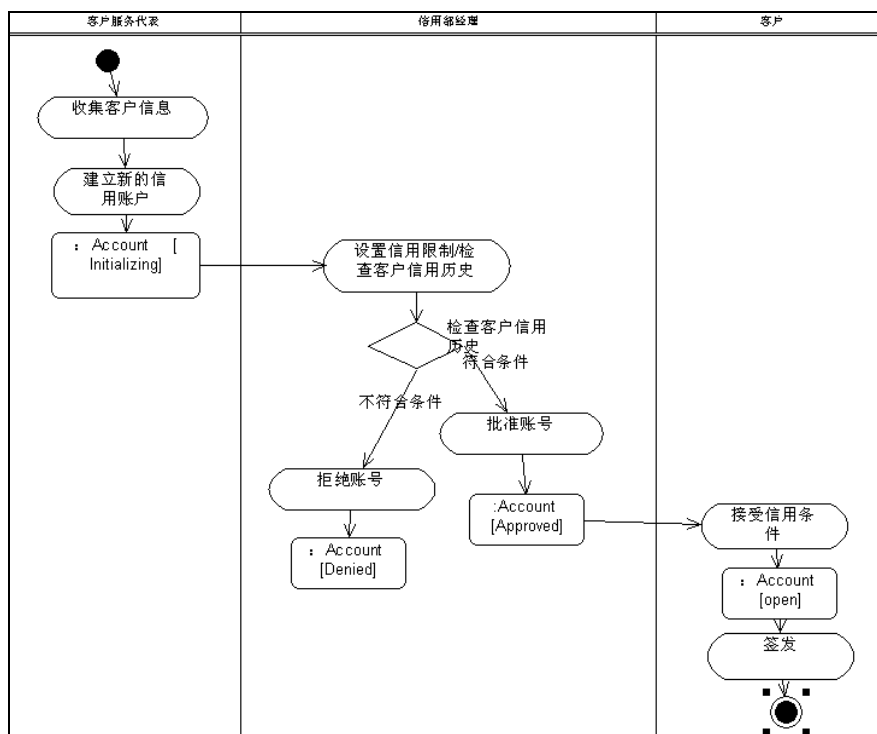


图 10-17 “开户”的活动图

#### 10.4.2 时序图

表现系统流程以及系统元素之间的交互关系可以用：时序图与协作图。

时序图与协作图能够很清晰的表达系统流程以及系统元素之间的交互关系，在时间与空间顺序上说明系统元素之间的关系。

时序图的功能是按时间顺序描述系统元素间的交互，协作图的功能是按照时间和空间顺序描述系统元素间的交互和它们之间的关系。

时序图显示使用案例中的功能流程。例如，取钱使用案例有几个可能的程序，如想取而没钱，想取而 PIN 错误等。

时序图中靠箭头在各个对象间传递信息，而且这些箭头要么是水平的，要么是自反的。箭头的方向表示了信息的流向，可以流向其他对象，也可以流向自己本身的对象。步骤用数字来标志。

例如取 100 元人民币的正常情形如图 10-18 所示。

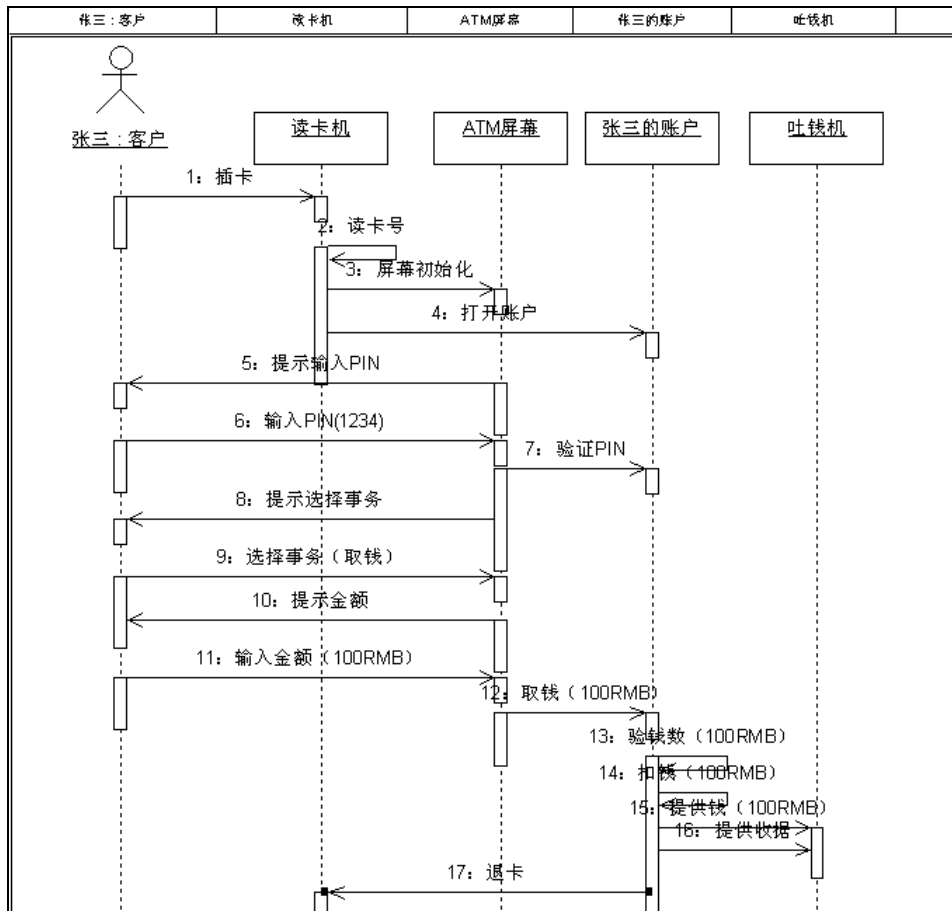


图 10-18 取 100 元人民币的时序图

时序图中显示了取钱使用案例的流程。图顶部显示了涉及的角色。系统完成取钱使用案例所需的对象在框图顶部显示。每个箭头表示角色与对象或对象与对象之间为完成所需功能而传递的消息。

关于时序图要说明的另一点是，它只显示对象而不显示类，类表示对象的类型。

取钱使用案例从用户将卡插入读卡机开始，读卡机读卡号，打开张三的账目对象，并初始化屏幕。屏幕提示输入 PIN，张三输入 PIN (1234)，然后屏幕验证 PIN 与账户对象，发出相符的信息。屏幕向张三提供选项，张三选择取钱。然后屏幕提示张三输入金额，他选择 100RMB。然后从账户中取钱，启动一系列账目对象要完成的过程。首先，验证张三的账目中至少有 100 RMB，然后从中扣掉 100RMB，再让吐钱机提供 100RMB 现金。另外，还需要让吐钱机提供收据，最后让读卡机退卡。

这个例子演示了取钱使用案例的全过程。用户可以从这个框图看到取钱过程的细节。分析人员可以看到处理流程。开发人员看到要开发的对象和这些对象的操作。质量保证工程师可以看到过程细节，并根据这个过程开发测试案例。时序图对项目中的所有人都是有用的。



## 10.4.3 协作图

如图 10-19 所示的时序图对应的协作图如图 10-19 所示。

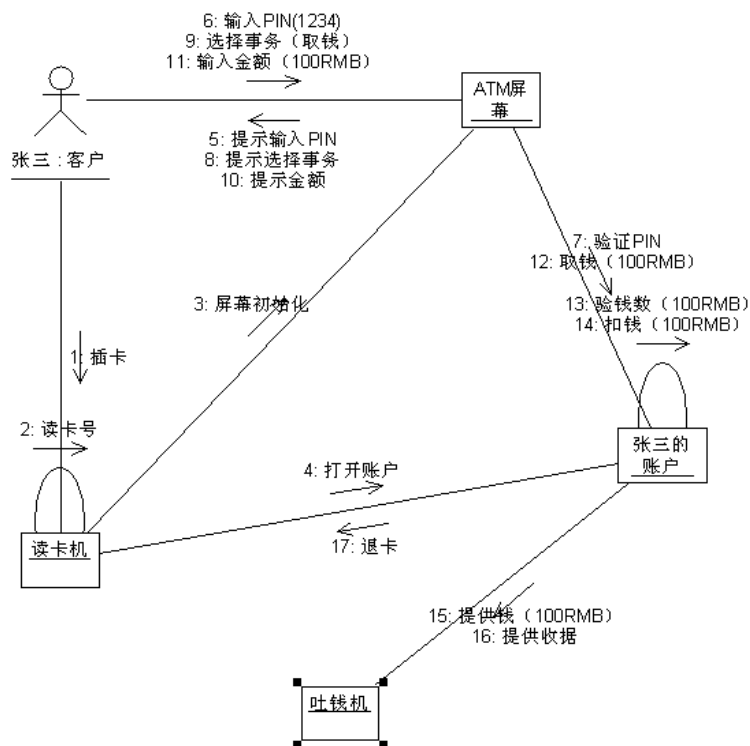


图 10-19 取 100 元人民币的协作图

协作图显示的信息与时序图相同,但协作图用不同的方式来显示这个信息,两种图具有不同作用。

时序图演示的是对象与角色随时间变化的交互,而协作图则不参照时间而显示对象与角色的交互。

假设协作图是星形的,几个对象与一个中央对象通信,则系统分析师可能认为系统对中央的依赖太强,可能重新设计对象,以便均匀的分配处理工作。这种交互很难在时序图中看到。

## 10.5 创建系统包图

包将具有一些共性的类组合在一起,封装类时有常用的几个方法:按版型、按功能、按嵌套、以上方法的组合。

### 10.5.1 ATM 系统包图

在定义具体的类之前，先在宏观的角度上将整个系统分割成多个独立的包。此处把整个 ATM 系统分成如图 10-20 所示的包。

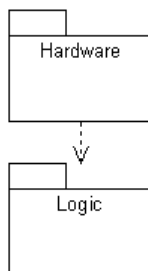


图 10-20 系统包图

整个系统可以看成硬件和逻辑两块，分别控制不同的应用。这样做的好处是为以后升级做准备。因为在面向对象系统里，可维护性、可升级性比什么都重要。读者要明白这一点。

### 10.5.2 Hardware 包内的类

Hardware 包内的类组织如图 10-21 所示。

把 ATM 系统内所有与硬件相关的内容全部放在 Hardware 包内，好处是想增加或删除硬件可以在一个包内进行，而与其他包无关。比如要增加一个硬件用于把 100 元人民币换成 2 张 50 元人民币。

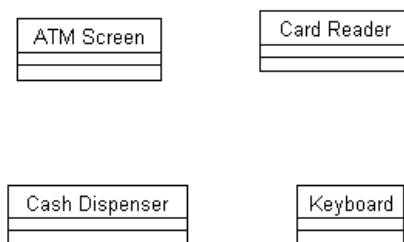


图 10-21 Hardware 包内类

在 Hardware 包内，有以下 4 部分内容：ATM Screen（屏幕）、Card Reader（读卡器）、Cash Dispenser（吐钱机）和 Keyboard（键盘）。

### 10.5.3 Logic 包内的类

Logic 包内的类如图 10-22 所示。这个包里是一些逻辑性的内容。起关键作用的是账目类，它先从远程服务器得到张三的账目信息，然后根据硬件发出的指令进行相关的操作。

在 Logic 包内，有两部分内容：Account（账户）、（Database Connector）数据库连接。

此处不涉及到 ATM 硬件，是纯粹的软件模块。把包图搞清楚后，就可以开始下面类的详细建模了。

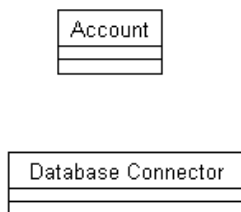


图 10-22 Logic 包内类

## 10.6 系统类模型

类模型是在 Logical 视图中显示的。Logical 视图关注系统如何实现使用案例中提出的功能。它提供详细的图形，描述组件间如何关联。此外，还包括需要的类、类图和状态图。利用这些元素，开发人员可以构造系统的详细设计。

### 10.6.1 Logical 视图

Logical 视图关注的焦点是系统的逻辑结构。重复使用是一个主要目的，通过指定类的信息和行为、组合类，以及检查类和包之间的关系，就可以确定可以重复使用的类和包。完成多个项目后，可以将新类和包加进重复使用库中。今后的项目可以组装现有的类和包，而不必一切从头开始。

在 Logical 视图中，可以看到 ATM 系统中类和包的组织关系。认真的观察该视图有助于开发人员在编码之前计划系统结构，保证一开始就设计合理。

Logical 视图如图 10-23 所示。

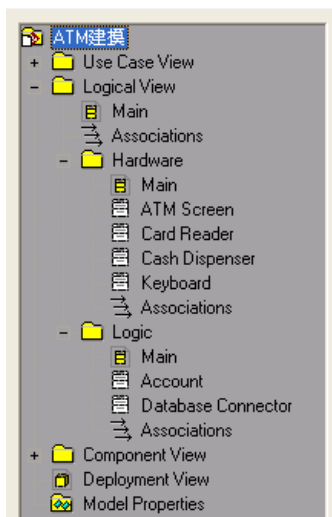


图 10-23 Logical 视图

### 10.6.2 类图

类图中的类是针对时序图和协作图中每种对象而创建的。如图 10-24 ~ 图 10-26 所示分别演示了逻辑包、硬件包和整个系统中的类图。

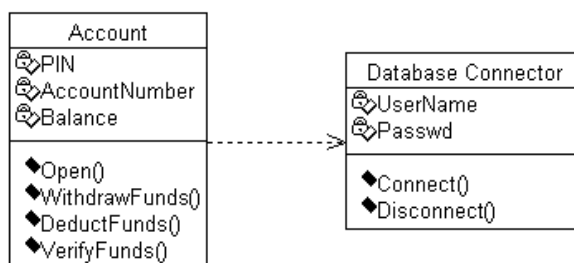


图 10-24 Logical 包内的类图

图 10-24 所示联系是把 Account (账户) 和 Database Connector (数据库连接) 两个类。在 Account, 有 3 个属性, 4 个方法, Database Connector 中有 2 个属性和 2 个方法。

箭头从 Account 指向 Database Connector, 说明了 Account 类要发消息给 Database Connector 类, 请求得到信息。在用户名和密码经过验证后, 就可以访问数据库了。

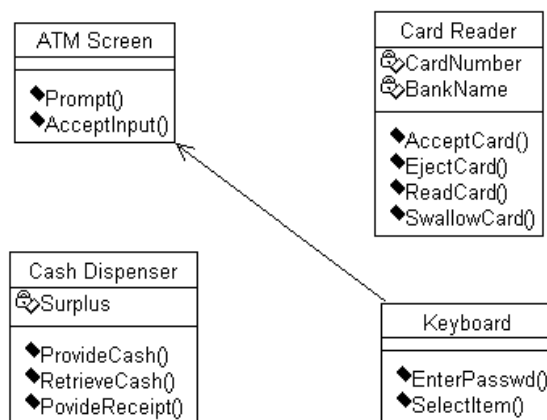


图 10-25 Hardware 包内的类图

在硬件的包内，有 4 个类，其中有两个类有相互的联系。Keyboard 类要发送消息来传递客户发出的指令，例如客户输入的密码、选择存钱还是取钱等操作。

还有两个类是 Cash Dispenser 和 Card Reader，这两个都跟 Account 类有联系，但是彼此之间没有消息联系，在这个包内是相对独立的。Card Reader 类的属性是卡号和银行代号，有 4 个方法，接受、拒绝、读卡 and 吞卡。Cash Dispenser 类的属性只有 Surplus（金额）1 个，方法有提供现金、收回现金等。

在这里，已经把硬件的操作都建模完毕了，如果还有其他额外的功能，可以自己添加，不影响其他功能的。

下面重点介绍系统类图。如图 10-26 所示，共有 6 个类，其中 Account 类跟其他 4 个类都有关联。连接类的直线显示类之间的通信关系。

例如，Account 类连接 ATM Screen 类，因为两者要直接通信。Card Reader 类与 Cash Dispenser 类不直接相连，因为两者需要通信。有些属性和操作的左边有小锁状的图标代表什么意思呢？小锁状图标表示专用属性和操作，只能在该类中访问。

开发人员用类图开发类。Rational Rose 可以产生类的框架代码，然后开发人员便可用所选语言来填充细节。

分析人员用类图显示系统细节。如果需要相互通信的类之间没有建立联系，则可以马上在系统类图中看出。

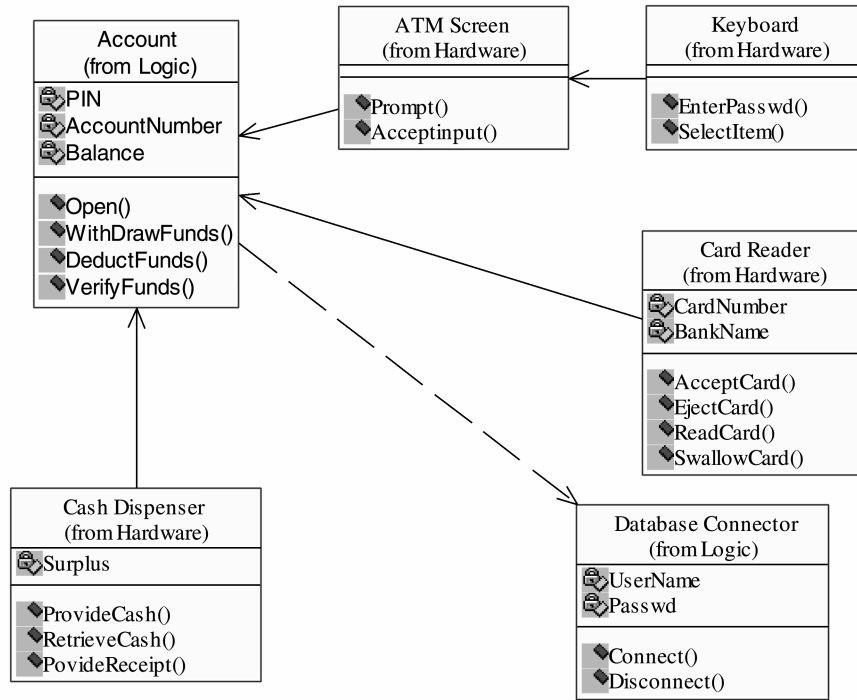


图 10-26 系统类图

### 10.6.3 状态图

状态图提供了建模对象各种状态的方式。类图提供了类及其相互关系的静态图形，而状态图则可以建模系统的动态功能。

状态图实现对象的功能，例如银行的账目可能有几种不同的状态，可以打开、关闭或者透支。因为账目在不同状态下功能是不同的。如图 10-27 所示，显示了银行账目的状态图，有 3 个状态：开启、透支和关闭。其中有很多消息通过箭头流动，上面的文字描述了消息流动的条件。

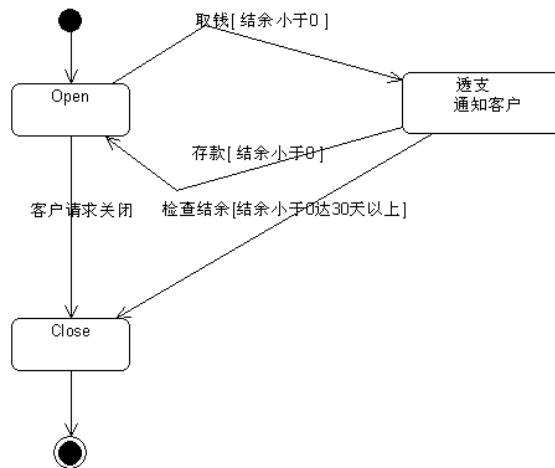


图 10-27 账目类的状态图

## 10.7 系统部署

ATM 系统部署是整个项目实施过程的最后的阶段，其实就是把该系统中涉及到的硬件、软件整合到一起，并且可以让系统运行起来。

在部署中有两种视图，组件图和配置图。

### 10.7.1 组件图

在组件图中，着重考虑系统的实际结构。首先，要确定类如何组成代码库，然后要考虑不同的可执行文件，动态链接库（DLL）文件和系统中的其他文件。

组件图包含了模型代码库、可执行文件、运行库和其他组件的信息。组件是代码的实际模块。

如图 10-28 所示，在 Rose 中，组件图里面分为两部分：Client 和 Server。在 Client 中包含了 ATM 取款机的基本硬件。在 main 图中显示了整个结构。

前面提到，组件是代码的物理模块。组件可以包括代码库和运行文件。例如，如果使用 C++，则每个.cpp 和.h 文件是单独的组件。编译代码后创建的.exe 文件也是组件。

生成代码之前，可以将每个文件映射相应的组件。例如在 C++中，每个类映射两个组件，一个表示类的.cpp 文件，一个表示类的.h 文件。在 Java 中，每个类映射到一个组件，表示这个类的.java 文件。将来生成代码时，Rose 用组件信息来创建相应的代码库文件。

一旦组件创建后，它们就被加进组件图中。组件间惟一的关系类型就是依赖性关系。依赖性关系要求一个类要在另一个类之前编译。

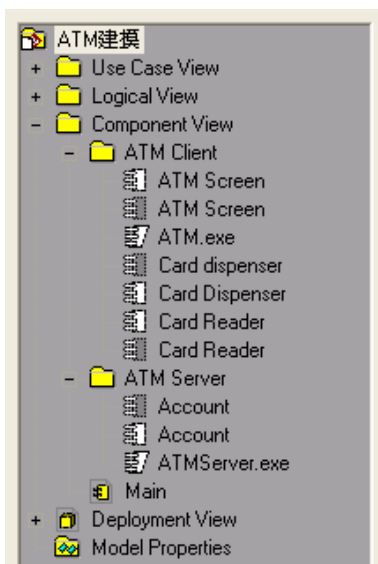


图 10-28 ATM 系统的组件图

组件的类型主要有两种：源代码库和运行组件。

如图 10-29 所示是 ATM 系统 Server 的组件图，图中 Account 类跟 Server 服务器有紧密的联系。箭头表示联系的方向。深灰色表示是账目类的一个实例。

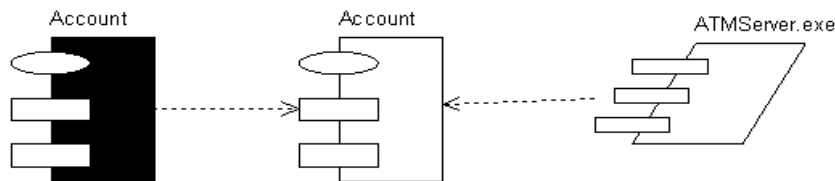


图 10-29 ATM 系统 Server 的组件图

如图 10-30 所示是 ATM 系统 Client 的组件图，图中深灰色的部分表示了实例，最终客户端被打包成一个 exe 应用程序，包括 3 个组件：Card Reader、ATM Screen 和 Card Dispenser。

此处已经把组件图建立完毕。感兴趣的读者可以深入分析某些细节，以提高自己的分析水平。

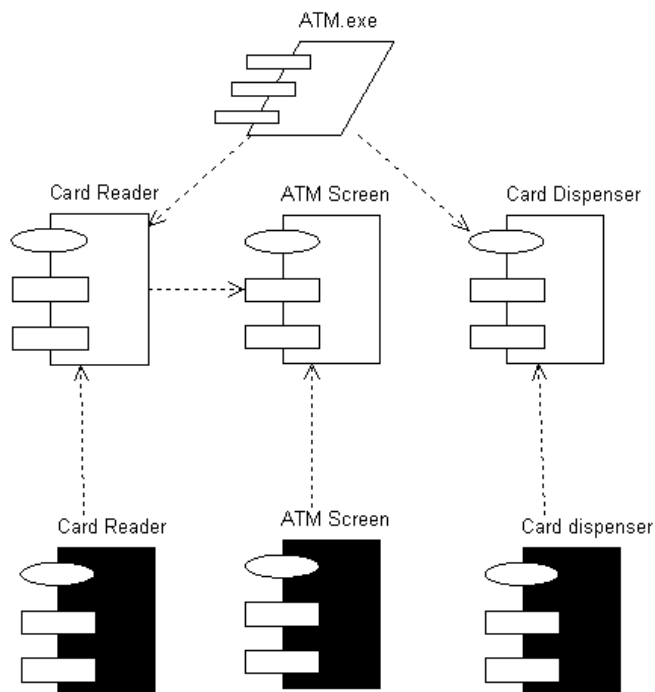


图 10-30 Client 的组件图



## 10.7.2 配置图

配置图考虑应用程序的物理部署,如网络布局和组件在网络上的位置等问题。还需要进一步考虑以下几个问题:具有多少网络带宽、希望出现多少并发用户,以及服务器关闭怎么办等。

配置图关注系统的实际部署,但与系统的逻辑结构有所不同。配置图包含了处理器、设备、进程和处理器与设备之间的连接。

ATM 系统的配置图如图 10-31 所示。

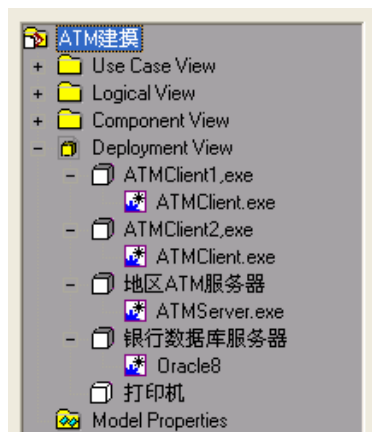


图 10-31 ATM 系统的配置图

如图 10-32 所示是 ATM 系统的配置图。

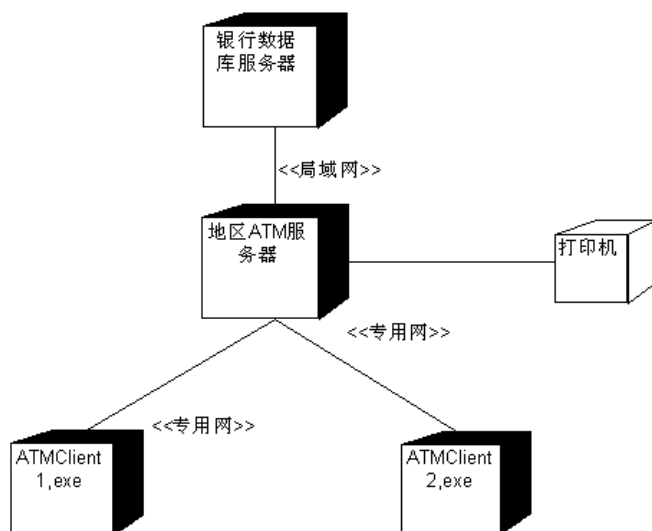


图 10-32 ATM 系统的 Component 框图

在配置图中只画出了两个 ATM Client,一个地区的 ATM 服务器,银行数据库服务器,以及一个打印机,大致上描述了整个系统的物理部署情况。

在配置图中涉及到了网络连接，这些硬件设备处在专用网和局域网中。一个地区的 ATM 服务器可以为许多个 Client 服务。

# 第 11 章 大型仓库信息管理系统开发

## 11.1 系统概述

“仓库信息系统”是一套功能强大而又操作简便、实用的仓库管理软件，包括用户登录、仓库管理、业务查询和系统设置 4 大管理功能。本系统在功能设计上具有前瞻性，吸收和借鉴了国际上先进的仓储管理思想。通过供应商、仓库及客户三者之间信息的沟通与指令的有效传递，将制造商和供应商的库存成本与资金占有率降到最低限度。本系统主要适用于第三方物流企业的仓储管理。

仓库存放的货物品种繁多，堆存方式以及处理过程也非常复杂，随着业务量的增加，仓库管理者需要处理的信息量会大幅上升，因此往往很难及时准确的掌握整个仓库的运作状态。针对这一情况，本系统在满足仓库的基本管理功能基础上发挥信息系统的智能化，减轻仓库管理人员和操作人员的工作负担。

系统主要的实现目标是监控整个仓库的运转情况；提供完善的任务计划功能，由整个操作的指令中心来安排进出任务，确认任务的开始，进货管理和出货管理按其指令执行即可；实时监控所有货物的在线运动情况，实时提供库存变化的信息。

## 11.2 需求分析

在软件开发的初期，开发人员及用户往往忽略信息沟通，导致软件开发完成后，不能很好的满足实际的需要。而返工不仅在技术上给开发人员带来巨大的麻烦，还会造成人力、物力的浪费。因此，只有弄清楚客户的需求，才能真正开发出满足客户需求的管理信息系统，才能够让整个系统发挥其相应的作用。

### 11.2.1 系统总体功能需求

一个功能完善的仓库信息管理系统，必须包括以下的几个模块。

#### (1) 用户登录

由用户登录、用户注销、退出系统 3 个部分组成。用户可以用两种身份登录本系统：普通操作员或经理（管理人员）；不同身份登录被系统授予不同的使用权限，这样提高了本系统的安全性，避免了无关人员获取不在他权限范围内的信息。用户在登录后可以不退出本系统，而采用用户注销的方式使系统不存在激活状态下的用户。

### (2) 仓库管理

仓库管理系统中,包括仓库进货、仓库退货、仓库领料、仓库退料、商品调拨和仓库盘点 6 个功能模块组成。仓库管理系统是整个仓库信息系统的核心,是所有数据的来源。用户通过本系统的使用,可以使仓库的空间得以优化,减低无效和冗余的作业,使库存精度更准确,库存周转率提高及库存资金占用减少。“仓库进货→仓库领料→仓库盘点”组成了仓库管理的重要过程。

### (3) 业务查询

业务查询系统中,包括库存查询、销售查询和仓库历史记录查询 3 个功能模块组成。库存查询实时提供库存变化的信息,随时应不同客户的要求得到其当前的库存。销售查询提供了一个完整的出货查询平台:用户可以根据货物的 ID 号查询某个时间段里该货物的销售情况,该功能可以使企业的管理人员能够以最快的速度了解仓库的出货情况和与仓库相配套的商场的销售情况,方便企业管理人员根据不同的情况及时的调整经营战略。仓库历史记录查询功能模块:在本系统中仓库进货、仓库退货、仓库领料、仓库退料、商品调拨和仓库盘点的任一个操作都储存在数据库中,本功能模块就是查询任意一条操作记录。

由此可知,本仓储管理模式通过供应商、仓库及客户间的信息沟通与指令的及时有效传递,将制造商和供应商的库存成本与资金占压降到最低限度。

### (4) 系统设置

系统设置包括供应商设置和仓库设置两个部分。供应商是货物的提供者,在供应商设置中:用户可以输入详细的供应商信息,包括联系方法、供应商名称和主要经营项目等信息,方便企业管理查询和维护。仓库设置:在本系统中,用户可以将整个仓库虚拟的分成数个仓库,每个仓库储存不同类型的货物,这样方便仓库货物的分类管理,也有利于提高仓库进货、出货的效率。

综上所述,系统的功能需求可用如图 11-1 所示的框图简要表示。

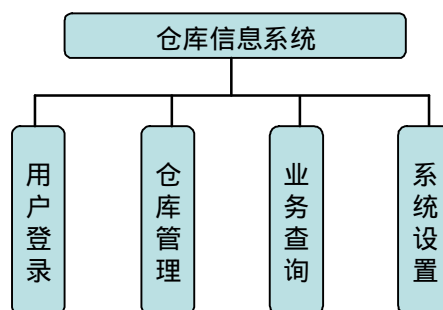


图 11-1 系统总体功能需求框图

## 11.2.2 用户登录

用户登录所包括的具体功能模块如图 11-2 所示。

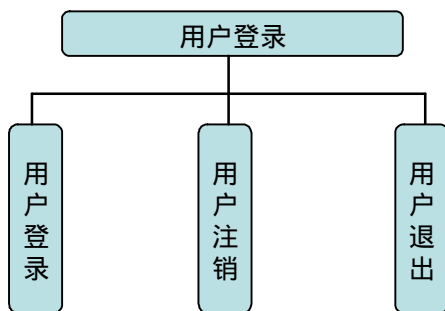


图 11-2 用户登录包括的功能模块需求框图

### (1) 用户登录

用户进入本仓库信息管理系统的入口，没有得到身份验证的用户只能拥有最低的使用权限，即只能选择退出系统或用户登录。本系统的使用者可以用两种身份登录到系统：普通操作员或经理（管理人员），不同的身份意味着不同的使用权限，这是一个稳定、安全的系统所必须具备的。

前置条件：无。

### (2) 用户注销

本系统中引入了类似 Windows 操作系统的用户注销功能，当用户在短时间内不使用本系统，他不必退出，只需要选择用户注销，这样可以使系统中不存在处于激活状态的用户，以便其他用户使用本系统。

前置条件：无。

### (3) 退出系统

用户在完成工作后，点击退出系统按钮可以安全的退出，以免不安全退出导致数据丢失情况的发生。

前置条件：无。

## 11.2.3 仓库管理

仓库管理包括的具体功能模块如图 11-3 所示。

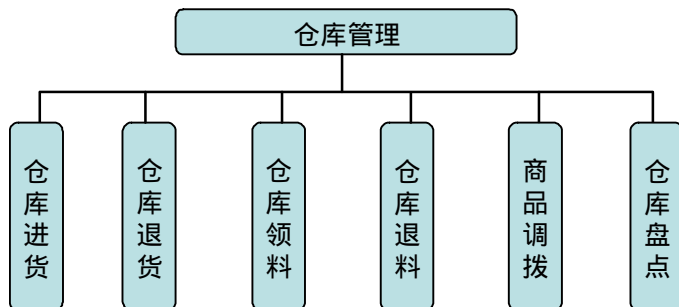


图 11-3 仓库管理包括的功能模块需求框图

仓库管理系统是整个仓库信息系统的核心,是所有数据的来源。根据详细的需求分析,企业在库存中面临的主要问题体现在:库存量较大,库存资金周转慢;不能及时统计库存物料;)库房人员重复工作多,效率低;不知道库存物资积压时间长短。

本系统从最初的采购到存储和交货,仓库管理将决定企业是否兑现了其承诺。从仓储计划到仓库操作和交叉运输,优化的仓储将有助于大幅减少企业的存货量和存货成本,因为企业将能保持较低的仓库存货水平,优化入库,保管和出库活动,并且协调载货量。

#### (1) 仓库进货

仓库存放货物品种繁多,堆存方式以及处理过程也非常复杂,随着业务量的增加,仓库管理者所需要处理的信息量大幅上升,因此管理者往往很难及时准确掌握整个仓库的运作状态。分析其原因在于:仓库在进货时没有输入详细、有效、完整的信息。

在本仓库信息系统中,仓库进货模块要求操作员输入商品号、进货数量、单价和供应商,系统会自动的将当前系统时间作为进货时间更新到数据库,并且会自动统计总进货金额。该操作完成后,相应货物的数量为原数量加进货的数量,并更新数据库。仓库进货功能如图 11-4 所示。

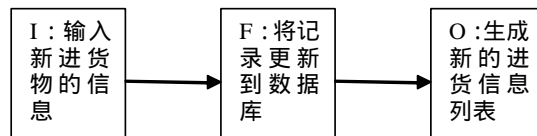


图 11-4 仓库进货功能

前置条件: 无。

#### (2) 仓库退货

仓库退货功能如图 11-5 所示。

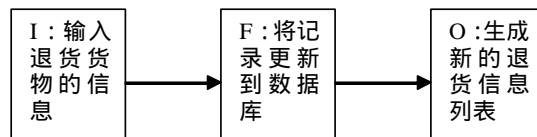


图 11-5 仓库退货功能

需求分析表明,企业仓库中的货物离开仓库主要有两种原因:企业无法销售某商品,将其退还给供应商;企业销售了一部分某商品,再从仓库调出部分库存的该种商品。

在本仓库信息系统中,仓库进货模块是为了第一种原因而设计的,它要求操作员输入退货商品号、退货数量、单价和供应商,系统会自动的将当前系统时间作为退货时间更新到数据库,并且会自动统计总退货金额。该操作完成后,相应货物的数量为原数量减退货的数量,并更新数据库。

前置条件: 存在该商品的进货信息。

## (3) 仓库领料

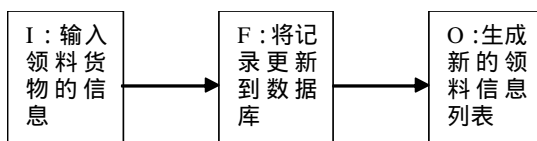


图 11-6 仓库领料功能

根据上述仓库退货中列举的原因,在本仓库信息系统中,仓库领料模块是为了第二种原因而设计的,它要求操作员输入领取商品号、领料数量、领料人和仓库管理员,系统会自动将当前系统时间作为退货时间更新到数据库,并且会自动统计总领料数量。在打印的单据中将会给出上述的所有信息。该操作完成后,相应货物的数量为原数量减领料的数量并更新数据库。

前置条件: 该商品库存信息。

## (4) 仓库退料

仓库退料功能如图 11-7 所示。

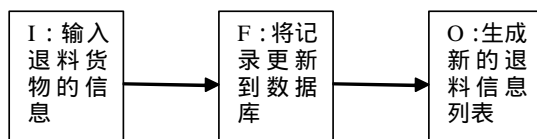


图 11-7 仓库退料功能

根据需求分析,企业功能遇到下述问题:企业销售部门在某段时间内没有销售出某件商品,这可能会造成销售部门的货物积压,因此部门就需要将该件商品返回一部分到仓库,这就是所谓的仓库退料。

在本仓库信息系统中,仓库退料模块要求操作员输入退料商品号、退料数量、退料人和仓库管理员,系统会自动的将当前系统时间作为退料时间更新到数据库,并且会自动统计总退料数量。在打印的单据中将会给出上述的所有信息。该操作完成后,相应货物的数量为原数量加退料的数量并更新数据库。

前置条件: 无。

## (5) 商品调拨

商品调拨功能如图 11-8 所示。

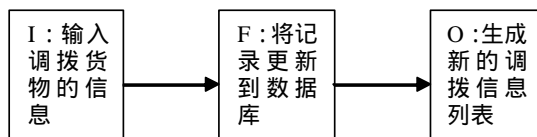


图 11-8 商品调拨功能

企业中很可能不止一个销售部门,而各个销售部门销售业绩也不相同。按照传统的仓库物流管理模式,业绩不好的销售部门要将其积压的商品退回仓库,业绩好的销售部门从仓库领取

一定数量的商品。本系统中引入了商品调拨的概念，即业绩不好的销售部门可以直接将其积压的商品移交一部分给业绩好的销售部门，不必通过仓库中转。该功能具有较大的灵活性和可扩展性，能够满足客户在仓储管理方面更多个性化的需求。

商品调拨模块要求操作员输入退料商品号、调拨数量、调拨人和仓库管理员，系统会自动的将当前系统时间作为调拨时间更新到数据库，并且会自动统计总调拨数量。在打印的单据中将会给出上述的所有信息。

前置条件：无。

#### (6) 仓库盘点

仓库盘点功能如图 11-9 所示。

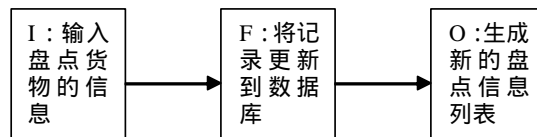


图 11-9 仓库盘点功能

仓库盘点的目的是为了更好地了解仓库准确的库存信息。盘点的周期和盘点的方式，企业可以根据自身的情况加以选择，不合理的仓库盘点，将会降低仓库库存信息的准确性、物料计划的准确性；不必要的仓库盘点将浪费企业的人力和物力。仓库盘点一般可以按照周期盘点、循环盘点和零点盘点 3 种方式进行，企业采用周期盘点这种方式的情况较多。

操作员可以在仓库盘点中任意增加或减少某件商品的库存数据，因此，出于安全性方面的考虑，本功能模块需要管理者（部门经理）能使用，并且所有的修改信息将会被存储到数据库中。

仓库盘点模块要求管理员输入某商品号、该商品实际数量，系统会自动的将当前系统时间作为盘点时间更新到数据库，并且会自动统计总盘点过程中修改的数量。在打印的单据中将会给出上述的所有信息。

前置条件：管理员身份登录。

### 11.2.4 业务查询

业务查询包括的具体功能模块如图 11-10 所示。

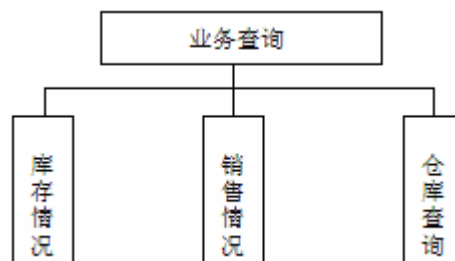


图 11-10 业务包括的功能模块需求框图



随着客户要求的不断提高,仓储管理在整个供应链管理当中占有非常重要的地位。以单据打印和数据记录为设计目标的传统仓储管理软件已远远无法适应现代仓储发展的要求。用户所需要的是仓储企业在实现信息化管理的基础上,不但可以向客户报告其产品的实时动态信息,还可以站在更高层面上为客户制定生产和销售计划,及时调整市场策略等方面提供持续、综合的参考信息,帮助仓储企业成为客户在整个供应链上最为紧密的合作伙伴。业务查询功能模块就是为用户提供了传统仓库管理系统以外的一些功能。

#### (1) 库存查询

库存查询如图 11-11 所示。



图 11-11 仓库库存查询功能

库存的可见性是决定企业的分销战略是否成功的最重要的一点。如果库存水平和组成,或所计划的对这些水平和组成的更新是模糊的、不正确的、过时的或完全不可信的,那么所有的仓储,运输和供应链管理活动都很有可能失败。换句话说,如果企业拥有清楚的、正确的、最新的和可靠的库存信息,将能更好地保证仓储,运输和供应链管理的成功。

在本系统的库存查询功能模块中,用户可以查询所有的商品的库存,也可以输入某件商品的 ID 号从而得到该商品的库存。总之,用户可以通过本查询模块轻松得到及时的库存信息。

☞ 前置条件: 商品库存表。

#### (2) 销售查询

销售查询如图 11-12 所示。



图 11-12 销售查询功能

该功能模块主要为企业管理者的经营决策提供参考的信息,更高层面上为客户在制定生产和销售计划,及时调整市场策略等方面提供持续、综合的参考信息。

在销售查询功能模块中,用户只需要选择某各时间段,计算机就会根据数据库中的资料给出该时间段中所有商品的销售情况。企业的经营者可以参考这样的信息来做出一些营销策略。由于本功能模块涉及到企业的经营信息,考虑到商业信息的安全性,需要管理员级的用户才可以使用本模块。

☞ 前置条件: 管理员身份登录。

#### (3) 仓库查询

仓库查询如图 11-13 所示。

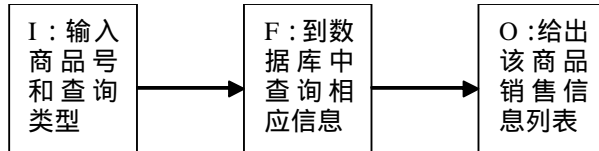


图 11-13 仓库查询功能

为适用客户不断提高的要求，增强仓储企业的核心竞争力，超越客户的期望。本系统提供了非常完整的信息查询，在仓库管理中输入的任何一条信息都可以在仓库查询模块中查询到。

在本模块中，用户可以选择查询的类型包括：仓库进货、仓库退货、仓库领料、仓库退料、商品调拨、仓库盘点；用户在选择查询的类型后，再输入需要查询的时间，系统就会返回用户所需的信息。

前置条件：无。

### 11.2.5 系统设置

系统设置包括供应商设置和仓库设置两个功能模块组成。供应商设置主要是提供一些供应商的信息以方便用户查询和使用。仓库设置的主要功能是用户可以将整个仓库虚拟的分成数个仓库，每个仓库储存不同类型的货物，这样方便仓库货物的分类管理，也有利于提高仓库进货、出货的效率。

仓库管理包括的具体功能模块如图 11-14 所示。

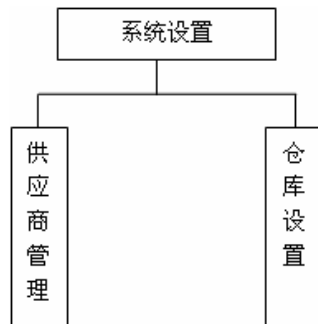


图 11-14 系统设置包括的功能模块需求框图

#### (1) 供应商管理

供应商管理如图 11-15 所示。

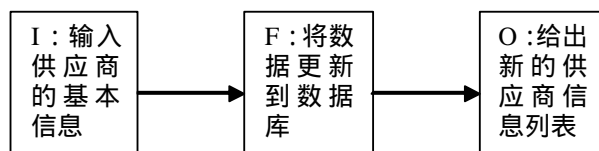


图 11-15 供应商管理功能

在本功能模块中,用户可以增加新的供应商,需要输入供应商的一些信息,包括供应商号、名称、联系人、联系电话、传真、地址和邮政编码。用户也可以对已经输入的供应商信息进行修改和查询。

☞ 前置条件:无。

#### (2) 仓库设置

仓库设置如图 11-16 所示。

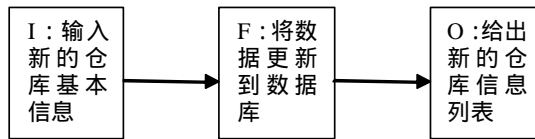


图 11-16 仓库设置功能

☞ 前置条件:无。

## 11.3 系统用例模型

前面的章节作者已经对本系统的任务和需求做了详细的说明。接下来,将对系统的流程和各个参与者之间的相互作用做详细的说明,将以 Rational Rose 作为 UML 建模的工具,使用用例图、时序图、协作图和类图等对整个系统进行描述、构造、可视化和文档编制。

用例视图是被称为参与者的外部用户所能观察到的系统功能的模型图。用例是系统中的一个功能单元,可以被描述为参与者与系统之间的一次交互作用。用例模型的用途是列出系统中的用例和参与者,并显示哪个参与者参与了哪个用例的执行。

本章的主要内容是引导读者熟悉建模的顺序,掌握 UML 建模的一些基本方法,领会面向对象对象的实质。

### 11.3.1 角色的确定

在 UML 中,Actors 代表位于系统之外和系统进行交互的一类对象。用它可以对软件系统与外界发生的交互进行分析和描述。

在仓库信息系统中,可以归纳出来的主要问题有:

- 购买的商品入库;
- 将积压的商品退给供应商;
- 将商品移送到销售部门;
- 销售部门将商品移送到仓库;
- 管理员盘点仓库;
- 供应商提供各种货物;
- 用户查询销售部门的营销记录;

- 用户查询仓库中的所有变动记录。

从上面所归纳的问题可以看出,本系统所涉及的操作主要是仓库信息的管理、维护以及各种信息的分析查询。

在本系统 UML 建模中,可以创建以下角色 (Actors):

- 操作员;
- 管理员;
- 供应商;
- 商品领料人;
- 商品退料人。

使用 Rational Rose 的 Use Case View 中建立 Actors 如图 11-17 所示。

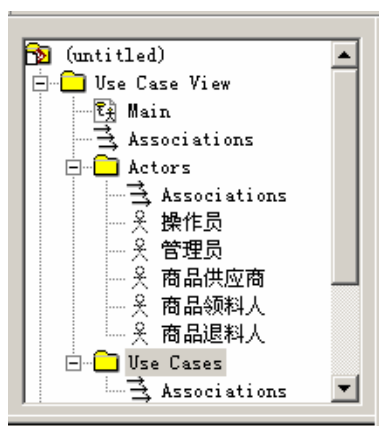


图 11-17 在 Use Case View 中创建角色

### 11.3.2 创建用例

用例本身是指一个用户或其他系统与要设计的系统进行的一个交互,这个交互是为了达到某个目标 (goal)。角色用来描述有该目标的人或系统。这个术语强调了任何人或系统拥有目标的事实。目标本身是一个动词短语,如“客户:下订单”,“店员:记录库存”。作为用例的一部分,有必要记录目标成功和失败对于活动者和系统的含义。在下订单的实例中,目标达成可能包括货物交给活动者和公司收到相应的货款。仔细定义目标成败是定义系统范围 (scope) 的基础。因为对于一个简易的订单输入系统,目标达成可能仅仅意味着订单已经经过验证并且交货已经排定日程。

仓库信息系统根据业务流程可以分为以下几个用例 (Use Cases):

- 仓库进货;
- 仓库退货;
- 仓库领料;
- 仓库退料;
- 商品调拨;
- 仓库盘点;

- 库存查询；
- 业务分析；
- 仓库历史记录查询；
- 供应商信息维护；
- 仓库信息维护；
- 用户登录；
- 用户注销；
- 退出系统。

使用 Rational Rose 的 Use Case View 中建立用例 (Use Cases) 如图 11-18 所示。



图 11-18 在 Use Case View 中创建用例

### 11.3.3 创建角色用例关系图

用例图 (Use Case Diagram) 采用了面向对象的思想, 又是基于用户视角, 绘制非常容易, 简单的图形表示便于让人们理解。

用例图表示了角色和用例以及它们之间的关系。它描述了系统、子系统和类的一致功能集合, 表现为系统和一个或多个外部交互者 (角色) 的消息交互动作序列。也就是角色 (用户或外部系统) 和系统 (要设计的系统) 的一个交互, 为了实现一个目的, 这个目的的描述通常是一个动词短语, 例如, 开立信用证, 给客户回单等。

操作员的用例关系图如图 11-19 所示。

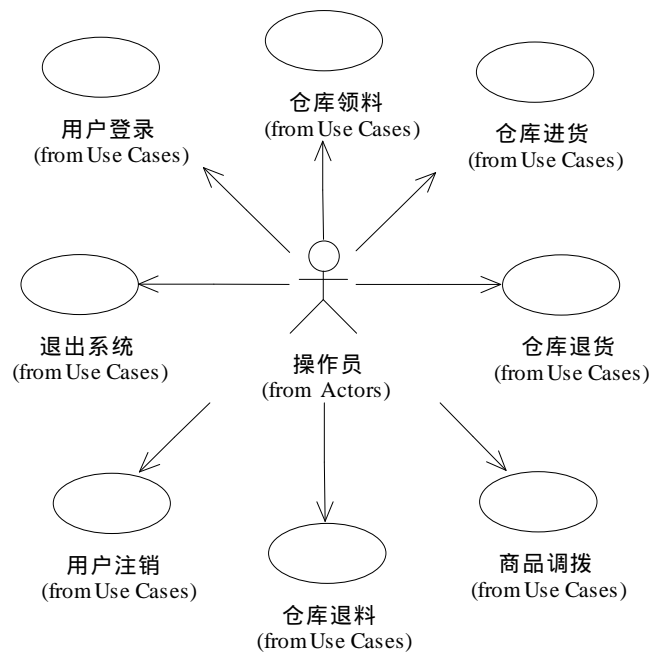


图 11-19 操作员的用例关系图

管理员的用例关系图如图 11-20 所示。

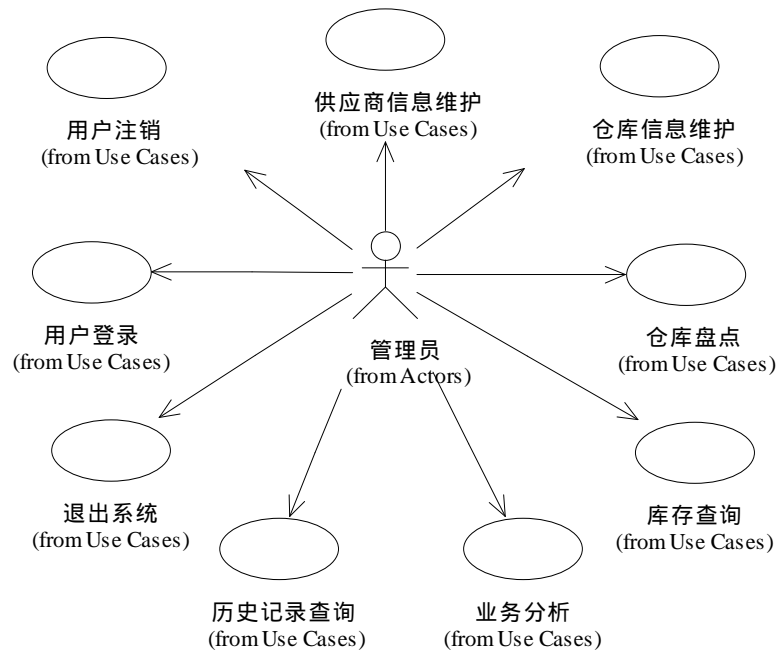


图 11-20 管理员的用例关系图

领料人的用例关系图如图 11-21 所示。

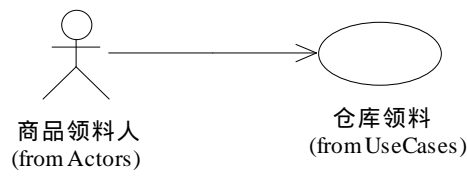


图 11-21 商品领料人的用例关系图

退料人的用例关系图如图 11-22 所示。

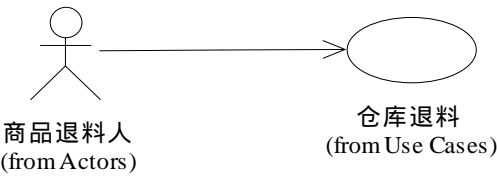


图 11-22 商品退料人的用例关系图

商品供应商的用例关系图如图 11-23 所示。

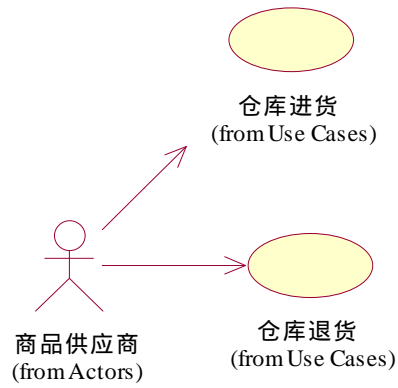


图 11-23 商品供应商的用例关系图

下面给出整个系统的用例关系图如图 11-24 所示。

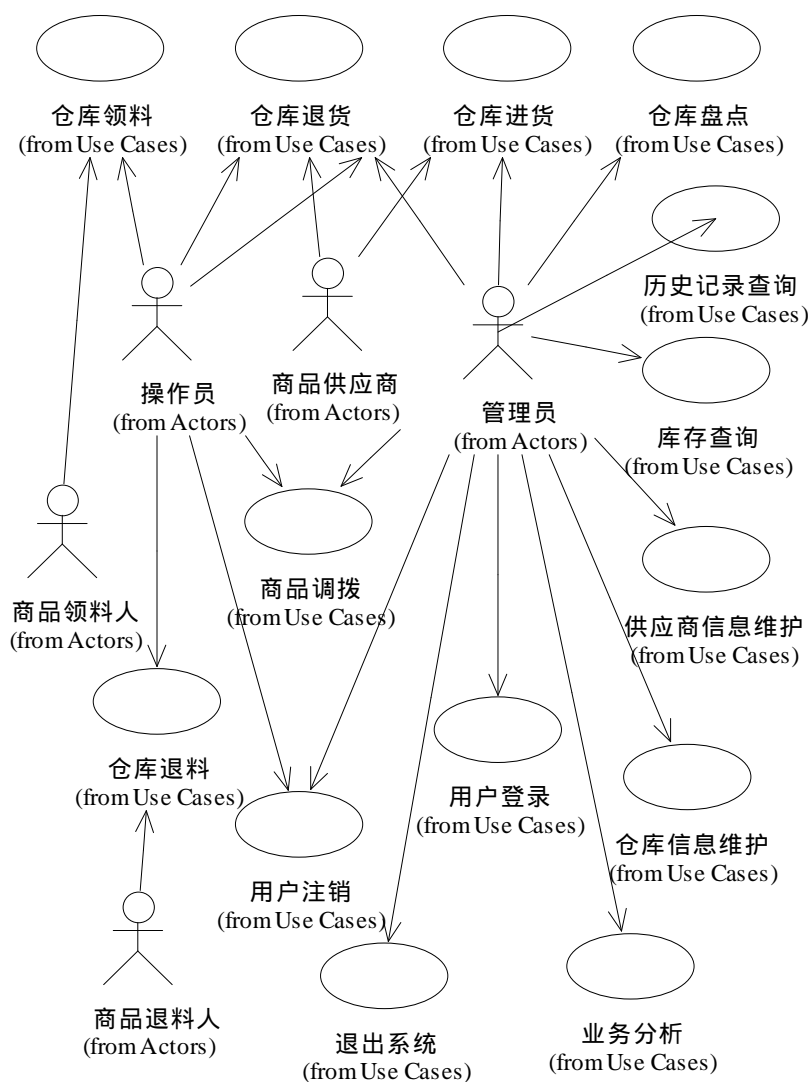


图 11-24 整个系统的 Use Cases 关系图

## 11.4 系统动态模型

动态模型包括许多框图：活动图（Activity）、时序图（Sequence）、协作图（Collaboration）等。建立这些框图目的是为了更好地了解业务流程。这些框图的出现是对用例图的巨大的补充。

### 11.4.1 活动图

活动图是一种特殊形式的状态图，用于对计算流程建模。活动图中的状态表示计算过程中



所处的各种状态，而不是普通对象的状态。通常，活动图假定在整个计算处理的过程中没有外部事件引起的中断，否则，普通的状态机更适合于描述这种情况。

活动图是对状态图的扩展。状态图突出显示的是状态，状态之间的转移箭头代表的是活动。而活动图突出显示的是活动。每个活动的图表示为圆角矩形，比状态图标更接近椭圆。活动图的起始点和中止点图标与状态图一样。

如图 11-25 所示描述了一个活动图的例子。框图中的活动用圆角矩形表示，这是工作流程期间发生的步骤。 workflow 影响的对象用方框表示。开始状态表示 workflow 开始，结束状态表示 workflow 结束，决策点用菱形表示。

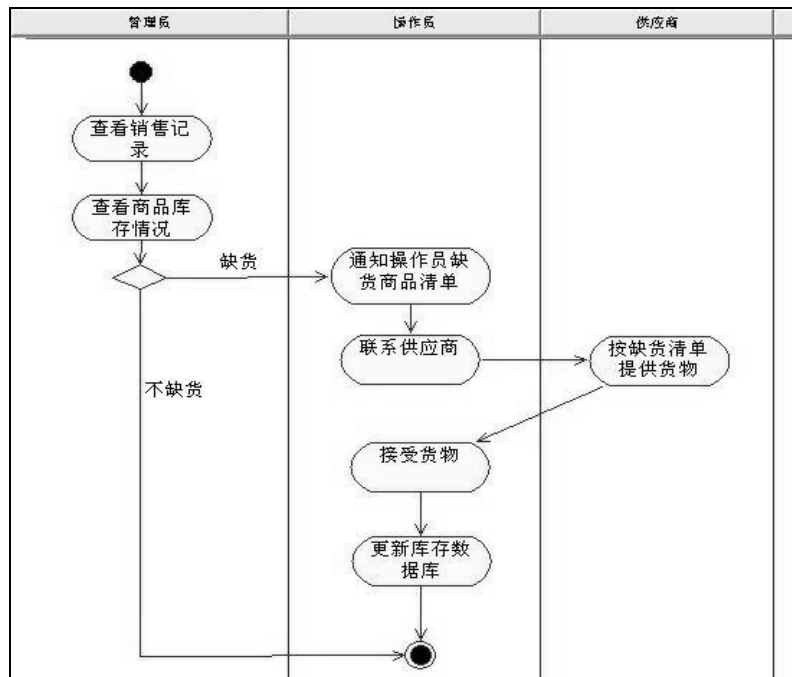


图 11-25 进货的活动图

在图中，管理员、操作员还有供应商三者发生了相互的关系。首先管理员查看销售记录判断商品销售状况，然后查看商品库存情况。如果发现仓库中商品库存充足则操作完毕，如果发现仓库中某商品库存出现不足，则通知操作员缺货商品清单，操作员领取清单后立即联系相应的供应商，供应商提供相应是商品，操作员接受货物，更新数据库，操作完成。

经过这样的可视化建模可以比较清楚的知道整个进货过程的业务流程。

#### 11.4.2 时序图

时序图 (Sequence Diagram) 表示对象之间传送消息的时间顺序。时序图可以用来进行一个场景的说明，即一个事务的历史过程。时序图中每一个类元角色用一条生命线来表示 (用垂

直线代表整个交互过程中对象的生命期)。生命线之间的箭头连接代表消息。时序图可以用来进行一个场景说明，即一个事务的历史过程。

时序图的用途是用来表示用例中行为的时间顺序。当执行一个用例行为时，时序图中的每条消息对应一个类操作或状态机中引起转换的触发事件。

(1) 管理员盘点过程时序图如图 11-26 所示。

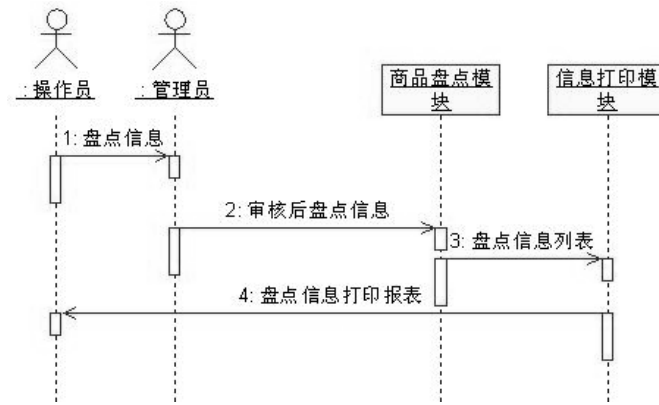


图 11-26 仓库盘点过程时序图

(2) 商品管理时序图如图 11-27 所示。

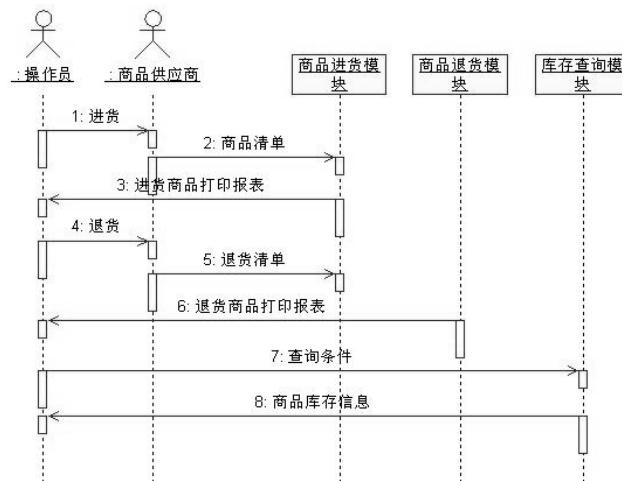


图 11-27 商品管理时序图

(3) 仓库历史记录查询时序图如图 11-28 所示。

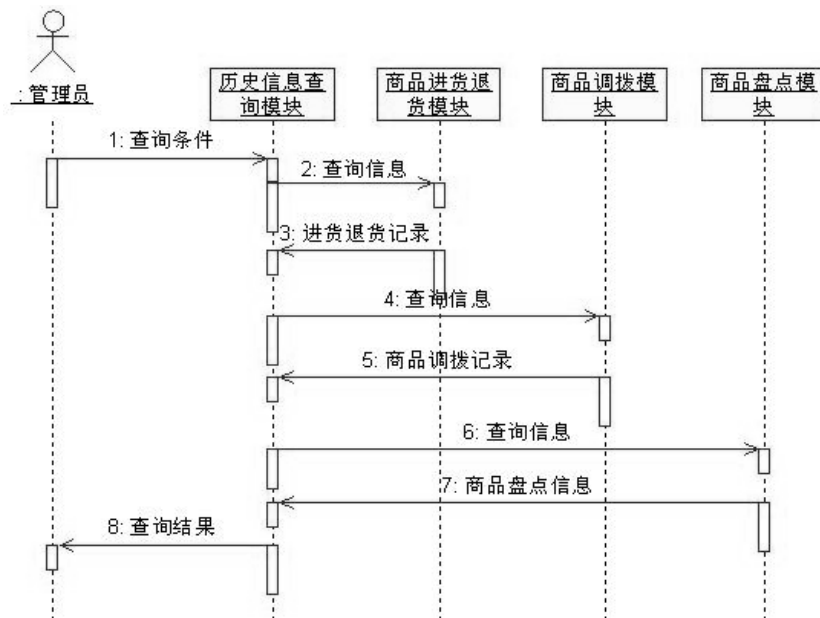


图 11-28 仓库历史记录查询时序图

### 11.4.3 协作图

协作图 (Collaboration Diagram) 用于在一次交互中对有意义的对象和对象间的链建模。对象和关系只有在交互时才有意义。类元角色描述了一个对象, 关联角色描述了协作关系中的一个链。

协作图的用途是表示一个类操作的实现, 协作图可以说明类操作中用到的参数和局部变量以及操作中类之间的关联。当实现一个行为时, 消息编号对应程序中的嵌套调用结构和信号传递过程。

(1) 管理员盘点过程协作图如图 11-29 所示。

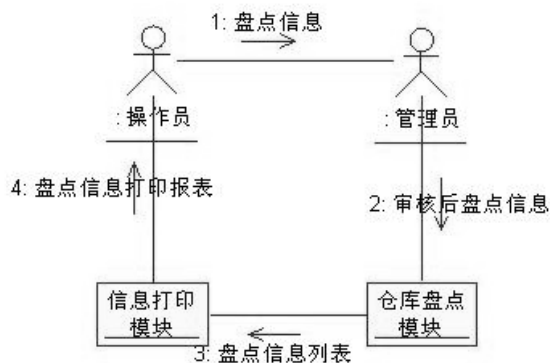


图 11-29 仓库盘点过程协作图

(2) 商品管理协作图如图 11-30 所示。

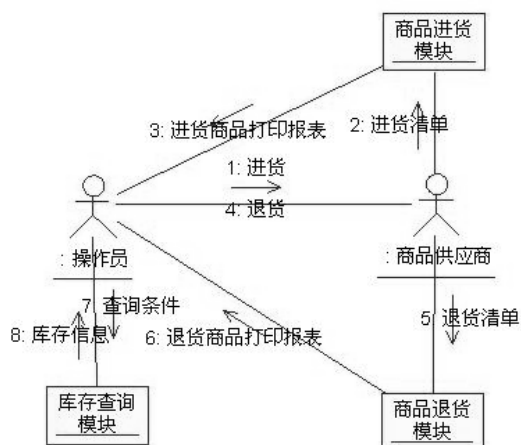


图 11-30 商品管理协作图

(3) 仓库历史记录查询协作图如图 11-31 所示。

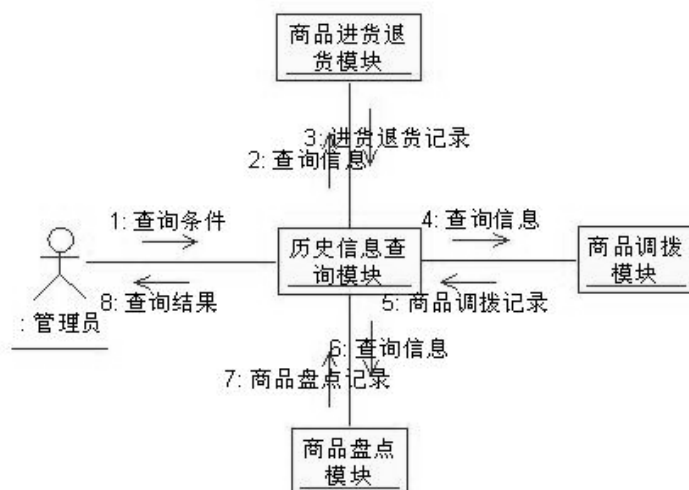


图 11-31 仓库历史记录查询协作图

协作图和时序图都可以表示各对象间的交互关系，但它们的侧重点不同。时序图用消息的几何排列关系来表达消息的时间顺序，各角色之间的相互关系是隐含的。协作图用各角色的几何排列图形来表示角色之间的关系，并用消息来说明这些关系。在实际中可以根据需要选用这两种图。

## 11.5 创建系统包图

包是模型的一部分，模型的每一部分必须属于某个包。建模者可以将模型的内容分配到包中。但是为了使其能够工作，分配必须遵循一些合理原则，如公用规则、紧密耦合的实现和公用观点等。UML 对如何组包并不强制使用什么规则，但是良好的解组会很大的增强模型的可维护性。

一个包可以包含其他包，根包间接的包含系统的整个模型。组织系统中的包有几种可能的方式，可以用视图、功能或建模者选择的其他基本原则来规划包。包是 UML 模型中一般的层次组织单元，他们可以被用来进行存储、访问控制、配置管理和构造可重用模型部件库。

如果包的规划比较合理，那么能够反映系统的高层框架——相关系统由子系统和它们之间的依赖关系组合而成。包之间的依赖关系概述了包的内容之间的依赖关系。

### 11.5.1 仓库管理系统包图

在定义具体的类之前，先在宏观的角度上将整个系统分割成多个独立的包。在这里把整个仓库管理系统分成的包如图 11-32 所示。

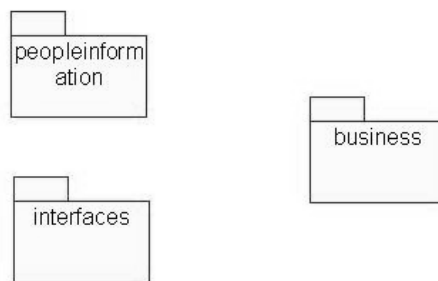


图 11-32 系统包图

整个系统可以看成人员信息（peopleinformation）、事务（business）和接口（interfaces）3 块，分别控制不同的应用。

### 11.5.2 人员信息（peopleinformation）包内的类

人员信息（peopleinformation）包内的类组织如图 11-33 所示。

在这里，仓库管理系统所涉及到的所有人员信息都包括在本包中，这样做的好处是仓库再添加新的人员时就不会影响到别的包。

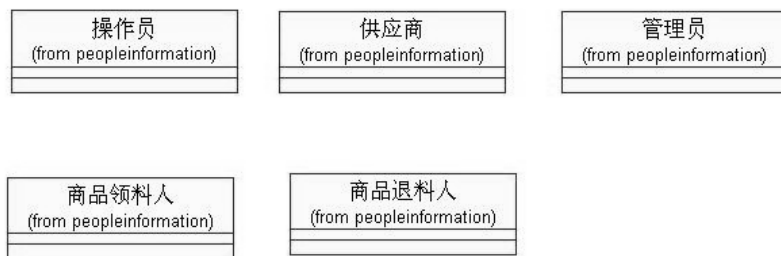


图 11-33 人员信息包内的类

在人员信息包内，有以下 5 块内容：

- 操作员；
- 供应商；
- 管理员；
- 商品领料人；
- 商品退料人。

### 11.5.3 事务包 (business) 包内的类

事务包 (business) 包内的类组织如图 11-34 所示。

仓库所有的事务都包含在本包中，如果仓库管理过程中需要增加某事务，那么只需要在本包中添加相应的类即可。

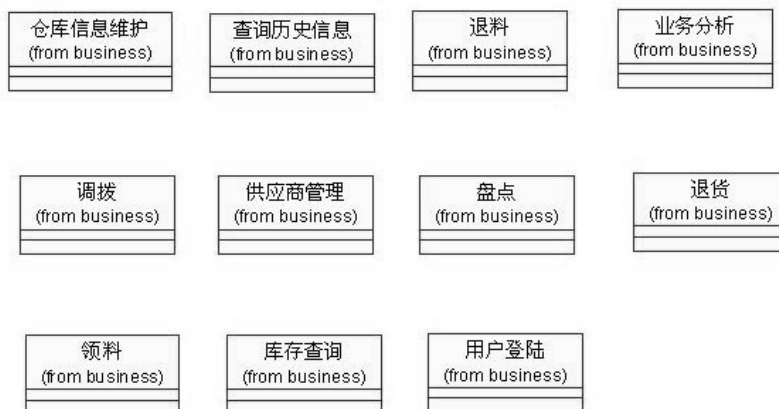


图 11-34 事务包内的类

### 11.5.4 接口包 (interfaces) 包内的类

接口 (interfaces) 包内的类组织如图 11-35 所示。

接口 (interfaces) 包内包括了所有的用户接口类，这样，当用户需要更改某界面或者是需要添加界面时就可以在本包中完成。

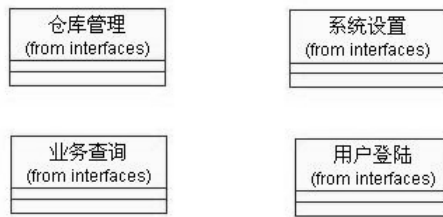


图 11-35 接口包内类

在接口包内，有以下 4 块内容：

- 仓库管理；
- 系统设置；
- 业务查询；
- 用户登录。

## 11.6 系统类模型

类图是面向对象系统的建模中最常见的图。类图显示了一组类、接口、协作以及他们之间的关系。

类图用于对系统静态设计视图建模。其大部分涉及到对系统的词汇建模、对协作建模或对模式建模。类图也是两个相关（组件图和配置图）的基础。

类图不仅对结构模型的可视化、详述和文档化很重要，而且对通过正向与逆向工程构造可执行的系统也很重要。

### 11.6.1 Logical 视图

Logical 视图关注的焦点是系统的逻辑结构。重复使用是一个主要目的。通过认真的指定类的信息和行为、组合类，以及检查类和包之间的关系，就可以确定可以重复使用的类和包。完成多个项目后，就可以将新类和包加进重复使用库中。今后的项目可以组装现有的类和包，而不必一切从头开始。

Logical 视图如图 11-36 所示。



图 11-36 Logical 视图

### 11.6.2 类图

类图中的类是针对时序图和协作图中每种对象创建的。如图 11-37 ~ 图 11-39 所示分别显示了人员信息包，接口包和事务包中类的类图。

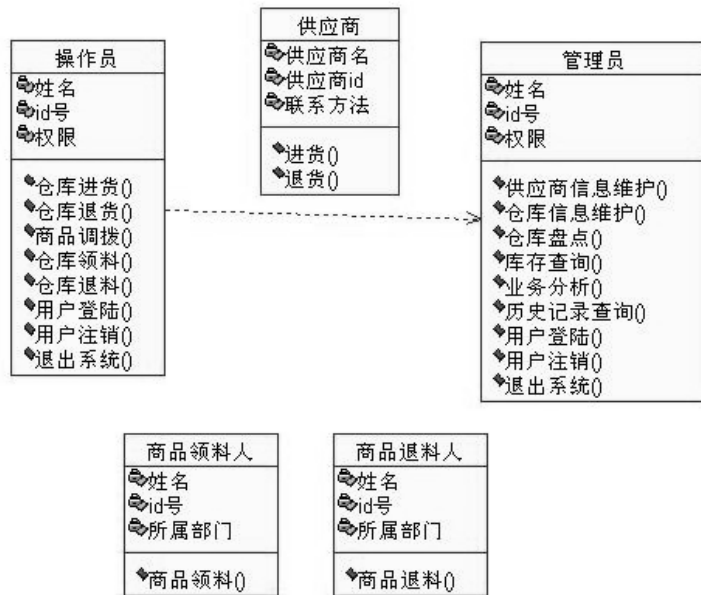


图 11-37 人员信息包内的类图



因为操作员的操作往往都是来自管理员的指令，可以理解成操作员的操作依赖于管理员，因此从操作员到管理员的虚线箭头，表示两者之间的依赖关系。除此之外，人员间没有明显的关系。

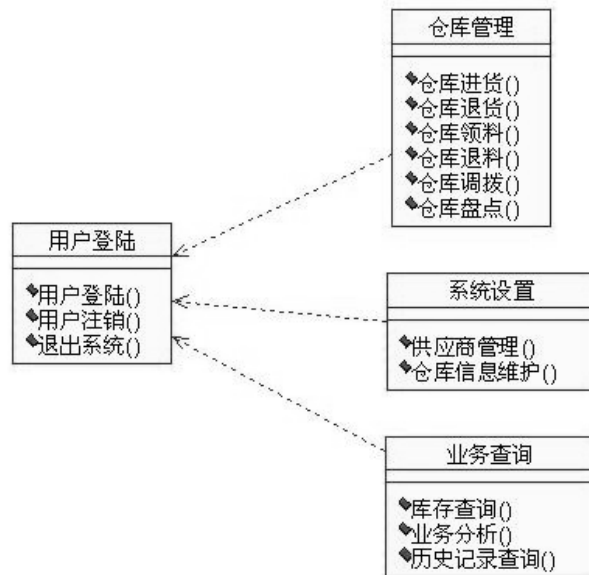


图 11-38 接口信息包内的类图

处于企业数据安全性方面的考虑，在仓库管理系统中，要进行仓库管理、系统设置和业务查询的操作都必须事先登录系统，因此在接口信息包内的类图中有由仓库管理、系统设置、业务查询到用户登录的虚线箭头，表示它们之间存在依赖关系。

仓库管理系统中的任何操作都必须在用户登录的前提下进行，因此在系统事务的类图中，所有事务都依赖与用户登录事务，它们以虚线箭头指向用户登录。仓库进货、退货、领料、退料、调拨和盘点都会影响到仓库中商品的库存，因此库存查询操作就依赖与上述的操作，他们之间也用虚线箭头相连。

开发人员用类图开发类。Rational Rose 可以产生类的框架代码，然后开发人员可以用所选语言来填充细节。分析人员用类图显示系统细节。如果需要相互通信的类之间没有建立联系，则可以马上在系统类图中看出。

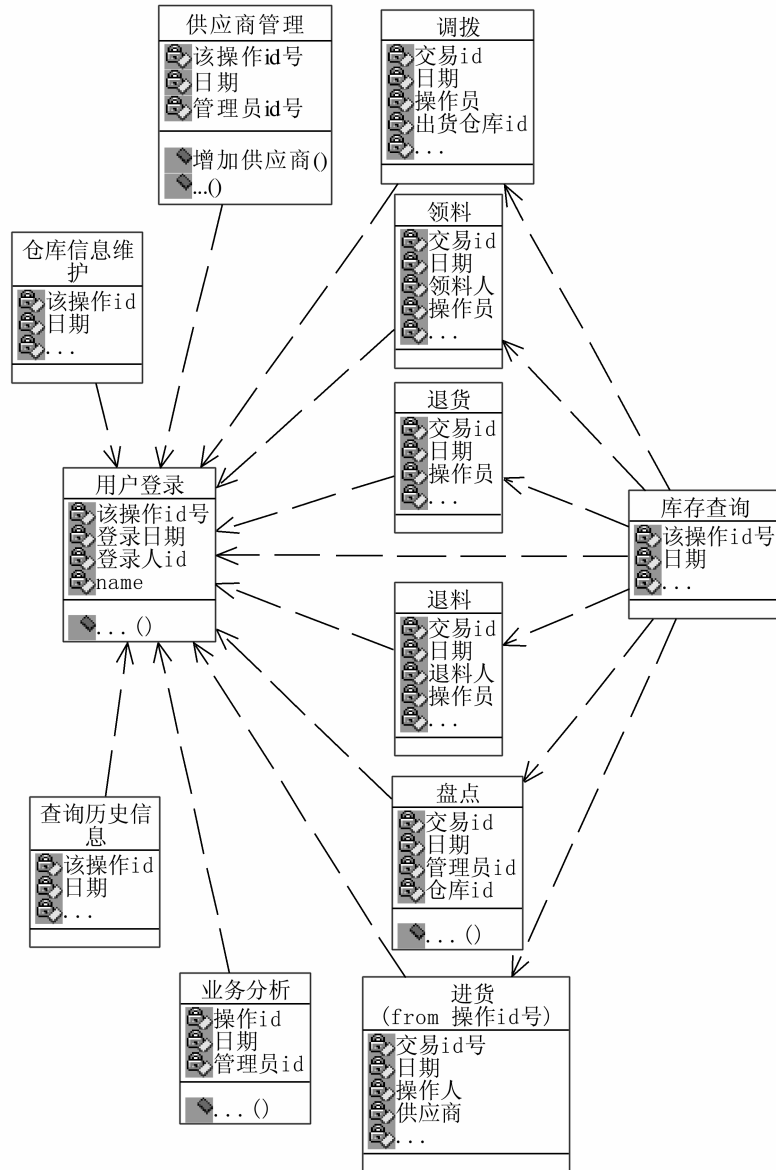


图 11-39 系统事务信息包内的类图

## 11.7 系统部署

仓库管理系统部署是整个项目实施过程中最后的阶段，就是把该系统中涉及到的硬件软件、整合到一起，并且可以让系统运行起来。

读者可以发现，在部署中有两个视图，组件图和配置图。这两个视图分别在下面介绍。

### 11.7.1 组件图

组件是系统中遵从一组接口且提供其实现的物理的、可替换的部分。组件的图形表示法是把组件画成带有两个标签的矩形。组件图包含了模型代码库、可执行文件、运行库和其他组件的信息。组件是代码的实际模块。

一个结构良好的组件，应该满足如下的要求：

- 提供系统物理方面抽取的一些事物的明确抽象；
- 提供对小组小的、定义完好的接口的实现；
- 经济有效的直接实现一组共同工作以完成这些接口语义的类；
- 相对其他组件是松散耦合的，通常对组件建模一般只涉及依赖关系和实现关系；

当在 UML 中绘制一个组件时，要遵循如下策略：

- 除非有必要显示的展示接口提供的操作，否则一般只需图符关系和实现关系；
- 仅显示在给定语境中对理解组件的含义是必要的那些接口；
- 当用组件为库和源代码建模时，显示与版本有关的标记值。

在 Rational Rose 中，仓库管理系统的组件（Component）视图显示如图 11-40 所示。

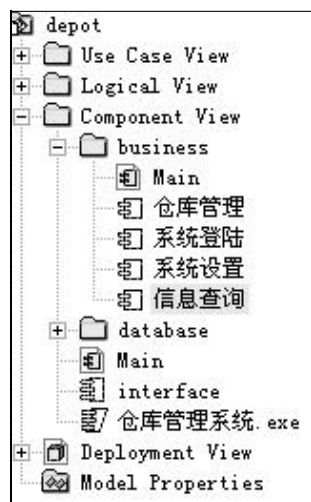


图 11-40 仓库管理系统的 Component 视图

如图 11-41 所示是仓库管理系统系统的 Component 视图。

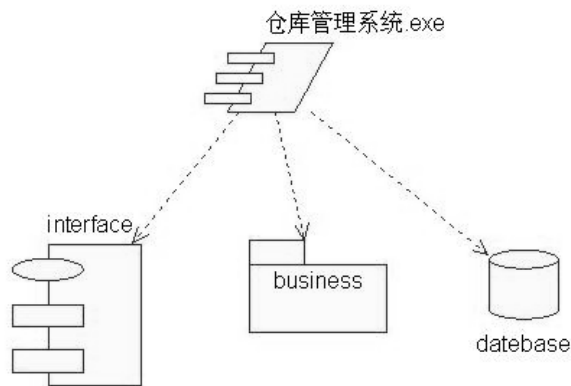


图 11-41 仓库管理系统系统的 Component 视图

此处已经把 Component 视图建立完毕。读者有兴趣可以深入分析某些细节，以提高自己的分析水平。

### 11.7.2 配置图

配置图 (Deployment) 视图考虑应用程序的物理部署，如网络布局和组件在网络上的位置等问题。进一步还需要考虑以下几个问题：具有多少网络带宽；希望出现多少并发用户；服务器关闭怎么办等。

仓库管理系统的 Deployment 视图如图 11-42 所示。



图 11-42 仓库管理系统的 Deployment 视图

如图 11-43 所示是仓库管理系统的 Deployment 框图。

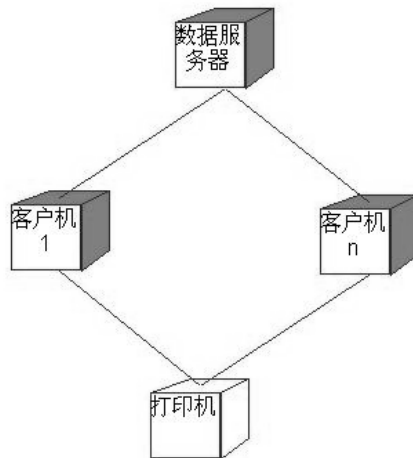


图 11-43 仓库管理系统的 Component 框图

在配置图中只画出了两个客户机，在实际的大型仓库中，应该不止两台客户机。上述配置图大致描述了整个系统的物理部署情况。

# 附录 A

## A.1 术语

本词汇表定义了用来描述统一建模语言 (UML) 的术语。除了 UML 特定的术语外,还包括 OMG 标准和面向对象分析设计方法中的相关术语。

### A.1.1 范围

本词汇表主要包括 UML 语义和 UML 记号法。另外,也包括如下一些使用到的词汇:

- (1) 对象管理结构的对象模型【OMA】;
- (2) CORBA 2.0【CORBA】;
- (3) 面向对象分析与设计 RFP-1【OA&D RFP】;
- (4) Rational Process (正由 Philippe Kruchten 等开发【RATLPROC】)。

【OMA】、【CORBA】和【OA&D RFP】在对 OMG 的适应程度和提供分布式对象的术语两方面对 UML 进行补充(当 OMG 的上述 3 个方面之间出现不一致时,将按照上面列出的顺序确定其权威性)。【RATLPROC】用来在提供关于体系结构和进程的术语方面对 UML 进行补充。

### A.1.2 部分术语

#### 1. 抽象 (abstraction)

抽象是对某事物本质特征行为的描述,这种行为使其能区别于别的事物。抽象往往依赖于观察者的视角,不同的观察角度导致不同的抽象。

抽象是一种在不同层次上同一概念的两种元素联系起来的依赖关系。

#### 语义

抽象的依赖关系是不同抽象层次上的两个元素间的关系。一般情况下,客户元素会描述的更详细些,但如果建模不明确,可以把两个元素都建模成客户。抽象依赖关系的构造型是跟踪 (trace)、精化 (refine)、实现 (realize) 和导出 (derive)。

<<trace>>声明不同模型中的元素之间存在的一些连接,一般提供者和客户是在不同的位置。例如提供者可以是类的分析视图,客户则可以是更详细的设计视图,系统分析师可以用 <<trace>>来描述它们之间的关系。

<<refine>>声明具有不同语义层次上的元素之间的映射。抽象依赖中的 <<trace>>是不同模型之间的纯粹历史关系,<<refine>>却是相同模型中元素间的依赖。例如在分析阶段遇到一个

类 Student，在设计时，这个类细化成更具体的类 Student。

`<<derive>>` 声明一个类可以从另一个类导出。当想要表示一个事物能从另一事物派生而来时，就使用这个依赖构造型。

### 表示法

抽象依赖关系用由客户元素指向提供者元素的箭头表示，一般还要附上关键字 `<<trace>>`、`<<refine>>` 或 `<<derive>>`。实现用指向提供者元素的三角形虚线箭头表示。

### 示例

如图 A-1 所示，类 Account 包含 Transactions 列表，每个 Transactions 包含资金的 Quantity，可以计算当前的资金余额，也就是把所有 Transactions 上的资金 Quantity 相加。类 Account 具有到 Quantity 的派生关联，Quantity 在与 Account 的关联中扮演 balance 的角色。

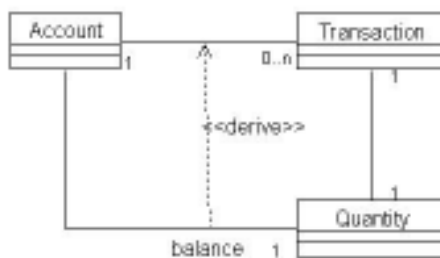


图 A-1 抽象依赖

## 2. 抽象类 (abstract class)

抽象类是可能不会被实例化的类。

### 语义

如果抽象类是不可被实例化的类，即它没有直接的实例，则既可能是因为它描述是不完整的（如缺少一个或多个操作的方法），也可能是因为即使它的描述是完整的，它也不想被实例化。抽象类必须有可能含有实例的后代才能使用，一个抽象的叶类是没用的（它可以作为叶在框架中出现，但是最终它必须被说明）。

具体类可以没有抽象操作，如果某类只包含抽象操作，那该类一定是抽象类。具体操作是可以被实现一次并在所有子类中不变地使用的操作。继承的目的之一是将这些操作在抽象的超类中分解，以使得它们可以被所有的子类分享，抽象类可以有具体操作。如果一个操作在类中被声明为是抽象的，那么该操作缺少在类中的实现，且类本身也必须是抽象的，操作的实现必须由具体的后代来满足。

### 表示法

抽象类的名字用斜体表示。在白板或纸张上手工画 UML 草图时，很难区分是否是斜体，因此，一些人建议在这些场合可以在类名的右下角加上 {abstract} 标记以示区别。标准的 UML 中，关键字 abstract 一般放置在名字下面或后面的特征表中，例如 Account {abstract}。

### 示例

如图 A-2 所示表示了抽象类的应用。其中的文本编辑器是独立于平台的，为此定义了一个独立于平台的窗口对象类 Window，它是一个抽象类，抽象类的名字用斜体表示。类 Window 包含有两个方法的名称 toFront() 和 toBack()，但是没有方法体。类 Window 本身不能有实例，

但它有两个特化的子类 Windows Window 和 MAC Window ,它们包含了操作 toFront()和 toBack() 分别在微软的 Windows 操作平台和 Apple 的 Macintosh 操作平台上的实现。在本例中 ,对象类 Window 的作用是作为文本编辑器类 TextEditor 的一个接口。

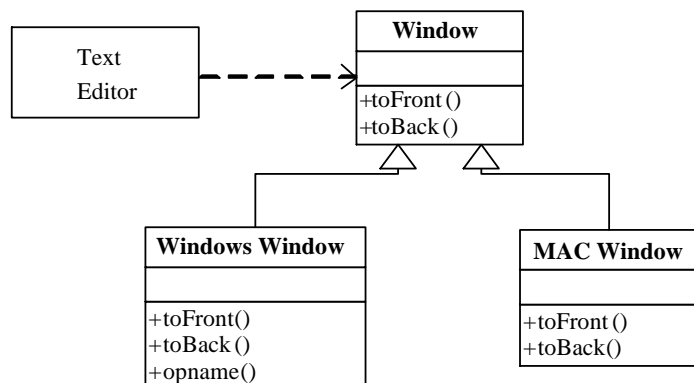


图 A-2 抽象类的应用

### 3. 抽象操作 ( abstract operation )

抽象操作是指该操作没有实现 ,即它只有说明而没有方法。抽象操作的实现必须被具体后代类补充。

#### 语义

如果一个类中的操作是被声明为抽象的 ,那么该操作就只有说明而没有方法 ,且这个类也必须是抽象的。抽象操作的实现必须被具体后代类补充。如果一个操作在类中被声明是具体的 ,那么类必须满足或继承从祖先那里得到的实现 ,这个实现可能是一个方法或调用事件。如果类继承了一个操作的实现但是将操作声明是抽象的 ,那么抽象的声明就使类中被继承的方法无效。如果操作在类中根本没有被声明 ,那么类继承从它的祖先那里得到的操作声明和实现。

#### 表示法

抽象操作的名字用斜体来表示 ,和抽象类的表示方法类似。

#### 示例

如图 A-3 所示 ,抽象类 Color 具有两个操作 ,它们分别是抽象操作 draw 和具体操作 change ,其中抽象操作用斜体字表示。

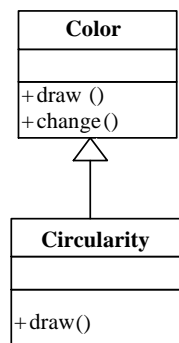


图 A-3 抽象操作



#### 4. 访问 (access)

访问是一种授权依赖关系，它允许一个包引用另外一个包中的元素。

##### 语义

<<access>>是包间的依赖关系。包用在 UML 中来分组物件。访问依赖<<access>>允许一个包访问另一个包的所有公共内容。然而包定义的命名空间是相互独立的，也就是说，当客户包中的物件要引用提供者包中的物件时，必须使用路径名。

##### 表示法

访问依赖关系用一条虚线箭头表示，该虚线箭头从客户包指向提供者包。箭头用关键字<<access>>作为标号。

##### 示例

如图 A-4 所示。包 package1 和包 package2 通过<<access>>依赖相互关联。<<access>>允许包 package1 中的元素访问 package2 中的公共元素，但不合并这两个包的名字空间。因此，需要在 package1 中引用类 B，这里必须使用类 B 的路径名。

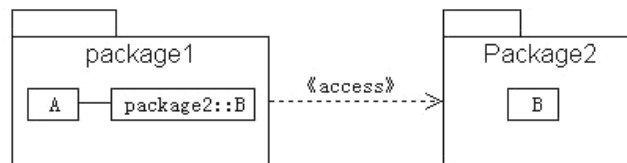


图 A-4 访问依赖

#### 5. 动作 (action)

动作是定义可执行的语句或计算程序的类。

##### 语义

动作是一个可执行的原子计算，它可以包括操作调用、另一个对象的创建或撤销、向一个对象发送信号。动作也可以是一个动作序列，即包括一序列的简单动作。动作或动作序列的执行不会被同时发生的其他动作所影响。

根据 UML 的概念，动作的执行时间是非常短的，与外界的时间相比几乎可以忽略，这里所指的时间非常短也就是说动作使得系统的反应时间不会被减少，系统中可以同时执行几个动作，但是动作的执行应该是独立的，因此在动作执行过程中不允许被中断，这点正好与活动相反，活动是可以被其他事件中中断的。在某动作执行时，一般新进的事件会被安排在一个等待队列里。

动作也可以附属于转换，当转换被激发时动作被执行。它们也可以作为状态的入口动作和出口动作出现。这些动作由进入或离开状态的转换触发。所有动作都是原子的，即它们执行时完全不会被别的动作所干扰。

##### 结构

动作包括一个目标对象集合、一个对将要被发送的信号或将要被执行的动作的引用（即请求）、一张参量值表和一个可选的用于指明迭代的递归表达式。

(1) 对象集合。对象集合表达式产生一个对象的集合。在许多情况下，这个集合包含一个

独立的固定的对象。给定参量表的消息拷贝被同时发送到集合中的每个对象，即广播到每个对象。每个目标都是独立接收和处理消息的单独的实例。如果集合是空的，那么什么都不会发生。

(2) 请求。指明一个信号或声明一个操作。信号被发送到对象，操作被调用（对于具有返回值的操作，对象集合必须包含一个对象）。

(3) 参量表和参量列表。当赋值的时候，参量表中的值必须与信号或操作的参数相一致。参量被作为发送或调用的一部分。

(4) 再发生。一个迭代表达式，说明需要执行多少次动作，并指定迭代变量（可选）。这个表达式也可以描述一个条件动作（即进行一次或不进行）。

以下介绍几种动作。

(1) 赋值动作。赋值动作将一个对象的属性值设置为给定值。该动作包含一个对目标对象的表达式、对象属性的名字和一个被分配到对象内属性槽的值的表达式。

(2) 调用动作。调用动作导致一个对象上操作的发生，即该对象上操作的调用。该动作包含一个消息名、一张参量表达式的表和一个目标对象集合表达式。目标可能是一个对象集合。在这种情况下，调用同时发生并且操作不会有返回值。如果操作有了返回值，那么它必须有一个对象作为目标。调用动作是同步的。调用者再次接收控制之前等待被调用操作的完成。如果操作被作为调用事件实现，那么调用者再次接收控制之前一直等待，直到接收者执行被调用触发的转换。如果操作的执行返回了值，那么调用者在它再次接收到控制时接受了这些值。

(3) 创建动作。创建动作导致了对象的实例化和初始化。该动作包含一个对类的引用和一个可选的带有参量表的类作用域操作。动作的执行创建了一个类的新实例，其属性值从计算它们的初始值表达式中得到。如果一个明确的创建动作被给定，那么它将被执行。操作常常会用创建动作的参量值覆盖属性值的初始值。

(4) 销毁动作。销毁动作导致目标对象的销毁，该动作有一个针对对象的表达式。销毁动作没有其他的参量。执行该动作的结果是销毁对象及到它的所有链接及组成部分。

(5) 返回动作。返回动作导致了一个到操作调用者的控制转换。该动作只允许在被调用的操作中存在。该动作包含一个可选的返回值表，当调用者接收到控制时，该表对调用者是有效的。如果包含的操作被异步使用，那么调用者必须明确地选择返回消息（作为一个信号），否则它将会遗失。

(6) 发送动作。发送动作创建了一个信号实例，并且用通过计算动作中的参量表达式得到的自变量初始化这个信号实例。信号被送到对象集合里，这些对象通过计算动作中的目标表达式而得到。每一个对象接收它自己的信号拷贝。发送者保持其控制线程和收益，且发送信号是异步的。该动作包含信号的名称、一张信号参量表达式表和对目标对象的对象集合表达式。

(7) 如果对象集合被遗漏，那么信号被发送到由信号和系统配置决定的一个或多个对象。例如，一个异常被发送到由系统策略决定的包含作用域。

(8) 终止动作。终止动作引起某种对象的销毁，这个对象拥有包含该动作的状态机，即该动作是一种“自杀”行为。其他对象会对对象的销毁事件做出反应。

(9) 无解释动作。这是一种控制构造的动作。

### 表示法

UML 没有一种固定的动作语言，它希望建模者使用一种实际的编程语言去编写动作。面对 OCL 的改编是为了写动作伪代码，但这并不是标准的一部分。

赋值动作：

```
target := expression
```

调用动作：

```
object-set.operation-name (argument list,)
```

创建动作：

```
new class-name (argument list,)
```

销毁动作：

```
object.destroy()
```

返回动作：

```
return expression list,
```

发送动作：

```
object-set.signa-name (argument list,)
```

终止动作：

```
terminate
```

无解释动作：

```
if (expression) then (action) else (action)
```

如果需要明确区别调用与发送，关键字 call 和 send 可以作为表达式的前缀，它们是可选的。

## 6. 活动 (activity)

活动是状态机内正在进行的非原子执行。

### 语义

活动是状态机内子结构的执行，子结构允许中断点的存在。活动并不由内部转换的激发终止，因为并没有状态的改变。内部转换的动作会明确地终止活动。

动作和活动的区别：活动是在状态机中一个非原子的执行，它由一系列的动作组成，动作由可执行的原子计算组成，这些计算能够使系统的状态发生变化或返回一个值。

活动也是一种计算，但是它可以有内部结构并且会被外部事件的转换中断，所以活动只能附属于状态中，不能附属于转换。与动作不同，虽然活动自己也能中断，但是如果外界不中断它，它可以无限期地持续下去。而动作则不能从外部中断并且可以附属于转换或状态的入口或出口，而非状态本身。

活动可以由嵌套状态、子机引用或活动表达式建模。

### 示例

如图 A-5 所示是一个接听电话的过程，它说明了动作和活动的区别。当事件 detect be called (检测到被呼叫) 发生时，系统激发一个转换。作为转换的一部分，动作 show number (显示来电号码) 发生。它是一个动作，所以通常是原子的。当动作被执行时，没有事件会被接受。当动作被执行后，系统进入 Ringing 状态。当系统处于这个状态时，它执行 ring (振铃) 活动。活动需要时间来完成，只有用户接听来电或者拒绝来电，该活动才会停止。当 finish (完成呼叫) 事件发生时，转换被激发并将系统返回到 Waiting 状态。当 Ringing 状态不再是活动的，它的活动 ring 也被终止了。

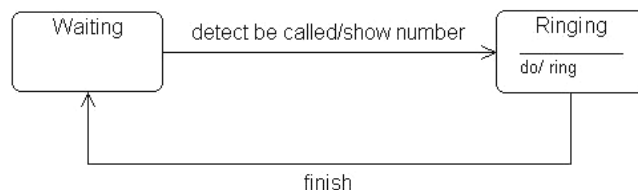


图 A-5 动作和活动

## 7. 活动图 ( activity diagram )

活动图是 UML 中描述系统动态行为的图之一，它用于展现参与行为的类的活动或动作。

### 语义

状态机是展示状态与状态转换的图。通常一个状态机依附于一个类，并且描述这个类的实例对接受到的事物的反应。状态机有两种可视化方式，它们分别为：状态图和活动图。活动图被设计用于描述一个过程或操作的工作步骤，从这方面理解，它可以算是状态的一种扩展方式。其实质性区别在于，活动图描述的是响应内部处理的对象类的行为，状态图描述的是对象类响应事件的外部行为。活动图着重表现的是从一个活动到另一个活动的控制流，是内部处理驱动的流程。而状态图着重表现的是从一个状态到另一个状态的流程，常用于有异步事件发生的情形。

从一定的意义上说，活动图是一种特殊的状态图。如果在一个状态图中的大多数的状态是表示操作的活动，而转移则是由状态中的动作的完成来触发的，即全部或绝大多数的事件是内部产生的动作完成的，这就是活动图。因此，活动图的许多术语和状态图是相似的。活动图和状态图都用于为一个对象（或模型元素）的生命周期中的行为建立模型。

### 表示法

UML 中，活动图里的活动用圆角矩形表示。一个活动结束自动引发下一个活动，则两个活动之间用带箭头的连线相连接，连线的箭头指向下一个活动。活动图的起点也是用实心圆表示，终点用半实心圆表示。

### 示例

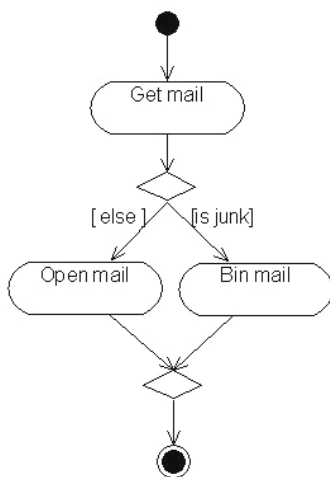


图 A-6 活动图

如图 A-6 所示，初始状态变迁到动作状态 Get mail，当该状态完成工作时，变迁到分支，该分支有两种可能的变迁。如果邮件是邮件宣传品，那么[is junk]计算为真，从而触发右边的变迁，进入状态 Bin mail，最后到终止状态。如果触发的是左边的变迁，进入 Open mail，最后到终止状态。

#### 8. 活动状态 (activity state)

活动状态是具有内部计算和至少一个输出完成转换的状态。

##### 语义

对象的活动状态可以被理解成一个组合，它的控制流由其他活动状态或动作状态组成。因此活动状态的特点是：它可以被分解成其他子活动或动作状态，它能够被中断，它占有有限的事件。动作状态可以被理解成原子的活动状态，即当它活动时不会被转换中断。动作状态可以被建模成只有一个入口动作的活动状态。

动作状态和活动状态仅仅是状态机中状态的特殊种类。当系统进入一个动作或活动状态时，系统是简单的执行该动作或活动；当结束一个动作或活动时，控制权就传递给下一个动作或活动。因此，活动状态有点类似于速记。一个活动状态在语义上等同于在适当地方展开它的活动图，直到图中仅看到动作为止。

从程序设计的角度来理解，活动状态对应于软件对象的实现过程中的一个子过程。

##### 表示法

动作状态和活动状态的图标没有什么区别，都是圆端的方框。只是活动状态可以有附加的部分，如可以指定入口动作、出口动作、状态动作以及内嵌状态机，如图 A-7 所示。



图 A-7 活动状态

#### 9. 参与者 (actor)

参与者是系统外部的一个实体（可以是任何的事物或人），它以某种方式参与了用例的执行过程。

##### 语义

参与者是用户作用于系统的一个角色 (Role)。参与者有自己的目标，通过与系统的交互达到目标。参与者用来建立一个系统的外部用户模型，参与者直接与系统交互作用。参与者是对系统边界之外的对象的描述，在系统边界之外的是参与者。参与者对系统的交互包括信息交换（数据信息和控制信息）和与系统的协作。

参与者包括人参与者 (Human Actor) 和外部系统参与者 (System Actor)。系统的用户是人参与者，用户通过与系统的交互，操纵系统，完成所需要的工作。参与者不一定是人，它可以是一个外部系统，该系统与本系统相互作用，交换信息外部系统可以是软件系统，也可以是个硬设备，例如在实时监控系统中的数据采集器，自动化生产系统上的数控机床等。

在建模参与者过程中，记住以下要点。

- (1) 参与者对于系统而言总是外部的，因此它们在你的控制之外。
- (2) 参与者直接同系统交互，这可以帮助定义系统边界。

(3) 参与者表示人和事物同系统发生交互时所扮演的角色，而不是特定的人或特定的事物。

(4) 一个人或事物在与系统发生交互时，可以同时或不同时扮演多个角色。例如某研究生担任某教授的助教，同职业的角度看，他扮演了两个角色学生和助教。

(5) 每一个参与者需要有一个具有业务一样的名字，在建模中，不推荐使用诸如 NewActor 这样的名字。

(6) 每个参与者必须有简短的描述，从业务角度描述参与者是什么。

(7) 像类一样，参与者可以具有分栏，表示参与者属性和它可接受的事件。一般情况下，这种分栏使用的并不多，很少显示在用例图中。

### 表示法

在图形上，参与者用人形图符表示，参与者的名字在图的下面，如图 A-8 所示。



图 A-8 参与者

## 10. 关联 (association)

关联是描述一组具有共同结构特征、行为特征、关系和语义的链接。

### 语义

关联是一种结构关系，它指明一个事物的对象与另一个事物的对象间的联系。也就是说，如果两事物间存在链接，这些事物的类间必定存在着关联关系，因为链接是关联的实例，就如同对象是类的实例一样。举例来说，学生在大学里学习，大学又包括许多的学院，我们可以清楚的了解在学生、学院和大学间存在着某种链接。在 UML 建模设计类图时，就可以在学生 (Student)、学院 (Institute) 和大学 (University) 3 个类间之间建立关联关系。

关联可以有个名称，用于描述该关系的性质。此关联名应该是动词短语，因为它表明源对象正在目标对象上执行动作。名称也可以用一个指引阅读方向的小黑箭头，为的是消除名称含义上可能存在的歧义。但关联的名称并不是必需的，当要明确的给关联提供角色名或当一个模型存在许多关联且要对这些关联进行查阅和区别时才会给出关联名，如图 A-9 所示。

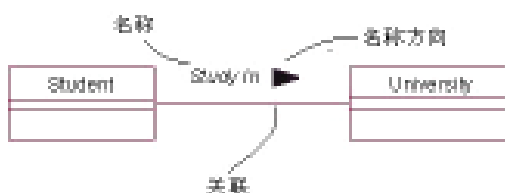


图 A-9 关联名称

对于关联可以加上一些约束，以规定关联的含义。约束的字符串括在花括号 {} 内。UML 定义了 6 种约束可以施加在目标关联端点上，这 6 种约束分别如下。

(1) implicit：规定该关联只是概念性的，在模型的箱化中不会再用。

(2) ordered：规定一个具有多重性大于一的关联的一端对象是有序的。

(3) changeable：规定被关联的对象之间的链接 (Link) 是可变的，可以被任意添加、删除和改变。

(4) addonly：规定可以在任何时间从源对象添加新的链接。

(5) frozen：规定当源对象已经创建和初始化后，冻结它的链接，不能再添加、修改或删除它的链接。

(6) xor：使用了约束 xor 的关联称为 xor 关联，它代表一组关联的互斥的情况。xor 关联规定在某一时刻，对于每一个被关联的对象，只有其中的一个关联的实例是当前的关联实例。xor 约束用关联之间的虚线加 “( xor )” 表示。例如在图 A-10 中，对象类 Account ( 账户 ) 与对象类 individual ( 个人 ) 的关联、对象类 Account 与对象类 corporation ( 企业 ) 的关联之间存在约束 xor，它表示一个 Account 不是属于 individual 就是属于 corporation，二者必居其一。

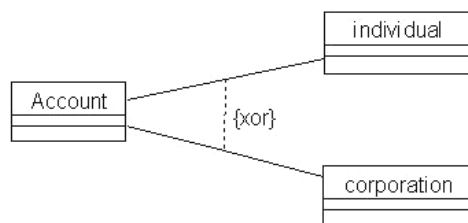


图 A-10 xor 关联

## 11. 关联类 ( association class )

关联类既是关联又是类。

### 语义

关联类有着关联和类的特性，它将多个类连接起来又有着属性和操作。可以这样理解，关联类是具有类属性的关联，或具有关联属性的类。

关联类的名称可以置于关联路径上和关联类符号中。当同时出现在这两个位置上时，关联类名称必须相同。关联类的名称可置于关联路径上，并从关联类符号中省略。这种做法一般用于具有属性、但没有操作或其他关联的关联类，它强调的是关联的关联性本质或关联在联系其他类时扮演的角色。关联类的名称还可以置于类符号中，并从关联路径上省略。这种做法一般用于具有操作或其他关联的关联类，它强调关联的类本质或者关联作为类所扮演的角色。

关联类可以有操作，这些操作可以改变连接的属性或者增加连接本身和移走它。因为关联类也是一个类，它也可以参与到关联本身。

关联类可以不把自己作为参与类中的一个 ( 虽然有些人能够确切地找到这种递归结构的意义 )。

### 表示法

关联类用类的符号表示，由一条虚线与关联路径连接，如图 A-11 所示。

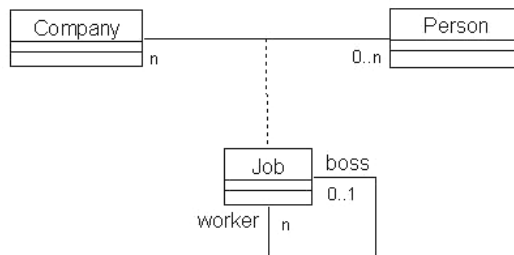


图 A-11 关联类

## 12. 关联端点 (association end)

关联包含一张有多个关联端点的有序表(也就是说这些端点是可以被区分的并且是不可替换的)。每个关联端点定义了在该关联中给定位置的一个类(角色)的参与。关联端点是关联的一个结构部分,它定义了在该关联中类的参与。在同一个关联中一个类可以连接到多个端点。

### 语义

关联端点保持一个目标类元的引用。它定义了在该关联中类元的参与。每个关联端点指定了应用于对应对象的参与特性,例如在关联中一个独立的对象在链接中会出现多少次(多重性)。某些特性,如导航性只应用于二元关联,但是多数可以应用于  $n$  元关联。

一个关联端点有如下的特性。

**聚集 (aggregation):** 聚合关系是一种特殊的关联关系,它表示类间的关系是整体与部分的关系。更简单的说,关联关系中一个类描述了一个较大的事物,它由较小的事物组成,这种关系就是聚合,它描述了“has-a”的关系,即整体对象拥有部分对象。

**可修改性 (changeability):** 判定与对象有关的连接集合是否可修改,有枚举值{changeable, frozen, addOnly}。缺省情况下是 changeable。

**接口说明 (interface specifier):** 相关对象及类元的说明类型的可选的约束(某些人称之为角色)。

**多重性 (multiplicity):** 定义了一个类型 A 的实例在一段特定的时间里能够和多个类型 B 的实例发生关联。

**导航性 (navigability):** 导航性表示可从源类的任何对象到目标类的一个或多个对象遍历。也就是说给定源类的一个对象,可以达到目标类的所有对象。

**定序 (ordering):** 判定一组不相关对象是否是有序的,枚举值有{unordered, ordered}。出于设计的目的,sorted 值也可以被用到。

**限定符 (qualifier):** 寻找与关联相关的对象的选择器的属性表。

**角色名 (rolename):** 关联端点的名称,一个标识符字符串。该名称标识关联内对应类的特定角色。角色名在关联中以及在源类的直接和继承的伪属性(属性及类可见的其他角色名)中必须是唯一的。

**目标范围 (target scope):** 判定链与对象或整个类是否相关,枚举值有{instance, classifier},缺省为 instance。

**可见性 (visibility):** 连接是否可访问类而不能访问关联中相反的一端。可见性位于连接到目标类的一端。转换的每个方向有它自己的可见性值。



## 表示法

关联路径的端点连接到对应类符号的矩形边缘上。关联端点特性表示为在路径端点上或旁边的符号，这条路径连接到一个类元符号。下面是对每个特性的符号的简要的总结。

聚集 (aggregation) : 位于聚集端的一个空的菱形，对于组成来说是一个实菱形。

可修改性 (changeability) : 目标端的文字属性 {frozen} 或 {addOnly} , 通常省略 {changeable}。

接口说明 (interface specifier) : 角色名上的名字后缀，形式是 typename。

多重性 (multiplicity) : 靠近路径端点的文字标记，形式是 min..max。

导航性 (navigability) : 路径端点上的箭头表示了导航的方向。如果两个端点都没有箭头，就是假设关联在两个方向上都是可导航的。

定序 (ordering) : 靠近目标端点的文字属性 {ordered}，有目标类实例的有序表。

限定符 (qualifier) : 路径端点和源类之间的小矩形。矩形内包含了一个或多个关联的属性。

角色名 (rolename) : 靠近目标端的一个名字标签。

目标范围 (target scope) : 类作用域角色名是加下划线的，除非是实例范围。

可见性 (visibility) : 可见性符号 {+ # -} 位于角色名前。

如果在一个独立的角色上有多个符号，那么从连接类的路径的一端读起。

### 13. 属性 (attribute)

类的属性描述了为类的所有对象所共有的特征。

#### 语义

类的属性是类的一个组成部分，它描述了类在软件系统中代表的事物所具备的特性。类可以有任意数目的属性，也可以没有属性。属性描述了正在建模的事物的一些特性，这些特性是所有对象所共有的。

类的属性的选取可以参考以下原则：

- (1) 原则上，类的属性应能描述并区分每个特定的对象；
- (2) 只有与系统有关的特征才包含在类的属性中；
- (3) 系统建模的目的也会影响到属性的选取，因此，相同的事物在不同的系统中可能具有不同的属性。

在 UML 中类属性的语法为：

[可见性] 属性名 [ : 类型] [= 初始值] [{属性字符串}]

其中 [ ] 中的部分是可选的。类中的属性的可见性主要包括 public、private 和 protected 3 种，它们分别用 “+”、“-”、“#” 来表示。

根据定义，类的属性首先是类的一部分，并且每个属性都必须有一个名字以区别于类的其他属性，通常情况下属性名由描述所属类的特性的短名词或名词短语构成（通常以小写字母开头）。类的属性还有取值范围，因此还需为属性指定数据类型。例如布尔类型的属性可以取两个值 TRUE 和 FALSE。当一个类的属性被完备的定义后，它的任何一个对象的状态都由这些属性的特定值所决定。

### 14. 行为 (behavior)

一个操作或一个事件的可见的影响，包括结果。

## 15. 行为特征 (behavioral feature)

表示动态行为的模型元素,比如操作或方法,它可以是类元的一部分。一个类元处理一个信号的声明也是一个行为特征。

**标准元素**

创建 (create), 销毁 (destroy) 和叶 (leaf)。

## 16. 调用 (call)

用于激活一个操作。

**语义**

调用是在一个过程的执行点上激发一个操作。它将一个控制线程暂时从调用过程转换到被调用过程。调用使用依赖关系建模一种情况,在这个情况中一个客户类的操作 (或操作本身) 调用提供者类的操作 (或操作本身)。它用<<call>>构造型表示。

**表示法**

在时序图或协作图上,调用表示成指向目标对象或类的文字消息。调用依赖关系表示从调用者指向调用类或操作的含有构造型<<call>>虚线箭头。在编程语言中,大多数的调用可以表示成文本过程的一部分。这种类型的依赖在 UML 建模中趋于不被广泛使用,它适用于更深的建模层次。同样,现在也很少有 CASE 工具支持操作之间的依赖。

## 17. 规范表示法 (canonical notation)

UML 定义了规范表示法,它用单色线和文字表示模型。这就是 UML 模型的标准“出版格式”,可用于印刷图。

图形编辑工具可以扩展规范表示法并且提供交互能力。交互显示减少了模棱两可的弊端。所以,UML 标准的焦点是印刷的规范形式,一个交互工具可以而且应该提供交互扩展。

## 18. 类 (class)

类是一种重要的分类器 (Classifier),分类器主要用来描述结构和行为特性的机制,它包括类、接口、数据类型、信号、组件、节点、用例和子系统。类是任何面向对象系统中最重要构造块。

**语义**

类是对一组具有相同属性、操作、关系和语义的对象的描述。这些对象包括现实世界中的软件事物和硬件事物,甚至也可以包括纯粹概念性的事物,它们是类的实例。一个类可以实现一个或多个接口。结构良好的类具有清晰的边界,并成为系统中职责均衡分布的一部分。

类定义了一组有着状态和行为对象。属性和关联用来描述状态。属性通常用没有身份的纯数据值表示,如数字和字符串。关联则用有身份的对象之间的关系表示。个体行为由操作来描述,方法是操作的实现。对象的生命期由附加给类的状态机来描述。类的表示法是一个矩形,由带有类名、属性和操作的分格框组成。

一组类可以用泛化关系和建立在其内的继承机制分享公用的状态和行为描述。泛化使更具体的类 (子类) 与含有几个子类共同特性的更普通的类 (超类) 联系起来。一个类可以有零个或多个父类 (超类) 和零个或多个后代 (子类)。一个类从它的双亲和祖先那里继承状态和行为描述,并且定义它的后代所继承的状态和行为描述。

### （1）边界类

边界类处理系统环境和系统内部之间的通信，边界类为用户或另一个系统提供了一个接口。边界类组成了系统中依赖于环境的部分，边界类用于为系统的接口建模，边界类代表了系统和系统外的一些实体之间的接口。它是系统与外界交换信息的媒介，并将系统与系统环境中的变化隔离开来。

### （2）实体类

实体类是模拟必须被储存的信息和关联行为的类。实体对象是实体类的实例，被用来保存或更新关于某现象（例如某事件、某对象）的信息，它们通常是永久性的（persistent）。实体类通常是独立于它们的环境，也就是说，实体类对于系统环境如何与系统通信是不敏感的。大多数情况下，它们是独立于应用程序的，也就是说它们可以被用于多个应用程序。实体类通常是那些系统用来完成某些责任的类。

### （3）控制类

控制类是用来为特定一个或几个用例的控制行为建模的类。控制对象是控制类的实例，它通常控制其他对象，所以控制对象的行为是协调类型的，控制类协调实现用例的规定行为所需要的事件。控制类封装了特定于用例的行为，控制类通常是依赖于应用程序的类。

### （4）参数类

参数类又称为模板类，模板类定义了类族。模板包含类槽、对象槽和值槽，这些槽可以充当模板的参数。模板不能直接使用，使用前必须实例化模板类，实例化包括将这些形式的模板参数绑定到实际的参数。

## 命名

类的名称是每个类所必需的构成，用于和其他类相区分。名称（name）是一个文本串，可分为简单名称和路径名称。单独的名称，即不包含冒号的字符串叫做简单名（single name）；用类所在的包的名称作为前缀的类名叫做路径名（path name），如图 A-12 所示。

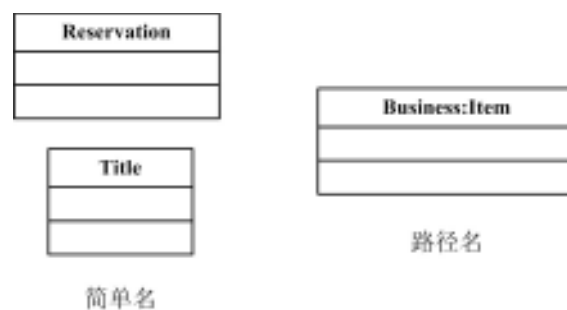


图 A-12 类的命名

类也是一个命名空间，并为嵌套类元声明建立了作用域。嵌套类元并不是类的实例的结构部分。在类对象与嵌套类对象之间并没有数据联系。嵌套类是一个可能被外层类的方法所使用的类的声明。在类内的类声明是私有的，除非清楚的设定为可见，否则在该类的外部是不可访问的。没有可见的标识符来表示嵌套类的声明，只有在利用超级链接工具时，才有可能对它们进行访问。嵌套的名字必须用路径名来指定。

## 表示法

类在 UML 中由专门的图符表达，它是一个分成 3 个分隔区的长方行。其中顶端的分隔区为类的名字，中间的分隔区存放类的属性、属性的类型和值（在 UML 符号表示中给出类的初始值），第 3 个分隔区存放操作、操作的参数表和返回类型，如图 A-13 所示。



图 A-13 类的表示法

在给出类的 UML 表示时，可以根据建模的实际情况来选择隐藏属性区或操作区或者两者都隐藏。例如图 A-14 和图 A-15 所示表示了图书馆信息管理系统中借书信息类，但图 A-15 中左边类的操作的分隔区为空，这并不代表没有操作，只是因为没有显示，右边类则隐藏了属性分隔区。



图 A-14 类

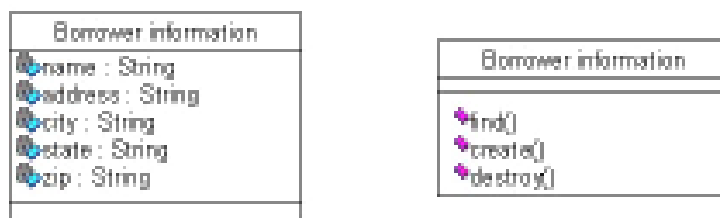


图 A-15 类的省略表示法

## 类表示格式指导

用正常字体在类名字的上方书写构造类型。

用黑体字居中或者左对齐书写类名。

用正常字体，左对齐书写属性和操作。

用斜体字书写抽象类的名字，或者抽象操作的符号。

在需要处表示类的属性和操作格（在一套图中至少出现一次），在其他上下文或提及处可以隐藏它们。最好为一套图中的每个类定义一个“原始位置（home）”，并在此处对类进

行详细描述。在其他位置，可以使用类的最小化表示形式。

### 19. 类图 (class diagram)

类图是描述类、接口、协作以及它们之间关系的图。它是系统中静态视图的一部分，静态视图可以包括许多的类图。

#### 语义

类图是面向对象系统建模中最常用的图，它是定义其他图的基础，在类图的基础上，状态图、协作图、组件图和配置图等进一步描述系统的其他方面的特性。

类图的用途如下。

#### (1) 对系统的词汇建模

在用 UML 构建系统的开始通常是从构造系统的基本词汇开始，用于描述系统的边界，也就是说用来决定哪些抽象是要建模系统中的一部分，哪些抽象是处于要建模系统之外。这是非常主要的一项工作，因为系统最基本的元素在这里被确定。系统分析师可以用类图描述抽象和它们的职责。

#### (2) 对简单协作建模

现实世界中的事物大多都是相互联系、影响的，将这些事物抽象成类后，情况也是如此。所要构造软件系统中的类很少有孤立存在的，它们总是和其他类协作工作，以实现强于单个类的语义。因此，在抽象了系统词汇后，系统分析师还必须将这些词汇中对事物协作工作的方式进行可视化并详述。

#### (3) 对逻辑数据库模式建模

在设计一个数据库时，通常使用数据库模式来描述数据库的概念设计。数据库模式建模是对数据库概念设计的蓝本。可以使用类图对这些数据库的模式进行建模。

#### 表示法

类图是用图形方式表示静态视图。通常，为了表示一个完整的静态视图，需要几个类图。每个独立的类图不需要说明基础模型中的划分，即使是某些逻辑划分，例如包是构成该图的自然边界。

### 20. 类名 (class name)

每个类都必须有一个非空的类名，这在类的外壳（例如包或者包含类）内对于类元而言是唯一的。在实践中，类名通常用问题域中的短名词或名词词组来表示。如果类名用英文来表示，那么通常情况下，类名中每一个组成词的第一个字母应该大写，例如 Teacher、MobilePhone。

### 21. 包 (package)

包是一个用来将模型的单元分组的通用机制。可以把一个系统看作是一个单一的、高级的包。

#### 语义

包是模型的一部分，模型的每一部分必须属于某个包。系统建模人员可以将模型的内容分配到包中。但为了使其能工作，分配必须遵循一些合理的原则，如公共规则、紧密耦合的实现和公共观点等。UML 对如何组包并不强制使用什么规则，但是良好的解组会极大的增强模型的可维护性。

包被作为访问及配置控制机制,以便允许开发人员在互不妨碍的情况下组织大的模型并实现它们。更为特殊的是,要想能够起作用,包必须遵循一定的语意规则。因为它们是作为配置控制单元,所以应该包含那些可能发展到一起的元素。包也必须把一并编译的元素分组。如果对一个元素的改变会导致其他元素的重新编译,那这些元素也应该放到相同的包内。

每个模型元素必须包含在一个且仅一个包或别的模型元素里。否则的话,模型的维护、修改和配置控制就成为不可能的。拥有模型元素的包控制它的定义。它可以在别的包里被引用和使用,但是对这个包的改变会要求访问授权并对拥有该包的包进行更新。

UML 的扩充机制同样适用于包。可以使用标记值来增加包的新特性,用构造型来描述包的新种类。UML 定义了 5 种构造型来为其标准扩充。它们分别是:虚包 (facade)、框架 (framework)、桩 (stub)、子系统 (subsystem) 和系统 (system)。

虚包 (facade) 是包的一种扩充,它只拥有对其他包内元素的引用,自己本身不包括任何定义模型元素。

框架 (framework) 是一个主要由样式 (pattern) 组成的包。

桩 (stub) 描述一个作为另一个包的公共内容代理的包。

子系统 (subsystem) 代表系统模型中一个独立的组成部分。子系统代表系统中一个语义内聚的元素的集合,可以用接口来指定与外界的联系和其外部行为特征。

系统 (system) 代表当前模型描述的整个软件系统。

### 表示法

包用一个矩形框来表示,矩形框的左上角带一个突出的小矩形。

可以画出包符号之间的关系以显示包中的一些元素之间的联系。特别的,包之间的依赖关系 (不同于授权依赖关系,比如访问和导入) 表明元素之间存在一种或多种的依赖关系。

工具可以通过选择性的显示某种可见性级别的元素,比如所有的公共元素,来说明可见性。工具也可以通过图像标记,比如颜色或字体来说明可见性。

一个包 (Package) 元素对外的可见性可以通过在该元素的名字前面添加可见性标志来说明 (“+” 表示公共,“-” 表示私有,“#” 表示被保护)。

### 示例

图 A-16 所示描述了一个图书馆信息系统的包图。它把整个系统分为了 4 个包,分别为 Business Object Package、Utility Package、Database Package、UI Package。包间的依靠关系使用虚线来表示。

### 标准元素

访问、扩展、虚包 (Facade)、框架、桩 (Stub) 和系统。

#### 22. 类元 (classifier)

类元是模型中的离散概念,拥有身份、状态、行为和关系。有几种类元包括类、接口和数据类型。其他几种类元是行为概念、环境事物、执行结构的具体化。这些类元中包括用例、参与者、组件、节点和子系统。

类是最常见的类元。虽然每种类元都有各自的元模型为代表,但是它们都可以按照类的概念来理解,只是在内容和使用上有某些特殊限制。类的大多数特性都适用于类元,通常只是为每种类元而增加了某些特殊限制条件。

## 标准元素

枚举 (enumeration) \ 位置 (location) \ 元类 (metaclass) \ 持久性 (persistence) \ 强类型 (powertype) \ 进程 (process) \ 语义 (semantics) \ 构造型 (stereotype) \ 线程 (thread) \ 效用 (utility) \

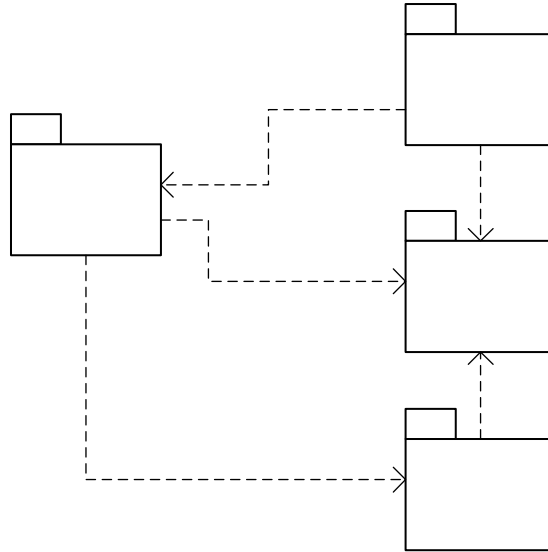


图 A-16 包图

### 23. 静态视图 (static view)

静态视图用于对应用领域中的概念以及系统实现有关的内部概念建模,它将行为实体描述成离散的模型元素,但不描述与时间有关的系统行为。静态视图包含类元和它们相互之间的联系:关联、泛化、依赖和实现。有时它也被称为类视图。

#### 语义

静态视图是 UML 的基础。模型中静态视图的元素是应用中有意义的概念,这些概念包括真实世界中的概念、抽象的概念、实现方面的概念和计算机领域的概念,即系统中的各种概念。静态视图显示了系统的静态结构,特别是存在事物的种类(例如类或者类型),它们的内部结构,相互之间的联系。尽管静态视图可能包含具有或者描述暂时性行为的事物的具体发生,但静态视图不显示暂时性的信息。

静态视图说明了对象的结构。一个面向对象的系统使数据结构和行为特征统一到一个独立的对象结构中。静态视图包括所有的传统数据结构思想,同时也包括了数据操作的组织。数据和操作都可量化为类。根据面向对象的观点,数据和行为是紧密相关的。

静态视图将行为实体描述成离散的模型元素,但是不包括它们动态行为的细节。静态视图将这些行为实体看作是将被类所指定、拥有并使用的物体。这些实体的动态行为由描述它们内部行为细节的其他视图来描述,包括交互视图和状态机视图。动态视图要求静态视图描述动态交互的事物—如果不首先说清楚什么是交互作用,就无法说清楚交互作用怎样进行的。静态视图是建立其他视图的基础。

静态视图中的关键元素是类元及它们之间的关系。类元是描述事物的建模元素。有几种类元，包括类、接口和数据类型。包括用例和信号在内的其他类元具体化了行为方面的事物。实现目的位于像子系统、组件和节点这几种类元之后。

类元之间的关系有关联、泛化及各种不同的依赖关系，包括实现和使用关系。

#### 24. 协作 (collaboration)

协作定义了对某些服务有意义的一组参加者和它们的联系，这些参加者定义了交互中的对象所扮演的角色。在协作中规定了它的上下文和交互。从系统的外部可以把协作作为一个单独的文体。

##### 语义

协作描述了在一定的语境中一组对象以及用以实现某些行为的这些对象间的相互作用。它描述了为实现某种目的而相互合作的“对象社会”。协作中有在运行时被对象和连接占用的槽。协作槽也叫做角色，因为它描述了协作中的对象或连接的目的。类元角色表示参与协作执行的对象的描述；关联角色表示参与协作执行的关联的描述。类元角色是在协作中被部分约束的类元；关联角色是在协作中被部分约束的关联。协作中的类元角色与关联角色之间的关系只在特定的语境中才有意义。通常，同样的关系不适用于协作外的潜在的类元和关联。

一个协作有两个方面：结构和行为。在结构方面，协作可以包含任意的分类符的组合，如类、接口、组件、节点，以及它们的联系等。但是，一个协作并不拥有参与协作的这些模型元素，而只是引用它们。协作只是一种概念性的结构块，而不是系统的一个物理性的结构块，在这一点上协作与包、子系统是不同的。在行为方面，一个协作规定了参与协作的模型元素相互交互的动态行为。

当在系统分析中建立了系统的 Use Case 模型后，需要进一步把它映射为设计模型，即需要实现这些 Use Case，则要描述每一个 Use Case 的具体结构和行为，这就要使用协作。一个 Use Case 可以用一个或多个协作实现。协作本身则用协作图、时序图、类图或对象图分别展开表示。

参数化协作 (Parameterized Collaboration) 定义一个协作家族，家族中的协作有共同的形式，但是参与协作的对象类等模型元素是不同的。参数化协作又称为方案 (Pattern) 或模板协作 (Template Collaboration)。

参数化协作中的参数代表参与协作的角色。当把一个参数化协作中的参数绑定到具体的模型元素，就产生一个实例协作。一个参数化协作可以生成多个实例协作。

##### 表示法

参数化协作的图形表示是在一个虚线椭圆的右上角嵌一个虚线矩形，在虚线椭圆中有参数化协作的名字，还可以包含表达协作结构的类及其联系，在虚线矩形中列出参数名。

##### 示例

如图 A-17 所示表示一个“商品经销”参数化协作，它有参数“顾客”、“经销商”、“提供”和“商品批次”。在图中给出了由参与协作的角色“顾客”、“经销商”、“提供”和“商品批次”所构成的协作的结构。



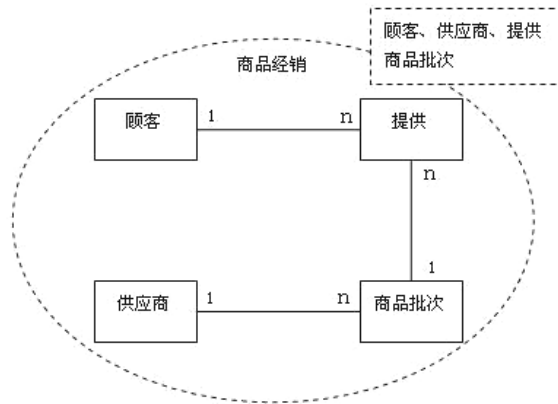


图 A-17 参数化协作

## 25. 协作图 ( Collaboration diagram )

协作图是一种类图，它包含类元角色和关联角色，而不仅仅是类元和关联。协作图强调参加交互的各对象的组织。协作图只对相互间有交互作用的对象和这些对象间的关系建模，而忽略了其他对象和关联。

### 语义

协作图是表示角色间交互的视图，也就是协作中的实例及其链。在形成协作图时，首先将参与交互作用的对象放在图中，然后连接这些对象，并用对象发送和接收的消息来修饰这些连接。协作图描述了两个方面的内容：第一是对交互作用的对象静态结构的描述，包括相关的对象的关系、属性和操作，这也就是对协作所提供的语境建模；第二是为完成工作在对象间交换的消息的时间顺序的描述。

时序图和协作图都可用于对系统的动态方面的建模，而协作图更强调参加交互的各对象的组织。以下是协作图有别于时序图的两点特性。

#### (1) 协作图有路径

为了说明一个对象如何与另一个对象链接，可以在链的末路上附上一个路径构造型。例如构造型 `<<local>>`，它表示指定对象对发送者而言是局部的。

#### (2) 协作图有顺序号

为了描述交互过程中消息的时间顺序，需要给消息添加顺序号。顺序号是消息的一个数字前缀，它是一个整数，由 1 开始递增，每个消息都必须有唯一的顺序号。可以通过点表示法代表控制的嵌套关系，也就是说在激活期 1 中，消息 1.1 是嵌套在消息 1 中的第 1 个消息，它在消息 1.2 之前，消息 1.2 是嵌套在消息 1 中的第 2 个消息，它在消息 1.3 之前。嵌套可以有任何深度。与时序图相比，协作图能显示更为复杂的分支。

协作图中包括如下元素：类角色( class role )、关联角色( association role )和消息流( message flow )。

类角色代表协作图中对象在交互中所扮演的角色。

关联角色代表协作图中链接在交互中所扮演的角色。

消息流代表协作图中对象间通过链接发送的消息。

## 表示法

UML 中，交互图中的对象用矩形表示，矩形内是此对象的名字，链接用对象间相连的直线表示，连线可以有名字，它标于表示链接的直线上。如果对象间的链接有消息传递，则把消息的图标沿直线方向绘制，消息的箭头指向接受消息的对象，消息上保留对应时序图的消息顺序号。

## 示例

如图 A-18 所示是订单购物的协作图。

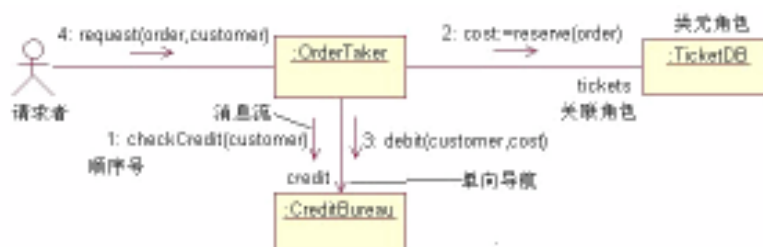


图 A-18 协作图

## 26. 组合 (combination)

组合是对不同类或包进行性质相似的融合。

## 语义

组合描述符的其他两种方法是使用扩展和包含关系。(泛化本来也可以列在该条目下，但是由于其特殊的重要性，所以将它作为独立的基本关系)。

## 表示法

组合用带有构造类型关键字的虚线箭头表示。

## 27. 注释 (comment)

注释是一个图框内的文字说明，它不对模型元素的语义产生影响。注释没有直接的语义，但是它可以表示对于建模者或工具有意义的语义信息或者其他信息。

## 语义

在机械制图和电子线路图中，注解是大量存在的，设计者用这些注解说明、表示产品的工艺要求。

在 UML 中，注释用来描述施加于一个或多个模型元素的限制或对模型元素的语义加以说明。注释包含文本字符串，如果工具允许，也可以带有嵌入的文件。注释可以附在模型元素，或者元素集上。

## 表示法

注释在 UML 模型图中用折了右上角的长方形表示，并且用虚线与被注释的一个或几个元素相连，如图 A-19 所示。在此长方形中写注释的内容，注释的内容可以是形式或非形式化的文本，也可以是图形。

工具可以使用其他的形式来表示或者导航注释信息，例如弹出式备注，特殊字体等。

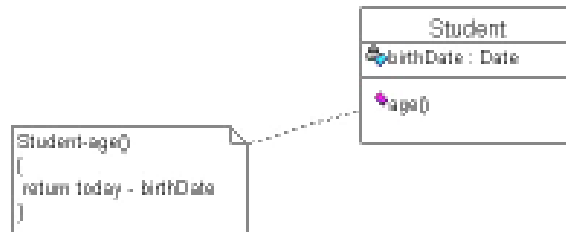


图 A-19 注释

## 标准元素

需求 (requirement), 职责 (responsibility)

### 28. 组件 (构件, component)

组件是定义开发时和运行时的物理对象的类。组件是系统中可替换的物理部件, 它包装了实现而且遵从并统一提供一组接口的实现。

#### 语义

组件是定义了良好接口的物理实现单元, 它是系统中可替换的部分。每个组件体现了系统设计中特定类的实现。良好定义的组件不直接依赖于其他组件而依赖于组件所支持的接口。在这种情况下, 系统中的一个组件可以被支持正确接口的其他组件所替代。

组件具有它们支持的接口和需要从其他组件得到的接口。接口是被软件或硬件所支持的一个操作集。通过使用命名的接口, 可以避免在系统的各个组件之间直接发生依赖关系, 有利于新组件的替换。组件视图展示了组件间相互依赖的网络结构。组件视图可以表示成两种形式, 一种是含有依赖关系的可用组件 (组件库) 的集合, 它是构造系统的物理组织单元。它也可以表示为一个配置好的系统, 用来建造它的组件已被选出。在这种形式中, 每个组件与给它提供服务和其他组件连接, 这些连接必须与组件的接口要求相符合。

常见的组件有系统的配置组件, 如 COM+ 组件、Java Beans 等。组件也可以是软件开发过程中的产物, 如软件代码 (源码、二进制码和执行码) 等。

按照组件的作用可以把组件分为以下 3 种。

配置组件 (Deployment Component) 是构成一个可执行的系统的必需的组件, 如动态链接库 (DLL) 执行程序 (EXE) 等。UML 的组件可以表达典型的对象模型, 如 COM+、CORBA、JAVA Beans、Web 页和数据库库表等内容。

工作产品组件 (Work Product Component) 是在软件开发阶段使用的组件, 它们包括源程序文件、数据文件等。这些组件并不直接构成可执行系统, 它们是系统开发过程中的产品, 配置组件是根据工作产品组件建立的。

执行组件 (Execution Component) 是执行系统的部件, 如 COM+ 的一个对象, 它是一个动态链接库 (DLL) 的实例。

UML 的所有扩展机制都可以用于组件。例如, 可以在组件上加上标记值, 描述组件的性质。使用构造型规定组件的种类。UML 定义了以下 5 个用于组件的标准构造型。

构造型 <<executable>> 说明一个组件在系统的节点上执行。

构造型 <<library>> 说明一个组件是一个静态的或动态的对象库。

构造型<<table>>说明一个组件代表的是一个数据库表。

构造型<<file>>说明一个组件代表的是一个文档，它包含的是源代码或数据。

构造型<<document>>说明一个组件代表的是一个文档。

### 表示法

在 UML 中，图形上组件使用左侧带有两个突出的小矩形的矩形表示。

组件的名字位于组件图标的内部，组件名是一个文本串（如图 A-20 左边图标所示）。如果组件被某包所包含，可以在它的组件名前加上它所在包的名字（如图 A-20 中间图标所示），Reservation.java 组件是属于事务包（Business）的。图 A-20 右边的图标还增加了一些表达组件的细节信息，它在图标中添加了实施该组件所需要的类。

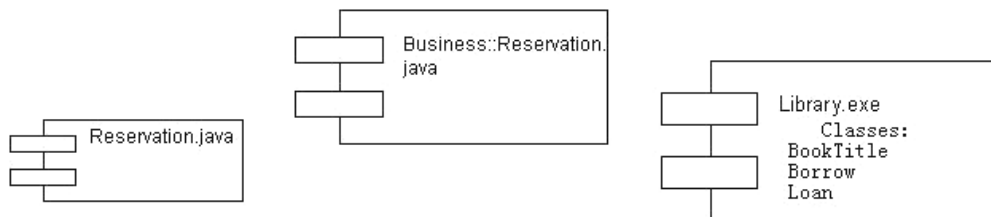


图 A-20 组件

下列扩展定义说明了设立组件的意图，以及确定系统的一个部分是否被作为有意义组件的考虑。

组件是重要的，在功能和概验上都比一个类或者一行代码强。典型的，组件拥有类的一个合作的结构和行为。

一个组件基本独立于其他组件，但是很少独立存在。一个给定组件与其他组件协作完成某种功能，为了完成这项功能，组件假设了一个结构化的上下文。

组件是系统中可以替换的部分。它的可替换性使得可以用满足相同接口的其他组件进行替换。替换或者插入组件来形成一个运行系统的机制一般对组件的使用者是透明的。对象模型不需要多少转换就可使用或利用某些工具自动实现该机制。

组件完成明确的功能，在逻辑上和物理上有粘聚性，因此它表示一个更大系统中一段有意义的结构和/或行为块。

组件存在于具有良好定义结构的上下文中。它是系统设计和组建的基石。这种定义是递归的，在某种层次上抽象的系统仅是更高层次抽象上的组件。

组件不会单独存在，每一个组件都预示了它将处于的结构和/或技术的上下文。

一个组件符合一系列接口。符合一个接口的组件满足接口指定的约定，在接口适用的所有上下文中都可替换。

### 标准元素

文档（document）、可执行（executable）、文件（file）、库（library）、位置（location）、表（table）。

### 29. 组件图（component diagram）

组件图描述软件组件以及组件之间的关系，组件本身是代码的物理模块，组件图则显示了

代码的结构。

### 语义

组件图由组件、接口和组件之间的联系构成，其中的组件可以是源码、二进制码或可执行程序。组件图表示系统中的不同物理部件及其联系，它表达的是系统代码本身的结构。

组件图只有型（type）的形式，没有实例形式。为了显示组件的实例需要使用配置图。

一个包含了组件型和节点型的组件图用于表示系统的静态依赖关系，例如程序间的编译依赖，它用虚箭线从一个客户组件（Client Component）指向供应者组件（Supplier Component），依赖的种类则是由实现决定的，可用依赖的构造型表示。

组件图中可以包括包和子系统，它们可以将系统中的模型元素组织成更大的组块。有时，当系统有需要可视化一个基于组件的一个实例时，还需要在组件图中加入实例。

在对系统的静态实现视图建模时，通常将按下列 4 种方式之一来使用组件图。

#### （1）对源代码建模

当前大多数面向对象编程语言，使用集成化开发环境来分割代码，并将源代码存储到文件中。如果用 Java 语言开发系统，要将源代码储存在.java 文件中。如果使用 C++ 开发系统，要将源代码存储在头文件（.h 文件）和体文件（.cpp 文件）中。

#### （2）对可执行体的发布建模

当用组件图对发布建模时，其实是在对构成软件的物理部分所做的决策进行可视化、详述、文档化。用组件图来可视化、详述、构造和文档化可执行体的发布配置，包括形成每个发布的实施组件以及这些组件间的关系。

#### （3）对物理数据库建模

可以把物理数据库看作模式（schema）的具体实现。可以用组件图表示这些以及其他种类的物理数据库。组件可以有属性，所以对物理数据库建模常用的做法是用这些属性来指定每个表的列。组件也可以有操作，这些操作可以用来表示存储的过程。

#### （4）对可适应的系统建模

某些系统是相对静态的；其组件进入现场，参与执行，然后离开。另一些系统则是较为动态的，其中包括一些活动代理或者为了负载均衡和故障恢复而进行迁移的组件。可以将组件图与对行为建模的 UML 的一些图结合起来表示这类系统；

在实际建模过程中，读者可以参照以下步骤进行：对系统中的组件建模；定义相应组件提供的接口；对它们间的关系建模；对建模的结果进行精华和细化。

### 表示法

可以用包含组件类元和节点类元的图来表示编译依赖关系，这种依赖关系用带箭头的虚线表示，箭头从用户组件指向它所依赖的服务组件。依赖关系的类型是用语言说明的，可作为依赖关系的构造型显示。

图还可以用于表示组件之间的接口和调用关系。虚线箭头从一个组件指向其他组件上的接口。

### 示例

组件图表示了组件之间的依赖关系（如图 A-21 所示）。每个组件实现（支持）一些接口，并使用另一些接口。如果组件间的依赖关系与接口有关，那么组件可以被具有同样接口的其他

组件替代。

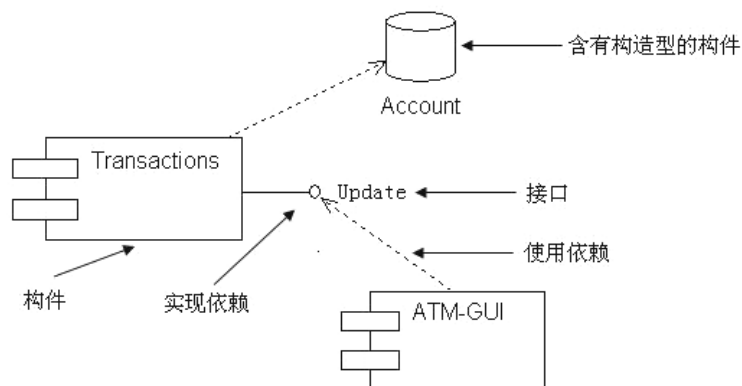


图 A-21 组件图

### 30. 复合状态 ( composite state )

复合状态是指包含一个或多个嵌套状态机的状态，也称为子状态机。

#### 语义

复合状态由状态和子状态组成。对这些状态而言，复合状态是超状态，每个子状态是所属超状态的全部迁移。

每个子状态本身可能就是复合状态。显然，应该对嵌套个数有所限制。为了系统建模结构的清晰，一般情况下，嵌套级数不超过二级或三级。如果超状态只含有一个嵌套状态机，那么它就是顺序复合状态。如果超状态含有两个或多个状态机，这些状态机将并发执行，这个超状态被称为并发复合状态。

#### 表示法

组成状态是包含有从属细节的状态。它带有名字分格，内部转换分格和图形分格。图形分格中有用于表示细节的嵌套图。所有的分格都是可选的。为了方便起见，文本分格（名称分格和内部转换分格）可以缩略为图形分格内的制表符，而无需水平延伸它。

将图形分格用虚线分成子区域，表示将并发组成状态分为并发子状态。每个子区域代表一个并列子状态，它的名字是可选的，但必须包括带有互斥的子状态的嵌套状态图。用实线将整个状态的文字分格与并发子状态分格分开。

在图形分格中，用嵌套的状态表图表示将状态扩展为互斥的子状态。

初始状态用小实心圆表示。在顶层状态机中，源自初始状态的转换上可能标有创建对象的事件，否则转换必须是不带标签的。如果没有标签，则它代表所有到封装状态的转换。初始转换可以有一个动作。初始状态是一个符号设备，对象可以不处于这种状态中，但必须转换到实际的状态中。

终态用外面套了圆环的实心圆表示。它表示封装状态中的活动完成。它触发标有隐含的完成事件活动的封装状态上的转换（通常为无标签转换）。

#### 举例

如图 A-22 所示表示包含两个互斥子状态的顺序组成状态，一个初始状态和一个终态。当

组成状态为活动时，子状态 Start（初始状态的目标状态）首先变为活动的。

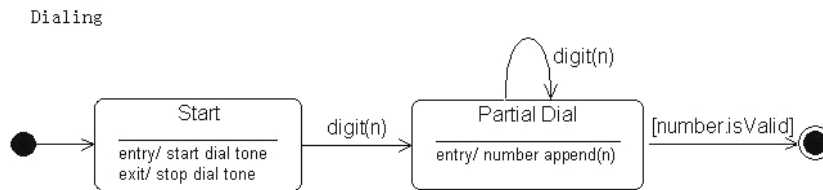


图 A-22 顺序组成状态

图 A-23 所示表示带有 3 个正交状态的并发组成状态。每个并发子状态又进一步分为顺序子状态。当组成状态 Incomplete 成为活动状态时，初始状态的目标状态成为活动的。当 3 个子状态都达到终态后，外部组成状态的完成转换被触发，Passed 成为活动状态。如果在 Incomplete 为活动状态时发生 fail 事件，则所有的 3 个并发子状态结束，Failed 成为活动状态。

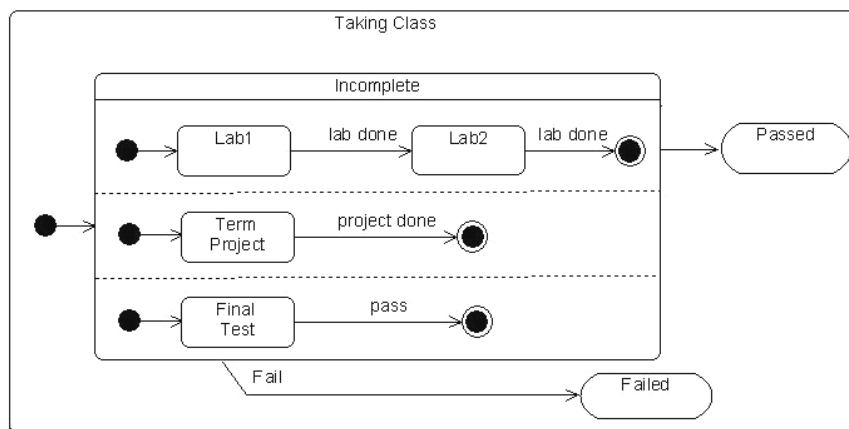


图 A-23 并列组成状态

### 31. 并发（concurrency）

在同一时间间隔内，两个或两个以上的活动的执行。每个活动对象具有自身的控制线程，并不隐含的要求这些活动同步。通常，除了明确的同步点之外，它们的活动是相互独立的，可以通过插入或者同时执行多个线程来实现并发。

### 32. 并发子状态（concurrency substate）

一个可以与同一个组成状态中的其他子状态同时存在的子状态。

### 33. 约束（constraint）

约束是一种为模型元素指定必须为真的语义或条件机制。UML 预定义了某些约束，其他可以由建模者自行定义。

### 语义

约束是 UML 3 种扩展机制之一，另外两种是标记值和构造型。需要一提的是，UML 的扩

展机制是违反 UML 的标准形式的，并且使用它们会导致相互影响。在使用扩展机制之前，建模者应该仔细权衡它的好处和代价，特别是当现有机制能够合理工作时。典型地，扩展用于特定的应用域或编程环境，但是它们导致了 UML 方言的出现，包括所有方言的一些优点和缺点。

在 UML 中，约束用来扩展 UML 建模的语义，以便增加新的规则或修改已经存在的规则。在 UML 中，每一个建模元素都有明确的语义。语义规定了建模元素为软件系统建模的规则。如果在建模时，有些特定的规则不包含在现有的 UML 语义中，就可以使用约束对建模元素的现有建模规则进行扩展。约束为对应的建模元素规定了一个条件，对于一个完备的模型而言，此建模对象必须使该条件被满足。

### 表示法

约束是用文字表达式表达的语义限制。每个表达式有一种隐含的解释语言，这种语言可以是正式的数学符号，如集合论表示法；或是一种基于计算机的约束语言，如 OCL；或是一种编程语言，如 Java，或是伪代码或非正式的自然语言。当然，如果这种语言是非正式的，那么它的解释也是非正式的，并且要由人来解释。即使约束由一种正式诺言来表示，也不意味着它自动为有效约束。

约束可以表示不能用 UML 表示法来表示的约束相关关系。当陈述全局条件或影响许多元素的条件时约束特别有用。

在 UML 中，约束被图形化为一个文本串，此文本串被括在一对花括号内，并被放置在被约束的建模元素附近。

对于简单图形符号（例如类或者关联路径）：约束字符串可以标在图形符号边上，如果图形符号有名字，就标在名字边上。

对于两个图形符号（例如两个类或两个关联）：约束用虚线箭头表示。箭头从一个元素连向另一个，并带有约束字符串（在大括号内）。箭头的方向与约束的信息相关。

对于 3 个或更多的图形符号：约束用注释符号表示，并用虚线与各个图形符号相连。这种表示法适用于其他情况。对 3 个或更多的同类路径（例如泛化路径或者关联路径），约束标在穿过所有路径的虚线上。为避免混淆，不同的连线可以标号或加标签，从而建立它们与约束之间的对应关系，如图 A-24 所示。

### 标准元素

不变量（invariant），后置条件（postcondition），前置条件（precondition）。

#### 34. 数据类型（data type）

没有标识符的一组值的描述符（独立存在，可能有副作用）。数据类型包括原始预定义的类型和用户自定义的类型。原始类型有数字、字符串、乘方。

### 语义

数据类型是用户可定义类型所需的预定义的基础。它们的语义是在语言结构之外用数学定义的。数字是预定义的，包括实数和整数。字符串也是预定义的。这些类型是用户不可定义的。

数据类型的值没有标记，如基本类型 int、float 和 char，系统开发人员可以在诸如 C++ 和 Java 等语义中找到它们。例如，int 的所有实例具有值 5，并且完全相同、相互无法区分。纯面向对象语言、例如 Smalltalk 没有数据类型。



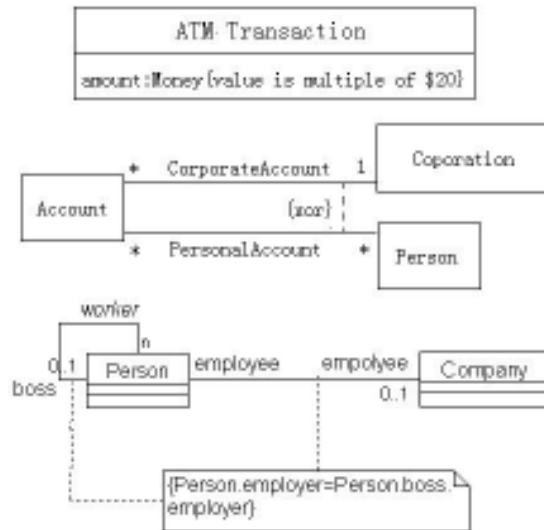


图 A-24 约束表示法

## 35. 数据值 (data value)

数据值是数据类型的实例，是不带标识符的值。

**语义**

数据只是一个数学域的号码，一个纯数值。因为数据值没有标识符，两个表示法相同的数据值是无法区分的。在程序设计语言中，数据值使用值传递。用引用传递数据值是没有意义的，变换数据值也是没有意义的，它的值是永远不变的。实际上，数据值就是值本身。通常所说的变换数据值，是指变换一个有数据值的变量的内容，使其带有新的数据值。而数据值本身是不变的。

## 36. 默认值 (default value)

作为某些程序设计语言或者工具的一部分而自动提供的值。元素属性的默认值不是 UML 语义的组成部分，在模型中不出现。

## 37. 依赖 (dependency)

依赖是两个 (或两组) 模型元素间的语义联系，依赖双方某一个模型元素的变化必影响到另一个模型元素。

**语义**

依赖表示两个或多个模型元素之间语义上的关系。它只将模型元素本身连接起来而不需要用一组实例来表达它的意思。它表示了这样一种情形，提供者的某些变化会要求或指示依赖关系中客户的变化。根据这个定义，关联和泛化都是依赖关系，但是它们有更特别的语义，故它们有自己的名字和详细的语义。

UML 定义了 4 种基本依赖类型。它们分别是使用 (Usage) 依赖、抽象 (Abstraction) 依赖、授权 (Permission) 依赖和绑定 (Binding) 依赖。在定义依赖关系时，要用到两个概念：客户和提供者。客户是指依赖关系起始的模型元素，提供者是指依赖关系箭头所指的模型元素。

### (1) 使用依赖

所有的使用依赖是非常直接的,它通常表示客户使用有提供者提供的服务,以实现它的行为。以下给出 5 种使用依赖定义的应用于依赖关系的原型。

`<<use>>` 依赖是类见最常用的依赖。它声明使用一个模型元素需要用到已存在的另一个模型元素,这样才能正确实现使用者的功能(包括了调用、实例化、参数和发送)。

`<<call>>` 是操作间的依赖,它声明了一个类调用其他类的操作的方法。这种类型的依赖在 UML 建模中趋于不被广泛使用,它适用于更深的建模层次。同样,现在也很少有 CASE 工具支持操作之间的依赖。

`<<parameter>>` 是操作和类之间的依赖,它描述的是一个操作和它的参数之间的关系。同样,这种依赖方式在实际中也较少被使用。

`<<send>>` 描述的是信号发送者和信号接收者之间的关系,它规定客户把信号发送到非指定的目标。

`<<instantiate>>` 是关于一个类的方法创建了另一个类的实例声明,它规定客户创建目标元素的实例。

### (2) 抽象依赖

抽象依赖建模表示客户和提供者之间的关系,它依赖于在不同抽象层次上的物件。以下给出 3 种抽象依赖定义的应用于依赖关系的原型。

`<<trace>>` 声明不同模型中的元素之间存在的一些连接。例如提供者可以是类的分析视图,客户则可以是更详细的设计视图,系统分析师可以用 `<<trace>>` 来描述它们之间的关系。

`<<refine>>` 声明具有不同语义层次上的元素之间的映射。抽象依赖中的 `<<trace>>` 可以用来描述不同模型中的元素间的连接关系,`<<refine>>` 则用于相同模型中元素间的依赖。例如在分析阶段遇到一个类 Student,在设计时,这个类细化成更具体的类 Student。

`<<derive>>` 声明一个类可以从另一个类导出。当想要表示一个事物能从另一事物派生而来时就使用这个依赖构造型。

### (3) 授权依赖

授权依赖表达一个事物访问另一事物的能力。提供者可以规定客户的权限,这是提供者控制和限制对其内容访问的方法。以下给出 3 种授权依赖定义的应用于依赖关系的原型。

`<<access>>` 是包间的依赖,它描述允许一个包访问另一个包的内容。`<<access>>` 允许一个包(客户)引用另一个包(提供者)内的元素,但客户包必须使用路径名称。

`<<import>>` 是与 `<<access>>` 概念相似的依赖,它允许一个包访问另一个包的内容并为被访问包的组成部分增加别名。`<<import>>` 将提供者的命名空间整合到客户的命名空间,但当客户包中的元素与提供者中的元素同名时会产生冲突。在这种情况下,可以使用路径名或增加别名来解决冲突。

`<<friend>>` 允许一个元素访问另一个元素,不管被访问的元素是否可见,这大大的便利了客户类访问提供者的私有成员。但并不是所有的计算机语言都支持 `<<friend>>` 依赖,C++ 允许类间的 `<<friend>>` 依赖,而 Java 和 C# 则不支持。

### (4) 绑定依赖

绑定依赖是较高级的依赖类型,它用于绑定模板以创建新的模型元素。

`<<bind>>` 规定了客户用给定的实际参数实例化提供者模板。

## 表示法

依赖用一个从客户指向提供者的虚箭头表示，用一个构造型的关键字来区分它的种类。箭尾处的模型元素（客户）依赖于箭头处的模型元素（服务者），如图 A-25 所示。

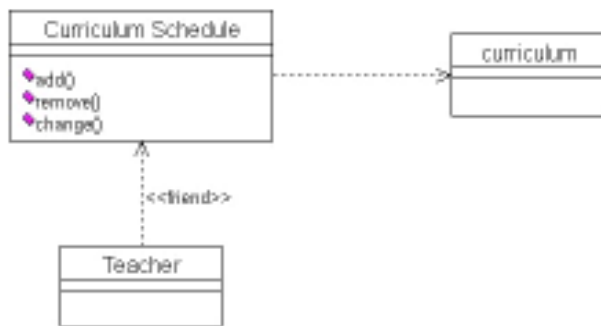


图 A-25 依赖

有几种其他的关系也使用有关键字的虚线箭头表示，但它们不是依赖关系。这些元素有流（成为和复制）、组合（扩展和包含）、约束和注释。如果注释或者约束是元素之一，可以省略箭头，因为约束或者注释总是在箭尾处。

图 A-25 所示表示一个教学管理系统的对象类 Curriculum（课程）与对象类 Curriculum Schedule（课表）之间、对象类 Curriculum Schedule 与对象类 Teacher（教师）之间的依赖关系，其中类 Curriculum 是独立的提供者，类 Curriculum Schedule 是依赖于类 Curriculum 的客户。Curriculum Schedule 的操作 Add、Move 和 Change 都使用了 Curriculum，在这里类 Curriculum 是这两个操作的参数的类型，这是使用依赖的最常见的情况。一旦 Curriculum 发生改变，Curriculum Schedule 必定也随之改变。

在类 curriculum schedule 与类 Teacher 之间的依赖有一个构造型 <<friend>>，说明它们之间不是使用关系，而是友元关系，正如 C++ 语言中的友元那样。

## 标准元素

成为（become）、绑定（bind）、调用（call）、复制（copy）、创建（create）、派生（derive）、扩展（extend）、包含（include）、导入（import）、友元（friend）、~ 的实例（instanceOf）、实例化（instantiate）、强类型（powertype）、发送（send）、跟踪（trace）、使用（use）。

### 38. 配置（deployment）

描述现实世界环境运行系统的配置的开发步骤。在这一步骤中，必须决定配置参数、实现、资源配置、分布性和并行性。

### 39. 配置图（deployment diagram）

配置图描述系统硬件的物理拓扑结构以及在此结构上执行的软件。配置图可以显示计算结点的拓扑结构和通信路径、结点上运行的软件组件、软件组件包含的逻辑单元（对象、类）等。配置图常常用于帮助理解分布式系统。

## 语义

配置图是对面向对象系统的物理方面建模时使用的两种图之一，另一种图是组件图。配置

图显示了运行软件系统的物理硬件，以及如何将软件配置到硬件上。也就是说，这些图描述了执行处理过程的系统资源元素的配置情况以及软件到这些资源元素的映射。

配置图中可以包括包和子系统，它们可以将系统中的模型元素组织成更大的组块。有时，当系统有需要可视化硬件拓扑结构的一个实例时，还需要在配置图中加入实例。配置图中还可以包含组件，这些组件都必须存在于配置图中的节点上。

配置图有描述符形式和实例形式（前文已经介绍过）表现了作为系统结构的一部分的具体节点上的具体组件实例的位置，这是配置图的常见形式。描述符形式说明哪种组件可以存在于哪种节点上，哪些节点可以被连结，类似于类图

配置图描述了运行系统的硬件拓扑。在实际使用中，配置图常被用于模拟系统的静态配置视图。系统的静态配置视图主要包括构成物理系统的组成部分的分布和安装。

### 表示法

配置图是节点符号与表示通信的路径构成的网状图（如图 A-26 所示）。节点符号可以带有组件实例，说明组件存在或运行于该节点上。组件符号可以带有对象，说明对象是组件的一部分。组件之间用虚线箭头相连（可能穿过接口），说明一个组件使用了另一个组件的服务。必要时可以用构造类型说明依赖关系。

配置图类似于对象图，通常用于表示系统中的各个节点的实例。很少用配置图定义节点的种类和节点之间的关系

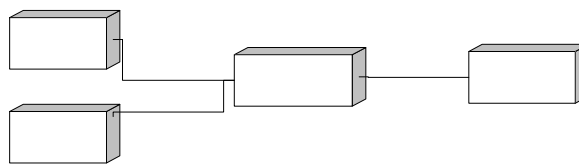


图 A-26 配置图

#### 40. 配置视图（deployment view）

建模把组件物理地配置到一组物理的、可计算的节点上，如计算机和外设上，它允许开发人员建模横跨分布式系统节点上的组件的分布。

配置视图描述了系统的拓扑结构、分布、移交和安装。

#### 41. 派生（derivation）

两个元素间的一种关系，可从一个元素计算出另一个元素。派生可建模成一个带有关键字 derive 的抽象依赖的构造型。

## 42. 设计 ( design )

系统的一个阶段，它从逻辑层次说明系统将如何实现。在设计中，系统开发人员需要确定如何满足功能需求和质量要求。这一步的成果体现为设计层模型，特别是静态视图，状态机图和交互视图。

## 43. 设计时间 ( design time )

设计时间是指在软件开发过程的设计活动中出现的情况。

## 44. 开发过程 ( development process )

开发过程是软件系统的创建、提交和维护等相关活动的组织方式。

UML 是一种建模语言，而不是过程。它的目的是描述模型，而该模型可以用不同的开发过程实现。为了标准化，描述开发工作的结果比解释开发过程更为重要。

从根本上说，一个软件开发过程的描述应该包括从需求分析一直到软件提交的全过程。除此之外，一个完整的开发过程还涉及到软件产品化相关的更广泛的方面。例如软件产品的生命周期、文档的生成和管理、技术支持和培训、各个系统开发工作小组之间的并行工作和相互协作等。

图 A-27 所示表示了在后继步骤和迭代过程中的着重程度的对比。在初期，主要着重于分析；在加工中建立面向设计和实现的元素过程模型；在构造和转变中完成所有元素。

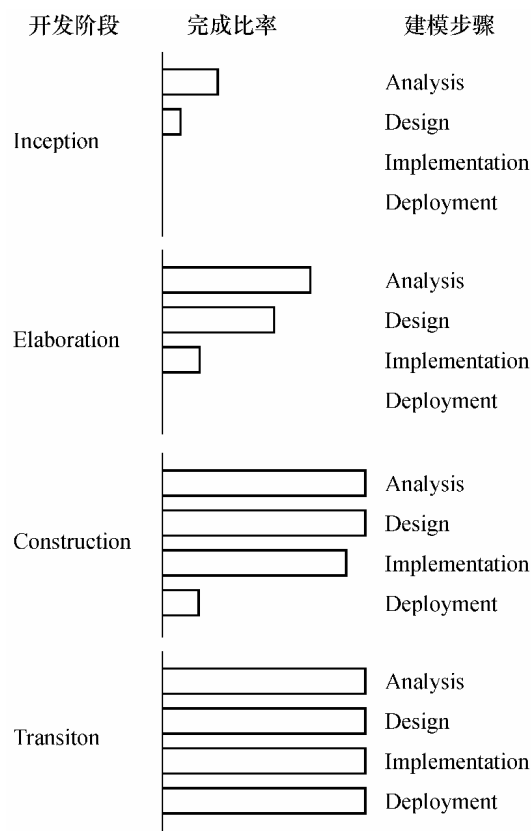


图 A-27 开发阶段后的进展

#### 45. 分布单元 (distribution unit)

定位于一个操作系统进程或者处理器中成组存在的一些对象或组件,可以表现为运行时的组成或者聚集。它是配置视图中的一个设计概念。

#### 46. 动态视图 (dynamic view)

动态模型描述了系统随时间变化的行为,这些行为是用从静态视图中抽取的系统的瞬间值的变化来描述的。UML 的动态视图由状态机视图、活动视图和交互视图组成。

##### 语义

在 UML 的表现上,动态模型主要是建立系统的交互图和行为图。其中交互图包括时序图和协作图;行为图则包括状态图和活动图。

时序图用来显示对象之间的关系,并强调对象之间消息的时间顺序,同时显示对象之间的交互;协作图主要用来描述对象间的交互关系;状态图通过对类对象的生存周期建立模型来描述对象随时间变化的动态行为;活动图是一种特殊形式的状态机,用于对计算流程和 workflow 建模。一个完整的动态视图包括这 4 种图,每种图分别描述对象之间动态关系的一个侧面。

#### 47. 元素 (element)

组成模型的原子。此处所说的是可以用于 UML 模型的元素——即表达语义信息的模型元素以及用图形表示模型元素的表达元素。

##### 语义

元素的意义相当广泛,没有什么具体的语义。

#### 48. 入口动作 (entry action)

入口动作是在状态被激活的时候执行的动作,在活动状态机中,动作状态所对应的动作就是此状态的入口动作。

##### 语义

入口动作通常用来进行状态所需要的内部初始化。因为不能回避一个入口动作,任何状态内的动作在执行前都可以假定状态的初始化工作已经完成,不需要考虑如何进入这个状态。

出口动作可以处理这种情况以使对象的状态保持前后一致。入口动作和出口动作原则上依附于进来的和出去的转换,但是将它们声明为特殊的动作可以使状态的定义不依赖状态的转换,因此起到封装的作用。

##### 表示法

入口活动按照内部转换的语法编码,带有虚拟事件名字 entry (entry 是保留字,不能用作实际事件的名字)。

##### 讨论

入口和出口活动在语义上不是必需的(入口活动可以从属于所有进入状态),但它们实现了状态的封装,从而使外部使用与内部结构分开。它们允许定义初始化和终态活动,而不必担心会被略过。对于异常尤其有用,因为它定义的活动即使出现异常时也会被执行。通常,出口活动与入口活动一起使用。入口活动分配资源,出口活动释放它们。即使出现外部转换,资源也会被释放。

#### 49. 事件 (event)

事件是对一个在时间和空间上占有一定位置的有意义的事情的规格说明。

##### 语义

在状态机中，一个事件是一次激发产生的，激发能够触发一个状态的转换。信号是一种事件，表示一个实例间进行通信的异步激发规格说明。

事件可以是内部的事件和外部的事件。外部的事件是在系统和它的参与者之间传送的事情。例如电脑 Reset 键的按下就是一个外部事件。内部事件是在系统内部的对象之间传送的事件。内存溢出事件就是一个内部事件。

实际建模过程中，一般使用 UML 对 4 种事件建模：信号、调用、事件推移和状态的一次改变。

在对事件建模可以参照以下原则：建立信号的层次关系，以便发掘相关信号的公共特性；不要使用发送信号，特别是不要发送异常来代替正常的控制流；确保每个可能接收事件都对应于一个适当的状态机；建模过程中，不仅要对接收事件的元素建模，还要对发送事件的元素建模。

##### 标准元素

创建 (create)，销毁 (destroy)。

#### 50. 出口动作 (exit action)

退出某状态时执行的动作。

##### 语义

无论何时从一个状态离开都要执行一个出口动作来进行后处理工作。当出现代表错误情况的高层转换使嵌套状态异常终止时，出口动作特别有用。

出口动作可以处理这种情况以使对象的状态保持前后一致。入口动作和出口动作原则上依附于进来的和出去的转换，但是将它们声明为特殊的动作可以使状态的定义不依赖状态的转换，因此起到封装的作用。

##### 表示法

出口动作的语法是：exit/执行的动作。这里所指的动作可以是原子动作，也可以是动作序列 (action sequence)

#### 51. 异常 (exception)

类、接口的操作执行失败行为引起的信号。

##### 语义

开发人员在对系统的类或接口的行为建模时，有必要说明它们的操作会产生的异常情况。在使用某类或接口时，每个调用操作可能发生的异常则不清楚，这就需要对它们进行可视化的建模。

在 UML 中，异常是一种信号，并被建模为构造型化的类。异常可以被附加到操作说明中。对异常建模在某种程度上与对信号的一般建模相反。对一个信号的族建模主要是说明一个主动对象接收的各种信号；而对异常建模主要是说明一个对象可能通过它的操作来发出的各种异常。

**表示法**

用构造类型<<exception>>来区别声明和异常。状态机中的事件名字不用标以构造型。

## 52. 框架 ( framework )

为某个域中的应用程序提供可扩展模板的泛化结构。

## 53. 功能视图 ( functional view )

传统开发方法中，数据流图是功能视图的核心。UML 不直接支持这种视图，但是活动图中有一些功能特性。功能视图将系统分解为功能或者提供功能的操作。通常认为功能视图不是面向对象的，可能导致不易维护的结构。

## 54. 可泛化元素 ( generalizable element )

可以参与泛化关系的模型元素。

**语义**

可泛化元素可以有父和子，被类元成带有元素的变量可以带有该元素后代的实例。

可泛化元素包括类、用例、其他类元、关联、状态和协作，它们继承其祖先的特征。每种可泛化元素的哪个部分是继承来的，要看元素的种类。例如类继承属性、方法、操作、关联中的地位 and 约束；关联继承参与类（本身可被特化）和关联端的特性；用例继承属性、操作、与参与者的关联、与其他用例的扩展和包含关系、行为序列。状态继承转换。

**结构**

可泛化元素的属性声明它可以在泛化关系中的何处出现。

抽象：说明可泛化元素是描述直接实例的，还是抽象元素的。True 表示元素是抽象的（不能有直接实例）；False 表示它是具体的（可以由直接实例）。抽象元素的实体后代才能使用。带有无方法操作的类是抽象的。

叶：说明可泛化元素是否可以特化。True 表示元素不能有后代（叶）；False 表示可以有后代（无论当前是否有后代）。作为叶的抽象类只能起组织全局属性和操作的作用。

根：说明元素是否必须为无祖先的根。True 表示元素必须为根；False 表示不必为根，可以有祖先（无论当前是否有祖先）。

声明叶或者根不影响语义，但这种声明可以给设计者提示。如果能避免对全局变量的分析或者对多态操作的全局保护，就可以有更高的编译效率。

**标准元素**

叶 ( leaf )

## 55. 泛化 ( generalization )

泛化是一般事物（称为超类或父类）和该事物的较为特殊的种类（称为子类）之间的关系，子类继承父类的属性和操作，除此之外通常子类还添加新的属性和操作，或者修改了父类的某些操作。

**语义**

泛化关系是类元的一般描述和具体描述之间的关系，具体描述建立在一般描述的基础上，并对其进行了扩展。具体描述与一般描述完全一致，具有其所有特性、成员和关系，并且包含补充的信息。例如，抵押是借贷中具体的一种，抵押保持了借贷的基本特性并且加入了附



加的特性,如房子可以作为借贷的一种抵押品。一般描述被称作父,具体描述被称作子,如借贷是父而抵押则是子。泛化在类元(类、接口、数据类型、用例、参与者和信号等)、包、状态机和其他元素中使用。在类中,术语超类和子类代表父和子。

泛化有两个用途。第一个用途是用来定义下列情况:当一个变量(如参数或过程变量)被声明承载某个给定类的值时,可使用类(或其他元素)的实例作为值,这被称作可替代性原则(由 Barbara Liskov 提出)。该原则表明无论何时祖先被声明了,则后代的一个实例可以被使用。例如,如果一个变量被声明拥有借贷,那么一个抵押对象就是一个合法的值。

泛化的另一个用途是在共享祖先所定义的成分的前提下允许它自身定义增加的描述,这被称作继承。继承是一种机制,通过该机制类的对象的描述从类及其祖先的声明部分聚集起来。继承允许描述的共享部分只被声明一次而可以被许多类所共享,而不是在每个类中重复声明并使用它,这种共享机制减小了模型的规模。更重要的是,它减少了为了模型的更新而必须做的改变和意外的前后定义不一致。对于其他成分,如状态、信号和用例,继承通过相似的方法起作用。

### 约束

约束可以应用于一系列泛化关系,以及有同一个父的子。可以规定下列属性。

- (1) 互斥:一个祖先不能有两个子(多重继承时),实例不能同时成为两个子的间接实例(多重类元语义)。
- (2) 重叠:一个祖先可以有两个或者更多的子,实例可以属于两个或者更多的子。
- (3) 完整:列出了所有可能的子,不能再增加。
- (4) 不完整:没有列出所有可能的子,有些已知子没有声明,还可以增加新的子。

### 表示法

在图形上,泛化用从子类指向父类的空心三角形箭头表示,多个泛化关系可以用箭头线表示的树来表示,每一个分支指向一个子类(如图 A-28 所示)。指向父的线可以是组成或者树(如图 A-29 所示)。

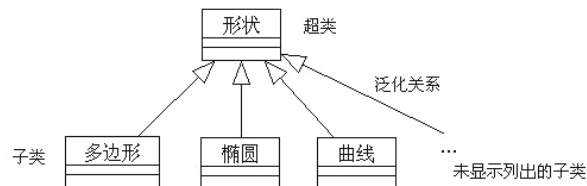


图 A-28 泛化关系

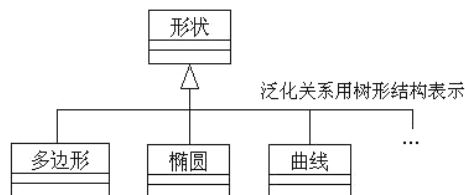


图 A-29 树形泛化关系

泛化还可以应用于关联，但是太多的线条可能使图很乱。为了标明泛化箭头，可以将关联表示为关联类。

如果图中没有标出模型中存在的类，应该在相应位置上用省略号 (...) 代替（这不表示将来要加入新的类，而是说明当前已经有类存在，这种方法表示忽略的信息，而不是语义元素）。一个类的子类表示为省略号说明当前至少有一个子类没有在视图中标处。省略号可以有描述符。这种表示法是由编辑工具自动维护的，不用手工输入。

### 标准元素

完整、互斥、实现、不完整、重叠。

#### 56. 书名号 ( guillemets )

书名号 (<< >>) 在法语，意大利语和西班牙语中表示引用。在 UML 表示法中表示关键字和构造类型。许多字体中都有，必要时可以用两个尖括号代替。

#### 57 交互图 ( interaction view )

交互图描述了一个交互，它由一组对象和它们之间的关系组成，并且还包括在对象间传递的信息。交互图表达对象之间的交互，是描述一组对象如何协作完成某个行为的模型化工具。

### 语义

一张交互图显示的是一个交互，由一组对象和它们之间的关系组成，包含它们之间可能传递的消息。时序图和协作图都是交互图，时序图是强调消息时间顺序的交互图；协作图则是强调接收和发送消息的对象的结构组织的交互图。

交互应用于对一个系统的动态方面建模。在多数情况下，它包括对类、接口、组件和节点的具体的或原型化的实例以及它们之间传递的消息进行建模，所有这些都位于一个表达行为的脚本的语境中。交互图可以单独使用，来可视化、详述、构造和文档化一个特定的对象群体的动态方面，也可以用来对一个用例的特定的控制流进行建模。

交互图不仅对系统的动态方面建模是重要的，而且对通过正向和逆向工程构造可执行的系统也是重要的。

在绘制交互图时，可以参考如下原则。

- (1) 给出一个能表达其目的的名称。
- (2) 如果想强调消息的时间顺序，则使用时序图，如果想强调参加交互对象的组织结构，则使用协作图。
- (3) 合理的摆放元素以尽量减少线的交叉。
- (4) 用注解和颜色作为可视化提示，以突出图形中重要的特征。
- (5) 尽量少使用分支，用活动图来表示复杂的分支要更好些。

#### 58. 生命线 ( lifeline )

生命线代表时序图中的对象在一段时期内的存在。每个对象底部中心都有一条垂直的虚线，这就是对象的生命线，对象间的消息存在于两条虚线间。

#### 59. 状态 ( state )

状态是状态机的重要组成部分，它描述了状态机所在对象的动态行为的执行所产生的结果。这里的结果一般是指能影响此对象对后续事件响应的结果。

## 语义

状态描述了一个类对象生命期中的一个时间段。它可以用 3 种附加方式说明：在某些方面性质相似的一组对象值；一个对象等待一些事件发生时的一段时间；对象执行持续活动时的一段时间。虽然状态通常是匿名的并仅用处于该状态时对象进行的活动描述，但它也可以有名字。

在状态机中，一组状态由转换相连接。虽然转换连接着两个状态（或多个状态，如果图中含有分支和结合控制），但转换只由转换出发的状态处理。当对象处于某种状态时，它对触发状态转换的触发器事件很敏感

一个完整的状态有 5 个组成部分。

### （1）名字（name）

状态的名字由一个字符串构成，用以识别不同的状态。状态可以是匿名的，即没有名字。状态名一般放置在状态图的顶部。

### （2）入口/出口动作（entry/exit action）

入口/出口动作表示进入/退出这个状态所执行的动作。

### （3）内部转换（internal transition）

内部转换是不会引起状态变化的转换，此转换的触发不会导致状态的入口/出口动作被执行。定义内部转换的原因是有时候入口/出口动作显得是多余的。

### （4）延迟事件（deferred event）

延迟事件该状态下暂不处理，但将推迟到该对象的另一个状态下的事件处理队列。也就是所延迟事件是事件的一个列表，此列表内的事件在当前状态下不会处理，在系统进入其他状态时再行处理。

具有某些动态行为的对象在运行过程中，在某个状态下，总会有一些事件被处理，而另一些事件不能被处理。但对于这个对象来说，有些不能被处理的事件是不可以被忽略的，它们会以队列的方式被缓存起来，等待系统在合适的状态下在处理它们。对于这些被延迟的事件，可以使用状态的延迟事件来建模。

### （5）子状态（substate）

在复杂的系统中，当状态机处于特定的状态时，状态机所在的对象在此刻的动态行为可以使用另一个状态机来描述。也是说在状态的内部构成另一个状态机。

## 表示法

在 UML 中，图形上每一个状态图都有一个初始状态（实心圆），用来表示状态机的开始，还有一个终止状态（半实心圆），用来表示状态机的终止，其他的状态用一个圆角的矩形表示。

状态名称部分。容纳状态的（可选）名称作为一个字符串。没有名称的状态是匿名的，而且互不相同。但是不能在同一个图里重复相同的命名状态符号，因为易使人混淆。

内部转换部分。容纳一系列活动或者动作。这些活动或者动作是在对象处于该状态时，接收到事件而作出响应执行的，结果没有状态改变。

内部转换具有下面的形式：事件名称（参数表）[监护条件]/动作表达式。

动作表达式可以使用拥有对象的属性和连接以及进入转换的参数（如果它们出现在所有的进入转换里）。

参数列表（包括圆括弧）如果没参数就可以省略。监护条件（包括方括弧）和动作表达式（包括斜杠）是可选的。

进入和退出动作使用相同的形式,但是它们使用不能用作事件名称的相反的词汇:进入和退出。进入和退出动作不能有参数或者监护条件。为了在进入动作上获得参数,当前事件可以通过动作访问。这一点在获得新对象的生成参数时特别有用。

#### 60. 子状态 (sub state)

子状态被定义成包含在某一状态内部的状态。

##### 语义

在复杂的应用中,当状态机处于某特定的状态时,状态机所在的对象在此刻的行为还可以用一个状态机来描述,也就是说,一个状态内部还包括其他状态。在 UML 里子状态被定义成状态的嵌套结构,即包含在某状态内部的状态。

在 UML 里,包含子状态的状态被称为复合状态 (composite state),不包含子状态的状态被称为简单状态 (simple state)。子状态以两种形式出现:顺序子状态 (sequential substate) 和并发子状态 (concurrent substate)。

##### (1) 顺序子状态 (sequential substate)

如果一个复合状态的子状态对应的对象在其生命期内任何时刻都只能处于一个子状态,即不会有多个子状态同时发生的情况,这个子状态被称为顺序子状态。

当状态机通过转换从某状态转入复合状态时,此转换的目的可能是这个复合状态本身,也可能是复合状态的子状态。如果是前者,状态机所指的对象首先执行复合状态的入口动作,然后子状态进入初始状态并以此为起点开始运行。如果此转换的目的是复合状态的子状态,复合状态的入口动作首先被执行,然后复合状态的内嵌状态机以此转换的目标子状态为起点开始运行。

##### (2) 并发子状态 (concurrent substate)

如果复合状态内部只有顺序子状态,那么这个复合状态机只有一个内嵌状态机。但有时可能需要在复合状态中有两个或多个并发执行的子状态机。这时,称复合状态的子状态为并发子状态。

顺序子状态与并发子状态的区别在于后者在同一层次给出两个或多个顺序子状态,对象处于同一层次中,来自每个并发子状态的一个顺序状态中。当一个转换所到的组合状态被分解成多个并发子状态组合时,控制就分成与并发子状态对应的控制流。有两种情况下控制流会汇合成一个。第一,当一个复合状态转出的转移被激发时。这时,所有的内嵌状态机的运行被打断,控制流会汇合成一个。对象的状态从复合状态转出。第二,每个内嵌状态机都运行到终止状态。这时,所有的内嵌状态机的运行被打断,但对象还处于复合状态。

#### 61. 历史状态 (history state)

历史状态代表上次离开组合状态时的最后一个活动子状态。

##### 语义

在一个组合状态中所包含的一个由顺序子状态构成的子状态机中,必定有一个子初始状态。每次进入该组合状态,被嵌套的子状态机从它的子初始状态开始运作(除非直接转移到特定的子状态)。

当离开一个组合状态后,又更新进入该组合状态,但是不希望从它的初始状态开始运作,而是直接进入上次离开该组合状态时的最后一个子状态。这种情况下,使用一般的状态图或

顺序状态来表达都不方便。这就需要使用历史状态的概念。

历史状态代表上次离开组合状态时的最后一个活动子状态。每当转移到组合状态中的历史状态时，对象便恢复上次离开该组合状态时的最后一个活动子状态，并执行入口动作。

历史状态只是一个伪状态的图形标记，只能作为一个组合状态中的子状态，不能在顶层状态图中使用。历史状态可以有任意个从外部状态来的入转移，至多有一个无标签的出转移，它进入到一个子状态机。

### 表示法

浅历史状态用带有 H 的小圆圈表示，如图 A-30 所示。深历史状态用带有 H\* 的圆圈表示。

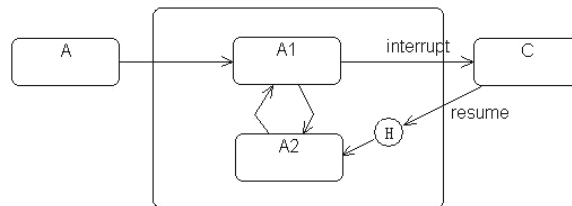


图 A-30 历史状态

## 62. 对象图 ( object diagram )

对象图 ( object diagram ) 是表示在某一时间上一组对象以及它们之间的关系的图。对象图可以被看作是类图在系统某一时刻的实例

### 语义

对象图除了描述对象以及对象间的连接关系外，还可包含标注和约束。如果有必要强调与对象相关类的定义，还可以把类描绘到对象图上。当系统的交互情况非常复杂时，对象图还可包含模型包和子系统。

和类图一样可以使用对象图对系统的静态设计或静态进程视图建模，但对象图更侧重于现实或原型实例。这种视图主要支持系统的功能需求，也就是说，系统提供给其最终用户的服务。对象图描述了静态的数据结构。

### 表示法

在图形上，对象图由节点以及连接这些节点的连线组成，节点可以是对象也可以是类，连线表达对象间的关系。

## 63. 用例 ( use case )

用例是对一个系统或一个应用的一种单一的使用方式所作的描述，是关于单个活动者在与系统对话中所执行的处理行为的陈述序列。

### 语义

用例是一个叙述型的文档，用来描述一个参与者 ( Actor ) 使用系统完成某个事件时的事情发生顺序。用例是系统的使用过程。更确切的说，用例不是需求或者功能的规格说明，但用例也展示和体现出了其所描述的过程中的需求情况。

从这些定义可知，用例是对系统的用户需求 ( 主要是功能需求 ) 的描述，用例表达了系统

的功能和所提供的服务。

用例描述活动者与系统交互中的对话。例如，活动者向系统发出请求，做某项数据处理，并向系统输入初始数据，系统响应活动者的请求，进行所要求的处理，把结果返回给活动者。这种对话表达了活动者与系统的交互过程，它可以用一系列的步骤来描述。这些步骤构成一个“场景”(Scenario)，而“场景”的集合就是用例。全部的用例构成了对于系统外部是可见的描述。

在识别用例的过程中，通过以下几个问题可以帮助系统开发人员识别用例。

- (1) 特定参与者希望系统提供什么功能？
- (2) 系统是否存储和检索信息？如果是，这个行为由哪个参与者触发？
- (3) 当系统改变状态时，通知参与者吗？
- (4) 存在影响系统的外部事件吗？
- (5) 是哪个参与者通知系统这些事件？

### 表示法

图形上用例用一个椭圆来表示，用例的名字可以书写在椭圆的内部或下方。

#### 64. 用例图 (use case diagram)

用例图是用例的可视化工具，它提供计算机系统的高层次的用户视图，表示以外部活动者的角度来看系统将是怎样使用的。

### 语义

UML 中的用例图是对系统的动态方面建模的 5 种图之一。用例图主要用于对系统、子系统或类的行为进行建模。每张图都显示一组用例、参与者以及它们之间的关系。用例图用于对一个系统的用例视图建模。多数情况下包括对系统、子系统或类的语境建模，或者对这些元素的行为需求建模。

对系统语境建模。在 UML 建模过程中，系统分析师可以使用用例图对系统的语境进行建模，强调系统外部的参与者。系统语境是由那些处于系统外部并且与系统进行交互的事物所构成。语境定义了系统存在的环境。

对需求建模。软件需求就是根据用户对产品的功能的期望，提出产品外部功能的描述。需求分析师的工作是获取系统的需求，归纳系统所要实现的功能，使最终的软件产品最大限度的贴近用户的要求。需求分析师的一般只考虑系统做什么 (what)，而尽可能的不去考虑怎么做 (how)。UML 用例图可以表达和管理系统大多数的功能需求。

### 表示法

参与者用人形图标表示，用例用椭圆形符号表示，连线描述它们之间的关系。

### 示例

图 A-31 所示是一个图书管理系统中读者得到服务的用例图。其中系统名称为图书管理系统；Librarian (图书馆工作人员) 和 Borrower (读者) 是参与者；箭头表示参与者之间、参与者与用例之间或用例与用例之间的关联；Return of Item、Lend Item、Remove Reservation 和 Make Reservation 都是用例名，分别代表还书用例、借书用例、删除预留书籍用例和预留书籍用例。

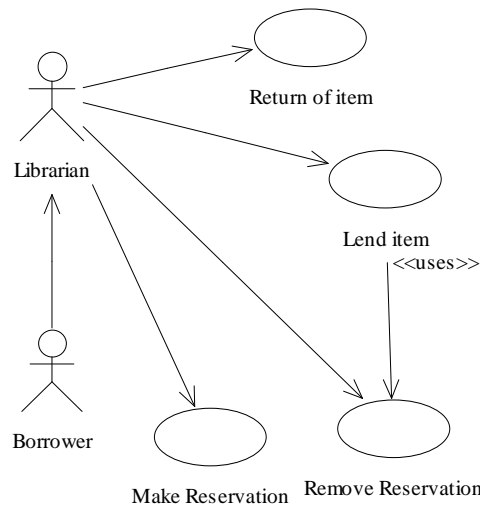


图 A-31 用例图

### 65. 用例视图 (use case view)

用例视图是松散地成组的建模概念中的一部分，就像静态视图。当用例视图在外部用户前出现时，它捕获到系统、子系统或类的行为。

## A.2 标准元素

标准元素是为约束、构造型和标签而预定义的关键字。它们代表通用效用的概念，这些通用效用没有足够的重要性或者与核心概念存在足够的差异用以包含在 UML 核心概念中。它们和 UML 核心概念的关系就如同内建的子例程库和一种编程语言的关系。它们不是核心语言的一部分，但它们是用户在使用这种语言时可以依赖的环境的一部分。列表中也包括了表示法关键字——出现在别的模型元素的符号上但代表的是内建模型元素而不是构造型的关键字。为关键字列出了表示法符号。

本节将交叉引用了上一节中的术语。

#### 1. 访问 (access)

(授权依赖的构造型)

两个包之间的构造型依赖，表示目标包的公共内容对于源包的名称空间是可以访问的。

#### 2. 关联 (association)

(关联端点的构造型)

应用于关联端点 (包括链接的端点和关联角色的端点) 上的一个约束，声明对应的实例是通过一个实际的关联可见的，而不是像参数或者局部变量一样通过暂时链。

#### 3. 变成 (become)

(流联系的构造型)

其源和目标代表不同时间点的相同实例的构造型依赖，但源和目标有潜在的不同的值，状

态实例和角色。从 A 到 B 的一个变成依赖意味着在时间/空间上的一个不同的时刻实例 A 变成 B，可能具有了新的值，状态实例和角色。变成的表示法是一个从源到目标的一个虚箭头，带有关键字 become。

#### 4. 绑定 (bind)

(依赖符号上的关键字)

代表绑定联系的依赖上的关键字。它后面跟着由括弧括起逗号分割的参数列表。

#### 5. 调用 (call)

(使用依赖的构造型)

源是一个操作而目标也是一个操作的构造型依赖。调用依赖声明源操作激发目标操作。调用依赖可以把源操作连接到范围内的任何目标操作，包括封闭类元和别的可见类元的操作，但不限制在这些操作上。

#### 6. 完全 (complete)

(泛化上的约束)

应用于一个泛化集合的约束，声明所有的孩子已经被声明（尽管有的可能被省略）而附加的孩子不应该在后面声明。

#### 7. 复制 (copy)

(流联系的构造型)

源和目标是不同的实例，但有相同的值，状态实例和角色的构造型流联系。从 A 到 B 的复制依赖意味着 B 是 A 的一个精确复本。A 中的特征改变不必反映到 B 中。复制表示法是一个从源到目标的箭头，带有关键字 copy。

#### 8. 创建 (create)

(行为特征的构造型)

一个构造型行为特征，表示指定的特征生成该特征所附的类元的一个实例。

(事件的构造型)

一个构造型事件，该事件表示生成一个实例，该实例封装了事件类型所作用的状态机。创建只能作用于该状态机顶层的初始转换。实际上，这是唯一可以作用于初始转换的触发源。

(使用依赖的构造型)

创建是表示客户类元创建了提供者类元的实例的构造型依赖。

#### 9. 派生 (derive)

(抽象依赖的构造型)

源和目标通常是相同类型的构造型依赖，但不一定总是相同类型。派生依赖声明源可以从目标计算得到。源可以为设计原因而实现，尽管逻辑上它是冗余的。

见派生 (derivation)。

#### 10. 销毁 (destroyed)

(行为特征的构造型)

表明指定的特征销毁了该特征所附的类元的一个实例的构造型行为特征。



(事件构造型)

表示封装了事件类型所作用的状态机的实例被销毁的构造型事件。

11. 被销毁的 (destroyed)

(类元角色和关联角色上的约束)

表示角色实例在封闭交互执行开始时存在而在执行结束之前被销毁。

12. 互斥 (disjoint)

(泛化上的约束)

作用于泛化集合的约束,声明对象不能是该泛化集合里的多个孩子的实例。这种情况只会出现在多继承中。

见泛化 (generation)。

13. 文档 (document)

(组件的构造型)

代表一个文档的构造型组件。

见 22. 组件 (component)。

14. 文档编制 (documentation)

(元素上的标签)

对所附的元素进行的注释,描述或解释。

见注释 (comment)。

15. 枚举 (enumeration)

(类元符号上的关键字)

枚举数据类型的关键字,它的细节声明了由一个标识符集合构成的域,那些标识符是该数据类型的实例的可能取值。

16. 可执行 (executable)

(组件的构造型)

代表一个可以在某结点上运行的程序的构造型组件。

见组件 (component)。

17. 扩展 (extend)

(依赖符号上的关键字)

表示用例之间的扩展联系的依赖符号上的关键字。

18. 虚包 (facade)

(包的构造型)

只包含对其他包所具有的元素进行的引用的构造型包。它被用来提供一个包的某些内容的公共视图。虚包不包含任何自己的模型元素。

见包 (package)。

## 19. 文件 (file)

(组件的构造型)

文件是一个代表包含源代码或者数据的文档的构造型组件。

见组件 (component)。

## 20. 框架 (framework)

(包的构造型)

包含模式的构造型包。

见包 (package)。

## 21. 友员 (friend)

(授权依赖的构造型)

一个构造型依赖,它的源是一个模型元素,如操作、类或包,而目标是一个不同的包模型元素,如类或包。友员联系授权源访问目标,不管所声明的可见性。它扩展了源的可见性,使目标可以看到源的内部。

## 22. 全局 (global)

(关联端点的构造型)

应用于关联端点(包括链接端点和关联角色端点)的约束,声明由于和链接另一端的对象相比,所附的对象具有全局范围而可见。

见关联 (association) 关联端点 (association end)。

## 23. 实现 (implementation)

(泛化的构造型)

一个构造型泛化,它表示客户继承了提供者的实现(它的属性,操作和方法)但没有把提供者的接口公共化,也不保证支持这些接口,因此违反可替代性。这是私有继承。

见泛化 (generalization)。

## 24. 实现类 (implementationClass)

(类的构造型)

一个构造型类,它不是一个类型,它代表了某种编程语言的一个类的实现。一个对象可以是一个(最多一个)实现类的实例。相反,一个对象可以同时是多个普通类的实例,随时间得到或者丢失类。实现类的实例也可以是零个或者多个类型的实例。

## 25. 隐含 (implicit)

(关联构造型)

关联的构造型,说明该关联没有实现(只是概念上)。

见关联 (association)。

## 26. 导入 (import)

(授权依赖的构造型)

两个包之间的构造型依赖,表示目标包的公共元素加到源包的名称空间里。

## 27. 包含 (include)

(依赖符号上的关键字)

表示用例之间的包含联系的依赖符号上的关键字。

## 28. 不完整 (incomplete)

(泛化上的约束)

不完整是应用于一个泛化集合的约束,它声明不是所有的孩子都已经被声明,后面还可以增加额外的孩子。

见泛化 (generalization)

## 29. .....的实例 (instanceOf)

(依赖符号上的关键字)

其源是一个实例而目标是一个类元的源联系。从 A 到 B 的这种依赖意味着 A 是 B 的一个实例。它的表示方法是一个带有关键字 instanceOf 的虚箭头。

## 30. 实例化 (instantiate)

(使用依赖的构造型)

类元之间的构造型依赖,这些类元表明客户的操作创建提供者的实例。

## 31. 不变量 (invariant)

(约束的构造型)

必须附在类元或联系集合上的一个构造型约束。它表明必须为类元或联系以及它们的实例维持约束的条件。

## 32. 叶 (leaf)

(泛化元素和行为特征的关键字)

不能有后代也不能被重载的元素,即不是多态的元素。

## 33. 库 (library)

(组件的构造型)

代表静态或者动态库的一个构造型组件。

见组件 (component)

## 34. 局部 (local)

(关联端点的构造型)

关联端点,链接端点或者关联角色端点的构造型,它声明所附的对象在另一端对象的局部范围里。

见关联 (association) 关联端点 (association end)

## 35. 位置 (location)

(类元符号上的标签)

支持该类元的组件。

(组件实例符号上的关键字)

组件实例所在的结点实例。

见组件 ( component )

### 36. 元类 ( metaclass )

( 类元的构造型 )

一个构造型类元, 它表明这个类是某个其他类的元类。

### 37. 新 ( new )

( 类元角色和关联角色上的约束 )

表示角色的实例在封闭交互执行过程中被创建, 在执行结束后依然存在。

### 38. 重叠 ( overlapping )

( 泛化上的约束 )

应用于一个泛化集合的约束, 它声明一个对象可以是泛化集合里的多个孩子的实例。这种情况只在多继承或多类元中出现。

见泛化 ( generalization )

### 39. 参数 ( parameter )

( 关联端点的构造型 )

关联端点 ( 包括链接端点和关联角色端点 ) 的构造型, 它声明所附的对象是对另一端对象操作的调用的一个参数。

### 40. 持久 ( persistence )

( 类元、关联和属性上的标签 )

表示一个实例值是否比生成它的过程存在得更久。值是持久或者是暂时的。如果持久被用在属性上, 就可以更好地确定是否应该在类元里保存属性值。

### 41. 后置条件 ( postcondition )

( 约束的构造型 )

必须附在一个操作上的构造型约束。它表示在激发该操作之后必须保持该条件。

### 42. 强类型 ( powertype )

( 类元的构造型 )

一个构造型类元, 它表示该类元是一个元类, 该元类的实例是别的类的子类。

( 依赖符号上的关键字 )

其客户是一个泛化集合而提供者是一个强类型的联系。提供者是客户的强类型。

### 43. 前置条件 ( precondition )

( 约束的构造型 )

必须附在一个操作上的构造型约束, 在激发该操作时该条件必须被保持。

### 44. 过程 ( process )

( 类元的构造型 )

一个构造型类元, 它是一个代表重量进程的主动类。

见主动类 ( active class ) 过程 ( process ) 线程 ( thread )

## 45. 细化 (refine)

(抽象依赖上的构造型)

代表细化联系的依赖上的构造型。

## 46. 需求 (requirement)

(注释的构造型)

声明职责或义务的构造型注释。

## 47. 职责 (responsibility)

(注释上的构造型)

类元的协议或者义务，它表示为一个文本字符串。

## 48. 自身 (self)

(关联端点的构造型)

关联端点(包括链接端点和关联角色端点)的构造型,声明一个从对象到其自身的伪链接,目的是为了在交互中调用作用在相同对象上的操作。它没有暗含实际的数据结构。

## 49. 语义 (semantics)

(类元上的标签)

对类元含义的声明。

(操作上的标签)

对操作含义的声明。

## 50. 发送 (send)

(使用依赖的构造型)

其客户是一个操作或类元,其提供者是一个信号的构造型依赖,它声明客户发送信号到某个未声明目标。

## 51. 构造型 (stereotype)

(类元符号上的关键字)

用来定义构造型的关键字。其名称可能用作别的模型元素上的构造型名称。

## 52. 桩 (stub)

(包的构造型)

一个构造型包,它代表一个只提供另外一个包的公共部分的包。

注意该词汇也被 UML 用来描述桩转换。

见包 (package)。

## 53. 系统 (system)

(包的构造型)

包含系统模型构成的集合的构造型包,它从不同的视点描述系统,不一定互斥——在系统声明中的最高层构成物。它也包含了不同模型的模型元素之间的联系和约束。这些联系和约束没有增加模型的语义信息。相反它们描述了模型本身的联系,例如需求跟踪和开发历史。一个系统可以通过一个附属子系统构成的集合来实现,每个子系统有独立的系统包里的模型集合来

描述。一个系统包只能包含在另一个系统包里。

见包 ( package )

#### 54. 表 ( table )

( 组件的构造型 )

代表一个数据库表的构造型组件。

见组件 ( component )

#### 55. 线程 ( thread )

( 类元的构造型 )

一个主动类的构造型类元，它代表一个轻量控制流。

注意在本书中该词汇具有更广的含义，可以表示任何独立并发的执行。

#### 56. 跟踪 ( trace )

( 抽象依赖的关键字 )

代表跟踪联系的依赖符号上的关键字。

#### 57. 暂时 ( transient )

( 类元角色和关联角色上的约束 )

声明角色的一个实例在封闭交互的执行过程中被创建，但在执行结束之前被销毁。

#### 58. 类型 ( type )

( 类的构造型 )

和应用域对象的操作一起声明实例 ( 对象 ) 的域的构造型类。一个类型不可能包含任何方法，但是它可以有属性和关联。

#### 59. 使用 ( use )

( 依赖符号上的关键字 )

代表使用联系的依赖符号上的关键字。

#### 60. 效用 ( utility )

( 类元的构造型 )

没有实例的构造型类元，它描述了一个由类范围的非成员属性和操作构成的集合。

#### 61. 异或 ( xor )

( 关联上的约束 )

作用在由共享连到同一个类上的关联构成的集合上的约束，它声明任何被共享的类的对象只能有一个来自这些关联的链接。它是异或 ( 不是或 ) 的约束。

见关联 ( association )

## A.3 元模型

### A.3.1 简介

在文档 UML 语义中描述的 UML 元模型定义了使用 UML 表示对象模型的完整语义。而它本身是用元递归的方式定义的,即用 UML 记号表示法和语义的一个子集来说明自己。这样,UML 元模型用一种类似于把一个编译器用于编译自己的方式来自扩展。

此处提供了元模型体系结构的一些背景,并且定义了 UML 元元模型。体系结构的方法有以下几点优越性。

- (1) 通过建立一个体系结构基础,增加了 UML 元模型的严格性。
- (2) 它有助于对 UML 元模型中的核心元对象的进一步理解。
- (3) 它为今后对 UML 元模型的扩展定义奠定了体系结构基础。
- (4) 它提供了把 UML 元模型和其他基于四层元建模体系结构(如 OMG 元对象 Facility 工具,CDIF)的概念统一起来的体系结构基础。
- (5) 在许多情况下,被称为元模型技术的应用将基于元元模型,而不是元模型。例如,一个用于模型互换的转换传送格式应当基于一个元元模型,这个元元模型很容易映射到所涉及到的不同的元模型上。因而,有必要对元元模型加以适当的定义。

### A.3.2 背景

通常公认的元建模的概念框架基于一个四层的体系结构:

- (1) 元元模型(meta-metamodel);
- (2) 元模型(metamodel);
- (3) 模型(model);
- (4) 用户对象(user object)。

元元建模层(meta-metamodeling)构成了元建模(metamodeling)体系结构的基础结构。这一层的主要责任是定义描述元模型的语言。一个元元模型定义了这样一个模型,它比元模型具有更高的抽象级别,而且比它定义的元模型更加简洁。一个元元模型能够定义多个元模型,而每个元模型也可以与多个元元模型相关联。通常所说的相关联的元模型和元元模型共享同一个设计原理和构造,也不是绝对的准则。每一层都需要维持自己设计的完整性。在元元模型层上的元元对象的例子有元类、元属性和元操作。

一个元模型是一个元元模型的实例。元模型层的主要责任是定义描述模型的语言。一般来说,元模型比定义它的元元模型更加精细,尤其是当它们定义动态语义时。在元模型层上的元对象的例子如类、属性、操作和组件。

一个模型是一个元模型的实例。模型层的主要责任是定义描述信息论域的语言。在建模层上的对象的例子如 StockShare、askPrice、sellLimitOrder 和 StockQuoteServer。

用户对象是一个模型的实例。用户对象层的主要责任是描述一个特定的信息论域。在用户对象层的对象的例子如<Acme\_Software\_Share 98789>、654.56、sell\_limit\_order 和<Stock\_

Quote\_Svr 32123>。

对元建模层的描述的总结见表 A-1。

表 A-1 四层元建模体系结构

层	说明	例子
元元模型	元建模体系结构的基础构造，定义了描述元模型的语言	元类、元属性、元操作
元模型	元元模型的实例，定义了描述模型的语言	类、属性、操作、组件
模型	元模型的实例，定义了描述信息论域的语言	StockShare ,askPrice ,sell LimitOrder , Stock Quote Server
用户对象 ( 用户数据 )	模型的实例，定义了一个特定的信息论域	<Acme_Software_Share98789> , 654.56 , sell_limit_order , <Stock_Quote_Svr 32123>

元建模层之间的依赖关系以 UML 的方法表示如图 A-32 所示。

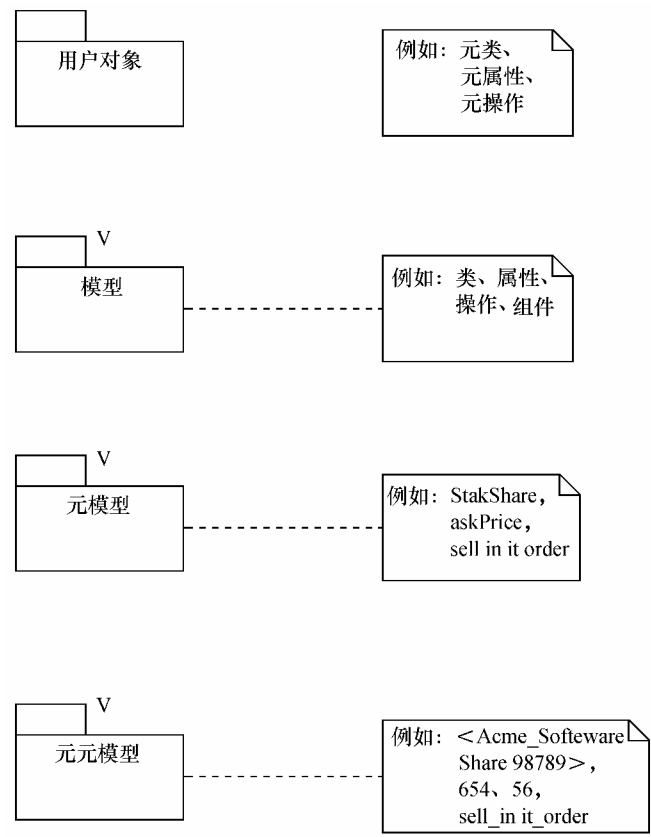


图 A-32 四层元建模体系结构



尽管元建模体系结构可以扩展成含有附加层的结构,但是这一般是没有用的。附加的元层(如元元建模层)之间往往很相似并且在语义上也没有明显的区别。因此,把讨论限定在传统的四层元建模体系结构上。

### A.3.3 元元模型

UML 元元模型描述基本的元元类型、元元属性和元元关系,这些都用于定义 UML 元模型。元元模型实现了下述要求的一个基本设计:强调使用少数功能较强的建模成分,而这些成分易于组合起来表达复杂的语义。尽管在本文档中元元模型的提出和定义 UML 元模型有关,但它被设计在一个方法和技术都相对独立的抽象层次上。它可以用于其他用途,例如定义库 Repositories 或者模型转换格式。

由于在四层元模型体系结构中元层次之间的关系是元递归的,需要某些基本的模型概念来定义元元模型本身。因此,假定已有以下的元元对象:元类型、元一般化、元关联、元角色和元属性。

与元建模的元递归特性是相一致的,这些概念将在后面的元元模型中加以说明。

UML 元元模型分成几部分加以描述。

元元对象继承层次(层次图、层次结构),提供元元对象在元元模型中的分类法。

元元对象。描述这样一些元元对象,它们用于定义元元模型结构上的和行为上的概念。

非对象类型。描述元元模型使用的原始数据类型。

UML 元元模型形成了 UML 元模型的元建模体系的基础结构。特别的,它为描述 UML 元模型的语言定义了语法和语义。它的基本设计要求强调使用少数功能较强的建模成分,而这些成分易于组合起来表达复杂语义。

在元元模型中定义的多数元元对象被看作结构上的定义。只有元操作和元参数被看作是行为上的定义。这些元元对象之所以被包含在元元模型中,是因为元操作是用于定义当前元模型的;行为化的元元对象对于表达动态语义是很重要的,而这种能力将随着建模的进展变得愈加重要。

## 参考文献

- 1 .James Rumbaugh ,Ivar Jacobson ,Grady Booch .UML 参考手册 .机械工业出版社 ,2001
- 2 . James Rumbaugh , Ivar Jacobson , Grady Booch . The Unified Modeling Language User Guide . Addison-wesley , 2001
- 3 . Ivar Jacobson , Grady Booch , James Rumbaugh . 软件统一开发过程 . 机械工业出版社 , 2002
- 4 . Karl E , Wieggers. software Requirements . Microsoft Press , 2000
- 5 . 齐治平 , 谭庆平 , 宁洪 . 软件工程 . 高等教育出版社 , 1997
- 6 . Wendy Boggs , Michael Boggs . Mastering UML with Rational Rose 2002 . SYBEX , 2002
- 7 . Jim Arlow , Ila Neustadt . UML 和统一过程 . 机械工业出版社 , 2003
- 8 . Alistair Cockburn . 编写有效用例 . 机械工业出版社 , 2002
- 9 . Craig Larman . UML 和模式应用 . 机械工业出版社 , 2002
- 10 . Sinan Si Alhir . UML 技术手册 . 中国电力出版社 , 2002
- 11 . Joseph Schmuller . UML 基础、案例与应用 . 人民邮电出版社 , 2002
- 12 . Klaus Bergner , Andreas Rausch , Marc Sihling . Using UML for Modeling a Distributed Java Application , 1997
- 13 . Carol Britton&Jill Doake . Object-Oriented Systems Development : a gentle introduction . McGraw-Hill International Limited , 2000
- 14 . UML whitepaper. <http://www.rational.com/uml/resource/whitepaper/index.jsp>
- 15 . <http://www.umlchina.com>

# 读者意见反馈表

亲爱的读者：

感谢您对我们的支持与爱护，为了更加深入地了解您的需求，以便向您提供更适合您阅读的图书，请抽出宝贵的时间填写这份调查表。填好后剪下寄到：北京市崇文区夕照寺街 14 号人民邮电出版社 406 汤倩（邮编 100061）。或者采用传真（010-67132692）、电子邮件（tangqian@ptpress.com.cn）的方式把您的意见及时反馈给我们。我们将选出意见中肯的热心读者，赠送本社的其他图书作为奖励。同时，我们也将充分考虑您的意见和建议，并尽可能地给您满意的答复。谢谢！

书名：《UML 基础与 Rose 建模案例》

个人资料 姓名：\_\_\_\_\_ 性别：\_\_\_\_\_ 年龄：\_\_\_\_\_  
E-mail：\_\_\_\_\_ 文化程度：\_\_\_\_\_ 职业：\_\_\_\_\_  
通信地址：\_\_\_\_\_ 邮编：\_\_\_\_\_ 电话：\_\_\_\_\_

1. 您是如何得知本书的：

别人推荐 \_\_\_\_\_ 书店 \_\_\_\_\_ 出版社图书目录 \_\_\_\_\_ 其他（请指明）\_\_\_\_\_

2. 您从何处得到本书的：

书店 \_\_\_\_\_ 邮购 \_\_\_\_\_ 图书销售的网站 \_\_\_\_\_ 其他（请指明）\_\_\_\_\_

3. 影响您购买本书的因素（可多选）：

书中的内容 \_\_\_\_\_ 工作、生活和学习的需要 \_\_\_\_\_ 价格 \_\_\_\_\_ 其他\_\_\_\_\_

4. 您对本书封面设计的满意程度：

很满意 \_\_\_\_\_ 比较满意 \_\_\_\_\_ 一般 \_\_\_\_\_ 不满意 \_\_\_\_\_ 改进建议\_\_\_\_\_

5. 您对本书的总体满意程度：

从文字的角度	很满意	比较满意	一般	不满意
从技术的角度	很满意	比较满意	一般	不满意

6. 您希望本书的定价是多少：

20 元以下 \_\_\_\_\_ 20 ~ 30 元 \_\_\_\_\_ 30 元以上 \_\_\_\_\_

7. 本书最令您满意的是：

\_\_\_\_\_

8. 您在使用本书时遇到哪些困难：

\_\_\_\_\_

9. 您希望本书在哪些方面进行改进：

\_\_\_\_\_

10. 您在使用 Rose 建模的过程中遇到了什么困难：

\_\_\_\_\_

11. 您需要购买哪些方面的图书，对我社现有图书有什么好的建议：

\_\_\_\_\_

12. 您更喜欢阅读哪些类型和层次的计算机书籍：

入门类      提高类      技巧类      实例类      精通类      综合类

13. 您的其他要求：\_\_\_\_\_

\_\_\_\_\_

-----  
您认为本书有哪些错误：

章节\_\_\_\_\_ 页码\_\_\_\_\_ 行\_\_\_\_\_ 图号\_\_\_\_\_ 错误\_\_\_\_\_ 应改为\_\_\_\_\_

章节\_\_\_\_\_ 页码\_\_\_\_\_ 行\_\_\_\_\_ 图号\_\_\_\_\_ 错误\_\_\_\_\_ 应改为\_\_\_\_\_

章节\_\_\_\_\_ 页码\_\_\_\_\_ 行\_\_\_\_\_ 图号\_\_\_\_\_ 错误\_\_\_\_\_ 应改为\_\_\_\_\_

章节\_\_\_\_\_ 页码\_\_\_\_\_ 行\_\_\_\_\_ 图号\_\_\_\_\_ 错误\_\_\_\_\_ 应改为\_\_\_\_\_