UML 应用建模实践过程

尤克滨　编著

机 械 工 业 出 版 社

本书立足工程实践，以应用 UML 进行面向对象分析和设计为主题，帮助软件工程师在排除关键障碍的基础上，通过推敲实例，有步骤地掌握一套切实可行的方法和流程。

全书分为三个部分。第一部分，基本理念和准备知识。是本书的铺垫。解释分析和设计模型的含义和价值，概述面向对象技术的内涵、优势和原则，介绍模型内容的组织和相关的 UML 表述。第二部分，UML 应用建模实践过程。是本书的核心。详细展现分析和设计过程中的 5 项任务，即全局分析、局部分析、全局设计、局部设计和细节设计。其中包括 14 项基础活动、39 个核心概念、30 个关键步骤、52 条实践技巧以及贯穿全程的示例。本书的实践过程遵循 Rational 统一过程(RUP)的核心思想和基本原则，即以 Use Case 驱动的、体系构架为核心的迭代化面向对象分析和设计过程。第三部分，设计模型的沿用。是本书内容的延伸。概要地介绍与设计模型直接相关的活动和内容，包括设计模型向实施模型的过渡、设计模型和数据模型的关联以及如何整理主要的设计文档。

本书立足实践者的视角，适合于应用面向对象技术的软件工程师，尤其是系统构架师和设计师。本书可以作为应用 UML 进行面向对象分析和设计的实践课程教材。

图书在版编目（CIP）数据

UML 应用建模实践过程/尤克滨编著．—北京：机械工业出版社，2003．1
ISBN 7-111-11436-1

Ⅰ．U… Ⅱ．尤… Ⅲ．面向对象语言，UML—程序设计 Ⅳ．TP312

中国版本图书馆 CIP 数据核字（2002）第 109104 号

机械工业出版社（北京市百万庄大街 22 号 邮政编码 100037）
策 划：胡毓坚
责任编辑：陈振虹
封面设计：于 鹏 王从然
责任印制：

印刷厂印刷·新华书店北京发行所发行

2003 年 1 月第 1 版第 1 次印刷
787mm×1092mm 1/16·14 印张·343 千字
0001-5000 册
定价：28.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
本社购书热线电话（010）68993821、88379646

前　言

目　标

如果你只热衷于作“程序员”而无意成为“软件工程师”，这本书并不适合你。如果你想通过本书成为面向对象技术专家，坦率讲，也没有可能。但是，如果你打算应用统一建模语言（Unified Modeling Language，UML）、采用面向对象技术分析和设计软件，这本立足于工程实践的书能够给你实实在在的帮助。

本书以应用 UML 进行面向对象分析和设计为主题，帮助软件工程师在排除关键障碍的基础上，通过推敲实例，有步骤地掌握一套切实可行的方法和流程。但愿本书能成为你前行道路上的几块垫路石或一座里程碑。

背　景

20 世纪 90 年代末期，UML 成为国际对象管理组织（Object Management Group，OMG）认可的标准建模语言。UML 的几位创始人是面向对象方法学的世界级大师，UML 的核心价值和优势深刻地体现在面向对象的方法和流程之中。采用 UML 规范地表述面向对象分析和设计过程，能够显著提升开发效率并保障可持续发展，是公认的发展趋势。掌握 UML 已成为软件工程师极具潜在价值的技能。

在国内，相关的教学研究和工程实践越来越受重视，对面向对象分析和设计方面图书的要求越来越迫切。基于教学和咨询的实践，作者了解国内软件开发人员在接受、掌握和应用 UML 与面向对象方法过程中存在的主要困惑和实际困难。例如基础知识缺陷对理解概念的影响，东西方思维模式差异对领悟过程的影响，以及传统思维对接受面向对象方法的影响…，这些壁垒导致广大的实践者对这项先进技术的理解表面化、片面化甚至神秘化，失去大量实践与提高的良机。

内　容

基于对民族软件工业的责任感，立足于实践者的视角，作者力求在书中突出

方法和流程的可用性与针对性，不追求博大精深和面面俱到，但强调目标明确和重点突出。全书分为以下三个部分。

第一部分，基本理念和准备知识，是本书的铺垫。首先解释分析和设计模型的含义和价值，然后概述面向对象技术的内涵、优势和原则，最后介绍模型内容的组织和相关的 UML 表述。

第二部分，UML 应用建模实践过程，是本书的核心。详细展现分析和设计过程中的五项任务，分别为全局分析、局部分析、全局设计、局部设计和细节设计。其中包括 14 项基础活动、39 个核心概念、30 个关键步骤、52 条实践技巧以及贯穿全程的示例。本书描述的实践过程充分借鉴了 Rational 统一过程（RUP）的核心思想和基本原则。RUP 汇集了众多的成功经验并逐步成为该领域的事实标准，它倡导以 Use Case 驱动的、体系构架为核心的迭代化面向对象分析和设计过程。通俗地讲，就是软件需求牵引的、既往经验支撑的和风险前驱的开发过程。

第三部分，设计模型的沿用，是本书内容的延伸。概要地介绍与设计模型直接相关的活动和内容，包括设计模型向实施模型的过渡、设计模型和数据模型的关联以及如何整理主要的设计文档。

读　　者

如果你是面向对象的程序员，你会发觉，缺乏面向对象分析和设计指引的面向对象编程只能得到似是而非的面向对象软件。本书帮助你将以往随机的分析和设计活动融入系统化的过程。在介绍 UML 重要语义的过程中，引用程序代码片段作为辅助诠释，帮助读者利用已有知识，快速建立更有价值的新概念。

如果你是应用面向对象技术的系统构架师或设计师，本书的核心内容正是你的主要职责。实践过程中详述了分析和设计任务从全局到局部再到细节的渐变和迭代，强调了系统构架师和设计师的不同侧重与配合，突出了软件需求在分析和设计活动中的牵引作用，说明了成熟设计经验用于简化和支撑分析和设计活动的价值。

对于项目经理和研发主管而言，UML 与面向对象技术提供了打破时间与质量之间僵局的机遇。本书的实践过程中，具体地说明如何在分析和设计活动中贯彻风险前驱的迭代化开发策略，明确地指出不同角色人员的基本素质要求，详细地介绍可操作的团队并行协作方式。

如果你读过《统一软件开发流程》、《设计模式》和《UML 用户指南》[1]，相信你学到的战略思想、战术策略和符号体系将给你带来更大的收获。当然，这不能脱离你严谨的态度、批判的眼光和不辍的实践。

---

[1] 三本书的原文名称分别为《The Unified Software Development Process》、《Design Patterns：Elements of Reusable Object-Oriented Software》和《The Unified Modeling Language User Guide》，Addison-Wesley 出版。

致　　谢

感谢 Ivar Jacobson 博士（UML 创始人之一）在建模问题上给作者高屋建瓴地点拨。和 Terry Quatrani 女士（UML 制定者之一）关于统一建模问题的讨论使作者获益非浅，在此表示深切感谢。

感谢参与审阅本书的朋友们。感谢来自著名企业的软件专家，他们是 IBM 的张晨曦、陈曦、梁朝东，Microsoft 的李峻，Rational Software 的吴穹，Motorola 的李玉山，Nortel 的白静原，用友软件的游志强，中国金融电子化公司的周夕崇，亚信公司的刘炜，中兴通讯的张雪敏。感谢 UMLCHINA 和《XProgrammer》的创始人潘家宇。感谢来自著名高校的老师，他们是北京航空航天大学软件学院的姚淑珍、张莉、林广艳，清化大学软件学院的刘强，王少峰，复旦大学软件学院的薛云蛟、吕钊、上海交通大学软件学院的步丰林、黄小平，西安交通大学软件学院的何亮、杨新宇，华中科技大学软件学院的胡迎松、陈中新，武汉大学软件学院的薛超英、胡启平。感谢所有为本书提出建议的朋友。

尤克滨

2003 年元旦 北京

目　　录

第一部分　基本理念和准备知识

一　脚踏实地

第 1 章　分析和设计的逻辑模型

1.1　模型在认知和求解中的价值

模型（Model）可以帮助我们在简约繁复的基础上，捕捉现实问题（Problem）的本质，勾勒软件方案（Solution）的雏形。模型有助于在问题到方案的过渡过程中更好地认知、理解和沟通。

模型是什么

简单讲，模型是现实的简化。准确讲，模型是能动的抽象认知结果，它对应一个认知活动的主体和认知活动的原则。认知活动的主体决定了特定的视角，可以理解为简化的动机；认知活动的原则决定了特定的抽象层次，可以理解为简化的水平。

对于一个系统，基于不同的简化动机和简化水平，可以得到多个模型，有助于更深刻和更准确地把握系统的本质。模型可以描述系统的静态结构，也可以描述系统的动态行为；模型可以描述系统的宏观面貌，也可以描述系统的微观情境。

模型是现实的简化，并不隐含时间顺序。也许是巧合，如果将“现实的简化”的语序反过来，则变成“化简的实现”，也同样成立。面向方案的设计模型会先于方案而存在，模型提供了营造方案的蓝图，也可以包括详尽的规划。

利用价值高的模型就是好模型，它们通常会忽略那些与特定抽象层次无关的次要因素，强调那些具有广泛影响力的主要因素。换言之，内容多的模型未必是好模型，因为价值高的内容有可能被价值不高的内容淹没。

模型是一组具有完整语义的信息，包括两个方面的内容：一方面，对现实的简化或者对实现的化简；另一方面，认知主体的视角和抽象层次。前者是被认知的客体，表现为各种类型的图（Diagram）及其包含的元素和关联；后者反映认知的主体，表现为不同类型的视图（View）。两者都是模型不可或缺的要素。

尽管强调模型的简化和化简价值，这并不意味着可以片面地夸大图示信息的能量，好的模型应该图文并茂，关键是可用和易用。

建模的价值

对于问题，模型是现实的简化，对于方案，模型是化简的实现。建模（Modeling）

是捕捉系统本质[1]的过程。

为了降低风险和获得高回报，建模活动普遍应用于各种行业，软件开发更不例外。为说明建模的价值，Grady Booch 曾经给出过一个经典的类比：盖一个宠物窝棚，修一个乡间别墅和建一座摩天大楼，三种工作对建筑规划图纸的依赖程度有质的差异。建立一个简单的系统，模型可有可无；建立一个比较复杂的系统，模型的必要性增大；建立一个高度复杂的系统，模型则不可缺少。应用处理简单系统的方法对待复杂系统通常行不通，这好比用搭建一个宠物窝棚的方法来营造一座摩天大厦。

建模的意义随着系统复杂程度的增加而越发显著，从起初借助模型更好地理解系统，到后来不得不借助模型来理解系统，充分说明了这一切。人脑对复杂问题的理解能力是有限的，与模型相应的特定视角和抽象层次是简化复杂问题的有效出发点。

当今，我们开发的软件，特别是商业软件，通常一开始就很不简单，并且，复杂性随着时间的演进和技术的发展持续上升。一个复杂软件系统的开发必须面对多种未知因素，多个开发人员，复杂的开发工具和永远不够用的时间。开发人员没有可能，更没有必要去了解从问题到方案的所有细节。他们需要那些基于特定视角的、有助于解决问题的、并且是完整的某一部分信息，即所谓的模型。总之，建模对于复杂软件系统的开发是必要的。

广义讲，无论出于何种动机，只要在问题到方案之间作出一些过渡性的努力，哪怕只是在草稿纸或是白板上画了几笔，实际上就是在建模了。不过，有意识和无意识的建模活动对模型质量或价值的影响很大。有意识的建模活动通常是有计划的、有准备的和早动手的，得到的模型通常是完整的、一致的和可复用的；无意识的建模活动通常是随机的、无准备的和补救性的，得到的模型往往是零散的、混乱的和一次性的。

准确地讲，建模活动直观地记录下认知和求解的过程，支持团队成员之间的有效沟通，为重复利用各个阶段积累的智力成果创造有利的条件。

概括地讲，建模简化了认知过程，化简了求解过程。如果读者希望进一步拓展对模型和建模的理解，可以参考《UML 用户指南》。

统一建模语言 UML

统一建模语言 UML，全称 Unified Modeling Language。

首先，简要地回顾一下 UML 的由来。20 世纪 90 年代初，很多面向对象的方法已经拥有自己的符号体系，其中有三种比较突出：Jim Rumbaugh 的 OMT 方法，Grady Booch 的 Booch 方法以及 Ivar Jacobson 的 OOSE 方法。不同的方法和符号体系各有所长：OMT 擅长分析，Booch 擅长设计，OOSE 则擅长业务建模。那个时期的面向对象技术人员远没有我们这么幸运，为了建立比较丰满的模型并进行

[1] Modeling captures essential parts of the system. -Dr. James Rumbaugh

有效的沟通，他们需要掌握不同的符号体系，并且花费一些精力去翻译和转述用不同符号体系记录的模型。在后来的几年中，上述三位大师在各自的著作中自然而然地融入了其他两种方法的技术内容。Jim Rumbaugh 于 1994 年离开 GE 加入 Grady Booch 所在的 Rational 公司，开始和 Grady Booch 协同研究一种统一的方法。一年后，Unified Method 0.8 诞生了。同年，Rational 收购了 Ivar Jacobson 所在的 Objectory 公司，Ivar Jacobson 从此也成为 Rational 的一员。Unified Method 不久更名为 UML，仰仗三位面向对象方法学大师的威望，基于数十位行业内重量级人物历时两年的通力合作，并充分考虑到多方合作伙伴的反馈意见，UML 一步步趋向成熟。1997 年 9 月，UML 1.1 被提交到国际对象管理组织，同年 11 月被该组织认定为标准的建模语言。

统一建模语言，顾名思义，有三个要点：统一（Unified）、建模（Modeling）和语言（Language）。

很难想像如果没有五线谱，指挥家和乐手如何奏出交响乐。软件开发活动对统一表述符号的要求是类似的，并且，软件开发活动的规模和复杂程度通常不亚于大型交响乐团的排练和演出。

“统一”是 UML 的核心，它提升了软件开发团队的沟通效率，节约了以往用于翻译和转述的开销，屏蔽了藏匿于含糊语义中的风险。

UML 表述的内容能被各类人员所理解：包括客户、领域专家、分析师、设计师、程序员、测试工程师以及培训人员等。他们可以通过 UML 充分地理解和表达自己所关注的那部分内容。在传统概念中，他们各自拥有专用的符号系统，这也是长期以来潜在的沟通壁垒。

除了在多工种人员之间形成统一，在工具辅助下，UML 还可以通过双向工程（Round-trip Engineering）实现开发人员和开发环境的统一。

“建模”体现了 UML 的使用价值。UML 在制定过程中汲取了多种建模方法的精华，包括业务建模和数据建模等。UML 的使用价值不可能脱离特定类型的建模活动。对于学习者而言，如果以掌握 UML 的符号和规则为最终目的，你将所获甚微。尽管 UML 所表述的内容可以贯穿软件开发的生命周期，但 UML 不同于普通的程序设计语言，仅仅掌握 UML，并不能得到实际的解决方案。

“语言”是 UML 普遍价值的表现。语言的一层基本含义是一套按照特定规则和模式组成的符号系统，被拥有相同传统和习惯的人群所使用。我们在日常生活中将“拥有共同语言”视作有效沟通的必要条件。近年来，软件开发所涉及的技术飞速发展，不同技术门类所使用的建模语言自成体系，同时也具有很大局限性，表现形式的差异往往掩盖了本质内容的相通。幸好，与人类的自然语言不同，在软件开发过程中使用的建模语言不涉及宗教和文化等诸多历史或地域障碍。在博采众长的基础上，UML 作为一种共通的和可扩展的语言，其描述能力适用于软件开发中各种技术门类的建模活动。自然语言是人类对客观世界建模最直接有效的表述形式；类似地，UML 是迄今软件开发人员进行统一建模活动最直接有效的表述形式。不仅如此，UML 还是能被软件开发环境所理解的语言。

1.2 分析和设计的对立、关联和统一

客户根据业务需要（Needs）提出软件需求（Software Requirements），开发人员根据技术环境实施程序代码（Code），分析和设计是需求到代码之间平滑过渡的桥梁。概括地讲，分析是一个翻译软件需求和深入理解问题的过程，设计是一个逐步精化方案和适应实施环境的过程。尽管二者的基本目标对立，但两种思辨过程却紧密关联，为了在对立的目标和关联的思辨过程之间建立明确的映射和平滑的过渡，我们需要统一的思想原则、统一的表述形式以及融合思想原则和表述形式的统一实践过程。

目标的对立

分析的目标是理解问题并开发一个简要描述方案的可视化模型，不依赖于具体的实施技术环境。分析活动回答“要做什么”的问题，工作的重点是将功能性需求翻译成软件的概念，或者说用软件的概念来诠释问题所要求的功能。分析的核心是捕获问题的行为，在屏蔽实施细节的基础上得到构成方案的粗略对象模型。

设计的目标是精化方案并开发一个明确描述方案的可视化模型，保障设计模型最终能平滑地过渡到程序代码。设计活动回答“该怎么做”的问题，工作的重点是适应特定的实施环境和部属环境。设计的核心是规划方案的构造，在揭示实施细节的基础上得到方案的详细对象模型。实施环境指构建软件的环境，部属环境指运行软件的环境。

分析面向问题，是明确动力的过程，重在理解和翻译，灵活性高；设计面向方案，是排除阻力的过程，重在精化和适应，受约束大。两种活动的对立是客观的。因而，没有不存在的对立，只有未被意识到的对立。从整体上看，分析和设计的对立是保障问题和方案趋于一致的基本动力。开发一个好软件的基本任务之一就是让这种对立更明确地表现出来，更好地被认识和利用。

思辨过程的关联

人们从问题到方案的认知和求解思辨过程非常复杂，分析和设计的交错演进客观存在，试图将两种活动截然分开通常会带来不利影响。

有时，提出阶段性方案的目的就是为了更好地理解问题。在实际的建模过程中，一开始从特定的要点入手，创建简单的对象模型去模拟现实问题，即所谓的分析；然后，从分析的结果出发，以一些通用的元素为基础，创建较为复杂的对象模型来规划解决方案，即所谓的设计。设计的价值不仅在于它给出了一个方案空间对问题空间的映射，同时，设计往往展现出当前对问题理解的深度和精度的不足，这些具有启发价值的信息有助于在更明确的范围内展开进一步的分析活动。

当掌握了一些设计的经验和技巧之后，在分析活动中能够有意识地作出合理的假定，屏蔽和推延一些设计细节的出现，更好地保障分析和问题之间的简明对

应关联。

即便是解决一个简单的问题，人的归纳和演绎思维也可能经历几个反复的过程。在复杂软件系统的分析和设计过程中，思维的关联和频繁交迭是必然的，应当有意识地把握和利用这种关联。缺乏分析指导的设计和缺乏设计支撑的分析都不可取。希望读者不要误解，有意识地关联两种活动并不意味分析的时候要考虑很多设计的问题或者设计的时候要考虑很多分析的问题；相反，这样做的目的是希望开发人员在分析的时候能够主动地意识到哪些设计问题暂时无须考虑，或者在设计的时候能够主动地意识到哪些分析问题需要作出折衷。

建模的根本目标是捕获系统的本质，有意识地关联分析和设计的思辨过程是接近本质的有效途径。

原则、形式和过程的统一

总体上，分析面向问题，设计面向方案，开发软件的目标是用方案去映射并解决问题。如果是规模很小的短时间一次性开发，实现方案和问题在整体上的映射是可行的。但是，面对一个复杂软件系统的开发，仅仅实现方案和问题在整体上的映射并不充分。在问题和方案的局部之间甚至是重要细节之间同样要保持明确和稳定的映射关联。我们应该能够按部就班地将这种映射关联清晰地表述出来。更重要的是，这种映射关联的稳定性随着时间和规模的变化不应该发散。这要求分析和设计活动应该基于统一的原则、形式和过程。

首先是统一的原则。应用传统的结构化分析和设计方法，最大的困难在于保持问题域和方案域之间明确和稳定的映射关联。相反，这正是面向对象方法的核心优势，理论和实践都表明，面向对象的原则可以保证问题要素和方案要素之间形成稳定的映射。

第二是统一的表述形式。在没有统一的建模语言之前，不同建模语言对同一内容的表述存在差异，致使开发人员付出很多额外努力去发掘那些从问题到方案过渡中相通的本质。为了更加明确而有效地描述问题到方案的映射关系，在形式上，分析和设计有必要使用同一种建模语言。统一建模语言 UML 是国际对象管理组织认可的标准，是面向对象分析和设计的首选建模语言。UML 不仅支持分析和设计的建模活动，同时支持生命周期中的其他建模活动，UML 可用于描述业务模型、需求模型、应用模型以及数据模型。值得注意的是，统一的表述形式主要解决效率问题。如果没有基本的效率保障，面向对象分析和设计方法的回报将大打折扣，甚至得不偿失。基于效率的眼光，发挥 UML 的价值在很大程度上要借助工具的支撑。统一建模语言 UML 并不解决根本问题，如果开发人员以统一建模语言作为出发点而忽视面向对象的思想原则，得到的将是一个似是而非的结果。

最后是统一的过程。如果是一个人开发非常简单的软件，统一过程的价值很小。但是，如果开发复杂的软件系统，会有很多人参与协作，系统规模还可能持续地扩张。建模活动仅仅依靠统一的原则和形式尚不充分，统一的分析和设计过程将是成功的必要条件之一。面向对象的基本原则和统一建模语言已经得到广泛

的认可，但面向对象分析和设计的方法流派众多并且还在不断完善。实践中，理解面向对象原则和统一建模语言之后，开发人员往往仍旧找不到着力点：搞不清楚先做什么，后做什么；参照什么信息，得到什么结果；自己该做什么，别人该做什么……统一过程的核心就是解决可操作性问题，帮助开发人员尽可能少地依赖那些“不可描述的经验”。我们可以直接借鉴 Rational 统一过程（RUP）中以 Use Case 驱动的、体系构架为核心的迭代化面向对象分析和设计过程，它是经过实践反复验证的成功经验和该领域的事实标准。当然这并不意味着我们必须全盘接受 RUP，而是借鉴其分析和设计工作流程中的核心思想和关键概念。第一是“Use Case 驱动”。用 Use Case 作为划分问题的组织单元，分析和设计活动的局部粒度都遵照这一划分原则。Use Case 的定义反映了系统外部要素根据特定目标使用拟建系统的状况，能确保问题的局部划分不过于粗略，也不过于琐碎，保持了全局焦点和局部焦点的平衡。第二是“体系构架为核心”。这样做能确保方案从一开始就具备高内聚和低耦合的可持续成长构架。第三是“迭代化的过程”。基于风险前驱的原则，渐进地展开分析、设计及其相关活动，每个迭代都会提供一次验证和调整模型的机会，推动软件质量的提升。面向对象的方法和流程多种多样，对于个人和开发团队而言，关键是成功地应用一种方法。鉴于共通的面向对象思想原则，只要真正掌握一种方法和流程，就不难做到举一反三，触类旁通。

综上，我们需要统一的原则、统一的语言和统一的过程。其中，统一的过程往往是开发团队的盲区或薄弱环节，它也是面向对象方法与分析和设计实践相结合的要素。本书的第二部分将以统一的实践过程为核心线索，介绍如何应用统一的建模语言（UML）贯彻统一的面向对象原则，建立分析和设计的逻辑模型，勾勒问题到方案之间的平滑过渡。

1.3 分析和设计的逻辑模型

广义上，一个完整的软件开发过程中会涉及不同种类的模型。针对不同的研究目标和阶段，可以分成四大类：业务模型（Business Model）、需求模型（Requirements Model）、数据模型（Data Model）和应用模型（Application Model）。每类模型又可能包含概念，逻辑和物理等不同的层次。本书讨论的分析和设计的逻辑模型是应用模型的逻辑层次内容。当然应用模型还包括物理层次的内容，比如代码模块的结构（Component Diagram）或者应用程序在网络节点上的分布（Deployment Diagram）等。

如果您认为建模是必要的，那么在明确软件需求到编写程序代码之间，会有意识地去完成很多工作。建立并完善分析和设计的逻辑模型是这些工作中最有价值、最有挑战性的部分。

如果将分析和设计的逻辑模型视为一个整体，那么它的主要依据来源于需求模型，有时也要参照业务模型和数据模型，分析和设计的逻辑模型将进一步演化成应用模型中物理层面的内容，同时要在不同的层面和数据模型保持一致。

如果将分析和设计的逻辑模型看作是两个部分，分析模型偏重于整体的外向型行为实现，其要素是功能性需求的反映，对非功能性要求只给出必要的假设，因而分析模型通常比较简单；设计模型则包含更多的内向型结构细节，其要素必须顾及非功能性的要求，因而设计模型通常比较复杂。根据一般的经验，分析模型和设计模型中的要素比例至少为一比 1:5。

需要强调的是，两种模型的获取过程不是一个简单的自顶向下的单向过程，换言之，并不是在分析模型全部完成之后才着手建立设计模型。迭代的方法使得两种模型交错成长，这样能确保方案有效地反映问题的要求。笼统地讲，分析和设计的逻辑模型都可以称作对象模型，但是分析模型中更强调对象针对软件需求的行为和结构特征，而设计模型在分析模型的基础上添加了大量针对支撑环境的行为和结构特征，两种对象模型在形式上并没有清晰的界限。设计模型由分析模型逐步演化而成，如果不强调分析模型的独立存在价值和复用机会，分析模型通常只作为一种短期存在的过渡。

第 2 章　面向对象的内涵、优势和原则

2.1　方法、技术和工具的综合

作为认知方法，面向对象提供从一般到特殊的演绎手段和从特殊到一般的归纳形式；作为程序设计方法，面向对象使问题空间和求解空间在结构上尽可能地一致。

面向对象技术是一个综合概念，既包括指导开发软件的一系列原则，也包括支撑这些原则的程序设计语言、数据库及其他工具[1]。

面向对象技术还在发展和不断地完善。作为实践者，有必要不断地挑战我们学到的东西。

2.2　改善沟通、复用与应变能力

面向对象技术在各个层面对软件的开发产生着积极的影响，其优势的外在表现主要有三个方面。

第一，减少沟通障碍。一般情况下，软件开发总要涉及到各类工种的参与者，常见的诸如用户、分析师、系统构架师、设计师和编码人员等等。如果基于传统的方法，在问题求解过程中，不同工种的人员组织其工作内容要素的原则有所不同，通常不具有整体的一致性和连贯性，从而带来严重的沟通障碍。问题到方案映射的明确程度往往过分依赖于参与者的技能和团队的组织水平。基于面向对象技术，求解空间中的要素直接而紧密地反映问题空间中的要素，参与问题求解过程的多工种人员可以基于这条核心线索理解其他人员的工作，从而打破大量沟通壁垒。

第二，提高开发生产率。面向对象技术用于提高开发生产率的途径是“复用”。事实上，为了提升效率，无论使用何种方法，都有必要复用已经存在的工作成果，包括重复利用代码乃至重复利用方案的构架。但是，真正意义上的“复用”不应该等同于简单的“复制”；因为这种“基于复制的复用”将带来高昂维护代价。面向对象技术不仅在理论层面给出“复用”的方法，而且在程序设计语言和工具层面为“复用”提供了有力的支撑。

---

[1] Object Technology - A Manager’s Guide，Taylor，1997

第三，增强对变化的适应能力。缘于各种起因的软件需求变化是客观的现实，通常也是软件优化的动力。然而，如果问题中很小的变化在解决方案中“一石激起千层浪”，变化的积极因素就很可能导致得不偿失的影响，然而这正是基于传统方法经常难于避免的窘境。在这一点上，应用面向对象的方法具有明显的优势。鉴于求解空间和问题空间的相似性，来自于问题空间的一个局部变化在求解空间中的影响也将是局部的，而不会秧及整体结构。

面向对象技术，尤其是面向对象的分析和设计技术已经在众多复杂系统的开发中得到成功的应用，是公认的发展方向。尽管面向对象技术并非包治百病的灵丹妙药，但该技术的确带来了并正在带来实实在在的好处。

面向对象技术是实现软件生产工程化的关键技术之一。面向对象技术未必使得分析和设计变得比原来的方法更简单，但这种方法能使复杂的内容更有秩序，从而帮助开发团队铸造出更好的软件产品。因为，对于比较复杂的软件而言，需求到代码的明确映射关系和渐进演变过程将决定软件可维护性的高低，对于那些针对关键业务的应用软件，这一点至关重要。

特别需要指出，我们在生活中接触最多的“面向对象编程技术”仅仅是面向对象技术中的一个组成部分。如果仅仅使用面向对象的编程技术，通常只能开发出一个似是而非的面向对象软件系统，通常不能体现面向对象技术的本质优势，甚至可能无法从面向对象技术中得到任何好处。

发挥面向对象技术的优势是一个综合的技术问题，不仅需要面向对象的分析、设计和编程技术，而且需要借助必要的建模和开发工具。此外，还要借鉴已有的成功经验和科学的开发流程。

2.3 抽象、封装与层次

为了真正实现面向对象的优势，在应用面向对象技术，尤其是分析和设计的方法过程中，需要遵循一系列基本的原则。最根本的有三个方面：抽象、封装和层次。以下针对每一方面的原则，讨论其含义、目的、价值以及在分析和设计中的主要表现形式。

抽象

抽象（Abstraction）的结果反映出事物重要的、本质和显著的特征，言外之意是忽略那些次要的、非本质和分散注意力的特征。抽象的过程强调被抽象事物的重要共性，而忽略不重要的差异[1]。在面向对象方法中，抽象活动主要抽取事物的结构特征和行为特征，两方面特征是有机的整体。抽象的结果有赖于特定的领域，即被抽象者所处的上下文环境，具有客观针对性；抽象的结果有赖于特定的视角，即作出抽象动作的主体，具有主观针对性。换言之，即便是讨论同一个被

[1] Dictionary of Object Technology，Firesmith，Eykholt，1995

抽象的事物，针对不同的抽象者和上下文环境，抽象的结果也有可能不同。

举一个生活中的例子，你去餐馆吃饭，对你而言，一道菜可以抽象为色、香、味和售价；对于厨师而言，抽象的结果除了色、香、味之外可能还要包括配料、工艺流程；对于餐馆的老板而言，抽象的结果可能是成本和售价；对于跑堂的服务生而言，抽象的结果可能就是点这道菜的客人、桌位……，依此类推。

抽象是人类在认识复杂现象的本质过程中最强有力的思维工具。抽象对认知求解过程的最大贡献在于帮助我们获取问题和方案相通的本质，更好地管理复杂的系统：在一个特定的上下文环境之中，将注意力集中于事物最本质的部分，或者说是一个事物能和其他事物区别开的部分。

抽象活动的水平和结果的质量直接影响问题求解过程参与者之间的沟通效率。根据面向对象的基本思想，对问题领域中关键事物的抽象会相对稳定地延续到求解领域。如果在问题领域的关键抽象中有很多无关的细枝末节，意味着在解决方案中将会出现大量没用的代码，它们有可能分散问题求解过程中所有参与者的注意力并造成沟通的负担甚至是障碍。比如，你要开发一个顾客使用的定菜系统，就没有必要将“工艺流程”作为“菜”的特征加以抽象。

在概念上，最核心的抽象内容是对象（Object）。准确地讲，对象是一个具有明确边界和唯一标识的，封装了行为和状态的实体。这里的实体是广义的概念，可以代表一个具有物理意义的实体，可以是一个纯粹软件意义上的实体，甚至可以是一个概念上的实体。例如一个化学反映过程。类（Class）是对具有相同属性、操作、关系和其他语义特征的一组对象的静态描述。笼统地讲，在面向对象的分析和设计中，抽象的结果表现为各种类型要素的可视化描述。

封装

封装（Encapsulation）将对象特征的实现方式（包括相关的设计决定）隐藏在一个公共接口之后的黑盒之中[1]。封装概念的关键点在于被封装对象的消息接口，所有与该对象进行的沟通都要通过响应消息的操作来完成。除了对象本身，其他任何对象都没有可能改变它的属性。封装在很多时候也被称作“信息隐藏”，信息有两个层面的含义，一方面是接口中操作的具体实施方法，另一方面是对象内部的状态信息。对于和某对象沟通的其他对象而言，只需了解它的消息接口，即可顺利地与该对象进行沟通。

汽车的刹车脚踏板是一个很好的封装类比。如果将汽车的刹车系统看作一个对象，对于使用刹车系统的驾驶员而言，只要知道刹车脚踏板踩下去之后汽车会减速就等于会使用刹车系统，没有必要了解刹车系统如何工作以及刹车系统内部的状态。驾驶员使用任何一辆汽车的刹车系统都是一样的，即便它们的内部结构可能大不相同；反过来即便汽车的刹车系统内部采用了新技术也无须去调整驾驶员使用刹车系统的方法。换言之，汽车生产厂商制造刹车系统的工作和驾驶学校

[1] Dictionary of Object Technology，Firesmith，Eykholt，1995

训练驾驶员使用刹车系统的内容是互不影响的，原因是双方都有一个共识：“刹车脚踏板踩下去之后汽车会减速”，这即是逻辑上的接口概念。

封装起到两个方向的保护作用：一方面，对象内部的状态被保护起来，不会被与该对象沟通的对象直接篡改；另一方面，对象内部特征的变化不会改变其他对象与该对象的沟通方式。封装为面向对象系统带来一种叫作“多态”的能力（Polymorphism），即呈现在一个接口后面的多种实施形态。对接口的使用者没有任何影响。

封装使得系统具有更明显的高内聚、低耦合特征，进而，整个系统的体系构架将变得更加具有延展性（Resilience）。封装对认知求解过程的最大贡献在于帮助我们控制变化的影响范围，从而更好地管理复杂的系统：对于一个局部进行良好封装的系统而言，需求的变化在拟建系统中的影响将只可能在一个较小的范围内传播，而不至于产生严重的连锁反应。

概念上，接口是封装原则的准确描述手段。接口用于声明类或者构件能够提供的服务[1]。所谓服务就是响应消息的能力。该定义中的类是广义的，既可以是一个简单的类，也可以是一个子系统。所谓的子系统是一组要素的集合，其中一部分要素提供由另一部分要素声明要做的事情。笼统地讲，在面向对象的分析和设计中，封装往往被表现为多种要素的可见度（Visibility）。

层次

层次（Hierarchy）的基本含义是不同级别的抽象组成一个树形的结构[2]。层次的种类是多种多样的：可以是集合的层次、类属的层次、包含的层次、继承的层次、分区的层次、专业化的层次等等。

简单讲，层次就是一个描述分类的结构。层次的典型例子是生物中的门、纲、属、种、科等。注意，层次化的思维活动建立在抽象思维活动基础之上。在某种意义上，不同的层次反映了抽象主体关注被抽象客体的概括程度。举一个通俗的例子，我们在上班之前对天气的了解通常限于下雨或者不下雨，我们只关注要不要带一把伞；对于水利部门的人士而言，为了预防洪涝灾害，则需要了解究竟是大雨、中雨，还是小雨或者没雨。

层次的本质目的是表述并使用事物之间的相似性，同时，事物之间的区别得以更加明显。这带来了两个方面的益处：一方面，对同层次事物之间所具有的相同特征，没有必要在这个层次内作分别的（重复）描述，可以将这部分内容放到更高的一个层次中去描述；另一方面，主体理解客体的概括程度是可选择的，主体可以根据实际的需要决定采用较高层次的描述或者是较低层次的描述来认识客体。

不同层次之间的描述不存在重复和冗余，而是自上而下的重复使用。在得到必要的技术支撑之后，层次概念带来的直接价值是提升事物特征描述的可复用能力。

---

[1] UML User’s Guide (Booch，1999)

[2] Dictionary of Object Technology，Firesmith，Eykholt，1995

层次概念主要的表现形式为类之间的泛化关系（Generalization）。但是，层次概念不仅仅局限于类之间的泛化关系，对模式（Pattern）乃至框架（Framework）的复用在本质上同样是层次概念的应用，只不过研究的对象变成了系统的一个局部或者是整个系统。在面向对象的分析和设计中，模式通常描述为一组类的参数化协作关系，框架则接近于一个完整软件的模板，偏重于对整体结构和支撑构架的描述。

第 3 章　模型内容的组织和 UML 表述

通常，我们不会在掌握所有的词汇和语法之后才开始使用一种语言。掌握语言的关键在于有目的地使用，学习 UML 的情况很类似。在开始阶段，基于一个明确目标，集中精力理解一些必要的词汇和语法，在使用中深入体会是事半功倍的做法。本章无意累牍 UML 的基础语义和一般规则[1]，而是以用为本，针对本书实践过程中应用的 UML 内容作必要的介绍。

3.1　模型的基本组织结构

3.1.1　基本内容

概念上，UML 用于描述模型的基本词汇有三种：要素（Things），关系（Relationships）和图（Diagrams）。或者说，模型是一系列要素、关系和图的排列组合。其中，要素是模型中的核心内容，可以形象地理解为“点”；关系在逻辑上将要素联系在一起，可以形象地理解为“线”；图将一组要素和关系展现出来，可以形象地理解为“面”。这些“点”、“线”、“面”组成了“立体”的模型。

第一，要素。UML 有四种类型的要素，本书的实践过程全部涉及。

- 表述结构的要素，包括“Use Case[2]”、“类”（Class）、“接口”（Interface）和“协作”（Collaboration）。
- 表述行为的要素，包括“交互”（Interaction）和“状态机”（State Machine）。
- 用于组织的要素，即“包”（Package）。
- 用作辅助说明的要素，即“注释”（Notes）。

第二，关系。UML 中有 4 种类型的关系，本书的实践过程全部涉及。

- 关联关系（Association），表达两个类的实例之间存在连接。聚合关系和（Aggregation）与组合关系（Composition）是关联关系的强化形式。
- 依赖关系（Dependency），依赖者“使用”被依赖者的关系。
- 泛化关系（Generalization），表达“特殊的”是“一般的”一种。
- 实现关系（Realization），“被实现者”是对要求的说明，“实现者”是针对要求的解决方案。

---

[1] 《UML 用户指南》中深入浅出地介绍了如何用 UML 解决一般性的建模问题。

[2] 为避免读者混淆，本书中不对 Use Case 作中文翻译，读者可以将“Use Case”作为一个符号。在其他文献中比较常见的中文翻译有“用例”、“用况”、“案例”或“使用案例”等。

第三，图。UML 中有九种图，本书的实践过程中涉及最常用的两种静态图与全部四种动态图。

- Use Case 图（Use Case Diagram）。它是一种静态图，主要用于展示 Use Case、Actor[1]及其关系。
- 类图（Class Diagram）。它是一种静态图，主要用于展示类、接口、包及其关系。
- 序列图（Sequence Diagram）。它是一种动态图，用于按时序展示对象间的消息传递。
- 协作图（Collaboration Diagram）。它是一种动态图，其核心内容与序列图相对应，强调（收发消息的）对象间的结构组织。序列图和协作图统称为交互图（Interaction Diagram）。
- 状态图（Statechart Diagram）。它是一种动态图，主要用于展示对象在其生命周期中可能经历的状态以及在这些状态上对事件的响应能力。
- 活动图（Activity Diagram）。它是一种动态图，用于展示系统从一个活动流转到另一个活动的可能路径与判断条件。

其他三种静态图分别为对象图（Object Diagram）、构件图[2]（Component Diagram）和部署图（Deployment Diagram）。

3.1.2 语义扩展

作为一种语言，UML 除了提供基本的词汇，还给出了对自身描述能力的三种扩展机制，即构造型（Stereotype）、标注值（Tagged value）和约束（Constraint）。本书实践过程将使用“构造型”，扩展基本模型词汇的语义，从而表达新的概念。主要包括以下几种。

- 类的构造型。在“提取分析类”活动中将使用实体类《entity》、控制类《control》和边界类《boundary》；在“确定核心元素”活动中将使用“子系统代理”《subsystem proxy》；在“引入外围元素”活动中将使用角色《role》。实质上，接口也是类的一种构造型《Interface》。
- 包的构造型。在“选用构架模式”活动将使用层次《layer》；在“确定核心元素”活动中将使用子系统《subsystem》。
- Use Case 的构造型。“Use Case 实现”《use case realization》，表述用分析或设计元素实现局部需求的协作内容；“设计机制”《mechanism》，表述解决特定技术问题的协作模式。

上述构造型是 UML 应用建模中常见的语义扩展形式，本书的实践过程将结合特定应用场合具体介绍相关的概念与用法。

---

[1] 为避免读者混淆，本书中不对 Actor 作中文翻译，读者可以将“Actor”作为一个符号。在其他文献中比较常见的中文翻译有“主角”、“操作者”等。

[2] 本书第三部分介绍构件图，作为实践过程的扩展内容。

3.1.3 组织方式

模型的内容通过包以及包的层层嵌套组织在一起，模型中的包类似于文件系统中的目录。包将一堆零散的模型内容简单地组织在一起，目的是更易于理解和管理。

模型应该能够反映建模者和使用者的特定视角，即所谓的“构架视图”（Architecture View）。打一个比方，一座大楼在土木设计师的眼里可能是一堆钢筋混凝土和表面材质，在管道设计师眼里可能是一堆管子和接头，在网络工程师眼里可能是一堆网络设备和连线…，不同主体对同一客体的认识结果有赖于各自的视角。这样能更好地集中注意力，从而有效地解决关键问题。在模型中，构架视图用包的形式表达。每一种特定的视角都可以对应一种类型的构架视图，RUP 中给出了几种比较典型的构架视图[1]。本书实践过程的基本依据是需求模型，属于 Use Case 视图（Use Case View）；实践过程的工作结果是设计模型，属于逻辑视图（Logical View），参见图 3-1[2]。

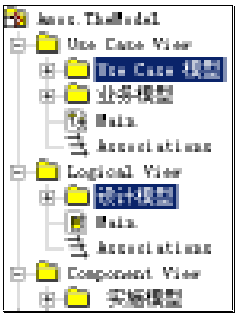


图 3-1　模型的组织方式

3.2　常见图的用法与内容

图是为了实现建模目的而使用的一种表现手段，一种图可用于不同场合以满足特定的要求。图不仅表述建模的最终结果，同样记录认知求解的轨迹。基于“以用为本”的原则，针对本书实践过程，概念性地说明几种图，着重强调两方面内容。

- 用法与相对位置。
- 包含的关键内容。

---

[1] Use Case 视图、逻辑视图、进程视图、实施视图和部署视图。
[2] 本书采用著名建模工具 Rational Rose 的 Browser 示意模型内容的结构。

3.2.1 Use Case 图：描述拟建系统与外部环境的关系

用法与相对位置

Use Case 图主要用于描述拟建系统[1]和外部环境的关系，参见图 3-2。概念上，Use Case 的集合表达拟建系统，Actor 的集合表达外部环境，Use Case 和 Actor 之间连线的集合则表达拟建系统和外部环境的边界。

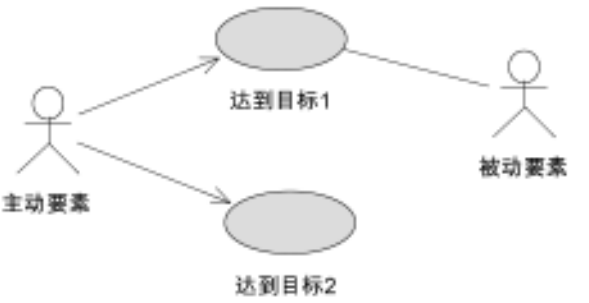


图 3-2　描述拟建系统与外部环境关系的 Use Case 图

这种用法的 Use Case 图通常位于“Use Case 模型”的“Use Cases”包内，参见图 3-3。通常将 Actor 和 Use Case 放在不同的包中。

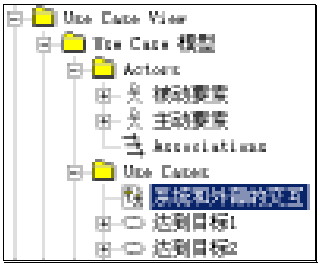


图 3-3　描述拟建系统与外部环境关系的 Use Case 图的位置

关键内容

Actor

[1] “拟建系统”即为拟议开发的软件系统。

Actor 在图中表现为火柴棍儿小人儿。简单讲，Actor 代表拟建系统外部和系统进行交互的某类人或系统。

Use Case

Use Case 在图中表现为一个椭圆。Use Case 定义了一组相关的由系统执行的动作序列，将有价值的可见结果提供给某个 Actor。

Use Case 与外部的交互活动中可能涉及若干个 Actor，但是只有一个 Actor 主动要求得到有价值的可见结果。通常称之为主导（Primary）Actor。主导 Actor 是触发交互活动的 Actor，它到相应的 Use Case 之间的连线标有箭头。

通信关联（Communication Association）

Actor 和 Use Case 之间的连线称为通信关联，表示 Actor 和相应 Use Case 之间的交互。无论有没有箭头，通信关联都表示介于 Actor 和相应 Use Case 之间的双向会话，本书参照主流的方法[□]，用箭头表示 Actor 触发 Use Case 的执行。

3.2.2 Use Case 图：描述需求模型与设计模型的关系

用法与相对位置

可以用 Use Case 图描述功能需求的局部（即 Use Case）与相应分析和设计内容（即“Use Case 实现”）之间的可追溯关联，参见图 3-5。“Use Case 实现”的概念参见第 4 章“设计模型的内容和演进”部分。严格地讲，这种图并不是真正意义上的 Use Case 图，而是有 Use Case 出现的图。这种图通常位于设计模型中一个名为“Use Case 实现”的包内，参见图 3-4。

图 3-4　描述需求模型与设计模型关系的 Use Case 图的位置

[□] 在 UML 规范中，Communication-Association 的箭头是可选的，选用箭头通常是为了更有效地说明图示的内容。

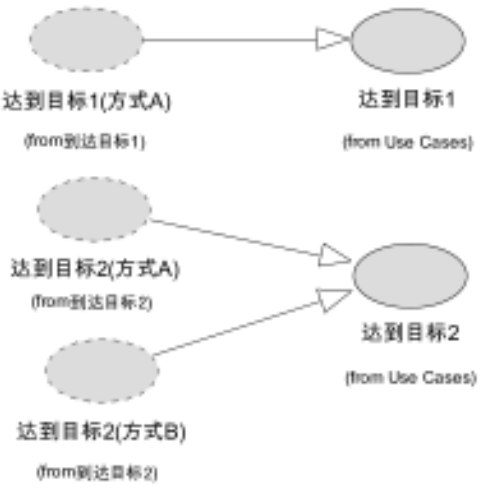


图 3-5　**描述需求模型与设计模型关系的** Use Case **图**

关键内容

Use Case

这里的 Use Case **和前一种用法** Use Case **图中的含义一致。**

“Use Case 实现”(Use Case Realization)

“ Use Case **实现”在图中表现为虚线边框的椭圆，用于表达基于设计视角的** Use Case **内容，即相关的分析或设计元素的协作关系。“** Use Case **实现”包括动态描述内容（交互图组）和静态描述内容（类图）。**

实现关系

实现关系是“ Use Case **实现”到** Use Case **之间的连线。设计工作完成时，** Use Case **模型中的每个** Use Case **在设计模型中至少有一个“** Use Case **实现”与之对应，“** Use Case **实现”和** Use Case **之间有可能是多对一的关系。通俗地讲，一种要求可以通过多种办法解决。**

通常，在设计模型的“ Use Case **实现”包中，对应每个** Use Case **建立一个以该** Use Case **命名的包，在此放置对应该** Use Case **的“** Use Case **实现”(组)。参见图** 3-4。

3.2.3 类图：描述类、接口和子系统之间的关系

用法与相对位置

类图是应用最广泛的一种图，描述拟建系统各个层面的静态结构，主要用于表述类、接口和子系统之间的关系，参见图 3-6。

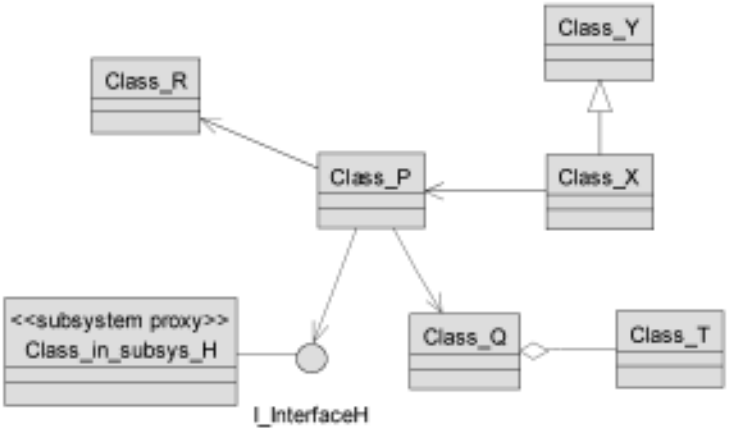


图 3-6　描述类、接口和子系统之间关系的类图

这种用法的类图可以进一步划分为三种不同的情形，尽管表现形式相似，但是它们通常位于模型的不同位置。

其一，表述参与某一特定协作的类、接口和子系统之间的关系。这种情形的类图被称为“参与类图”（VOPC，View of Participating Classes）。“Use Case 实现”与“构架机制”是两种典型的协作，“参与类图”隶属于这两种类型的协作内容，参见图 3-7 与图 3-8。“构架机制”概念参见第 4 章“设计模型的内容和演进部分”。

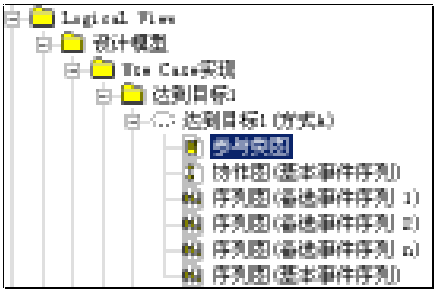


图 3-7　“Use Case 实现”中的“参与类图”

图 3-8　“构架机制”中的“参与类图”

其二，表述同一包中的类、接口和子系统之间的关系，这种类图通常出现在相应的包中，参见图 3-9**。**

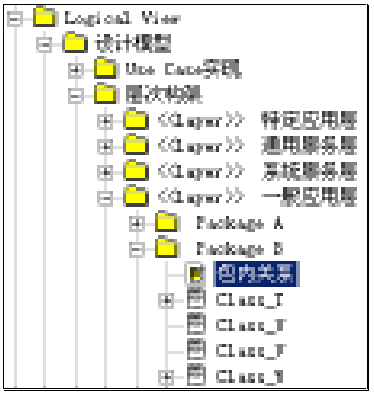


图 3-9　**表述包内部关系的类图**

其三，针对上述两种情形以外的其他目的，表述类、接口和子系统之间的关系，这种情形的类图可以出现在任何需要的位置。

关键内容

类（Class）

类用于描述一组具有相同属性、操作、关系和语义的对象。类的 UML **表述参见图** 3-10**：上面是类的名称（通常首字母大写），中间是类的属性（**Attribute**，通常首字母小写），下面是类的操作（**Operation**，通常首字母小写）。**

属性是类的一项冠名特征（Named Property），对象在属性上取值；操作也是类的一项冠名特征，描述对象响应某种要求的能力。

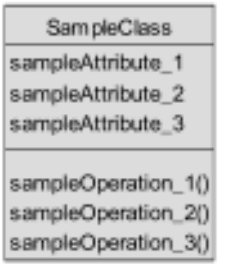


图 3-10　类的表述

接口（Interface）

接口用来说明一个类或子系统应该提供的服务，形式上是一组操作的集合。接口以一种规范的形式表述多态的概念[1]。

图 3-11 和图 3-12 是接口的两种 UML 表述：第一种是简略的表述（Elide），一个圆圈和接口的名称，不显式地指出接口中定义的操作；第二种是详细的表述（Canonical），显式说明接口包含的操作集合。

图 3-11　接口的简略描述

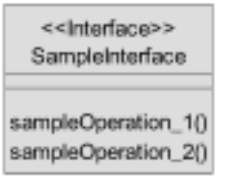


图 3-12　接口的详细描述

子系统（Subsystem）

子系统是一组元素的集合，其中一部分元素说明这组元素能够提供哪些行为，而另外一部分元素则具体提供相应的行为[2]。子系统所具有的行为特征在概念上与类相似。用包的构造型《subsystem》表述子系统，参见图 3-13。

---

[1] 多态是面向对象软件表现出的一种能力，即在统一的接口背后隐藏多种不同的实现。基于特定程序设计语言实现的多态在逻辑层面有局限性。注意，具有 UML 基本语义的接口与某些程序设计语言中的接口含义略有差别。

[2] UML User's Guide (Booch，1999)

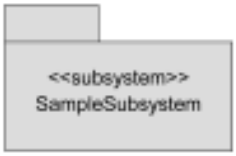


图 3-13　子系统的表述方式

关联关系（Association）

关联关系是一种普遍存在而内容丰富的结构化关系。关联关系的基本含义是两个类的实例之间存在稳定的“连接”（link），可以用于传递消息。当一个类的对象作为另一个类的对象的变量成员时，两个类之间有关联关系。图 3-14 给出一个通俗的示例。

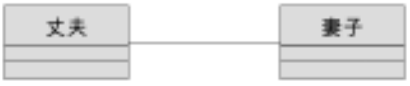


图 3-14　关联关系的基本含义示例

可以用 Java 代码片段表述丈夫和妻子的关联关系。

```
public class 丈夫 {
   private 妻子 the妻子;
   public 丈夫() {}
}

public class 妻子 {
   private 丈夫 the丈夫;
   public 妻子() {}
}
```

关联关系中的多重性（Multiplicity）是指类 A 的一个实例对应类 B 的实例个数。图 3-15 给出一个通俗的示例：一只麻雀两条腿…，一只螃蟹八条腿…。

当多重性没有把握的时候，姑且不作标注。当然，如果多重性非常明显，也省略，参见图 3-15（一个萝卜一个坑）。只有结构化的关系才存在多重性，在分析和设计过程中，多重性主要用来表述业务规则，常见的标识方式有：1、0..1、*、1..*、0..* 等。

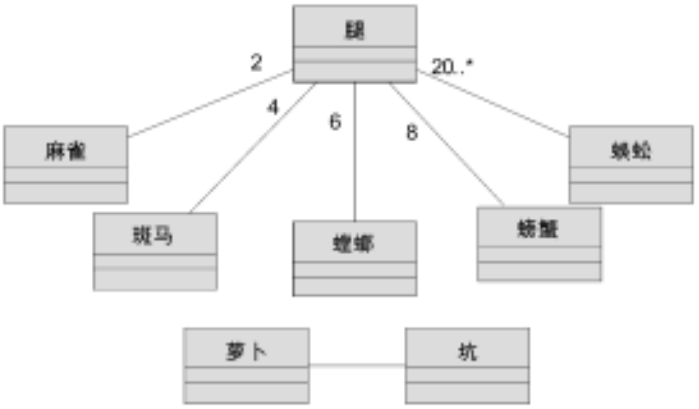


图 3-15　关联关系中的多重性示例

可以用 Java 代码片段[□]表述麻雀和腿之间的多重性概念。

```
public class 麻雀 {
   private 腿 the腿[2];
   public 麻雀() {}
}
```

关联关系的访问方向（Navigability）表示某一方的实例能够“访问”另一方的实例。至少存在一个可用的访问方向，如果仅存在一个可用的访问方向，那么关联关系是单向的，用箭头表述这一概念。两端都没有箭头的连线表述关联关系是双向的。

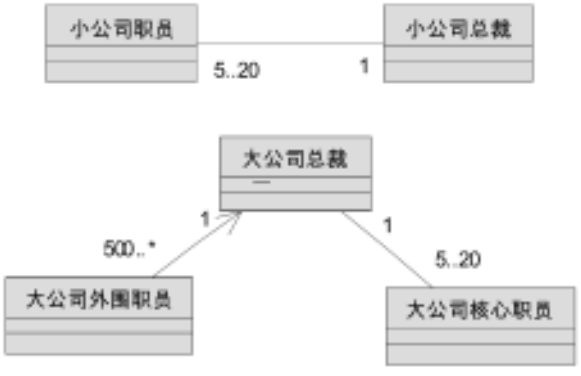


图 3-16　关联关系的访问方向示例

□ UML 表达的语义内容在特定程序设计语言中的反映并不是惟一的。

作一个通俗的类比，参见图 3-16：在一个小公司里，公司的总裁认识所有的职员，当然每个职员都认识公司的总裁；在大公司里则有所不同，所有的职员都认识公司的总裁，但是公司的总裁通常只认识那些核心职员，而并不认识那些外围职员，因而大公司总裁和大公司外围职员之间的关联关系是单方向的。

可以用 Java 代码片段表述相关概念。

```
public class 大公司总裁 {
   private 大公司核心职员 the大公司核心职员[20];
   public 大公司总裁() {}
}

public class 大公司外围职员 {
   private 大公司总裁 the大公司总裁;
   public 大公司外围职员() {}
}

public class 大公司核心职员 {
   private 大公司总裁 the大公司总裁;
   public 大公司核心职员() {}
}
```

关联关系中的角色（Role）表述类 A 的实例对类 B 的实例的具体含义。可以结合图 3-17 中的示例加以理解：一个领域专家对一个行业协会而言是一个成员，一个行业协会对一个领域专家而言是一个信息来源；以此类推。

图 3-17　关联关系中的角色示例

可以用 Java 代码片段表述领域专家和行业协会之间的角色概念。

```
public class 领域专家 {
   private 行业协会 信息来源;
   public 领域专家() {}
```

```
}

public class 行业协会 {
   private 领域专家 成员;
   public 行业协会() {}
}
```

聚合关系（Aggregation）是关联关系的一种强化形式，表示两个类的实例之间有“整体”与“部分”的关系：处于空心菱形符号一端的类是“整体”，另外一端的类是“部分”。读者可以参考图 3-18 加以理解，连队和士兵是整体和部分的关系。如果“整体”的多重性大于 1，表示“部分”的实例可以被多个“整体”的实例“共享”。参见图 3-18，球队和球员就可能是这种关系：一个球员既可以是俱乐部球队的球员，同时也可以是国家队的球员。聚合关系并不隐含“整体”的实例消失将导致“部分”的实例消失。例如，球队解散了，球员还在。

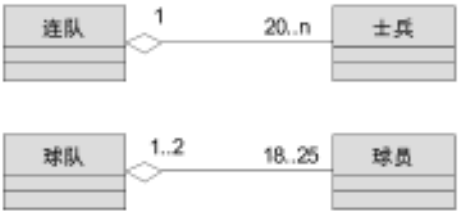


图 3-18　关联关系中的聚合关系示例

可以用 Java 代码片段表述连队和士兵之间的聚合关系概念。

```
public class 士兵 {
   private 连队 the连队;
   public 士兵() {}
}

public class 连队 {
   private Vector the士兵;
   public 连队() {}
}
```

组合关系（Composition）是进一步强化的聚合关系，在聚合关系含义的基础上，增加了“整体”与“部分”之间“皮之不存，毛将焉附”的语义。“整体”一端用实心的菱形表示。在这种关系中，“整体”的实例之间不可能“共享”“部分”

的实例。读者可以参考图 3-19 加以理解。

图 3-19　关联关系中的组合关系示例

可以用 Java 代码片段表述组合关系的概念。

```
public class 跳动的心脏 {
   private 活人 the活人;
   public 跳动的心脏() {}
}

public class 活人 {
   private 跳动的心脏 the跳动的心脏 = new 跳动的心脏();
   public 活人() {}
}
```

依赖关系（Dependency）

依赖关系表达“使用”的语义，用带有箭头的虚线表示，参见图 3-20。相对于关联关系，依赖关系是一种比较弱的关系，“被依赖者”类的变化有可能影响“依赖者”。

图 3-20　依赖关系示例

泛化关系（Generalization）

类 A（相对特殊）到类 B（相对一般）的泛化关系表示“类 A 是类 B 的一种”。通常称类 A 为子类，称类 B 为父类，子类的实例同样是父类的实例。读者可以参照图 3-21 加以理解：一名足球运动员同时也是一位大球运动员、球类运动员和运动员，越来越笼统。

在一般意义上，泛化关系和继承（Inheritance）是可以互换的概念。如果加以

区别，那么泛化关系是一种关系的名称，而继承则表达一种反映和实现这种关系的机制，主要指子类能够引用父类的结构和行为特征。UML 中使用泛化关系表述这一概念，有助于避免特定程序设计语言中继承的实现方式可能带来的干扰。

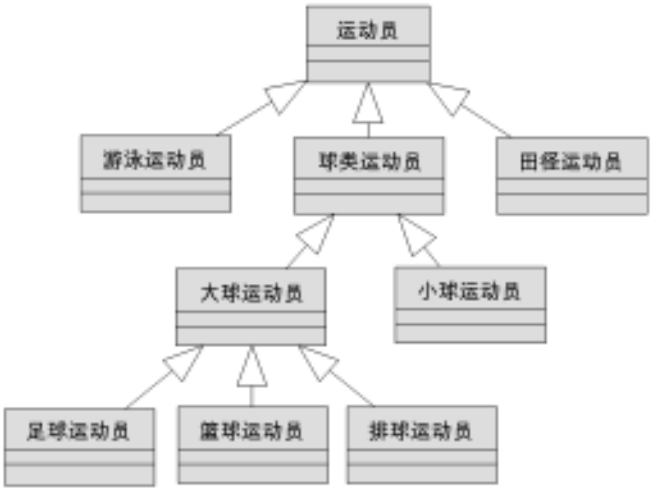


图 3-21　泛化关系的示例

可以用 Java 代码片段表述足球运动员和大球员动员之间的泛化关系。

```
public class 运动员 {
   public 运动员() {}
}

public class 足球运动员 extends 大球运动员 {
   public 足球运动员() {}
}
```

实现关系（Realization）

实现关系中的一方（甲方）作为要求被提出，另一方（乙方）具体履行要求中声明的任务。类图中出现的实现关系大多表述子系统或类实现接口，参见图 3-22。例如，雇佣“家庭保姆”或将孩子送到“幼儿园”都可以完成接口“照顾学龄前儿童”中规定的任务。

实现关系的表述方式为虚线加上一个空心的箭头，参见图 3-22。如果甲方是接口，对应于接口的两种表述形式，实现关系也有两种表现形式，简略的形式（Elide，参见图 3-23）和详细的形式（Canonical，参见图 3-22）。

图 3-22　**实现关系的（详细）表述**

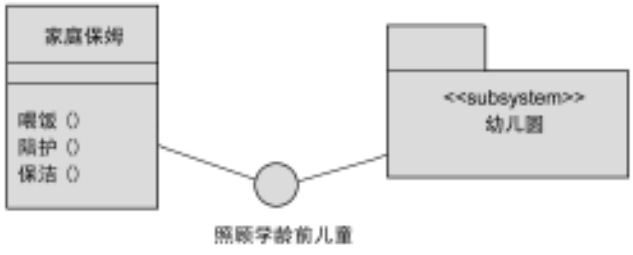


图 3-23　**实现关系的（简略）表述**

以日常生活为例，将关联、依赖、泛化和实现四种关系反映在同一张类图中，希望有助于读者理解它们之间的语义区别，参见图 3-24。

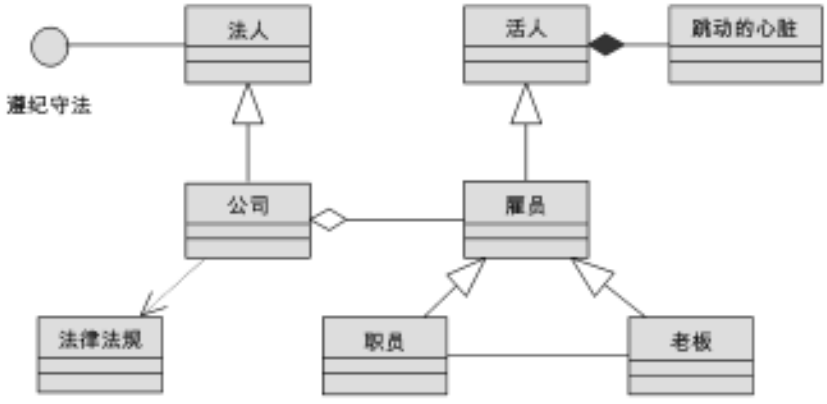


图 3-24　**通过日常生活中例子理解四种关系**

3.2.4 类图：描述包之间的依赖关系

用法与相对位置

类图经常用于表述包之间的依赖关系，从而反映模型的组织结构，参见图3-25。这种类图的位置比较灵活，通常应当在所涉及的包之外。

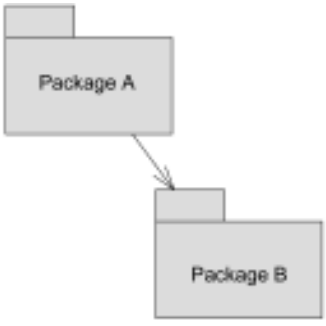


图 3-25　表述包之间关系的类图

关键内容为包和依赖关系。

包（Package）

包是用于分组放置模型要素的简单组织机制，包之间可以多层嵌套。

依赖关系（Dependency）

包之间的依赖关系取决于处于两个包内的模型要素，但视角更加概括。

3.2.5 序列图：描述局部分析和设计的场景

用法与相对位置

图 3-26　序列图

序列图用于描述动态行为，直观易懂，是最常用的一种交互图，参见图 3-26。描述局部分析和设计场景的序列图位于“ Use Case 实现 ”的协作中，参见图 3-27。通常，Use Case 中的每一个事件序列对应“ Use Case 实现 ”中的一张序列图。

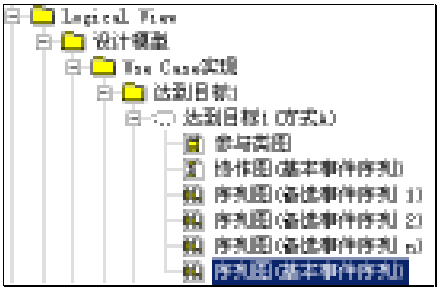


图 3-27　用于描述局部分析和设计场景的序列图

关键内容

对象（Objects）

概念上，对象是一个具有明确边界的、封装着状态与行为的实体。状态表现为属性的一组取值以及同其他对象的“ 连接 ”状况，行为是指对象自身的行动以及对外界激励的响应。

对象的符号有三种表述方式。其一，仅仅给出所属类的名字，不指明特定的对象。例如：Author 表示某个作者。其二，仅仅给出对象的名字，不指出所属类[□]。例如 amos：表示一个名称为 amos 的对象。其三，同时给出对象和类的名字。例如 amos：Author 表示一个叫作 amos 的作者。同一序列图中出现某一类的多个不同实例时，须指明每个对象的具体名称。

对象符号下方是一条垂直的虚线，称为对象生存线（Lifeline）。沿对象生存线上展开的细长矩形称为控制焦点（Focus of Control），表述来自其他对象的消息被回应的时间跨度。

消息（Messages）

消息是对象间通信的具体内容，消息表述为一条对象生存线到另一条对象生存线的带箭头水平实线，箭头指向接受消息的对象，参见图 3-26。如果消息被对象发至其自身，称为返身消息（Reflexive Message）。消息由序号、名称和参数组成。序号可以是连续编号或层次化编号，名称是必须的内容；参数按需而定。

□ 在序列图的绘制过程中，允许先用特定对象参与序列图的创建，然后再指定其所属类。这种方式适用于分析新增的需求而后重复利用现有设计内容。

3.2.6 序列图：描述“构架机制”的典型场景

序列图还用于描述“构架机制”的典型应用场景，模型中的相对位置参见图3-28。这种序列图中的关键内容与前一种用法没有区别。

图 3-28 用于描述“构架机制”典型场景的序列图

3.2.7 协作图：描述局部分析和设计的场景

协作图是另一种形式的交互图，参见图3-29。协作图的本质内容（“对象”和“消息”）与相关序列图存在明确的对应关系，例如，图3-29和图3-26等价。在协作图中，传递消息的对象之间存在一条连线，表示它们之间的“连接”(Link)，即消息传递通道。

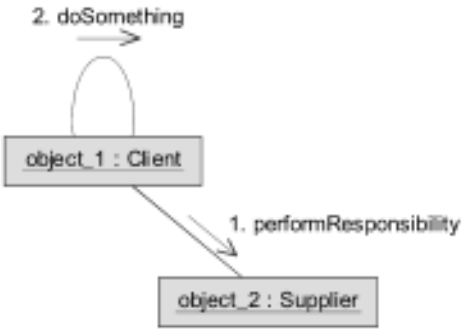


图 3-29 协作图

协作图主要用于描述局部分析和设计中的特定场景，其特点是强调对象间的结构关系。这种用法的协作图位于“ Use Case 实现 ”的协作中，参见图 3-30。通常，只关注那些对应重要事件序列的协作图。

图 3-30　用于描述局部分析和设计场景的协作图

3.2.8　状态图：描述具有明显状态特征的类

用法与相对位置

状态图用于展示对象（类的实例）生命周期中可能处于的状态、在这些状态中的行为、发生状态转换的事件及其相关的动作。一个类所处的全部可能状态及相关的行为构成了这个类的状态机（State Machine）。

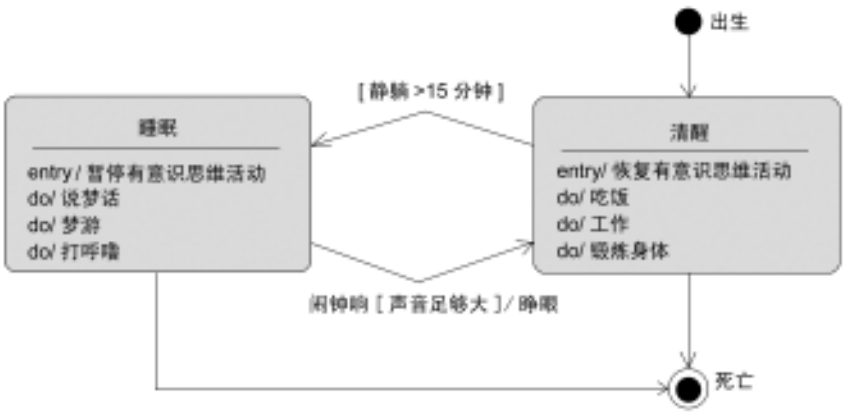


图 3-31　状态图示例

基于必要的简化和假设[1]，图 3-31 给出了通俗的状态机：人从出生到死亡的生命周期内，要么处于清醒状态，要么处于睡眠状态；静躺 15 分钟会从清醒转入睡眠；闹钟一响，人会睁眼醒来；进入睡眠状态必然暂停有意识的思维活动，睡眠中可能说梦话、梦游或者打呼噜；进入清醒状态必然恢复有意识的思维活动，在清醒时可能吃饭、工作或者锻炼身体。

描述类状态特征的状态图隶属于该类的状态机，参见图 3-32。一个状态机可以由多张状态图来描述。

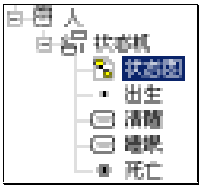


图 3-32　类的状态图在模型中的位置

关键内容

状态（State）

状态是一个对象在生存周期内处于的某一种情形（Condition or Situation），限定了该对象对事件的响应能力。状态在图中表述为没有棱角的矩形。有两种比较特殊的状态：初始状态（实心圆点）和结束状态（实心圆点外加一个圆圈）。一个状态机只能有一个初始状态，可能有多种结束状态。

对象在某个状态中有两种类型的行为：动作（Action）是与状态转变相关原子化（Atomic）的行为，不可能被打断；活动（Activity）是与某一状态相关的非原子化的行为，有可能被打断。

转移（Transition）

转移是指在某种激励作用下从一个状态转到另一个状态[2]。转移通常是满足特定条件（Guard Condition）时对某种事件（Event）的响应。

3.2.9　活动图：描述 Use Case 的事件流结构

用法与相对位置

[1] 不考虑“半梦半醒”的状态。
[2] “另一个状态”也可以是对象原来所处的状态。

活动图就是通常所说的流程图。图 3-33 给出一个活动图的示例，说明“开发软件需求”的流程。

本书中，活动图主要用于描述 Use Case 中的事件流结构。这种用法的活动图归属于某一特定 Use Case，在模型中的相对位置参见图 3-34。

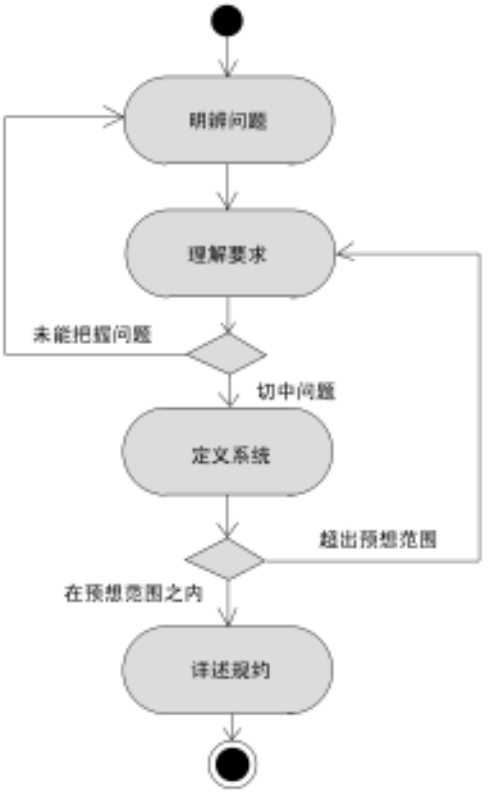


图 3-33　活动图

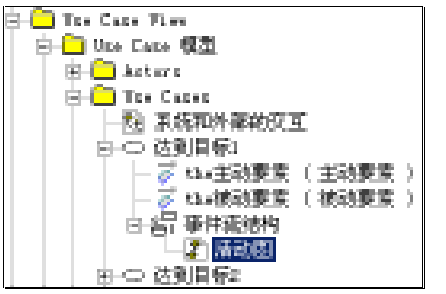


图 3-34　描述 Use Case 事件流结构的活动图在模型中的相对位置

关键内容

活动（Activity）

活动的内涵顾名思义，在图中表现为一个标有名称的鼓形图标。

判断（Decision）

根据特定条件判断活动的转移路径，在图中表现为菱形图标。

第二部分　UML 应用建模实践过程

一　步步为营

第 4 章　应用建模实践过程概述

4.1　任务和活动（Activities）

简单讲，面向对象应用建模（Application Modeling）的实践过程有 3 个目标：

- 有步骤、分层次地演进系统构架。
- 将软件需求逐渐转变为软件的设计方案。
- 保障软件的设计方案能够适应实施环境。

总体框架

如图 4-1 所示，应用建模实践过程由五项“任务”组成，依次为“全局分析”、“局部分析”、“全局设计”、“局部设计”和“细节设计”。前两项任务以分析为核心，后三项任务以设计为核心。五项任务中包括 14 个活动，活动进一步被细化为 30 个步骤。

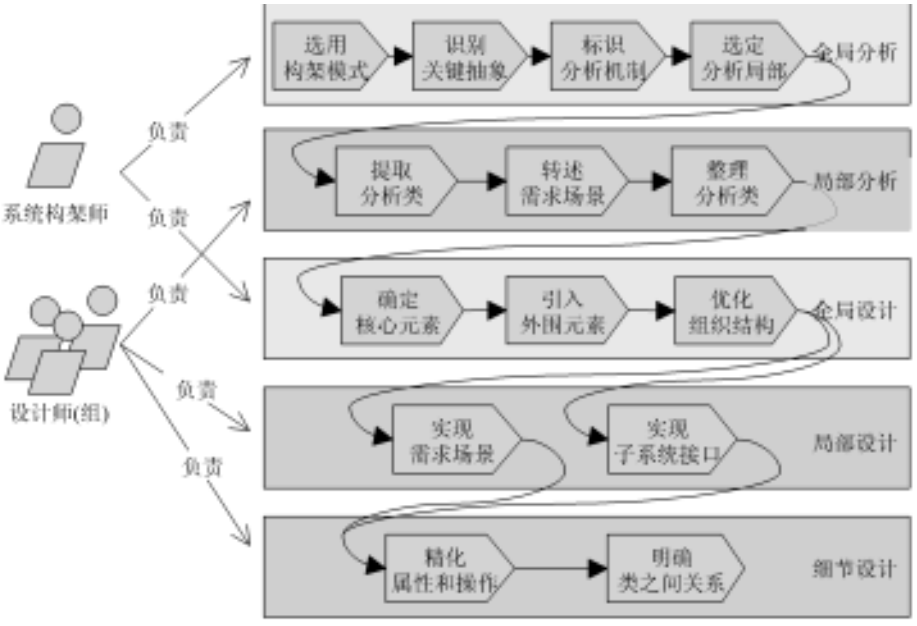


图 4-1　应用建模实践过程的框架

实践过程的每一项任务的内容构成一章，每一项活动作为一节，其中涵盖相关的核心概念、关键步骤、常用的实践技巧和贯穿全程的示例。

Rational 统一过程是千锤百炼的成功经验集合，其中关于分析和设计的内容非常宽泛，本书介绍的实践过程与 Rational 统一过程的“分析和设计”规程（Discipline）相容，可以看作为 Rational 统一过程在应用建模工作方面的一个实例。Rational 统一过程的分析和设计规程是具有普遍意义的流程框架，借鉴这一框架能够充分利用前人的最佳经验（The Best Practices）。不过，实践中没有可能，更没有必要“求全责备”或“惟命是从”。

迭代策略

本书展示的实践过程充分遵循 Rational 统一过程的核心思想原则：Use Case 驱动、体系构架为核心的迭代化开发。图 4-1 展现的框架可以被看作是一次迭代的过程。由于在“全局分析”任务中引入了“选定分析局部”活动，实践过程可以充分地支撑迭代化开发的策略，参见图 4-2。通常，“全局分析”任务中的前几项活动在后续迭代中可以被略去。

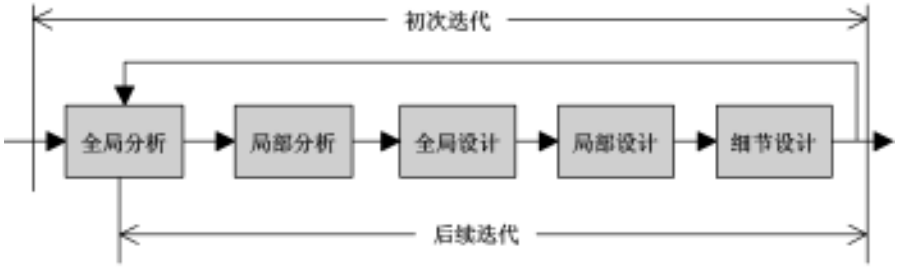


图 4-2　可迭代的的应用建模实践过程

迭代化方法中通常不作过多的假设，尽量降低对既往工作结果进行大面积否定的可能。在现实生活中，前期活动中过度的假设往往会导致后续工作不得不将错就错，表面上还能满足要求，但暗中牺牲了整体的质量和持续发展的能力。

4.2　角色和分工（Roles）

两类人员角色

参与应用建模的人员主要分成两类角色：系统构架师和设计师。

系统构架师负责领导和协调整个项目中的技术活动。在个人综合素养方面，

系统构架师应该具有领导才能，能够在压力下作出关键性的决策并善始善终；能够赢得项目经理、客户、用户群体以及管理团队的认同和尊敬，尤其要善于和项目经理紧密协作[1]；在各个方面都能展现出面向目标的实干作风。在专业技能方面，与其他角色相比，系统构架师通常具有全方位的技能，其见解重在广度，而不是深度。系统构架师不仅需要具备设计师的各项技能，而且应该具有问题领域和软件工程领域的实践经验，从而有能力在无法获得完整信息的情况下迅速领会问题并根据经验作出审慎的判断。如果项目较大，系统构架师将是一个团队，上述的关键素质要求可由团队成员来分担，但其中要有一名系统构架师具有足够的权威。

设计师的工作对象通常是系统的局部或者细节。在本书的实践过程中，设计师应该掌握的技能包括：理解以 Use Case 建模技术捕获和描述的软件需求；在系统构架师的统一协调下，应用 UML 进行局部的面向对象分析和设计；了解主流的实施技术（程序设计语言和开发环境）。

两种职责范围

从图 4-1 中不难看出，系统构架师负责全局性的分析和设计问题，设计师负责局部性的分析和设计问题以及细节性的设计问题。实践过程中并没有采用单一的自顶向下的策略（从全局到局部），而是在一个迭代中完成两次全局和局部的过渡，每一次过渡都为系统构架师和设计师之间提供了沟通的机会，在本质上，为提升设计的质量和完整性创造了有利的客观条件。

4.3 设计模型的内容和演进

设计模型的内容

笼统地说，广义的设计模型包括在分析和设计活动中得到的所有结果。针对本书将要讨论的实践过程，设计模型由 3 个部分组成，参见图 4-3。

- “Use Case 实现”。反映软件需求对设计内容的驱动。
- 层次构架（Layers）。一种典型的构架模式，也是本书实践过程选用的构架，它将分析和设计的结果按照特殊到一般的等级分组，层次构架中的内容是后续开发活动的直接依据。
- “构架机制”（Mechanism）。描述可复用的设计经验。

图 4-3 展示了设计模型的主要内容以及它们与软件需求工件（Artifacts）的基本关系。

逻辑上，层次构架依赖“Use Case 实现”和“构架机制”。层次构架中的内容

[1] 系统构架师主要负责技术问题，项目经理主要负责行政管理问题，两种角色的关系类似于电影导演和制片人之间的关系。

代表了分析和设计活动的实质结果，处于设计模型的核心地位。

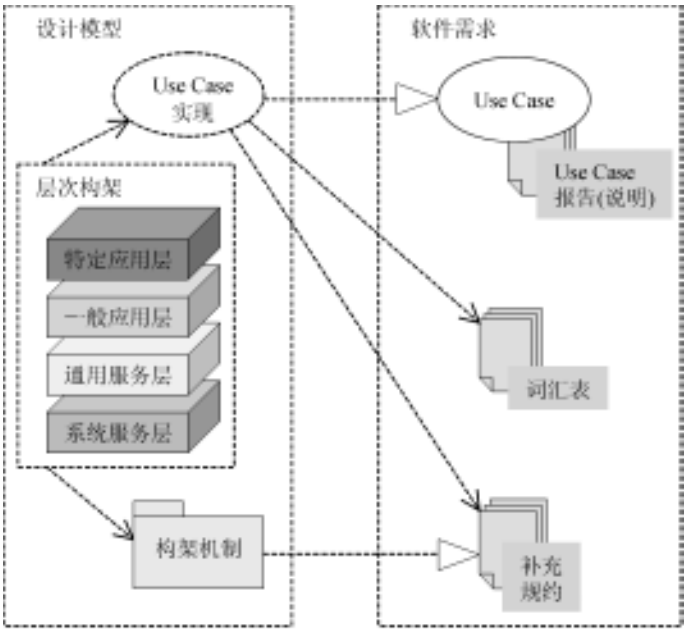


图 4-3　设计模型的组成以及和软件需求的关系

整个设计模型对软件需求的依赖关系主要体现在两个方面：一方面，“ Use Case 实现 ”通过一组协作实现“ Use Case ”描述的功能需求，“ Use Case 实现 ”是软件功能需求[1]内容过渡到层次构架中内容的桥梁；另一方面，“ 构架机制 ”主要实现作为非功能需求载体的“ 补充规约 ”，“ 构架机制 ”是软件非功能需求得以体现的主要形式。形象地讲，可以将“ Use Case 实现 ”比作牵引力，而“ 构架机制 ”比作支撑力，层次构架中的内容由软件需求牵引并被既往经验支撑。此外，“ Use Case 实现 ”作为核心纽带，还有赖于“ 词汇表 ”和“ 补充归约 ”中的相关内容。

本书采用的层次构架是软件构架模式中比较典型的一种。从图 4-3 中不难看出，层次构架内容间接依赖软件需求。因而，本书的实践过程在典型的基础上并没有失去普遍适用性。换言之，如果将层次构架变换成其他形式的构架模式，设计模型和软件需求之间的整体关系是类似的。如果使用（Model-View-Controller，MVC）构架模式，设计模型和软件需求的关系参见图 4-4。

以下，简要介绍设计模型中的主要组成部分。

[1] 功能需求主要是那些和应用逻辑直接相关的需求内容；非功能需求主要是那些和应用逻辑不直接相关的需求内容，通常由一些纯粹的计算机软件概念表达。

层次构架

构架（Architecture）是系统（在特定上下文环境中的）最高层概念。对构架的描述往往有赖于特定的视角。在本书中，构架主要是指拟建系统重要设计内容的逻辑组织及结构。

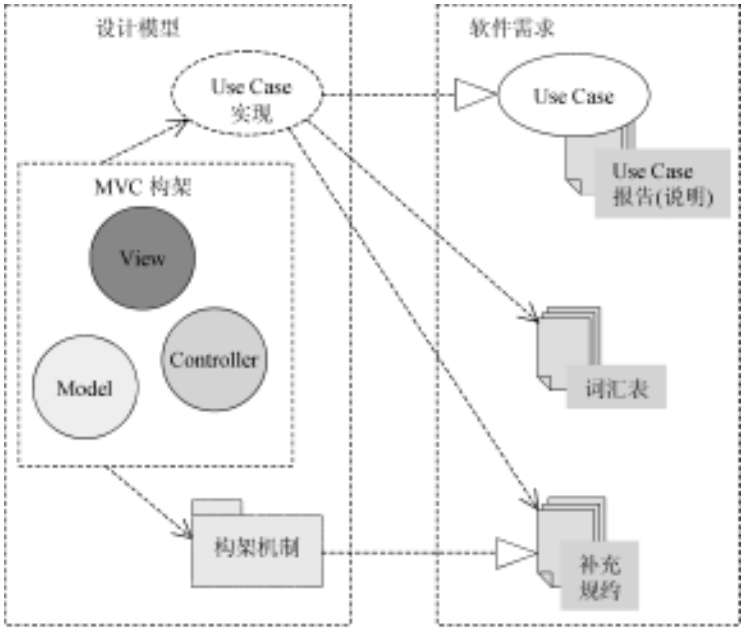


图 4-4　设计模型的组成以及和软件需求的关系（MVC 为构架模式）

经过人们反复的实践、总结和提炼，形成了所谓的构架模式，比较常见的诸如层次构架模式，MVC 构架模式等，不同的构架模式各有特色和针对性，不同构架模式的应用场合之间并不互相排斥，不同的构架模式有可能同时适用于一个拟建系统的设计，甚至可以根据需要在同一个拟建系统的分析和设计中应用多种构架模式。

层次构架是本书实践过程所选用的构架模式，尤其适用于中、大型系统的分析和设计。分层的基本原则是越靠下的层次中所包含的内容越具有一般（普遍）性，或者说与软件需求中特定应用逻辑的关系越松散。这种特征带来的直接价值是提高日后重复利用设计结果的可能性和可操作性。实践中可以结合实际情况决定适宜的层数以及层次界定的内涵。

图 4-3 给出一种典型的划分方式，包括四个层次。

- 特定应用层。包括那些仅仅与当前应用逻辑相关的设计要素及组合。
- 一般应用层。包括那些不仅在当前应用中有价值，在其他相关应用中可能具有重复利用价值、并且不属于纯粹软件技术范畴的“设计元素”及组合。

- 通用服务层。包括那些和应用领域无必然关系、属于软件技术范畴（不涉及针对操作系统软件问题）的“设计元素”及相关组合。例如解决跨平台操作问题的“设计元素”及其组合。
- 系统服务层。包括那些用于提供（针对操作系统软件问题的）基础性服务的“设计元素”及相关组合。例如解决设备驱动的设计元素及其组合。通常的软件系统不会涉及这一层次内容的设计和开发，鉴于这个层次的内容具有很高的复用能力，通常可以获得成熟的产品。本书的后续内容将忽略这个层次的内容。

层次构架中的内容并非一蹴而就，而是一个渐进的积累和完善过程。在本书实践过程中，对层次构架“结构性贡献”最显著的是两项全局意义的任务，即“全局分析”和“全局设计”，它们是系统构架师的主要职责。图 4-5 给出了前三项任务对典型层次构架中较高层内容的“贡献”。注意，在分析阶段，与应用逻辑不直接相关的较低层次中没有内容。

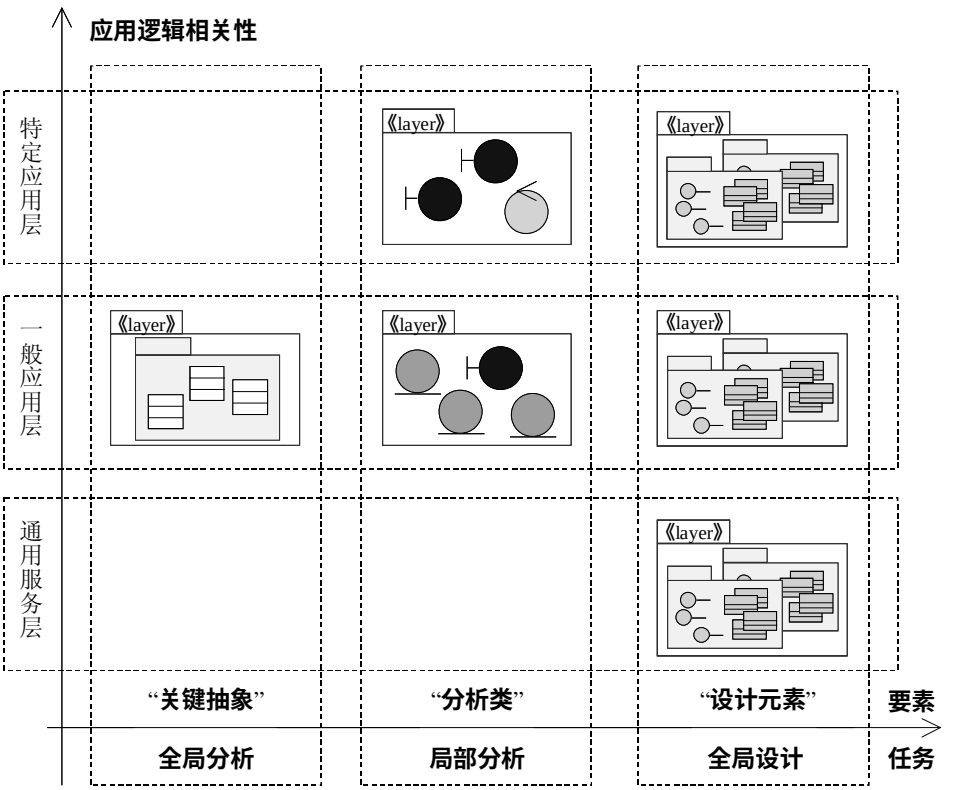


图 4-5　层次构架的逐渐充实过程示意

“Use Case 实现”

“Use Case 实现”作为设计模型的一部分，描述了一组对象的协作关系，用于实现特定 Use Case 表述的软件需求。对象协作关系的动态表述（序列图和协作图）与 Use Case 中的应用场景（Scenario）有直接的对应关系；多个场景将丰富的状态和行为特征赋予参加协作的对象；这些特征的静态描述被综合为“参与类图”；“参与类图”中记述的关系是层次构架中“设计元素”关系的基本依据。图 4-6 给出“Use Case 实现”所包含内容的示意。

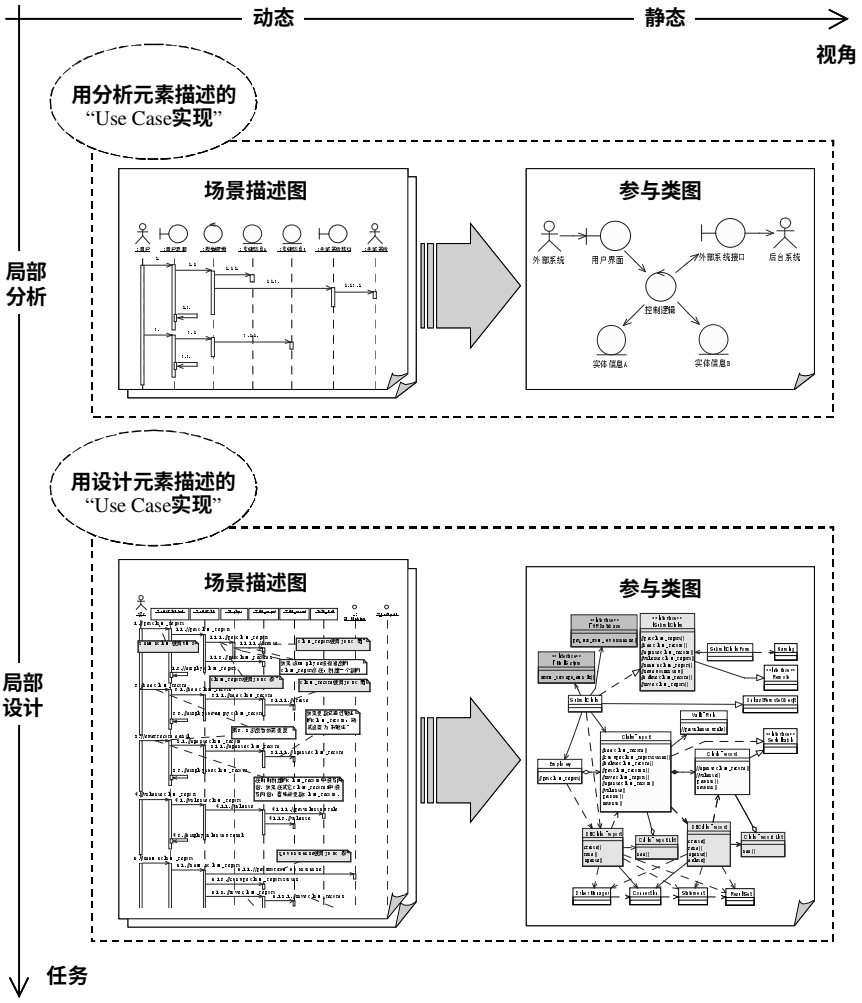


图 4-6 “Use Case 实现”内容示意

“Use Case 实现”是功能需求向设计方案（层次构架中的内容）过渡的核心纽带，这一概念贯穿整个实践过程。充实“Use Case 实现”内容的主要任务集中

于“局部分析”和“局部设计”。在不同阶段，用于描述“Use Case 实现”的元素粒度和性质有所不同：在“局部分析”任务中应用分析元素（“分析类”），侧重于对需求作面向对象的转述，而在“局部设计”任务中应用“设计元素”（“设计类”和“子系统接口”），侧重于对实施环境的适应和折衷。

需要强调，参与“Use Case 实现”的分析元素或“设计元素”并不隶属于某一个“Use Case 实现”包，它们以层次构架中的包为栖身之地。在特定的“Use Case 实现”上下文当中，这些元素通过各种“图”展现其动态和静态特征。换言之，“Use Case 实现”为设计活动提供了一种挖掘信息、记录需求与构架内容追溯关联的媒质。“Use Case 实现”并不是最终的设计结果，但重要性不亚于最终的设计结果。因为它记录了需求到设计结果之间的映射关系，能够确保设计结果具有明确的根据，或者说具有可维护性[1]。这一点对于中、大规模的软件开发至关重要。

“构架机制”

“构架机制”（Architectural Mechanism）是解决常见软件技术问题的设计经验描述，包括模式化的结构特征和行为特征。基于面向对象设计的视角，“构架机制”是应用于一组类上的、能够解决特定类型问题的模式[2]。

举一个通俗的例子，假设你准备编写一个新的应用程序 B，其中的“数据存取”问题在以往的应用程序 A 中曾经被很好地解决；于是你会将应用程序 A 中的那部分代码 X"复制过来，简单地改头换面（变成 X'）之后，就能够解决应用程序 B 中的相似问题。类似地，你还可以在应用程序 C 中将 X 改换成 X"解决相似的 1 问题。那么，X、X 和 X"中相同的部分以及 X 被改换成 X'或 X"过程中的经验和规律就构成了一个解决“数据存取”问题的“构架机制”。“构架机制”描述了可复用的设计经验，大大地简化了应用程序 B 和 C 的分析和设计过程。当然，机制不仅可以来源于自己的成功经验，更多地是来自于别人的或通行的成功经验。

“构架机制”基于通用标准、原则和实践经验，在整个项目的实施过程中应当被视作通行的规范。团队成员处理相似问题的时候应当复用统一的框架和做法。这样有利于屏蔽潜在的沟通障碍和无效的重复建设，从而大幅度提升劳动生产率。

当拟建系统比较复杂的时候，“构架机制”将具有概念、逻辑和物理多个层次的含义。“构架机制”是“分析机制”（Analysis Mechanism）、“设计机制”（Design Mechanism）和“实施机制”（Implementation Mechanism）的统称。

“构架机制”在不同的任务阶段渐进地展现必要的细节，是将软件非功能需求融入层次构架的重要纽带。

“分析机制”基本上停留在概念层面（Conceptual），是某种复杂行为的快捷表述方式，具有更多的符号含义；“设计机制”拥有比较具体的内容（Concrete），通常涵盖一部分实施环境的细节，能够说明可行性；“实施机制”对应真实的实施环境（Actual），针对特定的技术和厂商，能够明确地指导实施。简单讲，“分析

[1] 设计的变化和需求的变化有明确的对应和追踪关联。
[2] 广义的模式针对某一类普遍问题。

机制”是“构架机制”在分析任务中的表现形式，“设计机制”是“构架机制”在设计任务中的表现形式，“实施机制”是“构架机制”在实施任务中的表现形式。例如“留存”（Persistency）是一种非常典型的“分析机制”，“RDBMS”是能够实现“留存”“分析机制”的一种“设计机制”，而“JDBC”是能够实现“RDBMS”“设计机制”的一种“实施机制”。

系统构架师负责决定“构架机制”的选用和维护，并说明如何将“构架机制”的实例平滑地“嵌入”构架之中。

设计模型的演进

根据前面的讨论可以看出：层次构架中的内容是分析和设计工作为后续开发活动提供的依据，是设计模型的核心内容；从“应用逻辑牵引”和“软件技术支撑”两个方面，“Use Case 实现”和“构架机制”描述了软件需求向设计方案的平滑过渡历程。

在本书的实践过程中，完成特定任务的标志是得到相关的设计模型内容。设计模型中，三部分内容的充实过程相辅相成，表 4-1 给出一个概要的描述。读者在完成后续每一章内容时，可以对照该表总览渐进充实的设计模型内容，同时体会不同任务和活动时设计模型的针对性贡献。希望该列表的内容能够帮助读者在整体的高度上、以面向结果的视角纵览实践过程的演进。

4.4 示例软件需求说明

为了帮助读者理解应用建模实践过程，本书提供贯穿全过程的示例。本节内容介绍示例所依据的软件需求，包括问题说明、软件功能需求总览、具体的局部功能需求、（作为非功能性需求主体的）“补充规约”要点、以及“词汇表”要点。这些信息只是软件需求的一个子集，用于帮助读者深度遍历应用建模实践过程中的概念与步骤。

问题说明

啟元公司的总部设在北京，在上海、广州、成都和西安有分支机构，全公司接近 700 名员工。鉴于业务和员工团队的快速发展，为了提升整体工作效率，啟元公司准备开发一套员工报帐系统，取代原来的人工处理方式。

报帐系统将支持员工记录（或预见）日常业务活动的开销，并自动结算每个月应该返还员工的补偿金额，补偿额会直接存入员工的工资户中。

报帐系统应具有基于先进技术的图形化界面，员工可以输入业务活动的种类和简短描述，活动开销的类别，选择不同的支付方式，并可以生成灵活的报表。

报帐系统应该有能力根据员工提供的信息和要求返还补偿额，同时保存全部员工的报帐信息。员工可以通过他们自己的电脑来使用报帐系统。由于牵涉到财

务信息，报帐系统必须提供可信的安全机制。

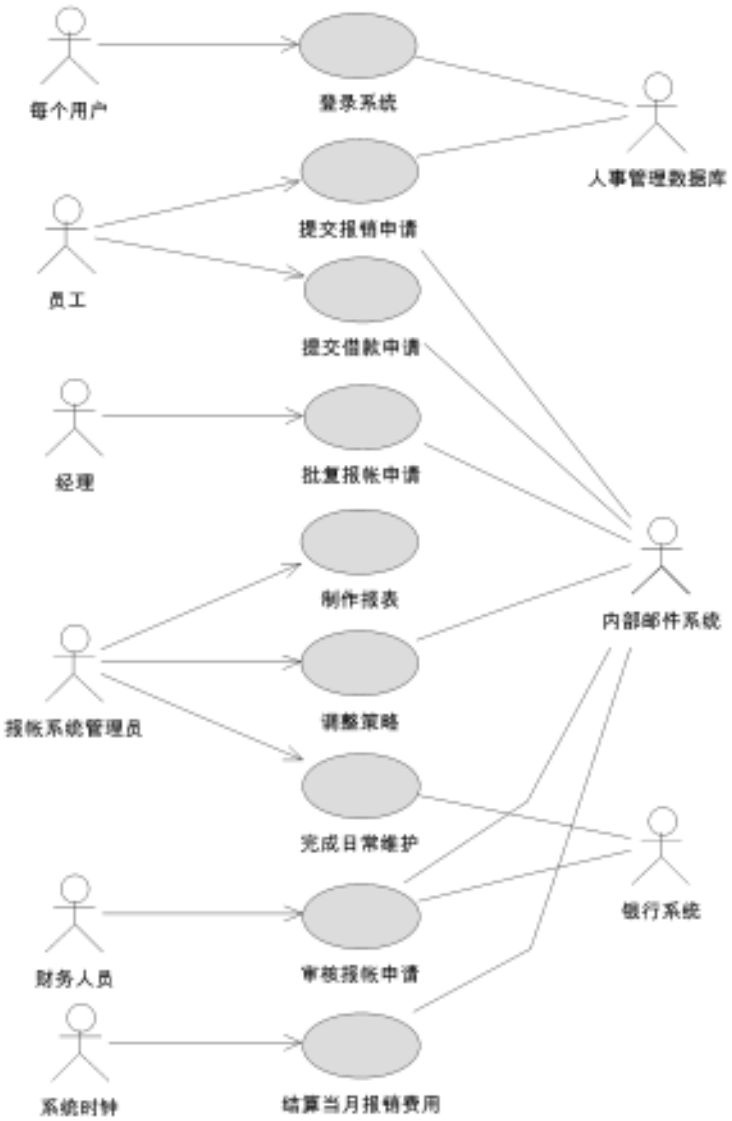


图 4-7　**示例系统的** Use Case **图**

启**元公司现有一套基于微软** SQL Server **的人事管理数据库系统，记录员工的基本信息和团队的组织结构。报帐系统将和现有人事管理数据库系统协同工作，需要引用人事管理数据库系统中的部分信息，但不会更新其内容。**

表 4-1　设计模型内容的演进

任务	活　动	设计模型内容		
		“Use Case 实现”	层 次 构 架	“构架机制”
全局分析	选用构架模式		● 表述层次构架的包的构造型	
	识别关键抽象		● 表述“关键抽象”的类（通常位于“一般应用层”中的“关键抽象”包）	
	标识分析机制		● 表述“关键抽象”与“分析机制”映射关系的注释信息（文字）	● 表述“构架机制”的包（以“分析机制”命名）
	选定分析局部	● 表述“Use Case 实现”结构的包（以 Use Case 命名）表述可追溯关联的 Use Case 图		
局部分析	提取分析类		● 表述分析元素“分析类”（通常位于“特定应用层”和“一般应用层”） ● 表述“分析类”与“分析机制”映射关系的注释信息（文字）	● 表述“分析机制”技术特征的注释信息（文字）
	转述需求场景	● 表述 Use Case 事件流场景的交互图组（以“分析类”的实例作为交互的对象）	● 表述“分析类”实例在交互中响应消息而承担的“责任”（类的操作雏形）	
	整理分析类	● 表述交互参与者相应的“分析类”的“参与类图”	● “分析类”由于承担“责任”而具备的属性	

（续）

任务	活　动	设计模型内容		
		“Use Case 实现”	层 次 构 架	“构架机制”
全局设计	确定核心元素		● 用于替换“分析类”的“设计元素”（“设计类”和“子系统接口”） ● 表述“子系统接口”依赖关系的类图	
	引入外围元素		● 用于支撑“设计机制”的“基础设计元素”（通常位于“通用服务层”）	● 构造型为《Role》的类 ● 表述“设计机制”协作关系的序列图组和“参与类图”
	优化组织结构		● 表述层次和包之间依赖关系的类图 ● 表述包内部“设计元素”关系的类图	
局部设计	实现需求场景	● 表述 Use Case 事件流场景的交互图组（以“设计元素”的实例作为交互的对象） ● 表述交互参与者相应的“设计元素”的“参与类图”	● 用于落实“设计机制”的“衔接设计元素”	
	实现子系统接口		● 用于实现“子系统接口”的“设计元素” ● 表述子系统行为和结构的交互图组和“参与类图”	
细节设计	精化属性和操作		● 被精化的“设计类” ● 表述某些“设计类”状态特征的状态图	
	精化类之间关系		● 被精化的关系（反映在各种类图中）	

在保存当月报销单之后直接退出系统。

[返回位置]：同“起始位置”。

A5 报销记录不合理

[起始位置]：基本事件序列中，“验证报销单”步骤中对每一条报销记录验证结束之后。

[触发条件]：报销记录不满足某一条适用的准则。有两种情形：第一，某报销记录的金额超出了其对应类型费用的上限，已知有三种：请客户用餐人均超过 300 元，出差时每天住宿费超过 800 元，移动电话费在无特殊说明情况下超过 800 元；第二，报销费用的类型和员工所处的部门及职能不匹配，已知的情形是业务部门的员工申请加班补助。

[具体内容]：告知员工不合理的报销记录编号，以及未通过验证的原因。

[返回位置]：基本事件序列中的“填写报销单”步骤，目的是更正有问题的报销记录。

A6 人事管理数据库不可用

[起始位置]：基本事件序列中，“提交报销单”步骤的结尾。

[触发条件]：当报帐系统向人事管理数据库索取信息而该数据库没有正常的响应。

[具体内容]：以对话框形式告知员工“人事管理数据库不可用，报帐单没有提交成功。”

[返回位置]：Use Case 执行结束。

A7 邮件未及时发出

[起始位置]：基本事件序列中，“提交报销单”步骤的结尾，成功地从人事管理数据库获得相关信息之后。

[触发条件]：报帐系统要求发送相关邮件时，邮件系统没有及时的响应。

[具体内容]：系统将以提示信息的方式告知员工，“邮件没有及时发出，但是报销单在系统内已经提交成功，待邮件系统恢复后，相关邮件会自动发出。”

[返回位置]：Use Case 执行结束。

特殊需求列表（专属于该 Use Case）

暂无。

启动条件

员工成功登录系统，通过身份验证。被系统提示进入“报销申请”或者“借款申请”功能。

结束状态（组）

如果该 Use Case 顺利执行，员工的报销申请记录将被建立，更新、保存或者

保存并提交；否则，系统的状态应该保持和该 Use Case 执行之前相同。

Use Case 图

参见图 4-8，这张 Use Case 图以 Use Case“提交报销申请”为关注焦点。

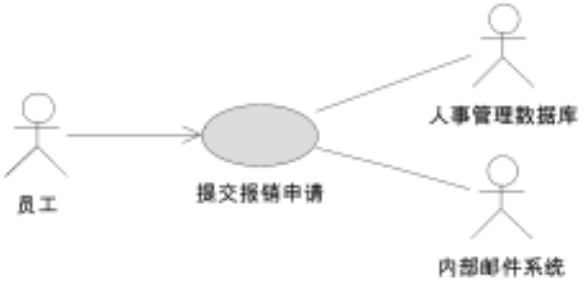


图 4-8 “提交报销申请”Use Case 图

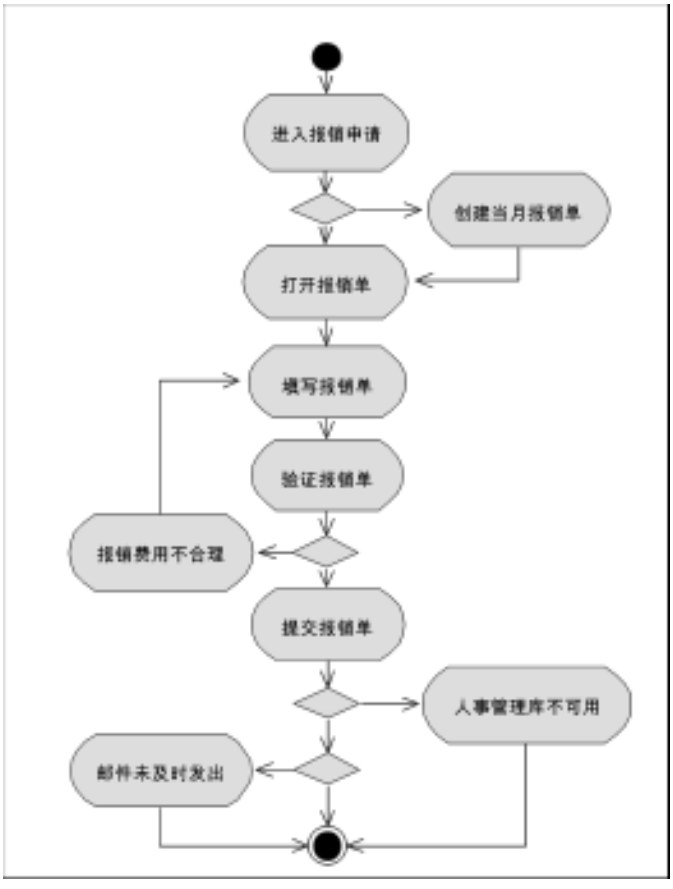


图 4-9 “提交报销申请”Use Case 的事件流结构

辅助图示（可选）

参见图 4-9，活动图帮助读者快速理解“提交报销申请”的整体事件流程。

“补充规约”要点

为了突出重点，在后续的实践过程中只考虑两个非功能需求：用 Java 实现对关系型数据库的访问和分布式处理。

“词汇表”要点

此处仅给出实践过程中的示例所涉及的关键词汇。

- 员工。敞元公司的正式雇员。
- 经理。负责审批某员工当月开销的管理者，是较高级别的员工。
- 报销记录。与业务有关的的某一项具体的花费，包括业务活动发生的时间、地点、客户名称（可选）、原因以及费用金额和种类（交通、餐饮、会议、通信和杂项）。
- 报销单。员工在一个（自然）月内的所有报销记录的集合。
- 工资户头。公司将员工用于日常业务活动开销的补偿金额返还至员工的银行帐户，该帐户的基本功能是供员工接收工资。
- 人事管理数据库。该数据库中记录了有关人事管理的相关信息，与报帐系统有关的是公司的组织机构（“员工”和“经理”的关系）。
- 内部邮件系统。该邮件系统负责收发与公司业务有关的电子邮件信息。

第 5 章　全 局 分 析

“ 全局分析 ”侧重于定义拟建系统所采用的构架以及影响构架的要素。“ 全局分析 ” 充分利用相似系统或问题中的经验，避免在确定构架上浪费人力和物力。

在 “ 全局分析 ” 任务中，有侧重不同的四项活动，参见图 5-1。

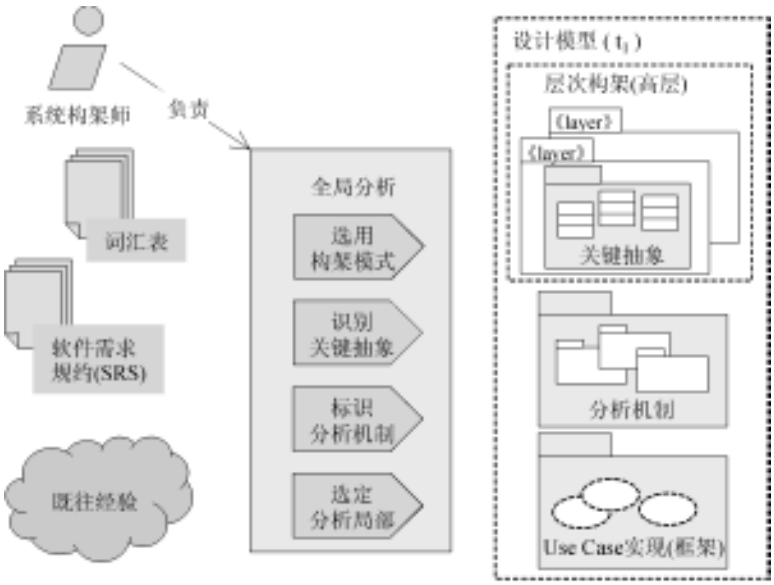


图 5-1　“ 全局分析 ” 任务的责任人 － 依据 － 活动 － 结果[□]

- 选用构架模式。在本书范围内，即指采用层次构架模式并定义其高层次内涵。
- 识别关键抽象。寻找那些无论在问题领域和方案领域都具有普遍意义的概念点。
- 标识分析机制。将那些和问题领域（应用逻辑）没有直接关联的计算机概念及相应的复杂行为表述为支撑（简化）分析工作的 “ 占位符号 ”。

□ 本书描述 “ 任务 ” 框架的图示包括四方面的内容：任务的 “ 责任人 ” 位于图示的左上角，任务的 “ 依据 ” 位于图示的左侧，任务中执行的 “ 活动 ” 位于图示的中间，任务的 “ 结果 ” 位于图示的右侧。

- 选定分析局部。针对拟建系统的整体构架，找出那些蕴含相对高风险的局部作为此次迭代的工作内容。

其中，“选用构架模式”和“选定分析局部”的主要目的是有步骤地演进稳定的系统框架；“识别关键抽象”的主要目的是从“点”入手，将软件需求内容逐渐转变为设计方案的内容；“标识分析机制”是保障软件设计方案将来能适应实施环境的最初努力。

5.1 选用构架模式

“选用构架模式”活动的主要依据是既往的经验；在本书的实践过程，该活动的结果是层次型构架模式，现阶段初步界定较高层次。参见图 5-2。

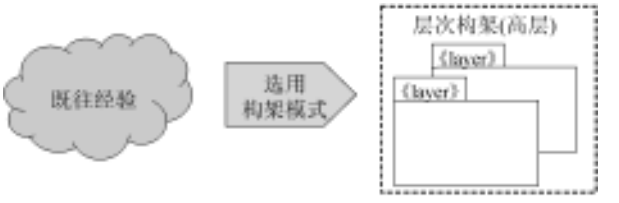


图 5-2 “选用构架模式”活动图示[1]

5.1.1 概念：构架的沿用

表面上看，“选用构架模式”活动只是在模型中摆放了一个没有内容的“空架子”；而实质上，它为所有后续活动设立了一个共有的基础框架，用以承载逐步演进和累加的设计内容。在特定的阶段，框架本身的重要性远远超出可能出现在框架中的内容。现实中，比较常见的误区是急于求成：在没有明确框架的约束下，搜集大量的内容，杂乱无章，事后重新构筑框架，其难度和负担将成倍增长，把握框架的主动性也变得非常有限。

构架模式的核心是根据某种规则将关注点在宏观上作一个区隔，目的是确保构架在后续活动中稳定地被充实，同时促进构架中的内容更易于被复用。本书的实践过程中，选用层次型构架模式进行讨论；事实上，选用其他种类构架模式的道理和方法是类似和相通的。

[1] 本书描述“活动”框架的图示包括三方面的内容：活动的“依据”位于图示的左侧，活动的名称位于图示的中间，活动的“结果”位于图示的右侧。

5.1.2 步骤 1：选用构架模式

参照 Rational 统一过程的指导，为帮助读者沿着一条清晰的思路理解面向对象分析和设计的实践过程，本书采用层次型构架模式。层次型构架模式是一种具有普遍适用性的构架模式，尤其适用于中、大型系统的面向对象软件开发。

5.1.3 步骤 2：定义构架的应用逻辑相关部分

鉴于选用层次构架，“定义与应用逻辑相关的构架”的具体含义就是定义层次构架中的较高层。在“全局分析”任务中，存在很多未知因素，层次的界定在很大程度上只能是一种假设。由于前期的工作重点是对问题本身的“分析”，因而只须相对明确地界定层次构架的较高层，即“特定应用层”和“一般应用层”，这两个层次将承载那些与应用逻辑密切相关的要素。当前阶段作出的层次界定将在后续活动中得到验证和调整。假如此时对较低层次作出界定，付出无效劳动的可能性比较高，因为层次构架中，较低层次的界定往往有赖于较高层次对较低层的具体服务要求。

5.1.4 技巧：划分层次的经验规则

逻辑上，层次构架根据内容的通用性程度高低，将要素划分成若干集合，形成层次的概念。层次之间的关系取决于要素之间的关系；同时，层次的划分也会影响要素之间的关系的调整，进而推动整个构架具有更显著的高内聚、低耦合特征，从而更具有延展性和更易于维护。以下是一些划分层次经验规则，可以借鉴。

- 层次构架的层数。层次构架的层数主要取决于系统构架师基于经验的判断，一个比较重要的原则是考虑整个构架中可能容纳的要素数量，参考表 5-1。在分析和设计活动的初期，构架中要素的个数只能依靠系统构架师粗略的估计和判断。只有合理地增加层数，才能达到简化问题的基本目的，盲目划分过多的层次通常会适得其反。
- 要素间的关系。通常情形 ，某一层次内的要素只与同层以及相邻下一层的要素存在依赖关系；否则，系统构架将不易于扩展和维护。
- 要素的稳定性。层次越高，其包含要素的稳定性相对越低，越直接反映问题领域（应用逻辑）的内容及其变化；相反，在较低层次中，那些对应纯粹软件概念的要素会相对稳定得多。

表 5-1　层次构架层数的经验规则

要素个数	0～10	10～50	25～150	100～1000
层次个数	1（不分层）	2	3	4

5.1.5 技巧：层次内分区的出发点

在层次构架的某一层次内部进一步分区，对于规模较大系统而言很有裨益，以下是一些考虑分区的出发点，可以单独或结合在一起被采用。

- 功能的必然性水平。将那些可选功能与必须实现的功能加以区分。
- 开发团队（小组）的技能专长，包括问题领域和软件技术领域。
- 拟建系统的物理分布方案。这种划分方案使物理节点之间的通信问题更加明确，适合于拟建系统的部署方案比较稳定的系统构架。
- 信息的保密级别。可以根据安全访问的权限进行分区。
- 拟建系统用户的组织机构，适用于解决业务问题的应用系统。企业通常具有明确的组织机构划分，在设计初期，这种划分通常可以被用作“分区”的依据。鉴于用户组织结构有可能重组，这种划分原则不宜用作构架层次内分区的长期基础。随着设计工作的深入，用户的组织机构应该和系统的内部要素的组织结构松散对应。

5.1.6 示例

层次构架是设计模型的核心内容的框架。在层次构架中，层次的语义用包的构造型《layer》表达。参见图 5-3，层次构架在设计模型中，设计模型在逻辑视图中。

图 5-3　层次构架在模型中的位置

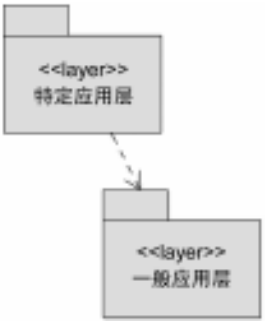


图 5-4　层次之间的关联

在初步确立两个较高层次之后，还需要建立一张类图，反映层次之间依赖关系。图 5-4 是图的内容，图 5-3 中反映了图（“层次关联”）在模型中所处的位置。

5.2 识别“关键抽象”

“识别关键抽象”活动的主要依据是词汇表、Use Case 报告（组）和既往的经验；该活动的结果是一组“关键抽象”。参见图 5-5。

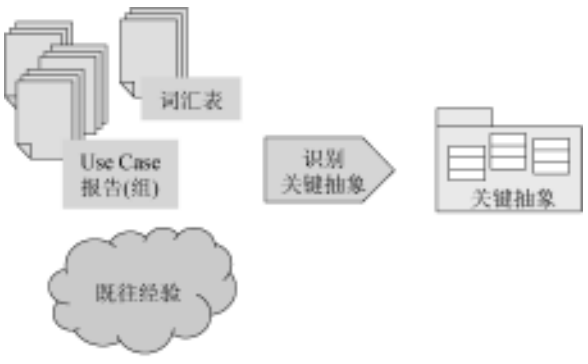


图 5-5 “识别关键抽象”活动图示

5.2.1 概念：“关键抽象”的含义

业务需求和软件需求中通常会揭示拟建系统必须处理的核心概念，这些概念同样将成为设计模型中的核心要素。我们称之为“关键抽象”，它们就是那些能够始终贯穿分析和设计的类及相应对象。根据一般经验，“关键抽象”往往对应重要的实体信息。

“关键抽象”针对全局范围。如果在进入“局部分析”任务之前没有提取这些全局性的“关键抽象”，在不同的局部很可能（甚至是必然）会出现诸多类似的抽象内容。比如，在不同的局部，“经理”、“老板”、“上司”、“领导”，“管理者”或“头儿”所表达的含义可能都是同一个“关键抽象”，如果不在全局范围统一定义和命名，势必招致无谓的麻烦。

5.2.2 概念：“关键抽象”的沿用

“识别关键抽象”活动的目的是确定拟建系统必须处理的核心概念。概念上，“关键抽象”是部分“分析元素”的前身和雏形。“分析元素”即指“局部分析”任务中获取的“分析类”。“部分”的含义说明“关键抽象”通常只包括那些可能

出现在系统多个局部的“分析元素”。换言之，在“识别关键抽象”活动中，应该关注那些具有全局影响的概念。

在全局高度上考虑问题，没有必要过分关注“关键抽象”的细节，应当结合进一步丰富的上下文信息充实相关内容。

5.2.3 步骤 1：搜集“关键抽象”的来源

“识别关键抽象”并不是从零开始的工作，应该最大限度地利用已有的劳动成果。充分搜集“关键抽象”比较集中的资料是事半功倍的做法。

比较典型的来源有三种。

- 词汇表，面向用户和软件投资者的、微观层面的、权威的术语定义集合。
- 软件需求规约（Software Requirements Specifications）中的 Use Case 报告集合，其内容是基于用户视角的应用情境。
- 反映领域知识的既往经验，这部分内容通常能提高“关键抽象”的识别效率和质量。

5.2.4 步骤 2：识别“关键抽象”

就建模而言，“识别关键抽象”的过程并不复杂，主要有三个要点。

- 从上述来源中找出候选的“关键抽象”集合，根据“关键抽象”的基本含义作出相应取舍。
- 将被确认的“关键抽象”以类的形式加入设计模型，为每个“关键抽象”作简要文字说明。通常将它们放在“一般应用层”的“关键抽象”包中。
- 将“关键抽象”绘制在一张（或者多张）描述类之间关系的类图中，标识现阶段能够确认的关系。这张类图和“关键抽象”位于同一个包。

5.2.5 技巧：“关键抽象”包的价值

鉴于“关键抽象”用于标识拟建系统中必须处理的核心概念，它们通常具有较高的一般性而并不限于针对某一特定局部问题。通常，可以将“关键抽象”放在“一般应用层”[1]的“关键抽象”包中。当然，这并不是硬性规则，实践中可以灵活处理。“关键抽象”包是过渡性的，当与“关键抽象”概念对应的内容成为实际的分析和“设计元素”之后，它们将被移动到构架中更为恰当的位置。假设挖掘的“关键抽象”确实能反映拟建系统中必须要处理的重要概念，在经历几个迭代之后，如果“关键抽象”包中仍留有未转化成分析和设计元素的内容，那么比较重要的问题很可能还没有被触及。

[1] 假如你采用 MVC 的构架模式，那么这些“关键抽象”通常会被放置在“Model”中。

5.2.6 技巧：利用业务模型

如果在分析和设计之前进行过充分的业务建模工作并得到相对完整的业务模型，“识别关键抽象”活动可以获得更直接而有效的依据。尤其是所谓的“领域模型”，它包括和拟建系统直接相关的业务对象模型。

5.2.7 技巧：利用成熟的领域经验

如果能找到现成的分析模式，可以显著提升“识别关键抽象”活动的效率和质量。分析模式中通常会记录在特定环境中建模所需的重要抽象概念。当然，领域和系统的新颖性在很大程度上决定着分析模式的成熟度和可用性。

参考权威机构制定的规范，系统构架师能够得益于成熟的领域最佳经验。国际对象管理组织正尝试通过其下属的专业领域技术委员会（Domain Technology Committee）及相关力量定义用于多个业务领域的接口和规程。

5.2.8 示例

以下列出部分“关键抽象”内容作为示例。

- 报销单(Claim_report)。
- 报销记录(Claim_record)。
- 员工(Employee)。
- 经理(Manager)。
- 工资户头(Payroll_account)。

…

示例的模型中，与后续开发活动直接相关的要素均采用英文命名。“关键抽象”被放置在“一般应用层”的“关键抽象”包中，为了说明这些“关键抽象”之间的关系，添加一个取名为“关键抽象概念间关系”的类图，参见图 5-6。

图 5-6 “关键抽象”在模型中的位置

具体的内容参见图 5-7。该图表达的基本的含义是：一个“ 经理 ”手下有多名“ 员工 ”，一个“ 员工 ”对应一个“ 工资帐户 ”，每个员工拥有多张“ 报销单 ”，每张“ 报销单 ”中包含多条“ 报销记录 ”。

图 5-7 “ 关键抽象 ”之间的关系

5.3 标识“ 分析机制 ”

“ 标识分析机制 ”活动的主要依据是“ 补充规约 “ 、既往经验以及前一活动获得的“ 关键抽象 ”；结果是一组被识别出的“ 分析机制 ”。参见图 5-8。

图 5-8 “ 标识分析机制 ”活动图示

5.3.1 概念：“ 分析机制 ”的含义

“ 构架机制 ”表述常见问题的通用解决模式，“ 分析机制 ”是“ 构架机制 ”的概念层面表述。在分析过程中，“ 分析机制 ’向设计人员提供复杂行为的简明表述，

降低（全局）分析活动的复杂性并提高（局部）分析活动的一致性。通过这些机制，可以使分析工作更有重点。借助“分析机制”，可以姑且不深究那些用于支撑核心功能（应用逻辑）但其自身并非核心功能的复杂行为，这些内容通常是解决特定软件技术问题的设计经验。

“分析机制”是“构架机制”在分析任务中的表现形式。“分析机制”提供概念化的服务模式，“分析类”将使用这些服务模式的实例。“分析机制”的实例在系统构架中充当某些复杂行为的“占位符”。运用“分析机制”，可以避免分散“全局分析”任务的工作重点。一个比较典型的例子是数据存取问题，事实上这并不是一个简单的问题，但却是一个很普遍的问题，如果在分析和设计的初期就陷入数据存取功能的细节，势必（严重）影响设计人员对整体构架的把握。在“全局分析”任务中，可以利用一个称作“留存”（Persistency，亦作“永久性”）的“分析机制”封装相关的技术细节，这样做对“全局分析”的进展大有裨益。

“分析机制”通常与应用逻辑中的特定内容关系松散，“分析机制”大多只涉及计算机软件技术的概念和要素。“分析机制”为那些与应用逻辑密切相关的要素提供必要的软件技术支撑，根据层次构架的基本原则，用于实现“分析机制”的设计内容大多位于构架的中低层。

5.3.2 概念：常见的“分析机制”

“分析机制”是“构架机制”在“全局分析”阶段的表现形式，以下是一些比较常见的“分析机制”，从命名中很容易理解它们各自针对的问题。

- 留存。
- 分布式处理。
- 安全性。
- 进程间通信。
- 消息路由。
- 进程控制与同步。
- 交易事务管理。
- 信息交换。
- 信息的冗余。
- 错误检测、处理和报告。
- 数据格式转换。

5.3.3 概念：“分析机制”的沿用

“分析机制”概括地说明如何实现必须具备的基本功能，并不需要考虑部署平台及实施语言。通常，可以采用多种方式设计和实现一种“分析机制”；换言之，对一种“分析机制”，可以通过多种“设计机制”实现；类似地，一种“设计机制”又可以被多种“实施机制”实现。“分析机制”具有更多的符号意味，随着分析和

设计活动由“面向需求”朝“面向实施环境”推进，“分析机制”将被与之相应的一个或多个“设计机制”及相关“实施机制”所实现。

如果立足于设计视角，“构架机制”将实现为一组类的协作[□]。其中一部分类将和那些对应拟建系统核心功能的类绑定，另一部分类并不直接参与实现核心功能，而是起到支撑的作用。通常，这些起支撑作用的类是实现“构架机制”复杂行为的“中坚力量”。

5.3.4 步骤 1：确定“分析机制”

根据既往的经验并参照“补充归约”的内容，系统构架师估计出方案中可能会遇到的软件技术问题及其解决之道。“分析机制”涉及的问题是大多数系统必须面对的一般性功能，参照“概念”中“常见的分析机制”。

类似于“关键抽象”，相似或相同的“分析机制”可能会被冠以不同的名称。鉴于“分析机制”的确定和使用将影响后续任务中的多个设计人员，应该将“分析机制”收集在一个列表中并采用统一的命名。

5.3.5 步骤 2：简述“分析机制”

首先，概述“分析机制”的主要技术特征，它们将成为以后向“设计机制”过渡的重要考量点。例如，对于解决数据存取问题的“留存”“分析机制”而言，通常需要获知粒度、容量、留存的持续时间、检索机制、更新频率以及可靠性级别等等。在后续活动中，这些技术特征的取值将直接影响相应“设计机制”和“实施机制”的选用。

然后，列出“分析机制”和“关键抽象”的映射关联表，参见表 5-2。

表 5-2 “关键抽象”和“分析机制”的映射关联

“分析机制” / “关键抽象”	“分析机制”1	“分析机制”2	…	“分析机制”m
“关键抽象”1		√		
“关键抽象”2	√			√
…		√	√	
“关键抽象”n	√			

[□] 《UML 用户指南》

5.3.6 技巧：确定“分析机制”的方式

实践中，确定“分析机制”的工作并非完全凭借既往经验以自底向上方式进行。有些时候，“分析机制”在相关的“设计机制”和“实施机制”成熟之前已经在分析活动中被引用。这种类型的“分析机制”会随着分析和设计过程的演进逐步明确，可以称之为自顶向下的方式。这种“分析机制”最初比较模糊，随着各方面问题逐步澄清而演化成相对通用的框架（完整意义上的“构架机制”）。当然，这种做法的风险和对开发团队的要求比较高。

5.3.7 技巧：抽取自己的成功经验

对于个人或团队而言，经验和经历的区别在于是否具有复用价值。现实生活中，我们不断地整理那些具有复用价值的经历，即从以往的成熟应用系统中抽取您自己的“构架机制”。不过在大多时候，我们没有明确的目标和清晰的意识。因而，很多宝贵经验只驻留在实践者的大脑之中，而并没有通过易于交流和分享的形式表达出来（例如“构架机制”）。希望读者能够最大限度地提取和表述自身的宝贵经验，发挥它们的复用价值。

5.3.8 技巧：利用他人的成功经验

经验是从多次实践中得到的知识或技能，经验的积累者和经验的使用者未必是同一主体。文献资料（主要是书籍和网站）中记录了很多有复用价值的成功经验；另外，一些权威厂商[1]的软件开发平台通常会附带很多实用性强并经过验证的应用模式，可以直接套用。

即便是一个很出色的团队也没有可能、更没有必要成为全能的专家。但是，至少应该是两个方面的专家：第一，能够明确地把握应用逻辑内容在设计中的表现；第二，能够最大限度地利用前人（包括自己，但主要是他人）的成功经验。

宏观上，应用逻辑内容在设计中的表现反映在“Use Case 实现”当中，前人的成功经验体现在“构架机制”当中，如果两方面处理得当，作为设计核心结果的体系构架内容势必“左右逢源”、“水到渠成”。

5.3.9 示例

以下列举示例中可能涉及四种“构架机制”：安全、留存、分布处理和信息交换。在设计模型中添加一个“构架机制”包，在“构架机制”包中为每一个机制建立一个以相应机制命名的包。参见图 5-9。

---

[1] IBM e-business patterns 与 Sun Java Center 都是很好的例子，详情参见http://www.ibm.com/framework/patterns/ 和http://developer.java.sun.com/developer/technicalArticles/J2EE/patterns/。

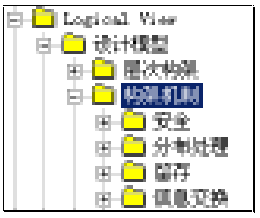


图 5-9 “构架机制”在模型中的位置

以下简单描述“分析机制”的主要技术特征。

- 安全。用户名要求、密码要求和权限要求等。
- 留存。粒度、容量、留存的持续时间等。
- 分布处理。分布模式（Tiers 结构）、时延、同步、消息粒度、协议等。
- 信息交换。信息记录格式，编码转换。

表 5-3 展现“分析机制”和“关键抽象”的映射关联。

表 5-3 “关键抽象”和“分析机制”的映射关联表

“分析机制” / “关键抽象”	安　全	留　存	分布处理	信息交换
报销单	√	√	√	
报销记录	√	√	√	
员工		√		
经理		√		
工资户头	√	√		√

5.4 选定分析局部

“选定分析局部”活动的主要依据是整体的 Use Case 模型和基于以往经验的风险判断；该活动的结果是被选定分析局部（Use Case 组）的“Use Case 实现”框架。参见 5-10。

5.4.1 概念：“Use Case 实现”的桥梁作用

Use Case 是软件需求规约中关于功能需求的核心载体，Use Case 以拟建系统使用者的目标为核心线索，将各种应用场景以高内聚、低耦合的方式组织在一起。

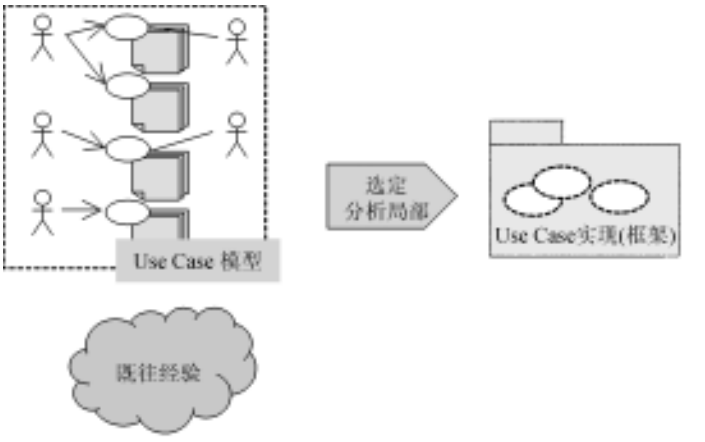


图 5-10 “选定分析局部”活动图示

“Use Case 实现”是设计模型中的重要组合部分，是从功能需求过渡到设计方案的核心纽带，“Use Case 实现”的概念贯穿整个实践过程。

为了实现从围绕需求的活动过渡到围绕设计的活动，“Use Case 实现”提供了一种将设计模型要素的行为特征回溯到 Use Case 事件序列的框架和方法。概念上，Use Case 模型中的每个 Use Case 在设计模型中应该至少有一个“Use Case 实现”与之对应。通常，“Use Case 实现”的名称中应反映出相应的 Use Case 名称，并且需要在它们之间显式地建立可追溯关联。

一个具体的“Use Case 实现”提供了一个场所，用于表达基于设计视角的 Use Case 内容，主要包括以 Use Case 事件序列为线索的动态和静态图解：一种是直接对应 Use Case 事件序列的交互图（一系列的序列图和协作图）；另一种是类图，反映参与 Use Case 事件序列的要素之间的静态关系。

将具体的“Use Case 实现”与相应的 Use Case 在组织结构上进行分离是一种重要的策略，对于中、大型的开发项目更能体现出优势。一方面，同一个 Use Case 可以有多个具体的“Use Case 实现”与之对应，这样做，可以得到拟建系统的不同变种，还可以隔离原型系统的分析和设计轨迹；另一方面，将 Use Case 与相应的“Use Case 实现”分离，变更针对 Use Case 的设计内容不会影响已设置基线的 Use Case 内容本身。

5.4.2 概念：风险前驱的迭代化开发策略

宏观上，Use Case 主要用于描述拟建系统外在可见的动态行为特征，系统构架主要用于描述拟建系统内部的静态结构特征。经验表明，并非所有的 Use Case 都会影响拟建系统构架的关键部分。或者说，不同的 Use Case 对设计方案的影响力通常并不均衡。在有些方面，多个 Use Case 对设计方案的影响力相互重叠、甚至有依赖关系。

假如按照纯粹的“瀑布式”开发思想，将软件需求向设计方案作一次性的映射，通常不能避免三方面的问题。

- 众多非重点问题分散对核心问题的注意力。
- 重复解决类似的问题，降低设计方案取得实质性进展的效率。
- 某一局部的设计被验证有问题之后，其他相关局部的工作成为无用功。

如果采用迭代化的开发思想，能够较好地避免上述问题。通俗地讲，一个迭代做一部分，然后像滚雪球一样推进。高内聚、低耦合的 Use Case 模型为实现迭代策略提供了天然的有利条件，即 Use Case 可以充当很好的任务划分单元。参见图 5-11。

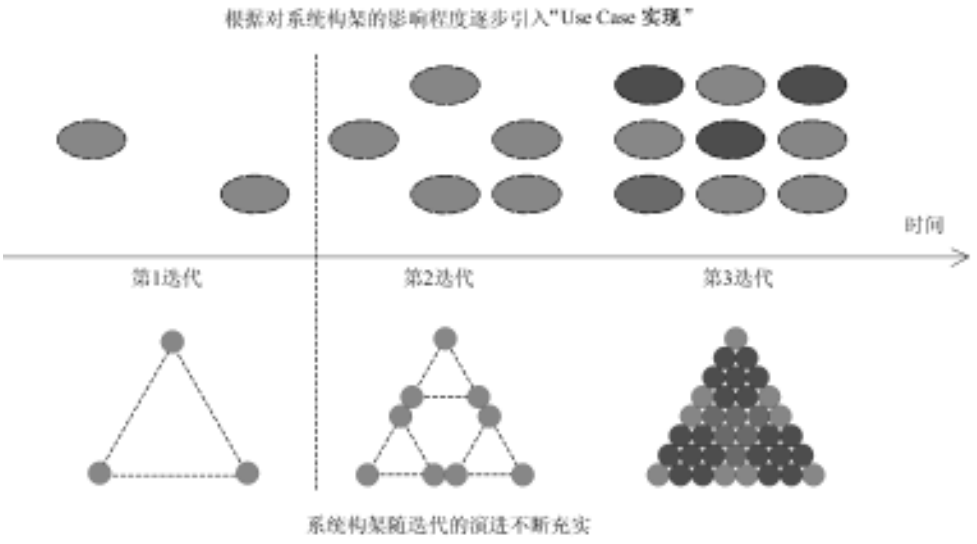


图 5-11　迭代策略的示意

笼统地讲，先做优先级高的部分。任务单元的优先级评判标准是 Use Case 所蕴含的风险，即影响拟建系统构架的能力。简而言之，风险是安排迭代内容的依据，迭代内容即是待选定的分析局部。不过，当拟建系统的构架尚不明确的时候，这种评判标准有赖于系统构架师的经验。

5.4.3　步骤 1：选定当前的待分析局部

在项目初期的可行性分析活动中通常会得到一个风险列表。首先要明确 Use Case 内容和主要风险的关联，可以通过类似表 5-4 的列表进行标识。

选定分析局部的基本原则是在覆盖主要风险的基础上，涉及的内容越少越好。覆盖主要风险的经验性原则包括两个方面。

- 覆盖全部风险的 80%。

- 覆盖前 50%（优先级）内的全部风险。

表 5-4　风险和 Use Case 内容的关系

优先级 / 风险 \ Use Case		Use Case 1	Use Case 2	…	Use Case n
1	风险 x	√		…	
…	…	√		…	√
M	风险 z		√	…	

5.4.4　步骤 2：建立“ Use Case 实现 ”框架

首先，建立“ Use Case 实现 ”包的结构。在设计模型中建立一个称为“ Use Case 实现 ”的包；在该包中，为每个被选定的 Use Case 建立一个以相应 Use Case 命名的包；在这些包中分别建立至少一个具体的“ Use Case 实现 ”。

然后，建立可追溯关联图。展示 Use Case 和与之相应的（一或多个）具体“ Use Case 实现 ”之间的实现关系，即可追溯关联。基于模型绘制者的视角，这张图看起来非常简单，似乎没有存在的必要；但是，这张图是反映设计模型和 Use Case 模型之间关联的惟一显示说明。基于模型使用者的视角，这张图起着航标的作用。请注意，模型内容的存在价值应该针对使用者，因而，模型中作为结果的部分和作为演进轨迹的部分都很重要。

5.4.5　技巧：既往经验的价值

针对迭代内容的选取，尽管在“ 步骤 ”中提供了可操作的方法；但是在实践中，需要突出系统构架师既往经验的作用，系统构架师的一些直觉判断往往具有更高的利用价值。必须承认一个基本的事实：由于风险的核心特征是不确定性，因而，风险的优先级排列具有较高的主观性。

5.4.6　技巧：复杂的未必重要

风险的优先级综合了不确定性的高低与针对关键软件需求贡献的大小。实施技术的复杂程度和风险的优先级有联系，但并没有直接对应关系。因而，复杂的未必是重要的。为了更有效地实现具有核心价值的需求，某些时候，在澄清某些复杂技术问题的高昂代价[1]之后，有可能[2]简化、推迟、甚至取消某些不重要的软

[1] 非常低的投入产出比通常意味着高昂的机会成本。

[2] 有效的沟通和关键涉众的认可是必要条件。

件需求。

5.4.7 技巧：借鉴 80－20 规则

“选定分析局部”的结果通常会符合（至少应该接近）Pareto 的 80-20 原则[□]。具体讲，20%左右的 Use Case 能够覆盖 80%的主要风险。如果按照“步骤”中的经验原则得到的“分析局部”占全局的比重过高（例如超过 50%），则有必要对评判准则进行适当的调整，否则“选定分析局部”的结果将不适宜支撑迭代开发的基本策略，“选定分析局部”活动失去实际价值。总之，借鉴 80－20 原则，调整“选定分析局部”的评判原则，可以适度弥补评判依据本身的主观偏差。

5.4.8 示例

表 5-5 给出了一个风险和 Use Case 的关系示例。

表 5-5 风险和 Use Case 的关系示例

优先级	风险 \ Use Case	调整策略	结算当月报销费用	批复报帐申请	审核报帐申请	提交报销申请	提交借款申请	完成日常维护	制作报表	登录系统
1	数据库的响应速度			√	√	√	√			
2	数据库的容量					√	√			
3	与内部邮件系统联接	√	√	√	√	√	√			
4	与银行系统联接		√		√					
5	与人事管理数据库联接					√	√			√
6	内部网络有足够带宽			√	√	√	√			√
7	数据备份							√		

不难看出，Use Case“提交报销申请”和“结算当月报销费用”几乎覆盖了全部风险，并且它们只占全部 Use Case 的很小一部分（2／9），符合一般的规律。在当前迭代中，将这两个 Use Case 选定为“分析局部”。

在设计模型中，建立“Use Case 实现”的结构框架，即命名为“Use Case 实

[□]意大利经济学家 Vilfredo Pareto 曾经作过一项研究，发现全球近乎 80%的财富掌握在 20%的

现”的包。针对当前迭代，在“Use Case 实现”包中建立以上述两个 Use Case 命名的包，然后分别建立相应的（第一个）“Use Case 实现”，参见图 5-12。“Use Case 实现”的基本语义是协作（Collaboration）。在模型中，“Use Case 实现”用 Use Case 的构造型《use case realization》描述，表现为一个虚线椭圆。

在“Use Case 实现”包中建立“可追溯关联”图，图 5-12 中展示了该图在模型中的位置，图 5-13 是“可追溯关联”图的内容。注意，这张图描述的内容说明了 Use Case 模型向设计模型的过渡；更宽泛地讲，它是从 Use Case 视图向逻辑视图过渡的纽带，是两种不同视角的交汇点。

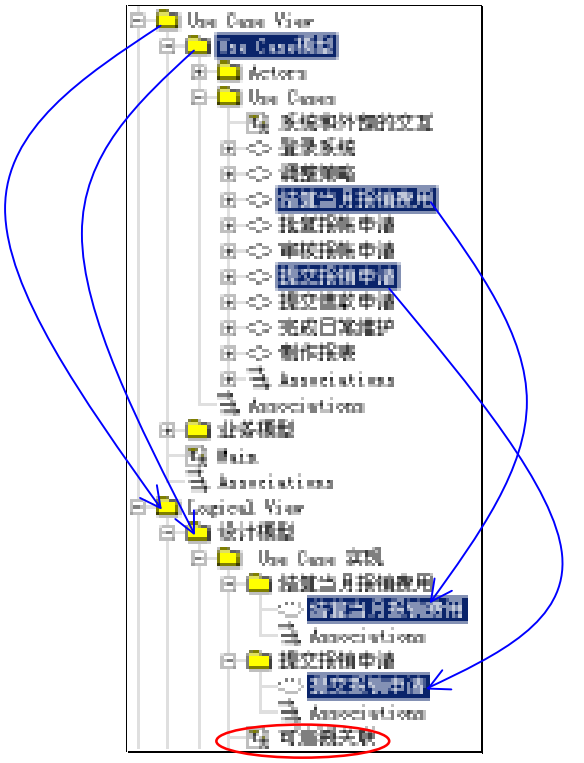


图 5-12　“Use Case 实现”的结构框架

至此，基本上完成了“全局分析”任务中的主要建模任务，为展开后续分析和设计工作准备了整体的结构框架。为了让读者更好地把握设计模型的整体图景，给出两张相关的图示：图 5-14 给出了设计模型在现阶段的结构和内容；图 5-15 描述了设计模型中三个部分之间的逻辑关系，该图取名为“设计模型总体”。

人手中。后来人们发现这条被称作 80－20 的规律，在很多领域都有类似的表现。

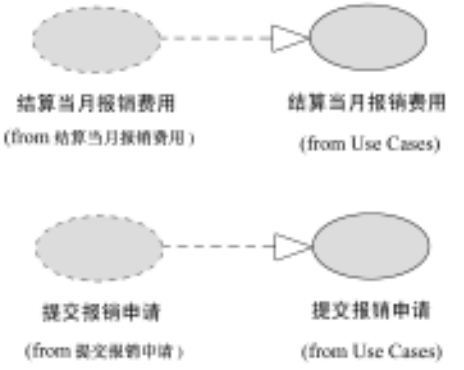


图 5-13　**可追溯关联**

图 5-14　**现阶段设计模型的主要内容**

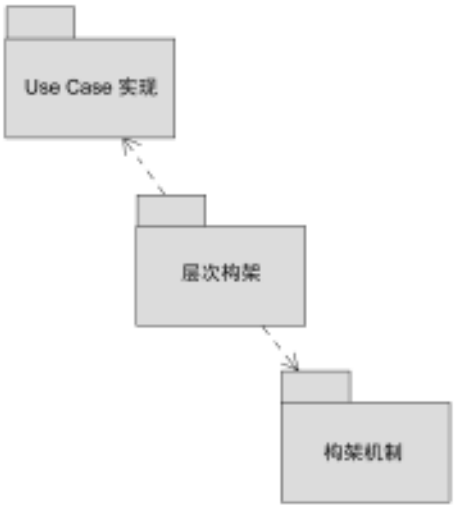


图 5-15　设计模型的整体组织结构

针对示例，表 5-6 给出了全局分析任务中积累的设计模型内容。

表 5-6　全局分析任务中积累的设计模型内容汇总

任务	活　动	设计模型内容		
		“Use Case 实现”	层次构架	“构架机制”
全局分析	选用构架模式		图 5-3，图 5-4	
	识别“关键抽象”		图 5-6，图 5-7	
	标识“分析机制”		表 5-3	图 5-9
	选定分析局部	图 5-12，图 5-13		

第 6 章　局 部 分 析

“局部分析”以选定的 Use Case 为研究对象，以相对粗大的颗粒，用面向对象的概念和方法对问题进行转述，为后续以相对细小的颗粒作进一步的设计活动提供必要的铺垫。

在“局部分析”任务中，有不同侧重的三项活动，参见图 6-1。

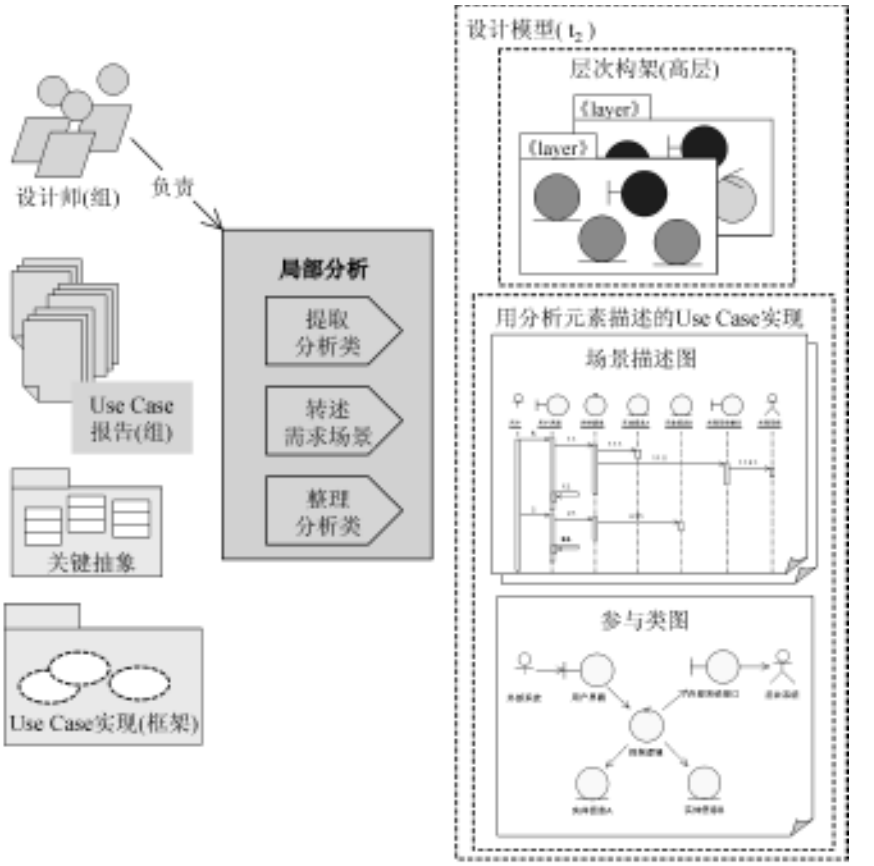


图 6-1　“局部分析”任务的责任人 — 依据 — 活动 — 结果

- 提取“分析类”。根据“全局分析”任务中获得的“关键抽象”以及特定 Use Case 的文字内容，在三个不同的维度抽取那些能够协作完成该 Use Case 行为的分析元素，即“分析类”。“分析类”可以形象地理解为一系列“点”的集合。

- 转述需求场景。用“分析类”的实例作为行为的载体，通过消息传递的方式实现对 Use Case 场景的分解。用消息传递串联表达的场景，可以形象地理解为若干条“线”。
- 整理“分析类”。根据“分析类”的实例在 Use Case 需求（若干）场景中的消息传递关系，归纳“分析类”所承担的“责任”和“分析类”之间的“关系”。对“分析类”的“责任”和“关系”的描述，可以形象地理解为一个“面”。

简而言之，以一个 Use Case 为讨论范围，上述三项活动从“点”到“线”再到“面”将软件需求逐渐转变为设计方案中的初始内容。

6.1 提取“分析类”

“提取分析类”活动的基本依据是当前“分析局部”所对应的 Use Case 报告内容以及在“全局分析”任务中得到的“关键抽象”；该活动的结果是当前分析局部涉及的“分析类”，这些“分析类”分布在层次构架的较高层中。参见图 6-2。

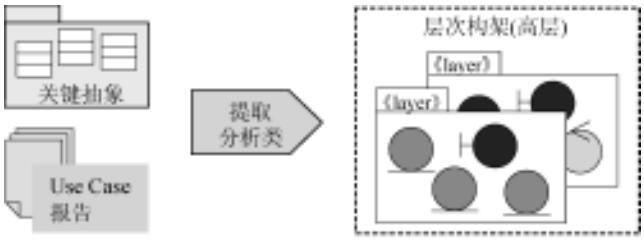


图 6-2 “提取分析类”活动图示

6.1.1 概念：“分析类”的含义

“分析类”是概念层面的内容，与应用逻辑直接相关。“分析类”的实例所具备的行为，用于捕获拟建系统对象模型的雏形。如果基于纯粹的设计视角，“分析类”相当粗略；但是粗略并不影响它在这个特定阶段的价值，相反，分析活动中急功近利的精细往往会导致事倍功半。

“分析类”直接针对软件的功能需求，因而“分析类”实例的行为来自于对软件功能需求的描述（Use Case 的内容）。换言之，“分析类”用于描述拟建系统中那些较高层次的对象。在使用“分析类”的时候，姑且不必关注某些与应用逻辑不直接相关的细节，特别是那些纯粹的软件技术问题。 在某种意义上，可以认为“分析类”是“技术解耦的”。

6.1.2 概念：“分析类”的类型划分

众多实践经验表明，如果立足于软件功能需求，拟建系统往往在三个维度易于发生变化：第一，拟建系统和外部要素之间交互的边界；第二，拟建系统要记录和维护的信息；第三，拟建系统在运行中的控制逻辑。通常按照这三个变化因素的维度，将“分析类”划分为三种类型，参见图 6-3。这种原则根据变化因素将拟建系统行为的承担者作出了划分。“分析类”的集合是拟建系统静态结构的雏形，某个维度的需求变化对于系统结构的影响有条件被限制在一个相对明确的范围内。简而言之，这种划分方式使得拟建系统的结构对于软件功能需求变化的“反响”具有显著的“高内聚、低耦合”特征。

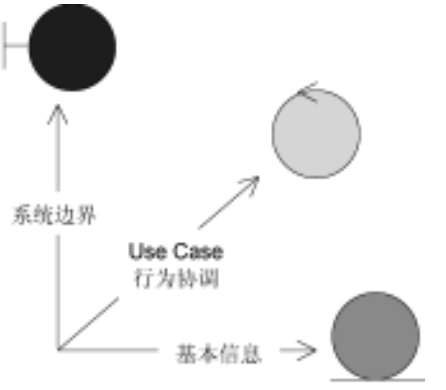


图 6-3　三种“分析类”相互弱耦合

为了在模型中明确地体现上述划分原则，将属于不同维度的“分析类”用类的构造型显式地表达：即边界类《boundary》，控制类《control》和实体类《entity》。

6.1.3 概念：边界类的含义

边界类用于描述拟建系统外部环境与内部运作之间的交互，主要负责内容的翻译和形式的转换，并表达相应的结果。

边界类对拟建系统中依赖于外部环境的部分进行建模，具有良好的隔离作用。概念上，拟建系统的其他部分（即控制类和实体类）将和拟建系统所依赖的外部环境解耦。

边界类主要用于描述三种类型的内容：拟建系统和用户的界面，拟建系统和外部系统的接口以及拟建系统与设备[1]的接口。不同类型边界的建模工作有所不同。例如，与外部系统接口的建模中，主要关注通信协议；用户界面的建模

[1] 设备基本上指具有执行专项功能的节点，不具有或者具有非常有限的计算能力。

中，主要关注用户界面的交互内容，在概念上表述整个界面，而不是具体窗体构件。

边界类的构造型《boundary》在类图中的表述参见图 6-4。

图 6-4　边界类

6.1.4　概念：控制类的含义

通常，控制类用于描述一个 Use Case 所特有的事件流控制行为。控制类相当于协调人，被那些提出具体任务要求的类所共知；它自己通常不处理具体的任务，但它知道那些类有能力完成具体的任务。

控制类将 Use Case 所特有的行为进行封装，具有良好的隔离作用。概念上，拟建系统的其他部分（即边界类和实体类）将与 Use Case 的具体执行逻辑形成松散耦合。换言之，边界类和实体类通常具有跨越多个 Use Case 的通用性。

控制类的构造型《control》在类图中的表述参见图 6-5。

图 6-5　控制类

6.1.5　概念：实体类的含义

实体类用于描述必须存储的信息，同时描述相关的行为。实体类代表拟建系统中的核心信息[1]，是拟建系统中最重要的部分，通常需要长期保留。

鉴于边界类和控制类的双重隔离作用，概念上，实体类和系统的外部环境以及特定 Use Case 的控制逻辑弱度耦合。实体类主要的任务是装载信息，同时也具有行为，但是这部分行为具有“向内收敛”的特征，主要包括那些和实体类自身信息直

[1] 例如，银行系统中，账户和客户是典型的实体类；网络管理系统中，节点和链接是典型的实体类。

接相关的操作。这是实体类能够独立于外部环境以及特定控制流程的必要条件。

实体类为纵深理解系统提供了一个很好的视角：概念上，实体类及其关联展示了拟建系统的逻辑数据结构，和传统意义的“实体－关系”图异曲同工；实践中，它们直接对应拟建系统中体现用户核心价值的那部分内容。

实体类的构造型《entity》在类图中的表述参见图 6-6。

图 6-6　实体类

6.1.6　概念：“分析类”的沿用

“分析类”构成了拟建系统对象模型的雏形，或称之为概念化的设计模型。“分析类”是这种概念模型中的元素，它们将在后续的活动中不断演进：在设计活动中演化为模型中的“设计元素”（“设计类”和“子系统接口”），在实施活动中演化为实施元素（具有物理意义的构件），参见图 6-7。

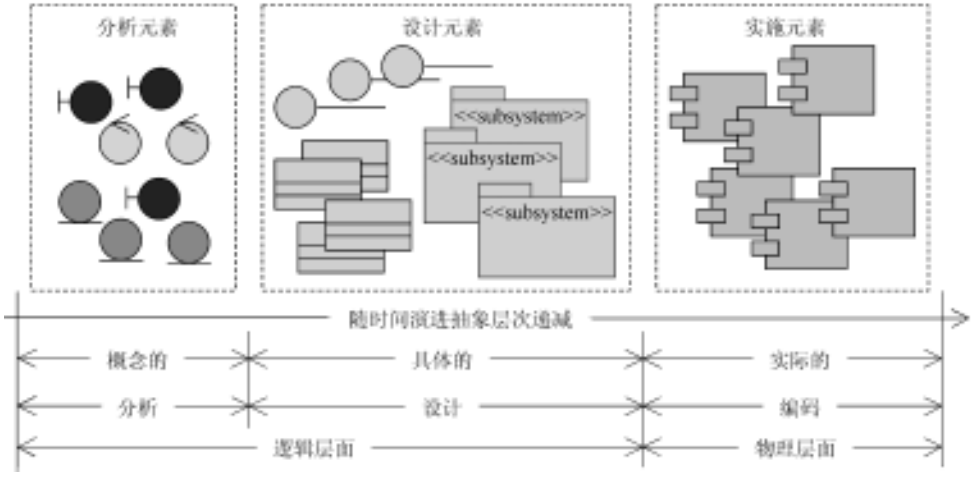


图 6-7　“分析类”的演变

6.1.7　步骤 1：充实 Use Case 内容

通常，Use Case 报告对于系统内部情况的描述比较粗略和概括。为了提取“分

析类”，往往需要更多地了解系统内部的行为。因而，有必要补充说明系统必须做什么才能响应外部的要求。注意，没有必要（也不应该）规定系统的哪些部分完成哪些特定任务。

当然，充实 Use Case 报告内容要结合实际情况，以下是几种典型情形。

- Use Case 报告的内容直接可用。有些时候，Use Case 的撰写人不仅具有深厚的应用领域背景，同时在软件和拟建系统方面也有丰富的知识，这种情况下，Use Case 往往能揭示足够充分的内容。
- 在原有 Use Case 报告的基础上作补充。这是比较多见的情况。如果现有事件流中没有明确定义系统应该执行的行为，则需要作出补充说明。
- 独立于原始 Use Case 描述拟建系统内部行为。有些时候，Use Case 事件流描述中匮乏关于拟建系统内部行为的信息，并且拟建系统内部行为和外部行为的关联非常松散。这种情形下，独立于原始 Use Case 描述拟建系统的内部行为更有利于内容的组织和维护。

6.1.8 步骤 2：提取“分析类”

“提取分析类”就是确定一组备选的、能够执行 Use Case 中行为的“分析类”。“分析类”及其实例的交互将用于满足当前 Use Case 指定的需求。

从文字说明的软件需求过渡到图形描述的设计内容是一个渐进的过程，提取一组备选的“分析类”是这个过程的第一步。

为了获得高内聚、低耦合的设计，使用三种不同的构造型识别和划分候选的“分析类”。边界类、控制类和实体类分别对应相关的获取方法。

- 边界类。通常一个 Actor 和 Use Case 之间的通信关联对应一个边界类。注意，和一个 Use Case 存在交互的 Actor 可能不止一个，参见图 6-8。
- 控制类。通常一个 Use Case 对应一个控制类，参见图 6-8。
- 实体类。参考“全局分析”任务中得到的“关键抽象”集合，在当前 Use Case 的文字描述中挖掘必要的实体信息。

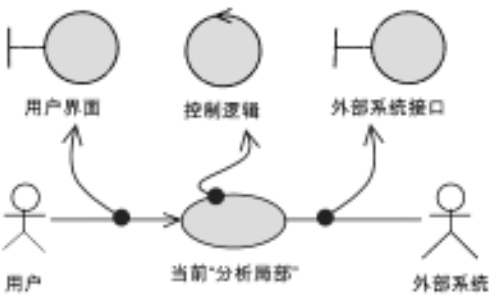


图 6-8 提取边界类和控制类的示意

对于确定下来的“分析类”，给予明确的标识和简要文字说明。

6.1.9 技巧：“分析类”在模型中的位置

在“全局分析”任务中，“关键抽象”位于“一般应用层”的“关键抽象”包；在“局部分析”任务中，“分析类”通常分布在“特定应用层”和“一般应用层”。读者可以参照以下的简单判断原则。

- 控制类通常放在“特定应用层”。
- 从特定“分析局部”中发掘的实体类通常放在“特定应用层”。
- 表述用户界面的边界类放在“特定应用层”。
- 表述外部系统接口的边界类放在“一般应用层”。

实践中，可以灵活掌握。

6.1.10 技巧：边界类的复用

边界类实例的适用范围和生命周期可能超越特定 Use Case 的事件流内容。如果两个 Use Case 同时和一个外部因素（Actor）交互，它们往往可以共用同一边界类，因为它们表现的行为和承担的“责任”有可能存在很多可复用的内容。参见图 6-9。

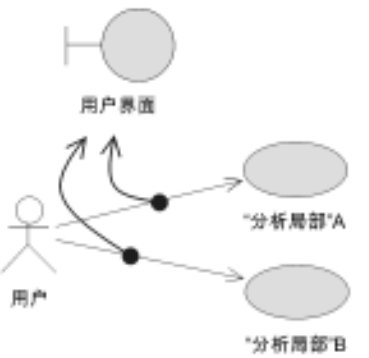


图 6-9 边界类的复用示意

6.1.11 技巧：控制类的变通

控制类实例的适用范围和生命周期通常和特定 Use Case 的事件流内容匹配。一个特定的“Use Case 实现”对应一个控制类是典型的情况，当然也一些变化的情形。

- 如果不同 Use Case 包含的任务之间有比较紧密的联系[1]，某些控制类可以参与多个“ Use Case 实现 ”。换言之，不同的控制类可以参与同一个“ Use Case 实现 ”。注意，这只是一种可能性，目的是重复利用相似部分从而降低整体的复杂性。通常，在明确多个“ Use Case 实现 ”之后，才有可能发现它们之间的相通之处，在此基础上，这种做法才可能带来积极的效果。不应该以此为目标，否则会适得其反。
- “ Use Case 实现 ”未必一定需要控制对象（控制类的实例）。在 Use Case 事件流的逻辑结构非常简单的情况，边界类有可能在相关实体类的协助下实现相应 Use Case 的行为。简言之，如果“ Use Case 实现 ”中的控制逻辑过于简单，那么（承担这类“ 责任 ”的）控制类的必要性明显降低。

6.1.12 技巧：实体类的建议

实体类实例的适用范围和生命周期可能超越特定 Use Case 的事件流内容。实体类通常不是某一特定“ Use Case 实现 ”所专有。以下是针对提取实体类的一些建议。

- 充分利用“ 关键抽象 ”和已经被识别出的“ 分析类 ”，可能是以往迭代中识别出的“ 分析类 ”，或者是当前迭代中从其他“ 分析局部 ”识别出的“ 分析类 ”。
- 有时，和 Use Case 相关的 Actor 会在系统内部有一个实体类与之对应，尤其当该 Actor 的相关信息（属性，操作和关联）对实现拟建系统行为有直接贡献的时候。当然，Actor 和与之相应的实体类在概念上截然不同，通俗讲，就是“ 内外有别 ”。
- 实体类不仅仅包含数据信息，还包含面向数据操作的行为，甚至是相当复杂的行为。
- 如果一个拟建的实体类 A 仅仅被另一个类 B 引用，并且实体类 A 不具有明显的行为特征；那么，可以考虑将实体类 A 作为类 B 的属性；相反，如果需求中的某一实体信息有可能被多个类引用，或者该实体信息具有显著的行为特征，通常将其建模为一个独立的实体类。

6.1.13 技巧：构造型的可选性

“ 分析类 ”的构造型有助于更加清晰地展现分析活动的结果，目的是显式地提高团队对模型内容的管理和理解效率。但是，这并不意味着在“ 局部分析 ”活动中一定要使用“ 分析类 ”的构造型。从逻辑上讲，每个“ 分析类 ”都会归属于边界、控制或实体中的某一个维度。但是，如果从事分析的人员对不同“ 分析类 ”所属维度有清楚的理解，“ 分析类 ”构造型可以被隐含，不作显式的表达。客观上，

[1] 这种联系并不是使用者目标之间的耦合。

顺利地进入设计阶段之后，“分析类”的构造型将不再起实质作用。

6.1.14 示例

以“提交报销申请”的“Use Case 实现”为例，解释提取“分析类”的过程。鉴于示例中使用的 Use Case 报告质量较高，直接从 Use Case 报告的文字描述中抽取“分析类”。对应候选“分析类”的文字被灰化并且加注上角标说明其所属的构造型及其在模型中的英文命名。对于来自“关键抽象”的“分析类”在上角标中也给出了说明。

Use Case“提交报销申请”[控制，SubmitClaim] 报告

简介

员工通过报帐系统填写报销申请，输入相关活动产生的费用，在一次或者多次填写后提交，经过验证之后，以电子邮件的方式通知相应经理批复。

事件流（Flow of Events）

基本事件序列（Basic Flow）

1. 打开报销单

   [员工]：员工[实体，关键抽象，Employee] 选择进入“报销申请”[边界，SubmitClaimForm] 功能。

   [系统]：如果该员工当月报销单[实体，关键抽象，Claim_report] 存在，系统将取出相应信息并展示给员工；如果该员工的当月报销单不存在，则转至 A1 备选事件序列。

2. 添加报销记录

   [员工]：员工要求添加一条包销记录。

   [系统]：系统显示一条空白的包销记录。

3. 填写报销单

   [员工]：员工开始填写报销记录[实体，关键抽象，Claim_record]，每条报销记录包括的信息有：业务活动发生的时间、地点、客户名称（可选）、原因以及费用金额和种类（交通、餐饮、会议、通信和杂项）。

   [系统]：系统显示并记录员工输入的信息。为了让员工方便而准确地输入相关信息，除了客户名称、业务活动原因和金额之外，其他信息域提供相应的下拉式选择列表。

（重复以上针对每一条报销记录的活动，直至所有记录填写完毕。）

4. 验证报销记录单

［员工］：员工填写完毕所有报销记录之后，要求系统验证这些记录的合理性［实体，Valid_Rule］。

［系统］：报销记录的初始状态为“未验证”，每当一条报销记录被验证为合理，系统将该报销记录的状态设置为“已验证”，系统在验证所有报销记录（为“已验证”）之后提示用户可以提交本月的报销单。验证为合理的记录必须满足几种条件：第一，不同种类的费用不超过相应的限额；第二，报销费用的类型要和员工的职能匹配。对于未通过验证的报销记录，转至 A5 备选事件序列。

5. 提交报销单

［员工］：所有报销记录经过验证之后，员工提交当月的报销单。

［系统］：系统保存这张报销单，将报销单的状态设置为“已提交”并记录提交日期，同时这张报销单被设为“只读”。系统要从人事管理数据库［边界，HRDatabase］中获知该员工及其经理（负担该员工当月开销者）的电子邮件地址。如果此时人事管理数据库不可用，转至 A6 备选事件序列。

为了及时通知相关人员，系统将自动生成一份以当前报销单为内容的电子邮件发送到该员工及其经理的信箱中。当邮件成功发送后，员工得到一个确认信息。如果此时邮件系统［边界，MailSystem］未能将邮件及时发出，转至备选事件序列 A7。

备选事件序列组（Alternative Flows）

A1 创建当月报销单

［起始位置］：基本事件序列中，员工进入报销申请程序并准备打开当月报销单。

［触发条件］：系统没有发现和该员工对应的当月报销单。

［具体内容］：系统为该员工创建一张当月报销单。

［返回位置］：基本事件序列中的“打开报销单”步骤。

A2 删除报销记录

［起始位置］：在提交报销单之前任意时间点。

［触发条件］：员工希望删除某一条报销记录。

［具体内容］：系统删除由员工指定的某一条报销记录。

［返回位置］：同“起始位置”。

A3 更新报销记录

［起始位置］：在提交报销单之前任意时间点。

［触发条件］：员工希望更新某一条报销记录。

［具体内容］：系统根据员工重新输入的内容更新相应的一条报销记录。将

该报销记录状态设置为“未验证”。

[返回位置]：同“起始位置”。

A4 保存当月报销单

[起始位置]：该 Use Case 允许员工在事件流中的任意时间点保存当月的报销单。

[触发条件]：员工希望将已经录入的报销记录保存在报帐系统中。

[具体内容]：系统保存该员工的当月报销单，并给出确认信息。员工可以在保存当月报销单之后直接退出系统。

[返回位置]：同“起始位置”。

A5 报销记录不合理

[起始位置]：基本事件序列中，“验证报销单”步骤中对每一条报销记录验证结束之后。

[触发条件]：报销记录不满足某一条适用的准则。有两种情形：第一，某报销记录的金额超出了其对应类型费用的上限，已知有三种：请客户用餐人均超过 300 元，出差时每天住宿费超过 800 元，移动电话在无特殊说明情况下超过 800 元；第二，报销费用的类型和员工所处的部门及职能不匹配，已知的情形是业务部门的员工申请加班补助；

[具体内容]：告知员工不合理的报销记录编号，以及未通过验证的原因。

[返回位置]：基本事件序列中的“填写报销单”步骤，目的是更正有问题的报销记录。

A6 人事管理数据库不可用

[起始位置]：基本事件序列中，“提交报销单”步骤的结尾。

[触发条件]：当报帐系统向人事管理数据库索取信息而该数据库没有正常的响应。

[具体内容]：以对话框形式告知员工“人事管理数据库不可用，报帐单没有提交成功。”

[返回位置]：Use Case 执行结束。

A7 邮件未及时发出

[起始位置]：基本事件序列中，“提交报销单”步骤的结尾，成功地从人事管理数据库获得相关信息之后。

[触发条件]：报帐系统要求发送相关邮件时，邮件系统没有及时的响应。

[具体内容]：系统将以提示信息的方式告知员工，“邮件没有及时发出，但是报销单在系统内已经提交成功，待邮件系统恢复后，相关邮件会自动发出。”

[返回位置]：Use Case 执行结束。

请注意，尽管备选事件序列占据 Use Case 报告文字的很大篇幅，但是，从基本事件序列中通常能够发掘大多数甚至全部的“分析类”。在示例 Use Case 中提取的“分析类”包括以下内容。

- 控制类。“提交报销申请”，即《control》SubmitClaim。
- 边界类。包括以下内容。
  “借款申请”。即《boundary》SubmitClaimForm。
  “人事数据库”。即《boundary》HRDatabase。
  “邮件系统”。即《boundary》MailSystem。
- 实体类。包括以下内容。
  “报销单”。即《entity》Claim_report。
  “报销记录”。即《entity》Claim_record。
  “员工”。即《entity》Employee。
  “验证规则”。即《entity》Valid_Rule。

实体类“验证规则”来自当前的“分析局部”；实体类“报销单”、“报销记录”和“员工”均参照了“关键抽象”的命名；“经理”和“工资户头”等其他“关键抽象”在该“分析局部”中没有出现。

根据“技巧”中的相关原则，上述“分析类”被分别安放在层次构架的“特定应用层”和“一般应用层”，参见图 6-10。

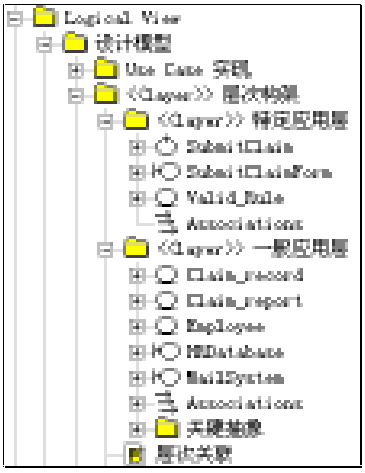


图 6-10 “分析类”在层次构架中的位置

6.2 转述需求场景

“转述需求场景”活动的主要依据是 Use Case 报告中用文字描述的需求场景；该活动的结果是基于面向对象概念、用 UML 交互图（主要是序列图）转述的需求场景。参见图 6-11。

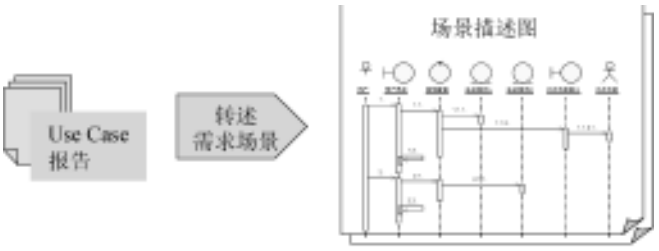


图 6-11 “转述需求场景”活动图示

6.2.1 概念：“消息”与“责任”

“局部分析”的目的是用面向对象的方法转述需求中的应用逻辑。“提取分析类”活动中找出了用于转述需求的分析元素。用面向对象的方法转述 Use Case 中的内容，就是用分布在一组“分析类”中的“责任”分担 Use Case 所要求的行为。

描述“分析类”实例之间的“消息”转递过程就是将这些“责任”指派到“分析类”的过程，参见图 6-12。这个过程是从软件需求过渡到设计内容的关键环节，其中的核心概念是“消息”和“责任”的对应关系。Use Case 的事件序列通常能用一组具有逻辑连续性的、介于“分析类”实例之间的“消息”传递加以表述。“消息”的发出者要求“消息”的接受者通过承担相应的“责任”作为对“消息”发出者的回应。一个“分析类”的实例在事件序列中接受的“消息”集合是该“分析类”应承担“责任”的依据。

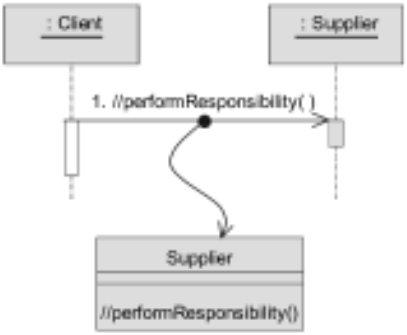


图 6-12 “消息”和“责任”的对应关系

“消息”在概念上具有显著的动态特征，和“分析类”的实例相关联；而“责任”在概念上具有显著的静态特征，和“分析类”的定义相关联。“消息”在客观上是有次序的，但是“责任”并没有次序的概念。一种类型的“责任”往往能够响应多种“消息”。

通俗地讲，“责任”的集合定义了“分析类”所具有的能力，之所以要具备这些能力是为了满足“消息”发出者的要求。“消息”的有序组合本质上表达出软件需求中的应用逻辑，“分析类”承担的“责任”集合本质上将驱动系统的设计，需求和设计在微观层面就这样被面向对象地关联在一起。

鉴于系统内部的要素之间通过消息驱动，整体上，要素之间具有显著的弱耦合特征，这是面向对象带来的益处。

6.2.2 概念：“责任”的沿用

“分析类”的“责任”约定了“分析类”的行为，即“分析类”实例响应“消息”的能力。“分析类”在后续的设计活动中将逐步地演变为具体的“设计元素”；相应地，“分析类”的“责任”也将逐步地演化为“设计元素”的行为，具体讲就是“设计类”的操作和“子系统接口”的行为规约。

6.2.3 概念：序列图中的 Actor 实例

序列图是表述消息传递的直接途径，是最常用的一种交互图。在转述 Use Case 场景的序列图中，通常会出现 Actor 的实例。Actor 的实例表述系统外部的要素，而“分析类”的实例表述的系统内部的要素，二者在概念上“内外有别”。对应一个 Use Case 有一个主导 Actor 作为交互时序的发起者，主导 Actor 的实例通常位于序列图的左侧。同一序列图中有可能出现多个 Actor 的实例，除了主导 Actor 的实例之外，其他（被动）Actor 的实例应尽量位于序列图的右侧。将 Actor 的实例摆放在序列图的两侧有助于明确概念。参见图 6-13。

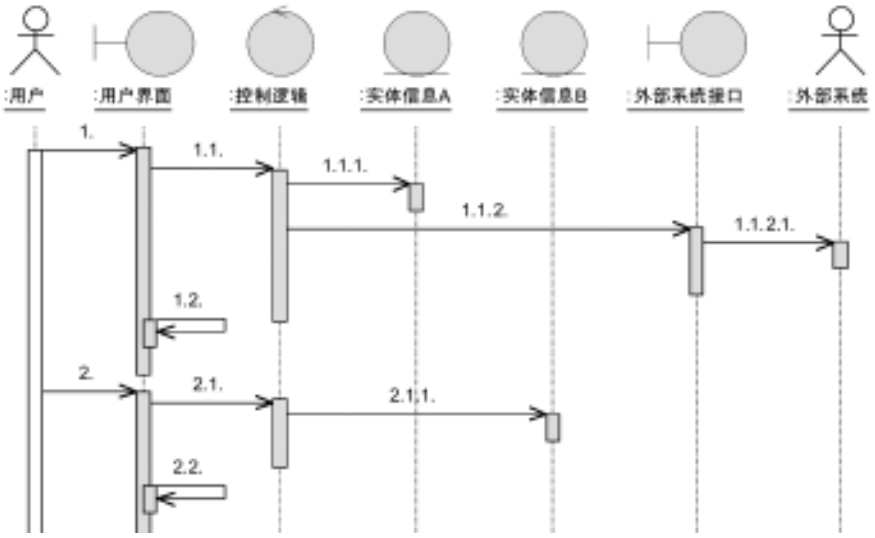


图 6-13　序列图中的 Actor 实例示意

6.2.4 步骤 1：描述 Use Case 事件序列

Use Case 的事件流中包含多个事件序列，其中有一个基本事件序列和若干备选事件序列。简单讲，绘制序列图的工作就是在对象之间用消息传递的方式将事件序列的内容复述出来。

概念上，Use Case 的每一个备选事件序列的内容需要用一张独立的序列图来描述。备选事件序列的序列图和基本事件序列的序列图没有本质的区别，但是，备选事件序列的序列图所涉及的参与对象通常比较少。当某一事件序列的内容比较复杂的时候，如果用一张序列图表述，会显得过于冗长，可以考虑在时间维度上分成两张图。

6.2.5 步骤 2：找出对象传递“消息”的通道

序列图中强调事件序列在时间维度上的轨迹，在轨迹的沿展过程中，我们了解到不同对象之间交互的丰富信息，具体讲就是“消息”。与此同时，我们发掘出消息赖以传递的通道，或者称之为对象之间的“连接”（Link）。笼统地讲[1]，对象之间的“连接”意味着相应的类之间存在关联关系，这是后续设计活动的重要依据。在序列图中，对象之间只有显式的“消息”传递，“连接”的语义表述是隐含的，为了更明确地反映这层语义，可以绘制与序列图对应的协作图，从而显式地表达“连接”。鉴于协作图和序列图在本质内容上等价，通常可以利用建模工具[2]基于序列图生成相应的协作图。

6.2.6 技巧：“未被指派的消息”

逻辑上，能够获得响应的“消息”意味着相应的软件需求已经在设计中得到落实。在实际的建模过程中，无特定“责任”与之对应的“消息”被称为“未被指派的消息”（Unassigned Message）。

“消息”直接转述需求的内容，“责任”则属于设计内容的范畴。设计模型中“未被指派的消息”为需求内容向设计内容的过渡创造了一个“时间差”，这个“时间差”对于成批地引入新增软件需求而后再成批转换成设计内容创造了便利的条件。

6.2.7 技巧：控制类在交互图中的表现特征

控制类的协调职能使得它在序列图和协作图中的表现具有比较明显的特点。按照一般的习惯，在序列图中，将主导 Actor 对应的边界类实例和被动 Actor 对应的边界类实例放在控制类实例的左右两端，实体类实例放在控制类实例的右侧。就结果而言，应该不存在跨越控制类实例生存线的消息。换言之，如果存在跨越

[1] 有些“连接”在后续的设计中将被逐步弱化。
[2] 例如 Rational Software 的著名建模工具 Rose

控制类实例生存线的消息，则有必要引起注意并作出相应的调整。在序列图中，控制类实例的生存线好像是消息的一条“分水岭”。基于前端界面设计者的角度，这种序列图有显著的优点，即前端的设计和后端的处理被很好地隔离。参见 6-13 中“：控制逻辑”对象的生存线。

在对应的协作图中，如果将边界类实例及相应 Actor 实例放在协作图的上方，实体类实例放在协作图的下方，控制类实例放在协作图的中央，那么控制类实例看起来很像一个“贯通南北的交通枢纽”，参见 6-14。

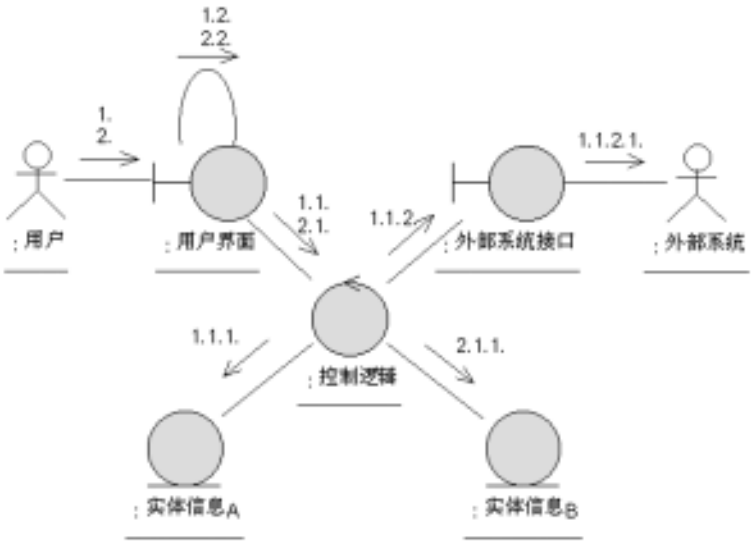


图 6-14　协作图的典型布局

6.2.8　技巧：省略序列图中被动 Actor 的实例

在序列图中，指向被动 Actor 实例的“消息”所对应的“责任”将在系统边界之外被实现（可能是已经存在的功能）。鉴于这个原因，被动 Actor 的“责任”并不能给拟建系统的设计和实施带来实际价值。因而，如果序列图比较复杂，并且被动 Actor 实例无异常行为，可以在序列图中省略。通常，与被动 Actor 对应的边界类能够确保后续设计活动的逻辑完整性。当然，被动 Actor 实例的存在有助于更好地理解上下文关系。读者可以根据实际情况灵活掌握，以便增加序列图的整体易读性。

6.2.9 技巧：“返回消息”

初学者容易犯的一个错误是习惯于在对象之间放置一去一回的两个消息，这通常是不妥的，参见图 6-15。技术上，一个消息的“传递”包含了一去一回[1]的含义。当然，如果序列图比较复杂，为了有助于理解，可以用带箭头的虚线表述“返回消息”，用于强调消息从“发出”到“回应”的跨度，被这种消息指向的对象并不需要为此承担“责任”。参见图 6-16。有关详细说明可参照 UML 的规范[2]。

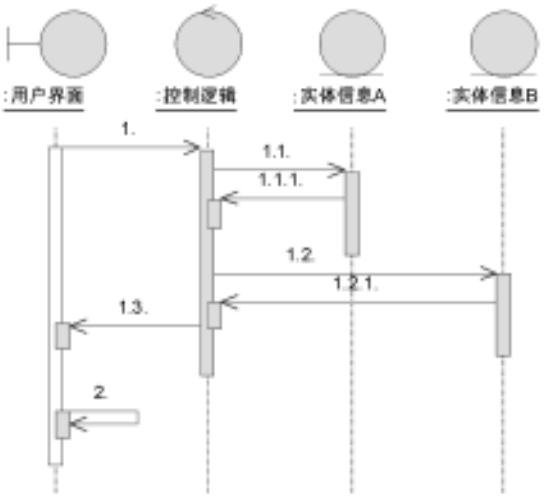


图 6-15　不妥地使用一去一回的消息示例

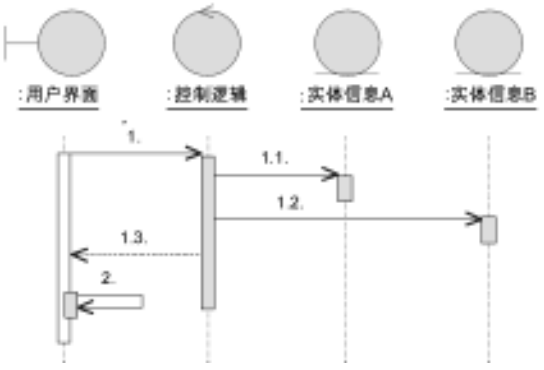


图 6-16　使用“返回消息”的示例

[1] 通过返回值或者参数表。

[2] pp3-106. OMG Unified Modeling Language Specification Version 1.3，June 1999

6.2.10 技巧：在序列图中作文字注释

序列图沿着时间维度详细展示事件序列，是理想的直观表述方式。但是，如果仅依靠隶属于图标的信息，往往并不十分易于理解。建模的一个重要目标是简化直观理解的过程，绘图过程中随时添加必要的文字说明是良好的习惯和实现图文并茂的要领。通常，建议在绘图过程中多作文字注释，以便读者能以更高的效率准确把握图示的内容。序列图中的文字注释通常有两种，参见图 6-17。

在不影响读者理解基本事件序列的前提下，通过文字注释，可以将表述某些局部判断的备选事件序列合并到基本事件序列当中。例如，很多实体信息在第一次使用时需要被创建，可以将创建新实体的“ 消息 ”合并到基本事件序列当中，为这条‘ 消息 ’加注一条文字注释（‘ 如果此时所需对象不存在，则创建一个新的 ’），说明这条“ 消息 ”是可选的。

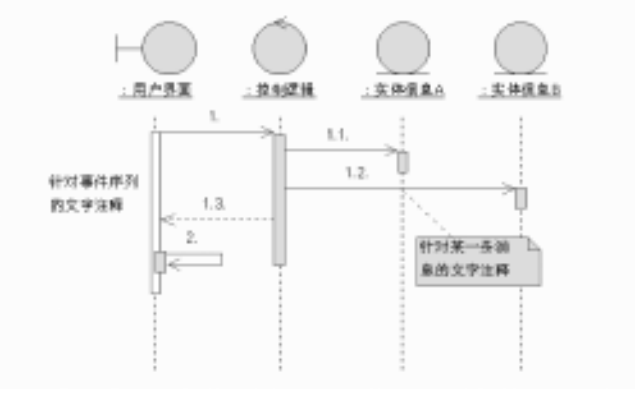


图 6-17　在序列图中作文字注释

6.2.11 技巧：根据需要建立协作图

通常，对应基本事件序列的协作图可以揭示参与“ Use Case 实现 ”的“ 分析类 ”实例之间存在的大部分（甚至是全部）“ 连接 ”。备选事件序列的协作图在这层意义上所作的“ 贡献 ”不明显，通常只能说明在这些“ 连接 ”中传递更多“ 消息 ”。通常情况，只需获取对应基本事件序列的协作图，因为协作图的主要价值在于通过对象之间的“ 连接 ”直观地发掘“ 分析类 ”之间的关联关系。

6.2.12 技巧：交互图的正确性

用交互图（主要是序列图）描述 Use Case 中的场景，通常没有绝对的正确与

错误，不同设计人员得到的结果有可能存在优劣之分。从另外一个角度看，即便是一个富有经验的设计人员用交互图描述 Use Case 中的场景，通常也不容易作到一蹴而就。在从无到有的过程中，某一特定阶段能获得的信息通常是片面和支离破碎的，关键要作出及时和恰当的调整。

6.2.13 示例

图 6-18 展示“提交报销申请”的“Use Case 实现”所包括内容，针对基本事件序列有一个序列图和协作图与之相应，针对每一个备选事件序列有一个序列图与之相应。

“转述需求场景”的过程有较大的灵活性，存在多种选择，为了使“Use Case 实现”中的内容更精简，可以在“提交报销申请”包中建立一个临时存储区（“临时存储区”包），在这里放置一些过渡性或者临时放弃的内容。

图 6-18 “提交报销申请”的“Use Case 实现”内容结构

“转述需求场景”活动很难作到一蹴而就。为了让示例具有更强的实践指导意义，在转述需求场景的示例中，模拟对初步结果作一次调整。在现实生活中，类似的调整应该是多次和连续的。图 6-19 至图 6-27 展示初步得到的交互图系列；图 6-28 至图 6-33 展示经过调整之后的结果。

图 6-19 展现对应于基本事件序列的序列图，以下解释几个要点。

- 由于基本事件序列的内容中涉及到当前局部的所有“分析类”，因而在序列图中出现了所有“分析类”的实例。
- 根据本节“技巧”中提及的原则，为了简化比较复杂的序列图，在基本事件序列对应的序列图中没有摆放被动 Actor 的实例。但是，在两个备选事件序列中，并没有省略被动 Actor 的实例。
- 可以看到，按照“概念”中介绍的一般规律摆放“分析类”的实例，没有一条消息跨跃控制类（SubmitClaim）实例的生存线。

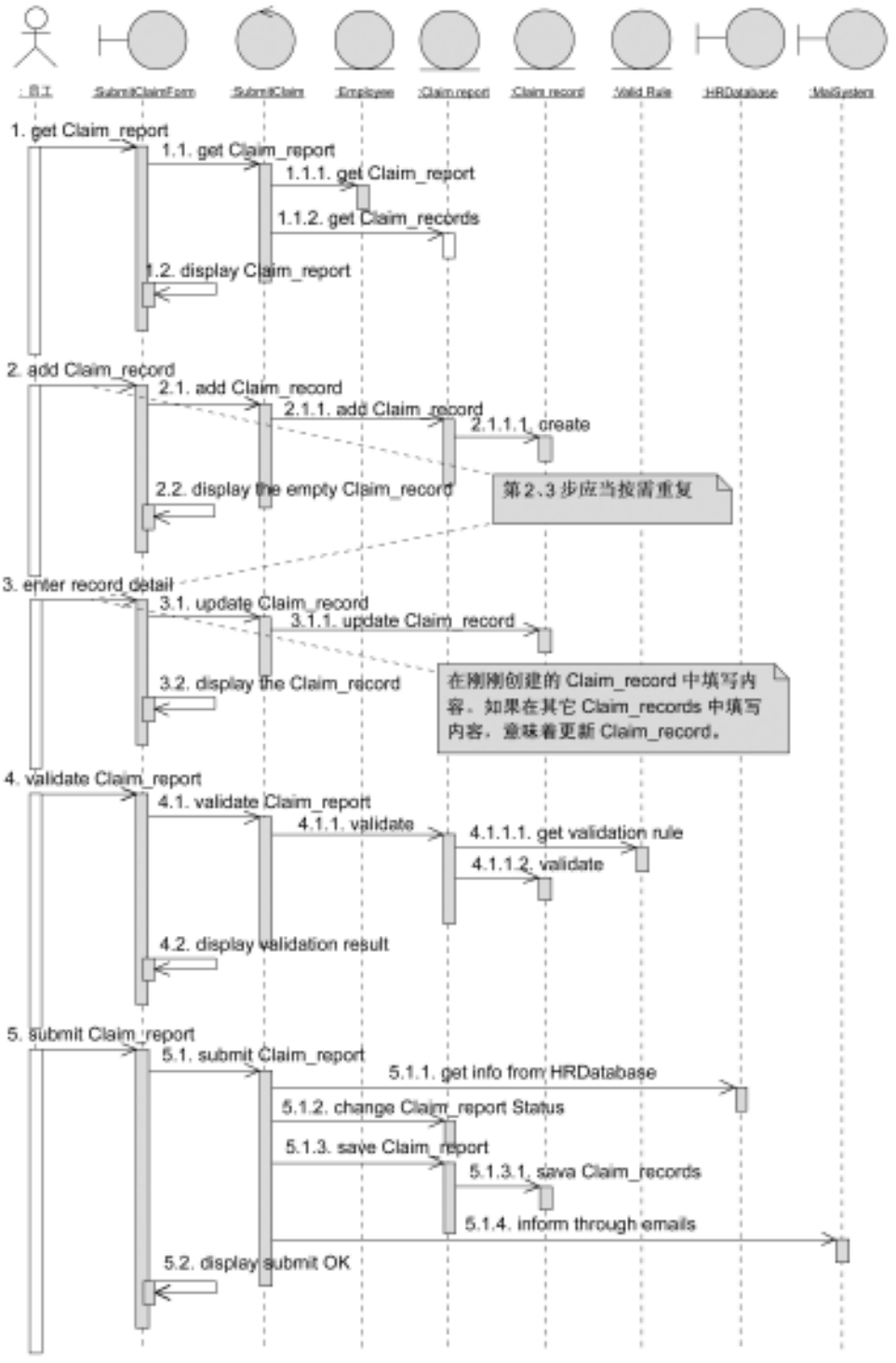


图 6-19　**基本事件序列的序列图示例**

图 6-20 是与图 6-19 序列图等价的协作图。

图 6-21 至图 6-27 是对应备选事件序列的序列图组。在备选事件序列中，绘制出那些作为上下文的消息（图中注释“取自基本事件序列的上下文”），它们来自基本事件序列，参见图 6-21、图 6-25、图 6-26 和图 6-27。图 6-22、图 6-23 和图 6-24 中没有具体的上下文消息，相应备选事件序列可以在任意时间点被执行。在某些备选事件序列当中，复用曾出现在基本事件序列中的消息（图中注释“曾经出现在基本事件序列的消息”），例如图 6-23 和图 6-24。

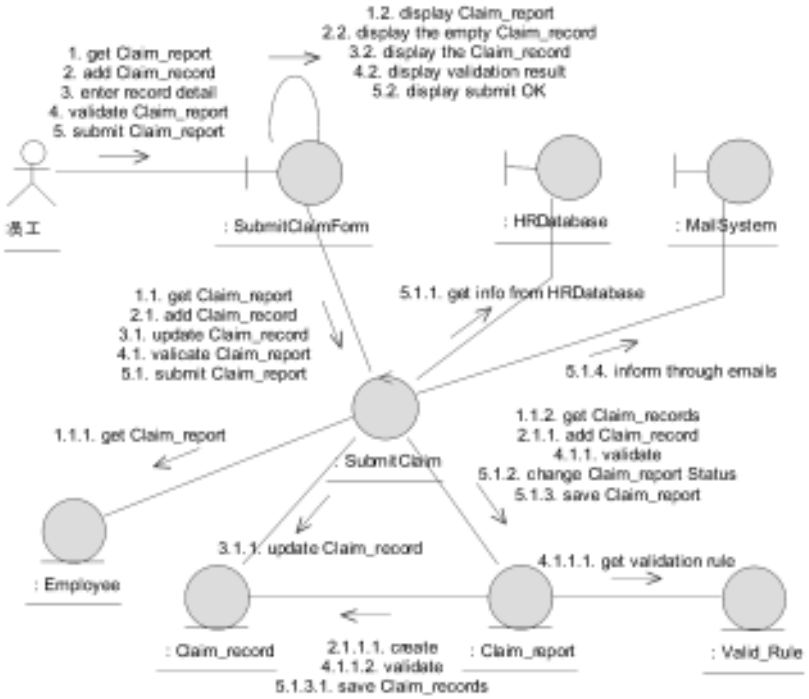


图 6-20　与基本事件序列的序列图相应的协作图

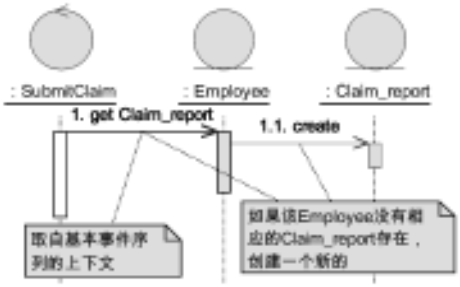


图 6-21　A1 备选事件序列（建立 Claim_report）

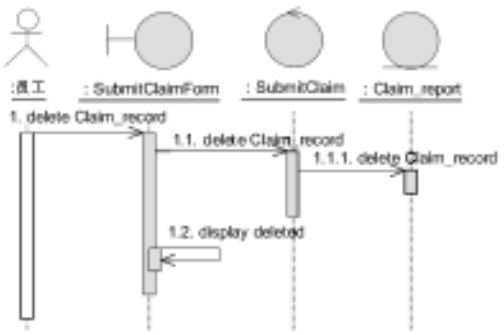


图 6-22　A2 **备选事件序列（删除** Claim_record**）**

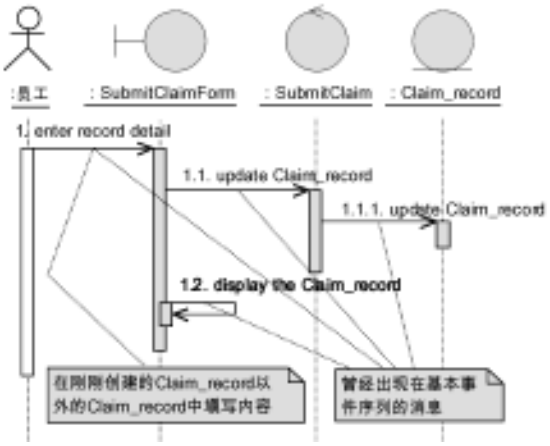


图 6-23　A3 **备选事件序列（更新** Claim_record**）**

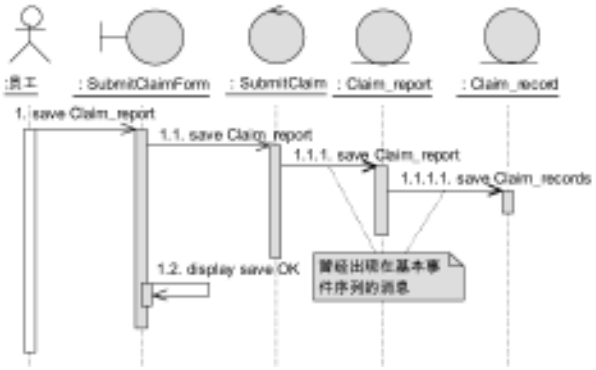


图 6-24　A4 **备选事件序列（保存** Claim_report**）**

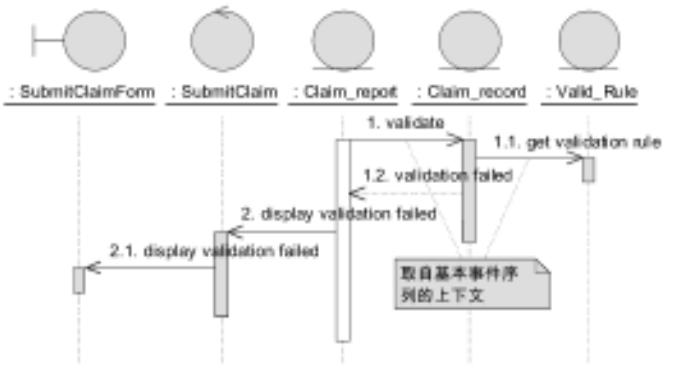


图 6-25　A5 **备选事件序列（**Claim_record **不合理）**

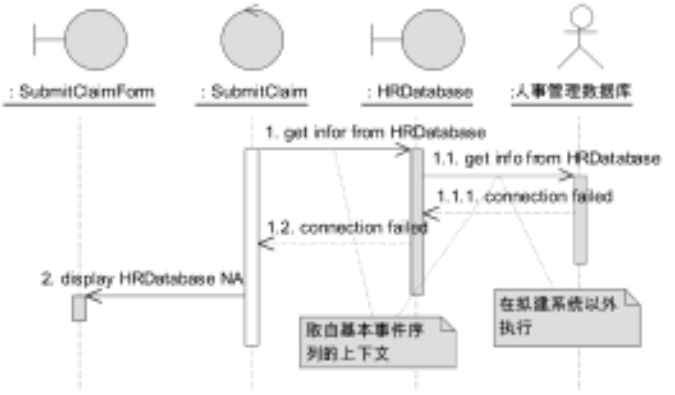


图 6-26　A6 **备选事件序列（**HRDatabase **不可用）**

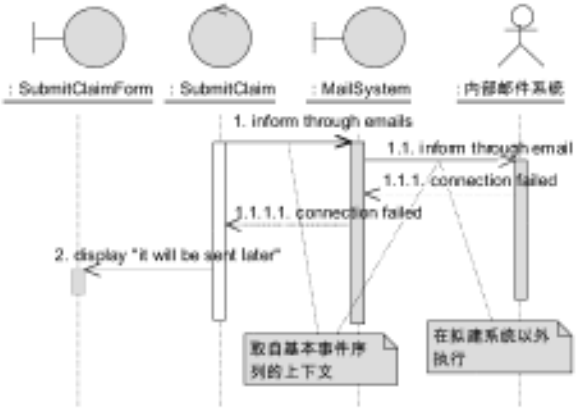


图 6-27　A7 **备选事件序列（**MailSystem **不可用）**

以上是初步得到的交互图组。接下来，将针对一些比较明显的问题和不足进行讨论并作出相应的调整。

A1 备选事件序列（建立 Claim_report）的内容过于简单，可以并入基本事件序列。当然，要给出相应文字说明，例如："如果该 Employee 没有当月 Claim_report 存在，创建一个新的"，参见图 6-28。请注意，在 A1 备选事件序列中的"create"消息比较特殊，用于建立一个新的（留存）对象。事实上，它并不是 Employee 对象和 Claim_report 对象之间真正意义上的消息传递，此时 Claim_report 对象还不存在。仅凭一条"create"消息并不能确认这两个对象之间存在稳定的"连接"（link），参见 6-29。

A3 备选事件序列（更新 Claim_record）并不像 A1 备选事件序列（建立 Claim_report）那么简单。但是，在基本事件序列中可以找到与该事件序列重合的内容，惟一区别是针对不同的报销记录（基本事件序列中针对新创建的报销记录，备选事件序列中针对任意选定的报销记录）。这种差异并不影响反映本质内容的"消息"传递，两种情形对应的后续设计内容没有区别。因而，在给出必要的文字说明之后，A3 备选事件序列（更新 Claim_record）可以"合并"到基本事件序列的序列图当中。事实上，这样做不仅仅少画一张序列图，还有深一层的意义：维护部分内容完全相同的（两个独立的）图非常困难，必须持续确保内容一致，否则将招致诸多麻烦。"合并"之后，那些被"合并"的图将不再出现于"Use Case 实现"的交互图组当中。为便于理解，在基本事件序列的序列图名称上作出相应的说明，参见图 6-30。与 A1 和 A3 备选事件序列对应的序列图被放置到"临时存储区"中，以备不时之需。

初步得到的基本事件序列（图）中与 A3 备选事件序列（图）重合部分的时序关系恰巧值得推敲。在基本事件序列的协作图（参见图 6-20）中，SubmitClaim 实例和 Claim_record 实例之间的"连接"并不是必须存在的消息通道。可以利用 Claim_report 实例作为中转站"接力"传递相关的消息。这样做可以减少一条"连接"，协作图中对象间的耦合将变得相对松散，这是设计师们希望看到的结果。参见图 6-29，SubmitClaim 实例与 Claim_record 实例之间的消息通道被间接的消息通道取代。按照这一思路，对这部分序列图作相应调整，具体情况可以参考图 6-19 和图 6-28 的相关部分。设想一下，如果没有将 A3 备选事件序列（更新 Claim_record）与基本事件序列进行"合并"，上述的调整工作会被重复一次，并且要保证一致。"合并"解消了类似的麻烦。

尽管不是一个规则，但在情况允许时，尽量使序列图中的消息指向右方。如果能作到，相应协作图中，介于对象间的信息流向单一；将来对应的关联关系就有可能是单向的[1]。根据这样的思路，应当对 A5 备选事件序列（Claim_record 不合理）进行调整，调整后的结果参见图 6-31。以 SubmitClaimForm 实例为例：在调整之前，它要接受来自 SubmitClaim 实例的消息"display validation failed"，它要承担的相应"责任"是"对外的"。在调整之后，相应"责任"的执行将基于其

[1] 具有进一步弱化耦合的潜力，参见"细节设计"中的相关内容。

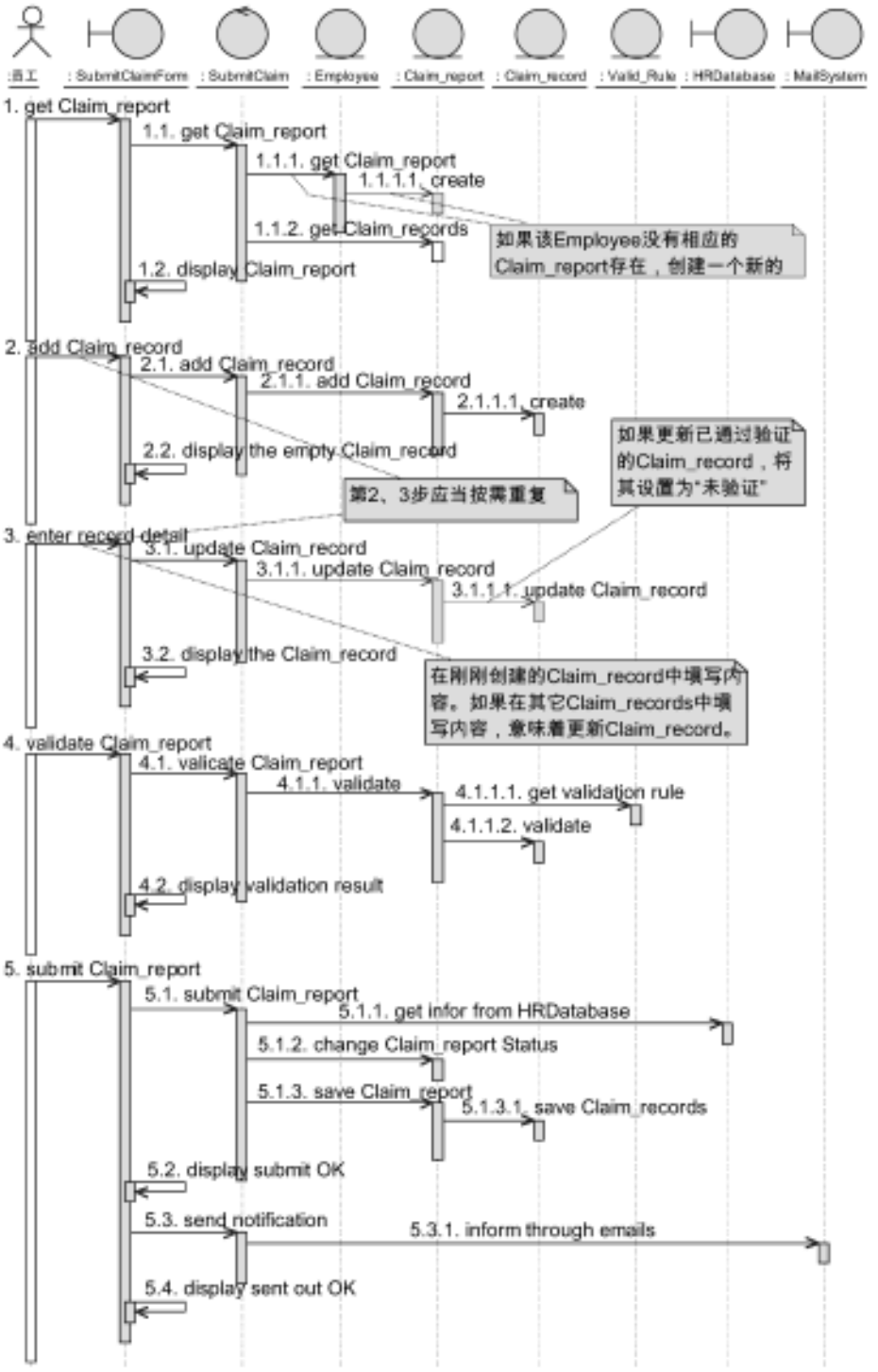


图 6-28　**改进后的基本事件序列（包括** A1 **和** A3**）**

发出的消息“validate Claim_report”所获得的结果，该“责任”是“对内的”。基于设计视角，“责任”从“对外”向“对内”的转换将获得更好的封装与解耦。类似地，对 A6 备选事件序列（HRDatabase 不可用）和 A7 备选事件序列（MailSystem 不可用）进行相应的调整，调整后的结果参见图 6-32 和图 6-33。鉴于图 6-33 中对 A7 备选事件序列（MailSystem 不可用）的改动，基本事件序列的序列图中也要作出相应的调整。调整后，逻辑更加清晰，参见图 6-28。

经过调整，“提交报销申请”“Use Case 实现”的协作关系有所改善。请注意，这里只能针对一些典型问题给出简要说明。针对一个不尽合理的问题，用于修正所复出的代价将随着分析和设计的深入而成倍地增长；因此，在“局部分析”任务中用于调整的努力通常能赢得很高的回报。

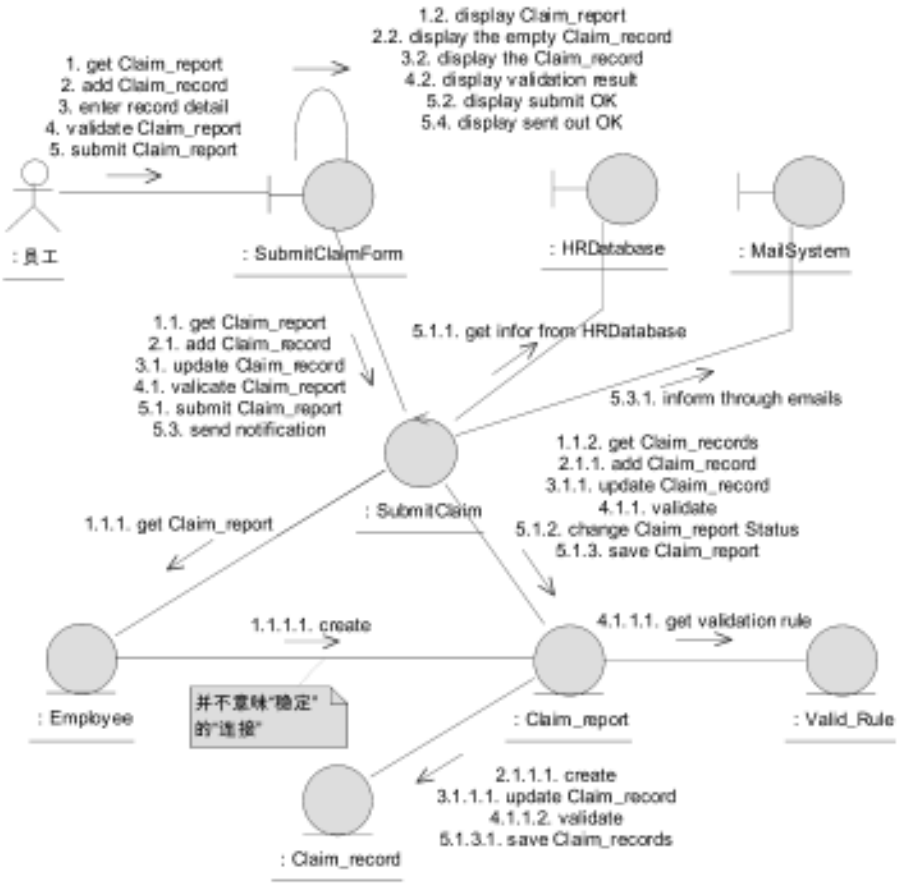


图 6-29　改进后的基本事件序列相应的协作图

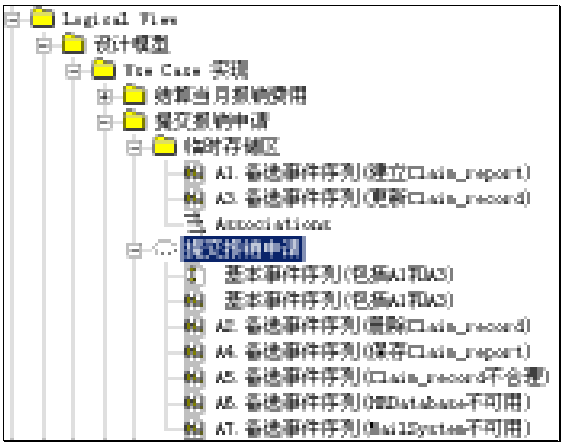


图 6-30　**改进之后的“ 提交报销申请 ”“** Use Case **实现 ” 所包含的图**

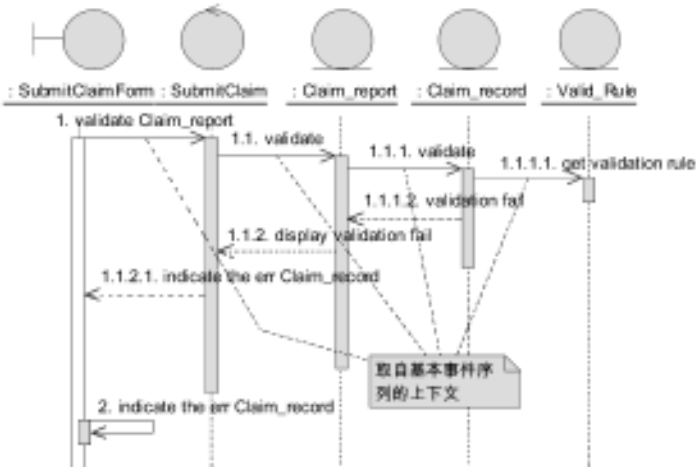


图 6-31　**改进后的** A5 **备选事件序列（**Claim_record **不合理）**

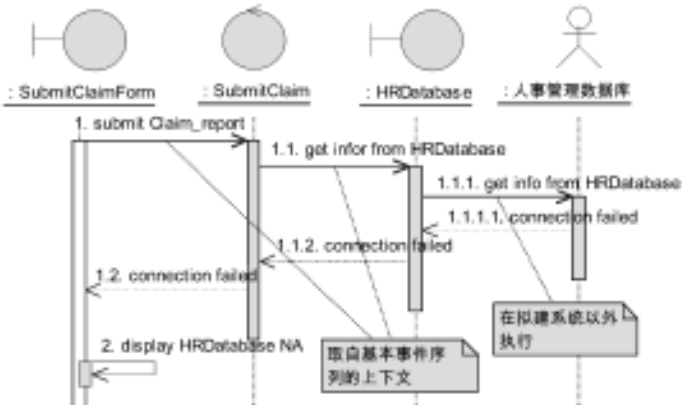


图 6-32　**改进后的** A6 **备选事件序列（**HRDatabase **不可用）**

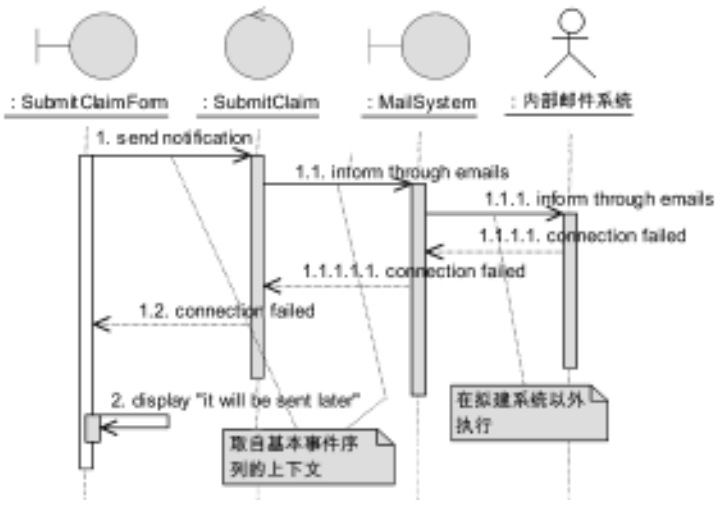


图 6-33　**改进后的** A7 **备选事件序列（**MailSystem **不可用）**

6.3　整理分析类

“ 整理分析类 ” 活动的主要依据是 “ 转述需求场景 ” 活动中得到的一系列交互图；该活动的结果是 “ 参与类图 ”（View of Participating Classes**），它反映参与特定 “** Use Case **实现 ” 的 “ 分析类 ” 之间的关系。参见图** 6-34**。**

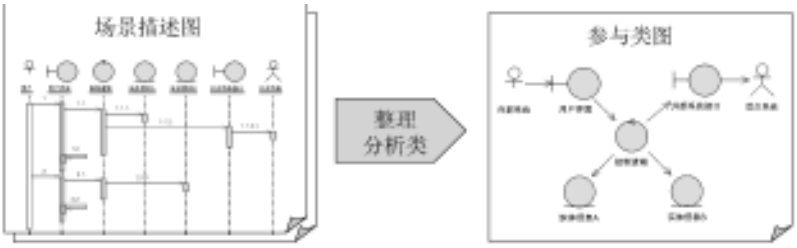


图 6-34　**“ 整理分析类 ” 活动图示**

6.3.1　概念：“ 分析类 ” 的 “ 责任 ” 和关联关系

根据 “ 转述需求场景 ” 中的内容，“ 责任 ” 是响应 “ 消息 ” 的能力。“ 消息 ” 被要求者提出，“ 责任 ” 由响应者承担。“ 消息 ” 作为单元将 Use Case **描述的需求**

场景分解成细小的颗粒。“消息”本身将映射为“分析类”的“责任”，“消息”的传递路径（“连接”）将初步映射成“分析类”之间的关联关系，这是分析到设计在微观层面的映射。“分析类”各自承担的“责任”和“分析类”之间的关联关系构成系统设计方案的雏形。

6.3.2 概念：动态与静态的关系

“分析类”的一条“责任”往往能够响应多条“消息”；类似地，“分析类”之间的关联关系通常能够为多条“消息”提供传递路径。这两点是动态与静态之间的本质关系。形象地讲，动态图是静态图内容的收集器，静态图是动态图的综合器。鉴于上述映射关系，在必要建模工具的支撑下[□]，绘制动态图的过程将伴随静态图内容的充实。

基于动态图内容和静态图内容的多对一映射关系，在分析和设计过程中，应当尽可能多地利用已经存在的“责任”来响应新的“消息”，尽可能利用已经存在的关联关系作为传递“消息”的通道。

6.3.3 概念：“分析类”的属性

承担“责任”的“分析类”应该具备相应的能力，这种能力主要依赖两方面的知识：一方面是“分析类”本身具有的信息，另一方面是“分析类”能够找到的其他“分析类”。如果用技术语言描述，“分析类”本身具有的信息就是“分析类”的属性，“分析类”用于找到其他“分析类”的知识是该“分析类”指向其他“分析类”的关联关系。广义上，关联关系也可以算作“分析类”的属性。

本书讨论的“属性”是狭义的属性，即“分析类”自身具有的简单信息，因而，属性的类型是简单数据类型，例如字符串型、整型、布尔型等。在“局部分析”阶段，“分析类”是粗略的，因而，“分析类”的属性也是相对粗略的，目的是在逻辑上支撑“分析类”所承担的“责任”。

6.3.4 概念：“参与类图”的含义

“责任”和“属性”属于某一个特定的“分析类”。“分析类”之间的关联关系所涉及的范围则不局限于某一个“分析类”，而是介乎多个“分析类”之间。“参与类图”（VOPC，View of Participating Classes）是表述这些关联关系的方式。“参与类图”中包含一组类和这些类之间的关系，这组类的实例参与特定“Use Case 实现”的协作。反映在“参与类图”中的类之间关系，取材于相应“Use Case 实现”的动态交互内容，即交互图组。“参与类图”的主要目的是从“Use Case 实现”中挖掘出参与类间的关系。

□ 客观地讲，如果不利用工具，这种充实过程几乎不具有实用价值，因而通用的面向对象建模工具都能解决这个问题。

6.3.5 步骤 1：确定“分析类”的“责任”

“责任”是“分析类”实例响应消息并完成特定任务的能力，包括为外部（其他对象）提供必要的服务和维护自身的信息。“责任”在后续的设计活动中将演化为“设计类”的一个或多个操作。确定“分析类”的“责任”，主要包括两个动作。

首先，找出“责任”。鉴于“责任”和“消息”的简明对应关系，“转述需求场景”的过程既是用“消息”分解和转述需求的过程，也是找出“责任”的过程。“消息”和“责任”并不是一回事，所谓找出“责任”是根据“消息”的要求定义“责任”，即用“责任”满足相应“消息”所提出的要求。不需要针对每一条“消息”定义一个新的“责任”，很多时候，利用已经存在的“责任”即可满足“消息”的要求。

然后，简要描述“责任”。为了获得简明的图示，“责任”的名称通常比较简短，“责任”的实例将取代“消息”出现在序列图或协作图当中。为了使模型易于理解和沿用，通常建议给“责任”附加简要的文字说明，描述该“责任”可能对应的操作逻辑以及该“责任”被调用之后将返回何种结果。

6.3.6 步骤 2：确定“分析类”间的关联关系

“分析类”之间的关联关系通过“参与类图”直观地表达出来。确定“分析类”之间的关联关系，主要有两个动作。

首先，建立“参与类图”。在特定“Use Case 实现”中添加“参与类图”，将所有“参与”该“Use Case 实现”的“分析类”添加到该类图中。一般情况下，根据“Use Case 实现”中基本事件序列的交互图（协作图），可获得大部分（甚至是全部）的“参与类”。

然后，确定关联关系。协作图中对象之间的“连接”是“参与类图”中相应“分析类”之间关联关系的动态表现形式。概念上，特定“Use Case 实现”的“参与类图”中，“分析类”之间完整的关联关系要根据所有事件序列中对象间的“连接”加以归纳，这是一个典型的动态向静态映射的过程。由于多张协作图对确定“分析类”关联关系的贡献在很多时候是重叠的，实践中，可以从基本事件序列的协作图入手，通过其他事件序列的序列图加以验证和补充。如果基本事件序列的协作图中涵盖了所有参与类的实例，那么“参与类图”在外观上与该协作图相似。注意，“参与类图”中的关联关系应该能在 Use Case 事件流中找到相应根据，而不是来自主观的臆断。另一方面，可以将一些显而易见的事实反映在“参与类图”中，例如明确的聚合关系或组合关系。

6.3.7 步骤 3：确定“分析类”的属性

属性是“分析类”的基本内容，属性的取值使得“分析类”的实例具有必要

的“知识”，从而履行其承担的“责任”。在“局部分析”任务中，确定属性的工作主要围绕实体类展开，包括两个动作。

首先，找出属性。属性的基本来源是 Use Case 的事件流描述。实践中，如果对“分析类”“责任”的简要描述比较明确，属性比较容易获取。“分析类”不仅仅依靠自身“力量”具体承担所有“责任”，有些“责任”的实际含义是“找出”一个更“胜任”的“分析类”承担相关的“责任”。通俗讲，就是将“责任”“转嫁”出去，类似于“代理”的概念。需要挖掘的属性主要用于支撑“分析类”“自身”承担的“责任”。当然，有时“转嫁责任”也要具备某些用于判断状态的信息。广义上，关联关系也是履行“责任”时可以利用的“知识”。这部分知识可以直接从事件序列的交互图中获取；属性则通过“分析类”的“责任”间接地获取。

然后，简要描述属性。属性名称应当是一个简短的名词，说明其保存的信息。分析活动中，属性应该是粗线条的，通常没必要在属性的数据类型和相关细节上耗费过多精力。为了使模型易于理解和沿用，通常建议给含义相对复杂的属性附加必要的上下文说明。

6.3.8 技巧：实体类与属性的差异

类的属性和独立的实体类有所不同。但是，客观地讲，它们之间的界限并不非常清晰。在以下两种情况，通常用独立的实体类为特定客体（信息）建模。

- 客体具有比较复杂的自身行为。
- 客体具有独立标识（Identification），有可能被多类对象共享或者传递。

相反地，在以下情形，通常用类的属性为特定客体建模。

- 客体除了非常简单的取／赋值操作（get，set 等），不具备更多其他行为。
- 客体不需要独立标识，仅供一类对象使用。

有些时候，建模形式的选择不容易一步到位。例如，在分析和设计的演进过程中，如果发现之前定义的实体类 A 几乎没有行为并且仅仅被另外一个类 B 使用，那么可以将类 A 转换成为类 B 的一项属性 a。相反地，如果类 X 的某个属性 y 越来越显现出复杂的行为特征，则可以考虑将该属性建模为一个独立的实体类 Y。

在示例中将“Valid_rule”作为独立实体类的原因，就是考虑到它可能在多个 Use Case 的事件流中被使用，例如“调整策略”和“提交报销申请”等。因为要参加多组对象的协作，“Valid_rule”对应的内容应该具有独立的标识。

6.3.9 技巧：不同“分析类”的同名“责任”

在实践中，不同的“分析类”通常会包含相同名称的“责任”。因为承担“责任”的“分析类”有两种选择，一是自己负责把问题解决，二是将问题转交出去。作为发送相应消息的对象并不关心“责任者”如何解决问题，这是封装带来的益处。

但是，不要让两个“分析类”（比如类 A 和类 B）拥有相同（或很相似）的能够“自行承担”的“责任”，否则会带来不利影响。提“要求”的对象所发出的

消息可以指向任一个类（类 A 或类 B）的实例；有可能造成随机的选择。在后续设计活动中，如果对两个类相似“责任”的调整不能保持一致，将招致不必要的麻烦。简言之，一种“责任”由一个“分析类”“自行承担”足矣。

6.3.10 技巧：复用已有的“责任”、属性和关联关系

在“局部分析”任务中，应当充分利用面向对象基本原理提供的天然优势，尽可能地复用已经存在的“责任”、属性和关联关系。这样做会带来两个方面的益处：一方面，减少重复性建设，使得模型的内容更加简单；更重要的是，避免相似内容的存在，提高对变化的后续响应效率，降低维护成本。

广泛“复用”“分析类”的内容，不仅限于特定的“局部分析”范围，应当尽可能复用已经出现于其他“局部”的“分析类”及其内容。注意，“局部分析”中所谓的“局部”是针对软件需求而言，是应用逻辑意义上的局部；对于构成拟建系统设计内容雏形的“分析类”而言，其适用范围是全局性的。

很多时候，当不同的“局部分析”各自取得一定进展之后，局部之间有一些“分析类”在功能上非常近似或者可能定义了几乎相同的属性。此时，需要通过必要的“合并”手段“强行”实现“复用”。表面上看，似乎多了一些负担，实际上，为后续设计工作扫除了很多不必要的障碍。广义上，“复用”的原则不仅在分析和设计活动中具有重要价值，在整个开发过程中都能带来有益的结果。面向对象技术从原理上为“复用”提供了坚实的支撑。

6.3.11 示例

确定“分析类”的“责任”

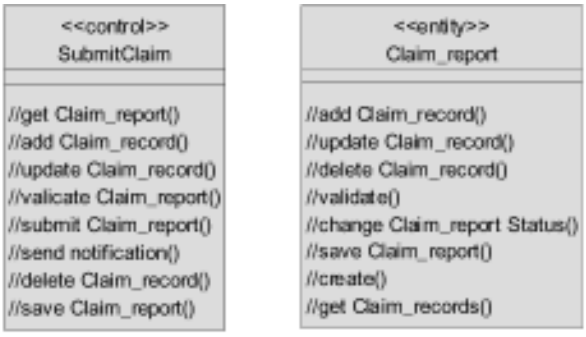


图 6-35 “分析类”承担的“责任”示例

首先，将交互图（主要是序列图）中的“消息”映射成“分析类”的“责任”。在示例中，“责任”的名称大多沿用“消息”的名称。这一映射过程的结果在交互

图中几乎显现不出差异[1]。“责任”还不是类的操作，故而习惯上用“//”作为前缀[2]。此处，以 SubmitClaim 和 Claim_report 为例展示“分析类”承担的“责任”，参见图 6-35。

确定“分析类”之间的关联关系

图 6-36　“参与类图”在模型中的位置

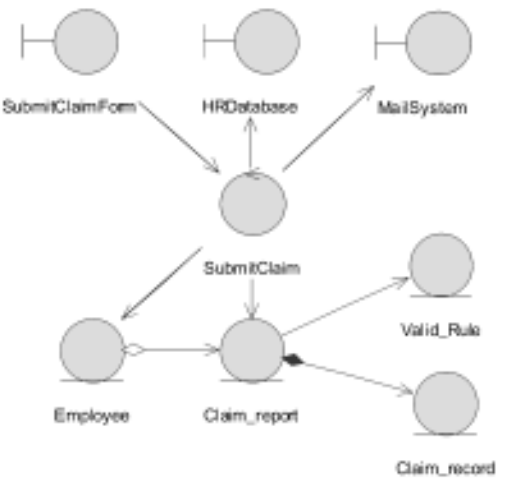


图 6-37　参与类图（简略）

表现“分析类”之间关联关系的主要手段是绘制“参与类图”。逻辑上，每个“Use Case 实现”对应一张完整的“参与类图”。实践中，可以绘制多张“参与类

[1] 通常“责任”以“//责任名称（[参数]，…）”作为表现形式上的特征。

[2] 在一些程序语言中“//”是用作注释行的标记，即便将来有了实际的操作，此时建立的责任可以一直保留下去，作为一种说明性的文字。

图”，目的是让每张图具有各自的“直观化”（可视化）重点。示例的“Use Case 实现”中，建立了“参与类图”的两种不同表现形式，参见图 6-36。

第一，“参与类图（简略）”，用图标表述“分析类”，突出“分析类”之间的关联关系，参见图 6-37。第二，“参与类图（详细）”，用标准方式表述“分析类”，在描述“分析类”之间关联关系的同时，展现每个类的“责任”，参见图 6-38；当“分析类”的属性被识别出来之后，“分析类”的内容将进一步被丰富。

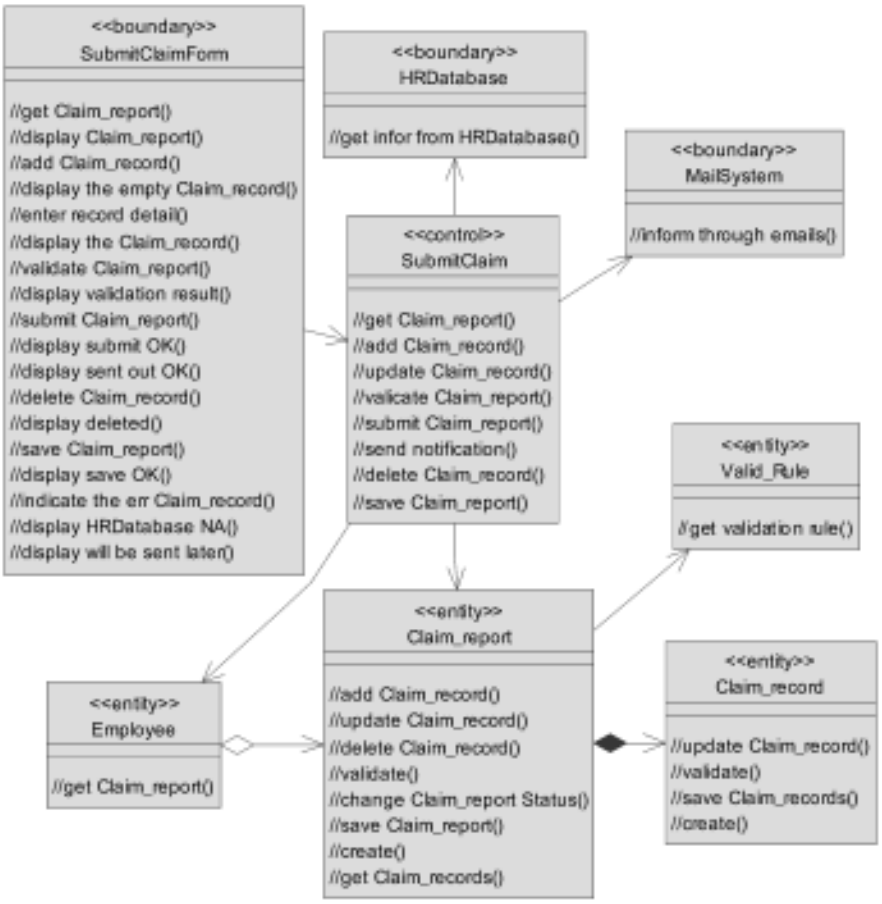


图 6-38　参与类图（详细）

Claim_report 和 Claim_record 之间的组合关系的依据是“全局分析”的结果。注意，Employee 实例到 Claim_report 实例之间用于传送“create”消息的通道在相应的“参与类图”中并不意味存在必然的关联关系。不过，在“全局分析”任务“识别关键抽象”活动中已经明确地给出 Employee 和 Claim_report 之间的聚合关系。

为了强调建模的循序渐进过程，不妨考察“转述需求场景”示例中所作调整对“整理分析类”活动的影响。

基本事件序列第 3 步的时序逻辑在调整前后存在两种情形。

- 情形 A。SubmitClaim 实例直接向 Claim_record 实例发送消息，参见图 6-19。相应的协作图参见图 6-20。
- 情形 B。SubmitClaim 实例通过 Claim_report 实例间接和 Claim_record 实例产生通信，参见图 6-28。相应的协作图参见图 6-29，介于控制类 SubmitClaim 实例和 Claim_record 实例间的“连接”被取消。

情形 B 对应的“参与类图”（图 6-37）比情形 A 对应的“参与类图”（图 6-39）少一个介于 SubmitClaim 和 Claim_record 之间的关联关系。这就是“转述需求场景”活动中调整在“整理分析类”活动中的反映。本质上，这个调整过程利用已经（或必然）存在的两个“连接”（介于 SubmitClaim 实例和 Claim_report 实例之间的“连接”以及介于 Claim_report 实例和 Claim_record 实例之间的“连接”）实现了介于 SubmitClaim 实例和 Claim_record 实例之间的“连接”。换言之，从 SubmitClaim 实例到 Claim_record 实例的消息可以通过已有的通道传递，这样可以省去建立一条新的消息通道。反映在静态结构方面的优势是“参与类图”中少了一个关联关系，耦合被减弱。

如果当初不作调整，相应的“参与类图”将具有图 6-39 的形式。调整前后的差异在“参与类图”中还有另外两个积极的反映：第一，SubmitClaim 和 Claim_report 之间的双向关联关系变成了单向的关联关系；第二，SubmitClaimForm 和 SubmitClaim 之间的双向关联关系变成了单向的关联关系。

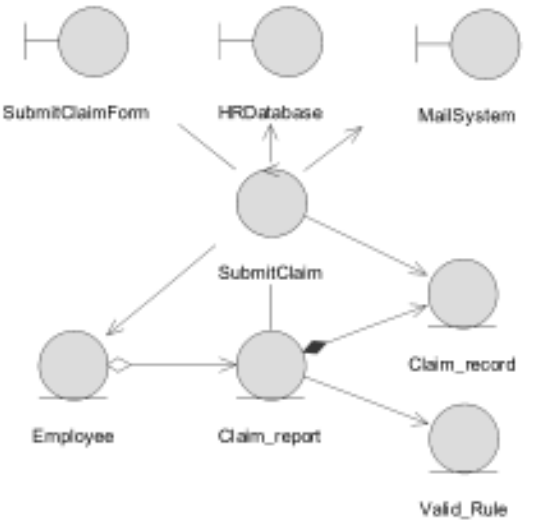


图 6-39 “参与类图”（没有优化场景的描述）

确定“分析类”的属性

接下来确定“分析类”的属性，主要依据是“分析类”“自行承担”的“责任”。以实体类 Claim_report 为例，根据它在多个事件序列中所承担的“责任”（参见图 6-40），对所需的“知识”作一简要的分析。请注意，Claim_report 指向其他“分析类”的关联关系也是可利用的“知识”。当前活动的重点是挖掘关联关系之外的必备“知识”，即“分析类”的属性。

- “责任”“//add Claim_record（）”。创建一个新的 Claim_record，需要 Claim_report 的标识。
- “责任”“//update Claim_record（）”。更新一个 Claim_record，需要当前 Claim_report 是否被提交的状态，以及 Claim_report 指向 Claim_record 的关联关系。
- “责任”“//validate（）”。验证一个 Claim_record，需要当前 Claim_report 所对应员工的部门职能特征，当前 Claim_report 是否被验证的标识（等价于当前 Claim_report 中现有 Claim_record 是否全部被验证），以及 Claim_report 指向 Claim_record 的关联关系。
- “责任”“//change Claim_report Status（）”。在 Claim_report 被提交的时候更改其状态，需要当前 Claim_report 是否被提交的状态，同时需要记录提交的时间和总金额。
- “责任”“//save Claim_report（）”。保存当前 Claim_report，仅需要 Claim_report 和 Claim_record 之间的关联关系。
- “责任”“//delete Claim_record（）”。删除指定的 Claim_record，仅需要 Claim_report 和 Claim_record 之间的关联关系。

根据现有信息，实体类 Claim_report 至少要具备几项属性，参见图 6-40。

- 属性 id。Claim_report 的标识。
- 属性 ownertype。Claim_report 对应的员工的部门职能属性。
- 属性 status。Claim_report 的状态，根据业务常识，该状态信息在不同的阶段可用于标识 Claim_report 是否被验证以及是否被提交。
- 属性 sum。Claim_report 被提交时的总金额。
- 属性 date。Claim_report 被提交时的日期。

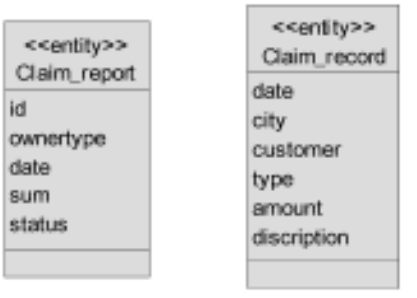


图 6-40　实体类 Claim_report 和 Claim_record 的属性

类似地，可以得到其他“分析类”的必要属性。

针对示例，表 6-1 给出了局部分析任务中积累的设计模型内容。

表 6-1　局部分任务中积累的设计模型内容汇总

任务	活　　动	设计模型内容		
		“ Use Case 实现 ”	层次构架	“ 构架机制 ”
局部分析	提取分析类		图 6-10	
	转述需求场景	图 6-18～图 6-32	图 6-35	
	整理分析类	图 6-36～图 6-38	图 6-40	

第 7 章　全 局 设 计

“ 全局设计 ” 任务将在拟建系统的全局的范围内，以分析活动的结果为出发点，将现有的“ 分析类 ”映射成模型中的“ 设计元素 ”，明确适用于拟建系统的“ 设计机制 ”，调整内容逐渐充实的拟建系统 “ 构架 ”。

在 “ 全局设计 ” 任务中，有不同侧重的三项活动，参见图 7-1。

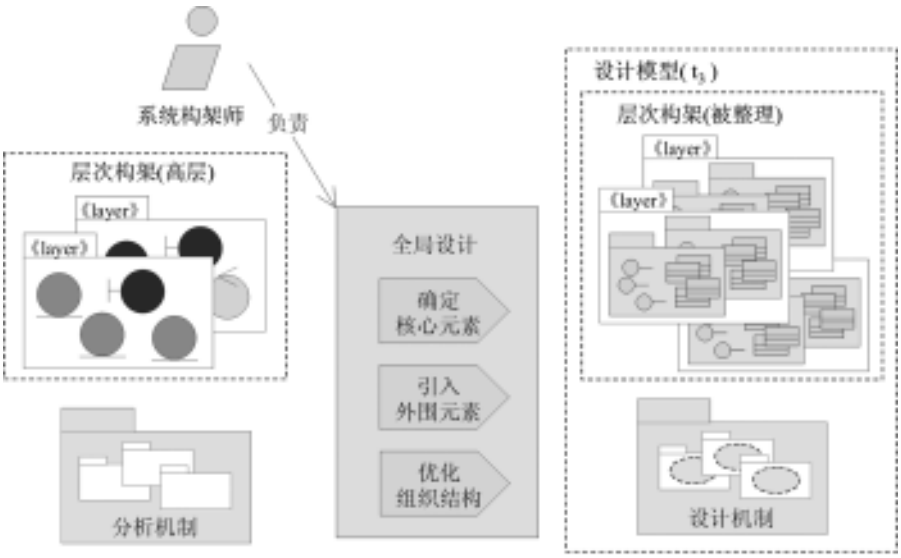


图 7-1　“ 全局设计 ” 任务的责任人 — 依据 — 活动 — 结果

- 确定核心元素。在系统构架的中高层，以 “ 分析类 ” 为出发点，确定相应的 “ 核心设计元素 ”，具体讲就是 “ 设计类 ” 和 “ 子系统接口 ”。
- 引入外围元素。在系统构架的中低层，以 “ 分析机制 ” 为出发点，确定能够满足相关 “ 分析类 ” 要求的 “ 设计机制 ”；根据 “ 设计机制 ” 所依赖的技术实施手段，将那些能够提供基本服务的构件所对应的逻辑内容引入设计模型。
- 优化组织结构。按照高内聚、低耦合的基本原则，整理逐渐充实起来的层次构架内容。

“ 全局设计 ” 任务的总体思路是从上向下充实层次构架中的内容，然后做贯穿层次的调整。

7.1 确定核心元素

“确定核心元素”活动的主要依据是多个“局部分析”活动中得到“分析类”集合；该活动的结果是与“分析类”相对应的“设计类”或“子系统接口”。参见图 7-2。

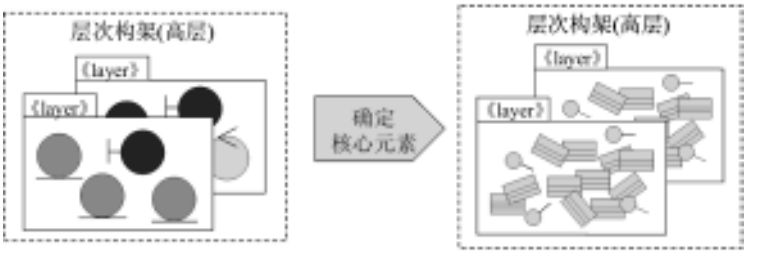


图 7-2 “确定核心元素”活动图示

7.1.1 概念：“核心设计元素”的含义

概念上，“设计元素”是能够直接用于指导实施（编码）的模型要素。在本书范围内，“核心设计元素”指由“分析类”映射而来的“设计类”或“子系统接口”。

“设计类”和“子系统接口”将“分析类”的“责任”归入不同的设计单元，以便对各个单元进行相对独立的后续设计。“设计类”承担那些对应于独立构件的“分析类”之“责任”集合；“子系统接口”定义了那些对应于复合构件（即子系统）的“分析类”之“责任”集合。子系统进一步由“设计类”和其他的子系统组成，“子系统接口”将拟建系统中相对复杂的一部分任务封装为完整的功能单元□。

在分析活动中，“分析类”主要针对“要做什么事情”（即所谓 What），目的是满足软件需求中的应用逻辑；在设计活动中，“设计元素”主要针对“怎么做才切合实际”（即所谓 How），目的是适应软件需求中的非功能性约束。

7.1.2 概念：“子系统接口”的定义

子系统是对一组承担“责任”的 “设计元素”的统称，“子系统接口”明确定义这些“责任”，但不约束“责任”的承担者及承担方式。广义上，也可以为“设计类”定义相应的接口，但“设计类”的公开操作本质上就是这个“设计类”的接口，因而单独说明接口的必要性不大。

通过明确的定义，“子系统接口”将其使用方法和实现方式彻底解耦。借助“子系统接口”的定义，可以将拟建系统中某一部分的复杂行为和其他的设计内容进行隔离。这样，在降低耦合程度的同时提升了整体设计模型的延展性，增加了组织设计活动的灵活性，尤其便于灵活安排时间。

“子系统接口”的定义为设计活动带来很多益处，是高度可复用的资源，对拟建系统构架具有广泛的影响。“子系统接口”的定义通常要充分考虑和复用已经存在的设计内容，在完整的基础上力求简单。

7.1.3 步骤 1：映射“分析类”到“设计元素”

如果一个“分析类”比较简单，代表单一的逻辑抽象，可直接映射为“设计类”。通常，主导 Actor 对应的边界类[1]、控制类和一般的实体类可以被映射成“设计类”。一个“分析类”可以映射为一个“设计类”或者多个“设计类”的简单组。这类设计决定需要考虑特定的技术实现环境，目的是为后续的设计活动带来直接的益处。

如果“分析类”的“责任”比较复杂，其行为很难由单个“设计类”或者“设计类的简单组合”承担，应该将其映射为“子系统接口”。后续设计活动中，特定的子系统将在相应“子系统接口”的封装下实现相应行为。从子系统的外部看，它和一个“设计类”在概念上没有严格的差异。通常，被动 Actor 对应的边界类被映射为“子系统接口[2]”。与此同时，显式表述拟建的子系统对其接口的实现关系。

7.1.4 步骤 2：定义“子系统接口”

首先，命名和简述“子系统接口”。在相应“分析类”的名称前加上前缀“I_”，将“分析类”的构造型标记为《Interface》，给出简明的文字描述，反映它在系统中的作用。

接着，描述拟建子系统的行为。描述拟建子系统的行为是定义“子系统接口”的核心任务，具体讲就是准确定义操作集合，取代笼统的“责任”。在表现形式上“责任”通常用“//”作为前缀，而操作不需要这样的前缀。操作的命名不能像“责任”的命名那么随意（接近自然语言），不仅要反映出该操作的结果，而且必须考虑到程序设计的通用规则。例如，不宜在操作名称中出现空格。为了实现“使用”和“实现”的解耦，“子系统接口”中对操作的描述应该具体化，至少应当说明以下内容，其中前三点构成了操作的标识（即 Operation

[1] 这种单元既可用作逻辑模型中的设计单元，又可用作控制与配置管理的单元。

[2] 实践中，主导 Actor 对应的边界类所表达的内容可以通过基于用户体验的方法进行建模，然后采用可视化的界面工具直接构造相应的实施内容。

[3] 在复杂系统的设计中，严格讲，这里得到的仅仅是“候选的”“子系统接口”，真正意义上的接口通常还要经过一些横向的综合考虑过程之后才能够确定下来。

Signature）。

- 操作的名称。
- 操作的返回值含义及类型。
- 操作使用的参数名称及类型。
- 操作应该做什么（文字描述，包括关键的算法）。

然后，确定“子系统接口”对其他“设计元素”的依赖关系。总的来讲，在“子系统接口”定义中，操作的返回值与参数的类型有三种：一种是简单数据类型（Primitive Type）、一个是“设计类”或者是另一个“子系统接口”。在当前“子系统接口”与所有出现在其操作中的“设计类”和“子系统接口[1]”之间，分别建立依赖关系，参见表 7-3。这些依赖关系为系统构架师了解和调整设计模型内部的耦合关系提供了关键信息。

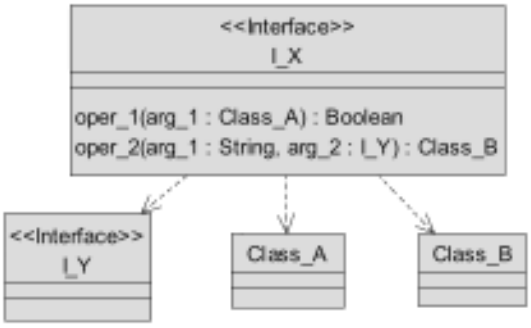


图 7-3 “子系统接口”对其他“设计元素”的依赖关系

7.1.5 技巧：“子系统接口”的动态表述

在设计模型中，经常使用类的构造型《subsystem proxy》，其逻辑上的含义是：如果使用一个特定子系统 S 提供的服务，都可以通过一个属于 S 的类 A 的公开操作而实现，那么类 A 就是子系统 S 的《subsystem proxy》。请注意，“子系统接口”只有逻辑上的意义，没有物理意义，它说明子系统的使用者和子系统服务的提供者之间共同认可的约定（合同、契约），是一种描述。《subsystem proxy》类是某一特定子系统的组成部分，具有实际的物理意义，它的公开操作集合覆盖了相关“子系统接口”中定义的所有操作。

在静态描述中，子系统或者相应《subsystem proxy》对“子系统接口”的实现关系是等价的，参见图 7-4。用《subsystem proxy》实现“子系统接口”的价值主要体现在动态描述（交互图）中，因为《subsystem proxy》的实例具有实际物

[1] 在后续的设计活动中，它将被属于特定子系统的《subsystem proxy》所取代。

理意义。严格地讲，“子系统接口”被实现之前在动态中没有相应的实际内容。在不影响逻辑的前提下，为了方便，可以将其保留[1]在序列图中，但含义是模糊的，仅作为一个拟建子系统的“替代符号”而已。由于没有实际的物理含义，这种“替代符号”不可能发送消息给其他对象。

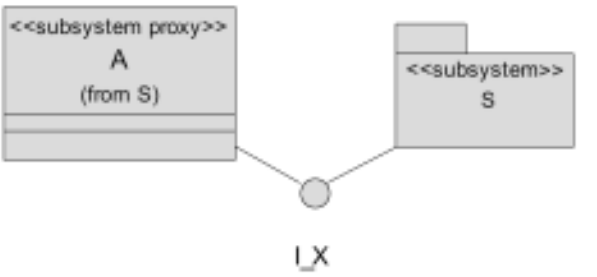


图 7-4　子系统或相应《subsystem proxy》实现“子系统接口”

7.1.6　技巧：“子系统接口”的辅助说明

“子系统接口”定义同时服务于两种类型的设计人员：一类设计人员设计那些需要利用“子系统接口”中所声明服务内容的“设计元素”；另一类设计人员将具体实现“子系统接口”中所声明的服务内容。所谓的“服务内容”就是“子系统接口”中定义的操作集合。

为了帮助“子系统接口”定义的使用者更好地理解接口所声明的服务内容，在基本信息基础上，可以添加一些辅助的描述。例如，通过序列图来说明操作的使用方式和执行顺序；通过状态图说明实现接口的“设计元素”可能处于的外部可观测状态，甚至可以包括那些用来测试这些“设计元素”行为的具体做法。

7.1.7　技巧：“子系统接口”的融合

在比较复杂的设计模型中，复杂的“分析类”未必和“子系统接口”一一对应。通常，先以这些“分析类”作为出发点，定义出一组候选的“子系统接口”集合；接着，从候选的“子系统接口”集合中，对照并发掘类似的名称和操作；然后，抽取并融合那些共通的内容组成最终的“子系统接口”集合。图 7-5 给出一个很简单的示例，融合的结果和依据之间存在“实现”的逻辑关系。本质上体现了“复用”的思想。

[1]由于“子系统接口”是由“分析类”演变而来，“分析类”的实例在交互图中的位置被“子系统接口”接替。

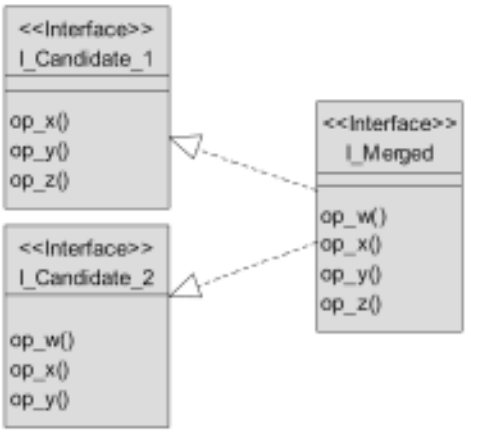


图 7-5　相似“子系统接口”的融合

7.1.8　技巧：“子系统接口”定义的调整

并非所有的“子系统接口”都将被拟建（待开发）的子系统实现。很多时候，以往开发的子系统或直接购买的构件能够提供与“子系统接口”定义类似的服务内容，但是并不能作到完全匹配。在确保实现应用逻辑的前提下，可以对“子系统接口”的定义作出适当的调整和折衷，从而扩大“复用”机会。作为“契约”的“子系统接口”定义拥有两种类型的使用者，因而调整和折衷工作宜早不宜迟。

7.1.9　技巧：“子系统接口”在模型中的位置

“子系统接口”可以被某一特定子系统实现，也可能通过多个子系统的协作实现。“子系统接口”作为设计内容，在设计模型中的相对位置有两种选择。如果一个“子系统接口”将被某一特定子系统实现，可以将该“子系统接口”与相应的子系统放在一起。如果“子系统接口”由多个子系统协同实现，通常将这类“子系统接口”放在一个由系统构架师统一管理的包中。

7.1.10　技巧：推迟明确“设计类”的操作

鉴于“子系统接口”对于拟建系统构架具有全局性的影响力，其操作的定义内容应当具体而明确。对于“设计类”而言，在“全局设计”任务中，尚且不需要将原有“分析类”的“责任”精准地转换成相应的操作。因为在后续设计活动中，一般的“设计类”还要经历很多调整与精化。与“子系统接口”不同，“设计类”的操作并不涉及两个完全解耦又需要保持一致的设计活动。

7.1.11 示例

首先，将“分析类”映射成核心“设计元素”，参见表 7-1。省略号代表由其他“问题局部”映射而来的“设计元素”。注意，由不同“问题局部”得到的“设计元素”存在重合的内容。

表 7-1 “分析类”映射成“设计元素”示例

构 造 型	“分析类”	“设计元素”
	SubmitClaimForm	“设计类” SubmitClaimForm
	HRDatabase	“子系统接口” I_HRDatabase
	MailSystem	“子系统接口” I_MailSystem
	…	…
	SubmitClaim	“设计类” SubmitClaim
	…	…
	Employee	“设计类” Employee
	Claim_report	“设计类” Claim_report
	Claim_record	“设计类” Claim_record
	Valid_rule	“设计类” Valid_rule
	…	…

将“分析类”映射为“设计元素”之后，“分析类”的三种构造型基本完成其历史使命。

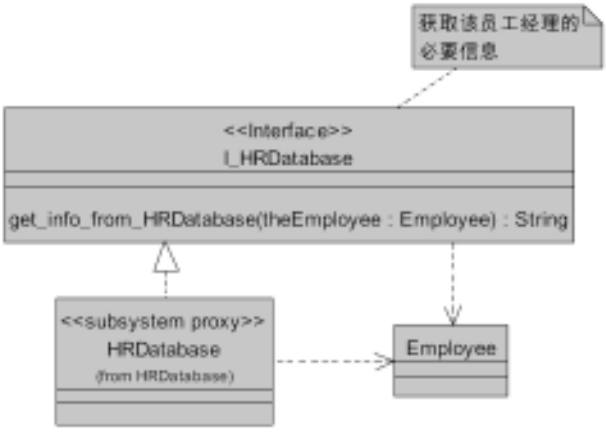


图 7-6 “子系统接口”I_HRDatabase 被特定子系统实现

接下来，定义“子系统接口”的操作并明确“子系统接口”对其他“设计元素”的依赖关系，参见图 7-6 和图 7-7。例如，“子系统接口”I_HRDatabase 的操作 get_info_from_HRDatabase（theEmployee：Employee）：String 中“设计类”Employee 作为一个参数的类型，因而 I_HRDatabase 和 Employee 之间存在依赖关系，参见图 7-6。

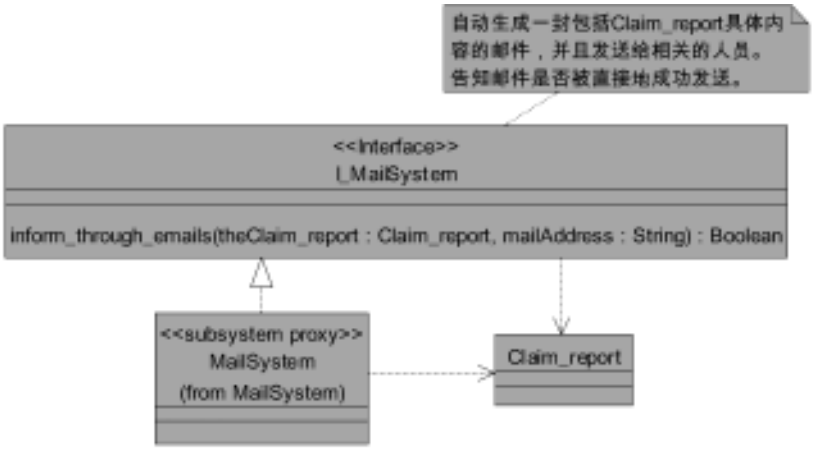


图 7-7 “子系统接口”I_MailSystem 被特定子系统实现

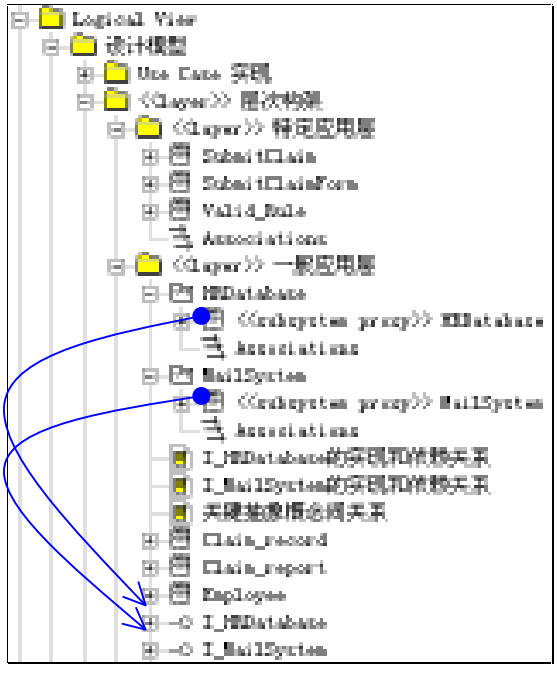


图 7-8 层次构架中的“核心设计元素”

经过“确定核心元素”活动，设计模型中层次构架的主要内容参见图 7-8。在每个用于实现“子系统接口”的子系统中建立了一个《subsystem proxy》类。现阶段的子系统内部，除了《subsystem proxy》类之外，尚且一片空白。针对每个“子系统接口”分别添加了名为“某某子系统接口的实现与依赖关系”的类图，具体内容参见图 7-6 和图 7-7。图中用《subsystem proxy》表述特定的拟建子系统，“实现关系”表达贯通黑盒（“子系统接口”）与白盒（特定的拟建子系统）的逻辑纽带。

7.2 引入外围元素

“引入外围元素”活动的主要依据包括位于中高层次中的“核心设计元素”以及分析活动中得到的“分析机制”;该活动的结果是包含具体内容的“设计机制”和被充实的层次构架中低层次，参见图 7-9。

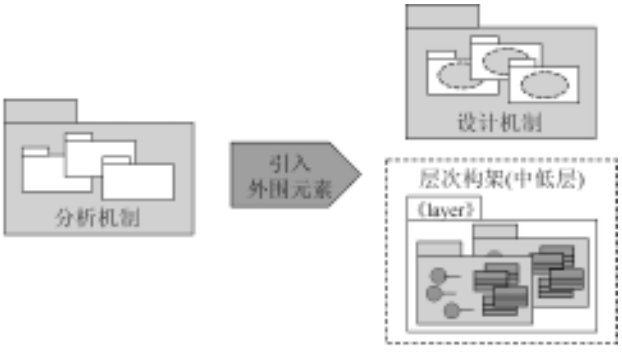


图 7-9 “引入外围元素”活动图示

7.2.1 概念：“设计机制”与“实施机制”

“分析机制”概括地表述那些有待具体化的基本服务，“分析机制”为较复杂的行为提供了一种简便的表达“符号”。由于这些具体行为和应用逻辑没有直接的联系，因而不必在早期分散分析活动注意力。但是，随着分析任务向设计任务的转换，“分析机制”需要通过“设计机制”及“实施机制”予以“落实”。

“设计机制”用于实现相应的“分析机制”，在“分析机制”的框架中添加具体内容。“实施机制”运用特定的实施技术实现相应的“设计机制”，“设计机制”是对“实施机制”的抽象表述。

“分析机制”通过“设计机制”过渡到“实施机制”。借助“设计机制”，可以在间接支撑应用逻辑的同时不至于全面陷入技术的细节。概念上，“设计机制”

为选取特定“实施机制”留出了余地。当然，实践中，讨论与“实施机制”完全“脱钩”的“设计机制”往往并不容易。关于“构架机制”的综合概念，读者可以参考“实践过程概述”和“全局分析”中的相关内容。

7.2.2 概念：“外围设计元素”的含义

“外围设计元素”与拟建系统的应用逻辑不直接相关，它们辅助“核心设计元素”具体地利用“分析机制”所概括的基本服务。“外围设计元素”通常伴随“设计机制”及“实施机制”的逐步“落实”而被引入拟建系统构架的中低层次。换言之，明确“设计机制”及“实施机制”的过程将“带动”向设计模型中引入“外围设计元素”。

在面向对象应用系统中，并不是所有的类都和软件需求中的应用逻辑有直接关系。实践中，有两种类和应用逻辑不直接相关：其一，已经被前人定义好的类，这些类有秩序地存在于各个类库中，不妨称这些类的逻辑表述为“基础设计元素”；其二，尽管不是被预定义好的类，但是它们在不同应用场合的形式和内容总是大同小异，不妨称这些类的逻辑表述为“衔接设计元素”。简单讲，“外围设计元素”就是“基础设计元素”和“衔接设计元素”的集合。

“设计机制”描述了如何利用“基础设计元素”和“衔接设计元素”实现“核心设计元素”所需要的那些与应用逻辑不直接相关的必要服务。“设计机制”的关键内容是“衔接设计元素”的使用方法。形象地讲，“衔接设计元素”是将“核心设计元素”（应用逻辑的反映）与“基础设计元素”（可复用的资产）“粘连”在一起的“胶”，是“需求驱动”与“经验支撑”在微观层面的枢纽，体现出“构架机制”的核心价值。

设计模型中的“构架机制”所包含的内容用于描述“构架机制”的使用方法。形象地讲，“构架机制”包中的内容就是“核心设计元素”与“外围设计元素”相互协作的“范例”（即所谓 Parameterized Collaboration）。在这种范例中，表达“核心设计元素”和“衔接设计元素”的内容用类的构造型《role》描述，它们位于相应的“构架机制”包中；表达“基础设计元素”的内容并不属于特定的“构架机制”包，它们被纳入层次构架的中低层。

7.2.3 步骤 1：“分析机制”向“设计机制”映射

首先，充实“分析机制”的内容。在“全局分析”任务中，我们建立了“关键抽象和分析机制的关联表”；在此基础上扩展，可以得到“核心设计元素和分析机制的关联表”。“核心设计元素”成为“分析机制”的使用者，针对当初圈定的“分析机制”技术特征，估计“核心设计元素”在各项技术特征上的取值范围。

然后，从“分析机制”向“设计机制”映射。针对同一种“分析机制”，不同“核心设计元素”在技术特性（Characteristic）上的取值范围有可能存在很大差异，从而对应不同的“设计机制”及“实施机制”。以解决数据存取问题的“留存”机

制为例，留存持续时间的差异会导致留存介质的区别（比如内存、临时文件或者数据库），进而导致完全不同“设计机制”及“实施机制”。根据使用同一“分析机制”的多个“核心设计元素”在技术特征取值方面的相似性或不同点，可以将这些“核心设计元素”大致分成几个组，每个组将对应一种具体的“设计机制”，参见图 7-10。这就是从“分析机制”向“设计机制” 的映射过程。为了便于后续设计活动参考，通常将“构架机制”在不同层面的映射关联以及机制的使用者用表格列出，参见表 7-2。

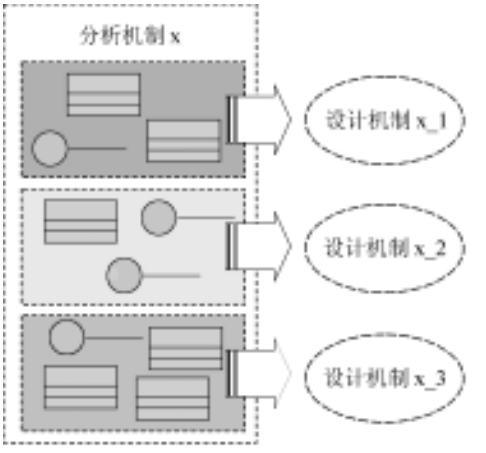


图 7-10 “分析机制”和“设计机制”的映射示意

表 7-2 “构架机制”及其使用者

“分析机制”	“设计机制”	“实施机制”	使用者（“核心设计元素”）
…	…	…	…

7.2.4 步骤 2：落实“设计机制”的具体内容

首先，引入特定“设计机制”中涉及的“基础设计元素”。前人定义好的类往往以物理形式存在，对它们进行逆向工程[1]，以便得到相应的逻辑表述，即所谓的“基础设计元素”。将这些“基础设计元素”放在层次构架的“通用服务层”中，它们将作为描述“设计机制”的素材（协作的参与者）。客观上，向层次构架的中低层中引入了一部分“外围设计元素”。

接着，在“构架机制”包中，为每一个（被初步认定的）“设计机制”建立一个独立的包。在每个包中，运用类的构造型《role》建立相关“核心设计元素”的“适配器”

[1] 通常借助建模工具实现。

以及“衔接设计元素”的“模子”。后者是部分“外围设计元素”的“孵化器”。

然后，通过动态和静态两种方式，描述“设计机制”的具体协作关系。一方面，通过“参与类图”描述“设计机制”的静态结构；简要描述“参与类”的用途和用法，“参与类”包括“基础设计元素”以及表达“衔接设计元素”和相关“核心设计元素”的构造型《role》。另一方面，对于“设计机制”的典型应用情景，用序列图描述其动态特征。

7.2.5 技巧：“设计机制”的分组

“分析机制”向“设计机制”映射过程中，根据“分析机制”使用者技术特征的取值范围进行分组，然后确定不同种类的“设计机制”。这种取值是一种大致的估计，尺度由系统构架师掌握。总的原则是在满足基本要求的条件下，分组越少越好。实践中，需要权衡利弊，避免走向极端。如果同一“分析机制”对应的不同“设计机制”过多，设计模型将变得很复杂；相反，如果“设计机制”很少，意味着每个“设计机制”需要适应的技术特征取值范围可能过大，通常会带来性能或其他方面的不利影响。

7.2.6 技巧：“实施机制”的综合考虑

由于“设计机制”对后续设计活动的影响是全局性的，因而找到适用的“实施机制”非常关键，通常要尽早对“实施机制”进行技术可行性的全面论证和实验。根据“设计机制”选择“实施机制”时，不仅考虑技术方面的需要，还应综合考虑一些非技术问题，比如成熟度、成本以及可持续发展能力等。“引入外围元素”活动在很大程度上依赖于系统构架师的经验。可用的经验每天都在大量产生，系统构架师不应该只依赖自身已有的经验，更应该具有开阔的视线和敏锐的借鉴力，多参阅资料，多与有经验的同行沟通。

7.2.7 示例

类似于表 7-2，结合示例的问题局部，得到“构架机制”在不同层面的映射以及相应的使用者，参见表 7-3。实践中，这种映射应该是全局范围的。

考虑到后续章节的沿用，描述两种典型“设计机制”[1]的具体内容：Java 实现的“留存”机制与“分布处理”机制。实践中[2]，这两种“构架机制”的适用性比较普遍。描述“机制”的过程将伴随着“外围设计元素”的引入。希望读者将理解重点放在“设计机制”的“描述过程”，因为它具有普遍意义，可用于描述读者自己的“设计机制”。

---

[1] 以 J2EE 为总的技术背景。

[2] 拟建系统的前端实现方式和可选的技术手段种类繁多，相应的内容不作为本书讨论的重点。

表 7-3 “构架机制”映射及其使用者示例

“分析机制”	“设计机制”	“实施机制”	使用者（“核心设计元素”）
留存	RDBMS	JDBC	HRDatabase； Employee； Claim_report； Claim_record； Valid_rule； …
	…	…	…
分布处理	RMI	Java 1.3	SubmitClaim …
	…	…	…
…	…	…	…

描述“留存”机制

第一步，引入“基础设计元素”，即那些不作任何调整而直接利用的类。

用 JDBC 实现“留存”机制，需要使用 java.sql 包中的类。对 java.sql 包进行逆向工程[1]，然后将相关“基础设计元素”加入层次构架的“通用服务层”，参见图 7-11。这些“基础设计元素”是为支撑“留存”机制而直接引入的“外围设计元素”。

图 7-11 将 java.sql 包引入层次构架的“通用服务层”

参与实现“留存”机制的“基础设计元素”主要包括以下内容[2]。

[1] 主流的建模工具提供基于某种技术的整体模型框架，这种情况下，相关的“逆向工程”已经被工具厂商预先完成，因而可以直接使用。

[2] 关于这些类的定义和详细用法，读者可以参考《JDBC Specification》

- DriverManager。用于建立数据库连接。
- Connection。用于保持与数据库之间的连接。
- Statement。用于直接和数据库进行交互。
- ResultSet。用于记录 SQL 查询返回的结果。

第二步，在“构架机制”包的“留存”包中，建立一个名为“RDBMS-JDBC”的包，参见图 7-12。

图 7-12　用 RDBMS-JDBC 实现“留存”机制的模型内容

第三步，运用类的构造型《role》建立相关“核心设计元素”的“适配器”以及“衔接设计元素”的“模子”，并且简述它们的用法。图 7-12 表明它们在模型中的相对位置（“RMI-Java 1.3”包中），主要包括以下内容。

- 《role》PersistentClass。这个类将与作为“留存”机制使用者的“核心设计元素”绑定在一起，简称“留存类”。[□]
- 《role》PersistencyClient。这个类代表“留存类”（在“留存”相关协作中）的交互者所对应的“核心设计元素”，简称“留存类的用户”。
- 《role》PersistentClassList。这个类代表一个“衔接设计元素”，用于返回一组作为数据库查询结果的“留存类”的对象。
- 《role》DBClass。这个类代表一个“衔接设计元素”，充当“活动协调人”的角色。在 JDBC 中，使用 DBClass 来读写需要留存的数据。DBClass 利用 DriverManager 建立数据库连接，然后借助 Connection 创建 SQL 语句，接下来通过 Statement 与数据库进行交互。DBClass 负责创

□ “留存类”为使用“留存”机制付出的“代价”是为 DBClass 提供访问其属性的途径，这些属性原本可以完全隐蔽。

建“留存类”实例，DBClass 了解 OO 与 RDBMS 之间的映射，能够代理拟建系统与 RDBMS 进行交互。每个“留存类”有一个相应的 DBClass 负责其“留存”事宜。

第四步，描述机制的静态结构，参见图 7-13。

描述机制静态结构的“参与类图”在模型中的位置参见图 7-12。不难看出，《role》DBClass 处于枢纽的位置。《role》PersistentClassList 和《role》PersistentClass 之间存在聚合关系；《role》DBClass 和 Connection 之间存在单方向的关联关系；其余的关系都是相对松散的依赖关系。值得一提的是，对于“留存类的用户”（《role》PersistencyClient）的要求很弱，仅需要提供该类到《role》DBClass 的依赖关系。

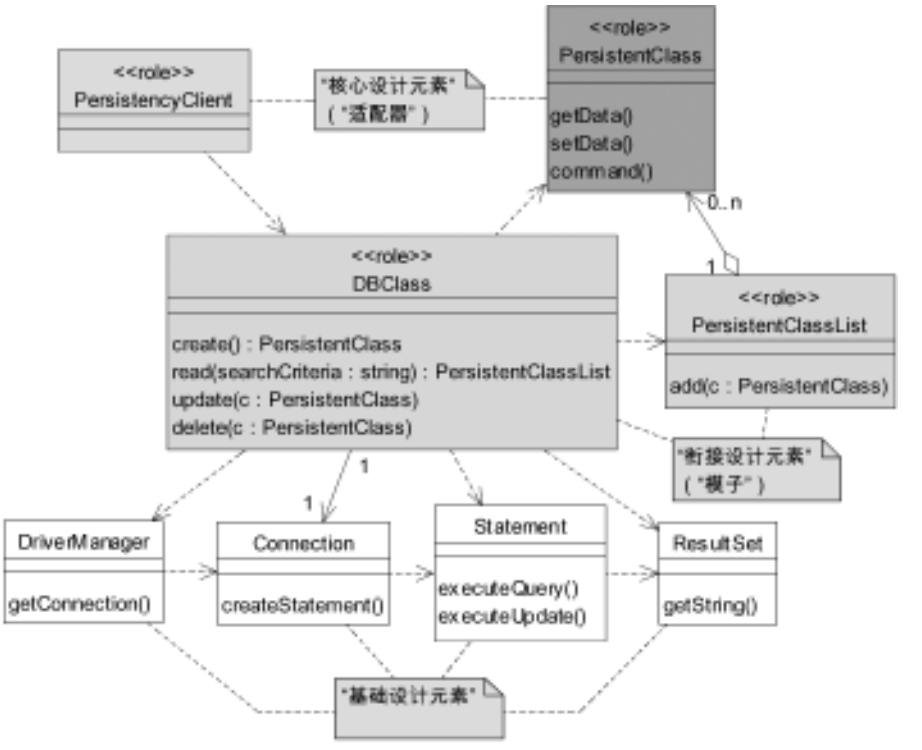


图 7-13　JDBC 实现“留存”机制的静态机构

第五步，描述机制的典型应用场景。

和数据库打交道，典型的应用场景就是“增、删、查、改”，以下分别列出相应的序列图。图 7-14 描述创建一个“留存类”实例对应的消息传递序列。在创建“留存类”实例时，DBClass 负责将相应的记录写入数据库。

图 7-15 描述删除一个”留存类”实例对应的消息传递序列。在删除“留存类”实例时，DBClass 负责将相应的记录从数据库中删除。

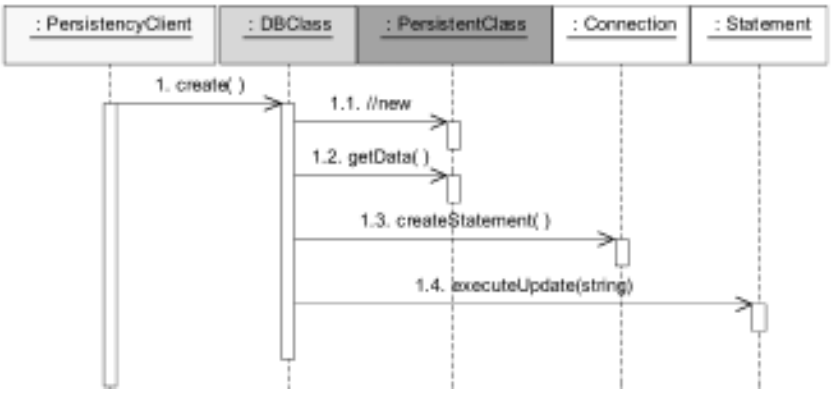


图 7-14　创建“留存类”的实例（“增”）

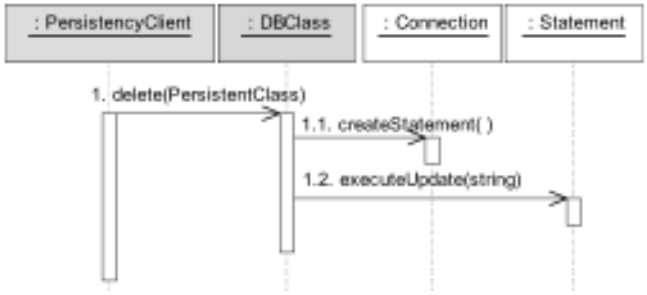


图 7-15　删除“留存类”的实例（“删”）

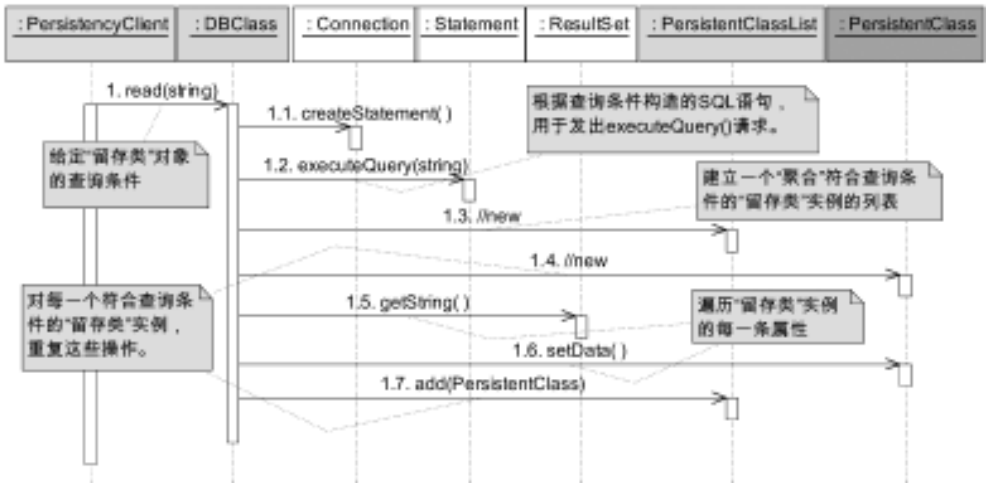


图 7-16　读取“留存类”的实例（“查”）

图 7-16 描述读取符合条件的“留存类”实例（组）所对应的消息传递序列。“查”和“增、删、改”的不同之处是有可能涉及“留存类”的多个实例（符合特定查询条件），因而序列图相对复杂。PersistentClassList 负责将多个“留存类”实例“聚合”在一起。注意，操作 read（String）与操作 executeQuery（String）都以字符串作参数提供查询的条件，但是两个操作的使用者视角有所区别。鉴于 DBClass 为 PersistencyClient 提供专业化的“留存”相关服务，操作 read（String）的使用者 PersistencyClient 可以在不了解数据库内部知识的情况下创建有效的查询。

图 7-17 描述更新一个“留存类”实例对应的消息传递序列。在更新“留存类”实例时，DBClass 负责更新数据库中的相应记录。

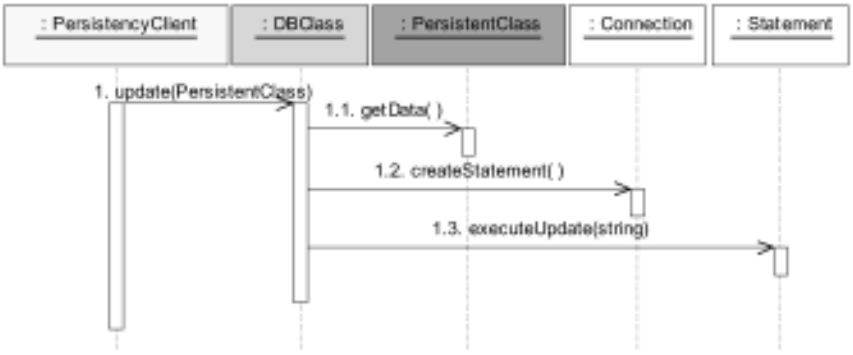


图 7-17　更新“留存类”的实例（“改”）

注意，“增、删、查、改”能够得以进行的大前提是“初始化数据库连接”，其时序关系参见图 7-18。DBClass 提供数据库的位置（参数 url）、合法的用户（参数 usr）和密码（参数 pass），呼叫 DriverManager 的服务，建立数据库连接。

图 7-18　初始化数据库连接

描述“分布处理”机制

第一步，引入“基础设计元素”，即那些不作任何调整而直接利用的类。

采用 RMI 实现“分布处理”机制，需要使用 java.oi 和 java.rmi 包中的类。将

该包进行逆向工程，然后将相关“基础设计元素”加入层次构架的“通用服务层”，参见图 7-19。这些“基础设计元素”是为支撑“分布处理”机制而直接引入的“外围设计元素”。

参与实现“分布处理”机制的“基础设计元素”主要包括以下内容[1]。

- Naming。用于寻找分布在异地的对象，每一“地点”有一个此类的实例。
- Serializable。一个 Java 的 Interface。异地之间作为参数传送的对象必须实现（Implement）这个 Java 的 Interface。
- Remote。一个 Java Interface。被分布到异地的对象的类必须（直接或间接）实现（Implement）这个 Java 的 Interface。对于实现 Remote Interface 的类，环境会建立相应的 remote stub 和 remote skeleton，由它们具体解决实际运行中的分布处理通讯。
- UnicastRemoteObject。支撑创建和导出（Exporting）被分布到异地的对象。

图 7-19　将 java.oi 和 java.rmi 引入层次构架的“通用服务层”

第二步，在“构架机制”包的“分布处理”包中，建立一个名为“RMI-Java 1.31.3”的包，参见图 7-20。

第三步，运用类的构造型《role》建立相关“核心设计元素”的“适配器”以及“衔接设计元素”的“模子”，并且简述它们的用法。图 7-20 表明它们在模型中的相对位置（“RMI-Java 1.31.3”包中），主要包括以下内容。

- 《role》DistributedClass。这个类将与作为“分布处理”机制使用者的“核心设计元素”绑定在一起，简称“分布类”。
- 《role》DistributedClassClient。这个类代表“分布类”（在“分布处理”相

[1] 关于这些类的定义和详细用法，读者可以参考《Java Remote Method Invocation Specification》。

关协作中）的交互者所对应的“核心设计元素”，简称“分布类的用户”。

- 《role》PassedData。在异地消息传送中，这个类代表那些被作为参数传递的对象所对应的“核心设计元素”，可能不止一个。
- 《role》IdistributedClass。一个 Java 的 Interface，逻辑上代表被分布到“异地”的“分布类”在“本地”的“代言人”，简称“代言者”。概念上，属于“衔接设计元素”。

图 7-20　用 RMI - Java1.3 实现“分布处理”机制的模型内容

第四步，描述机制的静态结构，参见图 7-21。

描述机制静态结构的“参与类图”在模型中的位置参见图 7-21。不难看出，《role》IDistributedClass 处于枢纽的位置。笼统地讲，有以下几个要点。

- “分布类的用户”要借助 Naming 的实例寻找“分布类”，因而“分布类的用户”和 Naming 之间存在依赖关系。
- 为了使“本地”的“代言者”和“异地”的“分布类”等效，“代言者”要“继承”（extend） Remote，“分布类”要“继承”（extend） UnicastRemoteObject，从而借助 Java 的支撑环境实现“近如咫尺”。
- 为了在异地之间传递对象作为远程操作的参数，相关的类需要实现（implement）Serializable。
- 《role》DistributedClassClient 对《role》IDistributedClass 的依赖关系是假设《role》DistributedClassClient 原先对《role》DistributedClass 存在依赖关系。如果《role》DistributedClassClient 原先对《role》DistributedClass 存在单向关联关系，那么《role》 DistributedClassClient 到《role》IDistributedClass 将有同样的关系。

第五步，描述机制的典型应用场景，即呼叫异地对象的操作，参见图 7-22。

Naming 实例中 lookup 操作的参数是用于寻找异地对象的统一资源定位符号（URL，Uniform Resource Locator），具体的形式为“rmi://host:port/name”，包含异地对象所在的主机名称，端口名称和对象本身的名称。

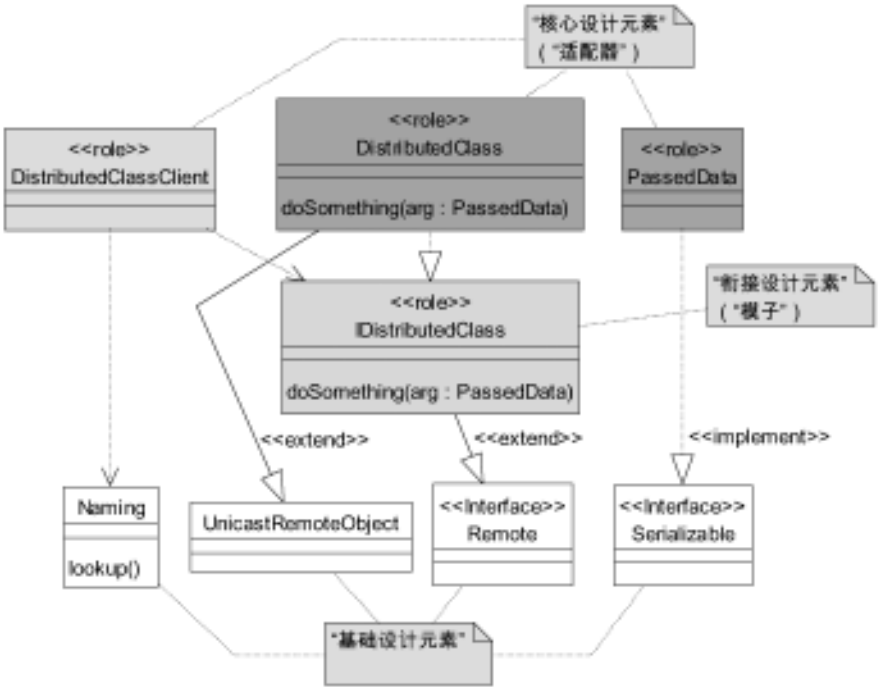


图 7-21　RMI **实现“分布处理”机制的静态结构**

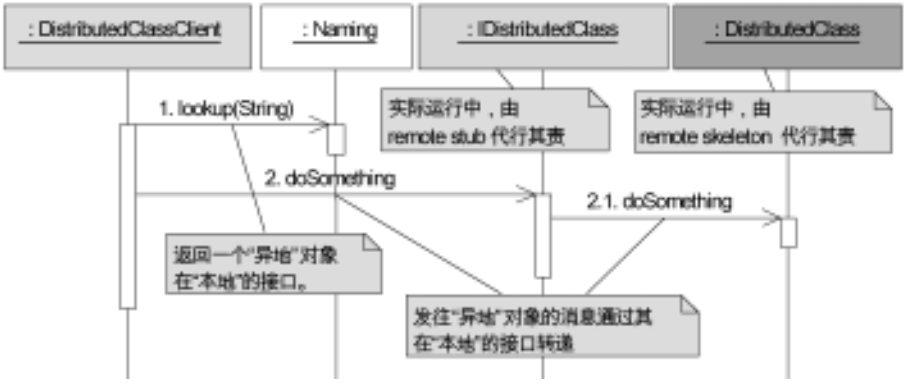


图 7-22　**呼叫异地对象的操作**

7.3　优化组织结构

“优化组织结构”活动的依据是“全局分析”任务以来在层次结构中不断积

累的内容，笼统称之为“被充实”的层次构架；该活动的结果将是被整理后的层次构架，参见图 7-23。在每个层次中会有多个包，在包中有“设计元素”以及表示“设计元素”之间关系或包之间关系的（类）图。

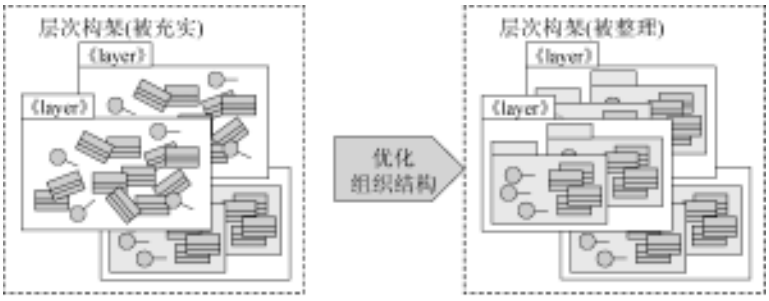


图 7-23 “优化组织结构”活动图示

7.3.1 概念：层次构架内容的复用价值

层次构架以复用价值为基本依据，将设计模型的核心内容划分成几个大规模的包。在模型中用包的构造型《layer》表达层的概念。层次的划分并没有绝对的标准，层次使得设计内容更加易于理解和便于管理。

在分析任务中，活动主要面向应用逻辑本身，相应的工作结果具有较强的（问题）针对性，但它们往往具有较弱的可复用价值，通常位于较高的层次，例如“特定应用层”和“一般应用层”。进入设计任务后，为落实“构架机制”的支撑作用，引入了许多与应用不直接相关的“外围设计元素”，它们具有较高的复用价值，通常被放置在“通用服务层”。

开发一般的应用系统，基本不会涉及“系统服务层”的内容，通常直接采用厂商提供的工具（Utilities）操控外接设备和网络设施。客观上，这类层次中的内容往往远离拟建系统的核心价值，是典型的“很难”且“不很重要”的部分。举一个传统工业中的例子，制造汽车的厂商未必自己制造轮胎，这并不意味着他们的汽车不用轮胎，他们更关注某一款车的外观、整体机动性能或驾驶舒适性等直接作用于汽车使用者的关键特性。

7.3.2 概念：层次构架中积累的内容

随着分析和设计活动的逐步展开，拟建系统层次构架中的内容逐渐地丰富起来。最初，在“全局分析”任务中，将捕获的“关键抽象”放置在“一般应用层”的“关键抽象”包中。接着，在多个“局部分析”任务中，得到的“分析类”分布于“特定应用层”和“一般应用层”，其中一部分是“关键抽象”的沿用。然后，

在“全局设计”任务中，为了“落实”早期充当“占位符”的“分析机制”，作为“基础设计元素”的“外围设计元素”被引入“通用服务层”。系统构架师和设计师共同介入分析和设计活动，层次构架中的要素和图在数量上大幅度增长，优化组织结构的必要性逐渐明显。

“全局设计”任务中的“优化组织结构”活动呼应“全局分析”任务中的“选定层次构架”活动。事实上，在“全局分析”任务中，不管选用何种类型的体系构架模式，都有必要在“全局设计”任务的后期对构架的内容和组织加以整理和优化。全局范围的优化工作不仅限于简单地重新摆放，这一过程往往能给系统构架师更多发现与利用复用价值的机会。

7.3.3 概念：包之间的依赖关系

按外部的“可见度”（Visibility）划分，一个包中可以包含“公开类”和“私有类”：“公开类”可以被包外的类引用，“私有类”只能被宿主包内的类引用。如果包 X 中的类 A 引用包 Y 中的“公开类”B 之间存在依赖关系，那么包 X“依赖”于包 Y。图 7-24 给出一个概念化的示例。

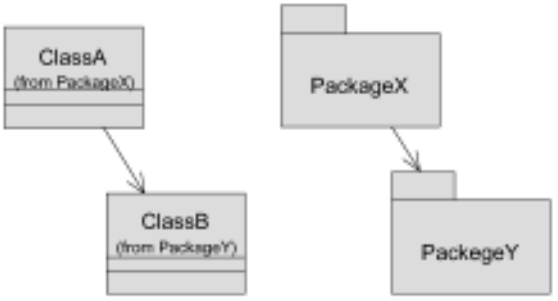


图 7-24　包之间依赖关系的基本依据

包之间的依赖关系取决于不同包中的“设计元素”之间的关系，这种依赖关系反映出构架的耦合状况。

7.3.4 步骤 1：分包组织“设计元素”

为了便于理解，可以在层次结构的基础上划分更细的组织单元。具体的作法是将“设计元素”分组放入特定的包中。逻辑上，分包的目的是使“设计元素”更有秩序，甚至呈现出更明显的高内聚、低耦合特征。实践中[□]，不同的包允许个人或团队相对独立地展开后续的设计和实施工作。尽管很难实现完全的独立，包之间相对松散的耦合通常也能大大简化复杂系统的开发。可以参照以下几个原则进行划分。

[□] 可以近似地认为是“具有更明确的物理意义”。

- 将直接和特定类型用户（主导 Actor）关联的边界类所对应的“设计元素”放在同一个包中[1]，通常意义上，这个包就是针对这种类型用户的界面。
- 将功能相关性比较强的“设计元素”放在同一个包中，可以近似理解[2]为通常意义上的“功能模块”。
- 将（针对被动 Actor 的）边界类相关的“设计元素”放在同一包中，可以近似理解为通常意义上的“后台系统接口”。

7.3.5 步骤 2：描述包之间的依赖关系

根据“整理分析类”、“确定核心元素”以及“引入外围元素”活动的结果，对构架中包之间的依赖关系作一简要描述。

根据“整理分析类”活动获得的参与类图，可以获得分处于不同包的类之间关系，进而推断出相应包之间的依赖关系。

在“确定核心元素”活动的“定义子系统接口”步骤中，得到了部分“设计元素”之间的依赖关系。根据这些依赖关系描述相关包之间的依赖关系。

在涉及“构架机制”的“核心设计元素”所在包与相关“基础设计元素”所在包之间建立依赖关系。

随着设计活动的进一步展开，将会得到更多的“设计元素”以及它们之间的依赖关系。准确地说，描述构架中包之间的依赖关系是一项持续的工作。这些依赖关系在后续活动中还会经历调整，故而，早期只须给出简单的图形化说明，当依赖关系相对稳定之后再作较为详细的文字描述。

7.3.6 技巧：利用层次内的分区信息

如果在建立层次构架之初，已经在层次内部根据拟建系统用户的组织机构、开发小组的技能专长、系统的物理分布、信息的保密等级或者功能的必然性水平作出了比较明确的分区，那么可以在“打包”的时候直接利用这种逻辑上的划分。可以参考“全局分析”任务中“选用构架模式”活动的相关“技巧”。

7.3.7 技巧：判别“紧密相关”的类

判断是否将两个类放入同一个包中时，需要确定类之间是否“紧密相关”，可以依次考虑下面的几种因素。

- 如果一个类 A 的行为或结构的变化使得另一个类 B 也必须相应地变化，

---

[1] 有可能包括那些和边界类紧密配合的控制类。

[2] 请注意，直到系统的构架已经相对明确和稳定之后，才引入类似“功能模块”的概念，而不是在一开始的时候。换言之，没有在一开始就作“功能分解”，相反，始终在应用面向对象的思想。与“功能模块”类似的概念是一种客观的结果，从某种意义上可以理解为“功能的聚合”。

这两个类在功能上“紧密相关”。设想一个极端的情形，假设将类 A 删除之后，类 B 完全失去用场（成为多余），说明类 B 只为类 A 所使用，完全依赖于类 A。此时，类 B 应当处于类 A 所在的包。

- 如果两个类的对象进行大量的消息交互，或者以其他方式进行复杂的通信，这两个类在功能上“紧密相关”。
- Actor 是组织需求的外在核心线索，是引入需求变化影响的起点。如果两个类的对象与同一个 Actor 进行交互，或受到对同一个 Actor 变化的影响，这两个类在功能上相关。
- 如果两个类之间存在关联关系甚至聚合关系，它们往往在功能上相关。
- 一个类与创建其实例的类往往在功能上相关。

反过来，两个类之间的相关性越弱意味着它们之间的耦合越松散，往往不宜放在同一个包中。例如，对于不共同涉及一个 Actor 的两个类，通常不将它们放在同一个包中；再如，可有可无的类和必须存在的类不宜放在同一个包中。

7.3.8 技巧：针对“不易分拆”的包

理想情况下，希望在层次构架中获得高内聚、低耦合、并且大小适度的一组包，但有些时候会遇到一些困难。假设拟议将包 A 分拆为包 A1 和包 A2，但是包 A 中的类 X 同时与包 A1 和包 A2 中的类存在相对密切的“关联”或“被使用”关系，很难简单地将类 X 放入包 A1 或者包 A2 中。另一方面，如果将所有的类全部留在包 A 中，这个包的个头未免偏大。包 A 即所谓的“不易拆分”的包。

这类问题通常有两种解决途径：如果类 X 中的内容可以根据包 A1 和包 A2 的不同要求进行分组（两组内容之间的关系相对松散），那么将类 X 拆分成两个类（X1 和 X2）分别纳入包 A1 和包 A2；否则，意味着类 X 的内容将以密不可分的形式被两个拟议分拆的包共用，此时可以考虑将类 X 转移到较低层次的另一个包 B 中，并且包 A1 和包 A2 同时依赖包 B。

7.3.9 技巧：弱化包之间的耦合关系

“高内聚、低耦合”原则是努力的大方向。包之间的耦合表现为依赖关系，并非“一无是处”。包之间的依赖关系意味着有机会使用其他包中对象的行为。当然，从复用局部静态构造的角度考虑，会带来一些困难。决定和调整包之间的依赖关系时，可以参考以下的一些原则。

- 尽量减少甚至消除包之间的“互相依赖关系”，或称为“循环依赖关系”。即避免包 A 依赖包 B 的同时包 B 依赖包 A。
- 复用价值较高的包不要依赖复用价值较低的包。在层次构架中，处于较低层次中的包不要依赖处于较高层次中的包。
- 当层次构架中的层数较多时，不同层次中“设计元素”的复用价值应当递阶变化。因而，较高层次内包对较低层次内包的依赖关系尽可能只涉

及相邻的层次。当然，跨越层次的依赖关系在有些情况下也在所难免，尤其是对那些提供“通用服务”的包。

- 用于实现“子系统接口”的特定子系统内容也被包装在包中，不要让其他的包直接依赖这类包，换言之，不要直接依赖这类包中的“设计元素”。正确的做法是依赖相应的“子系统接口”。

7.3.10 技巧：“包的事实接口”

本质上，一个包内的所有“公开类”的公开操作构成“包的事实接口”，它解除外部（其他包的内容）对该包内部内容的依赖关系。如果有条件及早获得比较稳定的“包的事实接口”，该包内外两侧的内容可以被很好地解耦，从而能够进一步独立地开发两部分的内容。“包的事实接口”带来的益处类似于“子系统接口”。不同的是，“子系统接口”是主动提出的要求，“包的事实接口”是客观形成的事实。

7.3.11 示例

针对本书示例中讨论的问题局部，经过前面的分析和设计活动，层次构架中积累了一些内容，以下给出“优化组织结构”活动的一些要点。

图 7-25　被充实的层次构架示例

图 7-25 展现整理之前“被充实的层次构架”，图 7-28 是优化之后的结果。必须指出，图 7-28 同样是一个过渡性的结果，即层次构架演进过程中的一个特定状态而已。贯穿于分析和设计的实践过程，层次构架将不断地被整理和优化。

首先，分包组织“设计元素”。

- 在“特定应用层”内建立 Claim Activities 包，将类 SubmitClaim 和类 SubmitClaimForm 放在一起，参见图 7-28。同时，在该包内用一张类图描述“包内部的关系”，参见图 7-26。注意，这种图中体现出的关系来自多个“Use Case 实现”中的“参与类图”，并参考和利用必要的业务常识与规则。
- 在“一般应用层”中建立 Claim Artifacts 包，将关系紧密的（实体）类 Claim_report、Claim_record 和 Valid_rule 放在一起，参见图 7-28。同时，在该包内用一张类图描述“包内部的关系”，参见图 7-27。将原本处于“特殊应用层”的类 Valid_rule 放在此处，主要有两个原因：一方面，考虑类 Valid_rule 与类 Claim_report 的关系比较紧密；另一方面，考虑其复用可能性，因为这个类很可能参与（“提交报销申请”以外，与财务人员相关的）其他“Use Case 实现”的协作。
- 在“一般应用层”中建立 External Interfaces 包，将表述与外部系统交互的“子系统接口”以及表述它们的实现和依赖关系的类图放在此处，参见图 7-28。

然后，描述构架中包之间的依赖关系，参见图 7-29。‘

- 包 Claim Activities 对包 Claim Artifacts 以及包 External Interfaces 的依赖关系来自于这些包内“设计元素”之间的基本关联关系。

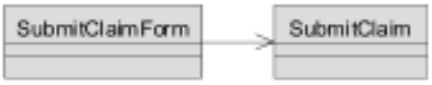


图 7-26 “特定应用层”中 Claim Activities 包内部的关系

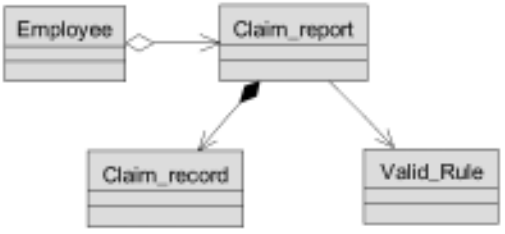


图 7-27 “一般应用层”中 Claim Artifacts 包内部的关系

图 7-28　被优化的层次构架示例

- **根据“子系统接口”I_HRDatabase 对类 Employee 的依赖关系（参见图 7-6），在相关的两个包之间建立依赖关系，即 I_HRDatabase 所在包 External Interfaces 对 Employee 所在包 Claim Artifacts 的依赖关系，参见图 7-29 的中间部分。**
- **“子系统接口”I_MailSystem 对类 Claim_report 的依赖关系（参见图 7-7），对于包之间的依赖关系没有新的贡献。**
- **类 SubmitClaim 将“分布处理”机制协作中的“分布类”（即“构架机制”的使用者），在该类所属包与支撑“分布处理”机制的“基础设计元素”所属包之间建立依赖关系，即包 Claim Activities 对包 java.rmi 的依赖关系，参见图 7-29。注意，此时尚未明确定义 SubmitClaim 中的操作标识（Operation Signature），因而并不能确定哪些类的对象将作为参数被“异地”传送（“局部设计”任务的示例中将涉及相关内容）。**
- **类 Claim_report 和类 Claim_record 将作为“留存”机制协作中的“留存**

类”（即“构架机制”的使用者），在它们所属包和支撑“留存”机制的“基础设计元素”所属包之间建立依赖关系，即包 Claim Artifacts 对包 java.sql 的依赖关系，参见图 7-29。

- 图 7-30 中概括出当前构架中不同层次间的依赖关系，其根据是包之间的依赖关系（参见图 7-29），包之间的依赖关系又是“设计元素”之间关系的一种概括表现形式，“设计元素”之间的关系在各项活动中被逐渐积累起来。

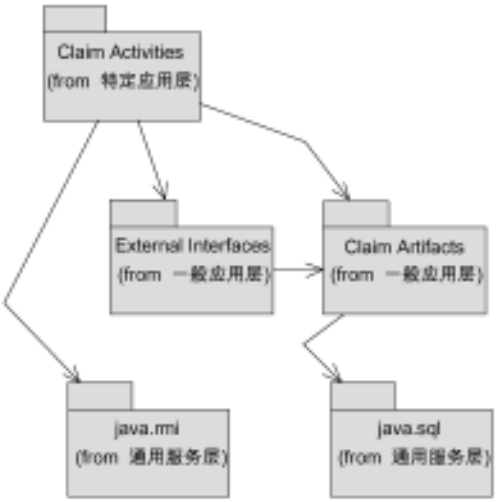


图 7-29　包之间依赖关系示例

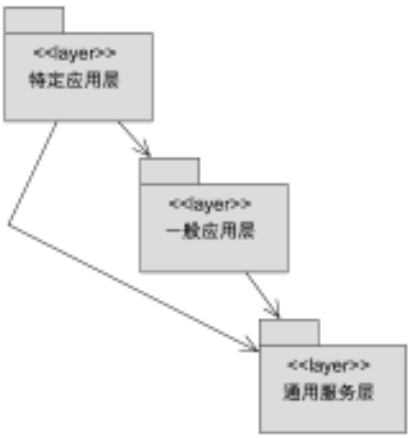


图 7-30　层次之间的依赖关系

针对示例，表 7-4 给出了全局设计任务中积累的模型设计内容。

表 7-4　全局设计任务中积累的设计模型内容汇总

任务	活　　动	设计模型内容		
		“Use Case 实现”	层次构架	“构架机制”
全局设计	确定核心元素		表 7-1， 图 7-6～图 7-8	
	引入外围元素		图 7-11，图 7-19	表 7-3， 图 7-12～图 7-18， 图 7-20～图 7-22
	优化组织结构		图 7-25～图 7-29	

第 8 章　局 部 设 计

“局部设计”任务基于“全局分析”和“局部分析”的框架，利用“全局设计”提供的素材，在不同的局部，将分析的结果用“设计元素”加以“替换”和“落实”。

在“局部设计”任务中，有不同侧重的两项活动，参见图 8-1。

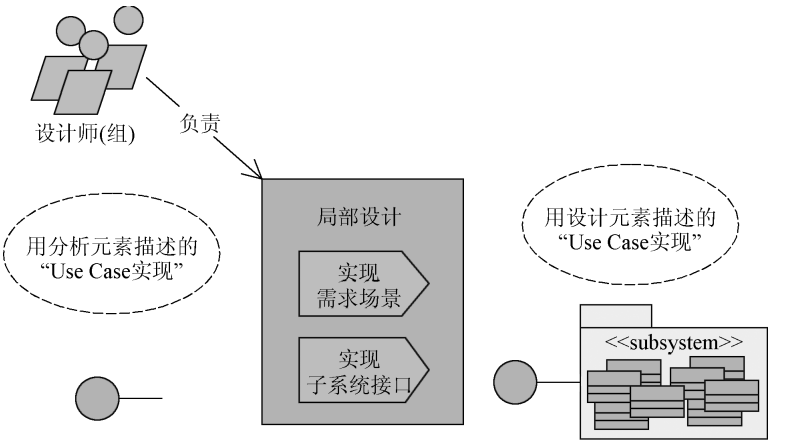


图 8-1　“局部设计”任务的责任人 — 依据 — 活动 — 结果

- 实现需求场景。以“Use Case 实现”为工作范围，将原本由分析元素描述的“Use Case 实现”转换成由“设计元素”描述的“Use Case 实现”。一方面，用“核心设计元素”取代“分析类”，另一方面，“落实”“外围设计元素”对“核心设计元素”（即“构架机制”使用者）的支撑作用。
- 实现子系统接口。以“全局设计”任务中用“子系统接口”解耦的部分作为工作范围，采用类似于“局部分析”的方法，实现“子系统接口”中规定的行为要求。

8.1　实现需求场景

“实现需求场景”活动的基本依据是“局部分析”任务中得到的“Use Case 实现”（即用“分析类”协作关系转述的需求场景）；该活动的结果是设计意义上

的“ Use Case 实现 ”，即用（“ 全局设计 ”任务中得到的）“ 设计元素 ”协作关系描述的“ Use Case 实现 ”，参见图 8-2。

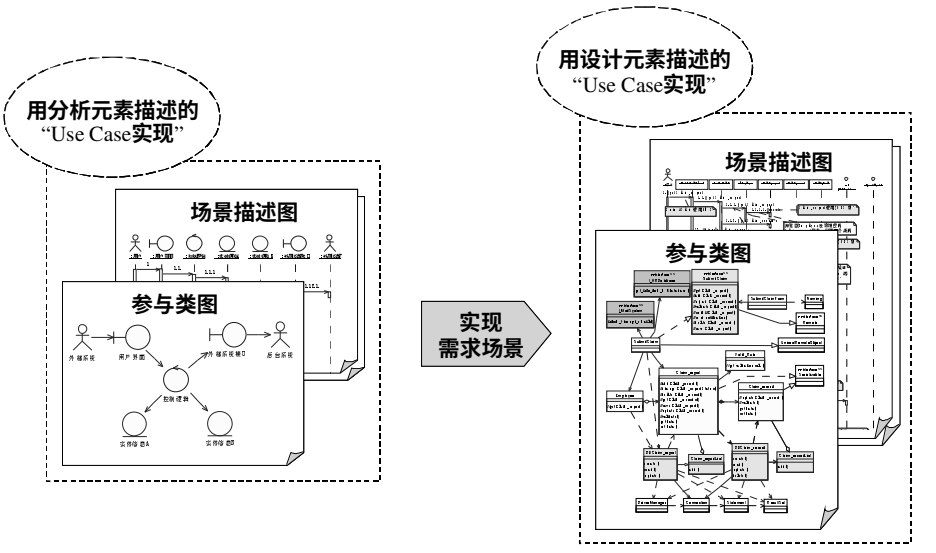


图 8-2 “ 实现需求场景 ”活动图示

8.1.1 概念：“ 分析类 ”和“ 设计元素 ”的差异

“ 分析类 ”和“ 设计元素 ”的本质区别是不同的使用目的，“ 分析类 ”针对问题描述中的要点，可以在 Use Case 事件流描述中找到直接的对应内容；“ 设计元素 ”针对解决方案，将作为后续实施活动切实可行的根据。在分析活动中，很多（与应用逻辑不直接相关的）技术问题细节可以用概要的方式描述和封装；在设计活动中，对大多数技术问题应当有一个明确的交代（逻辑层面）。

为了实现分析活动向设计活动的平滑过渡，在模型中，分析和设计在表现形式上的差异并不是泾渭分明。但是，设计者应该有明确的概念，这一点非常关键。如果没有特殊要求，通常不用保留一个专门的“ 分析模型 ”。注意，在迭代的分析和设计过程中，如果需要获得专门的“ 分析模型 ”，其内容也将经过多个迭代的分析活动逐步积累而成。

8.1.2 步骤 1：用“ 核心设计元素 ”替换“ 分析类 ”

一方面，替换对应“ 子系统接口 ”的“ 分析类 ”。在“ Use Case 实现 ”的“ 参与类图 ”中，用“ 子系统接口 ”表述（满足行为要求的）特定子系统。在“ Use Case 实现 ”的序列图（组）中，如果特定子系统只接收消息，可以用相应“ 子系统接口 ”的“ 实例 ”表述特定子系统；如果特定子系统有可能向外发送消息，用

《subsystem proxy》的实例相应的“分析类”实例。

另一方面，替换直接对应“设计类”的“分析类”，这种替换过程在“Use Case 实现”的动态和静态图中没有显著的变化[1]。围绕“设计类”，建立一个映射列表，枚举它们在当前“Use Case 实现”上下文环境中所使用的“构架机制”场景，参见表 8-1。纵向列出参与当前“Use Case 实现”的“设计类”，横向列出这些“设计类”使用的“构架机制”（现阶段表现为“设计机制”），表格的主体内容是“设计类”所参与的具体事件序列，在这些事件序列中使用了某一“设计机制”的特定应用场景。

表 8-1 “设计类”使用“构架机制”特定场景的列表

“设计机制” / “设计类”	“设计机制”A			“设计机制”B		…
	应用场景 1	应用场景 2	…	…	…	…
“设计类”1	事件序列 i	—	…	…	…	…
“设计类”2	—	事件序列 j	…	…	…	…
…	…	…	…	…	…	…

8.1.3 步骤 2：落实“构架机制”的支撑作用

静态结构方面，绑定“构架机制”。根据“构架机制”的指导，针对作为“构架机制”使用者的“设计类”，添加一些必要的“操作”和属性，建立并衔接必要的“外围设计元素”。在落实“构架机制”过程中会引入新的“设计元素”间关系，在表述包之间依赖关系的类图中作出相应。

动态行为方面，描述“构架机制”在“Use Case 实现”中的应用场景。针对“构架机制”使用者所参与的特定事件序列（参见前一步骤的列表），将“构架机制”的特定应用场景在当前“Use Case 实现”的相关序列图当中扩展。

不同“构架机制”的引入过程各有特色，需要结合具体情况执行相应的步骤。

8.1.4 技巧：为“责任”提供上下文信息

注意，在“局部设计”任务中，“分析类”的“责任”转化为“设计类”的“责任”而不细化为具体的操作。这部分工作将推延到“细节设计”任务中处理。这样做，局部的设计内容仍然保持比较高的概括水平，有利于进行局部间协调与全局性整理。

进入“局部设计”任务后，参与协作的元素逐渐增多，用于描述“Use Case

[1] 如果一个“分析类”映射成一个“设计类”，“分析类”和“设计类”的命名统一，在图示上几乎看不出变化。此时，属于“分析类”的构造型对于“设计类”已无实际价值，可以去掉。

实现”的序列图逐渐复杂起来。“设计类”的某一“责任”可能在不同场（序列图）景中被引用（使用），对“责任”的描述通常不能完全脱离上下文环境。为了维持模型的易读性，在特定场景（序列图）中注释“责任”的目的与内容是一种好习惯。在后续的“细节设计”任务中，“责任”将被具体的操作实现，丰富的上下文信息能够避免走无谓的弯路，从而提升工作效率。

8.1.5 示例

用“核心设计元素”替换“分析类”

首先，在“参与类图”中用“设计类”和“子系统接口”替换原有的“分析类”，参见图 8-3。与“局部分析”的结果无显著差异。

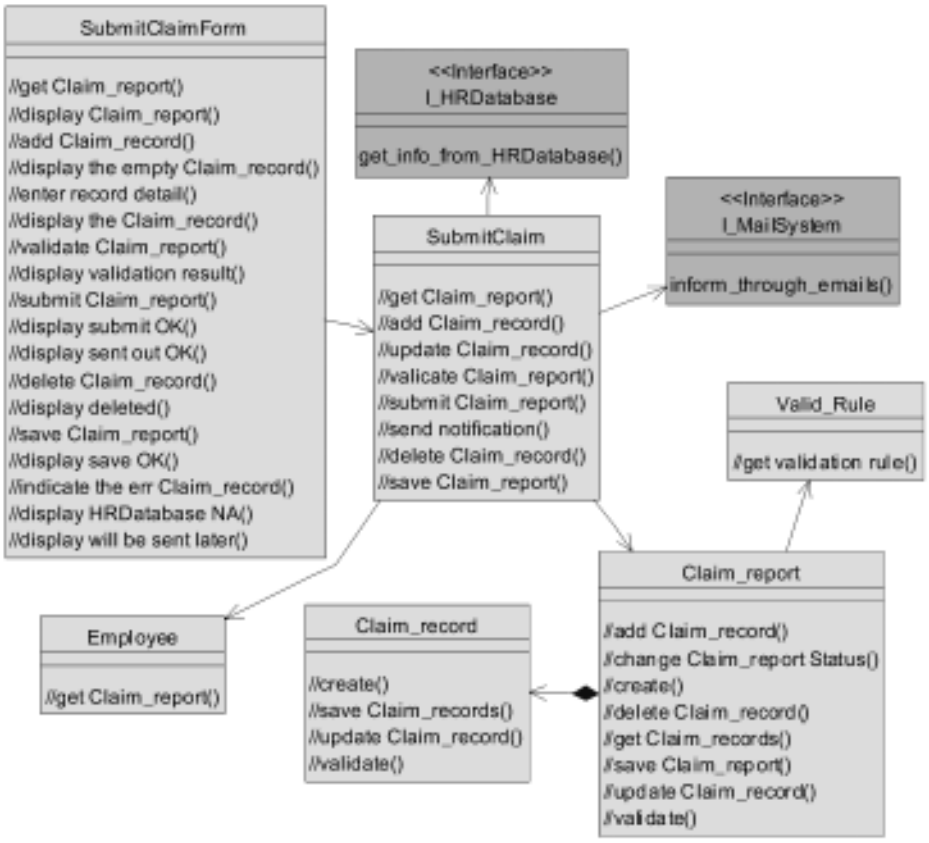


图 8-3　在“参与类图”中用“核心设计元素”替换“分析类”

在序列图中用“核心设计元素”的实例替换原有的“分析类”实例。以基本事件序列为例，参见图 8-4。在图中，被“子系统接口”替换的“分析类”的实例不向其他对象发送消息，可用“子系统接口”的“实例”代表某一特定子系统。

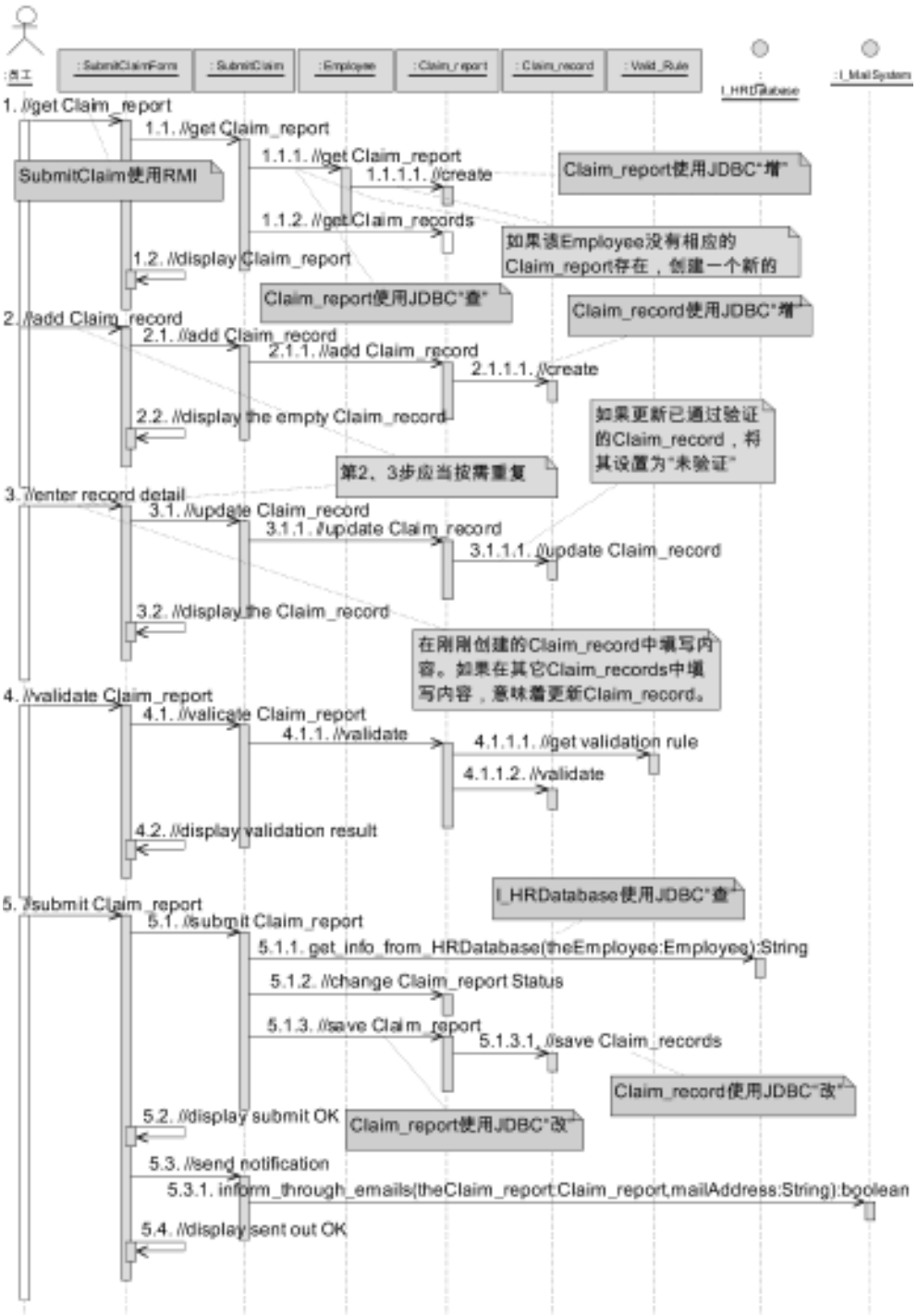


图 8-4　**在序列图中用“ 核心设计元素 ”替换“ 分析类 ”**

接下来，枚举“ 设计类 ”在特定事件序列中使用“ 构架机制 ”场景的情况，

参见表 8-2。实践中，可以在相关的序列图中标注“设计类”使用“构架机制”场景的具体情况，参见图 8-4 中的部分注释信息。

表 8-2 “设计类”使用“构架机制”场景的列表示例

“设计机制” / “设计类”	RDBMS-JDBC				RMI-Java 1.3	…	…
	增	删	查	改(存)	远程呼叫	…	…
Claim_record	B	A2	B	B，A4	—		
Claim_report	B(A1)	—	B	B，A4	—	…	…
SubmitClaim	—	—	—	—	B	…	…
…	…	…	…	…	…	…	…

注释：表中“B”表示基本事件序列，“A*”表示相应的备选事件序列。

落实“构架机制”的支撑作用

以表 8-2 中列举的三个“设计类”为例，展示如何落实“构架机制”的支撑作用。

表 8-3 给出了“设计类”Claim_record 应用“留存”机制时，构造型《role》所对应的具体“核心设计元素”与“衔接设计元素”。注意，“核心设计元素”已经存在，“衔接设计元素”需要创建。

表 8-3 “设计类”Claim_record 应用“留存”机制的《role》

RDBMS-JDBC 中的《role》	“核心设计元素”与“衔接设计元素”
《role》PersistentClass	Claim_record
《role》PersistencyClient	Claim_report
《role》PersistentClassList	Claim_recordList
《role》DBClass	DBClaim_record

图 8-5 展现 Claim_record 应用 RDBMS-JDBC 时所创建的“衔接设计元素”，即类 Claim_recordList 和类 DBClaim_record。它们与类 Claim_record 放在同一个包中。

参照“留存”机制的静态结构，图 8-6 给出了 Claim_record 应用该机制的静态结构。鉴于表 8-3 中列举的四个“设计类”处于同一个包 Claim Artifacts 中，并且在“优化组织结构”活动中已经建立了该包对包 java.sql 的依赖关系，因而没有增添包之间新的依赖关系。

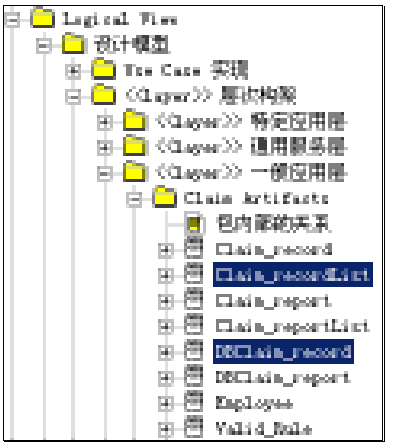


图 8-5　为 Claim_record 应用“留存”机制而创建的“衔接设计元素”

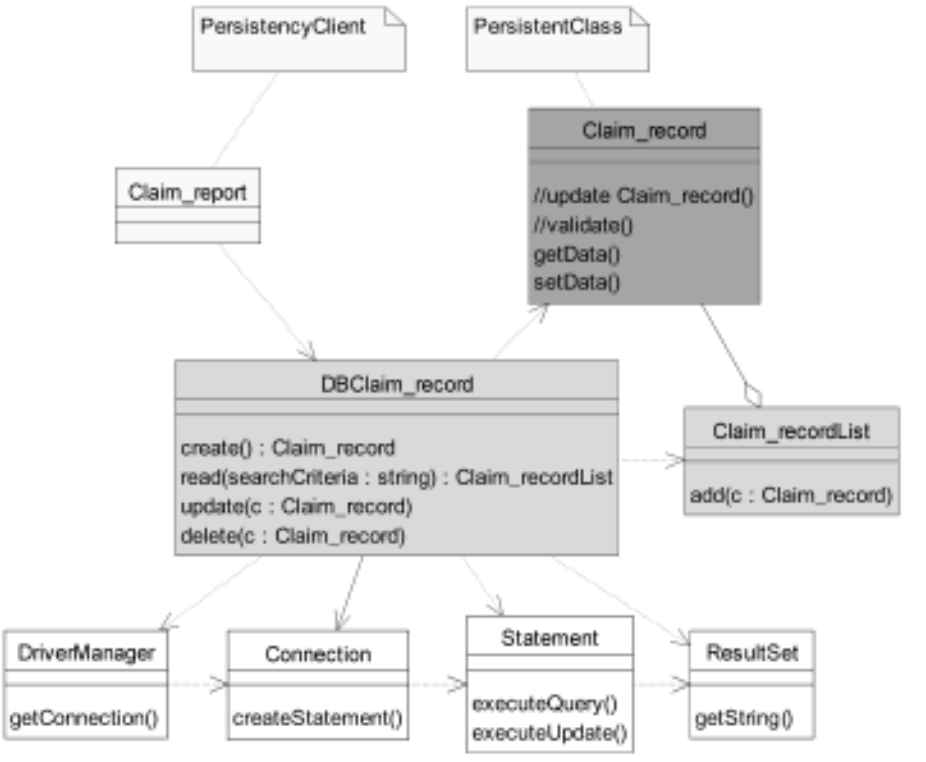


图 8-6　（单独）考虑“设计元素”Claim_record 使用 RDBMS-JDBC 的静态结构

图 8-7 给出了 Claim_record 使用 RDBMS-JDBC“增”场景的序列图，和基本事件序列中的消息 2.1.1 对应，参见图 8-4。鉴于基本事件序列已经比较复杂，故

不将这一部分融入其中，而是在相应消息上增加注释。

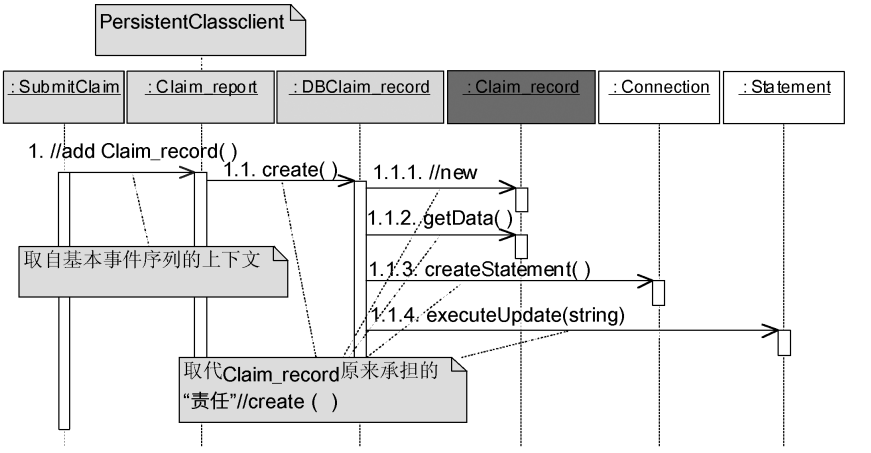


图 8-7　Claim_record 使用 RDBMS-JDBC 的“增”场景

图 8-8 给出了类 Claim_record 使用 RDBMS-JDBC“删”场景的序列图，被融入 A2 备选事件序列。注意，在这个序列图中，类 Claim_record 仅仅作为 DBClaim_record 一个操作的参数类型，即 delete（Claim_record)。

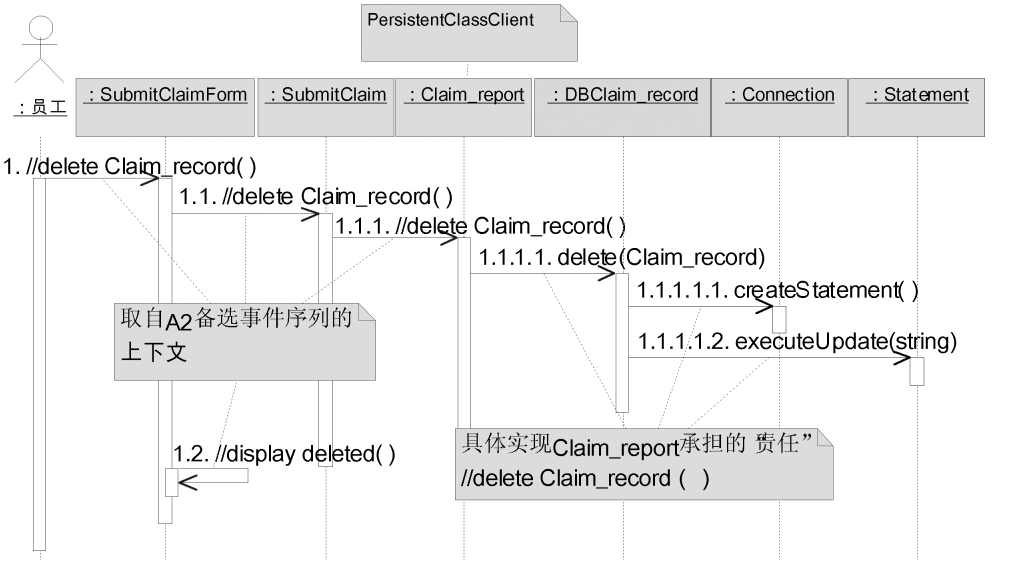


图 8-8　Claim_record 使用 RDBMS-JDBC 的“删”场景

图 8-9 给出了 Claim_record 使用 RDBMS-JDBC“查”场景的序列图，和基本

事件序列中的消息 1.1.2 对应，参见图 8-4。鉴于基本事件序列已经比较复杂，故不将这一部分融入其中，而是在相应消息上增加注释。

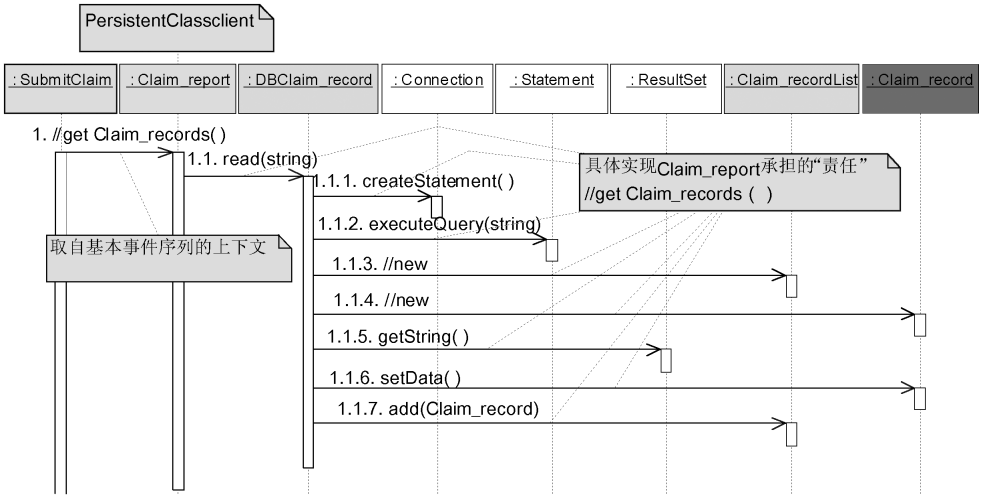


图 8-9　Claim_record 使用 RDBMS-JDBC 的“查”场景

图 8-10 给出了 Claim_record 在 A4 备选事件序列中使用 RDBMS-JDBC“改”场景的序列图。类 Claim_report 的“责任”//save Claim_report（）将重复至所有 Claim_record 在数据库得到更新为止。注意，类 Claim_record 在基本事件序列和 A4 备选事件序列中使用“留存”机制“改”场景的情形相同，因而只需说明其中的一个（A4 备选事件序列）即可。

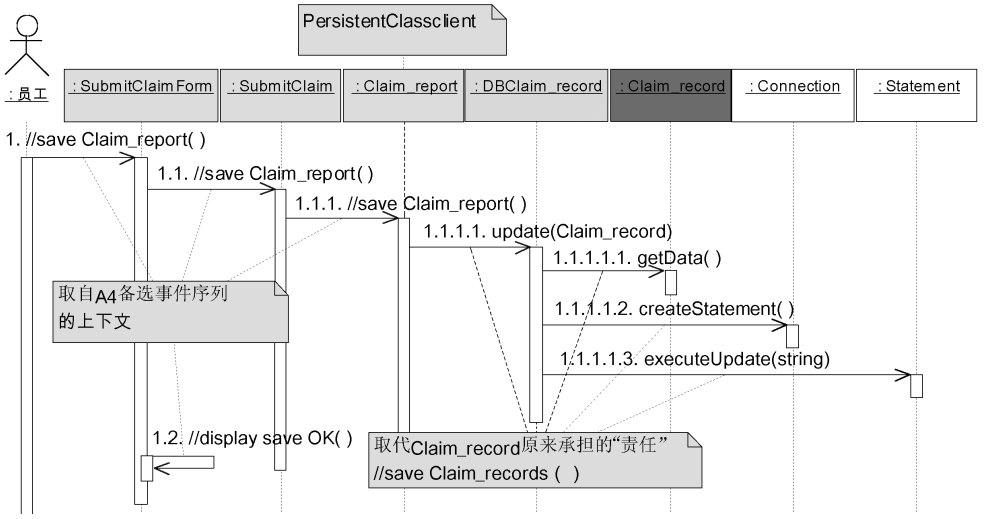


图 8-10　Claim_record 使用 RDBMS-JDBC 的“改”场景

表 8-4 给出了“设计类”Claim_report 应用“留存”机制时，构造型《role》所对应的具体“核心设计元素”与“衔接设计元素”。

表 8-4 “设计类”Claim_report 使用 RDBMS-JDBC 的《role》

RDBMS-JDBC 中的《role》	“核心设计元素”与“衔接设计元素”
PersistentClass	Claim_report
PersistencyClient	Employee（“增”和“查”），SubmitClaim （“改”）
PersistentClassList	Claim_reportList
DBClass	DBClaim_report

图 8-11 展现 Claim_report 应用 RDBMS-JDBC 时所创建的“衔接设计元素”，即类 Claim_reportList 和类 DBClaim_report，它们与类 Claim_report 放在同一个包中。

图 8-11 为 Claim_report 应用“留存”机制而创建的“衔接设计元素”

参照“留存”机制的静态结构，图 8-12 给出了 Claim_report 应用该机制的静态结构。鉴于类 SubmitClaim 所属包 Claim Activities 和类 Claim_report 所属包 Claim Artifacts 的依赖关系已经存在（参见“全局设计”中“优化组织结构”活动的示例），并且在引入“基础设计元素”时已经建立了包 Claim Artifacts 对包 java.sql 的依赖关系，因而没有增添包之间新的依赖关系。

图 8-13 给出了 Claim_report 使用 RDBMS-JDBC“增”场景的序列图，和基本事件序列中的消息 1.1.1 对应，参见图 8-4。鉴于基本事件序列已经比较复杂，故不将这一部分融入其中，而是在相应消息上增加注释。

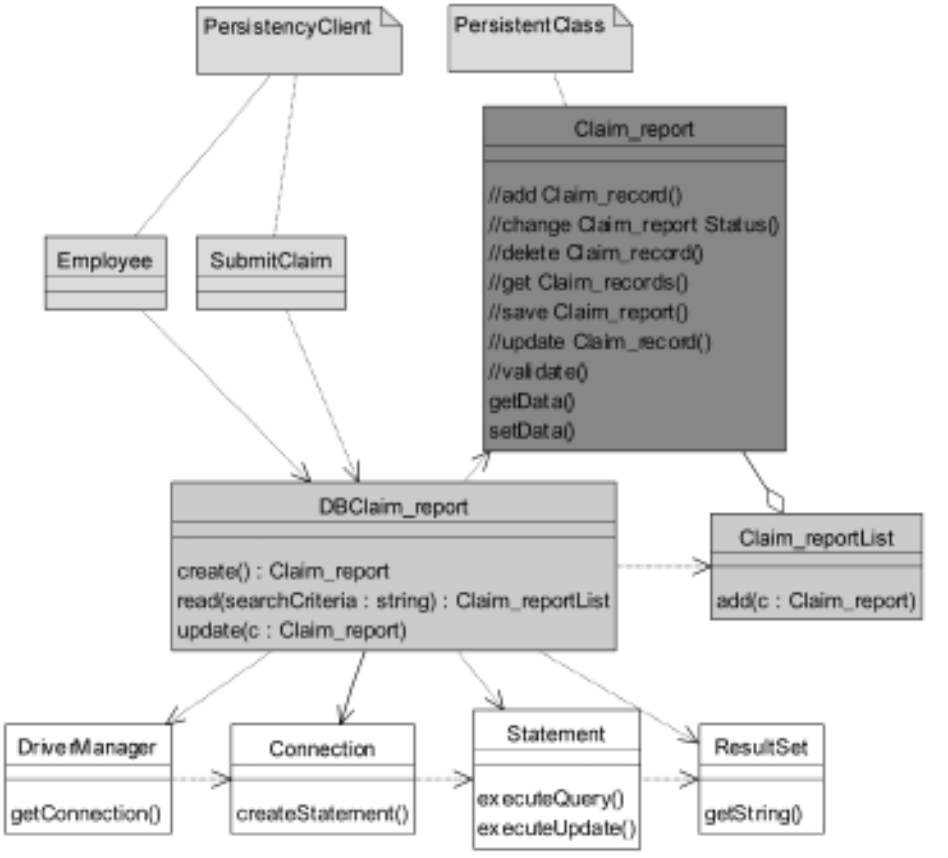


图 8-12　**（单独）考虑“设计元素”** Claim_report **使用** RDBMS-JDBC **的“参与类图”**

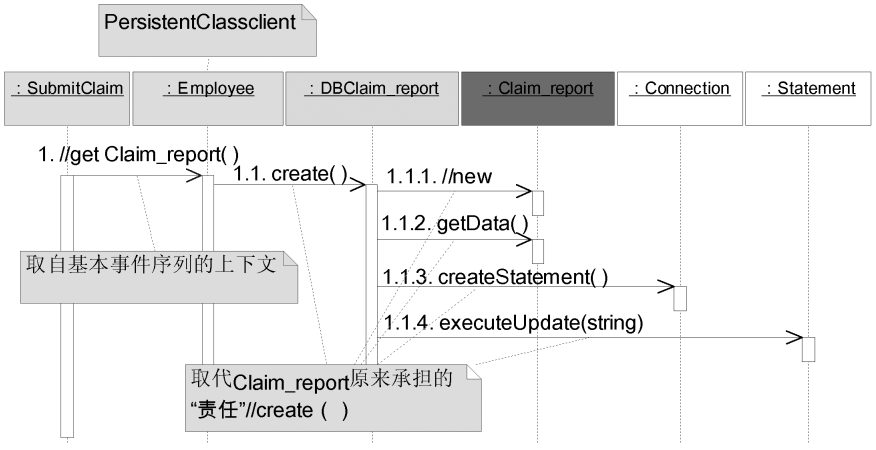


图 8-13　Claim_report **使用** RDBMS-JDBC **的“增”场景**

图 8-14 给出了 Claim_report 使用 RDBMS-JDBC“查”场景的序列图，和基本事件序列中的消息 1.1.1.1 对应，参见图 8-4。鉴于基本事件序列已经比较复杂，故不将这一部分融入其中，而是在相应消息上增加注释。

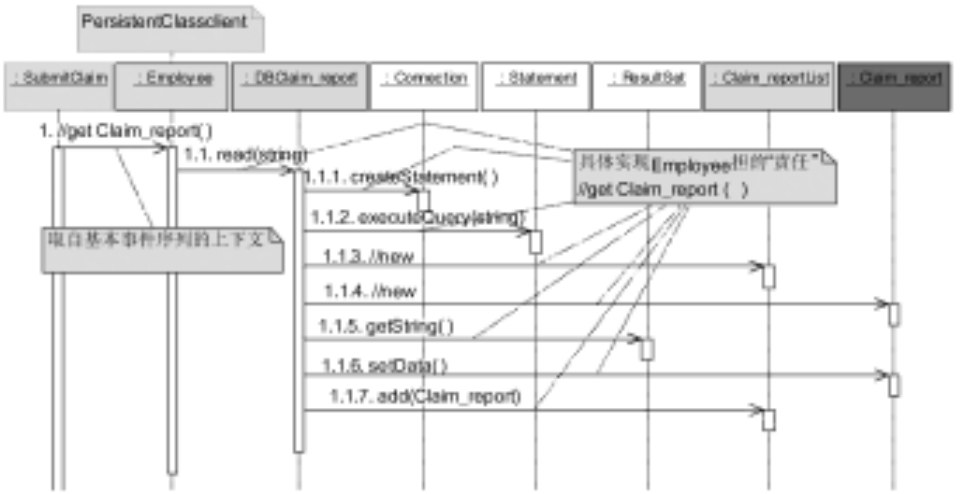


图 8-14　Claim_report 使用 RDBMS-JDBC 的“查”场景

图 8-15 给出了 Claim_report 在 A4 备选事件序列中使用 RDBMS-JDBC“改”场景的序列图。对比“局部分析”任务中的 A4 备选事件序列图，不难看出“构架机制”对化简分析任务的贡献。

图 8-15　Claim_report 使用 RDBMS-JDBC 的“改”场景

表 8-5 给出了“设计类”SubmitClaim 使用“分布处理”机制时，构造型《role》所对应的具体“核心设计元素”与“衔接设计元素”。

表 8-5 “设计类”SubmitClaim 使用 RMI-Java 1.3 的《role》

RMI-Java 1.3 中的《role》	“核心设计元素”与“衔接设计元素”
DistributedClass	SubmitClaim
DistributedClassClient	SubmitClaimForm
PassedData	Claim_report，Claim_record
IDistributedClass	ISubmitClaim

图 8-16 展现 SubmitClaim 应用 RMI-Java 1.3 时所创建的“衔接设计元素”，即接口“ISubmitClaim”，它与 SubmitClaim 放在同一个包中。

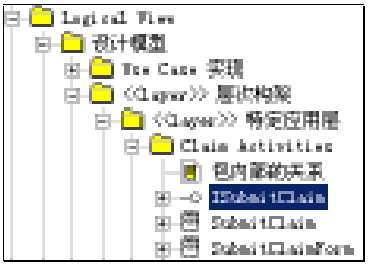


图 8-16 为 SubmitClaim 应用“分布处理”机制而创建的“衔接设计元素”

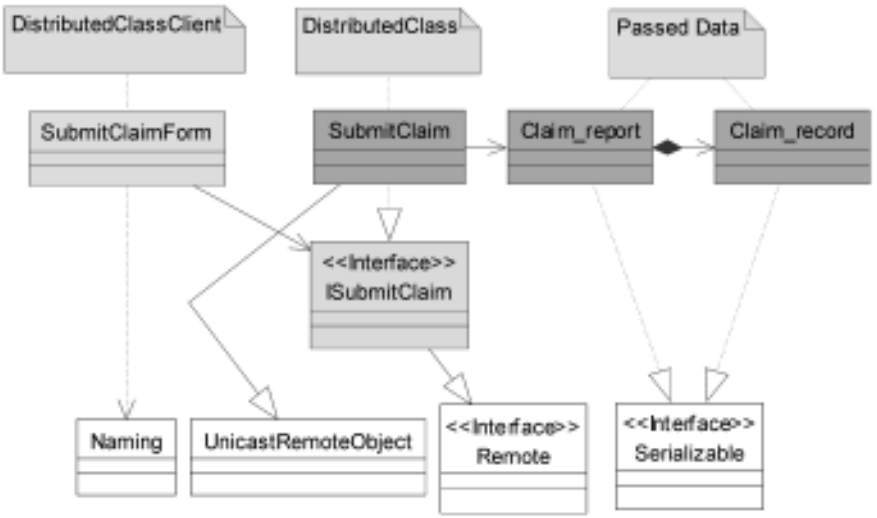


图 8-17 考虑“设计元素”SubmitClaim 使用 RMI-Java 1.3 机制的静态结构

参照“分布处理”机制的静态结构，图 8-17 给出了 SubmitClaim 使用该机制的静态结构。注意，SubmitClaimForm 指向 SubmitClaim 的单向关联关系被 SubmitClaimForm 指向 ISubmitClaim 的单向关联关系取代。经初步判断，“分布类” SubmitClaim 的操作将使用 Claim_report 与 Claim_record 的对象作为“异地”传送的参数。因而，作为《role》PassedData 的 Claim_report 和 Claim_record 需要实现（implement）Java 的接口 Serializable，相应的包 Claim Artifacts 需要依赖包 java.io，新添的包之间依赖关系，参见图 8-18。

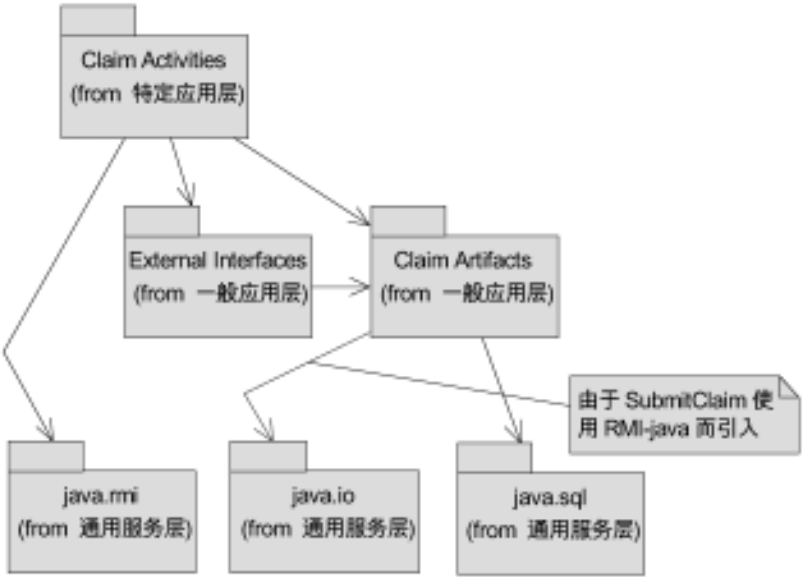


图 8-18 “设计类” SubmitClaim 使用 RMI-Java 1.3 而添加的包之间依赖关系

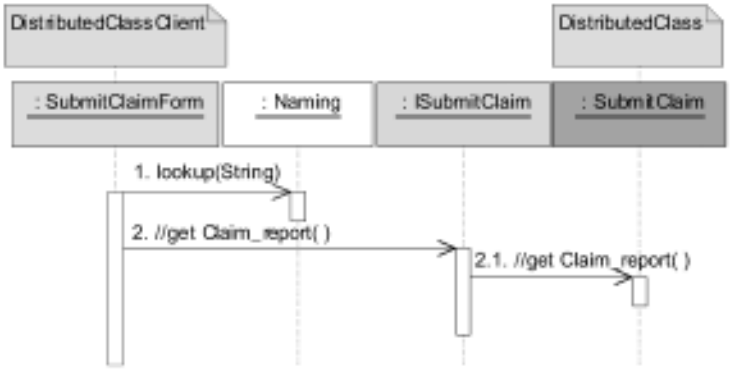


图 8-19 SubmitClaim 使用 RMI-Java 1.3 的时序

“设计类” SubmitClaim 使用 RMI-Java 1.3 实现“分布处理”之后，图 8-19 表达的时序关系将替换基本事件序列中的消息 1.1，消息 2.1、3.1、4.1 和 5.1 的情

况类似，但是不重复呼叫 Naming 的 lookup（）操作。在实际运行的系统中，：ISubmitClaim 和：SubmitClaim 所承担的任务将由（利用 rmic 自动生成的）Remote Stub 和 Remote Skeleton 完成。

图 8-20 展示引入“ 构架机制 ”过程中添加的序列图，它们归属于相应的“ Use Case 实现 ”。至此，分析活动中使用的简略“ 参与类图 ”已经没有实际价值。

引入各种“ 设计元素 ”之后，详细的“ 参与类图 ”内容变得更加具体，参见图 8-23。其中，I_MailSystem 对 Employee 的依赖关系和 I_HRDatabase 对 Claim_report 的依赖关系来自于“ 全局设计 ”中“ 确定核心元素 ”活动。

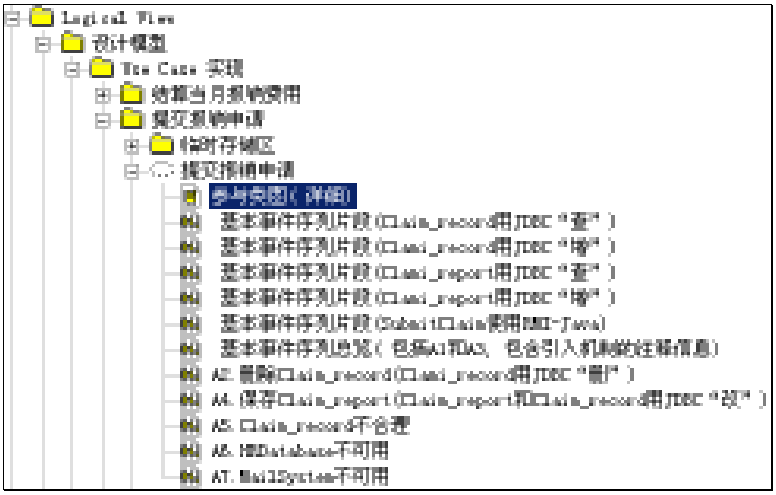


图 8-20　引入“ 构架机制 ”后更新的“ Use Case 实现 ”内容

引入“ 构架机制 ”过程中，进一步充实了层次构架中的内容，同时添加了一些“ 包内部的关系 ”，参见图 8-21 与图 8-22。

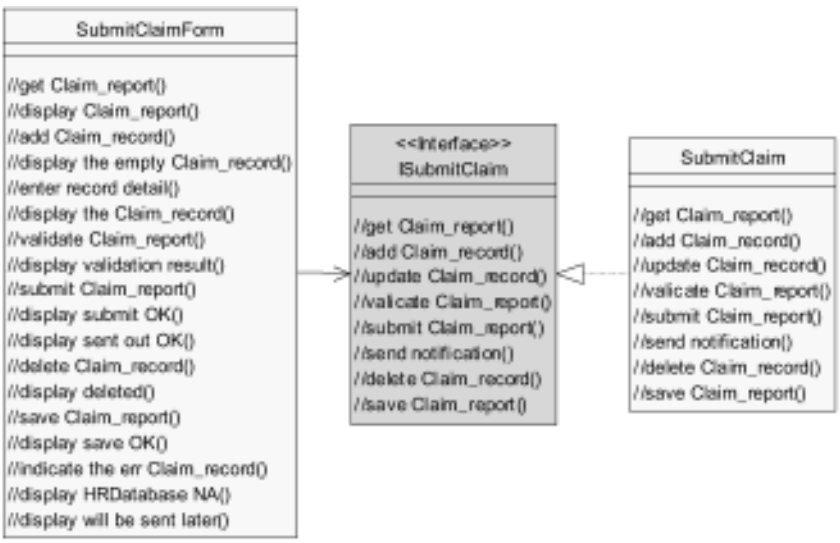


图 8-21　Claim Activities **包内部的关系**

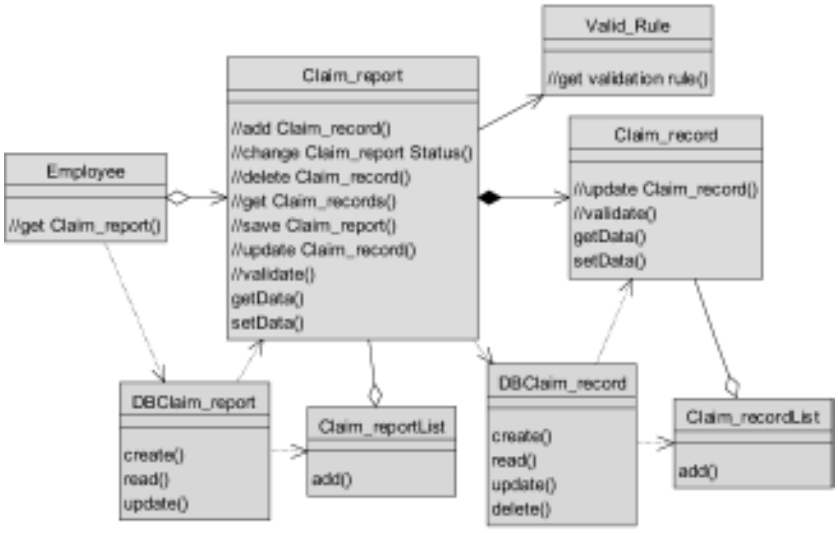


图 8-22　Claim Artifacts **包内部的关系**

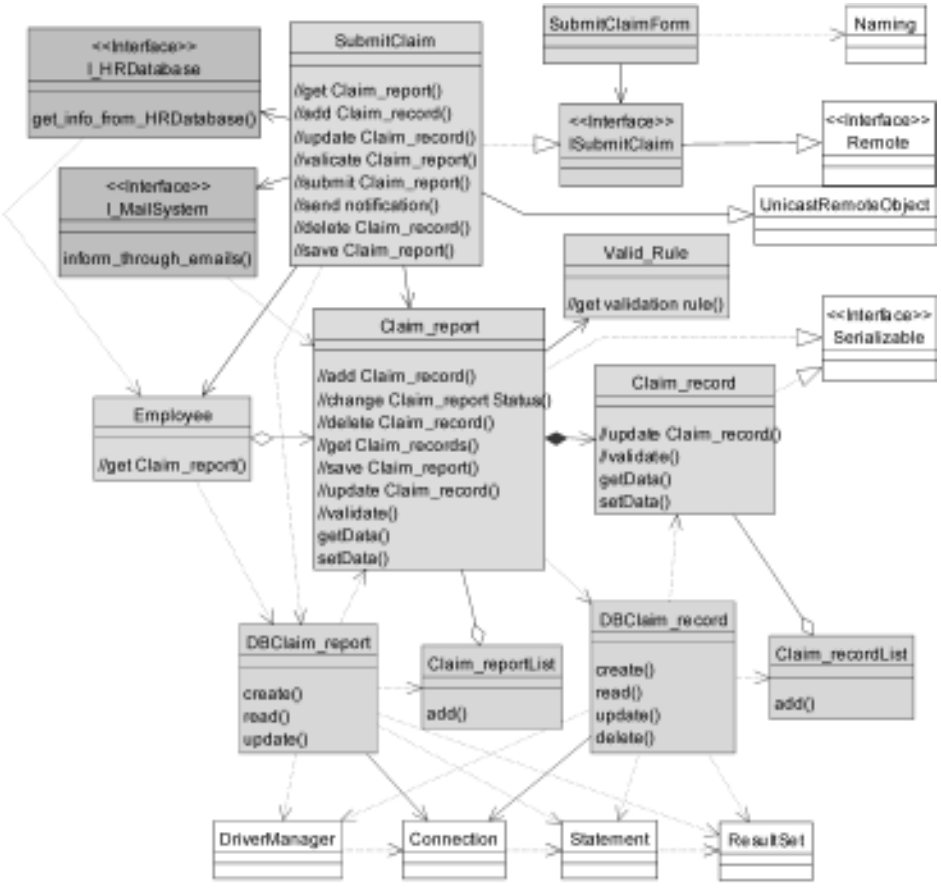


图 8-23　引入“ 留存 ”和“ 分布处理 ”两种机制后更新的“ 参与类图 ”

8.2　实现子系统接口

“ 实现子系统接口 ”活动的依据是“ 子系统接口 ”中定义的行为，该活动的结果是实现“ 子系统接口 ”要求的一组“ 设计元素 ”以及相关的静态和动态描述，参见图 8-24。

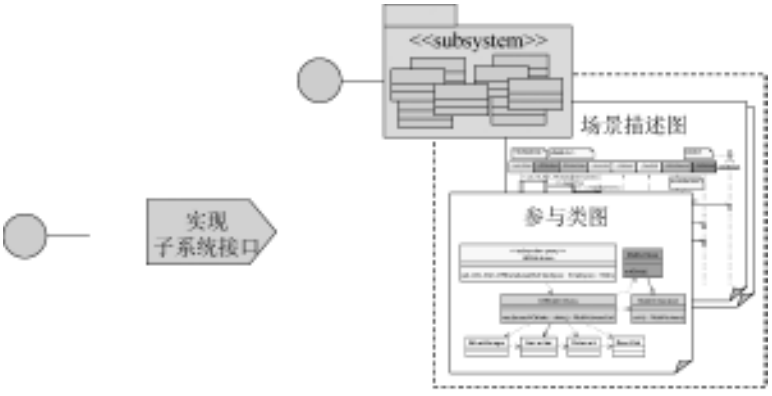


图 8-24　“ 实现子系统接口 ”活动图示

8.2.1　概念：“ 小型的 Use Case ”

特定子系统的外部可见行为由它所对应的“ 子系统接口 ”定义，“ 子系统接口 ”就是对特定子系统提出的行为需求。鉴于“ 子系统接口 ”在概念上具有良好的封装性，可以将“ 子系统的接口 ”提出的行为需求近似看作“ 小型的 Use Case ”，即所谓的“ 局部 ”需求。前面介绍的“ 局部分析 ”与“ 局部设计 ”概念及方法仍旧适用。由于范围缩小，通常直接在设计层面展开具体的工作。

8.2.2　步骤 1：实现“ 子系统接口 ”定义的行为

首先，确定参与实现子系统行为的“ 设计元素 ”。这些“ 设计元素 ”可以是新建的“ 设计元素 ”，它们将被放置在该子系统所对应的包内；这些“ 设计元素 ”也可以来自已经存在的“ 设计元素 ”。

然后，描述子系统的动态场景和静态结构。用序列图描述子系统动态行为的典型场景。类似“ 局部分析 ”任务中的概念，在描述场景的过程中，挖掘“ 设计

元素”所需的操作和属性。用“参与类图”描述子系统的静态结构，说明实现该子系统的“设计元素”间关系。

8.2.3 步骤 2：明确子系统与其外部设计元素的关系

如果参与实现子系统的“设计元素”全部新建于该子系统对应的包中，那么该子系统对其他设计元素的依赖关系等同于相应“子系统接口”对其他设计元素的依赖关系。否则，需要进一步说明该子系统对其他设计元素的依赖关系。例如，参与实现子系统 S 的“设计元素”A 位于子系统 S 以外的包 P 中，那么子系统 S 将依赖于包 P。

8.2.4 技巧：提前实现“子系统接口”

概念上，用特定子系统实现一个“子系统接口”的工作与系统“其他局部”的设计工作可以很好地解耦。“其他局部”可以是用于实现另一“子系统接口”的子系统或者是设计意义上的“Use Case 实现”。鉴于这种弱耦合关系，在明确定义“子系统接口”之后就有条件展开“实现子系统接口”活动。如果不是采用已有构件实现“子系统接口”，通常等到“优化组织结构”活动之后启动“实现子系统接口”活动，这样可以减少包之间依赖关系的反复调整。不难看出，“实现需求场景”活动和“实现子系统接口”活动可以并行推进。

8.2.5 技巧：确保子系统的独立性

原则上，尽量避免子系统包含的内容直接依赖于其外部的“设计元素”，以确保子系统的独立性（Independency）。“直接依赖”是指被依赖的“设计元素”参与实现子系统行为的协作（它的实例参加子系统动态场景的交互）。

尽力避免子系统直接依赖于自身可能产生变化的“设计元素”。“核心设计元素”与应用逻辑直接相关，是最易于引入变化的“设计元素”。

作为“基础设计元素”（参见“全局设计”任务中的相关概念）的设计内容非常稳定，对它们的依赖基本不会影响子系统的的独立性。

此外，由于子系统中《subsystem proxy》的公开操作在形式上与“子系统接口”一致，“子系统接口”对其他“设计元素”的依赖关系转嫁到《subsystem proxy》。这些“设计元素”通常作为“子系统接口”操作的参数或返回值类型，它们与子系统内部“设计元素”之间存在非结构化的关系（参考“细节设计”中相关内容）。

8.2.6 技巧：不同子系统之间的依赖关系

如果一个子系统 A 中使用另一子系统 B 所提供的行为，那么两个子系统之间需要建立依赖关系（A 依赖 B）。

“子系统接口”的定义明确地声明子系统的所有外在可见行为。原则上，子

系统之间的依赖关系应该建立在相应的“子系统接口”之上，而不是由一个子系统直接依赖另一个子系统。具体讲，避免使用子系统的非外在可见行为，即未出现在“子系统接口”定义中的操作。否则，将影响设计工作的灵活性，使“子系统接口”的解耦作用失去实际意义。

8.2.7 示例

以子系统 HRDatabase 为例，作简要说明。用子系统 HRDatabase 实现“子系统接口”I_HRDatabase 过程中，应用了“留存”机制。注意，虽然都是应用“留存”机制，前一节示例中所访问的是系统边界以内的数据库系统（拟建数据库），本节示例中所访问的是拟建系统边界之外的数据库系统（已有数据库）。

图 8-25 给出实现子系统 HRDatabase 的“设计元素”之“参与类图”。图 8-26 用序列图展示子系统的动态行为。

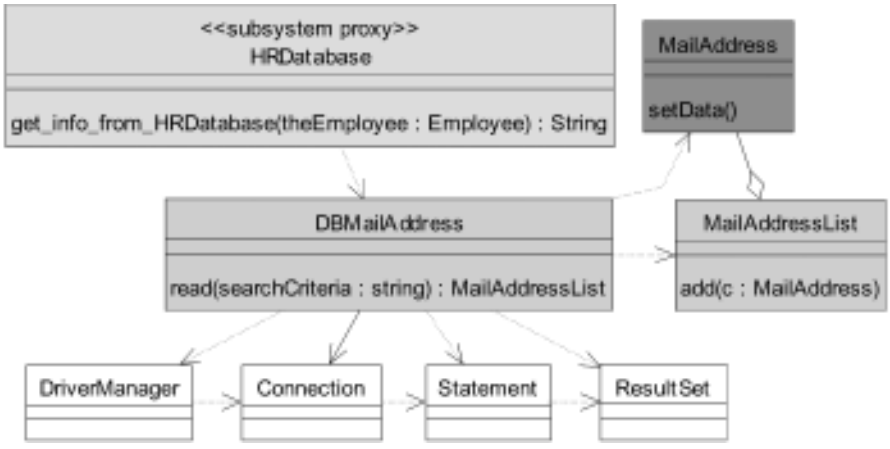


图 8-25　HRDatabase 子系统的“参与类图”

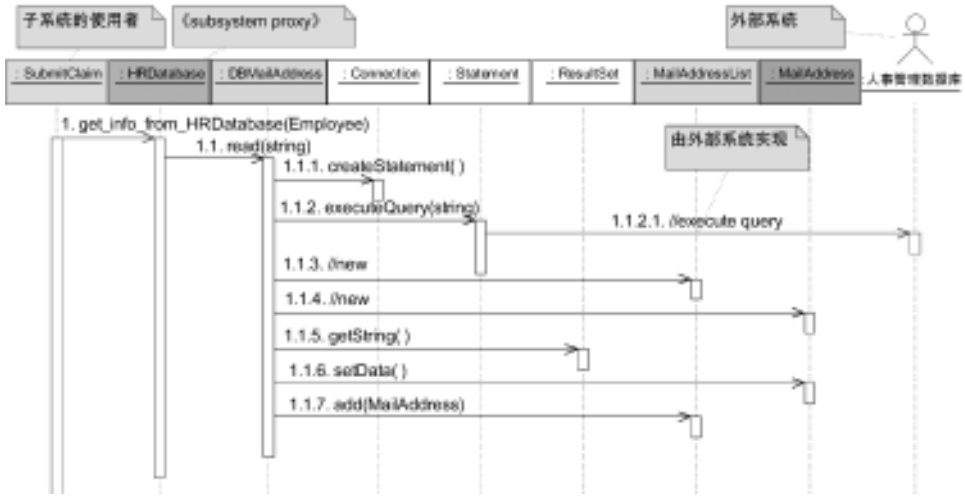


图 8-26　HRDatabase 子系统动态行为的序列图

图 8-27 展示该子系统对外部的依赖关系，其依据来自于子系统的“参与类图”以及相关“子系统接口”对其他“设计元素”的依赖关系。经过“局部分析”任务中的两个活动，层次构架中包之间的关系进一步被丰富，参见图 8-28。

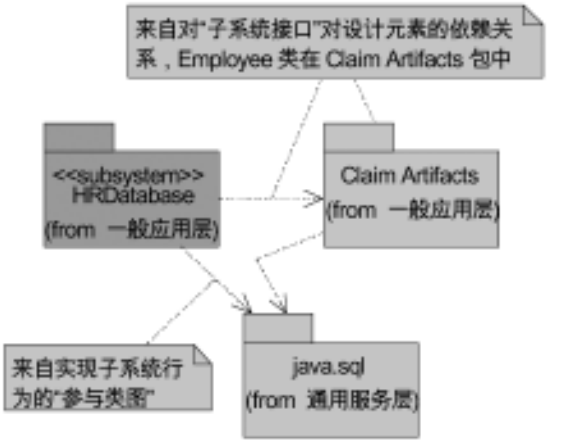


图 8-27　HRDatabase 对外部的依赖关系

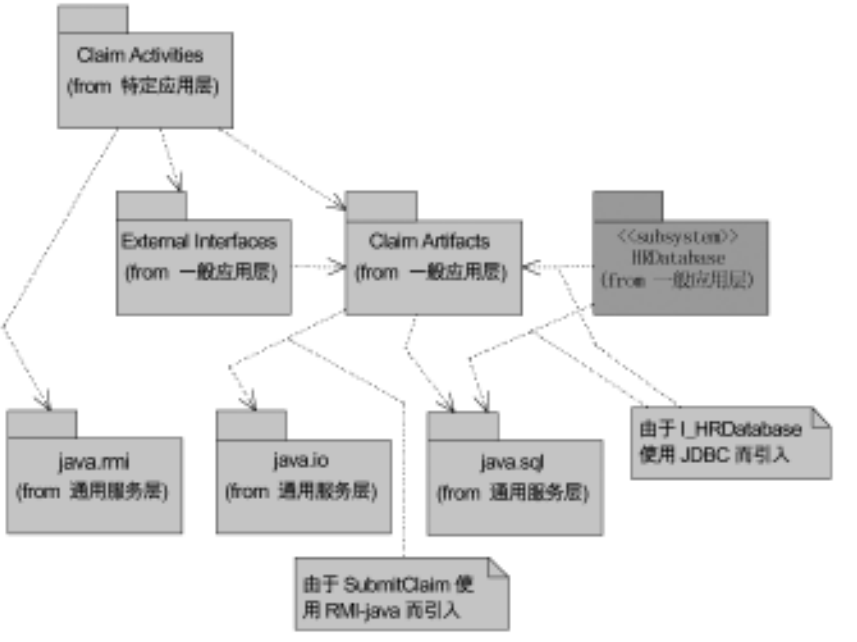


图 8-28　进一步丰富的包之间依赖关系

图 8-29 展示出整个“局部分析”任务对层次构架内容的充实结果。

图 8-29 “局部设计”任务对构架内容和关系的充实

针对示例，表 8-6 给出了局部设计任务中的积累的设计模型内容。

表 8-6 局部设计任务中积累的设计模型内容汇总

任　务	活　动	设计模型内容		
		“Use Case 实现”	层次构架	“构架机制”
局部设计	实现需求场景	图 8-3，图 8-4，图 8-3～图 8-4，图 8-3～图 8-4，图 8-3～图 8-4	图 8-5，图 8-11，图 8-16，图 8-18，图 8-22，图 8-23	

	实现子系统接口		**图** 8-25**～图** 8-29	

第 9 章　细 节 设 计

“细节设计”任务在细节上确保设计方案能够为后续实施活动提供明确的依据。类自身的定义以及相互的关系是“细节设计”任务所针对的具体内容。

在“细节设计”任务中，有不同侧重的两项活动，参见图 9-1。

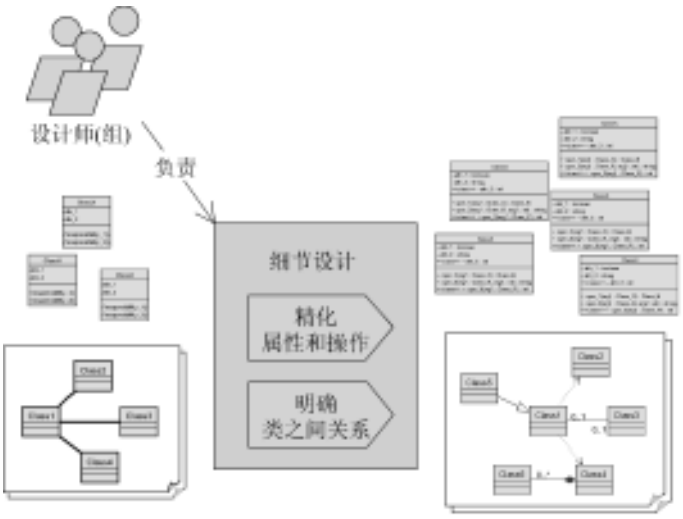


图 9-1　“细节设计”任务的责任人 — 依据 — 活动 — 结果

- 精化属性和操作。明确定义操作的参数和基本的实现逻辑，明确定义属性的类型和用途。
- 明确类之间关系。根据对象之间通信的实际情况，明确类之间的关系。

“细节设计”任务的灵活性较高，但对整个体系构架的影响并不显著。

9.1　精化属性和操作

“精化属性和操作”活动的基本依据是“局部设计”得到的“设计类”，该活动的结果是类的属性和操作被精化，参见图 9-2。

图 9-2 “精化属性和操作”活动图示

9.1.1 概念：需要精化的类

事实上，并不是所有的类都需要进行精化，工作的重点应针对“核心设计元素”。在“确定核心元素”活动中，已经精化了“子系统接口”中的操作，它们通常能覆盖那些隔离外部系统的“设计元素”。鉴于用户界面通常可以用直观化的工具构造，手工精化相应“设计元素”的必要性不强。需要精化的内容主要是由控制类和实体类演化而来的类。

9.1.2 概念：操作（Operation）

对象 x 访问或影响其他对象 y 的惟一途径就是对象 y 的操作。对象的行为在其操作被调用时表现出来。操作能影响对象的属性以及同其他对象“连接”的具体含义，操作还能引发执行其他相关操作。操作被明确的标识（Signature）封装，由具体的方法（Method）实现。操作定义的主要内容如下。

- 操作名称。通常是词组，遵循程序设计语言和项目统一的命名规则。
- 返回类型。建议用对象作为操作的返回值。
- 参数。包括参数的名称、类型和缺省取值[□]。
- 简短说明。从操作使用者（类）的角度，指明该操作要达到的目的。

9.1.3 概念：属性（Attribute）

属性是对象的一种冠名特征（Named Property）。属性为对象提供了保留信息的空间，用于记录对象的内容和状态。

对于每项属性，主要定义下列内容。

- 属性名称。通常是名词，遵循程序设计语言和项目统一的命名规则。
- 属性类型。来自程序设计语言支持的基本数据类型。
- 属性缺省值。创建类的实例时，用于初始化新建的对象。
- 简短说明。说明属性的含义和用途。

□ 有时需要说明参数用于传递“值”还是“引用”（by value or by reference），操作是否改动“引用”的内容。

9.1.4 概念：操作和属性的可见度（Visibility）

“可见”的含义对应一个主体和一个客体，主体可以看见客体的程度即客体的可见度。操作（或属性）的可见度有几种具体的类型，呈现出越来越“隐蔽”的特征。

- “公开”(Public)。操作（或属性）对所有的类可见。
- “受保护”(Protected)。操作（或属性）对类本身、其子类或友类[1]可见。
- “私有”(Private)。操作（或属性）对类本身和其友类可见。
- “实施”(Implementation)。操作（或属性）只对类本身可见。

9.1.5 概念：类的可见度

类本身也有可见度概念，具体含义相对于该类所在的包，有两种情形。

- “公开”表示类可由它所属包之外的类引用。
- “私有”表示类只能由同一包内的类引用。

当类的可见度为“公开”时，这个类的操作（或属性）才有可能选择任意一种可见度。缺省情况下，类的可见度为“公开”。“私有”类“断绝”与“外界”的关系，有助于提升包的封装性。

9.1.6 概念：操作和属性的适用范围（Scope）

广义上[2]，操作和属性有两种类型的适用范围。

- “属于对象”。某一操作（或属性）属于类的各个实例。
- “属于类”。某一操作（或属性）为该类的全部实例共有。

假如一个类的所有操作和属性[3]的适用范围都是“属于类”，那么这个类的多个实例之间将没有差异。

9.1.7 步骤 1：明确操作的定义

首先，找出满足基本逻辑要求的操作。操作的原始依据是类的“责任”及其相关的上下文信息。类承担的某个“责任”往往来自于多个“Use Case 实现”。

其次，补充必要的辅助操作。“Use Case 实现”中的内容并不能覆盖拟建系统的全部行为，通常还要解决一些典型问题的操作：例如初始化（类的）实例、验证（类的）两个实例是否等同、创建（类的）实例的副本等。

接下来，给出清晰的表述。包括操作的名称、参数、返回值、“可见度”、“适

---

[1] 取决于特定的程序设计语言。

[2] pp.2-80 OMG Unified Modeling Language Specification

[3] 更准确地讲，应该是“一个类的全部特征”，因为类属性和操作通常只是类的全部特征中的一部分。

用范围”以及简短文字说明。注意遵从程序设计语言的命名规则。

在必要的情况下，可以简要说明操作的内部实施逻辑，即如何实现具体的方法（Method）。

9.1.8 步骤 2：明确属性的定义

“设计类”属性的原始依据是“分析类”的属性，在“细节设计”任务中，具体说明属性的名称、类型、缺省值、“可见度”、“适用范围”以及简短文字说明。

9.1.9 技巧：应用状态图获得操作和属性

对于那些具有明显状态特征的类而言，通过状态图可以识别出相关的操作和属性。通常在状态图中，某一状态内的活动（Activity）、进出某一状态的原子化动作（Action）以及状态之间的转移（Transition）都为发掘操作提供了依据。一个特定的状态，通常可以用属性取值的组合表示，换言之，状态为发掘属性提供了线索。通常情况下，多数类并没有显著的状态特征。

9.1.10 技巧：“导出属性”的使用价值

识别与确定属性的原则是该属性有用，并且高效率地被使用。否则，将导致系统存储空间与性能的额外开销。

有时，某项属性可能是根据其他属性计算而来，称为“导出属性”（Derived Attribute）。预先算出的“导出属性”可以节省运行时的运算能力开销，当系统性能要求较高时，能带来显著的益处。但是，会耗费较多的记忆资源（Memory）。形象地讲，这种做法是用空间换时间。

9.1.11 技巧：操作命名的注意事项

操作的名称应该明确反映其目的与结果，并且遵循程序设计语言的语法。

属于不同的类但概念相同的操作，应该具有相同的名称，即便它们可能具有不同的参数列表或实施方法。这些操作应当返回同一类型的结果，这样有利于操作的使用者（类）获得“多态”带来的益处，即不同对象以类似方式响应相似的消息。

避免在操作名称中暗示其执行方法。例如 Employee.bonus（）优于 Employee.calculateBonus（），因为后者暗示了要执行的计算操作，而实际上可能只是提供数据库中的内容。

9.1.12 技巧：说明操作的实现逻辑

如果将操作的定义比作“黑盒”，那么方法（Method）就是相应的“白盒”内容。方法是对操作的实现，由具体的程序设计语言完成。方法说明了实现操作

的具体方式，其内容通常会涉及操作的参数、类的属性及关系在方法中的使用。如果方法的内容需要采用特定的算法（Algorithm），应给出相应的文字或图示说明，比较常用的是活动图、序列图和状态图。

9.1.13 技巧：可见度的判断

操作和属性的缺省可见度为“受保护”（Protected），这种可见度能在一定程度上限制不必要的耦合。实践中，需要结合具体情况选取恰当的可见度。尽量少用“公开”可见度，作为通用工具（Utilities）的类除外。“私有”可见度用于避免子类与父类之间的耦合。“实施”可见度具有最严格的访问限制，对封装性的贡献显著，适用面较广。

9.1.14 示例

以类 Claim_report 为例，简要说明“精化操作和属性”的结果与影响。

SubmitClaim 与 Claim_report 处于不同的两个包中，SubmitClaim 到 Claim_report 之间存在单向的关联关系。SubmitClaim 的实例能够访问 Claim_report 的实例，因而类 Claim_report 的可见度为“公开”。

为了帮助读者直观理解类的演化，图 9-3 给出了“局部分析”任务中得到的“分析类”Claim_report，图 9-4 展现初步精化后的“设计类” Claim_report。

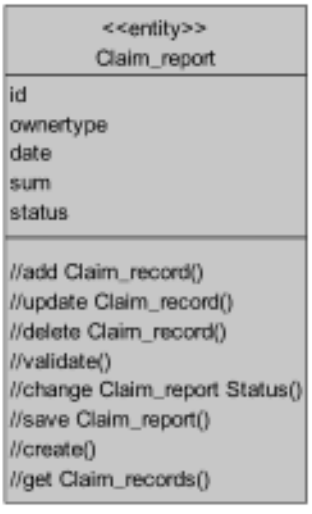


图 9-3 “局部分析”任务中的“分析类”Claim_report

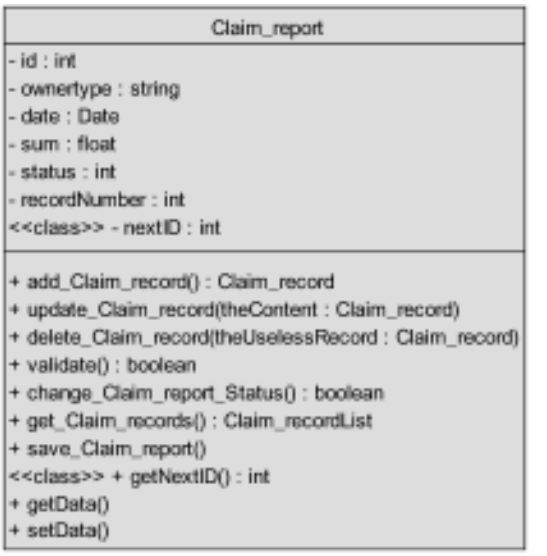


图 9-4　**初步“精化操作和属性”之后的“设计类”** Claim_report

以下简要说明那些新添的以及内容有显著变化的操作。注意，此时操作的“可见度”均为“公开”。

操作 save_Claim_report（）**的具体含义是保存与** Claim_report **相应的全部** Claim_record**。鉴于引入“留存”机制，** Claim_report **自身信息的保存由** DBClaim_report **代办。**

getData（）**和** setData（）**是在应用“留存”机制时引入的操作。注意，**getData（）**和** setData（）**概括地标记那些用于获取和设定属性的操作（**getter **和** setter**），其具体内容将针对每一项需要“留存”的属性，这部分细节将推迟到实施活动中落实。**

getNextID（）**是“属于类”的操作，其返回值在建立新的** Claim_report **时被（对象的** Constructor**）使用，相应的时序逻辑参见图** 9-5**。**

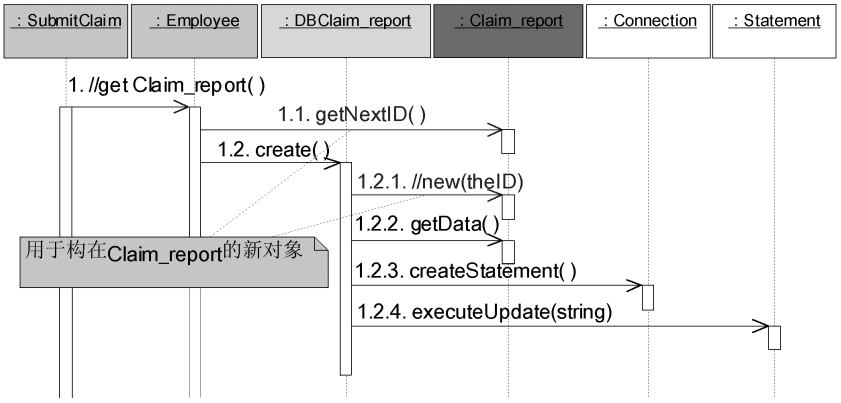


图 9-5　创建一个新的 Claim_report 的序列图

nextID 是“属于类”的属性，用于记录下一个新增 Claim_report 的序号。

属性 recordNumber 用于记录某一 Claim_report 当前包含的 Claim_record 的个数。在创建新的 Claim_record 时，该属性用作相应 Claim_record 的序号，该序号在当前的 Claim_report 中惟一。

以上的讨论仅仅针对“提交报销申请”的“Use Case 实现”内容。如果站在整体视角，Claim_report 具有比较丰富的状态特征，能够帮助设计人员挖掘更多的属性和操作，参见表 9-6。

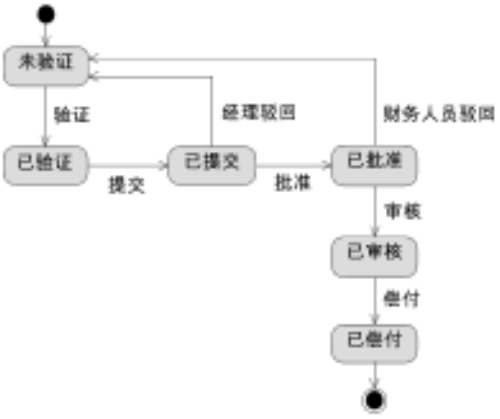


图 9-6　类 Claim_report 的状态转换图

表 9-1 和表 9-2 展示对状态转换图内容的具体应用。其中，状态对应现有和新添的属性及相关取值，状态的转移对应现有和新添的操作。

表 9-1　由状态图内容获得更多的属性

状　　态（State）	相关属性及取值	现有的／新添的
未验证	status=0	现有的
已验证	status=1	现有的
已提交	status=2	现有的
已批准	status=3； approver 不为空	现有的 新添的
已审核	status=4； reviewer 不为空	现有的 新添的
已偿付	status=5；	现有的

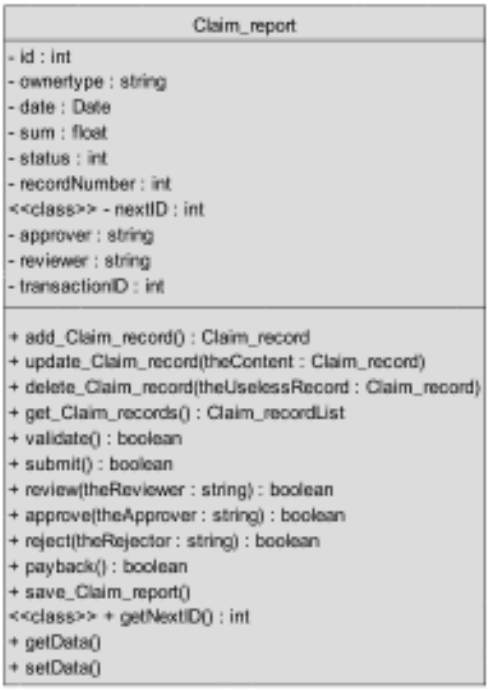
	transactionID 不为空	新添的

表 9-2　由状态图内容获得更多的操作

转移（Transition）	相 关 操 作	现有的／新添的
验证	validate（）	现有的
提交	change_Claim_report_Status（）	现有的
批准	approve（）	现有的
审核	review（）	新添的
偿付	payback（）	新添的
经理驳回	reject（）	新添的
财务人员驳回	reject（）	新添的

不难看出，操作 change_Claim_report_Status（）的命名并不理想，因为当初的着眼点在局部，属于正常现象。经过对该类状态特征的讨论之后，需要作出及时的调整，将其名称更换为 submit（）。

图 9-7 展示类 Claim_report 在现阶段的设计内容。图 9-8 展示[□]与 Claim_report 状态特征有关的属性和操作。



□ 如果使用 Rose，在类的上下文菜单中（右键），通过 Options－Select Compartment Items 实现。

图 9-7 “精化属性和操作”活动后的类 Claim_report

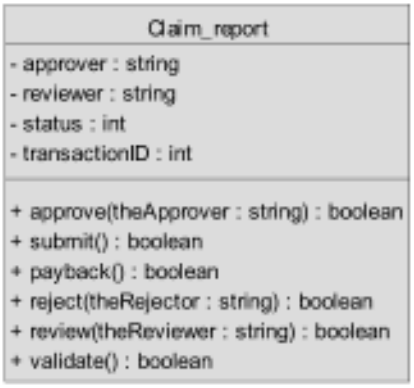


图 9-8 与 Claim_report 状态变化有关的属性和操作

9.2 明确类之间关系

“明确类之间关系”活动的基本依据是分析任务中初步得到的关联关系，它们来自于不同局部的“参与类图”，包括“Use Case 实现”与“子系统接口”的实现。该活动的结果是进一步明确的关系，主要是依赖关系（Dependency）和关联关系（Association），此外，还可以根据类之间的共性和差异，利用泛化关系对整体结构进行优化。参见图 9-9。

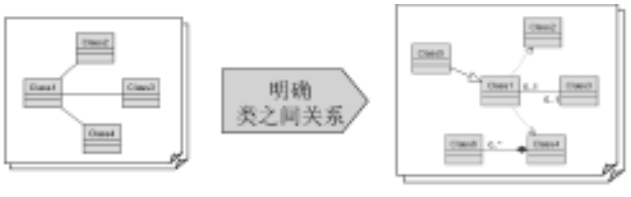


图 9-9 “明确类之间关系”活动图示

9.2.1 概念：对象间通信的“连接可见度”（Link Visibility）

在分析任务中，假设所有的关系都是结构化关系，即用关联关系及其强化形式笼统地表述两个类的对象之间存在（用于通信的）“连接”。实际运行的系统中，对象之间的“连接”有几种不同的情形，并不意味着必然存在结构化的关系。进

入设计任务之后，随着类的设计内容逐步明朗，有条件进一步确认对象间究竟需要何种类型的“连接”，从而明确类之间的关系。

类 A 的对象 a 与类 B 的对象 b 通信，a 能够向 b 发送消息的必要条件是 a 能够“引用”b，其概念在协作图中表现为 a 到 b 的“连接”。在面向对象软件系统中，a 可以通过四种方式“引用”b，对应于（从 a 到 b 的）四种类型的“连接可见度”。

- “全局”（Global）。b 是可以在全局范围内直接“引用”的对象。
- “参数”（Parameter）。b 作为 a 的某一项操作的参数或者返回值。
- “局部”（Local）。b 在 a 的某一操作中充当临时变量。
- “域”（Field）。b 作为 a 的数据成员（Data Member）。

前三种类型“连接可见度”具有暂时性，b 和 a 之间的“连接”仅在执行某个操作的过程中被建立（而后解除）。在静态结构中，这三种类型的“连接”被建模为类 A 对类 B 的依赖关系。最后一种类型的“连接可见度”具有稳定性。在静态结构中，这种“连接”被建模为类 A 到类 B 的关联关系及其强化形式（聚合或组合）。

9.2.2 概念：关联关系的细节内容

关联关系是一种结构化的关系，在后续实施活动中，其内容将作为类定义的组成部分。确认类之间存在关联关系之后，有条件进一步明确或改进其细节内容。

其一，是否存在关联关系的强化形式，即聚合甚至组合。如果相互关联的两个类所表达事物之间存在“整体”与“部分”的关系，关联关系将被强化为聚合关系。更进一步，如果“整体”与“部分”之间存在“皮之不存毛将焉附”的关系，那么聚合关系被强化为组合关系。

其二，关联关系的访问方向（Navigability）。有必要考察双方向关联关系中某一方向存在的必要性，因为就整体结构而言，两个类之间单方向的关联关系比双方向的关系弱。

其三，相互关联的类的对象多重性（Multiplicity）。明确的多重性是后续实施活动的必要的依据。如果多重性大于 1 的情形，通常需要进一步设计 Container Class，比较常见的诸如 sets、lists、dictionaries、stacks 和 queues。如果多重性的最小取值为 0，意味着与关联关系相应的“连接”有不存在的可能，通常需要建立判断“连接”是否存在的操作。

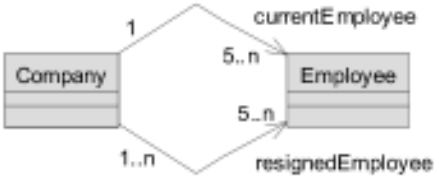


图 9-10　标注角色以明确关联关系的内涵

其四，关联关系中的角色（Role）。标注角色不仅有助于对业务逻辑和模型内容的理解，角色的命名实质上就是一个对象将另一对象作为自己数据成员（Data Member）的名称。如果两个类之间存在双重关联关系的情形，角色的作用更为显著，参见图 9-10。

9.2.3　概念：分解（Factoring）和委托（Delegation）

分解（Factoring）和委托（Delegation）是常见的复用设计内容的方式。

如果拟复用的设计内容所属的类定义允许改动，通常使用分解（Factoring）的方式。图 9-11 给出一个简单的例子，现有一个“动物”类，希望新建的“植物”类能够复用“动物”类中的部分设计内容，即生长（）和死亡（）。

如果拟复用的设计内容所属的类定义不允许改动，可以考虑使用委托（Delegation）的方式。图 9-12 给出一个简单的示例，表面上看起来有些不合逻辑，实质上是对限制条件的折衷。注意，此处的组合关系仅仅反映一种设计的变通方式，没有实际的逻辑内涵。

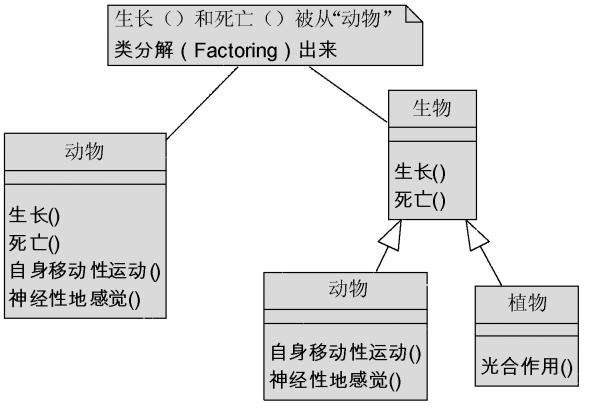


图 9-11　分解（Factoring）示例

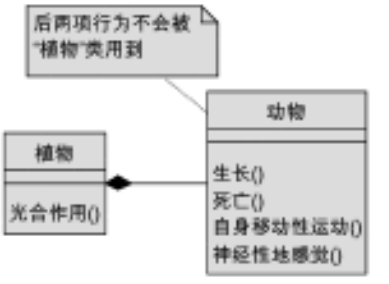


图 9-12　委托（Delegation）示例

9.2.4　步骤 1：明确依赖关系

在分析任务中建立的关联关系主要介乎于“核心设计元素”之间，不妨称之为“早期关联关系”，是“明确依赖关系”的工作重点。在设计任务中得到的“核心设计元素”与“外围设计元素”间关系来自对“设计机制”的应用，它们属于成熟的设计内容。

考察并判断哪些“早期关联关系”继续作为关联关系，哪些被“弱化”为依赖关系。如果对象之间的“连接可见度”为“域”，相应的类之间保持关联关系。如果对象之间的“连接可见度”为“全局”、“局部”或者“参数”，相应的“早期关联关系”被“弱化”依赖关系。

另外一方面，考察操作的标识（Signature）和实现方法（Method）中涉及的其他类，这些信息有助于发现对象间存在的“局部”或“参数”类型的“连接可见度”，从而充实和完善类之间的依赖关系。

9.2.5　步骤 2：细化关联关系

对于那些“保留下来”的关联关系，做进一步的细化，以便为后续的设计和实施活动提供必要的依据。需要明确几方面问题：根据应用逻辑，将更为“紧密”的关联关系转化为聚合关系甚至组合关系；在满足逻辑要求的条件下，尽可能避免双重的访问方向；标注影响后续设计和实施活动的多重性；标明类在特定关联关系中所扮演的角色。

9.2.6　步骤 3：构造泛化关系

与依赖或关联关系不同，类之间的泛化关系并不能通过渐进的分析和设计活动而得出。设计模型的内容增多之后，属于同一概念范畴的类（“子类”）之间会存在相似的行为与结构。为了提高设计内容的复用能力并降低维护的难度，可以将它们的共通部分抽取出来并定义成新的类（“父类”），在“子类”和“父类”之间建立泛化关系。实践中，获取泛化关系的途径不仅可以“从特殊到一般”，也可以“从一般到特殊”。很多时候，已有的“设计类”可以直接作为“父类”，用于简化相关“子类”的定义。

9.2.7　技巧：定义“关联类”（Association Class）

有些时候，关联关系本身也可能具有属性，可以用“关联类”（Association Class）为这种关联关系建模，其标记方法参见图 9-13。当实施环境不支持“关联类”的概念时，需要作出进一步的设计，图 9-14 给出了一个示例。

图 9-13 “关联类”示意

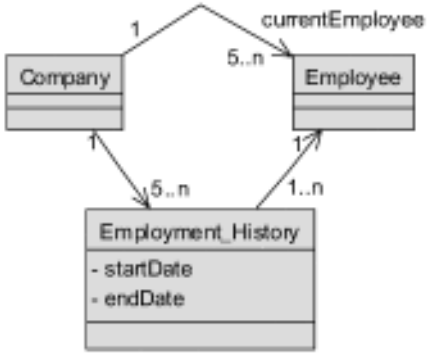


图 9-14 “关联类”进一步设计示意

9.2.8 技巧：定义“嵌入类”（Nested Class）

如果类 A 和类 B 之间存在关联关系，并且类 B 对象仅被类 A 对象引用，那么可以考虑将类 B 作为类 A 内部的嵌套类（Nested Class）。这样做的益处是能够获得更简单的设计模型，并且加快类 A 对象向类 B 对象的消息传递。不利的因素是无论类 B 的实例存在与否，都必须为其分配空间。

9.2.9 技巧：用组合关系分拆“胖”类

某些由实体类演化而来的“设计类”拥有很多属性，不妨称之为“胖”类 F。引用类 F 的对象，需要加载所有的属性，有可能造成资源的低效使用。在诸多属性中，往往只有一部分是“常用”的内容，还有一部分是“罕用”的内容。在这种情形下，可以考虑将那些“罕用”的属性单独组成一个类 H，并且在类 F 和类

H 之间建立一对一的组合关系。类 H 的实例将按需被“激活”。

9.2.10 技巧：引入适用的设计模式

设计模式比“构架机制”更具有普遍适用性，是针对一般设计问题的通用解决方案。“细节设计”任务是考虑引入成熟设计模式的较好时机。例如，借助适用的设计模式，可以分离类在不同状态下的行为定义（State），分离类的抽象内容多样性与实施方式多样性（Bridge），分离创建者与被创建者（Factory Method）…诸如此类[1]。笼统而言，引入设计模式的价值是重复利用前人的设计经验。落实到具体的层面，设计模型内容的耦合度被降低，适应变化的能力增强，面向对象的优势体现得更充分。

当然，设计模式在其他的任务中同样可以被引入。例如，当一个对象的状态发生变化将影响其他对象产生相应变化时，通常会采用 Observer 模式。这种情况很可能出现在“局部设计”任务当中。

9.2.11 示例

以“提交报销申请”的“Use Case 实现”为讨论范围，图 9-15 展示了迄今为止得到的“设计元素”间关系[2]。为了突出重点，在这张图中隐藏“设计元素”的属性和操作。

从图示中不难看出，“设计元素”之间的关系是通过各项任务逐步积累起来的。

“全局分析”任务中获得“关键抽象”之间的概念性关联关系。例如：Employee 和 Claim_report 之间的聚合关系，Claim_report 和 Claim_record 之间的组合关系。

“局部分析”任务中对消息传递路径的粗略表述，即“早期关联关系”。例如：SubmitClaimForm 和 ISubmitClaim（SubmitClaim 的“代言人”）之间的关联关系，SubmitClaim 和相关“设计元素”之间的关联关系（包括 I_HRDatabase、Employee、Claim_report 和 I_MailSystem），Claim_report 和 Valid_Rule 之间的关联关系。

“全局设计”任务中定义“子系统接口”时建立的依赖关系。例如：I_HRDatabase 对 Employee 的依赖关系，I_MailSystem 对 Claim_report 的依赖关系。

“局部设计”任务中引入“构架机制”所要求的关系，这些关系较多，同时也比较成熟。

在“细节设计”任务中，进一步明确那些“早期关联关系”。

图 9-16 展示“明确依赖关系”之后的结果。鉴于“子系统接口”I_HRDatabase 和 I_MailSystem 是全局范围内可用的资源（Utilities），SubmitClaim 与两个“子系统接口”之间的“早期关联关系”被“弱化”为依赖关系。

在 Claim_report 的操作 validate（）中将使用 Valid_Rule 的实例作为一个局部

---

[1] 《Design Pattern：Elements of Reuseable Object-Oriented Software》

[2] 在建模工具中，建立一张新的类图，将相关的“核心设计元素”拖入该类图中，可以确保不遗漏地发现从不同活动中捕获的各种关系。

变量，用作呼叫 Claim_record 操作 validate（the Rule：Valid_Rule）的参数内容。Claim_report 对象到 Valid_Rule 对象的“连接可见度”为“局部”，Claim_report 和 Valid_Rule 之间的“早期关联关系”被弱化为依赖关系；Claim_record 对象到 Valid_Rule 对象的“连接可见度”为“参数”，建立 Claim_record 对 Valid_rule 的依赖关系。

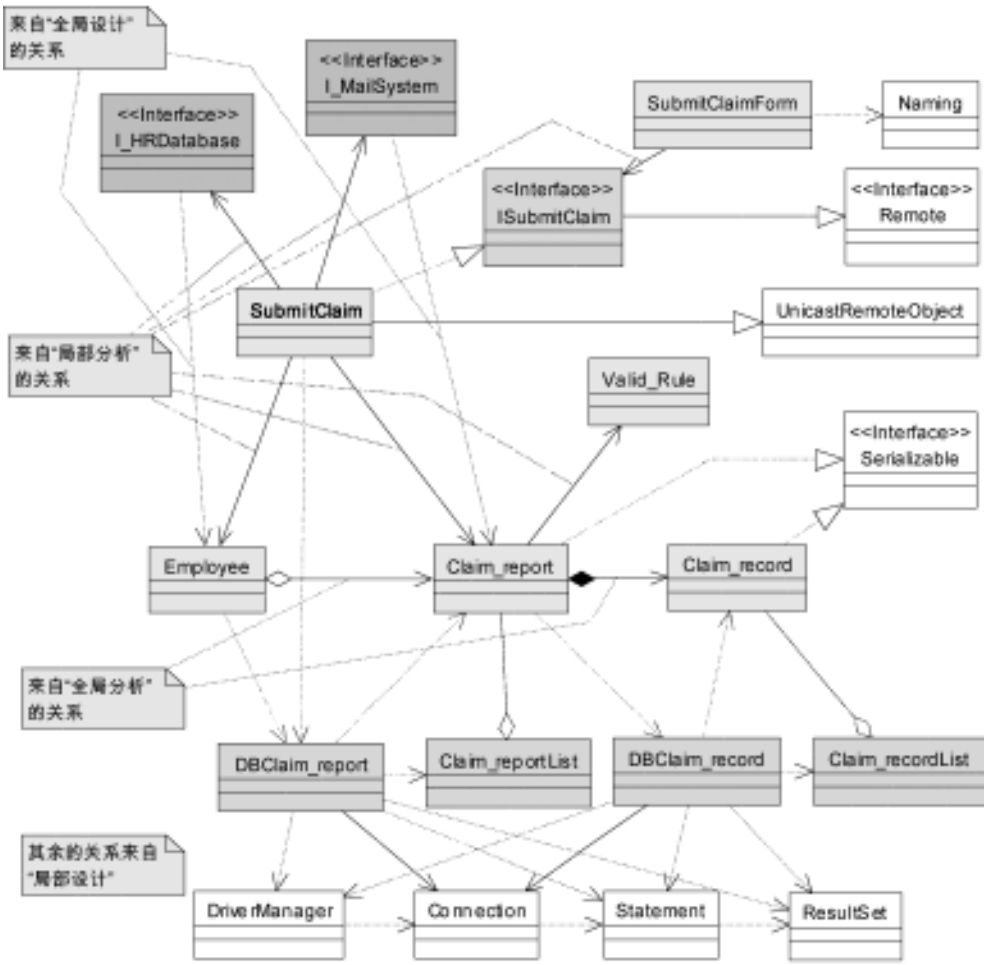


图 9-15 “细节设计”以前获得的“设计元素”间关系示例

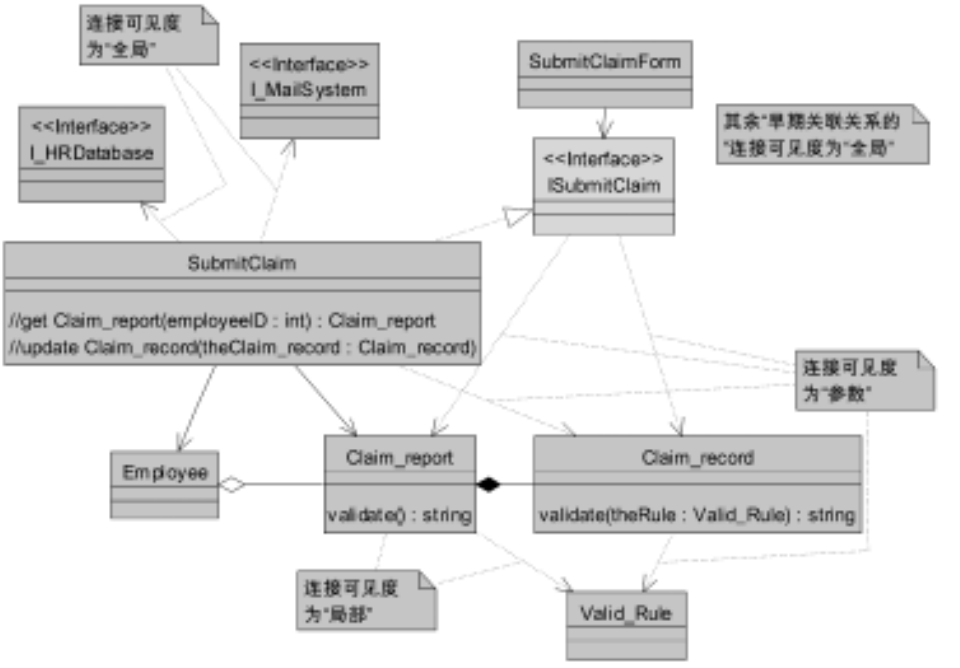


图 9-16 **“明确依赖关系”示例**

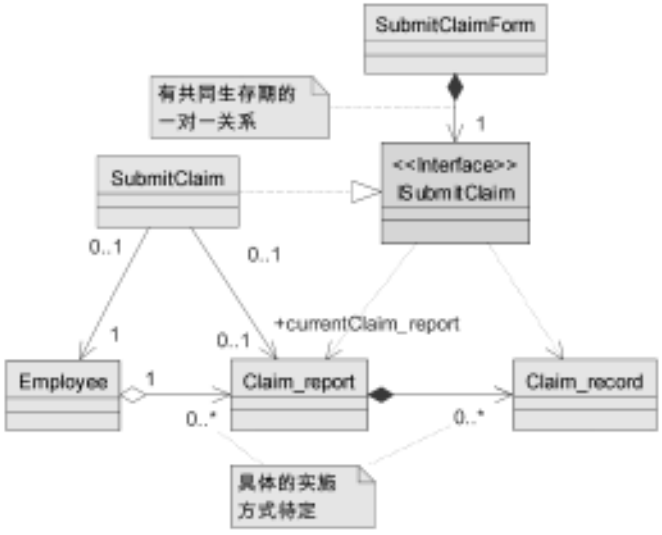


图 9-17 **“细化关联关系”示例**

图 9-17 **展示被进一步细化的关联关系。主要是明确关联关系中的角色（**Role**）和多重性（**Multiplicity**）。**

SubmitClaimForm 和 ISubmitClaim 的对象拥有共同的生存周期，并且是一对一的多重性，因而关联关系被强化为一对一的组合关系。

在 SubmitClaim 和 Claim_report 之间的关联关系中，标识出 Claim_report 在这一关系中所扮演的角色 currentClaim_report[□]，即 SubmitClaim 对象中 Claim_report 类型数据成员的名称。它们之间的多重性表示一个 SubmitClaim 对象对应一个（或不对应）一个 Claim_report 对象，反之亦然。

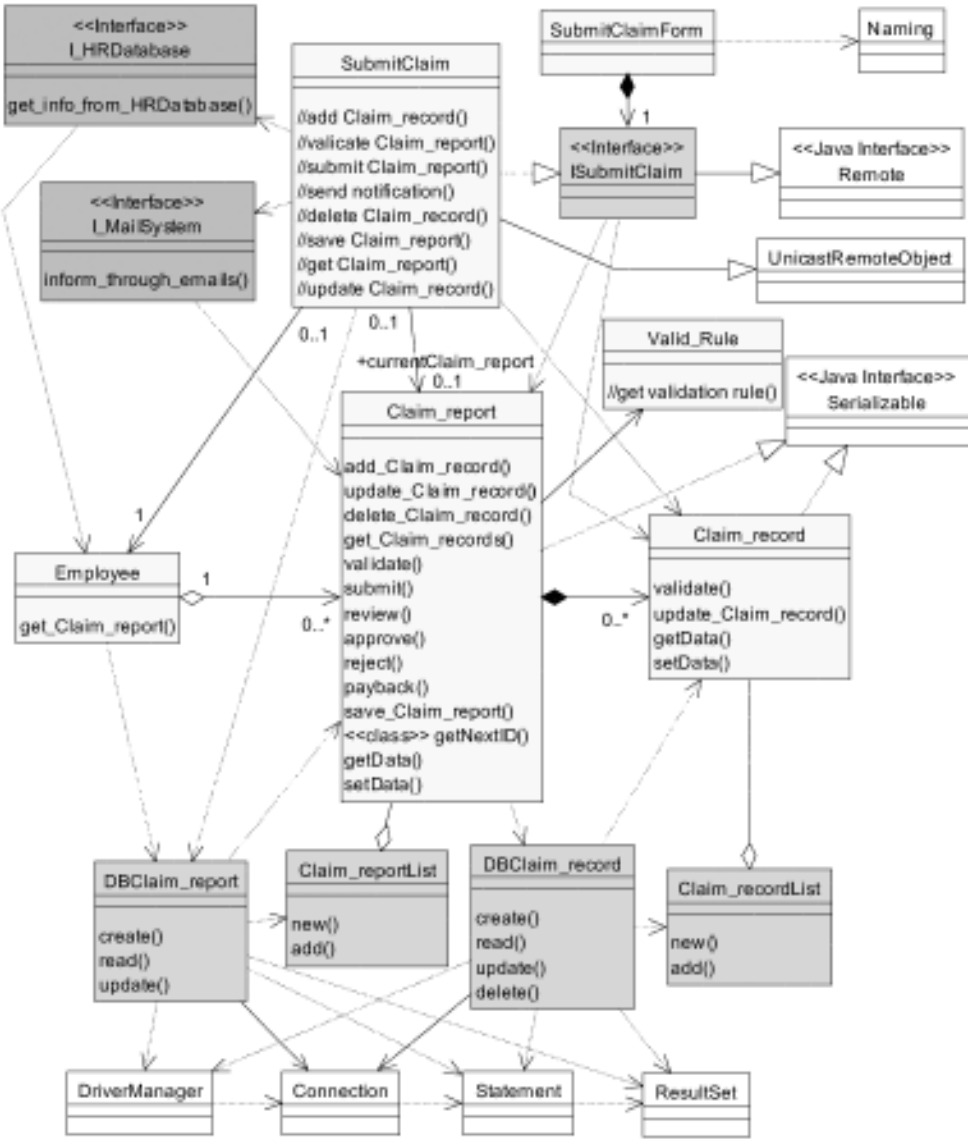


图 9-18　现阶段的“参与类图”

[□] “+”表示 SubmitClaim 对象的相应数据成员是“公开”的。

图 9-18 展示（至此为止）“提交报销申请”“Use Case 实现”的“参与类图”。

“构造泛化关系”步骤的示例中，引入两个参与其他“Use Case 实现”的“设计类”，即 Manager（参与“批复报帐申请”的“Use Case 实现”）和 Accountant（参与“审核报帐申请”的“Use Case 实现”）。根据基本的业务常识，经理（Manager）和财务人员（Accountant）同属于员工（Employee）范畴。三个类的结构和行为有很多相似之处，在它们之间构造泛化关系，能够带来明显的益处，参见图 9-19。图 9-20 是相同逻辑内容的变通形式，与这种结构对应的动态“变形”[□]过程负担较轻。

图 9-19　泛化关系示例

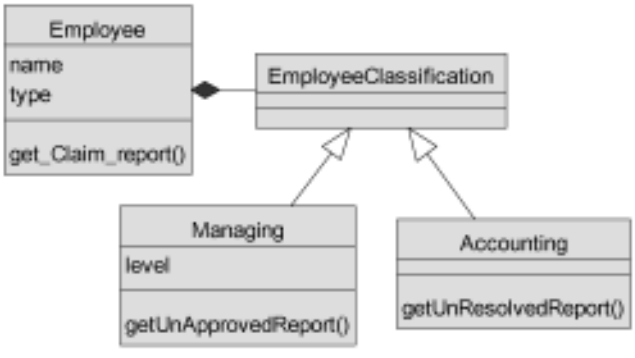


图 9-20　泛化关系的变通形式

□ 变形即 Metamorphism，此处的具体含义是从 Manager 对象变形为 Accountant 对象。

第三部分　设计模型的沿用

——再接再厉

第 10 章　设计模型向实施模型的过渡

作为分析和设计的结果，设计模型是建立实施模型的基本依据。本章先介绍实施模型的基本概念，然后说明设计模型向实施模型的过渡关系。

10.1　实施模型的基本概念

10.1.1　实施模型

实施模型（Implementation Model）是构件（Component）、构件所在的“实施子系统”（Implementation Subsystem）以及它们之间关系和图示内容的集合。实施模型可视化地表述拟建系统的代码框架，具有明确的物理意义，是构件视图（Component View）中的主要内容。实施模型可以具有类似于设计模型的层次划分（利用构造型为《layer》的包），参见图 10-1。

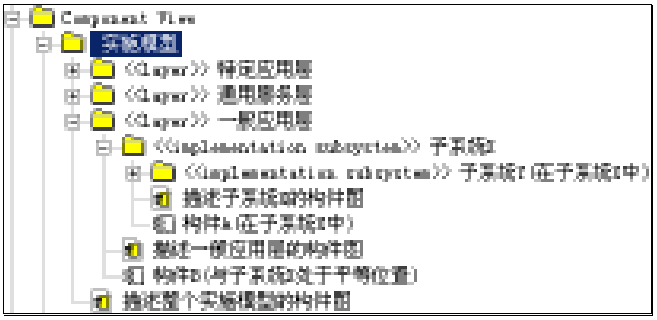


图 10-1　实施模型的结构示例

通常，在确定系统整体构架时，由系统构架师给出实施模型的框架，然后，根据设计内容的演化，在后续实施（编码）活动中不断作出调整。

类似于分析和设计活动，实施活动事实上也由两类角色协作完成，即系统构架师和实施员（编码人员）。其中，系统构架师负责全局范围的判断与决策，实施员负责完成符合设计要求的代码并进行相关的单元测试（Unit Test）。

10.1.2 构件

构件

构件可以是源代码、二进制代码、可执行代码、文档、其他类似内容以及它们的集合体。相对逻辑内容而言，构件是有形的实际内容（Physical）。鉴于构件的具体表现形式多种多样，在 UML 表述中，可以通过不同的构造型说明特定的语义。例如《EXE》、《DLL》或《HTML》，参见图 10-2。

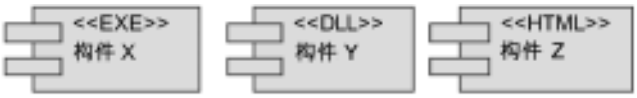


图 10-2　构件的 UML 表述

如果基于部署（Deployment）的视角，只有某些类型的构件是可部署的内容，例如可执行文件（EXE）、链接库（DLL）和超文本页面（HTML）。另外一些类型的构件通常不作为部署的内容，仅作为工作产品（Work Product）存在，例如源代码、目标代码和编译文件（Makefile）。

最初的构件可能出现于早期构造原型系统的活动中，尽管此时还没有完整意义上的实施模型。关键的构件通常在确定系统整体构架时被指定。大量的构件在实施活动中被确认并实现。在整个开发生命周期中，构件被持续地测试和更新。

构件之间的依赖关系

如果构件 A 对构件 B 存在依赖关系，那么存在两种可能：其一，构件 A 在编译过程中需要使用构件 B 中的相关内容，称为“编译依赖关系”[□]；其二，构件 A 的运行以构件 B 的运行为必要条件，称为“运行依赖关系”。构件之间依赖关系的 UML 表述参见图 10-3。

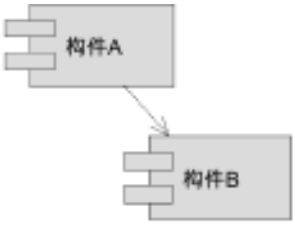


图 10-3　UML 表述构件之间的依赖关系

□“编译依赖关系”在不同程序设计语言中的表现形式有所区别，例如在 Java 中用 import 语句表示，在 C++中用#include 语句表示，诸如此类，不枚举。

构件对“子系统接口”的实现关系

如果一个构件能够完成（设计模型中定义的）“子系统接口”所规定的外部可见行为，则可以在该构件与相应“子系统接口”之间建立实现关系，其 UML 表述参见图 10-4。注意，构件具有明确物理意义，而“子系统接口”只具有逻辑意义。

图 10-4　UML 表述构件对接口的实现关系

10.1.3　“实施子系统”

“实施子系统”

“实施子系统”是构件与其他（小粒度）“实施子系统”的集合体，是实施模型的组织单元。其 UML 表述形式是构造型为《implementation subsystem》的包，参见图 10-5。广义上，一个“实施子系统”也可以被看作为（大粒度的）构件。

图 10-5　“实施子系统”的 UML 表述

实施模型中的“实施子系统”和设计模型中（设计）包的概念有些类似，但视角不同。注意，“实施子系统”和设计模型中的子系统概念不同，前者具有明确的物理意义，而后者强调逻辑层面的内容。

“实施子系统”的主要价值是降低模型的复杂度，当实施模型中包含很多构件时效果更加显著。通常，“实施子系统”在文件系统中表现为目录（Directories），类似于 Java 语言中包（Package）的概念。“实施子系统”是宏观模型框架向微观构件内容逐步过渡的组织形式，是系统构架师和实施员之间沟通的介质，同时为项目管理人员布置任务提供了有利的依据。

“实施子系统”约束其包含内容的“可见度”（Visibility）。如果“实施子系统”X 中的构件 A 是“外部可见”的，意味着“实施子系统”X 外部的构件 B 可以依赖于构件 A。缺省情况下，“实施子系统”中的构件“外部可见”。

通常，在确定系统构架时，由系统构架师划分“实施子系统”，具体任务是将设

计模型中的“设计包”映射为实施模型中的“实施子系统”。实施模型和设计模型在组织形式上的相似性取决于（设计）包和“实施子系统”的映射关系。理想情况下，一个“实施子系统”对应一个（设计）包，不仅可以复用逻辑视图中的分解策略，而且有可能实现设计到代码的无缝追踪，大幅度降低后续维护工作的成本。尽管现实中不容易达到理想状况，但至少应该作为努力的方向。对于那些不能形成一一映射的设计与实施内容，应该遵从相对统一的规范并且作出明确的说明。

“实施子系统”之间的“导入依赖关系”

如果“实施子系统”A 到“实施子系统”B 存在“导入依赖关系”（Import Dependency），意味着“实施子系统”A 中的构件可以依赖于“实施子系统”B 中的“外部可见”构件，参见图 10-6。比较常见的情形是“实施子系统”A 中的构件 P 对“实施子系统”中的构件 Q 存在“编译依赖关系”。

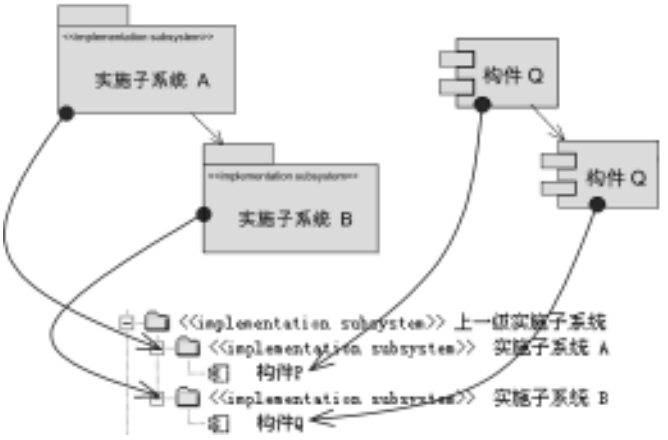


图 10-6 “实施子系统”之间的导入依赖关系

系统构架师可以通过“实施子系统”之间的“导入依赖关系”控制“实施子系统”之间的可见性，即物理层面体系构架中的耦合。实施员必须遵循系统构架师的总体考虑，按照指定的规则引用那些“外部可见”的构件，避免出现不必要的“导入依赖关系”，从而降低演进和维护整体构架的难度与成本。

10.1.4 构件图

构件图（Component Diagram）用于说明实施模型的静态结构，可视化地描述构件、“实施子系统”以及它们之间的关系，主要包括以下几个方面。

- “实施子系统”之间的“导入依赖关系”。
- 源代码之间的“编译依赖关系”。
- 可执行代码之间的“运行依赖关系”。
- （物理层面）构件对（逻辑层面）“子系统接口”的实现关系。

10.2 设计模型向实施模型的过渡

10.2.1 明确实施模型的依据

分析和设计活动主要基于设计师的逻辑（Logical）视角。实施活动主要基于实施员的物理（Physical）视角。两种视角具有不同的侧重，但是，他们各自关注的内容具有直接或间接的对应关系。概念上，设计模型用于实现需求模型中的应用逻辑和技术要求，实施模型的内容则是对设计模型的实现。尽管实施模型的大多内容在设计模型中能够找到相应的根据，实践中，实施活动仍然须要直接参照需求模型中的内容。

10.2.2 建立实施模型的框架

建立实施模型的框架是一项全局性的任务，由系统构架师承担，大致可以划分为三个步骤。

首先，创建初始的实施模型。逻辑视图向构件视图转换的起点是在实施模型中反映设计模型的结构，具体讲就是用实施模型中的层次关系与“实施子系统”的嵌套反映设计模型中的层次关系与（设计）包的嵌套。“实施子系统”的组织方式不仅要和设计模型建立明确的对应关系，而且还要考虑便于多个（组）实施员并行工作。在迭代开发过程的早期，就有必要关注和搭建实施模型的框架。某些情况下，有必要根据实施模型的内容适当调整设计模型的结构。如果实施活动中要使用预先存在的构件，例如遗留代码或买来的构件[1]，应及早将相关内容添加到实施模型中的适当位置，并指明它们所实现的“子系统接口”。

接下来，在“实施子系统”之间定义“导入依赖关系”。概念上，“实施子系统”之间的“导入依赖关系”是设计模型中（设计）包之间依赖关系的体现。有时，对“实施子系统”的 调整与折衷会引入新的“导入依赖关系”。举一个例子，如果“实施子系统”A 引用“实施子系统”B 中的类型声明[2]是它们存在“导入依赖关系”的惟一理由，则可以考虑将 B 中的类型声明内容提取为独立的“实施子系统”C，然后建立 A 到 C 以及 B 到 C 的“导入依赖关系”。这样做既解除了 A 和 B 之间的耦合，又提升了（C 所含）类型声明内容的复用能力。

然后，确定可执行模块。可执行模块用构造型为《EXE》的构件表述，对应于实施模型中特定的“实施子系统”，表述可执行模块的构件隶属于该“实施子系统”对应的包，参见图 10-7。注意，可执行模块的粒度（Granularity）通常大于一般的“实施子系统”，因而只有少数“大粒度”的“实施子系统”对应于可执行模块。

---

[1] COTS，Commercial-Off-the-Shell

[2] Declarations of types

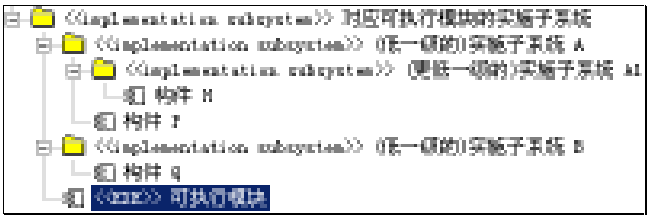


图 10-7　可执行模块隶属于相应“实施子系统”的包

10.2.3　实现设计模型的内容

从微观上看，设计模型的基础要素是“设计类”，实施模型的基础要素是构件。设计模型中的“设计类”可以通过两种方式映射成实施模型中的构件：复用已经存在的构件；采用特定程序设计语言编写“代码构件”。

实施活动的主要任务是用“代码构件”实现“设计类”，实施员一方面要遵循统一的程序编写规范（“设计类”向“代码构件”映射的一般性规则），另一方面要充分参考并利用已经被实施的内容（完成的代码）。实施“设计类”主要包括几方面具体内容。

- 操作。主要是选择算法、确定合适的数据结构、按需定义嵌入类（Nested Class）或私有操作（Private Operation）以及编写具体的方法。
- 状态。对于状态特征比较简单的类，基于属性取值的条件判断即可实现相应的状态转换逻辑；对于那些状态特征比较复杂的类，可以考虑应用 State 模式或采用表驱动法。
- 关联关系。类 A 到类 B 的单向关联关系被实施为类 A 中的属性，该属性是指向类 B 对象的引用；如果多重性大于 1，则需要将相应的属性实施为引用集。双向关联关系的情形类似。面向对象程序设计语言通常提供相关的可复用构件，用于实施各种关联关系。
- 属性。具体讲是确定属性的类型，可以使用程序设计语言提供的基本变量类型，也可以使用已定义或新定义的类作为属性的类型。

如果“设计类”和“代码构件”之间能作到一一对应，那么实施模型对设计模型的映射将一目了然，但这种状况通常只可能出现在某个局部。很多时候，一个“设计类”会由一组“代码构件”实现；其中一个“代码构件”直接对应该“设计类”，称之为“主代码构件”；其他的“代码构件”协助“主代码构件”实现“设计类”的逻辑内容，称它们为“辅代码构件”。

总体结构上，实施模型和设计模型具有映射关系和相似性，它们之间的差异主要缘于实施模型所针对的特定程序设计语言[1]。相对于设计模型，实施模型降低了抽象水平，从而更加贴近具体的实施环境。

[1] 特定程序设计语言相对于 UML 描述的一般化面向对象逻辑，或多或少地存在客观局限性。

第 11 章　设计模型和数据模型的关联

11.1　数据模型的基本概念

11.1.1　数据模型

数据模型描述拟建系统中“留存”数据的逻辑和物理内容与结构，还包括相关的数据库行为（如存储过程）。

数据库设计师负责建立和维护数据模型。最初的数据模型在确定系统整体构架时建立，主要内容是那些对体系构架影响较大的“留存”数据，数据模型在后续活动中不断地演进和完善。

UML 语言可以清楚地描述关系型数据模型，关于 UML 在数据建模中的语义扩展，请参考附录“UML 用于数据建模的构造型”。

11.1.2　实体和关系

在关系数据模型中，一个实体（Entity）对应于一个表（Table）或几个表的逻辑组合，即所谓的视图（View）。表的内容是若干个“行”（Row），称为“记录”（Record），参见表 11-1。表的结构是一组“列”（Column），每个列有相应的名称和类型，可以采用数据库固有的基本数据类型或自定义的类型，参见图 11-1（结构）。UML 中用类的构造型《Table》描述表的概念。

表 11-1　T_Employee 表／实体的内容

EmployeeID	name	band	Title	manager
0001	张三	5	信息系统专员	赵六
…	…	…	…	…
0069	李四	7	销售助理	钱七
…	…	…	…	…

图 11-1　用 UML 表述实体／表 T_Employee 的结构

在关系数据模型中，实体之间的关系（Relationship**）通常由一个表的外键（**Foreign Key**）对另一个表的主键（**Primary Key**）的引用（**Reference**）关系实现。参见表 11-2 和图 11-2，**T_Claim_report **表的外键为** employeeID **列，**T_Employee **表的主键为** employeeID **列。**

表 11-2　实体／表之间关系的具体内容

employeeID	name	band	title	Manager
0001	**张三**	5	**信息系统专员**	**赵六**
…	…	…	…	…
0069	**李四**	7	**销售助理**	**钱七**
…	…	…	…	…

reportID	date	city	type	amount	employeeID
…	…	…	…	…	…
000250	02/13/02	**上海**	**住宿**	620.00	0001
…	…	…	…	…	…
000356	09/11/02	**成都**	**机票**	945.00	0069
…	…	…	…	…	…

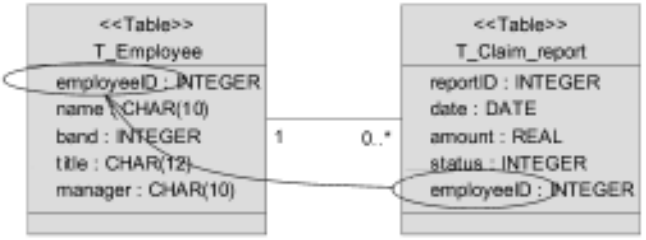


图 11-2　用 UML 描述实体之间的关系

11.1.3　存储过程

存储过程（Store Procedure）是运行于数据库管理系统进程空间内的可执行代码，是数据库行为的主要体现。灵活利用在数据库“本地”运行的存储过程，能够显著地提升软件系统的整体性能。

存储过程主要分为两种，过程（Procedure）与触发器（Trigger）。过程被“显式”地调用，有指定的参数和返回值，通常用来实现某种明确的目标；触发器在特定数据库事件（例如增、删记录）发生时被“隐式”地调用，没有指定的参数和返回值，通常用于完成某些必然的连锁动作。

11.2　设计模型和数据模型的映射

11.2.1　面向对象和关系型数据的差异

面向对象和关系型数据在概念与思维方式上存在明显的差异，前者强调对行为的描述，后者强调对数据本身的描述，优劣不能一概而论。面向对象在软件系统的分析、设计和实施中越来越普遍地被应用，而关系型数据结构又是信息存储的主导方式。实践中的关键是如何将两者更好地结合，换言之，面向对象的设计模型如何更好地与数据模型关联起来。

广义上，数据模型具有概念、逻辑和物理等不同层次的内容，它们可能出现于 Use Case 视图□、逻辑视图以及构件视图当中。本章所涉及的数据模型内容主要集中在逻辑层面，将设计模型中的“留存类”映射成相应的数据模型内容。注意，当“留存类”的结构相对稳定之后，方可将其映射到数据模型中。

11.2.2　映射“实体”

基于具体内容的视角，设计模型中“留存类”的一个实例（对象）和数据模型中表的一条记录（行）相对应。基于组织结构的视角，“留存类”和表相对应，类的名称对应表的名称（表的名称在相应类的名称基础上添加前缀“T_”），类的属性名称与类型对应表的列名称与类型，参见图 11-3。通常，没有必要将“留存类”的所有属性均映射到数据模型当中，只须映射那些需要“留存”的属性。

在关系数据库的表中，记录没有自己的标识（Identity），但是记录所在的表有一（或多）个列的取值能够保证惟一地识别每条记录，即所谓的主键（Primary Key）。因为主键用于惟一地标识表中的记录，通常采用系统指派的数值型列充当

主键，即通常所指的 ID。在面向对象系统中，每个对象都拥有区别于其他对象的标识，对象的标识通常作为一（或多）项属性，和主键的概念相似。

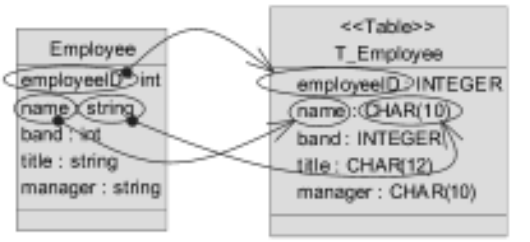


图 11-3　设计模型中的“留存类”和数据模型中的表相对应

11.2.3　映射“关系”

在面向对象的设计模型中，类 A 到类 B 的关联关系表现为类 A 的属性，该属性指向类 B 的对象（组）。如果设计模型中的类 A 和类 B 在数据模型中被映射为表 T_A 和表 T_B，那么类 A 到类 B 的关联关系在数据模型中表现为表 T_B 的外键，外键的取值来自于表 T_A 的主键取值。通常，称 T_B 为“子表”（Child Table），称 T_A 为“父表”（Parent Table）。读者可以参照图 11-4 给出的示例加以理解。

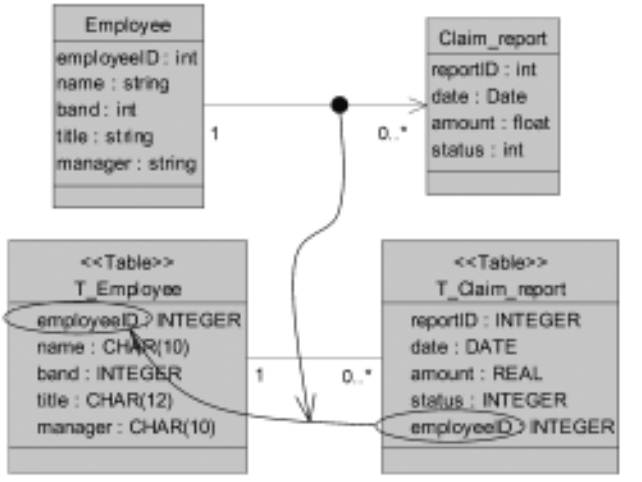


图 11-4　设计模型中关联关系在数据模型中的反映

注意，在面向对象设计模型中，Employee 对象通过（实现关联关系的）

□ 业务模型（Business Model）中的业务实体（Business Entity）及关系。

属性“获知”相关的 Claim_report 对象；在关系数据模型中，Claim_report 的记录通过（实现关联关系的）外键列取值“获知”与其对应的 Employee 记录。

在数据模型中，对聚合关系（Aggregation）的映射与关联关系（Association）类似，对组合关系（Composition）的映射则有所区别[1]。在组合关系中，如果类 A 表达“整体”概念，类 B 表达“部分”概念，那么，表 T_B 的外键将作为表 T_B 主键的组成部分。读者可以参照图 11-5 加以理解。一种通俗的解释是：Claim_report 记录的序号将作为惟一标识 Claim_record 记录的一个组成部分，因而 Claim_record 的 recordID 只需要在所属的 Claim_report 范围内保持惟一。

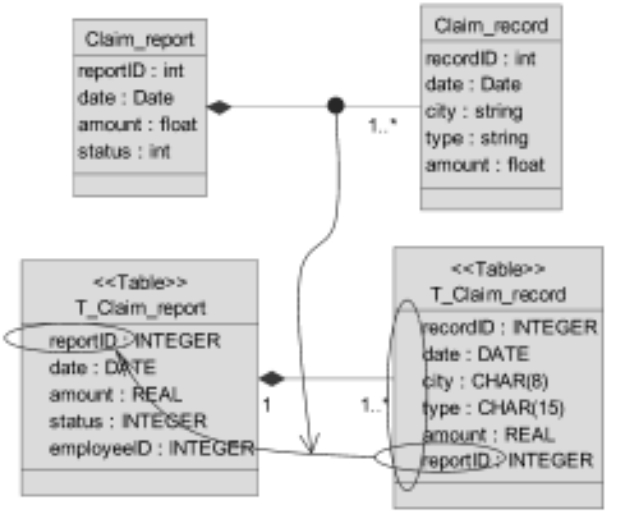
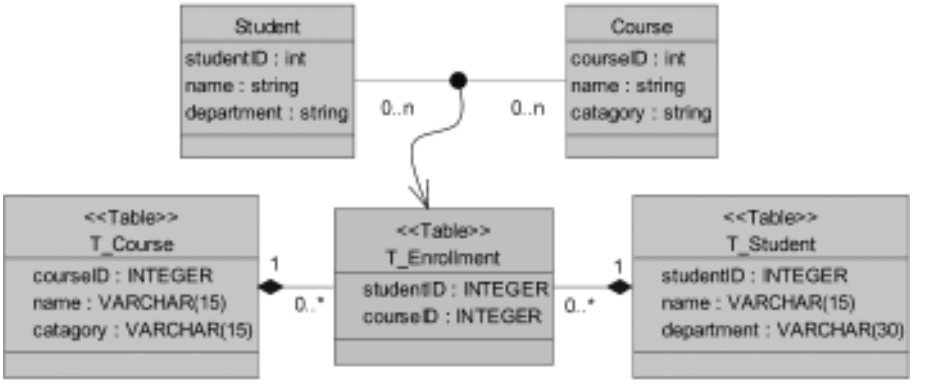


图 11-5　设计模型中组合关系在数据模型中的反映



[1] 在数据模型中，与关联关系或聚合关系对应的实体间关系为 Non-Identifying，与组合关系对应的实体间关系为 Identifying。

图 11-6　设计模型中多对多的关联关系在数据模型中的反映

在设计模型中，如果类之间存在多对多的关联关系，需要在数据模型中建立交叉实体（Intersection Entity）。图 11-6 给出一个学生与课程之间多重关联关系的数据模型映射示例，图 11-7 是使用交叉实体映射“关联类”的示例。

概念上，关系数据模型并不能直接表述泛化关系，但是可以采用变通方式实现。假设类 A（“子类”）和类 B（“父类”）之间存在泛化关系，以下分别解释两种间接的映射方式。

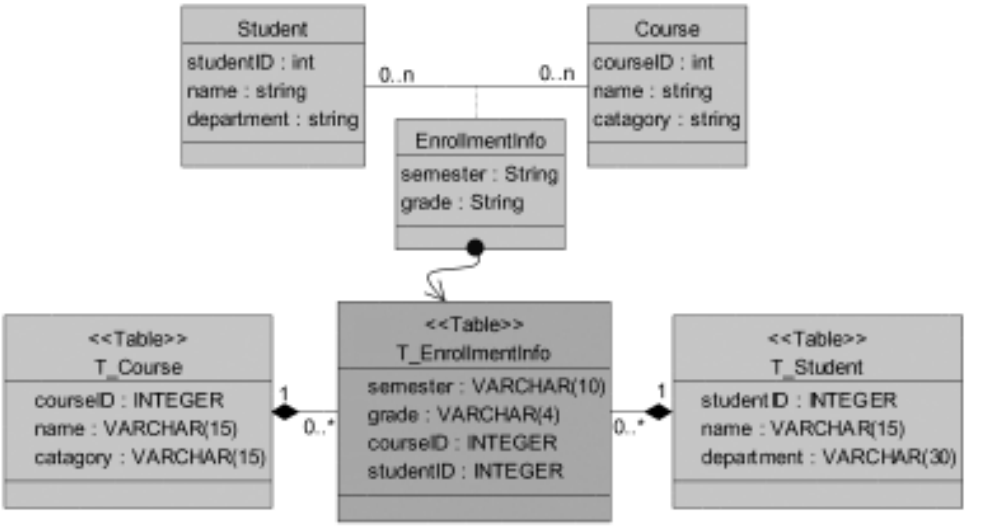


图 11-7　设计模型中“关联类”在数据模型中的反映

在第一种方式中，用两个表（T_A 和 T_B）分别映射类 A 和类 B，表 T_A 的外键引用表 T_B 的主键；然后基于 T_A 和 T_B 建立视图 V_A，完整地对应类 A 所表达的概念，参见图 11-8（Manager 为类 A，Employee 为类 B）。这种映射方式的缺点是造成额外性能开销，不过，概念比较清晰并且易于维护。

在第二种方式中，用两个表（T_A 和 T_B）分别映射类 A 和类 B，表 T_A 不仅映射类 A 所包含的内容，而且映射类 A 从类 B 中“传承”下来的内容，结构上表 T_A 将具有表 T_B 所拥有的全部列，并且表 T_A 和表 T_B 之间没有关系，参见图 11-9。这种映射方式得到数据模型结构比较简单，代价是信息的冗余和维护的困难。

注意，面向对象设计模型中的依赖关系（Dependency）和实现关系（Realization）都是非结构化的关系，因而不会映射到关系数据模型中。

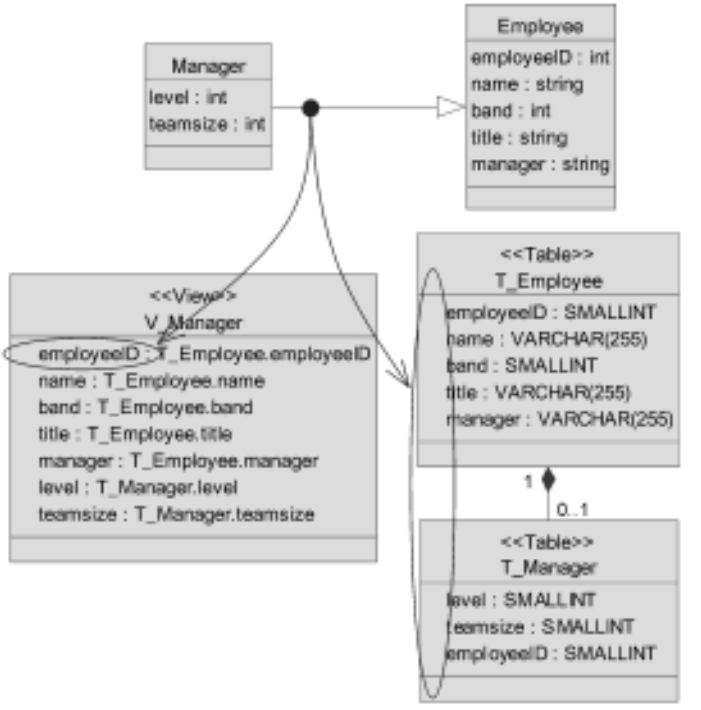


图 11-8　**设计模型中泛化关系在数据模型中的反映（1）**

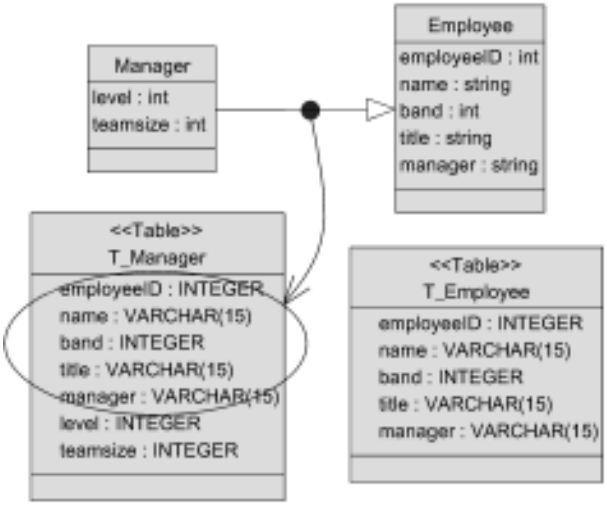


图 11-9　**设计模型中泛化关系在数据模型中的反映（2）**

11.2.4 映射围绕数据的行为

为了优化拟建系统的性能（主要是减少 I/O 和网络传输压力），有必要考察设计模型中“留存类”的操作，判断它们是否更适于被实施为存储过程。换言之，将那些紧密围绕数据的行为指派到数据库管理系统中完成。有几种常见的候选操作：用于简单处理“留存”对象或属性的操作（如成批增、删等）；用于计算查询内容的操作（如求和、求均值等）；用于验证完整性（Integrity）的操作。

数据库内容的完整性主要体现在两个方面，即数据完整性（Data Integrity）和引用完整性（Referential Integrity）。数据完整性规则是指数据的取值在规定的范围之内。拟建系统[1]应当对数据的完整性进行检验，同时，在数据库管理系统中可以通过约束（Constrain）来最终确保数据的完整性。引用完整性规则是指每一个外键取值（在所引用的表中）有一个主键取值与其对应。这种完整性为关系型数据结构所特有，通过外键约束（Foreign Key Constrain）得以保障。在关系数据模型中，一个表的记录要通过主键取值与其他表的记录相关联，是一种间接的引用关系，因而存在引用完整性的问题。在面向对象的设计模型中，对象彼此之间直接引用，不存在类似的问题。

实践中，确定哪些行为通过存储过程实施，需要充分参考系统构架师和设计师的建议。

11.2.5 优化性能的考虑

反规范化

最初的数据模型，通常是一组对应（设计模型中）“留存类”的表。很多情况下，面向对象的应用系统需要从数据库中一并调出多个类的对象，其中一个对象的信息是整个动作所关注的焦点，其他对象的信息将通过表联结（Table Join）操作被调出。这类操作通常会耗费较多的数据库运算资源，如果类似的操作很频繁，数据库将不堪重负。为了缓解这种状况，通常采用一种称为反规范化（De-normalization）的技术。反规范化是将多个[2]表的列合并到一个表中，该表对应那个作为关注焦点的“留存类”。反规范化的实质是对多个表进行预先的表联结操作。反规范化是在数据模型中映射嵌套类（Nested Class）的通用办法。

通常情况下，数据库中更新（Update）的单次操作成本要高于读取（Retrieve），但是更新的操作频率通常远远低于读取。反规范化技术提高了更新操作的成本，但降低了读取操作的成本，如果运用得当，在总体性能上将取得积极的效果。当然，这种做法有利有弊。如果较多的查询只针对（联结）表中的某一个列，那么

[1] 数据库管理系统之外的部分。

[2] 应避免对两个以上的表进行反规范化，因为这样不仅提高查询单个对象属性的性能开销，同时加重插入或更新记录的负担。

查询的性能会低于不做反规范化的情形。因为在查询过程中，所有（被联结）列的内容都将被一并读取。

在优化数据模型的过程中，如果陷入进退维谷的两难境地，有可能是面向对象设计模型中存在先天的性能瓶颈或者其他设计缺陷，需要与设计师甚至是系统构架师进行充分的沟通并做出必要的调整。

建立索引

在设计好的表结构基础上，进一步明确拟建系统需要执行的查询类型，目的是建立适用的索引（Index），从而提高数据存取的效率。最常见的是 B-Tree 索引，适用于随机分布的可变索引键值；如果索引键值的范围和取值相对稳定而惟一，散列（Hashed）索引具有较好的性能。

索引通常基于那些具有标识价值的列，这种列的取值往往确立于记录（对象）创建之初并且保持不变。此外，这种列如果是整数型，会比字符型获得更好的性能。以下是需要建立索引的几种常见情形。

- 为主键列建立索引，因为它频繁地被用作搜索和表联结操作的关键字。
- 为经常被查询的内容建立索引。
- 为必须快速得到检索结果的内容建立索引。
- 为用作组合查询条件的内容分别建立索引。

注意，索引在带来积极效应的同时也造成了额外的开销。每一个增、删或更改的操作都会导致索引信息的更新。此外，索引本身会耗费大量的数据存储空间。如果索引使用不妥，它的消极影响很有可能抵消其带来的价值。总之，不应该随意建立索引，以下是一些相关的建议。

- 列数少且行数少的“小”表通常能被一并放入数据库缓存区（Cache），因而编制索引在提升性能方面的实际价值不大。
- 不为那些不经常执行且无特殊速度要求的查询建立索引。
- 如果增或更改记录的性能要求比查记录的性能要求更为突出，应考虑少建索引。
- 如果有大量装（卸）载数据的操作要求，可以先行删除索引，待操作结束后重新建立索引。

第 12 章　整理设计文档

12.1　分析和设计活动中的主要文档

分析和设计活动的主要工作是可视化建模，模型的元素和图述被“立体”地组织在“逻辑视图”之中。但是在很多情况下，人们需要以一种“平面”化的方式了解模型的总体、局部及细节内容，即需要与分析和设计活动相关的文档，主要包括以下几种。

- 设计指南。描述设计过程与设计向实施过渡中所遵循的原则和方法。
- “Use Case 实现”报告。展示分析和设计内容与需求中应用逻辑的关联。
- 设计模型纵览报告。展示拟建系统构架的整体逻辑。
- 设计包报告。展示特定设计包中的元素和图述。
- “设计类”报告。展示特定“设计元素”及其相关内容。

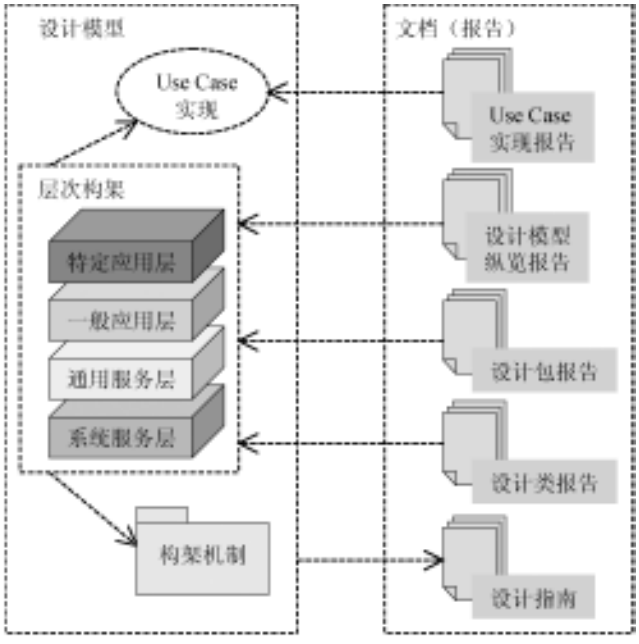


图 12-1　设计文档和设计模型的映射关系

图 12-1 描述了设计文档和设计模型的对应关系。鉴于分析和设计活动的可视化特征，设计文档的很多内容取材于设计模型[1]，大多设计文档事实上是基于模型内容的阶段性报告。相应地，设计文档不必过分拘泥于模板的格式。

上述设计文档的粒度是分层次的，有些针对全局，例如“设计指南”和“设计模型纵览报告”；有些针对局部，例如“Use Case 实现报告”和“设计包报告”；有些针对细节，例如“设计类报告”。不同层次的设计文档之间会存在适度的信息冗余，以利于团队内更平滑的沟通。

在开发活动中，经常会用到所谓的“体系构架文档”，其内容涉及体系构架的多个维度，其中涵盖了与分析和设计相关的部分，是对“设计模型纵览报告”的更高层次概括。

12.2 设计指南

概述

“设计指南”是具有全局意义的指导性文档，描述了设计以及设计向实施过渡过程中要遵循的原则和采用的方法。在确定系统构架时，系统构架师着手编制并不断完善该文档。“设计指南”的内容是建立和充实模型的依据。

在设计模型中起“支撑”作用的经验内容被纳入设计指南。具体讲，“设计指南”中的“机制应用指南”部分与设计模型中的“构架机制”包相对应。设计指南中还包括与设计活动直接相关的外延内容，即指导实施和数据库设计活动的“实施建模指南”和“数据建模指南”部分。

“设计指南”的目的是复用有价值的设计经验，其内容的编排要结合团队的实际情况。为了提升“设计指南”的阅读效率，应该避免将那些简单的常识[2]包括在内。

内容框架

- 简介。概述设计指南的目的、范围、参考资料和内容组织方式。
- 构架设计指南。基于逻辑视角，提供构架设计方面的规则和建议，给出符合高内聚、低耦合原则分解和细化策略。
- 特殊 UML 构造型指南。设计模型中有可能使用特殊的 UML 构造型，在此处解释这些构造型所扩展的语义和适用范围。
- 机制应用指南。说明与应用逻辑没有直接关系的成熟设计经验，即“构架机制”。一方面，列举“构架机制”在不同层面（分析、设计和实施）

[1] 如果建模工具和文档自动生成工具之间有良好的集成，那么可以定期基于模型内容生成所须的文档。Rational Rose 和 Rational SoDA 的集成应用就是一个很好的实例。

[2] 这些内容可以编排成基础培训资料，作为团队新成员阅读设计指南的辅助材料。

的映射关系；另一方面，具体表述机制的静态和动态特征。

- 设计建模指南。给出分析和设计活动遵从的通行规则，典型内容包括：“子系统接口”的指定方式与描述形式；消息、操作和属性的描述与标注风格，对现存物理构件的逻辑描述方式……诸如此类具体指导建议。
- 实施建模指南。基于特定程序设计语言，给出设计向实施映射的一般规则。这部分的典型内容还包括：检测、处理和报告故障的策略，内存管理策略，交易管理策略，特殊的程序结构和算法[1]诸如此类具体指导实施建模的建议。
- 数据建模指南。说明如何将设计模型中的“留存类”及其关系映射到数据模型中，同时给出数据模型特有要素的设计规则。

12.3 “Use Case 实现”报告

概述

当分析和设计活动告一段落[2]时，相应的设计模型会形成特定的基线版本，此时可以通过一组“Use Case 实现报告”全面地展现分析和设计活动与需求中应用逻辑的对应关系。“Use Case 实现报告”描述参与实现相应局部需求（即 Use Case）的“设计元素”协作关系，包括动态与静态两种表述。

不同时期“Use Case 实现报告”的内容有所不同，这种差异往往有助于团队成员理解分析和设计活动的演进历程。

内容框架

- 简介。简要说明 Use Case 以及现阶段“Use Case 实现”的针对性。
- 已实现的事件序列。指出那些当前被实现的事件序列。
- 交互图组。展示“Use Case 实现”中的动态内容，主要是一组序列图。
- 参与对象列表。枚举交互图中出现的对象，根据需要作出简要说明。
- 参与类图（组）。展示参与“Use Case 实现”的“设计元素”静态内容。
- 派生需求。指出与该“Use Case 实现”有关、在设计模型中尚未涉及、但需要在实施活动中引起注意的其他需求内容。

12.4 设计模型纵览报告

概述

[1] 注意，这里只是一个概要的说明，更具体的内容通常会在“××语言编程指南”中详细说明。
[2] 在某一个迭代过程中。

设计模型中的核心内容将直接被后续活动沿用，不妨称这部分内容为狭义设计模型。结合本书实践过程所选用的构架模式，其具体含义就是层次构架，也是“设计模型纵览报告”所针对的内容。广义的设计模型中关于“复用经验”和“回溯需求”的内容分别展现在“设计指南”和“Use Case 实现报告”当中。

通过阐述组成狭义设计模型的“设计包”、“设计类”（涵盖“子系统接口”）以及它们之间的各种关系，“设计模型纵览报告”全面地勾画狭义设计模型的整体结构。

内容框架

以下是采用“层次”作为构架模式的“设计模型纵览报告”内容框架。采用其他类型的构架模式，该设计文档的大致结构类似。

- 简介。简要介绍狭义设计模型的内容。
- 模型总览。一方面，给出模型顶层的总览，列举构架中的层次（《layer》）以及展示它们之间关系的类图（组）。另一方面，给出“设计元素”的总览，按照指定的次序，列举全部“设计元素”及所在的位置（层次和包）。
- 模型分层结构。以自顶向下递归的方式，展示“设计包”的内容：包括包的名称、内含的“设计类”、内含的其他“设计包”以及描述上述内容间关系的类图（组）。

12.5 设计包报告

概述

“设计包报告”针对特定“设计包”的内容。设计人员可以通过该报告了解包的外部可见部分与不可见（私有）部分。通常在模型形成某一基线时获取相应版本的“设计包报告”。

子系统的内容也以“设计包”的形式组织在一起，因而“设计包报告”同样适用于描述特定的子系统。

内容框架

- 简介。简要介绍“设计包”的内容。
- 设计包中的图。展示“设计包”中的各种图示。
- 公开的“设计类”。全面列出“设计包”中具有“公开”可见度的“设计类”，枚举它们的操作，并给出简要说明。
- 私有的“设计类”。全面列出“设计包”中具有“私有”可见度的“设计

类 ”，枚举它们的操作，并给出简要说明。

12.6 设计类报告

概述

“设计类报告”针对特定“设计类”，展示设计模型的微观内容。“设计类报告”同样适用于描述“子系统接口”。

内容框架

- **简介。对“设计类”的简要说明。**
- **操作。说明操作的标识、相关的参数和返回值，并给出简要说明。**
- **属性。说明属性的名称和类型，并给出简要说明。**
- **与其他“设计类”的关系。主要说明当前“设计类”与其他“设计类”的泛化关系和不同强度的关联关系[□]。**
- **状态图。描述“设计类”的状态特征，有可能空缺。**
- **相关类图。枚举“包含”当前“设计类”的所有类图。**

[□] **注意，如果关联关系是单向的，那么只描述当前“设计类”到其他“设计类”的关联关系。**

附　录

附录 A　应用建模实践过程中的术语

该附录给出本书第二部分中使用的一些术语，主要针对 UML 应用建模实践过程的上下文，并不包括那些非常通用的术语。

边界类（Boundary）——用于描述拟建系统外部环境与内部运作之间的交互，主要负责内容的翻译和形式的转换，并表达相应的结果。

补充规约（Supplementary Specification）——涵盖那些不适宜记录在 Use Case 报告中的比较一般化的需求内容，是非功能需求的主体，也可能涉及少量功能需求。

层次构架（Layers Architecture）——是一种构架模式，适用于中、大型系统的分析和设计。分层的基本原则是越靠下的层次中所包含的内容越具有一般（普遍）性，或者说与软件需求中特定应用逻辑的关系越松散。

词汇表（Glossary）——文档定义拟建系统开发项目中使用的重要术语。

迭代（Iteration）——是有计划和评价准则的各工种活动序列，本书指各项任务及活动的一次遍历，两次遍历之间可能还包括应用建模之外的活动。

分析机制（Analysis Mechanism）——是构架机制的概念层面表述。在分析过程中，分析机制向设计人员提供复杂行为的简明标识，降低（全局）分析活动的复杂性并提高（局部）分析活动的一致性。

分析局部——即指“ Use Case 实现 ”。

分析类（Analysis Class）——是概念层面的内容，与应用逻辑直接相关。“ 分析类 ” 的实例所具备的行为，用于捕获拟建系统对象模型的雏形。

分析元素（Analysis Element）——即指“ 分析类 ”。

构架机制（Architectural Mechanism）——是解决常见软件技术问题的设计经验描述，包括模式化的结构特征和行为特征。在不同的任务阶段有特定的表现形式，参见“ 分析机制 ”、“ 设计机制 ” 和“ 实施机制 ”。

构架模式（Architectural Pattern）——是经过人们反复实践、总结和提炼的设计方案的宏观组织形式。不同的构架模式各有特色和针对性，不同构架模式的应用场合之间并不互相排斥。

关键抽象（Key Abstraction）——是业务需求和软件需求中所揭示的拟建系统必须处理的核心概念，这些概念同样将成为设计模型中的核心要素。

过程——本书中的“ 过程 ”仅指应用 UML 进行面向对象分析和设计（OOAD）的过程，即 UML 应用建模实践过程，可被看作为 RUP 中 Analysis and Design Workflow 的一个应用实例。

核心设计元素——核心设计元素指由“ 分析类 ”映射而来的“ 设计类 ”或“ 子系统接口 ”。参见“ 设计类 ” 和“ 子系统接口 ”。

活动（Activity）——是针对某一角色的工作单元，由若干步骤组成。本书应用建模实践过程的 5 项任务中包含 14 个活动。

基础设计元素——基础设计元素是外围设计元素的一部分，是那些已经被前人定义好的类，这些类有秩序地存在于各个类库中，它们与应用逻辑不直接相关。参见“外围设计元素”。

角色（Role）——定义了个人或协同工作的小组的行为与职责，即负责完成那些活动（Activities）和获取何种工件（Artifacts）。角色、活动和工件是 RUP 内容在微观层面的基本结构和组织思想。

可见度（Visibility）——针对不同的客体有不同的语义：操作和属性的可见度（Operation Visibility and Attribute Visibility）是指一个类的操作和属性对于另外一个类的公开程度。按照越来越不公开的趋势，它们是“公开（public）”、“受保护（protected）”、“私有（private）”和“实施（Implementation）”。类的可见度（Class Visibility）是指一个包内的类对于该包以外的类的公开程度。有“公开（public）”和“私有（private）”两种。连接可见度（Link Visibility）是指一个对象到另外一个对象之间的通信连接（Link）的不同类型。包括“全局（Global）”、“参数（Parameter）”、“局部（Local）”和“域（Field）”四种情形。

控制类（Control）——用于描述一个 Use Case 所特有的事件流控制行为。

任务——是一组目标明确和关系紧密的活动。本书应用建模实践过程由 5 项任务组成，依次为“全局分析”、“局部分析”、“全局设计”、“局部设计”和“细节设计”。前两项任务以分析为核心，后三项任务以设计为核心。本书的“任务”概念类似于 RUP 中的 Workflow Details。参见“过程”和“活动”。

设计机制（Design Mechanism）——是构架机制在逻辑层面的具体表述。“设计机制”通过一组参数化的类的协作关系实现相应的“分析机制”。

设计类（Design Class）——承担那些对应于独立构件的“分析类”之“责任”集合。

设计模型（Design Model）——广义的设计模型包括在分析和设计活动中得到的所有结果。本书中的设计模型包括三方面内容，参见“Use Case 实现”、层次构架和“构架机制”。

设计师（Designer）——设计师的工作对象通常是系统的局部或者细节，设计师负责局部性的分析和设计问题以及细节性的设计问题。

设计元素（Design Element）——是能够直接用于指导实施（编码）的设计模型要素。

实施机制（Implementation Mechanism）——是构架机制在物理层面的表现形式。“实施机制”运用特定的实施技术实现相应的“设计机制”。

实体类（Entity）——代表拟建系统中的核心信息，同时包括相关的行为。

适用范围（Scope）——说明类的操作和属性属于类的各个实例或者属于该类的全部实例所共有。

Use Case 报告（Use Case Report）——是针对某一 Use Case 的文字描述，包括围绕实现特定 Actor 某一目标的一组事件序列，以及必要的辅助图文信息。

Use Case 实现（Use Case Realization）——作为设计模型的一部分，描述一组对象的协作关系，用于实现特定 Use Case 表述的软件需求。“Use Case 实现”是

功能需求向设计方案（某种构架中的内容）过渡的核心纽带。

外围设计元素——与拟建系统的应用逻辑不直接相关，它们辅助“核心设计元素”具体地利用“分析机制”所概括的基本服务。“外围设计元素”通常伴随“设计机制”及“实施机制”的逐步“落实”而被引入拟建系统构架中重用价值高的部分。参见“核心设计元素”，“衔接设计元素”和“基础设计元素”。

系统构架师（Software Architect）——负责领导和协调整个项目中的技术活动，系统构架师负责全局性的分析和设计问题。

衔接设计元素——是外围设计元素的一部分，在不同场合的形式和内容大同小异，用于辅助“核心设计元素”使用“基础设计元素”提供的服务。参见“核心设计元素”和“基础设计元素”。

消息（Message）——是对象之间传递信息的途径，用于呼叫对象的对外服务。

责任（Responsibility）——是类或者子系统接口定义中承诺的对外响应能力。

子系统（Subsystem）——是一组设计元素的集合，其中一部分元素说明这组元素能够提供哪些行为，而另外一部分元素则具体提供相应的行为。

子系统接口（Subsystem Interface）——定义了那些对应于复合构件（即子系统）的“分析类”之“责任”集合，“子系统接口”将拟建系统中相对复杂的一部分任务封装为完整的功能单元。

附录 B　应用建模实践过程的快速参考图述

总体框架

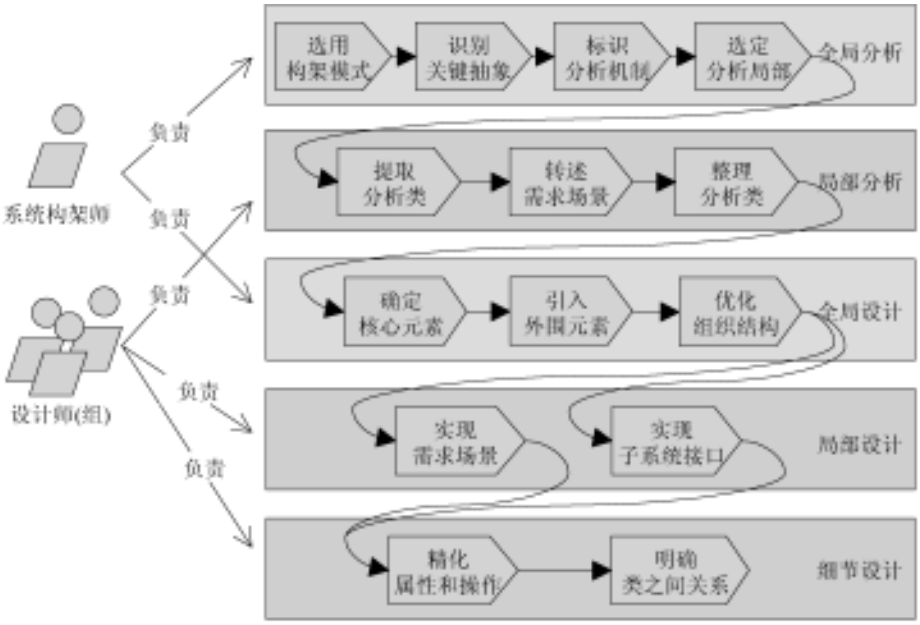


图 B-1　应用建模实践过程的框架

迭代策略

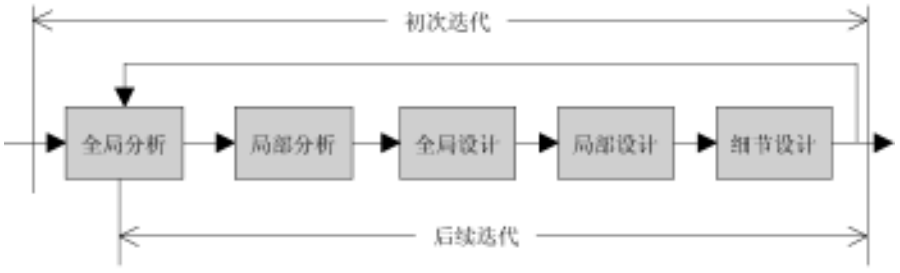


图 B-2　可迭代的的应用建模实践过程

全局分析

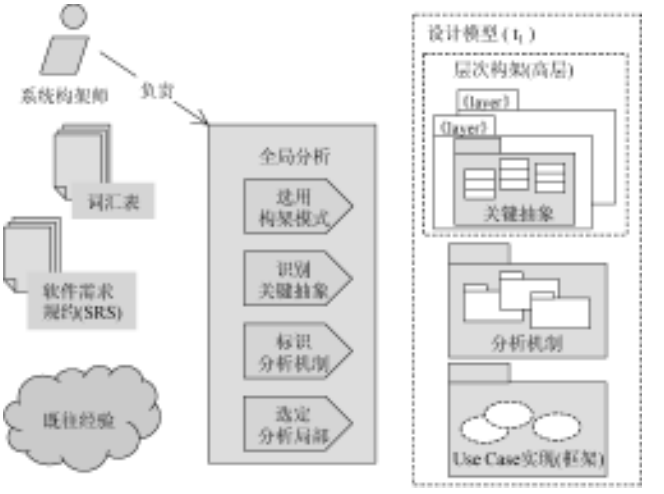


图 B-3 **“全局分析”任务的责任人—依据—活动—结果**

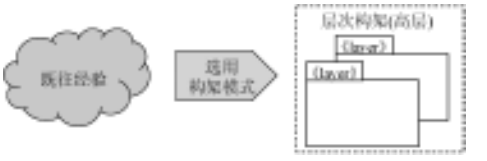


图 B-4 **“选用构架模式”活动图示**

图 B-5 **“识别关键抽象”活动图示**

图 B-6 **“标识分析机制”活动图示**

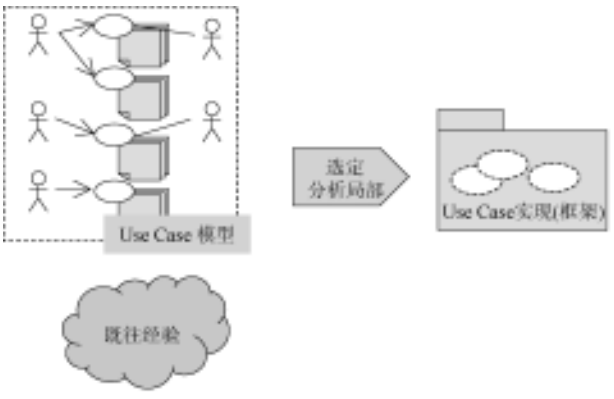


图 B-7 **“选定分析局部”活动图示**

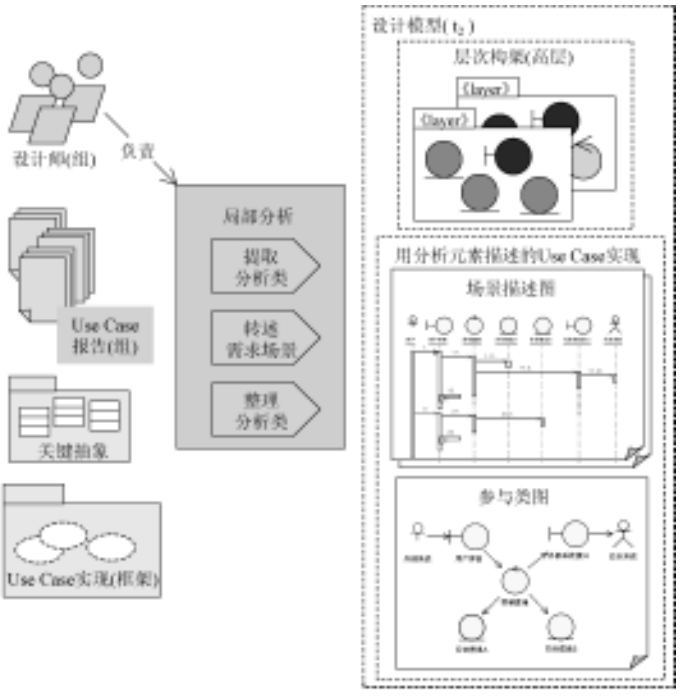


图 B-8 **“局部分析”任务的责任人—依据—活动—结果**

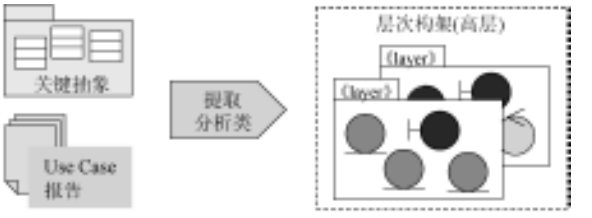


图 B-9 **“提取分析类”活动图示**

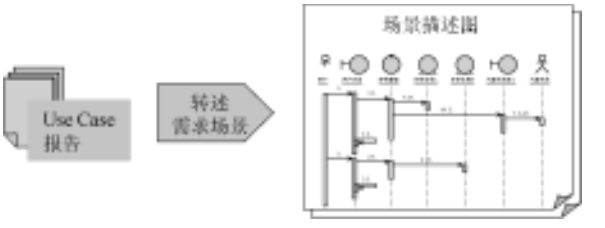


图 B-10 **“转述需求场景”活动图示**

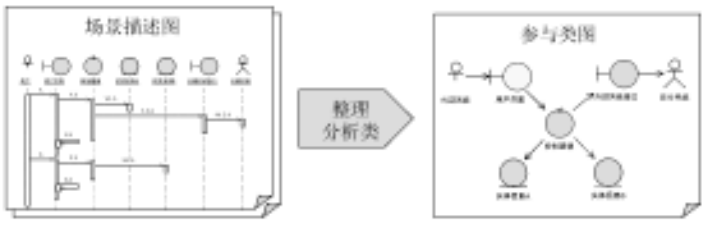


图 B-11 **“整理分析类”活动图示**

全局设计

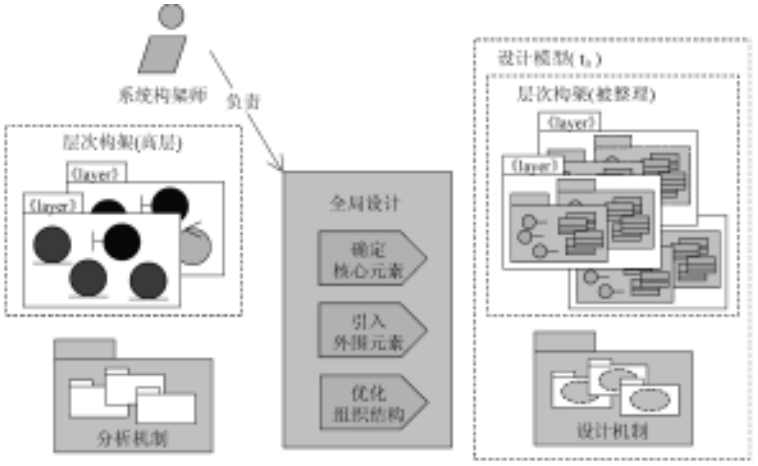


图 B-12 **“全局设计”任务的责任人—依据—活动—结果**

图 B-13 **“确定核心元素”活动图示**

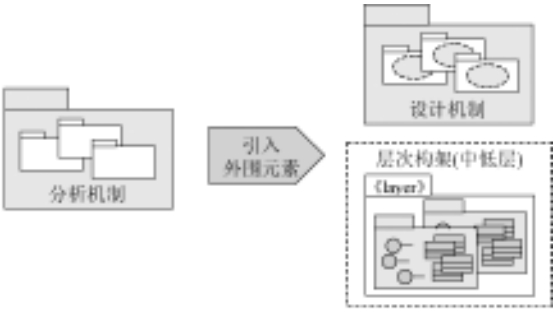


图 B-14 **“引入外围元素”活动图示**

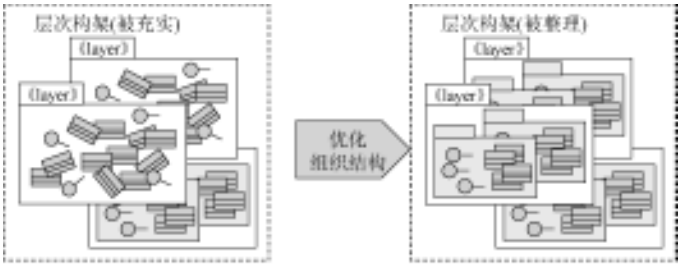


图 B-15 **“优化组织结构”活动图示**

局部设计

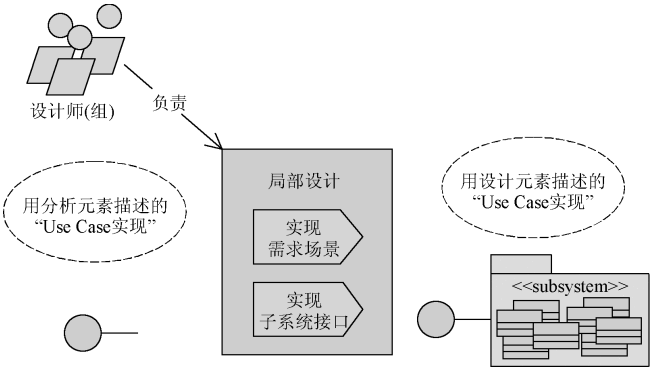


图 B-16 **“局部设计”任务的责任人—依据—活动—结果**

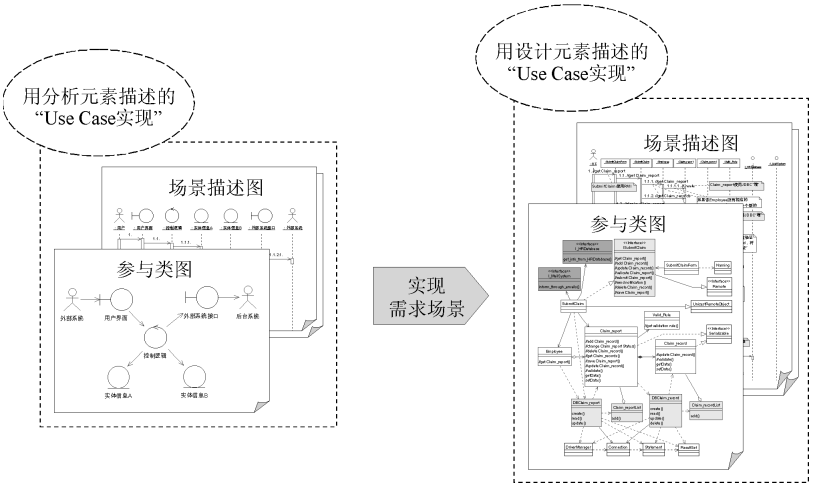


图 B-17 **“实现需求场景”活动图示**

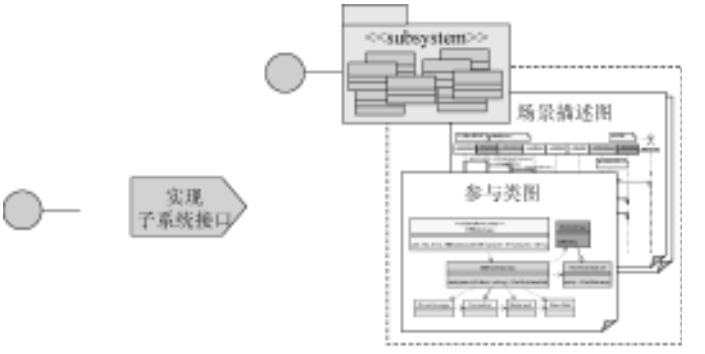


图 B-18 **“实现子系统接口”活动图示**

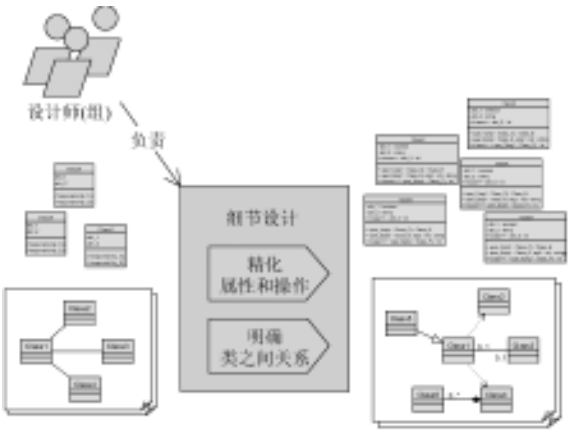


图 B-19 **“细节设计”任务的责任人—依据—活动—结果**

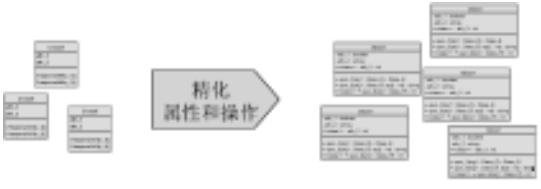


图 B-20 **“精化属性和操作”活动图示**

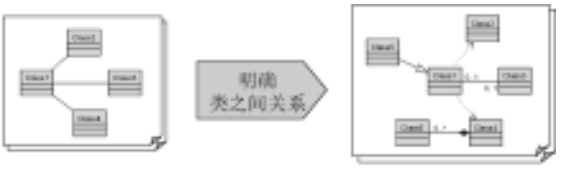


图 B-21 **“明确类之间关系”活动图示**

附录 C　UML 用于数据建模的模型元素构造型[1]

数据库的概念	UML 的基本模型元素	用于数据建模的构造型（Stereotype）
Database 数据库	Component 构件	<<Database>>
Schema 数据库（结构）模式	Package 包	<<Schema>>
Table／Entity 表／实体	Class 类	<<Table>>
Domain （数据类型）域	Class 类	<<Domain>>
Relationship 关系	Association 关联关系	<<Non-Identifying>>
Strong Relationship 强关系	Composite/Aggregate 组合/聚合	<<Identifying>>
Index 索引	Operation 操作	<<Index>>
Primary Key Constraint 主键约束	Operation 操作	<<PK>>
Foreign Key Constraint 外键约束	Operation 操作	<<FK>>
Unique Constraint 唯一性约束	Operation 操作	<<Unique>>
Check Constraint 检查约束	Operation 操作	<<Check>>
Trigger 触发器	Operation 操作	<<Trigger>>
Stored Procedure 存储过程	Utility Class 工具类	<<SP>>

[1] Rational Software《Using Rose Data Modeler》。

参 考 文 献

1 Barryw．Boehm Anchoring the Software Process．IEEE Software，July 1996

2 Craig Larman．Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design，Prentice Hall Press，1997

3 Deepark Alur，Jogn Crupi and Dan MalksCore J2EE Patterns：Best Practices and Design Strategies．Prentice Hall Press，2001

4 Erich Gamma，Richard Helm，Ralph Johnson and John Vlissides．Design Patterns：Elements of Reusable Object-Oriented Software．Addison Wesley Longman，1994

5 G. Booch，J. Rumbaugh， I. Jacobson．UML User Guide．Addison Wesley Longman，1998

6 Ivar Jacobson，Grady Booch，and James Rumbaugh．The Unified Software Development Process．Addison Wesley Longman，1998

7 J. Rumbaugh，I. Jacobson，and G. Booch．UML Reference Manual．Addison Wesley Longman，1998

8 Jon Ellis，Linda Ho，Maydene Fisher．JDBC? 3.0 Specification，Final Release，Sun Microsystems，Inc，2001

9 Mary Shaw and David Garlan．Software Architecture：Perspectives on an Emerging Discipline．Prentice-Hall，1996

10 Meilir Page-Jones．Fundamentals of Object-Oriented Design in UML．Addison Wesley Longman，2000

11 Object Management Group．OMG Unified Modeling Language Specification．Version 1.3．Rational Software Corporation and Object Management Group Inc．1999

12 Philippe Kruchten．The Rational Unified Process，An Introduction．Second Edition．Addison Wesley Longman，2000

13 Rational Software．Fundamentals of Rational Unified Process Student Manual．Rational Software Corporation，2001

14 Rational Software．Object-Oriented Analysis and Design Using UML Student Manual．Rational Software Corporation，2001

15 Rational Software．Rational Unified Process v2002．Rational Software Corporation，2002

16 Rational Software．Using Rational Rose Data Modeler v2001A．Rational Software Corporation，2002

17 Rogger S. Pressman．Software Engineering：A Practitioner’s Approach．Fifth Edition．McGraw-Hill，2001

18 Sun Microsystems，Inc．Java Remote Method Invocation Specification．Sun Microsystems，Inc.1999

19 Sun Microsystems，Inc．Java Programming Language StudentGuide．Sun Microsystems，Inc.2000

20 **宛延闓，定海．面向对象分析和设计．北京：清华大学出版社，**2001

21 **汪成为，郑小军，彭木昌．面向对象分析、设计及应用．北京：国防工业出版社，**1994

22 Walker Royce，Software Project Management：A Unified Framework．Addison Wesley Longman，1998