CHAPTER 5

第 5 章

用例分析

通过用例模型可对系统的需求进行完整的规格说明，这个规格说明将作为后续分析和设计的依据。而如何继续后续的分析过程也并没有一个统一的标准，甚至可以回到老路上——采用传统的结构化方法进行分析和设计；当然更好的选择显然还是继续采用面向对象方法。然而，面向对象的分析设计方法也有很多，比如一些 20 世纪 90 年代初期的分析设计方法[①]。而且，即使采用 UML 作为建模标准，也存在很多不同的实践。如 RUP 中的分析设计工作流，MSF[②] 中的逻辑设计和物理设计过程，以及一些敏捷方法中的简单设计和重构等。这些方法的实践过程和使用 UML 模型的技能或多或少地存在着不同。本章的用例分析技术则是一种典型的利用 UML 进行面向对象分析的方法，其主要思想来源于 RUP 分析设计工作流中的分析阶段，并适当地借鉴了其他一些方法中的成功经验。

本章目标

分析的目标是定义为了满足需求模型中所描述的功能，系统内部应该有什么样的业务核心机制。用例分析技术则是一种已经得到广泛认可的面向对象分析方法。通过对本章的学习，读者能够了解分析的基础概念，掌握利用用例分析方法进行面向对象分析的基本过程和实践技能，并能够动手完成某一给定系统的分析模型的创建工作。

主要内容

（1）了解分析和分析模型的基本概念。

（2）掌握从用例模型开始的迭代开发方法。

（3）了解架构分析的基本内容。

---

① 典型方法如 Coad/Yourdon 的 OOA 和 OOD 方法、Booch 方法、OMT 方法、OOSE 方法等。这些面向对象的方法是 UML 前身，采用了一些特定的建模语言和工具。感兴趣的读者可以参阅相关的软件工程书籍。

② MSF，微软解决方案框架（Microsoft Solution Framework），是微软提出的一种产品开发方法，该方法将软件开发分为构思、计划、开发、稳定和部署 5 个阶段。其中计划阶段分为概念设计、逻辑设计和物理设计，分别对应的需求、分析和设计过程；其需求阶段也推荐使用用例建模，分析设计也推荐使用 UML 模型。

(4) 掌握利用顺序图构造用例实现的基本方法和技能。

(5) 掌握定义分析类图的细节。

5.1 理解分析

分析是为了满足需求模型中所描述的功能,探讨系统内部应该有什么样的业务核心机制的过程。与需求相同,分析阶段还是在业务域中进行的。不过与之不同的是,需求阶段是以用户的角度描述用户所要实现的目标;而分析阶段,则需要以开发团队的角度描述系统为了实现用户目标应该提供哪些核心业务元素和关系。分析是采用技术的观点对业务领域进行建模的首次尝试,这些建模元素将作为后续设计的基础。在面向对象的方法中,这些业务核心机制表现为相应的对象(类)以及它们的静态和动态关系。

5.1.1 从需求到分析

分析架起了需求和设计之间的桥梁,它填平了业务域和技术域之间的鸿沟。其核心思想就是将以用户视角描述的需求模型转化为以开发团队视角描述的分析模型,之后在分析模型的基础上做进一步设计,从而获得设计模型,并最终实现软件系统。通过分析模型,可以有效地防止开发团队在彻底理解问题之前设计错误的解决方案,从而保证设计模型的正确性。

在需求模型中,需求的主要载体是采用自然语言描述的用例文档,这对于设计实现来说显然是不够精确的。而分析阶段则需要采用一种建模方法对用例文档进行精确化的描述,这就是分析模型。在面向对象分析中,需求被转化成由系统处理的对象模型,以及对象的属性、行为和交互。这些对象存在于系统内或系统边界上,可以通过一个或多个接口来访问。当然,与软件系统中的实现对象不同,目前的这些对象大多数都来源于业务领域中的物理对象或概念。

相对于用例模型而言,分析模型应该是准确、完整、一致、可验证的系统模型。通过该模型可以保证开发团队正确地理解用户问题,并及时纠正和澄清需求模型中的错误。因此,在迭代开发过程中,分析与需求在某种程度上存在着很大的重叠。这两个活动相辅相成,为了澄清和找出任何遗漏或歪曲的需求,常常需要在需求之上做一些分析。同样,通过分析可以进一步挖掘新的需求。

5.1.2 分析模型

分析模型是对分析所形成的目标工件的总称;具体来说,分析模型包含两个层次的两类模型。

两个层次是指架构分析和用例分析。架构分析是从宏观上考虑软件系统基本组织结构,定义系统的备选架构;并明确系统中的一些关键问题,从而为后续的用例分析和架构设计提供支持。用例分析是一种具体的分析技术,通过分析需求模型中的每一个用例获得分析类,并描述分析类之间的交互,从而实现并验证用例所要达到的目标。

两类模型是指静态模型和动态模型。静态模型关注系统组成的静态组成结构,又称结构模型;可以采用 UML 包图、类图、对象图等静态结构图来描述。动态模型则关注系统组

成的动态行为特征，重点描述系统运行期间对象和对象之间的交互过程，又称交互模型[①]；可以采用 UML 顺序图、通信图等交互图来进行描述。

面向对象的分析过程就是围绕这两个层次的两类模型展开的。早期的面向对象分析方法并不重视架构分析，主要关注更低层次的用例分析，围绕用例建立静态和动态分析模型（即结构建模和行为建模）；而对架构的设计则推迟到设计阶段进行。不过随着构件技术和复用技术的广泛应用，架构在系统开发中扮演着越来越重要的作用。尽早地引入系统架构有助于规范分析和设计工作，并最大限度地提高系统的复用级别，建立高质量的软件。架构分析的目标就是定义系统的备选架构，该备选架构将在架构设计中进行进一步的定义和完善。从另一方面来说，架构也包括静态模型和动态模型两个方面，不过在分析阶段，侧重于静态特征的建模。其动态特征则只是通过架构机制的方式记录下来，并不进行具体的建模操作。

5.1.3 分析的基本原则

面向对象的分析是以对象的视角来理解业务问题。它不同于以自然语言描述的需求，也不同于以技术语言来表示的设计。对于分析人员来说，如何把握这种介于业务和技术之间的分析活动的“度”是非常关键的：过度地分析会陷入设计误区，从而难以有效地达到分析的目标；而不够深入的分析则容易遗漏那些重要的信息，从而无法及早发现并处理需求中的问题。

一般来说，对于中等规模和复杂度的系统，分析模型中大约存在 50 个～100 个分析类[②]。但是，每个系统的业务各不相同，也很难对分析活动进行归纳。因此，分析人员应该注意把握一些分析的基本原则，从而保证在分析活动的范围内进行有效的分析。这其中最重要的原则就是：当构造分析模型时，把整个活动限制在业务问题域词汇，而不考虑任何技术领域的实现策略；从而保持分析模型是一个对系统结构和行为的精确和简单的陈述。所有与实现技术相关的内容都留给设计和实现阶段来考虑。一些具体的分析原则如下所示。

- 分析模型应使用业务语言。分析模型中的类主要应该是业务领域的术语。
- 分析模型中类的细节和关系等应该是业务中明确存在的，不要刻意去细化或封装这些细节。如：只有在业务领域中，分析类之间存在明确的继承层次时才使用泛化关系；而不是考虑代码的复用或者是多态等特性而使用泛化关系；这些都是设计的工作。
- 分析活动是对需求模型的重新表述，是一种以理想化的方式来实现用例所描述的行为，并不考虑具体的技术实现。
- 分析侧重于系统的主要部分，关注核心的业务场景。对于那些支撑性行为、非功能需求等内容一般不做深入分析。
- 所有的分析类应该都是为项目涉众产生价值的。

在分析期间，除了遵循这些原则外，还可以复用一些成功的分析案例。Martin Fowler

---

① 动态建模过程也称为行为建模，但其目标模型被称为交互模型更合适。因为主要采用的是 UML 的顺序图和通信图等交互图来建模；而不是采用诸如活动图、状态图之类的行为图建模。

② 对于类似于本书中的案例的小规模系统，其分析类大约在 10 个～30 个左右，甚至更少。

将这些分析案例总结为分析模式来供分析人员使用[①]。分析模式(Analysis Pattern)是描述通用业务/分析问题解决方案的一种模式。下面以开发一套软件来模仿台球游戏为例来说明分析模式的作用。

该问题可能用用例来表述:“游戏者击中白球,它以一定的速度前进,并以特定的角度碰到红球,于是红球在某个特定的方向上前进一段距离”。该用例描述中给出的只是事物的表面现象,仅依靠该现象的描述显然无法写出好的仿真程序。因为除了这些表面现象,还必须了解背后的本质,那就是和质量有关的运动定律、速度、动量,等等。了解这些规律才能开发出正确的软件。

当然,该例子的特别之处在于,这些规律已经成为定理;因此,开发人员能够依据这些定理很好地实现系统。而从软件开发的观点来看,这些定理就是一些固有的分析模式。

从这个例子可以看出,当开发人员在建造应用系统的时候,需要进行大量的分析和研究,才能接触到问题的本质。为此,将其中的一些通用的问题进行总结复用,形成软件领域的公理:“模式”。这意味着在新系统开发中,凡是遇到类似的问题都可以遵循已经形成的模式来解决,而不需要重新分析和设计。从而大大提高软件开发的效率和软件自身的质量。分析模式是这类模式的一种,它主要关注分析域中的业务问题:如果可以利用以往的经验,得到业务领域的通用解决方案,它们将直接影响到应用系统的设计,因而这种复用的价值将更加显著。事实上,目前,有关权限、异常处理、组织机构等很多问题都有一些通用的分析模式可以复用。在系统分析期间,可以有意识地利用这些模式来实现更高层次的复用,从而进一步提高系统的质量。

5.2 从用例开始分析

分析的基础是需求,而需求的表现形式为用例,因此分析的过程完全是基于用例模型展开的,用例模型确定了分析模型的结构。从另一方面来讲,用例模型确定了系统的外观,即对外部参与者提供哪些价值;而分析模型则描述系统的内部机理,即为了提供这些价值,系统内部应该有怎样的对象,进行怎样的交互。图 5-1 描述了这种用例模型和分析模型之间的关系。

从图 5-1 中可以看出,需求阶段将系统封装成用例[②],而分析阶段将深入到用例内部,将这个“黑盒子”拆开,分析其内部的结构和行为,这就是分析的基本思路。不过,每一次分析并不一定直接针对用例模型进行全面的分析,而是还需要做一些准备的工作。主要包括两个方面工作:(1)重新规划和完善本次迭代需要分析的用例;(2)为需要分析的用例定义用例实现。

---

① 目前模式主要用于设计领域,有关设计模式的研究也比较成熟,读者也可以参考本书第 7 章的内容。分析模式的概念主要来自 Martin Flowler 的著作 *Analysis Patterns: Reusable Object Models*,该书提出了一些典型的可复用的分析模式。此处有关分析模式的介绍也源自该著作,读者可以参阅该著作以了解更多细节。

② 从这里也能够体现为什么在用例建模时反复强调用例自身的完整性,只有相对独立而且完整的用例,才可以分析其内部机理,也才能够顺利地开展后续的分析设计工作。如果在前期过分细化用例,实际上是对用例内部的分析过程,而这个分析本身又不是面向对象的分析,而是功能分解式的分析。

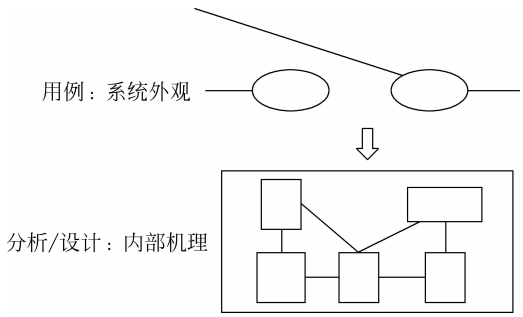

图 5-1 用例模型和分析模型的关系

5.2.1 用例驱动的迭代开发

随着软件项目规模的日益增加，传统的线性开发流程已被证明是无法满足要求的。因此在现代软件开发过程中，迭代开发的思想已经被广泛采纳，并且被证明是一种有效的应对大规模软件开发的策略。而迭代的基础就是需求，更确切地说就是用例模型。在迭代开发中，一般通过一次全面的用例建模迭代，获取并定义了系统的绝大多数需求后[①]，即可以开始分析设计的迭代工作。

1. 用例与迭代开发

由于在迭代开发中，为提高开发的效率，往往要求后一次迭代能够有效地复用前面迭代的成果，因此制定合理的迭代计划来规划每一次迭代是影响迭代开发是否成功的关键要素。其基本思想是通过早期的迭代明确用户需求，从而建立并证明系统的核心架构；而后期的迭代则以此架构为基础向系统的整体全面展开。一般来说，早期的迭代，主要关注以下内容。

- 对于用户来说，主要关注核心业务的主要部分。
- 对系统架构有重要影响的部分。
- 影响系统性能等其他关键非功能需求的部分。
- 存在高风险的部分，如需要采用新技术、新产品的部分。

从这里可以看出，此处有关早期的迭代所关注的内容与那些在用例分级时具有高优先级的用例是一致的。事实上，用例模型是制定整个迭代计划的基础，是它驱动了后续的分析设计工作，它的结构也决定了分析设计模型的结构。因此，早期的迭代就是针对那些高优先级的用例，从而尽早地建立和稳定软件的架构。

在开始分析之前，为了保证分析设计的正确性，首先需要评估用例模型，以确保所要分析的用例图、用例文档等需求载体是可靠的、一致的。这就是在很多实际项目开发中所要经历的需求评审活动。通过需求评审确定了可靠的需求后，就可以着手进行用例的分级工作。通过评估用例的重要性、开发风险等技术因素，并结合项目组的相关技能情况来抽取高级别的用例，从而开始早期的分析设计迭代。

---

① 一般至少获取 60%～80%的需求，而剩下的部分主要是那些支撑性需求，这些需求会在后续迭代中定义。当然在这次需求迭代中也会进行一些简单的分析、设计，甚至实现系统原型等工作。

迭代是基于用例的，因此每个迭代周期也完全可以通过用例来定义。一个迭代周期要被指派一个到多个用例。对于复杂的用例，如果完整版本在一个迭代周期中处理起来太复杂的话，可以采用其简化版本的用例，该简化版用例不考虑其中的一些主要路径，而忽略那些次要的、异常的路径；完整版本的用例将在后续迭代周期中处理。图 5-2 给出了一个基于用例的分级技术定义迭代周期的示例。

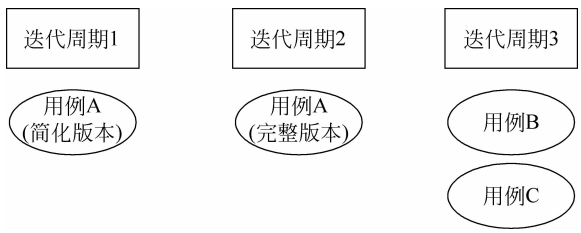

图 5-2　基于用例的分级定义迭代周期

图 5-2 中反映了某个具有三个用例(即用例 A、用例 B 和用例 C)的系统。其中，用例 A 为系统的核心业务，优先级较高，而且其内部事件流比较复杂；而用例 B 和用例 C 则是支撑性用例。为此，在最开始的迭代周期 1 中，只处理用例 A 即可；同时考虑用例 A 的复杂性，该周期内只考虑用例 A 的基本事件流和相关场景；通过对“用例 A(简化版本)”的分析和设计来定义系统的备选架构。之后，在该架构的基础上，通过下一次迭代再考虑用例 A 的分支、异常等备选事件流，从而完成用例 A 的全部内容。最后再通过第 3 个迭代完成用例 B、用例 C 的工作，从而完成整个系统的开发。

一个迭代周期一般在 2～4 周的时间内完成，这样便于及时发现并处理开发中所面临的问题；而且对于严重的需要返工的问题也可以节省时间[①]。因此，为了能够顺利地完成每一次迭代，每个迭代周期都不会被指派过多的用例。

2. 利用早期迭代建立软件架构

早期的迭代非常重要，其目标就是建立和证明软件的核心架构，这个架构将直接决定系统是否能够顺利地推进后续的迭代开发。然而，并没有一些明确的标准来确定哪些用例具有高优先级，以至于需要在早期迭代中完成。这也是整个迭代开发中的难点所在。针对这个普遍性的问题，Jacobson 博士一次在中国举行的演讲中给出了这样一个实例，来说明怎样的早期迭代是成功的，这也给我们提供了一个定义早期迭代的思路。图 5-3 是某系统的最终软件架构[②]，而这个架构是经过若干次架构分析和设计的迭代才能完全定义出来的。

从该架构图中可以看出，该软件架构总共分为四层：Application、Business Components、Persistence Framework、Infrastructure。每个层次内部又分为若干个子包和子系统，如 Infrastructure 层中有 User Interface Routines 包和 General Routines 包以及 Event Handling Routines 子系统和 Communication Routines 子系统。

---

① 如果定义了一个需要半年才能完成的迭代周期；而当该迭代周期完成后，发现所迭代的内容无法在后续迭代中得到有效的复用；那么这半年的工作可能都需要返工。这显然是不能接受的。因此，一般的迭代周期都不会太长，有些甚至只需要 1 周的时间。这样可以使得项目组及时评估自己的工作，从而可以对出现的问题做出快速反应。相对来说，策划一系列的小胜利和接受一些小的失误总要好一点；而策划一个巨大的胜利经常会导致灾难性的失败。

② 为了保证图形的清晰，我们省略了多数架构层以及内部包之间的依赖关系，以及相关子系统接口的定义。

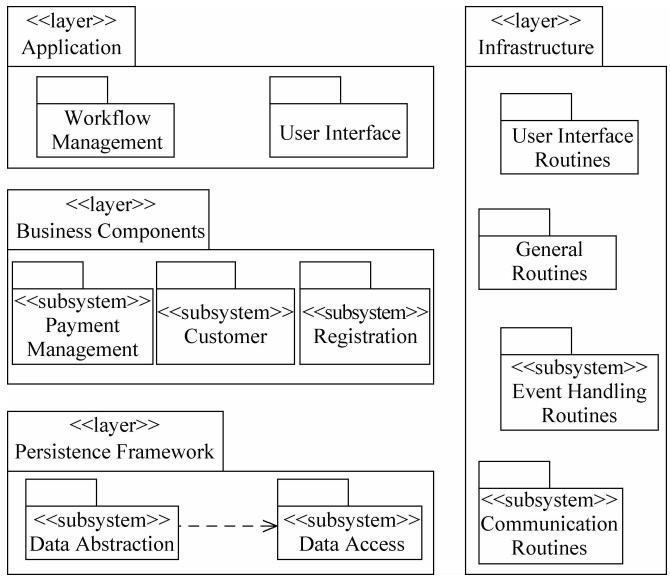

图 5-3 某系统软件架构

那么，针对该系统而言，成功的早期迭代应该能够及时发现并定义系统的这四层结构，并尽可能地明确每个层次内部的包和子系统结构。图 5-4 反映了通过早期迭代定义系统架构的示意图。

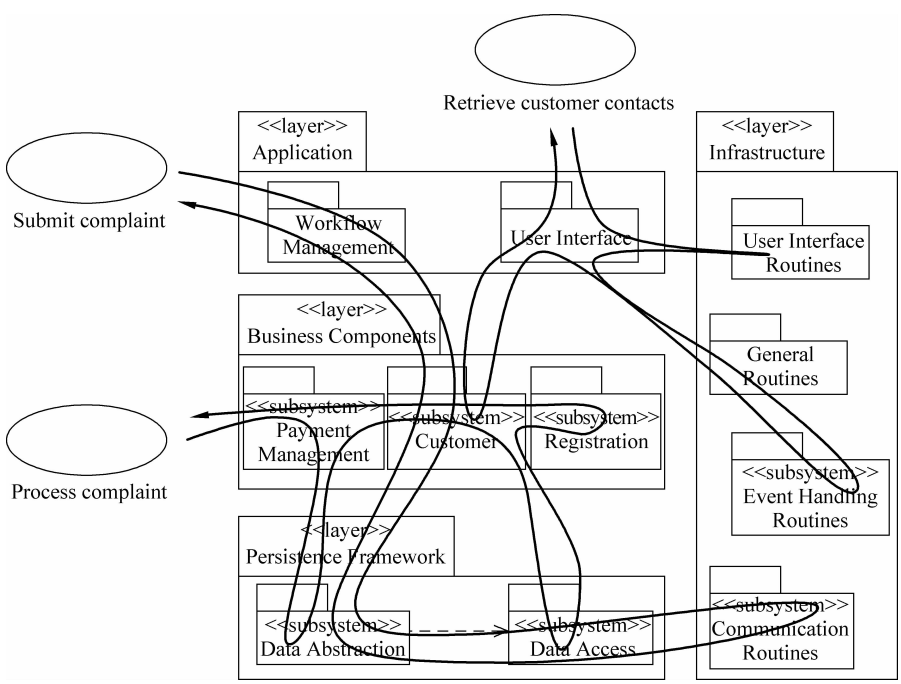

图 5-4 通过早期迭代定义系统架构

图 5-4 中展示了本次迭代所开发三个用例：Retrieve customer contacts、Submit complaint 和 Process complaint。而每个用例附近手绘的实线表明该用例中所获取的类在

架构中的位置。从中可以看出，这三个用例基本覆盖架构的全部内容。其中从 Retrieve customer contacts 用例中发现的类和行为覆盖了 Application 层的 User Interface 包、Business Components 层的 Customer 子系统以及 Infrastructure 层的 User Interface Routines 包、General Routines 包和 Event Handling Routings 包。从 Submit complaint 用例中发现的类和行为覆盖了 Application 层的 Workflow Management 包、Business Components 层的 Customer 子系统及 Persistence Framework 层的 Data Abstraction 子系统和 Data Access 子系统，以及 Infrastructure 层的 Communication Routines。而从 Process Complaint 用例中获得的行为覆盖了 Business Components 层的 Payment Management 子系统、Customer 子系统和 Registration 子系统以及 Persistence Framework 层的 Data Abstraction 子系统和 Data Access 子系统。

这意味着，仅通过这三个用例的一次迭代，就可以建立目标系统的最终架构，而更多的用例行为只不过是对该架构的进一步填充，不会带来新的架构元素。因此，后续迭代将不会对系统结构造成很大的冲击，迭代开发的成果可以被很好地复用，在提高开发效率的同时也保证了系统自身的稳定性。

当然，问题的关键还是落在如何能够准确找到可以定义系统架构的核心用例，这与具体的业务和开发团队的技能紧密相关。Jacobson 博士给出了这样的参考数据：一般来说，通过一次系统的用例建模过程来获取系统 60%～80%的需求后，从中找出 5%～10%的重要用例，来定义系统的架构。

下面将以本书中的旅店预订系统和旅游申请系统为例，来进一步考虑如何定义该系统的第一次分析设计迭代，通过该次迭代要完成系统架构的定义。书中后续的分析和设计也主要是基于这次迭代的。

3. 旅店预订系统的首次迭代

旅店预订系统的核心业务是处理旅店房间预订和取消业务，结合表 4-23 对该系统的用例分级，可以发现该系统最核心的用例就是“预订房间”，显然该用例是这次迭代的重点。对于“取消预订”用例，也是整个系统中一个重要的分支流程，不过显然取消中所涉及的业务实体在预订中都有所体现，因此该用例对架构影响不大(这也是为什么用例的优先级评为中的原因)；但考虑到该用例中涉及有关退定金等重要的业务逻辑，因此最好在首次迭代中实现。

此外，“调整价格”则完全是支撑用例，对主流程、业务规则都没有太多的影响，因此不需要在本次迭代中实现，它可以很容易在以后的迭代中递增到系统中。最后，“登录”用例由于涉及与安全性相关的问题，因此其优先级被评为中；但考虑该系统只是一个单机版系统，而且由于本次迭代并不实现经理参与的“调整价格”用例，这意味着本次迭代后的系统只有服务员一类用户，因此完全可以不需要登录功能，也就不需要实现登录用例。

结合对用例分析和架构定义的考虑，旅店预订系统的首次迭代将只实现两个用例：“预订房间”和“取消预订”。而且由于这两个用例的复杂度都有限，所以在本次迭代中将完全实现。图 5-5 给出了本系统本次迭代所要完成的用例图。

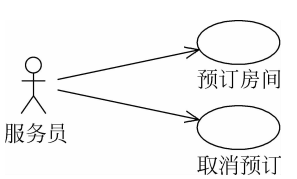

图 5-5 “旅店预订系统”首次迭代周期的用例图

4. 旅游申请系统的首次迭代

对于旅游申请系统，因为该系统的核心业务是为旅客办理申请旅游团相关的手续，所以首次迭代所实现的系统应该能够满足这项基本需求。整个办理的基本流程是这样的：前台服务员首先通过“办理申请手续”用例为旅客完成基本的旅行申请，之后通过“管理参加人”用例来添加需要参加本次申请的人员信息，最后通过“完成支付”用例来记录申请的支付信息。此外，在申请期间，收款员工还需要通过“打印旅游确认书和余额交款单”用例来完成打印业务，并邮寄给申请人。因此“办理申请手续”、“管理参加人”、“完成支付”和“打印旅游确认书和余额交款单”这 4 个用例都应该出现在本次迭代中。

虽然“管理参加人”用例内部包括增加、修改和删除参加人三个独立的子流程，而对一次正常的申请业务来说，只需要增加参加人就可以满足基本需求。而且修改和删除业务也是基于增加的参加人来进行的，这意味着这两个子流程并不会带来新的分析类。因此，在本次迭代中，可以只实现该用例的“增加参加人”子流程，其他子流程可以在后续迭代中递增实现。

另外，在用例分级时，用例“导出财务信息”也有很高的优先级，这是因为它由系统自动启动，并且涉及外部财务系统。显然，这两种机制对系统架构是有很大影响的，因此该用例也应该在本次迭代中。此外，由于本次迭代所实现的用例有两类参与者（前台服务员和催款员工），系统必须对用户进行身份认证，因此“登录”用例应该在本次迭代中实现。

与路线管理员相关的用例都是一些支撑性的管理、维护用例，而“取消申请”用例也是相对独立的分支流程。因此，这些用例都不在本次迭代的范围之内。

图 5-6 给出了旅游申请系统首次迭代周期的用例图。在本次迭代中面向两类用户实现 6 个用例，其中“管理参加人”用例只实现其中的一个子流程。不过，这个细节在当前用例图中并没有体现出来，将在第 5.2.2 节的用例实现中给出更清楚的表示方法。

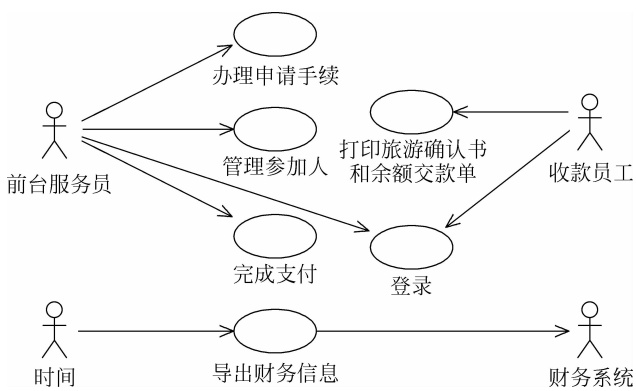

图 5-6 “旅游申请系统”首次迭代周期的用例图

此外，还需要注意的是，在第 4.5 节的重构用例模型中曾经为本系统中的某些用例引入了一些子用例。如：“办理申请手续”用例包含了“查询申请信息”用例，“导出财务信息”用例包含了一个“记录日志”的扩展用例。但在本次迭代中并没有利用重构后的模型。这主要是考虑引入用例关系后，会给分析和设计添加不必要的复杂性；因此，一般首次迭代中不处理这些关系，而是将子用例合并到主用例考虑或者根本不考虑子用例行为。本书将在

第 5.4.5 节单独介绍如何在分析设计期间处理这些用例关系。

当然，迭代周期并不是简单地利用一幅用例图就可以表示清楚的，该用例图只是界定了本次迭代的范围。为了能够明确定义本次迭代的内容，需要针对每一个待实现的用例进行进一步的分析，对其用例文档、用例间的关系在分析阶段都需要进行重新评估和定义，并最终构造其用例实现。

5.2.2 用例实现

定义本次需要迭代的用例后，就可以针对这些用例开始进行分析。不过用例是用来表示需求的，并不包含分析的要素。因此，针对这些待分析的用例，需要重新定义一个概念来表达，这就是用例实现。

1. 基本概念

用例实现(Use-Case Realization)是分析(设计)模型[①]中一个系统用例的表达式，它通过对象交互的方式描述了分析(设计)模型中指定的用例是如何实现的。通过用例实现将用例模型中的用例和分析(设计)模型中的类以及交互紧密联系起来，一个用例实现描述了一个用例需要哪些类来实现。

在 UML 2 中，用例实现采用协作来建模；协作的符号是虚线的椭圆。需要注意的是，早期的 UML 1.x 版本中协作主要是指协作图，并不能完全表示用例实现。因此，如果使用那些不支持 UML 2 的建模工具，则需要采用构造型≪use-case realization≫对用例进行扩展，从而正确地建模用例实现，其图形符号也可能受建模工具的影响而有所不同[②]。

分析模型中的用例实现是针对用例模型中的用例来定义的，这两者之间存在实现关系："用例实现"实现"用例"。在分析建模时，可以把这两者之间的关系通过类图展示出来，从而建立了分析模型和用例模型之间的跟踪关系。图 5-7 显示了用例实现和用例之间的实现关系，图 5-7(a)为来自用例模型中的用例，图 5-7(b)为来自分析模型中的用例实现；这两者之间存在实现关系。实现关系采用带三角形箭头的虚线表示，其中箭头指向被实现方。

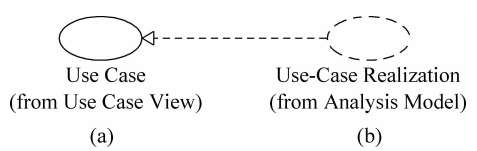

图 5-7 "用例实现"实现"用例"

2. 建模指南

用例实现是分析的要素，应该放在分析模型中。而按照 UML "4+1"视图的要求，分析模型是面向分析设计人员描述软件结构和行为的，属于逻辑视图。因此，利用 Rational Rose 进行分析建模的步骤如下所示。

(1) 首先，在逻辑视图中单击鼠标右键，在弹出的快捷菜单中选择 New|Package 命令新建一个包，将该包重命名为 Analysis Model 来表示分析模型。

---

① 分析和设计阶段都需要定义用例实现，其含义相同，只是内部处理细节不同。

② Rational Rose 虽然不支持 UML 2 建模，但却支持≪use-case realization≫构造型，并可提供正确的图符。

(2) 其次,在 Analysis Model 包,以同样的方式新建 Use-Case Realization 包,用来存放所要分析的用例实现。

(3) 再次,在 Use-Case Realization 包中定义每一个用例实现。由于 Rose 中用例实现是基于用例的扩展,因此实际操作时是在 Use-Case Realization 包上单击鼠标右键,在弹出快捷菜单中选择 New|Use Case(见图 5-8),将该用例重命名为"＊＊＊—用例实现",其中"＊＊＊"表示需求中的用例,如"预订房间—用例实现";再双击该用例,在弹出对话框中修改该用例的构造型为 use-case realization,从而将用例表示成用例实现(见图 5-9)。

图 5-8 在用例实现包中新建用例

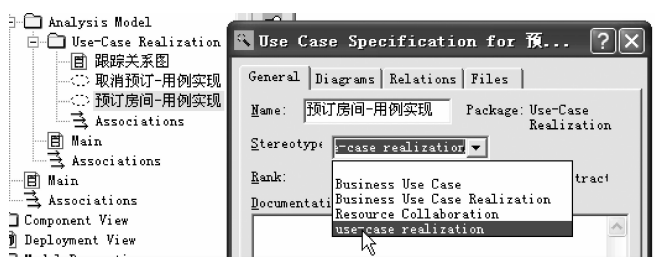

图 5-9 将用例实现包中的用例设置成用例实现

(4) 所有的用例实现都定义完成后(如图 5-9 的左侧资源浏览区中已经定义了该系统的两个用例实现),最后在 Use-Case Realization 包中新建一类图,重命名为"跟踪关系图",利用该类图建立这些新定义的用例实现和用例模型中的用例之间的实现关系。图 5-10 给出了旅店预订系统首次迭代的跟踪关系图。该类图可以明确地反映该系统首次迭代要实现两个用例,以及在分析模型中定义了两个用例实现,这两个用例实现分别实现对应的用例。注意,该图中的用例和用例实现并不需要从工具栏中新建,而是直接从左边的资源浏览区中将它们拖放到图中。

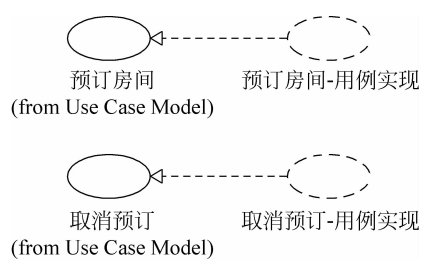

图 5-10 "旅店预订系统"首次迭代的跟踪关系图

对于旅游申请系统,也首先按照前面讲述的步骤建立分析模型的结构,并根据首次迭代所要完成的 6 个用例,定义相应的 6 个用例实现。该系统的首次迭代跟踪关系图如图 5-11 所示。

对于该系统中的用例实现,需要注意一点的是,由于本次迭代只处理"管理参加人"用例的"增加参加人"子流程,因此其对应用例实现为"增加参加人-用例实现"。这意味着,在后续的迭代中还会增加一些新的用例实现来实现该用例的其他子流程,即有多个用例实现来

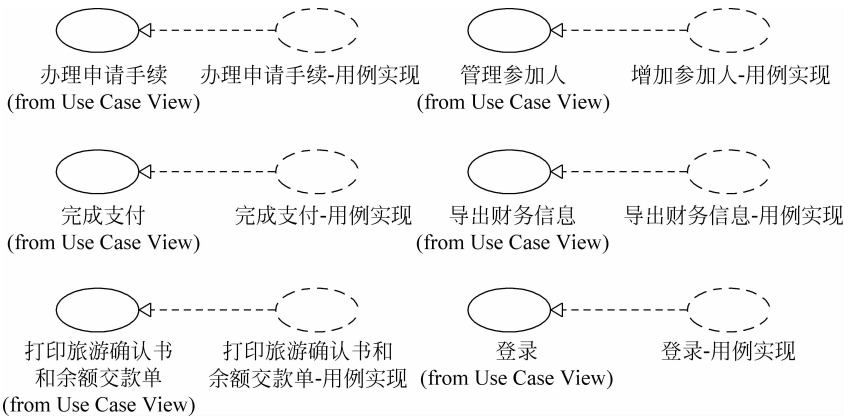

图 5-11 “旅游申请系统”首次迭代的跟踪关系图

实现一个用例；这也是一种通过对用例进行分解来简化分析的技巧。

5.3 架构分析

架构(Architecture)在面向对象的系统中扮演着越来越重要的角色，从某种意义上来说，面向对象的分析和设计都是以架构为中心进行的。在分析和设计的不同阶段，软件系统的架构被一步步细化和完善，最终形成一个规范的、稳定的、符合设计要求的架构模型。

每个软件系统都有架构，其结构可能简单也可能复杂；然而这个架构并不是通过简单的分析设计就可以完全定义出来的。架构也需要经过若干次分析和设计的迭代来完成。在早期的迭代中，架构设计师通过选取那些高优先级的用例，并结合系统的规模和类型，在选择合适的备选架构的基础上，经过几次分析和设计迭代来建立系统的基础架构。

在基础架构的基础上，开发团队内部即可针对架构内的组件进行进一步的细化设计。开发团队利用架构设计和构件设计的相关技术来分解和完善架构内容。可以通过架构机制、抽取子系统等具体的手段来不断优化架构。针对架构的分析和设计将贯穿整个分析和设计过程。本节将重点关注架构分析的内容，有关架构设计的内容我们将在第 8 章中进行进一步介绍。

架构分析的过程就是定义系统高层组织结构和核心架构机制的过程。从具体活动上来说，架构分析主要包括以下 4 方面内容。

(1) 定义系统的备选架构来描述系统的高层组织结构，以用例组织后续的分析模型。

(2) 确定分析机制以记录系统中的通用问题。

(3) 提取系统的关键抽象以揭示系统必须能够处理的核心概念。

(4) 创建用例实现来启动用例分析。这部分内容已经在第 5.2.2 节中阐述过。

5.3.1 备选架构

在早期迭代中，架构分析的主要目标是建立系统的备选架构，以用于组织后续的用例分析所获得的分析模型，该备选架构通过架构设计进行细化和调整，从而获得软件系统的基础架构。由于现阶段对系统缺乏足够的认识，因此不可能快速地定义目标软件的架构。为此，

早期迭代中的架构分析一般参照现有的、成功的架构模式来定义。

1. 备选架构模式

架构模式是那些在开发过程中积累下来，并经过实践验证行之有效的、可复用的软件架构。在架构分析中，可以直接借鉴已有的或类似项目中积累的经验，可套用某种成熟的架构模式的全部或部分内容。

Frank Buschmann 在 *Pattern-Oriented Software Architecture*，*Volume I*：*A System of Patterns*（《面向模式的软件架构——模式系统》）一书中给出了架构模式的定义："架构模式表示了对软件系统的一个基础结构组织形式。它提供了一套预定义子系统，详细说明它们的职责，并且包括组织它们之间的规则和指南"。表 5-1 给出了该书中所描述的几种典型的架构模式，并总结了其特点和用途。

表 5-1 典型的架构模式

软件类别	架 构 模 式	特点和用途
系统软件	层（Layer）	将系统划分为不同的抽象层次，每个抽象层次封装不同层次问题的对象，以处理系统不同方面的问题
	管道和过滤器（Pipes and Filters）	关注系统数据流的处理，数据通过管道流入不同的过滤器进行处理，从而获得最终的结果
	黑板（Blackboard）	面向数据结构的处理策略，系统由中心数据结构和相互独立的处理构件组成，通过构件来操作数据
分布式软件	客户/服务器（Client/Server）	在传统的客户/服务器模式中，服务器负责监听并响应客户端请求，客户端主动连接到服务器
	经纪人（Broker）	客户和服务器通过相应的代理进行通信，通过经纪人来协调客户和服务器之间的操作
	点对点（Peer to Peer）	各节点之间处于平等地位，可以互相连接；此外，一般有一些中心节点来负责发现和管理节点资源
交互式软件	模型-视图-控制器（Model-View-Controller）	是一种典型的交互式软件架构模式，将软件抽象成模型、视图、控制器三类构件，从而能够有效地分离用户界面和业务逻辑，以适应需求的变化
自适应软件	反射（Reflection）	将应用程序分成元层次和基本层次两个部分；元层次提供系统属性的相关信息，而基本层次包括应用程序逻辑，其实现建立在元层次之上；从而可以动态地改变基本层次以满足系统结构和行为的变更
	微核（Microkernel）	将最小功能核心与扩展功能和特定客户功能分离出来，以提供一个"即插即用"的软件环境；使得用户很容易连接扩展部分并把它们与系统的核心服务集成在一起；主要应用于对不同平台具有高度适应性并能满足客户定制需求的系统

2. B-C-E 三层架构

传统的结构化方法强调自顶向下的功能分解，因此在架构设计期间也侧重于对软件功能模块的划分。而随着软件规模的日益庞大，软件模块之间的关系也变得错综复杂；因此这种简单的横向模块分解策略并不是软件架构的好的组织方式。特别是在分析的早期，由

于缺乏对系统模块内部更深层次的认识，因此也难以进行合理的模块划分。此外，随着软件开发过程中分工也越来越细致，针对系统的不同层面，如界面、内部功能、数据库等的开发技术日益专业化，同一模块的不同层面往往会由不同的开发人员来实现。如出现了界面设计人员、用例设计人员、数据库设计人员等角色。正是因为这些方面的原因，决定了早期软件的备选架构不是对软件功能的简单分解，而更倾向于一种通用的分层策略。

分层架构的动机是将应用逻辑作为单独的构件从系统中分离出来，以便这些构件在其他系统中能得到复用。通过分层，可以将各个层次分配到各个不同的物理计算节点，或者分配给不同的进程；这样可以改善系统性能、更好地支持客户和服务器系统中的信息共享和协调。此外，分层策略还可以使得不同层的开发任务在开发者之间适当地分配，从而有效地利用开发人员的专长和开发技巧，并且能够提高并行开发能力。

经典的分层策略是三层架构，M-V-C 架构模式就是一种典型的三层架构。其中最底层模型层是面向系统内部的特定数据，它体现了系统的内在属性；而最高层视图层则是系统的外在表现形式，提供了系统与外界交互的功能；中间的控制器则是对系统核心业务流程的封装，它串接起系统的模型层和视图层，维护模型层的数据与视图层的表现形式的一致性，从而完成特定的业务流程。本书中的系统在早期架构选型时，其备选架构也采用了类似的 B-C-E 三层架构，如图 5-12 所示。

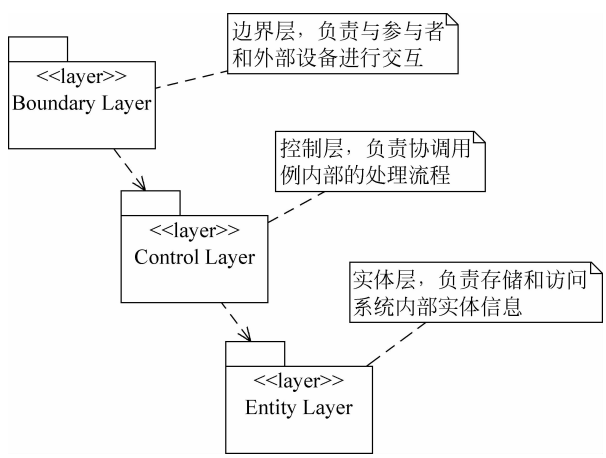

图 5-12　系统备选架构

在 B-C-E 三层架构的定义过程中，采用包图进行架构建模[①]。包是一种通用的模型分组机制，为了有效地表示分层的概念需要引入构造型<< layer >>，并在层和层之间建立依赖关系。通过 B-C-E 这三层划分系统中的三类处理逻辑，其中：

- 边界层(Boundary Layer)负责系统与参与者之间的交互。
- 控制层(Control Layer)处理系统的控制逻辑。
- 实体层(Entity Layer)管理系统使用的信息。

目前只是给出了这三个层次的基本定义，而各层的内部还没有任何元素，第 5.4 节的构造用例实现的过程将对架构中的各个层次进行填充。换句话说，当前分层的架构将为后续

① 有关利用包图进行架构设计的详细细节，可以进一步参阅本书第 8.2 节的内容。

的用例分析提供基本结构和指导规则。

5.3.2 分析机制

备选架构更多的是从系统的功能需求入手，对系统进行合理的分层。然而，在架构分析阶段，对于系统的非功能需求以及那些通用性问题也需要进行单独考虑。架构机制就是用来记录这类问题的一种策略。

1. 架构机制

架构机制是对通用问题的决策、方针和实践，它描述了针对一个经常发生的问题的一种通用解决方案。通过有效地应用架构机制，可以使项目组内部以相同的方式对待这些问题，并复用相同的解决方案。作为系统架构的一部分，架构机制常常集中和定位在系统的非功能需求上。

根据所关注的阶段和详细程度不同，共有三类架构机制，即分析机制、设计机制和实现机制。

分析机制以与实现无关的方式捕获解决方案的关键部分。它们可能表示结构模式或行为模式，也可能同时表示这两者。它们主要用于在分析过程中向设计人员提供复杂行为的简短表示，从而减少分析的复杂性并提高分析的一致性。通过这些机制，可以使分析工作集中于将功能性需求转换成软件概念，而不必细究那些需要用来支持功能但却不是功能核心的相对的复杂行为。分析机制通常源于对一个或多个架构或分析模式的实例化。

设计机制是对相应分析机制的更具体的定义。设计机制为概念上的分析机制添加具体的细节，但它并不具体到需要特定的技术。与分析机制相同，设计机制可以实例化一种或多种模式，在这种情况下为架构模式或设计模式。

实现机制则是详细说明了机制的准确实现；它使用特定的编程语言及其他实现技术（如特定厂商的中间件产品）对相应设计机制进行改进。一个实现机制可以实例化一个或多个代码模式或实现模式。

表 5-2 给出了三种典型的分析机制，从中也可以理解和区分三类架构机制。

表 5-2 架构机制示例

分析机制	设计机制	实现机制
持久性	关系型数据库	JDBC
	面向对象数据库	Object Store
分布	远程方法调用	JavaSE 6 RMI
安全性	RBAC 身份认证机制	C++算法实现

持久性分析机制是指拥有该机制的对象需要持久存在，即在创建它的程序退出后仍然逻辑存在的对象。存储和访问这些持久性对象是一件很复杂的工作，在分析阶段没有太多的精力去考虑这些细节。因此通过该机制来记录对象的这一特性，而具体的存储和访问方案将在后续的设计和实现机制中进行讨论。与之对应，在设计阶段，可以采用两类设计机制来解决该问题，即关系型数据库或面向对象数据库均可以达到持久化目标。相应地，在实现

阶段，需要采用 JDBC 或 Object Store[①] 实现机制来实现。

分布机制是指对象需要进行分布式访问；设计机制“远程方法调用”则可以满足该机制；具体实现时则采用 JavaSE 6.0 提供的 RMI。此外，安全性机制是指对对象的访问需要一定的安全性，为此设计阶段采用 RBAC（基于角色的访问控制）身份认证机制来保证对象的安全；在实现阶段则可以采用 C++来具体实现。

由此可见，分析机制通常与具体业务无关，它属于“计算机科学”的范畴，是对与软件实现相关的关键技术的描述。可以把它们看作是架构中的“占位符”，来表明相关问题的存在，从而尽量避免这些行为的细节分散架构工作的重点。通过应用这些分析机制，可以使分析工作集中在将功能需求转换成软件概念，而不必细究那些需要用来支持功能但却不是功能核心的相对复杂行为。这些行为细节将在设计阶段进行细化[②]。

2. 确定分析机制

在分析阶段，对于分析机制的处理主要包括三个方面的工作：首先需要抽取出系统中所有的分析机制，之后建立分析机制和所关联的类之间的关系，最后确定类所拥有的分析机制的相关特性。

可以从两个不同的角度来抽取分析机制：自顶向下和自底向上。自顶向下是根据类似项目的经验来估计出领域内会有哪些问题，以及如何解决这些问题。在分析过程中，常见的架构问题（即机制）如：持久化、安全性、事务处理、异常处理、分布式访问等。所有这些问题的共同之处在于它们是大多数系统的一般性要求，分别实现与基本应用功能的交互或支持基本应用功能。通过分析机制来描述这些问题，从而可以使分析阶段不必过多地考虑具体的实现平台或语言。通常，可以采用多种不同的方式设计和实现分析机制。即对应于每种分析机制可以有多种设计机制，而每种设计机制可以有多种实现机制。自底向上则是随分析过程的深入逐步确定，分析机制是最后生成的。在定义每一个分析类的过程中，发现相应的问题，这些问题最初比较模糊，将它们定义为与具体实现策略无关的分析机制，从而在后续迭代中进行进一步细化。

考查“旅店预订系统”，由于该系统只是一个简单的单机版系统，其中涉及的通用问题较少，只存在一个持久性分析机制。而“旅游申请系统”中的分析机制较多，除了持久性之外，还涉及分布、安全性、遗留接口等，表 5-3 列出了该系统中可能存在的分析机制。

表 5-3　旅游申请系统分析机制

分析机制	含　义	使用该机制的元素
持久性	使一个元素持久化的方法	路线、申请、申请人等信息
分布	将一个元素跨越系统现有代码而远程访问的方法	办理申请手续的处理过程以及其他的远程访问处理请求
安全性	控制一个元素的访问权限的方法	申请人的支付信息
遗留接口	使用现有接口访问一个遗留系统的方法	访问财务系统的接口

① 可以采用某种对象访问技术，此处并未具体给出。如可以是 XQuery、XPath 等 XML 访问技术。

② 参阅第 8.4 节，相关内容详细说明了定义设计机制的过程。

抽取了这些分析机制之后，下一步就需要明确哪些类与它们相关，可以建立一张分析类与分析机制的对照表，具体的实例将在第 5.5.4 节给出。

最后，分析机制不仅仅只是一个简单的名字，不同的分析机制需要考虑不同的细节特征，而且同一分析机制对于不同的类而言也存在不同的特征值。这些信息会影响到后续的设计，因此在分析阶段还需要描述分析机制的特征。表 5-4 给出了部分分析机制所拥有的特征。

表 5-4 分析机制的特征

分析机制	特　征	含　义
持久性	粒度	单个持久性对象的大小
	容量	所需要存储的持久性对象的个数
	访问频率	对对象进行 CRUD 等操作的频率
	访问机制	如何唯一地标识和检索对象
	存储时间	对象需要保存的时间
分布	分布机制	分布式访问的方式或协议的约束
	通信方式	同步或异步访问方式
	通信协议	消息的大小、流量控制、缓存等相关通信约束
安全性	安全规则	使用何种安全访问规则
	授权策略	CRUD 等其他操作的权限
	数据粒度	数据访问权限的粒度，如包级、类级、属性级
	用户粒度	用户权限的定义级别，如单一用户、角色/分组用户
遗留接口	响应时间	要求的访问响应时间
	持续时间	每次访问遗留系统所持续的时间
	访问机制	访问方式或协议
	访问频率	访问遗留系统的频率

5.3.3 关键抽象

业务建模阶段和需求阶段通常会揭示系统必须能够处理的核心概念，而这些核心概念也往往会作为系统设计和实现阶段的关键类，因此这些概念需要在分析阶段进一步明确，这就是关键抽象。

关键抽象是一个通常在需求上被揭示的概念，系统必须能够对其处理。它来源于业务，体现了业务的核心价值，即业务需要处理哪些信息；这些信息所构成的实体即可作为初步的实体分析类。关键抽象的具体来源有需求描述、词汇表，特别是业务对象模型。在架构分析阶段定义系统的关键抽象，可以便于后续用例分析阶段的展开，以避免不同的用例分析团队在分析各自的用例时可能产生的重复工作。

通过一个或多个类图来展示关键抽象，并为其编写简要说明。这些关键抽象一般也会作为用例分析阶段的实体类而存在。此外，类图中除了展示这些概念外，还可以进一步描述它们之间的关系。

对于“旅店预订系统”而言，其核心业务价值在于处理旅客的房间预订信息，因此有关旅客、房间、预订信息等概念即作为系统的关键抽象而存在。

而图 5-13 给出了“旅游申请系统”的关键抽象，显然这些概念是该系统所需要处理的最

基本的信息实体。当然，此处并没有给出它们之间的关系，有关类之间关系的定义将在第5.5.3节进行详细介绍。表5-5给出了这些关键抽象的简单描述。

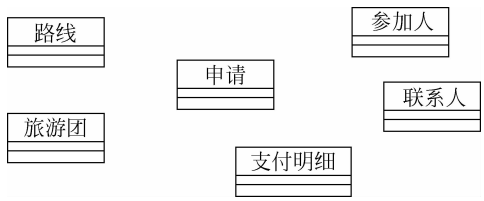

图5-13 “旅游申请系统”关键抽象类图

表5-5 “旅游申请系统”关键抽象说明

关键抽象	含　义	相关联的关键抽象
路线	包含具体旅游安排和介绍的旅游路线	
旅游团	各个旅行路线的具体组团；各个线路可组织多次不同的旅游团	路线
申请	参加人提交的旅游团申请表	旅游团
支付明细	与申请相关的订金、旅费、退费等费用支付信息	申请
参加人	某个申请的参加人员信息，分责任人和普通人员两种角色；具体人员还需要区分大人和小孩	申请
联系人	参加人指定的联系人信息	参加人

5.4 构造用例实现

通过架构分析的过程，获得了系统的备选架构、分析机制和关键抽象，下一阶段将以这些素材为基本的输入，来构造每一个待分析的用例实现内部细节。与需求阶段通过文档来表示用例不同，分析设计阶段将通过构造更多的UML模型来表示用例实现，通过这些模型来描述系统内部类之间的静态关系以及对象间的动态交互。本节所阐述的构造用例实现的过程将是针对每一个用例实现重复进行的。

5.4.1 完善用例文档

正如第5.2.2节中所阐述的，用例实现和用例是两个不同的概念。它们关注的重点不同：用例面向用户描述功能，而用例实现则面向分析设计人员描述软件的内部结构。它们的范围也可能不同：用例描述了某个用户目标，而用例实现可能只包含了复杂用例某次迭代的部分行为。因此，在构造用例实现之前，首先需要对原始的用例文档进行进一步的分析和完善，以便获取理解系统的必要内部行为所需的其他信息，而这些信息可能是在为系统客户编写的用例说明中被遗漏的，但却是系统内部所需的处理信息。

一般情况下，用户对有关系统内部情况的处理信息并不感兴趣，因此在用例文档中很可能遗漏了这些信息。在这种情况下，用例文档更像是一份“黑盒”说明文档，只描述了用户的输入和期望的系统输出，而省略了相关系统响应参与者动作的内部详细信息，或者相关细节过于简单。然而，在构造用例实现阶段，需要有一份从内部角度观察系统响应的“白盒”文档，这些文档中待完善的细节也就是第4.4.4节所描述的用例交互四步曲中的第(2)步和

第(3)步。

在“旅游申请系统”的办理申请手续业务中,当服务员录入了申请的信息后,系统能够计算并显示本次申请所需的费用的总额和订金金额等信息。对于系统用户(即服务员)来说,这样描述就可以了,因此在第 4.4.7 节的用例文档中就描述为“6.系统显示旅行费用的总额和申请订金金额;”。但是,这并没有给出任何具体的系统实现方面的描述,即系统到底依据什么来计算这些费用信息,显然分析设计人员必须清楚而且准确地定义这些信息。为此,用例分析阶段可能需要这样来描述“系统根据旅游团价格和参加人情况计算费用总额以及订金金额,相关计算规则如下所述……”。这种程度的详细信息为分析设计人员提供了一个清晰的概念,即需要什么样的信息(旅游团价格和参加人情况)以及如何来计算(相关的业务规则)。基于这些信息,可以确定两个可能存在的对象(即包含价格的旅游团和区分大人和小孩的参加人)以及它们的职责。

当然,分析阶段的目的只是确保系统的内部行为清晰明了,从而能够明确系统必须做什么。但没有必要定义系统内负责执行行为的元素(对象)以及具体的算法规则(这是设计的事),只要清楚地定义需要做什么即可。

在实际完善用例文档的过程中,提供这些详细信息的人员包括能够帮助确定系统需要执行什么操作的领域专家。在考虑系统的某个特定行为时,可以提出一个针对性很强的问题:“执行此行为对系统意味着什么?”如果没有明确规定系统需要进行什么操作来执行此行为,就无法回答上述问题,因此可能需要发现更多的信息,从而弥补需求阶段的不足。

当然,这个工作并不意味着要对用例文档中的每个细节都重新细化,这样做的话工作量将非常大。一般只对一些重要的、复杂的系统处理进行进一步分析和描述。此外,这个活动本质上和后面通过顺序图来分析交互是类似的,只不过通过文字更具体一些。因此,在实际分析过程中,当在后面分析交互时发现通过 UML 图形无法准确地描述那些细节时,就可以通过文字的方式适当地进行补充描述。

5.4.2 识别分析类

在对象系统中,系统的所有功能都是通过相应的类来实现的。因此首先需要从用例文档中找出这些可用的类,之后再将其所描述的系统行为分配到这些类中。面向对象的分析是对这个过程的第一次尝试,这是一个从“无”到“有”的跨越,也是整个分析过程中最难的任务之一。而分析阶段所定义的类被称为分析类。

分析类代表了系统中具备职责和行为的事物的早期概念模型,这些概念模型最终会演化为设计模型中的类或子系统。分析类关注系统的核心功能需求,用来建模问题域对象。此外,分析类主要用来表现“系统应该提供哪些对象来满足需求”,而不关注具体的软硬件实现的细节。

架构分析中所定义的 B-C-E 三层备选架构为识别分析类提供了很好的思路。按照该备选架构,系统中的类相应地对应三个层次,即边界类、控制类和实体类。识别分析类的过程就是从用例文档中来定义这三类分析类的过程。

1. 边界类

边界类处于系统的最上层,它从那些系统和外界进行交互的对象中归纳和抽象出来,代表了系统与外部参与者交互的边界。存在两类边界类:用户界面类和系统/设备接口类。

用户界面类代表了系统与外部用户进行交互的类，在分析期间并不关注界面的细节（如窗体或页面的个数、布局）以及实现方案，而是侧重于为用户提供哪些操作（即用户通过界面能做什么，如录入特定信息等）。

系统/设备接口类代表了系统与外部系统或设备之间交互的接口，在分析期间通过接口类主要关注交互的协议，而不关注协议实现的细节。

边界类是一种用于对系统外部环境与其内部运作之间的交互进行建模的类，通过这些边界类封装了系统与外界的交互。由于明确了系统的边界，边界类能帮助人们更容易地理解系统，并且为后续阶段确定相关服务提供了一个很好的依据。如果确定了一个针对打印机的设备接口类，那么很明显，就必须对打印输出格式进行建模。

在用例分析阶段，对边界类识别的基本原则是：为每一对参与者或用例确定一个边界类。虽然最终的交互界面可能会有很多个，但这并不是分析的内容（在设计阶段会进一步分解）。为区分于其他的分析类，采用构造型<< boudary >>表示边界类。此外，Rational Rose可以将该构造型表示为小图标的方式（即在类名字的右上角放置一个图标），或者采用更形象的类图标来代表边界类，图 5-14 给出了三种边界类的表示形式，本书主要采用类图标的方式来表示（即最右边的一种方式）。图 5-15 给出了基于图 4-14 中的用例图来识别边界类的示例。

图 5-14　边界类的三种表示方法

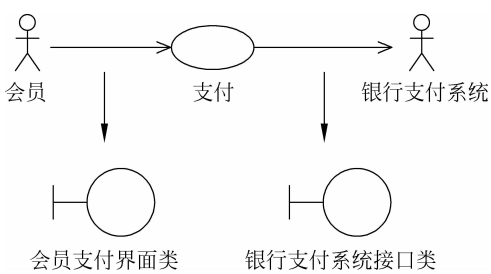

图 5-15　识别边界类

从图 5-15 中可以看出，针对参与者和用例之间的每一个关联关系就可以定义一个边界类。当然，在实际应用中也可灵活处理，例如对于包含多个复杂分支流程或子流程的用例，可以针对不同的处理业务定义不同的边界类。此外，对于边界类的命名一般采用“××界面类”来表示用户界面，采用“××接口类”来表示系统/设备接口类[①]。

2. 控制类

控制类处于三层架构的中间层，它封装控制系统上层的边界类和下层的实体类之间的交互行为，是整个用例行为的协调器。控制类能够有效地将边界对象和实体对象分开，让系统更能够适应其边界对象内发生的变更；并且还可以将用例所特有的行为与实体对象分开，使实体对象在用例和系统中具有更高的复用性。控制类所提供的行为具有以下特点。

- 独立于外部环境，不依赖于环境的变更。
- 定义用例中的控制逻辑和事务管理。

① 本书中分析阶段类的名字都采用中文标注，以便于用户理解。在实际项目开发时，可根据项目组相关的命名规范来进行命名。此外，在实际项目中，类的名字一般也不会有“类”字这个后缀。

- 在实体类的内部结构或行为发生变更的情况下，不会有大的变更。
- 使用或修改若干实体类的内容，因此需要协调这些实体类的行为。
- 并不是每次用例激活时都以同样的方式运行(因为事件流本身具有不同的状态)。

在用例分析阶段，对控制类识别的基本原则是：为每个用例确定一个控制类。为区分于其他的分析类，采用构造型<< control >>表示控制类。同样，Rational Rose 也提供了相应的图标来表示，图 5-16 给出了三种控制类的表示形式，本书主要采用类图标的方式来表示(即最右边的一种方式)。图 5-17 给出了基于图 4-15 中的用例图来识别控制类的示例。

图 5-16 控制类的三种表示方法

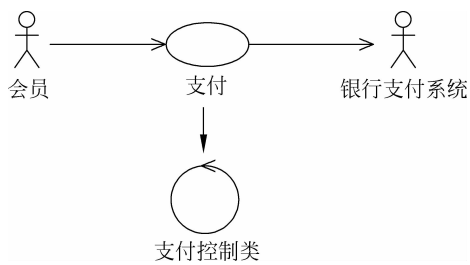

图 5-17 识别控制类

从图 5-16 中可以看出，针对一个用例可以定义一个控制类。当然，在实际应用中也可灵活处理，例如对于那些复杂的用例(如包含多个复杂分支流程或子流程的用例)，可以定义多个控制类来处理不同的业务逻辑。此外，对于控制类的命名一般可以采用用例的名字加上“××控制类”的后缀的方式[①]。

3. 实体类

实体类代表了系统的核心概念，来自于对业务中的实体对象的归纳和抽象，用于记录系统所需要维护的数据和对这些数据的处理行为。实体类提供了另一个理解系统的观点，即从系统的逻辑数据结构来描述系统应该为用户提供哪些功能[②]。

实体类是用来表示数据信息的名词，因此识别实体类的基本思路是分析用例事件流中的名词、名词短语找出所需的实体信息。不过与边界类和控制类不同，实体类通常并不是某个用例实现所特有的(即可能跨越多个用例)，甚至于可能跨越多个系统。因此，除了用例事件流，业务模型、词汇表等业务和需求的载体中均可能获得实体类；当然，架构分析中的关键抽象是识别实体类的最重要来源。为区分于其他的分析类，采用构造型<< entity >>表示实体类。同样，Rational Rose 也提供了相应的图标来表示，图 5-18 给出了三种实体类的表示形式，本书主要采用类图标的方式来表示(即最右边的一种方式)。图 5-19 给出了基于图 4-15 中的用例图来识别实体类的示例。

---

① 控制类也可以采用其他后缀，如“××管理类”；或不采用后缀，如直接命名为“支付”类；从这个动词性质的类名(其他类基本上是名词，控制类一般是动名词)就可以看出该类是对支付业务处理的控制类。

② 由于边界类和控制类比较容易确定，因此，对实体类的识别才是整个分析阶段的重点和难点，遗漏了实体类则意味着丢失了系统的数据和处理行为，从而不能满足需求。事实上，存在一些面向对象的方法在分析阶段就只关注实体类，而不考虑边界类和控制类。

图 5-18　实体类的三种表示方法

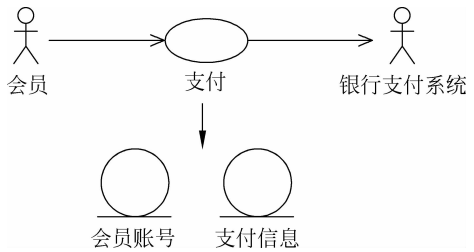

图 5-19　识别实体类

从图 5-19 中可以看出，从一个用例可以识别出若干个实体类。具体的数目与该用例所处理的数据有关，由于用例之间的差别很大，因此能识别的实体类数目差别很大；此外，从不同的用例中可能获得相同的实体类，还需要考虑这些跨用例的实体类的一致性等问题。在“支付”用例中，会员为了完成支付操作，需要通过“会员支付界面类”向系统提供其账号信息和所需支付的金额，系统控制类依据这些信息生成一条支付记录，并将该支付记录通过“银行支付系统接口类”提交给银行支付系统进行支付。因此该用例中存在“会员账号”和“支付信息”两个实体类。

由于并不存在一些具体的方法或原则来识别实体类，因此为了能够有效地识别，还需要采取其他的辅助手段。一种最常用的方法就是名词筛选法，基本思路如下所示。

(1) 将用例事件流作为输入，找出事件流中的名词或名词性短语(可采用下划线标注出来)，这些名词形成了实体类的初始候选列表。

(2) 合并那些含义相同的名词。因为事件流描述的可能不准确，所以相同的概念可能采用了不同的名词，需要将这些不同的名词进行统一定义，并重新确定合适的名字。

(3) 删除那些系统不需要处理的名词。有些名词可能只是用例中的描述信息，并不需要处理，这些名词也不会作为实体类存在。

(4) 删除作为参与者的名词。因为参与者是在系统范围外的，所以在当前用例中不作为实体类被定义。不过，由于大部分系统都会维护那些用户类型的参与者，因此这些用户信息将会在其他的用例中(如登录、管理用户等用例)被定义为实体类。

(5) 删除与实现相关的名词。分析阶段不考虑系统实现方案，因此与实现相关的内容也不会作为实体类存在。

(6) 删除那些作为其他实体类属性的名词。有些名词可能只简单地描述一个值，这些单一的值一般也不作为类存在，而会作为其他类的属性。

(7) 对剩余的名词，综合考虑它在当前用例以及整个系统中的含义、作用以及职责，并基于此确定合适的名字，从而作为初始的实体类存在。

名词筛选法是一种最原始也是最有效的识别实体类的方法，不过相对来说效率比较低。此外，由于该方法的输入是用自然语言描述的用例文档，自然语言的不精确以及一些名词词性的活用(如名词动词化、动词名词化等)也会给实体类的识别带来麻烦。因此，这种方法一般用于分析初期，由于缺乏对系统理解，只能通过这种方法来获取的实体信息。而有经验的分析人员更多地依赖于类似项目的经验和对业务及系统的理解，来获取系统的关键抽象，这些关键抽象构成了系统中最重要的那些实体类。此外，并不能指望在此阶段就能够发现所有的实体类，在分析交互和后续的迭代中也可能发现一些新的实体类。

下面以“旅游申请系统”中的“办理申请手续”用例为例，来简单地说明如何使用名词筛选法识别实体类，这里只考虑了该用例的基本事件流。

(1) 用下划线标记事件流中的名词。

① 该用例起始于旅客需要办理申请手续。

② 前台服务员录入要申请的旅游团旅行路线代码和出发日期。

③ 系统查询要申请的旅游团信息(A-1)。

④ 系统显示查询到的旅游团和相关路线信息(D-1)(A-2、A-3)。

⑤ 前台服务员录入本次申请信息(D-2)。

⑥ 系统显示旅行费用的总额和申请订金金额。

⑦ 前台服务员提交该申请信息。

⑧ 系统保存该申请信息(A-4)，用例结束。

(2) 合并含义相同的名词：旅游团和旅游团信息合并为旅游团。

(3) 删除系统不需要处理的名词：用例、申请手续、系统。

(4) 删除参与者：前台服务员。

(5) 删除实现相关的名词：无。

(6) 删除属性：路线代码作为路线的属性被删除，出发日期作为旅游团的属性被删除。旅行费用总额、申请订金金额等均为一个简单值，可作为申请信息的属性值存在。但费用和订金支付业务存在一定的处理流程和状态，因此可以考虑单独封装在一个“支付明细”实体类中(在该用例中代表订金支付明细，在“完成支付”用例中则代表总费用支付明细，而在“取消支付”用例中则可能代表退费的情况。可以把这些情况放在一个类中，也可以针对这些情况设计不同的子类，有关类细节的设计参见第 5.5 节)。

(7) 最终剩余的名词：旅客、旅游团、相关路线信息、申请信息、支付。结合申请系统的自身特点，并和关键抽象的定义保持一致，旅客可被命名为参加人；而相关路线信息可被简称为路线，申请信息可被简称为申请。最终确定的初始实体类有：申请人、旅游团、路线、申请和支付。

4. 总结：分析类

识别分析类的过程就是对于每个用例实现，根据系统备选架构的约定，从中抽取出相应的边界类、控制类和实体类来填充系统架构。通过所识别的这些类，来达到该用例实现所实现的用例的业务价值。图 5-20 给出了会员支付业务的用例模型和由该用例所获得到的初始分析类。

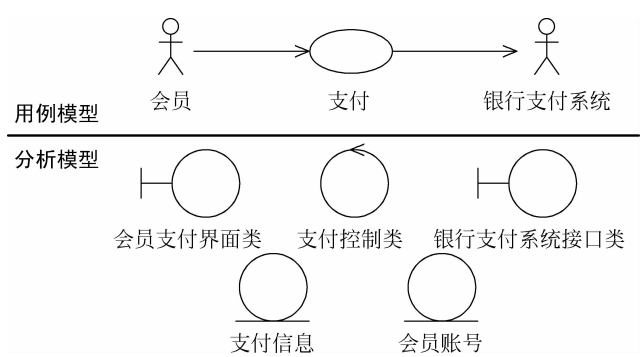

图 5-20 会员支付业务的用例模型和分析模型

从图 5-20 中可以看出，作为用例模型中的用例对外封装了相关的业务细节，而在分析模型中则需要将这些业务细节分解到相应的分析类中。这就是一个将用例这个“黑盒”展开进行“白盒”分析的基本思想。基本思路是这样的：首先需要定义边界类来响应系统与外界的交互；其次，定义控制类来处理外界请求并通过操作相应的实体类来返回结果；而系统内部的所有信息都需要定义实体类来存储。图 5-21 展示了识别三种分析类的过程，从中可以看出，一个系统用例在分析阶段最终被分解成若干个由分析类组成的结构。

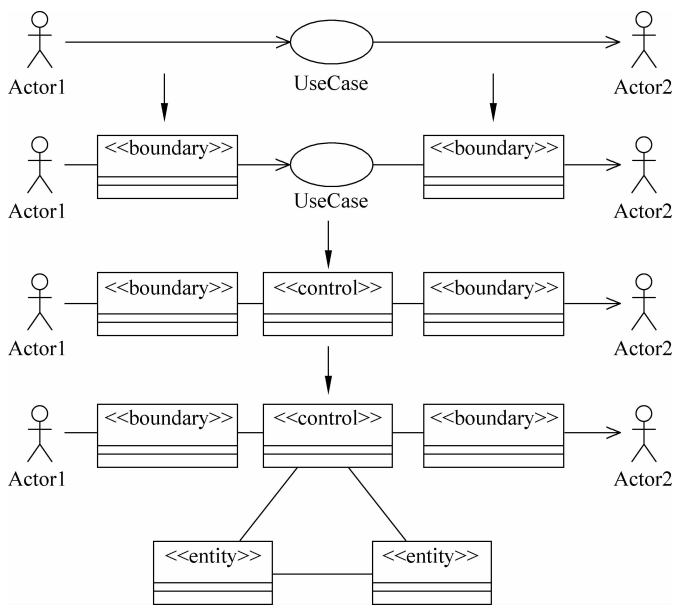

图 5-21　从用例模型到分析模型

5. 识别分析类实践

下面首先以“旅店预订系统”的首次迭代为例，介绍识别分析类的基本方法和实践技能。

根据第 5.2 节所定义的该系统首次迭代，本次用例分析主要处理两个用例实现，即“预订房间”和“取消预订”。结合图 5-5 可以很容易地识别出两个边界类（预订界面类 ReservationUI 和取消界面类 CancelUI）和两个控制类（预订控制类 ReservationController 和取消控制类 CancelController）。而至于实体类则需要利用名词筛选法从第 4.4.7 节的用例文档中获得。本系统中存在 4 个实体类（旅客 Customer、房间 Room、预订信息 Reservation 和支付信息 Payment）。由此可见，即使是类似于旅店预订这样的小规模系统，首次分析迭代就涉及将近十个左右的分析类，而到后续阶段类的数量会更多。因此，不可能把这些类都放在同一个层次来考虑，而是必须采取一种有效的方式来组织这些类，这就是软件架构所要解决的问题。对于初始的分析类而言，备选架构提供了一种很好的组织方式，可以把这三种分析类分别存放到其三个层次中。图 5-22 显示了在 Rose 中如何组织这些初始的分析类，从图 5-22(a)中能够看出，可以将每一个架构层次看作一个模型包，其内部包含相应的分析类；而图 5-22(b)则显示了每个架构包中的主视图，将相应的类展示在类图中，这些类之间的关系将会在后续阶段进一步定义（参见第 5.5 节）。

旅游申请系统首次迭代规模相对较大，其分析类更多，备选架构的作用就更加明显。

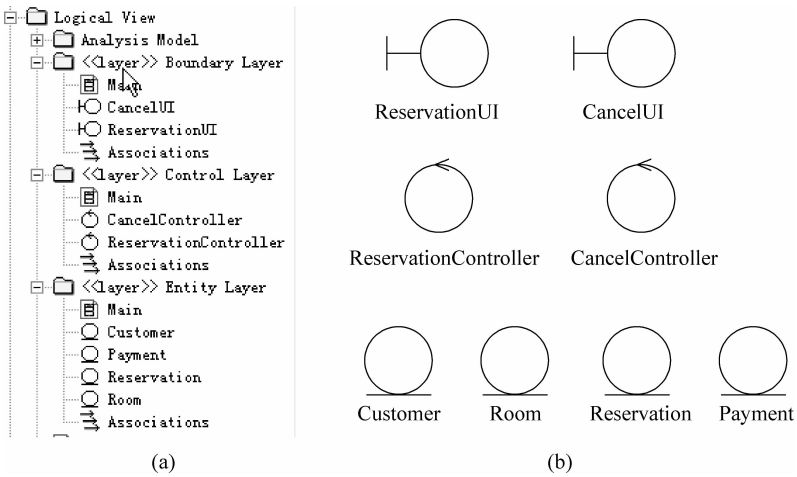

图 5-22 旅店预订系统初始分析类

图 5-23～图 5-25 分别给出了该系统的边界类、控制类和实体类的主视图，从图中可以看出该系统首次迭代所识别的全部分析类，这些分析类完全是按照前面所介绍的方法识别出来的。

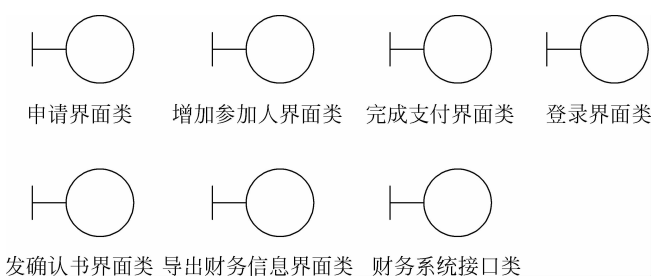

图 5-23 旅游申请系统初始边界类

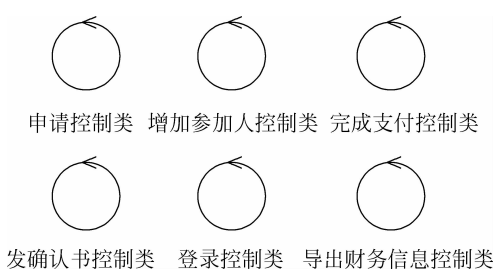

图 5-24 旅游申请系统初识控制类

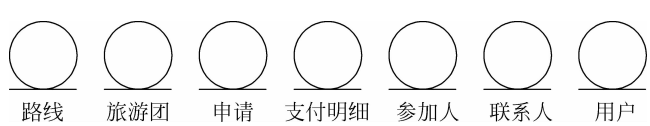

图 5-25 旅游申请系统初始实体类

本系统首次迭代共定义了 7 个边界类，其中包括 6 个界面类和 1 个系统接口类。由于首次迭代针对“管理参加人”用例只实现“增加参加人”子流程，因此其界面类也被定义为“增加参加人界面类”。而“登录界面类”则同时处理前台服务员和收款员工两类用户的登录。此外，虽然时间参与者可能并不需要操作界面来启动系统（一般情况下是系统后台自动运行的业务），但为了后续分析的一致性，目前也定义了一个“导出财务信息界面类”。

控制类的定义比较简单，对应于首次迭代的 6 个用例，定义了 6 个控制类来封装相应用例的业务流程和逻辑规则。

本系统目前初步定义了 7 个实体类，这些实体类与前面的关键抽象基本一致。多出来的“用户”类是从登录用例中提取出来的，该类记录了系统的用户（包括前台服务员、收款员工、路线管理员等不同角色）信息，如用户名、密码等。关键抽象的提取更多的是凭借着对业务的理解和相关项目的经验；而此阶段对实体类的提取还可以按照前面所提到的名词筛选法来进一步明确，以获得更多的实体类。

5.4.3 分析交互

目前所识别的分析类都是静态的描述，而为了确认所识别的分析类是否达成用例实现的目标，必须分析由这些类所产生的对象的动态行为，这就是分析交互的过程。

交互是一种对象间的行为，这种行为由一系列对象为实现某一目标而相互传递的一组消息构成。消息是对传送消息的对象之间所进行的通信的规约，在面向对象语言实现中一般表示为对象操作的调用。

在面向对象系统中，系统所提供的所有行为都是通过对象间的交互来完成的。在需求阶段，系统行为通过事件流的方式以自然语言进行描述；而在分析阶段需要将这些文字描述转换成 UML 模型。分析交互的过程就是利用 UML 相关模型来描述对象间的交互，以表示用例实现是如何达到用例目标的。

在 UML 模型中，通过交互图来表示对象间的交互，它由一组对象和它们之间的消息传递组成。有两种典型的交互图，即顺序图和通信图。顺序图是强调消息时间顺序的交互图，而通信图则是强调接收和发送消息的对象间关系的交互图。本章主要采用顺序图来表示交互，第 6 章将会介绍如何通过通信图来表示交互。

1. 顺序图

顺序图强调消息的时间顺序，图 5-26 是一个典型的顺序图，其中包含了顺序图的基本要素，即对象、对象生命线、控制焦点等概念。

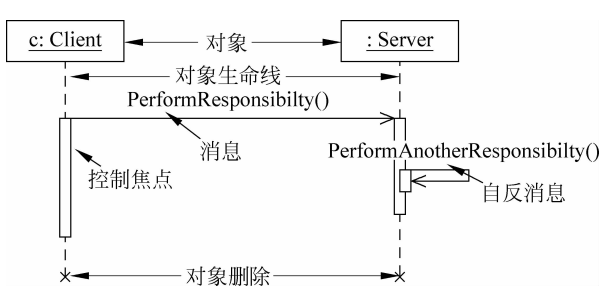

图 5-26 顺序图

1）对象和生命线

图 5-26 中的矩形框代表对象，在顺序图中的对象依次排列在图的上方，为了便于对图形的理解，通常将发起交互的对象（即图中 Client 类的 c 对象）放在左边，而将接收消息的对象放在右边。矩形框里面可以指定对象的名字和类的名字，冒号前面为对象的名字，冒号后面为类的名字。对于一个交互来说，对象通常只是表示某个类的一般实例，并不特指某个特定的对象，此时无需指定特定的对象名，只需要指定类的名字即可（图中 Server 类的对象没有被命名，可称为匿名对象）。当然，有时交互只针对某个特定的个体对象的行为，此时则需要同时指定对象名和类名（图中 Client 类的对象命名为 c）。严格来说，由于对象必须通过相应的类来构造，必须指定顺序图中对象的类名（在有些并不严谨的模型中可能存在没有类名的对象）。

顺序图中的对象有生命线，它是一条垂直的虚线，表示一个对象在一段时间内存在。在顺序图中出现的对象大部分是存在于整个交互过程中，所以这些对象全部排列在图的顶部，其生命线从图的顶部延续到图的底部。当然，也可以在交互的过程中创建对象，此时这些新创建对象的生命线从接收到创建（create）消息开始（可以将创建消息发送到表示对象的矩形框上）。此外，也可以在交互的过程中删除对象，它的生命线在接收到删除（destroy）消息时结束，在顺序图中采用“×”表示对象生命的结束。

2）消息

顺序图中最重要的是消息，消息表示为从一条生命线到另一条生命线的箭头，箭头指向消息的接收者，表示对其操作的调用。消息的调用顺序是沿着对象的生命线从上往下的，通过这一系列的消息调用来完成交互。

有不同种类的消息，典型的有同步消息、异步消息、返回消息、创建消息和删除消息。同步消息表明调用者需要等到操作调用结束后才能返回执行下一条消息，采用带实心三角箭头的实线表示。异步消息则表明调用者发出调用消息后，并不等待操作的执行结果而直接返回执行下一条消息，采用带有枝状箭头的虚线表示（见表 2-6）。图 5-26 的顺序图是采用 Rose 2003 绘制的，图中的枝状箭头消息只是表示一条简单消息（不是异步消息），并没有明确指定消息的类型（一般默认为同步消息）[①]。返回消息本质上并不是一个消息调用，它表明对同步消息调用的返回结果，用带有枝状箭头的虚线表示，其消息内容为消息的返回值，一般为一个对象或简单变量。由于每个同步调用后都隐含一个返回，一般情况下可以省略返回消息。但如果需要明确表示返回结构的内容，则可以使用返回消息。创建消息是指创建一个新对象的调用，可以利用构造型<< create >>来区分。删除消息表示删除接收消息的对象，可以利用构造型<< destroy >>来区分。图 5-27 展示了这几种不同的消息类型。

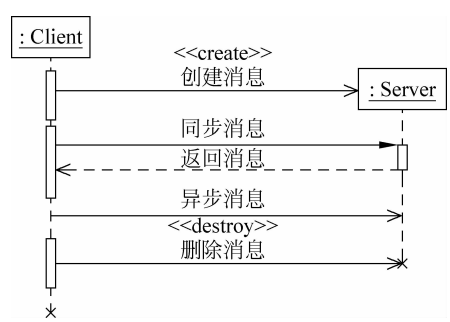

图 5-27 顺序图中的消息

① 在消息类型的可视化方面，Rose 2003 中某些消息的图示与 UML 2 的标准并不一致，其异步消息为半枝状箭头（即只有上面的斜线，没有下面的斜线箭头）。创建消息也没有单独的表示符号。

此外，消息除了可以由一个对象发给另外一个对象，还可以发给该对象本身，这类消息称为自反消息，表明一个对象对自身操作的调用，图 5-26 中的自反消息表明 Server 类的对象会调用自身的 PerformAnotherResponsibilty()。

顺序图中对象的生命线反映了时间的推移，因此不需要编号就可以明确消息的执行序列。但是，在很多情况下，通过对消息进行适当编号可以更直观、更清楚地理解消息的执行序列和层次关系。有两种消息的编号方式。一种是顺序编号，即按照消息执行序列采用阿拉伯数字依次编号。另一种方式是层次编号，同时按照消息的执行序列和层次关系，采用 1、1.1、1.2，2、2.1、2.2…这样的方式分层编号，其中 1.1 和 1.2 表示嵌套在消息 1 中的第一个和第二个消息；这种嵌套可以任意深度，即 1.1 的嵌套是 1.1.1、1.1.2 等。在 Rose 2003 中，可以进行相应的设置：选择 Tools|Option 命令，选择 Diagram 选项卡，选择 Sequence numbering 选项则表示对顺序图进行编号，选择 Hierarchical Message 选项则表示采用层次编号，如图 5-28 所示。由于层次编号反映的信息最全(既体现消息的执行序列，又反映消息的层次关系)，因此本书后面的顺序图都采用层次编号的方式。

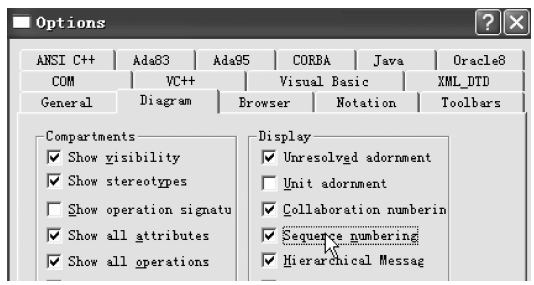

图 5-28　顺序图中消息的编号方式

图 5-29 给出了采用这两种消息编号的示例，图 5-29(a)采用的是顺序编号，图 5-29(b)则采用层级编号。注意理解这两种编号方式的含义和区别。

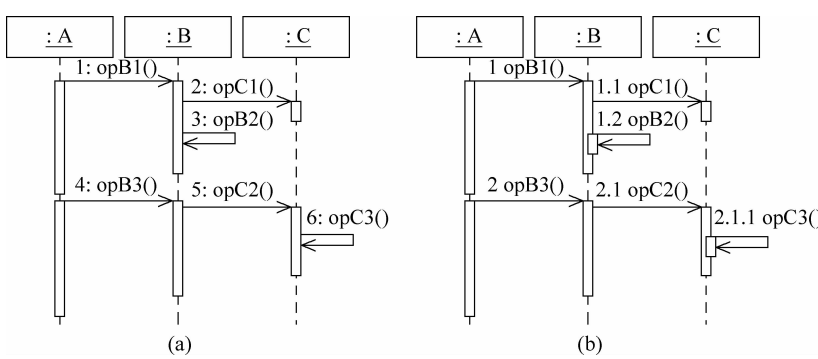

图 5-29　消息的顺序编号和层级编号

3) 控制焦点

控制焦点是顺序图的另一个特色，在顺序图中表示为附加在对象生命线上的瘦高矩形。它表示对象执行一个动作所经历的时间段，既可以是直接执行，也可以是通过下一级的消息执行。矩形的顶部表示执行动作的开始，底部表示动作的结束(可以通过一个返回消息来标

记)。还可以通过将一个控制焦点放在父控制焦点的右边来表示控制焦点的嵌套,这种嵌套可以由循环、自反消息或从另一个对象的回调所引起。

控制焦点还可以反映消息之间的层次关系,在图 5-29(a)中,类 B 对象的第一个控制焦点的输入消息 1 和输出消息 2、输出消息 3,则表明消息 2 和消息 3 是嵌套在消息 1 中的(即为了实现消息 1 而产生的后续交互),而这种嵌套与层次编号中体现的关系是一致的(即 2、3 表示为 1.1 和 1.2),如图 5-29(b)所示。

2. 利用顺序图描述交互

掌握了顺序图的基本用法之后,下面将详细介绍如何利用顺序图发现和描述对象之间的交互以完成某个用例实现。这个过程针对每一个用例场景按照如下三个步骤进行。

(1) 放置对象:从已识别的参与用例的分析类中构造相应的对象放置到顺序图中。

(2) 描述交互:从参与者开始,按照用例事件流(或场景)的叙述,将系统行为转化为对象间的消息。

(3) 验证行为:从后往前,验证对象的行为序列,确保每一个对象能够实现该行为序列。

下面将以“旅店预订系统”中“预订房间-用例实现”的基本场景为例,详细讲解利用顺序图分析交互的过程。

1) 放置对象

放置对象的基本顺序是 ABCE(Actor、Boundary、Control、Entity)。首先在顺序图中放置该用例的外部参与者,然后放入边界对象,接着放入控制对象,最后放入实体对象。由于每个用例都是由一个外部参与者触发的,所以最先放置参与者。其次,参与者需要通过边界对象与系统进行交互,因此下一个就是该边界对象。而边界对象接收到用户行为后,交由控制对象进行后续处理;控制对象将按照用例所约定的业务规则和流程来操作相应的实体对象;最后,根据用例的复杂程度,可能会有若干个实体对象。

图 5-30 显示了按照这种基本规则放置的“预订房间-用例实现”顺序图中的对象。

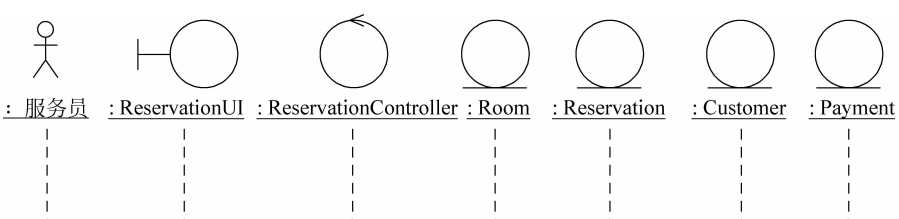

图 5-30 放置对象

在该用例实现的顺序图中,存在 7 个对象,由于使用构造型的原因显示为不同的图标。同时考虑到这些对象的一般性,并没有指定具体对象名,而采用了匿名对象。

首先,该用例是由“服务员”启动的,因此被放置的第一个对象为参与者“服务员”的实例。之后放置从该用例中提取出来的相应的边界类(ReservationUI)和控制类(ReservationController)。最后是实体类:在预订房间需要用到房间信息(Room),以及生成的预订信息(Reservation)、旅客信息(Customer)和支付信息(Payment)。当然,在最开始放置对象时,并不是必须一次性放置全部实体类对象,可以在下一步描述交互的过程中按照

需要逐步加入。

2）描述交互

一旦在顺序图上放置好了对象，下一步就可以描述对象间的交互过程，这是面向对象分析最关键的一步，通过这个过程将用例文档中以文字形式描述的事件流转换为利用顺序图描述的对象之间的交互序列。这个交互过程是通过向相应的对象发送消息来完成的；由于分析阶段并不考虑实现的细节，因此现阶段可以考虑以更直观的、自然语言的方式对对象消息进行命名，以便于理解。此外，考虑到一个对象向另一个对象发出消息意味着对接收该消息对象的操作调用，因此每一个接收该消息的对象都需要提供相应的操作来响应该消息调用。当然这些操作只是最初的定义，并不代表最终的实现，故称为分析操作。为了与最终实现时的操作相区分，可在该操作前面加上“//”，以表明当前只是初步进行类的职责分配，具体的操作细节尚未制定完全。这个描述交互的过程也就是类的职责分配过程，也是分析图 4-20 所描述的用例交互的第 2 步和第 3 步的过程。

由于接收消息的对象是通过其分析操作来响应消息调用的，因此在职责分配过程中必须考虑到对象能够提供该分析操作，或者说有能力实现该职责，这是分配职责时最基本的原则。而在具体职责分配过程中，还可以从以下两个方面来考虑职责分配问题。

（1）以分析类[①]的构造型作为职责分配的基本依据，其中：

- 边界类承担与参与者进行通信的职责。
- 控制类承担协调用例参与者与内部数据操作之间交互的职责。
- 实体类承担对被封装的内部数据进行操作的职责。

（2）专家模式[②]将职责分配给具有当前职责所需要的数据的类，其中：

- 如果一个类有这个数据，就将职责分配给这个类。
- 如果多个类有这个数据[③]：
  - 方案 1 是将职责分配给其中的一个类，并对其他类增加一个关系。
  - 方案 2 是将职责放在控制类中，并对需要该职责的类增加关系。
  - 方案 3 是创建一个新类，将职责分配给该类，并对需要该职责的类增加关系。

图 5-31 显示了按照这些职责分配规则对“预订房间-用例实现”的基本事件流进行分析交互后所绘制的顺序图。

绘制该顺序图中消息交互的步骤完全是按照表 4-8 所提供的用例文档中的基本事件流来进行的。

基本流 1——“用例起始于旅客现场需要预订房间”表明用例何时启动，在该顺序图中没有体现[④]。

---

① 严格来说，在分析交互期间所涉及的都是类的对象，而不是类；本书为了便于读者理解，仍采用类的概念，表明该类的所有对象均适用，而不特指某个具体对象。

② 一个职责分配模式，将在第 7 章再展开介绍，这些模式在设计时用得更多。

③ 方案 1 和方案 2 相对比较简单，在分析阶段经常选择这两种方案。笔者更倾向于方案 2，以便于设计阶段统一处理这类问题。而在设计阶段则需要考虑控制类的内聚性以及类之间的耦合关系，从而选择方案 3。

④ 实际实现时是服务员在主界面上选择“预订房间”功能。因此可以考虑定义一个主界面的边界类，服务员向该边界类发送预订房间的消息，再由主界面创建预定房间界面类（ReservationUI）。本书中的所有案例都没有考虑这一个主界面类，因此，基本流的第一句话都没有在顺序图中体现。

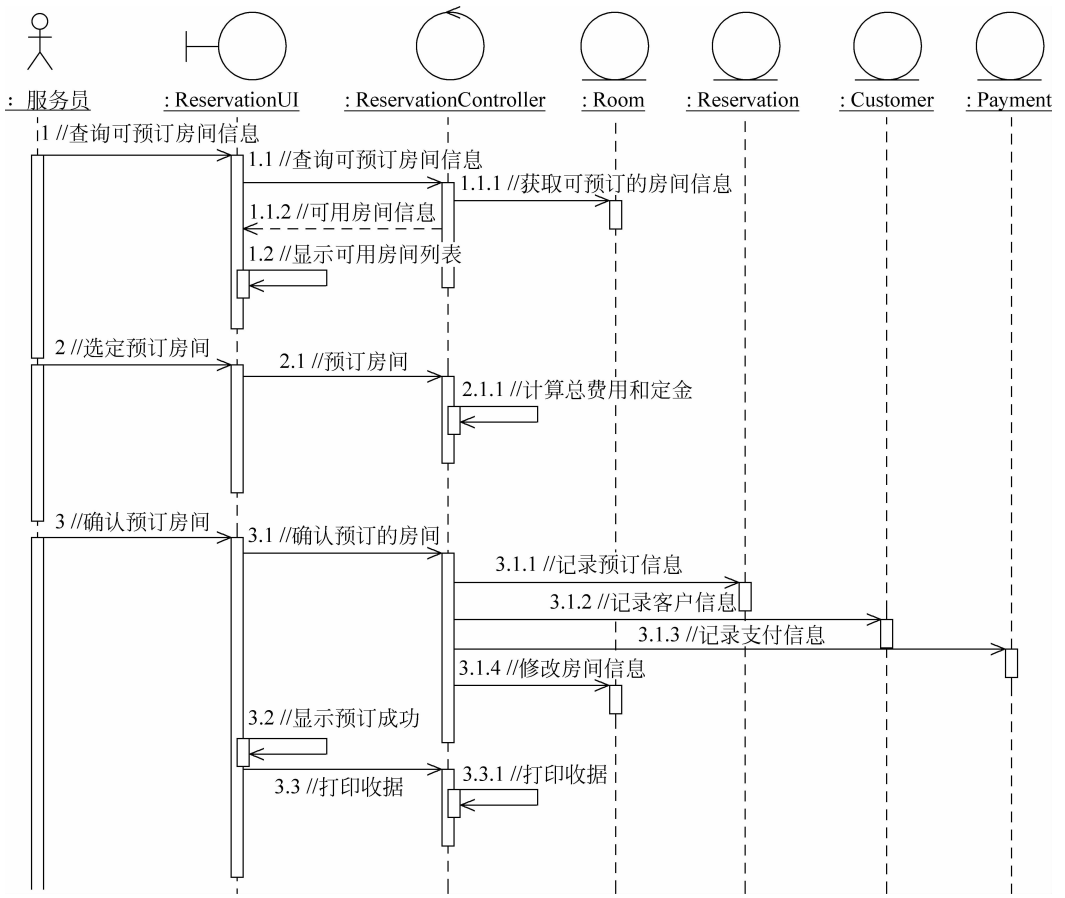

图 5-31 描述交互

基本流 2——“服务员按照旅客的要求设定查询条件来查询可预订的房间信息”，即服务员在界面上输入查询条件启动查询，该步骤对应顺序图中的 1 号消息“查询可预订房间信息”，该消息发往界面类，表明启动了一个查询动作。

基本流 3——“系统显示所有可预订的房间列表”是一个系统的处理动作，与基本流 2 对应，这两步就构成了图 4-20 所描述的一次交互，对应其中的 1(动作)和 4(响应)。在分析阶段，必须将系统如何进行响应，从而产生所需的结果描述出来，也就是要将这次交互的 2(验证)和 3(处理)表示出来，这就体现对需求进行分析的过程。目前，查询的请求到了界面类，但界面类只负责与参与者通信(即接收参与者消息、向参与者显示结果)，后续的处理需要交给控制类。所以，下一个消息(消息 1.1)就是界面类(ReservationUI)将查询的动作通知控制类(ReservationController)，由控制类负责后续的验证和处理(分析阶段的类图均采用这种方案：系统的内部处理均由控制类负责)。控制类接收到查询请求后，就不能再转发了，而要考虑应该如何实现这个查询请求。控制类首先可以分析查询条件是否正确(即对动作 2 进行验证)，这一细节在分析阶段也可以不用做太多考虑。其次就是进行处理，来获得可预订的房间信息。房间信息存储在什么位置呢？在当前分析阶段，所有的这些实体信息都保存在实体类中(实际实现是一般存储在数据库中，不过由于目前是分析阶段，还没有涉

及数据的存储问题和访问问题，这些与实现相关的技术应该在设计时再考虑）。因此房间类（Room）知道房间的信息，按照专家模式，控制类从房间类获得可预订的房间信息（即消息 1.1.1）。当然，此处需要强调的一点是，实际上控制类应该是在一个房间的集合中查找可用的房间，而不是向单个的房间对象发送消息，所以该顺序图中的房间对象是一个多重实例（即有多个房间对象）。在 Rose 中可以通过属性对话框修改该对象为多重实例，如图 5-32 所示。不过遗憾的是，在图 5-32 中并没有特别的标记来区分单个对象和多重实例（在通信图中，多重实例有更形象的表示）。

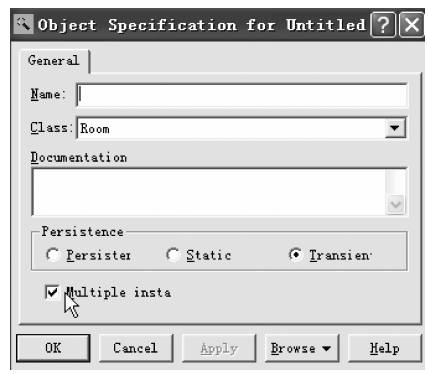

图 5-32　将对象设置为多重实例

1.1.2 为一个返回消息，表明控制类向界面类返回一个可预订的房间列表，这里专门把该消息返回表示出来，以强调返回的内容。界面类接收到返回值后，需要刷新界面显示返回的结果，所以消息 1.2 就是界面类发给自己进行刷新的动作，从而显示出可预订的房间，完成基本流 3。

基本流 4——“服务员为旅客选择所需的房间，并输入预订的时间和天数”，即服务员在界面上输入预订房间的信息，消息 2 即用于完成该步骤。

基本流 5——“系统计算所需的总费用和预付订金金额”，界面类接收到预订请求后，提交给控制类（消息 2.1）计算预订的费用和金额。为了计算预订的费用和金额，需要知道房间的信息、预订的日期、天数和当前日期。房间信息在房间类中，而预订的日期、天数由界面接收到后交给控制类，当前日期控制类可以从系统时间中获得。按照前面的专家模式，采用方案 2 由控制类实现比较合适，消息 2.1.1 即表示了这个过程。

基本流 6——“旅客现场用现金支付所需的订金”，这是一个旅客的手工行为，不需要系统处理，顺序图中也不需要体现。

基本流 7——“服务员将支付信息记录到系统中，并进行预订操作”，即服务员录入用户提交的订金信息，并提交，消息 3 即完成该步骤。

基本流 8——“系统保存本次预订信息，显示预订成功消息”。同样，界面类把保存的请求提交给控制类（消息 3.1），控制类需要将与本次预订相关的全部信息保存下来（这些信息包括预订信息（Reservation）、旅客信息（Customer）和订金支付信息（Payment）），并修改房间的状态。这些保存和修改的动作都是记录在实体类中的，因此消息将发往相应的实体类，消息 3.1.1～3.1.4 即表明了这个过程。所有操作完成后，界面类显示成功消息（消息 3.2）。

基本流 9——“系统打印预订收据后，用例结束”，界面类启动打印请求（消息 3.3），这些打印的信息来自于多个不同的实体类。因此，按照专家模式的方案 2，由控制类完成打印业务（消息 3.3.1），基本流的分析结束。

当前的顺序图只分析了基本事件流。此后，可以针对系统不同的备选事件流重新绘制另一张顺序图进行分析。当然对于简单的备选流（即不会引用新的对象、不会定义新的职责的分支）没有必要画单独的顺序图，只需要在基本事件流的顺序图中添加适当的注释进行说明即可。如：本用例中的基本流 3 可能产生找不到可预订房间的分支 A-1，此时可以直接

结束该用例，而不需要做进一步处理（即不会产生新的对象和行为）；因此可以在消息 1.1 上添加相应的注释来说明这种情况，而不需要再针对该 A-1 备选流单独进行分析。

3）验证行为

最后，当完成了整个交互过程后，还需要去验证该交互中所描述的消息序列是否是可实现的。验证的基本思路是从最后一个序列反向进行，不断地询问每一个对象是否有能力履行消息所要求的职责，如果无法履行这些职责，则可能要对顺序图中的消息序列进行调整，如添加新的对象或新的消息。此外，如果前一阶段找出来的类不能满足当前消息序列，则可能漏掉了一些类，需要重新识别新的类添加到备选架构中，并更新顺序图。

3. 分析交互实践

下面将利用前面所阐述的分析交互的技术，详细讲解“旅游申请系统”迭代 1 所要处理的 6 个用例实现的分析交互过程，它们根据图 5-11 给出的跟踪关系图与图 5-6 给出的用例对应，而分析交互的主要依据则是第 4 章中的表 4-12～表 4-16 给出的用例文档（缺少该系统的“登录”用例文档，可参见“旅店预订系统”的“登录”用例文档）。

1）建模指南

当前所要绘制的顺序图是针对每一个用例实现的，这些用例实现已经在第 5.2.2 节添加完毕，本节将为每一个用例实现绘制其交互模型。绘制的基本过程是在该用例实现上单击鼠标右键，在弹出的快捷菜单中选择 New|Sequence Diagram 命令，如图 5-33 所示为“登录-用例实现”添加一个顺序图，修改该图的名字为“基本场景”，用来表示该用例基本场景的顺序图。如果还需要绘制其他场景的顺序图（如输入密码错误），则按照同样的步骤再新建一张顺序图，并取名为“备选场景-密码错误”，用于区分基本场景的顺序图。

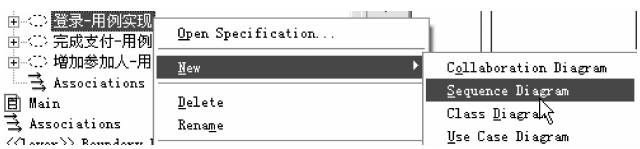

图 5-33 新建顺序图

图 5-34 为该系统 6 个用例实现绘制顺序图后的分析模型的组织结构图，目前每个用例实现只绘制了基本场景的顺序图，没有考虑任何备选场景。下面将详细讲解通过绘制顺序图来对每个用例实现进行分析的过程。

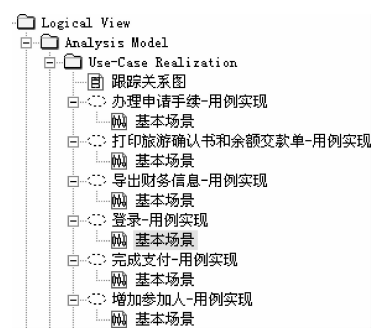

图 5-34 添加顺序图后分析模型的组织结构

2）登录-用例实现

图 5-35 给出了“登录-用例实现”的基本场景顺序图。该顺序图描述的基本流程为：前台服务员在登录界面上输入用户名和密码后请求登录（消息 1）；界面类获得这些信息后请求控制类来验证用户名和密码（消息 1.1）；控制类根据用户名和密码来查询已有的用户信息（消息 1.1.1），当控制类找到该用户后，创建该用户对象（消息 1.1.2）并返回给界面类（返回消息 1.1.3）；最后，界面类显示登录成功信息，并显示系统主界面（消息 1.2）。

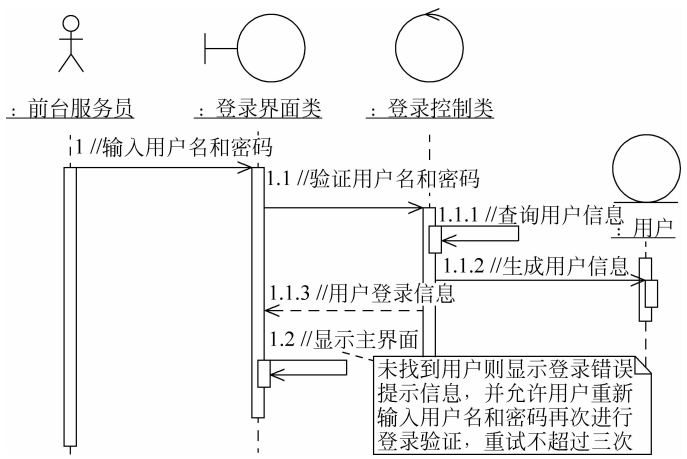

图 5-35　登录-用例实现的基本场景顺序图

针对该顺序图的绘制，重点需要说明 4 个方面的问题。在以后的顺序图中也存在类似的情况，此处进行统一说明。

（1）与查询相关的类职责的实现方案。对于控制类的职责 1.1“验证用户名和密码”，考虑其最终的实现策略一般是在数据库中查询用户表，从而判断该用户名和密码是否一致；在后面的顺序图中也会存在很多类似的查询职责。然而在分析阶段由于并没有考虑具体的数据库的设计和数据库表的访问策略（这些应该是在后面的设计中再进一步考虑的，分析阶段并不考虑这种具体实现技术）。因此，在分析阶段，针对这类问题，可以将其封装在控制类内部，认为控制类自身就可执行此类查询，从而获得指定的用户信息，至于如何查询则不做进一步展开。这也是提出控制类的一个初衷，即将一些复杂的业务逻辑封装在控制类的内部。在该顺序图中，控制类为了实现该职责，首先执行一次查询（即消息 1.1.1），之后根据查询结果生成用户信息（即消息 1.1.2），并将该用户信息返回给界面（即返回消息 1.1.3）。这种处理策略与图 5-31 的策略有所不同，图 5-31 中针对房间信息的查询是由一个房间的多重实例来完成的（即认为房间信息全部存在该多重实例中），即在该多重实例中查询满足条件的实例，这是分析阶段的两种处理策略。本书倾向于采用此处的策略，即为控制类添加单独的查询职责来完成对数据集合的查询，根据查询结果来创建相应的实体类。对于该查询职责分析阶段不做进一步细化，而在设计阶段再做单独处理。

（2）用户对象的处理。从顺序图 5-35 中可以看出，用户对象的位置比其他对象都要低，这表明该对象的生命周期并不是在用例启动时就存在的，而是在用例执行过程中动态生成的。此处是在控制类查询到该用户信息后，通过消息 1.1.2 生成用户对象来存储用户登录信息，即消息 1.1.2 实际上是一个创建消息。不过由于 Rose 不直接支持此类消息，因此图中针对该消息并没有特殊的标记，本书中将通过“生成……”这样的消息名来区分创建消息。此外，严格来说，针对所有创建消息，其对象的生命线应该从此消息之后才开始，即顺序图中该对象应该放在较低的位置（如图 5-35 中的用户对象）。不过本书中考虑绘制顺序图的方便性和图形布局等问题，后面的顺序图中都没有采用此画法，而是直接把所有的对象都放在最上面。而针对某些不是用例启动时就存在的对象，可以通过创建消息来明确该对象

的生命周期是何时开始的。如图 5-36 所示，通过消息 1.1.1 和 1.1.2 两个创建消息就可以明确旅游团对象和路线对象的生命周期是从该消息才开始的，而不是用例启动时就存在的。当然，对于那些直接支持创建消息建模的工具，工具本身会自动将该对象放置在更低的位置，如图 5-27 所示的效果。

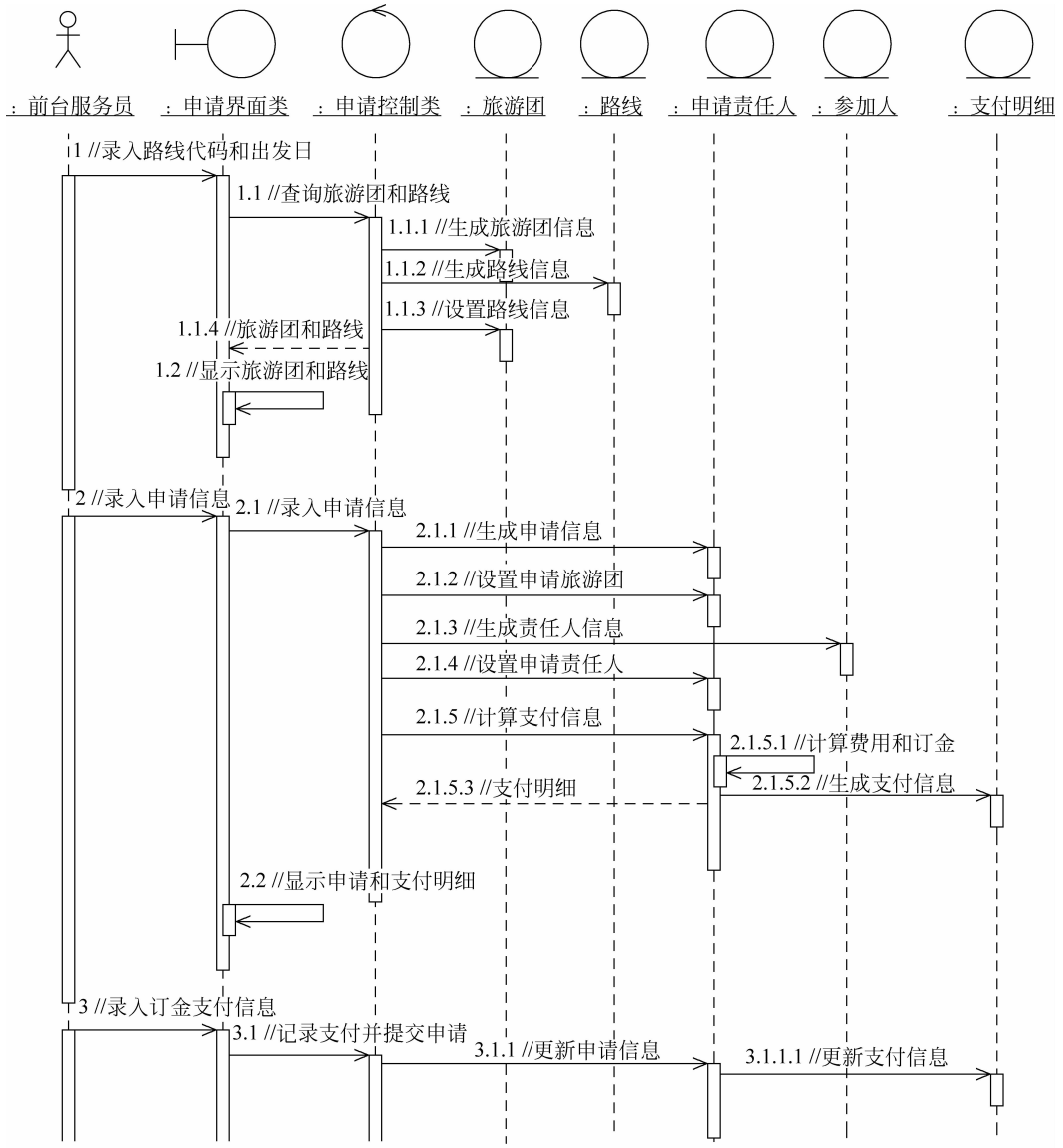

图 5-36 办理申请手续-用例实现的基本场景顺序图

（3）备选事件流的处理。当前顺序图描述的是基本路径的交互，即用户输入了正确的用户名和密码。而针对可能出现的异常情况，如用户名或密码错误，在该顺序图中通过注释的方式进行了说明，如图 5-35 所示。针对简单的备选流，这种方式是可行的；但针对一些复杂的备选流，则应该考虑再重新绘制相应的顺序图来进行分析。图 5-37 就是处理用户登录错误时最多只能重试三次的顺序图。

(4) 参与者“前台服务员”和实体类“用户”的区别。虽然在该顺序图中,这两个对象实际上所指的是同一个人,即那个当前使用系统的人。但在系统中这是两个不同的角色:前台服务员是作为系统的一个外部激励,从而来启动该用例,并完成与系统的交互,是系统外的参与者;但用户则是系统所要管理的与登录相关的用户信息,是系统内的实体类。要注意这两个角色的区别和联系。

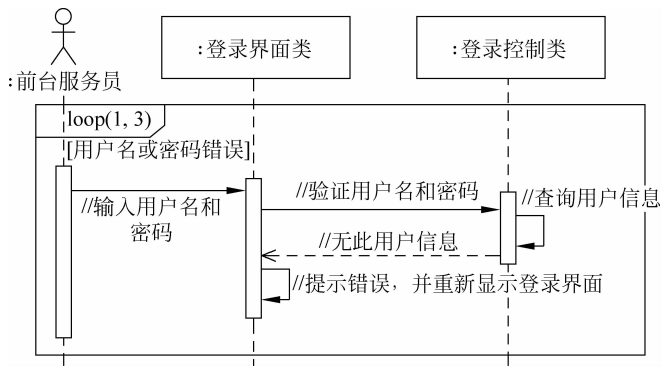

图 5-37 利用 loop 交互片段处理登录错误

3) 办理申请手续-用例实现

图 5-36 给出了“办理申请手续-用例实现”的基本场景顺序图。该顺序图描述的基本流程如下所示。

前台服务员在申请界面上首先录入路线代码和出发日期(消息 1)。界面类根据这些信息向控制类查询所要申请的旅游团和路线信息(消息 1.1),控制类执行查询请求(与图 5-34 的查询相比,此处为了简化顺序图,省略了调用自身查询的消息,可以认为消息 1.1 即包含这个能力,后面的顺序图也类似地省略了该查询动作)。根据查询结果生成相应的旅游团对象(创建消息 1.1.1)和路线对象(创建消息 1.1.2),并将这两个对象关联起来(消息 1.1.3,即设定该旅游团对应的路线),最后返回旅游团对象(返回消息 1.1.4)。界面对象接收到返回结果后,进行刷新,从而显示所查询到的旅游团和路线信息。基本事件流的第 1 步～第 4 步已完成。

之后,用户确认需要申请该旅游团,前台服务员向界面录入用户的申请信息(消息 2),界面类将申请内容提交给控制类(消息 2.1),控制类针对申请信息的不同方面交由相应的实体类进行处理。首先生成一个申请对象(消息 2.1.1),并与旅游团对象关联(消息 2.1.2),表明该申请所对应的旅游团;其次,生成一个参加人对象(消息 2.1.3),来存储申请的责任人信息,并在申请对象中关联责任人信息(消息 2.1.4);最后,控制类要求申请对象计算本次申请有关的支付信息(消息 2.1.5),申请对象根据自身的信息(包括申请的大人人数、小孩人数、申请日期等)和所关联的旅游团信息(包括旅游团的价格、出发日期等)来计算费用和订金等支付信息(消息 2.1.5.1),并生成支付明细对象来保存相应的结果(消息 2.1.5.2),并将结果返回给控制类(返回消息 2.1.5.3)。控制对象将本次申请的明细返回给界面后(省略了该返回消息),界面类进行刷新显示(消息 2.2)。

最后,用户根据系统计算出来的订金进行支付,服务员通过界面类将用户的支付信息录

入到系统中(消息 3),界面将支付结果提交给控制类(消息 3.1),控制类根据支付结果更新申请对象的状态(消息 3.1.1),同时申请对象也会把支付情况记录到支付明细对象中(消息 3.1.1.1)。

针对该顺序图的绘制工作,还存在三个方面的问题需要进一步说明。

(1) 命名对象。正如前面所展示的顺序图所示,图中大部分对象都是匿名对象,并不需要指定特定的对象名称,只需要明确所属的类类型,从而泛指所有通过该类所构造的对象。但在有些特殊场合下,为了特指该类某个特殊的对象,需要为该对象取特定的名字。在该图中,为“参加人”类定义了一个“责任人”对象,从而区分不同参加人的身份。这意味着,在当前的办理申请手续-用例实现中,只需要记录那个作为责任人的参加人信息,而不需要维护其他普通参加人的信息。

(2) 设置对象间的关联。在该顺序图中,有诸如 1.1.3、2.1.2、2.1.4 等“设置某某信息”的职责,这类职责实际上是建立两个对象之间的关系。比如消息 1.1.3 的职责是为旅游团设置其路线信息:通过控制类查询到了旅客所要申请的旅游团和路线,并分别存储到两个类中,但对于某个旅游团对象而言,应该要指定该旅游团是针对哪个路线的,即要建立旅游团类和路线类之间的关联关系,这个操作就是消息 1.1.3 所要达到的目标。同样的道理,消息 2.1.2 用于建立申请和旅游团之间的关系,而消息 2.1.4 用于建立申请和责任人之间的关系。有关类之间的关系的细节可以进一步参考第 5.5.3 节。

(3)“计算费用和订金”职责的处理。对于此类与业务逻辑相关的、涉及多个对象的职责,很多时候都是由控制类来处理的,但该顺序图中将该职责分配给申请类。而这种分配策略在此处是非常合适的,因为计算费用和订金所需的申请人数信息和旅游团费用信息都与申请类之间存在关系,即通过申请类可以明确地获得这些内容,因此按照“专家模式”(见第 7.4.1 节),这是一个很合理的方案。

4) 增加参加人-用例实现

图 5-38 给出了“增加参加人-用例实现”的基本场景顺序图[①]。该顺序图描述的基本流程为:前台服务员根据申请编号查询所要添加参加人的申请信息(消息 1),界面对象将该查询请求提交给控制对象(消息 1.1),控制对象查询到该申请后生成申请对象(消息 1.1.1)以及该申请的责任人对象(消息 1.1.2),并建立这两个对象之间的关系(消息 1.1.3),界面对象根据控制对象返回的结果刷新显示申请信息(消息 1.2)。

之后,前台服务员首先添加责任人和责任人的联系人信息(消息 2),界面类先把责任人的信息提交给控制类(消息 2.1),控制类区更新责任人的详细信息(消息 2.1.1);接着界面对象将联系人信息通知给控制对象(消息 2.2),控制对象通知责任人对象(消息 2.2.1)生成其联系人对象(消息 2.2.1.1)。

最后,对于每一对参加人和联系人对象,用户循环添加其他参加人和联系人的信息(消息 3)。界面类首先将录入的其他参加人的信息提交给控制类(消息 3.1),控制类生成参加人对象保存该信息(消息 3.1.1),同时添加到申请对象中,以建立申请和参加人之间的关系

① 由于图 5-38 较宽,为了图形的整体效果,图中最上面一排对象并没有放在一个水平线上,而是交错放置,以便能够完整地显示对象名和类名。

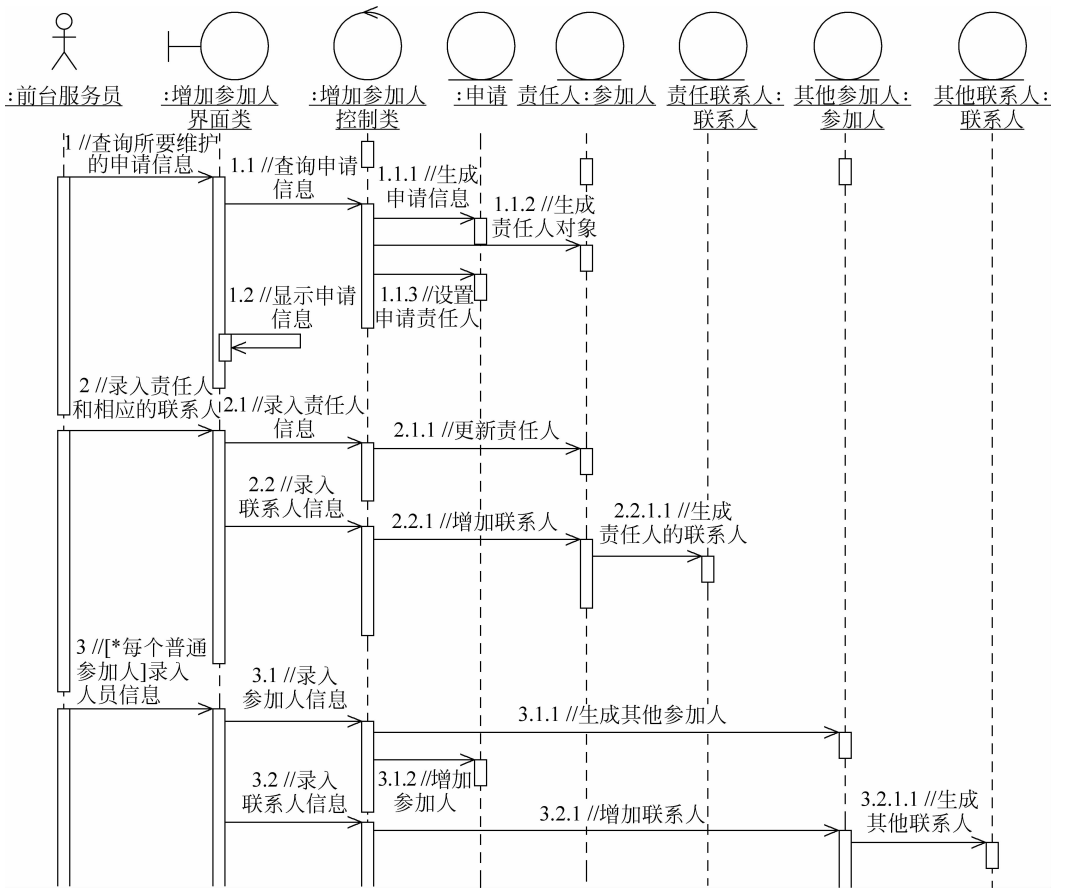

图 5-38　增加参加人-用例实现的基本场景顺序图

(消息 3.1.2)；界面类最后将联系人的信息提交给控制类(消息 3.2)，控制类通知参加人对象(消息 3.2.1)生成其联系人对象(消息 3.2.1.1)。

该顺序图一个典型的特点就是很多实体对象均给出了特定的对象名，在前一个顺序图中已经说明了原因。虽然责任人和其他普通参加人都是参加人类，但在当前用例中扮演着不同的角色，同理责任人的联系人和其他联系人也可通过对象的名字进行区分。

该顺序图中另一个问题就是针对循环的处理：消息 3 以及相应的嵌套消息是针对每一对参加人和联系人进行循环执行的。然而，早期的顺序图(UML 1.x)并没有提供一种机制来直接表示消息的循环；因此只能采用一种辅助手段来描述，本书的顺序图中采用[]表明循环的条件，并在条件前面加上“*”来表示循环操作，其效果如图 5-38 中的消息 3。当然，UML 2 的顺序图专门提供了交互片段这种机制来表示循环、分支等各种非顺序的操作，当使用支持 UML 2 的工具进行建模时可以适当使用该机制。有关交互片段的详细内容请参见本节的“4.顺序图中的交互片段”部分。

5) 完成支付-用例实现

图 5-39 给出了“完成支付-用例实现”的基本场景顺序图。

该顺序图描述的基本流程如下所示。前台服务员首先通过界面类查询所要完成支付的

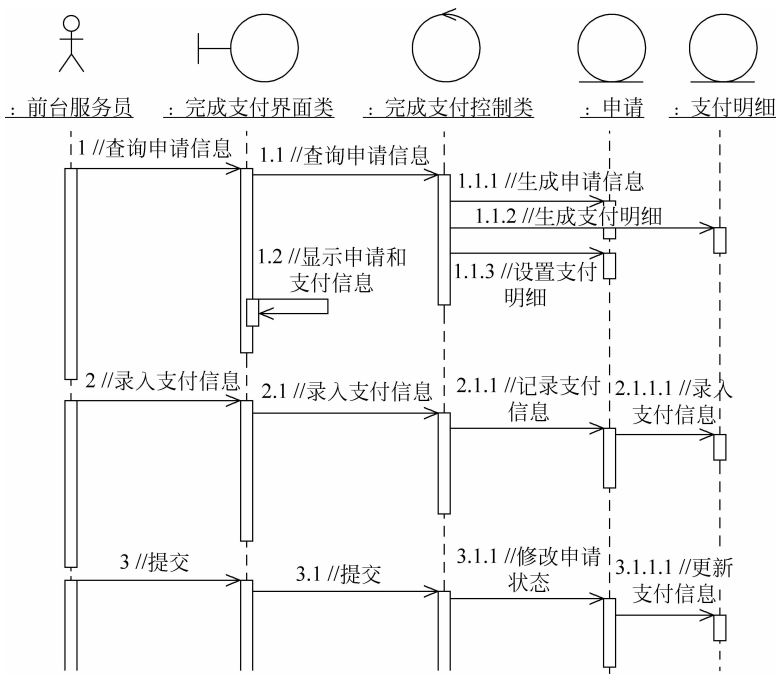

图 5-39 完成支付-用例实现的基本场景顺序图

申请信息(消息 1),界面类将查询条件提交给控制类(消息 1.1),控制类进行查询后生成申请信息(消息 1.1.1)和相应的支付信息(消息 1.1.2),并建立这两者之间的关系(消息 1.1.3);界面类根据返回的申请和支付信息进行刷新显示(消息 1.2)。

之后,前台服务员针对所查询到的申请录入其相应的支付信息(消息 2),界面类将接收到的输入传递给控制类进行后续处理(消息 2.1),控制类将与这些支付信息通过申请类(消息 2.1.1)写入支付明细对象(消息 2.1.1.1)。

最后,前台服务员统一提交所录入的支付信息(消息 3),界面类通知控制类进行提交(消息 3.1),控制类通知申请类修改申请的状态为已支付(消息 3.1.1),并同时更新申请状态(消息 3.1.1.1)。

针对该顺序图需要重点说明的一个问题就是对支付明细类的操作。可以看出由控制类生成支付明细对象后,将该对象与申请对象建立关联;此后针对该对象的所有操作都通过申请对象来完成(如后续的记录支付信息和更新支付状态等操作)。这与前面的一些处理不同,控制类并不直接操作支付明细对象,这样就可以降低控制类与支付明细类之间的耦合度。由于支付明细是完全依附于某个申请的,因此这两者之间存在紧密的关联关系(具体的关系定义参见第 5.5.3 节),而支付明细类和控制类之间并没有固有的联系,因此现阶段尽可能降低这两者之间的耦合度是有利于后续的设计是一种尝试,也为类设计提供了一些参考。

6) 打印旅游确认书和余额催款单-用例实现

图 5-40 给出了“打印旅游团确认书和余额催款单-用例实现”的基本场景顺序图。该顺序图描述的基本流程如下所示。收款员工首先通过界面查询本次需要处理的所有申请(消

息 1)，界面类将查询请求提交给控制类(消息 1.1)，控制类针对所查询出的每一个申请循环执行(消息 1.1.1)生成相应的申请对象(消息 1.1.1.1)和支付明细对象(消息 1.1.1.2)，并为其建立关联(消息 1.1.1.3)；界面类接收到返回结果后进行刷新显示(消息 1.2)。然后，收款员工针对每一个要处理的申请执行打印动作(消息 2)，界面类通知控制类打印旅游确认书(消息 2.1)，而针对尚有余额未支付的申请控制类还要打印交款单(消息 2.2)。

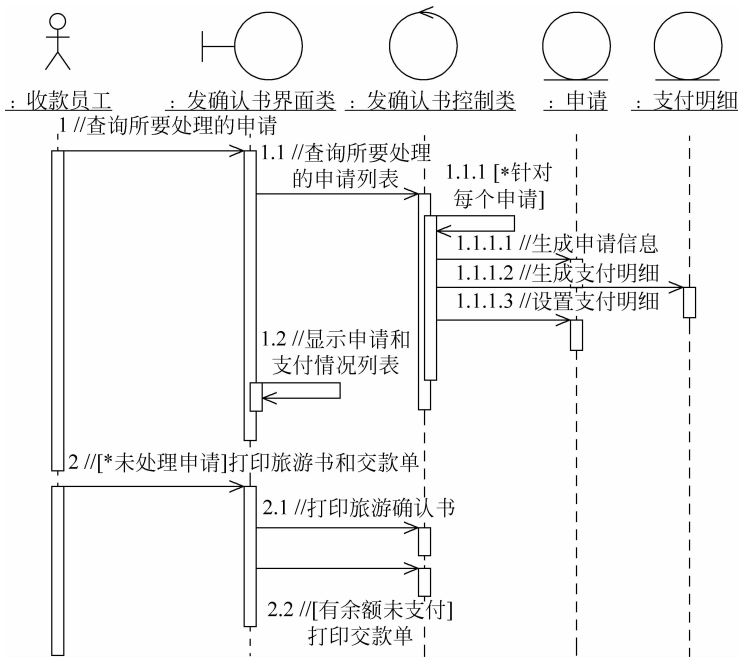

图 5-40　打印旅游确认书和余额催款单-用例实现的基本场景顺序图

此顺序图主要说明的问题就是循环和分支的处理工作。在图 5-40 中，针对所有未处理的申请存在两个循环操作，既要生成其申请和支付对象，同时又要执行打印旅游书和交款单。为了表示这两个循环操作，依然采用前面提到的“[*]”的方式，并将循环条件放在“ * ”号后面。而消息 2.2 只是在“[]”内部放置了守卫条件，并没有“ * ”，这表明该消息只是一个判断，当条件为真的时候执行，否则不执行；这是一种对分支的表示方法。

7) 导出财务信息-用例实现

图 5-41 给出了“导出财务信息-用例实现”的基本场景顺序图。该顺序图描述的基本流程如下所示。当系统达到预先设定好的时间后就通知界面类启动该用例(消息 1)，界面类通知控制类开始导出财务信息(消息 1.1)，控制类首先查询所有需要导出的财务信息(消息 1.1.1)，并依次生成相应的支付明细对象来存储这些需要导出的财务信息(消息 1.1.1.1)。最后控制类将所有记录在支付明细对象中的财务信息提交给财务系统接口(消息 1.1.2)，财务系统接口将这些信息最终导入到外部财务系统中(消息 1.1.2.1)。

针对该顺序图的绘制，也存在两个方面的问题需要进一步说明。

(1) 时间参与者和相应界面类的使用。与前面几个用例实现都是由普通用户启动不同，该用例是由系统自动启动的，按照用例建模的习惯用法专门定义了一个时间参与者来启动该用例。对于该参与者来说，它并不能像普通用户那样向系统输入数据或执行某些操作，

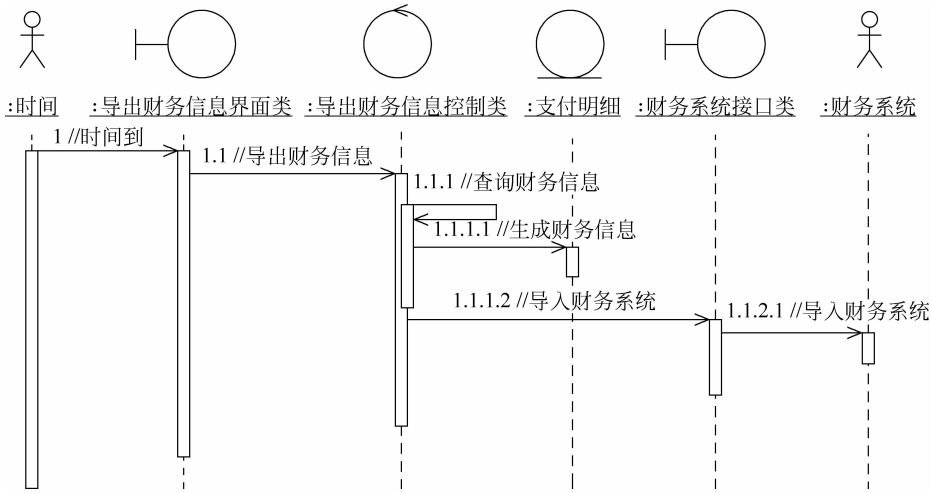

图 5-41 导出财务信息-用例实现的基本场景顺序图

它只能传递一个“时间到”的消息来启动用例实现(图 5-41 中的消息 1)。同理,导出财务信息界面类实际上也并不是一个系统界面,它也只能通知控制类可以启动某个操作(图 5-41 中的消息 1.1),而没有其他任何功能。从分析的角度来说,也完全可以不需要这个界面类,不过本书为了保持整个分析过程的一致性,还是保留了它。

(2) 外部系统接口类和相应参与者的使用。财务信息最终要导入到外部系统中,这个外部系统在当前系统中作为一个参与者而存在,对于导出财务信息控制类而言,为了访问财务系统这个参与者,也需要通过边界类来完成,这个边界类就是财务系统接口类。从图 5-41 中可以看出,不同于界面类接收参与者的输入并提交给控制类,财务系统接口类是接收控制类的请求,去操作代表外部系统的参与者(即消息的方向是反向的,这与图 4-18 的用例图中参与者和用例之间关联的方向是一致的)。

4. 顺序图中的交互片段

正如其名,顺序图主要用于描述顺序的执行流程,这也是为什么前面在编写用例文档时要将基本事件流和备选事件流分开描述的原因所在。当然对于基本流中分离出来的简单分支或循环,也可以直接在基本流的顺序图中进行描述。虽然 UML 1.x 中的顺序图并没有提供一种通用的描述机制,但是可以选择采用一些辅助的方式来描述,建议的描述规则如下所示。

- 执行的条件用“[]”括起来描述,表示条件为“真”时才执行,否则不执行,如图 5-40 中的消息 2.2。
- 循环条件要在条件前加上“*”来描述,表示条件为真时重复执行,如图 5-27 中的消息 3 和图 5-40 中的消息 2.1。
- 其他的约束用“{ }”括起来,可在任意位置进行描述。

当然,对于复杂的分支场景,还是应该绘制更多的顺序图单独进行描述,而不是过分地采用这种非正式的手段进行描述。

此外，UML 2 为顺序图提供了一种新的机制来更方便地描述分支、循环、并发等各种非顺序的情况，这就是交互片段(Interaction Frame)。

交互片段将顺序图中的若干消息和对象封装为一个片段，针对这个片段可以实施不同的操作，从而来表示这个片段是以选择、循环还是并行等各种非顺序方式执行。在顺序图中，交互片段显示为一个矩形区域，该矩形区域内的消息和对象为一个整体；矩形区域的左上角有一个写在小五边形内的文字标签，用来表示所执行操作的类型。有各种不同类型的操作符，从而实现不同的控制结构，典型的操作符有可选(opt)、选择(alt)、循环(loop)、并行(par)等。

1) 可选片段

操作符为 opt，类似于 C++语言中的 if-then 控制结构，表示该片段只有在守卫条件成立时才能够执行，否则跳过该片段往后执行。守卫条件是一个用方括号括起来的布尔表达式，它可能出现在片段内部任何一条生命线的顶端，还可以引用该对象的属性。

2) 选择片段

操作符为 alt，类似于 C++语言中的 swith 控制结构，该片段的主体用水平虚线分割成几个分区(类似于 switch 语句的 case 子句)。每个分区都有一个守卫条件，表示当守卫条件为真时执行该分区。此外，每次最多只能执行一个分区，如果有多于一个守卫条件为“真”，那么选择哪个分区是不确定的(这一点与 C++不同，C++中是执行第一个为“真”的 case 子句)，而且每次执行的选择可能不同。如果所有的守卫条件都不为真，那么该片段将不被执行。此外，还可以定义一个 else 分区，该分区的守卫条件为[else]，如果其他所有的分区都不为真，则执行该分区(类似于 switch 语句中的 default 子句)。

3) 循环片段

操作符为 loop，类似于 C++语句中的 while 控制结构(也可利用 loop 后面的参数模拟 for 循环)，表示该片段在守卫条件为“真”的情况下循环执行；一旦守卫条件为“假”，则跳过该片段往后执行。

4) 并行片段

操作符为 par，该片段的主体也被水平虚线分割成几个分区，不同的分区可能覆盖不同的生命线，表示当进入该片段后，这几个分区要并行(或并发)执行。每个分区内的消息是顺序执行的，但是并行分区之间的消息的相对次序则是任意的(即不同并行分区内的消息可以并发地执行)。在当今多核环境下，并行(或并发)程序正日益普及，而该操作符将在这类应用程序的设计中被广泛应用。

图 5-37 展示了如何利用 loop 交互片段来处理登录用例的备选事件流 A-1 和相应的约束规则(即最多只能重试三次)。其中 loop 后面的圆括号内的数字(1,3)表示循环主体应当执行的最少次数和最多次数。当用户启动该用例后，初始情况下并没有输入用户名和密码，所以条件(用户名或密码错误)为“真”，至少执行一遍该循环；在循环内，用户输入用户名和密码，控制类进行验证；只要用户名或密码不正确，该循环就会继续。但是，如果超过了三次，那无论如何循环都会结束，即不再允许用户重新登录。

其他几类交互片段的使用方法基本相同，读者可以使用支持 UML 2 的建模工具去尝试使用这些交互片段。

5. 顺序图的分拆和引用

交互片段的基本思想就是将那些非顺序执行的部分封装为一个片段，针对该片段实施各种不同的操作，从而表达各种非顺序的执行。这种交互片段的思想其实也完全可以扩展到对整个顺序图的封装，即将一个完整的顺序图封装为一个片段。这个片段采用 sd (sequence diagram)操作符(事实上，这不能算操作符，因为并不实施任何操作；只是用一个标签来说明该片段为一个完整的顺序图)，操作符后面可以写上该顺序图的名字。

把顺序图封装为片段之后，就可以在其他顺序图或交互纵览图中进行引用(即实现交互序列的复用，采用 ref 操作符)，从而来分析更复杂的用例行为，这就是顺序图的分拆和引用的基本思想。在 UML 2 中，可以从纵向或横向来分别实现顺序图的分拆。

纵向分拆是沿着顺序图的纵轴(即时间序列)的方向，将某一段时间段内的若干对象之间传递的一组消息拆分出去，同时在当前顺序图中给出相应的引用标识，指向另一个展现被拆分出来的内部消息传递过程的顺序图。通过这种方式允许不同顺序图引用相同的消息交互片段，避免了无谓的重复描述，提升了模型内容的可维护性。图 5-42 展示了利用纵向分拆技术绘制的用户利用自动售货机购买商品场景的顺序图。

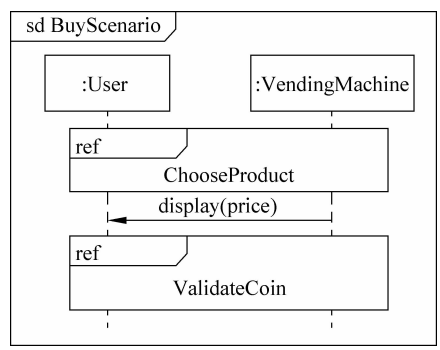

图 5-42 顺序图的纵向分拆和引用

该顺序图被封装为一个整体片段，利用 sd 表示该片段为一个顺序图，顺序图的名字为 BuyScenario，以便在其他场景中引用。而图中两个对象分别来自用户(User)类和自动售货机(VendingMachine)类。而在描述这两个对象的消息序列时，首先采用 ref 片段引入另外一个顺序图 ChooseProduct，即用户通过该顺序图所描述的场景来选择所需要购买的产品，之后自动售货机根据所选择的产品显示其价格(display(price)消息)，最后又通过引用第三个顺序图(ValidateCoin)来接受并验证用户所投入的金额是否满足要求。当然，对于 ChooseProduct 和 ValidateCoin 这两幅顺序图需要单独进行描述，以说明这些场景的实现过程。

横向分拆是沿着顺序图的横轴(即对象序列)的方向，在全时间范围内，将一组对象及其相互之间的消息封装为一个与其他对象交互的“组合结构”，该“组合结构”指向相应的具体消息传递情形。图 5-43 显示了利用横向分拆来实现自动售货机验证用户所投入的金额是否合适的场景。

图 5-43(a)所示的顺序图描述了自动售货机接受用户投入金额，然后拒绝所投入的货币的场景。该图的一个典型特点是自动售货机对象后面跟着关键字 ref 和标识符 Decomposition，这说明该对象并不是由一个简单类生成的对象，而是引用另一组合结构(Decomposition 为该组合结构的名字)。组合结构是对多个对象间的链接和消息的封装，用于提供更大粒度复用。图 5-43(b)详细说明了该组合结构内部的交互模型(即动态模型)，另外还可以通过组合结构图进一步说明其内部的静态模型。

除了对单个顺序图的分拆和引用之外，不同的顺序图之间还可能存在顺序的或非顺序的关系，如用例的备选事件流一般是在基本事件流的某个步骤出现分支而分离出去的，这样描述备选事件流的顺序图和基本场景的顺序图之间也应该是在某个消息中出现分支。针对

这种整体相关的顺序图之间的关系可以通过 UML 2 中新增的交互纵览图来表示。交互纵览图是活动图和顺序图的结合体，它能够像活动图一样表达流转逻辑；所不同的是，它不是描述活动间的流转，而是描述顺序图的流转，通过 ref 交互片段来引用所要描述的顺序图。图 5-44 的交互纵览图描述了自动售货机初始化、验证投币金额和出货几个场景之间的流转关系。图 5-44 中的起点、终点、控制流、决策点等元素均来自于活动图，其含义也完全相同，而图中的主体部分为利用 ref 片段引用已有的顺序图，从而描述这些顺序图之间的流转逻辑。

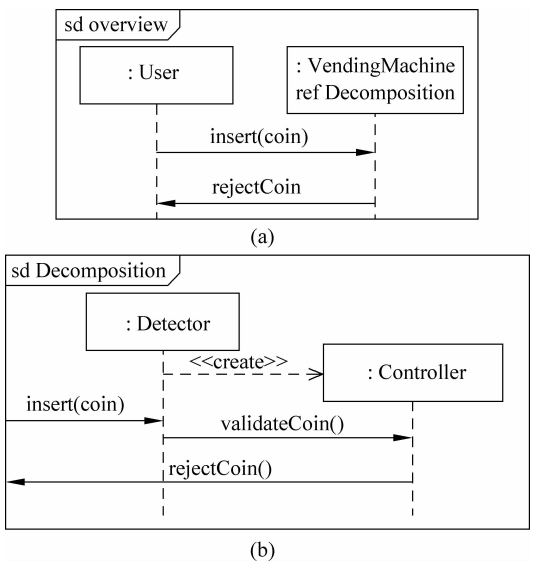

图 5-43　顺序图的横向分拆和引用

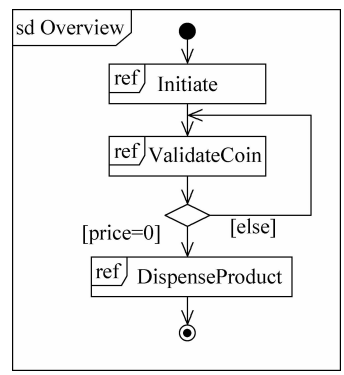

图 5-44　在交互纵览图中引用顺序图

5.4.4　完成参与类类图

分析交互的过程主要关注用例实现的交互行为特征，是用例实现的动态视图。而对于每一个用例实现而言，为了完成交互需要通过类产生相应的对象，同时对象间的消息传递也需要类之间的关系来支持。这些与用例实现相关的类以及类之间的静态关系就需要通过静态类图来描述。该静态类图是针对特定的用例实现来绘制的，用来表示参与该用例实现的类以及关系，简称参与类类图(View Of Participating Classes Class Diagram，VOPC 类图)。

参与类类图是用例实现的静态视图，针对每一个用例实现绘制一张类图。由于已经完成了该用例实现的动态视图，因此绘制静态视图的过程将主要依据动态视图来完成。

(1) 参与类类图中的类来自顺序图中的对象。对象都要由相应的类构造出来，因此将这些类放置到参与类类图中。它们是在识别分析类的过程中找到的，不过在那个阶段是按照备选架构分层组织的，而现阶段则需要从用例实现的视角来绘制，同时还需要进一步描述它们之间的关系。需要说明的是，此阶段并不会发现新的类，只是从另一视角进一步描述类。此外，顺序图中的参与者并不是系统的成分，因此不作为类出现在参与类类图中。

(2) 在参与类类图中，类之间的关系来自顺序图中的消息。对象 A 要向对象 B 发送消息，就必须能够访问到对象 B，因此 A 和 B 之间存在着一定的关系。它们之间具体是哪种

关系将在第 5.5.3 节和后面的设计中进行详细讲解，本节将主要使用关联关系。当然，早期业务对象模型中所发现的实体类之间的关系也可以在当前参与类类图中继续存在。此外，对于实体类之间更多的内在关系可能还需要进一步分析和定义，这部分内容同样将在第 5.5.3 节进行说明。

由于是用例实现的静态视图，因此参与类类图是绘制在每个用例实现下面的。在相应的用例实现上单击鼠标右键，在弹出的快捷菜单中选择 New|Class Diagram 命令，将类图的名字修改为 VOPC 类图，图 5-45 展示了为“办理申请手续-用例实现”添加了参与类类图后，模型的组织结构。

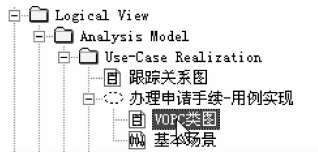

图 5-45 添加参与类类图后，分析模型的组织结构

绘制参与类类图的过程就是将相应的类放置到类图中，一般按照边界类、控制类和实体类的顺序从上到下放置，之后即根据顺序图中的消息为这些类添加关系。图 5-46 展示了旅游申请系统中“办理申请手续-用例实现”的参与类类图。

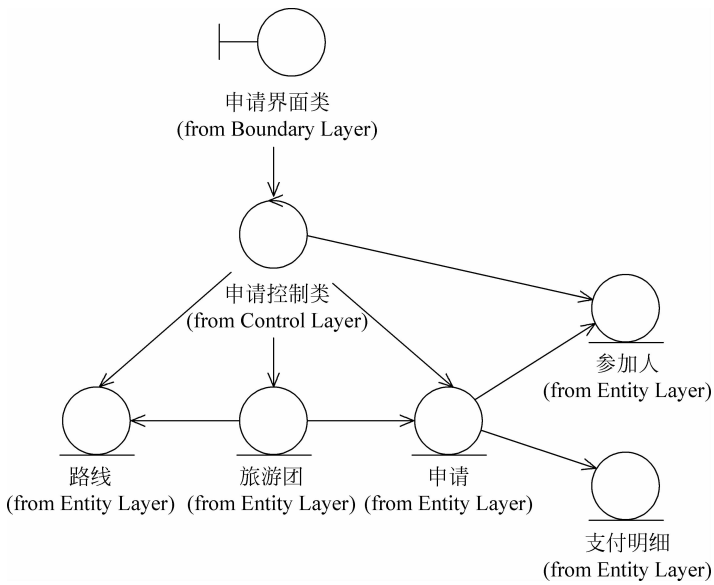

图 5-46 办理申请手续-用例实现的 VOPC 类图

首先，由于界面类需要将参与者输入的信息传递给控制类进行后续处理，因此申请界面类有到申请控制类之间的关联关系，注意该关联关系的方向与顺序图中的消息是一致的。其次，控制类需要操作各种实体类以存取所要处理的数据，因此控制类有到实体类的关联关系。而支付明细实体类并不需要通过控制类进行访问，因此控制类和支付明细类之间没有关系。最后，各个实体类之间也存在着一些固有的关系，有关这些关系是如何定义出来的，我们将在第 5.5.3 节中详细讲解。

从该参与类类图中还可注意到另外一个问题：这些类之间关系的依赖方向与备选架构中各层的依赖方向是一致的。按照前面分层架构的思想，界面类放在边界层中，控制类放在控制层中，实体类放在实体层中。相应的界面类依赖于控制类，控制类依赖于实体类。这也是软件架构所要描述的另一个问题，即各层之间的依赖关系决定了类之间关系的方向。

图 5-47 展示了旅游申请系统中“导出财务信息-用例实现”的参与类类图。

该参与类类图典型的特点是存在两种不同的边界类：其中界面类还是去访问控制类，因此有到控制类的关联关系。然而，接口类则是由控制类来操作的，即控制类操作该接口类来访问外部财务系统，因此控制类有到该接口类的关联关系（见图 5-47）。而这种关联的方向却违背了类备选架构所定义的依赖方向，因为备选架构中并没有控制层到边界层的依赖。为了解决这个问题，需要重新调整备选架构，添加从控制层到边界层的依赖关系，调整后该系统的备选架构如图 5-48 所示，注意新添加的控制层到边界层的依赖关系的表示方法。

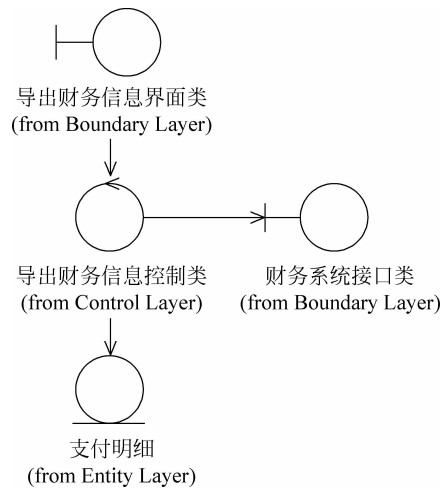

图 5-47 导出财务信息-用例实现的 VOPC 类图

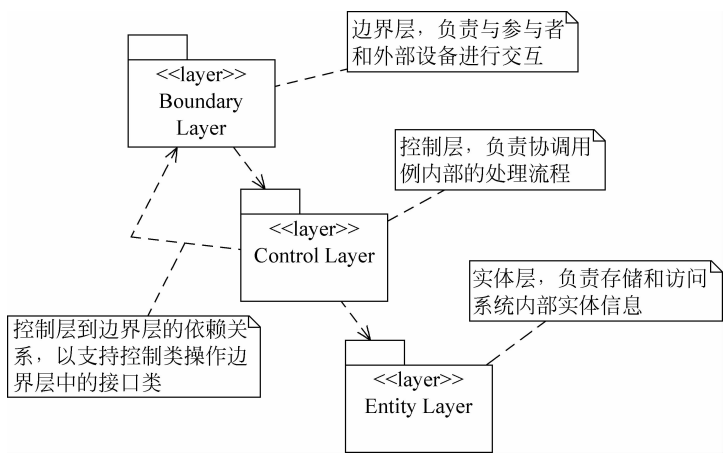

图 5-48 旅游申请系统调整后的备选架构

参与类类图的绘制相对比较简单，本书中只以这两个用例实现为例讲解基本绘制方法和技巧。有关其他用例实现的参与类类图，读者可以自行绘制。

5.4.5 处理用例间的关系

前面所阐述的构造用例实现过程是针对每一个用例实现独立进行的，并没有考虑多个用例实现之间可能存在的关系。在前面针对旅游申请系统的分析过程中，采用的是重构前未引入关系的用例模型，并通过定义首次迭代来简化用例实现，从而屏蔽了用例模型中用例间的关系[①]。然而，不管如何处理，由于用例建模中用例之间可以定义包含、扩展和泛化这

① 本书如此处理的目的是屏蔽用例关系给分析过程带来的复杂性，以便于集中讲解用例分析的基本方法和技巧；实际项目中可以在保留这些关系的基础上定义迭代，并进行分析。而有关这些关系的处理则在本节单独讲解，读者也可根据本节所描述的方法在保留用例关系的基础上重新完成旅游申请系统的分析过程。

三种关系，因此，在分析的过程中也需要针对这三种用例关系做进一步处理。

1. 包含关系

包含关系表达了基用例对子用例（即被包含用例）的直接引用，而这种引用在分析阶段即表现为基用例实现对子用例实现的交互的引用。这类引用可以采用 UML 2 所提供的 ref 片段来表达：在基用例实现的顺序图中通过 ref 引用子用例的顺序图片段，以包含子用例实现的行为。而对于子用例实现来说，基用例实现就是它的外部用户，来使用它所定义的行为。包含关系的具体分析过程如下所示。

（1）为子用例定义一个边界类，可以将该边界类看作是基用例对子用例的调用界面，由基用例的控制类来启动该边界类。

（2）基用例实现通过横向分拆或纵向分拆的方式引用子用例实现的交互。

下面以第 4.5.1 节的图 4-21 中所描述的“管理参加人”（分析阶段只考虑增加参加人子流程）和“查询申请信息”之间的包含关系为例，介绍其分析过程。

图 5-49 首先给出了子用例“查询申请信息-用例实现”基本场景的顺序图，该顺序图描述了如何根据提交的查询申请请求查询出指定的申请信息以及相关的责任人和支付明细信息。

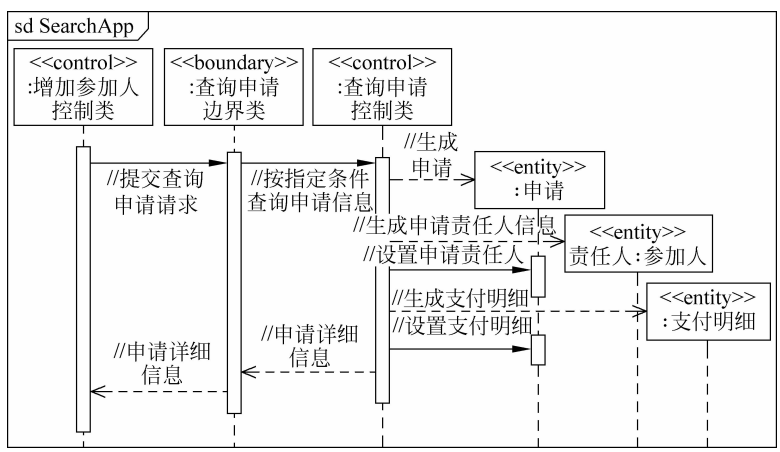

图 5-49 “查询申请信息-用例实现”基本场景的顺序图

该顺序图与之前图形的典型区别在于其外部启动对象（即第一个对象）并不是一个参与者，而是其基用例的控制类。这表明该交互并不是直接对系统外部提供服务的，而是为基用例服务的。此外，包含关系中往往存在多个基用例（即多个用例包含一个子用例），此时需要考虑不同的基用例是否对子用例的交互有不同的要求。如果有，则针对不同基用例的控制类绘制相应的顺序图来描述其交互的不同；否则，只需要选择其中一个典型的控制类来绘制顺序图即可。在旅游申请系统中，“查询申请信息”用例虽然有两个基用例——“管理参加人”和“完成支付”，但考虑到当前所描述的查询申请信息的交互对于这两个基用例都是适用的，因此只需要绘制这一个顺序图即可。

另外还需要说明一点的是有关“查询申请信息边界类”的含义。正如前面所阐述的，存在用户界面和系统接口两类边界类；而此处并没有明确指明当前的边界类到底是哪一类，这意味着该边界类可能是一个用户界面类，也可能是一个接口类。如果是一个用户界面，则

表明基用例直接包含子用例提供的查询界面来完成查询功能；如果是一个接口类则表明基用例通过调用子用例的查询接口获得查询结果，而查询条件的输入和查询结果的输出等均由基用例的界面类来实现。

图 5-50 给出了保留包含关系后“增加参加人-用例实现”的顺序图的片段，为了节省篇幅，图中省略了查询之后的业务操作，这些操作与图 5-38 是一致的。

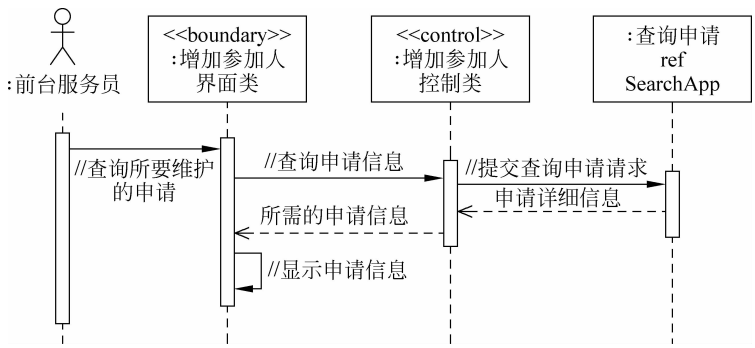

图 5-50　利用横向引用实现包含关系

与图 5-38 所不同的是，查询动作并不是由“增加参加人控制类”来实现的，而是通过横向引用将该业务提交给“查询申请”组合结构，其内部实现逻辑在该组合结构中描述（即图 5-49）。当然，此处倾向于将查询申请边界类作为一个接口来考虑。如果是一个用户界面，则可以考虑直接由前台服务员向该组合结构发送查询消息，而省略掉图中所描述的通过增加参加人界面类和控制类进行中转和显示的所有消息。

关于包含关系的分析，最后还需要说明一点就是当使用不支持 UML 2 的建模工具（如 Rose）时，由于没有分拆和引用机制，此时可以直接在基用例中引入子用例的边界类来表达这种调用关系。如图 5-51 所示，在“增加参加人-用例实现”中直接将查询请求提交给子用例查询申请的边界类，而至于具体的查询实现细节则仍保留在子用例中，不在基用例中展示。

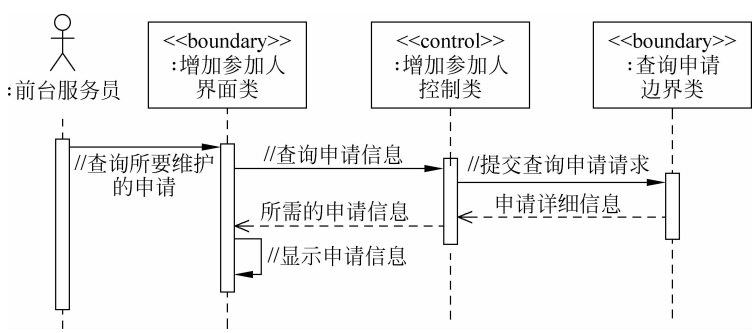

图 5-51　利用边界类实现包含关系

2. 扩展关系

与包含关系中的直接引用子用例行为不同，扩展关系中基用例并不知道子用例（即扩展用例）的存在，而是通过扩展点描述了可扩展的行为接口，由子用例去实现这些扩展点。因

此，对于用例分析来说，扩展关系中的基用例实现也就不能直接引用子用例实现的交互片段，而是定义并使用指定的接口来描述其所需要的行为。相应地，其子用例实现则是来实现这些接口所描述的行为。扩展关系的具体分析过程如下所示。

（1）为基用例的每一个扩展点定义一个边界类。该边界类是一个系统接口类，描述了基用例所需的扩展行为。

（2）针对每一个扩展用例，以该边界类的职责作为启动消息，分析职责的内部实现。

下面以第4.5.1节图4-22中所描述的“导出财务信息”和“记录日志”之间的扩展关系为例，介绍其分析过程。

图5-52给出了“导出财务信息-用例实现”在出现异常的场景下的顺序图。与图5-41所描述的基本场景不同。在该场景中，当消息1.1.1.1生成财务信息产生异常时，这个异常在需求阶段表示为一个扩展点；在分析阶段，将该扩展点定义为记录日志接口类；当异常情况出现时，将该异常信息提交给记录日志接口类（消息1.1.1.2），由该接口类来负责后续处理。同样，在导入到财务系统过程中产生异常时也需要提交给记录日志接口类。

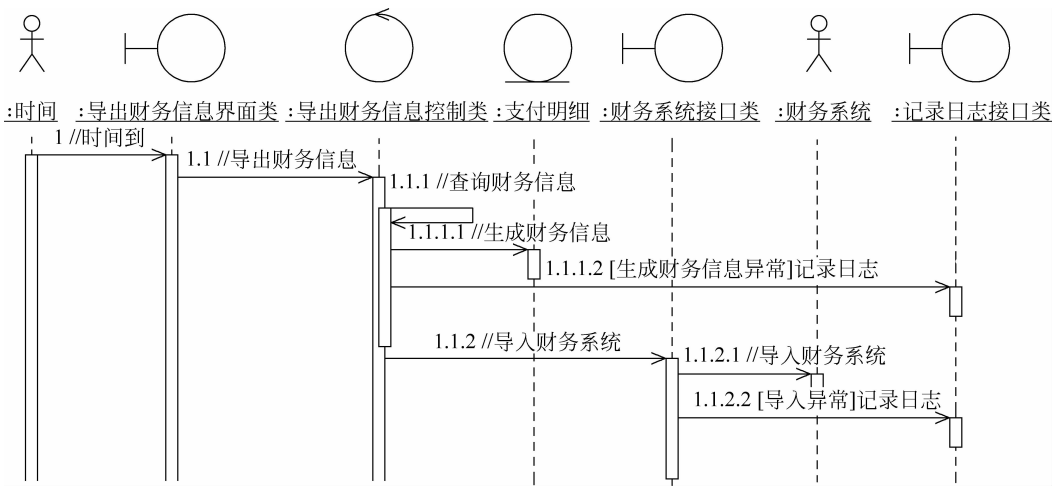

图5-52 导出财务信息-用例实现的出现异常辅助场景顺序图

至于记录日志的实现细节，则需要在子用例“记录日志-用例实现”中描述。图5-53描述了该用例实现对应的记录日志职责（即图5-52中的消息1.1.1.2和消息1.1.2.2）的实现过程。

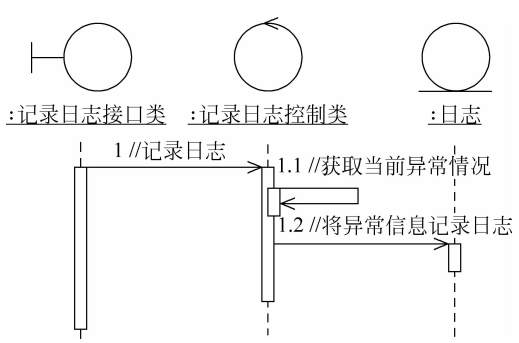

图5-53 记录日志-用例实现的记录日志职责顺序图

注意该用例实现启动对象为记入日志的接口类，该接口类可能存在多个职责，针对每一个职责都需要绘制一张交互图来进行分析。此外，还需要明确的是，该接口类和包含关系中的边界类不同。这是一个明确的接口，对于每一个子用例实现，虽然接口相同，但可能存在不同的实现逻辑（需要定义不同的实现类来实现该接口，有关接口的设计细节可以参见第9.2节的相关内容），从而实现对基用例灵活的扩展。

3. 泛化关系

泛化关系的处理则相对比较麻烦，因为在用例分析阶段，并没有一种机制来直接描述交互间的继承层次，而只能采取一些折中的方案。

考虑到特化用例是完全继承并复用泛化用例的行为，因此一种可行的方案是参照包含关系的分析思路，在特化用例实现中通过 ref 交互片段来直接引用泛化用例实现的交互。不过，由于在泛化关系中，特化用例可以具体化或重新实现泛化用例的行为。这就意味着特化用例实现的交互可能会对其所引用的泛化用例实现的交互进行修改，而这种修改是无法通过 ref 交互片段来表示的。

而另一种方案则是泛化用例实现和特化用例实现相对独立地分析：首先分析泛化用例实现；之后将该用例实现的交互作为基础来重新分析特化用例实现，并通过适当的机制（如注释、约束）来分别标记哪些交互是继承而来的，哪些交互是针对泛化用例实现进行重新定义的和哪些交互是新增的。这种方案的缺点是当修改了泛化用例实现时，会影响到所有的特化用例实现，从而需要重新绘制所有特化用例实现的相关模型。

5.4.6 总结：构造用例实现

构造用例实现是整个用例分析最核心的工作，其最终目标是获得实现用例行为所必需的分析类，并利用这些分析类来描述其实现逻辑。具体的分析过程是针对每一个用例实现，完成下面4个步骤。

（1）完善用例文档。结合分析的目标和分析策略，完善待分析的用例文档。

（2）识别分析类。从完善后的用例文档中识别分析类，包括三类分析类：边界类、控制类和实体类。其中的重点是实体类，要结合架构分析中的关键抽象和名词筛选法来识别实体类。

（3）分析交互。利用识别的分析类，利用交互图来分析用例实现的交互过程。早期的迭代重点在基本场景，后期的迭代可能需要针对不同的场景绘制更多的交互图。

（4）完成参与类类图。根据交互图中的消息和实体类内在的关系来绘制参与当前用例实现的类的类图。

一般情况下，上面4个步骤是针对每一个用例实现独立完成的；而对于存在关系的两个用例实现，则需要采用特定的技术将其分析成果联系起来。

最后，还需要强调一点的是，本书中是按照先后顺序依次介绍这4个步骤的。但这并不意味着在实际项目中这些步骤会严格按照顺序进行。事实上，每一步之间是紧密联系、相辅相成并且互相制约的，它们往往会进行若干次迭代，从而交错进行。虽然说必须要识别一定数量的分析类才能够分析交互；但在分析交互的过程中也可能识别出新的分析类（特别是实体类）。同样，在完成参与类类图时可能结合分析类之间的关系对交互图中的消息进行适当的调整。

5.5 定义分析类

通过构造用例实现验证了需求的可实现性。即可以利用识别出的分析类和它们之间的交互来达到用例的目标；这是整个分析过程最核心的工作。然而，对于面向对象的系统来说，所描述的这些交互最终都应在相应的类中来定义并由类的对象来实现。虽然在构造用例实现的过程中已经获得了类的基本定义，但那是在一个个用例实现的基础上完成的，主要关注的是用例事件流的交互过程，而对单个类自身的特征和行为缺少统一的考虑。因此，下一阶段就需要将注意力集中到每一个分析类本身，在关注类自身定义的基础上再重新评估每个用例实现的需要。此外，一个类及其对象常常参与多个用例实现，此时更需要从类整体角度去协调用例的行为。

定义分析类的最终目标就是从系统的角度明确说明每一个分析类的职责和属性以及类之间的关系，从而构造系统的分析类视图；并根据这些视图来描述和理解目标系统，从而为后续的设计提供基本的素材。

5.5.1 定义职责

职责(Responsibility)是要求某个类的对象所要履行的行为契约，它说明了该对象能够对外提供哪些行为，在设计中将演化为类的一个或多个操作。构造用例实现的过程实际上就是进行类的职责分配的过程，通过向目标对象发送消息来定义其所要履行的职责，而这也是面向对象分析和设计的最核心工作。

从职责所履行的功能来划分，有两种类型的职责。

(1) 做(Do)型职责：对象能够完成某些动作的职责；包括某个具体的业务操作、发起其他对象执行动作或者控制和协调其他对象内部的活动。

(2) 知道(Know)型职责：对象提供自己所知道信息的职责；包括提供或修改自身的私有的数据、获取与之关联的对象信息或自己派生出来或者计算出来的事物。

知道型职责取决于对象自身的属性和关系，而做型职责则反映了对象的行为特征。在分析阶段，控制类主要由做型职责构成，来协调和发起实体类与边界类的操作；而实体类则主要由知道型职责构成，为控制类提供其内在数据；边界类也主要由知道型职责构成，来为用户展示其操作界面，另外包括一些做型职责来发起控制对象进行交互。分析阶段的重点在于做型职责，相对而言知道型职责比较简单明了。为了能够有效地获得这些职责，可以从两个方面来考虑。

(1) 从交互图中的消息获得职责；对于每一条消息，接收该消息的对象需要提供相应的职责来响应。为此，当接收该消息的对象没有相应的职责进行处理时，就需要在该对象的类中添加这项职责以提供需要的行为。

(2) 从非功能需求中获得其他职责；交互图主要关注系统的功能需求，而系统的非功能需求也会要求类提供相应的行为来处理，为此可能需要添加与之相关的职责。不过由于分析阶段是通过限定分析机制来描述非功能需求的，并没有进行展开分析，因此此方面的职责更多地将在设计期间获得。

获得类的职责后需要采取一种通用的方法表示出来，在分析阶段可以采用“分析”操作

和文本描述两种方式定义职责。

1. “分析”操作

分析类的职责是该类目标操作的雏形，这些职责在设计阶段最终都会演变为类的操作。因此，在分析阶段可以直接采用操作的形式来说明类的职责；不过为了与类的目标操作区别对待，需要采用一些特殊的标记来说明，这些操作也被形象地称为“分析”操作。一种通用的表示方法就是在操作的前面加上“//”，以表明该操作是一个分析操作，目前只是做了初步的职责定义，这与前面分析交互时消息的表示方法是一致的。

图 5-54 给出了从“办理申请手续-用例实现”的基本场景顺序图中提取出来的分析操作。为了操作的显示更加直观，图中的分析类并没有采用前面所介绍的特殊图标表示，而是采用标准的类图格式，并利用构造型标签区分不同的分析类。

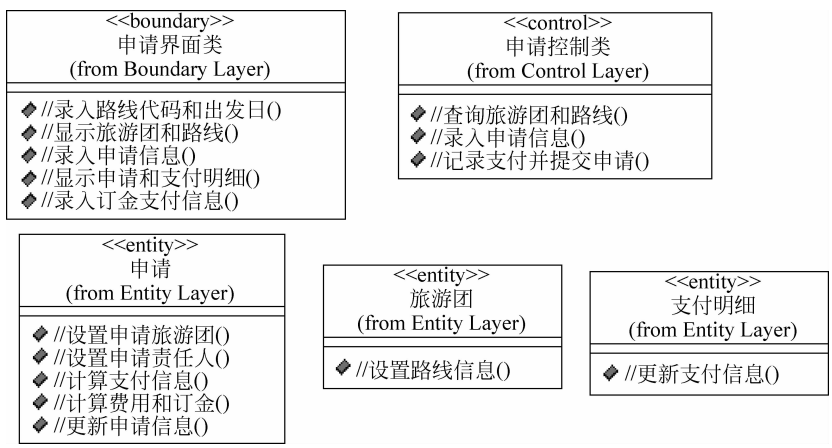

图 5-54　为分析类添加“分析”操作

从图 5-54 中可以看出，申请界面类包括 5 个分析操作，这些分析操作与顺序图中的该对象接收到的消息是一致的(分别对应顺序图中的消息 1、消息 1.2、消息 2、消息 2.2 和消息 3)。此处需要注意的是消息 1.1.4 虽然也是由申请界面类接收，但由于它是一个返回消息，实际上是消息 1.1 的返回结果，并不是对申请界面类操作的调用，因此消息 1.1.4 不是申请界面类的分析操作。同理，返回消息 2.1.5.3 也不是申请控制类的分析操作，申请控制类只有消息 1.1、消息 2.1 和消息 3.1 所对应的三个分析操作。

申请实体类也包括 5 个分析操作，分别对应顺序图中的该对象所接收到的消息 2.1.2、消息 2.1.4、消息 2.1.5、消息 2.1.5.1 和消息 3.1.1。此处需要说明的是消息 2.1.1“生成申请信息”接收对象虽然也是申请类，但由于它是一个创建消息，因此并没有定义相应的分析操作来支持该消息。这是因为，与普通的消息直接由操作来响应不同，创建消息一般是通过类的构造函数来创建对象，而且其调用也是通过一些特定的方式来实现的(不同的编程语言，其调用机制可能有所不同，如 Java 中通过 new 操作调用)，因此这些与对象创建和删除等生命周期相关的操作在分析阶段一般不做处理，也就不需要定义分析操作来表示。同理，图 5-54 中其他实体类的创建消息也都没有定义相应的分析操作。

2. 文本描述

文本描述是指采用一种约定的文档模板，在该文档中对类及其职责进行详细的描述。并没有一种统一的类职责描述文档，项目组会根据类似项目的经历和项目的特点编写所需的类职责文档。不过，早期面向对象的方法提供了一种 CRC 卡技术可以很好地用于描述类的职责，在分析阶段完全可以借助于 CRC 卡来定义职责。

CRC 卡(Class-Responsibility-Collaborator cards，类-职责-协作卡)虽然不是 UML 的组成部分，但是在定义类的职责和描述与对象之间的协作方面与 UML 类图相比有它自己的特点，在对象分析和设计中被广泛的采用。

CRC 卡是由一系列卡片组成，每张卡对应一个类。卡中包括类名、类的职责和一系列完成这些职责的协作对象(即参与该职责的对象)，表 5-6 给出了一种简单的 CRC 卡的格式。

表 5-6 CRC 卡

类 名	
职责 1	职责 1 的协作
职责 2	职责 2 的协作
……	……

这些卡片在分析的初期即被开发出来(识别出分析类后，即可建立其 CRC 卡)，并随着分析设计过程的深入而不断地完善。这种卡片作为 UML 类图和交互图的有益补充，在类的职责分配期间扮演着重要的角色。表 5-7 给出了申请实体类的 CRC 卡。

表 5-7 申请类的 CRC 卡

申 请	
设置申请旅游团	旅游团
设置申请责任人	参加人
计算支付信息	旅游团、路线、参加人、支付明细
计算费用和订金	旅游团、路线、参加人
更新申请状态	支付明细

3. 保持类职责的一致性

正如前面所提到的，有些分析类跨越多个用例实现(如旅游申请系统中的申请类)。这意味着从不同的用例实现中能够为同一个类发现不同的职责，而这些职责之间可能存在不一致甚至冲突的地方。因此，定义类的职责时还应确保它们有一致的职责，可以从以下几个方面着手来保持类职责的一致性。

- 当一个类的职责互不相干时，可以将这些不相干的职责分成不同的类，并更新交互图。
- 当两个(甚至更多个)不同的类有相似的职责时，合并这些相似的职责形成新的类，并更新交互图。
- 当在分析另一个用例实现时，发现一种更好的职责分配方案，此时可以返回之前的交互图并重新采用新的分配方案来分配职责。
- 只有一个职责的类虽然没有什么问题，但对它存在的必要性是值得怀疑的。可以考

虑把这仅有的职责合并到其他类中。当然，没有职责的类就更没有存在的必要了。

虽然，在分析阶段可以就类职责的一致性进行评估，但这并不是分析的重点。只要适当注意上面所提到的这些问题并进行相应的处理即可，没有必要在保持类职责一致性方面浪费更多的时间，这些工作将会在设计中结合相关的设计理论和原则去重点考虑。

5.5.2 定义属性

属性(Attribute)是类的已命名特性(Property)，它用来存储对象的数据信息，是没有职责的原子事物。类可以有任意数目的属性，也可以根本没有属性。在分析阶段需要从分析的类职责入手，描述其必备的属性，并为其确定适当的名字。属性名应当是一个名词，它清楚地表达了属性保留的信息；同时还可以进一步利用文字详细说明属性中将要存储的相关信息。对于属性的类型定义，在分析期间应当来自业务领域，而与特定的编程语言没有关系。为了能够有效地获得类的属性，可以从以下几个方面来考虑。

- 在通过用例文档识别分析类的过程中，也可同时发现类的属性。这主要包括：接在所有格后面的名词或形容词(即某某的属性)、不能成为类的名词以及字段列表中所描述的数据需求。
- 作为一般业务常识，是否有从类的职责范围考虑所应包括的属性。
- 该业务领域的专家意见以及过去的类似系统。

图 5-55 给出了从“办理申请手续-用例实现”中提取出来的实体类的属性。分析阶段属性的定义主要是针对实体类而言的，这些实体类存储了系统业务所需要处理的各类数据；而边界类和控制类则主要是为了分析阶段的职责分配而提取出来的，与业务本身关系不大，因此它们在分析阶段也很少存在需要表示的属性。

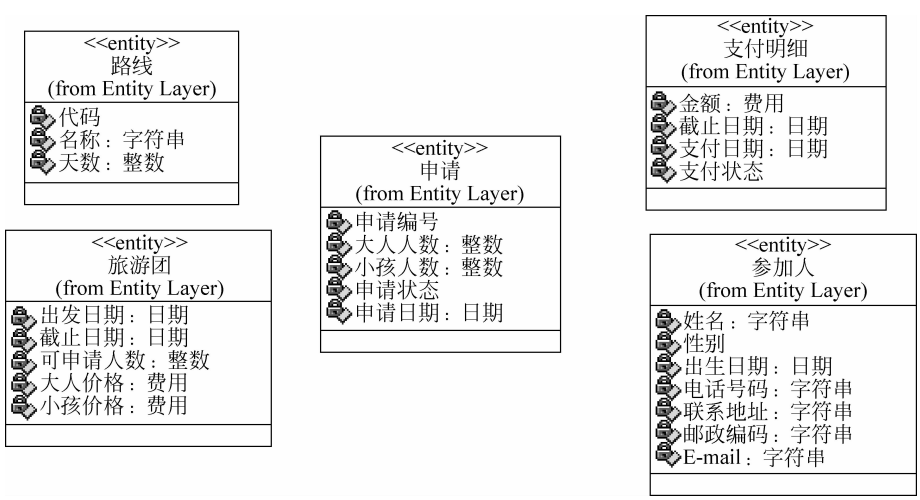

图 5-55 为分析类添加属性

从图 5-55 中可以看出，图中的大部分属性都是通过用例文档中的字段列表获得的，如旅游团类的出发日期、截止日期、大人价格、小孩价格；申请类的大人人数、小孩人数等。而有些属性则是根据业务常识而定义的，如申请类应该需要有一个申请编号属性。另外还有一些属性则是进一步考虑类似系统或分配职责过程中对象行为的要求，如申请类的申请状

态属性、支付明细类的支付状态属性等，是因为这些类的对象需要根据所处的状态不同做不同的处理，当然这类属性可以在设计时进一步明确定义。

而对于属性类型的定义，此处为了表示方便仍然采用中文表示；同时这些类型应该是用户可以理解的业务术语。如路线类的名称为字符串类型、旅游团的出发日期为日期类型、大人价格为费用类型；这些类型与编程语言无关，并不考虑其实现。当然对于某些属性也可做进一步定义，如字符串类型说明其最大、最小长度，费用类型可以说明其精度等。另外，还有一些属性没有指定类型，如路线类的代码属性、申请类的申请状态属性、参加人的性别属性，这说明这些类型在分析阶段并没有明确，设计时可以根据设计的要求选择合适的类型。如路线类的代码属性是采用整数，还是字符串或者其他类型等，可以结合数据库设计、性能等设计问题一起考虑，从而选择出合适的类型。

5.5.3 定义关系

系统的对象不能孤立地存在，它们之间需要频繁地通过消息进行交互从而执行有价值的工作，并达到用例的目标。为了完成这种对象间的交互，就要求交互的对象必须能够访问到对方。对象间的这种联系称为链接(Link)，对象通过链接互相协作。

如果两个对象间存在链接，那么它们的类之间也必定存在某种语义联系。即对象间的直接通信，必须要求这些对象的类之间以某种方式相互了解。类之间的这种联系被称为关联(Association)，而对象间的链接实际上是它们的类之间关联的实例。

1. 对象间的链接

链接是两个对象之间的语义联系，它允许消息从一个对象发送到另一个对象。面向对象的系统包含很多不同的对象和连接这些对象的链接。消息通过链接在对象之间传递，一旦接收到消息，对象将调用相应的操作来响应该消息。可以通过对象图来描述这种对象间的链接，通过通信图[①]来从对象间链接的角度描述对象间的消息传递。

对象图是显示系统某个时刻的对象及其关系的图，它是在特定瞬间对系统某部分的快照，表明当前时刻所存在的对象以及对象间的链接。图 5-56 为旅游申请系统某时刻某个申请以及参加人之间的对象图。

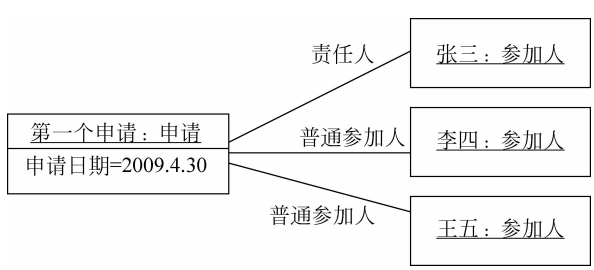

图 5-56 某个申请与其参加人关系的对象图

从图 5-56 中可以看出“第一个申请”对象与三个“参加人”对象相链接。其中为了区分责任人和普通的参加人，可以在链接的末端指定对象所扮演的角色，图 5-56 中张三为责任

① 有关通信图的相关细节和进一步的使用方法请参见第 6.6.3 节。

人，而李四和王五则是普通参加人。此外，对于每个对象还可以指定其特定的属性值，如图 5-56 中第一个申请对象，指定其申请日期属性为“2009.4.30”；当然其他属性也有相应的取值，但在当前图中并没有体现出来。

由于系统中的对象是动态的，不同时刻的对象可能不同，而且系统中的对象可能很多；因此，一般情况下并不会专门为整个系统绘制对象图，而主要是针对系统中某个关键部分的一些特殊场合绘制对象图，从而便于理解系统运行时对象间的关系。此外，通信图基本上覆盖了对象图的功能，所以，在实际系统建模过程中，很少单独使用对象图为系统建模。很多建模工具也不单独提供绘制对象图的功能（如 Rose 2003 中就没有对象图）。

2. 关联关系

对象是由特定的类生成的，对象之间的链接也需要类之间的关系来生成，这种关系就是关联。关联是类之间的一种结构化关系，是类之间的语义联系，表明类的对象之间存在着链接。可以说，对象是类的实例，而链接则是关联的实例。

1）识别关联

为了能够有效地识别类之间的关联关系，可以采用两类方法进行分析和抽取。

第一类方法是根据分析交互过程中所绘制的交互模型，来发现对象之间的链接，从而在相应的类上建立关联关系。正如第 5.4.4 节所提到的，对象之间为了进行消息传递，必须建立某种程度上的关系，这些关系在分析的初期都可以表示为对象间的链接，这些不同类对象之间的链接就构成了类之间的关联关系。当然，由于顺序图中并不能直接反映对象间的链接，可以将其转换为对应的通信图，从而更加清楚地描述对象间的链接。Rose 2003 提供了由顺序图自动创建通信图（UML 1.x 中的协作图）的功能，具体操作如下所示。在左边的资源浏览器中选择需要转换为通信图的顺序图，再选择 Browse|Create Collaboration Diagram 命令，即生成了与该顺序图同构的通信图，最后适当调整通信图中各对象的位置以保持图形的清晰。

图 5-57 是由 Rose 自动生成的图 5-35 所对应的通信图，从图中可以很清楚地发现对象之间所存在的链接。

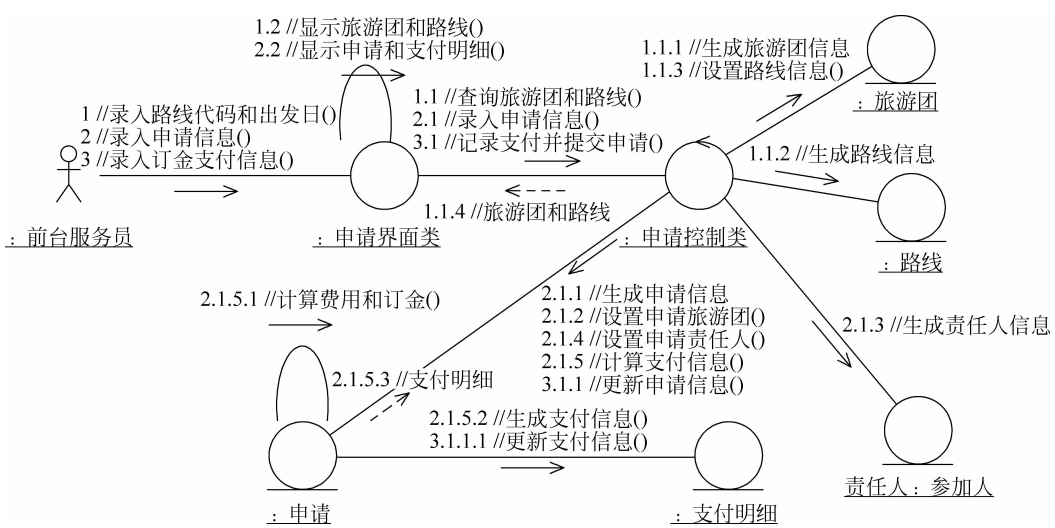

图 5-57 办理申请手续-用例实现基本场景通信图

从图 5-57 中可以看出由于申请界面对象要向申请控制对象发送消息 1.1、消息 2.1 和消息 3.1，因此这两个对象之间存在链接。而申请控制对象由于需要操作多个实体对象（如旅游团、路线、申请、参加人等），因此它们之间也存在链接。这就意味着相应的类之间就可以初步定义关联关系，图 5-46 所示用例实现的参与类类图中即体现了这些类之间的关联关系。

由于职责分配的过程主要按照 B-C-E 的原则进行，因此对象之间链接关系也主要是在边界类和控制类、控制类和实体类之间建立。这样通过交互模型方面可以很容易地定义边界类和控制类以及控制类和相应的实体类之间的关联关系。此外，同时也可以发现少量的实体类之间的关联关系，如图 5-57 中“申请”和“支付明细”之间的链接。事实上，这类关联关系在构造用例实现的参与类类图中就定义清楚了，图 5-46 中即体现了这类关联关系。

此外，还需要强调一点的是：对象之间的链接（对应类之间的关联关系）和职责分配过程中的消息传递是相互影响和制约的。有时候为了便于对象间的消息传递而建立对象间的链接，从而添加新的关联关系；而在进行职责分配时也要充分利用对象间现有的链接，而这些现有链接来自于类之间已定义的关联关系。在分析阶段，有很多实体类从业务上来说就存在语义联系，职责分配可以充分利用这些实体类之间的关系来传递消息。当然，这些实体类之间的关联关系有可能在业务建模阶段就已经定义，也有可能在分析的过程中逐步完善和定义出来；这就涉及了识别关联的第二类方法。

识别关联的第二类方法则是从系统自身的业务领域出发，分析领域中所存在的实体类之间的语义联系，为那些存在语义联系的类之间建立关联关系。按照经典的面向对象观点，关联关系是一种“has a”的关系，即两个关联的类 A、B 之间存在“A has a B”的含义。更具体地来说，A 和 B 之间可能的联系有：B 是 A 的一部分或成员（物理上或逻辑上）、B 是对 A 的描述、A 与 B 通信、A 使用或管理 B 等。这类方法是识别关联关系最原始的出发点，也是在业务建模阶段描述业务对象之间的关系、分析阶段描述实体类之间的关系最重要的手段。识别此类关联还有一个技巧是查找用例文档中的动词或动词短语；当该动词是用来连接两个作为类的名词时，这两个类之间也就可能存在关联关系，并可以以该动词作为关联关系的名称。

下面以旅游申请系统为例，来分析该系统中的实体类之间所存在的关联关系。

- 旅游团和路线：根据业务场景和相关用例文档的描述，每一个旅游团都是在已经规划好的路线上开设的；因此路线是对旅游团中相关信息的描述，它们之间也构成了一种关联关系。
- 申请和旅游团：每个旅游申请都需要指定并维护所申请的旅游团信息，它们之间显然也构成了一种关联关系。
- 申请和参加人：一个申请中存在若干个参加人，即申请需要维护其参加人信息；它们之间也构成了一种关联关系。
- 申请和支付明细：一个申请包括所需的费用、定金等支付信息，这种关系也构成了一种关联关系。
- 参加人和联系人：每个申请的参加人都需要指定其紧急情况下的联系人，这也构成了一种关联关系。

图 5-58 通过一副类图展示了通过上面的分析而得到的该系统实体类之间所存在的关

联关系。由于主要是展示类之间的关系，因此没有显示相关类的属性和操作以及构造型图标。

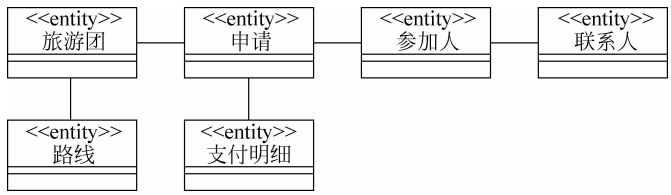

图 5-58　旅游申请系统实体类之间的关联关系

正如图 5-58 所看到的，默认的关联关系是没有任何箭头指示符的，这意味着该关系是双向可见的，即两个关联的类之间互相引用。但很多情况下，这种双向的关联是没有必要的。在“办理申请手续-用例实现”中，申请界面类需要访问申请控制类以发送界面消息，但申请控制类并不需要向申请界面类传递独立的消息（返回消息只是同步消息的返回结果，不是独立的消息调用）。因此，只需要建立申请界面类到申请控制类方向上的关联关系，即该关联关系是单方向的，通过在关联线的一端加上箭头来表明该关联的方向。图 5-46 中的 VOPC 类图中即给出了关联的方向。

事实上，关联的方向与交互图中消息传递的方向是一致的。当只需要单方向传递消息时，就可定义关联的方向；而当需要双方向传递消息时则采用默认的不带箭头的实线表示双方向的关联。因此，通过第一类方法识别出来的关联可以很容易地描述其方向；而通过第二类方法识别出来的关联就不太好明确其关系。不过，需要强调的一点是，分析阶段并不需要花太多的精力来分析关联的方向，只有在明确是单方向关联时才表示出来；否则就使用默认的双方向关联。

最后，有关识别关联还需要说明的是，分析阶段的重点在定义分析类并明确其职责。因此，并不需要花太多时间去深入地识别关联；要记住，在分析阶段“识别实体类比识别关联更重要”。太多的关联不仅不能有效地表示分析模型，反而会使分析模型变得混乱。有时，发现某些关联很费时，但带来的好处并不大。此外，还需要避免显示冗余或导出的关联，只表示那些分析模型中必须使用的关联即可。

2）定义关联名和端点名

关联可以有一个名称，用以描述该关系的含义。关联名一般采用动词或动词短语（即前面提到的连接两个作为“类”的名词的动词），用来连接两个作为名词的类，放置在关联的中央。

在“旅游申请系统”中，可以这样理解“申请”类和“旅游团”类之间的关联关系：申请人通过填写申请（表）来申请所要参加的旅游团信息。这就意味着“申请”类和“旅游团”类之间通过动词“申请”建立关联，而作为类的“申请”是一个名词（更准确的名字可以采用申请信息、申请表等名词来表示）。按照这种理解，该关联关系的名称即为“申请”，如图 5-59 所示。

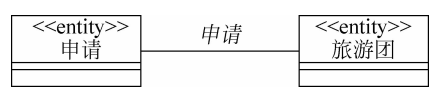

图 5-59　定义关联名

从图 5-59 中可以看出，通过关联名可以表达关联关系的含义，以帮助用户理解类之间的关系（当然，图中的关系本身比较好理解，关联名并非必要）。但关联名一般只能从一个方向上去描述这种类关系（可以通过在关联名的后面标注箭头来表明该名称的方向）。为此 UML 提供了另外一种更普遍的方式，可以从关联的两端分别描述该关系的作用，这就是角

色和端点名。

当一个类参与了某个关联时，它就在该关系中扮演了一个特定的角色。角色是关联中靠近它的一端的类在另外一个类中所呈现的面孔，或者说所发挥的作用。可以为一个类在关联中所扮演的角色进行命名，这就是关联的端点名(在 UML 1.x 中称为角色名)，端点名放置在关联线的一端。

虽然可以同时为关联关系定义关联名和端点名。但在明确给出了关联的端点名的情况下就不需要给出关联名，因为通过端点名完全可以反映关联的含义。有关使用关联名和端点名的场合，可以参考下面的一些规则。

(1) 如果用多个关联链接同一个类，则应该使用关联名或端点名来区分不同的关联(一般更倾向于使用端点名)。

(2) 如果一个关联有多于一个端点在同一个类上(即关联的两个端点同时附加在同一个类上，这种关联称为自反关联，参见本小节稍后的“4)自反关联和 n 元关联”部分)，则需要使用关联端点名来区分端点。

(3) 如果两个类之间只有一个关联，一般可以省略关联名；但有时为了使关联的作用更加清晰可以使用关联名。

一个极端的、必须使用关联名或端点名的例子是在两个类之间同时存在多个关联(属于第一种情况的特例，又称为多重关联)；此时，必须通过名字来区分这两个关联关系。考虑“旅游申请系统”中“申请”类和“参加人”类之间的关联关系，通过分析“管理参加人”的用例文档可以发现：参加人分为“申请责任人”和“其他参加人”两类，每个申请必须指定一位责任人并包含若干个参加人。显然，这两类参加人和申请之间的关联关系是不同的，通过这一个关联关系难以反映他们的不同。因此，此时可以为这两个类之间同时建立两个关联关系，如图 5-60 所示。

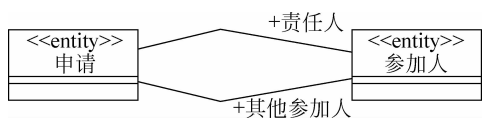

图 5-60 定义多重关联中的角色和端点名

从图 5-60 中可以看出，通过在参加人的一端定义了不同的端点名来表达参加人在这两个关联中所扮演的不同的角色，从而可以很容易地区分这两个关联关系。注意端点名前面的“+”号表明该角色是公有的，分析阶段并不需要去特别指定角色的可见性，有关该可见性的作用将在后面的设计中再进行讨论。

3) 定义多重性

关联表示了对象间的结构关系，然而一个类可以生成多个对象，这也意味着由一个关联可能生成若干个链接实例，或者说一个类的对象可能链接到所关联的类的多个对象上，这种“多少”即为关联角色的多重性，它表示一个整数的范围，通过多重性表达式来指名一组相关对象的可能个数。

多重性表达式用逗号分隔为多个区间，每个区间为“min..max”的形式，其中该区间的 min 为最小值，max 为最大值，即对象的个数可以取该从 min 到 max 的个数，表 5-8 列出了一些典型的多重性表达式。

表 5-8 典型的多重性表达式

多重性表达式	含　义
0..1	0 个或者 1 个
1	正好 1 个
0..*	0 个或者更多个(即没有上限限定)
*	0 个或者更多个(同 0..*)
1..*	1 个或者更多个
2..5	最少 2 个,最多 5 个(即 2 个～5 个)
1..3, 8, 10, 20..*	1 个～3 个,或者 8 个,或者 10 个,或者 20 到更多个

与端点名一样,多重性表达式也放在关联线的一端,表明另一端的一个对象可以与本方的多少个对象相链接。图 5-61 展示了在图 5-60 的基础上添加多重性表达式之后“申请”和“责任人”类之间的关联关系。

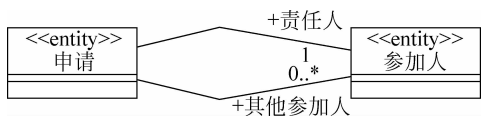

图 5-61 定义多重性

从图 5-61 中两个多重性表达式可以看出:对于一个申请来说,必须关联(指定)一个参加人作为其责任人(角色);而除了作为责任人的参加人之外,该申请可能没有其他参加人(0 个),也可能有很多(*,个数没有限制)其他参加人。

此外,如果没有显式地指定多重性,如图 5-61 中“申请”类一端,那么该多重性就是不确定的;一般来说,出现这种情况表明分析人员并不关心该多重性,其对后续的设计和实现没有什么影响。

4) 自反关联和 n 元关联

关联关系一般是建立在两个类之间的,这种关联称之为二元关联。但有时,也会在一个类或两个以上(三个或更多)的类之间建立关联。一个类自身之间的关联称为自反关联(又称递归关联);而三个或更多的类之间的关联则称之为 n 元关联。

自反关联是指一个类自身之间存在关联,它表明同一个类的不同对象之间存在链接[①]。考虑“旅游申请系统”中的“路线”类,在问题陈述中有关路线信息的维护提到:“变更后的线路作为新线路录入系统,同时留下变更历史,以记录这些路线的变化过程”。这意味着一个新的路线并不一定是全新设计的,可能来自原来某个老路线的调整。举个例子,2009 年 6 月,上海发生在建楼房倒塌事件的“楼脆脆”事件后,某旅行社推出了“华东五市＋乌镇、赠送上海倒塌楼房双飞六日游”的新旅游路线,显然该旅游路线是在原旅游路线的基础上进行适当调整后建立的新路线;而且在一段时间之后,随着人们对倒楼事件关注度的下降,该路线

① 注意是同一个类的不同对象,而不是同一个对象之间存在链接。要和通信图中的自反消息(是同一个对象自己给自己发消息,这个对象自身存在自反链接)区分开,如图 5-56 中“申请界面类”存在自反链接,但该类并不存在自反关联。因为对象自身是可以发消息给自己的,而不需要关联关系的支持;但是不同的对象就不能直接发消息,必须要有关系的支持。

可能又会重新调整。由此可见，不同的路线之间可能存在语义联系，也即存在关联关系。图 5-62 展示了路线类所存在的自反关联。

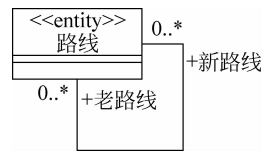

图 5-62 定义自反关联

从图 5-62 中可以看出，由于自反关联的两端都在同一个类上，因此至少要在一端定义端点名以明确关联的含义（参见本节“2)定义关联名和端点名”小节中定义端点名的第二种情况）。根据图 5-62 中的端点名和多重性表达式可以看出，对于一条路线来说，它可能不产生新路线，也可能产生多个新路线（新路线一端的多重性为 0..*）；同样一条路线可能来自一个或多个老路线，也可能没有与之关联的老路线（老路线一端的多重性也为 0..*）。

自反关联是一种应用广泛且非常有效的关联机制，在某些场合下应用自反关联能够实现更加灵活的系统结构。某企业的组织结构如图 5-63 所示。

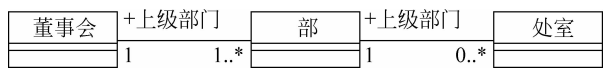

图 5-63 某企业组织结构的类图

从图 5-63 中可以看出，该企业内部是一个三层的组织结构：最顶层设立一个董事会，董事会下面按照不同的业务设置不同的部，而各个部下面可设若干个处室。当企业的组织结构没有变更时，这种关联的方案是有效的。但当该结构需要变更时，如设立直接由董事会负责的科室（即不属于任何部）、或者追加新的组织单位（如在处室下面在设立科级单位）等，该方案都无法实现，必须对整个结构进行调整。此时，可以将各个层次的类归纳为部门类，利用自反关联来实现，如图 5-64 所示。

从图 5-64 中可以看出，通过归纳出的部门类来表示企业内部的某个组织单位。对于一个部门来说可以有一个或者没有上级部门，也可以有若干个或者没有上级部门。这完全覆盖了图 5-63 所表达的含义，同时能够很好地适应组织结构的变更需求。

另外一种特殊的关联是在 3 个或者更多的类之间建立的 n 元关联；不过这种关联在实际项目中则很少使用，大多数 n 元关联都可以分解成带端点名和属性的二元关联，或者通过提取关联类将其转变为多个二元关联（常见编程语言都不支持这种 n 元关联的实现，因此到项目设计阶段都应该考虑转变为二元关联）。

当然也有一些 n 元关联很难在不丢失任何信息的情况下转化为二元关联。图 5-65 展示了一个典型的 n 元（三元）关联：程序员在项目中使用编程语言；一个程序员可能熟悉一门编程语言，并从事某个项目，但可能无法在此项目中使用这种语言。这样这个 n 元关联就很难分解成多个二元关联而不丢失任何信息。从图 5-65 中还可以看出，n 元关联通过一个菱形框将关联的各端联系起来。

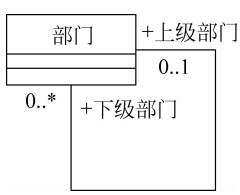

图 5-64 利用自反关联实现企业组织结构

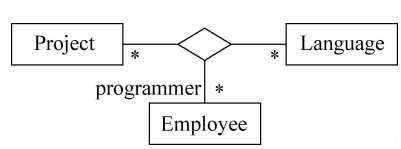

图 5-65 三元关联

5）识别关联类

关联关系代表了类之间的语义联系，而这种语义联系也可能存在一些属性信息，UML用关联类来表示这些信息。

关联类(Association Class)是一种被附加到关联关系上的类，用来描述该关联关系自身所拥有的一些属性和行为。当某些属于关联关系自身的特征信息无法被附加到关联两端的类时，就需要为该关联关系定义关联类。

考虑“旅游申请系统”中“参加人”和“联系人”之间所存在的关联关系，对于一个参加人而言，他需要在每次申请旅游团时指定与之存在关系的联系人。比如，张三在申请“北京3日游”旅游团时指定其父亲作为其联系人，那么他在填写申请表中的“与本人关系”一栏中就需要填写“父子关系”。而对于“父子关系”这一特征信息是用来描述该关联关系的，它不属于参加人“张三”对象，也不属于联系人“张三的父亲”对象，也就不可能存储在这两个对象中；因此，必须为该关联关系定义一个关联类来存储该特征，如图5-66所示。

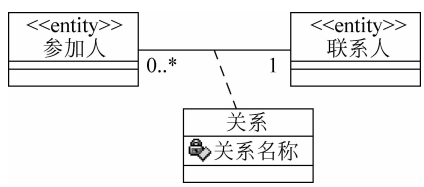

图5-66　定义关联类

从图5-66中可以看出，“关系”类是一个关联类，它通过一条虚线连接到与之相关的关联关系上，从而维护与该关联关系相关的特征信息。

3. 聚合关系

对于普通的关联关系而言，关联两端的类在该关系中是处于平等地位的。然而，在实际应用中，两个关联的类可能还存在一种整体和部分的含义：作为整体的对象包含部分对象。这种存在整体和部分含义的关联可以进一步表示成聚合关系。

聚合(Aggregation)关系是一种特殊的关联关系，除了拥有关联关系所有的基本特征之外，两个关联的类还分别代表“整体”和“部分”，意味着整体包含部分。对于聚合关系的识别，可以在已有的关联关系基础上，通过分析两个关联的类之间是否存在“A(整体)由B(部分)构成”、“B(部分)是A(整体)的一部分”等整体和部分的语义来完成。

考虑图5-58所描述的“旅游申请系统”中所存在的关联关系，对于“申请”和“支付明细”这两个关联的类来说，就存在着整体和部分的含义。支付明细作为申请的一部分，依附于某个申请，并构成了申请的一个基本要素；或者说每个申请都包含若干个支付明细信息。这样就可以把该关联关系进一步定义成聚合关系，如图5-67所示。

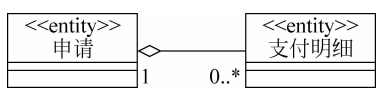

图5-67　定义聚合关系

从图5-67中可以看出，在聚合关系中，通过在整体(“申请”类)的一端加上空心的菱形

框来区分整体和部分(“支付明细”类)。此外,与普通的关联关系一样,聚合关系也可以定义名称、端点名、多重性以及使用自反聚合等。

4. 泛化关系

泛化(Generalization)是指类间的结构关系、亲子关系;子类继承父类所具有的所有的属性、操作和关系。其基本概念可参见第 1.4.3 节。

分析阶段的泛化关系主要来自于业务对象模型。针对实体类,结合业务领域的需求,从两个方面来提取泛化关系。

(1) 是否有类似的结构和行为的类,从而可以抽取出通用的结构(属性)和行为(操作)构成父类。

(2) 单个实体类是否存在一些不同类别的结构和行为,从而可以将这些不同类别的结构抽取出来构成不同的子类。

找出这些泛化关系后,可以通过类之间[is-a 关系]或者[kind-of 关系]是否成立来验证。具体来说,就是“子类是父类”,或“子类是父类的一种”。

考虑“旅游申请系统”已找出的实体类,可以发现对于“参加人”实体类而言,在实际旅游申请业务中,存在两类不同的参加人,即“大人”和“小孩”;他们在某些结构或行为上是不完全相同的,如旅游费用的计算、是否能作为责任人等。因此,可以将这些不同的结构和行为提取为不同的子类,从而构成父子类之间的泛化关系,如图 5-68 所示。

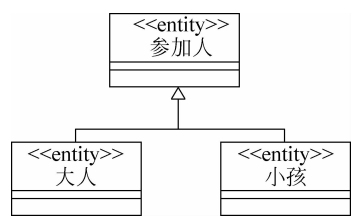

图 5-68 定义泛化关系

从图 5-68 中可以看出,在该泛化关系中,“参加人”作为父类用来描述其基本结构和行为;而“大人”和“小孩”作为子类可以用于描述它们所特有的结构和行为。

需要说明的是,由于泛化关系表达了类之间的亲子关系;因此,与关联不同,不需要再定义名称、角色、多重性等内容。

此外,泛化关系更深层次的目的是达到类间的可替换性,从而支持多态;这些内容主要是在设计阶段为提高系统设计质量而考虑的,分析阶段只需要从业务领域本身来考虑是否存在明确的亲子关系即可。

5.5.4 限定分析机制

在定义职责、属性和关系之后,分析类自身已基本定义完成。然而,当前的分析模型还缺少一部分内容,即对非功能需求的分析。分析类的定义主要来自前一阶段构造用例实现的成果,而用例实现主要关注系统的功能需求;而非功能需求也需要在分析模型中体现出来,这就是分析机制。正如第 5.3.2 节所描述的,在分析阶段并不对非功能需求进行深入分析,而是通过分析机制将其主要特性表述清楚即可,本节将为已经定义的分析类限定相应的分析机制。

表 5-3 列出了“旅游申请系统”中可能存在的分析机制,本节将把这些分析机制与前面所定义的分析类关联起来,从而将这些非功能需求分配给相应的类。表 5-9 列出了当前已经定义的部分分析类所存在的典型分析机制。

表 5-9 为分析类限定分析机制

分析类	主要的分析机制	说明
申请控制类	分布	前台服务员可以通过本地客户机访问服务器上的旅游团和路线信息
导出财务信息控制类	遗留接口	导出的财务信息需要导入到遗留的财务系统
旅游团、路线	持久性	旅游团和路线信息需存储在数据库中
申请	持久性	申请相关信息需存储在数据库中
支付明细	持久性、安全性、遗留接口	支付明细信息需存储在数据库中，并不允许随意修改，同时要与外部财务系统保持一致
参加人、联系人	持久性	参加人和联系人等信息需存储在数据库中

从中可以看出，大部分需要存储的实体类都通过持久性分析机制进行限定，而控制类则通过分布机制实现远程访问；有关于财务系统的接口则通过遗留接口分析机制表示；另外有关支付等敏感信息则需要通过安全性分析机制进行限定。

建立了分析类和分析机制的关联之后，下一步就需要进一步说明分析机制的特征。参照表 5-4 所给出的分析机制的特征为分析机制明确不同的特征值，这些特征值将为后续的设计提供重要的参考数据。当然，考虑到实际应用价值和工作效率问题，并不需要为每一个分析类逐一定义，只需要抓住主要的、反映关键性能指标的分析类进行定义即可。

以申请类的持久性分析机制为例，说明其分析机制的特征，这些特征值将为数据库设计中申请相关表的存储方案、索引设计等提供评价依据。表 5-10 给出了几个典型类的典型分析机制的特征值，这些特征值将约束后面的设计方案的建立。

表 5-10 申请类的持久性分析机制特征值

类	分析机制	特征	特征值
申请类	持久性	粒度	单个申请数据约 1～5KB
		容量	每天平均约 1000 个申请，高峰时约 5000 个
		访问频率	读取：每天 5000 次；写入：每天 1000 次 更新：每天 1000 次；删除：每天 1000 次
		访问机制	主要按申请编号查询，也可能按申请日期、申请人、旅游团编号等信息进行查询
		存储时间	永久保存，不删除；已完成的申请也需要保存历史信息
申请控制类	分布	分布机制	通过 HTTP 请求进行远程访问
		通信方式	以同步通信为主，部分复杂交易可能需要采用异步机制
		通信协议	相关参数没有明确的约束条件
支付明细	安全性	安全规则	使用何种安全访问规则
		授权策略	前台服务员录入支付信息，不能修改；只有特殊的授权用户可以修改
		数据粒度	以每一个支付项为单位
		用户粒度	按用户角色定义权限
导出财务信息控制类	遗留接口	响应时间	没有明确的时间约束
		持续时间	持续时间约 30 分钟
		访问机制	通过数据库 SQL 接口直接访问
		访问频率	每天晚上 1 次

5.5.5 统一分析类

至此，我们已经建立了一个基本完整的分析模型，有关分析类、其职责、属性以及它们所需要实现的分析机制和需要支持用例实现的相关协作的定义已全部完成。最后，还需要评估已经完成的工作，从而确保在开始架构设计时分析类的定义是完整的和一致的，这就是统一分析类的工作。

统一分析类的主要内容是评估已定义的分析类和用例实现，从而确保每个分析类表示一个单一的、明确定义的概念，并且不会出现职责重叠。要从系统全局角度确保创建了最小数量的分析类。通过统一分析类的过程，要达到以下两个目标。

- 验证分析类满足系统的功能需求。
- 验证分析类及其职责与它们支持的协作是一致的。

为此，我们可以通过一些检查点来评估分析类和相关的用例实现。可以从以下几个角度来评估分析类：

- 每个类的名字都清楚地反映了其所扮演的角色。
- 类表示了一个单一的、明确定义的抽象。
- 所有属性和职责在功能上是与类联系在一起的。
- 类提供了必要的行为支持用例实现。
- 类的所有需求都已经满足。
- 所有的属性和关系是必要的，并且用例实现需要它们的支持。

对于用例实现，可以从以下角度来进行评估：

- 所有的基本流、子流、备选流等都已被处理。
- 所有必要的对象都已被发现，并明确了所属的类。
- 所有行为都已被明确分配到参与对象。
- 存在多个交互图时，它们的关系是清晰的、一致的。

通过统一分析类，最终得出系统全部分析类的定义。此时，可以构造出反映系统全部分析类关系的完整的类图，该类图是对前面多个参与类类图的总结，也可作为分析模型最重要的交付成果。以“旅店预订系统”为例，图 5-69 给出了该系统在首次迭代时所要完成的两个用例实现所对应的参与类类图（图中省略了类的属性和操作）。

结合这两个参与类类图，可以构造该系统最终的分析类图，如图 5-70 所示。从中可以看出，对于参与多个用例实现的类来说，它们之间的关系分别从两个参与类类图中综合而来。

此外，从图 5-70 中也可以发现，对于“旅店预订系统”这么小规模的系统而言，其分析类图就包括 8 个类和若干个关系，存在一定的复杂性。而随着系统规模的增大，分析类图也变得更加复杂。此时，为了保持图形的清晰和有效，可以考虑在该分析类图中只显示那些重要的、全局范围内的类，而对于那些只参与单个用例实现的内部类，可以不展示在全局类图中。按照这种观点，为每个用例实现提取出来的边界类、控制类都可以不用展示在分析类图中（它们只与当前用例实现有关，与其他用例实现无关）；这样就可以构造出由系统核心实体类所构成的分析类图。图 5-71 展示了“旅游申请系统”实体类类图（为保持图形的清晰，图中有些类的属性和操作并没有完全显示出来，如参加人类的操作等），该图将前面小节所讲

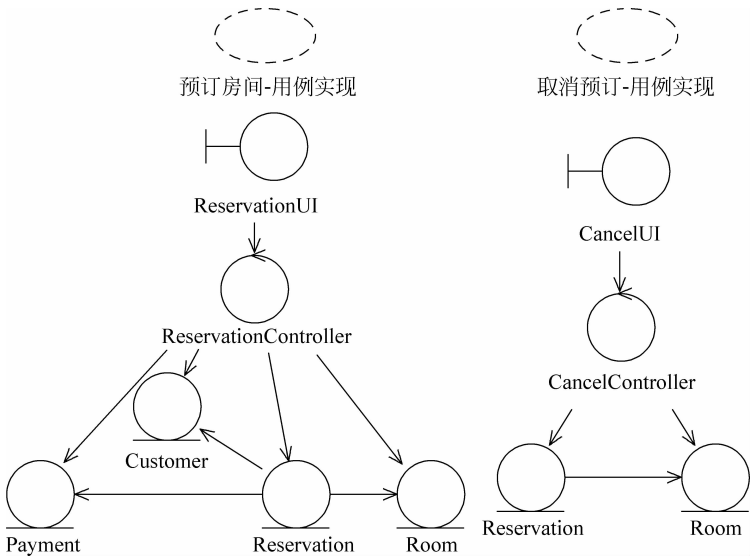

图 5-69　旅店预订系统的参与类类图

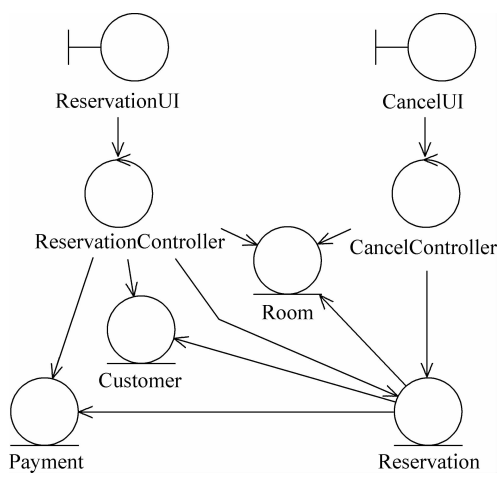

图 5-70　旅店预订系统分析类图

解的有关职责、属性和关系等的定义成果都展示出来了。

需要说明的是，事实上整个分析阶段的重点就在于找出体现系统核心业务所需数据的实体类，而界面和业务逻辑细节分别由边界类和控制类隐藏；因此图 5-71 所展示的实体类类图就可以很好地反映“旅游申请系统”的分析成果。在有些面向对象分析方法中，分析阶段的工作就是找到这些实体类，这些实体类即构成了系统概念模型这一最主要的分析成果。在实际分析过程中，以识别的初始实体类为依据，通过各个用例的 VOPC 图，删除那些没有引用的实体类，即可得到由实体类组成的分析类图，这就是分析的关键。

对于“旅游申请系统”最后还有一个问题需要处理，由于图 5-12 所提供的系统备选架构中，边界层有到控制层的依赖，这符合大部分分析类的依赖方向（参见各用例的 VOPC 类图）。然而，由于该系统的边界类中还包括一个系统接口类，即“财务系统接口类”。从该接

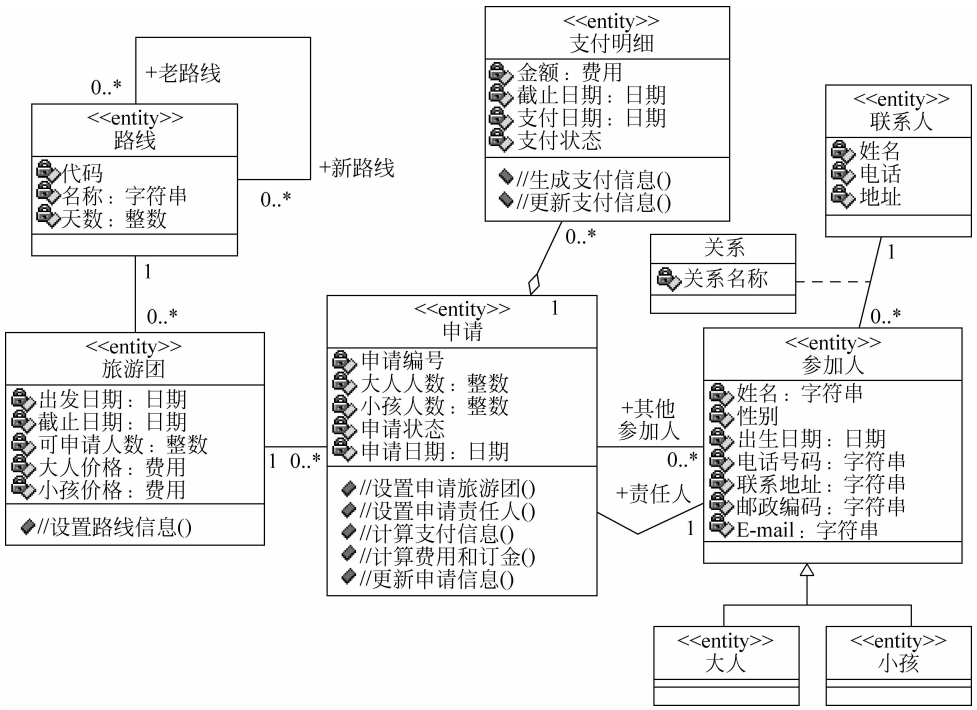

图 5-71 旅游申请系统实体类类图

口类所在的 VOPC 类图(见图 5-47)可以看出,此时“导出财务信息控制类”有到“财务系统接口类”的单向关联,这意味着该类所在的“控制层”也应该有到接口类所在的接口层的依赖关系,为此需要调整系统的备选架构,以体现类之间的关系。调整后的备选架构如图 5-72 所示,注意此时边界层和接口层存在双向的依赖关系,这意味着这两个包之间存在循环依赖,在设计中会进行单独处理。

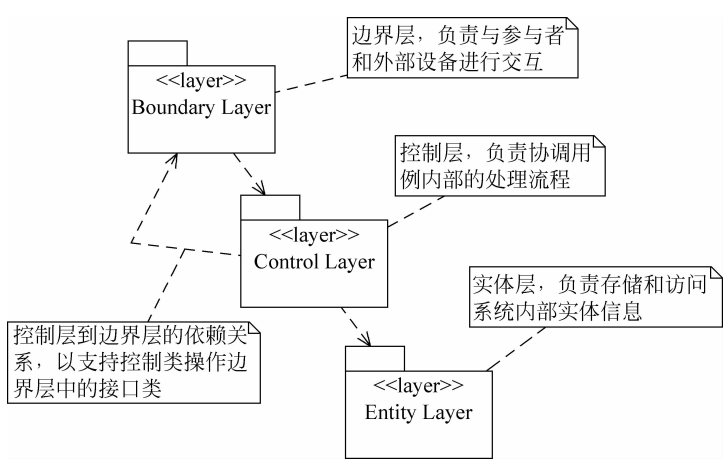

图 5-72 旅游申请系统调整后的备选架构