第 8 章 完善需求模型

学习目标

- 在软件开发中重用的重要意义
- 面向对象原则如何实现重用
- 如何确认并且建模一般化和组合
- 建模可重用组件的一种方法
- 软件开发中“模式”一词的含义
- 分析模式如何帮助构建模型

8.1 介绍

完善并且进一步为分析模型的结构添加内容，这么做的目标是创建可重用的条件。这可能意味着重用现有软件或者重用最初为早期版本的系统编写的软件。另外，为当前开发项目创建的新规范或软件，对未来其他系统可能也会有用。不管什么情况，模型必须按照如下方式构建，使用这种方式可以轻而易举地确认重用。虽然重用的实际优势更有可能在设计和构造工作流的过程中获得，但在分析的时候就考虑重用也是很重要的，因为分析需要创建逻辑规范。面向对象分析为重用提供了三种主要机制：

- 一般化、组合、封装和信息隐藏的基本抽象机制。
- 可重用软件组件的规范。
- 分析模式的应用。

在面向对象中，抽象的关键作用在第 4 章已经介绍。在这里我们会考虑如何对第 7 章介绍的分析类图引入一般化和组合结构。

基于组件的开发是根据组合结构规范进行的，而组合结构规范可以作为可重用软件组件使用。有效的信息隐藏需要将对象或结构的内部情况隐藏于界面之后。组件则更进一步，它们被设计为相互独立的构件。在本章，我们将介绍建模组件的 UML 标记法，及其内部细节的独立接口。组件也在面向服务的架构中扮演着重要的角色(在第 20 章我们会讨论 Web 服务，这是一种常见的面向服务架构的实现)

自 20 世纪 80 年代以来，“模式”运动已经为获取和交流有关一般化的知识提供了途径，特别是在分析和设计活动中。在本章，我们介绍模式的概念，说明如何应用分析模式(稍后，分别在第 12 章和第 15 章，我们会讨论架构和设计模式)。

软件和规范的重用需要仔细管理，因为它们的应用涉及整个生命周期，并且不会仅限于单个工作流。我们在后续章节将进一步讨论这一点(特别是在第 20 章)。

8.2 软件和规范重用

近几年来，软件开发领域已经经历了一次小的革命。很多人主要从事先存在的组件来关注系统的集成，而不是基于整个新的软件系统的开发来关注。面向对象开发总是旨在最小化新的规范、设计和编程的工作量，而这些必须在新系统构建的时候就已经完成。理论上，面向对象方法通过抽象和信息隐藏建立完好的原则，使得这一想法成为可能，然而很多年来，大型的重用被证明是难以捉摸的。现在，重用至少正在成为现实。这部分归因于上述熟知的面向对象原则，也归因于从模式运动中涌现出的新思想，以及基于组件的开发和面向服务架构。

8.2.1 为何重用

一般来说，从头开始生成任何已经在其他地方生成的产品以满足标准，是一种对时间和精力的浪费。如果需要为房间安装新的电灯，那么不必重新发明电灯并且自己装配电灯。即便拥有此类知识、技能以及设备，成本也会令人望而却步。这可以应用于其他生产领域和软件开发中。好的专业人员已经从已有的经历中和其他同事那里知之甚多。程序员已经组建了各种库，从个人收集到的有用子例程，到商用的已发布产品(包括为数众多的业界标准组件)。后面的例子包含在 Microsoft Examples 中使用的 DLL(动态链接库)文件，以及对 Java 程序员可以使用的类库。设计者已经组建了由设计、模板、模式和框架片段组成的对应的库。在大部分专业情况下，重复前人已经完成的工作意义不大。

8.2.2 难以重用的原因

大部分作者都认为，软件重用的潜在优势直到最近也没有在实践中得到完全发挥。为何如此呢？

重用并不总是合适的

重用具有好处这一通用规则也有例外。例如，学生经常会被要求(似乎他们是首个解决该问题的人)解决之前已经完全被其他人解决的问题。这会收到好的教育效果，因为一般情况下，该过程有利于加快理解，而不仅仅是解决问题。这就是教育人士对抄袭嫉恨如仇的原因之一：学生将别人的工作作为自己的成果，但从这一过程中不会学到任何东西。

无须先入为主的了解，也可以开始新的项目。例如，最初我们对新系统的需求一无所知。分析师应该考虑所提交系统的独有特性及其环境。因此，我们应该尽可能地知道详细

情况，开始调查新的情况。但是假定我们对碰到的任何问题如何解决都一无所知，这无疑是痴心妄想。不管在什么时候，如果过去成功的工作对当前问题的解决相关，我们就应该加以利用。

“此处发明不合适”综合症

一些软件开发者(甚至偶尔会出现整个部门的情况)可能会忽略职业经历中积累下来的智慧。为什么呢？一个原因就是 NIH(“此处发明不合适”)综合症，据说是程序员的主要病症。NIH 描述了一些人的态度，他们认为：“我不会信任任何人的部件——即便看起来有用，符合我的目的并且能够承受——不管怎样，我希望发明自己的部件。”这对于希望享受技术挑战的人，或者有原因不信任其他人的人来说，是可以理解的，但是通常从商业角度来说，这很不明智。

重用难以管理

重用成功的一个关键是过程的管理。开发者希望找到可以重用的工件(模型、模板、程序子例程、整个程序等)，这需要进行分类。分类必须是全面的，到目前为止，以如下方式组织，即易于找到满足用户需求的工件。工件自身必须被设计为可重用的，这通常使得构建更为复杂，成本更高。但是在设计中过于细化的任何工件，在重用到其他上下文中时，都必须仔细地适应上下文。适应性比创建新的精确符合上下文的工件更有难度。良好地管理这一过程的困难有时意味着在实践中重用很难达到目标，尽管少有人质疑重用的优点。

分析工作比设计和软件更难以重用

就简单性而言，软件重用非常容易达到。例如，使用程序中的库函数，可能需要导入库，通过名称调用函数，为函数传递需要的参数。您的程序然后可以使用上述结果。对于程序员来说，需要的只是一份库的副本，对函数签名的知识以及对函数功能的一些了解。这些事情通常会在任何编程课程的早期进行学习。

设计工件的重用相对来说也比较直接。在用户界面设计中，一个熟悉的例子是给予用户灵活性受限的模板，据此他们可以在社交网站上自制主页面。每一个模板在本质上都是由不同用户重用了很多次的设计。

然而，分析模型仍旧是一种高度抽象。因此这是开发领域最难重用的一部分，因为本质上很复杂，只有部分模型能够重用。还需要组织模型，以便能够抽象出(隐藏)某一项需求的特征，这些特征对于其他项目中类似的需求来说，不需要对比验证。接下来，重用的整个关键是节省工作量，因此也应该不需要为了作出比较而开发新需求的完整模型。最后，两种需求之间的相关不同之处的任何对比，应该是清晰可见的——因此也应该不需要为了看到这些特征而开发新需求的完整模型。稍后我们会看到使用模式克服这些困难的一种方式。

8.2.3　面向对象对重用的贡献

面向对象软件开发有赖于两种主要的有助于重用的抽象形式：第一种是一般化，第二种是组合了信息隐藏的封装。毋庸置疑，它们的使用类似于很多其他行业中的实践。

一般化

一般化是一种抽象(已在第 4 章讲述)形式，关注设计或与多个场景相关的规范的一些方面，而忽略只依赖于某个特定场景的那些方面。通常可以确认设计的某种元素，或者确认问题的某种解决方案，它们可以应用于一系列的场景或问题中。车轮就是典型的示例。车轮几乎可以是任何大小，并且采用很多不同的设计和材料，这取决于用途。鼠标中的小塑料滚珠定义了滚珠在鼠标垫上的移动。自行车的轮胎式车轮沿着轮胎轨迹起到推动和引导的作用，为骑车人提供一些缓冲摩擦。旧蒸汽机引擎上的巨大铁滑轮可以平衡活塞运动，尽管它们的大小、材料以及制造的方法各不相同，但所有的轮子都具有一些共同特点，即都是圆形，并且围绕中心点滚动。

设计鼠标滚轮的工程师必须考虑特定的场景，例如需要它们绝缘、小型轻便，并且制作廉价。但是沿轴滚动的基本原理不需要过多考虑，因为对于此类工程问题有很好的解决方案。滚轮特定的大小、重量等对于应用来说是特别的。但是圆形是从所有轮子中抽象出来的通用原则，可以反复用于设计诸多其他类型的轮子。

软件中的一般化在很多方面都与此相似。目标是确认很可能在系统中行之有效的规范或设计的特征，或者专门用于不是特定开发的规范和设计。在第 4 章，我们看到了抽象的超类 Employee，是如何将不同种类的实际雇员(按小时、周和月支付的雇员)的描述中具备的共同特征抽象出来的。接下来使用 Agate 案例研究中的例子进一步介绍如何找到一般化。

封装和信息隐藏

封装和信息隐藏一同表示(如第 4 章所述)某种类型的抽象，这种类型的抽象关注事物的外部行为，而忽略如何产生这些行为的内部细节。这对于成功运用模块化操作是必需的。现代台式计算机的组装提供了很多例子。例如，PC 鼠标可以使用很多不同的技术，但是从用户的角度看，行为方式都很类似。在笔者的办公桌上，使用了三种不同的鼠标：两个是有线鼠标，其中之一是滚轮鼠标，另一个是光电鼠标，而第三个鼠标则是无线鼠标。可以说明它们之间的不同，因为会有一些外部暗示(是否是有线的，以及是否是光电的)。但是大部分时候，不管鼠标是滚轮的还是光电的，不管是有线的还是无线的，使用差别不大。更重要的是，模块化允许一个模块由另一个模块替换，而无需严格匹配。如果有线滚轮鼠标出了问题，可以由无线光电鼠标代替。不管鼠标自身是如何工作的，在鼠标和与其交互的计算机之间有标准的接口。接口是根据插口类型和管脚连接(每一个管脚承载的信号类型以及额定电压)定义的。软件模块化旨在进行同样效果的替代，而软件接口也需要据此以标准的方式定义。软件接口的定义通常是根据提供的服务以及调用每一个服务的消息签名来确定的。

组合涉及一组类的封装，这些类在整体上具有成为可重用子组件的能力：换言之，它们是单独的模块。这基于如下思想：复杂的整体由更为简单的组件组成。这些没有整体上那么复杂的组件，自身可能也是由更为简单的子组件(初级组件或者二者的混合)组成的。接下来考虑 Agate 案例研究中的示例，讨论组合的使用。

8.3　进一步向结构中添加内容

8.3.1　寻找和建模一般化

图 8-1 显示了在 Agate 案例研究中由分析师进行的访谈记录。她的主要目标是更多地了解职员的不同类型。在匆忙的工作中，分析师只收集到少量的资料，但是这些资料强调了一些有用的信息，必须合适地进行建模：

3月17日——与Amarjeet Grewal(财务总监)的简单访谈
目的——澄清上周四访谈的一些内容

询问职员类型
——与系统相关的看起来只有两种类型
创意人员(C)和行政人员(A)
他们有何不同之处？
——主要的不同是奖金支付……
1. (C)根据广告团队收益计算奖金(只是他们所工作的那些广告团队)
2. (A)根据所有广告团队收益的平均值计算
还有其他不同之处吗？Amarjeet认为——
——C的资格需要被记录
——C可以被分配作为客户的联络员
——A不能分配给特定的广告团队
没有其他明显的差异。
(注意——在下一次访谈中，获得两种奖金计算算法的细节)

图 8-1　分析师关于 Agate 职员类型差异的记录

- 有两种类型的职员。
- 奖金分别计算。
- 不同类型职员的数据应该分别记录。

图 8-2 显示了对应这些信息的部分类图(为了简单起见，只显示相关的类)。

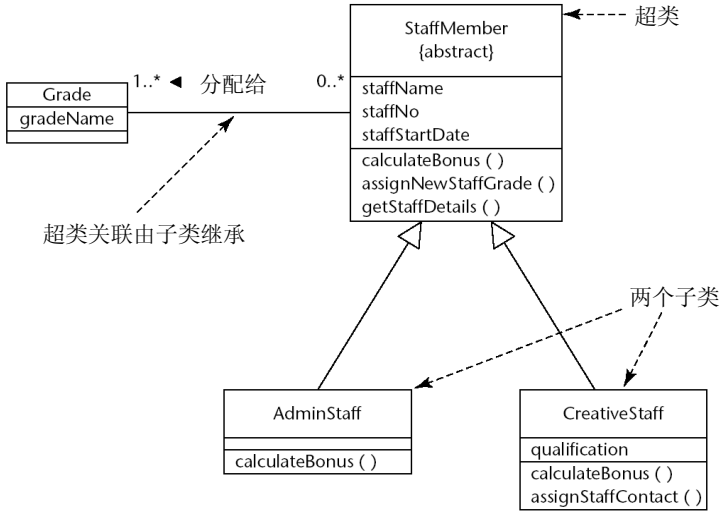

图 8-2　Agate 职员角色的一般化层级结构

重定义的操作

为何在图 8-2 中，层级结构中的所有三个类中都有 calculateBonus()操作？是分析师出错了吗？还是她没能有效利用一般化标记法提供的间接优势？

原因是：在子类中，超类需要被重写(第 4 章介绍了继承特征的重写)。当 AdminStaff 和 CreativeStaff 都需要操作 calculateBonus()的时候，在不同的情况下，工作方式也不同。因为用于计算的精确逻辑在这两组职员之间有所不同，这两类操作需要在设计各自算法，以及编写实现代码的时候区别对待。这决定了在两种看起来独立的子类中，会出现表面相似的操作。

那么为何要在超类中包含操作呢？答案是为了适应未来的需要。超类可能在稍后会包含目前尚未知晓的其他子类。这里，分析师已经知道——或者假定知道——属于 StaffMember 所有子类的对象很可能需要操作去计算奖金。因此，至少该操作的“骨架”会在超类中包含。这至少包含该操作的签名，但是因为接口是其他类所能了解到的全部信息，所以在超类的定义中将接口列入。即便超类操作被完整定义，一些子类的程序员也可能不会选择使用，因为他们对于操作的逻辑版本要求会不一样。

抽象类和具体类

StaffMember 是抽象的，因为没有包含任何实例。这意味着在 Agate 中不存在职员的“一般”成员，也不存在特定小组中的成员，这是由图 8-2 中类名下面的{abstract}标识标记的(另一种标记法是将类的名称书写为斜体)。在一般化层级结构中，只有超类才是抽象的。其他所有的类都会有一个或多个实例，它们被称为具体类或实例化的类。到目前为止，见到的所有职员(即在相关模型中出现的职员)都定义为 AdminStaff 或 CreativeStaff。如果我们稍后发现其他组中职员的行为、数据或关联有区别，并且如果我们需要建模新组，那么可以在图中使用新的子类表示。超类的关键点是：超类处于比子类更高一层的抽象中。这一通用性允许广告类在其他子系统的使用中被采用。虽然：超类自身对于宣称作为抽象类并不充分，通常的情况却是如此，并且此时我们可以忽略该规则的例外情况。

一般化如何有助于达到重用

创建一般化层级结构的原因是：为了能够使子类的规范在其他上下文中被重用。通常，这一重用是在当前的应用程序中进行的。

想象 Agate 系统已经完成，并且付诸使用。在之后的某一时间，总监重新组织了公司，结果是财务经理会被支付与广告特定利润相关的奖金。奖金的计算与行政经理和其他创意职员的奖金计算方式不同。例如，会包含来自财务经理所监管广告团队的一部分，以及来自公司一般盈利的一部分。很容易通过添加其他的子类来符合这一新的行为，如图 8-3 所示。

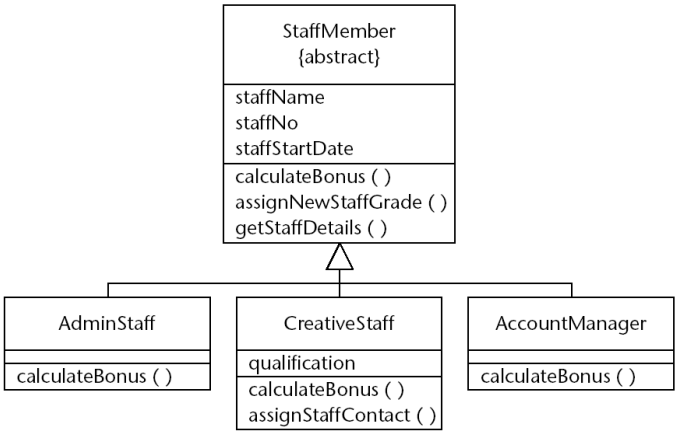

图 8-3　新的子类很容易添加

还应该注意的是替代的标记法风格。在图 8-2 中，每一个子类都通过自己的一般化关联加入超类；而在图 8-3 中，三个子类通过树状结构组织，与子类有三角形的连接点。这一树状结构被称为标记法的共享目标形式。这两种形式都是可以接受的，但是共享目标形式的标记法只会在一般化关系从属于同一般化集合的时候才会使用。在图 8-3 所示的示例中，使用的标记法是合适的，因为显示的雇员类型表示超类的连贯划分方式。然而，假定因为某种原因，我们还需要将雇员特殊化为 Male 和 Female。这一新的一般化关系从属于不同的一般化集合。一般化集合的名称可以选择性地在图中紧接着关系的地方显示，如图 8-4 所示。

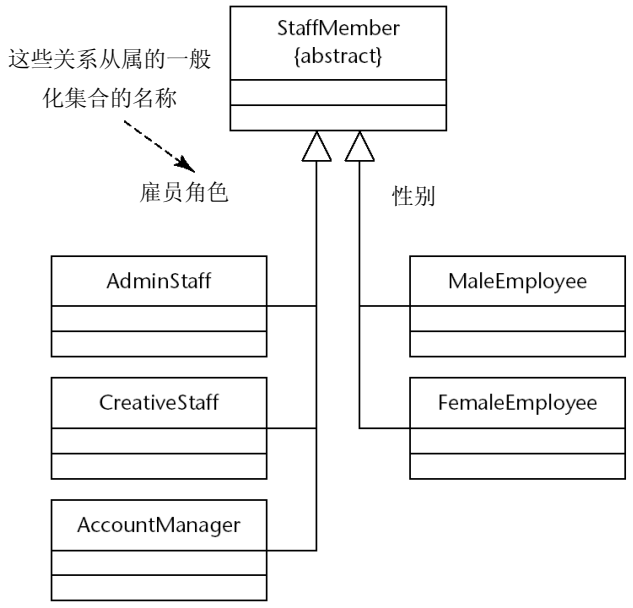

图 8-4　一般化按照不同的方式划分超类

添加新的 AccountManager 子类，对类模型的其余部分影响甚微或者毫无影响。重用是

从现有抽象类 StaffMember 中的 AccountManager 派生出来的。新的类继承了不是自身特殊化一部分的所有特性和操作——在此情况下，大约是规范的 85%。这只是相对于类来说的；对于大型系统的很多类来说，设计和编码工作量的节省是非常显著的。但是会有这种机会的，因为我们之前已经确定了职员的一般方面。这是一般化的主要好处，一般化的层级结构通常可以扩展，而无须对现有结构进行明显的改动。

一般化也能为其他应用程序中的重用提供可能。例如，Agate 应用程序的开发者会发现，抽象类 StaffMember 已经存在于之前项目记录的一般化层级结构的某个类别中。因此不需要再次归档，分析可以关注当前应用程序特有的特征。当然，如果该超类像图 8-3 中所示的超类那么简单，优势就不是那么明显。但是在实际项目中，类的层级结构有时很复杂，继承的类的定义除了包含特性和操作之外，还会伴随关联(结构很复杂)。

使用自顶向下的方法查找一般化

在已经确定超类和子类的情况下，发现一般化相对比较容易。如果关联可以由“是一种”这样的表述进行描述，那么通常可以使用一般化进行建模。有时也会疑惑，是否所有的关联都是如此。例如，“行政经理是一种经理”。同样，“直升机是一种飞机，也是一种固定翼喷气式飞机”，“卡车是一种汽车，也是一种货车”，它们都暗示了相似结构的一般化。

找出多层一般化也并非易事。这只是说明，某个关系中的超类可能是另一个关系中的子类。例如，Aircraft 同时是 Helicopter 和 Vehicle 的超类。实际中，很多类的模型具有超过 4 层或 5 层的一般化结构，但这主要缘于设计原因。

使用自底向上的方法查找一般化

另一种方法是查找模型中类的相似性，以及考虑模型是否通过引入抽象相似性的超类，这需要细心。目的是为了增加模型抽象的层面，但是引入的进一步的任何抽象都应该是“有用的”。指导原则仍然是：任何新的一般化必须满足第 4 章描述的所有测试。

何时不使用一般化

一般化会被过度使用，因此需要作出一些判断，从而决定在每一个场景下最有用的一般化。例如在 Agate 中，职员和(一些)客户是人员(考虑到阐述的缘故，我们会忽略实际上大部分的客户是公司而不是个人这一事实)。经验不足的分析师可能觉得这就能确定 Person 超类的创建，以包含 Client 和 StaffMember 的任何特性和操作。但是很快会表现为：新的类定义包含的内容虽然很少，但是却包含了特性 personName。这实际上是强制通过一般化的层级结构来包含差异过大的子类。

其次，我们不应该臆断尚未被当前需求判别的超类。例如在 Agate 中，AdminStaff 和 CreativeStaff 是不同的类，它们的特性和操作各不相同。我们也知道组织中其他类型的职员，例如总监。但是我们不应该自动创建 StaffMember 的其他被称为 Director-Staff 的子类。即便是在总监与系统有一些关联的情况下，也没有原因支持它们成为单独的类。现有的类(例如 AdminStaff)就足够对之建模，除非我们找到了行为或信息结构在某一方面不一样，才可以将之作为单独的类。这里会有一些矛盾。一方面，一般化会被建模，以便在分析师不能合理预测的时候，允许创建未知的子类。这一可能性利用了构建一般化层级结构的主

要优势。另一方面，如果一般化被过度使用，就会增加模型的复杂度而获益甚少。除了使用符合经验的判断，以及由组织制定的指导原则外，对于该问题没有简单的答案。

多重继承

在第 4 章介绍了多重继承。多重继承通常适用于类从多个超类中继承的情况，这与日常生活中的分类相似。例如，如果我们根据用途将家用电器分类，那么咖啡杯属于容器；如果我们根据价值以及美观分类，那么咖啡杯可能属于“日用品”而非“最佳品”；如果我们按照健康标准分类，那么咖啡杯属于危险品(因为易碎)。咖啡杯在同一时间可以根据不同的分类标准属于各种类别，而没有任何逻辑上的冲突。

在面向对象建模中，特别是在设计期间，定义从多个超类继承特性的类是有用的。在每种情况下，所有的特性都继承自超类。

8.3.2　寻找和建模组合

组合(或组合聚集)基于聚集这一概念，聚集是很多面向对象编程语言的特征。最简单的情况是，聚集表示类之间的整体一部分关系，而组合显示了整体中各个部分之间更强形式的归属权。

组合的一种应用对于使用通用计算机绘图软件包的任何用户来说都很熟悉。例如，本书中的很多绘图都是使用众所周知的绘图软件包准备并编辑的。该应用允许用户选择并且分组多个对象。分组对象的行为与单个对象一样，可以使用单个命令进行放大和缩小、旋转、复制、移动或删除。图 8-5 阐述了这一点，而图 8-6 将组合建模为单个类图。注意组合结构是嵌套的——组合自身可以包含进一步的组合。以同样方式组合的绘制对象，只能作为单个绘图组件被处理，组合结构中的部分对象不能被直接访问，而整个对象对系统的其他部分显示单个接口。

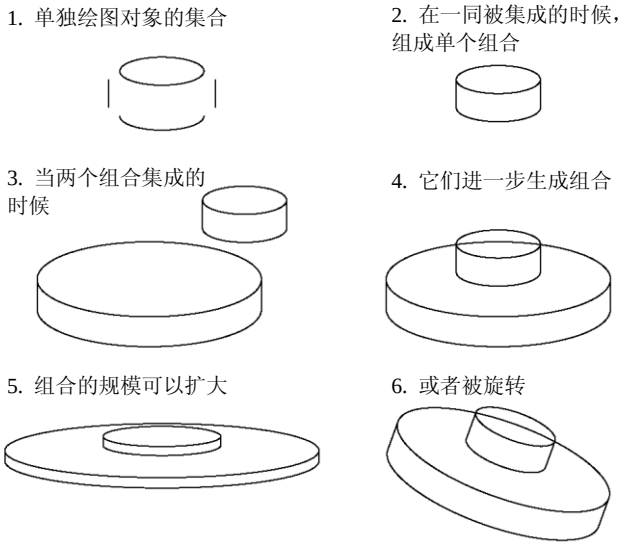

图 8-5　绘图包中对象的组合

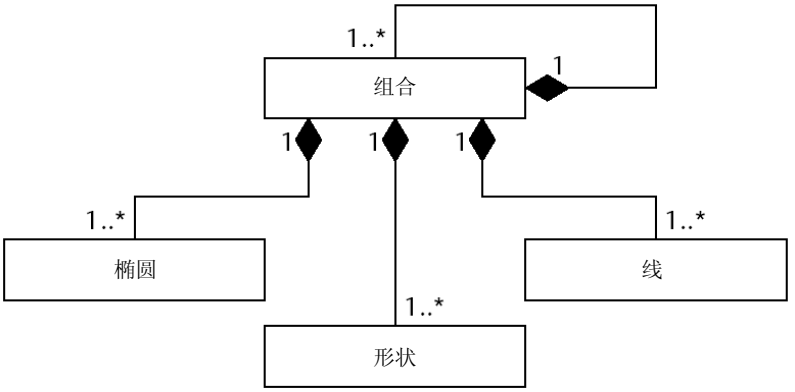

图 8-6　使用 UML 类表示的相同组合结构

组合和聚集都会在需求期间确认，但是它们主要应用于设计和实现活动期间，此时它们可以用于将对象的结构封装为潜在可重用的子组件。这不仅仅是使用单个名称标记结构的问题。封装应该是内聚的，内聚的模块更为重要。组合的外部接口作为关联的“整体”端，是单个对象的接口。组合的内部接口的细节——组合包含的其他对象，以及所代表的一些职责——对于客户不可见。

该标记法类似于简单的关联，但是在“整体”端会有菱形。实心菱形表示组合，虚心菱形表示聚集。组合结构的另一种标记法如图 8-7 所示，这明确显示了作为部件容器的组合对象。

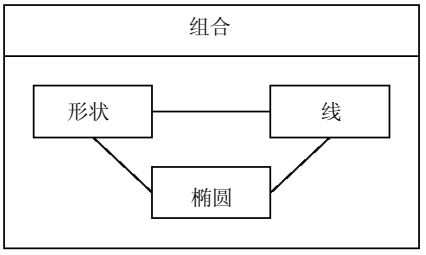

图 8-7　组合结构的另一种标记法

在面向业务的应用程序中使用组合和聚集，比在绘图软件包中使用存在更多的问题，但是仍旧值得对之进行建模，因为这表达了业务域结构的有用信息。接下来使用 Agate 案例的一个示例阐述如何确认聚集。

8.3.3　组合或聚集与一般化的合并

Agate 案例研究至少提供了一个场景来对一般化和组合的合并进行建模。这在 A1.3 节的语句“广告可以有多种类型”中说明。对于每一种类型来说，例如对于“报纸广告是一种广告”来说，该语句是正确的。为了简单起见，我们假定“广告”指代设计，而不是插入——因此，在一份报纸中出现 5 次的广告就说明同一广告出现了 5 次，而不是 5 个广告出现一次。这表明广告可以是超类，而报纸广告则是子类。这符合 4.2.4 节描述的测试吗？

要得到权威的答案，就需要详细检查每一个类的特性和操作，但是答案似乎应该是肯定的。

广告也包括多个部分。每一种类型广告的精确组合是互不相同的，因此，关联的结构不能在超类的层面定义(电视广告的特性、操作和组合结构可能在一些方面类似于报纸广告，但是会在其他方面与之不同)。

我们可以看到，在 Campaign 和 Advert 之间的关联中的可能组合，以及 Advert 及其关联部分的可能组合。广告团队包含了一个或多个广告。报纸广告包括文字副本、图片和照片。

这真的是组合而非聚集吗？首先，每一个广告都从属于多个广告团队吗？这在 Agate 案例研究中毋庸置疑，但是看起来一个广告不可能同时属于多个广告团队。其次，每一个 Advert 都有与对应 Campaign 相同的生命线吗？同样，该判断并不明确，但是客户或许希望在其他广告团队中再次使用昂贵的广告。需要澄清这一点，但是同时不能将之简单建模为组合。再次，副本、图片和照片能够属于多个报纸广告吗？或许图片和照片可以被重用，但是文字副本则不能。最后，每一个组件的生命线与广告的一致吗？或许有些一致，有些不一致。这都需要进一步澄清，但同时只有在 NewspaperAdvertCopy 的情况下才能确定组合，而聚集会在其他地方确认。图 8-8 显示了根据上述分析生成的部分类模型。

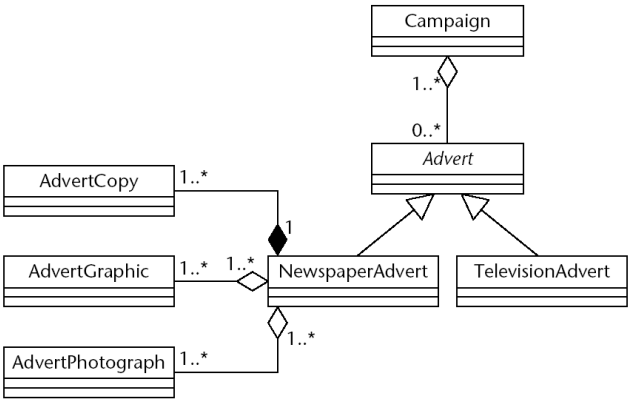

图 8-8　Agate 模型中的聚集、组合和一般化

8.3.4　组织分析模型——包和依赖关系

分析模型可能包括完整的独立组件，我们会在下面讨论。即便没有包含，也必须以这种方式组织，以便在面临新需求的时候能够保持健壮性。这需要分析师的技能和判断。该任务的其中一部分是定义分析软件包，而分析软件包相对独立，在内部却高度内聚(在第 14 章详细讨论内聚)。我们在第 7 章看到包(在第 5 章介绍)的含义是，开发者可以“分解出”较项目自身应用更为广泛的类或结构。但是当模型被划分为包的时候(或者预先存在的组件用于支持当前项目的时候)，跟踪不同类和包之间的依赖关系是很重要的。

我们在“案例 A2 中看到：Agate 案例研究表明三个相互关联但是又有区分的应用程序区域：广告准备、广告团队管理和职员管理。将之建模为单独的软件包可能是有意义的，但是一些实体对象会被多个包使用。根据我们目前为止所做的分析，StaffMember 在 Campaign Management 和 Staff Management 包中扮演角色。这会产生架构决策。图 8-9 阐述了下面描

述的一些变种。

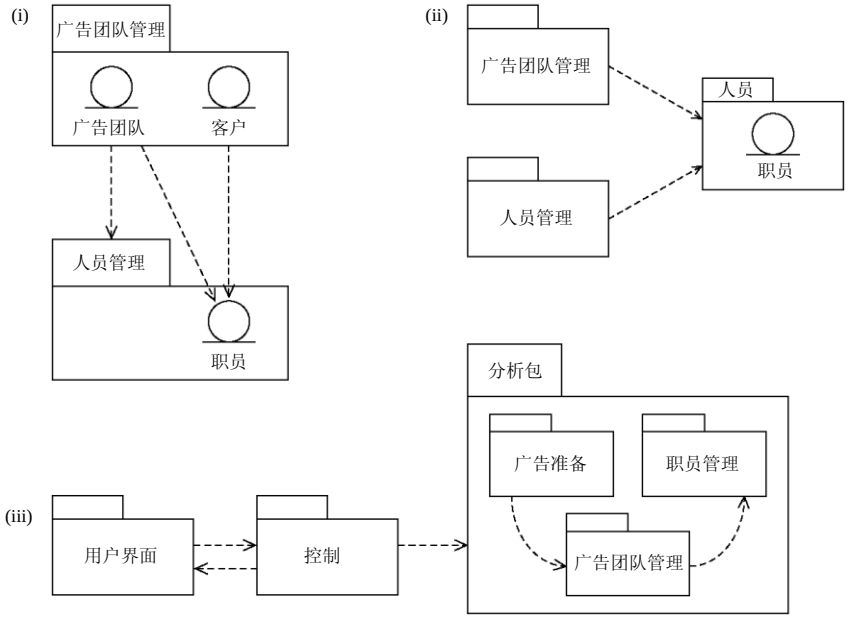

图 8-9　包和对象之间的另一种依赖关系

- 可以在 Staff Management 软件包中放置 StaffMember。在这种情况下，需要建模从 Campaign Management 到 Staff Management 的依赖关系，因为 Client 和 Campaign 类需要与 StaffMember 进行关联。
- 可以将 StaffMember 移动到单独的软件包中。如果还有更广泛的用途，StaffMember 会被判断使用。例如，也会有工资、人员、福利、养老金应用程序需要这个类。在这种情况下，需要建模从对应的所有软件包到包含 StaffMember 的软件包的依赖关系。
- 进一步的分析会揭示，实际上 StaffMember 不是单独的类。在这种情况下，派生类会保留在各自的包中，但是它们之间很可能会有关联，仍必须归档依赖关系。
- 另外在第 7 章，我们决定将所有边界对象划分到 User Interface 包中，将所有控制对象划分到 Control 包中。在这些特殊的包中，对象理所当然与其他包中的对象存在依赖关系。

对包的依赖关系进行归档，这么做的重要性或许没有像在这里处理相对简单的模型那样明显。但是在模型规模更大、更为复杂的时候会更加重要，综上所述，当涉及潜在的重用元素时，不管是在单个类级别上发生，还是在组件级别上发生，都会更为复杂。

8.4　重用软件组件

软件组件是为其他组件或系统提供服务的包或模块。可重用的组件是指那些被设计用来在多个场合使用的组件。一方面，组件是组合结构的一种特殊情况，如上所述。实际上，在一些情况下，单个类可以是可重用的组件。然而，这一术语现在一般被用于相对复杂的

结构，即互相独立操作的结构。通常在不同的组织中，在不同的时间单独开发，然后被“合并在一起”以达到预期的整体功能。我们会在第 20 章更详细地讨论可重用组件；这里，我们只是简单地介绍概念，说明 UML 标记法。

采用标准化组件的原因众所周知，并且显而易见，实际上，难以想象哪个行业没有广泛使用它们。例如，房屋通常是由砖块、屋顶木材、瓷砖、门、窗框、电器元件、排水管道、地板等等设计和组建的，这些都是从建筑材料中挑选出来的。建筑师可能使用这些组件去设计(或者由建筑工人去建造)整体形象、平面图和房间数量方面与众不同的房屋。区别在于标准组件集成的方式。

对于被当作组件的所有事物来说(例如窗户)，必须按照一定的方式被规范，以允许建筑师、建筑工人等等作为单个简单的事情来完成，即便实际并非如此。另外，专业设计师处理窗户设计的问题(采用什么材料和结构等等，从而制作好的窗户)，专业制造师将之组建为设计者的规范。标准窗户的构造细节可能会因为设计的改善、可使用的材料或者新的建筑方法而改变。然而只要诸如高度、宽度和整体外观之类的关键特征没有改变——实际上，窗户的接口——在将来需要的时候，新的窗户就能够替换相同类型的旧窗户。

同样，软件组件对请求其服务的其他组件来说，会隐藏实现细节。因此，不同的子系统在操作上会有效地分离。这大大降低了互相交互造成的问题，即便它们是在不同的时间，使用不同的语言开发的，或者在不同的硬件平台上执行的；也允许一个组件可以被另一组件替换，只要二者具有相同的接口即可。按照这种方式规定的子系统被认为是互相解耦合的。

这一方法可以被扩展到任意复杂度级别。假定满足如下指标，软件系统的任何部分——或者模型的任何部分，在一些情况下都将被认为是可重用的：

- 组件满足明确且通用的需求(换言之，发布了连贯一致的服务或服务集)。
- 组件具有一个或多个简单的、定义完好的外部接口。

面向对象开发特别适用于设计可重用的组件。精心挑选的对象已经满足上述两条指标。面向对象的模型以及代码，也会按照有利于重用的方式组织。例如，Coleman 等人指出：一般化层级结构是组织组件分类的一种特别实用的方法。这是因为它们鼓励搜索者首先查找通用的分类目录，然后逐步修改搜索，使之达到更为具体的级别。

继承允许“软件架构师”从现有的类中派生新的类。因此，新的软件组件的某些部分通常可以不费力气就能构建。只需要添加特定的细节。在其他大部分行业中，没有合适的可类比之处(虽然在设计活动之间会有更为密切的比较)。新窗口的生成类似于之前的所有窗户。维护也是如此，可能会更加简单——在一般化层级结构中，在超类级别定义的特性在任何子类的实例中也是可以使用的。

8.4.1　组件的 UML 标记法

在 UML 中，建模组件的基本标记法如图 8-10 所示。这显示两个组件之间的接口是通过球状连接符连接的。组件 A 具有供给接口，为其他知道如何请求这些服务的组件提供服务。同时，组件 B 具有需求接口，请求由其他组件上的接口提供的服务。这种请求服务的协议形式与调用操作相同。请求组件发送包含操作名称和任何必要参数的消息。

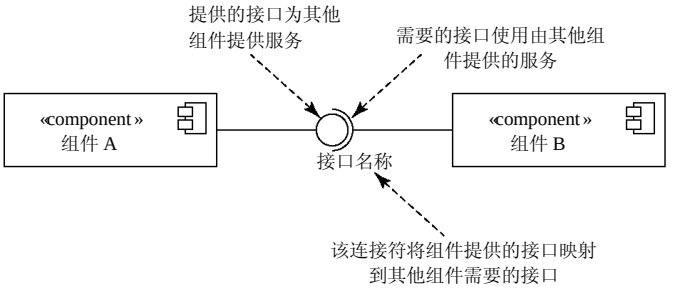

图 8-10　组件结构图的 UML 2.0 基本标记法

每一个接口可能提供一系列服务，每一个服务有其特定的协议或签名。对它们的建模与组件的内部类接口完全独立。上述分析生成的接口操作暗示了 Cheesman 和 Daniels 所说的接口的“信息模型”，这可以为组件内部设计的建模给出良好的起点。

在本节，我们介绍了组件标记法的一些元素，显示了它们如何被用于建模基于组件的架构，并且规范组件接口的重要方面。然而，这当然不是意在为基于组件的开发提供如何分析软件组件的完整描述。更有兴趣的读者应该参考该领域专业人士的图书作品，例如 Cheesman 和 Daniels 的作品。在我们看来，这是在组件建模方面条理最为清晰的一本书。

8.4.2　基于组件的开发

基于组件的开发(CBD)中的很多活动对于其他软件系统的开发来说基本相同。组成单个组件的类必须确认、规范、设计和编写代码。这些活动的执行正如本书其他部分描述的那样。CBD 不同于简单的面向对象分析和设计，主要在于 CBD 涉及了组件架构和组件交互的规范。较之我们到目前为止描述的单个类或小型组合结构的种类来说，CBD 在更高的抽象层面上与界面相关。

组件自身可以在不同的抽象层面被规范。Cheesman 和 Daniels 讨论了单个类可以采取的不同形式。首先，如果在集成的时候互相工作，组件必须遵循某种类型的通用标准。日常中的示例就是很多电子设备遵循了定义工作电压和插口大小形状的标准。不同国家的电压和插口设计也不一样；因此机场需要出售旅行充电器。软件组件的标准被有效实现，以匹配被设计用来工作的给定组件所处的环境。

组件功能也是很重要的。大部分电子吉他放大器的扬声器和耳机的输出插座都被设计为接收同样的插头连接器。但是，如果您将一对扬声器插入耳机的输出插座，那么可能会烧毁耳机。或者，如果是晶体管放大器，那么可能会将之烧坏插入扬声器只是允许您听到吉他。组件的行为是根据组件规范描述的，大部分内容是由组件与其他组件的接口来定义的。

每一个规范会有多种实现。规范的不同实现之间应该可以互相替换，就像吉他手能够拔掉一个吉他而插入另一个吉他一样。毕竟，这就是使用组件的全部意义。

可以想象组件实现只能被安装一次，但是更为可能的情况是，组件的实现可以多次被安装。考虑将 Web 浏览器(例如 Firefox 3.0 版本)作为示例。我们可以假定浏览器有单个规范，但是对主要的操作系统(Windows、Mac 和 Linux)都有不同的实现。每一种实现都会在全世界的上百万台计算机上安装。对于每一台计算机来说，会安装实现的一个副本，为了正确运行，需要注册操作系统。最后，每次打开软件的时候，就会启动新的组件“对象”。这将我们最

终带入实际工作的组件实现的级别——在该级别，数据被存储，进程被执行。

CBD 与传统系统开发的另一不同之处在于，CBD 不仅仅创建了新的组件，也重用了现有的组件。在项目早期，可能会有内部已经开发的组件，或者通过外部从第三方购买的组件。不管何种情况，都需要以标准化的方式描述组件的分类，以便开发者能够找到有用的组件，或者——同等重要的是——可以确定尚未存在的组件。因此，CBD 生命周期通常会有一个模式，用于在组件的开发和使用之间区分，并且用于认识到在从规范到最终使用的过程中，全过程管理组件的需要。我们会在第 20 章再次讨论管理组件的问题。现在我们关注的是，介绍如何在规范中使用 UML。

8.4.3　组件建模实例

飞机座位预定系统提供了一种常见的软件组件使用场景，这部分是因为遗留系统(参见 6.2.1 节)在业界仍旧广泛使用。这些遗留系统必须与新系统兼容操作，它们自身通常也使用面向对象的方法开发，并且也(往往越来越多地)在 Web 上交互。为这种环境开发的新系统组件必须能够与已经使用的组件交互——一些正在使用的组件在设计和实现上非常陈旧——并且能够与即将使用的组件交互，这样的一些组件还未被规范。

假定可以从各种渠道预订机票，包括传统的旅行社、在线旅行社和公司自己的网站。组件架构是合理的，因为单个组件可以按照即插即用的方式被替换，而不需要扰乱其他系统操作。这允许最低程度地更新旧的组件。这也允许在使用单个组件管理所有事务的时候，允许替换不同的预定程序和平台。

图 8-11 显示了飞机可能的基于组件架构的一部分。在这一简单的视图中，主要的组件是处理预定、处理支付、保留顾客记录、检查机场乘客、管理航班信息(包括乘客在飞机上的位置)的系统。实际的飞机当然可能需要很多其他的系统来支持运作。

“预订”组件提供了被称为 MakeBooking 的接口。这对于其他任何了解该接口如何使用的系统来说，也是可以使用的，而实际上只是意味着知道由接口和协议提供的服务或者每项服务的签名即可。组件的实现细节被封装，封装方式与封装对象的特性和操作时使用的相同。

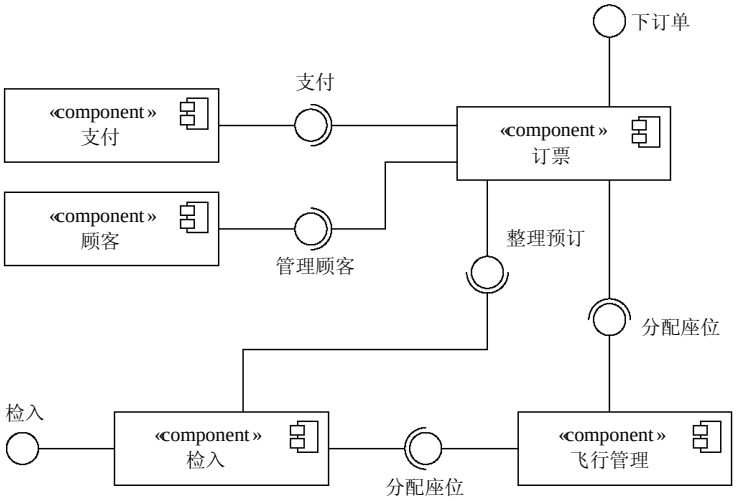

图 8-11　航线系统的组件结构图

在这种情况下，可能使用该接口的其他系统包括旅行社和机场网站的电子商务系统。接口与客户系统解耦合，允许从其他网络或者通过 Web 远程预订机票，而客户系统旨在从预定系统组件的实际操作中预订机票。所显示的架构也允许在其他平台上实现新的客户端。例如，如果机场希望能够从移动设备预订，就不需要对预定系统进行修改。对任何使用该接口的新系统来说，所需要做的只是被设计，以便能够对 MakeBooking 接口发送合适形式的请求。

“订票”组件为“检入”组件提供了接口，“检入”组件需要预定的细节，以保证检入正确的乘客。使用由“支付”组件提供的接口，“支付”组件处理信用卡和其他任何种类的支付方式。“订票”和“检入”组件都使用由“飞行管理”组件提供的接口，以获取(和更新)飞机上可以乘坐的座位的细节。最后，“订票”组件使用由“顾客”组件提供的接口，以获取和更新航班乘客的细节。

可以使用通信图来更加详细地对这些接口的交互进行建模(在第 9 章会全面介绍通信图)。图 8-12 显示了某次预订与检入乘客的交互。注意，单个消息与其参数一起显示，允许分析师详细规范操作，依此类推，类需要去实现每一个接口。一旦组建模型，给出交互的全部情况，组件的规范就可以表示为在所有接口上的全部操作。

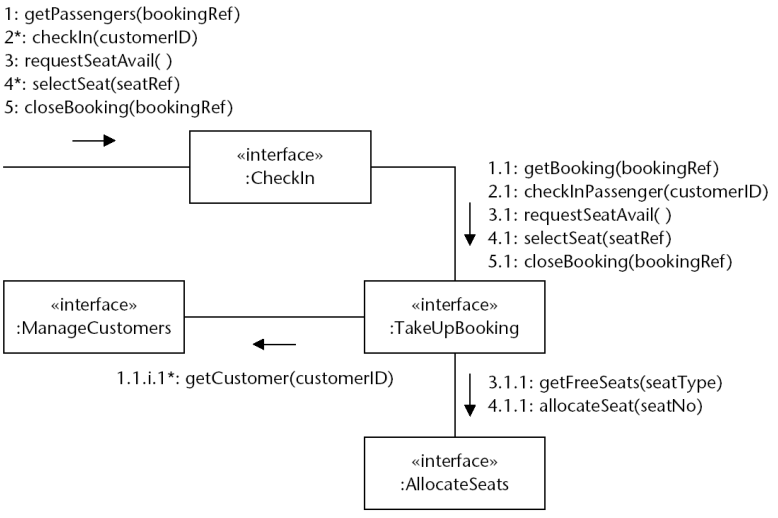

图 8-12　显示组件接口实例之间交互的通信图

图 8-13 显示了“飞行管理”组件及其“分配座位”接口的《realize》依赖关系。组件自身是由在本书其他部分分析和描述的类的包来实现的。

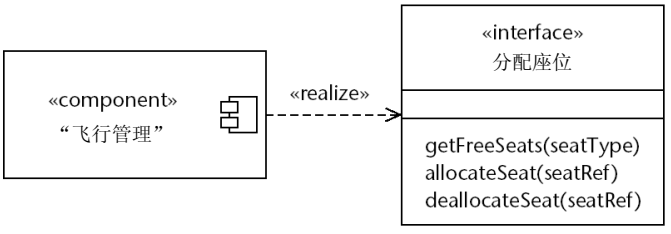

图 8-13　接口与实现它的包之间的依赖关系

注意这幅图，即便是完整的，并且显示了“飞行管理”组件提供的所有接口，也仍旧表示的是组件的规范，而非组件的实现。组件规范关注的是操作和特性的外部视图，与类的规范方式相同。组件实现(换言之，存在于组件中的类和关联的规范)比我们这里显示得更为详细，但是超出了本章的讲述范围。

8.5　软件开发模式

软件开发模式(通常简称为模式)是对重复发生的问题的一种抽象解决方案，这种重复发生的问题根据特定的场景可以使用不同的方式解决。模式广泛应用于系统开发中，大部分都应用于设计阶段，但是也有应用于需求分析、项目管理、业务过程设计、测试等其他阶段的。第 7 章介绍的边界、控制和实体对象架构，实际上就是一种在需求分析和系统设计期间广泛使用的模式。之后我们考虑的其他架构，也是模式应用于分析和设计活动中的示例。如果模式抓住了问题和可能的解决方案的本质，而不是过于指令性，那么模式是很有用的。

8.5.1　模式的起源

在日常用语中，模式指的是一种设计，用于以重复方式(例如壁纸或花布)生成图片或产品。这说明模式是一种一般化，该术语在软件开发中也经常使用。模式一词的使用要追溯到建筑师 Christopher Alexander，他首次使用模式一词描述建筑中重复发生的事情的解决方案。Alexander 确认了建筑物中很多相关的有效、和谐的建筑形式的开发模式。Alexander 的模式解决了很多建筑问题——例如，在房间中安装门的最佳位置，如何组织和设计建筑物中等候区域的结构，以保证等候是一种舒服的经历。Alexander 认为自己的模式已经成为一种“设计”语言，可以用来开发和描述对于重复发生的建筑问题的解决方案。Alexander 对于模式的定义如下：

> 每一种模式都描述了在我们的环境中一次次重复发生的问题，然后描述了问题的核心解决方案，这样就可以多次使用此解决方案，而不需要重复劳动。
>
> ——Alexander et al.(1977)

Alexander 没有引用建筑物或建筑结构，但是对于“我们的环境”，他指的是我们所生活的环境。显然可以类比软件开发，面向对象的一些早期社区采用了他的思想。Beck 和 Cunningham 归档了最早期的一些软件模式，以便在 Smalltalk 环境中描述界面设计的一些方面。Coplien 对一系列特别适用 C++编程设计的模式进行了分类(使用特定编程语言构建的模式现在被称为 idiom)。

Gamma 等人撰写的 *Design Patterns: Elements of Reusable Object-Oriented Software*[1]一书的出版也说明了在软件设计中使用模式的重要意义，而其他学者考虑了在分析、组织问题和系统架构方面的模式。软件模式也被应用于非面向对象的开发方法中。例如，Hay 为数据

1 此书的 4 名作者号称 GOF，因此本书也被称为“GOF 之书”。

建模确认了一系列模式。这些模式包括与诸如参与方和合同相关的概念，它们在信息系统中有着广泛的应用。

8.5.2 什么是软件模式

Riehle 和 Zullighoven 将模式描述为在特定场景下重复发生的具体形式的一种一般化抽象。可以将“具体”解释为“专用”或“特定”。Gabriel 的定义更为详细，表达了模式的结构：

每一个模式都是一个三方规则，表明了特定上下文、在特定上下文中重复发生的特定系统作用力以及允许这些作用力自我发生作用的软件配置之间的关系。

在上述定义中，上下文可以被理解为场景和前提条件的集合，作用力是需要解决的问题，而软件配置处理和解决这些作用力。Coplien 确认了如下所示的模式的关键方面：

- 解决问题
- 是证明过的概念
- 解决方案并不明显
- 描述了关系
- 模式具有明显的人为组件

人为组件不仅仅是可以应用于可工作应用程序的好的用户界面；人为组件还关注在组建应用程序时所使用软件结构的本质。软件模式应该能够从人类的视角生成结构。好的软件模式提供了不仅仅能够工作的解决方案，也提供了具有审美标准的解决方案——在一定程度上是优雅的。这一审美标准有时被称为 QWAN。QWAN 的精确本质属于有争议的话题，我们针对模式的讨论不会涉及优雅方面的内容。然而，读者自己可以从开发优雅模式的角度来确定模式。

获取并且归档模式被证明是好的实践，反面模式以同样的方式获取，这被证明是不好的实践。我们不仅仅应该确保软件系统具有好的实践，也应该使之避免所谓的误区。反面模式是归档失败解决方案的一种方式，这种解决方案尝试解决重复发生的问题。反面模式也可以包含被证明是有效的经过返工的解决方案。典型示例就是蘑菇管理，与软件开发组织的域有关。描述了采用特定的策略将系统开发者与用户分离，从而试图限制需求漂移。在这类组织中，需求从诸如项目经理或需求分析师之类的中间人员传递给开发者。该模式的消极结果是，不可避免地出现分析文档的不充分，并且难以解决。另外，设计决定的作出也缺少用户的参与，发布的系统可能无法反映用户的需求。Brown 等人建议的这种返工的解决方案使用一种螺旋模型(参见第 3 章)。其他返工的解决方案包括由动态系统开发方案(DSDM)建议的(我们会在第 21 章介绍 DSDM)开发团队中域专家的参与。

Coad 等人区分了策略和模式：他们将策略描述为旨在达到某个已经定义的目的的计划，而模式是体现值得模拟的示例的模板。这与之前描述的模板的概念有些许差别，因为在一定程度上没有强调情景方面。Coad 等人的策略示例是“组织和设定优先特征”，这与设定优先需求的需要相关(在第 3 章已经讨论)。

8.5.3　分析模式

分析模式在本质上是类和关联的一种结构，这种结构在很多不同的建模场景下重复发生，每一种模式都可以就如何建模特定的需求取得一般的理解，因此模型不需要每次在类似场景发生的时候都重新开发。模式可能是由类的整个结构组成的，相比通常单独使用一般化，这种抽象层次可能更高。另一方面，模式需要将很长时间用于详细规范，因此不应该与组合结构或组件混淆。

分析模式的简单示例是 Coad 等人的事务-事务细目模式，如图 8-14 所示。

图 8-15 显示了可能在销售订单处理系统中使用的模式。此处的事务是 SalesOrder 类，而事务细目是 SalesOrderLine 类。注意，我们已经将关联建模为组合。这不同于已发布的模式，但是会在此进行判断。

在各种不同的场景下，会使用非常类似的结构(例如船运和船运细目，支付和支付细目)。软件开发初学者需要学习这一结构，或者重新实现——但是后者效率更低。将之描述为模式突出了作为一种有用的开发知识，使得对于初学者更具可读性。另一个使得该模式具有优势的原则是模式的交互低耦合特性(参见第 14 章)。

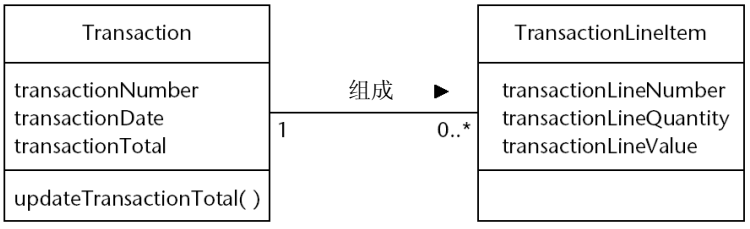

图 8-14　事务-事务细目模式

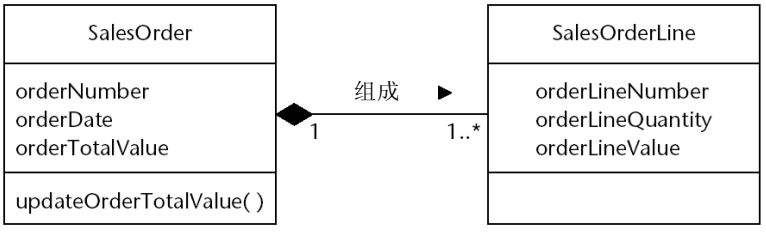

图 8-15　事务-事务细目模式的简单应用

Fowler(1997)描述了在业务建模场景(例如财务、贸易和组织结构)中重复出现的大量模式。图 8-16 显示了 Fowler 的责任模式，作为实践中分析模式的一种阐述。为了简单起见，我们只讨论类结构，尽管模式的归档通常比这更为详细(参见第 15 章)。责任结构可以有很多种，例如管理或监视合同。在 Agate 案例中，该模式应用于几种不同的关系：经理与其管理的职员之间的关系、客户和客户合同之间的关系，或者客户和广告团队经理之间的关系。因为关系自身的细节被抽象为 AccountabilityType，这是类结构，所以对于这些关系来说足够通用，给出了一系列合适的特性、操作，以及与特定应用程序模型中其他类的关联。类似地，将 Person 和 Organization 一般化为 Party，进而允许模式表示个人、组织或个人与

组织的混合体之间的关系。

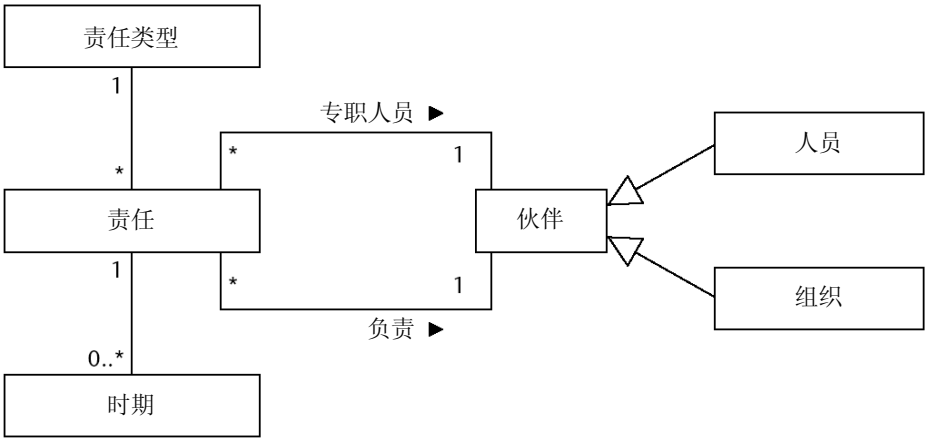

图 8-16　责任分析模式

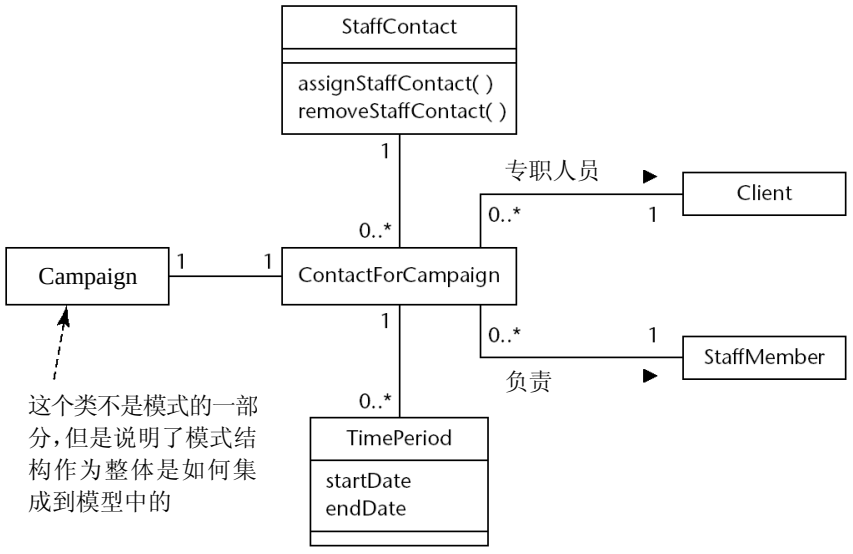

图 8-17　应用于 Agate 公司职员-合同关系的责任模式

图 8-17 显示了应用于 Agate 公司职员-合同关系的模式。在该上下文中，无须对将“专职人员”和“负责”角色一般化为“伙伴”进行建模。这阐述了可以合理地采用模式建议的结构以符合场景。

在为 Agate 开发分析模型的时候，另外一种可能有用的模式是组合模式。然而，因为该模式从分析和设计的角度来说都很有用，所以我们会推迟到第 15 章才介绍。进一步使用分析模式是经验丰富的分析师主要使用的一种先进方法，我们留给感兴趣的读者对此进行深入阅读。它们与设计模式密切相关，我们会在第 15 章详细介绍设计模式。

8.6　本章小结

在本章，我们介绍了完善分析模型的主要方法。这么做主要是为了最大化重用的机会。有三种特定的方法能达到这一目的。面向对象方法(一般化、组合、封装、信息隐藏)的抽象机制是第一种方法。它们从自身意义上讲是很重要的，并且有助于显著增加重用之前分析工作的可能性。它们也对其他两种重用方法有影响，这两种方法分别是可重用组件以及软件开发模式。尽管细致的分析和规范依然十分重要，但是组件能够极大地降低所有开发阶段的工作量。模式则是具体化最佳实践的知识的总结。

不管采取什么样的形式，重用在本质上只会以这三种方式进行。首先，现有的组件或结构会从项目之外的源中导入。这需要仔细地评估当前项目需求和可使用组件及结构的特征之间的吻合度。其次，新的可重用组件或结构会被开发用于当前系统的多个部分。这需要分析模型能够高度地抽象项目不同层面之间的共同特征，使它们变得清晰。最后，新的可重用组件或结构可能会被开发用于导出到其他项目。此处同样需要建模需求，以确认项目的这些方面能够被一般化并用于其他上下文中。

可替代软件组件的明确建模在软件开发中是一种相对较新的方法，我们希望未来在这一领域有更多的革新。模式现在是更为成熟的方式，可以对软件开发活动的诸多方面有益地进行共享和重用。本章已经考虑了如何在软件开发中使用组件和模式，特别是从分析的角度讲述。后续章节会更加详细地讨论模式和组件的应用，以解决架构和设计中的问题。

问题回顾

8.1 组件的优点是什么？
8.2 为何会发生 NIH 综合症？
8.3 面向对象的哪些特征有助于创建可重用组件？
8.4 区分组合和聚集。
8.5 操作为何有时在子类中重定义？
8.6 抽象类的目的是什么？
8.7 封装为何对于创建可重用的组件很重要？
8.8 一般化为何对于创建可重用的组件很重要？
8.9 在什么时候不应该在模型中使用一般化？
8.10 在软件开发的上下文中，术语“模式”的含义是什么？
8.11 模式如何帮助开发者？
8.12 什么是反面模式？

案例研究、练习和项目

8.A 从库中找到用于对图书、视频等分类的编码系统。使用 UML 标记法绘制部分结构为一般化层级结构。在模型中考虑“类”的一些特性，以显示底层是如何被逐步细化的。

8.B 选择自己熟悉的一种商业领域(业务、工业、政府机构等)。确认软件产品显示一般化的一些使用方式，以及组件用于输入、显示一般化的一些方式。

8.C 在 8.3.4 节，我们看到：一般化可能是建模 Advert 和 NewspaperAdvert 之间关联的一种适合方式。从 A1.3 节确认 Advert 其他可能的子类，重复检查每一个子类。如果有的话，哪一个会通过检查？您真的认为这里只有两层的层级结构吗？解释原因。重新绘制 Agate 类图以包含您认为恰当的所有一般化。

8.D 对于每一个新的 Advert 子类，给出合适的特性、操作以及聚集或组合结构。

8.E 重新阅读 FoodCo 案例研究资料，确认合适的一般化或组合。将它们添加到您为练习 7.C 和 7.D 绘制的类图中。

8.F 考虑 FoodCo 案例中的类图。尝试确定合适的事务-事务细目模式或责任模式，并且重新绘制合适的图。

拓展阅读

Ambler 讨论了类和用例重用中的一般化问题。然而，该主题在最近的书中都少有关注。因此，Rumbaugh 等人依旧保留了最清晰的总结。本书成书于 UML 之前，但 Rumbaugh 是之后修订 UML 的三位作者(与 Jacobson 和 Booch 一起)之一。

Jacobson 等人非常清晰地讲解了需求分析中，与组合、一般化和包的确认相关的架构和重用问题。

现在有大量关于开发可重用软件组件的书。Cheesman 和 Daniels 撰写的书中简洁而透彻地介绍了组件规范中 UML 的使用，并且是该主题最好的书籍之一(本书成书于 UML 2.0 之前，因此其中使用标记法与更新后的规范有些许不同)。

Gamma 和 Buschmann 等人编写的图书仍旧是介绍各种模式方面的重要资料。

Withal 归档了超过 30 种需求模式，很多应用于分析问题，而其他的关注诸如性能和安全之类的非功能性需求。Fowler 描述了大量的模式，特别地，这些模式与分析相关，并且仍旧高度关联。这些模式没有使用 UML 表示，而现在大部分模式已经使用 UML 格式重新绘制，在 Fowler 的网站 martinfowler.com 上可以看到，其中还有更为深入的模式示例。Coad 等人也给出了一些分析和设计模式。大部分使用 Coad 自己的标记法表示，但是一些使用 OMT 标记法和统一标记法(UML 的一种早期版本)表示。

模式主页可以在 http://hillside.net/patterns/找到。更多有用的模式保存在 Portland 模式库中，网址为 www.c2.com/ppr。