返回总目录

目　　录

第 10 章　UML 与设计模式

过去几年，在面向对象领域中的一个重要突破就是提出了设计模式的概念。设计模式由于实用而受到欢迎：它们能够表达和重用专家技术和经验，能进行系统框架设计，在表达上既经济又清楚，从而受到人们越来越多的重视。

模式概念是建筑师 Christopher Alexander 提出的，他提出可以把现实中一些已经实现的较好的建筑和房屋的设计经验作为模式，在以后的设计中直接加以运用。他还定义了一种“模式语言”来描述建筑和城市中成功的架构。Alexander 的方法得到软件业人士的青睐，在 20 世纪 90 年代掀起了一股在软件设计中应用模式的讨论。1994 年 8 月，召开了编程模式语言（Pattern Languages of Programs，PLoP）大会。尽管软件的设计模式不一定要和面向对象有关，但由于面向对象很容易描述设计抽象，因而许多设计模式都和面向开发有关。Erich Gamma，Richard Helm，Ralph Johnson 和 John Vlissides 四个人（被称为“四人组”）在 1995 年初出版了一本书《Design Patterns: Elements of Reusable Object-Oriented Software》(设计模式：可重用的面向对象软件的元素)。其中对设计模式进行了基本分类，并且讨论了一些新的需要研究的模式。不过，软件界的设计模式仍然处于起步阶段，远不如它在建筑业中那么成熟和完善。

对模式的研究有许多方面，例如，有的讨论如何在不同的领域内如 CORBA 和项目管理中应用模式，有的则讨论模式系统，希望能够识别出不同级别的模式，最终形成一个完整的模式系统；还有的则研究组织系统的架构模式：子系统、责任和规则分配以及有关子系统如何通信和合作的准则。而 UML 的设计师们也正在研究如何用设计模式支持软件开发过程，同时 UML 本身就支持模式的表达。

在本章我们首先介绍模式的一些基本概念，如模式的定义、分类和它的优点，接着将介绍 UML 对设计模式的支持。最后，将通过几个具体的例子来讨论如何使用设计模式进行系统设计。

10.1　什么是模式

那么，什么是模式（Pattern）呢？Alexander 给出的定义最为经典：

每个模式都描述了一个在我们的环境中不断出现的问题，然后描述了该问题的解决方案的核心。通过这种方式，你可以无数次地使用那些已有的解决方案，无需再重复相同的工作。

模式作为现实世界中的一个元素，都是以下这三者之间的关系，它们是：特定的情景、在该情景下反复出现的特定压力系统和使这些压力能够自我释放的空间配置。

作为语言的一个元素，模式是一条指令，说明了如何重复地使用这个空间配置，一旦给定的情景适当就释放给定的压力系统。简而言之，模式是一种出现在现实世界的事物，同时，它也是一条告诉我们如何创建、何时创建该事物的规则。它既是一个过程，又是一种事物；既是对一个存在事物的描述，又是对生成该事物过程的描述。

模式具有双重性：它既是生成的（generative）又是描述的（descriptive）。因为它既是对重复生成的架构元素的描述，又是对如何以及何时创建该元素的规则。从本体论（ontological）的观点来说，模式的“生成的”属性指模式的内容（content），即指反复出现的事物的自身，从认识论（epistemological）的观点来说，“描述的”属性指模式的形式（form），是我们捕捉并表述这一事物的方式（通过“问题—情况—压力—解决”的形式）。

简而言之，设计模式的核心是问题描述和解决方案。问题描述说明模式的最佳使用场合以及它将如何解决问题。解决方案是用一组类和对象及其结构和动态协作来描述的。

10.2　为什么要使用设计模式

在软件业中，设计模式是面向对象系统使用的一些可重用的已经得到了很好证明的巧妙、通用、简单的设计解决方案。软件设计中的设计模式具有以下特性：

- 巧妙：设计模式是一些优雅的解决方案，一般很难立刻设计出来。
- 通用：设计模式通常不依赖某个特定的系统类型、程序设计语言或应用领域，它们是通用的。
- 得到了很好的证明：设计模式在实际系统和面向对象系统中得到广泛应用，它们并不仅仅停留在理论上。
- 简单：设计模式通常都非常小，只涉及很少一些类。为了构建更多更复杂的解决方案，可以把不同的设计模式与应用代码结合或混合起来使用。
- 可重用：设计模式的建档方式使它们非常易于使用。正如前面提到的，它们非常通用，因而可用于任何类型的系统。注意此处的可重用性是在设计层，而不是在代码层。设计模式平在类库中，只有系统架构使用它们。
- 面向对象：设计模式是用最基本的面向对象机制如类、对象、通用化和多态等构造的。

那么，为什么要使用模式呢？首先，使用设计模式，就可以更准确地描述问题和它们的解决方案；其次，使用设计模式可以具有一致性（consistency），即如果对一些已知的问题我们有一类标准的解决方案，那么在遇到相同问题时，我们能够采取一致的方法，这就使代码更容易理解；使用设计模式的再一个原因就是，在遇到问题时，无需每次都从底层做起，而是可以从标准解决方案——设计模式入手，并对它改编，使之适应特殊问题的需要。这样就节省了时间，并且提高了开发的质量。模式为面向对象软件开发人员提供了以下机制：

- 根据实际系统的开发经验提供对共同问题的可重用的解决方案。
- 提供类和对象级之上抽象的名称。通过设计模式，开发人员能够在更高的层次上讨论方案，例如：“我建议我们可以用 Bridge 或 Adapter 来解决这个问题”（ Bridge 和 Adapter 是两种设计模式）。
- 提供对开发的功能性和非功能性方面的处理方案。许多模式特别强调了某些面向对象设计擅长的领域：区分接口和实现，放松各部分之间的依赖性，隔离硬件和软件平台，并且具有重用设计和代码的潜能。

- 提供了开发框架（framework）和工具的基础，在设计可重用框架时，设计模式是最基本的架构。
- 为面向编程和设计学习提供教育和训练支持。通过对设计模式的研究，能够深入理解良好设计的最基本性质，从而在后面的设计中加以模拟和应用。

10.3 模式的分类

不同的书籍和论文在描述设计模式时的方式各有差异，有的采用文本方式，有的采用文本和建模语言模型的方式，也有多种对模式的分类方法。“四人组”书中定义了 23 种模式的分类，该书的文档风格现在成为定义新模式的模板。该书定义了以下几种模式分类：

- 创建模式：针对例程（包括创建什么对象，以及如何及何时创建）和类及对象的配置。允许系统中有不同结构和功能的“产品”对象；常见的创建模式有：Abstract，Factory，Factory Method，Prototype，Singleton，Object Pool。图 10-1 是一个 Builder 的例子。

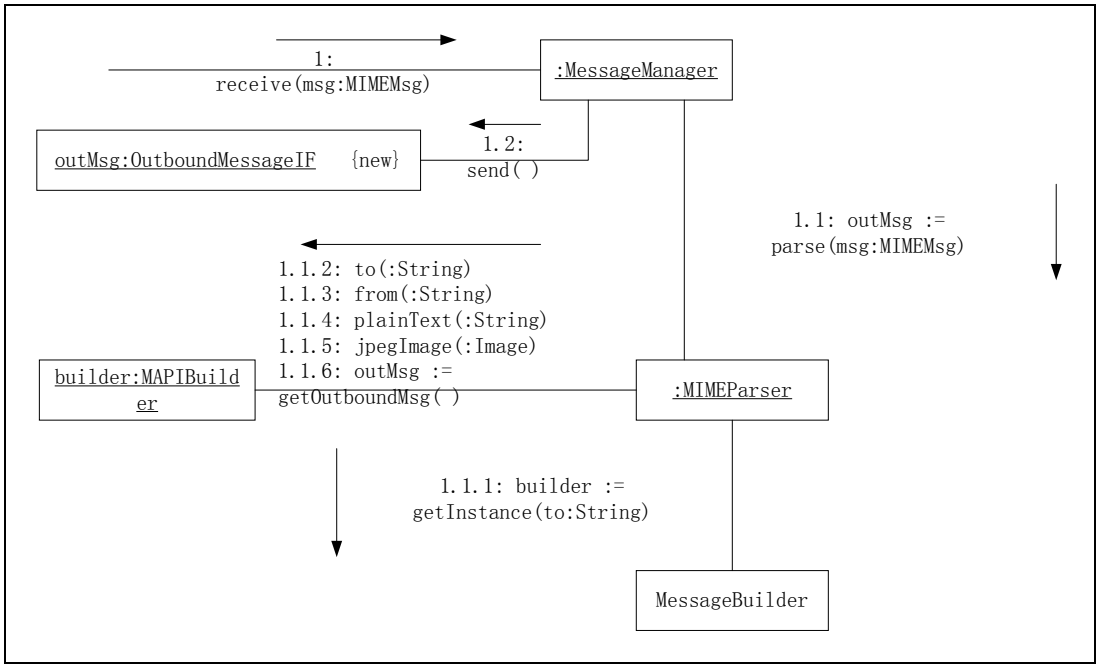


图 10-1 Builder 模式举例

- 结构模式：针对在大的结构中使用类和对象的方式，并把接口与实现分离开来。

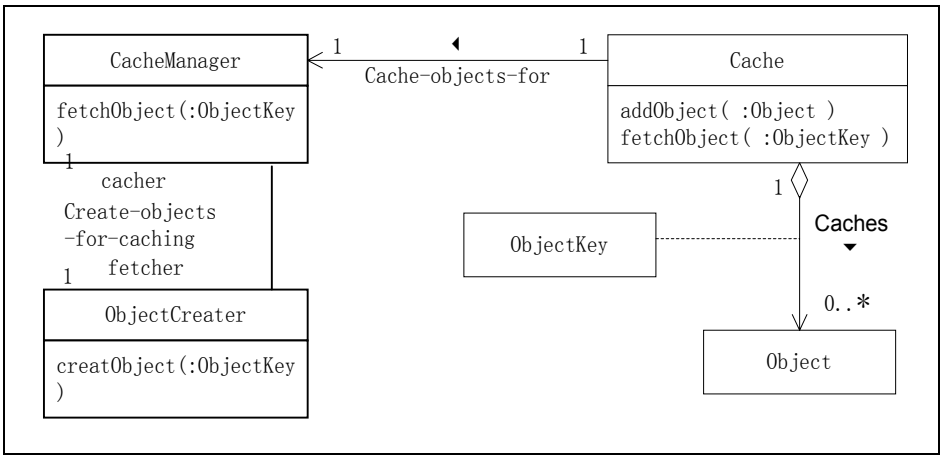


图 10-2　Cache Management 模式

- 行为模式：针对在对象之间责任分配的算法，以及类和对象之间的动态交互。行为模式不仅仅处理对象的结构，还处理它们之间的通信 Chain of Responsibility，Command，Little Language/Interpreter，Mediator，Snapshot，Observer，State，Null Object，Strategy Template，Method，Visitor。图 10-3 是 Visitor 模式。

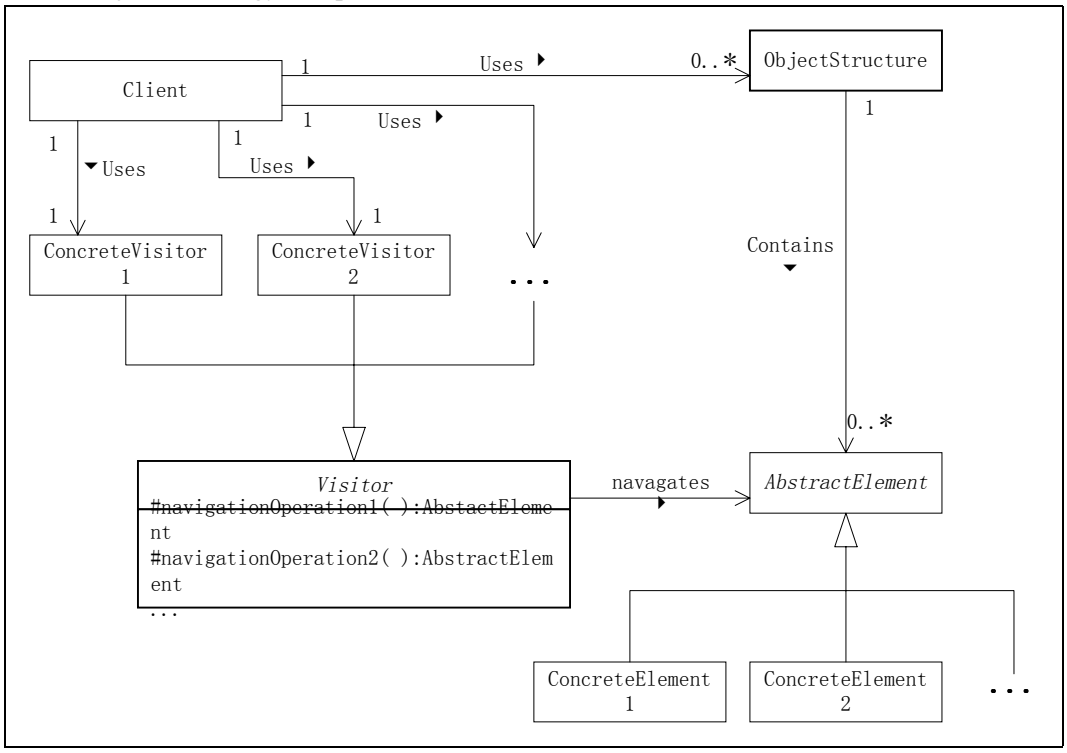


图 10-3　Visitor 模式

10.4 模式的组成元素

Alexander 说，我们定义的每个模式，都应该以规则的形式，在情景、情景下产生的压力系统以及允许这种压力自我释放的配置之间建立关系。他还推荐使用图来描述模式。Alexander 所使用的模式的描述格式称为“Alexandrian 方式”，在[GoF]中使用的称为“GoF 格式”。不管模式的记录方式有多大的区别，它都应该包括一些本质的元素。以下是一些公认的模式基本元素：

- 名

模式必须具有一个有意义的名。这样就可以用一个词或短语来指代该模式，以及它所描述的知识和结构，好的模式名组成讨论概念抽象的词汇表。有时，一种模式可能有多个公认的名字，这种情况下最好把它的绰号以及同义词在“别名”或“也作为”中列出，有的模式格式还提供模式的分类。

- 问题

陈述问题并描述它的意图：它在给定的情景和压力下所要达到的目标和目的。通常压力是阻碍完成这些目的的。

- 情景

是问题以及它的解决方法重复发生的“前置条件”，告诉我们模式的可应用性（applicability）。也可以把它看作应用模式前的系统初始配置。

- 压力

描述有关的压力和约束、它们之间以及和要达到的目标之间是如何交互/冲突的。常常使用一个具体的想法作为模式的动机（motivation）。压力揭示了问题的错综复杂，同时定义了在它们产生冲突时必须做的折衷。好的模式描述应该完全封装所有对它有影响的压力。

- 解决方案

描述如何实现预期结果的静态关系和动态规则，相当于给出指令来构造所需要的产品。描述中可以有画、图和句子，它们指明模式的结构、参与人和他们之间的协作，进而说明问题是如何解决的。解决方案不仅要描述静态结构，还要描述动态行为。静态结构说明了模式的形式和组织，而行为动态则使模式变得生动。对模式解决方案的描述可能指明了在尝试构造该方案的具体实现时应该记住的指南，有时也会描述该解决方案的一些变形或专有化。

- 例子

一个或多个应用模式的例子显示：特定的初始情景；如何应用模式；模式如何改变情景；以及结果的情景。例子帮助读者理解模式的使用和可应用性，虚拟的例子和模拟尤其具有说服力。例子还可以附加一个实现例子，显示解决方案实现的一种方式。

- 结果情景

在应用模式之后系统的状态或配置，包括应用模式所产生的结果（包括好的和坏的），以及模式可能带来的其它问题，它描述了模式的“后置条件”和“边际效应”。这有时称作压力的释放，因为它描述了哪个压力已经被释放，哪个还没有释放，现在哪个模式是可

应用的。为模式产生的最后情景编制文档有助于把其它模式的初始情景关联起来（因为在大型项目中，一个模式通常只是完成整个任务的一小步）。

- 基本原理

对模式中的步骤和规则的证明性解释，告诉我们模式实际上的工作方式，以及为什么要采取这样的方式，为什么这样是最好的。模式的解决方案组件可能描述模式的外部可见的结构和行为，而基本原理组件则说明了模式内在的结构和主要的机制。

- 相关模式

在同一模式语言或系统中，该模式与其它模式之间的静态和动态关系。相关模式通常会有相同的压力，并且会有与其它模式相容的初始或结果情景。这样的模式可能是前任模式，其应用导致该模式的应用；也可能是后继模式，其应用紧跟在该模式应用之后；可能是其它可选模式，针对与该模式相同的问题，在不同的压力和约束下，提出不同的解决方案；还可能是互相关的模式，与该模式同时应用。

- 已知使用

描述该模式在已有系统中的出现和应用。这有助于确认该模式的确是针对一个“重复出现的问题”的一个“得到证明的解决”。模式的已知使用经常用作模式的使用指导。

虽然没有提出严格的要求，但通常好的模式前面都有一个摘要，提供一个简短的总结和概述，为模式描绘出一个清晰的图画，提供有关该模式能够解决问题的快速信息。有时这种描述称为模式的缩略概要，或一个模式缩略图。模式应该说明它的目标读者，以及对读者有哪些知识要求。

10.5 模式的质量

除了包含上述的元素外，定义良好的模式还应该具有一些重要的质量。包括：

- 封装和抽象。每个模式都封装了在特定领域内一个定义良好的问题以及它的解决方案。模式应该有清晰的边界概念，以便于明确问题空间和解决方案空间。模式还应该是抽象，体现了领域知识和经验，并且可能出现在该领域内不同的概念层次上。
- 开放性和可变性。每个模式都应该是开放的，允许对它进行扩展或被其它模式参数化来共同解决大问题。模式的解决方案也应该能够有无限种实现的方式（孤立地或与其它任何模式一起）。
- 可再生性和可共存性。一旦被应用，那么每个模式都会生成一个结果情景，这个情景可能会与一个或多个其它模式的初始情景匹配。而后会继续应用后面的模式，直到最终生成一个“完整的”解决方案。但模式的应用通常不是线性的，很可能一个模式的一部分会在特定的抽象和精细层次上导致其它不同层次上的模式，或与它们复合。
- 平衡。每个模式必须实现它的压力和约束之间的某种平衡，这种平衡可能来自用来使解决方案空间冲突最小化的一个或多个不变量和启发式。这种不变量通常代表了解决问题最基本的原则或思想，并解释了该模式中的每一个步骤（或规则）。

总之，模式的目标是，通过把它的每一组成部分技巧性地组织起来，使它的整体和大于部分和。

10.6　一个简单的模式例子：代理模式

了解设计模式的最好办法就是去研究一个实际的模式，下面我们就以代理模式为例进行讨论。

代理模式是一个结构模式，把接口从实现中分离成不同的类。它的主要思想是用一个代理对象作为另一个真实对象的代理，由这个代理控制所有对真实对象的接入，从而解决了当由于性质、位置或接入限制而使得真实对象无法被直接实例化的问题。这是一个非常简单的模式，但很容易说明 UML 是如何描述模式的。图 10-4 是 UML 中 Proxy 模式的类图。

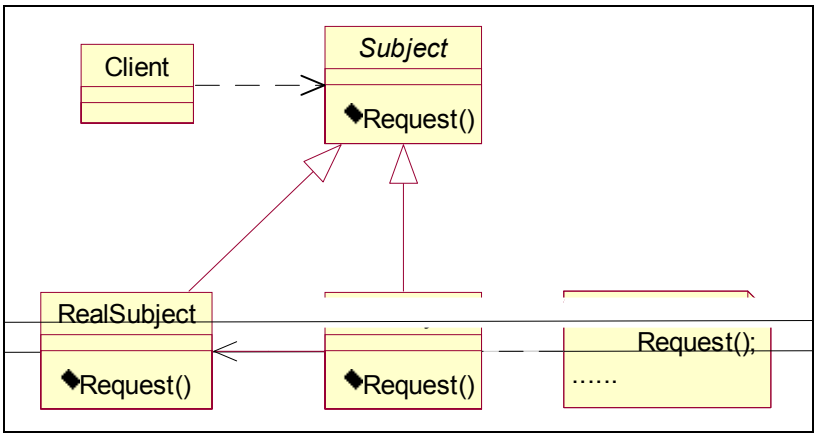


图 10-4　把 Proxy 设计模式描述成一个 UML 类图（注意，Subject 是一个抽象类，因而是斜体）

Proxy 模式涉及三个类：Subject，RealSubject 和 Proxy。图中的 Client 类使用该模式，使用抽象类 Subject 中定义的接口。在 RealSubject 和 Proxy 中都实现了 Subject 定义的接口：RealSubject 实现了接口中的操作，而 Proxy 类则把它收到的所有调用都委托给 RealSubject 去执行，由 Proxy 对象控制对 RealSubject 对象的接入。这一模式有以下几种应用场合：

- 更高的性能和效率：在需要真正的对象之间，系统可以用廉价的代理来减少耗费。当在代理处调用操作时，它首先检查是否已经实例化 RealSubject 对象。如果没有，就把它实例化并把请求委托给它，如果已经实例化了，则立即转发请求。这种方法在 RealSubject 对象的创建非常“昂贵”时很有效，比如，必须读取数据库，复杂的初始化等。许多具有很长的启动时间的系统，通过使用 Proxy 模式可以显著地提高性能。
- 授权：如果需要检查调用方是否具有调用 RealSubject 对象的授权，则可以由 Proxy 来完成。这时，调用方必须给出它自己的身份，Proxy 必须和某个授权对象通信，是否允许接入。

- 局部化：RealSubject 对象可以位于其它系统中，本地的 Proxy 只是在本系统中“扮演它的角色”，而所有请求都被转发给其它系统。本地客户只看到 Proxy。

Proxy 模式可以有许多种变形，如：完成其它功能无需全部委托给 RealSubject，可以改变参数的类型或操作的名词，或者做一些准备工作来减少 RealSubject 的负荷。这些变形体现了模式背后的思想：模式是一种解决方案的核心，可以有多种变形、改装或扩展，但不改变基本的方案。

在“四人组”的书中，是用类图描述模式的，用一个对象图和一个消息图（有点类似 UML 中的序列图）。很多地方是用文本来描述某些方面，如意图、动机、可应用性、结构、协作、实现、举例的代码以及模式的结合，另外还包括同一模式其它的名字、有类似性质的相关模式以及已知的该模式的用法。

模式的代码通常都简单，根据需要把 RealSubject 实例化的 Proxy 类的 Java 代码很简单，如图 10-5 所示。

```
public class Proxy extends Object
{
     RealSubject refersTo;
   public void Request()
   {
        if (refersTo == null)
            refersTo = new RealSubject();
        refersTo.Request();
     }
}
```

图 10-5　把 RealSubject 实例化的 Proxy 类的 Java 代码

10.7　UML 对模式的支持

在应用设计模式时，UML 非常重要，因为它允许模式作为架构的一部分，如图 10-6 所示。

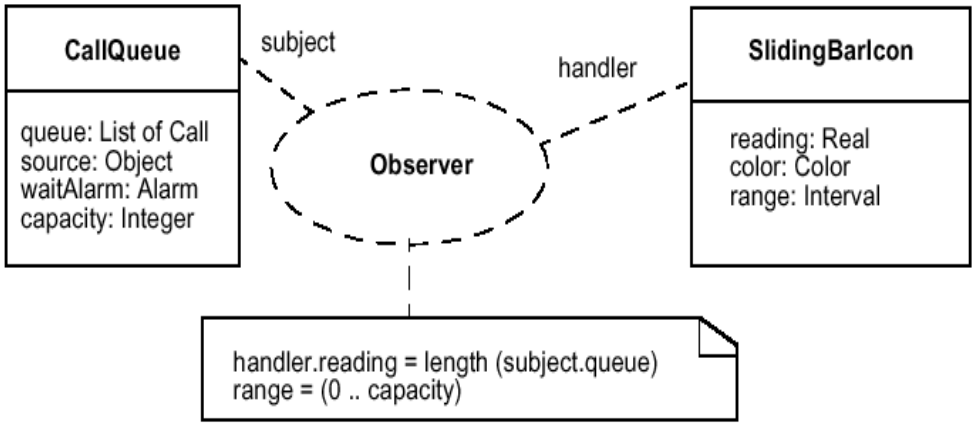


图 10-6　在 UML 中模式的具体化。图中 Oberserver 是设计模式，同时还显示在该模式的特定使用中出现的实际的类

在 UML 中模式是用一个协作描述的。协作描述了情景和交互，其中情景是指协作所涉及的所有对象，它们之间的相互关系，以及分别是哪个类的实例。交互显示了在协作中对象完成的操作（发送消息以及相互调用）。模式具有情景和交互，因此很适合用协作来描述（事实上是用一个参数化协作来描述的）。

图 10-7 中的虚椭圆就是模式的符号，里面是模式的名字，模式必须有名字。图 10-8 是 Proxy 模式对象图，图中表明 Client 对象有一个链接是连到 Proxy 对象的，然后又连接到 RealSubject 对象上。

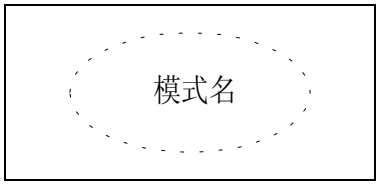


图 10-7　一个协作符号代表一个设计模式

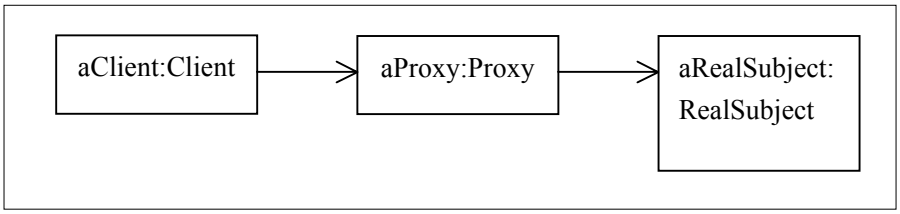


图 10-8　用对象图描述 Proxy 模式的情景

在描述 Proxy 模式时，交互显示了当客户提出请求时对象是如何交互的。图 10-9 中的序列图显示了如何把请求授权给 RealSubject 对象以及结果是如何返回给客户的。

协作图可以在同一个图中显示出情景和交互。图 10-8 的对象图中的情景以及图 10-9 中的交互都体现在图 10-10 中的协作图中了。是否使用协作图或者分别表示在一个对象图和一个序列图中依情况而定。更复杂的模式可能需要描述几个交互来显示该模式不同的行为。

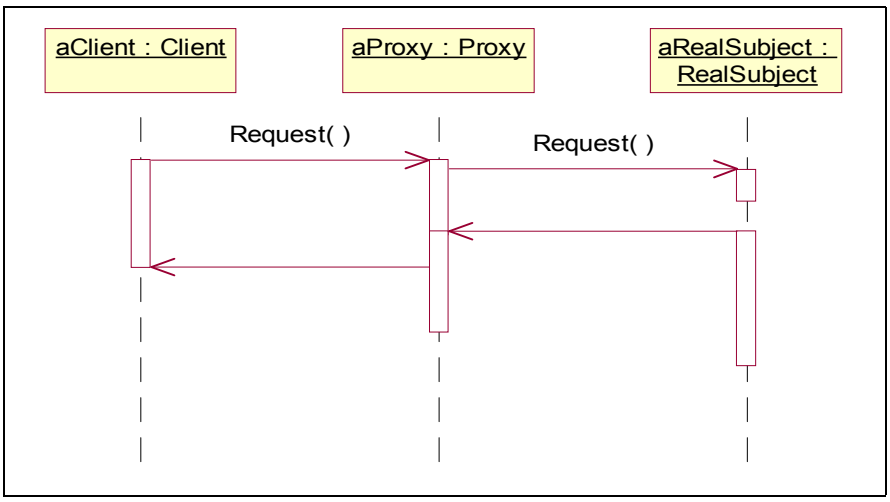


图 10-9　用序列图描述 Proxy 设计模式

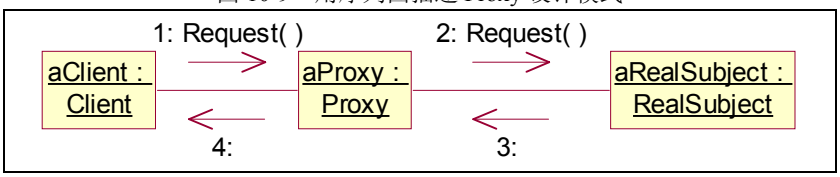


图 10-10　用协作图描述 Proxy 设计模式

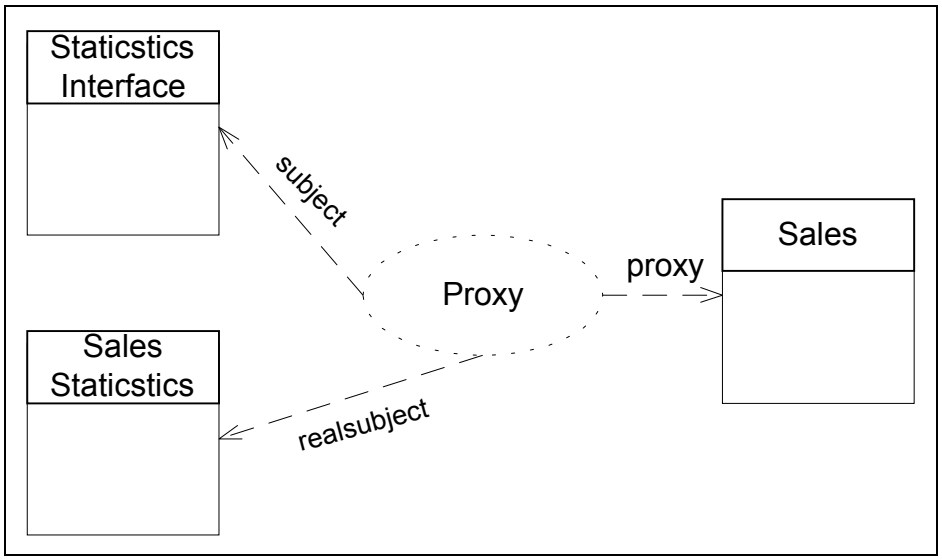


图 10-11　在一个类图中使用的 Proxy 模式，其中 Statistics Interface 类有 subject 参与，Sales 类有 proxy 类参与，Sale Statistics 类有 realsubject 参与

10.7.1　参数化协作

前面提到，实际上是用参数化协作或一个模板模式描述模式的。参数可以是在实例化模板时指定的类型（如类）、关系或者操作。在一个参数化协作中，参数叫做参加人（participant），是类、关系或操作，设计模式参加人是模式定义的完成角色的应用元素。例如，在 Proxy 模式中的担任 Subject、Proxy 和 RealSubject 类的类。在图中，参加人的表示是一个从模式符号与完成不同角色元素之间的一个相关性关联描述的。该相关性上标注了参加角色的名，说明了在设计模式中该类所扮演的角色。在 Proxy 模式中，参加的角色是 Subject，Proxy 和 RealSubject。

如果要扩展协作，则必须用参加的类显示出模式的情景和接口。图 10-11 有意识地使参加人的类名与它们所扮演的角色不同，以此表明它在模式中的任务是由相关中的角色名定义的。在实践中，通常都使类的名适合模式，例如，Sales 类叫做 SalesProxy，SaleStatistics 类叫做 SalesStatisticsSubject 类，等等。但是，如果已经定义了该类或它们参加了几种模式，就不能这样了。如果要对协作进行扩展，则显示出参加人在该模式中所“扮演”的角色，如图 10-12 所示。

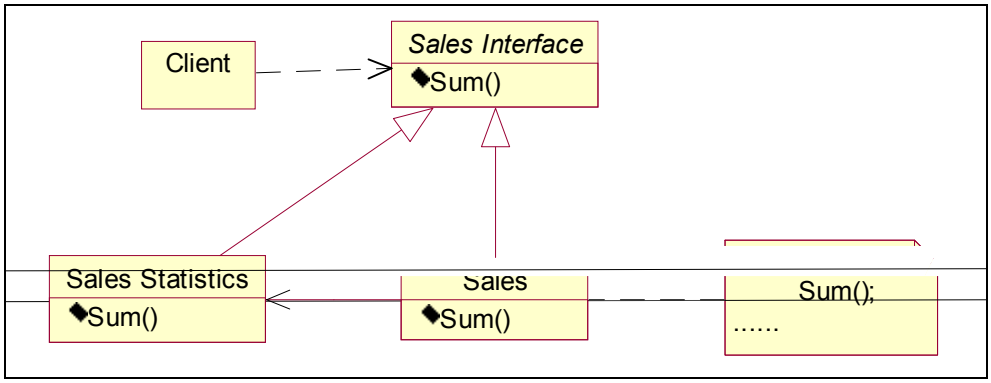


图 10-12　用图 10-11 中的参加人扩展 Proxy 模式

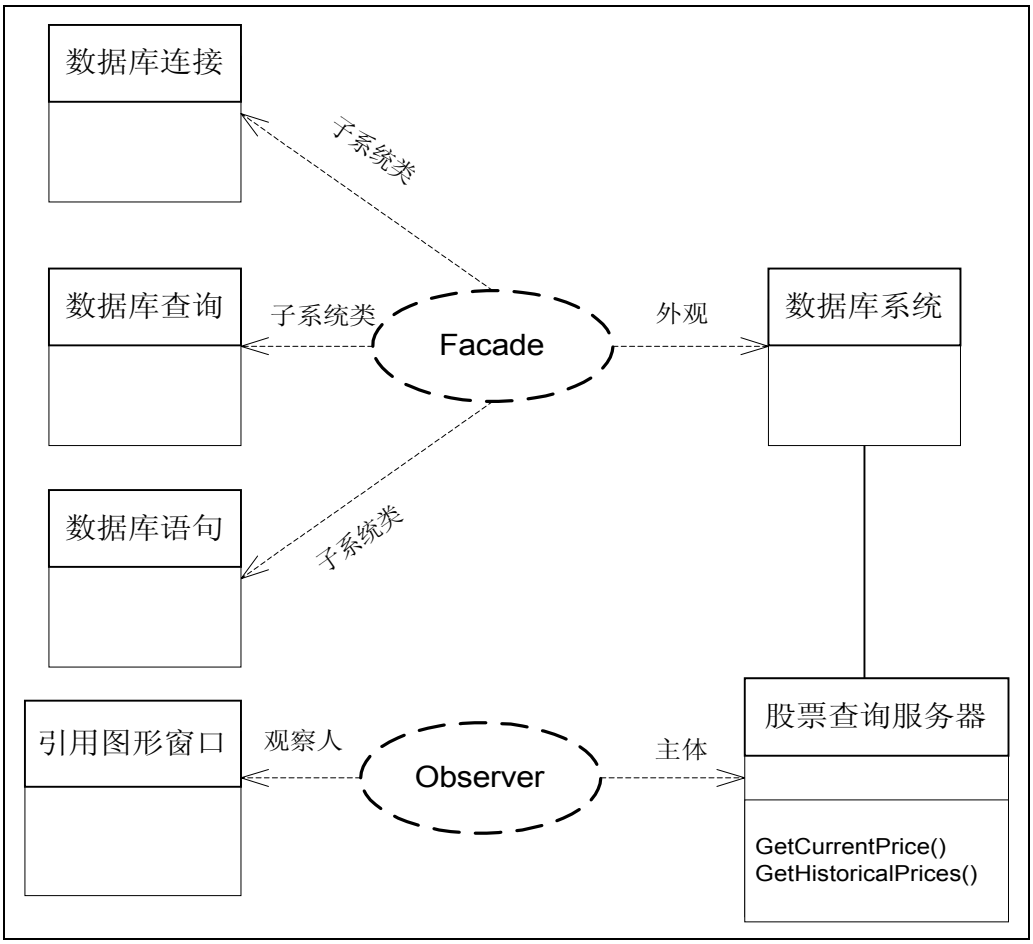


图 10-13　模式是生成子。这个类图显示了一组类，它们的部分情景和协作是用模式描述的

10.7.2　对使用模式的建议

模式被定义成参数化协作，有自己的名字，比基本元素的抽象级别高，代表了设计层的构造。在不同的设计中可以重复使用同一个模式，因为使用模式后，无需显示出系统的

所有部分，所以设计良好的模式可简化系统的描述。模式中的情景和交互是隐含的，因此可以把模式看作设计方案的通用化子，如图 10-13 所示。

在开发过程中，开发人员可以在图中扩展或隐藏模式所代表的情景和交互，用参加模式的类表示。UML 中用参数化协作来描述模式是很自然的。以下是一些使用设计模式的建议：

- 模式的名非常重要：模式的名是比模式中的设计元素抽象级别符号，设计模式的一个非常重要的性质就是能够在很高的抽象级别上进行通信和讨论。
- 确保项目中所有的人都了解模式：有的模式非常简单，很容易给开发人员造成错觉认为完全理解了模式的所有隐含意义，但往往事与愿违。所以在使用模式时一定要注重建档和训练。
- 可以改装或修改模式，但不要改变它们的基本性质：在不改变模式的基本核心的前提下，可以有很多方法来改装或修改模式。因为模式的核心通常是使用它的最根本原因，最好不要随便删除或改变。
- 强调模式是一种可重用的设计：许多机构都想把模式变换成可重用的代码（类、库和程序），这样模式就只有一种特定的实现，而其它的变形都无法使用了。应该强调，模式是在设计层面上的重用，而不是在代码层面的重用。
- 注意新模式：会有许多新模式以及模式在新领域的应用不断涌现出来，不断跟踪各种书籍、杂志、参加会议并游览互联网，将十分有利。

10.7.3 模式和用例之间的联系

当我们更深入研究模式后，会发现模式与用例的实现之间有着某种相似之处，它们都有以下几种特性：

- 情景：两者都是用具有某种结构的一个对象的网络描述的；
- 交互：关于对象如何协作都有一个主要的交互（它们在模式或用例中的行为）；
- 参加人：它们都由一组应用类“实例化”，这组类在模式或用例扮演特定的角色。

另外，在 UML 中用例的实现方式和模式的建档方式是一样的。参数化协作可以代表一个模式，也可以代表对用例实现的描述。协作的名是模式或用例的名，协作与参加的类相关。可以把模式看成一个通用的协作，多个系统都能用它，而用例是专用的，通常只能由一个系统使用（见图 10-14 所示）。

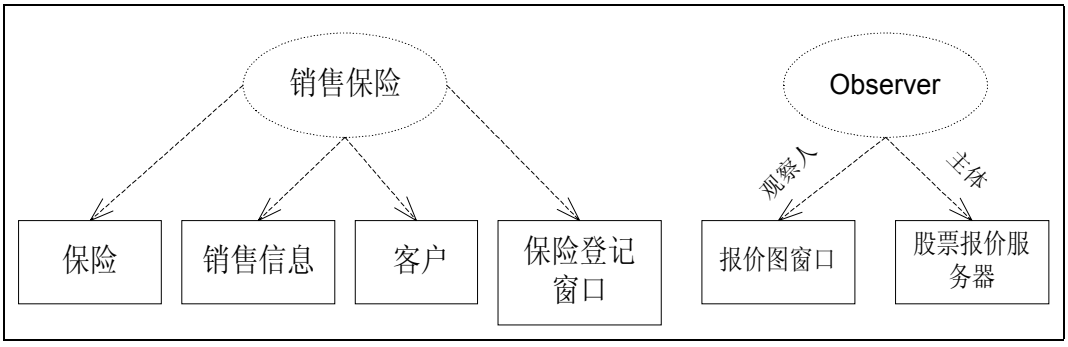


图 10-14　用例协作和模式协作，其中模式协作与参加的类相关

10.8　应用设计模式进行系统设计

模式自身是无法构成设计方法的，它是在软件开发周期的某些阶段支持设计人员进行设计的基本构件，它们可以使某些决策过程更为具体。本节我们将举例说明如何在软件的开发周期中使用设计模式。 这一节我们将讨论如何在系统设计中应用设计模式。

10.8.1　应用设计模式的主要活动

设计模式与软件开发过程有什么样的关系呢？我们可以把以下设计模式的主要活动应用在开发周期中（如图 10-15 所示）：

- 模式实例化。这是标准的模式实践：选择一个设计模式，生成设计的一部分；
- 标识候选模式。这是一个非标准的模式实践：在给定类结构中找到一个位置，为了达到设计灵活性的目的，插入一个设计模式实例。

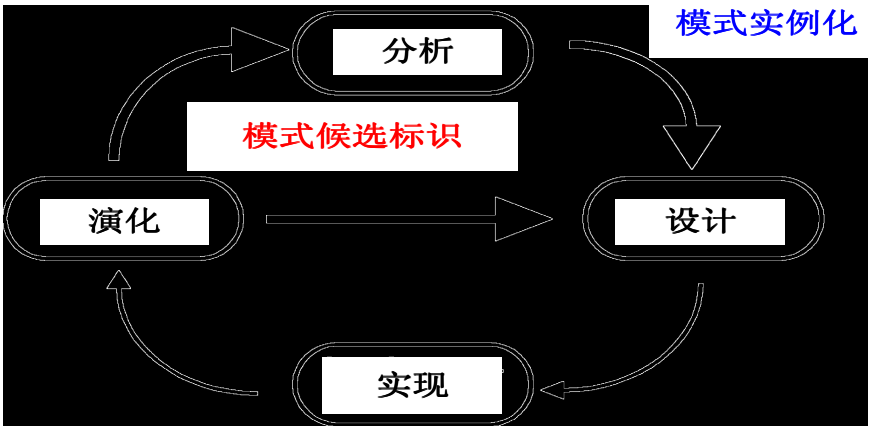


图 10-15 与模式有关的活动

10.8.2　实例化和标识模式的步骤

前面我们区分了两种主要的模式活动：（1）用模式进行设计；（2）通过模式，使设计更为灵活。 第一种称为“把模式实例化”，第二种称为在给定类结构中“标识候选模式”。这两种活动都可以划分为四个：前两个有关做决策的问题，后两个有关结构变化。注意，这些步骤并不代表设计方法——它们只是明确使用模式的认知和技术过程。

- 步骤 1：搜索和选择。 设计人员寻求一种适合的设计模式，或者在给定的设计中标识出模式候选。对模式实例化的工作是在设计的初始阶段进行的，而标识候选模式则是在原型或工作系统开始工作以后。例如，当证明某个设计组件的功能还不够时，通过在类结构中加入一个模式就可以使它变得更为灵活。这两种活动都要求设计人员对模式有着全面的了解。设计人员知道的越多，他所选择的模式就越匹配。通常，可能会几种选择，因此选择或标识的过程可能是一个做决策的问题。这时，设计模式的压力部分就起到指导的作用。
- 步骤 2：规划和分配。把模式实例化的过程将引起以下问题：在问题领域内模式

的类是什么？模式的类还应有什么其它任务？在标识模式候选的情况下，所引起的问题上：给定的类、操作和属性在模式中担任的是什么角色？

- 步骤 3：过滤。把一个设计模式实例化时，有可能会改变它的原始结构。有时改变或去掉某些类会减少模式原来的灵活性（下面的例子中会有显示）。但是只要给模式实例编制恰当的文档，就可以在必要的时候恢复原来的灵活性。当在一给定的设计中插入一个模式时，也可能会需要改变类的接口或增加新的类。因此将由模式的抽象类来保持所需要的灵活性。
- 步骤 4：细化。最后，由技术类完成设计。它们还会增加任何新的语义，但把有关问题领域的考虑和技术问题以及编码问题分开。 另外还可能需要对类的接口做一些扩展，这些都完全由设计人员来决定了。

10.9 模式选择举例：评估项目

该项目的目标是开发一个面向对象的接口，使 SAP-R/3 商业对象知识库和 Apple/IBM 的开放脚本架构（OSA）之间可进行互操作。OSA 与 Microsoft 的 OLE 自动机类似。 商业对象知识库是对商业对象的管理单元，主要由 SAP 商业工作流程使用。商业对象允许对 R/3 数据进行面向对象的接入和修改。我们将以该项目中的“存储商业对象类型”和图 10-16 中显示的“过程控制”作为例子。

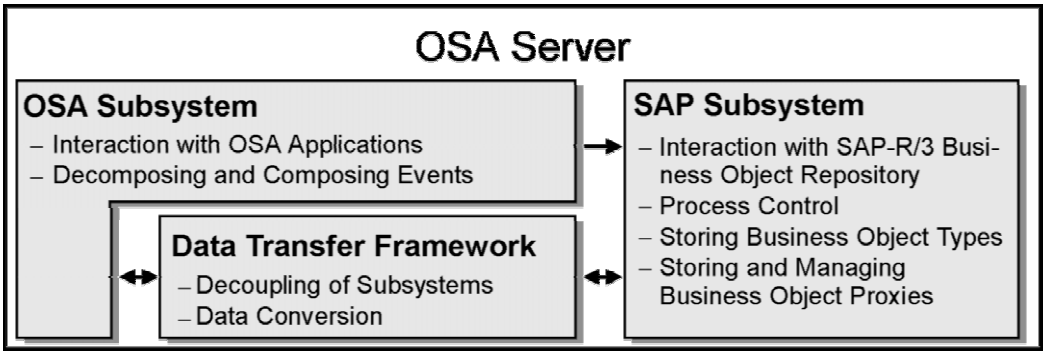


图 10-16 OSA 服务器的组件

10.9.1 实例化模式：“存储商业对象类型”模式

情景：SAP 商业对象的类型是在商业对象知识库中定义和维护的。这种类的接口由一组属性和操作组成，而操作由一组参数组成。

问题：开发一个单元来维护和保存类的接口。

解决：根据数据的分层结构，复合模式是一种可能的选择。对该模式一种直接的实例化如图 10-18 所示，原始的复合模式结构如图 10-17 所示。

压力：考虑情景，我们修改第一个选择：把属性和参数分开没有必要。因此把复合模式转换成设计，继续考察模式的结构时，发现根本没有叶类。因此，把复合类从抽象类变成具体类。这样做虽然为了简单而降低了复合模式的灵活性，但是如果需求发生变化时，

我们可以简单地再加上叶类恢复模式的灵活性。 最后，模式就可以工作了。而保存参数等技术任务就落到类模板 *List* 和 *Iterator* 上了（步骤 4）。

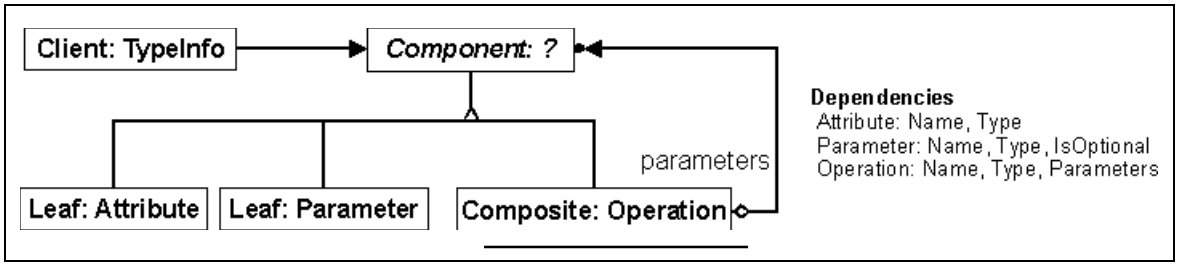


图 10-17　首先尝试 Composite 模式

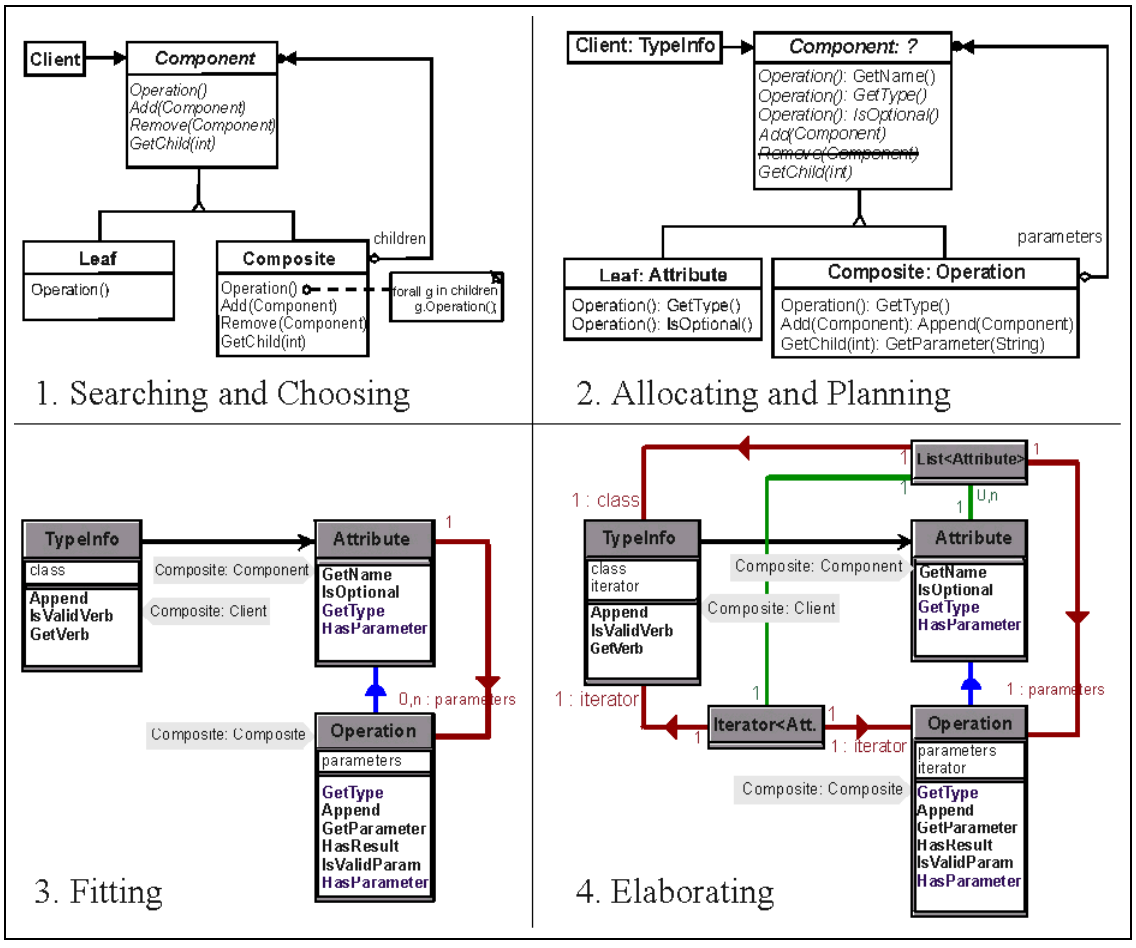


图 10-18　实例化 Composite 模式的步骤

10.9.2　标识模式候选：“过程控制”的例子

减弱子系统之间的相互影响是设计 OSA 的重要目标，指定与 SAP R/3 和 OSA 交互的子系统被严格分离开来，目的是保证具有互操作性接口相互之间易于替换。这种消弱的目标对数据流和控制流都做到了。在本例中，我们将围绕控制流进行讨论，也就是对 OSA

事件（OSADispatch）的响应。图 10-19 是设计的一部分，代表了一般问题：在事件的接收方和表示 SAP 系统的 BOR 组件的类之间是如果维护控制流的？下面就是需求的动机：

- 来自快速原型的早期反馈：放松子系统之间的耦合；BOR 之间的过程控制；
- 分析阶段推迟的需求：认为不能改变 BOR 类所代表的 SAP 子系统。OSADispatch 应该很容易被其它有互操作性的接口如 OLE 等所取代；
- 对分析阶段的需求进行扩展：OSADispatch 和 BOR 之间一对多的关系，使几个 SAP 系统可以被一个 OSA 事件同时找到。

这些需求的目的都是通过加入更多的灵活性来提高设计的质量，从这个出发点，我们对现有一些模式进行分析，找出最适合的一个：

- Facade 封装了子系统，并定义了一个通用化的接口，使子系统更容易处理；
- Adapter 把一个类的接口转换成客户需要的；
- Chain of Responsibility 把一组接收对象链接到一个发送对象上。请求将沿着接收方链传递，直到有对象处理它；
- Observer 定义了一个“主体（Subject）”和一个或多个“观察人（Observers）”之间的一对多的相关，确保当主体发生变化时，所有的观察人都得到通报。

选择模式是应用模式过程中最重要的一步，因为后面所有对设计的改变以及最后的质量都依赖于它。设计员必须依靠他对设计模式的知识以及他对问题需求的理解作出正确的选择，而对模式的深入理解必须经过几次应用模式后才能得到。图 10-19 的下面部分代表了 Observer 模式的核心：

- 把组件的接口分成两部分：一部分是不可改变的抽象耦合（上面部分），以及从 ConcreteObserver 到 ConcreteSubject 的特定请求（下面部分）；
- 主体和它的观察人之间一对多的关系；
- 从直觉上看，整个过程是按照顺时针进行的：AttachTo()-> Action()-> Notify() -> Update() -> GetState()。

现在来分析一下我们所做的决定：Facade 和 Adapter 不适合我们的情况，因为它们主要是结构模式，OSADispatch 和 BOR 之间控制流的行为方面是最重要的。Chain of Responsibility 也不适合，因为这个模式的本质在于对象具有沿着它们的类层次传递请求的功能。把一个子系统的类组织成一个有继承关系的层次从语义上讲是没有道理的——只是在设计插入一个模式——没有任何意义。最后， Observer 模式看起来最适合，下面将讨论如何把它转换到设计中。

我们首先研究一下现有的设计（图 10-19 上面部分）：在接收到一个事件后，OSADispatch 类把请求传给 BOR 接口。因此，BOR 起到服务器的作用，而 OSADispatch 则是客户。但这个方法的缺点很快就体现出来，因为由 OSADispatch 负责控制通信和在几个 SAP 系统中进行选择。 通过改变客户和服务器的角色，就能使设计更为灵活。因此，把 OSA 服务器内的通信流逆转，如图 10-20 所示。现在 OSADispatch 的功能就成了服务器。当接收到一个事件后，它只是把通知的消息转发（Notify()）， 告诉说发生了改变。所有连接到主体的 SAP 子系统都得到更新的消息，并自己决定进行什么响应。然后，OSADispatch 会发一些其它请求（GetCommand()）。

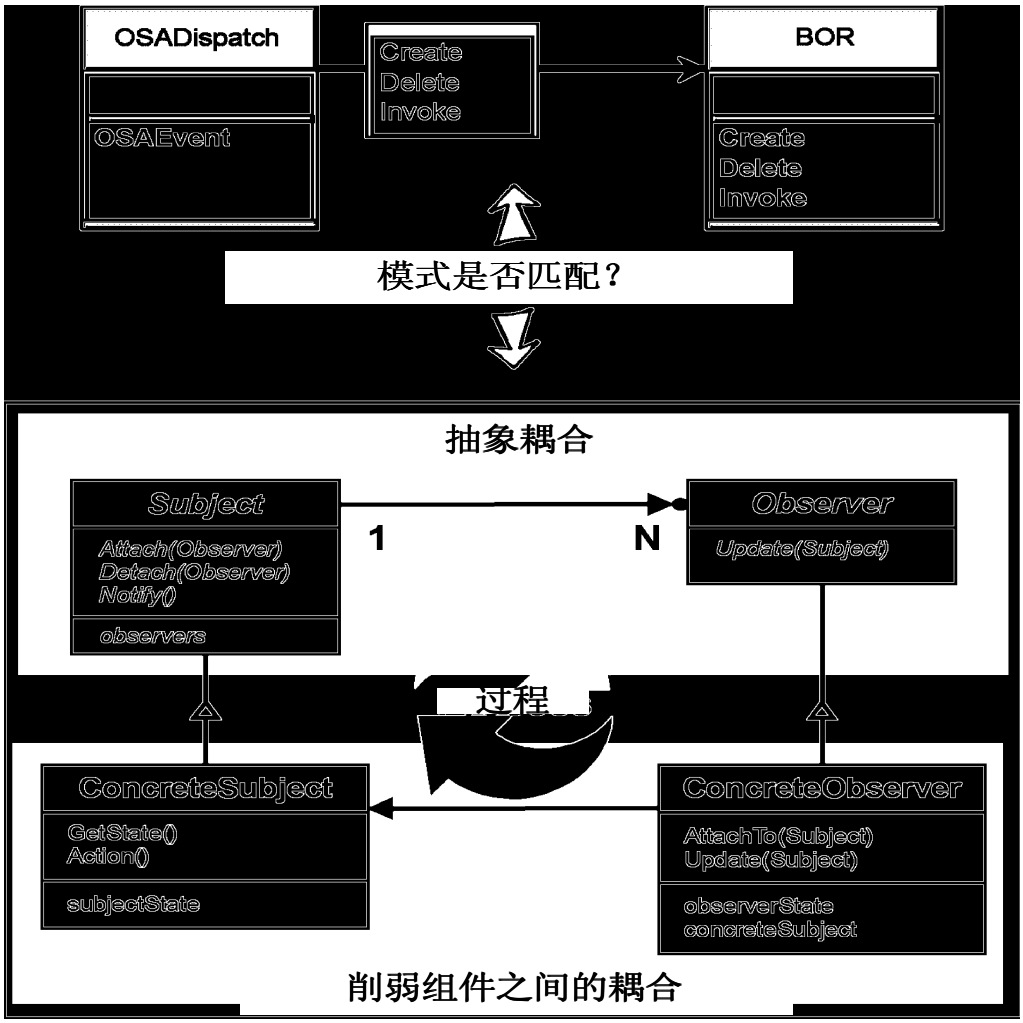


图 10-19　步骤 1：搜索正确的匹配

在步骤 2 中（图 10-20），很明显 BOR 类的整个接口，即 Create(), Delete(), and Invoke(), 不能给它们分配模式的功能。结果是，我们把 BOR 的接口变成如图 10-21 所示，这时客户就不能调用 BOR 以前的函数了，它们成为保护成员函数。它自己会通过调用 Update() 进行控制。最后，在步骤 4（图 10-21），我们还插入了技术类，如 List<Observer> 和 Iterator<Observer>。

为了使 OSADispatch 更容易替换的，Observer 模式的实例化只能作为一个框架实现。为此，将插入一个抽象消息调度员（AbstractDispatch），来定义 BOR 和 AbstractDispatch 之间的通信协议。在这之间通信是由 `GetCommand()` 处理的。具体的调度员如 OSADispatch 将从主体和 AbstractDispatch 派生而来。可应用 Adapter 模式把通信协议翻译过来。这里的描述清楚地表明，原来的方法变得灵活了，同时责任也发生了转移：映射出向框架开发的平滑过渡（图 10-22）。

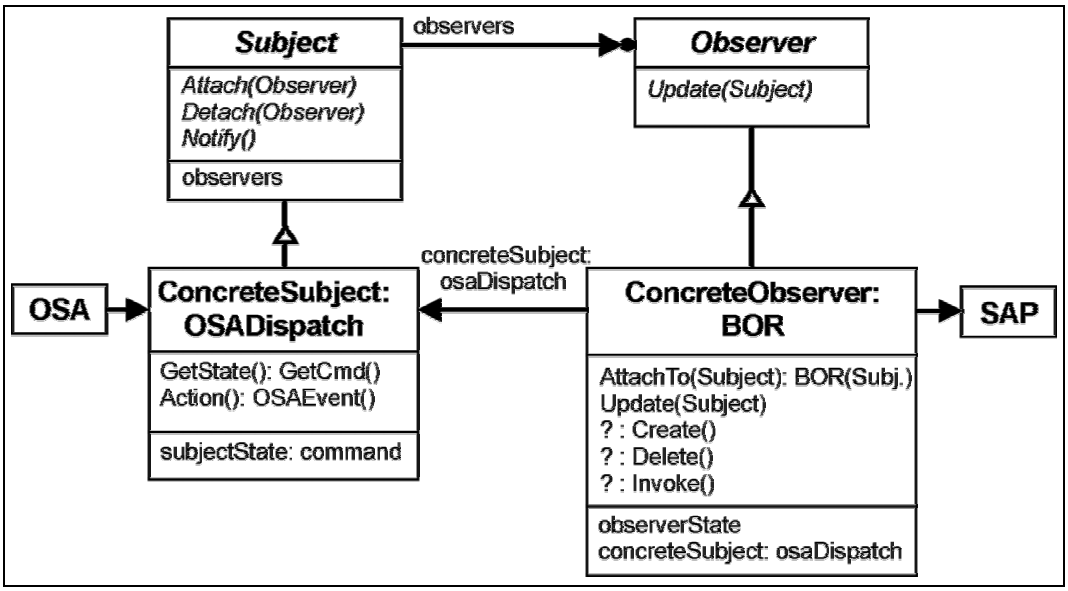


图 10-20　步骤 2：分配和规划

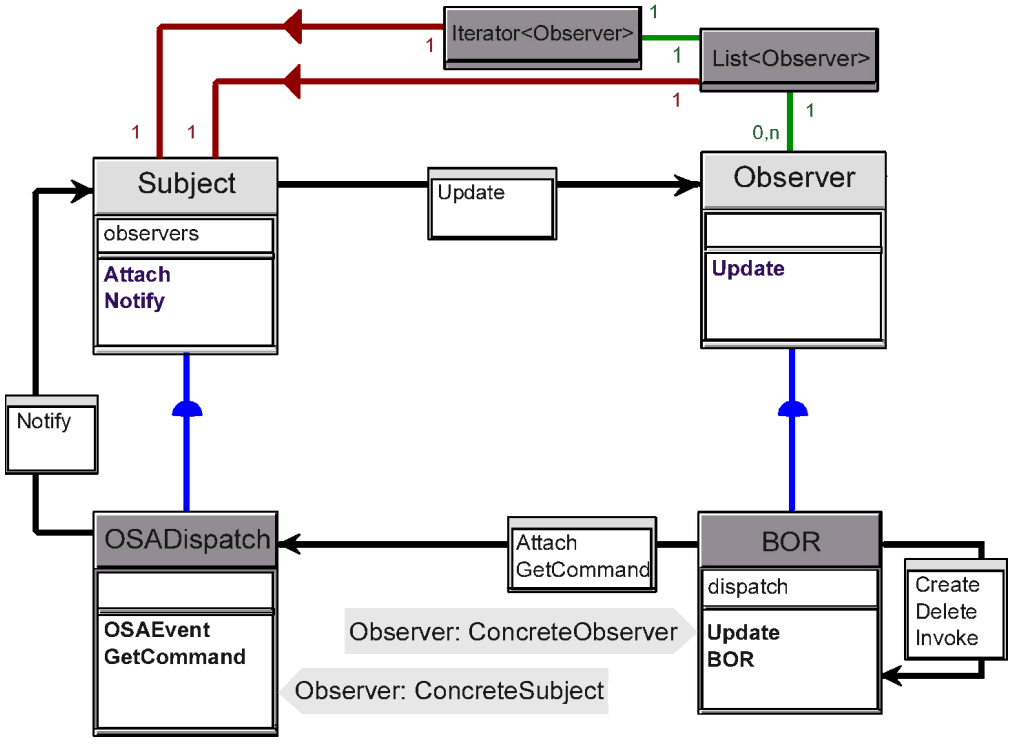


图 10-21　步骤 3 和步骤 4：过滤和细化

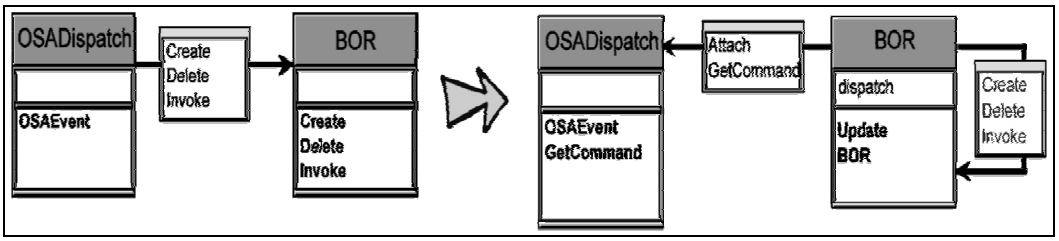


图 10-22　角色的变化：无缝地过渡到框架设计

10.10　模式应用举例：形状编辑器

Johnson，Helm，Gamma 和 Vlissides 把设计模式分成三类： 行为、结构和创建的。在这一节中，我们以一个作图工具为例，分别举出这三种模式的例子，包括它们相应的 Java 程序，来说明设计模式的应用。首先我们看一下这个工具的关键需求，然后讨论如何用设计模式实现这些特殊需求。它们是：

- 同一个图的多种视图
- 删除（Delete）、取消删除（Undelete）和重做（Redo）
- 用户可定义的复杂的复合形状
- 形状选择
- 使编辑器可扩展

下面我们就讨论如何用模式来实现这些需求。

10.10.1　同一个图的多个视图

任何作图工具都应该具备的一个功能就是允许一个模型有多个视图。比如表格工具 Excel 就允许对同一组数据，有一个饼图视图，一个条形图视图和一个表格视图。并且如果数据更新的话，那么这些视图都会自动更新。在我们的作图工具中，我们也希望能用一个以上的视图来描述底层模型。如果对其中一个视图的改变能自动地体现在其它视图中，这样后来我们就能在 Diagram 类中加上新的视图，但 Diagram 类必须能够知道这些视图，从而当其中一个改变时，它可以更新其它视图。

为这一问题提供解决方案的模式恐怕是最老的模式了，这就是 Apple 称之为模型/视图/控制器的模式，而在 Microsoft 的 MFC 中是模型/视图，而在 Java 中是观察员/可观察的接口/类对。Gamma 等称之为观察员模式。本节我们就用 Java 的术语 Observer/Observable。

Observer/Observable 模式类是从 Observer 派生而来的，负责描述模型，通过调用 addObserver()注册，并把它们自己加到 Observers 的清单中，这个清单是由 Observable 派生而来的类在其内部维护的。当 Observable 对象中的模型改变时，它对 Observers 清单中的每一项，调用每个 Observer 的 onUpdate()操作。Observer 派生类为了响应对它的 onUpdate()操作的调用，查询模型（Observable 派生的类），并且更新它的显示。 很明显，Observer 派生类知道 Observable 派生类——但 Observable 派生类只需要知道它的 Observers 是

Observer 类型的，并且还定义了操作 onUpdate()。

在我们的作图工具中，Diagram 是 Observable 类，如图 10-23 所示，DrawShapes 类是唯一的 Observer 类（Diagram 和 DrawShapes 代码分别见图 10-25 和 10-24 所示）。那为什么还要用模式呢？答案就是为了可扩展性。

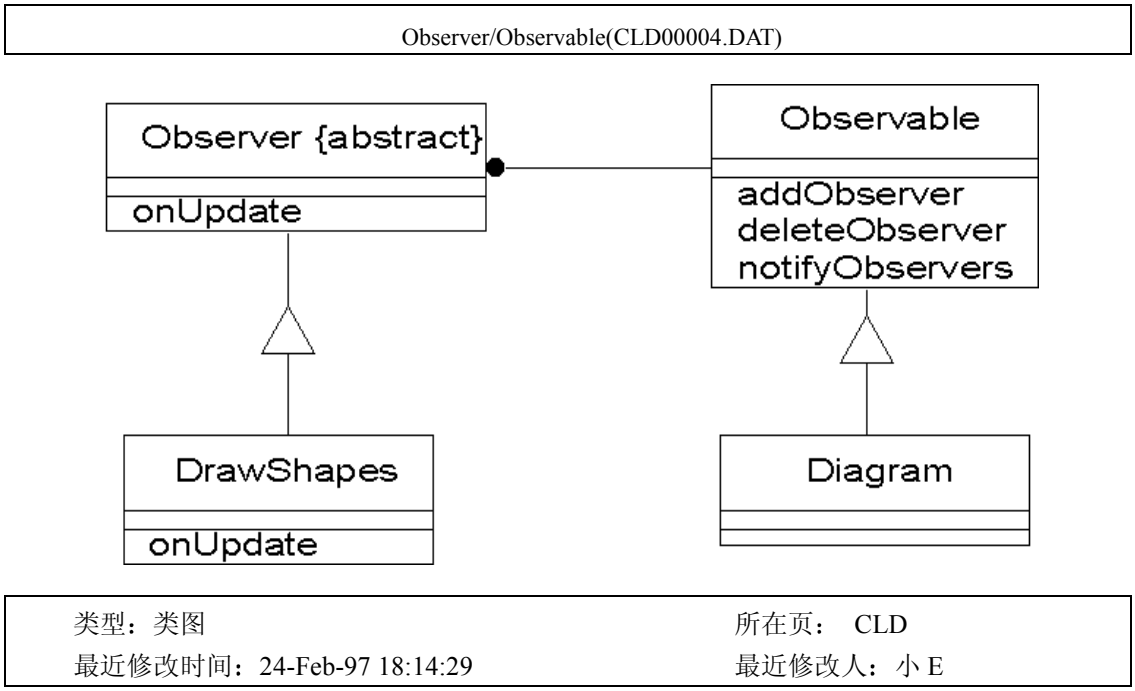


图 10-23　Observer/Observable（类图）

```
    //**************************************************************************
*
    // DrawShapes.java: Applet
    //
    //**************************************************************************
*
    import java.applet.*;
    import java.awt.*;
    import java.util.Observer;
    import java.util.Observable;
    import DrawShapesFrame;
    import MainMenu;
    import ShapeFactory;
    import Diagram;
    //========================================================================
=
    // Main Class for applet DrawShapes
    //
```

```
    //=============================================================================
=
    public class DrawShapes extends Applet implements Observer
    {
    // The diagram that contains all the shapes in the diagram...
    static Diagram m_Diagram = new Diagram();
    public static void main(String args[])
    {
    // Initialise the ShapeFactory
    ShapeFactory.Initialise();
    // Create Toplevel Window to contain applet DrawShapes
    //-----------------------------------------------------------------------
    DrawShapesFrame frame = new DrawShapesFrame("DrawShapes");
    frame.show();
    frame.hide()
    frame.resize(frame.insets().left + frame.insets().right + 320,
    frame.insets().top + frame.insets().bottom + 240);
    DrawShapes applet_DrawShapes = new DrawShapes();
    frame.add("Center", applet_DrawShapes);
    applet_DrawShapes.m_fStandAlone = true;
    applet_DrawShapes.init();
    applet_DrawShapes.start();
    // Setup the main menu bar
    MainMenu menu = new MainMenu(frame);
    menu.CreateMenu();
    // Initialse the shape menu with all the shapes
    // in the factory.
    int i = 0;
    while(ShapeFactory.NumberOfAvailableShapes() > i)
    {
    menu.AddShapeToInsertMenu(ShapeFactory.GetShapeAt(i));
    i++;
    }
    frame.show();
    }
    public void init()
    {
    resize(320, 240);
    // Register as an observer with the Diagram
    m_Diagram.addObserver(this);
    }
    // DrawShapes Paint Handler
    //-----------------------------------------------------------------------
```

```
    public void paint(Graphics g)
    {
    int i = 0;
    while (m_Diagram.NumberOfShapes() > i)
    {
    ((Shape)m_Diagram.GetShapeAt(i)).Draw(g);
    i++;
    }
    }
    // MOUSE SUPPORT:
    // The mouseDown() method is called if the mouse button is pressed
    // while the mouse cursor is over the applet's portion of the screen.
    //--------------------------------------------------------------------------
    public boolean mouseDown(Event evt, int x, int y)
    {
    if (evt.clickCount > 1 && ShapeFactory.NumberOfAvailableShapes() > 0)
    // Double click
    {
    try
    {
    Shape    newShape    =    ShapeFactory.BuildShape(new    Dimension(x,y),    new
Dimension(100,100));
    m_Diagram.AddShape(newShape);
    // Shouldn't need this. It seems MS's
    // observer/observable is a pile of cack.
    update(m_Diagram, newShape);
    }
    catch (Exception e)
    {
    // Throw the exception away
    }
    }
    return true;
    }
    // Handles updates from the Diagram
    public void update(Observable o, Object a)
    {
    // force the whole screen to be re-drawn in response
    // to the diagram changing. Easy but inefficient.
    repaint();
    }
    }
```

图10-1　代码 DrawShapes.java

```
    //*************************************************************************
*
    import java.util.Observable;
    import java.util.Vector;
    /*
    * Diagram
    * Contains all the shapes in the current diagram.
    */
    class Diagram extends Observable
    {
    public int NumberOfShapes()
    {
    return m_AllShapes.size();
    }
    public Shape GetShapeAt(int i)
    {
    return (Shape)m_AllShapes.elementAt(i);
    }
    public void AddShape(Shape newShape)
    {
    m_AllShapes.addElement(newShape);
    notifyObservers(newShape);
    }
    public Shape RemoveShapeAt(int i)
    {
    Shape s = (Shape)m_AllShapes.elementAt(i);
    m_AllShapes.removeElementAt(i);
    return s;
    }
    // The list of all shapes in the diagram
    Vector m_AllShapes = new Vector();
    }
```

图 10-25　代码 Diagram.java

10.10.2　删除、取消删除和重做

现在对任何编辑器都有一个最基本的要求，那就是能够删除，取消删除和重做。有一种设计模式就解决了这一个问题，如图 10-26 所示。事实上命令模式描述的功能比我们对编辑器的要求还多。

简单讲（具体内容见代码），当如图 10-26 和 10-27 所示的 DeleteShape 命令被创建时，就告诉了它从哪个 Diagram 中删除哪个 Shape，以及当删除该 Shape 时，什么时候调用它的 Do()方法。同时它还保存了足够的信息，确保一旦它的 UnDo()操作被调用，就重做该删除。ReDo()也是类似的。同时它也保持一个内部状态，例如它不会取消一个它还没有执

行的删除。

我们的 DeleteShape 命令相对简单——我们只支持一级取消操作。但我们可以通过两种方式来增加这个命令的复杂性。首先，只要简单维护一个删除的历史记录，只要当每个 DeleteShape 对象的 Do()方法被调用时，就把该对象放到一个后进先出的表中，从而提供 n 级取消操作。其次，我们可以增加 Undo()方法的复杂性。

在更复杂的模型作图中，重复的删除和取消删除的操作增加了出错的程度，而此时取消删除不再是简单地把被删除的项重新插入的问题，而我们的方法也不同适用。在这种情况下，Command 对象需要记住模型的内部状态，或者至少状态的子集，允许完全恢复模型。很明显，这破坏了面向对象的封装规则——因为 Command 对象需要知道模型的内部实现！幸运的是，有一种设计模型解决了这个问题。Memento 是一个黑箱，只能由创建它的对象访问（用 C++的话说，Memento 只有私有成员，它的创建者是它的友类），并且封装了它的创建者足够的内部信息，确保创建者能够完成恢复到它创建 Memento 时的状态。

如果我们要修改编辑器，让它在进行删除时使用 Memento 模式（而不是简单记录把哪个 Shape 从哪个 Diagram 中删除），然后下面就是当 DeleteShape 的 Do()操作被调用时发生的事件顺序：

1. 从 Diagram 得到一个 Memento，并保存起来；

2. 从图中把 Shape 删除并移走。

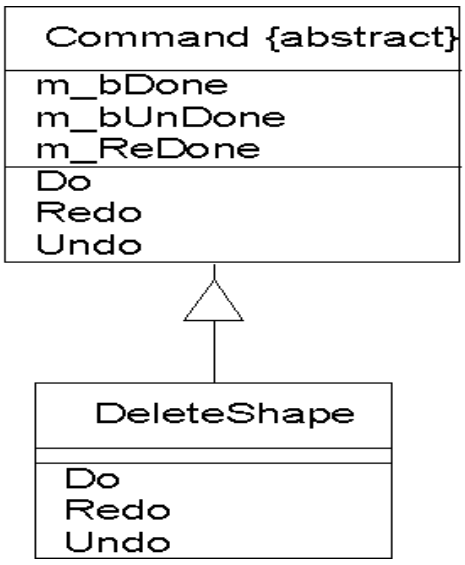
而当 DeleteShape 的 Undo()操作被调用时：

1. DeleteShape 首先检查是否它的 Do()操作已经被调用（显然，不能取消那些还没有做的操作）；

2. 然后它把保存的 Memento 传回给 Diagram，Diagram 恢复到它创建 Memento 时的状态。

注意，在 Diagram 类中也需要做一些改变（提供对 Mementos 的支持）来实现这个更复杂的操作/取消操作/重做（Do/UnDo/ReDo）。

Command(CLD00003.DAT)



类型：类图	所在页： CLD
最近修改时间：20-Feb-97 18:14:29	最近修改人：小 E

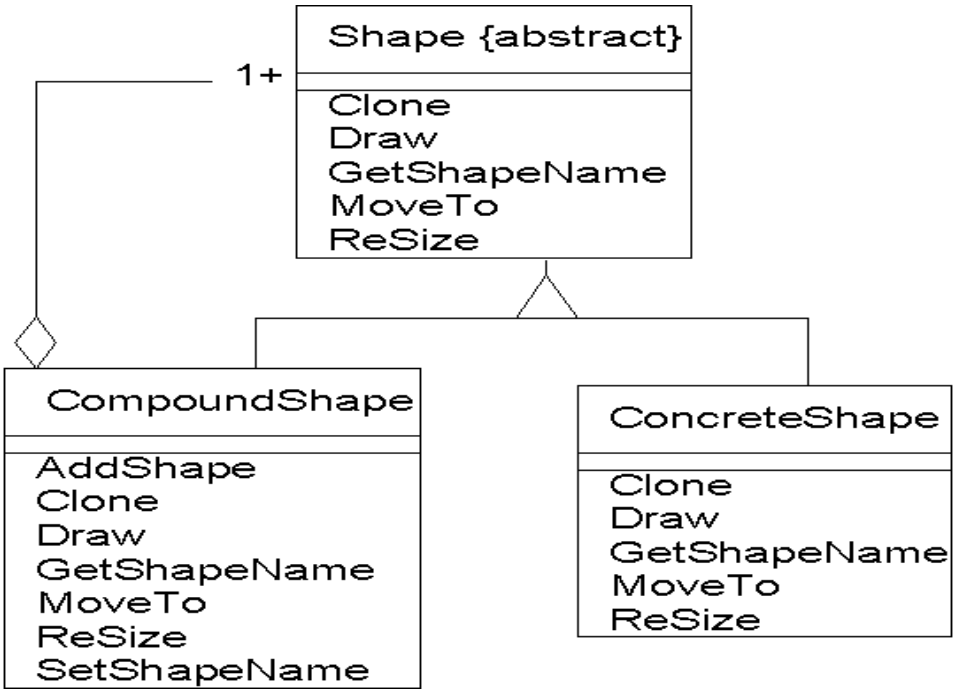
图 10-26　命令（Command）（类图）

10.10.3　用户可定义的复杂的复合形状

在我们的工具中，我们已经说明需要能够把一些形状组合在一起，并把它们的复合形状作为一个 Shape 来对待，从而可以把它作为一个整体，移动它或重新定义它的大小。而这个复合形状又可以用来组成一个或多个其它复合形状等等。解决这个问题的设计模式是复合模式，如图 10-27 所示。

复合模式定义了一种方式，使我们可以同样的方式处理个体部分或聚合的部分。Shape 抽象类，如图 10-27 和 10-28 所示，定义了原子形状（ConcreteShape 和它的子类）和复合形状（CompoundShape 子类）。在这两种情况下，由子类负责实现 Draw()和 clone()操作。在 ConcreteShape 子类的情况下，这直接由类实现；在 CompoundShape 类的情况下，这些方法的实现是通过重复调用和组件形状的 Draw()或 Clone()方法来完成的。如果组件形状中有的自身就是一个 CompoundShape，那么它调用它自己组件的 Draw （或 clone）方法，等等。

Composite(CLD00002.DAT)



类型：类图	所在页： CLD
最近修改时间：18-Feb-97 18:14:29	最近修改人：小 E

图 10-27　复合（Composition）（类图）

```
    //*************************************************************************
*
    import java.lang.Object;
    import java.util.Vector;
    import java.awt.*;
    /*
    *
    * Shape
    *
    * This is the basic shape class.
    *
    */
    abstract class Shape extends Object implements Cloneable
    {
    protected Shape(Dimension TopLeft, Dimension Size)
    {
    m_Dimension = Size;
    m_TopLeft = TopLeft;
    }
    public abstract void Draw(Graphics g);
    public void ReSize(int iNewWidth, int iNewHeight)
    {
    m_Dimension = new Dimension(iNewWidth, iNewHeight);
    }
    public void ReSize(Dimension dimNewDimension)
    {
    m_Dimension = dimNewDimension;
    }
    public void MoveTo(int iNewLeft, int iNewTop)
    {
    m_TopLeft = new Dimension(iNewLeft, iNewTop);
    }
    public void MoveTo(Dimension dimNewTopLeft)
    {
    m_TopLeft = dimNewTopLeft;
    }
    public Dimension GetTopLeft()
    {
    return m_TopLeft;
    }
    public Dimension GetDimension()
    {
    return m_Dimension;
```

```
}
public abstract String GetShapeName();
// Position and size of the shape
protected Dimension m_Dimension; // size
protected Dimension m_TopLeft; // position
}

/*
* CompoundShape
*
* This class represents a shape made up of a collection
* of Shapes - which maybe basic shapes or other compound shapes.
*
*/
class CompoundShape extends Shape
{
public CompoundShape (String strShapeName, Dimension TopLeft, Dimension Size)
{
super(TopLeft, Size);
m_ShapeName = strShapeName;
}
public void Draw(Graphics g)
{
// for each shape in the list of shapes that
// make up this CompoundShape invoke its Draw
// method.
int i = 0;
while (m_ComponentShapes.size() > i)
{
((Shape)(m_ComponentShapes.elementAt(i))).Draw(g);
i++;
}
}
public void AddShape(Shape NewShape)
{
m_ComponentShapes.addElement(NewShape);
}
public String GetShapeName()
{
return m_ShapeName;
}

// Clone is part of implementing the Prototype design pattern
```

```
    public Shape clone() throws CloneNotSupportedException
    {
    CompoundShape NewShape = new CompoundShape(m_ShapeName, GetTopLeft(),
GetDimension());
    // for each shape in the list of shapes that
    // make up this CompoundShape invoke its Clone
    // and add it into the new CompoundShape.
    int i = 0;
    while (m_ComponentShapes.size() > i)
    {
    Shape s = ((Shape)(m_ComponentShapes.elementAt(i)));
    NewShape.AddShape((Shape)s.clone());
    i++;
    }
    return NewShape;
    }
    // The component shapes that makeup this ComponentShape
    private Vector m_ComponentShapes;
    String m_ShapeName;
    }


    // The concrete shapes
    abstract class ConcreteShape extends Shape
    {
    protected ConcreteShape(Dimension TopLeft, Dimension Size)
    {
    super(TopLeft, Size);
    }
    }
    class Square extends ConcreteShape
    {
    protected Square(Dimension TopLeft, Dimension Size)
    {
    super(TopLeft, Size);
    }
    public Square()
    {
    super(new Dimension(0,0), new Dimension(0,0));
    }
    public void Draw(Graphics g)
    {
    g.drawLine(m_TopLeft.width, m_TopLeft.height, m_TopLeft.width + m_Dimension.width,
```

```
m_TopLeft.height);
    g.drawLine(m_TopLeft.width, m_TopLeft.height, m_TopLeft.width, m_TopLeft.height +
m_Dimension.height);
    g.drawLine(m_TopLeft.width    +    m_Dimension.width,    m_TopLeft.height    +
m_Dimension.height, m_TopLeft.width + m_Dimension.width, m_TopLeft.height);
    g.drawLine(m_TopLeft.width    +    m_Dimension.width,    m_TopLeft.height    +
m_Dimension.height, m_TopLeft.width, m_TopLeft.height + m_Dimension.height);
    }
    public String GetShapeName()
    {
    return new String("Square");
    }
    // Clone is part of implementing the Prototype design pattern
    public ConcreteShape clone()
    {
    return new Square(GetTopLeft(), GetDimension());
    }
    }

    class Circle extends ConcreteShape
    {
    protected Circle(Dimension TopLeft, Dimension Size)
    {
    super(TopLeft, Size);
    }
    public Circle()
    {
    super(new Dimension(0,0), new Dimension(0,0));
    }
    public void Draw(Graphics g)
    {
    g.drawOval(m_TopLeft.width,         m_TopLeft.height,         m_Dimension.width,
m_Dimension.height);
    }
    public String GetShapeName()
    {
    return new String("Circle");
    }
    // Clone is part of implementing the Prototype design pattern
    public ConcreteShape clone()
    {
    return new Circle(GetTopLeft(), GetDimension());
    }
```

```
    }
    class Star extends ConcreteShape
    {
    protected Star(Dimension TopLeft, Dimension Size)
    {
    super(TopLeft, Size);
    }
    public Star()
    {
    super(new Dimension(0,0), new Dimension(0,0));
    }
    public void Draw(Graphics g)
    {
    // Draw a five pointed star
    Dimension Point1 = m_TopLeft;
    Dimension Point2 = new Dimension(m_TopLeft.width + m_Dimension.width/2,
m_TopLeft.height + m_Dimension.height);
    Dimension Point3 = new Dimension(m_TopLeft.width + m_Dimension.width,
m_TopLeft.height);
    Dimension Point4 = new Dimension(m_TopLeft.width, m_TopLeft.height +
3*m_Dimension.height/4);
    Dimension Point5 = new Dimension(m_TopLeft.width + m_Dimension.width,
m_TopLeft.height + 3*m_Dimension.height/4);
    g.drawLine(Point1.width, Point1.height, Point2.width, Point2.height);
    g.drawLine(Point2.width, Point2.height, Point3.width, Point3.height);
    g.drawLine(Point3.width, Point3.height, Point4.width, Point4.height);
    g.drawLine(Point4.width, Point4.height, Point5.width, Point5.height);
    g.drawLine(Point5.width, Point5.height, Point1.width, Point1.height);
    }
    public String GetShapeName()
    {
    return new String("Star");
    }
    // Clone is part of implementing the Prototype design pattern
    public ConcreteShape clone()
    {
    return new Star(GetTopLeft(), GetDimension());
    }
    }
```

图 10-28　代码 Shape.java

10.10.4　形状选择

Shape 类也是原型模式的例子，从它的方法 clone()可以很明显地看出来。clone()操作

是为 Java 类 Object 定义的，我们将说明我们的类通过实现 Cloneable 接口实现了它，从而使编辑器具有两大功能：首先我们可以建立 ShapeFactory 类，限制 ConcreteShapes 的知识，使得即使是 Factory Method 如图 10-29，也不知道到底把 Shape 成什么类型——它只是要求一给定的原型实例，复制它自己并返回创建的 Shape。其次，它允许我们处理由用户根据 ConcreteShapes 定义的新的形状以及任何预定义的形状。它是如何做到的呢？当用户创建一个 CompoundShape 对象后就用 ShapeFactory 注册它，从而就可以使用它了。新的 CompoundShape 就自动可以使用了，只要选择“当前”形状。ShapeFactory 也具有原型管理员的作用，因为它有一个可用形状的盖章，包括用户定义并注册的 CompositeShape。

这是一个通过复合而不是继承来创建新类型的例子。在继承之上进行聚合的好处是，继承是静态的，而聚合是动态的。

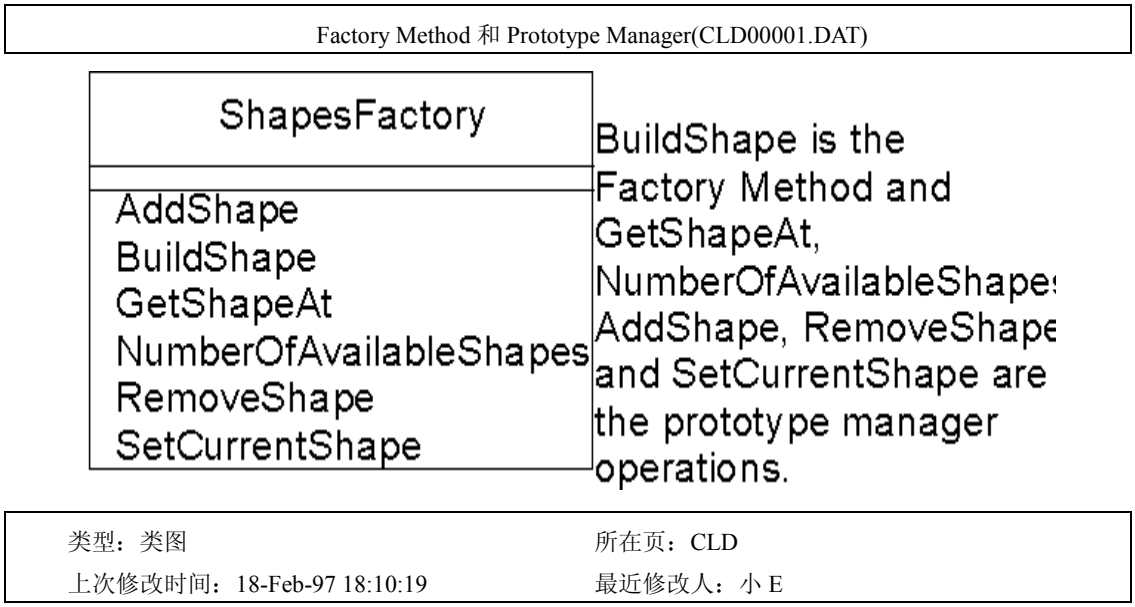


图 10-29　Factory Method（类图）

```
import java.util.Vector;
import java.lang.Exception;
import java.awt.Dimension;
import AbstractFactory;
/*
*
* ShapeFactory
*
*
* This factory holds all the available ConcreteShapes that
* that can be drawn and the "currently selected" or "enabled" shape
* ie the one that is built if the factory is asked
* to build a shape.
*/
class ShapeFactory extends AbstractFactory
```

```
    {
    static Vector m_AllAvailableShapes = new Vector();
    static int m_iCurrentShapeIndex = 0;
    private ShapeFactory() {} // disable the default ctor
    static public void Initialise()
    {
    // setup the list of basic shapes
    AddShape(new Circle());
    AddShape(new Square());
    AddShape(new Star());
    }
    static private void AddShape(Shape NewShape)
    {
    m_AllAvailableShapes.addElement(NewShape);
    }
    static public Shape BuildShape(Dimension TopLeft, Dimension Size) throws
Exception
    {
    if (m_AllAvailableShapes.size() != 0)
    {
    // uses the prototype to create the shape
    return
(Shape)(((Shape)m_AllAvailableShapes.elementAt(m_iCurrentShapeIndex)).clone())
;
    }
    else
    {
    throw new Exception("ShapeFactory::BuildShape() No shapes available or
selected.");
    }
    }
    static public void SetCurrentShape(String ShapeName) throws Exception
    {
    int i = 0;
    while (m_AllAvailableShapes.size() > i)
    {
    if (((Shape)(m_AllAvailableShapes.elementAt(i))).GetShapeName() == ShapeName)
    {
    // Shape found so make it the current one
    m_iCurrentShapeIndex = i;
    return;
    }
    i++;
```

```
}
throw new Exception("ShapeFactory::Shape '" + ShapeName + "' Not in catalogue.");
}
static public int NumberOfAvailableShapes()
{
int iNumberOfShapes = m_AllAvailableShapes.size();
return iNumberOfShapes;
}
static public Shape GetShapeAt(int i)
{
return (Shape)m_AllAvailableShapes.elementAt(i);
}
}
```

图 10-30　代码 ShapeFactory.java

很明显，应该只有一个 ShapeFactory，如图 10-30，那么如何用模型体现出来呢？答案是使用模式 Singleton。在 Java（和 C++）中，这是通过类成员来实现的——也就是说，通过静态方法和属性实现的。ShapeFactory 只有静态成员，因此会有一个 ShapeFactory 自动实例类——而还需要对它进行实例化。事实上，不可能明确地对一个类实例进行实例化，因此只能有一个。问题就解决了。

10.10.5　使编辑器可扩展

面向对象设计的一大优点是具有好的可维护性，而好的可维护性的一个功能就是可以用最小的努力和风险扩展系统。下面就对上面的作图工具进行扩展——支持其它更多的形状，同时对系统影响最小。

这个目的主要是通过对 ShapeFactory 类进行扩展来提供创建新 Shapes 方法达到的，ShapeFactory 也有一个所有可用 Shapes 的清单，是通过它们的名字动态引用的。图 10-31 中的 MainMenu 使用这个可用 Shapes 的清单来动态地创建“插入形状”（Insert-Shape）菜单。事实上，目前 Shape 的子类只有 ShapeFactory Singleton。所以对可用 Shape 的所有改变都只影响一个类中的一个操作。

```
//---------------------------------------------------------------------------
// MainMenu.java:
// Implementation for menu creation class MainMenu
//
//---------------------------------------------------------------------------
import java.awt.*;
public class MainMenu
{
Frame m_Frame = null;
boolean m_fInitialized = false;
// MenuBar definitions
//---------------------------------------------------------------------------
```

```
MenuBar mb;
// Menu and Menu item definitions
//---------------------------------------------------------------------------
Menu m1; // File
MenuItem ID_FILE_OPEN; // Open
MenuItem ID_FILE_SAVE; // Save
MenuItem ID_FILE_CLOSE; // Close
MenuItem ID_FILE_EXIT; // Exit
Menu m5; // Edit
MenuItem ID_EDIT_COPY; // Copy
MenuItem ID_EDIT_CUT; // Cut
MenuItem ID_EDIT_PASTE; // Paste
Menu mInsert; // Insert
Menu mInsertShape; // Insert-Shape
MenuItem[] InsertMenuID = new MenuItem[100];
int m_NextMenuID = 0; // index into above array
// Constructor
//---------------------------------------------------------------------------
public MainMenu (Frame frame)
{
m_Frame = frame;
}
// Initialization.
//---------------------------------------------------------------------------
public boolean CreateMenu()
{
// Can only init controls once
//---------------------------------------------------------------------
if (m_fInitialized || m_Frame == null)
return false;
// Create menubar and attach to the frame
//---------------------------------------------------------------------
mb = new MenuBar();
m_Frame.setMenuBar(mb);
// Create menu and menu items and assign to menubar
//---------------------------------------------------------------------
m1 = new Menu("File");
mb.add(m1);
ID_FILE_OPEN = new MenuItem("Open");
m1.add(ID_FILE_OPEN);
ID_FILE_SAVE = new MenuItem("Save");
m1.add(ID_FILE_SAVE);
ID_FILE_CLOSE = new MenuItem("Close");
```

```
m1.add(ID_FILE_CLOSE);
m1.addSeparator();
ID_FILE_EXIT = new MenuItem("Exit");
m1.add(ID_FILE_EXIT);
m5 = new Menu("Edit");
mb.add(m5);
ID_EDIT_COPY = new MenuItem("Copy");
m5.add(ID_EDIT_COPY);
ID_EDIT_CUT = new MenuItem("Cut");
m5.add(ID_EDIT_CUT);
ID_EDIT_PASTE = new MenuItem("Paste");
m5.add(ID_EDIT_PASTE);
mInsert = new Menu("Insert");
mInsertShape = new Menu("Shape", true);
mInsert.add(mInsertShape);
mb.add(mInsert);
m_fInitialized = true;
return true;
}
public void AddShapeToInsertMenu(Shape newShape)
{
InsertMenuID[m_NextMenuID] = new MenuItem(newShape.GetShapeName());
mInsertShape.add(InsertMenuID[m_NextMenuID]);
m_NextMenuID++;
}
}
```

图 10-31　代码 MainMenu.java

10.11　小　　结

设计模式是巧妙、通用、得到泛证明和可重用的设计方案，可以应用到面向对象软件开发过程中的许多共同问题。模式描述了方案的核心，在不丢失这个核心的情况下，可以各种方式对模式进行变形和改装。

UML 是用参数化协作描述模式的，其中包括了一个情景（一组对象以及它们之间的关系）和一个交互（对象之间为了完成模式的行为而进行的某个特定的通信）。当作为一个简单元素时，协作是用一个虚椭圆表示的，当扩展它时，可以成一个对象图加一个序列图，或者一个协作图。当用一个协作符号表示时，相关关系说明了系统中的哪个类参加了该模式。参加的类扮演模式中相应的类、对象或操作的角色。

模式和用例的实现非常相似，因为两者都有一个情景和一个交互。模式或用例中的类，是用一个相关表示的。模式和用例的不同之处在于，模型表示一个通用的协作，可以被多个系统使用，而用例是的，只能出现在一个系统中。

本章后面举例说明了如何在软件设计和开发活动中应用模式。模式在软件开发过程有两个主要活动：模式实例化和设计模式候选鉴定。

最后，本章以一个图形编辑器的设计和实现为例，分别列举了行为模式、结构模式和创建模式的例子，说明如何在具体设计中应用它们，包括 Factory Method、Prototype、Command、Memento、Singleton、Composite 和 Observer 及 Prototype Manager。