

第1章 引言

设计面向对象软件比较困难，而设计可复用的面向对象软件就更加困难。你必须找到相关的对象，以适当的粒度将它们归类，再定义类的接口和继承层次，建立对象之间的基本关系。你的设计应该对手头的问题有针对性，同时对将来的问题和需求也要有足够的通用性。你也希望避免重复设计或尽可能少做重复设计。有经验的面向对象设计者会告诉你，要一下子就得到复用性和灵活性好的设计，即使不是不可能的至少也是非常困难的。一个设计在最终完成之前常要被复用好几次，而且每一次都有所修改。

有经验的面向对象设计者的确能做出良好的设计，而新手则面对众多选择无从下手，总是求助于以前使用过的非面向对象技术。新手需要花费较长时间领会良好的面向对象设计是怎么回事。有经验的设计者显然知道一些新手所不知道的东西，这又是什么呢？

内行的设计者知道：不是解决任何问题都要从头做起。他们更愿意复用以前使用过的解决方案。当找到一个好的解决方案，他们会一遍又一遍地使用。这些经验是他们成为内行的部分原因。因此，你会在许多面向对象系统中看到类和相互通信的对象（communicating object）的重复模式。这些模式解决特定的设计问题，使面向对象设计更灵活、优雅，最终复用性更好。它们帮助设计者将新的设计建立在以往工作的基础上，复用以往成功的设计方案。一个熟悉这些模式的设计者不需要再去发现它们，而能够立即将它们应用于设计问题中。

以下类比可以帮助说明这一点。小说家和剧本作家很少从头开始设计剧情。他们总是沿袭一些业已存在的模式，像“悲剧性英雄”模式（《麦克白》、《哈姆雷特》等）或“浪漫小说”模式（存在着无数浪漫小说）。同样地，面向对象设计员也沿袭一些模式，像“用对象表示状态”和“修饰对象以便于你能容易地添加/删除属性”等。一旦懂得了模式，许多设计决策自然而然就产生了。

我们都知道设计经验的重要价值。你曾经多少次有过这种感觉——你已经解决过了一个问题但就是不能确切知道是在什么地方或怎么解决的？如果你能记起以前问题的细节和怎么解决它的，你就可以复用以前的经验而不需要重新发现它。然而，我们并没有很好记录下可供他人使用的软件设计经验。

这本书的目的就是将面向对象软件的设计经验作为设计模式记录下来。每一个设计模式系统地命名、解释和评价了面向对象系统中一个重要的和重复出现的设计。我们的目标是将设计经验以人们能够有效利用的形式记录下来。鉴于此目的，我们编写了一些最重要的设计模式，并以编目分类的形式将它们展现出来。

设计模式使人们可以更加简单方便地复用成功的设计和体系结构。将已证实的技术表述成设计模式也会使新系统开发者更容易理解其设计思路。设计模式帮助你做出有利于系统复用的选择，避免设计损害了系统复用性。通过提供一个显式类和对象作用关系以及它们之间潜在联系的说明规范，设计模式甚至能够提高已有系统的文档管理和系统维护的有效性。简而言之，设计模式可以帮助设计者更快更好地完成系统设计。

本书中涉及的设计模式并不描述新的或未经证实的设计，我们只收录那些在不同系统中

多次使用过的成功设计。这些设计的绝大部分以往并无文档记录，它们或是来源于面向对象设计者圈子里的非正式交流，或是来源于某些成功的面向对象系统的某些部分，但对设计新手来说，这些东西是很难学得到的。尽管这些设计不包含新的思路，但我们用一种新的、便于理解的方式将其展现给读者，即：具有统一格式的、已分类编目的若干组设计模式。

尽管该书涉及较多的内容，但书中讨论的设计模式仅仅包含了一个设计行家所知道的部分。书中没有讨论与并发或分布式或实时程序设计有关的模式，也没有收录面向特定应用领域的模式。本书并不准备告诉你怎样构造用户界面、怎样写设备驱动程序或怎样使用面向对象数据库，这些方面都有自己的模式，将这些模式分类编目也是件很有意义的事。

1.1 什么是设计模式

Christopher Alexander说过：“每一个模式描述了一个在我们周围不断重复发生的问题，以及该问题的解决方案的核心。这样，你就能一次又一次地使用该方案而不必做重复劳动”[AIS+77，第10页]。尽管Alexander所指的是城市和建筑模式，但他的思想也同样适用于面向对象设计模式，只是在面向对象的解决方案里，我们用对象和接口代替了墙壁和门窗。两类模式的核心都在于提供了相关问题的解决方案。

一般而言，一个模式有四个基本要素：

1. 模式名称 (pattern name) 一个助记名，它用一两个词来描述模式的问题、解决方案和效果。命名一个新的模式增加了我们的设计词汇。设计模式允许我们在较高的抽象层次上进行设计。基于一个模式词汇表，我们自己以及同事之间就可以讨论模式并在编写文档时使用它们。模式名可以帮助我们思考，便于我们与其他人交流设计思想及设计结果。找到恰当的模式名也是我们设计模式编目工作的难点之一。

2. 问题(problem) 描述了应该在何时使用模式。它解释了设计问题和存在的问题的前因后果，它可能描述了特定的设计问题，如怎样用对象表示算法等。也可能描述了导致不灵活设计的类或对象结构。有时候，问题部分会包括使用模式必须满足的一系列先决条件。

3. 解决方案(solution) 描述了设计的组成成分，它们之间的相互关系及各自的职责和协作方式。因为模式就像一个模板，可应用于多种不同场合，所以解决方案并不描述一个特定而具体的设计或实现，而是提供设计问题的抽象描述和怎样用一个具有一般意义的元素组合（类或对象组合）来解决这个问题。

4. 效果(consequences) 描述了模式应用的效果及使用模式应权衡的问题。尽管我们描述设计决策时，并不总提到模式效果，但它们对于评价设计选择和理解使用模式的代价及好处具有重要意义。软件效果大多关注对时间和空间的衡量，它们也表述了语言和实现问题。因为复用是面向对象设计的要素之一，所以模式效果包括它对系统的灵活性、扩充性或可移植性的影响，显式地列出这些效果对理解和评价这些模式很有帮助。

出发点的不同会产生对什么是模式和什么不是模式的理解不同。一个人的模式对另一个人来说可能只是基本构造部件。本书中我们将在一定的抽象层次上讨论模式。《设计模式》并不描述链表和hash表那样的设计，尽管它们可以用类来封装，也可复用；也不包括那些复杂的、特定领域内的对整个应用或子系统的设计。本书中的设计模式是对被用来在特定场景下解决一般设计问题的类和相互通信的对象的描述。

一个设计模式命名、抽象和确定了一个通用设计结构的主要方面，这些设计结构能被用

来构造可复用的面向对象设计。设计模式确定了所包含的类和实例，它们的角色、协作方式以及职责分配。每一个设计模式都集中于一个特定的面向对象设计问题或设计要点，描述了什么时候使用它，在另一些设计约束条件下是否还能使用，以及使用的效果和如何取舍。既然我们最终要实现设计，设计模式还提供了 C++ 和 Smalltalk 示例代码来阐明其实现。

虽然设计模式描述的是面向对象设计，但它们都基于实际的解决方案，这些方案的实现语言是 Smalltalk 和 C++ 等主流面向对象编程语言，而不是过程式语言 (Pascal、C、Ada) 或更具动态特性的面向对象语言 (CLOS、Dylan、Self)。我们从实用角度出发选择了 Smalltalk 和 C++，因为在这些语言的使用上，我们积累了许多经验，况且它们也变得越来越流行。

程序设计语言的选择非常重要，它将影响人们理解问题的出发点。我们的设计模式采用了 Smalltalk 和 C++ 层的语言特性，这个选择实际上决定了哪些机制可以方便地实现，而哪些则不能。若我们采用过程式语言，可能就要包括诸如“继承”、“封装”和“多态”的设计模式。相应地，一些特殊的面向对象语言可以直接支持我们的某些模式，例如：CLOS 支持多方法 (multi-method) 概念，这就减少了 Visitor 模式的必要性。事实上，Smalltalk 和 C++ 已有足够的差别来说明对某些模式一种语言比另一种语言表述起来更容易一些 (参见 5.4 节 Iterator 模式)。

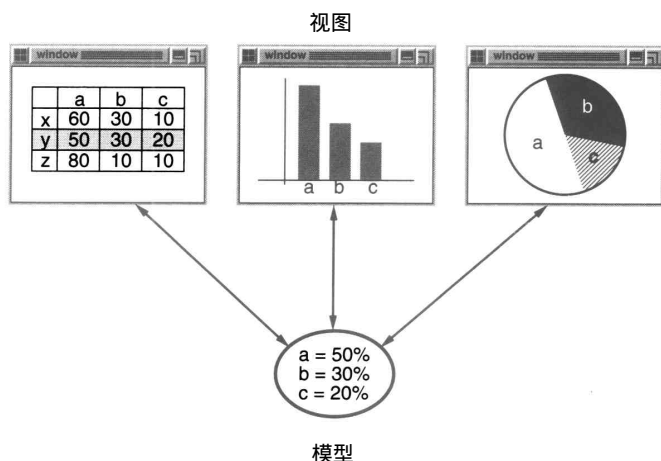
1.2 Smalltalk MVC 中的设计模式

在 Smalltalk-80 中，类的模型/视图/控制器 (Model/View/Controller) 三元组 (MVC) 被用来构建用户界面。透过 MVC 来看设计模式将帮助我们理解“模式”这一术语的含义。

MVC 包括三类对象。模型 Model 是应用对象，视图 View 是它在屏幕上的表示，控制器 Controller 定义用户界面对用户输入的响应方式。不使用 MVC，用户界面设计往往将这些对象混在一起，而 MVC 则将它们分离以提高灵活性和复用性。

MVC 通过建立一个“订购/通知”协议来分离视图和模型。视图必须保证它的显示正确地反映了模型的状态。一旦模型的数据发生变化，模型将通知有关的视图，每个视图相应地得到刷新自己的机会。这种方法可以让你为一个模型提供不同的多个视图表现形式，也能够为一个模型创建新的视图而无须重写模型。

下图显示了一个模型和三个视图 (为了简单起见我们省略了控制器)。模型包含一些数据值，视图通过电子表格、柱状图、饼图这些不同的方式来显示这些数据。当模型的数据发生变化时，模型就通知它的视图，而视图将与模型通信以访问这些数据值。



表面上看，这个例子反映了将视图和模型分离的设计，然而这个设计还可用于解决更一般的问题：将对象分离，使得一个对象的改变能够影响另一些对象，而这个对象并不需要知道那些被影响的对象的细节。这个更一般的设计被描述成 Observer (5.7) 模式。

MVC的另一个特征是视图可以嵌套。例如，按钮控制面板可以用一个嵌套了按钮的复杂视图来实现。对象查看器的用户界面可由嵌套的视图构成，这些视图又可复用于调试器。MVC用View类的子类——CompositeView类来支持嵌套视图。CompositeView类的对象行为类似于View类对象，一个组合视图可用于任何视图可用的地方，但是它包含并管理嵌套视图。

上例反映了可以将组合视图与其构件平等对待的设计，同样地，该设计也适用于更一般的问题：将一些对象划为一组，并将该组对象当作一个对象来使用。这个设计被描述为 Composite (4.3) 模式，该模式允许你创建一个类层次结构，一些子类定义了原子对象（如 Button）而其他类定义了组合对象（CompositeView），这些组合对象是由原子对象组合而成的更复杂的对象。

MVC允许你在不改变视图外观的情况下改变视图对用户输入的响应方式。例如，你可能希望改变视图对键盘的响应方式，或希望使用弹出菜单而不是原来的命令键方式。MVC将响应机制封装在 Controller 对象中。存在着一个 Controller 的类层次结构，使得可以方便地对原有 Controller 做适当改变而创建新的 Controller。

View使用Controller子类的实例来实现一个特定的响应策略。要实现不同的响应策略只要用不同种类的 Controller 实例替换即可。甚至可以在运行时刻通过改变 View 的 Controller 来改变 View 对用户输入的响应方式。例如，一个 View 可以被禁止接收任何输入，只需给它一个忽略输入事件的 Controller。

View-Controller 关系是 Strategy (5.9) 模式的一个例子。一个策略是一个表述算法的对象。当你想静态或动态地替换一个算法，或你有很多不同的算法，或算法中包含你想封装的复杂数据结构，这时策略模式是非常有用的。

MVC还使用了其他的设计模式，如：用来指定视图缺省控制器的 Factory Method(3.3) 和用来增加视图滚动的 Decorator(4.4)。但是 MVC 的主要关系还是由 Observer、Composite 和 Strategy 三个设计模式给出的。

1.3 描述设计模式

我们怎样描述设计模式呢？图形符号虽然很重要也很有用，却还远远不够，它们只是将设计过程的结果简单记录为类和对象之间的关系。为了达到设计复用，我们必须同时记录设计产生的决定过程、选择过程和权衡过程。具体的例子也是很重要的，它们让你看到实际的设计。

我们将用统一的格式描述设计模式，每一个模式根据以下的模板被分成若干部分。模板具有统一的信息描述结构，有助于你更容易地学习、比较和使用设计模式。

模式名和分类

模式名简洁地描述了模式的本质。一个好的名字非常重要，因为它将成为你的设计词汇表的一部分。模式的分类反映了我们将在 1.5 节介绍的方案。

意图

是回答下列问题的简单陈述：设计模式是做什么的？它的基本原理和意图是什么？它解

决的是什么样的特定设计问题？

别名

模式的其他名称。

动机

用以说明一个设计问题以及如何用模式中的类、对象来解决该问题的特定情景。该情景会帮助你理解随后对模式更抽象的描述。

适用性

什么情况下可以使用该设计模式？该模式可用来改进哪些不良设计？你怎样识别这些情况？

结构

采用基于对象建模技术（OMT）[RBP+91]的表示法对模式中的类进行图形描述。我们也使用了交互图 [JCJO92, BOO94]来说明对象之间的请求序列和协作关系。附录 B详细描述了这些表示法。

参与者

指设计模式中的类和/或对象以及它们各自的职责。

协作

模式的参与者怎样协作以实现它们的职责。

效果

模式怎样支持它的目标？使用模式的效果和所需做的权衡取舍？系统结构的哪些方面可以独立改变？

实现

实现模式时需要知道的一些提示、技术要点及应避免的缺陷，以及是否存在某些特定于实现语言的问题。

代码示例

用来说明怎样用 C++ 或 Smalltalk 实现该模式的代码片段。

已知应用

实际系统中发现的模式的例子。每个模式至少包括了两个不同领域的实例。

相关模式

与这个模式紧密相关的模式有哪些？其间重要的不同之处是什么？这个模式应与哪些其他模式一起使用？

附录提供的背景资料将帮助你理解模式以及关于模式的讨论。附录 A给出了我们使用的术语列表。前面已经提到过的附录 B则给出了各种表示法，我们也会在以后的讨论中简单介绍它们。最后，附录 C给出了我们在例子中使用的各基本类的源代码。

1.4 设计模式的编目

从第3章开始的模式目录中共包含 23 个设计模式。它们的名字和意图列举如下，以使你有基本了解。每个模式名后括号中标出模式所在的章节（我们整本书都将遵从这个约定）。

Abstract Factory(3.1)：提供一个创建一系列相关或相互依赖对象的接口，而无需指定它们具体的类。

Adapter(4.1)：将一个类的接口转换成客户希望的另外一个接口。Adapter模式使得原本由于接口不兼容而不能一起工作的那些类可以一起工作。

Bridge(4.2)：将抽象部分与它的实现部分分离，使它们都可以独立地变化。

Builder(3.2)：将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。

Chain of Responsibility(5.1)：为解除请求的发送者和接收者之间耦合，而使多个对象都有机会处理这个请求。将这些对象连成一条链，并沿着这条链传递该请求，直到有一个对象处理它。

Command(5.2)：将一个请求封装为一个对象，从而使你可用不同的请求对客户进行参数化；对请求排队或记录请求日志，以及支持可取消的操作。

Composite(4.3)：将对象组合成树形结构以表示“部分-整体”的层次结构。Composite使得客户对单个对象和复合对象的使用具有一致性。

Decorator(4.4)：动态地给一个对象添加一些额外的职责。就扩展功能而言，Decorator模式比生成子类方式更为灵活。

Facade(4.5)：为子系统中的一组接口提供一个一致的界面，Facade模式定义了一个高层接口，这个接口使得这一子系统更加容易使用。

Factory Method(3.3)：定义一个用于创建对象的接口，让子类决定将哪一个类实例化。Factory Method使一个类的实例化延迟到其子类。

Flyweight(4.6)：运用共享技术有效地支持大量细粒度的对象。

Interpreter(5.3)：给定一个语言，定义它的文法的一种表示，并定义一个解释器，该解释器使用该表示来解释语言中的句子。

Iterator(5.4)：提供一种方法顺序访问一个聚合对象中各个元素，而又不需暴露该对象的内部表示。

Mediator(5.5)：用一个中介对象来封装一系列的对象交互。中介者使各对象不需要显式地相互引用，从而使其耦合松散，而且可以独立地改变它们之间的交互。

Memento(5.6)：在不破坏封装性的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态。这样以后就可将该对象恢复到保存的状态。

Observer(5.7)：定义对象间的一种一对多的依赖关系，以便当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并自动刷新。

Prototype(3.4)：用原型实例指定创建对象的种类，并且通过拷贝这个原型来创建新的对象。

Proxy(4.7)：为其他对象提供一个代理以控制对这个对象的访问。

Singleton(3.5)：保证一个类仅有一个实例，并提供一个访问它的全局访问点。

State(5.8)：允许一个对象在其内部状态改变时改变它的行为。对象看起来似乎修改了它所属的类。

Strategy(5.9)：定义一系列的算法，把它们一个个封装起来，并且使它们可相互替换。本模式使得算法的变化可独立于使用它的客户。

Template Method(5.10)：定义一个操作中的算法的骨架，而将一些步骤延迟到子类中。Template Method使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤。

Visitor(5.11)：表示一个作用于某对象结构中的各元素的操作。它使你可以在不改变各元素的类的前提下定义作用于这些元素的新操作。

1.5 组织编目

设计模式在粒度和抽象层次上各不相同。由于存在众多的设计模式，我们希望用一种方式将它们组织起来。这一节将对设计模式进行分类以便于我们对各族相关的模式进行引用。分类有助于更快地学习目录中的模式，且对发现新的模式也有指导作用，如表1-1所示。

表1-1 设计模式空间

		目 的		
		创 建 型	结 构 型	行 为 型
范围	类	Factory Method(3.3)	Adapter(类)(4.1)	Interpreter(5.3) Template Method(5.10)
	对象	Abstract Factory(3.1) Builder(3.2) Prototype(3.4) Singleton(3.5)	Adapter(对象)(4.1) Bridge(4.2) Composite(4.3) Decorator(4.4) Facade(4.5) Flyweight(4.6) Proxy(4.7)	Chain of Responsibility(5.1) Command(5.2) Iterator(5.4) Mediator(5.5) Memento(5.6) Observer(5.7) State(5.8) Strategy(5.9) Visitor(5.10)

我们根据两条准则(表1-1)对模式进行分类。第一是目的准则，即模式是用来完成什么工作的。模式依据其目的可分为创建型 (Creational)、结构型 (Structural)、或行为型 (Behavioral)三种。创建型模式与对象的创建有关；结构型模式处理类或对象的组合；行为型模式对类或对象怎样交互和怎样分配职责进行描述。

第二是范围准则，指定模式主要是用于类还是用于对象。类模式处理类和子类之间的关系，这些关系通过继承建立，是静态的，在编译时刻便确定下来了。对象模式处理对象间的关系，这些关系在运行时刻是可以变化的，更具动态性。从某种意义上来说，几乎所有模式都使用继承机制，所以“类模式”只指那些集中于处理类间关系的模式，而大部分模式都属于对象模式的范畴。

创建型类模式将对象的部分创建工作延迟到子类，而创建型对象模式则将它延迟到另一个对象中。结构型类模式使用继承机制来组合类，而结构型对象模式则描述了对对象的组装方式。行为型类模式使用继承描述算法和控制流，而行为型对象模式则描述一组对象怎样协作完成单个对象所无法完成的任务。

还有其他组织模式的方式。有些模式经常会被绑在一起使用，例如， Composite常和 Iterator或 Visitor一起使用；有些模式是可替代的，例如， Prototype常用来替代 Abstract Factory；有些模式尽管使用意图不同，但产生的设计结果是很相似的，例如， Composite和 Decorator的结构图是相似的。

还有一种方式是根据模式的“相关模式”部分所描述的它们怎样互相引用来组织设计模式。图1-1给出了模式关系的图形说明。

显然，存在着许多组织设计模式的方法。从多角度去思考模式有助于对它们的功能、差异和应用场合的更深入理解。

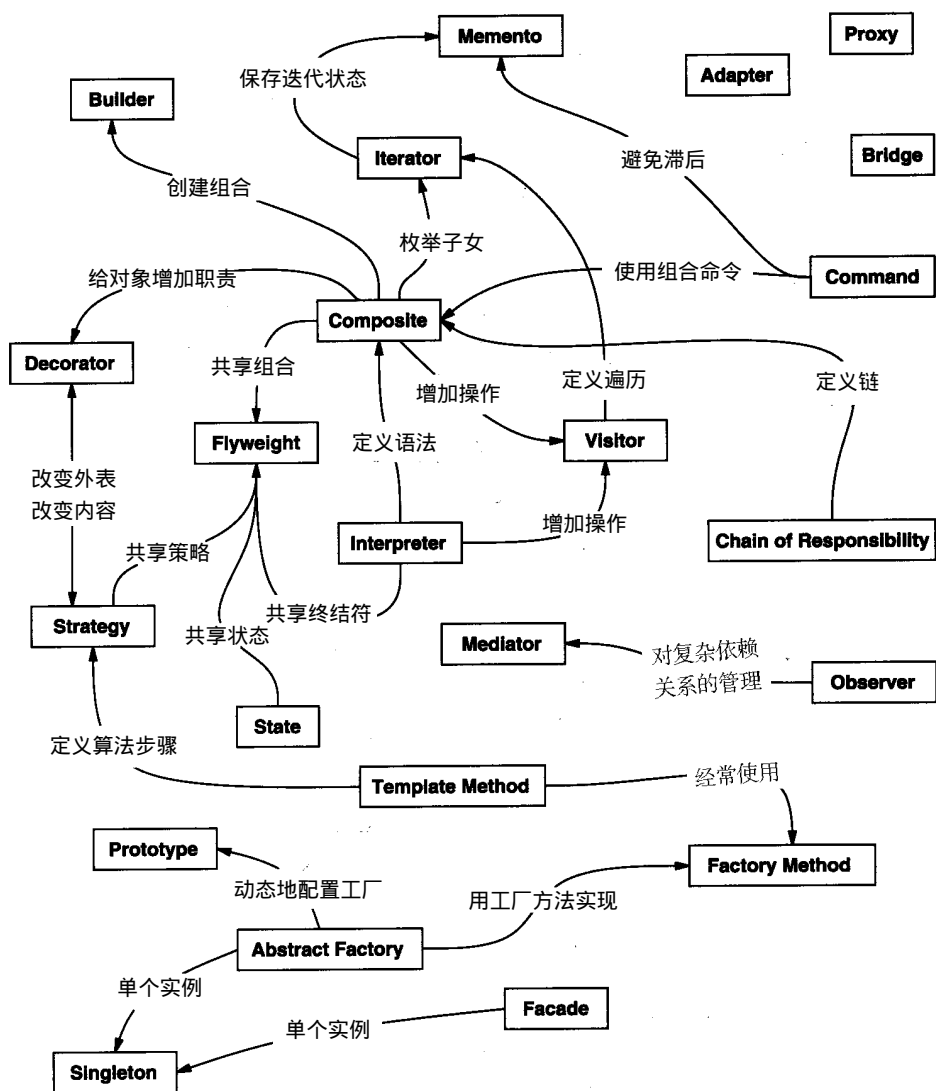


图1-1 设计模式之间的关系

1.6 设计模式怎样解决设计问题

设计模式采用多种方法解决面向对象设计者经常碰到的问题。这里给出几个问题以及使用设计模式解决它们的方法。

1.6.1 寻找合适的对象

面向对象程序由对象组成，对象包括数据和对数据进行操作的过程，过程通常称为方法或操作。对象在收到客户的请求(或消息)后，执行相应的操作。

客户请求是使对象执行操作的唯一方法，操作又是对象改变内部数据的唯一方法。由于这些限制，对象的内部状态是被封装的，它不能被直接访问，它的表示对于对象外部是不可见的。

面向对象设计最困难的部分是将系统分解成对象集合。因为要考虑许多因素：封装、粒度、依赖关系、灵活性、性能、演化、复用等等，它们都影响着系统的分解，并且这些因素通常还是互相冲突的。

面向对象设计方法学支持许多设计方法。你可以写出一个问题描述，挑出名词和动词，进而创建相应的类和操作；或者，你可以关注于系统的协作和职责关系；或者，你可以对现实世界建模，再将分析时发现的对象转化至设计中。至于哪一种方法最好，并无定论。

设计的许多对象来源于现实世界的分析模型。但是，设计结果所得到的类通常在现实世界中并不存在，有些是像数组之类的低层类，而另一些则层次较高。例如，Composite(4.3)模式引入了统一对待现实世界中并不存在的对象的抽象方法。严格反映当前现实世界的模型并不能产生也能反映将来世界的系统。设计中的抽象对于产生灵活的设计是至关重要的。

设计模式帮你确定并不明显的抽象和描述这些抽象的对象。例如，描述过程或算法的对象现实中并不存在，但它们却是设计的关键部分。Strategy(5.9)模式描述了怎样实现可互换的算法族。State(5.8)模式将实体的每一个状态描述为一个对象。这些对象在分析阶段，甚至在设计阶段的早期都不存在，后来为使设计更灵活、复用性更好才将它们发掘出来。

1.6.2 决定对象的粒度

对象在大小和数目上变化极大。它们能表示下自硬件或上自整个应用的任何事物。那么我们怎样决定一个对象应该是什么呢？

设计模式很好地讲述了这个问题。Facade(4.5)模式描述了怎样用对象表示完整的子系统，Flyweight(4.6)模式描述了如何支持大量的最小粒度的对象。其他一些设计模式描述了将一个对象分解成许多小对象的特定方法。Abstract Factory(3.1)和Builder(3.2)产生那些专门负责生成其他对象的对象。Visitor(5.10)和Command(5.2)生成的对象专门负责实现对其他对象或对象组的请求。

1.6.3 指定对象接口

对象声明的每一个操作指定操作名、作为参数的对象和返回值，这就是所谓的操作的型构(signature)。对象操作所定义的所有操作型构的集合被称为该对象的接口(interface)。对象接口描述了该对象所能接受的全部请求的集合，任何匹配对象接口中型构的请求都可以发送给该对象。

类型(type)是用来标识特定接口的一个名字。如果一个对象接受“Window”接口所定义的所有操作请求，那么我们就说该对象具有“Window”类型。一个对象可以有許多类型，并且不同的对象可以共享同一个类型。对象接口的某部分可以用某个类型来刻画，而其他部分则可用其他类型刻画。两个类型相同的对象只需要共享它们的部分接口。接口可以包含其他接口作为子集。当一个类型的接口包含另一个类型的接口时，我们就说它是另一个类型的子类型(subtype)，另一个类型称之为它的超类型(supertype)。我们常说子类型继承了它的超类型的接口。

在面向对象系统中，接口是基本的组成部分。对象只有通过它们的接口才能与外部交流，如果不通过对象的接口就无法知道对象的任何事情，也无法请求对象做任何事情。对象接口与其功能实现是分离的，不同对象可以对请求做不同的实现，也就是说，两个有相同接口的对象可以有完全不同的实现。

当给对象发送请求时，所引起的具体操作既与请求本身有关又与接受对象有关。支持相同请求的不同对象可能对请求激发的操作有不同的实现。发送给对象的请求和它的相应操作在运行时刻的连接就称之为动态绑定(dynamic binding)。

动态绑定是指发送的请求直到运行时刻才受你的具体的实现的约束。因而，在知道任何有正确接口的对象都将接受此请求时，你可以写一个一般的程序，它期待着那些具有该特定接口的对象。进一步讲，动态绑定允许你在运行时刻彼此替换有相同接口的对象。这种可替换性就称为多态(polymorphism)，它是面向对象系统中的核心概念之一。多态允许客户对象仅要求其他对象支持特定接口，除此之外对其假设几近于无。多态简化了客户的定义，使得对象间彼此独立，并可以在运行时刻动态改变它们相互的关系。

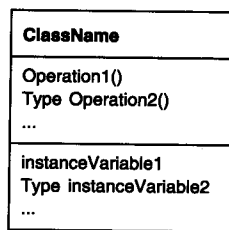
设计模式通过确定接口的主要组成成分及经接口发送的数据类型，来帮助你定义接口。设计模式也许还会告诉你接口中不应包括哪些东西。Memento(5.6)模式是一个很好的例子，它描述了怎样封装和保存对象内部的状态，以便一段时间后对象能恢复到这一状态。它规定了Memento对象必须定义两个接口：一个允许客户保持和复制memento的限制接口，和一个只有原对象才能使用的用来储存和提取memento中状态的特权接口。

设计模式也指定了接口之间的关系。特别地，它们经常要求一些类具有相似的接口；或它们对一些类的接口做了限制。例如，Decorator(4.4)和Proxy(4.7)模式要求Decorator和Proxy对象的接口与被修饰的对象和受委托的对象一致。而Visitor(5.11)模式中，Visitor接口必须反映出visitor能访问的对象的所有类。

1.6.4 描述对象的实现

至此，我们很少提及到实际上怎么去定义一个对象。对象的实现是由它的类决定的，类指定了对象的内部数据和表示，也定义了对象所能完成的操作，如右图所示。

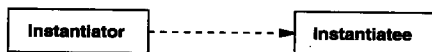
我们基于OMT的表示法，将类描述成一个矩形，其中的类名以黑体表示的。操作在类名下面，以常规字体表示。类所定义的任何数据都在操作的下面。类名与操作之间以及操作与数据之间用横线分割。



返回类型和实例变量类型是可选的，因为我们并未假设一定要用具有静态类型的实现语言。

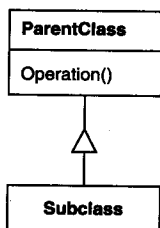
对象通过实例化类来创建，此对象被称为该类的实例。当实例化类时，要给对象的内部数据(由实例变量组成)分配存储空间，并将操作与这些数据联系起来。对象的许多类似实例是由实例化同一个类来创建的。

下图中的虚箭头线表示一个类实例化另一个类的对象，箭头指向被实例化的对象的类。



新的类可以由已存在的类通过类继承(class inheritance)来定义。当子类(subclass)继承父类

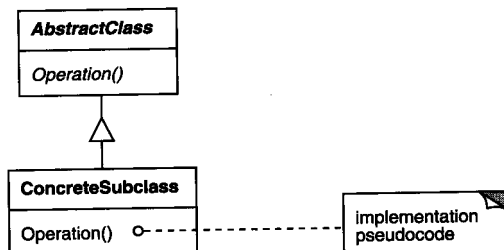
(parent class)时，子类包含了父类定义的所有数据和操作。子类的实例对象包含所有子类和父类定义的数据，且它们能完成子类和父类定义的所有操作。我们以竖线和三角表示子类关系，如下图所示。



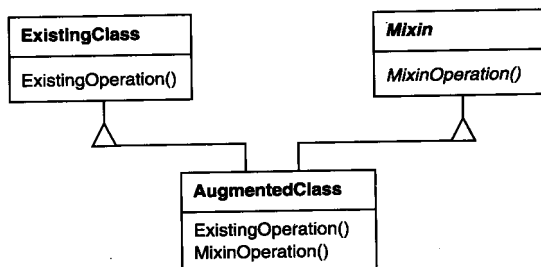
抽象类(abstract class)的主要目的是为它的子类定义公共接口。一个抽象类将它的部分或全部操作的实现延迟到子类中，因此，一个抽象类不能被实例化。在抽象类中定义却没有实现的操作被称为抽象操作(abstract operation)。非抽象类称为具体类(concrete class)。

子类能够改进和重新定义它们父类的操作。更具体地说，类能够重定义(override)父类定义的操作，重定义使得子类能接管父类对请求的处理操作。类继承允许你只需简单的扩展其他类就可以定义新类，从而可以很容易地定义具有相近功能的对象族。

抽象类的类名以斜体表示，以与具体类相区别。抽象操作也用斜体表示。图中可以包括实现操作的伪代码，如果这样，则代码将出现在带有摺角的框中，并用虚线将该摺角框与代码所实现的操作相连，图示如下。



混入类(mixin class)是给其他类提供可选的接口或功能的类。它与抽象类一样不能实例化。混入类要求多继承，图示如下。



1. 类继承与接口继承的比较

理解对象的类(class)与对象的类型(type)之间的差别非常重要。

一个对象的类定义了对象是怎样实现的，同时也定义了对象的内部状态和操作的实现。但是对象的类型只与它的接口有关，接口即对象能响应的请求的集合。一个对象可以有多个

类型，不同类的对象可以有相同的类型。

当然，对象的类和类型是有紧密关系的。因为类定义了对象所能执行的操作，也定义了对象的类型。当我们说一个对象是一个类的实例时，即指该对象支持类所定义的接口。

C++和Eiffel语言的类既指定对象的类型又指定对象的实现。Smalltalk程序不声明变量的类型，所以编译器不检查赋给变量的对象类型是否是该变量的类型的子类型。发送消息时需要检查消息接收者是否实现了该消息，但不检查接收者是否是某个特定类的实例。

理解类继承和接口继承(或子类型化)之间的差别也十分重要。类继承根据一个对象的实现定义了另一个对象的实现。简而言之，它是代码和表示的共享机制。然而，接口继承(或子类型化)描述了一个对象什么时候能被用来替代另一个对象。

因为许多语言并不显式地区分这两个概念，所以容易被混淆。在C++和Eiffel语言中，继承既指接口的继承又指实现的继承。C++中接口继承的标准方法是公有继承一个含(纯)虚成员函数的类。C++中纯接口继承接近于公有继承纯抽象类，纯实现继承或纯类继承接近于私有继承。Smalltalk中的继承只指实现继承。只要任何类的实例支持对变量值的操作，你就可以将这些实例赋给变量。

尽管大部分程序设计语言并不区分接口继承和实现继承的差别，但使用中人们还是分别对待它们的。Smalltalk程序员通常将子类当作子类型(尽管有一些熟知的例外情况[Coo92])，C++程序员通过抽象类所定义的类型来操纵对象。

很多设计模式依赖于这种差别。例如，Chain of Responsibility(5.1)模式中的对象必须有一个公共的类型，但一般情况下它们不具有公共的实现。在Composite(4.3)模式中，构件定义了一个公共的接口，但Composite通常定义一个公共的实现。Command(5.2)、Observer(5.7)、State(5.8)和Strategy(5.9)通常纯粹作为接口的抽象类来实现。

2. 对接口编程，而不是对实现编程

类继承是一个通过复用父类功能而扩展应用功能的基本机制。它允许你根据旧对象快速定义新对象。它允许你从已存在的类中继承所需要的绝大部分功能，从而几乎无需任何代价就可以获得新的实现。

然而，实现的复用只是成功的一半，继承所拥有的定义具有相同接口的对象族的能力也是很重要的(通常可以从抽象类来继承)。为什么？因为多态依赖于这种能力。

当继承被恰当使用时，所有从抽象类导出的类将共享该抽象类的接口。这意味着子类仅仅添加或重定义操作，而没有隐藏父类的操作。这时，所有的子类都能响应抽象类接口中的请求，从而子类的类型都是抽象类的子类型。

只根据抽象类中定义的接口来操纵对象有以下两个好处：

- 1) 客户无须知道他们使用对象的特定类型，只须对象有客户所期望的接口。
- 2) 客户无须知道他们使用的对象是用什么类来实现的，他们只须知道定义接口的抽象类。

这将极大地减少子系统实现之间的相互依赖关系，也产生了可复用的面向对象设计的如下原则：

针对接口编程，而不是针对实现编程。

不将变量声明为某个特定的具体类的实例对象，而是让它遵从抽象类所定义的接口。这是本书设计模式的一个常见主题。

当你不得不在系统的某个地方实例化具体的类(即指定一个特定的实现)时，创建型模式

(Abstract Factory(3.1), Builder(3.2), Factory Method(3.3), Prototype(3.4)和Singleton(3.5))可以帮助你。通过抽象对象的创建过程,这些模式提供不同方式以在实例化时建立接口和实现的透明连接。创建型模式确保你的系统是采用针对接口的方式书写的,而不是针对实现而书写的。

1.6.5 运用复用机制

理解对象、接口、类和继承之类的概念对大多数人来说并不难,问题的关键在于如何运用它们写出灵活的、可复用的软件。设计模式将告诉你怎样去做。

1. 继承和组合的比较

面向对象系统中功能复用的两种最常用技术是类继承和对象组合(object composition)。正如我们已解释过的,类继承允许你根据其他类的实现来定义一个类的实现。这种通过生成子类的复用通常被称为白箱复用(white-box reuse)。术语“白箱”是相对可视性而言:在继承方式中,父类的内部细节对子类可见。

对象组合是类继承之外的另一种复用选择。新的更复杂的功能可以通过组装或组合对象来获得。对象组合要求被组合的对象具有良好定义的接口。这种复用风格被称为黑箱复用(black-box reuse),因为对象的内部细节是不可见的。对象只以“黑箱”的形式出现。

继承和组合各有优缺点。类继承是在编译时刻静态定义的,且可直接使用,因为程序设计语言直接支持类继承。类继承可以较方便地改变被复用的实现。当一个子类重定义一些而不是全部操作时,它也能影响它所继承的操作,只要在这些操作中调用了被重定义的操作。

但是类继承也有一些不足之处。首先,因为继承在编译时刻就定义了,所以无法在运行时刻改变从父类继承的实现。更糟的是,父类通常至少定义了部分子类的具体表示。因为继承对子类揭示了其父类的实现细节,所以继承常被认为“破坏了封装性”[Sny86]。子类中的实现与它的父类有如此紧密的依赖关系,以至于父类实现中的任何变化必然会导致子类发生变化。

当你需要复用子类时,实现上的依赖性就会产生一些问题。如果继承下来的实现不适合解决新的问题,则父类必须重写或被其他更适合的类替换。这种依赖关系限制了灵活性并最终限制了复用性。一个可用的解决方法就是只继承抽象类,因为抽象类通常提供较少的实现。

对象组合是通过获得对其他对象的引用而在运行时刻动态定义的。组合要求对象遵守彼此的接口约定,进而要求更仔细地定义接口,而这些接口并不妨碍你将一个对象和其他对象一起使用。这还会产生良好的结果:因为对象只能通过接口访问,所以我们并不破坏封装性;只要类型一致,运行时刻还可以用一个对象来替代另一个对象;更进一步,因为对象的实现是基于接口写的,所以实现上存在较少的依赖关系。

对象组合对系统设计还有另一个作用,即优先使用对象组合有助于你保持每个类被封装,并被集中在单个任务上。这样类和类继承层次会保持较小规模,并且不太可能增长为不可控制的庞然大物。另一方面,基于对象组合的设计会有更多的对象(而有较少的类),且系统的行为将依赖于对象间的关系而不是被定义在某个类中。

这导出了我们的面向对象设计的第二个原则:

优先使用对象组合,而不是类继承。

理想情况下,你不应为获得复用而去创建新的构件。你应该能够只使用对象组合技术,

通过组装已有的构件就能获得你需要的功能。但是事实很少如此，因为可用构件的集合实际上并不足够丰富。使用继承的复用使得创建新的构件要比组装旧的构件来得容易。这样，继承和对象组合常一起使用。

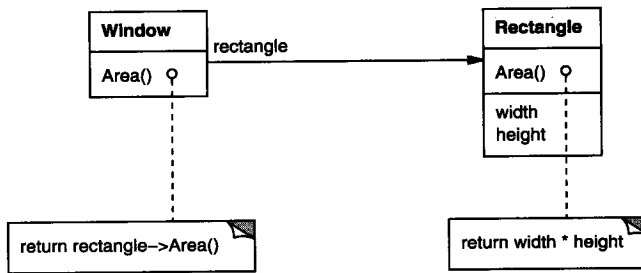
然而，我们的经验表明：设计者往往过度使用了继承这种复用技术。但依赖于对象组合技术的设计却有更好的复用性(或更简单)。你将会看到设计模式中一再使用对象组合技术。

2. 委托

委托(delegation)是一种组合方法，它使组合具有与继承同样的复用能力 [Lie86,JZ91]。在委托方式下，有两个对象参与处理一个请求，接受请求的对象将操作委托给它的代理者(delegate)。这类似于子类将请求交给它的父类处理。使用继承时，被继承的操作总能引用接受请求的对象，C++中通过this成员变量，Smalltalk中则通过self。委托方式为了得到同样的效果，接受请求的对象将自己传给被委托者(代理人)，使被委托的操作可以引用接受请求的对象。

举例来说，我们可以在窗口类中保存一个矩形类的实例变量来代理矩形类的特定操作，这样窗口类可以复用矩形类的操作，而不必像继承时那样定义成矩形类的子类。也就是说，一个窗口拥有一个矩形，而不是一个窗口就是一个矩形。窗口现在必须显式的将请求转发给它的矩形实例，而不是像以前它必须继承矩形的操作。

下面的图显示了窗口类将它的Area操作委托给一个矩形实例。



箭头线表示一个类对另一个类实例的引用关系。引用名是可选的，本例为“rectangle”。

委托的主要优点在于它便于运行时刻组合对象操作以及改变这些操作的组合方式。假定矩形对象和圆对象有相同的类型，我们只需简单的用圆对象替换矩形对象，则得到的窗口就是圆形的。

委托与那些通过对象组合以取得软件灵活性的技术一样，具有如下不足之处：动态的、高度参数化的软件比静态软件更难于理解。还有运行低效问题，不过从长远来看人的低效才是更主要的。只有当委托使设计比较简单而不是更复杂时，它才是好的选择。要给出一个能确切告诉你什么时候可以使用委托的规则是很困难的。因为委托可以得到的效率是与上下文有关的，并且还依赖于你的经验。委托最适用于符合特定程式的情形，即标准模式的情形。

有一些模式使用了委托，如State(5.8)、Strategy(5.9)和Visitor(5.11)。在State模式中，一个对象将请求委托给一个描述当前状态的State对象来处理。在Strategy模式中，一个对象将一个特定的请求委托给一个描述请求执行策略的对象，一个对象只会有一个状态，但它对不同的请求可以有許多策略。这两个模式的目的是通过改变受托对象来改变委托对象的行为。在Visitor中，对象结构的每个元素上的操作总是被委托到Visitor对象。

其他模式则没有这么多地用到委托。Mediator(5.5)引进了一个中介其他对象间通信的对

象。有时，Mediator对象只是简单地将请求转发给其他对象；有时，它沿着指向自己的引用来传递请求，使用真正意义的委托。Chain of Responsibility(5.1)通过将请求沿着对象链传递来处理请求，有时，这个请求本身带有一个接受请求对象的引用，这时该模式就使用了委托。Bridge(4.2)将实现和抽象分离开，如果抽象和一个特定实现非常匹配，那么这个实现可以代理抽象的操作。

委托是对象组合的特例。它告诉你对象组合作为一个代码复用机制可以替代继承。

3. 继承和参数化类型的比较

另一种功能复用技术(并非严格的面向对象技术)是参数化类型(parameterized type)，也就是类属(generic)(Ada、Eiffel)或模板(templates)(C++)。它允许你在定义一个类型时并不指定该类型所用到的其他所有类型。未经指定的类型在使用时以参数形式提供。例如，一个列表类能够以它所包含元素的类型来进行参数化。如果你想声明一个 Integer列表，只需将Integer类型作为列表参数化类型的参数值；声明一个 String列表，只需提供String类型作为参数值。语言的实现将会为各种元素类型创建相应的列表类模板的定制版本。

参数化类型给我们提供除了类继承和对象组合外的第三种方法来组合面向对象系统中的行为。许多设计可以使用这三种技术中的任何一种来实现。实现一个以元素比较操作为可变元的排序例程，可有如下方法：

- 1) 通过子类实现该操作(Template Method(5.10)的一个应用)。
- 2) 实现为传给排序例程的对象的职责 (Strategy(5.9))。
- 3) 作为C++模板或Ada类属的参数，以指定元素比较操作的名称。

这些技术存在着极大的不同之处。对象组合技术允许你在运行时刻改变被组合的行为，但是它存在间接性，比较低效。继承允许你提供操作的缺省实现，并通过子类重定义这些操作。参数化类型允许你改变类所用到的类型。但是继承和参数化类型都不能在运行时刻改变。哪一种方法最佳，取决于你设计和实现的约束条件。

本书没有一种模式是与参数化类型有关的，尽管我们在定制一个模式的 C++实现时用到了参数化类型。参数化类型在像 Smalltalk那样的编译时刻不进行类型检查的语言中是完全不必要的。

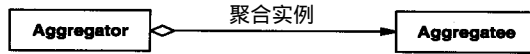
1.6.6 关联运行时刻和编译时刻的结构

一个面向对象程序运行时刻的结构通常与它的代码结构相差较大。代码结构在编译时刻就被确定下来了，它由继承关系固定的类组成。而程序的运行时刻结构是由快速变化的通信对象网络组成。事实上两个结构是彼此独立的，试图由一个去理解另一个就好像试图从静态的动、植物分类去理解活生生的生态系统的动态性。反之亦然。

考虑对象聚合(aggregation)和相识(acquaintance)的差别以及它们在编译和运行时刻的表示是多么的不同。聚合意味着一个对象拥有另一个对象或对另一个对象负责。一般我们称一个对象包含另一个对象或者是另一个对象的一部分。聚合意味着聚合对象和其所有者具有相同的生命周期。

相识意味着一个对象仅仅知道另一个对象。有时相识也被称为“关联”或“引用”关系。相识的对象可能请求彼此的操作，但是它们不为对方负责。相识是一种比聚合要弱的关系，它只标识了对象间较松散的耦合关系。

在下图中，普通的箭头线表示相识，尾部带有菱形的箭头线表示聚合：



聚合和相识很容易混淆，因为它们通常以相同的方法实现。Smalltalk中，所有变量都是其他对象的引用，程序设计语言中两者并无区别。C++中，聚合可以通过定义表示真正实例的成员变量来实现，但更通常的是将这些成员变量定义为实例指针或引用；相识也是以指针或引用来实现。

从根本上讲，是聚合还是相识是由你的意图而不是由显式的语言机制决定的。尽管它们之间的区别在编译时刻的结构中很难看出来，但这些区别还是很大的。聚合关系使用较少且比相识关系更持久；而相识关系则出现频率较高，但有时只存在于一个操作期间，相识也更具动态性，使得它在源代码中更难被辨别出来。

程序的运行时刻结构和编译时刻结构存在这么大的差别，很明显代码不可能揭示关于系统如何工作的全部。系统的运行时刻结构更多地受到设计者的影响，而不是编程语言。对象和它们的类型之间的关系必须更加仔细地设计，因为它们决定了运行时刻程序结构的好坏。

许多设计模式（特别是那些属于对象范围的）显式地记述了编译时刻和运行时刻结构的差别。Composite(4.3)和Decorator(4.4)对于构造复杂的运行时刻结构特别有用。Observer(5.7)也与运行时刻结构有关，但这些结构对于不了解该模式的人来说是很难理解的。Chain of Responsibility(5.1)也产生了继承所无法展现的通信模式。总之，只有理解了模式，你才能清楚代码中的运行时刻结构。

1.6.7 设计应支持变化

获得最大限度复用的关键在于对新需求和已有需求发生变化时的预见性，要求你的系统设计要能够相应地改进。

为了设计适应这种变化、且具有健壮性的系统，你必须考虑系统在它的生命周期内会发生怎样的变化。一个不考虑系统变化的设计在将来就有可能需要重新设计。这些变化可能是类的重新定义和实现，修改客户和重新测试。重新设计会影响软件系统的许多方面，并且未曾料到的变化总是代价巨大的。

设计模式可以确保系统能以特定方式变化，从而帮助你避免重新设计系统。每一个设计模式允许系统结构的某个方面的变化独立于其他方面，这样产生的系统对于某一种特殊变化将更健壮。

下面阐述了一些导致重新设计的一般原因，以及解决这些问题的设计模式：

1) 通过显式地指定一个类来创建对象 在创建对象时指定类名将使你受特定实现的约束而不是特定接口的约束。这会使未来的变化更复杂。要避免这种情况，应该间接地创建对象。

设计模式：Abstract Factory(3.1)，Factory Method(3.3)，Prototype(3.4)。

2) 对特殊操作的依赖 当你为请求指定一个特殊的操作时，完成该请求的方式就固定下来了。为避免把请求代码写死，你将可以在编译时刻或运行时刻很方便地改变响应请求的方法。

设计模式：Chain of Responsibility(5.1)，Command(5.2)。

3) 对硬件和软件平台的依赖 外部的操作系统接口和应用编程接口(API)在不同的软硬件

平台上是不同的。依赖于特定平台的软件将很难移植到其他平台上，甚至都很难跟上本地平台的更新。所以设计系统时限制其平台相关性就很重要了。

设计模式：Abstract Factory(3.1)，Bridge(4.2)。

4) 对对象表示或实现的依赖 知道对象怎样表示、保存、定位或实现的客户在对象发生变化时可能也需要变化。对客户隐藏这些信息能阻止连锁变化。

设计模式：Abstract Factory(3.1)，Bridge(4.2)，Memento(5.6)，Proxy(4.7)

5) 算法依赖 算法在开发和复用时常常被扩展、优化和替代。依赖于某个特定算法的对象在算法发生变化时不得不变化。因此有可能发生变化的算法应该被孤立起来。

设计模式：Builder(3.2)，Iterator(5.4)，Strategy(5.9)，Template Method(5.10)，Visitor(5.11)

6) 紧耦合 紧耦合的类很难独立地被复用，因为它们是互相依赖的。紧耦合产生单块的系统，要改变或删掉一个类，你必须理解和改变其他许多类。这样的系统是一个很难学习、移植和维护的密集体。

松散耦合提高了一个类本身被复用的可能性，并且系统更易于学习、移植、修改和扩展。设计模式使用抽象耦合和分层技术来提高系统的松散耦合性。

设计模式：Abstract Factory(3.1)，Command(5.2)，Facade(4.5)，Mediator(5.5)，Observer(5.7)，Chain of Responsibility(5.1)。

7) 通过生成子类来扩充功能 通常很难通过定义子类来定制对象。每一个新类都有固定的实现开销(初始化、终止处理等)。定义子类还需要对父类有深入的了解。如，重定义一个操作可能需要重定义其他操作。一个被重定义的操作可能需要调用继承下来的操作。并且子类方法会导致类爆炸，因为即使对于一个简单的扩充，你也不得不引入许多新的子类。

一般的对象组合技术和具体的委托技术，是继承之外组合对象行为的另一种灵活方法。新的功能可以通过以新的方式组合已有对象，而不是通过定义已存在类的子类的方式加到应用中去。另一方面，过多使用对象组合会使设计难于理解。许多设计模式产生的设计中，你可以定义一个子类，且将它的实例和已存在实例进行组合来引入定制的功能。

设计模式：Bridge(4.2)，Chain of Responsibility(5.1)，Composite(4.3)，Decorator(4.4)，Observer(5.7)，Strategy(5.9)。

8) 不能方便地对类进行修改 有时你不得不改变一个难以修改的类。也许你需要源代码而又没有(对于商业类库就有这种情况)，或者可能对类的任何改变会要求修改许多已存在的其他子类。设计模式提供在这些情况下对类进行修改的方法。

设计模式：Adapter(4.1)，Decorator(4.4)，Visitor(5.11)。

这些例子反映了使用设计模式有助于增强软件的灵活性。这种灵活性所具有的重要程度取决于你将要建造的软件系统。让我们看一看设计模式在开发如下三类主要软件中所起的作用：应用程序、工具箱和框架。

1. 应用程序

如果你将要建造像文档编辑器或电子制表软件这样的应用程序(Application Program)，那么它的内部复用性、可维护性和可扩充性是要优先考虑的。内部复用性确保你不会做多余的设计和实现。设计模式通过减少依赖性来提高内部复用性。松散耦合也增强了一类对象与其他多个对象协作的可能性。例如，通过孤立和封装每一个操作，以消除对特定操作的依赖，

可在不同上下文中复用一个操作变得更简单。消除对算法和表示的依赖可达到同样的效果。

当设计模式被用来对系统分层和限制对平台的依赖性时，它们还会使一个应用更具可维护性。通过显示怎样扩展类层次结构和怎样使用对象复用，它们可增强系统的易扩充性。同时，耦合程度的降低也会增强可扩充性。如果一个类不过多地依赖其他类，扩充这个孤立的类还是很容易的。

2. 工具箱

一个应用经常会使用来自一个或多个被称为工具箱(Toolkit)的预定义类库中的类。工具箱是一组相关的、可复用的类的集合，这些类提供了通用的功能。工具箱的一个典型例子就是列表、关联表单、堆栈等类的集合，C++的I/O流库是另一个例子。工具箱并不强制应用采用某个特定的设计，它们只是为你的应用提供功能上的帮助。工具箱强调的是代码复用，它们是面向对象环境下的“子程序库”。

工具箱的设计比应用设计要难得多，因为它要求对许多应用是可用的和有效的。再者，工具箱的设计者并不知道什么应用使用该工具箱及它们有什么特殊需求。这样，避免假设和依赖就变得很重要，否则会限制工具箱的灵活性，进而影响它的适用性和效率。

3. 框架

框架(Framework)是构成一类特定软件可复用设计的一组相互协作的类 [Deu89,JF88]。例如，一个框架能帮助建立适合不同领域的图形编辑器，像艺术绘画、音乐作曲和机械CAD[VL90,Joh92]。另一个框架也许能帮助你建立针对不同程序设计语言和目标机器的编译器[JML92]。而再一个也许能帮助你建立财务建模应用 [BE93]。你可以定义框架抽象类的应用相关的子类，从而将一个框架定制为特定应用。

框架规定了你的应用的体系结构。它定义了整体结构，类和对象的分割，各部分的主要责任，类和对象怎么协作，以及控制流程。框架预定义了这些设计参数，以便于应用设计者或实现者能集中精力于应用本身的特定细节。框架记录了其应用领域的共同的设计决策。因而框架更强调设计复用，尽管框架常包括具体的立即可用的子类。

这个层次的复用导致了应用和它所基于的软件之间的反向控制 (inversion of control)。当你使用工具箱(或传统的子程序库)时，你需要写应用软件的主体并且调用你想复用的代码。而当你使用框架时，你应该复用应用的主体，写主体调用的代码。你不得不以特定的名字和调用约定来写操作地实现，但这会减少你需要做出的设计决策。

你不仅可以更快地建立应用，而且应用还具有相似的结构。它们很容易维护，且用户看来也更一致。另一方面，你也失去了一些表现创造性的自由，因为许多设计决策无须你来作出。

如果说应用程序难以设计，那么工具箱就更难了，而框架则是最难的。框架设计者必须冒险决定一个要适应于该领域的所有应用的体系结构。任何对框架设计的实质性修改都会大大降低框架所带来的好处，因为框架对应用的最主要贡献在于它所定义的体系结构。因此设计的框架必须尽可能地灵活、可扩充。

更进一步讲，因为应用的设计如此依赖于框架，所以应用对框架接口的变化是极其敏感的。当框架演化时，应用不得不随之演化。这使得松散耦合更加重要，否则框架的一个细微变化都将引起强烈反应。

刚才讨论的主要设计问题对框架设计而言最具重要性。一个使用设计模式的框架比不用

设计模式的框架更可能获得高层次的设计复用和代码复用。成熟的框架通常使用了多种设计模式。设计模式有助于获得无须重新设计就可适用于多种应用的框架体系结构。

当框架和它所使用的模式一起写入文档时，我们可以得到另外一个好处 [BJ94]。了解设计模式的人能较快地洞悉框架。甚至不了解设计模式的人也可以从产生框架文档的结构中受益。加强文档工作对于所有软件而言都是重要的，但对于框架其重要性显得尤为突出。学会使用框架常常是一个必须克服很多困难的过程。设计模式虽然无法彻底克服这些困难，但它通过对框架设计的主要元素做更显式的说明可以降低框架学习的难度。

因为模式和框架有些类似，人们常常对它们有怎样的区别和它们是否有区别感到疑惑。它们最主要的不同在于如下三个方面：

1) 设计模式比框架更抽象 框架能够用代码表示，而设计模式只有其实例才能表示为代码。框架的威力在于它们能够使用程序设计语言写出来，它们不仅能被学习，也能被直接执行和复用。而本书中的设计模式在每一次被复用时，都需要被实现。设计模式还解释了它的意图、权衡和设计效果。

2) 设计模式是比框架更小的体系结构元素 一个典型的框架包括了多个设计模式，而反之决非如此。

3) 框架比设计模式更加特例化 框架总是针对一个特定的应用领域。一个图形编辑器框架可能被用于一个工厂模拟，但它不会被错认为是一个模拟框架。而本书收录的设计模式几乎能被用于任何应用。当然比我们的模式更特殊的设计模式也是可能的（如，分布式系统和并发程序的设计模式），尽管这些模式不会像框架那样描述应用的体系结构。

框架变得越来越普遍和重要。它们是面向对象系统获得最大复用的方式。较大的面向对象应用将会由多层彼此合作的框架组成。应用的大部分设计和代码将来自于它所使用的框架或受其影响。

1.7 怎样选择设计模式

本书中有20多个设计模式供你选择，要从中找出一个针对特定设计问题的模式可能还是很困难的，尤其是当面对一组新模式，你还不怎么熟悉它的时候。这里给出几个不同的方法，帮助你发现适合你手头问题的设计模式：

- 考虑设计模式是怎样解决设计问题的 1.6节讨论了设计模式怎样帮助你找到合适的对象、决定对象的粒度、指定对象接口以及设计模式解决设计问题的几个其他方法。参考这些讨论会有助于你找到合适的模式。
- 浏览模式的意图部分 1.4节列出了目录中所有模式的意图(intent)部分。通读每个模式的意图，找出和你的问题相关的一个或多个模式。你可以使用表 1-1所显示的分类方法缩小你的搜查范围。
- 研究模式怎样互相关联 图1-1 以图形方式显示了设计模式之间的关系。研究这些关系能指导你获得合适的模式或模式组。
- 研究目的相似的模式 模式分类描述部分共有三章，一章介绍创建型模式，一章介绍结构型模式，一章介绍行为型模式。每一章都以对模式介绍性的评价开始，以一个小节的比较和对照结束。这些小节使你得以洞察具有相似目的的模式之间的共同点和不同点。
- 检查重新设计的原因 看一看从“设计应支持变化”小节开始讨论的引起重新设计的各

种原因，再看看你的问题是否与它们有关，然后再找出哪些模式可以帮助你避免这些会导致重新设计的因素。

- 考虑你的设计中哪些是可变的 这个方法与关注引起重新设计的原因刚好相反。它不是考虑什么会迫使你的设计改变，而是考虑你想要什么变化却又不会引起重新设计。最主要的一点是封装变化的概念，这是许多设计模式的主题。表 1-2列出了设计模式允许你独立变化的方面，你可以改变它们而又不会导致重新设计。

表1-2 设计模式所支持的设计的可变方面

目 的	设计模式	可变的方面
创建	Abstract Factory(3.1) Builder(3.2) Factory Method(3.3) Prototype(3.4) Singleton(3.5)	产品对象家族 如何创建一个组合对象 被实例化的子类 被实例化的类 一个类的唯一实例
结构	Adapter(4.1) Bridge(4.2) Composite(4.3) Decorator(4.4) Facade(4.5) Flyweight(4.6) Proxy(4.7)	对象的接口 对象的实现 一个对象的结构和组成 对象的职责，不生成子类 一个子系统的接口 对象的存储开销 如何访问一个对象；该对象的位置
行 为	Chain of Responsibility(5.1) Command(5.2) Interpreter(5.3) Iterator(5.4) Mediator(5.5) Memento(5.6) Observer(5.7) State(5.8) Strategy(5.9) Template Method(5.10) Visitor(5.11)	满足一个请求的对象 何时、怎样满足一个请求 一个语言的文法及解释 如何遍历、访问一个聚合的各元素 对象间怎样交互、和谁交互 一个对象中哪些私有信息存放在该对象之外，以及在什么时候进行存储 多个对象依赖于另外一个对象，而这些对象又如何保持一致 对象的状态 算法 算法中的某些步骤 某些可作用于一个（组）对象上的操作，但不修改这些对象的类

1.8 怎样使用设计模式

一旦你选择了一个设计模式，你怎么使用它呢？这里给出一个有效应用设计模式的循序渐进的方法。

- 大致浏览一遍模式 特别注意其适用性部分和效果部分，确定它适合你的问题。
- 回头研究结构部分、参与者部分和协作部分 确保你理解这个模式的类和对象以及它们是怎样关联的。
- 看代码示例部分，看看这个模式代码形式的具体例子 研究代码将有助于你实现模式。
- 选择模式参与者的名字，使它们在应用上下文中有意义 设计模式参与者的名字通常

过于抽象而不会直接出现在应用中。然而，将参与者的名字和应用中出现的名字合并起来是很有用的。这会帮助你在实现中更显式的体现出模式来。例如，如果你在文本组合算法中使用了Strategy模式，那么你可能有名为SimpleLayoutStrategy或TeXLayoutStrategy这样的类。

5) 定义类 声明它们的接口，建立它们的继承关系，定义代表数据和对象引用的实例变量。识别模式会影响到的你的应用中存在的类，做出相应的修改。

6) 定义模式中专用于应用的操作名称 这里再一次体现出，名字一般依赖于应用。使用与每一个操作相关联的责任和协作作为指导。还有，你的名字约定要一致。例如，可以使用“Create”前缀统一标记Factory方法。

7) 实现执行模式中责任和协作的操作 实现部分提供线索指导你进行实现。代码示例部分的例子也能提供帮助。

这些只对你一开始使用模式起指导作用。以后你会有自己的设计模式使用方法。

关于设计模式，如果不提一下它们的使用限制，那么关于怎样使用它们的讨论就是不完整的。设计模式不能够随意使用。通常你通过引入额外的间接层次获得灵活性和可变性的同时，你也使设计变得更复杂并/或牺牲了一定的性能。一个设计模式只有当它提供的灵活性是真正需要的时候，才有必要使用。当衡量一个模式的得失时，它的效果部分是最能提供帮助的，如表1-2所示。