Thinking in Patterns

Problem-Solving Techniques using Java

Bruce Eckel

President, MindView, Inc.

Revision 0.9

翻译：刘晓伟

译序

首先应该感谢 Bruce Eckel 一如既往地无偿开放他的著作的电子版本。本书的英文原版和所有章节的配套源码，您都可以从 Bruce Eckel 的网站上下载：
http://www.mindview.net/Books/TIPatterns/

还要特别感谢在我最初开始翻译这本书的时候给予我肯定和鼓励的的朋友们（http://blog.csdn.net/lxwde/archive/2004/08/04/60395.aspx）。没有你们关注和鼓励我也不会有动力坚持把这个译本完成。

需要说明的是，这是一本原文尚未最终定稿的书籍，Bruce Eckel 最后修改的时间是 2003 年 5 月，版本号是 0.9。尽管如此，除了个别词句和尚未动笔的章节（比如讲述 Momento 模式的那一章），此书英文版的绝大部分已经比较完整，基本不会影响阅读。我会关注原书的版本更新，及时调整相应的译文。

如果您对这个译本有任何意见或者建议，您可以通过 liuxiaoweide@gmail.com 和我联系，也可以直接到下面这个地址给我留言：http://blog.csdn.net/lxwde

liu xiao wei

2005.07.16

目录

前言

这本书里的内容是由我（大多数时候和 Bill Venners 一起）在过去几年里开办的一个讲座课程发展而来的。Bill 和我举办这个讲座已经有好多期了，在过去几年里随着我们自身学习到更多关于模式以及如何办好讲座班的知识，这个课程的内容也发生了很大的变化。

在这个过程中，我们积累了足够我们俩各自开一个班的信息，但我们不愿意这么做，因为我们俩一起上课实在是太有意思了。我们在美国不计其数的地方举办过讲座，我们还在布拉格（Prague）开办讲座（我们试图在每年春天和其它讲座一起举行一次小型的会议）。我们还曾把它办成过现场（on-site）讲座。

在这里要向这几年参与这些讲座的人表示衷心的感谢，他们帮助我理清和提炼了很多想法。我希望在接下来的许多年里能够通过本书和举办讲座继续总结和发展类似的想法。

这本书不会就此打住。经过一番慎重考虑之后，我打算一开始就把《Thinking in Python》写成本书的 Python 版本，而不是介绍 Python 语言。（已经有很多好的教材来介绍这门出色的语言了）。我觉得这么做比费力再写一个语言手册激动人心的多（我要向那些原本希望我把《Thinking in Python》 写成一本 Python 教程的人道歉）。

绪论

本书是关于设计的，多年来我一直从事这项工作。基本上说，从我第一次试着阅读《设计模式》（Gamma, Helm,Johnson & Vlissides, Addison-Wesley, 1995，通常被称作“四人帮[1]（Gang of Four）”或者 GOF）这本书开始。

在《Thinking in C++》第一版里有一章是专门讲设计模式的，后来我把这一章放到了《Thinking in C++》第二版的卷二。而且在《Thinking in Java》的第一版里你也能找到一章是关于模式的（第二版里我把它拿掉了，因为那本书篇幅太长了，更主要的原因是我打算写现在这本书）。

本书不是一本介绍性的书籍。我假定你在阅读本书之前通读过《Thinking in Java》或者与之相当的其它教材。

此外，我还假定你对 Java 的语法有一定程度的了解。你应该对“对象（object）”及其内涵有深刻的理解，包括多态（polymorphism）。重申一遍，这些东西在《Thinking in Java》里都有讲解。

从另一方面讲，阅读本书的过程中你可以通过研究“对象”在不同情形下的应用学到很多关于面向对象编程的知识。如果你对“对象”只有一些初步的认识，在理解本书讲到的这些设计方法的过程中，你的这些认识会不断加深。

Y2K 综合症

本书的副标题是“解决问题的技术”，所以在这里有必要提一下编程领域里的一个大陷阱：过早优化。每次我提出这个看法的时候，大家实际上都是同意的。与此同时，每个人似乎都在心里保留自己的 special case “只是我这次碰巧遇到的是一个特例”。

我之所以把这叫做 Y2K 综合症与那种特殊知识有很大关系。电脑对很多人来说是神秘的。所以当有人宣称那些愚蠢的程序员忘了使用足够长的数字来保存 1999 年以后的日期的时候，一下子大家都变成电脑专家了——“这些东西本来一点都不难，如果我早注意到这个浅显的问题的话。”拿我自己来说吧，我的背景最初是计算机工程，一开始我是搞嵌入式系统编程的。因此，我知道许多嵌入式系统根本没有日期或者时间的概念，即便是有，它们也不会被用于任何重要的运算场合。我被明确告知所有的嵌入式系统都会在 2000 年 1 月 1 号荡机。但是据我所知，那天只有那些预言灾难必将降临的家伙们的大脑内存丢失了——似乎他们自己从来就没有说过那些话。

---

[1] 这是半开玩笑地援引毛泽东死后发生在中国的一个事件，当时包括毛的遗孀在内的四个人合伙发起了权利角逐，后来中共把他们四人叫做这个名字。

我的意思是说，我们非常容易陷入一种思维定式，即认为我们部分或完全理解的某一算法或者代码段会成为整个系统的瓶颈，而（做出这种判断）仅仅因为我们通过对这部分已知代码段的想象就认定它肯定比我们不知道的那些代码段效率低。但是除非你作过实际测试，通常是 profiler，否则你不可能真正知道实际的运行情况。即使你已经知道某个代码段效率非常低，但是别忘了大部分程序 90％的时间运行的都只是占整个程序 10％不到的代码。所以除非你认为效率低下的这部分代码属于那 10％，否则对它们的优化是无关紧要的。

“97%的情况下我们不需要考虑细微的运行效率问题，“过早优化”是所有麻烦的根源”——Donald Kruth

上下文和组合（Context and composition）

你将会看到，设计模式文献里用的最多的一个术语就是“上下文（context）”。实际上，“设计模式”通常的一个定义就是“针对某个上下文特定问题的解决方案”。GOF 总结的这些模式经常会包含一个“上下文对象”，使用这些模式的程序员直接与这个上下文对象打交道。我曾经一度觉得这些对象似乎在许多模式中居于主导地位，于是我就想，它们到底是用来干嘛的。

“上下文对象”经常充当“外观（facade）”的角色，用以掩盖模式其它部分的复杂性。此外，它还经常是控制整个模式运转的控制器。最初，我以为这些东西对于实现、运用和理解模式并不是十分重要的，但是，我记得 GOF 书里有一句话给人印象很深：“优先使用组合而不是继承（prefer composition to inheritance）。”上下文对象使你可以在组合（composition）里使用模式。或许，这才是它最大的好处。

关于“已检测异常（checked exceptions）”

1)异常的价值在于它统一的错误报告机制：这是一种标准的错误报告机制，而不是像 C 里面的那些**非强制性**的大杂烩似的错误报告方法。（而 C++只是把异常加入 C 的混合错误报告机制，而没有使异常成为唯一的错误报告方法）Java 相对与 C++而言有一个巨大的优势就是，异常是报告错误的唯一方法。

2)上一段的“非强制性”这个词是说另外一个问题。从理论上说，如果编译器强制程序员处理或者按照规范传递异常，那么程序员的注意力总是会被带到可能出错的地方，进而用适当的方法来处理这些错误。我认为，问题是这只是一个未被验证过的假设，而我们是把自己当成语言的设计者凭主管臆断来做这个假设的。我的理论是，当有人想做一件事情，而你却不停的用令人厌烦的手段催促他，他就会用取巧的方法掩盖那些令人厌烦的东西，天知道以后他们会不会回头重新拿掉那些取巧的东西。我发现我在《Thinking in Java》第一版里就是这么干的。

```
...
} catch (SomeKindOfException e) {}
```

（我写了上面那些代码，）然后多多少少就把它忘掉了，直到重写的时候才想起来。有多少人会认为这是一个好例子而效仿它呢？Martin Fowler 也注意到了这个问题，他发现人们屏蔽掉这些异常后就再也不管了。Checked exceptions 所带来的负面问题使它背离了设计它的初衷，当你自己试验的时候你就会发现这些问题。（现在我相信 checked exceptions 是个试验性的东西，某些人认为它是个好主意，直到不久以前我还相信它确实是个不错的想法）。

当我开始使用 Python 以后，所有的异常都会显现，没有一个会突然消失。你可以扑获你所关心的异常，但是你不会被强制要求写一大坨代码仅仅为了让异常能够传递下去。它们会在你想要捕获他们的地方出现。如果碰到最糟糕的情况，你忘了扑获异常，他们会暂时躲起来（以后会提示你）而不会彻底消失。现在我相信 Checked exceptions 是在鼓励人们让异常彻底消失。而且它使得代码可读性更差。

最后，我想我们必须意识到异常本身就是试验性的东西，在假定所有 Java 的异常机制都是正确之前，我们必须更谨慎的审视它。我相信，只采用一种统一的错误处理机制是非常棒的，我还相信通过采用一个分离的通道（异常处理机制）来传递异常也是好的。我记得最初关于 C++的异常处理机制的一个主张，就是它可以使程序员把工作代码和错误处理代码区分开来。但是，我觉得 checked exceptions 似乎没有这么做；恰恰相反，它总是想往你的工作代码里硬塞一些东西，这不能不说是一个倒退。我使用 Python 的经验更加坚定了我的这个看法，除非我对这个问题的看法有所改变，否则我倾向于在我的 Java 代码里用更多的运行时异常（Runtime Exceptions） 。

模式概念

“设计模式帮助你从别人的成功经验而不是你自己的失败那里学到更多东西[2]”。

或许，面向对象设计领域迈出的最重要的一步就是“设计模式”运动，这一运动被记录整理成《设计模式》一书[3]。那本书展示了 23 种针对特定类型问题的解决办法。本书将用例子来介绍设计模式的基本概念。这将会激起你阅读 Gamma 等人所著的《设计模式》一书的兴趣，该书已成为从事面相对象编程的程序员的重要的和几乎必需的“词汇表”。

本书后面的章节用一个例子描述了设计演化的过程，从最初的解决方案开始，按照合理的推理和步骤，最终演化成更为合理的设计。这个例子程序（一个模拟废品分拣（trash sorting） 的例子程序）随着时间不断演化，你可以把它看作是一个原型，你自己做设计的时候也是从一个适用于某一特定问题的解决方案开始，逐渐演化成一个灵活的能够解决某一类问题的方法。

什么是模式？

一开始，你可以把模式想象成一种特别巧妙和敏锐的用以解决某类特定问题的方法。更确切地说，许多人从不同角度解决了某个问题，最终大家提出了最通用和灵活的解决办法。这个问题可能是你以前见过并解决过的，但是你的方法可能比不上你将看到的模式所体现的方法来的完整。

尽管它们被称作“设计模式”，实际上它们没有仅仅限于设计的范畴。模式看起来似乎跟传统的分析、设计和实现相去甚远；恰恰相反，模式体现的是程序整体的构思，所以有时候它也会出现在分析或者是概要设计阶段。这是个有趣的现象，因为模式可以由代码直接实现，所以你可能不希望在详细设计或编码以前使用模式，（实际上在详细设计和编码之前你可能都不会意识到你需要某个特定的模式）。

模式的基本概念也可以看作是设计的基本概念：即增加一个抽象层。无论什么时候，当你想把某些东西抽象出来的时候，实际上你是在分离特定的细节，这么做的一个有说服力的动机就是把变化的东西从那些不变的东西里分离出来。这个问题的另一种说法是，当你发现程序的某一部分由于某种原因有可能会变化的话，你会希望这些变化不会传播给程序代码的其它部分。这么做不但使程序更容易维护，而且它通常使程序更容易理解（这将降低成本）。

很多情况下，对于能否设计出优雅和容易维护的系统来说，最难的就是找到“一系列变化的东西。”（这里，“vector”指最大梯度，而不是指容器类。）这就意味

---

[2] 这是 Mark Johnson 说的.

[3] 但是得警告你一下：那些例子都是 C++写的。

着最重要的是找出你的系统里变化的部分，或者说是找到成本最高的部分。一旦你找出了这一系列变化，你就可以以之为焦点来构造你的设计。

设计模式的目的就是为了把代码里变化的那一部分分离出来。如果你这么看待设计模式，你会发现本书实际上已经讲了一些设计模式了。比如说，你可以认为继承就是一种设计模式（只不过他是由编译器来实现罢了）。通过继承你可以使拥有相同接口（这些接口是不变的）的对象具有不通的行为（这就是变化的部分），组合（composition）也可以被认为使一种模式，它可以静态或者动态地改变你用以实现某个类的对象，从而改变这个类的行为。

《设计模式》这本书里讲的另外一个模式：迭代器（iterator），你也已经见到过了。（Java1.0 和 1.1 自以为是的把它叫做**枚举器（Enumeration）**；而 Java 2 的容器类用了“迭代器”这个术语）。这种方法在你需要遍历和选定某些元素的时候隐藏了容器类的特定实现。通过使用迭代器，你可以写出针对某一序列元素的泛型（generic）代码，而不用管这一序列元素是如何生成的。这样，这些泛型代码就可以用于所有能产生迭代器的容器类了。

模式范畴 （pattern taxonomy）

设计模式兴起以后，有人担心“设计模式”这个词被滥用，因为人们开始把所有它们觉得好的东西都说成是设计模式。慎重考虑之后，我把这一系列相近的东西区分为一下几个范畴：

1. **惯用法（Idiom）：**如何针对某种特定的语言，为了实现某种特定的功能，书写代码。比如说，就像你如何在 C 里面遍历一个数组（你不会越界访问数组）。

2. **特定设计 （Specific Design）：**为了解决某一特定问题，所提出的某一方法。它可能是一个很巧妙的设计，但是我们并不指望它成为通用的解决方案。

3. **标准设计 （standard design）：**为了解决某一类问题，所提出的更为通用的一种设计方案，通常通过复用（reuse）来满足它的通用性。

4. **设计模式 （Design pattern）：**用以解决某一大类相似的问题。通常是我们多次应用某一标准设计后，发现某一通用模式可以从这些应用里抽象出来。

我觉得这么区分可以帮助我们以更全面的眼光来看待这些范畴，从而找到它们适用的地方。但是，这并不意味着某一个范畴就比另外一个好。试图把每一个解决方案都泛化成（generalize）设计模式是没有意义的，这么做纯粹是浪费时间而且也不大可能发现（新的）模式。模式通常是随着时间的推移在不经意间产生的。

上面那些范畴的分类里还可以再加入分析模式（Analysis Pattern）和架构模式（Architectural Pattern）。

设计原则

（从幻灯片里搬到这儿来了）

当我通过自己的电子通讯（newsletter）征集观点的时候[4]，很多人会给我许多有用的建议，但是它们不同于上面讨论的那些范畴。我意识到，一个有关设计原则的列表至少是和设计构造同等重要的。但是，还有另外一个原因，有了这些设计原则你就可以用它们来对自己所建议的设计提出质疑，并检验这些设计的好坏。

- **尽量不要让人感到奇怪**（不要对这条原则感到奇怪）。
- **使一般问题简单化，罕见问题行得通**。
- **一致性（consistency）**。这个对我来说已经很清楚，尤其是有了 Python 以后：如果你强加给程序员越多条条框框，而这些条条框框对于解决手头问题没多大用处，程序员编程的效率就越低。而且这个负面影响不是线性增长的，而是成几何级数增长的。
- **得墨忒耳（Demeter）法则**：“不要跟陌生人讲话。”一个对象最好只引用它自己，它自己的属性和它自己方法的参数。
- **减法（Subtraction）原则**：当你不能从一个设计里去掉任何东西的时候，这个设计就算完成了。
- **先简单（Simplicity）后通用（generality）**[5]。（这是奥卡姆剃刀原则（Occam’s Razor）的另一种说法。它的原话是“最简单的就是最好的”）。一个常见的问题是，许多框架在设计的时候总是一味强调通用性而丝毫没有考虑到实际系统。这导致一大堆使人眼花缭乱的选项，而这些选项要么经常用不到，要么被误用或者根本就没什么用处。而且，大多数开发人员开发的都是专用系统，很多时候寻求通用性对他们来说并没有太大用处。寻求通用性最好的做法是通过理解现有的完善成熟的特定例子。因此，这条准则使得在简单设计和通用设计都可行的情况下选择简单设计。当然，简单设计正好就是一个通用设计也是完全可能的。
- **自反性（Reflexivity）**（我推荐这个术语）。每个类一个抽象，每个抽象对应一个类。这个准则也可以称作同构（isomorphism）。
- **独立性（Independence）**，或者叫正交性（Orthogonality）。独立表达每一个单独的想法。这条准则是分离（Separation），封装（Encapsulation）和

[4] 这是一个免费发行的 email。可以到 www.BruceEckel.com 订阅.

[5] 来自 Kevlin Henney 的一封 email.

变化（Variation）的补充。它是“低耦合高内聚（Low-Coupling-High-Cohesion）”思想的一部分。

- **一次且只能出现一次（Once and once only）**：如果两段代码干的是同一件事情，应该避免逻辑上和结构上的重复。

最初思考这些设计原则的的时候，我只是希望总结出一两条，以便你在分析一个问题的时候直接就能用上。然而，你可以用上面列表里的其它准则作为对照表来检查和分析你的设计。

模式分类

《设计模式》讨论了 23 种不同的模式，按照目的把它们分成三类（这些目的围绕某一可变化的特定情况）。这 3 种分类分别是：

1. **创建（Creational）型**：如何创建一个对象。这通常包括分离对象创建的细节，这样你的代码就不依赖于对象的类型，而且当加入新的类型的对象时也不必改代码。下面将要讲到的单件模式（Singleton）就是一种创建型模式，在本书稍后的章节你还会看到工厂方法（Factory Method）和原型模式（Prototype）的例子。

2. **结构（structural）型**：设计出满足某些工程里特定约束的对象。它的工作原理是这样的：一组对象与其它对象相关联，当系统发生变化的时候，这些对象之间的关联关系不变。

3. **行为（Behavioral）型**：指一个程序里处理一系列特定类型操作的对象。它们封装了一系列操作，比如某种语言的解释（interpreting a language），填表单，遍历一个序列（像迭代器（iterator）那样），或者实现了某种算法。本书以观察者模式（Observer）和访问者模式（Visitor）来举例说明。

《设计模式》专门有一部分用一个或多个例子来说明这 23 个模式，这些例子基本上都是用 C++写的(rather restricted C++, at that)，有时候也用 Smalltalk。（你会发现这并不是个大问题，因为你可以毫不费力的把那些概念从 C++或 Smalltalk 转换到 Java）。本书将用面向 Java 的思想重新讲解《设计模式》里的许多模式，因为更换语言带来了对模式表示方法和理解的变化。但是，这里不会重复 GOF 的那些例子，因为我相信通过努力，我们可以设计出更容易让人理解的例子。我的目标是使你知道模式是干什么用的并且为什么它们这么重要。

多年来接触这些东西，我逐渐意识到模式本身用到的其实都是很基本的组织结构，而不是像《设计模式》里所描述的那么复杂（比那些要简单简单的多）。我之所以这么说是因为，大多说设计模式（不限于《设计模式》所总结的那些），就其实现所应用的结构而言，是存在很大的相似性的。尽管我们总是试图避免使用实现，取而

代之以接口，但在这里，我认为用“结构要素”（structural principles）来讲解，会使大家更容易理解这些模式。我试图以这些模式结构上的相似性来给它们重新归类，而不采用《设计模式》里的那些分类。

然而，后来我发现，根据这些模式所解决的问题来给他们归类会更有用。我期望，这种归类方法与 Metsker 在《Java 设计模式手册》（《 Design Patterns Java Workshop》(Addison-Wesley 2002)）里所提出的根据目的（intent）来归类的方法会有所不同，这种不同虽然微妙但却是重大的。如果说模式都是用我说的这种方法归类的的话，我希望读者（在学完本书后）能够辨别出问题所在并找到一个解决方案。

在不断“重构本书（book refactoring）”的过程中，我意识到如果我更改过某个地方，那么很有可能以后我还会改那个地方（当然会有一个次数限制），所以我去掉了所有的章节号索引，这样便于以后的改动（这就是鲜为人知的“无章节号” 模式（numberless chapter pattern））：

开发所面临的挑战

关于程序开发，UML 过程，极限编程（Extreme Programming）。

评估有价值么（Is evaluation valuable）？能力成熟度模型（The Capability Immaturity Model）：

Wiki Page: http://c2.com/cgi-bin/wiki?CapabilityImMaturityModel

Article: http://www.embedded.com/98/9807br.htm

关于成对编程的研究(Pair programming research):

http://collaboration.csc.ncsu.edu/laurie/

单元测试

在本书的早些时候的一个版本里，我认定单元测试是必不可少的（和我其它所有的书一样），但是那时候的 JUnit 用起来实在是太不爽了。当时，我自己写了一套单元测试的框架，它使用 Java 的反射（reflection）技术使单元测试所必需的一些语法得以简化。

《Thinking in Java》的第三版里，我开发了另外一套单元测试的框架用以测试那本书中例子的输出结果。

在此期间，JUnit 增加了支持单元测试的语法，这些语法跟我在本书以前版本里用的那些如出一辙。我不知道自己对那些改动起到了多大的影响，但是我很高兴 JUnit

作了这些改动，这样我就不用维护我自己的测试框架了（但你还是可以在这儿找到它），我所需要做的只是向大家推荐这个既成事实的标准。

我在《Thinking in Java》第三版第十五章里介绍和描述了 JUnit 的编程风格，我认为那些代码是“最好的实践”（最主要是因为它们很简单）。与本书有关的单元测试，那一章里提供了足够多的介绍（但是，本书通常不会在正文里包含单元测试的代码）。当你下载本书配套代码的时候，你会发现，绝大多数情况下那些代码都是包含单元测试的。

测试代码的位置

（From Bill）

public：在 test 子目录里；不同的 package（不要包含在 jar 里）。

Package access：同一个 package，一个库里的子目录（不要包含在 jar 里）。

Private access：（白盒测试）。嵌套类，strip out，或者 JUnit 插件。

简单化（Simplifying Idioms）

在研究复杂技术之前，了解一下使代码简单明了的基本方法是很有帮助的。

信使（Messenger）

最普通的方法就是通过信使（messenger），它简单的将信息打包到一个用于传送的对象，而不是将这些信息碎片单独传送。注意，如果没有信使，translate( )的代码读起来会相当混乱。

```
//: simplifying:MessengerDemo.java
package simplifying;
import junit.framework.*;

class Point { // A messenger
 public int x, y, z; // Since it's just a carrier
 public Point(int x, int y, int z) {
  this.x = x;
  this.y = y;
  this.z = z;
 }
 public Point(Point p) { // Copy-constructor
  this.x = p.x;
  this.y = p.y;
  this.z = p.z;
 }
 public String toString() {
  return "x: " + x + " y: " + y + " z: " + z;
 }
}

class Vector {
 public int magnitude, direction;
 public Vector(int magnitude, int direction) {
  this.magnitude = magnitude;
  this.direction = direction;
 }
}
```

```
class Space {
  public static Point translate(Point p, Vector v) {
    p = new Point(p); // Don't modify the original
    // Perform calculation using v. Dummy calculation:
     p.x = p.x + 1;
    p.y = p.y + 1;
    p.z = p.z + 1;
    return p;
   }
}

public class MessengerDemo extends TestCase {
  public void test() {
    Point p1 = new Point(1, 2, 3);
    Point p2 = Space.translate(p1, new Vector(11, 47));
    String result = "p1: " + p1 + " p2: " + p2;
    System.out.println(result);
    assertEquals(result,
      "p1: x: 1 y: 2 z: 3 p2: x: 2 y: 3 z: 4");
   }
  public static void main(String[] args) {
    junit.textui.TestRunner.run(MessengerDemo.class);
   }
} ///:~
```

因为 messenger 只是用来传送数据，它所传送的数据通常声明为公有的，以便于存取。但是，你可以根据自己的需要把它们声明成私有的。

集合型参数（collecting parameter）

collecting parameter 是 messenger 的兄弟，messenger 传参数给某个方法，而 collecting parameter 从这个方法获取信息。一般说来，这通常会用在 collecting parameter 传给多个方法（multiple methods）的情况下，就像一只传粉的蜜蜂。

容器是一种特别有用的 collecting parameter，因为它本来就是用来动态添加对象的。

```
//: simplifying:CollectingParameterDemo.java
package simplifying;
import java.util.*;
```

```
import junit.framework.*;

class CollectingParameter extends ArrayList {}

class Filler {
 public void f(CollectingParameter cp) {
  cp.add("accumulating");
 }
 public void g(CollectingParameter cp) {
  cp.add("items");
 }
 public void h(CollectingParameter cp) {
  cp.add("as we go");
 }
}

public class CollectingParameterDemo extends TestCase {
 public void test() {
  Filler filler = new Filler();
  CollectingParameter cp = new CollectingParameter();
  filler.f(cp);
  filler.g(cp);
  filler.h(cp);
  String result = "" + cp;
  System.out.println(cp);
  assertEquals(result,"[accumulating, items, as we go]");
 }
 public static void main(String[] args) {
  junit.textui.TestRunner.run(
   CollectingParameterDemo.class);
 }
} ///:~
```

Collecting parameter 必须支持通过某些方法设置或者插入一些值。根据这个定义，信使可以当作 collecting parameter 来用，前提是 collecting parameter 是由它所传递给的方法来修改的。

对象数量（Object quantity）

这里描述的两种模式都可以单独用来控制对象的数量。

单件（Singleton）实际上可以认为是对象池（Object Pool）的一个特例，但是对象池的应用和单件是如此的不同，将二者区分开来对待是非常必要的。

单件（Singleton）

Singleton 提供一种方法使得某一特定类型存在一个，并且只能是一个对象。它可能是最简单的模式了。Singleton 应用的一个重要方面是提供一个全局的存取点。Singleton 是 C 里面全局变量的一个替代方法。

除此之外，单件还常常提供注册（registry）和查找（lookup）的功能——你可以从单件那里找到其它对象的引用。

Singleton 可以在 java 库里找到，但是下面提供了一个更直接的例子。

```
//: singleton:SingletonPattern.java
// The Singleton design pattern: you can
// never instantiate more than one.
package singleton;
import junit.framework.*;
// Since this isn't inherited from a Cloneable
// base class and cloneability isn't added,
// making it final prevents cloneability from
// being added through inheritance:
final class Singleton {
  private static Singleton s = new Singleton(47);
  private int i;
  private Singleton(int x) { i = x; }
  public static Singleton getReference() {
    return s;
  }
  public int getValue() { return i; }
  public void setValue(int x) { i = x; }
}



public class SingletonPattern extends TestCase {
  public void test() {
```

```
    Singleton s = Singleton.getReference();
    String result = "" + s.getValue();
    System.out.println(result);
    assertEquals(result, "47");
    Singleton s2 = Singleton.getReference();
    s2.setValue(9);
    result = "" + s.getValue();
    System.out.println(result);
    assertEquals(result, "9");
    try {
      // Can't do this: compile-time error.
      // Singleton s3 = (Singleton)s2.clone();
    } catch(Exception e) {
      throw new RuntimeException(e);
    }
  }
  public static void main(String[] args) {
    junit.textui.TestRunner.run(SingletonPattern.class);
  }
} ///:~
```

Singleton 模式的关键是防止用户以其它任何方式创建对象，而只能用你所提供的方式。所有的构造函数必须被声明为私有的(private)，而且必须至少声明一个构造函数，否则编译器就会以 package 权限帮你创建一个默认的构造函数。

在这一点上，你自己决定如何创建对象。上面的例子里，对象是静态（statically）创建的，但是你也可以等到客户端提出请求需要创建对象时才创建。不管是哪种情况，（创建的）对象必须以私有方式（privately）存储，而通过公有方法（public methods）来访问。上面的例子里，getReference( )返回对单件对象的引用。剩下的接口（getValue( )和 setValue( )）是常规的类接口。

另外，Java 允许使用克隆（cloning）来创建对象。这个例子里，把这个类声明为 final 就是为了防止通过克隆方法创建对象。因为 Singleton 是直接从 Object 继承下来的，由于 clone( )是受保护的（protected）方法因而不能用来（复制对象），如果这么做就会导致编译时错误。

但是，如果以 public 的方式继承了一个重载了 clone( )方法的类，并且实现了 Cloneable 接口的话，为了防止对象被克隆，就必须重载 clone( )方法并抛出一个 CloneNotSupportedException 异常。这一点在《Thinking in Java》第二版的附录 A 里也讲到了。（你也可以重载 clone( )方法使它只返回 this，但是这样做有一定的欺

骗性，客户端程序员会想当然的认为他是在克隆（复制）那个对象，但实际上他处理的还是原来那个对象。）实际上，这么说并不确切，即使是上面所说的情况，你仍然可以使用反射（reflection）来调用 clone( )方法（真的是这样么？因为 Clone( )方法是受保护的（protected），所以我也不那么肯定。如果真是这样的话，那就必须得抛出 CloneNotSupportedException 异常，这是保证对象不被克隆的唯一方法了。）

练习

1. SingletonPattern.java 总是创建一个对象，即使这个对象从来不会被用到。请用延迟初始化(Lazy Initialization)的方法修改这个程序，使单件对象只在它第一次被用到的时候才创建。

2. 写一个注册/查找（registry/lookup）的服务，它可以接受一个 Java 接口（interface）并且产生一个关于实现了那个接口的一个对象的引用。

对象池（Object pool）

并没有限制说只能创建一个对象。这种技术同样适用于创建固定数量的对象，但是，这种情况下，你就得面对如何共享对象池里的对象这种问题。如果共享对象很成问题得话，你可以考虑以签入（check-in）签出（check-out）共享对象作为一种解决方案。比如，就数据库来说，商业数据库通常会限制某一时刻可以使用的连接的个数。下面这个例子就用对象池（object pool）实现了对这些数据库连接的管理。首先，对象池对象（a pool of objects）的基本管理是作为一个单独的类来实现的。

```
//: singleton:PoolManager.java
package singleton;
import java.util.*;

public class PoolManager {
 private static class PoolItem {
  boolean inUse = false;
  Object item;
  PoolItem(Object item) { this.item = item; }
 }
 private ArrayList items = new ArrayList();
 public void add(Object item) {
  items.add(new PoolItem(item));
 }
 static class EmptyPoolException extends Exception {}
 public Object get() throws EmptyPoolException {
  for(int i = 0; i < items.size(); i++) {
```

```
   PoolItem pitem = (PoolItem)items.get(i);
   if(pitem.inUse == false) {
    pitem.inUse = true;
    return pitem.item;
   }
  }
  // Fail early:
  throw new EmptyPoolException();
  // return null; // Delayed failure
 }
 public void release(Object item) {
  for(int i = 0; i < items.size(); i++) {
   PoolItem pitem = (PoolItem)items.get(i);
   if(item == pitem.item) {
    pitem.inUse = false;
    return;
   }
  }
  throw new RuntimeException(item + " not found");
 }
} ///:~

//: singleton:ConnectionPoolDemo.java
package singleton;
import junit.framework.*;

interface Connection {
 Object get();
 void set(Object x);
}

class ConnectionImplementation implements Connection {
 public Object get() { return null; }
 public void set(Object s) {}
}



class ConnectionPool { // A singleton
```

```
 private static PoolManager pool = new PoolManager();
 public static void addConnections(int number) {
  for(int i = 0; i < number; i++)
   pool.add(new ConnectionImplementation());
 }
 public static Connection getConnection()
  throws PoolManager.EmptyPoolException {
   return (Connection)pool.get();
  }
  public static void releaseConnection(Connection c) {
   pool.release(c);
  }
}

public class ConnectionPoolDemo extends TestCase {
 static {
  ConnectionPool.addConnections(5);
 }
 public void test() {
  Connection c = null;
  try {
   c = ConnectionPool.getConnection();
  } catch (PoolManager.EmptyPoolException e) {
   throw new RuntimeException(e);
  }
  c.set(new Object());
  c.get();
  ConnectionPool.releaseConnection(c);
 }
 public void test2() {
  Connection c = null;
  try {
   c = ConnectionPool.getConnection();
  } catch (PoolManager.EmptyPoolException e) {
   throw new RuntimeException(e);
  }

  c.set(new Object());
```

```
  c.get();
  ConnectionPool.releaseConnection(c);
 }
 public static void main(String args[]) {
  junit.textui.TestRunner.run(ConnectionPoolDemo.class);
 }
} ///:~
```

练习

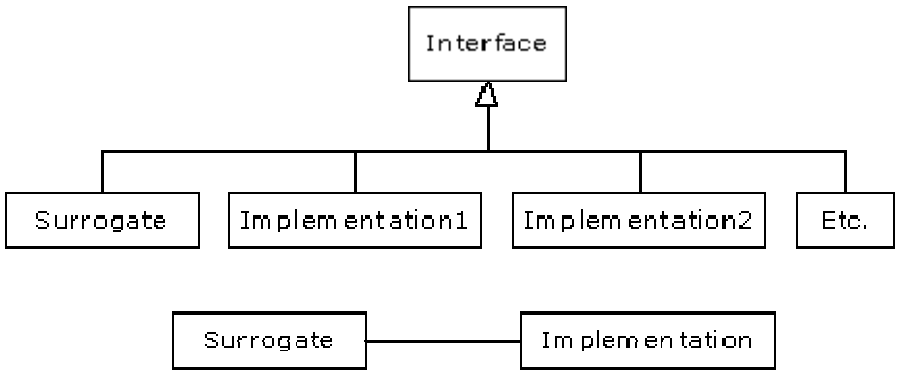
1. 给 ConnectionPoolDemo.java 添加单元测试，要示范以下问题：如果客户端已经释放了某个连接但是却仍然在使用它。

对象去耦（Object decoupling）

代理模式（Proxy）和状态模式（State）分别提供了供你使用的代理类(surrogate class)；正真干活的那个类被代理类隐藏了。当你调用代理类的一个方法的时候，代理类只是简单的调用实现类（implementing class）所对应的方法。这两种模式非常相似，实际上，代理模式只是状态模式的一个特例。

有人试图将这两种模式合在一起统称为 Surrogate 模式，但是“代理（proxy）”这个术语已经用了很长时间了，而且它有自己特殊的含义，它的这些含义基本上体现了这两种模式的差别所在。

这两种模式的基本概念非常简单：代理类（surrogate）和 实现类都由同一个基类派生出来：



当创建一个代理对象（surrogate object）时，同时会创建一个实现（对象），代理对象会把所有的方法调用传递给实现对象。

从结构上看，代理模式和状态模式之间的差别非常简单：一个代理（Proxy）只对应一个实现（implementation），而一个状态（State）却可以对应多个实现。《设计模式》一书认为，这两种两种模式的应用场合是截然不同的：代理模式用于控制对实现（类）的访问，而状态模式可以动态地改变实现（类）。但是，如果把“控制对实现类的访问”这个概念扩展开来的话，这两种模式就可以优雅的结合在一起了。

代理：替另外一个对象打点一切

我们按照上面的图示实现代理模式，下面是实现代码：

```
//: proxy:ProxyDemo.java
// Simple demonstration of the Proxy pattern.
package proxy;
import junit.framework.*;

interface ProxyBase {
```

```
 void f();
 void g();
 void h();
}

class Proxy implements ProxyBase {
 private ProxyBase implementation;
 public Proxy() {
  implementation = new Implementation();
 }
 // Pass method calls to the implementation:
 public void f() { implementation.f(); }
 public void g() { implementation.g(); }
 public void h() { implementation.h(); }
}

class Implementation implements ProxyBase {
 public void f() {
  System.out.println("Implementation.f()");
 }
 public void g() {
  System.out.println("Implementation.g()");
 }
 public void h() {
  System.out.println("Implementation.h()");
 }
}

public class ProxyDemo extends TestCase  {
 Proxy p = new Proxy();
 public void test() {
  // This just makes sure it will complete
  // without throwing an exception.
  p.f();
  p.g();
  p.h();
 }
 public static void main(String args[]) {
```

```
  junit.textui.TestRunner.run(ProxyDemo.class);
 }
} ///:~
```

当然，并不是说实现类和代理类必须实现完全相同的接口；既然代理类只是在一定程度上代表那个需要它提交(referring)方法的类，这就已经满足了 Proxy 模式的基本要求（注意这里的陈述和 GoF 一书所给出的定义是有差别的）。尽管如此，定义一个公共的接口还是很方便的，这样就可以强制实现类（Implementation）实现（fulfill）代理类（Proxy）需要调用的所有方法。

用 Proxy 模式实现 PoolManager

```
//: proxy:PoolManager.java
package proxy;
import java.util.*;
public class PoolManager {
 private static class PoolItem {
  boolean inUse = false;
  Object item;
  PoolItem(Object item) { this.item = item; }
 }
 public class ReleasableReference {  // Used to build the proxy
  private PoolItem reference;
  private boolean released = false;
  public ReleasableReference(PoolItem reference) {
   this.reference = reference;
  }
  public Object getReference() {
   if(released)
    throw new RuntimeException(
    "Tried to use reference after it was released");
   return reference.item;
  }
  public void release() {
   released = true;
   reference.inUse = false;
  }
 }
 private ArrayList items = new ArrayList();
```

```
 public void add(Object item) {
  items.add(new PoolItem(item));
 }

 // Different (better?) approach to running out of items:
 public static class EmptyPoolItem {}
 public ReleasableReference get() {
  for(int i = 0; i < items.size(); i++) {
   PoolItem pitem = (PoolItem)items.get(i);
   if(pitem.inUse == false) {
    pitem.inUse = true;
    return new ReleasableReference(pitem);
   }
  }
  // Fail as soon as you try to cast it:
  // return new EmptyPoolItem();
  return null; // temporary
 }
} ///:~

//: proxy:ConnectionPoolProxyDemo.java
package proxy;
import junit.framework.*;
 interface Connection {
 Object get();
 void set(Object x);
 void release();
}

class ConnectionImplementation implements Connection {
 public Object get() { return null; }
 public void set(Object s) {}
 public void release() {} // Never called directly
}

class ConnectionPool { // A singleton
 private static PoolManager pool = new PoolManager();
 private ConnectionPool() {} // Prevent synthesized constructor
```

```
 public static void addConnections(int number) {
  for(int i = 0; i < number; i++)
   pool.add(new ConnectionImplementation());
 }

 public static Connection getConnection() {
  PoolManager.ReleasableReference rr =
   (PoolManager.ReleasableReference)pool.get();
  if(rr == null) return null;
  return new ConnectionProxy(rr);
 }

 // The proxy as a nested class:
 private static
 class ConnectionProxy implements Connection {
  private PoolManager.ReleasableReference implementation;
  public ConnectionProxy(PoolManager.ReleasableReference rr) {
    implementation = rr;
   }
   public Object get() {
    return ((Connection)implementation.getReference()).get();
   }
   public void set(Object x) {
    ((Connection)implementation.getReference()).set(x);
   }
   public void release() { implementation.release(); }
 }
}

public class ConnectionPoolProxyDemo extends TestCase {
 static {
  ConnectionPool.addConnections(5);
 }
 public void test() {
  Connection c = ConnectionPool.getConnection();
  c.set(new Object());
  c.get();
  c.release();
```

```
 }
 public void testDisable() {
  Connection c = ConnectionPool.getConnection();
  String s = null;
  c.set(new Object());
  c.get();
  c.release();
  try {
   c.get();
  } catch(Exception e) {
   s = e.getMessage();
   System.out.println(s);
  }
  assertEquals(s, "Tried to use reference after it was released");
 }
 public static void main(String args[]) {
  junit.textui.TestRunner.run(
   ConnectionPoolProxyDemo.class);
 }
} ///:~
```

动态代理（Dynamic Proxies）

JDK1.3 引入了动态代理 (Dynamic Proxy). 尽管一开始有些复杂，但它确实是一个吸引人的工具。下面这个有趣的小例子证明了这一点， 当 invocation handler 被调用的时候，代理机制（proxying）开始工作。这是非常酷的一个例子，它就在我的脑海里，但是我必须得想出一些合理的东西给 invocation handler，这样才能举出一个有用的例子…(译注：作者还没有写完)

```
// proxy:DynamicProxyDemo.java
// Broken in JDK 1.4.1_01
package proxy;
import java.lang.reflect.*;
 interface Foo {
 void f(String s);
 void g(int i);
 String h(int i, String s);
}
```

```
public class DynamicProxyDemo {
 public static void main(String[] clargs) {
  Foo prox = (Foo)Proxy.newProxyInstance(
   Foo.class.getClassLoader(),
   new Class[]{ Foo.class },
   new InvocationHandler() {
    public Object invoke(
     Object proxy, Method method,
     Object[] args) {
      System.out.println(
       "InvocationHandler called:" +
       "\n\tMethod = " + method);
      if (args != null) {
       System.out.println("\targs = ");
       for (int i = 0; i < args.length; i++)
        System.out.println("\t\t" + args[i]);
      }
      return null;
     }
   });
  prox.f("hello");
  prox.g(47);
  prox.h(47, "hello");
 }
} ///:~
```

练习：用 java 的动态代理创建一个对象作为某个简单配置文件的前端。例如，在 good_stuff.txt 文件里有如下条目：

```
a=1
b=2
c="Hello World"
```

客户端程序员可以使用（你写的）NeatPropertyBundle 类：

```
NeatPropertyBundle p = new NeatPropertyBundle("good_stuff");
System.out.println(p.a);
System.out.println(p.b);
System.out.println(p.c);
```

配置文件可以包含任何内用，任意的变量名。动态代理要么返回对应属性的值要么告诉你它不存在（可能通过返回 null）。如果你摇设置一个原本不存在的属性值，动态代理会创建一个新的条目。ToString()方法应该显示当前的所有条目。

练习：和上一道练习类似，用 Java 的动态代理连接一个 DOS 的 Autoexec.bat 文件。

练习：接受一个可以返回数据的 SQL 查询语句，然后读取数据库的元数据（metadata）。为每一条记录（record）提供一个对象，这个对象拥有一下属性：列名(column names)和对应的数据类型(data types).

练习：用 XML-RPC 写一个简单的服务器和客户端.每一个客户端返回的对象都必须使用动态代理的概念（dynamic proxy concept）来实现（exercise）远端的方法。

读者 Andrea 写道：

『除了最后一个练习，我觉得你给出的上面几个练习都不咋的。我更愿意把 Invocation handler 看成是能和被代理对象正交的（orthogonal）东东。

换句话说，invocation handler 的实现应该是和动态创建的代理对象所提供的那些接口完全无关的。也就是说，一旦 invocation handler 写好之后，你就可以把它用于任何暴露接口的类，甚至是那些晚于 invocation handler 出现的类和接口。

这就是我为什么要说 invocation handler 所提供的服务是和被代理对象正交的(orthognal)。Rickard 在他的 SmartWorld 例子里给出了几个 handler，其中我最喜欢的是那个调用－重试(call-retry)handler。它首先调用那个（被代理的）实际对象，如果调用产生异常或者等待超时，就重试三次。如果这三次都失败了，那就返回一个异常。这个 Handler 可以被用于任何一个类。

那个 handler 的实现相对于你这里讲的来说过于复杂了，我用这个例子仅仅是想说明我所指的正交（orthogonal）服务到底是什么意思。

您所给出的那几个练习，在我看来，唯一适合用动态代理实现的就是最后那个用 XML-RPC 与对象通信的那个练习。因为你所使用的用以分发消息的机制（指 XML-RPC）是和你想要建立通信的那个对象完全正交的。』

状态模式：改变对象的行为

一个用来改变类的（状态的）对象。

迹象：几乎所有方法里都出现（相同的）条件（表达式）代码。

为了使同一个方法调用可以产生不同的行为，State 模式在代理（surrogate）的生命周期内切换它所对应的实现（implementation）。当你发现，在决定如何实现任

何一个方法之前都必须作很多测试的情况下，这是一种优化实现代码的方法。例如，童话故事青蛙王子就包含一个对象（一个生物），这个对象的行为取决于它自己所处的状态。你可以用一个布尔（boolean）值来表示它的状态，测试程序如下：

```
//: state:KissingPrincess.java
package state;
import junit.framework.*;
class Creature {
 private boolean isFrog = true;
 public void greet() {
  if(isFrog)
   System.out.println("Ribbet!");
  else
   System.out.println("Darling!");
 }
 public void kiss() { isFrog = false; }
}
public class KissingPrincess extends TestCase  {
 Creature creature = new Creature();
 public void test() {
  creature.greet();
  creature.kiss();
  creature.greet();
}
 public static void main(String args[]) {
  junit.textui.TestRunner.run(KissingPrincess.class);
 }
} ///:~
```

但是，greet() 方法（以及其它所有在完成操作之前必须测试 isFrog 值的那些方法）最终要产生一大堆难以处理的代码。如果把这些操作都委托给一个可以改变的状态对象（State object），那代码会简单很多。

```
//: state:KissingPrincess2.java
package state;
import junit.framework.*;
class Creature {
```

```
 private interface State {
  String response();
 }
 private class Frog implements State {
  public String response() { return "Ribbet!"; }
 }
 private class Prince implements State {
  public String response() { return "Darling!"; }
 }
 private State state = new Frog();
 public void greet() {
  System.out.println(state.response());
 }
 public void kiss() { state = new Prince(); }
}

public class KissingPrincess2 extends TestCase  {
 Creature creature = new Creature();
 public void test() {
  creature.greet();
  creature.kiss();
  creature.greet();
 }
 public static void main(String args[]) {
  junit.textui.TestRunner.run(KissingPrincess2.class);
 }
} ///:~
```

此外，状态（State）的改变会自动传递到所有用到它的地方，而不需要手工编辑类的方法以使改变生效。

下面的代码演示了 State 模式的基本结构。

```
//: state:StateDemo.java
// Simple demonstration of the State pattern.
package state;
import junit.framework.*;
```

```
interface State {
 void operation1();
 void operation2();
 void operation3();
}

class ServiceProvider {
 private State state;
 public ServiceProvider(State state) {
  this.state = state;
 }
 public void changeState(State newState) {
  state = newState;
 }
 // Pass method calls to the implementation:
 public void service1() {
  // ...
  state.operation1();
  // ...
  state.operation3();
 }
 public void service2() {
  // ...
  state.operation1();
  // ...
  state.operation2();
 }
 public void service3() {
  // ...
  state.operation3();
  // ...
  state.operation2();
 }
}

class Implementation1 implements State {
 public void operation1() {
  System.out.println("Implementation1.operation1()");
```

```
 }
 public void operation2() {
  System.out.println("Implementation1.operation2()");
 }
 public void operation3() {
  System.out.println("Implementation1.operation3()");
 }
}

class Implementation2 implements State {
 public void operation1() {
  System.out.println("Implementation2.operation1()");
 }
 public void operation2() {
  System.out.println("Implementation2.operation2()");
 }
 public void operation3() {
  System.out.println("Implementation2.operation3()");
 }
}

public class StateDemo extends TestCase  {
 static void run(ServiceProvider sp) {
  sp.service1();
  sp.service2();
  sp.service3();
 }
 ServiceProvider sp =
  new ServiceProvider(new Implementation1());
 public void test() {
  run(sp);
  sp.changeState(new Implementation2());
  run(sp);
 }
 public static void main(String args[]) {
  junit.textui.TestRunner.run(StateDemo.class);
 }
} ///:~
```

在 main( )函数里，先用到的是第一个实现，然后转入第二个实现。

当你自己实现 State 模式的时候就会碰到很多细节的问题，你必须根据自己的需要选择合适的实现方法，比如用到的状态（State）是否要暴露给调用的客户，以及如何使状态发生变化。有些情况下（比如 Swing 的 LayoutManager），，客户端可以直接传对象进来，但是在 KissingPrincess2.java 那个例子里，状态对于客户端来说是不可见的。此外，用于改变状态的机制可能很简单也可能很复杂－比如本书后面将要提到的状态机（State Machine），那里会讲到一系列的状态以及改变状态的不同机制。

上面提到 Swing 的 LayoutManager 那个例子非常有趣，它同时体现了 Strategy 模式和 State 模式的行为。

Proxy 模式和 State 模式的区别在于它们所解决的问题不同。《设计模式》里是这么描述 Proxy 模式的一般应用的：

1. 远程代理(Remote Proxy)为一个对象在不同的地址空间提供局部代理。RMI 编译器（rmic）在创建 stubs 和 skeletons 的时候会自动为你创建一个远端代理。

2. 虚代理（Virtual proxy），根据需要，在创建复杂对象时使用 “延迟初始化（lazy initialization）” .

3. 保护代理(protection proxy) 用于你不希望客户端程序员完全控制被代理对象（proxied object）的情况下。

4. 智能引用（smart reference）. 当访问被代理对象时提供额外的动作。例如，它可以用来对特定对象的引用进行计数，从而实现写时复制（copy-on-write），进而避免对象别名（object aliasing）. 更简单的一个例子是用来记录一个特定方法被调用的次数。

你可以把 java 里的引用（reference）看作是一种保护代理，它控制对分配在堆(heap)上的实际对象的访问（而且可以保证你不会用到一个空引用（null reference））。

『重写：在《设计模式》一书里，Proxy 模式和 State 模式被认为是互不相干的，因为那本书给出的用以实现这两种模式的结构是完全不同的（我认为这种实现有点武断）。尤其是 State 模式，它用了一个分离的实现层次结构，但我觉着完全没有必要，除非你认定实现代码不是由你来控制的（当然这也是一种可能的情况，但是如果代码是由你来控制的，那还是用一个单独的基类更简洁实用）。此外，Proxy 模式的实现不需要用一个公共的基类，因为代理对象只是控制对被代理对象的访问。尽管有细节上的差异，Proxy 模式和 State 模式都是用一个代理（surrogate）把方法调用传递给实现对象。』

State 模式到处可见，因为它是最基本的一个想法，比如，在 Builder 模式里，“Director”就是用一个后端（backend）的 Buider object 来产生不同的行为。

迭代器：分离算法和容器

Alexander Stepanov（和 Dave Musser 一起）写 STL 以前 ，已经用了好几年思考泛型编程（generic programming）的问题。最后他得出结论：所有的算法都是定义在代数结构（algebraic structures）之上的——我们把代数结构称作容器（container）。

在这个过程中，他意识到迭代器对于算法的应用是至关重要的，因为迭代器将算法从它所使用的特定类型的容器中分离出来。这就意味着在描述算法的时候，可以不必考虑它所操作的特定序列。更为一般情况，用迭代器写的任何代码都与它所操作的数据结构相分离，这样一来这些代码就更为通用并且易于重用。

迭代器的另外一个应用领域就是函数式编程（functional programming），它的目标是描述程序的每一步是干什么的，而不是描述程序的每一步是怎么做的。也就是说，使用“sort”（来排序），而不是具体描述排序的算法实现。C++STL 的目的就是为 C++语言提供对这种泛型编程方法的支持(这种方法成功与否还需要时间来验证)。

如果你用过 Java 的容器类（写代码不用到它们是很难的），那你肯定用过迭代器——Java1.0/1.1 把它叫作枚举器（Enumeration），Java2.0 叫作迭代器——你肯定已经熟悉它们的一般用法。如果你还不熟悉的话，可以参考《Thinking in Java》第二版第九章（可以从 http://www.bruceeckel.com/免费下载)。

因为 Java2 的容器非常依赖于迭代器，所以它们就成了泛型编程/函数式编程的最佳候选技术。这一章节通过把 STL 移植到 Java 来讲解这些技术，（移植的迭代器）会和 Java2 的容器类一起使用。

类型安全的迭代器

在《Thinking in Java》第二版里，我实现了一个类型安全的容器类，它只接受某一特定类型的对象。读者 Linda Pazzaglia 想要我实现另外一个类型安全的组件，一个可以和 java.util 里定义的容器类兼容的迭代器，但要限制它所遍历的对象必须都是同一类型的。

如果 Java 有模板（template）机制，上面这种（类型安全的）迭代器很容易就可以返回某一特定类型的对象。但是没有模板机制，就必须得返回 generic Objects，或者为每一种需要遍历的对象都手工添加代码。这里我会使用前一种方法。

另外一个需要在设计时决定的问题是什么时候判定对象的类型。一种方法是以迭代器遍历的第一个对象的类型（作为迭代器的类型），但是这种方法当容器类根据它

自己的内部算法（比如 hash 表）重新为对象排序时就会有问题，这样同一迭代器的两次遍历就可能得到不同的结果。安全的做法是在构造迭代器的时候让用户指定迭代器的类型。

最后的问题是如何构建迭代器。我们不可能重写现有的 Java 类库，它已经包含了枚举器和迭代器。但是，我们可以用 Decorator 模式简单的创建一个枚举器或者迭代器的外覆类，产生一个具有我们想要的迭代行为（本例中，指在类型不正确的时候抛出 RuntimeException 异常）的新对象，而这个新对象跟原来的枚举器或者迭代器有相同的接口，这样一来，它就可以用在相同的场合（或许你会争论说这实际上是 Proxy 模式，但是从它的目的（intent）来说它更像 Decorator 模式）。

实现代码如下:

```
//: com:bruceeckel:util:TypedIterator.java
package com.bruceeckel.util;
import java.util.*;

public class TypedIterator implements Iterator {
 private Iterator imp;
 private Class type;
 public TypedIterator(Iterator it, Class type) {
  imp = it;
  this.type = type;
 }
 public boolean hasNext() {
  return imp.hasNext();
 }
 public void remove() { imp.remove(); }
 public Object next() {
  Object obj = imp.next();
  if(!type.isInstance(obj))
   throw new ClassCastException(
   "TypedIterator for type " + type +
   " encountered type: " + obj.getClass());
  return obj;
 }
} ///:~
```

练习

1. 写一个“virtual proxy”。
2. 写一个“Smartreference”代理，用这个代理记录某个特定对象的方法调用次数。
3. 仿照某个 DBMS 系统，写一个程序限制最大连接数。用类似于 singleton 的方法控制连接对象的数量。当用户释放某个连接时，必须通知系统将释放的连接收回以便重用。为了保证这一点，写一个 proxy 对象代替对连接的引用计数，进一步设计这个 proxy 使它能够将连接释放回系统。
4. 用 State 模式，写一个 UnpredictablePerson 类，它根据自己的情绪（Mood）改变对 hello( )方法的响应。再写一个额外的 Mood 类：Prozac。
5. 写一个简单的 copy-on write 实现。
6. java.util.Map 没有提供直接从一个两维数组读入“key-value”对的方法。写一个 adapter 类实现这个功能。
7. 写一个适配器工厂（Adapter Factory），用它动态地找到并且产生适配器，这个适配器用以把一个给定的对象连接到一个预期的接口。
8. 用 java 标准库的动态代理重做练习 7。
9. 改写本节的 Object Pool，使得对象再一段时间以后自动回收到对象池。
10. 改写练习 9，用“租借（leasing）”的方法使得客户端可以刷新“租借对象”，从而阻止对象定时自动释放。
11. 考虑线程因素，重写 Object Pool。

分解共同性（Factoring Commonality）

应用“一次且只能有一次” 原则产生最基本的模式，将变化的那部分代码放到方法里。

这可以用两种方法来表达：

策略模式（Strategy）：运行时刻选择算法

另外，Strategy 模式还可以添加一个“上下文（context）”，这个 context 可以是一个代理类（surrogate class），用来控制对某个特定 strategy 对象的选择和使用。

```
//: strategy:StrategyPattern.java
package strategy;
import com.bruceeckel.util.*; // Arrays2.toString()
import junit.framework.*;

// The strategy interface:
interface FindMinima {
 // Line is a sequence of points:
 double[] algorithm(double[] line);
}
// The various strategies:
class LeastSquares implements FindMinima {
 public double[] algorithm(double[] line) {
  return new double[] { 1.1, 2.2 }; // Dummy
 }
}

class NewtonsMethod implements FindMinima {
 public double[] algorithm(double[] line) {
  return new double[] { 3.3, 4.4 }; // Dummy
 }
}

class Bisection implements FindMinima {
 public double[] algorithm(double[] line) {
  return new double[] { 5.5, 6.6 }; // Dummy
 }
```

```
}

class ConjugateGradient implements FindMinima {
 public double[] algorithm(double[] line) {
  return new double[] { 3.3, 4.4 }; // Dummy
 }
}
// The "Context" controls the strategy:
class MinimaSolver {
 private FindMinima strategy;
 public MinimaSolver(FindMinima strat) {
  strategy = strat;
 }
 double[] minima(double[] line) {
  return strategy.algorithm(line);
 }
 void changeAlgorithm(FindMinima newAlgorithm) {
  strategy = newAlgorithm;
 }
}

public class StrategyPattern extends TestCase  {
 MinimaSolver solver =
  new MinimaSolver(new LeastSquares());
 double[] line = {
  1.0, 2.0, 1.0, 2.0, -1.0,
   3.0, 4.0, 5.0, 4.0 };
  public void test() {
   System.out.println(
    Arrays2.toString(solver.minima(line)));
   solver.changeAlgorithm(new Bisection());
   System.out.println(
    Arrays2.toString(solver.minima(line)));
  }
  public static void main(String args[]) {
   junit.textui.TestRunner.run(StrategyPattern.class);
  }
} ///:~
```

请注意模板方法（template method）和 strategy 模式的相似性——template method 最显著的特征是它有多个方法需要调用，它是分段的实现某个功能。但是，这并不是说 strategy 对象就不能有多个方法调用；考虑 Shalloway 给出的的订单处理系统的例子，每一个 strategy 包含不同的国别信息。

JDK 里 Strategy 模式的例子：comparator objects.

Policy 模式：泛化的 strategy 模式

尽管 GoF 说 Policy 模式只是 Strategy 模式的一个别名，但是他们给出的 Strategy 模式的例子隐含的假定了 strategy 对象只使用一个方法－－也就是说假定你已经把可能变化的算法拆成了一段单独的代码。

其他人[6]用 Policy 模式来表示一个对象有多个方法的情况，对于不同的类，这些方法可能相互独立变化。相对于(Strategy 模式) 只能有一个方法而言，Policy 模式具有更大的灵活性。

例如，当某个产品需要被运送到不同国家的时候，与该产品相关的海运政策（shipping policy）可以被用来说明一些与运输有关的问题。这些问题可能包括运输方式，如何计算邮资或者运费，客户需求以及费用，特殊处理的费用等等。所有这些东西可能会互不相干的变化，而且很重要的一点是在运输过程中你可能会在不同地点（points）需要上述信息。

通常情况下，把 Strategy 模式和 Policy 模式区别开来是很有好处的，用 Strategy 模式处理一个方法（变化）的情况，而用 Policy 模式处理多个方法。

模板方法（Template method）

应用程序框架使你可以从一个或者一系列类继承下来，进而创建一个新的应用程序，你可以重用既有类的大多数代码并且按照你自己的需要覆写其中的某些方法，从而实现应用程序的定制。Template Method 是应用程序框架的一个基本概念，它通常隐藏在（框架）背后，通过调用基类的一组方法（有些方法你可能已经覆写（overridden）过了）来驱动应用程序。

比如，当创建一个 applet 的时候你就已经在使用应用程序框架了：你从 Japplet 继承下来，然后重载 init( )方法。Applet 机制（实际上就是 template method）完成剩下的工作，比如屏幕显示、处理事件循环、调整大小等等。

[6] Shalloway 所著的《Design Patterns Explained》以及 Alexandrescu 的《Advanced C++ Design》(??译注：应为《Modern C++ Design》)。

Template Method 的一个重要特征是：它是在基类里定义的，而且不能够被（派生类）更改。有时候它是私有方法（private method），但实际上它经常被声明为 final。它通过调用其它的基类方法（覆写过的）来工作，但它经常是作为初始化过程的一部分被调用的，这样就没必要让客户端程序员能够直接调用它了。

```
//: templatemethod:TemplateMethod.java
// Simple demonstration of Template Method.
package templatemethod;
import junit.framework.*;

abstract class ApplicationFramework {
 public ApplicationFramework() {
  templateMethod(); // Dangerous!
 }
 abstract void customize1();
 abstract void customize2();
 final void templateMethod() {
  for(int i = 0; i < 5; i++) {
   customize1();
   customize2();
  }
 }
}

// Create a new "application":
class MyApp extends ApplicationFramework {
 void customize1() {
  System.out.print("Hello ");
 }
 void customize2() {
  System.out.println("World!");
 }
}



public class TemplateMethod extends TestCase  {
 MyApp app = new MyApp();
 public void test() {
```

```
  // The MyApp constructor does all the work.
  // This just makes sure it will complete
  // without throwing an exception.
 }
 public static void main(String args[]) {
  junit.textui.TestRunner.run(TemplateMethod.class);
 }
} ///:~
```

基类的构造函数负责完成必要的初始化和启动应用程序“引擎”（template method）（在一个图形用户界面（GUI）程序里，这个“引擎”通常是“主事件循环（main event loop）”）。客户端程序员只需简单的提供 customize1( ) 和 customize2( )方法的定义，整个程序就可以跑起来了。

练习

1. 写一个框架，从命令行读入一系列文件名。打开除最后一个文件的所有文件，用于读入；打开最后一个文件用于写入。框架用待定的策略（policy）处理输入文件，并将输出结果写入到最后一个文件。继承这个框架，定制两个不同的应用程序。

   a. 把每个（输入）文件的文本转换为大写字母。

   b. 用第一个文件给出的单词搜索（输入）文件。

封装创建（Encapsulating creation）

当你发现需要向某个系统添加一些新类型的时候，最明智的做法就是先利用多态（polymorphism）为这些新类型创建一个公共接口。这可以使系统其余部分的代码与新加入的那些特定类型相分离。加入新的类型可能并不需要更改现有的代码。。。或者至少看上去不需要。初看起来，如果你用上面说的这种设计方法修改代码，那么唯一需要改动的就是继承新类型的地方，但事实上并非完全如此。你仍然需要创建新类型的对象，而且在创建的地方你必须指定用哪个构造函数。因此，如果用于创建对象的那部分代码分散在整个程序里（添加新类型的时候你会面临这个问题），你仍然需要找出所有与类型相关的代码。这种情况下，主要的问题是新类型（对象）的创建而不是新类型的使用（这个问题可以用多态来解决），但实际效果是一样的：添加一个新类型总会带来问题。

解决办法是强制使用一个通用工厂（common factory）来创建对象，而不允许创建对象的代码分散在整个系统里。如果所有需要创建对象的那些代码都借助于这个工厂（factory）来完成创建，那么添加新类型的时候你需要做的只是修改这个工厂就可以了。

因为所有面向对象的代码都会用到创建对象，而且大多数情况下你会通过添加新的类型来扩展程序，所以我猜想 factory 模式（系列）可能是最广泛使用的设计模式。

尽管只有简单工厂方法（Simple Factory Method）才是真正意义上的单件（Singleton），但你会发现每一个特定的工厂类（这些类属于更为通用的 factory 模式）实际上都只有一个实例。

简单工厂方法（Simple Factory method）

我们用下面的例子，重温一下 Shape 系统。

（实现 factory 模式）常用的方法是把 factory 声明为基类的静态方法（static method）。

```
//: factory:shapefact1:ShapeFactory1.java
// A simple static factory method.
package factory.shapefact1;
import java.util.*;
import junit.framework.*;

abstract class Shape {
 public abstract void draw();
 public abstract void erase();
```

```
 public static Shape factory(String type) {
  if(type.equals("Circle")) return new Circle();
  if(type.equals("Square")) return new Square();
  throw new RuntimeException(
   "Bad shape creation: " + type);
 }
}

class Circle extends Shape {
 Circle() {} // Package-access constructor
 public void draw() {
  System.out.println("Circle.draw");
 }
 public void erase() {
  System.out.println("Circle.erase");
 }
}

class Square extends Shape {
 Square() {} // Package-access constructor
 public void draw() {
  System.out.println("Square.draw");
 }
 public void erase() {
  System.out.println("Square.erase");
 }
}

public class ShapeFactory1 extends TestCase  {
 String shlist[] = { "Circle", "Square",
  "Square", "Circle", "Circle", "Square" };
 List shapes = new ArrayList();
 public void test() {
  Iterator it = Arrays.asList(shlist).iterator();
  while(it.hasNext())
   shapes.add(Shape.factory((String)it.next()));
  it = shapes.iterator();
  while(it.hasNext()) {
```

```
    Shape s = (Shape)it.next();
    s.draw();
    s.erase();
   }
 }
 public static void main(String args[]) {
  junit.textui.TestRunner.run(ShapeFactory1.class);
 }
} ///:~
```

factory() 方法需要传入一个参数来决定要创建的 Shape 的具体类型； 在上面的例子里（参数）碰巧是一个字符串（String），它也可以是其它任意类型。当加入新的 Shape 类型的时候（我们假定被创建对象的初始化代码是来自系统以外的，而不是像上面那个例子使用一个硬编码（hard-coded）的数组），系统唯一需要改动的代码就是 factory( )方法。为了促使创建对象的代码只包含在 factory( )方法里，特定类型的 Shape 类的构造函数都被声明为 package 权限，这样一来只有 factory( )方法可以调用这些构造函数，而位于包（package）以外的那部分代码则没有足够的权限（调用这些构造函数）。

多态工厂（Polymorphic factories）

上例中，静态的 factory( )方法使得所有创建对象的操作都集中在一个地方完成，这也就是唯一需要你修改代码的地方。这当然是一个还算不错的的解决办法，它封装了创建对象的过程。然而，《设计模式》强调使用 Factory Method 是为了使不同类型的工厂可以由基本类型的工厂派生（subclass）出来（上面例子是一个特例）。但是，那本书没有给出具体例子，只是重复了用于说明抽象工厂（Abstract Factory）的那个例子（你会在本书下一节看到一个 Abstract Factory 的例子）。下面的例子，我们修改了 ShapeFactory1.java 使得工厂方法成为一个单独的类的虚函数。请注意，特定类型的 Shape 类是根据需要动态加载的。

```
//: factory:shapefact2:ShapeFactory2.java
// Polymorphic factory methods.
package factory.shapefact2;
import java.util.*;
import junit.framework.*;

interface Shape {
 void draw();
 void erase();
```

```
}
abstract class ShapeFactory {
 protected abstract Shape create();
 private static Map factories = new HashMap();
 public static void
  addFactory(String id, ShapeFactory f) {
   factories.put(id, f);
  }
  // A Template Method:
  public static final
   Shape createShape(String id) {
    if(!factories.containsKey(id)) {
     try {
      // Load dynamically
      Class.forName("factory.shapefact2." + id);
     } catch(ClassNotFoundException e) {
      throw new RuntimeException(
       "Bad shape creation: " + id);
     }
     // See if it was put in:
     if(!factories.containsKey(id))
      throw new RuntimeException(
      "Bad shape creation: " + id);
    }
    return
     ((ShapeFactory)factories.get(id)).create();
   }
}

class Circle implements Shape {
 private Circle() {}
 public void draw() {
  System.out.println("Circle.draw");
 }
 public void erase() {
  System.out.println("Circle.erase");
 }
 private static class Factory
```

```
  extends ShapeFactory {
   protected Shape create() {
    return new Circle();
   }
  }
  static {
   ShapeFactory.addFactory(
    "Circle", new Factory());
  }
}

class Square implements Shape {
 private Square() {}
 public void draw() {
  System.out.println("Square.draw");
 }
 public void erase() {
  System.out.println("Square.erase");
 }
 private static class Factory
  extends ShapeFactory {
   protected Shape create() {
    return new Square();
   }
  }
  static {
   ShapeFactory.addFactory(
    "Square", new Factory());
  }
}

public class ShapeFactory2 extends TestCase  {
 String shlist[] = { "Circle", "Square",
  "Square", "Circle", "Circle", "Square" };
 List shapes = new ArrayList();
 public void test() {
  // This just makes sure it will complete
  // without throwing an exception.
```

```
  Iterator it = Arrays.asList(shlist).iterator();
  while(it.hasNext())
   shapes.add(
   ShapeFactory.createShape((String)it.next()));
  it = shapes.iterator();
  while(it.hasNext()) {
   Shape s = (Shape)it.next();
   s.draw();
   s.erase();
  }
 }
 public static void main(String args[]) {
  junit.textui.TestRunner.run(ShapeFactory2.class);
 }
} ///:~
```

现在工厂方法(factory method)出现在它自己的类 ShapeFactory 里，名字改成了 create( )方法。它是一个受保护的（protected）方法，也就是说它不能被直接调用，但可以被重载。Shape 类的子类必须创建与之对应的 ShapeFactory 的子类，并且通过重载 create( )函数来创建它自己的实例。实际上一系列 Shape 对象的创建是通过调用 ShapeFactory.createShape( ) 来完成的。 CreateShape( )是个静态方法， 它根据传入的标示，通过查找 ShapeFactory 的 Map 成员变量找到与之相应的工厂对象 （factory obejct）。 然后，找到的 factory 对象即被用来创建 shape 对象，但你可以想象一下更为棘手的问题：（与某种 Shape 类型相对应的）工厂对象被调用者用来以更为复杂的方式创建对象。但是，大多数情况下你似乎并不需要用到复杂的多态工厂方法（polymorphic factory method），在基类里加一个静态方法（像 ShapeFactory1.java 那样）就足以解决问题了。

注意到，ShapeFactory 的初始化必须通过加载 Map 数据成员才能完成（Map 的元素是 factory 对象），而这些初始化代码又位于 Shape 实现类的静态初始化语句里。这样一来，每加入一个新的类型你就必须得继承原来的类型（指 Shape？），创建一个 factory，然后添加静态初始化语句用以加载 Map 对象。这些额外的复杂性又一次暗示我们：如果不需要创建单独的 factory 对象，那最好还是使用静态工厂方法。

抽象工厂（Abstract factories）

抽象工厂（abstract factory）模式看起来很像前面我们看到的那些 factory 对象，只不过它有多个而不是一个 factory 方法。每一个 factory 方法创建一个不同类

型的对象。基本思想是：在创建工厂对象的地方，由你来决定如何使用该工厂对象创建的那些对象。《设计模式》里给出的例子实现了在不同用户图形界面（GUIs）之间的可移植性：你根据自己使用的 GUI 来创建一个与之对应的 factory 对象，在这以后，当你需要用到菜单，按钮，滚动条这些东西的时候，它会根据你使用的 GUI 自动创建合适的对象。这样，你就可以把实现不同 GUI 之间切换的代码分离出来，使它集中在一个地方。

作为另外一个例子，假设你要创建一个通用的游戏环境，而且你还想要支持不同类型的游戏。下面的例子用抽象工厂给出了它的一种可能的实现。

```
//: factory:Games.java
// An example of the Abstract Factory pattern.
package factory;
import junit.framework.*;

interface Obstacle {
  void action();
}

interface Player {
  void interactWith(Obstacle o);
}

class Kitty implements Player {
  public void interactWith(Obstacle ob) {
    System.out.print("Kitty has encountered a ");
    ob.action();
  }
}

class KungFuGuy implements Player {
  public void interactWith(Obstacle ob) {
    System.out.print("KungFuGuy now battles a ");
    ob.action();
  }
}

class Puzzle implements Obstacle {
  public void action() {
```

```
    System.out.println("Puzzle");
  }
}

class NastyWeapon implements Obstacle {
  public void action() {
    System.out.println("NastyWeapon");
  }
}

// The Abstract Factory:
interface GameElementFactory {
  Player makePlayer();
  Obstacle makeObstacle();
}

// Concrete factories:
class KittiesAndPuzzles
implements GameElementFactory {
  public Player makePlayer() {
    return new Kitty();
  }
  public Obstacle makeObstacle() {
    return new Puzzle();
  }
}

class KillAndDismember
implements GameElementFactory {
  public Player makePlayer() {
    return new KungFuGuy();
  }
  public Obstacle makeObstacle() {
    return new NastyWeapon();
  }
}
```

```
class GameEnvironment {
  private GameElementFactory gef;
  private Player p;
  private Obstacle ob;
  public GameEnvironment(
    GameElementFactory factory) {
    gef = factory;
    p = factory.makePlayer();
    ob = factory.makeObstacle();
  }
  public void play() { p.interactWith(ob); }
}

public class Games extends TestCase  {
  GameElementFactory
    kp = new KittiesAndPuzzles(),
    kd = new KillAndDismember();
  GameEnvironment
    g1 = new GameEnvironment(kp),
    g2 = new GameEnvironment(kd);
  // These just ensure no exceptions are thrown:
  public void test1() { g1.play(); }
  public void test2() { g2.play(); }
  public static void main(String args[]) {
    junit.textui.TestRunner.run(Games.class);
  }
} ///:~
```

在上面的游戏环境里，Player 对象与 Obstale 对象交互，根据你所选择的游戏类型，player 和 obstacle 的各自类型也会不同。你通过选择某个特定的 GameElementFactory 来决定游戏的类型，然后 GameElementFactory 会控制初始化和游戏的进行。在上面的例子里，初始化和游戏的进行都非常简单，但那些活动（初始条件和状态变化）可以在很大程度上决定游戏的结局。这里，GameEnvironment 不是用来给其它类继承的 ，尽管那么做很可能也说的通。

上面的代码也包含了双重分派（Double Dispatching）和工厂方法（Factory Method）两种模式，我们会在后面讲解它们。

练习

1. 写一个 Triangle（三角形）类，添加到 ShapeFactory1.java。
2. 写一个 Triangle（三角形）类，添加到 ShapeFactory2.java。
3. 给 GameEnvironment 添加新的类型，写一个 GnomesAndFairies 添加到 Games.java。
4. 改写 ShapeFactory2.java，用一个 Abstract Factory 来创建不同规格（sets）的 shapes(比如说，用一个特定类型的工厂对象来创建“thick shapes”，用另外一个工厂对象来创建“thin shapes”，但是每个工厂对象都可以创建所有类型的 shapes，包括：circles，squares，triangles 等等。)。

特化创建（Specialized creation）

原型模式（Prototype）

通过克隆某个原型的实例来创建对象。“模式重构(Pattern Refactoring)”一章会给出这种模式的一个例子。

生成器（Builder）

Builder 模式的目的是为了将对象的构造与它的“表示”（representation）分离开来，这样对象就可以有多个不同的“表示”。对象构造的过程保持不变，但是最终创建的对象可以有不同的表示。GoF 指出，Abstract Factory 模式与 Builder 模式的主要区别就在于 Builder 模式遵循一定的步骤一步步创建对象，这样一来，按照时间顺序创建对象就显得非常重要了。此外，“director”似乎是把一个序列的很多片断（pieces）传给 Builder，每个片断用来完成构建过程中的一个步骤。

GoF 给出的一个例子是文本格式转换器。待转换的文本是 RTF 格式的，当解析文本的时候，解析指令被传给文本转换器，文本转换器根据不同的输出格式（ASCII、TeX、或者“GUI 文本工具”）可能有不同的实现。尽管最终生成的“对象”（整个转换后的文件）的创建是有时间上的先后的，但是如果把每个 RTF 格式的转换指令都认为是一个对象，我觉得这更像是 Bridge 模式，因为某一具体类型的转换器扩展了基类的接口。另外，这个问题通常的解决方案使得前端（front end）可以有多个读取者，而转换器则在后端(back end)工作，这正是 Bridge 模式的主要特征。

在我看来，Builder 和常规 factory 最本质的区别在于，Builder 用多个步骤来创建对象，而且这些步骤对于 Builder 对象来说都是外部的。但是，GoF 强调的是（通过 Builder）你可以用同样的步骤创建不同的对象实体（representations）。他们从未说明“representation”到底指什么。（难道说“representation”是指过大的对象？ 那么，如果对象实体被分割成小的对象，是否就不需要使用 Builder 了呢？）

GoF 还给出了另外一个例子，创建迷宫对象，在迷宫内部添加房间，给房间加上门。这是一个需要多步完成的过程，但是，所谓不同的 “对象实体（representations）”，也就是是“常规的（standard）”和“复杂的(complex)”迷宫——实际上并不是说它们是不同类型的迷宫，而是指它们的复杂程度不一样。换了是我的话，我想我会试图创建一个迷宫构造器（maze builder）用它来创建任意复杂度的迷宫。Maze builder 的最终变体其实根本就不创建迷宫对象了，它只是计算现有迷宫所包含的房间数量。

RFT 转换器和迷宫构造器这两个例子都不是讲解 Builder 模式的非常有说服力的例子。有读者建议说 Sax 的 XML 解析器，或者是标准的编译器解析器可能更适合用来说明 Builder 模式。

下面这个例子可能比较适合用来说明 Builder 模式，至少它更能说明 Builder 模式的意图。我们可能会把媒体（media）创建为不同的表现形式，本例中可能是书籍，杂志或者网站。这个例子主要说明创建对象的步骤是相同的，这样它们才可以被抽象到 director 类里。

```
//: builder:BuildMedia.java
// Example of the Builder pattern
package builder;
import java.util.*;
import junit.framework.*;

// Different "representations" of media:
class Media extends ArrayList {}

class Book extends Media {}

class Magazine extends Media {}

class WebSite extends Media {}

// ... contain different kinds of media items:
class MediaItem {
  private String s;
  public MediaItem(String s) { this.s = s; }
  public String toString() { return s; }
}

class Chapter extends MediaItem {
  public Chapter(String s) { super(s); }
}

class Article extends MediaItem {
  public Article(String s) { super(s); }
}

class WebItem extends MediaItem {
  public WebItem(String s) { super(s); }
}
```

```
// ... but use the same basic construction steps:
class MediaBuilder {
  public void buildBase() {}
  public void addMediaItem(MediaItem item) {}
  public Media getFinishedMedia() { return null; }
}

class BookBuilder extends MediaBuilder {
  private Book b;
  public void buildBase() {
    System.out.println("Building book framework");
    b = new Book();
  }
  public void addMediaItem(MediaItem chapter) {
    System.out.println("Adding chapter " + chapter);
    b.add(chapter);
  }
  public Media getFinishedMedia() { return b; }
}

class MagazineBuilder extends MediaBuilder {
  private Magazine m;
  public void buildBase() {
    System.out.println("Building magazine framework");
    m = new Magazine();
  }
  public void addMediaItem(MediaItem article) {
    System.out.println("Adding article " + article);
    m.add(article);
  }
  public Media getFinishedMedia() { return m; }
}

class WebSiteBuilder extends MediaBuilder {
  private WebSite w;
  public void buildBase() {
    System.out.println("Building web site framework");
    w = new WebSite();
```

```
  }
  public void addMediaItem(MediaItem webItem) {
    System.out.println("Adding web item " + webItem);
    w.add(webItem);
  }
  public Media getFinishedMedia() { return w; }
}

class MediaDirector { // a.k.a. "Context"
  private MediaBuilder mb;
  public MediaDirector(MediaBuilder mb) {
    this.mb = mb; // Strategy-ish
  }
  public Media produceMedia(List input) {
    mb.buildBase();
    for(Iterator it = input.iterator(); it.hasNext();)
      mb.addMediaItem((MediaItem)it.next());
    return mb.getFinishedMedia();
  }
};

public class BuildMedia extends TestCase {
  private List input = Arrays.asList(new MediaItem[] {
    new MediaItem("item1"), new MediaItem("item2"),
    new MediaItem("item3"), new MediaItem("item4"),
  });
  public void testBook() {
    MediaDirector buildBook =
      new MediaDirector(new BookBuilder());
    Media book = buildBook.produceMedia(input);
    String result = "book: " + book;
    System.out.println(result);
    assertEquals(result,
      "book: [item1, item2, item3, item4]");
  }
  public void testMagazine() {
    MediaDirector buildMagazine =
      new MediaDirector(new MagazineBuilder());
```

```
    Media magazine = buildMagazine.produceMedia(input);
    String result = "magazine: " + magazine;
    System.out.println(result);
    assertEquals(result,
      "magazine: [item1, item2, item3, item4]");
  }
  public void testWebSite() {
    MediaDirector buildWebSite =
      new MediaDirector(new WebSiteBuilder());
    Media webSite = buildWebSite.produceMedia(input);
    String result = "web site: " + webSite;
    System.out.println(result);
    assertEquals(result,
      "web site: [item1, item2, item3, item4]");
  }
  public static void main(String[] args) {
    junit.textui.TestRunner.run(BuildMedia.class);
  }
} ///:~
```

注意到，从某种角度来说上面这个例子也可以看成是比较复杂的 State 模式，因为 director 的行为取决于你所使用的 builder 的具体类型。Director 不是简单的将请求传递给下层的 State 对象，而是把 State 对象当作一个 Policy 对象来使用，最终自己来完成一系列的操作。这么一来，Builder 模式就可以被描述成使用某种策略来创建对象。

练习

1. 将一个文本文件作为一系列单词读入（an input stream of words），考虑使用正则表达式。写一个 Builder 将这些单词读入到一个 java.util.TreeSet，然后再写一个 Builder 创建一个 java.util.HashMap 用来保存单词和它出现的次数（也就是统计单词出现的次数）。

太多（Too many）

享元模式（Flyweight）:太多对象

Flyweidht 模式可能让人感到奇怪的一点是： 它（在其它模式的辅助下）是改善性能（performance hack）的一种方法。 通常来说比较好的做法就是简单的把系统里所有东西都弄成对象，但是有时候这样做会使得产生的对象数量非常巨大，这可能会导致系统过于缓慢或者内存耗尽。

Flyweight 模式通过减少对象数量来解决这个问题。为了达到这个目的，需要外部化（externalize）原本属于对象一些数据，这样使得看起来对象的数量要比实际的多。但是，这同时也增加了应用这些对象时接口的复杂度，因为你必须传入额外的信息用以告诉调用方法如何找到那些外部化的信息。

作为一个非常简单的例子，考虑一个 DataPoint 对象，它包含一个 int 和 float 类型的数据成员，还有一个 id 数据成员用来表示对象编号。假设你需要创建一百万个这样的对象，然后对它们进行操作，像下面这样：

```
//: flyweight:ManyObjects.java
class DataPoint {
  private static int count = 0;
  private int id = count++;
  private int i;
  private float f;
  public int getI() { return i; }
  public void setI(int i) { this.i = i; }
  public float getF() { return f; }
  public void setF(float f) { this.f = f; }
  public String toString() {
    return "id: " + id + ", i = " + i + ", f = " + f;
  }
}

public class ManyObjects {
  static final int size = 1000000;
  public static void main(String[] args) {
    DataPoint[] array = new DataPoint[size];
    for(int i = 0; i < array.length; i++)
      array[i] = new DataPoint();
    for(int i = 0; i < array.length; i++) {
```

```
      DataPoint dp = array[i];
      dp.setI(dp.getI() + 1);
      dp.setF(47.0f);
    }
    System.out.println(array[size -1]);
  }
} ///:~
```

上面这个程序运行下来可能需要几秒钟，这取决于你的机器。更复杂的对象和更多的操作可能会使额外开销难以维持。为了解决这个问题，我们通过外部化原本属于DataPoint 数据成员的方法，使对象个数从一百万个减少到一个。

```
//: flyweight:FlyWeightObjects.java
class ExternalizedData {
  static final int size = 5000000;
  static int[] id = new int[size];
  static int[] i = new int[size];
  static float[] f = new float[size];
  static {
    for(int i = 0; i < size; i++)
      id[i] = i;
  }
}

class FlyPoint {
  private FlyPoint() {}
  public static int getI(int obnum) {
    return ExternalizedData.i[obnum];
  }
  public static void setI(int obnum, int i) {
    ExternalizedData.i[obnum] = i;
  }
  public static float getF(int obnum) {
    return ExternalizedData.f[obnum];
  }
  public static void setF(int obnum, float f) {
    ExternalizedData.f[obnum] = f;
  }
```

```
  public static String str(int obnum) {
    return "id: " +
      ExternalizedData.id[obnum] +
      ", i = " +
      ExternalizedData.i[obnum] +
      ", f = " +
      ExternalizedData.f[obnum];
  }
}

public class FlyWeightObjects {
  public static void main(String[] args) {
    for(int i = 0; i < ExternalizedData.size; i++) {
      FlyPoint.setI(i, FlyPoint.getI(i) + 1);
      FlyPoint.setF(i, 47.0f);
    }
    System.out.println(
      FlyPoint.str(ExternalizedData.size -1));
  }
} ///:~
```

这么一来，因为所有数据都存放在 ExternalizedData 里，所以每个对 FlyPoint 方法的调用都必须包含对 ExternalizedData 的索引。出于一致性的考虑，而且也是为了提醒读者注意（FlyPoint）各个方法的隐含（implicit）this 指针的相似性，“this 索引”是作为第一个参数传入的。

自然，这里又要对过早优化（premature optimization）提出警告。“先让程序能够运行起来，如果确有这个必要的话，再想办法提高速度。”再说一次，用 profiler 工具来检测性能瓶颈，而不要靠猜测。

装饰模式（Decorator）：太多类

Decorator 模式是指：用分层的对象（layered objects）动态地和透明地给单个对象添加功能（responsibilities）。

当使用派生类方法产生过多（不灵活的）类的时候考虑应用 Decorator 模式。

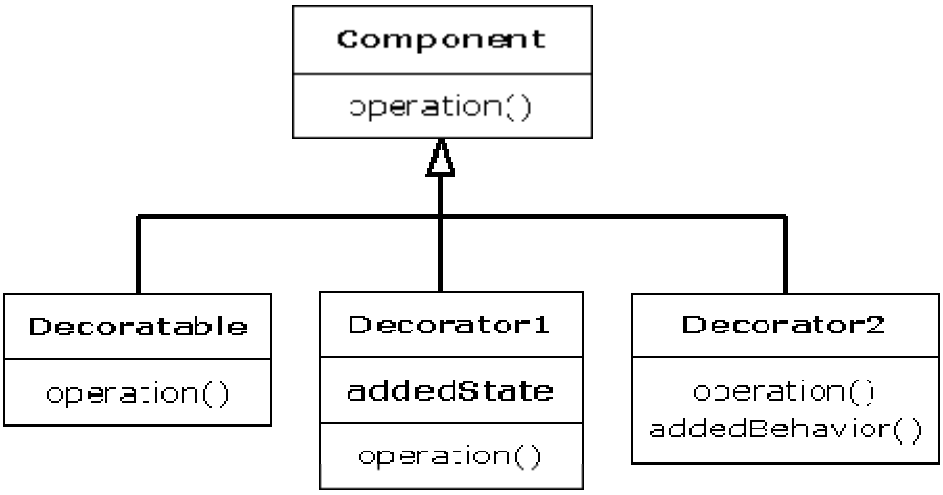
所有外覆在原始对象上的装饰类都必须有同样的基本接口。

动态 proxy/surrogate?

Decorator 模式对于非常规的继承结构特别管用。

权衡：使用 decoator 模式通常会使代码变得比较复杂。

基本 decorator 结构



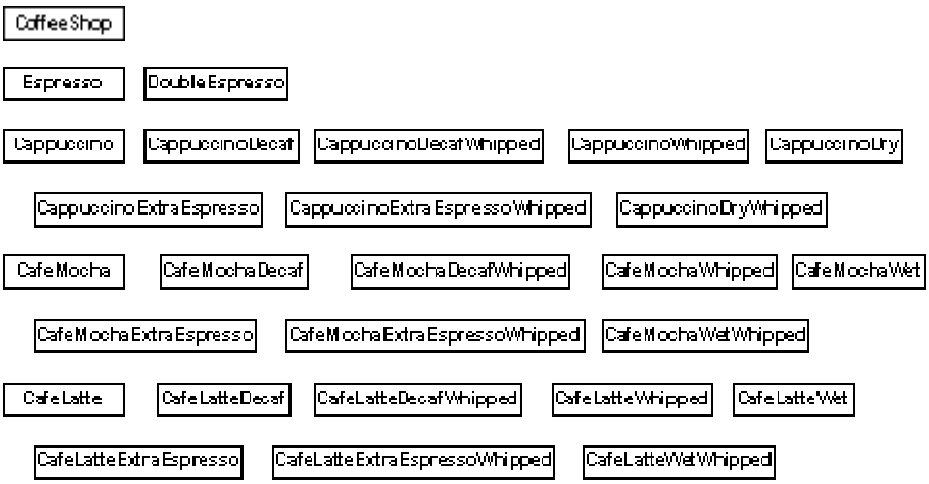
一个关于咖啡的例子

考虑去当地的一家咖啡馆 BeanMeUp 喝杯咖啡。那里通常会提供许多种不同的饮料——蒸馏咖啡, lattes, 茶, 冰咖啡，热巧克力饮品，而且还有许多额外的添加物，比如生奶油或者额外的蒸馏咖啡粉末（当然这些是要另外收钱的）。你也可以不支付额外的费用而改变你的饮料，比如你可以要求把普通咖啡换成脱咖啡因咖啡。

有一点是非常清楚的，如果我们想给上面所有这些饮料和它们的组合建模，那将会产生一张非常大的类图。为了能够清楚的说明问题，我们只考虑所有咖啡品种的一个子集：Espresso, Espresso Con Panna, Café Late, Cappuccino and Café Mocha. 我们提供 2 种额外的添加物——生奶油（“whipped”）和蒸馏咖啡粉，和三种可选择的饮品——脱咖啡因咖啡，steamed milk ("wet") 和 foamed milk ("dry").

每种组合对应一个类

一个解决办法是给每种组合都创建一个类。用它来描述每种饮料（的组成）和价格等等。最终产生的饮料单是巨大的，整个类图的一部分可能看起来是这样的：



下面是 Cappuccino 这种组合的一种简单实现：

```
class Cappuccino {
  private float cost = 1;
  private String description = "Cappucino";
  public float getCost() {
    return cost;
  }
  public String getDescription() {
    return description;
  }
}
```

使用这种方法最关键的一点是找到你需要的那种特定组合。找到你想要的饮料之后，下面 CoffeeShop 类的代码演示了如何使用它。

```
//: decorator:nodecorators:CoffeeShop.java
// Coffee example with no decorators
package decorator.nodecorators;
import junit.framework.*;

class Espresso {}

class DoubleEspresso {}

class EspressoConPanna {}

class Cappuccino {
```

```
  private float cost = 1;
  private String description = "Cappucino";
  public float getCost() {
    return cost;
  }
  public String getDescription() {
    return description;
  }
}

class CappuccinoDecaf {}

class CappuccinoDecafWhipped {}

class CappuccinoDry {}

class CappuccinoDryWhipped {}

class CappuccinoExtraEspresso {}

class CappuccinoExtraEspressoWhipped {}

class CappuccinoWhipped {}

class CafeMocha {}

class CafeMochaDecaf {}

class CafeMochaDecafWhipped {
  private float cost = 1.25f;
  private String description =
    "Cafe Mocha decaf whipped cream";
  public float getCost() {
    return cost;
  }

  public String getDescription() {
    return description;
```

```
  }
}

class CafeMochaExtraEspresso {}

class CafeMochaExtraEspressoWhipped {}

class CafeMochaWet {}

class CafeMochaWetWhipped {}

class CafeMochaWhipped {}

class CafeLatte {}

class CafeLatteDecaf {}

class CafeLatteDecafWhipped {}

class CafeLatteExtraEspresso {}

class CafeLatteExtraEspressoWhipped {}

class CafeLatteWet {}

class CafeLatteWetWhipped {}

class CafeLatteWhipped {}

public class CoffeeShop extends TestCase  {
  public void testCappuccino() {
    // This just makes sure it will complete
    // without throwing an exception.
    // Create a plain cappuccino
    Cappuccino cappuccino = new Cappuccino();
    System.out.println(cappuccino.getDescription()
      + ": $" + cappuccino.getCost());
  }
```

```
  public void testCafeMocha() {
    // This just makes sure it will complete
    // without throwing an exception.
    // Create a decaf cafe mocha with whipped
    // cream
    CafeMochaDecafWhipped cafeMocha =
      new CafeMochaDecafWhipped();
    System.out.println(cafeMocha.getDescription()
      + ": $" + cafeMocha.getCost());
  }
  public static void main(String[] args) {
    junit.textui.TestRunner.run(CoffeeShop.class);
  }
} ///:~
```

对应的输出结果如下：

Cappucino: $1.0

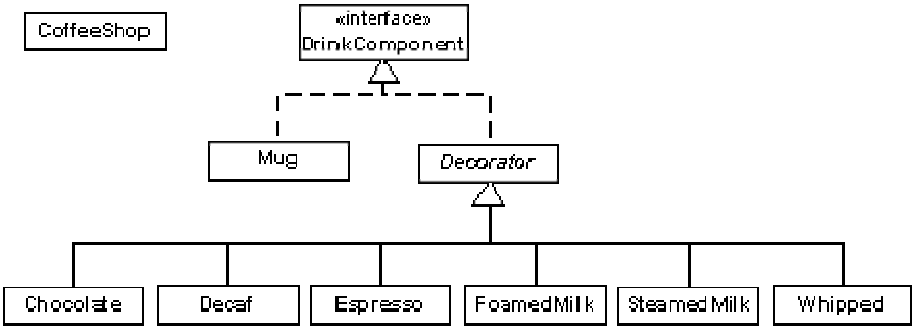
Cafe Mocha decaf whipped cream: $1.25

你可以看到，创建你想要的组合是非常简单的，因为你只要创建某个类的一个实例就可以了。但是，这种方法存在许多问题。首先，这些组合都是静态固定死的，客户想要的每个组合都必须预先创建好。其次，整个组合生成的菜单实在是太大了，要找到某个特定的组合非常困难而且耗时。

Decorator 方法

另外可以采用把这些饮料拆成类似于 esspresso 和 foamed milk 的组件（components），然后让顾客通过组合这些组件来描述某种特定的咖啡饮料。

我们使用 Decorator 模式，通过编程来实现上述想法。Decorator 通过包裹（wrapping）一个组件来给它增加功能，但是 Decorator 兼容它所包裹的那个类的接口，这样，对 component 的包裹就是透明的。Decorator 自身又可以被（别的 Decorator）包裹而不丧失其透明性。



调用 Decorator 的某个方法，会依次调用到 Decorator 所包裹的 component 的方法，而 component 的方法可以在 Decorator 方法被调用之前或者之后被调用。

如果为 DrinkComponent 接口添加 getTotalCost( )和 getDescription( )方法，Esspresso 看起来就会像下面的样子：

```
class Espresso extends Decorator {
  private float cost = 0.75f;
  private String description = " espresso";
  public Espresso(DrinkComponent component) {
    super(component);
  }
  public float getTotalCost() {
    return component.getTotalCost() + cost;
  }
  public String getDescription() {
    return component.getDescription() +
      description;
  }
}
```

你可以通过组合来创建一种饮料，就像下面的代码那样：

```
//: decorator:alldecorators:CoffeeShop2.java
// Coffee example using decorators
package decorator.alldecorators;
import junit.framework.*;

interface DrinkComponent {
  String getDescription();
  float getTotalCost();
```

```
}

class Mug implements DrinkComponent {
  public String getDescription() {
    return "mug";
  }
  public float getTotalCost() {
    return 0;
  }
}

abstract class Decorator implements DrinkComponent
{
  protected DrinkComponent component;
  Decorator(DrinkComponent component) {
    this.component = component;
  }
  public float getTotalCost() {
    return component.getTotalCost();
  }
  public abstract String getDescription();
}

class Espresso extends Decorator {
  private float cost = 0.75f;
  private String description = " espresso";
  public Espresso(DrinkComponent component) {
    super(component);
  }
  public float getTotalCost() {
    return component.getTotalCost() + cost;
  }
  public String getDescription() {
    return component.getDescription() +
      description;
  }
}
```

```
class Decaf extends Decorator {
  private String description = " decaf";
  public Decaf(DrinkComponent component) {
    super(component);
  }
  public String getDescription() {
    return component.getDescription() +
      description;
  }
}

class FoamedMilk extends Decorator {
  private float cost = 0.25f;
  private String description = " foamed milk";
  public FoamedMilk(DrinkComponent component) {
    super(component);
  }
  public float getTotalCost() {
    return component.getTotalCost() + cost;
  }
  public String getDescription() {
    return component.getDescription() +
      description;
  }
}

class SteamedMilk extends Decorator {
  private float cost = 0.25f;
  private String description = " steamed milk";
  public SteamedMilk(DrinkComponent component) {
    super(component);
  }
  public float getTotalCost() {
    return component.getTotalCost() + cost;
  }
  public String getDescription() {
    return component.getDescription() +
      description;
```

```
  }
}

class Whipped extends Decorator {
  private float cost = 0.25f;
  private String description = " whipped cream";
  public Whipped(DrinkComponent component) {
    super(component);
  }
  public float getTotalCost() {
    return component.getTotalCost() + cost;
  }
  public String getDescription() {
    return component.getDescription() +
      description;
  }
}

class Chocolate extends Decorator {
  private float cost = 0.25f;
  private String description = " chocolate";
  public Chocolate(DrinkComponent component) {
    super(component);
  }
  public float getTotalCost() {
    return component.getTotalCost() + cost;
  }
  public String getDescription() {
    return component.getDescription() +
      description;
  }
}

public class CoffeeShop2 extends TestCase  {
  public void testCappuccino() {
    // This just makes sure it will complete
    // without throwing an exception.
    // Create a plain cappucino
```

```
    DrinkComponent cappuccino = new Espresso(
      new FoamedMilk(new Mug()));
    System.out.println(cappuccino.
      getDescription().trim() + ": $" +
      cappuccino.getTotalCost());
  }
  public void testCafeMocha() {
    // This just makes sure it will complete
    // without throwing an exception.
    // Create a decaf cafe mocha with whipped
    // cream
    DrinkComponent cafeMocha = new Espresso(
      new SteamedMilk(new Chocolate(new Whipped(
      new Decaf(new Mug())))));
    System.out.println(cafeMocha.getDescription().
      trim() + ": $" + cafeMocha.getTotalCost());
  }
  public static void main(String[] args) {
    junit.textui.TestRunner.run(CoffeeShop2.class);
  }
} ///:~
```

这种方法当然提供了最灵活和体积最小的菜单。你只需从很少几个组件（components）里挑出你需要的，但是拼接这些“关于咖啡的描述”就变的十分费劲。

如果你想描述纯的 cappussino 咖啡，你可以这么写：

```
        new Espresso(new FoamedMilk(new Mug()));
```

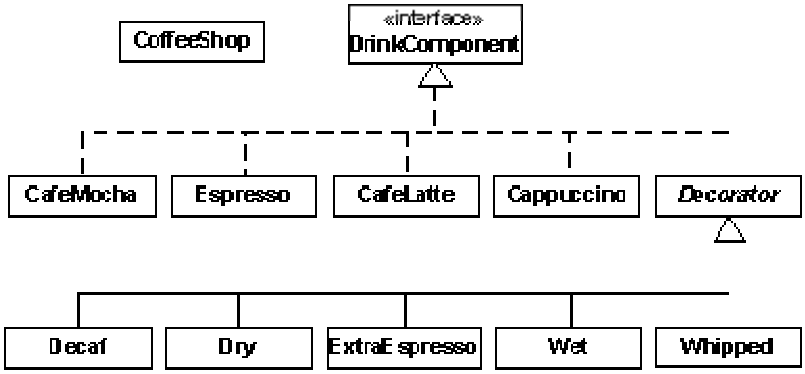
但是创建“a decaf Café Mocha with whipped cream”就需要相当冗长的描述。

折衷（Compromise）

用上面这种方法来描述某种咖啡饮品实在是太冗长了。对于那些经常会被用到的组合，如果用一种更快的方法来描述它们会方便很多。

这里的第三种方法就是前面两种方法的组合，也就是灵活性和易用性相结合。这种折衷是通过创建合适大小的供常规选择的菜单来实现的，这个菜单基本上是不怎么

变化的，但是如果你想给这些基本饮品加点伴侣（比如 whipped cream, decaf 等），那你可以用 decorators 来改变基本饮品。这其实也是大多数咖啡馆提供的那种菜单。



下面就是常规饮品（basic selection）和加了伴侣之后的饮品（decorated selection）的一个例子：

```
//: decorator:compromise:CoffeeShop3.java
// Coffee example with a compromise of basic
// combinations and decorators

package decorator.compromise;
import junit.framework.*;

interface DrinkComponent {
  float getTotalCost();
  String getDescription();
}

class Espresso implements DrinkComponent {
  private String description = "Espresso";
  private float cost = 0.75f;
  public float getTotalCost() {
    return cost;
  }
  public String getDescription() {
    return description;
  }
}

class EspressoConPanna implements DrinkComponent {
```

```
  private String description = "EspressoConPare";
  private float cost = 1;
  public float getTotalCost() {
    return cost;
  }
  public String getDescription() {
    return description;
  }
}

class Cappuccino implements DrinkComponent {
  private float cost = 1;
  private String description = "Cappuccino";
  public float getTotalCost() {
    return cost;
  }
  public String getDescription() {
    return description;
  }
}

class CafeLatte implements DrinkComponent {
  private float cost = 1;
  private String description = "Cafe Late";
  public float getTotalCost() {
    return cost;
  }
  public String getDescription() {
    return description;
  }
}

class CafeMocha implements DrinkComponent {
  private float cost = 1.25f;
  private String description = "Cafe Mocha";
  public float getTotalCost() {
    return cost;
  }
```

```
  public String getDescription() {
    return description;
  }
}

abstract class Decorator implements DrinkComponent {
  protected DrinkComponent component;
  public Decorator(DrinkComponent component) {
    this.component = component;
  }
  public float getTotalCost() {
    return component.getTotalCost();
  }
  public String getDescription() {
    return component.getDescription();
  }
}

class ExtraEspresso extends Decorator {
  private float cost = 0.75f;
  public ExtraEspresso(DrinkComponent component) {
    super(component);
  }
  public String getDescription() {
    return component.getDescription() +
      " extra espresso";
  }
  public float getTotalCost() {
    return cost + component.getTotalCost();
  }
}

class Whipped extends Decorator {
  private float cost = 0.50f;
  public Whipped(DrinkComponent component) {
    super(component);
  }
  public float getTotalCost() {
```

```
    return cost + component.getTotalCost();
  }
  public String getDescription() {
    return component.getDescription() +
      " whipped cream";
  }
}

class Decaf extends Decorator{
  public Decaf(DrinkComponent component) {
    super(component);
  }
  public String getDescription() {
    return component.getDescription() + " decaf";
  }
}

class Dry extends Decorator {
  public Dry(DrinkComponent component) {
    super(component);
  }
  public String getDescription() {
    return component.getDescription() +
      " extra foamed milk";
  }
}

class Wet extends Decorator {
  public Wet(DrinkComponent component) {
    super(component);
  }
  public String getDescription() {
    return component.getDescription() +
      " extra steamed milk";
  }
}

public class CoffeeShop3 extends TestCase  {
```

```
  public void testCappuccino() {
    // This just makes sure it will complete
    // without throwing an exception.
    // Create a plain cappucino
    DrinkComponent cappuccino = new Cappuccino();
    System.out.println(cappuccino.getDescription()
      + ": $" + cappuccino.getTotalCost());
  }
  public void testCafeMocha() {
    // This just makes sure it will complete
    // without throwing an exception.
    // Create a decaf cafe mocha with whipped
    // cream
    DrinkComponent cafeMocha = new Whipped(
      new Decaf(new CafeMocha()));
    System.out.println(cafeMocha.getDescription()
      + ": $" + cafeMocha.getTotalCost());
  }
  public static void main(String[] args) {
    junit.textui.TestRunner.run(CoffeeShop3.class);
  }
} ///:~
```

你可以看到，创建一个基本饮品（basic selection）是非常快且简单的，因为描述基本饮品成为一项常规工作。描述添加伴侣的饮品（decorated drink）比每种组合对应一个类那种方法要麻烦一些，但是，很明显比只使用 decorator 模式要简单很多。最后的结果是产生不算太多的类，同样，也不算太多的咖啡伴侣（decorators）。大多数情况下，我们碰到的都是不加伴侣的情况，这样我们就可以得益于上述两种方法带来的好处。

其它考虑

假如说我们打算后来改变菜单，比如添加一种新类型的饮品，那会发生什么情况呢？如果我们使用的是每种组合对应一个类的那种方法，添加一种额外饮品（比如糖浆 syrup）将会导致类的个数呈指数增长。但是，对于只使用 decorator 的方法和上述那种折衷的方法，都只需要创建一个额外的类。

当牛奶涨价的时候，如果我们改变 steamed milk 和 foamed milk 的价格，会带来什么影响呢？如果是每种组合对应一个类，你必须得更改所有类的（getCost）方法，这样就得维护非常多的类。通过使用 Decorators，只要在一个地方维护（价格改变相关）的逻辑代码就可以了。

练习

1. 使用上面的 decorator 方法，添加一个 Syrup 类。然后创建一个添加了 Café Latte 的 Syrup（你得用到 steamed milk 和 espresso）。

2. 用“折衷”的那种方法重做练习 1。

3. 写一个简单的系统，用 decorator 模式模拟以下情况：有些鸟会飞而有一些不会，有一些鸟会有用而有一些不会，还有一些既会飞又会游泳。

4. 用 decorator 模式写一个 Pizza 餐馆，它提供可供选择的基本菜单，也允许你自己定制 pizza。用“折衷”的方法写一个菜单包括 Margherita, Havaiian, Regina 和 Vegetarian pizza(反正都是 pizza 的品种)，和浇头（toppings，即 decorator）大蒜 Garlic，橄榄 Olives，菠菜 Spinach，鳄梨？Avocade. 奶酪 Feta,胡椒？Pepperdews。 写一个 Hawaiian pizza，再写一个添加了菠菜，奶酪，和胡椒的 Margherita pizza.

5. 关于 Decorator 模式，《设计模式》一书是这么说的:“通过附加（attching）或者分离（detaching）装饰者（decorators），可以在运行时刻（run-time）添加或者去处某项功能（responsibilities）”。写一个咖啡的 Decoration 系统，这个系统可以从某种复杂的咖啡饮料的一系列添加物（decorators）里“简单”的去除某项功能。

连接不同类型

适配器（Adapter）

适配器（Adaper）接受一种类型，并为其它类型产生一个接口。当你手头有某个类，而你需要的却是另外一个类，你可以通过 Adapter 来解决问题。唯一需要做的就是产生出你需要的那个类，有许多种方法可以完成这种配接。

```
//: adapter:SimpleAdapter.java
// "Object Adapter" from GoF diagram

package adapter;
import junit.framework.*;

class Target {
  public void request() {}
}

class Adaptee {
  public void specificRequest() {
    System.out.println("Adaptee: SpecificRequest");
  }
}

class Adapter extends Target {
  private Adaptee adaptee;
  public Adapter(Adaptee a) {
    adaptee = a;
  }
  public void request() {
    adaptee.specificRequest();
  }
}

public class SimpleAdapter extends TestCase  {
  Adaptee a = new Adaptee();
  Target t = new Adapter(a);
  public void test() {
    t.request();
```

```
  }
  public static void main(String args[]) {
    junit.textui.TestRunner.run(SimpleAdapter.class);
  }
} ///:~

//: adapter:AdapterVariations.java
// Variations on the Adapter pattern.

package adapter;
import junit.framework.*;

class WhatIHave {
  public void g() {}
  public void h() {}
}

interface WhatIWant {
  void f();
}

class SurrogateAdapter implements WhatIWant {
  WhatIHave whatIHave;
  public SurrogateAdapter(WhatIHave wih) {
    whatIHave = wih;
  }
  public void f() {
    // Implement behavior using
    // methods in WhatIHave:
    whatIHave.g();
    whatIHave.h();
  }
}

class WhatIUse {
  public void op(WhatIWant wiw) {
    wiw.f();
  }
```

```
}

// Approach 2: build adapter use into op():

class WhatIUse2 extends WhatIUse {
  public void op(WhatIHave wih) {
    new SurrogateAdapter(wih).f();
  }
}

// Approach 3: build adapter into WhatIHave:

class WhatIHave2 extends WhatIHave
implements WhatIWant {
  public void f() {
    g();
    h();
  }
}

// Approach 4: use an inner class:

class WhatIHave3 extends WhatIHave {
  private class InnerAdapter implements WhatIWant{
    public void f() {
      g();
      h();
    }
  }
  public WhatIWant whatIWant() {
    return new InnerAdapter();
  }
}

public class AdapterVariations extends TestCase  {
  WhatIUse whatIUse = new WhatIUse();
  WhatIHave whatIHave = new WhatIHave();
  WhatIWant adapt= new SurrogateAdapter(whatIHave);
```

```
  WhatIUse2 whatIUse2 = new WhatIUse2();
  WhatIHave2 whatIHave2 = new WhatIHave2();
  WhatIHave3 whatIHave3 = new WhatIHave3();
  public void test() {
    whatIUse.op(adapt);
    // Approach 2:
    whatIUse2.op(whatIHave);
    // Approach 3:
    whatIUse.op(whatIHave2);
    // Approach 4:
    whatIUse.op(whatIHave3.whatIWant());
  }
  public static void main(String args[]) {
    junit.textui.TestRunner.run(AdapterVariations.class);
  }
} ///:~
```

我想冒昧的借用一下术语“proxy”，因为在《设计模式》里，他们坚持认为一个代理（proxy）必须拥有和它所代理的对象一模一样的接口。但是，如果把这两个词一起使用，叫做“代理适配器（proxy adapter）”，似乎更合理一些。

桥接（Bridge）

研究 Bridge 模式的过程中，我发现它几乎是 GoF 描述的最差的一个模式。我开始下这个结论是在我读到 Alan Shalloway 的《Design Patterns Explained》中 Bridge 模式这一章的时候——他指出 GoF 关于 Bridge 模式的描述使他觉得一头雾水。

在一次会议上，我和两位与会者讨论过这个问题，这两位都写过关于设计模式的文章并且做过关于设计模式的演讲（包括 Bridge 模式）。这两次讨论使我对于 Bridge 模式的结构产生了完全不同的看法。

带着这些疑惑，我重新钻研 GoF 的著作，并且意识到上面那两位与会者的观点都与 GoF 书中不符。我还发现书中对于 Bridge 模式的描述实在是糟糕，描述 Bridge 模式通用结构的那张图根本不顶用，倒是描述 Bridge 模式实际例子的那个图还说的过去。你只要多看看那张图就明白 Bridge 模式到底是怎么一回事了。

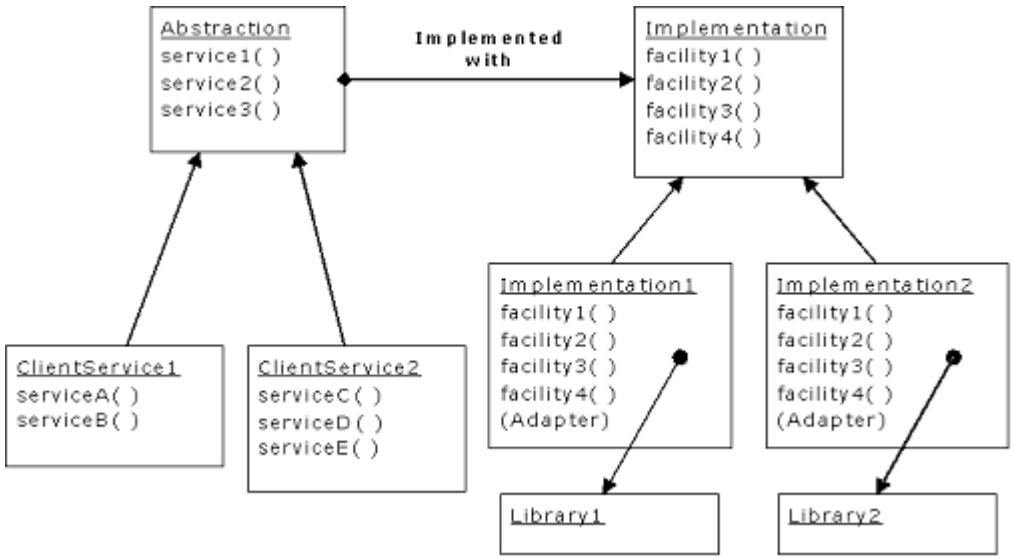
审视 Bridge 模式的时候你会发现，它的一个重要作用就是经常可以作为辅助你书写代码的一种既定结构。我们可能会根据特定的情况在编译时或运行时选择使用不同的对象，Bridge 模式的目的就是为了结构化你的代码，从而使得你可以很容易的添加新类型的前端（front-end）对象(这些前端对象是通过调用新类型的后端（back-

end）对象的功能实现的)。这么一来，就可以对前端(front-end)和后端(back-end)互不干扰地进行修改。

前端(front-end)类之间可以拥有完全不同的接口，而且通常就是这样的。它们的共同之处是可以通过使用任意数量的后端（back-end）对象来实现自身的某些功能。后端对象的接口通常也不一样。后端对象之间唯一必须相同的一点是它们都得实现某种类似的功能——例如，一组采用不同方法实现的图形库，或者一系列不同的数据存储解决方案。

Bridge 模式实际上就是一个组织代码的工具，它使得你可以添加任意数量的新的前端服务，而这些前端服务又可以通过把这些操作委托给任意数量的后端对象来实现。

通过使用 Bridge 模式，你可以避免（类）组合所带来的类的数目的爆炸省增长。但是别忘了 Bridge 模式所处理的一系列变化通常都是发生在编码阶段：当为了实现某个功能而必须处理越来越多的选项（options）时，Bridge 模式可以使你的代码保持条例性。



下面这个例子的目的就是为了说明 Bridge 模式的结构（它实现了上面那张图）：

```
//: bridge:BridgeStructure.java
// A demonstration of the structure and operation
// of the Bridge Pattern.

package bridge;
import junit.framework.*;

class Abstraction {
```

```
  private Implementation implementation;
  public Abstraction(Implementation imp) {
    implementation = imp;
  }

  // Abstraction used by the various front-end
  // objects in order to implement their
  // different interfaces.
  public void service1() {
    // Implement this feature using some
    // combination of back-end implementation:
    implementation.facility1();
    implementation.facility2();
  }
  public void service2() {
    // Implement this feature using some other
    // combination of back-end implementation:
    implementation.facility2();
    implementation.facility3();
  }
  public void service3() {
    // Implement this feature using some other
    // combination of back-end implementation:
    implementation.facility1();
    implementation.facility2();
    implementation.facility4();
  }
  // For use by subclasses:
  protected Implementation getImplementation() {
    return implementation;
  }
}

class ClientService1 extends Abstraction {
  public ClientService1(Implementation imp) { super(imp); }
  public void serviceA() {
    service1();
    service2();
```

```
  }
  public void serviceB() {
    service3();
  }
}

class ClientService2 extends Abstraction {
  public ClientService2(Implementation imp) { super(imp); }
  public void serviceC() {
    service2();
    service3();
  }
  public void serviceD() {
    service1();
    service3();
  }
  public void serviceE() {
    getImplementation().facility3();
  }
}

interface Implementation {
  // The common implementation provided by the
  // back-end objects, each in their own way.
  void facility1();
  void facility2();
  void facility3();
  void facility4();
}

class Library1 {
  public void method1() {
    System.out.println("Library1.method1()");
  }
  public void method2() {
    System.out.println("Library1.method2()");
  }
}
```

```
class Library2 {
  public void operation1() {
    System.out.println("Library2.operation1()");
  }
  public void operation2() {
    System.out.println("Library2.operation2()");
  }
  public void operation3() {
    System.out.println("Library2.operation3()");
  }
}
 class Implementation1 implements Implementation {
  // Each facility delegates to a different library
  // in order to fulfill the obligations.
  private Library1 delegate = new Library1();
  public void facility1() {
    System.out.println("Implementation1.facility1");
    delegate.method1();
  }
  public void facility2() {
    System.out.println("Implementation1.facility2");
    delegate.method2();
  }
  public void facility3() {
    System.out.println("Implementation1.facility3");
    delegate.method2();
    delegate.method1();
  }
  public void facility4() {
    System.out.println("Implementation1.facility4");
    delegate.method1();
  }
}

class Implementation2 implements Implementation {
  private Library2 delegate = new Library2();
  public void facility1() {
```

```
    System.out.println("Implementation2.facility1");
    delegate.operation1();
  }
  public void facility2() {
    System.out.println("Implementation2.facility2");
    delegate.operation2();
  }
  public void facility3() {
    System.out.println("Implementation2.facility3");
    delegate.operation3();
  }
  public void facility4() {
    System.out.println("Implementation2.facility4");
    delegate.operation1();
  }
}

public class BridgeStructure extends TestCase  {
  public void test1() {
    // Here, the implementation is determined by
    // the client at creation time:
    ClientService1 cs1 =
      new ClientService1(new Implementation1());
    cs1.serviceA();
    cs1.serviceB();
  }
  public void test2() {
    ClientService1 cs1 =
      new ClientService1(new Implementation2());
    cs1.serviceA();
    cs1.serviceB();
  }
  public void test3() {
    ClientService2 cs2 =
      new ClientService2(new Implementation1());
    cs2.serviceC();
    cs2.serviceD();
    cs2.serviceE();
```

```
  }
  public void test4() {
    ClientService2 cs2 =
      new ClientService2(new Implementation2());
    cs2.serviceC();
    cs2.serviceD();
    cs2.serviceE();
  }
  public static void main(String[] args) {
    junit.textui.TestRunner.run(BridgeStructure.class);
  }
} ///:~
```

前端基类（front-end base class）根据后端基类（back-end base class）（所声明）的方法，提供了用以实现前端派生类（front-end derived class）所需的操作。 这样一来，任何后端派生类都可以被用来实现前端类所需要的操作。 注意到桥接（bridging）是分步完成的，每一步都提供了一个抽象层。上面的例子里，Implementation 被定义为一个接口是为了强调所有实际功能都是通过后端派生类来实现的，而后端基类不参与实现。

后端派生类通过委托给（delegating）别的对象（本例中，是 Library1 和 Library2）来实现基类所定义的操作，这些接受委托的对象通常拥有完全不同的接口，但是又提供几乎相同的功能（这也是 Bridge 模式的一个重要目的）。一言以蔽之，每一个后端类都可以看作是某一个库（library）或者工具的适配器（adapter），这些库或者工具用不同的方法实现预期的功能。

练习

1. 修改 BridgeStructure.java，写一个 factroy 用来（动态）选择（后端对象）的实现。

2. 修改 BridgeStructure.java，针对前端对象，使用委托（delegation）而不是继承。这么做有什么好处和坏处？

3. 写一个 Bridge 模式的例子，要求用到关联数组（associative array）。要能够通过传入键值对象（key object）取出元素。在构造函数里从一个数组读入并初始化一系列的“键－值”对（key-value pairs）。读数据的时候，用数组(array)作为后端实现，但是写入 key-value pair 的时候，后端实现要切换到 map.

4. 用 bridge 模式，结合 java.util.collections 的 ArrayList 写一个栈（stack）和队列（queue）。写好之后，再写一个双端队列（double-ended queue）。再加入一个 LinkedList 作为后端实现。这写步骤做下来，你就会明白如何使用 Bridge 模式在影响最小的情况下往你的代码里添加前端和后端对象。

5. 用 bridge 模式写一个类，连接不同种类的记帐（bookkeeping）程序（连同它们的接口和数据格式）和不同的银行（这些银行提供不同类型的服务和接口）。

灵活的结构

组合模式（Composite）

关于 Composite 模式，很重要的一点是，所有属于部分－整体(part-whole)的这些元素都是可以被操作的，也就是说对某个节点(node)/组合(composite)的操作也同样会作用于该节点的所有子节点。GoF 在他们的书里给出了如何在基类接口里包含和访问子节点的实现细节，但看起来似乎没这个必要。下面给出的例子，Composite 类只是简单继承 ArrayList 类就实现了包含（子节点）的功能。

```
//: composite:CompositeStructure.java
package composite;
import java.util.*;
import junit.framework.*;

interface Component {
  void operation();
}

class Leaf implements Component {
  private String name;
  public Leaf(String name) { this.name = name; }
  public String toString() { return name; }
  public void operation() {
    System.out.println(this);
  }
}

class Node extends ArrayList implements Component {
  private String name;
  public Node(String name) { this.name = name; }
  public String toString() { return name; }
  public void operation() {
    System.out.println(this);
    for(Iterator it = iterator(); it.hasNext(); )
      ((Component)it.next()).operation();
  }
}
```

```
public class CompositeStructure extends TestCase {
  public void test() {
    Node root = new Node("root");
    root.add(new Leaf("Leaf1"));
    Node c2 = new Node("Node1");
    c2.add(new Leaf("Leaf2"));
    c2.add(new Leaf("Leaf3"));
    root.add(c2);
    c2 = new Node("Node2");
    c2.add(new Leaf("Leaf4"));
    c2.add(new Leaf("Leaf5"));
    root.add(c2);
    root.operation();
  }
  public static void main(String args[]) {
    junit.textui.TestRunner.run(CompositeStructure.class);
  }
} ///:~
```

上面这种方法可能是“实现 composite 模式的最简单的方法”，当然这种方法在用于大型系统的时候可能会出现问题。但是，最好的做法可能就是从最简单的方法入手，而只在需要的时候才去改变它。

系统解耦（System decoupling）

观察者模式（Observer）

和其它形式的回调函数（callback）类似，Observer 模式也允许你通过挂钩程序（hook point）改变代码。不同之处在于，从本质上说，Observer 模式是完全动态的。它经常被用于需要根据其它对象的状态变化来改变自身（状态）的场合，而且它还经常是事件管理系统（event management）的基本组成部分。无论什么时候，当你需要用完全动态的方式分离呼叫源和被呼叫代码的时候，（Observer 模式都是你的首选）。（译注：作者最后一句话好像没写完 ）

Observer 模式解决的是一个相当常见的问题：当某个对象改变状态的时候，另外一组（与之相关的）对象如何更新它们自己。比如说，Smalltalk 里的“model-view”结构,（它是 MVC(model-view-controller)结构的一部分），再比如基本与之相当的“文档－视图（document-view）”结构。假设说你有一些数据（也就是“文档”）和多于一个的视图，比如说是一个图表（plot）和一个文本视图。当你改变数据的时候，这两个视图必须知道进而（根据需要）更新它们自己，这也就是 Observer 模式所要帮你解决的问题。这个问题是如此的常见，以至于它的解决办法已经成了标准 java.util 库的一部分。

用 Java 实现 observer 模式需要用到两种类型的对象，Observable 类负责记住发生变化时需要通知哪些类，而不论“状态”改变与否。当被观察对象说“OK，你们（指观察者）可以根据需要更新你们自己了，”Observable 类通过调用 notifyObservers( )方法通知列表上的每个观察者，进而完成这个任务。notifyObservers( )是基类 Observable 的一个方法。

实际上，Observer 模式真正变化的有两样东西：观察者（observing objects）的数量和它们如何更新自己。也就是说，observer 模式使得你在不必改动其它代码的情况下只针对这两种变化更改代码。

。。。。。。（译注：作者还没写完 ）

Observer 实际上是只有一个成员函数的接口类，这个成员函数就是 update( )。当被观察者决定更新所有的观察者的时候，它就调用 update( )函数。是否需要传递参数是可选的；即使是没有参数的 Update( )函数也同样符合 observer 模式；但是，更通常的做法是让被观察者（通过 update( )函数）把引起更新的对象（也就是它自己 ）和其它有用的信息传递给观察者， 因为一个观察者可能会注册到多于一个的被观察者。这样，观察者对象就不用再费劲查找是哪个被观察者引起的更新，并且它所需要的信息也已经传递过来。

决定何时以及如何发起更新（updating）的那个“被观察者对象”被命名为 Observable。

Observable 类用一个标志（flag）来指示它自己是否改变。对于比较简单的设计来说，不用 flag 也是可以的；如果有变化，就通知所有的观察者。如果用 flag 的话，你可以使通知的时间延迟，并且由你来决定只在合适的时候通知观察者。但是，请注意，控制 flag 状态的方法是受保护的（protected），也就是说，只有（Observable 类的）派生类可以决定哪些东西可以构成一个变化,而不是由 Observer 派生类的最终用户来决定。

大多数工作是在 notifyObservers( )这个方法里完成的。如果没有将 flag 置为“已改变”，那 notifyObservers( )什么也不做；否则，它先清除 flag 的“已改变”状态，从而避免重复调用 notifyObservers( )的时候浪费时间。这些要在通知观察者之前完成，为了避免对于 update( )的调用有可能引起被观察对象的一个反向的改变。

然后 notifyObservers()方法就遍历它所保存的观察者序列，并且调用每个观察者的 update( )成员函数。

初看起来，似乎可以用一个普通的 Observable 对象来管理更新。但是实际上办不到；为了达到这个效果，你必须继承 Observable 类，并且在派生类的代码里调用 setChanged( )方法。它就是用来将 flag 置为“已改变”的那个成员函数，这么一来，当你调用 notifyObservers( )的时候所有的观察者都会被准确无误的通知道。在什么地方调用 setChanged( )，这取决于你程序的逻辑结构。

观察花朵

下面是 observer 模式的一个例子。

```
//: observer:ObservedFlower.java
// Demonstration of "observer" pattern.
package observer;
import java.util.*;
import junit.framework.*;

class Flower {
  private boolean isOpen;
  private OpenNotifier oNotify =
    new OpenNotifier();
  private CloseNotifier cNotify =
    new CloseNotifier();
  public Flower() { isOpen = false; }
  public void open() { // Opens its petals
    isOpen = true;
    oNotify.notifyObservers();
```

```
    cNotify.open();
  }
  public void close() { // Closes its petals
    isOpen = false;
    cNotify.notifyObservers();
    oNotify.close();
  }
  public Observable opening() { return oNotify; }
  public Observable closing() { return cNotify; }
  private class OpenNotifier extends Observable {
    private boolean alreadyOpen = false;
    public void notifyObservers() {
      if(isOpen && !alreadyOpen) {
        setChanged();
        super.notifyObservers();
        alreadyOpen = true;
      }
    }
    public void close() { alreadyOpen = false; }
  }
  private class CloseNotifier extends Observable{
    private boolean alreadyClosed = false;
    public void notifyObservers() {
      if(!isOpen && !alreadyClosed) {
        setChanged();
        super.notifyObservers();
        alreadyClosed = true;
      }
    }
    public void open() { alreadyClosed = false; }
  }
}

class Bee {
  private String name;
  private OpenObserver openObsrv =
    new OpenObserver();
  private CloseObserver closeObsrv =
```

```
    new CloseObserver();
  public Bee(String nm)  { name = nm; }
  // An inner class for observing openings:
  private class OpenObserver implements Observer{
    public void update(Observable ob, Object a) {
      System.out.println("Bee " + name
        + "'s breakfast time!");
    }
  }
  // Another inner class for closings:
  private class CloseObserver implements Observer{
    public void update(Observable ob, Object a) {
      System.out.println("Bee " + name
        + "'s bed time!");
    }
  }
  public Observer openObserver() {
    return openObsrv;
  }
  public Observer closeObserver() {
    return closeObsrv;
  }
}

class Hummingbird {
  private String name;
  private OpenObserver openObsrv =
    new OpenObserver();
  private CloseObserver closeObsrv =
    new CloseObserver();
  public Hummingbird(String nm) { name = nm; }
  private class OpenObserver implements Observer{
    public void update(Observable ob, Object a) {
      System.out.println("Hummingbird " + name
        + "'s breakfast time!");
    }
  }
  private class CloseObserver implements Observer{
```

```
    public void update(Observable ob, Object a) {
      System.out.println("Hummingbird " + name
        + "'s bed time!");
    }
  }
  public Observer openObserver() {
    return openObsrv;
  }
  public Observer closeObserver() {
    return closeObsrv;
  }
}

public class ObservedFlower extends TestCase {
  Flower f = new Flower();
  Bee
    ba = new Bee("A"),
    bb = new Bee("B");
  Hummingbird
    ha = new Hummingbird("A"),
    hb = new Hummingbird("B");
  public void test() {
    f.opening().addObserver(ha.openObserver());
    f.opening().addObserver(hb.openObserver());
    f.opening().addObserver(ba.openObserver());
    f.opening().addObserver(bb.openObserver());
    f.closing().addObserver(ha.closeObserver());
    f.closing().addObserver(hb.closeObserver());
    f.closing().addObserver(ba.closeObserver());
    f.closing().addObserver(bb.closeObserver());
    // Hummingbird B decides to sleep in:
    f.opening().deleteObserver(
      hb.openObserver());
    // A change that interests observers:
    f.open();
    f.open(); // It's already open, no change.
    // Bee A doesn't want to go to bed:
    f.closing().deleteObserver(
```

```
      ba.closeObserver());
    f.close();
    f.close(); // It's already closed; no change
    f.opening().deleteObservers();
    f.open();
    f.close();
  }
  public static void main(String args[]) {
    junit.textui.TestRunner.run(ObservedFlower.class);
  }
} ///:~
```

（观察者）感兴趣的事件是花朵的张开和关闭（open or close）。因为有内部类(Inner class) 可以用，所以这两种事件可以被分开观察。OpenNotifier 和 CloseNotifier 都从 Observable 继承而来，它们都可以使用 setChanged( )方法，都可以（作为 Observable 的子类）传递给需要一个 Observable 对象的地方。

使用 inner class 来定义多个 Observer 也是非常方便的，对于 Bee 和 Hummingbird（蜂鸟），它们可能需要相互独立的观察花朵的张开和关闭。请注意一下，通过使用 inner class 这种方法，我们实现了一些通过继承才能得到的好处，而且更进一步，通过 inner class 你还可以访问外部类的私有数据成员，这是继承所不能做到的。

在 Main( )函数里，你可以看到 observer 模式所带来的最大好处：通过向被观察者（Observables）动态的注册和卸载观察者对象（Observers），从而在运行时刻改变被观察者的行为。

研究一下上面那些代码你就会发现，OpenNotifier 和 CloseNotifier 只用到了 Observable 类的基本接口。（译注：这里似乎作者没写完，感觉不连贯）这就意味着你也可以继承其它完全不同的 Observer 类；观察者和花朵之间唯一的联系就是 Observer 接口。

关于 observers 的一个可视化的例子

下面这个例子很像《Thinking in Java》第二版第 14 章 ColorBoxes 那个例子。在屏幕网格上放上方块（boxes），用随机选出的颜色给这些方块着色。除了这些，每个方块都实现了 Observer 接口并注册到一个 Observable 对象。当你单击某个方块的时候，所有其它的方块都会被通知到，因为 Observable 对象自动调用每一个 Observer 对象的 update( )函数。在 update( )函数内部，方块对象会检查一下（位置信息）看自己是否与被单击的那个方块相邻，如果相邻它就把自己的颜色改的和它一样。

```
//: observer:BoxObserver.java
// Demonstration of Observer pattern using
// Java's built-in observer classes.
package observer;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;

// You must inherit a new type of Observable:
class BoxObservable extends Observable {
  public void notifyObservers(Object b) {
    // Otherwise it won't propagate changes:
    setChanged();
    super.notifyObservers(b);
  }
}

public class BoxObserver extends JFrame {
  Observable notifier = new BoxObservable();
  public BoxObserver(int grid) {
    setTitle("Demonstrates Observer pattern");
    Container cp = getContentPane();
    cp.setLayout(new GridLayout(grid, grid));
    for(int x = 0; x < grid; x++)
      for(int y = 0; y < grid; y++)
        cp.add(new OCBox(x, y, notifier));
  }
  public static void main(String[] args) {
    int grid = 8;
    if(args.length > 0)
      grid = Integer.parseInt(args[0]);
    JFrame f = new BoxObserver(grid);
    f.setSize(500, 400);
    f.setVisible(true);
    f.setDefaultCloseOperation(EXIT_ON_CLOSE);
  }
}
```

```
class OCBox extends JPanel implements Observer {
  Observable notifier;
  int x, y; // Locations in grid
  Color cColor = newColor();
  static final Color[] colors = {
    Color.BLACK, Color.BLUE, Color.CYAN,
    Color.DARK_GRAY, Color.GRAY, Color.GREEN,
    Color.LIGHT_GRAY, Color.MAGENTA,
    Color.ORANGE, Color.PINK, Color.RED,
    Color.WHITE, Color.YELLOW
  };
  static Random rand = new Random();
  static final Color newColor() {
    return colors[rand.nextInt(colors.length)];
  }
  OCBox(int x, int y, Observable notifier) {
    this.x = x;
    this.y = y;
    notifier.addObserver(this);
    this.notifier = notifier;
    addMouseListener(new ML());
  }
  public void paintComponent(Graphics g) {
    super.paintComponent(g);
    g.setColor(cColor);
    Dimension s = getSize();
    g.fillRect(0, 0, s.width, s.height);
  }
  class ML extends MouseAdapter {
    public void mousePressed(MouseEvent e) {
      notifier.notifyObservers(OCBox.this);
    }
  }
  public void update(Observable o, Object arg) {
    OCBox clicked = (OCBox)arg;
    if(nextTo(clicked)) {
      cColor = clicked.cColor;
```

```
      repaint();
    }
  }
  private final boolean nextTo(OCBox b) {
    return Math.abs(x - b.x) <= 1 &&
           Math.abs(y - b.y) <= 1;
  }
} ///:~
```

刚开始看 Observable 的联机帮助的时候，有一点可能会令你比较困惑，从文档上看似乎你可以使用一个普通的 Observable 对象来管理更新(updates)。但是实际上这么做不行；你可以试一下——更改 BoxObserver 类，（直接）创建一个 Observable 类而不是 BoxObservable 类，然后看看会发生什么：实际上，什么也不会发生。为了让 Update 生效，你必须得先继承 Observable 类，然后在派生类的代码里调用 setChanged( )方法。这个方法设置“changed”标志（flag），这就意味着当你调用 notifyObservers( )的时候，所有的观察者（observers）都会无一例外地被通知到。上面的例子里，setChanged( )方法只是简单的在 notifyObservers()里被调用，但是你可以用任何（其它）准则来决定什么时候调用 setChanged( )方法。

BoxObserver 类只包括一个 Observable 对象，就是 notifier，每次创建新的 OCBox 对象的时候，它就会和 notifier 绑定（tied）。在 OCBox 内部，无论什么时候你单击鼠标都会触发 notifyObservers( )方法被调用，然后把被单击对象当作一个参数传递给其它方块对象，这样接收这个消息（通过它们自己的 update( )方法）的方块对象就知道被单击的是哪个方块，进而决定是否需要改变自己。通过组合使用 NotifyObservers( )和 update( )方法所提供的代码，你可以实现一些相当复杂的东西。

看起来似乎只能通过 notifyObservers( )方法在编译期(compile time)决定如何通知观察者对象。但是，如果你更加仔细的看看上面那些代码,你就会发现，无论是对于 BoxObservable 还是 OCBox，只有在创建 Observable 对象的地方你才会意识到你在和 BoxObservable 打交道，创建完成之后，所有的工作都是通过 Observable 所提供的基本接口来完成的。这就意味着，如果你希望改变 notification 的工作状态（notification behavior）,你可以通过继承其它的 Observable 类，并且通过在运行时刻交换它们来实现。

中介者（Mediator）

理顺地毯下面打结的毛刺（Sweep coupling under the rug），它与 MVC 的区别在哪里呢。

MVC 有特定的 modle 和 view；而 mediator 可以是任何东西。MVC 有 mediator 的味道。

练习

1. 给出一个最简单的 Observer-Observable 设计。只需要写出两个类所必需的（minimum）那些方法即可。然后创建一个 Observable 对象和多个 Observers 对象，并用 Observable 对象来更新（update）Observers 对象。

2. 写一个非常简化的 Observer 系统，在你的 Observable 类内部使用 java.util.Timer，用以产生需要报告给 Observers 的事件。用内部类的方法创建几个不同的 Observer 对象，并把它们注册到 Observable 对象，当 Timer 事件触发时，Observers 必须得被通知到。

3. 改写 BoxObserver.java，把它变成一个简单的游戏。对于被单击的那个方块周围的其它任一方块，如果它属于某个相连的同色区域，那就把所有属于这个同色区域的方块都改为和被单击方块同样的颜色（这句累死了）。你可以通过改变游戏的配置，把它变成多玩家相竞争的游戏，或者你可以记录单个玩家完成游戏（指所有方块变为同一颜色）所用的步数。你还可以限制玩家只能以它第一步选择的那种颜色作为最终结束的颜色。

降低接口复杂度

有时候你需要解决的是很简单的问题，比如“当前的接口不是你正好需要的”。Façade 模式通过为库或者一堆资源提供一个更易用的使用方法，为一系列类创建一个接口。

外观模式（Façade）

当我想方设法试图将需求初步（first-cut）转化成对象的时候，通常我使用的原则是:“把所有丑陋的东西都隐藏到对象里去”。基本上说，Façade 干的就是这个事情。如果你有一堆让人头晕的类以及交互（Interactions），而它们又不是客户端程序员必须了解的，那你就可以为客户端程序员创建一个接口只提供那些必要的功能。

Facade 模式经常被实现为一个符合 singleton 模式的抽象工厂（abstract factory）。当然，你可以通过创建包含静态工厂方法（static factory methods）的类来达到上述效果。

```
//: facade:Facade.java
package facade;
import junit.framework.*;

class A { public A(int x) {} }
class B { public B(long x) {} }
class C { public C(double x) {} }

// Other classes that aren't exposed
// by the facade go here ...

public class Facade extends TestCase  {
  static A makeA(int x) { return new A(x); }
  static B makeB(long x) { return new B(x); }
  static C makeC(double x) { return new C(x); }
  public void test() {
    // The client programmer gets the objects
    // by calling the static methods:
    A a = Facade.makeA(1);
    B b = Facade.makeB(1);
    C c = Facade.makeC(1.0);
  }
  public static void main(String args[]) {
```

```
    junit.textui.TestRunner.run(Facade.class);
  }
} ///:~
```

《设计模式》给出的例子是通过一个类使用另外一个类（来实现 façade 模式的）。

税务顾问是你和税法（tax code）之间的 façade，他也是你和税务系统之间的中介者（mediator）。

Package 作为 facade 的变体

我感觉，façade 更倾向于“过程式的（procedural）”，也就是非面向对象的（non-object-oriented）：你是通过调用某些函数才得到对象。它和 Abstract factory 到底有多大差别呢？ Façade 模式关键的一点是隐藏某个库的一部分类（以及它们的交互），使它们对于客户端程序员不可见，这样那些类的接口就更加简练和易于理解。

其实，这也正是 java 的 packaging 功能所完成的事情：在库以外，你只能创建和使用被声明为公共（public）的那些类；所有非公共（non-public）的类只能被同一 package 的类使用。看起来，façade 似乎是 java 内嵌的一个功能。

公平起见（To be fair），《设计模式》主要是写给 C++读者的。尽管 C++有命名空间（namespaces）机制来防止全局变量和类名称之间的冲突，但它并没有提供类隐藏的机制，而在 java 里你可以通过声明 non-public 类实现这一点。我认为，大多数情况下 java 的 package 功能就足以解决针对 façade 模式的问题了。

算法分解（Algorithmic partitioning）

命令模式（Command）:运行时刻选择操作

在《Advanced C++: Programming Styles And Idioms》 (Addison-Wesley, 1992)一书中，Jim Copline 借用了 functor 这个术语，用以指代那些只为封装一个函数而构造的对象（因为“functor”在数学上有专门的含义，所以在本书中我会使用更明确的术语：函数对象（function object））。这个模式很重要的一点是，将被调用函数的选择和被调用函数的调用分离开来。

《设计模式》只是提到函数对象这个术语而没有使用它。但是，关于函数对象的话题在那本书的模式章节里倒是重复了好几次。

从本质上说，Command 就是一个函数对象：一个被封装成对象的方法[7]。通过把方法封装到一个对象，你可以把它当作参数传给其它方法或者对象，让它们在实现你的某个请求（request）的时候完成一些特殊的操作。你可能会说 Command 其实就是一个把数据成员换成行为的 messenger（因为它的目的和使用方法都很直接）。

```
//: command:CommandPattern.java
package command;
import java.util.*;
import junit.framework.*;

interface Command {
  void execute();
}

class Hello implements Command {
  public void execute() {
    System.out.print("Hello ");
  }
}

class World implements Command {
  public void execute() {
    System.out.print("World! ");
  }
}
```

[7] 在 Python 语言里，所有的函数已经是对象了，所以 Command 模式经常是多余的。

```
class IAm implements Command {
  public void execute() {
    System.out.print("I'm the command pattern!");
  }
}

// An object that holds commands:
class Macro {
  private List commands = new ArrayList();
  public void add(Command c) { commands.add(c); }
  public void run() {
    Iterator it = commands.iterator();
    while(it.hasNext())
      ((Command)it.next()).execute();
  }
}

public class CommandPattern extends TestCase  {
  Macro macro = new Macro();
  public void test() {
    macro.add(new Hello());
    macro.add(new World());
    macro.add(new IAm());
    macro.run();
  }
  public static void main(String args[]) {
    junit.textui.TestRunner.run(CommandPattern.class);
  }
} ///:~
```

Command 模式最重要的一点就是，你可以通过它把想要完成的动作（action）交给一个方法或者对象。上面的例子提供了一种方法把一系列的动作排队后集中处理。这种情况下，你可以动态地创建新的行为，而通常你只能靠编写新的代码才能做到这一点；上例中，你可以通过解释脚本来完成（如果你需要（动态改变）的行为非常复杂，那么请参看 Interpreter 模式）。

另外一个关于 Command 模式的例子是 refactor：DirList.java[????]。DirFilter 类是一个 command 对象，它的动作包含在 accept( )方法里，而这些动作又需要传递给 list( )方法。List( )方法通过调用 accept( )来决定结果中需要包含哪些东西。

《设计模式》一书写到“Commands 其实就是回调函数（callbacks）的面向对象替代品[8]（replacement）。”但是，我认为，“回（back）”对于回调（callback）的概念来说是非常关键的。也就是说，回调函数实际上会回溯到它的创建者。而另一方面，通常你只是创建 Command 对象然后把它传递给某个方法或对象，随后创建者和 Command 对象之间不再有什么联系。总之，这是我个人的一点看法。本书后面的章节，我会把一组相关的设计模式放到一起，标题是“回调”。

Strategy 模式看起来像是从同一个基类继承而来的一系列 Command 类。但是仔细看看 Command 模式，你就会发现它也具有同样的结构：一系列分层次的函数对象。不同之处在于，这些函数对象的用法和 Strategy 模式不同。就像前面 Refactor: DirList.Java 那个例子，使用 Command 是为了解决特定问题 ——从一个列表选择文件。“不变的部分”是被调用的那个方法，变化的部分被分离出来放到函数对象里。我想冒昧的下个结论：Command 模式在编码阶段提供灵活性，而 Strategy 模式的灵活性在运行时体现出来。尽管如此，这种区别却是非常模糊的。

练习

4. 用 Command 模式重做第三章的练习 1。

职责链模式（Chain of responsibility）

职责链模式可以被想象成递归（recursion）的动态泛化（generalization），这种泛化通过使用 Strategy 对象来完成。被调用的时候，链表里的每个 Strategy 对象都试图去满足这次调用。当某个策略（strategy）调用成功或者整个 strategy 链到达末尾的时候，这个过程结束。递归的时候，某个方法不断的调用它自己直到满足某个结束条件；对于职责链，某个方法调用它自己，进而（通过遍历 strategy 链表）调用它自己的不同实现，直到满足某个结束条件。所谓结束条件，要么是到达链表的末尾，要么是某一个 strategy 调用成功。对于第一种情况，得返回一个默认对象；如果不能提供一个默认结果，那就必须以某种方式告知调用者链表访问成功与否。

由于 strategy 链表中可能会有多于一个的方法满足要求，所以职责链似乎有点专家系统（expert system）的味道。因为这一系列 Strategy 实际上是一个链表，它可以被动态创建，所以你也可以把职责链想象成更一般化的，动态构建的 switch 语句。

[8] 第 235 页。

在 GoF 的书中，有很多关于如何把职责链创建成链表的讨论。但是，仔细看看这个模式你就会发现如何管理链表实际上并不重要；那只是一个实现上的细节。因为 GoF 那本书是在标准模板库（STL）被大多数 C++编译器支持之前写的，他们之所以要讨论链表的原因是（1）没有现成的链表类，所以他们得自己写一个（2）学术界常常将数据结构作为一项基本的技能来讲授，而且 GoF 那时候可能还没意识到数据结构应该成为编程语言的标准工具。我的主张是，把职责链作为链表来实现对于问题的解决没有任何裨益，它可以简单的通过使用标准的 Java List 来实现，下面的例子会说明这一点。更进一步，你会看到我还费了些劲把链表管理的部分从不同 Strategy 的实现里分离出来了，从而使这部分代码更易于重用。

前面章节 StategyPattern.java 那个例子里，很可能你需要的是自动找到一个解决问题的方案。职责链通过另外一种方法达到这种效果，它把一系列 Strategy 对象放到链表里，并提供一种机制让它自动遍历链表的每一个节点。

```
//: chainofresponsibility:FindMinima.java
package chainofresponsibility;
import com.bruceeckel.util.*; // Arrays2.toString()
import junit.framework.*;

// Carries the result data and
// whether the strategy was successful:
class LineData {
  public double[] data;
  public LineData(double[] data) { this.data = data; }
  private boolean succeeded;
  public boolean isSuccessful() { return succeeded; }
  public void setSuccessful(boolean b) { succeeded = b; }
}

interface Strategy {
  LineData strategy(LineData m);
}

class LeastSquares implements Strategy {
  public LineData strategy(LineData m) {
    System.out.println("Trying LeastSquares algorithm");
    LineData ld = (LineData)m;
    // [ Actual test/calculation here ]
    LineData r = new LineData(
```

```
      new double[] { 1.1, 2.2 }); // Dummy data
    r.setSuccessful(false);
    return r;
  }
}

class NewtonsMethod implements Strategy {
  public LineData strategy(LineData m) {
    System.out.println("Trying NewtonsMethod algorithm");
    LineData ld = (LineData)m;
    // [ Actual test/calculation here ]
    LineData r = new LineData(
      new double[] { 3.3, 4.4 }); // Dummy data
    r.setSuccessful(false);
    return r;
  }
}

class Bisection implements Strategy {
  public LineData strategy(LineData m) {
    System.out.println("Trying Bisection algorithm");
    LineData ld = (LineData)m;
    // [ Actual test/calculation here ]
    LineData r = new LineData(
      new double[] { 5.5, 6.6 }); // Dummy data
    r.setSuccessful(true);
    return r;
  }
}

class ConjugateGradient implements Strategy {
  public LineData strategy(LineData m) {
    System.out.println(
      "Trying ConjugateGradient algorithm");
    LineData ld = (LineData)m;
    // [ Actual test/calculation here ]
    LineData r = new LineData(
      new double[] { 5.5, 6.6 }); // Dummy data
```

```
    r.setSuccessful(true);
    return r;
  }
}

class MinimaFinder {
  private static Strategy[] solutions = {
    new LeastSquares(),
    new NewtonsMethod(),
    new Bisection(),
    new ConjugateGradient(),
  };
  public static LineData solve(LineData line) {
    LineData r = line;
    for(int i = 0; i < solutions.length; i++) {
      r = solutions[i].strategy(r);
      if(r.isSuccessful())
        return r;
    }
    throw new RuntimeException("unsolved: " + line);
  }
}

public class FindMinima extends TestCase  {
  LineData line = new LineData(new double[]{
    1.0, 2.0, 1.0, 2.0, -1.0, 3.0, 4.0, 5.0, 4.0
  });
  public void test() {
    System.out.println(Arrays2.toString(
      ((LineData)MinimaFinder.solve(line)).data));
  }
  public static void main(String args[]) {
    junit.textui.TestRunner.run(FindMinima.class);
  }
} ///:~
```

练习

1. 用职责链写一个专家系统，它一个接一个的尝试不同的解决方法，直到找到某个解决问题的方法为止。要求专家系统可以动态的添加解决方法。测试方法只要用字符串匹配就可以了，但是当匹配以后专家系统必须返回适当类型的 ProblemSolver 对象。这里还会用到什么其它的模式？

2. 用职责链写一个语言翻译系统，程序先访问本地的一个专用翻译系统（它可能可以针对你的问题所属的领域提供一些细节），然后访问一个更综合的通用翻译系统，如果不能完全翻译的话，最后访问 BabelFish。注意：每种翻译系统都会尽可能翻译它们能够翻的那部分。

3. 用职责链写一个重新格式化 java 源代码的工具，它通过尝试不同的方法来分行。注意，普通代码和注释可能需要区别对待，这可能需要实现一个职责树 Tree of Responsibility. 还需注意这种方法和 Composite 模式之间的相似性；或许这种技术更一般的描述应该叫做：组合策略（Composite of Strategies）。

外部化对象状态（Externalizing object state）

备忘录模式（Memento）

用序列化（serialization）写一个系统实现 undo 机制。

复杂的交互（Complex interactions）

多重分派（Multiple dispatching）

处理多种类型之间的交互可能会使程序变的相当杂乱。比如，考虑一个解析和执行数学表达式的系统。你需要支持数字+数字，数字×数字，等等，这里的数字(Number) 是一系列数字对象的基类。但是，当我们仅仅给出 a+b，我们并不知道 a 或者 b 的确切类型，那我们又如何让它们正确的交互呢？

问题的答案可能是你所没有想到的：Java 只能做单次分派（single dispatching）。也就是说，如果对多于一个的类型未知的对象进行操作，Java 只能对这些类型中的一种类型启用动态绑定机制。这并不能解决上面的问题，所以最后你还是得自己手工（写代码）帧测类型并且产生你自己的动态绑定行为。

这个方案就是多重分派(multiple dispatching)。别忘了多态只能通过成员函数调用来实现，所以如果你想实现双重分派(double dispatching)，就必须得有两次成员函数调用：第一次调用决定第一个未知类型，第二次调用决定第二个未知类型。对于多重分派，你必须得有一个可供调用的多态方法来决定所有类型。通常，你可以通过设置配置项来实现用一个成员函数调用产生多于一个的动态成员函数调用，从而在这个过程中决定多于一种的类型。为了做到这一点，你需要多于一个的多态方法调用：每一个分派都需要一个调用。下面例子里的（多态）方法是 compete( )和 eval( )，它们都是同一个类（型）的成员函数。（这种情况下，只有两次分派，也就是双重分派）。如果你需要处理的是两个体系（hierarchies）的不同类型之间的交互，那每个体系都必须得实现一个多态方法调用。

下面是一个多重分派的例子：（译注：石头剪子布）

```
//: multipledispatch:PaperScissorsRock.java
// Demonstration of multiple dispatching.
package multipledispatch;
import java.util.*;
import junit.framework.*;

// An enumeration type:
class Outcome {
  private String name;
  private Outcome(String name) { this.name = name; }
  public final static Outcome
    WIN = new Outcome("wins"),
    LOSE = new Outcome("loses"),
    DRAW = new Outcome("draws");
```

```
  public String toString() { return name; }
}

interface Item {
  Outcome compete(Item it);
  Outcome eval(Paper p);
  Outcome eval(Scissors s);
  Outcome eval(Rock r);
}

class Paper implements Item {
  public Outcome compete(Item it) { return it.eval(this); }
  public Outcome eval(Paper p) { return Outcome.DRAW; }
  public Outcome eval(Scissors s) { return Outcome.WIN; }
  public Outcome eval(Rock r) { return Outcome.LOSE; }
  public String toString() { return "Paper"; }
}

class Scissors implements Item {
  public Outcome compete(Item it) { return it.eval(this); }
  public Outcome eval(Paper p) { return Outcome.LOSE; }
  public Outcome eval(Scissors s) { return Outcome.DRAW; }
  public Outcome eval(Rock r) { return Outcome.WIN; }
  public String toString() { return "Scissors"; }
}

class Rock implements Item {
  public Outcome compete(Item it) { return it.eval(this); }
  public Outcome eval(Paper p) { return Outcome.WIN; }
  public Outcome eval(Scissors s) { return Outcome.LOSE; }
  public Outcome eval(Rock r) { return Outcome.DRAW; }
  public String toString() { return "Rock"; }
}

class ItemGenerator {
  private static Random rand = new Random();
  public static Item newItem() {
    switch(rand.nextInt(3)) {
```

```
      default:
      case 0: return new Scissors();
      case 1: return new Paper();
      case 2: return new Rock();
    }
  }
}

class Compete {
  public static void match(Item a, Item b) {
    System.out.println(
      a + " " + a.compete(b) + " vs. " + b);
  }
}

public class PaperScissorsRock extends TestCase  {
  static int SIZE = 20;
  public void test() {
    for(int i = 0; i < SIZE; i++)
      Compete.match(ItemGenerator.newItem(),
        ItemGenerator.newItem());
  }
  public static void main(String args[]) {
    junit.textui.TestRunner.run(PaperScissorsRock.class);
  }
} ///:~
```

访问者模式（Visitor）,多重分派的一种

假设说你手头有一组早先的类体系(class hierachy)，这些类都是固定不能改变的；可能它们是从第三方买来的，所以你没法改变这个类体系。但是，你可能想给这个类体系添加新的多态方法，通常情况下这必须得向基类接口添加新的东西。所以问题就来了：你既需要给基类添加新的方法，而你又不能动基类。那到底该怎么办呢？

解决这类问题的设计模式叫做“访问者(visitor)”（《设计模式》里最后讲到的那个），它是建立在上一节的双重分派机制之上的。

Visitor 模式使得你可以通过创建另外一个独立的 visitor 类型的类体系来扩展原始类型的接口，从而仿真（virtualize）实现原本针对原始类型的操作。原始类型的对象只是简单的“接受”访问者，然后调用访问者动态绑定的成员函数。

```
//: visitor:BeeAndFlowers.java
// Demonstration of "visitor" pattern.
package visitor;
import java.util.*;
import junit.framework.*;

interface Visitor {
  void visit(Gladiolus g);
  void visit(Runuculus r);
  void visit(Chrysanthemum c);
}

// The Flower hierarchy cannot be changed:
interface Flower {
  void accept(Visitor v);
}

class Gladiolus implements Flower {
  public void accept(Visitor v) { v.visit(this);}
}

class Runuculus implements Flower {
  public void accept(Visitor v) { v.visit(this);}
}

class Chrysanthemum implements Flower {
  public void accept(Visitor v) { v.visit(this);}
}

// Add the ability to produce a string:
class StringVal implements Visitor {
  String s;
  public String toString() { return s; }
  public void visit(Gladiolus g) {
```

```
    s = "Gladiolus";
  }
  public void visit(Runuculus r) {
    s = "Runuculus";
  }
  public void visit(Chrysanthemum c) {
    s = "Chrysanthemum";
  }
}

// Add the ability to do "Bee" activities:
class Bee implements Visitor {
  public void visit(Gladiolus g) {
    System.out.println("Bee and Gladiolus");
  }
  public void visit(Runuculus r) {
    System.out.println("Bee and Runuculus");
  }
  public void visit(Chrysanthemum c) {
    System.out.println("Bee and Chrysanthemum");
  }
}

class FlowerGenerator {
  private static Random rand = new Random();
  public static Flower newFlower() {
    switch(rand.nextInt(3)) {
      default:
      case 0: return new Gladiolus();
      case 1: return new Runuculus();
      case 2: return new Chrysanthemum();
    }
  }
}

public class BeeAndFlowers extends TestCase  {
  List flowers = new ArrayList();
  public BeeAndFlowers() {
```

```
    for(int i = 0; i < 10; i++)
      flowers.add(FlowerGenerator.newFlower());
  }
  public void test() {
    // It's almost as if I had a function to
    // produce a Flower string representation:
    StringVal sval = new StringVal();
    Iterator it = flowers.iterator();
    while(it.hasNext()) {
      ((Flower)it.next()).accept(sval);
      System.out.println(sval);
    }
    // Perform "Bee" operation on all Flowers:
    Bee bee = new Bee();
    it = flowers.iterator();
    while(it.hasNext())
      ((Flower)it.next()).accept(bee);
  }
  public static void main(String args[]) {
    junit.textui.TestRunner.run(BeeAndFlowers.class);
  }
} ///:~
```

练习

1. 写一个商务建模系统，它包括三种类型的 Inhabitant：侏儒（代表工程师），精灵（代表市场人员），巨人（代表经理）。写一个 Project 类，创建不同的 inhabitants 并且让他们交互，必须要用到多重分派。

2. 改写练习 1，使他们的交互更详细。每一种 Inhabitant 都可以用 getWeapon()随机的产生一个武器：侏儒用 Jargon 或者 Play，精灵用 InventFeature 或者 SellImaginaryProduct，巨人用 Edioct 和 Schedule。你必须决定在每次交互中哪个武器赢那个武器输（就像 PaperScissorsRock.java 那样）。给 Project 类添加一个 battle () 成员函数，它接受两种 Inhabitants（作为参数），使它们俩火并。再给 Project 类添加一个 meeting()成员函数，创建一组侏儒，精灵和巨人，并让各个组之间开战直到最后剩下一组获胜。

3. 改写 PaperScissorsrock.java，用表查询的方法代替双路分派。最简单的做法使写一个值为 Maps 的 Map，用每个对象的类名称来作为 Map 的键。然后通过下面这种方法查找：

```
((Map)map.get(o1.getClass())).get(o2.getClass())
```

使用这种方法非常容易重新配置系统。什么时候使用这种方法会比硬编码的动态分发更合适呢？你能否用表查找的方法写一个语法简单的系统来实现动态分派？

4. 用练习 3 那种表查找的方法重做练习 2。

多个编程语言（Multiple languages）

本章我们讨论跨越语言边界所带来的好处。通常来说，对于某个问题的解决，使用多于一种的编程语言比死抱住一门语言不放，会更便利。本章你将会看到，对于某种语言而言非常困难或者棘手的问题，如果用另外一种语言就可能很容易并且很快被解决掉。如果能够把多种语言结合起来，你就可以更快更节约的创造产品。

上面这个想法最直接的应用就是被称为 Interpreter 的设计模式，它可以给你的程序添加一个解释语言（interpreted language），从而允许最终用户很容易的定制一个解决方案。对于 Java 来说，实现上述想法最简单和最强大的方式就是通过 Jython[9]，它是用 Java 的纯字节码写的一个 Python 语言的实现。

Interpreter 解决的是一个特定的问题——为用户创建一个脚本语言。但是有时候暂时转到另外一种语言解决你所面临的问题的某一方面会更简单和快捷。并不是说要你创建一个解释器，你只需要使用另外一种语言写一些代码就可以了。再次重申一下，Jython 是这方面很好的一个例子，而 CORBA 也允许你跨越语言边界。

Interpreter 模式的动机

如果程序的用户在运行时刻需要更大的灵活性，例如，为了通过创建脚本来描述所期望的系统行为，你就可以使用 Interpreter 设计模式。这种情况下，你（需要）创建一个语言解释器并把它嵌入到你的程序中。

别忘了，每种设计模式都允许一个或多个可以变化的因素，所以，重要的是首先要知道哪个因素是变化的。有些时候是你的程序的最终用户（而不是程序的编写人员）在他们配置程序的某些方面的时候需要更大的灵活性。也就是说，他们需要做一些简单的编程工作。Interpreter 模式通过添加一个语言解释器提供了这种灵活性。

问题是，开发你自己的语言并为它构建一个解释器是一项耗时的工作，而且它会分散你开发应用程序的精力。你必须问问自己是想要写完应用程序呢还是要创造一门新的语言。最好的解决方案就是代码重用：嵌入一个已经构建好并且调试过的解释器。Python 语言可以被免费的嵌入以盈利为目的的应用程序，而不需要签署任何许可协议和支付版税，也不必遵循任何的附加条件。基本上说，使用 Python 的时候你不受任何限制。

Python 是一门非常容易学习的语言，它的逻辑性很强，便于读写，它支持函数和对象，有一大堆可用的库，几乎可以在所有平台上运行。你可以到 http://www.python.org/下载 Python 并在那里找到更多的信息。

[9] 最初的版本叫做 JPython，但是后来项目变动之后，为了强调新版本的特性，就把这个名字也改了。

为了解决与 Java 相关的问题，我们会着眼于 Python 的一个特殊版本 Jython。它是完全由 Java 的字节码生成的，所以要把它并入你的应用程序非常简单，而且它和 Java 一样有很好的可移植性。它和 Java 之间有一个非常干净的接口：Java 可以调用 Python 类，Python 也可以调用 Java 类。

Python 从一开始就是用类来设计的，它是一门真正的纯面向对象的语言（C++和 Java 都以不同的方式违反了纯面向对象的原则）。由于 Python 是递增的（scales up），所以即使你创建非常大的程序也不会失去对其代码的控制。

要安装 Python，访问 www.Python.org 按那些链接和说明做就可以了。要安装 Jython，访问 http://jython.sourceforge.net/下载下来的是一个.class 文件，你用 Java 执行它的时候它会启动一个安装程序。你还需要把 jython.jar 添加到你的 CLASSPATH。你可以访问 http://www.bruceeckel.com/TIPatterns/Building-Code.html 找到更进一步的安装说明。

Python 概览

为了让你能够上手，下面是关于 Python 的一个简短介绍，这个介绍是针对有经验的程序员的（如果你正在读这本书那你必须得是个有经验的程序员）。你可以参考 http://www.python.org/上更为全面的 Python 文档（尤其是非常有用的那个 HTML 页面：Python 快速参考手册），此外，你还可以参考众多的 Python 书籍，比如 Mark Lutz 和 David Aschor 写的《Learning Python》(O'Reilly, 1999)。

Python 经常被说成是一种脚本语言，但是脚本语言似乎有很大的局限性，尤其是就它们所解决的问题的领域而言。与此不同，Python 是一门支持脚本编程的语言。用它来写脚本确实是很棒的，你甚至会想用 Python 脚本替换掉你所有的批处理文件，shell 脚本和一些简单的程序。但是它远远超出了一门脚本语言。

Python 被设计成这样一门语言，用它写出来的代码非常干净，而且很容易读懂。你会发现即使过了很长时间你还是很容易读懂自己写的那些代码，而且读别人的代码也是如此。这在某种程度上得归功于它简明扼要的语法，但是对于代码可读性来说很重要的一个因素是缩进——对于 Python 来说，作用域是通过缩进来确定的。例如：

```
##interpreter:if.py
response = "yes"
if response == "yes":
  print "affirmative"
  val = 1
print "continuing..."
##~
```

‘#’号表示直到该行结尾的一个注释，就像 C++和 Java 里的‘//’注释那样。

首先我们注意到 Python 的基本语法是 C 风格的；请注意上面的 if 语句。但是 C 语言的 if 语句，你必须得用括号把条件语句括起来，而在 Python 里这么做并不是必须的（但是如果你用了括号它也不会抱怨）。

条件语句是以一个冒号结尾的，这意味着紧跟着的是一组缩进的语句，也就是 if 语句的“then”那一部分。上面的例子里，先是一个“print”语句把结果送到标准输出，接下来是给一个叫做 val 的变量赋值的语句。再接下来的语句就没有缩进了，所以它不是 if 语句的一部分。缩进可以嵌套到任何一级，就像 C++或者 Java 里的大括号，但是和 C++，Java 不同的是这里没有括号该放在哪里的问题（也就没有争论）——编译器强制使所有人的代码都以同一种方式格式化，这也是 Python 具有很强的可读性的一个主要原因。

通常 Python 每行只有一个语句（你可以通过使用分号在一行放置多个语句并把它们分开），这样标志语句结束的那个分号就没有必要了。即使看看上面那个简短的例子，你也会发现这门语言被设计的尽可能的简单，而同时又具有良好的可读性。

内置容器

对于像 C++和 Java 那样的语言，容器是以库的形式而附加的，并没有和语言集成在一起。在 Python 里，容器之于编程语言的重要地位（essential nature）是通过把它们构建入语言核心来使其得到承认的：链表（lists）和关联数组（即映射表，字典，哈希表）都成了基本的数据类型。这使得这门语言更加优雅。

除此之外，for 语句自动遍历链表，而不仅仅是 couting through a sequence of numbers。仔细想想，这其实很有意义，因为大多数情况下你都是用一个 for 循环来单步遍历（step through）一个数组或容器。Python 通过自动地让 for 语句使用一个作用在（works through）某个序列上的迭代器把 for 语句的这种用法标准化了。下面是一个例子：

```
## interpreter:list.py
list = [ 1, 3, 5, 7, 9, 11 ]
print list
list.append(13)
for x in list:
  print x
##~
```

第一行代码创建了一个链表。你可以把它打印出来，打印的结果应该和你放进去的一摸一样（作为对比，回忆一下在《Thinking in Java》第二版里，为了把数组打

印出来我们必需得创建一个特别的 Array2 类）。这里的链表类似于 Java 里的容器类——你可以往里面添加新的元素（上例中，是用 append()来添加的），链表会自动调整自己的大小。For 语句创建了 x 迭代器，它会作用到链表的每一个元素。

你可以使用 range()函数创建一个存放数字的链表，所以如果你真的需要模拟 C 里面的 for 语句，你就可以这么做。请注意，上面的代码没有任何类型声明——对象只要有名字就可以了，Python 会根据你使用它们的方式推断出其类型。感觉上好像设计 Python 的目的就是为了让你仅在绝对必要的时候才敲一下键盘。使用 Python 一小段时间以后，你就会发现，对于非 Python 的编程语言来讲必不可少、而又让你绞尽脑汁的分号、大括号和其它额外的谓词，实际并没有描述程序的意图。

函数

在 Python 里创建一个函数，需要用到 def 关键字，接下来是函数名称和参数列表，然后是一个冒号表示函数体的开始。下面的代码把上例改写成一个函数：

```
## interpreter:myFunction.py
def myFunction(response):
  val = 0
  if response == "yes":
    print "affirmative"
    val = 1
  print "continuing..."
  return val

print myFunction("no")
print myFunction("yes")
##~
```

注意到函数签名（function signature）里并没有类型信息——它只是指定了函数的名称和参数标识，但是并没有参数类型或者返回类型。Python 是一种弱类型（weakly-typed）语言，也就是说它对类型信息的要求降到了最低。比如，你可以针对同一个函数传入和返回不同的类型。

```
## interpreter:differentReturns.py
def differentReturns(arg):
  if arg == 1:
    return "one"
  if arg == "one":
    return 1
```

```
print differentReturns(1)
print differentReturns("one")
##~
```

对于传入某个函数的对象来说，唯一的限制是，函数的操作必须得能够应用到这个对象上，除此之外，函数并不关心对象的其它东西。上例中，同一个函数把‘＋’操作符应用到了数字和字符串。

```
### interpreter:sum.py
def sum(arg1, arg2):
  return arg1 + arg2

print sum(42, 47)
print sum('spam ', "eggs")
##~
```

当‘＋’操作符应用到字符串的时候，它的意思是，把两个字符串连接起来（是的，Python 支持操作符重载，而且它在这方面做的不错）。

字符串

上面的例子还展示了一点点 Python 的字符串处理功能，它在这方面是我所见过的语言里做的最好的。你可以使用单引号或者双引号来表示字符串，这一点是非常好的，因为如果你用双引号来把一个字符串引起来，你就可以在其中嵌入单引号，反过来也是如此。

```
## interpreter:strings.py
print "That isn't a horse"
print 'You are not a "Viking"'
print """You're just pounding two
coconut halves together."""
print '''"Oh no!" He exclaimed.
"It's the blemange!"'''
print r'c:\python\lib\utils'
##~
```

请记住 Python 并不是根据蛇的名字来命名的，实际上它是一个 Monty 大蟒蛇飞行马戏团，所以上面的例子理论上来说都需要包括 Python-esque references.

三个双引号连用，这种语法表示把它们之间的所有东西都引起来，包括新的行。这在处理诸如产生 web 页面（Python 是一门非常棒的 CGI 语言）等工作的时候尤为有用，因为你可以简单的使用三个连续的双引号把你想要的整个页面都引起来，而不用做任何其它的编辑。

一个字符串右边的 r 字符表示这个字符串是个“raw 字符串”，也就是说按照字面来解释反斜杠，所以你就不用再加一个额外的反斜杠了。

Python 的字符串替换出奇的简单，因为它使用了类似于 C 里面 printf( )的替换语法，对于任何字符串，你只需要在字符串后面加上一个‘%’和用来做替换的值。

```
## interpreter:stringFormatting.py
val = 47
print "The number is %d" % val
val2 = 63.4
s = "val: %d, val2: %f" % (val, val2)
print s
##~
```

你可以看到上面第二种情况下，如果有多于一个的参数，你可以用括号把它们括起来（这组成了一个垫片（tuple），也就是一个不能作改动的链表）。

所有适用于 printf() 的格式化操作都可以用在这里，包括对小数位的个数和对齐的控制。Python 还有非常复杂的正则表达式（语法）。

类

像 Python 里面的其它东西一样，定义一个类只需要用到最少量的额外语法。使用 class 关键字(定义类)，在类定义体的内部使用 def 创建方法（methods）。下面是一个例子：

```
## interpreter:SimpleClass.py
class Simple:
  def __init__(self, str):
    print "Inside the Simple constructor"
    self.s = str
  # Two methods:
  def show(self):
```

```
    print self.s
  def showMsg(self, msg):
    print msg + ':',
    self.show() # Calling another method

if __name__ == "__main__":
  # Create an object:
  x = Simple("constructor argument")
  x.show()
  x.showMsg("A message")
##~
```

上面两个方法都把“self”作为它们的第一个参数。C++和 Java 的类方法都有一个隐藏的第一参数，也就是指向调用这个方法的对象的一个指针，可以通过 this 关键字来访问。Python 方法也使用了当前对象的引用，但是当定义一个方法的时候，你必须显式的把这个引用指定为第一个参数。传统上，这个引用被称作 self，然而你也可以使用任何你想用的标示符（但是如果你不用 self，很可能会让很多人感到迷惑）。如果你需要用到对象的 field 或者它的其它方法，那你必须在表达式里使用 self。但是，当你是通过像 x.show( )这样调用某个对象的方法的时候，并不需要把 self 引用传给对象——Python 已经帮你做好了。

上面的第一个方法有它的特别之处，所有以双下划线开始并且结束的标示符都是特殊的。对于上面的情况来说，它定义了构造函数，当创建对象的的时候它会自动被调用，就像 C++和 Java 那样。然而，在上例下半部分你会看到，对象的创建就像一次使用类名称的函数调用。Python 宽松（spare）的语法会让你觉得 C++或者 Java 里的 new 关键字其实都不是必需的。

后一部分的所有代码被一个 if 语句隔开（set off） 了，这个 if 语句检查__name__这个东西是不是等于__main__。再说一次，双下划线意味着特殊的名字。使用 if 的原因是因为每一个文件都有可能在另外的程序里被用作库模块（马上就会讲到模块）。在那种情况下，你只需要那些定义好了的类，而不想让文件下半部分的代码被执行。这个特殊的 if 语句只有当你直接运行这个文件的时候才为真，也就是说，如果你在命令行输入：

```
    Python SimpleClass.py
```

然而，如果这个文件是被另外一个程序作为一个模块引入的，那__main__的代码就不会被执行。

可能会让你觉得有点奇怪的是，你得在方法内部定义成员变量（fields），而不是像 C++或者 Java 那样在方法外部定义（如果你像在 C++/Java 里那样定义成员变量，它们就隐含的称为静态成员变量）。要创建一个对象的成员变量，你只需要在某个方法内部（通常是在构造函数里，但也并非全部如此）——使用 self 关键字——给它起个名字，当运行那个方法的时候会为成员变量分配空间。在 C++或 Java 看来，这似乎有点奇怪，因为对于 C++或 Java 来说你必须提前决定对象需要占用多少空间，但是 Python 的做法已经证明这是一种非常灵活的编程方法。

继承

因为 Python 是弱类型的（weakly typed），所以它并不真正关心接口——它所关心的只是把操作应用到对象上（实际上，Java 的 interface 关键字在 Python 里是个浪费）。这就是说 Python 的继承和 C++或者 Java 是不同的，对后两者而言经常是通过继承简单的建立一个公共接口。而对 Python 来说，使用继承的唯一理由是为了继承一个实现——为了重用属于基类的代码。

如果你打算从某个类继承下来，你必须告诉 Python 把那个类引入的你的新文件。Python 像 Java 那样对命名空间（name spaces）进行强有力的（aggressively）控制，而且风格也和 Java 类似（尽管如此，Python 保留着它一贯的简单性）。每次创建一个文件的时候，你就隐含的创建了一个和该文件同名的模块（类似于 Java 里的 package）。这么一来，package 关键字在 Python 里也不需要了。当你想要使用某个模块的时候，只要用 import 关键字，并且给出模块的名字就可以了。Python 会搜索 PYTHONPATH，Java 用同样的方法搜索 CLASSPATH（但是由于某种原因，Python 并没有像 Java 那样的缺陷），然后读取搜索到的文件。要引用某个模块的函数或者类，你需要给出模块名称，接着是一个句号，然后是函数名或者类名。如果你觉得给出确切的名称太麻烦，你可以用：

```
    #from module import name(s)
```

这里，“names(s)”可以是由逗号分开的一组名称。

你可以通过在继承类（inheriting class）后面的括号里列出被继承类的名字，让继承类继承自某个类（或者是多个类——Python 支持多继承）。请注意，Simple 类属于 SimpleClass 文件（也就是模块），通过使用 import 语句把它引入新的命名空间。

```
## interpreter:Simple2.py
from SimpleClass import Simple


class Simple2(Simple):
  def __init__(self, str):
    print "Inside Simple2 constructor"
```

```
    # You must explicitly call
    # the base-class constructor:
    Simple.__init__(self, str)
  def display(self):
    self.showMsg("Called from display()")
  # Overriding a base-class method
  def show(self):
    print "Overridden show() method"
    # Calling a base-class method from inside
    # the overridden method:
    Simple.show(self)

class Different:
  def show(self):
    print "Not derived from Simple"

if __name__ == "__main__":
  x = Simple2("Simple2 constructor argument")
  x.display()
  x.show()
  x.showMsg("Inside main")
  def f(obj): obj.show() # One-line definition
  f(x)
  f(Different())
##~
```

Simple2 继承自 Simple，它在构造函数里调用基类的构造函数。在 display()方法里可以把 showMsg()作为 self 的一个方法来调用，但是当调用这个方法的基类版本的时候，就得用到覆写（overriding）了，你必须给出完整的名字（译注：类名称加方法名称）并且把 self 作为第一个参数传给它，就像调用基类的构造函数那样。show()方法覆写过的版本也是这么做的。

在__main__函数里，你会看到（当运行这个程序的时候）基类的构造函数会被调用。你还会看到派生类可以调用 showMsg( )方法，这正是你期望通过继承所达到的效果。

Different 类也有一个叫作 show( )的方法，但是这个类不是由 Simple 类继承而来的。__main__函数里的 f( )方法描述了什么是弱类型（weak typing）：它所关心的

只是 show( )方法可以被应用到 obj 上，而对类型并没有任何其它的要求。你会看到 f( )可以被同样地应用到一个继承自 Simple 的对象和一个不是继承自 Simple 的对象，这两者并没有什么区别。 如果你是个 C++程序员，你应该知道 C++的模板（template）功能是为了在一个强类型语言里提供弱类型支持。但在 Python 里你自动的就得到了与模板等效的功能——而不必学习非常麻烦的语法和语义（semantics）。

创造一门语言

使用 Jython，在你的程序内部创造一种解释（interpreted）语言是极其简单的。考虑 《Thinking in Java》第二版第八章 greenhouse 控制器的那个例子。那种情形下你会希望最终用户——也就是管理 greenhouse 的那个人——能够通过配置来控制整个系统，于是一个简单的脚本语言就成了理想的解决方案。

要创建这个语言，我们只需简单的写一组 Python 类，每个类的构造函数会把它自己添加到一个（静态的）master 链表。公共的数据和行为会被 factor 到叫作 Event 的基类。每个 Event 对象都会包括一个 action 字符串（简单起见——实际应用中，你应该安排一些 functionality）和运行该事件（event）的设定时间。构造函数初始化这些成员变量，然后把新创建的 Event 对象添加到一个叫作 events（在类的内部定义，但是位于所有方法的外部，所以它是静态的）的静态链表.

```
#:interpreter:GreenHouseLanguage.py


class Event:
  events = [] # static
  def __init__(self, action, time):
    self.action = action
    self.time = time
    Event.events.append(self)
  # Used by sort(). This will cause
  # comparisons to be based only on time:
  def __cmp__ (self, other):
    if self.time < other.time: return -1
    if self.time > other.time: return 1
    return 0
  def run(self):
    print "%.2f: %s" % (self.time, self.action)


class LightOn(Event):
  def __init__(self, time):
    Event.__init__(self, "Light on", time)
```

```
class LightOff(Event):
  def __init__(self, time):
    Event.__init__(self, "Light off", time)

class WaterOn(Event):
  def __init__(self, time):
    Event.__init__(self, "Water on", time)

class WaterOff(Event):
  def __init__(self, time):
    Event.__init__(self, "Water off", time)

class ThermostatNight(Event):
  def __init__(self, time):
    Event.__init__(self,"Thermostat night", time)

class ThermostatDay(Event):
  def __init__(self, time):
    Event.__init__(self, "Thermostat day", time)

class Bell(Event):
  def __init__(self, time):
    Event.__init__(self, "Ring bell", time)

def run():
  Event.events.sort();
  for e in Event.events:
    e.run()

# To test, this will be run when you say:
# python GreenHouseLanguage.py
if __name__ == "__main__":
  ThermostatNight(5.00)
  LightOff(2.00)
  WaterOn(3.30)
  WaterOff(4.45)
  LightOn(1.00)
```

```
  ThermostatDay(6.00)
  Bell(7.00)
  run()
##~
```

每个派生类的构造函数都要调用基类的构造函数，基类的构造函数把新建的对象加入到链表。run( )函数给整个链表排序，它自动的使用 Event 类所定义的__cmp__( )方法只根据时间来进行比较。上面的例子，只是打印链表的内容，但是在实际系统中，它会等待每个事件设定的时间，然后运行那个事件。

__main__函数这部分对这些类做了一下简单的测试。

上面的文件现在已经是一个模块了，它可以被包括进（included）另外一个 Python 程序，用以定义那个程序所包含的所有类。但是，我们不用普通的 Python 程序，我们使用 Java 的 Jython。这其实是非常简单的：你只需引入一些 Jython 类，创建一个 PythonInterpreter 对象，然后把 Python 文件加载进来：

```
//- interpreter:GreenHouseController.java
package interpreter;
import org.python.util.PythonInterpreter;
import org.python.core.*;
import junit.framework.*;

public class
GreenHouseController extends TestCase  {
  PythonInterpreter interp =
    new PythonInterpreter();
  public void test() throws PyException  {
    System.out.println(
      "Loading GreenHouse Language");
    interp.execfile("GreenHouseLanguage.py");
    System.out.println(
      "Loading GreenHouse Script");
    interp.execfile("Schedule.ghs");
    System.out.println(
      "Executing GreenHouse Script");
    interp.exec("run()");
  }
  public static void
```

```
  main(String[] args) throws PyException  {
    junit.textui.TestRunner.run(GreenHouseController.class);
  }
} ///:~
```

PythonInterpreter 对象是一个完整的 Python 解释器，它可以从 Java 程序接受命令。其中一个命令是 execfile( )，这个命令告诉解释器去执行它所找到的特定文件所包含的所有语句。通过执行 GreenHouseLanguage.py，那个文件里所有的类都会被加载到 PythonInterpreter 对象，于是解释器就“拥有了” greehouse 控制器语言。Schedule.ghs 是由最终用户创建的用来控制 greenhouse 的文件。下面是一个例子：

```
//:! interpreter:Schedule.ghs
Bell(7.00)
ThermostatDay(6.00)
WaterOn(3.30)
LightOn(1.00)
ThermostatNight(5.00)
LightOff(2.00)
WaterOff(4.45)
///:~
```

这就是 interpreter 设计模式所要达到的目的：使你的程序的配置对于最终用户来说尽可能的简单。使用 Jython 你几乎可以毫不费力的达到这个效果。

PythonInterpreter 还有一个可供使用的方法是 exec( )，通过它你可以向解释器发送命令。上例中，run( )函数就是通过 exec( ).被调用的。

记住，要运行这个程序，你必须到 http://jython.sourceforge.net/下载并运行 Jython（实际上，你只需要把 jython.jar 放到你的 CLASSPATH 就可以了）。上面这些做好以后，它就可以像其它 Java 程序那样运行了。

控制解释器（Controlling the interpreter）

前面的例子只是创建解释器并且让它运行外部脚本。本章的剩余部分，我们会讲述更为复杂的与Ｐｙｔｈｏｎ交互的方法。要在 Java 范围内对 PythonInterpreter 施加更多的控制，最简单的方法就是，发送数据给解释器，然后再从它取回数据（pull data back out）。

提交数据（Putting data in）

为了向你的 Python 程序插入数据，PythonInterpreter 类有一个看似简单的方法：set( )。实际上，set( )可以接受不同类型的数据并且对它们进行转换。下面的例子是一个关于 set( )不同用法的相当全面的练习，一道给出的那些注释提供了相当完整的解释。

```
//- interpreter:PythonInterpreterSetting.java
// Passing data from Java to python when using
// the PythonInterpreter object.
package interpreter;
import org.python.util.PythonInterpreter;
import org.python.core.*;
import java.util.*;
import com.bruceeckel.python.*;
import junit.framework.*;

public class
PythonInterpreterSetting extends TestCase  {
  PythonInterpreter interp =
    new PythonInterpreter();
  public void test() throws PyException  {
    // It automatically converts Strings
    // into native Python strings:
    interp.set("a", "This is a test");
    interp.exec("print a");
    interp.exec("print a[5:]"); // A slice
    // It also knows what to do with arrays:
    String[] s = { "How", "Do", "You", "Do?" };
    interp.set("b", s);
    interp.exec("for x in b: print x[0], x");
    // set() only takes Objects, so it can't
    // figure out primitives. Instead,
    // you have to use wrappers:
    interp.set("c", new PyInteger(1));
    interp.set("d", new PyFloat(2.2));
    interp.exec("print c + d");
    // You can also use Java's object wrappers:
    interp.set("c", new Integer(9));
```

```
interp.set("d", new Float(3.14));
interp.exec("print c + d");
// Define a Python function to print arrays:
interp.exec(
  "def prt(x): \n" +
  "  print x \n" +
  "  for i in x: \n" +
  "    print i, \n" +
  "  print x.__class__\n");
// Arrays are Objects, so it has no trouble
// figuring out the types contained in arrays:
Object[] types = {
  new boolean[]{ true, false, false, true },
  new char[]{ 'a', 'b', 'c', 'd' },
  new byte[]{ 1, 2, 3, 4 },
  new int[]{ 10, 20, 30, 40 },
  new long[]{ 100, 200, 300, 400 },
  new float[]{ 1.1f, 2.2f, 3.3f, 4.4f },
  new double[]{ 1.1, 2.2, 3.3, 4.4 },
};
for(int i = 0; i < types.length; i++) {
  interp.set("e", types[i]);
  interp.exec("prt(e)");
}
// It uses toString() to print Java objects:
interp.set("f", new Date());
interp.exec("print f");
// You can pass it a List
// and index into it...
List x = new ArrayList();
for(int i = 0; i < 10; i++)
    x.add(new Integer(i * 10));
interp.set("g", x);
interp.exec("print g");
interp.exec("print g[1]");
// ... But it's not quite smart enough
// to treat it as a Python array:
interp.exec("print g.__class__");
```

```
    // interp.exec("print g[5:]); // Fails
    // If you want it to be a python array, you
    // must extract the Java array:
    System.out.println("ArrayList to array:");
    interp.set("h", x.toArray());
    interp.exec("print h.__class__");
    interp.exec("print h[5:]");
    // Passing in a Map:
    Map m = new HashMap();
    m.put(new Integer(1), new Character('a'));
    m.put(new Integer(3), new Character('b'));
    m.put(new Integer(5), new Character('c'));
    m.put(new Integer(7), new Character('d'));
    m.put(new Integer(11), new Character('e'));
    System.out.println("m: " + m);
    interp.set("m", m);
    interp.exec("print m, m.__class__, " +
      "m[1], m[1].__class__");
    // Not a Python dictionary, so this fails:
    //! interp.exec("for x in m.keys():" +
    //!   "print x, m[x]");
    // To convert a Map to a Python dictionary,
    // use com.bruceeckel.python.PyUtil:
    interp.set("m", PyUtil.toPyDictionary(m));
    interp.exec("print m, m.__class__, " +
      "m[1], m[1].__class__");
    interp.exec("for x in m.keys():print x,m[x]");
  }
  public static void
  main(String[] args) throws PyException  {
    junit.textui.TestRunner.run(
      PythonInterpreterSetting.class);
  }
} ///:~
```

对于 Java 来说，大多数时候真正的对象（real objects）和基元类型（primitive types）之间的差别总会带来麻烦。一般而言，如果你传递给 set( ),方

法的是一个通常的对象（regular object），它是知道如何处理的，但是如果你想传入一个基本类型（primitive type）的对象，就必须得作转换。一种做法是创建一个“Py”类型，比如 PyInteger 和 PyFloat，但实际上你也可以使用 Java 自带的对象外覆类比如 Integer 和 Float，这些可能更容易记住。

上面程序的前一部分你会看到有一个 exec( )含有以下的 Python 语句：

```
    pint a[5:]
```

索引语句里的那个分号表明这是一个 Python 切片（slice），所谓切片就是从一个原始数组里产生出某一范围内的元素。在这里，它产生出一个包含从第 5 号元素开始直到原来数组最后一个元素的新数组。你也可以用“a[3:5]”产生第 3 号到第 5 号元素，或者用“a[:5]”产生第 0 号到第 5 号元素。在这个语句里使用切片的原因是为了保证 Java 的 String 确实被转换成了一个 Python 字符串，而 Python 字符串是可以被当作一个字符数组来对待的。

你会看到我们是能够用 exec( ),来创建一个 Python 函数的（虽然有点别扭）。prt( )函数打印整个数组，然后（to make sure it’s a real Python array）遍历数组的每一个元素并且把它打印出来。最后，它打印数组的类名称，我们可以据此看看发生是什么转换（Python 不仅有运行时刻信息，它还有与 Java 的反射相当的东西）。prt( )函数用以打印来自 Java 的基本类型的数组。

尽管可以使用 set( )把一个 Java 的 ArrayList 传入解释器，而且你也能把它当作数组那样进行索引，但是试图从它产生一个切片是不会成功的。为了把它完全转换成一个数组，一种方法是简单的利用 toArray( )从中取出一个 Java 数组，然后再把它传给解释器。set( )方法会把它转换成一个 PyArray ——Jython 提供的一个类——它可以被当作一个 Python 数组来处理（你也可以显式的创建一个 PyArray，但是似乎没有这个必要）。

最后，我们创建了一个 Map 并且把它直接传给了解释器。虽然可以对 While it is possible to do simple things like index into the resulting object，但它并不是一个真正的 Python 字典（dictionary），所以你不能调用像 keys( )那样的方法。并没有直接了当的方法可以把一个 Java 的 Map 转换成一个 Python 字典，于是我就写了一个叫做 toPyDictionary( )的小程序并且把它作为 com.bruceeckel.python.PyUtil 的一个静态方法。它还包括一些从 Python 数组提取数据到 Java List 和从 Python 字典提取数据到 Java Map 的一些小程序。

```
//- com:bruceeckel:python:PyUtil.java
// PythonInterpreter utilities
package com.bruceeckel.python;
import org.python.util.PythonInterpreter;
import org.python.core.*;
```

```
import java.util.*;

public class PyUtil {
  /** Extract a Python tuple or array into a Java
  List (which can be converted into other kinds
  of lists and sets inside Java).
  @param interp The Python interpreter object
  @param pyName The id of the python list object
  */
  public static List
  toList(PythonInterpreter interp, String pyName){
    return new ArrayList(Arrays.asList(
      (Object[])interp.get(
        pyName, Object[].class)));
  }
  /** Extract a Python dictionary into a Java Map
  @param interp The Python interpreter object
  @param pyName The id of the python dictionary
  */
  public static Map
  toMap(PythonInterpreter interp, String pyName){
    PyList pa = ((PyDictionary)interp.get(
      pyName)).items();
    Map map = new HashMap();
    while(pa.__len__() != 0) {
      PyTuple po = (PyTuple)pa.pop();
      Object first = po.__finditem__(0)
        .__tojava__(Object.class);
      Object second = po.__finditem__(1)
        .__tojava__(Object.class);
      map.put(first, second);
    }
    return map;
  }
  /** Turn a Java Map into a PyDictionary,
  suitable for placing into a PythonInterpreter
  @param map The Java Map object
  */
```

```
  public static PyDictionary
  toPyDictionary(Map map) {
    Map m = new HashMap();
    Iterator it = map.entrySet().iterator();
    while(it.hasNext()) {
      Map.Entry e = (Map.Entry)it.next();
      m.put(Py.java2py(e.getKey()),
        Py.java2py(e.getValue()));
    }
    // PyDictionary constructor wants a Hashtable:
    return new PyDictionary(new Hashtable(m));
  }
} ///:~
```

下面是（黑盒）单元测试的代码：

```
//- com:bruceeckel:python:Test.java
package com.bruceeckel.python;
import org.python.util.PythonInterpreter;
import java.util.*;
import junit.framework.*;

public class Test extends TestCase  {
  PythonInterpreter pi =
    new PythonInterpreter();
  public void test1() {
    pi.exec("tup=('fee','fi','fo','fum','fi')");
    List lst = PyUtil.toList(pi, "tup");
    System.out.println(lst);
    System.out.println(new HashSet(lst));
  }
  public void test2() {
    pi.exec("ints=[1,3,5,7,9,11,13,17,19]");
    List lst = PyUtil.toList(pi, "ints");
    System.out.println(lst);
  }
  public void test3() {
    pi.exec("dict = { 1 : 'a', 3 : 'b', " +
```

```
      "5 : 'c', 9 : 'd', 11 : 'e'}");
    Map mp = PyUtil.toMap(pi, "dict");
    System.out.println(mp);
  }
  public void test4() {
    Map m = new HashMap();
    m.put("twas", new Integer(11));
    m.put("brillig", new Integer(27));
    m.put("and", new Integer(47));
    m.put("the", new Integer(42));
    m.put("slithy", new Integer(33));
    m.put("toves", new Integer(55));
    System.out.println(m);
    pi.set("m", PyUtil.toPyDictionary(m));
    pi.exec("print m");
    pi.exec("print m['slithy']");
  }
  public static void main(String args[]) {
    junit.textui.TestRunner.run(Test.class);
  }
} ///:~
```

下一小节我们会讲述（数据）提取工具的用法。

取出数据（Getting data out）

存在很多种不同的方法从 PythonInterpreter 提取数据。如果你只是调用 get( ) 方法，把对象标识符作为一个字符串传给它，它会返回一个 PyObject（由 org.python.core 所提供的支持类的一部分）。可以用__tojava__( )方法对它进行“cast”，但是还有比这更好的方法：

1. Py 类有一些很方便的方法，比如 py2int( )，可以接受一个 PyObjec 并且把它转换成若干不同的类型。

2. get( )有一个重载过的版本可以接受预期的 Java Class 对象作为第二个参数，并且产生出一个具有其运行时刻类型（run-time type）的对象（所以说在你的 Java 代码里你仍然需要对得到的结果进行一次 cast）。

使用第二种方法，从 PythonInterpreter 取出一个数组是非常简单的。这一点显得尤为有用，因为 Python 对字符串和文件处理都异常的出色，所以通常你会希望把结果作为一个字符串数组提出出来。例如，你可以使用 Python 的 glob( )函数对文件名进行通配符扩展（wildcard expansion），就像接下来的例子所展示的那样：

```
//- interpreter:PythonInterpreterGetting.java
// Getting data from the PythonInterpreter object.
package interpreter;
import org.python.util.PythonInterpreter;
import org.python.core.*;
import java.util.*;
import com.bruceeckel.python.*;
import junit.framework.*;

public class
PythonInterpreterGetting extends TestCase {
  PythonInterpreter interp =
    new PythonInterpreter();
  public void test() throws PyException  {
    interp.exec("a = 100");
    // If you just use the ordinary get(),
    // it returns a PyObject:
    PyObject a = interp.get("a");
    // There's not much you can do with a generic
    // PyObject, but you can print it out:
    System.out.println("a = " + a);
    // If you know the type it's supposed to be,
    // you can "cast" it using __tojava__() to
    // that Java type and manipulate it in Java.
    // To use 'a' as an int, you must use
    // the Integer wrapper class:
    int ai= ((Integer)a.__tojava__(Integer.class))
      .intValue();
    // There are also convenience functions:
    ai = Py.py2int(a);
    System.out.println("ai + 47 = " + (ai + 47));
    // You can convert it to different types:
    float af = Py.py2float(a);
```

```
System.out.println("af + 47 = " + (af + 47));
// If you try to cast it to an inappropriate
// type you'll get a runtime exception:
//! String as = (String)a.__tojava__(
//!   String.class);

// If you know the type, a more useful method
// is the overloaded get() that takes the
// desired class as the 2nd argument:
interp.exec("x = 1 + 2");
int x = ((Integer)interp
  .get("x", Integer.class)).intValue();
System.out.println("x = " + x);

// Since Python is so good at manipulating
// strings and files, you will often need to
// extract an array of Strings. Here, a file
// is read as a Python array:
interp.exec("lines = " +
  "open('PythonInterpreterGetting.java')" +
  ".readlines()");
// Pull it in as a Java array of String:
String[] lines = (String[])
  interp.get("lines", String[].class);
for(int i = 0; i < 10; i++)
  System.out.print(lines[i]);

// As an example of useful string tools,
// global expansion of ambiguous file names
// using glob is very useful, but it's not
// part of the standard Jython package, so
// you'll have to make sure that your
// Python path is set to include these, or
// that you deliver the necessary Python
// files with your application.
interp.exec("from glob import glob");
interp.exec("files = glob('*.java')");
String[] files = (String[])
```

```
    interp.get("files", String[].class);
  for(int i = 0; i < files.length; i++)
    System.out.println(files[i]);

  // You can extract tuples and arrays into
  // Java Lists with com.bruceeckel.PyUtil:
  interp.exec(
    "tup = ('fee', 'fi', 'fo', 'fum', 'fi')");
  List tup = PyUtil.toList(interp, "tup");
  System.out.println(tup);
  // It really is a list of String objects:
  System.out.println(tup.get(0).getClass());
  // You can easily convert it to a Set:
  Set tups = new HashSet(tup);
  System.out.println(tups);
  interp.exec("ints=[1,3,5,7,9,11,13,17,19]");
  List ints = PyUtil.toList(interp, "ints");
  System.out.println(ints);
  // It really is a List of Integer objects:
  System.out.println((ints.get(1)).getClass());

  // If you have a Python dictionary, it can
  // be extracted into a Java Map, again with
  // com.bruceeckel.PyUtil:
  interp.exec("dict = { 1 : 'a', 3 : 'b'," +
    "5 : 'c', 9 : 'd', 11 : 'e' }");
  Map map = PyUtil.toMap(interp, "dict");
  System.out.println("map: " + map);
  // It really is Java objects, not PyObjects:
  Iterator it = map.entrySet().iterator();
  Map.Entry e = (Map.Entry)it.next();
  System.out.println(e.getKey().getClass());
  System.out.println(e.getValue().getClass());
}
public static void
main(String[] args) throws PyException  {
  junit.textui.TestRunner.run(
    PythonInterpreterGetting.class);
```

```
  }
} ///:~
```

最后两个例子展示了从 Python 的垫片（tuples）和链表（lists）中提取数据到 Java Lists，以及从 Python 字典中提取数据到 Java Maps。上述两种情况都需要用到比标准 Jython 库所提供的更多的处理方法，所以我又写了一些小程序放在 com.bruceeckel.pyton 里。PyUtil: toList( )是用来从一个 Python 序列产生一个 List，toMap( )用来从 Python 字典产生出一个 Map。PyUtil 所提供的方法使得在 Java 和 Python 之间来回传递重要的数据结构变得更为简单。

多个解释器（Multiple interpreters）

很有必要再提一下，你可以在一个程序里声明多个 PythonInterpreter 对象，每个对象有它自己的名字空间：

```
//- interpreter:MultipleJythons.java
// You can run multiple interpreters, each
// with its own name space.
package interpreter;
import org.python.util.PythonInterpreter;
import org.python.core.*;
import junit.framework.*;

public class MultipleJythons extends TestCase  {
  PythonInterpreter
    interp1 =  new PythonInterpreter(),
    interp2 =  new PythonInterpreter();
  public void test() throws PyException {
    interp1.set("a", new PyInteger(42));
    interp2.set("a", new PyInteger(47));
    interp1.exec("print a");
    interp2.exec("print a");
    PyObject x1 = interp1.get("a");
    PyObject x2 = interp2.get("a");
    System.out.println("a from interp1: " + x1);
    System.out.println("a from interp2: " + x2);
  }
  public static void
  main(String[] args) throws PyException  {
```

```
    junit.textui.TestRunner.run(MultipleJythons.class);
  }
} ///:~
```

当运行程序的时候你会看到，每个 PythonInterpreter 所包含的 a 的值是不同的。

通过 Jython 控制 Java

既然有了 Java 语言可以使用，而且你还可以设置和取回解释器里面的值，使用上面这些方法（通过 Java 控制 Python）你已经可以完成非常多的功能。但是 Jython 还有一个让人称奇的地方，就是它可以从 Jython 内部几乎完全透明的访问 Java 类。基本上，一个 Java 类看起来就像是一个 Python 类。对于标准 Java 库里面的类，以及你自己写的类来说都是如此，就像你下面会看到的那样：

```
## interpreter:JavaClassInPython.py
#=M jython.bat JavaClassInPython.py
# Using Java classes within Jython
from java.util import Date, HashSet, HashMap
from interpreter.javaclass import JavaClass
from math import sin

d = Date() # Creating a Java Date object
print d # Calls toString()

# A "generator" to easily create data:
class ValGen:
  def __init__(self, maxVal):
    self.val = range(maxVal)
  # Called during 'for' iteration:
  def __getitem__(self, i):
    # Returns a tuple of two elements:
    return self.val[i], sin(self.val[i])

# Java standard containers:
map = HashMap()
set = HashSet()

for x, y in ValGen(10):
```

```
  map.put(x, y)
  set.add(y)
  set.add(y)

print map
print set

# Iterating through a set:
for z in set:
  print z, z.__class__

print map[3] # Uses Python dictionary indexing
for x in map.keySet(): # keySet() is a Map method
  print x, map[x]

# Using a Java class that you create yourself is
# just as easy:
jc = JavaClass()
jc2 = JavaClass("Created within Jython")
print jc2.getVal()
jc.setVal("Using a Java class is trivial")
print jc.getVal()
print jc.getChars()
jc.val = "Using bean properties"
print jc.val
##~
```

“=M”这个注释可以被 makefile 生成工具（我为本书而写的一个工具）识别为 makefile 命令的替代。我们会使用这种注释的方式而不使用通常情况下提取工具放在 makefile 里面的那些命令。

请注意，正如你所期望的那样，import 语句会映射到 Java 的 package 结构。第一个例子中，我们创建了一个 Date( )对象，就好像它原本就是一个 Python 类，打印这个对象只需要调用 toString( )就可以了。

ValGen 实现了“生成器（generator）”这个概念，“生成器”在 C++ STL（标准模板库，C++标准库的一部分）里面有着大量的应用。生成器是这么一个对象：每次调用它的“生成方法”的时候，它都会产生一个新的对象，使用它可以很方便的填充容

器。这里，我想在一个 for 循环里面使用它，所以我就需要让生成方法成为循环过程所调用的那个函数。这是一个被称作__getitem__( )的特殊方法，实际上它是索引操作符‘[ ]’的重载。每次 for 循环要循环到下一步的时候它就调用这个方法，当元素用完以后，__getitem__( )抛出一个越界异常，它意味着 for 循环的结束（在其它语言里，你从来也不会把异常用到一般的控制流程，但是在 Python 里它似乎工作的很好）。当 self.val[i]的元素耗尽的时候，这个异常会自动发生，所以__getitem__( )的代码其实很简单。唯一复杂的地方就是__getitem__( )看起来是返回两个而不是一个对象。Python 所做的就是自动地把多返回值打包进一个垫片（tuple），所以最终你仍然可以返回一个单一的对象（在 C++或者 Java 里你就必须得创建自己的数据结构来完成这个功能）。此外，在 for 循环里面用到 ValGen 的地方，Python 会自动对垫片进行“拆包”，所以你可以在 for 循环里使用多个迭代器。正是这些对于语法的简化才使 Python 显得如此惹人喜爱。

map 和 set 对象是 Java 的 HashMap 和 HashSet 的实例，我们又一次看到，创建这些对象就好像它们原本就是 Python 的一部分。在 for 循环内部，put( )和 add( )方法的工作方式跟它们在 Java 里一样。此外，对 Java Map 进行索引使用和字典（dictionary）一样的记号（notation），但是请注意，要遍历（iterate through）一个 Map 的键值（keys）你必须使用 Map 的方法 keyset( )，而不是 Python 字典的方法 keys( )。

上面例子的最后一部分展示了如何使用我从头开始创建的 Java 类，这么做是为了说明使用它是多么的寻常。另外请注意，Jython 天生就能够理解 JavaBeans 属性（properties），因为你既可以使用 getVal( )和 setVal( )方法，也可以给对应的 val 属性赋值或者读取它的值。此外，getChars( )在 Java 里会返回一个 Character[]，这在 Python 里就变成了一个数组。

在一个 Python 程序里使用你自己创建的 Java 类，最简单的方法是把它们放到一个 package 里。尽管 Jython 也可以引入（import）未打包的（unpackaged）java 类（import JavaClass），但是所有这些未打包的 java 类都会被当作定义于不同的 package 来对待，这样它们就只能看到彼此的公有（public）方法。

Java 包（packages）被翻译成 Python 的模块（modules） ，为了能够使用 Java 类，Python 必须引入一个模块。下面是 JavaClass 的 Java 代码：

```
//- interpreter:javaclass:JavaClass.java
package interpreter.javaclass;
import junit.framework.*;
import com.bruceeckel.util.*;

public class JavaClass {
  private String s = "";
```

```
  public JavaClass() {
    System.out.println("JavaClass()");
  }
  public JavaClass(String a) {
    s = a;
    System.out.println("JavaClass(String)");
  }
  public String getVal() {
    System.out.println("getVal()");
    return s;
  }
  public void setVal(String a) {
    System.out.println("setVal()");
    s = a;
  }
  public Character[] getChars() {
    System.out.println("getChars()");
    Character[] r = new Character[s.length()];
    for(int i = 0; i < s.length(); i++)
      r[i] = new Character(s.charAt(i));
    return r;
  }
  public static class Test extends TestCase  {
    JavaClass
      x1 = new JavaClass(),
      x2 = new JavaClass("UnitTest");
    public void test1() {
      System.out.println(x2.getVal());
      x1.setVal("SpamEggsSausageAndSpam");
      System.out.println(
        Arrays2.toString(x1.getChars()));
    }
  }
  public static void main(String[] args) {
    junit.textui.TestRunner.run(Test.class);
  }
} ///:~
```

你可以看到这只是一个普通的 Java 类，它根本不知道自己会被一个 Jython 程序使用。因此，Jython 的一个重要的应用就是用来测试 Java 代码[10]。因为 Python 是一门如此强大、灵活的动态语言，所以它是自动测试框架的一个理想工具，而且它不用对被测试的 Java 代码做任何改动。

内部类（Inner Classes）

内部类成为类对象的属性（Inner classes becomes attributes on the class object）. 静态的（static）内部类实例可以通过通常的调用来创建：

```
com.foo.JavaClass.StaticInnerClass()
```

非静态的内部类必须显式提供一个外部类（outer class）的实例作为第一个参数：

```
com.foo.JavaClass.InnerClass(com.foo.JavaClass())
```

使用 Java 库

Jython 包裹了（wrap）Java 库，所以所有的 Java 库都可以直接地或者通过继承被使用。此外，Python 简略的表达方式简化了代码的书写。

作为一个例子，请考虑《Thinking in Java》第二版第 9 章 HTMLButton.java 那个例子（你大概已经从 www.BruceEckel.com 下载并安装了那本书的配套源码，因为本书有好几个例子都用到了那本书的某些库）。下面把那个例子转换成了 Jython 代码：

```
## interpreter:PythonSwing.py
# The HTMLButton.java example from
# "Thinking in Java, 2nd edition," Chapter 13,
# converted into Jython.
# Don't run this as part of the automatic make:
#=M @echo skipping PythonSwing.py
from javax.swing import JFrame, JButton, JLabel
from java.awt import FlowLayout

frame = JFrame("HTMLButton", visible=1,
  defaultCloseOperation=JFrame.EXIT_ON_CLOSE)

def kapow(e):
```

[10] 把注册表里的 python.security.respectJavaAccessibility = true 改成 false 可以让测试功能更加强大，因为它允许测试脚本使用“*所有*”方法，即使是受保护的和包私有的（package-private）方法。

```
  frame.contentPane.add(JLabel("<html>"+
    "<i><font size=+4>Kapow!"))
  # Force a re-layout to
  # include the new label:
  frame.validate()

button = JButton("<html><b><font size=+2>" +
  "<center>Hello!<br><i>Press me now!",
  actionPerformed=kapow)
frame.contentPane.layout = FlowLayout()
frame.contentPane.add(button)
frame.pack()
frame.size=200, 500
##~
```

如果你把这个程序的 Java 版本与上面的 Jython 实现做一个比较，就会发现 Jython 版本更短而且通常更容易理解。比如，在 Java 版本里为了初始化（set up）一个 frame，你必须完成好几个调用：JFrame( )的构造函数，setVisible( )方法和 setDefaultCloseOperation( )方法。而在上面的代码里这三个操作通过一个构造函数调用就全部完成了。

另外请注意，JButton 是在它自己的构造函数里通过 actionListener( )方法配置的，把它赋给了 kapow。此外，Jython 对于 JavaBean 的感知意味着任何一个以“set”打头的方法调用都可以被替换为一个赋值操作，就像你上面看到的那样。

唯一并非来自 Java 的方法就是 pack( )方法，它似乎对于正确的布局是至关重要的。另外重要的一点是 pack( )的调用要出现在设置 size 之前。

继承 Java 库里的类（Inheriting from Java library classes）

在 Jython 里，你可以很容易地继承 Java 标准库里的类。下面是《Thinking in Java》第二版第 13 章 Dialogs.java 那个例子，转换成了 Jython 代码：

```
## interpreter:PythonDialogs.py
# Dialogs.java from "Thinking in Java, 2nd
# edition," Chapter 13, converted into Jython.
# Don't run this as part of the automatic make:
#=M @echo skipping PythonDialogs.py
from java.awt import FlowLayout
from javax.swing import JFrame, JDialog, JLabel
```

```
from javax.swing import JButton

class MyDialog(JDialog):
  def __init__(self, parent=None):
    JDialog.__init__(self,
      title="My dialog", modal=1)
    self.contentPane.layout = FlowLayout()
    self.contentPane.add(JLabel("A dialog!"))
    self.contentPane.add(JButton("OK",
      actionPerformed =
        lambda e, t=self: t.dispose()))
    self.pack()

frame = JFrame("Dialogs", visible=1,
  defaultCloseOperation=JFrame.EXIT_ON_CLOSE)
dlg = MyDialog()
frame.contentPane.add(
  JButton("Press here to get a Dialog Box",
    actionPerformed = lambda e: dlg.show()))
frame.pack()
##~
```

MyDialog 继承自 JDialog，你可以看到在调用基类构造函数的时候使用了命名参数（named arguments）。

在创建“OK”这个 JButton 的过程中，请注意，actionPerformed 方法是直接在构造函数里被设置的，这个函数是通过使用 Python 的 lamba 关键字来创建的。它创建一个没有名字的函数，把出现在冒号之前的部分作为参数，而把冒号之后的部分作为可以产生一个返回值的表达式。你需要知道的是，Java 里面 actionPerformed( )方法的原型只包含一个参数，但是 lambda 表达式显示出有两个参数。尽管如此，第二个参数是和一个默认值一起提供的，所以这个函数可以在只有一个参数的情况下被调用。使用第二个参数的理由可以通过它的默认值看出来，这是把 self 传入 lambda 表达式的一种方法，然后它可以被用来 dispose of the dialog。

把这段代码与《Thinkin in Java》第二版里的代码做一个比较，你会发现 Python 语言的特性使得更加简洁和直接的实现成为可能。

使用 Jython 创建 Java 类

尽管与本章最初的问题（创建一个解释器）没有直接的关系，但是 Jython 有一项额外的功能，它可以直接从你的 Jython 代码创建 Java 类。这可以产生非常有用的结果，因为这么一来你就可以把产生的东西当成原生的（native）Java 类来对待，尽管它在背后使用的是 Python 的魔力。

要从 Python 代码产生 Java 类，Jython 提供了一个叫做 jythonc 的编译器。

创建能够产生 Java 类的 Python 类，这个过程比通过 Python 调用 Java 类要复杂一些，因为 Java 类的方法是强类型的（strongly typed），而 Python 的函数和方法是弱类型的（weakly typed）。所以，你必须以某种方式告诉 jythonc 某个 Python 方法打算接受某一系列特定类型的参数以及他的返回值是哪种特定类型。你可以通过“@sig”字符串来达到这个目的，把这个字符串放在 Python 方法定义的后边（这是为了 Python 文档化而放置字符串的标准地方）。例如：

```
  def returnArray(self):
    "@sig public java.lang.String[] returnArray()"
```

Python 给出的定义并没有指定任何返回类型，但是@sig 字符串给出了关于正在传递和返回的所有类型信息。Jythonc 编译器使用这些信息来产生正确的 Java 代码。

要得到一个成功的编译，你必须还得遵循另外一系列的约束：你必须在你的 Python 类里继承自一个 Java 类或接口（你不需要为那些在父类/父接口里定义的方法指定@sig 签名）。如果你不这么做，你就得不到期望的方法——不幸的是，在这种情况下，jythonc 不会给你任何警告或者指出任何错误，但是你得到你所期望的。如果不知道少了些什么，可能是让人非常恼火的。

此外，你必须引入合适的 Java 类并且给出正确的 package 说明。在下面的例子里，引入了 java，所以你必须继承自 java.lang.Object，但是你也可以写成 from java.lang import Object 然后只是继承 Object 就可以了，不用给出 package 说明。很不幸，如果你写错了，并不会得到任何警告或者错误，所以你必须得有耐心并且不断尝试。

这里是一个用以产生 Java 类的 Python 类的例子。它还引入了针对 makefile 构建工具的“=T”指令，它指定一个不同于这个工具通常使用的目标文件（target）。本例中，Python 文件是用来构建一个 Java 的.class 文件，所以 class 文件才是期望的 makefile 目标文件。为了达到这个目的，默认的 makefile 命令被“=M”指令替代了（请注意你是如何使用“\”拆行的）。

```
## interpreter:PythonToJavaClass.py
#=T python\java\test\PythonToJavaClass.class
#=M jythonc.bat --package python.java.test \
```

```
#=M PythonToJavaClass.py
# A Python class created to produce a Java class
from jarray import array
import java

class PythonToJavaClass(java.lang.Object):
  # The '@sig' signature string is used to create
  # the proper signature in the resulting
  # Java code:
  def __init__(self):
    "@sig public PythonToJavaClass()"
    print "Constructor for PythonToJavaClass"

  def simple(self):
    "@sig public void simple()"
    print "simple()"

  # Returning values to Java:
  def returnString(self):
    "@sig public java.lang.String returnString()"
    return "howdy"

  # You must construct arrays to return along
  # with the type of the array:
  def returnArray(self):
    "@sig public java.lang.String[] returnArray()"
    test = [ "fee", "fi", "fo", "fum" ]
    return array(test, java.lang.String)

  def ints(self):
    "@sig public java.lang.Integer[] ints()"
    test = [ 1, 3, 5, 7, 11, 13, 17, 19, 23 ]
    return array(test, java.lang.Integer)

  def doubles(self):
    "@sig public java.lang.Double[] doubles()"
    test = [ 1, 3, 5, 7, 11, 13, 17, 19, 23 ]
    return array(test, java.lang.Double)
```

```
  # Passing arguments in from Java:
  def argIn1(self, a):
    "@sig public void argIn1(java.lang.String a)"
    print "a: %s" % a
    print "a.__class__", a.__class__

  def argIn2(self, a):
    "@sig public void argIn1(java.lang.Integer a)"
    print "a + 100: %d" % (a + 100)
    print "a.__class__", a.__class__

  def argIn3(self, a):
    "@sig public void argIn3(java.util.List a)"
    print "received List:", a, a.__class__
    print "element type:", a[0].__class__
    print "a[3] + a[5]:", a[5] + a[7]
    #! print "a[2:5]:", a[2:5] # Doesn't work

  def argIn4(self, a):
    "@sig public void \
       argIn4(org.python.core.PyArray a)"
    print "received type:", a.__class__
    print "a: ", a
    print "element type:", a[0].__class__
    print "a[3] + a[5]:", a[5] + a[7]
    print "a[2:5]:", a[2:5] # A real Python array

  # A map must be passed in as a PyDictionary:
  def argIn5(self, m):
    "@sig public void \
       argIn5(org.python.core.PyDictionary m)"
    print "received Map: ", m, m.__class__
    print "m['3']:", m['3']
    for x in m.keys():
      print x, m[x]
##~
```

第一点请注意，PythonToJavaClass 是从 java.lang.Object 继承而来的；如果你不这么做就会不知不觉地得到一个缺少正确签名的 Java 类。没有要求说必须要你继承自 Object；其它任何 Java 类都可以。

设计这个类是为了示范不同的参数和返回值，是为了给你提供足够多的例子以使你能够很容易地创建你自己的签名字符串。前三个基本上是不言自明的，但是请注意签名字符串中 Java 限定词（qualification）用的都是全名。

在 returnArray( )里，一个 Python 数组必须作为一个 Java 数组来返回。为此目的，必须使用 Jython 的 array( )函数（来自 jarry 模块），另外还得需要最终数组所持有类的类型。任何时候如果你需要为 Java 返回一个数组，你必须使用 array( )，就像你在 ints( )和 doubles( )方法里看到的那样。

最后一个方法展示了如何从 Java 传入参数。对于基本类型，只要你在@sig 字符串里指定它们，它们就可以自动传入，但是你必须使用对象，而且你不能传入基原类型（primitves）（也就是说，基原类型必须放置在外覆对象内，比如 Integer）。

在 argIn3( )里，你会看到一个 Java List 被透明地转化成行为类似于 Python 数组（array）的东西，但是它不是一个真正的数组，因为你不能从它得到一个切片（slice）。如果你想得到一个真正的 Python 数组，那么你必须像在 argIn4( )里那样创建并且传入一个 PyArray，这样切片操作才会成功。类似地，一个 Java Map 必须作为一个 PyDictionary 传入才能被作为 Python 字典对待。

下面的 Java 程序用以演练由上述 Python 代码生成的 Java 类。另外，它针对 makefile 构建工具引入了“=D”指令，用以指定除了该工具自己侦测出的依赖关系（dependency）。这里，直到 PythonToJavaClass.class 可用，否则你不能够编译 TestPythonToJavaClass.java。

```
//- interpreter:TestPythonToJavaClass.java
//+D python\java\test\PythonToJavaClass.class
package interpreter;
import java.lang.reflect.*;
import java.util.*;
import org.python.core.*;
import junit.framework.*;
import com.bruceeckel.util.*;
import com.bruceeckel.python.*;
// The package with the Python-generated classes:
import python.java.test.*;

public class
```

```
TestPythonToJavaClass extends TestCase  {
  PythonToJavaClass p2j = new PythonToJavaClass();
  public void testDumpClassInfo() {
    System.out.println(
      Arrays2.toString(
        p2j.getClass().getConstructors()));
    Method[] methods =
      p2j.getClass().getMethods();
    for(int i = 0; i < methods.length; i++) {
      String nm = methods[i].toString();
      if(nm.indexOf("PythonToJavaClass") != -1)
        System.out.println(nm);
    }
  }
  public void test1() {
    p2j.simple();
    System.out.println(p2j.returnString());
    System.out.println(
      Arrays2.toString(p2j.returnArray()));
    System.out.println(
      Arrays2.toString(p2j.ints());
    System.out.println(
      Arrays2.toString(p2j.doubles()));
    p2j.argIn1("Testing argIn1()");
    p2j.argIn2(new Integer(47));
    ArrayList a = new ArrayList();
    for(int i = 0; i < 10; i++)
      a.add(new Integer(i));
    p2j.argIn3(a);
    p2j.argIn4(
      new PyArray(Integer.class, a.toArray()));
    Map m = new HashMap();
    for(int i = 0; i < 10; i++)
      m.put("" + i, new Float(i));
    p2j.argIn5(PyUtil.toPyDictionary(m));
  }
  public static void main(String[] args) {
    junit.textui.TestRunner.run(
```

```
      TestPythonToJavaClass.class);
  }
} ///:~
```

要添加 Python 支持，你通常只需要引入 org.python.core 里的类。上面例子里的其它东西是非常一目了然的，从 Java 的立场来看，PythonToJavaClass 只是作为另外一个 Java 类出现。DumpClassInfo( )使用反射（reflection）来确认在 PythonToJavaClass.py 里指定的方法签名依然正确。

编译来自 Python 代码的 Java 类

从 Python 代码创建 Java 类的一部分诀窍（trick）是位于方法文档化字符串里的 @sig 信息。但是还有第二个问题，这个问题的起因是 Python 没有“package”关键字——Python 里与 packages 等价的东西（modules）是根据文件名隐式（implicitly）创建的。但是，为了把最终的类文件引入 Java 程序，必须要传给 jythonc 如何为 Python 代码创建 Java package 的相关信息。通过 jythonc 命令行使用——package 标识，紧跟着一个你想要产生的 package 的名称（包括用于分割的点号，就像你在 Java 程序里使用 package 关键字给 package 命名一样）。这样就可以把最终的.class 文件放入相对于当前目录的相应的子目录。然后你只需要在你的 Java 程序里引入这个 package 就可以了，如上所示（要从代码所在目录运行程序，你需要在你的 CLASSPATH 里设置“.”。）

下面是我用以编译上面例子的 make 命令的依赖关系（make 命令知道行尾的斜杠符号是续行符）。使用我的 makefile 构建工具能够理解的注释语法，这些依赖关系被编码进了上述的 Java 和 Python 文件。

```
TestPythonToJavaClass.class: \
        TestPythonToJavaClass.java \
        python\java\test\PythonToJavaClass.class
    javac TestPythonToJavaClass.java

python\java\test\PythonToJavaClass.class: \
        PythonToJavaClass.py
    jythonc.bat --package python.java.test \
    PythonToJavaClass.py
```

第一个目标文件，TestPythonToJavaClass.class，依赖于 TestPythonToJavaClass.java 和 PythonToJavaClass.class，PythonToJavaClass.class 是由 Python 代码转化而来的 class 文件。它又依赖于自己

的 Python 源码。请注意，很重要的一点是指定目标文件所在的目录，这样 makefile 才能以最少的重新编译次数创建 Java 程序。

Java-Python 扩展（JPE）

Java-Python Extension (JPE)是 Jython 的一个替代方案，它直接连接原生的（native）C-Python 实现。

Jython 完全运行于 Java 虚拟机（VM）之内，这会导致两个根本的限制：Jython 不能被 CPython 所调用，而且原生的 Python 扩展也不能被 Jython 访问。JPE 链接的是 Python 的 C 语言库，所以 JPE 可以被 C-Python 所调用，而且原生的 Python 扩展可以被 Java 通过 JPE 调用。

如果你需要访问你的原生平台（native platform）的某些功能，JPE 可能是更简单的解决方案。你可以在 http://www.arakne.com/jpe.htm 找到 JPE。

总结

本章以有争议的方式深入讲解了 Jython，与使用 interpreter 设计模式所要求的比起来要深入的多。实际上，如果你决定需要使用 interpreter 并且不想因为发明你自己的一门语言而迷失方向，安装 Jython 是想一个相当简单的解决方案，至少你可以仿照 GreenHouseController 那个例子慢慢上手。

当然，那个例子经常是过于简单了，你可能需要更复杂的一些东西，常常需要传入传出一些更有意思的数据。当碰到文档不够用的情况的时候，我觉得有必要更全面地检查一下 Jython。

在这个过程中，可能潜藏着另外一个同样有效的设计模式，大概可以把它叫做多个编程语言（multiple languages）。这是基于以下经验，让每种语言解决它（相对于其它语言）更擅长的某一类问题；把多种语言结合起来你可以比单独使用其中一种语言更快地解决问题。CORBA 是跨越语言边界的另外一种方法，同时它也可以跨越计算机和操作系统。

对我来说，Python 和 Java 为程序开发展现了一个非常强有力的组合，这是基于 Java 的架构和工具集，以及 Python 超强的快速开发能力（一般认为比 C++或者 Java 快 5 到 10 倍）。Python 通常比较慢，但是，即使你最后要为了速度重写某部分代码，最初的快速开发也可以让你更快地构建出系统，然后找到并解决问题的关键部分。Python 的执行速度通常并不是一个问题——这时候所得到的好处就更大了。有许多商业产品已经使用了 Java 和 Jython，基于它们在生产效率方面的巨大优势，我期望在未来它们有更多的应用。

练习

1. 改写 GreenHouseLanguage.py，让它检测事件的时间并且在合适的时间运行这些事件。

2. 改写 GreenHouseLanguage.py，让它为 action 调用一个函数而不是仅仅打印一个字符串。

3. 写一个 Swing 程序，使用 JTextField（让用户输入命令）和 JTextArea（显示命令执行结果）。连接到一个 PythonInterpreter 对象，把输出结果送到 JTextArea（应该可以滚屏）。你得需要找出把输出重定向到 Java stream 的 PythonInterpreter 命令。

4. 改写 GreenHouselanguage.py，不采用 Event 内部的静态数组，添加一个主控制器类（master contrller class），并且为每一个子类提供一个 run( )方法。在执行的时候，每个 run( )方法应该创建并且使用一个来自标准 Java 库的对象。改写 GreenHouseController.java 并且使用这个新类。

5. 改写练习 2 的 GreenHouseLanguage.py，让它产生 Java 类（加入@sig 文档化字符串用以产生正确的 Java 签名，创建一个 makefile 用来编译成 Java 的.class 文件）。写一个 Java 程序使用这些类。

复杂系统的状态（Complex system states）

状态机（StateMachine）

就像 State 模式可以通过某种方法让客户端程序员改变（类的）实现一样，状态机（StateMachine）利用某种结构自动的使类的实现从一个对象改变为另外一个对象。正在使用的实现代表系统的当前状态，系统在某一种状态和下一个状态之间表现出不同的行为（因为使用了 State 模式）。基本上说，这就是一个使用了对象的“状态机”。

下面你将看到的一个基本状态机的框架，使整个系统从一个状态迁移到下一个状态的代码通常都是符合模板方法模式(Template Method)的。一开始，我们先定义一个标记接口（tagging interface）用来输入对象。

```
//: statemachine:Input.java
// Inputs to a state machine
package statemachine;

public interface Input {} ///:~
```

每个状态都通过 run( )函数来完成它的行为，而且（在这个设计里）你可以传给它一个输入对象，这样它就能根据这个输入对象告诉它会迁移到哪个新的状态。这个设计和下一小节的设计的主要区别是：对于这个设计，每个 State 对象根据输入对象决定它自己可以迁移到哪些其它状态；下面那个设计，所有状态迁移都包含在一张单独的表里。换一种说法就是：这个设计里，每个 State 对象都拥有一张自己的小型状态表，而下面那个设计里整个系统只有一张状态迁移的总表。

```
//: statemachine:State.java
// A State has an operation, and can be moved
// into the next State given an Input:
package statemachine;

public interface State {
  void run();
  State next(Input i);
} ///:~
```

StateMachine 会记录当前状态，这个状态是在 StateMachine 的构造函数里初始化的。RunAll( )方法接受一个关于一系列输入对象的迭代器（这里使用迭代器是为了图

个简单方便，重要的是输入信息是从别的地方传进来的）。这个方法不但使状态机迁移到下一个状态，而且它还调用每一个状态对象的 run( )方法——你会发现它是 State 模式的扩展，因为 run( )根据系统所处的不同状态做出不同的事情。

```
//: statemachine:StateMachine.java
// Takes an Iterator of Inputs to move from State
// to State using a template method.
package statemachine;
import java.util.*;

public class StateMachine {
  private State currentState;
  public StateMachine(State initialState) {
    currentState = initialState;
    currentState.run();
  }
  // Template method:
  public final void runAll(Iterator inputs) {
    while(inputs.hasNext()) {
      Input i = (Input)inputs.next();
      System.out.println(i);
      currentState = currentState.next(i);
      currentState.run();
    }
  }
} ///:~
```

我把 runAll()当作一个模板方法来实现。这是一种典型的做法，当然并不是说必须得这么做——你可能想要 concievably 重载它，但是通常这些行为的改变还是会发生在 State 对象的 run( )方法里。

到这里为止，这种风格的状态机（每个状态决定它的下一个状态）的基本框架就算完成了。我会写一个假想的捕鼠器（mousetrap）作为例子，这个捕鼠器在诱捕老鼠的过程中会经历几个不同的状态[11]。Mouse 类和与之相关的信息都存储在 mouse package 里，包括一个用来代表老鼠所有可能动作的类，这些动作会作为状态机的输入。

[11] 创建这个例子的过程种不会有老鼠受到伤害。

```
//: statemachine:mouse:MouseAction.java

package statemachine.mouse;
import java.util.*;
import statemachine.*;

public class MouseAction implements Input {
  private String action;
  private static List instances = new ArrayList();
  private MouseAction(String a) {
    action = a;
    instances.add(this);
  }
  public String toString() { return action; }
  public int hashCode() {
    return action.hashCode();
  }
  public boolean equals(Object o) {
    return (o instanceof MouseAction)
      && action.equals(((MouseAction)o).action);
  }
  public static MouseAction forString(String description) {
    Iterator it = instances.iterator();
    while(it.hasNext()) {
      MouseAction ma = (MouseAction)it.next();
      if(ma.action.equals(description))
        return ma;
    }
    throw new RuntimeException("not found: " + description);
  }
  public static MouseAction
    appears = new MouseAction("mouse appears"),
    runsAway = new MouseAction("mouse runs away"),
    enters = new MouseAction("mouse enters trap"),
    escapes = new MouseAction("mouse escapes"),
    trapped = new MouseAction("mouse trapped"),
    removed = new MouseAction("mouse removed");
} ///:~
```

你会注意到，hashCode( )和 equals( )都被重载了，这么做是为了 MouseAction 对象能够作为 HashMap 的键值来使用，但是在 mousetrap 的第一个版本里我们先不这么做。另外，老鼠所有可能的动作都作为一个静态的 MouseAction 对象被枚举出来。

为了便于书写测试代码，一系列有关老鼠（动作）的输入是作为一个文本文件提供的。

```
//:! statemachine:mouse:MouseMoves.txt
mouse appears
mouse runs away
mouse appears
mouse enters trap
mouse escapes
mouse appears
mouse enters trap
mouse trapped
mouse removed
mouse appears
mouse runs away
mouse appears
mouse enters trap
mouse trapped
mouse removed
///:~
```

为了用通用的（generic fashion）方法来读这个文件，我写了一个通用的 StringList 工具：

```
//: com:bruceeckel:util:StringList.java
// General-purpose tool that reads a file of text
// lines into a List, one line per list.
package com.bruceeckel.util;
import java.io.*;
import java.util.*;

public class StringList extends ArrayList {
  public StringList(String textFilePath) {
    try {
      BufferedReader inputs = new BufferedReader (
        new FileReader(textFilePath));
```

```
      String line;
      while((line = inputs.readLine()) != null)
        add(line.trim());
    } catch(IOException e) {
      throw new RuntimeException(e);
    }
  }
} ///:~
```

这个 StringList 只能像 ArrayList 那样存储对象，所以我们还需要一个适配器（adapter）用来把 String 对象转换成 MouseAction 对象：

```
//: statemachine:mouse:MouseMoveList.java
// A "transformer" to produce a
// List of MouseAction objects.
package statemachine.mouse;
import java.util.*;
import com.bruceeckel.util.*;

public class MouseMoveList extends ArrayList {
  public MouseMoveList(Iterator it) {
    while(it.hasNext())
      add(MouseAction.forString((String)it.next()));
  }
} ///:~
```

MouseMoveList 看起来有点像一个装饰者（decorator），而实际上干的又有点像 adapter 干的活。不管到底是什么，adapter 把一个接口改变为另外一个接口，decorator（给现有类）添加功能或者数据。MouseMoveList 改变了容器的内容，所以可能叫做 Transformer 更合适。

有了上面这些工具，现在就可以创建第一个版本的 mousetrap 了。每一个由 State 派生的类都定义它自己的 run( )行为，然后用 if-else 语句确定它的下一个状态。

```
//: statemachine:mousetrap1:MouseTrapTest.java
// State Machine pattern using 'if' statements
// to determine the next state.
package statemachine.mousetrap1;
import statemachine.mouse.*;
```

```
import statemachine.*;
import com.bruceeckel.util.*;
import java.util.*;
import java.io.*;
import junit.framework.*;

// A different subclass for each state:

class Waiting implements State {
  public void run() {
    System.out.println(
      "Waiting: Broadcasting cheese smell");
  }
  public State next(Input i) {
    MouseAction ma = (MouseAction)i;
    if(ma.equals(MouseAction.appears))
      return MouseTrap.luring;
    return MouseTrap.waiting;
  }
}

class Luring implements State {
  public void run() {
    System.out.println(
      "Luring: Presenting Cheese, door open");
  }
  public State next(Input i) {
    MouseAction ma = (MouseAction)i;
    if(ma.equals(MouseAction.runsAway))
      return MouseTrap.waiting;
    if(ma.equals(MouseAction.enters))
      return MouseTrap.trapping;
    return MouseTrap.luring;
  }
}

class Trapping implements State {
  public void run() {
```

```
    System.out.println("Trapping: Closing door");
  }
  public State next(Input i) {
    MouseAction ma = (MouseAction)i;
    if(ma.equals(MouseAction.escapes))
      return MouseTrap.waiting;
    if(ma.equals(MouseAction.trapped))
      return MouseTrap.holding;
    return MouseTrap.trapping;
  }
}

class Holding implements State {
  public void run() {
    System.out.println("Holding: Mouse caught");
  }
  public State next(Input i) {
    MouseAction ma = (MouseAction)i;
    if(ma.equals(MouseAction.removed))
      return MouseTrap.waiting;
    return MouseTrap.holding;
  }
}

class MouseTrap extends StateMachine {
  public static State
    waiting = new Waiting(),
    luring = new Luring(),
    trapping = new Trapping(),
    holding = new Holding();
  public MouseTrap() {
    super(waiting); // Initial state
  }
}

public class MouseTrapTest extends TestCase  {
  MouseTrap trap = new MouseTrap();
  MouseMoveList moves =
```

```
    new MouseMoveList(
      new StringList("../mouse/MouseMoves.txt")
        .iterator());
  public void test() {
    trap.runAll(moves.iterator());
  }
  public static void main(String args[]) {
    junit.textui.TestRunner.run(MouseTrapTest.class);
  }
} ///:~
```

StateMachine 类简单地把所有可能的状态都定义成静态对象，然后设置自己的初始状态。UnitTest 创建一个 MouseTrap 对象，然后用 MouseMoveList（所存储的 MouseMove）作为输入来测试它。

虽然在 next( )方法内部使用 if-else 语句是无可厚非的，但是如果状态数目非常多，用这种方法就会变的很麻烦。另一种方法是在每个 State 对象内部创建一个表结构，根据输入信息定义不同的次状态（next state）。

刚开始，这看起来似乎应该很简单。你需要在每一个 State 子类里定义一个静态表结构，而这个表又要根据其它 State 对象来确定它们之间该如何迁移。但是，这种方法在初始化的时候会产生循环依赖。为了解决这个问题，我必须得把表结构的初始化延迟到某个特定的 State 对象的 next( )方法第一次被调用的时候。由于这个原因，next( )方法初看起来可能会显得有点怪。

StateT 类是 State 接口的一个实现（这样它就能和上面那个例子共用同一个 StateMachine 类），此外它还添加了一个 Map 和一个用来从两维数组初始化这个 map 的方法。Next( )方法在基类里有一个实现，派生类重载过的 next( )方法首先测试 Map 是否为空（如果为空，则初始化之），然后它会调用基类的 next( )方法。

```
//: statemachine:mousetrap2:MouseTrap2Test.java
// A better mousetrap using tables
package statemachine.mousetrap2;
import statemachine.mouse.*;
import statemachine.*;
import java.util.*;
import java.io.*;
import com.bruceeckel.util.*;
import junit.framework.*;
```

```
abstract class StateT implements State {
  protected Map transitions = null;
  protected void init(Object[][] states) {
    transitions = new HashMap();
    for(int i = 0; i < states.length; i++)
      transitions.put(states[i][0], states[i][1]);
  }
  public abstract void run();
  public State next(Input i) {
    if(transitions.containsKey(i))
      return (State)transitions.get(i);
    else
      throw new RuntimeException(
        "Input not supported for current state");
  }
}

class MouseTrap extends StateMachine {
  public static State
    waiting = new Waiting(),
    luring = new Luring(),
    trapping = new Trapping(),
    holding = new Holding();
  public MouseTrap() {
    super(waiting); // Initial state
  }
}

class Waiting extends StateT {
  public void run() {
    System.out.println(
      "Waiting: Broadcasting cheese smell");
  }
  public State next(Input i) {
    // Delayed initialization:
    if(transitions == null)
      init(new Object[][] {
        { MouseAction.appears, MouseTrap.luring },
```

```
      });
    return super.next(i);
  }
}

class Luring extends StateT {
  public void run() {
    System.out.println(
      "Luring: Presenting Cheese, door open");
  }
  public State next(Input i) {
    if(transitions == null)
      init(new Object[][] {
        { MouseAction.enters, MouseTrap.trapping },
        { MouseAction.runsAway, MouseTrap.waiting },
      });
    return super.next(i);
  }
}

class Trapping extends StateT {
  public void run() {
    System.out.println("Trapping: Closing door");
  }
  public State next(Input i) {
    if(transitions == null)
      init(new Object[][] {
        { MouseAction.escapes, MouseTrap.waiting },
        { MouseAction.trapped, MouseTrap.holding },
      });
    return super.next(i);
  }
}

class Holding extends StateT {
  public void run() {
    System.out.println("Holding: Mouse caught");
  }
```

```
  public State next(Input i) {
    if(transitions == null)
      init(new Object[][] {
        { MouseAction.removed, MouseTrap.waiting },
      });
    return super.next(i);
  }
}

public class MouseTrap2Test extends TestCase  {
  MouseTrap trap = new MouseTrap();
  MouseMoveList moves =
    new MouseMoveList(
      new StringList("../mouse/MouseMoves.txt")
        .iterator());
  public void test() {
    trap.runAll(moves.iterator());
  }
  public static void main(String args[]) {
    junit.textui.TestRunner.run(MouseTrap2Test.class);
  }
} ///:~
```

代码的后面一部分和上面那个例子是完全相同的——不同的部分在 next( )方法和 StateT 类里。

如果你必须得创建并且维护非常多的 State 类，这种方法（相对于上一个例子）有所改善，因为通过查看表结构可以更容易和迅速的读懂各个状态之间的迁移关系。

练习

1. 改写 MouseTRap2Test.java，从一个只包含状态表信息的外部的文本文件加载状态表信息。

表驱动的状态机（Table-Driven State Machine）

上面那个设计的好处是所有有关状态的信息，包括状态迁移信息，都位于 state 类的内部。通常来说，这是一个好的设计习惯。

但是，对于一个纯粹的状态机来说，它完全可以用一个单独的状态迁移表来表示。这么做的好处是所有有关状态机的信息都可以放在同一个地方，也就意味着你可以简单的根据一个传统的状态迁移图来创建和维护它。

传统的状态迁移图用圆圈来代表每各个状态，用不同的细线来指向某一状态可以迁移到的所有其它状态。每条迁移线都标注有迁移条件和动作。下面是一张状态图：

(一张简单的状态机图)

目标：

- 直接翻译状态图
- 一系列的变化：状态图表示
- 合理的实现
- 没有过剩的状态（你可以用新的状态来表示每个单独的变化）
- 简单灵活

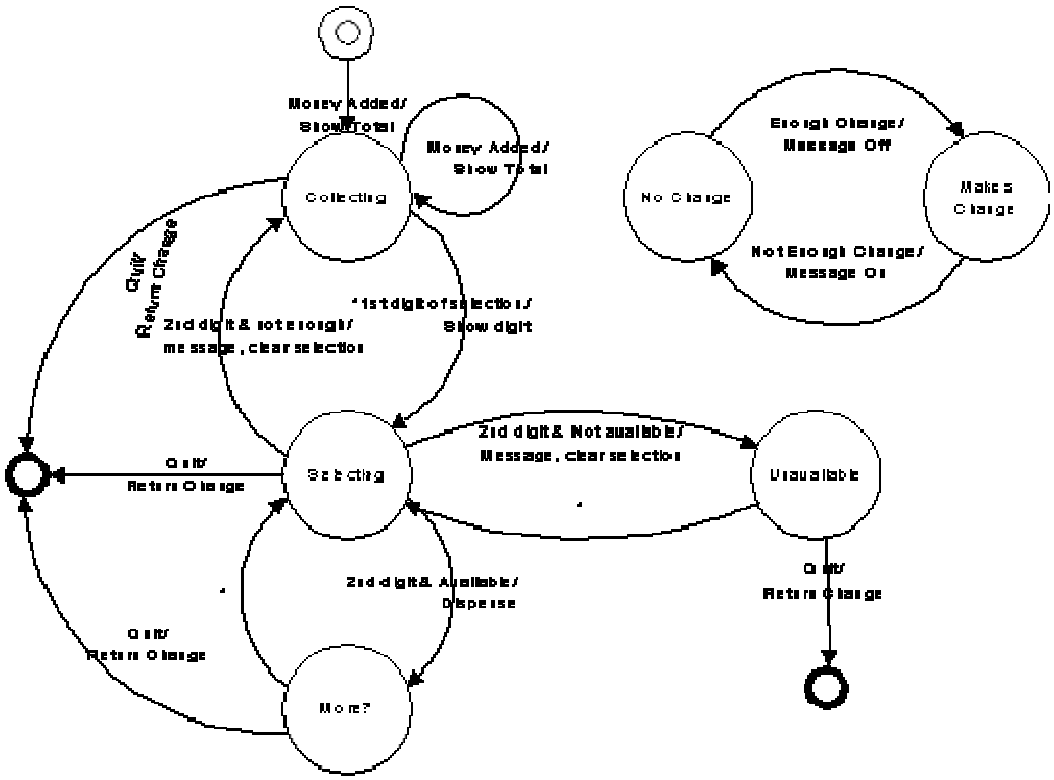
观察结果：

- 状态不是最重要的——它们不包含信息或者函数/数据，只是一个标识
- 跟 State 模式没有相似性
- 状态机控制各个状态之间的迁移
- 和享元（flyweight）有些类似
- 每个状态都可能迁移到多个其它状态
- 迁移条件和动作必须位于状态的外部
- 为了使配置方便，集中描述所有状态变化，把它放到一张单独的表里面。

例子：

- 状态机和用于表驱动的代码
- 实现一个自动售货机
- 用到其它的几个模式

- 把通用的状态机代码和特定的程序代码相分离（像 template method 模式那样）
- 为每个输入搜索合适的解决方案（像职责链模式那样）
- 测试和状态迁移都封装在函数对象里（封装函数的对象）
- Java 约束：methods are not first-class objects



State 类

这个 State 类和上面的完全不同，它只是个有名字的占位符（placeholder）。所以它就没有继承上一个 State 类。

```
//: statemachine2:State.java
package statemachine2;

public class State {
  private String name;
  public State(String nm) { name = nm; }
  public String toString() { return name; }
} ///:~
```

迁移条件（Conditions for transition）

在状态迁移图上，会针对输入检验它是否满足迁移条件，进而决定它能否迁移到对应的次状态。跟前面的例子一样，这里 Input 类还是一个标记接口(tagging interface)。

```
//: statemachine2:Input.java
// Inputs to a state machine
package statemachine2;

public interface Input {} ///:~
```

Condition 类通过对输入的 Input 对象求值（evaluates）来决定表中这一行是否是符合条件的迁移。

```
//: statemachine2:Condition.java
// Condition function object for state machine
package statemachine2;

public interface Condition {
  boolean condition(Input i);
} ///:~
```

迁移动作（Transition actions）

如果 Condition 类返回 true，（当前状态）就会迁移到新的状态，在这个迁移的过程中会发生一些动作（在前面那个状态机的设计里，这个动作就是 run()方法）：

```
//: statemachine2:Transition.java
// Transition function object for state machine
package statemachine2;

public interface Transition {
  void transition(Input i);
} ///:~
```

表结构（The table）

有了前面这些类，我们就可以建立一个三维的表结构，每一行正好描述一种状态。第一行代表当前状态，其余的那些行包括输入类型，必须满足的状态迁移条件，

迁移过程中发生的动作，以及要迁移到的次状态。请注意 Input 对象并不仅仅代表输入，它本身也是个 Messenger 对象，它会传递信息给 Condition 和 Transition 对象。

```
{ {CurrentState},
  {Input, Condition(Input), Transition(Input), Next},
  {Input, Condition(Input), Transition(Input), Next},
  {Input, Condition(Input), Transition(Input), Next},
   ...
}
```

基本状态机（The basic machine）

```
//: statemachine2:StateMachine.java
// A table-driven state machine

package statemachine2;
import java.util.*;

public class StateMachine {
  private State state;
  private Map map = new HashMap();
  public StateMachine(State initial) {
    state = initial;
  }
  public void buildTable(Object[][][] table) {
    for(int i = 0; i < table.length; i++) {
      Object[][] row = table[i];
      Object currentState = row[0][0];
      List transitions = new ArrayList();
      for(int j = 1; j < row.length; j++)
        transitions.add(row[j]);
      map.put(currentState, transitions);
    }
  }
  public void nextState(Input input) {
    Iterator it=((List)map.get(state)).iterator();
    while(it.hasNext()) {
      Object[] tran = (Object[])it.next();
```

```
      if(input == tran[0] ||
         input.getClass() == tran[0]) {
        if(tran[1] != null) {
          Condition c = (Condition)tran[1];
          if(!c.condition(input))
            continue; // Failed test
        }
        if(tran[2] != null)
          ((Transition)tran[2]).transition(input);
        state = (State)tran[3];
        return;
      }
    }
    throw new RuntimeException(
      "Input not supported for current state");
  }
} ///:~
```

简单的自动售货机（Simple vending machine）

```
//: statemachine:vendingmachine:VendingMachine.java
// Demonstrates use of StateMachine.java

package statemachine.vendingmachine;
import statemachine2.*;

final class VM extends State {
  private VM(String nm) { super(nm); }
  public final static VM
    quiescent = new VM("Quiesecent"),
    collecting = new VM("Collecting"),
    selecting = new VM("Selecting"),
    unavailable = new VM("Unavailable"),
    wantMore = new VM("Want More?"),
    noChange = new VM("Use Exact Change Only"),
    makesChange = new VM("Machine makes change");
}
```

```
final class HasChange implements Input {
  private String name;
  private HasChange(String nm) { name = nm; }
  public String toString() { return name; }
  public final static HasChange
    yes = new HasChange("Has change"),
    no = new HasChange("Cannot make change");
}

class ChangeAvailable extends StateMachine {
  public ChangeAvailable() {
    super(VM.makesChange);
    buildTable(new Object[][][]{
      { {VM.makesChange}, // Current state
        // Input, test, transition, next state:
        {HasChange.no, null, null, VM.noChange}},
      { {VM.noChange}, // Current state
        // Input, test, transition, next state:
        {HasChange.yes, null,
         null, VM.makesChange}},
    });
  }
}

final class Money implements Input {
  private String name;
  private int value;
  private Money(String nm, int val) {
    name = nm;
    value = val;
  }
  public String toString() { return name; }
  public int getValue() { return value; }
  public final static Money
    quarter = new Money("Quarter", 25),
    dollar = new Money("Dollar", 100);
}
```

```
final class Quit implements Input {
  private Quit() {}
  public String toString() { return "Quit"; }
  public final static Quit quit = new Quit();
}

final class FirstDigit implements Input {
  private String name;
  private int value;
  private FirstDigit(String nm, int val) {
    name = nm;
    value = val;
  }
  public String toString() { return name; }
  public int getValue() { return value; }
  public final static FirstDigit
    A = new FirstDigit("A", 0),
    B = new FirstDigit("B", 1),
    C = new FirstDigit("C", 2),
    D = new FirstDigit("D", 3);
}

final class SecondDigit implements Input {
  private String name;
  private int value;
  private SecondDigit(String nm, int val) {
    name = nm;
    value = val;
  }
  public String toString() { return name; }
  public int getValue() { return value; }
  public final static SecondDigit
    one = new SecondDigit("one", 0),
    two = new SecondDigit("two", 1),
    three = new SecondDigit("three", 2),
    four = new SecondDigit("four", 3);
}
```

```
class ItemSlot {
  int price;
  int quantity;
  static int counter = 0;
  String id = Integer.toString(counter++);
  public ItemSlot(int prc, int quant) {
    price = prc;
    quantity = quant;
  }
  public String toString() { return id; }
  public int getPrice() { return price; }
  public int getQuantity() { return quantity; }
  public void decrQuantity() { quantity--; }
}

public class VendingMachine extends StateMachine{
  StateMachine changeAvailable =
    new ChangeAvailable();
  int amount = 0;
  FirstDigit first = null;
  ItemSlot[][] items = new ItemSlot[4][4];
  Condition notEnough = new Condition() {
    public boolean condition(Input input) {
      int i1 = first.getValue();
      int i2 = ((SecondDigit)input).getValue();
      return items[i1][i2].getPrice() > amount;
    }
  };
  Condition itemAvailable = new Condition() {
    public boolean condition(Input input) {
      int i1 = first.getValue();
      int i2 = ((SecondDigit)input).getValue();
      return items[i1][i2].getQuantity() > 0;
    }
  };
  Condition itemNotAvailable = new Condition() {
    public boolean condition(Input input) {
      return !itemAvailable.condition(input);
```

```
    }
  };
  Transition clearSelection = new Transition() {
    public void transition(Input input) {
      int i1 = first.getValue();
      int i2 = ((SecondDigit)input).getValue();
      ItemSlot is = items[i1][i2];
      System.out.println(
        "Clearing selection: item " + is +
        " costs " + is.getPrice() +
        " and has quantity " + is.getQuantity());
      first = null;
    }
  };
  Transition dispense = new Transition() {
    public void transition(Input input) {
      int i1 = first.getValue();
      int i2 = ((SecondDigit)input).getValue();
      ItemSlot is = items[i1][i2];
      System.out.println("Dispensing item " +
        is + " costs " + is.getPrice() +
        " and has quantity " + is.getQuantity());
      items[i1][i2].decrQuantity();
      System.out.println("New Quantity " +
        is.getQuantity());
      amount -= is.getPrice();
      System.out.println("Amount remaining " +
        amount);
    }
  };
  Transition showTotal = new Transition() {
    public void transition(Input input) {
      amount += ((Money)input).getValue();
      System.out.println("Total amount = " +
        amount);
    }
  };
  Transition returnChange = new Transition() {
```

```
    public void transition(Input input) {
      System.out.println("Returning " + amount);
      amount = 0;
    }
  };
  Transition showDigit = new Transition() {
    public void transition(Input input) {
      first = (FirstDigit)input;
      System.out.println("First Digit= "+ first);
    }
  };
  public VendingMachine() {
    super(VM.quiescent); // Initial state
    for(int i = 0; i < items.length; i++)
      for(int j = 0; j < items[i].length; j++)
        items[i][j] = new ItemSlot((j+1)*25, 5);
    items[3][0] = new ItemSlot(25, 0);
    buildTable(new Object[][][]{
      { {VM.quiescent}, // Current state
        // Input, test, transition, next state:
        {Money.class, null,
         showTotal, VM.collecting}},
      { {VM.collecting}, // Current state
        // Input, test, transition, next state:
        {Quit.quit, null,
         returnChange, VM.quiescent},
        {Money.class, null,
         showTotal, VM.collecting},
        {FirstDigit.class, null,
         showDigit, VM.selecting}},
      { {VM.selecting}, // Current state
        // Input, test, transition, next state:
        {Quit.quit, null,
         returnChange, VM.quiescent},
        {SecondDigit.class, notEnough,
         clearSelection, VM.collecting},
        {SecondDigit.class, itemNotAvailable,
         clearSelection, VM.unavailable},
```

```
        {SecondDigit.class, itemAvailable,
         dispense, VM.wantMore}},
      { {VM.unavailable}, // Current state
        // Input, test, transition, next state:
        {Quit.quit, null,
         returnChange, VM.quiescent},
        {FirstDigit.class, null,
         showDigit, VM.selecting}},
      { {VM.wantMore}, // Current state
        // Input, test, transition, next state:
        {Quit.quit, null,
         returnChange, VM.quiescent},
        {FirstDigit.class, null,
         showDigit, VM.selecting}},
    });
  }
} ///:~
```

测试自动售货机（Testing the machine）

```
//: statemachine:vendingmachine:VendingMachineTest.java
// Demonstrates use of StateMachine.java

package statemachine.vendingmachine;
import statemachine2.*;
import junit.framework.*;

public class VendingMachineTest extends TestCase {
  VendingMachine vm = new VendingMachine();
  Input[] inputs = {
    Money.quarter,
    Money.quarter,
    Money.dollar,
    FirstDigit.A,
    SecondDigit.two,
    FirstDigit.A,
    SecondDigit.two,
    FirstDigit.C,
```

```
    SecondDigit.three,
    FirstDigit.D,
    SecondDigit.one,
    Quit.quit,
  };
  public void test() {
    for(int i = 0; i < inputs.length; i++)
      vm.nextState(inputs[i]);
  }
  public static void main(String[] args) {
    junit.textui.TestRunner.run(VendingMachineTest.class);
  }
} ///:~
```

工具

当状态机越来越复杂的时候，另外一种方法就是使用自动化工具通过配置一张表让工具来产生状态机的代码。你可以用像 Python 这样的语言自己写一个，但是已经有人写好了免费的，并且是开放源码的工具，比如 Libero，可以在这里找到 http://www.imatix.com./

用于表驱动（table-driven）的代码：通过配置达到灵活性

通过匿名内部类来实现表驱动

参见 TIJ 第 9 章 ListPerformance.java 那个例子。

以及 GreenHouse.java。

练习

1. 用类似 TransitionTable.java 的那种方法解决 “Washer” problem。
2. 写一个状态机系统，由当前状态和输入信息共同决定系统的下一个状态。每个状态必须保存一个引用到代理对象（状态控制器），这样代理对象才能发起改变状态的请求。用 HashMap 创建一个状态表，用一个 String 来做主键代表新状态的名称，用新状态对象做 HashMap 的值。在每个状态子类内部用它自己的状态转换表重载 nextState()函数。NextState()的输入是从一个文本文件读入的一个单词，这个文本文件每行是一个单词。
3. 改写上面那个练习，通过创建/修改一个单独的多维数组来配置状态机。

4. 改写 state 模式那一节 关于“mood”的那个练习，要求用 StateMachine.java 把它写成一个状态机。

5. 用 StateMachine.java 来实现一个电梯的状态机系统。

6. 用 StateMachine.java 来实现一个空调系统。

7. 生成器（generator）是一个能够产生别的对象的对象，有点类似于工厂（factory），但是生成器函数不需要任何参数。写一个 MouseMoveGenerator，每次调用的时候它会产生正确的 MouseMove 动作作为输出（也就是说，老鼠必须按照正确的顺序动作，这样以来可能的动作都取决于前一个动作——这也是一个状态机）。添加一个 iterator()方法用来产生一个迭代器，这个方法需要传入一个 int 型的参数来指定总共所需动作的个数。

模式重构（Pattern refactoring）

这一章我们会专注于通过逐步演化的方式应用设计模式来解决问题。也就是说，一开始我们会用比较粗糙的设计作为最初的解决方案，然后检验这个解决方案，进而针对这个问题使用不同的设计模式（有些模式是可行的，有些是不合适的）。在寻找更好的解决方案的过程中，最关键的问题是，“哪些东西是变化的？”

这个过程有点像 Martin Fowler 在《重构：改善既有代码的设计》[12]那本书里谈到的那样（尽管他是通过代码片断而不是模式级别的设计来讨论重构）。以某个解决方案作为开始，当你发现这个解决方案不能再满足你的需要的时候就修正它。当然，这是一种很自然的做法，但是对于过程式的计算机编程来说要完成它是相当困难的，大家对于代码重构和设计重构的接受更加说明了面向对象编程是个“好东西”。

模拟一个废品回收器（Simulating the trash recycler）

这个问题的本质是这样的，废品在未分类的情况下被扔进垃圾箱，这样特定的类别信息就丢失了。但是，到后面为了给这些废品正确的分类，那些特定的类别信息又得被恢复出来。一开始，我们采用 RTTI（《Thinking in Java》第二版第 12 章有述）作为解决方案。

这并非是一个轻而易举就能完成的设计，因为它有一些额外的限制。也正是因为有了这些限制才是这个问题更加有趣——它更像你在工作中可能会碰到的那些棘手的问题。额外的限制是指，这些废品运到废品回收工厂(trash recycling plant)的时候，它们是混合在一起的。我们的程序必须要模拟废品分类。这正是需要 RTTI 的地方，你有一大堆叫不出名字的废品碎片，而我们的程序需要找出它们的确切类型。

```
//: refactor:recyclea:RecycleA.java
// Recycling with RTTI.


package refactor.recyclea;
import java.util.*;
import java.io.*;
import junit.framework.*;


abstract class Trash {
  private double weight;
  Trash(double wt) { weight = wt; }
  abstract double getValue();
```

[12] Addison-Wesley, 1999。

```
  double getWeight() { return weight; }
  // Sums the value of Trash in a bin:
  static void sumValue(Iterator it) {
    double val = 0.0f;
    while(it.hasNext()) {
      // One kind of RTTI:
      // A dynamically-checked cast
      Trash t = (Trash)it.next();
      // Polymorphism in action:
      val += t.getWeight() * t.getValue();
      System.out.println(
        "weight of " +
        // Using RTTI to get type
        // information about the class:
        t.getClass().getName() +
        " = " + t.getWeight());
    }
    System.out.println("Total value = " + val);
  }
}

class Aluminum extends Trash {
  static double val  = 1.67f;
  Aluminum(double wt) { super(wt); }
  double getValue() { return val; }
  static void setValue(double newval) {
    val = newval;
  }
}

class Paper extends Trash {
  static double val = 0.10f;
  Paper(double wt) { super(wt); }
  double getValue() { return val; }
  static void setValue(double newval) {
    val = newval;
  }
}
```

```
class Glass extends Trash {
  static double val = 0.23f;
  Glass(double wt) { super(wt); }
  double getValue() { return val; }
  static void setValue(double newval) {
    val = newval;
  }
}

public class RecycleA extends TestCase  {
  Collection
    bin = new ArrayList(),
    glassBin = new ArrayList(),
    paperBin = new ArrayList(),
    alBin = new ArrayList();
  private static Random rand = new Random();
  public RecycleA() {
    // Fill up the Trash bin:
    for(int i = 0; i < 30; i++)
      switch(rand.nextInt(3)) {
        case 0 :
          bin.add(new
            Aluminum(rand.nextDouble() * 100));
          break;
        case 1 :
          bin.add(new
            Paper(rand.nextDouble() * 100));
          break;
        case 2 :
          bin.add(new
            Glass(rand.nextDouble() * 100));
    }
  }
  public void test() {
    Iterator sorter = bin.iterator();
    // Sort the Trash:
    while(sorter.hasNext()) {
```

```
      Object t = sorter.next();
      // RTTI to show class membership:
      if(t instanceof Aluminum)
        alBin.add(t);
      if(t instanceof Paper)
        paperBin.add(t);
      if(t instanceof Glass)
        glassBin.add(t);
    }
    Trash.sumValue(alBin.iterator());
    Trash.sumValue(paperBin.iterator());
    Trash.sumValue(glassBin.iterator());
    Trash.sumValue(bin.iterator());
  }
  public static void main(String args[]) {
    junit.textui.TestRunner.run(RecycleA.class);
  }
} ///:~
```

在与本书配套的源代码清单上，上面那个文件位于 recyclea 子目录里，而 recyclea 又是 refactor 子目录的一个分支。拆包（unpacking）工具会把它放到适当的子目录里。这么做的理由是，本章把这个特定的例子重写了好多次，把每个版本放到它们自己的目录里（通过使用每个目录的默认 package，程序调用也很简单）可以避免类名字冲突。

程序创建了几个 ArrayList 对象用来存放 Trash 对象的引用。当然，ArrayLists 实际上存放的是 Objects 对象，这样它们就可以存放任何东西。它们之所以存放 Trash 对象（或者由 Trash 派生出来的对象）只不过是因为你的小心翼翼，你不把 Trash 以外的东西传给它们。如果你往 ArrayList 里存放了“错误”的东西，你不会在编译时刻得到警告或者错误——你只能在运行时刻通过异常发现这些错误。

当 Trash 对象的引用添加到 ArrayList 以后，它们就丢失了特定的类别信息，变为只是 Object 对象的引用（它们被 upcast 了）。但是，由于多态的存在，当通过 Iterator sorter 调用动态绑定的方法时它还是会产生合适的行为，一旦最终的 Object 对象被 cast 回 Trash 对象，Trash.sumValue( )也采用一个 Iterator 来完成针对 ArrayList 中每个对象的操作。

先把不同类型的 Trash 对象 upcast 并放到一个能够存放基类引用的容器里，然后再把它们 downcast 出来，这么做看起来很傻。为什么不在一开始直接把 Trash 对象放

到适当的容器里？ （事实上，这就是废品回收这个例子令人迷惑的地方）。对于上面的程序要这么改动是很容易的，但是有时候采用 downcasting 这种方法对于某些系统的结构和灵活性都是很有好处的。

上面的程序满足了设计要求：它能够工作。如果只要求一个一次性的解决方案，那上面的方法就可以了。但是实用的程序通常是需要随着时间演化的，所以你必须得问问，“如果情况改变了会怎么样呢？”比如，现在硬纸板成了有用的可循环利用的物品，那该怎么把它集成到上面的系统呢（尤其是当程序又大又复杂的时候）。因为上面的例子里 Switch 语句里那些类型检查的代码是分散在整个程序里的，每次添加新类型的时候你就得查找所有类型检查的代码，如果你漏掉一个编译器是不会通过报告错误的方式给你提供任何帮助的。

这里针对每一种类型都进行测试，其实是对 RTTI 的一种误用。如果你只是因为某种子类型需要特殊对待而对其进行测试，那可能是恰当的。但是如果你是在针对 Switch 语句的每一个类型都进行测试，那么你可能是错过了某些重要的东西，而且这将注定使你的代码更加难以维护。下一小节，我们会看看这一程序是如何通过几个阶段的演化变得更加灵活的。对于程序设计，这是一个有价值的例子。

改进现有设计（Improving the design）

《设计模式》一书中，是围绕着“随着程序不断演化哪些东西将会发生变化？”这个问题来组织解决方案的。这对于任何设计来说通常都是最重要的一个问题。如果你能够围绕这个问题的答案来构建你的系统，将会带来一举两得的好处：不仅仅是你的系统容易维护（而且廉价），而且你还很可能创造出可以重用的组件（components），这样别的系统就更容易构建。这是面向对象编程本来就有的好处，但它不会自动发生；它需要你对于问题的思考和洞察力。这一小节我们来看看在完善系统的过程中它是怎么发生的。

对于我们的垃圾回收系统来说，“什么是变化的？”这个问题的答案是非常普通的：更多类型的（废品）会被加入到系统中来。也就是说，这个设计的最终目标是使得添加新的类型尽可能的方便。对于废品回收程序，我们想要做的是把所有涉及到特定类型信息的地方都封装起来，这样（如果没有别的原因）任何改动都可以被放到那些封装好的地方。最后的结果是这个过程也在相当程度上使程序其余部分的代码变得整洁。

“多弄些对象（Make more objects）”

这将引出一条常用的面向对象设计原则，我第一次是从 Grady Booch 那里听到的：“如果你的设计过于复杂，那就多弄些对象。”这条原则不但违反直觉而且简单的近乎荒谬，但它却是我所见过的最有用的指导原则。（你可能已经觉察到“多弄些对象”经常等同于“添加另外一个中间层。”）通常来说，如果你发现哪个地方代码

非常凌乱，就可以考虑加入什么样的类可以把它弄的整洁一些。整理代码经常会带来另外一个好处是系统会拥有更好的结构和灵活性。

Trash 对象最初是在 main（ ）函数的 switch 语句里被创建的，

```
for(int i = 0; i < 30; i++)
  switch((int)(rand.nextInt(3)) {
    case 0 :
      bin.add(new
        Aluminum(rand.nextDouble() * 100));
      break;
    case 1 :
      bin.add(new
        Paper(rand.nextDouble() * 100));
      break;
    case 2 :
      bin.add(new
        Glass(rand.nextDouble() * 100));
  }
```

毫无疑问，上面的代码显的有些凌乱，而且当加入新的类型的时候你必须得改变这段代码。如果经常需要添加新类型，比较好的解决办法是使用一个单独的方法（method），这个方法利用所有必需的信息产生一个针对某一合适类型的对象的引用，这个引用会先被 upcast 成一个 trash 对象。《设计模式》一书提到这种方法的时候笼统的把它叫做创建型模式（creational pattern ）（实际上有好几种创建型模式）。这里将要用到的特定模式是工厂方法（Factory Method）的一个变种。这里，工厂方法是 Trash 的一个静态成员函数，而更多的情况下它是作为一个被派生类覆写的方法而存在的。

factory method 模式的思想是这样的，你把创建对象所需要的关键信息传递给它，然后它会把（已经 upcast 成基类的）引用作为返回值传给你。然后，你就可以利用这个对象的多态性了。这么一来，你甚至再也不需要知道被创建对象的确切类型。实际上，factory method 为了防止你意外的误用所创建的对象，而把它的类型信息隐藏起来了。如果你想在不使用多态的情况下操纵对象，就必须得显式的使用 RTTI 和 casting。

但是会有一些小问题，尤其是当你使用更为复杂的方法(这里没有列出)，在基类里定义 factory method 而在派生类里覆写它的时候。

如果（创建）派生类所需的信息需要（比基类）更多的或者是不同的参数，那该怎么办呢？

“创建更多的对象”就可以解决这个问题。为了实现 factory method 模式，Trash 类添加了一个新的叫 factory 的方法。为了隐藏创建对象所需的数据，新加了一个 Messenger 类，它携带了 factory 方法创建合适的 Trash 对象所必需的所有信息（本书开始的时候我们把 Messenger 也称作一个设计模式，但是它确实太简单了，可能你不想把它提升到这么高的高度）。下面是 Messenger 的一个简单实现：

```
class Messenger {
  int type;
  // Must change this to add another type:
  static final int MAX_NUM = 4;
  double data;
  Messenger(int typeNum, double val) {
    type = typeNum % MAX_NUM;
    data = val;
  }
}
```

Messenger 对象的唯一任务就是为 factory( )方法保存它所需的信息。现在，如果某种情况下 factory 方法为了创建某一新类型的 Trash 对象需要更多的或者不同的信息，factory( )接口就没必要改变了。Messenger 类可以通过添加新的数据和新的构造函数来改变，或者采用更典型的面向对象的方法——子类化（subclassing）。

这个简单例子里的 factory( )方法看起来像下面的样子：

```
   static Trash factory(Messenger i) {
    switch(i.type) {
      default: // To quiet the compiler
      case 0:
        return new Aluminum(i.data);
      case 1:
        return new Paper(i.data);
      case 2:
        return new Glass(i.data);
      // Two lines here:
      case 3:
        return new Cardboard(i.data);
    }
```

```
  }
```

这里，可以很简单的决定对象的确切类型，但你可以想象一下更为复杂的系统，在那个系统里 factory( )方法使用复杂难懂的算法。关键问题是这些东西现在都被隐藏到了同一个地方，当添加新类型的时候，你很清楚该到这里来改。

现在，创建新对象要比在 main( )函数里简单多了

```
    for(int i = 0; i < 30; i++)
      bin.add(
        Trash.factory(
          new Messenger(
            rand.nextInt(Messenger.MAX_NUM),
            rand.nextDouble() * 100)));
```

创建 Messenger 对象是为了用它传递数据给 factory( )方法，然后 factory( )方法会在堆上(heap)创建某一类型的 Trash 对象并返回添加到 ArrayList bin 的那些引用。

当然，如果你改变了参数的个数和类型，上面的代码还需要改动，但是如果 Messenger 对象是自动生成的那就可以避免这样的改动了。例如，可以用一个包含所需参数的 ArrayList 传给 Messenger 对象的构造函数（或者直接传给 factory( )方法也可以）。这么做需要在运行时刻解析和检验传入的参数，但它的确提供了最大的灵活性。

从这段代码你可以看出 factory 是负责解决哪一类“一系列变化”的问题的：如果你向系统添加新的类型（所谓变化），必需要改变的只是 factory 内部的代码，也就是说 factory 把这部分变化所带来的影响隔离出来了。

根据原型创建对象的模式

上面的设计有一个问题，它仍然需要在一个集中的地方知道所有对象的类型：就是在 factory( )方法内部。如果经常需要添加新类型到这个系统，那么每添加一种新类型，都需要改变 factory( )方法。当你发现类似情况的时候，有效的做法是，试着更进一步把所有有关类型的信息——包括它的创建——都搬到指代这种类型的类里面去。通过这种方法，添加一个新类型到这个系统，你唯一需要做的就是继承一个单一的类。

为了把与类型创建相关的信息挪到每个特定的 Trash 类型里去，这里将会使用“原型” 模式（出自《设计模式》一书）。大致的思路是这样的，你有一个作为母本（master）的对象序列，你有兴趣想要创建的每种类型都有一个母本。这个序列里的

对象只被用于创建新的对象，它使用的是和内建于 Java 的 Object 根类的 clone( )机制（scheme）并无太大差别的一种操作。在本例中，我们会给这种克隆方法起名为 tClone( )。当你做好准备要创建一个新对象的时候，假设你拥有关于构建你想要创建的那种类型的对象的一些信息，你就可以遍历（move through）母本序列，并且用你手头的信息与母本序列里的原型对象的任何合适的信息进行比较。当你找到一个满足需要的对象时，就克隆它。

这种方案里，创建对象并不需要任何硬编码（hard-coded）的信息。每个对象都知道如何暴露合适的信息和如何克隆它自己。这么一来，当加入一个新的类型到这个系统的时候，factory( )方法就不需要做改动了。

解决 prototyping 问题的一种办法是，添加一些方法用以支持新对象的创建。但是，在 Java1.1 里，如果你持有对于 Class 对象的引用，它已经支持根据这个引用来创建新对象。通过 Java1.1 的反射（《Thinking in Java》第二版第 12 章有介绍），即使你只有一个对于 Class 对象的引用，你也可以调用构造函数。对于 prototyping 问题，这是一个完美的解决方案。

原型列表将会通过一组引用（references）来间接的表示，这些引用是关于你想要创建的 Class 对象的。此外，如果通过原型创建对象失败了，factory( )方法会假定这是因为某个特定的 Class 对象不在列表当中，然后会试图把它加载进来。通过这样动态的加载原型对象，Trash 类不需要知道它所处理的是哪种类型，所以当你添加新类型的时候，它也不需要做任何改动。这使得它在本章接下来的部分很容易的被重用。

```
//: refactor:trash:Trash.java
// Base class for Trash recycling examples.
package refactor.trash;
import java.util.*;
import java.lang.reflect.*;

public abstract class Trash {
  private double weight;
  public Trash(double wt) { weight = wt; }
  public Trash() {}
  public abstract double getValue();
  public double getWeight() { return weight; }
  // Sums the value of Trash given an
  // Iterator to any container of Trash:
  public static void sumValue(Iterator it) {
    double val = 0.0f;
```

```
  while(it.hasNext()) {
    // One kind of RTTI: A dynamically-checked cast
    Trash t = (Trash)it.next();
    val += t.getWeight() * t.getValue();
    System.out.println(
      "weight of " +
      // Using RTTI to get type
      // information about the class:
      t.getClass().getName() + " = " + t.getWeight());
  }
  System.out.println("Total value = " + val);
}
public static class Messenger {
  public String id;
  public double data;
  public Messenger(String name, double val) {
    id = name;
    data = val;
  }
}
// Remainder of class provides
// support for prototyping:
private static List trashTypes = new ArrayList();
public static Trash factory(Messenger info) {
  Iterator it = trashTypes.iterator();
  while(it.hasNext()) {
    // Somehow determine the new type
    // to create, and create one:
    Class tc = (Class)it.next();
    if (tc.getName().indexOf(info.id) != -1) {
      try {
        // Get the dynamic constructor method
        // that takes a double argument:
        Constructor ctor = tc.getConstructor(
            new Class[]{ double.class });
        // Call the constructor
        // to create a new object:
        return (Trash)ctor.newInstance(
```

```
            new Object[]{new Double(info.data)});
        } catch(Exception e) {
          throw new RuntimeException(
            "Cannot Create Trash", e);
        }
      }
    }
    // Class was not in the list. Try to load it,
    // but it must be in your class path!
    try {
      System.out.println("Loading " + info.id);
      trashTypes.add(Class.forName(info.id));
    } catch(Exception e) {
      throw new RuntimeException("Prototype not found", e);
    }
    // Loaded successfully.
    // Recursive call should work:
    return factory(info);
  }
} ///:~
```

基本的 Trash 类以及 sumValue( )都保持和以前一样。这个类其余的部分就是用来支持原型模式的。首先，你会看到两个内部类（它们被声明成静态的，所以它们成为内部类仅仅是为了更好的组织代码）描述了可能出现的异常。接下来是一个叫 trashTypes 的 ArrayList，它是用来保存那些关于 Class 的引用的。

在 Trash.factory( )内部，Messenger 对象的 id（另一个版本的 messenger 类，不同于前面所讨论的那个）里面的 String，包含了要创建的 Trash 对象的类型名称；这个 String 与列表里的 Class 名字进行比较。如果找到一个与之相符的，就是那个需要被创建的对象。当然，存在许多种方法来决定你想要创建的是什么对象。采用现在这种方法是为了能够从一个文件读入信息并将其转化为对象。

一旦你已经了解到需要创建的是哪一种类型的 Trash，那么接下来的事情就可以交给反射方法了。getConstructor( )方法接受一个数组作为参数，这个数组指代你正在查找的那个构造函数的参数，这些参数以它们合适的顺序给出。这里，这个数组是通过 Java1.1 的数组－创建语法（array-creation）动态的被创建的。

```
new Class[] {double.class}
```

这行代码假定每个 Trash 类型都有一个可以接受一个 double 参数的构造函数（注意 double.class 与 Double.class 是截然不同的）。对于一个更加灵活的解决方案，也可以调用 getConstructors( )，它会返回一个包含所有构造函数的数组。

getConstructor( )返回的是一个关于 Constructor（java.lang.reflect 的一部分）对象的引用。你可以通过 newInstance( )方法动态地调用构造函数，newInstance( )接受一个包含实际参数的 Object 数组。再次用 Java 1.1 的语法来创建这个数组：

```
new Object[]{new Double(Messenger.data)}
```

但是在这种情况下，double 必须被放置在一个外覆类（wrapper class）的内部，以使其能够成为这个对象数组的一部分。调用 newInstance( )这个过程，会把 double 提取出来，但是你会看到它有点让人迷惑——这个参数可能是 double，也可能是 Double，但是当你调用的时候你必须总是传入一个 Double。幸运的是，这个问题只有当（参数是）基元类型（primitive type）的时候才存在。

一旦你理解了如何做这件事情，只根据给出的一个关于 Class 的引用来创建新对象这个过程就极其简单了。通过反射（Reflection）你还可以以同样动态的方式来调用方法。

当然，合适的 Class 引用可能不在 trashTypes 列表里。这种情况下，内层循环的 return 语句从来不会被执行，程序会在最后退出（drop out）。这里，程序试图通过动态地加载 Class 对象，并把它加到 trashTypes 列表来调整这种局面。如果还是不能找到，那就真的是出错了，但如果加载是成功的，那么 factory 方法就会以递归（recursive）的方式被调用，再试一次。

就像你将会看到的那样，这个设计的美妙之处在于，尽管它将在不同的情形下被使用，但它的代码不用再改了（假设所有 Trash 的子类都有一个可以接受 double 作为参数的构造函数）。

Trash 的子类

为了适合这种原型创建的机制（scheme），每个 Trash 子类所需要的唯一一件事情就是，它得有一个接受一个 double 参数的构造函数。其余的事情都由 Java 的反射来处理。

下面是不同类型的 Trash，每个都位于它们自己的文件里，而且是 Trash package 的一部分（再说一遍，这是为了更方便地在本章重用）。

```
//: refactor:trash:Aluminum.java
// The Aluminum class with prototyping.
package refactor.trash;
```

```
public class Aluminum extends Trash {
  private static double val = 1.67f;
  public Aluminum(double wt) { super(wt); }
  public double getValue() { return val; }
  public static void setValue(double newVal) {
    val = newVal;
  }
} ///:~

//: refactor:trash:Paper.java
// The Paper class with prototyping.
package refactor.trash;

public class Paper extends Trash {
  private static double val = 0.10f;
  public Paper(double wt) { super(wt); }
  public double getValue() { return val; }
  public static void setValue(double newVal) {
    val = newVal;
  }
} ///:~

//: refactor:trash:Glass.java
// The Glass class with prototyping.
package refactor.trash;

public class Glass extends Trash {
  private static double val = 0.23f;
  public Glass(double wt) { super(wt); }
  public double getValue() { return val; }
  public static void setValue(double newVal) {
    val = newVal;
  }
} ///:~
```

在这里添加一个新型的 Trash 类：

```
//: refactor:trash:Cardboard.java
```

```
// The Cardboard class with prototyping.
package refactor.trash;

public class Cardboard extends Trash {
  private static double val = 0.23f;
  public Cardboard(double wt) { super(wt); }
  public double getValue() { return val; }
  public static void setValue(double newVal) {
    val = newVal;
  }
} ///:~
```

你可以看到，除了构造函数，这些类没有任何特别之处。

从一个外部文件解析 Trash

有关 Trash 对象的信息将会从一个外部文件读入。这个文件包含所有关于每片废品的必要信息，每个条目以 Trash：weight 的形式占据单独的一行，比如：

```
//:! refactor:trash:Trash.dat
refactor.trash.Glass:54
refactor.trash.Paper:22
refactor.trash.Paper:11
refactor.trash.Glass:17
refactor.trash.Aluminum:89
refactor.trash.Paper:88
refactor.trash.Aluminum:76
refactor.trash.Cardboard:96
refactor.trash.Aluminum:25
refactor.trash.Aluminum:34
refactor.trash.Glass:11
refactor.trash.Glass:68
refactor.trash.Glass:43
refactor.trash.Aluminum:27
refactor.trash.Cardboard:44
refactor.trash.Aluminum:18
refactor.trash.Paper:91
refactor.trash.Glass:63
refactor.trash.Glass:50
```

```
refactor.trash.Glass:80
refactor.trash.Aluminum:81
refactor.trash.Cardboard:12
refactor.trash.Glass:12
refactor.trash.Glass:54
refactor.trash.Aluminum:36
refactor.trash.Aluminum:93
refactor.trash.Glass:93
refactor.trash.Paper:80
refactor.trash.Glass:36
refactor.trash.Glass:12
refactor.trash.Glass:60
refactor.trash.Paper:66
refactor.trash.Aluminum:36
refactor.trash.Cardboard:22
///:~
```

别忘了，当给出类名时必须同时包含类所在的路径，否则将会找不到这个类。

读入这个文件使用的是前面定义过的 StringList 那个工具，每一行由 String 的 indexOf( )方法产生“：”的索引号将该行分解。先是利用这个索引号通过 String 的 subString( )方法提取出 trash 类型的名字，接下来是得到重量，这个重量使用静态的 Double.valueOf( )方法转换成一个 double 值。Trim()方法去掉字符串两端的空格。

Trash 解析器放在一个单独的文件中，这是因为它将在本章不断被重用。

```
//: refactor:trash:ParseTrash.java
// Parse file contents into Trash objects,
// placing each into a Fillable holder.
package refactor.trash;
import java.util.*;
import java.io.*;
import com.bruceeckel.util.StringList;

public class ParseTrash {
  public static void
  fillBin(String filePath, Fillable bin) {
    Iterator it = new StringList(filePath).iterator();
```

```
    while(it.hasNext()) {
      String line = (String)it.next();
      String type = line.substring(0,
        line.indexOf(':')).trim();
      double weight = Double.valueOf(
        line.substring(line.indexOf(':') + 1)
          .trim()).doubleValue();
      bin.addTrash(
        Trash.factory(new Trash.Messenger(type, weight)));
    }
  }
  // Special case to handle Collection:
  public static void
  fillBin(String filePath, Collection bin) {
    fillBin(filePath, new FillableCollection(bin));
  }
} ///:~
```

在 RecycleA.java 里，使用一个 ArrayList 来保存 Trash 对象。其实，其他类型的容器也可以使用。为了允许这么做，fillBin()的第一个版本接受一个关于 Fillable 的引用，而 Fillable 也就是支持 addTrash()方法的一个接口。

```
//: refactor:trash:Fillable.java
// Any object that can be filled with Trash.
package refactor.trash;

public interface Fillable {
  void addTrash(Trash t);
} ///:~
```

任何支持这个接口的东西都可以被 fillBin 使用。当然，Collection 并没有实现 Fillable，所以它不会工作。因为在大多数例子里使用的都是 Collection，所以添加第二个被重载过并且接受一个 Collection（作为参数）的 fillBin( )方法就显得很有意义了。任何 Collection 都可以通过一个适配器（adapter）类被当作一个 Fillable 对象来使用。

```
//: refactor:trash:FillableCollection.java
// Adapter that makes a Collection Fillable.
```

```
package refactor.trash;
import java.util.*;

public class FillableCollection implements Fillable {
  private Collection c;
  public FillableCollection(Collection cc) { c = cc; }
  public void addTrash(Trash t) { c.add(t); }
} ///:~
```

你可以看到这个类的唯一工作就是把 Fillable 的 addTrash( )方法和 Collection 的 add( )方法连接起来。有了这个类，重载过的 fillBin( )方法就可以在 ParseTrash.java 里和 Collection 一起使用了：

```
  public static void
  fillBin(String filePath, Collection bin) {
    fillBin(filePath, new FillableCollection(bin));
  }
```

这种方法适用于任何常用的容器类。作为另外一种选择，容器类也可以提供它自己的适配器，只要这个适配器实现了 Fillable 接口。（稍后你会在 DynaTrash.java 里看到这种方法）

使用原型创建做废品回收（Recycling with prototyping）

现在你可以看到改动过的 RecycleA.java 版本，它使用了原型创建（prototyping）技术。

```
//: refactor:recycleap:RecycleAP.java
// Recycling with RTTI and Prototypes.
package refactor.recycleap;
import refactor.trash.*;
import java.util.*;
import junit.framework.*;

public class RecycleAP extends TestCase  {
  Collection
    bin = new ArrayList(),
    glassBin = new ArrayList(),
    paperBin = new ArrayList(),
    alBin = new ArrayList();
```

```
  public RecycleAP() {
    // Fill up the Trash bin:
    ParseTrash.fillBin("../trash/Trash.dat", bin);
  }
  public void test() {
    Iterator sorter = bin.iterator();
    // Sort the Trash:
    while(sorter.hasNext()) {
      Object t = sorter.next();
      // RTTI to show class membership:
      if(t instanceof Aluminum)
        alBin.add(t);
      else if(t instanceof Paper)
        paperBin.add(t);
      else if(t instanceof Glass)
        glassBin.add(t);
      else
        System.err.println("Unknown type " + t);
    }
    Trash.sumValue(alBin.iterator());
    Trash.sumValue(paperBin.iterator());
    Trash.sumValue(glassBin.iterator());
    Trash.sumValue(bin.iterator());
  }
  public static void main(String args[]) {
    junit.textui.TestRunner.run(RecycleAP.class);
  }
} ///:~
```

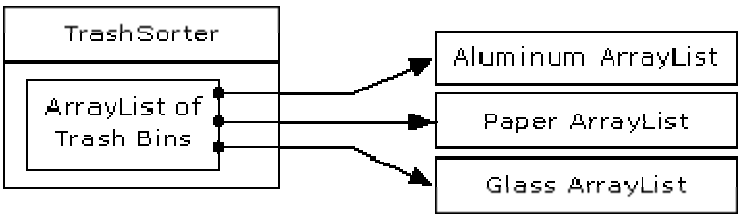
所有的 Trash 对象，以及 ParseTrash 和其它的辅助类，现在都是 refactor.trash package 的一部分，所以就可以简单地把它们引入了。

打开包含 Trash 描述的数据文件并对这个文件进行解析，这个过程被包裹（wrap）进了 ParseTrash.fillBin( )这个静态方法，所以现在它就不再是我们设计焦点的一部分了。你会看到在本章的其余部分，无论加入什么新类，ParseTrash.fillBin( )会一直工作并且不用再作改动，这意味着一个好的设计。

就对象创建来说，这个设计确实在你需要往系统添加新类型的时候，很大程度上把变化集中到了某个地方。然而，这里也有一个关于使用 RTTI 的明显的问题，上面的代码清楚地显示了这一点。程序看起来跑的很好，但是它从来检测不到任何硬纸板（Cardboard），尽管确实有一个 Cardboard 在列表里！发生这种情况是因为使用了 RTTI，而它只寻找你告诉它要找的那些类型。RTTI 被误用的一个预兆（clue）是系统里的每一种类型都在被检验，而不是只有一种或者是所有类型的一个子集在被检验。稍后你会看到，当你是要检验所有类型的时候，存在一些方法使用多态来作为替代。但是如果你以这种方式大量地使用 RTTI，当你往系统添加一个新类型的时候，你会很容易地忘记在程序里做必要的改动，这会产生一个难以查找的 bug。所以在这种情况下试着去掉 RTTI 是值得的，不仅仅是为了审美上的理由——它会产生出更容易维护的代码。

Abstracting usage

解决了创建的问题，现在是时候剖析一下这个设计剩下的部分了：就是使用这些类的地方。既然把废品分拣到垃圾箱这个行为是极其丑陋和过度暴露的，为什么不把这个过程隐藏到一个类里去呢？这就是以下原则，“如果必须做一些丑陋的事情，那至少把它集中到一个类里”。它看起来像下面这样：



无论什么时候，当有新类型加入这个模型，TrashSorter 对象的初始化现在必须得更改。你可能会想象 TrashSorter 类应该看起来像下面的样子：

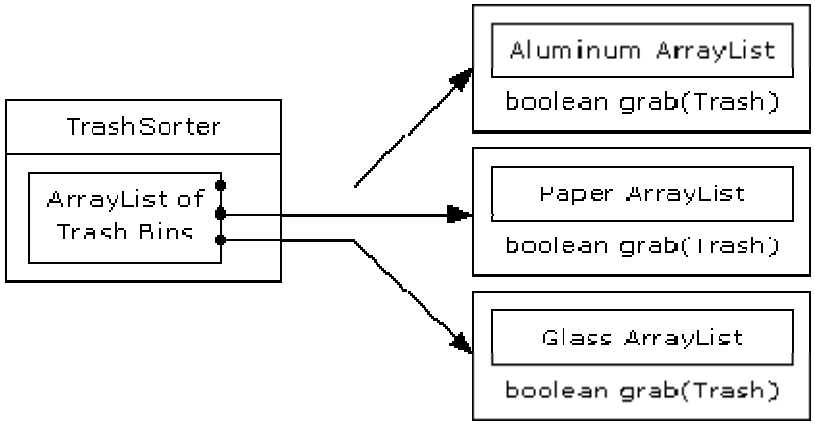
```
class TrashSorter extends ArrayList {
  void sort(Trash t) { /* ... */ }
}
```

也就是说，TrashSorter 是一个 ArrayList，它的元素是关于一对 ArrayLists 的引用，而这些 ArrayList 的元素又是关于 Trash 的引用，而且使用 add( )方法你可以加载另一个 ArrayList，像这样：

```
TrashSorter ts = new TrashSorter();
ts.add(new ArrayList());
```

但是现在 sort( )成了一个问题。静态编码的（statically-coded）方法如何处理一个新加入的类型呢？为了解决这个问题，类型信息必须从 sort( )移除，这样他所需

要做的就只是调用一个能够处理类型细节的泛型（generic）方法。这种方法当然是另外一种描述动态绑定（dynamically-bound）的方法。所以 sort()仅仅是遍历（move through）这个序列，并针对每一个 ArrayList 调用一个动态绑定的方法。既然这个方法的任务是抓取（grab）它感兴趣的废品碎片，所以它被叫做 grab(Trash)。现在结构看起来是这样的：



TrashSorter 需要调用每一个 grab( )方法并且根据当前 ArrayList 所持有的 Trash 的类型得到不同的结果。也就是说，每个 ArrayList 必须知道它所持有的类型。针对这个问题，经典的做法是创建一个“Trash bin”基类，并且让每一个你想持有的类型从它继承而来成为一个派生类。如果 Java 有一种把类型参数化（parameterized type）的机制，那将是最直接的方法。但是我们不会手工写出（这种机制会为我们构建的）所有的类，进一步的观察会产生一个更好的方法。

有一个基本的 OOP 的原则是说“状态变化的时候使用数据成员，行为变化的时候使用多态。”你的第一想法可能会认为 grab( )方法的行为对于一个持有 Paper 的 ArrayList 和一个持有 Glass 的 ArrayList 当然会有所不同。但是它所做的严格取决于类型，而不是任何其它的东西。这也可以被解释成一个不同的状态，而且既然 Java 有一个用以表示类型（Class）的类，它就可以被用来决定某个特定的 Tbin 将要持有的 Trash 类型。

Tbin 的构造函数要求你传给它你所选择的 Class。这会告诉 ArrayList 它将要持有的是什么类型。然后 grab( )方法使用 Class BinType 和 RTTI 来确定你传给它的 Trash 对象是否与它将要抓取的类型相符。

下面是程序的一个新的版本。

```
//: refactor:recycleb:RecycleB.java
// Containers that grab objects of interest.
package refactor.recycleb;
import refactor.trash.*;
import java.util.*;
```

```
import junit.framework.*;

// A container that admits only the right type
// of Trash (established in the constructor):
class Tbin {
  private List list = new ArrayList();
  private Class type;
  public Tbin(Class binType) { type = binType; }
  public boolean grab(Trash t) {
    // Comparing class types:
    if(t.getClass().equals(type)) {
      list.add(t);
      return true; // Object grabbed
    }
    return false; // Object not grabbed
  }
  public Iterator iterator() {
    return list.iterator();
  }
}

class TbinList extends ArrayList {
  void sortTrashItem(Trash t) {
    Iterator e = iterator(); // Iterate over self
    while(e.hasNext())
      if(((Tbin)e.next()).grab(t)) return;
    // Need a new Tbin for this type:
    add(new Tbin(t.getClass()));
    sortTrashItem(t); // Recursive call
  }
}

public class RecycleB extends TestCase  {
  Collection bin = new ArrayList();
  TbinList trashBins = new TbinList();
  public RecycleB() {
    ParseTrash.fillBin("../trash/Trash.dat",bin);
  }
```

```
  public void test() {
    Iterator it = bin.iterator();
    while(it.hasNext())
      trashBins.sortTrashItem((Trash)it.next());
    Iterator e = trashBins.iterator();
    while(e.hasNext())
      Trash.sumValue(((Tbin)e.next()).iterator());
    Trash.sumValue(bin.iterator());
  }
  public static void main(String args[]) {
    junit.textui.TestRunner.run(RecycleB.class);
  }
} ///:~
```

Tbin 包含一个 Class 引用类型，它在构造函数里设置所要抓取的类型。Grab( )方法把这个类型和你所传递给它的对象进行比照。注意在这个设计里，grab( )只接受 Trash 对象，所以你会在编译时刻得到针对基本类型的类型检验，但是你也可以只是接受 Objct，它仍然可以工作。

TbinList 持有一组关于 Tbin 的引用，所以 sort( )可以在它寻找那个和你所传递给它的 Trash 对象相匹配的对象的时候遍历（iterate through）Tbins。如果没有找到一个与之匹配的对象，它就为这个未能找到匹配对象的类型创建一个新的 Tbin，然后对它自己进行一次递归调用——下一轮的时候，新的 bin 就可以被找到了。

请注意这些代码的通用性（genericity）：加入新类型的时候它们根本无需做改动。如果你这一堆代码在加入一个新类型（或者发生其它改变）的时候不需要改动，那你就有了一个易于扩展的系统。

多重分派（Multiple dispatching）

上面的设计当然是令人满意的。往系统里添加新类型（包括添加或修改重要的类）都不会导致代码改变波及到整个系统。此外，RTTI 并没有像在 RevycleA.java 里那样被误用。但是，我们有可能更进一步以一个力求纯化者的眼光来审视一下 RTTI，进而得出结论，所有有关 RTTI 的东西都应改从那些把废品分拣到垃圾箱的操作里移除。

为了实现这个想法，你必须首先接受这么一个观点，即所有依赖于类型的活动——比如侦测某片废品的类型并把它放入相应的垃圾箱——都应改通过多态和动态绑定来控制。

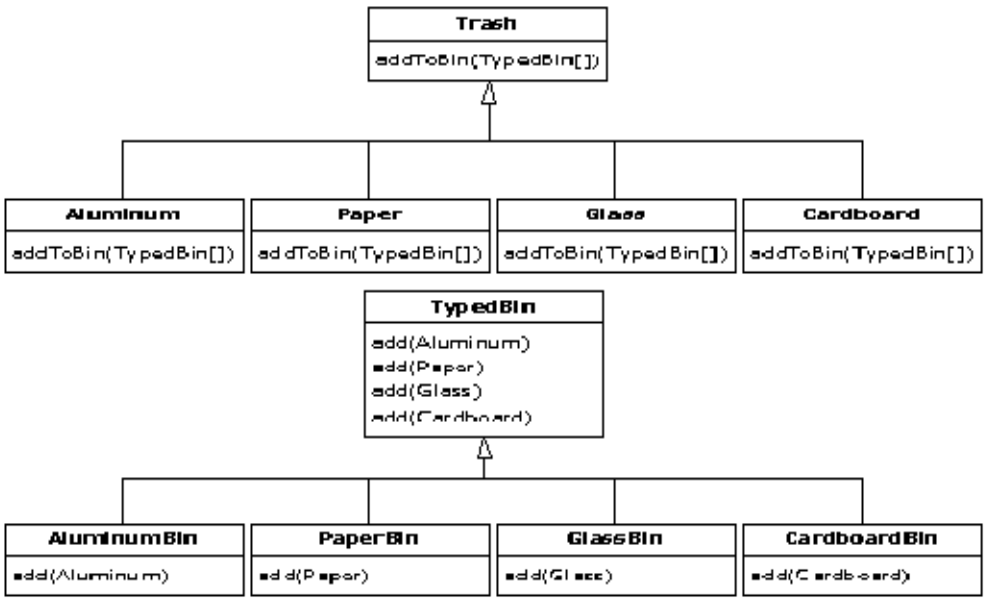
先前的那个例子，子类先根据类型来分拣，然后在元素都是某一特定类型的序列上进行操作。但是无论什么时候你发现自己实在取出特定的类型，就应改停下来想一想。多态（动态绑定的方法调用）的全部思想就是替你处理特定类型的信息。所以你为什么还要捕捉类型呢？

答案可能是你所未曾想到的：Java 只能做单次分派（single dispatching）。也就是说，如果你正在针对多于一个的“类型未知的对象”进行操作，Java 将会只针对其中一种类型触发动态绑定机制。这并没有解决问题，所以你最终会手动地侦测某些类型并且有效地产生你自己的动态绑定行为。

上述问题的解决方案被称为多重分派（multiple dispatching），意思是说搭一个架子，从而一个单独的方法调可以产生多于一个的动态方法调用，进而在这个过程中决定多于一种的类型体系（hierarchy）。为了达到这个效果，你需要处理一种以上的类型体系：每个分派都需要一个类型体系。下面的例子就采用了两个体系：现有的 Trash 体系和一个将要存放这些废品的废品箱的体系。第二个体系并不总是显而易见的，这种情况下为了产生多重分派就需要创建它（这里只有两个分派，所以就称为双重分派（double dispatching））。

实现双重分派（Implementing the double dispatch）

记住，多态只能通过方法调用来体现，所以如果你想让双重分派出现，必须得有两个方法调用：每个用来决定一个体系内部的类型。Trash 体系会有一个新的方法叫做 addToBin( )，它接受一个 TypedBin 类型的数组作为参数。它会遍历（step through）这个数组并且试着把它自己添加到相应的废品箱，这就是你会看到双重分派（发生）的地方。



还有一个关于 TypedBin 的新的层次体系，它包含自己的方法，叫做 add( )，这个方法使用的时候也是多态的。但是还有一点比较拧（twist）：为了接受不同的 trash 类型作为参数，add( )是重载过的。所以，双重分派机制的核心部分也包括重载。

重新设计这个程序产生一个让人进退两难的问题：现在 Trash 基类必须包含一个 addBin( )方法。一种做法是拷贝所有的代码并且改变基类。当你没法控制源代码的时候可以采用另外一种方法，把 addBin( )方法方到一个接口里，让 Trash 单独存在，并且由它继承而来新的特定类型，诸如铝（Aluminum），纸（Paper），玻璃（Glass）和硬纸板（Cardboard）。这里将要采用的就是这种方法。

这个设计里，大多数类都必须是公共的（public），所以它们被放在各自的文件内。下面是接口：

```
//: refactor:doubledispatch:TypedBinMember.java
// An interface for adding the double
// dispatching method to the trash hierarchy
// without modifying the original hierarchy.
package refactor.doubledispatch;

interface TypedBinMember {
  // The new method:
  boolean addToBin(TypedBin[] tb);
} ///:~
```

在 Aluminum、Paper、Glass、以及 Cardboard，每个特定子类型内部，TypedBinMember 接口的 addToBin( )方法都会被实现，但是看起来每种情况下它们的代码又都是一样的：

```
//: refactor:doubledispatch:DDAluminum.java
// Aluminum for double dispatching.
package refactor.doubledispatch;
import refactor.trash.*;

public class DDAluminum extends Aluminum
    implements TypedBinMember {
  public DDAluminum(double wt) { super(wt); }
  public boolean addToBin(TypedBin[] tb) {
    for(int i = 0; i < tb.length; i++)
      if(tb[i].add(this))
        return true;
```

```
    return false;
  }
} ///:~

//: refactor:doubledispatch:DDPaper.java
// Paper for double dispatching.
package refactor.doubledispatch;
import refactor.trash.*;

public class DDPaper extends Paper
    implements TypedBinMember {
  public DDPaper(double wt) { super(wt); }
  public boolean addToBin(TypedBin[] tb) {
    for(int i = 0; i < tb.length; i++)
      if(tb[i].add(this))
        return true;
    return false;
  }
} ///:~

//: refactor:doubledispatch:DDGlass.java
// Glass for double dispatching.
package refactor.doubledispatch;
import refactor.trash.*;

public class DDGlass extends Glass
    implements TypedBinMember {
  public DDGlass(double wt) { super(wt); }
  public boolean addToBin(TypedBin[] tb) {
    for(int i = 0; i < tb.length; i++)
      if(tb[i].add(this))
        return true;
    return false;
  }
} ///:~

//: refactor:doubledispatch:DDCardboard.java
// Cardboard for double dispatching.
```

```
package refactor.doubledispatch;
import refactor.trash.*;

public class DDCardboard extends Cardboard
    implements TypedBinMember {
  public DDCardboard(double wt) { super(wt); }
  public boolean addToBin(TypedBin[] tb) {
    for(int i = 0; i < tb.length; i++)
      if(tb[i].add(this))
        return true;
    return false;
  }
} ///:~
```

每个 addToBin( )方法的代码调用数组里每个 TypedBin 对象的 add( )方法。但是请注意，参数是：this。针对每种 Trash 子类的 this 的类型是不同的，所以代码也是不同的（尽管如果某种参数化类型的机制曾经加入 Java 的话，这些代码会从中受益）。这就是双重分派发生的第一个地方，因为一旦处于这个方法内部，你就可以知道是 Aluminum 还是 Paper，等等。在调用 add( )的过程中，这个信息是通过 this 的类型来传递的，编译器会找出恰当的重载过的 add( )方法来调用。但是因为 tb[i]产生一个针对 TypedBin 基类的引用，所以这个调用会根据当前选中的 TypedBin 的类型来最终调用一个不同的方法。这就是第二个分派。

下面是 TypedBin 基类：

```
//: refactor:doubledispatch:TypedBin.java
// A container for the second dispatch.
package refactor.doubledispatch;
import refactor.trash.*;
import java.util.*;

public abstract class TypedBin {
  Collection c = new ArrayList();
  protected boolean addIt(Trash t) {
    c.add(t);
    return true;
  }
  public Iterator iterator() {
    return c.iterator();
  }
```

```
  public boolean add(DDAluminum a) {
    return false;
  }
  public boolean add(DDPaper a) {
    return false;
  }
  public boolean add(DDGlass a) {
    return false;
  }
  public boolean add(DDCardboard a) {
    return false;
  }
} ///:~
```

你可以看到，被重载的 add( )方法全都返回 false。如果这个方法在某个派生类里没有被重载，它会继续返回 false，而调用者（这个例子里指 addToBin( )方法）会假设当前的 Trash 对象还没有被成功地加入到容器，并且继续查找合适的容器。

在 TypedBin 的每种子类内部，只有一个重载的（overloaded）方法根据被创建的废品箱（bin）的类型被覆写了（overridden）。例如，CardboardBin 覆写成了 add(DDCardboard)，覆写过的方法把 trash 对象添加到它的容器里并且返回 true，而 CardboardBin 里所有其余的 add( )方法都继续返回 false，因为它们没有被覆写。如果 Java 里有参数化类型的机制，它就可以允许自动产生代码，这里是又一次碰到这种情况。（使用 C++模板，你就用不着显式地写出那些子类或者把 addToBin( )方法放到 Trash 里。）

因为本例中 trash 类型都被定制了，并且放到了不同的目录，所以为了让它工作你得需要一个不同的 trash 数据文件。这里式一个可能的 DDTrash.dat 文件：

```
//:! refactor:doubledispatch:DDTrash.dat
refactor.doubledispatch.DDGlass:54
refactor.doubledispatch.DDPaper:22
refactor.doubledispatch.DDPaper:11
refactor.doubledispatch.DDGlass:17
refactor.doubledispatch.DDAluminum:89
refactor.doubledispatch.DDPaper:88
refactor.doubledispatch.DDAluminum:76
refactor.doubledispatch.DDCardboard:96
refactor.doubledispatch.DDAluminum:25
refactor.doubledispatch.DDAluminum:34
```

```
refactor.doubledispatch.DDGlass:11
refactor.doubledispatch.DDGlass:68
refactor.doubledispatch.DDGlass:43
refactor.doubledispatch.DDAluminum:27
refactor.doubledispatch.DDCardboard:44
refactor.doubledispatch.DDAluminum:18
refactor.doubledispatch.DDPaper:91
refactor.doubledispatch.DDGlass:63
refactor.doubledispatch.DDGlass:50
refactor.doubledispatch.DDGlass:80
refactor.doubledispatch.DDAluminum:81
refactor.doubledispatch.DDCardboard:12
refactor.doubledispatch.DDGlass:12
refactor.doubledispatch.DDGlass:54
refactor.doubledispatch.DDAluminum:36
refactor.doubledispatch.DDAluminum:93
refactor.doubledispatch.DDGlass:93
refactor.doubledispatch.DDPaper:80
refactor.doubledispatch.DDGlass:36
refactor.doubledispatch.DDGlass:12
refactor.doubledispatch.DDGlass:60
refactor.doubledispatch.DDPaper:66
refactor.doubledispatch.DDAluminum:36
refactor.doubledispatch.DDCardboard:22
///:~
```

下面是程序的其余部分：

```
//: refactor:doubledispatch:DoubleDispatch.java
// Using multiple dispatching to handle more
// than one unknown type during a method call.
package refactor.doubledispatch;
import refactor.trash.*;
import java.util.*;
import junit.framework.*;

class AluminumBin extends TypedBin {
  public boolean add(DDAluminum a) {
    return addIt(a);
```

```
  }
}

class PaperBin extends TypedBin {
  public boolean add(DDPaper a) {
    return addIt(a);
  }
}

class GlassBin extends TypedBin {
  public boolean add(DDGlass a) {
    return addIt(a);
  }
}

class CardboardBin extends TypedBin {
  public boolean add(DDCardboard a) {
    return addIt(a);
  }
}

class TrashBinSet {
  private TypedBin[] binSet = {
    new AluminumBin(),
    new PaperBin(),
    new GlassBin(),
    new CardboardBin()
  };
  public void sortIntoBins(Iterator it) {
    while(it.hasNext()) {
      TypedBinMember t = (TypedBinMember)it.next();
      if(!t.addToBin(binSet))
        System.err.println("Couldn't add " + t);
    }
  }
  public TypedBin[] binSet() { return binSet; }
}
```

```
public class DoubleDispatch extends TestCase  {
  Collection bin = new ArrayList();
  TrashBinSet bins = new TrashBinSet();
  public DoubleDispatch() {
    // ParseTrash still works, without changes:
    ParseTrash.fillBin("DDTrash.dat", bin);
  }
  public void test() {
    // Sort from the master bin into
    // the individually-typed bins:
    bins.sortIntoBins(bin.iterator());
    TypedBin[] tb = bins.binSet();
    // Perform sumValue for each bin...
    for(int i = 0; i < tb.length; i++)
      Trash.sumValue(tb[i].c.iterator());
    // ... and for the master bin
    Trash.sumValue(bin.iterator());
  }
  public static void main(String args[]) {
    junit.textui.TestRunner.run(DoubleDispatch.class);
  }
} ///:~
```

TrashBinSet 封装了所有不同类型的 TypedBins，还有一个 sortIntoBins( )方法，这个方法是所有双重分派发生的地方。你可以看到，一旦这个结构搭起来以后，（把废品）分拣到不同的 TypedBins 就相当容易了。此外，调用两个动态方法的效率可能比其它任何的分拣（sort）方法都高。

请注意，这个系统在 main( )函数里是多么容易被使用，而且它完全独立于 main( )内的任何特定类型信息。所有其它与 Trash 基类接口打交道的方法同样不受 Trash 类型变化的影响。

添加一个新类型需要做的改动相对来说被隔离开了：你对 TypedBin 进行改动，从 Trash 继承一个新类型，并且继承它的 addToBin( )方法，然后继承一个新的 TypedBin（这实际上只需要拷贝然后简单编辑一下），最后在 TrashBinSet 集中初始化的地方添上这个新类型。

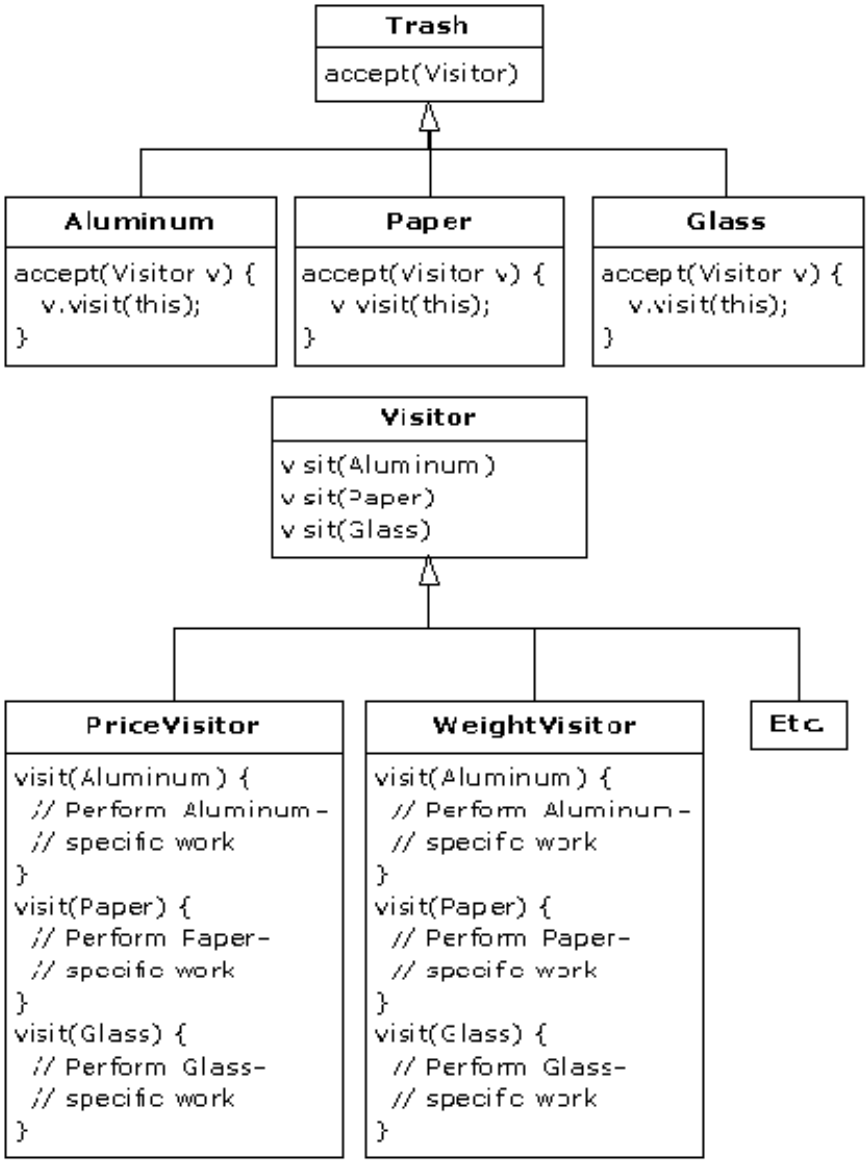
访问者（Visitor）模式

现在考虑应用一个与废品分拣问题目的完全不同的设计模式。

对于这个模式，我们不关心如何就向系统加入新的 Trash 类型进行优化。实际上，这个模式使得添加一个新的 Trash 类型更加复杂。我们所作的假设是，你有一个不能被改变的主类体系（a primary class hierarchy），可能它是从另外一个供应商那里来的，你没法对这个类体系进行改动。但是，你想要往那个类体系里添加新的多态方法，这就意味着通常情况下你必须得向基类接口添加东西。所以说，进退两难的问题在于，你既需要往基类里添加方法，但你又不能动它。那你该怎么办呢？

解决这类问题的设计模式称为“访问者模式（visitor）”（《设计模式》一书最后讲到的那个），它是基于上一节讲的双重分派机制构建的。

访问者模式允许你通过创建一个新的关于 Visitor 类型的类体系来扩展主类型的接口，这个关于 Visitor 类型的类体系用以模拟（virtualize）作用于主类型的那些操作。主类型对象只是简单地接受“访问者”，然后调用访问者的动态绑定的方法。看起来就像下面这样：



现在，如果 v 是一个关于 Aluminum 对象的 Visitable 引用，下面的代码；

```
PriceVisitor pv = new PriceVisitor();
v.accept(pv);
```

使用双重分派来触发两次多态方法调用：第一个用来选择 Aluminum 版本的 accept( )方法，第二个位于 accept( )方法的内部，当特定版本的 visit( )方法通过关于基类 Visitor 的引用 v 被动态调用的时候（才触发）。

这种配置结构（configuration）意味着新的功能可以通过创建 Visitor 的新的子类的形式加入系统。Trash 体系不用改动。这就是访问者模式最大的好处：你可以往一个类体系添加新的多态功能而不用触及那个体系（前提是 accept( )方法已经存在）。请注意，这种好处在这里是有帮助的，但并不是我们最开始想要完成的东西，所以你的第一反应可能会认定这不是我们想要的方案。

但是请看看我们所完成的事情：使用 visitor 这种方案避免了从母本 Trash 序列到各个类型序列的分拣。因此，你可以把所有东西都留在单独的母本序列而只要使用合适的 visitor 遍历（pass through）母本序列就能完成这个目标。尽管这种行为看上去像是访问者模式的副产品，但它确实给我我们想要的（避免使用 RTTI）。

访问者模式里的双重分派负责决定 Trash 和 Visitor 的类型。在下面的例子里，有两个 Visitor 的实现：PriceVisitor 用以决定价格和求总价，WeightVisitor 用以记录重量。

你可以在 recycling 程序改进过的新版本里看到，所有这些都被实现了。

对于 DoubleDispatch.java，Trash 类被单独弄出来，有一个新的接口用来添加 accept( )方法：

```
//: refactor:trashvisitor:Visitable.java
// An interface to add visitor functionality
// to the Trash hierarchy without
// modifying the base class.
package refactor.trashvisitor;
import refactor.trash.*;

interface Visitable {
  // The new method:
  void accept(Visitor v);
} ///:~
```

因为 Visitor 基类里没有任何具体的东西，所以它可以被当作一个接口来创建：

```
//: refactor:trashvisitor:Visitor.java
// The base interface for visitors.
package refactor.trashvisitor;
import refactor.trash.*;

interface Visitor {
  void visit(Aluminum a);
  void visit(Paper p);
  void visit(Glass g);
  void visit(Cardboard c);
} ///:~
```

一个使用反射实现的 Decorator

现在，你可以按照同样于双重分派所使用的方法创建 Aluminum、Paper、Glass 和 Cardboard 的新的子类型，这些子类型要实现 accept( )方法。例如，新的 Visitable Aluminum 看起来就像下面这样子：

```
//: refactor:trashvisitor:VAluminum.java
// Taking the previous approach of creating a
// specialized Aluminum for the visitor pattern.
package refactor.trashvisitor;
import refactor.trash.*;

public class VAluminum extends Aluminum
    implements Visitable {
  public VAluminum(double wt) { super(wt); }
  public void accept(Visitor v) {
    v.visit(this);
  }
} ///:~
```

但是，我们似乎要面临“接口激增（explosion of interfaces）”的问题：基本 Trash、针对双路分派的特殊版本、现在又需要针对 visitor 的更为特殊的版本。当然，这种“接口激增”是由你来控制的——你可以简单地把额外的方法放到 Trash 类里。（译注：此句原文似乎不完整）我们不能忽视有可能使用 Decorator 模式作为替代：看起来应该有可能创建一个 Decorator，用它来包裹一个普通的 Trash 对象，并且创建与 Trash 相同的接口，然后添加额外的 accept( )方法。实际上，它是体现 Decorator 模式价值的一个绝好的例子。

双重分派无论如何都会带来一个问题。因为它依赖于 accept( )和 visit( )的重载（overloading），这似乎需要针对每一个不同版本的 accept( )方法的专门代码。使用 C++模板，完成这个任务是相当简单的（因为模板自动产生针对特定类型的代码），但是 Java 没有类似的机制——至少表面上看起来没有。然而，反射（reflection）允许你在运行时刻决定类型信息，实际上它可以解决许多看起来需要模板来解决的问题（尽管没有模板那么简单）。下面是使用了这个诀窍[13]（trick）的 decorator 类：

```
//: refactor:trashvisitor:VisitableDecorator.java
// A decorator that adapts the generic Trash
// classes to the visitor pattern.
// [ Use a Dynamic Proxy here?? ]
package refactor.trashvisitor;
import refactor.trash.*;
import java.lang.reflect.*;

public class VisitableDecorator
extends Trash implements Visitable {
  private Trash delegate;
  private Method dispatch;
  public VisitableDecorator(Trash t) {
    delegate = t;
    try {
      dispatch = Visitor.class.getMethod (
        "visit", new Class[] { t.getClass() }
      );
    } catch(Exception e) {
      throw new RuntimeException(e);
    }
  }
  public double getValue() {
    return delegate.getValue();
  }
  public double getWeight() {
    return delegate.getWeight();
  }
```

[13] 这是我在 Prague 讲授设计模式的课堂上 Jaroslav Tulach 给出的一个解决方案。

```
  public void accept(Visitor v) {
    try {
      dispatch.invoke(v, new Object[]{delegate});
    } catch(Exception e) {
      throw new RuntimeException(e);
    }
  }
} ///:~
```

[[描述一下反射的使用]]

[[动态代理（Dynamic Proxy）可能也可以应用在这里。]]

我们另外唯一需要的工具是一个新型的 Fillable 适配器，它可以自动装饰（decorates）那些原先从 Trash.dat 创建的对象。但是它自己本身可能也是个 decorator，它可以装饰任何类型的 Fillable 类：

```
//: refactor:trashvisitor:FillableVisitor.java
// Adapter Decorator that adds the visitable
// decorator as the Trash objects are
// being created.
package refactor.trashvisitor;
import refactor.trash.*;
import java.util.*;

public class FillableVisitor
implements Fillable {
  private Fillable f;
  public FillableVisitor(Fillable ff) { f = ff; }
  public void addTrash(Trash t) {
    f.addTrash(new VisitableDecorator(t));
  }
} ///:~
```

现在你可以把它外覆于任何现有的 Fillable 类，或者是任何尚未被创建的新的类型。

程序剩下的部分创建特定的 Visitor 类型并送它们遍历一个 Trash 对象的列表。

```
//: refactor:trashvisitor:TrashVisitor.java
```

```
// The "visitor" pattern with VisitableDecorators.
package refactor.trashvisitor;
import refactor.trash.*;
import java.util.*;
import junit.framework.*;

// Specific group of algorithms packaged
// in each implementation of Visitor:
class PriceVisitor implements Visitor {
  private double alSum; // Aluminum
  private double pSum; // Paper
  private double gSum; // Glass
  private double cSum; // Cardboard
  public void visit(Aluminum al) {
    double v = al.getWeight() * al.getValue();
    System.out.println(
      "value of Aluminum= " + v);
    alSum += v;
  }
  public void visit(Paper p) {
    double v = p.getWeight() * p.getValue();
    System.out.println(
      "value of Paper= " + v);
    pSum += v;
  }
  public void visit(Glass g) {
    double v = g.getWeight() * g.getValue();
    System.out.println(
      "value of Glass= " + v);
    gSum += v;
  }
  public void visit(Cardboard c) {
    double v = c.getWeight() * c.getValue();
    System.out.println(
      "value of Cardboard = " + v);
    cSum += v;
  }
  void total() {
```

```
    System.out.println(
      "Total Aluminum: $" + alSum +
      "\n Total Paper: $" + pSum +
      "\nTotal Glass: $" + gSum +
      "\nTotal Cardboard: $" + cSum +
      "\nTotal: $" +
        (alSum + pSum + gSum + cSum));
  }
}

class WeightVisitor implements Visitor {
  private double alSum; // Aluminum
  private double pSum; // Paper
  private double gSum; // Glass
  private double cSum; // Cardboard
  public void visit(Aluminum al) {
    alSum += al.getWeight();
    System.out.println("weight of Aluminum = "
        + al.getWeight());
  }
  public void visit(Paper p) {
    pSum += p.getWeight();
    System.out.println("weight of Paper = "
        + p.getWeight());
  }
  public void visit(Glass g) {
    gSum += g.getWeight();
    System.out.println("weight of Glass = "
        + g.getWeight());
  }
  public void visit(Cardboard c) {
    cSum += c.getWeight();
    System.out.println("weight of Cardboard = "
        + c.getWeight());
  }
  void total() {
    System.out.println(
      "Total weight Aluminum: " + alSum +
```

```
      "\nTotal weight Paper: " + pSum +
      "\nTotal weight Glass: " + gSum +
      "\nTotal weight Cardboard: " + cSum +
      "\nTotal weight: " + (alSum + pSum + gSum + cSum));

  }
}

public class TrashVisitor extends TestCase  {
  Collection bin = new ArrayList();
  PriceVisitor pv = new PriceVisitor();
  WeightVisitor wv = new WeightVisitor();
  public TrashVisitor() {
    ParseTrash.fillBin("../trash/Trash.dat",
      new FillableVisitor(
        new FillableCollection(bin)));
  }
  public void test() {
    Iterator it = bin.iterator();
    while(it.hasNext()) {
      Visitable v = (Visitable)it.next();
      v.accept(pv);
      v.accept(wv);
    }
    pv.total();
    wv.total();
  }
  public static void main(String args[]) {
    junit.textui.TestRunner.run(TrashVisitor.class);
  }
} ///:~
```

请注意，在 Test( )里可访问性（visitability）是如何通过使用 decorator 简单地创建一个不同种类的 bin 而添加的。还有一点请注意，FillableCollection 适配器在这种情形下表面上看起来是当作一个（ArrayList 的）decorator 来使用的。但是，它完全改变了 ArrayList 的接口，而 Decorator 模式的定义是说，被装饰类（decorated calss）的接口在装饰（decoration）发生以后仍然维持原状。

请注意，客户端代码（Test 类里所展示的那些）的轮廓（shape）又一次（相对于这个问题最初的解决方法）被改变了。现在只有一个单独的废品箱（Trash bin）了。这两个 Visitor 对象被序列里的每个元素所接受，然后它们实施自己的操作。访问者（visitors）保存它们自己的内部数据用以计算总重量和总价格。

最终，除了不可回避地需要把从序列里取出东西转换（cast）成 Trash 对象之外，再没有其它运行时刻类型识别了。而这个转换，实际上也可以通过在 Java 里实现参数化类型而移除。

你可以通过下面的方法区别出这个设计和前面所描述的双重分派解决方案之间的不同之处，在双重分派的解决方案里，只有一个重载过的（overloaded）方法，add( )，在每个子类创建的时候被覆写（overridden）了，而在这里每个重载过的 visit( )方法在 Visitor 的所有子类里都被覆写了。

耦合的更厉害了？

现在有了更多的代码，而且 Trash 体系与 Visitor 体系之间存在勿庸置疑的耦合。但是，在两个类集合的内部又都是高内聚（high cohesion）的：它们各自只做一件事情（Trash 描述 Trash，而 Visitor 描述作用于 Trash 之上的动作（actions）），这是一个好的设计的征兆。当然，在本例中它只在你添加新类型的 Visitor 的时候工作正常，但是当你添加新类型的 Trash 的时候它就会妨碍到你了。

类之间的低耦合与类内部的高内聚毫无疑问是一个重要的设计目标。不过，如果是漫无目的的应用，它可能会妨碍你完成一个更加优雅的设计。看起来有些类之间不可避免地具有某种亲密关系。这些东西经常是成对出现的，或许可以把它们叫做 couplets；例如，容器（container）和迭代器（iterator）。上面的 Trash-Visitor 对看起来像是另一个这种 couplet。

RTTI 果真有害么？

本章的好几个设计都试图去除 RTTI，这可能给你留下 RTTI 是“极其有害”的印象（正如饱受谴责的、可怜而倒霉的 goto 从来就没有被引入 Java 一样）。其实并不是这样的：对 RTTI 的误用才是问题的所在。我们在设计中去除 RTTI 的理由是因为，对这个功能的不恰当的应用妨碍到了扩展性，而我们设定的目标是能够在往系统里添加新类型的时候尽可能少的影响到周围的代码。因为 RTTI 经常被不正确的使用以查找系统里每一个单独的类型，它导致了不可扩展的代码：当你添加一个新类型的时候，你必须查找所有使用了 RTTI 的代码，如果你漏掉一个的话也不会从编译器那里得到任何帮助。

但是，RTTI 并不是自动地产生不可扩展的代码。让我们再次温习一下 trash recycler。这一次，我们会介绍一个新的工具，我把这个工具叫做 TypeMap。它包括一个持有 ArrayLists 的 HashMap，但是接口很简单：你可以使用 add( )添加一个新对象

以及使用 get( )取出一个 ArrayList，这个 ArrayList 含有某一特定类型的所有对象。HashMap 的键值（key）是对应 ArrayList 所包含的元素的类型。这个设计的美妙之处在于，TypeMap 在遇到新的类型的时候动态地添加新的 pair，所以无论什么时候你往系统里添加一个新的类型（即使你在运行时刻添加一个新的类型），它都可以适应。

我们的例子会再一次根据 refactor.Trash 包里的 Trash 类型来构建（以前使用 Trash.dat 文件可以不用做改动）：

```
//: refactor:dynatrash:DynaTrash.java
// Using a Map of Lists and RTTI to automatically sort
// trash into ArrayLists. This solution, despite the
// use of RTTI, is extensible.
package refactor.dynatrash;
import refactor.trash.*;
import java.util.*;
import junit.framework.*;

// Generic TypeMap works in any situation:
class TypeMap {
  private Map t = new HashMap();
  public void add(Object o) {
    Class type = o.getClass();
    if(t.containsKey(type))
      ((List)t.get(type)).add(o);
    else {
      List v = new ArrayList();
      v.add(o);
      t.put(type,v);
    }
  }
  public List get(Class type) {
    return (List)t.get(type);
  }
  public Iterator keys() {
    return t.keySet().iterator();
  }
}
```

```
// Adapter class to allow callbacks
// from ParseTrash.fillBin():
class TypeMapAdapter implements Fillable {
  TypeMap map;
  public TypeMapAdapter(TypeMap tm) { map = tm; }
  public void addTrash(Trash t) { map.add(t); }
}

public class DynaTrash extends TestCase  {
  TypeMap bin = new TypeMap();
  public DynaTrash() {
    ParseTrash.fillBin("../trash/Trash.dat",
      new TypeMapAdapter(bin));
  }
  public void test() {
    Iterator keys = bin.keys();
    while(keys.hasNext())
      Trash.sumValue(
        bin.get((Class)keys.next()).iterator());
  }
  public static void main(String args[]) {
    junit.textui.TestRunner.run(DynaTrash.class);
  }
} ///:~
```

尽管很强大，TypeMap 的定义却是很简单的。它包含一个 HashMap，add( )方法会完成大部分的工作。当你通过 add( )添加一个新对象，关于那种类型的类对象的引用就被提取出来了。它会被用作一个键值来决定是否持有这种类型的对象的 ArrayList 已经存在于 HashMap 中。如果已经存在，那个 ArrayList 会被提取出来，然后把这个对象加入那个 ArrayList。如果不存在，会把这个类对象（Class object）和一个新的 ArrayList 作为一个 key-value 对添加（到 HashMap）。

你可以通过 keys( )方法得到所有的类对象（Class objects），然后使用每个类对象通过 get( )方法取出对应的 ArrayList。这就是关于它的一切。

filler( )方法很有意思，因为它利用（takes advantage of）了 ParseTrash.fillBin( )的设计，它并不仅仅试图填充一个 ArrayList，而且它会试着填充任何实现了 Fillable 接口和 addTrash( )方法的对象。Filler( )所需要做的就是

返回一个关于实现了 Fillable 的接口的引用，然后这个引用可以被当作 fillBin( )的一个参数，像这样:

```
ParseTrash.fillBin("Trash.dat", bin.filler());
```

为了产生这个引用，我们使用了一个匿名的内部类（《Thinking in Java》第二版第 8 章有述）。你根本不需要使用一个有名字的类来实现 Fillable 接口，你只需要关于那种类的一个对象的引用，因此这里使用匿名内部类是合适的。

这个设计有意思的一点是，尽管创建它的初衷不是用它来处理分拣（sorting），但是每次往 bin 里面插入一个 Trash 对象的时候，fillBin( )都会实施一个分拣操作。

DynaTrash 类的绝大部分都和前面的例子很类似。这一次，并不是把新类型的 Trash 对象放入一个元素类型是 ArrayList 的 bin，现在 bin 的类型是 TypeMap，所以当 trash 被扔进 bin 的时候它立刻就会被 TypeMap 的内部分拣机制所分拣。遍历（stepping through）TypeMap 并且针对每个单独的 ArrayList 实施操作变成了一件简单的事情。

如你所见，往系统里添加一个新类型根本不会影响到这些代码，TypeMap 内部的那些代码是完全独立的。这当然是这个问题最小的解决方案，似乎（arguably）也是最优雅的解决方案。它确实大量依赖于 RTTI，但是请注意，HashMap 里面的每一个 key-value 对只查找一种类型。此外，当添加一个新类型的时候，你不可能“忘记”给系统添加合适的代码，因为你跟本不需要添加任何代码。

总结

最后得到一个像 TrashVisitor.java 这样比早先的设计多出许多的码的设计可能乍看起来似乎是吃力不讨好的。它的代价都花费在你通过不同的设计所试图解决的问题上了。设计模式通常力争把变化的东西从那些保持不变的东西里分离出来。“变化的东西”可以指很多不同种类的变化。出现变化可能是因为程序被用到了新的环境中或者是因为现有环境的某些东西变化了（可能是：“用户想要往当前屏幕上的图表中添加一个新的形状（shape）”）。或者，像现在这个例子，变化的东西可能是代码体的演化。正如废品分拣这个例子的前一个版本所强调的是往系统里添加新类型（尽可能的容易），TrashVisitor.java 允许你很容易地添加新功能而不用打扰 Trash 类体系。TrashVisitor.java 里有了更多的代码，但是给 Visitor 添加新的功能就毫不费力了。如果添加新功能是经常发生的，那么为了更方便的添加这些新功能，额外的努力和额外代码就是值得的。

要发现一系列的变化并不是一件轻而易举的事情；分析师通常在看到程序设计草稿之前是很难发现这些变化的。必要的信息可能直到项目的较晚阶段才会出现：有时候只有在设计或者实现阶段你才能发现一个深层次的或者微妙的需求。对于添加新类

型来说（这是大多数“recycle”例子的焦点所在）你可能只有在维护阶段才会意识到你会需要一个特定的继承体系，然后开始扩展这个系统。

通过学习设计模式你学到的最重要的一点可能是，相对于本书一直鼓吹的东西的一个观念上的完全改变（an about-face from what has been promoted so far in this book）。也就是：“所有 OOP 都是关于多态的。”这个论断可能会导致像“拿锤子的两岁小孩（two-year-old with a hammer）”那样把所有东西都当成钉子。换种说法，想要“得到”多态是很难的，而且即使办到了，你又得试着把你所有的设计转化（cast）成某个特定的模子。

设计模式认为 OOP 并不仅仅是关于多态的。它是关于“把变化的东西从那些保持不变的东西里分离出来”的。多态是达到这个目的的一种特别重要的方法，实践证明如果编程语言直接支持多态（你就用不着自己写了，自己写的代价可能是让人望而却步的）会很有用。但是设计模式通常展示了达到基本目标的其它的方法，一旦你开始留意这些你就会开始找寻更有创造性的设计了。

因为《设计模式》一书的面世以及其所造成的如此深远的影响，大家一直都在搜寻其它的模式。随着时间的推移，你应该会看到更多的模式涌现出来。下面是 Jim Coplien（他在 C++领域久负盛名 http://www.bell-labs.com/~cope，并且是模式运动的主要倡导者之一）推荐的一些站点：

http://st-www.cs.uiuc.edu/users/patterns

http://c2.com/cgi/wiki

http://c2.com/ppr

http://www.bell-labs.com/people/cope/Patterns/Process/index.html

http://www.bell-labs.com/cgi-user/OrgPatterns/OrgPatterns

http://st-www.cs.uiuc.edu/cgi-bin/wikic/wikic

http://www.cs.wustl.edu/~schmidt/patterns.html

http://www.espinc.com/patterns/overview.html

还有一点需要告诉你，每年有一个关于设计模式的会议，叫做 PLOP，这个会议有一个出版的论文集，第三本论文集是 1997 年下半年出版（所有这些论文集都由 Addison-Wesley 出版）。

练习

1. 添加一个 Plastic 类到 TrashVisitor.java。
2. 添加一个 Plastic 类到 DynaTrash.java。
3. 写一个像 VisitableDecorator 那样的 decorator，但是要针对多重分派那个例子，还要写一个类似于为了 VisitableDecorator 所创建的那种“adapter decorator”类。构建那个例子剩下的部分让它能够工作。

项目

这里是一些留待你解决的更具挑战性的项目。『其中的某一些可能以后会作为本书的例子，所以可能会从这里拿掉』

老鼠和迷宫

首先，创建一个黑板（cite reference?）对象用来记录信息。在这个特殊的黑板上画迷宫，并且用它来显示由老鼠探测出来的迷宫的结构信息。

现在创建一个真正的迷宫，这个对象只暴露它自身很少一部分的信息——针对给定的坐标，它会告诉你紧挨着给它四周是墙壁还是空地，但是不会有其它更多的信息。对于新手，可以从一个文本文件读入迷宫（数据），但是可以考虑从因特网上找一个构造迷宫的算法。无论如何，最后的结果都应该是这么一个对象，这个对象可以根据给定的坐标找出它周围的墙壁和空地。还有一点，它还必须提供这个迷宫的入口以供查询。

最后，创建迷宫探路的老鼠 Rat 类。每个老鼠都可以与黑板和迷宫打交道，它向黑板报告自己的当前信息，并且根据自己所处的位置向迷宫请求新的信息。不管怎样，每次老鼠碰到迷宫分支点需要做出决策的时候，它就创建一个新的线程探测每一个分支。每个老鼠由它自己的线程驱动，当它走近死胡同以后，它会先向黑板报告自己探测的最终结果，然后它会结束自己所在的线程。

最终目标是绘出迷宫的完整地图，但是你必须得决定最后的结束条件是顺其自然找到的还是由黑板来判定。

下面是 Jeremy Meyer 写的一个实现的例子：

```
//: projects:Maze.java

package projects;

import java.util.*;
import java.io.*;
import java.awt.*;

public class Maze extends Canvas {
  private Vector lines; // a line is a char array
  private int width = -1;
  private int height = -1;
  public static void main (String [] args)
```

```
throws IOException {
  if (args.length < 1) {
    System.out.println("Enter filename");
    System.exit(0);
  }
  Maze m = new Maze();
  m.load(args[0]);
  Frame f = new Frame();
  f.setSize(m.width*20, m.height*20);
  f.add(m);
  Rat r = new Rat(m, 0, 0);
  f.setVisible(true);
}
public Maze() {
  lines = new Vector();
  setBackground(Color.lightGray);
}
synchronized public boolean
isEmptyXY(int x, int y) {
  if (x < 0) x += width;
  if (y < 0) y += height;
  // Use mod arithmetic to bring rat in line:
  byte[] by =
    (byte[])(lines.elementAt(y%height));
  return by[x%width]==' ';
}
synchronized public void
setXY(int x, int y, byte newByte) {
  if (x < 0) x += width;
  if (y < 0) y += height;
  byte[] by =
    (byte[])(lines.elementAt(y%height));
  by[x%width] = newByte;
  repaint();
}
public void
load(String filename) throws IOException {
  String currentLine = null;
```

```
    BufferedReader br = new BufferedReader(
      new FileReader(filename));
    for(currentLine = br.readLine();
        currentLine != null;
        currentLine = br.readLine())  {
      lines.addElement(currentLine.getBytes());
      if(width < 0 ||
         currentLine.getBytes().length > width)
        width = currentLine.getBytes().length;
    }
    height = lines.size();
    br.close();
  }
  public void update(Graphics g) { paint(g); }
  public void paint (Graphics g) {
    int canvasHeight = this.getBounds().height;
    int canvasWidth  = this.getBounds().width;
    if (height < 1 || width < 1)
      return; // nothing to do
    int width =
      ((byte[])(lines.elementAt(0))).length;
    for (int y = 0; y < lines.size(); y++) {
      byte[] b;
      b = (byte[])(lines.elementAt(y));
      for (int x = 0; x < width; x++) {
        switch(b[x]) {
          case ' ': // empty part of maze
            g.setColor(Color.lightGray);
            g.fillRect(
              x*(canvasWidth/width),
              y*(canvasHeight/height),
              canvasWidth/width,
              canvasHeight/height);
            break;
          case '*':     // a wall
            g.setColor(Color.darkGray);
            g.fillRect(
              x*(canvasWidth/width),
```

```
              y*(canvasHeight/height),
              (canvasWidth/width)-1,
              (canvasHeight/height)-1);
            break;
          default:      // must be rat
            g.setColor(Color.red);
            g.fillOval(x*(canvasWidth/width),
            y*(canvasHeight/height),
            canvasWidth/width,
            canvasHeight/height);
            break;
        }
      }
    }
  }
} ///:~

//: projects:Rat.java
package projects;

public class Rat {
  static int ratCount = 0;
  private Maze prison;
  private int vertDir = 0;
  private int horizDir = 0;
  private int x,y;
  private int myRatNo = 0;
  public Rat(Maze maze, int xStart, int yStart) {
    myRatNo = ratCount++;
    System.out.println("Rat no." + myRatNo +
      " ready to scurry.");
    prison = maze;
    x = xStart;
    y = yStart;
    prison.setXY(x,y, (byte)'R');
    new Thread() {
      public void run(){ scurry(); }
    }.start();
```

```
  }
  public void scurry() {
    // Try and maintain direction if possible.
    // Horizontal backward
    boolean ratCanMove = true;
    while(ratCanMove) {
      ratCanMove = false;
      // South
      if (prison.isEmptyXY(x, y + 1)) {
        vertDir = 1; horizDir = 0;
        ratCanMove = true;
      }
      // North
      if (prison.isEmptyXY(x, y - 1))
        if (ratCanMove)
          new Rat(prison, x, y-1);
          // Rat can move already, so give
          // this choice to the next rat.
        else {
          vertDir = -1; horizDir = 0;
          ratCanMove = true;
        }
      // West
      if (prison.isEmptyXY(x-1, y))
        if (ratCanMove)
          new Rat(prison, x-1, y);
          // Rat can move already, so give
          // this choice to the next rat.
        else {
          vertDir = 0; horizDir = -1;
          ratCanMove = true;
        }
      // East
      if (prison.isEmptyXY(x+1, y))
        if (ratCanMove)
          new Rat(prison, x+1, y);
          // Rat can move already, so give
          // this choice to the next rat.
```

```
        else {
          vertDir = 0; horizDir = 1;
          ratCanMove = true;
        }
      if (ratCanMove) { // Move original rat.
        x += horizDir;
        y += vertDir;
        prison.setXY(x,y,(byte)'R');
      }  // If not then the rat will die.
      try {
        Thread.sleep(2000);
      } catch(InterruptedException e) {
        throw new RuntimeException(e);
      }
    }
    System.out.println("Rat no." + myRatNo +
      " can't move..dying..aarrgggh.");
  }
} ///:~
```

迷宫初始化文件：

```
//:! projects:Amaze.txt
* ** * * ** *
*** * ******* * ****
*** ***
***** ********** *****
* * * * ** ** * * * ** *
* * * * ** * * * * **
* ** * **
* ** * ** * ** * **
*** * *** ***** * *** **
* * * * * *
* ** * * * ** * *
///:~
```

其它关于迷宫的资源

关于创建迷宫算法的讨论以及实现它们的 java 源代码：

http://www.mazeworks.com/mazegen/mazegen.htm

关于碰撞检测算法的讨论和其它针对自主物理对象（autonomous physical objects）单个或群体运动行为的讨论：

http://www.red3d.com/cwr/steer/

XML 修饰器（XML Decorator）

针对 I/O 读写器写一对修饰器（decorators）用来对 XML 编码（写修饰器）和解码（读修饰器）。

附录：工具

包括编译本书（代码）用到的一些工具。其中一些可能是临时性的，如果以后基准代码移到 CVS，它们可能会从这里消失。

Ant 扩展

Ant 提供扩展 API，你可以利用它们用 java 创建你自己的任务。你可以从 Ant 的官方文档或者已出版的关于 Ant 的书籍中找到详尽的信息。

作为另外一种选择，你可以简单的写一个 java 程序，并且在 Ant 里调用；使用这种方法，你就用不着学习扩展 API 了。例如，为了编译本书的代码，我们需要确定用户使用的 Java 版本是 JDK 1.3 或者更高，所以就有了下面的程序：

```
//: com:bruceeckel:tools:CheckVersion.java
// {RunByHand}
package com.bruceeckel.tools;

public class CheckVersion {
  public static void main(String[] args) {
    String version = System.getProperty("java.version");
    char minor = version.charAt(2);
    char point = version.charAt(4);
    if(minor < '3' || point < '0')
      throw new RuntimeException("JDK 1.3.0 or higher " +
        "is required to run the examples in this book.");
    System.out.println("JDK version "+ version + " found");
  }
} ///:~
```

这个程序只是简单的使用 System.getProperty( )来获取 java 版本，如果版本号小于 1.3 就抛出一个异常。当 Ant 碰到这个异常的时候，它就会停下来。这么一来，在想要检测版本号的时候，你就可以在任何 buildfile 里加上下面几行脚本：

```
    <java
      taskname="CheckVersion"
      classname="com.bruceeckel.tools.CheckVersion"
      classpath="${basedir}"
      fork="true"
      failonerror="true"
    />
```

用这种方法添加新的工具，你可以很快的完成从编码到测试。如果它们被证明是合理的话，你可以再花些力气写个 Ant 的扩展程序。

Array utilities

尽管有用，（java 自带的）Arrays 类还是功能不够全面。例如，如果能够直接打印一个数组的所有元素，而不是每次必须手写 for 循环才能完成，那就再好不过了。你会看到，fill( )方法只能把一个值放到数组里面，假如说你想用一组随即产生的数字来填充一个数组，fill( )方法就无能为力了。

这样一来，写一些额外的实用程序（utilities）作为 Arrays 类的补充就显的有意义了，方便起见，我把它们放到了 com.bruceeckel.util 这个 package 里。这些小程序可以打印一个任意类型的数组，也可以填充由你所定义的一个叫做 generator 的东东产生的值或者对象。

因为（实用程序的）代码需要支持每一种基元类型（primitive type）和 Object 类型，所以就产生了许多近乎重复的代码[14]。例如，每种类型都需要一个“generator”接口，因为每一种情况下 next( )方法的返回类型都是不同的。

```
//: com:bruceeckel:util:Generator.java
package com.bruceeckel.util;
public interface Generator { Object next(); } ///:~

//: com:bruceeckel:util:BooleanGenerator.java
package com.bruceeckel.util;
public interface BooleanGenerator { boolean next(); } ///:~

//: com:bruceeckel:util:ByteGenerator.java
package com.bruceeckel.util;
public interface ByteGenerator { byte next(); } ///:~

//: com:bruceeckel:util:CharGenerator.java
package com.bruceeckel.util;
public interface CharGenerator { char next(); } ///:~

//: com:bruceeckel:util:ShortGenerator.java
package com.bruceeckel.util;
```

[14] C++程序员会发现这些代码可以通过使用默认参数（default arguments）和模板（templates）大大压缩。Python 程序员会发现，对于 Python 语言来说，这里使用的整个库很大程度上都是不必要的。

```
public interface ShortGenerator { short next(); } ///:~

//: com:bruceeckel:util:IntGenerator.java
package com.bruceeckel.util;
public interface IntGenerator { int next(); } ///:~

//: com:bruceeckel:util:LongGenerator.java
package com.bruceeckel.util;
public interface LongGenerator { long next(); } ///:~

//: com:bruceeckel:util:FloatGenerator.java
package com.bruceeckel.util;
public interface FloatGenerator { float next(); } ///:~

//: com:bruceeckel:util:DoubleGenerator.java
package com.bruceeckel.util;
public interface DoubleGenerator { double next(); } ///:~
```

Array2 包括一系列为每种类型重载过的 toString()方法。这些方法使你可以很容易的打印一个数组。ToString( )的代码用了 StringBuffer 对象而不是 String 对象。这是出于效率的原因；当你是在某个可能会被多次调用的方法里装配一个字符串，更明智的做法是采用效率更高的 StringBuffer 而不是使用起来比较方便的 String 类型的那些操作。在这里，创建 StringBuffger 的时候给它一个初始值，然后把 String 对象追加到它的后面。最后，把 result 对象转换成一个 String 对象作为函数的返回值。

```
//: com:bruceeckel:util:Arrays2.java
// A supplement to java.util.Arrays, to provide additional
// useful functionality when working with arrays. Allows
// any array to be converted to a String, and to be filled
// via a user-defined "generator" object.

package com.bruceeckel.util;

import java.util.*;

public class Arrays2 {
  public static String toString(boolean[] a) {
```

```
    StringBuffer result = new StringBuffer("[");
    for(int i = 0; i < a.length; i++) {
      result.append(a[i]);
      if(i < a.length - 1)
        result.append(", ");
    }
    result.append("]");
    return result.toString();
  }
  public static String toString(byte[] a) {
    StringBuffer result = new StringBuffer("[");
    for(int i = 0; i < a.length; i++) {
      result.append(a[i]);
      if(i < a.length - 1)
        result.append(", ");
    }
    result.append("]");
    return result.toString();
  }
  public static String toString(char[] a) {
    StringBuffer result = new StringBuffer("[");
    for(int i = 0; i < a.length; i++) {
      result.append(a[i]);
      if(i < a.length - 1)
        result.append(", ");
    }
    result.append("]");
    return result.toString();
  }
  public static String toString(short[] a) {
    StringBuffer result = new StringBuffer("[");
    for(int i = 0; i < a.length; i++) {
      result.append(a[i]);
      if(i < a.length - 1)
        result.append(", ");
    }
    result.append("]");
    return result.toString();
```

```
  }
  public static String toString(int[] a) {
    StringBuffer result = new StringBuffer("[");
    for(int i = 0; i < a.length; i++) {
      result.append(a[i]);
      if(i < a.length - 1)
        result.append(", ");
    }
    result.append("]");
    return result.toString();
  }
  public static String toString(long[] a) {
    StringBuffer result = new StringBuffer("[");
    for(int i = 0; i < a.length; i++) {
      result.append(a[i]);
      if(i < a.length - 1)
        result.append(", ");
    }
    result.append("]");
    return result.toString();
  }
  public static String toString(float[] a) {
    StringBuffer result = new StringBuffer("[");
    for(int i = 0; i < a.length; i++) {
      result.append(a[i]);
      if(i < a.length - 1)
        result.append(", ");
    }
    result.append("]");
    return result.toString();
  }
  public static String toString(double[] a) {
    StringBuffer result = new StringBuffer("[");
    for(int i = 0; i < a.length; i++) {
      result.append(a[i]);
      if(i < a.length - 1)
        result.append(", ");
    }
```

```
    result.append("]");
    return result.toString();
  }
  // Fill an array using a generator:
  public static void fill(Object[] a, Generator gen) {
    fill(a, 0, a.length, gen);
  }
  public static void
  fill(Object[] a, int from, int to, Generator gen) {
    for(int i = from; i < to; i++)
      a[i] = gen.next();
  }
  public static void
  fill(boolean[] a, BooleanGenerator gen) {
      fill(a, 0, a.length, gen);
  }
  public static void
  fill(boolean[] a, int from, int to,BooleanGenerator gen){
    for(int i = from; i < to; i++)
      a[i] = gen.next();
  }
  public static void fill(byte[] a, ByteGenerator gen) {
    fill(a, 0, a.length, gen);
  }
  public static void
  fill(byte[] a, int from, int to, ByteGenerator gen) {
    for(int i = from; i < to; i++)
      a[i] = gen.next();
  }
  public static void fill(char[] a, CharGenerator gen) {
    fill(a, 0, a.length, gen);
  }
  public static void
  fill(char[] a, int from, int to, CharGenerator gen) {
    for(int i = from; i < to; i++)
      a[i] = gen.next();
  }
  public static void fill(short[] a, ShortGenerator gen) {
```

```
    fill(a, 0, a.length, gen);
  }
  public static void
  fill(short[] a, int from, int to, ShortGenerator gen) {
    for(int i = from; i < to; i++)
      a[i] = gen.next();
  }
  public static void fill(int[] a, IntGenerator gen) {
      fill(a, 0, a.length, gen);
  }
  public static void
  fill(int[] a, int from, int to, IntGenerator gen) {
    for(int i = from; i < to; i++)
      a[i] = gen.next();
  }
  public static void fill(long[] a, LongGenerator gen) {
    fill(a, 0, a.length, gen);
  }
  public static void
  fill(long[] a, int from, int to, LongGenerator gen) {
    for(int i = from; i < to; i++)
      a[i] = gen.next();
  }
  public static void fill(float[] a, FloatGenerator gen) {
    fill(a, 0, a.length, gen);
  }
  public static void
  fill(float[] a, int from, int to, FloatGenerator gen) {
    for(int i = from; i < to; i++)
      a[i] = gen.next();
  }
  public static void fill(double[] a, DoubleGenerator gen){
    fill(a, 0, a.length, gen);
  }
  public static void
  fill(double[] a, int from, int to, DoubleGenerator gen) {
    for(int i = from; i < to; i++)
      a[i] = gen.next();
```

```
}
private static Random r = new Random();
public static class
RandBooleanGenerator implements BooleanGenerator {
  public boolean next() { return r.nextBoolean(); }
}
public static class
RandByteGenerator implements ByteGenerator {
  public byte next() { return (byte)r.nextInt(); }
}
private static String ssource =
  "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz";
private static char[] src = ssource.toCharArray();
public static class
RandCharGenerator implements CharGenerator {
  public char next() {
    return src[r.nextInt(src.length)];
  }
}
public static class
RandStringGenerator implements Generator {
  private int len;
  private RandCharGenerator cg = new RandCharGenerator();
  public RandStringGenerator(int length) {
    len = length;
  }
  public Object next() {
    char[] buf = new char[len];
    for(int i = 0; i < len; i++)
      buf[i] = cg.next();
    return new String(buf);
  }
}
public static class
RandShortGenerator implements ShortGenerator {
  public short next() { return (short)r.nextInt(); }
}
```

```
  public static class
  RandIntGenerator implements IntGenerator {
    private int mod = 10000;
    public RandIntGenerator() {}
    public RandIntGenerator(int modulo) { mod = modulo; }
    public int next() { return r.nextInt(mod); }
  }
  public static class
  RandLongGenerator implements LongGenerator {
    public long next() { return r.nextLong(); }
  }
  public static class
  RandFloatGenerator implements FloatGenerator {
    public float next() { return r.nextFloat(); }
  }
  public static class
  RandDoubleGenerator implements DoubleGenerator {
    public double next() {return r.nextDouble();}
  }
} ///:~
```

为了使用 generator 对象填充一个数组的所有元素，fill( )方法使用一个合适的 generator 接口，这个接口的 next( )方法以某种方式产生一个特定类型的对象（这取决于这个接口是如何实现的）。fill( )方法只是简单的调用 next( )方法直到预期的范围都被填充。现在你就可以通过实现合适的接口自己来创建任意的 generator 并通过调用 fill( )方法来使用你自己的 generator。

随机数据发生器（Random data generators）对于测试来说是非常有用的，所以就有了一系列的内部类（inner classes）用来实现所有基本类型的 generator 接口，此外还有一个 String 发生器用以代表 Object 类型。你会看到随机字符串发生器（RandStringGenerator ）使用了随机字符发生器（RandCharGenerator）来填充一个字符数组，然后这个数组会被转换成一个 String。数组的大小是由（RandStringGenerator ）的构造函数参数决定的。

如果需要产生的数字不是非常的大，RandIntGenerator 在默认情况下会以 10,000 为系数（modulus）产生随机数，但是重载的构造函数允许你选择更小的值作为系数。