Thinking in Patterns by Bruce Eckel

Revision 0.9, 5-20-2003 (This version contains the material that will be used in the Crested Butte seminar; see http://www.mindview.net/Seminars/ThinkingInPatterns/)

Please note this document is under development and incomplete. Updates to this document can be found at http://www.Mindview.net

Best viewed with Mozilla! (free at www.Mozilla.org) (Even though this document was created with MS Word, IE6 garbles lines with superscripts on them. Mozilla seems to do a much better job).

_____

Note: This document requires the installation of the fonts Georgia, Verdana and Andale Mono (code font) for proper viewing. These can be found at:

http://sourceforge.net/project/showfiles.php?group_id=34153&release_id=105355

Modifications in Revision 0.9:

- Prose has still had little/no work. My current goal is to get the structure and examples worked out so that the seminar works well. Once it has been proven in the seminars, then I will spend time on the prose.

- Added proxy:PoolManager.java to make a more generic/customizeable Pool Manager, and modified proxy:ConnectionPoolProxyDemo.java accordingly [[Still need to decide what to return when you run out of objects in the pool]]

- Changed PoolManager.java to use an ArrayList (and thus does not require a fixed size at initialization)

- Added KissingPrincess.java to State description, as a motivational example for the pattern

- Added a simple Flyweight example

- Simplified the enumeration in PaperScissorsRock.java

Modifications in Revision 0.8:

- Changed Bridge example to improve clarity.

- Removed superscripts for better viewing with IE (see note above)

Modifications in Revision 0.7:

- **NOTE primary changes have been made to structure of book and code examples, but not to prose. Prose can be considered to be mostly a mess in this revision.**

- Complete reorganization under headings that attempt to describe the problems you are trying to solve with a pattern.

- Addition of placeholders for the remainder of the GoF patterns

- Addition of "Simplifying Idioms" section and examples

- Addition of Builder section and examples

- Removed unit-testing chapter; replaced with reference to "new" JUnit (which uses reflection)

- (4-30-2003) Added Ant build.xml files, and support files from TIJ necessary to do a full standalone build. You should be able to type "ant" from the code root directory and get a successful build.

- Dramatically simplified chainofresponsibility:FindMinima.java

- Added object pool/connection pool examples

- Refactored small things in many examples

- Some exercises may have been left behind when patterns were moved.

- For simplicity, saved from Word into a single HTML document, using "filtered" version to remove Office stuff. Seems to work pretty well; checked it with both IE and Mozilla (actually seems to work better on Mozilla than on IE!).

TODO:

- Reconfigure for new backtalk system

- Replace references to TIJ2 with TIJ3

Thinking

in

Patterns

Problem-Solving Techniques using Java

Bruce Eckel

President, MindView, Inc.

Contents

Preface

The material in this book has been developed in conjunction with a seminar that I have given for several years, mostly with Bill Venners. Bill and I have given many iterations of this seminar and we've changed it

many times over the years as we both have learned more about patterns and about giving the seminar.

In the process we've both produced more than enough information for us each to have our own seminars, an urge that we've both strongly resisted because we have so much fun giving the seminar together. We've given the seminar in numerous places in the US, as well as in Prague (where we try to have a mini-conference every Spring together with a number of other seminars). We've also given it as an on-site seminar.

A great deal of appreciation goes to the people who have participated in these seminars over the years, as they have helped me work through these ideas and to refine them. I hope to be able to continue to form and develop these kinds of ideas through this book and seminar for many years to come.

This book will not stop here, either. After much pondering, I've realized that I want *Thinking in Python* to be, initially, a translation of this book rather than an introduction to Python (there are already plenty of fine introductions to that superb language). I find this prospect to be much more exciting than the idea of struggling through another language tutorial (my apologies to those who were hoping for that).

Introduction

This is a book about design that I have been working on for years, basically ever since I first started trying to read *Design Patterns* (Gamma, Helm, Johnson & Vlissides, Addison-Wesley, 1995), commonly referred to as the *Gang of Four*[1] or just GoF).

There is a chapter on design patterns in the first edition of *Thinking in C++*, which has evolved in Volume 2 of the second edition of *Thinking in C++*, and you'll also find a chapter on patterns in the first edition of

Thinking in Java (I took it out of the second edition because that book was getting too big, and also because I had decided to write this book).

This is not an introductory book. I am assuming that you have worked your way through *Thinking in Java* or an equivalent text before coming to this book.

In addition, I assume you have more than just a grasp of the syntax of Java. You should have a good understanding of objects and what they're about, including polymorphism. Again, these are topics covered in *Thinking in Java*.

On the other hand, by going through this book you're going to learn a *lot* about object-oriented programming by seeing objects used in many different situations. If your knowledge of objects is rudimentary, it will get much stronger in the process of understanding the designs in this book.

The Y2K syndrome

In a book that has "problem-solving techniques" in its subtitle, it's worth mentioning one of the biggest pitfalls in programming: premature optimization. Every time I bring this concept forward, virtually everyone agrees to it. Also, everyone seems to reserve in their own mind a special case "except for this thing that I happen to know is a particular problem."

The reason I call this the Y2K syndrome has to do with that special knowledge. Computers are a mystery to most people, so when someone announced that those silly computer programmers had forgotten to put in enough digits to hold dates past the year 1999, then suddenly everyone became a computer expert – "these things aren't so difficult after all, if I can see such an obvious problem." For example, my background was originally in computer engineering, and I started out by programming embedded systems. As a result, I know that many embedded systems have no idea what the date or time is, and even if they do that data often isn't used in any important calculations. And yet I was told in no uncertain terms that all the embedded systems were going to crash on January 1, 2000. As far as I can tell the only memory that was lost on that particular date was that of the people who were predicting doom – it's as if they had never said any of that stuff.

The point is that it's very easy to fall into a habit of thinking that the particular algorithm or piece of code that you happen to partly or thoroughly understand is naturally going to be the bottleneck in your

system, simply because you can imagine what's going on in that piece of code and so you think that it must somehow be much less efficient than all the other pieces of code that you don't know about. But unless you've run actual tests, typically with a profiler, you can't really know what's going on. And even if you are right, that a piece of code is very inefficient, remember that most programs spend something like 90% of their time in less than 10% of the code in the program, so unless the piece of code you're thinking about happens to fall into that 10% it isn't going to be important.

"We should forget about small efficiencies, say about 97% of the time: Premature optimization is the root of all evil."—Donald Knuth

Context and composition

One of the terms you will see used over and over in design patterns literature is *context*. In fact, one common definition of a design pattern is "a solution to a problem in a context." The GoF patterns often have a "context object" that the client programmer interacts with. At one point it occurred to me that such objects seemed to dominate the landscape of many patterns, and so I began asking what they were about.

The context object often acts as a little façade to hide the complexity of the rest of the pattern, and in addition it will often be the controller that manages the operation of the pattern. Initially, it seemed to me that these were not really essential to the implementation, use and understanding of the pattern. However, I remembered one of the more dramatic statements made in the GoF: "prefer composition to inheritance." The context object allows you to use the pattern in a composition, and that may be it's primary value.

A word about checked exceptions

1) The great value of exceptions is the unification of error reporting: a standard mechanism by which to report errors, rather than the popourri of ignorable approaches that we had in C (and thus, C++, which only adds exceptions to the mix, and doesn't make it the exclusive approach). The big advantage Java has over C++ is that exceptions are the only way to report errors.

2) "Ignorable" in the previous paragraph is the other issue. The theory is that if the compiler forces the programmer to either handle the exception or pass it on in an exception specification, then the programmer's attention will always be brought back to the possibility of errors and they will thus properly take care of them. I think the problem is that this is an untested assumption we're making as language designers that falls into the field of psychology. My theory is that when someone is trying to do something and you are constantly prodding them with annoyances, they will use the quickest device available to make those annoyances go away so they can get their thing done, perhaps assuming they'll go back and take out the device later. I discovered I had done this in the first edition of Thinking in Java:

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

And then more or less forgot it until the rewrite. How many people thought this was a good example and followed it? Martin Fowler began seeing the same kind of code, and realized people were stubbing out exceptions and then they were disappearing. The overhead of checked exceptions was having the opposite effect of what was intended, something that can happen when you experiment (and I now believe that checked exceptions were an experiment based on what someone thought was a good idea, and which I believed was a good idea until recently).

When I started using Python, all the exceptions appeared, none were accidentally "disappeared." If you *want* to catch an exception, you can, but you aren't forced to write reams of code all the time just to be passing the exceptions around. They go up to where you want to catch them, or they go all the way out if you forget (and thus they remind you) but they don't vanish, which is the worst of all possible cases. I now believe that checked exceptions encourage people to make them vanish. Plus they make much less readable code.

In the end, I think we must realize the experimental nature of exceptions and look at them carefully before assuming that everything about exceptions in Java are good. I believe that having a single mechanism for handling errors is excellent, and I believe that using a separate channel (the exception handling mechanism) for moving the exceptions around is good. But I do remember one of the early arguments for exception handling in C++ was that it would allow the programmer to separate the sections of code where you just wanted to get work done from the sections where you handled errors, and it seems to me that checked exceptions do

not do this; instead, they tend to intrude (a lot) into your "normal working code" and thus are a step backwards. My experience with Python exceptions supports this, and unless I get turned around on this issue I intend to put a lot more **RuntimeException**s into my Java code.

The pattern concept

"Design patterns help you learn from others' successes instead of your own failures[2]."

Probably the most important step forward in object-oriented design is the "design patterns" movement, chronicled in *Design Patterns (ibid)*[3]. That book shows 23 different solutions to particular classes of problems. In this book, the basic concepts of design patterns will be introduced along with examples. This should whet your appetite to read *Design Patterns* by Gamma, et. al., a source of what has now become an essential, almost mandatory, vocabulary for OOP programmers.

The latter part of this book contains an example of the design evolution process, starting with an initial solution and moving through the logic and process of evolving the solution to more appropriate designs. The program shown (a trash sorting simulation) has evolved over time, and you can look at that evolution as a prototype for the way your own design can start as an adequate solution to a particular problem and evolve into a flexible approach to a class of problems.

What is a pattern?

Initially, you can think of a pattern as an especially clever and insightful way of solving a particular class of problems. That is, it looks like a lot of people have worked out all the angles of a problem and have come up with the most general, flexible solution for it. The problem could be one you

have seen and solved before, but your solution probably didn't have the kind of completeness you'll see embodied in a pattern.

Although they're called "design patterns," they really aren't tied to the realm of design. A pattern seems to stand apart from the traditional way of thinking about analysis, design, and implementation. Instead, a pattern embodies a complete idea within a program, and thus it can sometimes appear at the analysis phase or high-level design phase. This is interesting because a pattern has a direct implementation in code and so you might not expect it to show up before low-level design or implementation (and in fact you might not realize that you need a particular pattern until you get to those phases).

The basic concept of a pattern can also be seen as the basic concept of program design: adding a layer of abstraction. Whenever you abstract something you're isolating particular details, and one of the most compelling motivations behind this is to *separate things that change from things that stay the same*. Another way to put this is that once you find some part of your program that's likely to change for one reason or another, you'll want to keep those changes from propagating other changes throughout your code. Not only does this make the code much cheaper to maintain, but it also turns out that it is usually simpler to understand (which results in lowered costs).

Often, the most difficult part of developing an elegant and cheap-to-maintain design is in discovering what I call "the vector of change." (Here, "vector" refers to the maximum gradient and not a container class.) This means finding the most important thing that changes in your system, or put another way, discovering where your greatest cost is. Once you discover the vector of change, you have the focal point around which to structure your design.

So the goal of design patterns is to isolate changes in your code. If you look at it this way, you've been seeing some design patterns already in this book. For example, inheritance can be thought of as a design pattern (albeit one implemented by the compiler). It allows you to express differences in behavior (that's the thing that changes) in objects that all have the same interface (that's what stays the same). Composition can also be considered a pattern, since it allows you to change—dynamically or statically—the objects that implement your class, and thus the way that class works.

You've also already seen another pattern that appears in *Design Patterns*: the *iterator* (Java 1.0 and 1.1 capriciously calls it the **Enumeration**; Java

2 containers use "iterator"). This hides the particular implementation of the container as you're stepping through and selecting the elements one by one. The iterator allows you to write generic code that performs an operation on all of the elements in a sequence without regard to the way that sequence is built. Thus your generic code can be used with any container that can produce an iterator.

Pattern taxonomy

One of the events that's occurred with the rise of design patterns is what could be thought of as the "pollution" of the term – people have begun to use the term to mean just about anything synonymous with "good." After some pondering, I've come up with a sort of hierarchy describing a succession of different types of categories:

1. **Idiom**: how we write code in a particular language to do this particular type of thing. This could be something as common as the way that you code the process of stepping through an array in C (and not running off the end).

2. **Specific Design**: the solution that we came up with to solve this particular problem. This might be a clever design, but it makes no attempt to be general.

3. **Standard Design**: a way to solve this *kind* of problem. A design that has become more general, typically through reuse.

4. **Design Pattern**: how to solve an entire class of similar problem. This usually only appears after applying a standard design a number of times, and then seeing a common pattern throughout these applications.

I feel this helps put things in perspective, and to show where something might fit. However, it doesn't say that one is better than another. It doesn't make sense to try to take every problem solution and generalize it to a design pattern – it's not a good use of your time, and you can't force the discovery of patterns that way; they tend to be subtle and appear over time.

One could also argue for the inclusion of *Analysis Pattern* and *Architectural Pattern* in this taxonomy.

Design principles

(Update from slides to here)

When I put out a call for ideas in my newsletter[4], a number of suggestions came back which turned out to be very useful, but different than the above classification, and I realized that a list of design principles is at least as important as design structures, but for a different reason: these allow you to ask questions about your proposed design, to apply tests for quality.

- **Principle of least astonishment** (don't be astonishing).

- **Make common things easy, and rare things possible**

- **Consistency**. One thing has become very clear to me, especially because of Python: the more random rules you pile onto the programmer, rules that have nothing to do with solving the problem at hand, the slower the programmer can produce. And this does not appear to be a linear factor, but an exponential one.

- **Law of Demeter**: a.k.a. "Don't talk to strangers." An object should only reference itself, its attributes, and the arguments of its methods.

- **Subtraction**: a design is finished when you cannot take anything else away.

- **Simplicity before generality[5]**. (A variation of *Occam's Razor*, which says "the simplest solution is the best"). A common problem we find in frameworks is that they are designed to be general purpose without reference to actual systems. This leads to a dizzying array of options that are often unused, misused or just not useful. However, most developers work on specific systems, and the quest for generality does not always serve them well. The best route to generality is through understanding well-defined specific examples. So, this principle acts as the tie breaker between otherwise equally viable design alternatives. Of course, it is entirely possible that the simpler solution is the more general one.

- **Reflexivity** (my suggested term). One abstraction per class, one class per abstraction. Might also be called **Isomorphism**.

- **Independence** or **Orthogonality**. Express independent ideas independently. This complements Separation, Encapsulation and Variation, and is part of the Low-Coupling-High-Cohesion message.

- **Once and once only**: Avoid duplication of logic and structure where the duplication is not accidental, ie where both pieces of code express the same intent for the same reason.

In the process of brainstorming this idea, I hope to come up with a small handful of fundamental ideas that can be held in your head while you analyze a problem. However, other ideas that come from this list may end up being useful as a checklist while walking through and analyzing your design.

Classifying patterns

The *Design Patterns* book discusses 23 different patterns, classified under three purposes (all of which revolve around the particular aspect that can vary). The three purposes are:

1. **Creational**: how an object can be created. This often involves isolating the details of object creation so your code isn't dependent on what types of objects there are and thus doesn't have to be changed when you add a new type of object. The aforementioned *Singleton* is classified as a creational pattern, and later in this book you'll see examples of *Factory Method* and *Prototype*.

2. **Structural**: designing objects to satisfy particular project constraints. These work with the way objects are connected with other objects to ensure that changes in the system don't require changes to those connections.

3. **Behavioral**: objects that handle particular types of actions within a program. These encapsulate processes that you want to perform, such as interpreting a language, fulfilling a request, moving through a sequence (as in an iterator), or implementing an algorithm. This book contains examples of the *Observer* and the *Visitor* patterns.

The *Design Patterns* book has a section on each of its 23 patterns along with one or more examples for each, typically in C++ (rather restricted C++, at that) but sometimes in Smalltalk. (You'll find that this doesn't matter too much since you can easily translate the concepts from either language into Java.) This book will revisit many of the patterns shown in *Design Patterns* but with a Java orientation, since the language changes the expression and understanding of the patterns. However, the GoF examples will not be repeated here, since I believe that it's possible to produce more illuminating examples given some effort. The goal is to provide you with a decent feel for what patterns are about and why they are so important.

After years of looking at these things, it began to occur to me that the patterns themselves use basic principles of organization, other than (and more fundamental than) those described in *Design Patterns*. These principles are based on the structure of the implementations, which is where I have seen great similarities between patterns (more than those expressed in *Design Patterns*). Although we generally try to avoid implementation in favor of interface, for awhile I thought that it was easier to understand the patterns in terms of these structural principles, and tried reorganizing the book around the patterns based on their structure instead of the categories presented in *Design Patterns*.

However, a later insight made me realize that it's more useful to organize the patterns in terms of the problems they solve. I believe this is a subtle but important distinction from the way Metsker organizes the patterns by intent in *Design Patterns Java Workshop* (Addison-Wesley 2002), because I hope that you will then be able to recognize your problem and search for a solution, if the patterns are organized this way.

In the process of doing all this "book refactoring" I realized that if I changed it once, I would probably change it again (there's definitely a design maxim in there), so I removed all references to chapter numbers in order to facilitate this change (the little-known "numberless chapter" pattern ☺).

The development challenge

Issues of development, the UML process, Extreme Programming.

Is evaluation valuable? The Capability Immaturity Model:

Wiki Page: http://c2.com/cgi-bin/wiki?CapabilityImMaturityModel
Article: http://www.embedded.com/98/9807br.htm

Pair programming research:

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

Unit testing

In an earlier version of this book I decided that unit testing was essential (for all of my books) and that JUnit was too verbose and clunky to consider. At that time I wrote my own unit testing framework using Java reflection to simplify the syntax necessary to achieve unit testing. For the third edition of *Thinking in Java*, we developed another unit testing framework for that book which would test the output of examples.

In the meantime, JUnit has changed to add a syntax remarkably similar to the one that I used in an earlier version of this book. I don't know how much influence I may have had on that change, but I'm simply happy that it has happened, because I no longer feel the need to support my own system (which you can still find <some URL here>) and can simply recommend the defacto standard.

I have introduced and described the style of JUnit coding that I consider a "best practice" (primarily because of simplicity), in *Thinking in Java, 3rd edition*, chapter 15. That section provides an adequate introduction to any of the unit testing you will see associated with this book (however, the unit testing code will not normally be included in the text of this book). When you download the code for this book, you will find (4/9/2003: Eventually, not yet) unit tests along with the code examples whenever possible.

Location of test code

(From Bill):

Public: in **test** subdirectory; different package (don't include in jar).

Package access: same package, subdirectory path underneath library code (don't include in jar)

Private access: (white box testing). Nested class, strip out, or Junit addons.

Simplifying Idioms

Before getting into more complex techniques, it's helpful to look at some basic ways to keep code simple and straightforward.

Messenger

The most trivial of these is the *messenger*, which simply packages information into an object to be passed around, instead of passing all the pieces around separately. Note that without the messenger, the code for **translate()** would be much more confusing to read:

```
//: 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);
  }
} ///:~
```

Since the goal of a messenger is only to carry data, that data is made **public** for easy access. However, you may also have reasons to make the fields **private**.

Collecting Parameter

Messenger's big brother is the *collecting parameter*, whose job is to capture information from the method to which it is passed. Generally, this is used when the collecting parameter is passed to multiple methods, so it's like a bee collecting pollen.

A container makes an especially useful collecting parameter, since it is already set up to dynamically add objects:

```
//: 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);
  }
} ///:~
```

The collecting parameter must have some way to set or insert values. Note that by this definition, a messenger could be used as a collecting parameter. The key is that a collecting parameter is passed about and modified by the methods it is passed to.

Object quantity

The two patterns described here are solely used to control the quantity of objects.

Singleton could actually be thought of as a special case of *Object Pool*, but the applications of the *Object Pool* tend to be uniqe enough from *Singleton* that it's worth treating the two separately.

Singleton

Possibly the simplest design pattern is the *singleton*, which is a way to provide one and only one object of a particular type. An important aspect of Singleton is that you provide a global access point, so singletons are often a solution for what you would have used a global variable for in C. In addition, a singleton often has the characteristics of a registry or lookup service – it's a place you go to find references to other objects.

Singletons can be found in the Java libraries, but here's a more direct example:

```
//: 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; }
```

```java
  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);
  }
} ///:~
```

The key to creating a singleton is to prevent the client programmer from having any way to create an object except the ways you provide. You must make all constructors **private**, and you must create at least one constructor to prevent the compiler from synthesizing a default constructor for you (which it will create using package access).

At this point, you decide how you're going to create your object. Here, it's created statically, but you can also wait until the client programmer asks for one and create it on demand. In any case, the object should be stored privately. You provide access through **public** methods. Here, **getReference( )** produces the reference to the **Singleton** object. The rest of the interface (**getValue( )** and **setValue( )**) is the regular class interface.

Java also allows the creation of objects through cloning. In this example, making the class **final** prevents cloning. Since **Singleton** is inherited

directly from **Object**, the **clone( )** method remains **protected** so it cannot be used (doing so produces a compile-time error). However, if you're inheriting from a class hierarchy that has already overridden **clone( )** as **public** and implemented **Cloneable**, the way to prevent cloning is to override **clone( )** and throw a **CloneNotSupportedException** as described in Appendix A of *Thinking in Java, 2nd edition*. (You could also override **clone( )** and simply return **this**, but that would be deceiving since the client programmer would think they were cloning the object, but would instead still be dealing with the original.) Actually, this isn't precisely true, because even in the above situation someone could still use reflection to invoke **clone( )** [[is this true? **clone( )** is still protected so I'm not so sure. If it is true, you'd have to throw **CloneNotSupportedException** as the only way to guarantee un-cloneability]]

Exercises

1. **SingletonPattern.java** always creates an object, even if it's never used. Modify this program to use *lazy initialization*, so the singleton object is only created the first time that it is needed.

2. Create a registry/lookup service that accepts a Java interface and produces a reference to an object that implements that interface.

Object pool

Note that you aren't restricted to creating only one object. This is also a technique to create a limited pool of objects. In that situation, however, you can be confronted with the problem of sharing objects in the pool. If this is an issue, you can create a solution involving a check-out and check-in of the shared objects.

As an example, consider a database. Commercial databases often restrict the number of connections that you can use at any one time. Here is an implementation that uses an object pool to manage the connections. First, the basic concept of managing a pool of objects is implemented as a separate class:

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

```java
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");
  }
} ///:~
```

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

interface Connection {
  Object get();
```

```java
    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);
  }
} ///:~
```

Exercises

1. Add unit tests to ConnectionPoolDemo.java to demonstrate the problem that the client may release the connection but still continue to use it.

Object decoupling

Both *Proxy* and *State* provide a surrogate class that you use in your code; the real class that does the work is hidden behind this surrogate class. When you call a method in the surrogate, it simply turns around and calls the method in the implementing class. These two patterns are so similar that the *Proxy* is simply a special case of *State*. One is tempted to just lump the two together into a pattern called *Surrogate*, but the term "proxy" has a long-standing and specialized meaning, which probably explains the reason for the two different patterns.

The basic idea is simple: from a base class, the surrogate is derived along with the class or classes that provide the actual implementation:
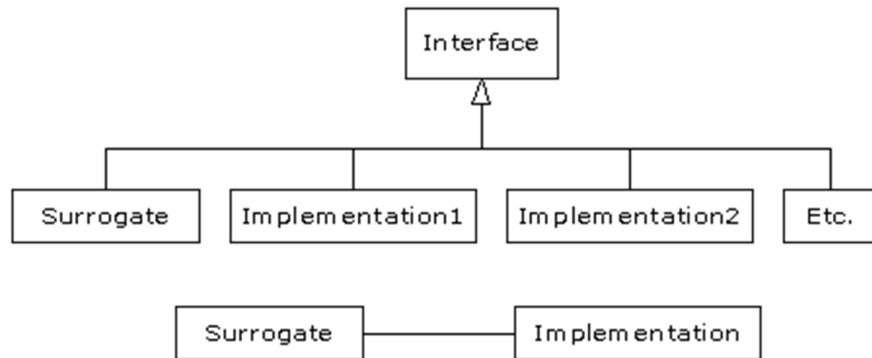
When a surrogate object is created, it is given an implementation to which to send all of the method calls.

Structurally, the difference between *Proxy* and *State* is simple: a *Proxy* has only one implementation, while *State* has more than one. The application of the patterns is considered (in *Design Patterns*) to be distinct: *Proxy* is used to control access to its implementation, while *State* allows you to change the implementation dynamically. However, if you expand your notion of "controlling access to implementation" then the two fit neatly together.

Proxy: fronting for another object

If we implement *Proxy* by following the above diagram, it looks like this:

```
//: 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;
```

```java
  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);
  }
} ///:~
```

Of course, it isn't necessary that **Implementation** have the same interface as **Proxy**; as long as **Proxy** is somehow "speaking for" the class that it is referring method calls to then the basic idea is satisfied (note that this statement is at odds with the definition for Proxy in GoF). However, it is convenient to have a common interface so that **Implementation** is forced to fulfill all the methods that **Proxy** needs to call.

The PoolManager using Proxy

```java
//: 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:
```

```java
    // 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() {
```

```java
    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);
  }
} ///:~
```

Dynamic Proxies

In JDK 1.3, the *Dynamic Proxy* was introduced. Although a little complex at first, this is an intruiging tool.

Here's an interesting little starting example, which works and proves that yes, indeed, the invocation handler is being called so the proxying etc. is actually working. So it's pretty cool, and it's in my head now, but I still have to figure out something reasonable to do with the invocation handler to come up with a useful example...

```java
// 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");
  }
} ///:~
```

Exercise: Use the Java dynamic proxy to create an object that acts as a front end for a simple configuration file. For example, in **good_stuff.txt** you can have entries like this:

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

A client programmer of this **NeatPropertyBundle** could then write:

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

The contents of the configuration file can contain anything, with any variable names. The dynamic proxy will either map to the name or tell you it doesn't exist somehow (probably by returning **null**). If you set a property and it doesn't already exist, the dynamic proxy will create the new entry. The **toString( )** method should display all the current entries.

Exercise: similar to the previous exercise, use the Java dynamic proxy to make a connection to the DOS Autoexec.bat file.

Exercise: Accept an SQL query which returns data, then read the DB metadata. Now, for each record, provide an object which has attributes corresponding to the column names and of appropriate data types.

Exercise: Create a simple server and client that uses XML-RPC. Each object the client returns should use the dynamic proxy concept to exercise the remote methods.

Andrea writes:

I'm not sure about the exercises you suggest, except for the last one. The thing is that I like to think at invocation handler as somthing providing features that are orthogonal to the ones provided by the object being "proxied".

In other words: the implementation of the invocation handler is completely independent from the interface(s) of the object that the dynamically-generated proxy represent. Which means that once you have implemented an invocation handler, you can use for any class that exposes interfaces, even for classes and interfaces that were not present when the handler was implemented.

That's why I say the the handler provides services that are orthogonal to the ones provided by the proxied object. Rickard has a few handlers in his SmartWorld example, and they one I like the best is a call-retry handler. It basically makes a call into the actual object, and if the call generates an exception if waits for a while, then makes the same call again for a total of three times. If all three calls fails, it returns an exception. And you can use such a handler on _any_ class.

The implementation is way too complex for what you are trying to demonstrate. I'm using this example just to explain what I mean by orthogonal services.

In your list of exercises, the only one that, in my opinion, makes sense to implement using dynamic proxies is the last one, the one using XML-RPC to communicate with an object. And that's because the mechanism you use to dispatch the message (XML-RPC) is orthogonal to the services provided by the object you want to reach.

State: changing object behavior

An object that appears to change its class.

Indications: conditional code in most or all methods.

The *State* pattern switches from one implementation to another during the lifetime of the surrogate, in order to produce different behavior from the same method call(s). It's a way to improve the implementation of your code when you seem to be doing a lot of testing inside each of your methods before deciding what to do for that method. For example, the fairy tale of the frog-prince contains an object (the creature) that behaves differently depending on what state it's in. You could implement this using a boolean that you test:

```
//: 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);
  }
} ///:~
```

However, the **greet()** method, and any other methods that must test
isFrog before they perform their operations, ends up with awkward code.
By delegating the operations to a State object that can be changed, this
code is simplified.

```
//: 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);
  }
} ///:~
```

In addition, changes to the State are automatically propagated throughout, rather than requiring an edit across the class methods in order to effect changes.

Here's the basic structure of State:

```
//: 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;
  }
```

```java
  // 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);
  }
} ///:~
```

In **main( )**, you can see that the first implementation is used for a bit, then the second implementation is swapped in and that is used.

There are a number of details that are choices that you must make according to the needs of your own implementation, such as whether the fact that you are using State is exposed to the client, and how the changes to State are made. Sometimes (as in the Swing LayoutManager) the client may pass in the object directly, but in KissingPrincess2.java the fact that State is used is invisible to the client. In addition, the mechanism for changing state may be simple or complex – in State Machine, described later in this book, larger sets of states and different mechanisms for changing are explored.

The Swing LayoutManager example mentioned above is an interesting example because it show behavior of both Strategy and State.

The difference between *Proxy* and *State* is in the problems that are solved. The common uses for *Proxy* as described in *Design Patterns* are:

1.   **Remote proxy**. This proxies for an object in a different address space. A remote proxy is created for you automatically by the RMI compiler **rmic** as it creates stubs and skeletons.

2. **Virtual proxy**. This provides "lazy initialization" to create expensive objects on demand.

3. **Protection proxy**. Used when you don't want the client programmer to have full access to the proxied object.

4. **Smart reference**. To add additional actions when the proxied object is accessed. For example, or to keep track of the number of references that are held for a particular object, in order to implement the *copy-on-write* idiom and prevent object aliasing. A simpler example is keeping track of the number of calls to a particular method.

You could look at a Java reference as a kind of protection proxy, since it controls access to the actual object on the heap (and ensures, for example, that you don't use a **null** reference).

[[Rewrite this: In *Design Patterns*, *Proxy* and *State* are not seen as related to each other because the two are given (what I consider arbitrarily) different structures. *State*, in particular, uses a separate implementation hierarchy but this seems to me to be unnecessary unless you have decided that the implementation is not under your control (certainly a possibility, but if you own all the code there seems to be no reason not to benefit from the elegance and helpfulness of the single base class). In addition, *Proxy* need not use the same base class for its implementation, as long as the proxy object is controlling access to the object it "fronting" for. Regardless of the specifics, in both *Proxy* and *State* a surrogate is passing method calls through to an implementation object.]]]

State can be found everywhere because it's such a fundamental idea. For example, in Builder, the "Director" uses a backend Builder object to produce different behaviors.

Iterators: decoupling algorithms from containers

Alexander Stepanov thought for years about the problem of generic programming techniques before creating the STL (along with Dave Musser). He came to the conclusion that all algorithms are defined on algebraic structures – what we would call containers.

In the process, he realized that iterators are central to the use of algorithms, because they decouple the algorithms from the specific type of container that the algorithm might currently be working with. This means that you can describe the algorithm without worrying about the particular sequence it is operating on. More generally, *any* code that you write using iterators is decoupled from the data structure that the code is manipulating, and thus your code is more general and reusable.

The use of iterators also extends your code into the realm of *functional programming*, whose objective is to describe *what* a program is doing at every step rather than *how* it is doing it. That is, you say "sort" rather than describing the sort. The objective of the C++ STL was to provide this *generic programming* approach for C++ (how successful this approach will actually be remains to be seen).

If you've used containers in Java (and it's hard to write code without using them), you've used iterators – in the form of the **Enumeration** in Java 1.0/1.1 and the **Iterator** in Java 2. So you should already be familiar with their general use. If not, see Chapter 9, *Holding Your Objects*, under *Iterators* in *Thinking in Java, 2nd edition* (freely downloadable from *www.BruceEckel.com*).

Because the Java 2 containers rely heavily on iterators they become excellent candidates for generic/functional programming techniques. This chapter will explore these techniques by converting the STL algorithms to Java, for use with the Java 2 container library.

Type-safe iterators

In *Thinking in Java, 2nd edition*, I show the creation of a type-safe container that will only accept a particular type of object. A reader, Linda Pazzaglia, asked for the other obvious type-safe component, an iterator that would work with the basic **java.util** containers, but impose the constraint that the type of objects that it iterates over be of a particular type.

If Java ever includes a template mechanism, this kind of iterator will have the added advantage of being able to return a specific type of object, but without templates you are forced to return generic **Object**s, or to require a bit of hand-coding for every type that you want to iterate through. I will take the former approach.

A second design decision involves the time that the type of object is determined. One approach is to take the type of the first object that the iterator encounters, but this is problematic because the containers may rearrange the objects according to an internal ordering mechanism (such as a hash table) and thus you may get different results from one iteration to the next. The safe approach is to require the user to establish the type during construction of the iterator.

Lastly, how do we build the iterator? We cannot rewrite the existing Java library classes that already produce **Enumeration**s and **Iterator**s. However, we can use the *Decorator* design pattern, and create a class that simply wraps the **Enumeration** or **Iterator** that is produced, generating a new object that has the iteration behavior that we want (which is, in this case, to throw a **RuntimeException** if an incorrect type is encountered) but with the same interface as the original **Enumeration** or **Iterator**, so that it can be used in the same places (you may argue that this is actually a *Proxy* pattern, but it's more likely *Decorator* because of its intent). Here is the code:

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

Exercises

1. Create an example of the "virtual proxy."

2. Create an example of the "Smart reference" proxy where you keep count of the number of method calls to a particular object.

3. Create a program similar to certain DBMS systems that only allow a certain number of connections at any time. To implement this, use a singleton-like system that controls the number of "connection" objects that it creates. When a user is finished with a connection, the system must be informed so that it can check that connection back in to be reused. To guarantee this, provide a proxy object instead of a reference to the actual connection, and design the proxy so that it will cause the connection to be released back to the system.

4. Using the *State* pattern, make a class called **UnpredictablePerson** which changes the kind of response to its **hello( )** method depending on what kind of **Mood** it's in. Add an additional kind of **Mood** called **Prozac**.

5. Create a simple copy-on write implementation.

6.   The **java.util.Map** has no way to automatically load a two-dimensional array of objects into a **Map** as key-value pairs. Create an adapter class that does this.

7.   Create an *Adapter Factory* that dynamically finds and produces the adapter that you need to connect a given object to a desired interface.

8.   Solve the above exercise using the *dynamic proxy* that's part of the Java standard library.

9.   Modify the Object Pool solution so that the objects are returned to the pool automatically after a certain amount of time.

10.   Modify the above solution to use "leasing" so that the client can renew the lease on the object to prevent it from being automatically released by the timer.

11.   Modify the Object Pool system to take threading issues into account.

Factoring commonality

Applying the "once and only once" principle produces the most basic pattern of putting code that changes into a method.

This can be expressed two ways:

Strategy: choosing the algorithm at run-time

Strategy also adds a "Context" which can be a surrogate class that controls the selection and use of the particular strategy object—just like *State*! Here's what it looks like:

```
//: 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);
  }
} ///:~
```

Note similarity with template method – TM claims distinction that it has
more than one method to call, does things piecewise. However, it's not
unlikely that strategy object would have more than one method call;
consider Shalloway's order fulfullment system with country information
in each strategy.

Strategy example from JDK: comparator objects.

Policy: generalized strategy

Although GoF says that Policy is just another name for strategy, their use of Strategy implicitly assumes a single method in the strategy object – that you've broken out your changing algorithm as a single piece of code.

Others[6] use Policy to mean an object that has multiple methods that may vary independently from class to class. This gives more flexibility than being restricted to a single method.

For example, a shipping policy for a product can be used to describe shipping issues for sending a package to various different countries. This may include the available methods of shipping, how to calculate postage or shipping cost, customs requirements and fees, and special handling costs. All these things may vary independently of each other, and more importantly you may need the information from each at different points in the shipping process.

It also seems generally useful to distinguish Strategies with single methods from Policies with multiple methods.

Template method

An application framework allows you to inherit from a class or set of classes and create a new application, reusing most of the code in the existing classes and overriding one or more methods in order to customize the application to your needs. A fundamental concept in the application framework is the *Template Method* which is typically hidden beneath the covers and drives the application by calling the various methods in the base class (some of which you have overridden in order to create the application).

For example, whenever you create an applet you're using an application framework: you inherit from **JApplet** and then override **init( )**. The

applet mechanism (which is a *Template Method*) does the rest by drawing the screen, handling the event loop, resizing, etc.

An important characteristic of the *Template Method* is that it is defined in the base class and cannot be changed. It's sometimes a **private** method but it's virtually always **final**. It calls other base-class methods (the ones you override) in order to do its job, but it is usually called only as part of an initialization process (and thus the client programmer isn't necessarily able to call it directly).

```java
//: 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);
  }
} ///:~
```

The base-class constructor is responsible for performing the necessary
initialization and then starting the "engine" (the template method) that
runs the application (in a GUI application, this "engine" would be the
main event loop). The client programmer simply provides definitions for
customize1( ) and **customize2( )** and the "application" is ready to run.

Exercises

1.   Create a framework that takes a list of file names on the
        command line. It opens each file except the last for reading, and
        the last for writing. The framework will process each input file
        using an undetermined policy and write the output to the last file.
        Inherit to customize this framework to create two separate
        applications:
        1) Converts all the letters in each file to uppercase.
        2) Searches the files for words given in the first file.

Encapsulating creation

When you discover that you need to add new types to a system, the most
sensible first step is to use polymorphism to create a common interface to
those new types. This separates the rest of the code in your system from
the knowledge of the specific types that you are adding. New types may be
added without disturbing existing code ... or so it seems. At first it would
appear that the only place you need to change the code in such a design is
the place where you inherit a new type, but this is not quite true. You

must still create an object of your new type, and at the point of creation you must specify the exact constructor to use. Thus, if the code that creates objects is distributed throughout your application, you have the same problem when adding new types—you must still chase down all the points of your code where type matters. It happens to be the *creation* of the type that matters in this case rather than the *use* of the type (which is taken care of by polymorphism), but the effect is the same: adding a new type can cause problems.

The solution is to force the creation of objects to occur through a common *factory* rather than to allow the creational code to be spread throughout your system. If all the code in your program must go through this factory whenever it needs to create one of your objects, then all you must do when you add a new object is to modify the factory.

Since every object-oriented program creates objects, and since it's very likely you will extend your program by adding new types, I suspect that factories may be the most universally useful kinds of design patterns.

Although only the *Simple Factory Method* is a true singleton, you'll find that each specify factory class in the more general types of factories will only have a single instance.

Simple Factory method

As an example, let's revisit the **Shape** system.

One approach is to make the factory a **static** method of the base class:

```
//: 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);
  }
```

```java
}

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);
  }
} ///:~
```

The **factory( )** takes an argument that allows it to determine what type of **Shape** to create; it happens to be a **String** in this case but it could be any set of data. The **factory( )** is now the only other code in the system that needs to be changed when a new type of **Shape** is added (the initialization data for the objects will presumably come from somewhere outside the system, and not be a hard-coded array as in the above example).

To encourage creation to only happen in the **factory( )**, the constructors for the specific types of **Shape** are give package access, so **factory( )** has access to the constructors but they are not available outside the package.

Polymorphic factories

The **static factory( )** method in the previous example forces all the creation operations to be focused in one spot, so that's the only place you need to change the code. This is certainly a reasonable solution, as it throws a box around the process of creating objects. However, the *Design Patterns* book emphasizes that the reason for the *Factory Method* pattern is so that different types of factories can be subclassed from the basic factory (the above design is mentioned as a special case). However, the book does not provide an example, but instead just repeats the example used for the *Abstract Factory* (you'll see an example of this in the next section). Here is **ShapeFactory1.java** modified so the factory methods are in a separate class as virtual functions. Notice also that the specific **Shape** classes are dynamically loaded on demand:

```
//: 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) {
```

```java
      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 {
```

```java
  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);
  }
} ///:~
```

Now the factory method appears in its own class, **ShapeFactory**, as the **create( )** method. This is a **protected** method which means it cannot be called directly, but it can be overridden. The subclasses of **Shape** must each create their own subclasses of **ShapeFactory** and override the **create( )** method to create an object of their own type. The actual creation of shapes is performed by calling **ShapeFactory.createShape( )**, which is a static method that uses the **Map** in **ShapeFactory** to find the appropriate factory object based on an identifier that you pass it. The factory is immediately used to create the shape object, but you could imagine a more complex problem where the appropriate factory object is returned and then used by the caller to create an object in a more sophisticated way. However, it seems that much of the time you don't need the intricacies of the polymorphic factory method, and a single static method in the base class (as shown in **ShapeFactory1.java**) will work fine.

Notice that the **ShapeFactory** must be initialized by loading its **Map** with factory objects, which takes place in the static initialization clause of each of the **Shape** implementations. So to add a new type to this design you must inherit the type, create a factory, and add the static initialization clause to load the **Map**. This extra complexity again suggests the use of a **static** factory method if you don't need to create individual factory objects.

Abstract factories

The *Abstract Factory* pattern looks like the factory objects we've seen previously, with not one but several factory methods. Each of the factory methods creates a different kind of object. The idea is that at the point of creation of the factory object, you decide how all the objects created by that factory will be used. The example given in *Design Patterns* implements portability across various graphical user interfaces (GUIs): you create a factory object appropriate to the GUI that you're working with, and from then on when you ask it for a menu, button, slider, etc. it will automatically create the appropriate version of that item for the GUI. Thus you're able to isolate, in one place, the effect of changing from one GUI to another.

As another example suppose you are creating a general-purpose gaming environment and you want to be able to support different types of games. Here's how it might look using an abstract factory:

```
//: factory:Games.java
```

```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();
}
```

```java
// 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);
  }
} ///:~
```

In this environment, **Player** objects interact with **Obstacle** objects, but there are different types of players and obstacles depending on what kind of game you're playing. You determine the kind of game by choosing a particular **GameElementFactory**, and then the **GameEnvironment** controls the setup and play of the game. In this example, the setup and play is very simple, but those activities (the *initial conditions* and the *state change*) can determine much of the game's outcome. Here, **GameEnvironment** is not designed to be inherited, although it could very possibly make sense to do that.

This also contains examples of *Double Dispatching* and the *Factory Method*, both of which will be explained later.

Exercises

1.   Add a class Triangle to ShapeFactory1.java

2.   Add a class **Triangle** to **ShapeFactory2.java**

3.   Add a new type of **GameEnvironment** called **GnomesAndFairies** to **Games.java**

4.   Modify **ShapeFactory2.java** so that it uses an *Abstract Factory* to create different sets of shapes (for example, one particular type of factory object creates "thick shapes," another creates "thin shapes," but each factory object can create all the shapes: circles, squares, triangles etc.).

Specialized creation

Prototype

Objects are created by cloning a prototypical instance. An example of this appears in the "Pattern Refactoring" chapter.

Builder

The goal of builder is to separate the construction from the "representation," to allow multiple different representations. The construction process stays the same, but the resulting object has different possible representations. GoF points out that the main difference with Abstract Factory is that a Builder creates the object step-by-step, so the fact that the creation process is spread out in time seems to be important. In addition, it seems that the "director" gets a stream of pieces that it passes to the Builder, and each piece is used to perform one of the steps in the build process.

One example given in GoF is that of a text format converter. The incoming format is RTF, and once it is parsed the directives are passed to the text converter, which may be implemented in different ways depending on whether the resulting format is ASCII, TeX, or a "GUI Text Widget." Although the resulting "object" (the entire converted text file) is created over time, if you consider the conversion of each RTF directive to be an object, this feels to me a little more like Bridge, because the specific types of converters extend the interface of the base class. Also, the general solution to the problem would allow multiple readers on the "front end" and multiple converters on the "back end," which is a primary characteristic of Bridge.

To me, the fact that Builder has multiple steps in creating an object, and those steps are accessed externally to the Builder object, is the essence of what distinguishes it (structurally, anyway) from a regular

factory. However, GoF emphasizes that you're able to create different representations using the same process. They never define exactly what they mean by representation. (Does the "representation" involve an object that is too large? Would the need for Builder vanish if the representation was broken into smaller objects?)

The other example in GoF creates a maze object and adds rooms within the maze and doors within the rooms. Thus it is a multistep process, but alas, the different "representations" are the "Standard" and "Complex" mazes – not really different kinds of mazes, but instead different complexity. I think I would have tried to create one maze builder that could handle arbitrarily complex mazes. The final variation of the maze builder is something that doesn't create mazes at all, but instead counts the rooms in an existing maze.

Neither the RTF converter nor the Mazebuilder example makes an overwhelmingly compelling case for Builder. Readers have suggested that the output of the Sax XML parser, and standard compiler parsers, might naturally be fed into a Builder.

Here's an example that may be a little more compelling, or at least give more of an idea of what Builder is trying to do. Media may be constructed into different representations, in this case books, magazines and web sites. The example argues that the steps involved are the same, and so can be abstracted into the director class.

```java
//: 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 {
```

```java
  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;
```

```java
    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);
  }
} ///:~
```

Note that in some ways this could be seen as a more complicated State pattern, since the behavior of the director depends on what type of builder you use. Instead of simply forwarding the requests through to the underlying State object, however, the director has a sequence of operations to perform, and it uses the State object as a Policy to fulfill its job. Thus, Builder could be described as using a Policy to create objects.

Exercises

1. Break a text file up into an input stream of words (consider using regular expressions for this). Create one Builder that puts the words into a **java.util.TreeSet**, and another that produces a **java.util.HashMap** containing words and occurrences of those words (that is, it does a word count).

Too many

Flyweight: too many objects

The odd thing about flyweight, in the company of the other design patterns, is that it's a performance hack. It's generally ideal to simply make an object for every item in your system, but some problems generate a prohibitive number of objects, which may result in excessive slowness or running out of memory.

Flyweight solves this problem by reducing the number of objects. To do this, you externalize some of the data in an object, so that you can pretend that you have more objects than you really do. However, this adds complexity to the interface for using such objects, because you must pass in additional information to method calls in order to tell the method how to find the externalized information.

As a very simple example, consider a **DataPoint** object that holds an **int**, a **float**, and an **id** that carries the object number. Suppose you need to create a million of these objects, and then manipulate them, like so:

```
//: 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]);
  }
} ///:~
```

Depending on your computer, this program may take several seconds to run. More complex objects and more involved operations may cause the overhead to become untenable. To solve the problem the **DataPoint** can be reduced from a million objects to one object by externalizing the data held in the **DataPoint**:

```
//: 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));
  }
} ///:~
```

Since all the data is now in **ExternalizedData**, each call to a **FlyPoint** method must include the index into **ExternalizedData**. For consistency, and to remind the reader of the similarity with the implicit **this** pointer in method calls, the "this index" is passed in as the first argument.

Naturally, it's worth repeating admonishments against premature optimization. "First make it work, then make it fast – if you have to." Also, a profiler is the tool to use for discovering performance bottlenecks, not guesswork.

Decorator: too many classes

The use of layered objects to dynamically and transparently add responsibilities to individual objects is referred to as the *decorator* pattern.

Used when subclassing creates too many (& inflexible) classes

All decorators that wrap around the original object must have the same basic interface

Dynamic proxy/surrogate?

This accounts for the odd inheritance structure

Tradeoff: coding is more complicated when using decorators

Basic decorator structure
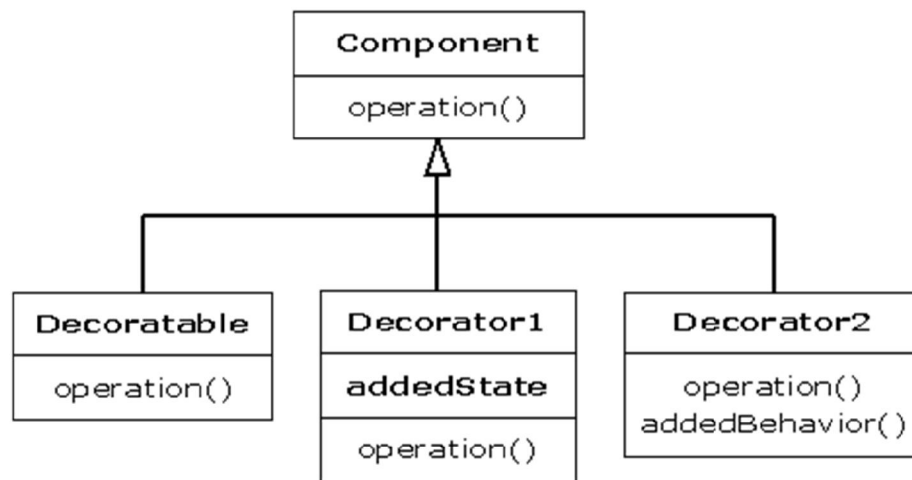
A coffee example

Consider going down to the local coffee shop, *BeanMeUp*, for a coffee. There are typically many different drinks on offer -- espressos, lattes, teas, iced coffees, hot chocolate to name a few, as well as a number of extras (which cost extra too) such as whipped cream or an extra shot of espresso. You can also make certain changes to your drink at no extra cost, such as asking for decaf coffee instead of regular coffee.

Quite clearly if we are going to model all these drinks and combinations, there will be sizeable class diagrams. So for clarity we will only consider a subset of the coffees: Espresso, Espresso Con Panna, Café Late, Cappuccino and Café Mocha. We'll include 2 extras - whipped cream ("whipped") and an extra shot of espresso; and three changes - decaf, steamed milk ("wet") and foamed milk ("dry").

Class for each combination

One solution is to create an individual class for every combination. Each class describes the drink and is responsible for the cost etc. The resulting menu is huge, and a part of the class diagram would look something like this:

Here is one of the combinations, a simple implementation of a Cappuccino:

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

The key to using this method is to find the particular combination you want. So, once you've found the drink you would like, here is how you would use it, as shown in the CoffeeShop class in the following code:

//: decorator:nodecorators:CoffeeShop.java

```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);
  }
} ///:~
```

And here is the corresponding output:

```
Cappucino: $1.0
Cafe Mocha decaf whipped cream: $1.25
```

You can see that creating the particular combination you want is easy, since you are just creating an instance of a class. However, there are a number of problems with this approach. Firstly, the combinations are fixed statically so that any combination a customer may wish to order

needs to be created up front. Secondly, the resulting menu is so huge that finding your particular combination is difficult and time consuming.

The decorator approach

Another approach would be to break the drinks down into the various components such as espresso and foamed milk, and then let the customer combine the components to describe a particular coffee.

In order to do this programmatically, we use the Decorator pattern. A Decorator adds responsibility to a component by wrapping it, but the Decorator conforms to the interface of the component it encloses, so the wrapping is transparent. Decorators can also be nested without the loss of this transparency.
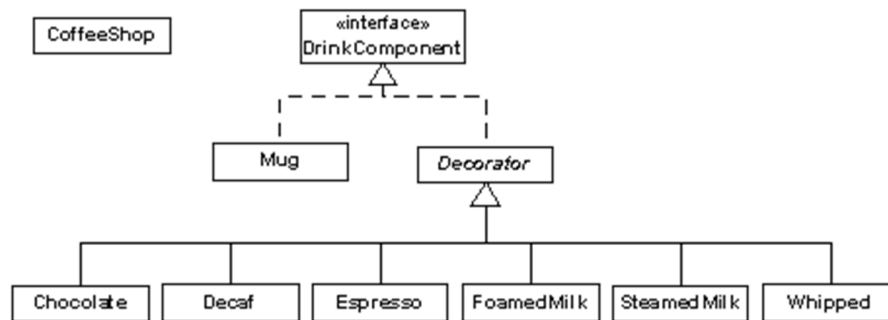
Methods invoked on the Decorator can in turn invoke methods in the component, and can of course perform processing before or after the invocation.

So if we added **getTotalCost()** and **getDescription()** methods to the **DrinkComponent** interface, an Espresso looks like this:

```
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;
  }
}
```

You combine the components to create a drink as follows, as shown in the code below:

```
//: 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);
  }
```

```java
  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;
  }
```

```java
    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);
  }
} ///:~
```

This approach would certainly provide the most flexibility and the smallest menu. You have a small number of components to choose from, but assembling the description of the coffee then becomes rather arduous.

If you want to describe a plain cappuccino, you create it with

```
new Espresso(new FoamedMilk(new Mug()))
```
Creating a decaf Café Mocha with whipped cream requires an even longer description.

Compromise

The previous approach takes too long to describe a coffee. There will also be certain combinations that you will describe regularly, and it would be convenient to have a quick way of describing them.

The 3rd approach is a mixture of the first 2 approaches, and combines flexibility with ease of use. This compromise is achieved by creating a reasonably sized menu of basic selections, which would often work exactly as they are, but if you wanted to decorate them (whipped cream, decaf etc.) then you would use decorators to make the modifications. This is the type of menu you are presented with in most coffee shops.
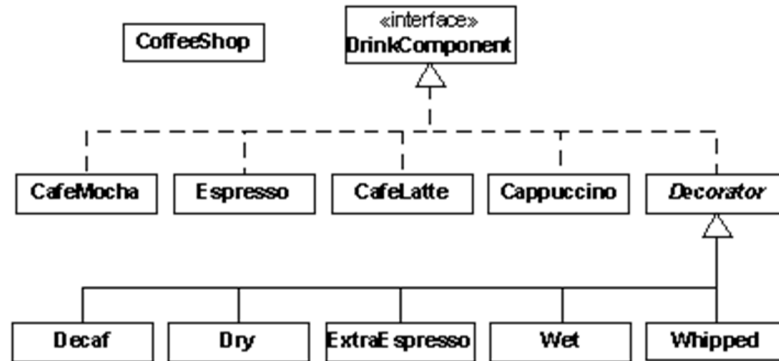
Here is how to create a basic selection, as well as a decorated selection:

```
//: 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() {
```

```java
    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();
```

```java
  }
  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 {
```

```java
  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);
  }
} ///:~
```

You can see that creating a basic selection is quick and easy, which makes sense since they will be described regularly. Describing a decorated drink is more work than when using a class per combination, but clearly less work than when only using decorators.

The final result is not too many classes, but not too many decorators either. Most of the time it's possible to get away without using any decorators at all, so we have the benefits of both approaches.

Other considerations

What happens if we decide to change the menu at a later stage, such as by adding a new type of drink? If we had used the class per combination approach, the effect of adding an extra such as syrup would be an exponential growth in the number of classes. However, the implications to the all decorator or compromise approaches are the same - one extra class is created.

How about the effect of changing the cost of steamed milk and foamed milk, when the price of milk goes up? Having a class for each combination means that you need to change a method in each class, and thus maintain many classes. By using decorators, maintenance is reduced by defining the logic in one place.

Exercises

1.   Add a Syrup class to the decorator approach described above. Then create a Café Latte (you'll need to use steamed milk with an espresso) with syrup.

2.   Repeat Exercise 1 for the compromise approach.

3.   Create a simple decorator system that models the fact that some birds fly and some don't, some swim and some don't, and some do both.

4.   Implement the decorator pattern to create a Pizza restaurant, which has a set menu of choices as well as the option to design your own pizza. Follow the compromise approach to create a menu consisting of a Margherita, Hawaiian, Regina, and Vegetarian pizzas, with toppings (decorators) of Garlic, Olives, Spinach, Avocado, Feta and Pepperdews. Create a Hawaiian pizza,

as well as a Margherita decorated with Spinach, Feta, Pepperdews and Olives.

5.   About *Decorator,* the *Design Patterns* book states: "With decorators, responsibilities can be added and removed at run-time simply by attaching and detaching them." Implement the coffee decoration system to allow this "simple" detaching of a responsibility from the middle of the list of decorators of a complex coffee beverage.

Connecting different types

Adapter

Adapter takes one type and produces an interface to some other type. When you've got *this*, and you need *that*, *Adapter* solves the problem. The only requirement is to produce a *that*, and there are a number of ways you can accomplish this adaptation.

```java
//: 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");
  }
```

```java
}

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);
  }
} ///:~
```

```java
//: 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
```

```java
    // 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);
 }
} ///:~
```

I'm taking liberties with the term "proxy" here, because in *Design Patterns* they assert that a proxy must have an identical interface with the object that it is a surrogate for. However, if you have the two words together: "proxy adapter," it is perhaps more reasonable.

Bridge

While researching *Bridge*, I discovered that it appears to be the most poorly-described pattern in the GoF. I began to come to this conclusion when reading Alan Shalloway's chapter on *Bridge* in his book *Design Patterns Explained* – he begins by pointing out that the description in GoF left him quite unenlightened.

At a conference, I talked to two people who had written about and were giving talks on design patterns, including *Bridge*. In two separate discussions I got completely different perspectives on the structure of *Bridge*.

Armed with misinformation, I delved back into the GoF and realized that neither of the above perspectives agreed with the book. I also found that the book did a miserable job of describing *Bridge*, except in one place – not the general structure chart describing the pattern, which wasn't helpful, but in the structure chart describing their specific example. Only if you stare at that for a bit does *Bridge* begin to make sense.

An important feature to understand when looking at *Bridge* is that it is often a construct that is used to help you *write* code. You may choose the objects you use for a particular situation at compile-time or runtime, but the goal of *Bridge* is to allow you to structure your code so that you can easily add new kinds of front-end objects which are implemented with functionality in new kinds of back-end objects. Thus, both front-end and back-end can vary independently of each other.

The front-end classes can have completely different interfaces from each other, and typically do. What they have in common is that they can implement their functionality using facilities from any number of different back-end objects. The back-end objects also don't have the same interface. The only thing the back-end objects must have in common is that they implement the same kind of functionality – for example, a group of different ways to implement a graphics library or a set of different data-storage solutions.

Bridge is really a code-organization tool that allows you to add in any number of new front-end services that implement their operations by delegating to any number of back-end options. Using *Bridge*, you can accomplish this without the normal combinatorial explosion of possibilities that would otherwise occur. But keep in mind that the vector of change with *Bridge* is typically happening at coding time: it keeps your code organized when you are dealing with an increasing number of options for implementing functionality.
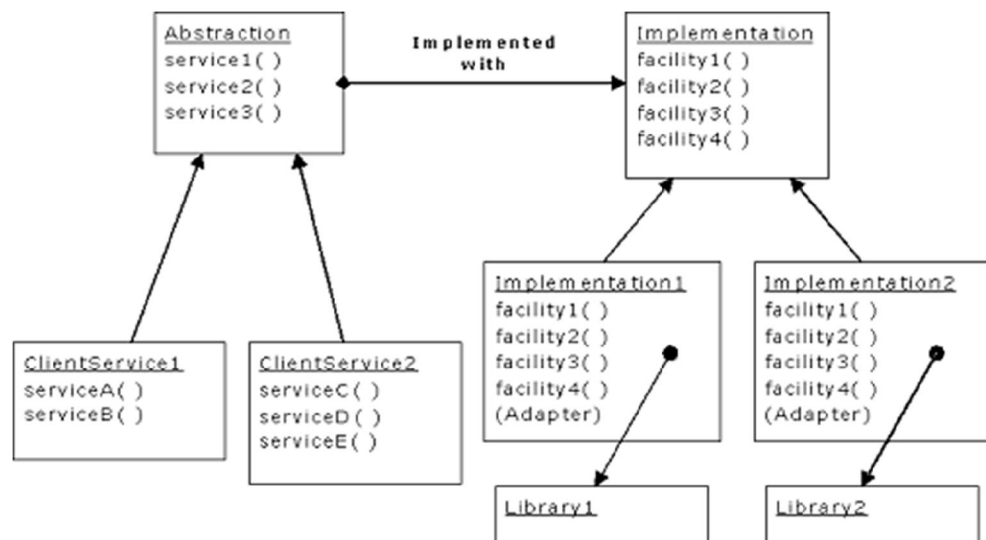
Here's an example whose sole purpose is to demonstrate the structure of *Bridge* (it implements the above diagram):

```
//: 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() {
```

```java
      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()");
```

```java
  }
  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");
```

```java
      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);
  }
} ///:~
```

The front-end base class provides the operations used to fulfill the front-end derived classes, *in terms of the methods of the back-end base class.* Thus, any back-end derived class can be used to perform the operations

needed by the front-end class. Notice that the bridging happens in this sequence of steps, each of which is providing a layer of abstraction. Here, **Implementation** is defined as an interface to emphasize that all the functionality is implemented in the back-end derived classes, and none in the back-end base.

The back-end derived classes perform the operations defined in the base class by delegating to objects (of class **Library1** and **Library2**, in this case) that typically have radically different interfaces, but somehow provide the same functionality (this is one of the important objectives of *Bridge*). Effectively, each of the back-end implementations is an adapter to a different library or tool used to implement the desired functionality in a different way.

Exercises

1. Modify **BridgeStructure.java** so that the implementations are chosen using a factory.

2. Modify **BridgeStructure.java** to use delegation instead of inheritance on the front end. What benefits and drawbacks do you see by using delegation rather than inheritance?

3. Create an example of Bridge with an abstraction that is an *associative array*. This allows you to fetch elements by passing in a key Object. The constructor provides an initial set of key-value pairs which are placed in an array. As long as you only fetch elements, the array is used, but as soon as you set a new key-value pair, the implementation is switched to a map.

4. Use a bridge along with the collections in **java.util.collections** to create stack and queue classes using an **ArrayList**. After you get the system working, add a double-ended queue class. Now add a **LinkedList** as an implementation. These steps will demonstrate how *Bridge* allows you to add new front-end classes and new back-end classes in your code, with minimal impact.

5. Create a *Bridge* that provides a connection between various kinds of bookkeeping programs (along with their interfaces and data

formats) and different banks (which provide different kinds of services and interfaces).

Flexible structure

Composite

The important thing here is that all elements in the part-whole have operations, and that performing an operation on a node/composite also performs that operation on any children of that node/composite. GoF includes implementation details of containment and visitation of children in the interface of the base class, but this doesn't seem necessary. In the following example, the **Composite** class simply inherits **ArrayList** in order to gain its containment abilities.

```
//: 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);
  }
} ///:~
```

While this approach seems to be "the simplest thing that could possibly work," it's possible that in a larger system problems could arise. However, it's probably best to start with the simplest approach and change it only if the situation demands.

System decoupling

Observer

Like the other forms of callback, this contains a hook point where you can change code. The difference is in the observer's completely dynamic nature. It is often used for the specific case of changes based on other

object's change of state, but is also the basis of event management. Anytime you want to decouple the source of the call from the called code in a completely dynamic way.

The observer pattern solves a fairly common problem: What if a group of objects needs to update themselves when some object changes state? This can be seen in the "model-view" aspect of Smalltalk's MVC (model-view-controller), or the almost-equivalent "Document-View Architecture." Suppose that you have some data (the "document") and more than one view, say a plot and a textual view. When you change the data, the two views must know to update themselves, and that's what the observer facilitates. It's a common enough problem that its solution has been made a part of the standard **java.util** library.

There are two types of objects used to implement the observer pattern in Java. The **Observable** class keeps track of everybody who wants to be informed when a change happens, whether the "state" has changed or not. When someone says "OK, everybody should check and potentially update themselves," the **Observable** class performs this task by calling the **notifyObservers( )** method for each one on the list. The **notifyObservers( )** method is part of the base class **Observable**.

There are actually two "things that change" in the observer pattern: the quantity of observing objects and the way an update occurs. That is, the observer pattern allows you to modify both of these without affecting the surrounding code.

Observer is an "interface" class that only has one member function, **update( )**. This function is called by the object that's being observed, when that object decides its time to update all its observers. The arguments are optional; you could have an **update( )** with no arguments and that would still fit the observer pattern; however this is more general—it allows the observed object to pass the object that caused the update (since an **Observer** may be registered with more than one observed object) and any extra information if that's helpful, rather than forcing the **Observer** object to hunt around to see who is updating and to fetch any other information it needs.

The "observed object" that decides when and how to do the updating will be called the **Observable**.

Observable has a flag to indicate whether it's been changed. In a simpler design, there would be no flag; if something happened, everyone would be

notified. The flag allows you to wait, and only notify the **Observer**s when you decide the time is right. Notice, however, that the control of the flag's state is **protected**, so that only an inheritor can decide what constitutes a change, and not the end user of the resulting derived **Observer** class.

Most of the work is done in **notifyObservers( )**. If the **changed** flag has not been set, this does nothing. Otherwise, it first clears the **changed** flag so repeated calls to **notifyObservers( )** won't waste time. This is done before notifying the observers in case the calls to **update( )** do anything that causes a change back to this **Observable** object. Then it moves through the **set** and calls back to the **update( )** member function of each **Observer**.

At first it may appear that you can use an ordinary **Observable** object to manage the updates. But this doesn't work; to get an effect, you *must* inherit from **Observable** and somewhere in your derived-class code call **setChanged( )**. This is the member function that sets the "changed" flag, which means that when you call **notifyObservers( )** all of the observers will, in fact, get notified. *Where* you call **setChanged( )** depends on the logic of your program.

Observing flowers

Here is an example of the observer pattern:

```
//: 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
```

```java
    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!");
    }
  }
```

```java
    // 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);
  }
} ///:~
```

The events of interest are that a **Flower** can open or close. Because of the use of the inner class idiom, both these events can be separately observable phenomena. **OpenNotifier** and **CloseNotifier** both inherit **Observable**, so they have access to **setChanged( )** and can be handed to anything that needs an **Observable**.

The inner class idiom also comes in handy to define more than one kind of **Observer**, in **Bee** and **Hummingbird**, since both those classes may want to independently observe **Flower** openings and closings. Notice how the inner class idiom provides something that has most of the benefits of inheritance (the ability to access the **private** data in the outer class, for example) without the same restrictions.

In **main( )**, you can see one of the prime benefits of the observer pattern: the ability to change behavior at run time by dynamically registering and un-registering **Observer**s with **Observable**s.

If you study the code above you'll see that **OpenNotifier** and **CloseNotifier** use the basic **Observable** interface. This means that you could inherit other completely different **Observer** classes; the only connection the **Observer**s have with **Flower**s is the **Observer** interface.

A visual example of observers

The following example is similar to the **ColorBoxes** example from Chapter 14 in *Thinking in Java, 2nd Edition*. Boxes are placed in a grid on the screen and each one is initialized to a random color. In addition, each box **implements** the **Observer** interface and is registered with an **Observable** object. When you click on a box, all of the other boxes are notified that a change has been made because the **Observable** object automatically calls each **Observer** object's **update( )** method. Inside this method, the box checks to see if it's adjacent to the one that was clicked, and if so it changes its color to match the clicked box.

```
//: 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);
  }
```

```java
  }

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;
  }
} ///:~
```

When you first look at the online documentation for **Observable**, it's a bit confusing because it appears that you can use an ordinary **Observable** object to manage the updates. But this doesn't work; try it— inside **BoxObserver**, create an **Observable** object instead of a **BoxObservable** object and see what happens: nothing. To get an effect, you *must* inherit from **Observable** and somewhere in your derived-class code call **setChanged( )**. This is the method that sets the "changed" flag, which means that when you call **notifyObservers( )** all of the observers will, in fact, get notified. In the example above **setChanged( )** is simply called within **notifyObservers( )**, but you could use any criterion you want to decide when to call **setChanged( )**.

BoxObserver contains a single **Observable** object called **notifier**, and every time an **OCBox** object is created, it is tied to **notifier**. In **OCBox**, whenever you click the mouse the **notifyObservers( )** method is called, passing the clicked object in as an argument so that all the boxes receiving the message (in their **update( )** method) know who was clicked and can decide whether to change themselves or not. Using a combination of code

in **notifyObservers( )** and **update( )** you can work out some fairly complex schemes.

It might appear that the way the observers are notified must be frozen at compile time in the **notifyObservers( )** method. However, if you look more closely at the code above you'll see that the only place in **BoxObserver** or **OCBox** where you're aware that you're working with a **BoxObservable** is at the point of creation of the **Observable** object—from then on everything uses the basic **Observable** interface. This means that you could inherit other **Observable** classes and swap them at run time if you want to change notification behavior then.

Mediator

Sweep coupling under the rug, how is this different from MVC?

MVC has distinct model and view; mediator could be anything. MVC a flavor of mediator

Exercises

1. Create a minimal Observer-Observable design in two classes. Just create the bare minimum in the two classes, then demonstrate your design by creating one **Observable** and many **Observer**s, and cause the **Observable** to update the **Observer**s.

2. Create a minimal **Observer** system using **java.util.Timer** inside your **Observable**, to generate events that are reported to the **Observer**s. Create several different **Observer**s using anonymous inner classes, register these with the **Observable**, and show that they are called when the **Timer** events occur.

3. Modify **BoxObserver.java** to turn it into a simple game. If any of the squares surrounding the one you clicked is part of a contiguous patch of the same color, then all the squares in that patch are changed to the color you clicked on. You can configure the game for competition between players or to keep track of the number of clicks that a single player uses to turn the field into a

single color. You may also want to restrict a player's color to the first one that was chosen.

Reducing interface complexity

Sometimes the problem that you're solving is as simple as "I don't have the interface that I want." *Façade* creates an interface to a set of classes, simply to provide a more comfortable way to deal with a library or bundle of resources.

Façade

A general principle that I apply when I'm casting about trying to mold requirements into a first-cut object is "If something is ugly, hide it inside an object." This is basically what *Façade* accomplishes. If you have a rather confusing collection of classes and interactions that the client programmer doesn't really need to see, then you can create an interface that is useful for the client programmer and that only presents what's necessary.

Façade is often implemented as singleton abstract factory. Of course, you can easily get this effect by creating a class containing **static** factory methods:

```
//: 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);
  }
} ///:~
```

The example given in *Design Patterns* is just a class that uses the other classes.

A tax adviser is a Façade between you and the tax code, and a mediator between you and the tax system.

Package as a variation of *Façade*

To me, the *Façade* has a rather "procedural" (non-object-oriented) feel to it: you are just calling some functions to give you objects. And how different is it, really, from *Abstract Factory*? The point of *Façade* is to hide part of a library of classes (and their interactions) from the client programmer, to make the interface to that group of classes more digestible and easier to understand.

However, this is precisely what the packaging features in Java accomplish: outside of the library, you can only create and use **public** classes; all the non-**public** classes are only accessible within the package. It's as if *Façade* is a built-in feature of Java.

To be fair, *Design Patterns* is written primarily for a C++ audience. Although C++ has namespaces to prevent clashes of globals and class names, this does not provide the class hiding mechanism that you get with

non-**public** classes in Java. The majority of the time I think that Java packages will solve the *Façade* problem.

Algorithmic partitioning

Command: choosing the operation at run-time

In *Advanced C++:Programming Styles And Idioms (Addison-Wesley, 1992)*, Jim Coplien coins the term *functor* which is an object whose sole purpose is to encapsulate a function (since "functor" has a meaning in mathematics, in this book I shall use the more explicit term *function object*). The point is to decouple the choice of function to be called from the site where that function is called.

This term is mentioned but not used in *Design Patterns*. However, the theme of the function object is repeated in a number of patterns in that book.

A Command is a function object in its purest sense: a method that's an object[7]. By wrapping a method in an object, you can pass it to other methods or objects as a parameter, to tell them to perform this particular operation in the process of fulfilling your request. You could say that a *Command* is a messenger (because its intent and use is very straightforward) that carries behavior, rather than data.

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

interface Command {
  void execute();
```

```java
}

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

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

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);
  }
} ///:~
```

The primary point of *Command* is to allow you to hand a desired action to a method or object. In the above example, this provides a way to queue a set of actions to be performed collectively. In this case, it allows you to dynamically create new behavior, something you can normally only do by writing new code but in the above example could be done by interpreting a script (see the *Interpreter* pattern if what you need to do gets very complex).

Another example of *Command* is **refactor:DirList.java [????]**. The **DirFilter** class is the command object which contains its action in the method **accept( )** that is passed to the **list( )** method. The **list( )** method determines what to include in its result by calling **accept( )**.

Design Patterns says that "Commands are an object-oriented replacement for callbacks[8]." However, I think that the word "back" is an essential part of the concept of callbacks. That is, I think a callback actually reaches back to the creator of the callback. On the other hand, with a *Command* object you typically just create it and hand it to some method or object, and are not otherwise connected over time to the *Command* object. That's my take on it, anyway. Later in this book, I combine a group of design patterns under the heading of "callbacks."

Strategy appears to be a family of *Command* classes, all inherited from the same base. But if you look at *Command*, you'll see that it has the same structure: a hierarchy of function objects. The difference is in the way this hierarchy is used. As seen in **refactor:DirList.java**, you use *Command* to solve a particular problem—in that case, selecting files from a list. The "thing that stays the same" is the body of the method that's being called, and the part that varies is isolated in the function object. I would hazard to say that *Command* provides flexibility while you're writing the program, whereas *Strategy*'s flexibility is at run time. Nonetheless, it seems a rather fragile distinction.

Exercises

4.   Use *Command* in Chapter 3, Exercise 1.

Chain of responsibility

Example: translation service (local->global->Babelfish).

Chain of Responsibility might be thought of as a dynamic generalization of recursion using *Strategy* objects. You make a call, and each *Strategy* in

a linked sequence tries to satisfy the call. The process ends when one of the strategies is successful or the chain ends. In recursion, one method calls itself over and over until a termination condition is reached; with *Chain of Responsibility*, a method calls itself, which (by moving down the chain of *Strategies*) calls a different implementation of the method, etc., until a termination condition is reached. The termination condition is either the bottom of the chain is reached (in which case a default object is returned; you may or may not be able to provide a default result so you must be able to determine the success or failure of the chain) or one of the *Strategies* is successful.

Instead of calling a single method to satisfy a request, multiple methods in the chain have a chance to satisfy the request, so it has the flavor of an expert system. Since the chain is effectively a linked list, it can be dynamically created, so you could also think of it as a more general, dynamically-built **switch** statement.

In the GoF, there's a fair amount of discussion of how to create the chain of responsibility as a linked list. However, when you look at the pattern it really shouldn't matter how the chain is maintained; that's an implementation detail. Since GoF was written before the Standard Template Library (STL) was incorporated into most C++ compilers, the reason for this is most likely (1) there was no list and thus they had to create one and (2) data structures are often taught as a fundamental skill in academia, and the idea that data structures should be standard tools available with the programming language may not have occurred to the GoF authors. I maintain that the implementation of *Chain of Responsibility* as a chain (specifically, a linked list) adds nothing to the solution and can just as easily be implemented using a standard Java **List**, as shown below. Furthermore, you'll see that I've gone to some effort to separate the chain-management parts of the implementation from the various *Strategies*, so that the code can be more easily reused.

In **StrategyPattern.java**, above, what you probably want is to automatically find a solution. *Chain of Responsibility* provides a way to do this by chaining the *Strategy* objects together and providing a mechanism for them to automatically recurse through each one in the chain:

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

```java
// 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(
```

```java
      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);
  }
} ///:~
```

Exercises

1.   Implement *Chain of Responsibility* to create an "expert system"
     that solves problems by successively trying one solution after
     another until one matches. You should be able to dynamically add
     solutions to the expert system. The test for solution should just be
     a string match, but when a solution fits, the expert system should
     return the appropriate type of **ProblemSolver** object. What
     other pattern/patterns show up here?

2.   Implement *Chain of Responsibility* to create a language
     translator which begins by searching for a local specialized
     translation system (which may know specifics about your problem
     domain), then a more global generalized system, and finally falls
     back on BabelFish if it can't translate everything. Note that each
     link in the chain may partially translate what it's able to.

3.   Implement *Chain of Responsibility* to create an tool to help
     reformat Java source code by trying multiple approaches to
     breaking lines. Note that normal code and comments will probably
     need to be treated differently, leading to the possibility of
     implementing a *Tree of Responsibility*. Also note the similarity
     between this approach and the *Composite* design pattern; perhaps
     the more general description of this technique is a *Composite of
     Strategies*.

Externalizing object state

Memento

Use serialization to create an undo mechanism.

Complex interactions

Multiple dispatching

When dealing with multiple types which are interacting, a program can get particularly messy. For example, consider a system that parses and executes mathematical expressions. You want to be able to say **Number + Number**, **Number * Number**, etc., where **Number** is the base class for a family of numerical objects. But when you say **a + b**, and you don't know the exact type of either **a** or **b**, so how can you get them to interact properly?

The answer starts with something you probably don't think about: Java performs only single dispatching. That is, if you are performing an operation on more than one object whose type is unknown, Java can invoke the dynamic binding mechanism on only one of those types. This doesn't solve the problem, so you end up detecting some types manually and effectively producing your own dynamic binding behavior.

The solution is called *multiple dispatching*. Remember that polymorphism can occur only via member function calls, so if you want

double dispatching to occur, there must be two member function calls: the first to determine the first unknown type, and the second to determine the second unknown type. With multiple dispatching, you must have a polymorphic method call to determine each of the types. Generally, you'll set up a configuration such that a single member function call produces more than one dynamic member function call and thus determines more than one type in the process. To get this effect, you need to work with more than one polymorphic method call: you'll need one call for each dispatch. The methods in the following example are called **compete( )** and **eval( )**, and are both members of the same type. (In this case there will be only two dispatches, which is referred to as *double dispatching*). If you are working with two different type hierarchies that are interacting, then you'll have to have a polymorphic method call in each hierarchy.

Here's an example of multiple dispatching:

```
//: 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; }
```

```java
    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);
  }
} ///:~
```

Visitor, a type of multiple dispatching

The assumption is that you have a primary class hierarchy that is fixed; perhaps it's from another vendor and you can't make changes to that hierarchy. However, you'd like to add new polymorphic methods to that hierarchy, which means that normally you'd have to add something to the base class interface. So the dilemma is that you need to add methods to the base class, but you can't touch the base class. How do you get around this?

The design pattern that solves this kind of problem is called a "visitor" (the final one in the *Design Patterns* book), and it builds on the double dispatching scheme shown in the last section.

The visitor pattern allows you to extend the interface of the primary type by creating a separate class hierarchy of type **Visitor** to virtualize the operations performed upon the primary type. The objects of the primary type simply "accept" the visitor, then call the visitor's dynamically-bound member function.

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

interface Visitor {
```

```java
  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");
```

```java
  }
  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);
  }
} ///:~
```

Exercises

1.   Create a business-modeling environment with three types of **Inhabitant**: **Dwarf** (for engineers), **Elf** (for marketers) and **Troll** (for managers). Now create a class called **Project** that creates the different inhabitants and causes them to **interact( )** with each other using multiple dispatching.

2.   Modify the above example to make the interactions more detailed. Each **Inhabitant** can randomly produce a **Weapon** using **getWeapon( )**: a **Dwarf** uses **Jargon** or **Play**, an **Elf** uses **InventFeature** or **SellImaginaryProduct**, and a **Troll** uses **Edict** and **Schedule**. You must decide which weapons "win" and "lose" in each interaction (as in **PaperScissorsRock.java**). Add a **battle( )** member function to **Project** that takes two **Inhabitant**s and matches them against each other. Now create a **meeting( )** member function for **Project** that creates groups of **Dwarf**, **Elf** and **Manager** and battles the groups against each other until only members of one group are left standing. These are the "winners."

3.   Modify **PaperScissorsRock.java** to replace the double dispatching with a table lookup. The easiest way to do this is to create a **Map** of **Map**s, with the key of each **Map** the class of each object. Then you can do the lookup by saying: **((Map)map.get(o1.getClass())).get(o2.getClass())** Notice how much easier it is to reconfigure the system. When is it more appropriate to use this approach vs. hard-coding the dynamic dispatches? Can you create a system that has the syntactic simplicity of use of the dynamic dispatch but uses a table lookup?

4.   Modify Exercise 2 to use the table lookup technique described in Exercise 3.

Multiple languages

This chapter looks at the value of crossing language boundaries. It is often very advantageous to solve a problem using more than one programming language, rather than being arbitrarily stuck using a single language. As you'll see in this chapter, a problem that is very difficult or tedious to solve in one language can often be solved quickly and easily in another. If you can combine the use of languages, you can create your product much more quickly and cheaply.

The most straightforward use of this idea is the *Interpreter* design pattern, which adds an interpreted language to your program to allow the end user to easily customize a solution. In Java, the easiest and most powerful way to do this is with *Jython*[9], an implementation of the Python language in pure Java byte codes.

Interpreter solves a particular problem – that of creating a scripting language for the user. But sometimes it's just easier and faster to temporarily step into another language to solve a particular aspect of your problem. You're not creating an interpreter, you're just writing some code in another language. Again, Jython is a good example of this, but CORBA also allows you to cross language boundaries.

Interpreter motivation

If the application user needs greater run time flexibility, for example to create scripts describing the desired behavior of the system, you can use the *Interpreter* design pattern. Here, you create and embed a language interpreter into your program.

Remember that each design pattern allows one or more factors to change, so it's important to first be aware of which factor is changing. Sometimes the end users of your application (rather than the programmers of that application) need complete flexibility in the way that they configure some aspect of the program. That is, they need to do some kind of simple programming. The interpreter pattern provides this flexibility by adding a language interpreter.

The problem is that developing your own language and building an interpreter is a time-consuming distraction from the process of developing your application. You must ask whether you want to finish writing your application or create a new language. The best solution is to reuse code: embed an interpreter that's already been built and debugged for you. The Python language can be freely embedded into your for-profit application without signing any license agreement, paying royalties, or dealing with strings of any kind. There are basically no restrictions at all when you're using Python.

Python is a language that is very easy to learn, very logical to read and write, supports functions and objects, has a large set of available libraries, and runs on virtually every platform. You can download Python and learn more about it by going to *www.Python.org*.

For solving Java problems, we will look at a special version of Python called Jython. This is generated entirely in Java byte codes, so incorporating it into your application is quite simple, and it's as portable as Java is. It has an extremely clean interface with Java: Java can call Python classes, and Python can call Java classes.

Python is designed with classes from the ground up and is a truly pure object oriented language (both C++ and Java violate purity in various ways). Python scales up so that you can create very big programs without losing control of the code.

To install Python, go to *www.Python.org* and follow the links and instructions. To install Jython, go to *http://jython.sourceforge.net*. The download is a **.class** file, which will run an installer when you execute it with Java. You also need to add **jython.jar** to your CLASSPATH. You can find further installation instructions at http://www.bruceeckel.com/TIPatterns/Building-Code.html.

Python overview

To get you started, here is a brief introduction for the experienced programmer (which is what you should be if you're reading this book). You can refer to the full documentation at *www.Python.org* (especially the incredibly useful HTML page *A Python Quick Reference*), and also numerous books such as *Learning Python* by Mark Lutz and David Ascher (O'Reilly, 1999).

Python is often referred to as a scripting language, but scripting languages tend to be limiting, especially in the scope of the problems that they solve. Python, on the other hand, is a programming language that also supports scripting. It *is* marvelous for scripting, and you may find yourself replacing all your batch files, shell scripts, and simple programs with Python scripts. But it is far more than a scripting language.

Python is designed to be very clean to write and especially to read. You will find that it's quite easy to read your own code long after you've written it, and also to read other people's code. This is accomplished partially through clean, to-the-point syntax, but a major factor in code readability is indentation – scoping in Python is determined by indentation. For example:

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

The '#' denotes a comment that goes until the end of the line, just like C++ and Java '//' comments.

First notice that the basic syntax of Python is C-ish; notice the **if** statement. But in a C **if**, you would be required to use parentheses around the conditional, whereas they are not necessary in Python (but it won't complain if you use them anyway).

The conditional clause ends with a colon, and this indicates that what follows will be a group of indented statements, which are the "then" part of the **if** statement. In this case, there is a "print" statement which sends the result to standard output, followed by an assignment to a variable named **val**. The subsequent statement is not indented so it is no longer part of the **if**. Indenting can nest to any level, just like curly braces in C++ or Java, but unlike those languages there is no option (and no argument) about where the braces are placed – the compiler forces everyone's code to be formatted the same way, which is one of the main reasons for Python's consistent readability.

Python normally has only one statement per line (you can put more by separating them with semicolons), thus no terminating semicolon is necessary. Even from the brief example above you can see that the

language is designed to be as simple as possible, and yet still very readable.

Built-in containers

With languages like C++ and Java, containers are add-on libraries and not integral to the language. In Python, the essential nature of containers for programming is acknowledged by building them into the core of the language: both lists and associative arrays (a.k.a. maps, dictionaries, hash tables) are fundamental data types. This adds much to the elegance of the language.

In addition, the **for** statement automatically iterates through lists rather than just counting through a sequence of numbers. This makes a lot of sense when you think about it, since you're almost always using a **for** loop to step through an array or a container. Python formalizes this by automatically making **for** use an iterator that works through a sequence. Here's an example:

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

The first line creates a list. You can print the list and it will look exactly as you put it in (in contrast, remember that I had to create a special **Arrays2** class in *Thinking in Java, 2nd Edition* in order to print arrays in Java). Lists are like Java containers – you can add new elements to them (here, **append( )** is used) and they will automatically resize themselves. The **for** statement creates an iterator **x** which takes on each value in the list.

You can create a list of numbers with the **range( )** function, so if you really need to imitate C's **for**, you can.

Notice that there aren't any type declarations – the object names simply appear, and Python infers their type by the way that you use them. It's as if Python is designed so that you only need to press the keys that absolutely must. You'll find after you've worked with Python for a short while that you've been using up a lot of brain cycles parsing semicolons, curly braces, and all sorts of other extra verbiage that was demanded by

your non-Python programming language but didn't actually describe what your program was supposed to do.

Functions

To create a function in Python, you use the **def** keyword, followed by the function name and argument list, and a colon to begin the function body. Here is the first example turned into a function:

```
## 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")
##~
```

Notice there is no type information in the function signature – all it specifies is the name of the function and the argument identifiers, but no argument types or return types. Python is a *weakly-typed* language, which means it puts the minimum possible requirements on typing. For example, you could pass and return different types from the same function:

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

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

The only constraints on an object that is passed into the function are that the function can apply its operations to that object, but other than that, it doesn't care. Here, the same function applies the '+' operator to integers and strings:

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

```
  return arg1 + arg2

print sum(42, 47)
print sum('spam ', "eggs")
##~
```
When the operator '**+**' is used with strings, it means concatenation (yes, Python supports operator overloading, and it does a nice job of it).

Strings

The above example also shows a little bit about Python string handling, which is the best of any language I've seen. You can use single or double quotes to represent strings, which is very nice because if you surround a string with double quotes, you can embed single quotes and vice versa:

```
## 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'
##~
```
Note that Python was not named after the snake, but rather the Monty Python comedy troupe, and so examples are virtually required to include Python-esque references.

The triple-quote syntax quotes everything, including newlines. This makes it particularly useful for doing things like generating web pages (Python is an especially good CGI language), since you can just triple-quote the entire page that you want without any other editing.

The '**r**' right before a string means "raw," which takes the backslashes literally so you don't have to put in an extra backslash.

Substitution in strings is exceptionally easy, since Python uses C's **printf( )** substitution syntax, but for any string at all. You simply follow the string with a '**%**' and the values to substitute:

```
## interpreter:stringFormatting.py
val = 47
print "The number is %d" % val
val2 = 63.4
```

```
s = "val: %d, val2: %f" % (val, val2)
print s
##~
```
As you can see in the second case, if you have more than one argument you surround them in parentheses (this forms a *tuple*, which is a list that cannot be modified).

All the formatting from **printf( )** is available, including control over the number of decimal places and alignment. Python also has very sophisticated regular expressions.

Classes

Like everything else in Python, the definition of a class uses a minimum of additional syntax. You use the **class** keyword, and inside the body you use **def** to create methods. Here's a simple class:

```
## 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")
##~
```
Both methods have "**self**" as their first argument. C++ and Java both have a hidden first argument in their class methods, which points to the object that the method was called for and can be accessed using the keyword **this**. Python methods also use a reference to the current object, but when you are *defining* a method you must explicitly specify the reference as the first argument. Traditionally, the reference is called **self** but you could use any identifier you want (if you do not use **self** you will probably confuse a lot of people, however). If you need to refer to fields in the object or other methods in the object, you must use **self** in the expression. However,

when you call a method for an object as in **x.show( )**, you do not hand it the reference to the object – *that* is done for you.

Here, the first method is special, as is any identifier that begins and ends with double underscores. In this case, it defines the constructor, which is automatically called when the object is created, just like in C++ and Java. However, at the bottom of the example you can see that the creation of an object looks just like a function call using the class name. Python's spare syntax makes you realize that the **new** keyword isn't really necessary in C++ or Java, either.

All the code at the bottom is set off by an **if** clause, which checks to see if something called __**name**__ is equivalent to __**main**__. Again, the double underscores indicate special names. The reason for the **if** is that any file can also be used as a library module within another program (modules are described shortly). In that case, you just want the classes defined, but you don't want the code at the bottom of the file to be executed. This particular **if** statement is only true when you are running this file directly; that is, if you say on the command line:

Python SimpleClass.py
However, if this file is imported as a module into another program, the __**main**__ code is not executed.

Something that's a little surprising at first is that you define fields inside methods, and not outside of the methods like C++ or Java (if you create fields using the C++/Java style, they implicitly become static fields). To create an object field, you just name it – using **self** – inside of one of the methods (usually in the constructor, but not always), and space is created when that method is run. This seems a little strange coming from C++ or Java where you must decide ahead of time how much space your object is going to occupy, but it turns out to be a very flexible way to program.

Inheritance

Because Python is weakly typed, it doesn't really care about interfaces – all it cares about is applying operations to objects (in fact, Java's **interface** keyword would be wasted in Python). This means that inheritance in Python is different from inheritance in C++ or Java, where you often inherit simply to establish a common interface. In Python, the only reason you inherit is to inherit an implementation – to re-use the code in the base class.

If you're going to inherit from a class, you must tell Python to bring that class into your new file. Python controls its name spaces as aggressively as Java does, and in a similar fashion (albeit with Python's penchant for simplicity). Every time you create a file, you implicitly create a module (which is like a package in Java) with the same name as that file. Thus, no **package** keyword is needed in Python. When you want to use a module, you just say **import** and give the name of the module. Python searches the PYTHONPATH in the same way that Java searches the CLASSPATH (but for some reason, Python doesn't have the same kinds of pitfalls as Java does) and reads in the file. To refer to any of the functions or classes within a module, you give the module name, a period, and the function or class name. If you don't want the trouble of qualifying the name, you can say

from *module* **import** *name(s)*

Where "name(s)" can be a list of names separated by commas.

You inherit a class (or classes – Python supports multiple inheritance) by listing the name(s) of the class inside parentheses after the name of the inheriting class. Note that the **Simple** class, which resides in the file (and thus, module) named **SimpleClass** is brought into this new name space using an **import** statement:

```
## 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 is inherited from **Simple**, and in the constructor, the base-class constructor is called. In **display( )**, **showMsg( )** can be called as a method of **self**, but when calling the base-class version of the method you are overriding, you must fully qualify the name and pass **self** in as the first argument, as shown in the base-class constructor call. This can also be seen in the overridden version of **show( )**.

In ___**main**___, you will see (when you run the program) that the base-class constructor is called. You can also see that the **showMsg( )** method is available in the derived class, just as you would expect with inheritance.

The class **Different** also has a method named **show( )**, but this class is not derived from **Simple**. The **f( )** method defined in ___**main**___ demonstrates weak typing: all it cares about is that **show( )** can be applied to **obj**, and it doesn't have any other type requirements. You can see that **f( )** can be applied equally to an object of a class derived from **Simple** and one that isn't, without discrimination. If you're a C++ programmer, you should see that the objective of the C++ **template** feature is exactly this: to provide weak typing in a strongly-typed language. Thus, in Python you automatically get the equivalent of templates – without having to learn that particularly difficult syntax and semantics.

Creating a language

It turns out to be remarkably simple to use Jython to create an interpreted language inside your application. Consider the greenhouse controller example from Chapter 8 of *Thinking in Java, 2nd edition*. This is a situation where you want the end user – the person managing the greenhouse – to have configuration control over the system, and so a simple scripting language is the ideal solution.

To create the language, we'll simply write a set of Python classes, and the constructor of each will add itself to a (static) master list. The common data and behavior will be factored into the base class **Event**. Each **Event** object will contain an **action** string (for simplicity – in reality, you'd have some sort of functionality) and a time when the event is supposed to run. The constructor initializes these fields, and then adds the new **Event** object to a static list called **events** (defining it in the class, but outside of any methods, is what makes it static):

```
#: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()
##~
```

The constructor of each derived class calls the base-class constructor, which adds the new object to the list. The **run( )** function sorts the list, which automatically uses the **___cmp___( )** method that was defined in **Event** to base comparisons on time only. In this example, it only prints out the list, but in the real system it would wait for the time of each event to come up and then run the event.

The **___main___** section performs a simple test on the classes.

The above file is now a module that can be included in another Python program to define all the classes it contains. But instead of an ordinary Python program, let's use Jython, inside of Java. This turns out to be remarkably simple: you import some Jython classes, create a **PythonInterpreter** object, and cause the Python files to be loaded:

```
//- 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);
  }
} ///:~
```

The **PythonInterpreter** object is a complete Python interpreter that accepts commands from the Java program. One of these commands is **execfile( )**, which tells it to execute all the statements it finds in a particular file. By executing **GreenHouseLanguage.py**, all the classes from that file are loaded into our **PythonInterpreter** object, and so it now "holds" the greenhouse controller language. The **Schedule.ghs** file is the one created by the end user to control the greenhouse. Here's an example:

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

This is the goal of the interpreter design pattern: to make the configuration of your program as simple as possible for the end user. With Jython you can achieve this with almost no effort at all.

One of the other methods available to the **PythonInterpreter** is **exec( )**, which allows you to send a command to the interpreter. Here, the **run( )** function is called using **exec( )**.

Remember, to run this program you must go to *http://jython.sourceforge.net* and download and install Jython (actually, you only need **jython.jar** in your CLASSPATH). Once that's in place, it's just like running any other Java program.

Controlling the interpreter

The prior example only creates and runs the interpreter using external scripts. In the rest of this chapter, we shall look at more sophisticated ways to interact with Jython. The simplest way to exercise more control over the **PythonInterpreter** object from within Java is to send data to the interpreter, and pull data back out.

Putting data in

To inject data into your Python program, the **PythonInterpreter** class has a deceptively simple method: **set( )**. However, **set( )** takes many different data types and performs conversions upon them. The following example is a reasonably thorough exercise of the various **set( )** possibilities, along with comments that should give a fairly complete explanation:

```
//- 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 =
```

```java
    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:
```

```java
    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);
  }
} ///:~
```

As usual with Java, the distinction between real objects and primitive types causes trouble. In general, if you pass a regular object to **set( )**, it knows what to do with it, but if you want to pass in a primitive you must perform a conversion. One way to do this is to create a "Py" type, such as **PyInteger** or **PyFloat**. but it turns out you can also use Java's own object wrappers like **Integer** and **Float**, which is probably going to be a lot easier to remember.

Early in the program you'll see an **exec( )** containing the Python statement:

```
print a[5:]
```
The colon inside the indexing statement indicates a Python *slice*, which produces a range of elements from the original array. In this case, it produces an array containing the elements from number 5 until the end of the array. You could also say '**a[3:5]**' to produce elements 3 through 5, or '**a[:5]**' to produce the elements zero through 5. The reason a slice is used in this statement is to make sure that the Java **String** has really been converted to a Python string, which can also be treated as an array of characters.

You can see that it's possible, using **exec( )**, to create a Python function (although it's a bit awkward). The **prt( )** function prints the whole array, and then (to make sure it's a real Python array), iterates through each element of the array and prints it. Finally, it prints the class of the array, so we can see what conversion has taken place (Python not only has run-time type information, it also has the equivalent of Java reflection). The **prt( )** function is used to print arrays that come from each of the Java primitive types.

Although a Java **ArrayList** does pass into the interpreter using **set( )**, and you can index into it as if it were an array, trying to create a slice fails. To completely convert it into an array, one approach is to simply extract a Java array using **toArray( )**, and pass that in. The **set( )** method converts it to a **PyArray** – one of the classes provided with Jython – which can be treated as a Python array (you can also explicitly create a **PyArray**, but this seems unnecessary).

Finally, a **Map** is created and passed directly into the interpreter. While it is possible to do simple things like index into the resulting object, it's not a real Python dictionary so you can't (for example) call the **keys( )** method. There is no straightforward way to convert a Java **Map** into a Python dictionary, and so I wrote a utility called **toPyDictionary( )** and made it a **static** method of **com.bruceeckel.python.PyUtil**. This also includes utilities to extract a Python array into a Java **List**, and a Python dictionary into a Java **Map**:

```
//- 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));
  }
} ///:~
```

Here is the (black-box) unit testing code:

```
//- 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);
  }
} ///:~
```

We'll see the use of the extraction tools in the next section.

Getting data out

There are a number of different ways to extract data from the **PythonInterpreter**. If you simply call the **get( )** method, passing it the object identifier as a string, it returns a **PyObject** (part of the **org.python.core** support classes). It's possible to "cast" it using the **__tojava__( )** method, but there are better alternatives:

1. The convenience methods in the **Py** class, such as **py2int( )**, take a **PyObject** and convert it to a number of different types.

2. An overloaded version of **get( )** takes the desired Java **Class** object as a second argument, and produces an object that has that run-time type (so you still need to perform a cast on the result in your Java code).

Using the second approach, getting an array from the **PythonInterpreter** is quite easy. This is especially useful because Python is exceptionally good at manipulating strings and files, and so you will commonly want to extract the results as an array of strings. For example, you can do a wildcard expansion of file names using Python's **glob( )**, as shown further down in the following code:

```
//- 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__(
```

```java
//!   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);
  }
} ///:~
```

The last two examples show the extraction of Python tuples and lists into Java **List**s, and Python dictionaries into Java **Map**s. Both of these cases require more processing than is provided in the standard Jython library, so I have again created utilities in **com.bruceeckel.pyton.PyUtil**: **toList( )** to produce a **List** from a Python sequence, and **toMap( )** to produce a **Map** from a Python dictionary. The **PyUtil** methods make it easier to take important data structures back and forth between Java and Python.

Multiple interpreters

It's also worth noting that you can have multiple **PythonInterpreter** objects in a program, and each one has its own name space:

```
//- 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);
  }
} ///:~
```

When you run the program you'll see that the value of **a** is distinct within each **PythonInterpreter**.

Controlling Java from Jython

Since you have the Java language at your disposal, and you can set and retrieve values in the interpreter, there's a tremendous amount that you can accomplish with the above approach (controlling Python from Java). But one of the amazing things about Jython is that it makes Java classes almost transparently available from within Jython. Basically, a Java class looks like a Python class. This is true for standard Java library classes as well as classes that you create yourself, as you can see here:

```
## 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
##~
```

The "=M" comment is recognized by the makefile generator tool (that I created for this book) as a replacement makefile command. This will be used instead of the commands that the extraction tool would normally place in the makefile.

Note that the **import** statements map to the Java package structure exactly as you would expect. In the first example, a **Date( )** object is created as if it were a native Python class, and printing this object just calls **toString( )**.

ValGen implements the concept of a "generator" which is used a great deal in the C++ STL (*Standard Template Library*, part of the Standard C++ Library). A generator is an object that produces a new object every time its "generation method" is called, and it is quite convenient for filling containers. Here, I wanted to use it in a **for** iteration, and so I needed the generation method to be the one that is called by the iteration process. This is a special method called __**getitem**__**( )**, which is actually the overloaded operator for indexing, '**[ ]**'. A **for** loop calls this method every time it wants to move the iteration forward, and when the elements run out, __**getitem**__**( )** throws an out-of-bounds exception and that signals the end of the **for** loop (in other languages, you would never use an exception for ordinary control flow, but in Python it seems to work quite well). This exception happens automatically when **self.val[i]** runs out of elements, so the __**getitem**__**( )** code turns out to be simple. The

only complexity is that ___**getitem___( )** appears to return *two* objects instead of just one. What Python does is automatically package multiple return values into a tuple, so you still only end up returning a single object (in C++ or Java you would have to create your own data structure to accomplish this). In addition, in the **for** loop where **ValGen** is used, Python automatically "unpacks" the tuple so that you can have multiple iterators in the **for**. These are the kinds of syntax simplifications that make Python so endearing.

The **map** and **set** objects are instances of Java's **HashMap** and **HashSet**, again created as if those classes were just native Python components. In the **for** loop, the **put( )** and **add( )** methods work just like they do in Java. Also, indexing into a Java **Map** uses the same notation as for dictionaries, but note that to iterate through the keys in a **Map** you must use the **Map** method **keySet( )** rather than the Python dictionary method **keys( )**.

The final part of the example shows the use of a Java class that I created from scratch, to demonstrate how trivial it is. Notice also that Jython intuitively understands JavaBeans properties, since you can either use the **getVal( )** and **setVal( )** methods, or assign to and read from the equivalent **val** property. Also, **getChars( )** returns a **Character[]** in Java, and this becomes an array in Python.

The easiest way to use Java classes that you create for use inside a Python program is to put them inside a package. Although Jython can also import unpackaged java classes (**import JavaClass**), all such unpackaged java classes will be treated as if they were defined in different packages so they can only see each other's public methods.

Java packages translate into Python modules, and Python must import a module in order to be able to use the Java class. Here is the Java code for **JavaClass**:

```
//- 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()");
  }
```

```java
  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);
  }
} ///:~
```

You can see that this is just an ordinary Java class, without any awareness that it will be used in a Jython program. For this reason, one of the important uses of Jython is in testing Java code[10]. Because Python is such a powerful, flexible, dynamic language it is an ideal tool for automated test frameworks, without making any changes to the Java code that's being tested.

Inner Classes

Inner classes becomes attributes on the class object. Instances of **static** inner classes can be create by the usual call:

```
com.foo.JavaClass.StaticInnerClass()
```
Non-**static** inner classes must have an outer class instance supplied explicitly as the first argument:

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

Using Java libraries

Jython wraps the Java libraries so that any of them can be used directly or via inheritance. In addition, Python shorthand simplifies coding.

As an example, consider the **HTMLButton.java** example from Chapter 9 of *Thinking in Java, 2nd edition* (you presumably have already downloaded and installed the source code for that book from *www.BruceEckel.com*, since a number of examples in this book use libraries from that book). Here is its conversion to Jython:

```
## 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):
  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
##~
```

If you compare the Java version of the program to the above Jython implementation, you'll see that Jython is shorter and generally easier to understand. For example, in the Java version to set up the frame you had to make several calls: the constructor for **JFrame( )**, the **setVisible( )** method and the **setDefaultCloseOperation( )** method, whereas in the above code all three of these operations are performed with a single constructor call.

Also notice that the **JButton** is configured with an **actionListener( )** method inside the constructor, with the assignment to **kapow**. In addition, Jython's JavaBean awareness means that a call to any method with a name that begins with "**set**" can be replaced with an assignment, as you can see above.

The only method that did not come over from Java is the **pack( )** method, which seems to be essential in order to force the layout to happen properly. It's also important that the call to **pack( )** appear *before* the **size** setting.

Inheriting from Java library classes

You can easily inherit from standard Java library classes in Jython. Here's the **Dialogs.java** example from Chapter 13 of *Thinking in Java, 2nd edition*, converted into Jython:

```
## 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 is inherited from **JDialog**, and you can see named arguments being used in the call to the base-class constructor.

In the creation of the "OK" **JButton**, note that the **actionPerformed** method is set right inside the constructor, and that the function is created using the Python **lambda** keyword. This creates a nameless function with the arguments appearing before the colon and the expression that generates the returned value after the colon. As you should know, the Java prototype for the **actionPerformed( )** method only contains a single argument, but the lambda expression indicates two. However, the second argument is provided with a default value, so the function *can* be called with only one argument. The reason for the second argument is seen in the default value, because this is a way to pass **self** into the lambda expression, so that it can be used to dispose of the dialog.

Compare this code with the version that's published in *Thinking in Java, 2nd edition*. You'll find that Python language features allow a much more succinct and direct implementation.

Creating Java classes with Jython

Although it does not directly relate to the original problem of this chapter (creating an interpreter), Jython has the additional ability to create Java classes directly from your Jython code. This can produce very useful results, as you are then able to treat the results as if they are native Java classes, albeit with Python power under the hood.

To produce Java classes from Python code, Jython comes with a compiler called **jythonc**.

The process of creating Python classes that will produce Java classes is a bit more complex than when calling Java classes from Python, because the methods in Java classes are strongly typed, while Python functions and methods are weakly typed. Thus, you must somehow tell **jythonc** that a Python method is intended to have a particular set of argument types and that its return value is a particular type. You accomplish this with the "@sig" string, which is placed right after the beginning of the Python method definition (this is the standard location for the Python documentation string). For example:

```
def returnArray(self):
  "@sig public java.lang.String[] returnArray()"
```
The Python definition doesn't specify any return type, but the @sig string gives the full type information about what is being passed and returned. The **jythonc** compiler uses this information to generate the correct Java code.

There's one other set of rules you must follow in order to get a successful compilation: you must inherit from a Java class or interface in your Python class (you do not need to specify the **@sig** signature for methods defined in the superclass/interface). If you do not do this, you won't get your desired methods – unfortunately, **jythonc** gives you no warnings or errors in this case, but you won't get what you want. If you don't see what's missing, it can be very frustrating.

In addition, you must import the appropriate java class and give the correct package specification. In the example below, **java** is imported so you must inherit from **java.lang.Object**, but you could also say **from java.lang import Object** and then you'd just inherit from **Object** without the package specification. Unfortunately, you don't get any warnings or errors if you get this wrong, so you must be patient and keep trying.

Here is an example of a Python class created to produce a Java class. This also introduces the '=**T**' directive for the makefile builder tool, which specifies a different target than the one that is normally used by the tool. In this case, the Python file is used to build a Java **.class** file, so the class file is the desired makefile target. To accomplish this, the default makefile command is replaced using the '=**M**' directive (notice how you can break across lines using '\'):

```
## 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):
```

```python
    "@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]
##~
```

First note that **PythonToJavaClass** is inherited from **java.lang.Object**; if you don't do this you will quietly get a Java class without the right signatures. You are not required to inherit from **Object**; any other Java class will do.

This class is designed to demonstrate different arguments and return values, to provide you with enough examples that you'll be able to easily create your own signature strings. The first three of these are fairly self-explanatory, but note the full qualification of the Java name in the signature string.

In **returnArray( )**, a Python array must be returned as a Java array. To do this, the Jython **array( )** function (from the **jarray** module) must be used, along with the type of the class for the resulting array. Any time you need to return an array to Java, you must use **array( )**, as seen in the methods **ints( )** and **doubles( )**.

The last methods show how to pass arguments in from Java. Basic types happen automatically as long as you specify them in the **@sig** string, but you must use objects and you cannot pass in primitives (that is, primitives must be ensconced in wrapper objects, such as **Integer**).

In **argIn3( )**, you can see that a Java **List** is transparently converted to something that behaves just like a Python array, but is not a true array because you cannot take a slice from it. If you want a true Python array, then you must create and pass a **PyArray** as in **argIn4( )**, where the slice is successful. Similarly, a Java **Map** must come in as a **PyDictionary** in order to be treated as a Python dictionary.

Here is the Java program to exercise the Java classes produced by the above Python code. This also introduces the '**=D**' directive for the makefile builder tool, which specifies a dependency in addition to those detected by the tool. Here, you can't compile **TestPythonToJavaClass.java** until **PythonToJavaClass.class** is available:

```
//- 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);
  }
} ///:~
```

For Python support, you'll usually only need to import the classes in
org.python.core. Everything else in the above example is fairly
straightforward, as **PythonToJavaClass** appears, from the Java side, to
be just another Java class. **dumpClassInfo( )** uses reflection to verify
that the method signatures specified in **PythonToJavaClass.py** have
come through properly.

Building the Java classes from the Python code

Part of the trick of creating Java classes from Python code is the @sig information in the method documentation strings. But there's a second problem which stems from the fact that Python has no "package" keyword – the Python equivalent of packages (modules) are implicitly created based on the file name. However, to bring the resulting class files into the Java program, **jythonc** must be given information about how to create the Java package for the Python code. This is done on the **jythonc** command line using the **--package** flag, followed by the package name you wish to produce (including the separation dots, just as you would give the package name using the **package** keyword in a Java program). This will put the resulting **.class** files in the appropriate subdirectory off of the current directory. Then you only need to import the package in your Java program, as shown above (you'll need '**.**' in your CLASSPATH in order to run it from the code directory).

Here are the **make** dependency rules that I used to build the above example (the backslashes at the ends of the lines are understood by **make** to be line continuations). These rules are encoded into the above Java and Python files using the comment syntax that's understood by my makefile builder tool:

```
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
```

The first target, **TestPythonToJavaClass.class**, depends on both **TestPythonToJavaClass.java** and the **PythonToJavaClass.class**, which is the Python code that's converted to a class file. This latter, in turn, depends on the Python source code. Note that it's important that the directory where the target lives be specified, so that the makefile will create the Java program with the minimum necessary amount of rebuilding.

The Java-Python Extension (JPE)

An alternative to Jython is the *Java-Python Extension* (JPE), which directly connects with your native C-Python implementation.

Jython runs entirely within the JavaVM, which produces two fundamental limitations: Jython cannot be called from CPython, and native Python extensions are not accessible from JPython. JPE is linked with the Python C libraries, so JPE can be called from C-Python, and native Python extensions can be called from Java through JPE.

If you need to access the features on your native platform, JPE might be the easiest solution. You can find JPE at http://www.arakne.com/jpe.htm.

Summary

This chapter has arguably gone much deeper into Jython than required to use the interpreter design pattern. Indeed, once you decide that you need to use interpreter and that you're not going to get lost inventing your own language, the solution of installing Jython is quite simple, and you can at least get started by following the **GreenHouseController** example.

Of course, that example is often too simple and you may need something more sophisticated, often requiring more interesting data to be passed back and forth. When I encountered the limited documentation, I felt it necessary to come up with a more thorough examination of Jython.

In the process, note that there could be another equally powerful design pattern lurking in here, which could perhaps be called *multiple languages*. This is based on the experience of having each language solve a certain class of problems better than the other; by combining languages you can solve problems much faster than with either language by itself. CORBA is another way to bridge across languages, and at the same time bridging between computers and operating systems.

To me, Python and Java present a very potent combination for program development because of Java's architecture and tool set, and Python's extremely rapid development (generally considered to be 5-10 times faster

than C++ or Java). Python is usually slower, however, but even if you end up re-coding parts of your program for speed, the initial fast development will allow you to more quickly flesh out the system and uncover and solve the critical sections. And often, the execution speed of Python is not a problem – in those cases it's an even bigger win. A number of commercial products already use Java and Jython, and because of the terrific productivity leverage I expect to see this happen more in the future.

Exercises

1. Modify **GreenHouseLanguage.py** so that it checks the times for the events and runs those events at the appropriate times.

2. Modify **GreenHouseLanguage.py** so that it calls a function for **action** instead of just printing a string.

3. Create a Swing application with a **JTextField** (where the user will enter commands) and a **JTextArea** (where the command results will be displayed). Connect to a **PythonInterpreter** object so that the output will be sent to the **JTextArea** (which should scroll). You'll need to locate the **PythonInterpreter** command that redirects the output to a Java stream.

4. Modify **GreenHouseLanguage.py** to add a master controller class (instead of the static array inside **Event**) and provide a **run( )** method for each of the subclasses. Each **run( )** should create and use an object from the standard Java library during its execution. Modify **GreenHouseController.java** to use this new class.

5. Modify the resulting **GreenHouseLanguage.py** from exercise two to produce Java classes (add the @sig documentation strings to produce the correct Java signatures, and create a makefile to build the Java **.class** files). Write a Java program that uses these classes.

Complex system states

StateMachine

While *State* has a way to allow the client programmer to change the implementation, *StateMachine* imposes a structure to automatically change the implementation from one object to the next. The current implementation represents the state that a system is in, and the system behaves differently from one state to the next (because it uses *State*). Basically, this is a "state machine" using objects.

The code that moves the system from one state to the next is often a *Template Method*, as seen in the following framework for a basic state machine. We start by defining a tagging interface for input objects:

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

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

Each state can be **run( )** to perform its behavior, and (in this design) you can also pass it an **Input** object so it can tell you what new state to move to based on that **Input**. The key distinction between this design and the next is that here, each **State** object decides what other states it can move to, based on the **Input**, whereas in the subsequent design all of the state transitions are held in a single table. Another way to put it is that here, each **State** object has its own little **State** table, and in the subsequent design there is a single master state transition table for the whole system.

```
//: 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);
} ///:~
```

The **StateMachine** keeps track of the current state, which is initialized
by the constructor. The **runAll( )** method takes an **Iterator** to a list of
Input objects (an **Iterator** is used here for convenience and simplicity;
the important issue is that the input information comes from
somewhere). This method not only moves to the next state, but it also
calls **run( )** for each state object – thus you can see it's an expansion of
the idea of the **State** pattern, since **run( )** does something different
depending on the state that the system is in.

```
//: 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();
    }
  }
} ///:~
```

I've also treated **runAll( )** as a template method. This is typical, but
certainly not required – you could concievably want to override it, but
typically the behavior change will occur in **State**'s **run( )** instead.

At this point the basic framework for this style of *StateMachine* (where
each state decides the next states) is complete. As an example, I'll use a
fancy mousetrap that can move through several states in the process of
trapping a mouse[11]. The mouse classes and information are stored in

the **mouse** package, including a class representing all the possible moves that a mouse can make, which will be the inputs to the state machine:

```
//: 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");
} ///:~
```

You'll note that **hashCode( )** and **equals( )** have been overriden so that **MouseAction** objects can be used as keys in a **HashMap**, but in the

first version of the mousetrap we won't do this. Also, each possible move by a mouse is enumerated as a **static MouseAction** object.

For creating test code, a sequence of mouse inputs is provided from a text file:

```
//:! 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
///:~
```

To read this file in a generic fashion, here is a general-purpose tool called **StringList**:

```
//: 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);
    }
  }
}
```

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

This **StringList** only holds **Object**s, just as an **ArrayList** does, so we
need an adapter to turn the **String**s into **MouseAction**s:

```
//: 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()));
  }
} ///:~
```

The **MouseMoveList** looks a bit like a decorator, and acts a bit like an
adapter. However, an adapter changes one interface to another, and a
decorator adds functionality or data. **MouseMoveList** changes the
contents of the container, so it might be thought of as a *Transformer*.

With these tools in place, it's now possible to create the first version of the
mousetrap program. Each **State** subclass defines it's **run( )** behavior,
and also establishes its next state with an **if-else** clause:

```
//: 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");
```

```java
    }
    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);
  }
} ///:~
```
The **StateMachine** class simply defines all the possible states as static objects, and also sets up the initial state. The **UnitTest** creates a **MouseTrap** and then tests it with all the inputs from a **MouseMoveList**.

While the use of **if-else** statements inside the **next( )** methods is perfectly reasonable, managing a large number of these could become difficult. Another approach is to create tables inside each **State** object defining the various next states based on the input.

Initially, this seems like it ought to be quite simple. You should be able to define a **static** table in each **State** subclass that defines the transitions in terms of the other **State** objects. However, it turns out that this approach generates cyclic initialization dependencies. To solve the problem, I've

had to delay the initialization of the tables until the first time that the **next( )** method is called for a particular **State** object. Initially, the **next( )** methods can appear a little strange because of this.

The **StateT** class is an implementation of **State** (so that the same **StateMachine** class can be used from the previous example) that adds a **Map** and a method to initialize the map from a two-dimensional array. The **next( )** method has a base-class implementation which must be called from the overridden derived class **next( )** methods after they test for a **null Map** (and initialize it if it's **null**):

```
//: 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() {
```

```java
      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);
  }
} ///:~
```
The rest of the code is identical – the difference is in the **next( )** methods
and the **StateT** class.

If you have to create and maintain a lot of **State** classes, this approach is
an improvement, since it's easier to quickly read and understand the state
transitions from looking at the table.

Exercises

1.   Modify **MouseTrap2Test.java** so that the state table
       information is loaded from an external text file, which only
       contains the state table information.

Table-Driven State Machine

The advantage of the previous design is that all the information about a state, including the state transition information, is located within the state class itself. This is generally a good design principle.

However, in a pure state machine, the machine can be completely represented by a single state-transition table. This has the advantage of locating all the information about the state machine in a single place, which means that you can more easily create and maintain the table based on a classic state-transition diagram.

The classic state-transition diagram uses a circle to represent each state, and lines from the state pointing to all states that state can transition into. Each transition line is annotated with conditions for transition and an action during transition. Here's what it looks like:

(Simple State Machine Diagram)

Goals:

- Direct translation of state diagram

- Vector of change: the state diagram representation

- Reasonable implementation

- No excess of states (you could represent every single change with a new state)

- Simplicity and flexibility

Observations:

- States are trivial – no information or functions/data, just an identity

- Not like the State pattern!

- The machine governs the move from state to state

- Similar to flyweight

- Each state may move to many others

- Condition & action functions must also be external to states

- Centralize description in a single table containing all variations, for ease of configuration

Example:

- State Machine & Table-Driven Code

- Implements a vending machine

- Uses several other patterns

- Separates common state-machine code from specific application (like template method)

- Each input causes a seek for appropriate solution (like chain of responsibility)

- Tests and transitions are encapsulated in function objects (objects that hold functions)

- Java constraint: methods are not first-class objects
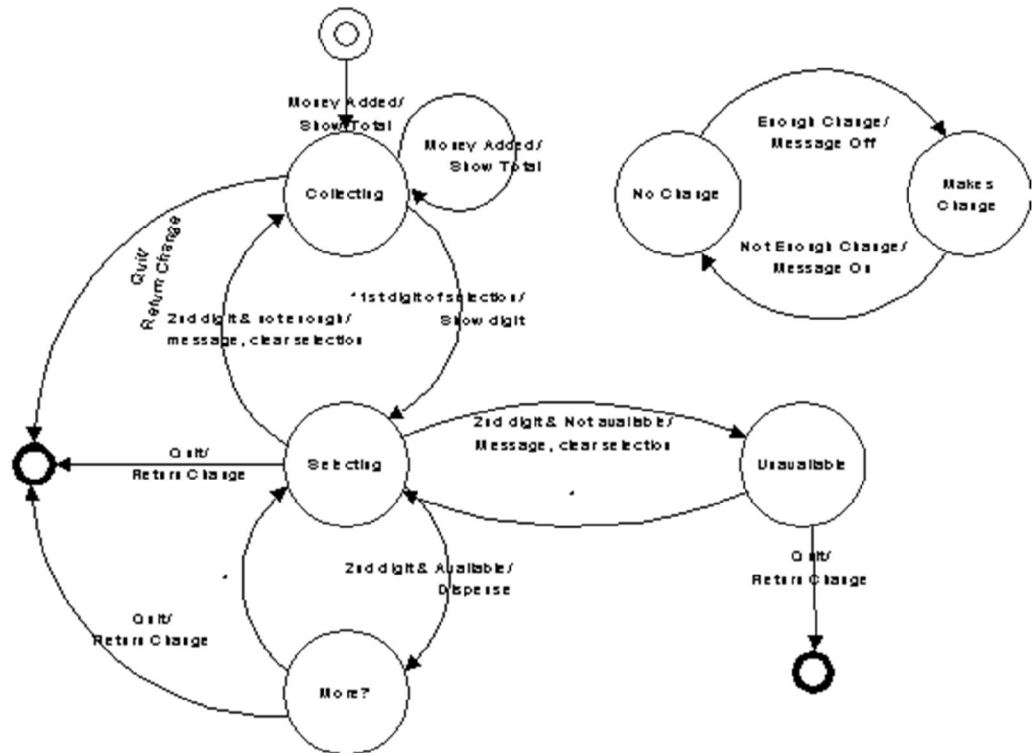
The State class

The **State** class is distinctly different from before, since it is really just a placeholder with a name. Thus it is not inherited from previous **State** classes:

```
//: statemachine2:State.java
package statemachine2;

public class State {
  private String name;
  public State(String nm) { name = nm; }
  public String toString() { return name; }
} ///:~
```

Conditions for transition

In the state transition diagram, an input is tested to see if it meets the condition necessary to transfer to the state under question. As before, the **Input** is just a tagging interface:

```
//: statemachine2:Input.java
// Inputs to a state machine
package statemachine2;

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

The **Condition** evaluates the **Input** to decide whether this row in the table is the correct transition:

```
//: statemachine2:Condition.java
// Condition function object for state machine
package statemachine2;

public interface Condition {
  boolean condition(Input i);
} ///:~
```

Transition actions

If the **Condition** returns **true**, then the transition to a new state is made, and as that transition is made some kind of action occurs (in the previous state machine design, this was the **run( )** method):

```
//: statemachine2:Transition.java
// Transition function object for state machine
package statemachine2;

public interface Transition {
  void transition(Input i);
} ///:~
```

The table

With these classes in place, we can set up a 3-dimensional table where each row completely describes a state. The first element in the row is the current state, and the rest of the elements are each a row indicating what the *type* of the input can be, the condition that must be satisfied in order for this state change to be the correct one, the action that happens during transition, and the new state to move into. Note that the **Input** object is not just used for its type, it is also a *Messenger* object that carries information to the **Condition** and **Transition** objects:

```
{ {CurrentState},
  {Input, Condition(Input), Transition(Input), Next},
  {Input, Condition(Input), Transition(Input), Next},
  {Input, Condition(Input), Transition(Input), Next},
   ...
}
```

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");
    }
} ///:~
```

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;
```

```java
    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;
```

```java
    }
  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];
```

```java
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 " +
```

```java
        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}},
      });
    }
} ///:~
```

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);
  }
} ///:~
```

Tools

Another approach, as your state machine gets bigger, is to use an automation tool whereby you configure a table and let the tool generate the state machine code for you. This can be created yourself using a language like Python, but there are also free, open-source tools such as *Libero*, at http://www.imatix.com.

Table-driven code: configuration flexibility

Table-driven code using anonymous inner classes

See **ListPerformance.java** example in TIJ from Chapter 9

Also **GreenHouse.java**

Exercises

1.	Apply **TransitionTable.java** to the "Washer" problem.

2.	Create a *StateMachine* system whereby the current state along with input information determines the next state that the system will be in. To do this, each state must store a reference back to the proxy object (the state controller) so that it can request the state change. Use a **HashMap** to create a table of states, where the key is a **String** naming the new state and the value is the new state object. Inside each state subclass override a method **nextState( )** that has its own state-transition table. The input to **nextState( )** should be a single word that comes from a text file containing one word per line.

3.	Modify the previous exercise so that the state machine can be configured by creating/modifying a single multi-dimensional array.

4.	Modify the "mood" exercise from the previous session so that it becomes a state machine using StateMachine.java

5.	Create an elevator state machine system using StateMachine.java

6.	Create a heating/air-conditioning system using StateMachine.java

7.	A *generator* is an object that produces other objects, just like a factory, except that the generator function doesn't require any arguments. Create a **MouseMoveGenerator** which produces correct **MouseMove** actions as outputs each time the generator function is called (that is, the mouse must move in the proper sequence, thus the possible moves are based on the previous move – it's another state machine). Add a method **iterator( )** to produce an iterator, but this method should take an **int** argument that specifies the number of moves to produce before **hasNext( )** returns **false**.

Pattern refactoring

This chapter will look at the process of solving a problem by applying design patterns in an evolutionary fashion. That is, a first cut design will be used for the initial solution, and then this solution will be examined and various design patterns will be applied to the problem (some of which will work, and some of which won't). The key question that will always be asked in seeking improved solutions is "what will change?"

This process is similar to what Martin Fowler talks about in his book *Refactoring: Improving the Design of Existing Code[12]* (although he tends to talk about pieces of code more than pattern-level designs). You start with a solution, and then when you discover that it doesn't continue to meet your needs, you fix it. Of course, this is a natural tendency but in computer programming it's been extremely difficult to accomplish with procedural programs, and the acceptance of the idea that we *can* refactor code and designs adds to the body of proof that object-oriented programming is "a good thing."

Simulating the trash recycler

The nature of this problem is that the trash is thrown unclassified into a single bin, so the specific type information is lost. But later, the specific type information must be recovered to properly sort the trash. In the initial solution, RTTI (described in Chapter 12 of *Thinking in Java, 2ⁿᵈ edition*) is used.

This is not a trivial design because it has an added constraint. That's what makes it interesting—it's more like the messy problems you're likely to encounter in your work. The extra constraint is that the trash arrives at the trash recycling plant all mixed together. The program must model the sorting of that trash. This is where RTTI comes in: you have a bunch of anonymous pieces of trash, and the program figures out exactly what type they are.

```java
//: 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();
  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); }
```

```java
    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);
 }
} ///:~
```

In the source code listings available for this book, this file will be placed in
the subdirectory **recyclea** that branches off from the subdirectory
refactor. The unpacking tool takes care of placing it into the correct
subdirectory. The reason for doing this is that this chapter rewrites this
particular example a number of times and by putting each version in its
own directory (using the default package in each directory so that
invoking the program is easy), the class names will not clash.

Several **ArrayList** objects are created to hold **Trash** references. Of
course, **ArrayList**s actually hold **Object**s so they'll hold anything at all.
The reason they hold **Trash** (or something derived from **Trash**) is only
because you've been careful to not put in anything except **Trash**. If you
do put something "wrong" into the **ArrayList**, you won't get any compile-
time warnings or errors—you'll find out only via an exception at run time.

When the **Trash** references are added, they lose their specific identities
and become simply **Object reference**s (they are *upcast*). However,
because of polymorphism the proper behavior still occurs when the
dynamically-bound methods are called through the **Iterator sorter**,
once the resulting **Object** has been cast back to **Trash**. **sumValue( )**
also takes an **Iterator** to perform operations on every object in the
ArrayList.

It looks silly to upcast the types of **Trash** into a container holding base
type references, and then turn around and downcast. Why not just put the

trash into the appropriate receptacle in the first place? (Indeed, this is the whole enigma of recycling). In this program it would be easy to repair, but sometimes a system's structure and flexibility can benefit greatly from downcasting.

The program satisfies the design requirements: it works. This might be fine as long as it's a one-shot solution. However, a useful program tends to evolve over time, so you must ask, "What if the situation changes?" For example, cardboard is now a valuable recyclable commodity, so how will that be integrated into the system (especially if the program is large and complicated). Since the above type-check coding in the **switch** statement could be scattered throughout the program, you must go find all that code every time a new type is added, and if you miss one the compiler won't give you any help by pointing out an error.

The key to the misuse of RTTI here is that *every type is tested.* If you're looking for only a subset of types because that subset needs special treatment, that's probably fine. But if you're hunting for every type inside a switch statement, then you're probably missing an important point, and definitely making your code less maintainable. In the next section we'll look at how this program evolved over several stages to become much more flexible. This should prove a valuable example in program design.

Improving the design

The solutions in *Design Patterns* are organized around the question "What will change as this program evolves?" This is usually the most important question that you can ask about any design. If you can build your system around the answer, the results will be two-pronged: not only will your system allow easy (and inexpensive) maintenance, but you might also produce components that are reusable, so that other systems can be built more cheaply. This is the promise of object-oriented programming, but it doesn't happen automatically; it requires thought and insight on your part. In this section we'll see how this process can happen during the refinement of a system.

The answer to the question "What will change?" for the recycling system is a common one: more types will be added to the system. The goal of the design, then, is to make this addition of types as painless as possible. In the recycling program, we'd like to encapsulate all places where specific type information is mentioned, so (if for no other reason) any changes can be localized to those encapsulations. It turns out that this process also cleans up the rest of the code considerably.

"Make more objects"

This brings up a general object-oriented design principle that I first heard spoken by Grady Booch: "If the design is too complicated, make more objects." This is simultaneously counterintuitive and ludicrously simple, and yet it's the most useful guideline I've found. (You might observe that "making more objects" is often equivalent to "add another level of indirection.") In general, if you find a place with messy code, consider what sort of class would clean that up. Often the side effect of cleaning up the code will be a system that has better structure and is more flexible.

Consider first the place where **Trash** objects are created, which is a **switch** statement inside **main( )**:

```
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));
  }
```

This is definitely messy, and also a place where you must change code whenever a new type is added. If new types are commonly added, a better solution is a single method that takes all of the necessary information and produces a reference to an object of the correct type, already upcast to a trash object. In *Design Patterns* this is broadly referred to as a *creational pattern* (of which there are several). The specific pattern that will be applied here is a variant of the *Factory Method*. Here, the factory method is a **static** member of **Trash**, but more commonly it is a method that is overridden in the derived class.

The idea of the factory method is that you pass it the essential information it needs to know to create your object, then stand back and wait for the reference (already upcast to the base type) to pop out as the return value. From then on, you treat the object polymorphically. Thus, you never even

need to know the exact type of object that's created. In fact, the factory method hides it from you to prevent accidental misuse. If you want to use the object without polymorphism, you must explicitly use RTTI and casting.

But there's a little problem, especially when you use the more complicated approach (not shown here) of making the factory method in the base class and overriding it in the derived classes. What if the information required in the derived class requires more or different arguments? "Creating more objects" solves this problem. To implement the factory method, the **Trash** class gets a new method called **factory**. To hide the creational data, there's a new class called **Messenger** that carries all of the necessary information for the **factory** method to create the appropriate **Trash** object (we've started referring to *Messenger* as a design pattern, but it's simple enough that you may not choose to elevate it to that status). Here's a simple implementation of **Messenger**:

```
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;
  }
}
```

A **Messenger** object's only job is to hold information for the **factory( )** method. Now, if there's a situation in which **factory( )** needs more or different information to create a new type of **Trash** object, the **factory( )** interface doesn't need to be changed. The **Messenger** class can be changed by adding new data and new constructors, or in the more typical object-oriented fashion of subclassing.

The **factory( )** method for this simple example looks like this:

```
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);
  }
 }
```

Here, the determination of the exact type of object is simple, but you can imagine a more complicated system in which **factory( )** uses an elaborate algorithm. The point is that it's now hidden away in one place, and you know to come to this place when you add new types.

The creation of new objects is now much simpler in **main( )**:

```
   for(int i = 0; i < 30; i++)
     bin.add(
       Trash.factory(
        new Messenger(
          rand.nextInt(Messenger.MAX_NUM),
          rand.nextDouble() * 100)));
```

A **Messenger** object is created to pass the data into **factory( )**, which in turn produces some kind of **Trash** object on the heap and returns the reference that's added to the **ArrayList bin**. Of course, if you change the quantity and type of argument, this statement will still need to be modified, but that can be eliminated if the creation of the **Messenger** object is automated. For example, an **ArrayList** of arguments can be passed into the constructor of a **Messenger** object (or directly into a **factory( )** call, for that matter). This requires that the arguments be parsed and checked at run time, but it does provide the greatest flexibility.

You can see from this code what "vector of change" problem the factory is responsible for solving: if you add new types to the system (the change), the only code that must be modified is within the factory, so the factory isolates the effect of that change.

A pattern for prototyping creation

A problem with the design above is that it still requires a central location where all the types of the objects must be known: inside the **factory( )** method. If new types are regularly being added to the system, the **factory( )** method must be changed for each new type. When you discover something like this, it is useful to try to go one step further and move *all* of the information about the type—including its creation—into the class representing that type. This way, the only thing you need to do to add a new type to the system is to inherit a single class.

To move the information concerning type creation into each specific type of **Trash**, the "prototype" pattern (from the *Design Patterns* book) will be used. The general idea is that you have a master sequence of objects, one of each type you're interested in making. The objects in this sequence are used *only* for making new objects, using an operation that's not unlike the **clone( )** scheme built into Java's root class **Object**. In this case, we'll name the cloning method **tClone( )**. When you're ready to make a new object, presumably you have some sort of information that establishes the type of object you want to create, then you move through the master sequence comparing your information with whatever appropriate information is in the prototype objects in the master sequence. When you find one that matches your needs, you clone it.

In this scheme there is no hard-coded information for creation. Each object knows how to expose appropriate information and how to clone itself. Thus, the **factory( )** method doesn't need to be changed when a new type is added to the system.

One approach to the problem of prototyping is to add a number of methods to support the creation of new objects. However, in Java 1.1 there's already support for creating new objects if you have a reference to the **Class** object. With Java 1.1 *reflection* (introduced in Chapter 12 of *Thinking in Java, 2nd edition*) you can call a constructor even if you have only a reference to the **Class** object. This is the perfect solution for the prototyping problem.

The list of prototypes will be represented indirectly by a list of references to all the **Class** objects you want to create. In addition, if the prototyping fails, the **factory( )** method will assume that it's because a particular

Class object wasn't in the list, and it will attempt to load it. By loading the prototypes dynamically like this, the **Trash** class doesn't need to know what types it is working with, so it doesn't need any modifications when you add new types. This allows it to be easily reused throughout the rest of the chapter.

```
//: 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:
```

```java
  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);
  }
} ///:~
```

The basic **Trash** class and **sumValue( )** remain as before. The rest of the class supports the prototyping pattern. You first see two inner classes (which are made **static**, so they are inner classes only for code organization purposes) describing exceptions that can occur. This is followed by an **ArrayList** called **trashTypes**, which is used to hold the **Class** references.

In **Trash.factory( )**, the **String** inside the **Messenger** object **id** (a different version of the **Messenger** class than that of the prior discussion) contains the type name of the **Trash** to be created; this **String** is compared to the **Class** names in the list. If there's a match, then that's the object to create. Of course, there are many ways to determine what object you want to make. This one is used so that information read in from a file can be turned into objects.

Once you've discovered which kind of **Trash** to create, then the reflection methods come into play. The **getConstructor( )** method takes an argument that's an array of **Class** references. This array represents the arguments, in their proper order, for the constructor that you're looking for. Here, the array is dynamically created using the Java 1.1 array-creation syntax:

```
new Class[] {double.class}
```

This code assumes that every **Trash** type has a constructor that takes a **double** (and notice that **double.class** is distinct from **Double.class**). It's also possible, for a more flexible solution, to call **getConstructors( )**, which returns an array of the possible constructors.

What comes back from **getConstructor( )** is a reference to a **Constructor** object (part of **java.lang.reflect**). You call the constructor dynamically with the method **newInstance( )**, which takes an array of **Object** containing the actual arguments. This array is again created using the Java 1.1 syntax:

```
new Object[]{new Double(Messenger.data)}
```

In this case, however, the **double** must be placed inside a wrapper class so that it can be part of this array of objects. The process of calling **newInstance( )** extracts the **double**, but you can see it is a bit confusing—an argument might be a **double** or a **Double**, but when you make the call you must always pass in a **Double**. Fortunately, this issue exists only for the primitive types.

Once you understand how to do it, the process of creating a new object given only a **Class** reference is remarkably simple. Reflection also allows you to call methods in this same dynamic fashion.

Of course, the appropriate **Class** reference might not be in the **trashTypes** list. In this case, the **return** in the inner loop is never

executed and you'll drop out at the end. Here, the program tries to rectify the situation by loading the **Class** object dynamically and adding it to the **trashTypes** list. If it still can't be found something is really wrong, but if the load is successful then the **factory** method is called recursively to try again.

As you'll see, the beauty of this design is that this code doesn't need to be changed, regardless of the different situations it will be used in (assuming that all **Trash** subclasses contain a constructor that takes a single **double** argument).

Trash subclasses

To fit into the prototyping scheme, the only thing that's required of each new subclass of **Trash** is that it contain a constructor that takes a **double** argument. Java reflection handles everything else.

Here are the different types of **Trash**, each in their own file but part of the **Trash** package (again, to facilitate reuse within the chapter):

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

And here's a new type of **Trash**:

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

You can see that, other than the constructor, there's nothing special about any of these classes.

Parsing **Trash** from an external file

The information about **Trash** objects will be read from an outside file. The file has all of the necessary information about each piece of trash on a single line in the form **Trash:weight**, such as:

```
//:! 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
///:~
```

Note that the class path must be included when giving the class names, otherwise the class will not be found.

This file is read using the previously-defined **StringList** tool, and each line is picked aparat using the **String** method **indexOf( )** to produce the index of the '**:**'. This is first used with the **String** method **substring( )** to extract the name of the trash type, and next to get the weight that is turned into a **double** with the **static Double.valueOf( )** method. The **trim( )** method removes white space at both ends of a string.

The **Trash** parser is placed in a separate file since it will be reused throughout this chapter:

```
//: 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));
  }
} ///:~
```

In **RecycleA.java**, an **ArrayList** was used to hold the **Trash** objects. However, other types of containers can be used as well. To allow for this, the first version of **fillBin( )** takes a reference to a **Fillable**, which is simply an **interface** that supports a method called **addTrash( )**:

```
//: refactor:trash:Fillable.java
// Any object that can be filled with Trash.
package refactor.trash;
```

```
public interface Fillable {
  void addTrash(Trash t);
} ///:~
```

Anything that supports this interface can be used with **fillBin**. Of course, **Collection** doesn't implement **Fillable**, so it won't work. Since **Collection** is used in most of the examples, it makes sense to add a second overloaded **fillBin( )** method that takes a **Collection**. Any **Collection** can then be used as a **Fillable** object using an adapter class:

```
//: 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); }
} ///:~
```

You can see that the only job of this class is to connect **Fillable**'s **addTrash( )** method to **Collection's add( )**. With this class in hand, the overloaded **fillBin( )** method can be used with a **Collection** in **ParseTrash.java**:

```
public static void
fillBin(String filePath, Collection bin) {
  fillBin(filePath, new FillableCollection(bin));
}
```

This approach works for any container class that's used frequently. Alternatively, the container class can provide its own adapter that implements **Fillable**. (You'll see this later, in **DynaTrash.java**.)

Recycling with prototyping

Now you can see the revised version of **RecycleA.java** using the prototyping technique:

```java
//: 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);
```

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

All of the **Trash** objects, as well as the **ParseTrash** and support classes, are now part of the package **refactor.trash**, so they are simply imported.

The process of opening the data file containing **Trash** descriptions and the parsing of that file have been wrapped into the **static** method **ParseTrash.fillBin( )**, so now it's no longer a part of our design focus. You will see that throughout the rest of the chapter, no matter what new classes are added, **ParseTrash.fillBin( )** will continue to work without change, which indicates a good design.

In terms of object creation, this design does indeed severely localize the changes you need to make to add a new type to the system. However, there's a significant problem in the use of RTTI that shows up clearly here. The program seems to run fine, and yet it never detects any cardboard, even though there is cardboard in the list! This happens *because* of the use of RTTI, which looks for only the types that you tell it to look for. The clue that RTTI is being misused is that *every type in the system* is being tested, rather than a single type or subset of types. As you will see later, there are ways to use polymorphism instead when you're testing for every type. But if you use RTTI a lot in this fashion, and you add a new type to your system, you can easily forget to make the necessary changes in your program and produce a difficult-to-find bug. So it's worth trying to eliminate RTTI in this case, not just for aesthetic reasons—it produces more maintainable code.

Abstracting usage

With creation out of the way, it's time to tackle the remainder of the design: where the classes are used. Since it's the act of sorting into bins that's particularly ugly and exposed, why not take that process and hide it inside a class? This is the principle of "If you must do something ugly, at least localize the ugliness inside a class." It looks like this:
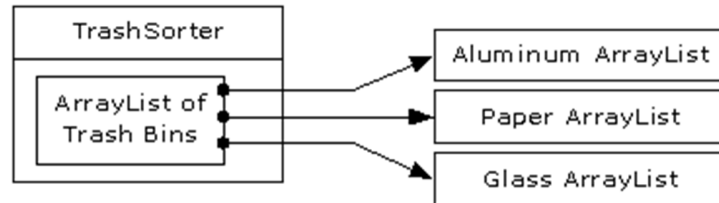
The **TrashSorter** object initialization must now be changed whenever a new type of **Trash** is added to the model. You could imagine that the **TrashSorter** class might look something like this:

```
class TrashSorter extends ArrayList {
  void sort(Trash t) { /* ... */ }
}
```

That is, **TrashSorter** is an **ArrayList** of references to **ArrayList**s of **Trash** references, and with **add( )** you can install another one, like so:

```
TrashSorter ts = new TrashSorter();
ts.add(new ArrayList());
```

Now, however, **sort( )** becomes a problem. How does the statically-coded method deal with the fact that a new type has been added? To solve this, the type information must be removed from **sort( )** so that all it needs to do is call a generic method that takes care of the details of type. This, of course, is another way to describe a dynamically-bound method. So **sort( )** will simply move through the sequence and call a dynamically-bound method for each **ArrayList**. Since the job of this method is to grab the pieces of trash it is interested in, it's called **grab(Trash)**. The structure now looks like:
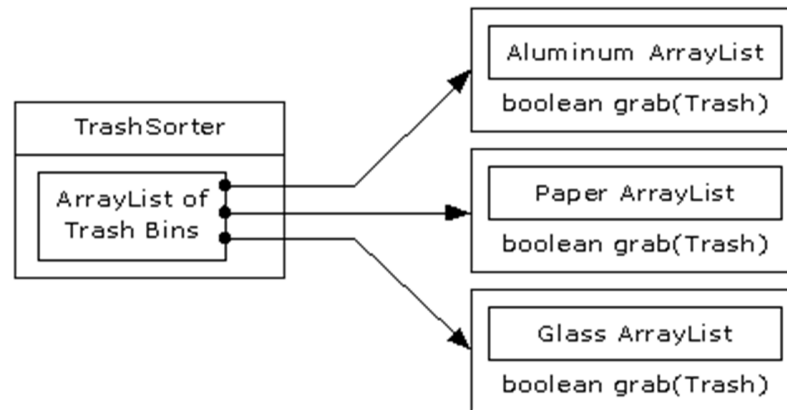
TrashSorter needs to call each **grab( )** method and get a different result depending on what type of **Trash** the current **ArrayList** is holding. That is, each **ArrayList** must be aware of the type it holds. The classic approach to this problem is to create a base "**Trash** bin" class and inherit a new derived class for each different type you want to hold. If Java had a parameterized type mechanism that would probably be the most straightforward approach. But rather than hand-coding all the classes that such a mechanism should be building for us, further observation can produce a better approach.

A basic OOP design principle is "Use data members for variation in state, use polymorphism for variation in behavior." Your first thought might be that the **grab( )** method certainly behaves differently for an **ArrayList** that holds **Paper** than for one that holds **Glass**. But what it does is strictly dependent on the type, and nothing else. This could be interpreted as a different state, and since Java has a class to represent type (**Class**) this can be used to determine the type of **Trash** a particular **Tbin** will hold.

The constructor for this **Tbin** requires that you hand it the **Class** of your choice. This tells the **ArrayList** what type it is supposed to hold. Then the **grab( )** method uses **Class BinType** and RTTI to see if the **Trash** object you've handed it matches the type it's supposed to grab.

Here is the new version of the program:

```
//: refactor:recycleb:RecycleB.java
// Containers that grab objects of interest.
package refactor.recycleb;
import refactor.trash.*;
import java.util.*;
```

```java
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 contains a **Class** reference **type** which establishes in the constructor what what type it should grab. The **grab()** method checks this type against the object you pass it. Note that in this design, **grab()** only accepts **Trash** objects so you get compile-time type checking on the base type, but you could also just accept **Object** and it would still work.

TbinList holds a set of **Tbin** references, so that **sort( )** can iterate through the **Tbin**s when it's looking for a match for the **Trash** object you've handed it. If it doesn't find a match, it creates a new **Tbin** for the type that hasn't been found, and makes a recursive call to itself – the next time around, the new bin will be found.

Notice the genericity of this code: it doesn't change at all if new types are added. If the bulk of your code doesn't need changing when a new type is added (or some other change occurs) then you have an easily extensible system.

Multiple dispatching

The above design is certainly satisfactory. Adding new types to the system consists of adding or modifying distinct classes without causing code changes to be propagated throughout the system. In addition, RTTI is not "misused" as it was in **RecycleA.java**. However, it's possible to go one step further and take a purist viewpoint about RTTI and say that it should be eliminated altogether from the operation of sorting the trash into bins.

To accomplish this, you must first take the perspective that all type-dependent activities—such as detecting the type of a piece of trash and putting it into the appropriate bin—should be controlled through polymorphism and dynamic binding.

The previous examples first sorted by type, then acted on sequences of elements that were all of a particular type. But whenever you find yourself picking out particular types, stop and think. The whole idea of

polymorphism (dynamically-bound method calls) is to handle type-specific information for you. So why are you hunting for types?

The answer is something you probably don't think about: Java performs only single dispatching. That is, if you are performing an operation on more than one object whose type is unknown, Java will invoke the dynamic binding mechanism on only one of those types. This doesn't solve the problem, so you end up detecting some types manually and effectively producing your own dynamic binding behavior.

The solution is called *multiple dispatching*, which means setting up a configuration such that a single method call produces more than one dynamic method call and thus determines more than one type in the process. To get this effect, you need to work with more than one type hierarchy: you'll need a type hierarchy for each dispatch. The following example works with two hierarchies: the existing **Trash** family and a hierarchy of the types of trash bins that the trash will be placed into. This second hierarchy isn't always obvious and in this case it needed to be created in order to produce multiple dispatching (in this case there will be only two dispatches, which is referred to as *double dispatching*).

Implementing the double dispatch

Remember that polymorphism can occur only via method calls, so if you want double dispatching to occur, there must be two method calls: one used to determine the type within each hierarchy. In the Trash hierarchy there will be a new method called addToBin( ), which takes an argument of an array of TypedBin. It uses this array to step through and try to add itself to the appropriate bin, and this is where you'll see the double

dispatch.
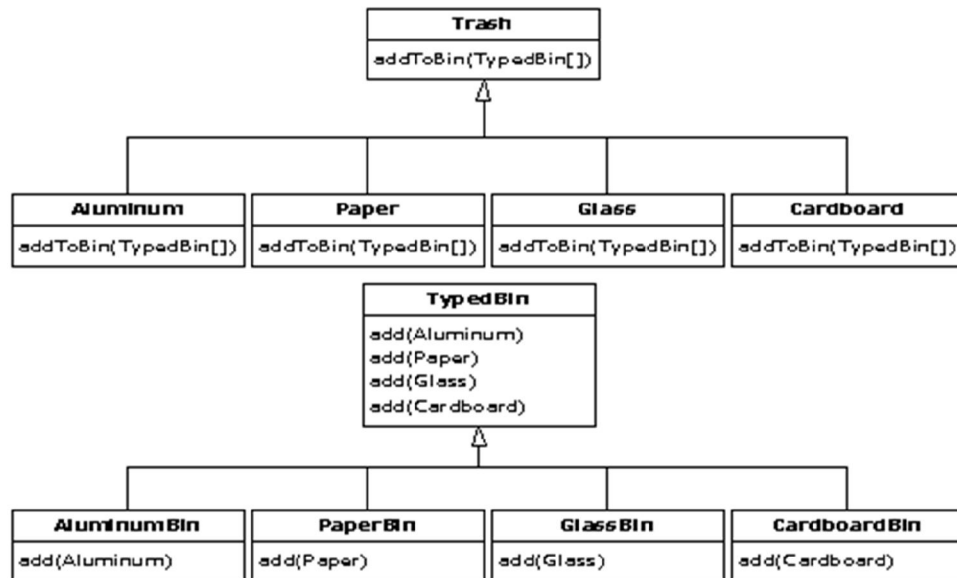
The new hierarchy is TypedBin, and it contains its own method called add( ) that is also used polymorphically. But here's an additional twist: add( ) is overloaded to take arguments of the different types of trash. So an essential part of the double dispatching scheme also involves overloading.

Redesigning the program produces a dilemma: it's now necessary for the base class **Trash** to contain an **addToBin( )** method. One approach is to copy all of the code and change the base class. Another approach, which you can take when you don't have control of the source code, is to put the **addToBin( )** method into an **interface**, leave **Trash** alone, and inherit new specific types of **Aluminum**, **Paper**, **Glass**, and **Cardboard**. This is the approach that will be taken here.

Most of the classes in this design must be **public**, so they are placed in their own files. Here's the interface:

```
//: 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);
} ///:~
```

In each particular subtype of **Aluminum**, **Paper**, **Glass,** and **Cardboard**, the **addToBin( )** method in the **interface TypedBinMember** is implemented, but it *looks* like the code is exactly the same in each case:

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

The code in each **addToBin( )** calls **add( )** for each **TypedBin** object in the array. But notice the argument: **this**. The type of **this** is different for each subclass of **Trash**, so the code is different. (Although this code will benefit if a parameterized type mechanism is ever added to Java.) So this is the first part of the double dispatch, because once you're inside this method you know you're **Aluminum**, or **Paper**, etc. During the call to **add( )**, this information is passed via the type of **this**. The compiler resolves the call to the proper overloaded version of **add( )**. But since **tb[i]** produces a reference to the base type **TypedBin**, this call will end

up calling a different method depending on the type of **TypedBin** that's currently selected. That is the second dispatch.

Here's the base class for **TypedBin**:

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

You can see that the overloaded **add( )** methods all return **false.** If the method is not overloaded in a derived class, it will continue to return **false**, and the caller (**addToBin( )**, in this case) will assume that the current **Trash** object has not been added successfully to a container, and continue searching for the right container.

In each of the subclasses of **TypedBin**, only one overloaded method is overridden, according to the type of bin that's being created. For example, **CardboardBin** overrides **add(DDCardboard)**. The overridden method adds the trash object to its container and returns **true**, while all

the rest of the **add( )** methods in **CardboardBin** continue to return **false**, since they haven't been overridden. This is another case in which a parameterized type mechanism in Java would allow automatic generation of code. (With C++ **template**s, you wouldn't have to explicitly write the subclasses or place the **addToBin( )** method in **Trash**.)

Since for this example the trash types have been customized and placed in a different directory, you'll need a different trash data file to make it work. Here's a possible **DDTrash.dat**:

```
//:! 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
```

///:~

Here's the rest of the program:

```
//: 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 encapsulates all of the different types of **TypedBin**s, along with the **sortIntoBins( )** method, which is where all the double dispatching takes place. You can see that once the structure is set up, sorting into the various **TypedBin**s is remarkably easy. In addition, the efficiency of two dynamic method calls is probably better than any other way you could sort.

Notice the ease of use of this system in **main( )**, as well as the complete independence of any specific type information within **main( )**. All other methods that talk only to the **Trash** base-class interface will be equally invulnerable to changes in **Trash** types.

The changes necessary to add a new type are relatively isolated: you modify **TypedBin**, inherit the new type of **Trash** with its **addToBin( )** method, then inherit a new **TypedBin** (this is really just a copy and simple edit), and finally add a new type into the aggregate initialization for **TrashBinSet**.

The *Visitor* pattern

Now consider applying a design pattern that has an entirely different goal to the trash sorting problem.

For this pattern, we are no longer concerned with optimizing the addition of new types of **Trash** to the system. Indeed, this pattern makes adding a new type of **Trash** *more* complicated. The assumption is that you have a primary class hierarchy that is fixed; perhaps it's from another vendor and you can't make changes to that hierarchy. However, you'd like to add new polymorphic methods to that hierarchy, which means that normally you'd have to add something to the base class interface. So the dilemma is that you need to add methods to the base class, but you can't touch the base class. How do you get around this?

The design pattern that solves this kind of problem is called a "visitor" (the final one in the *Design Patterns* book), and it builds on the double dispatching scheme shown in the last section.

The visitor pattern allows you to extend the interface of the primary type by creating a separate class hierarchy of type **Visitor** to virtualize the operations performed upon the primary type. The objects of the primary type simply "accept" the visitor, then call the visitor's dynamically-bound method. It looks like this:

```
┌─────────────────────┐
│       Trash         │
├─────────────────────┤
│  accept(Visitor)    │
└─────────────────────┘
```

```
┌──────────────────────┐  ┌──────────────────────┐  ┌──────────────────────┐
│      Aluminum        │  │        Paper         │  │        Glass         │
├──────────────────────┤  ├──────────────────────┤  ├──────────────────────┤
│  accept(Visitor v) { │  │  accept(Visitor v) { │  │  accept(Visitor v) { │
│     v.visit(this);   │  │     v.visit(this);   │  │     v.visit(this);   │
│  }                   │  │  }                   │  │  }                   │
└──────────────────────┘  └──────────────────────┘  └──────────────────────┘
```

```
┌─────────────────────┐
│      Visitor        │
├─────────────────────┤
│  visit(Aluminum)    │
│  visit(Paper)       │
│  visit(Glass)       │
└─────────────────────┘
```

```
┌──────────────────────┐  ┌──────────────────────┐  ┌──────────┐
│     PriceVisitor     │  │    WeightVisitor     │  │   Etc.   │
├──────────────────────┤  ├──────────────────────┤  └──────────┘
│ visit(Aluminum) {    │  │ visit(Aluminum) {    │
│  // Perform Aluminum-│  │  // Perform Aluminum-│
│  // specific work    │  │  // specific work    │
│ }                    │  │ }                    │
│ visit(Paper) {       │  │ visit(Paper) {       │
│  // Perform Paper-   │  │  // Perform Paper-   │
│  // specific work    │  │  // specific work    │
│ }                    │  │ }                    │
│ visit(Glass) {       │  │ visit(Glass) {       │
│  // Perform Glass-   │  │  // Perform Glass-   │
│  // specific work    │  │  // specific work    │
│ }                    │  │ }                    │
└──────────────────────┘  └──────────────────────┘
```

Now, if **v** is a **Visitable** reference to an **Aluminum** object, the code:

```
PriceVisitor pv = new PriceVisitor();
v.accept(pv);
```

uses double dispatching to cause two polymorphic method calls: the first one to select **Aluminum**'s version of **accept( )**, and the second one within **accept( )** when the specific version of **visit( )** is called dynamically using the base-class **Visitor** reference **v**.

This configuration means that new functionality can be added to the system in the form of new subclasses of **Visitor**. The **Trash** hierarchy doesn't need to be touched. This is the prime benefit of the visitor pattern: you can add new polymorphic functionality to a class hierarchy without touching that hierarchy (once the **accept( )** methods have been installed). Note that the benefit is helpful here but not exactly what we started out to accomplish, so at first blush you might decide that this isn't the desired solution.

But look at one thing that's been accomplished: the visitor solution avoids sorting from the master **Trash** sequence into individual typed sequences. Thus, you can leave everything in the single master sequence and simply pass through that sequence using the appropriate visitor to accomplish the goal. Although this behavior seems to be a side effect of visitor, it does give us what we want (avoiding RTTI).

The double dispatching in the visitor pattern takes care of determining both the type of **Trash** and the type of **Visitor**. In the following example, there are two implementations of **Visitor**: **PriceVisitor** to both determine and sum the price, and **WeightVisitor** to keep track of the weights.

You can see all of this implemented in the new, improved version of the recycling program.

As with **DoubleDispatch.java**, the **Trash** class is left alone and a new interface is created to add the **accept( )** method:

```
//: 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);
} ///:~
```

Since there's nothing concrete in the **Visitor** base class, it can be created as an **interface**:

```
//: 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);
} ///:~
```

A Reflective Decorator

At this point, you *could* follow the same approach that was used for double dispatching and create new subtypes of **Aluminum**, **Paper**, **Glass,** and **Cardboard** that implement the **accept( )** method. For example, the new **Visitable Aluminum** would look like this:

```
//: 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);
  }
} ///:~
```

However, we seem to be encountering an "explosion of interfaces:" basic **Trash**, special versions for double dispatching, and now more special versions for visitor. Of course, this "explosion of interfaces" is arbitrary—one could simply put the additional methods in the **Trash** class. If we ignore that we can instead see an opportunity to use the *Decorator* pattern: it seems like it should be possible to create a *Decorator* that can be wrapped around an ordinary **Trash** object and will produce the same

interface as **Trash** and add the extra **accept( )** method. In fact, it's a perfect example of the value of *Decorator.*

The double dispatch creates a problem, however. Since it relies on overloading of both **accept( )** and **visit( )**, it would seem to require specialized code for each different version of the **accept( )** method. With C++ templates, this would be fairly easy to accomplish (since templates automatically generate type-specialized code) but Java has no such mechanism—at least it does not appear to. However, reflection allows you to determine type information at run time, and it turns out to solve many problems that would seem to require templates (albeit not as simply). Here's the decorator that does the trick[13]:

```
//: 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();
  }
  public void accept(Visitor v) {
    try {
      dispatch.invoke(v, new Object[]{delegate});
```

```
    } catch(Exception e) {
      throw new RuntimeException(e);
    }
  }
} ///:~
```

[[Description of Reflection use]]

[[Note that a Dynamic Proxy might also be applied here.]]

The only other tool we need is a new type of **Fillable** adapter that automatically decorates the objects as they are being created from the original **Trash.dat** file. But this might as well be a decorator itself, decorating any kind of **Fillable**:

```
//: 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));
  }
} ///:~
```

Now you can wrap it around any kind of existing **Fillable**, or any new ones that haven't yet been created.

The rest of the program creates specific **Visitor** types and sends them through a single list of **Trash** objects:

```
//: refactor:trashvisitor:TrashVisitor.java
// The "visitor" pattern with VisitableDecorators.
package refactor.trashvisitor;
import refactor.trash.*;
import java.util.*;
import junit.framework.*;
```

```java
// 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 {
```

```java
  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);
  }
} ///:~
```

In **Test( )**, note how visitability is added by simply creating a different kind of bin using the decorator. Also notice that the **FillableCollection** adapter has the appearance of being used as a decorator (for **ArrayList**) in this situation. However, it completely changes the interface of the **ArrayList**, whereas the definition of *Decorator* is that the interface of the decorated class must still be there after decoration.

Note that the shape of the client code (shown in the **Test** class) has changed again, from the original approaches to the problem. Now there's only a single **Trash** bin. The two **Visitor** objects are accepted into every element in the sequence, and they perform their operations. The visitors keep their own internal data to tally the total weights and prices.

Finally, there's no run time type identification other than the inevitable cast to **Trash** when pulling things out of the sequence. This, too, could be eliminated with the implementation of parameterized types in Java.

One way you can distinguish this solution from the double dispatching solution described previously is to note that, in the double dispatching solution, only one of the overloaded methods, **add( )**, was overridden when each subclass was created, while here *each* one of the overloaded **visit( )** methods is overridden in every subclass of **Visitor**.

More coupling?

There's a lot more code here, and there's definite coupling between the **Trash** hierarchy and the **Visitor** hierarchy. However, there's also high cohesion within the respective sets of classes: they each do only one thing

(**Trash** describes Trash, while **Visitor** describes actions performed on **Trash**), which is an indicator of a good design. Of course, in this case it works well only if you're adding new **Visitor**s, but it gets in the way when you add new types of **Trash**.

Low coupling between classes and high cohesion within a class is definitely an important design goal. Applied mindlessly, though, it can prevent you from achieving a more elegant design. It seems that some classes inevitably have a certain intimacy with each other. These often occur in pairs that could perhaps be called *couplets*; for example, containers and iterators. The **Trash-Visitor** pair above appears to be another such couplet.

RTTI considered harmful?

Various designs in this chapter attempt to remove RTTI, which might give you the impression that it's "considered harmful" (the condemnation used for poor, ill-fated **goto**, which was thus never put into Java). This isn't true; it is the *misuse* of RTTI that is the problem. The reason our designs removed RTTI is because the misapplication of that feature prevented extensibility, while the stated goal was to be able to add a new type to the system with as little impact on surrounding code as possible. Since RTTI is often misused by having it look for every single type in your system, it causes code to be non-extensible: when you add a new type, you have to go hunting for all the code in which RTTI is used, and if you miss any you won't get help from the compiler.

However, RTTI doesn't automatically create non-extensible code. Let's revisit the trash recycler once more. This time, a new tool will be introduced, which I call a **TypeMap**. It contains a **HashMap** that holds **ArrayList**s, but the interface is simple: you can **add( )** a new object, and you can **get( )** an **ArrayList** containing all the objects of a particular type. The keys for the contained **HashMap** are the types in the associated **ArrayList**. The beauty of this design is that the **TypeMap** dynamically adds a new pair whenever it encounters a new type, so whenever you add a new type to the system (even if you add the new type at run time), it adapts.

Our example will again build on the structure of the **Trash** types in **package refactor.Trash** (and the **Trash.dat** file used there can be used here without change):

//: refactor:dynatrash:DynaTrash.java

```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);
  }
} ///:~
```

Although powerful, the definition for **TypeMap** is simple. It contains a **HashMap**, and the **add( )** method does most of the work. When you **add( )** a new object, the reference for the **Class** object for that type is extracted. This is used as a key to determine whether an **ArrayList** holding objects of that type is already present in the **HashMap**. If so, that **ArrayList** is extracted and the object is added to the **ArrayList**. If not, the **Class** object and a new **ArrayList** are added as a key-value pair.

You can get an **Iterator** of all the **Class** objects from **keys( )**, and use each **Class** object to fetch the corresponding **ArrayList** with **get( )**. And that's all there is to it.

The **filler( )** method is interesting because it takes advantage of the design of **ParseTrash.fillBin( )**, which doesn't just try to fill an **ArrayList** but instead anything that implements the **Fillable** interface with its **addTrash( )** method. All **filler( )** needs to do is to return a reference to an **interface** that implements **Fillable**, and then this reference can be used as an argument to **fillBin( )** like this:

```
ParseTrash.fillBin("Trash.dat", bin.filler());
```

To produce this reference, an *anonymous inner class* (described in Chapter 8 of *Thinking in Java, 2nd edition*) is used. You never need a named class to implement **Fillable**, you just need a reference to an object of that class, thus this is an appropriate use of anonymous inner classes.

An interesting thing about this design is that even though it wasn't created to handle the sorting, **fillBin( )** is performing a sort every time it inserts a **Trash** object into **bin**.

Much of **class DynaTrash** should be familiar from the previous examples. This time, instead of placing the new **Trash** objects into a **bin** of type **ArrayList**, the **bin** is of type **TypeMap**, so when the trash is thrown into **bin** it's immediately sorted by **TypeMap**'s internal sorting mechanism. Stepping through the **TypeMap** and operating on each individual **ArrayList** becomes a simple matter.

As you can see, adding a new type to the system won't affect this code at all, and the code in **TypeMap** is completely independent. This is certainly the smallest solution to the problem, and arguably the most elegant as well. It does rely heavily on RTTI, but notice that each key-value pair in the **HashMap** is looking for only one type. In addition, there's no way you can "forget" to add the proper code to this system when you add a new type, since there isn't any code you need to add.

Summary

Coming up with a design such as **TrashVisitor.java** that contains a larger amount of code than the earlier designs can seem at first to be counterproductive. It pays to notice what you're trying to accomplish with various designs. Design patterns in general strive to *separate the things that change from the things that stay the same.* The "things that change" can refer to many different kinds of changes. Perhaps the change occurs because the program is placed into a new environment or because something in the current environment changes (this could be: "The user wants to add a new shape to the diagram currently on the screen"). Or, as in this case, the change could be the evolution of the code body. While previous versions of the trash sorting example emphasized the addition of new *types* of **Trash** to the system, **TrashVisitor.java** allows you to easily add new *functionality* without disturbing the **Trash** hierarchy. There's more code in **TrashVisitor.java**, but adding new functionality to **Visitor** is cheap. If this is something that happens a lot, then it's worth the extra effort and code to make it happen more easily.

The discovery of the vector of change is no trivial matter; it's not something that an analyst can usually detect before the program sees its initial design. The necessary information will probably not appear until later phases in the project: sometimes only at the design or implementation phases do you discover a deeper or more subtle need in your system. In the case of adding new types (which was the focus of most of the "recycle" examples) you might realize that you need a particular

inheritance hierarchy only when you are in the maintenance phase and you begin extending the system!

One of the most important things that you'll learn by studying design patterns seems to be an about-face from what has been promoted so far in this book. That is: "OOP is all about polymorphism." This statement can produce the "two-year-old with a hammer" syndrome (everything looks like a nail). Put another way, it's hard enough to "get" polymorphism, and once you do, you try to cast all your designs into that one particular mold.

What design patterns say is that OOP isn't just about polymorphism. It's about "separating the things that change from the things that stay the same." Polymorphism is an especially important way to do this, and it turns out to be helpful if the programming language directly supports polymorphism (so you don't have to wire it in yourself, which would tend to make it prohibitively expensive). But design patterns in general show *other* ways to accomplish the basic goal, and once your eyes have been opened to this you will begin to search for more creative designs.

Since the *Design Patterns* book came out and made such an impact, people have been searching for other patterns. You can expect to see more of these appear as time goes on. Here are some sites recommended by Jim Coplien, of C++ fame (*http://www.bell-labs.com/~cope*), who is one of the main proponents of the patterns movement:

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

Also note there has been a yearly conference on design patterns, called PLOP, that produces a published proceedings, the third of which came out in late 1997 (all published by Addison-Wesley).

Exercises

1.   Add a class **Plastic** to **TrashVisitor.java**.

2.   Add a class **Plastic** to **DynaTrash.java**.

3.   Create a decorator like **VisitableDecorator**, but for the multiple dispatching example, along with an "adapter decorator" class like the one created for **VisitableDecorator**. Build the rest of the example and show that it works.

Projects

A number of more challenging projects for you to solve. [[Some of these may turn into examples in the book, and so at some point might disappear from here]]

Rats & Mazes

First, create a *Blackboard* (cite reference) which is an object on which anyone may record information. This particular blackboard draws a maze, and is used as information comes back about the structure of a maze from the rats that are investigating it.

Now create the maze itself. Like a real maze, this object reveals very little information about itself — given a coordinate, it will tell you whether there are walls or spaces in the four directions immediately surrounding that coordinate, but no more. For starters, read the maze in from a text file but consider hunting on the internet for a maze-generating algorithm. In any event, the result should be an object that, given a maze coordinate, will report walls and spaces around that coordinate. Also, you must be able to ask it for an entry point to the maze.

Finally, create the maze-investigating **Rat** class. Each rat can communicate with both the blackboard to give the current information and the maze to request new information based on the current position of the rat. However, each time a rat reaches a decision point where the maze branches, it creates a new rat to go down each of the branches. Each rat is driven by its own thread. When a rat reaches a dead end, it terminates itself after reporting the results of its final investigation to the blackboard.

The goal is to completely map the maze, but you must also determine whether the end condition will be naturally found or whether the blackboard must be responsible for the decision.

An example implementation by Jeremy Meyer:

```java
//: 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) {
```

```java
    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() {
```

```java
// 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.");
  }
} ///:~
```

The maze initialization file:

```
//:! projects:Amaze.txt
   * **    * * **    *
 ***   * *******  * ****
    ***      ***
 *****  ********** *****
 * * * ** ** * * * ** *
  * * * ** * * * * **
 *   **   *   **
  * **  * ** * **  * **
 *** * *** ***** * *** **
 *   * * **    * *
  * ** * *   * ** * *
///:~
```

Other maze resources

A discussion of algorithms to create mazes as well as Java source code to implement them:

http://www.mazeworks.com/mazegen/mazegen.htm

A discussion of algorithms for collision detection and other individual/group moving behavior for autonomous physical objects:

http://www.red3d.com/cwr/steer/

XML Decorator

Create a pair of decorators for I/O Readers and Writers that encode (for the Writer decorator) and decode (for the reader decorator) XML

A: Tools

Contains tools needed to build the book etc. Some of these may be temporary and disappear when the code base is moved to CVS.

Ant extensions

Ant comes with an extension API so that you can create your own tasks by writing them in Java. You can find full details in the official Ant documentation and in the published books on Ant.

As an alternative, you can simply write a Java program and call it from Ant; this way, you don't have to learn the extension API. For example, to compile the code in this book, we need to verify that the version of Java that the user is running is JDK 1.3 or greater, so we created the following program:

```
//: 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");
  }
} ///:~
```

This simply uses **System.getProperty( )** to discover the Java version, and throws an exception if it isn't at least 1.3. When Ant sees the exception, it will halt. Now you can include the following in any buildfile where you want to check the version number:

```
<java
  taskname="CheckVersion"
  classname="com.bruceeckel.tools.CheckVersion"
  classpath="${basedir}"
  fork="true"
  failonerror="true"
/>
```

If you use this approach to adding tools, you can write them and test them quickly, and if it's justified, you can invest the extra effort and write an Ant extension.

Array utilities

Although useful, the **Arrays** class stops short of being fully functional. For example, it would be nice to be able to easily print the elements of an array without having to code a **for** loop by hand every time. And as you'll see, the **fill( )** method only takes a single value and places it in the array, so if you wanted, for example, to fill an array with randomly generated numbers, **fill( )** is no help.

Thus it makes sense to supplement the **Arrays** class with some additional utilities, which will be placed in the **package com.bruceeckel.util** for convenience. These will print an array of any type and fill an array with values or objects that are created by an object called a *generator* that you can define.

Because code needs to be created for each primitive type as well as **Object**, there's a lot of nearly duplicated code.[14] For example, a "generator" interface is required for each type because the return type of **next( )** must be different in each case:

```
//: 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;
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(); } ///:~
```

Arrays2 contains a variety of **toString( )** methods, overloaded for each
type. These methods allow you to easily print an array. The **toString( )**
code introduces the use of **StringBuffer** instead of **String** objects. This
is a nod to efficiency; when you're assembling a string in a method that
might be called a lot, it's wiser to use the more efficient **StringBuffer**
rather than the more convenient **String** operations. Here, the
StringBuffer is created with an initial value, and **Strings** are appended.
Finally, the **result** is converted to a **String** as the return value:

```
//: 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) {
```

```java
    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)
```

```java
      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();
  }
```

```java
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);
}
```

```java
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();}
  }
} ///:~
```

To fill an array of elements using a generator, the **fill( )** method takes a reference to an appropriate generator **interface**, which has a **next( )** method that will somehow produce an object of the right type (depending on how the interface is implemented). The **fill( )** method simply calls **next( )** until the desired range has been filled. Now you can create any generator by implementing the appropriate **interface** and use your generator with **fill( )**.

Random data generators are useful for testing, so a set of inner classes is created to implement all the primitive generator interfaces, as well as a **String** generator to represent **Object**. You can see that

RandStringGenerator uses **RandCharGenerator** to fill an array of characters, which is then turned into a **String**. The size of the array is determined by the constructor argument.

To generate numbers that aren't too large, **RandIntGenerator** defaults to a modulus of 10,000, but the overloaded constructor allows you to choose a smaller value.

[1] This is a tongue-in-cheek reference to an event in China after the death of Mao-Tze Tung, when four persons including Mao's widow made a power play, and were demonized by the Chinese Communist Party under that name.

[2] From Mark Johnson.

[3] But be warned: the examples are in C++.

[4] A free email publication. See www.BruceEckel.com to subscribe.

[5] From an email from Kevlin Henney.

[6] Shalloway, *Design Patterns Explained*, and Alexandrescu, *Advanced C++ Design (??)*

[7] In the Python language, all functions are already objects and so the *Command* pattern is often redundant.

[8] Page 235.

[9] The original version of this was called *JPython*, but the project changed and the name was changed to emphasize the distinctness of the new version.

[10] Changing the registry setting **python.security.respectJavaAccessibility = true** to **false** makes testing even more powerful because it allows the test script to use *all* methods, even protected and package-private.

[11] No mice were harmed in the creation of this example.

[12] Addison-Wesley, 1999.

[13] This was a solution created by Jaroslav Tulach in a design patterns class that I gave in Prague.

[14] The C++ programmer will note how much the code could be collapsed with the use of default arguments and templates. The Python programmer will note that this entire library would be largely unnecessary in that language.