Thinking in Python

Design Patterns and Problem-Solving Techniques

Bruce Eckel

President, MindView, Inc.

Please note that this document is in its initial form, and much remains to be done.

Contents

Preface 5

Introduction 7

The Y2K syndrome 8
Context and composition 9

A quick course in Python for programmers 11

Python overview 11
Built-in containers 12
Functions 13
Strings 14
Classes 16

The pattern concept 21

What is a pattern? 21
Pattern taxonomy 23
Design Structures 24
Design principles 25
The Singleton 27
Classifying patterns 32
The development challenge33
Exercises 34

2: Unit Testing 34

Write tests first 36
Simple Python testing 37
A very simple framework .. 38
Writing tests 39
White-box & black-box tests42
Running tests 44
Automatically executing tests47
Exercises 47

3: Building application frameworks 47

Template method 48
Exercises 49

4:Fronting for an implementation 49

Proxy 50
State 52
StateMachine 54
Table-Driven State Machine61
The State class 63
Conditions for transition..............63
Transition actions.........................64
The table 64
The basic machine 65
Simple vending machine 65
Testing the machine 70
Tools..................................70
Exercises 70

X: Decorators: dynamic type selection 72

Basic decorator structure ..73
A coffee example................73
Class for each combination73
The decorator approach 76
Compromise 79
Other considerations.........82
Exercises 82

Y: Iterators: decoupling algorithms from containers 83

Type-safe iterators............84

5: Factories: encapsulating object creation 85

Simple Factory method 86
Polymorphic factories 88
Abstract factories.............. 90
Exercises 93

6: Function objects 94

Command: choosing the operation at run-time 94
Strategy: choosing the algorithm at run-time96
Chain of responsibility 97
Exercises 101

7: Changing the interface 101

Adapter 101
Façade..............................103
Exercises 104

8: Table-driven code: configuration flexibility 105

Table-driven code using anonymous inner classes 105

10: Callbacks 107

Observer 107

Observing flowers 109

A visual example of observers 116

Exercises 122

11: Multiple dispatching 122

Visitor, a type of multiple dispatching 127

Exercises 128

12: Pattern refactoring 130

Simulating the trash recycler 130

Improving the design 134

"Make more objects" 135

A pattern for prototyping creation 137

Trash subclasses 142

Parsing **Trash** from an external file 143

Recycling with prototyping 146

Abstracting usage 147

Multiple dispatching 151

Implementing the double dispatch 152

The *Visitor* pattern 158

A Reflective Decorator 161

More coupling? 166

RTTI considered harmful? 166

Summary 169

Exercises 171

13: Projects 171

Rats & Mazes 171

Other maze resources.................. 176

XML Decorator................ 176

Preface

The material in this book began in conjunction with a Java seminar that I have given for several years, a couple of times with Larry O'Brien, then 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. Add Comment

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 occasionally given it as an on-site seminar, but this is expensive and difficult to schedule, because there are two of us. Add Comment

A great deal of appreciation goes to the people who have participated in these seminars over the years, and to Larry and Bill, 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. Add Comment

This book will not stop here, either. Originally, this material was part of a C++ book, then a Java book, then it broke off into its own Java-based book, and finally, after much pondering, I decided that the best way to initially create my design patterns treatise is to write it in Python first (since we know Python makes an ideal prototyping language!) and then translate the relevant parts of the book *back* into the Java version. I've had the experience before of casting an idea in a more powerful language, then translating it back into another language, and I've found that it's much easier to gain insights and keep the idea clear. Add Comment

So *Thinking in Python* is, initially, a translation of *Thinking in Patterns with Java*, 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). Add Comment

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). Add Comment

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 that chapter out of the second edition of *Thinking in Java* because that book was getting too big, and also because I had decided to write *Thinking in Patterns*. That book, still to be finished, has become this one. The ease of expressing these more complex ideas in Python will, I think, finally allow me to get it all out. Add Comment

This is not an introductory book. I am assuming that you have worked your way through at least *Learning Python* (by Mark Lutz & David Ascher; OReilly, 1999) or an equivalent text before coming to this book. Add Comment

In addition, I assume you have more than just a grasp of the syntax of Python. You should have a good understanding of objects and what they're about, including polymorphism. Add Comment

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. Add Comment

---

[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.

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.” Add Comment

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[2]. 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. Add Comment

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. Add Comment

[2] These same people were also convinced that all the computers were going to crash then, too. But since virtually everyone had the experience of their Windows machine crashing all the time without particularly dire results, this didn’t seem to carry the same drama of impending doom.

“Premature optimization is the root of all evil.” is sometimes referred to as “Knuth’s law” (from Donald E. Knuth). Add Comment

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. Add Comment

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 its primary value. Add Comment

A quick course in Python for programmers

This book assumes you're an experienced programmer, and it's best if you have learned Python through another book. For everyone else, this chapter gives a fast introduction to the language. Add Comment

Python overview

This brief introduction is 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). Add Comment

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. Add Comment

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: Add Comment

```
#: c01: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. Add Comment

First notice that the basic syntax of Python is C-ish as you can see in 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 (it won't complain if you use them anyway). Add Comment

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. Add Comment

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. Add Comment

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. Add Comment

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: Add Comment

```
#: c01: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. Add Comment

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

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. Add Comment

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: Add Comment

```
#: c01: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: Add Comment

```
#: c01: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: Add Comment

```
#: c01: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). Add Comment

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: Add Comment

```
#: c01: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. Add Comment

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. Add Comment

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

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: Add Comment

```
#: c01: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 – you can also use regular lists for multiple arguments, but tuples are typical). Add Comment

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

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: Add Comment

```
#: c01: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. Add Comment

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. Add Comment

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: Add Comment

```
Python SimpleClass.py
```

However, if this file is imported as a module into another program, the **__main__** code is not executed. Add Comment

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. Add Comment

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. Add Comment

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. Add Comment

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: Add Comment

```
#: c01: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( )**. Add Comment

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. Add Comment

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. Add Comment

[[Suggest Further Topics for inclusion in the introductory chapter]] Add Comment

The pattern concept

"Design patterns help you learn from others' successes instead of your own failures[3]." Add Comment

Probably the most important step forward in object-oriented design is the "design patterns" movement, chronicled in *Design Patterns (ibid)*[4]. 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. Add Comment

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. Add Comment

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. Add Comment

[3] From Mark Johnson.

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

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). Add Comment

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). Add Comment

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. Add Comment

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. Add Comment

Another pattern that appears in *Design Patterns* is the *iterator*, which has been implicitly available in **for** loops from the beginning of the language, and was introduced as an explicit feature in Python 2.2. An iterator allows you to hide the particular implementation of the container as you're stepping through and selecting the elements one by one. Thus, you can 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 object that can produce an iterator. Add Comment

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: Add Comment

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). Add Comment

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. Add Comment

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

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. Add Comment

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. Add Comment

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

Design Structures

One of the struggles that I've had with design patterns is their classification – I've often found the GoF approach to be too obscure, and not always very helpful. Certainly, the *Creational* patterns are fairly straightforward: how are you going to create your objects? This is a question you normally need to ask, and the name brings you right to that group of patterns. But I find *Structural* and *Behavioral* to be far less useful distinctions. I have not been able to look at a problem and say "clearly, you need a structural pattern here," so that classification doesn't lead me to a solution (I'll readily admit that I may be missing something here). Add Comment

I've labored for awhile with this problem, first noting that the underlying structure of some of the GoF patterns are similar to each other, and trying to develop relationships based on that similarity. While this was an interesting experiment, I don't think it produced much of use in the end because the point is to solve problems, so a helpful approach will look at the problem to solve and try to find relationships between the problem and potential solutions. Add Comment

To that end, I've begun to try to collect basic design structures, and to try to see if there's a way to relate those structures to the various design patterns that appear in well thought-out systems. Currently, I'm just trying to make a list, but eventually I hope to make steps towards connecting these structures with patterns (or I may come up with a different approach altogether – this is still in its formative stages). Add Comment

Here[5] is the present list of candidates, only some of which will make it to the final list. Feel free to suggest others, or possibly relationships with patterns. Add Comment

- **Encapsulation**: self containment and embodying a model of usage
- **Gathering** Add Comment
- **Localization** Add Comment

[5] This list includes suggestions by Kevlin Henney, David Scott, and others.

- **Separation** Add Comment
- **Hiding** Add Comment
- **Guarding** Add Comment
- **Connector** Add Comment
- **Barrier/fence** Add Comment
- **Variation in behavior** Add Comment
- **Notification** Add Comment
- **Transaction** Add Comment
- **Mirror**: "the ability to keep a parallel universe(s) in step with the golden world" Add Comment
- **Shadow** "follows your movement and does something different in a different medium" (May be a variation on Proxy). Add Comment

Design principles

When I put out a call for ideas in my newsletter[6], 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. Add Comment

- **Principle of least astonishment** (don't be astonishing). Add Comment
- **Make common things easy, and rare things possible** Add Comment
- **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

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

problem at hand, the slower the programmer can produce. And this does not appear to be a linear factor, but an exponential one. Add Comment

- **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. Add Comment

- **Subtraction**: a design is finished when you cannot take anything else away[7]. Add Comment

- **Simplicity before generality**[8]. (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. Add Comment

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

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

- **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. Add Comment

[7] This idea is generally attributed to Antoine de St. Exupery from *The Little Prince*: "La perfection est atteinte non quand il ne reste rien à ajouter, mais quand il ne reste rien à enlever," or: "perfection is reached not when there's nothing left to add, but when there's nothing left to remove".

[8] From an email from Kevlin Henney.

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. Add Comment

The Singleton

Possibly the simplest design pattern is the *singleton*, which is a way to provide one and only one object of a particular type. To accomplish this, you must take control of object creation out of the hands of the programmer. One convenient way to do this is to delegate to a single instance of a private nested inner class: Add Comment

```
#: c01:SingletonPattern.py

class OnlyOne:
  class __OnlyOne:
    def __init__(self, arg):
      self.val = arg
    def __str__(self):
      return `self` + self.val
  instance = None
  def __init__(self, arg):
    if not OnlyOne.instance:
      OnlyOne.instance = OnlyOne.__OnlyOne(arg)
    else:
      OnlyOne.instance.val = arg
  def __getattr__(self, name):
    return getattr(self.instance, name)

x = OnlyOne('sausage')
print x
y = OnlyOne('eggs')
print y
z = OnlyOne('spam')
print z
print x
print y
print `x`
print `y`
print `z`
output = '''
```

```
<__main__.__OnlyOne instance at 0076B7AC>sausage
<__main__.__OnlyOne instance at 0076B7AC>eggs
<__main__.__OnlyOne instance at 0076B7AC>spam
<__main__.__OnlyOne instance at 0076B7AC>spam
<__main__.__OnlyOne instance at 0076B7AC>spam
<__main__.OnlyOne instance at 0076C54C>
<__main__.OnlyOne instance at 0076DAAC>
<__main__.OnlyOne instance at 0076AA3C>
'''
#:~
```

Because the inner class is named with a double underscore, it is private so the user cannot directly access it. The inner class contains all the methods that you would normally put in the class if it weren't going to be a singleton, and then it is wrapped in the outer class which controls creation by using its constructor. The first time you create an **OnlyOne**, it initializes **instance**, but after that it just ignores you. Add Comment

Access comes through delegation, using the **__getattr__()** method to redirect calls to the single instance. You can see from the output that even though it appears that multiple objects have been created, the same **__OnlyOne** object is used for both. The instances of **OnlyOne** are distinct but they all proxy to the same **__OnlyOne** object. Add Comment

Note that the above approach doesn't restrict you 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. Add Comment

A variation on this technique uses the class method **__new__** added in Python 2.2:

```
#: c01:NewSingleton.py

class OnlyOne(object):
  class __OnlyOne:
    def __init__(self):
      self.val = None
    def __str__(self):
      return `self` + self.val
  instance = None
  def __new__(cls): # __new__ always a classmethod
    if not OnlyOne.instance:
      OnlyOne.instance = OnlyOne.__OnlyOne()
    return OnlyOne.instance
  def __getattr__(self, name):
    return getattr(self.instance, name)
  def __setattr__(self, name):
    return setattr(self.instance, name)

x = OnlyOne()
x.val = 'sausage'
print x
y = OnlyOne()
y.val = 'eggs'
print y
z = OnlyOne()
z.val = 'spam'
print z
print x
print y
#<hr>
output = '''
<__main__.__OnlyOne instance at 0x00798900>sausage
<__main__.__OnlyOne instance at 0x00798900>eggs
<__main__.__OnlyOne instance at 0x00798900>spam
<__main__.__OnlyOne instance at 0x00798900>spam
<__main__.__OnlyOne instance at 0x00798900>spam
'''
#:~
```

Alex Martelli makes the observation that what we really want with a Singleton is to have a single set of state data for all objects. That is, you could create as many objects as you want and as long as they all refer to the same state information then you achieve the effect of Singleton. He accomplishes this with what he calls the *Borg*[9], which is accomplished by setting all the **__dict__**s to the same static piece of storage: Add Comment

```
#: c01:BorgSingleton.py
# Alex Martelli's 'Borg'

class Borg:
  _shared_state = {}
  def __init__(self):
    self.__dict__ = self._shared_state
```

[9] From the television show *Star Trek: The Next Generation*. The Borg are a hive-mind collective: “we are all one.”

```
class Singleton(Borg):
  def __init__(self, arg):
    Borg.__init__(self)
    self.val = arg
  def __str__(self): return self.val

x = Singleton('sausage')
print x
y = Singleton('eggs')
print y
z = Singleton('spam')
print z
print x
print y
print `x`
print `y`
print `z`
output = '''
sausage
eggs
spam
spam
spam
<__main__.Singleton instance at 0079EF2C>
<__main__.Singleton instance at 0079E10C>
<__main__.Singleton instance at 00798F9C>
'''
#:~
```

This has an identical effect as **SingletonPattern.py** does, but it's more elegant. In the former case, you must wire in *Singleton* behavior to each of your classes, but *Borg* is designed to be easily reused through inheritance. Add Comment

Two other interesting ways to define singleton[10] include wrapping a class and using metaclasses. The first approach could be thought of as a *class decorator* (decorators will be defined later in the book), because it takes the class of interest and adds functionality to it by wrapping it in another class:

```
#: c01:SingletonDecorator.py
```

[10] Suggested by Chih-Chung Chang.

```
class SingletonDecorator:
  def __init__(self,klass):
    self.klass = klass
    self.instance = None
  def __call__(self,*args,**kwds):
    if self.instance == None:
      self.instance = self.klass(*args,**kwds)
    return self.instance

class foo: pass
foo = SingletonDecorator(foo)

x=foo()
y=foo()
z=foo()
x.val = 'sausage'
y.val = 'eggs'
z.val = 'spam'
print x.val
print y.val
print z.val
print x is y is z
#:~
```

[[Description]] Add Comment

The second approach uses metaclasses, a topic I do not yet understand but which looks very interesting and powerful indeed (note that Python 2.2 has improved/simplified the metaclass syntax, and so this example may change):

```
#: c01:SingletonMetaClass.py
class SingletonMetaClass(type):
  def __init__(cls,name,bases,dict):
    super(SingletonMetaClass,cls)\
      .__init__(name,bases,dict)
    original_new = cls.__new__
    def my_new(cls,*args,**kwds):
      if cls.instance == None:
        cls.instance = \
          original_new(cls,*args,**kwds)
      return cls.instance
    cls.instance = None
    cls.__new__ = staticmethod(my_new)

class bar(object):
  __metaclass__ = SingletonMetaClass
  def __init__(self,val):
    self.val = val
  def __str__(self):
    return `self` + self.val

x=bar('sausage')
y=bar('eggs')
z=bar('spam')
print x
print y
print z
print x is y is z
#:~
```

[[Long, detailed, informative description of what metaclasses are and how they work, magically inserted here]] Add Comment

Exercise:

Modify **BorgSingleton.py** so that it uses a class **__new__( )** method. Add Comment

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: Add Comment

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*. Add Comment

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. Add Comment

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. Add Comment

The *Design Patterns* book has a section on each of its 23 patterns along with one or more examples for each, typically in C++ 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 Python.) This book will not repeat all the patterns shown in *Design Patterns* since that book stands on its own and should be studied separately. Instead, this book will give some examples that should provide you with a decent feel for what patterns are about and why they are so important. Add Comment

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, I have found that it's often easier to think about, and especially to learn about, the patterns in terms of these structural principles. This book will attempt to present the patterns based on their structure instead of the categories presented in *Design Patterns*. Add Comment

The development challenge

Issues of development, the UML process, Extreme Programming. Add Comment

Is evaluation valuable? The Capability Immaturity Model:

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

Add Comment

Pair programming research:

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

Add Comment

Exercises

1. **SingletonPattern.py** 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. Add Comment

2. Using **SingletonPattern.py** as a starting point, create a class that manages a fixed number of its own objects. Assume the objects are database connections and you only have a license to use a fixed quantity of these at any one time. Add Comment

2: Unit Testing

This chapter has not had any significant translation yet.

One of the important recent realizations is the dramatic value of unit testing. Add Comment

This is the process of building integrated tests into all the code that you create, and running those tests every time you do a build. It's as if you are extending the compiler, telling it more about what your program is supposed to do. That way, the build process can check for more than just syntax errors, since you teach it how to check for semantic errors as well. Add Comment

C-style programming languages, and C++ in particular, have typically valued performance over programming safety. The reason that developing programs in Java is so much faster than in C++ (roughly twice as fast, by most accounts) is because of Java's safety net: features like better type checking, enforced exceptions and garbage collection. By integrating unit testing into your build process, you are extending this safety net, and the result is that you can develop faster. You can also be bolder in the changes that you make, and more easily refactor your code when you discover design or implementation flaws, and in general produce a better product, faster. Add Comment

Unit testing is not generally considered a design pattern; in fact, it might be considered a “development pattern,” but perhaps there are enough “pattern” phrases in the world already. Its effect on development is so significant that it will be used throughout this book, and thus will be introduced here. Add Comment

My own experience with unit testing began when I realized that every program in a book must be automatically extracted and organized into a source tree, along with appropriate makefiles (or some equivalent technology) so that you could just type **make** to build the whole tree. The effect of this process on the code quality of the book was so immediate and dramatic that it soon became (in my mind) a requisite for any programming book—how can you trust code that you didn’t compile? I also discovered that if I wanted to make sweeping changes, I could do so using search-and-replace throughout the book, and also bashing the code around at will. I knew that if I introduced a flaw, the code extractor and the makefiles would flush it out. Add Comment

As programs became more complex, however, I also found that there was a serious hole in my system. Being able to successfully compile programs is clearly an important first step, and for a published book it seemed a fairly revolutionary one—usually due to the pressures of publishing, it’s quite typical to randomly open a programming book and discover a coding flaw. However, I kept getting messages from readers reporting semantic problems in my code (in *Thinking in Java*). These problems could only be discovered by running the code. Naturally, I understood this and had taken some early faltering steps towards implementing a system that would perform automatic execution tests, but I had succumbed to the pressures of publishing, all the while knowing that there was definitely something wrong with my process and that it would come back to bite me in the form of embarrassing bug reports (in the open source world, embarrassment is one of the prime motivating factors towards increasing the quality of one’s code!). Add Comment

The other problem was that I was lacking a structure for the testing system. Eventually, I started hearing about unit testing and JUnit[11], which

[11] http://www.junit.org

provided a basis for a testing structure. However, even though JUnit is intended to make the creation of test code easy, I wanted to see if I could make it even easier, applying the Extreme Programming principle of "do the simplest thing that could possibly work" as a starting point, and then evolving the system as usage demands (In addition, I wanted to try to reduce the amount of test code, in an attempt to fit more functionality in less code for screen presentations). This chapter is the result. Add Comment

Write tests first

As I mentioned, one of the problems that I encountered—that most people encounter, it turns out—was submitting to the pressures of publishing and as a result letting tests fall by the wayside. This is easy to do if you forge ahead and write your program code because there's a little voice that tells you that, after all, you've got it working now, and wouldn't it be more interesting/useful/expedient to just go on and write that other part (we can always go back and write the tests later). As a result, the tests take on less importance, as they often do in a development project. Add Comment

The answer to this problem, which I first found described in *Extreme Programming Explained*, is to write the tests *before* you write the code. This may seem to artificially force testing to the forefront of the development process, but what it actually does is to give testing enough additional value to make it essential. If you write the tests first, you: Add Comment

1. Describe what the code is supposed to do, not with some external graphical tool but with code that actually lays the specification down in concrete, verifiable terms. Add Comment

2. Provide an example of how the code should be used; again, this is a working, tested example, normally showing all the important method calls, rather than just an academic description of a library. Add Comment

3. Provide a way to verify when the code is finished (when all the tests run correctly). Add Comment

Thus, if you write the tests first then testing becomes a development tool, not just a verification step that can be skipped if you happen to feel comfortable about the code that you just wrote (a comfort, I have found, that is usually wrong). Add Comment

You can find convincing arguments in *Extreme Programming Explained*, as “write tests first” is a fundamental principle of XP. If you aren’t convinced you need to adopt any of the changes suggested by XP, note that according to Software Engineering Institute (SEI) studies, nearly 70% of software organizations are stuck in the first two levels of SEI's scale of sophistication: chaos, and slightly better than chaos. If you change nothing else, add automated testing. Add Comment

Simple Python testing

Sanity check for a quick test of the programs in this book, and to append the output of each program (as a string) to its listing: Add Comment

```
#: SanityCheck.py
import string, glob, os
# Do not include the following in the automatic
# tests:
exclude = ("SanityCheck.py", "BoxObserver.py",)

def visitor(arg, dirname, names):
  dir = os.getcwd()
  os.chdir(dirname)
  try:
    pyprogs = [p for p in glob.glob('*.py')
               if p not in exclude ]
    if not pyprogs: return
    print '[' + os.getcwd() + ']'
    for program in pyprogs:
      print '\t', program
      os.system("python %s > tmp" % program)
      file = open(program).read()
      output = open('tmp').read()
      # Append output if it's not already there:
      if file.find("output = '''") == -1 and \
        len(output) > 0:
        divider = '#' * 50 + '\n'
        file = file.replace('#' + ':~', '#<hr>\n')
        file += "output = '''\n" + \
          open('tmp').read() + "'''\n"
        open(program,'w').write(file)
  finally:
    os.chdir(dir)

if __name__ == "__main__":
  os.path.walk('.', visitor, None)
#:~
```

Just run this from the root directory of the code listings for the book; it will descend into each subdirectory and run the program there. An easy way to check things is to redirect standard output to a file, then if there are any errors they will be the only thing that appears at the console during program execution. Add Comment

A very simple framework

As mentioned, a primary goal of this code is to make the writing of unit testing code very simple, even simpler than with JUnit. As further needs are discovered *during the use* of this system, then that functionality can be added, but to start with the framework will just provide a way to easily create and run tests, and report failure if something breaks (success will produce no results other than normal output that may occur during the running of the test). My intended use of this framework is in makefiles, and **make** aborts if there is a non-zero return value from the execution of a command. The build process will consist of compilation of the programs and execution of unit tests, and if **make** gets all the way through successfully then the system will be validated, otherwise it will abort at the place of failure. The error messages will report the test that failed but not much else, so that you can provide whatever granularity that you need by writing as many tests as you want, each one covering as much or as little as you find necessary. Add Comment

In some sense, this framework provides an alternative place for all those "print" statements I've written and later erased over the years. Add Comment

To create a set of tests, you start by making a **static** inner class inside the class you wish to test (your test code may also test other classes; it's up to you). This test code is distinguished by inheriting from **UnitTest**: Add Comment

```
#  test:UnitTest.py
# The basic unit testing class

class UnitTest:
  static String testID
  static List errors = ArrayList()
  # Override cleanup() if test object
  # creation allocates non-memory
  # resources that must be cleaned up:
  def cleanup(self):
  # Verify the truth of a condition:
  protected final void affirm(boolean condition){
    if(!condition)
      errors.add("failed: " + testID)

# :~
```

The only testing method [[So far]] is **affirm( )**[12], which is **protected** so that it can be used from the inheriting class. All this method does is verify that something is **true**. If not, it adds an error to the list, reporting that the current test (established by the **static testID**, which is set by the test-running program that you shall see shortly) has failed. Although this is not a lot of information—you might also wish to have the line number, which could be extracted from an exception—it may be enough for most situations. Add Comment

Unlike JUnit (which uses **setUp( )** and **tearDown( )** methods), test objects will be built using ordinary Python construction. You define the test objects by creating them as ordinary class members of the test class, and a new test class object will be created for each test method (thus preventing any problems that might occur from side effects between tests). Occasionally, the creation of a test object will allocate non-memory resources, in which case you must override **cleanup( )** to release those resources. Add Comment

Writing tests

Writing tests becomes very simple. Here's an example that creates the necessary **static** inner class and performs trivial tests: Add Comment

```
#  c02:TestDemo.py
# Creating a test
```

[12] I had originally called this **assert()**, but that word became reserved in JDK 1.4 when assertions were added to the language.

```
class TestDemo:
  private static int objCounter = 0
  private int id = ++objCounter
  public TestDemo(String s):
    print (s + ": count = " + id)

  def close(self):
    print ("Cleaning up: " + id)

  def someCondition(self): return 1
  public static class Test(UnitTest):
    TestDemo test1 = TestDemo("test1")
    TestDemo test2 = TestDemo("test2")
    def cleanup(self):
      test2.close()
      test1.close()

    def testA(self):
      print “TestDemo.testA“
      affirm(test1.someCondition())

    def testB(self):
      print “TestDemo.testB“
      affirm(test2.someCondition())
      affirm(TestDemo.objCounter != 0)

    # Causes the build to halt:
    #! public void test3(): affirm(0)

# :~
```
The **test3( )** method is commented out because, as you'll see, it causes the automatic build of this book's source-code tree to stop. Add Comment

You can name your inner class anything you'd like; the only important factor is that it **extends UnitTest**. You can also include any necessary support code in other methods. Only **public** methods that take no arguments and return **void** will be treated as tests (the names of these methods are also not constrained). Add Comment

The above test class creates two instances of **TestDemo**. The **TestDemo** constructor prints something, so that we can see it being called. You could also define a default constructor (the only kind that is used by the test framework), although none is necessary here. The **TestDemo** class has a

close( ) method which suggests it is used as part of object cleanup, so this is called in the overridden **cleanup( )** method in **Test**. Add Comment

The testing methods use the **affirm( )** method to validate expressions, and if there is a failure the information is stored and printed after all the tests are run. Of course, the **affirm( )** arguments are usually more complicated than this; you'll see more examples throughout the rest of this book. Add Comment

Notice that in **testB( )**, the **private** field **objCounter** is accessible to the testing code—this is because **Test** has the permissions of an inner class. Add Comment

You can see that writing test code requires very little extra effort, and no knowledge other than that used for writing ordinary classes. Add Comment

To run the tests, you use **RunUnitTests.py** (which will be introduced shortly). The command for the above code looks like this: Add Comment

java com.bruceeckel.test.RunUnitTests TestDemo

It produces the following output:

```
test1: count = 1
test2: count = 2
TestDemo.testA
Cleaning up: 2
Cleaning up: 1
test1: count = 3
test2: count = 4
TestDemo.testB
Cleaning up: 4
Cleaning up: 3
```

All the output is noise as far as the success or failure of the unit testing is concerned. Only if one or more of the unit tests fail does the program returns a non-zero value to terminate the **make** process after the error messages are produced. Thus, you can choose to produce output or not, as it suits your needs, and the test class becomes a good place to put any printing code you might need—if you do this, you tend to keep such code around rather than putting it in and stripping it out as is typically done with tracing code. Add Comment

If you need to add a test to a class derived from one that already has a test class, it's no problem, as you can see here:

```
#  c02:TestDemo2.py
# Inheriting from a class that
# already has a test is no problem.

class TestDemo2(TestDemo):
  public TestDemo2(String s): .__init__(s)
  # You can even use the same name
  # as the test class in the base class:
  public static class Test(UnitTest):
    def testA(self):
      print "TestDemo2.testA"
      affirm(1 + 1 == 2)

    def testB(self):
      print "TestDemo2.testB"
      affirm(2 * 2 == 4)

# :~
```

Even the name of the inner class can be the same. In the above code, all the assertions are always true so the tests will never fail. Add Comment

White-box & black-box tests

The unit test examples so far are what are traditionally called *white-box tests*. This means that the test code has complete access to the internals of the class that's being tested (so it might be more appropriately called "transparent box" testing). White-box testing happens automatically when you make the unit test class as an inner class of the class being tested, since inner classes automatically have access to all their outer class elements, even those that are **private**. Add Comment

A possibly more common form of testing is *black-box testing*, which refers to treating the class under test as an impenetrable box. You can't see the internals; you can only access the **public** portions of the class. Thus, black-box testing corresponds more closely to functional testing, to verify the methods that the client programmer is going to use. In addition, black-box testing provides a minimal instruction sheet to the client programmer – in the absence of all other documentation, the black-box tests at least demonstrate how to make basic calls to the **public** class methods. Add Comment

To perform black-box tests using the unit-testing framework presented in this book, all you need to do is create your test class as a global class instead of an inner class. All the other rules are the same (for example, the unit test class must be **public**, and derived from **UnitTest**). Add Comment

There's one other caveat, which will also provide a little review of Java packages. If you want to be completely rigorous, you must put your black-box test class in a separate directory than the class it tests, otherwise it will have package access to the elements of the class being tested. That is, you'll be able to access **protected** and **friendly** elements of the class being tested. Here's an example: Add Comment

```
#  c02:Testable.py

class Testable:
  private void f1():
  def f2(self): # "Friendly": package access
  def f3(self): # Also package access
  def f4(self):
# :~
```

Normally, the only method that should be directly accessible to the client programmer is **f4( )**. However, if you put your black-box test in the same directory, it automatically becomes part of the same package (in this case, the default package since none is specified) and then has inappropriate access: Add Comment

```
#  c02:TooMuchAccess.py

class TooMuchAccess(UnitTest):
  Testable tst = Testable()
  def test1(self):
    tst.f2() # Oops!
    tst.f3() # Oops!
    tst.f4() # OK

# :~
```

You can solve the problem by moving **TooMuchAccess.py** into its own subdirectory, thereby putting it in its own default package (thus a different package from **Testable.py**). Of course, when you do this, then **Testable** must be in its own package, so that it can be imported (note that it is also possible to import a "package-less" class by giving the class name in the **import** statement and ensuring that the class is in your CLASSPATH): Add Comment

```
#  c02:testable:Testable.py
package c02.testable

class Testable:
  private void f1():
  def f2(self): # "Friendly": package access
  def f3(self): # Also package access
  def f4(self):
# :~
```

Here's the black-box test in its own package, showing how only public methods may be called: Add Comment

```
#  c02:test:BlackBoxTest.py

class BlackBoxTest(UnitTest):
  Testable tst = Testable()
  def test1(self):
    #! tst.f2() # Nope!
    #! tst.f3() # Nope!
    tst.f4() # Only public methods available

# :~
```

Note that the above program is indeed very similar to the one that the client programmer would write to use your class, including the imports and available methods. So it does make a good programming example. Of course, it's easier from a coding standpoint to just make an inner class, and unless you're ardent about the need for specific black-box testing you may just want to go ahead and use the inner classes (with the knowledge that if you need to you can later extract the inner classes into separate black-box test classes, without too much effort). Add Comment

Running tests

The program that runs the tests makes significant use of reflection so that writing the tests can be simple for the client programmer. Add Comment

```
# test:RunUnitTests.py
# Discovering the unit test
# class and running each test.

class RunUnitTests:
  public static void
  require(boolean requirement, String errmsg):
```

```
  if(!requirement):
    System.err.println(errmsg)
    System.exit(1)

def main(self, String[] args):
  require(args.length == 1,
    "Usage: RunUnitTests qualified-class")
  try:
    Class c = Class.forName(args[0])
    # Only finds the inner classes
    # declared in the current class:
    Class[] classes = c.getDeclaredClasses()
    Class ut = null
    for(int j = 0 j < classes.length j++):
      # Skip inner classes that are
      # not derived from UnitTest:
      if(!UnitTest.class.
          isAssignableFrom(classes[j]))
            continue
      ut = classes[j]
      break # Finds the first test class only

    # If it found an inner class,
    # that class must be static:
    if(ut != null)
      require(
        Modifier.isStatic(ut.getModifiers()),
        "inner UnitTest class must be static")
    # If it couldn't find the inner class,
    # maybe it's a regular class (for black-
    # box testing:
    if(ut == null)
      if(UnitTest.class.isAssignableFrom(c))
        ut = c
    require(ut != null,
      "No UnitTest class found")
    require(
      Modifier.isPublic(ut.getModifiers()),
      "UnitTest class must be public")
    Method[] methods = ut.getDeclaredMethods()
    for(int k = 0 k < methods.length k++):
      Method m = methods[k]
      # Ignore overridden UnitTest methods:
      if(m.getName().equals("cleanup"))
```

```
          continue
        # Only public methods with no
        # arguments and void return
        # types will be used as test code:
        if(m.getParameterTypes().length == 0 &&
           m.getReturnType() == void.class &&
           Modifier.isPublic(m.getModifiers())):
             # The name of the test is
             # used in error messages:
             UnitTest.testID = m.getName()
             # A instance of the
             # test object is created and
             # cleaned up for each test:
             Object test = ut.newInstance()
             m.invoke(test, Object[0])
             ((UnitTest)test).cleanup()


     catch(Exception e):
      e.printStackTrace(System.err)
      # Any exception will return a nonzero
      # value to the console, so that
      # 'make' will abort:
      System.err.println("Aborting make")
      System.exit(1)

    # After all tests in this class are run,
    # display any results. If there were errors,
    # abort 'make' by returning a nonzero value.
    if(UnitTest.errors.size() != 0):
      Iterator it = UnitTest.errors.iterator()
      while(it.hasNext())
        System.err.println(it.next())
      System.exit(1)

# :~
```

Automatically executing tests

Exercises

1. Install this book's source code tree and ensure that you have a **make** utility installed on your system (Gnu **make** is freely available on the internet at various locations). In **TestDemo.py**, un-comment **test3( )**, then type **make** and observe the results. Add Comment

2. Modify TestDemo.java by adding a new test that throws an exception. Type **make** and observe the results. Add Comment

3. Modify your solutions to the exercises in Chapter 1 by adding unit tests. Write makefiles that incorporate the unit tests. Add Comment

3: Building application frameworks

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). Add Comment

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. Add Comment

Template method

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). Add Comment

```
#: c03:TemplateMethod.py
# Simple demonstration of Template Method.

class ApplicationFramework:
  def __init__(self):
    self.__templateMethod()
  def __templateMethod(self):
    for i in range(5):
      self.customize1()
      self.customize2()

# Create a "application":
class MyApp(ApplicationFramework):
  def customize1(self):
    print "Nudge, nudge, wink, wink! ",
  def customize2(self):
    print "Say no more, Say no more!"

MyApp()
#:~
```

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. Add Comment

We'll see *Template Method* numerous other times throughout the book. Add Comment

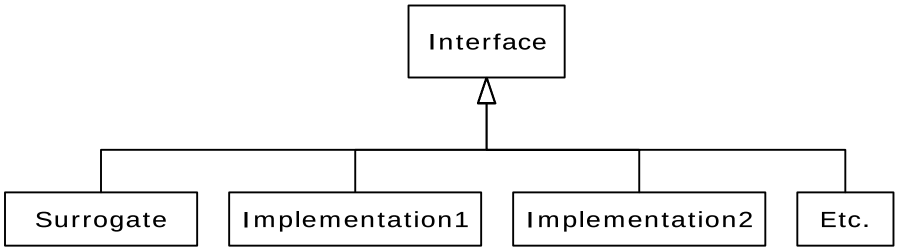
Exercises

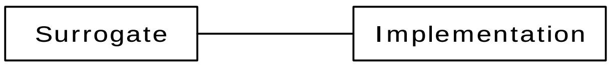
1. Create a framework that takes a list of file names on the command line. It opens each file except the last for reading, and the last for writing. The framework will process each input file using an undetermined policy and write the output to the last file. Inherit to customize this framework to create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

4:Fronting for an implementation

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. Add Comment

The basic idea is simple: from a base class, the surrogate is derived along with the class or classes that provide the actual implementation: Add Comment





When a surrogate object is created, it is given an implementation to which to send all of the method calls. Add Comment

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. Add Comment

Proxy

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

```
#: c04:ProxyDemo.py
# Simple demonstration of the Proxy pattern.

class Implementation:
  def f(self):
    print "Implementation.f()"
  def g(self):
    print "Implementation.g()"
  def h(self):
    print "Implementation.h()"

class Proxy:
  def __init__(self):
    self.__implementation = Implementation()
  # Pass method calls to the implementation:
```

```
  def f(self): self.__implementation.f()
  def g(self): self.__implementation.g()
  def h(self): self.__implementation.h()

p = Proxy()
p.f(); p.g(); p.h()
#:~
```

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. Add Comment

Of course, in Python we have a delegation mechanism built in, so it makes the **Proxy** even simpler to implement: Add Comment

```
#: c04:ProxyDemo2.py
# Simple demonstration of the Proxy pattern.

class Implementation2:
  def f(self):
    print "Implementation.f()"
  def g(self):
    print "Implementation.g()"
  def h(self):
    print "Implementation.h()"

class Proxy2:
  def __init__(self):
    self.__implementation = Implementation2()
  def __getattr__(self, name):
    return getattr(self.__implementation, name)

p = Proxy2()
p.f(); p.g(); p.h();
#:~
```

The beauty of using **__getattr__( )** is that **Proxy2** is completely generic, and not tied to any particular implementation (in Java, a rather complicated “dynamic proxy” has been invented to accomplish this same thing). Add Comment

State

The *State* pattern adds more implementations to *Proxy*, along with a way to switch from one implementation to another during the lifetime of the surrogate: Add Comment

```
#: c04:StateDemo.py
# Simple demonstration of the State pattern.

class State_d:
  def __init__(self, imp):
    self.__implementation = imp
  def changeImp(self, newImp):
    self.__implementation = newImp
  # Delegate calls to the implementation:
  def __getattr__(self, name):
    return getattr(self.__implementation, name)

class Implementation1:
  def f(self):
    print "Fiddle de dum, Fiddle de dee,"
  def g(self):
    print "Eric the half a bee."
  def h(self):
    print "Ho ho ho, tee hee hee,"

class Implementation2:
  def f(self):
    print "We're Knights of the Round Table."
  def g(self):
    print "We dance whene'er we're able."
  def h(self):
    print "We do routines and chorus scenes"

def run(b):
  b.f()
  b.g()
  b.h()
  b.g()

b = State_d(Implementation1())
run(b)
b.changeImp(Implementation2())
```

```
run(b)
#:~
```

You can see that the first implementation is used for a bit, then the second implementation is swapped in and that is used. Add Comment

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

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. Add Comment

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

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

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. Add Comment

You could look at a Python 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). Add Comment

[[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.]]] Add Comment

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. Add Comment

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. Add Comment

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. Add Comment

```
#: c04:statemachine:State.py
# A State has an operation, and can be moved
# into the next State given an Input:

class State:
  def run(self):
    assert 1, "run not implemented"
  def next(self, input):
    assert 1, "next not implemented"
#:~
```

This class is clearly unnecessary, but it allows us to say that something is a **State** object in code, and provide a slightly different error message when all the methods are not implemented. We could have gotten basically the same effect by saying: Add Comment

```
class State: pass
```

because we would still get exceptions if **run( )** or **next( )** were called for a derived type, and they hadn’t been implemented. Add Comment

The **StateMachine** keeps track of the current state, which is initialized by the constructor. The **runAll( )** method takes a list of **Input** objects. 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. Add Comment

```
#: c04:statemachine:StateMachine.py
# Takes a list of Inputs to move from State to
# State using a template method.

class StateMachine:
  def __init__(self, initialState):
    self.currentState = initialState
    self.currentState.run()
  # Template method:
  def runAll(self, inputs):
    for i in inputs:
      print i
      self.currentState = self.currentState.next(i)
      self.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. Add Comment

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[13]. 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: Add Comment

```
#: c04:mouse:MouseAction.py

class MouseAction:
  def __init__(self, action):
    self.action = action
  def __str__(self): return self.action
```

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

```
  def __cmp__(self, other):
    return cmp(self.action, other.action)
  # Necessary when __cmp__ or __eq__ is defined
  # in order to make this class usable as a
  # dictionary key:
  def __hash__(self):
    return hash(self.action)

# Static fields; an enumeration of instances:
MouseAction.appears = MouseAction("mouse appears")
MouseAction.runsAway = MouseAction("mouse runs away")
MouseAction.enters = MouseAction("mouse enters trap")
MouseAction.escapes = MouseAction("mouse escapes")
MouseAction.trapped = MouseAction("mouse trapped")
MouseAction.removed = MouseAction("mouse removed")
#:~
```

You'll note that **__cmp__( )** has been overidden to implement a comparison between **action** values. Also, each possible move by a mouse is enumerated as a **MouseAction** object, all of which are static fields in **MouseAction**. Add Comment

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

```
#:! c04: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
#:~
```

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

```
#: c04:mousetrap1:MouseTrapTest.py
# State Machine pattern using 'if' statements
# to determine the next state.
import string, sys
sys.path += ['../statemachine', '../mouse']
from State import State
from StateMachine import StateMachine
from MouseAction import MouseAction
# A different subclass for each state:

class Waiting(State):
  def run(self):
    print "Waiting: Broadcasting cheese smell"

  def next(self, input):
    if input == MouseAction.appears:
      return MouseTrap.luring
    return MouseTrap.waiting

class Luring(State):
  def run(self):
    print "Luring: Presenting Cheese, door open"

  def next(self, input):
    if input == MouseAction.runsAway:
      return MouseTrap.waiting
    if input == MouseAction.enters:
      return MouseTrap.trapping
    return MouseTrap.luring

class Trapping(State):
  def run(self):
    print "Trapping: Closing door"

  def next(self, input):
    if input == MouseAction.escapes:
      return MouseTrap.waiting
    if input == MouseAction.trapped:
      return MouseTrap.holding
    return MouseTrap.trapping

class Holding(State):
  def run(self):
    print "Holding: Mouse caught"
  def next(self, input):
    if input == MouseAction.removed:
      return MouseTrap.waiting
    return MouseTrap.holding

class MouseTrap(StateMachine):
  def __init__(self):
    # Initial state
    StateMachine.__init__(self, MouseTrap.waiting)

# Static variable initialization:
MouseTrap.waiting = Waiting()
MouseTrap.luring = Luring()
MouseTrap.trapping = Trapping()
MouseTrap.holding = Holding()

moves = map(string.strip,
  open("../mouse/MouseMoves.txt").readlines())
MouseTrap().runAll(map(MouseAction, moves))
#:~
```

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**. Add Comment

While the use of **if** 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. Add Comment

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. Add Comment

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**): Add Comment

```
#: c04:mousetrap2:MouseTrap2Test.py
# A better mousetrap using tables
import string, sys
sys.path += ['../statemachine', '../mouse']
from State import State
from StateMachine import StateMachine
from MouseAction import MouseAction

class StateT(State):
  def __init__(self):
    self.transitions = None
  def next(self, input):
    if self.transitions.has_key(input):
      return self.transitions[input]
    else:
      raise "Input not supported for current state"

class Waiting(StateT):
  def run(self):
    print "Waiting: Broadcasting cheese smell"
  def next(self, input):
    # Lazy initialization:
    if not self.transitions:
      self.transitions = {
        MouseAction.appears : MouseTrap.luring
      }
    return StateT.next(self, input)

class Luring(StateT):
  def run(self):
    print "Luring: Presenting Cheese, door open"
  def next(self, input):
    # Lazy initialization:
    if not self.transitions:
      self.transitions = {
        MouseAction.enters : MouseTrap.trapping,
        MouseAction.runsAway : MouseTrap.waiting
      }
    return StateT.next(self, input)

class Trapping(StateT):
  def run(self):
    print "Trapping: Closing door"
  def next(self, input):
    # Lazy initialization:
    if not self.transitions:
      self.transitions = {
        MouseAction.escapes : MouseTrap.waiting,
        MouseAction.trapped : MouseTrap.holding
      }
    return StateT.next(self, input)

class Holding(StateT):
  def run(self):
    print "Holding: Mouse caught"
  def next(self, input):
    # Lazy initialization:
    if not self.transitions:
      self.transitions = {
        MouseAction.removed : MouseTrap.waiting
      }
    return StateT.next(self, input)

class MouseTrap(StateMachine):
  def __init__(self):
    # Initial state
    StateMachine.__init__(self, MouseTrap.waiting)

# Static variable initialization:
MouseTrap.waiting = Waiting()
MouseTrap.luring = Luring()
MouseTrap.trapping = Trapping()
MouseTrap.holding = Holding()

moves = map(string.strip,
  open("../mouse/MouseMoves.txt").readlines())
mouseMoves = map(MouseAction, moves)
MouseTrap().runAll(mouseMoves)
#:~
```

The rest of the code is identical – the difference is in the **next( )** methods and the **StateT** class. Add Comment

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. Add Comment

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. Add Comment

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. Add Comment

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: Add Comment

(Simple State Machine Diagram) Add Comment

Goals:

- Direct translation of state diagram Add Comment
- Vector of change: the state diagram representation Add Comment
- Reasonable implementation Add Comment
- No excess of states (you could represent every single change with a new state) Add Comment
- Simplicity and flexibility Add Comment

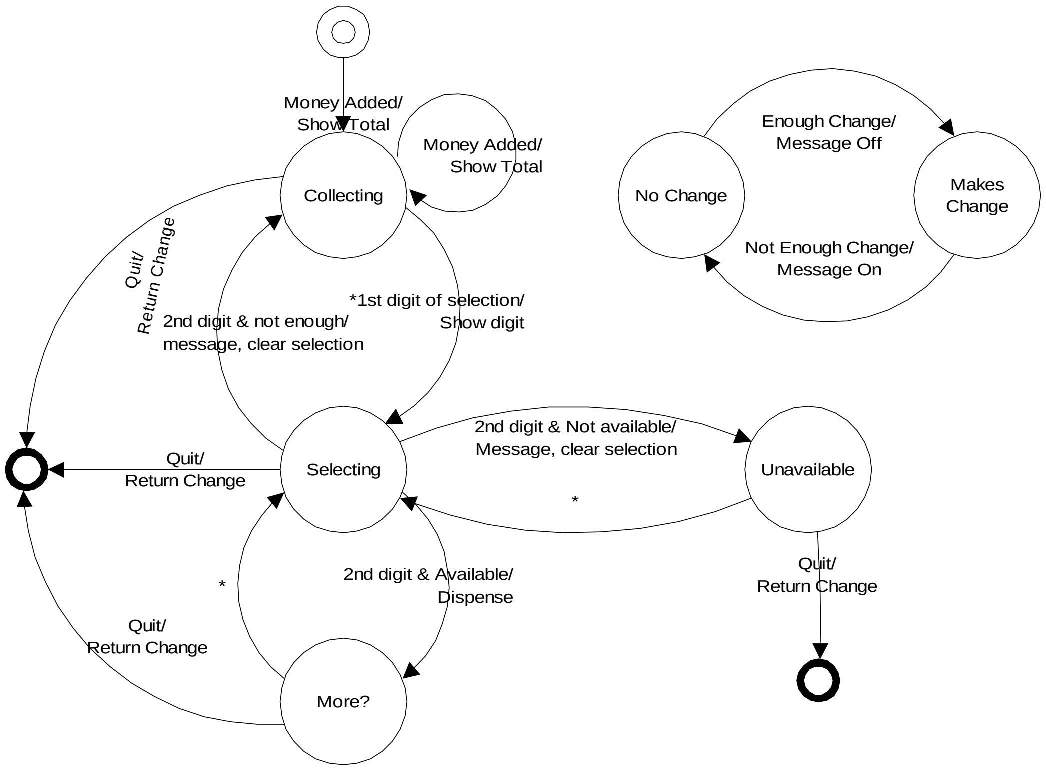
Observations:

- States are trivial – no information or functions/data, just an identity Add Comment
- Not like the State pattern! Add Comment
- The machine governs the move from state to state Add Comment
- Similar to flyweight Add Comment

- Each state may move to many others Add Comment
- Condition & action functions must also be external to states Add Comment
- Centralize description in a single table containing all variations, for ease of configuration Add Comment

Example: Add Comment

- State Machine & Table-Driven Code Add Comment
- Implements a vending machine Add Comment
- Uses several other patterns Add Comment
- Separates common state-machine code from specific application (like template method) Add Comment
- Each input causes a seek for appropriate solution (like chain of responsibility) Add Comment
- Tests and transitions are encapsulated in function objects (objects that hold functions) Add Comment
- Java constraint: methods are not first-class objects Add Comment



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: Add Comment

```
# c04:statemachine2:State.py

class State:
  def __init__(self, name): self.name = name
  def __str__(self): return self.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: Add Comment

```
# c04:statemachine2:Input.py
# Inputs to a state machine

class Input: pass
# :~
```

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

```
# c04:statemachine2:Condition.py
# Condition function object for state machine

class Condition:
  boolean condition(input) :
    assert 1, "condition() not implemented"
# :~
```

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): Add Comment

```
# c04:statemachine2:Transition.py
# Transition function object for state machine

class Transition:
  def transition(self, input):
    assert 1, "transition() not implemented"
# :~
```

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: Add Comment

```
{(CurrentState, InputA) : (ConditionA, TransitionA, NextA),
 (CurrentState, InputB) : (ConditionB, TransitionB, NextB),
 (CurrentState, InputC) : (ConditionC, TransitionC, NextC),
 ...
}
```

The basic machine

```
# c04:statemachine2:StateMachine.py
# A table-driven state machine

class StateMachine:
  def __init__(self, initialState, tranTable):
    self.state = initialState
    self.transitionTable = tranTable

  def nextState(self, 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 RuntimeException(
      "Input not supported for current state")

# :~
```

Simple vending machine

```
# c04:vendingmachine:VendingMachine.py
# Demonstrates use of StateMachine.py
import sys
sys.path += ['../statemachine2']
import StateMachine

class State:
  def __init__(self, name): self.name = name
  def __str__(self): return self.name

State.quiescent = State("Quiesecent")
State.collecting = State("Collecting")
State.selecting = State("Selecting")
State.unavailable = State("Unavailable")
State.wantMore = State("Want More?")
State.noChange = State("Use Exact Change Only")
State.makesChange = State("Machine makes change")

class HasChange:
  def __init__(self, name): self.name = name
  def __str__(self): return self.name

HasChange.yes = HasChange("Has change")
HasChange.no = HasChange("Cannot make change")

class ChangeAvailable(StateMachine):
  def __init__(self):
    StateMachine.__init__(State.makesChange, {
      # Current state, input
      (State.makesChange, HasChange.no) :
        # test, transition, next state:
        (null, null, State.noChange),
      (State.noChange, HasChange.yes) :
        (null, null, State.noChange)
    })

class Money:
  def __init__(self, name, value):
    self.name = name
    self.value = value
  def __str__(self): return self.name
  def getValue(self): return self.value

Money.quarter = Money("Quarter", 25)
Money.dollar = Money("Dollar", 100)

class Quit:
  def __str__(self): return "Quit"

Quit.quit = Quit()
```

```
class Digit:
  def __init__(self, name, value):
    self.name = name
    self.value = value
  def __str__(self): return self.name
  def getValue(self): return self.value

class FirstDigit(Digit): pass
FirstDigit.A = FirstDigit("A", 0)
FirstDigit.B = FirstDigit("B", 1)
FirstDigit.C = FirstDigit("C", 2)
FirstDigit.D = FirstDigit("D", 3)

class SecondDigit(Digit): pass
SecondDigit.one = SecondDigit("one", 0)
SecondDigit.two = SecondDigit("two", 1)
SecondDigit.three = SecondDigit("three", 2)
SecondDigit.four = SecondDigit("four", 3)

class ItemSlot:
  id = 0
  def __init__(self, price, quantity):
    self.price = price
    self.quantity = quantity
  def __str__(self): return `ItemSlot.id`
  def getPrice(self): return self.price
  def getQuantity(self): return self.quantity
  def decrQuantity(self): self.quantity -= 1

class VendingMachine(StateMachine):
  changeAvailable = ChangeAvailable()
  amount = 0
  FirstDigit first = null
  ItemSlot[][] items = ItemSlot[4][4]

  # Conditions:
  def notEnough(self, input):
    i1 = first.getValue()
    i2 = input.getValue()
    return items[i1][i2].getPrice() > amount

  def itemAvailable(self, input):
    i1 = first.getValue()
    i2 = input.getValue()
    return items[i1][i2].getQuantity() > 0

  def itemNotAvailable(self, input):
    return !itemAvailable.condition(input)
    #i1 = first.getValue()
    #i2 = input.getValue()
    #return items[i1][i2].getQuantity() == 0

  # Transitions:
  def clearSelection(self, input):
    i1 = first.getValue()
    i2 = input.getValue()
    ItemSlot is = items[i1][i2]
    print (
      "Clearing selection: item " + is +
      " costs " + is.getPrice() +
      " and has quantity " + is.getQuantity())
    first = null

  def dispense(self, input):
    i1 = first.getValue()
    i2 = input.getValue()
    ItemSlot is = items[i1][i2]
    print ("Dispensing item " +
      is + " costs " + is.getPrice() +
      " and has quantity " + is.getQuantity())
    items[i1][i2].decrQuantity()
    print ("Quantity " +
      is.getQuantity())
    amount -= is.getPrice()
    print("Amount remaining " +
      amount)

  def showTotal(self, input):
    amount += ((Money)input).getValue()
    print "Total amount = " + amount

  def returnChange(self, input):
    print "Returning " + amount
    amount = 0

  def showDigit(self, input):
    first = (FirstDigit)input
    print "First Digit= "+ first
```

```
def __init__(self):
  StateMachine.__init__(self, State.quiescent)
  for(int i = 0 i < items.length i++)
    for(int j = 0 j < items[i].length j++)
      items[i][j] = ItemSlot((j+1)*25, 5)
  items[3][0] = ItemSlot(25, 0)
  buildTable(Object[][][]{
   ::State.quiescent, # Current state
      # Input, test, transition, next state:
     :Money.class, null,
       showTotal, State.collecting,
   ::State.collecting, # Current state
      # Input, test, transition, next state:
     :Quit.quit, null,
       returnChange, State.quiescent,
     :Money.class, null,
       showTotal, State.collecting,
     :FirstDigit.class, null,
       showDigit, State.selecting,
   ::State.selecting, # Current state
      # Input, test, transition, next state:
     :Quit.quit, null,
       returnChange, State.quiescent,
     :SecondDigit.class, notEnough,
       clearSelection, State.collecting,
     :SecondDigit.class, itemNotAvailable,
       clearSelection, State.unavailable,
     :SecondDigit.class, itemAvailable,
       dispense, State.wantMore,
   ::State.unavailable, # Current state
      # Input, test, transition, next state:
     :Quit.quit, null,
       returnChange, State.quiescent,
     :FirstDigit.class, null,
       showDigit, State.selecting,
   ::State.wantMore, # Current state
      # Input, test, transition, next state:
     :Quit.quit, null,
       returnChange, State.quiescent,
     :FirstDigit.class, null,
       showDigit, State.selecting,
  )
```

```
# :~
```

Testing the machine

```
# c04:vendingmachine:VendingMachineTest.py
# Demonstrates use of StateMachine.py

vm = VendingMachine()
for input in [
    Money.quarter,
    Money.quarter,
    Money.dollar,
    FirstDigit.A,
    SecondDigit.two,
    FirstDigit.A,
    SecondDigit.two,
    FirstDigit.C,
    SecondDigit.three,
    FirstDigit.D,
    SecondDigit.one,
    Quit.quit]:
  vm.nextState(input)

# :~
```

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. Add Comment

Exercises

1. Create an example of the “virtual proxy.” Add Comment
2. Create an example of the “Smart reference” proxy where you keep count of the number of method calls to a particular object. Add Comment

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. Add Comment

4. Using the *State*, 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**. Add Comment

5. Create a simple copy-on write implementation. Add Comment

6. Apply **TransitionTable.py** to the "Washer" problem. Add Comment

7. 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. Add Comment

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

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

10. Create an elevator state machine system using StateMachine.java Add Comment

11. Create a heating/air-conditioning system using StateMachine.java Add Comment

12. 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**. Add Comment

X: Decorators: dynamic type selection

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

Used when subclassing creates too many (& inflexible) classes Add Comment

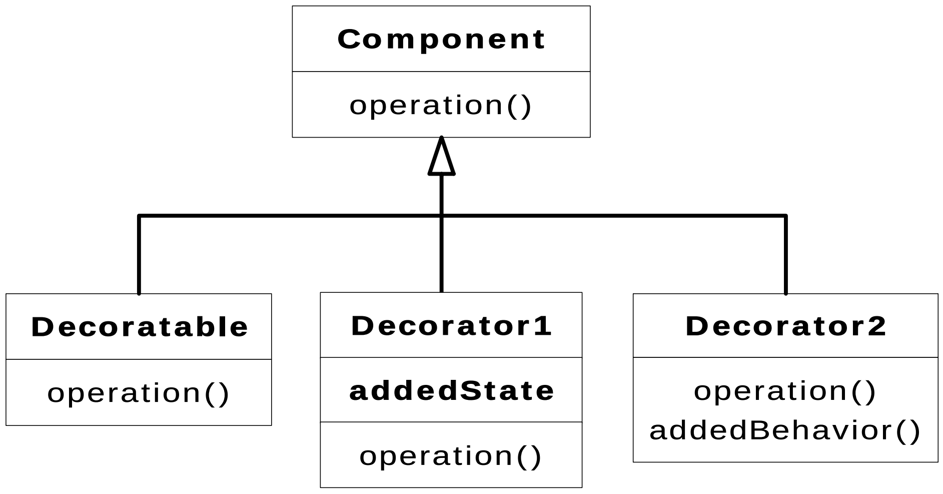
All decorators that wrap around the original object must have the same basic interface Add Comment

Dynamic proxy/surrogate? Add Comment

This accounts for the odd inheritance structure Add Comment

Tradeoff: coding is more complicated when using decorators Add Comment

Basic decorator structure



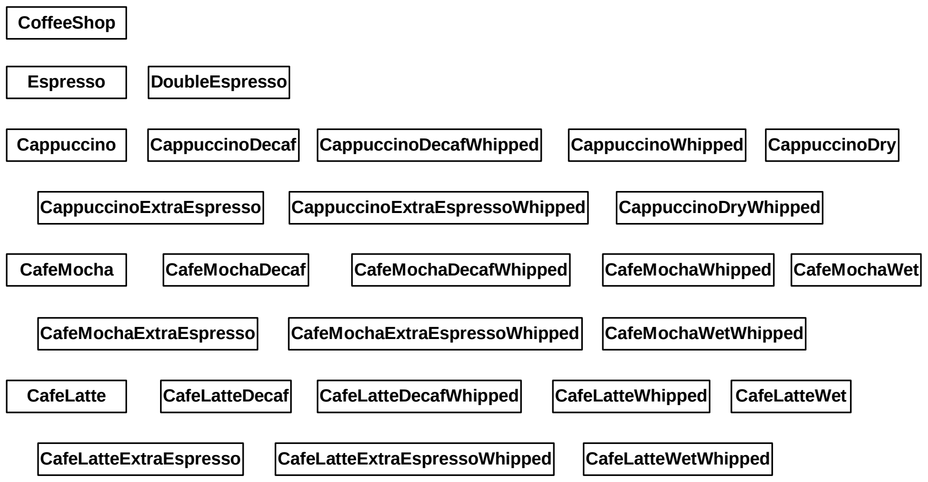
A coffee example

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. Add Comment

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"). Add Comment

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: Add Comment



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

```
class Cappuccino:
  def __init__(self):
    self.cost = 1
    self.description = "Cappucino"
  def getCost(self):
    return self.cost
  def getDescription(self):
    return self.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: Add Comment

```
#: cX:decorator:nodecorators:CoffeeShop.py
# Coffee example with no decorators

class Espresso: pass
class DoubleEspresso: pass
class EspressoConPanna: pass

class Cappuccino:
  def __init__(self):
    self.cost = 1
    self.description = "Cappucino"
  def getCost(self):
    return self.cost
  def getDescription(self):
    return self.description

class CappuccinoDecaf: pass
class CappuccinoDecafWhipped: pass
class CappuccinoDry: pass
class CappuccinoDryWhipped: pass
class CappuccinoExtraEspresso: pass
class CappuccinoExtraEspressoWhipped: pass
class CappuccinoWhipped: pass

class CafeMocha: pass
class CafeMochaDecaf: pass
class CafeMochaDecafWhipped:
  def __init__(self):
    self.cost = 1.25
    self.description = \
      "Cafe Mocha decaf whipped cream"
  def getCost(self):
    return self.cost
  def getDescription(self):
    return self.description

class CafeMochaExtraEspresso: pass
class CafeMochaExtraEspressoWhipped: pass
class CafeMochaWet: pass
class CafeMochaWetWhipped: pass
class CafeMochaWhipped: pass

class CafeLatte: pass
class CafeLatteDecaf: pass
class CafeLatteDecafWhipped: pass
class CafeLatteExtraEspresso: pass
class CafeLatteExtraEspressoWhipped: pass
class CafeLatteWet: pass
class CafeLatteWetWhipped: pass
class CafeLatteWhipped: pass

cappuccino = Cappuccino()
print (cappuccino.getDescription() + ": $" +
  `cappuccino.getCost()`)

cafeMocha = CafeMochaDecafWhipped()
print (cafeMocha.getDescription()
  + ": $" + `cafeMocha.getCost()`)
#:~
```

And here is the corresponding output: Add Comment

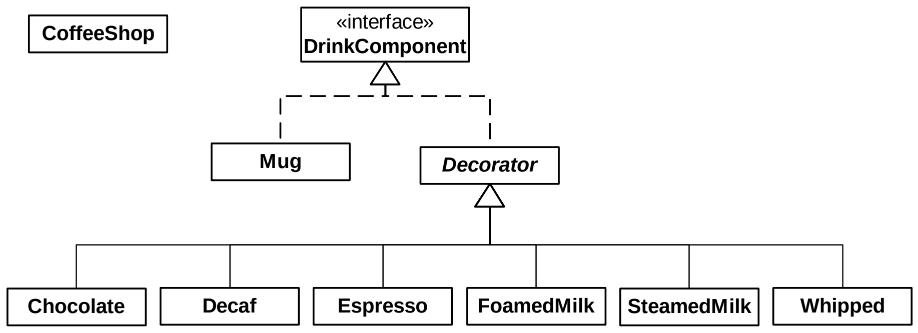
```
Cappucino: $1.0Cafe Mocha decaf whipped cream: $1.25
```

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. Add Comment

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. Add Comment

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. Add Comment



Methods invoked on the Decorator can in turn invoke methods in the component, and can of course perform processing before or after the invocation. Add Comment

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

```
class Espresso(Decorator):
  cost = 0.75f
  description = " espresso"
  public Espresso(DrinkComponent):
    Decorator.__init__(self, component)

  def getTotalCost(self):
    return self.component.getTotalCost() + cost

  def getDescription(self):
    return self.component.getDescription() +
      description
```

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

```
#: cX:decorator:alldecorators:CoffeeShop.py
# Coffee example using decorators

class DrinkComponent:
  def getDescription(self):
    return self.__class__.__name__
  def getTotalCost(self):
    return self.__class__.cost

class Mug(DrinkComponent):
  cost = 0.0

class Decorator(DrinkComponent):
  def __init__(self, drinkComponent):
    self.component = drinkComponent
  def getTotalCost(self):
    return self.component.getTotalCost() + \
      DrinkComponent.getTotalCost(self)
  def getDescription(self):
    return self.component.getDescription() + \
      ' ' + DrinkComponent.getDescription(self)

class Espresso(Decorator):
  cost = 0.75
  def __init__(self, drinkComponent):
    Decorator.__init__(self, drinkComponent)

class Decaf(Decorator):
  cost = 0.0
  def __init__(self, drinkComponent):
    Decorator.__init__(self, drinkComponent)

class FoamedMilk(Decorator):
  cost = 0.25
  def __init__(self, drinkComponent):
    Decorator.__init__(self, drinkComponent)

class SteamedMilk(Decorator):
  cost = 0.25
  def __init__(self, drinkComponent):
    Decorator.__init__(self, drinkComponent)

class Whipped(Decorator):
  cost = 0.25
  def __init__(self, drinkComponent):
    Decorator.__init__(self, drinkComponent)

class Chocolate(Decorator):
  cost = 0.25
  def __init__(self, drinkComponent):
    Decorator.__init__(self, drinkComponent)

cappuccino = Espresso(FoamedMilk(Mug()))
print cappuccino.getDescription().strip() + \
  ": $" + `cappuccino.getTotalCost()`

cafeMocha = Espresso(SteamedMilk(Chocolate(
  Whipped(Decaf(Mug())))))

print cafeMocha.getDescription().strip() + \
  ": $" + `cafeMocha.getTotalCost()`
#:~
```

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. Add Comment

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

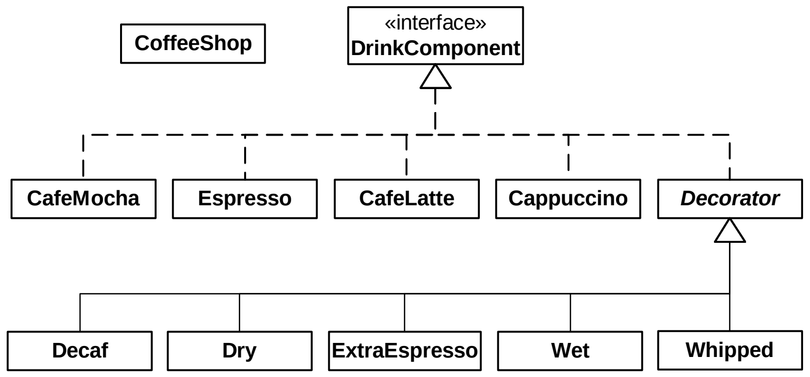
```
plainCap = Espresso(FoamedMilk(Mug()))
```

Creating a decaf Café Mocha with whipped cream requires an even longer description. Add Comment

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. Add Comment

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. Add Comment



Here is how to create a basic selection, as well as a decorated selection:

```
#: cX:decorator:compromise:CoffeeShop.py
# Coffee example with a compromise of basic
# combinations and decorators

class DrinkComponent:
  def getDescription(self):
    return self.__class__.__name__
  def getTotalCost(self):
    return self.__class__.cost

class Espresso(DrinkComponent):
  cost = 0.75

class EspressoConPanna(DrinkComponent):
  cost = 1.0

class Cappuccino(DrinkComponent):
  cost = 1.0

class CafeLatte(DrinkComponent):
  cost = 1.0

class CafeMocha(DrinkComponent):
  cost = 1.25

class Decorator(DrinkComponent):
  def __init__(self, drinkComponent):
    self.component = drinkComponent
  def getTotalCost(self):
    return self.component.getTotalCost() + \
      DrinkComponent.getTotalCost(self)
  def getDescription(self):
    return self.component.getDescription() + \
      ' ' + DrinkComponent.getDescription(self)

class ExtraEspresso(Decorator):
  cost = 0.75
  def __init__(self, drinkComponent):
    Decorator.__init__(self, drinkComponent)

class Whipped(Decorator):
  cost = 0.50
  def __init__(self, drinkComponent):
    Decorator.__init__(self, drinkComponent)

class Decaf(Decorator):
  cost = 0.0
  def __init__(self, drinkComponent):
    Decorator.__init__(self, drinkComponent)

class Dry(Decorator):
  cost = 0.0
  def __init__(self, drinkComponent):
    Decorator.__init__(self, drinkComponent)

class Wet(Decorator):
  cost = 0.0
  def __init__(self, drinkComponent):
    Decorator.__init__(self, drinkComponent)

cappuccino = Cappuccino()
print cappuccino.getDescription() + ": $" + \
  `cappuccino.getTotalCost()`

cafeMocha = Whipped(Decaf(CafeMocha()))
print cafeMocha.getDescription() + ": $" + \
  `cafeMocha.getTotalCost()`
#:~
```

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. Add Comment

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. Add Comment

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. Add Comment

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. Add Comment

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. Add Comment
2. Repeat Exercise 1 for the compromise approach. Add Comment
3. 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. Add Comment

Y: Iterators: decoupling algorithms from containers

This chapter has not had any significant translation yet.

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. Add Comment

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. Add Comment

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). Add Comment

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*). Add Comment

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. Add Comment

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. Add Comment

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. Add Comment

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. Add Comment

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: Add Comment

```
# util:TypedIterator.py

class TypedIterator(Iterator):
  private Iterator imp
  private Class type
  def __init__(self, Iterator it, Class type):
    imp = it
    self.type = type

  def hasNext(self):
    return imp.hasNext()

  def remove(self): imp.remove()
  def next(self):
    Object obj = imp.next()
    if(!type.isInstance(obj))
      throw ClassCastException(
        "TypedIterator for type " + type +
        " encountered type: " + obj.getClass())
    return obj
# :~
```

5: Factories: encapsulating object 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. Add Comment

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. Add Comment

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. Add Comment

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: Add Comment

```
#: c05:shapefact1:ShapeFactory1.py
# A simple static factory method.
from __future__ import generators
import random

class Shape(object):
  # Create based on class name:
  def factory(type):
    #return eval(type + "()")
    if type == "Circle": return Circle()
    if type == "Square": return Square()
    assert 1, "Bad shape creation: " + type
  factory = staticmethod(factory)

class Circle(Shape):
  def draw(self): print "Circle.draw"
  def erase(self): print "Circle.erase"

class Square(Shape):
  def draw(self): print "Square.draw"
  def erase(self): print "Square.erase"
```

```
# Generate shape name strings:
def shapeNameGen(n):
  types = Shape.__subclasses__()
  for i in range(n):
    yield random.choice(types).__name__

shapes = \
  [ Shape.factory(i) for i in shapeNameGen(7)]

for shape in shapes:
  shape.draw()
  shape.erase()
#:~
```

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).Add Comment

Note that this example also shows the new Python 2.2 **staticmethod( )** technique for creating static methods in a class. Add Comment

I have also used a tool which is new in Python 2.2 called a *generator*. A generator is a special case of a factory: it's a factory that takes no arguments in order to create a new object. Normally you hand some information to a factory in order to tell it what kind of object to create and how to create it, but a generator has some kind of internal algorithm that tells it what and how to build. It "generates out of thin air" rather than being told what to create. Add Comment

Now, this may not look consistent with the code you see above: Add Comment

```
for i in shapeNameGen(7)
```

looks like there's an initialization taking place. This is where a generator is a bit strange – when you call a function that contains a **yield** statement (**yield** is a new keyword that determines that a function is a generator), that function actually returns a generator object that has an iterator. This iterator is implicitly used in the **for** statement above, so it appears that you are iterating through the generator function, not what it returns. This was done for convenience of use. Add Comment

Thus, the code that you write is actually a kind of factory, that creates the generator objects that do the actual generation. You can use the generator explicitly if you want, for example: Add Comment

```
gen = shapeNameGen(7)
print gen.next()
```

So **next( )** is the iterator method that's actually called to generate the next object, and it takes no arguments. **shapeNameGen( )** is the factory, and **gen** is the generator. Add Comment

Inside the generator-factory, you can see the call to **__subclasses__( )**, which produces a list of references to each of the subclasses of **Shape** (which must be inherited from **object** for this to work). You should be aware, however, that this only works for the first level of inheritance from **Item**, so if you were to inherit a new class from **Circle**, it wouldn't show up in the list generated by **__subclasses__( )**. If you need to create a deeper hierarchy this way, you must recurse the **__subclasses__( )** list. Add Comment

Also note that in **shapeNameGen( )** the statement Add Comment

```
types = Shape.__subclasses__()
```

Is only executed when the generator object is produced; each time the **next( )** method of this generator object is called (which, as noted above, may happen implicitly), only the code in the **for** loop will be executed, so you don't have wasteful execution (as you would if this were an ordinary function). Add Comment

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.py** 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: Add Comment

```
#: c05:shapefact2:ShapeFactory2.py
# Polymorphic factory methods.
from __future__ import generators
import random

class ShapeFactory:
  factories = {}
  def addFactory(id, shapeFactory):
    ShapeFactory.factories.put[id] = shapeFactory
  addFactory = staticmethod(addFactory)
  # A Template Method:
  def createShape(id):
    if not ShapeFactory.factories.has_key(id):
      ShapeFactory.factories[id] = \
        eval(id + '.Factory()')
    return ShapeFactory.factories[id].create()
  createShape = staticmethod(createShape)

class Shape(object): pass

class Circle(Shape):
  def draw(self): print "Circle.draw"
  def erase(self): print "Circle.erase"
  class Factory:
    def create(self): return Circle()

class Square(Shape):
  def draw(self):
    print "Square.draw"
  def erase(self):
    print "Square.erase"
  class Factory:
    def create(self): return Square()

def shapeNameGen(n):
  types = Shape.__subclasses__()
  for i in range(n):
    yield random.choice(types).__name__

shapes = [ ShapeFactory.createShape(i)
           for i in shapeNameGen(7)]

for shape in shapes:
  shape.draw()
  shape.erase()
#:~
```

Now the factory method appears in its own class, **ShapeFactory**, as the **create( )** method. The different types of shapes must each create their own factory with a **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 dictionary 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.py**) will work fine. Add Comment

Notice that the **ShapeFactory** must be initialized by loading its dictionary with factory objects, which takes place in the static initialization clause of each of the shape implementations. Add Comment

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. Add Comment

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: Add Comment

```
#: c05:Games.py
# An example of the Abstract Factory pattern.

class Obstacle:
  def action(self): pass
```

```
class Player:
  def interactWith(self, obstacle): pass

class Kitty(Player):
  def interactWith(self, obstacle):
    print "Kitty has encountered a",
    obstacle.action()

class KungFuGuy(Player):
  def interactWith(self, obstacle):
    print "KungFuGuy now battles a",
    obstacle.action()

class Puzzle(Obstacle):
  def action(self):
    print "Puzzle"

class NastyWeapon(Obstacle):
  def action(self):
    print "NastyWeapon"

# The Abstract Factory:
class GameElementFactory:
  def makePlayer(self): pass
  def makeObstacle(self): pass

# Concrete factories:
class KittiesAndPuzzles(GameElementFactory):
  def makePlayer(self): return Kitty()
  def makeObstacle(self): return Puzzle()

class KillAndDismember(GameElementFactory):
  def makePlayer(self): return KungFuGuy()
  def makeObstacle(self): return NastyWeapon()

class GameEnvironment:
  def __init__(self, factory):
    self.factory = factory
    self.p = factory.makePlayer()
    self.ob = factory.makeObstacle()
  def play(self):
    self.p.interactWith(self.ob)
```

```
g1 = GameEnvironment(KittiesAndPuzzles())
g2 = GameEnvironment(KillAndDismember())
g1.play()
g2.play()
#:~
```

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. Add Comment

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

Of course, the above scaffolding of **Obstacle**, **Player** and **GameElementFactory** (which was translated from the Java version of this example) is unnecessary – it's only required for languages that have static type checking. As long as the concrete Python classes follow the form of the required classes, we don't need any base classes: Add Comment

```
#: c05:Games2.py
# Simplified Abstract Factory.

class Kitty:
  def interactWith(self, obstacle):
    print "Kitty has encountered a",
    obstacle.action()

class KungFuGuy:
  def interactWith(self, obstacle):
    print "KungFuGuy now battles a",
    obstacle.action()

class Puzzle:
  def action(self): print "Puzzle"

class NastyWeapon:
  def action(self): print "NastyWeapon"

# Concrete factories:
```

```
class KittiesAndPuzzles:
  def makePlayer(self): return Kitty()
  def makeObstacle(self): return Puzzle()

class KillAndDismember:
  def makePlayer(self): return KungFuGuy()
  def makeObstacle(self): return NastyWeapon()

class GameEnvironment:
  def __init__(self, factory):
    self.factory = factory
    self.p = factory.makePlayer()
    self.ob = factory.makeObstacle()
  def play(self):
    self.p.interactWith(self.ob)

g1 = GameEnvironment(KittiesAndPuzzles())
g2 = GameEnvironment(KillAndDismember())
g1.play()
g2.play()
#:~
```

Another way to put this is that all inheritance in Python is implementation inheritance; since Python does its type-checking at runtime, there's no need to use interface inheritance so that you can upcast to the base type. Add Comment

You might want to study the two examples for comparison, however. Does the first one add enough useful information about the pattern that it's worth keeping some aspect of it? Perhaps all you need is "tagging classes" like this: Add Comment

```
class Obstacle: pass
class Player: pass
class GameElementFactory: pass
```

Then the inheritance serves only to indicate the type of the derived classes. Add Comment

Exercises

1. Add a class **Triangle** to **ShapeFactory1.py** Add Comment
2. Add a class **Triangle** to **ShapeFactory2.py** Add Comment

3. Add a new type of **GameEnvironment** called **GnomesAndFairies** to **GameEnvironment.py** Add Comment

4. Modify **ShapeFactory2.py** 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.). Add Comment

6: Function objects

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. Add Comment

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. Add Comment

Command: choosing the operation at run-time

This is the function object in its purest sense: a method that’s an object[14]. 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. Add Comment

```
#: c06:CommandPattern.py

class Command:
```

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

```
  def execute(self): pass

class Loony(Command):
  def execute(self):
    print "You're a loony."

class NewBrain(Command):
  def execute(self):
    print "You might even need a new brain."

class Afford(Command):
  def execute(self):
    print "I couldn't afford a whole new brain."

# An object that holds commands:
class Macro:
  def __init__(self):
    self.commands = []
  def add(self, command):
    self.commands.append(command)
  def run(self):
    for c in self.commands:
      c.execute()

macro = Macro()
macro.add(Loony())
macro.add(NewBrain())
macro.add(Afford())
macro.run()
#:~
```

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). Add Comment

Design Patterns says that "Commands are an object-oriented replacement for callbacks[15]." 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

[15] Page 235.

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." Add Comment

Strategy: choosing the algorithm at run-time

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 **c12:DirList.py**, 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. Add Comment

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: Add Comment

```
#: c06:StrategyPattern.py

# The strategy interface:
class FindMinima:
  # Line is a sequence of points:
  def algorithm(self, line) : pass

# The various strategies:
class LeastSquares(FindMinima):
  def algorithm(self, line):
    return [ 1.1, 2.2 ] # Dummy

class NewtonsMethod(FindMinima):
  def algorithm(self, line):
    return [ 3.3, 4.4 ]  # Dummy

class Bisection(FindMinima):
  def algorithm(self, line):
    return [ 5.5, 6.6 ] # Dummy

class ConjugateGradient(FindMinima):
  def algorithm(self, line):
    return [ 3.3, 4.4 ] # Dummy

# The "Context" controls the strategy:
class MinimaSolver:
  def __init__(self, strategy):
    self.strategy = strategy

  def minima(self, line):
    return self.strategy.algorithm(line)

  def changeAlgorithm(self, newAlgorithm):
    self.strategy = newAlgorithm

solver = MinimaSolver(LeastSquares())
line = [
    1.0, 2.0, 1.0, 2.0, -1.0, 3.0, 4.0, 5.0, 4.0
  ]
print solver.minima(line)
solver.changeAlgorithm(Bisection())
print solver.minima(line)
#:~
```

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. Add Comment

Strategy example from standard Python: **sort( )** takes a second optional argument that acts as a comparator object; this is a strategy. Add Comment

Chain of responsibility

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. Add Comment

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. Add Comment

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 Python 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. Add Comment

In **StrategyPattern.py**, 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: Add Comment

```
#: c06:ChainOfResponsibility.py

# Carry the information into the strategy:
class Messenger: pass

# The Result object carries the result data and
# whether the strategy was successful:
class Result:
  def __init__(self):
    self.succeeded = 0
  def isSuccessful(self):
    return self.succeeded
  def setSuccessful(self, succeeded):
    self.succeeded = succeeded

class Strategy:
  def __call__(messenger): pass
  def __str__(self):
    return "Trying " + self.__class__.__name__ \
      + " algorithm"

# Manage the movement through the chain and
# find a successful result:
class ChainLink:
  def __init__(self, chain, strategy):
    self.strategy = strategy
    self.chain = chain
    self.chain.append(self)

  def next(self):
    # Where this link is in the chain:
    location = self.chain.index(self)
    if not self.end():
      return self.chain[location + 1]

  def end(self):
    return (self.chain.index(self) + 1 >=
            len(self.chain))

  def __call__(self, messenger):
    r = self.strategy(messenger)
    if r.isSuccessful() or self.end(): return r
    return self.next()(messenger)

# For this example, the Messenger
# and Result can be the same type:
class LineData(Result, Messenger):
  def __init__(self, data):
    self.data = data
  def __str__(self): return `self.data`

class LeastSquares(Strategy):
  def __call__(self, messenger):
    print self
    linedata = messenger
    # [ Actual test/calculation here ]
    result = LineData([1.1, 2.2]) # Dummy data
    result.setSuccessful(0)
    return result

class NewtonsMethod(Strategy):
  def __call__(self, messenger):
    print self
    linedata = messenger
    # [ Actual test/calculation here ]
    result = LineData([3.3, 4.4]) # Dummy data
    result.setSuccessful(0)
    return result

class Bisection(Strategy):
  def __call__(self, messenger):
    print self
    linedata = messenger
    # [ Actual test/calculation here ]
    result = LineData([5.5, 6.6]) # Dummy data
    result.setSuccessful(1)
    return result

class ConjugateGradient(Strategy):
  def __call__(self, messenger):
    print self
    linedata = messenger
    # [ Actual test/calculation here ]
    result = LineData([7.7, 8.8]) # Dummy data
    result.setSuccessful(1)
    return result

solutions = []
solutions = [
  ChainLink(solutions, LeastSquares()),
  ChainLink(solutions, NewtonsMethod()),
  ChainLink(solutions, Bisection()),
  ChainLink(solutions, ConjugateGradient())
]

line = LineData([
  1.0, 2.0, 1.0, 2.0, -1.0,
  3.0, 4.0, 5.0, 4.0
])

print solutions[0](line)
#:~
```

Exercises

1. Use *Command* in Chapter 3, Exercise 1. Add Comment
2. 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? Add Comment

7: Changing the interface

Sometimes the problem that you're solving is as simple as "I don't have the interface that I want." Two of the patterns in *Design Patterns* solve this problem: *Adapter* takes one type and produces an interface to some other type. *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. Add Comment

Adapter

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. Add Comment

```python
#: c07:Adapter.py
# Variations on the Adapter pattern.

class WhatIHave:
  def g(self): pass
  def h(self): pass

class WhatIWant:
  def f(self): pass

class ProxyAdapter(WhatIWant):
  def __init__(self, whatIHave):
    self.whatIHave = whatIHave

  def f(self):
    # Implement behavior using
    # methods in WhatIHave:
    self.whatIHave.g()
    self.whatIHave.h()

class WhatIUse:
  def op(self, whatIWant):
    whatIWant.f()

# Approach 2: build adapter use into op():
class WhatIUse2(WhatIUse):
  def op(self, whatIHave):
    ProxyAdapter(whatIHave).f()

# Approach 3: build adapter into WhatIHave:
class WhatIHave2(WhatIHave, WhatIWant):
  def f(self):
    self.g()
    self.h()

# Approach 4: use an inner class:
class WhatIHave3(WhatIHave):
  class InnerAdapter(WhatIWant):
    def __init__(self, outer):
      self.outer = outer
    def f(self):
      self.outer.g()
      self.outer.h()
  def whatIWant(self):
    return WhatIHave3.InnerAdapter(self)

whatIUse = WhatIUse()
whatIHave = WhatIHave()
adapt= ProxyAdapter(whatIHave)
whatIUse2 = WhatIUse2()
whatIHave2 = WhatIHave2()
whatIHave3 = WhatIHave3()
whatIUse.op(adapt)
# Approach 2:
whatIUse2.op(whatIHave)
# Approach 3:
whatIUse.op(whatIHave2)
# Approach 4:
whatIUse.op(whatIHave3.whatIWant())
#:~
```

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. Add Comment

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. Add Comment

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

```
# c07:Facade.py
class A:
  def __init__(self, x): pass
class B:
  def __init__(self, x): pass
class C:
  def __init__(self, x): pass
```

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

class Facade:
  def makeA(x): return A(x)
  makeA = staticmethod(makeA)
  def makeB(x): return B(x)
  makeB = staticmethod(makeB)
  def makeC(x): return C(x)
  makeC = staticmethod(makeC)

# The client programmer gets the objects
# by calling the static methods:
a = Facade.makeA(1);
b = Facade.makeB(1);
c = Facade.makeC(1.0);
# :~
```

[rewrite this section using research from Larman's book]Add Comment

Exercises

1. Create an adapter class that automatically loads a two-dimensional array of objects into a dictionary as key-value pairs. Add Comment

8: Table-driven code: configuration flexibility

Table-driven code using anonymous inner classes

See **ListPerformance** example in TIJ from Chapter 9 Add Comment

Also **GreenHouse.py**

10: Callbacks

Decoupling code behavior

Observer, and a category of callbacks called "multiple dispatching (not in *Design Patterns*)" including the *Visitor* from *Design Patterns*. Add Comment

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. Add Comment

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. Add Comment

There are two types of objects used to implement the observer pattern in Python. 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**. Add Comment

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. Add Comment

----

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. Add Comment

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

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. Add Comment

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**. Add Comment

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. Add Comment

Observing flowers

Since Python doesn’t have standard library components to support the observer pattern (like Java does), we must first create one. The simplest thing to do is translate the Java standard library **Observer** and **Observable** classes. This also provides easier translation from Java code that uses these libraries. Add Comment

In trying to do this, we encounter a minor snag, which is the fact that Java has a **synchronized** keyword that provides built-in support for thread synchronization. We could certainly accomplish the same thing by hand, using code like this: Add Comment

```
import threading
class ToSynch:
  def __init__(self):
    self.mutex = threading.RLock()
    self.val = 1
  def aSynchronizedMethod(self):
    self.mutex.acquire()
    try:
      self.val += 1
      return self.val
    finally:
      self.mutex.release()
```

But this rapidly becomes tedious to write and to read. Peter Norvig provided me with a much nicer solution: Add Comment

```
#: util:Synchronization.py
'''Simple emulation of Java's 'synchronized'
keyword, from Peter Norvig.'''
import threading

def synchronized(method):
  def f(*args):
    self = args[0]
    self.mutex.acquire();
    # print method.__name__, 'acquired'
    try:
      return apply(method, args)
    finally:
      self.mutex.release();
      # print method.__name__, 'released'
  return f
```

```
def synchronize(klass, names=None):
  """Synchronize methods in the given class.
  Only synchronize the methods whose names are
  given, or all methods if names=None."""
  if type(names)==type(''): names = names.split()
  for (name, val) in klass.__dict__.items():
    if callable(val) and name != '__init__' and \
      (names == None or name in names):
        # print "synchronizing", name
        klass.__dict__[name] = synchronized(val)

# You can create your own self.mutex, or inherit
# from this class:
class Synchronization:
  def __init__(self):
    self.mutex = threading.RLock()
#:~
```

The **synchronized( )** function takes a method and wraps it in a function that adds the mutex functionality. The method is called inside this function: Add Comment

```
return apply(method, args)
```

and as the **return** statement passes through the **finally** clause, the mutex is released. Add Comment

This is in some ways the *Decorator* design pattern, but much simpler to create and use. All you have to say is:

```
myMethod = synchronized(myMethod)
```

To surround your method with a mutex. Add Comment

synchronize( ) is a convenience function that applies **synchronized( )** to an entire class, either all the methods in the class (the default) or selected methods which are named in a string as the second argument. Add Comment

Finally, for **synchronized( )** to work there must be a **self.mutex** created in every class that uses **synchronized( )**. This can be created by hand by the class author, but it's more consistent to use inheritance, so the base class **Synchronization** is provided. Add Comment

Here's a simple test of the **Synchronization** module.

```
#: util:TestSynchronization.py
from Synchronization import *
```

```
# To use for a method:
class C(Synchronization):
  def __init__(self):
    Synchronization.__init__(self)
    self.data = 1
  def m(self):
    self.data += 1
    return self.data
  m = synchronized(m)
  def f(self): return 47
  def g(self): return 'spam'

# So m is synchronized, f and g are not.
c = C()

# On the class level:
class D(C):
  def __init__(self):
    C.__init__(self)
  # You must override an un-synchronized method
  # in order to synchronize it (just like Java):
  def f(self): C.f(self)

# Synchronize every (defined) method in the class:
synchronize(D)
d = D()
d.f() # Synchronized
d.g() # Not synchronized
d.m() # Synchronized (in the base class)

class E(C):
  def __init__(self):
    C.__init__(self)
  def m(self): C.m(self)
  def g(self): C.g(self)
  def f(self): C.f(self)
# Only synchronizes m and g. Note that m ends up
# being doubly-wrapped in synchronization, which
# doesn't hurt anything but is inefficient:
synchronize(E, 'm g')
e = E()
e.f()
e.g()
e.m()
```

```
#:~
```

You must call the base class constructor for **Synchronization**, but that's all. In class **C** you can see the use of **synchronized( )** for **m**, leaving **f** and **g** alone. Class **D** has all it's methods synchronized en masse, and class **E** uses the convenience function to synchronize **m** and **g**. Note that since **m** ends up being synchronized twice, it will be entered and left twice for every call, which isn't very desirable [there may be a fix for this] Add Comment

```
#: util:Observer.py
# Class support for "observer" pattern.
from Synchronization import *

class Observer:
  def update(observable, arg):
    '''Called when the observed object is
    modified. You call an Observable object's
    notifyObservers method to notify all the
    object's observers of the change.'''
    pass

class Observable(Synchronization):
  def __init__(self):
    self.obs = []
    self.changed = 0
    Synchronization.__init__(self)

  def addObserver(self, observer):
    if observer not in self.obs:
      self.obs.append(observer)

  def deleteObserver(self, observer):
    self.obs.remove(observer)

  def notifyObservers(self, arg = None):
    '''If 'changed' indicates that this object
    has changed, notify all its observers, then
    call clearChanged(). Each observer has its
    update() called with two arguments: this
    observable object and the generic 'arg'.'''

    self.mutex.acquire()
    try:
      if not self.changed: return
      # Make a local copy in case of synchronous
      # additions of observers:
      localArray = self.obs[:]
      self.clearChanged()
    finally:
      self.mutex.release()
    # Updating is not required to be synchronized:
    for observer in localArray:
      observer.update(self, arg)

  def deleteObservers(self): self.obs = []
  def setChanged(self): self.changed = 1
  def clearChanged(self): self.changed = 0
  def hasChanged(self): return self.changed
  def countObservers(self): return len(self.obs)

synchronize(Observable,
  "addObserver deleteObserver deleteObservers " +
  "setChanged clearChanged hasChanged " +
  "countObservers")
#:~
```

Using this library, here is an example of the observer pattern: Add Comment

```
#: c10:ObservedFlower.py
# Demonstration of "observer" pattern.
import sys
sys.path += ['../util']
from Observer import Observer, Observable

class Flower:
  def __init__(self):
    self.isOpen = 0
    self.openNotifier = Flower.OpenNotifier(self)
    self.closeNotifier= Flower.CloseNotifier(self)
  def open(self): # Opens its petals
    self.isOpen = 1
    self.openNotifier.notifyObservers()
    self.closeNotifier.open()
  def close(self): # Closes its petals
    self.isOpen = 0
    self.closeNotifier.notifyObservers()
    self.openNotifier.close()
  def closing(self): return self.closeNotifier
  class OpenNotifier(Observable):
    def __init__(self, outer):
      Observable.__init__(self)
      self.outer = outer
      self.alreadyOpen = 0
    def notifyObservers(self):
      if self.outer.isOpen and \
      not self.alreadyOpen:
        self.setChanged()
        Observable.notifyObservers(self)
        self.alreadyOpen = 1
    def close(self):
      self.alreadyOpen = 0

  class CloseNotifier(Observable):
    def __init__(self, outer):
      Observable.__init__(self)
      self.outer = outer
      self.alreadyClosed = 0
    def notifyObservers(self):
      if not self.outer.isOpen and \
      not self.alreadyClosed:
        self.setChanged()
        Observable.notifyObservers(self)
        self.alreadyClosed = 1
    def open(self):
      alreadyClosed = 0

class Bee:
  def __init__(self, name):
    self.name = name
    self.openObserver = Bee.OpenObserver(self)
    self.closeObserver = Bee.CloseObserver(self)
  # An inner class for observing openings:
  class OpenObserver(Observer):
    def __init__(self, outer):
      self.outer = outer
    def update(self, observable, arg):
      print "Bee " + self.outer.name + \
        "'s breakfast time!"
  # Another inner class for closings:
  class CloseObserver(Observer):
    def __init__(self, outer):
      self.outer = outer
    def update(self, observable, arg):
      print "Bee " + self.outer.name + \
        "'s bed time!"

class Hummingbird:
  def __init__(self, name):
    self.name = name
    self.openObserver = \
      Hummingbird.OpenObserver(self)
    self.closeObserver = \
      Hummingbird.CloseObserver(self)
  class OpenObserver(Observer):
    def __init__(self, outer):
      self.outer = outer
    def update(self, observable, arg):
      print "Hummingbird " + self.outer.name + \
        "'s breakfast time!"
  class CloseObserver(Observer):
    def __init__(self, outer):
      self.outer = outer
    def update(self, observable, arg):
      print "Hummingbird " + self.outer.name + \
        "'s bed time!"

f = Flower()
ba = Bee("Eric")
bb = Bee("Eric 0.5")
ha = Hummingbird("A")
hb = Hummingbird("B")
f.openNotifier.addObserver(ha.openObserver)
f.openNotifier.addObserver(hb.openObserver)
f.openNotifier.addObserver(ba.openObserver)
f.openNotifier.addObserver(bb.openObserver)
f.closeNotifier.addObserver(ha.closeObserver)
f.closeNotifier.addObserver(hb.closeObserver)
f.closeNotifier.addObserver(ba.closeObserver)
f.closeNotifier.addObserver(bb.closeObserver)
# Hummingbird 2 decides to sleep in:
f.openNotifier.deleteObserver(hb.openObserver)
# A change that interests observers:
f.open()
f.open() # It's already open, no change.
# Bee 1 doesn't want to go to bed:
f.closeNotifier.deleteObserver(ba.closeObserver)
```

```
f.close()
f.close() # It's already closed; no change
f.openNotifier.deleteObservers()
f.open()
f.close()
#:~
```

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**. Add Comment

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. Add Comment

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. Add Comment

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. Add Comment

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. Add Comment

[[NOTE: this example has not been converted. See further down for a version that has the GUI but not the Observers, in PythonCard.]]

```
#  c10:BoxObserver.py
# Demonstration of Observer pattern using
# Java's built-in observer classes.

# You must inherit a type of Observable:
class BoxObservable(Observable):
  def notifyObservers(self, Object b):
    # Otherwise it won't propagate changes:
    setChanged()
    super.notifyObservers(b)

class BoxObserver(JFrame):
  Observable notifier = BoxObservable()
  def __init__(self, int grid):
    setTitle("Demonstrates Observer pattern")
    Container cp = getContentPane()
    cp.setLayout(GridLayout(grid, grid))
    for(int x = 0 x < grid x++)
      for(int y = 0 y < grid y++)
        cp.add(OCBox(x, y, notifier))

  def main(self, String[] args):
    int grid = 8
    if(args.length > 0)
      grid = Integer.parseInt(args[0])
    JFrame f = BoxObserver(grid)
    f.setSize(500, 400)
    f.setVisible(1)
    # JDK 1.3:
    f.setDefaultCloseOperation(EXIT_ON_CLOSE)
    # Add a WindowAdapter if you have JDK 1.2

class OCBox(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.darkGray, Color.gray, Color.green,
    Color.lightGray, Color.magenta,
    Color.orange, Color.pink, Color.red,
    Color.white, Color.yellow
  static final Color newColor():
    return colors[
      (int)(Math.random() * colors.length)
    ]

  def __init__(self, int x, int y, Observable
notifier):
    self.x = x
    self.y = y
    notifier.addObserver(self)
    self.notifier = notifier
    addMouseListener(ML())

  def paintComponent(self, Graphics g):
    super.paintComponent(g)
    g.setColor(cColor)
    Dimension s = getSize()
    g.fillRect(0, 0, s.width, s.height)

  class ML(MouseAdapter):
    def mousePressed(self, MouseEvent e):
      notifier.notifyObservers(OCBox.self)

  def update(self, 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( )**. Add Comment

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. Add Comment

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. Add Comment

Here is a version of the above that doesn't use the Observer pattern, written by Kevin Altis using PythonCard, and placed here as a starting point for a translation that does include Observer:

```
#: c10:BoxObserver.py
""" Written by Kevin Altis as a first-cut for
converting BoxObserver to Python. The Observer
hasn't been integrated yet.
To run this program, you must:
Install WxPython from
http://www.wxpython.org/download.php
Install PythonCard. See:
http://pythoncard.sourceforge.net
"""

from PythonCardPrototype import log, model
import random

GRID = 8

class ColorBoxesTest(model.Background):
  def on_openBackground(self, target, event):
    self.document = []
    for row in range(GRID):
      line = []
      for column in range(GRID):
        line.append(self.createBox(row, column))
      self.document.append(line[:])
  def createBox(self, row, column):
    colors = ['black', 'blue', 'cyan',
    'darkGray', 'gray', 'green',
    'lightGray', 'magenta',
    'orange', 'pink', 'red',
    'white', 'yellow']
    width, height = self.panel.GetSizeTuple()
    boxWidth = width / GRID
    boxHeight = height / GRID
    log.info("width:" + str(width) +
      " height:" + str(height))
    log.info("boxWidth:" + str(boxWidth) +
      " boxHeight:" + str(boxHeight))
    # use an empty image, though some other
    # widgets would work just as well
    boxDesc = {'type':'Image',
      'size':(boxWidth, boxHeight), 'file':''}
    name = 'box-%d-%d' % (row, column)
    # There is probably a 1 off error in the
    # calculation below since the boxes should
    # probably have a slightly different offset
    # to prevent overlaps
    boxDesc['position'] =
      (column * boxWidth, row * boxHeight)
    boxDesc['name'] = name
    boxDesc['backgroundColor'] =
      random.choice(colors)
    self.components[name] =  boxDesc
    return self.components[name]

  def changeNeighbors(self, row, column, color):

    # This algorithm will result in changing the
    # color of some boxes more than once, so an
    # OOP solution where only neighbors are asked
    # to change or boxes check to see if they are
    # neighbors before changing would be better
    # per the original example does the whole grid
    # need to change its state at once like in a
    # Life program? should the color change
    # in the propogation of another notification
    # event?

    for r in range(max(0, row - 1),
                   min(GRID, row + 2)):
      for c in range(max(0, column - 1),
                     min(GRID, column + 2)):
        self.document[r][c].backgroundColor=color

  # this is a background handler, so it isn't
  # specific to a single widget. Image widgets
  # don't have a mouseClick event (wxCommandEvent
  # in wxPython)
  def on_mouseUp(self, target, event):
    prefix, row, column = target.name.split('-')
    self.changeNeighbors(int(row), int(column),
                         target.backgroundColor)

if __name__ == '__main__':
  app = model.PythonCardApp(ColorBoxesTest)
  app.MainLoop()
#:~
```

This is the resource file for running the program (see PythonCard for details):

```
#: c10:BoxObserver.rsrc.py
{'stack':{'type':'Stack',
          'name':'BoxObserver',
    'backgrounds': [
      { 'type':'Background',
        'name':'bgBoxObserver',
        'title':'Demonstrates Observer pattern',
        'position':(5, 5),
        'size':(500, 400),
        'components': [

] # end components
} # end background
] # end backgrounds
} }
#:~
```

Exercises

1. Using the approach in **Synchronization.py**, create a tool that will automatically wrap all the methods in a class to provide an execution trace, so that you can see the name of the method and when it is entered and exited. Add Comment
2. 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. Add Comment
3. Modify **BoxObserver.py** 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. Add Comment

11: 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? Add Comment

The answer starts with something you probably don't think about: Python performs only single dispatching. That is, if you are performing an operation on more than one object whose type is unknown, Python 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. Add Comment

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. Add Comment

Here's an example of multiple dispatching:

```
#: c11:PaperScissorsRock.py
# Demonstration of multiple dispatching.
from __future__ import generators
import random

# An enumeration type:
class Outcome:
  def __init__(self, value, name):
    self.value = value
    self.name = name
  def __str__(self): return self.name
  def __eq__(self, other):
      return self.value == other.value

Outcome.WIN = Outcome(0, "win")
Outcome.LOSE = Outcome(1, "lose")
Outcome.DRAW = Outcome(2, "draw")

class Item(object):
  def __str__(self):
    return self.__class__.__name__

class Paper(Item):
  def compete(self, item):
    # First dispatch: self was Paper
    return item.evalPaper(self)
  def evalPaper(self, item):
    # Item was Paper, we're in Paper
    return Outcome.DRAW
  def evalScissors(self, item):
    # Item was Scissors, we're in Paper
    return Outcome.WIN
  def evalRock(self, item):
    # Item was Rock, we're in Paper
    return Outcome.LOSE

class Scissors(Item):
  def compete(self, item):
    # First dispatch: self was Scissors
    return item.evalScissors(self)
  def evalPaper(self, item):
    # Item was Paper, we're in Scissors
    return Outcome.LOSE
  def evalScissors(self, item):
    # Item was Scissors, we're in Scissors
    return Outcome.DRAW
  def evalRock(self, item):
    # Item was Rock, we're in Scissors
    return Outcome.WIN

class Rock(Item):
  def compete(self, item):
    # First dispatch: self was Rock
    return item.evalRock(self)
  def evalPaper(self, item):
    # Item was Paper, we're in Rock
    return Outcome.WIN
  def evalScissors(self, item):
    # Item was Scissors, we're in Rock
    return Outcome.LOSE
  def evalRock(self, item):
    # Item was Rock, we're in Rock
    return Outcome.DRAW
```

```
def match(item1, item2):
  print "%s <--> %s : %s" % (
    item1, item2, item1.compete(item2))

# Generate the items:
def itemPairGen(n):
  # Create a list of instances of all Items:
  Items = Item.__subclasses__()
  for i in range(n):
    yield (random.choice(Items)(),
           random.choice(Items)())

for item1, item2 in itemPairGen(20):
  match(item1, item2)
#:~
```

This was a fairly literal translation from the Java version, and one of the things you might notice is that the information about the various combinations is encoded into each type of **Item**. It actually ends up being a kind of table, except that it is spread out through all the classes. This is not very easy to maintain if you ever expect to modify the behavior or to add a new **Item** class. Instead, it can be more sensible to make the table explicit, like this: Add Comment

```
#: c11:PaperScissorsRock2.py
# Multiple dispatching using a table
from __future__ import generators
import random

class Outcome:
  def __init__(self, value, name):
    self.value = value
    self.name = name
  def __str__(self): return self.name
  def __eq__(self, other):
      return self.value == other.value

Outcome.WIN = Outcome(0, "win")
Outcome.LOSE = Outcome(1, "lose")
Outcome.DRAW = Outcome(2, "draw")

class Item(object):
  def compete(self, item):
    # Use a tuple for table lookup:
```

```
    return outcome[self.__class__, item.__class__]
  def __str__(self):
    return self.__class__.__name__

class Paper(Item): pass
class Scissors(Item): pass
class Rock(Item): pass

outcome = {
  (Paper, Rock): Outcome.WIN,
  (Paper, Scissors): Outcome.LOSE,
  (Paper, Paper): Outcome.DRAW,
  (Scissors, Paper): Outcome.WIN,
  (Scissors, Rock): Outcome.LOSE,
  (Scissors, Scissors): Outcome.DRAW,
  (Rock, Scissors): Outcome.WIN,
  (Rock, Paper): Outcome.LOSE,
  (Rock, Rock): Outcome.DRAW,
}

def match(item1, item2):
  print "%s <--> %s : %s" % (
    item1, item2, item1.compete(item2))

# Generate the items:
def itemPairGen(n):
  # Create a list of instances of all Items:
  Items = Item.__subclasses__()
  for i in range(n):
    yield (random.choice(Items)(),
           random.choice(Items)())

for item1, item2 in itemPairGen(20):
  match(item1, item2)
#:~
```

It’s a tribute to the flexibility of dictionaries that a tuple can be used as a key just as easily as a single object. Add Comment

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? Add Comment

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. Add Comment

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. Add Comment

```
#: c11:FlowerVisitors.py
# Demonstration of "visitor" pattern.
from __future__ import generators
import random

# The Flower hierarchy cannot be changed:
class Flower(object):
  def accept(self, visitor):
    visitor.visit(self)
  def pollinate(self, pollinator):
    print self, "pollinated by", pollinator
  def eat(self, eater):
    print self, "eaten by", eater
  def __str__(self):
    return self.__class__.__name__

class Gladiolus(Flower): pass
class Runuculus(Flower): pass
class Chrysanthemum(Flower): pass

class Visitor:
  def __str__(self):
    return self.__class__.__name__

class Bug(Visitor): pass
class Pollinator(Bug): pass
class Predator(Bug): pass

# Add the ability to do "Bee" activities:
class Bee(Pollinator):
  def visit(self, flower):
      flower.pollinate(self)

# Add the ability to do "Fly" activities:
class Fly(Pollinator):
  def visit(self, flower):
      flower.pollinate(self)

# Add the ability to do "Worm" activities:
class Worm(Predator):
  def visit(self, flower):
      flower.eat(self)

def flowerGen(n):
  flwrs = Flower.__subclasses__()
  for i in range(n):
    yield random.choice(flwrs)()

# It's almost as if I had a method to Perform
# various "Bug" operations on all Flowers:
bee = Bee()
fly = Fly()
worm = Worm()
for flower in flowerGen(10):
  flower.accept(bee)
  flower.accept(fly)
  flower.accept(worm)
#:~
```
Add Comment

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. Add Comment

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.py**). 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." Add Comment

3. Modify **PaperScissorsRock.py** 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? Add Comment

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

12: Pattern refactoring

This chapter has not had any significant translation yet. Add Comment

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?" Add Comment

This process is similar to what Martin Fowler talks about in his book *Refactoring: Improving the Design of Existing Code*[16] (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." Add Comment

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. Add Comment

[16] Addison-Wesley, 1999.

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. Add Comment

```
# c12:recyclea:RecycleA.py
# Recycling with RTTI.

class Trash:
  private double weight
  def __init__(self, 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()
      print (
        "weight of " +
        # Using RTTI to get type
        # information about the class:
        t.getClass().getName() +
        " = " + t.getWeight())

    print "Total value = " + val

class Aluminum(Trash):
  static double val  = 1.67f
  def __init__(self, double wt): .__init__(wt)
  double getValue(): return val
  static void setValue(double newval):
    val = newval

class Paper(Trash):
  static double val = 0.10f
  def __init__(self, double wt): .__init__(wt)
  double getValue(): return val
  static void setValue(double newval):
    val = newval

class Glass(Trash):
  static double val = 0.23f
  def __init__(self, double wt): .__init__(wt)
  double getValue(): return val
  static void setValue(double newval):
    val = newval

class RecycleA(UnitTest):
  Collection
    bin = ArrayList(),
    glassBin = ArrayList(),
    paperBin = ArrayList(),
    alBin = ArrayList()
  def __init__(self):
    # Fill up the Trash bin:
    for(int i = 0 i < 30 i++)
      switch((int)(Math.random() * 3)):
        case 0 :
          bin.add(new
            Aluminum(Math.random() * 100))
          break
        case 1 :
          bin.add(new
            Paper(Math.random() * 100))
          break
        case 2 :
          bin.add(new
            Glass(Math.random() * 100))

  def test(self):
    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())

  def main(self, String args[]):
    RecycleA().test()

# :~
```

In the source code listings available for this book, this file will be placed in the subdirectory **recyclea** that branches off from the subdirectory **c12** (for Chapter 12). 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. Add Comment

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. Add Comment

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**. Add Comment

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. Add Comment

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. Add Comment

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. Add Comment

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. Add Comment

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. Add Comment

“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. Add Comment

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

```
    for(int i = 0 i < 30 i++)
      switch((int)(Math.random() * 3)):
        case 0 :
          bin.add(new
            Aluminum(Math.random() * 100))
          break
        case 1 :
          bin.add(new
            Paper(Math.random() * 100))
          break
        case 2 :
          bin.add(new
            Glass(Math.random() * 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. Add Comment

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. Add Comment

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**: Add Comment

```
class Messenger:
  int type
  # Must change this to add another type:
  static final int MAX_NUM = 4
  double data
  def __init__(self, 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. Add Comment

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 Aluminum(i.data)
      case 1:
        return Paper(i.data)
      case 2:
        return Glass(i.data)
      # Two lines here:
      case 3:
        return 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. Add Comment

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

```
    for(int i = 0 i < 30 i++)
      bin.add(
        Trash.factory(
          Messenger(
            (int)(Math.random() * Messenger.MAX_NUM),
            Math.random() * 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. Add Comment

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. Add Comment

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. Add Comment

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. Add Comment

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. Add Comment

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. Add Comment

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. Add Comment

```
# c12:trash:Trash.py
# Base class for Trash recycling examples.

class Trash:
  private double weight
  def __init__(self, double wt): weight = wt
  def __init__(self):
  def getValue(self)
  def getWeight(self): return weight
  # Sums the value of Trash given an
  # Iterator to any container of Trash:
  def sumValue(self, 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()
      print (
        "weight of " +
        # Using RTTI to get type
        # information about the class:
        t.getClass().getName() +
        " = " + t.getWeight())

    print "Total value = " + val

  # Remainder of class provides
  # support for prototyping:
  private static List trashTypes =
    ArrayList()
  def factory(self, Messenger info):
    for(int i = 0 i < len(trashTypes) i++):
      # Somehow determine the type
      # to create, and create one:
      Class tc = (Class)trashTypes.get(i)
      if (tc.getName().index(info.id) != -1):
        try:
          # Get the dynamic constructor method
          # that takes a double argument:
          Constructor ctor = tc.getConstructor(
              Class[]{ double.class )
          # Call the constructor
          # to create a object:
          return (Trash)ctor.newInstance(
            Object[]{Double(info.data))
         catch(Exception ex):
          ex.printStackTrace(System.err)
          throw RuntimeException(
            "Cannot Create Trash")
    # Class was not in the list. Try to load it,
    # but it must be in your class path!
    try:
      print "Loading " + info.id
      trashTypes.add(Class.forName(info.id))
     catch(Exception e):
      e.printStackTrace(System.err)
      throw RuntimeException(
        "Prototype not found")

    # Loaded successfully.
    # Recursive call should work:
    return factory(info)

  public static class Messenger:
    public String id
    public double data
    public Messenger(String name, double val):
      id = name
      data = val

# :~
```

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. Add Comment

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. Add Comment

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: Add Comment

```
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. Add Comment

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: Add Comment

```
Object[]{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. Add Comment

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. Add Comment

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. Add Comment

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). Add Comment

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. Add Comment

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): Add Comment

```
# c12:trash:Aluminum.py
# The Aluminum class with prototyping.

class Aluminum(Trash):
  private static double val = 1.67f
  def __init__(self, double wt): .__init__(wt)
  def getValue(self): return val
  def setValue(self, double newVal):
    val = newVal

# :~
# c12:trash:Paper.py
# The Paper class with prototyping.

class Paper(Trash):
  private static double val = 0.10f
  def __init__(self, double wt): .__init__(wt)
  def getValue(self): return val
  def setValue(self, double newVal):
    val = newVal

# :~
# c12:trash:Glass.py
# The Glass class with prototyping.

class Glass(Trash):
  private static double val = 0.23f
  def __init__(self, double wt): .__init__(wt)
  def getValue(self): return val
  def setValue(self, double newVal):
    val = newVal

# :~
```

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

```
# c12:trash:Cardboard.py
# The Cardboard class with prototyping.

class Cardboard(Trash):
  private static double val = 0.23f
  def __init__(self, double wt): .__init__(wt)
  def getValue(self): return val
  def setValue(self, double newVal):
    val = newVal

# :~
```

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

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: Add Comment

```
# c12:trash:Trash.dat
c12.trash.Glass:54
c12.trash.Paper:22
c12.trash.Paper:11
c12.trash.Glass:17
c12.trash.Aluminum:89
c12.trash.Paper:88
c12.trash.Aluminum:76
c12.trash.Cardboard:96
c12.trash.Aluminum:25
c12.trash.Aluminum:34
c12.trash.Glass:11
c12.trash.Glass:68
c12.trash.Glass:43
c12.trash.Aluminum:27
c12.trash.Cardboard:44
c12.trash.Aluminum:18
c12.trash.Paper:91
c12.trash.Glass:63
c12.trash.Glass:50
c12.trash.Glass:80
c12.trash.Aluminum:81
c12.trash.Cardboard:12
c12.trash.Glass:12
```

```
c12.trash.Glass:54
c12.trash.Aluminum:36
c12.trash.Aluminum:93
c12.trash.Glass:93
c12.trash.Paper:80
c12.trash.Glass:36
c12.trash.Glass:12
c12.trash.Glass:60
c12.trash.Paper:66
c12.trash.Aluminum:36
c12.trash.Cardboard:22
# :~
```

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

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. Add Comment

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

```
# c12:trash:ParseTrash.py
# Parse file contents into Trash objects,
# placing each into a Fillable holder.

class ParseTrash:
  def fillBin(String filename, Fillable bin):
    for line in open(filename).readlines():
      String type = line.substring(0,
        line.index(':')).strip()
      double weight = Double.valueOf(
        line.substring(line.index(':') + 1)
          .strip()).doubleValue()
      bin.addTrash(
        Trash.factory(
          Trash.Messenger(type, weight)))

  # Special case to handle Collection:
  def fillBin(String filename, Collection bin):
    fillBin(filename, FillableCollection(bin))
```

```
# :~
```

In **RecycleA.py**, 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( )**: Add Comment

```
# c12:trash:Fillable.py
# Any object that can be filled with Trash.

class Fillable:
  def addTrash(self, 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: Add Comment

```
# c12:trash:FillableCollection.py
# Adapter that makes a Collection Fillable.

class FillableCollection(Fillable):
  private Collection c
  def __init__(self, Collection cc):
    c = cc

  def addTrash(self, 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.py**: Add Comment

```
  public static void
  fillBin(String filename, Collection bin):
    fillBin(filename, 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.py**.) Add Comment

Recycling with prototyping

Now you can see the revised version of **RecycleA.py** using the prototyping technique: Add Comment

```
# c12:recycleap:RecycleAP.py
# Recycling with RTTI and Prototypes.

class RecycleAP(UnitTest):
  Collection
    bin = ArrayList(),
    glassBin = ArrayList(),
    paperBin = ArrayList(),
    alBin = ArrayList()
  def __init__(self):
    # Fill up the Trash bin:
    ParseTrash.fillBin(
      "../trash/Trash.dat", bin)

  def test(self):
    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())

  def main(self, String args[]):
    RecycleAP().test()

# :~
```

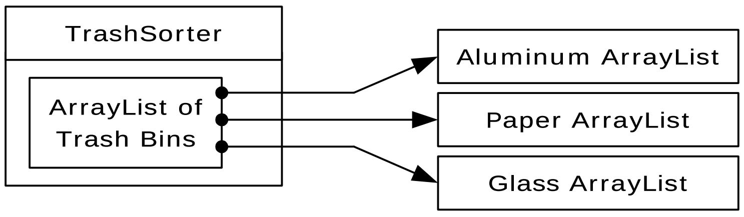
All of the **Trash** objects, as well as the **ParseTrash** and support classes, are now part of the package **c12.trash**, so they are simply imported. Add Comment

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. Add Comment

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. Add Comment

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: Add Comment



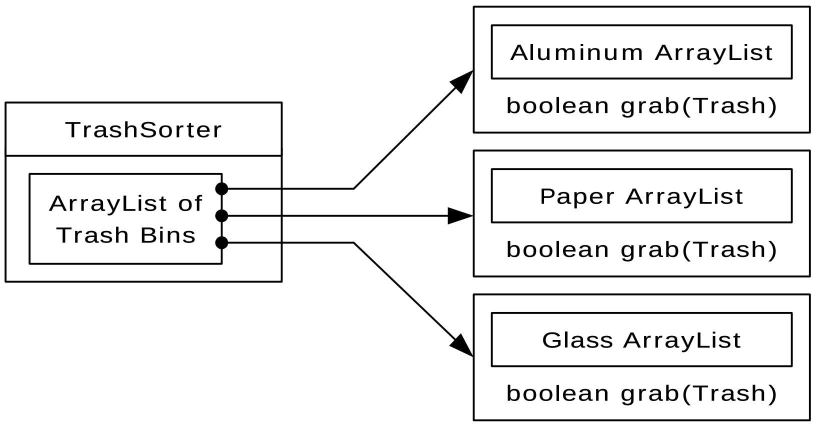
The **TrashSorter** object initialization must now be changed whenever a new type of **Trash** is added to the model. You could imagine that the **TrashSorter** class might look something like this: Add Comment

```
class TrashSorter(ArrayList):
  def sort(self, 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: Add Comment

```
TrashSorter ts = TrashSorter()
ts.add(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: Add Comment



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. Add Comment

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. Add Comment

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. Add Comment

Here is the new version of the program:

```
# c12:recycleb:RecycleB.py
# Containers that grab objects of interest.

# A container that admits only the right type
# of Trash (established in the constructor):
class Tbin:
  private Collection list = ArrayList()
  private Class type
  def __init__(self, Class binType): type = binType
  def grab(self, Trash t):
    # Comparing class types:
    if(t.getClass().equals(type)):
      list.add(t)
      return 1 # Object grabbed

    return 0 # Object not grabbed

  def iterator(self):
    return list.iterator()

class TbinList(ArrayList):
  def sort(self, Trash t):
    Iterator e = iterator() # Iterate over self
    while(e.hasNext())
      if(((Tbin)e.next()).grab(t)) return
    # Need a Tbin for this type:
```

```
    add(Tbin(t.getClass()))
    sort(t) # Recursive call

class RecycleB(UnitTest):
  Collection bin = ArrayList()
  TbinList trashBins = TbinList()
  def __init__(self):
    ParseTrash.fillBin("../trash/Trash.dat",bin)

  def test(self):
    Iterator it = bin.iterator()
    while(it.hasNext())
      trashBins.sort((Trash)it.next())
    Iterator e = trashBins.iterator()
    while(e.hasNext()):
      Tbin b = (Tbin)e.next()
      Trash.sumValue(b.iterator())

    Trash.sumValue(bin.iterator())

  def main(self, String args[]):
    RecycleB().test()

# :~
```

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. Add Comment

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. Add Comment

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.Add Comment

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.py**. 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. Add Comment

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. Add Comment

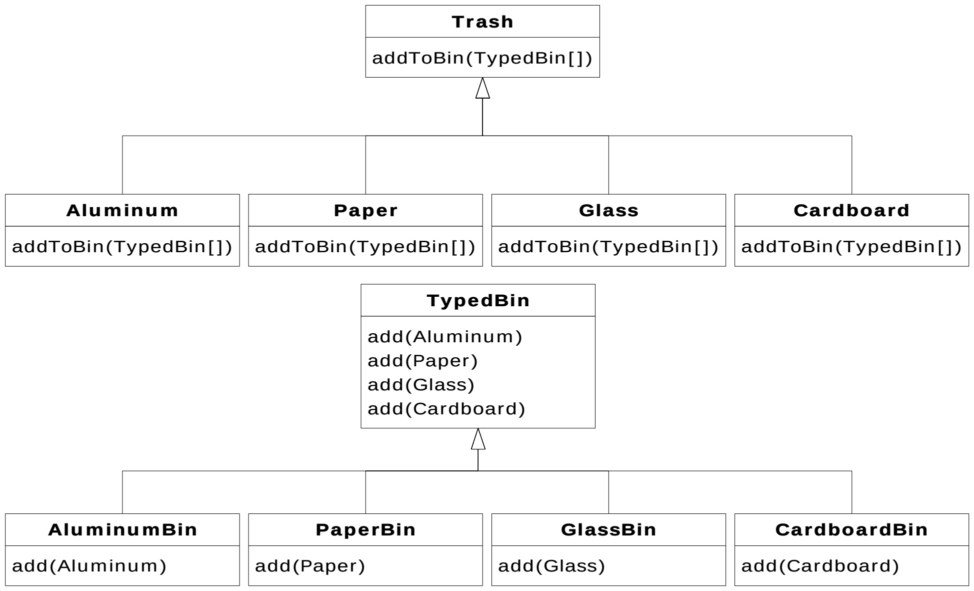
The previous examples first sorted by type, then acted on sequences of elements that were all of a particular type. But whenever you find yourself picking out particular types, stop and think. The whole idea of polymorphism (dynamically-bound method calls) is to handle type-specific information for you. So why are you hunting for types? Add Comment

The answer is something you probably don't think about: Python performs only single dispatching. That is, if you are performing an operation on more than one object whose type is unknown, Python 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. Add Comment

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*). Add Comment

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. Add Comment

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

```
# c12:doubledispatch:TypedBinMember.py
# An class for adding the double
# dispatching method to the trash hierarchy
# without modifying the original hierarchy.

class TypedBinMember:
  # The 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: Add Comment

```
# c12:doubledispatch:DDAluminum.py
# Aluminum for double dispatching.

class DDAluminum(Aluminum)
    implements TypedBinMember:
  def __init__(self, double wt): .__init__(wt)
  def addToBin(self, TypedBin[] tb):
    for(int i = 0 i < tb.length i++)
      if(tb[i].add(self))
        return 1
    return 0

# :~
# c12:doubledispatch:DDPaper.py
# Paper for double dispatching.

class DDPaper(Paper)
    implements TypedBinMember:
  def __init__(self, double wt): .__init__(wt)
  def addToBin(self, TypedBin[] tb):
    for(int i = 0 i < tb.length i++)
      if(tb[i].add(self))
        return 1
    return 0

# :~
# c12:doubledispatch:DDGlass.py
# Glass for double dispatching.

class DDGlass(Glass)
    implements TypedBinMember:
```

```
  def __init__(self, double wt): .__init__(wt)
  def addToBin(self, TypedBin[] tb):
    for(int i = 0 i < tb.length i++)
      if(tb[i].add(self))
        return 1
    return 0

# :~
# c12:doubledispatch:DDCardboard.py
# Cardboard for double dispatching.

class DDCardboard(Cardboard)
    implements TypedBinMember:
  def __init__(self, double wt): .__init__(wt)
  def addToBin(self, TypedBin[] tb):
    for(int i = 0 i < tb.length i++)
      if(tb[i].add(self))
        return 1
    return 0

# :~
```

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. Add Comment

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

```
# c12:doubledispatch:TypedBin.py
# A container for the second dispatch.

class TypedBin:
  Collection c = ArrayList()
  def addIt(self, Trash t):
    c.add(t)
    return 1
  def iterator(self):
    return c.iterator()

  def add(self, DDAluminum a):
    return 0

  def add(self, DDPaper a):
    return 0

  def add(self, DDGlass a):
    return 0

  def add(self, DDCardboard a):
    return 0

# :~
```

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. Add Comment

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**.) Add Comment

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**: Add Comment

```
# c12:doubledispatch:DDTrash.dat
DDGlass:54
DDPaper:22
DDPaper:11
DDGlass:17
DDAluminum:89
DDPaper:88
DDAluminum:76
DDCardboard:96
```

```
DDAluminum:25
DDAluminum:34
DDGlass:11
DDGlass:68
DDGlass:43
DDAluminum:27
DDCardboard:44
DDAluminum:18
DDPaper:91
DDGlass:63
DDGlass:50
DDGlass:80
DDAluminum:81
DDCardboard:12
DDGlass:12
DDGlass:54
DDAluminum:36
DDAluminum:93
DDGlass:93
DDPaper:80
DDGlass:36
DDGlass:12
DDGlass:60
DDPaper:66
DDAluminum:36
DDCardboard:22
# :~
```

Here's the rest of the program: Add Comment

```
# c12:doubledispatch:DoubleDispatch.py
# Using multiple dispatching to handle more
# than one unknown type during a method call.

class AluminumBin(TypedBin):
  def add(self, DDAluminum a):
    return addIt(a)

class PaperBin(TypedBin):
  def add(self, DDPaper a):
    return addIt(a)

class GlassBin(TypedBin):
  def add(self, DDGlass a):
    return addIt(a)
```

```
class CardboardBin(TypedBin):
  def add(self, DDCardboard a):
    return addIt(a)

class TrashBinSet:
  private TypedBin[] binSet =:
    AluminumBin(),
    PaperBin(),
    GlassBin(),
    CardboardBin()

  def sortIntoBins(self, Collection bin):
    Iterator e = bin.iterator()
    while(e.hasNext()):
      TypedBinMember t =
        (TypedBinMember)e.next()
      if(!t.addToBin(binSet))
        System.err.println("Couldn't add " + t)

  public TypedBin[] binSet(): return binSet

class DoubleDispatch(UnitTest):
  Collection bin = ArrayList()
  TrashBinSet bins = TrashBinSet()
  def __init__(self):
    # ParseTrash still works, without changes:
    ParseTrash.fillBin("DDTrash.dat", bin)

  def test(self):
    # Sort from the master bin into
    # the individually-typed bins:
    bins.sortIntoBins(bin)
    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())

  def main(self, String args[]):
    DoubleDispatch().test()

# :~
```

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. Add Comment

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. Add Comment

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**. Add Comment

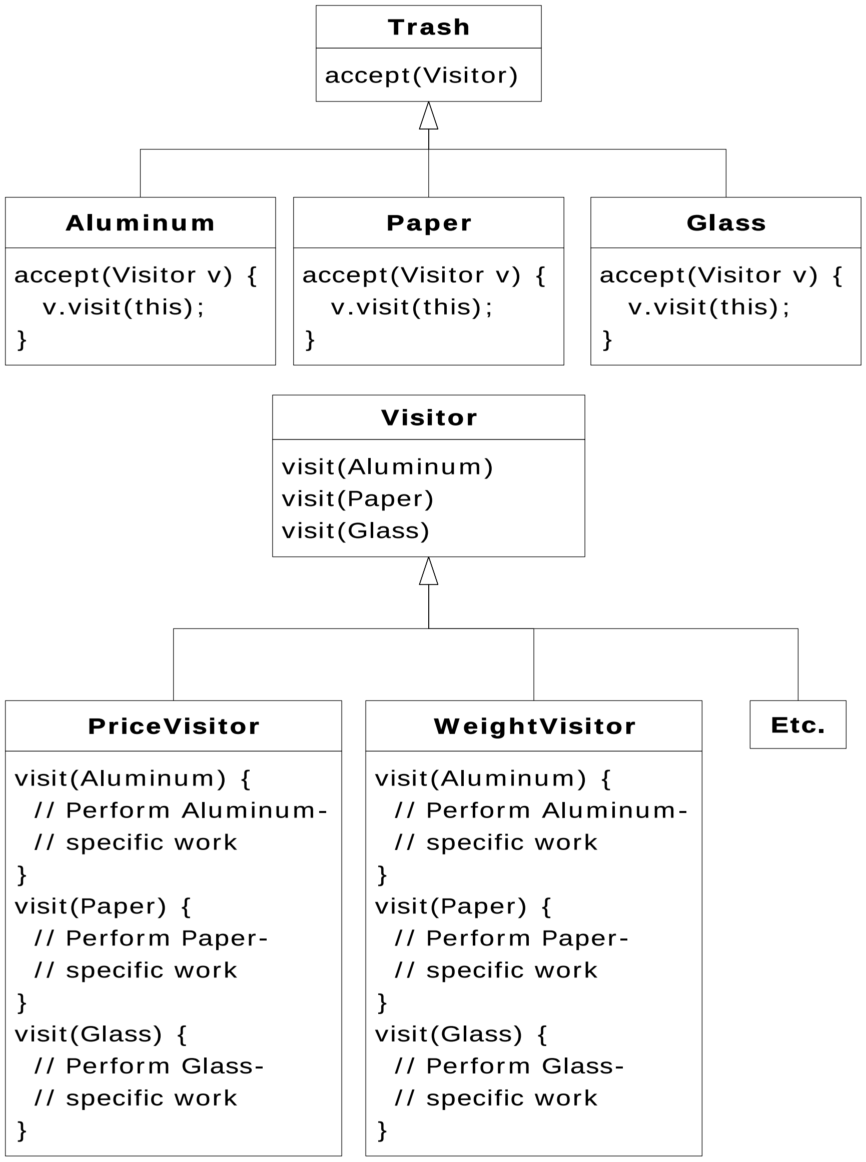
The *Visitor* pattern

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

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? Add Comment

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. Add Comment

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: Add Comment



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

```
PriceVisitor pv = 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**. Add Comment

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. Add Comment

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). Add Comment

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. Add Comment

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

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

```
# c12:trashvisitor:Visitable.py
# An class to add visitor functionality
# to the Trash hierarchy without
# modifying the base class.

class Visitable:
  # The method:
  def accept(self, Visitor v)
# :~
```

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

```
# c12:trashvisitor:Visitor.py
# The base class for visitors.
```

```
class Visitor:
  def visit(self, Aluminum a)
  def visit(self, Paper p)
  def visit(self, Glass g)
  def visit(self, 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: Add Comment

```
# c12:trashvisitor:VAluminum.py
# Taking the previous approach of creating a
# specialized Aluminum for the visitor pattern.

class VAluminum(Aluminum)
    implements Visitable:
  def __init__(self, double wt): .__init__(wt)
  def accept(self, Visitor v):
    v.visit(self)

# :~
```

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*. Add Comment

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 Python 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[17]: Add Comment

```
# c12:trashvisitor:VisitableDecorator.py
# A decorator that adapts the generic Trash
# classes to the visitor pattern.

class VisitableDecorator
extends Trash implements Visitable:
  private Trash delegate
  private Method dispatch
  def __init__(self, Trash t):
    delegate = t
    try:
      dispatch = Visitor.class.getMethod (
        "visit", Class[]: t.getClass()
      )
     catch (Exception ex):
      ex.printStackTrace()

  def getValue(self):
    return delegate.getValue()

  def getWeight(self):
    return delegate.getWeight()

  def accept(self, Visitor v):
    try:
      dispatch.invoke(v, Object[]{delegate)
     catch (Exception ex):
      ex.printStackTrace()

# :~
```

[[Description of Reflection use]] Add Comment

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**: Add Comment

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

```
# c12:trashvisitor:FillableVisitor.py
# Adapter Decorator that adds the visitable
# decorator as the Trash objects are
# being created.

class FillableVisitor
implements Fillable:
  private Fillable f
  def __init__(self, Fillable ff): f = ff
  def addTrash(self, Trash t):
    f.addTrash(VisitableDecorator(t))

# :~
```

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

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

```
# c12:trashvisitor:TrashVisitor.py
# The "visitor" pattern with VisitableDecorators.

# Specific group of algorithms packaged
# in each implementation of Visitor:
class PriceVisitor(Visitor):
  private double alSum # Aluminum
  private double pSum # Paper
  private double gSum # Glass
  private double cSum # Cardboard
  def visit(self, Aluminum al):
    double v = al.getWeight() * al.getValue()
    print "value of Aluminum= " + v
    alSum += v

  def visit(self, Paper p):
    double v = p.getWeight() * p.getValue()
    print "value of Paper= " + v
    pSum += v

  def visit(self, Glass g):
    double v = g.getWeight() * g.getValue()
    print "value of Glass= " + v
    gSum += v

  def visit(self, Cardboard c):
    double v = c.getWeight() * c.getValue()
    print "value of Cardboard = " + v
    cSum += v

  def total(self):
    print (
      "Total Aluminum: $" + alSum +
      "\n Total Paper: $" + pSum +
      "\nTotal Glass: $" + gSum +
      "\nTotal Cardboard: $" + cSum +
      "\nTotal: $" +
        (alSum + pSum + gSum + cSum))

class WeightVisitor(Visitor):
  private double alSum # Aluminum
  private double pSum # Paper
  private double gSum # Glass
  private double cSum # Cardboard
  def visit(self, Aluminum al):
    alSum += al.getWeight()
    print ("weight of Aluminum = "
        + al.getWeight())

  def visit(self, Paper p):
    pSum += p.getWeight()
    print ("weight of Paper = "
        + p.getWeight())

  def visit(self, Glass g):
    gSum += g.getWeight()
    print ("weight of Glass = "
        + g.getWeight())

  def visit(self, Cardboard c):
    cSum += c.getWeight()
    print ("weight of Cardboard = "
        + c.getWeight())

  def total(self):
    print (
      "Total weight Aluminum: "  + alSum +
      "\nTotal weight Paper: " + pSum +
      "\nTotal weight Glass: " + gSum +
      "\nTotal weight Cardboard: " + cSum +
      "\nTotal weight: " +
        (alSum + pSum + gSum + cSum))

class TrashVisitor(UnitTest):
  Collection bin = ArrayList()
  PriceVisitor pv = PriceVisitor()
  WeightVisitor wv = WeightVisitor()
  def __init__(self):
    ParseTrash.fillBin("../trash/Trash.dat",
      FillableVisitor(
        FillableCollection(bin)))

  def test(self):
    Iterator it = bin.iterator()
    while(it.hasNext()):
      Visitable v = (Visitable)it.next()
      v.accept(pv)
      v.accept(wv)

    pv.total()
    wv.total()

  def main(self, String args[]):
    TrashVisitor().test()

# :~
```

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. Add Comment

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. Add Comment

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. Add Comment

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**. Add Comment

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**. Add Comment

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. Add Comment

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. Add Comment

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

ArrayLists, 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 (suggested by Larry O'Brien) 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. Add Comment

Our example will again build on the structure of the **Trash** types in **package c12.Trash** (and the **Trash.dat** file used there can be used here without change): Add Comment

```
# c12:dynatrash:DynaTrash.py
# Using a Map of Lists and RTTI
# to automatically sort trash into
# ArrayLists. This solution, despite the
# use of RTTI, is extensible.

# Generic TypeMap works in any situation:
class TypeMap:
  private Map t = HashMap()
  def add(self, Object o):
    Class type = o.getClass()
    if(t.has_key(type))
      ((List)t.get(type)).add(o)
    else:
      List v = ArrayList()
      v.add(o)
      t.put(type,v)

  def get(self, Class type):
    return (List)t.get(type)

  def keys(self):
    return t.keySet().iterator()

# Adapter class to allow callbacks
# from ParseTrash.fillBin():
class TypeMapAdapter(Fillable):
  TypeMap map
  def __init__(self, TypeMap tm): map = tm
  def addTrash(self, Trash t): map.add(t)

class DynaTrash(UnitTest):
  TypeMap bin = TypeMap()
  def __init__(self):
    ParseTrash.fillBin("../trash/Trash.dat",
      TypeMapAdapter(bin))

  def test(self):
    Iterator keys = bin.keys()
    while(keys.hasNext())
      Trash.sumValue(
        bin.get((Class)keys.next()).iterator())

  def main(self, String args[]):
    DynaTrash().test()

# :~
```

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. Add Comment

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. Add Comment

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: Add Comment

```
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. Add Comment

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**. Add Comment

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. Add Comment

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. Add Comment

Summary

Coming up with a design such as **TrashVisitor.py** 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.py** allows you to easily add new *functionality* without disturbing the **Trash** hierarchy. There's more code in **TrashVisitor.py**, 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. Add Comment

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! Add Comment

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. Add Comment

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. Add Comment

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: Add Comment

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). Add Comment

Exercises

1. Add a class **Plastic** to **TrashVisitor.py**. Add Comment
2. Add a class **Plastic** to **DynaTrash.py**. Add Comment
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. Add Comment

13: Projects

This chapter has not had any significant translation yet.

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]] Add Comment

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. Add Comment

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. Add Comment

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. Add Comment

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. Add Comment

An example implementation by Jeremy Meyer:

```
# c13:Maze.py

class Maze(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):
      print “Enter filename“
      System.exit(0)

    Maze m = Maze()
    m.load(args[0])
    Frame f = Frame()
    f.setSize(m.width*20, m.height*20)
    f.add(m)
    Rat r = Rat(m, 0, 0)
    f.setVisible(1)

  def __init__(self):
    lines = Vector()
    setBackground(Color.lightGray)

  synchronized public boolean
  isEmptyXY(int x, int y):
    if (x < 0) x += width
    if (y < 0) y += height
    # Use mod arithmetic to bring rat in line:
    byte[] by =
      (byte[])(lines.elementAt(y%height))
    return by[x%width]==' '
```

```
synchronized public void
setXY(int x, int y, byte newByte):
  if (x < 0) x += width
  if (y < 0) y += height
  byte[] by =
    (byte[])(lines.elementAt(y%height))
  by[x%width] = newByte
  repaint()

public void
load(String filename) throws IOException:
  String currentLine = null
  BufferedReader br = BufferedReader(
    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 = len(lines)
  br.close()

def update(self, Graphics g): paint(g)
public void paint (Graphics g):
  int canvasHeight = self.getBounds().height
  int canvasWidth  = self.getBounds().width
  if (height < 1 || width < 1)
    return # nothing to do
  int width =
    ((byte[])(lines.elementAt(0))).length
  for (int y = 0 y < len(lines) 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



# :~
# c13:Rat.py

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
  def __init__(self, Maze maze, int xStart, int
yStart):
    myRatNo = ratCount++
    print ("Rat no." + myRatNo +
      " ready to scurry.")
    prison = maze
    x = xStart
    y = yStart
    prison.setXY(x,y, (byte)'R')
    Thread():
      def run(self){ scurry()
    .start()
```

```
def scurry(self):
  # Try and maintain direction if possible.
  # Horizontal backward
  boolean ratCanMove = 1
  while(ratCanMove):
    ratCanMove = 0
    # South
    if (prison.isEmptyXY(x, y + 1)):
      vertDir = 1 horizDir = 0
      ratCanMove = 1

    # North
    if (prison.isEmptyXY(x, y - 1))
      if (ratCanMove)
        Rat(prison, x, y-1)
        # Rat can move already, so give
        # this choice to the next rat.
      else:
        vertDir = -1 horizDir = 0
        ratCanMove = 1

    # West
    if (prison.isEmptyXY(x-1, y))
      if (ratCanMove)
        Rat(prison, x-1, y)
        # Rat can move already, so give
        # this choice to the next rat.
      else:
        vertDir = 0 horizDir = -1
        ratCanMove = 1

    # East
    if (prison.isEmptyXY(x+1, y))
      if (ratCanMove)
        Rat(prison, x+1, y)
        # Rat can move already, so give
        # this choice to the next rat.
      else:
        vertDir = 0 horizDir = 1
        ratCanMove = 1

    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 ie):

    print ("Rat no." + myRatNo +
      " can't move..dying..aarrgggh.")

# :~
```

The maze initialization file: Add Comment

```
#:! c13:Amaze.txt
   * **      *  * **      *
 ***    * *******    * ****
     ***          ***
 *****   **********   *****
 * * * * **  ** * * * **  *
   * * *  * **  * * *  * **
 *     **     *     **
   * **   * **  * **   * **
 *** *  *** ***** *  *** **
 *      *   * *      *   *
   * ** * *     * ** * *
# :~
```

Other maze resources

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

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) XMLAdd Comment