Thinking in Patterns with Java

Bruce Eckel

President, MindView, Inc.

Revision 0.5a

Please note that this document is in its initial form, and much remains to be done. Corrections and comments should go to http://www.topica.com/lists/TIPatterns/

Contents

Preface

Introduction

1: The pattern concept

This book introduces the important and yet non-traditional “patterns” approach to program design.

Probably the most important step forward in object-oriented design is the “design patterns” movement, chronicled in *Design Patterns*, by Gamma, Helm, Johnson & Vlissides (Addison-Wesley, 1995).[1] That book shows 23 different solutions to particular classes of problems. In this book, the basic concepts of design patterns will be introduced along with examples. This should whet your appetite to read *Design Patterns* by Gamma, et. al., a source of what has now become an essential, almost mandatory, vocabulary for OOP programmers.

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

What is a pattern?

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

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

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

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

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

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

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

You've also already seen another pattern that appears in *Design Patterns*: the *iterator* (Java 1.0 and 1.1 capriciously calls it the **Enumeration**; Java 2 containers use "iterator"). This hides the particular implementation of the container as you're stepping through and selecting the elements one by one. The iterator allows you to write generic code that performs an operation on all of the elements in a sequence without regard to the way that sequence is built. Thus your generic code can be used with any container that can produce an iterator.

Pattern taxonomy

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

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

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

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

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

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

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

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. This is used in the Java libraries, but here’s a more direct example:

```java
//: c01:SingletonPattern.java
// The Singleton design pattern: you can
// never instantiate more than one.

// Since this isn't inherited from a Cloneable
// base class and cloneability isn't added,
// making it final prevents cloneability from
// being added through inheritance:
final class Singleton {
  private static Singleton s = new Singleton(47);
  private int i;
  private Singleton(int x) { i = x; }
  public static Singleton getReference() {
    return s;
  }
  public int getValue() { return i; }
  public void setValue(int x) { i = x; }
}

public class SingletonPattern {
  public static void main(String[] args) {
    Singleton s = Singleton.getReference();
    System.out.println(s.getValue());
    Singleton s2 = Singleton.getReference();
    s2.setValue(9);
    System.out.println(s.getValue());
    try {
      // Can't do this: compile-time error.
      // Singleton s3 = (Singleton)s2.clone();
    } catch(Exception e) {
      e.printStackTrace(System.err);
    }
  }
} ///:~
```

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

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

Java also allows the creation of objects through cloning. In this example, making the class **final** prevents cloning. Since **Singleton** is inherited directly from **Object**, the **clone( )** method remains **protected** so it cannot be used (doing so produces a compile-time error). However, if you're inheriting from a class hierarchy that has already overridden **clone( )** as **public** and implemented **Cloneable**, the way to prevent cloning is to override **clone( )** and throw a **CloneNotSupportedException** as described in Appendix A. (You could also override **clone( )** and simply return **this**, but that would be deceiving since the client programmer would think they were cloning the object, but would instead still be dealing with the original.)

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

Classifying patterns

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

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

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

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

The *Design Patterns* book has a section on each of its 23 patterns along with one or more examples for each, typically in C++ but sometimes in Smalltalk. (You'll find that this doesn't matter too much since you can easily translate the concepts from either language into Java.) This book will 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.

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*.

The development challenge

Issues of development, the UML process, Extreme Programming.

Is evaluation valuable? The Capability Immaturity Model:

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

Pair programming research:

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

Exercises

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

2. Using **SingletonPattern.java** 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.

2: Unit Testing

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

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.

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.

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.

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.

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!).

The other problem was that I was lacking a structure for the testing system. Eventually, I started hearing about unit testing and JUnit[1], which 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.

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

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

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.

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:

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

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).

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.

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.

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

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**:

```
//: com:bruceeckel:test:UnitTest.java
// The basic unit testing class
package com.bruceeckel.test;
import java.util.ArrayList;

public class UnitTest {
  static String testID;
  static ArrayList errors = new ArrayList();
  // Override cleanup() if test object
  // creation allocates non-memory
  // resources that must be cleaned up:
  protected void cleanup() {}
  // Verify the truth of a condition:
  protected final void assert(boolean condition){
    if(!condition)
      errors.add("failed: " + testID);
  }
} ///:~
```

The only testing method [[So far]] is **assert( )**, 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.

Unlike JUnit (which uses **setUp( )** and **tearDown( )** methods), test objects will be built using ordinary Java 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.

Writing tests

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

```java
//: c02:TestDemo.java
// Creating a test
import com.bruceeckel.test.*;

public class TestDemo {
  private static int objCounter = 0;
  private int id = ++objCounter;
  public TestDemo(String s) {
    System.out.println(s + ": count = " + id);
  }
  public void close() {
    System.out.println("Cleaning up: " + id);
  }
  public boolean someCondition() { return true; }
  public static class Test extends UnitTest {
    TestDemo test1 = new TestDemo("test1");
    TestDemo test2 = new TestDemo("test2");
    public void cleanup() {
      test2.close();
      test1.close();
    }
    public void testA() {
      System.out.println("TestDemo.testA");
      assert(test1.someCondition());
    }
    public void testB() {
      System.out.println("TestDemo.testB");
      assert(test2.someCondition());
      assert(TestDemo.objCounter != 0);
    }
    // Causes the build to halt:
    //! public void test3() { assert(false); }
  }
} ///:~
```

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.

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).

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**.

The testing methods use the **assert( )** 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 **assert( )** arguments are usually more complicated than this; you'll see more examples throughout the rest of this book.

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.

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

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

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.

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:

```java
//: c02:TestDemo2.java
// Inheriting from a class that
// already has a test is no problem.
import com.bruceeckel.test.*;

public class TestDemo2 extends TestDemo {
  public TestDemo2(String s) { super(s); }
  // You can even use the same name
  // as the test class in the base class:
  public static class Test extends UnitTest {
    public void testA() {
      System.out.println("TestDemo2.testA");
      assert(1 + 1 == 2);
    }
    public void testB() {
      System.out.println("TestDemo2.testB");
      assert(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.

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**.

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.

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**).

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:

```
//: c02:Testable.java

public class Testable {
  private void f1() {}
  void f2() {} // "Friendly": package access
  protected void f3() {} // Also package access
  public void f4() {}
} ///:~
```

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:

```
//: c02:TooMuchAccess.java
import com.bruceeckel.test.*;

public class TooMuchAccess extends UnitTest {
  Testable tst = new Testable();
  public void test1() {
    tst.f2(); // Oops!
    tst.f3(); // Oops!
    tst.f4(); // OK
  }
} ///:~
```

You can solve the problem by moving **TooMuchAccess.java** into its own subdirectory, thereby putting it in its own default package (thus a different package from **Testable.java**). 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):

```
//: c02:testable:Testable.java
package c02.testable;

public class Testable {
  private void f1() {}
  void f2() {} // "Friendly": package access
  protected void f3() {} // Also package access
  public void f4() {}
} ///:~
```

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

```
//: c02:test:BlackBoxTest.java
import c02.testable.*;
import com.bruceeckel.test.*;

public class BlackBoxTest extends UnitTest {
  Testable tst = new Testable();
  public void test1() {
    //! 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).

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.

```
//: com:bruceeckel:test:RunUnitTests.java
// Discovering the unit test
// class and running each test.
package com.bruceeckel.test;
import java.lang.reflect.*;
import java.util.Iterator;

public class RunUnitTests {
  public static void
  require(boolean requirement, String errmsg) {
    if(!requirement) {
      System.err.println(errmsg);
      System.exit(1);
    }
  }
  public static void main(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 new instance of the
             // test object is created and
             // cleaned up for each test:
             Object test = ut.newInstance();
             m.invoke(test, new 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.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.java**, un-comment **test3( )**, then type **make** and observe the results.

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

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

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).

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.

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).

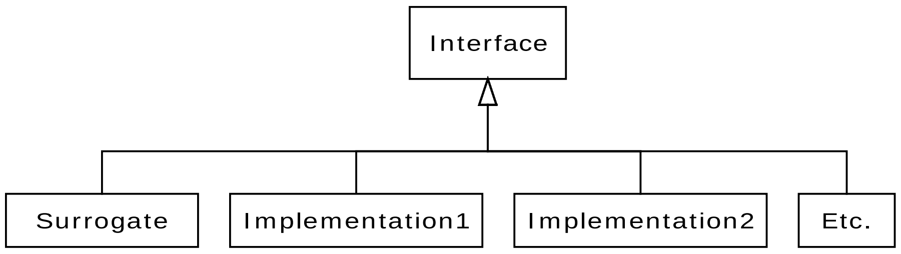
```
//: c03:TemplateMethod.java
// Simple demonstration of Template Method.
import com.bruceeckel.test.*;

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

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

public class TemplateMethod extends UnitTest {
  MyApp app = new MyApp();
  public void test() {
    // The MyApp constructor does all the work.
    // This just makes sure it will complete
    // without throwing an exception.
  }
  public static void main(String args[]) {
    new TemplateMethod().test();
  }
} ///:~
```

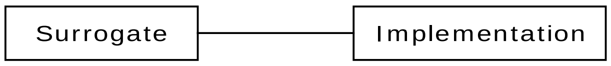
The base-class constructor is responsible for performing the necessary initialization and then starting the “engine” (the template method) that runs the application (in a GUI application, this “engine” would be the main event loop). The client programmer simply provides definitions for **customize1(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

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.

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





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

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

Proxy

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

```
//: c04:ProxyDemo.java
// Simple demonstration of the Proxy pattern.
import com.bruceeckel.test.*;

interface ProxyBase {
  void f();
  void g();
  void h();
}

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

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

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

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

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:

```
//: c04:StateDemo.java
// Simple demonstration of the State pattern.
import com.bruceeckel.test.*;

interface StateBase {
  void f();
  void g();
  void h();
}

class State implements StateBase {
  private StateBase implementation;
  public State(StateBase imp) {
    implementation = imp;
  }
  public void changeImp(StateBase newImp) {
    implementation = newImp;
  }
  // Pass method calls to the implementation:
  public void f() { implementation.f(); }
  public void g() { implementation.g(); }
  public void h() { implementation.h(); }
}

class Implementation1 implements StateBase {
  public void f() {
    System.out.println("Implementation1.f()");
  }
  public void g() {
    System.out.println("Implementation1.g()");
  }
  public void h() {
    System.out.println("Implementation1.h()");
  }
}

class Implementation2 implements StateBase {
  public void f() {
    System.out.println("Implementation2.f()");
  }
  public void g() {
    System.out.println("Implementation2.g()");
  }
  public void h() {
    System.out.println("Implementation2.h()");
  }
}

public class StateDemo extends UnitTest {
  static void run(State b) {
    b.f();
    b.g();
    b.h();
  }
  State b = new State(new Implementation1());
  public void test() {
    // This just makes sure it will complete
    // without throwing an exception.
    run(b);
    b.changeImp(new Implementation2());
    run(b);
  }
  public static void main(String args[]) {
    new StateDemo().test();
  }
} ///:~
```

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

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

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

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

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

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

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

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

StateMachine

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

The code that moves the system from one state to the next is often a *Template Method*, as seen in this example:

```java
//: c04:statemachine:StateMachineDemo.java
// Demonstrates StateMachine pattern
// and Template method.
package c04.statemachine;
import java.util.*;
import com.bruceeckel.test.*;

interface State {
  void run();
}

abstract class StateMachine {
  protected State currentState;
  abstract protected boolean changeState();
  // Template method:
  protected final void runAll() {
    while(changeState()) // Customizable
      currentState.run();
  }
}

// A different subclass for each state:

class Wash implements State {
  public void run() {
    System.out.println("Washing");
  }
}

class Spin implements State {
  public void run() {
    System.out.println("Spinning");
  }
}

class Rinse implements State {
  public void run() {
    System.out.println("Rinsing");
  }
}

class Washer extends StateMachine {
  private int i = 0;
  // The state table:
  private State states[] = {
    new Wash(), new Spin(),
    new Rinse(), new Spin(),
  };
  public Washer() { runAll(); }
  public boolean changeState() {
    if(i < states.length) {
      // Change the state by setting the
      // surrogate reference to a new object:
      currentState = states[i++];
      return true;
    } else
      return false;
  }
}

public class StateMachineDemo extends UnitTest {
  Washer w = new Washer();
  public void test() {
    // The constructor does the work.
    // This just makes sure it will complete
    // without throwing an exception.
  }
  public static void main(String args[]) {
    new StateMachineDemo().test();
  }
} ///:~
```

Here, the class that controls the states (**StateMachine** in this case) is responsible for deciding the next state to move to. Another approach is to allow the state objects themselves to decide what state to move to next, typically based on some kind of input to the system. This is a more flexible solution. Here it is, and in addition the program is evolved to use 2-d arrays to configure the state machines:

```
//: c04:washer:Washer.java
// An example of the State Machine pattern
package c04.washer;
import java.util.*;
import java.io.*;
import com.bruceeckel.test.*;

class MapLoader {
  public static void load(Map m, Object[][] pairs) {
    for(int i = 0; i < pairs.length; i++)
      m.put(pairs[i][0], pairs[i][1]);
  }
}

interface State {
  void run(String input);
}

public class Washer {
  private State currentState;
  static HashMap states = new HashMap();
  public Washer() {
    states.put("Wash", new Wash());
    states.put("Rinse", new Rinse());
    states.put("Spin", new Spin());
    currentState = (State)states.get("Wash");
  }
  private void
  nextState(Map stateTable, String input) {
    currentState = (State)states.get(
      stateTable.get(input));
  }

  class TState implements State {
    protected HashMap stateTable = new HashMap();
    public void run(String input) {
      String name = getClass().toString();
      System.out.println(
        name.substring(
          name.lastIndexOf("$") + 1));
      nextState(stateTable, input);
    }
  }
  // A different subclass for each state:
  class Wash extends TState {
    {
      MapLoader.load(stateTable, new String[][] {
        { "Wash", "Spin" },
        { "Spin", "Spin" },
        { "Rinse", "Rinse" }
      });
    }
  }
  class Spin extends TState {
    {
      MapLoader.load(stateTable, new String[][] {
        { "Wash", "Wash" },
        { "Spin", "Rinse" },
        { "Rinse", "Rinse" }
      });
    }
  }
  class Rinse extends TState {
    {
      MapLoader.load(stateTable, new String[][] {
        { "Wash", "Wash" },
        { "Spin", "Spin" },
        { "Rinse", "Spin" }
      });
    }
  }
  public void run() {
    try {
      BufferedReader inputStream =
        new BufferedReader (
          new FileReader("StateFile.txt"));
      while (inputStream.ready()) {
        // Get next state from file...
        String input =
          inputStream.readLine().trim();
        if (input != null)
          currentState.run(input);
      }
      inputStream.close ();
    } catch (IOException e) {
      e.printStackTrace(System.err);
    }
  }
  public static class Test extends UnitTest {
    Washer w = new Washer();
    public void test() { w.run(); }
  }
  public static void main(String args[]) {
    new Test().test();
  }
} ///:~
```

The input is read from the file **StateFile.txt**:

```
//:! c04:washer:StateFile.txt
Wash
Spin
Rinse
Spin
Wash
Spin
Rinse
Spin
Wash
Spin
Rinse
Spin
Wash
Spin
Rinse
Spin
///:~
```

If you look at the above program, you can easily see that having the proliferation of tables is annoying and messy to maintain. If you are going to be regularly configuring and modifying the state transition information, the best solution is to combine all the state information into a single table. This can be implemented using a **Map** of **Map**s, but at this point we might as well create a reusable tool for the job:

```
//: com:bruceeckel:util:TransitionTable.java
// Tool to assist creating &
// using state transition tables
package com.bruceeckel.util;
import java.util.*;

public class TransitionTable {
  public static Map
  build(Object[][][] table, Map m) {
    for(int i = 0; i < table.length; i++) {
      Object[][] row = table[i];
      Object key = row[0][0];
      Map val = new HashMap();
      for(int j = 1; j < row.length; j++)
        val.put(row[j][0], row[j][1]);
      m.put(key, val);
    }
    return m;
  }
  public interface Transitioner {
    Object nextState(Object curr, Object input);
  }
  // Default implementation and example
  // of nextState() method code:
  public static class StateChanger
    implements Transitioner {
    private Map map = new HashMap();
    public StateChanger(Object[][][] table) {
      TransitionTable.build(table, map);
    }
    public Object
    nextState(Object curr, Object input) {
      return ((Map)map.get(curr)).get(input);
    }
  }
} ///:~
```

Here is the unit test code that creates and runs an example transition table. It also includes a **main( )** for convenience:

```java
//: c04:TransitionTableTest.java
// Unit test code for TransitionTable.java
import com.bruceeckel.test.*;
import com.bruceeckel.util.*;

public class TransitionTableTest extends UnitTest{
  Object[][][] transitionTable = {
    { {"one"}, // Current state
      // Pairs of input & new state:
      {"one", "two"},
      {"two", "two"},
      {"three", "two"}},
    { {"two"}, // Current state
      // Pairs of input & new state:
      {"one", "three"},
      {"two", "three"},
      {"three", "three"}},
    { {"three"}, // Current state
      // Pairs of input & new state:
      {"one", "one"},
      {"two", "one"},
      {"three", "one"}},
  };
  TransitionTable.StateChanger m =
    new TransitionTable.StateChanger(
      transitionTable);
  public void test() {
    System.out.println(m);
    String current = "one";
    String[] inputs = { "one", "two", "three" };
    for(int i = 0; i < 20; i++) {
      String input = inputs[
        (int)(Math.random() * inputs.length)];
      System.out.print("input = " + input);
      current =
        (String)m.nextState(current, input);
      System.out.println(
        ", new state = " + current);
    }
  }
  public static void main(String[] args) {
    new TransitionTableTest().test();
  }
} ///:~
```

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.

Exercises

1. Create an example of the "virtual proxy."
2. Create an example of the "Smart reference" proxy where you keep count of the number of method calls to a particular object.
3. 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**.
4. Create a simple copy-on write implementation.
5. Apply **TransitionTable.java** to the "Washer" problem.
6. 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.

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

X: Iterators: decoupling algorithms from containers

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

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

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

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

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

Type-safe iterators

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

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

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

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

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

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

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.

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

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

Simple Factory method

As an example, let’s revisit the **Shape** system. Since the factory may fail in its creation of a requested **Shape**, an appropriate exception is needed:

```
//: c05:badshape:BadShapeCreation.java
package c05.badshape;
public class BadShapeCreation extends Exception {
  public BadShapeCreation(String msg) {
    super(msg);
  }
}///:~
```

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

```
//: c05:shapefact1:ShapeFactory1.java
// A simple static factory method.
package c05.shapefact1;
import c05.badshape.*;
import java.util.*;
import com.bruceeckel.test.*;

abstract class Shape {
  public abstract void draw();
  public abstract void erase();
  public static Shape factory(String type)
    throws BadShapeCreation {
    if(type.equals("Circle")) return new Circle();
    if(type.equals("Square")) return new Square();
    throw new BadShapeCreation(type);
  }
}

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

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

public class ShapeFactory1 extends UnitTest {
  String shlist[] = { "Circle", "Square",
    "Square", "Circle", "Circle", "Square" };
  List shapes = new ArrayList();
  public void test() {
    try {
      for(int i = 0; i < shlist.length; i++)
        shapes.add(Shape.factory(shlist[i]));
    } catch(BadShapeCreation e) {
      e.printStackTrace(System.err);
      assert(false); // Fail the unit test
    }
    Iterator i = shapes.iterator();
    while(i.hasNext()) {
      Shape s = (Shape)i.next();
      s.draw();
      s.erase();
    }
  }
  public static void main(String args[]) {
    new ShapeFactory1().test();
  }
} ///:~
```

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

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

Polymorphic factories

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

```
//: c05:shapefact2:ShapeFactory2.java
// Polymorphic factory methods.
package c05.shapefact2;
import c05.badshape.*;
import java.util.*;
import com.bruceeckel.test.*;

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

abstract class ShapeFactory {
  protected abstract Shape create();
  private static Map factories = new HashMap();
  public static void
  addFactory(String id, ShapeFactory f) {
    factories.put(id, f);
  }
  // A Template Method:
  public static final Shape createShape(String id)
  throws BadShapeCreation {
    if(!factories.containsKey(id)) {
      try {
        // Load dynamically
        Class.forName("c05.shapefact2." + id);
      } catch(ClassNotFoundException e) {
        throw new BadShapeCreation(id);
      }
      // See if it was put in:
      if(!factories.containsKey(id))
        throw new BadShapeCreation(id);
    }
    return
      ((ShapeFactory)factories.get(id)).create();
  }
}

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

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

public class ShapeFactory2 extends UnitTest {
  String shlist[] = { "Circle", "Square",
    "Square", "Circle", "Circle", "Square" };
  List shapes = new ArrayList();
  public void test() {
    // This just makes sure it will complete
    // without throwing an exception.
    try {
      for(int i = 0; i < shlist.length; i++)
        shapes.add(
          ShapeFactory.createShape(shlist[i]));
    } catch(BadShapeCreation e) {
      e.printStackTrace(System.err);
      assert(false); // Fail the unit test
    }
    Iterator i = shapes.iterator();
    while(i.hasNext()) {
      Shape s = (Shape)i.next();
      s.draw();
      s.erase();
    }
  }
  public static void main(String args[]) {
    new ShapeFactory2().test();
  }
} ///:~
```

Now the factory method appears in its own class, **ShapeFactory**, as the **create( )** method. This is a **protected** method which means it cannot be called directly, but it can be overridden. The subclasses of **Shape** must each create their own subclasses of **ShapeFactory** and override the

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

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

Abstract factories

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

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

```
//: c05:Games.java
// An example of the Abstract Factory pattern.
```

```
import com.bruceeckel.test.*;

interface Obstacle {
  void action();
}

interface Player {
  void interactWith(Obstacle o);
}

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

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

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

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

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

// Concrete factories:
class KittiesAndPuzzles
```

```
implements GameElementFactory {
  public Player makePlayer() {
    return new Kitty();
  }
  public Obstacle makeObstacle() {
    return new Puzzle();
  }
}

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

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

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

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

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

Exercises

1. Add a class Triangle to ShapeFactory1.java
2. Add a class **Triangle** to **ShapeFactory2.java**
3. Add a new type of **GameEnvironment** called **GnomesAndFairies** to **GameEnvironment.java**
4. Modify **ShapeFactory2.java** so that it uses an *Abstract Factory* to create different sets of shapes (for example, one particular type of factory object creates "thick shapes," another creates "thin shapes," but each factory object can create all the shapes: circles, squares, triangles etc.).

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. The point is to decouple the choice of function to be called from the site where that function is called.

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

Command

This is the functor in its purest sense: a method that's an object[1]. 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.

```
//: c06:CommandPattern.java
import java.util.*;
import com.bruceeckel.test.*;

interface Command {
  void execute();
}

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

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

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

```
  }
}

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

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

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

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

Another example of *Command* is **c12:DirList.java**. The **DirFilter** class is the command object which contains its action in the method **accept( )** that is passed to the **list( )** method. The **list( )** method determines what to include in its result by calling **accept( )**.

Design Patterns says that “Commands are an object-oriented replacement for callbacks[2].” However, I think that the word “back” is an essential part of the concept of callbacks. That is, I think a callback actually reaches back to the creator of the callback. On the other hand, with a *Command* object you typically just create it and hand it to some method or object, and are not otherwise connected over time to the *Command* object. That’s my take on it, anyway. Later in this book, I combine a group of design patterns under the heading of “callbacks.”

Strategy

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 functors. The difference is in the way this hierarchy is used. As seen in **c12:DirList.java**, you use *Command* to solve a particular problem—in that case, selecting files from a list. The “thing that stays the same” is the body of the method that’s being called, and the part that varies is isolated in the functor. I would hazard to say that *Command* provides flexibility while you’re writing the program, whereas *Strategy*’s flexibility is at run time. Nonetheless, it seems a rather fragile distinction.

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:

```
//: c06:strategy:StrategyPattern.java
package c06.strategy;
import com.bruceeckel.util.*; // Arrays2.print()
import com.bruceeckel.test.*;

// The strategy interface:
interface FindMinima {
  // Line is a sequence of points:
```

[2] Page 235.

```
  double[] algorithm(double[] line);
}

// The various strategies:
class LeastSquares implements FindMinima {
  public double[] algorithm(double[] line) {
    return new double[] { 1.1, 2.2 }; // Dummy
  }
}

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

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

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

public class StrategyPattern extends UnitTest {
  MinimaSolver solver =
    new MinimaSolver(new LeastSquares());
  double[] line = {
    1.0, 2.0, 1.0, 2.0, -1.0,
    3.0, 4.0, 5.0, 4.0 };
  public void test() {
    Arrays2.print(solver.minima(line));
    solver.changeAlgorithm(new Bisection());
    Arrays2.print(solver.minima(line));
  }
  public static void main(String args[]) {
    new StrategyPattern().test();
  }
} ///:~
```

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 the same base-class method (with different implementations) which calls another implementation of the base-class method, etc., until a termination condition is reached.

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

In **StrategyPattern.java**, above, what you probably want is to automatically find a solution. *Chain of Responsibility* provides a way to do this:

```
//: c06:ChainOfResponsibility.java
import com.bruceeckel.util.*; // Arrays2.print()
import com.bruceeckel.test.*;
import java.util.*;

class FindMinima {
  private FindMinima successor = null;
  public void add(FindMinima succ) {
    FindMinima end = this;
    while(end.successor != null)
```

```
      end = end.successor; // Traverse list
    end.successor = succ;
  }
  public double[] nextAlgorithm(double[] line) {
    if(successor != null)
      // Try the next one in the chain:
      return successor.algorithm(line);
    else
      return new double[] {}; // Nothing found
  }
  public double[] algorithm(double[] line) {
    // FindMinima algorithm() is only the
    // start of the chain; doesn't actually try
    // to solve the problem:
    return nextAlgorithm(line);
  }
}

class LeastSquares extends FindMinima {
  public double[] algorithm(double[] line) {
    System.out.println("LeastSquares.algorithm");
    boolean weSucceed = false;
    if(weSucceed) // Actual test/calculation here
      return new double[] { 1.1, 2.2 }; // Dummy
    else // Try the next one in the chain:
      return nextAlgorithm(line);
  }
}

class Perturbation extends FindMinima {
  public double[] algorithm(double[] line) {
    System.out.println("Perturbation.algorithm");
    boolean weSucceed = false;
    if(weSucceed) // Actual test/calculation here
      return new double[] { 3.3, 4.4 }; // Dummy
    else // Try the next one in the chain:
      return nextAlgorithm(line);
  }
}

class Bisection extends FindMinima {
  public double[] algorithm(double[] line) {
    System.out.println("Bisection.algorithm");
    boolean weSucceed = true;
    if(weSucceed) // Actual test/calculation here
      return new double[] { 5.5, 6.6 }; // Dummy
    else
      return nextAlgorithm(line);
  }
}

// The "Handler" proxies to the first functor:
class MinimaSolver {
  private FindMinima chain = new FindMinima();
  void add(FindMinima newAlgorithm) {
    chain.add(newAlgorithm);
  }
  // Make the call to the top of the chain:
  double[] minima(double[] line) {
    return chain.algorithm(line);
  }
}

public
class ChainOfResponsibility extends UnitTest {
  MinimaSolver solver = new MinimaSolver();
  double[] line = {
    1.0, 2.0, 1.0, 2.0, -1.0,
    3.0, 4.0, 5.0, 4.0 };
  public void test() {
    solver.add(new LeastSquares());
    solver.add(new Perturbation());
    solver.add(new Bisection());
    Arrays2.print(solver.minima(line));
  }
  public static void main(String args[]) {
    new ChainOfResponsibility().test();
  }
} ///:~
```

Exercises

1. Modify **ChainOfResponsibility.java** so that it uses an **ArrayList** to hold the different strategy objects. Use **Iterator**s to keep track

of the current item and to move to the next one. Does this implement the *Chain of Responsibility* according to GoF?

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?

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.

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.

```
//: c07:Adapter.java
// Variations on the Adapter pattern.
import com.bruceeckel.test.*;

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

interface WhatIWant {
  void f();
}

class ProxyAdapter implements WhatIWant {
  WhatIHave whatIHave;
  public ProxyAdapter(WhatIHave wih) {
    whatIHave = wih;
  }
  public void f() {
    // Implement behavior using
    // methods in WhatIHave:
```

```
    whatIHave.g();
    whatIHave.h();
  }
}

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

// Approach 2: build adapter use into op():
class WhatIUse2 extends WhatIUse {
  public void op(WhatIHave wih) {
    new ProxyAdapter(wih).f();
  }
}

// Approach 3: build adapter into WhatIHave:
class WhatIHave2 extends WhatIHave
implements WhatIWant {
  public void f() {
    g();
    h();
  }
}

// Approach 4: use an inner class:
class WhatIHave3 extends WhatIHave {
  private class InnerAdapter implements WhatIWant{
    public void f() {
      g();
      h();
    }
  }
  public WhatIWant whatIWant() {
    return new InnerAdapter();
  }
}

public class Adapter extends UnitTest {
  WhatIUse whatIUse = new WhatIUse();
  WhatIHave whatIHave = new WhatIHave();
  WhatIWant adapt= new ProxyAdapter(whatIHave);
  WhatIUse2 whatIUse2 = new WhatIUse2();
  WhatIHave2 whatIHave2 = new WhatIHave2();
  WhatIHave3 whatIHave3 = new WhatIHave3();
  public void test() {
    whatIUse.op(adapt);
    // Approach 2:
    whatIUse2.op(whatIHave);
    // Approach 3:
    whatIUse.op(whatIHave2);
    // Approach 4:
    whatIUse.op(whatIHave3.whatIWant());
  }
  public static void main(String args[]) {
    new Adapter().test();
  }
} ///:~
```

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.

Façade

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

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

```
//: c07:Facade.java
import com.bruceeckel.test.*;

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

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

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

The example given in *Design Patterns* isn't really a *Façade* but just a class that uses the other classes.

Package as a variation of *Façade*

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

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

To be fair, *Design Patterns* is written primarily for a C++ audience. Although C++ has namespaces to prevent clashes of globals and class names, this does not provide the class hiding mechanism that you get with non-**public** classes in Java. The majority of the time I think that Java packages will solve the *Façade* problem.

Exercises

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

2.

8: Table-driven code: configuration flexibility

Table-driven code using anonymous inner classes

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

Also **GreenHouse.java**

9: Interpreter/ Multiple Languages

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

The most straightforward use of this idea is the *Interpreter* design pattern, which adds an interpreted language to your program to allow the end user to easily customize a solution. In Java, the easiest and most powerful way to do this is with *Jython*[1], an implementation of the Python language in pure Java byte codes.

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

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

Interpreter

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

Motivation

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

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

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

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

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

To install Python, go to *www.Python.org* and follow the links and instructions. To install Jython, go to *http://sourceforge.net/projects/jython*. The download is a **.class** file, which will run an installer when you execute it with Java.

Python overview

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

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

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

```
#: c09:if.py
response = "yes"
if response == "yes":
  print "affirmative"
  val = 1
print "continuing..."
#:~
```

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

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

Python normally has only one statement per line (you can put more by separating them with semicolons) and so no terminating semicolon is necessary. Even from this brief example you can see that the language is designed to be as simple as possible, and yet still very readable.

Built-in containers

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

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

```
#: c09: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 (remember that I had to create a special **Arrays2** class 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 as it prints it.

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

Notice that there aren't any type declarations – the object names simply appear, and Python infers their type. Again, it's almost as if it's created 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 the programming language but didn't actually describe what your program was supposed to do.

Functions

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

```
#: c09: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:

```
#: c09: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:

```
#: c09:sum.py
def sum(arg1, arg2):
  return arg1 + arg2

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

When the operator '**+**' is used with strings, it means concatenation (yes, Python supports operator overloading, and it does a nice job of it).

Strings

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

```
#: c09:strings.py
print "That isn't a horse"
print 'You are not a "Viking"'
print """You're just pounding two
coconut halves together."""
```

```
print '''"Oh no!" He exclaimed.
"It's the blemange!"'''
print r'c:\python\lib\utils'
#:~
```

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

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

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

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

```
#: c09:stringFormatting.py
val = 47
print "The number is %d" % val
val2 = 63.4
s = "val: %d, val2: %f" % (val, val2)
print s
#:~
```

As you can see in the second case, if you have more than one argument you surround them in parentheses (this forms a *tuple*, which is a list that cannot be modified).

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

Classes

Like everything else in Python, the definition of a class uses a minimum of additional syntax. You use the **class** keyword, and inside the body you use **def** to create methods. Here's a simple class (The '**#**' denotes a comment that goes until the end of the line, just like C++ and Java '**//**' comments):

```
#: c09:SimpleClass.py
class Simple:
  def __init__(self, str):
    print "Inside the Simple constructor"
    self.s = str
  # Two methods:
  def show(self):
    print self.s
  def showMsg(self, msg):
    print msg + ':',
    self.show() # Calling another method

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

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

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

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

```
Python SimpleClass.py
```

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

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

Inheritance

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

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

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

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

You inherit a class (or classes – Python supports multiple inheritance) by listing the name(s) of the class inside parentheses after the class name. 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:

```
#: c09: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. Note that in **display( )**, **showMsg( )** can be called as a method of **self**, but when calling the base-class version of the method you are overriding, you must fully qualify the name and pass **self** in as the first argument, as shown in the base-class constructor call. This can also be seen in the overridden version of **show( )**.

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

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

Creating a language

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

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

```
#:c09:GreenHouseLanguage.py

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

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

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

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

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

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

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

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

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

# To test, this will be run when you say:
# python GreenHouseLanguage.py
if __name__ == "__main__":
  ThermostatNight(5.00)
  LightOff(2.00)
  WaterOn(3.30)
  WaterOff(4.45)
  LightOn(1.00)
  ThermostatDay(6.00)
  Bell(7.00)
  run()
#:~
```

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

The **__main__** section tests the classes to make sure they work right.

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

```
//: c09:GreenHouseController.java
import org.python.util.PythonInterpreter;
import org.python.core.*;
import com.bruceeckel.test.*;

public class
GreenHouseController extends UnitTest {
  PythonInterpreter interp =
    new PythonInterpreter();
  public void test() throws PyException  {
    System.out.println(
      "Loading GreenHouse Language");
    interp.execfile("GreenHouseLanguage.py");
    System.out.println(
      "Loading GreenHouse Script");
    interp.execfile("Schedule.ghs");
    System.out.println(
      "Executing GreenHouse Script");
    interp.exec("run()");
  }
  public static void
  main(String[] args) throws PyException  {
    new GreenHouseController().test();
  }
} ///:~
```

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

```
//:! c09:Schedule.ghs
Bell(7.00)
ThermostatDay(6.00)
WaterOn(3.30)
LightOn(1.00)
ThermostatNight(5.00)
LightOff(2.00)
WaterOff(4.45)
///:~
```

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

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

Configuring Jython

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

You may need to edit **jython.bat**, as it can incorrectly assume a path for your **java.exe** program.

Many improvements have been made in the installation process for Jython over the old JPython, but it's still important that **jython.jar** is in your CLASSPATH.

Generating documentation

Jython 2.0, for some reason, is distributed with only minimal API documentation. In fact, only **PythonInterpreter** has the Java documentation created for it. The Java documentation strings are there for the rest of the classes, but they weren't extracted. Although many of the classes are not necessary in order to program with Jython, many are and so it's valuable to run Javadoc in order to generate the HTML documentation.

Here is the makefile that I used to create the Java documentation:

```
all:
  javadoc -sourcepath C:\Progtools\Jython\ \
  -d C:\ProgTools\Jython\docs\new \
  org.python.core org.python.modules \
  org.python.util org.python.rmi
```

You'll have to adjust the paths to fit your own installation. The **sourcepath** is where you installed Jython, and **–d** indicates the destination directory for the generated HTML files. Look up the JDK online documentation for Javadoc for further details.

Once you generate the documentation, you can poke through and pick up a few bits and pieces you wouldn't otherwise find.

Controlling the interpreter

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

Putting data in

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

```
//: c09:PythonInterpreterSetting.java
// Passing data from Java to python when using
// the PythonInterpreter object.
import org.python.util.PythonInterpreter;
import org.python.core.*;
import java.util.*;
import com.bruceeckel.python.*;
import com.bruceeckel.test.*;

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

```
interp.set("c", new PyInteger(1));
interp.set("d", new PyFloat(2.2));
interp.exec("print c + d");
// You can also use Java's object wrappers:
interp.set("c", new Integer(9));
interp.set("d", new Float(3.14));
interp.exec("print c + d");
// Define a Python function to print arrays:
interp.exec(
  "def prt(x): \n" +
  "  print x \n" +
  "  for i in x: \n" +
  "    print i, \n" +
  "  print x.__class__\n");
// Arrays are Objects, so it has no trouble
// figuring out the types contained in arrays:
Object[] types = {
  new boolean[]{ true, false, false, true },
  new char[]{ 'a', 'b', 'c', 'd' },
  new byte[]{ 1, 2, 3, 4 },
  new int[]{ 10, 20, 30, 40 },
  new long[]{ 100, 200, 300, 400 },
  new float[]{ 1.1f, 2.2f, 3.3f, 4.4f },
  new double[]{ 1.1, 2.2, 3.3, 4.4 },
};
for(int i = 0; i < types.length; i++) {
  interp.set("e", types[i]);
  interp.exec("prt(e)");
}
// It uses toString() to print Java objects:
interp.set("f", new Date());
interp.exec("print f");
// You can pass it a List and
// index into it...
ArrayList x = new ArrayList();
for(int i = 0; i < 10; i++)
    x.add(new Integer(i * 10));
interp.set("g", x);
interp.exec("print g");
interp.exec("print g[1]");
// ... But it's not quite smart enough
// to treat it as a Python array:
interp.exec("print g.__class__");
// interp.exec("print g[5:]); // Fails
```

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

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

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

```
print a[5:]
```

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

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

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

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

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

public class PyUtil {
  /** Extract a Python tuple or array into a Java
  List (which can be converted into other kinds
  of lists and sets inside Java).
  @param interp The Python interpreter object
  @param pyName The id of the python list object
  */
  public static List
  toList(PythonInterpreter interp, String pyName){
    return new ArrayList(Arrays.asList(
      (Object[])interp.get(
        pyName, Object[].class)));
  }
  /** Extract a Python dictionary into a Java Map
  @param interp The Python interpreter object
  @param pyName The id of the python dictionary
  */
  public static Map
  toMap(PythonInterpreter interp, String pyName){
    PyList pa = ((PyDictionary)interp.get(
      pyName)).items();
    Map map = new HashMap();
    while(pa.__len__() != 0) {
      PyTuple po = (PyTuple)pa.pop();
      Object first = po.__finditem__(0)
        .__tojava__(Object.class);
      Object second = po.__finditem__(1)
        .__tojava__(Object.class);
      map.put(first, second);
    }
    return map;
  }
  /** Turn a Java Map into a PyDictionary,
  suitable for placing into a PythonInterpreter
  @param map The Java Map object
  */
  public static PyDictionary
  toPyDictionary(Map map) {
    Map m = new HashMap();
    Iterator it = map.entrySet().iterator();
    while(it.hasNext()) {
      Map.Entry e = (Map.Entry)it.next();
      m.put(Py.java2py(e.getKey()),
        Py.java2py(e.getValue()));
    }
    // PyDictionary constructor wants a Hashtable:
    return new PyDictionary(new Hashtable(m));
  }
} ///:~
```

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

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

public class Test extends UnitTest {
  PythonInterpreter pi =
    new PythonInterpreter();
  public void test1() {
    pi.exec("tup=('fee','fi','fo','fum','fi')");
    List lst = PyUtil.toList(pi, "tup");
    System.out.println(lst);
    System.out.println(new HashSet(lst));
  }
  public void test2() {
    pi.exec("ints=[1,3,5,7,9,11,13,17,19]");
    List lst = PyUtil.toList(pi, "ints");
    System.out.println(lst);
  }
  public void test3() {
    pi.exec("dict = { 1 : 'a', 3 : 'b', " +
      "5 : 'c', 9 : 'd', 11 : 'e'}");
    Map mp = PyUtil.toMap(pi, "dict");
    System.out.println(mp);
  }
  public void test4() {
    Map m = new HashMap();
    m.put("twas", new Integer(11));
    m.put("brillig", new Integer(27));
    m.put("and", new Integer(47));
    m.put("the", new Integer(42));
    m.put("slithy", new Integer(33));
    m.put("toves", new Integer(55));
    System.out.println(m);
    pi.set("m", PyUtil.toPyDictionary(m));
    pi.exec("print m");
    pi.exec("print m['slithy']");
  }
  public static void main(String args[]) {
    Test t = new Test();
    t.test1();
    t.test2();
    t.test3();
    t.test4();
  }
} ///:~
```

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

Getting data out

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

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

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

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

```
//: c09:PythonInterpreterGetting.java
// Getting data from the PythonInterpreter object.
import org.python.util.PythonInterpreter;
import org.python.core.*;
import java.util.*;
import com.bruceeckel.python.*;
import com.bruceeckel.test.*;

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

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

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

    // As an example of useful string tools,
    // global expansion of ambiguous file names
    // using glob is very useful, but it's not
    // part of the standard Jython package, so
    // you'll have to make sure that your
    // Python path is set to include these, or
    // that you deliver the necessary Python
    // files with your application.
    interp.exec("from glob import glob");
    interp.exec("files = glob('*.java')");
    String[] files = (String[])
      interp.get("files", String[].class);
    for(int i = 0; i < files.length; i++)
      System.out.println(files[i]);

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

    // If you have a Python dictionary, it can
    // be extracted into a Java Map, again with
```

```
    // com.bruceeckel.PyUtil:
    interp.exec("dict = { 1 : 'a', 3 : 'b'," +
      "5 : 'c', 9 : 'd', 11 : 'e' }");
    Map map = PyUtil.toMap(interp, "dict");
    System.out.println("map: " + map);
    // It really is Java objects, not PyObjects:
    Iterator it = map.entrySet().iterator();
    Map.Entry e = (Map.Entry)it.next();
    System.out.println(e.getKey().getClass());
    System.out.println(e.getValue().getClass());
  }
  public static void
  main(String[] args) throws PyException  {
    new PythonInterpreterGetting().test();
  }
} ///:~
```

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

Multiple interpreters

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

```
//: c09:MultipleJythons.java
// You can run multiple interpreters, each
// with its own name space.
import org.python.util.PythonInterpreter;
import org.python.core.*;
import com.bruceeckel.test.*;

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

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

Controlling Java from Jython

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

```
#: c09:JavaClassInPython.py
#=M jython JavaClassInPython.py
# Using Java classes within Jython
from java.util import Date, HashSet, HashMap
from c09.javaclass import JavaClass
from math import sin

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

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

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

for x, y in ValGen(10):
  map.put(x, y)
  set.add(y)
  set.add(y)

print map
print set

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

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

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

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

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

ValGen implements the concept of a "generator" which is used a great deal in the C++ STL (*Standard Template Library*, part of the Standard C++ Library). A generator is an object that produces a new object every time its "generation method" is called, and it is quite convenient for filling containers. Here, I wanted to use it in a **for** iteration, and so I needed the generation method to be the one that is called by the iteration process. This is a special method called **__getitem__( )**, which is actually the overloaded operator for indexing, '**[ ]**'. A **for** loop calls this method every time it wants to move the iteration forward, and when the elements run out, **__getitem__( )** throws an out-of-bounds exception and that signals the end of the **for** loop (in other languages, you would never use an exception for ordinary control flow, but in Python it seems to work quite well). This exception happens automatically when **self.val[i]** runs out of elements, so the **__getitem__( )** code turns out to be simple. The only complexity is that **__getitem__( )** appears to return *two* objects instead of just one. What Python does is automatically package multiple return values into a tuple, so you still only end up returning a single object (in C++ or Java you would have to create your own data structure to accomplish this). In addition, in the **for** loop where **ValGen** is used, Python automatically "unpacks" the tuple so that you can have multiple iterators in the **for**. These are the kinds of syntax simplifications that make Python so endearing.

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

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

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

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

```
//: c09:javaclass:JavaClass.java
package c09.javaclass;
import com.bruceeckel.test.*;
import com.bruceeckel.util.*;

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

You can see that this is just an ordinary Java class, without any awareness that it will be used in a Jython program. For this reason, one of the important uses of Jython is in testing Java code[2]. Because Python is such a powerful, flexible, dynamic language it is an ideal tool for automated test frameworks, without making any changes to the Java code that's being tested.

Using Java libraries

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

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

```
#: c09:PythonSwing.py
# The HTMLButton.java example from
# "Thinking in Java, 2nd edition," Chapter 13,
# converted into Jython.
# Don't run this as part of the automatic make:
#=M @echo skipping PythonSwing.py
from javax.swing import JFrame, JButton, JLabel
```

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

```
from java.awt import FlowLayout

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

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

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

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

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

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

Inheriting from Java library classes

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

```
#: c09:PythonDialogs.py
# Dialogs.java from "Thinking in Java, 2nd
# edition," Chapter 13, converted into Jython.
# Don't run this as part of the automatic make:
#=M @echo skipping PythonDialogs.py
from java.awt import FlowLayout
from javax.swing import JFrame, JDialog, JLabel
from javax.swing import JButton

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

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

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

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

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

Creating Java classes with Jython

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

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

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

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

The Python definition doesn't specify any return type, but the @sig string gives the full type information about what is being returned. The **jythonc** compiler uses this information to generate the correct Java code.

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

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

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

```
#: c09:PythonToJavaClass.py
#=T python\java\test\PythonToJavaClass.class
#=M jythonc --package python.java.test \
#=M PythonToJavaClass.py
# A Python class created to produce a Java class
from jarray import array
import java

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

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

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

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

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

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

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

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

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

def argIn4(self, a):
  "@sig public void \
     argIn4(org.python.core.PyArray a)"
  print "received type:", a.__class__
```

```
    print "a: ", a
    print "element type:", a[0].__class__
    print "a[3] + a[5]:", a[5] + a[7]
    print "a[2:5]:", a[2:5] # A real Python array

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

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

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

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

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

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

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

```
//: c09:TestPythonToJavaClass.java
//+D python\java\test\PythonToJavaClass.class
import java.lang.reflect.*;
import java.util.*;
import org.python.core.*;
import com.bruceeckel.test.*;
import com.bruceeckel.util.*;
import com.bruceeckel.python.*;
// The package with the Python-generated classes:
import python.java.test.*;

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

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

Building the Java classes from the Python code

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

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

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

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

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

Summary

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

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

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

To me, Python and Java present a very potent combination for program development because of Java's architecture and tool set, and Python's extremely rapid development (generally considered to be 5-10 times faster than C++ or Java). Python is usually slower, however, but even if you end up re-coding parts of your program for speed, the initial fast development will allow you to more quickly flesh out the system and uncover and solve the critical sections. And often, the execution speed of Python is not a problem – in those cases it's an even bigger win. A number of commercial products already use Java and Jython, and because of the terrific productivity leverage I expect to see this happen more in the future.

Exercises

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

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

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

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

10: Callbacks

Decoupling code behavior

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

Observer

Like the other forms of callback, this contains a hook point where you can change code. The difference is in the observer’s completely dynamic nature. It is often used for the specific case of changes based on other object’s change of state, but is also the basis of event management. Anytime you want to decouple the source of the call from the called code in a completely dynamic way.

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

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

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

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

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

Observable has a flag to indicate whether it’s been changed. In a simpler design, there would be no flag; if something happened, everyone would be notified. The flag allows you to wait, and only notify the **Observer**s when you decide the time is right. Notice, however, that the control of the flag’s state is **protected**, so that only an inheritor can decide what constitutes a change, and not the end user of the resulting derived **Observer** class.

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

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

Observing flowers

Here is an example of the observer pattern:

```
//: c10:ObservedFlower.java
// Demonstration of "observer" pattern.
import java.util.*;
import com.bruceeckel.test.*;

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

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

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

public class ObservedFlower extends UnitTest {
  Flower f = new Flower();
  Bee
    ba = new Bee("A"),
    bb = new Bee("B");
  Hummingbird
    ha = new Hummingbird("A"),
    hb = new Hummingbird("B");
  public void test() {
    f.opening().addObserver(ha.openObserver());
    f.opening().addObserver(hb.openObserver());
    f.opening().addObserver(ba.openObserver());
    f.opening().addObserver(bb.openObserver());
    f.closing().addObserver(ha.closeObserver());
    f.closing().addObserver(hb.closeObserver());
    f.closing().addObserver(ba.closeObserver());
    f.closing().addObserver(bb.closeObserver());
    // Hummingbird B decides to sleep in:
    f.opening().deleteObserver(
      hb.openObserver());
    // A change that interests observers:
```

```
    f.open();
    f.open(); // It's already open, no change.
    // Bee A doesn't want to go to bed:
    f.closing().deleteObserver(
      ba.closeObserver());
    f.close();
    f.close(); // It's already closed; no change
    f.opening().deleteObservers();
    f.open();
    f.close();
  }
  public static void main(String args[]) {
    new ObservedFlower().test();
  }
} ///:~
```

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

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

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

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

A visual example of observers

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

```
//: c10:BoxObserver.java
// Demonstration of Observer pattern using
// Java's built-in observer classes.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import com.bruceeckel.swing.*;

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

public class BoxObserver extends JFrame {
  Observable notifier = new BoxObservable();
  public BoxObserver(int grid) {
    setTitle("Demonstrates Observer pattern");
    Container cp = getContentPane();
    cp.setLayout(new GridLayout(grid, grid));
    for(int x = 0; x < grid; x++)
      for(int y = 0; y < grid; y++)
        cp.add(new OCBox(x, y, notifier));
  }
  public static void main(String[] args) {
    int grid = 8;
    if(args.length > 0)
      grid = Integer.parseInt(args[0]);
    JFrame f = new BoxObserver(grid);
    f.setSize(500, 400);
    f.setVisible(true);
    // JDK 1.3:
    f.setDefaultCloseOperation(EXIT_ON_CLOSE);
    // Add a WindowAdapter if you have JDK 1.2
  }
}

class OCBox extends JPanel implements Observer {
  Observable notifier;
  int x, y; // Locations in grid
  Color cColor = newColor();
  static final Color[] colors = {
    Color.black, Color.blue, Color.cyan,
    Color.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)
    ];
  }
  OCBox(int x, int y, Observable notifier) {
    this.x = x;
    this.y = y;
    notifier.addObserver(this);
    this.notifier = notifier;
    addMouseListener(new ML());
  }
  public void paintComponent(Graphics g) {
    super.paintComponent(g);
    g.setColor(cColor);
    Dimension s = getSize();
    g.fillRect(0, 0, s.width, s.height);
  }
  class ML extends MouseAdapter {
    public void mousePressed(MouseEvent e) {
      notifier.notifyObservers(OCBox.this);
    }
  }
  public void update(Observable o, Object arg) {
    OCBox clicked = (OCBox)arg;
    if(nextTo(clicked)) {
      cColor = clicked.cColor;
      repaint();
    }
  }
  private final boolean nextTo(OCBox b) {
    return Math.abs(x - b.x) <= 1 &&
           Math.abs(y - b.y) <= 1;
  }
} ///:~
```

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

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

It might appear that the way the observers are notified must be frozen at compile time in the **notifyObservers( )** method. However, if you look more closely at the code above you'll see that the only place in **BoxObserver** or **OCBox** where you're aware that you're working with a

BoxObservable is at the point of creation of the **Observable** object—from then on everything uses the basic **Observable** interface. This means that you could inherit other **Observable** classes and swap them at run time if you want to change notification behavior then.

Exercises

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

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

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?

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

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

Here’s an example of multiple dispatching:

```
//: c11:PaperScissorsRock.java
// Demonstration of multiple dispatching.
import java.util.*;
import com.bruceeckel.test.*;

// An enumeration type:
class Outcome {
  private int value;
  private String name;
  private Outcome(int val, String nm) {
    value = val;
    name = nm;
  }
  public final static Outcome
    WIN = new Outcome(0, "win"),
    LOSE = new Outcome(1, "lose"),
    DRAW = new Outcome(2, "draw");
  public String toString() { return name; }
  public boolean equals(Object o) {
    return (o instanceof Outcome)
      && (value == ((Outcome)o).value);
  }
}

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

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

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

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

class ItemGenerator {
  public static Item newItem() {
    switch((int)(Math.random() * 3)) {
      default:
      case 0:
        return new Scissors();
      case 1:
        return new Paper();
      case 2:
        return new Rock();
    }
  }
}

class Compete {
  public static Outcome match(Item a, Item b) {
    System.out.print(a + " <--> " + b + " : ");
    return a.compete(b);
  }
}

public class PaperScissorsRock extends UnitTest {
  List items = new ArrayList();
  public PaperScissorsRock() {
    for(int i = 0; i < 40; i++)
      items.add(ItemGenerator.newItem());
  }
  public void test() {
    for(int i = 0; i < items.size()/2; i++)
      System.out.println(
        Compete.match(
          (Item)items.get(i),
          (Item)items.get(i*2)));
  }
  public static void main(String args[]) {
    new PaperScissorsRock().test();
  }
} ///:~
```

Visitor, a type of multiple dispatching

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

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

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

```
//: c11:BeeAndFlowers.java
// Demonstration of "visitor" pattern.
import java.util.*;
import com.bruceeckel.test.*;

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

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

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

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

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

// Add the ability to produce a string:
class StringVal implements Visitor {
  String s;
  public String toString() { return s; }
  public void visit(Gladiolus g) {
    s = "Gladiolus";
  }
  public void visit(Renuculus r) {
    s = "Renuculus";
  }
  public void visit(Chrysanthemum c) {
    s = "Chrysanthemum";
  }
}

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

class FlowerGenerator {
  public static Flower newFlower() {
    switch((int)(Math.random() * 3)) {
      default:
      case 0: return new Gladiolus();
      case 1: return new Renuculus();
      case 2: return new Chrysanthemum();
    }
  }
}

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

Exercises

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

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

3. Modify the above example to replace the double dispatching with a table lookup instead, using a structure similar to **TransitionTable.java**. 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 simplicity of use of the dynamic dispatch but uses a table lookup?

12: Pattern refactoring

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

This process is similar to what Martin Fowler talks about in his book *Refactoring: Improving the Design of Existing Code*[1] (although he tends to talk about pieces of code more than pattern-level designs). You start with a solution, and then when you discover that it doesn't continue to meet your needs, you fix it. Of course, this is a natural tendency but in computer programming it's been extremely difficult to accomplish with procedural programs, and the acceptance of the idea that we *can* refactor code and designs adds to the body of proof that object-oriented programming is "a good thing."

Simulating the trash recycler

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

[1] 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.

```java
//: c12:recyclea:RecycleA.java
// Recycling with RTTI.
import java.util.*;
import java.io.*;
import com.bruceeckel.test.*;

abstract class Trash {
  private double weight;
  Trash(double wt) { weight = wt; }
  abstract double getValue();
  double getWeight() { return weight; }
  // Sums the value of Trash in a bin:
  static void sumValue(Collection bin) {
    Iterator e = bin.iterator();
    double val = 0.0f;
    while(e.hasNext()) {
      // One kind of RTTI:
      // A dynamically-checked cast
      Trash t = (Trash)e.next();
      // Polymorphism in action:
      val += t.getWeight() * t.getValue();
      System.out.println(
        "weight of " +
        // Using RTTI to get type
        // information about the class:
        t.getClass().getName() +
        " = " + t.getWeight());
    }
    System.out.println("Total value = " + val);
  }
}

class Aluminum extends Trash {
  static double val  = 1.67f;
  Aluminum(double wt) { super(wt); }
  double getValue() { return val; }
  static void setValue(double newval) {
    val = newval;
  }
}

class Paper extends Trash {
  static double val = 0.10f;
  Paper(double wt) { super(wt); }
  double getValue() { return val; }
  static void setValue(double newval) {
    val = newval;
  }
}

class Glass extends Trash {
  static double val = 0.23f;
  Glass(double wt) { super(wt); }
  double getValue() { return val; }
  static void setValue(double newval) {
    val = newval;
  }
}

public class RecycleA extends UnitTest {
  Collection
    bin = new ArrayList(),
    glassBin = new ArrayList(),
    paperBin = new ArrayList(),
    alBin = new ArrayList();
  public RecycleA() {
    // 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));
    }
  }
  public void test() {
    Iterator sorter = bin.iterator();
    // Sort the Trash:
    while(sorter.hasNext()) {
      Object t = sorter.next();
      // RTTI to show class membership:
      if(t instanceof Aluminum)
        alBin.add(t);
      if(t instanceof Paper)
        paperBin.add(t);
      if(t instanceof Glass)
        glassBin.add(t);
    }
    Trash.sumValue(alBin);
    Trash.sumValue(paperBin);
    Trash.sumValue(glassBin);
    Trash.sumValue(bin);
  }
  public static void main(String args[]) {
    new 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.

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

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

It looks silly to upcast the types of **Trash** into a container holding base type references, and then turn around and downcast. Why not just put the trash into the appropriate receptacle in the first place? (Indeed, this is the whole enigma of recycling). In this program it would be easy to repair, but sometimes a system's structure and flexibility can benefit greatly from downcasting.

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

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

Improving the design

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

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

"Make more objects"

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

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

```
    for(int i = 0; i < 30; i++)
      switch((int)(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.

The idea of the factory method is that you pass it the essential information it needs to know to create your object, then stand back and wait for the reference (already upcast to the base type) to pop out as the return value. From then on, you treat the object polymorphically. Thus, you never even need to know the exact type of object that's created. In fact, the factory method hides it from you to prevent accidental misuse. If you want to use the object without polymorphism, you must explicitly use RTTI and casting.

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

```
class Messenger {
  int type;
  // Must change this to add another type:
  static final int MAX_NUM = 4;
  double data;
  Messenger(int typeNum, double val) {
    type = typeNum % MAX_NUM;
    data = val;
  }
}
```

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

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

```
  static Trash factory(Messenger i) {
    switch(i.type) {
      default: // To quiet the compiler
      case 0:
        return new Aluminum(i.data);
      case 1:
        return new Paper(i.data);
      case 2:
        return new Glass(i.data);
      // Two lines here:
      case 3:
        return new Cardboard(i.data);
    }
  }
```

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

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

```
    for(int i = 0; i < 30; i++)
      bin.add(
        Trash.factory(
          new Messenger(
            (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.

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

A pattern for prototyping creation

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

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

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

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

The list of prototypes will be represented indirectly by a list of references to all the **Class** objects you want to create. In addition, if the prototyping fails, the **factory( )** method will assume that it's because a particular **Class** object wasn't in the list, and it will attempt to load it. By loading the prototypes dynamically like this, the **Trash** class doesn't need to know what types it is working with, so it doesn't need any modifications when you add new types. This allows it to be easily reused throughout the rest of the chapter.

```
//: c12:trash:Trash.java
// Base class for Trash recycling examples.
package c12.trash;
import java.util.*;
import java.lang.reflect.*;

public abstract class Trash {
  private double weight;
  public Trash(double wt) { weight = wt; }
  public Trash() {}
  public abstract double getValue();
  public double getWeight() { return weight; }
  // Sums the value of Trash given an
  // Iterator to any container of Trash:
  public static void sumValue(Iterator it) {
    double val = 0.0f;
    while(it.hasNext()) {
      // One kind of RTTI:
      // A dynamically-checked cast
      Trash t = (Trash)it.next();
      val += t.getWeight() * t.getValue();
      System.out.println(
        "weight of " +
        // Using RTTI to get type
        // information about the class:
        t.getClass().getName() +
        " = " + t.getWeight());
    }
    System.out.println("Total value = " + val);
  }
  // Remainder of class provides
  // support for prototyping:
  public static class PrototypeNotFoundException
    extends Exception {}
  public static class CannotCreateTrashException
    extends Exception {}
  private static List trashTypes =
    new ArrayList();
  public static Trash factory(Messenger info)
    throws PrototypeNotFoundException,
    CannotCreateTrashException {
    for(int i = 0; i < trashTypes.size(); i++) {
      // Somehow determine the new type
      // to create, and create one:
      Class tc = (Class)trashTypes.get(i);
      if (tc.getName().indexOf(info.id) != -1) {
        try {
          // Get the dynamic constructor method
          // that takes a double argument:
          Constructor ctor = tc.getConstructor(
              new Class[]{ double.class });
          // Call the constructor
          // to create a new object:
          return (Trash)ctor.newInstance(
            new Object[]{new Double(info.data)});
        } catch(Exception ex) {
          ex.printStackTrace(System.err);
          throw new CannotCreateTrashException();
        }
      }
    }
    // Class was not in the list. Try to load it,
    // but it must be in your class path!
    try {
      System.out.println("Loading " + info.id);
      trashTypes.add(Class.forName(info.id));
    } catch(Exception e) {
      e.printStackTrace(System.err);
      throw new PrototypeNotFoundException();
    }
    // 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, except that **SumValue( )** is now made more generic by taking an **Iterator** as an argument. The rest of the class supports the prototyping pattern. You first see two inner classes (which are made **static**, so they are inner classes only for code organization purposes) describing exceptions that can occur. This is followed by an **ArrayList** called **trashTypes**, which is used to hold the **Class** references.

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

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

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

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

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

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

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

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

Of course, the appropriate **Class** reference might not be in the **trashTypes** list. In this case, the **return** in the inner loop is never executed and you'll drop out at the end. Here, the program tries to rectify the situation by loading the **Class** object dynamically and adding it to the **trashTypes** list. If it still can't be found something is really wrong, but if the load is successful then the **factory** method is called recursively to try again.

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

Trash subclasses

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

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

```
//: c12:trash:Aluminum.java
// The Aluminum class with prototyping.
package c12.trash;

public class Aluminum extends Trash {
  private static double val = 1.67f;
  public Aluminum(double wt) { super(wt); }
  public double getValue() { return val; }
  public static void setValue(double newVal) {
    val = newVal;
  }
} ///:~
```

```
//: c12:trash:Paper.java
// The Paper class with prototyping.
package c12.trash;

public class Paper extends Trash {
  private static double val = 0.10f;
  public Paper(double wt) { super(wt); }
  public double getValue() { return val; }
  public static void setValue(double newVal) {
    val = newVal;
  }
} ///:~
```

```
//: c12:trash:Glass.java
// The Glass class with prototyping.
package c12.trash;

public class Glass extends Trash {
  private static double val = 0.23f;
  public Glass(double wt) { super(wt); }
  public double getValue() { return val; }
  public static void setValue(double newVal) {
    val = newVal;
  }
} ///:~
```

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

```
//: c12:trash:Cardboard.java
// The Cardboard class with prototyping.
package c12.trash;

public class Cardboard extends Trash {
  private static double val = 0.23f;
  public Cardboard(double wt) { super(wt); }
  public double getValue() { return val; }
  public static void setValue(double newVal) {
    val = newVal;
  }
} ///:~
```

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

Parsing **Trash** from an external file

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

```
//:! 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.

To parse this, the line is read and the **String** method **indexOf( )** produces the index of the '**:**'. This is first used with the **String** method **substring( )** to extract the name of the trash type, and next to get the weight that is turned into a **double** with the **static Double.valueOf( )** method. The **trim( )** method removes white space at both ends of a string.

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

```
//: c12:trash:ParseTrash.java
// Parse file contents into Trash objects,
// placing each into a Fillable holder.
package c12.trash;
import java.util.*;
import java.io.*;
```

```
public class ParseTrash {
  public static void
  fillBin(String filename, Fillable bin) {
    try {
      BufferedReader data =
        new BufferedReader(
          new FileReader(filename));
      String buf;
      while((buf = data.readLine())!= null) {
        String type = buf.substring(0,
          buf.indexOf(':')).trim();
        double weight = Double.valueOf(
          buf.substring(buf.indexOf(':') + 1)
          .trim()).doubleValue();
        bin.addTrash(
          Trash.factory(
            new Trash.Messenger(type, weight)));
      }
      data.close();
    } catch(Exception e) {
      e.printStackTrace(System.err);
      // Change to an unchecked exception, for
      // ease of coding, but the unit test
      // mechanism will still be triggered:
      throw new RuntimeException();
    }
  }
  // Special case to handle Collection:
  public static void
  fillBin(String filename, Collection bin) {
    fillBin(filename, new FillableCollection(bin));
  }
} ///:~
```

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

```
//: c12:trash:Fillable.java
// Any object that can be filled with Trash.
package c12.trash;
```

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

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

```
//: c12:trash:FillableCollection.java
// Adapter that makes a Collection Fillable.
package c12.trash;
import java.util.*;

public class FillableCollection
implements Fillable {
  private Collection c;
  public FillableCollection(Collection cc) {
    c = cc;
  }
  public void addTrash(Trash t) {
    c.add(t);
  }
} ///:~
```

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

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

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

Recycling with prototyping

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

```
//: c12:recycleap:RecycleAP.java
// Recycling with RTTI and Prototypes.
import c12.trash.*;
import java.util.*;
import com.bruceeckel.test.*;

public class RecycleAP extends UnitTest {
  Collection
    bin = new ArrayList(),
    glassBin = new ArrayList(),
    paperBin = new ArrayList(),
    alBin = new ArrayList();
  public RecycleAP() {
    // Fill up the Trash bin:
    ParseTrash.fillBin(
      "../trash/Trash.dat", bin);
  }
  public void test() {
    Iterator sorter = bin.iterator();
    // Sort the Trash:
    while(sorter.hasNext()) {
      Object t = sorter.next();
      // RTTI to show class membership:
      if(t instanceof Aluminum)
        alBin.add(t);
      if(t instanceof Paper)
        paperBin.add(t);
      if(t instanceof Glass)
        glassBin.add(t);
    }
    Trash.sumValue(alBin.iterator());
    Trash.sumValue(paperBin.iterator());
    Trash.sumValue(glassBin.iterator());
    Trash.sumValue(bin.iterator());
  }
  public static void main(String args[]) {
    new RecycleAP().test();
  }
} ///:~
```

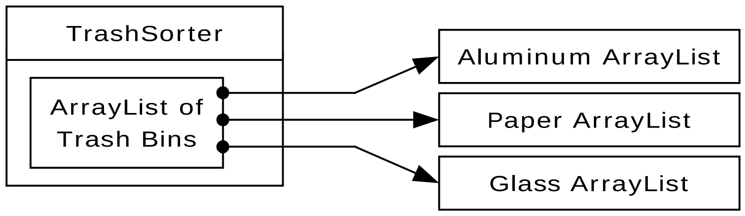
All of the **Trash** objects, as well as the **ParseTrash** and support classes, are now part of the package **c12.trash**, so they are simply imported.

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

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

Abstracting usage

With creation out of the way, it's time to tackle the remainder of the design: where the classes are used. Since it's the act of sorting into bins that's particularly ugly and exposed, why not take that process and hide it inside a class? This is the principle of "If you must do something ugly, at least localize the ugliness inside a class." It looks like this:



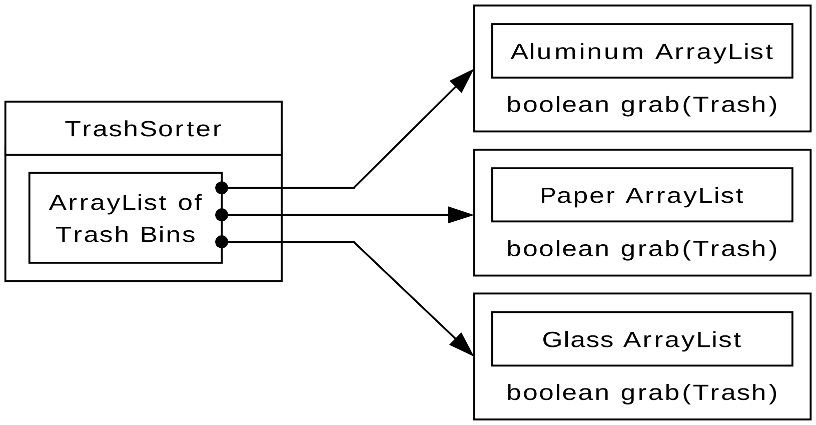
The **TrashSorter** object initialization must now be changed whenever a new type of **Trash** is added to the model. You could imagine that the **TrashSorter** class might look something like this:

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

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

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

Now, however, **sort( )** becomes a problem. How does the statically-coded method deal with the fact that a new type has been added? To solve this, the type information must be removed from **sort( )** so that all it needs to do is call a generic method that takes care of the details of type. This, of course, is another way to describe a dynamically-bound method. So **sort( )** will simply move through the sequence and call a dynamically-bound method for each **ArrayList**. Since the job of this method is to grab the pieces of trash it is interested in, it's called **grab(Trash)**. The structure now looks like:



TrashSorter needs to call each **grab( )** method and get a different result depending on what type of **Trash** the current **ArrayList** is holding. That is, each **ArrayList** must be aware of the type it holds. The classic approach to this problem is to create a base "**Trash** bin" class and inherit a new derived class for each different type you want to hold. If Java had a parameterized type mechanism that would probably be the most straightforward approach. But rather than hand-coding all the classes that such a mechanism should be building for us, further observation can produce a better approach.

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

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

Here is the whole program. The commented numbers (e.g., (*1*)) mark sections that will be described following the code.

```
//: c12:recycleb:RecycleB.java
// Adding more objects to the recycling problem.
import c12.trash.*;
import java.util.*;
import com.bruceeckel.test.*;

// A container that admits only the right type
// of Trash (established in the constructor):
class Tbin implements Fillable {
  private Collection list = new ArrayList();
  private Class type;
  public Tbin(Class binType) { type = binType; }
  public void addTrash(Trash t) { list.add(t); }
  public boolean grab(Trash t) {
    // Comparing class types:
    if(t.getClass().equals(type)) {
      list.add(t);
      return true; // Object grabbed
    }
    return false; // Object not grabbed
  }
  public Iterator iterator() {
    return list.iterator();
  }
}

class TbinList extends ArrayList { //(*1*)
  boolean sort(Trash t) {
    Iterator e = iterator();
    while(e.hasNext()) {
      Tbin bin = (Tbin)e.next();
      if(bin.grab(t)) return true;
    }
    return false; // bin not found for t
  }
  void sortBin(Tbin bin) { // (*2*)
    Iterator e = bin.iterator();
    while(e.hasNext())
      if(!sort((Trash)e.next()))
        System.out.println("Bin not found");
  }
}

public class RecycleB extends UnitTest {
  Tbin bin = new Tbin(Trash.class);
  TbinList trashBins = new TbinList();
  public RecycleB() {
    // Fill up the Trash bin:
    ParseTrash.fillBin(
      "../trash/Trash.dat", bin);
    trashBins.add(new Tbin(Aluminum.class));
    trashBins.add(new Tbin(Paper.class));
    trashBins.add(new Tbin(Glass.class));
    // add one line here: (*3*)
    trashBins.add(new Tbin(Cardboard.class));
  }
  public void test() {
    trashBins.sortBin(bin); // (*4*)
    Iterator e = trashBins.iterator();
    while(e.hasNext()) {
      Tbin b = (Tbin)e.next();
      Trash.sumValue(b.iterator());
    }
    Trash.sumValue(bin.iterator());
  }
  public static void main(String args[]) {
    new RecycleB().test();
  }
} ///:~
```

1. **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.

2. **sortBin( )** allows you to pass an entire **Tbin** in, and it moves through the **Tbin**, picks out each piece of **Trash**, and sorts it into the appropriate specific **Tbin**. 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.

3. Now you can see how easy it is to add a new type. Few lines must be changed to support the addition. If it's really important, you can squeeze out even more by further manipulating the design.

4. One method call causes the contents of **bin** to be sorted into the respective specifically-typed bins.

Multiple dispatching

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

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

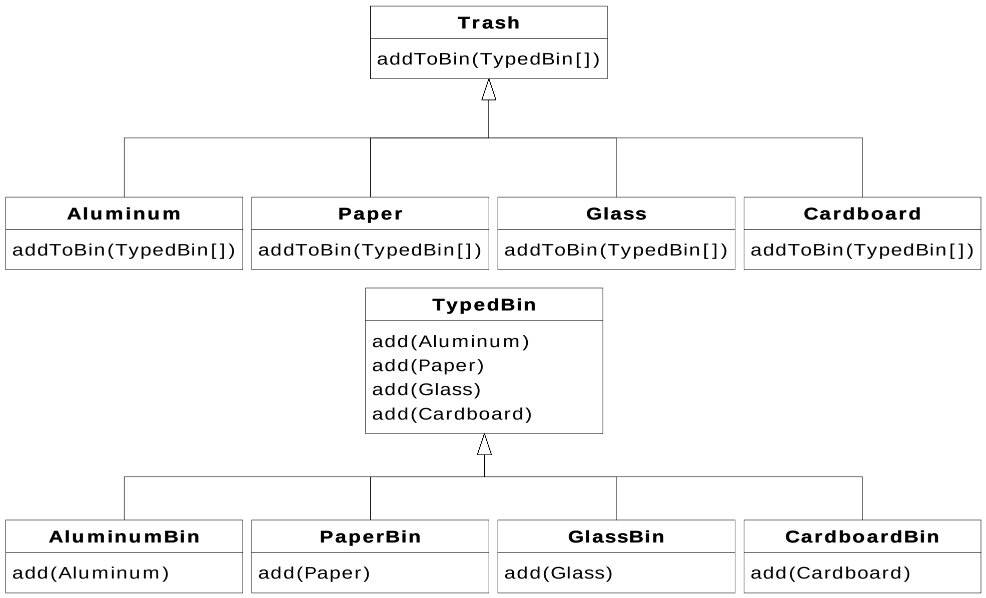
The previous examples first sorted by type, then acted on sequences of elements that were all of a particular type. But whenever you find yourself picking out particular types, stop and think. The whole idea of polymorphism (dynamically-bound method calls) is to handle type-specific information for you.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

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

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

Implementing the double dispatch

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



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

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

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

```
//: c12:doubledispatch:TypedBinMember.java
// An interface for adding the double
// dispatching method to the trash hierarchy
// without modifying the original hierarchy.

interface TypedBinMember {
  // The new method:
  boolean addToBin(TypedBin[] tb);
} ///:~
```

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

```
//: c12:doubledispatch:DDAluminum.java
// Aluminum for double dispatching.
import c12.trash.*;

public class DDAluminum extends Aluminum
    implements TypedBinMember {
  public DDAluminum(double wt) { super(wt); }
  public boolean addToBin(TypedBin[] tb) {
    for(int i = 0; i < tb.length; i++)
      if(tb[i].add(this))
        return true;
    return false;
  }
} ///:~
```

```
//: c12:doubledispatch:DDPaper.java
// Paper for double dispatching.
import c12.trash.*;

public class DDPaper extends Paper
    implements TypedBinMember {
  public DDPaper(double wt) { super(wt); }
  public boolean addToBin(TypedBin[] tb) {
    for(int i = 0; i < tb.length; i++)
      if(tb[i].add(this))
        return true;
    return false;
  }
} ///:~
```

```
//: c12:doubledispatch:DDGlass.java
// Glass for double dispatching.
import c12.trash.*;

public class DDGlass extends Glass
    implements TypedBinMember {
  public DDGlass(double wt) { super(wt); }
  public boolean addToBin(TypedBin[] tb) {
    for(int i = 0; i < tb.length; i++)
      if(tb[i].add(this))
        return true;
    return false;
  }
} ///:~
```

```
//: c12:doubledispatch:DDCardboard.java
// Cardboard for double dispatching.
import c12.trash.*;

public class DDCardboard extends Cardboard
    implements TypedBinMember {
  public DDCardboard(double wt) { super(wt); }
  public boolean addToBin(TypedBin[] tb) {
    for(int i = 0; i < tb.length; i++)
      if(tb[i].add(this))
        return true;
    return false;
  }
} ///:~
```

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

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

```
//: c12:doubledispatch:TypedBin.java
// A container for the second dispatch.
import c12.trash.*;
import java.util.*;

public abstract class TypedBin {
  Collection c = new ArrayList();
  protected boolean addIt(Trash t) {
    c.add(t);
    return true;
  }
  public Iterator iterator() {
    return c.iterator();
  }
  public boolean add(DDAluminum a) {
    return false;
  }
  public boolean add(DDPaper a) {
    return false;
  }
  public boolean add(DDGlass a) {
    return false;
  }
  public boolean add(DDCardboard a) {
    return false;
  }
} ///:~
```

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

In each of the subclasses of **TypedBin**, only one overloaded method is overridden, according to the type of bin that's being created. For example, **CardboardBin** overrides **add(DDCardboard)**. The overridden method adds the trash object to its container and returns **true**, while all the rest of the **add( )** methods in **CardboardBin** continue to return **false**, since they haven't been overridden. This is another case in which a parameterized type mechanism in Java would allow automatic generation of code. (With C++ **template**s, you wouldn't have to explicitly write the subclasses or place the **addToBin( )** method in **Trash**.)

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

```
//:! 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:

```
//: c12:doubledispatch:DoubleDispatch.java
// Using multiple dispatching to handle more
// than one unknown type during a method call.
import c12.trash.*;
import java.util.*;
import com.bruceeckel.test.*;

class AluminumBin extends TypedBin {
  public boolean add(DDAluminum a) {
    return addIt(a);
  }
}

class PaperBin extends TypedBin {
  public boolean add(DDPaper a) {
    return addIt(a);
  }
}

class GlassBin extends TypedBin {
  public boolean add(DDGlass a) {
    return addIt(a);
  }
}

class CardboardBin extends TypedBin {
  public boolean add(DDCardboard a) {
    return addIt(a);
  }
}
```

```
class TrashBinSet {
  private TypedBin[] binSet = {
    new AluminumBin(),
    new PaperBin(),
    new GlassBin(),
    new CardboardBin()
  };
  public void sortIntoBins(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; }
}

public class DoubleDispatch extends UnitTest {
  Collection bin = new ArrayList();
  TrashBinSet bins = new TrashBinSet();
  public DoubleDispatch() {
    // ParseTrash still works, without changes:
    ParseTrash.fillBin("DDTrash.dat", bin);
  }
  public void test() {
    // Sort from the master bin into
    // the individually-typed bins:
    bins.sortIntoBins(bin);
    TypedBin[] tb = bins.binSet();
    // Perform sumValue for each bin...
    for(int i = 0; i < tb.length; i++)
      Trash.sumValue(tb[i].c.iterator());
    // ... and for the master bin
    Trash.sumValue(bin.iterator());
  }
  public static void main(String args[]) {
    new 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.

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

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

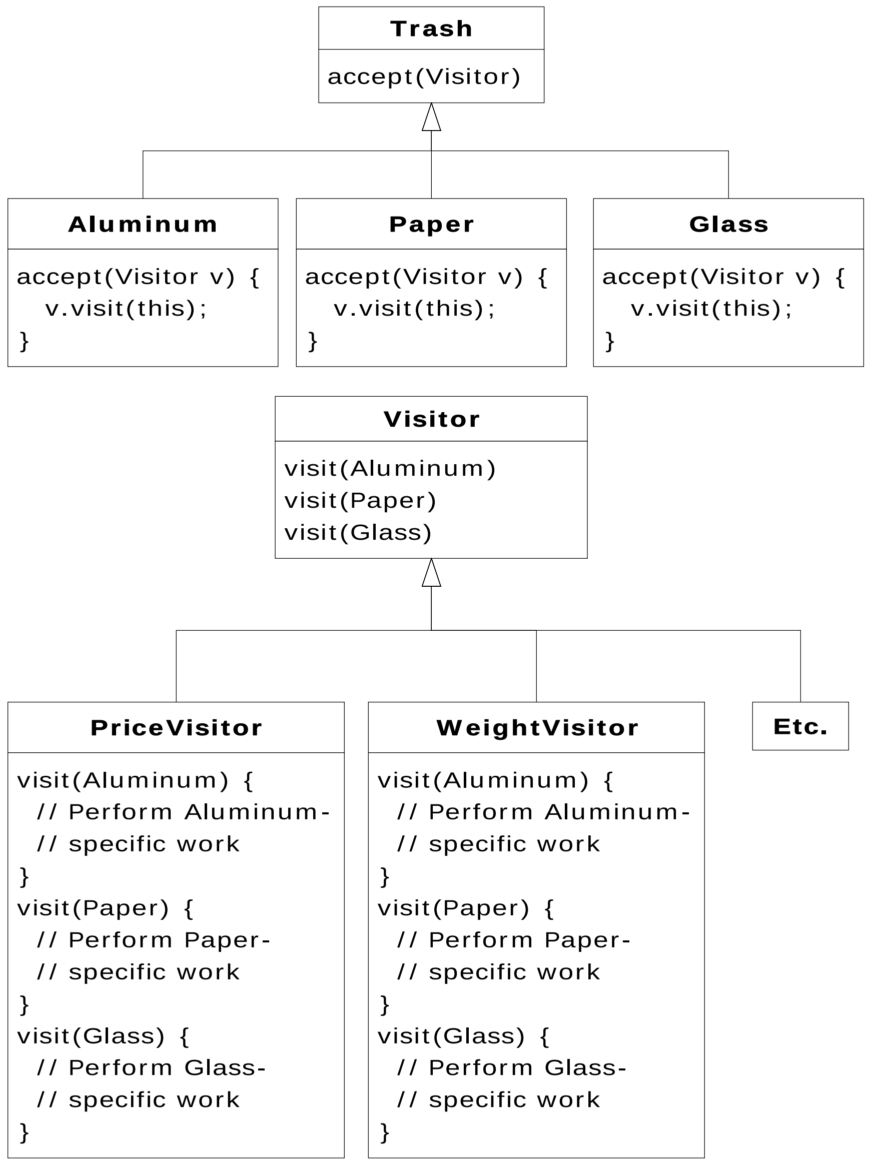
The *Visitor* pattern

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

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

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

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



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

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

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

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

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

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

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

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

```
//: c12:trashvisitor:Visitable.java
// An interface to add visitor functionality
// to the Trash hierarchy without
// modifying the base class.
import c12.trash.*;

interface Visitable {
  // The new method:
  void accept(Visitor v);
} ///:~
```

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

```
//: c12:trashvisitor:Visitor.java
// The base interface for visitors.
import c12.trash.*;

interface Visitor {
  void visit(Aluminum a);
  void visit(Paper p);
  void visit(Glass g);
  void visit(Cardboard c);
} ///:~
```

A Reflective Decorator

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

```
//: c12:trashvisitor:VAluminum.java
// Taking the previous approach of creating a
// specialized Aluminum for the visitor pattern.
import c12.trash.*;

public class VAluminum extends Aluminum
    implements Visitable {
  public VAluminum(double wt) { super(wt); }
  public void accept(Visitor v) {
    v.visit(this);
  }
} ///:~
```

However, we seem to be encountering an “explosion of interfaces:” basic **Trash**, special versions for double dispatching, and now more special versions for visitor. Of course, this “explosion of interfaces” is arbitrary—one could simply put the additional methods in the **Trash** class. If we ignore that we can instead see an opportunity to use the *Decorator* pattern: it seems like it should be possible to create a *Decorator* that can be wrapped around an ordinary **Trash** object and will produce the same interface as **Trash** and add the extra **accept( )** method. In fact, it's a perfect example of the value of *Decorator*.

The double dispatch creates a problem, however. Since it relies on overloading of both **accept( )** and **visit( )**, it would seem to require specialized code for each different version of the **accept( )** method. With C++ templates, this would be fairly easy to accomplish (since templates automatically generate type-specialized code) but Java has no such mechanism—at least it does not appear to. However, reflection allows you to determine type information at run time, and it turns out to solve many problems that would seem to require templates (albeit not as simply). Here's the decorator that does the trick[2]:

```
//: c12:trashvisitor:VisitableDecorator.java
// A decorator that adapts the generic Trash
// classes to the visitor pattern.
import c12.trash.*;
import java.lang.reflect.*;

public class VisitableDecorator
extends Trash implements Visitable {
  private Trash delegate;
  private Method dispatch;
  public VisitableDecorator(Trash t) {
    delegate = t;
    try {
      dispatch = Visitor.class.getMethod (
        "visit", new Class[] { t.getClass() }
      );
    } catch (Exception ex) {
      ex.printStackTrace();
    }
  }
  public double getValue() {
    return delegate.getValue();
  }
```

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

```
  public double getWeight() {
    return delegate.getWeight();
  }
  public void accept(Visitor v) {
    try {
      dispatch.invoke(v, new Object[]{delegate});
    } catch (Exception ex) {
      ex.printStackTrace();
    }
  }
} ///:~
```

[[Description of Reflection use]]

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**:

```
//: c12:trashvisitor:FillableVisitor.java
// Adapter Decorator that adds the visitable
// decorator as the Trash objects are
// being created.
import c12.trash.*;
import java.util.*;

public class FillableVisitor
implements Fillable {
  private Fillable f;
  public FillableVisitor(Fillable ff) { f = ff; }
  public void addTrash(Trash t) {
    f.addTrash(new VisitableDecorator(t));
  }
} ///:~
```

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

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

```
//: c12:trashvisitor:TrashVisitor.java
// The "visitor" pattern with VisitableDecorators.
```

```
import c12.trash.*;
import java.util.*;
import com.bruceeckel.test.*;

// Specific group of algorithms packaged
// in each implementation of Visitor:
class PriceVisitor implements Visitor {
  private double alSum; // Aluminum
  private double pSum; // Paper
  private double gSum; // Glass
  private double cSum; // Cardboard
  public void visit(Aluminum al) {
    double v = al.getWeight() * al.getValue();
    System.out.println(
      "value of Aluminum= " + v);
    alSum += v;
  }
  public void visit(Paper p) {
    double v = p.getWeight() * p.getValue();
    System.out.println(
      "value of Paper= " + v);
    pSum += v;
  }
  public void visit(Glass g) {
    double v = g.getWeight() * g.getValue();
    System.out.println(
      "value of Glass= " + v);
    gSum += v;
  }
  public void visit(Cardboard c) {
    double v = c.getWeight() * c.getValue();
    System.out.println(
      "value of Cardboard = " + v);
    cSum += v;
  }
  void total() {
    System.out.println(
      "Total Aluminum: $" + alSum + "\n" +
      "Total Paper: $" + pSum + "\n" +
      "Total Glass: $" + gSum + "\n" +
      "Total Cardboard: $" + cSum);
  }
}
```

```
class WeightVisitor implements Visitor {
  private double alSum; // Aluminum
  private double pSum; // Paper
  private double gSum; // Glass
  private double cSum; // Cardboard
  public void visit(Aluminum al) {
    alSum += al.getWeight();
    System.out.println("weight of Aluminum = "
        + al.getWeight());
  }
  public void visit(Paper p) {
    pSum += p.getWeight();
    System.out.println("weight of Paper = "
        + p.getWeight());
  }
  public void visit(Glass g) {
    gSum += g.getWeight();
    System.out.println("weight of Glass = "
        + g.getWeight());
  }
  public void visit(Cardboard c) {
    cSum += c.getWeight();
    System.out.println("weight of Cardboard = "
        + c.getWeight());
  }
  void total() {
    System.out.println("Total weight Aluminum:"
        + alSum);
    System.out.println("Total weight Paper:"
        + pSum);
    System.out.println("Total weight Glass:"
        + gSum);
    System.out.println("Total weight Cardboard:"
        + cSum);
  }
}

public class TrashVisitor extends UnitTest {
  Collection bin = new ArrayList();
  PriceVisitor pv = new PriceVisitor();
  WeightVisitor wv = new WeightVisitor();
  public TrashVisitor() {
    ParseTrash.fillBin("../trash/Trash.dat",
      new FillableVisitor(
        new FillableCollection(bin)));
  }
  public void test() {
    Iterator it = bin.iterator();
    while(it.hasNext()) {
      Visitable v = (Visitable)it.next();
      v.accept(pv);
      v.accept(wv);
    }
    pv.total();
    wv.total();
  }
  public static void main(String args[]) {
    new 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.

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

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

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

More coupling?

There's a lot more code here, and there's definite coupling between the **Trash** hierarchy and the **Visitor** hierarchy. However, there's also high cohesion within the respective sets of classes: they each do only one thing (**Trash** describes Trash, while **Visitor** describes actions performed on **Trash**), which is an indicator of a good design. Of course, in this case it works well only if you're adding new **Visitor**s, but it gets in the way when you add new types of **Trash**.

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

RTTI considered harmful?

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

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

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):

```
//: c12:dynatrash:DynaTrash.java
// Using a Map of Lists and RTTI
// to automatically sort trash into
// ArrayLists. This solution, despite the
// use of RTTI, is extensible.
import c12.trash.*;
import java.util.*;
import com.bruceeckel.test.*;

// Generic TypeMap works in any situation:
class TypeMap {
  private Map t = new HashMap();
  public void add(Object o) {
    Class type = o.getClass();
    if(t.containsKey(type))
      ((List)t.get(type)).add(o);
    else {
      List v = new ArrayList();
      v.add(o);
      t.put(type,v);
    }
  }
  public List get(Class type) {
    return (List)t.get(type);
  }
  public Iterator keys() {
    return t.keySet().iterator();
  }
}

// Adapter class to allow callbacks
// from ParseTrash.fillBin():
class TypeMapAdapter implements Fillable {
  TypeMap map;
  public TypeMapAdapter(TypeMap tm) { map = tm; }
  public void addTrash(Trash t) { map.add(t); }
}

public class DynaTrash extends UnitTest {
  TypeMap bin = new TypeMap();
  public DynaTrash() {
    ParseTrash.fillBin("../trash/Trash.dat",
      new TypeMapAdapter(bin));
  }
  public void test() {
    Iterator keys = bin.keys();
    while(keys.hasNext())
      Trash.sumValue(
        bin.get((Class)keys.next()).iterator());
  }
  public static void main(String args[]) {
    new 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.

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

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

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

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

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

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

```
    Iterator keys = bin.keys();
    while(keys.hasNext())
      Trash.sumValue(
        bin.get((Class)keys.next())iterator());
```

As you can see, adding a new type to the system won't affect this code at all, nor the code in **TypeMap**. This is certainly the smallest solution to the problem, and arguably the most elegant as well. It does rely heavily on RTTI, but notice that each key-value pair in the **HashMap** is looking for only one type. In addition, there's no way you can "forget" to add the proper code to this system when you add a new type, since there isn't any code you need to add.

Summary

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

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

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

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

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

http://st-www.cs.uiuc.edu/users/patterns
http://c2.com/cgi/wiki
http://c2.com/ppr
http://www.bell-labs.com/people/cope/Patterns/Process/index.html
http://www.bell-labs.com/cgi-user/OrgPatterns/OrgPatterns
http://st-www.cs.uiuc.edu/cgi-bin/wikic/wikic
http://www.cs.wustl.edu/~schmidt/patterns.html
http://www.espinc.com/patterns/overview.html

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

Exercises

1. Add a class **Plastic** to **TrashVisitor.java**.
2. Add a class **Plastic** to **DynaTrash.java**.
3. Create a decorator like **VisitableDecorator**, but for the multiple dispatching example, along with an "adapter decorator" class like the one created for **VisitableDecorator**. Build the rest of the example and show that it works.

13: Projects

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

Rats & Mazes

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

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

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

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

An example implementation by Jeremy Meyer:

```
//: c13:Maze.java
```

```
import java.util.*;
import java.io.*;
import java.awt.*;

public class Maze extends Canvas {
  private Vector lines; // a line is a char array
  private int width = -1;
  private int height = -1;
  public static void main (String [] args)
  throws IOException {
    if (args.length < 1) {
      System.out.println("Enter filename");
      System.exit(0);
    }
    Maze m = new Maze();
    m.load(args[0]);
    Frame f = new Frame();
    f.setSize(m.width*20, m.height*20);
    f.add(m);
    Rat r = new Rat(m, 0, 0);
    f.setVisible(true);
  }
  public Maze() {
    lines = new Vector();
    setBackground(Color.lightGray);
  }
  synchronized public boolean
  isEmptyXY(int x, int y) {
    if (x < 0) x += width;
    if (y < 0) y += height;
    // Use mod arithmetic to bring rat in line:
    byte[] by =
      (byte[])(lines.elementAt(y%height));
    return by[x%width]==' ';
  }
  synchronized public void
  setXY(int x, int y, byte newByte) {
    if (x < 0) x += width;
    if (y < 0) y += height;
    byte[] by =
      (byte[])(lines.elementAt(y%height));
    by[x%width] = newByte;
    repaint();
  }
```

```
public void
load(String filename) throws IOException {
  String currentLine = null;
  BufferedReader br = new BufferedReader(
    new FileReader(filename));
  for(currentLine = br.readLine();
      currentLine != null;
      currentLine = br.readLine())  {
    lines.addElement(currentLine.getBytes());
    if(width < 0 ||
       currentLine.getBytes().length > width)
      width = currentLine.getBytes().length;
  }
  height = lines.size();
  br.close();
}
public void update(Graphics g) { paint(g); }
public void paint (Graphics g) {
  int canvasHeight = this.getBounds().height;
  int canvasWidth  = this.getBounds().width;
  if (height < 1 || width < 1)
    return; // nothing to do
  int width =
    ((byte[])(lines.elementAt(0))).length;
  for (int y = 0; y < lines.size(); y++) {
    byte[] b;
    b = (byte[])(lines.elementAt(y));
    for (int x = 0; x < width; x++) {
      switch(b[x]) {
        case ' ': // empty part of maze
          g.setColor(Color.lightGray);
          g.fillRect(
            x*(canvasWidth/width),
            y*(canvasHeight/height),
            canvasWidth/width,
            canvasHeight/height);
          break;
        case '*':     // a wall
          g.setColor(Color.darkGray);
          g.fillRect(
            x*(canvasWidth/width),
            y*(canvasHeight/height),
            (canvasWidth/width)-1,
            (canvasHeight/height)-1);
          break;
        default:      // must be rat
          g.setColor(Color.red);
          g.fillOval(x*(canvasWidth/width),
          y*(canvasHeight/height),
          canvasWidth/width,
          canvasHeight/height);
          break;
        }
      }
    }
  }
} ///:~
```

```
//: c13:Rat.java

public class Rat {
  static int ratCount = 0;
  private Maze prison;
  private int vertDir = 0;
  private int horizDir = 0;
  private int x,y;
  private int myRatNo = 0;
  public Rat(Maze maze, int xStart, int yStart) {
    myRatNo = ratCount++;
    System.out.println("Rat no." + myRatNo +
      " ready to scurry.");
    prison = maze;
    x = xStart;
    y = yStart;
    prison.setXY(x,y, (byte)'R');
    new Thread() {
      public void run(){ scurry(); }
    }.start();
  }
  public void scurry() {
    // Try and maintain direction if possible.
    // Horizontal backward
    boolean ratCanMove = true;
    while(ratCanMove) {
      ratCanMove = false;
      // South
      if (prison.isEmptyXY(x, y + 1)) {
        vertDir = 1; horizDir = 0;
        ratCanMove = true;
      }
      // North
      if (prison.isEmptyXY(x, y - 1))
        if (ratCanMove)
          new Rat(prison, x, y-1);
          // Rat can move already, so give
          // this choice to the next rat.
        else {
          vertDir = -1; horizDir = 0;
          ratCanMove = true;
        }
      // West
      if (prison.isEmptyXY(x-1, y))
        if (ratCanMove)
          new Rat(prison, x-1, y);
          // Rat can move already, so give
          // this choice to the next rat.
        else {
          vertDir = 0; horizDir = -1;
          ratCanMove = true;
        }
      // East
      if (prison.isEmptyXY(x+1, y))
        if (ratCanMove)
          new Rat(prison, x+1, y);
          // Rat can move already, so give
          // this choice to the next rat.
        else {
          vertDir = 0; horizDir = 1;
          ratCanMove = true;
        }
      if (ratCanMove) { // Move original rat.
        x += horizDir;
        y += vertDir;
        prison.setXY(x,y,(byte)'R');
      }  // If not then the rat will die.
      try {
        Thread.sleep(2000);
      } catch(InterruptedException ie) {}
    }
    System.out.println("Rat no." + myRatNo +
      " can't move..dying..aarrgggh.");
  }
```

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

The maze initialization file:

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

Other maze resources

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

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

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

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

XML Decorator

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