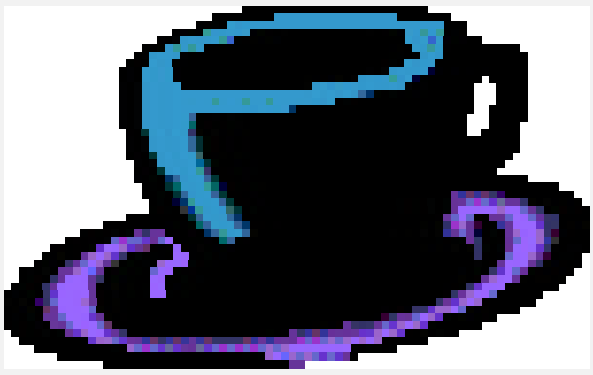
THE

DESIGN PATTERNS

JAVA COMPANION

JAMES W. COOPER



Addison-Wesley Design Patterns Series

October 2, 1998

Copyright © 1998, by James W. Cooper

Some Background on Design Patterns *10*
Defining Design Patterns 11
This Book and its Parentage 13
The Learning Process 13
Studying Design Patterns 14
Notes on Object Oriented Approaches 14
The Java Foundation Classes 15
Java Design Patterns 15
1. Creational Patterns 17
The Factory Pattern *18*
How a Factory Works 18
Sample Code 18
The Two Derived Classes 19
Building the Factory 20
Factory Patterns in Math Computation 22
When to Use a Factory Pattern 24
Thought Questions 25
The Abstract Factory Pattern *26*
A GardenMaker Factory 26
How the User Interface Works 28
Consequences of Abstract Factory 30
Thought Questions 30
The Singleton Pattern *31*
Throwing the Exception 32
Creating an Instance of the Class 32
Static Classes as Singleton Patterns 33
Creating Singleton Using a Static Method 34

Finding the Singletons in a Large Program 35
Other Consequences of the Singleton Pattern 35
The Builder Pattern 37
An Investment Tracker 38
Calling the Builders 40
The List Box Builder 42
The Checkbox Builder 43
Consequences of the Builder Pattern 44
Thought Questions 44
The Prototype Pattern 45
Cloning in Java 45
Using the Prototype 47
Consequences of the Prototype Pattern 50
Summary of Creational Patterns 51
2. The Java Foundation Classes 52
Installing and Using the JFC 52
Ideas Behind Swing 53
The Swing Class Hierarchy 53
Writing a Simple JFC Program 54
Setting the Look and Feel 54
Setting the Window Close Box 55
Making a JxFrame Class 55
A Simple Two Button Program 56
More on JButtons 57
Buttons and Toolbars 59
Radio Buttons 59
The JToolBar 59
Toggle Buttons 60

Sample Code 61

Menus and Actions ***62***

Action Objects 62

Design Patterns in the Action Object 65

The JList Class ***67***

List Selections and Events 68

Changing a List Display Dynamically 69

The JTable Class ***71***

A Simple JTable Program 71

Cell Renderers 74

The JTree Class ***77***

The TreeModel Interface 78

Summary 79

3. Structural Patterns 80

The Adapter Pattern ***81***

Moving Data between Lists 81

Using the JFC JList Class 83

Two Way Adapters 87

Pluggable Adapters 87

Adapters in Java 88

The Bridge Pattern ***90***

Building a Bridge 91

Consequences of the Bridge Pattern 93

The Composite Pattern ***95***

An Implementation of a Composite 96

Building the Employee Tree 98

Restrictions on Employee Classes 100

Consequences of the Composite Pattern 100
Other Implementation Issues 101
The Decorator Pattern 103
Decorating a CoolButton 103
Using a Decorator 105
Inheritance Order 107
Decorating Borders in Java 107
Non-Visual Decorators 109
Decorators, Adapters and Composites 110
Consequences of the Decorator Pattern 110
The Façade Pattern 111
Building the Façade Classes 112
Consequences of the Façade 115
The Flyweight Pattern 117
Discussion 117
Example Code 118
Flyweight Uses in Java 122
Sharable Objects 122
The Proxy Pattern 124
Sample Code 124
Copy-on-Write 127
Comparison with Related Patterns 127
Summary of structural patterns 128
4. Behavioral Patterns 129
Chain of Responsibility 130
Applicability 130
Sample Code 131

The List Boxes 133
A Chain or a Tree? 135
Kinds of Requests 137
Examples in Java 137
Consequences of the Chain of Responsibility 138
The Command Pattern 139
Motivation 139
The Command Pattern 140
Building Command Objects 141
The Command Pattern in Java 142
Consequences of the Command Pattern 143
Providing Undo 144
The Interpreter Pattern 145
Motivation 145
Applicability 145
Sample Code 146
Interpreting the Language 147
Objects Used in Parsing 148
Reducing the Parsed Stack 150
Consequences of the Interpreter Pattern 153
The Iterator Pattern 155
Motivation 155
Enumerations in Java 156
Filtered Iterators 156
Sample Code 157
Consequence of the Iterator Pattern 159
Composites and Iterators 160
The Mediator Pattern 161

An Example System 161
Interactions between Controls 162
Sample Code 164
Mediators and Command Objects 167
Consequences of the Mediator Pattern 167
Implementation Issues 168
The Memento Pattern 169
Motivation 169
Implementation 169
Sample Code 170
Consequences of the Memento 175
Other Kinds of Mementos 176
The Observer Pattern 177
Watching Colors Change 178
The Message to the Media 181
Th JList as an Observer 182
The MVC Architecture as an Observer 183
Consequences of the Observer Pattern 184
The State Pattern 185
Sample Code 185
Switching Between States 190
How the Mediator Interacts with the State Manager 191
Consequences of the State Pattern 192
State Transitions 192
Thought Questions 192
The Strategy Pattern 194
Motivation 194
Sample Code 195

The Context 196
The Program Commands 197
The Line and Bar Graph Strategies 198
Drawing Plots in Java 198
Consequences of the Strategy Pattern 201
The Template Pattern ***202***
Motivation 202
Kinds of Methods in a Template Class 203
Sample Code 204
The Triangle Drawing Program 207
Templates and Callbacks 208
Summary and Consequences 209
The Visitor Pattern ***210***
Motivation 210
When to Use the Visitor Pattern 211
Sample Code 212
Visiting Several Classes 214
Bosses are Employees, too 215
Double Dispatching 216
Traversing a Series of Classes 216
Consequence of the Visitor Pattern 216

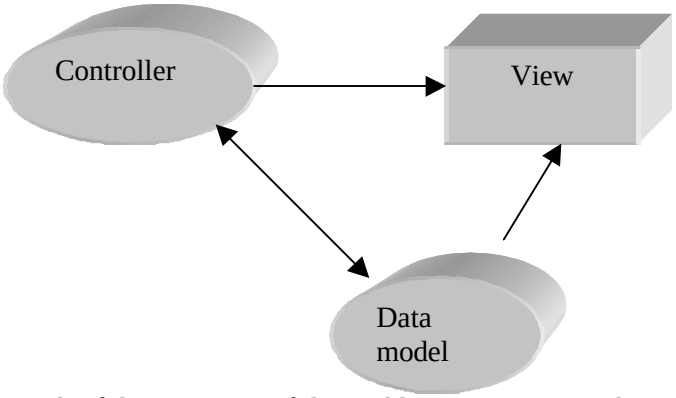
5.

SOME BACKGROUND ON DESIGN PATTERNS

The term “design patterns” sounds a bit formal to the uninitiated and can be somewhat off-putting when you first encounter it. But, in fact, design patterns are just convenient ways of reusing object-oriented code between projects and between programmers. The idea behind design patterns is simple-- write down and catalog common interactions between objects that programmers have frequently found useful.

The field of design patterns goes back at least to the early 1980s. At that time, Smalltalk was the most common OO language and C++ was still in its infancy. At that time, structured programming was a commonly-used phrased and OO programming was not yet as widely supported. The idea of programming frameworks was popular however, and as frameworks developed, some of what we now called design patterns began to emerge.

One of the frequently cited frameworks was the Model-View-Controller framework for Smalltalk [Krasner and Pope, 1988], which divided the user interface problem into three parts. The parts were referred to as a *data model* which contain the computational parts of the program, the *view*, which presented the user interface, and the c*ontroller*, which interacted between the user and the view.



Each of these aspects of the problem is a separate object and each has its own rules for managing its data. Communication between the user, the GUI and the data should be carefully controlled and this separation of functions accomplished that very nicely. Three objects talking to each other using this restrained set of connections is an example of a powerful *design pattern*.

In other words, design patterns describe how objects communicate without become entangled in each other's data models and methods. Keeping this separation has always been an objective of good OO programming, and if you have been trying to keep objects minding their own business, you are probably using some of the common design patterns already. Interestingly enough, the MVC pattern has resurfaced now and we find it used in Java 1.2 as part of the Java Foundation Classes (JFC, or the "Swing" components).

Design patterns began to be recognized more formally in the early 1990s by Helm (1990) and Erich Gamma (1992), who described patterns incorporated in the GUI application framework, ET++. The culmination of these discussions and a number of technical meetings was the publication of the parent book in this series, *Design Patterns -- Elements of Reusable Software*, by Gamma, Helm, Johnson and Vlissides.(1995). This book, commonly referred to as the Gang of Four or "GoF" book, has had a powerful impact on those seeking to understand how to use design patterns and has become an all-time best seller. We will refer to this groundbreaking book as *Design Patterns*, throughout this book and *The Design Patterns Smalltalk Companion (*Alpert, Brown and Woolf, 1998) as the *Smalltalk Companion.*

Defining Design Patterns

We all talk about the way we do things in our everyday work, hobbies and home life and recognize repeating patterns all the time.

- Sticky buns are like dinner rolls, but I add brown sugar and nut filling to them.
- Her front garden is like mine, but, in mine I use *astilbe*.
- This end table is constructed like that one, but in this one, the doors replace drawers.

We see the same thing in programming, when we tell a colleague how we accomplished a tricky bit of programming so he doesn't have to recreate it from scratch. We simply recognize effective ways for objects to communicate while maintaining their own separate existences.

Some useful definitions of design patterns have emerged as the literature in his field has expanded:

- "Design patterns are recurring solutions to design problems you see over
et. al., 1998).

- "Design patterns constitute a set of rules describing how to accomplish certain tasks in the realm of software development." (Pree, 1994)
- "Design patterns focus more on reuse of recurring architectural design themes, while frameworks focus on detailed design… and implementation." (Coplien & Schmidt, 1995).
- "A pattern addresses a recurring design problem that arises in specific design situations and presents a solution to it" (Buschmann, *et. al.* 1996)
- "Patterns identify and specify abstractions that are above the level of single classes and instances, or of components." (Gamma, et al., 1993)

But while it is helpful to draw analogies to architecture, cabinet making and logic, design patterns are not just about the design of objects, but about the *communication* between objects. In fact, we sometimes think of them as *communication patterns*. It is the design of simple, but elegant, methods of communication that makes many design patterns so important.

Design patterns can exist at many levels from very low level specific solutions to broadly generalized system issues. There are now in fact hundreds of patterns in the literature. They have been discussed in articles and at conferences of all levels of granularity. Some are examples which have wide applicability and a few (Kurata, 1998) solve but a single problem.

It has become apparent that you don't just *write* a design pattern off the top of your head. In fact, most such patterns are *discovered* rather than written. The process of looking for these patterns is called "pattern mining," and is worthy of a book of its own.

The 23 design patterns selected for inclusion in the original *Design Patterns* book were ones which had several known applications and which were on a middle level of generality, where they could easily cross application areas and encompass several objects.

The authors divided these patterns into three types creational, structural and behavioral.

- *Creational patterns* are ones that create objects for you, rather than having you instantiate objects directly. This gives your program more flexibility in deciding which objects need to be created for a given case.
- *Structural patterns* help you compose groups of objects into larger structures, such as complex user interfaces or accounting data.

- *Behavioral patterns* help you define the communication between objects in your system and how the flow is controlled in a complex program.

We'll be looking at Java versions of these patterns in the chapters that follow.

This Book and its Parentage

Design Patterns is a catalog of 23 generally useful patterns for writing object-oriented software. It is written as a catalog with short examples and substantial discussions of how the patterns can be constructed and applied. Most of its examples are in C++, with a few in Smalltalk. The *Smalltalk Companion* (Alpert, 1998) follows a similar approach, but with somewhat longer examples, all in Smalltalk. Further, the authors present some additional very useful advice on implementing and using these patterns.

This book takes a somewhat different approach; we provide at least one complete, visual Java program for each of the 23 patterns. This way you can not only examine the code snippets we provide, but run, edit and modify the complete working programs on the accompanying CD-ROM. You'll find a list of all the programs on the CD-ROM in Appendix A.

The Learning Process

We have found learning Design patterns is a multiple step process.

1. Acceptance
2. Recognition
3. Internalization

First, you accept the premise that design patterns are important in your work. Then, you recognize that you need to read about design patterns in order to know when you might use them. Finally, you internalize the patterns in sufficient detail that you know which ones might help you solve a given design problem.

For some lucky people, design patterns are obvious tools and they grasp their essential utility just by reading summaries of the patterns. For many of the rest of us, there is a slow induction period after we've read about a pattern followed by the proverbial "Aha!" when we see how we can apply them in our work. This book helps to take you to that final stage of internalization by providing complete, working programs that you can try out for yourself.

The examples in *Design Patterns* are brief, and are in C++ or in some cases, Smalltalk. If you are working in another language it is helpful to have the pattern examples in your language of choice. This book attempts to fill that need for Java programmers.

A set of Java examples takes on a form that is a little different than in C++, because Java is more strict in its application of OO precepts -- you can't have global variables, data structures or pointers. In addition, we'll see that the Java interfaces and abstract classes are a major contributor to how we build Java design patterns.

Studying Design Patterns

There are several alternate ways to become familiar with these patterns. In each approach, you should read this book and the parent *Design Patterns* book in one order or the other. We also strongly urge you to read the *Smalltalk Companion* for completeness, since it provides an alternate description of each of the patterns. Finally, there are a number of web sites on learning and discussing Design Patterns for you to peruse.

Notes on Object Oriented Approaches

The fundamental reason for using varies design patterns is to keep classes separated and prevent them from having to know too much about one another. There are a number of strategies that OO programmers use to achieve this separation, among them encapsulation and inheritance.

Nearly all languages that have OO capabilities support inheritance. A class that inherits from a parent class has access to all of the methods of that parent class. It also has access to all of its non-private variables. However, by starting your inheritance hierarchy with a complete, working class you may be unduly restricting yourself as well as carrying along specific method implementation baggage. Instead, *Design Patterns* suggests that you always

Program to an interface and not to an implementation.

Purring this more succinctly, you should define the top of any class hierarchy with an *abstract* class, which implements no methods, but simply defines the methods that class will support. Then, in all of your derived classes you have more freedom to implement these methods as most suits your purposes.

The other major concept you should recognize is that of *object composition.* This is simply the construction of objects that contain others: encapsulation of

several objects inside another one. While many beginning OO programmers use inheritance to solve every problem, as you begin to write more elaborate programs, the merits of object composition become apparent. Your new object can have the interface that is best for what you want to accomplish without having all the methods of the parent classes. Thus, the second major precept suggested by *Design Patterns* is

Favor object composition over inheritance.

At first this seems contrary to the customs of OO programming, but you will see any number of cases among the design patterns where we find that inclusion of one or more objects inside another is the preferred method.

The Java Foundation Classes

The Java Foundation Classes (JFC) which were introduced after Java 1.1 and incorporated into Java 1.2 are a critical part of writing good Java programs. These were also known during development as the "Swing" classes and still are informally referred to that way. They provide easy ways to write very professional-looking user interfaces and allow you to vary the look and feel of your interface to match the platform your program is running on. Further, these classes themselves utilize a number of the basic design patterns and thus make extremely good examples for study.

Nearly all of the example programs in this book use the JFC to produce the interfaces you see in the example code. Since not everyone may be familiar with these classes, and since we are going to build some basic classes from the JFC to use throughout our examples, we take a short break after introducing the creational patterns and spend a chapter introducing the JFC. While the chapter is not a complete tutorial in every aspect of the JFC, it does introduce the most useful interface controls and shows how to use them.

Many of the examples do require that the JFC libraries are installed, and we describe briefly what Jar files you need in this chapter as well.

Java Design Patterns

Each of the 23 design patterns in *Design Patterns* is discussed in the chapters that follow, along with at least one working program example for that pattern. The authors of *Design Patterns* have suggested that every pattern start with an abstract class and that you derive concrete working

classes from that abstraction. We have only followed that suggestion in cases where there may be several examples of a pattern within a program. In other cases, we start right in with a concrete class, since the abstract class only makes the explanation more involved and adds little to the elegance of the implementation.

James W. Cooper
Wilton, Connecticut
Nantucket, Massachusetts

Creational Patterns

All of the creational patterns deal with the best way to create instances of objects. This is important because your program should not depend on how objects are created and arranged. In Java, of course, the simplest way to create an instance of an object is by using the **new** operator.

```
Fred = new Fred();  //instance of Fred class
```

However, this really amounts to hard coding, depending on how you create the object within your program. In many cases, the exact nature of the object that is created could vary with the needs of the program and abstracting the creation process into a special “creator” class can make your program more flexible and general.

The Factory Method provides a simple decision making class that returns one of several possible subclasses of an abstract base class depending on the data that are provided.

The Abstract Factory Method provides an interface to create and return one of several families of related objects.

The Builder Pattern separates the construction of a complex object from its representation, so that several different representations can be created depending on the needs of the program.

The Prototype Pattern starts with an initialized and instantiated class and copies or clones it to make new instances rather than creating new instances.

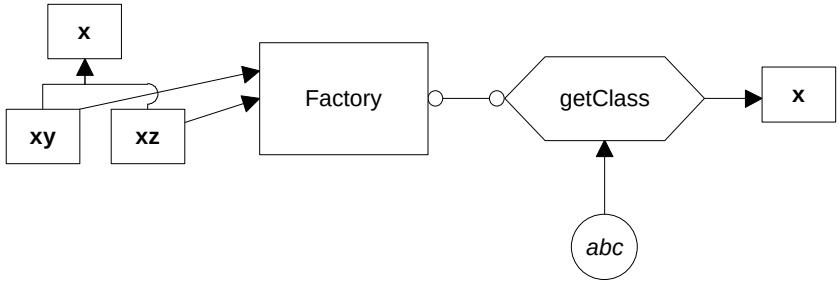
The Singleton Pattern is a class of which there can be no more than one instance. It provides a single global point of access to that instance.

THE FACTORY PATTERN

One type of pattern that we see again and again in OO programs is the Factory pattern or class. A Factory pattern is one that returns an instance of one of several possible classes depending on the data provided to it. Usually all of the classes it returns have a common parent class and common methods, but each of them performs a task differently and is optimized for different kinds of data.

How a Factory Works

To understand a Factory pattern, let's look at the Factory diagram below.



In this figure, **x** is a base class and classes **xy** and **xz** are derived from it. The Factory is a class that decides which of these subclasses to return depending on the arguments you give it. On the right, we define a *getClass* method to be one that passes in some value *abc*, and that returns some instance of the class **x**. Which one it returns doesn't matter to the programmer since they all have the same methods, but different implementations. How it decides which one to return is entirely up to the factory. It could be some very complex function but it is often quite simple.

Sample Code

Let's consider a simple case where we could use a Factory class. Suppose we have an entry form and we want to allow the user to enter his name either as "firstname lastname" or as "lastname, firstname". We'll make

the further simplifying assumption that we will always be able to decide the name order by whether there is a comma between the last and first name.

This is a pretty simple sort of decision to make, and you could make it with a simple *if* statement in a single class, but let's use it here to illustrate how a factory works and what it can produce. We'll start by defining a simple base class that takes a String and splits it (somehow) into two names:

```
class Namer {
//a simple class to take a string apart into two names
   protected String last;  //store last name here
   protected String first; //store first name here

   public String getFirst()    {
      return first;        //return first name
   }
   public String getLast()    {
      return last;         //return last name
   }
}
```

In this base class we don't actually do anything, but we do provide implementations of the *getFirst* and *getLast* methods. We'll store the split first and last names in the Strings *first* and *last*, and, since the derived classes will need access to these variables, we'll make them *protected*.

The Two Derived Classes

Now we can write two very simple derived classes that split the name into two parts in the constructor. In the FirstFirst class, we assume that everything before the last space is part of the first name:

```
class FirstFirst extends Namer {        //split first last
   public FirstFirst(String s)    {
      int i = s.lastIndexOf(" ");       //find sep space
      if (i > 0)  {
         //left is first name
         first = s.substring(0, i).trim();
          //right is last name
          last =s.substring(i+1).trim();
         }
       else  {
          first = “”;            // put all in last name
         last = s;               // if no space
         }
   }
}
```

And, in the LastFirst class, we assume that a comma delimits the last name. In both classes, we also provide error recovery in case the space or comma does not exist.

```
class LastFirst extends Namer {         //split last, first
   public LastFirst(String s)    {
      int i = s.indexOf(",");                 //find comma
      if (i > 0)  {
         //left is last name
          last = s.substring(0, i).trim();
          //right is first name
          first = s.substring(i + 1).trim();
         }
      else        {
         last = s;         // put all in last name
         first = "";             // if no comma
         }
   }
}
```

Building the Factory

Now our Factory class is extremely simple. We just test for the existence of a comma and then return an instance of one class or the other:

```
class NameFactory {
//returns an instance of LastFirst or FirstFirst
//depending on whether a comma is found
   public Namer getNamer(String entry)    {
      int i = entry.indexOf(",");   //comma determines name
order
      if (i>0)
         return new LastFirst(entry);   //return one class
      else
         return new FirstFirst(entry);  //or the other
   }
}
```

Using the Factory

Let’s see how we put this together.

We have constructed a simple Java user interface that allows you to enter the names in either order and see the two names separately displayed. You can see this program below.

You type in a name and then click on the **Compute** button, and the divided name appears in the text fields below. The crux of this program is the compute method that fetches the text, obtains an instance of a Namer class and displays the results.

In our constructor for the program, we initialize an instance of the factory class with

```
NameFactory nfactory = new NameFactory();
```

Then, when we process the button action event, we call the **computeName** method, which calls the **getNamer** factory method and then calls the first and last name methods of the class instance it returns:

```
private void computeName()   {
    //send the text to the factory and get a class back
     namer = nfactory.getNamer(entryField.getText());

    //compute the first and last names
    //using the returned class
     txFirstName.setText(namer.getFirst());
     txLastName.setText(namer.getLast());
  }
```

And that's the fundamental principle of Factory patterns. You create an abstraction which decides which of several possible classes to return and returns one. Then you call the methods of that class instance without ever

knowing which derived class you are actually using. This approach keeps the issues of data dependence separated from the classes' useful methods. You will find the complete code for Namer.java on the example CD-ROM.

Factory Patterns in Math Computation

Most people who use Factory patterns tend to think of them as tools for simplifying tangled programming classes. But it is perfectly possible to use them in programs that simply perform mathematical computations. For example, in the Fast Fourier Transform (FFT), you evaluate the following four equations repeatedly for a large number of point pairs over many passes through the array you are transforming. Because of the way the graphs of these computations are drawn, these equations constitute one instance of the FFT "butterfly." These are shown as Equations 1--4.

$$R_1' = R_1 + R_2\cos(y) - I_2\sin(y) \quad (1)$$

$$R_2' = R_1 - R_2\cos(y) + I_2\sin(y) \quad (2)$$

$$I_1' = I_1 + R_2\sin(y) + I_2\cos(y) \quad (3)$$

$$I_2' = I_1 - R_2\sin(y) - I_2\cos(y) \quad (4)$$

However, there are a number of times during each pass through the data where the angle *y* is zero. In this case, your complex math evaluation reduces to Equations (5-8):

$$R_1' = R_1 + R_2 \quad (5)$$

$$R_2' = R_1 - R_2 \quad (6)$$

$$I_1' = I_1 + I_2 \quad (7)$$

$$I_2' = I_1 - I_2 \quad (8)$$

So it is not unreasonable to package this computation in a couple of classes doing the simple or the expensive computation depending on the angle *y*. We'll start by creating a Complex class that allows us to manipulate real and imaginary number pairs:

```
class Complex {
   float real;
   float imag;
}
```

It also will have appropriate *get* and *set* functions.

Then we'll create our Butterfly class as an abstract class that we'll fill in with specific descendants:

```
abstract class Butterfly {
   float y;
   public Butterfly()    {
   }
   public Butterfly(float angle)    {
      y = angle;
   }
   abstract public void Execute(Complex x, Complex y);
}
```

Our two actual classes for carrying out the math are called *addButterfly* and *trigButterfly*. They implement the computations shown in equations (1--4) and (5--8) above.

```
class addButterfly extends Butterfly {
   float oldr1, oldi1;

   public addButterfly(float angle)    {
   }
   public void Execute(Complex xi, Complex xj)    {
     oldr1 = xi.getReal();
     oldi1 = xi.getImag();
     xi.setReal(oldr1 + xj.getReal()); //add and subtract
     xj.setReal(oldr1 - xj.getReal());
     xi.setImag(oldi1 + xj.getImag());
     xj.setImag(oldi1 - xj.getImag());
   }
}
```

and for the trigonometic version:

```
class trigButterfly extends Butterfly {
   float y;
   float oldr1, oldi1;
   float cosy, siny;
   float r2cosy, r2siny, i2cosy, i2siny;

   public trigButterfly(float angle)    {
      y = angle;
      cosy = (float) Math.cos(y);//precompute sine and cosine
      siny = (float)Math.sin(y);

   }
   public void Execute(Complex xi, Complex xj)    {
      oldr1 = xi.getReal();      //multiply by cos and sin
      oldi1 = xi.getImag();
      r2cosy = xj.getReal() * cosy;
      r2siny = xj.getReal() * siny;
      i2cosy = xj.getImag()*cosy;
```

```
      i2siny = xj.getImag()*siny;
      xi.setReal(oldr1 + r2cosy +i2siny);      //store sums
      xi.setImag(oldi1 - r2siny +i2cosy);
      xj.setReal(oldr1 - r2cosy - i2siny);
      xj.setImag(oldi1 + r2siny - i2cosy);
   }
}
```

Finally, we can make a simple factory class that decides which class instance to return. Since we are making Butterflies, we'll call our Factory a Cocoon:

```
class Cocoon {
   public Butterfly getButterfly(float y)    {
      if (y !=0)
         return new trigButterfly(y);   //get multiply class
      else
         return new addButterfly(y);    //get add/sub class
   }
}
```

You will find the complete FFT.java program on the example CDROM.

When to Use a Factory Pattern

You should consider using a Factory pattern when

- A class can't anticipate which kind of class of objects it must create.
- A class uses its subclasses to specify which objects it creates.
- You want to localize the knowledge of which class gets created.

There are several similar variations on the factory pattern to recognize.

1. The base class is abstract and the pattern must return a complete working class.
2. The base class contains default methods and is only subclassed for cases where the default methods are insufficient.
3. Parameters are passed to the factory telling it which of several class types to return. In this case the classes may share the same method names but may do something quite different.

Thought Questions

1. Consider a personal checkbook management program like Quicken. It manages several bank accounts and investments and can handle your bill paying. Where could you use a Factory pattern in designing a program like that?
2. Suppose are writing a program to assist homeowners in designing additions to their houses. What objects might a Factory be used to produce?

THE ABSTRACT FACTORY PATTERN

The Abstract Factory pattern is one level of abstraction higher than the factory pattern. You can use this pattern when you want to return one of several related classes of objects, each of which can return several different objects on request. In other words, the Abstract Factory is a factory object that returns one of several factories.

One classic application of the abstract factory is the case where your system needs to support multiple "look-and-feel" user interfaces, such as Windows-9x, Motif or Macintosh. You tell the factory that you want your program to look like Windows and it returns a GUI factory which returns Windows-like objects. Then when you request specific objects such as buttons, check boxes and windows, the GUI factory returns Windows instances of these visual interface components.

In Java 1.2 the pluggable look-and-feel classes accomplish this at the system level so that instances of the visual interface components are returned correctly once the type of look-and-feel is selected by the program. Here we find the name of the current windowing system and then tell the PLAF abstract factory to generate the correct objects for us.

```
String laf = UIManager.getSystemLookAndFeelClassName();
try {
UIManager.setLookAndFeel(laf);
 }
catch (UnsupportedLookAndFeelException exc)
  {System.err.println("UnsupportedL&F: " + laf);}
catch (Exception exc)
  {System.err.println("Error loading " + laf);
   }
```

A GardenMaker Factory

Let's consider a simple example where you might want to use the abstract factory at the application level.

Suppose you are writing a program to plan the layout of gardens. These could be annual gardens, vegetable gardens or perennial gardens. However, no matter which kind of garden you are planning, you want to ask the same questions:

1. What are good border plants?

2. What are good center plants?
3. What plants do well in partial shade?

…and probably many other plant questions that we'll omit in this simple example.

We want a base Garden class that can answer these questions:

```
public abstract class Garden {
   public abstract Plant getCenter();
   public abstract Plant getBorder();
   public abstract Plant getShade();
}
```

where our simple Plant object just contains and returns the plant name:

```
public class Plant {
String name;
  public Plant(String pname)   {
    name = pname;     //save name
  }
  public String getName()   {
     return name;
  }
}
```

Now in a real system, each type of garden would probably consult an elaborate database of plant information. In our simple example we'll return one kind of each plant. So, for example, for the vegetable garden we simply write

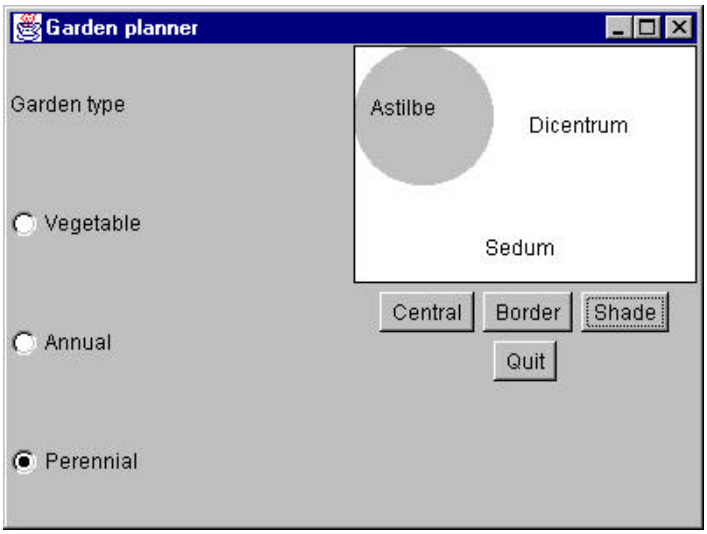
```
public class VegieGarden extends Garden {
   public Plant getShade()    {
      return new Plant("Broccoli");
   }
 public Plant getCenter()  {
    return new Plant("Corn");
 }
 public Plant getBorder()  {
    return new Plant("Peas");
 }
}
```

Now we have a series of Garden objects, each of which returns one of several Plant objects. We can easily construct our abstract factory to return one of these Garden objects based on the string it is given as an argument:

```
class GardenMaker
{
 //Abstract Factory which returns one of three gardens
```

```
   private Garden gd;

   public Garden getGarden(String gtype)
   {
      gd = new VegieGarden();   //default
      if(gtype.equals("Perennial"))
           gd = new PerennialGarden();
      if(gtype.equals("Annual"))
           gd = new AnnualGarden();
      return gd;
   }

}
```

This simple factory system can be used along with a more complex user interface to select the garden and begin planning it as shown below:



How the User Interface Works

This simple interface consists of two parts: the left side, that selects the garden type and the right side, that selects the plant category. When you click on one of the garden types, this actuates the MakeGarden Abstract Factory. This returns a type of garden that depends on the name of the text of the radio button caption.

```
public void itemStateChanged(ItemEvent e)
   {
      Checkbox ck = (Checkbox)e.getSource();
//get a garden type based on label of radio button
      garden = new GardenMaker().getGarden(ck.getLabel());
```

```
// Clear names of plants in display
      shadePlant=""; centerPlant=""; borderPlant = "";
      gardenPlot.repaint();             //display empty garden
   }
```

Then when a user clicks on one of the plant type buttons, the plant type is returned and the name of that plant displayed:

```
public void actionPerformed(ActionEvent e)    {
      Object obj = e.getSource();//get button type
      if(obj == Center)          //and choose plant type
         setCenter();
      if(obj == Border)
         setBorder();
      if(obj == Shade)
         setShade();
      if(obj == Quit)
         System.exit(0);
   }
   //---------------------------------
   private void setCenter()    {
      if (garden != null)
             centerPlant = garden.getCenter().getName();
      gardenPlot.repaint();
   }
   private void setBorder()    {
      if (garden != null)
             borderPlant = garden.getBorder().getName();
      gardenPlot.repaint();
   }
   private void setShade()    {
      if (garden != null)
             shadePlant = garden.getShade().getName();
      gardenPlot.repaint();
   }
```

The key to displaying the plant names is the garden plot panel, where they are drawn.

```
class GardenPanel extends Panel
 {
   public void paint (Graphics g)
   {
      //get panel size
       Dimension sz = getSize();
       //draw tree shadow
      g.setColor(Color.lightGray);
      g.fillArc( 0, 0, 80, 80,0, 360);
//draw plant names, some may be blank strings
      g.setColor(Color.black);
      g.drawRect(0,0, sz.width-1, sz.height-1);
      g.drawString(centerPlant, 100, 50);
```

```
      g.drawString( borderPlant, 75, 120);
      g.drawString(shadePlant, 10, 40);
   }
 }
}
```

You will find the complete code for Gardene.java on the example CDROM.

Consequences of Abstract Factory

One of the main purposes of the Abstract Factory is that it isolates the concrete classes that are generated. The actual class names of these classes are hidden in the factory and need not be known at the client level at all.

Because of the isolation of classes, you can change or interchange these product class families freely. Further, since you generate only one kind of concrete class, this system keeps you for inadvertently using classes from different families of products. However, it is some effort to add new class families, since you need to define new, unambiguous conditions that cause such a new family of classes to be returned.

While all of the classes that the Abstract Factory generates have the same base class, there is nothing to prevent some derived classes from having additional methods that differ from the methods of other classes. For example a BonsaiGarden class might have a Height or WateringFrequency method that is not present in other classes. This presents the same problem as occur in any derived classes-- you don't know whether you can call a class method unless you know whether the derived class is one that allows those methods. This problem has the same two solutions as in any similar case: you can either define all of the methods in the base class, even if they don't always have a actual function, or you can test to see which kind of class you have:

```
if (gard instanceof BonsaiGarden)
      int h = gard.Height();
```

Thought Questions

If you are writing a program to track investments, such as stocks, bonds, metal futures, derivatives, etc., how might you use an Abstract Factory?

THE SINGLETON PATTERN

The Singleton pattern is grouped with the other Creational patterns, although it is to some extent a "non-creational" pattern. There are any number of cases in programming where you need to make sure that there can be one and only one instance of a class. For example, your system can have only one window manager or print spooler, or a single point of access to a database engine.

The easiest way to make a class that can have only one instance is to embed a `static` variable inside the class that we set on the first instance and check for each time we enter the constructor. A static variable is one for which there is only one instance, no matter how many instances there are of the class.

```
static boolean instance_flag = false;
```

The problem is how to find out whether creating an instance was successful or not, since constructors do not return values. One way would be to call a method that checks for the success of creation, and which simply returns some value derived from the static variable. This is inelegant and prone to error, however, because there is nothing to keep you from creating many instances of such non-functional classes and forgetting to check for this error condition.

A better way is to create a class that throws an Exception when it is instantiated more than once. Let's create our own exception class for this case:

```
class SingletonException extends RuntimeException
{
   //new exception type for singleton classes
   public SingletonException()
   {
      super();
   }
//-----------------------------------------------
   public SingletonException(String s)
   {
      super(s);
   }
}
```

Note that other than calling its parent classes through the `super()`method, this new exception type doesn't do anything in particular. However, it is convenient to have our own named exception type so that the compiler will warn us of the type of exception we must catch when we attempt to create an instance of PrintSpooler.

Throwing the Exception

Let's write the skeleton of our PrintSpooler class; we'll omit all of the printing methods and just concentrate on correctly implementing the Singleton pattern:

```
class PrintSpooler
{
   //this is a prototype for a printer-spooler class
   //such that only one instance can ever exist
   static boolean
             instance_flag=false; //true if 1 instance

   public PrintSpooler() throws SingletonException
   {
   if (instance_flag)
      throw new SingletonException("Only one spooler allowed");
   else
      instance_flag = true;   //set flag for 1 instance
      System.out.println("spooler opened");
   }
   //----------------------------------------
   public void finalize()
   {
      instance_flag = false;     //clear if destroyed
   }
}
```

Creating an Instance of the Class

Now that we've created our simple Singleton pattern in the PrintSpooler class, let's see how we use it. Remember that we must enclose every method that may throw an exception in a `try - catch` block.

```
public class singleSpooler
{
   static public void main(String argv[])
   {
      PrintSpooler pr1, pr2;

     //open one spooler--this should always work
      System.out.println("Opening one spooler");
      try{
      pr1 = new PrintSpooler();
      }
      catch (SingletonException e)
      {System.out.println(e.getMessage());}

      //try to open another spooler --should fail
      System.out.println("Opening two spoolers");
```

```
      try{
      pr2 = new PrintSpooler();
      }
      catch (SingletonException e)
      {System.out.println(e.getMessage());}
   }
}
```

Then, if we execute this program, we get the following results:

```
Opening one spooler
printer opened
Opening two spoolers
Only one spooler allowed
```

where the last line indicates than an exception was thrown as expected. You will find the complete source of this program on the example CD-ROM as singleSpooler.java.

Static Classes as Singleton Patterns

There already is a kind of Singleton class in the standard Java class libraries: the Math class. This is a class that is declared *final* and all methods are declared *static*, meaning that the class cannot be extended. The purpose of the Math class is to wrap a number of common mathematical functions such as *sin* and *log* in a class-like structure, since the Java language does not support functions that are not methods in a class.

You can use the same approach to a Singleton pattern, making it a *final* class. You can't create *any* instance of classes like Math, and can only call the static methods directly in the existing final class.

```
final class PrintSpooler
{
 //a static class implementation of Singleton pattern
 static public void print(String s)
 {
 System.out.println(s);
 }
}
//==============================
public class staticPrint
{
   public static void main(String argv[])
   {
      Printer.print("here it is");
   }
}
```

One advantage of the final class approach is that you don't have to wrap things in awkward try blocks. The disadvantage is that if you would like to drop the restrictions of Singleton status, this is easier to do in the exception style class structure. We'd have a lot of reprogramming to do to make the static approach allow multiple instances.

Creating Singleton Using a Static Method

Another approach, suggested by *Design Patterns*, is to create Singletons using a static method to issue and keep track of instances. To prevent instantiating the class more than once, we make the constructor private so an instance can only be created from within the static method of the class.

```
class iSpooler
{
   //this is a prototype for a printer-spooler class
   //such that only one instance can ever exist
   static boolean instance_flag = false; //true if 1 instance

  //the constructor is privatized-
  //but need not have any content
   private iSpooler()    {  }
//static Instance method returns one instance or null
   static public iSpooler Instance()
   {
      if (! instance_flag)
      {
         instance_flag = true;
         return new iSpooler();  //only callable from within
      }
      else
         return null;      //return no further instances
   }
   //----------------------------------------
   public void finalize()
   {
      instance_flag = false;
   }
}
```

One major advantage to this approach is that you don't have to worry about exception handling if the singleton already exists-- you simply get a null return from the Instance method:

```
      iSpooler pr1, pr2;
      //open one spooler--this should always work
      System.out.println("Opening one spooler");
      pr1 = iSpooler.Instance();
```

```
if(pr1 != null)
   System.out.println("got 1 spooler");
//try to open another spooler --should fail
System.out.println("Opening two spoolers");

pr2 = iSpooler.Instance();
if(pr2 == null)
   System.out.println("no instance available");
```

And, should you try to create instances of the iSpooler class directly, this will fail at compile time because the constructor has been declared as private.

```
//fails at compile time because constructor is privatized
 iSpooler pr3 = new iSpooler();
```

Finding the Singletons in a Large Program

In a large, complex program it may not be simple to discover where in the code a Singleton has been instantiated. Remember that in Java, global variables do not really exist, so you can't save these Singletons conveniently in a single place.

One solution is to create such singletons at the beginning of the program and pass them as arguments to the major classes that might need to use them.

```
pr1 = iSpooler.Instance();
Customers cust = new Customers(pr1);
```

A more elaborate solution could be to create a registry of all the Singleton classes in the program and make the registry generally available. Each time a Singleton instantiates itself, it notes that in the Registry. Then any part of the program can ask for the instance of any singleton using an identifying string and get back that instance variable.

The disadvantage of the registry approach is that type checking may be reduced, since the table of singletons in the registry probably keeps all of the singletons as Objects, for example in a Hashtable object. And, of course, the registry itself is probably a Singleton and must be passed to all parts of the program using the constructor or various set functions.

Other Consequences of the Singleton Pattern

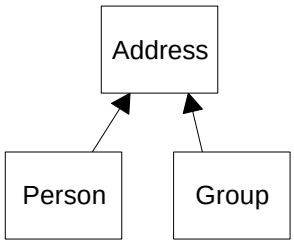
1. It can be difficult to subclass a Singleton, since this can only work if the base Singleton class has not yet been instantiated.

2. You can easily change a Singleton to allow a small number of instances where this is allowable and meaningful.

THE BUILDER PATTERN

We have already seen that the Factory Pattern returns one of several different subclasses depending on the data passed to in arguments to the creation methods. But suppose we don't want just a computing algorithm, but a whole different user interface depending on the data we need to display. A typical example might be your E-mail address book. You probably have both people and groups of people in your address book, and you would expect the display for the address book to change so that the People screen has places for first and last name, company, E-mail address and phone number.

On the other hand if you were displaying a group address page, you'd like to see the name of the group, its purpose, and a list of members and their E-mail addresses. You click on a person and get one display and on a group and get the other display. Let's assume that all E-mail addresses are kept in an object called an Address and that people and groups are derived from this base class as shown below:



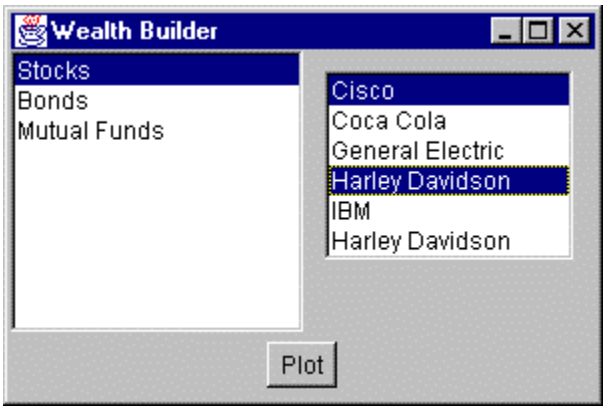
Depending on which type of Address object we click on, we'd like to see a somewhat different display of that object's properties. This is a little more than just a Factory pattern, because we aren't returning objects which are simple descendents of a base display object, but totally different user interfaces made up of different combinations of display objects. The *Builder Pattern* assembles a number of objects, such as display widgets, in various ways depending on the data. Furthermore, since Java is one of the few languages where you can cleanly separate the data from the display methods into simple objects, Java is the ideal language to implement Builder patterns.

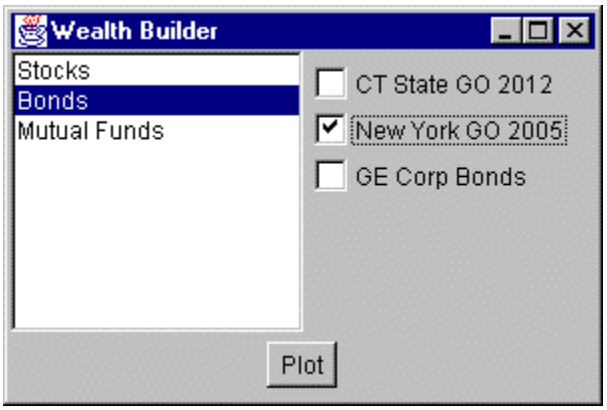
An Investment Tracker

Let's consider a somewhat simpler case where it would be useful to have a class build our UI for us. Suppose we are going to write a program to keep track of the performance of our investments. We might have stocks, bonds and mutual funds, and we'd like to display a list of our holdings in each category so we can select one or more of the investments and plot their comparative performance.

Even though we can't predict in advance how many of each kind of investment we might own at any given time, we'd like to have a display that is easy to use for either a large number of funds (such as stocks) or a small number of funds (such as mutual funds). In each case, we want some sort of a multiple-choice display so that we can select one or more funds to plot. If there is a large number of funds, we'll use a multi-choice list box and if there are 3 or fewer funds, we'll use a set of check boxes. We want our Builder class to generate an interface that depends on the number of items to be displayed, and yet have the same methods for returning the results.

Our displays are shown below. The first display contains a large number of stocks and the second a small number of bonds.





Now, let's consider how we can build the interface to carry out this variable display. We'll start with a *multiChoice* abstract class that defines the methods we need to implement:

```
abstract class multiChoice
{
   //This is the abstract base class
   //that the listbox and checkbox choice panels
   //are derived from
   Vector choices;    //array of labels
//-----------------------------------------
   public multiChoice(Vector choiceList)
   {
     choices = choiceList;    //save list
   }
   //to be implemented in derived classes
 abstract public Panel getUI();  //return a Panel of components
 abstract public String[] getSelected(); //get list of items
 abstract public void clearAll();       //clear selections
}
```

The *getUI* method returns a Panel container with a multiple-choice display. The two displays we're using here -- a checkbox panel or a list box panel -- are derived from this abstract class:

```
class listboxChoice extends multiChoice
```

or

```
class checkBoxChoice extends multiChoice
```

Then we create a simple Factory class that decides which of these two classes to return:

```
class choiceFactory
{
```

```
   multiChoice ui;
   //This class returns a Panel containing
   //a set of choices displayed by one of
   //several UI methods.
   public multiChoice getChoiceUI(Vector choices)
   {
      if(choices.size() <=3)
        //return a panel of checkboxes
        ui = new checkBoxChoice(choices);
      else
        //return a multi-select list box panel
        ui = new listboxChoice(choices);
      return ui;
   }
}
```

In the language of *Design Patterns*, this factory class is called the Director, and the actual classes derived from *multiChoice* are each Builders.

Calling the Builders

Since we're going to need one or more builders, we might have called our main class Architect or Contractor, but since we're dealing with lists of investments, we'll just call it WealthBuilder. In this main class, we create the user interface, consisting of a BorderLayout with the center divided into a 1 x 2 GridLayout. The left part contains our list of investment types and the right an empty panel that we'll fill depending on which kind of investments are selected.

```
public wealthBuilder()
    {
       super("Wealth Builder");     //frame title bar
       setGUI();                    //set up display
       buildStockLists();           //create stock lists
        choiceFactory cfact;        //the factory
    }
    //---------------------------------
    private void setGUI()
    {
       setLayout(new BorderLayout());
       Panel p = new Panel();
       add("Center", p);
  //center contains left and right panels
       p.setLayout(new GridLayout(1,2));

  //left is list of stocks
       stockList= new List(10);
       stockList.addItemListener(this);
       p.add(stockList);
```

```
        stockList.add("Stocks");
        stockList.add("Bonds");
        stockList.add("Mutual Funds");
        stockList.addItemListener(this);

    //Plot button along bottom of display
        Panel p1 = new Panel();
        p1.setBackground(Color.lightGray);
        add("South", p1);
        Plot = new Button("Plot");
        Plot.setEnabled(false);    //disabled until stock picked
        Plot.addActionListener(this);
        p1.add(Plot);

   //right is empty at first
        choicePanel = new Panel();
        choicePanel.setBackground(Color.lightGray);
        p.add(choicePanel);

        w = new Winder();  //intercepts WindowClosing
        addWindowListener(w);
        setBounds(100, 100, 300, 200);
        setVisible(true);
     }
```

In this simple pro gram, we keep our three lists of investments in three Vectors called Stocks, Bonds and Mutuals. We load them with arbitrary values as part of program initialization:

```
       Mutuals = new Vector();
        Mutuals.addElement("Fidelity Magellan");
        Mutuals.addElement("T Rowe Price");
        Mutuals.addElement("Vanguard PrimeCap");
        Mutuals.addElement("Lindner Fund");
```

and so forth. In a real system, we'd probably read them in from a file or database. Then, when the user clicks on one of the three investment types in the left list box, we pass the equivalent vector to our Factory, which returns one of the builders:

```
private void stockList_Click()
{
   Vector v = null;
   int index = stockList.getSelectedIndex();
   choicePanel.removeAll();  //remove previous ui panel

   //this just switches among 3 different Vectors
   //and passes the one you select to the Builder pattern
   switch(index)
   {
```

```
   case 0:
      v = Stocks;  break;
   case 1:
      v = Bonds;   break;
   case 2:
      v = Mutuals;
   }
   mchoice = cfact.getChoiceUI(v);      //get one of the UIs
   choicePanel.add(mchoice.getUI());    //insert in right panel
   choicePanel.validate();              //re-layout and display
   Plot.setEnabled(true);               //allow plots
}
```

We do save the multiChoice panel the factory creates in the **mchoice** variable so we can pass it to the Plot dialog.

The List Box Builder

The simpler of the two builders is the List box builder. It returns a panel containing a list box showing the list of investments.

```
class listboxChoice extends multiChoice
{
   List list;       //investment list goes here
//-----------------------------------------
   public listboxChoice(Vector choices)
   {
      super(choices);
   }
//-----------------------------------------
   public Panel getUI()
   {
      //create a panel containing a list box
      Panel p = new Panel();
      list = new List(choices.size());  //list box
      list.setMultipleMode(true);       //multiple
      p.add(list);
//add investments into list box
      for (int i=0; i< choices.size(); i++)
         list.addItem((String)choices.elementAt(i));
     return p;            //return the panel
  }
```

The other important method is the *getSelected* method that returns a String array of the investments the user selects:

```
public String[] getSelected()
   {
      int count =0;
      //count the selected listbox lines
```

```
      for (int i=0; i < list.getItemCount(); i++ )
         {
         if (list.isIndexSelected(i))
            count++;
         }
      //create a string array big enough for those selected
      String[] slist = new String[count];

      //copy list elements into string array
      int j = 0;
      for (int i=0; i < list.getItemCount(); i++ )
         {
         if (list.isIndexSelected(i))
            slist[j++] = list.getItem(i);
         }
      return(slist);
   }
```

The Checkbox Builder

The Checkbox builder is even simpler. Here we need to find out how many elements are to be displayed and create a horizontal grid of that many divisions. Then we insert a check box in each grid line:

```
public checkBoxChoice(Vector choices)
   {
      super(choices);
      count = 0;
      p = new Panel();
   }
//-----------------------------------------
   public Panel getUI()
   {
      String s;

      //create a grid layout 1 column by n rows
      p.setLayout(new GridLayout(choices.size(), 1));

      //and add labeled check boxes to it
      for (int i=0; i< choices.size(); i++)
         {
         s =(String)choices.elementAt(i);
         p.add(new Checkbox(s));
         count++;
         }
      return p;
   }
```

The *getSelected* method is analogous to the one we showed above, and is included in the example code on the CDROM.

Consequences of the Builder Pattern

1. A Builder lets you vary the internal representation of the product it builds. It also hides the details of how the product is assembled.
2. Each specific builder is independent of the others and of the rest of the program. This improves modularity and makes the addition of other builders relatively simple.
3. Because each builder constructs the final product step-by-step, depending on the data, you have more control over each final product that a Builder constructs.

A Builder pattern is somewhat like an Abstract Factory pattern in that both return classes made up of a number of methods and objects. The main difference is that while the Abstract Factory returns a family of related classes, the Builder constructs a complex object step by step depending on the data presented to it.

Thought Questions

1. Some word-processing and graphics programs construct menus dynamically based on the context of the data being displayed. How could you use a Builder effectively here?
2. Not all Builders must construct visual objects. What might you use a Builder to construct in the personal finance industry? Suppose you were scoring a track meet, made up of 5-6 different events? Can you use a Builder there?

THE PROTOTYPE PATTERN

The Protoype pattern is used when creating an instance of a class is very time-consuming or complex in some way. Then, rather than creating more instances, you make copies of the original instance, modifying them as appropriate.

Prototypes can also be used whenever you need classes that differ only in the type of processing they offer, for example in parsing of strings representing numbers in different radixes. In this sense, the prototype is nearly the same as the Examplar pattern described by Coplien [1992].

Let's consider the case of an extensive database where you need to make a number of queries to construct an answer. Once you have this answer as a table or ResultSet, you might like to manipulate it to produce other answers without issuing additional queries.

In a case like one we have been working on, we'll consider a database of a large number of swimmers in a league or statewide organization. Each swimmer swims several strokes and distances throughout a season. The "best times" for swimmers are tabulated by age group, and many swimmers will have birthdays and fall into new age groups within a single season. Thus the query to determine which swimmers did the best in their age group that season is dependent on the date of each meet and on each swimmer's birthday. The computational cost of assembling this table of times is therefore fairly high.

Once we have a class containing this table, sorted by sex, we could imagine wanting to examine this information sorted just by time, or by actual age rather than by age group. It would not be sensible to recompute these data, and we don't want to destroy the original data order, so some sort of copy of the data object is desirable.

Cloning in Java

You can make a copy of any Java object using the **clone** method.

```
Jobj j1 = (Jobj)j0.clone();
```

The clone method always returns an object of type Object. You must cast it to the actual type of the object you are cloning. There are three other significant restrictions on the clone method:

1. It is a protected method and can only be called from within the same class or the module that contains that class.
2. You can only clone objects which are declared to implement the Cloneable interface.
3. Objects that cannot be cloned throw the CloneNotSupported Exception.

This suggests packaging the actual clone method inside the class where it can access the real clone method:

```
public class SwimData implements Cloneable
{
  public Object clone()
  {
     try{
      return super.clone();
     }
     catch(Exception e)
            {System.out.println(e.getMessage());
     return null;
     }
  }
}
```

This also has the advantage of encapsulating the try-catch block inside the public clone method. Note that if you declare this public method to have the same name "clone," it must be of type Object, since the internal protected method has that signature. You could, however, change the name and do the typecasting within the method instead of forcing it onto the user:

```
public SwimData cloneMe()
  {
     try{
      return (SwimData)super.clone();
     }
     catch(Exception e)
            {System.out.println(e.getMessage());
     return null;
     }
  }
```

You can also make special cloning procedures that change the data or processing methods in the cloned class, based on arguments you pass to the clone method. In this case, method names such as *make* are probably more descriptive and suitable.

Using the Prototype

Now let's write a simple program that reads data from a database and then clones the resulting object. In our example program, SwimInfo, we just read these data from a file, but the original data were derived from a large database as we discussed above.

Then we create a class called Swimmer that holds one name, club name, sex and time

```
class Swimmer
{  String name;
   int age;
   String club;
   float time;
   boolean female;
```

and a class called SwimData that maintains a vector of the Swimmers we read in from the database.

```
public class SwimData implements Cloneable
{
  Vector swimmers;
  public SwimData(String filename)
  {
     String s = "";
     swimmers = new Vector();
     //open data file
     InputFile f = new InputFile(filename);
     s= f.readLine();   //read in and parse each line
     while(s != null)
     {
        swimmers.addElement(new Swimmer(s));
        s= f.readLine();
     }
     f.close();
  }
```

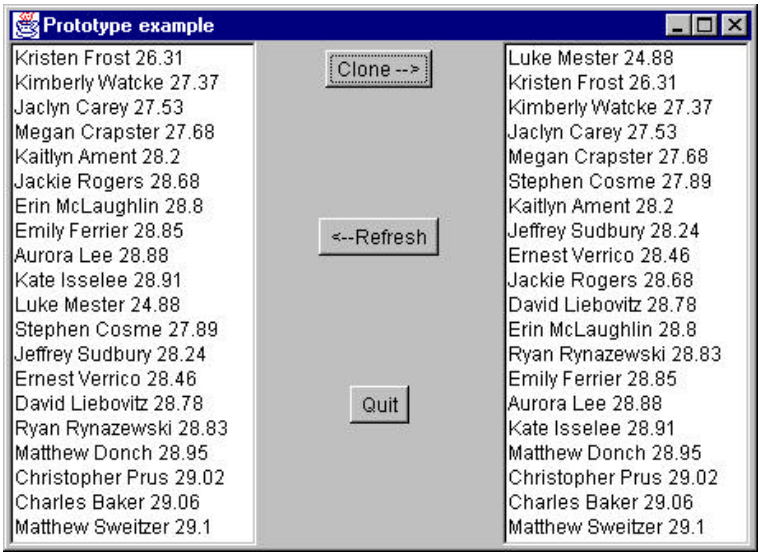
We also provide a *getSwimmer* method in SwimData and *getName*, *getAge* and *getTime* methods in the Swimmer class. Once we've read the data into SwimInfo, we can display it in a list box.

```
      swList.removeAll();     //clear list
      for (int i = 0; i < sdata.size(); i++)
      {
       sw = sdata.getSwimmer(i);
       swList.addItem(sw.getName()+" "+sw.getTime());
      }
```

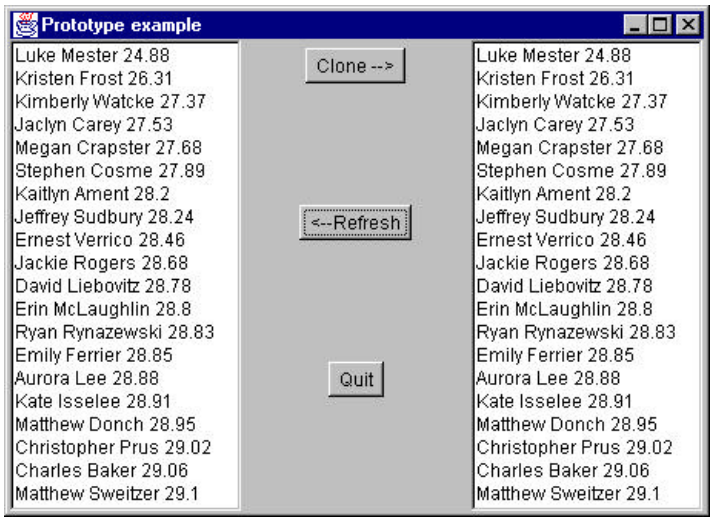
Then, when the user clicks on the Clone button, we'll clone this class and sort the data differently in the new class. Again, we clone the data because creating a new class instance would be much slower, and we want to keep the data in both forms.

```
sxdata = (SwimData)sdata.clone();
sxdata.sortByTime();   //re-sort
cloneList.removeAll(); //clear list

//now display sorted values from clone
for(int i=0; i< sxdata.size(); i++)
  {
   sw = sxdata.getSwimmer(i);
   cloneList.addItem(sw.getName()+" "+sw.getTime());
 }
```

In the original class, the names are sorted by sex and then by time, while in the cloned class, they are sorted only by time. In the figure below, we see the simple user interface that allows us to display the original data on the left and the sorted data in the cloned class on the right:



The left-hand list box is loaded when the program starts and the right-hand list box is loaded when you click on the **Clone** button. Now, let's click on the **Refresh** button to reload the left-hand list box from the original data.



Why have the names in the left-hand list box also been re-sorted?. This occurs in Java because the clone method is a *shallow copy* of the original class. In other words, the references to the data objects are copies, but they refer to the same underlying data. Thus, any operation we perform on the copied data will also occur on the original data in the Prototype class.

In some cases, this shallow copy may be acceptable, but if you want to make a deep copy of the data, there is a clever trick using the serializable interface. A class is said to be *serializable* if you can write it out as a stream of bytes and read those bytes back in to reconstruct the class. This is how Java remote method invocation (RMI) is implemented. However, if we declare both the Swimmer and SwimData classes as Serializable,

```
public class SwimData
   implements Cloneable, Serializable

class Swimmer implements Serializable
```

we can write the bytes to an output stream and reread them to create a complete data copy of that instance of a class:

```
public Object deepClone()
  {
    try{
     ByteArrayOutputStream b = new ByteArrayOutputStream();
     ObjectOutputStream out = new ObjectOutputStream(b);
     out.writeObject(this);
     ByteArrayInputStream bIn = new
              ByteArrayInputStream(b.toByteArray());
```

```
     ObjectInputStream oi = new ObjectInputStream(bIn);
     return (oi.readObject());
    }
    catch (Exception e)
    {  System.out.println("exception:"+e.getMessage());
       return null;
    }
  }
```

This deepClone method allows us to copy an instance of a class of any complexity and have data that is completely independent between the two copies. The program SwimInfo on the accompanying CD-ROM contains the complete code for this example, showing both cloning methods.

Consequences of the Prototype Pattern

Using the Prototype pattern, you can add and remove classes at run time by cloning them as needed. You can revise the internal data representation of a class at run time based on program conditions. You can also specify new objects at run time without creating a proliferation of classes and inheritance structures.

One difficulty in implementing the Prototype pattern in Java is that if the classes already exist, you may not be able to change them to add the required clone or deepClone methods. The deepClone method can be particularly difficult if all of the class objects contained in a class cannot be declared to implement Serializable. In addition, classes that have circular references to other classes cannot really be cloned.

Like the registry of Singletons discussed above, you can also create a registry of Prototype classes which can be cloned and ask the registry object for a list of possible prototypes. You may be able to clone an existing class rather than writing one from scratch.

Note that every class that you might use as a prototype must itself be instantiated (perhaps at some expense) in order for you to use a Prototype Registry. This can be a performance drawback.

Finally, the idea of having prototype classes to copy implies that you have sufficient access to the data or methods in these classes to change them after cloning. This may require adding data access methods to these prototype classes so that you can modify the data once you have cloned the class.

SUMMARY OF CREATIONAL PATTERNS

- **The Factory Pattern** is used to choose and return an instance of a class from a number of similar classes based on data you provide to the factory.
- **The Abstract Factory Pattern** is used to return one of several groups of classes. In some cases it actually returns a Factory for that group of classes.
- **The Builder Pattern** assembles a number of objects to make a new object, based on the data with which it is presented. Frequently, the choice of which way the objects are assembled is achieved using a Factory.
- **The Prototype Pattern** copies or clones an existing class rather than creating a new instance when creating new instances is more expensive.
- **The Singleton Pattern** is a pattern that insures there is one and only one instance of an object, and that it is possible to obtain global access to that one instance.

The Java Foundation Classes

The Java Foundation Classes (JFC or “Swing”) are a complete set of light-weight user interface components that enhance, extend and to a large degree replace the AWT components. In addition to the buttons, lists, tables and trees in the JFC, you will also find a pluggable look-and-feel that allows the components to take on the appearance of several popular windowing systems, as well as its own look and feel. The JFC actually uses a few common design patterns, and we will be using the JFC for most of the examples in this book. Thus, we are taking a short detour to outline how the JFC components work before going on to more patterns.

We should note at the outset, that this package was called “Swing” during development and it was intended that it be referred to as “JFC” upon release. However, the nickname has stuck, and this has led to the Java programmer’s explanation that “it’s spelled JFC, but it’s pronounced Swing.”

Installing and Using the JFC

All of the Swing classes are in 3 jar files, called swing.jar, swingall.jar and windows.jar. If you are using Java 1.1, you can download the Swing classes from java.sun.com site and install them by simply unzipping the downloaded file. It is important that your CLASSPATH variable contain the paths for these three jar files.

```
set CLASSPATH=.;d:\java11\lib\classes.zip;d:\swing\swing.jar;
d:\swing\windows.jar;d:\swing\swingall.jar;
```

All programs which are to make use of the JFC, must import the following files:

```
//swing classes
import com.sun.java.swing.*;
import com.sun.java.swing.event.*;
import com.sun.java.swing.border.*;
import com.sun.java.swing.text.*;
```

In the Java JDK 1.2, these change to “javax.swing”

```
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.border.*;
import javax.swing.text.*;
```

and so forth.

Ideas Behind Swing

The Swing components are referred to as “lightweight” components, because they don’t rely on native user-interface components. They are, in fact, 100% pure Java. Thus, a Swing JButton does not rely on a Windows button or a Motif button or a Mac button to implement its functionality. They also use fewer classes to achieve this interface than the previous heavier-weight awt classes. In addition, there are many more Swing user-interface components than there were awt components. Swing gives us image buttons, hover buttons, tooltips, tables, trees, splitter panels, customizable dialog boxes and quite a few other components.

Since Swing components create their look and feel completely within the Swing class hierarchy, you can have a pluggable look and feel to emulate Windows, Motif, Macintosh or the native Swing look.

In addition, Swing components make use of an architecture derived from the model-view-controller design pattern we discussed in the first chapter. The idea of this MVC pattern you recall, is to keep the data in a *model* class, display the data in a *view* class and vary the data and view using a *controller* class. We’ll see that this is exactly how the JList and Jtable handle their data.

The Swing Class Hierarchy

All Swing components inherit from the JComponent class. While JComponent is much like Component in its position in the hierarchy, JComponent is the level that provides the pluggable look and feel. It also provides

- Keystroke handling that works with nested components.
- A border property that defines both the border and the component’s insets.
- Tooltips that pop up when the mouse hovers over the component.
- Automatic scrolling of any component when placed in a scroller container.

Because of this ineraction with the user interface environement, Swing’s JComponent is actually more like the awt’s Canvas than its Component class.

WRITING A SIMPLE JFC PROGRAM

Getting started using the Swing classes is pretty simple. Application windows inherit from JFrame and applets inherit from JApplet. The only difference between Frame and JFrame is that you cannot add components or set the layout directly for JFrame. Instead, you must use the *getContentPane* method to obtain the container where you can add components and vary the layout.

```
getContentPane().setLayout(new BorderLayout());
JButton b = new Jbutton (“Hi”);
GetContentPane().add(b);          //add button to layout
```

This is sometimes a bit tedious to type each time, so we recommend creating a simple JPanel and adding it to the JFrame and then adding all the components to that panel.

```
JPanel jp = new JPanel();
getContentPane().add(jp);
JButton b = new JButton(“Hi”);
jp.add(b);
```

JPanels are containers much like the awt Panel object, except that they are automatically double buffered and repaint more quickly and smoothly.

Setting the Look and Feel

If you do nothing, Swing programs will start up in their own native look and feel rather than the Windows, Motif or Mac look. You must specifically set the look and feel in each program, using a simple method like the following:

```
private void setLF()
   {
   // Force to come up in the System L&F
       String laf = UIManager.getSystemLookAndFeelClassName();
       try {
       UIManager.setLookAndFeel(laf);
        }
       catch (UnsupportedLookAndFeelException exc)
         {System.err.println("Unsupported: " + laf);}
       catch (Exception exc)
         {System.err.println("Error loading " + laf);
          }
```

```
   }
```

The system that generates one of several look and feels and returns self-consistent classes of visual objects for a given look and feel is an example of an Abstract Factory pattern as we discussed in the previous chapter.

Setting the Window Close Box

Like the Frame component, the system exit procedure is *not* called automatically when a user clicks on the close box. In order to enable that behavior, you must add a WindowListener to the frame and catch the WindowClosing event. This can be done most effectively by subclassing the WindowAdapter class:

```
private void setCloseClick()
   {
      //create window listener to respond to window close click
      addWindowListener(new WindowAdapter()
       {
           public void windowClosing(WindowEvent e)
            {System.exit(0);}
           });
   }
```

Making a JxFrame Class

Since we must always set the look and feel and must always create a WindowAdapter to close the JFrame, we have created a JxFrame class which contains those two functions, and which calls them as part of initialization:

```
public class JxFrame extends JFrame
{
   public JxFrame(String title)
   {
      super(title);
      setCloseClick();
      setLF();
   }
}
```

The *setLF* and *setCloseClick* methods are included as well. It is this JxFrame class that we use in virtually all of our examples in this book, to avoid continually retyping the same code.

A Simple Two Button Program

Now with these fundamentals taken care of, we can write a simple program with two buttons in a frame.

One button switches the color of the background and the other causes the program to exit. We start by initializing our GUI and catching both button clicks in an *actionPerformed* method:

```
public class SimpleJFC extends JxFrame
   implements ActionListener
{
   JButton OK, Quit;       //these are the buttons
   JPanel jp;              //main panel
   Color color;            //background color
   //-----------------------------------------
   public SimpleJFC()    {
    super("Simple JFC Program");
    color = Color.yellow;  //start in yellow
    setGUI();
   }
   //-----------------------------------------
   private void setGUI()    {
    jp = new JPanel();           //central panel
    getContentPane().add(jp);

    //create and add buttons
    OK = new JButton("OK");
    Quit = new JButton("Quit");
    OK.addActionListener(this);
    Quit.addActionListener(this);
    jp.add(OK);
    jp.add(Quit);

    setSize(new Dimension(250,100));
    setVisible(true);
   }
   //-----------------------------------------
   public void actionPerformed(ActionEvent e)    {
      Object obj = e.getSource();
      if(obj == OK)
         switchColors();
      if(obj == Quit)
```

```
      System.exit(0);
   }
```

The only remaining part is the code that switches the background colors. This is, of course, extremely simple as well:

```
   private void switchColors()    {
      if(color == Color.green)
         color = Color.yellow;
      else
         color = Color.green;
      jp.setBackground(color);
      repaint();
   }
```

That's all there is to writing a basic JFC application. JFC applets are identical except for the applet's *init* routine replacing the constructor. Now let's look at some of the power of the more common JFC components.

More on JButtons

The JButton has several constructors to specify text, an icon or both:

```
JButton(String text);
JButton(Icon icon);
JButton(String text, Icon icon);
```

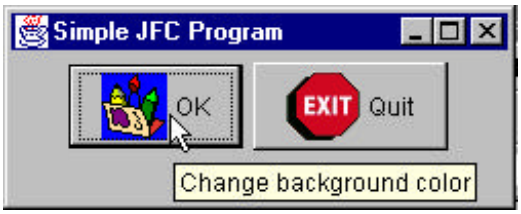
You can also set two other images to go with the button

```
setSelectedIcon(Icon icon);      //shown when clicked
setRolloverIcon(Icon icon);      //shown when mouse over
```

Finally, like all other JComponents, you can use *setTooltiptext* to set the text of a Tooltip to be displayed when the mouse hovers over the button. The code for implementing these small improvements is simply

```
OK = new JButton("OK",new ImageIcon("color.gif"));
OK.setRolloverIcon(new ImageIcon("overColor.gif"));
OK.setToolTipText("Change background color");
Quit = new JButton("Quit", new ImageIcon("exit.gif"));
Quit.setToolTipText("Exit from program");
```

The resulting application window is shown below.


Simple JFC Program
OK
EXIT
Quit
Change background color

BUTTONS AND TOOLBARS

Swing provides separate implementations of both the JRadioButton and the JCheckBox. A checkbox has two states and within a group of checkboxes, any number can be selected or deselected. Radio buttons should be grouped into a ButtonGroup object so that only one radio button of a group can be selected at a time.

Radio Buttons

Both radio buttons and check boxes can be instantiated with an image as well as a title and both can have rollover icons. The JCheckBox component is derived from the simpler JToggleButton object. JToggleButton is a button that can be switched between two states by clicking, but which stays in that new state (up or down) like a 2-state check box does. Further the JToggleButton can take on the exclusive aspects of a radio button by adding it to a ButtonGroup.

```
//create radio buttons in right panel
JRadioButton Rep, Dem, Flat;

right.add(Rep = new JRadioButton("Republicrat"));
right.add(Dem = new JRadioButton("Demmican"));
right.add(Flat = new JRadioButton("Flat Earth"));
ButtonGroup bgroup = new ButtonGroup();
bgroup.add(Rep);        //add to button group
bgroup.add(Dem);
bgroup.add(Flat);
```

If you neglect to add the radio buttons to a ButtonGroup, you can have several of them turned on at once. It is he ButtonGroup that assures that only one at a time can be turned on. The ButtonGroup object thus keeps track of the state of all the radio buttons in the group to enforce this only-one-on protocol. This is a clear example of the Mediator pattern we'll be discussing in the chapters ahead.

The JToolBar

JToolBar is a container bar for tool buttons of the type you see in many programs. Normally, the JDK documentation recommends that you add the JToolBar as the only component on one side of a Borderlayout typically the North side), and that you not add components to the other 3 sides. The

buttons you add to the toolbar are just small JButtons with picture icons and without text. The JToolBar class has two important methods: *add* and *addSeparator*.

```
JToolBar toolbar = new JtoolBar();
JButton Open = new JButton(“open.gif”);
toolbar.add(Open);
toolbar.addSeparator();
```

By default, JButtons have a rectangular shape, and to make the usual square-looking buttons, you need to use square icons and set the Insets of the button to zero. On most toolbars, the icons are 25 x 25 pixels. We thus develop the simple ToolButton class below, which handles both the insets and the size:

```
public class ToolButton extends JButton
{
   public ToolButton(Icon img)
   {
      super(img);
      setMargin(new Insets(0,0,0,0));
      setSize(25,25);
   }
}
```

The JToolBar also has the characteristic that you can detach it from its anchored position along the top side of the program and attach it to another side, or leave it floating. This allows some user customization of the running program, but is otherwise not terribly useful. It also is not particularly well implemented and can be confusing to the user. Thus, we recommend that you use the *setFloatable(false)* method to turn this feature off.

Toggle Buttons

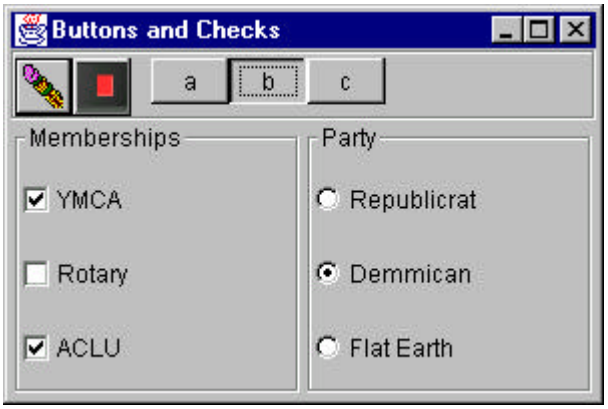
The JToggleButton class is actually the parent class for check boxes and radio buttons. It is a two-state button that will stay in an up or down position when clicked, and you can use it just like a check box. While toggle buttons look sort of strange on most screens, they look very reasonable as part of toolbars. You can use individual toggle buttons to indicate the state of actions the user might select. By themselves, toggle buttons behave lick check boxes, so you can press as many as you want, and you can “uncheck” or raise toggle buttons by using the *setSelected(false)* method.

You can also add toggle buttons to a ButtonGroup so that they behave like radio buttons: only one at a time can be pressed down. However, once a ButtonGroup object is mediating them, you can’t raise the buttons

using the *setSelected* method. If you want to be able to raise them, but still only allow one at a time to be pressed, you need to write your own Medator class to replace the ButtonGroup object.

Sample Code

The simple program display below illustrates checkboxes, radio buttons, toolbar buttons and toggle buttons:



Note the “b” JToggleButton is depressed permanently. While the user can select any number of organizations in which he holds memberships, he can only select one political party.

MENUS AND ACTIONS

The JMenuBar and JMenu classes in Swing work just about identically to those in the AWT. However, the JMenuItem class adds constructors that allow you to include an image alongside the menu text. To create a menu, you create a menu bar, add top-level menus and then add menu items to each of the top-level menus.

```
JMenuBar mbar = new JMenuBar();      //menu bar
setJMenuBar(mbar);                   //add to JFrame
JMenu mFile = new JMenu("File");     //top-level menu
mbar.add(mFile);                     //add to menu bar
JMenuItem Open = new JMenuItem("Open");     //menu items
JMenuItem Exit = new JMenuItem("Exit");
mFile.add(Open);                     //add to menu
mFile.addSeparator();                //put in separator
mFile.add(Exit);
```

JMenuItems also generate ActionEvents, and thus menu clicks causes these events to be generated. As with buttons, you simply add action listeners to each of them.

```
Open.addActionListener(this);          //for example
Exit.addActionListener(this);
```

Action Objects

Menus and toolbars are really two ways of representing the same thing: a single click interface to initiate some program function. Swing also provides an Action interface that encompasses both.

```
public void putValue(String key, Object value);
public Object getValue(String key);
public void actionPerformed(ActionEvent e);
```

You can add this interface to an existing class or create a JComponent with these methods and use it as an object which you can add to either a JMenu or JToolBar. The most effective way is simply to extend the *AbstractAction* class. The JMenu and JToolbar will then display it as a menu item or a button respectively. Further, since an Action object has a single action listener built in, you can be sure that selecting either one will have exactly the same effect. In addition, disabling the Action object has the advantage of disabling both representations on the screen.

Let's see how this works. We can start with a basic abstract ActionButton class, and use a Hashtable to store and retrieve the properties.

```
public abstract class ActionButton extends AbstractAction
       implements Action
{
   Hashtable properties;

   public ActionButton(String caption, Icon img)
   {
      properties = new Hashtable();
      properties.put(DEFAULT, caption);
      properties.put(NAME, caption);
      properties.put(SHORT_DESCRIPTION, caption);
      properties.put(SMALL_ICON, img);
   }
   public void putValue(String key, Object value)   {
      properties.put(key, value);
   }
   public Object getValue(String key) {
      return properties.get(key);
   }
   public abstract void actionPerformed(ActionEvent e);
}
```

The properties that Action objects recognize by key name are

- String DEFAULT
- String LONG_DESCRIPTION
- String NAME
- String SHORT_DESCRIPTION
- String SMALL_ICON

The NAME property determines the label for the menu item and the button, and in theory the LONG_DESCRIPTION should be used. This latter feature is not implemented in Swing 1.0x, but is expected to be in Java 1.2. The icon feature does work correctly.

Now we can easily derive an ExitButton from the ActionButton like this:

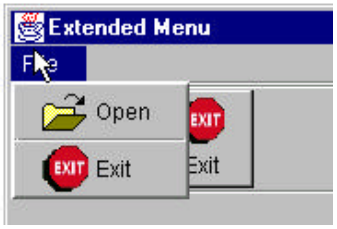
```
public class ExitButton extends ActionButton
{

   JFrame fr;
   public ExitButton(String caption, Icon img, JFrame frm)    {
     super(caption, img);
```

```
      fr = frm;
   }
   public void actionPerformed(ActionEvent e)    {
      System.exit(0);
   }
}
```

and similarly for the FileButton. We add these to the toolbar and menu as follows:

```
//Add File menu
   JMenu mFile = new JMenu("File");
   mbar.add(mFile);

   //create two Action Objects
   Action Open = new FileButton("Open",
             new ImageIcon("open.gif"), this);
   mFile.add(Open);

   Action Exit = new ExitButton("Exit",
             new ImageIcon("exit.gif"), this);
   mFile.addSeparator();
   mFile.add(Exit);
//add same objects to the toolbar
   toolbar = new JToolBar();
   getContentPane().add(jp = new JPanel());
   jp.setLayout(new BorderLayout());
   jp.add("North", toolbar);

   //add the two action objects
   toolbar.add(Open);
toolbar.add(Exit);
```

This code produces the program window shown below:



the menu or the text on the toolbar. Howev er, the *add* methods of the toolbar nd menu have a unique feature when usedto add an ACTION OBJECT. They return an object of type Jbutton or JmenuItem respectively. Then you can use these to set the features the way you want them. For the menu, we want to remove the icon

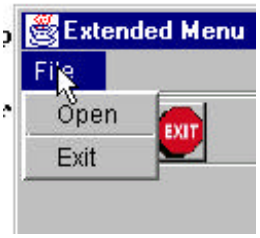
```
Action Open = new FileButton("Open",
```

```
            new ImageIcon("open.gif"), this);
    menuitem = mFile.add(Open);
    menuitem.setIcon(null);
```

and for the button, we want to remove the text and add a tooltip:

```
    JButton button = toolbar.add(act);
    button.setText("");
    button.setToolTipText(tip);
    button.setMargin(new Insets(0,0,0,0));
```

This gives us the screen look we want:



Design Patterns in the Action Object

One reason to spend a little time discussing objects that implement the Action interface is that they exemplify at least two design patterns. First, each Action object must have its own *actionListener* method, and thus can directly launch the code to respond that that action. In addition, even though these Action objects may have two (or more) visual instantiations, they provide a single point that launches this code. This is an excellent example of the Command pattern.

In addition, the Action object takes on different visual aspects depending on whether it is added to a menu or to a toolbar. In fact you could decide that the Action object is a Factory pattern which produces a button or menu object depending on where it is added. In fact, it does seem to be a Factory, because the toolbar and menu *add* methods return instances on those objects. On the other hand, the Action object seems to be a single object, and gives different appearances depending on its environment. This is a description of the State pattern, where an object seems to change class (or methods) depending on the internal state of the object.

One of the interesting and challenging things about the design patterns we discuss in this book is that once you start looking a it, you

discover that they are represented far more widely than you first assumed. In some cases their application and implementation is obvious, and in other cases the implementation is a bit subtle. In these cases you srot of step back, tilt your head, squint and realize that from that angle it *looks* like a pattern where you hadn't noticed one before. Again, being able to label the code as exemplifying a pattern makes it easier for you to remember how it works and easier for you to communicate to others how that code is constructed.

THE JLIST CLASS

The JList class is a more powerful replacement for the simple List class that is provided with the AWT. A JList can be instantiated using a Vector or array to represent its contents. The JList does not itself support scrolling and thus must be added to a JScrollPane to allow scrolling to take place.

In the simplest program you can write using a JList, you

1. add a JScrollPane to the Frame, and then
2. create a Vector of data
3. create a JList using that Vector
4. add the JlList to the JScrollPane's viewport

This is shown below

```
JPanel jp = new JPanel();         //panel in Frame
getContentPane().add(jp);

//create scroll pane
JScrollPane sp = new JScrollPane();
jp.add(sp);                       //add to layout
Vector dlist = new Vector();      //create vector
dlist.addElement("Anchovies");    //and add data
dlist.addElement("Bananas");
dlist.addElement("Cilantro");
dlist.addElement("Doughnuts");
dlist.addElement("Escarrole");
JList list= new JList(dlist); //create list with data
sp.getViewport().add(list);   //add list to scrollpane
```

This produces the display shown below:

You could just as easily use an array instead of a Vector and have the same result.

List Selections and Events

You can set the JList to allow users to select a single line, multiple contiguous lines or separated multiple lines with the *setselectionMode* method, where the arguments can be

```
SINGLE_SELECTION
SINGLE_INTERVALSELECTION
MULTIPLE_INTERVAL_SELECTION
```

You can then receive these events by using the *addListSelectionListener* method. Your ListSelectionListener must implement the interface

```
public void valueChanged(ListSelectionEvent e)
```

In our JListLDemo.java example we display the selected list item in a text field:

```
public void valueChanged(ListSelectionEvent e)   {
      text.setText((String)list.getSelectedValue());
}
```

This is shown below:

Changing a List Display Dynamically

If you want a list to change dynamically during your program, the problem is somewhat more involved because the JList displays the data it is initially loaded with and does not update the display unless you tell it to. One simple way to accomplish this is to use the *setListData* method of JList to keep passing it a new version of the updated Vector after you change it. In the JListADemo.java program, we add the contents of the top text field to the list each time the Add button is clicked. All of this takes place in the action routine for that button:

```
public void actionPerformed(ActionEvent e)    {
  dlist.addElement(text.getText()); //add text from field
  list.setListData(dlist);       //send new Vector to list
  list.repaint();                //and tell it to redraw
 }
```

One drawback to this simple solution is that you are passing the entire Vector to the list each time and it must update its entire contents each time rather than only the portion that has changed. This brings us to the underlying ListModel that contains the data the JList displays.

When you create a JList using an array or Vector, the JList automatically creates a simple ListModel object which contains that data. The ListModel objects are an extension of the AbstractListModel class. This model has the following simple methods:

```
void fireContentsChanged(Object source, int index0, int index1)
void fireIntervalAdded(Object source, int index0, int index1)
void fireIntervalRemoved(Object source, int index0, int index1)
```

and you need only implement those *fire* methods your program will be using. For example, in this case we really only need to implement the *fireIntervalAdded* method.

Our ListModel is an object that contains the data (in a Vector or other suitable structure) and notifies the JList whenever it changes. Here, the list model is just the following:

```
class JListData extends AbstractListModel {
   private Vector dlist;   //the food name list

   public JListData()    {
      dlist = new Vector();
      makeData();          //create the food names
   }
   public int getSize()    {
      return dlist.size();
   }
   private Vector makeData()
   {
       //add food names as before
       return dlist;
   }
   public Object getElementAt(int index)    {
      return dlist.elementAt(index);
   }
  //add string to list and tell the list about it
   public void addElement(String s)    {
      dlist.addElement(s);
      fireIntervalAdded(this, dlist.size()-1, dlist.size());
   }
}
```

This ListModel approach is really an implementation of the Observer design pattern we discuss in Chapter 4. The data are in one class and the rendering or display methods in another class, and the communication between them triggers new display activity.

Lists displayed by JList are not limited to text-only displays. The ListModel data can be a Vector or array of any kind of Objects. If you use the default methods, then only he String representation of those objects will be displayed. However, you can define your own display routines using the *setCellRenderer* method, and have it display icons or other colored text or graphics as well.

THE JTABLE CLASS

The JTable class is much like the JList class, in that you can program it very easily to do simple things. Similarly, in order to do sophisticated things, you need to create a class derived from the AbtractTableModel class to hold your data.

A Simple JTable Program

In the simplest program, you just create a rectangular array of objects and use it in the constructor for the Jtable. You can also include an array of strings to be used a column labels for the table.

```
public SimpleTable()
   {
      super("Simple table");
      JPanel jp = new JPanel();
      getContentPane().add(jp);
      Object[] [] musicData = {
         {"Tschaikovsky", "1812 Overture", new Boolean(true)},
         {"Stravinsky", "Le Sacre", new Boolean(true)},
         {"Lennon","Eleanor Rigby", new Boolean(false)},
         {"Wagner", "Gotterdammerung", new Boolean(true)}
      };
      String[] columnNames = {"Composer", "Title",
                                 "Orchestral"};
      JTable table = new JTable(musicData, columnNames);
      JScrollPane sp = new JScrollPane(table);
      table.setPreferredScrollableViewportSize(
                  new Dimension(250,170));
      jp.add(sp);

      setSize(300,200);
      setVisible(true);
   }
```

This produces the simple table display below:

This table has all cells editable and displays all the cells using the *toString* method of each object. Of course, like the JList interface, this simple interface to JTable creates a data model object under the covers. In order to produce a more flexible display you need to create that data model yourself.

You can create a TableModel by extending the AbstractTableModel class. All of the methods have default vales and operations except the following 3 which you must provide:

```
public int getRowCount();
public int getColumnCount();
public Object getValueAt(int row, int column);
```

However, you can gain a good deal more control by adding a couple of other methods. You can use the method

```
public boolean isCellEditable(int row, int col)
```

to protect some cells from being edited. If you want to allow editing of some cells, you must provide the implementation for the method

```
public void setValueAt(Object obj, int row, int col)
```

Further, by adding a method which returns the data class of each object to be displayed, you can make use of some default cell formatting behavior. The JTable's default cell renderer displays

- Numbers as right-aligned labels
- ImageIcons as centered labels
- Booleans as checkboxes
- Objects using their *toString* method

You simply need to return the class of the objects in each column:

```
   public Class getColumnClass( int col)       {
      return getValueAt(0, col).getClass();
   }
```

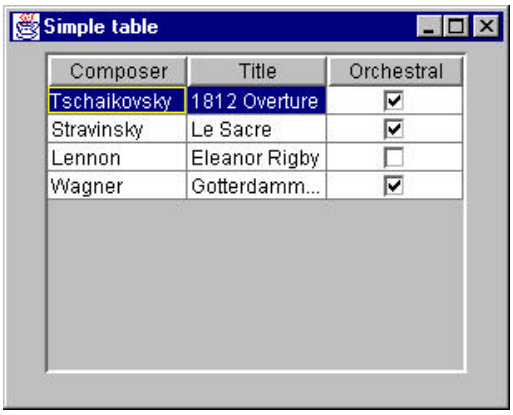
Our complete table model class creates exactly the same array and table column captions as before and implements the methods we just mentioned:

```
class MusicModel extends AbstractTableModel
{
   String[] columnNames = {"Composer", "Title", "Orchestral"};

   Object[] [] musicData = {
   {"Tschaikovsky", "1812 Overture", new Boolean(true)},
   {"Stravinsky", "Le Sacre", new Boolean(true)},
   {"Lennon","Eleanor Rigby", new Boolean(false)},
   {"Wagner", "Gotterdammerung", new Boolean(true)}
   };

   int rowCount, columnCount;
   //-----------------------------------------
   public MusicModel()    {
      rowCount = 4;
      columnCount =3;
   }
   //-----------------------------------------
   public String getColumnName(int col)    {
      return columnNames[col];
   }
   //-----------------------------------------
   public int getRowCount(){return rowCount;}
   public int getColumnCount(){return columnCount;}
   //-----------------------------------------
   public Class getColumnClass( int col)    {
      return getValueAt(0, col).getClass();
   }
   //-----------------------------------------
   public boolean isCellEditable(int row, int col)    {
      return (col > 1);
   }
   //-----------------------------------------
   public void setValueAt(Object obj, int row, int col)    {
      musicData[row][col] = obj;
        fireTableCellUpdated(row, col);
   }
   //-----------------------------------------
   public Object getValueAt(int row, int col)       {
         return musicData[row][col];
      }
```

The main program simply becomes:

```
public ModelTable()
{
   super("Simple table");
   JPanel jp = new JPanel();
   getContentPane().add(jp);
   JTable table = new JTable(new MusicModel());
   JScrollPane sp = new JScrollPane(table);
   table.setPreferredScrollableViewportSize(
                new Dimension(250,170));
   jp.add(sp);

   setSize(300,200);
   setVisible(true);
}
```

As you can see in our revised program display, the boolean column is now rendered as check boxes. We have also only allowed editing of the right-most column y using the *isCellEditable* method to disallow it for columns 0 and 1.



As we noted in the JList section, the TableModel class is a class which holds and manipulates the data and notifies the Jtable whenever it changes. Thus, the Jtable is an Observer pattern, operating on the TableModel data.

Cell Renderers

Each cell in a table is rendered by a cell renderer. The default renderer is a JLabel, and it may be used for all the data in several columns. Thus, these cell renderers can be thought of as Flyweight pattern implementations. The JTable class chooses the renderer accoring to the object's type as we outlined above. However, you can change to a different

rendered, such as one that uses another color, or another visual interface quite easily.

Cell renderers are registered by type of data:

```
table.setDefaultRenderer(String.class, new ourRenderer());
```

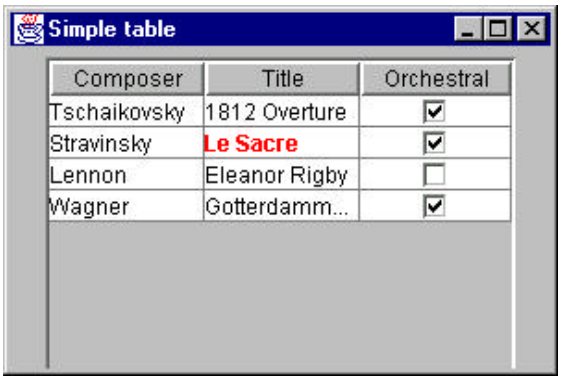
and each renderer is passed the object, selected mode, row and column using the only required public method:

```
public Component getTableCellRendererComponent(JTable jt,
       Object value, boolean isSelected,
       boolean hasFocus, int row, int column)
```

One common way to implement a cell renderer is to extend the JLabel type and catch each rendering request within the renderer and return a properly configured JLabel object, usually the renderer itself. The renderer below displays cell (1,1) in boldface red type and the remaining cells in plain, black type:

```
public class ourRenderer extends JLabel
       implements TableCellRenderer
{
   Font bold, plain;
  public ourRenderer()   {
     super();
     setOpaque(true);
     setBackground(Color.white);
     bold = new Font("SansSerif", Font.BOLD, 12);
     plain = new Font("SansSerif", Font.PLAIN, 12);
     setFont(plain);
  }
//-------------------------------------------------------
  public Component getTableCellRendererComponent(JTable jt,
       Object value, boolean isSelected,
       boolean hasFocus, int row, int column)
        {
          setText((String)value);
          if(row ==1 && column==1)           {
             setFont(bold);
             setForeground(Color.red);
          }
          else           {
            setFont(plain);
            setForeground(Color.black);
          }
        return this;
       }
  }
```

The results of this rendering are shown below:



In the simple cell renderer shown above the renderer is itself a JLabel which returns a different font, but the same object, depending on the row and column. More complex renderers are also possible where one of several already-instantiated objects is returned, making the renderer a Component Factory.

THE JTREE CLASS

Much like the JTable and JList, the JTree class consists of a data model and an observer. One of the easiest ways to build up the tree you want to display is to create a root node and then add child notes to it and to each of them as needed. The *DefaultMutableTreeNode* class is provided as an implementation of the *TreeNode* interface.

You create the JTree with a root node as its argument

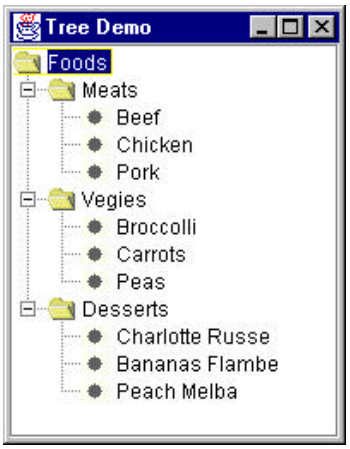
```
root = new DefaultMutableTreeNode("Foods");
JTree tree = new JTree(root);
```

and then add each node to the root, and additional nodes to those to any depth. The following simple program produces a food tree list by category:

```
public class TreeDemo extends JxFrame
{
   DefaultMutableTreeNode root;
   public TreeDemo()
   {
   super("Tree Demo");
   JPanel jp = new JPanel();   // create interior panel
   jp.setLayout(new BorderLayout());
   getContentPane().add(jp);

   //create scroll pane
   JScrollPane sp =  new JScrollPane();
   jp.add("Center", sp);

   //create root node
   root = new DefaultMutableTreeNode("Foods");
   JTree tree = new JTree(root);    //create tree
   sp.getViewport().add(tree);      //add to scroller

   //create 3 nodes, each with three sub nodes
   addNodes("Meats", "Beef", "Chicken", "Pork");
   addNodes("Vegies", "Broccolli", "Carrots", "Peas");
   addNodes("Desserts","Charlotte Russe",
             "Bananas Flambe","Peach Melba");

   setSize(200, 300);
   setVisible(true);
   }
   //---------------------------------------
   private void addNodes(String b, String n1, String n2,
                                                String n3)
   {
```

```
        DefaultMutableTreeNode base =
              new DefaultMutableTreeNode(b);
        root.add(base);
        base.add(new DefaultMutableTreeNode(n1));
        base.add(new DefaultMutableTreeNode(n2));
        base.add(new DefaultMutableTreeNode(n3));
}
```

The tree it generates is shown below.



If you want to know if a user has clicked on a particular line of this tree, you can add a TreeSelectionListener and catch the *valueChanegd* event. The *TreePath* you can obtain from the *getPath* method of the TreeSelectionEvent is the complete path back to the top of the tree. However the *getLastPathComponent* method will return the string of the line the user actually selected. You will see that we use this method and display in the Composite pattern example.

```
public void valueChanged(TreeSelectionEvent evt)     {
       TreePath path = evt.getPath();
       String selectedTerm =
             path.getLastPathComponent().toString();
```

The TreeModel Interface

The simple tree we build above is based on adding a set of nodes to make up a tree. This is an implementation of the DefaultTreeModel class which handles this structure. However, there might well be many other sorts of data structure that you'd like to display using this tree display. To do so, you create a class of your own to hold these data which implements the TreeModel interface. This interface is very simple indeed, consisting only of

```
void addTreeModelListener(TreeModelListener l);
Object getChils(Object parent, int index);
int getChildCount(Object parent);
int getIndexOf Child(Object parent, Object child);
Object getRoot();
boolean isLeaf(Object);
void removeTreeModelListener(TreeModelListener l);
void value ForPathChanges(TreePath path, Object newValue);
```

Note that this general interface model does not specify anything about how you add new nodes, or add nodes to nodes. You can implement that in any way that is appropriate for your data.

Summary

In this brief chapter, we've touched on some of the more common JFC controls, and noted how frequently the design patterns we're discussing in this books are represented. Now, we can go on and use these Swing controls in our programs as we develop code for the rest of the patterns.

Structural Patterns

Structural patterns describe how classes and objects can be combined to form larger structures. The difference between *class* patterns and *object* patterns is that class patterns describe how inheritance can be used to provide more useful program interfaces. Object patterns, on the other hand, describe how objects can be composed into larger structures using object composition, or the inclusion of objects within other objects.

For example, we'll see that the Adapter pattern can be used to make one class interface match another to make programming easier. We'll also look at a number of other structural patterns where we combine objects to provide new functionality. The Composite, for instance, is exactly that: a composition of objects, each of which may be either simple or itself a composite object. The Proxy pattern is frequently a simple object that takes the place of a more complex object that may be invoked later, for example when the program runs in a network environment.

The Flyweight pattern is a pattern for sharing objects, where each instance does not contain its own state, but stores it externally. This allows efficient sharing of objects to save space, when there are many instances, but only a few different types.

The Façade pattern is used to make a single class represent an entire subsystem, and the Bridge pattern separates an object's interface from its implementation, so you can vary them separately. Finally, we'll look at the Decorator pattern, which can be used to add responsibilities to objects dynamically.

You'll see that there is some overlap among these patterns and even some overlap with the behavioral patterns in the next chapter. We'll summarize these similarities after we describe the patterns.

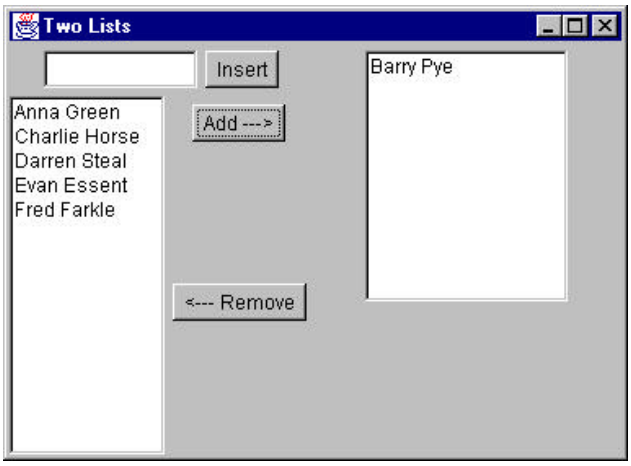
THE ADAPTER PATTERN

The Adapter pattern is used to convert the programming interface of one class into that of another. We use adapters whenever we want unrelated classes to work together in a single program. The concept of an adapter is thus pretty simple; we write a class that has the desired interface and then make it communicate with the class that has a different interface.

There are two ways to do this: by inheritance, and by object composition. In the first case, we derive a new class from the nonconforming one and add the methods we need to make the new derived class match the desired interface. The other way is to include the original class inside the new one and create the methods to translate calls within the new class. These two approaches, termed class adapters and object adapters are both fairly easy to implement in Java.

Moving Data between Lists

Let's consider a simple Java program that allows you to enter names into a list, and then select some of those names to be transferred to another list. Our initial list consists of a class roster and the second list, those who will be doing advanced work.



In this simple program, you enter names into the top entry field and click on Insert to move the names into the left-hand list box. Then, to move

names to the right-hand list box, you click on them, and then click on Add. To remove a name from the right hand list box, click on it and then on Remove. This moves the name back to the left-hand list.

This is a very simple program to write in Java 1.1. It consists of a GUI creation constructor and an actionListener routine for the three buttons:

```
public void actionPerformed(ActionEvent e)
   {
      Button b = (Button)e.getSource();
      if(b == Add)
         addName();
      if(b == MoveRight)
         moveNameRight();
      if(b == MoveLeft)
         moveNameLeft();
   }
```

The button action routines are then simply

```
   private void addName()
   {
      if (txt.getText().length() > 0)
         {
         leftList.add(txt.getText());
         txt.setText("");
         }
   }
   //-----------------------------------------
   private void moveNameRight()
   {
     String sel[] = leftList.getSelectedItems();
     if (sel != null)
     {
     rightList.add(sel[0]);
     leftList.remove(sel[0]);
     }
   }
   //-----------------------------------------
   public void moveNameLeft()
   {
    String sel[] = rightList.getSelectedItems();
     if (sel != null)
     {
     leftList.add(sel[0]);
     rightList.remove(sel[0]);
     }
   }
```

This program is called TwoList.java on your CD-ROM.

Using the JFC JList Class

This is all quite straightforward, but suppose you would like to rewrite the program using the Java Foundation Classes (JFC or "Swing"). Most of the methods you use for creating and manipulating the user interface remain the same. However, the JFC JList class is markedly different than the AWT List class. In fact, because the JList class was designed to represent far more complex kinds of lists, there are virtually no methods in common between the classes:

awt List class	JFC JList class
add(String);	---
remove(String)	---
String[] getSelectedItems()	Object[] getSelectedValues()

Both classes have quite a number of other methods and almost none of them are closely correlated. However, since we have already written the program once, and make use of two different list boxes, writing an adapter to make the JList class look like the List class seems a sensible solution to our problem.

The JList class is a window container which has an array, vector or other ListModel class associated with it. It is this ListModel that actually contains and manipulates the data. Further, the JList class does not contain a scroll bar, but instead relies on being inserted in the viewport of the JScrollPane class. Data in the JList class and its associated ListModel are not limited to strings, but may be almost any kind of objects, as long as you provide the cell drawing routine for them. This makes it possible to have list boxes with pictures illustrating each choice in the list.

In our case, we are only going to create a class that emulates the List class, and that in this simple case, needs only the three methods we showed in the table above.

We can define the needed methods as an interface and then make sure that the class we create implements those methods:

```
public interface awtList  {
     public void add(String s);
     public void remove(String s);
     public String[] getSelectedItems()
}
```

Interfaces are important in Java, because Java does not allow multiple inheritance as C++ does. Thus, by using the *implements* keyword, the class can take on methods and the appearance of being a class of either type.

The Object Adapter

In the object adapter approach, we create a class that *contains* a JList class but which implements the methods of the awtList interface above. This is a pretty good choice here, because the outer container for a JList is not the list element at all, but the JScrollPane that encloses it.

So, our basic JawtList class looks like this:

```
public class JawtList extends JScrollPane
   implements awtList
{
   private JList listWindow;
   private JListData listContents;
//---------------------------------------
   public JawtList(int rows)    {
      listContents = new JListData();
      listWindow = new JList(listContents);
       getViewport().add(listWindow);

   }
//---------------------------------------
   public void add(String s)    {
      listContents.addElement(s);
   }
//---------------------------------------
   public void remove(String s)    {
      listContents.removeElement(s);
   }
//---------------------------------------
   public String[] getSelectedItems()    {
      Object[] obj = listWindow.getSelectedValues();
      String[] s = new String[obj.length];
      for (int i =0; i<obj.length; i++)
            s[i] = obj[i].toString();
      return s;
   }
}
```

Note, however, that the actual data handling takes place in the JlistData class. This class is derived from the AbstractListModel, which defines the following methods:

addListDataListener(l)	Add a listener for changes in the data.

removeListDataListener(l)	Remove a listener
fireContentsChanged(obj, min,max)	Call this after any change occurs between the two indexes min and max
fireIntervalAdded(obj,min,max)	Call this after any data has been added between min and max.
fireIntervalRemoved(obj, min, max)	Call this after any data has been removed between min and max.

The three *fire* methods are the communication path between the data stored in the ListModel and the actual displayed list data. Firing them causes the displayed list to be updated.

In this case, the addElement, removeElement methods are all that are needed, although you could imagine a number of other useful methods. Each time we add data to the *data* vector, we call the *fireIntervalAdded* method to tell the list display to refresh that area of the displayed list.

```
class JListData extends AbstractListModel
{
   private Vector data;
//----------------------------------------
   public JListData()    {
      data = new Vector();
   }
//----------------------------------------
   public void addElement(String s)
   {
      data.addElement(s);
      fireIntervalAdded(this, data.size()-1,
                    data.size());
   }
//----------------------------------------
   public void removeElement(String s)    {
      data.removeElement(s);
      fireIntervalRemoved(this, 0, data.size());
   }
}
```

The Class Adapter

In Java, the class adapter approach isn't all that different. If we create a class JawtClassList that is derived from JList, then we have to create a JScrollPane in our main program's constructor:

```
        leftList = new JclassAwtList(15);
       JScrollPane lsp = new JScrollPane();
       pLeft.add("Center", lsp);
       lsp.getViewport().add(leftList);
```

and so forth.

The class-based adapter is much the same, except that some of the methods now refer to the enclosing class instead of an encapsulated class:

```
public class JclassAwtList extends JList
   implements awtList
{
   private JListData listContents;
//---------------------------------------
   public JclassAwtList(int rows)
   {
      listContents = new JListData();
      setModel(listContents);
      setPrototypeCellValue("Abcdefg Hijkmnop");
   }
```

There are some differences between the List and the adapted JList class that are not so easy to adapt, however. The List class constructor allows you to specify the length of the list in lines. There is no way to specify this directly in the JList class. You can compute the preferred size of the enclosing JScrollPane class based on the font size of the JList, but depending on the layout manager, this may not be honored exactly.

You will find the example class JawtClassList, called by JTwoClassList on your example CD-ROM.

There are also some differences between the class and the object adapter approaches, although they are less significant than in C++.

- The Class adapter
    - Won't work when we want to adapt a class and all of its subclasses, since you define the class it derives from when you create it.
    - Lets the adapter change some of the adapted class's methods but still allows the others to be used unchanged.
- An Object adapter
    - Could allow subclasses to be adapted by simply passing them in as part of a constructor.

- Requires that you specifically bring any of the adapted object's methods to the surface that you wish to make available.

Two Way Adapters

The two-way adapter is a clever concept that allows an object to be viewed by different classes as being either of type awtList or a type JList. This is most easily carried out using a class adapter, since all of the methods of the base class are automatically available to the derived class. However, this can only work if you do not override any of the base class's methods with ones that behave differently. As it happens, our JawtClassList class is an ideal two-way adapter, because the two classes have no methods in common. You can refer to the awtList methods or to the JList methods equally conveniently.

Pluggable Adapters

A pluggable adapter is one that adapts dynamically to one of several classes. Of course, the adapter can only adapt to classes it can recognize, and usually the adapter decides which class it is adapting based on differing constructors or setParameter methods.

Java has yet another way for adapters to recognize which of several classes it must adapt to: reflection. You can use reflection to discover the names of public methods and their parameters for any class. For example, for any arbitrary object you can use the *getClass()* method to obtain its class and the *getMethods()* method to obtain an array of the method names.

```
JList list = new JList();
      Method[] methods = list.getClass().getMethods();
       //print out methods
      for (int i = 0; i < methods.length; i++)       {
         System.out.println(methods[i].getName());
          //print out parameter types
         Class cl[] = methods[i].getParameterTypes();
         for(int j=0; j < cl.length; j++)
            System.out.println(cl[j].toString());
      }
```

A "method dump" like the one produced by the code shown above can generate a very large list of methods, and it is easier if you know the name of the method you are looking for and simply want to find out which arguments that method requires. From that method signature, you can then deduce the adapting you need to carry out.

However, since Java is a strongly typed language, it is more likely that you would simply invoke the adapter using one of several constructors, where each constructor is tailored for a specific class that needs adapting.

Adapters in Java

In a broad sense, there are already a number of adapters built into the Java language. In this case, the Java adapters serve to simplify an unnecessarily complicated event interface. One of the most commonly used of these Java adapters is the WindowAdapter class.

One of the inconveniences of Java is that windows do not close automatically when you click on the Close button or window Exit menu item. The general solution to this problem is to have your main Frame window implement the WindowListener interface, leaving all of the Window events empty except for windowClosing.

```
public void mainFrame extends Frame
       implements WindowListener
{
       public void mainFrame()    {
       addWindowListener(this);   //frame listens
                                         //for window events
       }

   public void windowClosing(WindowEvent wEvt)    {
    System.exit(0);   //exit on System exit box clicked
   }
   public void windowClosed(WindowEvent wEvt){}
   public void windowOpened(WindowEvent wEvt){}
   public void windowIconified(WindowEvent wEvt){}
   public void windowDeiconified(WindowEvent wEvt){}
   public void windowActivated(WindowEvent wEvt){}
   public void windowDeactivated(WindowEvent wEvt){}
}
```

As you can see, this is awkward and hard to read. The WindowAdapter class is provided to simplify this procedure. This class contains empty implementations of all seven of the above WindowEvents. You need then only override the windowClosing event and insert the appropriate exit code.

One such simple program is shown below:

```
//illustrates using the WindowAdapter class
public class Closer extends Frame {
  public Closer()   {
     WindAp windap = new WindAp();
     addWindowListener(windap);
     setSize(new Dimension(100,100));
```

```
     setVisible(true);
  }
  static public void main(String argv[])   {
     new Closer();
  }
}
//make an extended window adapter which
//closes the frame when the closing event is received
class WindAp extends WindowAdapter {
   public void windowClosing(WindowEvent e)    {
      System.exit(0);
   }
}
```

You can, however, make a much more compact, but less readable version of the same code by using an anonymous inner class:

```
//create window listener for window close click
      addWindowListener(new WindowAdapter()
       {
           public void windowClosing(WindowEvent e)
                    {System.exit(0);}
       });
```

Adapters like these are common in Java when a simple class can be used to encapsulate a number of events. They include ComponentAdapter, ContainerAdapter, FocusAdapter, KeyAdapter, MouseAdapter, and MouseMotionAdapter.

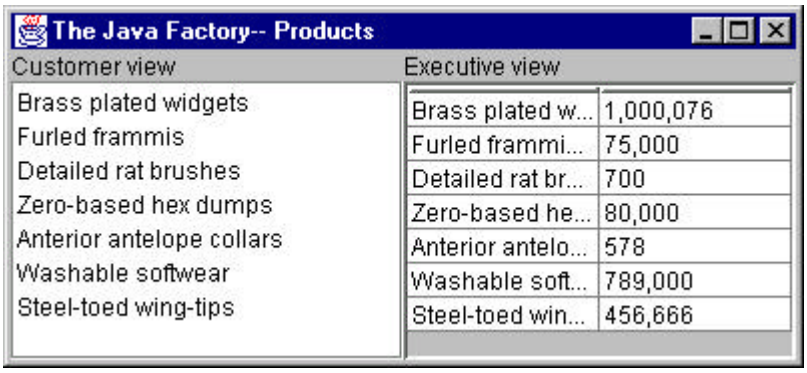
THE BRIDGE PATTERN

The Bridge pattern is used to separate the interface of class from its implementation, so that either can be varied separately. At first sight, the bridge pattern looks much like the Adapter pattern, in that a class is used to convert one kind of interface to another. However, the intent of the Adapter pattern is to make one or more classes' interfaces look the same as that of a particular class. The Bridge pattern is designed to separate a class's interface from its implementation, so that you can vary or replace the implementation without changing the client code.

Suppose that we have a program that displays a list of products in a window. The simplest interface for that display is a simple JList box. But, once a significant number of products have been sold, we may want to display the products in a table along with their sales figures.

Since we have just discussed the adapter pattern, you might think immediately of the class-based adapter, where we adapt the fairly elaborate interface of the JList to our simpler needs in this display. In simple programs, this will work fine, but as we'll see below there are limits to that approach.

Let's further suppose that we need to produce two kinds of displays from our product data, a customer view that is just the list of products we've mentioned, and an executive view which also shows the number of units shipped. Well display the product list in an ordinary JList box and the executive view in a JTable table display. To simplify peripheral programming issues, we'll just show both displays as two lists in a single window, as we see below:



At the top programming level, we just create instances of a table and a list from classes derived from JList and Jtable but designed to parse apart the names and the quantities of data.

```
pleft.setLayout(new BorderLayout());
pright.setLayout(new BorderLayout());

//add in customer view as list box
pleft.add("North", new JLabel("Customer view"));
pleft.add("Center", new productList(prod));

//add in execute view as table
pright.add("North", new JLabel("Executive view"));
pright.add("Center", new productTable(prod));
```

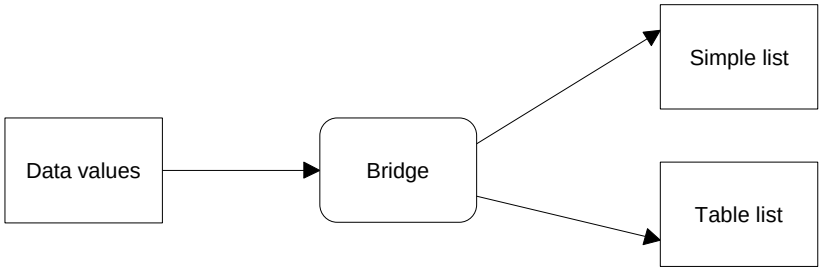
We derive the *productList* class directly from the JawtList class we just wrote, so that the Vector containing the list of products is the only input to the class.

```
public class productList extends JawtList
{
   public productList(Vector products)
   {
      super(products.size());    //for compatibility
      for (int i = 0; i < products.size(); i++)
      {
         //take each string apart and keep only
         //the product names, discarding the quantities
         String s = (String)products.elementAt(i);

          //separate qty from name
          int index = s.indexOf("--");
          if(index > 0)
            add(s.substring(0, index));
         else
            add(s);
      }
   }
}
```

Building a Bridge

Now suppose that we need to make some changes in the way these lists display the data. For example, you might want to have the products displayed in alphabetical order. In order to continue with this approach, you'd need to either modify or subclass *both* of these list classes. This can quickly get to be a maintenance nightmare, especially if more than two such displays eventually are needed. So rather than deriving new classes whenever we need

to change these displays further, let's build a single *bridge* that does this work for us:



We want the bridge class to return an appropriate visual component so we'll make it a kind of scroll pane class:

```
public class listBridge extends JscrollPane
```

When we design a bridge class, we have to decide how the bridge will determine which of the several classes it is to instantiate. It could decide based on the values or quantity of data to be displayed, or it could decide based on some simple constants. Here we define the two constants inside the listBridge class:

```
static public final int TABLE = 1, LIST = 2;
```

We'll keep the main program constructor much the same, replacing specialized classes with two calls to the constructor of our new listBridge class:

```
pleft.add("North", new JLabel("Customer view"));
pleft.add("Center",
             new listBridge(prod, listBridge.LIST));

 //add in execute view as table
pright.add("North", new JLabel("Executive view"));
pright.add("Center",
             new listBridge(prod, listBridge.TABLE));
```

Our constructor for the listBridge class is then simply

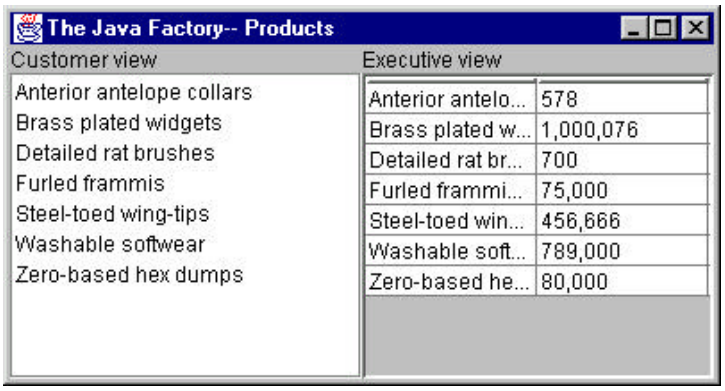
```
public listBridge(Vector v, int table_type)
   {
      Vector sort = sortVector(v);      //sort the vector
```

```
      if (table_type == LIST)
         getViewport().add(makeList(sort));  //make table

      if (table_type == TABLE)
         getViewport().add(makeTable(sort)); //make list

   }
```

The important difference in our bridge class is that we can use the JTable and JList class directly without modification and thus can put any adapting interface computations in the data models that construct the data for the list and table.

```
private JList makeList(Vector v)    {
     return new JList(new BridgeListData(v));
  }
//---------------------------------
private JTable makeTable(Vector v)    {
      return new JTable(new prodModel(v));
  }
```

The resulting sorted display is shown below:



Consequences of the Bridge Pattern

1. The Bridge pattern is intended to keep the interface to your client program constant while allowing you to change the actual kind of class you display or use. This can prevent you from recompiling a complicated set of user interface modules, and only require that you recompile the bridge itself and the actual end display class.
2. You can extend the implementation class and the bridge class separately, and usually without much interaction with each other.

3. You can hide implementation details from the client program much more easily.

THE COMPOSITE PATTERN

Frequently programmers develop systems in which a component may be an individual object or it may represent a collection of objects. The Composite pattern is designed to accommodate both cases. You can use the Composite to build part-whole hierarchies or to construct data representations of trees. In summary, a composite is a collection of objects, any one of which may be either a composite, or just a primitive object. In tree nomenclature, some objects may be nodes with additional branches and some may be leaves.

The problem that develops is the dichotomy between having a single, simple interface to access all the objects in a composite, and the ability to distinguish between nodes and leaves. Nodes have children and can have children added to them, while leaves do not at the moment have children, and in some implementations may be prevented from having children added to them.

Some authors have suggested creating a separate interface for nodes and leaves, where a leaf could have the methods

```
public String getName();
public String getValue();
```

and a node could have the additional methods:

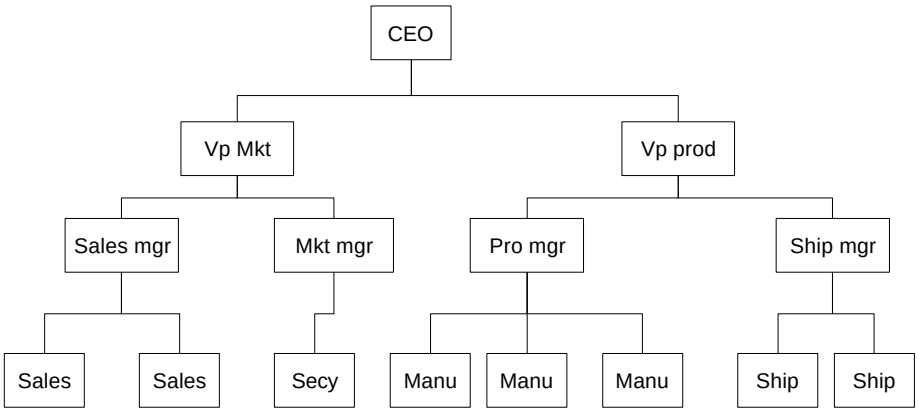
```
public Enumeration elements();
public Node getChild(String nodeName);
public void add(Object obj);
public void remove(Object obj);
```

This then leaves us with the programming problem of deciding which elements will be which when we construct the composite. However, *Design Patterns* suggests that each element should have the *same* interface, whether it is a composite or a primitive element. This is easier to accomplish, but we are left with the question of what the *getChild()* operation should accomplish when the object is actually a leaf.

Java makes this quite easy for us, since every node or leaf can return an Enumeration of the contents of the Vector where the children are stored. If there are no children, the *hasMoreElements()* method returns false at once. Thus, if we simply obtain the Enumeration from each element, we can quickly determine whether it has any children by checking the *hasMoreElements()* method.

An Implementation of a Composite

Let's consider a small company. It may have started with a single person who got the business going. He was, of course, the CEO, although he may have been too busy to think about it at first. Then he hired a couple of people to handle the marketing and manufacturing. Soon each of them hired some additional assistants to help with advertising, shipping and so forth, and they became the company's first two vice-presidents. As the company's success continued, the firm continued to grow until it has the organizational chart we see below:



Now, if the company is successful, each of these company members receives a salary, and we could at any time ask for the cost of any employee to the company. We define the cost as the salary of that person and those of all his subordinates. Here is an ideal example for a composite:

- The cost of an individual employee is simply his salary (and benefits).
- The cost of an employee who heads a department is his salary plus those of all his subordinates.

We would like a single interface that will produce the salary totals correctly whether the employee has subordinates or not.

```
public float getSalaries();
```

At this point, we realize that the idea of all Composites having the same standard interface is probably naïve. We'd prefer that the public methods be related to the kind of class we are actually developing. So rather than have generic methods like *getValue()*, we'll use *getSalaries()*.

The Employee Class

Our Employee class will store the name and salary of each employee, and allow us to fetch them as needed.

```
public class Employee
{
   String name;
   float salary;
   Vector subordinates;
//-------------------------------------
   public Employee(String _name, float _salary)    {
      name = _name;
      salary = _salary;
      subordinates = new Vector();
   }
   //-------------------------------------
   public float getSalary()    {
      return salary;
   }
   //-------------------------------------
   public String getName()    {
      return name;
   }
```

Note that we created a Vector called *subordinates* at the time the class was instantiated. Then, if that employee has subordinates, we can automatically add them to the Vector with the *add* method and remove them with the *remove* method.

```
   public void add(Employee e)    {
         subordinates.addElement(e);
   }
   //-------------------------------------
   public void remove(Employee e)    {
       subordinates.removeElement(e);
   }
```

If you want to get a list of employees of a given supervisor, you can obtain an Enumeration of them directly from the subordinates Vector:

```
public Enumeration elements()    {
      return subordinates.elements();
}
```

The important part of the class is how it returns a sum of salaries for the employee and his subordinates:

```
public float getSalaries()    {
   float sum = salary;                  //this one's salary
 //add in subordinates salaries
 for(int i = 0; i < subordinates.size(); i++) {
   sum +=
```

```
    ((Employee)subordinates.elementAt(i)).getSalaries();
return sum;
}
```

Note that this method starts with the salary of the current Employee, and then calls the *getSalaries()* method on each subordinate. This is, of course, recursive and any employees which themselves have subordinates will be included.

Building the Employee Tree

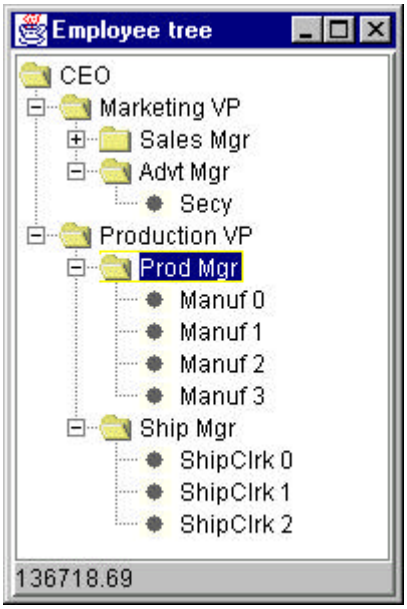
We start by creating a CEO Employee and then add his subordinates and their subordinates as follows:

```
boss = new Employee("CEO", 200000);
boss.add(marketVP =
       new Employee("Marketing VP", 100000));
boss.add(prodVP =
       new Employee("Production VP", 100000));
marketVP.add(salesMgr =
       new Employee("Sales Mgr", 50000));
marketVP.add(advMgr =
       new Employee("Advt Mgr", 50000));
 //add salesmen reporting to Sales Manager
for (int i=0; i<5; i++)
   salesMgr .add(new Employee("Sales "+
       new Integer(i).toString(), 30000.0F
              +(float)(Math.random()-0.5)*10000));
advMgr.add(new Employee("Secy", 20000));

prodVP.add(prodMgr =
       new Employee("Prod Mgr", 40000));
prodVP.add(shipMgr =
       new Employee("Ship Mgr", 35000));
       //add manufacturing staff
for (int i = 0; i < 4; i++)
  prodMgr.add( new Employee("Manuf "+
       new Integer(i).toString(), 25000.0F
              +(float)(Math.random()-0.5)*5000));
 //add shipping clerks
for (int i = 0; i < 3; i++)
  shipMgr.add( new Employee("ShipClrk "+
       new Integer(i).toString(), 20000.0F
              +(float)(Math.random()-0.5)*5000));
```

Once we have constructed this Composite structure, we can load a visual JTree list by starting at the top node and calling the *addNode()* method recursively until all the leaves in each node are accessed:

```
private void addNodes(DefaultMutableTreeNode pnode,
       Employee emp) {
 DefaultMutableTreeNode node;

 Enumeration e = emp.elements();
 while(e.hasMoreElements())
  {
  Employee newEmp = (Employee)e.nextElement();
  node = new DefaultMutableTreeNode(newEmp.getName());
  pnode.add(node);
  addNodes(node, newEmp);
  }
 }
```

The final program display is shown below:



In this implementation, the cost (sum of salaries) is shown in the bottom bar for any employee you click on. This simple computation calls the *getChild()* method recursively to obtain all the subordinates of that employee.

```
public void valueChanged(TreeSelectionEvent evt)
{
//called when employee is selected in tree llist
TreePath path = evt.getPath();
String selectedTerm =
```

```
            path.getLastPathComponent().toString();
//find that employee in the composite
Employee emp = boss.getChild(selectedTerm);
//display sum of salaries of subordinates(if any)
if(emp != null)
   cost.setText(new Float(emp.getSalaries()).toString());
}
```

Restrictions on Employee Classes

It could be that certain employees or job positions are designed so that they never should have subordinates. Assembly workers or salesmen may advance in the company by being named to a new position, but those holding these leaf positions will never have subordinates. In such a case, you may wish to design your Employee class so that you can specify that this is a permanent leaf position. One way to do this is to set a variable which is checked before it allows subordinates to be added. If the position is leaf position, the method returns *false* or throws an exception.

```
public void setLeaf(boolean b)    {
      isLeaf = b;    //if true, do not allow children
   }
//-------------------------------------
public boolean add(Employee e)    {
      if (! isLeaf)
         subordinates.addElement(e);
      return isLeaf;    //false if unsuccessful
   }
```

Consequences of the Composite Pattern

The Composite pattern allows you to define a class hierarchy of simple objects and more complex composite objects so that they appear to be the same to the client program. Because of this simplicity, the client can be that much simpler, since nodes and leaves are handled in the same way.

The Composite pattern also makes it easy for you to add new kinds of components to your collection, as long as they support a similar programming interface. On the other hand, this has the disadvantage of making your system overly general. You might find it harder to restrict certain classes, where this would normally be desirable.

The composite is essentially a singly-linked tree, in which any of the objects may themselves be additional composites. Normally, these objects do not remember their parents and only know their children as an array, hash

table or vector. However, it is perfectly possible for any composite element to remember its parent by including it as part of the constructor:

```
public Employee(Employee _parent, String _name,
                    float _salary)    {
      name = _name;              //save name
      salary = _salary;                //and salary
      parent = _parent;                //and parent
      subordinates = new Vector();
      isLeaf = false;            //allow children
   }
```

This simplifies searching for particular members and moving up the tree when needed.

Other Implementation Issues

Implementing the list in the parent. If there are a very large number of leaves in a composite but only a few nodes, then keeping an empty Vector object in each leaf has some space implications. An alternative approach is to declare all of the objects of type Member, which implements only *getName*() and *getValue*() methods. Then you derive a Node class from Member which implements the *add*, *remove* and *elements* methods. Now only objects that are Node classes can have an enumeration of members. You can check for this in the recursive loop instead of returning empty Vector enumerators.

```
   if(emp instanceof Node)     {
    Enumeration e = emp.elements();
    while(e.hasMoreElements())  {
       Employee newEmp = (Employee)e.nextElement();
             // etc.
   }
```

In most cases it is not clear that the space saving justifies this additional complexity.

Ordering components. In some programs, the order of the components may be important. If that order is somehow different from the order in which they were added to the parent, then the parent must do additional work to return them in the correct order. For example, you might sort the original Vector alphabetically and return the Enumerator to a new sorted vector.

Caching results. If you frequently ask for data which must be computed from a series of child components as we did here with salaries, it may be advantageous to cache these computed results in the parent. However,

unless the computation is relatively intensive and you are quite certain that the underlying data have not changed, this may not be worth the effort.

THE DECORATOR PATTERN

The Decorator pattern provides us with a way to modify the behavior of individual objects without having to create a new derived class. Suppose we have a program that uses eight objects, but three of them need an additional feature. You could create a derived class for each of these objects, and in many cases this would be a perfectly acceptable solution. However, if each of these three objects require *different* modifications, this would mean creating three derived classes. Further, if one of the classes has features of *both* of the other classes, you begin to create a complexity that is both confusing and unnecessary.

For example, suppose we wanted to draw a special border around some of the buttons in a toolbar. If we created a new derived button class, this means that all of the buttons in this new class would always have this same new border, when this might not be our intent.

Instead, we create a Decorator class that *decorates* the buttons. Then we derive any number of specific Decorators from the main Decorator class, each of which performs a specific kind of decoration. In order to decorate a button, the Decorator has to be an object derived from the visual environment, so it can receive paint method calls and forward calls to other useful graphic methods to the object that it is decorating. This is another case where object containment is favored over object inheritance. The decorator is a graphical object, but it contains the object it is decorating. It may intercept some graphical method calls, perform some additional computation and may pass them on to the underlying object it is decorating.

Decorating a CoolButton

Recent Windows applications such as Internet Explorer and Netscape Navigator have a row of flat, unbordered buttons that highlight themselves with outline borders when you move your mouse over them. Some Windows programmers call this toolbar a CoolBar and the buttons CoolButtons. There is no analogous button behavior in the JFC, but we can obtain that behavior by *decorating* a JButton. In this case, we decorate it by drawing plain gray lines over the button borders, erasing them.

Let's consider how to create this Decorator. *Design Patterns* suggests that Decorators should be derived from some general Visual Component class and then every message for the actual button should be forwarded from the

decorator. In Java, this is completely impractical, because there are literally hundreds of method calls in the base JComponent class that we would have to reimplement. Instead, while we will derive our Decorator from the JComponent class, we will use its container properties to forward all method calls to the button it will contain.

Design Patterns suggests that classes such as Decorator should be abstract classes and that you should derive all of your actual working (or concrete) decorators from the abstract class. In this Java implementation, this is scarcely necessary since the base Decorator class has no public methods at all other than the constructor, since all of them are methods of JComponent itself.

```
public class Decorator extends Jcomponent {
   public Decorator(JComponent c)    {
      setLayout(new BorderLayout());
       //add component to container
      add("Center", c);
   }
}
```

Now, let's look at how we could implement a CoolButton. All we really need to do is to draw the button as usual from the base class, and then draw gray lines around the border to remove the button highlighting.

```
//this class decorates a CoolButton so that
//the borders are invisible when the mouse
//is not over the button
public class CoolDecorator extends Decorator
{
   boolean mouse_over;    //true when mouse over button
   JComponent thisComp;

   public CoolDecorator(JComponent c)
   {
      super(c);
      mouse_over = false;
      thisComp = this;      //save this component
      //catch mouse movements in inner class
      c.addMouseListener(new MouseAdapter()
      {
      public void mouseEntered(MouseEvent e) {
         mouse_over=true;     //set flag when mouse over
         thisComp.repaint();
         }
      public void mouseExited(MouseEvent e) {
         mouse_over=false;    //clear if mouse not over
         thisComp.repaint();
         }
```

```
      });

   }
   //paint the button
   public void paint(Graphics g)
   {
      super.paint(g);      //first draw the parent button
      if(! mouse_over) {
         //if the mouse is not over the button
         //erase the borders
         Dimension size = super.getSize();
         g.setColor(Color.lightGray);
         g.drawRect(0, 0, size.width-1, size.height-1);
         g.drawLine(size.width-2, 0, size.width-2,
                           size.height-1);
         g.drawLine(0, size.height-2, size.width-2,
                           size.height-2);
      }
   }
}
```

Using a Decorator

Now that we've written a CoolDecorator class, how do we use it? We simply create an instance of the CoolDecorator and pass it the button it is to decorate. We can do all of this right in the constructor. Let's consider a simple program with two CoolButtons and one ordinary JButton. We create the layout as follows:

```
super ("Deco Button");
JPanel jp = new JPanel();

getContentPane().add(jp);
jp.add( new CoolDecorator(new JButton("Cbutton")));
jp.add( new CoolDecorator(new JButton("Dbutton")));
jp.add(Quit = new JButton("Quit"));
Quit.addActionListener(this);
```

This program is shown below, with the mouse hovering over one of the buttons.

Now that we see how a single decorator works, what about multiple decorators? It could be that we'd like to decorate our CoolButtons with another decoration, say, a red diagonal line. Since the argument to any Decorator is just a JComponent, we could create a new decorator with a decorator as its argument.

Let's consider the SlashDecorator, which draws that diagonal red line:

```
public class SlashDecorator extends Decorator
{
   int x1, y1, w1, h1;     //saved size and posn

   public SlashDecorator(JComponent c)    {
      super(c);
   }
//-----------------------------------------------
   public void setBounds(int x, int y, int w, int h)    {
      x1 = x; y1= y;             //save coordinates
      w1 = w; h1 = h;
      super.setBounds(x, y, w, h);
   }
//-----------------------------------------------
   public void paint(Graphics g)    {
      super.paint(g);            //draw button
      g.setColor(Color.red);     //set color
      g.drawLine(0, 0, w1, h1); //draw red line
   }
}
```

Here we save the size and position of the button when it is created, and then use those saved values to draw the diagonal line.

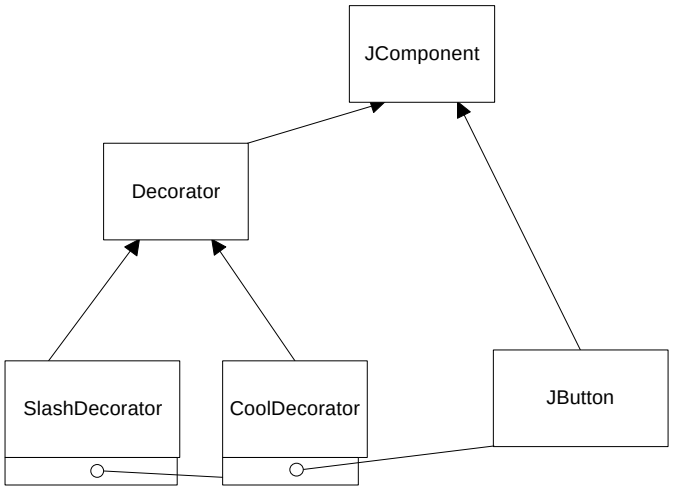
You can create the JButton with these two decorators by just calling one and then the other:

```
jp.add(new SlashDecorator(
       new CoolDecorator(new JButton("Dbutton"))));
```

This gives us a final program that displays the two buttons like this:

Inheritance Order

Some people find the order of inheritance in Decorators confusing, because we are surrounding a button with a decorator that inherits from a JComponent. We illustrate this inheritance tree below.



A JButton is a child of JComponent, and is encapsulated in a Decorator, which not only is a child of JComponent but encapsulates one as well. The JComponent it encapsulates is, in this case, a JButton.

Decorating Borders in Java

One problem with this particular implementation of Decorators is that it is not easy to expand the size of the component you are decorating, because you add the component to a container and allow it to fill the container completely. If you attempt to draw lines outside the area of this component, they are clipped by the graphics procedure and not drawn at all.

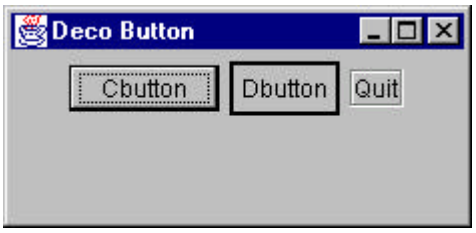
The JFC provides its own series of Border objects that are a kind of decorators. Like a Decorator pattern, you can add a new Border object to any JComponent, and there also is a way to add several borders. However, unlike the Decorator pattern, it is not a JComponent and you do not have the flexibility to intercept and change specific events.

The JFC defines several standard border classes:

BevelBorder(*n*)	Simple 2-line bevel, can be LOWERED or RAISED

CompoundBorder (*inner, outer)*	Allows you to add 2 borders
EmptyBorder(*top, left, bottom, right)*	Blank border width specified on each side.
EtchedBorder	Creates etched norder.
LineBorder(*width, color)*	Creates simple line border,
MatteBorder	Creates a matte border of a solid color or a tiled icon.
SofBeveledBorder	Creates beveled border with rounded corners.
TitledBorder	Creates a border containing a title. Use this to surround and label a JPanel.

These borders are simple to use, in conjunction with the *setBorder* method of each JComponent. The illustration below shows a normal JButton with a 2-pixel solid line border, combined with a 4-pixel EmptyBorder and an EtchedBorder.



This was created with the following simple code:

```
getContentPane().add(jp);
jp.add( Cbutton = new JButton("Cbutton"));
jp.add( Dbutton = new JButton("Dbutton"));
EmptyBorder ep = new EmptyBorder(4,4,4,4);
LineBorder lb = new LineBorder(Color.black, 2);
Dbutton.setBorder(new CompoundBorder(lb, ep));
jp.add(Quit = new JButton("Quit"));
EtchedBorder eb = new EtchedBorder();
Quit.addActionListener(this);
Quit.setBorder(eb);
```

One drawback of these Border objects is that they replace the default Insets values that determine the spacing around the component. Note that we had to add a 4-pixel EmptyBorder to the Dbutton to make it similar in size to the CButton. We did not do this for the Quit button, and it is therefore substantially smaller than the others.

Non-Visual Decorators

Decorators, of course, are not limited to objects that enhance visual classes. You can add or modify the methods of any object in a similar fashion. In fact, non-visual objects are usually easier to decorate, because there are usually fewer methods to intercept and forward.

While coming up with a simple example is difficult, a series of Decorators do occur naturally in the java.io classes. Note the following in the Java documentation:

> *The class FilterInputStream itself simply overrides all methods of InputStream with versions that pass all requests to the underlying input stream. Subclasses of FilterInputStream may further override some of these methods as well as provide additional methods and fields.*

The FilterInputStream class is thus a Decorator that can be wrapped around any input stream class. It is essentially an abstract class that doesn't do any processing, but provides a layer where the relevant methods have been duplicated. It normally forwards these method calls to the enclosed parent stream class.

The interesting classes derived from FilterInputStream include

BufferedInputStream	Adds buffering to stream so that every call does not cause I/O to occur.
CheckedInputStream	Maintains a checksum of bytes as they are read
DataInputStream	Reads primitive types (Long, Boolean, Float, etc.) from the input stream.
DigestInputStream	Computes a MessageDigest of any input stream.
InflaterInputStream	Implements methods for uncompressing data.
PushbackInputStream	Provides a buffer where data can be "unread," if during parsing you discover you need to back up.

These decorators can be nested, so that a pushback, buffered input stream is quite possible.

Decorators, Adapters and Composites

There is an essential similarity among these classes that you may have recognized. Adapters also seem to "decorate" an existing class. However, their function is to change the interface of one or more classes to one that is more convenient for a particular program. Decorators add methods to particular instances of classes, rather than to all of them. You could also imagine that a composite consisting of a single item is essentially a decorator. Once again, however, the intent is different

Consequences of the Decorator Pattern

The Decorator pattern provides a more flexible way to add responsibilities to a class than by using inheritance, since it can add these responsibilities to selected instances of the class. It also allows you to customize a class without creating subclasses high in the inheritance hierarchy. *Design Patterns* points out two disadvantages of the Decorator pattern One is that a Decorator and its enclosed component are not identical. Thus tests for object type will fail. The second is that Decorators can lead to a system with "lots of little objects" that all look alike to the programmer trying to maintain the code. This can be a maintenance headache.

Decorator and Façade evoke similar images in building architecture, but in design pattern terminology, the Façade is a way of hiding a complex system inside a simpler interface, while Decorator adds function by wrapping a class. We'll take up the Façade next.

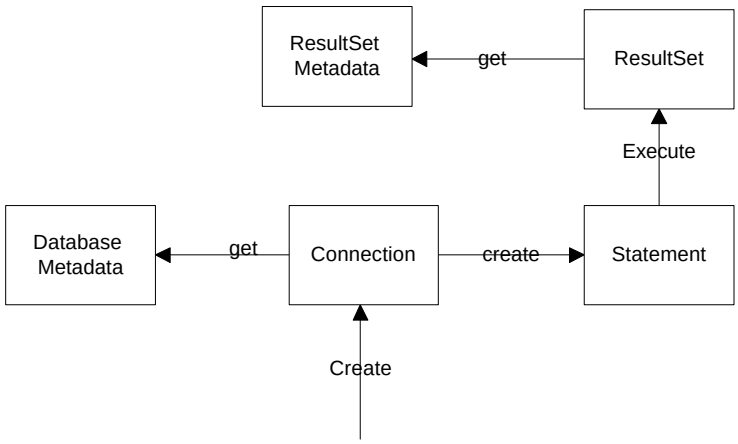
THE FAÇADE PATTERN

Frequently, as your programs evolve and develop, they grow in complexity. In fact, for all the excitement about using design patterns, these patterns sometimes generate so many classes that it is difficult to understand the program's flow. Furthermore, there may be a number of complicated subsystems, each of which has its own complex interface.

The Façade pattern allows you to simplify this complexity by providing a simplified interface to these subsystems. This simplification may in some cases reduce the flexibility of the underlying classes, but usually provides all the function needed for all but the most sophisticated users. These users can still, of course, access the underlying classes and methods.

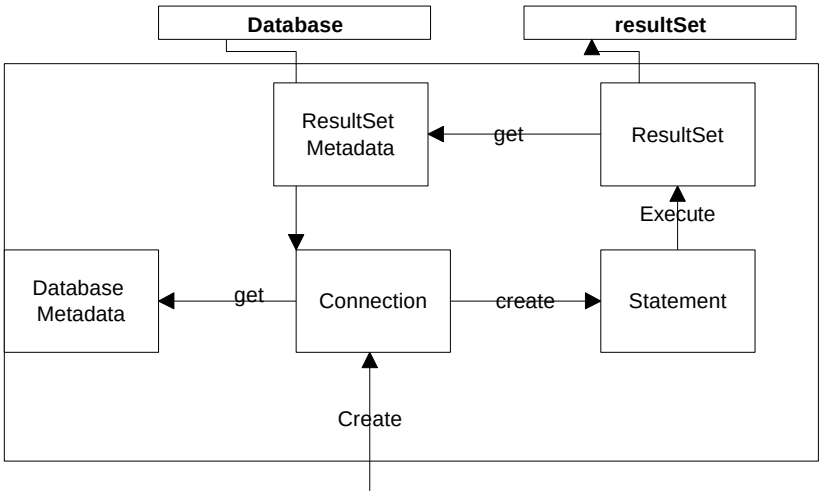
Fortunately, we don't have to write a complex system to provide an example of where a Facade can be useful. Java provides a set of classes that connect to databases using an interface called JDBC. You can connect to any database for which the manufacturer has provided a JDBC connection class -- almost every database on he market. Some databases have direct connections using JDBC and a few allow connection to ODBC driver using the JDBC-ODBC bridge class.

These database classes in the java.sql package provide an excellent example of a set of quite low level classes that interact in a convoluted manner, as shown below.



To connect to a database, you use an instance of the Connection class. Then, to find out the names of the database tables and fields, you need

to get an instance of the DatabaseMetadata class from the Connection. Next, to issue a query, you compose the SQL query string and use the Connection to create a Statement class. By executing the statement, you obtain a ResultSet class, and to find out the names of the column rows in that ResultSet, you need to obtain an instance of the ResultsetMetadata class. Thus, it can be quite difficult to juggle all of these classes and since most of the calls to their methods throw Exceptions, the coding can be messy at least.



However, by designing a Façade consisting of a Database class and a resultSet class (note the lowercase "r"), we can build a much more usable system.

Building the Façade Classes

Let's consider how we connect to a database. We first must load the database driver:

```
try{Class.forName(driver);}  //load the Bridge driver
    catch (Exception e)
    {System.out.println(e.getMessage());}
```

and then use the Connection class to connect to a database. We also obtain the database metadata to find out more about the database:

```
try {con = DriverManager.getConnection(url);
    dma =con.getMetaData();   //get the meta data
    }
    catch (Exception e)
    {System.out.println(e.getMessage());}
```

If we want to list the names of the tables in the database, we then need to call the *getTables* method on the database metadata class, which returns a ResultSet object. Finally, to get the list of names we have to iterate through that object, making sure that we obtain only user table names, and exclude internal system tables.

```
Vector tname = new Vector();
try {
      results = new resultSet(dma.getTables(catalog,
                          null, "%", types));
    }
    catch (Exception e) {System.out.println(e);}
    while (results.hasMoreElements())
        tname.addElement(
             results.getColumnValue("TABLE_NAME"));
```

This quickly becomes quite complex to manage, and we haven't even issued any queries yet.

One simplifying assumption we can make is that the exceptions that all these database class methods throw do not need complex handling. For the most part, the methods will work without error unless the network connection to the database fails. Thus, we can wrap all of these methods in classes in which we simply print out the infrequent errors and take no further action.

This makes it possible to write two simple enclosing classes which contain all of the significant methods of the Connection, ResultSet, Statement and Metadata classes. These are the Database class:

```
Class Database {
 public Database(String driver)()//constructor
 public void Open(String url, String cat);
 public String[] getTableNames();
 public String[] getColumnNames(String table);
 public String getColumnValue(String table,
             String columnName);
   public String getNextValue(String columnName);
   public resultSet Execute(String sql);
}
```

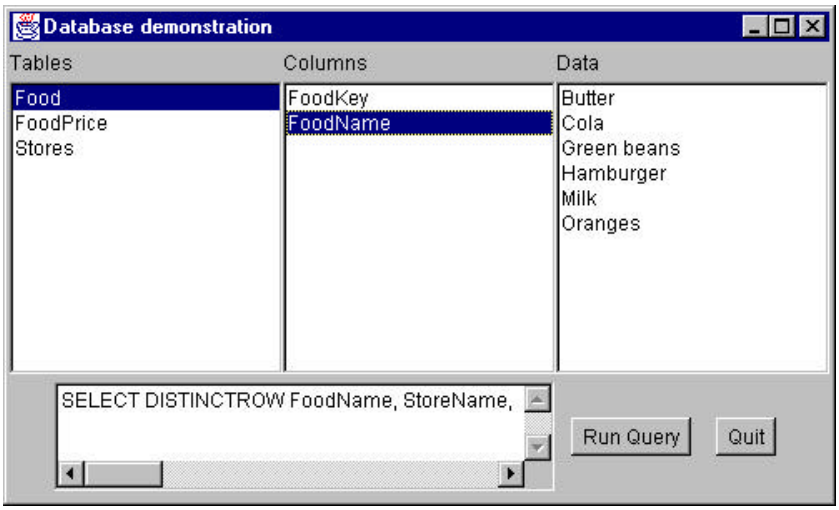
and the resultSet class:

```
class resultSet
{
   public resultSet(ResultSet rset)     //constructor
```

```
   public String[] getMetaData();
   public boolean hasMoreElements();
   public String[] nextElement();
   public String getColumnValue(String columnName);
   public String getColumnValue(int i);
}
```

These simple classes allow us to write a program for opening a database, displaying its table names, column names and contents, and running a simple SQL query on the database.

The dbFrame.java program accesses a simple database containing food prices at 3 local markets:



Clicking on a table name shows you the column names and clicking on a column name shows you the contents of that column. If you click on Run Query, it displays the food prices sorted by store for oranges:

This program starts by connecting to the database and getting a list of the table names:

```
db = new Database("sun.jdbc.odbc.JdbcOdbcDriver");
db.Open("jdbc:odbc:Grocery prices", null);
String tnames[] = db.getTableNames();
loadList(Tables, tnames);
```

Then clicking on one of the lists runs a simple query for table column names or contents:

```
public void itemStateChanged(ItemEvent e)    {
//get list box selection
      Object obj = e.getSource();
      if (obj == Tables)
         showColumns();
      if (obj == Columns)
         showData();
}
//-----------------------------------
private void showColumns()  {
//display column names
String cnames[] =
       db.getColumnNames(Tables.getSelectedItem());
loadList(Columns, cnames);
}
//-----------------------------------
private void showData() {
//display column contents
String colname = Columns.getSelectedItem();
String colval =
       db.getColumnValue(Tables.getSelectedItem(),
              colname);
Data.removeAll();          //clear list box
colval = db.getNextValue(Columns.getSelectedItem());

while (colval.length()>0)   {
       //load list box
   Data.add(colval);
   colval = db.getNextValue(Columns.getSelectedItem());
  }
}
```

Consequences of the Façade

The Façade pattern shields clients from complex subsystem components and provides a simpler programming interface for the general user. However, it does not prevent the advanced user from going to the deeper, more complex classes when necessary.

In addition, the Façade allows you to make changes in the underlying subsystems without requiring changes in the client code, and reduces compilation dependencies.

THE FLYWEIGHT PATTERN

There are cases in programming where it seems that you need to generate a very large number of small class instances to represent data. Sometimes you can greatly reduce the number of different classes that you need to instantiate if you can recognize that the instances are fundamentally the same except for a few parameters. If you can move those variables outside the class instance and pass them in as part of a method call, the number of separate instances can be greatly reduced.

The Flyweight design pattern provides an approach for handling such classes. It refers to the instance's *intrinsic* data that makes the instance unique, and the *extrinsic* data which is passed in as arguments. The Flyweight is appropriate for small, fine-grained classes like individual characters or icons on the screen. For example, if you are drawing a series of icons on the screen in a folder window, where each represents a person or data file, it does not make sense to have an individual class instance for each of them that remembers the person's name and the icon's screen position. Typically these icons are one of a few similar images and the position where they are drawn is calculated dynamically based on the window's size in any case.

In another example in *Design Patterns*, each character in a font is represented as a single instance of a character class, but the positions where the characters are drawn on the screen are kept as external data so that there needs to be only one instance of each character, rather than one for each appearance of that character.

Discussion

Flyweights are sharable instances of a class. It might at first seem that each class is a Singleton, but in fact there might be a small number of instances, such as one for every character, or one for every icon type. The number of instances that are allocated must be decided as the class instances are needed, and this is usually accomplished with a FlyweightFactory class. This factory class usually *is* a Singleton, since it needs to keep track of whether or not a particular instance has been generated yet. It then either returns a new instance or a reference to one it has already generated.

To decide if some part of your program is a candidate for using Flyweights, consider whether it is possible to remove some data from the class and make it extrinsic. If this makes it possible to reduce greatly the

number of different class instances your program needs to maintain, this might be a case where Flyweights will help.

Example Code

Suppose we want to draw a small folder icon with a name under it for each person in a an organization. If this is a large organization, there could be a large number of such icons, but they are actually all the same graphical image. Even if we have two icons, one for “is Selected” and one for “not Selected” the number of different icons is small. In such a system, having an icon object for each person, with its own coordinates, name and selected state is a waste of resources.

Instead, we’ll create a FolderFactory that returns either the selected or the unselected folder drawing class, but does not create additional instances once one of each has been created. Since this is such a simple case, we just create them both at the outset and then return one or the other:

```
class FolderFactory
{
   Folder unSelected, Selected;
   public FolderFactory()
      {
      Color brown = new Color(0x5f5f1c);
      Selected =  new Folder(brown);
      unSelected = new Folder(Color.yellow);
      }
//-------------------------------
   public Folder getFolder(boolean isSelected)
   {
      if (isSelected)
         return Selected;
      else
         return unSelected;
   }
}
```

For cases where more instances could exist, the factory could keep a table of the ones it had already created and only create new ones if they weren’t already in the table.

The unique thing about using Flyweights, however, is that we pass the coordinates and the name to be drawn into the folder when we draw it. These coordinates are the extrinsic data that allow us to share the folder objects, and in this case create only two instances. The complete folder class shown below simply creates a folder instance with one background color or

the other and has a public Draw method that draws the folder at the point you specify.

```
class Folder extends JPanel
{
   private Color color;
   final int W = 50, H = 30;
   public Folder(Color c)
   {
      color = c;
   }
//-------------------------------
public void Draw(Graphics g, int tx, int ty, String name)
   {
    g.setColor(Color.black);            //outline
    g.drawRect(tx, ty, W, H);
    g.drawString(name, tx, ty + H+15);  //title

    g.setColor(color);                  //fill rectangle
    g.fillRect(tx+1, ty+1, W-1, H-1);

    g.setColor(Color.lightGray);        //bend line
    g.drawLine(tx+1, ty+H-5, tx+W-1, ty+H-5);

    g.setColor(Color.black);            //shadow lines
    g.drawLine(tx, ty+H+1, tx+W-1, ty+H+1);
    g.drawLine(tx+W+1, ty, tx+W+1, ty+H);

    g.setColor(Color.white);            //highlight lines
    g.drawLine(tx+1, ty+1, tx+W-1, ty+1);
    g.drawLine(tx+1, ty+1, tx+1, ty+H-1);
   }
}
```

To use a Flyweight class like this, your main program must calculate the position of each folder as part of its paint routine and then pass the coordinates to the folder instance. This is actually rather common, since you need a different layout depending on the window's dimensions, and you would not want to have to keep telling each instance where its new location is going to be. Instead, we compute it dynamically during the paint routine.

Here we note that we could have generated an array or Vector of folders at the outset and simply scan through the array to draw each folder. Such an array is not as wasteful as a series of different instances because it is actually an array of references to one of only two folder instances. However, since we want to display one folder as "selected," and we would like to be

able to change which folder is selected dynamically, we just use the FolderFactory itself to give us the correct instance each time:

```
public void paint(Graphics g)
  {
     Folder f;
     String name;

     int j = 0;      //count number in row
     int row = Top;  //start in upper left
     int x = Left;

     //go through all the names and folders
     for (int i = 0; i< names.size(); i++)
     {
     name = (String)names.elementAt(i);
     if(name.equals(selectedName))
        f = fact.getFolder(true);
     else
        f = fact.getFolder(false);
     //have that folder draw itself at this spot
     f.Draw(g, x, row, name);

     x = x + HSpace;          //change to next posn
     j++;
     if (j >= HCount)         //reset for next row
        {
        j = 0;
        row += VSpace;
        x = Left;
        }
     }
  }
```

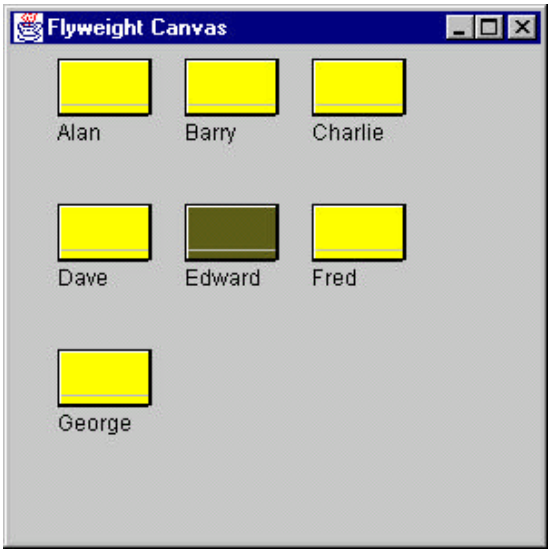
Selecting A Folder

Since we have two folder instances, that we termed selected and unselected, we'd like to be able to select folders by moving the mouse over them. In the paint routine above, we simply remember the name of the folder which was selected and ask the factory to return a "selected' folder for it. Since the folders are not individual instances, we can't listen for mouse motion within each folder instance. In fact, even if we did listen within a folder, we'd have to have a way to tell the other instances to deselect themselves.

Instead, we check for mouse motion at the window level and if the mouse is found to be within a Rectangle, we make that corresponding name the selected name. This allows us to just check each name when we redraw and create a selected folder instance where it is needed:

```
public void mouseMoved(MouseEvent e)
  {
  int j = 0;      //count number in row
  int row = Top;  //start in upper left
  int x = Left;

  //go through all the names and folders
  for (int i = 0; i< names.size(); i++)
  {
  //see if this folder contains the mouse
  Rectangle r = new Rectangle(x,row,W,H);
  if (r.contains(e.getX(), e.getY()))
   {
   selectedName=(String)names.elementAt(i);
   repaint();
   }
  x = x + HSpace;          //change to next posn
  j++;
  if (j >= HCount)         //reset for next row
     {
     j = 0;
     row += VSpace;
     x = Left;
     }
  }
```

The display program for 10 named folders is shown below:



Flyweight Uses in Java

Flyweights are not frequently used at the application level in Java. They are more of a system resource management technique, used at a lower level than Java. However, it is useful to recognize that this technique exists so you can use it if you need it.

One place where we have already seen the Flyweight is in the cell renderer code we use for tables and list boxes. Usually the cell renderer is just a JLabel, but there may be two or three types of labels or renderers for different colors or fonts. However, there are far fewer renderers than there are cells in the table or list.

Some objects within the Java language could be implemented under the covers as Flyweights. For example, if there are two instances of a String constant with identical characters, they could refer to the same storage location. Similarly, it might be that two Integer or Float objects that contain the same value could be implemented as Flyweights, although they probably are not. To prove the absence of Flyweights here, just run the following code:

```
Integer five = new Integer(5);
Integer myfive = new Integer(5);
System.out.println(five == myfive);

String fred=new String("fred");
String fred1 = new String("fred");
System.out.println(fred == fred1);
```

Both cases print out “false.” However it is useful to note that you can easily determine that you are dealing with two identical instances of a Flyweight by using the “==” operator. It compares actual object references (memory addresses) rather than the “equals” operator which will probably be slower if it is implemented at all.

Sharable Objects

The *Smalltalk Companion* points out that sharable objects are much like Flyweights, although the purpose is somewhat different. When you have a very large object containing a lot of complex data, such as tables or bitmaps, you would want to minimize the number of instances of that object. Instead, in such cases, you’d return one instance to every part of the program that asked for it and avoid creating other instances.

A problem with such sharable objects occurs when one part of a program wants to change some data in a shared object. You then must decide

whether to change the object for all users, prevent any change, or create a new instance with the changed data. If you change the object for every instance, you may have to notify them that the object has changed.

Sharable objects are also useful when you are referring to large data systems outside of Java, such as databases. The Database class we developed above in the Façade pattern could be a candidate for a sharable object. We might not want a number of separate connections to the database from different program modules, preferring that only one be instantiated. However, should several modules in different threads decide to make queries simultaneously, the Database class might have to queue the queries or spawn extra connections.

THE PROXY PATTERN

The Proxy pattern is used when you need to represent a complex object by a simpler one. If creating an object is expensive in time or computer resources, Proxy allows you to postpone this creation until you need the actual object. A Proxy usually has the same methods as the object it represents, and once the object is loaded, it passes on the method calls from the Proxy to the actual object.

There are several cases where a Proxy can be useful:

1. If an object, such as a large image, takes a long time to load.
2. If the object is on a remote machine and loading it over the network may be slow, especially during peak network load periods.
3. If the object has limited access rights, the proxy can validate the access permissions for that user.

Proxies can also be used to distinguish between requesting an instance of an object and the actual need to access it. For example, program initialization may set up a number of objects which may not all be used right away. In that case, the proxy can load the real object only when it is needed.

Let's consider the case of a large image that a program needs to load and display. When the program starts, there must be some indication that an image is to be displayed so that the screen lays out correctly, but the actual image display can be postponed until the image is completely loaded. This is particularly important in programs such as word processors and web browsers that lay out text around the images even before the images are available.

An image proxy can note the image and begin loading it in the background, while drawing a simple rectangle or other symbol to represent the image's extent on the screen before it appears. The proxy can even delay loading the image at all until it receives a paint request, and only then begin the process.

Sample Code

In this example program, we create a simple program to display an image on a JPanel when it is loaded. Rather than loading the image directly, we use a class we call ImageProxy to defer loading and draw a rectangle around the image area until loading is completed.

```
public class ProxyDisplay extends JxFrame
{
  public ProxyDisplay()
  {
  super("Display proxied image");
  JPanel p = new JPanel();
  getContentPane().add(p);
  p.setLayout(new BorderLayout());
  ImageProxy image = new ImageProxy(this, "elliott.jpg",
                                          321,271);
  p.add("Center", image);
  setSize(400,400);
  setVisible(true);
  }
```

Note that we create the instance of the ImageProxy just as we would have for an Image, and that we add it to the enclosing JPanel as we would an actual image.

The ImageProxy class sets up the image loading and creates a MediaTracker object to follow the loading process within the constructor:

```
public ImageProxy(JFrame f, String filename,
                  int w, int h)
{
height = h;
width = w;
frame = f;

tracker = new MediaTracker(f);
img = Toolkit.getDefaultToolkit().getImage(filename);
tracker.addImage(img, 0);     //watch for image loading

imageCheck = new Thread(this);
imageCheck.start();           //start 2nd thread monitor

//this begins actual image loading
try{
    tracker.waitForID(0,1);
 }
catch(InterruptedException e){}
}
```

The *waitForID* method of the MediaTracker actually initiates loading. In this case, we put in a minimum wait time of 1 msec so that we can minimize apparent program delays.

The constructor also creates a separate thread *imageCheck* that checks the loading status every few milliseconds, and starts that thread running.

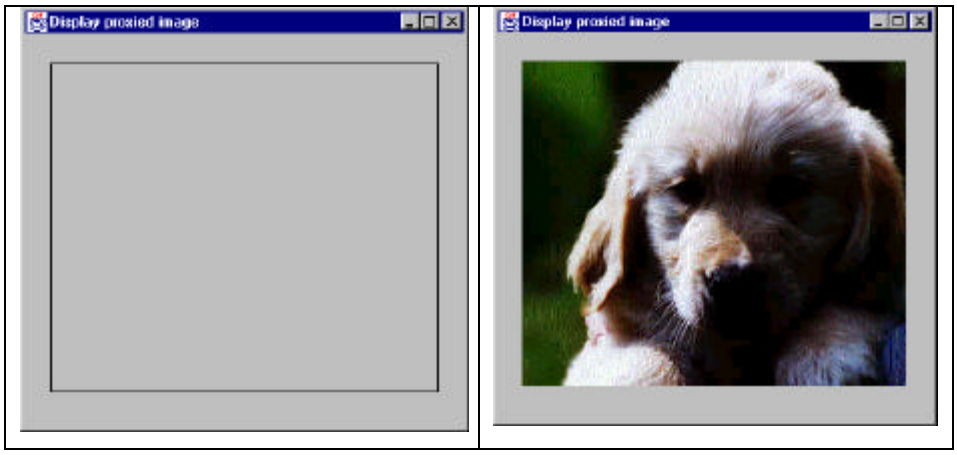
```
public void run()
   {
   //this thread monitors image loading
   //and repaints when the image is done
   try{
   Thread.sleep(1000);
   while(! tracker.checkID(0))
      Thread.sleep(1000);
   }
   catch(Exception e){}
   repaint();
   }
```

For the purposes of this illustration program, we slowed the polling time down to 1 second so you can see the program draw the rectangle and then refresh the final image.

Finally, the Proxy is derived from the JPanel component, and therefore, naturally has a *paint* method. In this method, we draw a rectangle if the image is not loaded. If the image has been loaded, we erase the rectangle and draw the image instead.

```
public void paint(Graphics g)
   {
    if (tracker.checkID(0))
      {
      height = img.getHeight(frame);   //get height
      width = img.getWidth(frame);     //and width
      g.setColor(Color.lightGray);     //erase box
      g.fillRect(0,0, width, height);
      g.drawImage(img, 0, 0, frame);   //draw image
      }
   else
      {
      //draw box outlining image if not loaded yet
      g.drawRect(0, 0, width-1, height-1);
      }
   }
```

The program's two states are illustrated below.



Copy-on-Write

You can also use proxies is to keep copies of large objects that may or may not change. If you create a second instance of an expensive object, a Proxy can decide there is no reason to make a copy yet. It simply uses the original object. Then, if the program makes a change in the new copy, the Proxy can copy the original object and make the change in the new instance. This can be a great time and space saver when objects do not always change after they are instantiated.

Comparison with Related Patterns

Both the Adapter and the Proxy constitute a thin layer around an object. However, the Adapter provides a different interface for an object, while the Proxy provides the same interface for the object, but interposes itself where it can save processing effort.

A Decorator also has the same interface as the object it surrounds, but its purpose is to add additional (usually visual) function to the original object. A proxy, by contrast, controls access to the contained class.

SUMMARY OF STRUCTURAL PATTERNS

In this chapter we have seen the

- The **Adapter** pattern, used to change the interface of one class to that of another one.
- The **Bridge** pattern, intended to keep the interface to your client program constant while allowing you to change the actual kind of class you display or use. You can then change the interface and the underlying class separately.
- The **Composite** pattern, a collection of objects, any one of which may be either itself a Composite, or just a primitive object.
- The **Decorator** pattern, a class that surrounds a given class, adds new capabilities to it, and passes all the unchanged methods to the underlying class.
- The **Façade** pattern, which groups a complex object hierarchy and provides a new, simpler interface to access those data.
- The **Flyweight** pattern, which provides a way to limit the proliferation of small, similar class instances by moving some of the class data outside the class and passing it in during various execution methods.
- The **Proxy** pattern, which provides a simple place-holder class for a more complex class which is expensive to instantiate.

Behavioral Patterns

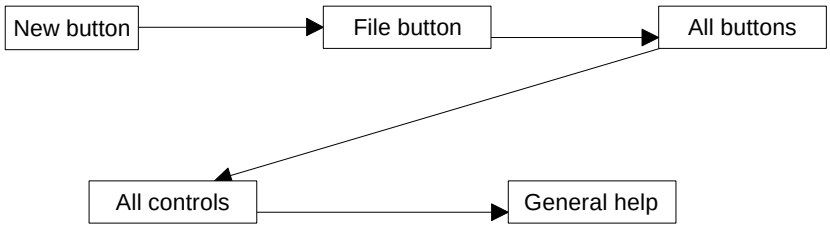
Behavioral patterns are those patterns that are most specifically concerned with communication between objects. In this chapter, we'll see that:

- The Observer pattern defines the way a number of classes can be notified of a change,
- The Mediator defines how communication between classes can be simplified by using another class to keep all classes from having to know about each other.
- The Chain of Responsibility allows an even further decoupling between classes, by passing a request between classes until it is recognized.
- The Template pattern provides an abstract definition of an algorithm, and
- The Interpreter provides a definition of how to include language elements in a program.
- The Strategy pattern encapsulates an algorithm inside a class,
- The Visitor pattern adds function to a class,
- The State pattern provides a memory for a class's instance variables.
- The Command pattern provides a simple way to separate execution of a command from the interface environment that produced it, and
- The Iterator pattern formalizes the way we move through a list of data within a class.

CHAIN OF RESPONSIBILITY

The Chain of Responsibility pattern allows a number of classes to attempt to handle a request, without any of them knowing about the capabilities of the other classes. It provides a loose coupling between these classes; the only common link is the request that is passed between them. The request is passed along until one of the classes can handle it.

One example of such a chain pattern is a Help system, where every screen region of an application invites you to seek help, but in which there are window background areas where more generic help is the only suitable result. When you select an area for help, that visual control forwards its ID or name to the chain. Suppose you selected the "New" button. If the first module can handle the New button, it displays the help message. If not, it forwards the request to the next module. Eventually, the message is forwarded to an "All buttons" class that can display a general message about how buttons work. If there is no general button help, the message is forwarded to the general help module that tells you how the system works in general. If that doesn't exist, the message is lost and no information is displayed. This is illustrated below.



There are two significant points we can observe from this example; first, the chain is organized from most specific to most general, and that there is no guarantee that the request will produce a response in all cases.

Applicability

We use the Chain of Responsibility when

- You have more than one handler that can handle a request and there is no way to know which handler to use. The handler must be determined automatically by the chain.

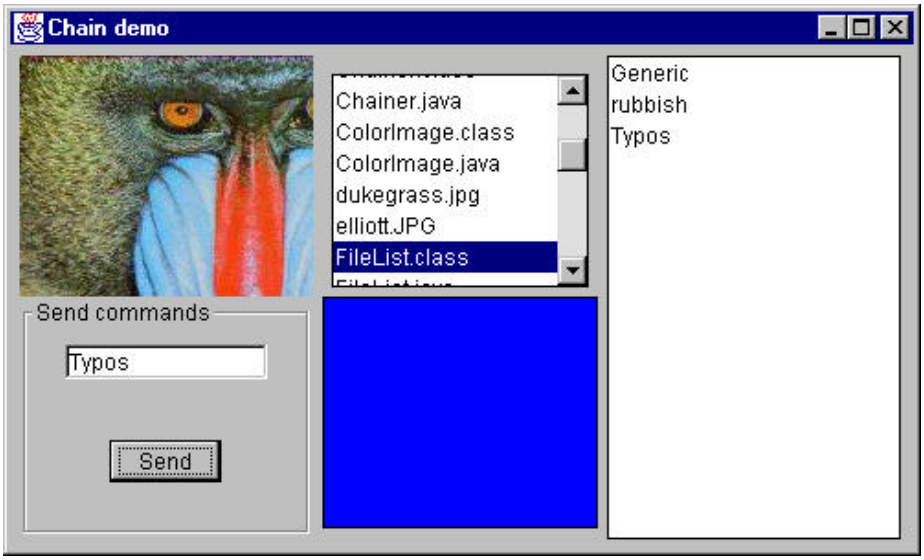
- You want to issue a request to one of several objects without specifying which one explicitly.
- You want to be able to modify the set of objects dynamically that can handle requests.

Sample Code

Let's consider a simple system for display the results of typed in requests. These requests can be

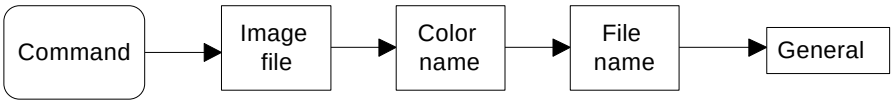
- Image filenames
- General filenames
- Colors
- Other commands

In three cases, we can display a concrete result of the request, and in the last case, we can only display the request text itself.



In the above example system, we type in "Mandrill" and see a display of the image Mandrill.jpg. Then, we type in "FileList" and that filename is highlighted in the center list box. Next, we type in "blue" and that color is displayed in the lower center panel. Finally, if we type in anything that is

neither a filename nor a color, that text is displayed in the final, right-hand list box. This is shown below:



To write this simple chain of responsibility program, we start with an abstract Chain class:

```
public interface Chain
{
public abstract void addChain(Chain c);
public abstract void sendToChain(String mesg);
public Chain getChain();
}
```

The *addChain* method adds another class to the chain of classes. The *getChain* method returns the current class to which messages are being forwarded. These two methods allow us to modify the chain dynamically and add additional classes in the middle of an existing chain. The *sendToChain* method forwards a message to the next object in the chain.

Our Imager class is thus derived from JPanel and implements our Chain interface. It takes the message and looks for ".jpg" files with that root name. If it finds one, it displays it.

```
public class Imager extends JPanel
       implements Chain
{
   private Chain nextChain;
   private Image img;
   private boolean loaded;

public void addChain(Chain c) {
   nextChain = c;    //next in chain of resp
}
//----------------------------------------
public void sendToChain(String mesg)
{
   //if there is a JPEG file with this root name
   //load it and display it.
   if (findImage(mesg))
      loadImage(mesg + ".jpg");
   else
   //Otherwise, pass request along chain
       nextChain.sendToChain(mesg);
```

```
}
//---------------------------------------
public Chain getChain() {
   return nextChain;
}
//---------------------------------------
public void paint(Graphics g) {
   if (loaded)       {
      g.drawImage(img, 0, 0, this);
      }
}
```

In a similar fashion, the ColorImage class simply interprets the message as a color name and displays it if it can. This example only interprets 3 colors, but you could implement any number:

```
public void sendToChain(String mesg) {
 Color c = getColor(mesg);
 if(c != null)    {
   setBackground(c);
   repaint();
   }
 else  {
    if (nextChain != null)
       nextChain.sendToChain(mesg);
 }
}
//-----------------------------------
private Color getColor(String mesg) {
   String lmesg = mesg.toLowerCase();
   Color c = null;

   if(lmesg.equals("red"))
      c = Color.red;
   if(lmesg.equals("blue"))
      c = Color.blue;
   if(lmesg.equals("green"))
      c= Color.green;
   return c;
}
```

The List Boxes

Both the file list and the list of unrecognized commands are JList boxes. Since we developed an adapter JawtList in the previous chapter to give JList a simpler interface, we'll use that adapter here. The RestList class is the end of the chain, and any command that reaches it is simply displayed in the list. However, to allow for convenient extension, we are able to forward the message to other classes as well.

```
public class RestList extends JawtList
        implements Chain
{
private Chain nextChain = null;
//-------------------------------------
  public RestList()    {
   super(10);     //arg to JawtList
   setBorder(new LineBorder(Color.black));
  }
  //-------------------------------------
  public void addChain(Chain c) {
    nextChain = c;
  }
  //-------------------------------------
  public void sendToChain(String mesg) {
    add(mesg);        //this is the end of the chain
    repaint();
    if(nextChain != null)
      nextChain.sendToChain(mesg);
  }
  //-------------------------------------
  public Chain getChain() {
    return nextChain;
  }
}
```

The FileList class is quite similar and can be derived from the RestList class, to avoid replicating the *addChain* and *getChain* methods. The only differences are that it loads a list of the files in the current directory into the list when initialized, and looks for one of those files when it receives a request.

```
public class FileList extends RestList
{
   String files[];
   private Chain nextChain;
//---------------------------------------
   public FileList()
   {
   super();
   File dir = new File(System.getProperty("user.dir"));
  files = dir.list();
  for(int i = 0; i<files.length; i++)
  add(files[i]);
 }
//-------------------------------------
public void sendToChain(String mesg)
   {
   boolean found = false;
   int i = 0;
```

```
   while ((! found) && (i < files.length))    {
      XFile xfile = new XFile(files[i]);
      found = xfile.matchRoot(mesg);
      if (! found) i++;
   }
   if(found)       {
      setSelectedIndex(i);
      }
   else    {
      if(nextChain != null)
         nextChain.sendToChain(mesg);
   }
 }
```

The Xfile class we introduce above is a simple child of the File class that contains a *matchRoot* method to compare a string to the root name of a file.

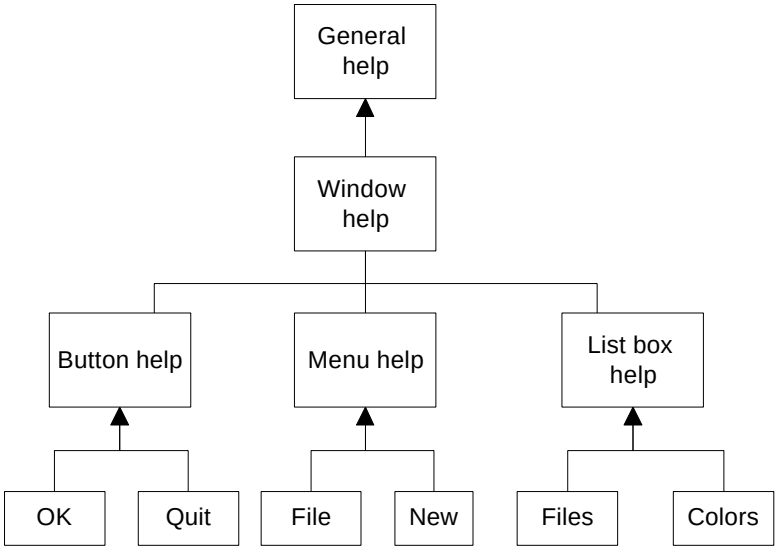
Finally, we link these classes together in the constructor to form the Chain:

```
//set up the chain of responsibility
   sender.addChain(imager);
   imager.addChain(colorImage);
   colorImage.addChain(fileList);
   fileList.addChain(restList);
```

This program is called *Chainer.java* on your CD-ROM.

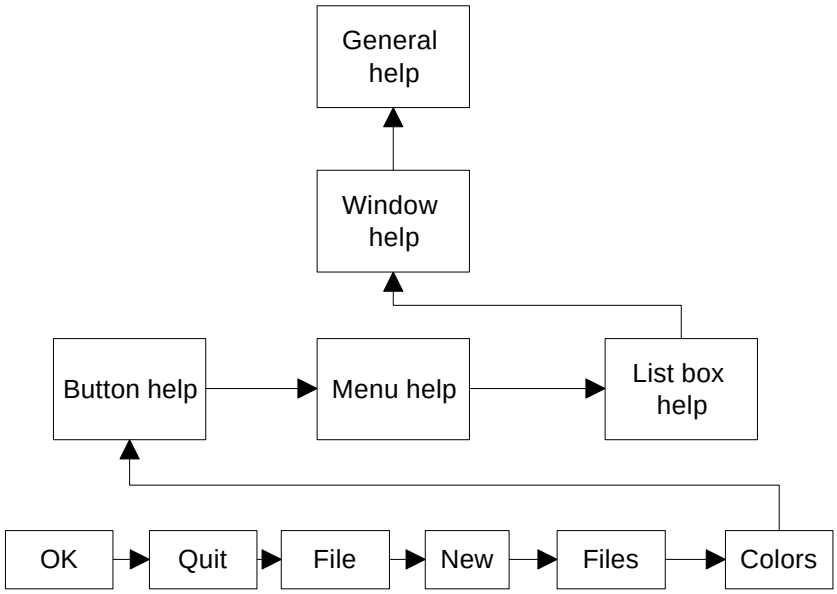
A Chain or a Tree?

Of course, a Chain of Responsibility does not have to be linear. The *Smalltalk Companion* suggests that it is more generally a tree structure with a number of specific entry points all pointing upward to the most general node.



However, this sort of structure seems to imply that each button, or is handler, knows where to enter the chain. This can complicate the design in some cases, and may preclude the need for the chain at all.

Another way of handling a tree-like structure is to have a single entry point that branches to the specific button, menu or other widget types, and then “un-branches” as above to more general help cases. There is little reason for that complexity -- you could align the classes into a single chain, starting at the bottom, and going left to right and up a row at a time until the entire system had been traversed, as shown below:



Kinds of Requests

The request or message passed along the Chain of Responsibility may well be a great deal more complicated than just the string that we conveniently used on this example. The information could include various data types or a complete object with a number of methods. Since various classes along the chain may use different properties of such a request object, you might end up designing an abstract Request type and any number of derived classes with additional methods.

Examples in Java

The most obvious example of the Chain of Responsibility is the class inheritance structure itself. If you call for a method to be executed in a deeply derived class, that method is passed up the inheritance chain until the first parent class containing that method is found. The fact that further parents contain other implementations of that method does not come into play.

Consequences of the Chain of Responsibility

1. The main purpose for this pattern, like a number of others, is to reduce coupling between objects. An object only needs to know how to forward the request to other objects.
2. This approach also gives you added flexibility in distributing responsibilities between objects. Any object can satisfy some or all of the requests, and you can change both the chain and the responsibilities at run time.
3. An advantage is that there may not be any object that can handle the request, however, the last object in the chain may simply discard any requests it can't handle.
4. Finally, since Java can not provide multiple inheritance, the basic Chain class needs to be an interface rather than an abstract class, so that the individual objects can inherit from another useful hierarchy, as we did here by deriving them all from JPanel. This disadvantage of this approach is that you often have to implement the linking, sending and forwarding code in each module separately.

THE COMMAND PATTERN

The Chain of Responsibility forwards requests along a chain of classes, but the Command pattern forwards a request only to a specific module. It encloses a request for a specific action inside an object and gives it a known public interface. It lets you give the client the ability to make requests without knowing anything about the actual action that will be performed, and allows you to change that action without affecting the client program in any way.

Motivation

When you build a Java user interface, you provide menu items, buttons, and checkboxes and so forth to allow the user to tell the program what to do. When a user selects one of these controls, the program receives an *ActionEvent*, which it must trap by subclassing, the *actionPerformed* event. Let's suppose we build a very simple program that allows you to select the menu items File | Open and File | Exit, and click on a button marked Red which turns the background of the window red. This program is shown below.

The program consists of the File Menu object with the mnuOpen and mnuExit MenuItems added to it. It also contains one button called btnRed. A click on any of these causes an *actionPerformed* event that we can trap with the following code:

```
public void actionPerformed(ActionEvent e)    {
      Object obj = e.getSource();
      if(obj == mnuOpen)
         fileOpen();             //open file
      if (obj == mnuExit)
         exitClicked();          //exit from program
      if (obj == btnRed)
```

```
      redClicked();           //turn red
   }
```

The three private methods this method calls are just

```
   private void exitClicked()    {
      System.exit(0);
   }
   //---------------------------------------
   private void fileOpen()     {
    FileDialog fDlg = new FileDialog(this, "Open a file",
                                FileDialog.LOAD);
      fDlg.show();
   }
   //---------------------------------------
   private void redClicked()     {
      p.setBackground(Color.red);
   }
```

Now, as long as there are only a few menu items and buttons, this approach works fine, but when you have dozens of menu items and several buttons, the *actionPerformed* code can get pretty unwieldy. In addition, this really seems a little inelegant, since we'd really hope that in an object-oriented language like Java, we could avoid a long series of if statements to identify the selected object. Instead, we'd like to find a way to have each object receive its commands directly.

The Command Pattern

One way to assure that every object receives its own commands directly is to use the Command object approach. A Command object always has an Execute() method that is called when an action occurs on that object. Most simply, a Command object implements at least the following interface:

```
public interface Command  {
   public void Execute();
}
```

The objective of using this interface is to reduce the actionPerformed method to:

```
public void actionPerformed(ActionEvent e)  {
      Command cmd = (Command)e.getSource();
      cmd.Execute();
   }
```

Then we can provide an Execute method for each object which carries out the desired action, thus keeping the knowledge of what to do inside the object where it belongs, instead of having another part of the program make these decisions.

One important purpose of the Command pattern is to keep the program and user interface objects completely separate from the actions that they initiate. In other words, these program objects should be completely separate from each other and should not have to know how other objects work. The user interface receives a command and tells a Command object to carry out whatever duties it has been instructed to do. The UI does not and should not need to know what tasks will be executed.

The Command object can also be used when you need to tell the program to execute the command when the resources are available rather than immediately. In such cases, you are *queuing* commands to be executed later. Finally, you can use Command objects to remember operations so that you can support Undo requests.

Building Command Objects

There are several ways to go about building Command objects for a program like this and each has some advantages. We'll start with the simplest one: deriving new classes from the MenuItem and Button classes and implementing the Command interface in each. Here are examples of extensions to the Button and Menu classes for our simple program:

```
class btnRedCommand extends Button
                    implements Command     {
      public btnRedCommand(String caption)       {
         super(caption);   //initialize the button
      }
      public void Execute()      {
         p.setBackground(Color.red);
      }
   }
   //----------------------------------------
class fileExitCommand extends MenuItem
                    implements Command    {
      public fileExitCommand(String caption)       {
         super(caption);         //initialize the Menu
      }
      public void Execute()       {
         System.exit(0);
      }
   }
```

This certainly lets us simplify the calls made in the actionPerformed method, but it does require that we create and instantiate a new class for each action we want to execute.

```
mnuOpen.addActionListener(new fileOpen());
```

```
mnuExit.addActionListener(new fileExit());
btnRed.addActionListener(new btnRed());
```

We can circumvent most of the problem of passing needed parameters to these classes by making them *inner classes*. This makes the Panel and Frame objects available directly.

However, interior classes are not such a good idea as commands proliferate, since any of them that access any other UI components have to remain inside the main class. This clutters up the code for this main class with a lot of confusing little inner classes.

Of course, if we are willing to pass the needed parameters to these classes, they can be independent. Here we pass in the Frame object and a Panel object:

```
mnuOpen = new fileOpenCommand("Open...", this);
mnuFile.add(mnuOpen);
mnuExit = new fileExitCommand("Exit");
mnuFile.add(mnuExit);
 p = new Panel();
add(p);
btnRed = new btnRedCommand("Red", p);
p.add(btnRed);
```

In this second case, our menu and button command classes can then be external to the main class, and even stored in separate files if we prefer.

The Command Pattern in Java

But there are still a couple of more ways to approach this. If you give every control its own actionListener class, you are in effect creating individual command objects for each of them. And, in fact, this is really what the designers of the Java 1.1 event model had in mind. We have become accustomed to using these multiple if test routines because they occur in most simple example texts (like mine) even if they are not the best way to catch these events.

To implement this approach, we create little classes each of which implements the ActionListener interface:

```
class btnRed implements ActionListener    {
   public void actionPerformed(ActionEvent e)       {
       p.setBackground(Color.red);
   }
```

```
   }
   //-------------------------------------
   class fileExit implements ActionListener    {
      public void actionPerformed(ActionEvent e) {
         System.exit(0);
      }
   }
}
```

and register them as listeners in the usual way.

```
      mnuOpen.addActionListener(new fileOpen());
      mnuExit.addActionListener(new fileExit());
      btnRed.addActionListener(new btnRed());
```

Here we have made these inner classes, but they also could be external with arguments passed in, as we did above.

Consequences of the Command Pattern

The main disadvantage of the Command pattern is a proliferation of little classes that either clutters up the main class if they are inner or clutters up the program namespace if they are outer classes.

Now even in the case where we put all of our *actionPerformed* events in a single basket, we usually call little private methods to carry out the actual function. It turns out that these private methods are just about as long as our little inner classes, so there is frequently little difference in complexity between inner and outer class approaches.

Anonymous Inner Classes

We can reduce the clutter of our name space by creating unnamed inner classes by declaring an instance of a class on the spot where we need it. For example, we could create our Red button and the class for manipulating the background all at once

```
btnRed.addActionListener(new ActionListener()              {
    public void actionPerformed(ActionEvent e) {
          p.setBackground(Color.red);
    }
 }  );
```

This is not very readable, however, and does not really improve the number of run-time classes since the compiler generates a class file even for these unnamed classes.

In fact, there is very little difference in the compiled code size among these various methods, as shown in Table 1, once you create classes in any form at all.

Table 1- Byte code size of Command class implementations

Program type	Byte code size
No command classes	**1719**
Named inner classes	**4450**
Unnamed inner classes	**3683**
External classes	**3838**

Providing Undo

Another of the main reasons for using Command design patterns is that they provide a convenient way to store and execute an Undo function. Each command object can remember what it just did and restore that state when requested to do so if the computational and memory requirements are not too overwhelming.

THE INTERPRETER PATTERN

Some programs benefit from having a language to describe operations they can perform. The Interpreter pattern generally describes defining a grammar for that language and using that grammar to interpret statements in that language.

Motivation

When a program presents a number of different, but somewhat similar cases it can deal with, it can be advantageous to use a simple language to describe these cases and then have the program interpret that language. Such cases can be as simple as the sort of Macro language recording facilities a number of office suite programs provide, or as complex as Visual Basic for Applications (VBA). VBA is not only included in Microsoft Office products, but can be embedded in any number of third party products quite simply.

One of the problems we must deal with is how to recognize when a language can be helpful. The Macro language recorder simply records menu and keystroke operations for later playback and just barely qualifies as a language; it may not actually have a written form or grammar. Languages such as VBA, on the other hand, are quite complex, but are far beyond the capabilities of the individual application developer. Further, embedding commercial languages such as VBA, Java or SmallTalk usually require substantial licensing fees, which make them less attractive to all but the largest developers.

Applicability

As the *SmallTalk Companion* notes, recognizing cases where an Interpreter can be helpful is much of the problem, and programmers without formal language/compiler training frequently overlook this approach. There are not large numbers of such cases, but there are two general places where languages are applicable:

1. **When the program must parse an algebraic string.** This case is fairly obvious. The program is asked to carry out its operations based on a computation where the user enters an equation of some sort. This frequently occurs in mathematical-graphics programs, where the program

renders a curve or surface based on any equation it can evaluate. Programs like *Mathematica* and graph drawing packages such as *Origin* work in this way.

2. **When the program must produce varying kinds of output.** This case is a little less obvious, but far more useful. Consider a program that can display columns of data in any order and sort them in various ways. These programs are frequently referred to as Report Generators, and while the underlying data may be stored in a relational database, the user interface to the report program is usually much simpler then the SQL language which the database uses. In fact, in some cases, the simple report language may be interpreted by the report program and translated into SQL.

Sample Code

Let's consider a simplified report generator that can operate on 5 columns of data in a table and return various reports on these data. Suppose we have the following sort of results from a swimming competition:

```
Amanda McCarthy             12  WCA         29.28
Jamie Falco                 12  HNHS        29.80
Meaghan O'Donnell           12  EDST        30.00
Greer Gibbs                 12  CDEV        30.04
Rhiannon Jeffrey            11  WYW         30.04
Sophie Connolly             12  WAC         30.05
Dana Helyer                 12  ARAC        30.18
```

where the 5 columns are *frname, lname, age, club* and *time.* If we consider the complete race results of 51 swimmers, we realize that it might be convenient to sort these results by club, by last name or by age. Since there are a number of useful reports we could produce from these data in which the order of the columns changes as well as the sorting, a language is one useful way to handle these reports.

We'll define a very simple non-recursive grammar of the sort

```
Print lname frname club time sortby club thenby time
```

For the purposes of this example, we define the 3 verbs shown above:

```
Print
Sortby
Thenby
```

and the 5 column names we listed earlier:

```
Frname
Lname
Age
Club
Time
```

For convenience, we'll assume that the language is case insensitive. We'll also note that the simple grammar of this language is punctuation free, and amounts in brief to

Print var[var] [sortby var [thenby var]]

Finally, there is only one main verb and while each statement is a declaration, there is no assignment statement or computational ability in this grammar.

Interpreting the Language

Interpreting the language takes place in three steps

1. Parsing the language symbols into tokens.
2. Reducing the tokens into actions.
3. Executing the actions.

We parse the language into tokens by simply scanning each statement with a StringTokenizer and then substituting a number for each word. Usually parsers push each parsed token onto a *stack* -- we will use that technique here. We implement the Stack class using a Vector, where we have *push*, *pop*, *top* and *nextTop* methods to examine and manipulate the stack contents.

After parsing, our stack could look like this:

Type	Token
Var	Time
Verb	Thenby
Var	Club
Verb	Sortby
Var	Time
Var	Club

<-top of stack

Var	Frname
verb	Lname

However, we quickly realize that the "verb" *thenby* has no real meaning other than clarification, and it is more likely that we'd parse the tokens and skip the *thenby* word altogether. Our initial stack then, looks like this

```
Time
Club
Sortby
Time
Club
Frname
Lname
Print
```

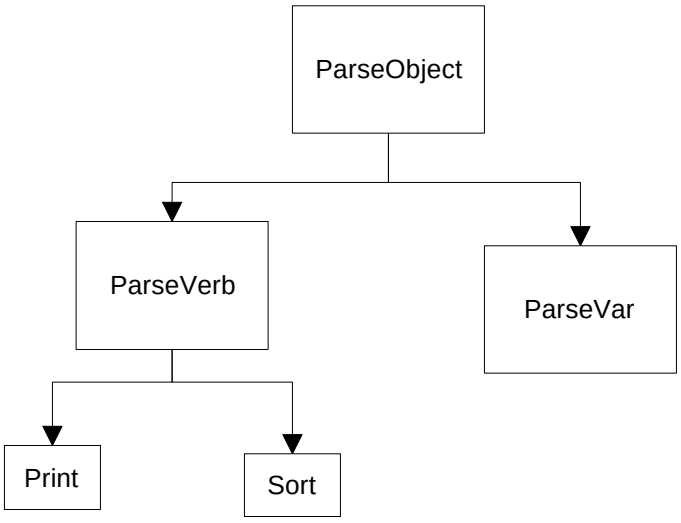
Objects Used in Parsing

Actually, we do not push just a numeric token onto the stack, but a *ParseObject* which has the both a type and a value property:

```
public class ParseObject
{
 public static final int VERB=1000, VAR = 1010,
                          MULTVAR = 1020;
   protected int value;
   protected int type;

   public int getValue() {return value;}
   public int getType() {return type;}
}
```

These objects can take on the type VERB or VAR. Then we extend this object into ParseVerb and ParseVar objects, whose value fields can take on PRINT or SORT for ParseVerb and FRNAME, LNAME, etc. for ParseVar. For later use in reducing the parse list, we then derive *Print* and *Sort* objects from ParseVerb.

This gives us a simple hierachy:



The parsing process is just the following simple code, using the StringTokenizer and the parse objects:

```
public Parser(String line)    {
      stk = new Stack();
      actionList = new Vector();

      StringTokenizer tok = new StringTokenizer(line);
      while(tok.hasMoreElements())       {
         ParseObject token = tokenize(tok.nextToken());
         if(token != null)
            stk.push(token);
      }
   }
//-----------------------------------
private ParseObject tokenize(String s)       {
         ParseObject obj = getVerb(s);
         if (obj == null)
            obj = getVar(s);
         return obj;
}
//----------------------------------------
 private ParseVerb getVerb(String s)  {
         ParseVerb v;
  v = new ParseVerb(s);
  if (v.isLegal())
      return v.getVerb(s);
 else
     return null;
}
```

```
//---------------------------------------
private ParseVar getVar(String s)       {
         ParseVar v;
 v = new ParseVar(s);
 if (v.isLegal())
     return v;
 else
     return null;
}
```

The ParseVerb and ParseVar classes return objects with *isLegal* set to true if they recognize the word.

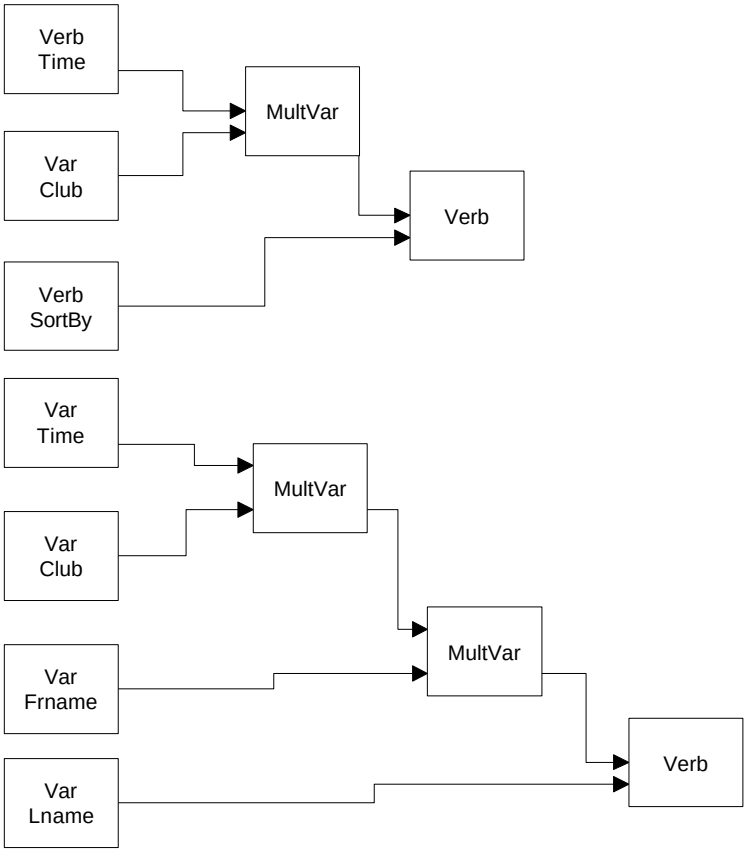
```
public class ParseVerb extends ParseObject
{
   static public final int PRINT=100,
              SORTBY=110, THENBY=120;
   protected Vector args;

public ParseVerb(String s) {
   args = new Vector();
   s = s.toLowerCase();
   value = -1;
   type = VERB;
   if (s.equals("print")) value = PRINT;
   if (s.equals("sortby")) value = SORTBY;
}
```

Reducing the Parsed Stack

The tokens on the stack have the form

```
Var
Var
Verb
Var
Var
Var
Var
Verb
```

We reduce the stack a token at a time, folding successive Vars into a MultVar class until the arguments are folded into the verb objects.



When the stack reduces to a verb, this verb and its arguments are placed in an action list; when the stack is empty the actions are executed.

This entire process is carried out by creating a Parser class that is a Command object, and executing it when the Go button is pressed on the user interface:

```
public void actionPerformed(ActionEvent e)
  {
     Parser p =  new Parser(tx.getText());
     p.setData(kdata, ptable);
     p.Execute();
  }
```

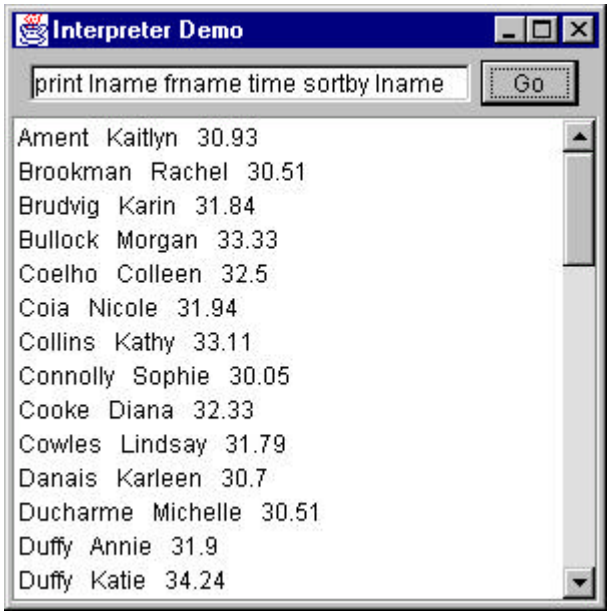
The parser itself just reduces the tokens as we show above. It checks for various pairs of tokens on the stack and reduces each pair to a single one for each of five different cases.

```
//executes parse of command line
   public void Execute()    {
    while(stk.hasMoreElements())   {
     if(topStack(ParseObject.VAR, ParseObject.VAR))
      {
       //reduce (Var Var) to Multvar
       ParseVar v = (ParseVar)stk.pop();
       ParseVar v1 = (ParseVar)stk.pop();
       MultVar mv = new MultVar(v1, v);
       stk.push(mv);
       }
     //reduce MULTVAR VAR to MULTVAR
    if(topStack(ParseObject.MULTVAR, ParseObject.VAR))
      {
       MultVar mv =  new MultVar();
       MultVar mvo = (MultVar)stk.pop();
       ParseVar v = (ParseVar)stk.pop();
       mv.add(v);
       Vector mvec = mvo.getVector();
       for (int i = 0; i< mvec.size(); i++)
          mv.add((ParseVar)mvec.elementAt(i));
       stk.push(mv);
      }
    if(topStack(ParseObject.VAR, ParseObject.MULTVAR))
      {
     //reduce (Multvar Var) to Multvar
       ParseVar v = (ParseVar)stk.pop();
       MultVar mv = (MultVar)stk.pop();
       mv.add(v);
       stk.push(mv);
       }
     //reduce Verb Var to Verb containing vars
     if (topStack(ParseObject.VAR, ParseObject.VERB))
        {
        addArgsToVerb();
        }
     //reduce Verb MultVar to Verb containing vars
     if (topStack(ParseObject.MULTVAR, ParseObject.VERB))
        {
        addArgsToVerb();
        }
     //move top verb to action list
     if(stk.top().getType() == ParseObject.VERB)
     {
        actionList.addElement(stk.pop());
     }

    }//while
    //now execute the verbs
    for (int i =0; i< actionList.size() ; i++)   {
      Verb v = (Verb)actionList.elementAt(i);
```

```
            v.Execute();
        }
    }
```

We also make the Print and Sort verb classes Command objects and Execute them one by one as the action list is enumerated.

The final visual program is shown below:



Consequences of the Interpreter Pattern

Whenever you introduce an interpreter into a program, you need to provide a simple way for the program user to enter commands in that language. It can be as simple as the Macro record button we noted earlier, or it can be an editable text field like the one in the program above.

However, introducing a language and its accompanying grammar also requires fairly extensive error checking for misspelled terms or misplaced grammatical elements. This can easily consume a great deal of programming effort unless some template code is available for implementing this checking. Further, effective methods for notifying the users of these errors are not easy to design and implement.

In the Interpreter example above, the only error handling is that keywords that are not recognized are not converted to ParseObjects and pushed onto the stack. Thus, nothing will happen, because the resulting stack

sequence probably cannot be parsed successfully, or if it can, the item represented by the misspelled keyword will not be included.

You can also consider generating a language automatically from a user interface of radio and command buttons and list boxes. While it may seem that having such an interface obviates the necessity for a language at all, the same requirements of sequence and computation still apply. When you have to have a way to specify the order of sequential operations, a language is a good way to do so, even if the language is generated from the user interface.

The Interpreter pattern has the advantage that you can extend or revise the grammar fairly easily one you have built the general parsing and reduction tools. You can also add new verbs or variables quite easily once the foundation is constructed.

In the simple parsing scheme we show in the Parser class above, there are only 6 cases to consider, and they are shown as a series of simple *if* statements. If you have many more than that, *Design Patterns* suggests that you create a class for each one of them. This again makes language extension easier, but has the disadvantage of proliferating lots of similar little classes.

Finally, as the syntax of the grammar becomes more complex, you run the risk of creating a hard to maintain program.

While interpreters are not all that common in solving general programming problems, the Iterator pattern we take up next is one of the most common ones you'll be using.

THE ITERATOR PATTERN

The Iterator is one of the simplest and most frequently used of the design patterns. The Iterator pattern allows you to move through a list or collection of data using a standard interface without having to know the details of the internal representations of that data. In addition you can also define special iterators that perform some special processing and return only specified elements of the data collection.

Motivation

The Iterator is useful because it provides a defined way to move through a set of data elements without exposing how it does it. Since the Iterator is an *interface*, you can implement it in any way that is convenient for the data you are returning. *Design Patterns* suggests that a suitable interface for an Iterator might be

```
public interface Iterator
{
public Object First();
public Object Next();
public boolean isDone();
public Object CurrentItem();
}
```

where you can move to the top of the list, move through the list, find out if there are more elements and find the current list item. This interface is easy to implement and it has certain advantages, but the Iterator of choice in Java is Java's built-in Enumeration type.

```
public interface Enumeration
{
public boolean hasMoreElements();
public Object nextElement();
}
```

While not having a method to move to the top of a list may seem restrictive a first, it is not a serious problem in Java, because it is customary to obtain a new instance of the Enumeration each time you want to move through a list. One disadvantage of the Java Enumeration over similar constructs in C++ and Smalltalk is the strong typing of the Java language. This prevents the *hasMoreElements()* method from returning an object of the actual type of the data in the collection without an annoying requirement to cast the returned Object type to the actual type. Thus, while the Iterator or

Enumeration interface is that is intended to be polymorphic, this is not directly possible in Java.

Enumerations in Java

The Enumeration type is built into the Vector and Hashtable classes. Rather than the Vector and Hashtable implementing the two methods of the Enumeration directly, both classes contain an *elements* method that returns an Enumeration of that class's data:

```
public Enumeration elements();
```

This *elements()* method is really a kind Factory method that produces instances of an Enumeration class.

Then, you move through the list with the following simple code:

```
Enumeration e = vector.elements();
while (e.hasMoreElements())
       {
       String name = (String)e.nextElement();
       System.out.println(name);
       }
```

In addition, the Hashtable also has the *keys* method, which returns an enumeration of the keys to each element in the table:

```
public Enumeration keys();
```

This is the preferred style for implementing Enumerations in Java and has the advantage that you can have any number of simultaneous active enumerations of the same data.

Filtered Iterators

While having a clearly defined method of moving through a collection is helpful, you can also define filtered Enumerations that perform some computation on the data before returning it. For example, you could return the data ordered in some particular way, or only those objects that match a particular criterion. Then, rather than have a lot of very similar interfaces for these filtered enumerations, you simply provide a method which returns each type of enumeration, with each one of these enumerations having the same methods.

Sample Code

Let's reuse the list of swimmers, clubs and times we described in the Interpreter chapter, and add some enumeration capabilities to the KidData class. This class is essentially a collection of Kids, each with a name, club and time, and these Kid objects are stored in a Vector.

```
public class KidData
{
   Vector kids;
//----------------------------------------
   public KidData(String filename)    {
     //read in the kids from the text file
      kids = new Vector();
      InputFile f = new InputFile(filename);
      String s = f.readLine();
      while(s != null)   {
         if(s.trim().length() > 0)  {
         Kid k = new Kid(s);
         kids.addElement(k);
         }
         s = f.readLine();
      }
   }
   //--------------------------------
   public Enumeration elements()    {
       //return an enumeration of the kids
      return kids.elements();
   }
```

To obtain an enumeration of all the Kids in the collection, we simply return the enumeration of the Vector itself.

The Filtered Enumeration

Suppose, however, that we wanted to enumerate only those kids who belonged to a certain club. This necessitates a special Enumeration class that has access to the data in the KidData class. This is very simple, because the *elements()* method we just defined gives us that access. Then we only need to write an Enumeration that only returns kids belonging to a specified club:

```
public class kidClub
   implements Enumeration
{
   String clubMask;     //name of club
   Kid kid;             //next kid to return
   Enumeration ke;      //gets all kids
   KidData kdata;       //class containing kids
//----------------------------------------
   public kidClub(KidData kd, String club)    {
```

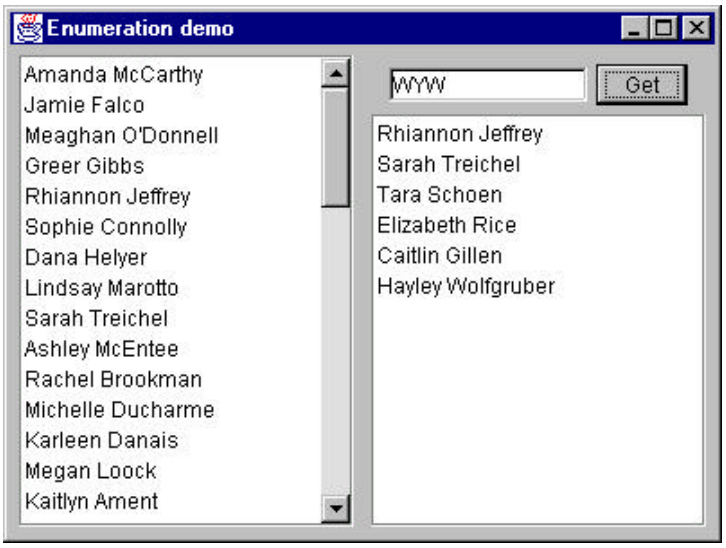
```
      clubMask = club;     //save the club
      kdata = kd;          //copy the class
      kid = null;          //default
      ke = kdata.elements();  //get Enumerator
   }
//---------------------------------------
   public boolean hasMoreElements()    {
   //return true if there are any more kids
   //belonging to the specified club
   boolean found = false;
   while(ke.hasMoreElements() && ! found)       {
      kid = (Kid)ke.nextElement();
      found = kid.getClub().equals(clubMask);
      }
   if(! found)
      kid = null;    //set to null if none left
   return found;
   }
//---------------------------------------
   public Object nextElement()    {
      if(kid != null)
         return kid;
      else
         //throw exception if access past end
         throw new NoSuchElementException();
   }
}
```

All of the work is done in the *hasMoreElements()* method, which scans through the collection for another kid belonging to the club specified in the constructor, and saves that kid in the *kid* variable, or sets it to *null*. Then, it returns either true or false. The *nextElement()* method either returns that next kid variable or throws an exception if there are no more kids. Note that under normal circumstances, this exception is never thrown, since the *hasMoreElements* boolean should have already told you not to ask for another element.

Finally, we need to add a method to KidData to return this new filtered Enumeration:

```
public Enumeration kidsInClub(String club)    {
      return new kidClub(this, club);
   }
```

This simple method passes the instance of KidClub to the Enumeration class kidClub along with the club initials. A simple program is shown below, that displays all of the kids on the left side and those belonging to a single club on the right.



Consequence of the Iterator Pattern

1. *Data modification.* The most significant question iterators may raise is the question of iterating through data while it is being changed. If your code is wide ranging and only occasionally moves to the next element, it is possible that an element might be added or deleted from the underlying collection while you are moving through it. It is also possible that another thread could change the collection. There are no simple answers to this problem. You can make an enumeration thread-safe by declaring the loop to be *synchronized*, but if you want to move through a loop using an Enumeration, and delete certain items, you must be careful of the consequences. Deleting or adding an element might mean that a particular element is skipped or accessed twice, depending on the storage mechanism you are using.
2. *Privileged access.* Enumeration classes may need to have some sort of privileged access to the underlying data structures of the original container class, so they can move through the data. If the data is stored in a Vector or Hashtable, this is pretty easy to accomplish, but if it is in some other collection structure contained in a class, you probably have to make that structure available through a *get* operation. Alternatively, you could make the Iterator a derived class of the containment class and access the data directly. The *friend* class solution available in C++ does

not apply in Java. However, classes defined in the same module as the containing class do have access to the containing classes variables.

3. *External versus Internal Iterators.* The *Design Patterns* text describes two types of iterators: external and internal. Thus far, we have only described external iterators. Internal iterators are methods that move through the entire collection, performing some operation on each element directly, without any specific requests from the user. These are less common in Java, but you could imagine methods that normalized a collection of data values to lie between 0 and 1 or converted all of the strings to a particular case. In general, external iterators give you more control, because the calling program accesses each element directly and can decide whether to perform an operation on.

Composites and Iterators

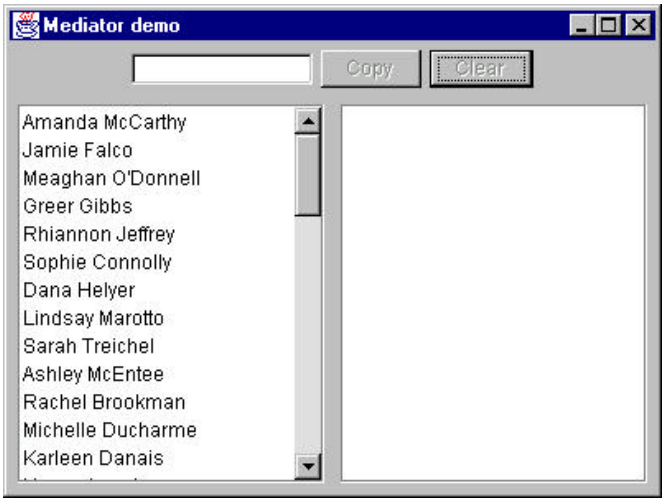
Iterators, or in our case Enumerations, are also an excellent way to move through Composite structures. In the Composite of an employee hierarchy we developed in the previous chapter, each Employee contains a Vector whose *elements()* method allows you to continue to enumerate down that chain. If that Employee has no subordinates, the *hasMoreElements()* method correctly returns false.

THE MEDIATOR PATTERN

When a program is made up of a number of classes, the logic and computation is divided logically among these classes. However, as more of these isolated classes are developed in a program, the problem of communication between these classes become more complex. The more each class needs to know about the methods of another class, the more tangled the class structure can become. This makes the program harder to read and harder to maintain. Further, it can become difficult to change the program, since any change may affect code in several other classes. The Mediator pattern addresses this problem by promoting looser coupling between these classes. Mediators accomplish this by being the only class that has detailed knowledge of the methods of other classes. Classes send inform the mediator when changes occur and the Mediator passes them on to any other classes that need to be informed.

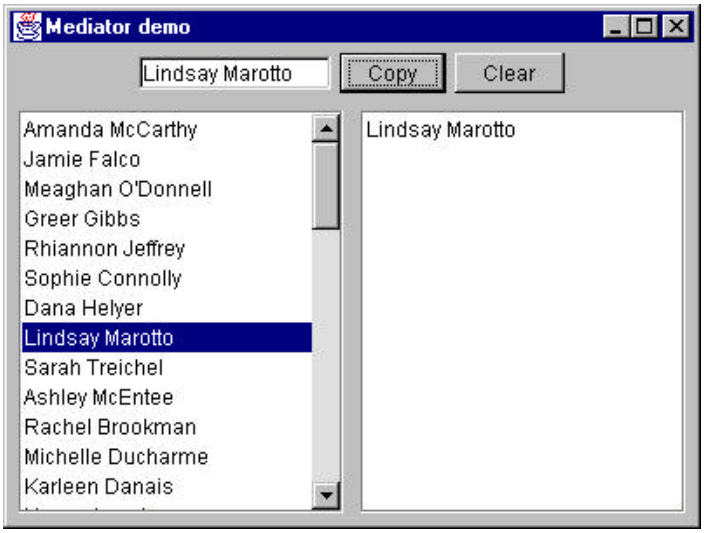
An Example System

Let's consider a program which has several buttons, two list boxes and a text entry field:



When the program starts, the Copy and Clear buttons are disabled.

1. When you select one of the names in the left-hand list box, it is copied into the text field for editing, and the *Copy* button is enabled.
2. When you click on *Copy*, that text is added to the right hand list box, and the *Clear* button is enabled.

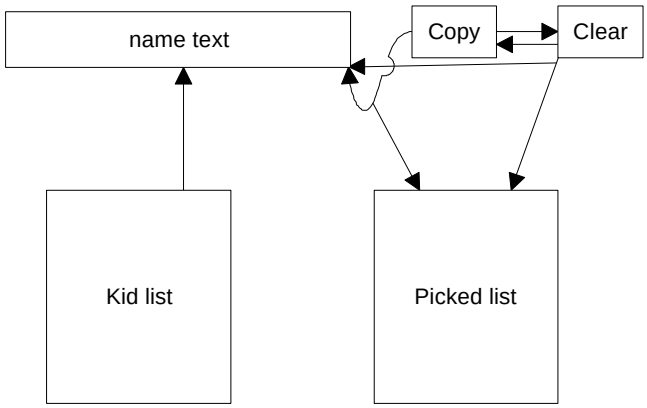


3. If you click on the *Clear* button, the right hand list box and the text field are cleared, the list box is deselected and the two buttons are again disabled.

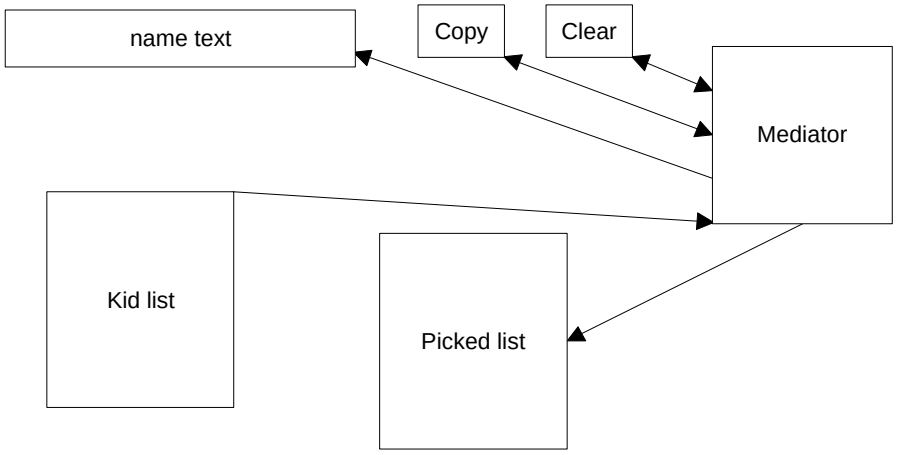
User interfaces such as this one are commonly used to select lists of people or products from longer lists. Further, they are usually even more complicated than this one, involving insert, delete and undo operations as well.

Interactions between Controls

The interactions between the visual controls are pretty complex, even in this simple example. Each visual object needs to know about two or more others, leading to quite a tangled relationship diagram as shown below.



The Mediator pattern simplifies this system by being the only class that is aware of the other classes in the system. Each of the controls that the Mediator communicates with is called a Colleague. Each Colleague informs the Mediator when it has received a user event, and the Mediator decides which other classes should be informed of this event. This simpler interaction scheme is illustrated below:



The advantage of the Mediator is clear-- it is the only class that knows of the other classes, and thus the only one that would need to be changed if one of the other classes changes or if other interface control classes are added.

Sample Code

Let's consider this program in detail and decide how each control is constructed. The main difference in writing a program using a Mediator class is that each class needs to be aware of the existence of the Mediator. You start by creating an instance of the Mediator and then pass the instance of the Mediator to each class in its constructor.

```
Mediator med = new Mediator();
kidList = new KidList( med);
tx = new KTextField(med);
 Move = new MoveButton(this, med);
Clear = new ClearButton(this, med);
med.init();
```

Since, we have created new classes for each control, each derived from base classes, we can handle the mediator operations within each class.

Our two buttons use the Command pattern and register themselves with the Mediator during their initialization. Here is the Copy button:

```
public class CopyButton extends JButton
      implements Command
{
  Mediator med;            //copy of the Mediator
public CopyButton(ActionListener fr, Mediator md)
 {
    super("Copy");         //create the button
    addActionListener(fr); //add its listener
    med = md;              //copy in Mediator instance
    med.registerMove(this); //register with the Mediator
 }
 public void Execute()
 {                         //execute the copy
    med.Copy();
 }
}
```

The Clear button is exactly analogous.

The Kid name list is based on the one we used in the last two examples, but expanded so that the data loading of the list and registering the list with the Mediator both take place in the constructor. In addition, we make the enclosing class the ListSelectionListener and pass the click on any list item on to the Mediator directly from this class.

```
public class KidList extends JawtList
   implements ListSelectionListener
```

```
{
   KidData kdata;          //reads the data from the file
   Mediator med;           //copy of the mediator

  public KidList(Mediator md)
  {
   super(20);       //create the JList
   kdata = new KidData ("50free.txt");
   fillKidList();          //fill the list with names
   med = md;        //save the mediator
   med.registerKidList(this);
   addListSelectionListener(this);
  }
 //---------------------------------
  public void valueChanged(ListSelectionEvent ls)
  {
      //if an item was selected pass on to mediator
     JList obj = (JList)ls.getSource();
     if (obj.getSelectedIndex() >= 0)
         med.select();
  }
  //---------------------------------
  private void fillKidList()
   {
      Enumeration ekid = kdata.elements();
      while (ekid.hasMoreElements())  {
         Kid k =(Kid)ekid.nextElement();
         add(k.getFrname()+" "+k.getLname());
         }
   }
}
```

The text field is even simpler, since all it does is register itself with the mediator.

```
public class KTextField extends JTextField
{
 Mediator med;
 public KTextField(Mediator md)  {
    super(10);
    med = md;
    med.registerText(this);
 }
}
```

The general point of all these classes is that each knows about the Mediator and tells the Mediator of its existence so the Mediator can send commands to it when appropriate.

The Mediator itself is very simple. It supports the Copy, Clear and Select methods, and has register methods for each of the controls:

```
public class Mediator
{
 private ClearButton clearButton;
 private CopyButton  copyButton;
 private KTextField  ktext;
 private KidList     klist;
 private PickedKidsList picked;

public void Copy()  {
    picked.add(ktext.getText()); //copy text
    clearButton.setEnabled(true);//enable Clear
 }
//-------------------------------------
 public void Clear()  {
    ktext.setText("");                 //clear text
    picked.clear();              //and list
//disable buttons
    copyButton.setEnabled(false);
    clearButton.setEnabled(false);
    klist.clearSelection();             //deselect list
}
 //-------------------------------------
 public void Select()  {
    String s = (String)klist.getSelectedValue();
    ktext.setText(s);                  //copy text
    copyButton.setEnabled(true); //enable Copy
}
 //-----------copy in controls-------------------------
 public void registerClear(ClearButton cb)  {
    clearButton = cb;  }
 public void registerCopy(CopyButton mv)  {
    copyButton = mv;  }
 public void registerText(KTextField tx)  {
    ktext = tx;  }
 public void registerPicked(PickedKidsList pl)  {
    picked = pl;  }
 public void registerKidList(KidList kl)  {
    klist = kl;  }
}
```

Initialization of the System

One further operation that is best delegated to the Mediator is the initialization of all the controls to the desired state. When we launch the program, each control must be in a known, default state, and since these states may change as the program evolves, we simply create an *init* method in the Mediator, which sets them all to the desired state. In this case, that state is the same as is achieved by the Clear button and we simply call that method:

```
public void init()  {
```

```
    Clear();
}
```

Mediators and Command Objects

The two buttons in this program are command objects, and we register the main user interface frame as the *ActionListener* when we initialize these buttons. Just as we noted earlier, this makes processing of the button click events quite simple:

```
public void actionPerformed(ActionEvent e)    {
      Command comd = (Command)e.getSource();
      comd.Execute();
   }
```

Alternatively, we could register each derived class as its own listener and pass the result directly to the Mediator.

In either case, however, this represents the solution to one of the problems we noted in the Command pattern chapter; each button needed knowledge of many of the other user interface classes in order to execute its command. Here, we delegate that knowledge to the Mediator, so that the Command buttons do not need any knowledge of the methods of the other visual objects.

Consequences of the Mediator Pattern

1. The Mediator makes loose coupling possible between objects in a program. It also localizes the behavior that otherwise would be distributed among several objects.
2. You can change the behavior of the program by simply changing or subclassing the Mediator.
3. The Mediator approach makes it possible to add new Colleagues to a system without having to change any other part of the program.
4. The Mediator solves the problem of each Command object needing to know too much about the objects and methods in the rest of a user interface.
5. The Mediator can become monolithic in complexity, making it hard to change and maintain. Sometimes you can improve this situation by revising the responsibilities you have given the Mediator. Each object should carry out it’s own tasks and the Mediator should only manage the interaction between objects.

6. Each Mediator is a custom-written class that has methods for each Colleague to call and knows what methods each Colleague has available. This makes it difficult to reuse Mediator code in different projects. On the other hand, most Mediators are quite simple and writing this code is far easier than managing the complex object interactions any other way.

Implementation Issues

The Mediator pattern we have described above acts as a kind of Observer pattern, observing changes in the Colleague elements. Another approach is to have a single interface to your Mediator, and pass that method various constants or objects which tell the Mediator which operations to perform. In the same fashion, you could have a single Colleague interface that each Colleague would implement, and each Colleague would then decide what operation it was to carry out.

Mediators are not limited to use in visual interface programs, however, it is their most common application. You can use them whenever you are faced with the problem of complex intercommunication between a number of objects.

THE MEMENTO PATTERN

Suppose you would like to save the internal state of an object so you can restore it later. Ideally, it should be possible to save and restore this state without making the object itself take care of this task, and without violating encapsulation. This is the purpose of the Memento pattern.

Motivation

Objects frequently expose only some of their internal state using public methods, but you would still like to be able to save the entire state of an object because you might need to restore it later. In some cases, you could obtain enough information from the public interfaces (such as the drawing position of graphical objects) to save and restore that data. In other cases, the color, shading, angle and connection relationship to other graphical objects need to be saved and this information is not readily available. This sort of information saving and restoration is common in systems that need to support Undo commands.

If all of the information describing an object is available in public variables, it is not that difficult to save them in some external store. However, making these data public makes the entire system vulnerable to change by external program code, when we usually expect data inside an object to be private and encapsulated from the outside world.

The Memento pattern attempts to solve this problem by having privileged access to the state of the object you want to save. Other objects have only a more restricted access to the object, thus preserving their encapsulation. This pattern defines three roles for objects:

1. The **Originator** is the object whose state we want to save.
2. The **Memento** is another object that saves the state of the Originator.
3. The **Caretaker** manages the timing of the saving of the state, saves the Memento and, if needed, uses the Memento to restore the state of the Originator.

Implementation

Saving the state of an object without making all of its variables publicly available is tricky and can be done with varying degrees of success

in various languages. *Design Patterns* suggests using the C++ *friend* construction to achieve this access, and the *Smalltalk Companion* notes that it is not directly possible in Smalltalk. In Java, this privileged access is possible using a little known and infrequently used protection mode. Variables within a Java class can be declared as

1. Private
2. Protected
3. Public, or
4. (private protected)

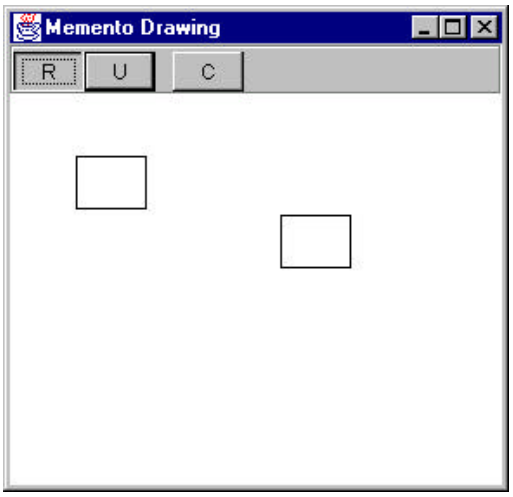
Variables with no declaration are treated as private protected. Other classes can access public variables, and derived classes can access protected variables. However, another class in the same module can access protected or private-protected variables. It is this last feature of Java that we can use to build Memento objects. For example, suppose you have classes A and B declared in the same module:

```
public class A {
int x, y:
public Square() {}
x = 5;                          //initialize x
}
//------------------------
class B {
public B() {
  A a = new A();                //create instance of A
  System.out.println (a.x);     //has access to variables in A
 }
}
```

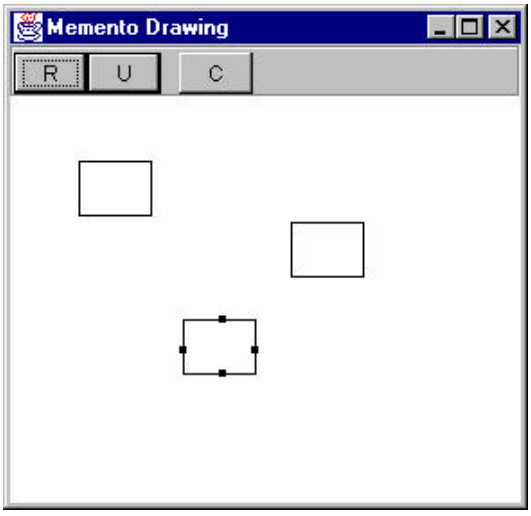
Class A contains a private-protected variable *x*. In class B in the same module, we create an instance of A, which automatically initializes *x* to 5. Class B has direct access to the variable *x* in class A and can print it out without compilation or execution error. It is exactly this feature that we will use to create a Memento.

Sample Code

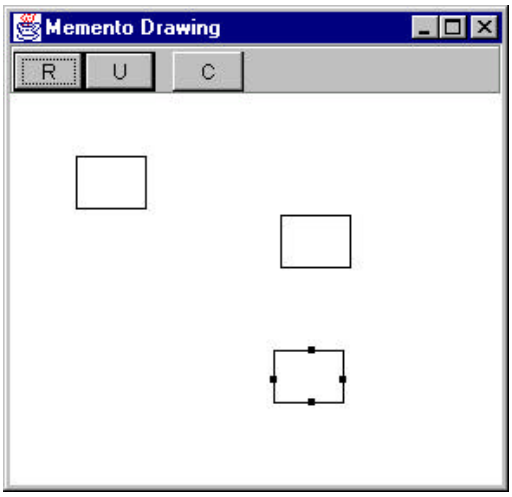
Let's consider a simple prototype of a graphics drawing program that creates rectangles, and allows you to select them and move them around by dragging them with the mouse. This program has a toolbar containing three buttons: Rectangle, Undo and Clear:



The Rectangle button is a JToggleButton which stays selected until you click the mouse to draw a new rectangle. Once you have drawn the rectangle, you can click in any rectangle to select it;



and once it is selected, you can drag that rectangle to a new position using the mouse:



The Undo button can undo a succession of operations. Specifically, it can undo moving a rectangle and it can undo the creation of each rectangle.

There are 5 actions we need to respond to in this program:

1. Rectangle button click
2. Undo button click
3. Clear button click
4. Mouse click
5. Mouse drag.

The three buttons can be constructed as Command objects and the mouse click and drag can be treated as commands as well. This suggests an opportunity to use the Mediator pattern, and that is, in fact, the way this program is constructed.

Moreover, our Mediator is an ideal place to manage the Undo action list; it can keep a list of the last *n* operations so that they can be undone. Thus, the Mediator also functions as the Caretaker object we described above. In fact, since there could be any number of actions to save and undo in such a program, a Mediator is virtually required so that there is a single place where these commands can be stored for undoing later.

In this program we save and undo only two actions: creating new rectangles and changing the position of rectangles. Let's start with our visRectangle class which actually draws each instance of the rectangles:

```
public class visRectangle
{
 int x, y, w, h;
 Rectangle rect;
 boolean selected;

  public visRectangle(int xpt, int ypt)   {
     x = xpt;   y = ypt;   //save location
     w = 40;    h = 30;    //use default size
     saveAsRect();
  }
  //-----------------------------------------
  public void setSelected(boolean b)   {
     selected = b;
  }
  //-----------------------------------------
  private void saveAsRect()   {
  //convert to rectangle so we can use the contains method
     rect = new Rectangle(x-w/2, y-h/2, w, h);
  }
 //-----------------------------------------
  public void draw(Graphics g)   {
     g.drawRect(x, y, w, h);
     if (selected)    {  //draw "handles"
        g.fillRect(x+w/2, y-2, 4, 4);
        g.fillRect(x-2, y+h/2, 4, 4);
        g.fillRect(x+w/2, y+h-2, 4, 4);
        g.fillRect(x+w-2, y+h/2, 4, 4);
     }
  }
  //-----------------------------------------
  public boolean contains(int x, int y)   {
     return rect.contains(x, y);
  }
  //-----------------------------------------
  public void move(int xpt, int ypt)   {
     x = xpt; y = ypt;
     saveAsRect();
  }
}
```

Drawing the rectangle is pretty straightforward. Now, let's look at our simple Memento class, which is contained in the same file, visRectangle.java, and thus has access to the position and size variables:

```
class Memento
{
   visRectangle rect;
   //saved fields- remember internal fields
   //of the specified visual rectangle
```

```
   int x, y, w, h;
   public Memento(visRectangle r)    {
      rect = r;     //Save copy of instance
      x = rect.x;  y = rect.y;   //save position
      w = rect.w;  h = rect.h;   //and size
   }
   //-----------------------------------------
   public void restore()    {
      //restore the internal state of
      //the specified rectangle
      rect.x = x;  rect.y = y;   //restore position
      rect.h = h;  rect.w = w;   //restore size
   }
}
```

When we create an instance of the Memento class, we pass it the visRectangle instance we want to save. It copies the size and position parameters and saves a copy of the instance of the visRectangle itself. Later, when we want to restore these parameters, the Memento knows which instance it has to restore them to and can do it directly, as we see in the *restore()* method.

The rest of the activity takes place in the Mediator class, where we save the previous state of the list of drawings as an Integer on the undo list:

```
public void createRect(int x, int y)
{
   unpick();   //make sure no rectangle is selected
   if(startRect)  //if rect button is depressed
   {
   Integer count = new Integer(drawings.size());
   undoList.addElement(count);   //Save previous list size
   visRectangle v = new visRectangle(x, y);
   drawings.addElement(v);       //add new element to list
   startRect = false;            //done with this rectangle
   rect.setSelected(false);      //unclick button
   canvas.repaint();
   }
   else
      pickRect(x, y); //if not pressed look for rect to select
   }
```

and save the previous position of a rectangle before moving it in a Memento:

```
public void rememberPosition()
{
   if(rectSelected){
   Memento m = new Memento(selectedRectangle);
   undoList.addElement(m);
   }
```

```
}
```

Our undo method simply decides whether to reduce the drawing list by one or to invoke the *restore* method of a Memento:

```
public void undo()
{
   if(undoList.size()>0)
   {
      //get last element in undo list
      Object obj = undoList.lastElement();
      undoList.removeElement(obj);   //and remove it
      //if this is an Integer,
      //the last action was a new rectangle
      if (obj instanceof Integer)
      {
         //remove last created rectangle
         Object drawObj = drawings.lastElement();
         drawings.removeElement(drawObj);
      }
      //if this is a Memento, the last action was a move
      if(obj instanceof Memento)
      {
         //get the Memento
         Memento m = (Memento)obj;
         m.restore();     //and restore the old position
      }
     repaint();
   }
 }
```

Consequences of the Memento

The Memento provides a way to preserve the state of an object while preserving encapsulation, in languages where this is possible. Thus, data that only the Originator class should have access to effectively remains private. It also preserves the simplicity of the Originator class by delegating the saving and restoring of information to the Memento class.

On the other hand, the amount of information that a Memento has to save might be quite large, thus taking up fair amounts of storage. This further has an effect on the Caretaker class (here the Mediator) which may have to design strategies to limit the number of objects for which it saves state. In our simple example, we impose no such limits. In cases where objects change in a predictable manner, each Memento may be able to get by with saving only incremental changes of an object’s state.

Other Kinds of Mementos

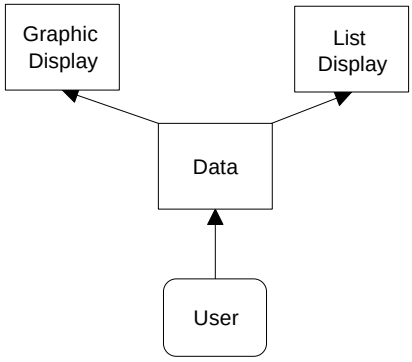
While supporting undo/redo operations in graphical interfaces is one significant use of the Memento pattern, you will also see Mementos used in database transactions. Here they save the state of data in a transaction where it is necessary to restore the data if the transaction fails or is incomplete.

THE OBSERVER PATTERN

In our new, more sophisticated windowing world, we often would like to display data in more than one form at the same time and have all of the displays reflect any changes in that data. For example, you might represent stock price changes both as a graph and as a table or list box. Each time the price changes, we'd expect both representations to change at once without any action on our part.

We expect this sort of behavior because there are any number of Windows applications, like Excel, where we see that behavior. Now there is nothing inherent in Windows to allow this activity and, as you may know, programming directly in Windows in C or C++ is pretty complicated. In Java, however, we can easily make use of the Observer Design Pattern to cause our program to behave in this way.

The Observer pattern assumes that the object containing the data is separate from the objects that display the data, and that these display objects *observe* changes in that data. This is simple to illustrate as we see below.



When we implement the Observer pattern, we usually refer to the data as the Subject and each of the displays as Observers. Each of these observers registers its interest in the data by calling a public method in the Subject. Then, each observer has a known interface that the subject calls when the data change. We could define these interfaces as follows:

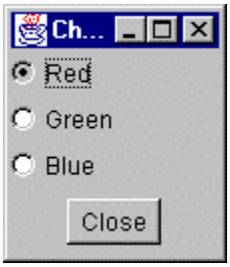
```
abstract interface Observer {
```

```
//notify the Observers that a change has taken place
   public void sendNotify(String s);
}
//===========================================
abstract interface Subject {
//tell the Subject you are interested in  changes
   public void registerInterest(Observer obs);
}
```

The advantage of defining these abstract interfaces is that you can write any sort of class objects you want as long as they implement these interfaces, and that you can declare these objects to be of type Subject and Observer no matter what else they do.

Watching Colors Change

Let's write a simple program to illustrate how we can use this powerful concept. Our program shows a display frame containing 3 radio buttons named Red, Blue and Green as shown below:



This main window is the Subject or data repository object. We create this window using the JFC classes in the following simple code:

```
public class Watch2L extends JFrame
   implements ActionListener, ItemListener, Subject {
   Button Close;
   JRadioButton red, green, blue;
   Vector observers;
//---------------------------------------
   public Watch2L()    {
   super("Change 2 other frames");
//list of observing frames
   observers = new Vector();
//add panel to content pane
   JPanel p = new JPanel(true);
   p.setLayout(new BorderLayout());
   getContentPane().add("Center", p);

//vertical box layout
   Box box = new Box(BoxLayout.Y_AXIS);
```

```
   p.add("Center", box);
//add 3 radio buttons
   box.add(red = new JRadioButton("Red"));
   box.add(green = new JRadioButton("Green"));
   box.add(blue = new JRadioButton("Blue"));

 //listen for clicks on radio buttons
   blue.addItemListener(this);
   red.addItemListener(this);
   green.addItemListener(this);

 //make all part of same button group
   ButtonGroup bgr = new ButtonGroup();
   bgr.add(red);
   bgr.add(green);
   bgr.add(blue);
```

Note that our main frame class implements the Subject interface. That means that it must provide a public method for registering interest in the data in this class. This method is the *registerInterest* method, which just adds Observer objects to a Vector:

```
public void registerInterest(Observer obs)    {
      //adds observer to list in Vector
      observers.addElement(obs);
   }
```

Now we create two observers, once which displays the color (and its name) and another which adds the current color to a list box.

```
//---------create observers---------
   ColorFrame cframe = new ColorFrame(this);
   ListFrame lframe = new ListFrame(this);
```

When we create our ColorFrame window, we register our interest in the data in the main program:

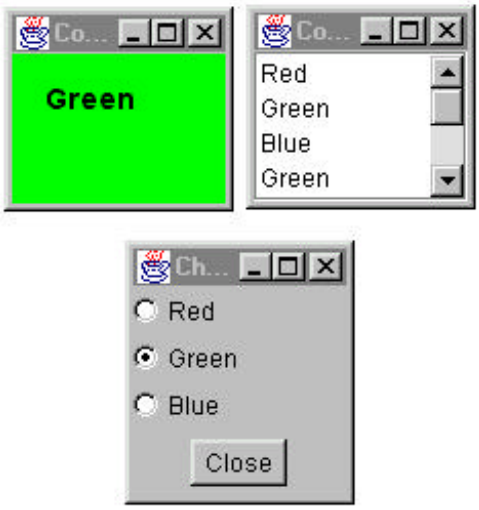
```
class ColorFrame extends Jframe
  implements Observer {
   Color color;
   String color_name="black";
   JPanel p = new JPanel(true);
//-------------------------------------
   public ColorFrame(Subject s)    {
    super("Colors");             //set frame caption
    getContentPane().add("Center",  p);
    s.registerInterest(this);  //register with Subject
    setBounds(100, 100, 100, 100);
    setVisible(true);
   }
```

```
//-------------------------------------
   public void sendNotify(String s)    {
       //Observer is notified of change here
      color_name = s;      //save color name
       //set background to that color
      if(s.toUpperCase().equals("RED"))
         color = Color.red;
      if(s.toUpperCase().equals("BLUE"))
        color =Color.blue;
      if(s.toUpperCase().equals("GREEN"))
        color = Color.green;
      setBackground(color);
   }
//-------------------------------------
   public void paint(Graphics g)    {
      g.drawString(color_name, 20, 50);
   }
```

Meanwhile in our main program, every time someone clicks on one of the radio buttons, it calls the *sendNotify* method of each Observer who has registered interest in these changes by simply running through the objects in the observers Vector:

```
   public void itemStateChanged(ItemEvent e)    {
      //responds to radio button clicks
      //if the button is selected
      if(e.getStateChange() == ItemEvent.SELECTED)
         notifyObservers((JRadioButton)e.getSource());
   }
   //---------------------------------------
   private void notifyObservers(JRadioButton rad)    {
      //sends text of selected button to all observers
      String color = rad.getText();
      for (int i=0; i< observers.size(); i++)   {

((Observer)(observers.elementAt(i))).sendNotify(color);
      }
   }
```

In the case of the ColorFrame observer, the sendNotify method changes the background color and the text string in the frame panel. In the case of the ListFrame observer, however, it just adds the name of the new color to the list box. We see the final program running below:



The Message to the Media

Now, what kind of notification should a subject send to its observers? In this carefully circumscribed example, the notification message is the string representing the color itself. When we click on one of the radio buttons, we can get the caption for that button and send it to the observers. This, of course, assumes that all the observers can handle that string representation. In more realistic situations, this might not always be the case, especially if the observers could also be used to observe other data objects. Here we undertake two simple data conversions:

1. we get the label from the radio button and send it to the observers, and
2. we convert the label to an actual color in the ColorFrame observer.

In more complicated systems, we might have observers that demand specific, but different, kinds of data. Rather than have each observer convert the message to the right data type, we could use an intermediate Adapter class to perform this conversion.

Another problem observers may have to deal with is the case where the data of the central subject class can change in several ways. We could delete points from a list of data, edit their values, or change the scale of the data we are viewing. In these cases we either need to send different change messages to the observers or send a single message and then have the observer ask which sort of change has occurred.

Th JList as an Observer

Now, what about that list box in our color changing example? We saved it for last because as we noted earlier in the Adapter class discussion, the JList is rather different in concept than the List object in the AWT. You can display a fixed list of data in the JList by simply putting the data into a Vector or String array. However, if you want to display a list of data that might grow or otherwise change, you need to put that data into a special data object derived from the AbstractListModel class, and then use that class in the constructor to the JList class. Our ListFrame class looks like this:

```
class ListFrame extends JFrame
   implements Observer {
   JList list;
   JPanel p;
   JScrollPane lsp;
   JListData listData;

   public ListFrame(Subject s)    {
    super("Color List");
    //put panel into the frmae
    p = new JPanel(true);
    getContentPane().add("Center",  p);
    p.setLayout(new BorderLayout());
    //Tell the Subject we are interested
    s.registerInterest(this);

    //Create the list
    listData = new JListData();  //the list model
    list = new JList(listData);  //the visual list
    lsp = new JScrollPane();     //the scroller
    lsp.getViewport().add(list);
    p.add("Center", lsp);
    lsp.setPreferredSize(new Dimension(100,100));
    setBounds(250, 100, 100, 100);
    setVisible(true);
   }
   //--------------------------------
   public void sendNotify(String s)    {
      listData.addElement(s);
```

```
   }
}
```

We name our ListModel class **JListData**. It holds the Vector that contains the growing list of color names.

```
class JListData extends AbstractListModel {
   private Vector data;    //the color name list
   public JListData()    {
      data = new Vector();
   }
   public int getSize()    {
      return data.size();
   }
   public Object getElementAt(int index)    {
      return data.elementAt(index);
   }
  //add string to list and tell the list about it
   public void addElement(String s)    {
      data.addElement(s);
      fireIntervalAdded(this, data.size()-1, data.size());
   }
}
```

Whenever the ColorList class is notified that the color has changed, it calls the *addElement* method of the JListData class. This method adds the string to the Vector, and then calls the *fireIntervalAdded* method. This base method of the AbstractListModel class connects to the JList class, telling that class that the data have changed. The JList class then redisplays the data as needed. There are also equivalent methods for two other kinds of changes: *fireIntervalRemoved* and *fireContentsChanged*. These represent the 3 kinds of changes that can occur in a list box: here each sends its own message to the JList display.

The MVC Architecture as an Observer

As we noted in Chapter 3, the JList, JTable, and JTree objects all operate as observers of a data model. In fact, all of the visual components derived from JComponent can have this same division of labor between the data and the visual representation. In JFC parlance, this is referred to as the Model-View-Controller (MVC) architecture, where the data are represented by the Model, and the View by the visual component. The Controller is the communication between the Model and View objects, and may be a separate class or it may be inherent in either the model or the view. This is the case for the JFC components, and they are all examples of the Observer pattern we've just been discussing.

Consequences of the Observer Pattern

Observers promote abstract coupling to Subjects. A subject doesn't know the details of any of its observers. However, this has the potential disadvantage of successive or repeated updates to the Observers when there are a series of incremental changes to the data. If the cost of these updates is high, it may be necessary to introduce some sort of change management, so that the Observers are not notified too soon or too frequently.

When one client makes a change in the underlying data, you need to decide which object will initiate the notification of the change to the other observers. If the Subject notifies all the observers when it is changed, each client is not responsible for remembering to initiate the notification. On the other hand, this can result in a number of small successive updates being triggered. If the clients tell the Subject when to notify the other clients, this cascading notification can be avoided, but the clients are left with the responsibility of telling the Subject when to send the notifications. If one client "forgets," the program simply won't work properly.

Finally, you can specify the kind of notification you choose to send by defining a number of update methods for the Observers to receive depending on the type or scope of change. In some cases, the clients will thus be able to ignore some of these notifications

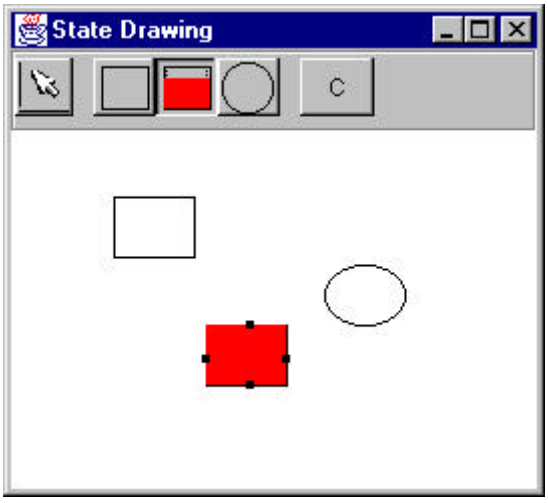
THE STATE PATTERN

The State pattern is used when you want to have an enclosing class switch between a number of related contained classes, and pass method calls on to the current contained class. *Design Patterns* suggests that the State pattern switches between internal classes in such a way that the enclosing object appears to change its class. In Java, at least, this is a bit of an exaggeration, but the actual purpose to which the classes are put can change significantly.

Many programmers have had the experience of creating a class which performs slightly different computations or displays different information based on the arguments passed into the class. This frequently leads to some sort of *switch* or *if-else* statements inside the class that determine which behavior to carry out. It is this inelegance that the State pattern seeks to replace.

Sample Code

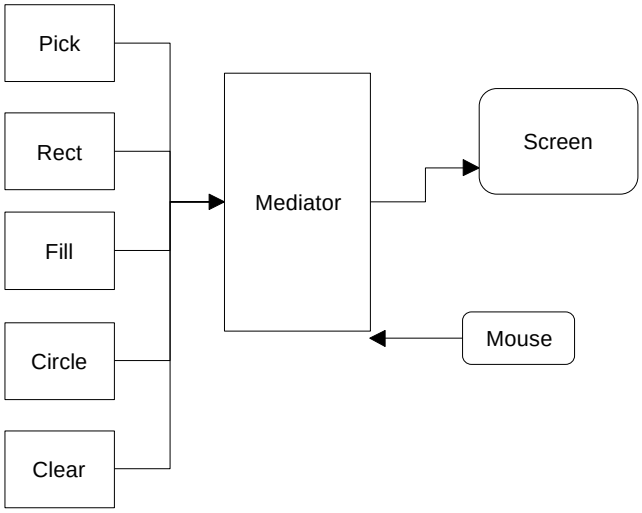
Let's consider the case of a drawing program similar to the one we developed for the Memento class. Our program will have toolbar buttons for Select, Rectangle, Fill, Circle and Clear.



Each one of the tool buttons does something rather different when it is selected and you click or drag your mouse across the screen. Thus, the *state*

of the graphical editor affects the behavior the program should exhibit. This suggests some sort of design using the State pattern.

Initially we might design our program like this, with a Mediator managing the actions of 5 command buttons:



However, this initial design puts the entire burden of maintaining the state of the program on the Mediator, and we know that the main purpose of a Mediator is to coordinate activities between various controls, such as the buttons. Keeping the state of the buttons and the desired mouse activity inside the Mediator can make it unduly complicated as well as leading to a set of *if* or *switch* tests which make the program difficult to read and maintain.

Further, this set of large, monolithic conditional statements might have to be repeated for each action the Mediator interprets, such as mouseUp, mouseDrag, rightClick and so forth. This makes the program very hard to read and maintain.

Instead, let's analyze the expected behavior for each of the buttons:

1. If the Pick button is selected, clicking inside a drawing element should cause it to be highlighted or appear with "handles." If the mouse is dragged and a drawing element is already selected, the element should move on the screen.
2. If the Rect button is selected, clicking on the screen should cause a new rectangle drawing element to be created.

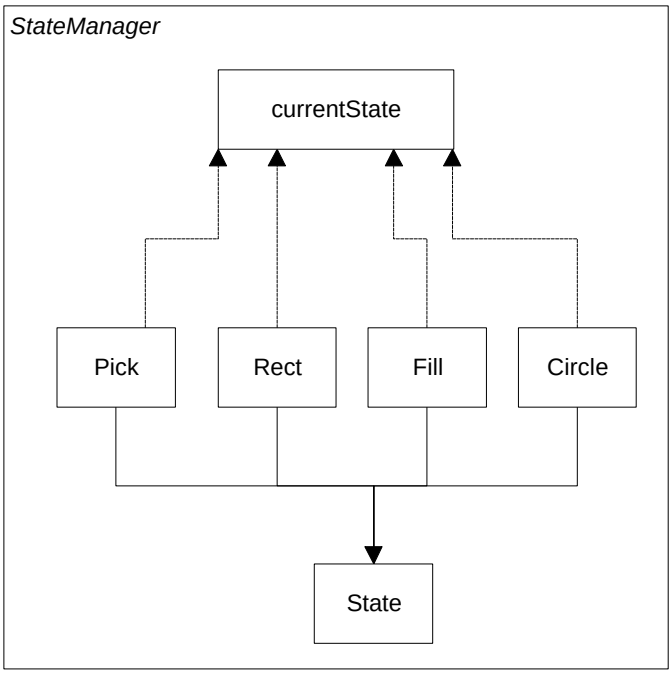
3. If the Fill button is selected and a drawing element is already selected, that element should be filled with the current color. If no drawing is selected, then clicking inside a drawing should fill it with the current color.
4. If the Circle button is selected, clicking on the screen should cause a new circle drawing element to be created.
5. If the Clear button is selected, all the drawing elements are removed.

There are some common threads among several of these actions we should explore. Four of them use the mouse click event to cause actions. One uses the mouse drag event to cause an action. Thus, we really want to create a system that can help us redirect these events based on which button is currently selected.

Let's consider creating a State object that handles mouse activities:

```
public class State {
public void mouseDown(int x, int y){}
public void mouseUp(int x, int y){}
public void mouseDrag(int x, int y){}
}
```

We'll include the mouseUp event in case we need it later. Since none of the cases we've described need all of these events, we'll give our base class empty methods rather than creating an abstract base class. Then we'll create 4 derived State classes for Pick, Rect, Circle and Fill and put instances of all of them inside a StateManager class which sets the current state and executes methods on that state object. In *Design Patterns,* this StateManager class is referred to as a *Context*. This object is illustrated below:



A typical State object simply overrides those event methods that it must handle specially. For example, this is the complete Rectangle state object:

```
public class RectState extends State
{
   private Mediator med;     //save the Mediator
   public RectState(Mediator md)     {
      med = md;
   }
   //-------------------------------------
   //create a new Rectangle where mouse clicks
   public void mouseDown(int x, int y)    {
      med.addDrawing(new visRectangle(x, y));
   }
}
```

The RectState object simply tells the Mediator to add a rectangle drawing to the drawing list. Similarly, the Circle state object tells the Mediator to add a circle to the drawin list:

```
public class CircleState extends State
{
   private Mediator med;        //save Mediator
   public CircleState(Mediator md)    {
```

```
      med = md;
   }
   //--------------------------------
   //Draw circle where mouse clicks
   public void mouseDown(int x, int y)    {
     med.addDrawing(new visCircle(x, y));
   }
}
```

The only tricky button is the Fill button, because we have defined two actions for it.

1. If an object is already selected, fill it.
2. If the mouse is clicked inside an object, fill that one.

In order to carry out these tasks, we need to add the *select* method to our base State class. This method is called when each tool button is selected:

```
public class State
{
public void mouseDown(int x, int y){}
public void mouseUp(int x, int y){}
public void mouseDrag(int x, int y){}
public void select(Drawing d, Color c){}
}
```

The Drawing argument is either the currently selected Drawing or null if none is selcted, and the color is the current fill color. In this simple program, we have arbitrarily set the fill color to red. So our Fill state class becomes:

```
public class FillState extends State
{
   private Mediator med;   //save Mediator
   private Color color;    //save current color
   public FillState(Mediator md)   {
      med = md;
   }
//------------------------------
   //Fill drawing if selected
   public void select(Drawing d, Color c)    {
      color = c;
       if(d!= null)
         {
         d.setFill(c);  //fill that drawing
         }
   }
    //--------------------------------
   //Fill drawing if you click inside one
   public void mouseDown(int x, int y)    {
      Vector drawings = med.getDrawings();
      for(int i=0; i< drawings.size(); i++)
```

```
      {
       Drawing d = (Drawing)drawings.elementAt(i);
       if(d.contains(x, y))
          d.setFill(color); //fill drawing
      }
   }
}
```

Switching Between States

Now that we have defined how each state behaves when mouse events are sent to it, we need to discuss how the StateManager switches between states; we simply set the currentState variable to the state is indicated by the button that is selected.

```
import java.awt.*;

public class StateManager
{
 private State currentState;
 RectState rState;         //states are kept here
 ArrowState aState;
 CircleState cState;
 FillState fState;

 public StateManager(Mediator med)
 {
  rState = new RectState(med);   //create instances
  cState = new CircleState(med); //of each state
  aState = new ArrowState(med);
  fState = new FillState(med);
  currentState = aState;
 }
//These methods are called when the tool buttons
//are selected
 public void setRect()  { currentState = rState;  }
 public void setCircle(){ currentState = cState;  }
 public void setFill()  { currentState = fState;  }
 public void setArrow() { currentState = aState;  }
```

Note that in this version of the StateManager, we create an instance of each state during the constructor and copy the correct one into the state variable when the set methods are called. It would also be possible to use a Factory to create these states on demand. This might be advisable if there are a large number of states which each consume a fair number of resources.

The remainder of the state manager code simply calls the methods of whichever state object is current. This is the critical piece -- there is no

conditional testing. Instead, the correct state is already in place and its methods are ready to be called.

```
 public void mouseDown(int x, int y)  {
    currentState.mouseDown(x, y);
 }
 public void mouseUp(int x, int y)  {
    currentState.mouseUp(x, y);
 }
 public void mouseDrag(int x, int y)  {
    currentState.mouseDrag(x, y);
 }
 public void select(Drawing d, Color c)  {
    currentState.select(d, c);
 }
}
```

How the Mediator Interacts with the State Manager

We mentioned that it is clearer to separate the state management from the Mediator's button and mouse event management. The Mediator is the critical class, however, since it tells the StateManager when the current program state changes. The beginning part of the Mediator illustrates how this state change takes place:

```
public Mediator() {
   startRect = false;
   dSelected = false;
   drawings = new Vector();
   undoList = new Vector();
   stMgr = new StateManager(this);
}
//-----------------------------------------
public void startRectangle() {
   stMgr.setRect();       //change to rectangle state
   arrowButton.setSelected(false);
   circButton.setSelected(false);
   fillButton.setSelected(false);
 }
//-----------------------------------------
public void startCircle() {
   stMgr.setCircle();     //change to circle state
   rectButton.setSelected(false);
   arrowButton.setSelected(false);
   fillButton.setSelected(false);
}
```

These *startXxx* methods are called from the Execute methods of each button as a Command object.

Consequences of the State Pattern

1. The State pattern localizes state-specific behavior in an individual class for each state, and puts all the behavior for that state in a single object.
2. It eliminates the necessity for a set of long, look-alike conditional statements scattered through the program's code.
3. It makes transition explicit. Rather than having a constant that specifies which state the program is in, and that may not always be checked correctly, this makes the change explicit by copying one of the states to the state variable.
4. State objects can be shared if they have no instance variables. Here only the Fill object has instance variables, and that color could easily be made an argument instead.
5. This approach generates a number of small class objects, but in the process, simplifies and clarifies the program.
6. In Java, all of the States must inherit from a common base class, and they must all have common methods, although some of those methods can be empty. In other languages, the states can be implemented by function pointers with much less type checking, and, of course, greater chance of error.

State Transitions

The transition between states can be specified internally or externally. In our example, the Mediator tells the StateManager when to switch between states. However, it is also possible that each state can decide automatically what each successor state will be. For example, when a rectangle or circle drawing object is created, the program could automatically switch back to the Arrow-object State.

Thought Questions

1. Rewrite the StateManager to use a Factory pattern to produce the states on demand.

2. While visual graphics programs provide obvious examples of State patterns, Java server programs can benefit by this approach. Outline a simple server which uses a state pattern.

THE STRATEGY PATTERN

The Strategy pattern is much like the State pattern in outline, but a little different in intent. The Strategy pattern consists of a number of related algorithms encapsulated in a driver class called the Context. Your client program can select one of these differing algorithms or in some cases the Context might select the best one for you. The intent, like the State pattern, is to switch easily between algorithms without any monolithic conditional statements. The difference between State and Strategy is that the user generally chooses which of several strategies to apply and that only one strategy at a time is likely to be instantiated and active within the Context class. By contrast, as we have seen, it is likely that all of the different States will be active at once and switching may occur frequently between them. In addition, Strategy encapsulates several algorithms that do more or less the same thing, while State encapsulates related classes that each do something somewhat different. Finally, the concept of transition between different states is completely missing in the Strategy pattern.

Motivation

A program which requires a particular service or function and which has several ways of carrying out that function is a candidate for the Strategy pattern. Programs choose between these algorithms based on computational efficiency or user choice. There can be any number of strategies and more can be added and any of them can be changed at any time.

There are a number of cases in programs where we'd like to do the same thing in several different ways. Some of these are listed in the *Smalltalk Companion*:

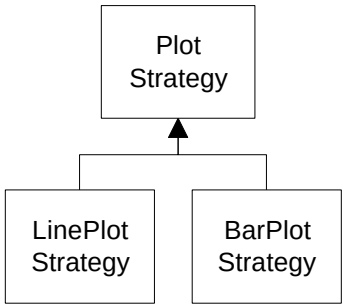
- Save files in different formats.
- Compress files using different algorithms
- Capture video data using different compression schemes
- Use different line-breaking strategies to display text data.
- Plot the same data in different formats: line graph, bar chart or pie chart.

In each case we could imagine the client program telling a driver module (Context) which of these strategies to use and then asking it to carry out the operation.

The idea behind Strategy is to encapsulate the various strategies in a single module and provide a simple interface to allow choice between these strategies. Each of them should have the same programming interface, although they need not all be members of the same class hierarchy. However, they do have to implement the same programming interface.

Sample Code

Let's consider a simplified graphing program that can present data as a line graph or a bar chart. We'll start with an abstract PlotStrategy class and derive the two plotting classes from it:



Since each plot will appear in its own frame, our base PlotStrategy class will be derived from JFrame:

```
public abstract class PlotStrategy extends JFrame
{
  protected float[] x, y;
  protected Color color;
  protected int width, height;

  public PlotStrategy(String title)   {
     super(title);
     width = 300;      height =200;
     color = Color.black;
     addWindowListener(new WindAp(this));
  }
  //-------------------------------------
  public abstract void plot(float xp[], float yp[]);
  //-------------------------------------
  public void setPenColor(Color c)   {
```

```
     color = c;
  }
```

The important part is that all of the derived classes must implement a method called *plot* with two float arrays as arguments. Each of these classes can do any kind of plot that is appropriate.

Note that we don't derive it from our special JxFrame class, because we don't want the entire program to exit if we close one of these subsidiary windows. Instead, we add a WindowAdapter class that just hides the window if it is closed.

```
class WindAp extends WindowAdapter
{
   JFrame fr;
   public WindAp(JFrame f)    {
      fr = f;              //copy Jframe instance
   }
   public void WindowClosing(WindowEvent e)    {
    fr.setVisible(false);  //hide window
   }
}
```

The Context

The Context class is the traffic cop that decides which strategy is to be called. The decision is usually based on a request from the client program, and all that the Context needs to do is to set a variable to refer to one concrete strategy or another.

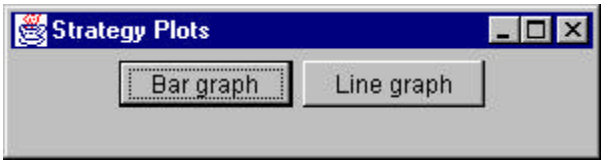
```
public class Context
{
 //this object selects one of the strategies
 //to be used for plotting
 //the plotStrategy variable points to selected strategy
   private PlotStrategy plotStrategy;
   float x[], y[];        //data stored here
//---------------------------------
  public Context() {
 setLinePlot(); //make sure it is not null
}
//---------------------------------
//make current strategy the Bar Plot
  public void setBarPlot()
{ plotStrategy = new BarPlotStrategy(); }
//---------------------------------
//make current strategy the Line Plot
  public void setLinePlot()
{ plotStrategy = new LinePlotStrategy(); }
//---------------------------------
```

```
//call plot method of current strategy
  public void plot() {
     plotStrategy.plot(x, y);
  }
//---------------------------------
  public void setPenColor(Color c) {
     plotStrategy.setPenColor(c);
  }
  //---------------------------------
  public void readData(String filename)
{      //read data from datafile somehow
}
}
```

The Context class is also responsible for handling the data. Either it obtains the data from a file or database or it is passed in when the Context is created. Depending on the magnitude of the data, it can either be passed on to the plot strategies or the Context can pass an instance of itself into the plot strategies and provide a public method to fetch the data.

The Program Commands

This simple program is just a panel with two buttons that call the two plots:



Each of the buttons is a command object that sets the correct strategy and then calls the Context's plot routine. For example, here is the complete Line graph button class:

```
public class JGraphButton extends JButton
       implements Command
{
   Context context;
   public JGraphButton(ActionListener act, Context ctx)
   {
   super("Line graph");          //button label
   addActionListener(act);       //add listener
   context  = ctx;               //copy context
   }
   //-------------------------------
   public void Execute()    {
      context.setPenColor(Color.red); //set color of plot
      context.setLinePlot();          //set kind of plot
```

```
      context.readData("data.txt");   //read the data
      context.plot();                 //plot the data
   }
}
```

The Line and Bar Graph Strategies

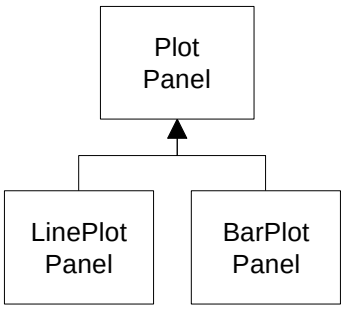
The two strategy classes are pretty much the same: they set up the window size for plotting and call a plot method specific for that display panel. Here is the Line graph Strategy:

```
public class LinePlotStrategy extends PlotStrategy
{
   LinePlotPanel lp;
         public LinePlotStrategy()
  {
     super("Line plot");
     lp = new LinePlotPanel();
     getContentPane().add(lp);
  }
//--------------------------------------
   public void plot(float[] xp, float[] yp)
   {
      x = xp;  y = yp;        //copy in data
      findBounds();           //sets maxes and mins
      setSize(width, height);
      setVisible(true);
      setBackground(Color.white);
      lp.setBounds(minX, minY, maxX, maxY);
      lp.plot(xp, yp, color); //set up plot data
      repaint();              //call paint to plot
   }
}
```

Drawing Plots in Java

Since Java GUI is event-driven, you don't actually write a routine that draws lines on the screen in direct response to the plot command event. Instead you provide a panel whose *paint* event carries out the plotting when that event is called. The *repaint()* method shown above ensures that it will be called right away.

We create a PlotPanel class based on JPanel and derive two classes from it for the actual line and bar plots:



The base PlotPanel class contains the common code for scaling the data to the window.

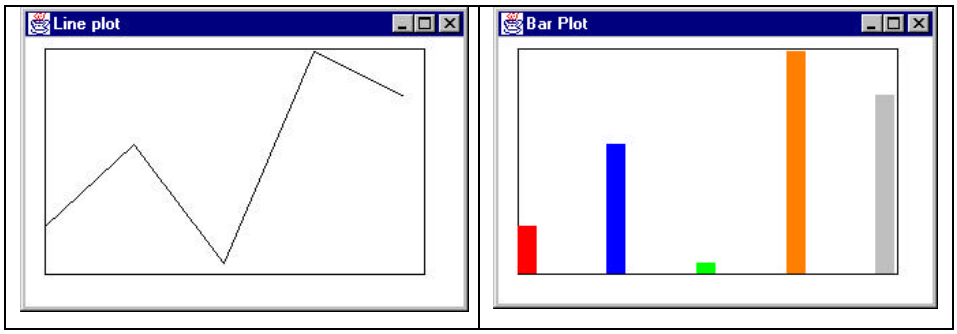
```
public class PlotPanel extends JPanel
{
   float xfactor, yfactor;
   int xpmin, ypmin, xpmax, ypmax;
   float minX, maxX, minY, maxY;
   float x[], y[];
   Color color;
//-----------------------------------------
 public void setBounds(float minx, float miny,
             float maxx, float maxy)   {
     minX=minx;      maxX= maxx;
     minY=miny;      maxY = maxy;
  }
//-----------------------------------------
 public void plot(float[] xp, float[] yp, Color c)  {
  x = xp;      //copy in the arrays
  y = yp;
  color = c;   //and color

  //compute bounds and sclaing factors
  int w = getWidth() - getInsets().left -
                           getInsets().right;
  int h = getHeight() - getInsets().top -
                                  getInsets().bottom;

  xfactor = (0.9f * w) / (maxX - minX);
  yfactor = (0.9f * h)/ (maxY - minY);

  xpmin = (int)(0.05f * w);   ypmin = (int)(0.05f * h);
  xpmax = w - xpmin;   ypmax = h - ypmin;
  repaint();      //this causes the actual plot
 }
//-------------------------------------
protected int calcx(float xp) {
   return (int)((xp-minX) * xfactor + xpmin);
```

```
}
 protected int calcy(float yp) {
   int ypnt = (int)((yp-minY) * yfactor);
   return ypmax - ypnt;
 }
}
```

The two derived classes simply implement the *paint* method for the two kinds of graphs. Here is the one for the Line plot.

```
public class LinePlotPanel extends PlotPanel
{
 public void paint(Graphics g)
 {
   int xp = calcx(x[0]);      //get first point
   int yp = calcy(y[0]);
   g.setColor(Color.white);   //flood background
   g.fillRect(0,0,getWidth(), getHeight());
   g.setColor(Color.black);

   //draw bounding rectangle
   g.drawRect(xpmin, ypmin, xpmax, ypmax);
   g.setColor(color);

   //draw line graph
   for(int i=1; i< x.length; i++)
   {
      int xp1 = calcx(x[i]);        //get n+1st point
      int yp1 = calcy(y[i]);
      g.drawLine(xp, yp, xp1, yp1); //draw line
      xp = xp1;                     //copy for next loop
      yp = yp1;
   }
 }
}
```

The final two plots are shown below:



Consequences of the Strategy Pattern

Strategy allows you to select one of several algorithms dynamically. These algorithms can be related in an inheritance hierarchy or they can be unrelated as long as they implement a common interface. Since the Context switches between strategies at your request, you have more flexibility than if you simply called the desired derived class. This approach also avoids the sort of condition statements than can make code hard to read ad maintain.

On the other hand, strategies don't hide everything. The client code must be aware that there are a number of alternative strategies and have some criteria for choosing among them. This shifts an algorithmic decision to the client programmer or the user.

Since there are a number of different parameters that you might pass to different algorithms, you have to develop a Context interface and strategy methods that are broad enough to allow for passing in parameters that are not used by that particular algorithm. For example the *setPenColor* method in our PlotStrategy is actually only used by the LineGraph strategy. It is ignored by the BarGraph strategy, since it sets up its own list of colors for the successive bars it draws.

THE TEMPLATE PATTERN

Whenever you write a parent class where you leave one or more of the methods to be implemented by derived classes, you are in essence using the Template pattern. The Template pattern formalizes the idea of defining an algorithm in a class, but leaving some of the details to be implemented in subclasses. In other words, if your base class is an abstract class, as often happens in these design patterns, you are using a simple form of the Template pattern.

Motivation

Templates are so fundamental, you have probably used them dozens of times without even thinking about it. The idea behind the Template pattern is that some parts of an algorithm are well defined and can be implemented in the base class, while other parts may have several implementations and are best left to derived classes. Another main theme is recognizing that there are some basic parts of a class that can be factored out and put in a base class so that they do not need to be repeated in several subclasses.

For example, in developing the PlotPanel classes we used in the Strategy pattern examples, we discovered that in plotting both line graphs and bar charts we needed similar code to scale the data and compute the x-and y pixel positions.

```
public class PlotPanel extends JPanel
{
   float xfactor, yfactor;
   int xpmin, ypmin, xpmax, ypmax;
   float minX, maxX, minY, maxY;
   float x[], y[];
   Color color;
//-----------------------------------------
 public void setBounds(float minx, float miny,
             float maxx, float maxy)   {
     minX=minx;      maxX= maxx;
     minY=miny;      maxY = maxy;
  }
//-----------------------------------------
 public void plot(float[] xp, float[] yp, Color c)  {
  x = xp;      //copy in the arrays
  y = yp;
  color = c;   //and color

  //compute bounds and scaling factors
```

```
  int w = getWidth();
  int h = getHeight();
  xfactor = (0.9f * w) / (maxX - minX);
  yfactor = (0.9f * h)/ (maxY - minY);

  xpmin = (int)(0.05f * w);   ypmin = (int)(0.05f * h);
  xpmax = w - xpmin;   ypmax = h - ypmin;
  repaint();      //this causes the actual plot
 }
//-------------------------------------
protected int calcx(float xp) {
   return (int)((xp-minX) * xfactor + xpmin);
}
 protected int calcy(float yp) {
   int ypnt = (int)((yp-minY) * yfactor);
   return ypmax - ypnt;
 }
}
```

Thus, these methods all belonged in a base PlotPanel class without any actual plotting capabilities. Note that the *plot* method sets up all the scaling constants and just calls *repaint*. The actual paint method is deferred to the derived classes. Since the JPanel class always has a *paint* method, we don't want to declare it as an abstract method in the base class, but we do need to override it in the derived classes.

Kinds of Methods in a Template Class

A Template has four kinds of methods that you can make use of in derive classes:

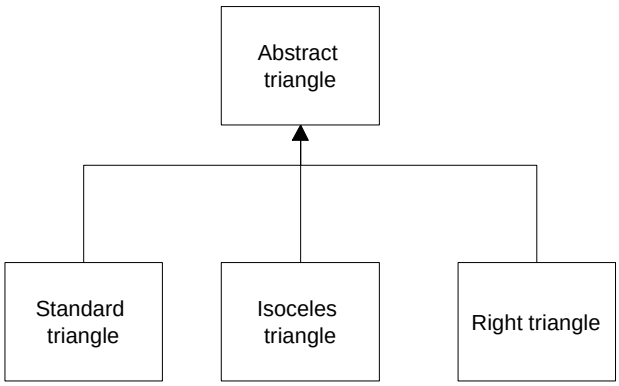
1. Complete methods that carry out some basic function that all the subclasses will want to use, such as *calcx* and *calcy* in the above example. These are called *Concrete methods.*
2. Methods that are not filled in at all and must be implemented in derived classes. In Java , you would declare these as *abstract* methods, and that is how they are referred to in the pattern description.
3. Methods that contain a default implementation of some operations, but which may be overridden in derived classes. These are called *Hook* methods. Of course this is somewhat arbitrary, because in Java you can override any public or

protected method in the derived class, but Hook methods are intended to be overridden, while Concrete methods are not.

4. Finally, a Template class may contain methods which themselves call any combination of abstract, hook and concrete methods. These methods are not intended to be overridden, but describe an algorithm without actually implementing its details. *Design Patterns* refers to these as Template methods.

Sample Code

Let's consider a simple program for drawing triangles on a screen. We'll start with an abstract Triangle class, and then derive some special triangle types from it.



Our abstract Triangle class illustrates the Template pattern:

```
public abstract class Triangle
{
  Point p1, p2, p3;
  //--------------------------------------
  public Triangle(Point a, Point b, Point c)
  {
     //save
     p1 = a; p2 = b; p3 = c;
  }
  //--------------------------------------
  public void draw(Graphics g)
  {
   //This routine draws a general triangle
     drawLine(g, p1, p2);
     Point current = draw2ndLine(g, p2, p3);
     closeTriangle(g, current);
```

```
  }
  //---------------------------------------
  public void drawLine(Graphics g, Point a, Point b)
  {
     g.drawLine(a.x, a.y, b.x, b.y);
  }
  //---------------------------------------
  //this routine has to be implemented
  //for each triangle type.
        abstract public Point
             draw2ndLine(Graphics g, Point a, Point b);
  //---------------------------------------
  public void closeTriangle(Graphics g, Point c)
  {
     //draw back to first point
     g.drawLine(c.x, c.y, p1.x, p1.y);
  }
}
```

This Triangle class saves the coordinates of three lines, but the *draw* routine draws only the first and the last lines. The all important *draw2ndLine* method that draws a line to the third point is left as an abstract method. That way the derived class can move the third point to create the kind of rectangle you wish to draw.

This is a general example of a class using the Template pattern. The *draw* method calls two concrete base class methods and one abstract method that must be overridden in any concrete class derived from Triangle.

Another very similar way to implement the case triangle class is to include default code for the *draw2ndLine* method.

```
public Point draw2ndLine(Graphics g, Point a, Point b)
{
  g.drawLine(a.x, a.y, b.x, b.y);
  return b;
}
```

In this case, the *draw2ndLine* method becomes a Hook method that can be overridden for other classes.

Drawing a Standard Triangle

To draw a general triangle with no restrictions on its shape, we simple implement the *draw2ndLine* method in a derived *stdTriangle* class:

```
public class stdTriangle extends Triangle
{
   public stdTriangle(Point a, Point b, Point c)
   {
```

```
    super(a, b, c);
   }
   public Point draw2ndLine(Graphics g, Point a, Point b)
   {
      g.drawLine(a.x, a.y, b.x, b.y);
      return  b;
   }
}
```

Drawing an Isoceles Triangle

This class computes a new third data point that will make the two sides equal and length and saves that new point inside the class.

```
public class IsocelesTriangle extends Triangle
{
   Point newc;
   int newcx, newcy;
   int incr;

   public IsocelesTriangle(Point a, Point b, Point c)
   {
      super(a, b, c);
      double dx1 = b.x - a.x;   double dy1 = b.y - a.y;
      double dx2 = c.x - b.x;   double dy2 = c.y - b.y;

      double side1 = calcSide(dx1, dy1);
      double side2 = calcSide(dx2, dy2);

      if (side2 < side1)
         incr = -1;
      else
         incr = 1;

      double slope = dy2 / dx2;
      double intercept = c.y - slope* c.x;

      //move point c so that this is an isoceles triangle
      newcx = c.x; newcy = c.y;
      while(Math.abs(side1 - side2) > 1)   {
         newcx += incr;    //iterate a pixel at a time
         newcy = (int)(slope* newcx + intercept);
         dx2 = newcx - b.x;
         dy2 = newcy - b.y;
         side2 = calcSide(dx2, dy2);
      }
      newc = new Point(newcx, newcy);
   }
   //---------------------------------------
   //calculate length of side
   private double calcSide(double dx, double dy)
```

```
   {
      return Math.sqrt(dx*dx + dy*dy);
   }
```

When the Triangle class calls the *draw* method, it calls this new version of *draw2ndLine* and draws a line to the new third point. Further, it returns that new point to the *draw* method so it will draw the closing side of the triangle correctly.

```
//draws 2nd line using saved new point
   public Point draw2ndLine(Graphics g, Point b, Point c)
   {
    g.drawLine(b.x, b.y, newc.x, newc.y);
    return newc;
   }
```

The Triangle Drawing Program

The main program simple creates instances of the triangles you want to draw. Then, it adds them to a Vector in the TPanel class.

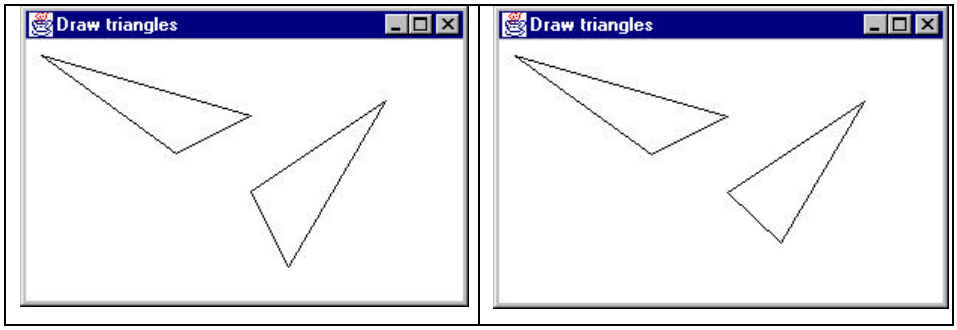
```
public TriangleDrawing()
{
 super("Draw triangles");
 TPanel tp = new TPanel();
 t = new stdTriangle(new Point(10,10), new Point(150,50),
                     new Point(100, 75));
 t1 = new stdTriangle(new Point(150,100), new Point(240,40), \
                           new Point(175, 150));
 tp.addTriangle(t);        //add to triangle list
 tp.addTriangle(t1);       //in the TPanel

 getContentPane().add(tp);
 setSize(300, 200);
 setBackground(Color.white);
 setVisible(true);
}
```

It is the *paint* routine in this class that actually draws the triangles.

```
class TPanel extends Jpanel {
  Vector triangles;
 public TPanel()  {
   triangles = new Vector();   //list of triangles
 }
//-----------------------------------------
 public void addTriangle(Triangle t)    {
   triangles.addElement(t);      //add more to list
 }
//-----------------------------------------
//draw all the triangles
```

```
 public void paint(Graphics g)    {
    for (int i = 0; i < triangles.size(); i++)  {
Triangle tngl = (Triangle)triangles.elementAt(i);
tngl.draw(g);
    }
 }
}
```

An example of two standard triangles is shown below in the left window, and the same code using an isoceles triangle in the right window.



Templates and Callbacks

Design Patterns points out that Templates can exemplify the "Hollywood Principle," or "Don't call us, we'll call you." The idea here is that methods in the base class seem to call methods in the derived classes. The operative word here is *seem*. If we consider the *draw* code in our base Triangle class, we see that there are 3 method calls:

```
drawLine(g, p1, p2);
Point current = draw2ndLine(g, p2, p3);
closeTriangle(g, current);
```

Now *drawLine* and *closeTriangle* are implemented in the base class. However, as we have seen, the *draw2ndLine* method is not implemented at all in the base class, and various derived classes can implement it differently. Since the actual methods that are being called are in the derived classes, it appears as though they are being called from the base class.

If this idea make you uncomfortable, you will probably take solace in recognizing that *all* the method calls originate from the derived class, and that these calls move up the inheritance chain until they find the first class which implements them. If this class is the base class, fine. If not, it could be any

other class in between. Now, when you call the *draw* method, the derived class moves up the inheritance tree until it finds an implementation of draw. Likewise, for each method called from within draw, the derived class starts at the currently class and moves up the tree to find each method. When it gets to the *draw2ndLine* method, it finds it immediately in the current class. So it isn't "really" called from the base class, but it does sort of seem that way.

Summary and Consequences

Template patterns occur all the time in OO software and are neither complex nor obscure in intent. They are normal part of OO programming and you shouldn't try to make them more abstract than they actually are.

The first significant point is that your base class may only define some of the methods it will be using, leaving the rest to be implemented in the derived classes. The second major point is that there may be methods in the base class which call a sequence of methods, some implemented in the base class and some implemented in the derived class. This Template method defines a general algorithm, although the details may not be worked out completely in the base class.

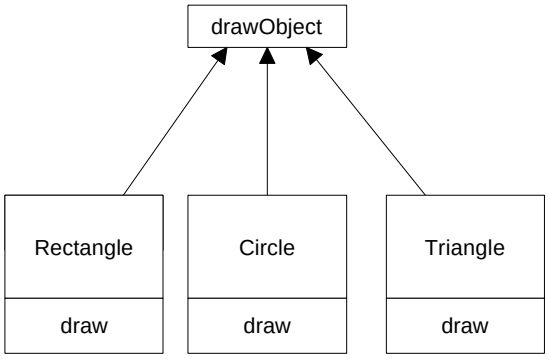
Template classes will frequently have some abstract methods that you must override in the derived classes, and may also have some classes with a simple "place-holder" implementation that you are free to override where this is appropriate. If these place-holder classes are called from another method in the base class, then we refer to these overridable methods are "Hook" methods.

THE VISITOR PATTERN

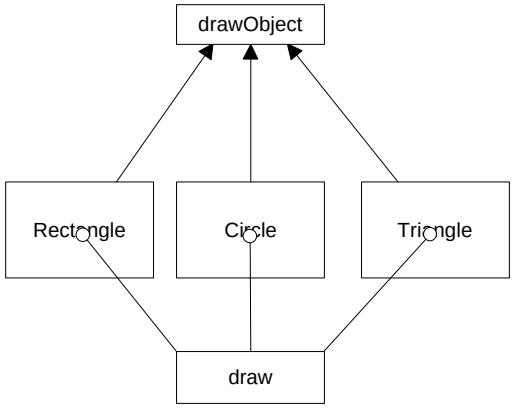
The Visitor pattern turns the tables on our object-oriented model and creates an external class to act on data in other classes. This is useful if there are a fair number of instances of a small number of classes and you want to perform some operation that involves all or most of them.

Motivation

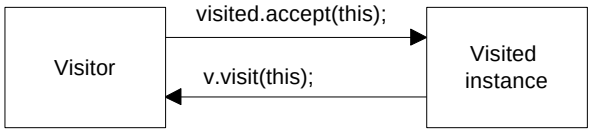
While at first it may seem "unclean" to put operations that should be inside a class in another class instead, there are good reasons for doing it. Suppose each of a number of drawing object classes has similar code for drawing itself. The drawing methods may be different, but they probably all use underlying utility functions that we might have to duplicate in each class. Further, a set of closely related functions is scattered throughout a number of different classes as shown below:



Instead, we write a Visitor class which contains all the related *draw* methods and have it visit each of the objects in succession:



The question that most people who first review this pattern ask is "what does visiting mean?" There is only one way that an outside class can gain access to another class, and that is by calling its public methods. In the Visitor case, visiting each class means that you are calling a method already installed for this purpose, called *accept*. The *accept* method has one argument: the instance of the visitor, and in return, it calls the *visit* method of the Visitor, passing itself as an argument.



Putting it in simple code terms, every object that you want to visit must have the following method:

```
public void accept(Visitor v)
   {
      v.visit(this);      //call visitor method
   }
```

In this way, the Visitor object receives a reference to each of the instances, one by one, and can then call its public methods to obtain data, perform calculations, generate reports, or just draw the object on the screen.

When to Use the Visitor Pattern

You should consider using a Visitor pattern when you want to perform an operation on the data contained in a number of objects that have

different interfaces. Visitors are also valuable if you have to perform a number of unrelated operations on these classes.

On the other hand, as we will see below, Visitors are a good choice only when you do not expect many new classes to be added to your program.

Sample Code

Let's consider a simple subset of the Employee problem we discussed in the Composite pattern. We have a simple Employee object which maintains a record of the employee's name, salary, vacation taken and number of sick days taken. A simple version of this class is:

```
public class Employee
{
   int sickDays, vacDays;
   float Salary;
   String Name;

 public Employee(String name, float salary,
             int vacdays, int sickdays)
 {
  vacDays = vacdays;       sickDays = sickdays;
  Salary = salary;         Name = name;
 }
 public String getName()  { return Name;     }
 public int getSickdays() { return sickDays; }
 public int getVacDays()  { return vacDays;  }
 public float getSalary() { return Salary;   }
 public void accept(Visitor v) { v.visit(this); }
}
```

Note that we have included the *accept* method in this class. Now let's suppose that we want to prepare a report of the number of vacation days that all employees have taken so far this year. We could just write some code in the client to sum the results of calls to each Employee's *getVacDays* function, or we could put this function into a Visitor.

Since Java is a strongly typed language, your base Visitor class needs to have a suitable abstract *visit* method for each kind of class in your program. In this first simple example, we only have Employees, so our basic abstract Visitor class is just

```
public abstract class Visitor
{
  public abstract void visit(Employee emp);
}
```

Notice that there is no indication what the Visitor does with teach class in either the client classes or the abstract Visitor class. We can in fact write a whole lot of visitors that do different things to the classes in our program. The Visitor we are going to write first just sums the vacation data for all our employees:

```
public class VacationVisitor extends Visitor
{
 protected int total_days;
 public VacationVisitor()  {    total_days = 0;  }
 //-----------------------------
 public void visit(Employee emp)
 {
    total_days += emp.getVacDays();
 }
 //-----------------------------
 public int getTotalDays()
 {
    return total_days;
 }
}
```

Visiting the Classes

Now, all we have to do to compute the total vacation taken is to go through a list of the employees and visit each of them, and then ask the Visitor for the total.

```
VacationVisitor vac = new VacationVisitor();
 for (int i = 0; i < employees.length; i++)
 {
  employees[i].accept(vac);
 }
 System.out.println(vac.getTotalDays());
```

Let's reiterate what happens for each visit:

1. We move through a loop of all the Employees.
2. The Visitor calls each Employee's *accept* method.
3. That instance of Employee calls the Visitor's *visit* method.
4. The Visitor fetches the vacation days and adds them into the total.
5. The main program prints out the total when the loop is complete.

Visiting Several Classes

The Visitor becomes more useful, when there are a number of different classes with different interfaces and we want to encapsulate how we get data from these classes. Let's extend our vacation days model by introducing a new Employee type called Boss. Let's further suppose that at this company, Bosses are rewarded with bonus vacation days (instead of money). So the Boss class as a couple of extra methods to set and obtain the bonus vacation day information:

```
public class Boss extends Employee
{
   private int bonusDays;

  public Boss(String name, float salary,
              int vacdays, int sickdays)    {
      super(name, salary, vacdays, sickdays);
   }
   public void setBonusDays(int bonus) { bonusDays = bonus; }
   public int getBonusDays() { return bonusDays; }
   public void accept(Visitor v)  { v.visit(this); }
}
```

When we add a class to our program, we have to add it to our Visitor as well, so that the abstract template for the Visitor is now:

```
public abstract class Visitor
{
  public abstract void visit(Employee emp);
  public abstract void visit(Boss emp);
}
```

This says that any concrete Visitor classes we write must provide polymorphic *visit* methods for both the Employee and the Boss class. In the case of our vacation day counter, we need to ask the Bosses for both regular and bonus days taken, so the visits are now different. We'll write a new bVacationVisitor class that takes account of this difference:

```
public class bVacationVisitor extends Visitor
{
  int total_days;

 public bVacationVisitor() { total_days = 0; }
 public int getTotalDays() { return total_days; }
//--------------------------------
 public void visit(Boss boss) {
    total_days += boss.getVacDays();
    total_days += boss.getBonusDays();
    }
 //----------------------------
 public void visit(Employee emp) {
```

```
 total_days += emp.getVacDays();
 }
}
```

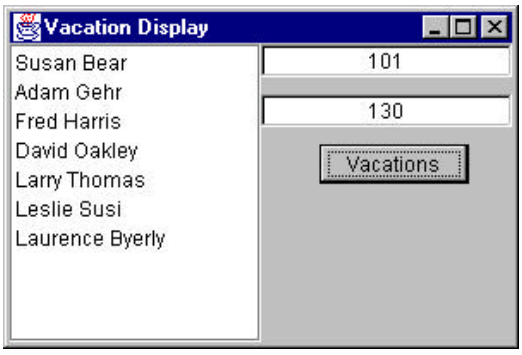
Note that while in this case Boss is derived from Employee, it need not be related at all as long as it has an *accept* method for the Visitor class. It is quite important, however, that you implement a *visit* method in the Visitor for *every class* you will be visiting and not count on inheriting this behavior, since the *visit* method from the parent class is an Employee rather than a Boss visit method. Likewise, each of your derived classes (Boss, Employee, etc. must have its own *accept* method rather than calling one in its parent class.

Bosses are Employees, too

We show below a simple application that carries out both Employee visits and Boss visits on the collection of Employees and Bosses. The original VacationVisitor will just treat Bosses as Employees and get only their ordinary vacation data. The bVacationVisitor will get both.

```
VacationVisitor vac = new VacationVisitor();
bVacationVisitor bvac = new bVacationVisitor();
 for (int i = 0; i < employees.length; i++)
 {
  employees[i].accept(vac);
  employees[i].accept(bvac);
 }
 total.setText(new Integer(vac.getTotalDays()).toString());
 btotal.setText(
         new Integer(bvac.getTotalDays()).toString());
```

The two lines of displayed data represent the two sums that are computed when the user clicks on the Vacations button.



Double Dispatching

No article on the Visitor pattern is complete without mentioning that you are really dispatching a method twice for the Visitor to work. The Visitor calls the polymorphic *accept* method of a given object, and the *accept* method calls the polymorphic *visit* method of the Visitor. It this bidirectional calling that allows you to add more operations on any class that has an *accept* method, since each new Visitor class we write can carry out whatever operations we might think of using the data available in these classes.

Traversing a Series of Classes

The calling program that passes the class instances to the Visitor must know about all the existing instances of classes to be visited and mus keep them in a simple structure such as an array or Vector. Another possibility would be to create an Enumeration of these classes and pass it to the Visitor. Finally, the Visitor itself could keep the list of objects that it is to visit. In our simple example program, we used an array of objects, but any of the other methods would work equally well.

Consequence of the Visitor Pattern

The Visitor pattern is useful when you want to encapsulate fetching data from a number of instances of several classes. *Design Patterns* suggests that the Visitor can provide additional functionality to a class without changing it. We prefer to say that a Visitor can add functionality to a collection of classes and encapsulate the methods it uses.

The Visitor is not magic, however, and cannot obtain private data from classes: it is limited to the data available from public methods. This might force you to provide public methods that you would otherwise not have provided. However, it can obtain data from a disparate collection of unrelated classes and utilize it to present the results of a global calculation to the user program.

It is easy to add new operations to a program using Visitors, since the Visitor contains the code instead of each of the individual classes. Further, Visitors can gather related operations into a single class rather than forcing you to change or derive classes to add these operations. This can make the program simpler to write and maintain.

Visitors are less helpful during a program's growth stage, since each time you add new classes which must be visited, you have to add an abstract

visit operation to the abstract Visitor class, and you must add an implementation for that class to each concrete Visitor you have written. Visitors can be powerful additions when the program reaches the point where many new classes are unlikely.

Visitors can be used very effectively in Composite systems and the boss-employee system we just illustrated could well be a Composite like the one we used in the Composite chapter.

Alexander, Christopher, Ishikawa, Sara, *et. al., A Pattern Language*, Oxford University Press, New York, 1977.

Alpert, S., Brown, K. and Woolf, B., *The Design Patterns Smalltalk Companion*, Addison-Wesley, 1998.

Buschman, F., Meunier, R., Rohnert, H., Sommerlad, P. and Stal, M., *A System of Patterns*, John Wiley and Sons, New York, 1996.

Cooper, J. W., *Principles of Object-Oriented Programming in Java 1.1 Coriolis (Ventana), 1997.*

Coplien, James O. *Advanced C++ Programming Styles and Idioms,* Addison-Wesley, Reading, MA., 1992.

Coplien, James O. and Schmidt, Douglas C., *Pattern Languages of Program Design*, Addison-Wesley, 1995.

Gamma, E., Helm, T., Johnson, R. and Vlissides, J., *Design Patterns: Abstraction and Reuse of Object Oriented Design.* Proceedings of ECOOP '93, 405-431.

Gamma, Eric; Helm, Richard; Johnson, Ralph and Vlissides, John, *Design Patterns. Elements of Reusable Software*., Addison-Wesley, Reading, MA, 1995

Krasner, G.E. and Pope, S.T., A cookbook for using the Model-View-Controller user interface paradigm in Smalltalk-80. *Journal of Object-Oriented Programmng* ***I****(3)., 1988*

Kurata, Deborah, "Programming with Objects," *Visual Basic Programmer's Journal,* June, 1998.

Pree, Wolfgang, *Design Patterns for Object Oriented Software Development*, Addison-Wesley, 1994.

Riel, Arthur J., *Object-Oriented Design Heuristics,* Addison-Wesley, Reading, MA, 1996