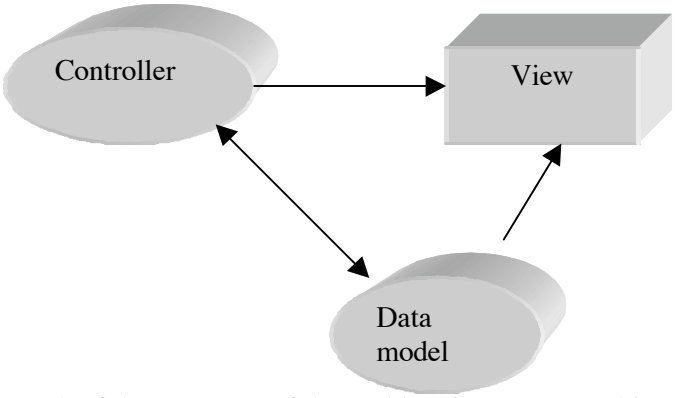
SOME BACKGROUND ON DESIGN PATTERNS

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.

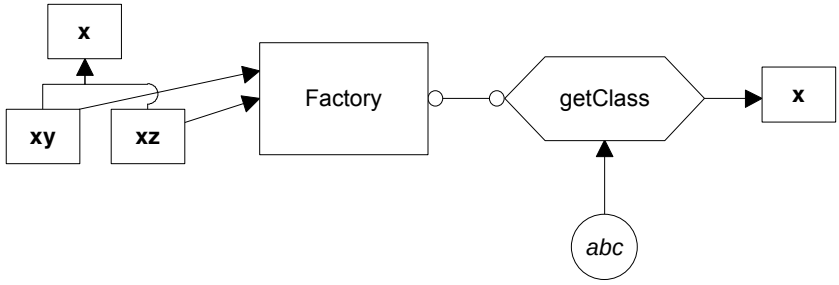
The Singleton Pattern is a class of which there can be no more than one instance. It provides a single global point of access to that instance.

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?

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.

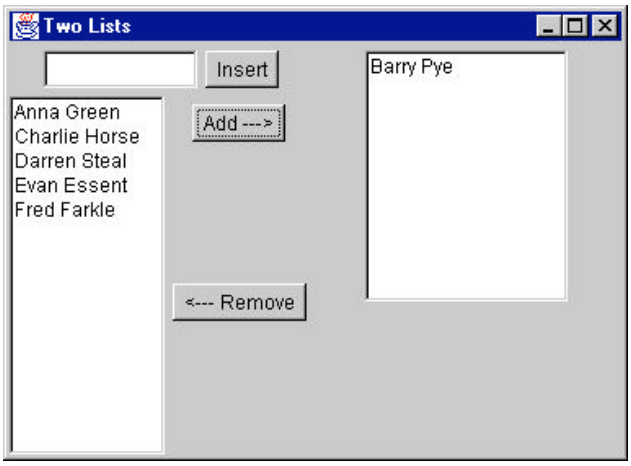
THE ADAPTER PATTERN

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.

Behavioral Patterns

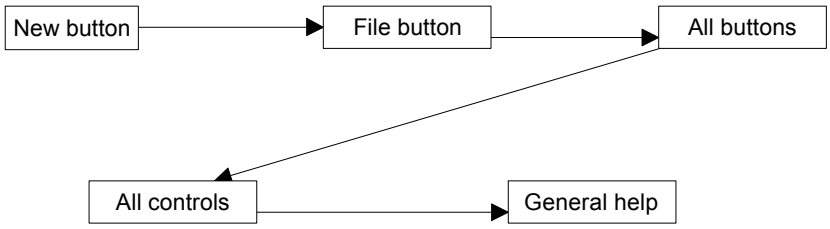
Behavioral patterns are those patterns that are most specifically concerned with communication between objects. In this chapter, we'll see that:

- 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.

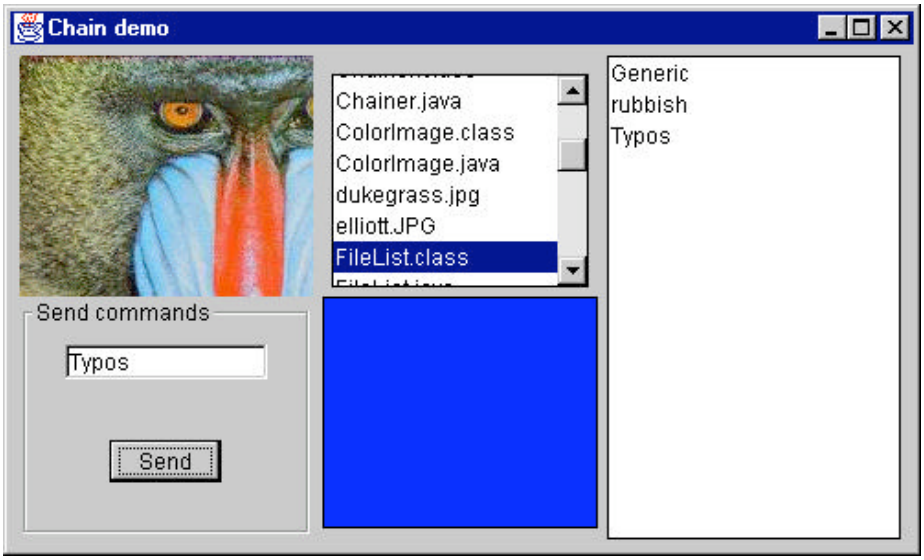
- You want to issue a request to one of several objects without specifying which one explicitly.
- You want to be able to modify the set of objects dynamically that can handle requests.

Sample Code

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

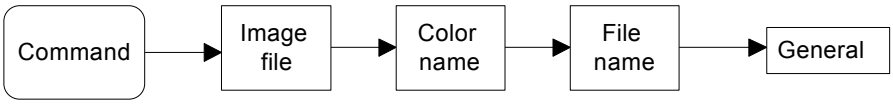
- Image filenames
- General filenames
- Colors
- Other commands

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.

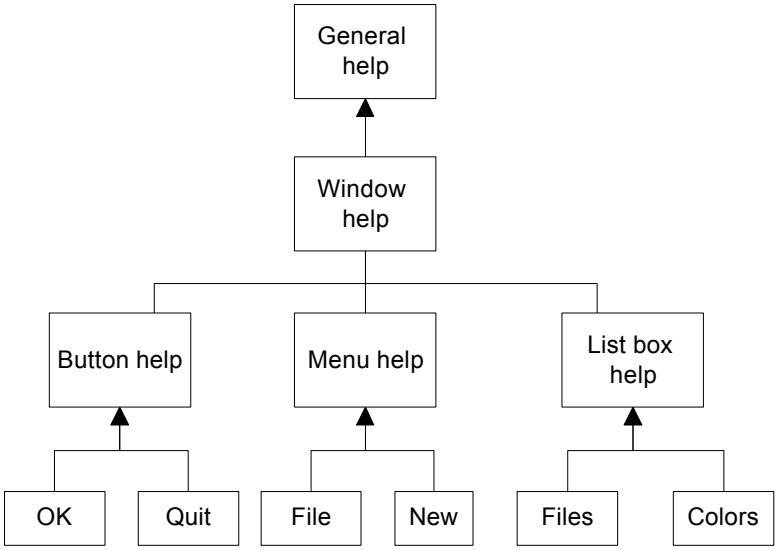
Finally, we link these classes together in the constructor to form the Chain:

```
//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.

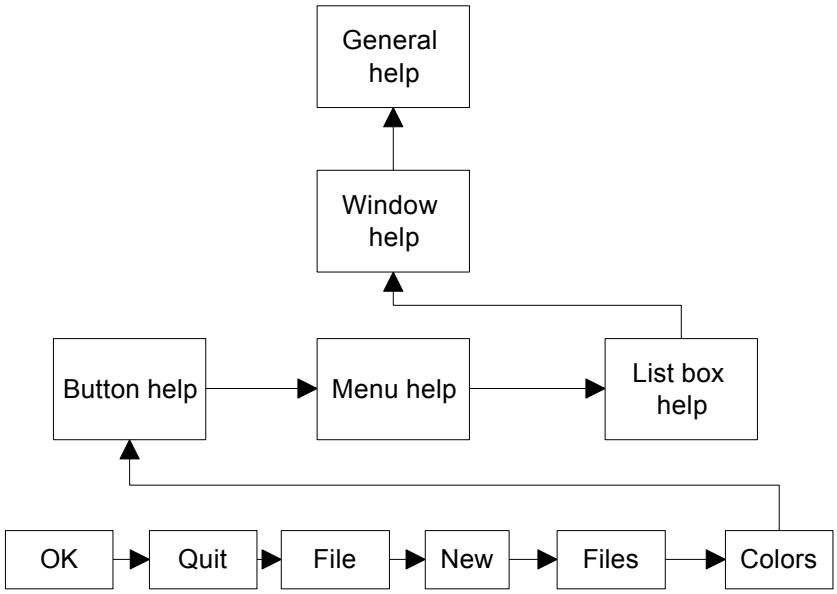
A Chain or a Tree?

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.