

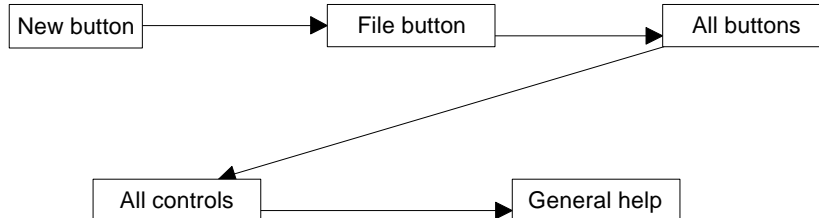
---

## 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.

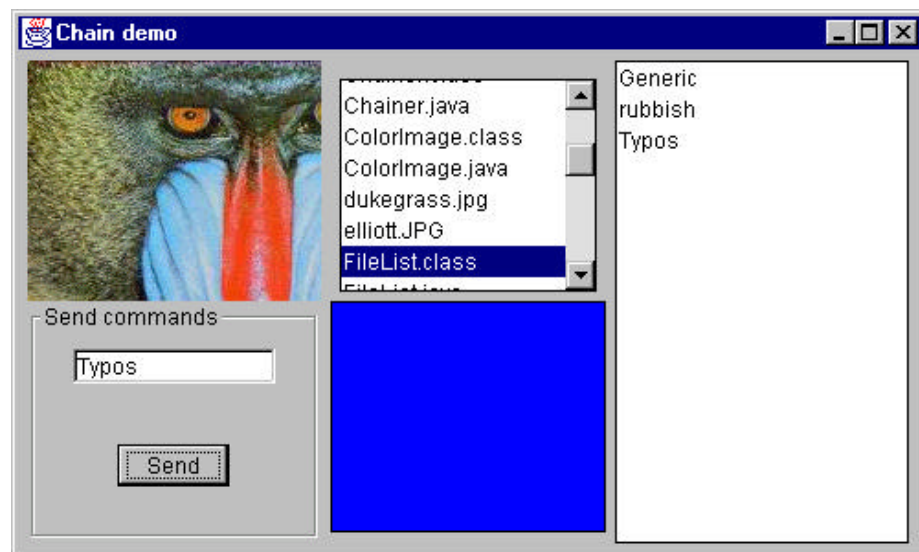
- You want to issue a request to one of several objects without specifying which one explicitly.
- You want to be able to modify the set of objects dynamically that can handle requests.

## Sample Code

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

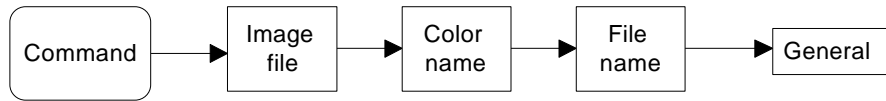
- Image filenames
- General filenames
- Colors
- Other commands

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 msg);
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 msg)
{
    //if there is a JPEG file with this root name
    //load it and display it.
    if (findImage(msg))
        loadImage(msg + ".jpg");
    else
        //Otherwise, pass request along chain
        nextChain.sendToChain(msg);
}
  
```

```

}
//-----
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 msg) {
    Color c = getColor(msg);
    if(c != null) {
        setBackground(c);
        repaint();
    }
    else {
        if (nextChain != null)
            nextChain.sendToChain(msg);
    }
}
//-----
private Color getColor(String msg) {
    String lmsg = msg.toLowerCase();
    Color c = null;

    if(lmsg.equals("red"))
        c = Color.red;
    if(lmsg.equals("blue"))
        c = Color.blue;
    if(lmsg.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(msg);
        if (! found) i++;
    }
    if(found)      {
        setSelectedIndex(i);
    }
    else          {
        if(nextChain != null)
            nextChain.sendToChain(msg);
    }
}

```

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:

```

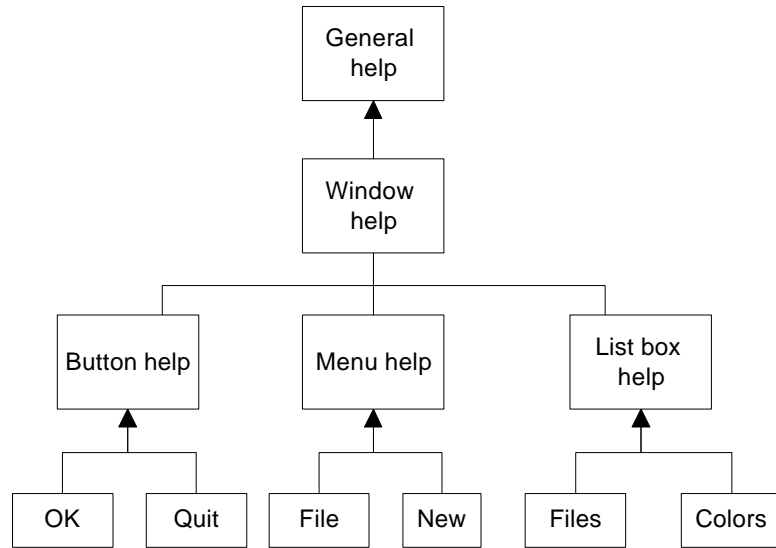
//set up the chain of responsibility
sender.addChain(imager);
imager.addChain(colorImage);
colorImage.addChain(fileList);
fileList.addChain(restList);

```

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

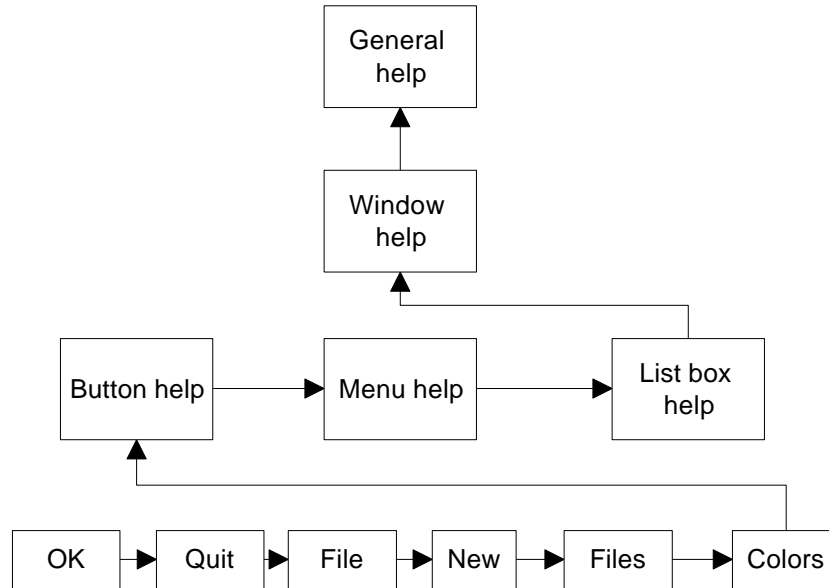
## A Chain or a Tree?

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.