Behavioral Patterns

- Patterns that are most specifically concerned with communication between objects.
  - The Iterator pattern formalizes the way we move through a list of data within a class.
  - 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.
  - 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.

CMSC 433 – Programming Language Technologies and Paradigms Spring 2007

Chain of Responsibility Pattern

Apr. 10, 2007

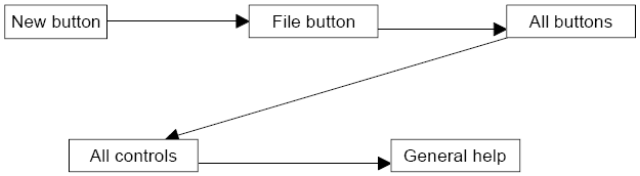
What is it?

- 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.

A Classic Example

- 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.

A Classic Example



- 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.

Note that…

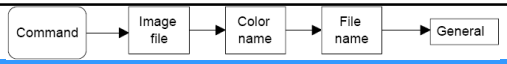
- The chain is organized from most specific to most general.
- There is no guarantee that the request will produce a response in all cases.

Use the Chain of Responsibility when…

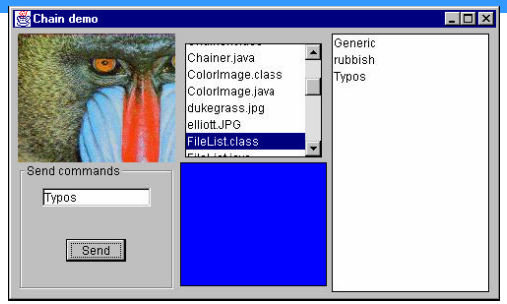
- You have more than one handler that can handle a request and there is no way to know which handler to use. The handler must be determined automatically by the chain.
- You want to issue a request to one of several objects without specifying which one explicitly.
- You want to be able to dynamically modify the set of objects that can handle requests.

A Simple Example

- 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.



The UI



- 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.

We Need…

- 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.

Lets Start Coding!

- Lets start with a Chain interface.

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


Imager Class

- Our Imager class is 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.

Code of Imager Class

```
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);
```


Code of Imager Class (cont…)

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


ColorImage Class

- 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.

Code of ColorImage Class

```
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 Other Two Classes

- Both the file list and the list of unrecognized commands are JList boxes.
- Since we developed an adapter JawtList in a previous lecture 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.

RestList Code

```
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;
  }
}
```


FileList

- 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.

FileList Code

```
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;
```

FileList Code (contd…)

```
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 is a simple child of the File class that contains a *matchRoot* method to compare a string to the root name of a file.

Linking the Chain…

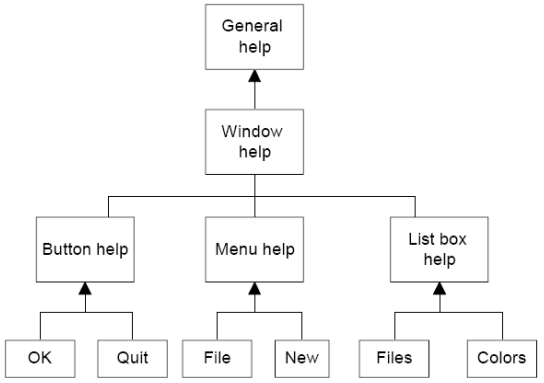
- Finally, we link these classes together in the constructor to form the Chain

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


A Tree

- A Chain of Responsibility does not have to be linear.
- A more general structure is a tree with a number of specific entry points all pointing upward to the most general node.

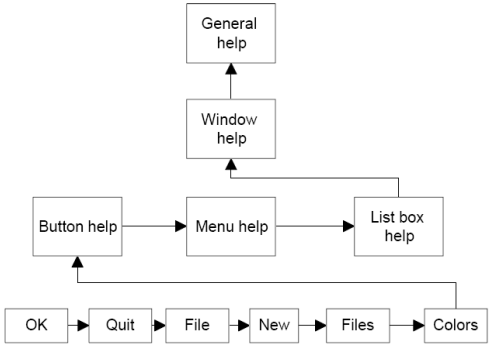
For Example…



Disadvantage

- However, this sort of structure seems to imply that each button, or its 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.
- 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.

The "Tree" Chain



More Complex 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.

Example 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.

Summary

- 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.
- 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.
- 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.
- 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.