The Focus Puller

James W Cooper

If you compulsively read the credit crawl at the end of a movie, you will find credits for "Focus puller," right down there but just above "Gaffer" and "Best Boy." If you indulge in stage theatricals, the actor who "pulls focus" is one who is distracting the audience from the part of the scene they are meant to be watching. If you wrote code for Java 1.3 and found that it somehow doesn't work any more, that will pull your focus, too.

It is this last example that pulled my chain, since I pulled out some old code for the Chain of Responsibility pattern that I had written for a column a couple of years ago. I was putting together a talk and wanted to show this code working. And guess what? It didn't work at all! What an annoyance. But it led me down the path of understanding the Java 1.4 AWT focus subsystem, and I suppose that was worth learning, too.

Let's consider the simple user interface illustrated in Figure 1:

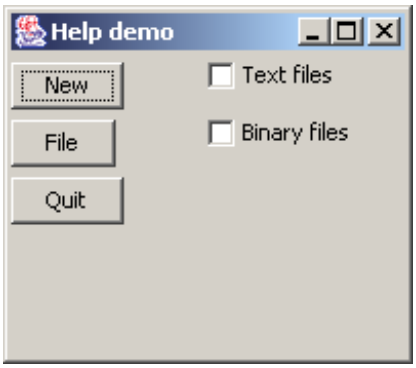

Figure 1 – A simple interface for a Chain of Responsibility program.

In such an interface, we might like to have a help message on using the buttons and checkboxes that appears when we press the F1 key. The message should depend on which control has the focus. In this example, we might have specific help for some buttons and checkboxes and more general help for others. If no specific help is available, we could put up a general help statement. In our example program we display

- Specific help for New.
- Button help for File and Quit.
- Specific help for the Text files checkbox.
- General control help for the Binary files checkbox.
- General help if no control is selected.

The way we do this using a Chain of Responsibility is to create a series of linked help classes, each of which handles a specific button or forwards the request to the next one in the chain until one of them handles the help message.

We do this by having each help class implement the Chain interface:

```
public interface Chain {
```

```
        //add the next chain to this one
        public void addChain(Chain c);
        //handle request or
        //send it to the next chain element
        public void sendToChain(Component c);
        //return the current chain link
        public Chain getChain();
}
```

Then, when an F1 keystroke occurs, we send the control that received the stroke down the chain until one of the help modules recognizes it. This is illustrated in Figure 2.

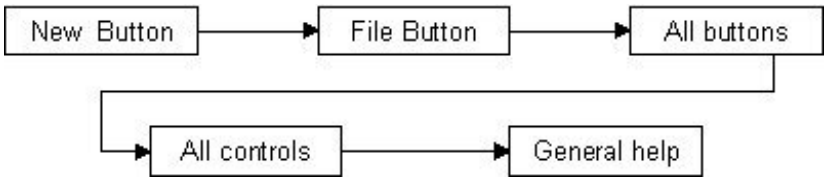

Figure 2 – How the Chain of Responsibility forwards requests

We create a general HelpWindow class and then subclass it for each of the help message classes. The general class is

```
public class HelpWindow implements Chain {
    protected Chain nextChain;
    protected String mesg, title;

    public HelpWindow() {
        mesg ="";
    }
//-----------------------------------------
    public void addChain(Chain c) {
        nextChain = c;
    }
    public Chain getChain() {
        return this;
    }
    public void sendToChain(Component c) {
        sendChain(c);
    }
//-----------------------------------------
    protected void sendChain (Component c) {
        if (nextChain != null) {
            nextChain.sendToChain(c);
        }
    }
}
```

Here is one of the classes for button help

```
public class ButtonHelp extends HelpWindow {
      String message, title;
      int icon;
      JFrame frame;

      public ButtonHelp(JFrame f) {
            mesg = "Click on any button to activate it";
            title = "Button help";
            icon = JOptionPane.DEFAULT_OPTION;
            frame = f;
      }
      public void sendToChain(Component c) {
```

```
            if (c instanceof JButton)
                  JOptionPane.showMessageDialog(frame, mesg,
                        title, icon);
            else {
                  sendChain(c);
            }
      }
}
```

Chains like this are useful not only in help systems, but in implementing compilers and interpreters, or any place where one of several classes may want to handle a request.

Problems on the Chain Gang

Now, when I first wrote about the Chain of Responsibility pattern, I said that you only needed to add a KeyListener to the outer frame, and that it would forward the KeyPress event to all the controls inside the frame.

First we created a simple inner class to send events down the chain:

```
class key_adapter extends KeyAdapter {
      public void keyPressed(KeyEvent e) {
            if (e.getKeyCode() == KeyEvent.VK_F1) {
                  sendToChain((JComponent) getFocusOwner());
            }
      }
      public void keyTyped(KeyEvent e) {
            System.out.println(e.getKeyChar() );
      }
}
```

Then we derive a concrete key_adapeter class from the abstract KeyAdapter class and make it listen for key strokes from the frame:

```
helpKey = new key_adapter();  //derived from KeyAdapter
addKeyListener(helpKey);
```

And, at the time we wrote this, this worked correctly. However, starting in Java 1.4, it didn't work! Key events are not forwarded from the frame to its contained components. One quick way around this is just to add the same KeyAdapter to each control and have its KeyListener forward the request down the chain:

```
helpKey = new key_adapter();
ckText.addKeyListener(helpKey);
ckBinary.addKeyListener(helpKey);
btNew.addKeyListener(helpKey);
btFile.addKeyListener(helpKey);
btQuit.addKeyListener(helpKey);
```

But this is a real pain, because we have to make sure we have assigned a listener to every single control, and if no control has the focus, the frame will not catch the help request, because it can't have the focus.

The AWT Focus Subsystem

What's happened is that in Java 1.4, an entirely new keyboard and focus-handling system has been added and this changes how keyboard events are handled. The designers of Java recognized that there were some deficiencies in the original focus system. First, there was no way to discover what control currently had the focus and there was no way to control

the order in which controls got the focus when you tabbed between them from the keyboard. Also, when a control received the focus, you could not determine which control had just lost the focus.

This new focus system introduces the abstract KeyboardFocusManager class and its concrete implementation the DefaultKeyboardFocusManager. These classes keep track of the active window and the focus traversal cycle among the components in the active window. It also allows for a hierarchy of roots of various traversal cycles. You can create instances of such a focus manager and assign them to manage the focus of controls in your frame windows. If you do not create such a system, focus traversal works the same way it did in the past: the focus order is the order you added the controls to the frame.

One of the interesting properties of these classes is that you can use them to intercept keyboard events before they are passed to the current (or default) focus traversal. We can use this property to receive any keystroke, no matter which control has the focus and then pass it down the chain of responsibility. This also gives us the opportunity to receive a keyboard event sent to the frame when no control has the focus. We can do this in just a few statements:

```
//create a focus manager
DefaultFocusManager dfocus = new DefaultFocusManager();
//set as the current focus manager
KeyboardFocusManager.setCurrentKeyboardFocusManager(dfocus);
//add a keyboard stroke interceptor
dfocus.addKeyEventDispatcher(new KeyIntercept(chain));
```

This KeyEventDispatcher class can be any class that implements the simple KeyEventDispatcher interface:

```
interface KeyEventDispatcher {
      boolean dispatchKeyEvent(KeyEvent e);
}
```

The dispatchKeyEvent method returns true if the key event has been consumed by this method and false if it has not.

In this case, we use our implementation of this interface to kick off our Chain of Responsibility:

```
public class KeyIntercept implements KeyEventDispatcher {
      private Chain chain;
      public KeyIntercept(Chain chn) {
            chain = chn;
      }

      public boolean dispatchKeyEvent(KeyEvent e) {
            //make sure the event is not consumed
            boolean consumed = e.isConsumed();
            //receive only pressed events
            if (e.getID() == KeyEvent.KEY_PRESSED) {
                  //on the F1 key
                  if (e.getKeyCode() == KeyEvent.VK_F1 && !consumed) {
                        //send them down the chain
                        chain.sendToChain((Component) e.getComponent());
                        consumed = true;
                  }
            }
            return consumed; //true means not passed on
```

```
        }
}
```

With the addition of this class and the focus manager, our Java 1.4 program works the same way as our Java 1.3 version did.

There are some tricks here, though. This dispatchKeyEvent method receives all key messages, including both KEY_PRESSED and KEY_RELEASED. We specifically filter out all but KEY_PRESSED. Further, some events will come to this method that have already been consumed. You need to check the state of the evenet's *isConsumed* property and only consume events that have not already been consumed.

Controlling the Focus

But, in addition, this new focus system gives us much more control over how we traverse focus between the controls. When we created the controls in our simple program, we added the buttons one at a time, and the checkboxes last. So the focus traversal order would be New button, File button, Quit button, Text checkbox, Binary checkbox. However, we can change this be creating a FocusPolicy class derived from the default ContainerOrderFocusPolicy class.

We simply need to create a class that keeps a list of the controls in the order we want them to be traversed and provide getComponentBefore and getComponentAfter methods, as well as getFirstComponent and getLastComponent methods. Here is how we create and load that class:

```
            //create a focus policy class
            FocusPolicy fpolicy = new FocusPolicy();
            //and add the components in the order
            //you want them to be traversed
            //here: left to right
            fpolicy.addComponent( btNew);
            fpolicy.addComponent( ckText);
            fpolicy.addComponent( btFile);
            fpolicy.addComponent( ckBinary);
            fpolicy.addComponent( btQuit);
            //set this as the frame's current policy
            this.setFocusTraversalPolicy( fpolicy);
```

The complete FocusPolicy class is shown in Listing 1.

```
public class FocusPolicy extends ContainerOrderFocusTraversalPolicy {
      private Vector components;
      private int index;
      //-----
      public FocusPolicy() {
            super();
            components = new Vector();
            index = 0;
      }
      //-----
      public void addComponent(Component c) {
            components.addElement(c);
      }
      //-----
      public Component getFirstComponent() {
            index = 0;
            return (Component) components.elementAt(index);
      }
```

```
	//-----
	public Component getComponentAfter(Container c, Component comp) {
		Iterator iter = components.iterator();
		boolean found = false;
		while (!found && iter.hasNext()) {
			Component test = (Component) iter.next();
			found = (test == comp);
		}
		if (iter.hasNext())
			return (Component) iter.next();
		else
			return getFirstComponent();
	}
	//-----
	public Component getLastComponent(Container c) {
		int i = components.size() - 1;
		if (i >= 0) {
			return (Component) components.elementAt(i);
		} else
			return null;
	}
	//-----
	public Component getComponentBefore(Container c, Component comp) {
		int i = 0;
		boolean found = false;
		while (!found && i < components.size()) {
			Component test = (Component) components.elementAt(i);
			found = (test == comp);
		}
		if (i > 0)
			return (Component) components.elementAt(i - 1);
		else
			return getLastComponent(null);
	}
}
```

Listing 1 – Our new FocusPolicy class.

If you look at Figure 3, you will see that no control has the focus. If you press the tab key the New button will get the focus, followed by the Text checkbox and then the File button, and so forth, in the order we specified in the code just above.

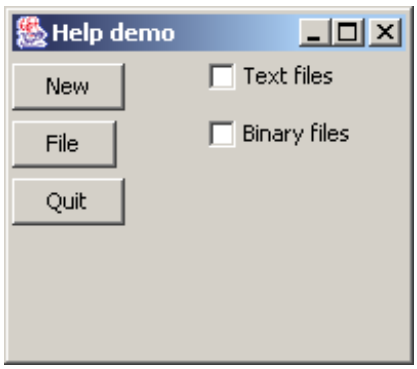

Figure 3- The same chain of responsibility using the FocusPolicy class.

Getting Things Back into Focus

Now that we have focused ourselves on this problem, we see how we can pull focus, change focus, and keep ourselves from being chained to old programming styles. But it's your responsibility to keep yourself focused on your work!