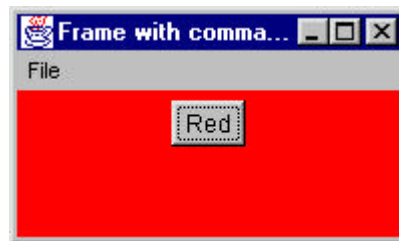

THE COMMAND PATTERN

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

Motivation

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



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

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

```

        redClicked();           //turn red
    }
    The three private methods this method calls are just
private void exitClicked()    {
    System.exit(0);
}
//-----
private void fileOpen()      {
    FileDialog fDlg = new FileDialog(this, "Open a file",
                                     FileDialog.LOAD);
    fDlg.show();
}
//-----
private void redClicked()    {
    p.setBackground(Color.red);
}

```

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

The Command Pattern

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

```

public interface Command {
    public void Execute();
}

```

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

```

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

```

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

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

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

Building Command Objects

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

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

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

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

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

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

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

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

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

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

The Command Pattern in Java

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

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

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

```

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

```

and register them as listeners in the usual way.

```

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

```

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

Consequences of the Command Pattern

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

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

Anonymous Inner Classes

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

```

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

```

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

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

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

Providing Undo

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