Functors and the Command Pattern

Functors

- Often when designing general-purpose software, we make use of callback functions
- A *callback function* is a function that is made known (registered) to the system to be called at a later time when certain events occur
- In C and C++ we can use pointers to functions as a callback mechanism, but this is not available in Java
- In Java we must use an object that serves the role of a pointer to a function. (We could also use this technique in C++.)
- A *functor* is a class with usually only one method whose instances serve the role of a pointer to a function. Functor objects can be created, passed as parameters and manipulated wherever function pointers are needed.
- Coplien coined the word *functor* for this type of class

Functor Example 1

- Consider a Utilities class with a class method that compares two Numbers. We would like to be able to specify the method of comparison at run-time, so we add a Comparator argument to the class method as follows:

```
public class Utilities {
  public static int compareNumbers(Number a, Number b,
                                   Comparator c) {
    return c.compare(a, b);
  }
}
```


Functor Example 1 (Continued)

- The Comparator object is a functor, since it acts like a pointer to the compare() function. To support different types of comparators, we'll use interface inheritance via a Java interface.
- The Comparator interface is:

```
public interface Comparator {
  public int compare(Number a, Number b);
}
```

- Many implementations of functors in Java involve the use of Java interfaces in this fashion

Functor Example 1 (Continued)

- Here is an Integer comparator:

```
//  Integer comparator
public class IntComparator implements Comparator {
  public int compare(Number a, Number b) {
    int x = a.intValue();
    int y = b.intValue();
    if (x < y)
      return -1;
    else if (x > y)
      return 1;
    else
      return 0;
  }
}
```


Functor Example 1 (Continued)

- And here is a String comparator:

```
// String comparator
public class StringComparator implements Comparator {
  public int compare(Number a, Number b) {
    String x = a.toString();
    String y = b.toString();
    if (x.compareTo(y) < 0)
      return -1;
    else if (x.compareTo(y) > 0)
      return 1;
    else
      return 0;
  }
}
```

Functor Example 1 (Continued)

- And here is a test program that demonstrates the use of the different comparators:

```
public class TestUtilities {
  public static void main(String args[]) {
    // Create an integer Comparator.
    Comparator c1 = new IntComparator();

    // Compare two objects.
    int result = Utilities.compareNumbers(new Float(5.5),
                                          new Double(12.0), c1);
    System.out.println("\nResult is: " + result);

    // Create a string Comparator.
    Comparator c2 = new StringComparator();
```


Functor Example 1 (Continued)

```
    // Compare the same two objects.
    result = Utilities.compareNumbers(new Float(5.5),
                                      new Double(12.0), c2);
    System.out.println("\nResult is: " + result);
  }
}
```

- Test program output:

```
Result is: -1
Result is: 1
```

Functor Example 2

- Consider a Java Observable object and its Observer objects. Each Observer implements the Observer interface and provides an implementation of the update() method. As far as the Observable is concerned, it essentially has a pointer to an update() method to callback when Observers should be notified. So, in this case, each Observer object is acting as a functor.
- Here's an Observer:

```
public class NameObserver implements Observer {
  public void update(Observable obj, Object arg) {
   //Whatever
  }
  …
}
```


Functor Example 2 (Continued)

- And here's an Observable. The notifyObservers() method makes the callback to update().

```
public class ConcreteSubject extends Observable {
  public void setName(String name) {
    this.name = name;
    setChanged();
    notifyObservers(name);
  }
  …
}
```

Functor Example 3

- Callbacks are used often in GUIs
- For example, when an AWT button is pressed and released, it generates an action event which is sent to all registered listeners. Listeners must implement the ActionListener interface and implement the actionPerformed() method. The button invokes (calls back) the actionPerformed() method of the listener object.
- This is really the Observer pattern!
- Simple GUI example:

```
public class MyApp extends Frame implements ActionListener {

  // GUI attributes.
  private Button goButton = new Button("Go");
  private Button exitButton = new Button("Exit");
```


Functor Example 3 (Continued)

```
  // MyApp Constructor
  public MyApp() {
    super("My Application");
    setupWindow();
  }

  // Setup GUI.
  private void setupWindow() {
    Panel bottomPanel = new Panel();
    bottomPanel.add(goButton);
    bottomPanel.add(exitButton);
    // Register myself as an action listener for these buttons!
    goButton.addActionListener(this);
    exitButton.addActionListener(this);
    pack();
  }
```

Functor Example 3 (Continued)

```
  // Handle GUI actions.
  // This is the callback function!
  public void actionPerformed(ActionEvent event) {
    Object src = event.getSource();
    if (src == goButton)
      go();
    else if (src == exitButton)
      System.exit(0);
  }

  // Main method.
  public static void main(String[] argv) {
    MyApp app = new MyApp();
    app.setVisible(true);
  }
}
```


Functor Example 3 (Continued)

- There are a couple of uncomfortable things about the last example:
  - ⇒ The object that handles the callback does a lot more than just handle the callback! The MyApp class may have many other methods, doing many other things, besides providing the actionPerformed() method as the callback function for the buttons. We have a sense that we should be designing classes that provide a limited, focused functionality. As far as the button is concerned, it just sees the actionPerformed() method of the callback object, but we have a feeling that we have not encapsulated functionality properly here. Perhaps, we should have a separate class just to handle the callback!

Functor Example 3 (Continued)

- ⇨ Also, the actionPerformed() method itself is bothersome. MyApp could be registered as a listener for many buttons (in this case, it is registered with just two buttons), and the actionPerformed() method must handle action events from all of these buttons. So we have a potentially large conditional in actionPerformed(). Again, it seems better to have a different callback object for each button.
- ⇨ But separate objects probably will need access to the methods and attributes of the MyApp class

- Java 1.1 has a neat answer to our concerns: inner classes!

Functor Example 3 (Continued)

- Here's the setupWindow() method in the MyApp class using anonymous inner classes in Java 1.1:

```
// Setup GUI.
private void setupWindow() {
  Panel bottomPanel = new Panel();
  bottomPanel.add(goButton);
  bottomPanel.add(exitButton);

  // Use instances of anonymous inner classes
  // as the callback objects for the button events!
  goButton.addActionListener(new ActionListener () {
    public void actionPerformed(ActionEvent event) {
      go();
    }
  });
```

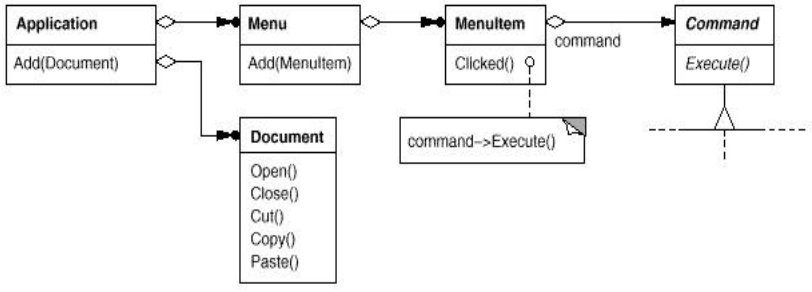
Functor Example 3 (Continued)

```
    exitButton.addActionListener(new ActionListener () {
      public void actionPerformed(ActionEvent event) {
        System.exit(0);
      }
    });
    pack();
  }
```

- These instances of an anonymous inner class are functors. They are objects acting as pointers to functions.

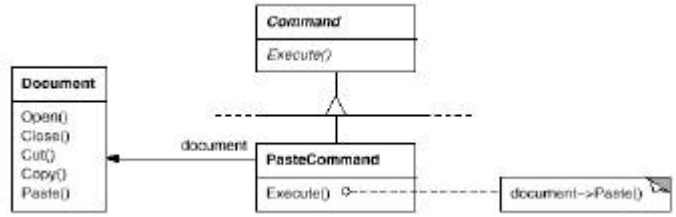
The Command Pattern

- Intent
  - ⇨ Encapsulate requests for service from an object inside other objects, thereby letting you manipulate the requests in various ways
- Motivation



The Command Pattern

- Motivation



- Applicability

  Use the Command pattern when

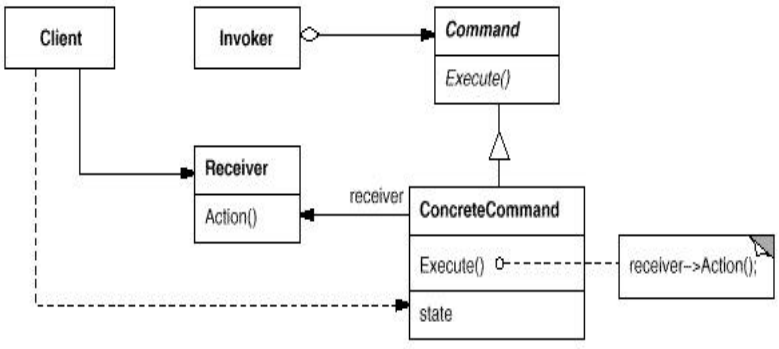
  ⇒ You want to implement a callback function capability

  ⇒ You want to specify, queue, and execute requests at different times

  ⇒ You need to support undo and change log operations

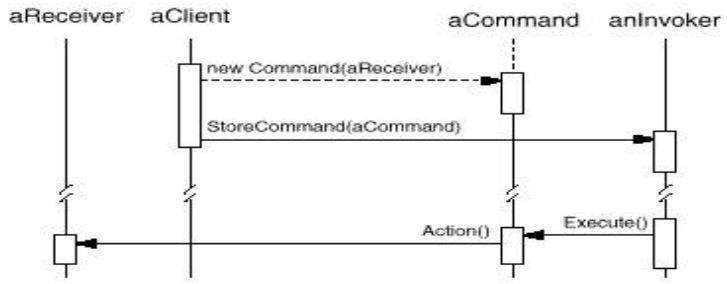
The Command Pattern

- Structure



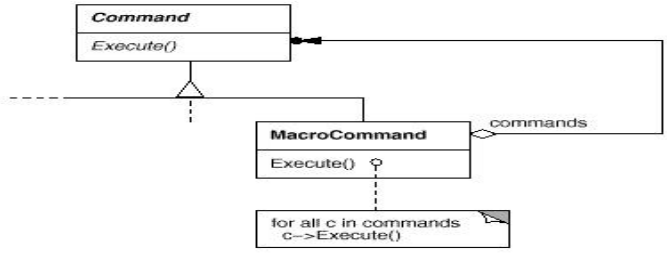
The Command Pattern

- Collaborations



The Command Pattern

- Consequences
  - ⇨ Command decouples the object that invokes the operation from the one that knows how to perform it
  - ⇨ Commands are first-class objects. They can be manipulated and extended like any other object.
  - ⇨ Commands can be made into a composite command



The Command Pattern

- Implementation Issues
  - How intelligent should a command object be?
    - Dumb: Delegates the required action to a receiver object
    - Smart: Implements everything itself without delegating to a receiver object at all

The Command Pattern And Functors

- A functor object usually implements the desired behavior itself without delegation to another object. A Command object frequently delegates the desired behavior to another receiver object.
- Functor:

```
Class X          Class Y
y.foo() ------>  foo()
```

- Command Pattern:

```
Class X           ConcreteCommand C        Class Y
c.execute() --->  execute ()
                                          y.foo() --->  foo()
```

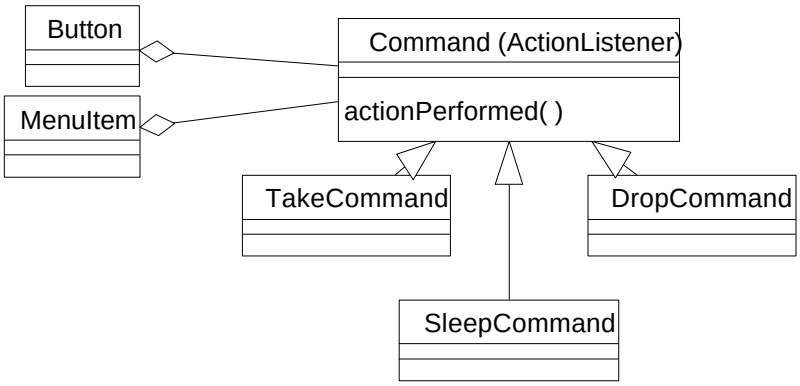
The Command Pattern And Functors

- If the ConcreteCommand provides the behavior itself, then it is acting like a simple functor
- Admittedly, this distinction is somewhat weak. It could be that the functor method foo() in Class Y delegates the desired behavior to another object already!

Command Pattern Example 1

- Situation: A GUI system has several buttons that perform various actions. We want to have menu items that perform the same action as its corresponding button.
- Solution 1: Have one action listener for all buttons and menu items. As seen earlier, this is not a good solution as the resulting actionPerformed() method violates the Open-Closed Principle.
- Solution 2: Have an action listener for each paired button and menu item. Keep the required actions in the actionPerformed() method of this one action listener. This solution is essentially the Command pattern with simple ConcreteCommand classes that perform the actions themselves, acting like functors
- In Java 2, Swing Action objects are used for this purpose

Command Pattern Example 1 (Continued)



Swing Actions

- A Swing Action object is really just an ActionListener object that not only provides the ActionEvent handling, but also centralized handling of the text, icon, and enabled state of toolbar buttons or menu items
- The same Action object can easily be associated with a toolbar button and a menu item
- The Action object also maintains the enabled state of the function associated with the toolbar button or menu item and allows listeners to be notified when this functionality is enabled or disabled

Swing Actions

- In addition to the actionPerformed method defined by the ActionListener interface, objects that implement the Action interface provide methods that allow the specification of:
  - ⇒ One or more text strings that describe the function of the button or menu item
  - ⇒ One or more icons that depict the function
  - ⇒ The enabled/disabled state of the functionality
- By adding an Action to a JToolBar or JMenu, you get:
  - ⇒ A new JButton (for JToolBar) or JMenuItem (for JMenu) that is automatically added to the tool bar or menu. The button or menu item automatically uses the icon and text specified by the Action.
  - ⇒ A registered action listener (the Action object) for the button or menu item
  - ⇒ Centralized handling of the button's or menu item's enabled state

Swing Actions Example

```
/**
 * Class SwingActions demonstrates the Command Pattern
 * using Swing Actions.
 */
public class SwingActions extends JFrame {

  private JToolBar tb;
  private JTextArea ta;
  private JMenu fileMenu;
  private Action openAction;
  private Action closeAction;

  public SwingActions() {
    super("SwingActions");
    setupGUI();
  }
```

Swing Actions Example (Continued)

```
private void setupGUI() {

  //Create the toolbar and menu.
  tb = new JToolBar();
  fileMenu = new JMenu("File");

  //Create the text area used for output.
  ta = new JTextArea(5, 30);
  JScrollPane scrollPane = new JScrollPane(ta);

  //Layout the content pane.
  JPanel contentPane = new JPanel();
  contentPane.setLayout(new BorderLayout());
  contentPane.setPreferredSize(new Dimension(400, 150));
  contentPane.add(tb, BorderLayout.NORTH);
  contentPane.add(scrollPane, BorderLayout.CENTER);
  setContentPane(contentPane);
```


Swing Actions Example (Continued)

```
  //Set up the menu bar.
  JMenuBar mb = new JMenuBar();
  mb.add(fileMenu);
  setJMenuBar(mb);

  // Create an action for "Open".
  ImageIcon openIcon = new ImageIcon("open.gif");
  openAction = new AbstractAction("Open", openIcon) {
    public void actionPerformed(ActionEvent e) {
      ta.append("Open action from " + e.getActionCommand() +"\n");
    }
  };

  // Use the action to add a button to the toolbar.
  JButton openButton = tb.add(openAction);
  openButton.setText("");
  openButton.setActionCommand("Open Button");
  openButton.setToolTipText("This is the open button");
```

Swing Actions Example (Continued)

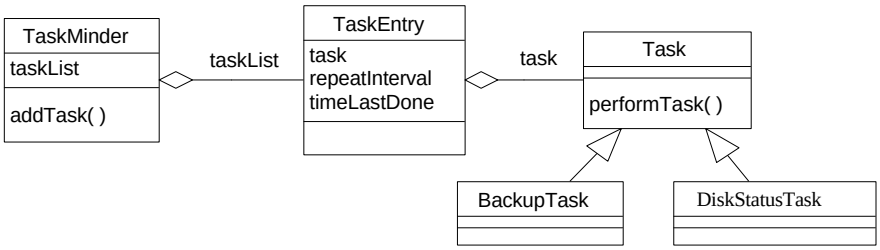
```
    // Use the action to add a menu item to the file menu.
    JMenuItem openMenuItem = fileMenu.add(openAction);
    openMenuItem.setIcon(null);
    openMenuItem.setActionCommand("Open Menu Item");

    // Create an action for "Close" and use the action to add
    // a button to the toolbar and a menu item to the menu.
    // Code NOT shown - similar to "open" code above.
  }

  public static void main(String[] args) {
    SwingActions frame = new SwingActions();
    frame.pack();
    frame.setVisible(true);
  }

}
```


Command Pattern Example 2

- Scenario: We want to write a class that can periodically execute one or more methods of various objects. For example, we want to run a backup operation every hour and a disk status operation every ten minutes. But we do not want the class to know the details of these operations or the objects that provide them. We want to decouple the class that schedules the execution of these methods with the classes that actually provide the behavior we want to execute.

Command Pattern Example 2 (Continued)

- Solution: The Command Pattern!



- Here's the Task interface:

```
public interface Task {
  public void performTask();
}
```


Command Pattern Example 2 (Continued)

- Instead of a BackupTask or DiskStatusClass, here is a simple Fortune Teller Task which cycles through a list of fortune sayings:

```
public class FortuneTask  implements Task {
  int nextFortune = 0;
  String[] fortunes = {"She who studies hard, gets A",
      "Seeth the pattern and knoweth the truth",
      "He who leaves state the day after final, graduates not" };

  public FortuneTask() {}
  public void performTask() {
    System.out.println("Your fortune is: " + fortunes[nextFortune]);
    nextFortune = (nextFortune + 1) % fortunes.length;
  }
  public String toString() {return ("Fortune Telling Task");}
}
```

Command Pattern Example 2 (Continued)

- And here is a Fibonacci Sequence Task which generates a sequence of Fibonacci numbers:

```
public class FibonacciTask implements Task {
  int n1 = 1;
  int n2 = 0;
  int num;

  public FibonacciTask() {}
  public void performTask() {
    num = n1 + n2;
    System.out.println("Next Fibonacci number is: " + num);
    n1 = n2;
    n2 = num;
  }
  public String toString() {return ("Fibonacci Sequence Task");}
}
```


Command Pattern Example 2 (Continued)

- Here is the TaskEntry class:

```
public class TaskEntry {

  private Task task                // The task to be done
                                   // It's a Command object!
  private long repeatInterval;     // How often task should be
                                   // executed
  private long timeLastDone;       // Time task was last done

  public TaskEntry(Task task, long repeatInterval) {
    this.task = task;
    this.repeatInterval = repeatInterval;
    this.timeLastDone = System.currentTimeMillis();
  }

  public Task getTask() {return task;}
```

Command Pattern Example 2 (Continued)

```
  public void setTask(Task task) {this.task = task;}

  public long getRepeatInterval() {return repeatInterval;}

  public void setRepeatInterval(long ri) {this.repeatInterval = ri;}

  public long getTimeLastDone() {return timeLastDone;}

  public void setTimeLastDone(long t) {this.timeLastDone = t;}

  public String toString() {
    return (task + " to be done every " + repeatInterval +
            " ms; last done " + timeLastDone);
  }

}
```


Command Pattern Example 2 (Continued)

- Here is the TaskMinder class:

```
public class TaskMinder extends Thread {
  private long sleepInterval;  // How often the TaskMinder should
                               // check for tasks to be run
  private Vector taskList;     // The list of tasks

  public TaskMinder(long sleepInterval, int maxTasks) {
    this.sleepInterval = sleepInterval;
    taskList = new Vector(maxTasks);
    start();
  }

  public void addTask(Task task, long repeatInterval) {
    long ri = (repeatInterval > 0) ? repeatInterval : 0;
    TaskEntry te = new TaskEntry(task, ri);
    taskList.addElement(te);
  }
```

Command Pattern Example 2 (Continued)

```
  public Enumeration getTasks() {return taskList.elements();}

  public long getSleepInterval() {return sleepInterval;}

  public void setSleepInterval(long si) {this.sleepInterval = si;}

  public void run() {
    while (true) {
      try {
        sleep(sleepInterval);
        long now = System.currentTimeMillis();
        Enumeration e = taskList.elements();
```


Command Pattern Example 2 (Continued)

```
        while (e.hasMoreElements()) {
            TaskEntry te = (TaskEntry) e.nextElement();
            if (te.getRepeatInterval() + te.getTimeLastDone() < now) {
              te.getTask().performTask();
              te.setTimeLastDone(now);
            }
          }
        }
        catch (Exception e) {
          System.out.println("Interrupted sleep: " + e);
        }
      }
    }

}
```

Command Pattern Example 2 (Continued)

- Test program:

```
public class TestTaskMinder {

  public static void main(String args[]) {

    // Create and start a TaskMinder.
    // This TaskMinder checks for things to do every 500 ms
    //   and allows a maximum of 100 tasks.
    TaskMinder tm = new TaskMinder(500, 100);

    // Create a Fortune Teller Task.
    FortuneTask fortuneTask = new FortuneTask();

    // Have the Fortune Teller execute every 3 seconds.
    tm.addTask(fortuneTask, 3000);
```


Command Pattern Example 2 (Continued)

```
    // Create a Fibonacci Sequence Task.
    FibonacciTask fibonacciTask = new FibonacciTask();

    // Have the Fibonacci Sequence execute every 700 milliseconds.
    tm.addTask(fibonacciTask, 700);

  }

}
```

Command Pattern Example 2 (Continued)

- Test program output:

```
Next Fibonacci number is: 1
Next Fibonacci number is: 1
Your fortune is: She who studies hard, gets A
Next Fibonacci number is: 2
Next Fibonacci number is: 3
Next Fibonacci number is: 5
Next Fibonacci number is: 8
Your fortune is: Seeth the pattern and knoweth the truth
Next Fibonacci number is: 13
Next Fibonacci number is: 21
Next Fibonacci number is: 34
Your fortune is: He who leaves state the day after final, graduates
   not
Next Fibonacci number is: 55
Next Fibonacci number is: 89
Next Fibonacci number is: 144
etc.
```