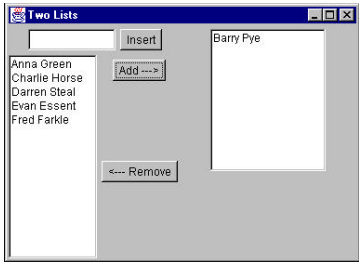
Design Patterns Taxonomy

- Creational patterns
  - Concern the process of object creation
- Structural patterns
  - Deal with the composition of classes or objects
- Behavioral patterns
  - Characterize the ways in which classes or objects interact and distribute responsibility

Structural Patterns

An Example Application

- Lets create a simple Java program that
  - allows you to enter names into a list,
  - and then select some of those names to be transferred to another list.
  - Our initial list consists of a class roster and the second list, those who will be doing advanced work.
- In the example UI,
  - You enter names into the top entry field and click on **Insert** to move the names into the left-hand list box.
  - To move names to the right-hand list box, you click on them, and then click on **Add**.
  - To remove a name from the right hand list box, click on it and then on **Remove**. This moves the name back to the left-hand list.



Lets Look at some Code!

- The code will consist of a GUI creation constructor and an **actionListener** routine for the three buttons:

```
public void actionPerformed(ActionEvent e)
   {
      Button b = (Button)e.getSource();
      if(b == Add)
         addName();
      if(b == MoveRight)
         moveNameRight();
      if(b == MoveLeft)
         moveNameLeft();
   }
```

ActionListener

- public interface ActionListener extends EventListener

- The listener interface for receiving action events.
  - The class that is interested in processing an action event implements this interface, and the object created with that class is registered with a component, using the component's **addActionListener** method. When the action event occurs, that object's **actionPerformed** method is invoked.

ActionEvent

- public class ActionEvent extends AWTEvent

- A semantic event which indicates that a component-defined action occurred.
- This high-level event is generated by a component (such as a Button) when the component-specific action occurs (such as being pressed).
- The event is passed to every ActionListener object that registered to receive such events using the component's addActionListener method.
- The object that implements the ActionListener interface gets this ActionEvent when the event occurs.
- The listener is therefore spared the details of processing individual mouse movements and mouse clicks, and can instead process a "meaningful" (semantic) event like "button pressed".

java.lang.Object
  java.util.EventObject
    java.awt.AWTEvent
      java.awt.event.ActionEvent

getSource()

- Method inherited from class java.util.EventObject

- public Object getSource()
  - Returns: The object on which the Event initially occurred.

addName()

```
private void addName()
{
   if (txt.getText().length() > 0)
      {
      leftList.add(txt.getText());
      txt.setText("");
      }
}
```

- add() is a method of AWT **List** class.

moveNameRight() and moveNameLeft()

```
private void moveNameRight()
{
  String sel[] = leftList.getSelectedItems();
  if (sel != null)
  {
  rightList.add(sel[0]);
  leftList.remove(sel[0]);
  }
}

public void moveNameLeft()
{
 String sel[] = rightList.getSelectedItems();
  if (sel != null)
  {
  leftList.add(sel[0]);
  rightList.remove(sel[0]);
  }
}
```


AWT List class

awt List class
add(String);
remove(String)
String[] getSelectedItems()

Code Maintenance Task

- Suppose you would like to rewrite the program using the Java Foundation Classes (JFC or "Swing").
- Most of the methods you use for creating and manipulating the user interface remain the same.
- However, the JFC JList class is markedly different than the AWT List class.
- In fact, because the JList class was designed to represent far more complex kinds of lists, there are virtually no methods in common between the classes.
- Both classes (AWT List class, JFC JList class) have quite a number of other methods and almost none of them are closely correlated.

A Possible Solution

- However, since we have already written the program once, and make use of two different list boxes, writing "some auxiliary code" to make the JList class *look like* the List class seems a sensible solution to our problem.

More on JList…

- The JList class is a window container which has an array, vector or other ListModel class associated with it.
- It is this ListModel that actually contains and manipulates the data.
- The JList class does not contain a scroll bar, but instead relies on being inserted in the viewport of the JScrollPane class.
- Data in the JList class and its associated ListModel are not limited to strings, but may be almost any kind of objects, as long as you provide the cell drawing routine for them.
- This makes it possible to have list boxes with pictures illustrating each choice in the list.

Our Task is Much Simpler!

- We are only going to create a class that emulates the List class, and that in this simple case, needs only the three methods:
  - add(String);
  - remove(String);
  - String[] getSelectedItems()
- We define the needed methods as an interface and then make sure that the class we create implements those methods:

```
public interface awtList  {
     public void add(String s);
     public void remove(String s);
     public String[] getSelectedItems()
}
```


JScrollPane

- We create a class that contains a JList class but which implements the methods of the awtList interface.
- This is a pretty good choice here, because the outer container for a JList is not the list element at all, but the JScrollPane that encloses it.


```
public class JawtList extends JScrollPane
   implements awtList
{
   private JList listWindow;
   private JListData listContents;
//----------------------------------------
   public JawtList(int rows)    {
      listContents = new JListData();
      listWindow = new JList(listContents);
       getViewport().add(listWindow);

   }
```

```
//-----------------------------------------
   public void add(String s)    {
      listContents.addElement(s);
   }
//-----------------------------------------
   public void remove(String s)    {
      listContents.removeElement(s);
   }
//-----------------------------------------
   public String[] getSelectedItems()    {
      Object[] obj = listWindow.getSelectedValues();
      String[] s = new String[obj.length];
      for (int i =0; i<obj.length; i++)
            s[i] = obj[i].toString();
      return s;
   }
}
```


JListData

- Note that the actual data handling takes place in the JListData class. This class is derived from the AbstractListModel, which defines the following methods:

addListDataListener(l)	Add a listener for changes in the data.
removeListDataListener(l)	Remove a listener
fireContentsChanged(obj, min,max)	Call this after any change occurs between the two indexes min and max
fireIntervalAdded(obj,min,max)	Call this after any data has been added between min and max.
fireIntervalRemoved(obj, min, max)	Call this after any data has been removed between min and max.

Fire?

- The three fire methods are the communication path between the data stored in the ListModel and the actual displayed list data.
- Firing them causes the displayed list to be updated.
- Hence, Each time we add data to the *data* vector, we call the *fireIntervalAdded* method to tell the list display to refresh that area of the displayed list.

Final Touches

```
class JListData extends AbstractListModel
{
   private Vector data;
//-----------------------------------------
   public JListData()    {
      data = new Vector();
   }
//-----------------------------------------
   public void addElement(String s)
   {
      data.addElement(s);
      fireIntervalAdded(this, data.size()-1,
                        data.size());
   }
//-----------------------------------------
   public void removeElement(String s)    {
      data.removeElement(s);
      fireIntervalRemoved(this, 0, data.size());
   }
}
```

CMSC 433 – Programming Language Technologies and Paradigms Spring 2007

Adapter Pattern

Mar. 27, 2007

What is it?

- The Adapter pattern is used to convert the programming interface of one class into that of another.
- We use adapters whenever we want unrelated classes to work together in a single program.
- The concept of an adapter is thus pretty simple;
  - we write a class that has the desired interface and then make it communicate with the class that has a different interface.

Two-way Adapter

- The two-way adapter is a clever concept that allows an object to be viewed by different classes as being either of type X or a type Y.

Pluggable Adapter

- A pluggable adapter is one that adapts dynamically to one of several classes.
- Of course, the adapter can only adapt to classes it can recognize, and usually the adapter decides which class it is adapting.

Reflection

- Java has yet another way for adapters to recognize which of several classes it must adapt to: reflection.
- You can use reflection to discover the names of public methods and their parameters for any class.
- For example, for any arbitrary object you can use the *getClass()* method to obtain its class and the *getMethods()* method to obtain an array of the method names.

```
JList list = new JList();
      Method[] methods = list.getClass().getMethods();
       //print out methods
      for (int i = 0; i < methods.length; i++)        {
         System.out.println(methods[i].getName());
          //print out parameter types
         Class cl[] = methods[i].getParameterTypes();
         for(int j=0; j < cl.length; j++)
            System.out.println(cl[j].toString());      ;
      }
```

Fine Print

- A “method dump” like the one produced by the code shown in the previous slide can generate a very large list of methods.
- It is easier if you know the name of the method you are looking for and simply want to find out which arguments that method requires.
- From that method signature, you can then deduce the adapting you need to carry out.