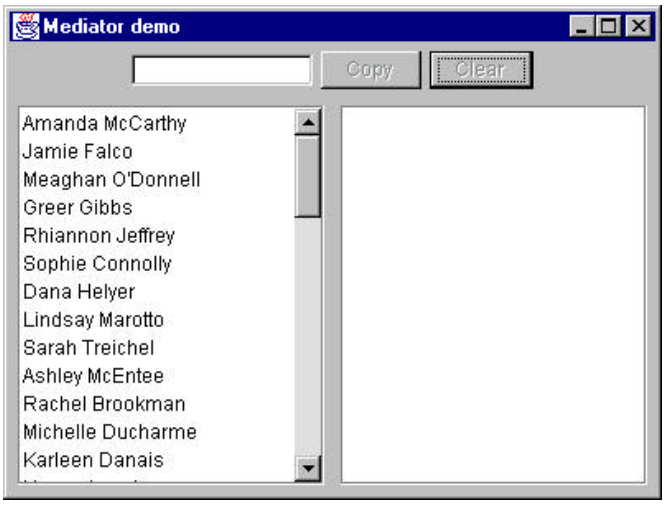
THE MEDIATOR PATTERN

When a program is made up of a number of classes, the logic and computation is divided logically among these classes. However, as more of these isolated classes are developed in a program, the problem of communication between these classes become more complex. The more each class needs to know about the methods of another class, the more tangled the class structure can become. This makes the program harder to read and harder to maintain. Further, it can become difficult to change the program, since any change may affect code in several other classes. The Mediator pattern addresses this problem by promoting looser coupling between these classes. Mediators accomplish this by being the only class that has detailed knowledge of the methods of other classes. Classes send inform the mediator when changes occur and the Mediator passes them on to any other classes that need to be informed.

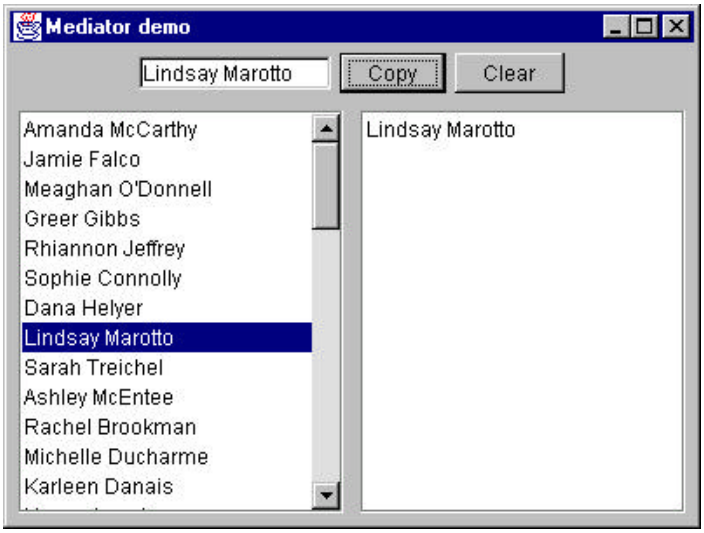
An Example System

Let's consider a program which has several buttons, two list boxes and a text entry field:



When the program starts, the Copy and Clear buttons are disabled.

1. When you select one of the names in the left-hand list box, it is copied into the text field for editing, and the *Copy* button is enabled.
2. When you click on *Copy*, that text is added to the right hand list box, and the *Clear* button is enabled.

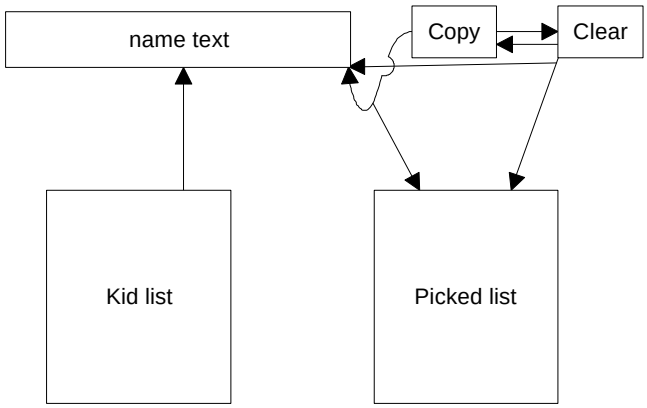


3. If you click on the *Clear* button, the right hand list box and the text field are cleared, the list box is deselected and the two buttons are again disabled.

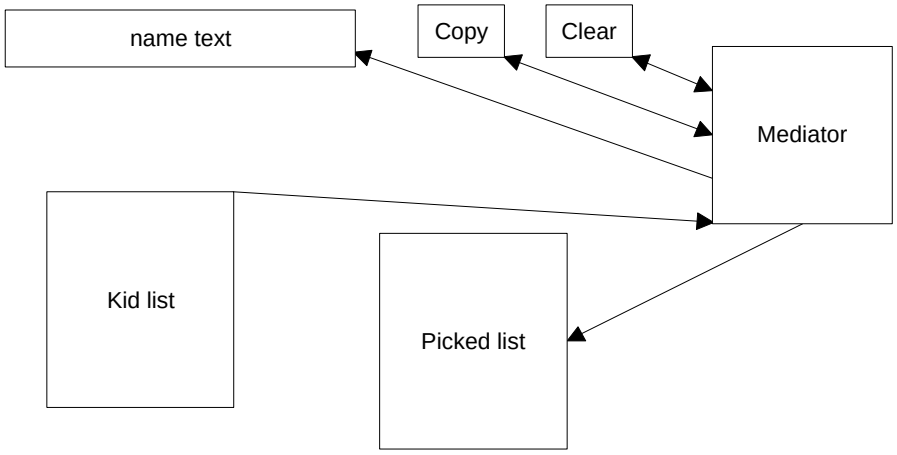
User interfaces such as this one are commonly used to select lists of people or products from longer lists. Further, they are usually even more complicated than this one, involving insert, delete and undo operations as well.

Interactions between Controls

The interactions between the visual controls are pretty complex, even in this simple example. Each visual object needs to know about two or more others, leading to quite a tangled relationship diagram as shown below.



The Mediator pattern simplifies this system by being the only class that is aware of the other classes in the system. Each of the controls that the Mediator communicates with is called a Colleague. Each Colleague informs the Mediator when it has received a user event, and the Mediator decides which other classes should be informed of this event. This simpler interaction scheme is illustrated below:



The advantage of the Mediator is clear-- it is the only class that knows of the other classes, and thus the only one that would need to be changed if one of the other classes changes or if other interface control classes are added.

Sample Code

Let's consider this program in detail and decide how each control is constructed. The main difference in writing a program using a Mediator class is that each class needs to be aware of the existence of the Mediator. You start by creating an instance of the Mediator and then pass the instance of the Mediator to each class in its constructor.

```
Mediator med = new Mediator();
kidList = new KidList( med);
tx = new KTextField(med);
 Move = new MoveButton(this, med);
Clear = new ClearButton(this, med);
med.init();
```

Since, we have created new classes for each control, each derived from base classes, we can handle the mediator operations within each class.

Our two buttons use the Command pattern and register themselves with the Mediator during their initialization. Here is the Copy button:

```
public class CopyButton extends JButton
      implements Command
{
  Mediator med;            //copy of the Mediator
public CopyButton(ActionListener fr, Mediator md)
 {
    super("Copy");         //create the button
    addActionListener(fr); //add its listener
    med = md;              //copy in Mediator instance
    med.registerMove(this); //register with the Mediator
 }
 public void Execute()
 {                         //execute the copy
    med.Copy();
 }
}
```

The Clear button is exactly analogous.

The Kid name list is based on the one we used in the last two examples, but expanded so that the data loading of the list and registering the list with the Mediator both take place in the constructor. In addition, we make the enclosing class the ListSelectionListener and pass the click on any list item on to the Mediator directly from this class.

```
public class KidList extends JawtList
   implements ListSelectionListener
{
```

```
   KidData kdata;          //reads the data from the file
   Mediator med;           //copy of the mediator

  public KidList(Mediator md)
  {
   super(20);       //create the JList
   kdata = new KidData ("50free.txt");
   fillKidList();          //fill the list with names
   med = md;        //save the mediator
   med.registerKidList(this);
   addListSelectionListener(this);
  }
 //---------------------------------
  public void valueChanged(ListSelectionEvent ls)
  {
       //if an item was selected pass on to mediator
     JList obj = (JList)ls.getSource();
     if (obj.getSelectedIndex() >= 0)
         med.select();
  }
  //---------------------------------
  private void fillKidList()
   {
      Enumeration ekid = kdata.elements();
      while (ekid.hasMoreElements())  {
         Kid k =(Kid)ekid.nextElement();
         add(k.getFrname()+" "+k.getLname());
         }
   }
}
```

The text field is even simpler, since all it does is register itself with the mediator.

```
public class KTextField extends JTextField
{
 Mediator med;
 public KTextField(Mediator md)  {
    super(10);
    med = md;
    med.registerText(this);
 }
}
```

The general point of all these classes is that each knows about the Mediator and tells the Mediator of its existence so the Mediator can send commands to it when appropriate.

The Mediator itself is very simple. It supports the Copy, Clear and Select methods, and has register methods for each of the controls:

```
public class Mediator
```

```
{
 private ClearButton clearButton;
 private CopyButton  copyButton;
 private KTextField  ktext;
 private KidList     klist;
 private PickedKidsList picked;

public void Copy()  {
    picked.add(ktext.getText()); //copy text
    clearButton.setEnabled(true);//enable Clear
 }
//-------------------------------------
 public void Clear()  {
    ktext.setText("");                //clear text
    picked.clear();              //and list
//disable buttons
    copyButton.setEnabled(false);
    clearButton.setEnabled(false);
    klist.clearSelection();             //deselect list
}
 //-------------------------------------
 public void Select()  {
    String s = (String)klist.getSelectedValue();
    ktext.setText(s);                   //copy text
    copyButton.setEnabled(true); //enable Copy
}
 //-----------copy in controls-------------------------
 public void registerClear(ClearButton cb)  {
    clearButton = cb;  }
 public void registerCopy(CopyButton mv)  {
    copyButton = mv;  }
 public void registerText(KTextField tx)  {
    ktext = tx;  }
 public void registerPicked(PickedKidsList pl)  {
    picked = pl;  }
 public void registerKidList(KidList kl)  {
    klist = kl;  }
}
```

Initialization of the System

One further operation that is best delegated to the Mediator is the initialization of all the controls to the desired state. When we launch the program, each control must be in a known, default state, and since these states may change as the program evolves, we simply create an *init* method in the Mediator, which sets them all to the desired state. In this case, that state is the same as is achieved by the Clear button and we simply call that method:

```
public void init()  {
    Clear();
```

```
}
```

Mediators and Command Objects

The two buttons in this program are command objects, and we register the main user interface frame as the *ActionListener* when we initialize these buttons. Just as we noted earlier, this makes processing of the button click events quite simple:

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

Alternatively, we could register each derived class as its own listener and pass the result directly to the Mediator.

In either case, however, this represents the solution to one of the problems we noted in the Command pattern chapter; each button needed knowledge of many of the other user interface classes in order to execute its command. Here, we delegate that knowledge to the Mediator, so that the Command buttons do not need any knowledge of the methods of the other visual objects.

Consequences of the Mediator Pattern

1. The Mediator makes loose coupling possible between objects in a program. It also localizes the behavior that otherwise would be distributed among several objects.
2. You can change the behavior of the program by simply changing or subclassing the Mediator.
3. The Mediator approach makes it possible to add new Colleagues to a system without having to change any other part of the program.
4. The Mediator solves the problem of each Command object needing to know too much about the objects and methods in the rest of a user interface.
5. The Mediator can become monolithic in complexity, making it hard to change and maintain. Sometimes you can improve this situation by revising the responsibilities you have given the Mediator. Each object should carry out it’s own tasks and the Mediator should only manage the interaction between objects.

6. Each Mediator is a custom-written class that has methods for each Colleague to call and knows what methods each Colleague has available. This makes it difficult to reuse Mediator code in different projects. On the other hand, most Mediators are quite simple and writing this code is far easier than managing the complex object interactions any other way.

Implementation Issues

The Mediator pattern we have described above acts as a kind of Observer pattern, observing changes in the Colleague elements. Another approach is to have a single interface to your Mediator, and pass that method various constants or objects which tell the Mediator which operations to perform. In the same fashion, you could have a single Colleague interface that each Colleague would implement, and each Colleague would then decide what operation it was to carry out.

Mediators are not limited to use in visual interface programs, however, it is their most common application. You can use them whenever you are faced with the problem of complex intercommunication between a number of objects.