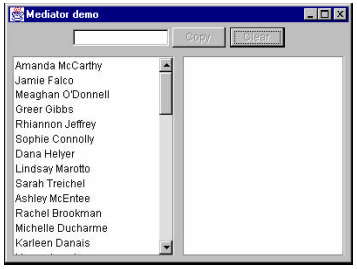
CMSC 433 – Programming Language Technologies and Paradigms Spring 2007

Mediator Pattern

Apr. 12, 2007

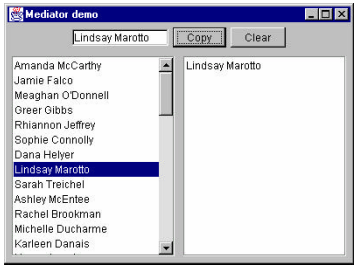
Example

- 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.
- 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.



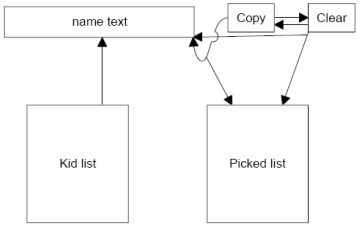
Example (cont…)

- When you click on *Copy,* that text is added to the right hand list box, and the *Clear* button is enabled.
- 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.



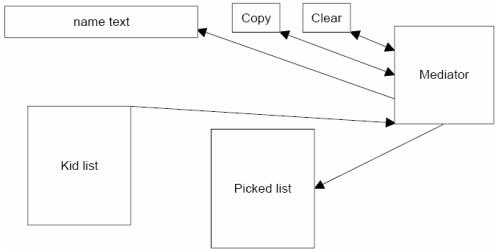
Relationship Diagram

- The interactions between the visual controls are pretty complex.
- Each visual object needs to know about two or more others, leading to quite a tangled relationship diagram.



Mediator

- The Mediator pattern simplifies this system by being the only class that is aware of the other classes in the system.
- The controls that the Mediator communicates with is 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.



Why Mediator?

- 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.

The Code!

- 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();
```


KTextField

- The text field registers itself with the mediator.

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

The Copy Button

- Our two buttons use the Command pattern and register themselves with the Mediator during their initialization.

```
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 Kid name list…

- The data loading and registering of the Kid name 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);
  }
```


The Rest of the Code…

```
//---------------------------------
 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());
        }
  }
}
```


General Mediator Philosophy

- 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 Code!

- 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
 }
```


More Mediator Code!

```
//-------------------------------------
 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
}
```


Yet More Mediator Code!

```
//-----------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;  }
}
```


System Initialization

- 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();
 }
```

Concluding Remarks

- The Mediator makes loose coupling possible between objects in a program.
  - It also localizes the behavior that otherwise would be distributed among several objects.
- You can change the behavior of the program by simply changing the Mediator.
- The Mediator approach makes it possible to add new Colleagues to a system without having to change any other part of the program.
- 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.

More Concluding Remarks

- The Mediator can become complex, 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.
- 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.