Design Patterns III

Ergude Bao

Beijing Jiaotong University

Contents

- Structural patterns I

Structural Patterns I

- Adapter Pattern
- Bridge Pattern
- Composite Pattern
- Decorator Pattern

Adapter Pattern

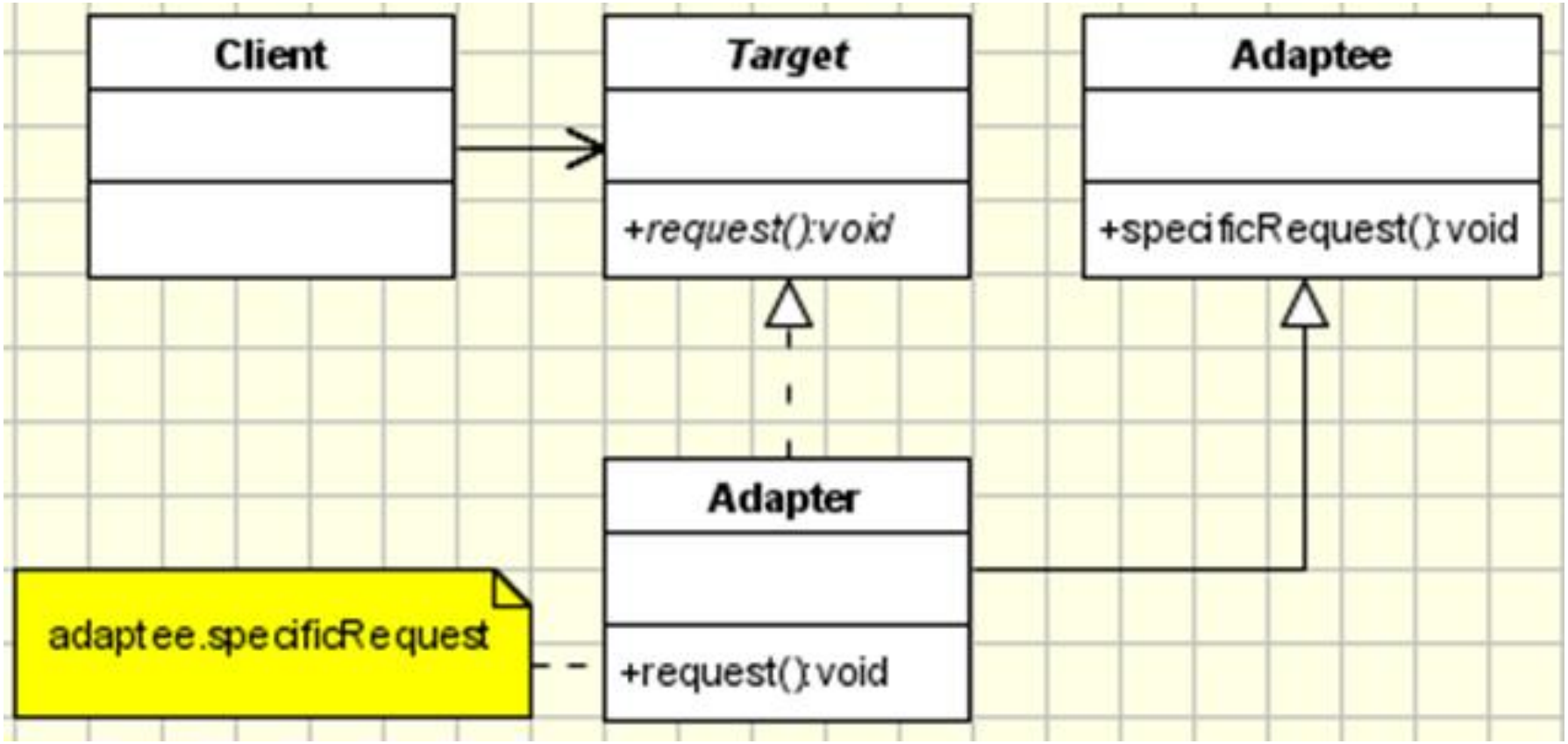
Adapter Pattern

- Motivation
  - We have a class with a desired interface and want to make it communicate with the other class with a different interface
- Solution
  - Convert interface of the other class into the desired interface
- Adapter allows classes with incompatible interfaces work together

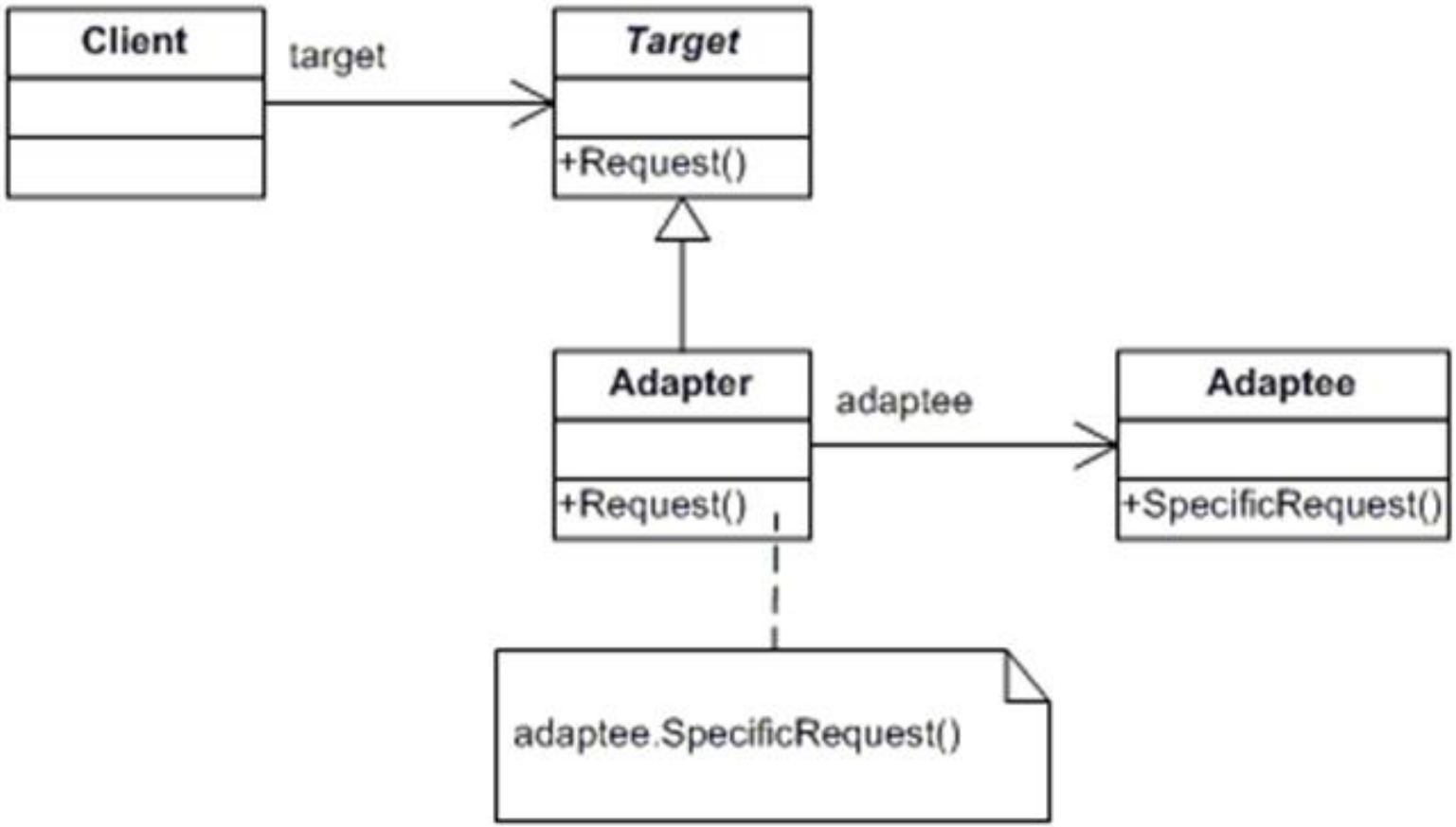
Two Variations

- Class adapter: derive a new class from the nonconforming one with methods to make adaption
- Object adapter: make a new class to include the nonconforming one with methods to make adaption

Class Adapter



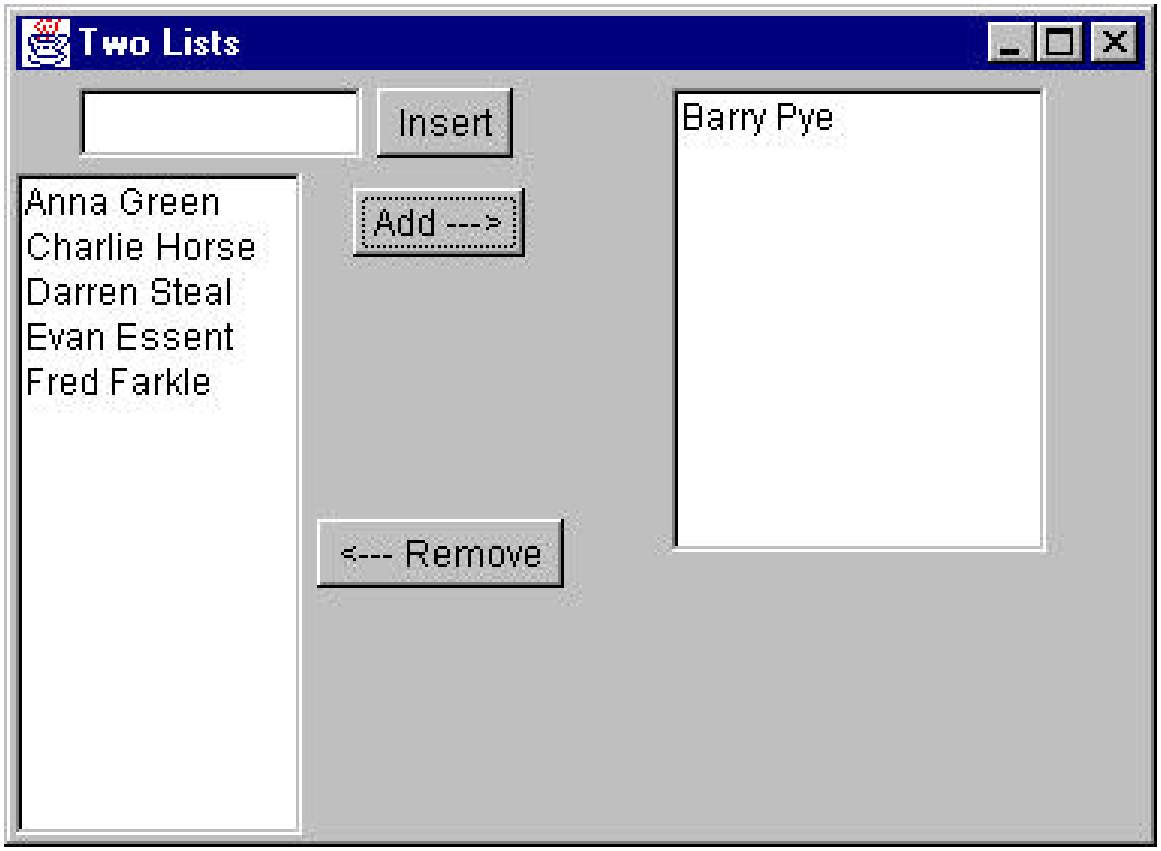
Object Adapter



Participants

- Target
  - Defines the domain-specific interface that Client uses
- Adaptee
  - Implements a different interface that needs adaption
- Adapter
  - Adapts Adaptee to Target
- Client
  - Uses Target and Adaptee

Example


Two Lists
Insert
Barry Pye
Anna Green
Charlie Horse
Darren Steal
Evan Essent
Fred Farkle
Add --->
<--- Remove

Example

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

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

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]);
        }
}
```

Example

- Use the JFC JList class to implement the awt List class
  - Create a class using the JList class
  - Create a class extending the JList class

awt List class	JFC JList class
add(String);	---
remove(String)	---
String[] getSelectedItems()	Object[] getSelectedValues()

Example

- JlistData 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

Example

```
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());
        }
}
```

Example

- Firstly, define the methods needed in an interface

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

Example

- Create a class using the JList class

```
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;
        }
}
```

Example

- Create a class extending the Jlist class

```
public class JclassAwtList extends JList implements awtList {
        private JListData listContents;

        public JclassAwtList(int rows) {
                listContents = new JListData();
                setModel(listContents);
                setPrototypeCellValue("Abcdefg Hijkmnop");
        }
…
leftList = new JclassAwtList(15);
JScrollPane lsp = new JScrollPane();
pLeft.add("Center", lsp);
lsp.getViewport().add(leftList);
…
```

Adapters in Java

- One of the inconveniences of Java is that windows do not close automatically when you click on the Close button or window Exit menu item
- There are a number of adapters built into the Java language
  - WindowAdapter, ComponentAdapter, ContainerAdapter, FocusAdapter, KeyAdapter, MouseAdapter, and MouseMotionAdapter

Adapters in Java

```
public void mainFrame extends Frame implements WindowListener {
        public void mainFrame() {
                addWindowListener(this);
                //frame listens for window events
        }

        public void windowClosing(WindowEvent wEvt) {
                System.exit(0);
                //exit on System exit box clicked
        }
        public void windowClosed(WindowEvent wEvt){}
        public void windowOpened(WindowEvent wEvt){}
        public void windowIconified(WindowEvent wEvt){}
        public void windowDeiconified(WindowEvent wEvt){}
        public void windowActivated(WindowEvent wEvt){}
        public void windowDeactivated(WindowEvent wEvt){}
}
```

Adapters in Java

- Alternatively, WindowAdapter can be used in a simpler manner

```
public class Closer extends Frame {
        public Closer() {
                WindAp windap = new WindAp();
                addWindowListener(windap);
                setSize(new Dimension(100,100));
                setVisible(true);
        }
        static public void main(String argv[]) {
                new Closer();
        }
}

//make an extended window adapter which closes the frame when the closing event is received
class WindAp extends WindowAdapter {
        public void windowClosing(WindowEvent e) {
                System.exit(0);
        }
}
```

Adapters in Java

- Inner class can be used in a shorter manner

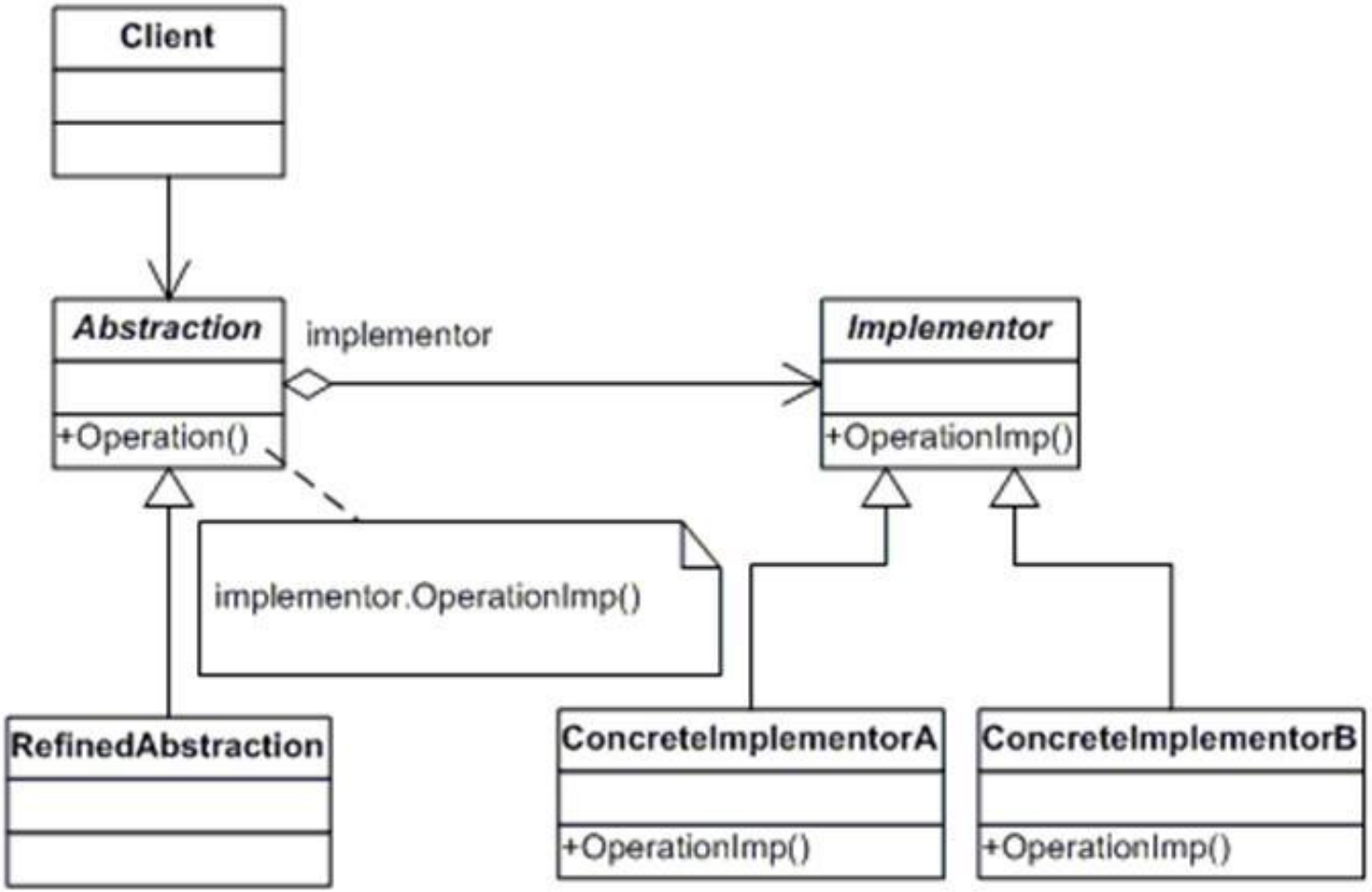
```
//create window listener for window close click
addWindowListener(new WindowAdapter()
{
      public void windowClosing(WindowEvent e) {
            System.exit(0);}
});
```

Bridge Pattern

Bridge Pattern

- Motivation
  - There is a need to avoid permanent binding between an abstraction and an implementation and when the abstraction and implementation need to vary independently
- Solution
  - Decouples an abstraction from its implementation so that the two can vary independently

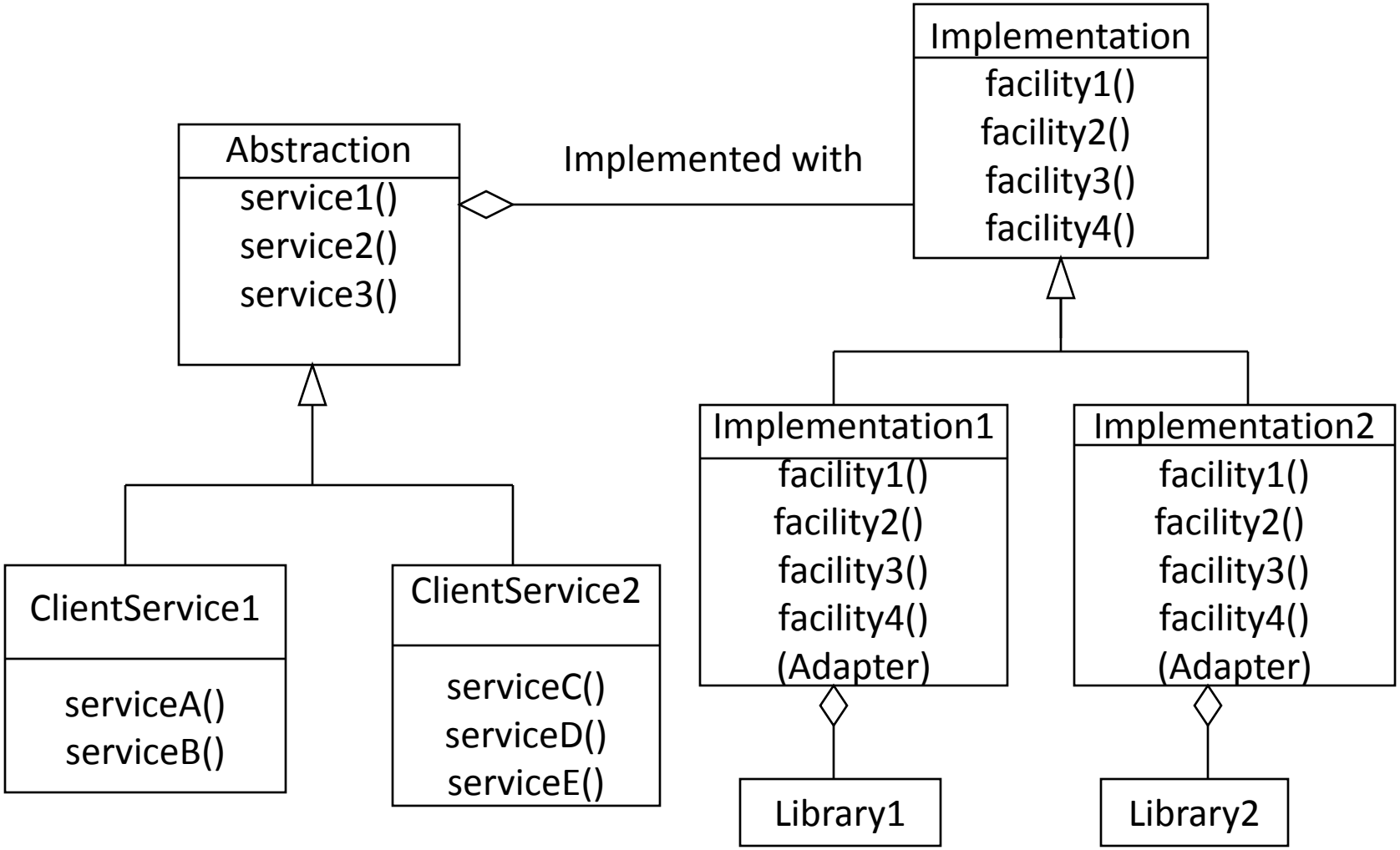
Bridge Pattern



Participants

- Abstraction
  - Defines the abstraction's interface
  - Maintains a reference to an object of type Implementor
- RefinedAbstraction
  - Extends the interface defined by Abstraction
- Implementor
  - Defines the interface for implementation classes
  - This interface does not have to correspond exactly to Abstraction's interface; in fact the two interfaces can be quite different
  - Typically the Implementation interface provides only primitive operations, and Abstraction defines higher-level operations based on these primitives
- ConcreteImplementor
  - Implements the Implementor interface and defines its concrete implementation

Example

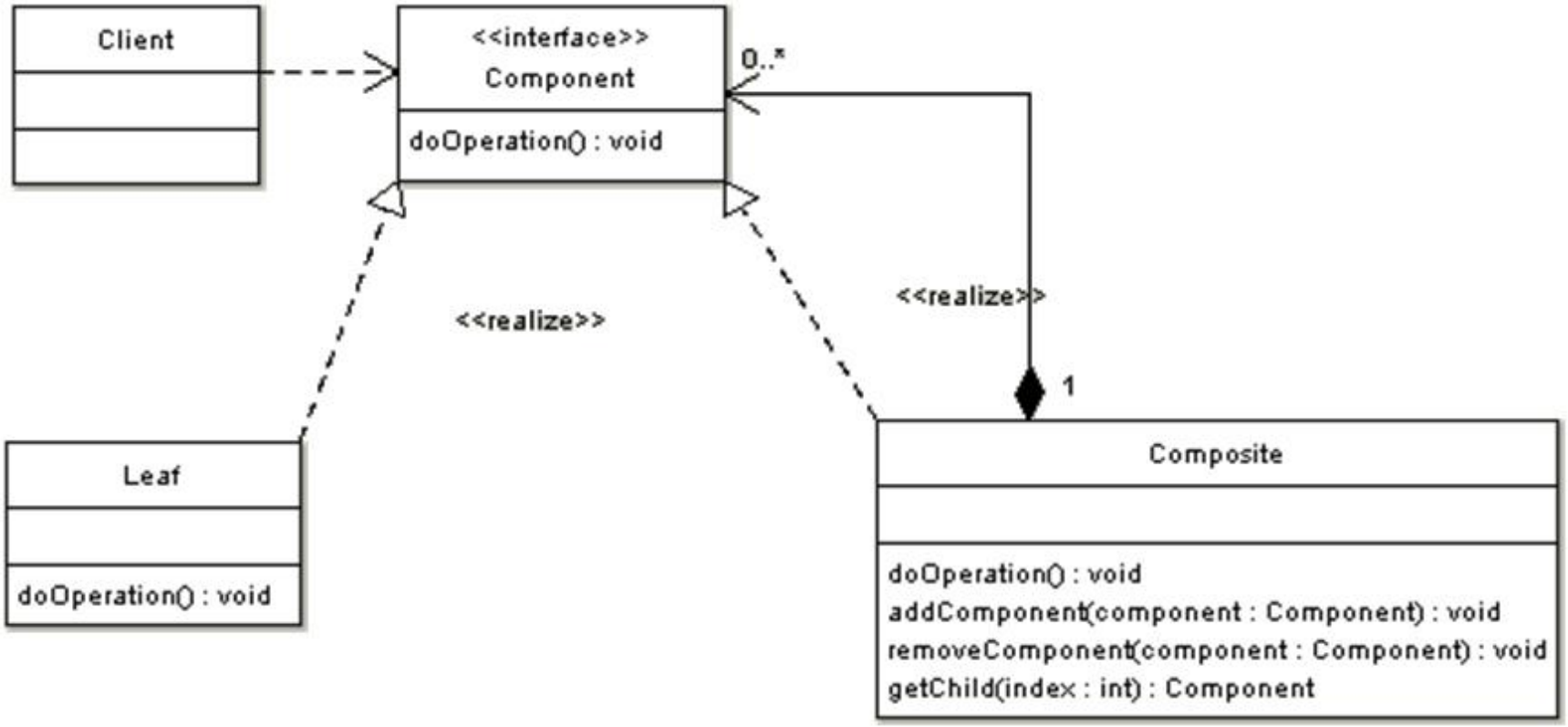

Implementation
facility1()
facility2()
facility3()
facility4()
Abstraction
service1()
service2()
service3()
Implemented with
Implementation1
facility1()
facility2()
facility3()
facility4()
(Adapter)
Implementation2
facility1()
facility2()
facility3()
facility4()
(Adapter)
ClientService1
serviceA()
serviceB()
ClientService2
serviceC()
serviceD()
serviceE()
Library1
Library2

Composite Pattern

Composite Pattern

- Motivation
  - We want to well organize individual objects and compositions of objects when they can be used equally
- Solution
  - Compose objects into a tree structure to represent the part-whole hierarchy

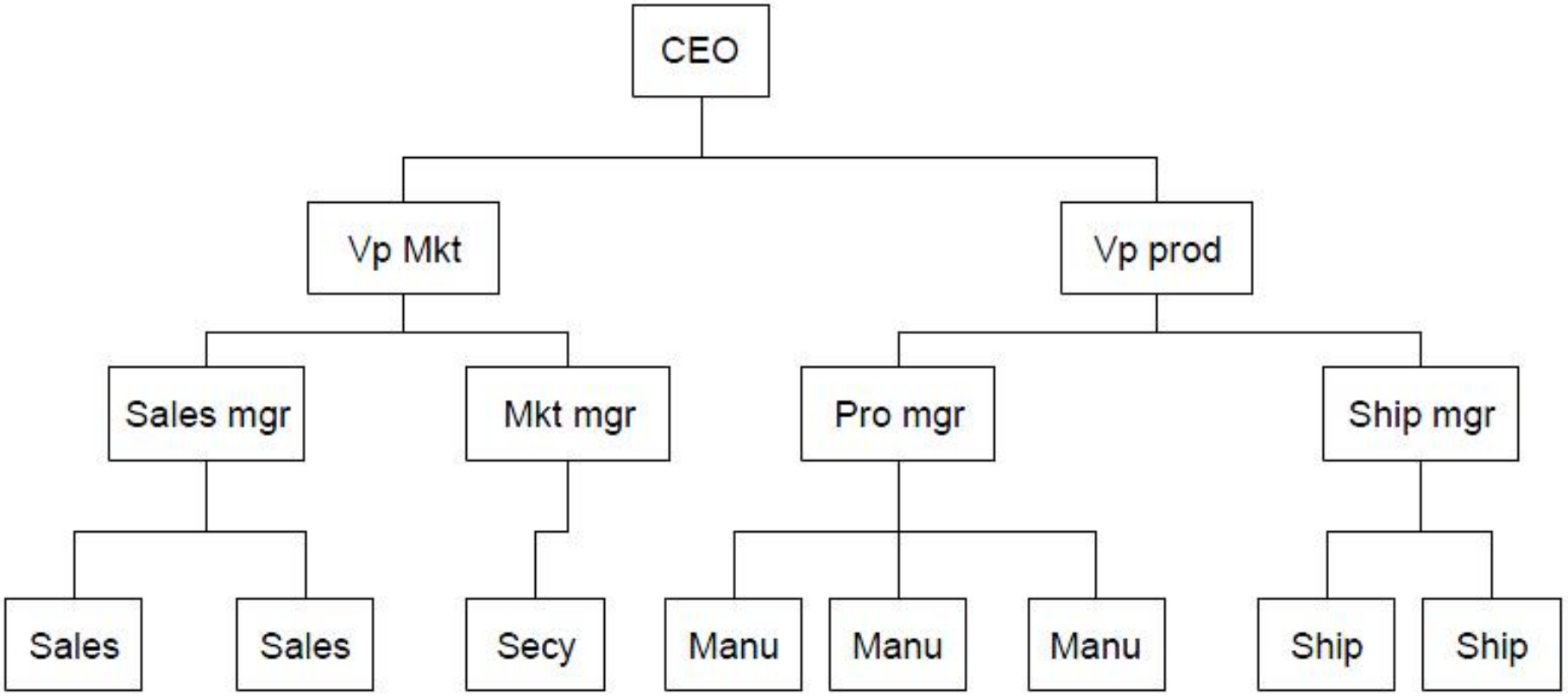
Composite Pattern



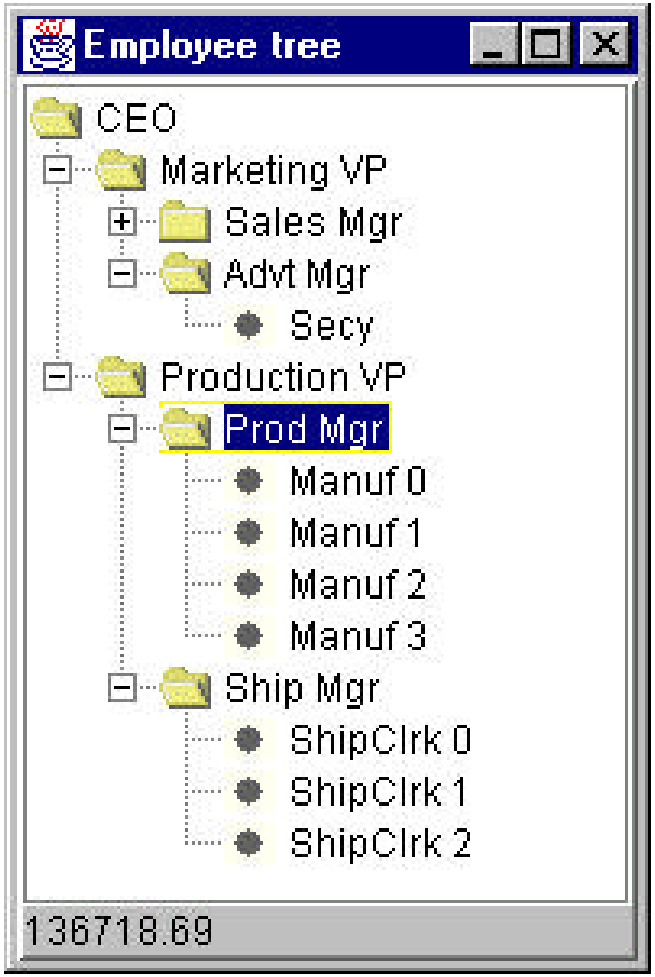
Participants

- Component
  - Interface for all objects of Leaf and Composite
- Leaf
  - Implements the Component interface and has no child
- Composite
  - Implements the Component interface
  - Stores child Components
  - Provides additional methods for adding, removing and obtaining components
- Client
  - Manipulates all objects of Leaf and Composite using Component interface

Example


CEO
Vp Mkt
Vp prod
Sales mgr
Mkt mgr
Pro mgr
Ship mgr
Sales
Sales
Secy
Manu
Manu
Manu
Ship
Ship

Example


Employee tree
CEO
Marketing VP
Sales Mgr
Advt Mgr
Secy
Production VP
Prod Mgr
Manuf 0
Manuf 1
Manuf 2
Manuf 3
Ship Mgr
ShipClrk 0
ShipClrk 1
ShipClrk 2
136718.69

Example

```
public class Employee {
        String name;
        float salary;
        Vector subordinates;

        public Employee(String _name, float _salary) {
                name = _name;
                salary = _salary;
                subordinates = new Vector();
        }

        public float getSalary() {
                return salary;
        }

        public String getName() {
                return name;
        }

        public void add(Employee e) {
                subordinates.addElement(e);
        }

        public void remove(Employee e) {
                subordinates.removeElement(e);
        }

        //get a list of employees of a given supervisor
        public Enumeration elements() {
                return subordinates.elements();
        }

        //returns a sum of salaries for the employee and his subordinates
        public float getSalaries() {
                float sum = salary; //this one's salary
                //add in subordinates salaries
                for(int i = 0; i < subordinates.size(); i++) {
                        sum += ((Employee)subordinates.elementAt(i)).getSalaries();
                }
                return sum;
        }
}
```

Example

```
boss = new Employee("CEO", 200000);
boss.add(marketVP =new Employee("Marketing VP", 100000));
boss.add(prodVP =new Employee("Production VP", 100000));
marketVP.add(salesMgr =new Employee("Sales Mgr", 50000));
marketVP.add(advMgr =new Employee("Advt Mgr", 50000));
//add salesmen reporting to Sales Manager
for (int i=0; i<5; i++)
        salesMgr .add(new Employee("Sales "+new Integer(i).toString(),
30000.0F+(float)(Math.random()-0.5)*10000));
advMgr.add(new Employee("Secy", 20000));
prodVP.add(prodMgr =new Employee("Prod Mgr", 40000));
prodVP.add(shipMgr =new Employee("Ship Mgr", 35000));
//add manufacturing staff
for (int i = 0; i < 4; i++)
        prodMgr.add( new Employee("Manuf "+new Integer(i).toString(),
25000.0F+(float)(Math.random()-0.5)*5000));
//add shipping clerks
for (int i = 0; i < 3; i++)
        shipMgr.add( new Employee("ShipClrk "+new Integer(i).toString(),
20000.0F+(float)(Math.random()-0.5)*5000));
```

Example

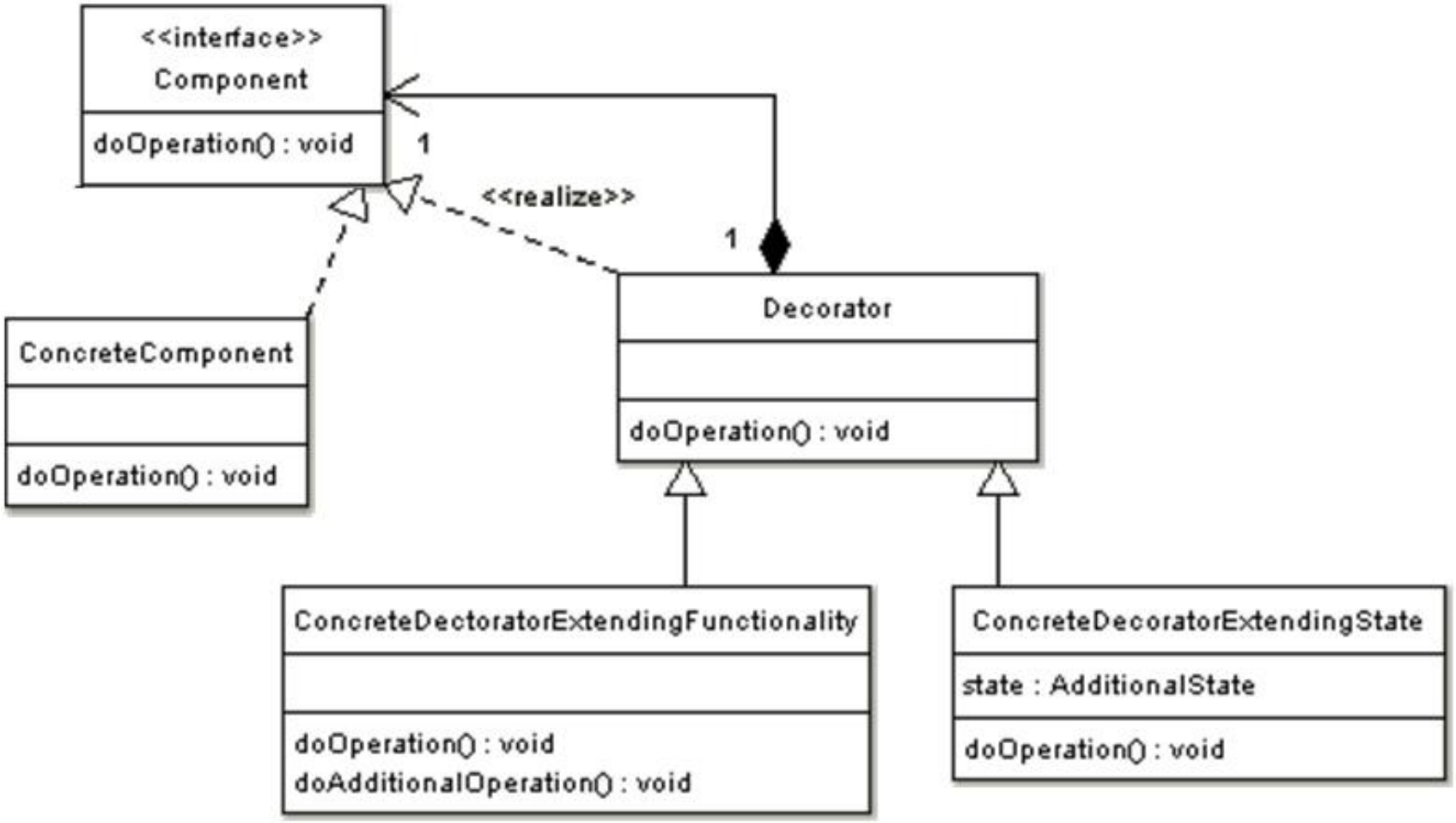
```
private void addNodes(DefaultMutableTreeNode pnode, Employee emp) {
        DefaultMutableTreeNode node;
        Enumeration e = emp.elements();
        while(e.hasMoreElements()){
                Employee newEmp = (Employee)e.nextElement();
                node = new DefaultMutableTreeNode(newEmp.getName());
                pnode.add(node);
                addNodes(node, newEmp);
        }
}
```

Decorator Pattern

Decorator Pattern

- Motivation
  - We want to modify the behavior of a class without deriving a new class from it
- Solution
  - Wrap object of the class and add additional responsibilities dynamically to it

Decorator Pattern



Participants

- Component
  - Defines the interface for ConcreteComponent
- ConcreteComponent
  - Defines an object that can have responsibilities added to it dynamically
- Decorator
  - Conforms to the Component interface
  - Maintains a reference to the object of ConcreteComponent
- ConcreteDecorator
  - Adds responsibilities to the object of ConcreteComponent

Example

```
public class Decorator extends JComponent {
    public Decorator(JComponent c) {
        setLayout(new BorderLayout());
        //add component to container
        add("Center", c);
    }
}
```

Example

```
public class CoolDecorator extends Decorator{
        boolean mouse_over;
        //true when mouse over button
        JComponent thisComp;

        public CoolDecorator(JComponent c) {
                super(c);
                mouse_over = false;
                thisComp = this; //save this component
                //catch mouse movements in inner class

                c.addMouseListener(new MouseAdapter() {
                        public void mouseEntered(MouseEvent e) {
                                mouse_over=true;
                                //set flag when mouse over
                                thisComp.repaint();
                        }
                        public void mouseExited(MouseEvent e) {
                                mouse_over=false;
                                //clear if mouse not over
                                thisComp.repaint();
                        }
                });
        }

        public void paint(Graphics g) {
                super.paint(g);
                if(! mouse_over) {
                        Dimension size = super.getSize();
                        g.setColor(Color.lightGray);
                        g.drawRect(0, 0, size.width-1, size.height-1);
                        g.drawLine(size.width-2, 0, size.width-2, size.height-1);
                        g.drawLine(0, size.height-2, size.width-2, size.height-2);
                }
        }
}
```

Example

```
super ("Deco Button");
JPanel jp = new JPanel();
getContentPane().add(jp);
jp.add( new CoolDecorator(new JButton("Cbutton")));
jp.add( new CoolDecorator(new JButton("Dbutton")));
jp.add(Quit = new JButton("Quit"));
Quit.addActionListener(this);
```

Example

```
public class SlashDecorator extends Decorator {
        int x1, y1, w1, h1; //saved size and posn

        public SlashDecorator(JComponent c) {
                super(c);
        }

        public void setBounds(int x, int y, int w, int h) {
                x1 = x; y1= y; //save coordinates
                w1 = w; h1 = h;
                super.setBounds(x, y, w, h);
        }

        public void paint(Graphics g) {
                super.paint(g); //draw button
                g.setColor(Color.red); //set color
                g.drawLine(0, 0, w1, h1); //draw red line
        }
}
```

Example

```
jp.add(new SlashDecorator( new
CoolDecorator(new JButton("Dbutton"))));
```

Decorator vs. Adapter

- Similarity: both make changes in class following a desired interface
- Difference: decorator maintains the desired interface of one class and adds features, while adapter changes the interface of another class to adapt it to the desired interface