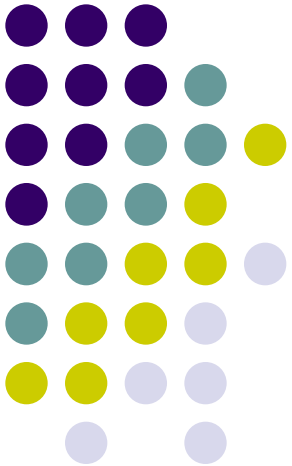
Software Architecture

Structural Patterns

Zhenyan Ji

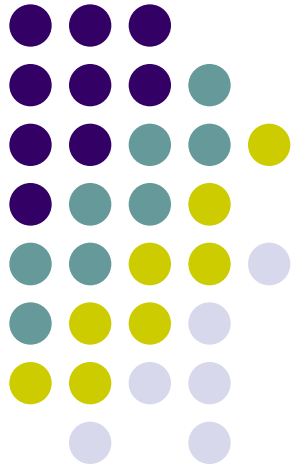
zhyji@bjtu.edu.cn



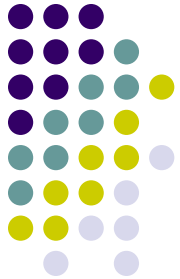
Structural Patterns

- describe how classes and objects can be combined to form larger structures.
  - Adapter Pattern
  - Bridge Pattern
  - Composite Pattern
  - Decorator Pattern
  - Façade pattern
  - Flyweight pattern
  - Proxy Pattern

Adapter Pattern

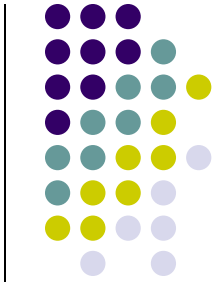


ADAPTER PATTERN



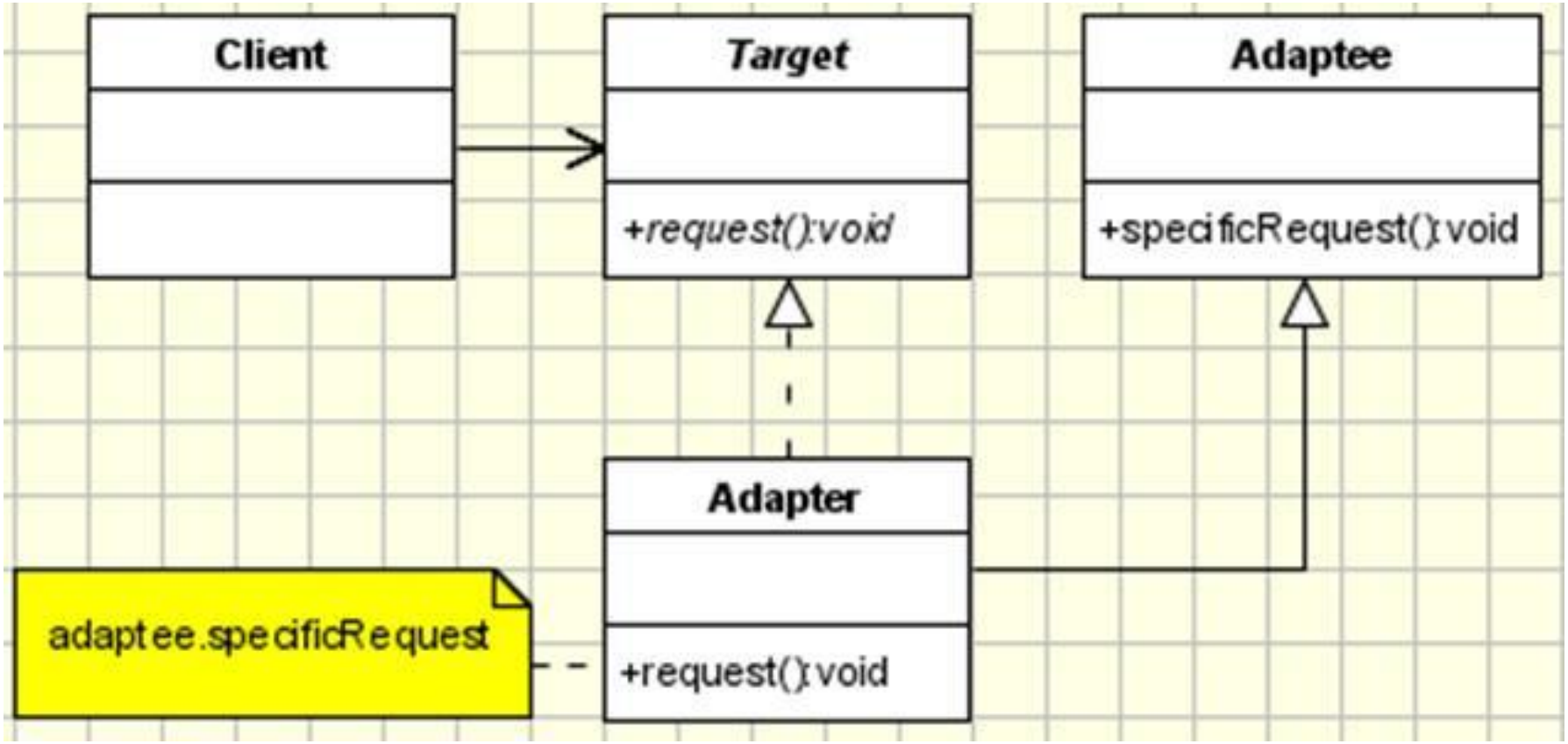
- The concept of an adapter: we write a class that has the desired interface and then make it communicate with the class that has a different interface.
- Intent
  - Convert the interface of a class into another interface clients expect.
  - Adapter lets classes work together that couldn’t otherwise because of incompatible interfaces.

ADAPTER PATTERN

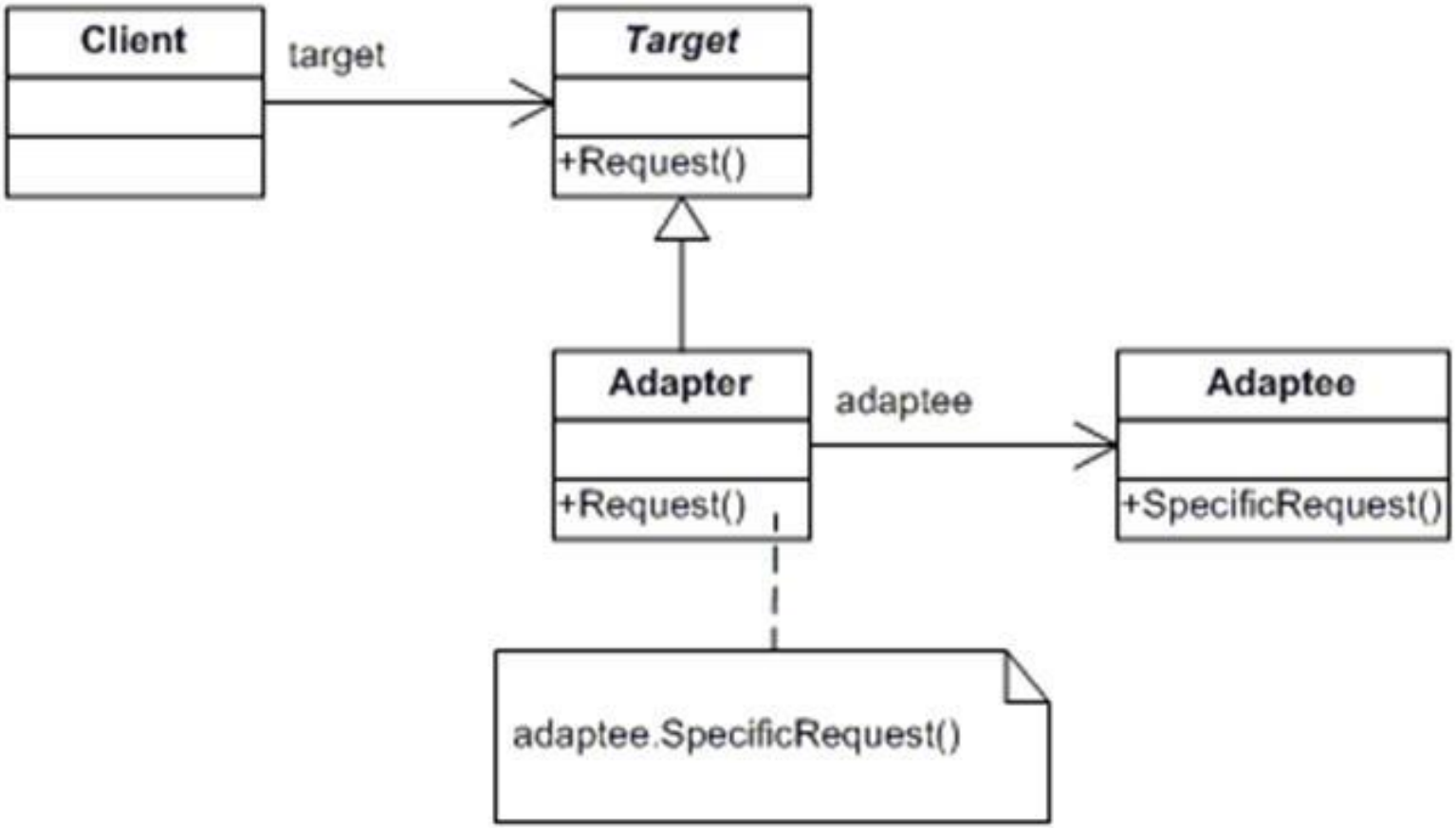


- Two ways
  - by inheritance: we derive a new class from the nonconforming one and add the methods we need to make the new derived class match the desired interface. class adapters
  - by object composition: include the original class inside the new one and create the methods to translate calls within the new class. object adapters

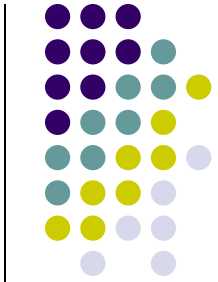
Class Adapters - Based on (Multiple) Inheritance



Objects Adapters - Based on Delegation

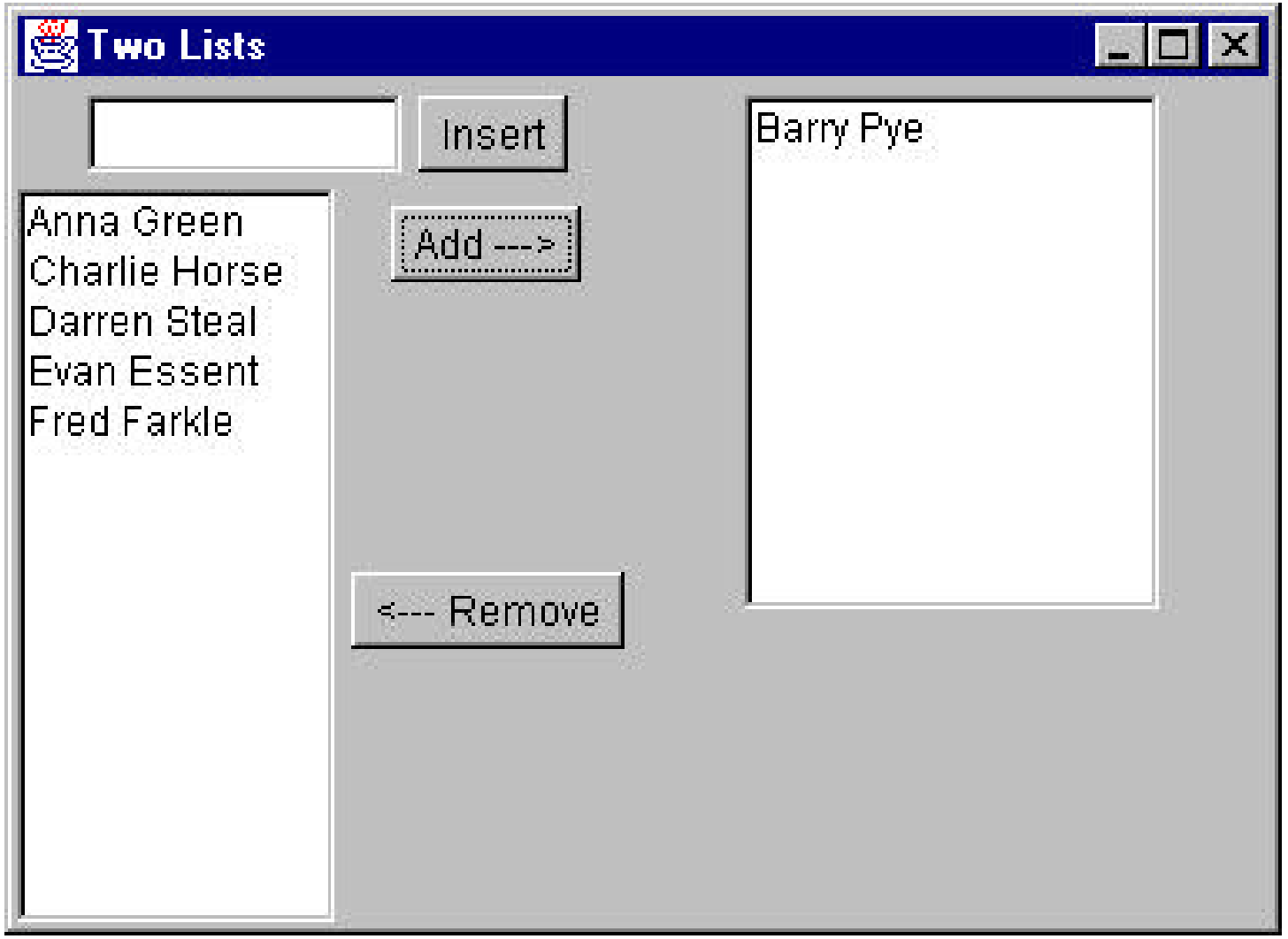


ADAPTER PATTERN

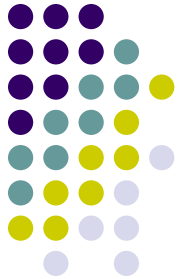


- **participants**
  - **Target :** defines the domain-specific interface that Client uses.
  - **Adapter:** adapts the interface Adaptee to the Target interface.
  - **Adaptee:** defines an existing interface that needs adapting.
  - **Client**
    - collaborates with objects conforming to the Target interface.

Example


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

Example: AWT

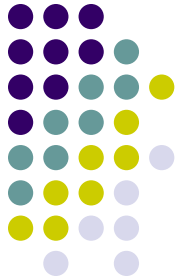


```
public void actionPerformed(ActionEvent e){
   Button b = (Button)e.getSource();
   if(b == Add)
        addName();
   if(b == MoveRight)
        moveNameRight();
   if(b == MoveLeft)
        moveNameLeft();
}
```

Example: AWT

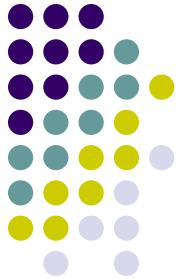
```
private void addName() {
   if (txt.getText().length() > 0) {
        leftList.add(txt.getText());
        txt.setText("");          }
}
private void moveNameRight() {
   String sel[] = leftList.getSelectedItems();
   if (sel != null) {
        rightList.add(sel[0]);
        leftList.remove(sel[0]);    }
}
```

Example: AWT



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

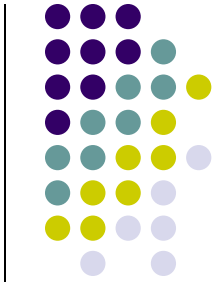
Using the JFC JList Class



- The JFC JList class is markedly different than the AWT List class
- writing an adapter to make the JList class look like the List class seems a sensible solution.

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

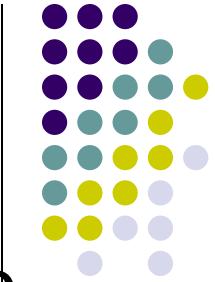
Example:



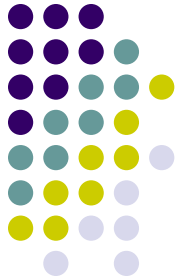
- create a class that emulates the List class. Firstly, define the needed methods as an interface.

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

The Object Adapter

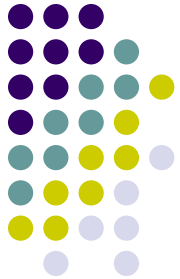


```
public class JawtList extends JScrollPane
    implements awtList {
    private JList listWindow;
    private JListData listContents;
    public JawtList(int rows) {
        listContents = new JListData();
        listWindow = new JList(listContents);
        getViewport().add(listWindow);       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



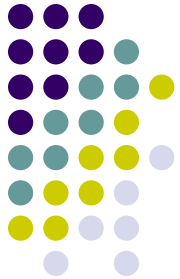
```
public void add(String s) {
        listContents.addElement(s);    }
   public void remove(String s) {
        listContents.removeElement(s);      }
   public String[] getSelectedItems() {
        Object[] obj =
   listWindow.getSelectedValues();
        String[] s = new String[obj.length];
    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

The Object Adapter



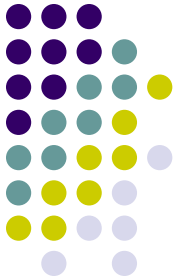
- the actual data handling takes place in the JlistData class. This class is derived from the AbstractListModel, which defines the following methods:
  - addListDataListener(l) : Add a listener for changes in the data.
  - removeListDataListener(l): Remove a listener

The Object Adapter



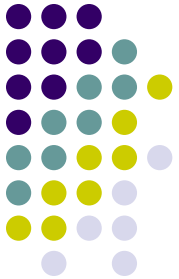
- fireContentsChanged(obj, min,max): Call this after any change occurs between the two indexes min and max
- fireIntervalAdded(obj,min,max): Call this after any data has been added between min and max.
- fireIntervalRemoved(obj, min, max): Call this after any data has been removed between min and max.

The Object Adapter



```
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());                     }
```

The Object Adapter

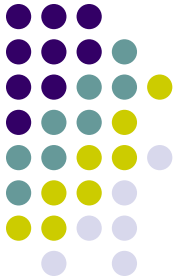


```
//-----------------------------------------
    public void removeElement(String s) {
         data.removeElement(s);
        fireIntervalRemoved(this, 0, data.size());
    }
}
```

The Class Adapter

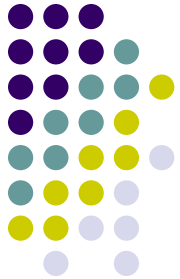
```
public class JclassAwtList extends JList
   implements awtList {
   private JListData listContents;
   //-----------------------------------
   public JclassAwtList(int rows) {
        listContents = new JListData();
        setModel(listContents);
        setPrototypeCellValue("Abcdefg
   Hijkmnop");
   }
```

The Class Adapter



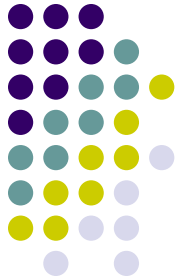
```
leftList = new JclassAwtList(15);
JScrollPane lsp = new JScrollPane();
pLeft.add("Center", lsp);
lsp.getViewport().add(leftList);
```

Adapters in Java

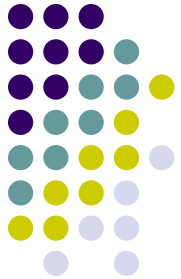


- there are a number of adapters built into the Java language.
  - WindowAdapter, ComponentAdapter, ContainerAdapter, FocusAdapter, KeyAdapter, MouseAdapter, and MouseMotionAdapter.

Adapters in Java



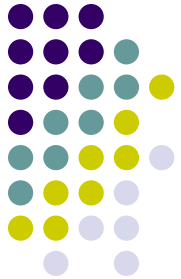
```
public void mainFrame extends Frame
   implements WindowListener {
   public void mainFrame() {
        addWindowListener(this);
       //frame listens for window events
   }
   public void windowClosing(WindowEvent
   wEvt) {
        System.exit(0);
       //exit on System exit box clicked
   }
```



```
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){}
}
```

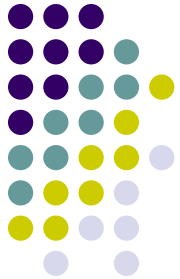
Java Adapter

```
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();                    }
}
```



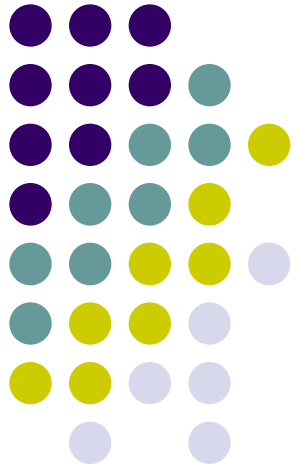
```
//make an extended window adapter which
//closes the frame when the closing event is
   received
class WindAp extends WindowAdapter {
  public void windowClosing(WindowEvent e)
   {
        System.exit(0);       }
}
```

Anonymous inner class

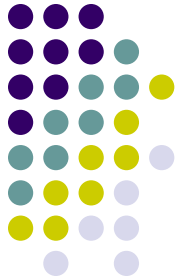


```
//create window listener for window close click
addWindowListener(new WindowAdapter()
{
   public void windowClosing(WindowEvent e)
   {
        System.exit(0);}
});
```

THE COMPOSITE PATTERN

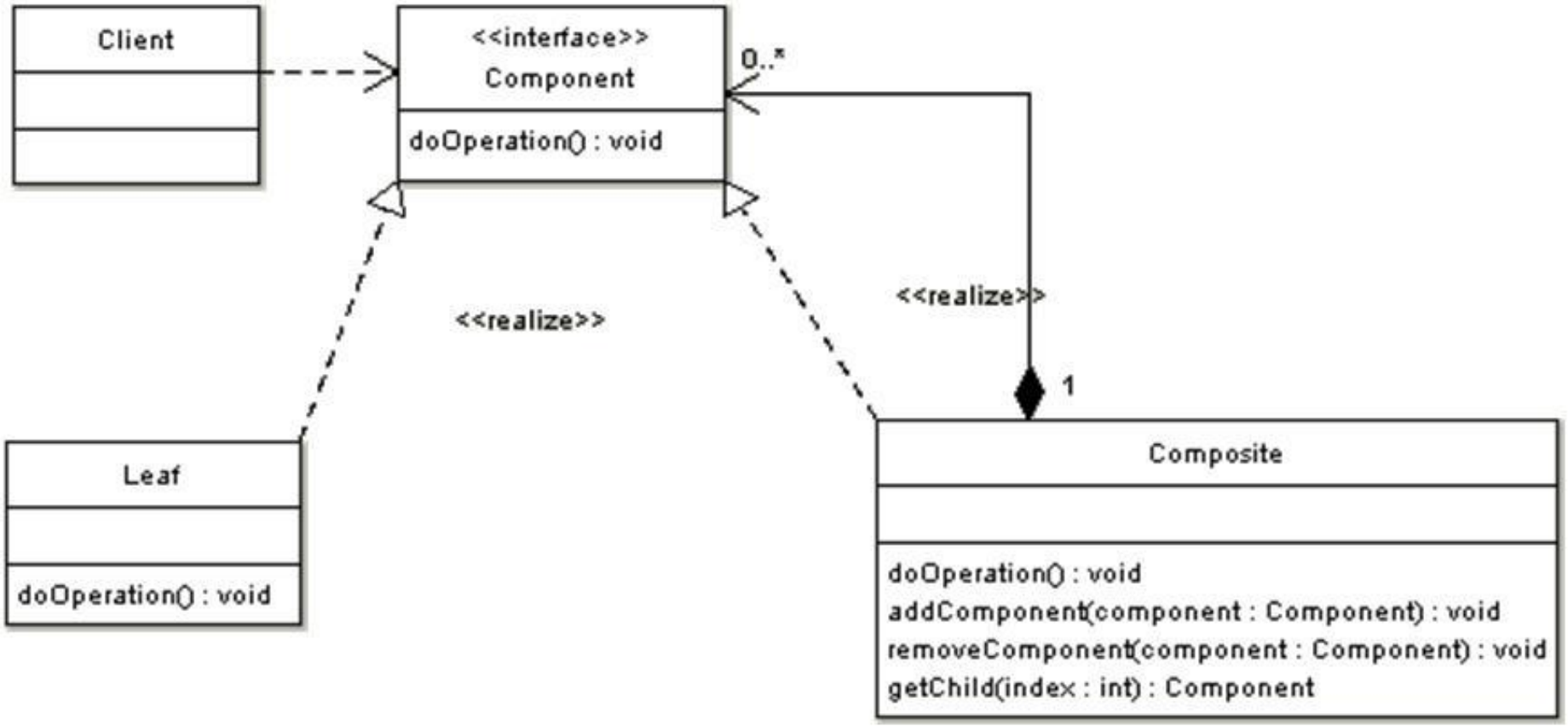


THE COMPOSITE PATTERN

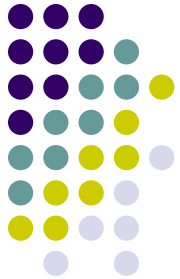


- Intent
  - The intent of this pattern is to compose objects into tree structures to represent part-whole hierarchies.
  - Composite lets clients treat individual objects and compositions of objects uniformly.

THE COMPOSITE PATTERN

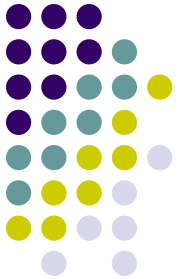


COMPOSITE PATTERN



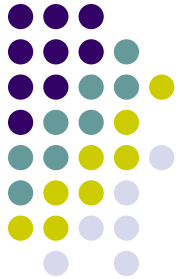
- **Participants:**
  - **Client** - The client manipulates objects in the hierarchy using the component interface.
  - **Component** - Component is the abstraction for leafs and composites. It defines the interface that must be implemented by the objects in the composition.

COMPOSITE PATTERN



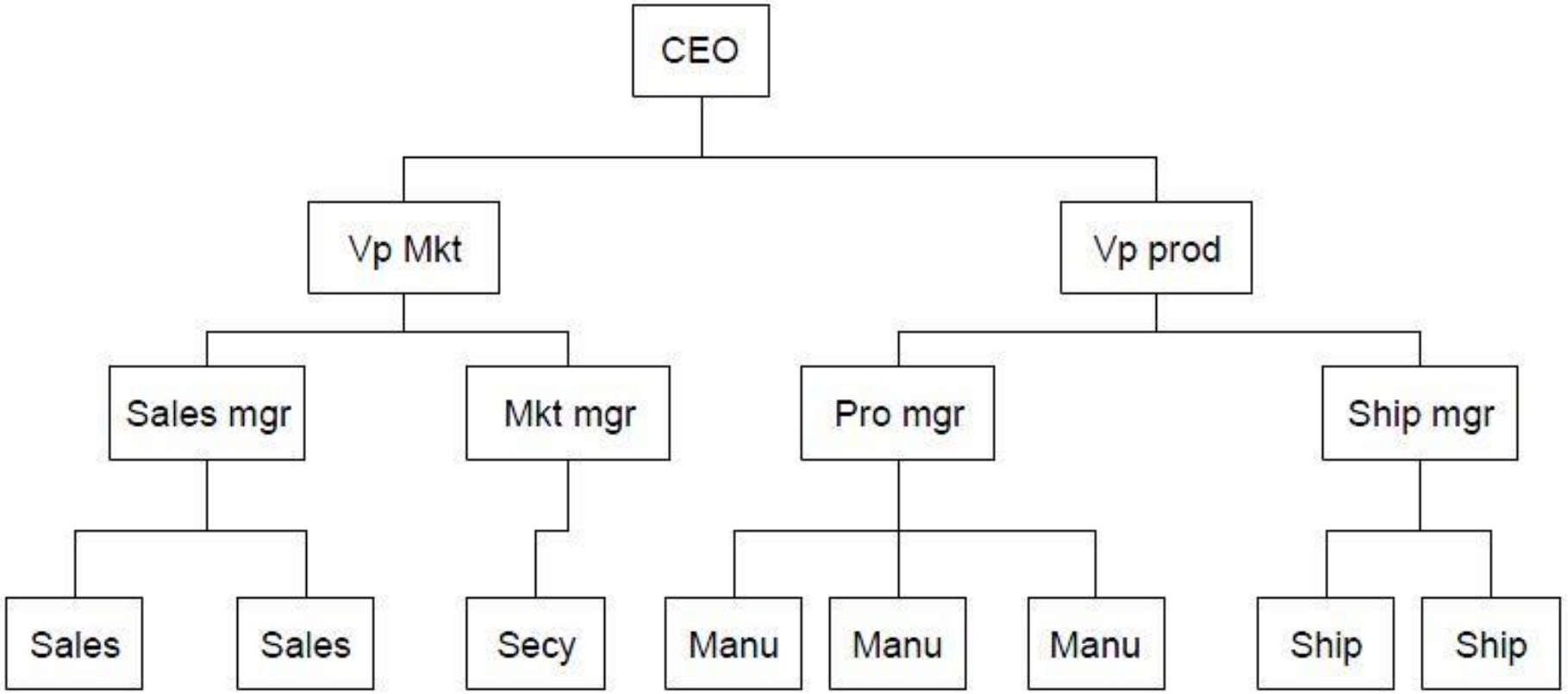
- **Leaf** - Leafs are objects that have no children. They implement services described by the Component interface.

THE COMPOSITE PATTERN



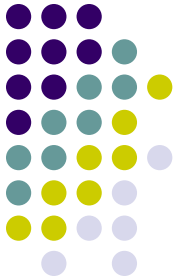
- **Composite** - A Composite stores child components in addition to implementing methods defined by the component interface. Composites implement methods defined in the Component interface by delegating to child components. In addition composites provide additional methods for adding, removing, as well as getting components.

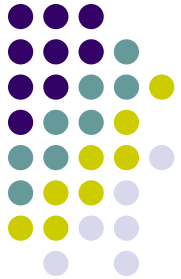
Example


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

Example

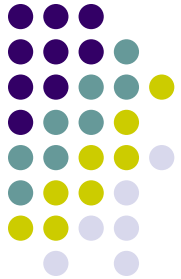
```
public class Employee {
    String name;
    float salary;
    Vector subordinates;
    //----------------------------------
    public Employee(String _name, float
    _salary) {
        name = _name;
        salary = _salary;
        subordinates = new Vector();   }
```





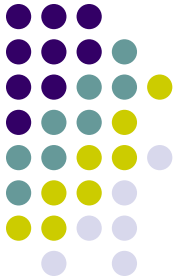
```
//-----------------------------------
    public float getSalary() {
        return salary;   }
    //-----------------------------------
    public String getName() {
        return name;   }
```

Example



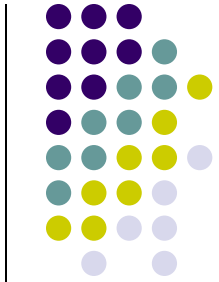
- created a Vector called *subordinates* at the time the class was instantiated.
- if the employee has subordinates, add them to the Vector with the *add* method and remove them with the *remove* method.

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



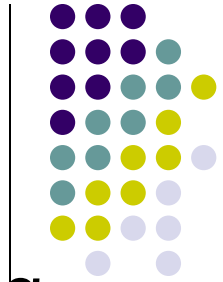
```
public void remove(Employee e) {
    subordinates.removeElement(e); }
//get a list of employees of a given
    supervisor
public Enumeration elements() {
    return subordinates.elements(); }
```

Example: the common interface

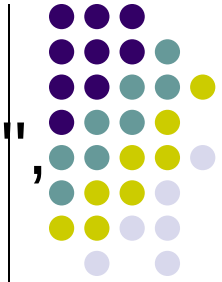


```
//returns a sum of salaries for the employee
   and his subordinates
public float getSalaries() {
   float sum = salary; //this one's salary
   //add in subordinates salaries
   for(int i = 0; i < subordinates.size(); i++) {
        sum +=
        ((Employee)subordinates.elementAt(i)).
   getSalaries();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

Example:Building the Employee Tree



```
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));
```

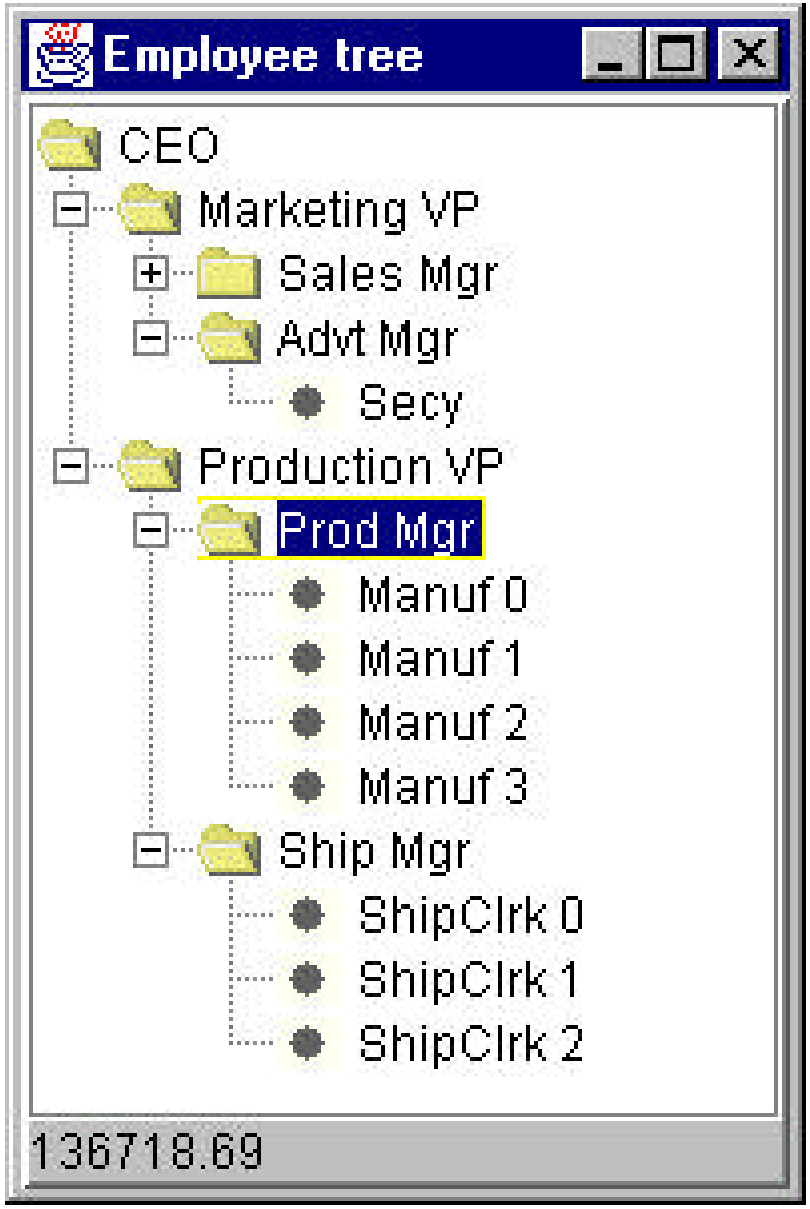


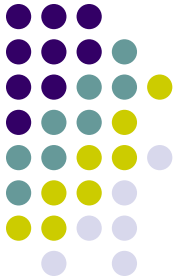
```
advMgr.add(new Employee("Secy", 20000));
prodVP.add(prodMgr =new Employee("Prod Mgr",
    40000));
prodVP.add(shipMgr =new Employee("Ship Mgr",
    35000));
//add manufacturing staff
for (int i = 0; i < 4; i++)
    prodMgr.add( new Employee("Manuf "+new
    Integer(i).toString(),
    25000.0F+(float)(Math.random()-0.5)*5000));
//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: Build visual JTree list

```
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);
} }
```

Example

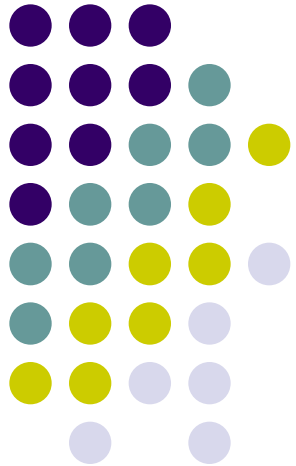

Employee tree
CEO
Marketing VP
Sales Mgr
Advt Mgr
Secy
Production VP
Prod Mgr
Manuf 0
Manuf 1
Manuf 2
Manuf 3
Ship Mgr
ShipClrk 0
ShipClrk 1
ShipClrk 2
136718.69



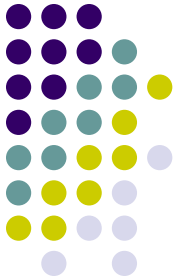
Applicability

- Composite can be used when clients should ignore the difference between compositions of objects and individual objects.

The Decorator Pattern

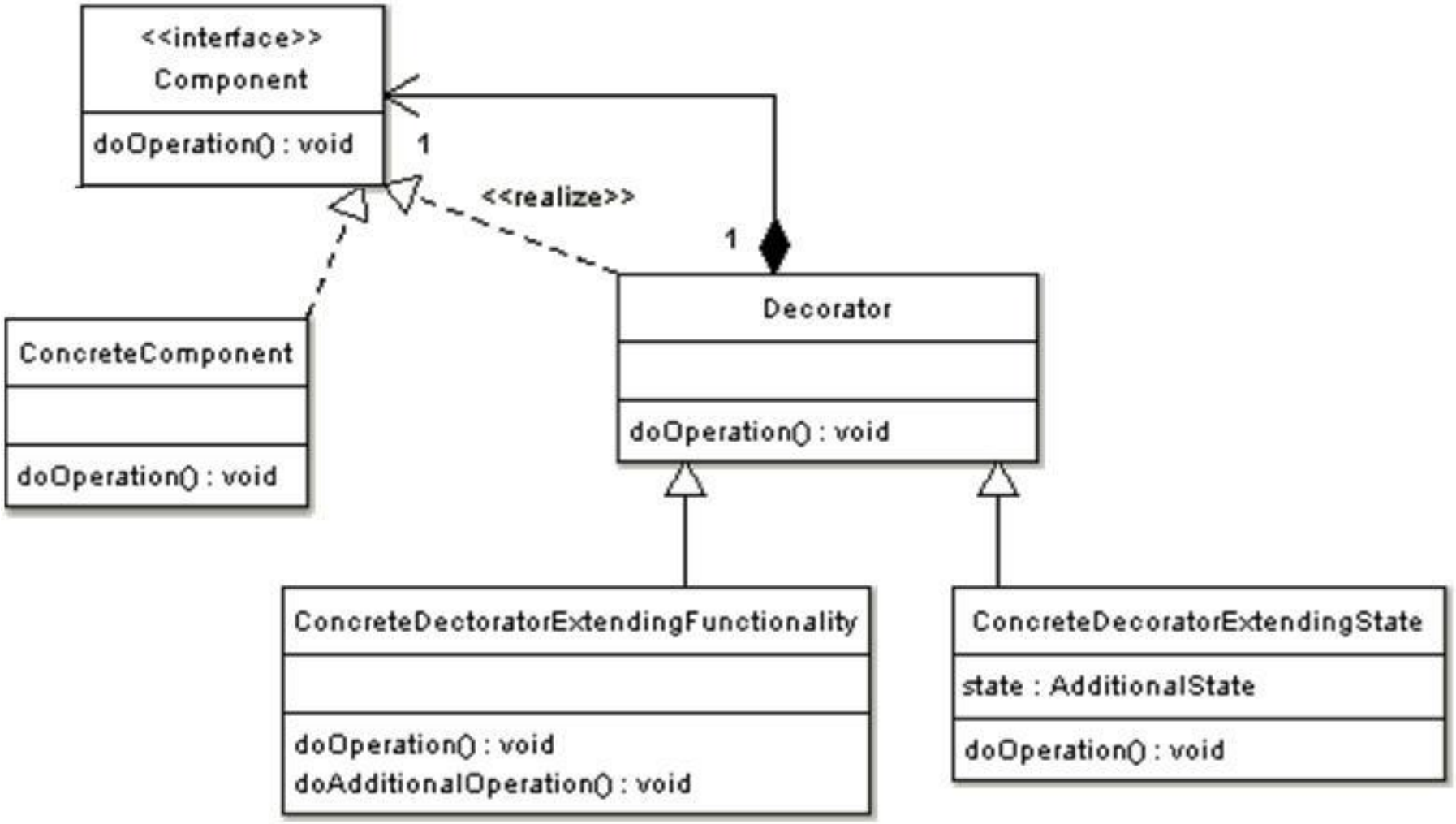


DECORATOR PATTERN

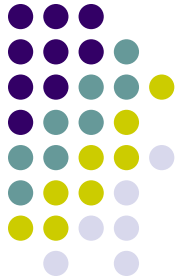


- The decorator pattern provides a way to modify the behavior of individual objects without having to create a new derived class.
- achieved by designing a new *decorator* class that wraps the original class.
- Intent
  - add additional responsibilities dynamically to an object.

DECORATOR PATTERN

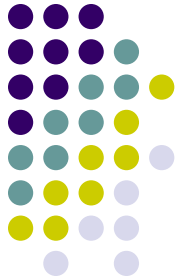


DECORATOR PATTERN



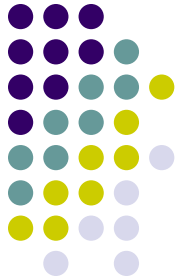
- **participants**
  - **Component**
    - defines the interface for objects that can have responsibilities added to them dynamically.
  - **ConcreteComponent**
    - defines an object to which additional responsibilities can be attached.

DECORATOR PATTERN



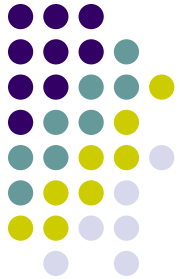
- **Decorator**
  - maintains a reference to a Component object and defines an interface that conforms to Component's interface.
- **ConcreteDecorator**
  - adds responsibilities to the component.

Decorator Pattern & Subclass



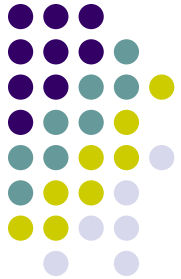
- The decorator pattern is an alternative to subclassing.
  - Subclassing adds behavior at compile time, and the change affects all instances of the original class
    - a new class would have to be made for every possible combination

Decorator Pattern & Subclass



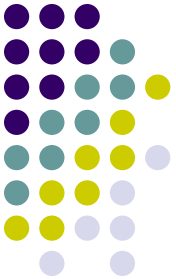
- Decorating can provide new behavior at runtime for individual objects.
  - decorators are objects, created at runtime, and can be combined on a per-use basis.

Example: Base Decorator Class

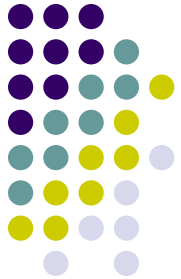


```
public class Decorator extends Jcomponent {
    public Decorator(JComponent c) {
        setLayout(new BorderLayout());
        //add component to container
        add("Center", c);
    }
}
```

Example: Concrete Decorator Class

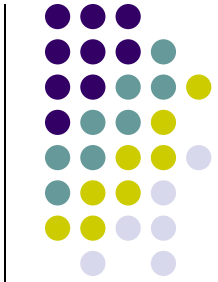


```
public class CoolDecorator extends Decorator{
    boolean mouse_over;
     //true when mouse over button
    JComponent thisComp;
    public CoolDecorator(JComponent c) {
          super(c);
          mouse_over = false;
          thisComp = this; //save this component
          //catch mouse movements 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();
                }
        });
    }
```

Example



```
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: Using a Decorator

```
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);
```

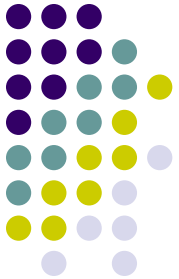
Decorate Decorator

```
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);
    }
```

Decorate Decorator

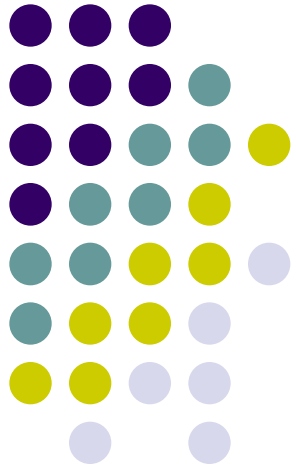
```
//-------------------------------------------
    public void paint(Graphics g) {
        super.paint(g); //draw button
        g.setColor(Color.red); //set color
        g.drawLine(0, 0, w1, h1); //draw red line
    }
}
```

Decorate Decorator

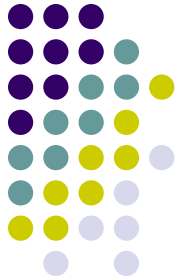


```
jp.add(new SlashDecorator( new
    CoolDecorator(new JButton("Dbutton"))));
```

The Façade Pattern

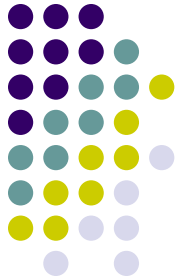


Facade Pattern



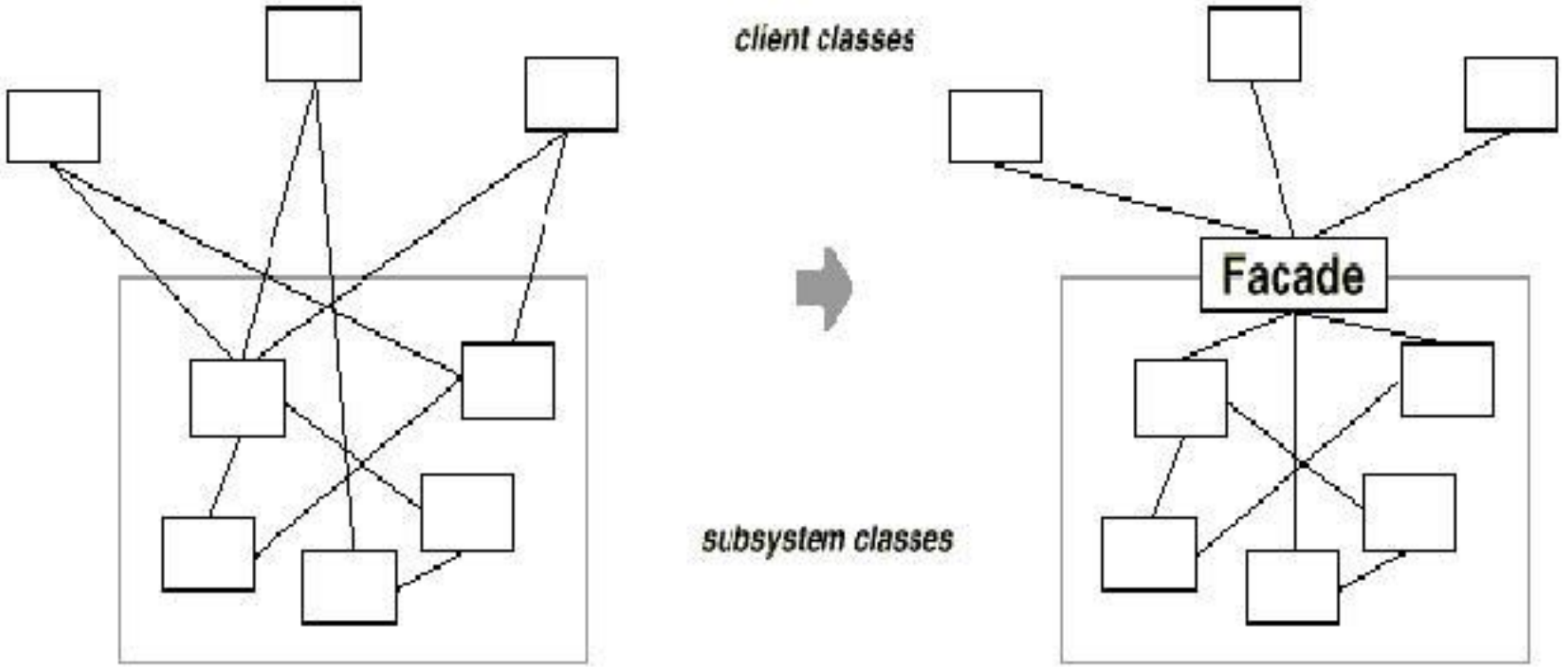
- Intent:
  - Provide a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easier to use.
- Motivation
  - Structuring a system into subsystems helps reduce complexity
  - Subsystems are groups of classes, or groups of classes and other subsystems

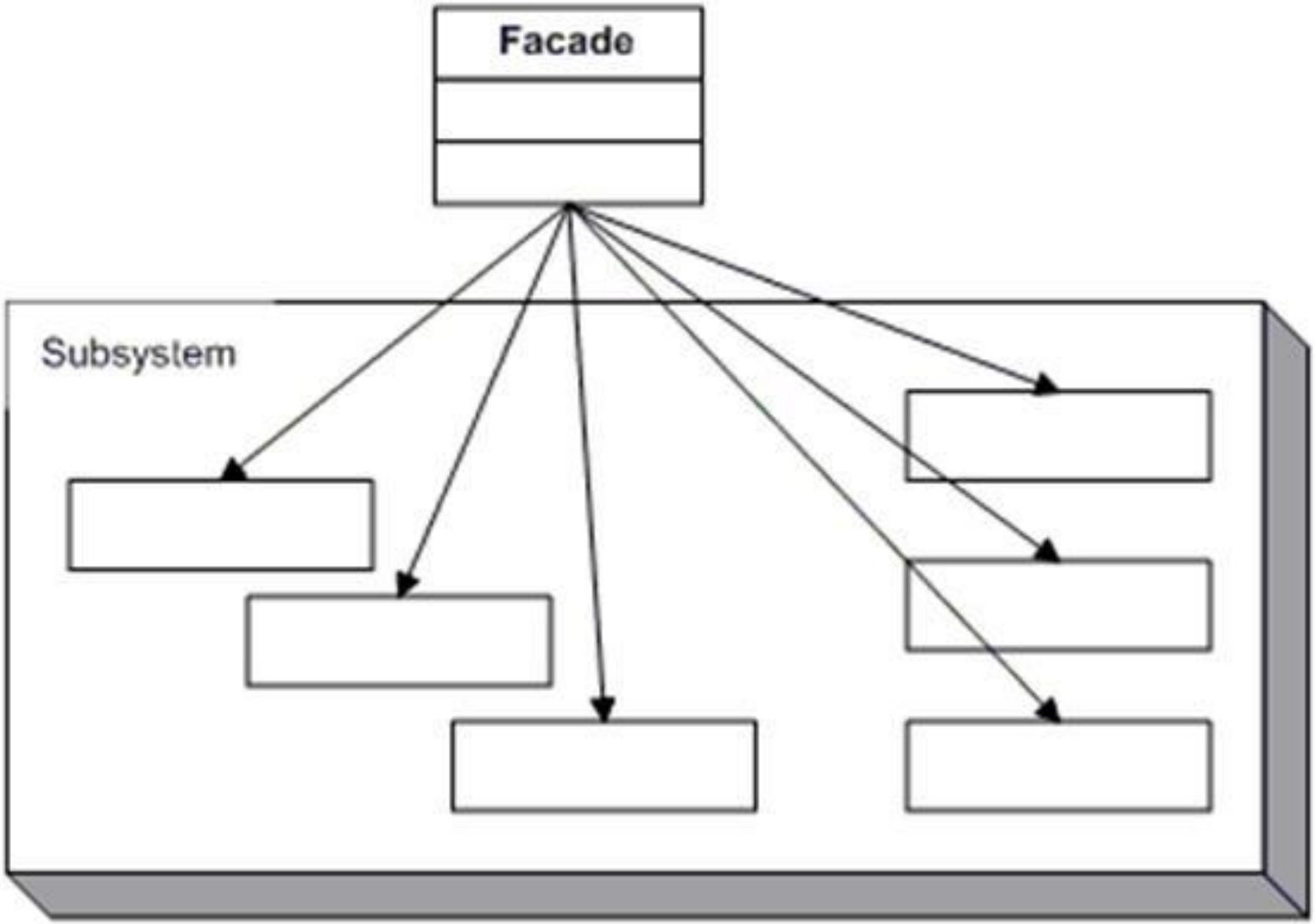
Facade Pattern



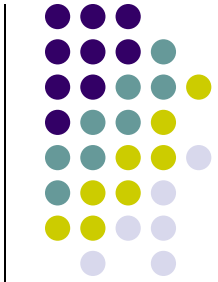
- The interface exposed by the classes in a subsystem or set of subsystems can become quite complex
- One way to reduce this complexity is to introduce a facade object that provides a single, simplified interface to the more general facilities of a subsystem

Facade Pattern



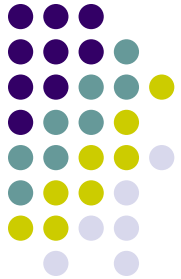

Facade
Subsystem

Facade Pattern



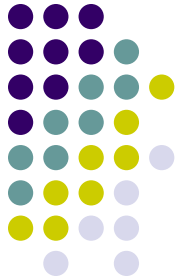
- **participants**
- **Facade**
  - knows which subsystem classes are responsible for a request.
  - delegates client requests to appropriate subsystem objects.
- **Subsystem classes**
  - implement subsystem functionality.
  - handle work assigned by the Facade object.

Facade Pattern



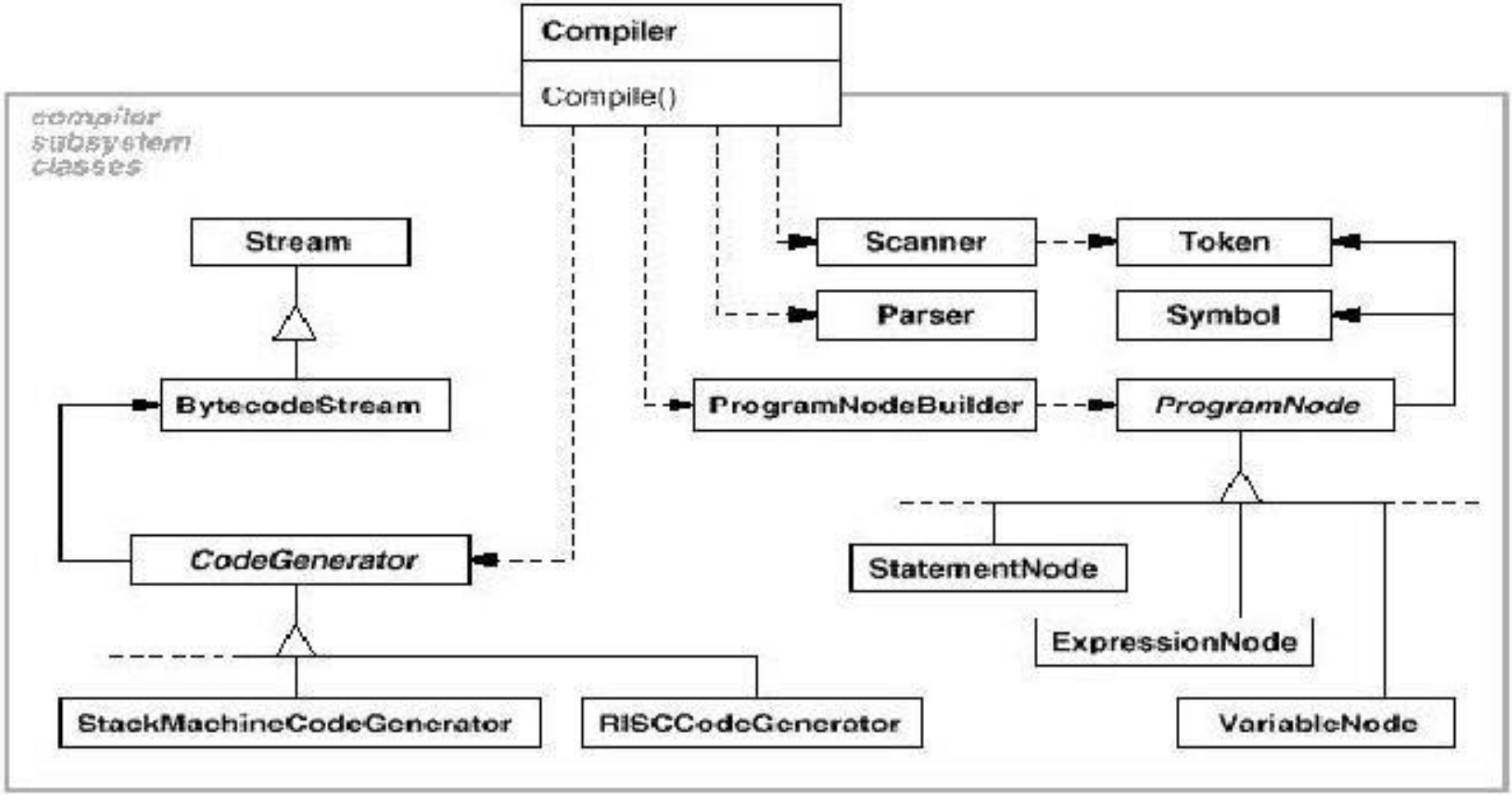
- have no knowledge of the facade and keep no reference to it.

The Facade Pattern

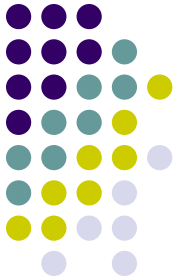


- Applicability
  - Use the Facade pattern:
    - To provide a simple interface to a complex subsystem. This interface is good enough for most clients; more sophisticated clients can look beyond the facade.
    - To decouple the classes of the subsystem from its clients and other subsystems, thereby promoting subsystem independence and portability

The Facade Pattern

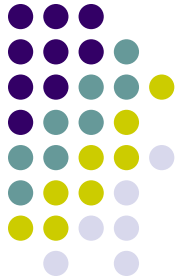


Consequences



- Benefits
  - It hides the implementation of the subsystem from clients, making the subsystem easier to use
  - It promotes weak coupling between the subsystem and its clients.
  - It reduces compilation dependencies in large software systems

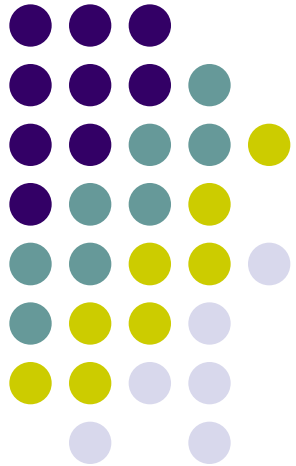
Consequences



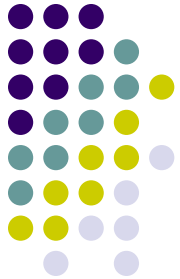
- It simplifies porting systems to other platforms.
- It does not prevent sophisticated clients from accessing the underlying classes
- Note that Facade does not add any functionality, it just simplifies interfaces

- Note: *It does not prevent clients from accessing the underlying classes!*

Proxy Pattern

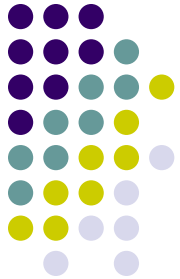


Proxy Pattern



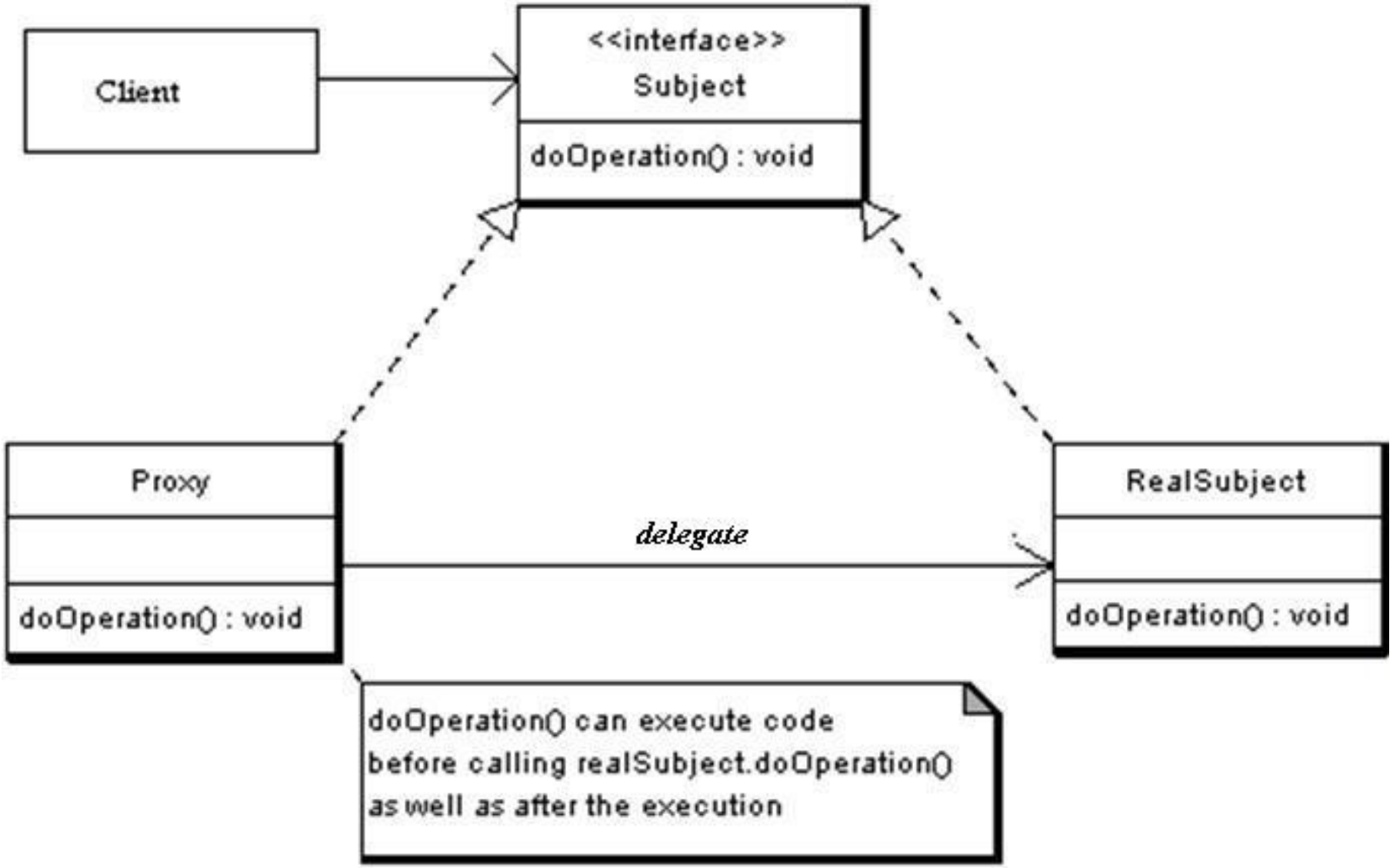
- Intent:
  - Provide a surrogate or placeholder for another object to control access to it.
- A proxy object
  - can act as the intermediary between the client and the target object

Proxy Pattern

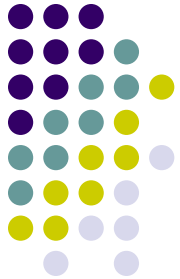


- has the same interface as the target object
- holds a reference to the target object and can forward requests to the target as required (delegation!)
- has the authority to act on behalf of the client to interact with the target object

Proxy Pattern

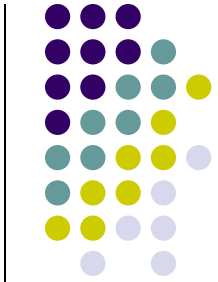


Proxy Pattern



- **Subject** - Interface implemented by the RealSubject and representing its services.
- **Proxy**
  - Maintains a reference that allows the Proxy to access the RealSubject.
  - Implements the same interface implemented by the RealSubject so that the Proxy can be substituted for the RealSubject.

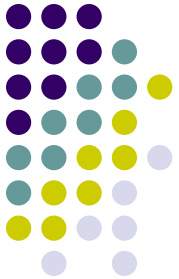
Proxy Pattern



- Controls access to the RealSubject and may be responsible for its creation and deletion.
- Other responsibilities depend on the kind of proxy.

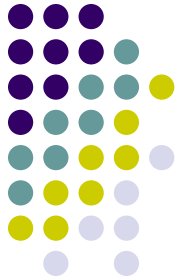
- **RealSubject** - the real object that the proxy represents.

Proxy Pattern



- Applicability
  - used when you need to represent a complex object by a simpler one.
  - If an object, such as a large image, takes a long time to load.

Proxy Pattern



- If the object is on a remote machine and loading it over the network may be slow, especially during peak network load periods.
- If the object has limited access rights, the proxy can validate the access permissions for that user.