Object Orientation with Design Patterns

Lecture 5:

Structural Patterns Decorator & Adapter

Structural Patterns

- A structural pattern describes how **classes can be combined together to form larger structures.**
- A *class pattern* describes how **inheritance can be used to provide more useful program interfaces.**

- An *object pattern* describes how **objects can be composed into larger structures using object composition** or by including objects within objects (aggregation).

Decorator Pattern

- **Intent:**
  Attach **additional responsibilities to an object dynamically**. Decorators provide a **flexible alternative** to **subclassing** for extending functionality.

- Also known as a **Wrapper class**.

Decorator Pattern - Applicability

Use Decorator:

- To **add responsibilities to individual objects dynamically and transparently**, that is, **without affecting** other objects.
- For **responsibilities** that can be **withdrawn**.
- When extension by **subclassing** is **impractical**. Sometimes a large number of independent extensions are possible and would **produce an explosion of subclasses to support every combination**.
  Or a class definition may be hidden or otherwise unavailable for subclassing.

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

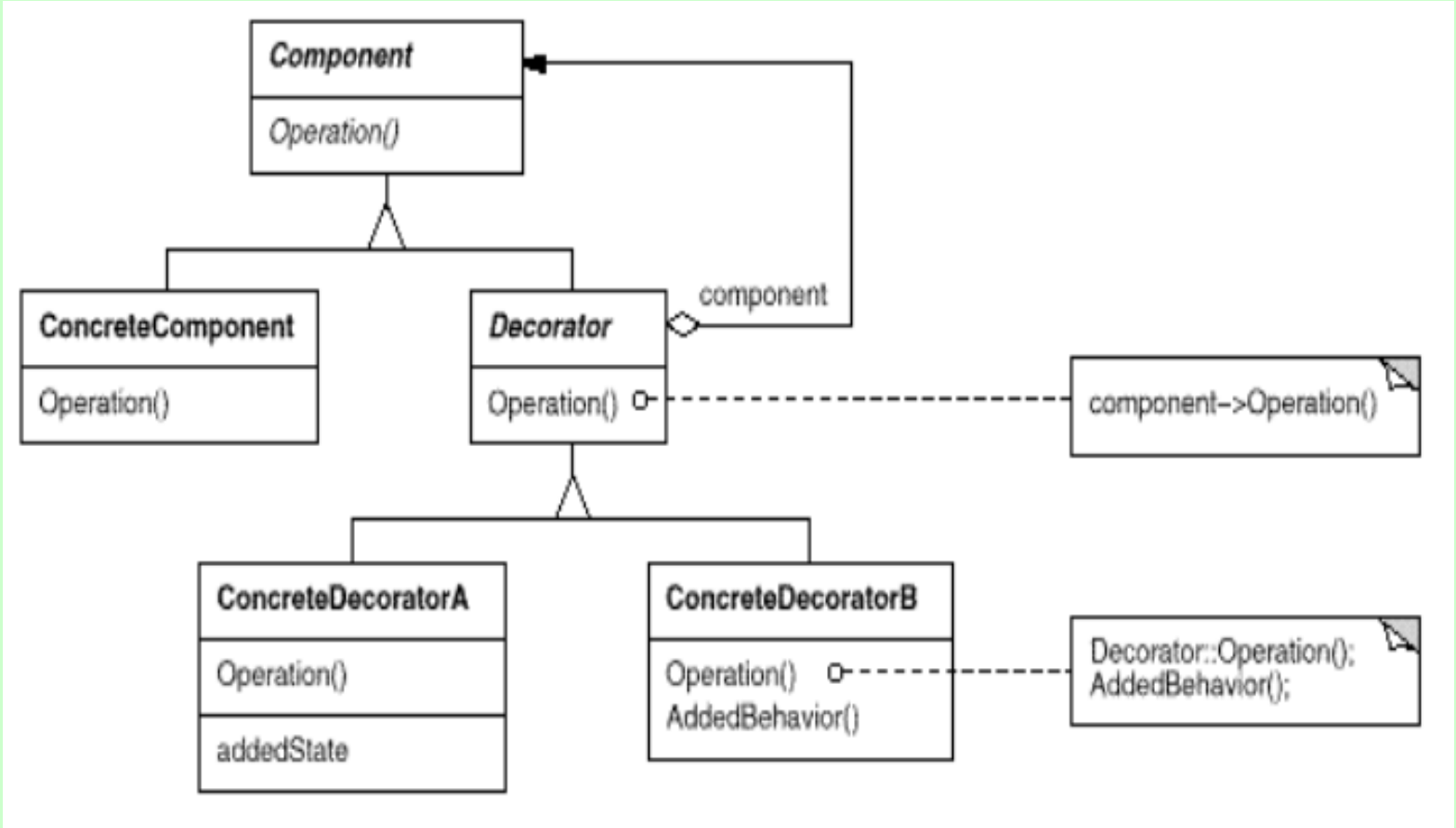
- **Decorator**

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



Decorator Collaborations

- Decorator **forwards requests** to its Component object.
  It may optionally perform additional operations before and after forwarding the request.

Decorator Pattern

- The ***Decorator Pattern*** provides us with a way to **modify the behavior of individual objects** without having to create a new derived class (subclass).

- Suppose we have a program that uses **eight objects**, but three of them need **additional features**. You could create a subclass for each of these objects, and in many cases this would be an acceptable solution.

- However, if each of the three objects requires **different features**, you would have to **create three separate subclasses**. Furthermore if one of the classes has features of both of the other classes , you begin to create complexity that is both confusing and unnecessary.

Decorator Pattern

- For example suppose we wanted to draw a **special border** around some of the buttons in a toolbar. If we created a **new derived button class**, this means that **all of the buttons** in this new class would have the **same border**, when this might not be our intent.

- Instead we can use a **Decorator Class** which ***decorates* the buttons**. Then we derive any number of specific decorators from the main decorator class, each of which performs a specific kind of decoration.

- In order to **decorate a button** there has to be an **object derived (subclassed) from an object** in the visual environment so that it can receive paint messages.

Decorator Pattern

- **The decorator is a graphical object, but it contains the object that it is decorating.**

.

- It might intercept some graphical method calls and perform some additional computation, and then it might pass them on to the underlying object that it is decorating.

.

Decorating a CoolButton

- Windows applications such as Internet Explorer and Netscape Navigator have a row of flat, unbordered buttons that highlight themselves with outline borders when you move the mouse over them.

- In Java there is no analogous button behavior, but we can get that behavior by ***decorating* a JButton**.

- In this case we decorate it by **drawing plain grey lines** over the **button borders**, effectively erasing them.

- The next examples illustrates how this can be achieved.

Decorating a CoolButton

- Decorators should be **derived from** some **general visual component class** and then **every message** for the actual button should be **forwarded from the decorator**.

- In Java we can derive our decorator class from the **JComponent class**. Then, since JComponent itself behaves as a container, it will **forward all method calls** to the components that it contains.

- In this case we simply **add the button** to the **JComponent**, and that button will receive all of the GUI method calls that the JComponent receives.

Decorating a CoolButton

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

- In order to create a CoolButton all we really need to do is draw the button as usual from the base class and then draw grey lines around the border to remove the button highlighting.

Decorating a CoolButton

```
public class CoolDecorator extends Decorator {
    boolean mouse_over;     //true when mose 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() {
                //set flag when mouse over
                public void mouseEntered(MouseEvent e) {
                        mouse_over = true;
                        thisComp.repaint();
                }
               //clear flag when mouse not over
               public void mouseExited(MouseEvent e) {
                        mouse_over = false;
                        thisComp.repaint();
                }
            });
    }
```

Decorating a CoolButton

```
    //paint the button
    public void paint(Graphics g) {
        super.paint(g);        //first draw the parent button
        //if the mouse is not over the button
        //erase the borders
        if (! mouse_over) {
            Dimension d = super.getSize();
            g.setColor(Color.lightGray);
            g.drawRect(0, 0, d.width-1, d.height-1);
            g.drawLine(d.width-2, 0, d.width-2, d.height-1);
            g.drawLine(0, d.height-2, d.width-2, d.height-2);
        }
    }
}
```

- There are two important features of the CoolDecorator
  - The **constructor which adds mouse adapters** so it can trap mouse movement.
  - The **overloaded paint** so that it can **erase the button borders**

Using a Decorator

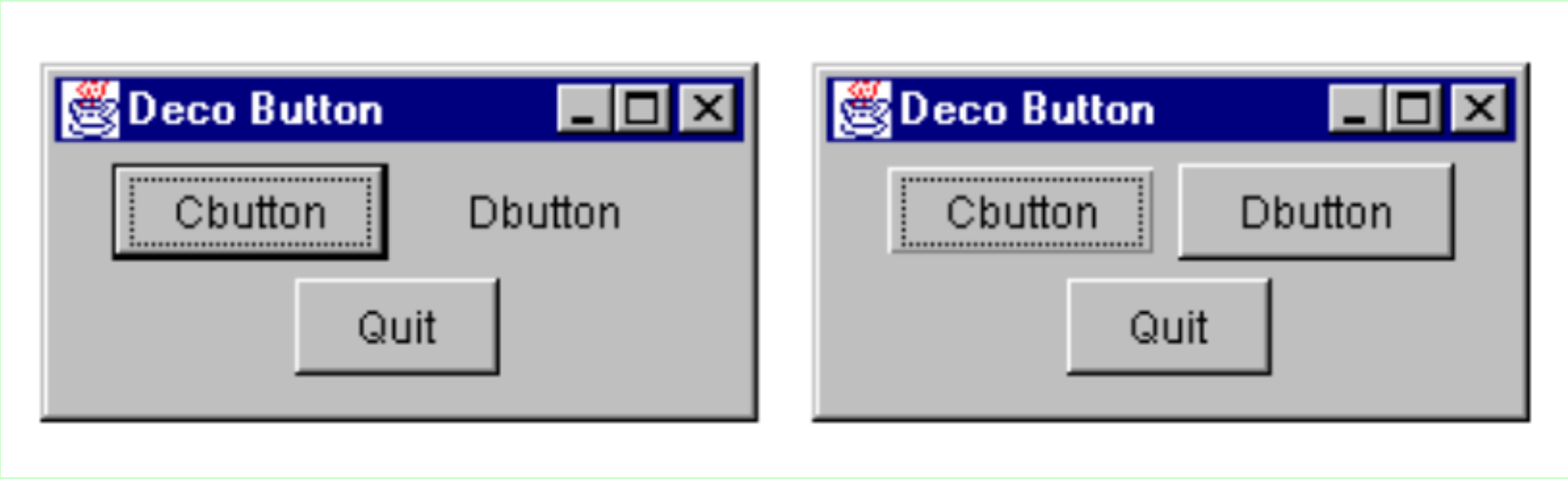
- Now that we have written a CoolDecorator class, **how do we use it?** We simply **create an instance of the CoolDecorator** and pass it the button that it to be decorated.
  We can do all of this in the **call to the constructor**.
  Lets consider a simple program that has two CoolButtons and an ordinary JButton.

- The next slide shows a code segment that illustrates the use of the CoolDecorator class.

Using a Decorator

```
super ("Deco Button");
JPanel jp = new JPanel();

getContentPane().add(jp);

jp.add( new CoolDecorator(
        CButton = new JButton("Cbutton")));

jp.add( new CoolDecorator(
        DButton = new JButton("Dbutton")));


jp.add(Quit = new JButton("Quit"));
Quit.addActionListener(this);
setSize(new Dimension(200,100));

setVisible(true);
Quit.requestFocus();
```

Using a Decorator



- In the first screen shot the mouse has not moved over any of the CoolButtons.

- In the second screen shot the mouse has moved over the Dbutton and the border is drawn.

Nonvisual Decorators

- Decorators are not **limited to objects** that enhance visual classes. You can add or modify the **methods** of any object in a similar fashion.

- Consider a program to **filter an email stream** to make it more readable. The user of the email program types a message in lower case. The sentences can be made **more readable** by making the **first letter in each sentence upper case**.

- We can achieve this by **decorating a FileInputStream** to read the data a transform it into a more readable form.

Nonvisual Decorators

- The FileFilter class simply extends the FileInputStream class and overloads its read method so that we can intercept the text and capitalize the first letter of each sentence.

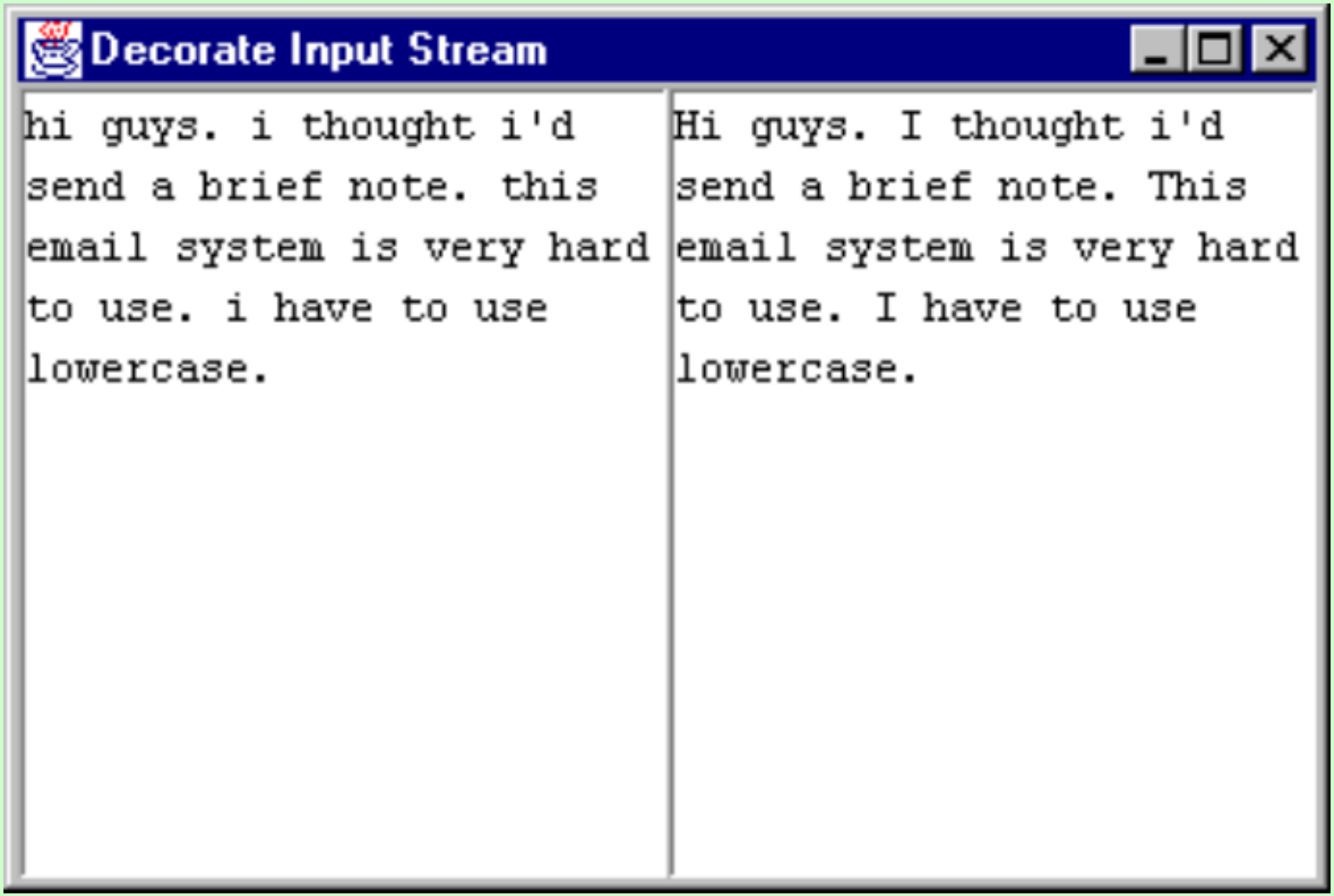
```
public class FileFilter extends FilterInputStream {

    static int MAX=1000;

    public FileFilter(InputStream f) {
        super(f);
    }
```

- The next slide show the overloaded read method of the FileFilter class.

Using a Decorator

```
public String readLine() {
    String s;
    int length = 0;

    byte b[] = new byte[MAX];
    try {
        length = super.read(b);
        s = new String(b, 0, length);
    } catch (IOException e) {
        s = "";
    }

    StringBuffer buf = new StringBuffer(s);
    boolean punctFound = true;

    for (int i=0; i < length; i++) {
        char ch = buf.charAt (i);
        if (punctFound && (ch != ' ') ) {
            ch = Character.toUpperCase (ch);
            buf.setCharAt (i, ch);
            punctFound = false;
        }
        if (ch == '.')
            punctFound = true;
    }
    s = buf.toString ();
    return s;
}
```

Using a Decorator



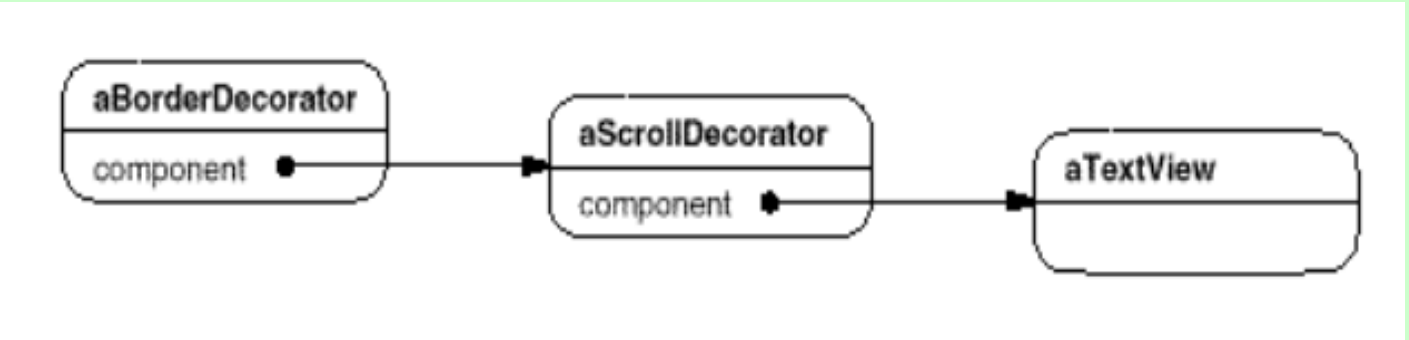
Decorator Consequences

- The decorator pattern provides a **more flexible** way to **add responsibilities to a class** than using inheritance, since it can add these responsibilities to **selected instances** of the class.

- It also allows the programmer to **customize a class** without subclassing it.

- The full source code for the examples given in this lecture are available on the student share folder and should be studied.

- You can combine decorators by nesting one decorator inside another, e.g.



Adapter Pattern

Adapter

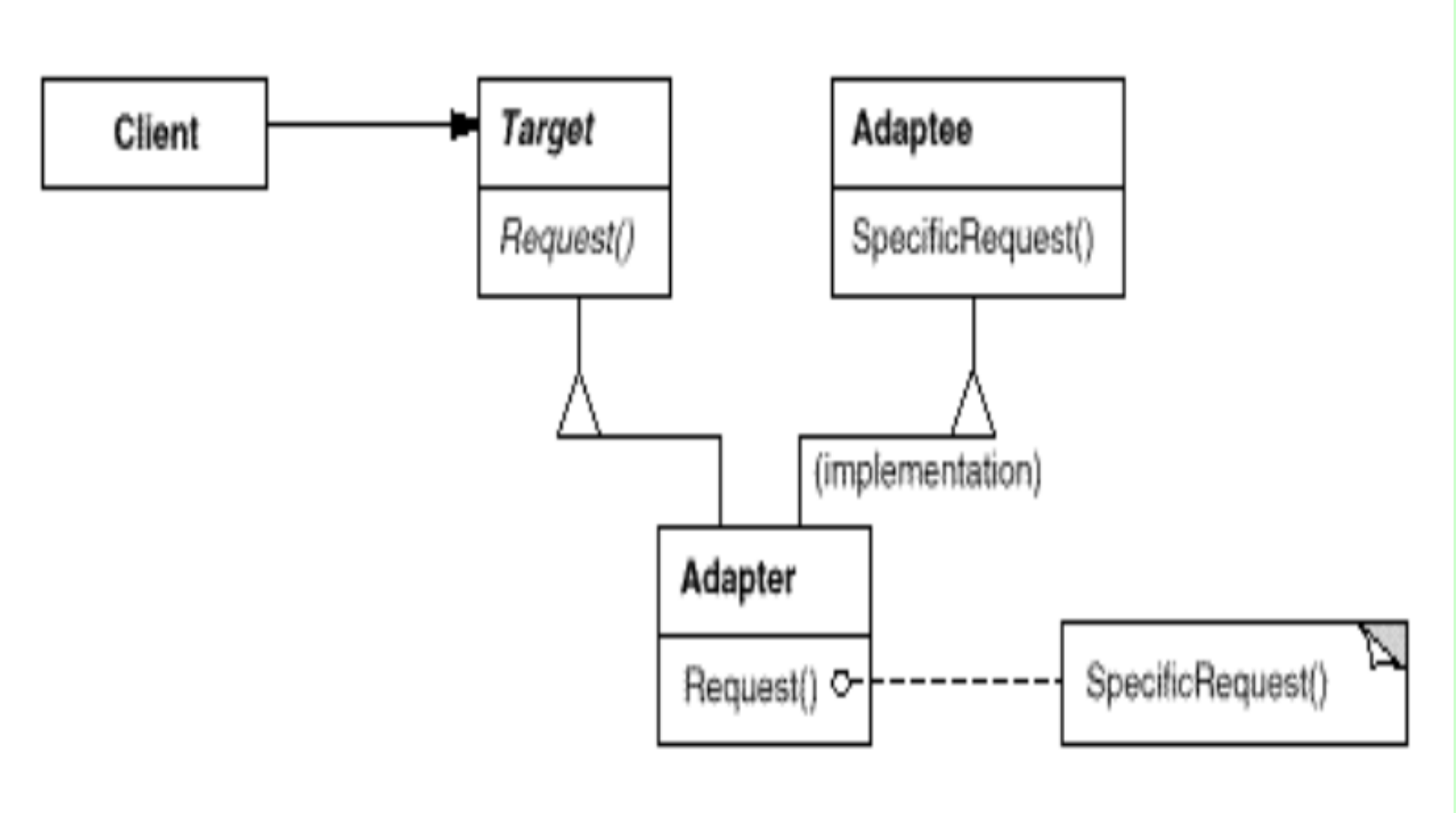
- **Intent:**
  Convert the **interface of a class** into **another interface that client expects**.
  Adapter lets classes work together that couldn't otherwise because of **incompatible interfaces.**

- Basically, this is saying that we need a way to create a **new interface** for **an object that does the right stuff** but has the **wrong interface.**

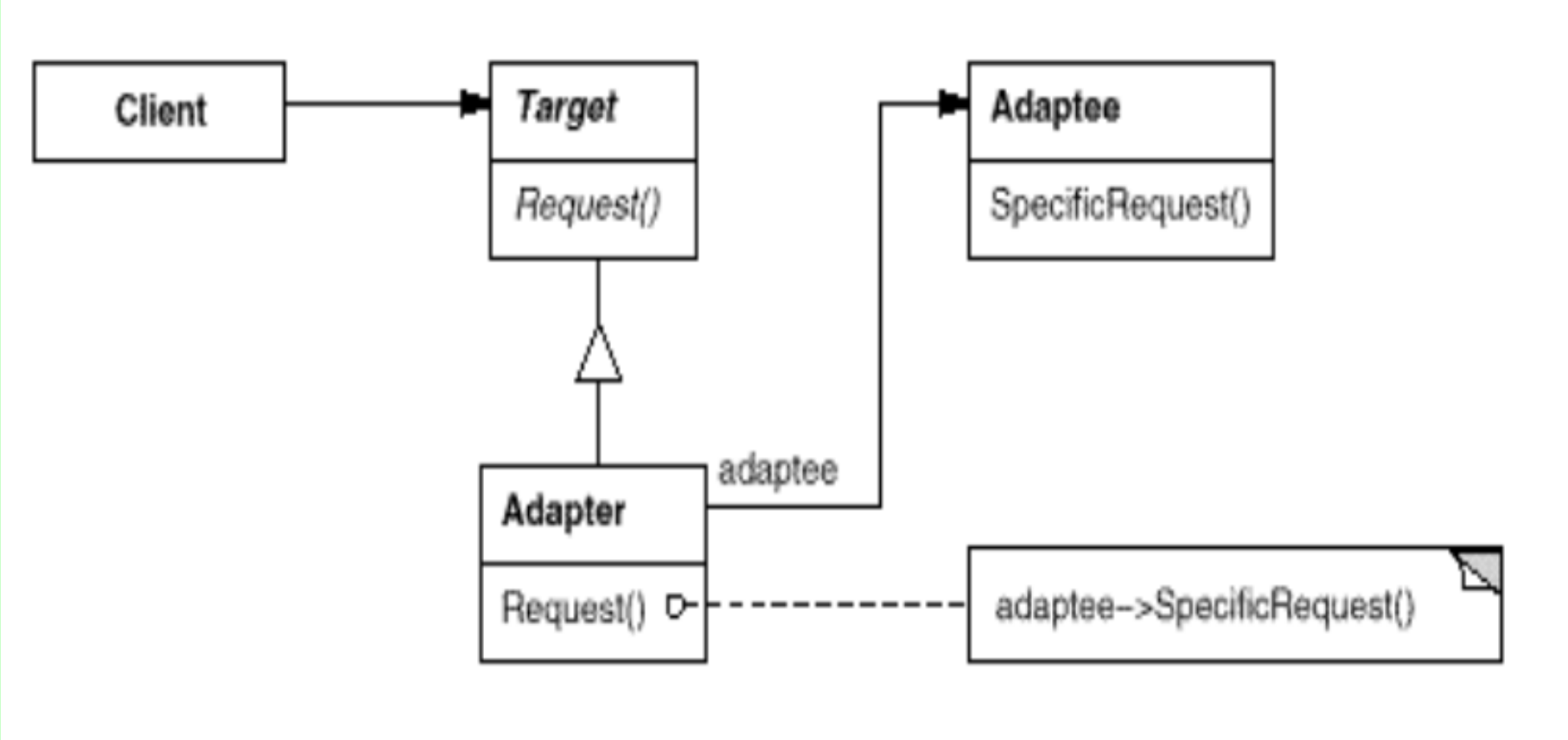
Adapter

- There are actually **two types** of adapter pattern:
  - **Object Adapter Pattern**
    The circle example is an example of this. It relies on one object (the adapting object) containing another (the adapted object)
  - **Class Adapter Pattern**
    Another way to implement the Adapter pattern is with **multiple inheritance**. In this case, it is called a Class Adapter pattern.

Adapter Pattern - Class



Adapter Pattern - Object



Adapter Pattern Participants

- **Target** (Shape)

Defines the domain-specific interface that Client uses.

- **Client** (DrawingEditor)

Collaborates with objects conforming to the Target interface.

- **Adaptee** (XXCircle)

Defines an existing interface that needs adapting.

- **Adapter** (Circle)

Adapts the interface of Adaptee to the Target interface.

Adapter Pattern Applicability

Use Adaptor when:

- you want to **use an existing class**, and **its interface** does **not match the one you need.**
- you want to create a reusable class that cooperates with unrelated or unforeseen classes, that is, **classes that don't necessarily have compatible interfaces.**
- *(object adapter only)* you need to use several existing subclasses, but it's impractical to adapt their interface by subclassing every one. An object adapter can adapt the interface of its parent class.

Learning the Adapter Pattern

- Let's look at an example of where it is useful.
- Let's say I've been given the following **requirements:**
  - Create **classes for points, lines and squares** that have the behaviour 'display'.
  - The client objects should not have to **know whether they actually have a point, line or square**. They just want to know that they have one of these shapes.
- In other words, I want to include these shapes in a **higher-level concept** that I will call a 'displayable shape'.

Learning the Adapter Pattern

- To accomplish this, I will create a Shape class and then derive from it the classes that represent points, lines and squares (see fig below)

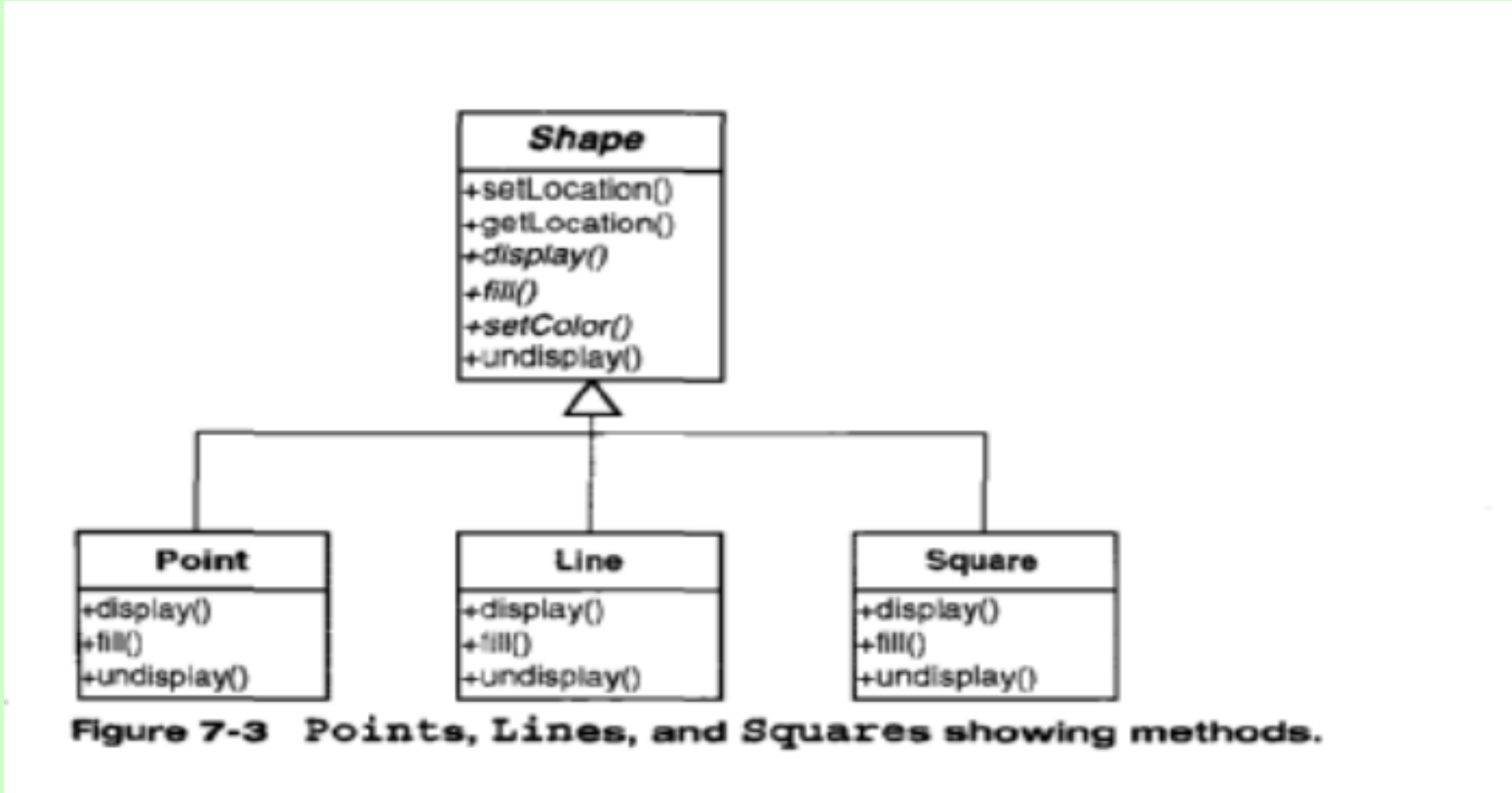


Figure 7-3 Points, Lines, and Squares showing methods.

Learning the Adapter Pattern

- Suppose I am asked to **implement a circle,** a new kind of Shape (remember requirements always change!).
- To do this, I will want to **create a new class Circle** – that implements the shape ‘circle’ and **derive it from the shape class** so that I can still get polymorphic behaviour.

Learning the Adapter Pattern

- Now, I am faced with the task of having to write the **display, fill and undisplay methods for Circle.** *That could be a pain!*
- Fortunately as I scout around, I discover that Jill down the hall has already written a class called **XXCircle** that deals with circles.
- Unfortunately, she has named the methods as follows:
  - displayIt
  - fillIt
  - undisplayIt

Learning the Adapter Pattern

- I cannot use XXCircle directly because I want to preserve polymorphic behaviour with Shape. There are two reasons for this:
  - I have **different names and parameter lists**
  - **I cannot derive it** – Not only must the names be the same, but the class must be derived from Shape as well.

Learning the Adapter Pattern

- Rather than change it I **adapt it.**
- I can make a **new class** that does **derive from Shape** and therefore implement's Shape's interface but **avoids rewriting** the circle implementation in XXCircle.
  - Class Circle derives from Shape
  - Circle contains XXCircle
  - Circle passes requests made to the Circle object on through to the XXCircle object

Learning the Adapter Pattern

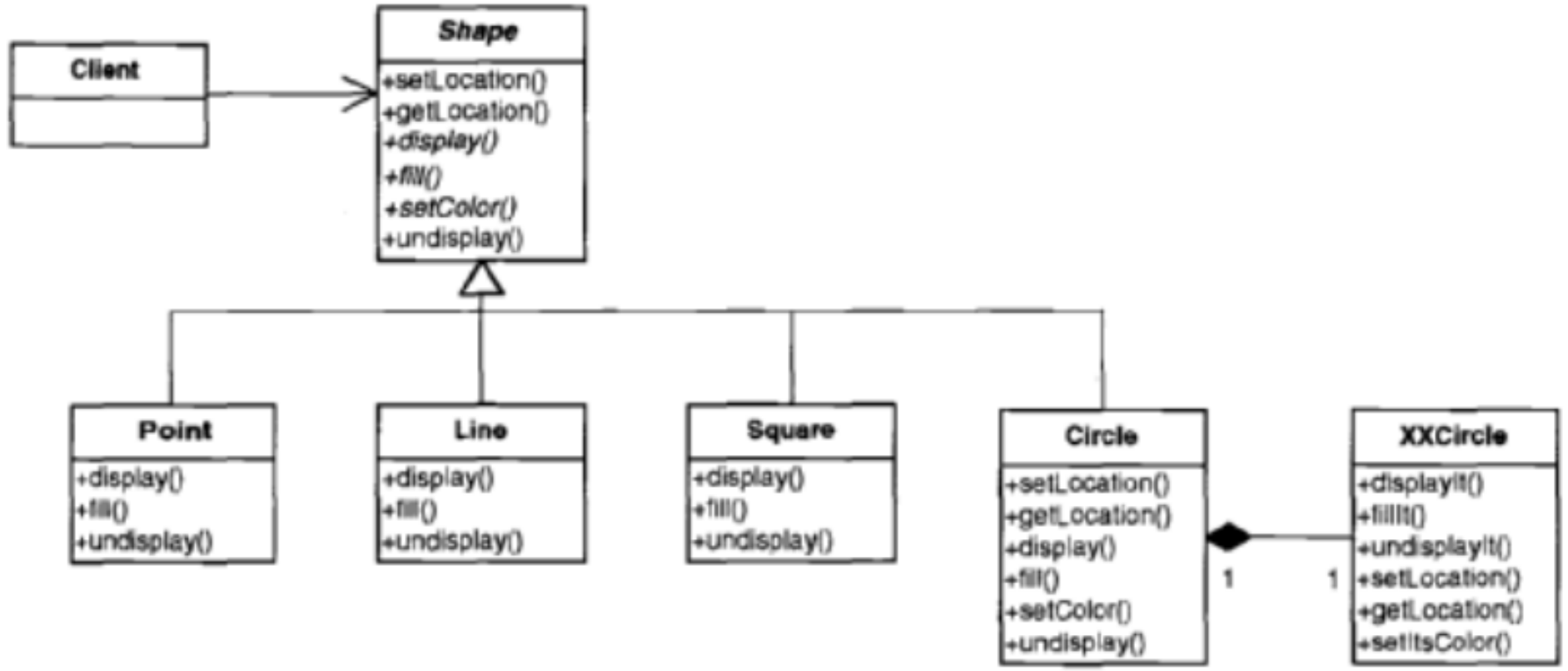


Figure 7-5 The Adapter pattern: `Circle` "wraps" `XXCircle`.

Learning the Adapter Pattern

An example of wrapping is shown in Example 7-1.

Example 7-1 Java Code Fragments: Implementing the Adapter Pattern

```
class Circle extends Shape {
   ...
   private XXCircle pxc;
   ...
   public Circle () {
     pxc= new XXCircle();
   }

   void public display() {
      pxc.displayIt();
   }
}
```

Learning the Adapter Pattern

- Using the adapter pattern allowed me to **continue using polymorphism** with Shape.
- ie., the client objects of shape **do not know** what types of shape are actually present.

The Adapter Pattern

The Adapter Pattern: Key Features

Intent	Match an existing object beyond your control to a particular interface.
Problem	A system has the right data and behavior but the wrong interface. Typically used when you have to make something a derivative of an abstract class we are defining or already have.
Solution	The Adapter provides a wrapper with the desired interface.
Participants and Collaborators	The **`Adapter`** adapts the interface of an **`Adaptee`** to match that of the **`Adapter`**'s **`Target`** (the class it derives from). This allows the **`Client`** to use the **`Adaptee`** as if it were a type of **`Target`**.
Consequences	The Adapter pattern allows for preexisting objects to fit into new class structures without being limited by their interfaces.
Implementation	Contain the existing class in another class. Have the containing class match the required interface and call the methods of the contained class.
GoF Reference	Pages 139–150.

Exercise 1 – Decorator Pattern

- Examine the classes in **code listing (on next page)**, then answer the following questions:
  - Create a new Decorator class called CrazyDecorator.
    The CrazyDecorator class must change a JComponent's background colour to red when the mouse enters the JComponent.
    The JComponent's background colour must be reset to it's original colour when the mouse exits the JComponent.
  - Create a simple test program called DecoratorTester which will decorate a JButton with the SlashDecorator **AND** the CrazyDecorator.

Exercise 1 – Decorator Pattern

Code Listing 2

Decorater.java

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

SlashDecorator.java

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.text.*;
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.border.*;
public class SlashDecorator extends Decorator {
int x1, y1, w1, h1;
public SlashDecorator(JComponent c) {
super(c);
}
public void setBounds(int x, int y, int w, int h) {
x1 = x; y1= y;
w1 = w; h1 = h;
super.setBounds(x, y, w, h);
}
public void paint(Graphics g) {
super.paint(g);
g.setColor(Color.red);
g.drawLine(0, 0, w1, h1);
}
}
```