CS 635 Advanced Object-Oriented Design & Programming
Spring Semester, 2002
Doc 15 Decorator, Chain of Responsibility, OO Recursion

Contents

References

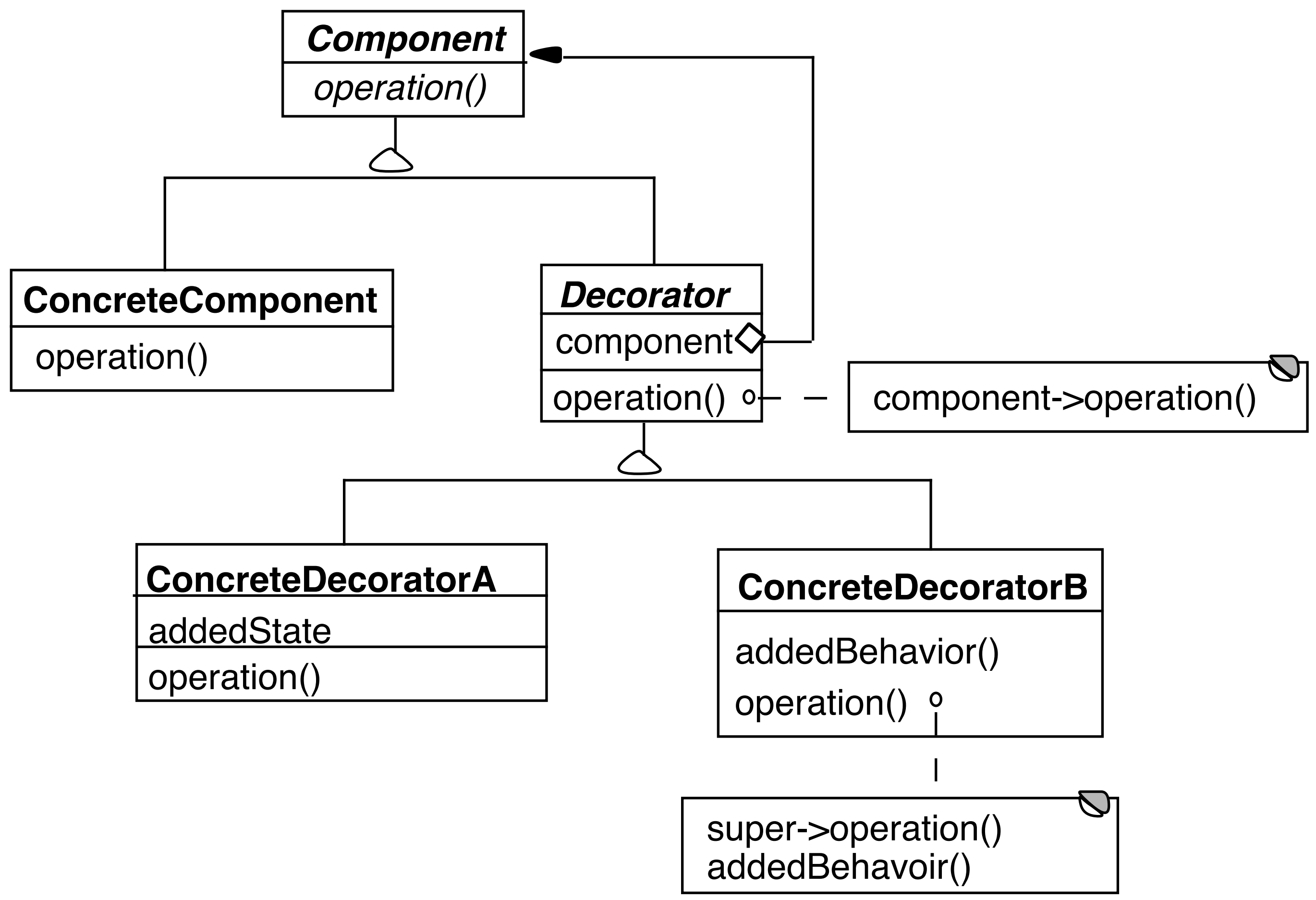
Design Patterns: Elements of Reusable Object-Oriented Software, Gamma, Helm, Johnson, Vlissides, Addison-Wesley, 1995, pp. 175-184, 223-232

The Design Patterns Smalltalk Companion, Alpert, Brown, Woolf, 1998, pp. 161-178, 225-244

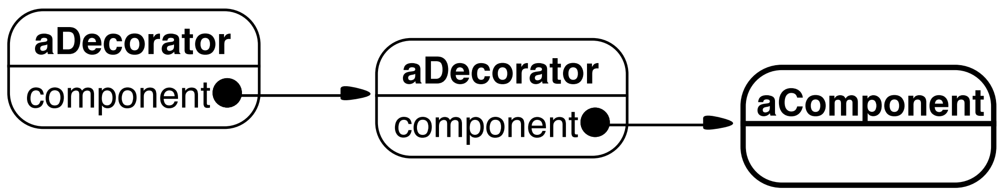
Decorator

Changing the Skin of an Object

Class Structure



Runtime Structure



Motivation - Text Views

A text view has the following features:

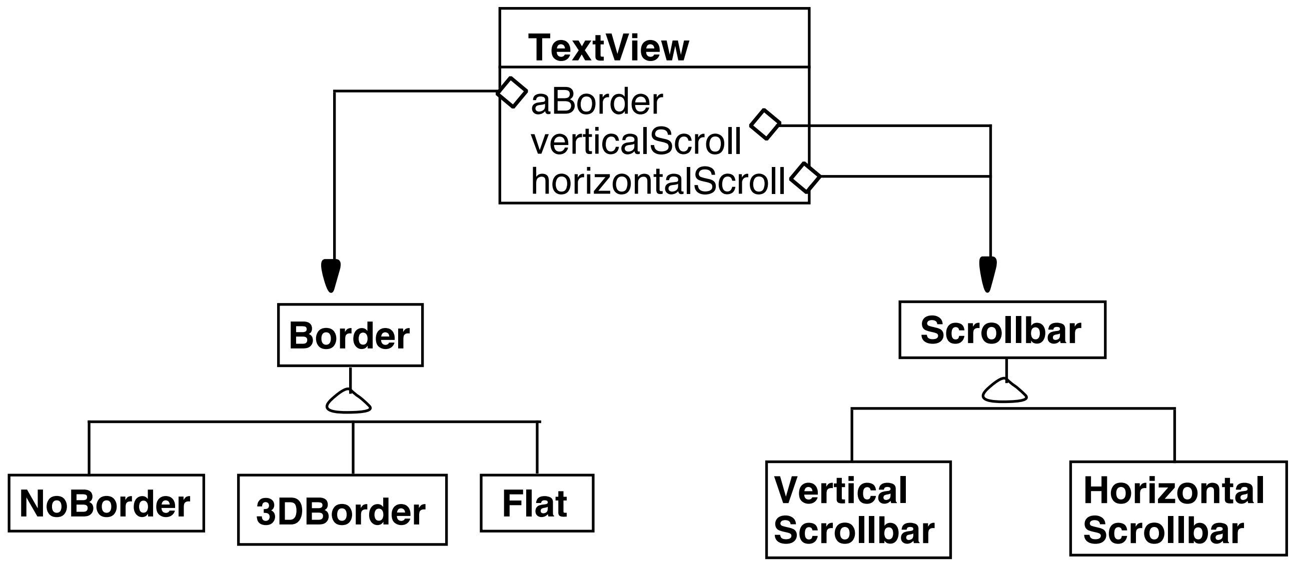
side scroll bar
Bottom scroll bar
3D border
Flat border

This gives 12 different options:

TextView
TextViewWithNoBorder&SideScrollbar
TextViewWithNoBorder&BottomScrollbar
TextViewWithNoBorder&Bottom&SideScrollbar
TextViewWith3DBorder
TextViewWith3DBorder&SideScrollbar
TextViewWith3DBorder&BottomScrollbar
TextViewWith3DBorder&Bottom&SideScrollbar
TextViewWithFlatBorder
TextViewWithFlatBorder&SideScrollbar
TextViewWithFlatBorder&BottomScrollbar
TextViewWithFlatBorder&Bottom&SideScrollbar

How to implement?

Solution 1 - Use Object Composition



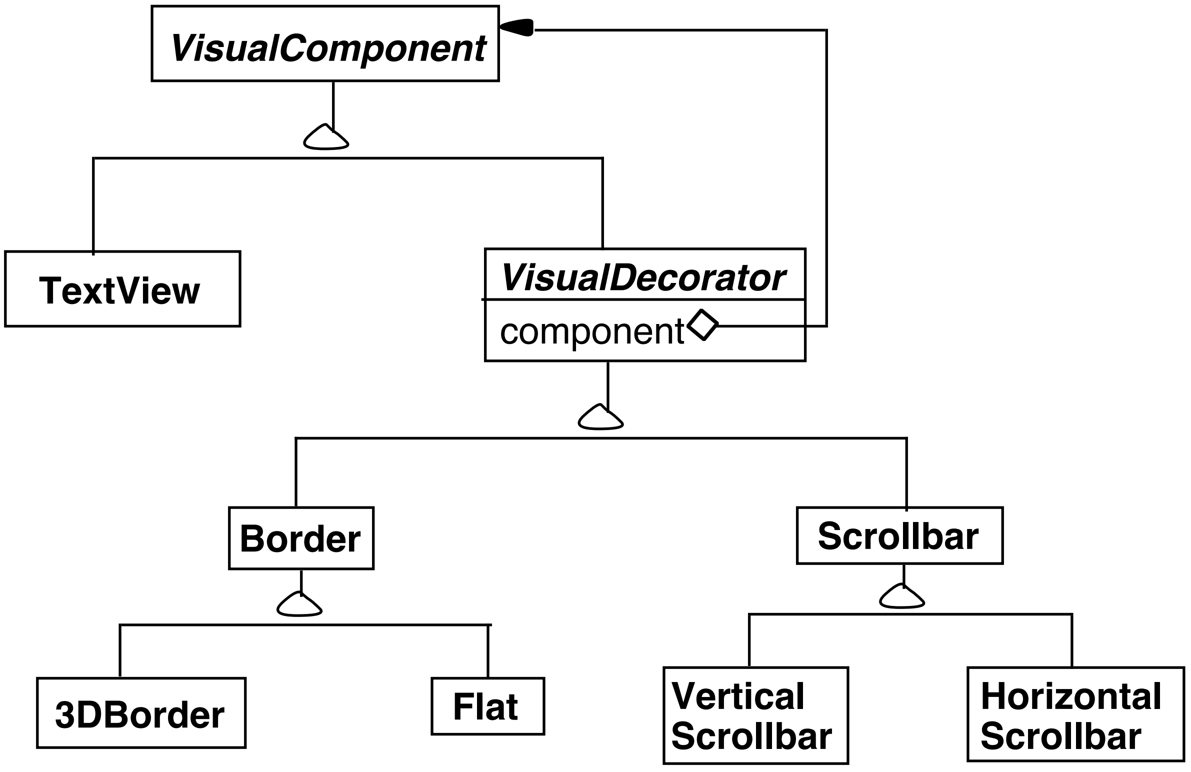
```
class TextView {
   Border myBorder;
   ScrollBar verticalBar;
   ScrollBar horizontalBar;

   public  void draw() {
      myBorder.draw();
      verticalBar.draw();
      horizontalBar.draw();
      code to draw self
   }
   etc.
}
```

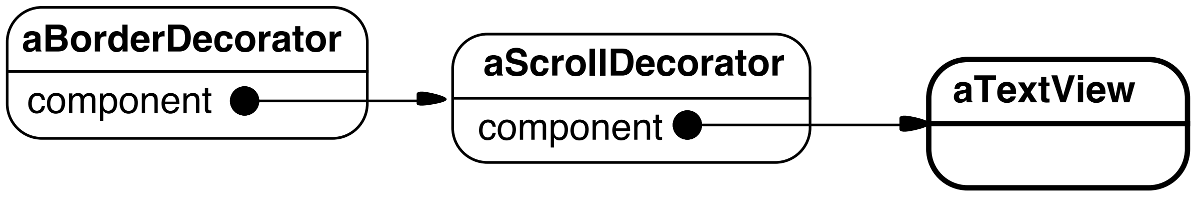
But TextView knows about all the variations!
New type of variations require changing TextView
 (and any other type of view we have)

Solution 2 - Use Decorator
Object Composition Inside out
Change the skin of an object not it guts

TextView has no borders or scrollbars!
Add borders and scrollbars on top of a TextView



Runtime Structure



Applicability

Use Decorator:

- To add responsibilities to individual objects dynamically and transparently
- For responsibilities that can be withdrawn
- When subclassing is impractical - may lead to too many subclasses

Commonly used in basic system frameworks

Windows, streams, fonts

Consequences

More flexible than static inheritance

Avoids feature laden classes high up in hierarchy

Lots of little objects

A decorator and its components are not identical

So checking object identification can cause problems

```
if ( aComponent instanceof TextView ) blah
```

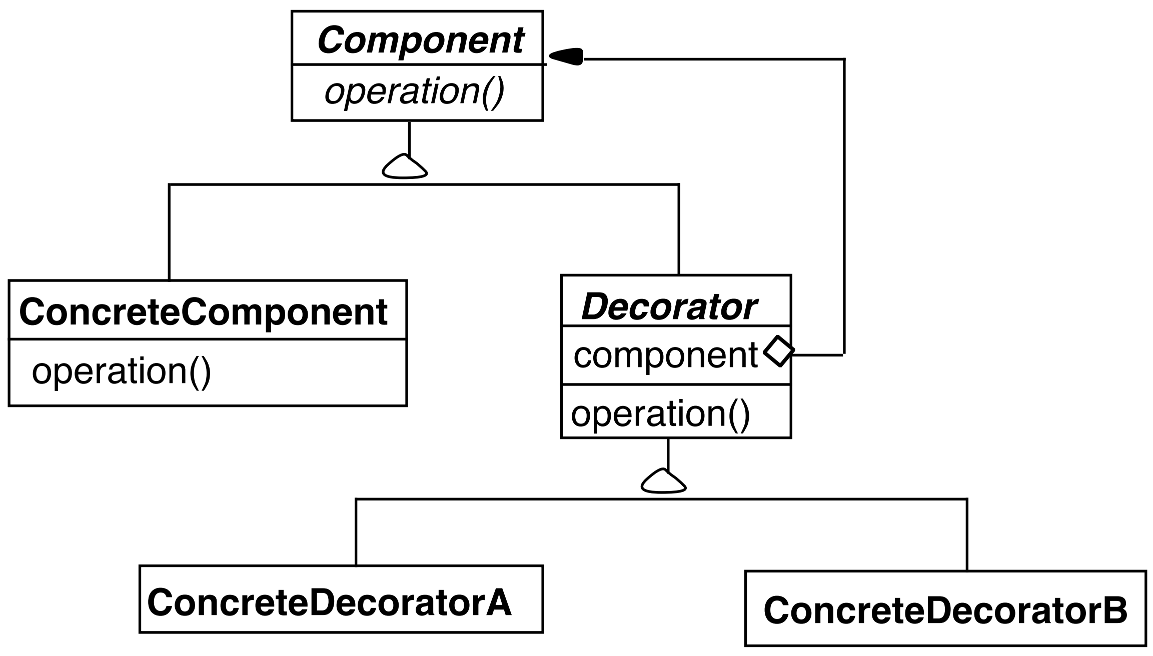
Implementation Issues

Keep Decorators lightweight

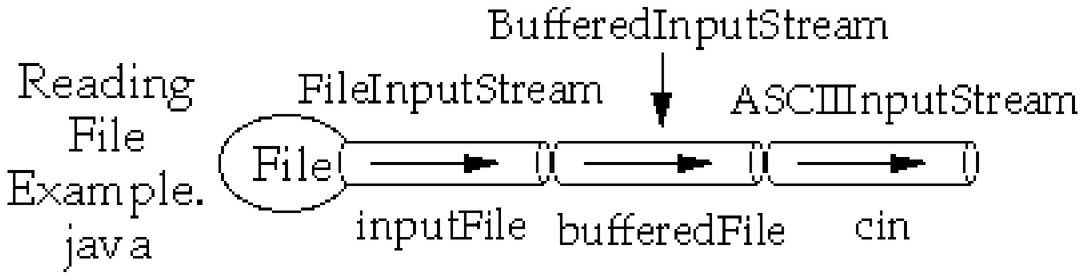
Don't put data members in VisualComponent

Have Decorator forward all component operations

Three ways to forward messages
- Simple forward
- Extended forward
- Override



Examples
Java Streams



```
import java.io.*;
import sdsu.io.*;
class  ReadingFileExample
  {
  public  static  void  main( String  args[]  ) throws Exception
    {
    FileInputStream inputFile;
    BufferedInputStream bufferedFile;
    ASCIIInputStream  cin;

    inputFile = new FileInputStream( "ReadingFileExample.java" );
    bufferedFile = new BufferedInputStream( inputFile );
    cin = new ASCIIInputStream( bufferedFile );

    System.out.println(  cin.readWord()  );

    for  ( int  k = 1 ;  k  <  4;  k++ )
      System.out.println(  cin.readLine()  );
    }
  }
```

Insurance

Insurance policies have payment caps for claims

Sometimes the people with the same policy will have different caps

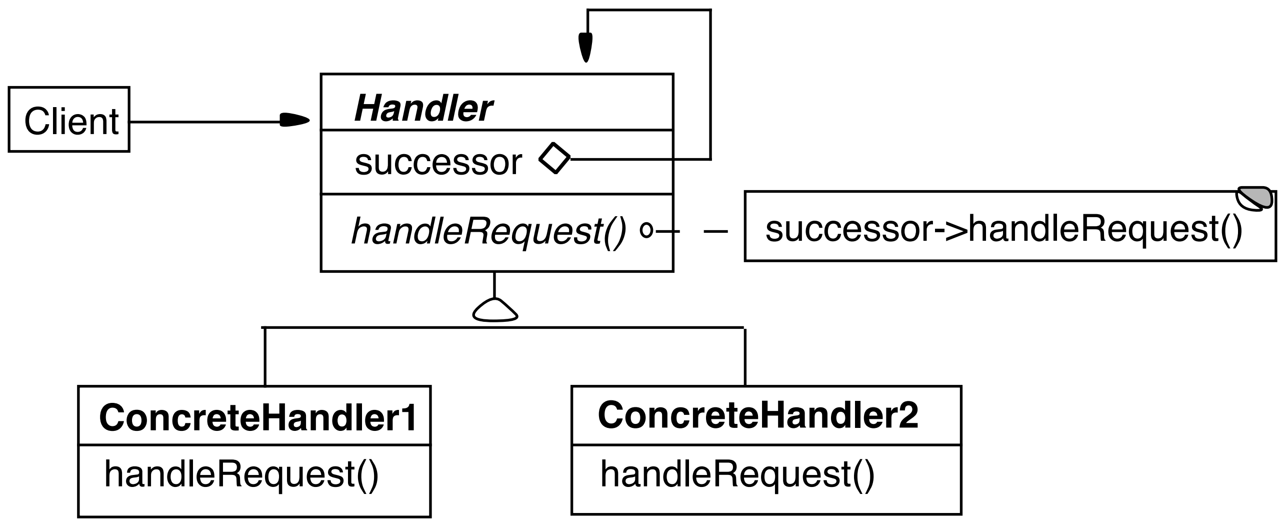
A decorator can be used to provide different caps on the same policy object

Similarly for deductibles & copayments

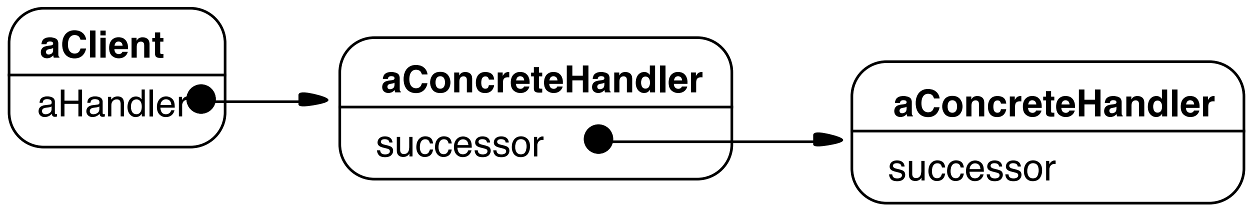
Chain of Responsibility
Intent

Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

Class Structure



Sample Object Structure



Participants

Handler

- Defines the interface for handling the requests
- May implement the successor link

ConcreteHandler

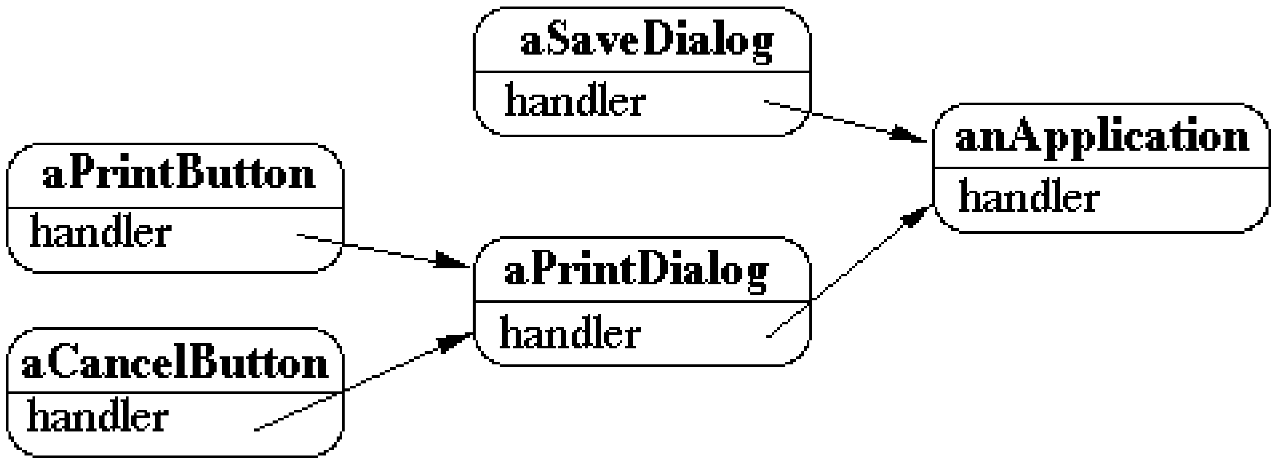
- Handles requests it is responsible for
- Can access its successor
- Handles the request if it can do so, otherwise it forwards the request to its successor

Consequences

- Reduced coupling
- Added flexibility in assigning responsibilities to objects
- Not guaranteed that request will be handled

Motivation

Context Help System



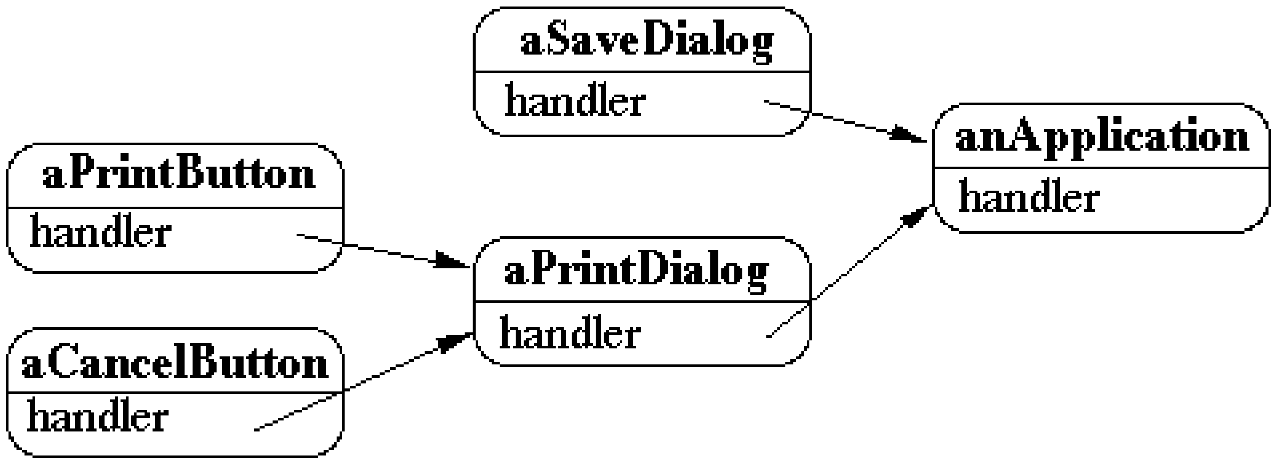
When to Use

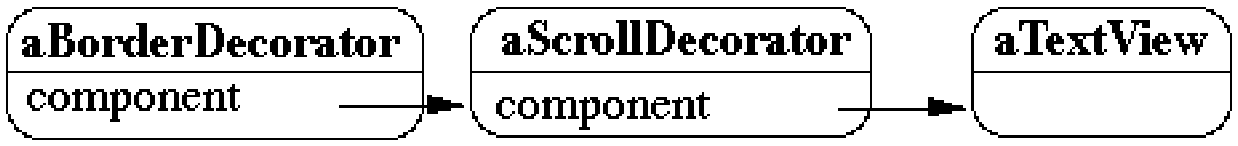
When more than on object may handle a request, and the handler isn't known a priori

When you want to issue a request to one of several objects without specifying the receiver explicitly

When the set of objects that can handle a request should be specified dynamically

How does this differ from Decorator?



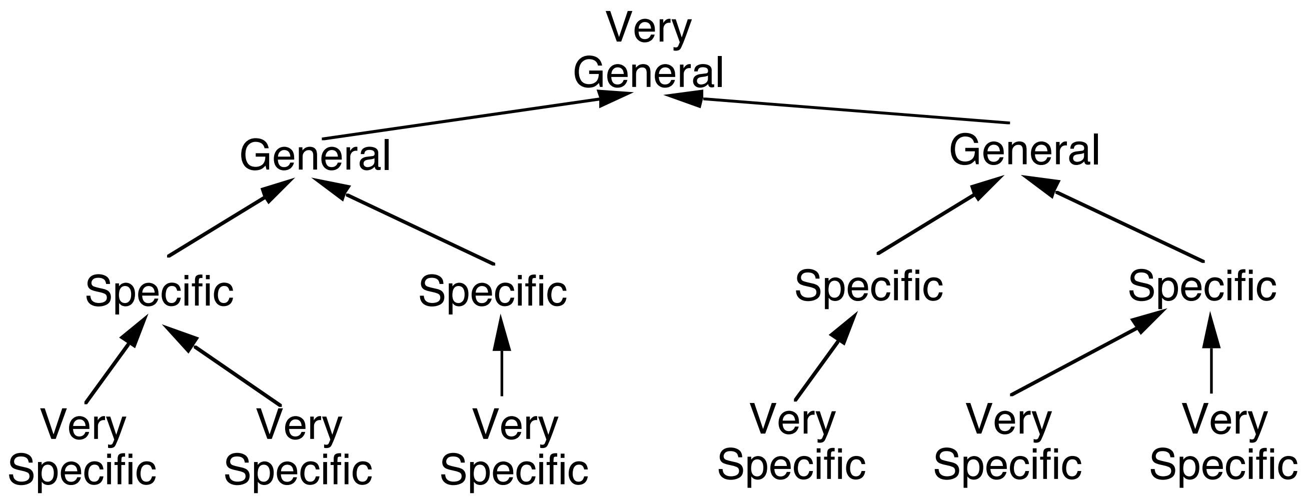


Chain of Command

Like the military

A request is made

It goes up the chain of command until someone has the authority to answer the request



Implementation Issues
The successor chain

- Use existing links in the program

  The concrete handlers may already have pointers to their successors, so just use them

- Define new links

  Give each handler a link to its successor

Representing Requests

- Each request can be a hard-coded

```
abstract class HardCodedHandler
  {
  private HardCodedHandler successor;

  public HardCodedHandler( HardCodedHandler aSuccessor)
    { successor = aSuccessor; }

  public void handleOpen()
    { successor.handleOpen(); }

  public void handleClose()
    { successor.handleClose(); }

  public void handleNew( String fileName)
    { successor.handleClose( fileName ); }
  }
```

Representing Requests

- A single method implements all requests

```
abstract class SingleHandler {
   private SingleHandler successor;

   public SingleHandler( SingleHandler aSuccessor) {
      successor = aSuccessor;
   }

   public void handle( String request) {
      successor.handle( request );
   }
}

class ConcreteOpenHandler extends SingleHandler {
   public void handle( String request) {
      switch ( request ) {
         case "Open" : do the right thing;
         case "Close" : more right things;
         case "New" : even more right things;
         default: successor.handle( request );
      }
   }
}
```

Representing Requests

- Single handle method with Request Object for parameters

```
abstract class SingleHandler {
   private SingleHandler successor;

   public SingleHandler( SingleHandler aSuccessor)
      {successor = aSuccessor; }

   public void handle( Request data)
      { successor.handle( data ); }
}

class ConcreteOpenHandler extends SingleHandler {
   public void handle( Open data)
      { // handle the open here }
}

class Request {
   private int size;
   private String name;
   public Request( int mySize, String myName)
      { size = mySize;  name = myName; }

   public int size() { return size; }
   public String name() { return name;}
}

class Open extends Request
   {// add Open specific stuff here}

class Close extends Request
   { // add Close specific stuff here}
```

Object-Oriented Recursion
Recursive Delegation

A method polymorphically sends its message to a different receiver

Eventually a method is called that performs the task

The recursion then unwinds back to the original message send

Example

```
class BinarySearchTree {
   Node root

   boolean containsKey( Object key ) {
      return root.containsKey(key);
   }

   String toString() {
      return "Tree( " + root.toString() + ")";
   }
   blah
}
```

Example Continued

```
class BinaryNode implements Node {
   Node left;
   Node right;
   Object key;
   Object value;

   boolean containsKey( Object key ) {
      if this.key == key
         return true;
      if this.key < key
         return right.containsKey(key);
      if this.key > key
         return left.containsKey( key);
   }

   String toString() {
      return "( " + left.toString() + key + right.toString() + ")";
   }
   blah
}

class NullNode implements Node {

   boolean containsKey( Object key ) {
      return false;
   }

   String toString() {
      return " ";
   }
   blah
}
```