CS 635 Advanced Object-Oriented Design & Programming
Spring Semester, 2005
Doc 6 Decorator, Proxy & Adapter
Contents

References

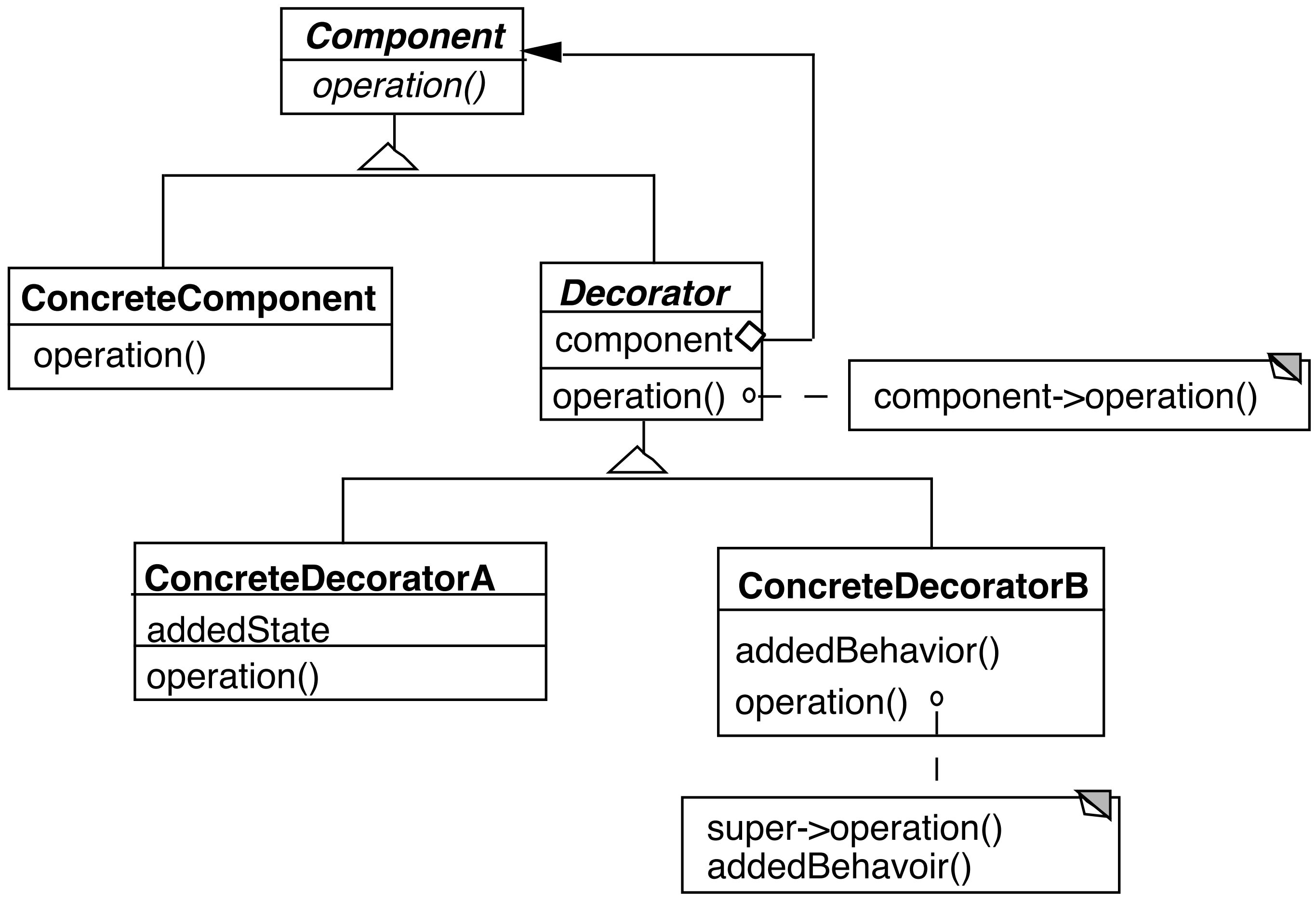
Design Patterns: Elements of Resuable Object-Oriented Software, Gamma, Helm, Johnson, Vlissides, Addison Wesley, 1995, pp. 175-185, 207-217, 139-150

The Design Patterns Smalltalk Companion, Alpert, Brown, Woolf, Addision-Wesley, 1998, pp. 161-178, 213-221, 105-120

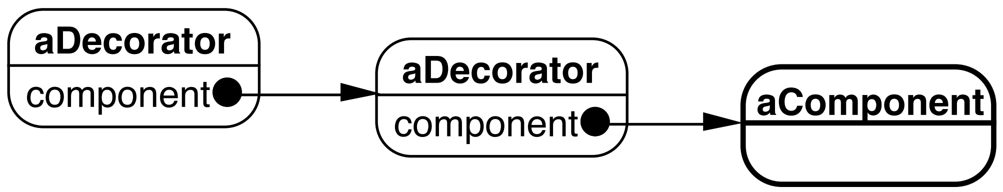
Decorator

Changing the Skin of an Object

Class Structure



Runtime Structure



Motivation - Text Views

A text view has the following features:

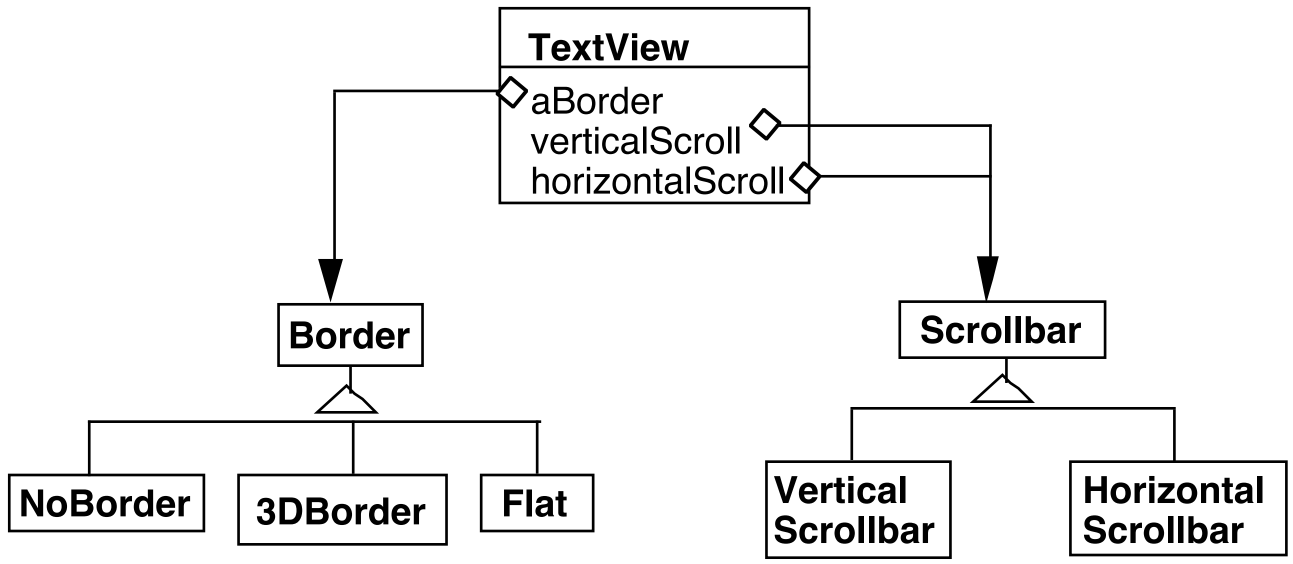
side scroll bar
Bottom scroll bar
3D border
Flat border

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



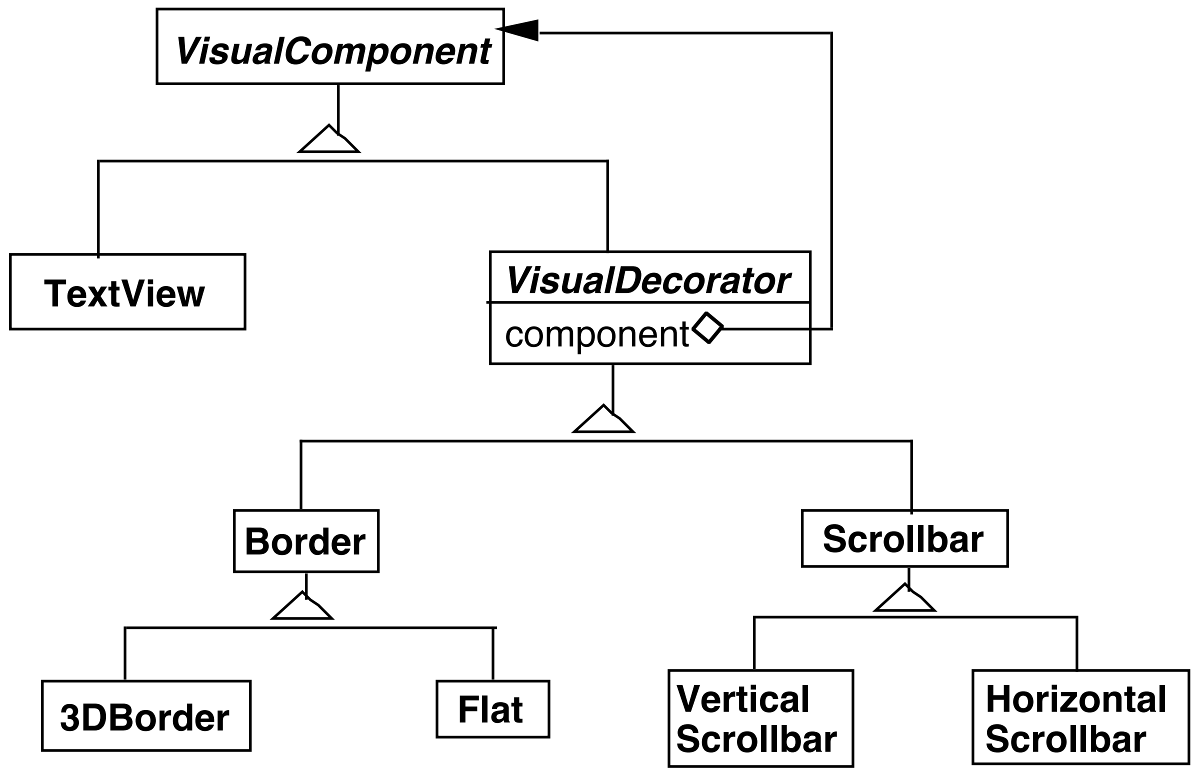
```
class TextView {
   Border myBorder;
   ScrollBar verticalBar;
   ScrollBar horizontalBar;

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

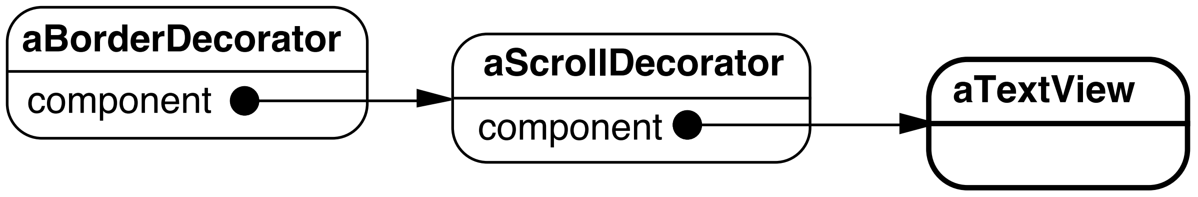
But TextView knows about all the variations!
New type of variations require changing TextView
(and any other type of view we have)

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
```

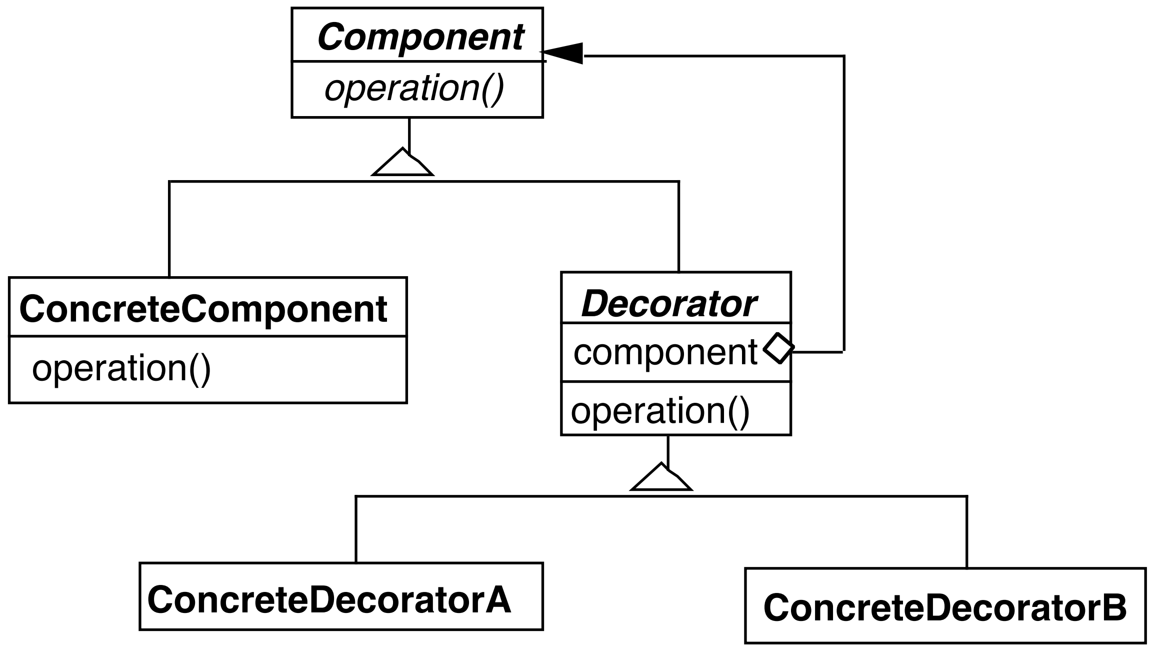
Implementation Issues

Keep Decorators lightweight

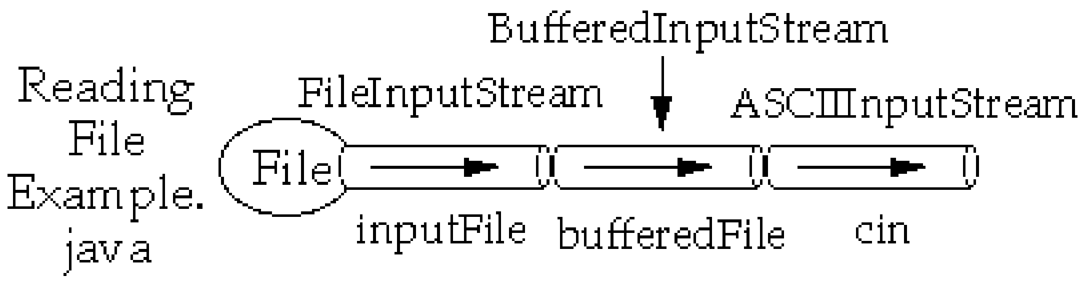
Don't put data members in VisualComponent

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

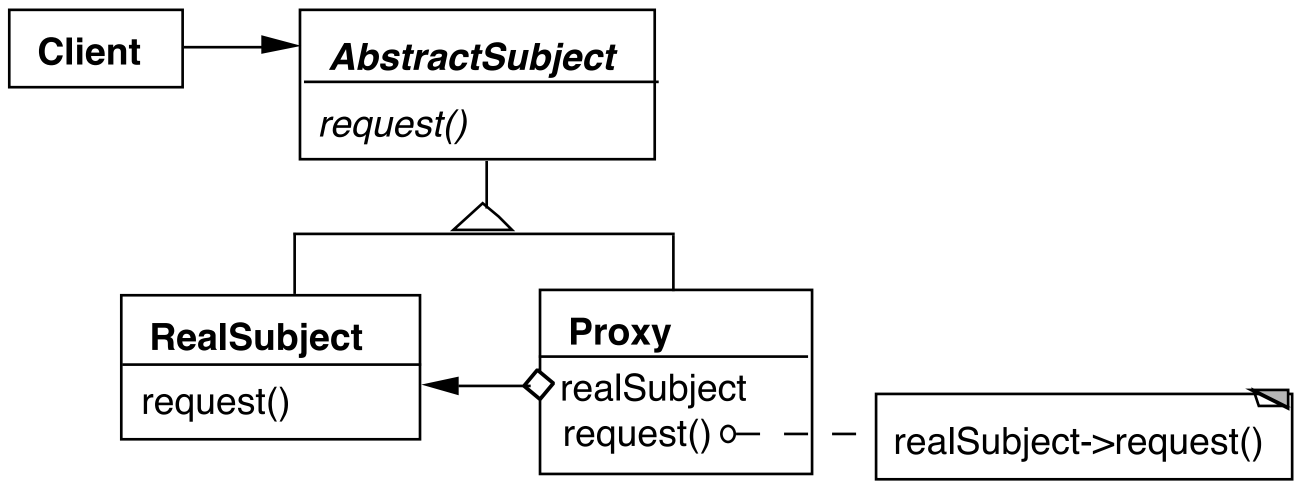
A decorator can be used to provide different caps on the same policy object

Similarly for deductibles & copayments

Proxy

proxy n. pl prox-ies The agency for a person who acts as a substitute for another person, authority to act for another

Structure



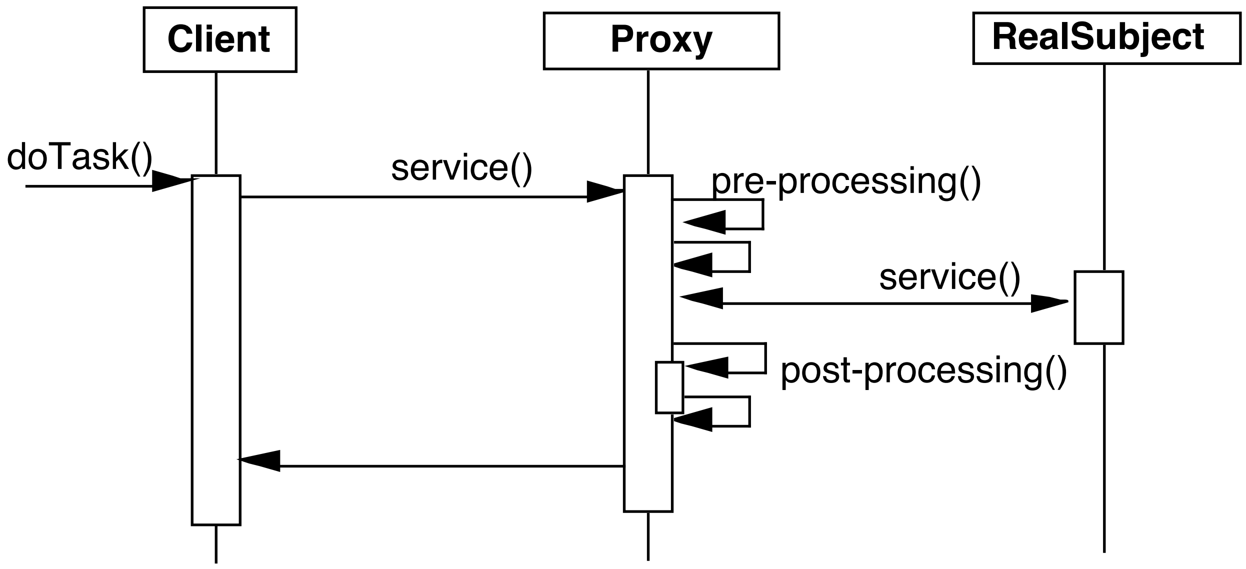
The Pattern

The proxy has the same interface as the original object

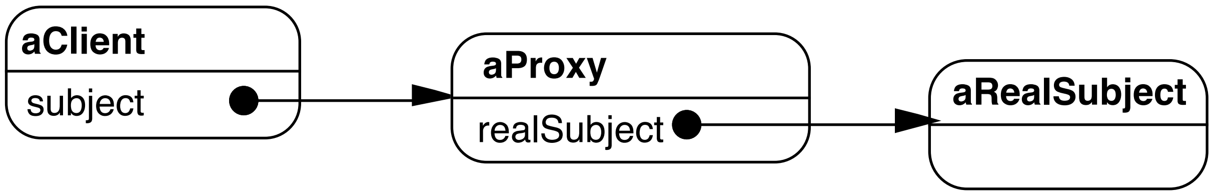
Use common interface (or abstract class) for both the proxy and original object

Proxy contains a reference to original object, so proxy can forward requests to the original object

Dynamics



Runtime Objects



Sample Proxy

```
public class Proxy
   {
   Foo target;

   public float bar(int size )
      {
      preprocess here
      float answer =  target.bar( size);
      postProcess here
      return answer;
      }

   other methods as needed
   }
```

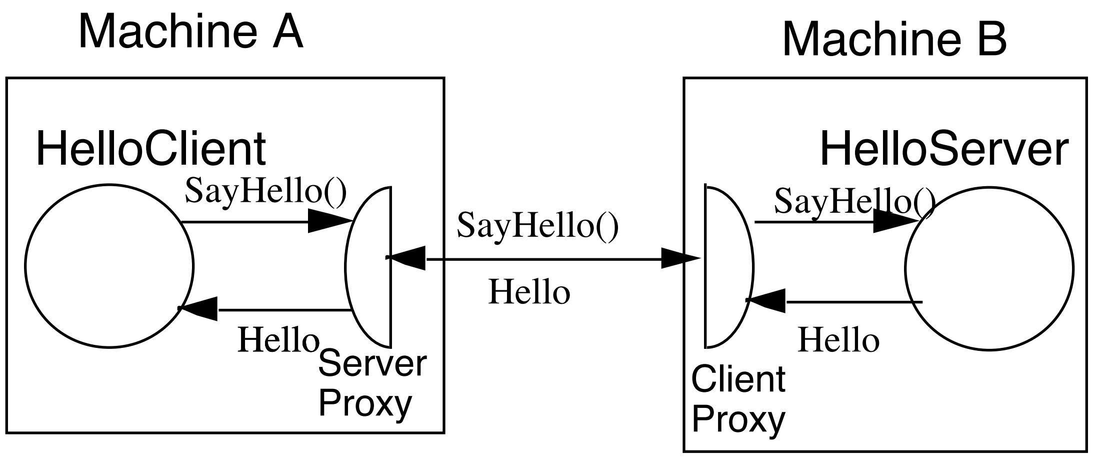
Preprocessing & post-processing depend on purpose of the proxy

Reasons for Object Proxies
Remote Proxy

The actual object is on a remote machine (remote address space)

Hide real details of accessing the object

Used in CORBA, Java RMI



```
public class HelloClient  {
   public static void main(String args[])  {
      try  {
         String server = getHelloHostAddress( args);
         Hello proxy = (Hello) Naming.lookup( server );
         String message = proxy.sayHello();
         System.out.println( message );
      }
      catch ( Exception error)
         {  error.printStackTrace(); }
   }
}
```

Reasons for Object Proxies Continued

Virtual Proxy

- Creates/accesses expensive objects on demand

- You may wish to delay creating an expensive object until it is really accessed

- It may be too expensive to keep entire state of the object in memory at one time

Protection Proxy

- Provides different objects different level of access to original object

Cache Proxy (Server Proxy)

- Multiple local clients can share results from expensive operations: remote accesses or long computations

Firewall Proxy

- Protect local clients from outside world

Synchronization Proxy

- Synchronize multiple accesses to real subject

```
public class Table {
   public Object elementAt( int row, int column ){ blah     }

   public void setElementAt(Object element, int row, int column )
      { blah}
}

public class RowLockTable {
   Table realTable;
   Integer[] locks;

   public RowLockTable( Table toLock) {
      realTable = toLock;
      locks = new String[ toLock.numberOfRows() ];
      for (int row = 0; row< toLock.numberOfRows(); row++ )
         locks[row] = new Integer(row);
      }

   public Object elementAt( int row, int column ) {
      synchronized ( locks[row] ) {
         return realTable.elementAt( row, column);
      }
   }

   public void setElementAt(Object element, int row, int column ){
      synchronized ( locks[row] )  {
         return realTable.setElementAt(element, row, column);
      }
   }
}
```

Counting Proxy

Delete original object when there are no references to it

Prevent accidental deletion of real subject

Collect usage statistics

Sample use is making C++ pointer safe

Smalltalk Proxy Tricks

When an object is sent a message

The object's class and the object's class’s superclasses are searched for the method

If the method is not found the object is sent the message:

```
doesNotUnderstand:
```

This method in Object raises an exception

Prototyping of a Proxy

One can use doesNotUnderstand: to implement a pluggable proxy

Example

```
Object subclass: #Proxy
  instanceVariableNames: 'target '
  classVariableNames: ''
  poolDictionaries: ''
  category: 'Whitney-Examples'
```

Class Method

```
on: anObject
  ^super new target: anObject
```

Instance Methods

```
doesNotUnderstand: aMessage
  ^target
    perform: aMessage selector
    withArguments: aMessage arguments

target: anObject
  target := anObject
```

Examples of Using the Proxy

```
| wrapper |
wrapper := Proxy on: Transcript.
wrapper open.
wrapper show: 'Hi mom'.

| wrapper |
wrapper := Proxy on: 3.
wrapper + 5.

| wrapper |
wrapper := Proxy on: 'Hi '.
wrapper , ' mom'.
```

Why just for Prototyping

doesNotUnderstand:

- Can be hard to debug
- Is slower than regular message send

Adapter
Motivating Adapter
Java CGI & Servlets

Both Java CGI and servlets are used for server-side processing of certain HTML requests, like processing HTML forms

Servlets have greater functionality and are faster, but require special Web servers or servers with special extensions

To help write Java CGI programs there is class sdsu.net.CGI

It would be useful in moving code between servers to avoid having to rewrite the code

One Problem

One issue is access to the CGI environment variables

There are about 20 common CGI environment variables

In servlets one has an HttpRequest class that has a getX() method for each CGI environment variable

sdsu.net.CGI class returns a hash table with one entry per CGI environment variable

Solution

We can write a wrapper around HttpRequest to make it act like a hash table

The Wrapper or Adapter

```
class CGIAdapter extends Hashtable
   {
   Hashtable CGIvariables = new Hashtable( 20);

   public CGIAdapter( HttpRequest CGIEnvironment )
      {
      CGIvariables.put( "AUTH_TYPE" ,
                  CGIEnvironment.getAuthType());

      CGIvariables.put( "REMOTE_USER" ,
                  CGIEnvironment.getRemoteUser());

      etc.
      }

   public Object get(Object key)
      {
      return CGIvariables.get( key );
      }

      etc.
   }
```

Going the other Direction

Adapting servlet code to normal CGI requires extracting the CGI environment variables out of the hash table and putting them into an object that implements the public interface of the HttpRequest class

```
class HTTPAdapter extends HttpRequest
   {
   Hashtable CGIvariables;

   public HTTPAdapter( Hashtable CGIEnvironment )
      {
      CGIvariables = CGIEnvironment;
      }

   public String getAuthType()
      {
      return (String) CGIvariables.get( "AUTH_TYPE" );
      }

   public String getRemoteUser()
      {
      return (String) CGIvariables.get( "REMOTE_USER" );
      }

      etc.
   }
```

Adapter

The adapter pattern converts the interface of a class into another interface.

Use the Adapter pattern when

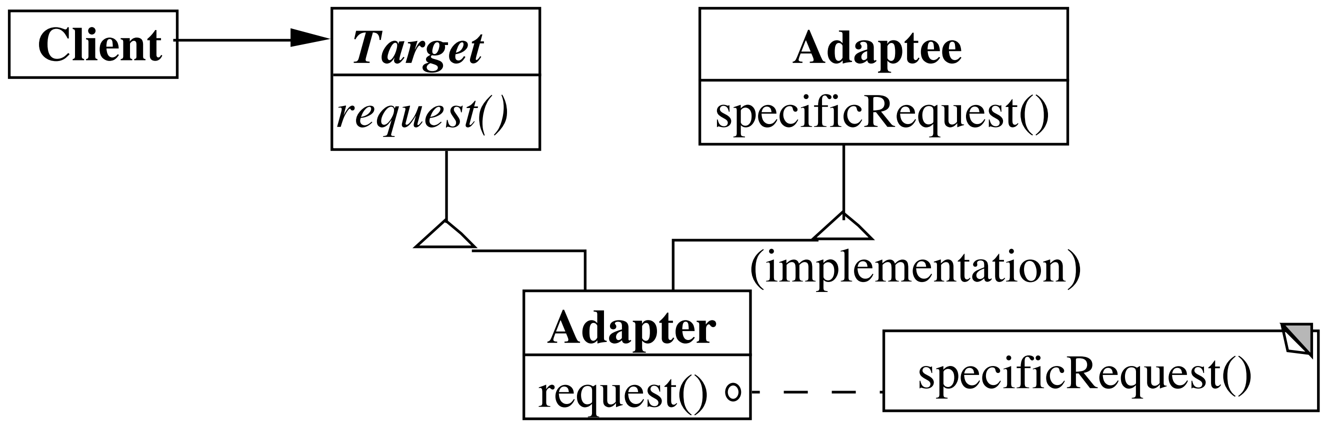
- You want to use an existing class and its interface does not match the one you need

- You want to create a reusable class that cooperates with unrelated or unforeseen classes, that is classes that don’t necessarily have compatible interfaces

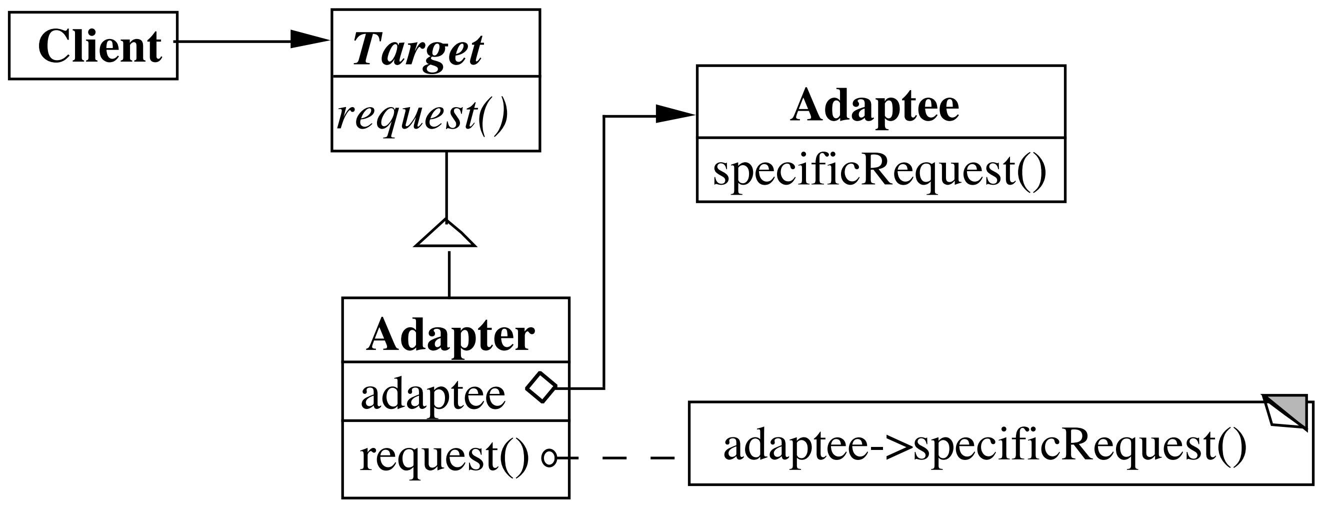
- You need to use several existing subclasses, but it’s impractical to adapt their interface by subclassing everyone. An object adapter can adapt the interface of its parent class

Adapter has two forms:
- Class Adapter
- Object Adapter

Class Adapter



Object Adapter



Class Adapter Example

```
class OldSquarePeg {
  public:
    void squarePegOperation()
      { do something }
}

class RoundPeg {
  public:
    void virtual roundPegOperation = 0;
}

class PegAdapter: private OldSquarePeg,
        public RoundPeg  {
  public:
    void virtual roundPegOperation()  {
      add some corners;
      squarePegOperation();
    }
}

void clientMethod() {
  RoundPeg* aPeg = new PegAdapter();
  aPeg->roundPegOperation();
}
```

Object Adapter Example

```
class OldSquarePeg     {
  public:
    void squarePegOperation() { do something }
  }

class RoundPeg{
  public:
    void virtual roundPegOperation = 0;
  }

class PegAdapter:  public RoundPeg   {
  private:
    OldSquarePeg* square;

  public:
    PegAdapter()  { square = new OldSquarePeg; }

    void virtual roundPegOperation() {
      add some corners;
      square->squarePegOperation();
      }
  }
```

Consequences

A Class adapter uses inheritance so

- Only adapts a class and all its parents, not all its subclasses
- Lets Adapter override some of Adaptee's behavior
- Does not introduce an additional pointer indirection

An object adapter uses object composition so

- Lets a single Adapter work with many Adaptees
- Makes it harder to override Adaptee behavior as the Adapter may not know with Adaptee it is working with

Other issues:

- How much adapting does the Adapter do?
- Pluggable adapters
- Using two-way adapters

How Much Adapting does the Adapter do?

The adapter may have to work very little or a great deal to adapt the Adaptee to the Target

The Adapter may just map one operation to another

```
class PegAdapter:  public RoundPeg   {
   private:
      OldSquarePeg* square;

   public:
      PegAdapter() { square = new OldSquarePeg;}

      void roundPegOperation()     {
         square->squarePegOperation();
      }
}
```

The Adapter may have to work hard if the Target operation does not have a comparable operation in the Adaptee

Pluggable Adapters

In the CGI example we adapted a class with getX() methods to a hash table interface

It is likely that we may adapt a class with getX() methods to a hashtable in the future

It would be nice to write one class to do all such adapting

This class would be given a list of keys to getX methods and an Adaptee object

```
HttpRequest CGIEnvironment = getHttpRequest();
PluggableHashAdapter sample =
   new PluggableHashAdapter( CGIEnvironment );

sample.adapt( "AUTH_TYPE" , getAuthType );
sample.adapt( "REMOTE_USER" , getRemoteUser );
etc.

sample.get( “REMOTE_USER” );
```

Pluggable Adapters are used in interface components, where we know in advance that we will adapt the component to other interfaces

Pluggable Adapters are common in Smalltalk, were it is easier to map strings to method calls

Using two-way Adapter

In the SquarePeg-RoundPeg example the SquarePeg is adapted to the RoundPeg

So a SquarePeg can be used where a RoundPeg is needed, but not the other way around.

A two-way adapter would also allow a RoundPeg be used in place of the SquarePeg

```
class OldSquarePeg {
  public:
    void virtual squarePegOperation() { blah }
}

class RoundPeg {
  public:
    void virtual roundPegOperation() { blah }
}

class PegAdapter: public OldSquarePeg, RoundPeg {
  public:
    void virtual roundPegOperation() {
      add some corners;
      squarePegOperation();
    }
    void virtual squarePegOperation() {
      add some corners;
      roundPegOperation();
    }
}
```