CS 635 Advanced Object-Oriented Design & Programming
Spring Semester, 2004
Doc 9 Command, Decorator & Proxy
Contents

Command 2
Structure 2
When to Use the Command Pattern 3
Consequences 4
Command Processor 18
Structure 19
Consequences 20
Functor 22
Decorator 24
Class Structure 24
Motivation - Text Views 25
Applicability 28
Consequences 28
Implementation Issues 29
Examples 30
Proxy 32
Structure 32
Dynamics 33
Reasons for Object Proxies 35
Smalltalk Proxy Tricks 39

References

Design Patterns: Elements of Reusable Object-Oriented Software, Gamma, Helm, Johnson, Vlissides, Addison-Wesley, 1995, pp. 233-242, 175-185, 207-217

The Design Patterns Smalltalk Companion, Alpert, Brown, Woolf, Addision-Wesley, 1998, pp. 245-260, 161-178, 213-221

Pattern-Oriented Software Architecture: A System of Patterns, Buschman, Meunier, Rohnert, Sommerlad, Stal, 1996, pp. 277-290, Command Processor

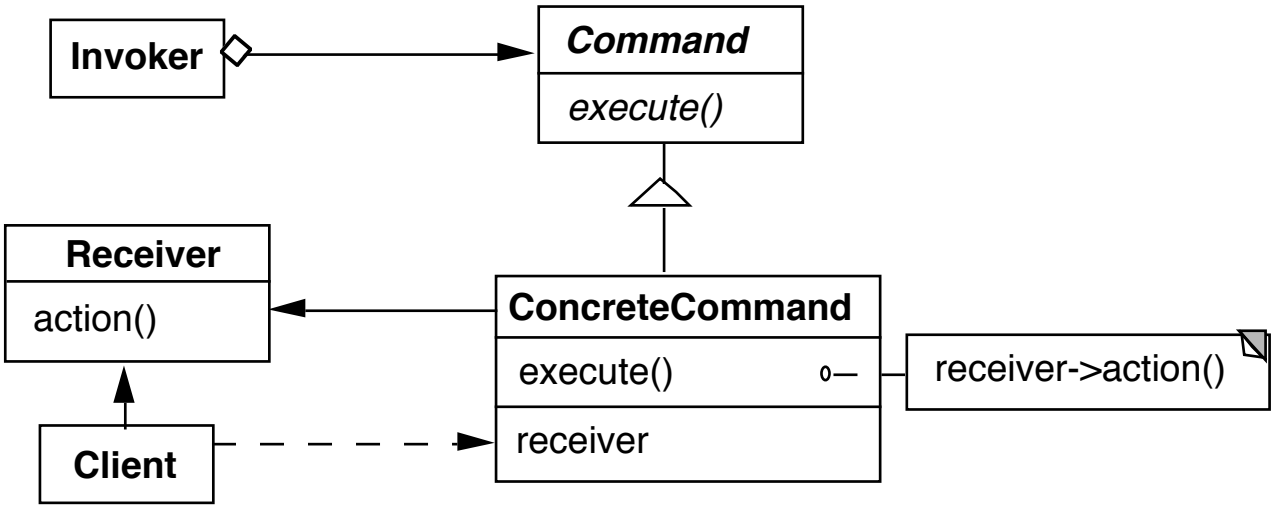
Command Processor, Sommerlad in Pattern Languages of Program Design 2, Eds. Vlissides, Coplien, Kerth, Addison-Wesley, 1996, pp. 63-74

Advanced C++: Programming Styles and Idioms, James Coplien, Addison Wesley, 1992, pp 165-170, Functor Pattern

Command

Encapsulates a request as an object

Structure



Example

Let
- Invoker be a menu
- Client be a word processing program
- Receiver a document
- Action be save

When to Use the Command Pattern

- When you need an action as a parameter
  Commands replace callback functions

- When you need to specify, queue, and execute requests at different times

- When you need to support undo

- When you need to support logging changes

- When you structure a system around high-level operations built on primitive operations

  A Transactions encapsulates a set of changes to data

  Systems that use transaction often can use the command pattern

- When you need to support a macro language

Consequences

Command decouples the object that invokes the operation from the one that knows how to perform it

It is easy to add new commands, because you do not have to change existing classes

You can assemble commands into a composite object

Example - Menu Callbacks

```
abstract class Command
   {
   abstract public void execute();
   }

class OpenCommand extends Command
   {
   private Application opener;

   public OpenCommand( Application theOpener )
      {
      opener = theOpener;
      }

   public void execute()
      {
      String documentName = AskUserSomeHow();

      if ( name != null )
         {
         Document toOpen =
               new Document( documentName );
         opener.add( toOpen );
         opener.open();
         }
      }
   }
```

Using Command

```
class Menu
  {
  private Hashtable menuActions = new Hashtable();

  public void addMenuItem( String displayString,
              Command itemAction )
    {
    menuActions.put( displayString, itemAction );
    }

  public void handleEvent( String itemSelected )
    {
    Command runMe;
    runMe = (Command) menuActions.get( itemSelected );
    runMe.execute();
    }

  // lots of stuff missing
  }
```

MacroCommand

```
class MacroCommand extends Command
  {
  private Vector commands = new Vector();

  public void add( Command toAdd )
    {
    commands.addElement( toAdd );
    }

  public void remove( Command toRemove )
    {
    commands.removeElement( toAdd );
    }

  public void execute()
    {
    Enumeration commandList = commands.elements();

    while ( commandList.hasMoreElements() )
      {
      Command nextCommand;
      nextCommand = (Command)
                commandList.nextElement();
      nextCommand.execute();
      }
    }
  }
```

Prevayler

http://www.prevayler.org/wiki.jsp

Prevalence layer for Java

Database that
- Serializes object to save them to disk
- Uses commands when modifying objects
- Keeps log of commands

Restaurant Example

```
import java.util.*;
import org.prevayler.implementation.AbstractPrevalentSystem;

public class Restaurant extends AbstractPrevalentSystem {
   private String name;
   ArrayList ratings = new ArrayList();

   public Restaurant(String newName) { name = newName;}

   public String name() {return name;}

   public void addRating( int newRating) {
      ratings.add( new Integer(newRating));
   }

   public float getRating() {
      if (ratings.size() == 0 )
         return -1;
      int total = 0;
      for (int k =0; k < ratings.size();k++)
         total = total + ((Integer)ratings.get(k)).intValue();
      return total/ ratings.size();
   }
}
```

Command

```
import java.io.Serializable;

import org.prevayler.Command;
import org.prevayler.PrevalentSystem;


public class AddRatingCommand implements Command {
  private final int newRating;

  public AddRatingCommand(int rating) {
    newRating = rating;
  }

  public Serializable execute(PrevalentSystem system) {
    ((Restaurant)system).addRating(newRating);
    return null;
  }

}
```

First Run

```
import java.util.*;
import org.prevayler.implementation.SnapshotPrevayler;

public class PrevaylerExample {

   public static void main (String args[]) throws Exception {
      SnapshotPrevayler samsDinerData =
         new SnapshotPrevayler(new Restaurant("Sams Diner"), "food");

      System.out.println( “Start”);
      Restaurant samsDiner = (Restaurant) samsDinerData.system();
      System.out.println( samsDiner.getRating() );
      samsDinerData.executeCommand( new AddRatingCommand( 5));
      System.out.println( samsDiner.getRating() );

   }
}
```

Output

```
Recovering system state...
Start
-1.0
5.0
```

Second Run

```
public class PrevaylerExample {

   public static void main (String args[]) throws Exception {
      SnapshotPrevayler samsDinerData =
         new SnapshotPrevayler(new Restaurant("Sams Diner"), "food");

      System.out.println( “Start”);
      Restaurant samsDiner = (Restaurant) samsDinerData.system();
      System.out.println( samsDiner.getRating() );
      samsDinerData.executeCommand( new AddRatingCommand( 10));
      System.out.println( samsDiner.getRating() );

   }
}
```

Output

```
Recovering system state...
Reading food/00000000000000000001.commandLog...
Start
5.0
7.0
```

Pluggable Commands

Using reflection it is possible to create one general Command

Don’t hard code the method called in the command

Pass the method to call an argument

Java Example of Pluggable Command

```
import java.util.*;
import java.lang.reflect.*;

public class Command
  {
  private Object receiver;
  private Method command;
  private Object[] arguments;

  public Command(Object receiver, Method command,
                 Object[] arguments )
    {
    this.receiver = receiver;
    this.command = command;
    this.arguments = arguments;
    }

  public void execute() throws InvocationTargetException,
                               IllegalAccessException
    {
    command.invoke( receiver, arguments );
    }
  }
```

Using the Pluggable Command

One does have to be careful with the primitive types

```
public class Test {
   public static void main(String[] args) throws Exception
      {
      Vector sample = new Vector();
      Class[] argumentTypes = { Object.class };
      Method add =
         Vector.class.getMethod( "addElement", argumentTypes);
      Object[] arguments = { "cat" };

      Command test = new Command(sample, add, arguments );
      test.execute();
      System.out.println( sample.elementAt( 0));
      }
   }
```

Output

cat

Pluggable Command Smalltalk Version

```
Object subclass: #PluggableCommand
   instanceVariableNames: 'receiver selector arguments '
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Whitney-Examples'
```

Class Methods

```
receiver: anObject selector: aSymbol arguments: anArrayOrNil
   ^super new
      setReceiver: anObject
      selector: aSymbol
      arguments: anArrayOrNil
```

Instance Methods

```
setReceiver: anObject selector: aSymbol arguments: anArrayOrNil
   receiver := anObject.
   selector := aSymbol.
   arguments := anArrayOrNil isNil
            ifTrue:[#( )]
            ifFalse: [anArrayOrNil]

execute
   ^receiver
      perform: selector
      withArguments: arguments
```

Using the Pluggable Command

```
| sample command |
sample := OrderedCollection new.
command := PluggableCommand
      receiver: sample
      selector: #add:
      arguments: #( 5 ).
command execute.
^sample at: 1
```

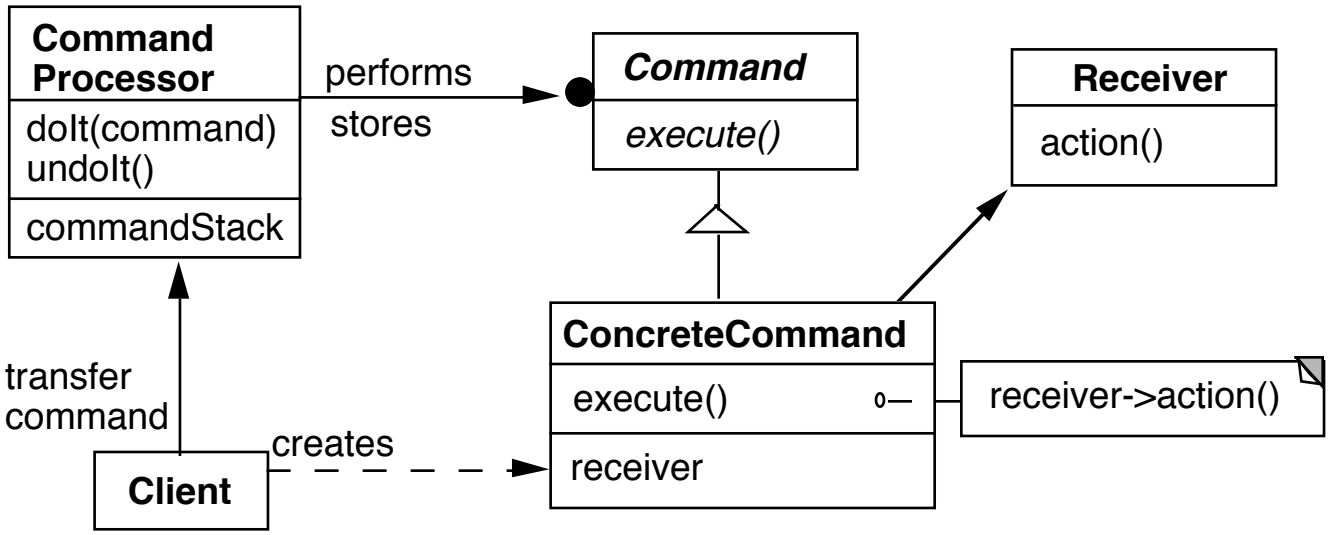
Command Processor

Command Processor manages the command objects

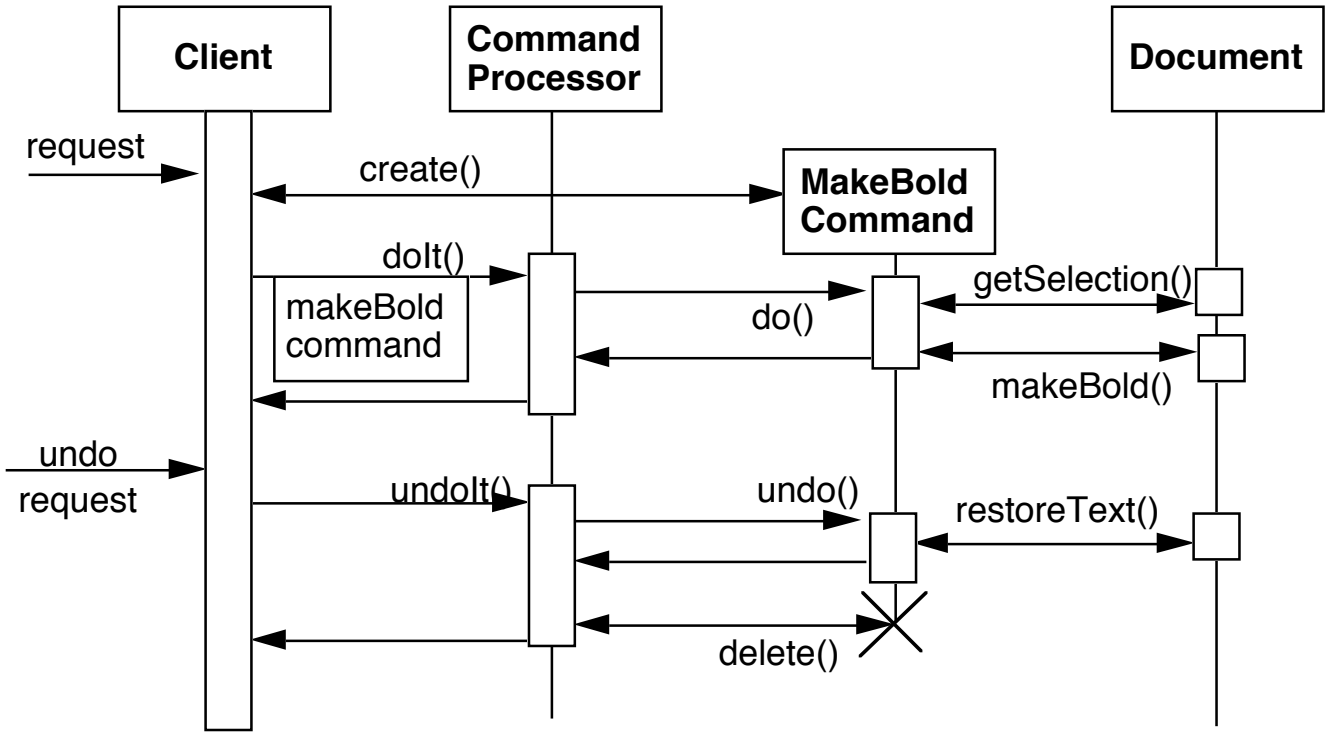
The command processor:

- Contains all command objects
- Schedules the execution of commands
- May store the commands for later unto
- May log the sequence of commands for testing purposes
- Uses singleton to insure only one instance

Structure



Dynamics



Consequences
Benefits

- Flexibility in the way requests are activated

  Different user interface elements can generate the same kind of command object

  Allows the user to configure commands performed by a user interface element

- Flexibility in the number and functionality of requests

  Adding new commands and providing for a macro language comes easy

- Programming execution-related services

  Commands can be stored for later replay
  Commands can be logged
  Commands can be rolled back

- Testability at application level

- Concurrency

  Allows for the execution of commands in separate threads

Liabilities

- Efficiency loss
- Potential for an excessive number of command classes

  Try reducing the number of command classes by:

  Grouping commands around abstractions

  Unifying simple commands classes by passing the receiver object as a parameter

- Complexity

  How do commands get additional parameters they need?

Functor
Functions as Objects

A functor is a class with

- A single member function (method)

Functors are functions that behave like objects

They serve the role of a function, but can be created, passed as parameters, and manipulated like objects

```
final class StudentNameComparator implements Comparator {

  public int compare( Object leftOp, Object rightOp ) {
    String leftName = ((Student) leftOp).name;
    String rightName = ((Student) rightOp).name;
    return leftName.compareTo( rightName );
  }
}
```

How Does a Functor Compare to Function Pointers?

- Using inheritance we can factor common code to a base class
- Same run-time flexibility as function pointer
- Lends it self to poor abstractions

How does A Functor compare with Command?

When to use the Functor

Coplien states:

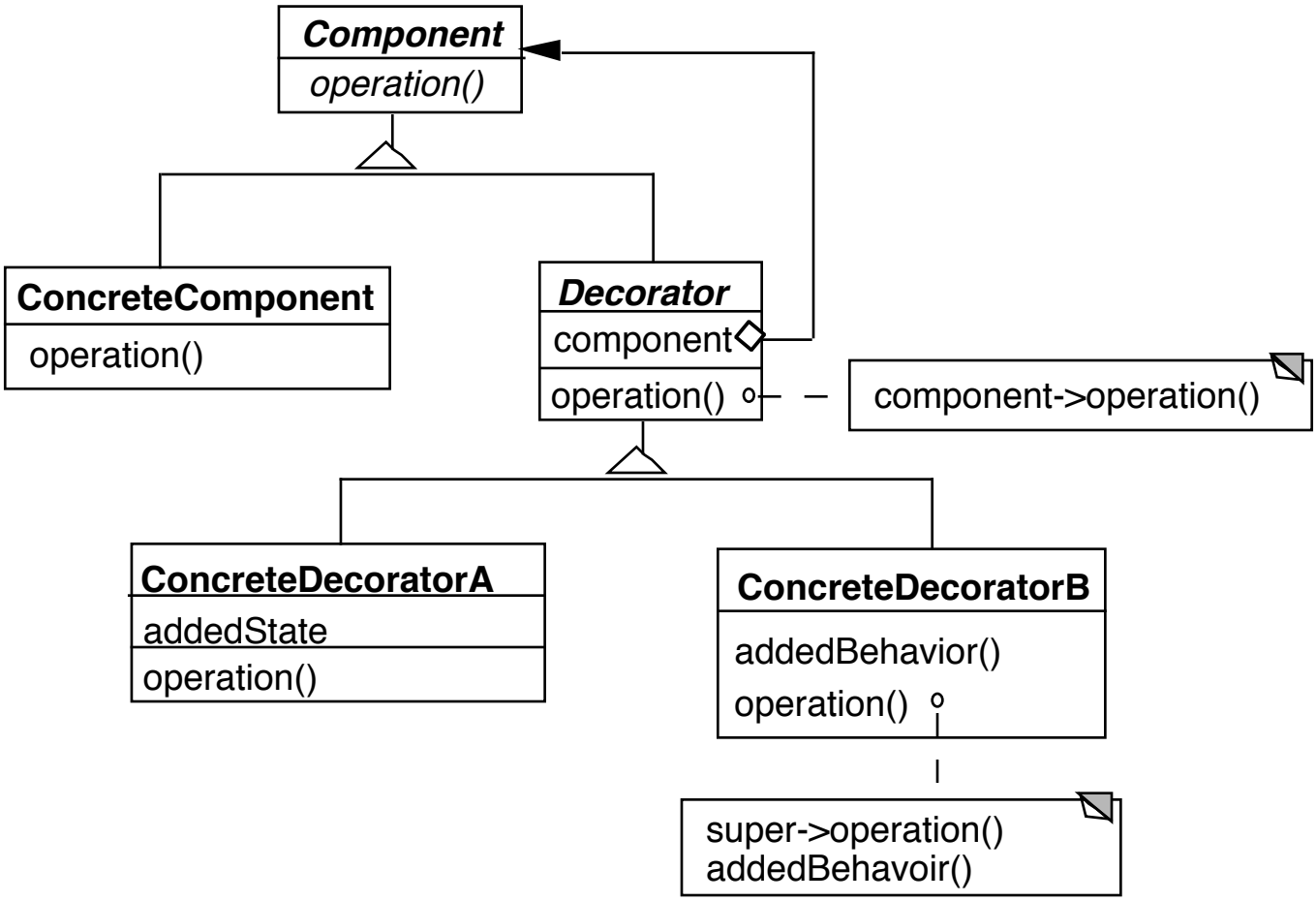
> Use functors when you would be tempted to use function pointers

Functors are commonly used for callbacks

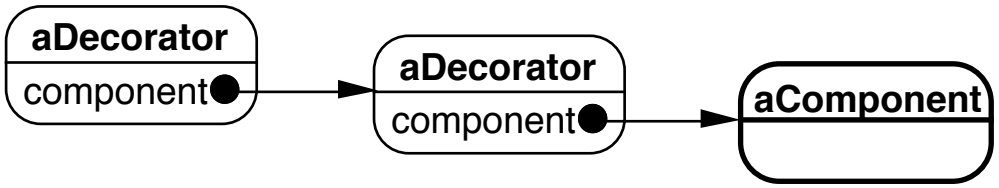
Decorator

Changing the Skin of an Object

Class Structure



Runtime Structure



Motivation - Text Views

A text view has the following features:

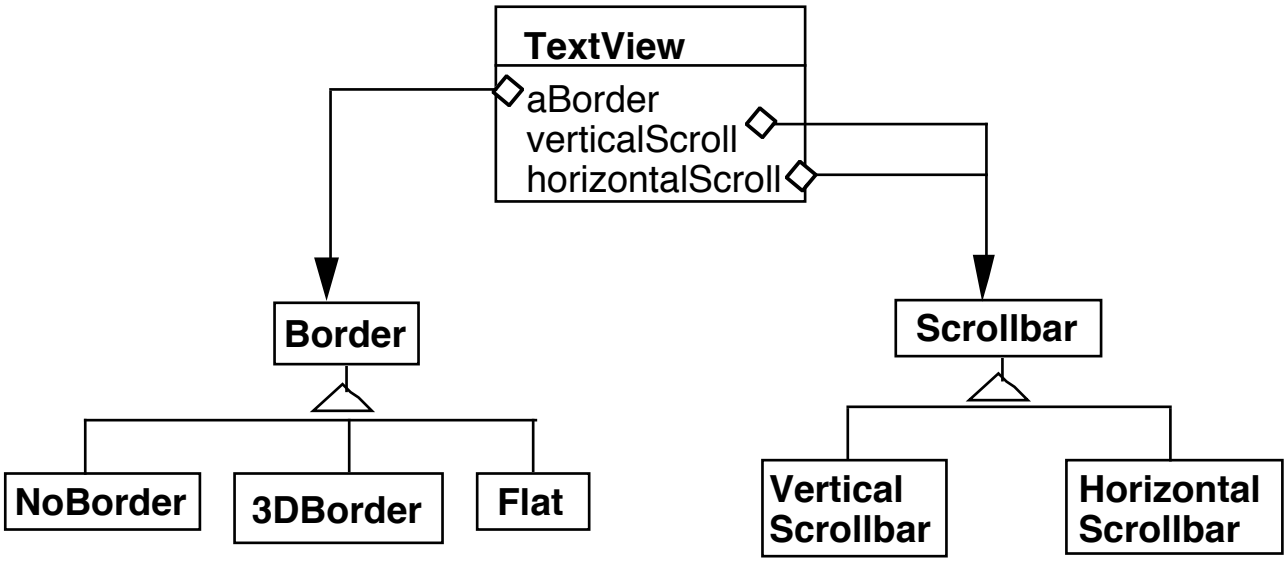
side scroll bar
Bottom scroll bar
3D border
Flat border

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



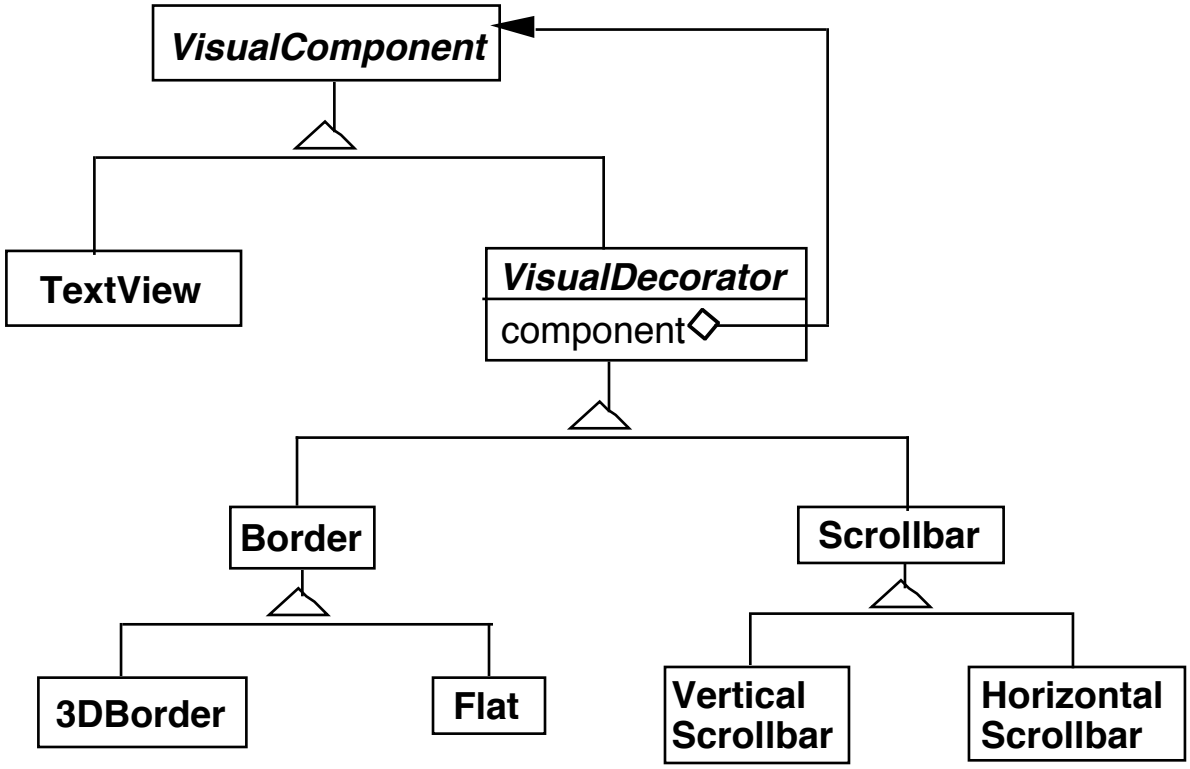

```
class TextView {
   Border myBorder;
   ScrollBar verticalBar;
   ScrollBar horizontalBar;

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

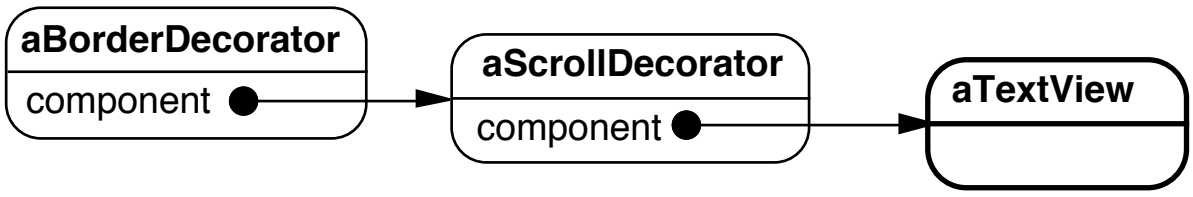
But TextView knows about all the variations!
New type of variations require changing TextView
(and any other type of view we have)

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
```

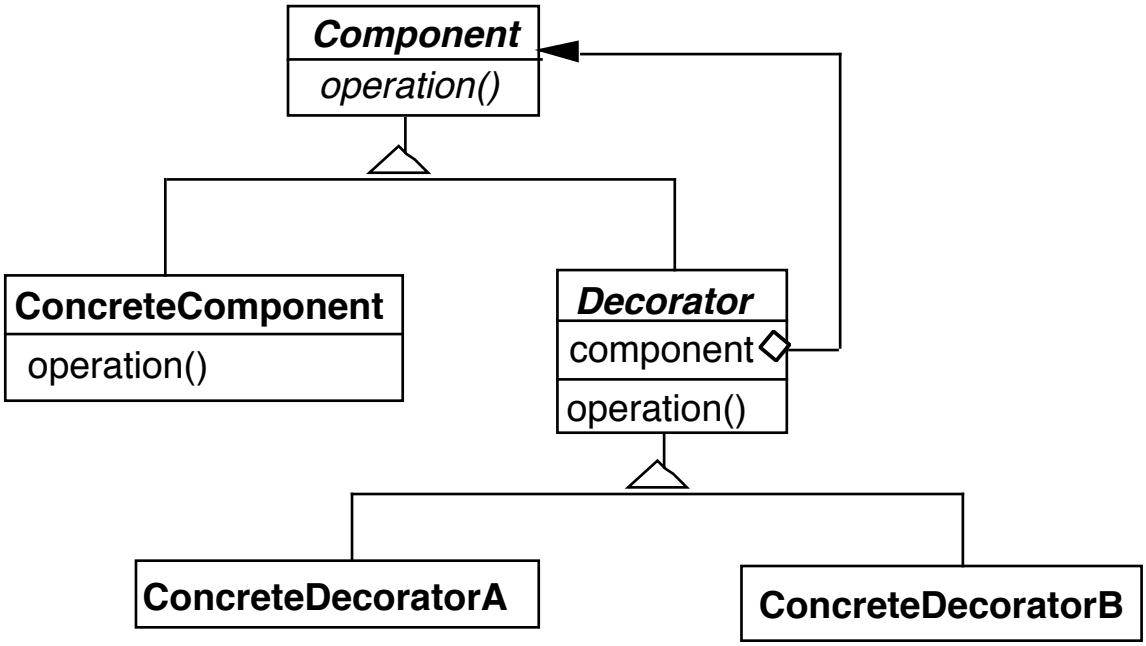
Implementation Issues

Keep Decorators lightweight

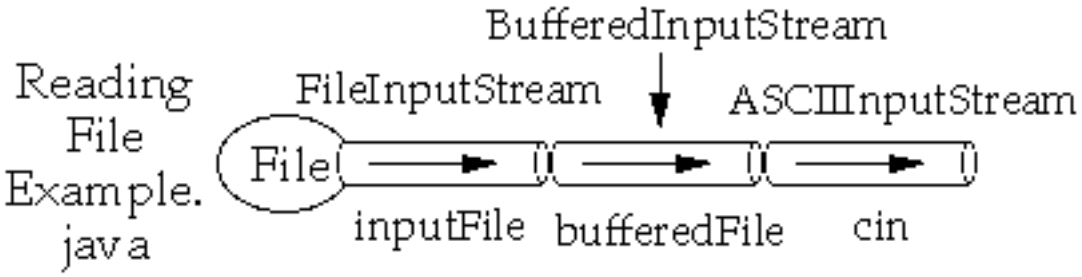
Don't put data members in VisualComponent

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

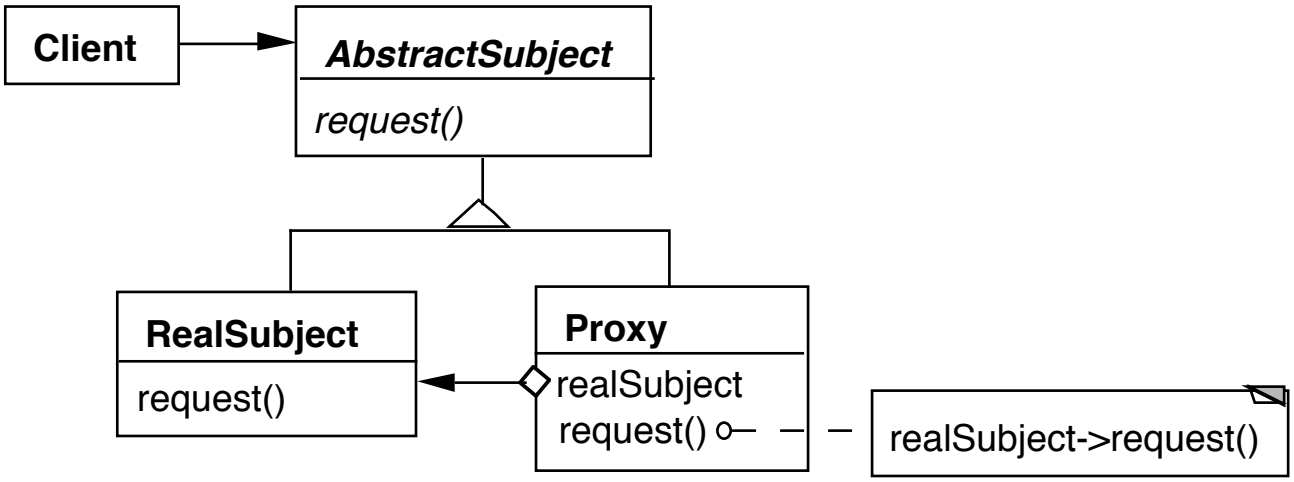
A decorator can be used to provide different caps on the same policy object

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



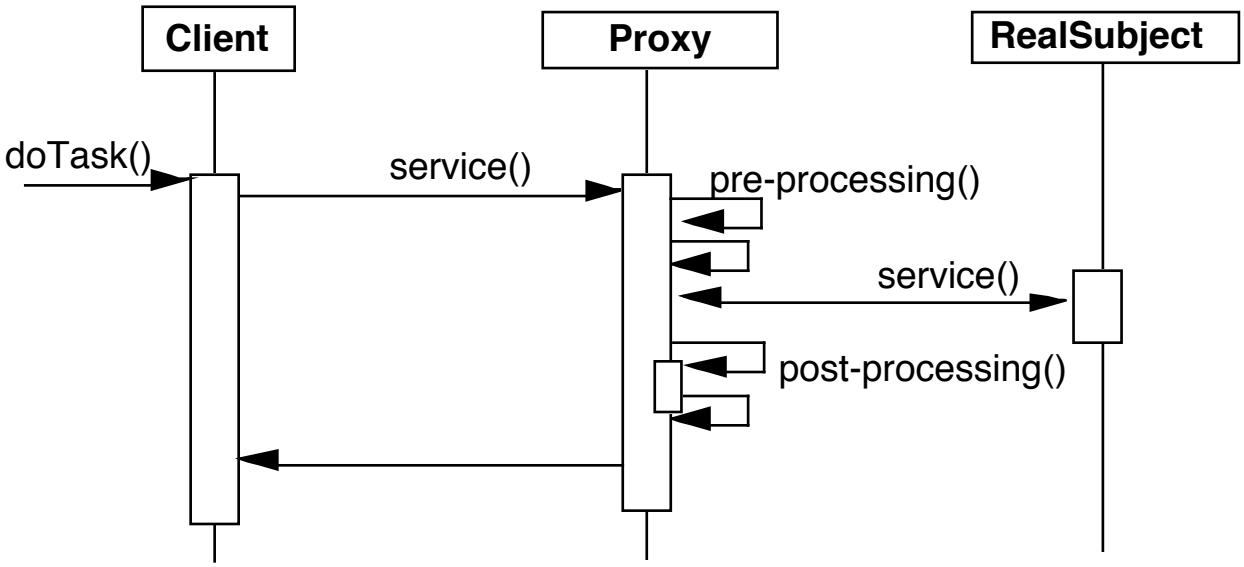
The Pattern

The proxy has the same interface as the original object

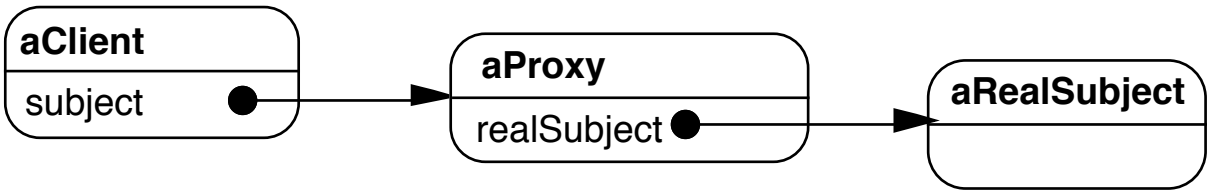
Use common interface (or abstract class) for both the proxy and original object

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
  }
```

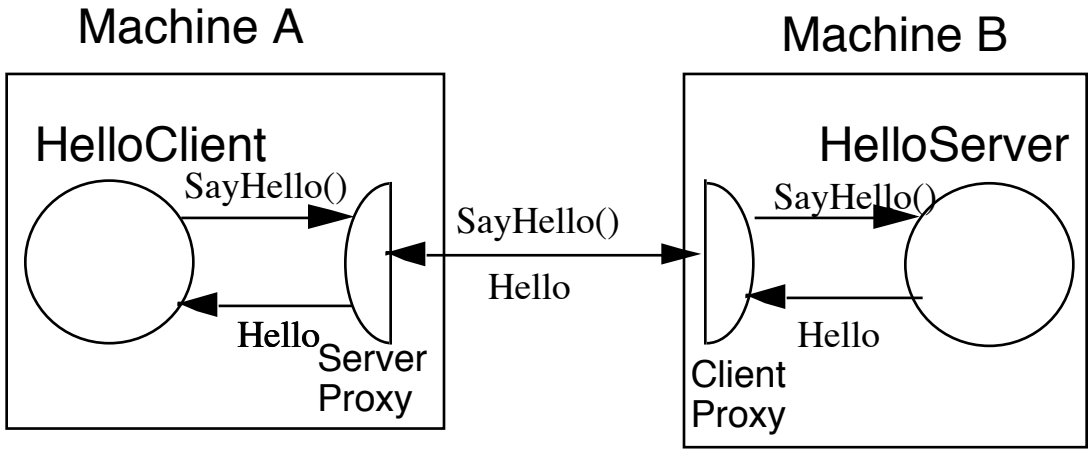
Preprocessing & post-processing depend on purpose of the proxy

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