CS 635 Advanced Object-Oriented Design & Programming
Spring Semester, 2016
Doc 7 Decorator, Pipes, Visitor
Feb 11, 2015

Copyright ©, All rights reserved. 2016 SDSU & Roger Whitney, 5500 Campanile Drive, San Diego, CA 92182-7700 USA. OpenContent (http://www.opencontent.org/opl.shtml) license defines the copyright on this document.
Decorator
Decorator Pattern
Adds responsibilities to individual objects

Dynamically
Transparently
import java.io.*;
import sdsu.io.*;

class ReadingFileExample
{
    public static void main( String args[] ) throws Exception
    {
        FileInputStream inputFile;
        BufferedReader bufferedFile;
        ASCIIInputStream cin;

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

        // Code continues here
    }
}
ConcreteDecoratorB
Component
operation()
Decorator
operation()
component
ConcreteComponent
operation()
ConcreteDecoratorA
ConcreteDecoratorB

Component
operation()
Decorator forwards all component operations
Favor Composition over Inheritance
Refactoring: Move Embellishment to Decorator

Client → aBinaryTree
  toArray

Client → startsWithVowelDecorator → aBinaryTree
  toArray → toArray
Benefits & Liabilities

Benefits

Simplifies a class
Distinguishes a class's core responsibilities from embellishments

Liabilities

Changes the object identity of a decorated object
Code harder to understand and debug
Combinations of decorators may not work correctly together
**Clojure Threading Macro**

```clojure
(-> "a b c d"
    .toUpperCase
    (.replace "A" "X")
    (.split " ")
    first)
```

```clojure
(first (.split (.replace (.toUpperCase "a b c d") "A" "X") " ")
```

From Clojure Docs
Pipes and Filters
Pipes & Filters

ls | grep -i b | wc -l

Context
Processing data streams

Problem
Building a system that processes or transforms a stream of data

Forces
Small processing steps are easier to reuse than large components
Non-adjacent processing steps do not share information
System changes should be possible by exchanging or recombining processing steps, even by users
Final results should be presented or stored in different ways
Solution

Divide task into multiple sequential processing steps or filter components

Output of one filter is the input of the next filter

Filters process data incrementally

FILTER does not wait to get all the data before processing
Data source – input to the system

Data sink – output of the system

Pipes - connect the data source, filters and data sink

Pipe implements the data flow between adjacent processes steps

Processing pipeline – sequence of filters and pipes

Pipeline can process batches of data
Python Interpreter

http://wiki.cs.uiuc.edu/cs427/Python+-+Batch+Sequential
Intercepting Filter - Problem

Preprocessing and post-processing of a client Web request and response

A Web request often must pass several tests prior to the main processing

Has the client been authenticated?
Does the client have a valid session?
Is the client's IP address from a trusted network?
Does the request path violate any constraints?
What encoding does the client use to send the data?
Do we support the browser type of the client?

Nested if statements lead to fragile code
Intercepting Filter - Forces

Common processing, such as checking the data-encoding scheme or logging information about each request, completes per request.

Centralization of common logic is desired.

Services should be easy to add or remove unobtrusively without affecting existing components, so that they can be used in a variety of combinations, such as

Logging and authentication

Debugging and transformation of output for a specific client

Uncompressing and converting encoding scheme of input
Visitor Pattern
Visitor

Intent
Represent an operation to be performed on the elements of an object structure

Visitor lets you define a new operation without changing the classes of the elements on which it operates
Tree Example

class Node { ... }

class InnerNode extends Node {...}

class LeafNode extends Node {...}

class Tree { ... }
## Tree Printing

<table>
<thead>
<tr>
<th>Printing Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTML Print</td>
<td>Operations are complex</td>
</tr>
<tr>
<td>PDF Print</td>
<td>Do different things on different types of nodes</td>
</tr>
<tr>
<td>TeX Print</td>
<td>Need to traverse tree</td>
</tr>
<tr>
<td>RTF Print</td>
<td></td>
</tr>
<tr>
<td>Others likely in future</td>
<td>Not part of BST abstraction</td>
</tr>
</tbody>
</table>
Assume

Diagram:

- Document
  - HTMLDocument
  - PDFDocument
  - TeXDocument
First Attempt

print(Tree source, Document output) {
    foreach( Node current : source ) {
        if current.isInnerNode() && output.isHtml() {
            print inner node on html document
        } else if current.isLeafNode() && output.isHtml() {
            print leaf node on html document
        } else if current.isInnerNode() && output.isPDF() {
            print inner node on pdf document
        } else if current.isLeafNode() && output.isPDF() {
            print leaf node on pdf document
        } etc.
    }
}
Second Attempt

Create Printer Classes

Use iterator to access all elements

Process each element
class TreePrinter {
    public void printTree (Tree toPrint, Document output) {
        foreach( Node current : source ) {
            if (current.isLeafNode())
                printLeafNode(current, output);
            else if (current.isInternalNode() )
                printInternalNode(current, output);
        }
    }

    private void printLeafNode(Node current, Document output) {
        if output.isHtml()
            print leaf node on html document
        else if output.isPDF()
            print leaf node on PDF document
        else if etc
    }
}
What we would like

class TreePrinter {
    public void printTree (Tree source, Document output) {
        foreach( Node current : source ) {
            printNode(current, output);
        }
    }
}

private void printNode(InnerNode current, HTMLDocument output) {
    print inner node on html document
}

private void printNode(LeafNode current, HTMLDocument output) {
    print leaf node on html document
}

private void printNode(InnerNode current, PDFDocument output) {
    print inner node on html document
}

etc
Overloaded Methods

Which overloaded method to run

Selected at compile time

Based on declared type of parameter

Does not use runtime information
Use Subclasses

TreePrinter

HTMLTreePrinter  PDFTreePrinter  TeXTreePrinter
class TreePrinter {
    Document output;
    public void printTree (Tree toPrint) {
        foreach( Node current : source ) {
            if (current.isLeafNode())
                printLeafNode(current, output);
            else if (current.isInternalNode() )
                printInternalNode(current, output);
        }
    }

    public Document getDocument() { return output;}

    private abstract void printLeafNode(Node current);
    private abstract void printInnerNode(Node current);
}

Thursday, February 11, 16
Third Attempt

class HTMLTreePrinter extends TreePrinter {

    private void printLeafNode(Node current) {
        print leaf node on html document
    }

    private void printInnerNode(Node current) {
        print inner node on html document
    }
}
Overloaded Method

class TreePrinter {
    Document output;
    public void printTree (Tree toPrint) {
        foreach( Node current : source ) {
            printNode(current);
        }
    }

    public Document getDocument() { return output;}

    private abstract void printNode(LeafNode current);
    private abstract void printNode(InnerNode current);
}
Key Idea

Receiver of method is determined at runtime

x.toString();

Send a message to Nodes to determine what type of node we have
Add Methods to Nodes

class Node {
    abstract public void print(TreePrinter printer);
}

class InnerNode extends Node {
    public void print(TreePrinter printer) {
        printer.printInnerNode( this );
    }
}

class LeafNode extends Node {
    public void print(TreePrinter printer) {
        printer.printLeafNode( this );
    }
}
Now we can Use Polymorphism

class TreePrinter {
    Document output;
    public void printTree (Tree toPrint) {
        foreach( Node current : source ) {
            current.print(this);
        }
    }
    public Document getDocument() { return output;}

    public abstract void printLeafNode(Node current);
    public abstract void printInnerNode(Node current);
}

Thursday, February 11, 16
What Have we gained

No if statements

Can add more types of Documents by adding subclasses

Work for a Document is in one place

Divided work into small parts
We can use method overloading

class TreePrinter {
    Document output;
    public void printTree (Tree toPrint) {
        foreach( Node current : source ) {
            current.print(this);
        }
    }
    public Document getDocument() { return output;}
    public abstract void printNode(InnerNode current);
    public abstract void printNode(LeafNode current);
}

class InnerNode extends Node {
    public void print(TreePrinter printer) {
        printer.printNode( this );
    }
}

class LeafNode extends Node {
    public void print(TreePrinter printer) {
        aVisitor.printNode( this );
    }
}
But We don’t gain anything

class TreePrinter {
    Document output;
    public void printTree (Tree toPrint) {
        foreach( Node current : source ) {
            current.print(this);
        }
    }

    public Document getDocument() { return output;}

    public abstract void printNode(InnerNode current);
    public abstract void printNode(LeafNode current);
}

Still need to know about each node type
One Last Problem

Modified the nodes for a specific issue

For each issue need to add methods to node!?!?

Make the structure generic
In The Nodes

class Node {
    abstract public void accept(Visitor aVisitor);
}

class BinaryTreeNode extends Node {
    public void accept(Visitor aVisitor) {
        aVisitor.visitBinaryTreeNode( this );
    }
}

class BinaryTreeLeaf extends Node {
    public void accept(Visitor aVisitor) {
        aVisitor.visitBinaryTreeLeaf( this );
    }
}
abstract class Visitor {

    abstract void visitBinaryTreeNode( BinaryTreeNode );

    abstract void visitBinaryTreeLeaf( BinaryTreeLeaf );
}

class HTMLPrintVisitor extends Visitor {

    public void visitBinaryTreeNode( BinaryTreeNode x ) {
        HTML print code here
    }

    public void visitBinaryTreeLeaf( BinaryTreeLeaf x){ ...}
}
Visitor printer = new HTMLPrintVisitor();
Tree toPrint;

Iterator nodes = toPrint.iterator();
foreach( Node current : source ) {
    current.accept(printer);
}

Node object calls correct method in Printer
Tree Example

class BinaryTreeNode extends Node {
    public void accept(Visitor aVisitor) {
        aVisitor.visitBinaryTreeNode( this );
    }
}

class BinaryTreeLeaf extends Node {
    public void accept(Visitor aVisitor) {
        aVisitor.visitBinaryTreeLeaf( this );
    }
}

abstract class Visitor {
    abstract void visitBinaryTreeNode( BinaryTreeNode );
    abstract void visitBinaryTreeLeaf( BinaryTreeLeaf );
}

class HTMLPrintVisitor extends Visitor {
    public void visitBinaryTreeNode( BinaryTreeNode x ) {
        HTML print code here
    }
    public void visitBinaryTreeLeaf( BinaryTreeLeaf x) {
        ...
    }
}
Tree Example
Note that a visit to one node requires two method calls:

Node example = new BinaryTreeLeaf();
Visitor traveler = new HTMLPrintVisitor();
example.accept(traveler);
Issue - Who does the traversal?

Visitor

Elements in the Structure

Iterator
When to Use the Visitor

Have many classes of objects with differing interfaces, and you want to perform operations on these objects that depend on their concrete classes.

When many distinct and unrelated operations need to be performed on objects in an object structure and you want to avoid cluttering the classes with these operations.

When the classes defining the structure rarely change, but you often want to define new operations over the structure.
Consequences

Visitors makes adding new operations easier

Visitors gathers related operations, separates unrelated ones

Adding new ConcreteElement classes is hard

Visiting across class hierarchies

Accumulating state

Breaking encapsulation
Avoiding the accept() method

Visitor pattern requires elements to have an accept method

Sometimes this is not possible

You don’t have the source for the elements

Aspect Oriented Programming

AspectJ eliminates the need for an accept method in aspect oriented Java

AspectS provides a similar process for Smalltalk
Example - Magritte

Web applications have data (domain models)

We need to
  Display the data
  Enter the data
  Validate data
  Store Data
For each field in a domain model (class) provide a description

Description contains
  Data type             Display string
  Field name           Constraints

descriptionFirstName
  ^ (MAStringDescription auto: 'firstName' label: 'First Name' priority: 20)
  beRequired;
  yourself.

descriptionBirthday
  ^ (MADateDescription auto: 'birthday' label: 'Birthday' priority: 70)
  between:(Date year: 1900) and:Datetoday;
  yourself
Each domain model has a collection of descriptions

Different visitors are used to

  Generate html to display data

  Generate form to enter the data

  Validate data from form

  Save data in database
Sample Page

editor := (Person new asComponent)
addValidatedSwitch;
yourself.
result := self call: editor.

Edit Person

Title: 
First Name: 
Last Name: 
Home Address: Create
Office Address: Create
Picture: Choose File no file selected upload
Birthday: 
Age: 

Phone Numbers: The report is empty.

Kind Number
Add

Save Cancel
Refactoring: Move Accumulation to Visitor

A method accumulates information from heterogeneous classes

so

Move the accumulation task to a Visitor that can visit each class to accumulate the information

See Refactoring to Patterns, Kerievsky, 2005, pp 320–338 for details
Clojure, Lisp & Multi-methods

(defmulti printNode (fn [node document] [(class node) (class document)]))

(defmethod printNode [InnerNode HTMLDocument]
  [node document]
  code to print InnerNode on HTMLDocument)

(defmethod printNode [InnerNode PDFDocument]
  [node document]
  code to print InnerNode on PDFDocument)

(defmethod printNode [LeafNode PDFDocument]
  [node document]
  code to print InnerNode on PDFDocument)

etc.
Clojure, Lisp & Multi-methods

Multi-methods in Clojure do select overloaded method
   At run-time
   Based on argument types

No need for visitor pattern