CS 635 Advanced Object-Oriented Design & Programming
Spring Semester, 2001
Doc 7 Visitor Pattern
Contents

Visitor 2
  Structure 6
  When to Use Visitor 7
  Consequences 8
  Variations 9

References

Design Patterns: Elements of Resuable Object-Oriented Software, Gamma, Helm, Johnson, Vlissides, Addison-Wesley, 1995, pp. 331-344

"Default and Extrinsic Visitor", Martin E. Nordberg III, in Pattern Languages of Program Design 3, Edited by Martin, Riehle, Buschmmann, Addison-Wesley, 1998, pp. 105-123

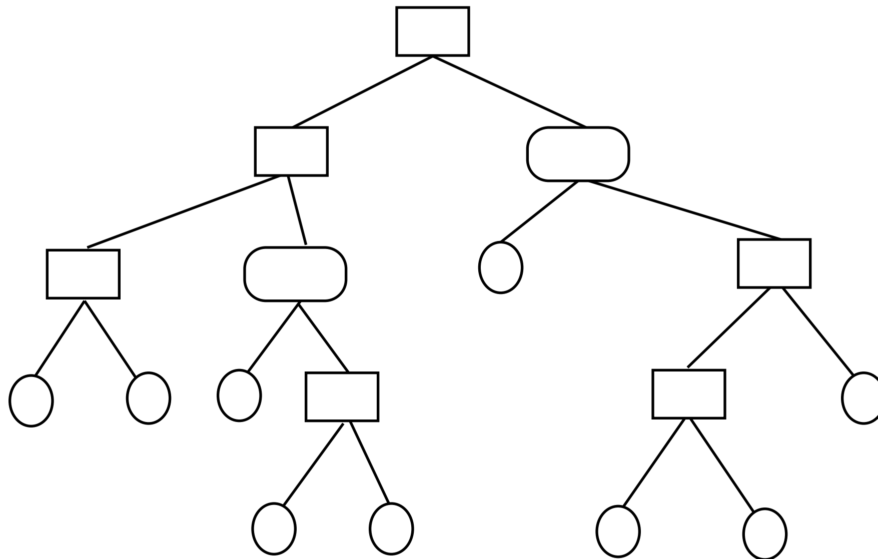
"Acyclic Visitor", Robert C. Martin, in Pattern Languages of Program Design 3, Edited by Martin, Riehle, Buschmmann, Addison-Wesley, 1998, pp. 94-104

Reading

Design Patterns text, pp. 331-334

Copyright ©, All rights reserved. 2001 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.

Visitor
Example - Trees



What about preorder visit, postorder visit?
What about an HTML print?
What about printing a 2D representation?
What if this was an expression tree - evaluation?
What if this was a binary search tree - adding, deleting
What about balancing the binary search tree -
  AVL
  Red-Black

Solution 1

Build all the operations into the tree structure

This works in many situations

If you need to add a lot of operations the classes can become cluttered

```
class BinaryTree {
   public String htmlPrint() { blah}
   public String 2DPrint() { blah}
   public String preorderTraversal() { blah}
   public String postorderTraversal() { blah}
   public String avlAdd() { blah}
   public String RedBlackAdd() { blah}
   etc.
}
```

Solution 2

Use different classes or subclasses for the different types of trees

Does not work in classes we want to be able to mix all combinations

```
class BinaryTree {
   blah
}

class AVLTree extends BinaryTree {
   blah
}

class RedBlackTree extends BinaryTree {
   blah
}

class PreorderTree extends BinayrTree {
   blah
}
```

etc.

Solution 3

- Put operations into separate object - a visitor
- Pass the visitor to each element in the structure
- The element then calls activates the visitor
- Visitor performs its operation on the element

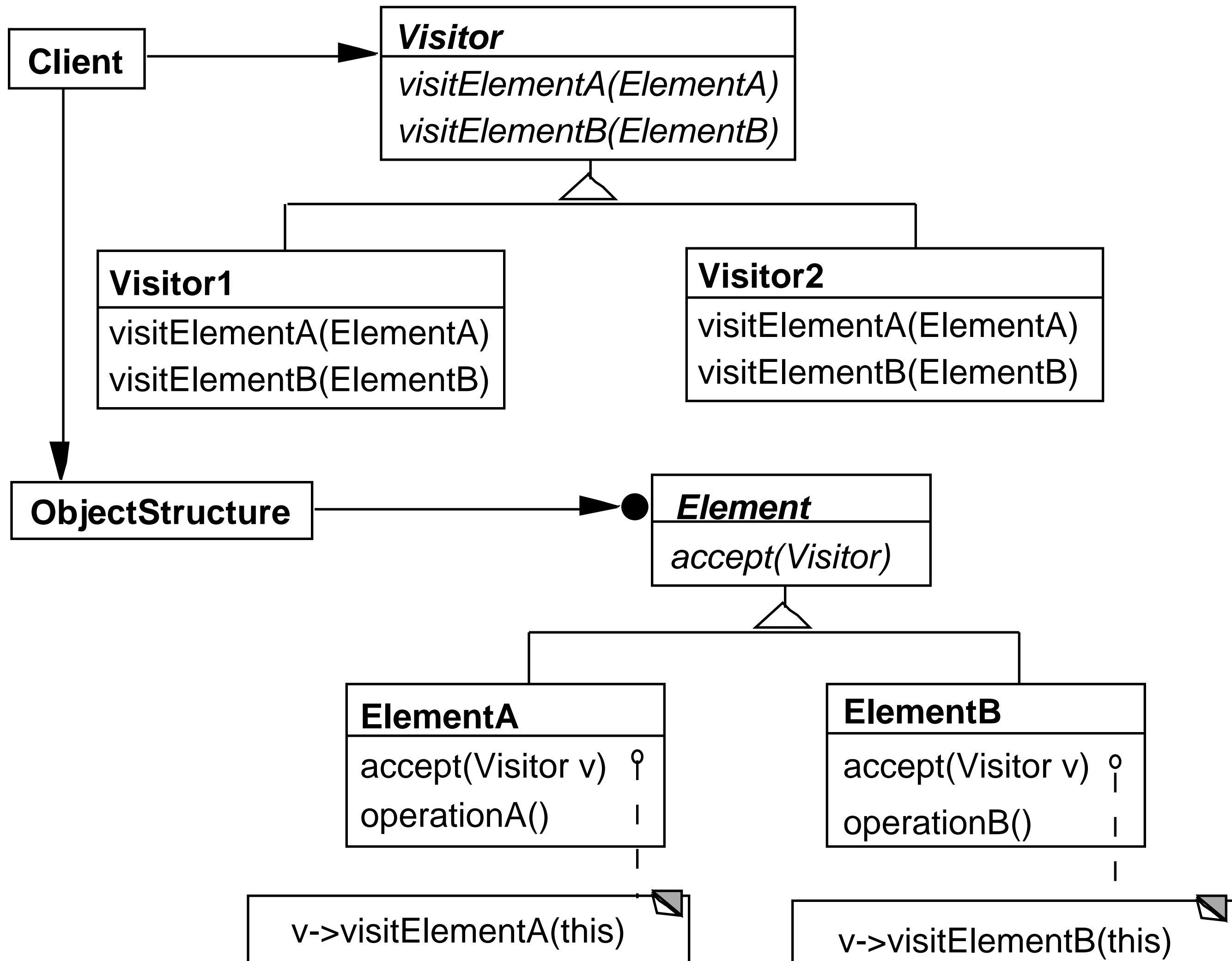
```
class BinaryTreeNode {
   public void accept(Visitor aVisitor) {
      aVisitor.visitBinaryTreeNode( this );
   }
   etc.
}

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

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

class HTMLPrintVisitor extends Visitor {
   public void visitBinaryTreeNode( BinaryTreeNode ) {
      HTML print code here
   }
   etc.
}
```

Structure



This is more complex than solutions 1 & 2

The structure, an iterator, or the visitor can do the traversal

- Usually the traversal is done by the structure
- Having the traversal in the visitor can lead to duplicated code

The visitor is told what type it is acting on, so using the wrong visitor will be a compile error

When to Use Visitor

- When an object structure contains 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 preformed 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

  If the structure involves many different classes then adding a new operation to the structure requires changing all those classes

- Visitors gathers related operations, separates unrelated ones

- Adding new ConcreteElement classes is hard

  To add a new ConcreteElement you need to change all existing visitors

- Visiting across class hierarchies

  Text claims iterators can not iterate through structure containing unrelated classes

  However, the visitor assumes that each element in the structure contains a visit method, which implies at least a common visit interface for all elements

- Accumulating state

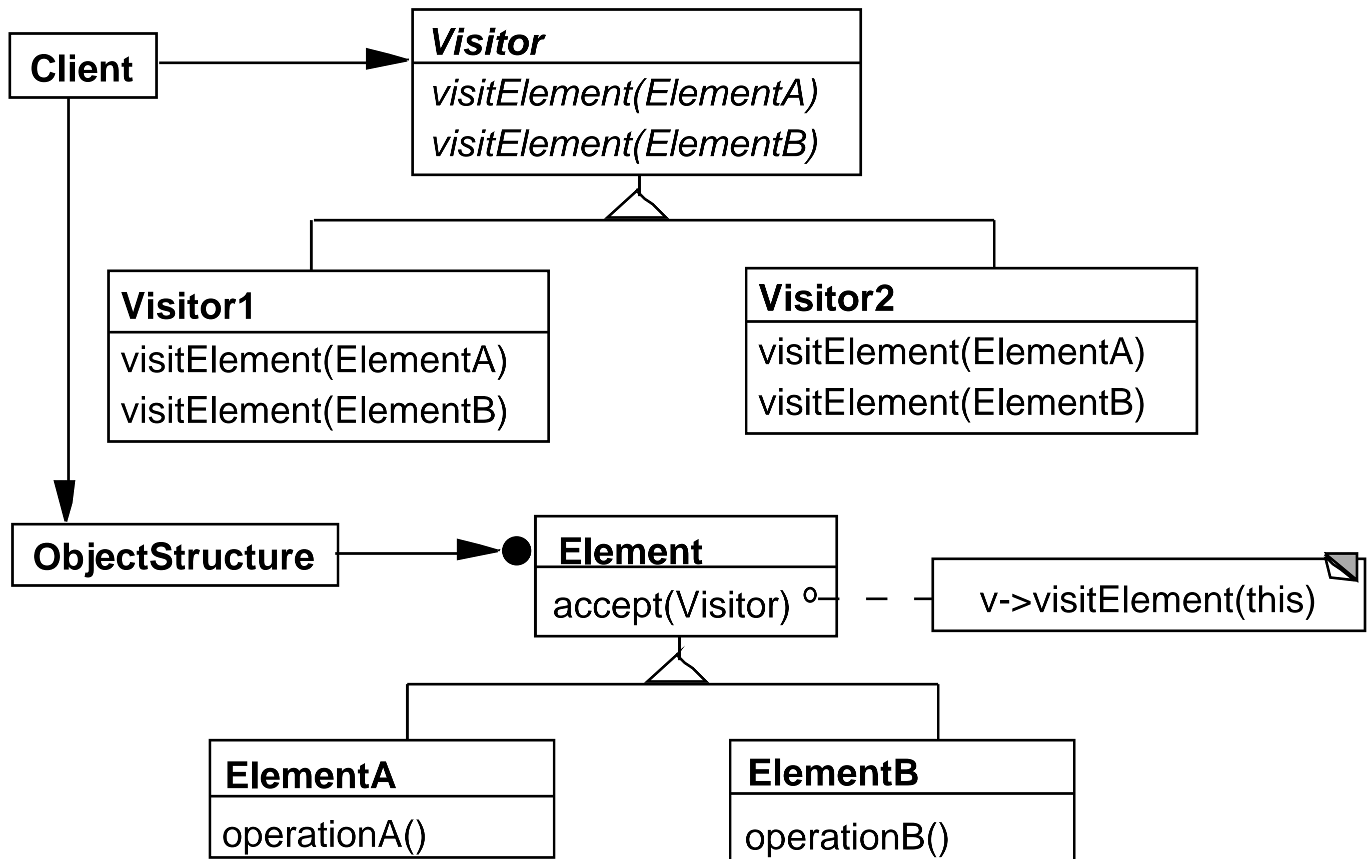
  Visitor can accumulate information

- Breaking encapsulation

  The visitor may force you to provide public operations in the elements that you would not otherwise make public

  C++ friends are useful here

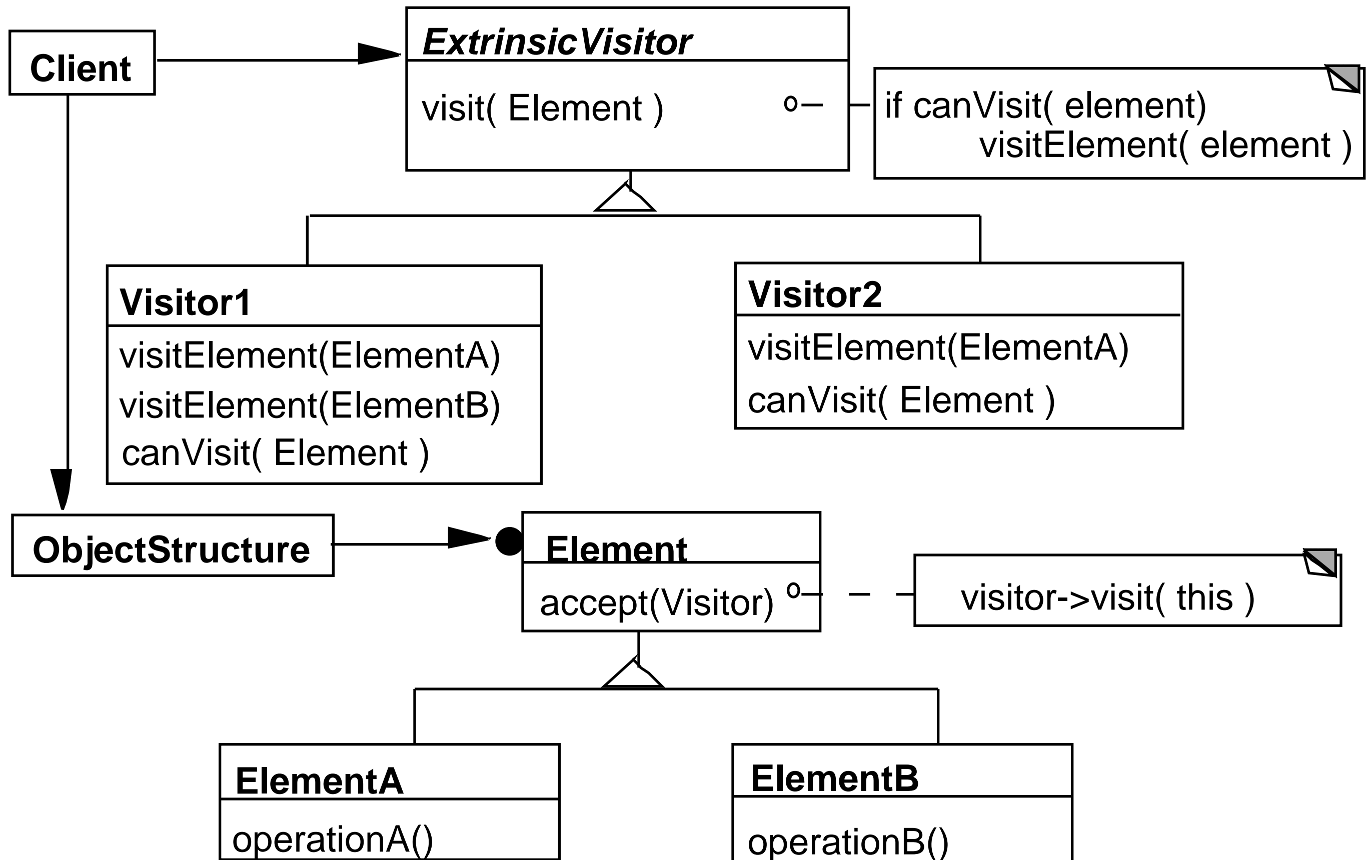
Variations
Function Overloading



Overloading allows the accept method to be implemented once in the Element class

Eliminates cyclic compile dependencies

ExtrinsicVisitor



The Visitor takes the responsibility of making sure a visitor can visit a particular concrete element

Each Visitor knows which types of concrete classes it can visit

The canVisit method of each visitor returns true if that visitor can visit that concrete type

New visitors can be created without changing the Element classes

New Element classes can be added without having to change old visitors

The above is slightly modified from the original ExtrinsicVisitor pattern

In the original

Element does not have an accept method

The visit method of ExtrinsicVisitor does not call canVisit

ObjectStructure calls canVisit to determine if an element can be visited, then calls visit

The visitElement methods are not over loaded