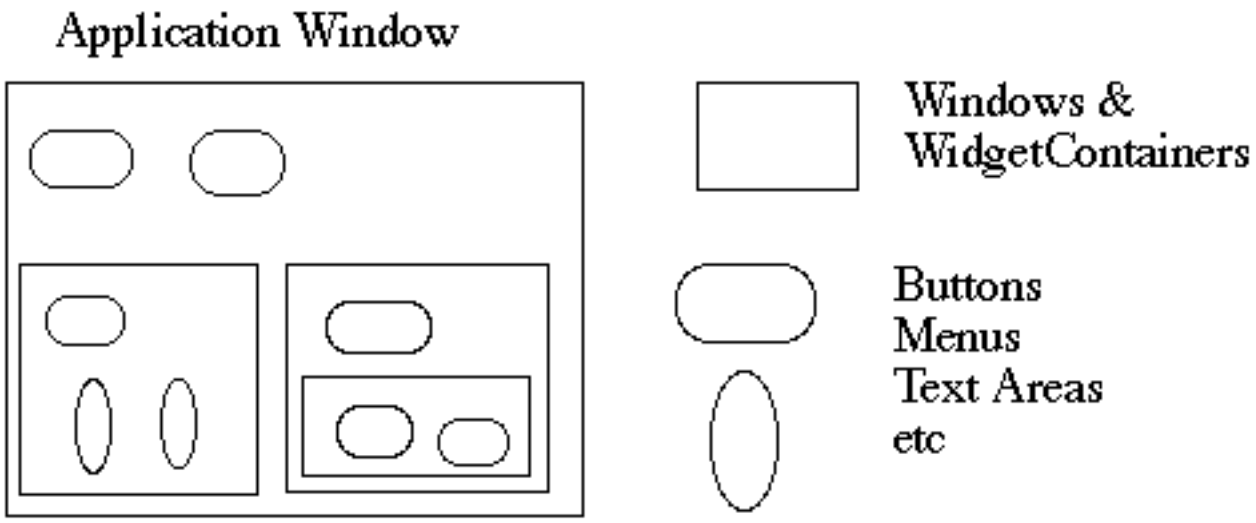
CS 635 Advanced Object-Oriented Design & Programming
Spring Semester, 2004
Doc 4 Composite & Visitor
Contents

References

Design Patterns: Elements of Resuable Object-Oriented Software, Gamma, Helm, Johnson, Vlissides, Addison-Wesley, 1995, pp. 163-174, 331-344

The Design Patterns Smalltalk Companion, Alpert, Brown, Woolf, Addison-Wesley, 1995, pp. 371-386

Composite
Example - Motivation
GUI Windows and GUI elements



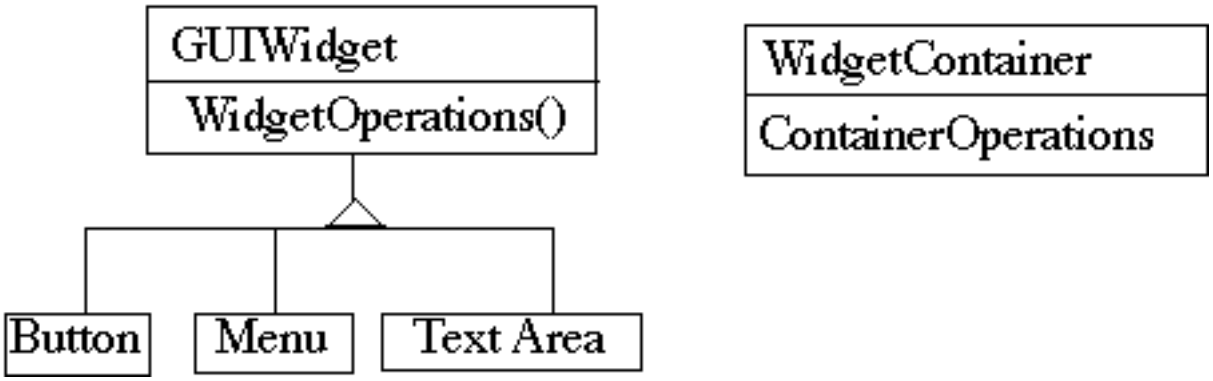
How does the window hold and deal with the different items it has to manage?

Widgets are different that WidgetContainers

Bad News Implementation

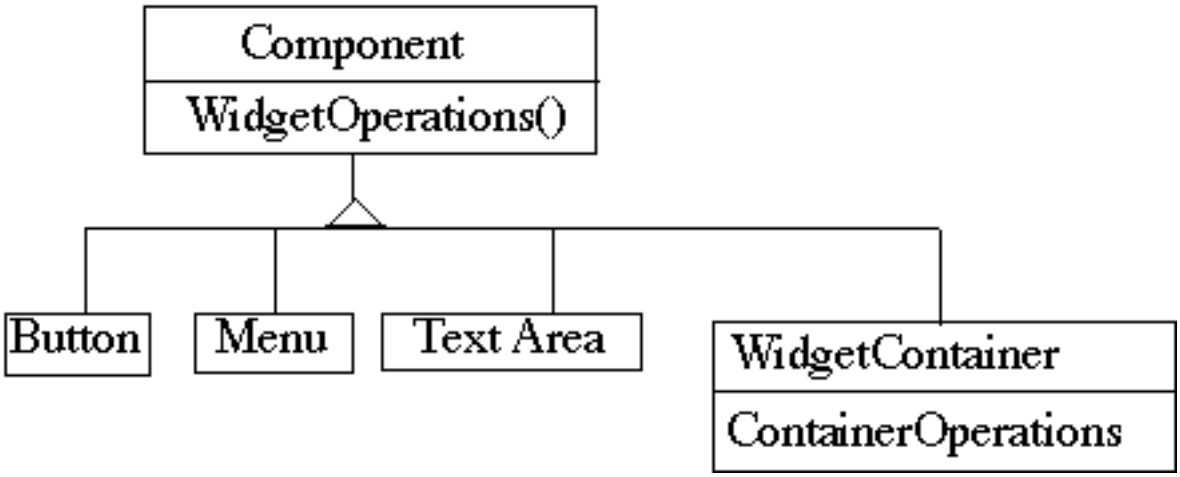
```
class Window
  {
  Buttons[] myButtons;
  Menus[] myMenus;
  TextAreas[] myTextAreas;
  WidgetContainer[] myContainers;

  public void update()
    {
    if ( myButtons != null )
      for ( int k = 0; k < myButtons.length(); k++ )
        myButtons[k].refresh();
    if ( myMenus != null )
      for ( int k = 0; k < myMenus.length(); k++ )
        myMenus[k].display();
    if ( myTextAreas != null )
      for ( int k = 0; k < myButtons.length(); k++ )
        myTextAreas[k].refresh();
    if ( myContainers != null )
      for ( int k = 0; k < myContainers.length(); k++ )
        myContainers[k].updateElements();
    etc.
    }
  public void fooOperation()
    {
    if ( blah ) etc.
    }
  }
```

A Better Idea - Program to an interface




```
class Window
  {
  GUIWidgets[] myWidgets;
  WidgetContainer[] myContainers;

  public void update()
    {
    if ( myWidgets != null )
      for ( int k = 0; k < myWidgets.length(); k++ )
        myWidgets[k].update();
    if ( myContainers != null )
      for ( int k = 0; k < myContainers.length(); k++ )
        myContainers[k].updateElements();
    etc.
    }
  }
```

The Composite Pattern



Component implements default behavior for widgets when possible

Button, Menu, etc overrides Component methods when needed

WidgetContainer will have to overrides all widgetOperations

```
class WidgetContainer
  {
  Component[] myComponents;

  public void update()
    {
    if ( myComponents != null )
      for ( int k = 0; k < myComponents.length(); k++ )
        myComponents[k].update();
    }
  }
```

Issue: WidgetContainer Operations

WidgetContainer operations tend to relate to adding, deleting and managing widgets

Should the WidgetContainer operations be declared in Component?

Pro - Transparency

Declaring them in the Component gives all subclasses the same interface

All subclasses can be treated alike. (?)

Con - Safety

Declaring them in WidgetContainer is safer

Adding or removing widgets to non-WidgetContainers is an error

What should be the proper response to adding a TextArea to a button? Throw an exception?

One out is to check the type of the object before using a WidgetContainer operation

Explicit Parent References

Aid in traversing the structure

```
class WidgetContainer
  {
  Component[] myComponents;

  public void update()
    {
    if ( myComponents != null )
      for ( int k = 0; k < myComponents.length(); k++ )
        myComponents[k].update();
    }
  public add( Component aComponent )
    {
    myComponents.append( aComponent );
    aComponent.setParent( this );
    }
  }

class Button extends Component
  {
  private Component parent;
  public void setParent( Component myParent)
    {
    parent = myParent;
    }

  etc.
  }
```

More Issues

Should Component implement a list of Components?

- The button etc. will have a useless data member

Child ordering is important in some cases

Who should delete components?

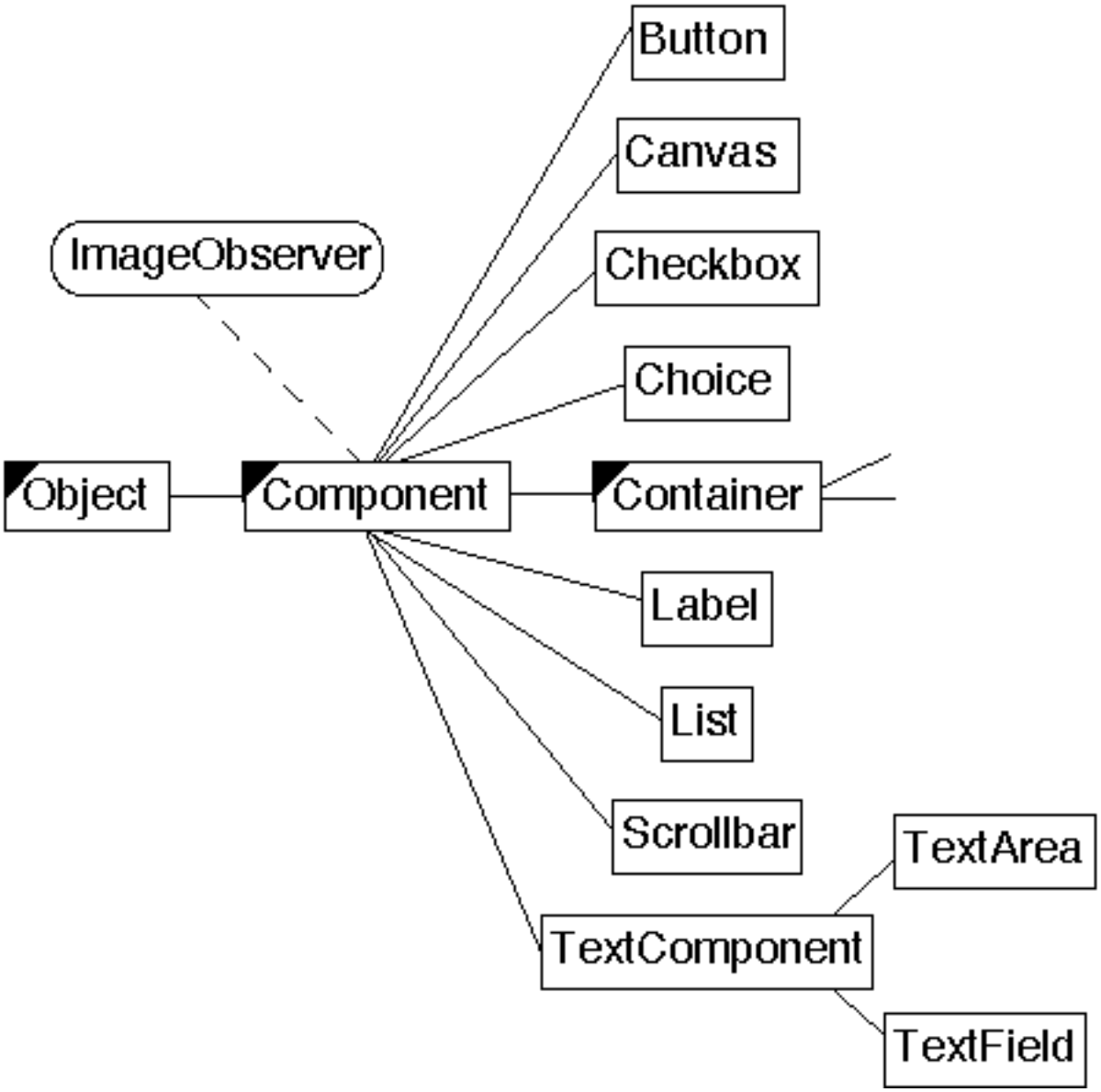
- If there is no garbage collection Container is best bet

Applicability

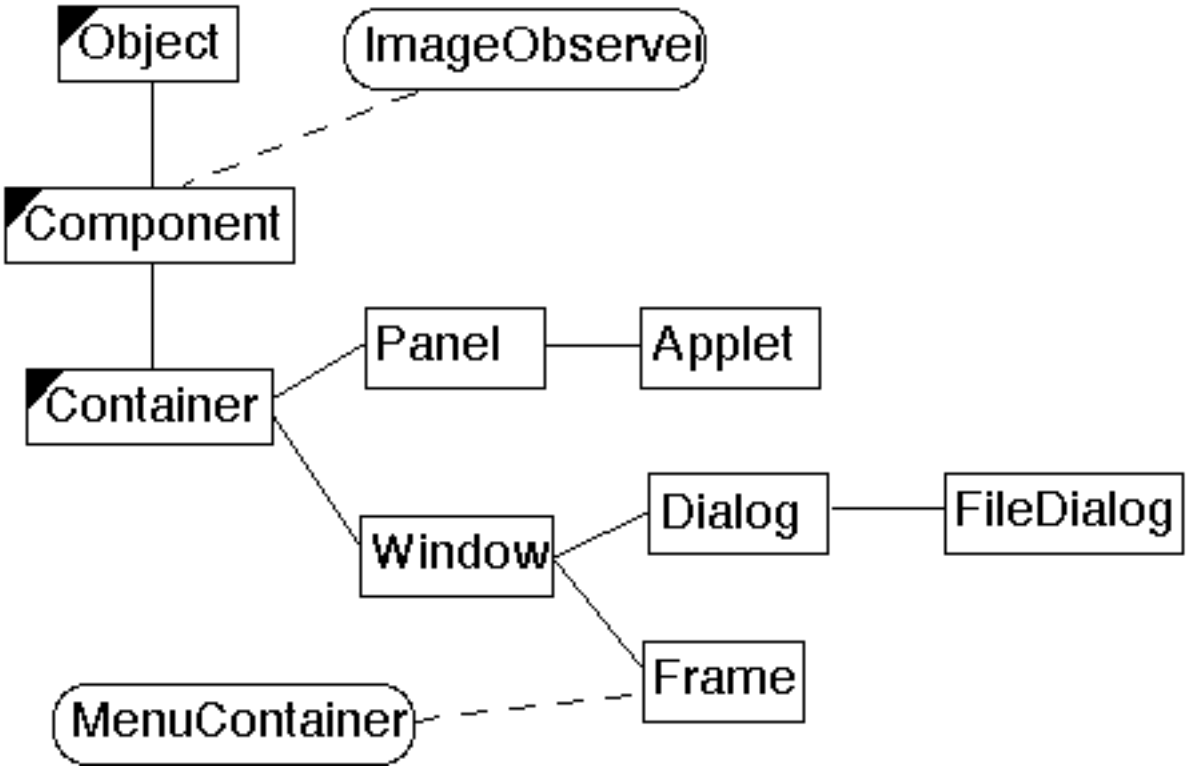
Use Composite pattern when you want

- To represent part-whole hierarchies of objects
- Clients to be able to ignore the difference between compositions of objects and individual objects

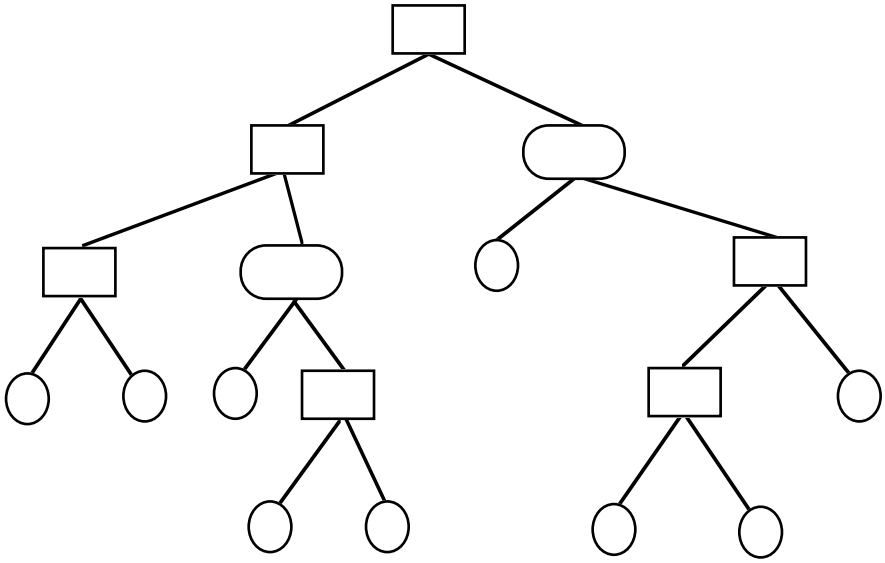
Java Use of Composite - AWT Widgets



Specialized Java Containers



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

class BinaryTreeLeaf extends Node {
   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.
}
```

Double Dispatch

Note that a visit to one node requires two method calls

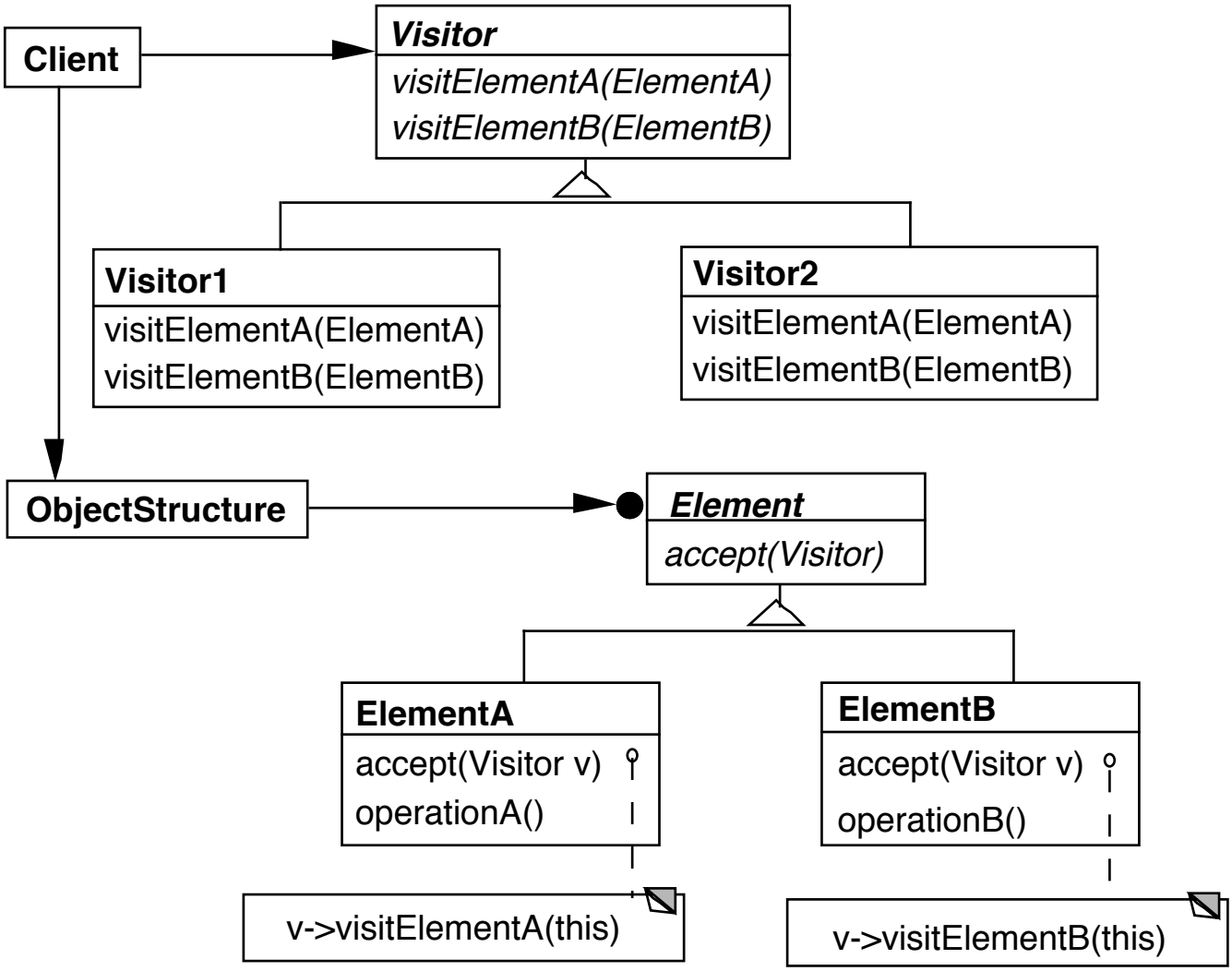
```
Node example = new BinaryTreeLeaf();

Visitor traveler = new HTMLPrintVisitor();

example.accept( traveler );
```

example.accept() calls aVisitor.visitBinaryTreeNode(this);

The first method selects the correct method in the Visitor class

The second method selects the correct Visitor class

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 cannot 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

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

Smalltalk Example

Use reflection to avoid the accept method in the elements

```
Visitor>>visit: anElement
  self
    perform: ('visit' , anElement class name , ':') asSymbol
    with: anElement
```

Aspect Oriented Programming

AspectJ provides a way around the use of Element accept methods in aspect oriented Java

AspectS provides a similar process for Smalltalk