

CS 635 Advanced Object-Oriented Design & Programming
Spring Semester, 2009
Doc 5 Strategy & Visitor
Feb 9, 2010

Copyright ©, All rights reserved. 2010 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.

References

Design Patterns: Elements of Resuable Object-Oriented Software,
Gamma, Helm, Johnson, Vlissides, Addison-Wesley, 1995, pp. 175-184, 315-324

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

Magritte Meta-Described Web Application Development, Lukas Renggli, June 2006,
Master Thesis Universität Bern, <http://www.iam.unibe.ch/~scg/Archive/Diploma/Reng06a.pdf>

Refactoring to Patterns, Kerievsky, 2005

Photographs used with permission from www.istockphoto.com

Favor
Composition
over
Inheritance

Orderable List

Sorted

Reverse Sorted

Random

OrderableList

SortedList

ReverseList

RandomList

One size does not fit all



Issue 1 - Orthogonal Features

Order

Sorted

Reverse Sorted

Random

Threads

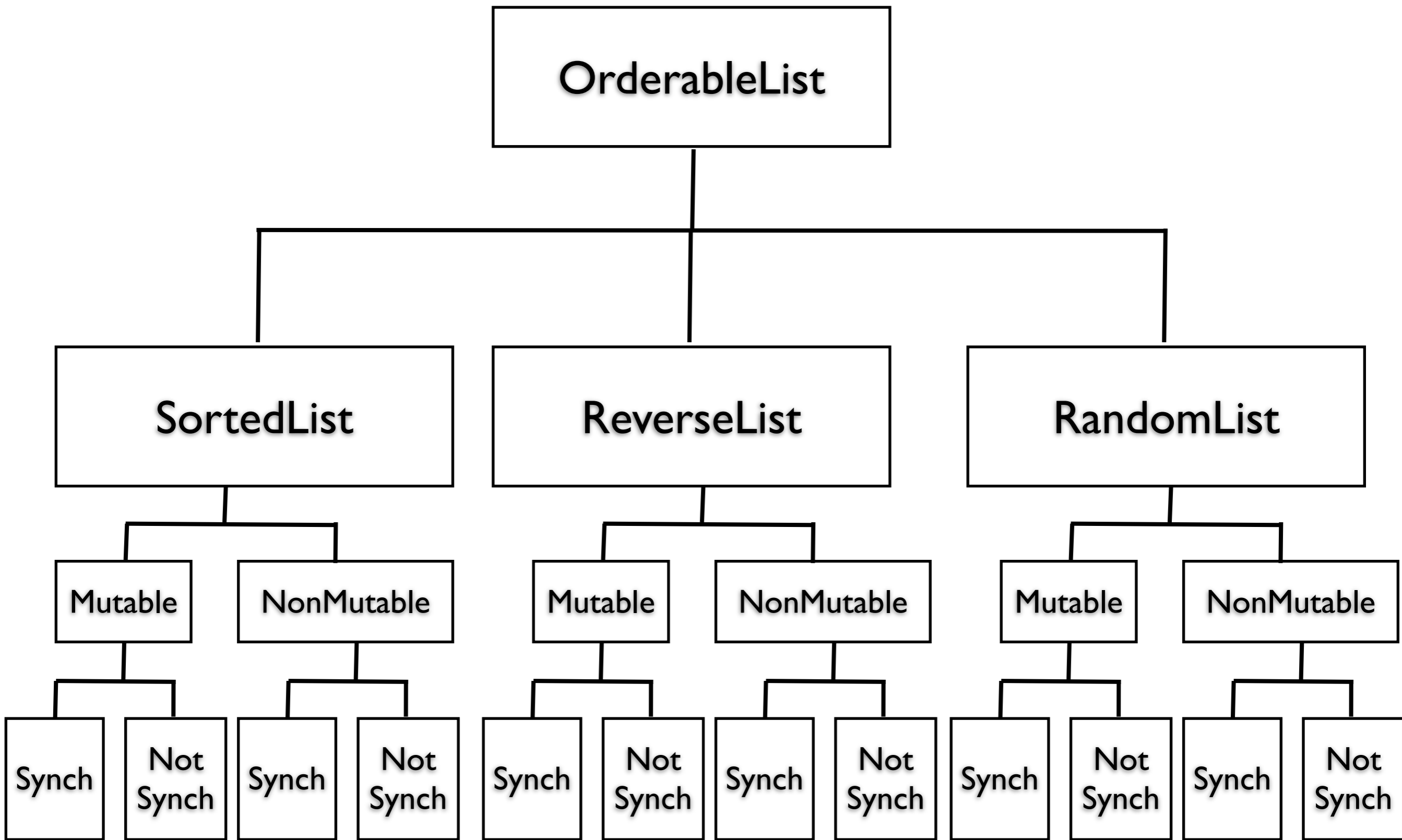
Synchronized

Unsynchronized

Mutability

Mutable

Non-mutable



Issue 2 - Flexibility



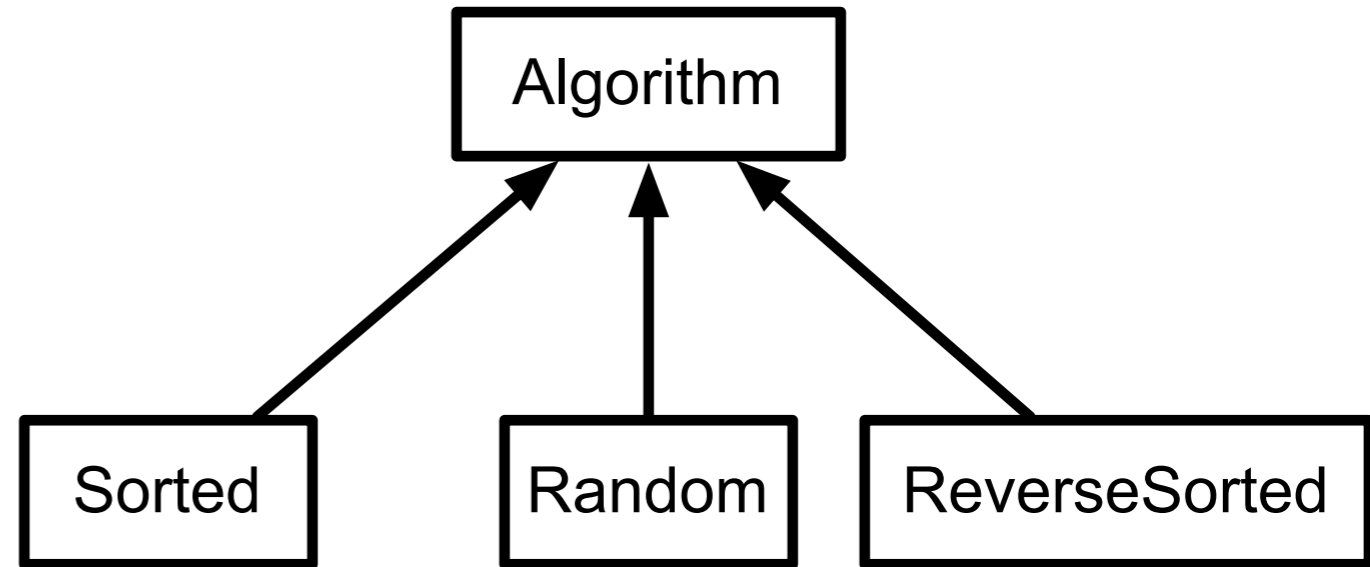
Change behavior at runtime

```
OrderableList x = new OrderableList();  
x.makeSorted();  
x.add(foo);  
x.add(bar);  
x.makeRandom();
```

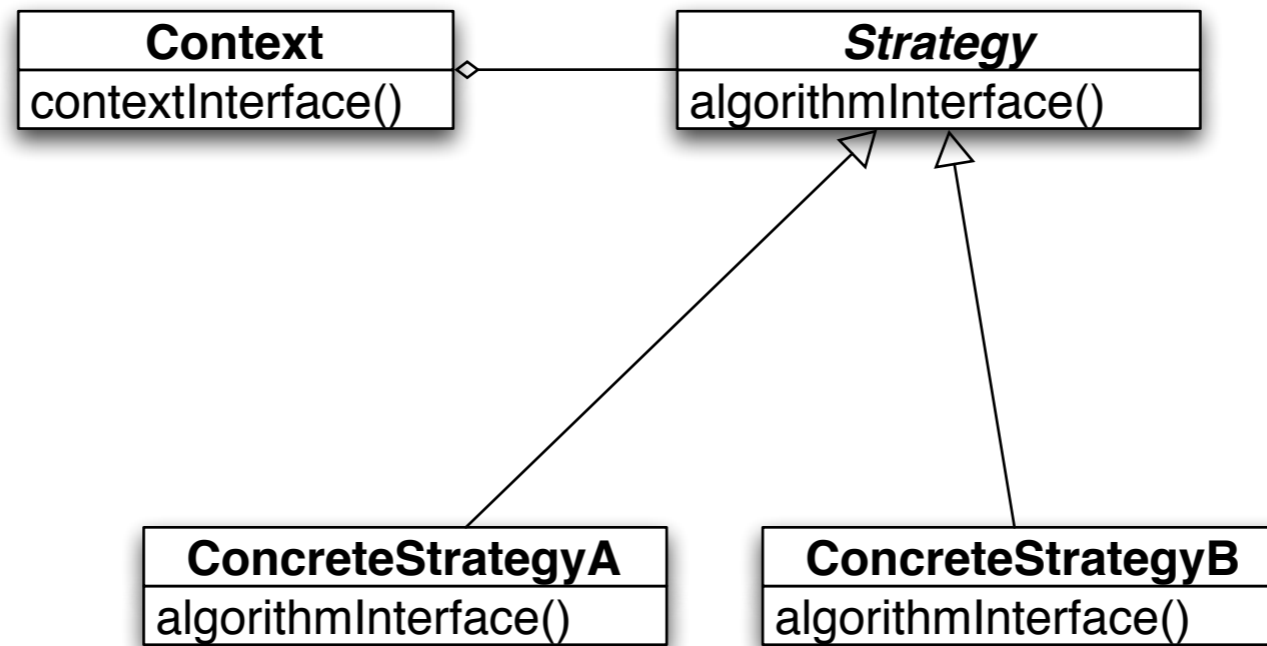
Configure objects behavior at runtime

Strategy Pattern

```
class OrderableList {  
    private Object[ ] elements;  
    private Algorithm orderer;  
  
    public OrderableList(Algorithm x) {  
        orderer = x;  
    }  
  
    public void add(Object element) {  
        elements = ordered.add(elements,element);  
    }  
}
```



Structure



The algorithm is the operation

Context contains the data

How does this work?

Prime Directive Data + Operations



How does Strategy Get the Data?

Pass needed data as parameters in strategy method

Give strategy object reference to context

Strategy extracts needed data from context

Example - Java Layout Manager

```
import java.awt.*;
class FlowExample extends Frame {

    public FlowExample( int width, int height ) {
        setTitle( "Flow Example" );
        setSize( width, height );
        setLayout( new
FlowLayout( FlowLayout.LEFT) ) );

        for ( int label = 1; label < 10; label++ )
            add( new
Button( String.valueOf( label ) ) );
        show();
    }

    public static void main( String args[] ) {
        new FlowExample( 175, 100 );
        new FlowExample( 175, 100 );
    }
}
```

Example - Smalltalk Sort blocks

```
| list |
```

```
list := #( 1 6 2 3 9 5 ) asSortedCollection.
```

```
Transcript
```

```
    print: list;
```

```
    cr.
```

```
list sortBlock: [:x :y | x > y].
```

```
Transcript
```

```
    print: list;
```

```
    cr;
```

```
    flush.
```

Costs

Clients must be aware of different Strategies

Communication overhead between Strategy and Context

Increase number of objects

Benefits

Alternative to subclassing of Context

Eliminates conditional statements

Replace in Context code like:

```
switch ( flag ) {  
    case A: doA(); break;  
    case B: doB(); break;  
    case C: doC(); break;  
}
```

With code like:

```
strategy.do();
```

Gives a choice of implementations

Refactoring: Replace Conditional Logic with Strategy

Conditional logic in a method controls which of several variants of a calculation are executed

so

Create a Strategy for each variant and make the method delegate the calculation to a Strategy instance

Replace Conditional Logic with Strategy

```
class Foo {  
    public void bar() {  
        switch ( flag ) {  
            case A: doA(); break;  
            case B: doB(); break;  
            case C: doC(); break;  
        }  
    }  
}
```



```
class Foo {  
    private strategy;  
    public void bar() {  
        strategy.do(data);  
    }  
}
```

Visitor Pattern

Linked List Assignment

Print out the elements that start with a vowel.

How to satisfy the requirements and still maintain Linked List abstraction?

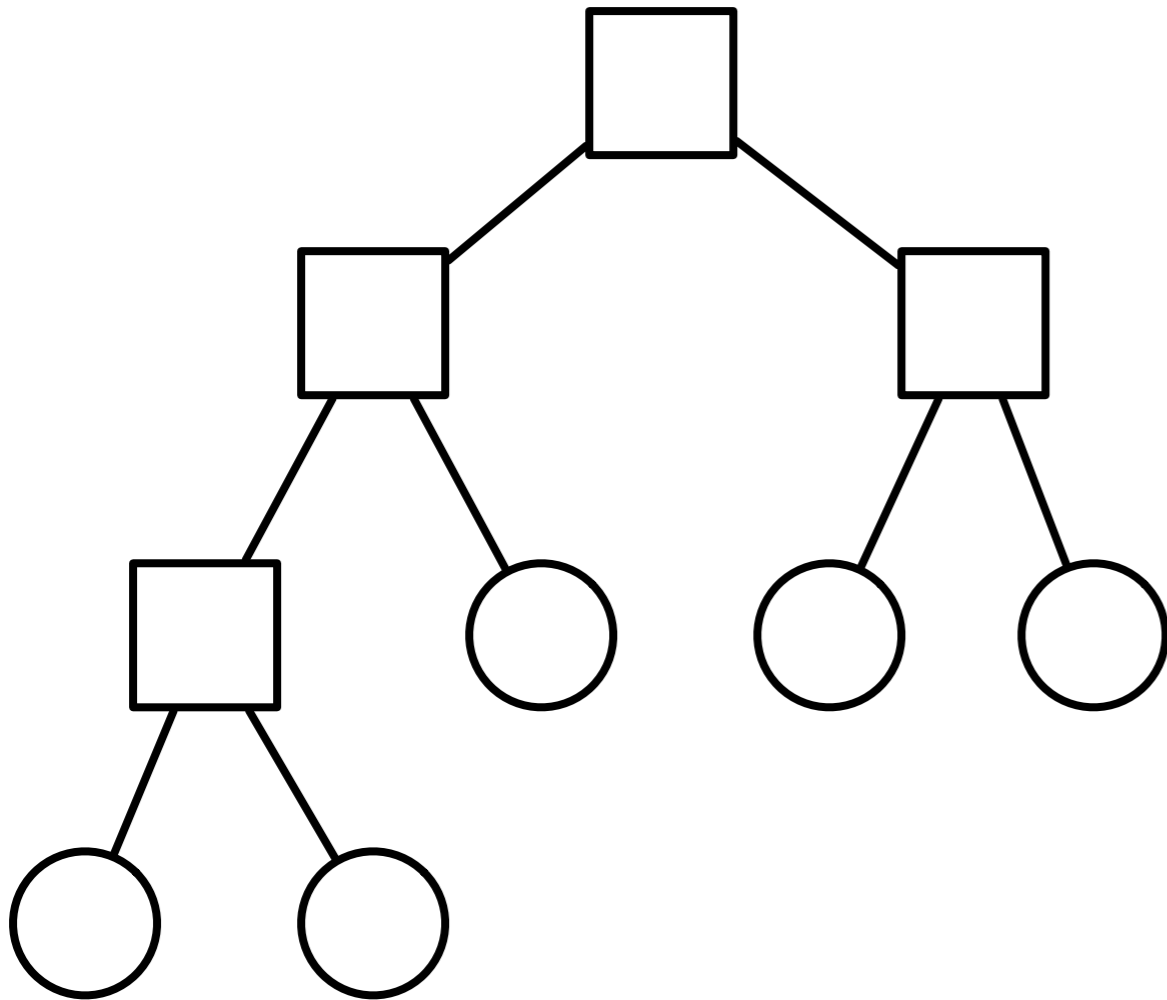
Visitor

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 BinaryTreeNode extends Node {...}
```

```
class BinaryTreeLeaf extends Node {...}
```

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){ ...}  
}
```

Put operations into separate object - a visitor

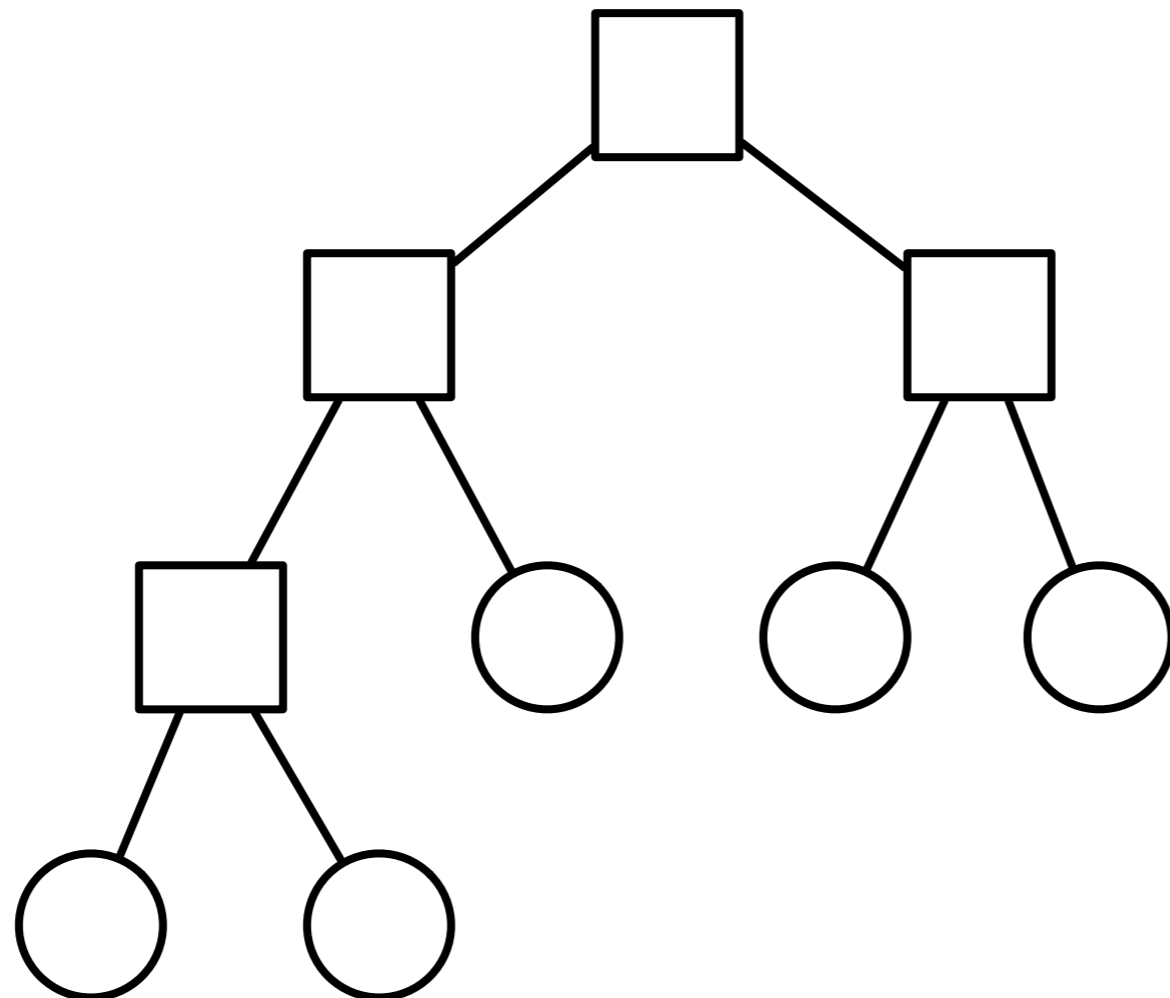
Pass the visitor to each element in the structure

The element then activates the visitor

Visitor performs its operation on the element

Each visitX method only deals with one type of element

Tree Example



Visitor

Double Dispatch

Note that a visit to one node requires two method calls

```
Node example = new BinaryTreeNode();  
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

Issue - Who does the traversal?

Visitor

Elements in the Structure

Iterator

What is Wrong with This?

```
class Node {  
    public void accept(Visitor aVisitor) {  
        aVisitor.visit( this );  
    }  
}
```

```
abstract class Visitor {  
    abstract void visit( Node );  
}
```

```
class HTMLPrintVisitor extends Visitor {  
    public void visit( Node x ) {  
        if x is BinaryTreeNode {  
            blah  
        }  
        else if x is BinaryTreeLeaf {  
            more blah  
        }  
    }  
}
```

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

Why not use one of this instead of the Visitor?

```
package example;
class BinaryTree {
    public Iterator iterator() {...}
    ...
}

class DoFoo {
    Iterator elements;
    public DoFoo(BinaryTree tree) {
        elements = tree.iterator();
    }

    public void doIt() {
        while (elements.hasNext() ) {
            Integer next = (Integer)
elements.next();
            do foo here with next
        }
    }
}
```

Magritte

Web applications have data (domain models)

We need to

- Display the data

- Enter the data

- Validate data

- Store Data

Magritte

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
```

Magritte

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:

Office Address:

Picture: no file selected

Birthday:

Age:

[Kind](#) [Number](#)

Phone Numbers: The report is empty.

[New Session](#) [Configure](#) [Toggle Halos](#) [Profile](#) [Terminate](#) [XHTML](#) 56/0 ms

Refactoring: Move Accumulation to Visitor

A method accumulates information from heterogenous classes

so

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