CS 635 Advanced Object-Oriented Design & Programming
Spring Semester, 2013
Doc 5 Assignment 1, Strategy, Visitor
Feb 5, 2013

About your Grades

Assignment 1 counts 1% of your grade

So even if you scored 0% you still can get a A

Struct -4

Utility Method -4

Functions -6

C-ish Solution

```
typedef
struct data {
    char letter;
    TrieNode[26] child;
    } * TrieNode;
```

```
void addWord(TrieNode node, String word) {
    if (word.length() == 0 ) return;
    char c = word.charAt(0);
    int index = c - 'a';
    if (node->child[index] == null) {
        node->child[index] = malloc(sizeof(TrieNode));
        node->child[index]->letter = c;
    }
    addWord(node->child[index], word.substring(1));
}
```

Translating C-ish Solution

```
typedef
struct data {
    char letter;
    TrieNode[26] child;
} * TrieNode;
```

```
void addWord(TrieNode node, String word) {
    if (word.length() == 0 ) return;
    char c = word.charAt(0);
    int index = c - 'a';
    if (node->child[index] == null) {
        node->child[index] = malloc(sizeof(TrieNode));
        node->child[index]->letter = c;
    }
    addWord(node->child[index], word.substring(1));
}
```

```
class TrieNode {
    char letter;
    TrieNode[26] child;
}
```

```
class Trie {
    public static void addWord(TrieNode node, String word) {
        if (word.length() == 0 ) return;
        char c = word.charAt(0);
        int index = c - 'a';
        if (node.child[index] == null) {
            node.child[index] = new TrieNode();
            node.child[index].letter = c;
        }
        addWord(node.child[index], word.substring(1));
    }
```

Keep Data and operations together

```
class TrieNode {
    char letter;
    TrieNode[26] child;

    public TrieNode(char letter) {
        this.letter = letter;
    }

    public void addWord(String word) {
        if (word.length() == 0 ) return;
        char c = word.charAt(0);
        int index = c - 'a';
        if (child[index] == null)
            child[index] = new TrieNode(c);
        child[index].addWord(word.substring(1));
    }
```

Keep Data and operations together

```
class TrieNode {
    char letter;
    TrieNode[26] child;

    public TrieNode(char letter) {
        this.letter = letter;
    }
```

```
public void addWord(String word) {
    if (word.length() == 0 ) return;
    TrieNode forWord = nodeFor(word);
    forWord.addWord(word.substring(1));
}

private TrieNode nodeFor(String word) {
    char c = word.charAt(0);
    int index = c - 'a';
    if (child[index] == null)
        child[index] = new TrieNode(c);
    return child[index];
}
```


We can use the nodeFor method in search. Now the code is a bit clearer. Now it is clearer that we could use a way to know when we are at the end of a word.

Keep Data and operations together

```
class TrieNode {
    char letter;
    TrieNode[26] child;
    boolean isEndOfWord;

    public TrieNode(char letter) {
        this.letter = letter;
        isEndOfWord = false;
    }
```

```
public void addWord(String word) {
    if (word.length() == 0 ) {
        isEndOfWord = true;
        return;
    }
    TrieNode forWord = nodeFor(word);
    forWord.addWord(word.substring(1));
}
```

```
private TrieNode nodeFor(String word) {
    char c = word.charAt(0);
    int index = c - 'a';
    if (child[index] == null)
        child[index] = new TrieNode(c);
    return child[index];
}
```


Now it is clear that it is not so clear when to set isEndOfWord.

Functions -6

```
class TrieNode {
    char letter;
    TrieNode[26] child;
}
```

```
class TrieOperations {
    public static void addWord(TrieNode n, String s) {
        Blah

    }
}
```

static -3

```
public class Trie {

    private static TrieNode root;

    public static void addWord(TrieNode n, String s) {
        Blah
        }

}
```

Information Hiding -4

```
public class Trie {

    private TrieNode root;

    public void addWord(TrieNode n, String s) {
        Blah
        }

}
```

Names -2

In Java the method to add to a collection is add

So we should use:

```
class Trie {
    boolean add(String word) { blah }
    boolean contains(Object o) { blah }
```

Names -4

Structure of Java names

ClassName
methodName
fieldName
localVariable

Not Field -3

```
public class Trie {
    private TrieNode root;
    private int wordCount;
    private int searchCount;
    private char[] wordContainingPhrase;
    private boolean match;
```

Hardcoded c & k -3

```
class Trie {
    boolean foundC;
    boolean foundMatch;
    public void findWordsMatchingPattern(TrieNode node) {
        if (node.isWord()) && foundMatch)
            System.out.println( node.word());
        if ((node.letter == 'k') && foundC)
            foundMatch = true;
        if (node.letter == 'c')
            foundC = true;
        else
            foundC = false;
        for (Node n : children)
            findWordsMatchingPattern(n);
        blah
    }
```


this code does not work. It is just here to help illustrate a few points

System.out.print -3

```
class Trie {
    boolean foundC;
    boolean foundMatch;
    public void findWordsMatchingPattern(TrieNode node) {
        if (node.isWord()) && foundMatch)
            System.out.println( node.word());
        if ((node.letter == 'k') && foundC)
            foundMatch = true;
        if (node.letter == 'c')
            foundC = true;
        else
            foundC = false;
        for (Node n : children)
            findWordsMatchingPattern(n);
        blah
    }
```

Duh -3

```
class Trie {
    //fields
    TrieNode root;              //root node

    //Constructor
    public Trie() {
        root = new TrieNode();
    }

    //methods

    //add a word to the trie
    public void addWord(String word) {
        //Ask the node add the word
        root.addWord(word);
    }
```

Long Methods -3

Methods should not be pages long

Comments

Comment the why not the what

The what should be clear from the code

If what is not clear try rewrite the code to make it clear

Use descriptive names
Add methods so can give name to blocks of code

System.exit(0) -1

```
try {
      blah
} catch (Exception error) {
      error.printStackTrace();
}
```

Strategy Pattern

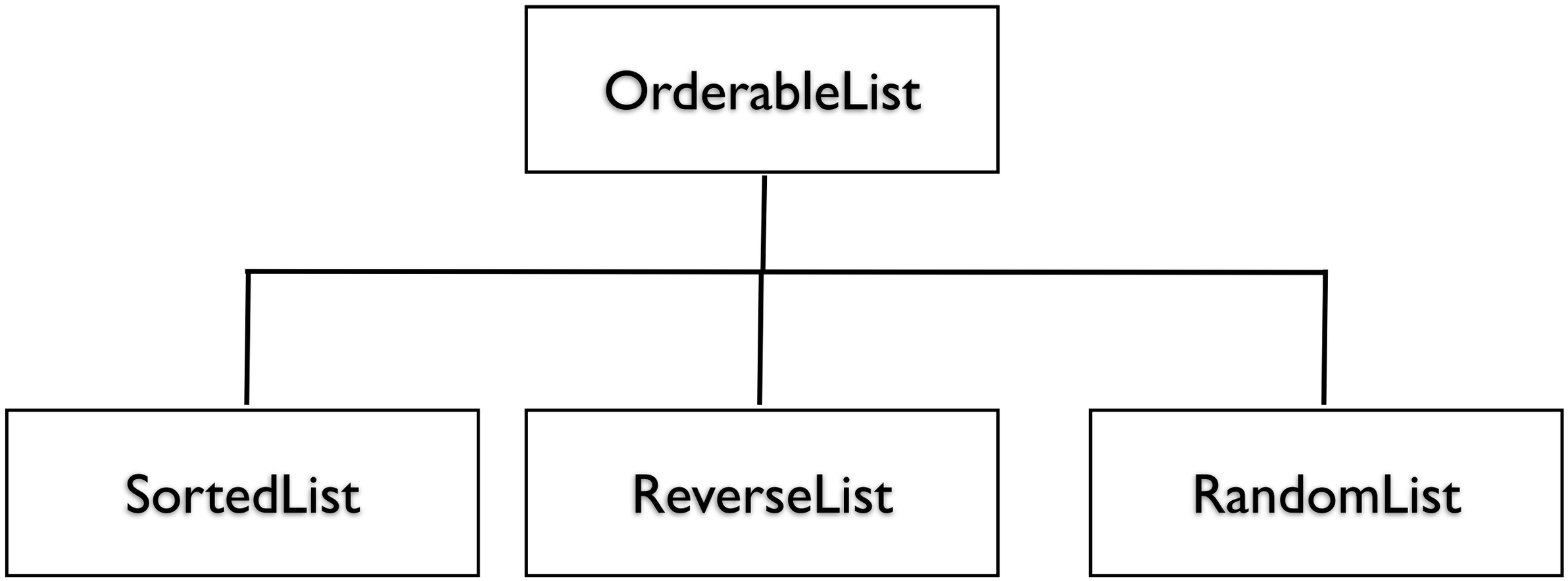
Favor
Composition
over
Inheritance

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

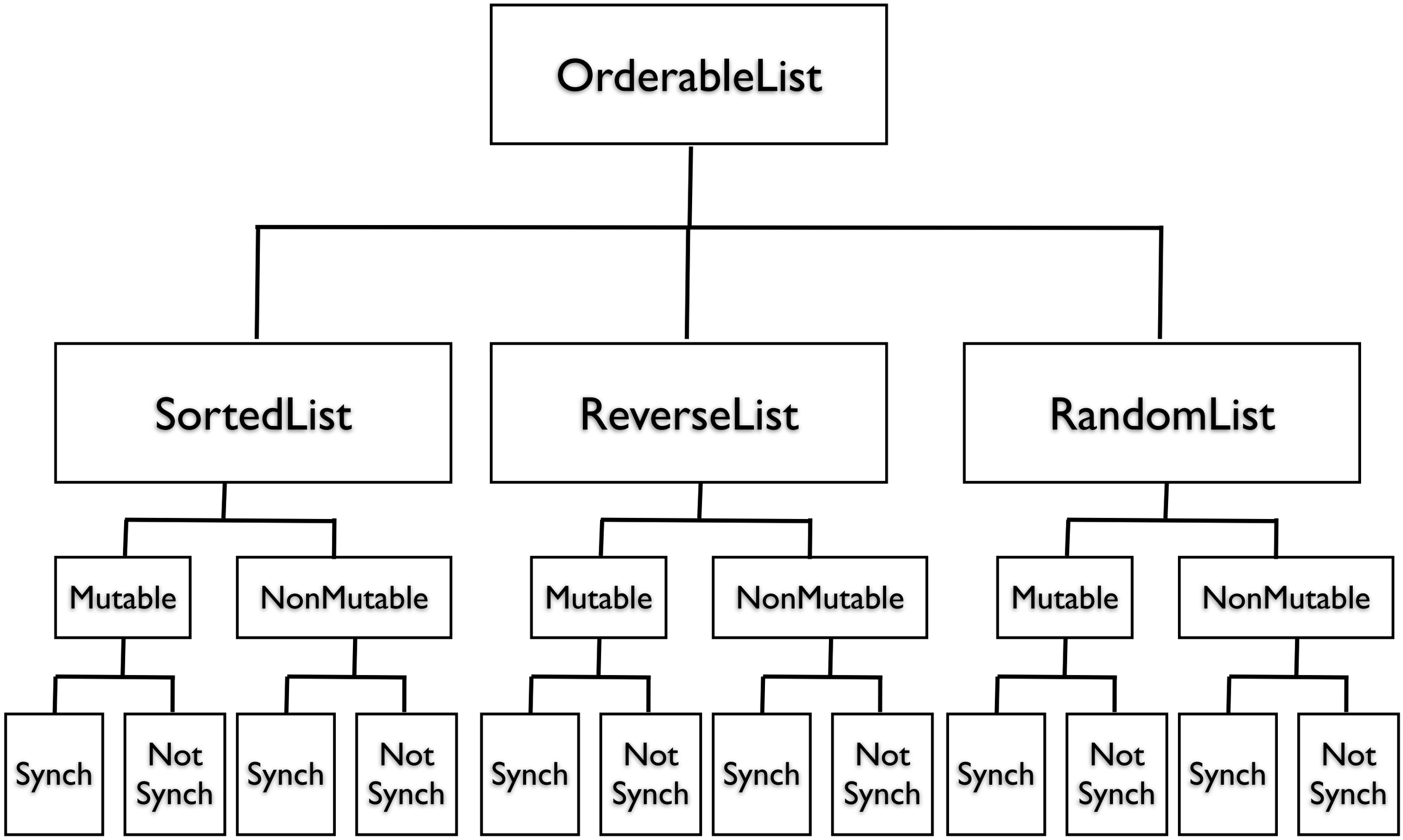
Synchronized

Unsynchronized

Mutability

Mutable

Non-mutable


OrderableList
SortedList
ReverseList
RandomList
Mutable
NonMutable
Mutable
NonMutable
Mutable
NonMutable
Synch
Not Synch
Synch
Not Synch
Synch
Not Synch
Synch
Not Synch
Synch
Not Synch
Synch
Not Synch

Issue 2 - Flexibility

Change behavior at runtime

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

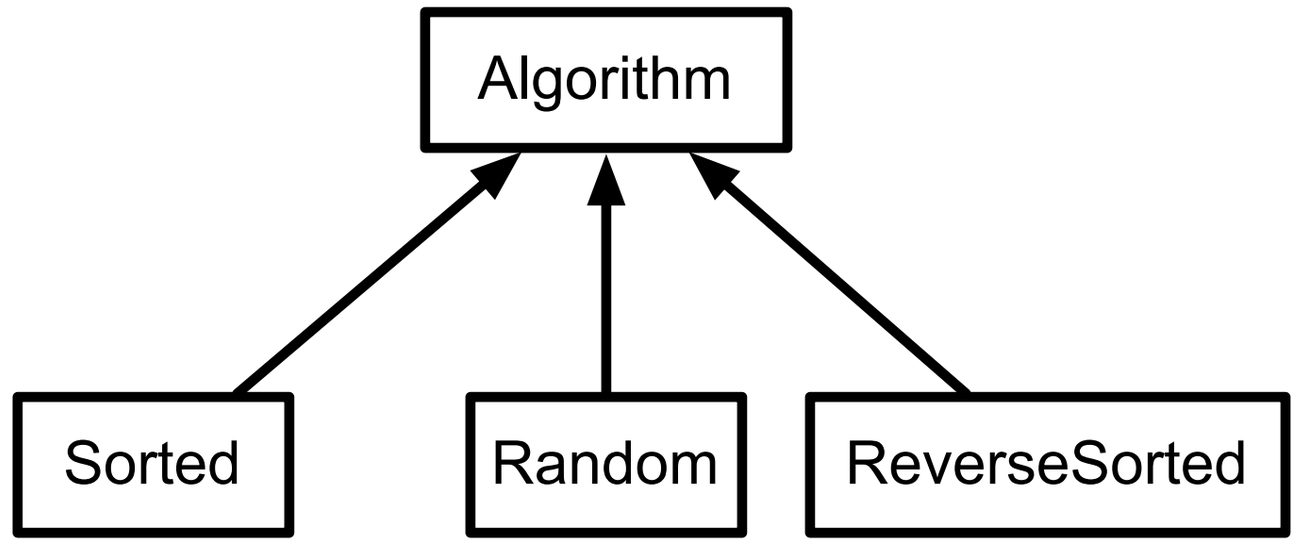
Configure objects behavior at runtime

Strategy Pattern

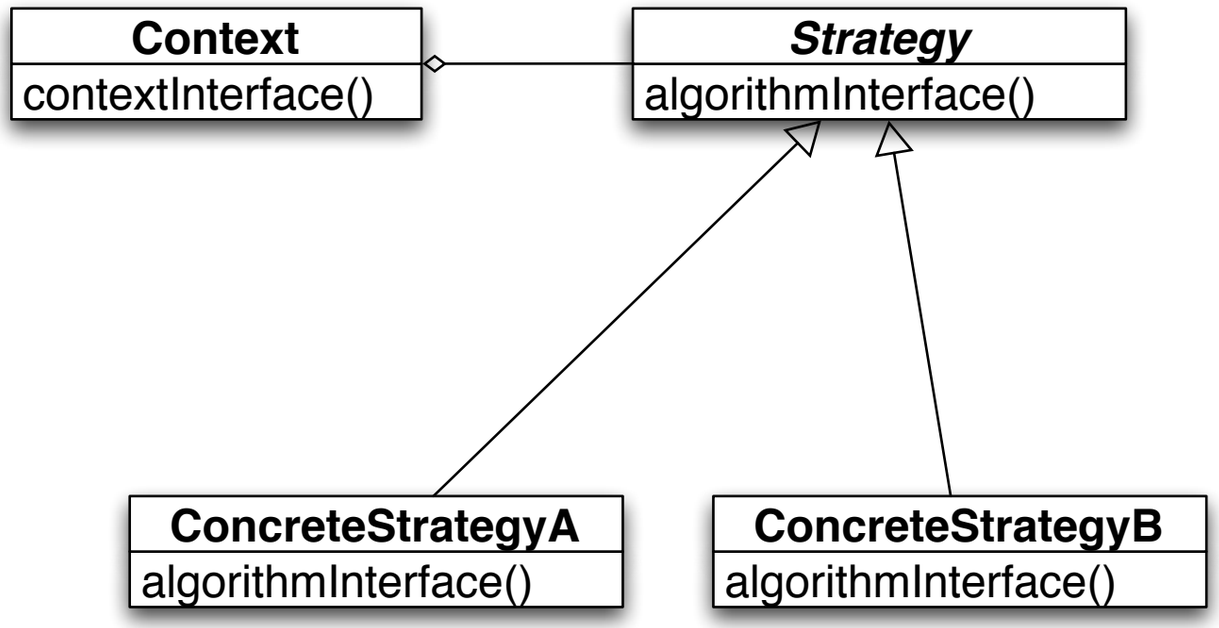
```
class OrderableList {
      private Object[ ] elements;
      private Algorithm orderer;

      public OrderableList(Algorithm x) {
            orderer = x;
      }

   public void add(Object element) {
      elements = ordered.add(elements,element);
   }
```



Structure


Context
contextInterface()
Strategy
algorithmInterface()
ConcreteStrategyA
algorithmInterface()
ConcreteStrategyB
algorithmInterface()

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);
    }
}
```


See Refactoring to Patterns, Kerievsky, 2005, pp 129-143 for detail steps

Visitor Pattern

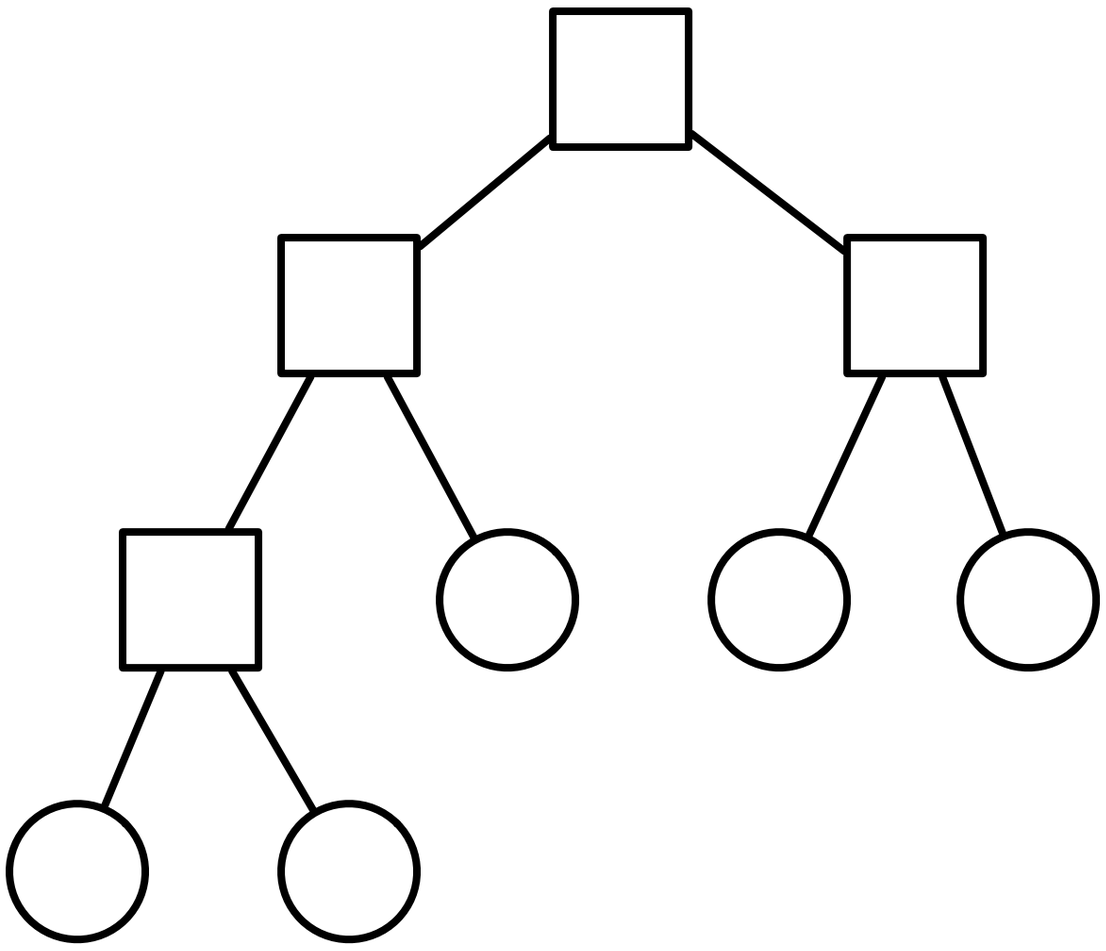
Visitor

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 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 );
        }
}
```

Put operations into separate object - a visitor

Pass the visitor to each element in the structure

The element then activates the visitor

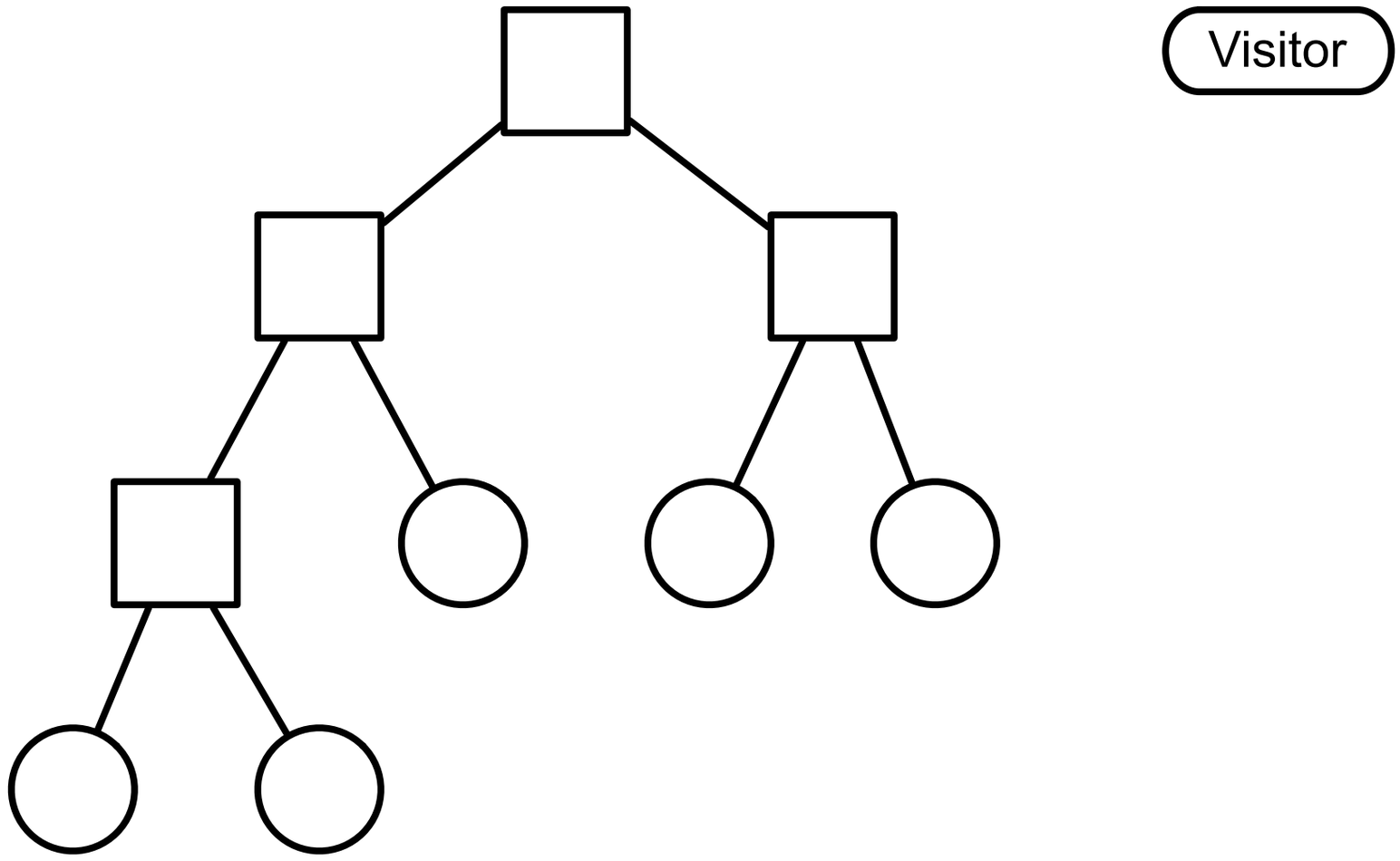
Visitor performs its operation on the element

Each visitX method only deals with on type of element

```
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



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

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 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

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 eleminates 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.
```

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

See Refactoring to Patterns, Kerievsky, 2005, pp 320-338 for details