CS 635 Advanced Object-Oriented Design & Programming
Spring Semester, 2014
Doc 6 Iterator, Visitor, Strategy
Feb 13, 2014

Iterator Pattern

Provide a way to access the elements of a collection sequentially without exposing its underlying representation

Iterator Solution

Java

```
LinkedList<Strings> strings =  new LinkedList<Strings>();

code to add strings

for (String element : strings) {
        if (element.size % 2 == 0)
                System.out.println(element);
}
```

```
Iterator<String> list = strings.iterator();
while (list.hasNext()){
        String element = list.next();
        if (element.size % 2 == 0)
                System.out.println(element);
        }
}
```


This is 1/2 the way to a good solution.

Ruby Iterator Examples

a = [1, 2, 3, 4]

a.each {|x| puts x}	1 2 3 4
result = a.collect {|x| x + 10} puts result	11 12 13 14
result = a.find_all {|x| x > 2 } puts result	3 4
puts a.any? {|x| x > 2}	true
puts a.detect {|x| x > 2 }	3

Ruby has a richer set of iterators than Java. Smalltalk, which inspired Ruby's iterators, has a richer set of iterators that Ruby. Perhaps the language that replaces Ruby will match the power that Smalltalk had 20 years ago.

Pattern Parts

Intent

Motivation

Applicability

Structure

Participants

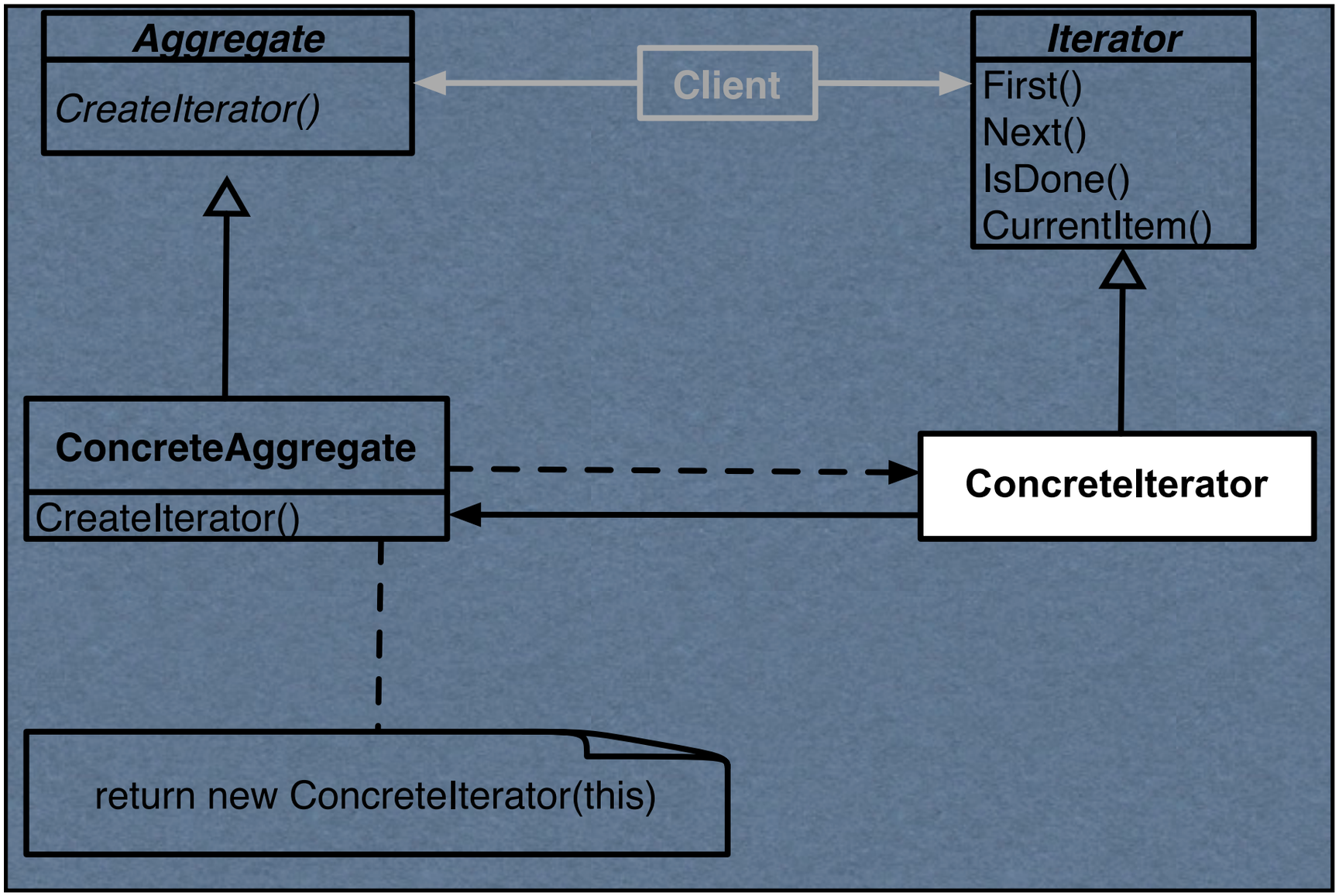
Collaborations

Consequences

Implementation

Sample Code

Iterator Structure



Issue - What is the big deal?

```
var numbers = new LinkedList();

code to add numbers

Iterator list = numbers.iterator();
while ( list.hasNext() ) {
        Integer a = (Integer) list.next();
        int    b = a.intValue();
        if ((b % 2) == 0)
                System.out.println( x );
}
```

```
var numbers = new LinkedList();

code to add numbers

for (int k =0; k < numbers.size(); k++ ) {
        Integer a = (Integer) numbers.get(k);
        int    b = a.intValue();
        if ((b % 2) == 0)
                System.out.println( x );
}
```


Java's Enumerations and iterators were awkward to use. C# pushed Sun to add better syntax.

Issues - Concrete vs. Polymorphic Iterators

Concrete

```
Reader iterator = new StringReader( "cat");
int c;
while (-1 != (c = iterator.read() ))
      System.out.println( (char) c);
```

Polymorphic

```
Vector listOfStudents = new Vector();

// code to add students not shown

Iterator list = listOfStudents.iterator();
while ( list.hasNext() )
      System.out.println( list.next() );
```

Memory leak issue in C++, Why?

Issue - Who Controls the Iteration?

External (Active)

```
var numbers = new LinkedList();

code to add numbers

Vector evens = new Vector();
Iterator list = numbers.iterator();
      while ( list.hasNext() ) {
            Integer a = (Integer) list.next();
            int    b = a.intValue();
 if ((b % 2) == 0)
                  evens.add(a);
}
```

Internal (Passive)

```
numbers  = LinkedList.new

code to add numbers

evens = numbers.find_all { |element| element.even? }
```

Issue - Who Defines the Traversal Algorithm

Object being iterated

Iterator

Issue - Robustness

What happens when items are added/removed from the iteratee while an iterator exists?

```
Vector listOfStudents = new Vector();

// code to add students not shown

Iterator list = listOfStudents.iterator();
listOfStudents.add( new Student( "Roger") );

list.hasNext();              //What happens here?
```

Visitor Pattern

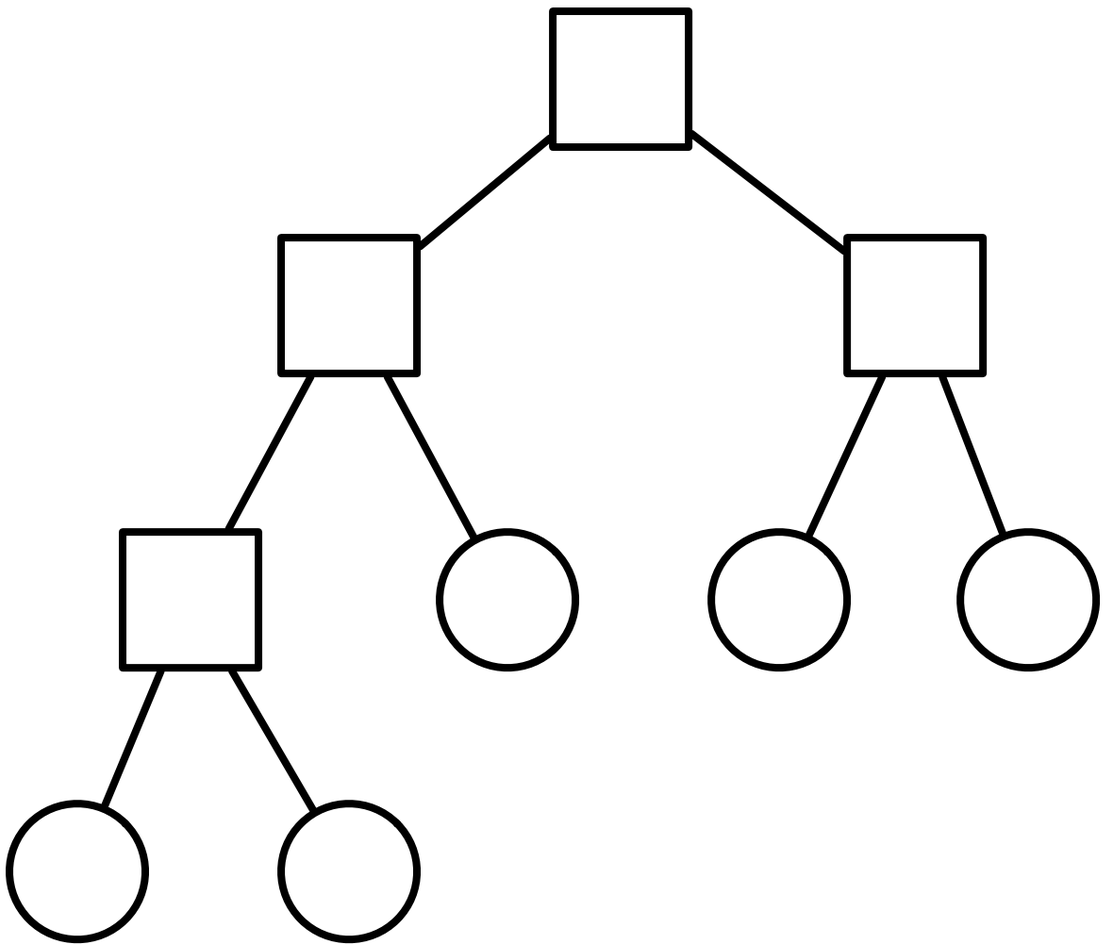
Visitor

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 {...}

class Tree { ... }

Tree Printing

HTML Print

PDF Print

TeX Print

RTF Print

Others likely in future

Operations are complex

Do different things on different types of nodes

Need to traverse tree

Not part of BST abstraction

First Attempt

Create Printer Classes

Use iterator to access all elements

Process each element

First Attempt

```
class TreePrinter {

    public String printTree (Tree toPrint) {
        Iterator nodes = toPrint.iterator();
        while (nodes.hasNext()) {
            Node current = nodes.next();
            if (current.isLeafNode())
                printLeafNode(current);
            else if (current.isInternalNode() )
                printInternalNode(current);
        }
    }

    private String printLeafNode(Node current) { blah }

    private String printInternalNode(Node current) { blah }
```

First Attempt - Issue

```
class TreePrinter {

    public String printTree (Tree toPrint) {
        Iterator nodes = toPrint.iterator();
        while (nodes.hasNext()) {
            Node current = nodes.next();
            if (current.isLeafNode())
                printLeafNode(current);
            else if (current.isInternalNode() )
                printInternalNode(current);
        }
    }

    private String printLeafNode(Node current) { blah }

    private String printInternalNode(Node current) { blah }
```

Hidden case statement

If add different type of node ...

Second Attempt - Overloaded Method

```
class TreePrinter {

    public String printTree (Tree toPrint) {
        Iterator nodes = toPrint.iterator();
        while (nodes.hasNext()) {
            Node current = nodes.next();
            printNode(current);
        }
    }

    public String printNode(BinaryTreeNode current) { blah }

    public String printNode(BinaryTreeLeaf current) { blah }
```

Overloaded Methods

Which overloaded method to run

- Selected at compile time
- Based on declared type of parameter
- Does not use runtime information

Second Attempt - Overloaded Method

```
class TreePrinter {

    public String printTree (Tree toPrint) {
        Iterator nodes = toPrint.iterator();
        while (nodes.hasNext()) {
            Node current = nodes.next();
            printNode(current);
        }
    }

    public String printNode(BinaryTreeNode current) { blah }

    public String printNode(BinaryTreeLeaf current) { blah }
```

Compile Error

Still Need case statment

Visitor pattern converts

Runtime case statement into Compile time case statement

So if add new type of Node compiler tells us i fwe forget to change case statement

Key Idea

Receiver of method is determined at runtime

```
x.toString();
```

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

Accept Method

```
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 );
      }
}
```

Visitor

```
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();
while (nodes.hasNext()) {
    Node current = nodes.next();
    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 );
        }
}
```

Put operations into separate object - a visitor

Pass the visitor to each element in the structure

The element then activates the visitor

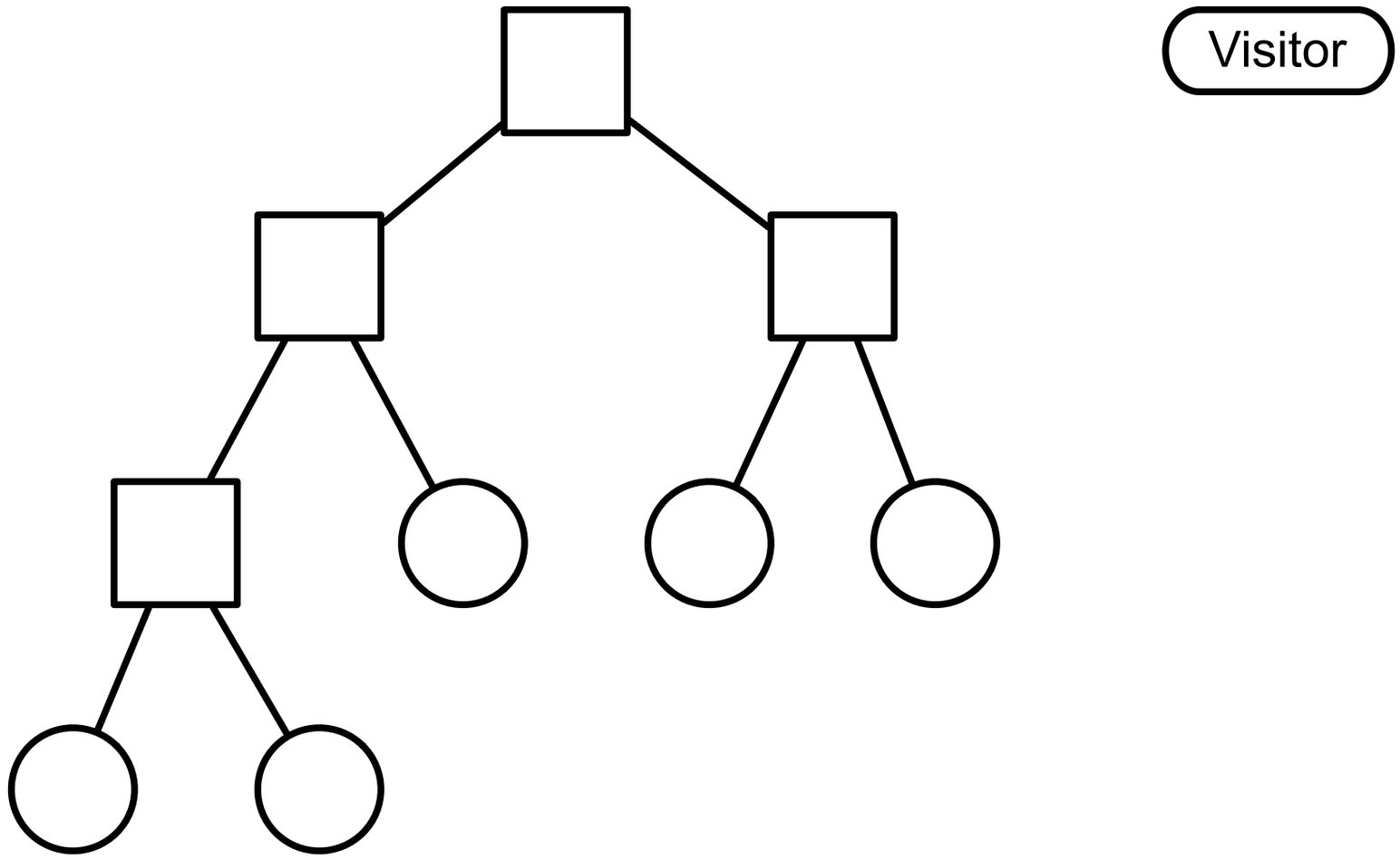
Visitor performs its operation on the element

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){ ...}
}
```

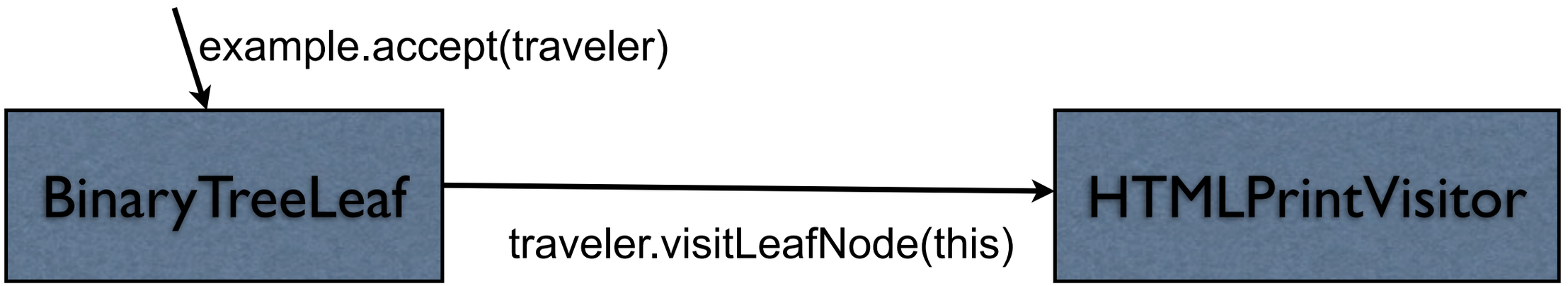
Tree Example



Double Dispatch

Note that a visit to one node requires two method calls

```
Node example = new BinaryTreeLeaf();
Visitor traveler = new HTMLPrintVisitor();
example.accept( traveler );
```



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 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

Clojure, Lisp & Multi-methods

Multi-methods in Clojure do select overloaded method

- At run-time
- Based on argument type

```
while (nodes.hasNext()) {
    Node current = nodes.next();
    printNode(current);
}
```

No need for visitor pattern

Example - 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

Strategy Pattern

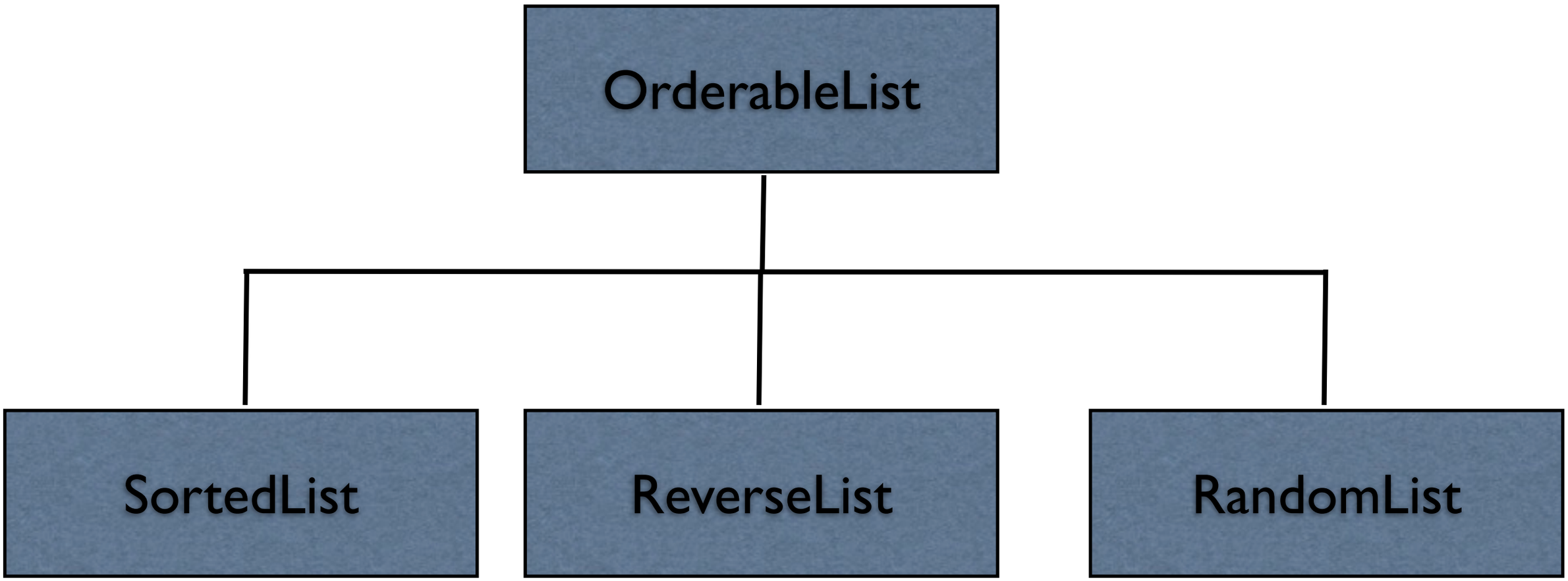
Favor
Composition
over
Inheritance

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

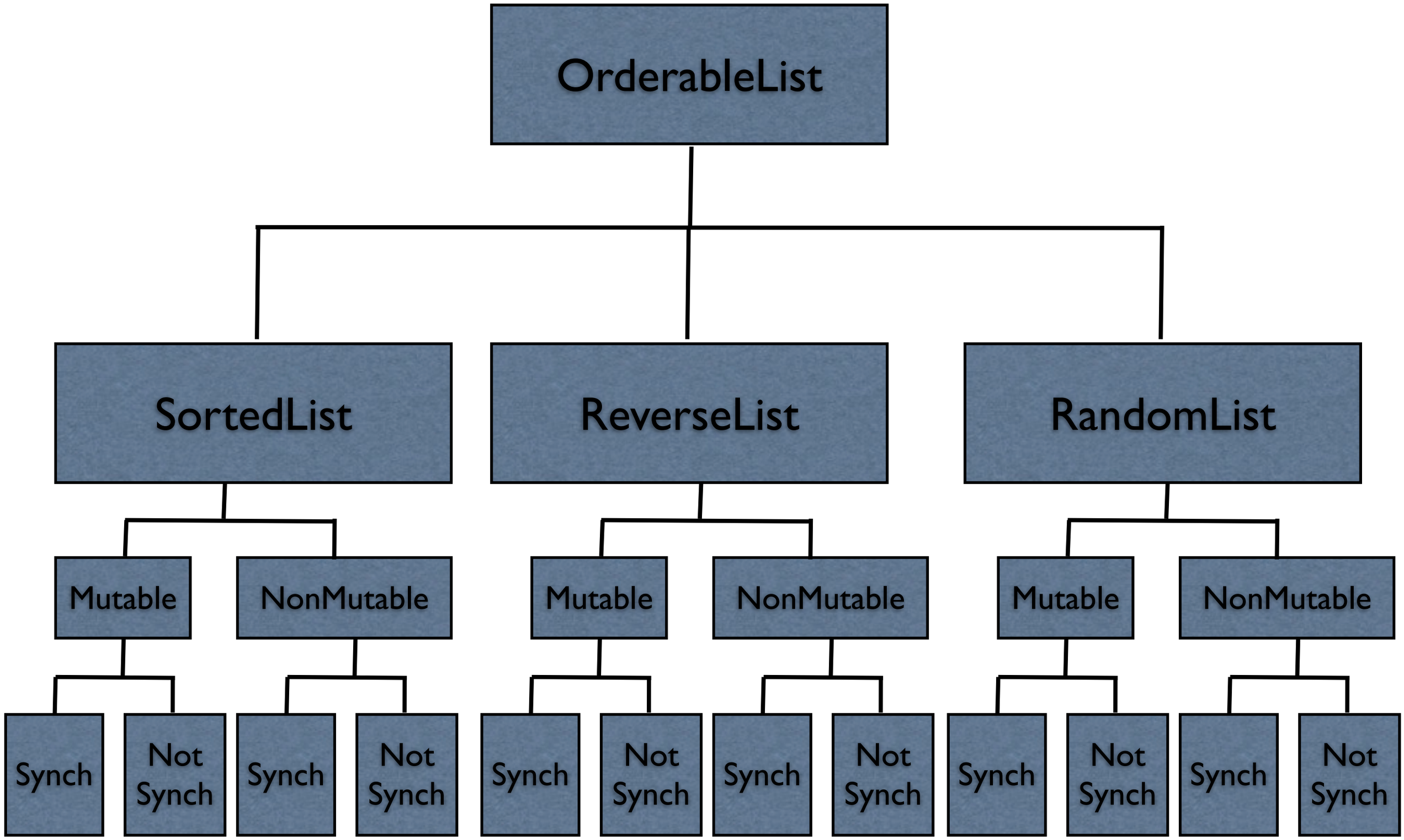
Synchronized

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();
```

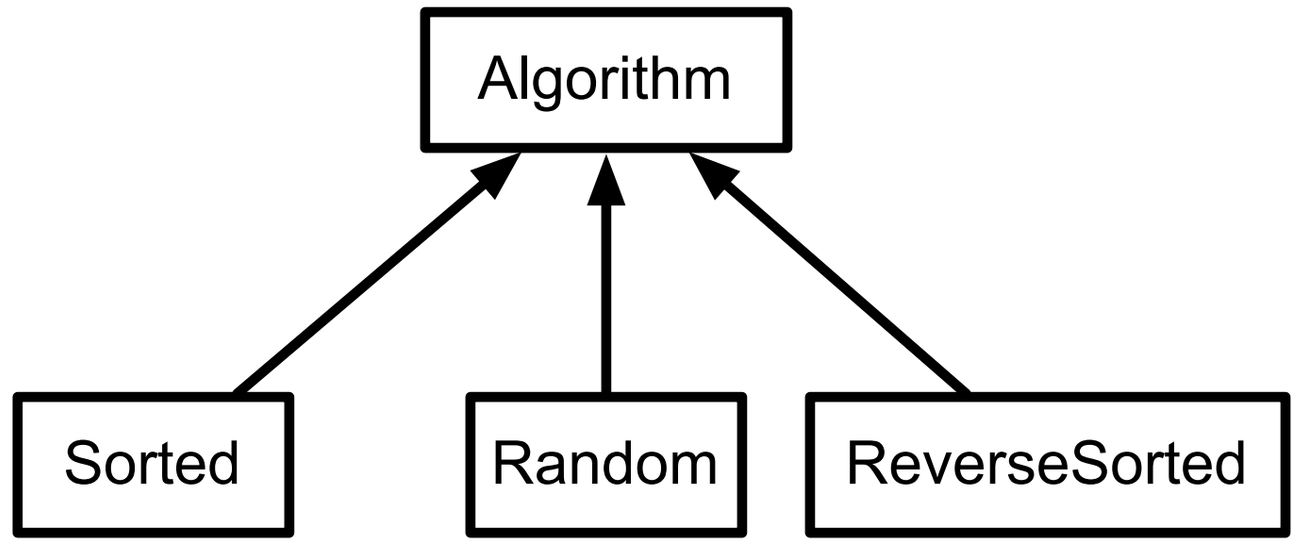
Configure objects behavior at runtime

Strategy Pattern

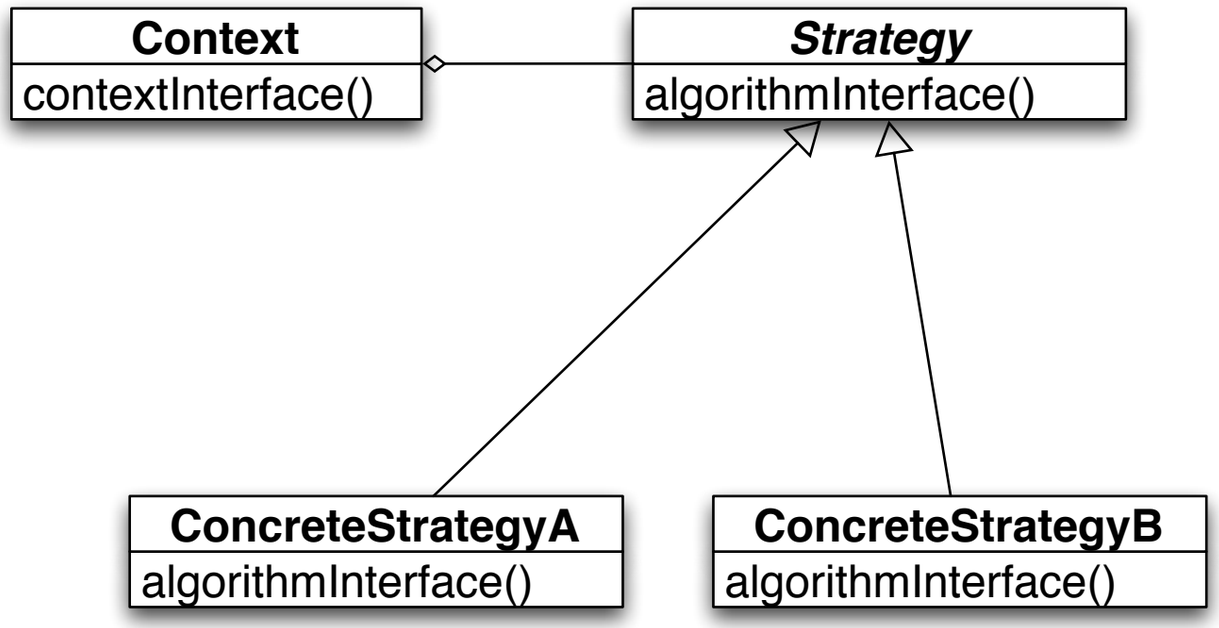
```
class OrderableList {
      private Object[ ] elements;
      private Algorithm orderer;

      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