CS 635 Advanced Object-Oriented Design & Programming
Spring Semester, 2018
Doc 8 Memento, Interpreter, Visitor
Sep 25, 2018

Undo

Undo

Some examples

Counter

```
counter.increase();        //increase counter by 1
counter.decrease();        //decrease counter by 1
```

Undo

Some examples

Text editing

Replace "Should" with "Could" at start of 3rd sentence in 5 paragraph

Undo - Some Issues

Redo

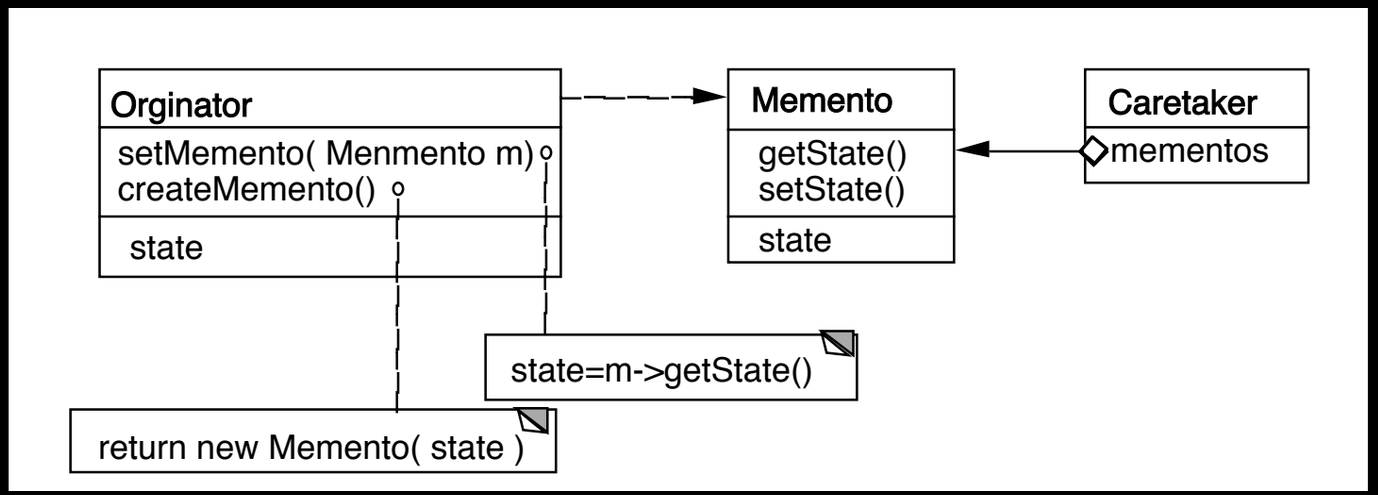
Multiple undo

Memento

Memento

Store an object's internal state, so the object can be restored to this state later without violating encapsulation

undo, rollbacks



Only originator:

- Can access Memento’s get/set state methods
- Create Memento

Example

```
package Examples;
class Memento{
    private Hashtable savedState = new Hashtable();

    protected Memento() {}; //Give some protection

    protected void setState( String stateName, Object stateValue ) {
        savedState.put( stateName, stateValue );
    }

    protected Object getState( String stateName) {
        return savedState.get( stateName);
    }

    protected Object getState(String stateName, Object defaultValue ) {
        if ( savedState.containsKey( stateName ) )
            return savedState.get( stateName);
        else
            return defaultValue;
    }
}
```

Sample Originator

```
package Examples;
class ComplexObject {
    private String name;
    private int someData;
    private Vector objectAsState = new Vector();

    public Memento createMemento() {
        Memento currentState = new Memento();
        currentState.setState( "name", name );
        currentState.setState( "someData", new Integer(someData) );
        currentState.setState( "objectAsState", objectAsState.clone() );
        return currentState;
    }

    public void restoreState( Memento oldState) {
        name = (String) oldState.getState( "name", name );
        objectAsState = (Vector) oldState.getState( "objectAsState" );
        Integer data = (Integer) oldState.getState( "someData");
        someData = data.intValue();
    }
```

Why not let the Originator save its old state?

```
class ComplexObject {
    private String name;
    private int someData;
    private Vector objectAsState = new Vector();
    private Stack history;

    public createMemento() {
        Memento currentState = new Memento();
        currentState.setState( "name", name );
        currentState.setState( "someData", new Integer(someData) );
        currentState.setState( "objectAsState", objectAsState.clone() );
        history.push(currentState);
    }

    public void restoreState() {
        Memento oldState = history.pop();
        name = (String) oldState.getState( "name", name );
        objectAsState = (Vector) oldState.getState( "objectAsState" );
        Integer data = (Integer) oldState.getState( "someData");
        someData = data.intValue();
    }
```

Some Consequences

Expensive
  Space

Narrow & Wide interfaces - Keep data hidden

```
Class Memento {
public:
   virtual ~Memento();
private:
   friend class Originator;
   Memento();
   void setState(State*);
   State* GetState();
```

```
class Originator {
    private String state;

    private class Memento {
     private String state;
     public Memento(String stateToSave)
         { state = stateToSave; }
     public String getState() { return state; }
     }

     public Object memento()
         { return new Memento(state);}
```

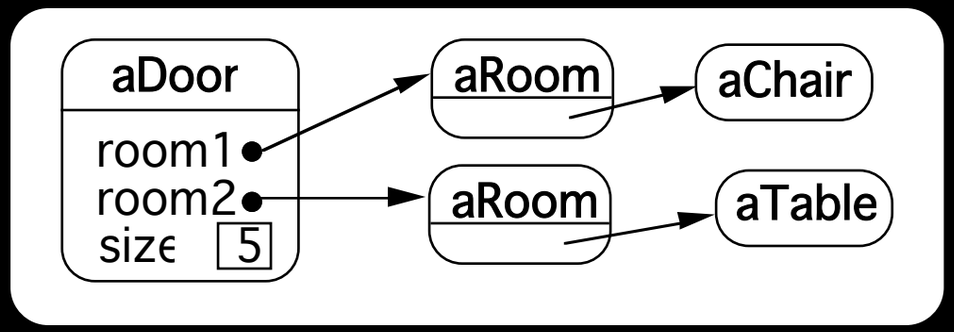
Using Clone to Save State

```
interface Memento extends Cloneable { }

class ComplexObject implements Memento {
    private String name;
    private int someData;

    public Memento createMemento() {
        Memento myState = null;
        try {
            myState =  (Memento) this.clone();
        }
        catch (CloneNotSupportedException notReachable) {
        }
        return myState;
    }

    public void  restoreState( Memento savedState) {
        ComplexObject myNewState = (ComplexObject)savedState;
        name = myNewState.name;
        someData = myNewState.someData;
    }
}
```

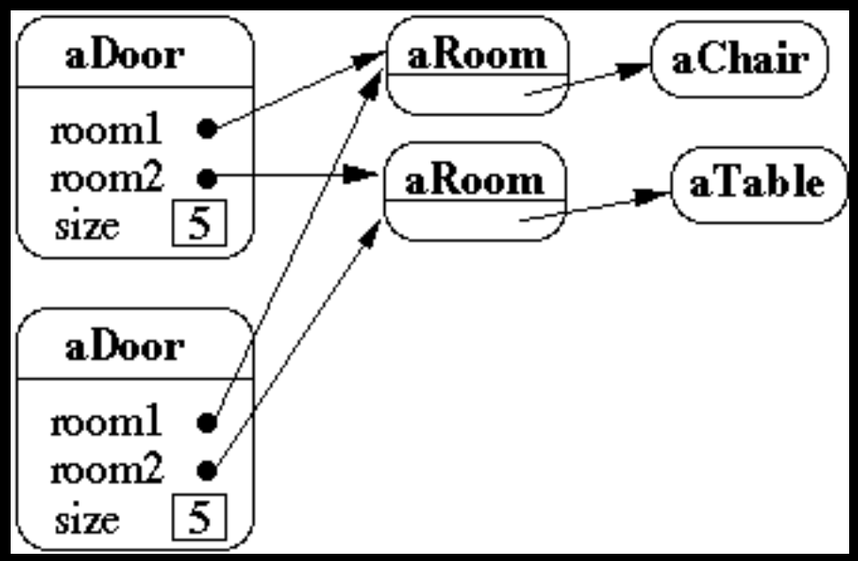
Copying Issues

Shallow Copy Verse Deep Copy

Original Objects

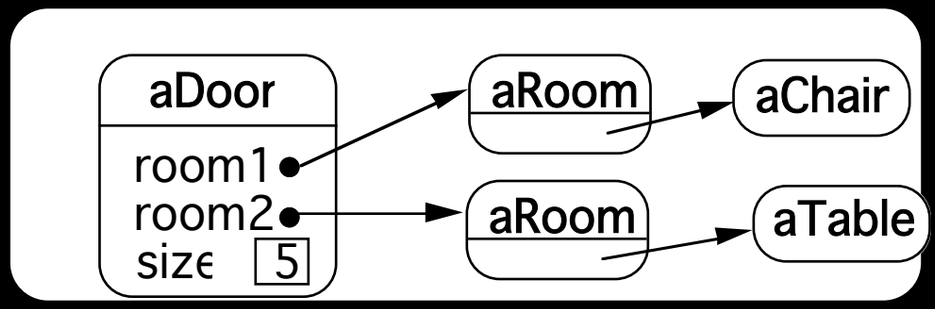


Shallow Copy

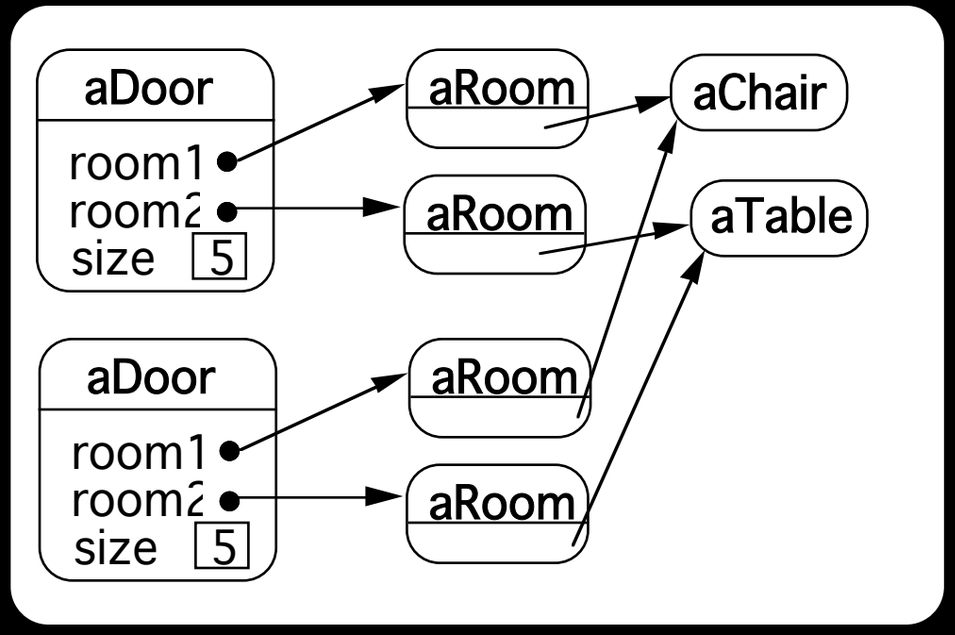


Shallow Copy Verse Deep Copy

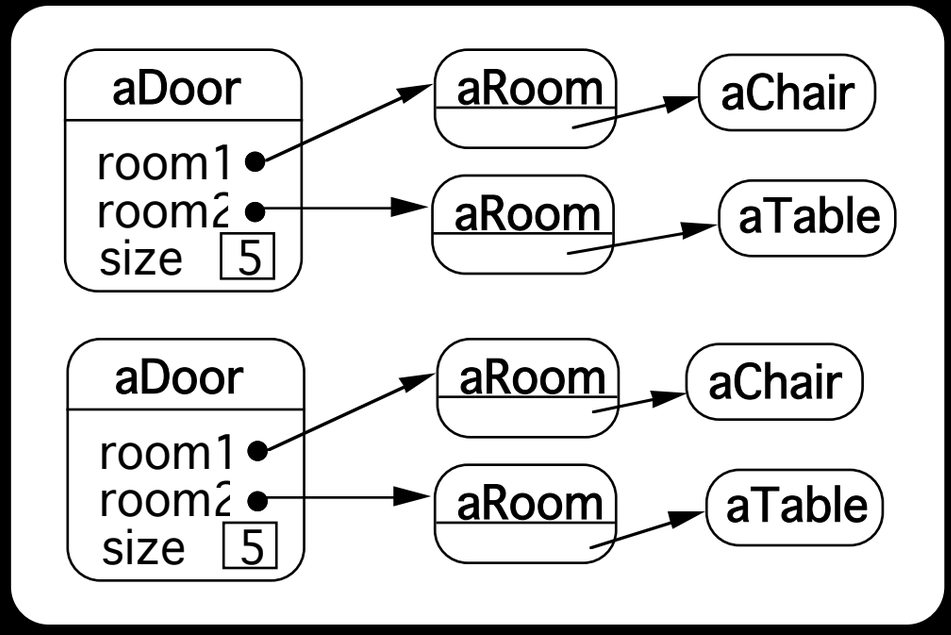
Original Objects



Deep Copy



Deeper Copy



Cloning Issues - C++ Copy Constructors

```
class Door {
    public:
        Door();
        Door( const Door&);
        virtual Door* clone() const;

        virtual void Initialize( Room*, Room* );
        // stuff not shown
    private:
        Room* room1;
        Room* room2;
    }

Door::Door ( const Door& other ) //Copy constructor {
    room1 = other.room1;
    room2 = other.room2;
    }

Door* Door::clone()  const {
    return new Door( *this );
    }
```

Cloning Issues - Java Clone

Shallow Copy

```
class Door implements Cloneable {
    private Room room1;
    private Room room2;

    public Object clone() throws  CloneNotSupportedException {
        return super.clone();
    }
}
```

Deep Copy

```
public class Door implements Cloneable {
    private Room room1;
    private Room room2;

    public Object clone() throws CloneNotSupportedException {
        Door thisCloned =(Door) super.clone();
        thisCloned.room1 = (Room)room1.clone();
        thisCloned.room2 = (Room)room2.clone();
        return thisCloned;
    }
}
```

What if Protocol

When there are complex validations or
performing operations that make it difficult to restore later

Make a copy of the Originator

Perform operations on the copy

Check if operations invalidate the internal state of copy

If so discard the copy & raise an exception

Else perform the operations on the Originator

Memento & Functional Programming

Immutable data

- Data that can not change
- Functional languages have primarily immutable data

If data can not change

- Don’t need memento pattern

Datomic

Database system where all data is immutable

Transactions become easy

Read and writes become independent

Historical data,

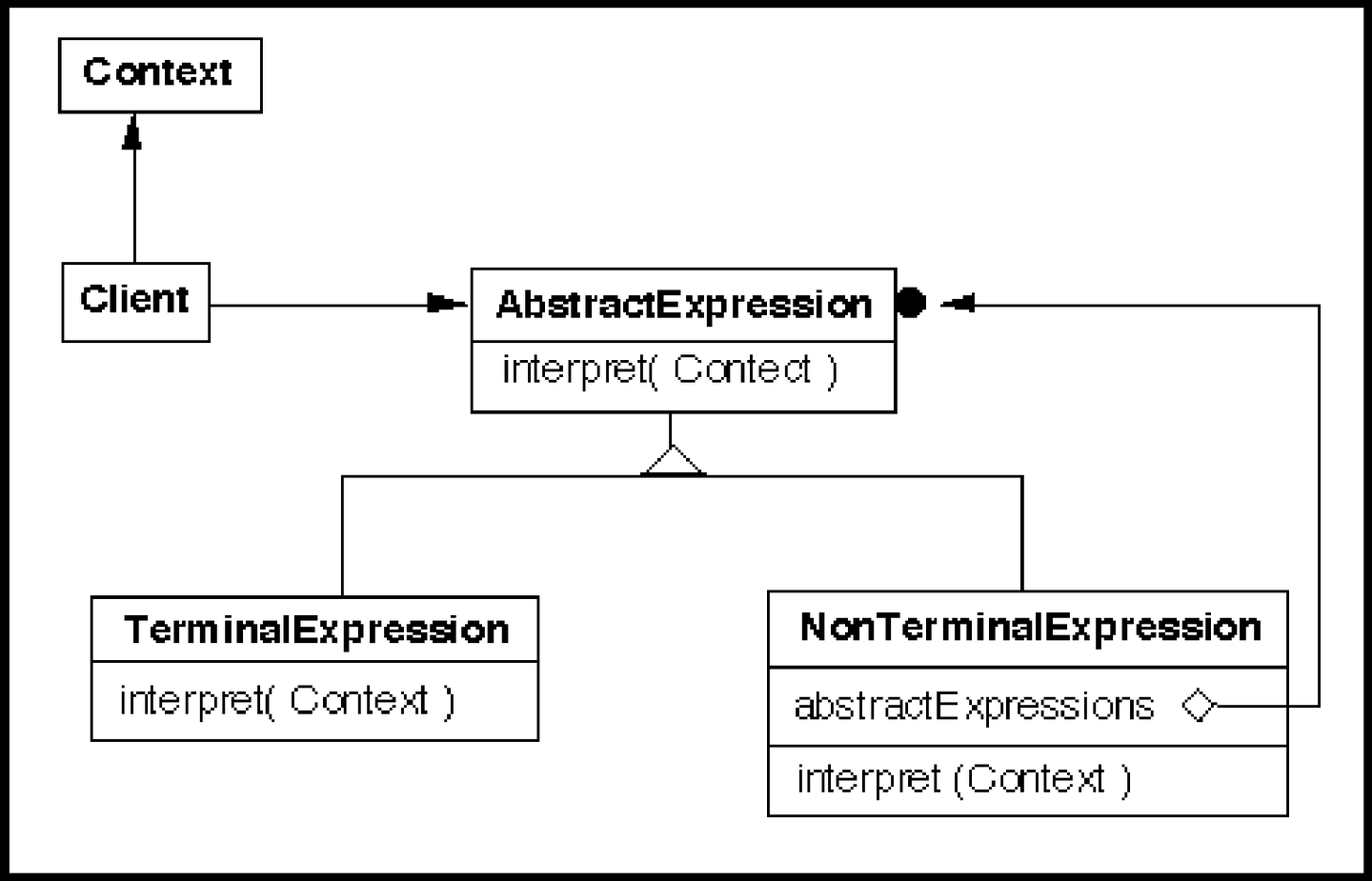
role backs are easy

Auditability

Interpreter

Interpreter

Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language



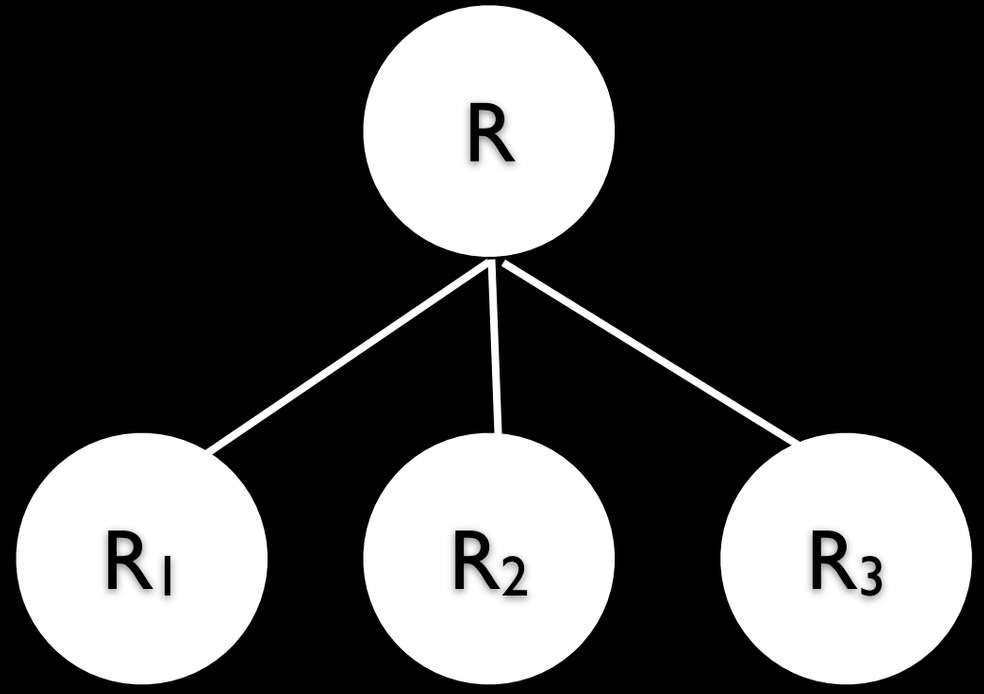
Grammar & Classes

Given a language defined by a grammar like:

$R ::= R_1\ R_2\ R_3$

you create a class for each rule

The classes can be used to construct a tree that represents elements of the language



Example - Boolean Expressions

BooleanExpression ::=

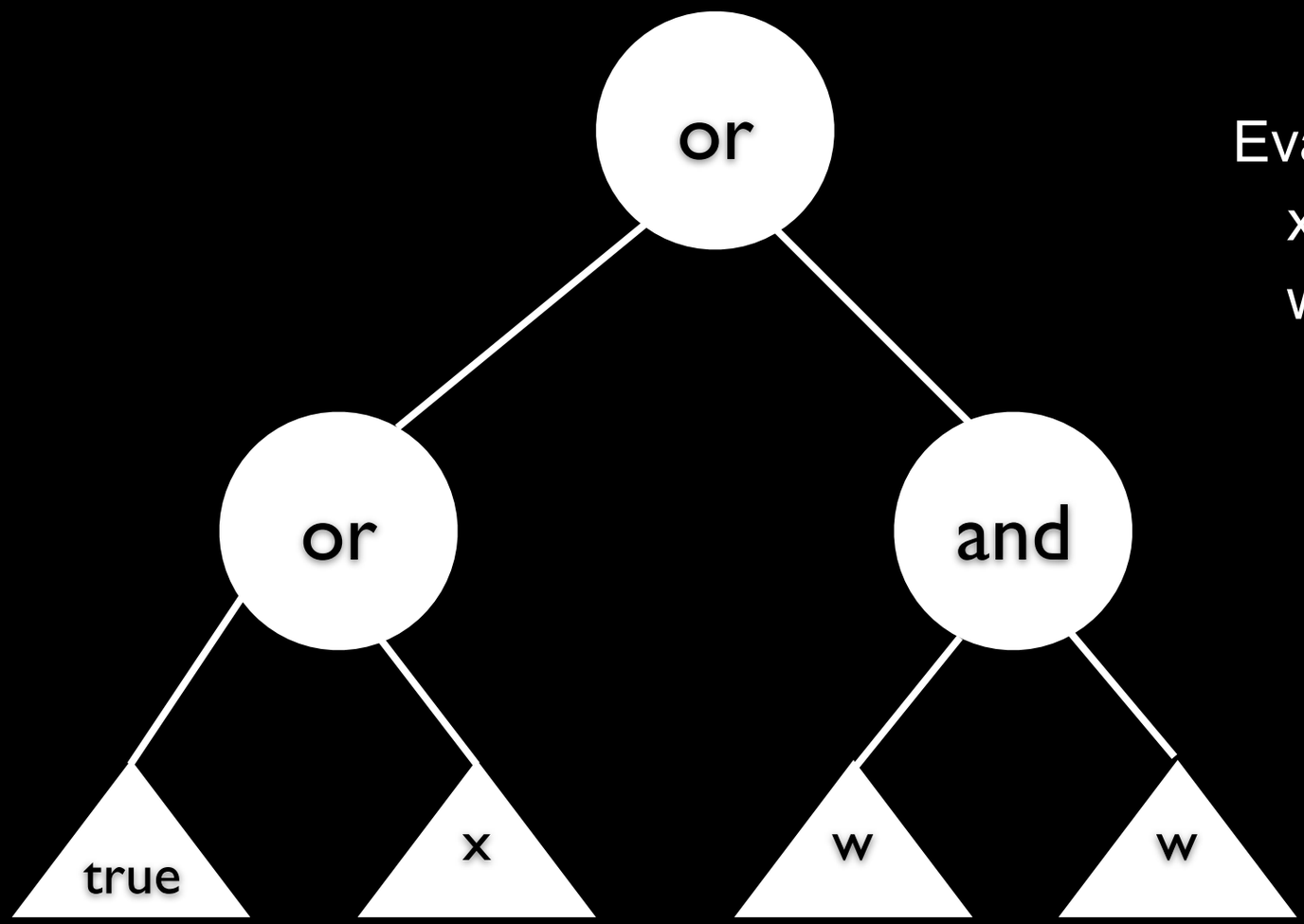
Variable	|
Constant	|
Or	|
And	|
Not	|
BooleanExpression	

And ::= '(' BooleanExpression ‘and’ BooleanExpression ')'
Or ::= '(' BooleanExpression ‘or’ BooleanExpression ')'
Not ::= ‘not’ BooleanExpression

Constant ::= ‘true’ | ‘false’
Variable ::= String

Sample Expression

((true or x) or (w and x))



Evaluate with

x = true

w = false

Sample Classes

```
public interface BooleanExpression{
    public boolean evaluate( Context values );
    public String toString();
}
```

And

```
public class And implements BooleanExpression {
    private BooleanExpression leftOperand;
    private BooleanExpression rightOperand;

    public And( BooleanExpression leftOperand, BooleanExpression rightOperand) {
        this.leftOperand = leftOperand;
        this.rightOperand = rightOperand;
    }

    public boolean evaluate( Context values ) {
        return leftOperand.evaluate( values ) &&  rightOperand.evaluate( values );
    }

    public String toString(){
        return "(" + leftOperand.toString() + " and " + rightOperand.toString() + ")";
    }
}
```

Constant

```
public class Constant implements BooleanExpression {
    private boolean value;
    private static Constant True = new Constant( true );
    private static Constant False = new Constant( false );

    public static Constant getTrue() {  return True; }

    public static Constant getFalse(){ return False; }

    private Constant( boolean value) { this.value = value; }

    public boolean evaluate( Context values ) { return  value;  }

    public String toString() {
       return String.valueOf( value );
    }
}
```

Variable

```
public class Variable implements BooleanExpression {

    private String name;

    private Variable( String name ) {
        this.name = name;
    }

    public boolean evaluate( Context values ) {
        return values.getValue( name );
    }

    public String toString() { return name; }
}
```

Context

```
public class Context {
   Hashtable<String,Boolean> values = new Hashtable<String,Boolean>();

   public boolean getValue( String variableName ) {
      return values.get( variableName );
   }

   public void setValue( String variableName, boolean value ) {
      values.put( variableName,  value ) );
   }
}
```

((true or x) or (w and x))

```
public class Test {
   public static void main( String args[] ) throws Exception  {
      BooleanExpression left =
         new Or(  Constant.getTrue(), new Variable( "x" ) );
      BooleanExpression right =
         new And(  new Variable( "w" ), new Variable( "x" ) );
      BooleanExpression all = new Or(  left, right );

      System.out.println( all );
      Context values = new Context();
      values.setValue( "x", true );
      values.setValue( "w", false );

      System.out.println( all.evaluate( values ) );
      }
}
```

Consequences

It's easy to change and extend the grammar

Implementing the grammar is easy

Complex grammars are hard to maintain

- Use JavaCC or SmaCC instead

Adding new ways to interpret expressions

- The visitor pattern is useful here

Complicates design when a language is simple

Supports combinations of elements better than implicit language

Implementation

The pattern does not talk about parsing!

Flyweight

If terminal symbols are repeated many times using the Flyweight pattern can reduce space usage

Composite

Abstract syntax tree is an instance of the composite

Iterator

Can be used to traverse the structure

Visitor

Can be used to place behavior in one class

Visitor Pattern

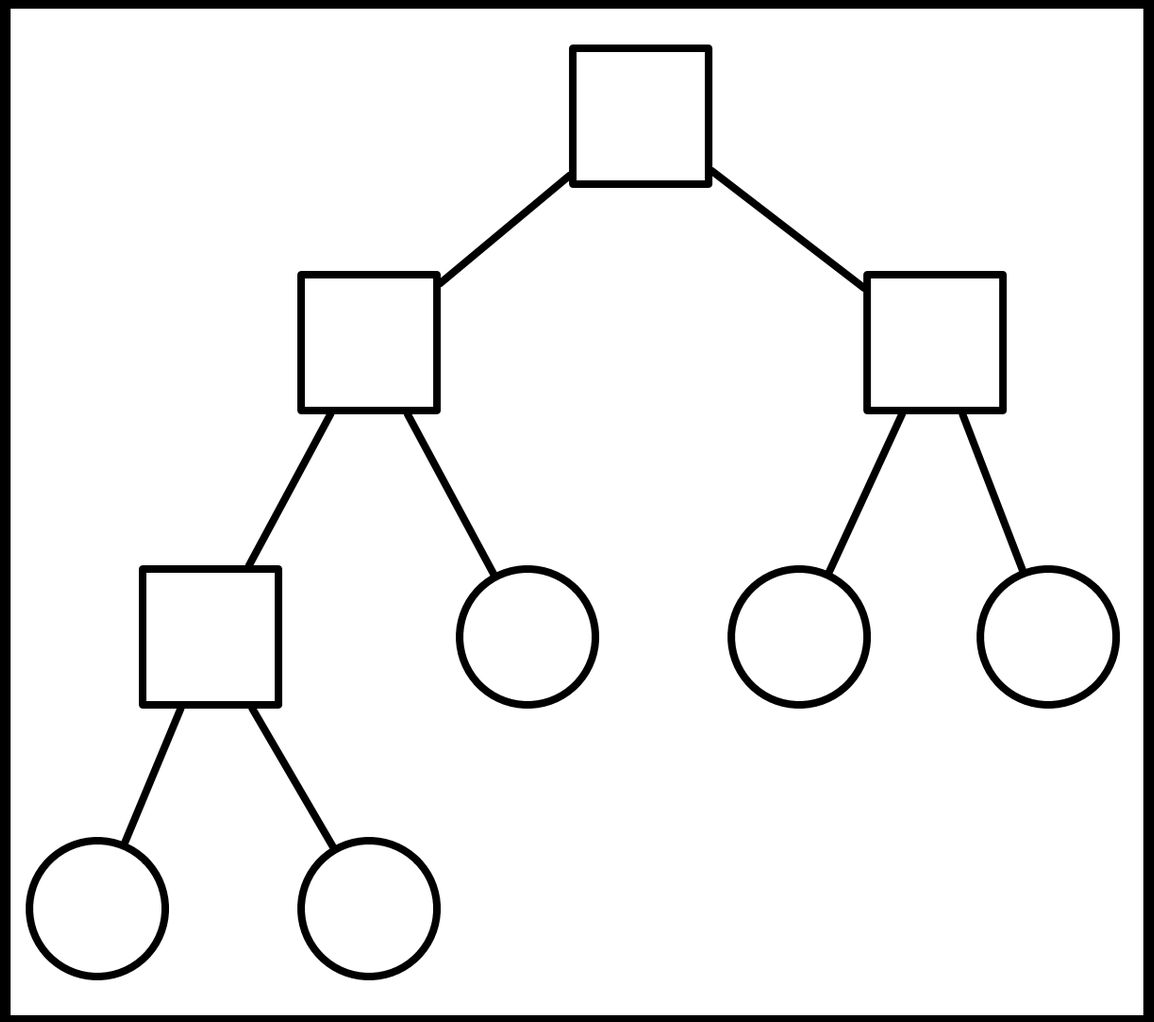
Visitor

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 InnerNode extends Node {...}

class LeafNode extends Node {...}

class Tree { ... }

Tree Printing

HTML Print

PDF Print

TeX Print

RTF Print

Others likely in future

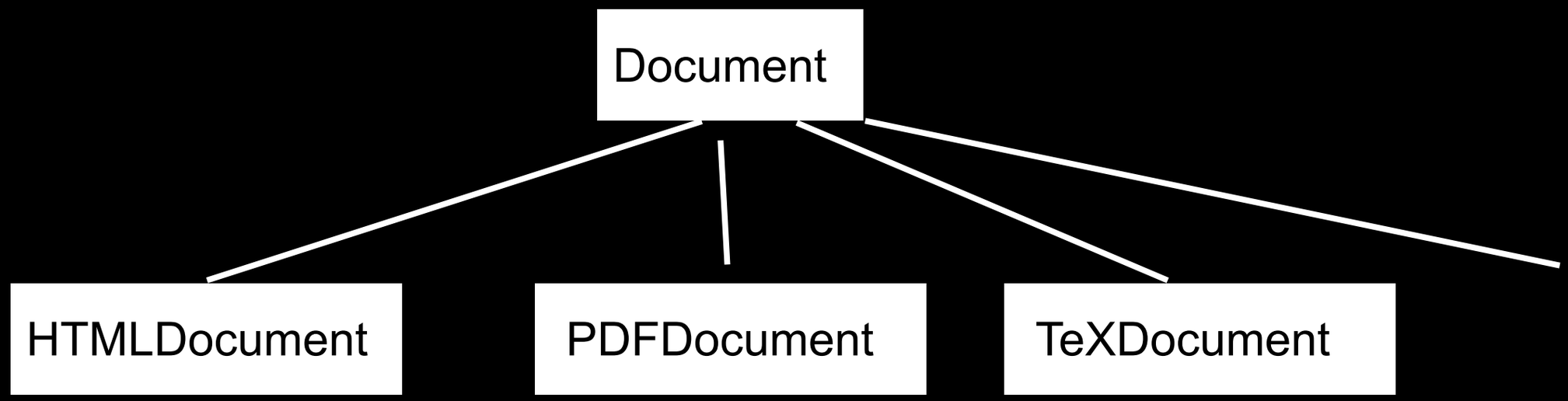
Operations are complex

Do different things on different types of nodes

Need to traverse tree

Not part of BST abstraction

Assume



First Attempt

```
print(Tree source, Document output) {
   foreach( Node current : source ) {
      if current.isInnerNode() && output.isHtml() {
         print inner node on html document
      } else if current.isLeafNode() && output.isHtml() {
         print leaf node on html document
} else if current.isInnerNode() && output.isPDF() {
   print inner node on pdf document
} else if current.isLeafNode() && output.isPDF() {
   print leaf node on pdf document
} etc.
```

Second Attempt

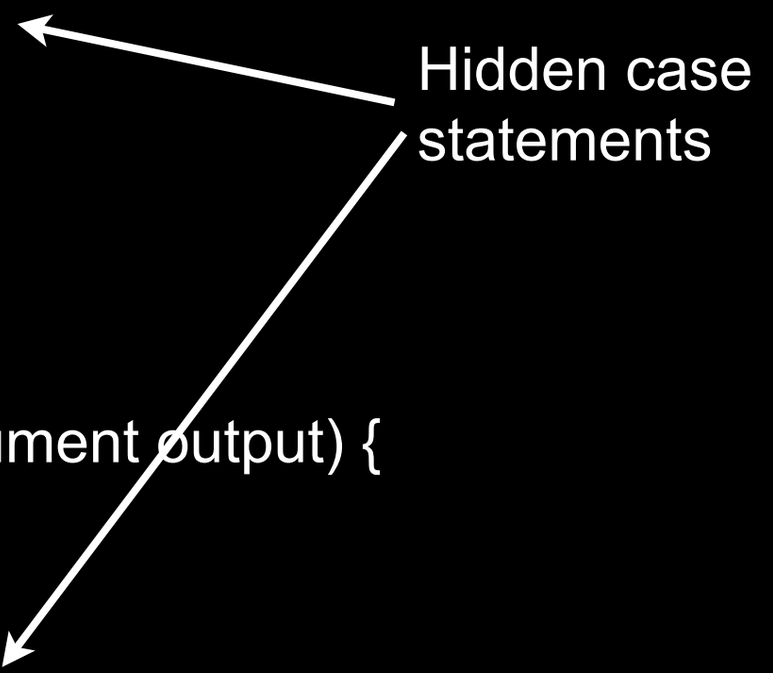
Create Printer Classes

Use iterator to access all elements

Process each element

Second Attempt

```
class TreePrinter {
  public void printTree (Tree toPrint, Document output) {
    foreach( Node current : source ) {
      if (current.isLeafNode())
        printLeafNode(current, output);
      else if (current.isInternalNode() )
        printInternalNode(current, output);
    }
  }

private void printLeafNode(Node current, Document output) {
  if output.isHtml()
    print leaf node on html document
  else if output.isPDF()
    print leaf node on PDF document
  else if etc
}
```



What we would like

```
class TreePrinter {
  public void printTree (Tree source, Document output) {
    foreach( Node current : source ) {
        printNode(current, output);
    }
  }

  private void printNode(InnerNode current, HTMLDocument output) {
    print inner node on html document
}

private void printNode(LeafNode current, HTMLDocument output) {
    print leaf node on html document
}

private void printNode(InnerNode current, PDFDocument output) {
    print inner node on PDF document
}
etc
```

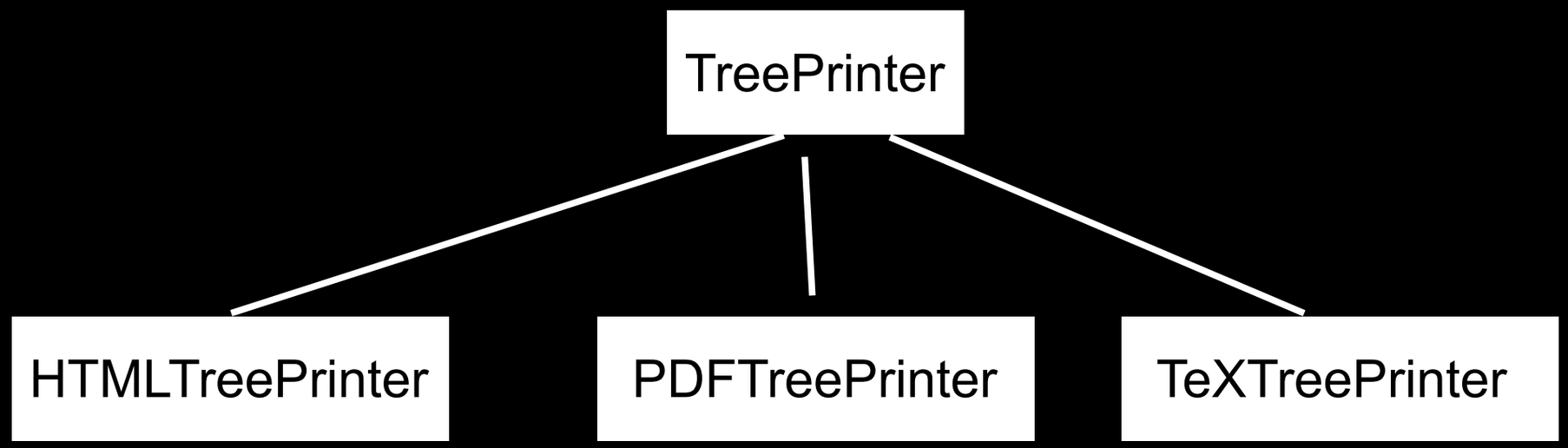
Compile Error

Overloaded Methods

Which overloaded method to run

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

Use Subclasses



Third Attempt

```
class TreePrinter {
   Document output;
   public void printTree (Tree toPrint) {
      foreach( Node current : source ) {
         if (current.isLeafNode())
            printLeafNode(current, output);
         else if (current.isInternalNode() )
            printInternalNode(current, output);
   }
}

public Document getDocument() { return output;}

private abstract void printLeafNode(Node current);
private abstract void printInnerNode(Node current);

}
```

Third Attempt

```
class HTMLTreePrinter extends TreePrinter {

  private void printLeafNode(Node current) {
      print leaf node on html document
  }

  private void printInnerNode(Node current) {
      print inner node on html document
  }
}
```

Overloaded Method

```
class TreePrinter {
   Document output;
   public void printTree (Tree toPrint) {
      foreach( Node current : source ) {
         printNode(current);
      }
   }
```

Compile Error

```
   public Document getDocument() { return output;}

   private abstract void printNode(LeafNode current);
   private abstract void printNode(InnerNode current);
   }
```

Key Idea

Receiver of method is determined at runtime

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

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

Add Methods to Nodes

```
class Node {
   abstract public void print(TreePrinter printer);
}


class InnerNode extends Node {
    public void print(TreePrinter printer) {
       printer.printInnerNode( this );
    }
}

class LeafNode extends Node {
    public void print(TreePrinter printer) {
       printer.printLeafNode( this );
    }
}
```

Now we can Use Polymorphism

```
class TreePrinter {
   Document output;
   public void printTree (Tree toPrint) {
      foreach( Node current : source ) {
            current.print(this);
      }
   }

   public Document getDocument() { return output;}

   public abstract void printLeafNode(Node current);
   public abstract void printInnerNode(Node current);

}
```

What Have we gained

No if statements

Can add more types of Documents by adding subclasses

Work for a Document is in one place

Divided work into small parts

We can use method overloading

```
class TreePrinter {
   Document output;
   public void printTree (Tree toPrint) {
      foreach( Node current : source ) {
         current.print(this);
      }
   }

public Document getDocument() { return output;}

public abstract void printNode(InnerNode current);
public abstract void printNode(LeafNode current);

}
```

```
class InnerNode extends Node {
   public void print(TreePrinter printer) {
      printer.printNode( this );
   }
}

class LeafNode extends Node {
   public void print(TreePrinter printer) {
      printer.printNode( this );
   }
}
```

But We don’t gain anything

```
class TreePrinter {
   Document output;
   public void printTree (Tree toPrint) {
      foreach( Node current : source ) {
            current.print(this);
      }
   }

   public Document getDocument() { return output;}

   public abstract void printNode(InnerNode current);
   public abstract void printNode(LeafNode current);

}
```

Still need to know about each node type

One Last Problem

Modified the nodes for a specific issue

For each issue need to add methods to node!?!

Make the structure generic

In The Nodes

```
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();
foreach( Node current : source ) {
   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 );
    }
}
```

```
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

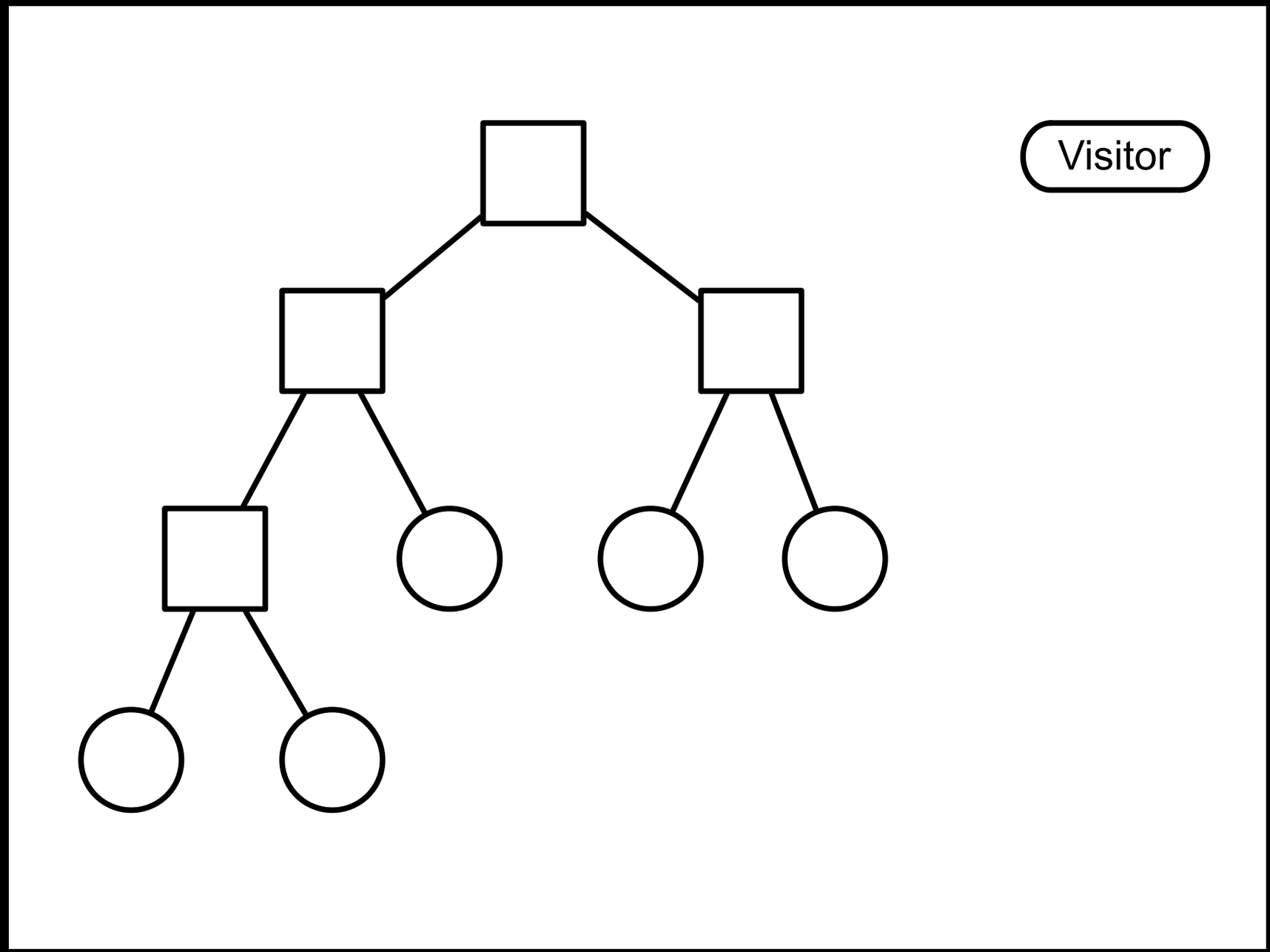
Pass the visitor to each element in the structure

The element then activates the visitor

Visitor performs its operation on the element

Each visitX method only deals with on type of element

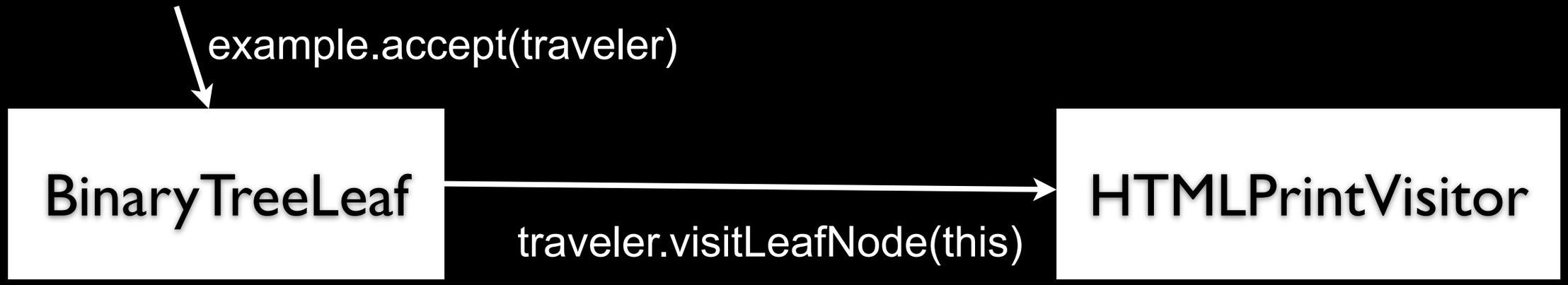
Tree Example



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

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

Clojure, Lisp & Multi-methods

```
(defmulti printNode (fn [node document] [(class node) (class document)]))

(defmethod printNode [InnerNode HTMLDocument]
  [node document]
  code to print InnerNode on HTMLDocument)

(defmethod printNode [InnerNode PDFDocument]
  [node document]
  code to print InnerNode on PDFDocument)

(defmethod printNode [LeafNode PDFDocument]
  [node document]
  code to print InnerNode on PDFDocument)

etc.
```

Clojure, Lisp & Multi-methods

Multi-methods in Clojure do select overloaded method
- At run-time
- Based on argument types

No need for visitor pattern