CS 635 Advanced Object-Oriented Design & Programming
Spring Semester, 2015
Doc 8 Visitor, Pattern Intro
Feb 26, 2015

Visitor Pattern

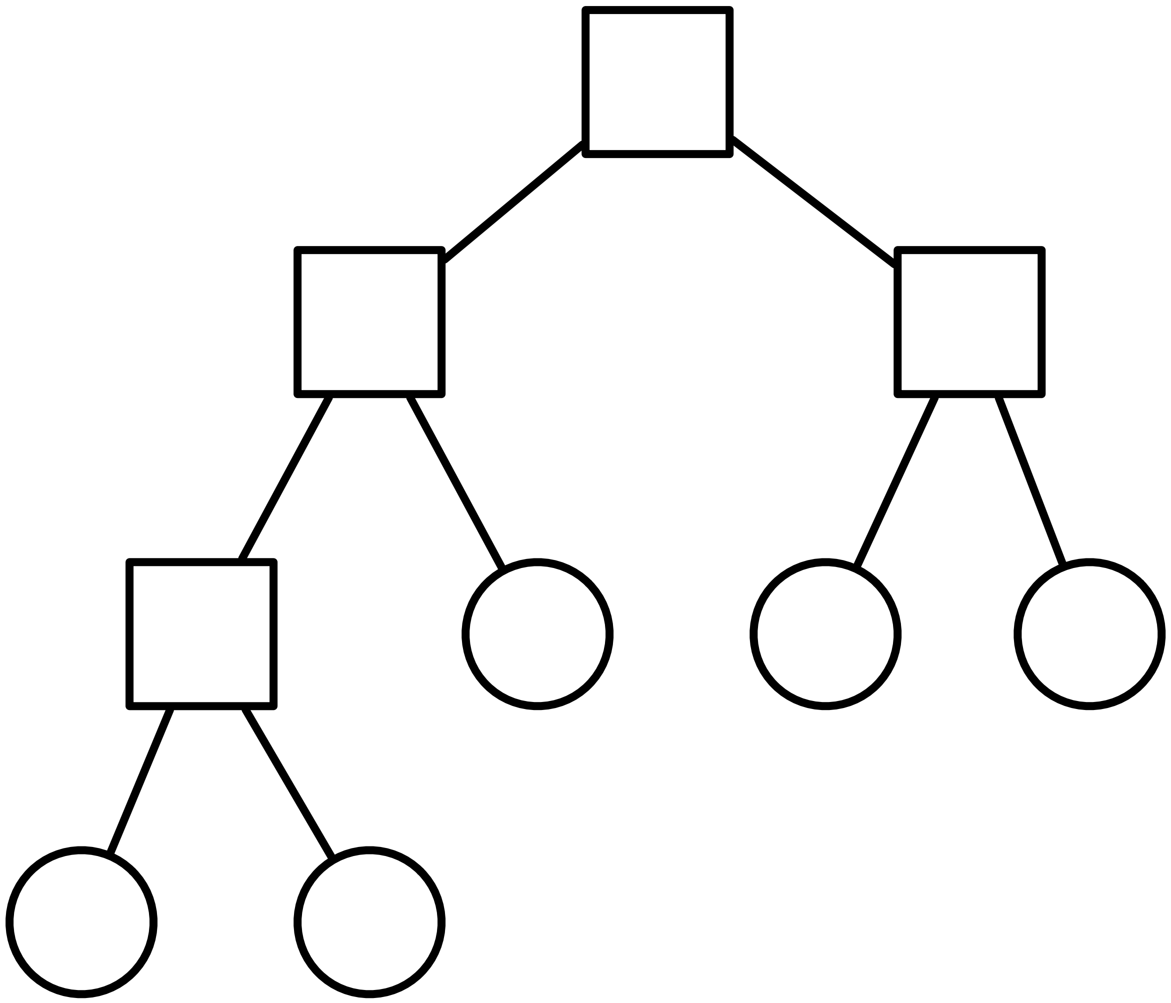
Visitor

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

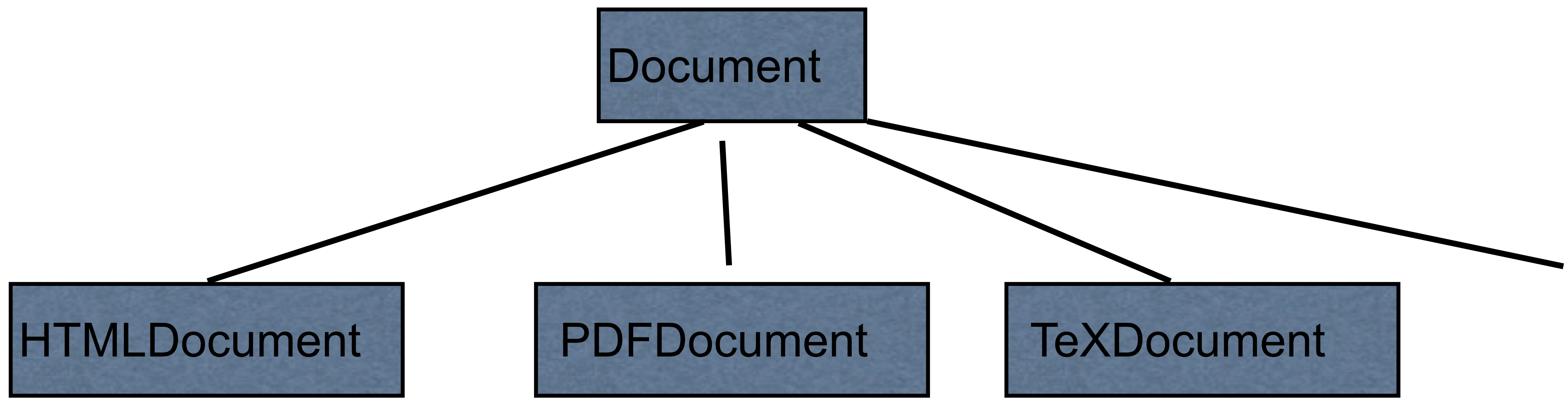
Operations are complex

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
    }
```

Hidden case statements

What we would like

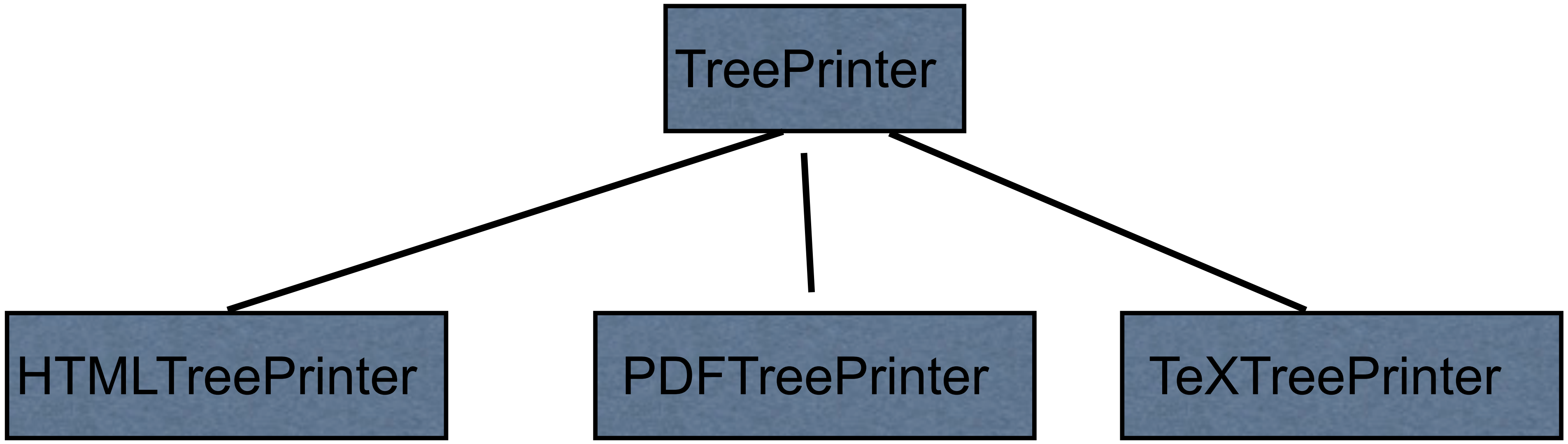
```
class TreePrinter {
    public void printTree (Tree source, Document output) {
        foreach( Node current : source ) {
                printNode(current, output);   <-------------------  Compile Error
        }
    }

    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 html document
    }
    etc
```

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);
        }
    }

    public Document getDocument() { return output;}

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

Compile Error

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) {
        aVisitor.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)
{
                aVisitor.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

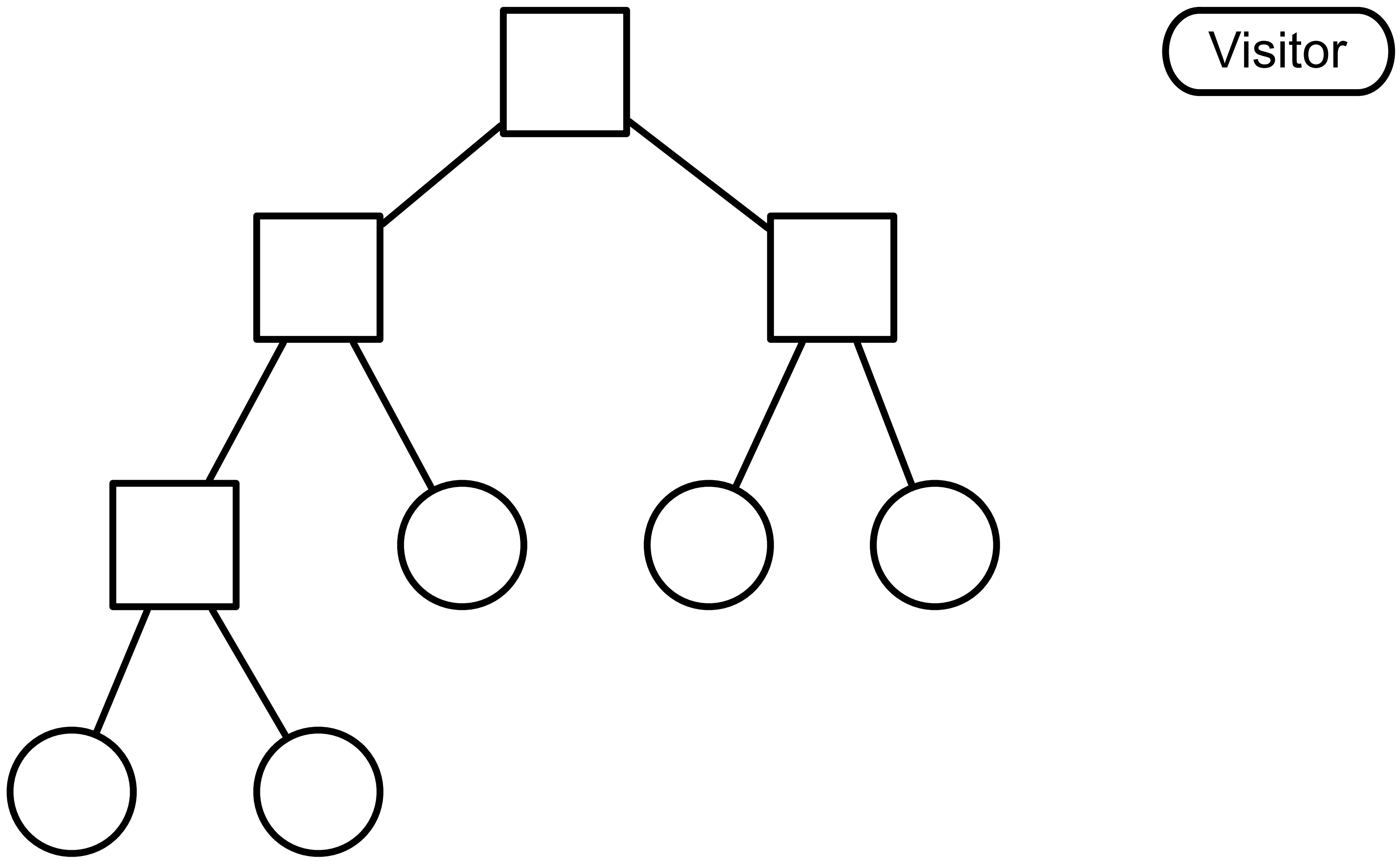
Pass the visitor to each element in the structure

The element then activates the visitor

Visitor performs its operation on the element

Each visitX method only deals with on type of element

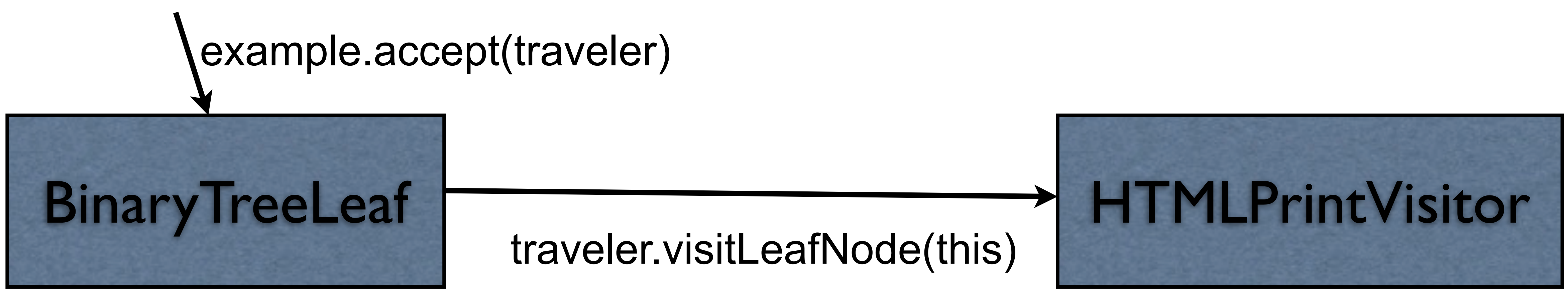
Tree Example



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

Clojure, Lisp & Multi-methods

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

No need for visitor pattern

```
Tree source = YYY;
Document output = new XXX();
foreach( Node current : source )
    printNode(current, output);
```

```
void printNode(InnerNode current, HTMLDocument output) { blah }

void printNode(LeafNode current, HTMLDocument output) { blah }

void printNode(InnerNode current, PDFDocument output) { blah }

etc.
```

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

Pattern Intro

Pattern Beginnings

"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice"

"Each pattern is a three-part rule, which expresses a relation between a certain context, a problem, and a solution"

A Pattern Language, Christopher Alexander, 1977

A Place To Wait

The process of waiting has inherent conflicts in it.

Waiting for doctor, airplane etc. requires spending time hanging around doing nothing

Cannot enjoy the time since you do not know when you must leave

Classic "waiting room"

Dreary little room
People staring at each other
Reading a few old magazines
Offers no solution

Fundamental problem

How to spend time "wholeheartedly" and
Still be on hand when doctor, airplane etc arrive

Fuse the waiting with other activity that keeps them in earshot

- Playground beside Pediatrics Clinic
- Horseshoe pit next to terrace where people waited

Allow the person to become still meditative

- A window seat that looks down on a street
- A protected seat in a garden
- A dark place and a glass of beer
- A private seat by a fish tank

A Place To Wait

Therefore:

"In places where people end up waiting create a situation which makes the waiting positive. Fuse the waiting with some other activity - newspaper, coffee, pool tables, horseshoes; something which draws people in who are not simple waiting. And also the opposite: make a place which can draw a person waiting into a reverie; quiet; a positive silence"

Chicken And Egg

Problem

Two concepts are each a prerequisite of the other

To understand A one must understand B

To understand B one must understand A

A "chicken and egg" situation

Constraints and Forces

First explain A then B

Everyone would be confused by the end

Simplify each concept to the point of incorrectness to explain the other one

People don't like being lied to

Solution

Explain A & B correctly by superficially

Iterate your explanations with more detail in each iteration

Patterns for Classroom Education, Dana Anthony, pp. 391-406, Pattern Languages of Program Design 2, Addison Wesley, 1996

Design Principle 1

Program to an interface, not an implementation

Use abstract classes (and/or interfaces in Java) to define common interfaces for a set of classes

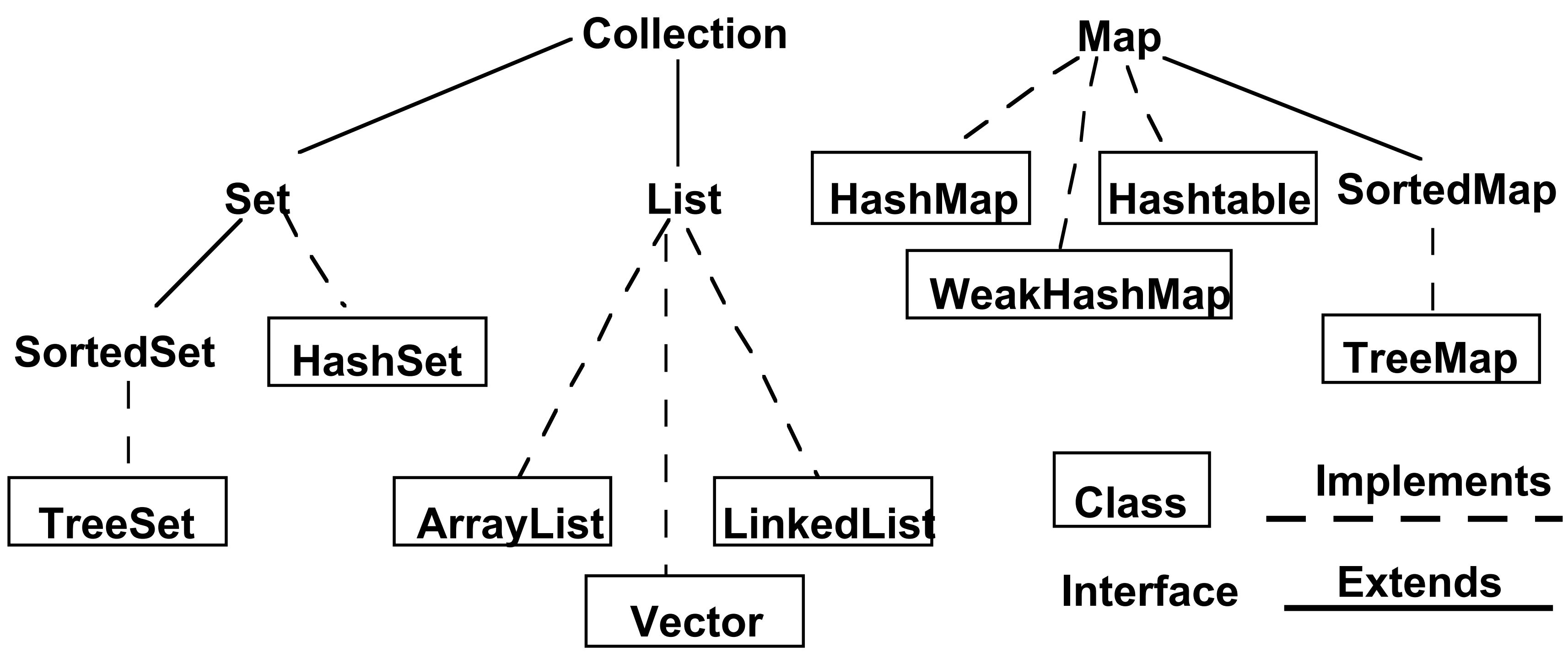
Declare variables to be instances of the abstract class not instances of particular classes

Benefits of programming to an interface

Client classes/objects remain unaware of the classes of objects they use, as long as the objects adhere to the interface the client expects

Client classes/objects remain unaware of the classes that implement these objects.
Clients only know about the abstract classes (or interfaces) that define the interface.

Programming to an Interface



Collection students = new XXX;
students.add(aStudent);

students can be any collection type

We can change our mind on what type to use

Interface & Duck Typing

In dynamically typed languages programming to an interface is the norm

Dynamically typed languages tend to lack a way to declare an interface

Design Principle 2

Favor object composition over class inheritance

Composition

- Allows behavior changes at run time
- Helps keep classes encapsulated and focused on one task
- Reduce implementation dependencies

Inheritance

```
class A {
    Foo x
    public int complexOperation() {
    blah }
}

class B extends A {
    public void bar() { blah}
}
```

Composition

```
class B {
    A myA;
    public int complexOperation() {
        return myA.complexOperation()
    }

    public void bar() { blah}
}
```

Designing for Change

Creating an object by specifying a class explicitly
- Abstract factory, Factory Method, Prototype

Dependence on hardware and software platforms
- Abstract factory, Bridge

Dependence on object representations or implementations
- Abstract factory, Bridge, Memento, Proxy

Algorithmic dependencies
- Builder, Iterator, Strategy, Template Method, Visitor

Tight Coupling
- Abstract factory, Bridge, Chain of Responsibility, Command, Facade, Mediator, Observer

Extending functionality by subclassing
- Bridge, Chain of Responsibility, Composite, Decorator, Observer, Strategy

Dependence on specific operations
- Chain of Responsibility, Command

Inability to alter classes conveniently
- Adapter, Decorator, Visitor

Kent Beck's Rules for Good Style

One and only once

In a program written in good style, everything is said once and only once

Methods with the same logic
Objects with same methods
Systems with similar objects

rule is not satisfied

Lots of little Pieces

"Good code invariably has small methods and small objects"

Small pieces are needed to satisfy "once and only once"

Make sure you communicate the big picture or you get a mess

Rates of change

Don't put two rates of change together

An object should not have a field that changes every second & a field that change once a month

A collection should not have some elements that are added/removed every second and some that are add/removed once a month

An object should not have code that has to change for each piece of hardware and code that has to change for each operating system

Replacing Objects

Good style leads to easily replaceable objects

"When you can extend a system solely by adding new objects without modifying any existing objects, then you have a system that is flexible and cheap to maintain"

Moving Objects

"Another property of systems with good style is that their objects can be easily moved to new contexts"