CS 635 Advanced Object-Oriented Design & Programming
Fall Semester, 2021
Doc 16 Visitor, Adapter, Bridge
Nov 18, 2021

The Controller Dilemma

The controller of a well-regulated system may not seem to be working hard.

The Controller Fallacy

If the controller isn't busy, it's not doing a good job.
If the controller is very busy, it must be a good controller.

Manager's Not Available

Busy managers mean bad management.

-- G. Weinberg

Visitor Pattern

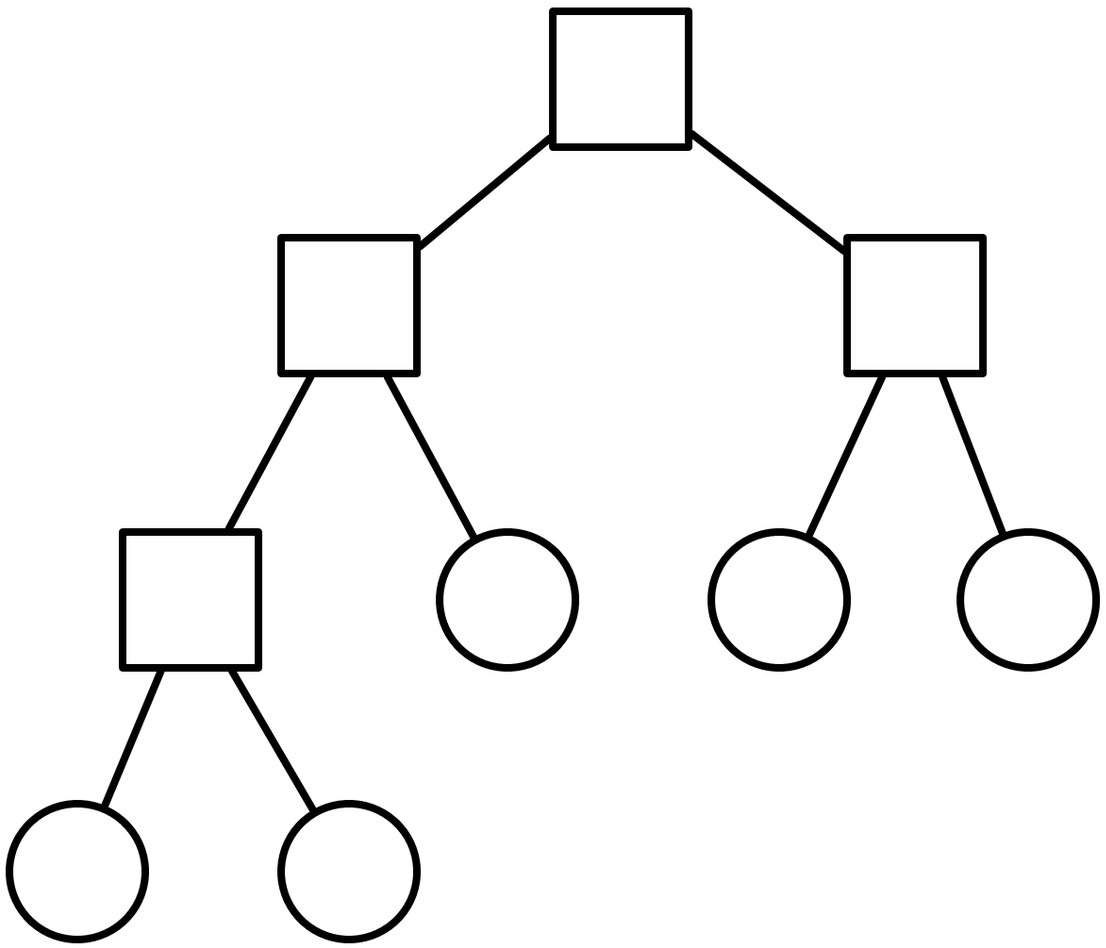
Visitor

Intent

Represent an operation to be performed on the elements of an heterogeneous 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

Operations are complex

Do different things on different types of nodes

Need to traverse tree

Others likely in future

Not part of BST abstraction

Visitor Solution

In The Nodes

```
class Node {
   abstract public void accept(Visitor aVisitor);
}
```

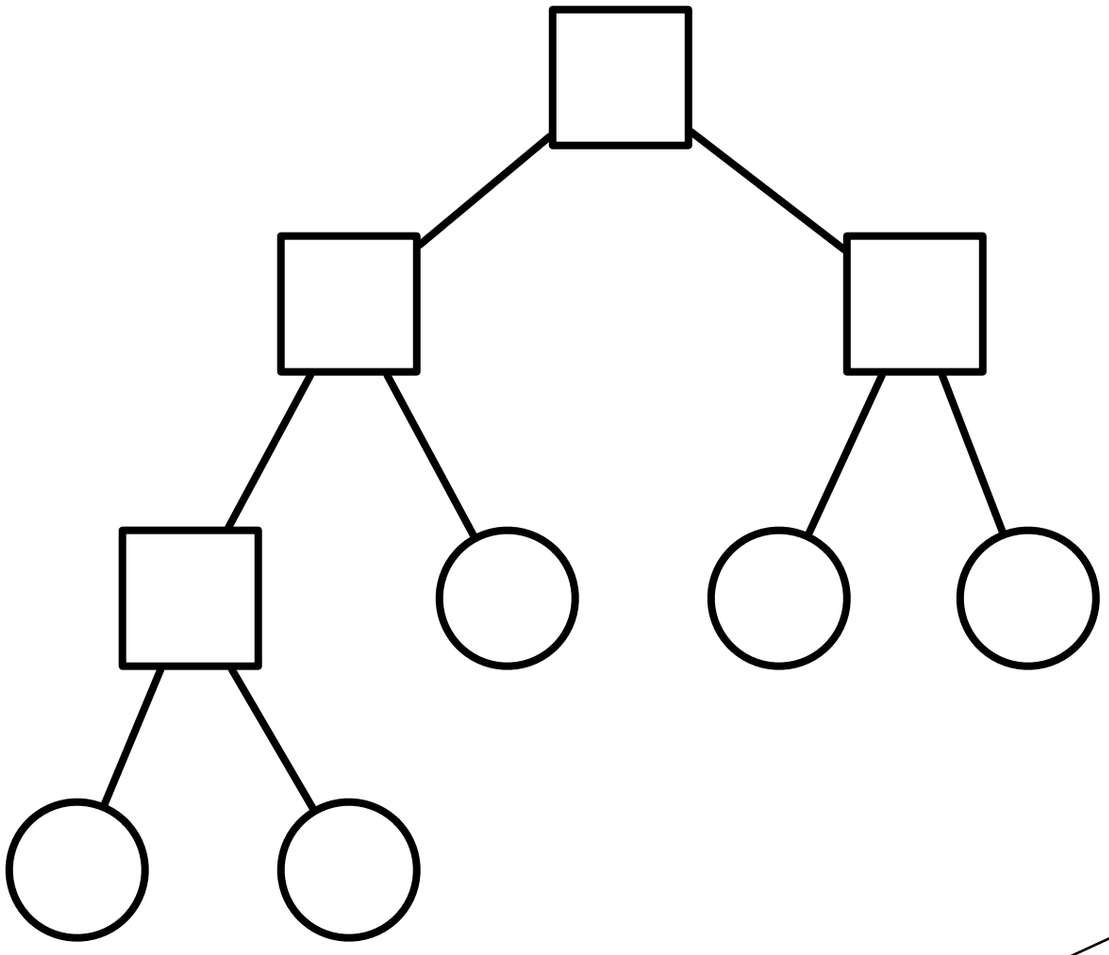
```
class BinaryTreeNode extends Node {
    public void accept(Visitor aVisitor) {
       aVisitor.visitBinaryTreeNode( this );
    }
}

class BinaryTreeLeaf extends Node {
    public void accept(Visitor aVisitor) {
       aVisitor.visitBinaryTreeLeaf( this );
    }
}
```

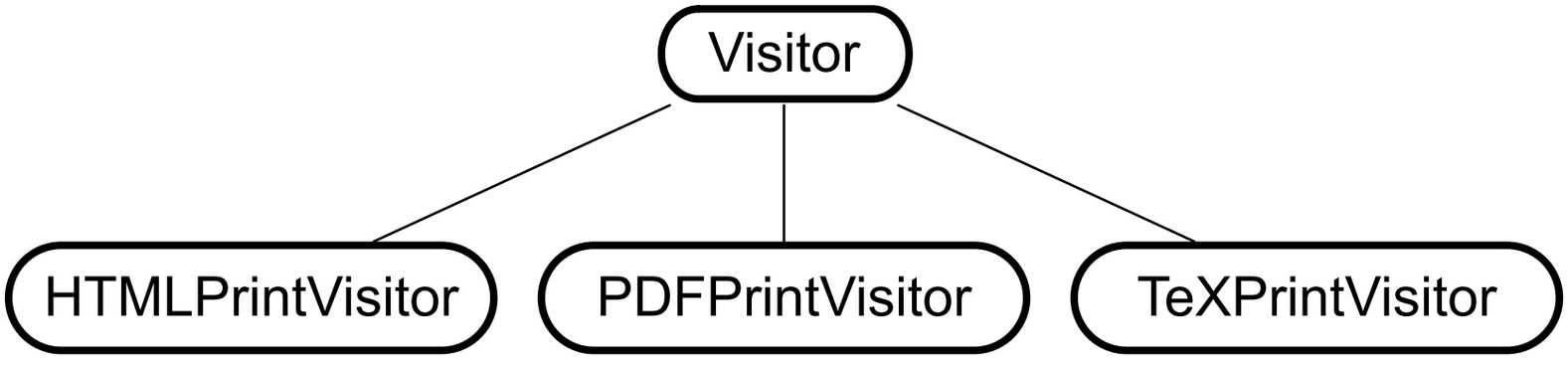
Visitor

```
abstract class Visitor {

    abstract void visitBinaryTreeNode( BinaryTreeNode );

    abstract void visitBinaryTreeLeaf( BinaryTreeLeaf );
}
```

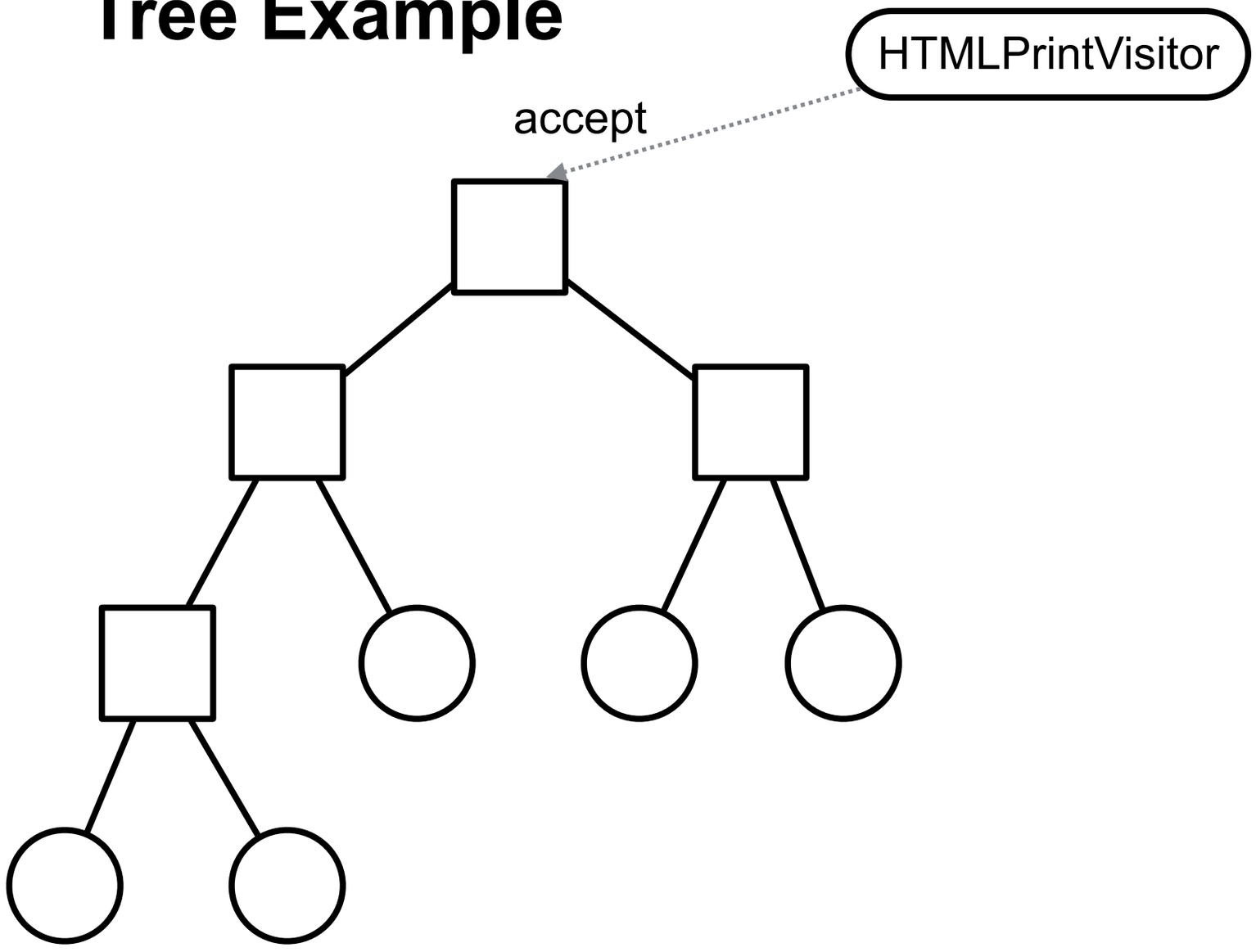
```
class HTMLPrintVisitor extends Visitor {

    public void visitBinaryTreeNode( BinaryTreeNode x ) {
        HTML print code here
    }

    public void visitBinaryTreeLeaf( BinaryTreeLeaf x){ ...}
}
```

Tree Example

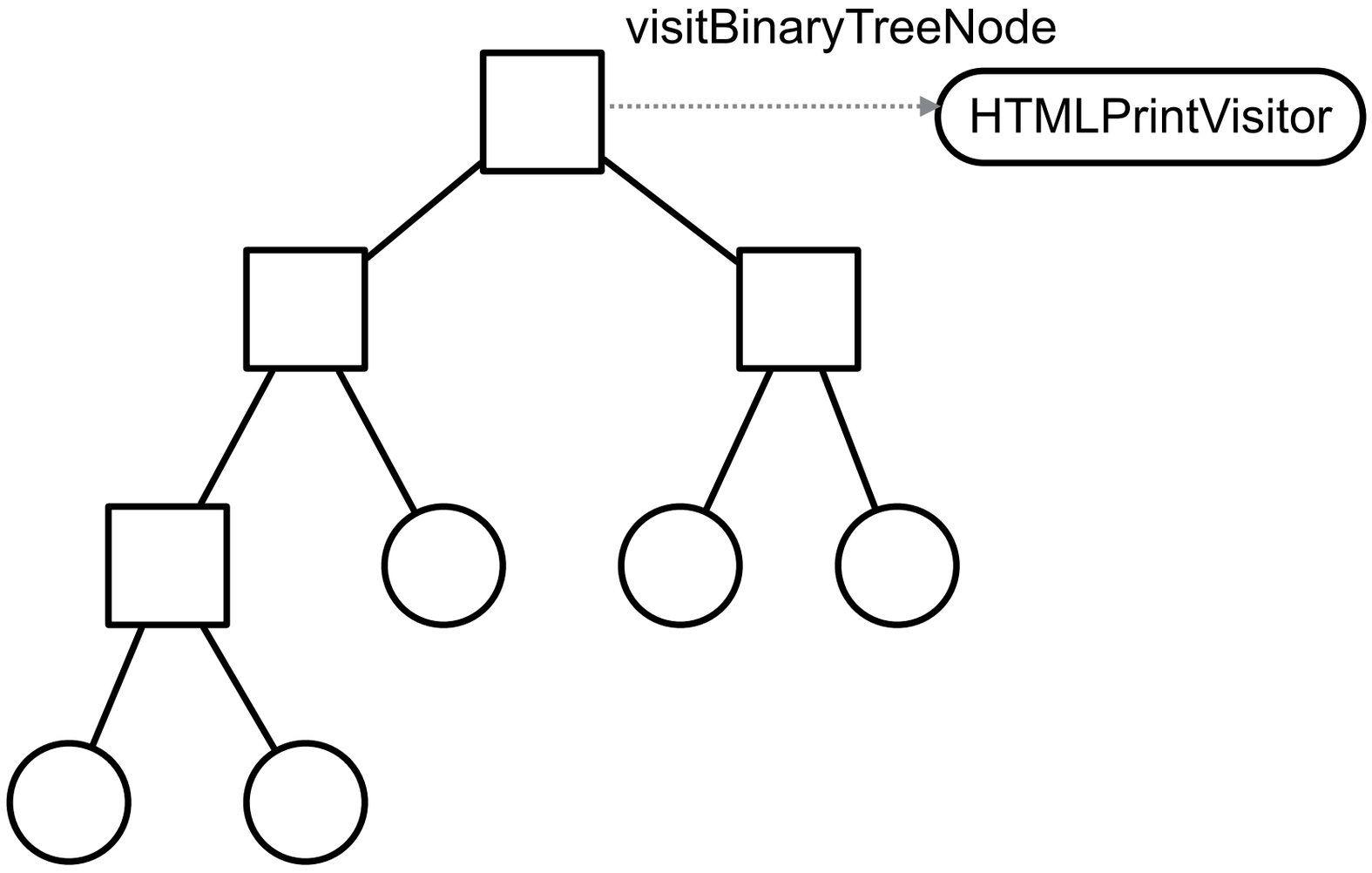




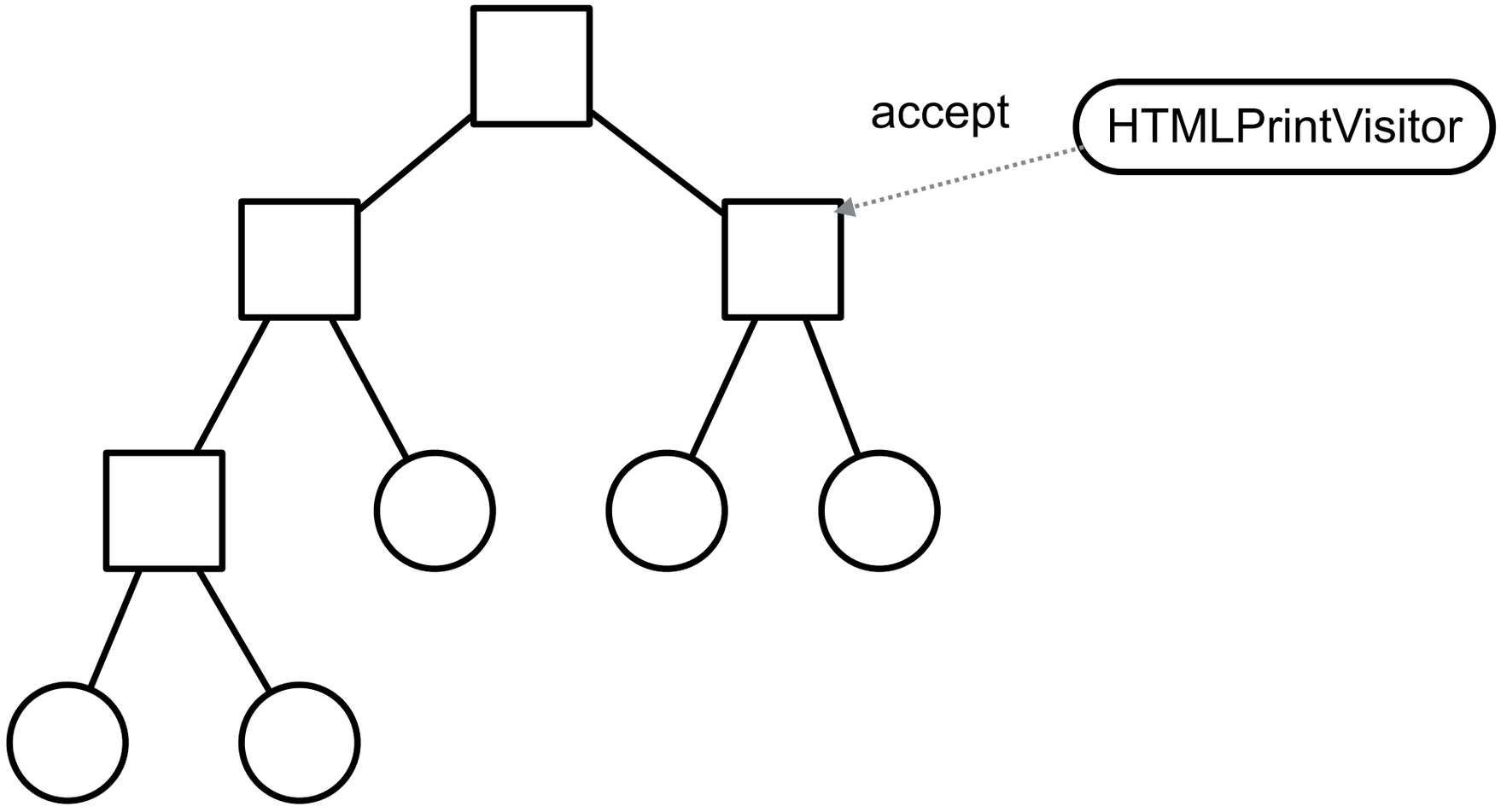
Tree Example



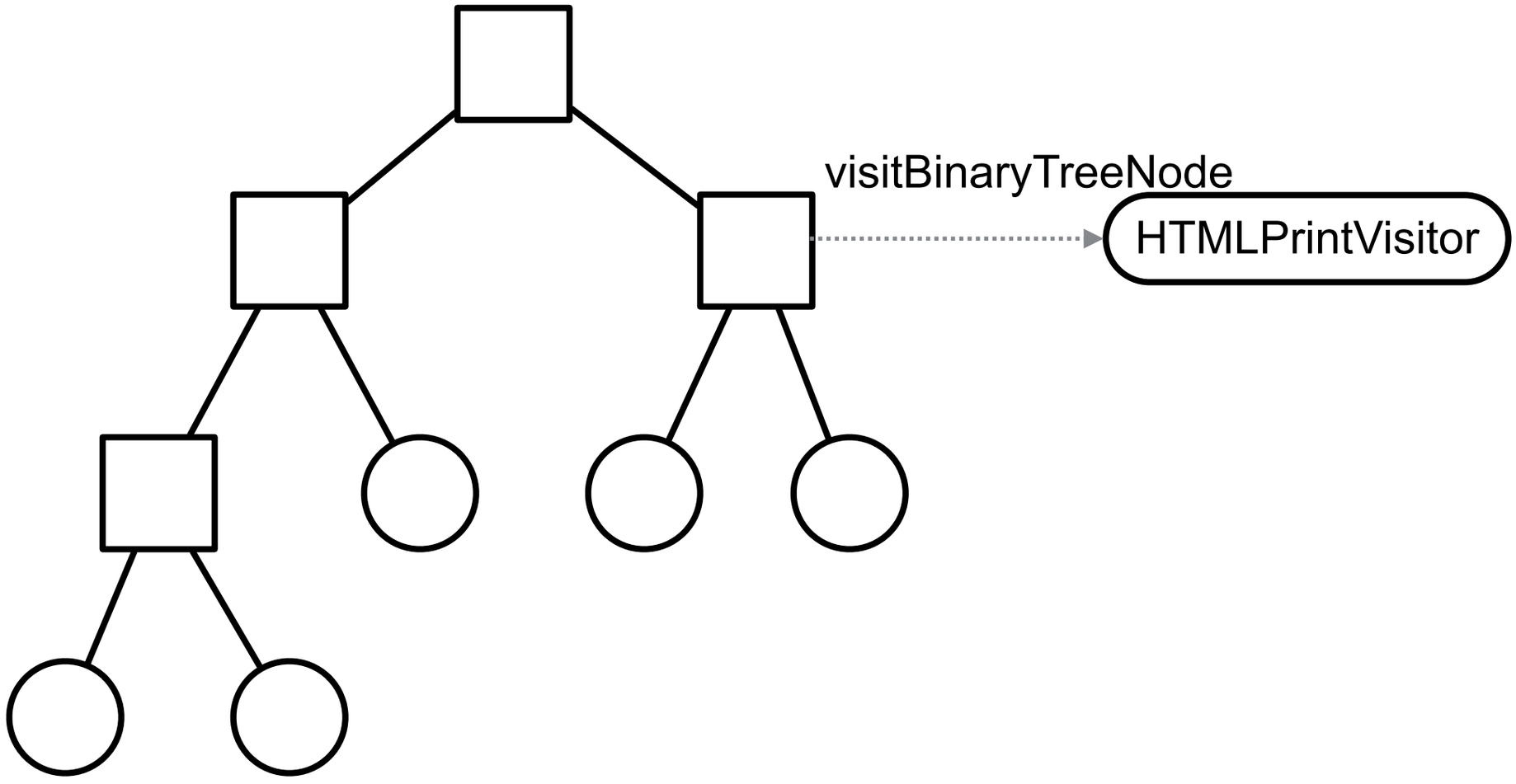
Tree Example



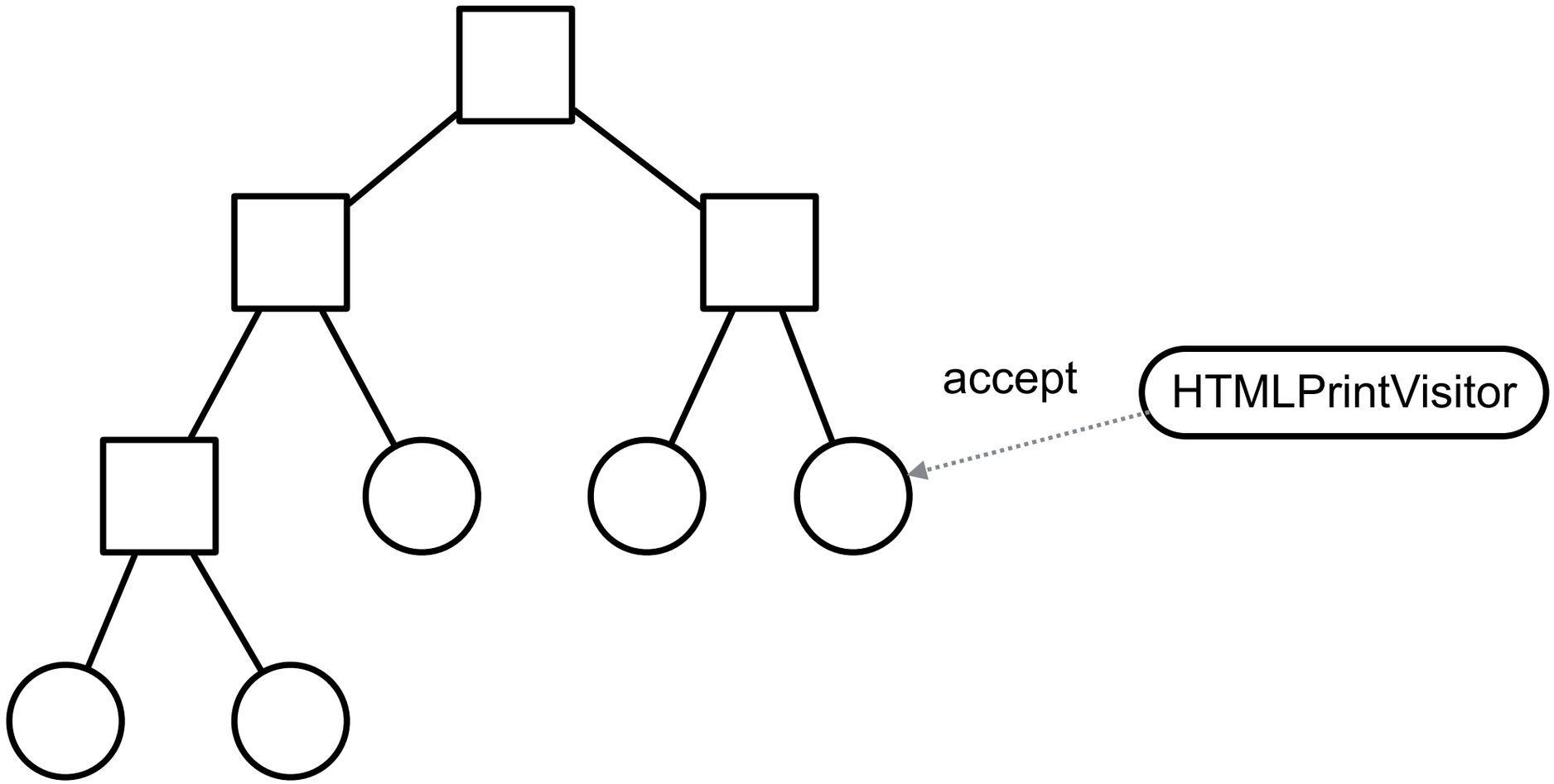
Tree Example



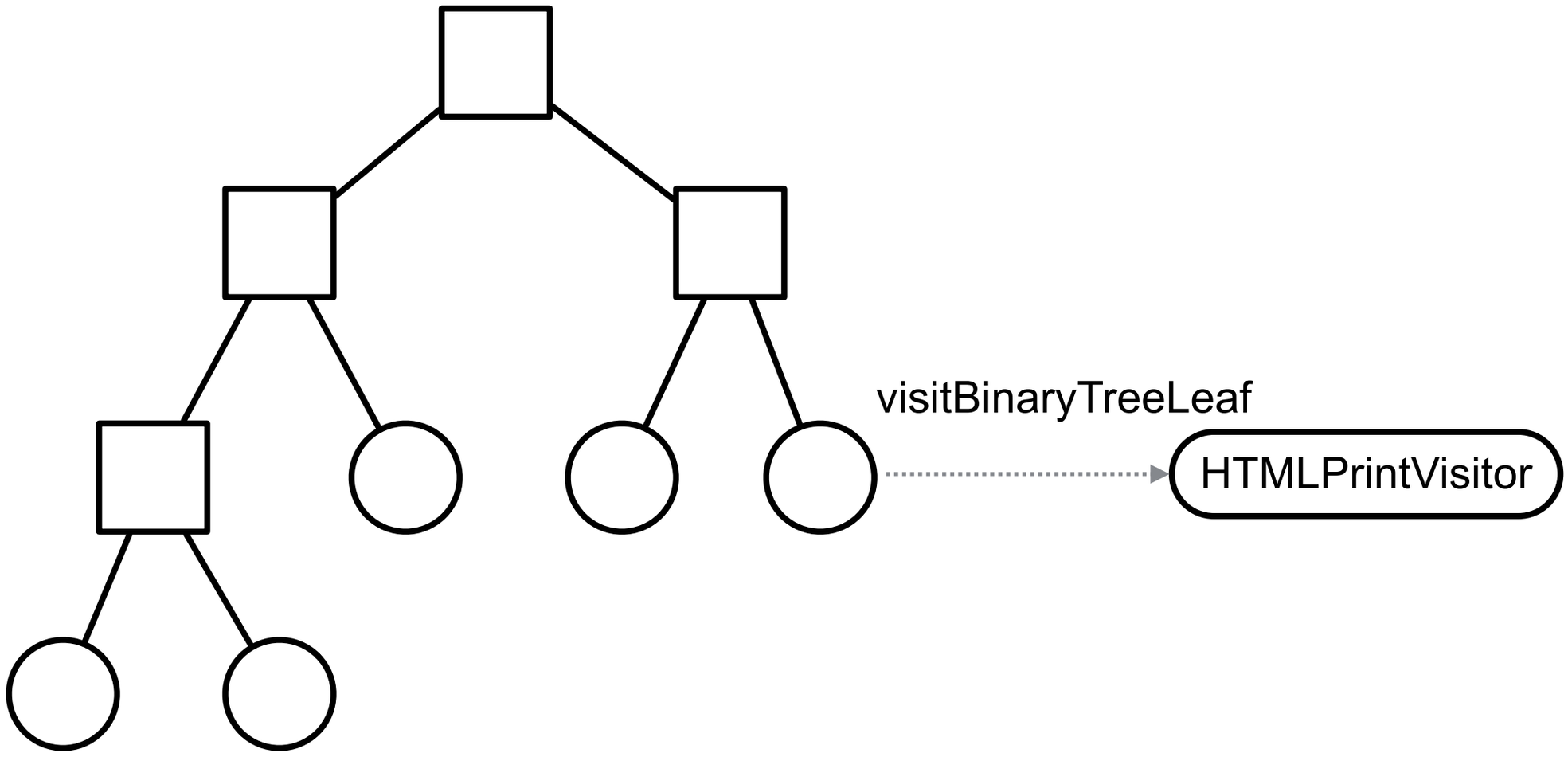
Tree Example



Tree Example



Tree Example



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

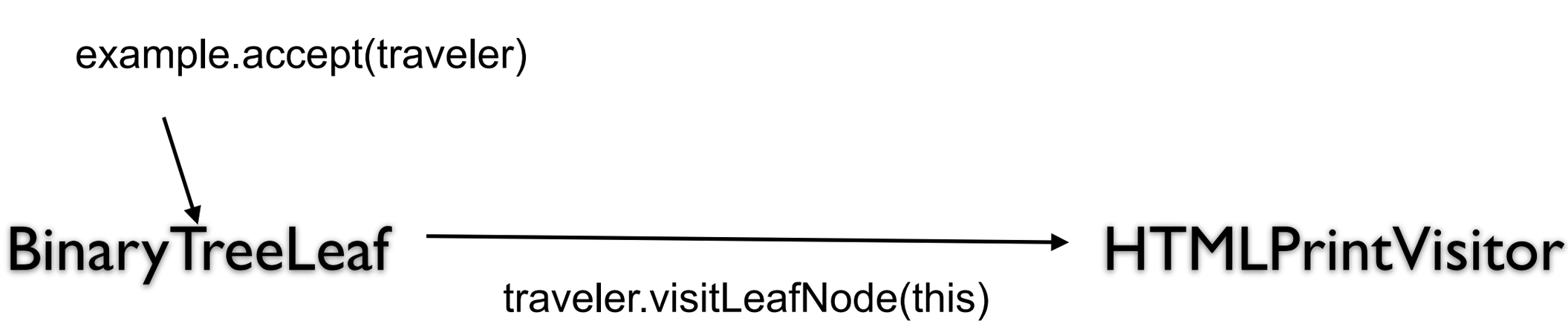
Each visitX method only deals with on type of element

```
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){ ...}
}
```

Double Dispatch

Note that a visit to one node requires two method calls

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



Why So Complicated?

Need to select methods to run at runtime based on:

- Type of Visitor
- Type of Document

Java & Python have single dispatch

- Can select method at run time based on receiver of the message

To select a method based on two types need to call two methods

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

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

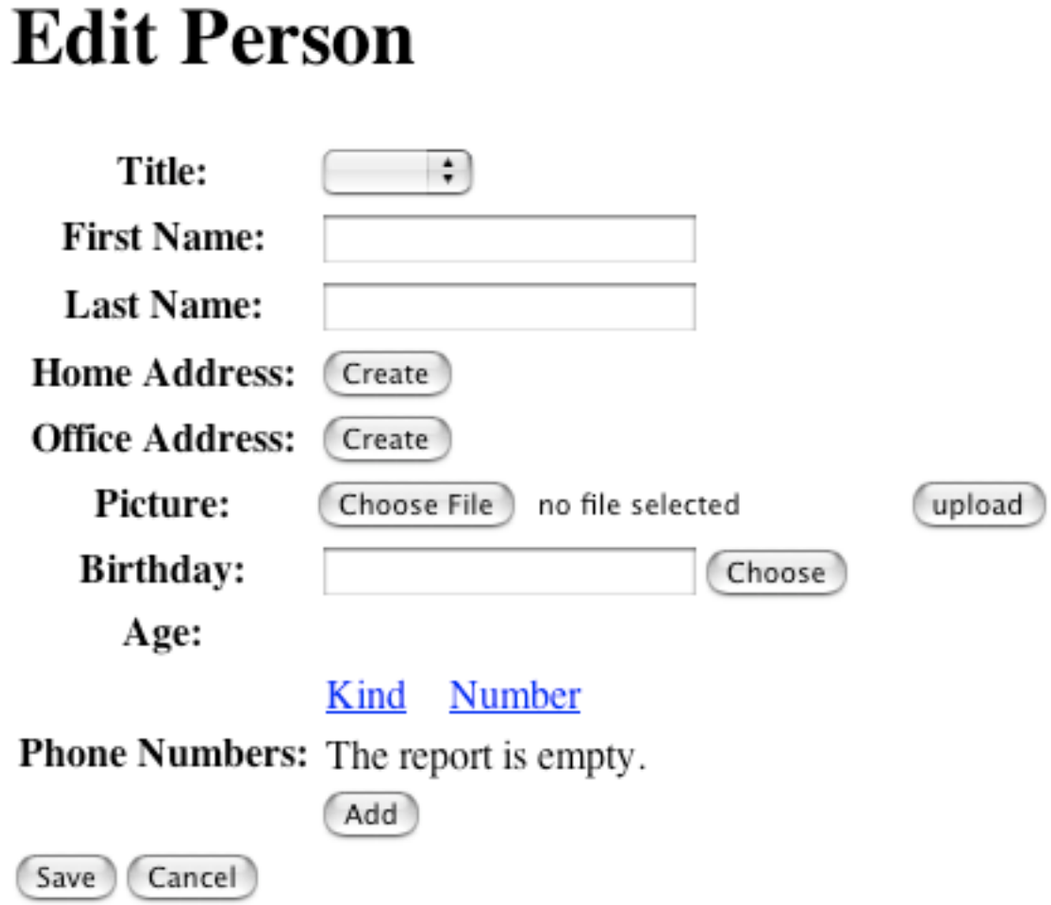
Each domain model has a collection of descriptions

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.
```



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

```
(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 & Julia

Multiple dispatch

- At run-time
- Based on argument types

No need for visitor pattern

Adapter

Adapter

Convert interface of a class into another interface

Use adapter when

- You want to use an existing class but does not have interface on needs
- You want to create a reusable class that works with unrelated or unforseen classes

Address Book & JTable

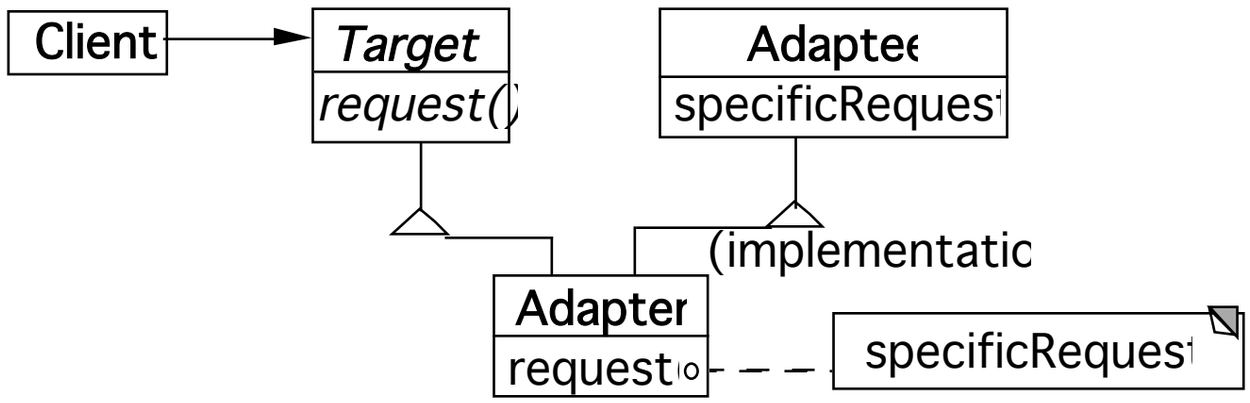
Display an AddressBook object in a JTable

JTables require objects of type TableModel

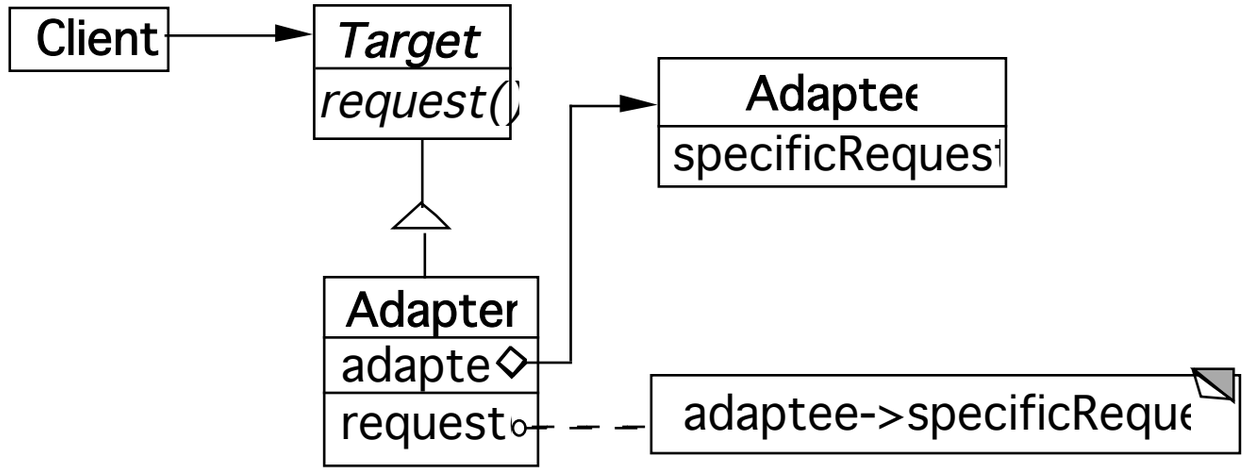
```
public class AddressBook{
    List personList;
    public int getSize(){...}
    public int addPerson(...){...}
    public Person getPerson(...){...}
    ...
}
```

```
public class AddressBookTableAdapter implements TableModel
{
    AddressBook ab;
    public AddressBookTableAdapter( AddressBook ab ){
        this.ab = ab;
    }
//TableModel impl
public getRowCount(){
    ab.getSize();

public Object getValueAt(int rowIndex, int columnIndex) {
    Person requested =
        ad.getPerson(convertRowToName(rowIndex);
return requested.get(convert(columnIndex));
}
```

Class Adapter



Object Adapter



Class Adapter Example

```
class OldSquarePeg {
    public: void squarePegOperation()  { do something }
}

class RoundPeg {
    public:  void virtual roundPegOperation = 0;
}

class PegAdapter: private OldSquarePeg,  public RoundPeg  {
    public:
        void virtual roundPegOperation()  {
            add some corners;
            squarePegOperation();
        }
}

void clientMethod() {
    RoundPeg* aPeg = new PegAdapter();
    aPeg->roundPegOperation();
}
```

Object Adapter

```
class OldSquarePeg{
    public: void squarePegOperation() { do something }
    }

class RoundPeg       {
    public: void virtual roundPegOperation = 0;
    }

class PegAdapter:  public RoundPeg       {
private:
    OldSquarePeg* square;

public:
    PegAdapter()  { square = new OldSquarePeg; }

    void virtual roundPegOperation()      {
        add some corners;
        square->squarePegOperation();
        }
}
```

How Much Adapting does the Adapter do?

Two-way Adapters

```
class OldSquarePeg {
    public:
        void virtual squarePegOperation() { blah }
}

class RoundPeg {
    public:
        void virtual roundPegOperation() { blah }
}

class PegAdapter: public OldSquarePeg, RoundPeg {
    public:
        void virtual roundPegOperation() {
            add some corners;
            squarePegOperation();
        }
        void virtual squarePegOperation() {
            add some corners;
            roundPegOperation();
        }
}
```

Flasher and MouseListener

```
class Flasher
    def toggle()
        @flashing = !@flashing
    end

    def pause()
        #etc
    end

    def resume()
        #etc
    end
end
```

```
class MouseListener
    def mouseClicked(event)
    end

    def mouseEntered(event)
    end

    def mouseExited(event)
    end
end
```

Actions we want

mouse click toggles flasher
mouse enter pauses
mouse exits resumes

Flasher as MouseListener

```
class Flasher
    def toggle()
        @flashing = !@flashing
    end

    def pause()
        #etc
    end

    def resume()
        #etc
    end

    def mouseClicked(event)
        toggle()
    end

    def mouseEntered(event)
        pause()
    end

    def mouseExited(event)
        resume()
    end
end
```

Simple Adapter

```
class Flasher
    def toggle()
        @flashing = !@flashing
    end

    def pause()
        #etc
    end

    def resume()
        #etc
    end
end
```

```
class FlasherAdaptor
    def initialize(aFlasher)
        @flasher = aFlasher
    end

    def mouseClicked(event)
        @flasher.toggle()
    end

    def mouseEntered(event)
        @flasher.pause()
    end

    def mouseExited(event)
        @flasher.resume()
    end
end
```

```
yellowFlasher = Flasher.new(yellow, fast)
FlasherAdapter.new(yellowFlasher)
```

A Ruby Adapter - Fowardable

```
class Flasher
    def toggle()
        @flashing = !@flashing
    end

    def pause()
        #etc
    end

    def resume()
        #etc
    end
end
```

```
require 'forwardable'

class FlasherMouseListener
    extend Forwardable

    def initialize()
        @flasher = Flasher.new()
     end
    def_delegator(:@flasher, :toggle, :mouseClick)
    def_delegator(:@flasher, :pause, :mouseEnter)
    def_delegator(:@flasher, :resume, :mouseExit)

end


adaptor = FlasherMouseListener.new()
adaptor.mouseClick()
```

Parameterized Adapter

```
class MouseListenerAdapter
    def initialize(adaptee, clickMethod, enterMethod, exitMethod)
        @adaptee = adaptee
        @clickMethod = clickMethod
        @enterMethod = enterMethod
        @exitMethod = exitMethod
    end

    def mouseClicked(event)
        @adaptee.send(clickMethod)
    end

    def mouseEntered(event)
        @adaptee.send(clickMethod)
    end

    def mouseExited(event)
        @adaptee.send(clickMethod)
    end
end
```

```
yellowFlasher = Flasher.new(yellow, fast)
MouseListenerAdapter.new(
                        yellowFlasher,
                        :toggle,
                        :pause,
                        :resume)
```

Better Parameterized Adapter

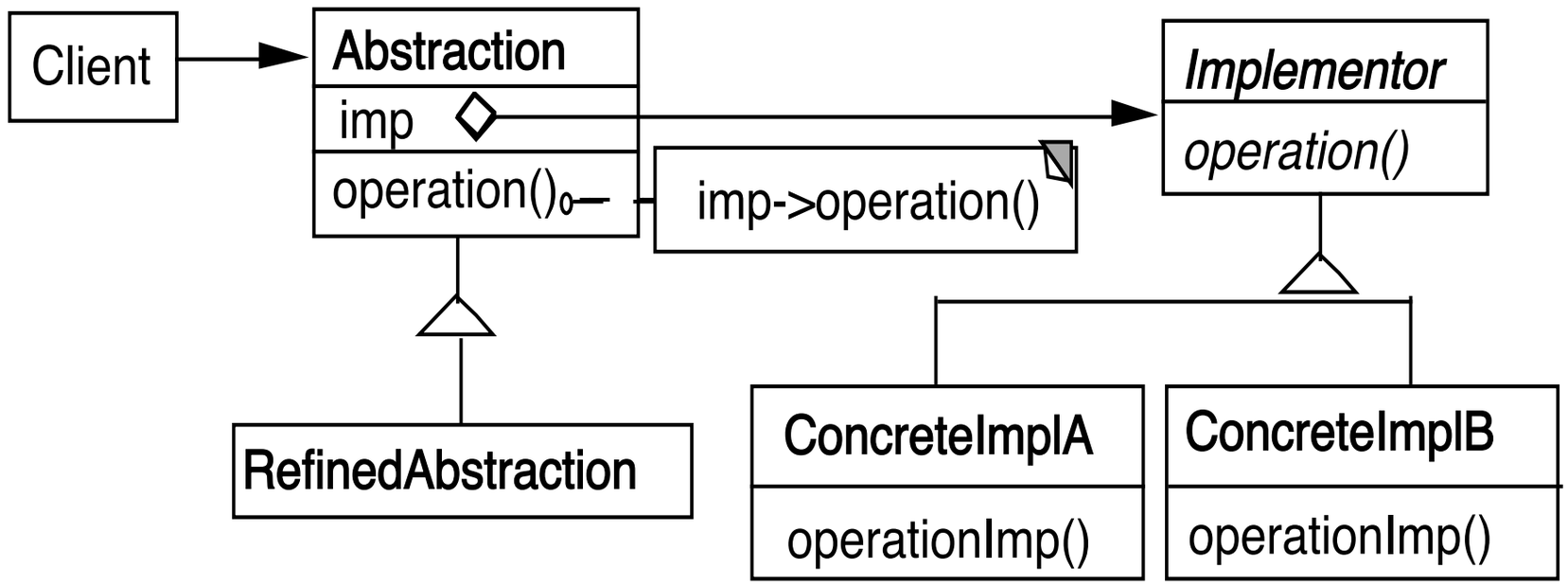
```
class MouseListenerAdapter
    def initialize(adaptee, clickLambda, enterLambda, exitLambda)
        @adaptee = adaptee
        @clickLambda = clickLambda
        @enterLambda = enterLambda
        @exitLambda = exitLambda
    end

    def mouseClicked(event)
        @clickLambda.call(adaptee)
    end

    def mouseEntered(event)
        @enterLambda.call(adaptee)
    end

    def mouseExited(event)
        @exitLambda.call(adaptee)
    end
end
```

```
yellowFlasher = Flasher.new(yellow, fast)
MouseListenerAdapter.new(
        yellowFlasher,
        lambda {|flasher | flasher.toggle()},
        lambda {|flasher | flasher.pause()},
        lambda {|flasher | flasher.resume()})
```

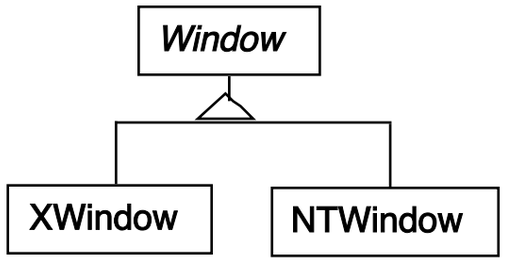
Bridge

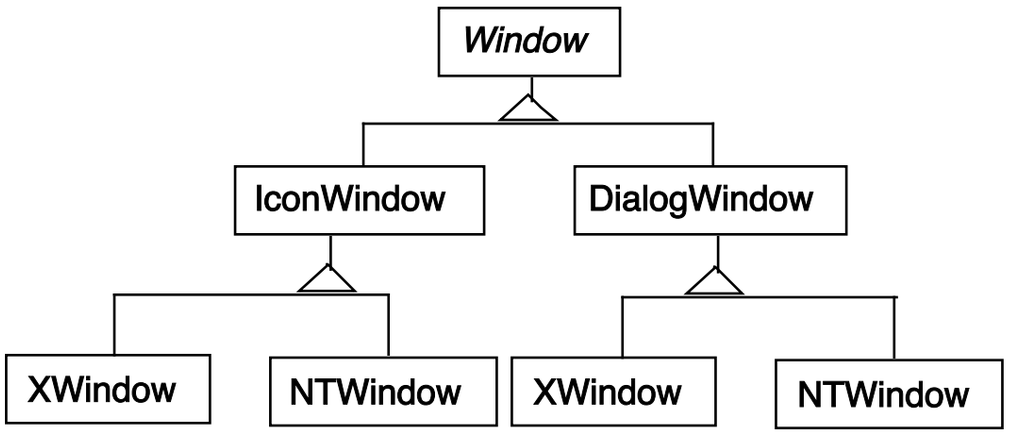
Bridge

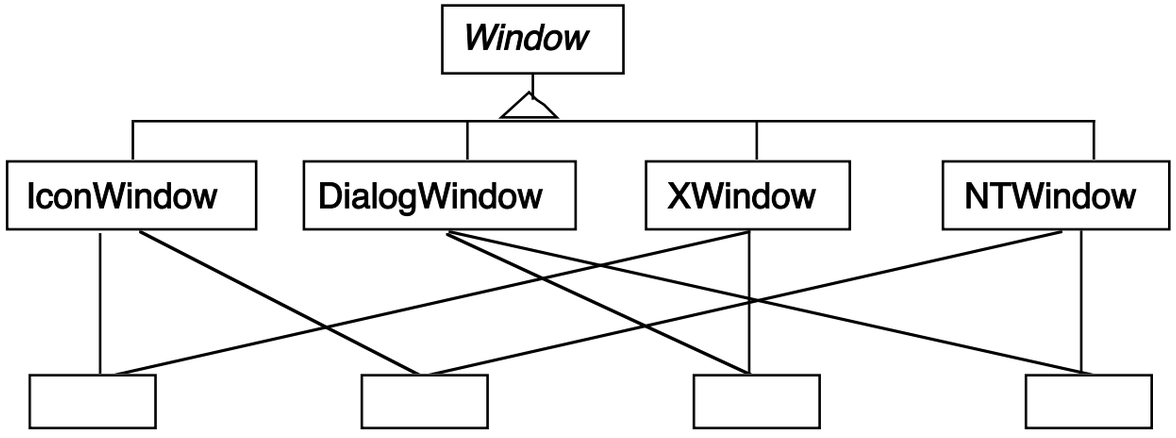
Decouple an abstraction from its implementation



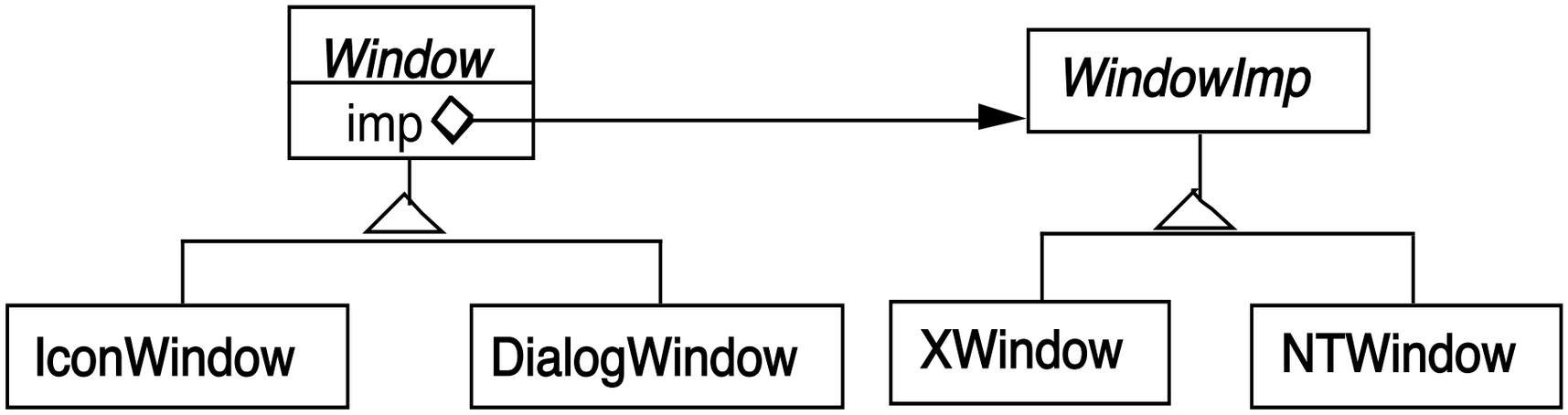
Windows


Window
XWindow
NTWindow


Window
IconWindow
DialogWindow
XWindow
NTWindow
XWindow
NTWindow


Window
IconWindow
DialogWindow
XWindow
NTWindow

Using the Bridge Pattern

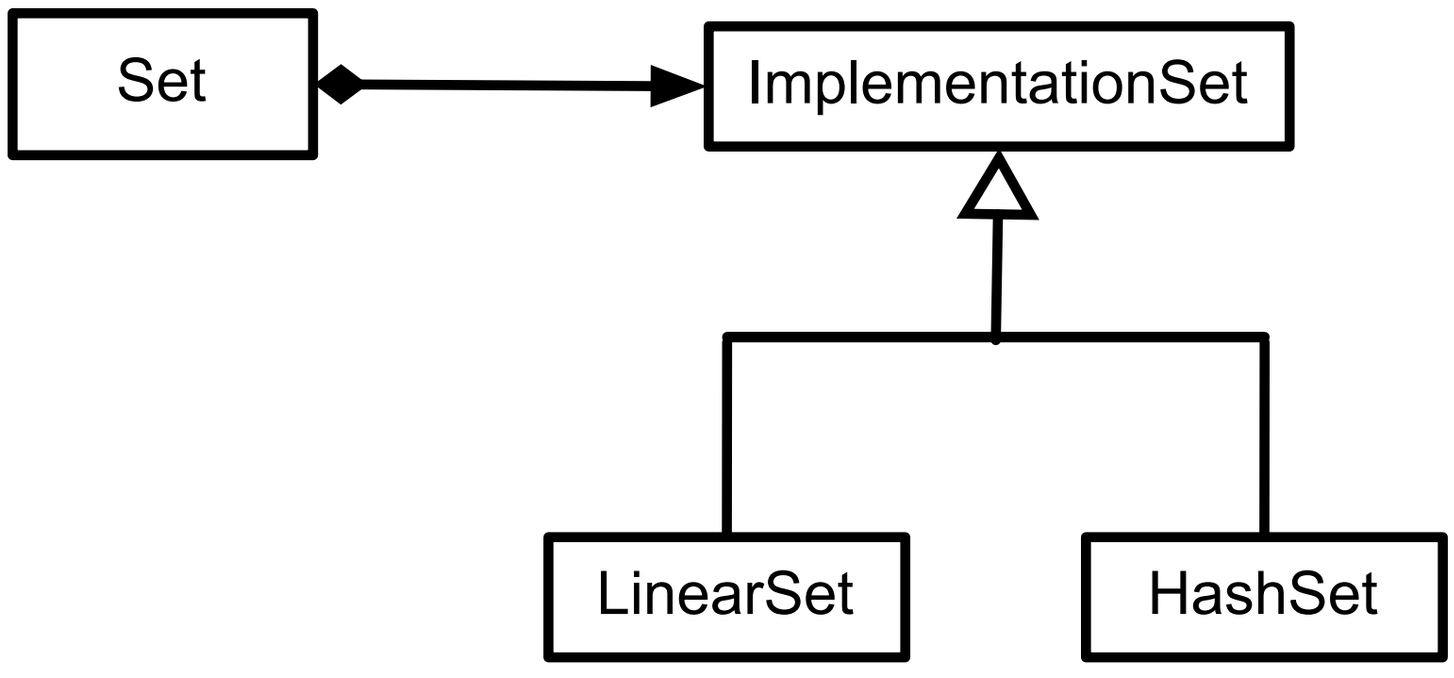


Peers in Java's AWT

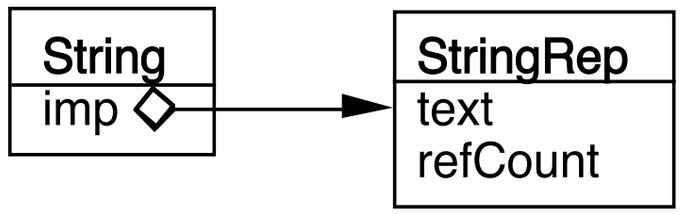
Peer = implementation

```
    public synchronized void setCursor(Cursor cursor) {
        this.cursor = cursor;
        ComponentPeer peer = this.peer;
        if (peer != null) {
            peer.setCursor(cursor);
        }
    }
```

IBM Smalltalk Collections



Smart Pointers in C++



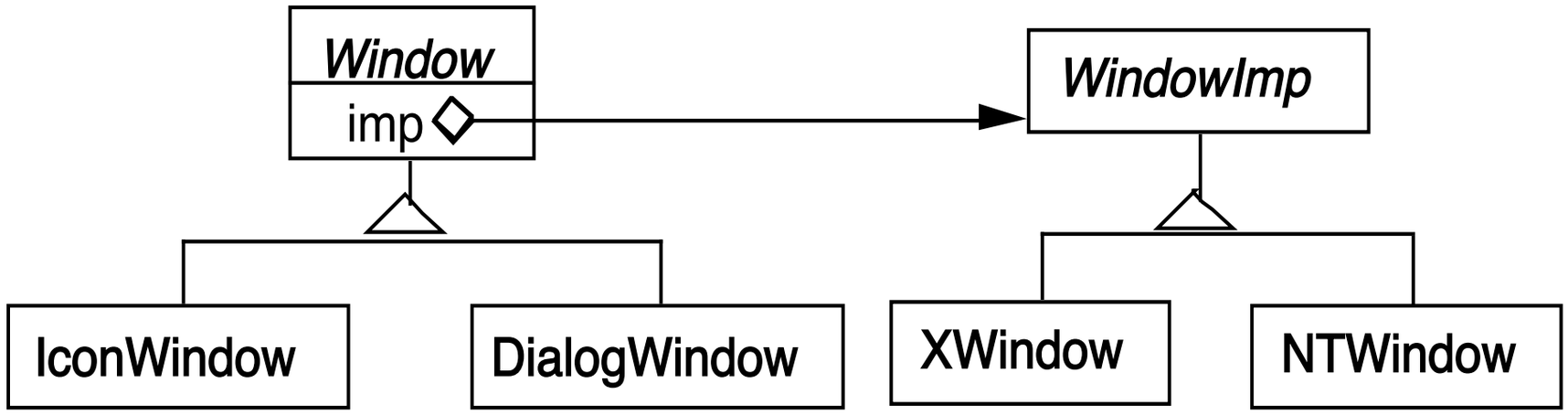
Why Use Bridge

Implementation selected at run-time

Implementation changed during run-time

Why Use Bridge

Abstraction & implementations are extensible by subclassing

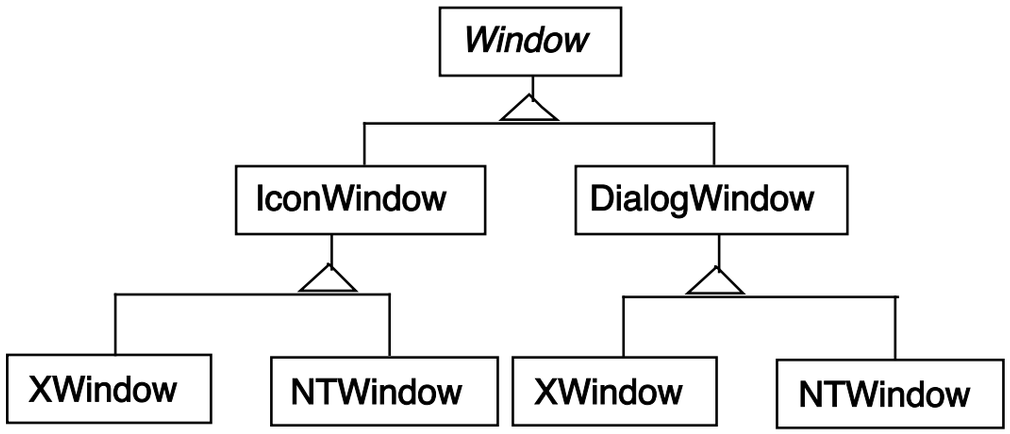


Why Use Bridge

When changes in the implementation should not require client code to be recompiled

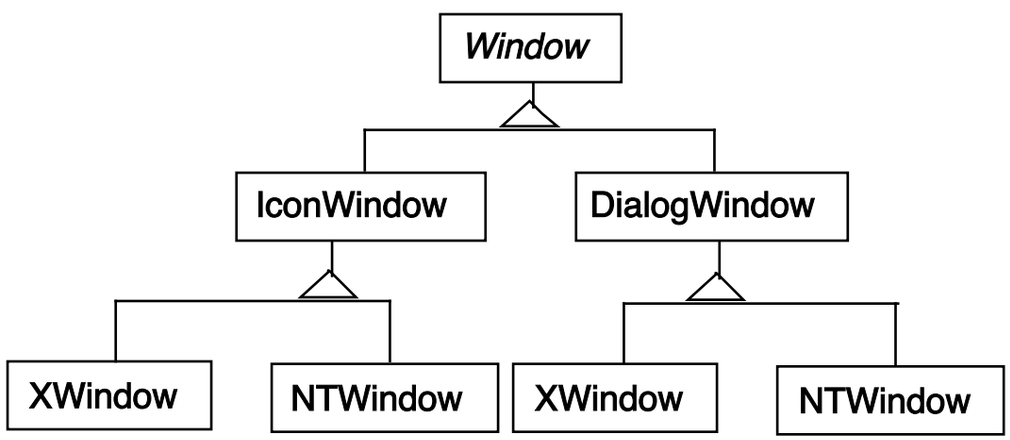
Why Use Bridge

Proliferation of classes



Why Use Bridge

Share implementation among multiple objects



Bridge verses Adapter