CS 635 Advanced Object-Oriented Design & Programming
Fall Semester, 2022
Doc 10 State, Visitor
Sept 27, 2022

The Rule of Three

If you can not think of three things that might go wrong with your plans (or software design), there is something wrong with your thinking

Gerald M. Weinberg

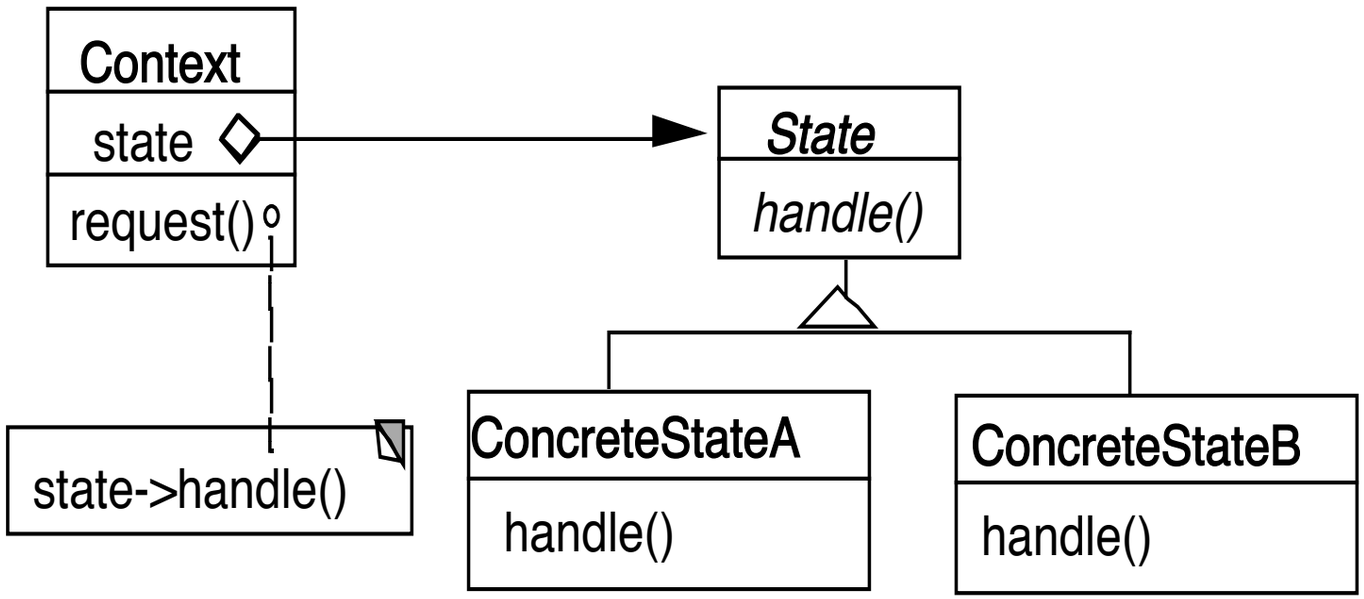
State

State Pattern

Allow an object to alter its behavior when its internal state changes

The object will appear to change it class

Structure


Context
state
request()
State
handle()
state->handle()
ConcreteStateA
handle()
ConcreteStateB
handle()

Grade Program

Operations

- View assignment dates
- Log in
- View grades
- Post grades

States

Not logged in

- View dates, Log in

Invalid operations

- View & post grades

Logged in - student

- View dates & grades

Invalid operations

- Post grades, log in

Logged in - instructor

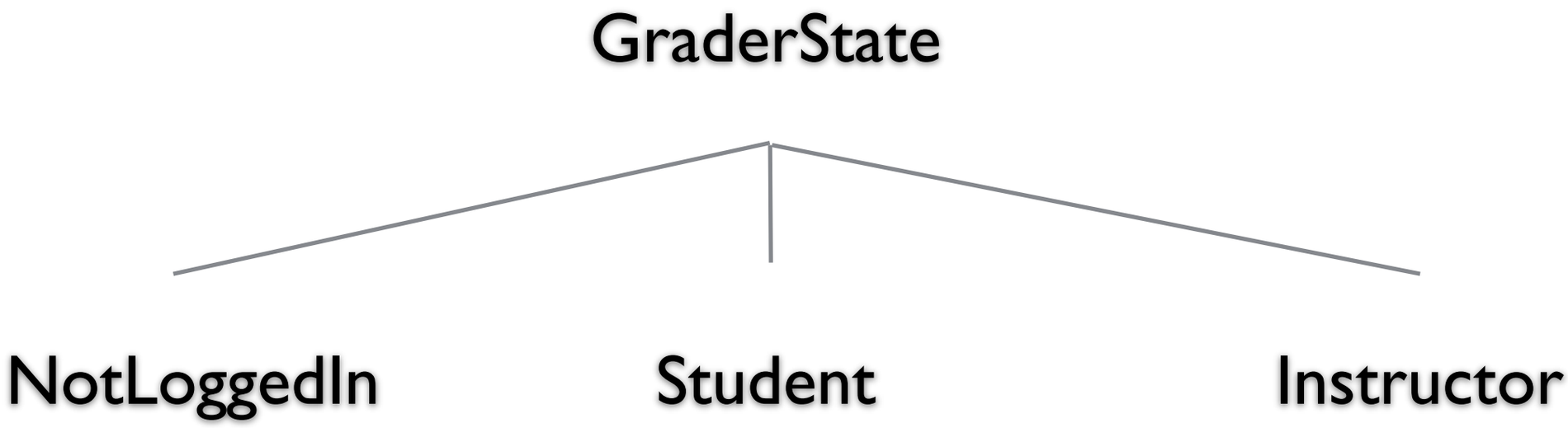
- View dates & grades,
- post grades

Invalid operations

- log in

```
public class Grader {
   static final int NOT_LOGGED_IN = 0;
   static final int STUDENT = 1;
   static final int INSTRUCTOR = 2;
   int state = NOT_LOGGED_IN;

   public viewGrades() {
      if (state == NOT_LOGGED_IN)
         redirectToLogin();
      if (state == STUDENT)
         showStudentGrade();
      if (state == INSTRUCTOR)
         showAllGrades();
   }
}
```

```
public postGrades() {
   if (state == NOT_LOGGED_IN)
      redirectToLogin();
   if (state == STUDENT)
      showError();
   if (state == INSTRUCTOR)
      getGradeFile();
}
```




```
public class GraderState {
   public GraderState login() {...}
   public GraderState viewGrades() {}
   public GraderState postGrades() {}
}
```

```
public class Grader {
   GraderState state = new NotLoggedIn();

   public void login() {
      state = state.login();
   }

   public viewGrades() {
      state = state.viewGrades();
   }
}
```

```
public class Student : GraderState {
   public GraderState login() {
      displayError(); // already login
      return self;
   }

   public GraderState viewGrades() {
      fetchStudentsGrades();
      display
   }

   public GraderState postGrades() {
      logStudentAttempt();
      displayError();
      return this;
   }
}
```

Who defines state Transitions - Context

```
class Context {
    private AbstractState state = new StartState();

    public Bar foo(int x) {
        int result = state.foo(x);
        if (someConditionHolds() )
            state = nextState();
    return result;
    }
```

Who defines state Transitions - States

```
class Context {
    private AbstractState state = new StartState();

    public void foo(int x) {
        state = state.foo(x);
        }
```

What if foo returns a value?

Who defines state Transitions - States

```
class Context {
    private AbstractState state = new StartState();

    public int foo(int x) {
        return state.foo(x, this);
    }

    protected void setState(AbstractState newState) {
        state = newState;
    }
```

Sharing State Objects

Stateless state

- State objects without fields
- Can be shared by multiple contexts

Can store date in context and pass as arguments

Large number of state transitions can be expensive

- Only create state once & reuse same object

State Verses Strategy

Rate of Change

Strategy
Context usually contains just one strategy object

State
Context often changes state objects

State Verses Strategy

Exposure of Change

Strategy

Strategies all do the same thing

Client do not see change in behavior of Context

State

States act differently

Client see the change in behavior

Changing Class - No Need for Context

Language Dependent Feature
Smalltalk & Lisp

```
class Truthful extends Oracle {

    public boolean foo(int x) {
        int result = state.foo(x);
        this.changeClassTo(Random);
        return result;
    }
```

Java/C++/C# Example - Single Dispatch

```
public class Parent {
}

public class Child extends Parent {
}

public class Bar {

    public foo(Parent x) {
       return “Parent”;
    }

    public foo(Child x) {
       return “Child”;
    }
}
```

```
Bar test = new Bar();

Parent x = new Parent();
test.foo(x);                    Parent

x = new Child();
test.foo(x);                    Parent

Child y = new Child();
test.foo(y);                    Child
```

Actual type of foo's argument is not used
Just the declared type

Only runtime selection(dispatch) is on type actual type of receive of method

Multiple Dispatch - Julia

```
foo(a::Integer,b::Integer) = "Integer,Integer"
foo(a::Integer,b::Number) = "Integer,Number"
foo(a::Number,b::Integer) = "Number,Integer"
foo(a::Number,b::Number) = "Number,Number"
foo(a::Number,b::Complex) = "Number,Complex"
```

foo(1,2)	Integer,Integer
foo(1,2.3)	Integer,Number
foo(2//3,1)	Number,Integer
foo(2.3,2im + 2)	Number,Complex

Why Important

```
function power(n::Number,exponent::Real)
     Some complicated process for float exponent
end

function power(n::Number,exponent::Complex)
    Deal with complex exponent
end
```

```
function power(n::Number, exponent::Integer)
  if exponent == 0
    return 1
  result = n
  for k in 1:n-1
    result *= n
  end
  result
end
```

Open & Closed

A module is open if can

- Add/remove fields
- Add/remove methods

A module is closed if

- It can be used by other modules

Open-Close Principle

Module should be open for extension

But closed for modification

Multiple Dispatch & State Pattern - Julia

```
function viewGrades(user::NotLoggedIn)
  goToLogInPage(user)
end

function viewGrades(user::Student)
  getAndDisplayGrades(user)
end
```

Multiple Dispatch & State Pattern - Clojure

```
(defmulti view-grades (fn [user] (:state user))

(defmethod view-grades :not-logged-in
  [user]
  (go-to-log-in-page user))

(defmethod view-grades :student
  [user]
  (student-grade user))

(defmethod view-grades :instructor
  [user]
  (all-course-grades user)))
```

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

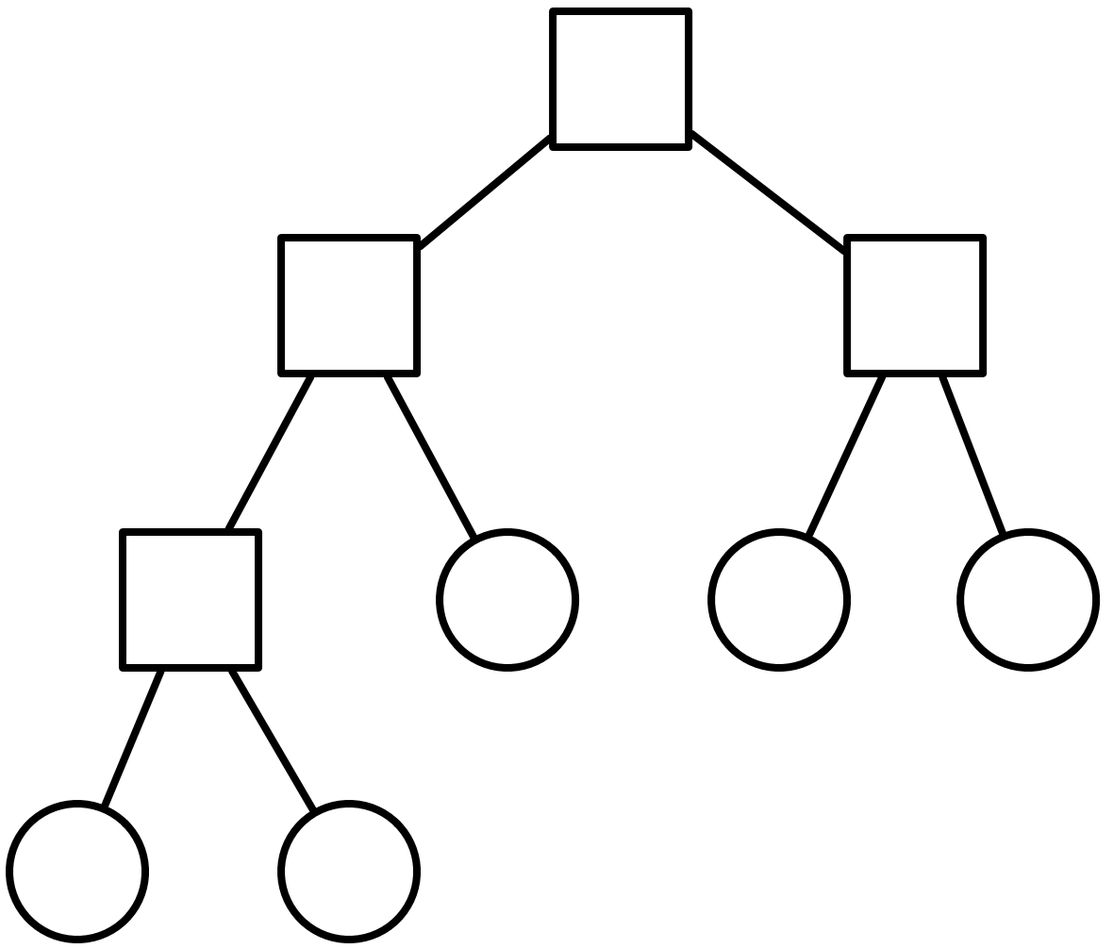
Visitor

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

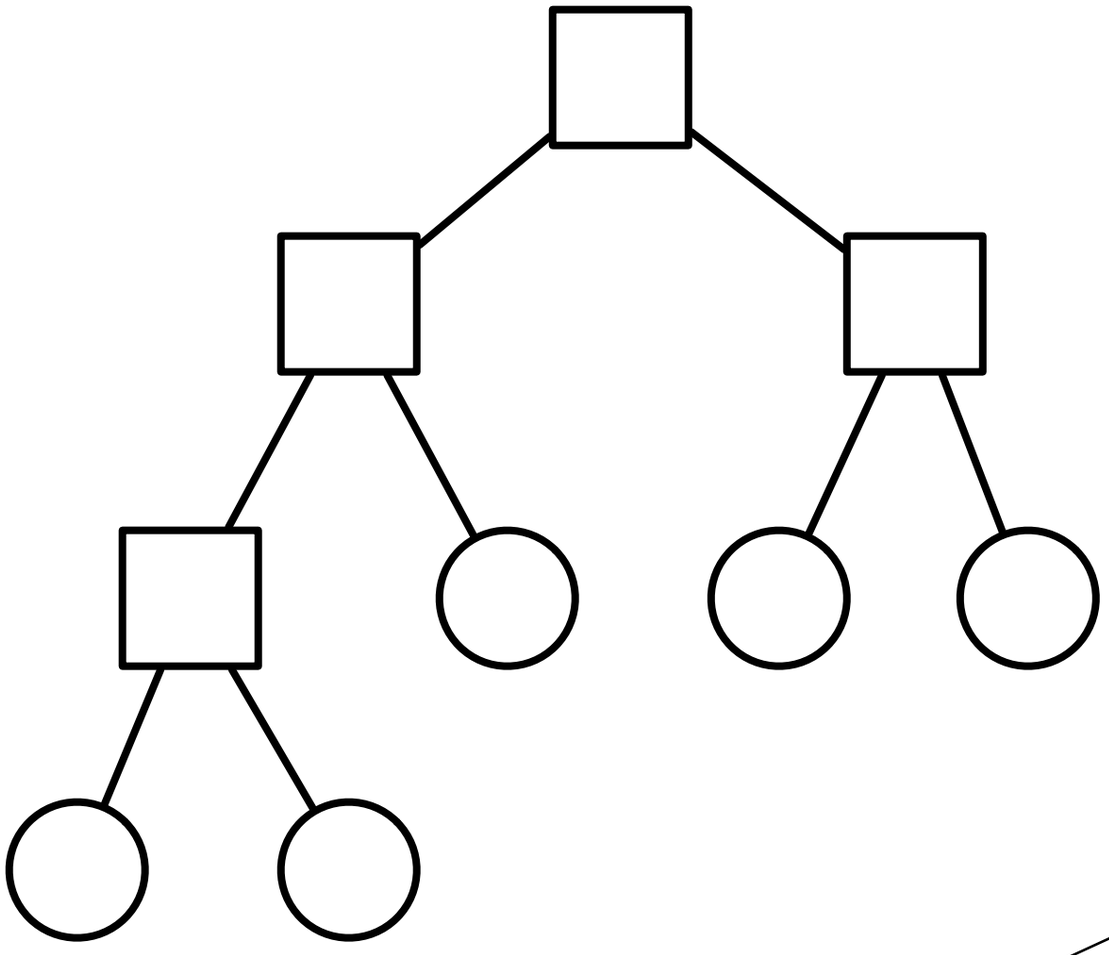
```
class Node {
   abstract public void accept(Visitor aVisitor);
}


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

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

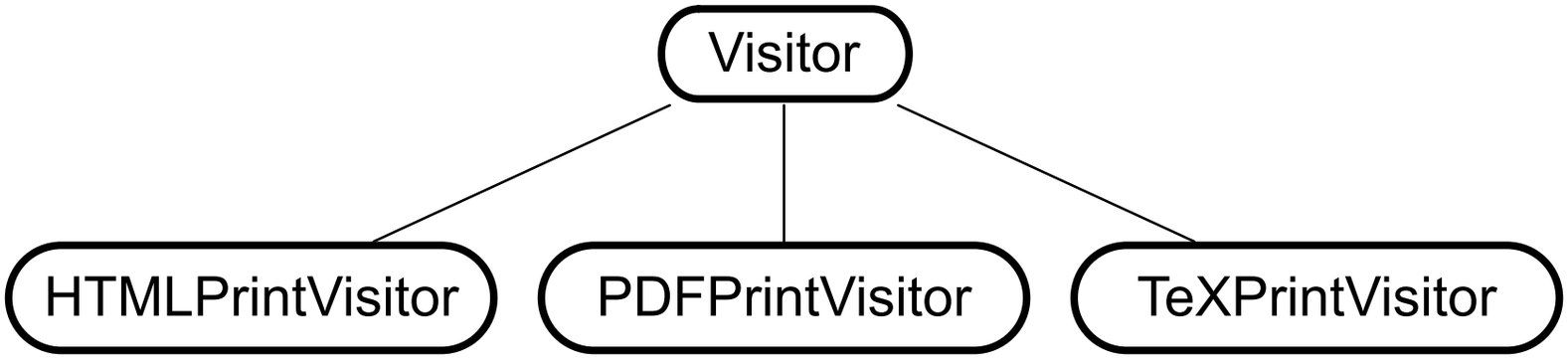
Visitor

```
abstract class Visitor {

    abstract void visitBinaryTreeNode( BinaryTreeNode );

    abstract void visitBinaryTreeLeaf( BinaryTreeLeaf );
}
```

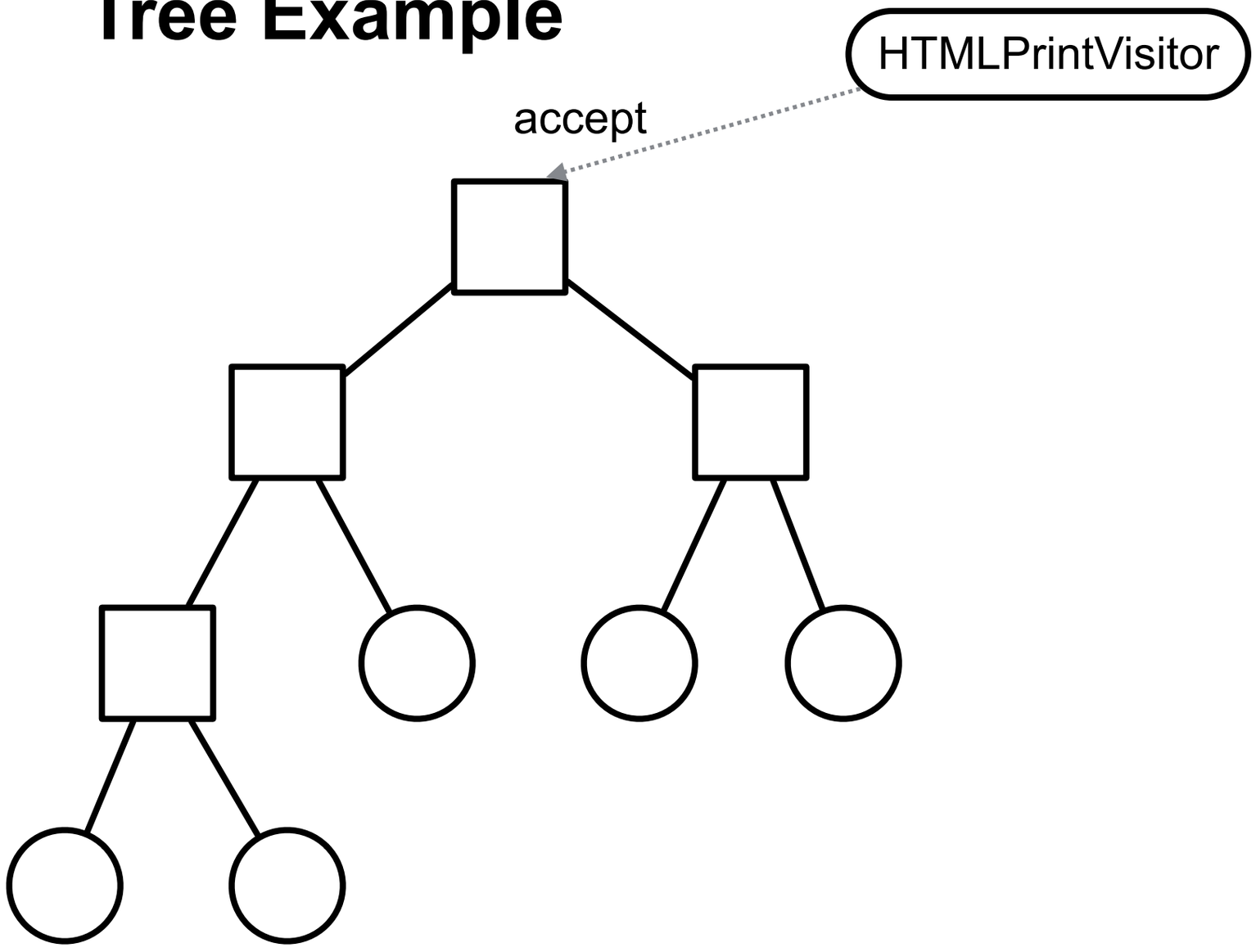
```
class HTMLPrintVisitor extends Visitor {

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

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

Tree Example

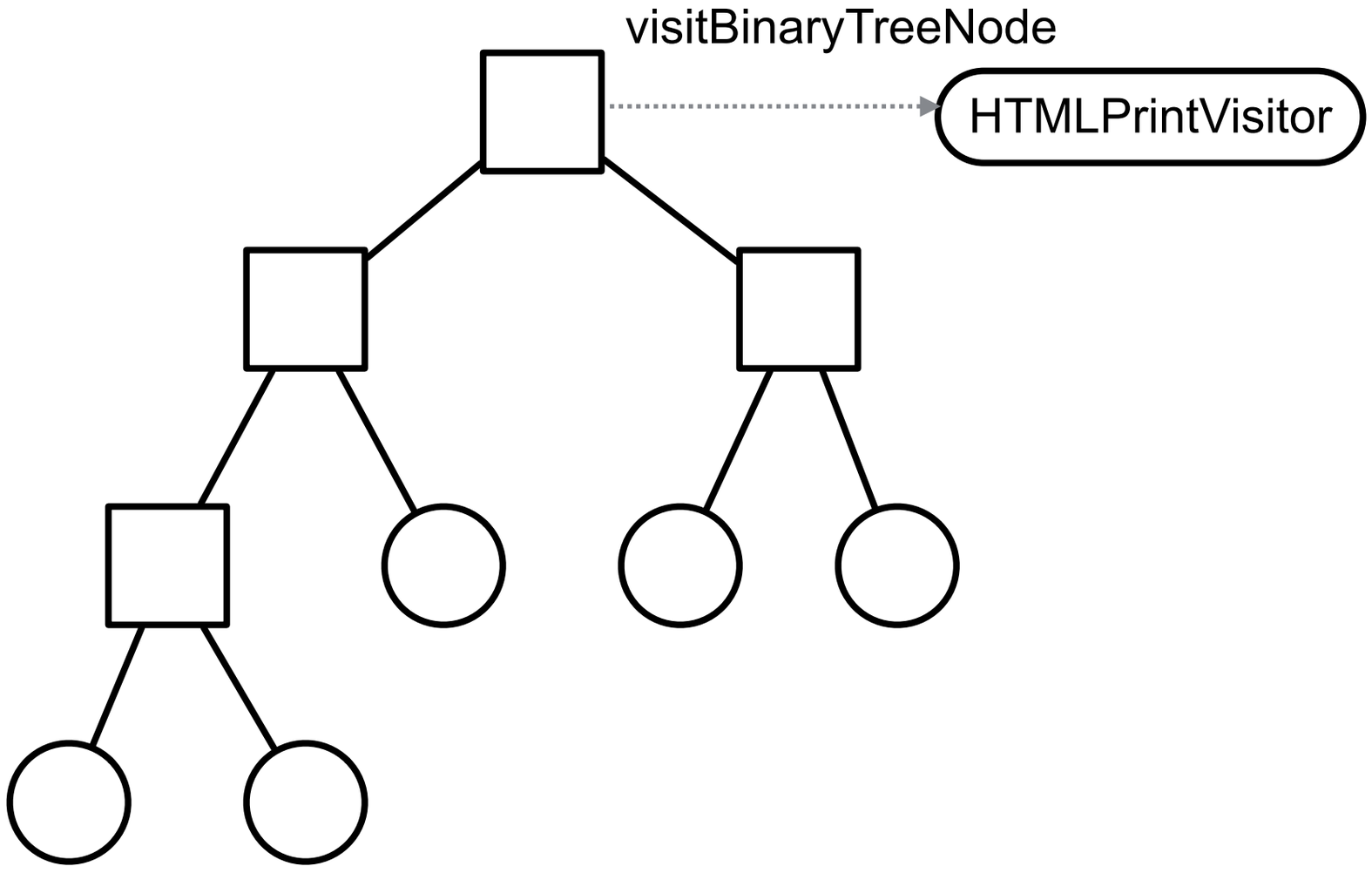




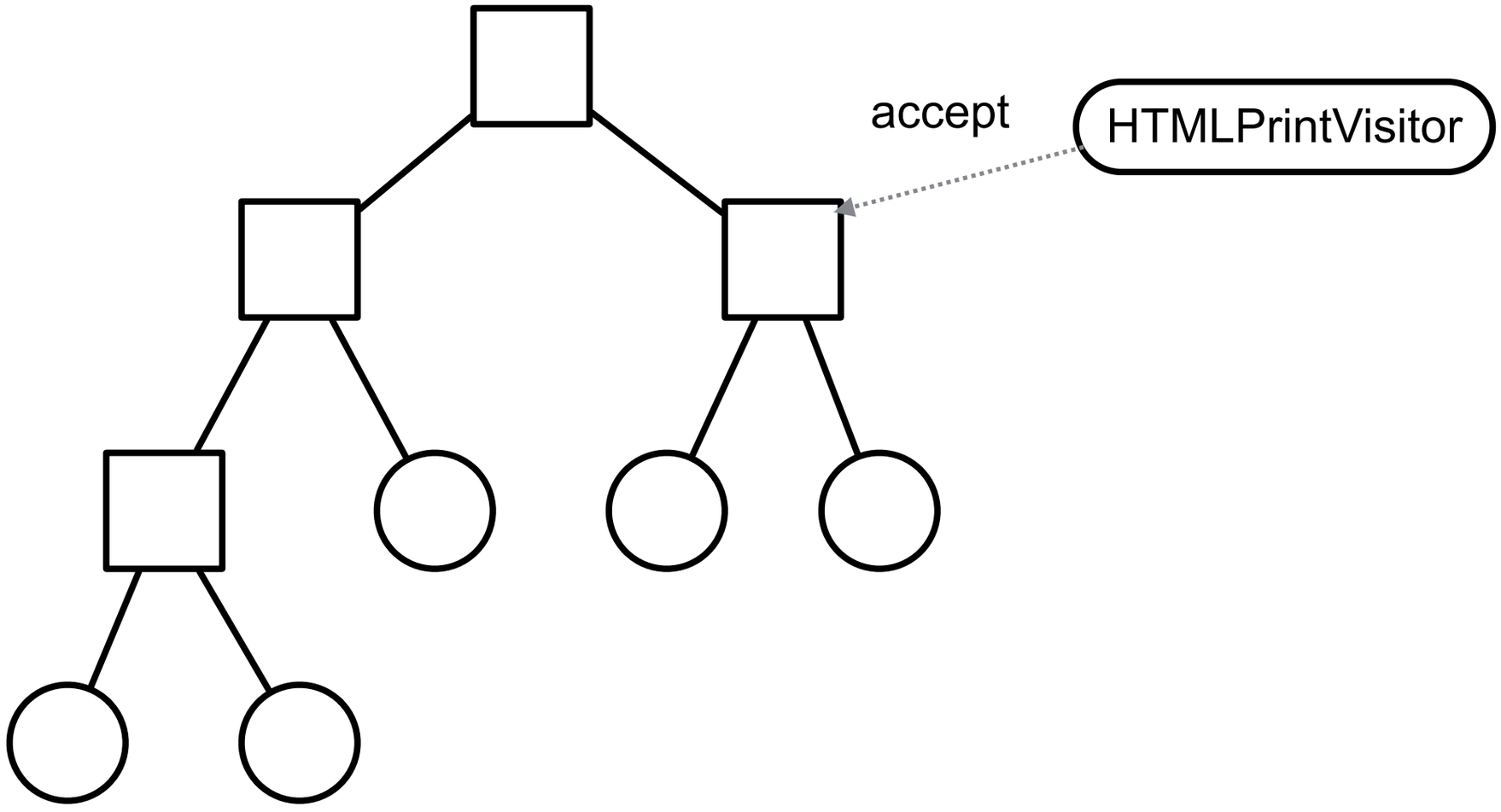
Tree Example



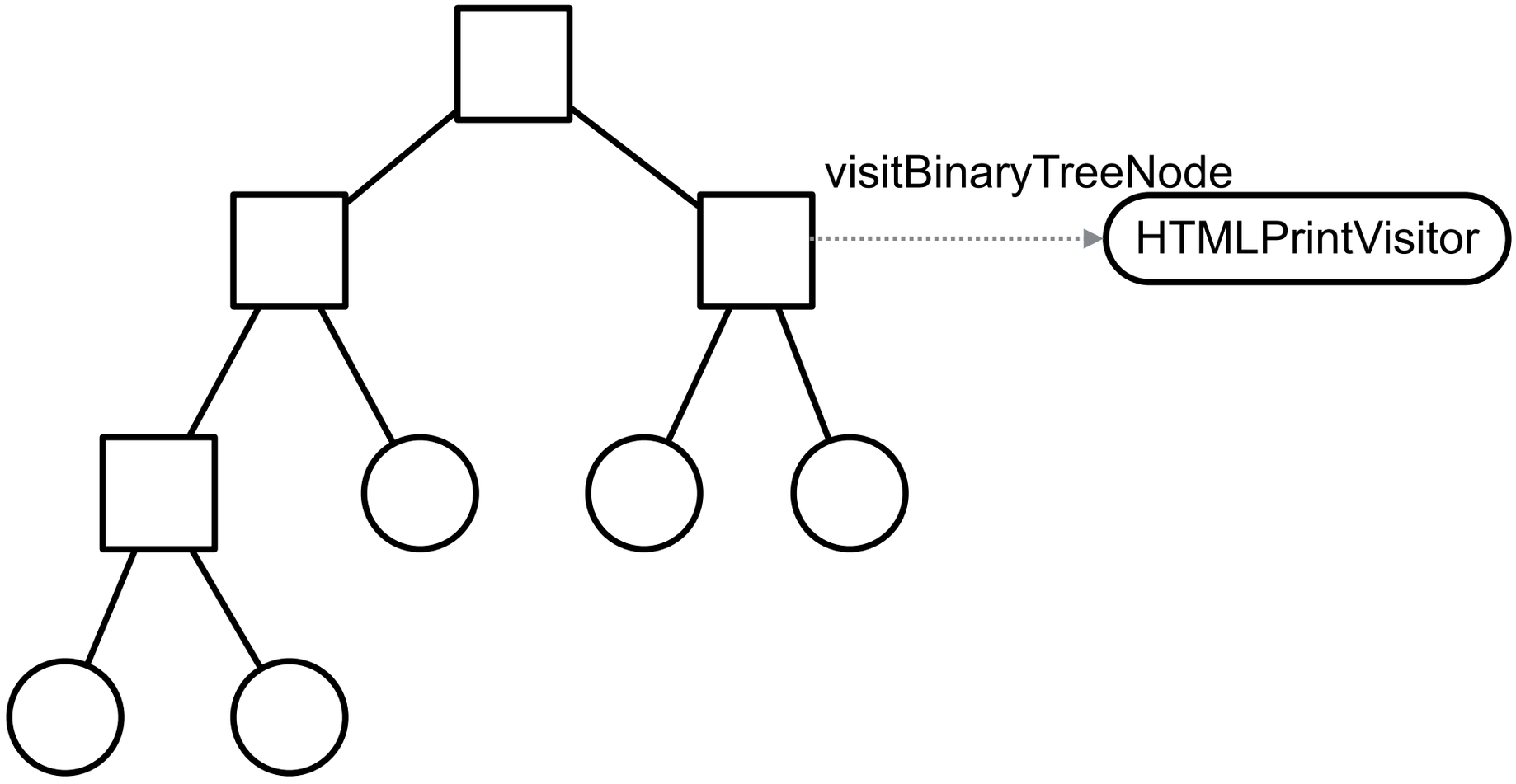
Tree Example



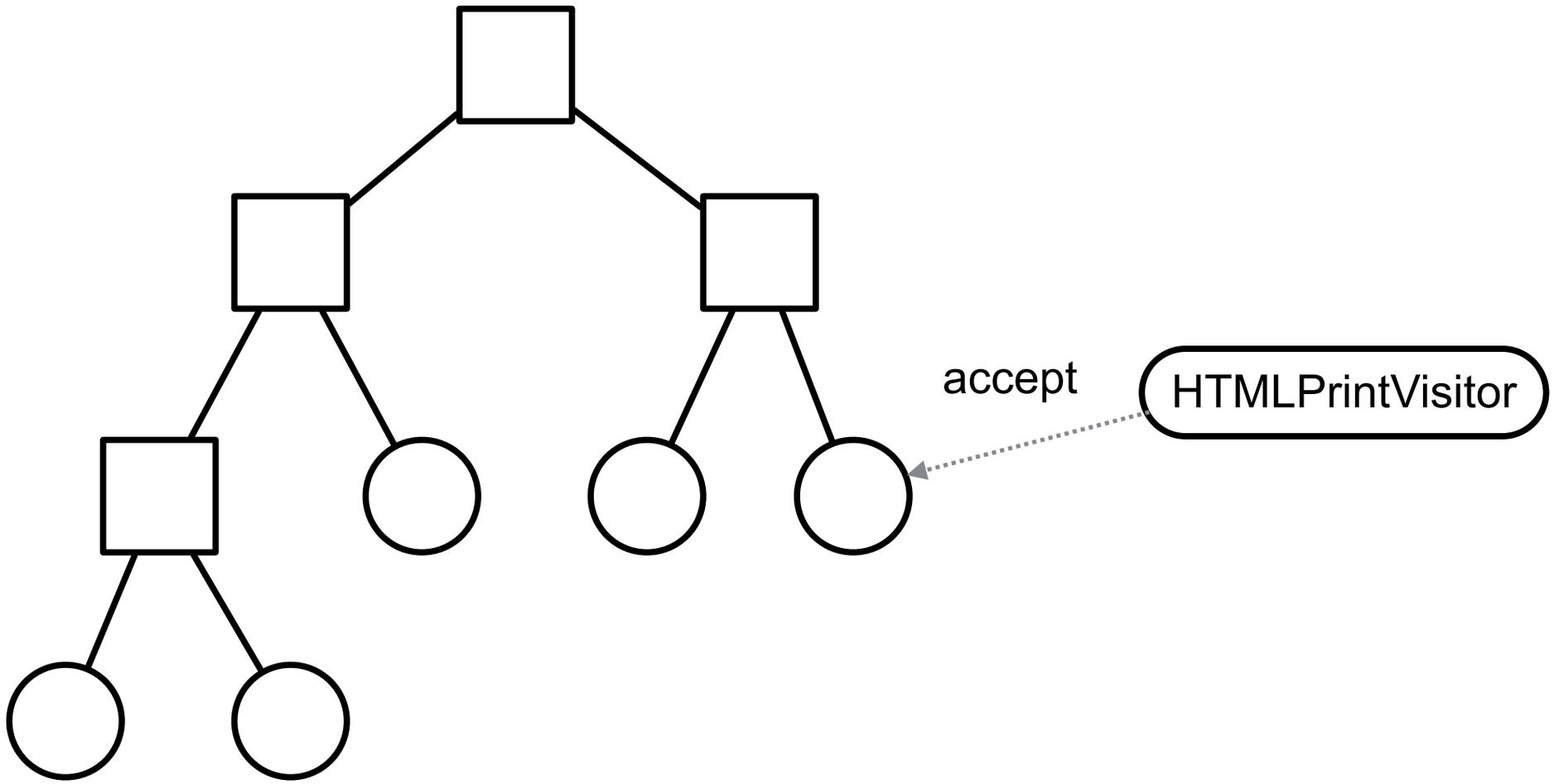
Tree Example



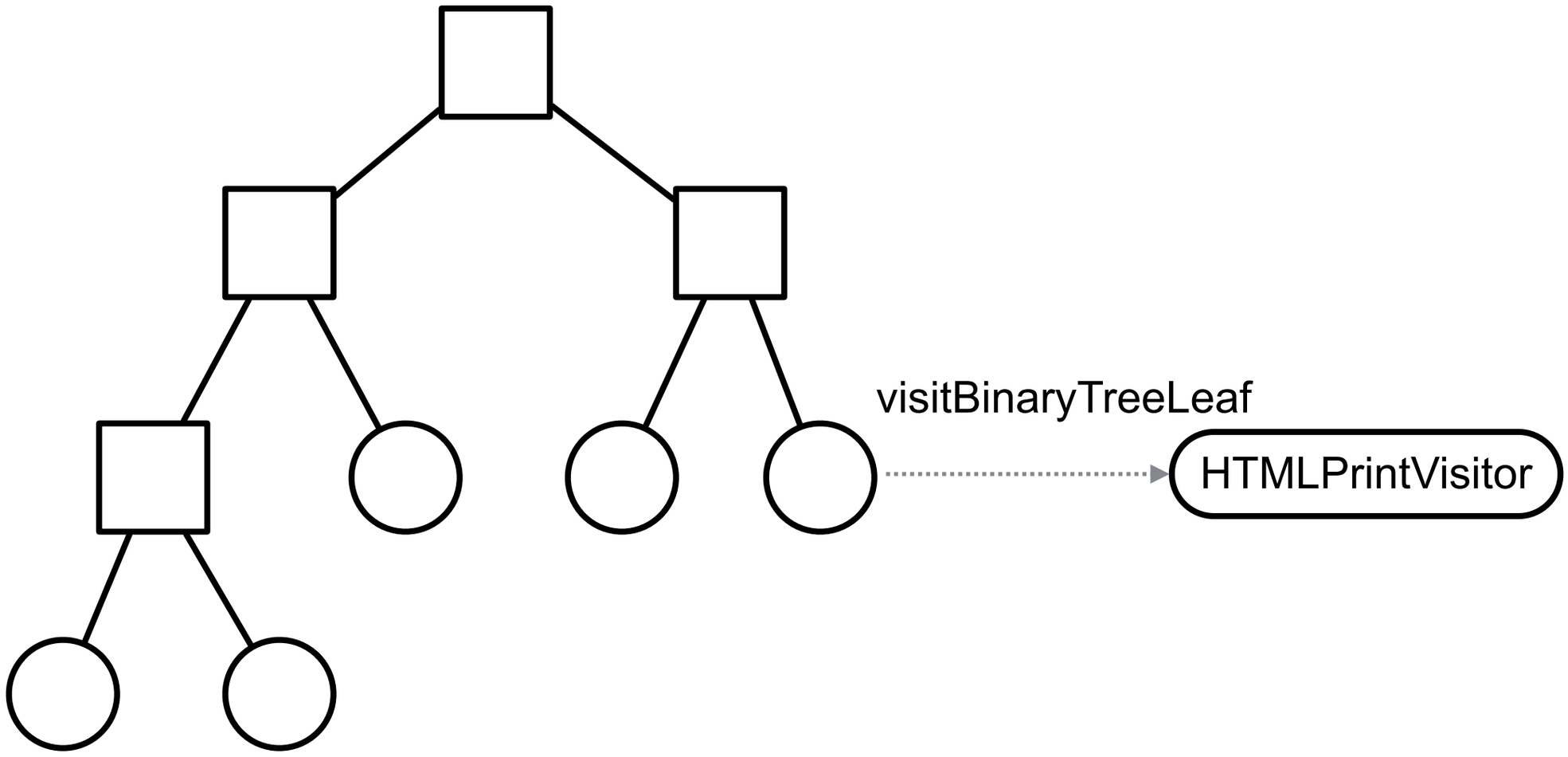
Tree Example



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

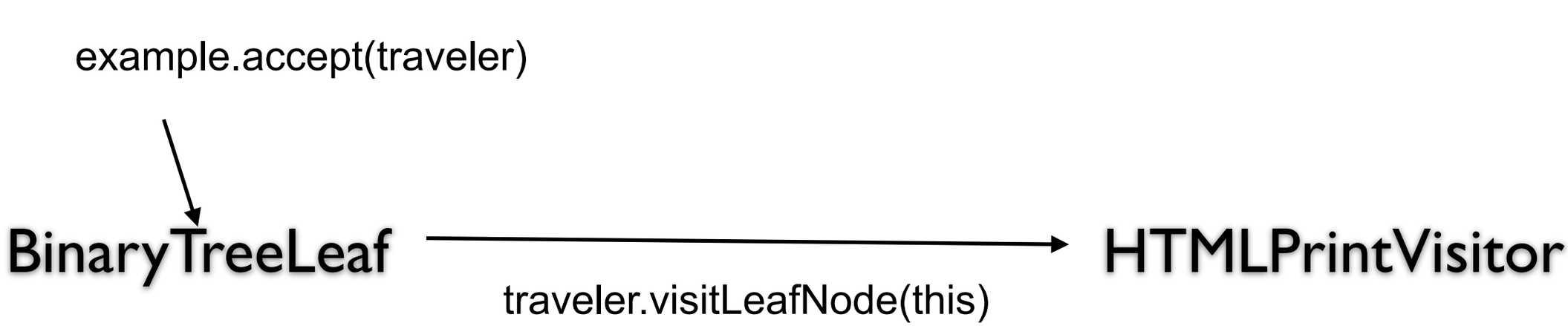
Each visitX method only deals with on type of element

```
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

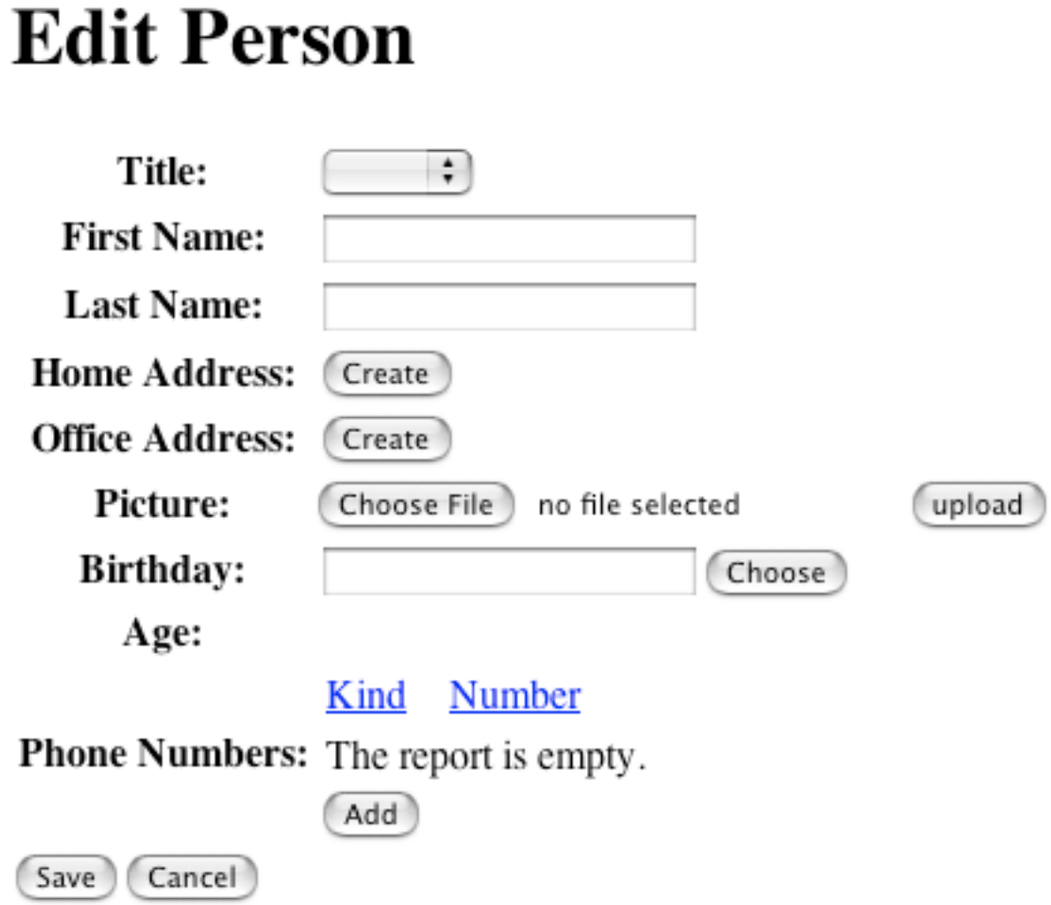
Each domain model has a collection of descriptions

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