

CS 635 Advanced Object-Oriented Design & Programming
Fall Semester, 2020
Doc 10 Visitor
Sep 24, 2020

Copyright ©, All rights reserved. 2020 SDSU & Roger Whitney,
5500 Campanile Drive, San Diego, CA 92182-7700 USA.
OpenContent (<http://www.opencontent.org/opl.shtml>) license
defines the copyright on this document.

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

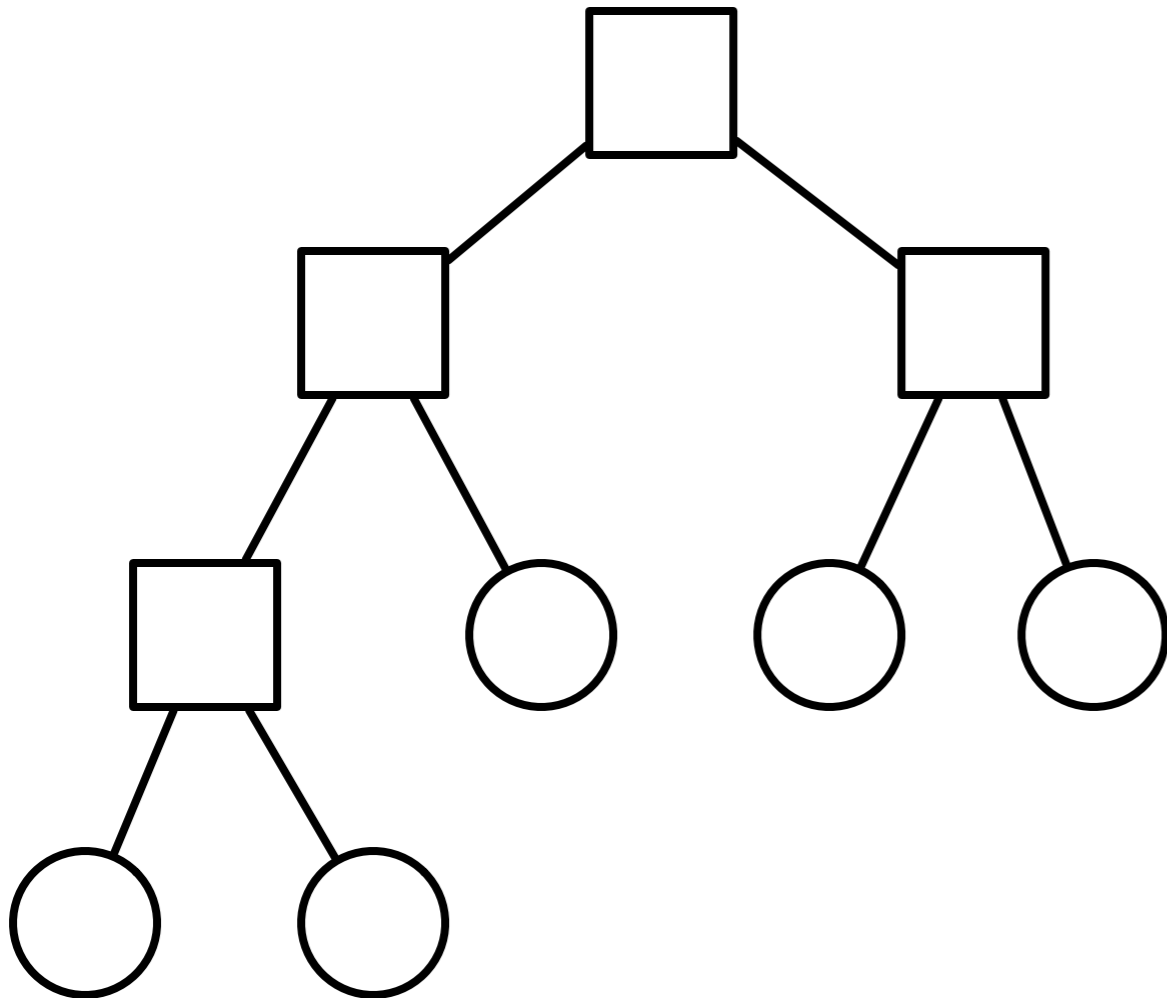
Visitor

Intent

Represent an operation to be performed on the elements of an 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

Operations are complex

PDF Print

Do different things on different types of nodes

TeX Print

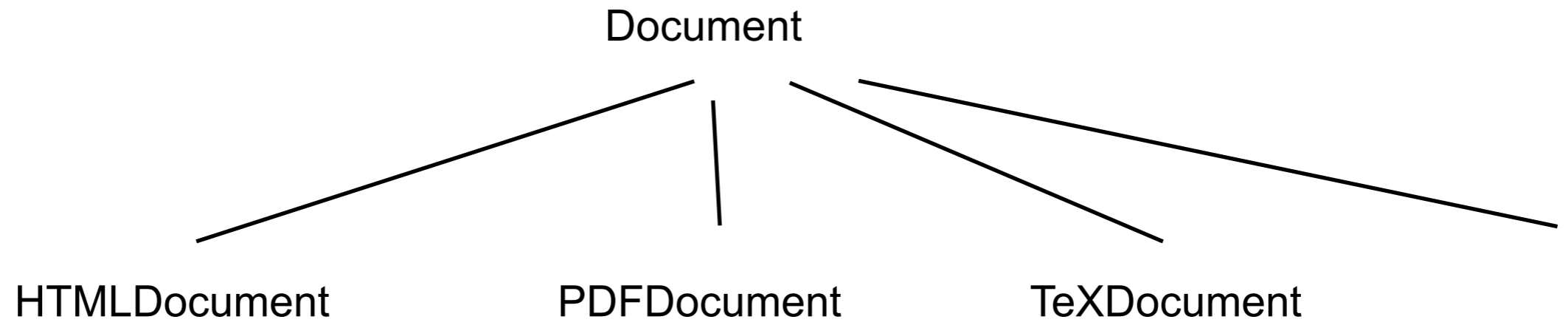
Need to traverse tree

RTF Print

Others likely in future

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);  
        }  
    }  
  
    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

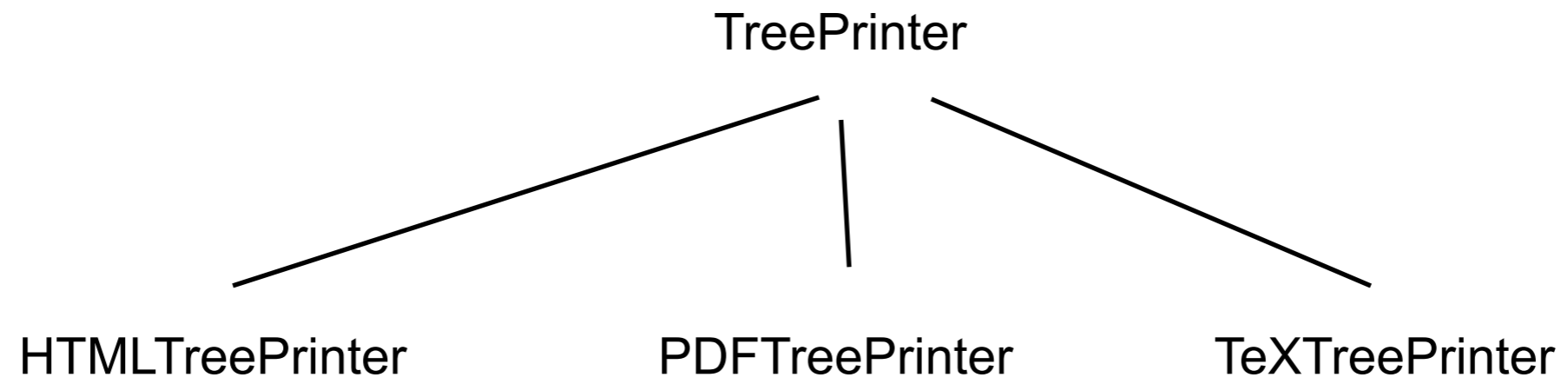
Which overloaded method to run

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

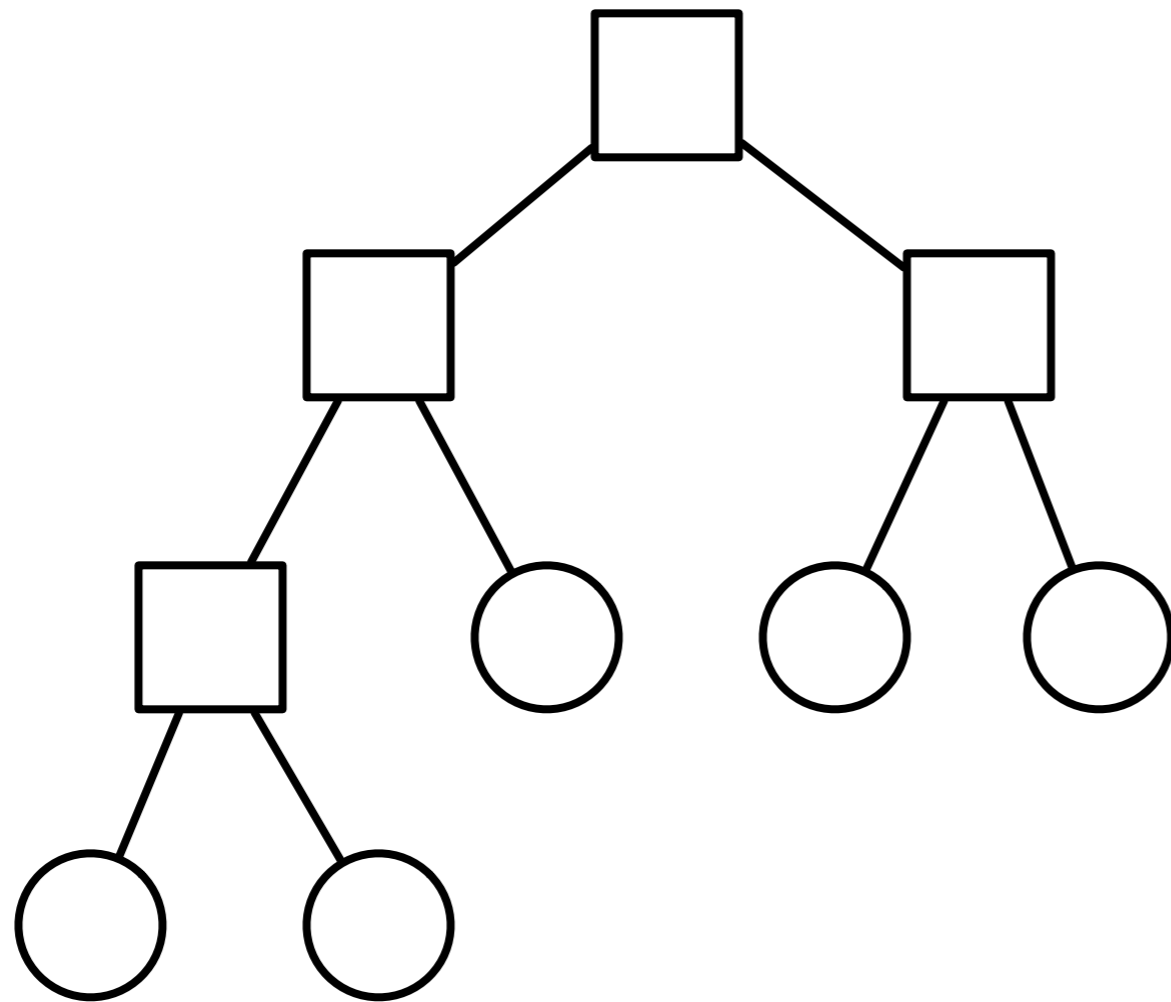
Pass the visitor to each element in the structure

The element then activates the visitor

Visitor performs its operation on the element

Each visitX method only deals with one type of element

Tree Example



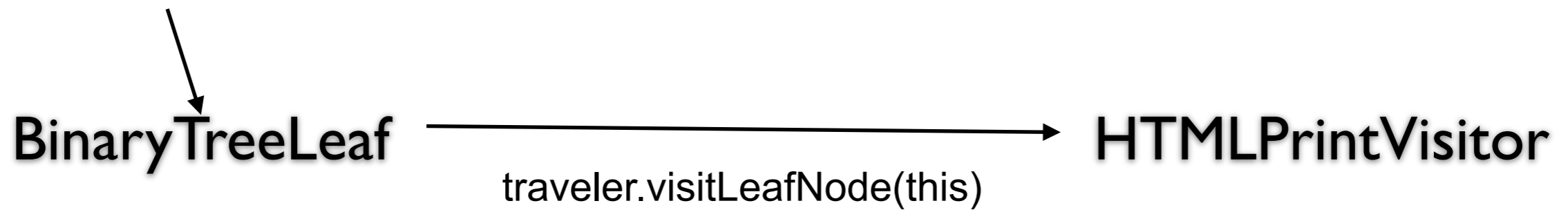
Visitor

Double Dispatch

Note that a visit to one node requires two method calls

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

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 performed 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 eliminates 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.
```

Edit Person

Title:

First Name:

Last Name:

Home Address:

Office Address:

Picture: no file selected

Birthday:

Age:

[Kind](#) [Number](#)

Phone Numbers: The report is empty.

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 & Julia

Multiple dispatch

At run-time

Based on argument types

No need for visitor pattern