# Visitor and Friends

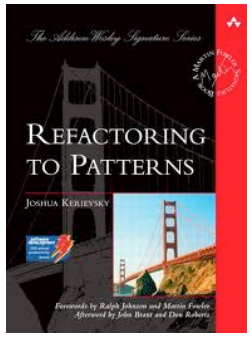
Alexandre Bergel
abergel@dcc.uchile.cl
29/12/2011


Design Patterns: Elements of Reusable Object-Oriented Software

*Erich Gamma, Richard Helm, Ralph Johnson, John M. Vlissides, 1994*

Refactoring to Patterns

*Joshua Kerievsky, Addison Wesley, 2004*



# Objective

The objective is this lecture is double

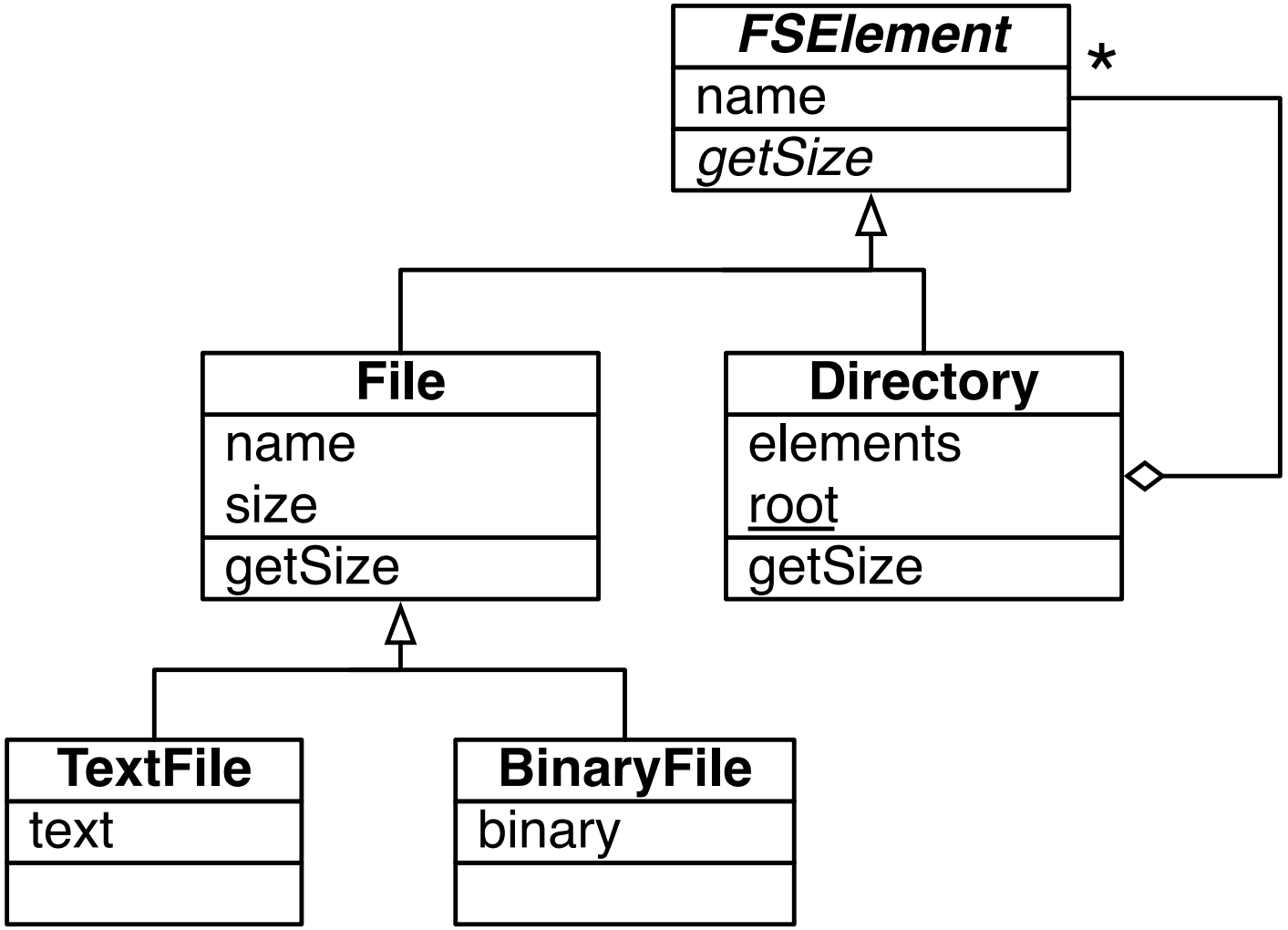
- face the problem addressed by the visitor pattern
- introduce the visitor pattern, which is a spectacular illustration of a proper separation of concern

# Exercise

A "file system" es un componente esencial de mucho sistemas operativos. Por este ejercicio, vamos a considerar los elementos siguientes:

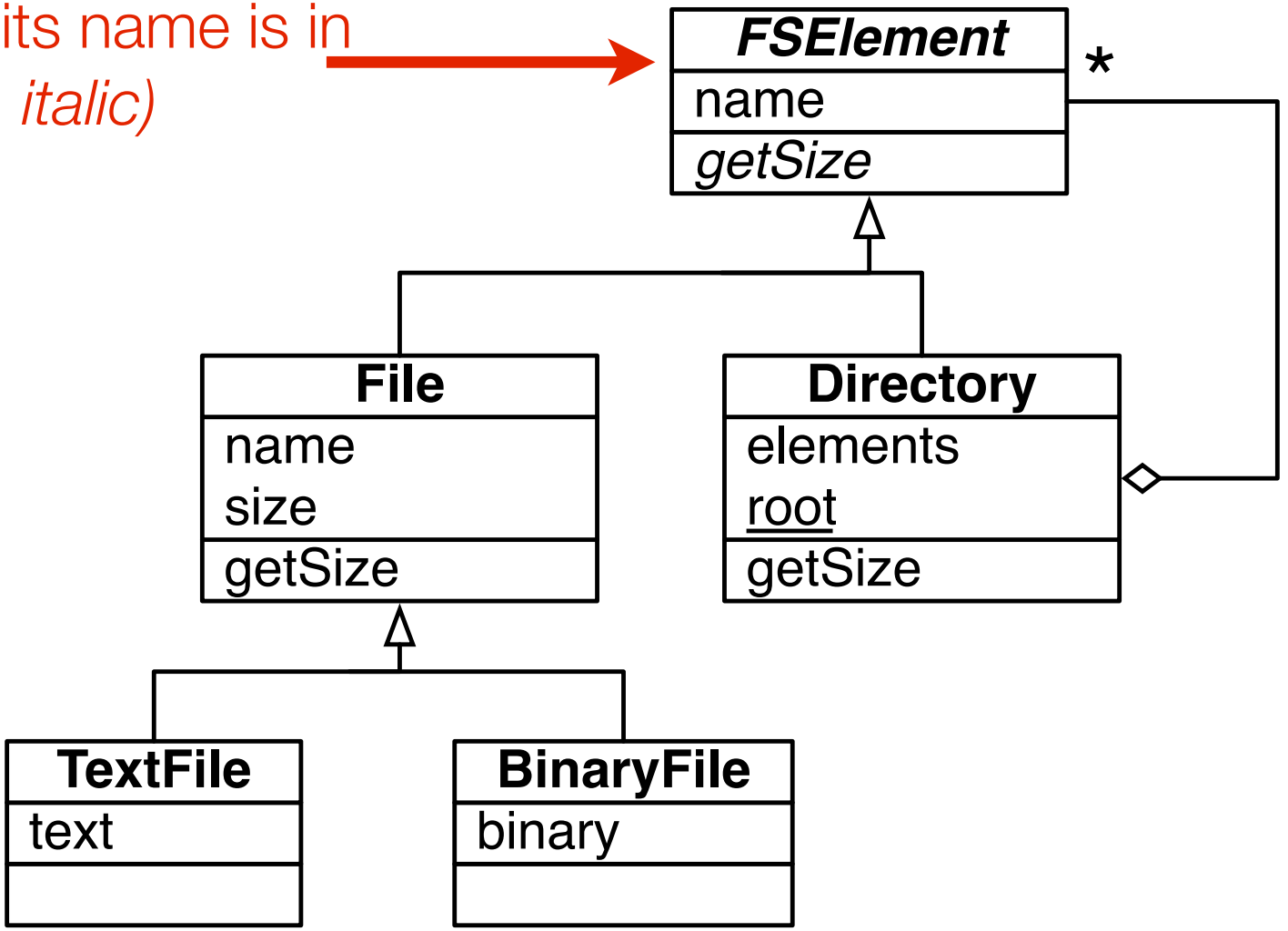
- un file system tiene files y directories
- un file puede ser un textual file o un binary file
- un file system tiene solamente un directory root
- un directory puede contenir textual files, binary files y directories
- cada elemento de un filesystem tiene un tamaño y un nombre

# A solution

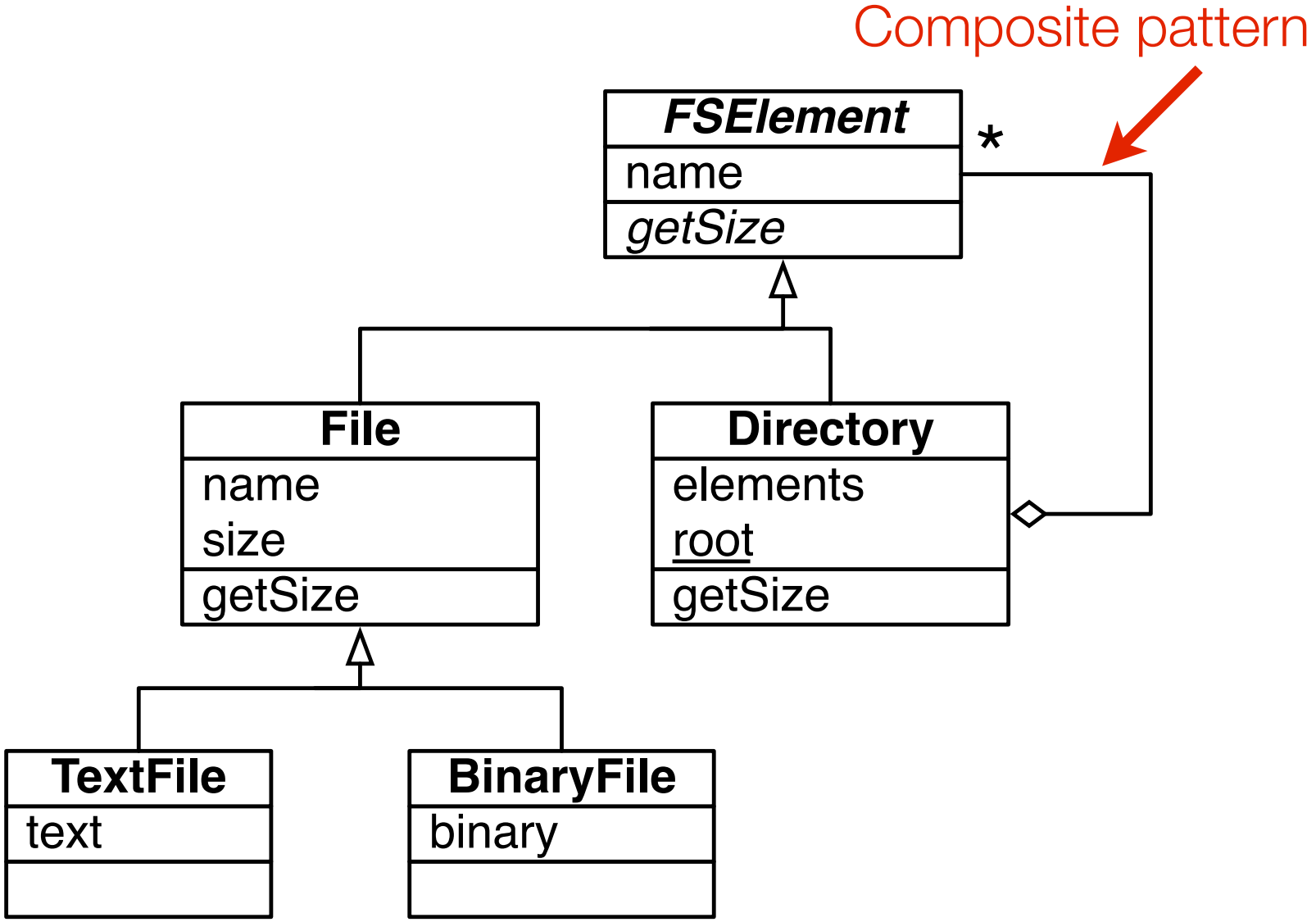

FSElement
name
getSize
*
File
name
size
getSize
Directory
elements
root
getSize
TextFile
text
BinaryFile
binary

# A solution

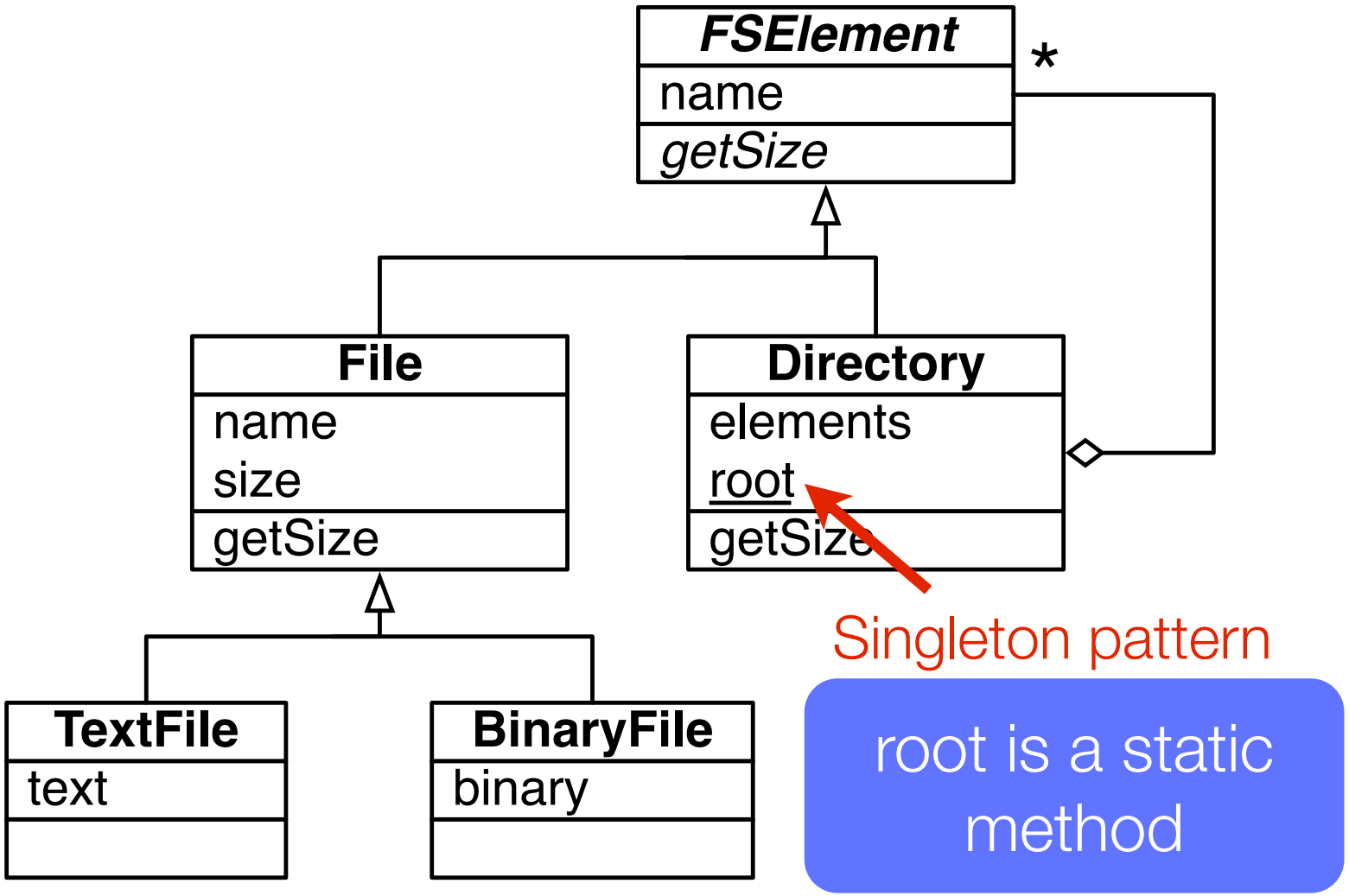
Abstract class (since its name is in *italic*)



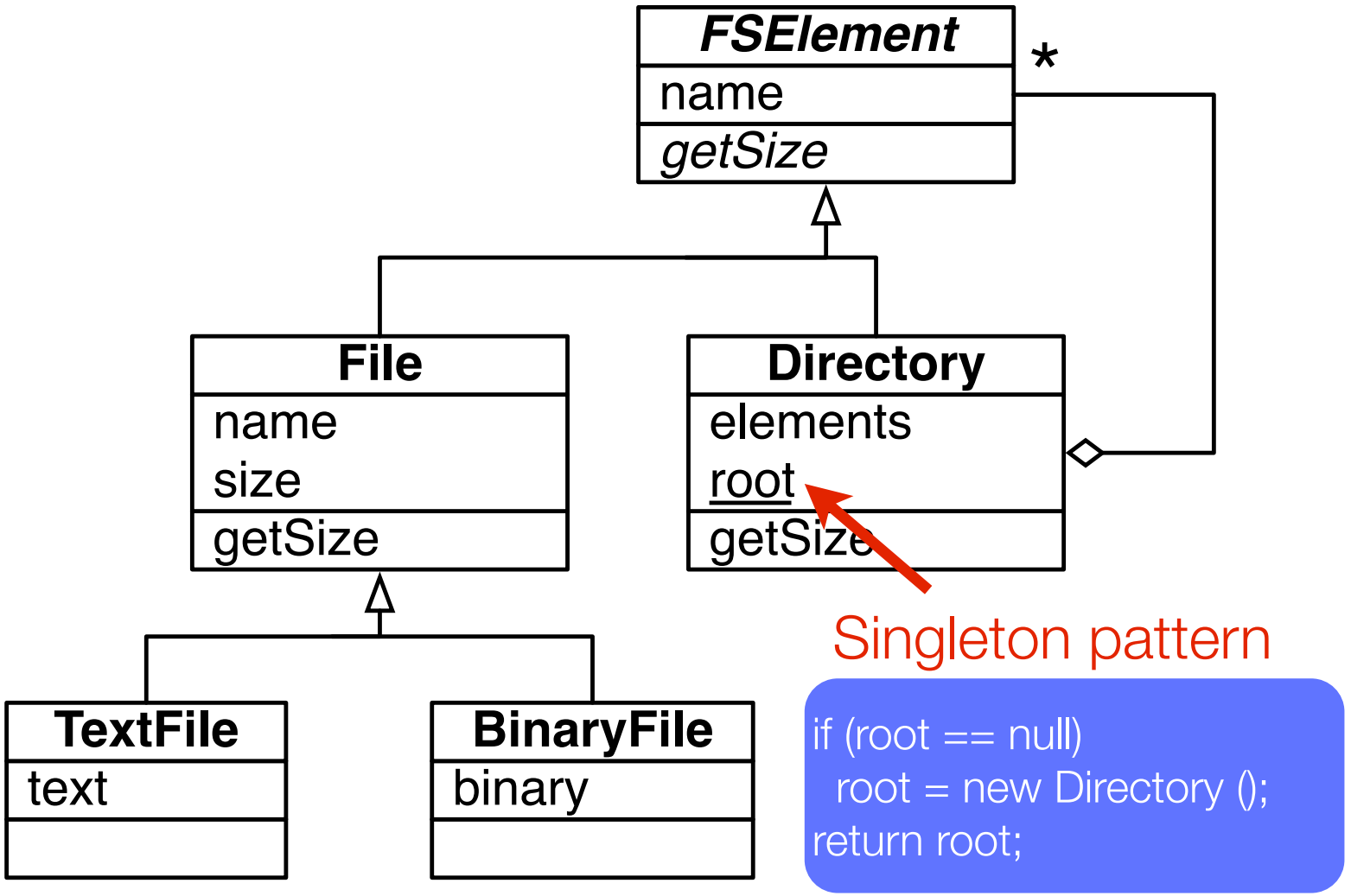
# A solution


Composite pattern
FSElement
name
getSize
*
File
name
size
getSize
Directory
elements
root
getSize
TextFile
text
BinaryFile
binary

# A solution


FSElement
name
getSize
*
File
name
size
getSize
Directory
elements
root
getSize
TextFile
text
BinaryFile
binary
Singleton pattern
root is a static method

# A solution


FSElement
name
getSize
*
File
name
size
getSize
Directory
elements
root
getSize
TextFile
text
BinaryFile
binary
Singleton pattern
if (root == null)
root = new Directory ();
return root;

# Size method

A version of getSize() can be

```
class File extends FSElement {
    private int size;
    public int getSize () { return size; }
    ...
}
```

# Size method

For the class Directory:

```
class Directory extends FSElement {
   private List<FSElement> elements = new List<FSElement>();
    public int getSize () {
        int size = 0;
        for (Element el : elements) size += el.getSize();
        return size;
    }
    ...
}
```

# Exercise...

Now, we would like to add some operations

- get the total number of files contained in a directory
- delete a particular element, which may be deeply nested
- do a recursive listing
- ...

# Important questions

How to write the *invocation* of such *operation*?

Do I need a *class hierarchy* for the different recursive operations?

What is the cost of adding a *new operation*?

Is there any *code duplication*?

# Adding operations

Implementing these operations has a high cost

- the domain (file and directory) has to be *modified* at each *operation*
- can be cumbersome if the domain is *externally provided*
- each of these operations contains *duplication*, notably the recursion over the structure

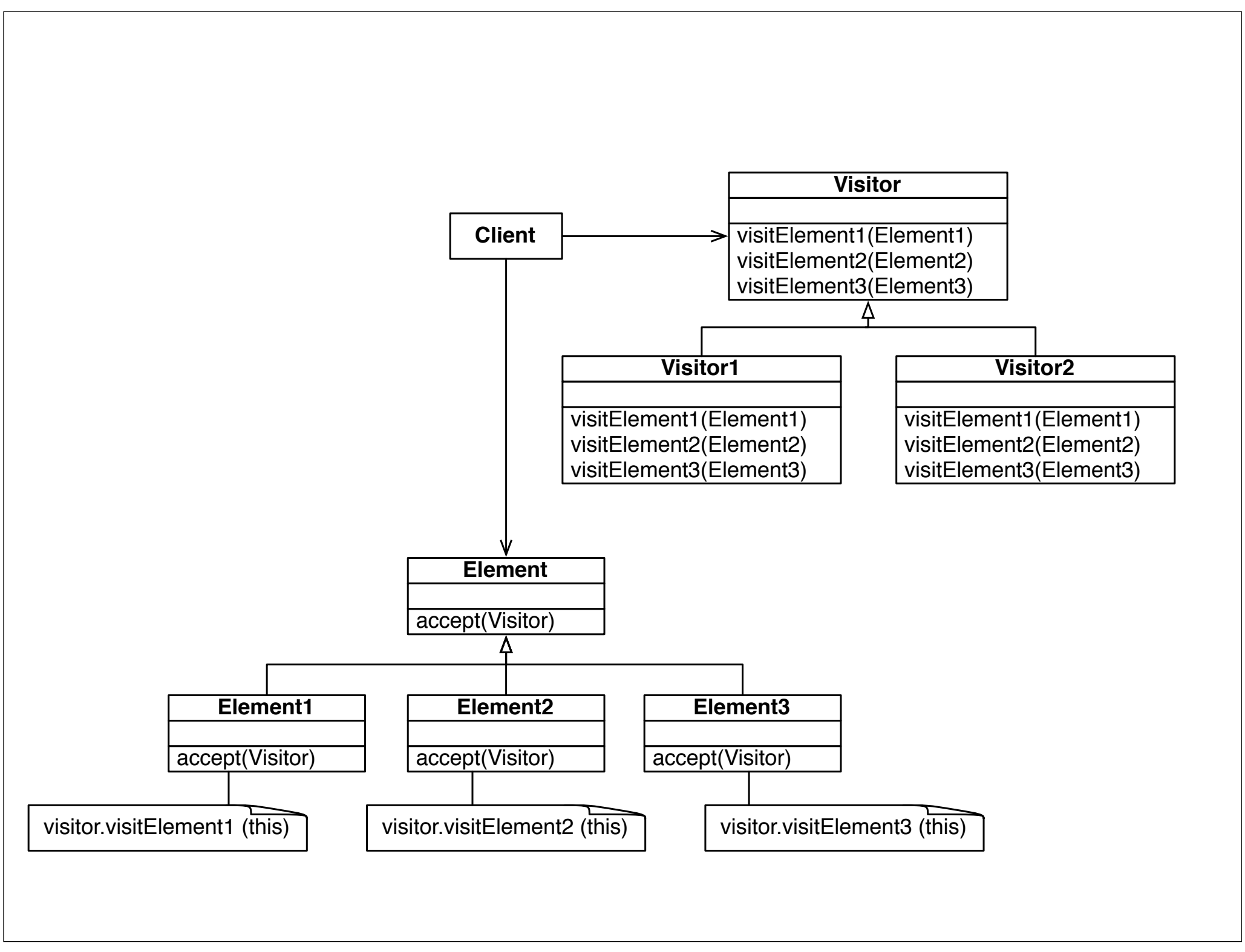
# Visitor Pattern

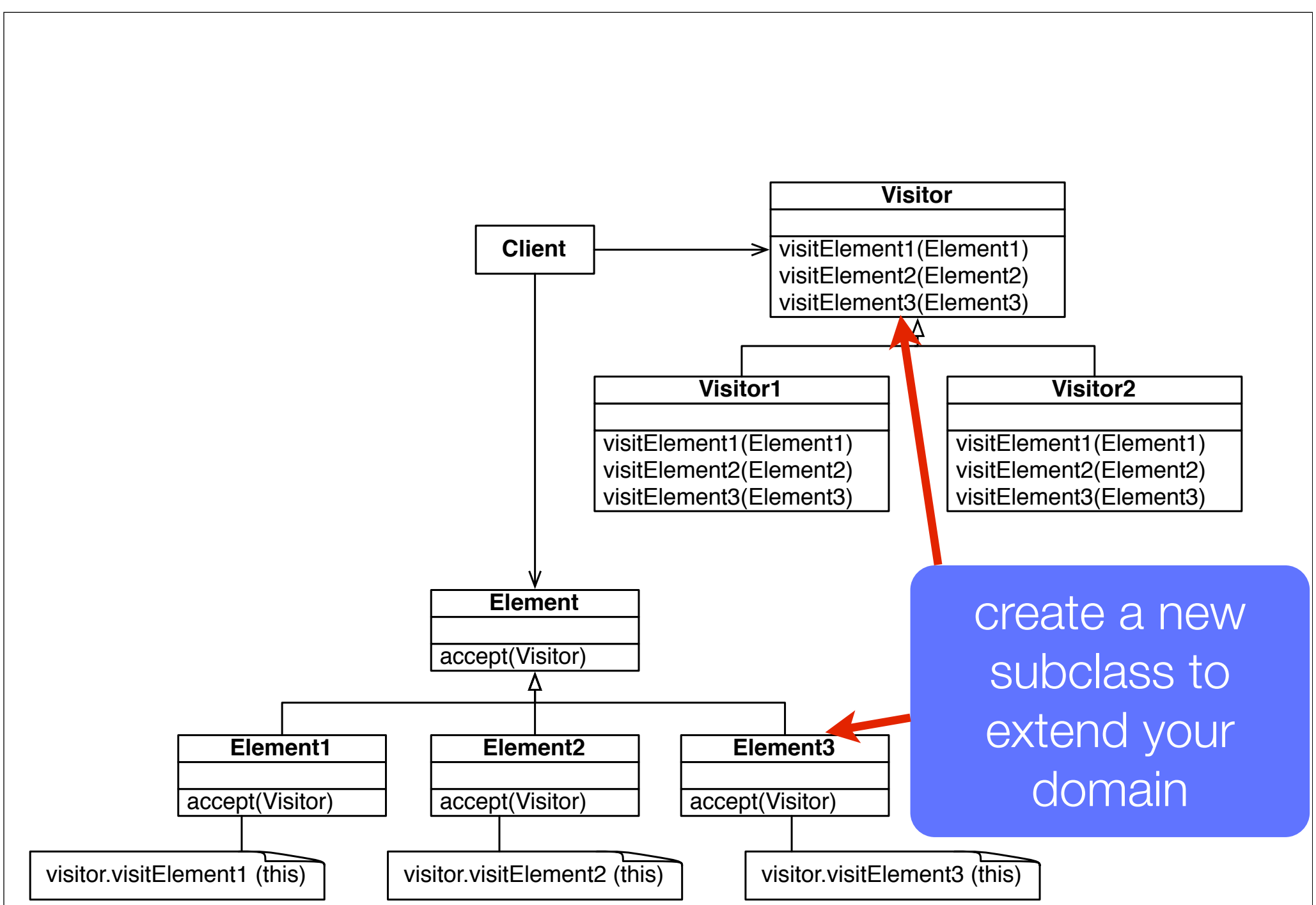
How do you accumulate information from heterogeneous classes?

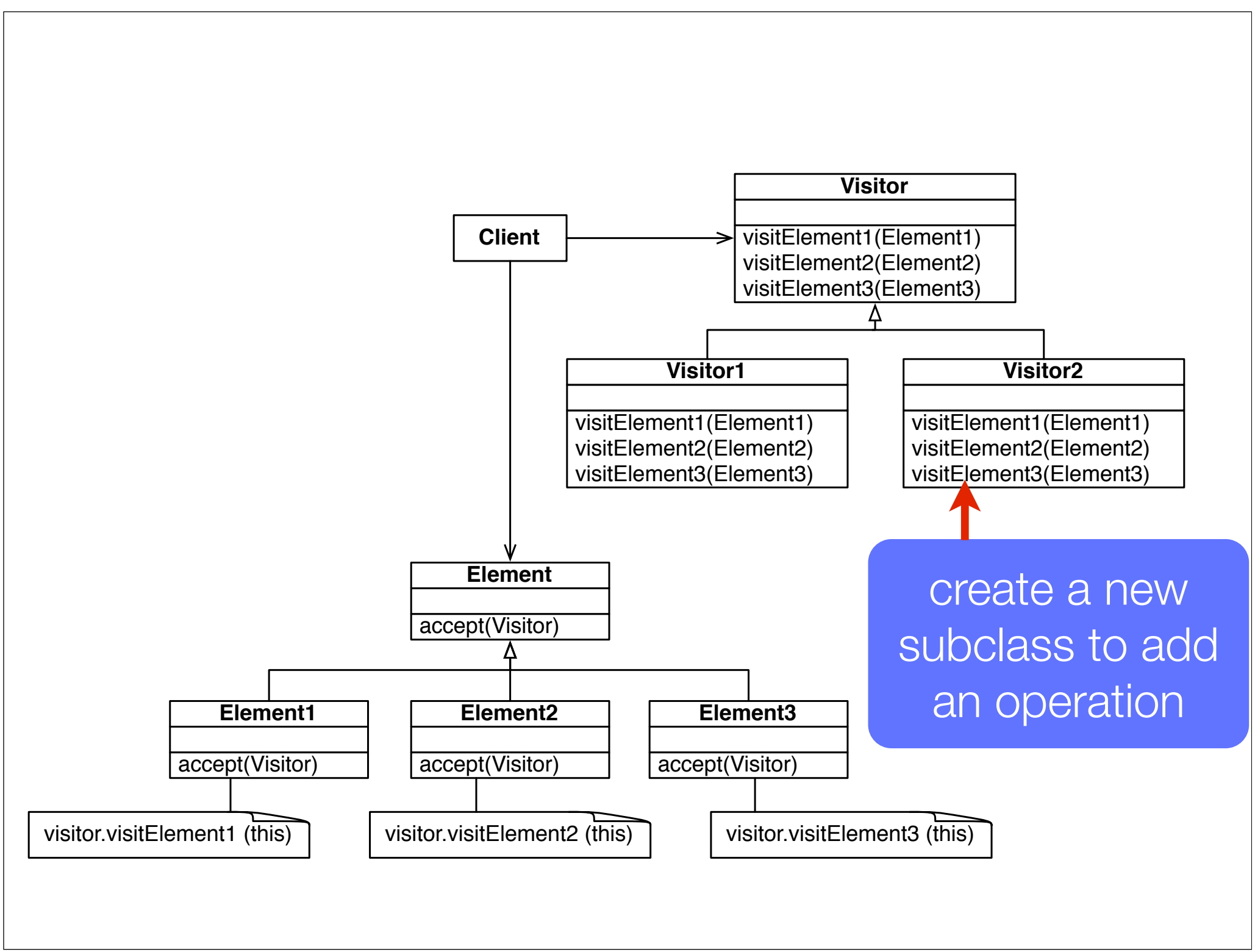
> Move the accumulation task to a Visitor that can visit each class to accumulate the information

A visitor is a class that performs an operation on an object structure. The classes that a Visitor visits are heterogenous.

> Intensively use double dispatch


Visitor
visitElement1(Element1)
visitElement2(Element2)
visitElement3(Element3)
Client
Visitor1
visitElement1(Element1)
visitElement2(Element2)
visitElement3(Element3)
Visitor2
visitElement1(Element1)
visitElement2(Element2)
visitElement3(Element3)
Element
accept(Visitor)
Element1
accept(Visitor)
Element2
accept(Visitor)
Element3
accept(Visitor)
visitor.visitElement1 (this)
visitor.visitElement2 (this)
visitor.visitElement3 (this)


Visitor
visitElement1(Element1)
visitElement2(Element2)
visitElement3(Element3)
Client
Visitor1
visitElement1(Element1)
visitElement2(Element2)
visitElement3(Element3)
Visitor2
visitElement1(Element1)
visitElement2(Element2)
visitElement3(Element3)
Element
accept(Visitor)
create a new subclass to extend your domain
Element1
accept(Visitor)
Element2
accept(Visitor)
Element3
accept(Visitor)
visitor.visitElement1 (this)
visitor.visitElement2 (this)
visitor.visitElement3 (this)


Visitor
visitElement1(Element1)
visitElement2(Element2)
visitElement3(Element3)
Client
Visitor1
visitElement1(Element1)
visitElement2(Element2)
visitElement3(Element3)
Visitor2
visitElement1(Element1)
visitElement2(Element2)
visitElement3(Element3)
create a new subclass to add an operation
Element
accept(Visitor)
Element1
accept(Visitor)
Element2
accept(Visitor)
Element3
accept(Visitor)
visitor.visitElement1 (this)
visitor.visitElement2 (this)
visitor.visitElement3 (this)

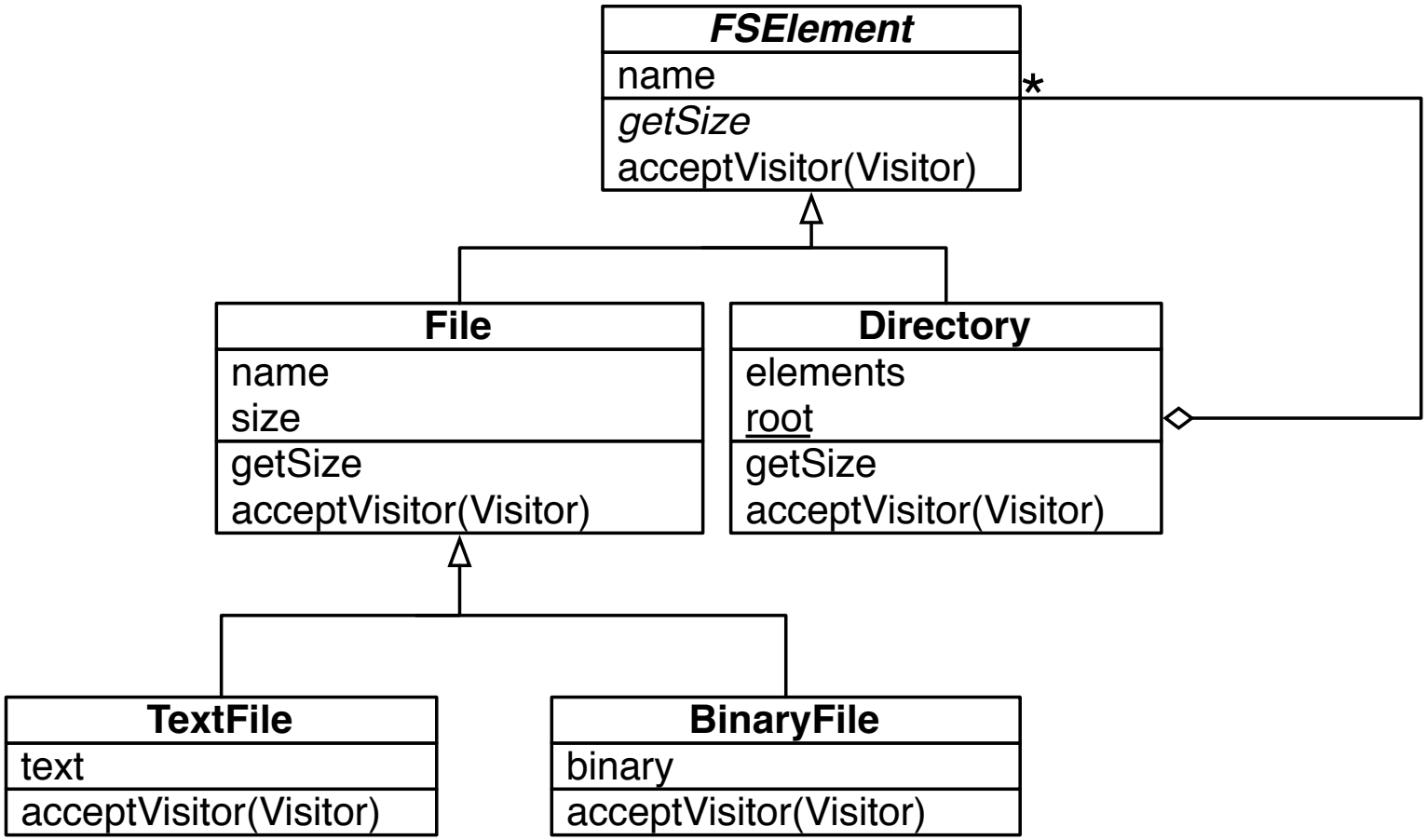
# Adding operations

The visitor pattern is a nice solution to *add new operations*

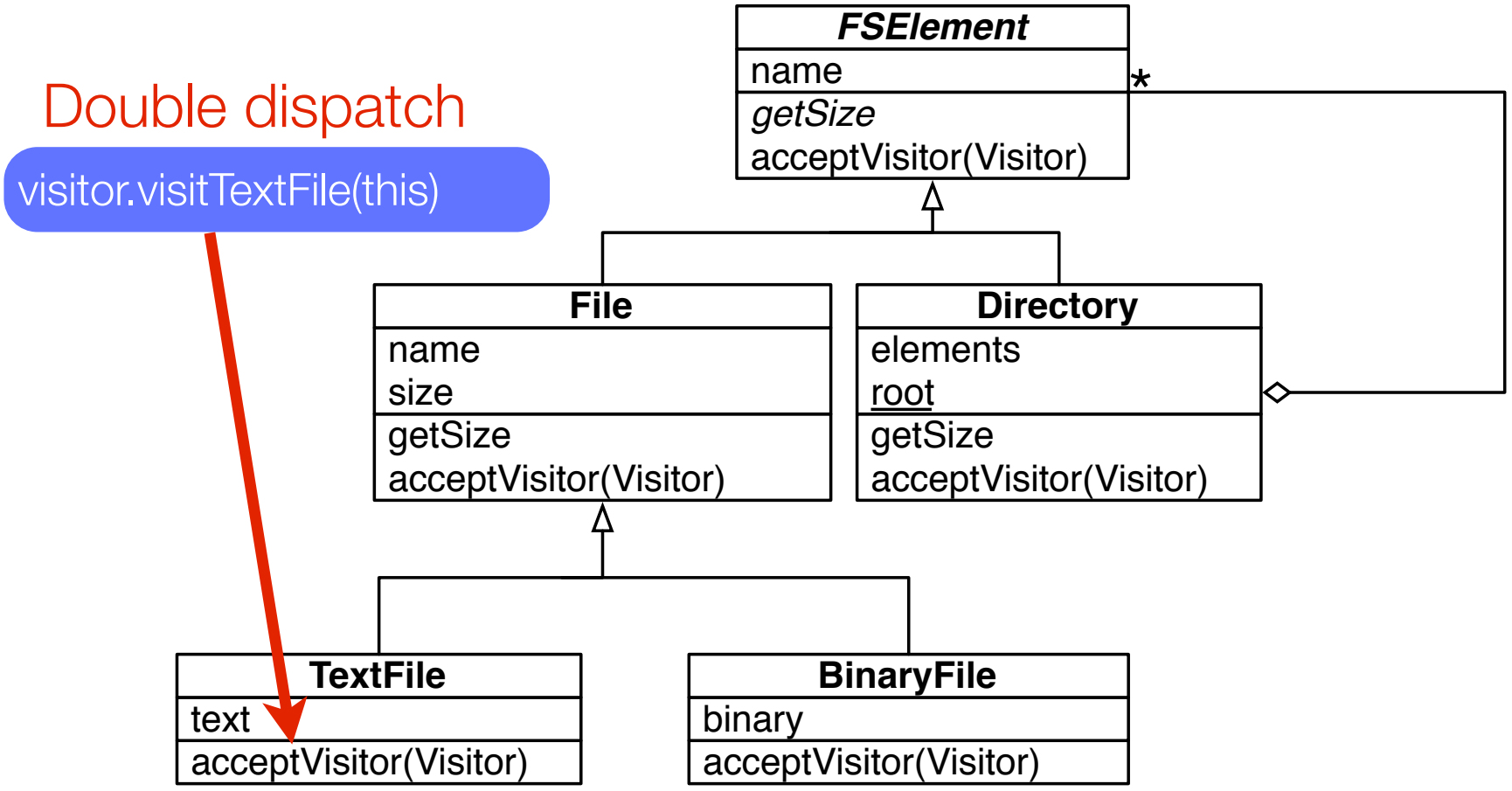
Operations are defined *externally* from the domain, by subclassing *Visitor*

The drawback is that it usually *enforces* the *state* of the objects to be *accessible* from *outside*

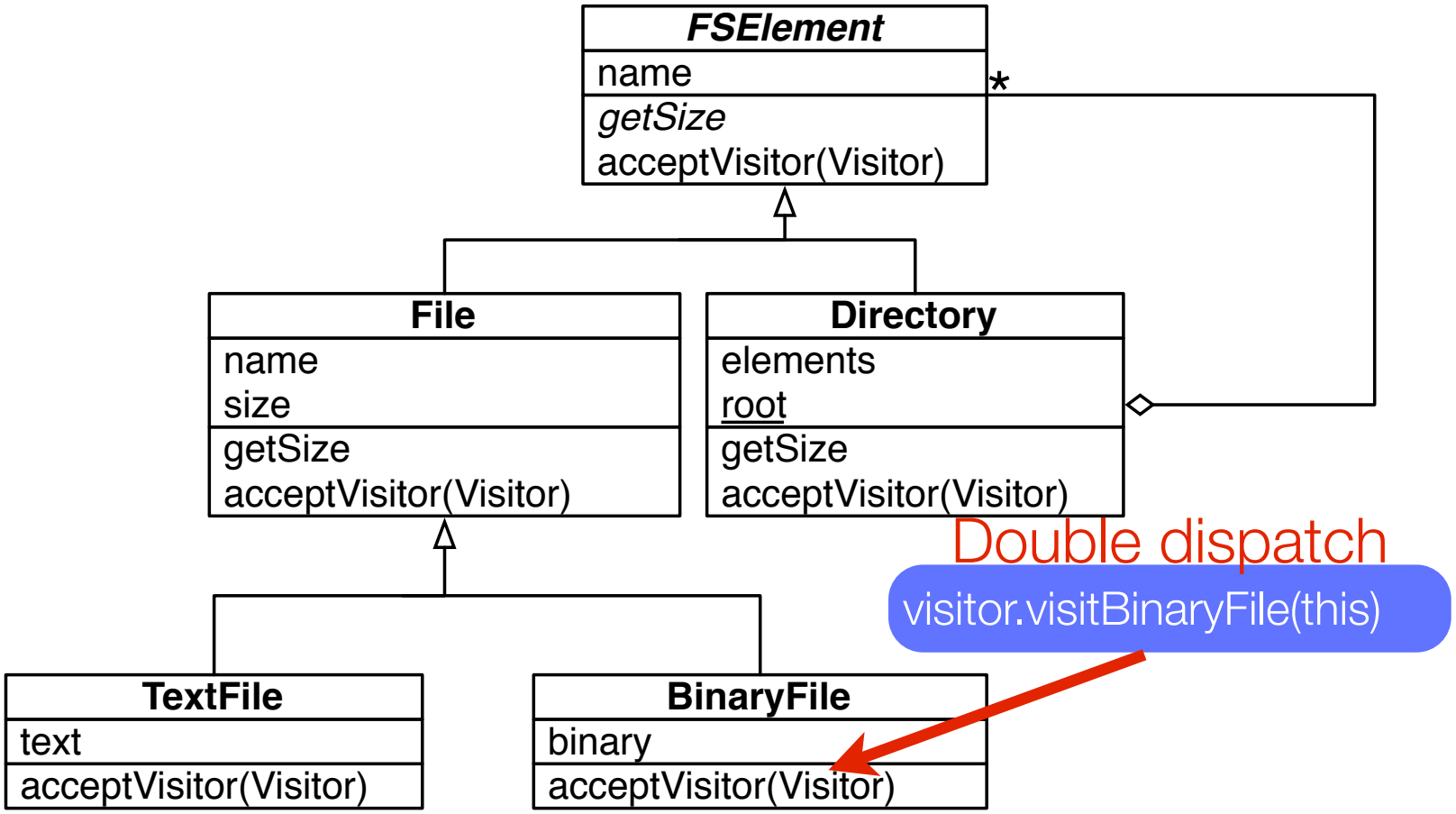
# New version of our file system



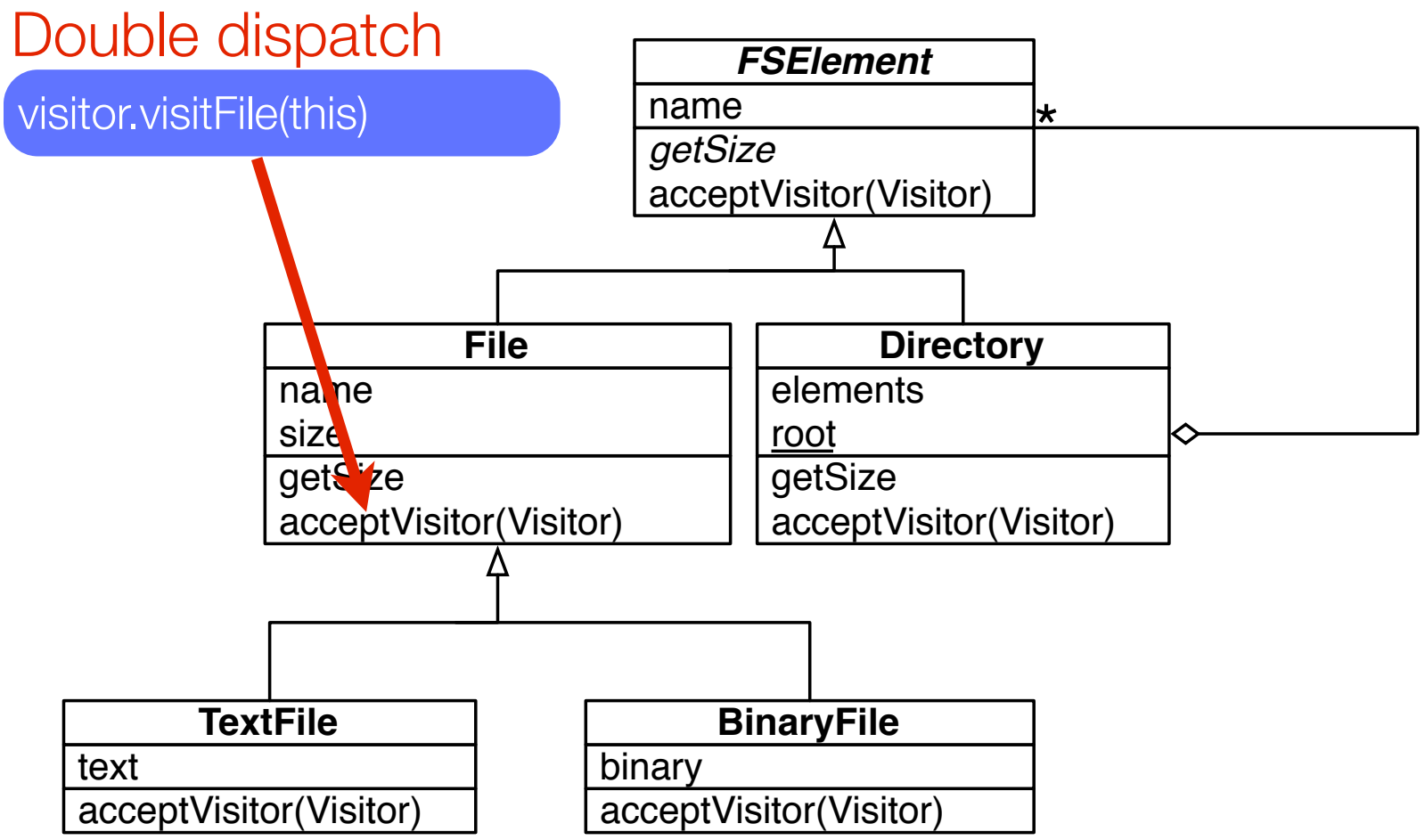
# New version of our file system



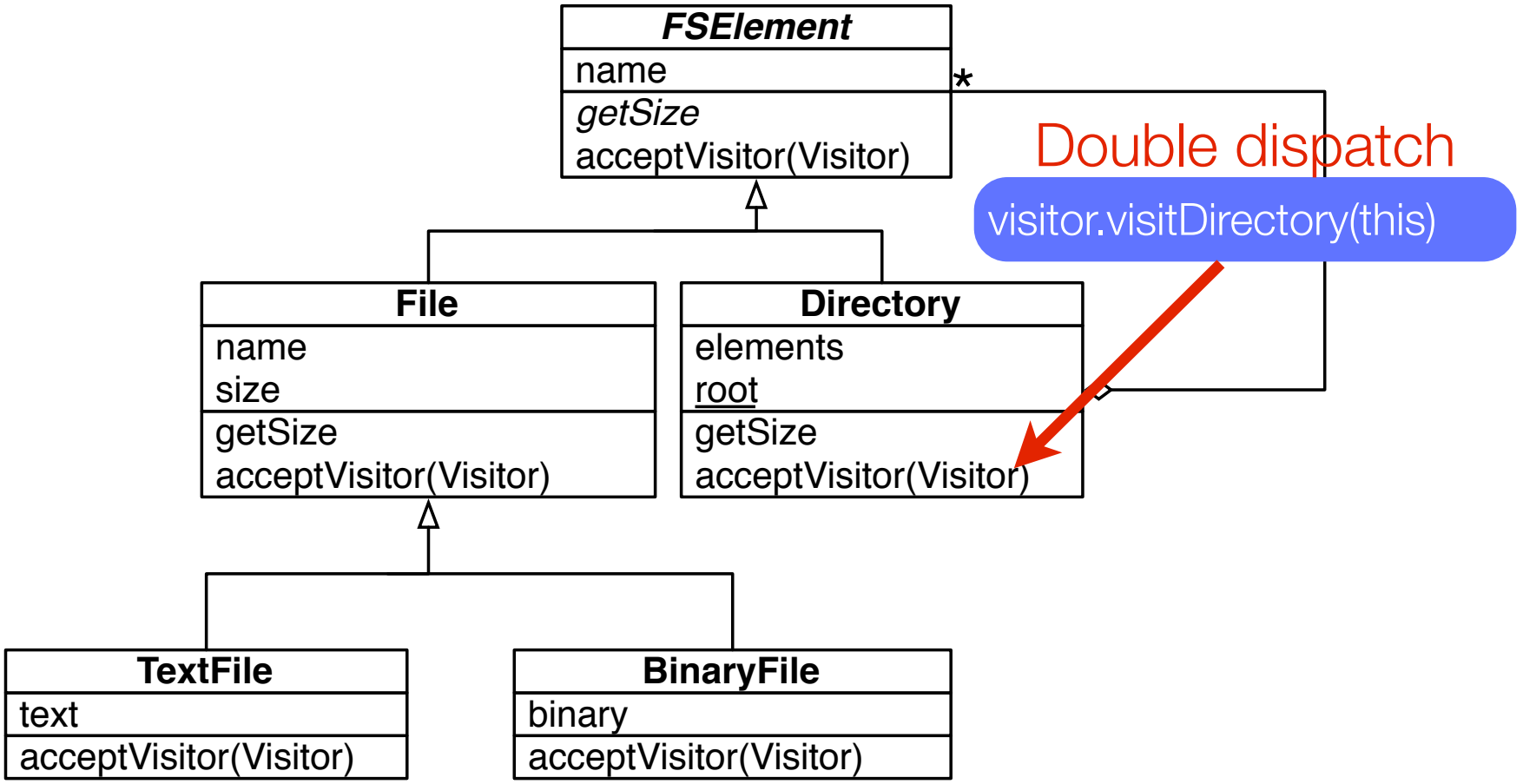
# New version of our file system

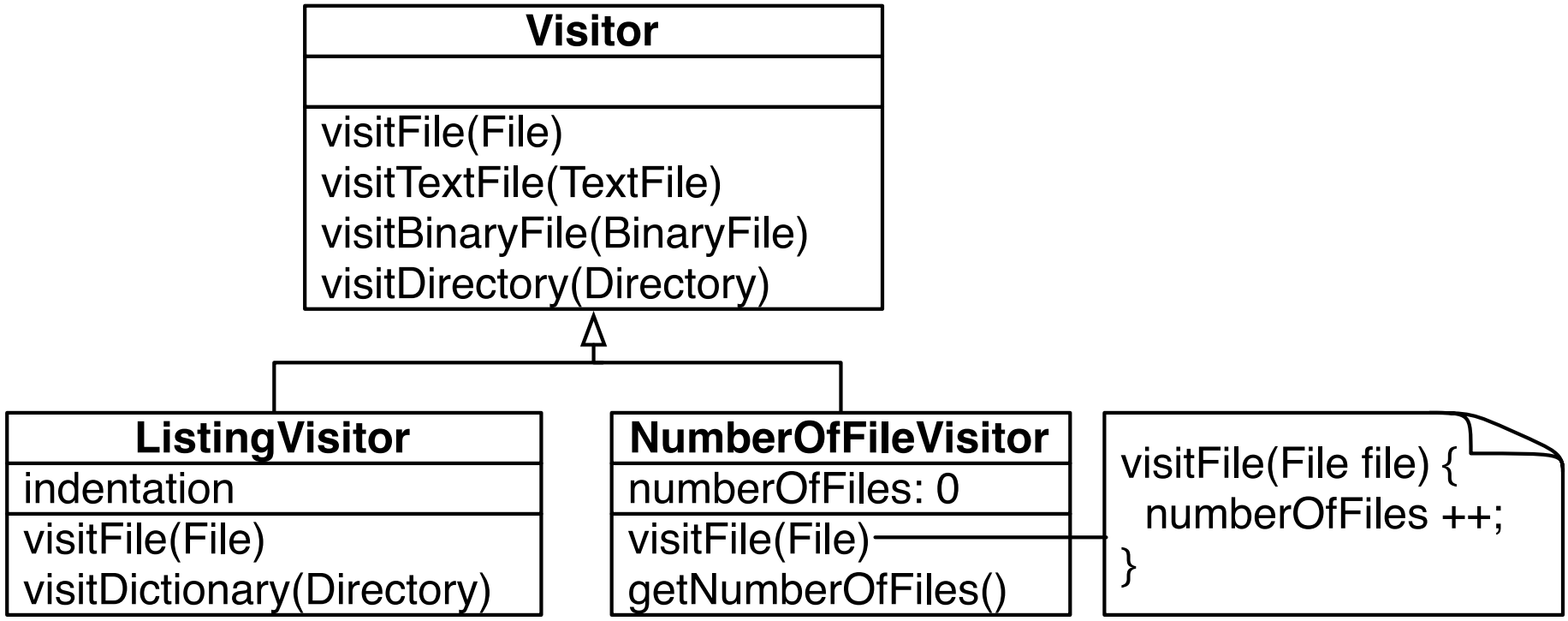


# New version of our file system



# New version of our file system




Visitor
visitFile(File)
visitTextFile(TextFile)
visitBinaryFile(BinaryFile)
visitDirectory(Directory)
ListingVisitor
indentation
visitFile(File)
visitDictionary(Directory)
NumberOfFileVisitor
numberOfFiles: 0
visitFile(File)
getNumberOfFiles()
visitFile(File file) {
numberOfFiles ++;
}

# The case for Directory

The recursion over the structure at runtime can be achieved in the Visitor

```
class Visitor {
    public void visitDirectory(Directory d) {
        for (Element e : d.getElements() )
            e.acceptVisitor(this);
    }
    public void visitFile(File d) { }
    ...
}
```

# Points worth to discuss

## Where to put the recursion?

In the class Directory? In the Visitor?

## Some solutions may favor code overloading

define “visit(Element1)” instead of “visitElement1(Element1)”

What is your opinion on this?

# Classification

## Creational patterns

abstract factory, builder, factory method, lazy initialization, singleton

## Structural patterns

adapter (wrapper), bridge, composite, decorator, façade, flyweight, proxy

## Behavioral patterns

command, interpreter, iterator, null object, observer, state, template method, visitor

# Classification

## Concurrency patterns

Active objects, balking, lock, monitor object, thread pool, scheduler, ...

There design are specific to the concurrency domain, and do not appear in the GoF book

# What you should know

When to use a visitor pattern?

What are the problems the visitor pattern solve?

What is the cost of adding new operations in a domain?

# Can you answer to these questions?

When can it disadvantageous to use the Visitor?

Variations found in the literature favor method overloading. What are the limitations? What are the dangers of it?

Is the visitor pattern always associated to a composite pattern?