The Visitor Pattern

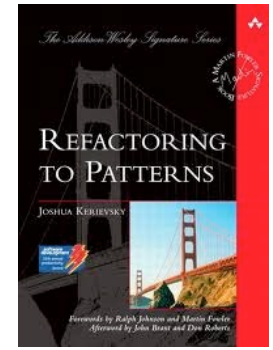
Nancy Hitschfeld
Matías Toro

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



Remember the Library example?

In the Library example, we have a library in which we can add items, including games, books, and journals

Each item has a name and a publish year

We need to formulate some queries to retrieve some items from the library:

- What are the items having a particular name? (e.g., “Starcraft”)
- What are the items published in a particular year? (e.g., 2015)

Remember the Library example?

We have seen that a naive implementation of the queries suffer from the following problems:

- Most of queries look like the same since most of their source codes are duplicated
- Adding a new query has a high cost since all the domain classes have to be modified

Remember the Library example?

We have seen that using (a simplified version of) the template we can easily

- Reduce the duplication of code
- Reduce the cost of adding a new operation

However, there are some queries that cannot be formulated. E.g.,

- All the games named “Starcraft”
- All the books published in 1985
- This is exactly what we will discuss about today

Objective of today

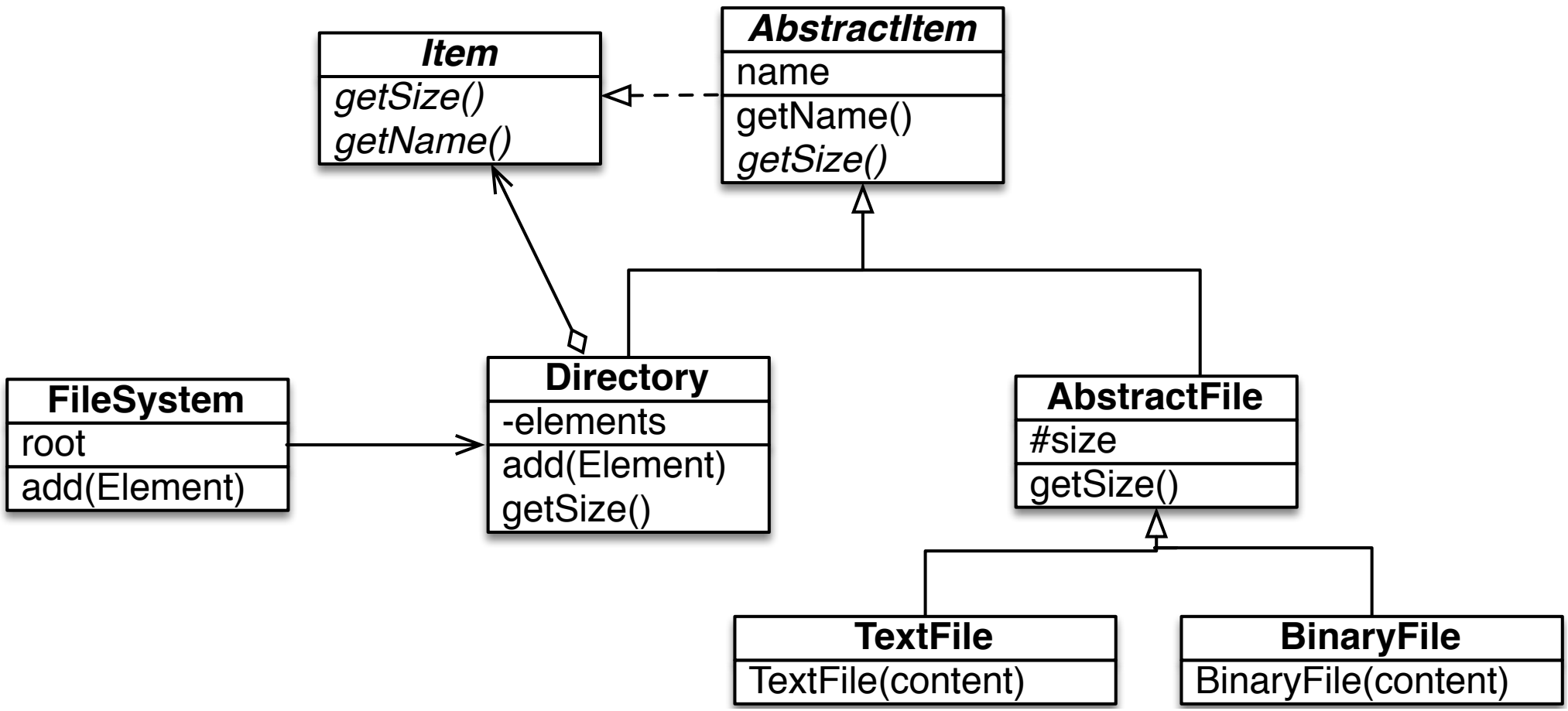
The objective is this lecture is double

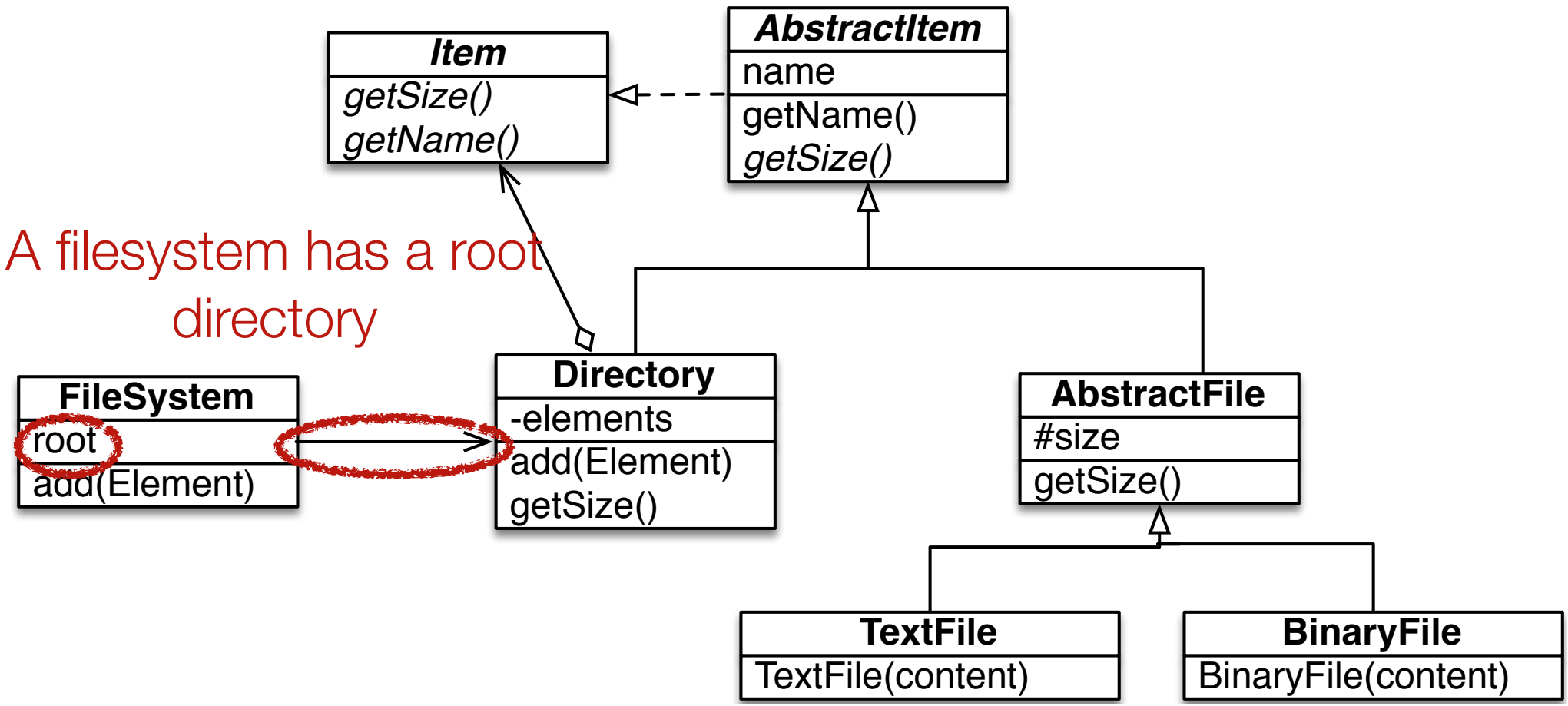
- Face the problem addressed by the visitor pattern
- Introduce the visitor pattern, which is a spectacular illustration of a proper separation of concern

Exercise

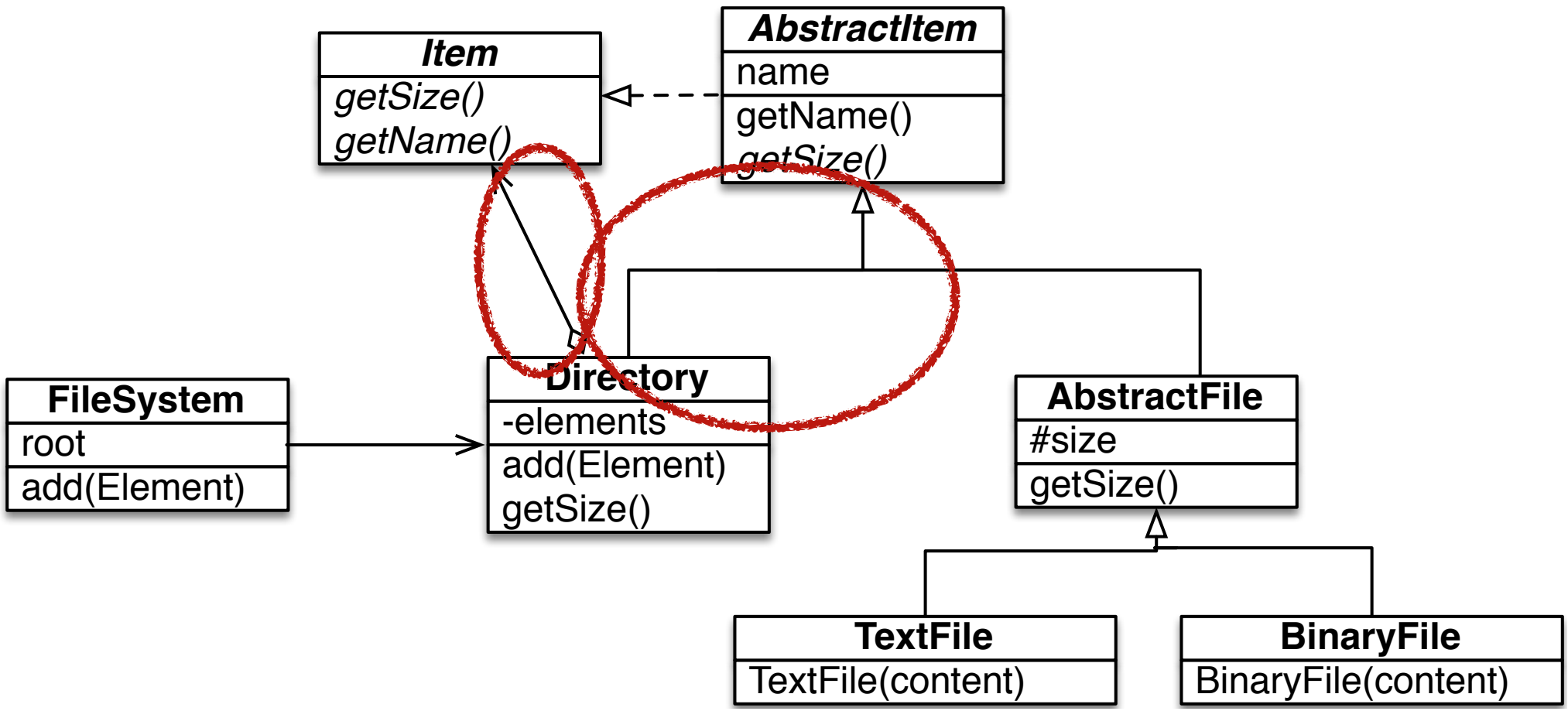
Un "file system" es un componente esencial de mucho sistemas operativos. Por este ejercicio, vamos a considerar los elementos siguientes:

- 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 contener textual files, binary files y directories
- cada elemento de un filesystem tiene un tamaño y un nombre


Item
getSize()
getName()
AbstractItem
name
getName()
getSize()
FileSystem
root
add(Element)
Directory
-elements
add(Element)
getSize()
AbstractFile
#size
getSize()
TextFile
TextFile(content)
BinaryFile
BinaryFile(content)

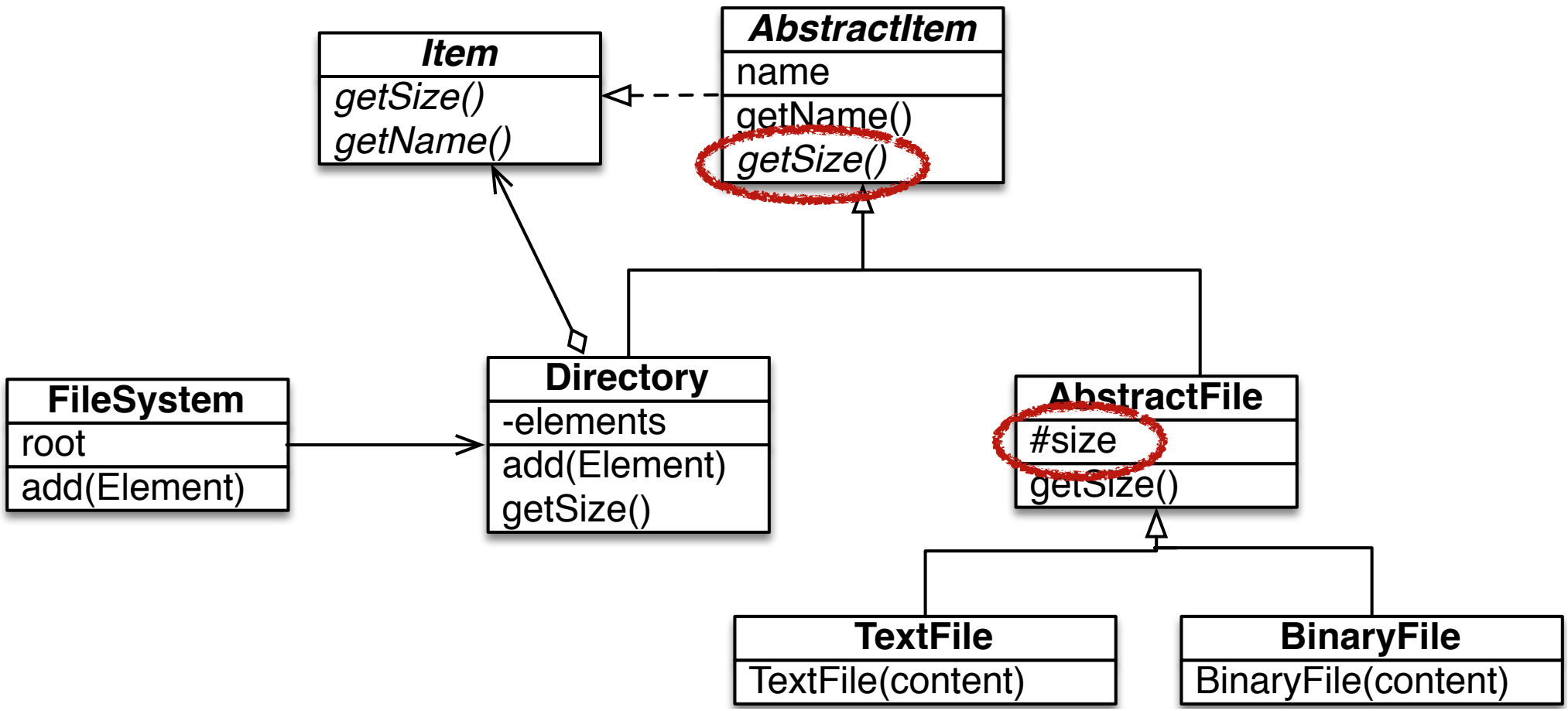

Item
getSize()
getName()
AbstractItem
name
getName()
getSize()
A filesystem has a root directory
FileSystem
root
add(Element)
Directory
-elements
add(Element)
getSize()
AbstractFile
#size
getSize()
TextFile
TextFile(content)
BinaryFile
BinaryFile(content)

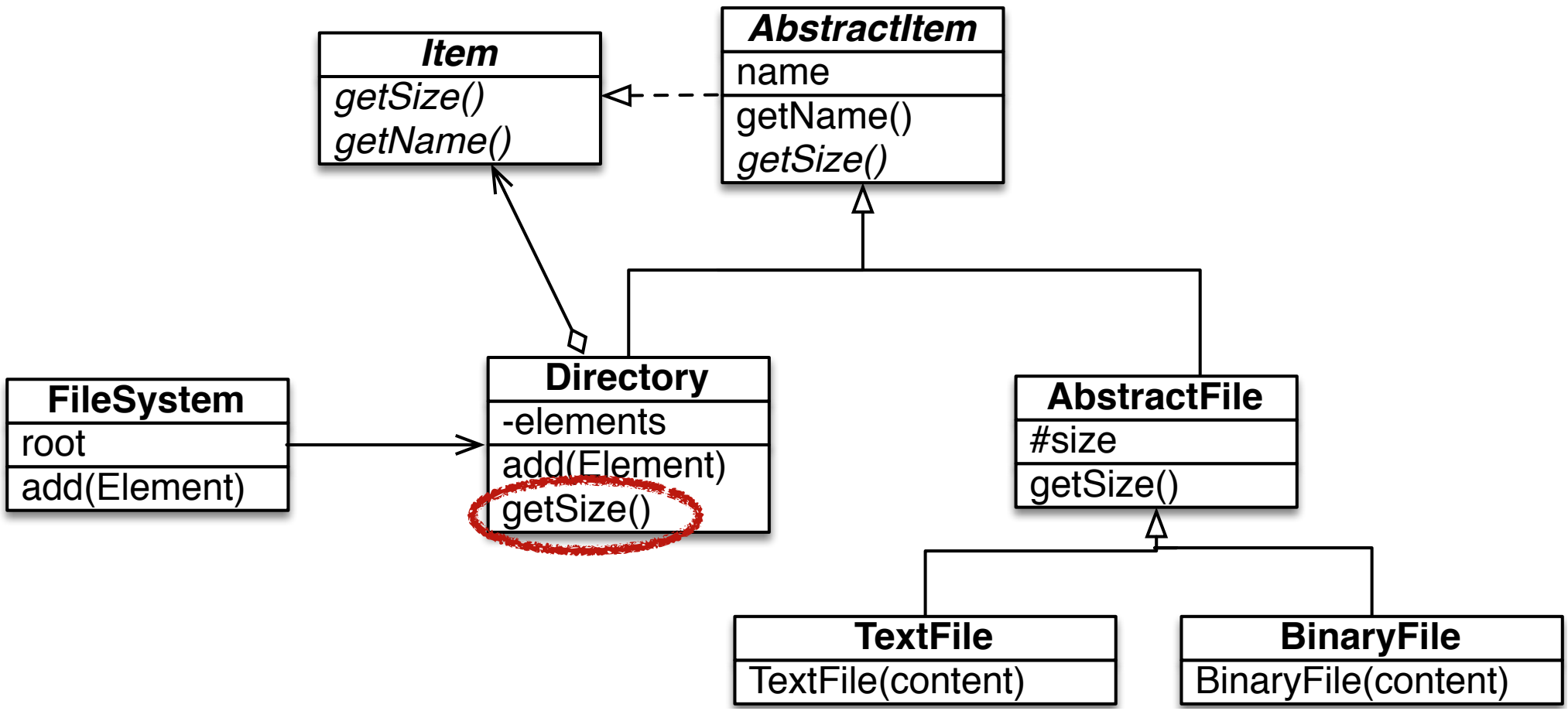
The composite patterns make a directory to contains items



Note that we have a small variant from the original definition of the composite pattern, but it is similar

There is no need to have a variable size in AbstractItem
Since the value is computed in Directory
Instead, we have the variable in AbstractFile





The getSize() method is recursive to compute the size

```
class FileSystemTest extends munit.FunSuite {
  private var emptyFs: FileSystem = null
  private var fs: FileSystem = null
  private var d1: Directory = null
  private var d2: Directory = null

  override def beforeEach(context: BeforeEach): Unit = {
    emptyFs = new FileSystem()
    d1 = new Directory("d1")
    d2 = new Directory("d2")
    d1.add(d2)
    d1.add(new TextFile("file.txt", "hello world!"))
    val c = Array[Byte]('1', 'c')
    d1.add(new BinaryFile("file.txt", c))
    fs = new FileSystem()
    fs.add(d1)
  }

  test("testGetSize"){
    assertEquals(0, emptyFs.getSize())
    assertEquals(14, fs.getSize())
  }
}
```

```
class FileSystem() {
  private val root = new Directory("root")

  def add(item: Item): Unit = {
    root.add(item)
  }

  def getSize(): Int = root.getSize()
}
```

```
trait Item {
  def getSize(): Int
  def getName(): String
}
```

```
abstract class AbstractItem(private var name: String)
             extends Item {
  def getName(): String = name
}
```

```
import scala.collection.mutable.ListBuffer

class Directory(name: String) extends
AbstractItem(name) {

  private val items = new ListBuffer[Item]()

  def add(anItem: Item): Unit = {
    items += anItem
  }

  override def getSize(): Int = {
    var result = 0
    for (item <- items) {
      result += item.getSize
    }
    result
  }
}
```

```
abstract class AbstractFile(name: String)
              extends AbstractItem(name) {
  protected var size: Int
  def getSize(): Int = size
}
```

```
class TextFile(val aName: String, val content: String)
                            extends AbstractFile(aName) {
  protected var size = content.length
}
```

```
class BinaryFile(val aName: String, val content: Array[Byte])
                                  extends AbstractFile(aName) {
  protected var size = content.length
}
```

Exercise...

Now, we would like to add some operations

- get the total number of files contained in a file system
- get the total number of directories contained in a file system
- do a recursive listing
- ...

```
class FileSystemTest extends munit.FunSuite {
  …

  test("testGetNumberOfFile"){
    assertEquals(0, emptyFs.getNumberOfFiles())
    assertEquals(2, fs.getNumberOfFiles())
    val aFile = new TextFile("tmp.txt", "a file system example")
    val d = new Directory("another directory")
    d.add(aFile)
    fs.add(d)
    assertEquals(3, fs.getNumberOfFiles())
  }

  test("testGetNumberOfDirectory"){
    assertEquals(1, emptyFs.getNumberOfDirectory())
    assertEquals(3, fs.getNumberOfDirectory())
    val aFile = new TextFile("tmp.txt", "a file system example")
    val d = new Directory("another directory")
    d.add(aFile)
    fs.add(d)
    assertEquals(4, fs.getNumberOfDirectory())
  }

  test("testListing"){
    var result = "root\n"
    assertEquals(result, emptyFs.listing())
    result = "root\nd1\nd2\nfile.txt\nfile.obj\n"
    print(fs.listing())
    assertEquals(result, fs.listing())
  }
}
```

```
trait Item {
  def getSize(): Int
  def getName(): String

  def getNumberOfFiles(): Int
  def getNumberOfDirectory(): Int
  def listing(): String
}
```

```
class Directory(name: String) extends AbstractItem(name) {
  …
  override def getNumberOfFiles(): Int = {
    var result: Int = 0
    for (item <- items) {
      result += item.getNumberOfFiles
    }
    result
  }

  override def getNumberOfDirectory(): Int = {
    var result: Int = 1
    for (item <- items) {
      result += item.getNumberOfDirectory
    }
    result
  }

  override def listing(): String = {
    val sb: mutable.StringBuilder = new mutable.StringBuilder
    sb.append(this.getName).append("\n")
    for (item <- items) {
      sb.append(item.listing)
    }
    sb.toString
  }
}
```

A more concise FP approach:

```
class DirectoryFP(name: String) extends AbstractItem(name) {

  …
  override def getNumberOfFiles(): Int = {
    items.foldLeft(0){
      (acc, item) => acc + item.getNumberOfFiles()
    }
  }

  override def getNumberOfDirectory(): Int = {
    items.foldLeft(1){
      (acc, item) => acc + item.getNumberOfDirectory()
    }
  }

  override def listing(): String = {
    items.foldLeft(new mutable.StringBuilder()){
      (acc, item) => acc.append(item.listing)
    }.toString
  }
}
```

```
abstract class AbstractFile(name: String)
                extends AbstractItem(name) {
  protected var size: Int
  def getSize(): Int = size

  def getNumberOfFiles(): Int = 1
  def getNumberOfDirectory(): Int = 0
  def listing(): String = getName() + "\n"
}
```

Important questions

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

Is there any *code duplication*?

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

Why not having a *class hierarchy* for the different recursive operations?

Comments

In this naive approach we can see a number of problems:

Each new operation requires to modify the classes `Directory`, `AbstractItem`, `Item`, `AbstractFile`. Which could be cumbersome if the domain is externally provided.

Most of the methods in the class `Directory` are very similar, notably the recursion over the structure

We see that the cost of adding a new operation is high

We can lower this cost by using the visitor pattern

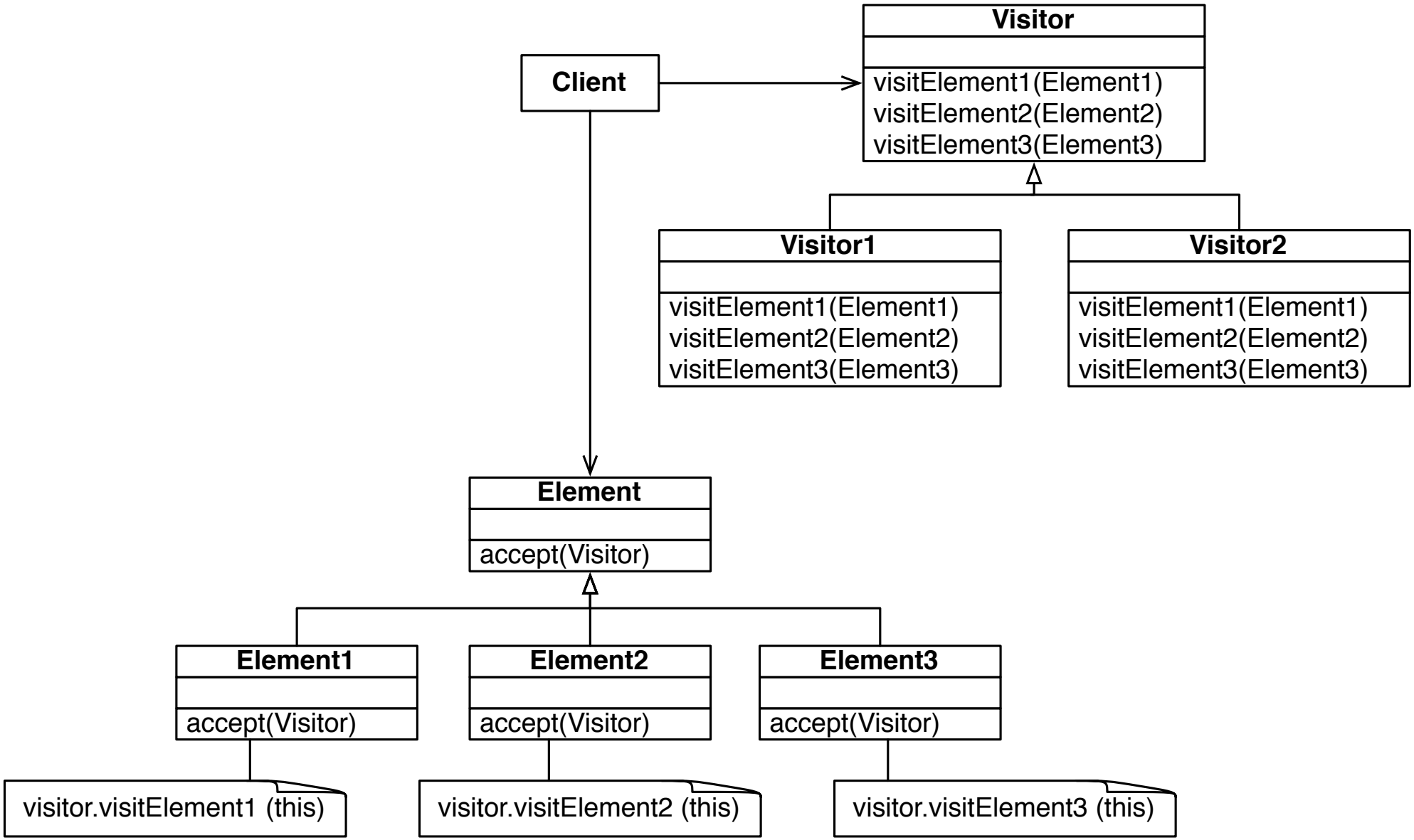
Visitor Pattern

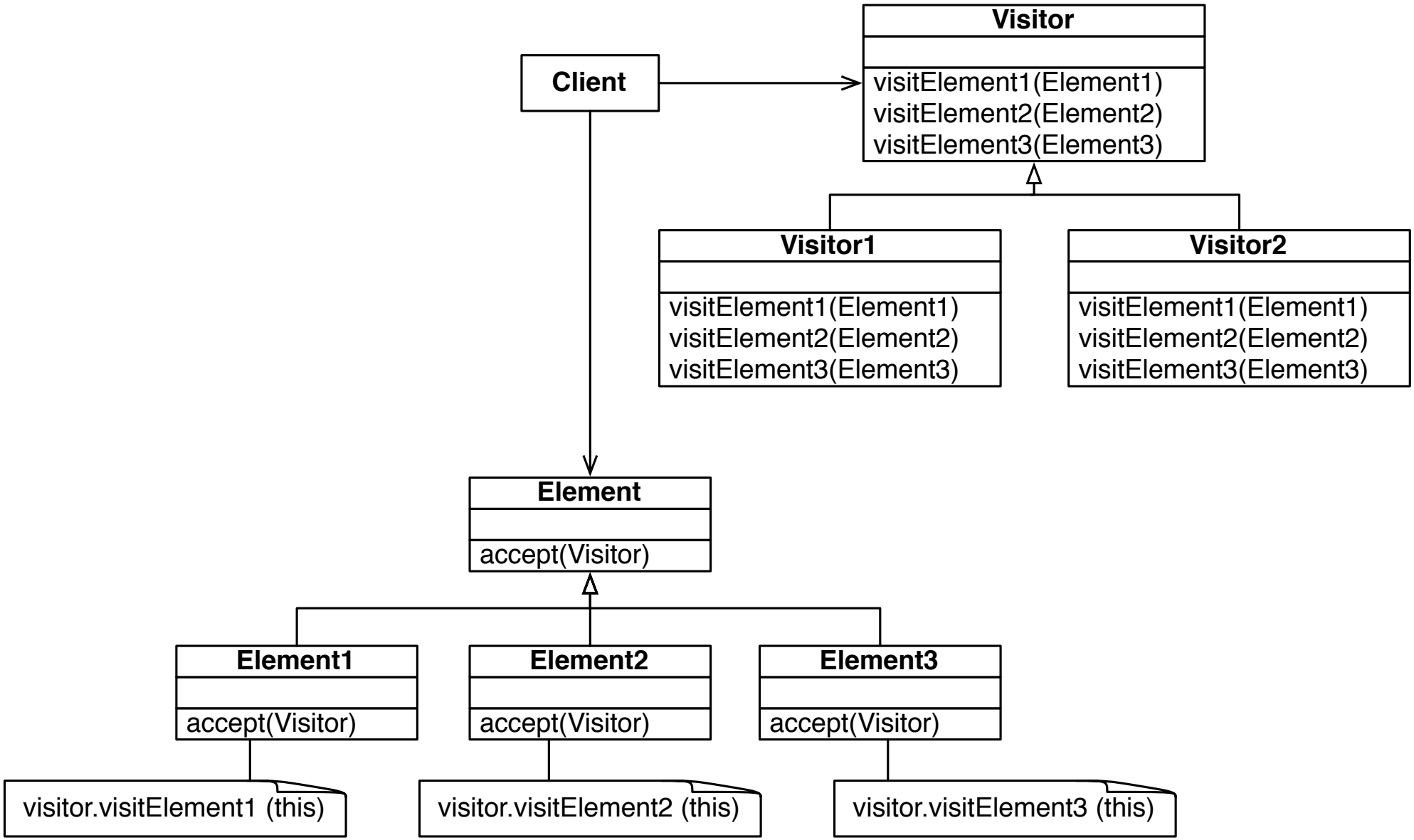
How do you accumulate information from heterogeneous classes?

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

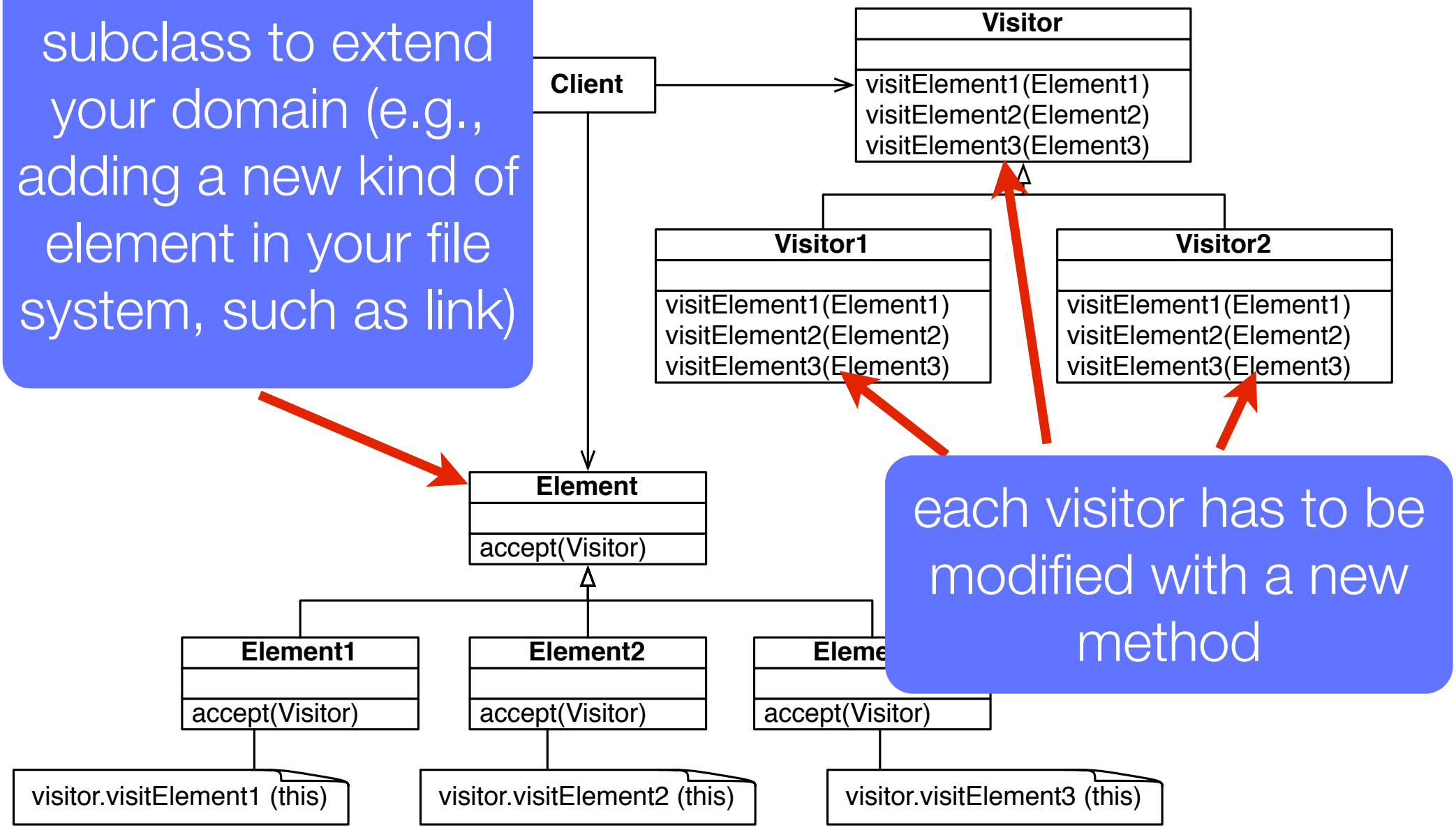

Visitor
visitElement1(Element1)
visitElement2(Element2)
visitElement3(Element3)
Client
Visitor1
visitElement1(Element1)
visitElement2(Element2)
visitElement3(Element3)
Visitor2
visitElement1(Element1)
visitElement2(Element2)
visitElement3(Element3)
Element
accept(Visitor)
Element1
accept(Visitor)
Element2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



Note that all methods have no return type (“void”)

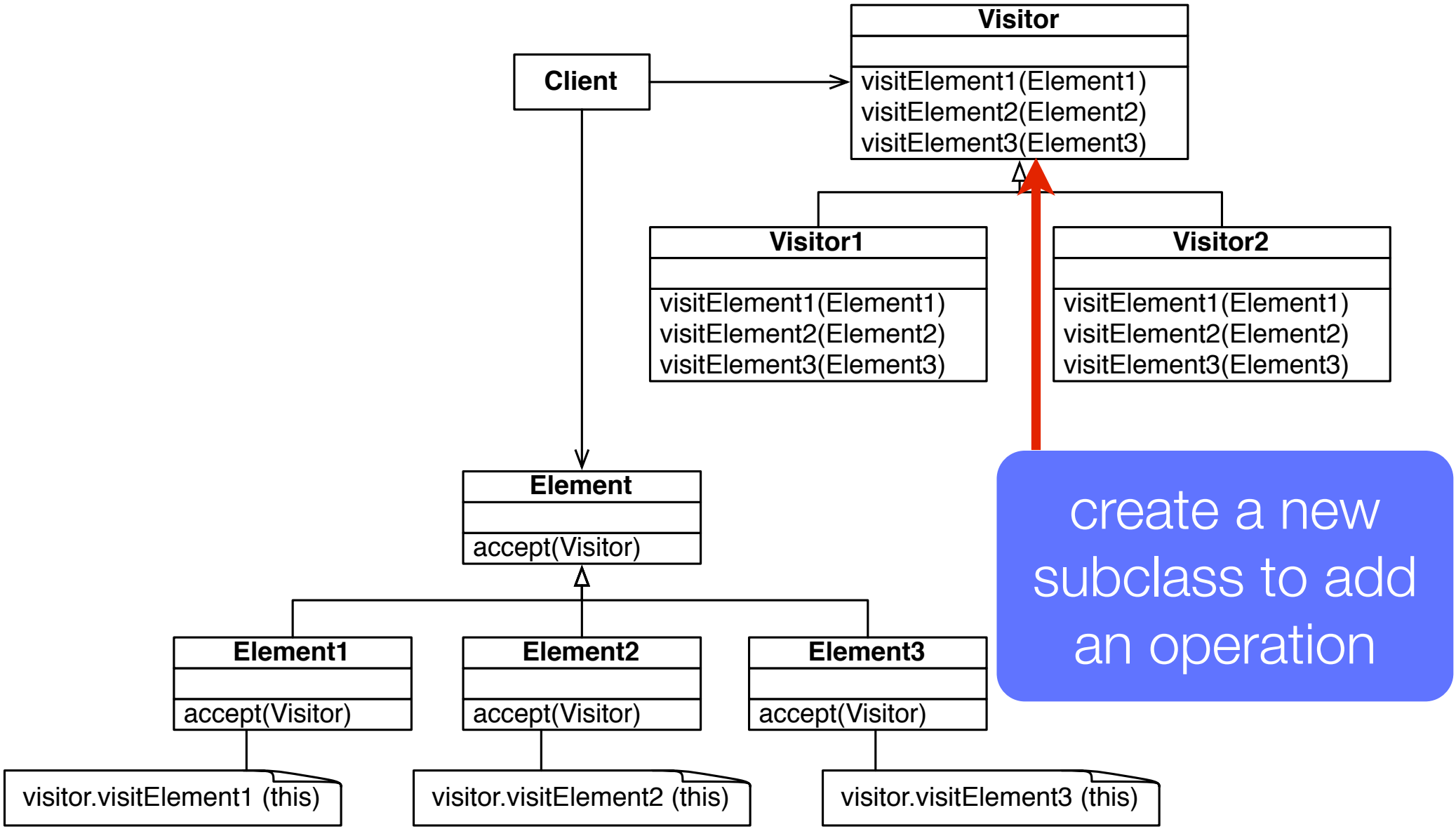
Extending the domain

create a new subclass to extend your domain (e.g., adding a new kind of element in your file system, such as link)



each visitor has to be modified with a new method

Adding an operation



Adding operations

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

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

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

Applying the visitor to our FileSystem

The client corresponds to the class `FileSystem`

The Element classes represents the `AbstractItem`, `Directory`, and `*File` classes

A visitor will replace each of the methods `getNumberOfFiles()`, `getNumberOfDirectory()`, and `listing()`

```
class FileSystem() {
  …
  def getNumberOfFiles(): Int = {
    val v = new NumberOfFileVisitor()
    root.accept(v)
    v.getResult()
  }

  def getNumberOfDirectory(): Int = {
    val v = new NumberOfDirectoryVisitor()
    root.accept(v)
    v.getResult()
  }

  def listing(): String = {
    val v = new ListingVisitor()
    root.accept(v)
    v.getResult()
  }
}
```

```
class Visitor {
  def visitBinaryFile(binaryFile: BinaryFile): Unit = {
  }

  def visitTextFile(textFile: TextFile): Unit = {
  }

  def visitDirectory(directory: Directory): Unit = {
    for (item <- directory.getItems()) {
      item.accept(this)
    }
  }
}
```

```
trait Item {
  def getSize(): Int
  def getName(): String

  def accept(aVisitor: Visitor): Unit
}
```

```
class BinaryFile(val aName: String, val content: Array[Byte])
                                        extends AbstractFile(aName) {
  protected var size = content.length

  def accept(aVisitor: Visitor): Unit = {
     aVisitor.visitBinaryFile(this)
  }
}
```

```
class TextFile(val aName: String, val content: String)
                                extends AbstractFile(aName) {
  protected var size = content.length

  def accept(aVisitor: Visitor): Unit = {
    aVisitor.visitTextFile(this)
  }
}
```

```
class NumberOfDirectoryVisitor extends Visitor {
  private var numberOfDirectory = 0

  override def visitDirectory(d: Directory): Unit = {
    super.visitDirectory(d)
    numberOfDirectory += 1
  }

  def getResult(): Int = numberOfDirectory
}
```

```
class NumberOfFileVisitor() extends Visitor {
  private var numberOfFiles: Int = 0

  override def visitBinaryFile(binaryFile: BinaryFile): Unit = {
    numberOfFiles += 1
  }

  override def visitTextFile(textFile: TextFile): Unit = {
    numberOfFiles += 1
  }

  def getResult(): Int = numberOfFiles
}
```

```
class ListingVisitor extends Visitor {
  private val sb = new mutable.StringBuilder

  override def visitTextFile(f: TextFile): Unit = {
    processItem(f.getName)
  }

  override def visitBinaryFile(f: BinaryFile): Unit = {
    processItem(f.getName)
  }

  override def visitDirectory(d: Directory): Unit = {
    processItem(d.getName)
    super.visitDirectory(d)
  }

  private def processItem(name: String): Unit = {
    sb.append(name).append("\n")
  }

  def getResult(): String = sb.toString
}
```

Points worth to discuss

Where to put the recursion?

In the class Directory or in the Visitor?

Having the recursion in the visitor requires an accessor to the Directory.elements variable. The pattern forces us to make some of the state public.

Some solutions found on internet may favor code overloading:

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

What is your opinion on this?

Points worth to discuss

In our domain, we have a class `AbstractItem` and `AbstractFile`. Shouldn't we also have a method `visitAbstractItem(AbstractItem)` and `visitAbstractFile(AbstractFile)`?

Yes, we could, but the visitor will be slightly more complex to write. In general, only the the leaf (and non-abstract classes) should have a corresponding visit method

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?

License

dcc

CIENCIAS DE LA COMPUTACIÓN
UNIVERSIDAD DE CHILE

www.dcc.uchile.cl

/ DCCUCHILE