The Visitor Pattern

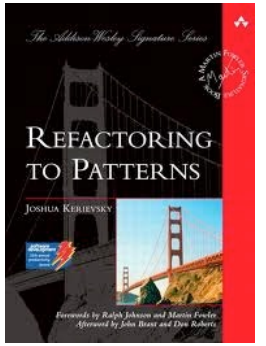
Alexandre Bergel
http://bergel.eu
08-11-2021

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

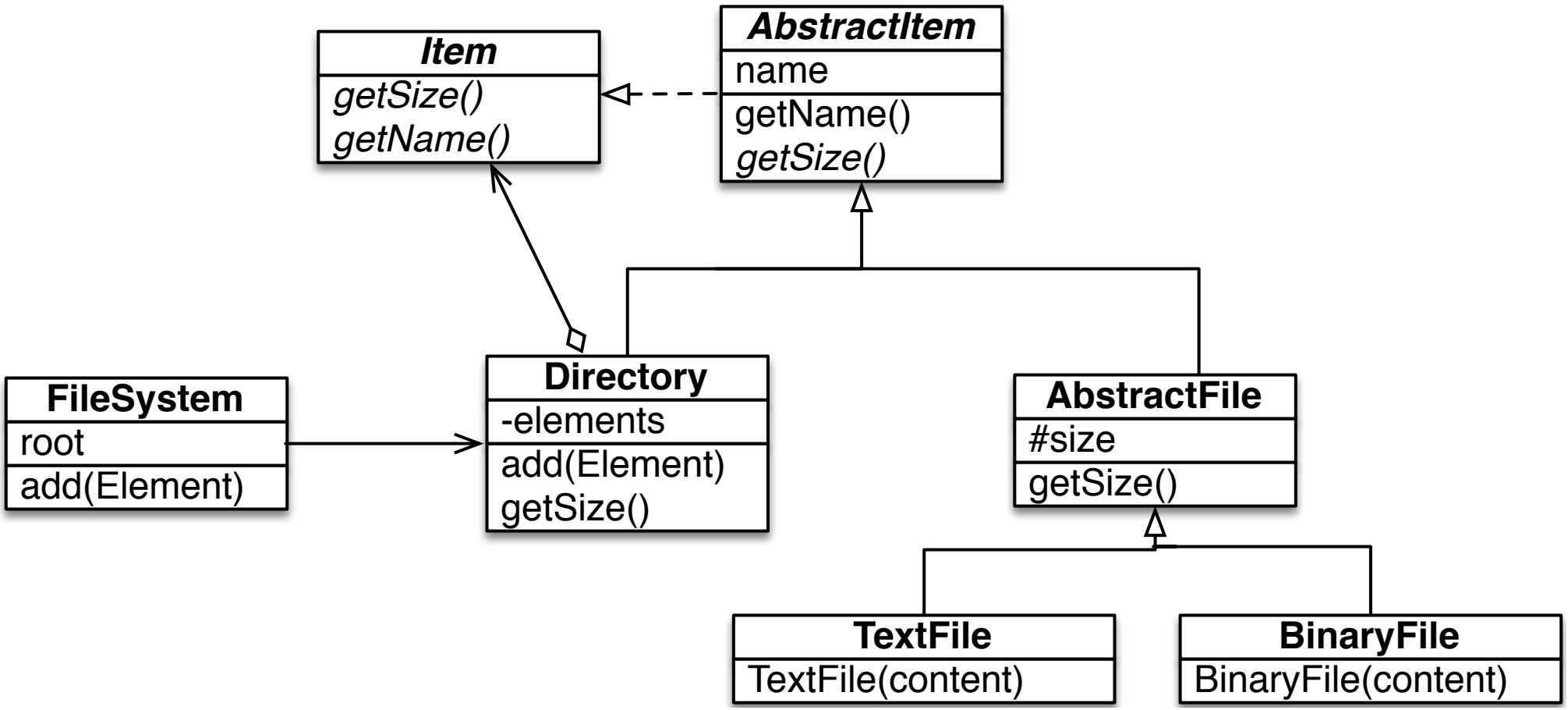
The objective is this lecture is double

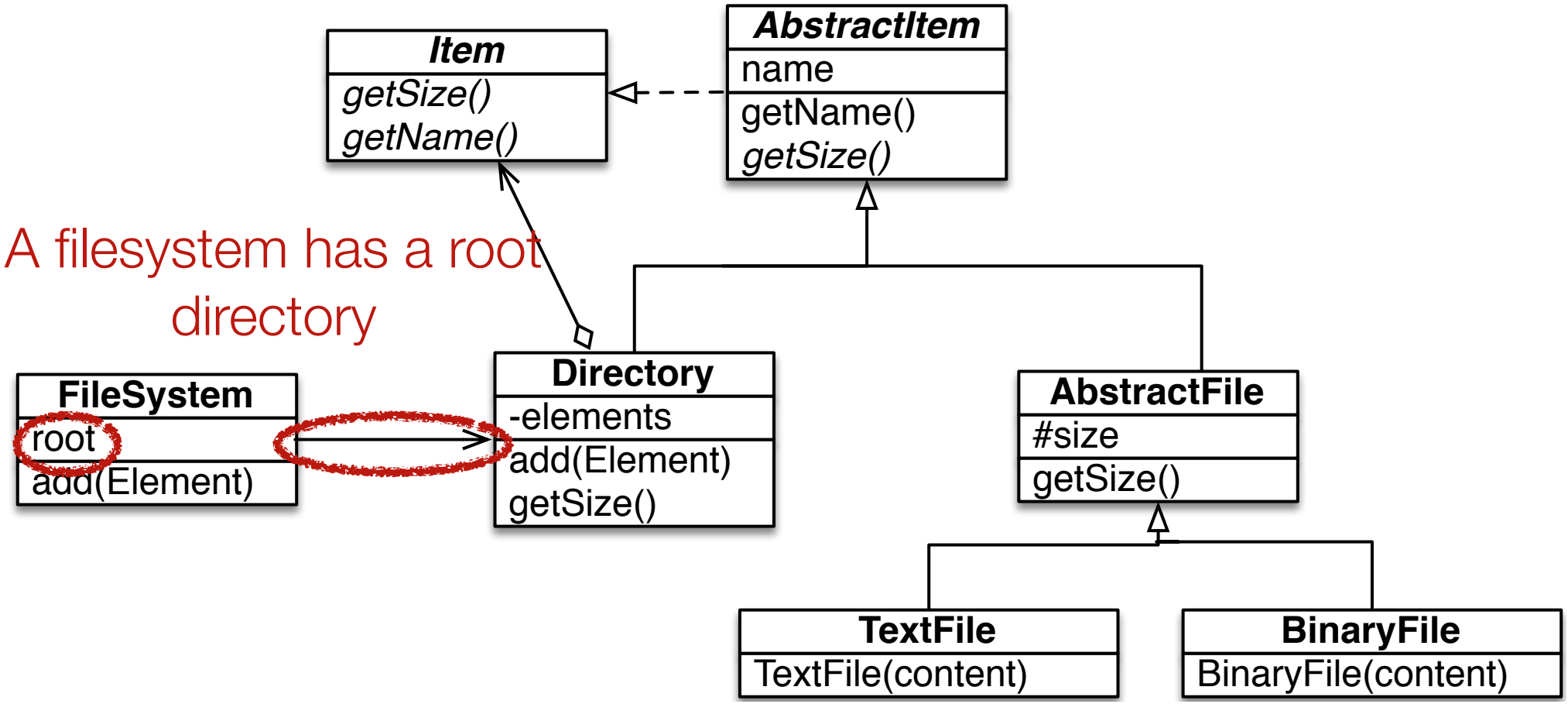
- Face the problem addressed by the visitor pattern
- Introduce the visitor pattern, which is a spectacular illustration of a proper separation of concern

Exercise

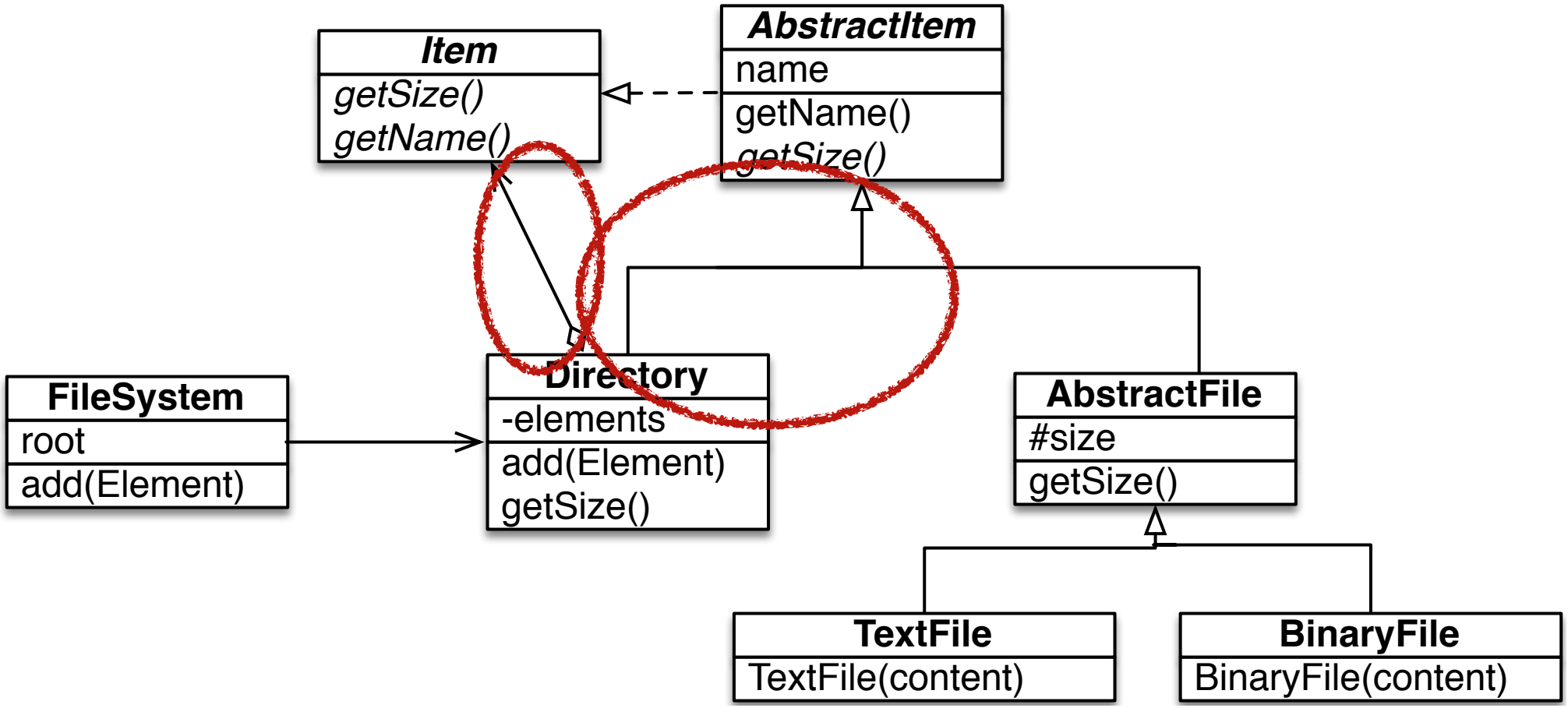
Un "file system" es un componente esencial de mucho sistemas operativos. Por este ejercicio, vamos a considerar los elementos siguientes:

- 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)

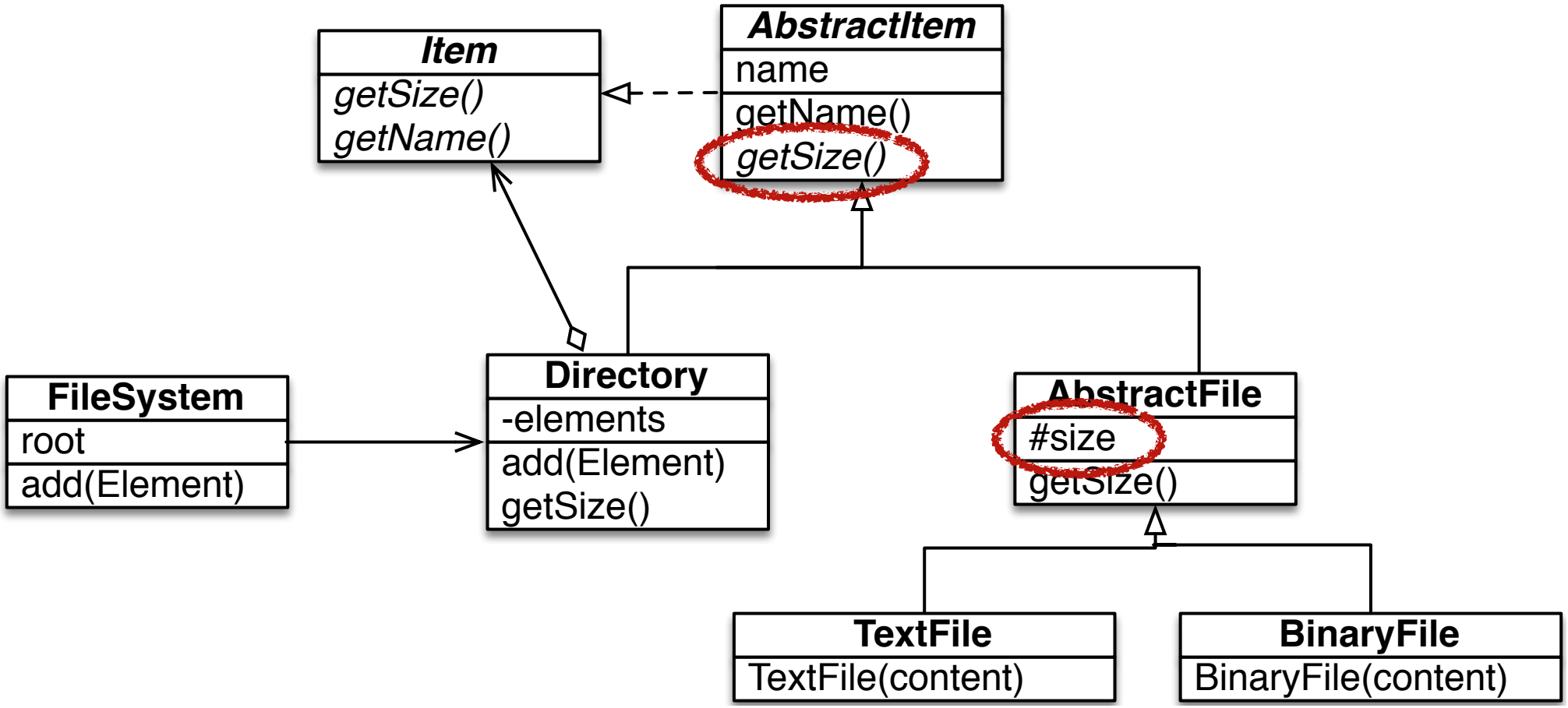

Item
getSize()
getName()
AbstractItem
name
getName()
getSize()
A filesystem has a root directory
FileSystem
root
add(Element)
Directory
-elements
add(Element)
getSize()
AbstractFile
#size
getSize()
TextFile
TextFile(content)
BinaryFile
BinaryFile(content)

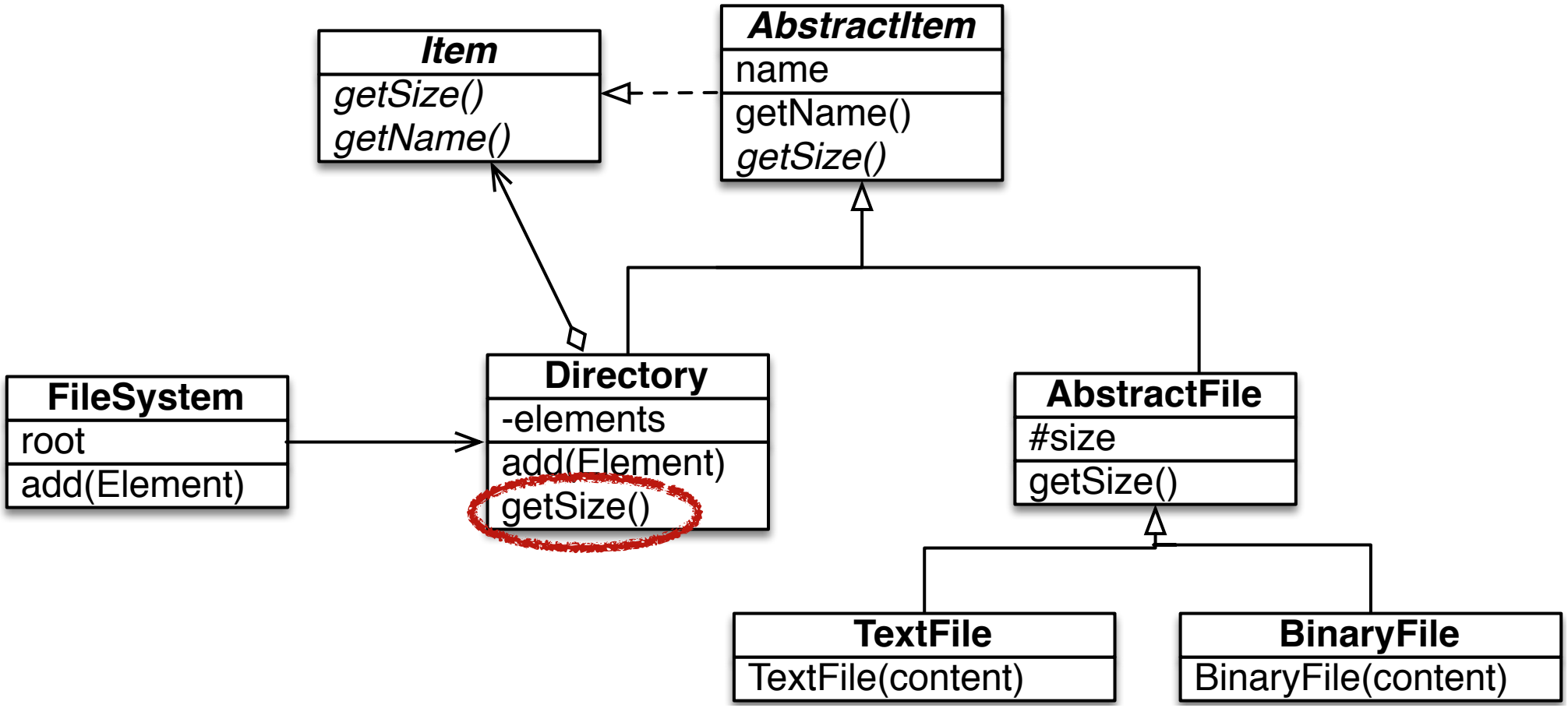
The composite patterns make a directory to contains items



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

```java
public class FileSystemTest {
    private FileSystem emptyFs, fs;
    private Directory d1;
    private Directory d2;

    @Before
    public void setUp() {
        emptyFs = new FileSystem();

        d1 = new Directory("d1");
        d2 = new Directory("d2");
        d1.add(d2);
        d1.add(new TextFile("file.txt", "hello world!"));

        byte[] c = {'1', 'c'};
        d1.add(new BinaryFile("file.txt", c));

        fs = new FileSystem();
        fs.add(d1);
    }

    @Test
    public void testGetSize() {
        assertEquals(0, emptyFs.getSize());
        assertEquals(14, fs.getSize());
    }
}
```

```java
public class FileSystem {
    private Directory root;

    public FileSystem() {
        root = new Directory("root");
    }

    public void add(Item item) {
        root.add(item);
    }

    public int getSize() {
        return root.getSize();
    }
}
```

```java
public interface Item {
    int getSize();
    String getName();
}
```

```java
public abstract class AbstractItem implements Item {
    private String name;

    @Override
    public abstract int getSize();

    @Override
    public String getName() {
        return name;
    }

    public AbstractItem(String aName) {
        name = aName;
    }
}
```

```java
public class Directory extends AbstractItem {
    private List<Item> items = new ArrayList<>();

    public Directory(String aName) {
        super(aName);
    }

    public void add(Item anItem) {
        items.add(anItem);
    }

    @Override
    public int getSize() {
        int result = 0;
        for(Item item : items)
            result += item.getSize();
        return result;
    }
}
```

```java
public class AbstractFile extends AbstractItem {
    protected int size;

    public AbstractFile(String aName) {
        super(aName);
    }

    @Override
    public int getSize() {
        return size;
    }
}
```

```java
public class TextFile extends AbstractFile {

    public TextFile(String aName, String content) {
        super(aName);
        size = content.length();
    }
}
```

```java
public class BinaryFile extends AbstractFile {
    public BinaryFile(String aName, byte[] content) {
        super(aName);
        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
- ...

```java
@Test
public void testGetNumberOfFile() {
    assertEquals(0, emptyFs.getNumberOfFiles());

    assertEquals(2, fs.getNumberOfFiles());

    TextFile aFile = new TextFile("tmp.txt", "a file system example");
    Directory d = new Directory("another directory");
    d.add(aFile);
    fs.add(d);
    assertEquals(3, fs.getNumberOfFiles());
}

@Test
public void testGetNumberOfDirectory() {
    assertEquals(1, emptyFs.getNumberOfDirectory());

    assertEquals(3, fs.getNumberOfDirectory());

    TextFile aFile = new TextFile("tmp.txt", "a file system example");
    Directory d = new Directory("another directory");
    d.add(aFile);
    fs.add(d);
    assertEquals(4, fs.getNumberOfDirectory());
}

@Test
public void testListing() {
    String result = "root\n";
    assertEquals(result, emptyFs.listing());

    result = "root\nd1\nd2\nfile.txt\nfile.obj\n";
    assertEquals(result, fs.listing());
}
```

```
public interface Item {
    int getSize();
    String getName();
    int getNumberOfFiles();
    int getNumberOfDirectory();
    String listing();
}
```

```java
public abstract class AbstractItem implements Item {
    private String name;

    @Override
    public abstract int getSize();

    @Override
    public String getName() {
        return name;
    }

    public AbstractItem(String aName) {
        name = aName;
    }

    public abstract int getNumberOfFiles();

    public abstract int getNumberOfDirectory();

    public abstract String listing();
}
```

```
public class Directory extends AbstractItem {
…
    @Override
    public int getNumberOfFiles() {
        int result = 0;
        for(Item item : items)
            result += item.getNumberOfFiles();
        return result;
    }

    @Override
    public int getNumberOfDirectory() {
        int result = 1;
        for(Item item : items)
            result += item.getNumberOfDirectory();
        return result;
    }

    @Override
    public String listing() {
        StringBuilder sb = new StringBuilder();
        sb.append(this.getName()).append("\n");
        for(Item item : items)
            sb.append(item.listing());
        return sb.toString();
    }
}
```

```java
public class AbstractFile extends AbstractItem {
    protected int size;

    public AbstractFile(String aName) {
        super(aName);
    }

    @Override
    public int getNumberOfFiles() {
        return 1;
    }

    @Override
    public int getNumberOfDirectory() {
        return 0;
    }

    @Override
    public String listing() {
        return this.getName() + "\n";
    }

    @Override
    public int getSize() {
        return size;
    }
}
```

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

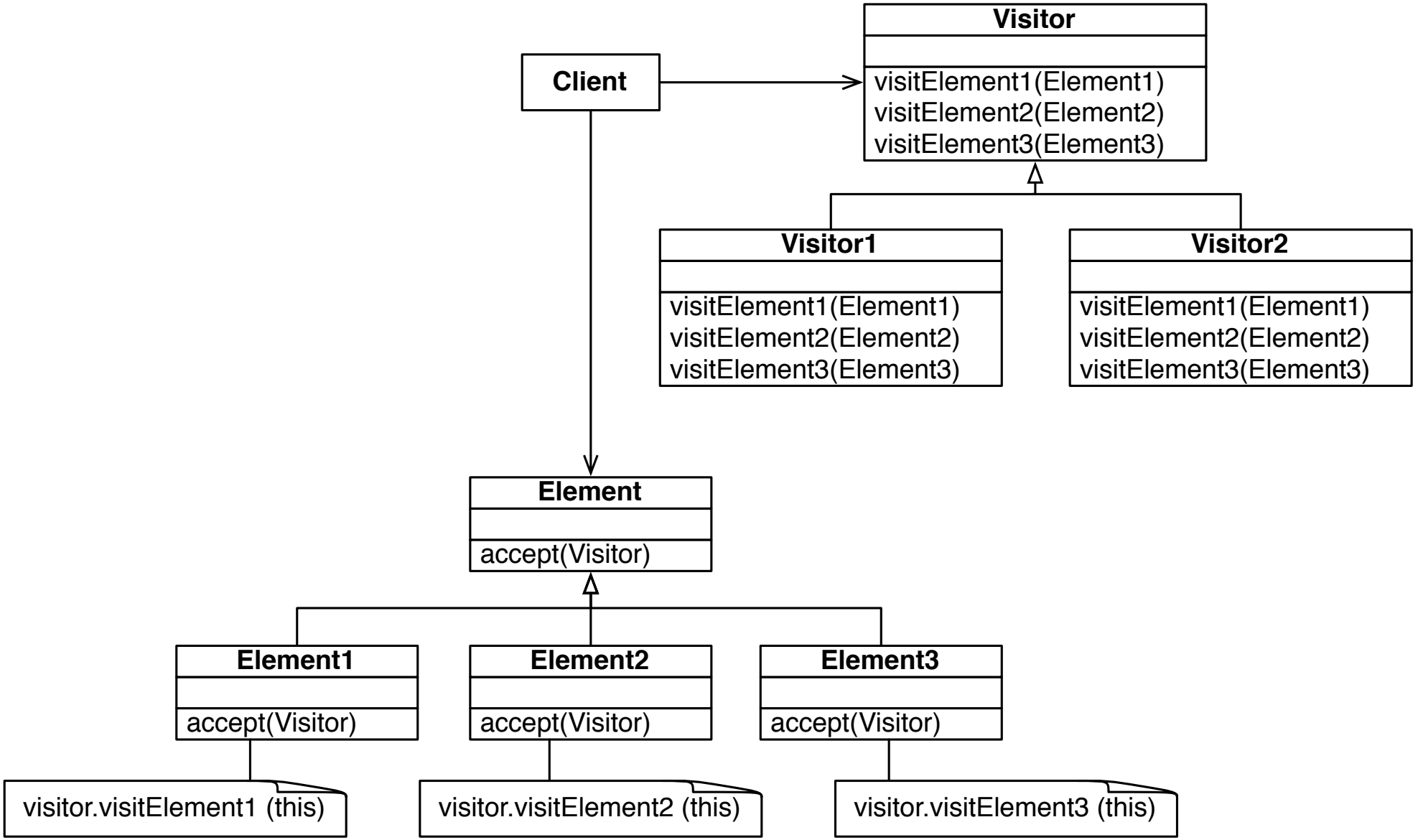
Visitor Pattern

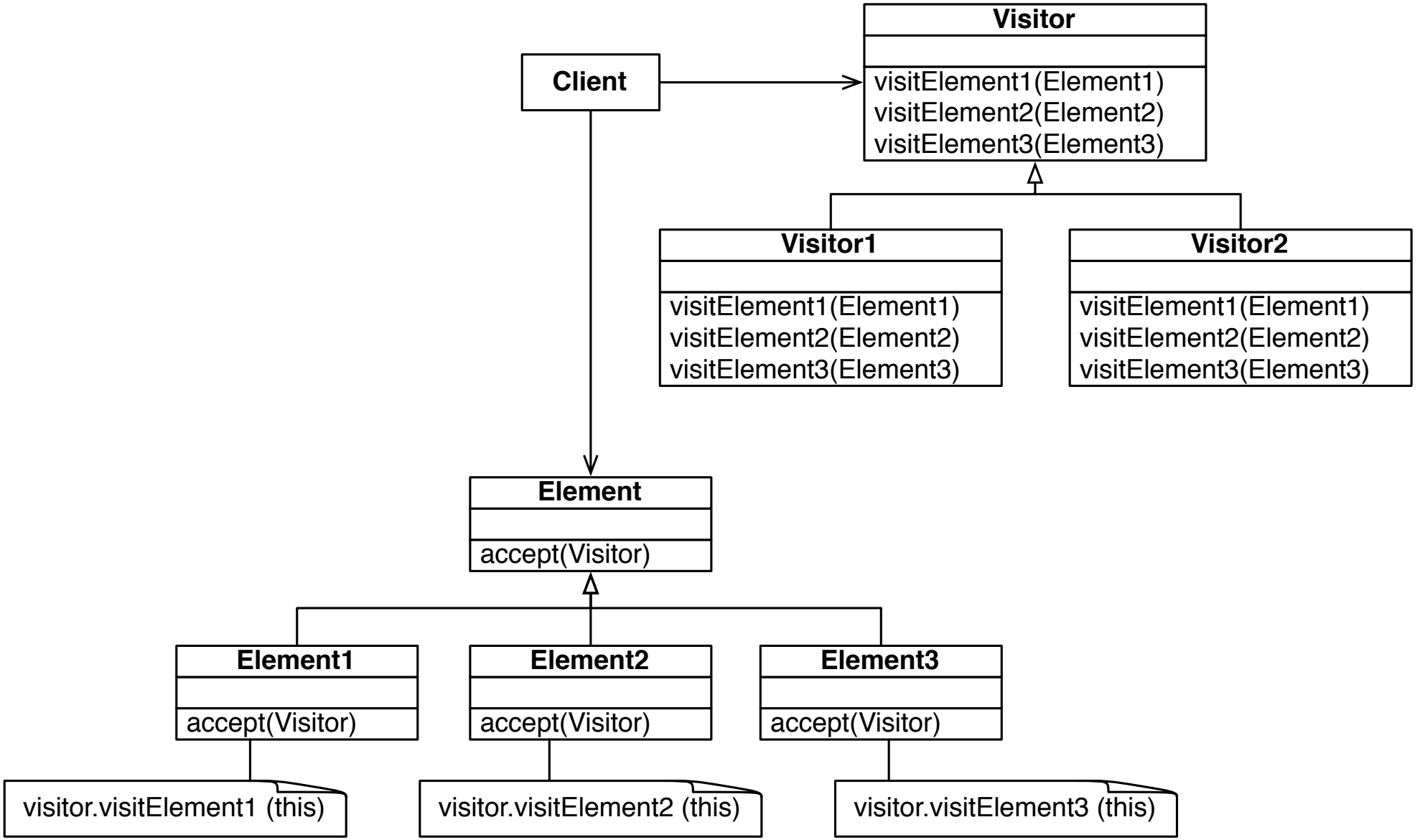
How do you accumulate information from heterogeneous classes?

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

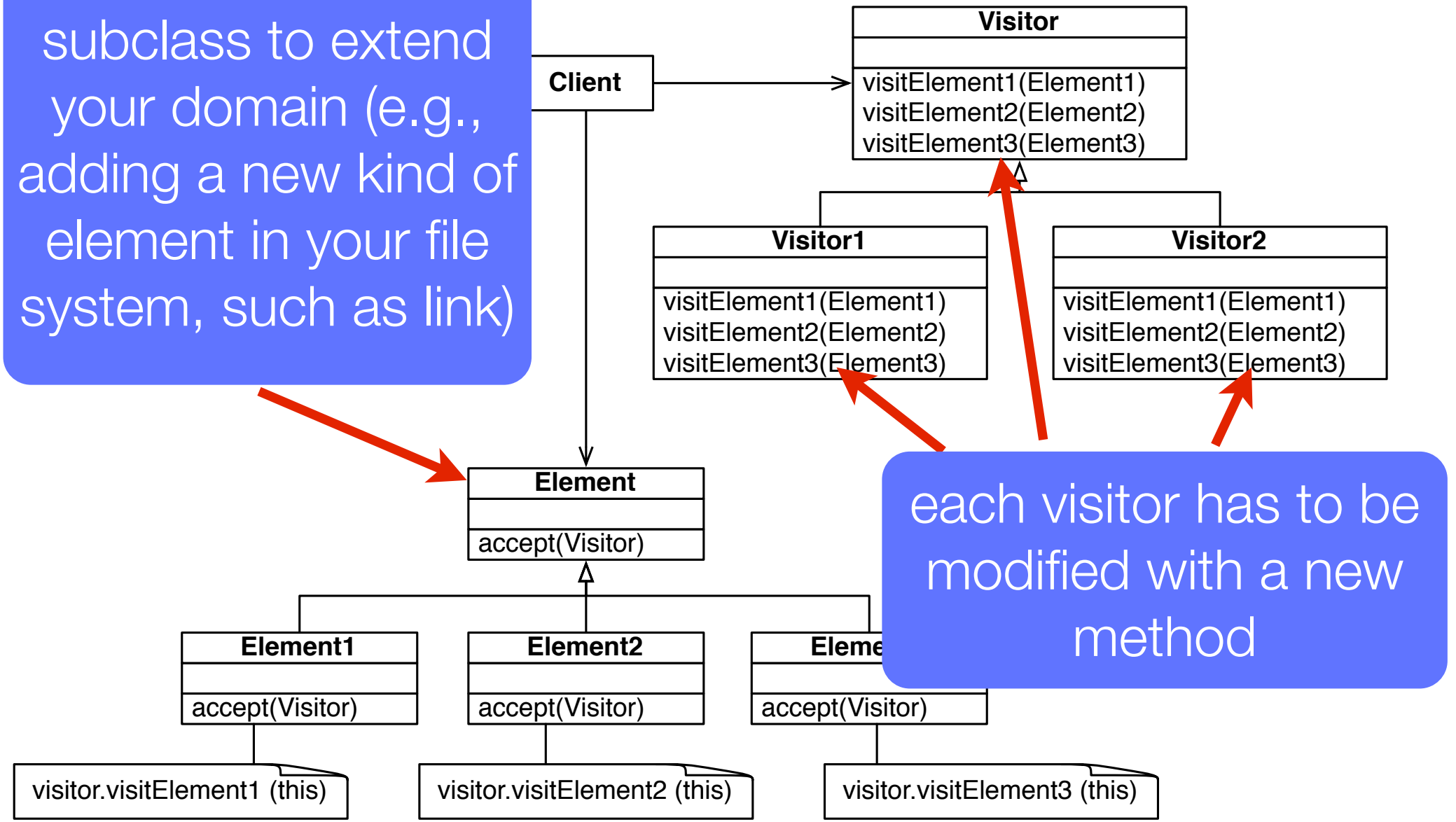

Visitor
visitElement1(Element1)
visitElement2(Element2)
visitElement3(Element3)
Client
Visitor1
visitElement1(Element1)
visitElement2(Element2)
visitElement3(Element3)
Visitor2
visitElement1(Element1)
visitElement2(Element2)
visitElement3(Element3)
Element
accept(Visitor)
Element1
accept(Visitor)
Element2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



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

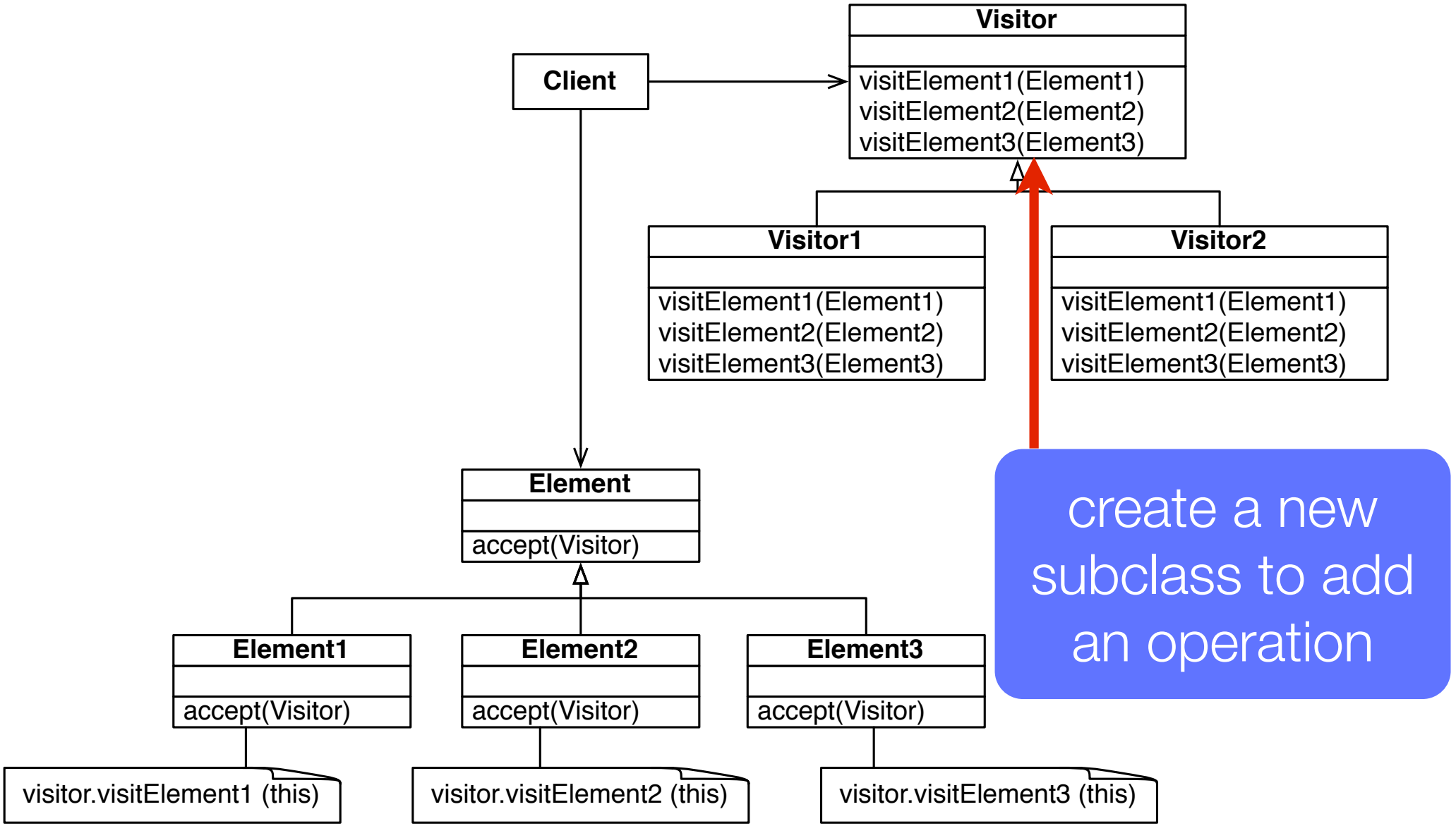
Extending the domain

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()`

```
public class FileSystem {
    private Directory root;

    public FileSystem() {
        root = new Directory("root");
    }

    public void add(Item item) {
        root.add(item);
    }

    public int getSize() {
        return root.getSize();
    }

    public int getNumberOfFiles() {
        NumberOfFileVisitor v = new NumberOfFileVisitor();
        root.accept(v);
        return v.getResult();
    }

    public int getNumberOfDirectory() {
        NumberOfDirectoryVisitor v = new NumberOfDirectoryVisitor();
        root.accept(v);
        return v.getResult();
    }

    public String listing() {
        ListingVisitor v = new ListingVisitor();
        root.accept(v);
        return v.getResult();
    }
}
```

```
public class Visitor {
    public void visitBinaryFile(BinaryFile binaryFile) {
    }

    public void visitTextFile(TextFile textFile) {
    }

    public void visitDirectory(Directory directory) {
        for(Item item : directory.getItems())
            item.accept(this);
    }
}
```

```
public interface Item {
    int getSize();
    String getName();
    void accept(Visitor aVisitor);
}
```

```java
public class BinaryFile extends AbstractFile {
    public BinaryFile(String aName, byte[] content) {
        super(aName);
        size = content.length;
    }

    @Override
    public void accept(Visitor aVisitor) {
        aVisitor.visitBinaryFile(this);
    }
}
```

```java
public class TextFile extends AbstractFile {

    public TextFile(String aName, String content) {
        super(aName);
        size = content.length();
    }

    @Override
    public void accept(Visitor aVisitor) {
        aVisitor.visitTextFile(this);
    }
}
```

```java
public class NumberOfDirectoryVisitor extends Visitor {
    private int numberOfDirectory = 0;

    @Override
    public void visitDirectory(Directory d) {
        super.visitDirectory(d);
        numberOfDirectory++;
    }
    public int getResult() {
        return numberOfDirectory;
    }
}
```

```java
public class NumberOfFileVisitor extends Visitor {
    private int numberOfFiles = 0;

    @Override
    public void visitTextFile(TextFile f) {
        numberOfFiles++;
    }

    @Override
    public void visitBinaryFile(BinaryFile f) {
        numberOfFiles++;
    }

    public int getResult() {
        return numberOfFiles;
    }
}
```

```
public class ListingVisitor extends Visitor {
    private StringBuilder sb = new StringBuilder();

    @Override
    public void visitTextFile(TextFile f){
        processItem(f.getName());
    }

    @Override
    public void visitBinaryFile(BinaryFile f){
        processItem(f.getName());
    }

    @Override
    public void visitDirectory(Directory d){
        processItem(d.getName());
        super.visitDirectory(d);
    }

    private void processItem(String name) {
        sb.append(name).append("\n");
    }

    public String getResult() {
        return 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