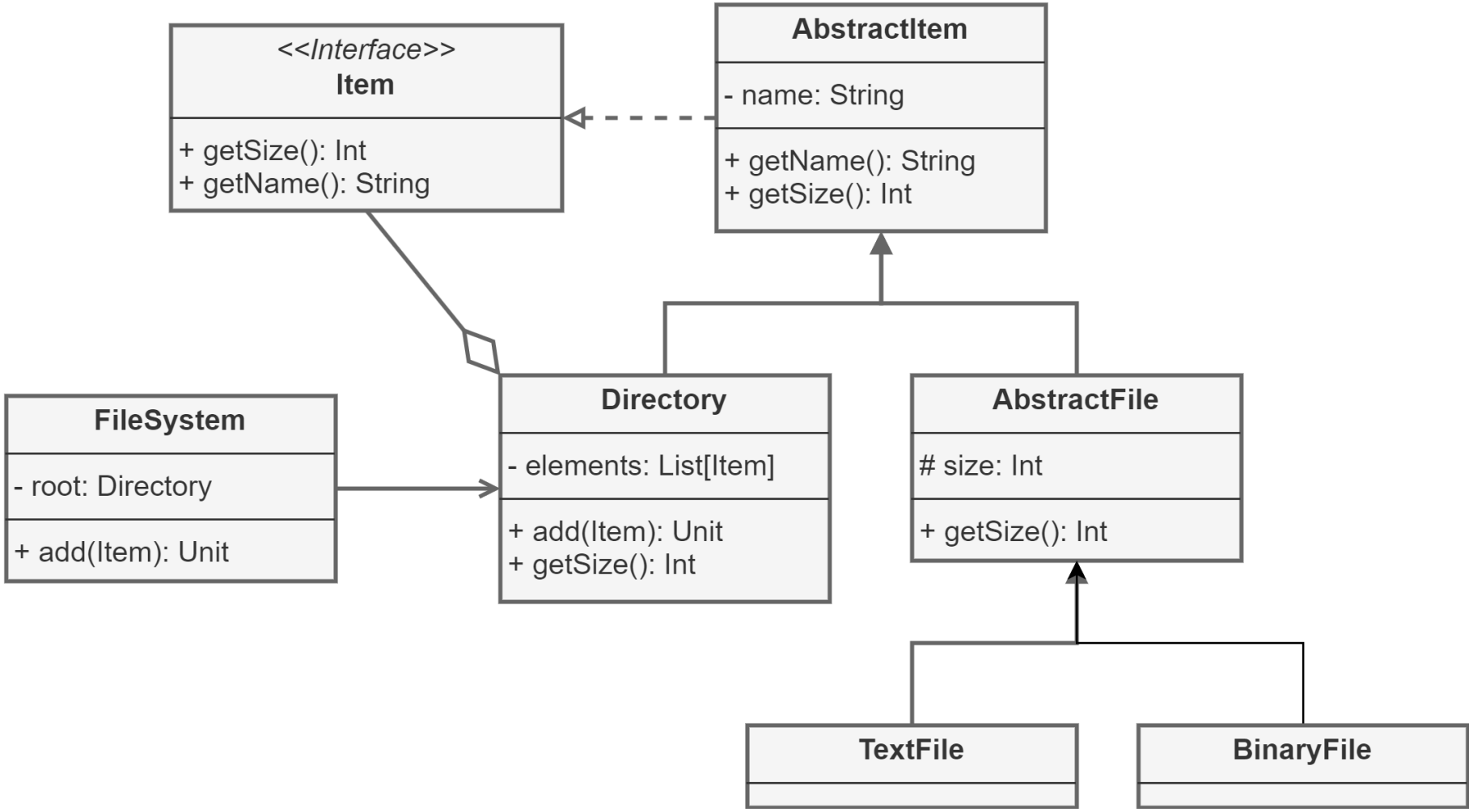
The Visitor Pattern

Ignacio Slater

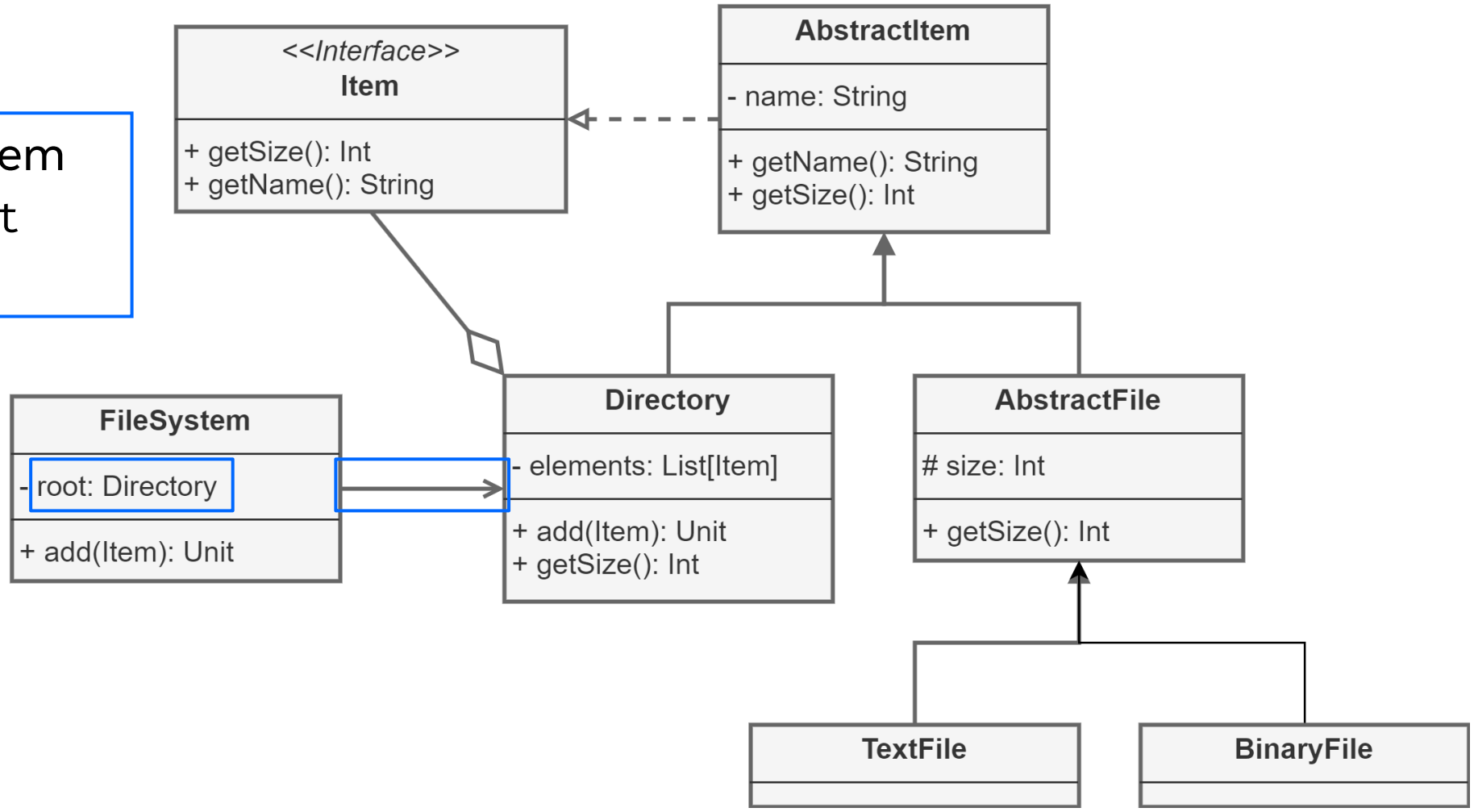
Matías Toro

Exercise

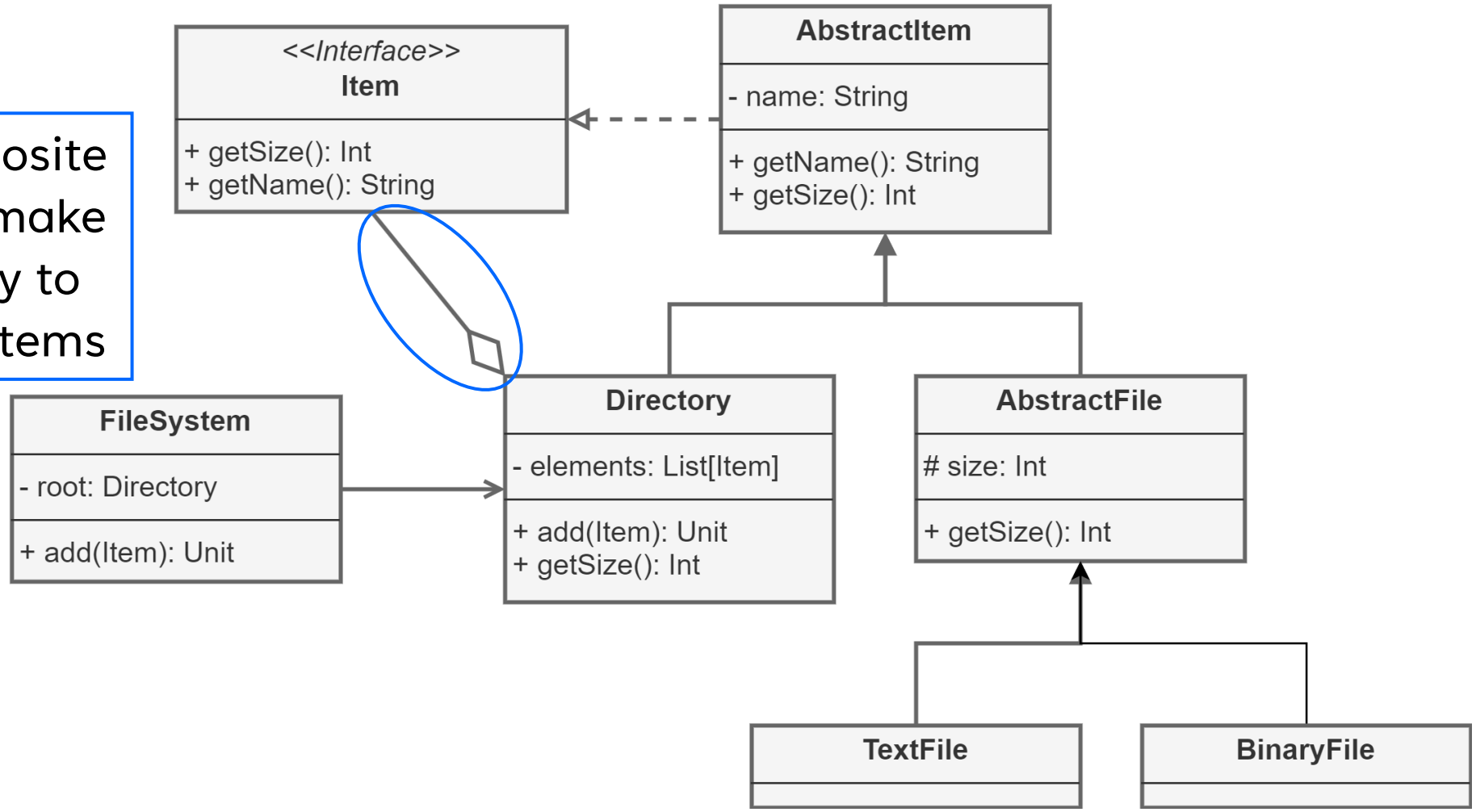
- A **file system** is an essential component of most operating systems
- For this exercise we will consider the following cases:
  - A file system is composed of files and directories
  - A file can be text or binary
  - A file system has only one root directory
  - A directory can have files and directories
  - Every element in the system has a *name* and a *size*

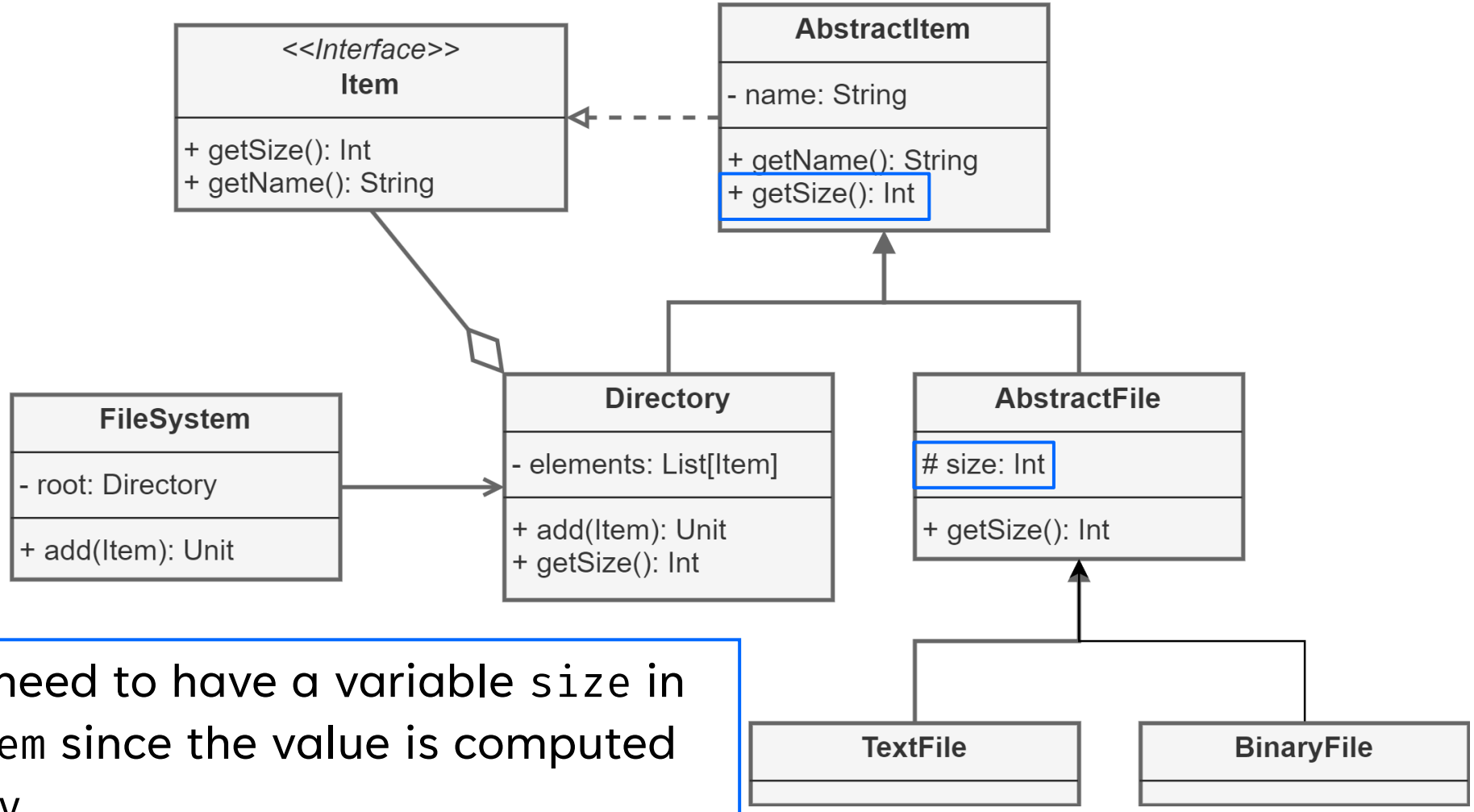

<<Interface>>
Item
+ getSize(): Int
+ getName(): String
AbstractItem
- name: String
+ getName(): String
+ getSize(): Int
FileSystem
- root: Directory
+ add(Item): Unit
Directory
- elements: List[Item]
+ add(Item): Unit
+ getSize(): Int
AbstractFile
size: Int
+ getSize(): Int
TextFile
BinaryFile

A file system has a `root` directory



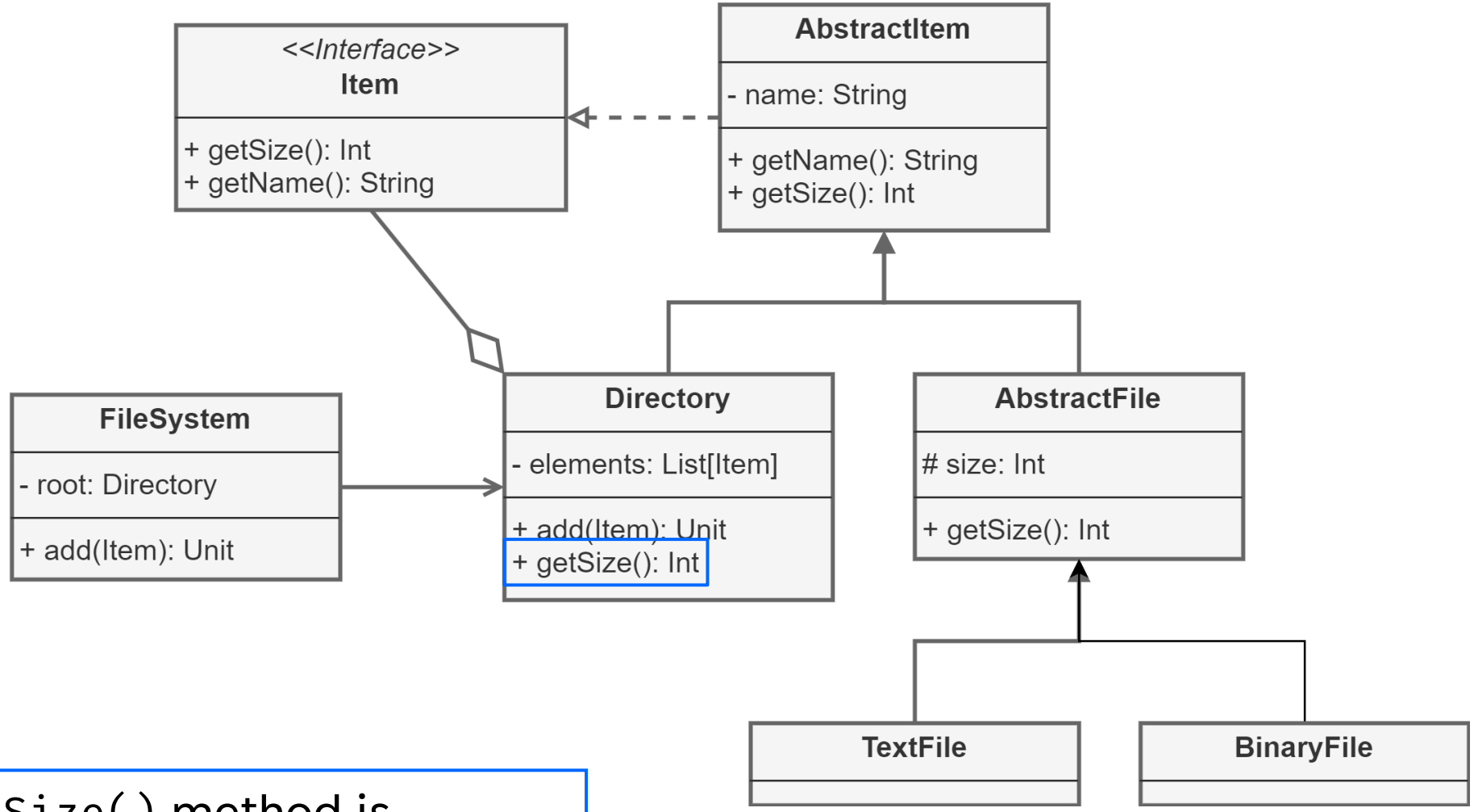
The composite patterns make a directory to contains items





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

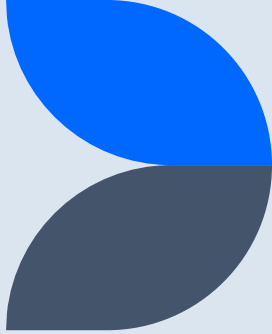
```
class FileSystemTest extends munit.FunSuite {
  private var emptyFs: Option[FileSystem] = None
  private var fs: Option[FileSystem] = None
  private var d1: Option[Directory] = None
  private var d2: Option[Directory] = None
}
```

```
class FileSystemTest extends munit.FunSuite {
  private var emptyFs: Option[FileSystem] = None
  private var fs: Option[FileSystem] = None
  private var d1: Option[Directory] = None
  private var d2: Option[Directory] = None

  override def beforeEach(context: BeforeEach): Unit = {
  }

  test("testGetSize"){
  }
}
```

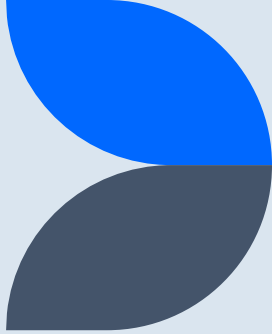
```
class FileSystemTest extends munit.FunSuite {
  private var emptyFs: Option[FileSystem] = None
  private var fs: Option[FileSystem] = None
  private var d1: Option[Directory] = None
  private var d2: Option[Directory] = None

  override def beforeEach(context: BeforeEach): Unit = {
    emptyFs = Some(new FileSystem())
    d1 = Some(new Directory("d1"))
    d2 = Some(new Directory("d2"))
  }

  test("testGetSize"){
  }
}
```

```
class FileSystemTest extends munit.FunSuite {
  private var emptyFs: Option[FileSystem] = None
  private var fs: Option[FileSystem] = None
  private var d1: Option[Directory] = None
  private var d2: Option[Directory] = None

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

  test("testGetSize"){
  }
}
```

```
class FileSystemTest extends munit.FunSuite {
  private var emptyFs: Option[FileSystem] = None
  private var fs: Option[FileSystem] = None
  private var d1: Option[Directory] = None
  private var d2: Option[Directory] = None

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

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



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

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

  def add(d: Directory): Unit = root.add(d)
}
```



```scala
trait Item {
  def getSize: Int
  def getName: String
}
```

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

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

  override def getSize: Int = size
}
```

```
class TextFile(name: String, val content: String) extends AbstractFile(name) {
  override protected def size: Int = content.length
}
```

```
class BinaryFile(name: String, val content: Array[Byte]) extends AbstractFile(name) {
  override protected def size: Int = content.length
}
```

Exercise

Now we would like to add some operations

- Get the **total number of files** contained in the system
- Get the **total number of directories** in the system
- Do a **recursive listing**

```
class FileSystemTest extends munit.FunSuite {
  /* ... */
  test("testNumberOfFiles") {
    assertEquals(0, emptyFs.get.numberOfDirectories)
    assertEquals(2, fs.get.numberOfFiles)
    val file = new TextFile("tmp.txt", "a file system example")
    val d = new Directory("another directory")
    d.add(file)
    fs.get.add(d)
    assertEquals(3, fs.get.numberOfFiles)
  }

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

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

```
trait Item {
  def getSize: Int
  def getName: String
  // New methods
  def getNumberOfFiles: Int
  def getNumberOfDirectories: Int
  def listing: String
}
```

```
class Directory(name: String) extends AbstractItem(name) {
  /* ... */
  override def getNumberOfFiles: Int = {
    var files = 0
    for (child <- children) {
      files += child.getNumberOfFiles
    }
    files
  }

  override def getNumberOfDirectories: Int = {
    var directories = 0
    for (child <- children) {
      directories += child.getNumberOfDirectories
    }
    directories
  }

  override def listing: String = {
    val builder = new StringBuilder
    builder.append(s"Directory: $name\n")
    for (child <- children) {
      builder.append(child.listing)
    }
    builder.toString()
  }
}
```

A more concise FP approach:

```
class Directory(name: String) extends AbstractItem(name) {
  /* ... */
  override def getNumberOfFiles: Int =
    children.foldLeft(0)((acc, child) => acc + child.getNumberOfFiles)

  override def getNumberOfDirectories: Int =
    children.foldLeft(1)((acc, child) => acc + child.getNumberOfDirectories)

  override def listing: String =
    children.foldLeft("")((acc, child) => acc + child.getName + "\n")
}
```

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

  override def getSize: Int = size

  override def getNumberOfFiles: Int = 1

  override def getNumberOfDirectories: Int = 0

  override def listing: String = s"$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 few 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*

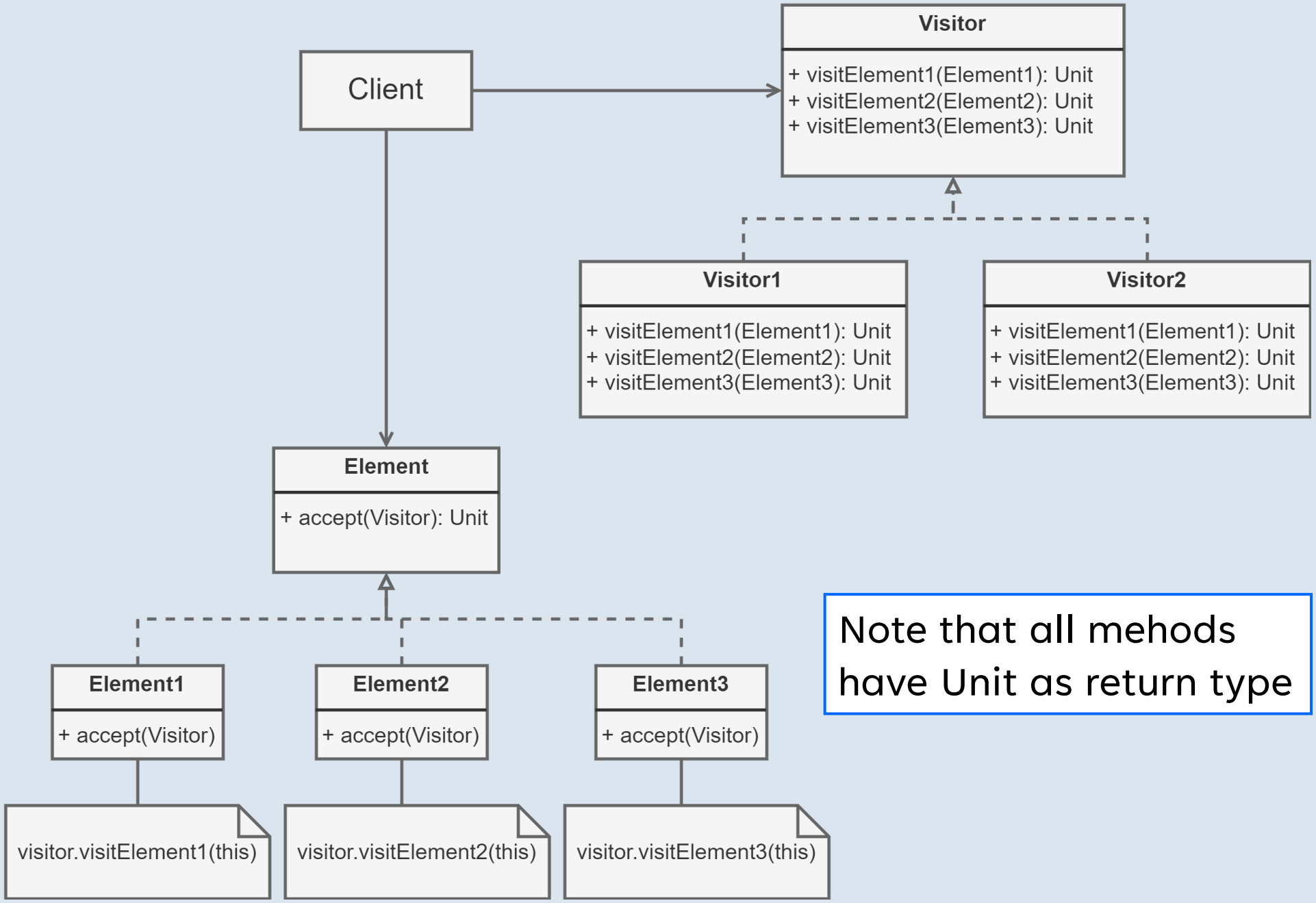
Visitor Pattern

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.

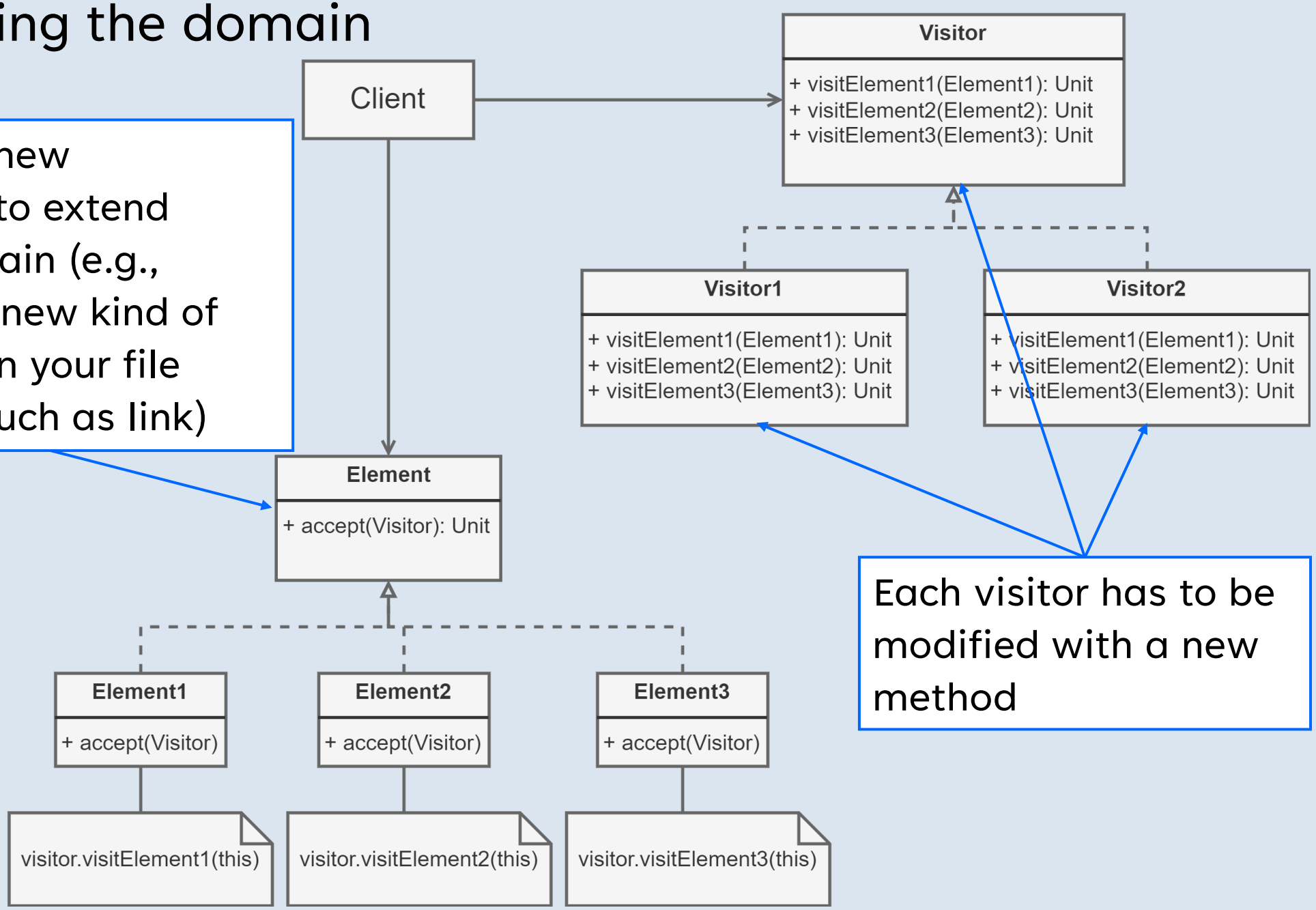
> Intensively use **double dispatch**



Note that all mehods have Unit as return type

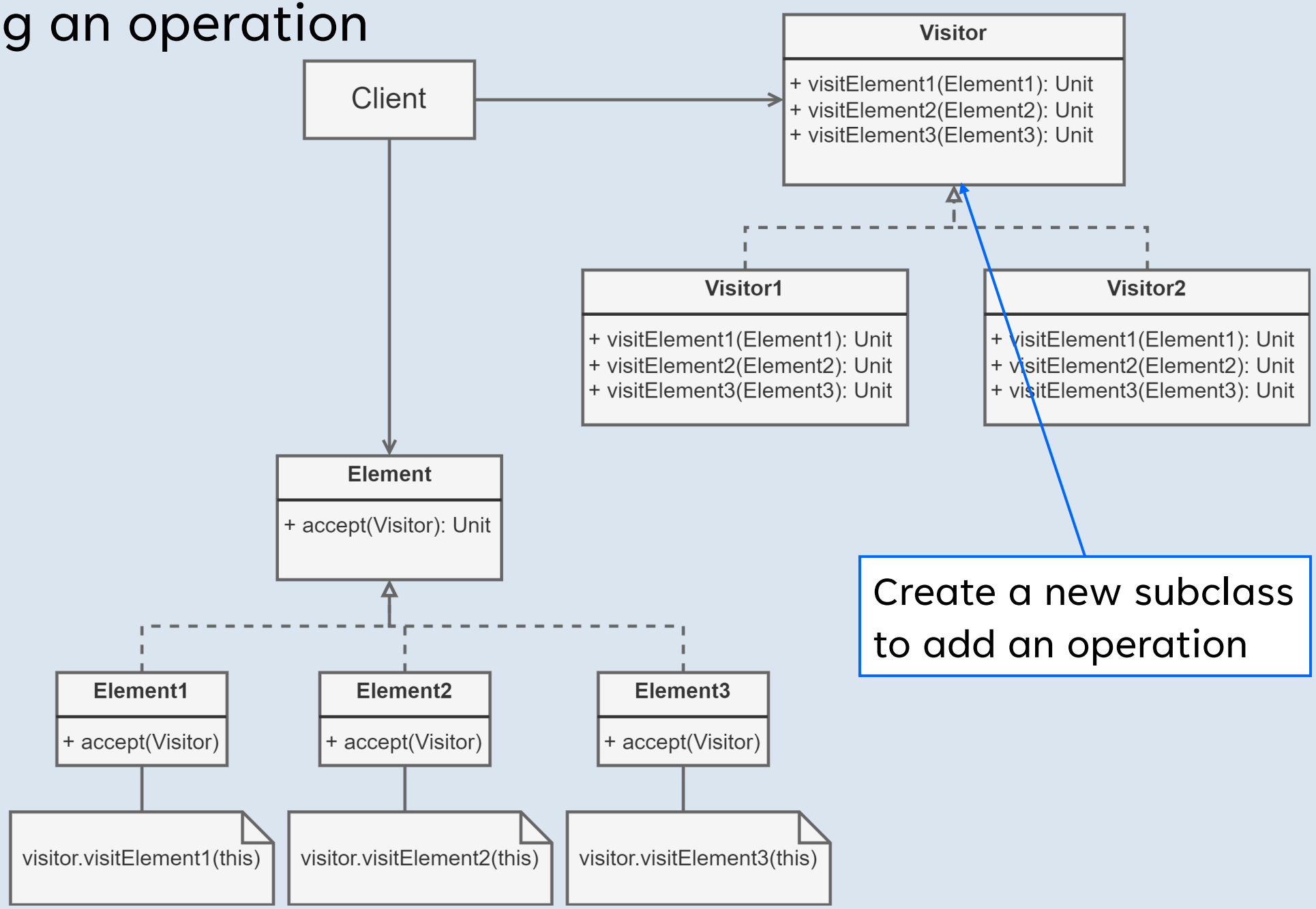
Extending the domain

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

Addding 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`, `getNumberOfDirectories`, and `listing`

```
class FileSystem {
  /* ... */
  def numberOfFiles: Int = {
    val v = new NumberOfFilesVisitor
    root.accept(v)
    v.numberOfFiles
  }

  def numberOfDirectories: Int = {
    val v = new NumberOfDirectoriesVisitor
    root.accept(v)
    v.numberOfDirectories
  }

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

```
class Visitor {
  def visitBinaryFile(file: BinaryFile): Unit = {
    // Do nothing
  }

  def visitTextFile(file: TextFile): Unit = {
    // Do nothing
  }

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

```
trait Item {
  def getSize: Int
  def getName: String
  def accept(v: Visitor): Unit
}
```

```
class BinaryFile(name: String, val content: Array[Byte]) extends AbstractFile(name) {
  override protected def size: Int = content.length

  override def accept(v: Visitor): Unit = v.visitBinaryFile(this)
}
```

```
class TextFile(name: String, val content: String) extends AbstractFile(name) {
  override protected def size: Int = content.length

  override def accept(v: Visitor): Unit = v.visitTextFile(this)
}
```

```scala
class Directory(name: String) extends AbstractItem(name) {
  val children = ListBuffer[Item]()

  def add(item: Item): Unit = children += item

  override def getSize: Int = {
    var size = 0
    for (child <- children) {
      size += child.getSize
    }
    size
  }

  override def accept(v: Visitor): Unit = v.visitDirectory(this)
}
```

```
class NumberOfDirectoriesVisitor extends Visitor {
  private var _numberOfDirectories: Int = 0

  def numberOfDirectories: Int = _numberOfDirectories

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

```
class NumberOfFilesVisitor extends Visitor {
  private var _numberOfFiles: Int = 0

  def numberOfFiles: Int = _numberOfFiles

  override def visitBinaryFile(file: BinaryFile): Unit =
    _numberOfFiles += 1

  override def visitTextFile(file: TextFile): Unit =
    _numberOfFiles += 1
}
```

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

  override def visitBinaryFile(file: BinaryFile): Unit =
    processItem(file.getName)

  override def visitTextFile(file: TextFile): Unit =
    processItem(file.getName)

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

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

  def listing: String = sb.toString
}
```

Points worth discussion

- Where to put the recursion?

  In the directory class or in the visitor?

- Having the recursion in the visitor requires an accessor to the directory children 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 discussion

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 this 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?