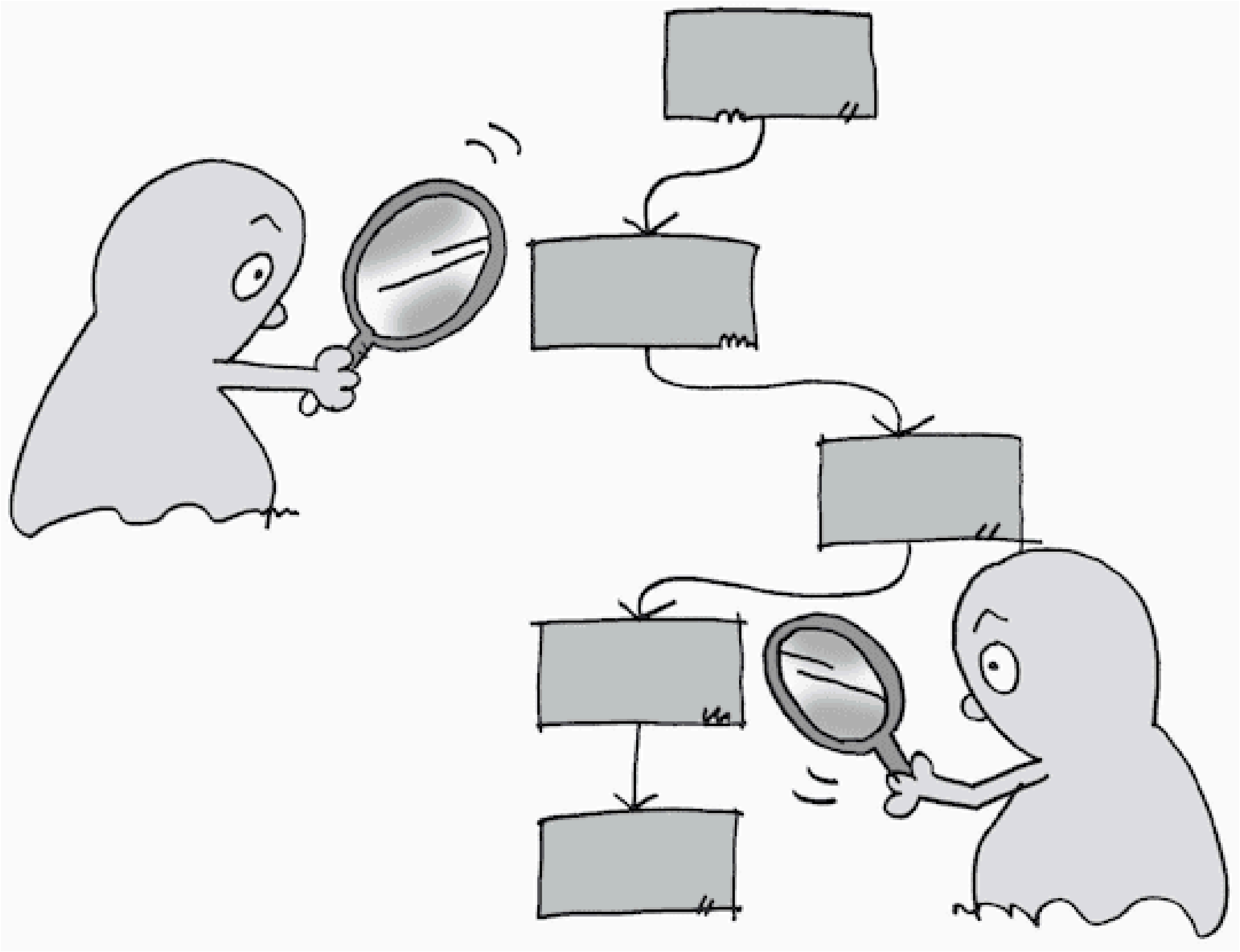
Visitor Pattern

Separate Operations from Object Structure using Double Dispatch



Visitor Pattern

Think of **different specialists** visiting the same **building**:

- **Fire inspector**: Checks fire safety in each room
- **Security auditor**: Examines security measures in each area
- **Cleaning crew**: Cleans each room with appropriate methods
- **Maintenance worker**: Repairs different equipment types

Each visitor performs different operations on the same structure without altering the building itself.

The Problem

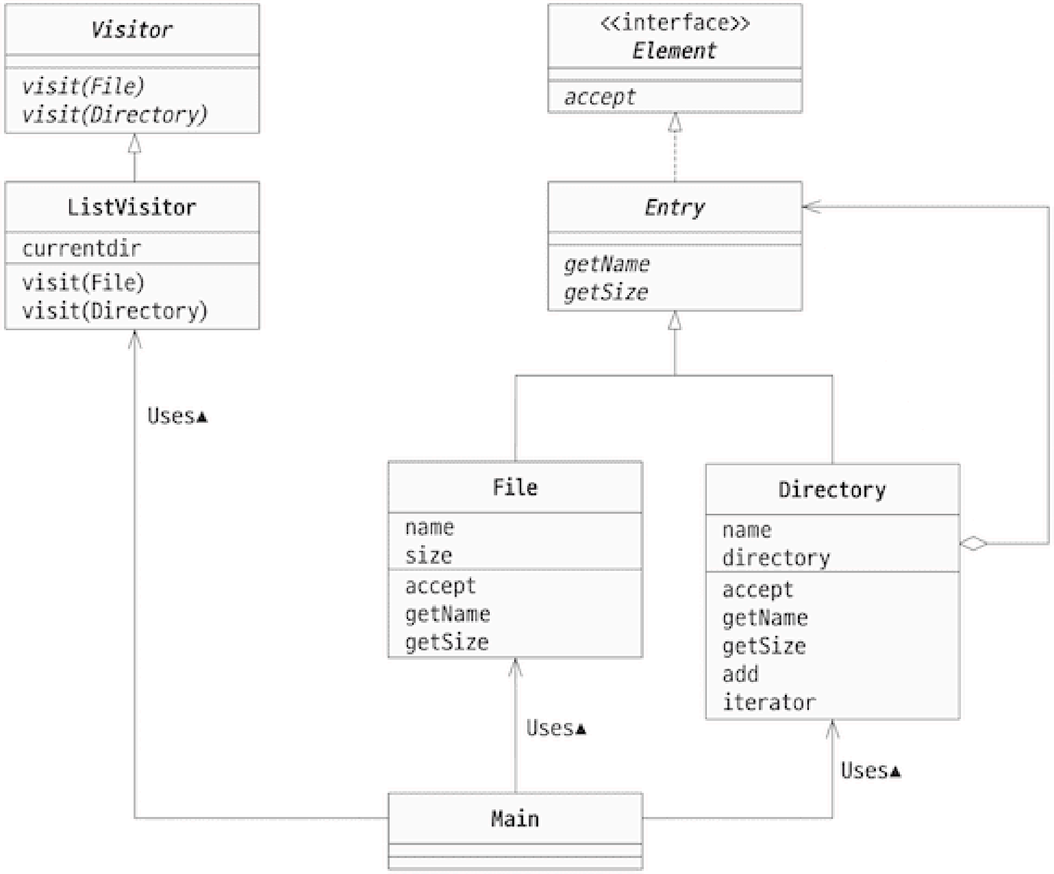
- We have a **file system** with **files** and **directories**.
- We want to add different operations: calculate size, list contents, find files, compress, backup, etc.
- Traditional approach: add methods to each class → **violates Open/Closed Principle**.

Challenge: How to add new operations without changing existing classes?

The *Visitor* as the Solution

- We separate **operations** (*Visitors*) from **data structures** (*Elements*).
- **Elements** `accept` *visitors* and provide access to their internal structure.
- **Visitors** implement specific *operations* for different element types.

The Solution (Design)



Step 1: Understand the Players

In this design, we have players:

- *Visitor* (Visitor)
  - **ConcreteVisitor** (ListVisitor, SizeVisitor)
- *Element* (Element)
  - **ConcreteElement** (File, Directory)

Step 2: Double Dispatch Mechanism

- **Elements** call `visitor.visit_xxx(self)` where `xxx` is the element type.
- **Visitors** implement `visit_xxx()` methods for each element type.

Step 3: Understand abstractions (Visitor-Element)

- We have a *Visitor* that `defines` the interface for operations on elements.
    - *Visitor* - declares visit operations for each element type
    - *Element* - declares `accept(visitor)` method
- **Elements** accept visitors and delegate operations to them.

- Notice that **Element** has an `accept(visitor)` method.
  - It calls the appropriate `visit_xxx()` method on the visitor.
- Notice that **Visitor** has `visit_xxx()` methods for each element type.
  - Each method implements the specific operation for that element type.

Step 4: Understand concretion (Visitor-Element)

- We have **File**, **Directory** (elements) and **ListVisitor**, **SizeVisitor** (visitors).
    - **ConcreteElement** (File, Directory): implement `accept()` method
    - **ConcreteVisitor** (ListVisitor, SizeVisitor): implement specific operations

Code

- Main Method
- Element Classes
- Visitor Classes

Main Method

```python
from directory import Directory
from file import File
from list_visitor import ListVisitor
from size_visitor import SizeVisitor

def main():
    print("=== Visitor Pattern Example ===\n")

    # Create file system structure
    root = create_file_system()

    # Use different visitors
    print("File system structure:")
    root.accept(ListVisitor()) # starting point

    print("\nCalculating total size:")
    size_visitor = SizeVisitor()
    total_size = root.accept(size_visitor) # starting point
    print(f"Total size: {total_size} bytes")
```

Step 1: Create object structure

```
def create_file_system():
    root_dir = Directory("root")
    bin_dir = Directory("bin")

    root_dir.add(bin_dir)
    bin_dir.add(File("vi", 10000))
    bin_dir.add(File("latex", 20000))

    return root_dir
```

- **Directory** and **File** are the **elements** in our object structure.
- They implement the *Element* interface with `accept()` method.

Step 2: Apply visitors to perform operations

```
# List contents
root.accept(ListVisitor())

# Calculate size
size_visitor = SizeVisitor()
total_size = root.accept(size_visitor)
```

- **Visitors** implement different *operations* on the same structure.
- Each **element** calls the appropriate `visit_xxx()` method on the visitor.

Step 3: Add new operations (visit functions) to the Visitor

```python
class FindVisitor(Visitor):
    def __init__(self, pattern):
        self.pattern = pattern
        self.found_files = []
    # visitor for file element
    def visit_file(self, file):
        if self.pattern in file.get_name():
            self.found_files.append(file)
    # visitor for directory element
    def visit_directory(self, directory):
        for entry in directory:
            entry.accept(self)
```

- New **operations** can be added without modifying existing classes.
- **Visitor** pattern keeps related operations together.

Element Classes

```
# element.py
from abc import ABC, abstractmethod

class Element(ABC):
    @abstractmethod
    def accept(self, visitor):
        pass

# file.py
class File(Element):
    def __init__(self, name, size):
        self.name = name
        self.size = size
    # any visitor with visit_file can access this
    def accept(self, visitor):
        return visitor.visit_file(self)
```

Visitor Classes

```
# visitor.py
class Visitor(ABC):
    # visit elements
    @abstractmethod
    def visit_file(self, file):
        pass
    @abstractmethod
    def visit_directory(self, directory):
        pass

# size_visitor.py
class SizeVisitor(Visitor):
    def __init__(self):
        self.total_size = 0

    def visit_file(self, file):
        self.total_size += file.get_size()
        return self.total_size
    def visit_directory(self, directory):
        for entry in directory:
            entry.accept(self)
        return self.total_size
```

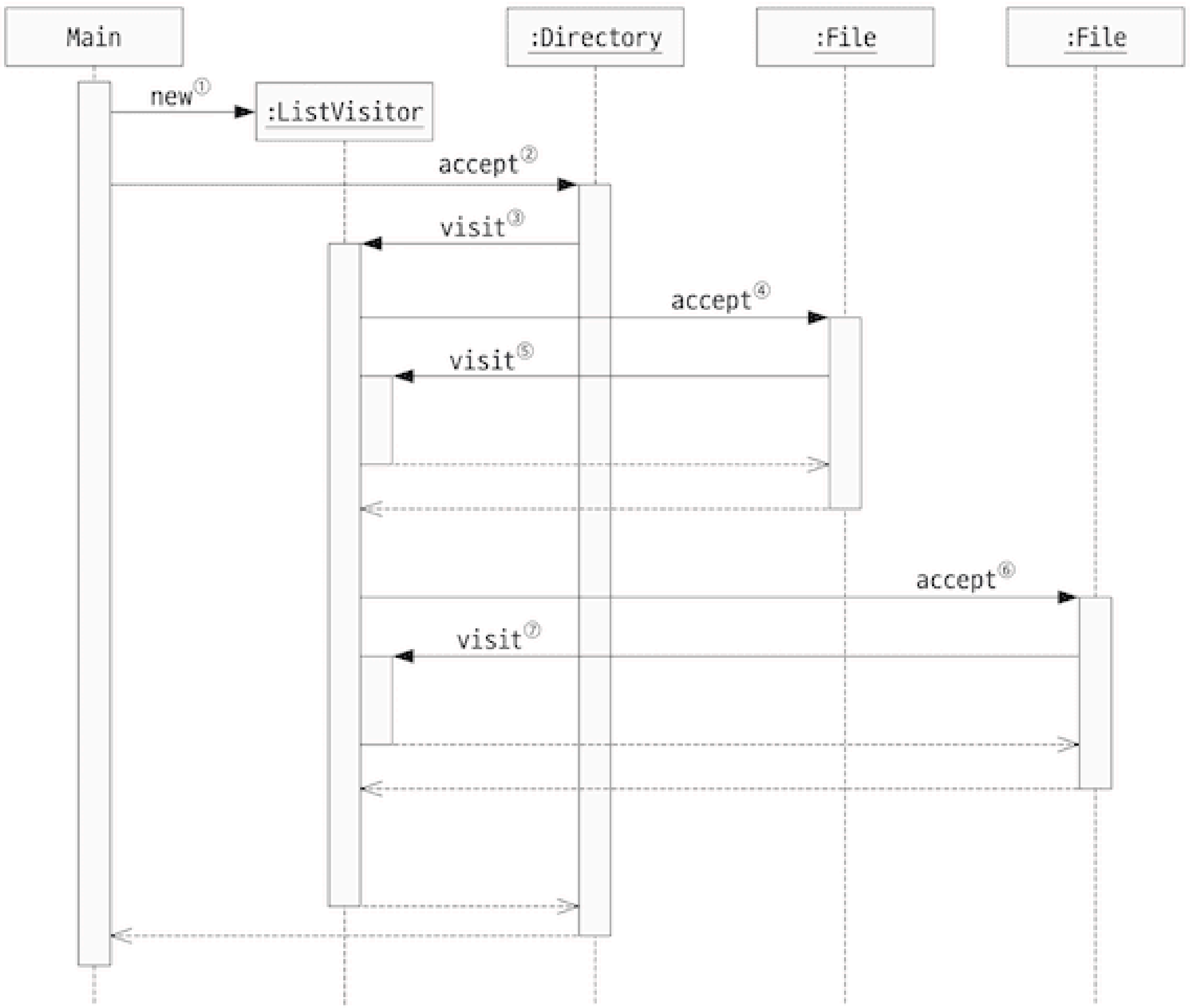
Discussion

Double Dispatch

- **Single dispatch**: Method chosen based on object type (`obj.method()`)
- **Double dispatch**: Method chosen based on *both* visitor and element types
- `element.accept(visitor)` → `visitor.visit_element_type(element)`

```
size_visitor = SizeVisitor()

file.accept(size_visitor)
# → size_visitor.visit_file(file)
directory.accept(size_visitor)
# → size_visitor.visit_directory(directory)
```

1. On the Visitor side, it has all the visit method for each element (file and directory).
2. On the Element side, it is ready to give its information (file and directory object) to any visitor.


Main
:Directory
:File
:File
new①
:ListVisitor
accept②
visit③
accept④
visit⑤
accept⑥
visit⑦

Java vs Python Double Dispatch

Java supports method overloading + dynamic dispatch (polymorphism).

```java
class SizeVisitor {
    void visit(File f) { ... }
    void visit(Directory d) { ... }
}
```

The compiler resolves which visit method to call based on the argument type at compile time.

```java
// based on type (file/directory),
// Java can invoke a different method
size_visitor.visit(file);
size_visitor.visit(directory);
```

Python does have polymorphism (method overriding works), but it does not support method overloading by argument type the way Java does.

```python
class SizeVisitor:
    def visit(self, file): ...
    def visit(self, directory):  # not work
```

So you need Duck Typing + naming convention (or manual dispatch inside one method):

```python
# size_visitor.visit(file) cannot work in Python
file.accept(size_visitor)
# => size_visitor.visit_file(file)
```

Key Benefits

1. **Open/Closed**: Add operations without modifying existing classes
2. **Single Responsibility**: Related operations grouped in one visitor
3. **Flexibility**: Different visitors can maintain different states
4. **Extensibility**: Easy to add new operations by creating new visitors

Key Drawbacks

1. **Breaking encapsulation**: Visitors may need access to element internals
2. **Difficult to add new element types**: All visitors must be updated
3. **Dependencies**: Visitors depend on concrete element interfaces
4. **Complexity**: More complex than adding methods directly

When to Use Visitor

- When you have **stable object structure** but **changing operations**
- When you want to perform **unrelated operations** on object hierarchy
- When you need to **gather related operations** in one place
- When operations would **clutter element classes** with unrelated functionality

When NOT to Use Visitor

- When **object structure changes frequently** (all visitors need updates)
- When operations are **closely related** to element data (better as methods)
- When you have **simple structures** with few operations
- When **performance** is critical (double dispatch has overhead)

Related Patterns

- **Composite**: Visitor is often used with Composite structures for tree traversal
- **Iterator**: Iterator accesses elements, Visitor processes them
- **Strategy**: Strategy changes algorithm within object, Visitor applies external algorithms

Visitor vs Adding Methods (Type vs Operations)

Adding Methods:

Operations (methods) are scattered across multiple classes.

✅ Easy to add new element type (Triangle Class)

❌ Hard to add new operation/method (must edit all classes)

```
class Shape(): # Interface
    def draw(self):pass; def area(self):pass

class Circle:
    def __init__(self, r): self.r = r
    def draw(self): print("Drawing Circle")
    def area(self): return 3.14 * self.r * self.r

class Square:
    def __init__(self, s): self.s = s
    def draw(self): print("Drawing Square")
    def area(self): return self.s * self.s
```

Visitor:

Operations are centralized in visitor classes.

```
# Elements use accept()
class Circle:
    def __init__(self, r): self.r = r
    def accept(self, visitor): visitor.visit_circle(self)

class Square:
    def __init__(self, s): self.s = s
    def accept(self, visitor): visitor.visit_square(self)
```

When an element is changed, we need to update all the visitors

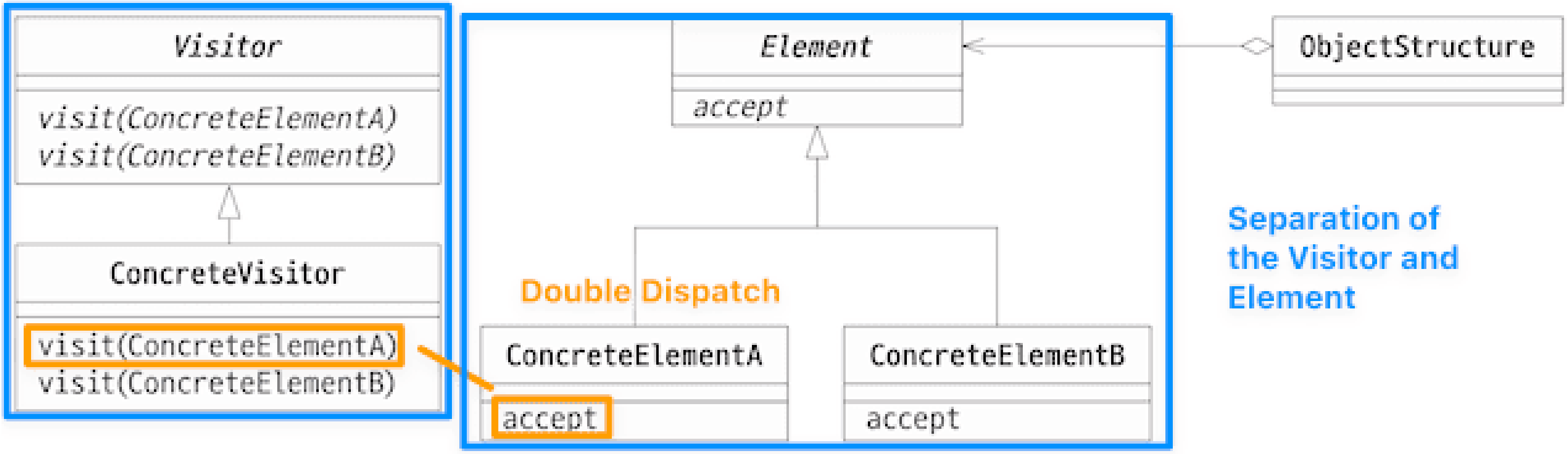
- ❌ Hard to add new element type (must edit all Visitors)

When we need to add a new visitor, we simply add a Visitor that supports the visit methods to the element.

- ✅ Easy to add new operations (new Visitor class)

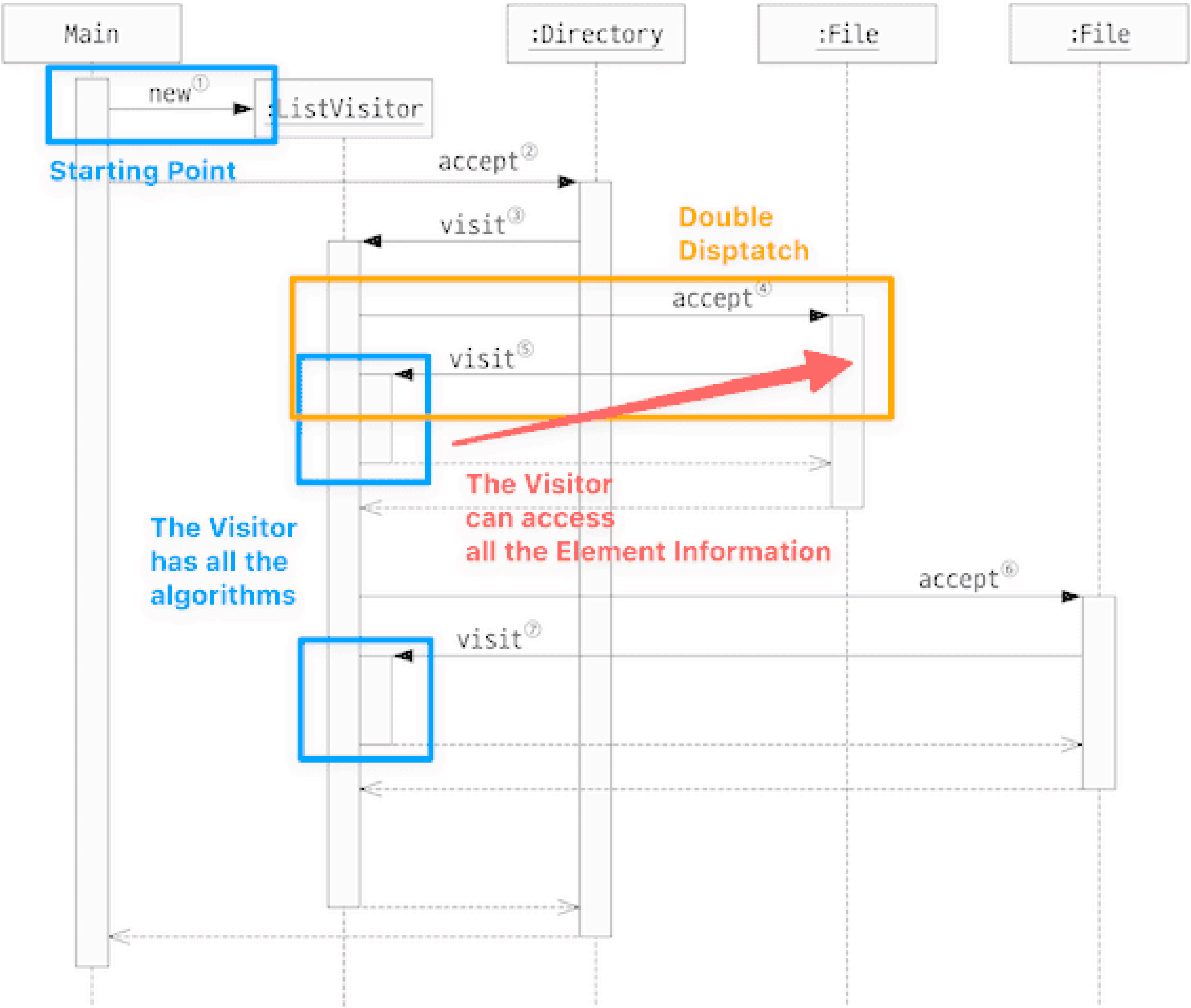
```python
# Visitor interface by convention
class ShapeVisitor:
    def visit_circle(self, circle): pass
    def visit_square(self, square): pass

class DrawVisitor(ShapeVisitor):
    def visit_circle(self, c): print("Drawing Circle")
    def visit_square(self, s): print("Drawing Square")
class AreaVisitor(ShapeVisitor):
    def visit_circle(self, c): print("Area =", 3.14 * c.r * c.r)
    def visit_square(self, s): print("Area =", s.s * s.s)
```

UML



Separation of the Visitor and Element

- Adding a new element is hard as all the visitors should be updated.
- Adding a new visitor operation is easy, as we simply add a new Visitor class.

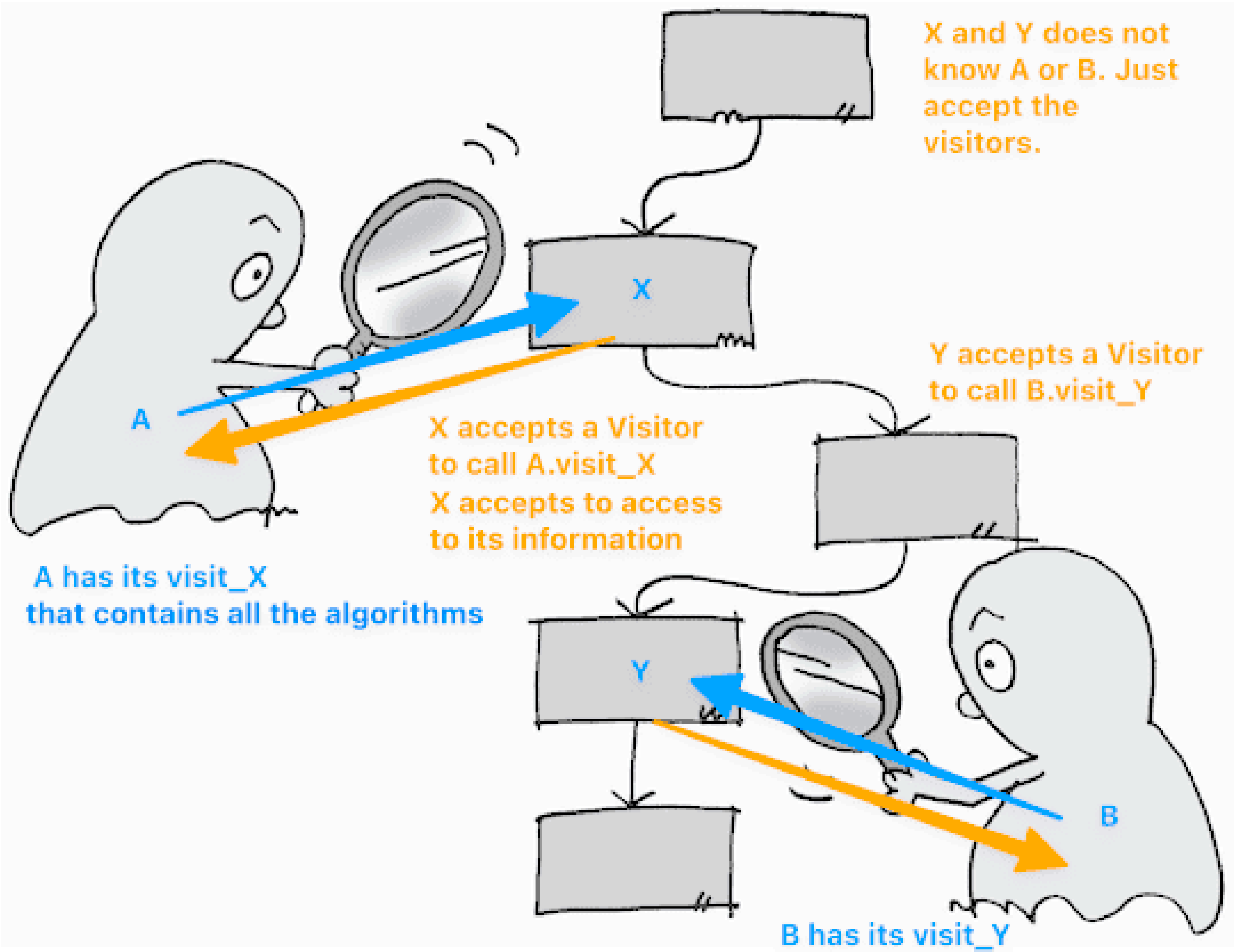


Starting Point

Double Disptatch

The Visitor has all the algorithms

The Visitor can access all the Element Information



X and Y does not
know A or B. Just
accept the
visitors.

Y accepts a Visitor
to call B.visit_Y

X accepts a Visitor
to call A.visit_X
X accepts to access
to its information

A has its visit_X
that contains all the algorithms

B has its visit_Y