In [9]:
```
from abc import abstractmethod
from math import pi
```

The Visitor Pattern

Introduction

The Visitor Pattern is a behavioral design pattern.
It allows us to separate algorithms from the objects on which they operate.

Key idea:
Instead of putting logic inside the object itself, we "visit" the object with a special visitor class.

In [ ]:
```
class NaiveSquare:
    def __init__(self, side):
        self.side = side

    def area(self):
        pass

    def perimeter(self):
        pass

class NaiveCircle:
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        pass

    def perimeter(self):
        pass
```

Problem

Imagine you have a structure of different elements like `Circle` or `Square` and you want to perform different operations on them.

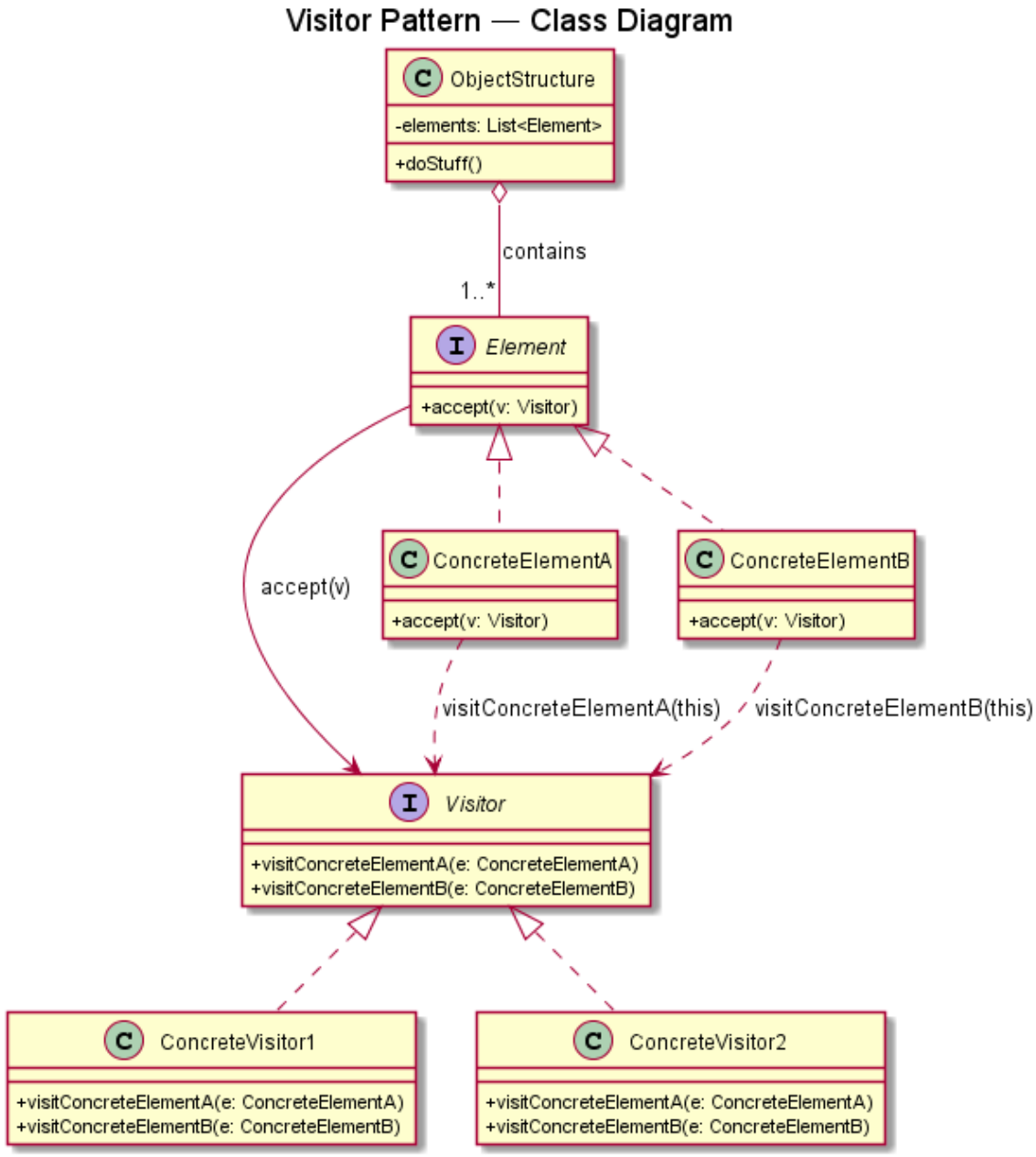
Naive approach:
Add methods for each operation inside every class.
→ This leads to bloat and violates the **Single Responsibility Principle**.

Visitor Pattern solution:

We define a `Visitor` that encapsulates these operations.

Structure of the Visitor Pattern:

- **Element**: Defines an `accept(visitor)` method.
- **Concrete Element**: Implements `accept` and calls the appropriate visitor method.
- **Visitor**: Declares visit methods for each element type.
- **Concrete Visitor**: Implements specific behavior for each element type.



In [10]:
```python
class Visitor:

    @abstractmethod
    def visit_square(self, element):
        pass
```

```
    @abstractmethod
    def visit_circle(self, element):
        pass


class AreaVisitor(Visitor):

    def visit_square(self, element):
        return element.side ** 2

    def visit_circle(self, element):
        return pi * element.radius ** 2


class PerimeterVisitor(Visitor):

    def visit_square(self, element):
        return element.side * 4

    def visit_circle(self, element):
        return 2 * pi * element.radius
```

In [11]:
```
class GeometricShape:
    @abstractmethod
    def accept(self, visitor: Visitor):
        pass


class Square(GeometricShape):
    def __init__(self, side):
        self.side = side

    def accept(self, visitor: Visitor):
        return visitor.visit_square(self)


class Circle(GeometricShape):
    def __init__(self, radius):
        self.radius = radius

    def accept(self, visitor: Visitor):
        return visitor.visit_circle(self)
```

Example Run

In [12]:
```
shape_elements = [Square(4), Circle(3)]

area_visitor = AreaVisitor()
perimeter_visitor = PerimeterVisitor()

for shape in shape_elements:
    print(f"Area: {shape.accept(area_visitor)}")
    print(f"Perimeter: {shape.accept(perimeter_visitor)}")
```

```
Area: 16
Perimeter: 16
Area: 28.274333882308138
Perimeter: 18.84955592153876
```

Benefits

- Separates algorithms from object structure
- Easy to add new operations without changing element classes

Drawbacks

- Harder to add new element types as all visitors must be updated
- Visitors might lack access to certain private fields and methods

---

Real-Life Example: File System Operations

Two element types:

- `File`
- `Directory`

We want to perform different operations on them:

- Calculate total size
- Generate a text report

Instead of embedding all these operations inside `File` and `Directory`, we use the Visitor Pattern.

In [13]:
```python
class File:
    def __init__(self, name, size):
        self.name = name
        self.size = size

    def accept(self, visitor):
        visitor.visit_file(self)

class Directory:
    def __init__(self, name, children=None):
        self.name = name
        self.children = children if children else []

    def accept(self, visitor):
        visitor.visit_directory(self)
```

The Visitor Interface

Each visitor knows how to "handle" both Files and Directories.

In [14]:
```
class FileSystemVisitor:
    def visit_file(self, file):
        pass

    def visit_directory(self, directory):
        pass


class SizeVisitor(FileSystemVisitor):
    def __init__(self):
        self.total_size = 0

    def visit_file(self, file):
        self.total_size += file.size

    def visit_directory(self, directory):
        for child in directory.children:
            child.accept(self)


class ReportVisitor(FileSystemVisitor):
    def __init__(self, indent=0):
        self.indent = indent

    def visit_file(self, file):
        print("  " * self.indent + f"File: {file.name} ({file.size} KB)")

    def visit_directory(self, directory):
        print("  " * self.indent + f"Directory: {directory.name}")
        for child in directory.children:
            child.accept(ReportVisitor(self.indent + 1))
```

Example Run

In [15]:
```
root = Directory("root", [
    File("readme.txt", 5),
    Directory("images", [
        File("logo.png", 150),
        File("banner.jpg", 300)
    ])
])

size_visitor = SizeVisitor()
root.accept(size_visitor)
print("Total size:", size_visitor.total_size, "KB")
print("\nFile system report:")
root.accept(ReportVisitor())
```

```
Total size: 455 KB

File system report:
Directory: root
  File: readme.txt (5 KB)
  Directory: images
    File: logo.png (150 KB)
    File: banner.jpg (300 KB)
```

Summary

Use the Visitor Pattern:

- when you need to perform an operation on all elements with different logics
- to clean up auxiliary behavior in business logic