04834580 Software Engineering (Honor Track) 2024-25

Visitor

Sergey Mechtaev
mechtaev@pku.edu.cn
School of Computer Science, Peking University

Definition (Visitor [1])

Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Visitor supports the Open-Close principle by applying double dispatch.

Definition (Double Dispatch)

A mechanism that dispatches a function call to different concrete functions depending on the runtime types of two objects involved in the call.

Double Dispatch: Motivation

This code violates the Open-Closed Principle:

```
class Asteroid:
    def collide_with(self, other):
        if isinstance(other, Asteroid):
            print("Asteroid bounces off another Asteroid.")
        elif isinstance(other, SpaceShip):
            print("Asteroid damages the SpaceShip.")
        else:
            print("Unknown collision.")

class SpaceShip:
    def collide_with(self, other):
        if isinstance(other, Asteroid):
            print("SpaceShip takes damage from an Asteroid.")
        elif isinstance(other, SpaceShip):
            print("SpaceShip avoids collision with another SpaceShip.")
        else:
            print("Unknown collision.")

# Different collisions
a = Asteroid()
s = SpaceShip()

a.collide_with(s)
s.collide_with(a)
```

Double Dispatch via Meta-Programming

北京大学

```
class GameObject:
    def collide_with(self, other):
        method_name = f'collide_with_{self.__class__.__name__.lower()}'
        if hasattr(other, method_name):
            return getattr(other, method_name)(self)
        else:
            print("Unknown collision.")

class Asteroid(GameObject):
    def collide_with_asteroid(self, other):
        print("Asteroid bounces off another Asteroid.")
    def collide_with_spaceship(self, other):
        print("Asteroid damages the SpaceShip.")

class SpaceShip(GameObject):
    def collide_with_asteroid(self, other):
        print("SpaceShip takes damage from an Asteroid.")
    def collide_with_spaceship(self, other):
        print("SpaceShip avoids collision with another SpaceShip.")

a = Asteroid()
s = SpaceShip()

a.collide_with(s)
s.collide_with(a)
a.collide_with(a)
s.collide_with(s)
```

Double Dispatch: Without Meta-Programming

北京大学

```
class Asteroid:
    def collide_with(self, other):
        other.collide_with_asteroid(self)

    def collide_with_asteroid(self, other):
        print("Asteroid bounces off another Asteroid.")

    def collide_with_spaceship(self, other):
        print("Asteroid damages the SpaceShip.")

class SpaceShip:
    def collide_with(self, other):
        other.collide_with_spaceship(self)

    def collide_with_asteroid(self, other):
        print("SpaceShip takes damage from an Asteroid.")

    def collide_with_spaceship(self, other):
        print("SpaceShip avoids collision with another SpaceShip.")

a = Asteroid()
s = SpaceShip()

a.collide_with(s)
s.collide_with(a)
a.collide_with(a)
s.collide_with(s)
```

Visitor Motivation

How to add another method like `eval`, e.g. `print`, without modifying the implementation?

```
class Expr:
    def eval(self):
        raise NotImplementedError("Subclasses must implement evaluate")

class Add(Expr):
    def __init__(self, left, right):
        self.left = left
        self.right = right

    def evaluate(self):
        return self.left.eval() + self.right.eval()

class Sub(Expr):
    def __init__(self, left, right):
        self.left = left
        self.right = right

    def evaluate(self):
        return self.left.eval() - self.right.eval()
```

This code violates the Open-Closed Principle:

```python
# External print function
def print_expr(expr):
    if isinstance(expr, Num):
        return str(expr.value)
    elif isinstance(expr, Add):
        return f"({print_expr(expr.left)} + {print_expr(expr.right)})"
    elif isinstance(expr, Sub):
        return f"({print_expr(expr.left)} - {print_expr(expr.right)})"
    else:
        raise ValueError("Unsupported expression")
```

Visitor Implementation

北京大学

Implement the Visitor pattern using double dispatch:

```
class Expr:
    def accept(self, visitor):
        raise NotImplementedError("Subclasses must implement accept")

class Add(Expr):
    def __init__(self, left, right):
        self.left = left
        self.right = right

    def accept(self, visitor):
        return visitor.visit_add(self)

class Sub(Expr):
    def __init__(self, left, right):
        self.left = left
        self.right = right

    def accept(self, visitor):
        return visitor.visit_sub(self)
```

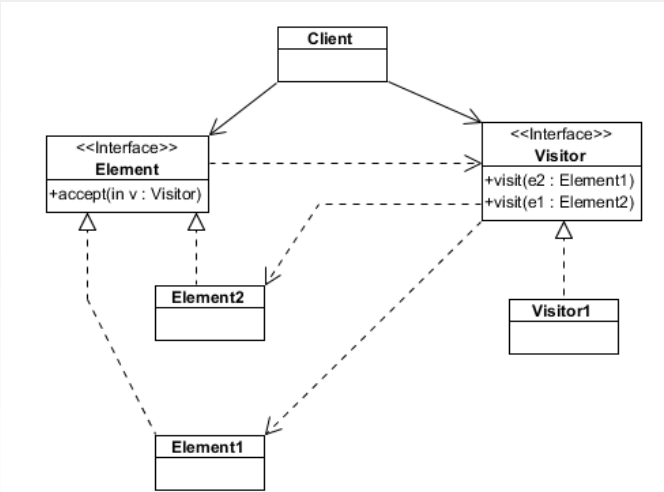
Visitor Implementation

北京大学

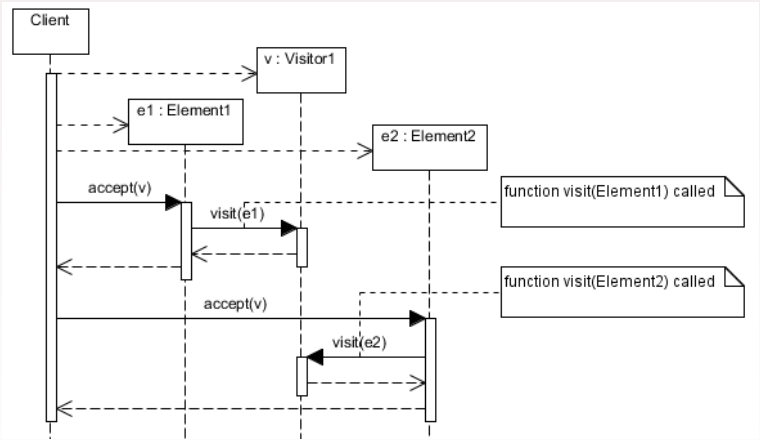
print and eval extracted as visitors:

```
class Visitor:
    def visit_add(self, add_expr):
        raise NotImplementedError("Subclasses must implement visit_add")

    def visit_sub(self, sub_expr):
        raise NotImplementedError("Subclasses must implement visit_sub")

class EvalVisitor(Visitor):
    def visit_add(self, add_expr):
        return add_expr.left.accept(self) + add_expr.right.accept(self)

    def visit_sub(self, sub_expr):
        return sub_expr.left.accept(self) - sub_expr.right.accept(self)

class PrintVisitor(Visitor):
    def visit_add(self, add_expr):
        return f"({add_expr.left.accept(self)} + {add_expr.right.accept(self)})"

    def visit_sub(self, sub_expr):
        return f"({add_expr.left.accept(self)} - {add_expr.right.accept(self)})"
```

Visitor Class Diagram



Visitor Sequence Diagram



References I

[1] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Deutschland GmbH, 1995.