Visitor Pattern

Tomás Felipe Melli

June 24, 2025

Índice

1 Intent

Representar una operación que se realiza sobre los elementos de una estructura de objetos. El patrón Visitor permite definir nuevas operaciones sin modificar las clases sobre las que opera.

2 Motivation

Un compilador representa programas como árboles de sintaxis abstracta (AST) y debe realizar muchas operaciones sobre ellos: análisis semánticos, generación de código, optimizaciones, impresión, métricas, etc. Estas operaciones tratan distintos tipos de nodos (asignaciones, variables, expresiones) de forma diferente, por lo que cada tipo de nodo es una clase distinta.

Distribuir todas estas operaciones dentro de las clases de los nodos hace el sistema difícil de mantener, entender y extender, ya que el código de operaciones muy distintas (tipo-checking, optimización, impresión) queda mezclado. Además, agregar una nueva operación implica modificar y recompilar las clases de nodos.

La solución es encapsular cada operación en un objeto separado llamado visitor y hacer que los nodos acepten un visitor. Al aceptar el visitor, el nodo llama al método específico del visitor para su tipo (por ejemplo, VisitAssignment para nodos de asignación). Esto separa la estructura de los nodos de las operaciones que se les aplican.

Con el patrón Visitor, se mantienen dos jerarquías:

- La jerarquía de elementos (nodos del AST).
- La jerarquía de visitors (operaciones sobre nodos).

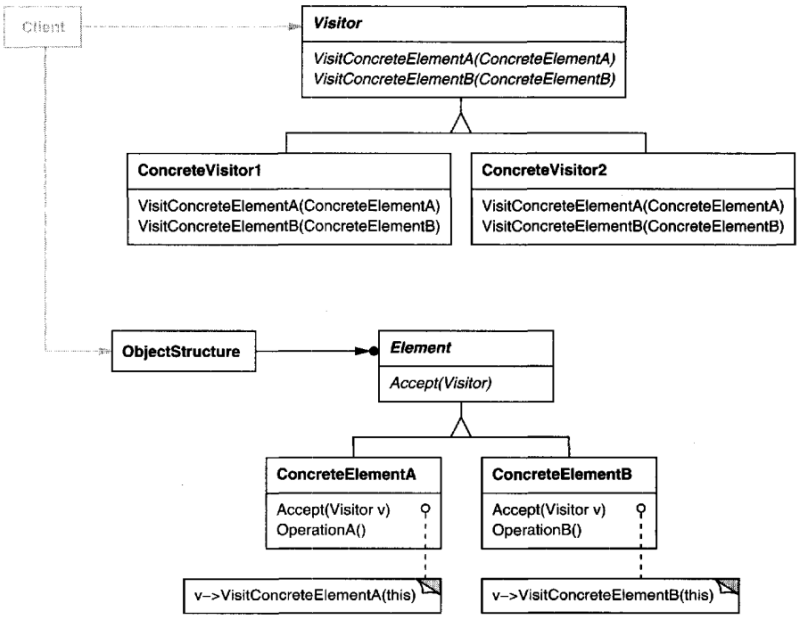
Así, para agregar una nueva operación, sólo se define un nuevo visitor, sin modificar las clases de los nodos, siempre que la estructura del AST no cambie. Esto facilita extender y mantener el compilador.

3 Applicability

Usar Visitor cuando:

- La estructura de objetos contiene muchas clases con interfaces diferentes y se quieren realizar operaciones que dependen de la clase concreta de cada objeto.
- Se necesitan muchas operaciones distintas y no relacionadas sobre los objetos, y se quiere evitar mezclar esas operaciones dentro de las clases de los objetos (evitar "contaminarlas"). Visitor agrupa operaciones relacionadas en una sola clase.
- Las clases de la estructura rara vez cambian, pero se suelen agregar nuevas operaciones. Cambiar la estructura implica actualizar la interfaz de todos los visitors, lo que es costoso. Si la estructura cambia frecuentemente, es mejor definir las operaciones directamente en las clases.

4 Structure



5 Participants

- Visitor (NodeVisitor) : Declara una operación Visit para cada clase concreta de elementos en la estructura. El nombre y firma de la operación identifican la clase concreta que envía la solicitud, permitiendo al visitor acceder directamente a la interfaz específica del elemento.
- Concrete Visitor (TypeCheckingVisitor) : Implementa las operaciones declaradas en Visitor. Cada operación contiene la lógica específica para el tipo de elemento visitado, manteniendo el contexto y estado local (por ejemplo, resultados acumulados durante el recorrido).
- Element (Node) : Define un método Accept que recibe un visitor como argumento.
- Concrete Element (AssignmentNode, VariableRefNode) : Implementan el método Accept que llama al visitor correspondiente.
- Object Structure (Program) : Puede enumerar sus elementos y proveer una interfaz para que el visitor los recorra. Puede ser una estructura compuesta (composite) o una colección (lista, conjunto, etc.).

6 Example

Contexto del problema

Tenemos una jerarquía de clases de equipamiento (Equipment) que pueden ser simples (disco floppy, tarjeta) o compuestos (chasis, bus) que contienen otros equipos. Queremos definir operaciones como calcular el costo total o hacer inventario, pero sin mezclar esas operaciones dentro de las clases de equipo.

Aquí entra Visitor: separa las operaciones en clases visitantes (EquipmentVisitor) y permite extender operaciones sin cambiar la jerarquía de equipos.

Clase Base (Equipment)

```
class Equipment {
public:
    virtual ~Equipment();
    const char* Name() { return _name; }

    virtual Watt Power();
    virtual Currency NetPrice();
    virtual Currency DiscountPrice();

    // M todo clave para Visitor: acepta un visitante
    virtual void Accept(EquipmentVisitor& visitor);

protected:
    Equipment(const char* name) : _name(name) {}

private:
    const char* _name;
};
```

Clase Visitor (EquipmentVisitor)

```
class EquipmentVisitor {
public:
    virtual ~EquipmentVisitor();

    // M todos para cada tipo concreto de Equipment
    virtual void VisitFloppyDisk(FloppyDisk* e) { }
    virtual void VisitCard(Card* e) { }
    virtual void VisitChassis(Chassis* e) { }
    virtual void VisitBus(Bus* e) { }

protected:
    EquipmentVisitor() = default;
};
```

Implementación de Accept en Equipos concretos

```
void FloppyDisk::Accept(EquipmentVisitor& visitor) {
    visitor.VisitFloppyDisk(this);
}
```

```
void Chassis::Accept(EquipmentVisitor& visitor) {
    for (auto i = _parts.begin(); i != _parts.end(); ++i) {
        (*i)->Accept(visitor);
    }
    visitor.VisitChassis(this);
}
```

Visitor Concreto 1: PricingVisitor (Cálculo de costos)

```
class PricingVisitor : public EquipmentVisitor {
public:
    PricingVisitor() : _total(0) {}

    Currency& GetTotalPrice() { return _total; }

    void VisitFloppyDisk(FloppyDisk* e) override {
        _total += e->NetPrice();
    }

    void VisitChassis(Chassis* e) override {
        _total += e->DiscountPrice();
    }

    // Otros visitadores...

private:
    Currency _total;
};
```

Visitor Concreto 2: InventoryVisitor (Inventario)

```
class InventoryVisitor : public EquipmentVisitor {
public:
    InventoryVisitor() {}

    Inventory GetInventory() { return _inventory; }

    void VisitFloppyDisk(FloppyDisk* e) override {
        _inventory.Accumulate(e);
    }

    void VisitChassis(Chassis* e) override {
        _inventory.Accumulate(e);
    }

    // Otros visitadores...

private:
    Inventory _inventory;
};
```

Resultado

Uso del patrón

```
Equipment* equipment = /* alg n equipo o estructura compuesta */;
PricingVisitor pricingVisitor;

equipment->Accept(pricingVisitor);

std::cout << "Costo total del equipo " << equipment->Name()
          << " es " << pricingVisitor.GetTotalPrice() << std::endl;

```

```
9  InventoryVisitor inventoryVisitor;
10 equipment->Accept(inventoryVisitor);
11
12 std::cout << "Inventario de " << equipment->Name() << ": "
13           << inventoryVisitor.GetInventory().Summary() << std::endl;
```