# Visitor 패턴

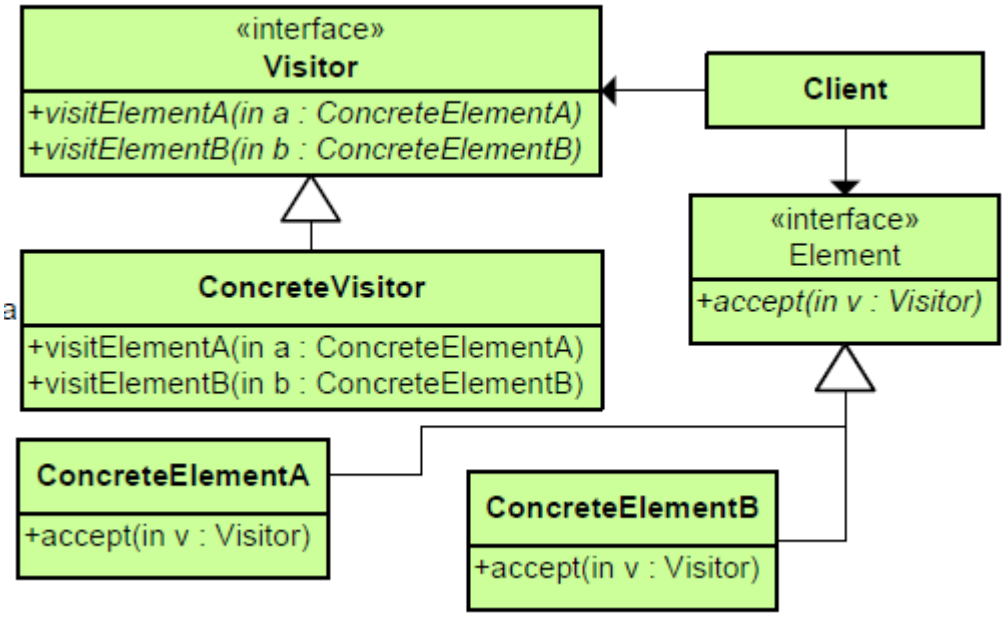
## 타입

Behavioral Pattern

## 문제

## 해결

## 클래스 다이어그램




## 예제

visitor.cpp

```
#include <string>
#include <iostream>
#include <vector>

using namespace std;

class Wheel;
class Engine;
class Body;
```

```
class Car;

// interface to all car 'parts'
struct CarElementVisitor
{
    virtual void visit(Wheel& wheel) const = 0;
    virtual void visit(Engine& engine) const = 0;
    virtual void visit(Body& body) const = 0;

    virtual void visitCar(Car& car) const = 0;
    virtual ~CarElementVisitor() {}
};

// interface to one part
struct CarElement
{
    virtual void accept(const CarElementVisitor& visitor) = 0;
    virtual ~CarElement() {}
};

// wheel element, there are four wheels with unique names
class Wheel : public CarElement
{
public:
    explicit Wheel(const string& name) :
    name_(name)
    {
    }
    const string& getName() const
    {
        return name_;
    }
    void accept(const CarElementVisitor& visitor)
    {
        visitor.visit(*this);
    }
private:
    string name_;
};

// engine
class Engine : public CarElement
{
public:
    void accept(const CarElementVisitor& visitor)
    {
        visitor.visit(*this);
    }
};

// body
```

```
class Body : public CarElement
{
public:
    void accept(const CarElementVisitor& visitor)
    {
        visitor.visit(*this);
    }
};

// car, all car elements(parts) together
class Car
{
public:
    vector<CarElement*>& getElements()
    {
        return elements_;
    }
    Car()
    {
        // assume that neither push_back nor Wheel(const string&) may
throw
        elements_.push_back( new Wheel("front left") );
        elements_.push_back( new Wheel("front right") );
        elements_.push_back( new Wheel("back left") );
        elements_.push_back( new Wheel("back right") );
        elements_.push_back( new Body() );
        elements_.push_back( new Engine() );
    }
    ~Car()
    {
        for(vector<CarElement*>::iterator it = elements_.begin();
            it != elements_.end(); ++it)
        {
            delete *it;
        }
    }
private:
    vector<CarElement*> elements_;
};

// PrintVisitor and DoVisitor show by using a different implementation
the Car class is unchanged
// even though the algorithm is different in PrintVisitor and
DoVisitor.
class CarElementPrintVisitor : public CarElementVisitor
{
public:
    void visit(Wheel& wheel) const
    {
        cout << "Visiting " << wheel.getName() << " wheel" << endl;
    }
```

```
    void visit(Engine& engine) const
    {
        cout << "Visiting engine" << endl;
    }
    void visit(Body& body) const
    {
        cout << "Visiting body" << endl;
    }
    void visitCar(Car& car) const
    {
        cout << endl << "Visiting car" << endl;
        vector<CarElement*>& elems = car.getElements();
        for(vector<CarElement*>::iterator it = elems.begin();
            it != elems.end(); ++it )
        {
            (*it)->accept(*this);    // this issues the callback i.e.
to this from the element
        }
        cout << "Visited car" << endl;
    }
};

class CarElementDoVisitor : public CarElementVisitor
{
public:
    // these are specific implementations added to the original object
without modifying the original struct
    void visit(Wheel& wheel) const
    {
        cout << "Kicking my " << wheel.getName() << " wheel" << endl;
    }
    void visit(Engine& engine) const
    {
        cout << "Starting my engine" << endl;
    }
    void visit(Body& body) const
    {
        cout << "Moving my body" << endl;
    }
    void visitCar(Car& car) const
    {
        cout << endl << "Starting my car" << endl;
        vector<CarElement*>& elems = car.getElements();
        for(vector<CarElement*>::iterator it = elems.begin();
            it != elems.end(); ++it )
        {
            (*it)->accept(*this);    // this issues the callback i.e.
to this from the element
        }
        cout << "Stopped car" << endl;
    }
```

```
};

int main()
{
    Car car;
    CarElementPrintVisitor printVisitor;
    CarElementDoVisitor doVisitor;

    printVisitor.visitCar(car);
    doVisitor.visitCar(car);

    return 0;
}
```

# 참고

http://en.wikibooks.org/wiki/C%2B%2B_Programming/Code/Design_Patterns#Visitor