Visitor Design Pattern

Purpose and Motivation

The purpose of the visitor pattern is to add new operations to object structure but without changing the object structure. It's very tempting to add new computations inside the object classes, which works fine if the program is simple or small. However, as the program grows more complex, the program will be hard to maintain if keep adding new methods to the original code. It's more error-prone. Visitor pattern solves this by separating the object structure and algorithms. New operations are added by creating new classes in the algorithm section. This pattern follows the open-close principle, meaning that the API is open for extensibility, but implementation close for modification.

Intended Use Cases

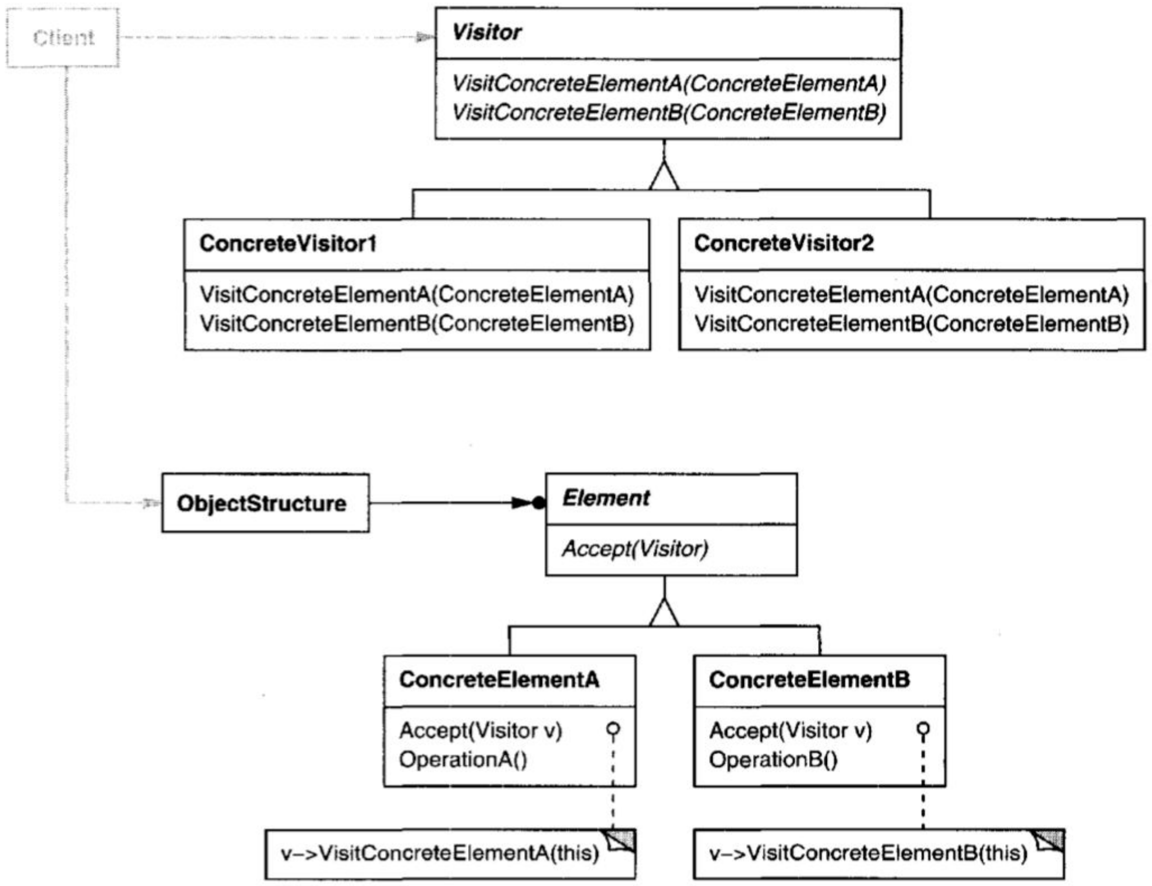
The Visitor design pattern is useful when many distinct and unrelated operations need to be performed on objects in an object structure, and you want to avoid "polluting" their classes with these operations. The Visitor design pattern lets you keep related operations together by defining them in one class. When object structure is shared by many applications, use visitor to put operations in just those applications that need them. In addition, this design pattern is used when you do not want your object structure to change much, but you want to keep expanding the operations performed on the objects.

Vocabulary

Visitor (Interface): Declares a "visit" operation for each class of concrete element in the object structure. The operation's name and signature identify the class that sends the Visit request to the visitor. That lets the visitor determine the concrete class of the element being visited. Then the visitor can access the element directly through its particular interface.
Concrete Visitor (Class): Implements each operation declared by Visitor. A Concrete Visitor provides the context for the algorithm and stores its local state. This state often accumulates results during the traversal of the structure.
Visitable (Interface): Defines an "accept" operation that takes a visitor as an argument.
Concrete Element (Class): Implement an "accept" operation that takes a visitor as an argument.

Structure and Runtime Behaviour

There are two parts: Visitor (Algorithm) and Visitable (Element), interacting in the Figure below. We have two interfaces for each part, and several concrete classes implementing the interfaces. The Client calls operation in Visitor through the method Accept(). The Visitable interface declares the Accept() method which takes in a Visitor. The Concrete Element implement the Accept() function: it uses the Visitor parameter; calls VisitConcreteElement() method and pass itself to the VisitConcreteElement() function. The "this" keyword allows the appropriate VisitConcreteElement() method to be called within the Visitor class at runtime. Notice that despite the Accept() method is exactly the same for each class, it is absolutely necessary to override it every time. Otherwise, if your class is inherited from a parent class, "this" will refer to the parent class's context. Later, the Visitor interface declares VisitConcreteElement() methods for different types of elements. Each Visitor class will implement the methods for all visitable types. If you want to add a new operation, just add another ConcreteVisitor class and implement methods for all visitable types.


Client
Visitor
VisitConcreteElementA(ConcreteElementA)
VisitConcreteElementB(ConcreteElementB)
ConcreteVisitor1
VisitConcreteElementA(ConcreteElementA)
VisitConcreteElementB(ConcreteElementB)
ConcreteVisitor2
VisitConcreteElementA(ConcreteElementA)
VisitConcreteElementB(ConcreteElementB)
ObjectStructure
Element
Accept(Visitor)
ConcreteElementA
Accept(Visitor v)
OperationA()
ConcreteElementB
Accept(Visitor v)
OperationB()
v->VisitConcreteElementA(this)
v->VisitConcreteElementB(this)

Known Consequences

Positive Consequences

- If the logic of operations in Element has to change i.e. we want to handle the Element differently, we only need to modify code implemented in the Visit() methods of the concrete Visitor classes.
- Visitors can have and maintain state relating to the different Elements.
- Adding new Elements does not affect current functionality implemented for the other Elements. This ensures that the Visitor pattern does not violate the Open/Closed principle.

Negative Consequences

- The return type of the VisitConcreteElement() methods must be known prior to implementation, in the designing phase, otherwise the signatures in the interface and all of the implemented VisitConcreteElement() methods will have to be changed to adhere to the new return type.
- As we increase the number of Visitor implementations, it makes the system difficult to maintain and extend. If a new Element is added, all the Visitors need to implement its VisitConcreteElement() method, even if that particular implementation is just an empty function.
- A Visitor has access to the Element object and can, therefore, modify its properties, which may result in unwanted side effects.

NFPs

Improved NFPS

- Scalability: Improves scalability in the case when new operations on Element classes are added frequently, and the program's class hierarchy consists of many unrelated classes.
- Readability: Improves readability as the code for the new operation is packed up in a Visitor class and not cluttered throughout the many Element classes.

Inhibited NFPS

- Maintainability: Degrades maintainability in the case when we need to add new Elements to the program's class hierarchy. When a new Element is added, all existing Visitors must be updated with the new corresponding VisitConcreteElement() method for processing this Element.

- Testability: Degrades testability due to the extensive use of polymorphism and the fact that its implementation is based on double dispatch.

Code Example

The Visitor class would be declared like this in C++:

```
class Visitor {
public:
    virtual void VisitElementA(ElementA*);
    virtual void VisitElementB(ElementB*);

    // and so on for other concrete elements
protected:
    Visitor();
};
```

Each class of ConcreteElement implements an Accept() operation that calls the matching VisitElement() operation on the visitor for that ConcreteElement. Thus the operation that ends up getting called depends on both the class of the element and the class of the visitor.
The concrete elements are declared as

```
class Element {
public:
    virtual ~Element();
    virtual void Accept(Visitor&) = 0;
protected:
    Element();
};
class ElementA : public Element {
public:
    ElementA();
    virtual void Accept(Visitor& v) { v.VisitElementA(this); }
};

class ElementB : public Element {
public:
    ElementB();
    virtual void Accept(Visitor& v) { v.VisitElementB(this); }
};
```

A CompositeElement class might implement Accept() like this:

```
class CompositeElement : public Element {
public:
    virtual void Accept(Visitor&);
private:
    List<Element*>* _children;
};

void CompositeElement::Accept (Visitor& v) {
    ListIterator<Element*> i(_children);

    for (i.First(); !i.IsDone(); i.Next()) {
        i.CurrentItem()->Accept(v);
    }
    v.VisitCompositeElement(this);
}
```

Credits

Students:

DDJC (Jennifer HGill, Di Wang, Chenshu Zhou)

Kaze (Zeyad Abdulhani, Karan Bhandari, Eric Luo, Adil Mian)

Instructor and TAs:

Mei Nagappan, Arman Naeimian, Ivens Portugal