Visitor

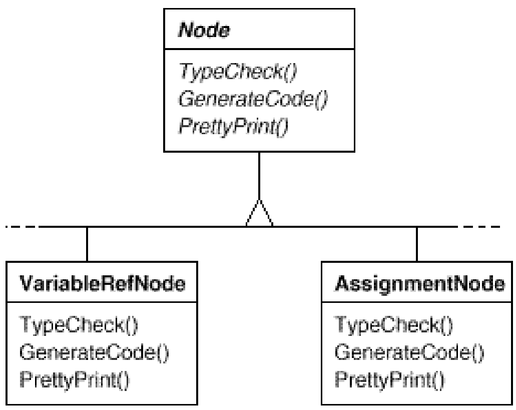
Intent

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

Motivation

Consider a compiler that represents programs as abstract syntax trees.It will need to perform operations on abstract syntax trees for "staticsemantic" analyses like checking that all variables are defined. Itwill also need to generate code. So it might define operations fortype-checking, code optimization, flow analysis, checking for variablesbeing assigned values before they're used, and so on. Moreover, we coulduse the abstract syntax trees for pretty-printing, programrestructuring, code instrumentation, and computing various metrics of aprogram.

Most of these operations will need to treat nodes that representassignment statements differently from nodes that represent variables orarithmetic expressions. Hence there will be one class for assignmentstatements, another for variable accesses, another for arithmeticexpressions, and so on. The set of node classes depends on the languagebeing compiled, of course, but it doesn't change much for a givenlanguage.

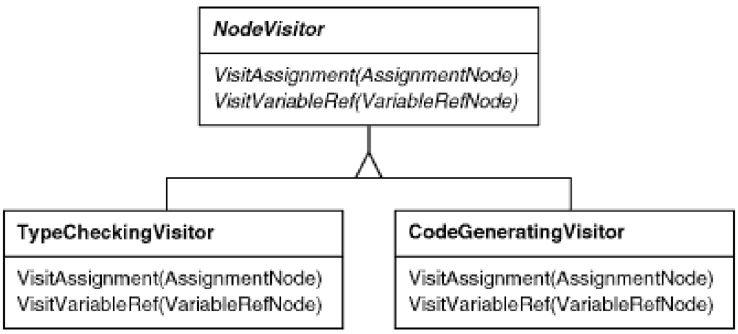


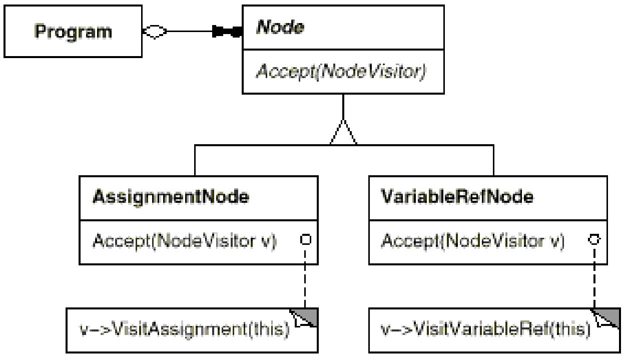
This diagram shows part of the Node class hierarchy. The problem hereis that distributing all these operations across the various nodeclasses leads to a system that's hard to understand, maintain, andchange. It will be confusing to have type-checking code mixed withpretty-printing code or flow analysis code. Moreover, adding a newoperation usually requires recompiling all of these classes. It would bebetter if each new operation could be added separately, and the nodeclasses were independent of the operations that apply to them.

We can have both by packaging related operations from each class in aseparate object, called a **visitor**, and passing it toelements of the abstract syntax tree as it's traversed. When an element"accepts" the visitor, it sends a request to the visitor that encodesthe element's class. It also includes the element as an argument. Thevisitor will then execute the operation for that element—theoperation that used to be in the class of the element.

For example, a compiler that didn't use visitors might type-check aprocedure by calling the TypeCheck operation on its abstract syntaxtree. Each of the nodes would implement TypeCheck by calling TypeCheckon its components (see the preceding class diagram). If the compilertype-checked a procedure using visitors, then it would create aTypeCheckingVisitor object and call the Accept operation on theabstract syntax tree with that object as an argument. Each of thenodes would implement Accept by calling back on the visitor: anassignment node calls VisitAssignment operation on the visitor, whilea variable reference calls VisitVariableReference. What used to be theTypeCheck operation in class AssignmentNode is now the VisitAssignmentoperation on TypeCheckingVisitor.

To make visitors work for more than just type-checking, we need anabstract parent class NodeVisitor for all visitors of an abstract syntaxtree. NodeVisitor must declare an operation for each node class. Anapplication that needs to compute program metrics will define newsubclasses of NodeVisitor and will no longer need to add application-specific code to the node classes. The Visitor pattern encapsulates the operations for each compilation phase in a Visitor associated with that phase.





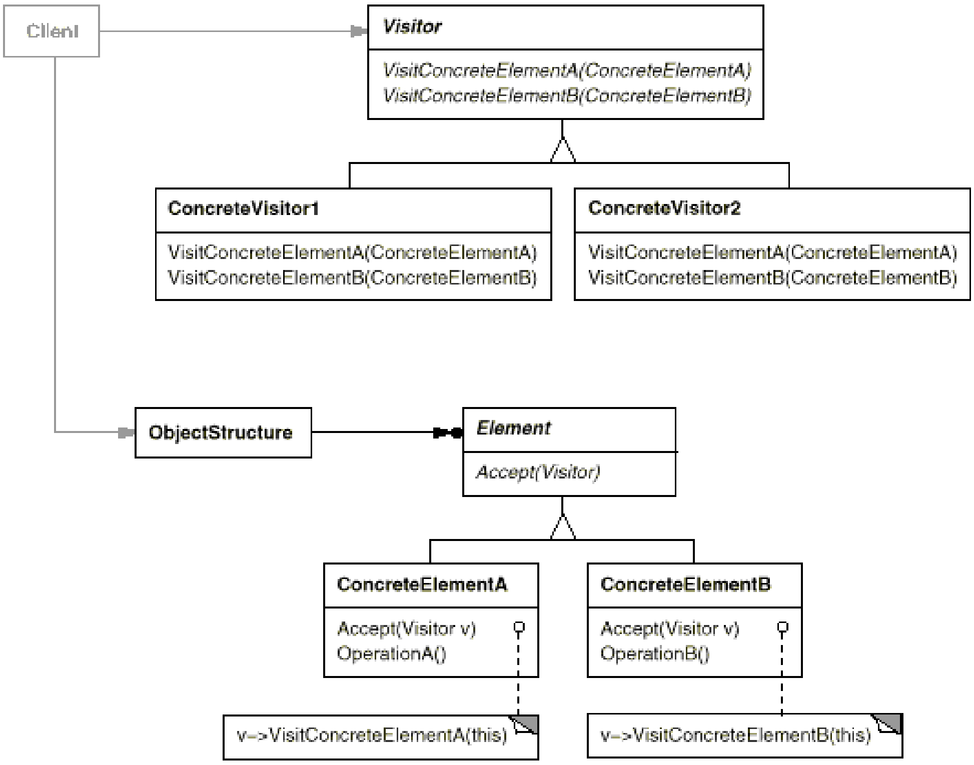
With the Visitor pattern, you define two class hierarchies: one for theelements being operated on (the Node hierarchy) and one for the visitorsthat define operations on the elements (the NodeVisitor hierarchy). Youcreate a new operation by adding a new subclass to the visitor classhierarchy. As long as the grammar that the compiler accepts doesn'tchange (that is, we don't have to add new Node subclasses), we can addnew functionality simply by defining new NodeVisitor subclasses.

Applicability

Use the Visitor pattern when

- an object structure contains many classes of objects with differing interfaces, and you want to perform operations on these objects that depend on their concrete classes.
- many distinct and unrelated operations need to be performed on objectsin an object structure, and you want to avoid "polluting" theirclasses with these operations. Visitor lets you keep related operationstogether by defining them in one class. When the object structure isshared by many applications, use Visitor to put operations in just thoseapplications that need them.
- the classes defining the object structure rarely change, but you oftenwant to define new operations over the structure. Changing the objectstructure classes requires redefining the interface to all visitors,which is potentially costly. If the object structure classes changeoften, then it's probably better to define the operations in those classes.

Structure



Participants

- **Visitor** (NodeVisitor)
  - o declares a Visit operation for each class of ConcreteElement in the object structure. The operation's name and signature identifies 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.
- **ConcreteVisitor** (TypeCheckingVisitor)
  - o implements each operation declared by Visitor. Each operation implements a fragment of the algorithm defined for the corresponding class of object in the structure. ConcreteVisitor provides the context for the algorithm and stores its local state. This state often accumulates results during the traversal of the structure.
- **Element** (Node)
  - o defines an Accept operation that takes a visitor as an argument.
- **ConcreteElement** (AssignmentNode,VariableRefNode)

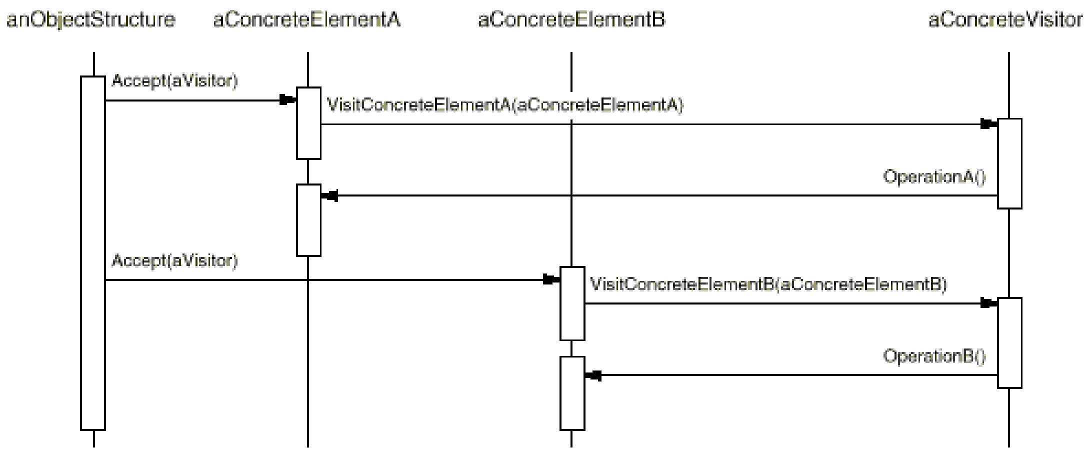
- implements an Accept operation that takes a visitor as an argument.

- **ObjectStructure** (Program)
  - can enumerate its elements.
  - may provide a high-level interface to allow the visitor to visit its elements.
  - may either be a composite (see Composite (183)) or a collection such as a list or a set.

Collaborations

- A client that uses the Visitor pattern must create a ConcreteVisitorobject and then traverse the object structure, visiting each elementwith the visitor.
- When an element is visited, it calls the Visitor operation thatcorresponds to its class. The element supplies itself as an argumentto this operation to let the visitor access its state, if necessary.

The following interaction diagram illustrates the collaborationsbetween an object structure, a visitor, and two elements:



Consequences

Some of the benefits and liabilities of the Visitor pattern are as follows:

1. *Visitor makes adding new operations easy.*Visitors make it easy to add operations that depend on the components ofcomplex objects. You can define a new operation over an object structuresimply by adding a new visitor. In contrast, if you spread functionalityover many classes, then you must change each class to define a newoperation.

2. *A visitor gathers related operations and separates unrelated ones.*Related behavior isn't spread over the classes defining the objectstructure; it's localized in a visitor. Unrelated sets of behavior arepartitioned in their own visitor subclasses. That simplifies both theclasses defining the elements and the algorithms defined in thevisitors. Any algorithm-specific data structures can be hidden in thevisitor.
3. *Adding new ConcreteElement classes is hard.*The Visitor pattern makes it hard to add new subclasses of Element. Eachnew ConcreteElement gives rise to a new abstract operation on Visitor anda corresponding implementation in every ConcreteVisitor class. Sometimes adefault implementation can be provided in Visitor that can be inheritedby most of the ConcreteVisitors, but this is the exception rather thanthe rule.

   So the key consideration in applying the Visitor pattern is whether youare mostly likely to change the algorithm applied over an objectstructure or the classes of objects that make up the structure. TheVisitor class hierarchy can be difficult to maintain when newConcreteElement classes are added frequently. In such cases, it'sprobably easier just to define operations on the classes that make upthe structure. If the Element class hierarchy is stable, but you arecontinually adding operations or changing algorithms, then the Visitorpattern will help you manage the changes.

4. *Visiting across class hierarchies.*An iterator (see Iterator (289)) can visit the objects in astructure as it traverses them by calling their operations. But an iteratorcan't work across object structures with different types of elements. Forexample, the Iterator interface defined on page 295 can access only objects of type Item:

```
template <class Item>
class Iterator {
        // ...
        Item CurrentItem() const;
};
```

This implies that all elements the iterator can visit have a common parentclass Item.

Visitor does not have this restriction. It can visit objects thatdon't have a common parent class. You can add any type of object to aVisitor interface. For example, in

```
class Visitor {
public:
        // ...
        void VisitMyType(MyType*);
```

```
    void VisitYourType(YourType*);
};
```

MyType and YourType do not have to be related throughinheritance at all.

5. *Accumulating state.*Visitors can accumulate state as they visit each element in the objectstructure. Without a visitor, this state would be passed as extraarguments to the operations that perform the traversal, or theymight appear as global variables.
6. *Breaking encapsulation.*Visitor's approach assumes that the ConcreteElement interface is powerfulenough to let visitors do their job. As a result, the pattern oftenforces you to provide public operations that access an element'sinternal state, which may compromise its encapsulation.

Implementation

Each object structure will have an associated Visitor class. Thisabstract visitor class declares a VisitConcreteElement operation foreach class of ConcreteElement defining the object structure. EachVisit operation on the Visitor declares its argument to be aparticular ConcreteElement, allowing the Visitor to access theinterface of the ConcreteElement directly. ConcreteVisitor classesoverride each Visit operation to implement visitor-specific behaviorfor the corresponding ConcreteElement class.

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 operationthat calls the matching Visit... operation on the visitorfor that ConcreteElement. Thus the operation that ends up gettingcalled depends on both the class of the element and the class of thevisitor.[10]

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 Acceptlike 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);
}
```

Here are two other implementation issues that arise when you apply theVisitor pattern:

1. *Double dispatch.*Effectively, the Visitor pattern lets you add operations to classeswithout changing them. Visitor achieves this by using a techniquecalled **double-dispatch.** It's a well-known technique. Infact, some

programming languages support it directly (CLOS, forexample). Languages like C++ and Smalltalk support**single-dispatch**.

In single-dispatch languages, two criteria determine which operationwill fulfill a request: the name of the request and the type ofreceiver. For example, the operation that a GenerateCode request willcall depends on the type of node object you ask. In C++, callingGenerateCode on an instance of VariableRefNode willcall VariableRefNode::GenerateCode (which generates code for avariable reference). Calling GenerateCode on anAssignmentNode will callAssignmentNode::GenerateCode (which will generate code for anassignment). The operation that gets executed depends both on the kindof request and the type of the receiver.

"Double-dispatch" simply means the operation that gets executeddepends on the kind of request and the types of *two* receivers.Accept is a double-dispatch operation. Its meaning dependson two types: the Visitor's and the Element's. Double-dispatchinglets visitors request different operations on each class ofelement.[11]

This is the key to the Visitor pattern: The operation that getsexecuted depends on both the type of Visitor and the type of Elementit visits. Instead of binding operations statically into the Elementinterface, you can consolidate the operations in a Visitor and useAccept to do the binding at run-time. Extending the Elementinterface amounts to defining one new Visitor subclass rather than many newElement subclasses.

2. *Who is responsible for traversing the object structure?*A visitor must visit each element of the object structure. The questionis, how does it get there? We can put responsibility for traversal inany of three places: in the object structure, in the visitor, or in aseparate iterator object (see Iterator (289)).

   Often the object structure is responsible for iteration. A collectionwill simply iterate over its elements, calling the Accept operation oneach. A composite will commonly traverse itself by having each Acceptoperation traverse the element's children and call Accept on each ofthem recursively.

   Another solution is to use an iterator to visit the elements. In C++,you could use either an internal or external iterator, depending on whatis available and what is most efficient. In Smalltalk, you usually usean internal iterator using do: and a block. Since internaliterators are implemented by the object structure, using an internaliterator is a lot like making the object structure responsible foriteration. The main difference is that an internal iterator will notcause double-dispatching—it will call an operation on the *visitor* with an *element* as an argument as opposed to calling anoperation on the *element* with the *visitor* as an argument.But it's easy to use the Visitor pattern with an internal iterator ifthe operation on the visitor simply calls the operation on the elementwithout recursing.

You could even put the traversal algorithm in the visitor, although you'llend up duplicating the traversal code in each ConcreteVisitor for eachaggregate ConcreteElement. The main reason to put the traversal strategyin the visitor is to implement a particularly complex traversal, onethat depends on the results of the operations on the object structure.We'll give an example of such a case in the Sample Code.

Sample Code

Because visitors are usually associated with composites, we'll use theEquipment classes defined in the Sample Code of Composite (183) to illustrate the Visitor pattern. Wewill use Visitor to define operations for computing theinventory of materials and the total cost for a piece of equipment.The Equipment classes are so simple that using Visitorisn't really necessary, but they make it easy to see what'sinvolved in implementing the pattern.

Here again is the Equipment class from Composite (183). We've augmented it with anAccept operation to let it work with a visitor.

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

        const char* Name() { return _name; }

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

        virtual void Accept(EquipmentVisitor&);
protected:
        Equipment(const char*);
private:
        const char* _name;
};
```

The Equipment operations return the attributes of a piece ofequipment, such as its power consumption and cost. Subclasses redefinethese operations appropriately for specific types of equipment (e.g.,a chassis, drives, and planar boards).

The abstract class for all visitors of equipment has a virtualfunction for each subclass of equipment, as shown next. All of thevirtual functions do nothing by default.

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

        virtual void VisitFloppyDisk(FloppyDisk*);
        virtual void VisitCard(Card*);
        virtual void VisitChassis(Chassis*);
        virtual void VisitBus(Bus*);

        // and so on for other concrete subclasses of Equipment
protected:
        EquipmentVisitor();
};
```

Equipment subclasses define Accept inbasically the same way: It calls theEquipmentVisitor operation that corresponds to the classthat received the Accept request, like this:

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

Equipment that contains other equipment (in particular, subclasses ofCompositeEquipment in the Composite pattern) implementsAccept by iterating over its children and callingAccept on each of them. Then it calls theVisit operation as usual.For example, Chassis::Accept could traverseall the parts in the chassis as follows:

```
void Chassis::Accept (EquipmentVisitor& visitor) {
        for (
                ListIterator i(_parts);
                !i.IsDone();
                i.Next()
        ) {
                i.CurrentItem()->Accept(visitor);
        }
        visitor.VisitChassis(this);
}
```

Subclasses of EquipmentVisitor define particular algorithmsover the equipment structure. The PricingVisitor computes thecost of the equipment structure. It computes the net price of all simpleequipment (e.g., floppies) and the discount price of all compositeequipment (e.g., chassis and buses).

```
class PricingVisitor : public EquipmentVisitor {
public:
        PricingVisitor();

        Currency& GetTotalPrice();

        virtual void VisitFloppyDisk(FloppyDisk*);
        virtual void VisitCard(Card*);
        virtual void VisitChassis(Chassis*);
        virtual void VisitBus(Bus*);
        // ...
private:
        Currency _total;
};

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

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

PricingVisitor will compute the total cost of all nodes in theequipment structure. Note that PricingVisitor chooses theappropriate pricing policy for a class of equipment by dispatching tothe corresponding member function. What's more, we can change thepricing policy of an equipment structure just by changing thePricingVisitor class.

We can define a visitor for computing inventory like this:

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

        Inventory& GetInventory();

        virtual void VisitFloppyDisk(FloppyDisk*);
        virtual void VisitCard(Card*);
        virtual void VisitChassis(Chassis*);
        virtual void VisitBus(Bus*);
```

```
    // ...
private:
    Inventory _inventory;
};
```

The InventoryVisitor accumulates the totals for each type ofequipment in the object structure. InventoryVisitor uses anInventory class that defines an interface for adding equipment(which we won't bother defining here).

```
void InventoryVisitor::VisitFloppyDisk (FloppyDisk* e) {
    _inventory.Accumulate(e);
}

void InventoryVisitor::VisitChassis (Chassis* e) {
    _inventory.Accumulate(e);
}
```

Here's how we can use an InventoryVisitor on anequipment structure:

```
Equipment* component;
InventoryVisitor visitor;

component->Accept(visitor);
cout << "Inventory "
    << component->Name()
    << visitor.GetInventory();
```

Now we'll show how to implement the Smalltalk example from the Interpreter pattern (see page 279) with theVisitor pattern. Like the previous example, this one is so small thatVisitor probably won't buy us much, but it provides a good illustration ofhow to use the pattern. Further, it illustrates a situation in whichiteration is the visitor's responsibility.

The object structure (regular expressions) is made of four classes,and all of them have an accept: method that takes thevisitor as an argument. In class SequenceExpression, theaccept: method is

```
accept: aVisitor
    ^ aVisitor visitSequence: self
```

In class RepeatExpression, the accept: methodsends the visitRepeat: message.In class AlternationExpression, it sends thevisitAlternation: message.In class LiteralExpression, it sends thevisitLiteral: message.

The four classes also must have accessing functions that the visitorcan use. For SequenceExpression these areexpression1 and expression2; forAlternationExpression these are alternative1and alternative2; forRepeatExpression it is repetition; and forLiteralExpression these are components.

The ConcreteVisitor class is REMatchingVisitor. Itis responsible for the traversal because its traversal algorithmis irregular. The biggest irregularity is that aRepeatExpression will repeatedly traverse its component.The class REMatchingVisitor has an instance variableinputState. Its methods are essentially the same asthe match: methods of the expression classes in theInterpreter pattern except theyreplace the argument named inputState with theexpression node being matched. However, theystill return the set of streams that the expression would matchto identify the current state.

```
visitSequence: sequenceExp
         inputState := sequenceExp expression1 accept: self.
         ^ sequenceExp expression2 accept: self.

visitRepeat: repeatExp
         | finalState |
         finalState := inputState copy.
         [inputState isEmpty]
                  whileFalse:
                           [inputState := repeatExp repetition accept: self.
                           finalState addAll: inputState].
         ^ finalState

visitAlternation: alternateExp
         | finalState originalState |
         originalState := inputState.
         finalState := alternateExp alternative1 accept: self.
         inputState := originalState.
         finalState addAll: (alternateExp alternative2 accept: self).
         ^ finalState

visitLiteral: literalExp
         | finalState tStream |
         finalState := Set new.
         inputState
                  do:
                           [:stream | tStream := stream copy.
                                    (tStream nextAvailable:
                                             literalExp components size
```

```
                                    ) = literalExp components
                                             ifTrue: [finalState add: tStream]
                          ].
          ^ finalState
```

Known Uses

The Smalltalk-80 compiler has a Visitor class called ProgramNodeEnumerator.It's used primarily for algorithms that analyze source code.It isn't used for code generation or pretty-printing, although it could be.

IRIS Inventor [Str93]is a toolkit for developing 3-D graphics applications. Inventorrepresents a three-dimensional scene as a hierarchy of nodes, eachrepresenting either a geometric object or an attribute of one.Operations like rendering a scene or mapping an input event requiretraversing this hierarchy in different ways. Inventor does thisusing visitors called "actions." There are different visitors forrendering, event handling, searching, filing, and determiningbounding boxes.

To make adding new nodes easier, Inventor implements adouble-dispatch scheme for C++. The scheme relies on run-time typeinformation and a two-dimensional table in which rows representvisitors and columns represent node classes. The cells store apointer to the function bound to the visitor and node class.

Mark Linton coined the term "Visitor" in the X Consortium'sFresco Application Toolkit specification [LP93].

Related Patterns

Composite (183):Visitors can be used to apply an operation over an object structuredefined by the Composite pattern.

Interpreter (274):Visitor may be applied to do the interpretation.

---

[10]We could use function overloading to give these operationsthe same simple name, like Visit, since the operations arealready differentiated by the parameter they're passed. There arepros and cons to such overloading. On the one hand, it reinforces thefact that each operation involves the same analysis, albeit on adifferent argument. On the other hand, that might make what's goingon at the

call site less obvious to someone reading the code. Itreally boils down to whether you believe function overloading is goodor not.

[11]If we can have *double*-dispatch, then why not*triple* or *quadruple*, or any other number? Actually,double-dispatch is just a special case of **multipledispatch**, in which the operation is chosen based on any number oftypes. (CLOS actually supports multiple dispatch.) Languages thatsupport double- or multiple dispatch lessen the need for theVisitor pattern.