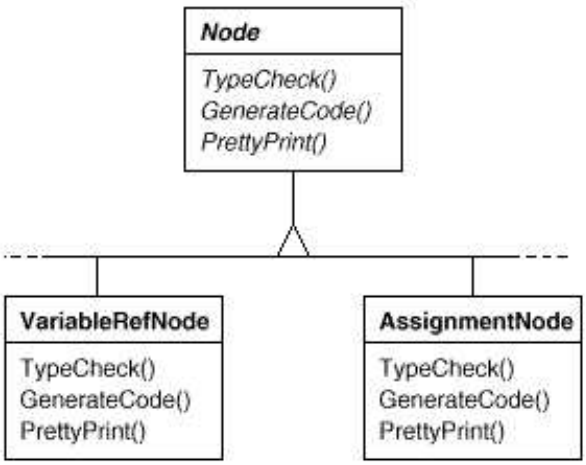
# Intent

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.

# Motivation

Consider a compiler that represents programs as abstract syntax trees. It will need to perform operations on abstract syntax trees for "static semantic" analyses like checking that all variables are defined. It will also need to generate code. So it might define operations for type-checking, code optimization, flow analysis, checking for variables being assigned values before they're used, and so on. Moreover, we could use the abstract syntax trees for pretty-printing, program restructuring, code instrumentation, and computing various metrics of a program.

Most of these operations will need to treat nodes that represent assignment statements differently from nodes that represent variables or arithmetic expressions. Hence there will be one class for assignment statements, another for variable accesses, another for arithmetic expressions, and so on. The set of node classes depends on the language being compiled, of course, but it doesn't change much for a given language.



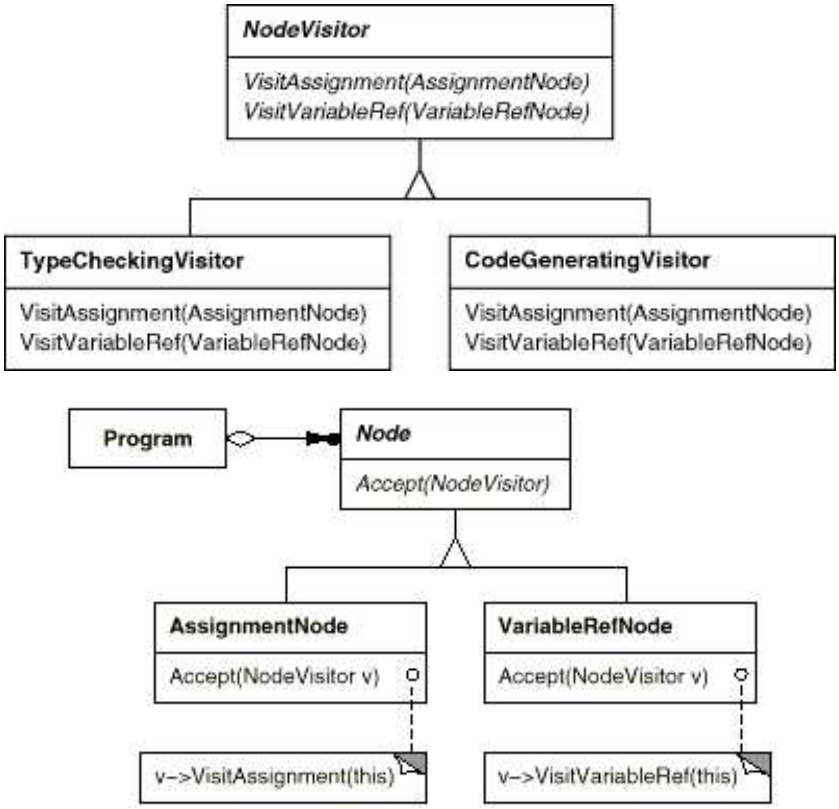

This diagram shows part of the Node class hierarchy. The problem here is that distributing all these operations across the various node classes leads to a system that's hard to understand, maintain, and change. It will be confusing to have type-checking code mixed with pretty-printing code or flow analysis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

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

For example, a compiler that didn't use visitors might type-check a procedure by calling the TypeCheck operation on its abstract syntax tree. Each of the nodes would implement TypeCheck by

calling TypeCheck on its components (see the preceding class diagram). If the compiler type-checked a procedure using visitors, then it would create a TypeCheckingVisitor object and call the Accept operation on the abstract syntax tree with that object as an argument. Each of the nodes would implement Accept by calling back on the visitor: an assignment node calls VisitAssignment operation on the visitor, while a variable reference calls VisitVariableReference. What used to be the TypeCheck operation in class AssignmentNode is now the VisitAssignment operation on TypeCheckingVisitor.

To make visitors work for more than just type-checking, we need an abstract parent class NodeVisitor for all visitors of an abstract syntax tree. NodeVisitor must declare an operation for each node class. An application that needs to compute program metrics will define new subclasses 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 the elements being operated on (the Node hierarchy) and one for the visitors that define operations on the elements (the NodeVisitor hierarchy). You create a new operation by adding a new subclass to the visitor class hierarchy. As long as the grammar that the compiler accepts doesn't change (that is, we don't have to add new Node subclasses), we can add new functionality simply by defining new NodeVisitor subclasses.

## Applicability

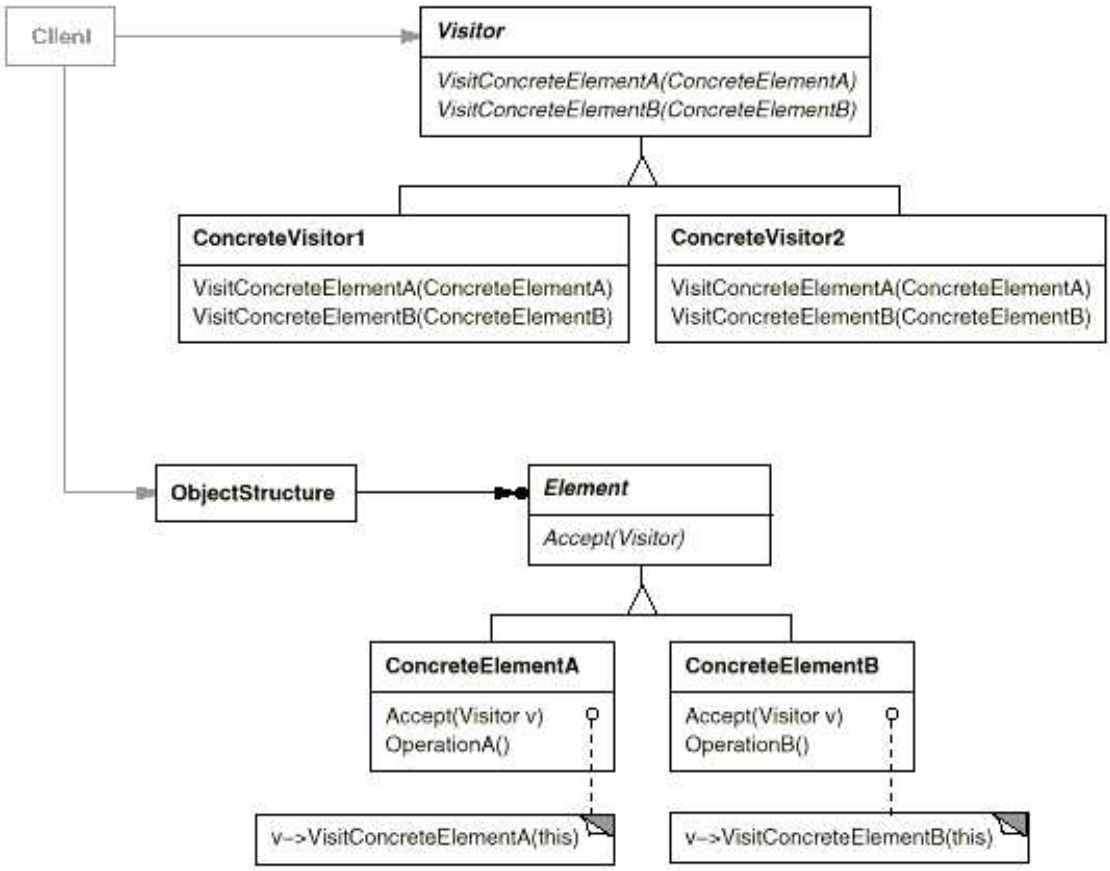
Use the Visitor pattern when

- 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 objects in an object structure,

and you want to avoid "polluting" their classes with these operations. Visitor lets you keep related operations together by defining them in one class. When the object structure is shared by many applications, use Visitor to put operations in just those applications that need them.

- the classes defining the object structure rarely change, but you often want to define new operations over the structure. Changing the object structure classes requires redefining the interface to all visitors, which is potentially costly. If the object structure classes change often, then it’s probably better to define the operations in those classes.

# Structure




# Participants

- **Visitor** (NodeVisitor)
    - 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)
    - 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

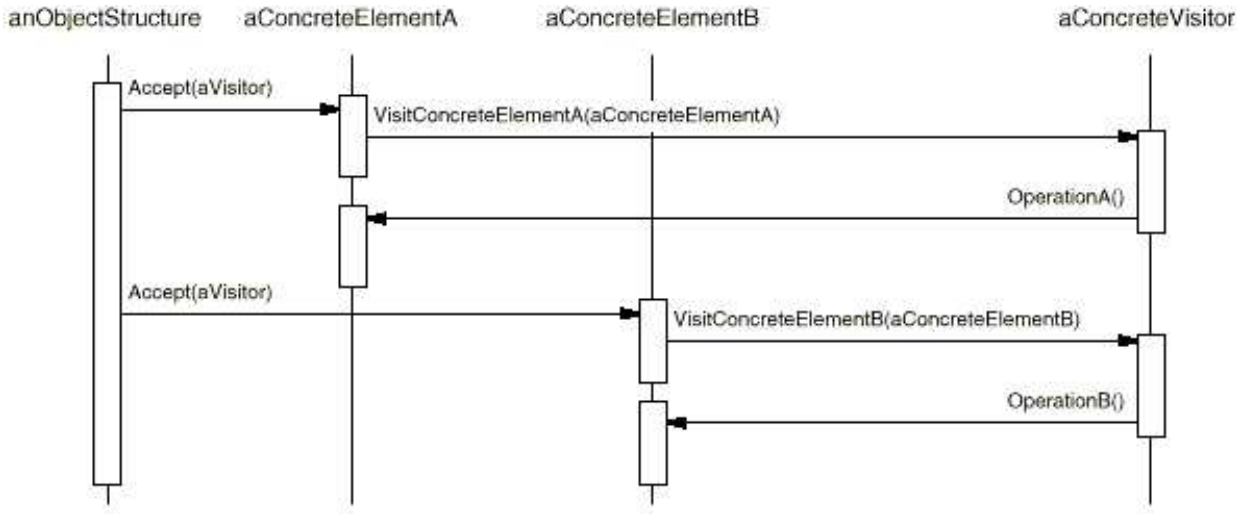
state often accumulates results during the traversal of the structure.

- **Element** (Node)
  - 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 (163)) or a collection such as a list or a set.

# Collaborations

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

  The following interaction diagram illustrates the collaborations between 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 of complex objects. You can define a new operation over an object structure

simply by adding a new visitor. In contrast, if you spread functionality over many classes, then you must change each class to define a new operation.

2. *A visitor gathers related operations and separates unrelated ones.* Related behavior isn't spread over the classes defining the object structure; it's localized in a visitor. Unrelated sets of behavior are partitioned in their own visitor subclasses. That simplifies both the classes defining the elements and the algorithms defined in the visitors. Any algorithm-specific data structures can be hidden in the visitor.

3. *Adding new ConcreteElement classes is hard.* The Visitor pattern makes it hard to add new subclasses of Element. Each new ConcreteElement gives rise to a new abstract operation on Visitor and a corresponding implementation in every ConcreteVisitor class. Sometimes a default implementation can be provided in Visitor that can be inherited by most of the ConcreteVisitors, but this is the exception rather than the rule.

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

4. *Visiting across class hierarchies.* An iterator (see Iterator (257)) can visit the objects in a structure as it traverses them by calling their operations. But an iterator can't work across object structures with different types of elements. For example, the Iterator interface defined on page 263 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 parent class `Item`.

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

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

   `MyType` and `YourType` do not have to be related through inheritance at all.

5. *Accumulating state.* Visitors can accumulate state as they visit each element in the object structure. Without a visitor, this state would be passed as extra arguments to the operations that perform the traversal, or they might appear as global variables.

6. *Breaking encapsulation.* Visitor's approach assumes that the ConcreteElement interface is powerful enough to let visitors do their job. As a result, the pattern often forces you to provide public operations that access an element's internal state, which may compromise its encapsulation.

# Implementation

Each object structure will have an associated Visitor class. This abstract visitor class declares a VisitConcreteElement operation for each class of ConcreteElement defining the object structure. Each Visit operation on the Visitor declares its argument to be a particular ConcreteElement, allowing the Visitor to access the interface of the ConcreteElement directly. ConcreteVisitor classes override each Visit operation to implement visitor-specific behavior for 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` operation that calls the matching `Visit...` 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.[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 `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);
}
```

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

1. *Double dispatch.* Effectively, the Visitor pattern lets you add operations to classes without changing them. Visitor achieves this by using a technique called **double-dispatch**. It's a well-known technique. In fact, some programming languages support it directly (CLOS, for example). Languages like C++ and Smalltalk support **single-dispatch**.

   In single-dispatch languages, two criteria determine which operation will fulfill a request: the name of the request and the type of receiver. For example, the operation that a GenerateCode request will call depends on the type of node object you ask. In C++, calling `GenerateCode` on an instance of `VariableRefNode` will call `VariableRefNode::GenerateCode` (which generates code for a variable reference). Calling `GenerateCode` on an `AssignmentNode` will call `AssignmentNode::GenerateCode` (which will generate code for an assignment). The operation that gets executed depends both on the kind of request and the type of the receiver.

   "Double-dispatch" simply means the operation that gets executed depends on the kind of request and the types of *two* receivers. `Accept` is a double-dispatch operation. Its meaning depends on two types: the Visitor's and the Element's. Double-dispatching lets visitors request different operations on each class of element.[11]

   This is the key to the Visitor pattern: The operation that gets executed depends on both the type of Visitor and the type of Element it visits. Instead of binding operations statically into the Element interface, you can consolidate the operations in a Visitor and use `Accept` to do the binding at run-time. Extending the Element interface amounts to defining one new Visitor subclass rather than many new Element subclasses.

2. *Who is responsible for traversing the object structure?* A visitor must visit each element of the object structure. The question is, how does it get there? We can put responsibility for traversal in any of three places: in the object structure, in the visitor, or in a separate iterator object (see Iterator (257)).

   Often the object structure is responsible for iteration. A collection will simply iterate over its elements, calling the Accept operation on each. A composite will commonly traverse itself by having each Accept operation traverse the element's children and call Accept on each of them 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 what is available and what is most efficient. In Smalltalk, you usually use an internal iterator using `do:` and a block. Since internal iterators are implemented by the object structure, using an internal iterator is a lot like making the object structure responsible for iteration. The main difference is that an internal iterator will not cause double-dispatching—it will call an operation on the *visitor* with an *element* as an argument as opposed to calling an operation on the *element* with the *visitor* as an argument. But it's easy to use the Visitor pattern with an internal iterator if the operation on the visitor simply calls the operation on the element without recursing.

   You could even put the traversal algorithm in the visitor, although you'll end up duplicating the traversal code in each ConcreteVisitor for each aggregate ConcreteElement. The main reason to put the traversal strategy in the visitor is to implement a particularly complex traversal, one that 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 the `Equipment` classes defined in the Sample Code of Composite (163) to illustrate the Visitor pattern. We will use Visitor to define operations for computing the inventory of materials and the total cost for a piece of equipment. The `Equipment` classes are so simple that using Visitor isn't really necessary, but they make it easy to see what's involved in implementing the pattern.

Here again is the `Equipment` class from Composite (163). We've augmented it with an `Accept` 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 of equipment, such as its power consumption and cost. Subclasses redefine these 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 virtual function for each subclass of equipment, as shown next. All of the virtual 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` in basically the same way: It calls the `EquipmentVisitor` operation that corresponds to the class that received the `Accept` request, like this:

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

Equipment that contains other equipment (in particular, subclasses of `CompositeEquipment` in the Composite pattern) implements `Accept` by iterating over its children and calling `Accept` on each of them. Then it calls the `Visit` operation as usual. For example, `Chassis::Accept` could traverse all 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 algorithms over the equipment structure. The `PricingVisitor` computes the cost of the equipment structure. It computes the net price of all simple equipment (e.g., floppies) and the discount price of all composite equipment (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 the equipment structure. Note that `PricingVisitor` chooses the appropriate pricing policy for a class of equipment by dispatching to the corresponding member function. What's more, we can change the pricing policy of an equipment structure just by changing the `PricingVisitor` 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 of equipment in the object structure. `InventoryVisitor` uses an `Inventory` 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 an equipment 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 248) with the Visitor pattern. Like the previous example, this one is so small that Visitor probably won't buy us much, but it provides a good illustration of how to use the pattern. Further, it illustrates a situation in which iteration 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 the visitor as an argument. In class `SequenceExpression`, the `accept:` method is

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

In class `RepeatExpression`, the `accept:` method sends the `visitRepeat:` message. In class `AlternationExpression`, it sends the `visitAlternation:` message. In class `LiteralExpression`, it sends the `visitLiteral:` message.

The four classes also must have accessing functions that the visitor can use. For `SequenceExpression` these are `expression1` and `expression2`; for `AlternationExpression` these are `alternative1` and `alternative2`; for `RepeatExpression` it is `repetition`; and for `LiteralExpression` these are `components`.

The ConcreteVisitor class is `REMatchingVisitor`. It is responsible for the traversal because its traversal algorithm is irregular. The biggest irregularity is that a `RepeatExpression` will repeatedly traverse its component. The class `REMatchingVisitor` has an instance variable `inputState`. Its methods are essentially the same as the `match:` methods of the expression classes in the Interpreter pattern except they replace the argument named `inputState` with the expression node being matched. However, they still return the set of streams that the expression would match to 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. Inventor represents a three-dimensional scene as a hierarchy of nodes, each representing either a geometric object or an attribute of one. Operations like rendering a scene or mapping an input event require traversing this hierarchy in different ways. Inventor does this using visitors called "actions." There are different visitors for rendering, event handling, searching, filing, and determining bounding boxes.

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

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

## Related Patterns

Composite (163): Visitors can be used to apply an operation over an object structure defined by the Composite pattern.

Interpreter (243): Visitor may be applied to do the interpretation.


Discussion of Behavioral Patterns
Template Method


---

[10]We could use function overloading to give these operations the same simple name, like `Visit`, since the operations are already differentiated by the parameter they're passed. There are pros and cons to such overloading. On the one hand, it reinforces the fact that each operation involves the same analysis, albeit on a different argument. On the other hand, that might make what's going on at the call site less obvious to someone reading the code. It really boils down to whether you believe function overloading is good or 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 **multiple dispatch**, in which the operation is chosen based on any number of types. (CLOS actually supports multiple dispatch.) Languages that support double- or multiple dispatch lessen the need for the Visitor pattern.