**Design pattern: Visitor**

**1. Name and classification:**

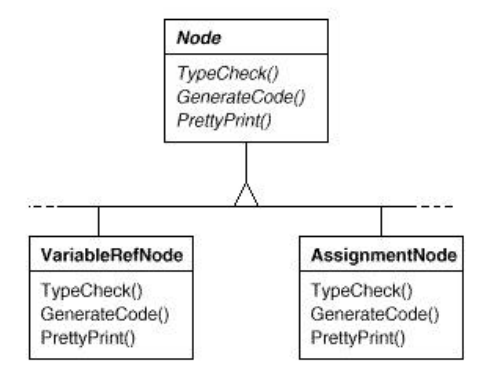
Visitor comes under behavior pattern category

**2. Also Known As:**

**3. Motivation:**

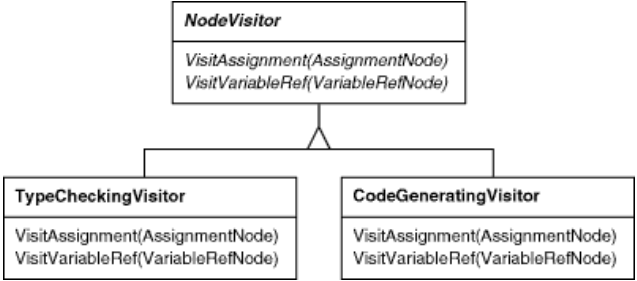
Consider a compiler that parses a program and represents the parsed program as an abstract syntax tree (AST). The AST has many different kinds of nodes, such as Assignment , Variable Reference.

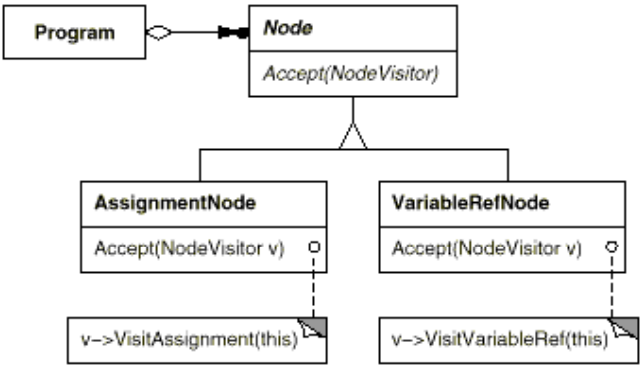
Consider two operation to perform on the AST: Type checking and code generation. These operations may need to treat each type of node differently. One way to do this is to define each operation in the specific node class.



Problems with this approach: › Adding new operations requires changes to all of the node classes (violate the open/close priciple)

Another solution is to encapsulate a desired operation in a separate object, called a visitor. The visitor object then traverses the elements of the tree. When an tree node "accepts" the visitor, it invokes a method on the visitor that includes the node type as an argument. The visitor will then execute the operation for that node - the operation that used to be in the node class.





**4. Applicability**

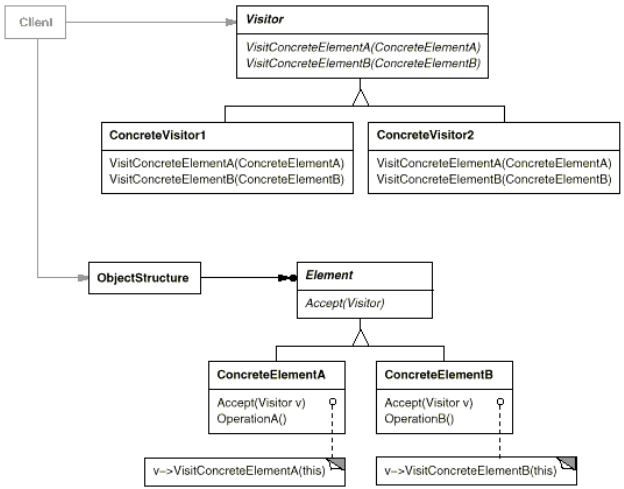
Use the Visitor pattern in any of the following situations:

- 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

- When the classes defining the object structure rarely change, but you often want to define new operations over the structure.

- 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

**5. Structure**



**6. Participants**

- Visitor: This is an interface or an abstract class used to declare the visit operations for all the types of visitable classes. Usually the name of the operation is the same and the operations are differentiated by the method signature: The input object type decides which of the method is called.

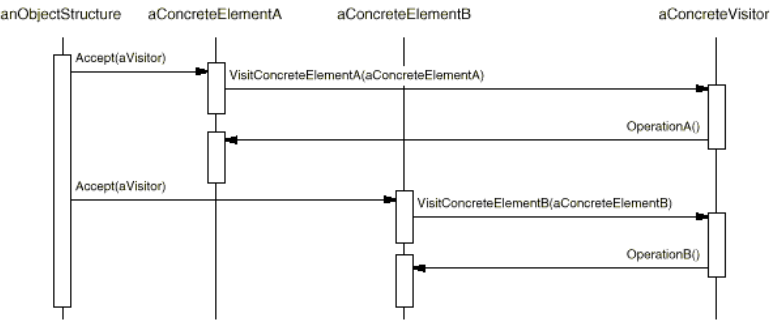
- ConcreteVisitor - For each type of visitor all the visit methods, declared in abstract visitor, must be implemented. Each Visitor will be responsible for different operations. When a new visitor is defined it has to be passed to the object structure.

- **Visitable**- is an abstraction which declares the accept operation. This is the entry point which enables an object to be "visited" by the visitor object. Each object from a collection should implement this abstraction in order to be able to be visited.-

- **ConcreteVisitable**- Those classes implements the Visitable interface or class and defines the accept operation. The visitor object is passed to this object using the accept operation.

- **ObjectStructure**- This is a class containing all the objects that can be visited. It offers a mechanism to iterate through all the elements. This structure is not necessarily a collection. In can be a complex structure, such as a composite object.

**7. Collaboration**



**8. Consequences**

**- Benefit:**

› Adding new operations is easy

› 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.

› Visitors can accumulate state as they visit each element in the object structure. Without a visitor, this state would have to be passed as extra arguments to the operations that perform the traversal.

**- Liabilities:**

› Adding new ConcreteElement classes is hard. Each new ConcreteElement gives rise to a new abstract operation on Visitor and a corresponding implementation in every ConcreteVisitor class.

› The ConcreteElement interface must be powerful enough to let visitors do their job. You may be forced to provide public operations that access an element's internal state, which may compromise its encapsulation.

**9. Implementations + 10. Sample code**

## Step 1

Define an interface to represent element.

*Element.java*

public interface Element {

public void accept(Visitor visitor);

}

## Step 2

Create concrete classes extending the above class.

*ConcreteElementA.java*

public class ConcreteElementA implements Element {

@Override

public void accept( Visitor visitor) {

visitor.visit(this);

}

}

*ConcreteElementB.java*

public class ConcreteElementB implements Element {

@Override

public void accept( Visitor visitor) {

visitor.visit(this);

}

}

*ContainerElement.java*

public class ContainerElement implements Element {

Element[] elements;

public ContainerElement(){

elements = new Element[] {new ConcreteElementA(), new ConcreteElementB()};

}

@Override

public void accept( Visitor visitor) {

for (int i = 0; i < elements.length; i++) {

elements[i].accept(visitor);

}

visitor.visit(this);

}

}

## Step 3

Define an interface to represent visitor.

*Visitor.java*

public interface Visitor {

public void visit( ConcreteElementA concreteElementA);

public void visit( ConcreteElementB concreteElementB);

}

## Step 4

Create concrete visitor implementing the above class.

*ElementDisplayVisitor.java*

public class ElementDisplayVisitor implements Visitor {

@Override

public void visit( ContainerElement containerElement) {

System.out.println("Displaying container element.");

}

@Override

public void visit( ConcreteElementA concreteElementA) {

System.out.println("Displaying concrete element A.");

}

@Override

public void visit( ConcreteElementB concreteElementB) {

System.out.println("Displaying concrete element B.");

}

## Step 5

Use *ElementDisplayVisitor to* display parts of *ContainerElement*.

*VisitorPatternDemo.java*

public class VisitorPatternDemo {

public static void main(String[] args) {

Element container = new Element();

container.accept(new ElementDisplayVisitor());

}

}

## Step 6

Verify the output.

Displaying concrete element A.

Displaying concrete element B.

Displaying container element.

**11. 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].

**12. Related Patterns**

[Iterator pattern](https://en.wikipedia.org/wiki/Iterator_pattern) – defines a traversal principle like the visitor pattern, without making a type differentiation within the traversed objects