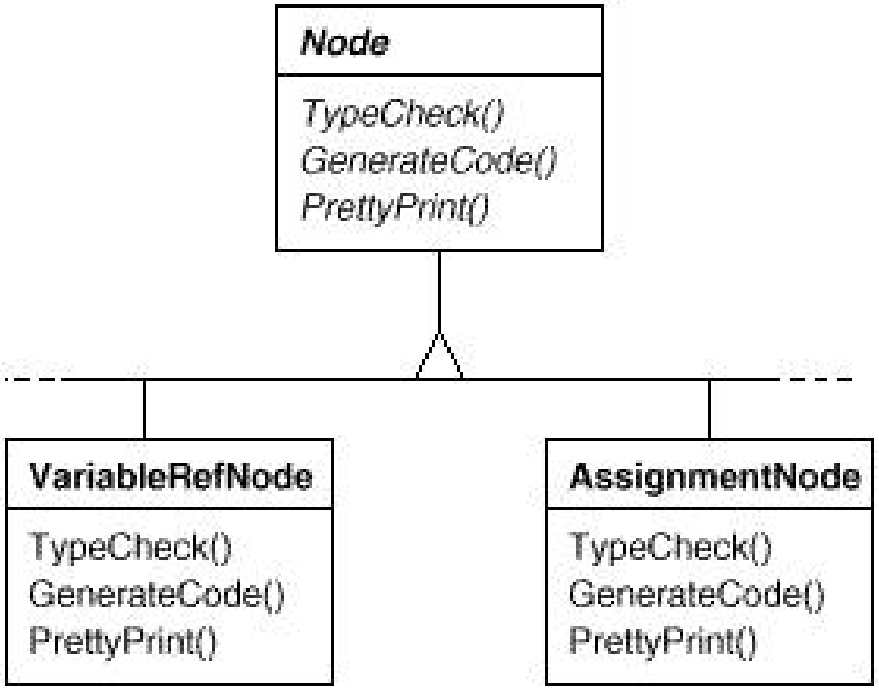
The Visitor Pattern

The Visitor Pattern

- 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 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, and Arithmetic Expression nodes.
  - Operations that one would like to perform on the AST include:
    - Checking that all variables are defined
    - Checking for variables being assigned before they are used
    - Type checking
    - Code generation
    - Pretty printing/formatting

The Visitor Pattern

- Motivation
  - ⇨ 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

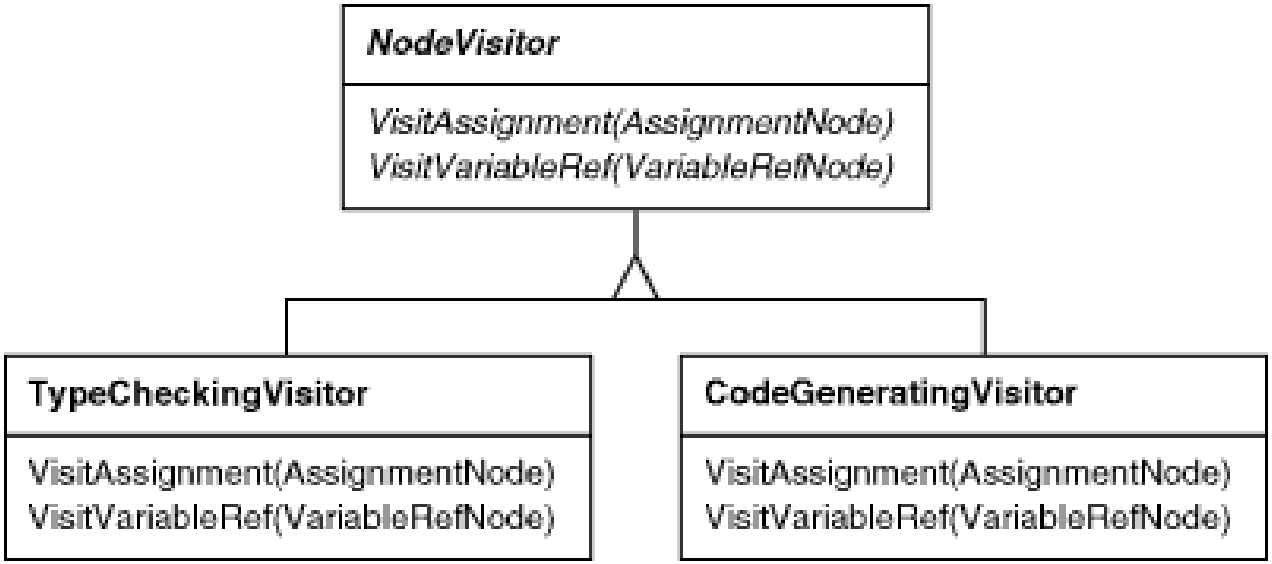


The Visitor Pattern

- Motivation
  - Problems with this approach:
    - Adding new operations requires changes to all of the node classes
    - It can be confusing to have such a diverse set of operations in each node class. For example, mixing type-checking code with pretty-printing code can be hard to understand and maintain.
  - 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.

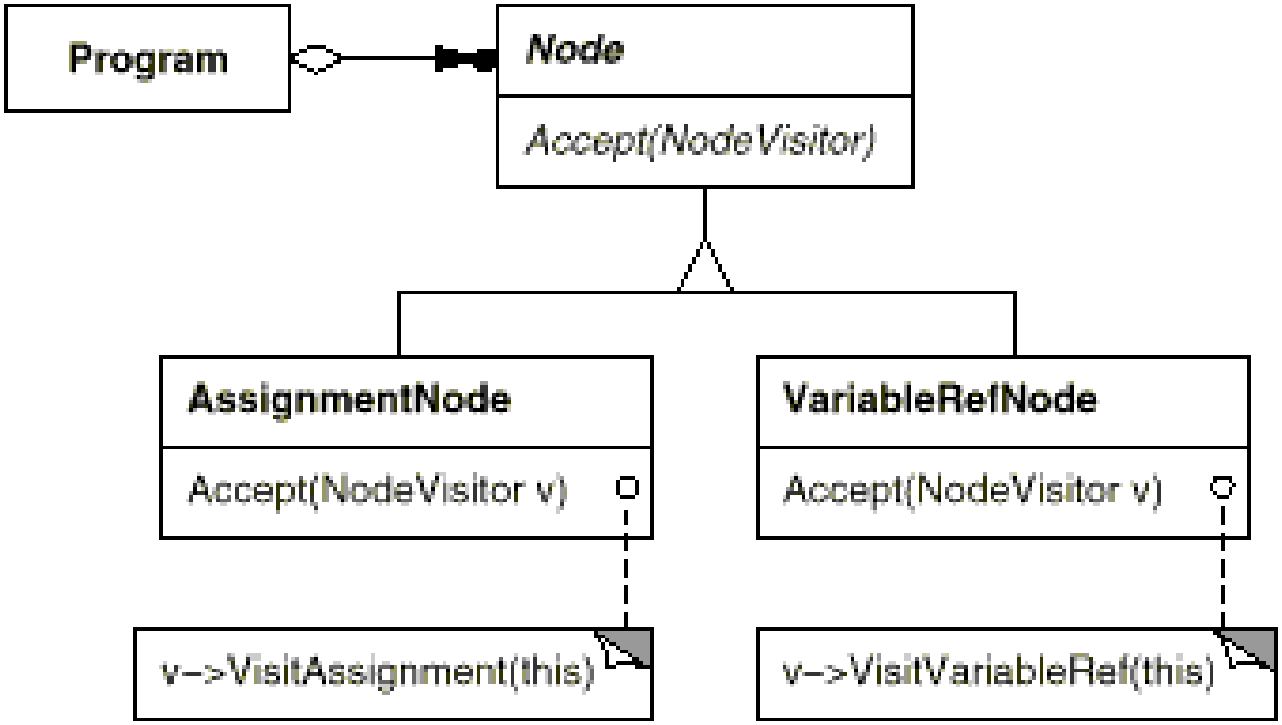
The Visitor Pattern

- Motivation



The Visitor Pattern

- Motivation



The Visitor Pattern

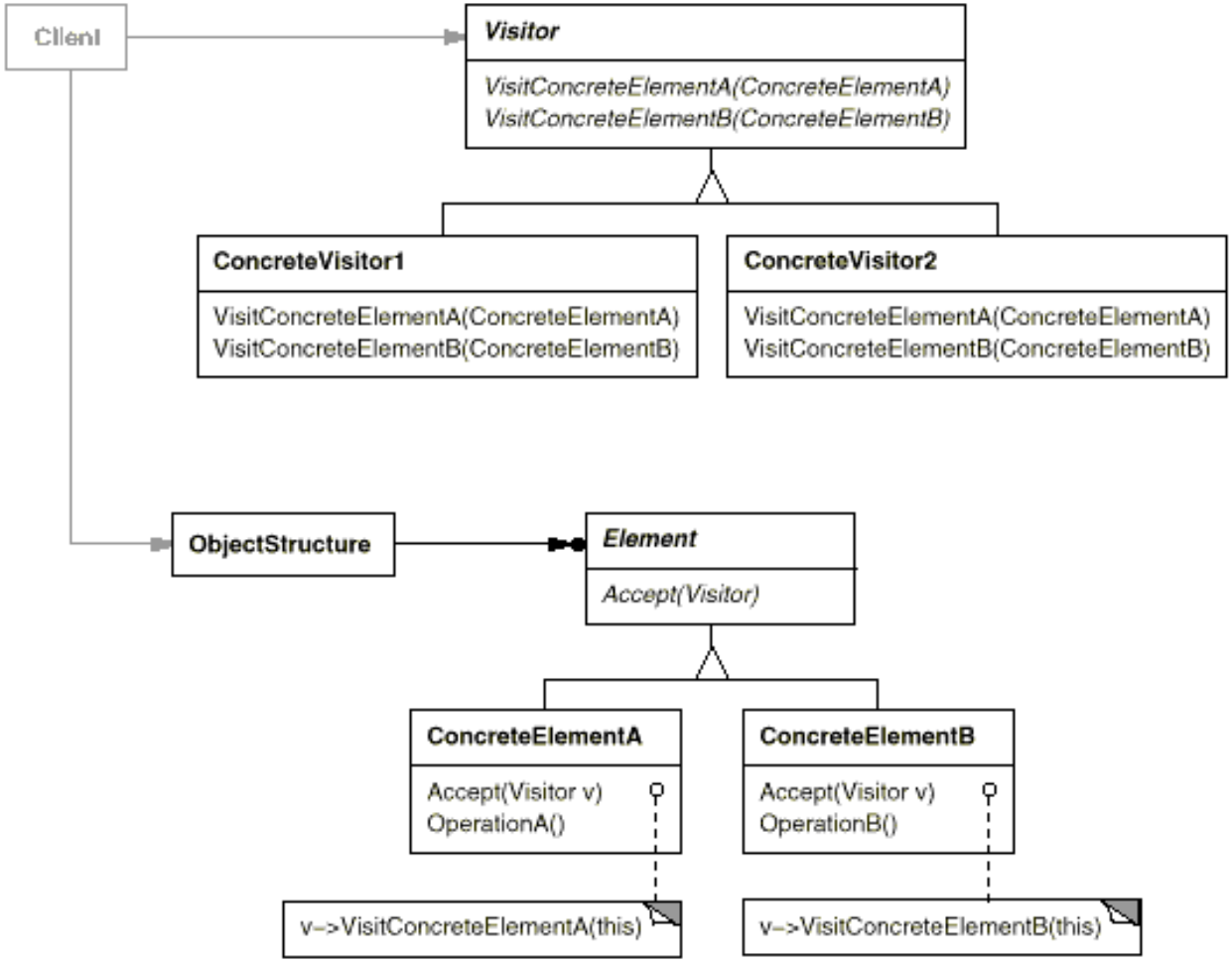
- Applicability

  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. (If the object structure classes change often, then it's probably better to define the operations in those classes.)
  - 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

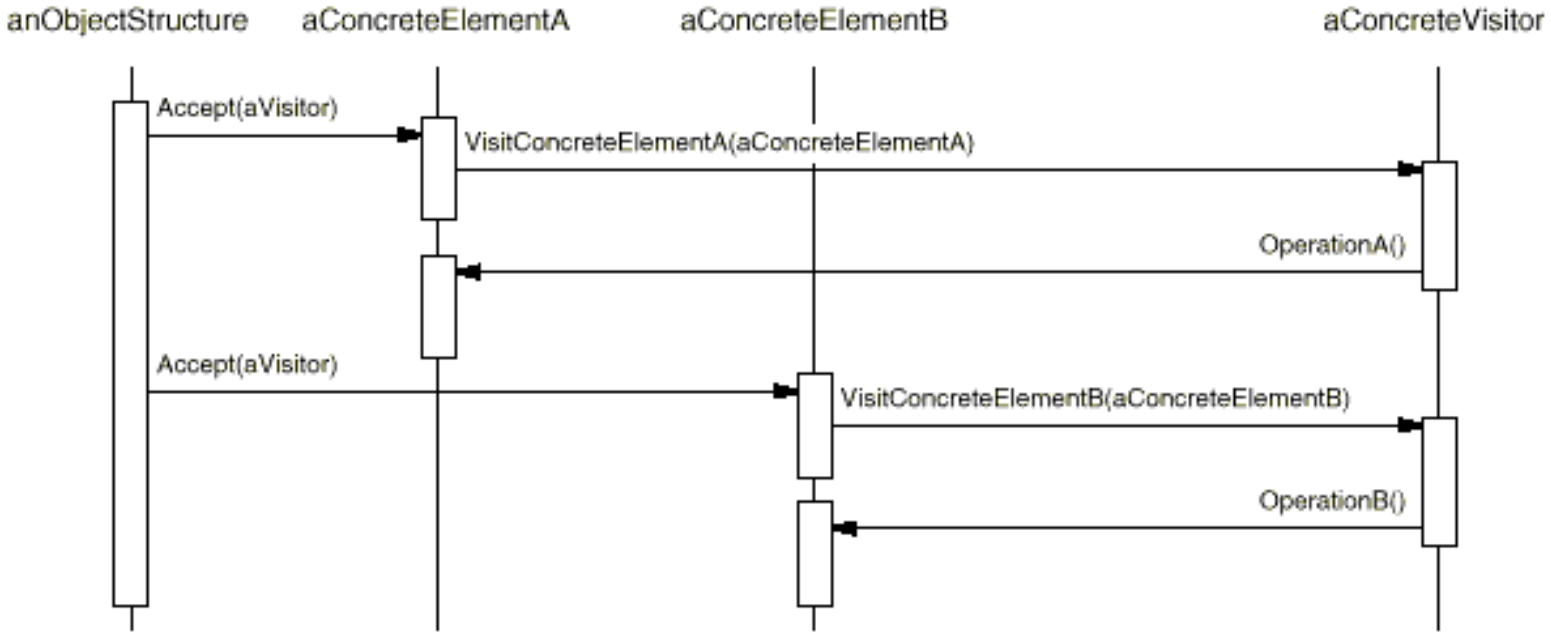
The Visitor Pattern

- Structure



The Visitor Pattern

- Collaborations



The Visitor Pattern

- Consequences
  - Benefits
    - 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.

Double-Dispatch

- The Visitor pattern allows you to add operations to classes without changing them using a technique called *double-dispatch*
- Single-Dispatch
  - ⇨ The actual method invoked depends on the name of the request (method signature) and the type of the receiver object
  - ⇨ For example, calling foo() on a object of Type X, invokes the foo() method of X
  - ⇨ The actual underlying type will be discovered through polymorphism
  - ⇨ This is the standard technique used in languages like Java and C++
- Double-Dispatch
  - ⇨ The actual method invoked depends on the name of the request and the types of two receivers

Double-Dispatch

- For example, consider an object of type Visitor1 calling accept(Visitor1) on an object of Type ElementA:
  - The Visitor1 object dispatches a call to the accept(Visitor) method of ElementA
  - The accept(Visitor) method of ElementA dispatches a call back to the visitor (Visitor1), invoking the visit(ElementA) method of Visitor1 and passing itself as an argument.
  - This round trip effectively picks up the right type of Element, ensuring that the correct visit() method of the Visitor object is called
- Effectively, then, the method invoked depends on the request name (accept(Visitor)), the type of the Element object (ElementA) and the type of the Visitor object (Visitor1)

Visitor With Composite Example

- Here's a example of using Visitor with the Composite pattern
- First, the Component class of Composite. Note the abstract accept() method.

```
public abstract class Component {
  protected String name;

  public Component(String name) {this.name = name;}

  public String getName() { return name; }
  public void setName(String name) { this.name = name; }

  public abstract double getPrice();

  public abstract void accept(ComponentVisitor v);
}
```

Visitor With Composite Example (Continued)

- Next, a leaf component called Widget. Note the implementation of the accept() method.

```
public class Widget extends Component {
  protected double price;

  public Widget(String name, double price) {
    super(name);
    this.price = price;
  }

  public void setPrice(double price) { this.price = price; }
  public double getPrice() { return price; }

  public void accept (ComponentVisitor v) { v.visit(this); }
}
```

Visitor With Composite Example (Continued)

- Finally, a composite component called WidgetAssembly. Again, note the implementation of the accept() method.

```
public class WidgetAssembly extends Component
  protected Vector components;

  public WidgetAssembly (String name) {
    super(name);
    components = new Vector();
  }

  public void addComponent (Component c) {
    components.addElement(c);
  }
```

Visitor With Composite Example (Continued)

```
  public void removeComponent (Component c) {
    components.removeElement(c);
  }

  public double getPrice() {

    double totalPrice = 0.0;

    Enumeration e = components.elements();
    while (e.hasMoreElements()) {
      totalPrice += ((Component) e.nextElement()).getPrice();
    }
    return totalPrice;
  }

  public void accept (ComponentVisitor v) { v.visit(this); }
}
```

Visitor With Composite Example (Continued)

- Here is the abstract superclass of the Visitor hierarchy. This could also be an interface.

```
public abstract class ComponentVisitor {

  public abstract void visit(Widget w);
  public abstract void visit(WidgetAssembly wa);

}
```

- Note that in order for the double-dispatch of Visitor to work properly, if we add any new subclasses of Component, we must add new visit() methods to ComponentVisitor

Visitor With Composite Example (Continued)

- Next, a simple visitor which just notes each Component object visited

```
public class SimpleVisitor extends ComponentVisitor {

  public SimpleVisitor() {}

  public void visit (Widget w) {
    System.out.println("Visiting a Widget");
  }

  public void visit (WidgetAssembly wa) {
    System.out.println("Visiting a WidgetAssembly");
  }

}
```

Visitor With Composite Example (Continued)

- And a visitor that does some comparative shopping!

```
public class PriceVisitor extends ComponentVisitor {
  private double maxPrice;

  public PriceVisitor(double maxPrice) {
    this.maxPrice = maxPrice; }

  public void visit (Widget w) {
    double price = w.getPrice();
    if (price > maxPrice)
      System.out.println("Don't Buy! Widget price of " +
        price + " exceeds maximum price (" + maxPrice + ").");
    else
      System.out.println("Buy! Widget price of " + price +
        " is less than maximum price (" + maxPrice + ").");
  }
```

Visitor With Composite Example (Continued)

```
  public void visit (WidgetAssembly wa) {
    double price = wa.getPrice();
    if (price > maxPrice)
      System.out.println("Don't Buy! WidgetAssembly price of " +
        price + " exceeds maximum price (" + maxPrice + ").");
    else
      System.out.println("Buy! WidgetAssembly price of " +
        price + " is less than maximum price (" +
        maxPrice + ").");
  }

}
```

Visitor With Composite Example (Continued)

- Visitor test program

```
public class VisitorTest {

  public static void main (String[] args) {

    // Create some widgets.
    Widget w1 = new Widget("Widget1", 10.00);
    Widget w2 = new Widget("Widget2", 20.00);
    Widget w3 = new Widget("Widget3", 30.00);

    // Add then to a widget assembly.
    WidgetAssembly wa = new WidgetAssembly("Chassis");
    wa.addComponent(w1);
    wa.addComponent(w2);
    wa.addComponent(w3);
```

Visitor With Composite Example (Continued)

```
    // Visit some nodes with a SimpleVisitor.
    SimpleVisitor sv = new SimpleVisitor();
    w1.accept(sv);
    w2.accept(sv);
    w3.accept(sv);
    wa.accept(sv);

    // Visit some nodes with a PriceVisitor.
    PriceVisitor pv = new PriceVisitor(25.00);
    w1.accept(pv);
    w2.accept(pv);
    w3.accept(pv);
    wa.accept(pv);

  }

}
```

Visitor With Composite Example (Continued)

- Test program output

```
Visiting a Widget
Visiting a Widget
Visiting a Widget
Visiting a WidgetAssembly
Buy! Widget price of 10.0 is less than maximum price (25.0).
Buy! Widget price of 20.0 is less than maximum price (25.0).
Don't Buy! Widget price of 30.0 exceeds maximum price (25.0).
Don't Buy! WidgetAssembly price of 60.0 exceeds maximum price
(25.0).
```