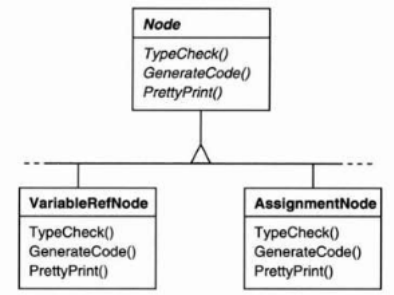
Visitor

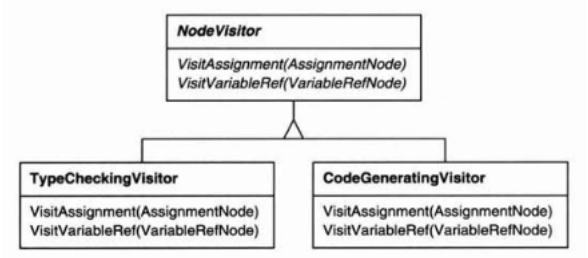
Visitor

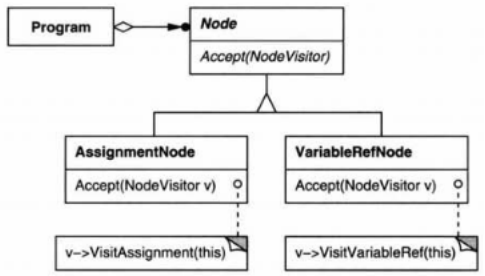
- Intent
  - Represent an operation to be performed on the elements of an object structure.
  - Visitor lets you define a new operation without changing the classes of the elements on which it operates.
- Motivation
  - Compiler operations on abstract syntax trees

Visitor



- Nodes are usually stable
- Operations are scattered over nodes
  - Hard to maintain and add new operations





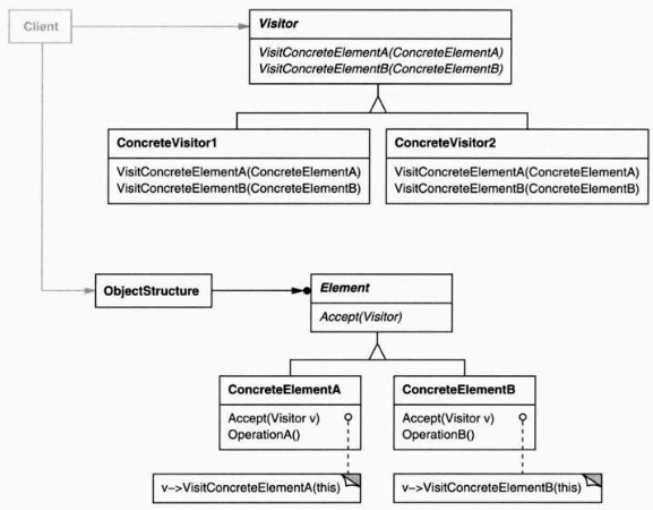
Visitor

- You define two class hierarchies:
  - one for the elements being operated on (the Node hierarchy)
  - one for the visitors that define operations on the elements (the NodeVisitor hierarchy).
- Create a new operation by adding a new subclass to the visitor class hierarchy.
- As long as we don't have to add new Node subclasses, we can add new functionality simply by defining new NodeVisitor subclasses.

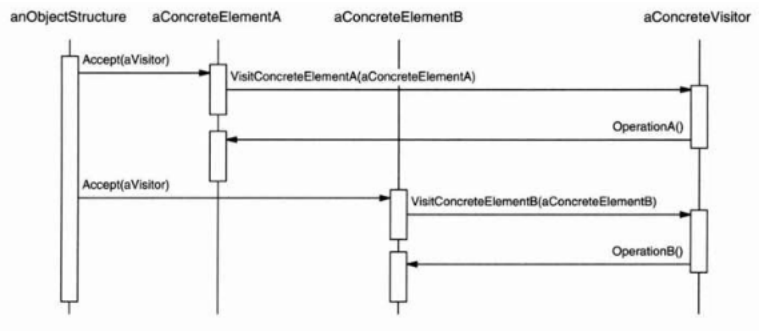
Applicability

- 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.
  - 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.

Structure



Collaborations



Consequences

- Visitor makes adding new operations easy.
- A visitor gathers related operations and separates unrelated ones.
- Adding new ConcreteElement classes is hard.
- Accumulating state.
  - Visitors can accumulate state as they visit each element in the object structure.
  - Without a visitor, this would be uglier.
- Breaking encapsulation.
  - Often, you need to provide public operations that access a ConcreteElement's internal state

Visitor vs. Iterator

- Visiting across class hierarchies.
  - An iterator 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.

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

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

Implementation

```
class Visitor {
public:
  virtual void VisitElementA(ElementA*);
  virtual void VisitElementB(ElementB*);
  // and so on for other concrete elements protected:
  Visitor();
};

class Element {
public:
  virtual ~Element();
  virtual void Accept(Visitor&) = 0;
protected:
  Element();
};
```


Implementation

```
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);
  }
};
```

Implementation

```
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);
}
```


Implementation

- Who is responsible for traversing the object structure?
  - the object structure
  - the visitor
    - requires duplicating the traversal code in each ConcreteVisitor for each aggregate ConcreteElement.
  - a separate iterator object

Sample Code

```
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;
};
```



```
class FloppyDisk: public Equipment {
public:
  FloppyDisk(const char*);
  virtual ~FloppyDisk();
  virtual Watt Power();
  virtual Currency NetPrice();
  virtual Currency DiscountPrice();
};

class CompositeEquipment: public Equipment {
public:

  virtual ~CompositeEquipment();
  virtual Watt Power();
  virtual Currency NetPrice();
  virtual Currency DiscountPrice();
  virtual void Add(Equipment*);
  virtual void Remove(Equipment*);
  virtual Iterator* CreateIterator();
protected:
  CompositeEquipment(const char*);
private:
  List _equipment;
};
```

```
class Chassis: public CompositeEquipment {
public:
  Chassis(const char*);
  virtual ~Chassis();
  virtual Watt Power();
  virtual Currency NetPrice();
  virtual Currency DiscountPrice();
};

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();
};
```



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

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

```
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();
}
```

Changing the pricing policy?

Price: net price of all simple equipment (e.g., floppies) and the discount price of all composite equipment (e.g., chassis and buses).


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

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