AP6
Le pattern "Visitor"

Petit Complément Utile
pour l'Interpréteur…

IUT 2 Grenoble
Université Pierre-Mendès-France
Informatique

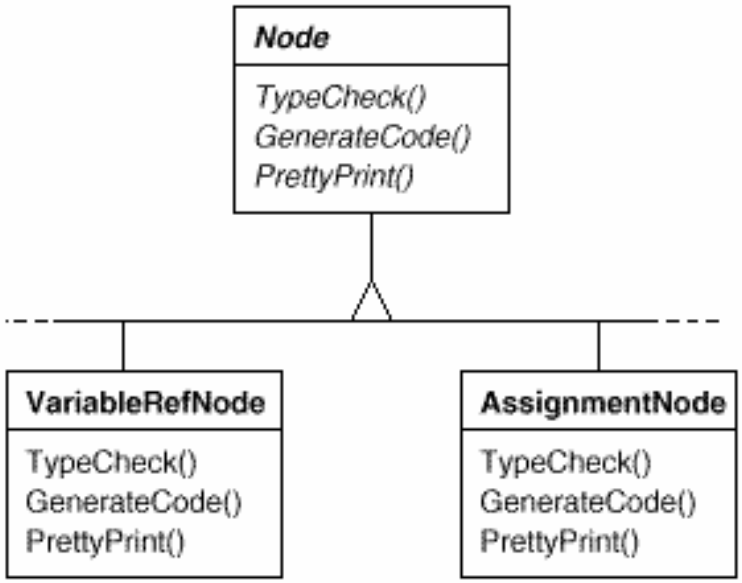
Objectif, motivation (1)

- Ce pattern permet de représenter dans une seule classe une opération qui doit être effectuée sur tous les éléments d'une structure "composite" (*cf pattern composite*)

- Un objet "visitor" permet de définir une telle opération sans avoir besoin de changer le code des objets sur lesquels il va opérer.

Objectif, motivation (2)

- Ce pattern est particulièrement adapté pour implémenter les différentes opérations que l'on doit réaliser sur un arbre abstrait tel que celui manipulé dans notre interpréteur :
  - vérification de type
  - génération de code
  - évaluation (getValeur)
  - impression (afficher)
  - *etc…*



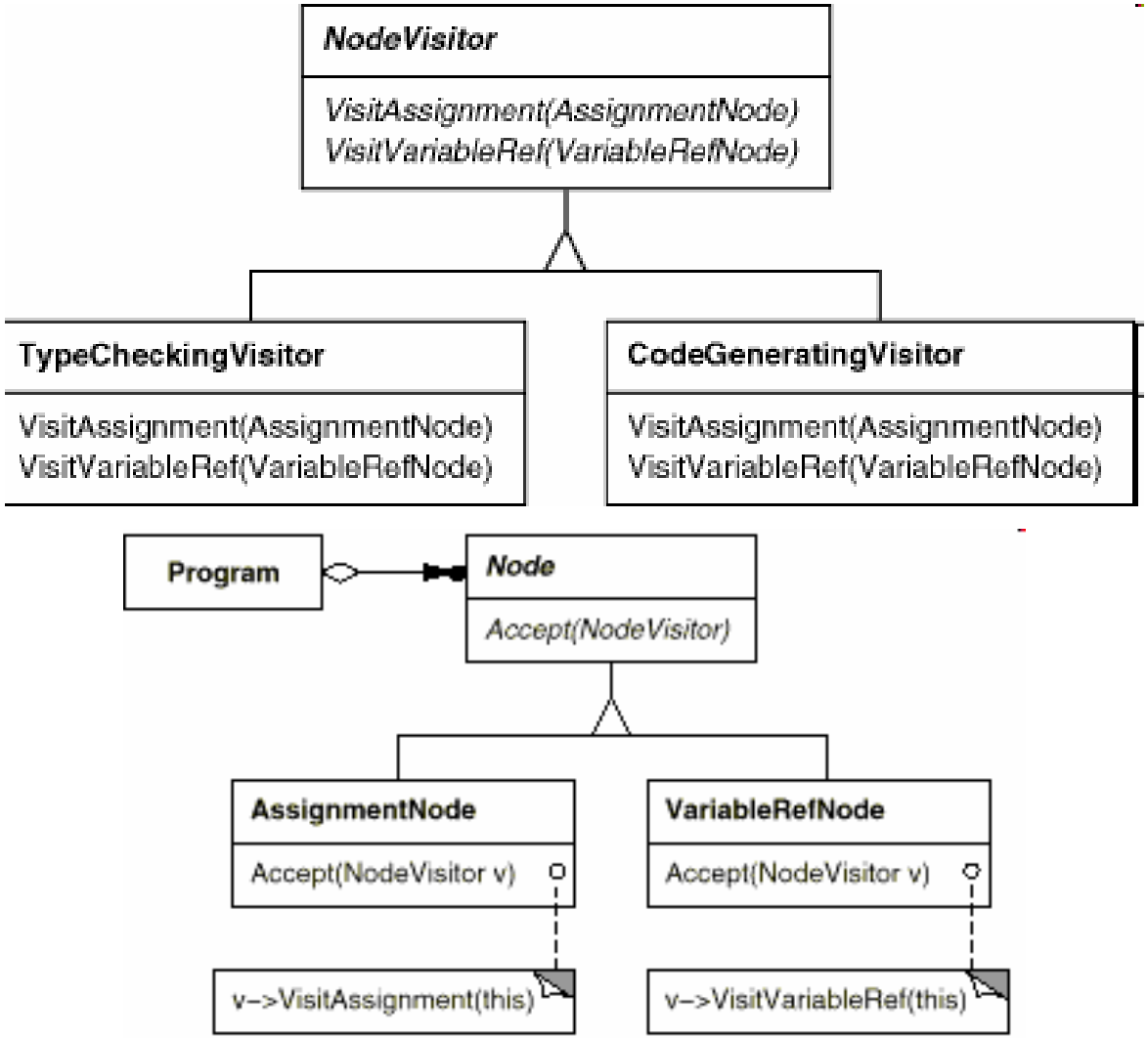
Implémentation des opérations sans le pattern visitor

- Problème : si chaque opération est implémentée dans chaque type de nœud de l'arbre, tout le code représentant une même opération se trouve "éparpillé" dans plusieurs classes…
- Le pattern Visitor permet de résoudre ce problème !

Objectif, motivation (3)

- Si l'on met en œuvre le pattern visitor, chaque type d'opération sera réalisée par un objet "visitor" qui contiendra tout le code propre à cette opération :
  - Il y aura ainsi un objet **TypeCheckingVisitor**, un objet **CodeGeneratingVisitor**, **afficherVisitor**, …*etc*.
- Tous ces objets "**visitor**" hériteront d'une même classe mère, abstraite : **NodeVisitor**.
- La classe **NodeVisitor** devra définir une méthode pour chaque type de nœud de l'arbre :
  - Une méthode **visitAssignment** pour visiter une nœud de type **AssignmentNode**, Une méthode **visitVariableRef** pour visiter un nœud de type **VariableRefNode**, etc…
- Ces méthodes seront implémentées dans chaque type d'objet **visitor** pour réaliser l'opération correspondante

Objectif, motivation (3)



Les "participants" du pattern (1)

- **Visitor** (ex : **NodeVisitor**)
  - Définit une méthode de "visite" pour chaque type de nœud de l'arbre.
  - Le nom de la méthode et sa signature (type du paramètre) identifient le type de nœud qui envoie la requête de visite à l'objet visiteur. Cela permet au visiteur de déterminer le type de l'élément visité. Le visiteur pourra alors accéder à l'élément en utilisant l'interface qui lui est propre

Les "participants" du pattern (2)

- **ConcreteVisitor** (ex : **TypeCheckingVisitor**)
  - Implémente chaque méthode déclarée par la classe **Visitor**.
  - Chaque méthode réalise un morceau de l'algorithme pour le type de nœud traité

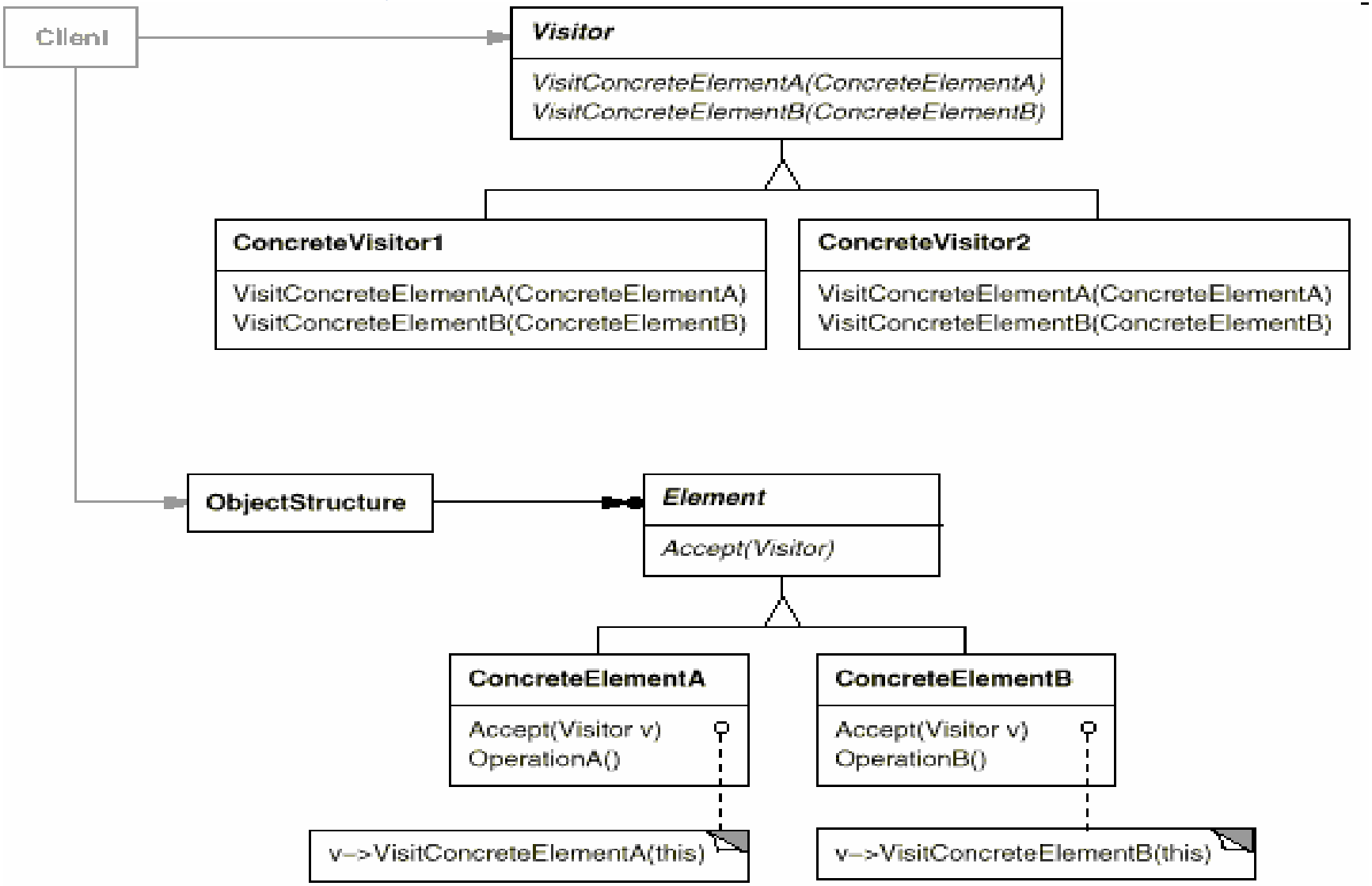
- **Element** (ex : **Node**)
  - Définit une méthode **accept** qui reçoit un **Visitor** en paramètre

Les "participants" du pattern (3)

- **ConcreteElement** (ex : **AssignmentNode**)
  - Implémente la méthode **accept** qui reçoit un **Visitor v** en paramètre et invoque la méthode de ce visiteur correspondant au type de l'élément. Exemple, pour un nœud **AssignmentNode** :

    v->visitAssignment(this)
- **ObjectStructure** (ex : **Program**)
  - Peut énumérer ses éléments
  - Peut fournir une interface pour permettre aux visiteurs de parcourir ses éléments
  - Peut être un "*composite*" ou une collection

Structure du Pattern



Conséquences du pattern Visitor

- Il est facile d'ajouter de nouvelles opérations à la structure composite
- Un visiteur rassemble des traitements qui participent à la même opération : facilite la maintenance de cette opération
- Il est plus compliqué de rajouter un nouveau type d'élément dans la structure composite (il faut modifier tous les visiteurs !)
- Un visiteur peut réaliser une opération qui nécessite d'accumuler un résultat… Ces données sont stockées dans le visiteur
- Un visiteur doit pouvoir accéder aux données des éléments de la structure composite… Ces éléments doivent fournir une interface pour cela, au risque de mettre à mal le principe d'encapsulation

Exemple d'implémentation (1)

- On considère l'exemple d'une structure composite pour représenter des équipements électroniques (chassis, bus, floppyDisk, …).
- La classe abstraite, mère des équipements serait :

```
class Equipment {
public:
    Equipment(const char*);
    virtual ~Equipment();
    const char* Name() { return _name; }
    virtual Watt Power();
    virtual Currency NetPrice();
    virtual Currency DiscountPrice().
    virtual void Accept(EquipmentVisitor&);

private:
    const char* _name;

};
```

Exemple d'implémentation (2)

- La classe abstraite, mère de tous les visiteurs, serait :

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

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

    // et ainsi de suite pour chaque sous-class concrète
    // représentant un type d'élément

};
```

Exemple d'implémentation (3)

- Les sous-classes représentant les types de composants devraient implémenter la méthode accept :

```
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);
}
```

Exemple d'implémentation (4)

- Un visiteur pour calculer le prix d'un équipement serait écrit ainsi:

```
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();
}
```

Exemple d'implémentation (5)

- Un visiteur pour réaliser un inventaire serait écrit ainsi :

```
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);
}
```

Exemple d'implémentation (6)

- Le programme principal pour lancer un inventaire serait :

```
int main() {
  Equipment* component;
  InventoryVisitor visitor;

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