ÉTS
Le génie pour l'industrie

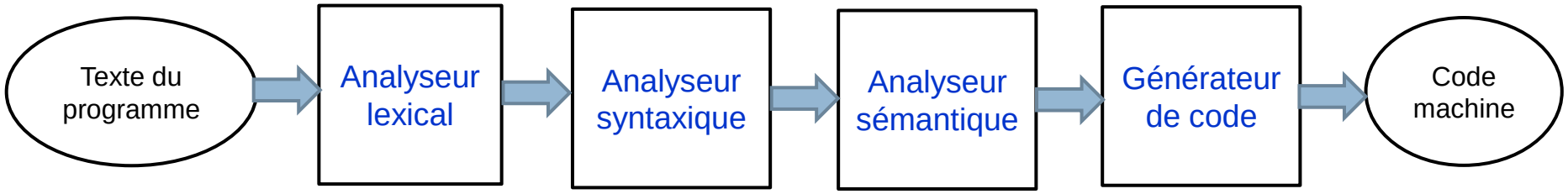
LOG121
Conception orientée objet

Patron Visiteur

Enseigante: Souad Hadjres

Exemple de problème de conception

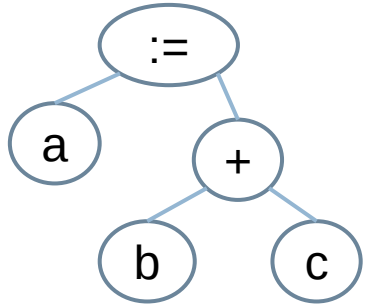
- Un compilateur représente un programme par un arbre syntaxique abstrait
- Les étapes réalisées par un compilateur



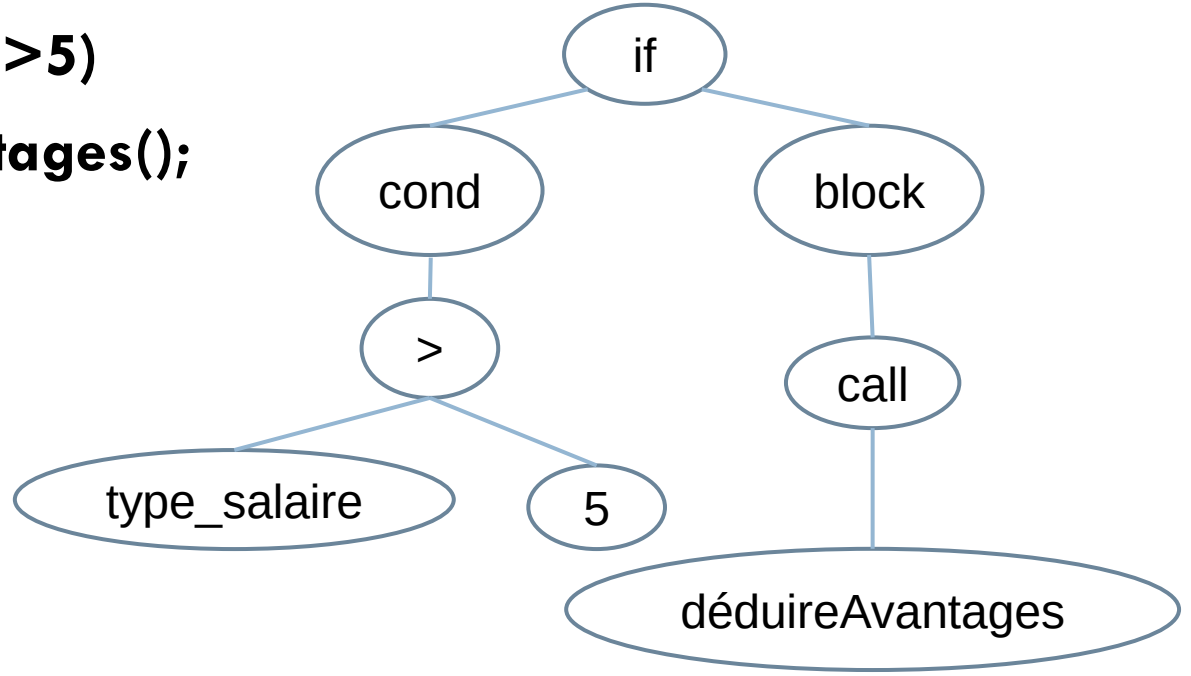
Exemple de problème de conception

- Exemples d'arbres syntaxiques

a := b+ c




```
If (type_salaire>5)
   déduireAvantages();
```



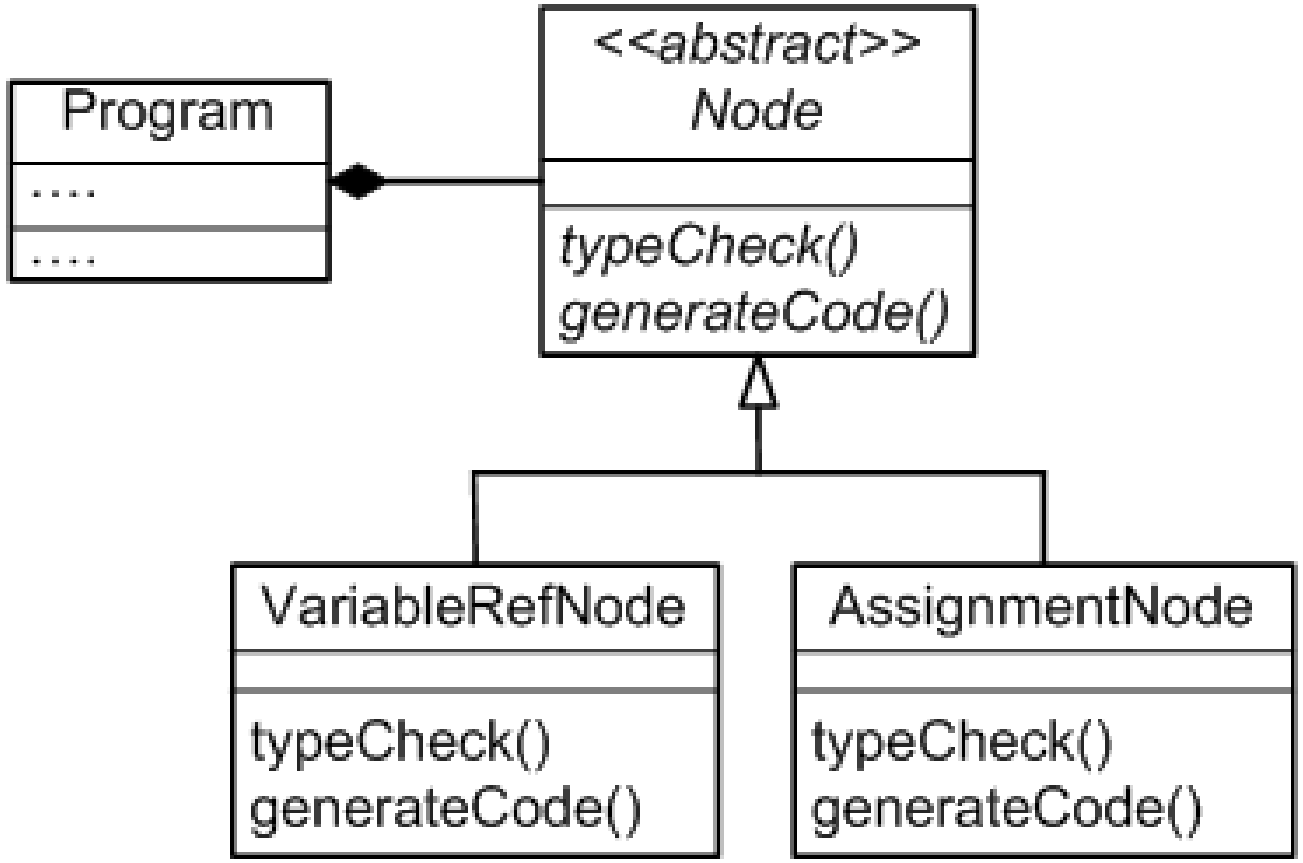
Exemple de problème de conception

- Le compilateur a besoin d'exécuter différentes opérations sur les nœuds de l'arbre.*
  - Vérifier le type, vérifier l'initialisation, générer du code machine, etc.
- L'arbre peut contenir différents types de nœuds
  - Des instructions d'affectation, des variables, des expressions arithmétiques, etc.
- Les opérations du compilateur doivent traiter les nœuds différemment selon leur type.

*Exemple tiré du livre "Design Patterns: Elements of Reusable Object-Oriented Software" de GoF.

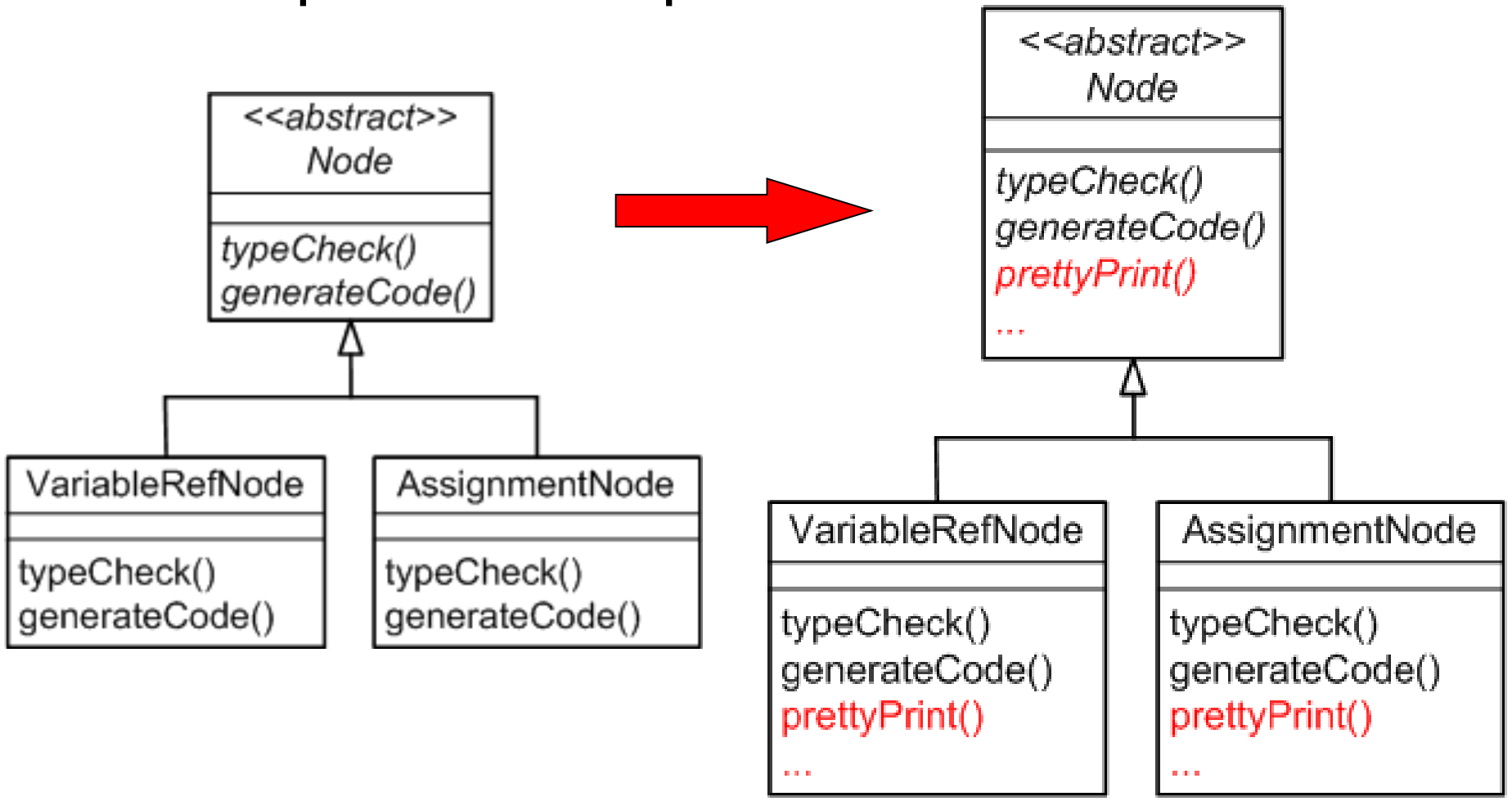
Exemple de problème de conception

- Une première conception



Exemple de problème de conception

- Pour ajouter un comportement



Exemple de problème de conception

- Problèmes avec cette conception
  - L'ajout d'une opération nécessite la modification et recompilation de toutes les classes de la hiérarchie.
  - L'implémentation de toutes ces opérations à travers les classes de la hiérarchie donne une conception difficile à maintenir et à modifier.

Solution au problème

- Séparer les opérations des nœuds sur lesquels elles s'appliquent
- Encapsuler chaque opération dans un objet appelé visiteur
  - Par exemple une classe TypeCheckVisitor pour l'opération typeCheck
- Ce visiteur est passé aux nœuds lorsqu'ils sont parcourus
  - Les nœuds définissent une méthode accept() qui accepte le visiteur en paramètre

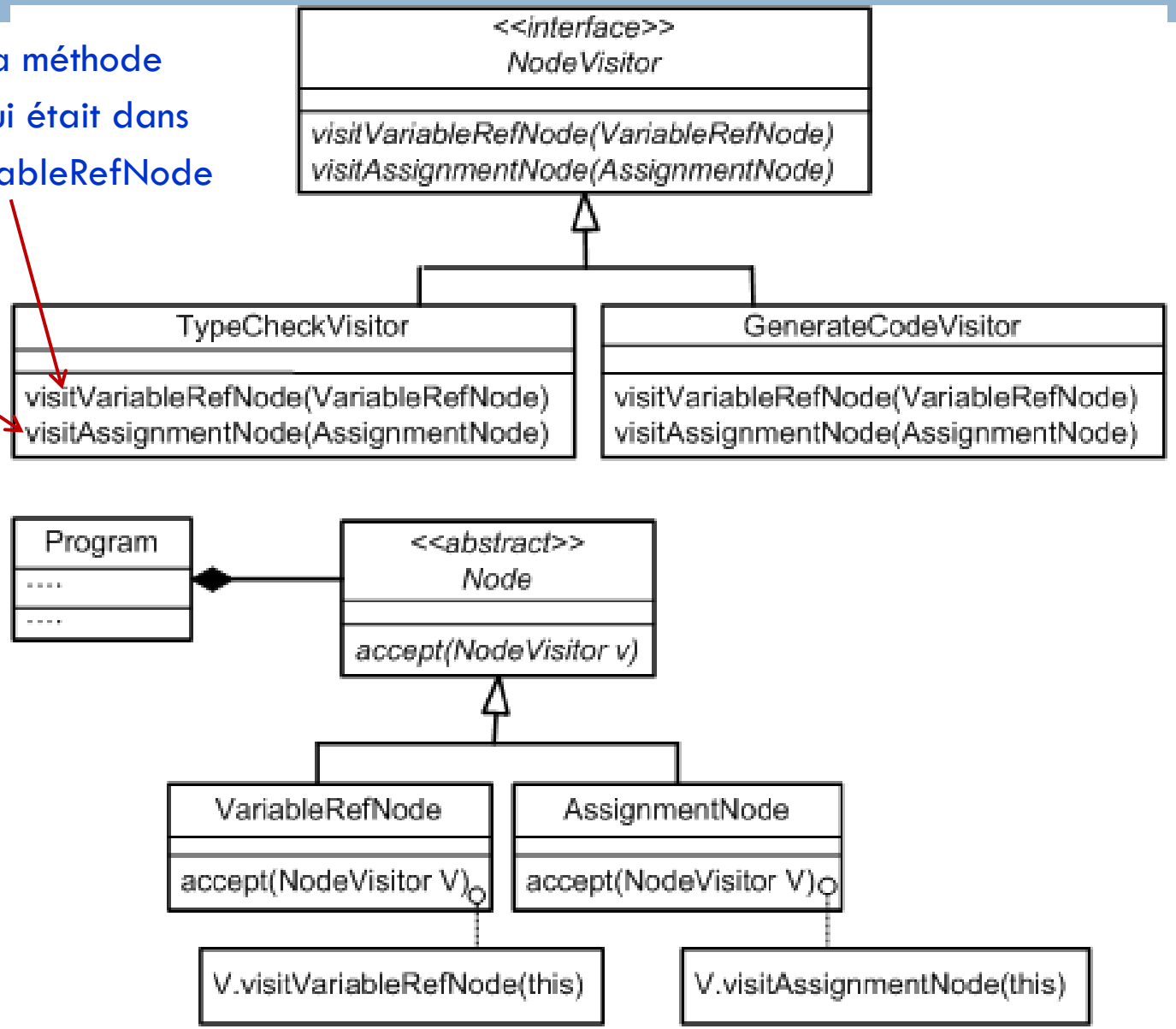
Solution au problème

- Lorsqu'un nœud accepte un objet visiteur, il lui transmet la requête qui inclut le nœud comme paramètre.
- Le visiteur exécute l'opération pour ce nœud.
- Un visiteur regroupe les différentes implémentations d'une opération pour les différents nœuds de la hiérarchie.
- L'interface commune Visitor des objets visiteurs définit une méthode pour visiter chaque type de nœud.
  - visitVariableRefNode(VariableRefNode)
  - visitAssignmentNode(AssignmentNode)

Solution au problème

Implémente la méthode typeCheck qui était dans le nœud VariableRefNode

Implémente la méthode typeCheck qui était dans le nœud AssignmentNode



Le patron Visiteur

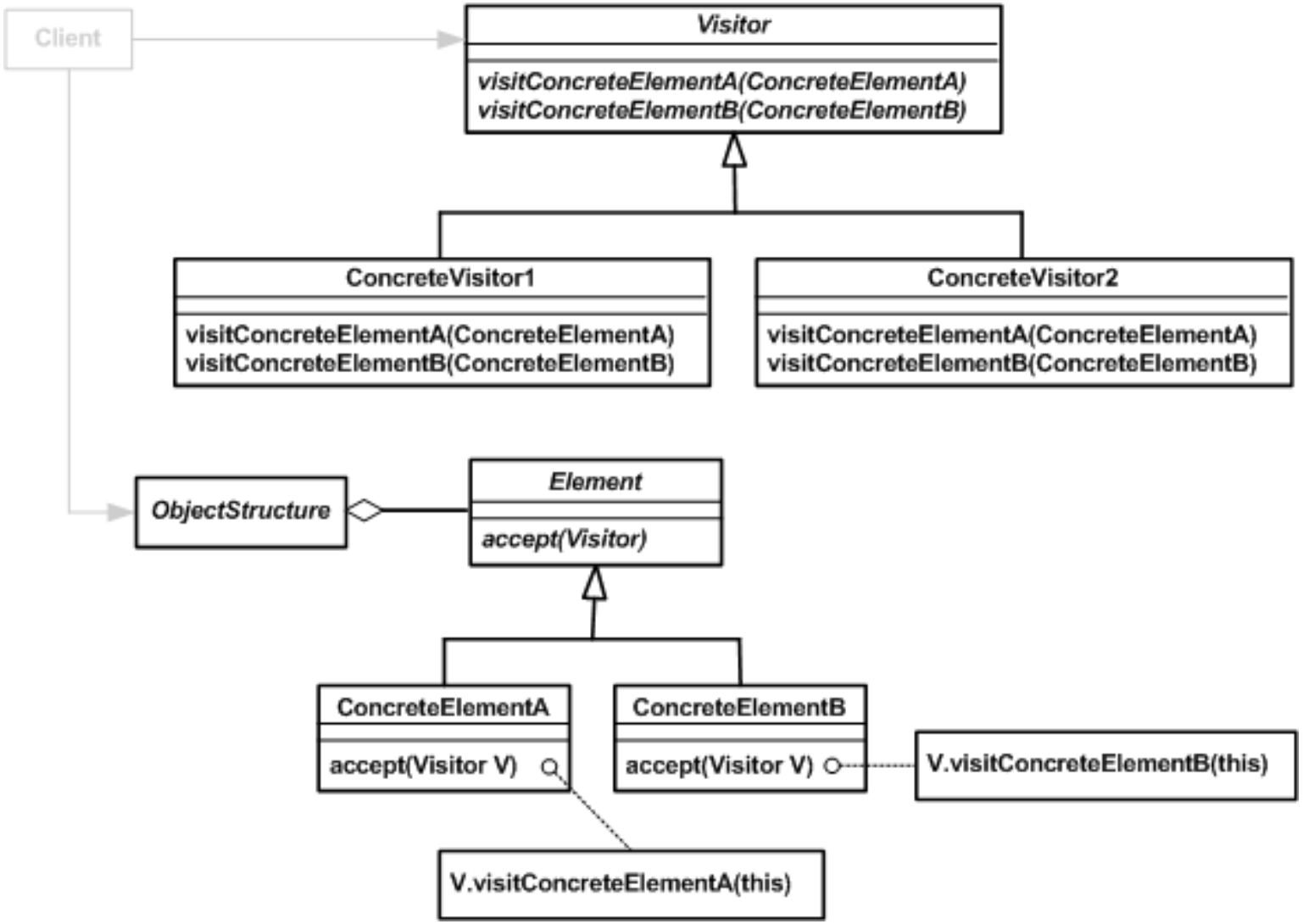
- Contexte
  - Une structure d'objets contient plusieurs types d'objets. On veut exécuter des opérations sur ces objets et l'exécution de ces opérations varie selon le type de l'objet.
  - Les classes définissant la structure changent rarement.
  - L'ensemble des opérations n'est pas stable et risque d'évoluer avec le temps.

Le patron Visiteur

- Solution
  - Définir une interface (Visitor) qui déclare une méthode pour visiter chaque classe concrète (ConcreteElement) de la structure d'objet.
  - Chaque classe concrète (ConcreteElement) définit une méthode (accept) qui reçoit un visiteur en paramètre et invoque la bonne méthode de ce visiteur.
  - Pour implémenter une opération, définir une classe qui implémente le type interface (Visitor) et qui regroupe les différentes implémentations de cette opération pour chaque type concret d'élément (ConcreteElement).

Le patron Visiteur

- La structure du patron dans GoF



Le patron Visiteur

Nom dans le patron	Nom dans l'exemple du compilateur
Element	Node
ConcreteElement	VariableRefNode, AssignmentNode
Visitor	NodeVisitor
ConcreteVisitor	TypeCheckVisitor, GenerateCodeVisitor

Le patron Visiteur

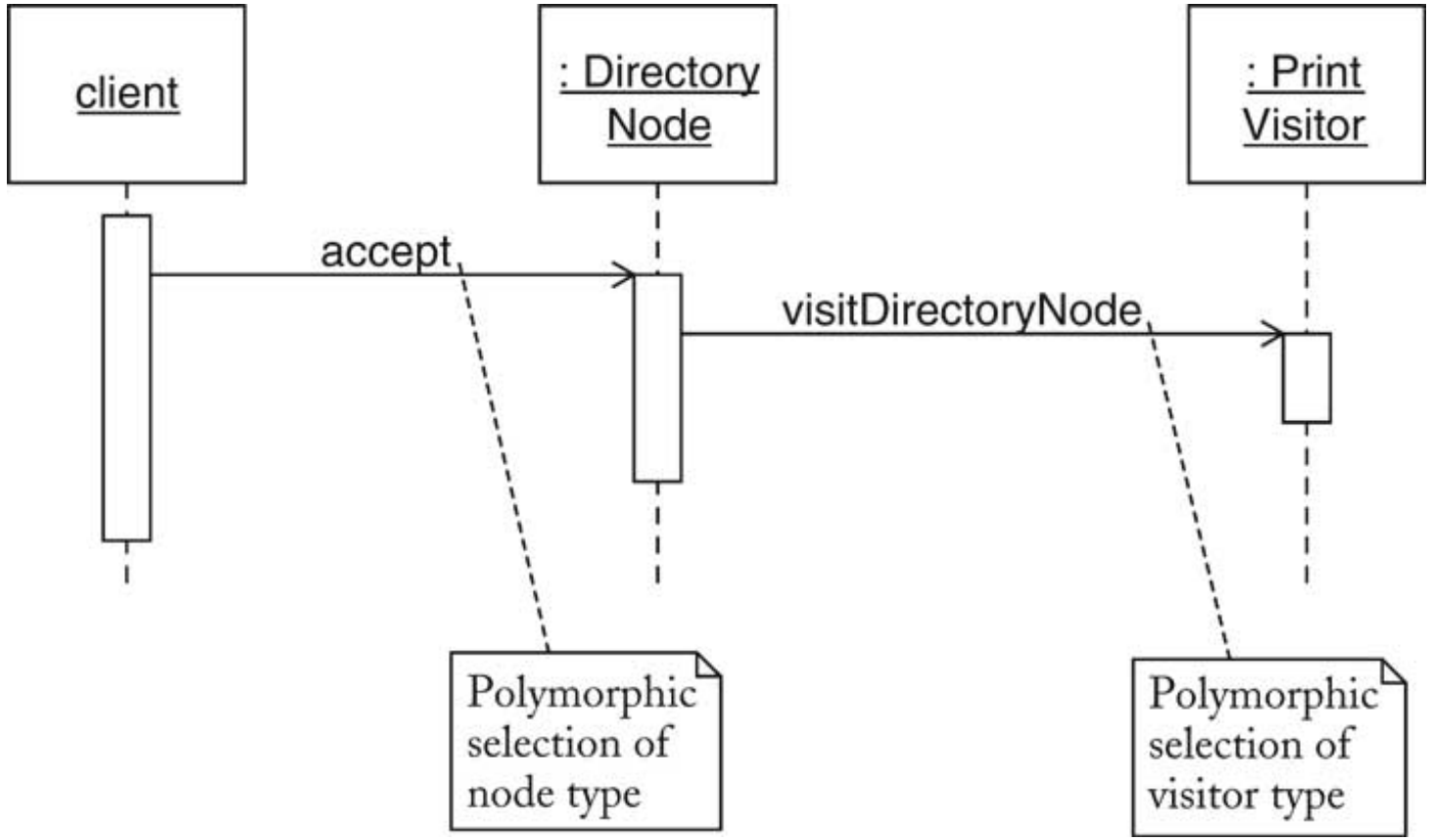
- « *Double dispatch* »: on utilise deux appels polymorphiques.

```
Node node = new VariableRefNode();
NodeVisitor v = new TypeCheckVisitor();
node.accept(v);
```

  - 1er polymorphisme
    - node.accept correspond à VariableRefNode.accept
    - Sélection polymorphique du type de nœud
  - 2ème polymorphisme
    - La méthode accept appelle v.visitVariableRefNode et v est un TypeCheckVisitor, alors on appelle TypeCheckVisitor. visitVariableRefNode
    - Sélection polymorphique du type de visiteur

Le patron Visiteur

- « *double dispatch* »: on utilise deux appels polymorphiques



Le patron Visiteur

- Quelles sont les conséquences d'utilisation de ce patron?
  - Il permet d'ajouter des méthodes sans changer une structure d'objets
  - Il facilite l'ajout d'une opération
  - Un visiteur regroupe des opérations et tout ce qui leur est nécessaire (ex: structure de données)
  - Il est difficile d'ajouter de nouveaux types d'éléments concrets (ConcreteElement)
  - L'encapsulation des objets (ConcreteElement) est violée

Question

- Quel patron peut être jumelé avec le patron Visiteur?
  - On peut utiliser le Visiteur pour appliquer des opérations sur une structure d'objets définie par la patron Composite

Question

- Patron composite versus patron visiteur

- Composite:

- structurel

- Permet de créer des structures hiérarchiques arborescentes récursives

- Permet d'appliquer la même opération à un ensemble d'éléments partageant la même interface

- Visiteur:

- comportemental

- Permet à un objet visiteur de «visiter» chaque élément d'une hiérarchie structurelle pour appliquer une opération sur cet élément (les opérations diffèrent d'un élément à l'autre)

```java
/**
* Élément visité (IElement)
*/
public interface InterfaceElement {
    public void print();
    public void accept(InterfaceVisitor v);
}
```

```java
/**
* Fichier
*/
public class Fichier implements InterfaceElement {
    private String nom;
    private int taille;
    public Fichier(String nom, int taille) {
        this.nom = nom;
        this.taille = taille;
    }

    public int getTaille() {
        return this.taille;
    }
    public void print() {
        System.out.println(" " + this.nom);
    }
    public void accept(InterfaceVisitor v) {
        v.visit(this);
    }
}
```

```java
/**
* Dossier
*/
public class Dossier implements InterfaceElement {
    private String nom;
    private Vector<InterfaceElement> children;

    public Dossier(String nom) {
        this.nom = nom;
        this.children = new Vector<InterfaceElement>();
    }
    public void add(InterfaceElement e) {
        this.children.add(e);
    }
    public void remove(InterfaceElement e) {
        this.children.remove(e);
    }
    public InterfaceElement get(int i) {
        return this.children.get(i);
    }
    public int size() {
        return this.children.size();
    }

    public void print() {
        System.out.println("Dossier (" + this.nom + ")");
        for (InterfaceElement e : this.children) {
            e.print();
        }
    }
    public void accept(InterfaceVisitor v) {
        v.visit(this);
    }
}
```

```java
/**
* Visiteur (IVisitor)
*/
public interface InterfaceVisitor {
    public void visit(Dossier d);
    public void visit(Fichier f);
}
```

```java
/**
* Visite la hiérarchie et calcul la taille totale
* des éléments contenus par la racine choisie (ConcreteVisitor)
*/
public class TailleVisitor implements InterfaceVisitor {
    private int taille;
    public TailleVisitor() {
        this.taille = 0;
    }
    public int getTaille() {
        return this.taille;
    }

    public void visit(Dossier d) {
        // Visite les éléments contenus dans le dossier

        InterfaceElement elem;
        for (int i = 0; i < d.size(); i++) {
            elem = d.get(i);
            elem.accept(this);
        }
    }
    public void visit(Fichier f) {
        // Additionne la taille du fichier au total
        this.taille += f.getTaille();
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        Dossier d1 = new Dossier("Racine");
        d1.add(new Fichier("bonjour.txt", 100));
        d1.add(new Fichier("toto.txt", 55));
        d1.add(new Fichier("titi.txt", 85));
        Dossier d2 = new Dossier("Doc");
        d2.add(new Fichier("guide.doc", 235));
        d1.add(d2);
        Dossier d3 = new Dossier("Temp");
        d1.add(d3);
        TailleVisitor visitor1 = new TailleVisitor();
        d1.accept(visitor1);
        // 100 + 55 + 85 + 235 = 475
        System.out.println("Taille du dossier Racine = " + visitor1.getTaille());
        TailleVisitor visitor2 = new TailleVisitor();
        d2.accept(visitor2);
        // 235
        System.out.println("Taille du dossier Doc = " + visitor2.getTaille());
        TailleVisitor visitor3 = new TailleVisitor();
        d3.accept(visitor3);
        // 0
        System.out.println("Taille du dossier Temp = " + visitor3.getTaille());
    }
}
```