Visitor : quand

- Nous avons une (ou plusieurs!) hiérarchie d'objets, utilisés dans une structure complexe
  - Arbre, graphe, …
- **On externalise des opérations**
  - pour ne pas trop compliquer la structure
  - parce qu'on ne peut pas modifier la structure (module fermé)
- Ces opérations doivent pouvoir exécuter un code qui dépend du type **réel** des objets
  - Sans avoir plein de `if` et/ou `instanceof` !
- Ces opérations ne sont pas connues par les objets de la structure
  - Nous ne connaissons ni ne voulons connaître le code du client
  - Et ce code peut changer en fonction du client

Visitor : exemple d'applicabilité

- On a une hiérarchie d'objets, on veut ajouter des méthodes de sérialisation pour obtenir différentes représentations XML.
- 2 solutions
  - Ajouter des méthodes toXml1(), toXml2(), … à chacune des classes de ma hiérarchie pour ensuite parcourir ma structure (comment?) et me reposer sur le polymorphisme
    - Impossible si je ne peux modifier les objets
    - Non souhaité si je veux que ma hiérarchie d'objets restent relativement simple (et ne pas la rouvrir à chaque nouvelle opération)
  - Créer une classe externe pour implémenter ce traitement
    - Avec plein de *instanceof* !

Visitor : comment

- On utilise une implémentation qui effectue un **double dispatch** :
  - on crée une interface Visitor qui contient une méthode visit(*X* x) différente par type **concret** *X* de la hiérarchie
  - le type le plus haut dans la hiérarchie a une méthode
    - abstract void accept(Visitor v)
  - Tous les types concrets de la hiérarchie **implémentent** cette méthode de la manière suivante :
    - void accept(Visitor v) {
      v.visit(this);
      }

La surcharge va permettre d'appeler la BONNE fonction

Visitor : comment

- Pour utiliser le visiteur, plutôt que d'appeler visit sur l'objet de la hiérarchie, on appelle accept de l'objet avec le visiteur en argument
  - ~~visitor.visit(object)~~ –› object.accept(visitor)
- *Attention* : même dans les méthodes visit, on n'appelle pas directement les méthodes visit sur l'objet, mais bien accept de l'objet sur visiteur (*i.e.* this)
  - ~~visit(object)~~ –› object.accept(this)

La surchage est static, pas le polymorphisme

Visitor : pourquoi ça marche

- Quand on appelle object.accept(visitor), la méthode accept choisie est celle du **type réel** de object
  - Polymorphisme classique
- Dans chaque type concret *X*, l'implémentation de accept() est codé avec
  - visitor.visit(this), appelle la *bonne* méthode visit(*X* x) car this est typé statiquement *X*
  - rappel : le choix de la méthode parmi les surcharges se fait en fonction du type déclaré

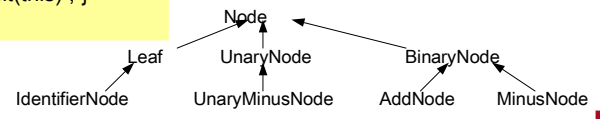
Visitor : exemple

```
public interface Visitor {
  void visit(AddNode node);
  void visit(MinusNode node);
  void visit(UnaryMinusNode node);
  void visit(IdentifierNode node);
}
```

```
public interface Node {
  void accept(Visitor visitor);
}
```

```
public class AddNode
    extends BinaryNode
{
  void accept(Visitor visitor) {
    visitor.visit(this) ; }
}
```

```
public class MyVisitor
    implements Visitor{
…
}
```



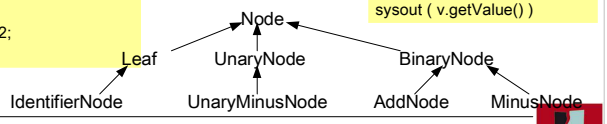
Visitor : exemple

```
public class EvalVisitor implements Visitor {
  private int value;
  private final Memory memory;
  EvalVisitor(Memory memory) {
     this.memory = memory ;
  }
  int getValue() { return value ; }

  void visit(AddNode node) {
    node.left.accept(this);
    int value1 = value;
    node.right.accept(this);
    int value2 = value;
    value = value1 + value2;
  }
  void visit(MinusNode node) {
    node.left.accept(this);
    int value1 = value;
    node.right.accept(this);
    int value2 = value;
    value = value1 - value2;
  }
```

```
  void visit(UnaryMinusNode node) {
    node.subNode.accept(this);
    value=-value;
  }
  void visit(IdentifierNode node) {
    value=memory.get(node.identifier);
  }
}
```

```
Expr e ;
…
v = new EvalVisitor(memory) ;
e.accept(v) ;
 …
sysout ( v.getValue() )
```



Prérequis : externaliser les opérations

```
public interface Operation {
  void doItForADog(Dog dog);
  void doItForACat(Cat cat);
}
```

```
public class Sound
    implements Operation {
{
   void doItForADog(Dog d) {
     sysout ("woof") ;
   }
   void doItForACat(Cat d) {
     sysout ("meow") ;
   }
};
```

```
Interface Animal { }
class Dog implements Animal {
    void letsDo(Operation v) {
        v.doItForADog(this) ;
    }
}
class Cat implements Animal { ... }
```

```
Animal a = new Cat();
Sound theSound;
a.letsDo(theSound);
```

Visitor : pour les pros !

- On a utilisé un champ pour transmettre un résultat :(
- On préférera paramétrer le visiteur par
  - le type de retour : R
  - un éventuel argument supplémentaire : P
  - l'exception lancée par les méthodes visit : E
- Les signatures sont modifiées :

```
public interface Visitor<R,P,E extends Throwable> {
  R visit(AddNode node, P parameter) throws E;
  R visit(MinusNode node, P parameter) throws E;
  R visit(UnaryMinusNode node, P parameter) throws E;
  R visit(IdentifierNode node, P parameter) throws E;
}
```

```
public class Minus extends BinaryOp {
  ....
 <R,P,E extends Throwable> R
    accept(Visitor<R,P,E> v,P p) throws E {
    return v.visit(this, p) ; }
```

```
public interface Node {
  ....
 <R,P,E extends Throwable> R
    accept(Visitor<R,P,E> v,P p) throws E ;
}
```


Visitor : astuce

- On veut pouvoir spécifier le même code pour tous les descendants d'un type de la hiérarchie
  - On remplace l'interface Visitor par un classe abstraite (mais sans méthode abstraite) où :
    - par défaut, une méthode visit appelle la méthode visit du supertype direct
    - la méthode visit du type le plus haut lève une AssertionError
- on ne change rien dans la hiérarchie, seul le visiteur change

Visiteur : variante

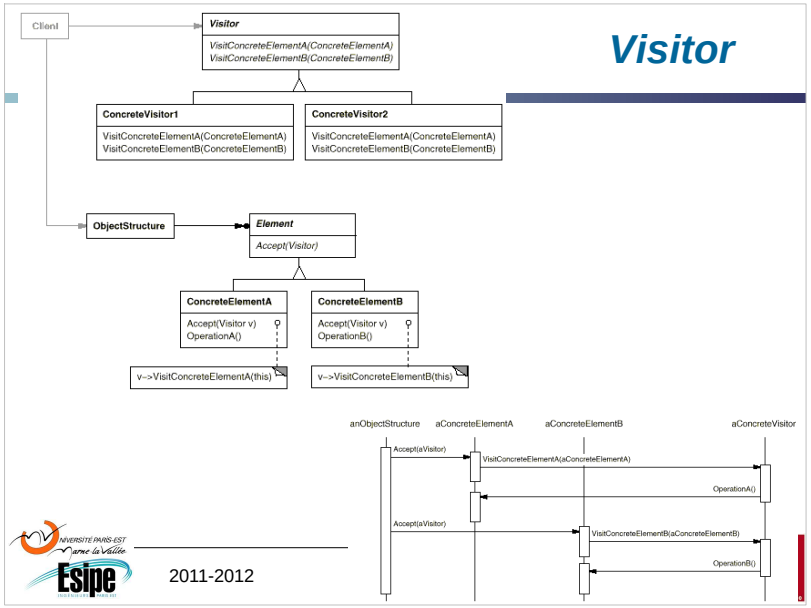
- Qui a la responsabilité du parcours de la structure ?
  - Le visiteur (comme dans les exemples abordés)
    - Le plus flexible
    - Risque de redondance de code ; oblige la structure à exposer les méthodes permettant le parcours
  - La structure
    - Factorisation naturelle de code
    - Simplifie les visiteurs mais moins flexible
  - Un itérateur externe

Visitor : check-list

Ou LES interfaces

- Vérifier que la hiérarchie de la structure d'objets sera « assez » stable; et que l'interface de ces classes fournit ce qui sera nécessaire aux visiteurs.
- Créer la classe de base Visitor avec une méthode visit(ElementXxx) pour chaque classe dérivée concrète d'Element
- Ajouter une méthode `accept(Visitor)` à la hiérarchie d'Element. L'implémentation standard est :
  `accept( Visitor v ) { v.visit( this ); }`
- La hiérarchie Element est juste couplée à l'interface Visitor. Les classes de la hiérarchie Visitor sont couplées à chaque classe de la hiérarchie Element.
- Chaque « operation » donne lieu à la création d'un Visiteur. L'implémentation des fonctions visit() doit s'appuyer sur l'interface public des classes Element's
- Le client crée les objets Visitor et les passe aux Element's en appelant accept()

Visitor



Visitor : conséquences

- « facile » d'ajouter une nouvelle opération
  - Un nouveau visiteur
- Sépare fortement les opérations indépendantes
  - Pas de couplage technique si pas de couplage fonctionnel
- Ajouter de nouvelles classes concrètes dans la structure est compliqué !
  - Nouvelle méthode visit + impl. Pour tous les visiteurs
- Visiteur est donc adapté quand
  - La structure ne change pas ou peu
  - La structure est "fermé"
  - Les opérations sur la structure changent souvent,
  - On veut découpler les opérations et la structure
- Le visiteur oblige souvent a étendre l'interface publique des objets de la structure