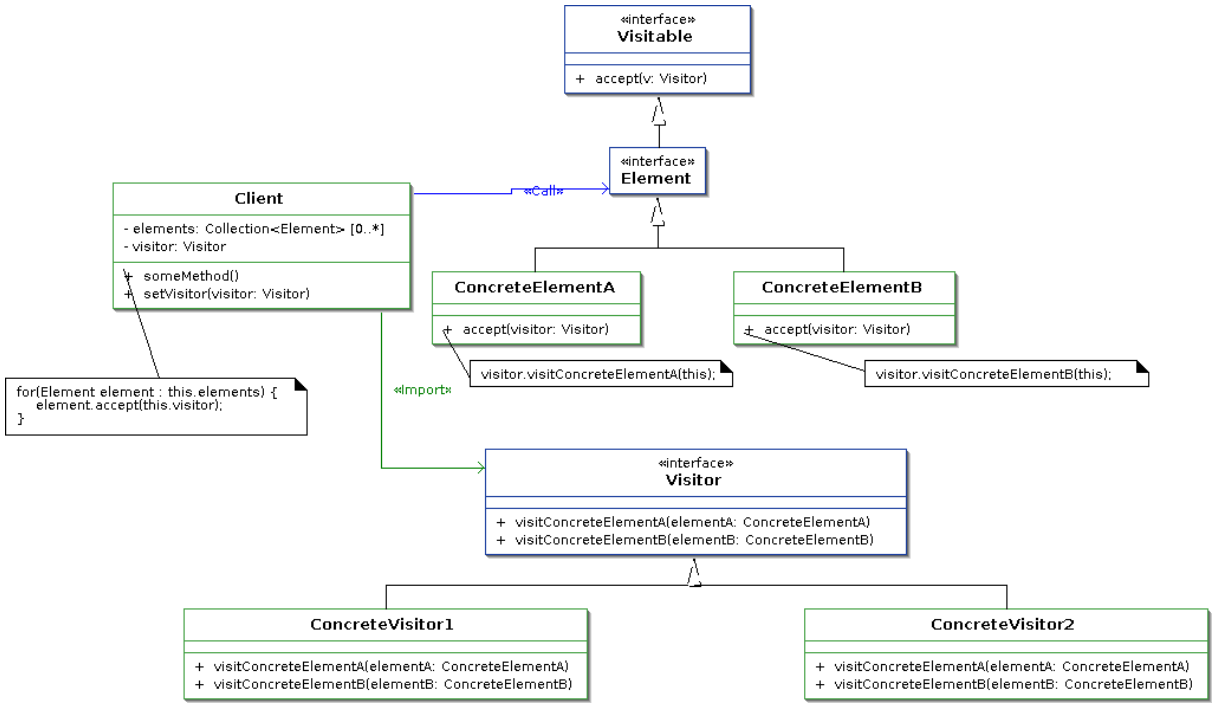
UE Conception Oriente Objet

Design Pattern : 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.

Structure



La classe *Client* représente les utilisateurs du *visiteur* : ses instances a accès à des élements et décident comment les visiter.

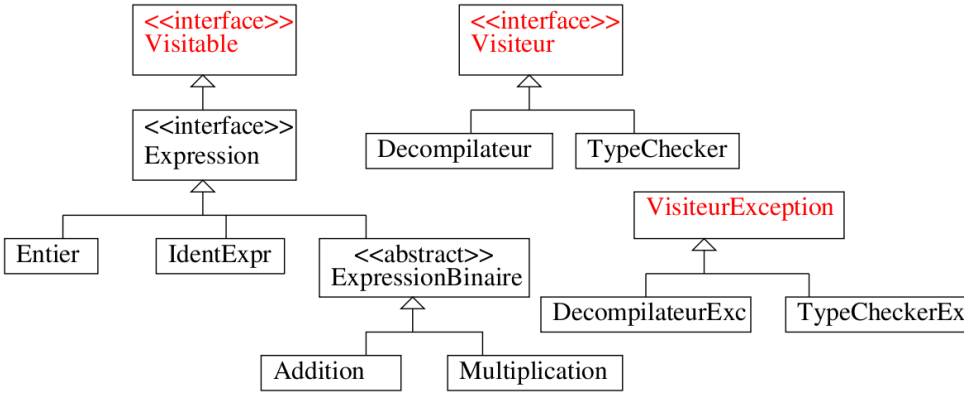
Eléments caractéristiques

- Un ensemble de type d'éléments à exploiter : les *visitables*.
- une méthode d'"*acceptation*" qui est appelée sur les éléments afin qu'ils orientent l'appel vers la méthode du visiteur qui convient pour l'élément en question : inversion du contrôle.

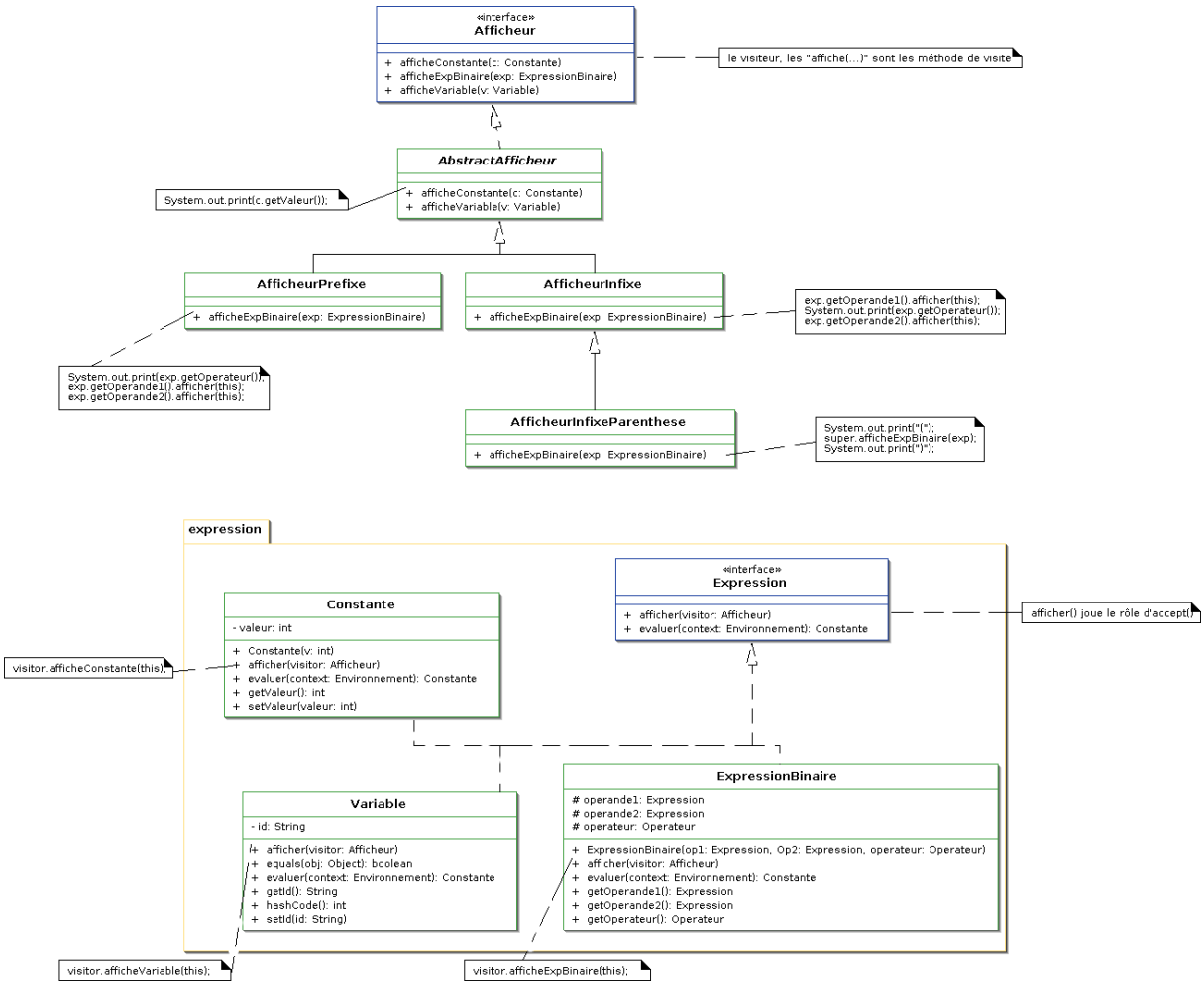
  On a ainsi un même appel pour tous les objets éléments chacun d'entre eux choisissant la méthode de visite appropriée.

Exemples rencontrés

En cours de Compilation : décompilateur et type-checker Dans l'UE Compilation, mise en œuvre du design pattern *visiteur* pour le décompilateur et le type-checker. Voir les notes de cours et TP de l'UE Compilation.



Affichage d'expressions artihmétiques.



Version "réflexive"

On peut à juste titre remarquer que la structure du *Visitor* ne respecte pas le principe ouvert-fermé : l'ajout d'un sous-type concret d'éléments nécessite la modification de l'interface (et donc des classes l'implémentant) avec l'ajout de la méthode de visite associée.
Il est cependant a priori délicat d'appeler simplement `visit` toutes les méthodes de visite à la place des `visitConcreteElementA` et `visitConcreteElementB`, en comptant sur les différences de signatures (types différents des paramètres) pour faire la différenciation des appels de méthodes. On aurait ainsi des méthodes `visit(ConcreteElementA)` ou `visit(ConcreteElementB)`. Le problème intervient notamment si l'on ajoute dans `Visitor` une méthode de visite par défaut dont la signature serait : `visit(Element element)` (et une telle situation est assez courante). Dans ce cas l'appel `visitor.visit(this)` mènerait nécessairement à l'invocation de la méthode `visit(Element)` au détriment complet des méthodes `visit(ConcreteElementA)` et `visit(ConcreteElementB)`.
Ce problème peut être contourné en mettant en place le mécanisme décrit dans les notes de cours sur les design patterns dans la section intitulée "*ReflectiveVisitor*". On a dans ce cas qu'une seule méthode `visit(Object)` et on utilise les aspects réflexifs de java pour trouver et invoquer la "bonne" méthode de `visit`.

```
public interface ReflectiveVisitor {
  public void visit(Object o);
}
public class PrintVisitor implements ReflectiveVisitor {
  public void visit(ConcreteElementA a) { ... }
  public void visit(ConcreteElementB b) { ... }
  public void defaultVisit(Object o) { ... }
  public void visit(Object o) {
    try {
      Method m = getClass().getMethod("visit", new Class[] { o.getClass() });
      m.invoke(this, new Object[] { o });
    }
    catch (NoSuchMethodException e) { this.defaultVisit(o); }
} }
```

(Une version plus "adaptée", au niveau de la recherche des signatures de méthodes, est dans les notes de cours.)