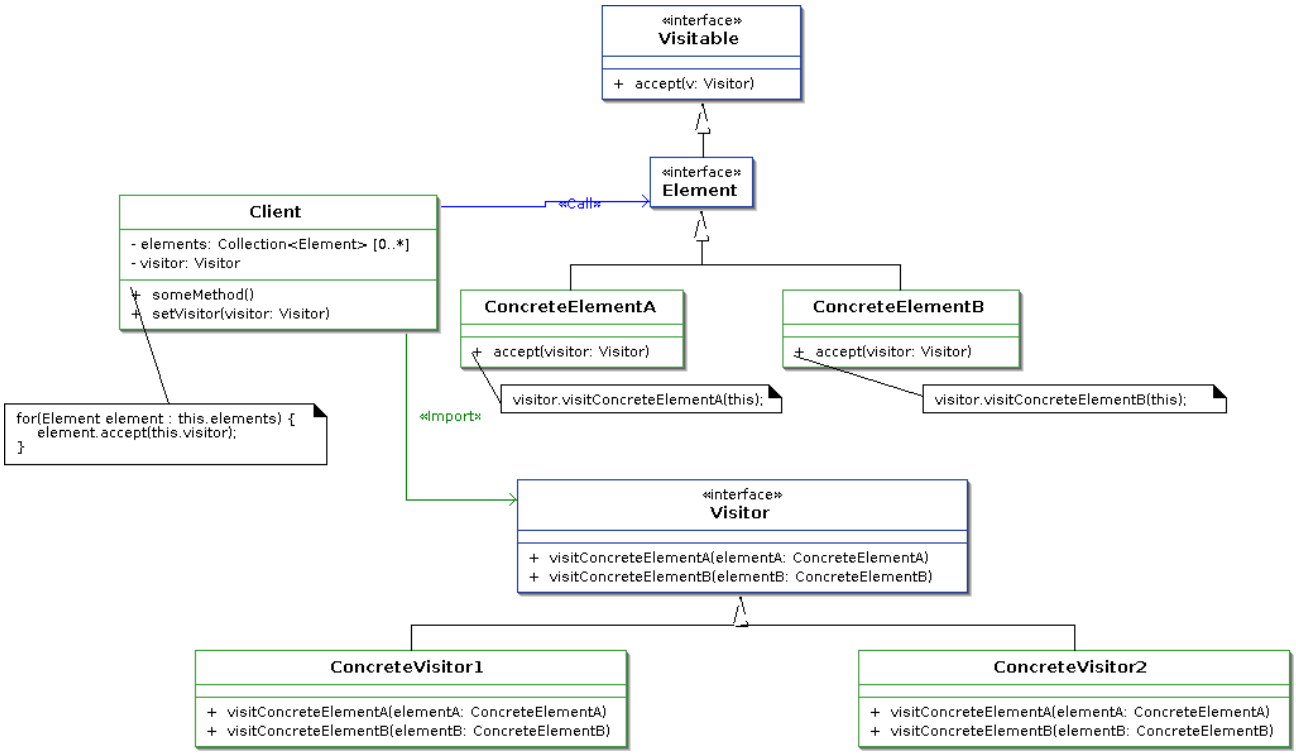
Design Pattern : visitor

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 ont accès à des élements et décident comment les manipuler à l'aide du *visiteur*. Faire varier le visiteur utilisé permet de modifier le traitement appliqué sur les éléments.

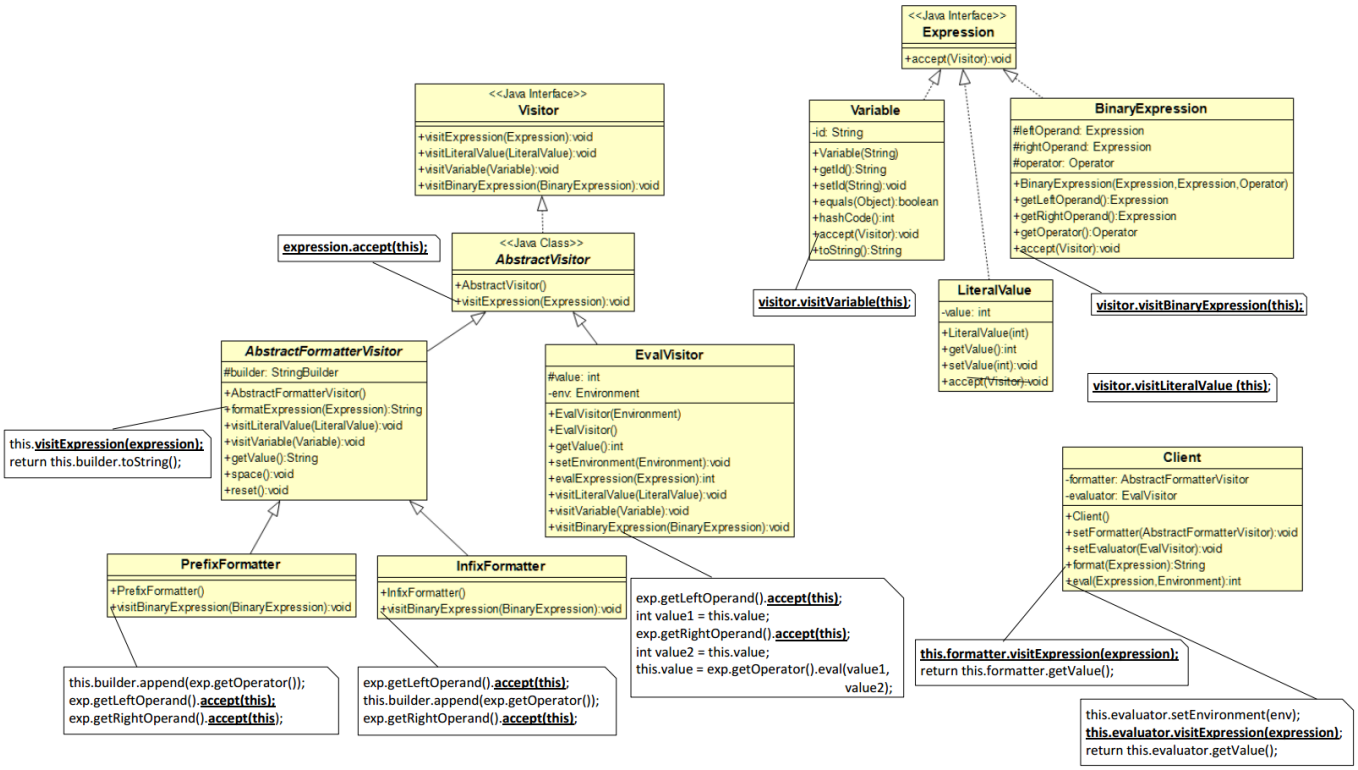
Eléments caractéristiques

- 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

TD – Expressions : affichage et évaluation des expressions. Chaque type d'expressions nécessitent un traitement particulier, or un objet expression est construit à partir de ces différents types qui sont agrégés (on a ici un *composite*) dans cet objet. Il est donc nécessaire de parcourir la structure ainsi constituée dans le cas des expressions binaires, tout en différenciant les comportements sur chacun des éléments de la structure. Il est donc nécessaire de « demander » à chaque élément de la structure comment il doit être manipulé : c'est le rôle de la méthode `accept`.
Deplus différents traitements sont possibles sur les expressions : le formattage et l'évaluation, et on doit pouvoir en envisager d'autres. Il doit donc être possible de faire varier les traitements à appliquer sans avoir à modifier les éléments de la structure. C'est ce que permet l'abstraction des méthodes `visitXxx`.
La classe `Client` peut ainsi manipuler (visiter) les expressions de différentes manières à l'aide des visiteurs adaptés.



On peut noter le *double-dispatch* mis en place avec la méthode `formatExpression()` de `AbstractFormatterVisitor` : cette méthode appelle la méthode `accept()` de l'expression considérée (via `visitExpression()` afin de s'adresser au « bon type » d'expressions, puis la méthode `accept()` de l'expression ainsi ciblée, revient à son tour vers le type `AbstractFormatterVisitor` on appellant la méthode `visitXxx` adaptée. La réalisation du formattage d'une expression par l'`AbstractFormatterVisitor` se fait ainsi par un aller-retour des appels entre ce formatteur et les types d'expression.

Version "réflexive" [1]

On pourrait 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 le double dispatch mentionné précédemment devient indispensable.

Ce problème peut être contourné en mettant en place un mécanisme de recherche de méthodes qui s'appuie sur la réflexivité. On a dans ce cas une seule méthode `visit(Object)` que l'on recherche puis invoque grâce à ces mécanismes d'introspection. Quelque chose comme :

```
public abstract ReflectiveVisitor {
  public abstract void defaultVisit(Object o);
  public final void visit(Object o) {
      try {
        // recherche de la méthode visit qui "va bien" pour o (càd qui prend le type de o en paramètre)
        Method m = this.getClass().getMethod("visit",new Class[] o.getClass() );
        m.invoke(this, new Object[] o );
      } catch (NoSuchMethodException e) {
        // si pas de méthode spécifique pour le type de o on applique le comportement par défaut
        this.defaultVisit(o);
      }
  }
}

public class ConcreteVisitor extends ReflectiveVisitor {
   // dans chaque "concrete visitor" on peut choisir les types d'éléments visités
   public void visit(ConcreteElementA a)  { ... }
   public void visit(ConcreteElementB b) { ... }
   public void defaultVisit(Object o) { ... default behaviour... }
}
```

[1] *Une version plus "adaptée", au niveau de la recherche des signatures de méthodes est possible mais plus longue et complexe à mettre en place pour sa présentation ici. Il faudrait notamment gérer les éventuelles interfaces ou autres cas de polymorphisme sur le type du paramètre de `visit()`. Ce qui est présenté ici donne cependant l'esprit de ce qu'il faudrait faire.*