Visiteurs

Un fois l'**arbre syntaxique** produit, on s'en sert pour les phases suivantes, par exemple le pretty printing, l'analyse de noms et de types, la génération de code (intermédiaire) ou l'interprétation.

On souhaite réaliser ces phases dans des classes séparées. Le concept des **visiteurs** permet d'ajouter de nouvelles fonctionnalités sans modifier la structure d'objets (la classe `Tree` avec ses sous-classes).

Il existe d'autres techniques de génie logiciel, par exemple la décomposition orientée-objet. Le but commun est habituellement la modularisation du code (voir "Design Patterns").

Les **visiteurs** sont utilisés lorsque

- une structure d'objets (la classe `Tree` avec ses sous-classes) contient des classes avec des champs (variables de classe) différents, et les opérations s'effectuent sur ces champs;
- on veut séparer la définition des méthodes de la déclaration des classes;
- la structure d'objets ne change pas, mais il y a beaucoup de méthodes définies là-dessus.

Survol de cette période:

- Fonctionnement
- Une interface dans la classe des arbres
- Comment se servir de l'interface
- Chaînes d'appel
- Exemple: un pretty printer

Fonctionnement

Comment réaliser un pretty printer sur un arbre?

```
abstract class Tree {

   ...

   class Assign extends Tree {
      String id; Tree exp;

      void prettyPrint(int offset) {
         printOffset(offset);
         System.out.print(id);
         System.out.print(" := ");
         exp.prettyPrint(0);
      }
   }

   ...
}
```

Problème:

Une méthode par fonctionnalité et sous-classe. La structure d'objets n'est plus lisible si elle contient toutes les méthodes.

Alternative:

Voir exercice 4. Mais là on a des casts à effectuer.

Fonctionnement (cont.)

Idée: Une seule méthode `apply()` par classe d'arbre qui fait appel à une méthode extérieure réalisant la fonctionnalité.

```
abstract class Tree {

   interface Visitor {
      ...
      void caseAssign(Assign tree);
      ...
   }

   ...

   class Assign extends Tree {
      String id; Tree exp;

      void apply(Visitor v) { v.caseAssign(this); }
   }

   ...
}
```

La classe `Tree` contient une interface `Visitor` qui

- spécifie une méthode pour chaque sous-classe d'arbre (overloading possible);
- est implémentée par chacune des fonctionnalités ⇒ **visiteur concret**.

Chacune des sous-classes contient une méthode `apply(Visitor v)` qui applique la méthode correspondante du visiteur.

Fonctionnement (cont.)

```
class PrettyPrinter implements Tree.Visitor {

   int offset;

   String visit(Tree tree, int offset) {
      PrettyPrinter v = new PrettyPrinter(offset);
      tree.apply(v);
   }

   void caseAssign(Tree.Assign tree) {
            printOffset(offset);
            System.out.print(tree.id);
            System.out.print(" := ");
            visit(tree.exp, offset);
         }
   }

   ...

}
```

Pour visiter un noeud de l'arbre (`visit()`): créer un objet de **visiteur concret** qui appelle le noeud précis de l'arbre (`tree.apply(v)`). Celui-ci accepte et appelle la/les fonction(s) correspondante(s) réalisée(s) dans le visiteur concret (`v.caseAssign(this)`).

Jeu de communications entre deux objets: arbre et visiteur concret.

Appels

visiteur `v`	**communication**	**arbre** `tree`
création de `v`		
	`--- tree.apply(v) --->`	
		arbre (sous-classe `XY`) accepte, choisit le cas correspondant (voir interface)
	`<--- v.caseXY(this) ---`	
exécution de la fonctionnalité		

Le **visiteur concret** est responsable de la fonctionnalité.

L'**arbre** est responsable du choix de la méthode selon les cas spécifiés dans l'interface.

Conséquences

1. Il est facile d'ajouter des opérations.
2. Il est difficile de changer la structure d'objets.
3. Modularisation: groupement des opérations correspondantes, séparation des opérations non-liées.
4. Overloading possible: plus "élégant" mais moins lisible.
5. Moins d'encapsulation: les champs doivent être accessibles.