M1 Master Informatique - Modularité et Réutilisation (UE HAI818I)
TD-TP No 1

Etude des Schémas de conception, de réutilisation, d'évolution - Suite
Discussions et solutions logicielles à propos du parcours de structure arborescentes - *Visitor* versus *Composite*

1 Etude du schéma Visiteur

Le schéma *Visiteur* propose une solution pour le parcours "non anticipé" de structures de données arborescentes, organisée selon le modèle du schéma *Composite* dans le but de les doter de fonctionnalités nouvelles, d'inspection ou de modification.
Une fois le mécanisme de visite installé au sein de la structure de données, alors sans que le code source de cette dernière doive être modifié et sans avoir besoin de le recompiler, il est possible de programmer de nouveaux visiteurs de la structure de données.
Le but du TD est de lire (exercice de lecture en anglais d'un texte technique avec vocabulaire spécifique"GL") et compredre le schéma *visiteur* tel que décrit dans le livre "Design Patterns, Elements of Reusable Software". Après votre lecture nous serons en mesure de discuter les quelques questions suivantes, et d'autres.

1. En quoi les noeuds d'un arbre de syntaxe abstraite (c'est l'exemple utilisé dans le texte) relèvent-ils du schéma de conception "Composite". Donnez le schéma UML de la hiérarchie des classes représentants ces arbres (noeuds et feuilles)
2. Quelles sont les exemples de visiteurs d'arbre de syntaxe proposés dans le texte ?
3. Par une instance de quelle classe un visiteur est-il globalement représenté ?
4. Combien de méthodes trouve-t-on sur une classe définissant un visiteur ?
5. Etudiez particulièrement le code solution de la page 338.
6. Les visiteurs ont-ils accès à la structure interne privée des objets qu'ils visitent ? Quelles sont les différentes alternatives à ce sujet ?
7.

2 Un exemple de données arborescentes

Le but du TP est de programmer une structure de visiteur dans des classes décrivant des données arborescentes (par exemples la simulation des éléments de stockage donnée ici (mais vous pouvez choisir un autre exemple), puis de doter cette structure d'un mécanisme de visite selon le schéma étudié.
Considérons donc l'exemple des éléments de stockage en mémoire secondaire fournis par un système d'exploitation : fichiers, liens symboliques et dossiers (répertoires) :

- Un dossier (*Directory*) peut contenir des éléments de stockages, dont des fichiers (*File*) et des dossiers. La collection des éléments est privée (ou protégée) et il n'y a pas d'accesseur pour l'obtenir. C'est une analogie au fait qu'il n'y a aucun moyen d'obtenir le contenu d'un répertoire Unix ; la commande 'ls' liste les noms mais ne rend pas la collection.
- un fichier possède un contenu que l'on représentera dans la simlulation par une *String*.
- tout élément de stockage est contenu dans un dossier, sauf le dossier racine qui n'a pas de conteneur.
- tout élément de stockage possède un attribut `basicSize` intiquant l'espace de base qu'il occupe en mémoire quand il est vide (0 pour un fichier, 0 pour un lien, 4 pour un dossier).
- tous les éléments de stockage possèdent les méthodes :
  - `int size()` qui rend la taille occupée par l'élément au moment de l'invocation de la méthode. Pour un fichier c'est le nombre de caractères. Pour un dossier, *basicSize* plus la somme des tailles de ses éléments.
  - `String absoluteAddress()` qui donne l'adresse absolue de l'élément. Dans notre simulation, cette méthode rend une string équivalente au résultat de la commande unix *pwd*.
  - `void ls()` avec les mêmes spécifications que la commande Unix (essayez).

— chaque élément de stockage possède des attributs et méthodes spécifiques :
  — pour un fichier ou un lien, `void cat()` qui affiche son contenu à l'écran, et `setContents(String)` qui permet de changer son contenu.
  — pour un répertoire ou pour un fichier `int nbElem()` qui rend respectivement le nombre d'éléments de premier niveau du répertoire ou de caractères du fichier,

Cet exemple est un prétexte, vous pouvez le modifier à votre guise.

3 Appliquer "Visiteur" aux éléments de stockage.

— Programmez les éléments de stockage. 1) Définissez les classes les représentant les éléments de stockage avec toutes les méthodes précedentes. Certains ont déjà fait cet exercice, les autres pourront donc trouver le code d'une solution ICI.
— Doter ces classes d'un mécanisme de visite selon le schéma étudié précédemment.
— Réaliser une classe `RazVisitor` dont une instance permet de remettre à zéro la taille de tous les fichiers du répertoire visité et récursivement de tous les fichiers de tous ses sous-répertoires. Un tel visiteur utilisera la méthode `setContents` de la classe `File`.
— `CountVisitor` : Une des questions que pose le schéma "visiteur" est de savoir comment rendre des résultats. Pour travailler cet exercice, on se propose d'écrire un visiteur capable de donner le nombre de fichiers dans un répertoire dont la taille dépasse 10 caractères. Pour cela, réaliser un visiteur à qui l'on puisse demander en premier lieu de visiter récursivement un répertoire pour faire le comptage et auquel on puisse ensuite envoyer le message `getCount()` qui rende le nombre calculé.
— Une autre question est de savoir avoir un paramètre (la solution est assez proche de celle permettant de rendre un résultat). Réalisez un findVisitor qui implante une fonction `Collection find(String name)` qui rend la collection des adresses absolues des éléments de nom `name` qu'il contient directement ou par transitivité.
— `JavaCleanVisitor` : Réaliser une classe `JavaCleanVisitor` qui permet de visiter un répertoire et ses sous-répertoires pour y détruire tous les fichiers dont le nom est suffixé par ".class".
  Cet exercice pose le problème intéressant du parcours modificateur d'une collection.
— Réalisez un visiteur **countVisitor2** paramétré par une fonction. Ceci évitera d'écrire une nouvelle classe pour chaque requète spécifique (par exemple compter le nombre de fichiers vides ou le nombre de fichier préfixés par "." ou autre). Utiliser des lambdas si le langage que vous utilisez en possède. Une solution en Java antérieur à 1.8, passe par l'utilisation du `package Reflect`.
— Qu'est ce que le "double dispatch" évoqué dans l'article ? A quoi sert-il ?
  Regardez dans Wikipedia.
  Connaissez vous le langage `Common-Lisp`. Il y a là un lien à faire avec les multi-méthodes de Common-Lisp-Object-System.

4 Discussion "Visitor" versus "Composite"

L'évolution des programmes est une question très sensible économiquement. Les refontes complètes de systèmes suite à des transformations successives mal maîtrisées coutent cher.
On peut en fait considérer *Visitor* et *Composite* comme deux solutions alternatives pour la représentation de données arborescentes. Dans une version purement "visiteur" de cette représentation, il n'y aurait plus aucune méthode de parcours qui ne serait pas représentée comme telle. On peut considérer l'alternative qu'ils représentent comme un bon exemple de ce qui est nommé en génie logiciel : "tyrannie de la décomposition dominante" (voyez avec un moteur de recherche). Une fois que l'architecture d'un programme est choisie, certaines choses sont aisées et d'autres deviennent complexes.

1. Quel est le parmi les deux schémas précédent celui permettant le plus facilement (en modifiant le minimum de choses) d'ajouter de nouveaux types de données (un nouveau type d'élément de stockage) et celui permettant le plus facilement d'ajouter de nouvelles fonctionnalités. Trouvez des exemples.
2. Imaginez un programme réalisant une transformation automatisée d'un schéma vers l'autre. Lisez à ce sujet cet article : Transformations between Composite and Visitor implementations in Java (http://www.lirmm.fr/ dony/enseig/RCd/RCStock/CompositeVisitor.pdf).