Programmation par visiteurs et par aspects dynamiques

Carine Courbis — Alexandre Fau — Didier Parigot

INRIA Sophia - Projet OASIS
2004, route des Lucioles - BP 93
06902 Sophia-Antipolis, France

Prénom.Nom@sophia.inria.fr

RÉSUMÉ. *SmartTools est un générateur d'environnements de développement fortement basé sur les technologies objet et les standards du W3C. A partir de spécifications, il produit automatiquement des environnements spécialisés pour des langages de programmation ou des langages métiers. Pour la description de traitements sémantiques ou de transformations, SmartTools s'appuie sur la technologie objet des «visitor patterns». Cette technologie a été étendue avec l'introduction d'une programmation par aspects. Les avantages de cette approche sont essentiellement sa simplicité de mise en œuvre et sa caractéristique dynamique. En effet, il est possible d'ajouter ou de retirer un aspect dynamiquement, sans transformation préalable du source.*

ABSTRACT. *SmartTools is a development environment generator, based on object technologies and on W3C standards. Thanks to a process of automatic generation from specifications, SmartTools makes it possible to quickly develop environments dedicated to programming or domain-specific languages. It uses the visitor pattern technology to describe semantic treatments and transformations. This technology was extended with the insertion of the aspect-oriented programming. The advantages of this approach are mainly its straightforwardness of the implementation and the dynamic characteristic of this programming. Indeed, it is possible to dynamically add or withdraw aspects without any program transformation.*

MOTS-CLÉS : *visitor pattern, programmation par aspect, sémantique*

KEYWORDS: *visitor pattern, aspect-oriented programming, semantics*

1re soumission à *LMO'2002*, le 7th September 2001.

Introduction

Dans le cadre des nouvelles technologies liées au traitement de l'information, le concept de langage est de plus en plus utilisé comme moyen de structuration de l'information. Les efforts de structuration et de formalisation des données pour l'Internet (introduction de formalismes comme les DTDs [XML], Data Type Definitions, ou encore les Schemas [sch]) popularisent la notion de «syntaxe abstraite», élément de base pour les techniques relatives aux langages. Dans le cadre particulier de l'Internet, la qualité logicielle et la rapidité du développement sont des soucis majeurs. Cela motive la réalisation d'un générateur d'environnements de développement basé sur les nouvelles technologies issues du W3C (World Wide Web Consortium). Nous proposons une plate-forme logicielle pour développer des techniques relatives aux langages. Cette plate-forme devra offrir, en particulier, des environnements interactifs avec des interfaces utilisateur de qualité. Les caractéristiques composant logiciel, programmation objets et distribuée, application intranet et l'émergence des technologies XML (eXtensible Markup Language) sont des atouts de diffusion et de nouveaux champs d'application pour ce type de logiciel.

Tout cela constitue les différents objectifs de l'outil SmartTools[1] [ATT 01]. A partir de la définition d'un langage (e.g. une DTD), un concepteur peut très rapidement développer, grâce à SmartTools, un environnement de développement et les outils de traitement associés. Cet article se focalise uniquement sur la présentation des outils et des techniques disponibles dans SmartTools pour la spécification des traitements sémantiques. Comme la technique des «visitor patterns» [GAM 95] a de bonnes propriétés d'extension, elle est proposée dans SmartTools pour décrire ce type de traitement. Elle a été largement automatisée par la génération de code source Java à partir de la définition des structures (Abstract Syntax Tree, AST). De plus, une programmation par aspect a été introduite pour enrichir cette technique des «visitors patterns». Cette nouvelle fonctionnalité ne nécessite aucune transformation de programme. Ainsi, l'ajout d'aspects sur un visiteur peut être totalement dynamique (sans recompilation).

Cet article comporte cinq sections. La première introduit très rapidement notre formalisme de description de syntaxe abstraite. Les deuxième, troisième et quatrième sections présentent des extensions à la technique de programmation par visiteurs : des visiteurs paramétrables éliminant les principaux défauts de cette technique, des visiteurs génériques et un visiteur composé d'un Tree Traversal et d'aspects. Puis, la cinquième section donne plus de détails sur l'implémentation dans le cadre de SmartTools.

1. Syntaxe abstraite

En interne, SmartTools utilise, comme brique de base à tous ses outils, la notion de syntaxe abstraite (AST) fortement typée et étendue (e.g. fils facultatif, liste). Les

1. http ://www-sop.inria.fr/oasis/SmartTools

concepts importants de la définition de cette syntaxe sont les opérateurs et les types. Les opérateurs sont regroupés en des ensembles nommés : les types. Les fils des opérateurs sont typés. La figure 1 montre la définition de notre langage jouet : tiny[2]. Par exemple, l'opérateur *while* est de type *Statement* et possède deux fils : le premier de type *ConditionExp* et le second de type *Statements*. Il existe trois catégories d'opérateur : atomique sans fils i.e une feuille (e.g. *var*), avec des fils de types différents (e.g. *while*) et avec des fils de même type i.e une liste (e.g. *statements*).

```
Formalism of tiny is
Root is %Top;
  Top = program(Decls declarationList, Statements statements);
  Decls = decls(Decl[] declarationList);
  Decl = intDecl(Var variable), booleanDecl(Var variable);
  Statements = statements(Statement[] statementList);
  Statement = affect(Var variable, Exp value),
    while(ConditionExp cond, Statements statements),
    if(ConditionExp cond, Statements statementsThen, Statements statementsElse);
  ConditionOp = equal(ArithmeticExp left, ArithmeticExp right),
     notEqual(ArithmeticExp left, ArithmeticExp right);
  ConditionExp = %ConditionOp, true(), false(), var;
  ArithmeticOp = plus(ArithmeticExp left, ArithmeticExp right),
    minus(ArithmeticExp left, ArithmeticExp right),
    mult(ArithmeticExp left, ArithmeticExp right),
    div(ArithmeticExp left, ArithmeticExp right);
  ArithmeticExp = %ArithmeticOp, int as STRING, var as STRING;
  Exp = %ArithmeticOp, %ConditionOp, var, int, true, false;
  Var = var;
End
```

FIG. 1. *La définition d'AST de notre langage jouet (tiny)*

Edition structurée

A partir de la définition d'AST, SmartTools engendre automatiquement un éditeur structuré spécifique au langage. Pour faciliter l'édition (les copier-coller), il est utile de permettre l'inclusion d'un type dans un autre. Par exemple dans la figure 1, le type *ConditionOp* contenant les opérateurs de condition (dont *equal*) est inclus dans le type *ConditionExp*. Ainsi lors de l'édition, un sous-arbre de type *ConditionExp* peut être remplacé par un sous-arbre de type *ConditionOp*.

2. Programmation par visiteurs

La méthodologie des «visitor patterns» est proposée comme technique de base pour décrire les traitements sémantiques. Si le lecteur souhaite avoir plus de détails et d'explications sur cette méthodologie bien connue, il pourra se référer à [GAM 95, PAL 96, PAL 98].

2. langage utilisé comme fils d'Ariane au cours de cet article

Nous avons étendu la forme classique des visiteurs (appelés par la suite *visiteurs V1*) de trois manières différentes : i) en utilisant le mécanisme de réflexivité avec les possibilités de spécifier les signatures des méthodes visites, le parcours et d'ajouter des aspects (*visiteurs V2* décrits dans cette section), ii) en rendant les visiteurs indépendants du formalisme (*visiteurs V3* décrits en section 3), iii) en décomposant le visiteur en une spécification de parcours (tree traversal) et un ensemble d'actions sémantiques (*visiteur V4* décrit en section 4).

Pour faciliter le développement de visiteurs V2, SmartTools génère automatiquement à partir d'une définition d'AST deux classes de visiteurs : *AbstractVisitor* et *TraversalVisitor*. Le visiteur abstrait déclare toutes les méthodes visites (une par opérateur). Le *TraversalVisitor* hérite de *AbstractVisitor* en implémentant toutes les méthodes visites de façon à effectuer un parcours en profondeur de l'arbre. Pour spécifier une analyse (e.g. vérificateur de type, évaluateur ou compilateur), il suffit d'étendre ce visiteur (*TraversalVisitor*) en redéfinissant les méthodes visites utiles à cette analyse.

2.1. *Signatures des méthodes visites*

Nous offrons la possibilité de préciser les signatures des visites (profils) des visiteurs V2 i.e. de générer des visites avec des types de retour, des noms, et des paramètres de type et de nombre différents. La granularité de cette personnalisation se situe au niveau des types. Ces signatures sont spécifiées dans un langage nommé *xprofile*. La figure 2 présente la spécification des signatures des visites du typechecker de *tiny*. A partir de cette spécification, le système génère automatiquement les visiteurs abstrait et de parcours correctement typés et avec les paramètres voulus. Les figures[3] 3 et 4 montrent le même traitement sémantique (vérification de type) pour l'opérateur *affect* en utilisant respectivement une visite V2 et une visite V1. Le code de la visite profilée est plus lisible car il n'y a pas de forçage de type et les appels récursifs sur les fils sont explicites. Cette possibilité de spécifier les signatures des visites évite donc les défauts de la technique des visiteurs i.e. un code illisible à cause des nombreux forçages de type (casts) en retour des méthodes visites ou sur les arguments (implémentés auparavant comme un *java.lang.Object*).

2.2. *Parcours*

Il est aussi possible de spécifier le parcours souhaité (du point de départ au(x) point(s) d'arrivée) du visiteur avec le langage *xprofile*. Ainsi, uniquement les visites des nœuds appartenant au chemin sont effectuées au lieu de visiter l'arbre complet. Cela permet d'optimiser la vitesse d'exécution des visiteurs sur des arbres conséquents et de réduire surtout la taille du code des visiteurs générés. L'exemple de la figure 5 indique que seules les visites des opérateurs *while* et *affect* et les visites des opérateurs

---

3. Nous avons volontairement mis en taille réduite les parties de code présentant peu d'interêt pour la compréhension.

```
XProfile TypeChecker;
Formalism tiny;
import tiny.visitors.TinyEnv;
Profiles
 Object check(%Top, TinyEnv env);
 Object check(%Decls, TinyEnv env);
 Object check(%Decl, TinyEnv env);
 Object check(%Statements, TinyEnv env);
 Object check(%Statement, TinyEnv env);
 String check(%Exp, TinyEnv env);
 String check(%ArithmeticOp, TinyEnv env);
 String check(%ConditionOp, TinyEnv env);
 String check(%ArithmeticExp, TinyEnv env);
 String check(%ConditionExp, TinyEnv env);
 String check(%Var, TinyEnv env);
Strategy TOPDOWN;
```

FIG. 2. *Fichier de personnalisation du typechecker de tiny - Signatures des visites*

```
1  public Object check(AffectNode node, TinyEnv env) throws VisitorException {
2      String varName = node.getVariableNode().getValue();
3      String typeLeft = env.getType(varName);
4      String typeRight =  check(node.getValueNode(), env); //visit the value node
5
6      if (typeLeft == null)
7          errors.setError(node, "This variable " + varName + " was not declared");
8      else {
9          if (!typeRight.equals(TinyEnv.ERROR) && (!typeLeft.equals(typeRight)))
10             errors.setError(node, "Incompatible types: " + varName + " is a" +
11                     typeLeft.equals(TinyEnv.INT)?"int":"bool") +" variable");
12     }
13     return null;
14 }
```

FIG. 3. *Vérification de type de l'opérateur affect avec une visite V2 profilée*

```
public Object visit(AffectNode node, Object env) throws VisitorException {
    String varName = node.getVariableNode().getValue();
    String typeLeft = ((TinyEnv)env).getType(varName);
    String typeRight = (String)node.getValueNode().accept(this, env);

    idem figure 3 pour le code du if
    return null;
}
```

FIG. 4. *Vérification de type de l'opérateur affect avec une visite V1*

contenus entre la racine et ces opérateurs (i.e au vu de la définition AST de la figure 1, celles de *program*, *statements* et *if*) seront appelées. Pour déterminer quelles visites sont nécessaires en fonction du chemin spécifié, une analyse de graphe de dépendance sur la définition d'AST est effectuée pour générer les visiteurs, abstrait et de parcours, correspondants.

```
Traversal Essai:
   %Top -> while, affect;
```

FIG. 5. *Spécification de parcours (langage xprofile) pour aller de la racine vers les opérateurs* while *et* affect

2.3. *Aspects*

Comme le mécanisme de refléxivité (cf section 5) est utilisé pour exécuter les visiteurs V2, il est possible d'exécuter du code supplémentaire avant ou après l'appel à une méthode visite. Cela permet d'introduire une notion d'aspect [AOP, BOU 01] spécifique à nos visiteurs sans transformation de programme contrairement aux premières versions d'AspectJ [AsJ]. Pour définir un aspect, il suffit de spécifier le code à exécuter avant et après les visites, puis de l'enregistrer sur le visiteur V2 de son choix. La figure 6 contient le code d'un aspect permettant de tracer toutes les visites appelées.

```
package fr.smarttools.debug;
import fr.smarttools.tree.visitorpattern.Aspect;
import fr.smarttools.tree.Type;
public class TraceAspect implements Aspect {
   public void before(Type t, Object[] param) {
      System.out.println ("Start visit on " + param[0].getClass());
   }
   public void after(Type t, Object[] param) {
      System.out.println ("End visit on " + param[0].getClass());
   }
}
```

FIG. 6. *Code d'un aspect traçant les visites appelées (*param *contient le nœud courant ainsi que les objets passés en paramètre des visites)*

Afin d'offrir une meilleure granularité aux aspects, il existe trois sortes d'enregistrement : sur toutes les visites (méthode *addAspect* de la classe *Visitor*), sur seulement les visites des nœuds appartenant à un certain type (*addAspectOnType*) et sur seulement les visites des nœuds d'un certain opérateur (*addAspectOnOperator*).

Plusieurs aspects différents peuvent être enregistrés sur un même visiteur. Ils seront alors exécutés en série (selon l'ordre d'enregistrement). Ce branchement (comme le débranchement) peut se faire dynamiquement et à tout moment pendant l'exécution d'un visiteur. Le comportement d'un visiteur peut donc être modifié dynamiquement par ajout ou retrait d'aspects. Par exemple, le code d'un mode de «debug» graphique d'exécution pas-à-pas pour les visiteurs a été spécifié comme un aspect indépendamment du code des visiteurs. Pour n'importe quel visiteur spécifié dans SmartTools, ce mode debug peut être ajouté statiquement ou dynamiquement.

3. Visiteurs génériques

A partir des visiteurs V1, on a introduit le concept de visiteurs génériques[4] (visiteurs V3) pour permettre la factorisation de comportements identiques applicables aux différentes catégories d'opérateurs (feuille, d'arité fixe ou liste). De tels visiteurs sont dits génériques car ils sont indépendants des formalismes et donc peuvent s'exécuter sur n'importe quel arbre de syntaxe abstraite. Ils se composent de seulement trois visites (sur *AtomicNode*, *FixedNode* et *ListeNode*) implémentées par défaut pour parcourir l'arbre dans la classe *GenericVisitor*, la sur-classe de tous les visiteurs génériques. Lors de l'exécution de ces visites, quatre méthodes particulières sont appelées : *preVisit* (appelée avant de commencer une visite sur un nœud non atomique), *postVisit* (après), *doVisit* (entre les visites des fils du nœud) et *leafVisit* (pour les feuilles). Ce sont ces dernières méthodes qui sont destinées à contenir le code utile (la sémantique) comme le montre le visiteur de la figure 7 qui compte le nombre de nœuds et de feuilles pour n'importe quel arbre. Contrairement aux visiteurs V2, ces visiteurs

```
package fr.smarttools.visitors;
import fr.smarttools.tree.visitorpattern.VisitorException;
import fr.smarttools.tree.UntypedNode;
public class GenericCountNodesAndLeaves extends GenericVisitor {
   private int nodesNb = 0,leavesNb = 0;
   public GenericCountNodesAndLeaves(Object obj) {super((fr.smarttools.comm.MsgSender)obj);}
   public Object leafVisit(UntypedNode node, Object param, Object value) throws
                                                            VisitorException {
       leavesNb++;return null;
   }
   public Object preVisit(UntypedNode node, Object param)throws VisitorException{
       nodesNb++;
       return null;
   }
   public int getLeavesNumber() {return leavesNb;}
   public int getNodesNumber() {return nodesNb;}
}
```

FIG. 7. *Code du visiteur générique comptabilisant le nombre de feuilles ou de nœuds pour n'importe quel arbre*

ne sont pas paramètrables i.e. leurs parcours sont fixes (parcours en profondeur des arbres) et aucun aspect ne peut être enregistré.

4. Visiteur à aspects et à Tree Traversal

Avec le concept de la programmation par aspects, il est possible de séparer le parcours de l'arbre (appels explicites des méthodes visites) des traitements sémantiques (actions sémantiques). Supposons que le code de la visite pour l'opérateur *affect(Var variable, Exp value)* soit de la forme :

4. L'implémentation de ces visiteurs n'utilise pas le mécanisme de reflexivité introduit pour les visiteurs V2.

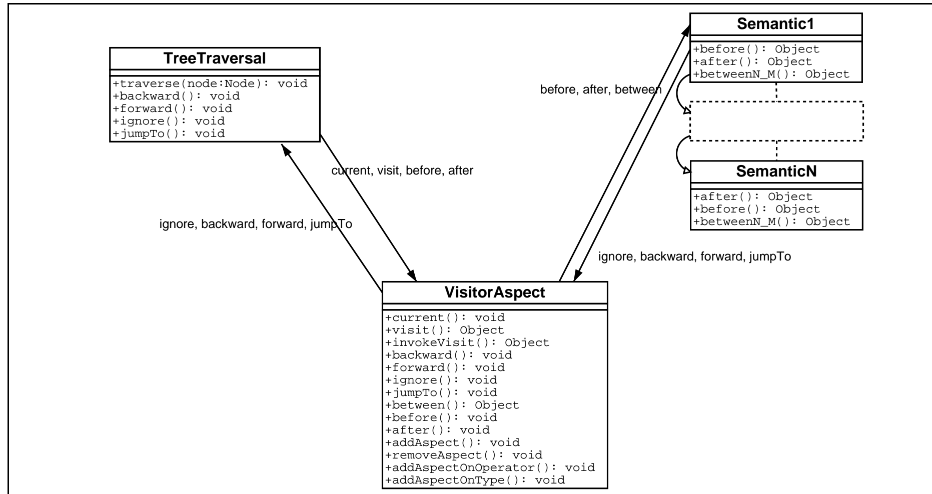

FIG. 8. *Structure du visiteur V4*

```
visit(AffectNode node ...) {
   codeBefore
   visite du premier fils
   codeBetween1_2
   visite du deuxième fils
   codeAfter
}
```

On peut observer que la partie sémantique de la méthode (i.e tout sauf les appels récursifs) est décomposée en N fils + 1 morceaux de code. Ces N+1 morceaux peuvent être traités comme des aspects avec des nouveaux points d'ancrages i.e. avant, entre et après les appels des méthodes visites des fils. Nous avons défini un nouveau visiteur (visiteur V4) qui prend en argument un parcours d'arbre (un Tree Traversal) et un ou plusieurs aspects (i.e. les actions sémantiques) comme le montre la figure 8. Ce visiteur permet d'appeler ces aspects sur ces nouveaux points d'ancrage. Ainsi, ces aspects devront contenir pour chaque opérateur, en plus des méthodes classiques *before* et *after*, les méthodes *betweeni_i+1* (code à exécuter entre les fils i et i+1). Ce nouveau visiteur s'utilise comme les visiteurs V2 (e.g. il est possible de brancher un ou plusieurs aspects de la section 2.3). La figure 9 présente, en utilisant cette nouvelle forme d'aspect, la sémantique associée à l'opérateur *affect* pour la vérification de type. Il n'y a plus d'appel récursif contrairement à l'autre forme des visiteurs (cf figure 3 ligne 4) mais il est nécessaire d'utiliser des piles (cf figure 9 lignes 5 et 6) pour transmettre les résultats des visites des fils.

A cause de la séparation de la sémantique et du parcours, il n'est plus possible d'influencer le parcours par la sémantique (avec le visiteur V4) et donc de décrire une sémantique dynamique (comme par exemple un évaluateur ou un interprèteur). Pour cela, un objet spécifique de parcours dynamique d'arbres a été implanté (comparable à

```
1 public void before(AffectNode node, Object param) {}
2 public void between1_2(AffectNode node, Object param) {}
3 public void after(AffectNode node, Object param) {
4    String varName = node.getVariableNode().getValue();
5    String typeRight = (String)typeStack.pop();
6    String typeLeft = (String)typeStack.pop();
7
8    idem figure 3 pour le code du if
9 }
```

FIG. 9. *Sémantique associée à la vérification de l'opérateur affect*

la notion de *TreeWalker* de DOM, Document Object Model [DOM]) avec des instructions particulières de modification de parcours. Ces instructions permettent d'ignorer un des fils du nœud (méthode *ignore*) ou de préciser le prochain fils à visiter (*jumpTo*). Avec cette approche, un évaluateur de notre petit langage a été complètement spécifié. Les figures 10 et 11 montrent les codes de cet évaluateur pour l'opérateur *while* respectivement avec un visiteur V2 (cf section 2) et la nouvelle forme à base d'aspect (V4).

```
public Object eval(WhileNode node, TinyEnv env) throws VisitorException {
   Boolean cond = eval(node.getCondNode(),env);
   if (cond.booleanValue()) {//while true
      eval(node.getStatementsNode(), env); //execute the statements
      eval(node, env); //redo this while tree evaluation -> recursion
   }
   return null;
}
```

FIG. 10. *Evaluation de l'opérateur while (visiteur V2)*

```
public void before(WhileNode node, Object param) {}
public void between1_2(WhileNode node, Object param) {
   if (((Boolean)conditionStack.peek()).booleanValue() == false)
      ignore(node.getStatementsNode());
}
public void after(WhileNode node, Object param) {
   if (((Boolean)conditionStack.pop()).booleanValue() == true)
      jumpTo(node.getCondNode());
}
```

FIG. 11. *Sémantique associée à l'évaluation de l'opérateur while (visiteur V4)*

Composition d'aspect

Par exemple, le typechecker de *tiny* a pu être enrichi d'une vérification d'initialisation de variable (cf figure 12 pour l'opérateur *affect*) uniquement en composant

les deux aspects (cf figure 13). L'interêt majeur de ce style de programmation est de permettre l'enrichissement des analyses (sans les modifier) simplement par l'ajout de nouveaux aspects (sémantiques). Ainsi, les analyses sont plus modulaires et réutilisables. Mais, ce style de programmation (visiteur V4) est plus complexe à écrire à cause de la séparation de la sémantique et du parcours (cf figures 9 et 3 ou figures 11 et 10). Actuellement, nous étudions des mécanismes pour résoudre les problèmes liés au partage de données entre sémantiques et au parcours commun (e.g que faire si une sémantique veut boucler sur un nœud et pas les autres ?) et pour faciliter l'écriture des aspects en masquant la gestion des piles.

```
public void before(AffectNode node, Object param) {unplugVariableCheck = true;}
public void visit1(AffectNode node, Object param) {unplugVariableCheck = false;}
public void after(AffectNode node, Object param) {
   env.setInitialized(node.getVariableNode().getValue());
}
```

FIG. 12. *Méthodes de l'aspect de vérification d'initialisation pour l'opérateur* affect *(V4)*

```
TypeCheckerVisitor typeCheck = new TypeCheckerVisitor();
TinyEnv env = typeCheck.getEnv();
InitVarCheckerVisitor initVarCheck = new InitVarCheckerVisitor(env);
new Visitor(new LeftToRightTreeTraversal(),
            new Semantics[]{typeCheck, initVarCheck}).start(tree, null);
```

FIG. 13. *Assemblage de deux aspects et du parcours de gauche à droite pour créer un vérificateur de type enrichi*

5. Implémentation

Implémentation au-dessus de l'API DOM

Pour l'implémentation, nous souhaitons utiliser le plus possible les composants logiciels existants issus des standards du W3C, comme par exemple l'API DOM de manipulation d'arbres XML. Mais, cette API ne considère que des structures d'arbre non fortement typées. Pour pouvoir manipuler des arbres fortement typés, elle a donc été étendue et complétée. L'avantage d'avoir un arbre typé est que la cohérence de l'arbre est garantie à sa construction par le typechecker de Java. Par opérateur, SmartTools génère automatiquement une classe et une interface (la figure 14 montre l'interface générée pour l'opérateur *affect*), et une interface par type. Ces classes contiennent les méthodes d'accès (e.g. *getValueNode*) et de modification (e.g. *setValueNode*) des fils (et si l'utilisateur le souhaite, la méthode *accept* utile pour les *visiteurs V1*).

```
package tiny.ast;
public interface AffectNode extends EVERYType, StatementType {
   public tiny.ast.VarType getVariableNode();
   public void setVariableNode(tiny.ast.VarType tree);
   public tiny.ast.ExpType getValueNode();
   public void setValueNode(tiny.ast.ExpType tree);
   public Object accept(tiny.visitors.Traversal vis, Object param)
         throws fr.smarttools.tree.visitorpattern.VisitorException ;
}
```

FIG. 14. *Interface de l'opérateur affect (la méthode accept est optionnelle)*

Visiteurs paramétrables (V2)

Dans SmartTools, pour implémenter la technique des *visiteurs V2*, on utilise le mécanisme de réflexivité et non le mécanisme d'indirection par l'appel d'une méthode particulière, *accept*, définie expressément pour chaque opérateur. En effet, l'introduction de la notion de signature des visiteurs interdit l'utilisation de cette solution car il faudrait avoir une méthode *accept* par signature. Plus précisément, pour tout appel récursif, on force l'appel à une méthode générique (*invokeVisit*) qui va appeler la «bonne» méthode par réflexivité. Avec cette approche, il a été facile d'introduire la possibilité d'exécuter du code supplémentaire (aspects décrits en 2.3) avant et après les visites.

L'utilisation de la réflexivité est coûteuse en temps d'exécution. Pour accélérer les branchements, une table d'indirection est produite statiquement lors de la génération du visiteur abstrait. Cette table contient pour tout couple *(opérateur, type)* la référence Java de l'objet *java.lang.reflect.Method* à invoquer. Cette table facilite aussi la gestion des noms des méthodes et le passage des arguments lors des invocations dans notre méthode générique *invokeVisit*.

Cette solution pour rechercher dynamiquement la «meilleure» méthode à appliquer est une simplification de l'approche des multi-méthodes. Pour comparer, nous avons utilisé une implémentation des multi-méthodes pour Java [FOR 00] et nous obtenons des temps d'exécution équivalents. Mais notre approche reste beaucoup plus simple dans sa réalisation. De plus, grâce à cette table d'indirection, de futures possibilités de paramétrage des appels pourront être très facilement mises en œuvre.

Visiteur composé d'un Tree Traversal et d'aspects (V4)

Le *visiteur V4* a lui aussi une méthode générique qui gère :

– le prochain nœud à visiter en fonction de la position courante, du parcours choisi et des instructions de modification de parcours contenues dans les sémantiques,

– la recherche de la méthode à appeler,

– et enfin l'invocation des aspects décrits en 2.3 sur ces visites.

En résumé, les visiteurs V1 ont comme seul avantage d'être efficaces (pas d'utilisation de la reflexivité). Par contre, les visiteurs V2 permettent d'avoir des méthodes visites avec des signatures explicites, un parcours spécifique et des aspects. Ainsi, une analyse sémantique écrite en visiteur V2 est plus lisible (peu de casts), peut être complétée dynamiquement par des aspects et seules les branches utiles à l'analyse sont visitées. Une analyse écrite en visiteur V4 est plus complexe à écrire mais peut être réutilisée par composition pour enrichir une autre analyse. Une analyse en V3 est moins spécialisée et a un parcours fixe mais elle est indépendante du formalisme et peut donc être utilisée sur n'importe quel arbre de syntaxe abstraite.

6. Conclusion

Dans cet article, nous avons présenté une implémentation particulière des visiteurs basée sur la réflexivité qui est un prolongement naturel des travaux sur les visiteurs de Palsberg et al [PAL 98] et sur les multi-méthodes [MIL 99, FOR 00]. On montre que dans notre cadre (structure de données arborescente) la génération d'une table d'indirection rend le mécanisme de réflexivité raisonnablement efficace. De plus, nous avons introduit les notions de signature des visites pour décrire plus lisiblement des visiteurs. Notre notion des visiteurs génériques et notre spécification de parcours de visites sont proches d'une programmation adaptative [PAL 96, KIC 97, LIE 97] qui a comme but de rendre indépendant le plus possible le traitement (l'algorithme) des structures de données.

Dans ce cadre, il a été facile d'introduire une notion de programmation par aspects. Il est important de noter que cette programmation par aspects ne nécessite pas de transformation de programme et permet donc une gestion dynamique. Cette approche se différencie essentiellement par sa simplicité de mise en œuvre en comparaison avec des travaux plus généraux comme AspectJ [AsJ]. Nous avons commencé à traiter le problème de la séparation vis-à-vis du parcours d'arbre et de la composition d'aspects. Nous sommes en train de définir un mécanisme de contrôle pour la composition d'aspects (échange d'informations).

Tout cela a été implémenté et est déjà fortement utilisé par notre outil SmartTools qui contient bien d'autres points [ATT] que nous n'avons pas décrits dans cet article. En effet, ses interfaces graphiques, son architecture modulaire, ses passerelles entre divers formalismes (DTD, Schema) et enfin sa forte utilisation des standards du W3C en font un outil de développement cohérent et complet. En conclusion, nous avons décrit dans ce papier une approche de la programmation par aspect couplée aux visiteurs, qui a le mérite d'être simple et surtout d'être intégrée dans un outil de développement complet et ouvert aux nouvelles technologies XML. Dans le cadre particulier des applications liées au Web, il est important de proposer un style de programmation ne requérant pas de connaissances théoriques approfondies dans le domaine des langages de programmation et étant offert dans un environnement interactif convivial.

Remerciements

Nous remercions vivement Rémy Forax, Gilles Roussel et Etienne Duris de l'université de Marne-la-Vallée pour leurs avis et leurs travaux sur les multi-méthodes. Nous remercions aussi tous les autres membres de notre équipe pour leur travail : Isabelle Attali, Pascal Degenne, Joël Fillon, Christophe Held et Claude Pasquier. Ce projet est partiellement financé par Bull CP8, Microsoft Research et l'INRIA.

7. Bibliographie

[AOP] « Aspect-Oriented Programming », http ://www.parc.xerox.com/csl/projects/aop/.

[AsJ] « AspectJ-Oriented Programming (AOP) for Java », http ://www.aspectj.org.

[ATT] ATTALI I., COURBIS C., DEGENNE P., FAU A., FILLON J., HELD C., PARIGOT D., PASQUIER C., « Aspect and XML-oriented Semantic Framework Generator : SmartTools », en soumission à ICSE'2002.

[ATT 01] ATTALI I., COURBIS C., DEGENNE P., FAU A., PARIGOT D., « SmartTools : a Generator of Interactive Environments Tools », *Compiler Construction CC'2001*, vol. 2027 de *LNCS*, Genova, Italy, April 2001, Springer-Verlag, Tool demonstration.

[BOU 01] BOURAQADI-SAÂDANI N. M. N., LEDOUX T., « Le point sur la programmation par aspects », *Technique et Sciences Informatiques*, vol. 20, page 505 à 528, Hermés, 2001.

[DOM] « http ://www.w3.org/DOM/ ».

[FOR 00] FORAX R., DURIS E., ROUSSEL G., « Java Multi-Method Framework », *TOOLS'00*, novembre 2000.

[GAM 95] GAMMA E., HELM R., JOHNSON R., VLISSIDES J., *Design Patterns*, Addison Wesley, Reading, MA, 1995.

[KIC 97] KICZALES G., LAMPING J., MENHDHEKAR A., MAEDA C., LOPES C., LOINGTIER J.-M., IRWIN J., « Aspect-Oriented Programming », AKŞIT M., MATSUOKA S., Eds., *ECOOP '97, Jyväskylä, Finland*, vol. 1241 de *LNCS*, p. 220–242, Springer-Verlag, juin 1997.

[LIE 97] LIEBERHERR K. J., ORLEANS D., « Preventive Program Maintenance in Demeter/Java », *ICSE*, ACM Press, mai 1997, p. 604–605.

[MIL 99] MILLSTEIN T., CHAMBERS C., « Modular Statically Typed Multimethods », GUERRAOUI R., Ed., *Proceedings ECOOP'99*, LCNS 1628, Lisbon, Portugal, juin 1999, Springer-Verlag, p. 279–303.

[PAL 96] PALSBERG J., PATT-SHAMIR B., LIEBERHERR K., « A New Approach to Compiling Adaptive Programs », NIELSON H. R., Ed., *European Symposium on Programming*, Linkoping, Sweden, 1996, Springer Verlag, p. 280–295.

[PAL 98] PALSBERG J., JAY C. B., « The Essence of the Visitor Pattern », *COMPSAC'98*, Vienna, Austria, août 1998.

[sch] « XML Schema, W3C Recommendation », http ://www.w3.org/TR/xmlschema-0/.

[XML] « Extensible Markup Language (XML) 1.0 (Second Edition) », http ://www.w3.org/TR/2000/REC-xml-20001006.